ALIBABA CLOUD

阿里云

表格存储Tablestore

SDK 参考

文档版本：20220221

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	**危险** 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	**警告** 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	**注意** 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	**说明** 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.SDK概览

介绍表格存储包含的主流语言SDK。

使用表格存储SDK之前，您需要：

- 了解并开通阿里云表格存储服务。
- 创建AccessKey。

表格存储支持如下主流语言的SDK包。

语言	参考文档
Java	Java SDK参考
Python	Python SDK参考
PHP	PHP SDK参考
Go	Go SDK参考
.NET	.NET SDK参考
NodeJS	NodeJS SDK参考

2.SDK功能支持列表

通过本文您可以了解表格存储各功能支持的主流语言SDK。

各功能支持的主流语言SDK请参见下表。

> **说明** “✓”表示支持，“×”表示不支持。

功能	Java	Go	Python	Node.js	.NET	PHP
Timeline	✓	✓	×	×	×	×
Timestream	✓	×	×	×	×	×
Grid	✓	×	×	×	×	×
表级别操作	✓	✓	✓	✓	✓	✓
多行数据操作	✓	✓	✓	✓	✓	✓
单行数据操作	✓	✓	✓	✓	✓	✓
通道服务	✓	✓	×	×	×	×
SQL查询	✓	×	×	×	×	×
数据湖投递	✓	✓	×	×	×	×
时序模型	✓	✓	×	×	×	×
全局二级索引	✓	✓	✓	✓	✓	✓
本地二级索引	✓	✓	✓	✓	×	×
多元索引：基础功能	✓	✓	✓	✓	✓	✓
多元索引：统计聚合	✓	✓	✓	×	×	×
主键列自增	✓	✓	✓	✓	✓	✓
条件更新	✓	✓	✓	✓	✓	✓
局部事务	✓	✓	✓	✓	×	✓
原子计数器	✓	✓	✓	✓	✓	×
过滤器	✓	✓	✓	✓	✓	✓

3.Java SDK

3.1. 前言

通过本文您可以了解表格存储Java SDK的使用前提条件、版本兼容性和历史迭代版本。本文适用于Java SDK 5.0.0以上版本。

前提条件

- 已开通表格存储服务。
- 已创建Access Key。具体操作，请参见获取AccessKey。
- 4.0.0以上版本SDK支持数据多版本和生命周期。已了解数据多版本和生命周期，更多信息，请参见数据版本和生命周期。

版本兼容性

当前最新版本：5.13.0

- 对4.x.x系列SDK：兼容
- 对2.x.x系列SDK：不兼容

历史迭代版本

历史迭代版本的更多信息，请参见Java SDK历史迭代版本。

3.2. 安装

本文介绍如何安装表格存储Java SDK。

环境准备

安装表格存储Java SDK需使用JDK 6及以上版本。

安装方式

您可以通过以下两种方式安装表格存储Java SDK：

- 在Maven项目中加入依赖项

  在Maven工程中使用表格存储Java SDK，只需在pom.xml中加入相应依赖即可。以5.11.0版本为例，在<dependencies>内加入如下内容：

  > **说明** 关于Java SDK历史迭代版本的更多信息，请参见Java SDK历史迭代版本。

  ```
  <dependency>
      <groupId>com.aliyun.openservices</groupId>
      <artifactId>tablestore</artifactId>
      <version>5.11.0</version>
  </dependency>
  ```

- Eclipse中导入JAR包

  以5.11.0版本为例，详细步骤如下：

i. 下载Java SDK开发包。
ii. 解压该开发包。
iii. 解压后将文件夹中的文件tablestore-<versionId>.jar以及lib文件夹下的所有文件拷贝到您的项目中。

其中<versionId>表示表格存储Java SDK的版本，例如5.11.0版本的文件名称为tablestore-5.11.0.jar。

iv. 在Eclipse中选择您的工程，右键选择**Properties > Java Build Path > Add JARs**。
v. 选中您在第3步拷贝的所有JAR文件。

示例程序

表格存储Java SDK提供丰富的示例程序，方便参考或直接使用。您可以解压下载好的SDK包，在examples文件夹中查看示例程序。

3.3. 初始化

OTSClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、读写单行数据、读写多行数据等。使用Java SDK发起表格存储的请求，您需要初始化一个OTSClient实例，并根据需要修改ClientConfiguration的默认配置项。如果要使用时序模型，您需要初始化TimeseriesClient。

确定Endpoint

Endpoint是阿里云表格存储服务各个实例的域名地址，目前支持下列形式。

示例	解释
http://sun.cn-hangzhou.ots.aliyuncs.com	HTTP协议，公网网络访问杭州区域的sun实例。
https://sun.cn-hangzhou.ots.aliyuncs.com	HTTPS协议，公网网络访问杭州区域的sun实例。

> **注意** 除了公网可以访问外，也支持私网地址。更多请参见服务地址。

请按照如下步骤获取实例的Endpoint：

1. 登录表格存储管理控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

配置密钥

要接入阿里云的表格存储服务，您需要拥有一个有效的访问密钥进行签名认证。目前支持下面三种方式：

- 阿里云账号的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 在阿里云官网注册阿里云账号。
  ii. 创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。
- 被授予访问表格存储权限RAM用户的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 使用阿里云账号前往访问控制RAM，创建一个新的RAM用户或者使用已经存在的RAM用户。
  ii. 使用阿里云账号授予RAM用户访问表格存储的权限。
  iii. RAM用户被授权后，即可使用自己的AccessKey ID和AccessKey Secret访问。
- 从STS获取的临时访问凭证。获取步骤如下：

i. 应用的服务器通过访问RAM/STS服务，获取一个临时的AccessKey ID、AccessKey Secret和SecurityToken发送给使用方。

ii. 使用方使用上述临时密钥访问表格存储服务。

初始化Client

用户使用表格存储的SDK时，必须首先构造一个Client，通过调用该Client的接口来访问表格存储服务，Client的接口与表格存储提供的RestfulAPI是一致的。

表格存储的SDK提供了SyncClient和AsyncClient两种Client，分别对应同步接口和异步接口。同步接口调用完毕后请求即执行完成，使用方便，用户可以先使用同步接口了解表格存储的各种功能。异步接口相比同步接口更加灵活，如果对性能有一定需求，可以在使用异步接口和使用多线程之间做一些取舍。

> **说明** 不管是SyncClient还是AsyncClient，都是线程安全的，且内部会自动管理线程和管理连接资源。不需要为每个线程创建一个Client，也不需要为每个请求创建一个Client，全局创建一个Client即可。

获取到AccessKey ID和AccessKey Secret后，您可以按照如下示例代码初始化Client。

- 使用默认配置创建SyncClient。

```
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
SyncClient client = new SyncClient(endPoint, accessKeyId, accessKeySecret, instanceName);
```

- 使用自定义配置创建SyncClient。

```
// ClientConfiguration提供了很多配置项，以下只列举部分。
ClientConfiguration clientConfiguration = new ClientConfiguration();
// 设置建立连接的超时时间。单位为毫秒。
clientConfiguration.setConnectionTimeoutInMillisecond(5000);
// 设置socket超时时间。单位为毫秒。
clientConfiguration.setSocketTimeoutInMillisecond(5000);
// 设置重试策略。如果不设置，则采用默认的重试策略。
clientConfiguration.setRetryStrategy(new AlwaysRetryStrategy());
SyncClient client = new SyncClient(endPoint, accessId, accessKey, instanceName, clientCon
figuration);
```

初始化TimeseriesClient

如果要使用时序模型，请初始化TimeseriesClient，时序模型需要初始化单独的Client。

表格存储的SDK提供了TimeseriesClient和AsyncTimeseriesClient两种TimeseriesClient，分别对应同步接口和异步接口。

获取到AccessKey ID和AccessKey Secret后，您可以按照如下示例代码初始化TimeseriesClient。

- 使用默认配置创建TimeseriesClient。

```
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
TimeseriesClient client = new TimeseriesClient(endPoint, accessKeyId, accessKeySecret, in
stanceName);
```

- 使用自定义配置创建TimeseriesClient。

```
// ClientConfiguration提供了很多配置项，以下只列举部分。
ClientConfiguration clientConfiguration = new ClientConfiguration();
// 设置建立连接的超时时间。单位为毫秒。
clientConfiguration.setConnectionTimeoutInMillisecond(5000);
// 设置socket超时时间。单位为毫秒。
 clientConfiguration.setSocketTimeoutInMillisecond(5000);
// 设置重试策略。如果不设置，则采用默认的重试策略。
clientConfiguration.setRetryStrategy(new AlwaysRetryStrategy());
TimeseriesClient client = new TimeseriesClient(endPoint, accessId, accessKey, instanceNam
e, clientConfiguration);
```

HTTPS

升级到java 7后即可。

多线程

- 支持多线程。
- 使用多线程时，建议共用一个OTSClient对象。

3.4. 表

3.4.1. 概述

表格存储的Java SDK提供了多种表级别的操作接口。

接口	说明
创建表	创建数据表或者创建带全局二级索引的数据表。
更新表	更新数据表的配置信息。
查询表描述信息	查询表的配置信息。
列出表名称	查看一个实例下的所有表名称。
指定大小计算分片	将表中数据逻辑上划分成接近指定大小的若干分片。
删除表	删除一个指定表。
创建全局二级索引	为数据表创建索引表。创建索引表后，可以读取索引表中的数据或者删除索引表。

3.4.2. 创建数据表

使用CreateTable接口创建数据表时，需要指定数据表的结构信息和配置信息，高性能实例中的数据表还可以根据需要设置预留读/写吞吐量。创建数据表的同时支持创建一个或者多个索引表。

> 说明
> - 创建数据表后需要几秒钟进行加载，在此期间对该数据表的读/写数据操作均会失败。请等待数据表加载完毕后再进行数据操作。
> - 创建数据表时必须指定数据表的主键。主键包含1个~4个主键列，每一个主键列都有名称和类型。

前提条件

- 已通过控制台创建实例。具体操作，请参见创建实例。
- 已初始化Client。具体操作，请参见初始化。

参数

参数	说明
tableMeta	数据表的结构信息，包括如下内容： • tableName：数据表名称。 • primaryKey：数据表的主键定义。更多信息，请参见主键和属性。 说明 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。 ◦ 表格存储可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体请参见表格存储数据模型和查询操作。 ◦ 第一列主键作为分区键。分区键相同的数据会存放在同一个分区内，所以相同分区键下最好不要超过10 GB以上数据，否则会导致单分区过大，无法分裂。另外，数据的读/写访问最好在不同的分区键上均匀分布，有利于负载均衡。 • definedColumns：预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。

参数	说明
tableOptions	数据表的配置信息。更多信息，请参见数据版本和生命周期。 配置信息包括如下内容： • timeToLive：数据生命周期，即数据的过期时间。当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 创建数据表时，如果希望数据永不过期，可以设置数据生命周期为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 单位为秒。 **说明** 如果需要使用索引，则数据生命周期必须设置为-1（数据永不过期）。 • maxVersions：最大版本数，即属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。 **说明** 如果需要使用索引，则最大版本数必须设置为1。 • maxTimeDeviation：有效版本偏差，即写入数据的时间戳与系统当前时间的偏差允许最大值。只有当写入数据所有列的版本号与写入时时间的差值在数据有效版本偏差范围内，数据才能成功写入。 属性列的有效版本范围为**[数据写入时间-有效版本偏差，数据写入时间+有效版本偏差)**。 创建数据表时，如果未设置有效版本偏差，系统会使用默认值86400；创建数据表后，可以通过UpdateTable接口动态修改有效版本偏差。 单位为秒。
reservedThroughtput	为数据表配置预留读吞吐量或预留写吞吐量。 容量型实例中的数据表的预留读/写吞吐量只能设置为0，不允许预留。 默认值为0，即完全按量计费。 单位为CU。 • 当预留读吞吐量或预留写吞吐量大于0时，表格存储会根据配置为数据表预留相应资源，且数据表创建成功后，将会立即按照预留吞吐量开始计费，超出预留的部分进行按量计费。更多信息，请参见计费概述。 • 当预留读吞吐量或预留写吞吐量设置为0时，表格存储不会为数据表预留相应资源。

参数	说明
indexMetas	索引表的结构信息，每个indexMeta都包括如下内容： • indexName：索引表名称。 • primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 • definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 • indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ◦ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ◦ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 • indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ◦ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ◦ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。

示例

- 创建数据表（不带索引）

```
private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME, PrimaryKeyType.STRING)); //为主表添加主键列。
        int timeToLive = -1; //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据生命周期必须设置为-1。
        int maxVersions = 3; //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引表的数据表最大版本数必须设置为1。
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
        request.setReservedThroughput(new ReservedThroughput(new CapacityUnit(0, 0))); //设置预留读写吞吐量，容量型实例中的数据表只能设置为0，高性能实例中的数据表可以设置为非零值。
        client.createTable(request);
}
```

- 创建数据表（带索引且索引类型为全局二级索引）

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType
.STRING)); //为数据表添加主键列。
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType
.INTEGER)); //为数据表添加主键列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnT
ype.STRING)); //为数据表添加预定义列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnT
ype.INTEGER)); //为数据表添加预定义列。
    int timeToLive = -1; //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据生命周期必须设置为-1。
    int maxVersions = 1; //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引表的数据表最大版本数必须设置为1。
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME);
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列。
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); //为索引表添加属性列。
    indexMetas.add(indexMeta);
    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMet
as); //创建数据表的同时创建索引表。
    client.createTable(request);
}
```

- 创建数据表（带索引且索引类型为本地二级索引）

```
private static void createTable(SyncClient client) {
    TableMeta tableMeta = new TableMeta(TABLE_NAME);
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType.STRING)); //为数据表添加主键列。
    tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType.INTEGER)); //为数据表添加主键列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnType.STRING)); //为数据表添加预定义列。
    tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnType.INTEGER)); //为数据表添加预定义列。
    int timeToLive = -1; //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据生命周期必须设置为-1。
    int maxVersions = 1; //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引表的数据表最大版本数必须设置为1。
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME);
    indexMeta.setIndexType(IT_LOCAL_INDEX);     //设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    indexMeta.setIndexUpdateMode(IUM_SYNC_INDEX);  //设置索引更新模式为同步更新（IUM_SYNC_INDEX）。当索引类型为本地二级索引时，索引更新模式必须为同步更新。
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1); //为索引表添加主键列。索引表的第一列主键必须与数据表的第一列主键相同。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列。
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); //为索引表添加属性列。
    indexMetas.add(indexMeta);
    CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMetas); //创建数据表的同时创建索引表。
    client.createTable(request);
}
```

3.4.3. 更新表

您可以使用更新表接口（UpdateTable）修改配置信息（TableOptions）以及预留读/写吞吐量（ReservedThroughput）。

参数说明

- TableOptions

  TableOptions 包含表的 TTL、MaxVersions 和 MaxTimeDeviation 。

参数	定义	说明
TTL	TimeToLive，数据存活时间	○ 单位：秒。 ○ 如果期望数据永不过期，TTL 可设置为 -1。 ○ 数据是否过期是根据数据的时间戳、当前时间、表的 TTL三者进行判断的。`当前时间 - 数据的时间戳 > 表的 TTL` 时，数据会过期并被清理。 ○ 在使用 TTL 功能时需要注意写入时是否指定了时间戳，以及指定的时间戳是否合理。如需指定时间戳，建议设置MaxTimeDeviation。

参数	定义	说明
MaxTimeDeviation	写入数据的时间戳与系统当前时间的偏差允许最大值	○ 默认情况下系统会为新写入的数据生成一个时间戳，数据自动过期功能需要根据这个时间戳判断数据是否过期。用户也可以指定写入数据的时间戳。如果用户写入的时间戳非常小，与当前时间偏差已经超过了表上设置的 TTL 时间，写入的数据会立即过期。设置 MaxTimeDeviation 可以避免这种情况。 ○ 单位：秒。
MaxVersions	每个属性列保留的最大版本数	如果写入的版本数超过 MaxVersions，服务端只会保留 MaxVersions 中指定的最大的版本。

- ReservedThroughtput

  表的预留读/写吞吐量配置。

  - ReservedThroughput 的调整有时间间隔限制，目前调整间隔为 1 分钟。
  - 设置 ReservedThroughtput 后，表格存储按照您预留读/写吞吐量进行计费。
  - 当 ReservedThroughtput 大于 0 时，表格存储会按照预留量和持续时间进行计费，超出预留的部分进行按量计费。更多信息参见计费，以免产生未期望的费用。
  - 默认值为 0，即完全按量计费。
  - 容量型实例的预留读/写吞吐量只能设置为 0，不允许预留。

示例

更新表的 TTL 和最大版本数。

```
private static void updateTable(SyncClient client) {
    int timeToLive = -1;
    int maxVersions = 5; // 将最大版本数更新为5。
    TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
    UpdateTableRequest request = new UpdateTableRequest(TABLE_NAME);
    request.setTableOptionsForUpdate(tableOptions);
    client.updateTable(request);
}
```

3.4.4. 列出表名称

使用ListTable接口获取当前实例下已创建的所有表的表名。

> **说明** API说明请参见ListTable。

示例

获取实例下所有表的表名。

```
private static void listTable(SyncClient client) {
    ListTableResponse response = client.listTable();
    System.out.println("表的列表如下：");
    for (String tableName : response.getTableNames()) {
        System.out.println(tableName);
    }
}
```

3.4.5. 预定义列操作

为数据表增加预定义列或删除数据表的预定义列。设置预定义列后，在创建二级索引时将预定义列作为索引表的索引列或者属性列。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表。

增加预定义列

使用二级索引时，如果未设置预定义列或者预定义列不满足需求，可以为数据表增加预定义列。

- 参数

参数	说明
tableName	数据表名称。
definedColumns	为数据表预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。包含如下设置： ○ name：预定义列名称。 ○ type：预定义列的数据类型。

- 示例

  为sampleTable数据表增加预定义列，预定义列分别为definedColumnName01（String类型）、definedColumnName02（INTEGER类型）和definedColumnName03（String类型）。

```
public static void addDefinedColumnRequest(SyncClient client) {
AddDefinedColumnRequest addDefinedColumnRequest = new AddDefinedColumnRequest();
addDefinedColumnRequest.setTableName("sampleTable");
addDefinedColumnRequest.addDefinedColumn("definedColumnName01",DefinedColumnType.STRING);
addDefinedColumnRequest.addDefinedColumn("definedColumnName02",DefinedColumnType.INTEGER)
;
addDefinedColumnRequest.addDefinedColumn("definedColumnName03",DefinedColumnType.STRING);
client.addDefinedColumn(addDefinedColumnRequest);
}
```

删除预定义列

删除数据表上不需要的预定义列。

- 参数

参数	说明
tableName	数据表名称。
name	预定义列名称。

- 示例

  删除sampleTable数据表的预定义列definedColumnName01和definedColumnName02。

```
public static void deleteDefinedColumnRequest(SyncClient client){
DeleteDefinedColumnRequest deleteDefinedColumnRequest = new DeleteDefinedColumnRequest();
deleteDefinedColumnRequest.setTableName("sampleTable");
deleteDefinedColumnRequest.addDefinedColumn("definedColumnName01");
deleteDefinedColumnRequest.addDefinedColumn("definedColumnName02");
client.deleteDefinedColumn(deleteDefinedColumnRequest);
}
```

3.4.6. 指定大小计算分片

使用ComputeSplitsBySize接口可以将全表数据逻辑上划分成若干接近指定大小的分片，并返回这些分片之间的分割点以及分片所在机器的提示。一般用于计算引擎规划并发度等执行计划。

> **说明** API说明请参见ComputeSplitPointsBySize。

前提条件

- 已初始化OTSClient，详情请参见初始化。
- 已创建数据表并写入数据。

参数

参数	说明
tableName	数据表名称。
splitSize	每个分片的近似大小。 单位为百兆（即100 MB）。

示例

将全表的数据在逻辑上划分成接近200 MB的若干分片。

```
private static void describeTable(SyncClient client) {
    //以200 MB划分分片。
    ComputeSplitsBySizeRequest request = new ComputeSplitsBySizeRequest(TABLE_NAME, 2);
    ComputeSplitsBySizeResponse response = client.computeSplitsBySize(computeSplitsBySizeRe
quest);
    System.out.println("ConsumedCapacity=" + response.getConsumedCapacity().jsonize());
    System.out.println("PrimaryKeySchema=" + response.getPrimaryKeySchema());
    System.out.println("RequestId=" + response.getRequestId());
    System.out.println("TraceId=" + response.getTraceId());
    List<Split> splits = response.getSplits();
    System.out.println("splits.size=" + splits.size());
    Iterator<Split> iterator = splits.iterator();
    while (iterator.hasNext()) {
        Split split = iterator.next();
        System.out.println("split.getLocation()=" + split.getLocation());
        //split.getLowerBound()和split.getUpperBound()可以直接灌进RangeRowQueryCriteria交给ge
tRange()或createRangeIterator()
        System.out.println("split.getLowerBound()=" + split.getLowerBound().jsonize());
        System.out.println("split.getUpperBound()=" + split.getUpperBound().jsonize());
    }
}
```

3.4.7. 查询表描述信息

使用DescribeTable接口可以查询指定表的结构、预留读/写吞吐量详情等信息。

> **说明** API说明请参见DescribeTable。

参数

参数	说明
tableName	表名。

示例

```
private static void describeTable(SyncClient client) {
    DescribeTableRequest request = new DescribeTableRequest(TABLE_NAME);
    DescribeTableResponse response = client.describeTable(request);
    TableMeta tableMeta = response.getTableMeta();
    System.out.println("表的名称: " + tableMeta.getTableName());
    System.out.println("表的主键: ");
    for (PrimaryKeySchema primaryKeySchema : tableMeta.getPrimaryKeyList()) {
        System.out.println(primaryKeySchema);
    }
    TableOptions tableOptions = response.getTableOptions();
    System.out.println("表的TTL:" + tableOptions.getTimeToLive());
    System.out.println("表的MaxVersions:" + tableOptions.getMaxVersions());
    ReservedThroughputDetails reservedThroughputDetails = response.getReservedThroughputDet
ails();
    System.out.println("表的预留读吞吐量: "
            + reservedThroughputDetails.getCapacityUnit().getReadCapacityUnit());
    System.out.println("表的预留写吞吐量: "
            + reservedThroughputDetails.getCapacityUnit().getWriteCapacityUnit());
}
```

3.4.8. 删除数据表

使用DeleteTable接口删除当前实例下指定数据表。

> **说明** API说明请参见DeleteTable。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表。
- 已删除数据表上的索引表和多元索引。

参数

参数	说明
tableName	数据表名称。

示例

删除指定数据表。

```
private static void deleteTable(SyncClient client) {
    DeleteTableRequest request = new DeleteTableRequest(TABLE_NAME);
    client.deleteTable(request);
}
```

3.4.9. 主键列自增

设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。

> **说明** 从Java SDK 4.2.0版本开始支持主键列自增功能。

前提条件

已初始化Client，详情请参见初始化。

使用方法

1. 创建表时，将非分区键的主键列设置为自增列。

   只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号长整型。

2. 写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符。

   如果需要获取写入数据后系统自动生成的自增列的值，将ReturnType设置为RT_PK，可以在数据写入成功后返回自增列的值。

   查询数据时，需要完整的主键值。通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为RT_PK可以获取完整的主键值。

示例

主键自增列功能主要涉及创建表（CreateTable）和写数据（PutRow、UpdateRow和BatchWriteRow）两类接口。

1. 创建表

   创建表时，只需将自增的主键属性设置为AUTO_INCREMENT。

```
private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta("table_name");
        //第一列为分区键。
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_1", PrimaryKeyType.STRIN
G));
        //第二列为自增列，类型为INTEGER，属性为AUTO_INCREMENT。
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_2", PrimaryKeyType.INTEG
ER, PrimaryKeyOption.AUTO_INCREMENT));
        int timeToLive = -1;  //数据永不过期。
        int maxVersions = 1;  //只保存一个数据版本。
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
        client.createTable(request);
    }
```

2. 写数据

   写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符AUTO_INCREMENT。

```
    private static void putRow(SyncClient client, String receive_id) {
        //构造主键。
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder
();
        //第一列的值为md5(receive_id)前4位。
        primaryKeyBuilder.addPrimaryKeyColumn("PK_1", PrimaryKeyValue.fromString("Hangz
hou"));
        //第二列是主键自增列，此处无需填入具体值，只需要一个占位符AUTO_INCREMENT，表格存储会自动
生成此值。
        primaryKeyBuilder.addPrimaryKeyColumn("PK_2", PrimaryKeyValue.AUTO_INCREMENT);
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        RowPutChange rowPutChange = new RowPutChange("table_name", primaryKey);
        //此处设置返回类型为RT_PK，即在返回结果中包含PK列的值。如果不设置ReturnType，默认不返回
。
        rowPutChange.setReturnType(ReturnType.RT_PK);
        //加入属性列。
        rowPutChange.addColumn(new Column("content", ColumnValue.fromString("content"))
);
        //写入数据到表格存储。
        PutRowResponse response = client.putRow(new PutRowRequest(rowPutChange));
        //打印返回的PK列。
        Row returnRow = response.getRow();
        if (returnRow != null) {
            System.out.println("PrimaryKey:" + returnRow.getPrimaryKey().toString());
        }
        //打印消耗的CU。
        CapacityUnit  cu = response.getConsumedCapacity().getCapacityUnit();
        System.out.println("Read CapacityUnit:" + cu.getReadCapacityUnit());
        System.out.println("Write CapacityUnit:" + cu.getWriteCapacityUnit());
    }
```

3.4.10. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

前提条件

- 已初始化OTSClient，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过PutRow、UpdateRow、DeleteRow或BatchWriteRow接口更新数据时，可以使用条件更新检查行存在性条件和列条件，只有满足条件时才能更新成功。

条件更新包括行存在性条件和列条件。

- 行存在性条件：包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。

  对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。

- 列条件：包括SingleColumnValueCondition和CompositeColumnValueCondition，是基于某一列或者某些列的列值进行条件判断。

- SingleColumnValueCondition支持一列和一个常量比较。不支持两列或者两个常量相比较。
- CompositeColumnValueCondition的内节点为逻辑运算，子条件可以是SingleColumnValueCondition或CompositeColumnValueCondition。

条件更新可以实现乐观锁功能，即在更新某行时，先获取某列的值，假设为列A，值为1，然后设置条件**列A = 1**，更新行使**列A = 2**。如果更新失败，表示有其他客户端已成功更新该行。

限制

条件更新的列条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。

参数

参数	说明
RowExistenceExpectation	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST。 • IGNORE：表示忽略，不做任何存在性检查。 • EXPECT_EXIST：表示期望存在，如果该行存在，则满足条件；如果该行不存在，则不满足条件。 • EXPECT_NOT_EXIST：期望行不存在，如果该行不存在，则满足条件；如果该行存在，则不满足条件。
ColumnName	列的名称。
ColumnValue	列的对比值。
CompareOperator	对列值进行比较的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=）。
LogicOperator	对多个条件进行组合的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR。 逻辑运算符不同可以添加的子条件个数不同。 • 当逻辑运算符为NOT时，只能添加一个子条件。 • 当逻辑运算符为AND或OR时，必须至少添加两个子条件。
PassIfMissing	当列在某行中不存在时，条件检查是否通过。类型为bool值，默认值为true，表示如果列在某行中不存在时，则条件检查通过，该行满足更新条件。 当设置PassIfMissing为false时，如果列在某行中不存在时，则条件检查不通过，该行不满足更新条件。

参数	说明
LatestVersionsOnly	当列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置LatestVersionsOnly为false时，如果列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就条件检查通过，该行满足更新条件。

示例

使用列判断条件和乐观锁的示例代码如下：

- 构造SingleColumnValueCondition。

```
 //设置条件为Col0==0。
 SingleColumnValueCondition singleColumnValueCondition = new SingleColumnValueCondition("Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(0));
 //如果不存在Col0列，条件检查不通过。
 singleColumnValueCondition.setPassIfMissing(false);
 //只判断最新版本。
 singleColumnValueCondition.setLatestVersionsOnly(true);
```

- 构造CompositeColumnValueCondition。

```
 //composite1的条件为(Col0 == 0) AND (Col1 > 100)。
 CompositeColumnValueCondition composite1 = new CompositeColumnValueCondition(CompositeColumnValueCondition.LogicOperator.AND);
 SingleColumnValueCondition single1 = new SingleColumnValueCondition("Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(0));
 SingleColumnValueCondition single2 = new SingleColumnValueCondition("Col1",
         SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong(100));
 composite1.addCondition(single1);
 composite1.addCondition(single2);
 //composite2的条件为( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10)。
 CompositeColumnValueCondition composite2 = new CompositeColumnValueCondition(CompositeColumnValueCondition.LogicOperator.OR);
 SingleColumnValueCondition single3 = new SingleColumnValueCondition("Col2",
         SingleColumnValueCondition.CompareOperator.LESS_EQUAL, ColumnValue.fromLong(10));
 composite2.addCondition(composite1);
 composite2.addCondition(single3);
```

- 通过Condition实现乐观锁机制，递增一列。

```
 private static void updateRowWithCondition(SyncClient client, String pkValue) {
     //构造主键。
     PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
     primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(p
kValue));
     PrimaryKey primaryKey = primaryKeyBuilder.build();
     //读取一行数据。
     SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey)
;
     criteria.setMaxVersions(1);
     GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
     Row row = getRowResponse.getRow();
     long col0Value = row.getLatestColumn("Col0").getValue().asLong();
     //条件更新Col0列，使列值加1。
     RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
     Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
     ColumnCondition columnCondition = new SingleColumnValueCondition("Col0", SingleColum
nValueCondition.CompareOperator.EQUAL, ColumnValue.fromLong(col0Value));
     condition.setColumnCondition(columnCondition);
     rowUpdateChange.setCondition(condition);
     rowUpdateChange.put(new Column("Col0", ColumnValue.fromLong(col0Value + 1)));
     try {
         client.updateRow(new UpdateRowRequest(rowUpdateChange));
     } catch (TableStoreException ex) {
         System.out.println(ex.toString());
     }
 }
```

3.4.11. 局部事务

使用局部事务功能，创建数据范围在一个分区键值内的局部事务。对局部事务中的数据进行读写操作后，可以根据实际提交或者丢弃局部事务。局部事务通过悲观锁（Pessimistic Lock）实现并发控制。

目前局部事务功能处于邀测中，默认关闭。如果需要使用该功能，请提交工单进行申请或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。

使用局部事务可以指定某个分区键值内的操作是原子的，对分区键值内的数据进行的操作要么全部成功要么全部失败，并且所提供的隔离级别为读已提交。

前提条件

- 已初始化OTSClient，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

1. 使用StartLocalTransaction在指定的分区键值创建一个局部事务，并获取局部事务ID。
2. 对局部事务范围内的数据进行读写操作。

   支持对局部事务进行操作的接口为GetRow、PutRow、DeleteRow、UpdateRow、BatchWriteRow和GetRange。
3. 使用CommitTransaction提交局部事务或者使用AbortTransaction丢弃局部事务。

限制

- 每个局部事务从创建开始生命周期最长为60秒。

  如果超过60秒未提交或丢弃局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
- 如果创建局部事务时超时，此请求可能在表格存储服务端已执行成功，此时用户需要等待该局部事务超时后重新创建。
- 未提交的局部事务可能失效，如果出现此情况，需要重试该局部事务内的操作。
- 在局部事务中读写数据有如下限制：
  - 不能使用局部事务ID访问局部事务范围（即创建时使用的分区键值）以外的数据。
  - 同一个局部事务中所有写请求的分区键值必须与创建局部事务时的分区键值相同，读请求则无此限制。
  - 一个局部事务同时只能用于一个请求中，在使用局部事务期间，其它使用此局部事务ID的操作均会失败。
  - 每个局部事务中两次读写操作的最大间隔为60秒。

    如果超过60秒未操作局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
  - 每个局部事务中写入的数据量最大为4 MB，按正常的写请求数据量计算规则累加。
  - 如果在局部事务中写入了未指定版本号的Cell，该Cell的版本号会在写入时（而非提交时）由表格存储服务端自动生成，生成规则与正常写入一个未指定版本号的Cell相同。
  - 如果BatchWriteRow请求中带有局部事务ID，则此请求中所有行只能操作该局部事务ID对应的表。
  - 在使用局部事务期间，对应分区键值的数据相当于被锁上，只有持有局部事务ID在局部事务范围内的写请求才会成功，其它不持有局部事务ID在局部事务范围内的写请求均会失败。在局部事务提交、丢弃或超时后，对应的锁也会被释放。
  - 带有局部事务ID的读写请求失败不会影响局部事务本身的存活情况，您可以按照正常的无局部事务ID的读写请求重试规则进行重试，或者主动丢弃当前局部事务。

参数

参数	说明
TableName	数据表名称。
PrimaryKey	数据表主键。 • 创建局部事务时，只需要指定局部事务对应的分区键值。 • 创建局部事务后，对局部事务范围内的数据进行读写操作时，需要指定完整主键。
TransactionId	局部事务ID，用于唯一标识一个局部事务。 创建局部事务后，操作局部事务时均需要带上局部事务ID。

示例

1. 调用AsyncClient或SyncClient的startLocalTransaction方法使用指定分区键值创建一个局部事务，并获取局部事务ID。

```
PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
PrimaryKey primaryKey = primaryKeyBuilder.build();
StartLocalTransactionRequest request = new StartLocalTransactionRequest(tableName, prim
aryKey);
String txnId = client.startLocalTransaction(request).getTransactionID();
```

2. 对局部事务范围内的数据进行读写操作。

   对局部事务范围内数据的读写操作与正常读写数据操作基本相同，只需填入局部事务ID即可。

   - 写入一行数据。

     ```
     PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
     primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
     primaryKeyBuilder.addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong("userId"));
     PrimaryKey primaryKey = primaryKeyBuilder.build();
     RowPutChange rowPutChange = new RowPutChange(tableName, primaryKey);
     rowPutChange.addColumn(new Column("Col", ColumnValue.fromLong(columnValue)));
     PutRowRequest request = new PutRowRequest(rowPutChange);
     request.setTransactionId(txnId);
     client.putRow(request);
     ```

   - 读取此行数据。

     ```
     PrimaryKeyBuilder primaryKeyBuilder;
     primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
     primaryKeyBuilder.addPrimaryKeyColumn("pk1", PrimaryKeyValue.fromString("txnKey"));
     primaryKeyBuilder.addPrimaryKeyColumn("pk2", PrimaryKeyValue.fromLong("userId"));
     PrimaryKey primaryKey = primaryKeyBuilder.build();
     SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(tableName, primaryKey);
     criteria.setMaxVersions(1); //设置读取最新版本的数据。
     GetRowRequest  request = new GetRowRequest(criteria);
     request.setTransactionId(txnId);
     GetRowResponse getRowResponse = client.getRow(request);
     ```

3. 提交或丢弃局部事务。

   - 提交局部事务，使局部事务中的所有数据修改生效。

     ```
     CommitTransactionRequest commitRequest = new CommitTransactionRequest(txnId);
     client.commitTransaction(commitRequest);
     ```

   - 丢弃局部事务，局部事务中的所有数据修改均不会应用到原有数据。

     ```
     AbortTransactionRequest abortRequest = new AbortTransactionRequest(txnId);
     client.abortTransaction(abortRequest);
     ```

3.4.12. 原子计数器

将列当成一个原子计数器使用，对该列进行原子计数操作，可用于为某些在线应用提供实时统计功能，例如统计帖子的PV（实时浏览量）等。

前提条件

- 已初始化OTSClient，详情请参见初始化。

- 已创建数据表并写入数据。

限制

- 只支持对整型列的列值进行原子计数操作。
- 作为原子计数器的列，如果写入数据前该列不存在，则默认值为0；如果写入数据前该列已存在且列值非整型，则产生OTSParameterInvalid错误。
- 增量值可以是正数或负数，但不能出现计算溢出。如果出现计算溢出，则产生OTSParameterInvalid错误。
- 默认不返回进行原子计数操作的列值，可以通过相应操作指定返回进行原子计数操作的列值。
- 在单次更新请求中，不能对某一列同时进行更新和原子计数操作。假设列A已经执行原子计数操作，则列A不能再执行其他操作（例如列的覆盖写，列删除等）。
- 在一次BatchWriteRow请求中，支持对同一行进行多次更新操作。但是如果某一行已进行原子计数操作，则该行在此批量请求中只能出现一次。
- 原子计数操作只能作用在列值的最新版本，不支持对列值的特定版本做原子计数操作。更新完成后，原子计数操作会插入一个新的数据版本。

接口

rowUpdateChange类中新增了原子计数器的操作接口，操作接口说明请参见下表。

接口	说明
RowUpdateChange increment(Column column)	对列执行增量变更，例如+X，-X等。
void addReturnColumn(String columnName)	对于进行原子计数操作的列，设置需要返回列值的列名。
void setReturnType(ReturnType returnType)	设置返回类型，返回进行原子计数操作的列的新值。

参数

参数	说明
tableName	数据表名称。
columnName	进行原子计数操作的列名。只支持对整型列的列值进行原子计数器操作。
value	对列进行增量变更的值。
returnType	设置返回类型为ReturnType.RT_AFTER_MODIFY，将进行原子计数操作的列值返回。

示例

写入数据时，使用rowUpdateChange接口对整型列做列值的增量变更，然后读取更新后的新值。

```
private static void incrementByUpdateRowApi(SyncClient client) {
        //构造主键。
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
        primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(
"pk0"));
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        //设置数据表名称。
        RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
        //将进行原子计数操作的price列的列值+10，不能设置时间戳。
        rowUpdateChange.increment(new Column("price", ColumnValue.fromLong(10)));
        //设置returnType为ReturnType.RT_AFTER_MODIFY，将进行原子计数操作的列值返回。
        rowUpdateChange.addReturnColumn("price");
        rowUpdateChange.setReturnType(ReturnType.RT_AFTER_MODIFY);
        //对price列进行原子计数操作。
        UpdateRowResponse response = client.updateRow(new UpdateRowRequest(rowUpdateChange)
);
        //打印更新后的新值。
        Row row = response.getRow();
        System.out.println(row);
    }
```

3.4.13. 过滤器

在服务端对读取结果再进行一次过滤，根据过滤器（Filter）中的条件决定返回的行。使用过滤器后，只返回符合条件的数据行。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过GetRow、BatchGetRow或GetRange接口查询数据时，可以使用过滤器只返回符合条件的数据行。

过滤器目前包括SingleColumnValueFilter、SingleColumnValueRegexFilter和CompositeColumnValueFilter。

- SingleColumnValueFilter：只判断某个参考列的列值。
- SingleColumnValueRegexFilter：支持对类型为String的列值，使用正则表达式进行子字符串匹配，然后根据实际将匹配到的子字符串转换为String、Integer或者Double类型，再对子值使用过滤器进行过滤。
- CompositeColumnValueFilter：根据多个参考列的列值的判断结果进行逻辑组合，决定是否过滤某行。

> **说明** 关于过滤器的更多信息，请参见功能介绍中的过滤器。

限制

- 过滤器的条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。
- 过滤器中的参考列必须在读取的结果内。如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。

- 在GetRow、BatchGetRow和GetRange接口中使用过滤器不会改变接口的原生语义和限制项。

  使用GetRange接口时，一次扫描数据的行数不能超过5000行或者数据大小不能超过4 MB。

  当在该次扫描的5000行或者4 MB数据中没有满足过滤器条件的数据时，得到的Response中的Rows为空，但是NextStartPrimaryKey可能不为空，此时需要使用NextStartPrimaryKey继续读取数据，直到NextStartPrimaryKey为空。

参数

参数	说明
ColumnName	过滤器中参考列的名称。
ColumnValue	过滤器中参考列的对比值。
CompareOperator	过滤器中的关系运算符。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=）。
LogicOperator	过滤器中的逻辑运算符。 逻辑运算符包括NOT、AND和OR。
PassIfMissing	当参考列在某行中不存在时，是否返回该行。取值范围如下： • true（默认）：如果参考列在某行中不存在时，则返回该行。 • false：如果参考列在某行中不存在时，则不返回该行。
LatestVersionsOnly	当参考列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果参考列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置LatestVersionsOnly为false时，如果参考列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就返回该行。
Regex	正则表达式，用于匹配子字段值。正则表达式必须满足以下条件： • 长度不能超过256个字节。 • 支持perl regular语法。 • 支持单字节正则表达式。 • 不支持中文的正则匹配。 • 支持正则表达式的全匹配模式和部分匹配模式。 部分匹配的正则表达式在模式中由一对括号（...）分隔。 如果正则表达式为全匹配模式，则返回第一个匹配结果；如果正则表达式中包含部分匹配语法，则返回第一个满足的子匹配结果。例如列值为1aaa51bbb5，如果正则表达式为1[a-z]+5时，则返回值为1aaa5；如果正则表达式为1([a-z]+)5，则返回值为aaa。
VariantType	使用正则表达式匹配到子字段值后，子字段值转换为的类型。取值范围为VT_INTEGER（整型）、VT_STRING（字符串类型）和VT_DOUBLE（双精度浮点型）。

示例

- 构造SingleColumnValueFilter。

```
//设置过滤器，当Col0列的值为0时，返回该行。
SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
        SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
//如果不存在Col0列，也不返回该行。
singleColumnValueFilter.setPassIfMissing(false);
//只使用该列最新版本的值进行比较。
singleColumnValueFilter.setLatestVersionsOnly(true);
```

- 构造SingleColumnValueRegexFilter。

```
//构造正则抽取规则。
RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
//设置过滤器，实现cast<int>(regex(col1)) > 0。
//构造SingleColumnValueRegexFilter，格式为"列名,正则规则,比较符,比较值"。
SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
    regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(0));
//如果不存在Col0列，也不返回该行。
filter.setPassIfMissing(false);
```

- 构造CompositeColumnValueFilter。

```
//composite1的条件为(Col0 == 0) AND (Col1 > 100)。
CompositeColumnValueFilter composite1 = new CompositeColumnValueFilter(CompositeColumnV
alueFilter.LogicOperator.AND);
SingleColumnValueFilter single1 = new SingleColumnValueFilter("Col0",
        SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
SingleColumnValueFilter single2 = new SingleColumnValueFilter("Col1",
        SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromLong(100)
);
composite1.addFilter(single1);
composite1.addFilter(single2);
//composite2的条件为( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10)。
CompositeColumnValueFilter composite2 = new CompositeColumnValueFilter(CompositeColumnV
alueFilter.LogicOperator.OR);
SingleColumnValueFilter single3 = new SingleColumnValueFilter("Col2",
        SingleColumnValueFilter.CompareOperator.LESS_EQUAL, ColumnValue.fromLong(10));
composite2.addFilter(composite1);
composite2.addFilter(single3);
```

3.5. 单行数据操作

表格存储提供了PutRow、GetRow、UpdateRow和DeleteRow等单行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

> **说明** 本文示例中的pkValue均表示主键列值，使用时请根据实际填写具体数据。

前提条件

- 已初始化OTSClient，详情请参见初始化。

- 已创建数据表并写入数据。

插入一行数据（PutRow）

PutRow接口用于新写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** ◦ 设置的主键个数和类型必须和数据表的主键个数和类型一致。 ◦ 当主键为自增列时，只需将自增列的值设置为占位符，详情请参见主键列自增。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。 **说明** ◦ RowExistenceExpectation.IGNORE表示无论此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。 ◦ RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。 ◦ RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。
column	行的属性列。 ◦ 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType（可选）、时间戳（可选）。 ◦ ColumnType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、BOOLEAN、DOUBLE五种，分别用ColumnType.INTEGER、ColumnType.STRING、ColumnType.BINARY、ColumnType.BOOLEAN、ColumnType.DOUBLE表示，其中BINARY不可省略，其他类型都可以省略。 ◦ 时间戳即数据的版本号，详情请参见数据版本和生命周期。 数据的版本号可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。

- 示例1

  写入10列属性列，每列写入1个版本，由系统自动生成数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //加入一些属性列。
    for (int i = 0; i < 10; i++) {
        rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

- 示例2

  写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //加入一些属性列。
    long ts = System.currentTimeMillis();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 3; j++) {
            rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(j), ts + j)
);
        }
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

- 示例3

  期望原行不存在时，写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //设置条件更新，行条件检查为期望原行不存在。
    rowPutChange.setCondition(new Condition(RowExistenceExpectation.EXPECT_NOT_EXIST));
    //加入一些属性列。
    long ts = System.currentTimeMillis();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 3; j++) {
            rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(j), ts + j)
);
        }
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

- 示例4

  期望原行存在且Col0列的值大于100时，写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME, primaryKey);
    //设置条件更新，期望原行存在且Col0列的值大于100时写入数据。
    Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
    condition.setColumnCondition(new SingleColumnValueCondition("Col0",
            SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(100)));
    rowPutChange.setCondition(condition);
    //加入一些属性列。
    long ts = System.currentTimeMillis();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 3; j++) {
            rowPutChange.addColumn(new Column("Col" + i, ColumnValue.fromLong(j), ts + j)
);
        }
    }
    client.putRow(new PutRowRequest(rowPutChange));
}
```

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

读取的结果可能有如下两种：

- 如果该行存在，则返回该行的各主键列以及属性列。
- 如果该行不存在，则返回中不包含行，并且不会报错。
- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ◦ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 ◦ 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 ◦ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。

参数	说明
timeRange	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 ◦ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ◦ 查询一个范围的数据，则需要设置start和end。start和end分别表示起始时间戳和结束时间戳，范围为前闭后开区间。 ◦ 如果查询特定版本号的数据，则需要设置timestamp。timestamp表示特定的时间戳。 timestamp和[start, end)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Long.MAX_VALUE。
filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行。具体操作，请参见过滤器。 **说明** 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。

- 示例1

  读取一行，设置读取最新版本的数据和读取的列。

```
private static void getRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //读取一行数据，设置数据表名称。
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey);
    //设置读取最新版本。
    criteria.setMaxVersions(1);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为： ");
    System.out.println(row);
    //设置读取某些列。
    criteria.addColumnsToGet("Col0");
    getRowResponse = client.getRow(new GetRowRequest(criteria));
    row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为：");
    System.out.println(row);
}
```

- 示例2

  在读取一行数据时使用过滤器。

```
private static void getRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //读取一行数据，设置数据表名称。
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, primaryKey);
    //设置读取最新版本。
    criteria.setMaxVersions(1);
    //设置过滤器，当Col0列的值为0时，返回该行。
    SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
            SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
    //如果Col0列不存在，则不返回该行。
    singleColumnValueFilter.setPassIfMissing(false);
    criteria.setFilter(singleColumnValueFilter);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为：");
    System.out.println(row);
}
```

- 示例3

  读取一行中Col1列的数据，并对该列的数据执行正则过滤。

```
private static void getRow(SyncClient client, String pkValue) {
    SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(tableName);
    //构造主键。
    PrimaryKey primaryKey = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pkValue))
        .build();
    criteria.setPrimaryKey(primaryKey);
    // 设置读取最新版本。
    criteria.setMaxVersions(1);
    // 设置过滤器，当cast<int>(regex(Col1)) > 100时，返回该行。
    RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
    SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
        regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN, ColumnValue.fromL
ong(100));
    criteria.setFilter(filter);
    GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
    Row row = getRowResponse.getRow();
    System.out.println("读取完毕，结果为：");
    System.out.println(row);
}
```

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，可以增加和删除一行中的属性列，删除属性列指定版本的数据，或者更新已存在的属性列的值。如果更新的行不存在，则新增一行数据。

> **说明** 当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

参数	说明
column	更新的属性列。 ○ 增加或更新数据时，需要设置属性名、属性值、属性类型（可选）、时间戳（可选）。 时间戳即数据的版本号，可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。详情请参见数据版本和生命周期。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。 ○ 删除属性列特定版本的数据时，只需要设置属性名和时间戳。 时间戳是64位整数，单位为毫秒，表示某个特定版本的数据。 ○ 删除属性列时，只需要设置属性名。 **说明** 删除一行的全部属性列不等同于删除该行，如果需要删除该行，请使用DeleteRow操作。

- 示例1

  更新一些列，删除某列的某一版本数据，删除某列。

```
private static void updateRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
    //更新一些列。
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //删除某列的某一版本。
    rowUpdateChange.deleteColumn("Col10", 1465373223000L);
    //删除某一列。
    rowUpdateChange.deleteColumns("Col11");
    client.updateRow(new UpdateRowRequest(rowUpdateChange));
}
```

- 示例2

  设置更新的条件。

```
private static void updateRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, primaryKey);
    //设置条件更新，期望原行存在且Col0列的值大于100时更新数据。
    Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
    condition.setColumnCondition(new SingleColumnValueCondition("Col0",
            SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(100)));
    rowUpdateChange.setCondition(condition);
    //更新一些列。
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //删除某列的某一版本。
    rowUpdateChange.deleteColumn("Col10", 1465373223000L);
    //删除某一列。
    rowUpdateChange.deleteColumns("Col11");
    client.updateRow(new UpdateRowRequest(rowUpdateChange));
}
```

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行数据。如果删除的行不存在，则不会发生任何变化。

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	支持使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

- 示例1

  删除一行数据。

```
private static void deleteRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey);
    client.deleteRow(new DeleteRowRequest(rowDeleteChange));
}
```

- 示例2

  设置删除条件。

```
private static void deleteRow(SyncClient client, String pkValue) {
    //构造主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(pk
Value));
    PrimaryKey primaryKey = primaryKeyBuilder.build();
    //设置数据表名称。
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey);
    //设置条件更新，期望原行存在且Col0列的值大于100时删除该行。
    Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
    condition.setColumnCondition(new SingleColumnValueCondition("Col0",
            SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.fromLong
(100)));
    rowDeleteChange.setCondition(condition);
    client.deleteRow(new DeleteRowRequest(rowDeleteChange));
}
```

3.6. 多行数据操作

表格存储提供了BatchWriteRow、BatchGetRow、GetRange和createRangeIterator等多行数据操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化OTSClient，详情请参见初始化。
- 已创建数据表并写入数据。

批量写（BatchWriteRow）

批量写接口用于在一次请求中进行批量的写入操作，也支持一次对多张表进行写入。BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，构造子操作的过程与使用PutRow接口、UpdateRow接口和DeleteRow接口时相同，也支持设置更新条件。

> **说明** 如果需要批量删除数据，请参见如何批量删除数据。

BatchWriteRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量写入可能存在部分行失败的情况，失败行的Index及错误信息在返回的BatchWriteRowResponse中，但并不抛出异常。因此调用BatchWriteRow接口时，需要检查返回值，可通过BatchWriteRowResponse的isAllSucceed方法判断是否全部成功；如果不检查返回值，则可能会忽略掉部分操作的失败。

当服务端检查到某些操作出现参数错误时，BatchWriteRow接口可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。

- 参数

  详细参数说明请参见单行数据操作。

- 示例

  一次BatchWriteRow请求，包含2个PutRow操作、1个UpdateRow操作和1个DeleteRow操作。

```
private static void batchWriteRow(SyncClient client) {
    BatchWriteRowRequest batchWriteRowRequest = new BatchWriteRowRequest();
    //构造rowPutChange1。
    PrimaryKeyBuilder pk1Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk1Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk1"));
    RowPutChange rowPutChange1 = new RowPutChange(TABLE_NAME, pk1Builder.build());
    //添加一些列。
    for (int i = 0; i < 10; i++) {
        rowPutChange1.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowPutChange1);
    //构造rowPutChange2。
    PrimaryKeyBuilder pk2Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk2Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk2"));
    RowPutChange rowPutChange2 = new RowPutChange(TABLE_NAME, pk2Builder.build());
    //添加一些列。
    for (int i = 0; i < 10; i++) {
        rowPutChange2.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowPutChange2);
    //构造rowUpdateChange。
    PrimaryKeyBuilder pk3Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk3Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk3"));
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, pk3Builder.build())
;
    //添加一些列。
    for (int i = 0; i < 10; i++) {
        rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
    }
    //删除一列。
    rowUpdateChange.deleteColumns("Col10");
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowUpdateChange);
    //构造rowDeleteChange。
    PrimaryKeyBuilder pk4Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    pk4Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk4"));
    RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, pk4Builder.build())
;
    //添加到batch操作中
```

```
    //添加到batch操作中。
    batchWriteRowRequest.addRowChange(rowDeleteChange);
    BatchWriteRowResponse response = client.batchWriteRow(batchWriteRowRequest);
    System.out.println("是否全部成功：" + response.isAllSucceed());
    if (!response.isAllSucceed()) {
        for (BatchWriteRowResponse.RowResult rowResult : response.getFailedRows()) {
            System.out.println("失败的行：" + batchWriteRowRequest.getRowChange(rowResult.
getTableName(), rowResult.getIndex()).getPrimaryKey());
            System.out.println("失败原因：" + rowResult.getError());
        }
        /**
         * 可以通过createRequestForRetry方法再构造一个请求对失败的行进行重试。此处只给出构造重试
请求的部分。
         * 推荐的重试方法是使用SDK的自定义重试策略功能，支持对batch操作的部分行错误进行重试。设置重
试策略后，调用接口处无需增加重试代码。
         */
        BatchWriteRowRequest retryRequest = batchWriteRowRequest.createRequestForRetry(re
sponse.getFailedRows());
    }
}
```

详细代码请参见BatchWriteRow@GitHub。

批量读（BatchGetRow）

批量读接口用于一次请求读取多行数据，也支持一次对多张表进行读取。BatchGetRow由多个GetRow子操作组成。构造子操作的过程与使用GetRow接口时相同。

批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA列。

BatchGetRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量读取可能存在部分行失败的情况，失败行的错误信息在返回的BatchGetRowResponse中，但并不抛出异常。因此调用BatchGetRow接口时，需要检查返回值，可通过BatchGetRowResponse的isAllSucceed方法判断是否所有行都获取成功；通过BatchGetRowResponse的getFailedRows方法获取失败行的信息。

- 参数

  详细参数说明请参见单行数据操作。

- 示例

  读取10行，设置版本条件、要读取的列、过滤器等。

```
private static void batchGetRow(SyncClient client) {
    MultiRowQueryCriteria multiRowQueryCriteria = new MultiRowQueryCriteria(TABLE_NAME);
    //加入10个要读取的行。
    for (int i = 0; i < 10; i++) {
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder()
;
        primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromStrin
g("pk" + i));
        PrimaryKey primaryKey = primaryKeyBuilder.build();
        multiRowQueryCriteria.addRow(primaryKey);
    }
    //添加条件。
    multiRowQueryCriteria.setMaxVersions(1);
    multiRowQueryCriteria.addColumnsToGet("Col0");
    multiRowQueryCriteria.addColumnsToGet("Col1");
    SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Col0",
            SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
    singleColumnValueFilter.setPassIfMissing(false);
    multiRowQueryCriteria.setFilter(singleColumnValueFilter);
    BatchGetRowRequest batchGetRowRequest = new BatchGetRowRequest();
    //BatchGetRow支持读取多个表的数据，一个multiRowQueryCriteria对应一个表的查询条件，可以添加多
个multiRowQueryCriteria。
    batchGetRowRequest.addMultiRowQueryCriteria(multiRowQueryCriteria);
    BatchGetRowResponse batchGetRowResponse = client.batchGetRow(batchGetRowRequest);
    System.out.println("是否全部成功：" + batchGetRowResponse.isAllSucceed());
    if (!batchGetRowResponse.isAllSucceed()) {
        for (BatchGetRowResponse.RowResult rowResult : batchGetRowResponse.getFailedRows(
)) {
            System.out.println("失败的行：" + batchGetRowRequest.getPrimaryKey(rowResult.g
etTableName(), rowResult.getIndex()));
            System.out.println("失败原因：" + rowResult.getError());
        }
        /**
         * 可以通过createRequestForRetry方法再构造一个请求对失败的行进行重试。此处只给出构造重试
请求的部分。
         * 推荐的重试方法是使用SDK的自定义重试策略功能，支持对batch操作的部分行错误进行重试。设置重
试策略后，调用接口处无需增加重试代码。
         */
        BatchGetRowRequest retryRequest = batchGetRowRequest.createRequestForRetry(batchG
etRowResponse.getFailedRows());
    }
}
```

详细代码请参见BatchGetRow@GitHub。

范围读（GetRange）

范围读取接口用于读取一个范围内的数据。

范围读取接口支持按照确定范围进行正序读取和逆序读取，可以设置要读取的行数。如果范围较大，已扫描的行数或者数据量超过一定限制，会停止扫描，并返回已获取的行和下一个主键信息。您可以根据返回的下一个主键信息，继续发起请求，获取范围内剩余的行。

> **说明** 表格存储表中的行都是按照主键排序的，而主键是由全部主键列按照顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

GetRange操作可能在如下情况停止执行并返回数据。

- 扫描的行数据大小之和达到4 MB。
- 扫描的行数等于5000。
- 返回的行数等于最大返回行数。
- 当前剩余的预留读吞吐量已全部使用，余量不足以读取下一条数据。
- 参数

参数	说明
tableName	数据表名称。
direction	读取方向。 ○ 如果值为正序（FORWARD），则起始主键必须小于结束主键，返回的行按照主键由小到大的顺序进行排列。 ○ 如果值为逆序（BACKWARD），则起始主键必须大于结束主键，返回的行按照主键由大到小的顺序进行排列。 例如同一表中有两个主键A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。
inclusiveStartPrimaryKey	本次范围读取的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由INF_MIN和INF_MAX类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中INF_MIN表示无限小，任何类型的值都比它大；INF_MAX表示无限大，任何类型的值都比它小。 ○ inclusiveStartPrimaryKey表示起始主键，如果该行存在，则返回结果中一定会包含此行。 ○ exclusiveEndPrimaryKey表示结束主键，无论该行是否存在，返回结果中都不会包含此行。 数据表中的行按主键从小到大排序，读取范围是一个左闭右开的区间，正序读取时，返回的是大于等于起始主键且小于结束主键的所有的行。
exclusiveEndPrimaryKey	
limit	数据的最大返回行数，此值必须大于 0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。此时可以通过返回结果中的nextStartPrimaryKey记录本次读取到的位置，用于下一次读取。

参数	说明
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ◦ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 ◦ 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。 ◦ 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 ◦ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。
timeRange	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 ◦ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ◦ 如果查询一个范围的数据，则需要设置start和end。start和end分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[start, end)。 ◦ 如果查询特定版本号的数据，则需要设置timestamp。timestamp表示特定的时间戳。 timestamp和[start, end)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Long.MAX_VALUE。

参数	说明
filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行。具体操作，请参见过滤器。 **说明** 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
nextStartPrimaryKey	根据返回结果中的nextStartPrimaryKey判断数据是否全部读取。 ○ 当返回结果中nextStartPrimaryKey不为空时，可以使用此返回值作为下一次GetRange操作的起始点继续读取数据。 ○ 当返回结果中nextStartPrimaryKey为空，表示读取范围内的数据全部返回。

- 示例1

  按照确定范围进行正序读取，判断nextStartPrimaryKey是否为空，读取完范围内的全部数据。

```
private static void getRange(SyncClient client, String startPkValue, String endPkValue) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    //设置起始主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(st
artPkValue));
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    //设置结束主键。
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(en
dPkValue));
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("GetRange的结果为：");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果NextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- 示例2

  按照第一个主键列确定范围、第二主键列从最小值（INF_MIN）到最大值（INF_MAX）进行正序读取，判断nextStartPrimaryKey是否为null，读取完范围内的全部数据。

```
private static void getRange(SyncClient client, String startPkValue, String endPkValue) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_NAME);
    //设置起始主键，以两个主键为例。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1, PrimaryKeyValue.fromString(s
tartPkValue));//确定值。
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2, PrimaryKeyValue.INF_MIN);//
最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    //设置结束主键。
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1, PrimaryKeyValue.fromString(e
ndPkValue));//确定值。
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2, PrimaryKeyValue.INF_MAX);//
最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("GetRange的结果为：");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

- 示例3

  读取主键范围为["pk:2020-01-01.log", "pk:2021-01-01.log")时Col1列的数据，并对该列的数据执行正则过滤。

```
private static void getRange(SyncClient client) {
    RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(TABLE_NAME);
    // 设置主键范围为["pk:2020-01-01.log", "pk:2021-01-01.log")，读取范围为左闭右开的区间。
    PrimaryKey pk0 = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("pk", PrimaryKeyValue.fromString("2020-01-01.log"))
        .build();
    PrimaryKey pk1 = PrimaryKeyBuilder.createPrimaryKeyBuilder()
        .addPrimaryKeyColumn("pk", PrimaryKeyValue.fromString("2021-01-01.log"))
        .build();
    criteria.setInclusiveStartPrimaryKey(pk0);
    criteria.setExclusiveEndPrimaryKey(pk1);
    // 设置读取最新版本。
    criteria.setMaxVersions(1);
    // 设置过滤器，当cast<int>(regex(Col1)) > 100时，返回该行。
    RegexRule regexRule = new RegexRule("t1:([0-9]+),", VariantType.Type.VT_INTEGER);
    SingleColumnValueRegexFilter filter =  new SingleColumnValueRegexFilter("Col1",
        regexRule,SingleColumnValueFilter.CompareOperator.GREATER_THAN,ColumnValue.fromLo
ng(100));
    criteria.setFilter(filter);
    while (true) {
        GetRangeResponse resp = client.getRange(new GetRangeRequest(criteria));
        for (Row row : resp.getRows()) {
            // do something
            System.out.println(row);
        }
        if (resp.getNextStartPrimaryKey() != null) {
            criteria.setInclusiveStartPrimaryKey(resp.getNextStartPrimaryKey())
        }
    }
}
```

详细代码请参见Get Range@Git Hub。

迭代读（createRangeIterator）

迭代读取数据。

```
private static void getRangeByIterator(SyncClient client, String startPkValue, String endPk
Value) {
    RangeIteratorParameter rangeIteratorParameter = new RangeIteratorParameter(TABLE_NAME);
    //设置起始主键。
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(star
tPkValue));
    rangeIteratorParameter.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
    //设置结束主键。
    primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString(endP
kValue));
    rangeIteratorParameter.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
    rangeIteratorParameter.setMaxVersions(1);
    Iterator<Row> iterator = client.createRangeIterator(rangeIteratorParameter);
    System.out.println("使用Iterator进行GetRange的结果为：");
    while (iterator.hasNext()) {
        Row row = iterator.next();
        System.out.println(row);
    }
}
```

详细代码请参见GetRangeByIterator@GitHub。

3.7. 增量数据操作

表格存储提供了 stream 的 list 和 describe 操作，以及 shard 的 getsharditerator 和 getshardrecord 操作。

列出所有的Stream（ListStream）

ListStream接口用于列出当前实例和表下的所有stream。

示例

列出某个表的所有 stream 信息。

```
private static void listStream(SyncClient client, String tableName) {
    ListStreamRequest listStreamRequest = new ListStreamRequest(tableName);
    ListStreamResponse result = client.listStream(listStreamRequest);
}
```

查询表Stream描述信息（DescribeStream）

DescribeStream接口可以查询 stream 的创建时间（creationTime）、过期时间（expirationTime）、当前的状态（status）、包含 shard 的列表（shards）和下一个起始 shard 的 id（如果还有尚未返回的 shard）。

示例 1

获取当前 stream 的所有 shard 信息。

```
    private static void describeStream(SyncClient client, String streamId) {
        DescribeStreamRequest desRequest = new DescribeStreamRequest(streamId);
        DescribeStreamResponse desStream = client.describeStream(desRequest);
    }
```

示例 2

设置开始 shardID（InclusiveStartShardId）和每次返回的最大 shard 数目。

```
    private static void describeStream(SyncClient client, String streamId) {
    DescribeStreamRequest dsRequest = new DescribeStreamRequest(streamId);
    dsRequest.setInclusiveStartShardId(startShardId);
    dsRequest.setShardLimit(10);
    DescribeStreamResponse dscStream = client.describeStream(dsRequest);
    }
```

获取Shard的读取迭代值（GetShardIterator）

GetShardIterator 接口用于获取 shard 的读取起始迭代值。

示例

获取 shard 的读取起始迭代值。

```
    private static void getShardIterator(SyncClient client, String streamId, String shardId
) {
    GetShardIteratorRequest getShardIteratorRequest = new GetShardIteratorRequest(streamId,
shardId);
    GetShardIteratorResponse shardIterator = client.getShardIterator(getShardIteratorReques
t);
    }
```

获取Shard的更新记录（GetStreamRecord）

GetStreamRecord 接口用于获取 shard 的每条更新记录。

示例

获取 shard 的最初 100 条更新。

```
    private static void getShardIterator(SyncClient client, String shardIterator) {
        GetStreamRecordRequest streamRecordRequest = new GetStreamRecordRequest(shardIterat
or);
        streamRecordRequest.setLimit(100);
        GetStreamRecordResponse streamRecordResponse = client.getStreamRecord(streamRecordR
equest);
        List<StreamRecord> records = streamRecordResponse.getRecords();
        for(int k=0;k<records.size();k++){
            System.out.println("record info:" +  records.get(k).toString());
        }
        System.out.println("next iterator:" + streamRecordResponse.getNextShardIterator());
    }
```

3.8. 多元索引

3.8.1. 创建多元索引

使用CreateSearchIndex接口在数据表上创建一个多元索引。一个数据表可以创建多个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。

参数

创建多元索引时，需要指定数据表名称（tableName）、多元索引名称（indexName）和索引的结构信息（indexSchema），其中indexSchema包含fieldSchemas（Index的所有字段的设置）、indexSetting（索引设置）、indexSort（索引预排序设置）和timeToLive（数据生命周期）。详细参数说明请参见下表。

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
fieldSchemas	fieldSchema的列表，每个fieldSchema包含如下内容： • fieldName（必选）：创建多元索引的字段名，即列名，类型为String。 多元索引中的字段可以是主键列或者属性列。 • fieldType（必选）：字段类型，类型为FieldType.XXX。更多信息，请参见字段。 • array（可选）：是否为数组，类型为Boolean。 如果设置为true，则表示该列是一个数组，在写入时，必须按照JSON数组格式写入，例如["a","b","c"]。 由于Nested类型是一个数组，当fieldType为Nested类型时，无需设置此参数。 • index（可选）：是否开启索引，类型为Boolean。 默认为true，表示对该列构建倒排索引或者空间索引；如果设置为false，则不会对该列构建索引。 • analyzer（可选）：分词器类型。当字段类型为Text时，可以设置此参数；如果不设置，则默认分词器类型为单字分词。关于分词的更多信息，请参见分词。 • enableSortAndAgg（可选）：是否开启排序与统计聚合功能，类型为Boolean。 只有enableSortAndAgg设置为true的字段才能进行排序。关于排序的更多信息，请参见排序和翻页。 注意 Nested类型的字段不支持开启排序与统计聚合功能，但是Nested类型内部的子列支持开启排序与统计聚合功能。 • store（可选）：是否在多元索引中附加存储该字段的值，类型为Boolean。 开启后，可以直接从多元索引中读取该字段的值，而不必反查数据表，可用于查询性能优化。 • isVirtualField（可选）：该字段是否为虚拟列，类型为Boolean类型，默认值为false。只有使用虚拟列时，才需要设置此参数。关于虚拟列的更多信息，请参见虚拟列。 • sourceFieldName（可选）：数据表中的字段名称，类型为String。当设置isVirtualField为true时，必须设置此参数。
indexSetting	索引设置，包含routingFields设置。 routingFields（可选）：自定义路由字段。可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

参数	说明
indexSort	索引预排序设置，包含sorters设置。如果不设置，则默认按照主键排序。 **说明** 含有Nested类型的索引不支持indexSort，没有预排序。 sorters（必选）：索引的预排序方式，支持按照主键排序和字段值排序。关于排序的更多信息，请参见排序和翻页。 • PrimaryKeySort表示按照主键排序，包含如下设置： order：排序的顺序，可按升序或者降序排序，默认为升序（SortOrder.ASC）。 • FieldSort表示按照字段值排序，包含如下设置： 只有建立索引且开启排序与统计聚合功能的字段才能进行预排序。 ○ fieldName：排序的字段名。 ○ order：排序的顺序，可按照升序或者降序排序，默认为升序（SortOrder.ASC）。 ○ mode：当字段存在多个值时的排序方式。
timeToLive	可选参数，默认值为-1。数据生命周期（TTL），即数据的保存时间。 当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。

示例

- 创建多元索引。

  创建一个多元索引，包含Col_Keyword和Col_Long两列，类型分别设置为字符串（String）和整型（Long）。

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); //设置数据表名称。
    request.setIndexName(indexName); //设置多元索引名称。
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD) //设置字段名和类型。
                    .setIndex(true) //设置开启索引。
                    .setEnableSortAndAgg(true) //设置开启排序与统计聚合功能。
                    .setStore(true), //设置在多元索引中附加存储该列的值。
            new FieldSchema("Col_Long", FieldType.LONG)
                    .setIndex(true)
                    .setEnableSortAndAgg(true)
                    .setStore(true)));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); //调用client创建多元索引。
}
```

- 创建多元索引时指定IndexSort。

```
private static void createSearchIndexWithIndexSort(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName);
    request.setIndexName(indexName);
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD).setIndex(true).setEnableSor
tAndAgg(true).setStore(true),
            new FieldSchema("Col_Long", FieldType.LONG).setIndex(true).setEnableSortAndAg
g(true).setStore(true),
            new FieldSchema("Col_Text", FieldType.TEXT).setIndex(true).setStore(true),
            new FieldSchema("Timestamp", FieldType.LONG).setIndex(true).setEnableSortAndA
gg(true).setStore(true)));
    //设置按照Timestamp列进行预排序，Timestamp列必须建立索引且开启enableSortAndAgg。
    indexSchema.setIndexSort(new Sort(
            Arrays.<Sort.Sorter>asList(new FieldSort("Timestamp", SortOrder.ASC))));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request);
}
```

- 创建多元索引时设置生命周期

> **注意** 请确保数据表的更新状态为禁止。

```
// 请使用5.12.0及以上版本的Java SDK。
public void createIndexWithTTL() {
    int days = 7;
    CreateSearchIndexRequest createRequest = new CreateSearchIndexRequest();
    createRequest.setTableName(tableName);
    createRequest.setIndexName(indexName);
    createRequest.setIndexSchema(indexSchema);
    // 设置多元索引TTL。
    createRequest.setTimeToLiveInDays(days);
    client.createSearchIndex(createRequest);
}
```

- 创建多元索引时指定虚拟列。

  创建一个多元索引，多元索引包含Col_Keyword和Col_Long两列，同时创建虚拟列Col_Keyword_Virtual_Long和Col_Long_Virtual_Keyword。Col_Keyword_Virtual_Long映射为数据表中Col_Keyword列，虚拟列Col_Long_Virtual_Keyword映射为数据表中Col_Long列。

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); //设置数据表名称。
    request.setIndexName(indexName); //设置多元索引名称。
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
        new FieldSchema("Col_Keyword", FieldType.KEYWORD) //设置字段名和类型。
            .setIndex(true) //设置开启索引。
            .setEnableSortAndAgg(true) //设置开启排序和统计功能。
            .setStore(true),
        new FieldSchema("Col_Keyword_Virtual_Long", FieldType.LONG) //设置字段名和类型。
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true) //设置字段是否为虚拟列。
            .setSourceFieldName("Col_Keyword"), //虚拟列对应的数据表中字段。
        new FieldSchema("Col_Long", FieldType.LONG)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true),
        new FieldSchema("Col_Long_Virtual_Keyword", FieldType.KEYWORD)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true)
            .setSourceFieldName("Col_Long")));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); //调用client创建多元索引。
}
```

3.8.2. 列出多元索引列表

创建多元索引后，使用ListSearchIndex接口可以获取当前实例下或某个数据表关联的所有多元索引的列表信息。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称，可以为空。 • 如果设置了数据表名称，则返回该数据表关联的所有多元索引的列表。 • 如果未设置数据表名称，则返回当前实例下所有多元索引的列表。

示例

```
private static List<SearchIndexInfo> listSearchIndex(SyncClient client) {
    ListSearchIndexRequest request = new ListSearchIndexRequest();
    request.setTableName(TABLE_NAME); //设置数据表名称。
    return client.listSearchIndex(request).getIndexInfos(); //获取数据表关联的所有多元索引。
}
```

3.8.3. 查询多元索引描述信息

创建多元索引后，使用DescribeSearchIndex接口可以查询多元索引的描述信息，包括多元索引的字段信息和索引配置等。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

示例

```
private static DescribeSearchIndexResponse describeSearchIndex(SyncClient client) {
    DescribeSearchIndexRequest request = new DescribeSearchIndexRequest();
    request.setTableName(TABLE_NAME); //设置数据表名称。
    request.setIndexName(INDEX_NAME); //设置多元索引名称。
    DescribeSearchIndexResponse response = client.describeSearchIndex(request);
    System.out.println(response.jsonize()); //打印response的详细信息。
    System.out.println(response.getSyncStat().getSyncPhase().name());//打印多元索引数据同步状态。
    return response;
}
```

3.8.4. 删除多元索引

使用DeleteSearchIndex接口可以删除指定数据表的一个多元索引。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

示例

```
private static void deleteSearchIndex(SyncClient client) {
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest();
    request.setTableName(TABLE_NAME); //设置数据表名称。
    request.setIndexName(INDEX_NAME); //设置多元索引名称。
    client.deleteSearchIndex(request); //调用client删除多元索引。
}
```

3.8.5. 精确查询

TermQuery采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。对于Text类型字段，只要分词后有词条可以精确匹配即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
query	设置查询类型为TermQuery。
fieldName	要匹配的字段。
term	查询关键词，即要匹配的值。 该词不会被分词，会被当做完整词去匹配。 对于Text类型字段，只要分词后有词条可以精确匹配即可。例如某个Text类型的字段，值为“tablestore is cool”，如果分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"的数据。
 * @param client
 */
private static void termQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermQuery termQuery = new TermQuery(); //设置查询类型为TermQuery。
    termQuery.setFieldName("Col_Keyword"); //设置要匹配的字段。
    termQuery.setTerm(ColumnValue.fromString("hangzhou")); //设置要匹配的值。
    searchQuery.setQuery(termQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.6. 多词精确查询

类似于TermQuery，但是TermsQuery可以指定多个查询关键词，查询匹配这些词的数据。多个查询关键词中只要有一个词精确匹配，该行数据就会被返回，等价于SQL中的In。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
query	设置查询类型为TermsQuery。
fieldName	要匹配的字段。
terms	多个查询关键词，即要匹配的值。最多支持设置1024个查询关键字。 多个查询关键词中只要有一个词精确匹配，该行数据就会被返回。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"或"xi'an"的数据。
 * @param client
 */
private static void termQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermsQuery termsQuery = new TermsQuery(); //设置查询类型为TermsQuery。
    termsQuery.setFieldName("Col_Keyword"); //设置要匹配的字段。
    termsQuery.addTerm(ColumnValue.fromString("hangzhou")); //设置要匹配的值。
    termsQuery.addTerm(ColumnValue.fromString("xi'an")); //设置要匹配的值。
    searchQuery.setQuery(termsQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.7. 全匹配查询

MatchAllQuery可以匹配所有行，常用于查询表中数据总行数，或者随机返回几条数据。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
query	设置查询类型为MatchAllQuery。
tableName	数据表名称。
indexName	多元索引名称。
limit	本次查询需要返回的最大数量。 如果要随机获取几行数据，请设置limit为正整数。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
columnsToGet	是否返回所有列。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。

示例

```
/**
 * 通过MatchAllQuery查询表中数据的总行数。
 * @param client
 */
private static void matchAllQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    /**
    * 设置查询类型为MatchAllQuery。
    */
    searchQuery.setQuery(new MatchAllQuery());
    /**
     * MatchAllQuery结果中的TotalCount可以表示表中数据的总行数。
     * 如果要随机获取几行数据，请设置limit为正整数。
     * 如果只为了获取行数，但不需要具体数据，可以设置limit=0，即不返回任意一行数据。
     */
    searchQuery.setLimit(0);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME, searchQuery);
    /**
    * 设置返回匹配的总行数。
    */
    searchQuery.setGetTotalCount(true);
    SearchResponse resp = client.search(searchRequest);
    /**
     * 判断返回的结果是否完整，当isAllSuccess为false时，表示可能存在部分节点查询失败，返回的是部分数
据。
     */
    if (!resp.isAllSuccess()) {
        System.out.println("NotAllSuccess!");
    }
    System.out.println("IsAllSuccess: " + resp.isAllSuccess());
    System.out.println("TotalCount: " + resp.getTotalCount()); //打印总行数。
    System.out.println(resp.getRequestId());
}
```

3.8.8. 匹配查询

MatchQuery采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用MatchPhraseQuery实现高性能的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
fieldName	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
query	设置查询类型为matchQuery。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
minimumShouldMatch	最小匹配个数。 只有当某一行数据的fieldName列的值中至少包括最小匹配个数的词时，才会返回该行数据。 说明 minimumShouldMatch需要与逻辑运算符OR配合使用。
operator	逻辑运算符。默认为OR，表示当分词后的多个词只要有部分匹配时，则行数据满足查询条件。 如果设置operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列的值能够匹配"hangzhou"的数据，返回匹配到的总行数和一些匹配成功的行。
 * @param client
 */
private static void matchQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    MatchQuery matchQuery = new MatchQuery(); //设置查询类型为MatchQuery。
    matchQuery.setFieldName("Col_Keyword"); //设置要匹配的列。
    matchQuery.setText("hangzhou"); //设置要匹配的值。
    searchQuery.setQuery(matchQuery);
    searchQuery.setOffset(0); //设置offset为0。
    searchQuery.setLimit(20); //设置limit为20，表示最多返回20行数据。
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.9. 短语匹配查询

类似于MatchQuery，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。如果查询列的分词类型为模糊分词，则使用MatchPhraseQuery可以实现比WildcardQuery更快的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
fieldName	要匹配的列。 匹配查询可应用于Text类型。

参数	说明
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“...，this is tablestore”、“this is a table”，但是无法匹配到“this table is ...”以及“is this a table”。
query	设置查询类型为matchPhraseQuery。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Text列的值能够匹配"hangzhou shanghai"的数据，匹配条件为短语匹配（要求短语完整的按照顺序匹配），返回匹配到的总行数和一些匹配成功的行。
 * @param client
 */
private static void matchPhraseQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    MatchPhraseQuery matchPhraseQuery = new MatchPhraseQuery(); //设置查询类型为MatchPhraseQuery。
    matchPhraseQuery.setFieldName("Col_Text"); //设置要匹配的列。
    matchPhraseQuery.setText("hangzhou shanghai"); //设置要匹配的值。
    searchQuery.setQuery(matchPhraseQuery);
    searchQuery.setOffset(0); //设置offset为0。
    searchQuery.setLimit(20); //设置limit为20，表示最多返回20行数据。
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.10. 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
query	设置查询类型为PrefixQuery。
fieldName	要匹配的字段。
prefix	前缀值。 对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

参数	说明
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列中前缀为"hangzhou"的数据。
 * @param client
 */
private static void prefixQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    PrefixQuery prefixQuery = new PrefixQuery(); //设置查询类型为PrefixQuery。
    searchQuery.setGetTotalCount(true);
    prefixQuery.setFieldName("Col_Keyword");
    prefixQuery.setPrefix("hangzhou");
    searchQuery.setQuery(prefixQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
```

3.8.11. 范围查询

RangeQuery根据范围条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足范围条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。

- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
fieldName	要匹配的字段。
from	起始位置的值。 设置范围条件时，大于（>）可以使用greaterThan表示，大于等于（>=）可以使用greaterThanOrEqual表示。
to	结束位置的值。 设置范围条件时，小于（<）可以使用lessThan表示；小于等于（<=）可以使用lessThanOrEqual表示。
includeLower	结果中是否需要包括from值，类型为Boolean。
includeUpper	结果中是否需要包括to值，类型为Boolean。
query	设置查询类型为RangeQuery。
sort	按照指定方式排序，详情请参见排序和翻页。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Long列大于3的数据，结果按照Col_Long列的值逆序排序。
 * @param client
 */
private static void rangeQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    RangeQuery rangeQuery = new RangeQuery(); //设置查询类型为RangeQuery。
    rangeQuery.setFieldName("Col_Long");  //设置要匹配的字段。
    rangeQuery.greaterThan(ColumnValue.fromLong(3));  //设置该字段的范围条件为大于3。
    searchQuery.setGetTotalCount(true);
    searchQuery.setQuery(rangeQuery);
    //设置按照Col_Long列逆序排序。
    FieldSort fieldSort = new FieldSort("Col_Long");
    fieldSort.setOrder(SortOrder.DESC);
    searchQuery.setSort(new Sort(Arrays.asList((Sort.Sorter)fieldSort)));
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.12. 通配符查询

通配符查询中，要匹配的值可以是一个带有通配符的字符串，目前支持星号（*）和问号（?）两种通配符。要匹配的值中可以用星号（*）代表任意字符序列，或者用问号（?）代表任意单个字符，且支持以星号（*）或问号（?）开头。例如查询“table*e”，可以匹配到“tablestore”。

如果查询的模式为*word*，则您可以使用性能更好的模糊查询，具体实现方法如下：

1. 创建多元索引时，设置列为Text类型且设置分词类型为模糊分词。
2. 使用多元索引查询数据时，使用MatchPhraseQuery且设置查询词为word。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	描述
fieldName	列名称。

参数	描述
value	带有通配符的字符串，字符串长度不能超过20个字符。
query	设置查询类型为WildcardQuery。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 使用通配符查询，查询表中Col_Keyword列的值中匹配"hang*u"的数据。
 * @param client
 */
private static void wildcardQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    WildcardQuery wildcardQuery = new WildcardQuery(); //设置查询类型为WildcardQuery。
    wildcardQuery.setFieldName("Col_Keyword");
    wildcardQuery.setValue("hang*u"); //wildcardQuery支持通配符。
    searchQuery.setQuery(wildcardQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.13. 地理位置查询

地理位置查询包括地理距离查询（GeoDistanceQuery）、地理长方形范围查询（GeoBoundingBoxQuery）和地理多边形范围查询（GeoPolygonQuery）三种方式。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

地理距离查询（GeoDistanceQuery）

GeoDistanceQuery根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，满足查询条件。

- 参数

参数	说明
fieldName	列名，类型为Geopoint。
centerPoint	中心地理坐标点，是一个经纬度值。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围为[-180,+180]。例如 `35.8,-45.91` 。
distanceInMeter	距离中心点的距离。类型为Double。单位为米。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
query	多元索引的查询语句。设置查询类型为GeoDistanceQuery。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

- 示例

  查询表中Col_GeoPoint列的值距离中心点不超过一定距离的数据。

```
public static void geoDistanceQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoDistanceQuery geoDistanceQuery = new GeoDistanceQuery();  //设置查询类型为GeoDistanceQuery。
    geoDistanceQuery.setFieldName("Col_GeoPoint");
    geoDistanceQuery.setCenterPoint("5,5"); //设置中心点。
    geoDistanceQuery.setDistanceInMeter(10000); //设置条件为到中心点的距离不超过10000米。
    searchQuery.setQuery(geoDistanceQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行数。
    System.out.println("Row: " + resp.getRows());
}
```

地理长方形范围查询（GeoBoundingBoxQuery）

GeoBoundingBoxQuery根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时满足查询条件。

- 参数

参数	说明
fieldName	列名，类型为Geopoint。
topLeft	长方形框的左上角的坐标。
bottomRight	长方形框的右下角的坐标，通过左上角和右下角就可以确定一个唯一的长方形。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。
query	多元索引的查询语句。设置查询类型为GeoBoundingBoxQuery。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

- 示例

  Col_Geopoint是GeoPoint类型，查询表中Col_GeoPoint列的值在左上角为"10,0", 右下角为"0,10"的长方形范围内的数据。

```
public static void geoBoundingBoxQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoBoundingBoxQuery geoBoundingBoxQuery = new GeoBoundingBoxQuery(); //设置查询类型为GeoBoundingBoxQuery。
    geoBoundingBoxQuery.setFieldName("Col_GeoPoint"); //设置列名。
    geoBoundingBoxQuery.setTopLeft("10,0"); //设置长方形左上角。
    geoBoundingBoxQuery.setBottomRight("0,10"); //设置长方形右下角。
    searchQuery.setQuery(geoBoundingBoxQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", searchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行数。
    System.out.println("Row: " + resp.getRows());
}
```

地理多边形范围查询（GeoPolygonQuery）

GeoPolygonQuery根据一个多边形范围条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时满足查询条件。

- 参数

参数	说明
fieldName	列名，类型为Geopoint。
points	组成多边形的距离坐标点。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围为[-180,+180]。例如 `35.8,-45.91` 。
query	多元索引的查询语句。设置查询类型为GeoPolygonQuery。

参数	说明
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

- 示例

  查询表中Col_GeoPoint列的值在一个给定多边形范围内的数据。

```
public static void geoPolygonQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoPolygonQuery geoPolygonQuery = new GeoPolygonQuery();  //设置查询类型为GeoPolygonQue
ry。
    geoPolygonQuery.setFieldName("Col_GeoPoint");
    geoPolygonQuery.setPoints(Arrays.asList("0,0","5,5","5,0")); //设置多边形的顶点。
    searchQuery.setQuery(geoPolygonQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返
回行数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.14. 多条件组合查询

BoolQuery查询条件包含一个或者多个子查询条件，根据子查询条件来判断一行数据是否满足查询条件。每个子查询条件可以是任意一种Query类型，包括BoolQuery。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
mustQueries	多个Query列表，行数据必须满足所有的子查询条件才算匹配，等价于And操作符。
mustNotQueries	多个Query列表，行数据必须不能满足任何的子查询条件才算匹配，等价于Not操作符。
filterQueries	多个Query列表，行数据必须满足所有的子filter才算匹配，filter类似于query，区别是filter不会根据满足的filterQueries个数进行相关性算分。
shouldQueries	多个Query列表，可以满足，也可以不满足，等价于Or操作符。 一行数据应该至少满足shouldQueries子查询条件的最小匹配个数才算匹配。 如果满足的shouldQueries子查询条件个数越多，则整体的相关性分数更高。
minimumShouldMatch	shouldQueries子查询条件的最小匹配个数。当同级没有其他Query，只有shouldQueries时，默认值为1；当同级已有其他Query，例如mustQueries、mustNotQueries和filterQueries时，默认值为0。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

- 示例1

  通过BoolQuery进行And条件查询。

```
/**
 * 通过BoolQuery进行And条件查询。
 * @param client
 */
public static void andQuery(SyncClient client){
    /**
     * 查询条件一：RangeQuery，Col_Long的列值大于3。
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));
    /**
     * 查询条件二：MatchQuery，Col_Keyword的列值要匹配"hangzhou"。
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * 构造一个BoolQuery，设置查询条件为必须同时满足"查询条件一"和"查询条件二"。
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustQueries(Arrays.asList(rangeQuery, matchQuery));
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键
列。
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); //设置为返回所有列。
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列
。
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse resp = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，
非返回行数。
        System.out.println("Row: " + resp.getRows());
    }
}
```

- 示例2

  通过BoolQuery进行Or条件查询。

```
/**
 * 通过BoolQuery进行Or条件查询。
 * @param client
 */
public static void orQuery(SyncClient client) {
    /**
     * 查询条件一：RangeQuery，Col_Long的列值大于3。
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));
    /**
     * 查询条件二：MatchQuery，Col_Keyword的列值要匹配"hangzhou"。
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
    /**
     * 构造一个BoolQuery，设置查询条件为至少满足"条件一"和"条件二"中的一个条件。
     */
    BoolQuery boolQuery = new BoolQuery();
    boolQuery.setShouldQueries(Arrays.asList(rangeQuery, matchQuery));
    boolQuery.setMinimumShouldMatch(1); //设置至少满足一个条件。
    searchQuery.setQuery(boolQuery);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返
回行数
    System.out.println("Row: " + resp.getRows());
    }
}
```

- 示例3

  通过BoolQuery进行Not条件查询。

```
/**
 * 通过BoolQuery进行Not条件查询。
 * @param client
 */
public static void notQuery(SyncClient client) {
    /**
     * 查询条件一：MatchQuery，Col_Keyword的列值要匹配"hangzhou"。
     */
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("Col_Keyword");
    matchQuery.setText("hangzhou");
    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * 构造一个BoolQuery，设置查询条件为不满足"查询条件一"。
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustNotQueries(Arrays.asList(matchQuery));
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键
列。
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); //设置为返回所有列。
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列
。
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse resp = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，
非返回行数。
        System.out.println("Row: " + resp.getRows());
    }
}
```

- 示例4

  含有子BoolQuery的组合查询。使用多条件组合查询实现(col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6))，每个and或or相当于一个BoolQuery，多个表达式的and或or就是多个BoolQuery的组合。

```
/**
 * (col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6))
 * 使用多条件组合查询上述表达式，每个and或or相当于一个BoolQuery，多个表达式的and或or就是多个BoolQu
ery的组合。
 * @param client
*/
private static void boolQuery2(SyncClient client){
        //条件1为col2<4。
        RangeQuery rangeQuery1 = new RangeQuery();
        rangeQuery1.setFieldName("col2");
        rangeQuery1.lessThan(ColumnValue.fromLong(4));
        //条件2为col3<5。
        RangeQuery rangeQuery2 = new RangeQuery();
        rangeQuery2.setFieldName("col3");
```

```
        rangeQuery2.lessThan(ColumnValue.fromLong(5));
        //条件3为col2=4。
        TermQuery termQuery = new TermQuery();
        termQuery.setFieldName("col2");
        termQuery.setTerm(ColumnValue.fromLong(4));
        //条件4为col3 = 5 or col3 = 6。
        TermsQuery termsQuery = new TermsQuery();
        termsQuery.setFieldName("col3");
        termsQuery.addTerm(ColumnValue.fromLong(5));
        termsQuery.addTerm(ColumnValue.fromLong(6));
        SearchQuery searchQuery = new SearchQuery();
        List<Query> queryList1 = new ArrayList<>();
        queryList1.add(rangeQuery1);
        queryList1.add(rangeQuery2);
        //组合条件1为col2<4 OR col3<5。
        BoolQuery boolQuery1 = new BoolQuery();
        boolQuery1.setShouldQueries(queryList1);
        //组合条件2为col2=4 and (col3=5 or col3=6)。
        List<Query> queryList2 = new ArrayList<>();
        queryList2.add(termQuery);
        queryList2.add(termsQuery);
        BoolQuery boolQuery2 = new BoolQuery();
        boolQuery2.setMustQueries(queryList2);
        //总组合条件为(col2<4 OR col3<5) or (col2=4 and (col3=5 or col3=6))。
        List<Query> queryList3 = new ArrayList<>();
        queryList3.add(boolQuery1);
        queryList3.add(boolQuery2);
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setShouldQueries(queryList3);
        searchQuery.setQuery(boolQuery);
        //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
        SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex
", searchQuery);
        //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键
列。
        //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
        //columnsToGet.setReturnAll(true); //设置为返回所有列。
        //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列
。
        //searchRequest.setColumnsToGet(columnsToGet);
        SearchResponse response = client.search(searchRequest);
        //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，
非返回行数。
        System.out.println(response.getRows());
    }
```

3.8.15. 嵌套类型查询

NestedQuery用于查询嵌套类型字段中子行的数据。嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型字段的路径和一个子查询，其中子查询可以是任意Query类型。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
path	路径名，嵌套类型字段的树状路径。例如news.title表示嵌套类型的news字段中的title子列。
query	嵌套类型字段的子列上的查询，子列上的查询可以是任意Query类型。
scoreMode	当字段存在多个值时基于哪个值计算分数。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

使用嵌套类型查询的示例如下：

- 示例1

  查询col_nested.nested_1为tablestore的数据。其中col_nested为嵌套类型字段，子行中包含nested_1和nested_2两列。

```
private static void nestedQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    NestedQuery nestedQuery = new NestedQuery(); //设置查询类型为NestedQuery。
    nestedQuery.setPath("col_nested"); //设置嵌套类型列的路径。
    TermQuery termQuery = new TermQuery(); //构造NestedQuery的子查询。
    termQuery.setFieldName("col_nested.nested_1"); //设置列名，注意带有嵌套类型列的路径。
    termQuery.setTerm(ColumnValue.fromString("tablestore")); //设置要查询的值。
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返
回行数。
    System.out.println("Row: " + resp.getRows());
}
```

- 示例2

  查询col_nested.nested_2.nested_2_2为tablestore的数据。其中col_nested为嵌套类型字段，col_nested的子行中包含nested_1和nested_2两列，nested_2的子行中又包含nested_2_1和nested_2_2两列。

```
private static void nestedQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    NestedQuery nestedQuery = new NestedQuery(); //设置查询类型为NestedQuery。
    nestedQuery.setPath("col_nested.nested_2"); //设置嵌套类型列的路径，即要查询字段的父路径。
    TermQuery termQuery = new TermQuery(); //构造NestedQuery的子查询。
    termQuery.setFieldName("col_nested.nested_2.nested_2_2"); //设置列名，即要查询字段的完整
路径。
    termQuery.setTerm(ColumnValue.fromString("tablestore")); //设置要查询的值。
    nestedQuery.setQuery(termQuery);
    nestedQuery.setScoreMode(ScoreMode.None);
    searchQuery.setQuery(nestedQuery);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返
回行数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.16. 排序和翻页

您可以在创建多元索引时指定索引预排序和在使用多元索引查询数据时指定排序方式，以及在获取返回结果时使用limit和offset或者使用token进行翻页。

索引预排序

多元索引默认按照设置的索引预排序（IndexSort）方式进行排序，使用多元索引查询数据时，IndexSort决定了数据的默认返回顺序。

在创建多元索引时，您可以自定义IndexSort，如果未自定义IndexSort，则IndexSort默认为主键排序。

> **注意** 含有Nested类型字段的多元索引不支持索引预排序。

查询时指定排序方式

只有enableSortAndAgg设置为true的字段才能进行排序。

在每次查询时，可以指定排序方式，多元索引支持如下四种排序方式（Sorter）。您也可以使用多个Sorter，实现先按照某种方式排序，再按照另一种方式排序的需求。

- ScoreSort

  按照查询结果的相关性（BM25算法）分数进行排序，适用于有相关性的场景，例如全文检索等。

  > **注意** 如果需要按照相关性打分进行排序，必须手动设置ScoreSort，否则会按照索引设置的IndexSort进行排序。

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(new ScoreSort())));
  ```

- PrimaryKeySort

  按照主键进行排序。

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(new PrimaryKeySort()))); //正序。
  //searchQuery.setSort(new Sort(Arrays.asList(new PrimaryKeySort(SortOrder.DESC)))); //逆序。
  ```

- FieldSort

  按照某列的值进行排序。

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(new FieldSort("col", SortOrder.ASC))));
  ```

  先按照某列的值进行排序，再按照另一列的值进行排序。

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setSort(new Sort(Arrays.asList(
      new FieldSort("col1", SortOrder.ASC), new FieldSort("col2", SortOrder.ASC))));
  ```

- GeoDistanceSort

  根据地理点距离进行排序。

```
SearchQuery searchQuery = new SearchQuery();
//geo列为Geopoint类型，按照此列的值距离"0,0"点的距离进行排序。
Sort.Sorter sorter = new GeoDistanceSort("geo", Arrays.asList("0, 0"));
searchQuery.setSort(new Sort(Arrays.asList(sorter)));
```

翻页方式

在获取返回结果时，可以使用limit和offset或者使用token进行翻页。

- 使用limit和offset进行翻页

  当需要获取的返回结果行数小于10000行时，可以使用limit和offset进行翻页，即limit+offset<=10000，其中limit的最大值为100。

  > **说明** 如果需要提高limit的上限，请参见如何将多元索引Search接口查询数据的limit提高到1000。

  如果使用此方式进行翻页时未设置limit和offset，则limit的默认值为10，offset的默认值为0。

  ```
  SearchQuery searchQuery = new SearchQuery();
  searchQuery.setQuery(new MatchAllQuery());
  searchQuery.setLimit(100);
  searchQuery.setOffset(100);
  ```

- 使用token进行翻页

  由于使用token进行翻页时翻页深度无限制，当需要进行深度翻页时，推荐使用token进行翻页。

  当符合查询条件的数据未读取完时，服务端会返回nextToken，此时可以使用nextToken继续读取后面的数据。

  使用token进行翻页时默认只能向后翻页。由于在一次查询的翻页过程中token长期有效，您可以通过缓存并使用之前的token实现向前翻页。

  > **注意** 如果需要持久化nexToken或者传输nexToken给前端页面，您可以使用Base64编码将nextToken编码为String进行保存和传输。token本身不是字符串，直接使用 `new String(nextToken)` 将token编码为String会造成token信息丢失。

  使用token翻页后的排序方式和上一次请求的一致，无论是系统默认使用IndexSort还是自定义排序，因此设置了token不能再设置Sort。另外使用token后不能设置offset，只能依次往后读取，即无法跳页。

  > **注意** 由于含有Nested类型字段的多元索引不支持索引预排序，如果使用含有Nested类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回nextToken。

```
private static void readMoreRowsWithToken(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    searchQuery.setQuery(new MatchAllQuery());
    searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", s
earchQuery);
    SearchResponse resp = client.search(searchRequest);
    if (!resp.isAllSuccess()) {
        throw new RuntimeException("not all success");
    }
    List<Row> rows = resp.getRows();
    while (resp.getNextToken()!=null) { // 当读取到nextToken为null时，表示读出全部数据。
        // 获取nextToken。
        byte[] nextToken = resp.getNextToken();
        {
            // 如果需要持久化nexToken或者传输nexToken给前端页面，您可以使用Base64编码将nexToken
编码为String进行保存和传输。
            // token本身不是字符串，直接使用new String(nextToken)将token编码为String会造成toke
n信息丢失。
            String tokenAsString = Base64.toBase64String(nextToken);
            // 将String解码为byte。
            byte[] tokenAsByte = Base64.fromBase64String(tokenAsString);
        }
        // 将token设置到下一次请求中。
        searchRequest.getSearchQuery().setToken(nextToken);
        resp = client.search(searchRequest);
        if (!resp.isAllSuccess()) {
            throw new RuntimeException("not all success");
        }
        rows.addAll(resp.getRows());
    }
    System.out.println("RowSize: " + rows.size());
    System.out.println("TotalCount: " + resp.getTotalCount());// 打印匹配到的总行数，非返回行
数。
}
```

3.8.17. 列存在性查询

ExistsQuery也叫NULL查询或者空值查询，一般用于判断稀疏数据中某一行的某一列是否存在。例如查询所有数据中address列不为空的行。

> **说明**
>
> - 如果需要查询某一列为空，则ExistsQuery需要和BoolQuery中的mustNotQueries结合使用。
> - 以下情况会认为某一列不存在，以city列为例说明。
>   - city列在本行数据中不存在。
>   - city列是空数组，即"city" = "[]"。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。

- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
fieldName	列名。
query	设置查询类型为ExistsQuery。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
/**
 * 使用列存在查询，查询表中address列的值不为空的数据。
 * @param client
 */
private static void existQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    ExistsQuery existQuery = new ExistsQuery(); //设置查询类型为ExistsQuery。
    existQuery.setFieldName("address");
    searchQuery.setQuery(existQuery);
    //searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);
    //通过设置columnsToGet参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    //System.out.println("TotalCount: " + resp.getTotalCount()); //打印匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

3.8.18. 折叠（去重）

当数据查询的结果中含有某种类型的数据较多时，可以使用折叠（Collapse）功能按照某一列对结果集做折叠，使对应类型的数据在结果展示中只出现一次，保证结果展示中类型的多样性。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

注意事项

- 折叠功能只能使用offset+limit方式翻页，不能使用token方式。
- 对结果集同时使用统计聚合与折叠功能时，统计聚合功能只作用于使用折叠功能前的结果集。
- 使用折叠功能后，返回的总分组数取决于offset+limit的最大值，目前支持返回的总分组数最大为10000。
- 执行结果中返回的总行数是使用折叠功能前的匹配行数，使用折叠功能后的总分组数无法获取。

参数

参数	说明
query	可以是任意Query类型。
collapse	折叠参数设置，包含fieldName设置。 fieldName：列名，按该列对结果集做折叠，只支持应用于整型、浮点数和Keyword类型的列，不支持数组类型的列。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
tableName	数据表名称。
indexName	多元索引名称。
columnsToGet	是否返回所有列，包含returnAll和columns设置。 returnAll默认为false，表示不返回所有列，此时可以通过columns指定返回的列；如果未通过columns指定返回的列，则只返回主键列。 当设置returnAll为true时，表示返回所有列。

示例

```
private static void UseCollapse(SyncClient client){
    SearchQuery searchQuery = new SearchQuery(); //构造查询条件。
    MatchQuery matchQuery = new MatchQuery();
    matchQuery.setFieldName("user_id");
    matchQuery.setText("00002");
    searchQuery.setQuery(matchQuery);
    Collapse collapse = new Collapse("product_name"); //根据"product_name"列对结果集做折叠。
    searchQuery.setCollapse(collapse);
    searchQuery.setOffset(1000);
    searchQuery.setLimit(20);
    //searchQuery.setGetTotalCount(true);//设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("sampleTable", "sampleSearchIndex", sea
rchQuery);//设置数据表名称和多元索引名称。      //通过设置columnsToGet参数可以指定返回的列或返回所有列
，如果不设置此参数，则默认只返回主键列。
    //SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    //columnsToGet.setReturnAll(true); //设置为返回所有列。
    //columnsToGet.setColumns(Arrays.asList("ColName1","ColName2")); //设置为返回指定列。
    //searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse response = client.search(searchRequest);
    //System.out.println(response.getTotalCount());
    //System.out.println(response.getRows().size()); //根据"product_name"列的产品种类返回个数
。
    System.out.println(response.getRows()); //根据"product_name"列的产品种类返回相应产品名称。
}
```

3.8.19. 统计聚合

使用统计聚合功能可以实现求最小值、求最大值、求和、求平均值、统计行数、去重统计行数、百分位统计、按字段值分组、按范围分组、按地理位置分组、按过滤条件分组、直方图统计、获取统计聚合分组内的行、嵌套查询等；同时多个统计聚合功能可以组合使用，满足复杂的查询需求。

最小值

返回一个字段中的最小值，类似于SQL中的min。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 *  商品库中有每一种商品的价格，求产地为浙江省的商品中，价格最低的商品价格是多少。
 *  等效的SQL语句是SELECT min(column_price) FROM product where place_of_production="浙江省"
。
 */
public void min(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production","浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性
能。
                .addAggregation(AggregationBuilders.min("min_agg_1", "column_price").miss
ing(100))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("min_agg_1"
).getValue());
}
```

最大值

返回一个字段中的最大值，类似于SQL中的max。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的价格，求产地为浙江省的商品中，价格最高的商品价格是多少。
 * 等效的SQL语句是SELECT max(column_price) FROM product where place_of_production="浙江省"。
 */
public void max(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
                .addAggregation(AggregationBuilders.max("max_agg_1", "column_price").missing(0))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMaxAggregationResult("max_agg_1").getValue());
}
```

和

返回数值字段的总数，类似于SQL中的sum。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的售出数量，求产地为浙江省的商品中，一共售出了多少件商品。如果某一件商品没有该值，默认售出了10件。
 * 等效的SQL语句是SELECT sum(column_price) FROM product where place_of_production="浙江省"。
 */
public void sum(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
                .addAggregation(AggregationBuilders.sum("sum_agg_1", "column_number").missing(10))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("sum_agg_1").getValue());
}
```

平均值

返回数值字段的平均值，类似于SQL中的avg。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的售出数量，求产地为浙江省的商品中，平均价格是多少。
 * 等效的SQL语句是SELECT avg(column_price) FROM product where place_of_production="浙江省"。
 */
public void avg(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place_of_production", "浙江省"))
                .limit(0) //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
                .addAggregation(AggregationBuilders.avg("avg_agg_1", "column_number"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsAvgAggregationResult("avg_agg_1").getValue());
}
```

统计行数

返回指定字段值的数量或者表中数据总行数，类似于SQL中的count。

> **说明** 通过如下方式可以统计表中数据总行数或者某个query匹配的行数。
>
> - 使用统计聚合的count功能，在请求中设置count(*)。
> - 使用query功能的匹配行数，在query中设置setGetTotalCount(true)；如果需要统计表中数据总行数，则使用MatchAllQuery。
>
> 如果需要获取表中数据某列出现的次数，则使用count（列名），可应用于稀疏列的场景。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。

- 示例

```
/**
 * 商家库中有每一种商家的惩罚记录，求浙江省的商家中，有惩罚记录的一共有多少个商家。如果商家没有惩罚记录，则商家信息中不存在该字段。
 * 等效的SQL语句是SELECT count(column_history) FROM product where place_of_production="浙江省"。
 */
public void count(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.term("place", "浙江省"))
                .limit(0)
                .addAggregation(AggregationBuilders.count("count_agg_1", "column_history"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsCountAggregationResult("count_agg_1").getValue());
}
```

去重统计行数

返回指定字段不同值的数量，类似于SQL中的count（distinct）。

> **说明** 去重统计行数的计算结果是个近似值。
>
> - 当去重统计行数小于1万时，计算结果是一个精确值。
> - 当去重统计行数达到1亿时，计算结果的误差为2%左右。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 求所有的商品，产地一共来自多少个省份。
 * 等效的SQL语句是SELECT count(distinct column_place) FROM product。
 */
public void distinctCount(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.distinctCount("dis_count_agg_1", "col
umn_place"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
dis_count_agg_1").getValue());
}
```

百分位统计

百分位统计常用来统计一组数据的百分位分布情况，例如在日常系统运维中统计每次请求访问的耗时情况时，需要关注系统请求耗时的P25、P50、P90、P99值等分布情况。

> **说明** 百分位统计为非精确统计，对不同百分位数值的计算精确度不同，较为极端的百分位数值更加准确，例如1%或99%的百分位数值会比50%的百分位数值准确。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
percentiles	百分位分布例如50、90、99，可根据需要设置一个或者多个百分位。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 分析系统请求耗时分位数分布情况。
 */
public  void percentilesAgg(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addAggregation(AggregationBuilders.percentiles("percentilesAgg", "latenc
y")
                    .percentiles(Arrays.asList(25.0d, 50.0d, 99.0d))
                    .missing(1.0))
                .build())
        .build();
    //执行查询
    SearchResponse resp = client.search(searchRequest);
    //获取结果
    PercentilesAggregationResult percentilesAggregationResult = resp.getAggregationResult
s().getAsPercentilesAggregationResult(
        "percentilesAgg");
    for (PercentilesAggregationItem item : percentilesAggregationResult.getPercentilesAgg
regationItems()) {
        System.out.println("key: " + item.getKey() + " value:" + item.getValue().asDouble
());
    }
}
```

字段值分组

根据一个字段的值对查询结果进行分组，相同的字段值放到同一分组内，返回每个分组的值和该值对应的个数。

> **说明** 当分组较大时，按字段值分组可能会存在误差。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long、Double、Boolean和Keyword类型。

参数	说明
groupBySorter	分组中的item排序规则，默认按照分组中item的数量降序排序，多个排序则按照添加的顺序进行排列。支持的参数如下。 ○ 按照值的字典序升序排列 ○ 按照值的字典序降序排列 ○ 按照行数升序排列 ○ 按照行数降序排列 ○ 按照子统计聚合结果中值升序排列 ○ 按照子统计聚合结果中值降序排列
size	返回的分组数量，最大值为2000。当分组数量超过2000时，只会返回前2000个分组。
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 ○ 场景 统计每个类别的商品数量，且统计每个类别价格的最大值和最小值。 ○ 方法 最外层的统计聚合是根据类别进行分组，再添加两个子统计聚合求价格的最大值和最小值。 ○ 结果示例 ■ 水果：5个（其中价格最高15元，最低3元） ■ 洗漱用品：10个（其中价格最高98元，最低1元） ■ 电子设备：3个（其中价格最高8699元，最低2300元） ■ 其它：15个（其中价格最高1000元，最低80元）

- 示例1

```
/**
 * 所有商品中每一个类别各有多少个，且统计每一个类别的价格最大值和最小值。
 * 返回结果举例："水果：5个（其中价格最贵15元，最便宜3元），洗漱用品：10个（其中价格最贵98元，最便宜
1元），电子设备：3个（其中价格最贵8699元，最便宜2300元），其它：15个（其中价格最贵1000
 * 元，最便宜80元）"。
 */
public void groupByField(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByField("name1", "column_type")
                    .addSubAggregation(AggregationBuilders.min("subName1", "column_price"
))
                    .addSubAggregation(AggregationBuilders.max("subName2", "column_price"
))
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("
name1").getGroupByFieldResultItems()) {
        //打印值。
        System.out.println(item.getKey());
        //打印个数。
        System.out.println(item.getRowCount());
        //打印最低价格。
        System.out.println(item.getSubAggregationResults().getAsMinAggregationResult("sub
Name1").getValue());
        //打印最高价格。
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("sub
Name2").getValue());
    }
}
```

- 示例2

```
    /**
     * 按照多字段分组的示例。
     * 多元索引目前不能原生支持SQL中的groupBy多字段，但是可以通过嵌套使用两个groupBy完成相似功能。
     * 等效的SQL语句是select a,d, sum(b),sum(c) from user group by a,d。
     */
    public void GroupByMultiField() {
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .returnAllColumns(true)    //设置为false时，指定addColumnsToGet，性能会高。
```

```
            //.addColumnsToGet("col_1","col_2")
            .searchQuery(SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())    //此处相当于SQL中的where条件，可以通过Quer
yBuilders.bool()嵌套查询实现复杂的查询。
                .addGroupBy(
                    GroupByBuilders
                        .groupByField("任意唯一名字标识_1", "field_a")
                        .size(20)
                        .addSubGroupBy(
                            GroupByBuilders
                                .groupByField("任意唯一名字标识_2", "field_d")
                                .size(20)
                                .addSubAggregation(AggregationBuilders.sum("任意唯一名字标
识_3", "field_b"))
                                .addSubAggregation(AggregationBuilders.sum("任意唯一名字标
识_4", "field_c"))
                        )
                )
                .build())
            .build();
        SearchResponse response = client.search(searchRequest);
        //查询符合条件的行。
        List<Row> rows = response.getRows();
        //获取统计聚合结果。
        GroupByFieldResult groupByFieldResult1 = response.getGroupByResults().getAsGroupB
yFieldResult("任意唯一名字标识_1");
        for (GroupByFieldResultItem resultItem : groupByFieldResult1.getGroupByFieldResul
tItems()) {
            System.out.println("field_a key:" + resultItem.getKey() + " Count:" + resultI
tem.getRowCount());
            //获取子统计聚合结果。
            GroupByFieldResult subGroupByResult = resultItem.getSubGroupByResults().getAs
GroupByFieldResult("任意唯一名字标识_2");
            for (GroupByFieldResultItem item : subGroupByResult.getGroupByFieldResultItem
s()) {
                System.out.println("field_a " + resultItem.getKey() + " field_d key:" + i
tem.getKey() + " Count: " + item.getRowCount());
                double sumOf_field_b = item.getSubAggregationResults().getAsSumAggregatio
nResult("任意唯一名字标识_3").getValue();
                double sumOf_field_c = item.getSubAggregationResults().getAsSumAggregatio
nResult("任意唯一名字标识_4").getValue();
                System.out.println("sumOf_field_b:" + sumOf_field_b);
                System.out.println("sumOf_field_c:" + sumOf_field_c);
            }
        }
    }
```

- 示例3

```
    /**
     * 使用统计聚合排序的示例。
     * 使用方法：按顺序添加GroupBySorter即可，添加多个GroupBySorter时排序结果按照添加顺序生效。GroupBySorter支持升序和降序两种方式。
     * 默认排序是按照行数降序排列即GroupBySorter.rowCountSortInDesc()。
     */
    public void groupByFieldWithSort(SyncClient client) {
        //构建查询语句。
        SearchRequest searchRequest = SearchRequest.newBuilder()
            .tableName("tableName")
            .indexName("indexName")
            .searchQuery(
                SearchQuery.newBuilder()
                    .query(QueryBuilders.matchAll())
                    .limit(0)
                    .addGroupBy(GroupByBuilders
                        .groupByField("name1", "column_type")
                        //.addGroupBySorter(GroupBySorter.subAggSortInAsc("subName1")) //按照子统计聚合结果中的值升序排序。
                        .addGroupBySorter(GroupBySorter.groupKeySortInAsc())           //按照统计聚合结果中的值升序排序。
                        //.addGroupBySorter(GroupBySorter.rowCountSortInDesc())        //按照统计聚合结果中的行数降序排序。
                        .size(20)
                        .addSubAggregation(AggregationBuilders.min("subName1", "column_price"))
                        .addSubAggregation(AggregationBuilders.max("subName2", "column_price"))
                    )
                    .build())
            .build();
        //执行查询。
        SearchResponse resp = client.search(searchRequest);
    }
```

范围分组

根据一个字段的范围对查询结果进行分组，字段值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
range[double_from, double_to)	分组的范围。 起始值double_from可以使用最小值Double.MIN_VALUE，结束值double_to可以使用最大值Double.MAX_VALUE。

参数	说明
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 例如按销量分组后再按省份分组，即可获得某个销量范围内哪个省比重比较大，实现方法是GroupByRange下添加一个GroupByField。

- 示例

```
/**
 * 求商品销量时按[0，1000）、[1000，5000）、[5000，Double.MAX_VALUE）这些分组计算每个范围的销量。
 */
public void groupByRange(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByRange("name1", "column_number")
                    .addRange(0, 1000)
                    .addRange(1000, 5000)
                    .addRange(5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    for (GroupByRangeResultItem item : resp.getGroupByResults().getAsGroupByRangeResult("name1").getGroupByRangeResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

地理位置分组

根据距离某一个中心点的范围对查询结果进行分组，距离差值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Geo_point类型。

参数	说明
origin(double lat, double lon)	起始中心点的经纬度。 double lat是起始中心点纬度，double lon是起始中心点经度。
range[double_from, double_to)	分组的范围，单位为米。 起始值double_from可以使用最小值Double.MIN_VALUE，结束值double_to可以使用最大值Double.MAX_VALUE。
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

```
/**
 * 求距离万达广场[0，1000）、[1000，5000）、[5000，Double.MAX_VALUE）这些范围内的人数，距离的单位为米。
 */
public void groupByGeoDistance(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByGeoDistance("name1", "column_geo_point")
                    .origin(3.1, 6.5)
                    .addRange(0, 1000)
                    .addRange(1000, 5000)
                    .addRange(5000, Double.MAX_VALUE)
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取统计聚合结果。
    for (GroupByGeoDistanceResultItem item : resp.getGroupByResults().getAsGroupByGeoDistanceResult("name1").getGroupByGeoDistanceResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

过滤条件分组

按照过滤条件对查询结果进行分组，获取每个过滤条件匹配到的数量，返回结果的顺序和添加过滤条件的顺序一致。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
filter	过滤条件，返回结果的顺序和添加过滤条件的顺序一致。
subAggregation和subGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

```
/**
 * 按照过滤条件进行分组，例如添加三个过滤条件（销量大于100、产地是浙江省、描述中包含杭州关键词），然后获取每个过滤条件匹配到的数量。
 */
public void groupByFilter(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addGroupBy(GroupByBuilders
                    .groupByFilter("name1")
                    .addFilter(QueryBuilders.range("number").greaterThanOrEqual(100))
                    .addFilter(QueryBuilders.term("place","浙江省"))
                    .addFilter(QueryBuilders.match("text","杭州"))
                )
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //按照过滤条件的顺序获取的统计聚合结果。
    for (GroupByFilterResultItem item : resp.getGroupByResults().getAsGroupByFilterResult
("name1").getGroupByFilterResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

直方图统计

按照指定数据间隔对查询结果进行分组，字段值在相同范围内放到同一分组内，返回每个分组的值和该值对应的个数。

- 参数

参数	说明
groupByName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
fieldName	用于统计聚合的字段，仅支持Long和Double类型。
interval	统计间隔。
fieldRange[min,max]	统计范围，与interval参数配合使用限制分组的数量。`(fieldRange.max-fieldRange.min)/interval` 的值不能超过2000。
minDocCount	最小行数。当分组中的行数小于最小行数时，不会返回此分组的统计结果。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 统计不同年龄段用户数量分布情况。
 */
public static void groupByHistogram(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .addGroupBy(GroupByBuilders
                    .groupByHistogram("groupByHistogram", "age")
                    .interval(10)
                    .minDocCount(0L)
                    .addFieldRange(0, 99))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = ots.search(searchRequest);
    //获取直方图的统计聚合结果。
    GroupByHistogramResult results = resp.getGroupByResults().getAsGroupByHistogramResult
("groupByHistogram");
    for (GroupByHistogramItem item : results.getGroupByHistogramItems()) {
        System.out.println("key: " + item.getKey().asLong() + " value:" + item.getValue()
);
    }
}
```

获取统计聚合分组中的行

对查询结果进行分组后，获取每个分组内的一些行数据，可实现和MySQL中ANY_VALUE(field)类似的功能。

> **说明** 获取统计聚合分组中的行时，如果多元索引中包含嵌套类型、地理位置类型或者数组类型的字段，则返回结果中只会包含主键信息，请手动反查数据表获取所需字段。

- 参数

参数	说明
aggregationName	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
limit	每个分组内最多返回的数据行数，默认返回1行数据。
sort	分组内数据的排序方式。
columnsToGet	指定返回的字段，仅支持多元索引中的字段，且不支持数组、Geopoint和Nested类型的字段。

- 示例

```
/**
 * 某学校有一个活动报名表，活动报名表中包含学生姓名、班级、班主任、班长等信息，如果希望按班级进行分组，以查看每个班级的报名情况，同时获取班级的属性信息。
 * 等效的SQL语句是select className, teacher, monitor, COUNT(*) as number from table GROUP BY className。
 */
public void testTopRows(SyncClient client) {
    SearchRequest searchRequest = SearchRequest.newBuilder()
            .indexName("indexName")
            .tableName("tableName")
            .searchQuery(
                    SearchQuery.newBuilder()
                            .query(QueryBuilders.matchAll())
                            .limit(0)
                            .addGroupBy(GroupByBuilders.groupByField("groupName", "className")
                                    .size(5)
                                    .addSubAggregation(AggregationBuilders.topRows("topRowsName")
                                            .limit(1)
                                            .sort(new Sort(Arrays.asList(new FieldSort("teacher", SortOrder.DESC)))) // topRows的排序
                                    )
                            )
                            .build())
            .addColumnsToGet(Arrays.asList("teacher", "monitor"))
            .build();
    SearchResponse resp = client.search(searchRequest);
    List<GroupByFieldResultItem> items = resp.getGroupByResults().getAsGroupByFieldResult("groupName").getGroupByFieldResultItems();
    for (GroupByFieldResultItem item : items) {
        String className = item.getKey();
        long number = item.getRowCount();
        List<Row> topRows = item.getSubAggregationResults().getAsTopRowsAggregationResult("topRowsName").getRows();
        Row row = topRows.get(0);
        String teacher = row.getLatestColumn("teacher").getValue().asString();
        String monitor = row.getLatestColumn("monitor").getValue().asString();
    }
}
```

嵌套

分组类型的统计聚合功能支持嵌套，其内部可以添加子统计聚合。

主要用于在分组内再次进行统计聚合，以两层的嵌套为例：

- GroupBy+SubGroupBy：按省份分组后再按照城市分组，获取每个省份下每个城市的数据。
- GroupBy+SubAggregation：按照省份分组后再求某个指标的最大值，获取每个省的某个指标最大值。

> **说明** 为了性能、复杂度等综合考虑，嵌套的层级只开放了一定的层数。更多信息，请参见多元索引限制。

示例

```
/**
 * 嵌套的统计聚合示例。
 * 外层2个Aggregation和1个GroupByField，GroupByField中又添加了2个Aggregation和1个GroupByRange。
 */
public void subGroupBy(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .indexName("index_name")
        .tableName("table_name")
        .returnAllColumns(true)
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.match("textField", "hello"))
                .limit(10)
                .addAggregation(AggregationBuilders.min("name1", "fieldName1"))
                .addAggregation(AggregationBuilders.max("name2", "fieldName2"))
                .addGroupBy(GroupByBuilders
                    .groupByField("name3", "fieldName3")
                    .addSubAggregation(AggregationBuilders.max("subName1", "fieldName4"))
                    .addSubAggregation(AggregationBuilders.sum("subName2", "fieldName5"))
                    .addSubGroupBy(GroupByBuilders
                        .groupByRange("subName3", "fieldName6")
                        .addRange(12, 90)
                        .addRange(100, 900)
                    ))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取第一层求最小值和求最大值的统计聚合结果。
    AggregationResults aggResults = resp.getAggregationResults();
    System.out.println(aggResults.getAsMinAggregationResult("name1").getValue());
    System.out.println(aggResults.getAsMaxAggregationResult("name2").getValue());
    //获取第一层按字段值分组的统计聚合结果，并同时获取其嵌套的子统计聚合结果。
    GroupByFieldResult results = resp.getGroupByResults().getAsGroupByFieldResult("someName
1");
    for (GroupByFieldResultItem item : results.getGroupByFieldResultItems()) {
        System.out.println("数量: " + item.getRowCount());
        System.out.println("key: " + item.getKey());
        //获取子统计聚合结果。
        //打印求最大值的子统计聚合结果。
        System.out.println(item.getSubAggregationResults().getAsMaxAggregationResult("subNa
me1"));
        //打印求和的子统计聚合结果。
        System.out.println(item.getSubAggregationResults().getAsSumAggregationResult("subNa
me2"));
        //打印按范围分组的子统计聚合结果。
        GroupByRangeResult subResults = resp.getGroupByResults().getAsGroupByRangeResult("s
ubName3");
        for (GroupByRangeResultItem subItem : subResults.getGroupByRangeResultItems()) {
            System.out.println("数量: " + subItem.getRowCount());
            System.out.println("key: " + subItem.getKey());
```

```
        }
    }
}
```

多个统计聚合

多个统计聚合功能可以组合使用。

> **说明** 当多个统计聚合的复杂度较高时可能会影响响应速度。

- 示例1

```
public void multipleAggregation(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取求最小值的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").ge
tValue());
    //获取求和的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").ge
tValue());
    //获取去重统计行数的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
name3").getValue());
}
```

- 示例2

```
public void multipleGroupBy(SyncClient client) {
    //构建查询语句。
    SearchRequest searchRequest = SearchRequest.newBuilder()
        .tableName("tableName")
        .indexName("indexName")
        .searchQuery(
            SearchQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(0)
                .addAggregation(AggregationBuilders.min("name1", "long"))
                .addAggregation(AggregationBuilders.sum("name2", "long"))
                .addAggregation(AggregationBuilders.distinctCount("name3", "long"))
                .addGroupBy(GroupByBuilders.groupByField("name4", "type"))
                .addGroupBy(GroupByBuilders.groupByRange("name5", "long").addRange(1, 15)
)
                .build())
        .build();
    //执行查询。
    SearchResponse resp = client.search(searchRequest);
    //获取求最小值的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsMinAggregationResult("name1").ge
tValue());
    //获取求和的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsSumAggregationResult("name2").ge
tValue());
    //获取去重统计行数的统计聚合结果。
    System.out.println(resp.getAggregationResults().getAsDistinctCountAggregationResult("
name3").getValue());
    //获取按字段值分组的统计聚合结果。
    for (GroupByFieldResultItem item : resp.getGroupByResults().getAsGroupByFieldResult("
name4").getGroupByFieldResultItems()) {
        //打印key。
        System.out.println(item.getKey());
        //打印个数。
        System.out.println(item.getRowCount());
    }
    //获取按范围分组的统计聚合结果。
    for (GroupByRangeResultItem item : resp.getGroupByResults().getAsGroupByRangeResult("
name5").getGroupByRangeResultItems()) {
        //打印个数。
        System.out.println(item.getRowCount());
    }
}
```

3.8.20. 并发导出数据

当使用场景中不关心整个结果集的顺序时，可以使用并发导出数据功能以更快的速度将命中的数据全部返回。

背景

多元索引中提供了Search接口，Search接口支持全功能集，包括所有的查询功能，以及排序、统计聚合等分析能力，其结果会按照指定的顺序返回。

但是在有些场景中，例如对接计算系统Spark、Presto等或者一些圈选场景，只需要使用完整的查询能力，能将命中的数据以更快的速度全部返回，不关心整个结果集的顺序。为了支持此类需求，多元索引中提供了ParallelScan接口。

> **说明** 5.6.0及其以上版本的SDK开始支持ParallelScan功能。

ParallelScan接口相对于Search接口，保留了所有的查询功能，但是舍弃了排序、统计聚合等分析功能，带来了5倍以上的性能提升，因此可以实现1分钟内上亿级别数据行的导出能力，导出能力可以水平扩展，不存在上限。

在接口的实现中，单次请求的limit限制更宽松。目前Search接口的limit最大允许100行，但是ParallelScan接口的limit最大允许2000行。同时支持多个线程一起并发请求，因此导出速度会极快。

场景

- 如果请求关心排序、统计聚合，或者是终端客户的直接查询请求，需要用Search接口。
- 如果请求不关心排序，只关心把所有符合条件的数据尽快返回，或者是计算系统（Spark、Presto等）拉取数据，可以考虑ParallelScan接口。

特点

ParallelScan接口相对于Search接口的典型特征如下：

- 结果稳定性

  ParallelScan任务是有状态的。在一个Session请求中，获取到的结果集是确定的，由发起第一次请求时的数据状态决定。如果发起第一次请求后插入了新的数据或者修改了原有的数据不会对结果集造成影响。

- 新增会话（Session）概念

  在ParallelScan系列接口中新增了Session概念。使用sessionId能够保证获取到的结果集是稳定的，具体流程如下：

  i. 通过ComputeSplits接口获取最大并发数和当前sessionId。

  ii. 通过发起多个并发ParallelScan请求读取数据，请求中需要指定当前的sessionId和当前并发ID。

  在某些不易获取sessionId的场景中，ParallelScan也支持不携带sessionId发起请求，但是不使用sessionId可能会有极小的概率导致获取到的结果集中有重复数据。

  如果在ParallelScan过程中发生网络异常、线程异常、动态Schema修改、索引切换等情况，导致ParallelScan不能继续扫描数据，服务端会返回“OTSSessionExpired”错误码，此时需要重新开始一个新的ParallelScan任务，从最开始重新拉取数据。

- 最大并发数

  ParallelScan支持的单请求的最大并发数由ComputeSplits的返回值确定。数据越多，支持的并发数越大。

  单请求指同一个查询语句，例如查询city=“杭州”的结果，如果使用Search接口，那么Search请求的返回值中会包括所有city=“杭州”的结果；如果使用ParallelScan接口且并发数是2，那么每个ParallelScan请求返回50%的结果，然后将两个并发的结果集合并在一起才是完整的结果集。

- 每次返回行数

  limit默认为2000，最大可以到2000。超过2000后，limit的变化对性能基本无影响。

- 性能

  ParallelScan接口单并发扫描数据的性能是Search接口的5倍。当增大并发数时，性能可以继续线性提高，例如8并发时仍可以继续提高4倍性能。

- 成本

  由于ParallelScan请求对资源的消耗更少，价格会更便宜，所以对于大数据量的导出类需求，强烈建议使用ParallelScan接口。

- 返回列

  只支持返回多元索引中已建立索引的列，不能返回多元索引中没有的列，可以支持的ReturnType是RETURN_ALL_INDEX或者RETURN_SPECIFIED，不支持RETURN_ALL。

  目前已支持返回数组和地理位置，但是返回的字段值会被格式化，可能和写入数据表的值不一致。例如对于数组类型，写入数据表的值为"[1,2, 3, 4]"，则通过ParallelScan接口导出的值为"[1,2,3,4]"；对于地理位置类型，写入数据表的值为 `10,50` ，则通过ParallelScan接口导出的值为 `10.0,50.0` 。

- 限制项

  同时存在的ParallelScan任务数量有一定的限制，当前为10个，后续会根据客户需求继续调整。同一个sessionId且ScanQuery相同的多个并发任务视为一个任务。一个ParallelScan任务的生命周期定义为：开始时间是第一次发出ParallelScan请求，结束时间是翻页扫描完所有数据或者请求的token失效。

接口

多并发数据导出功能涉及如下两个接口。

- ComputeSplits：获取当前ParallelScan单个请求的最大并发数。
- ParallelScan：执行具体的数据导出功能。

参数

参数		说明
tableName		数据表名称。
indexName		多元索引名称。
scanQuery	query	多元索引的查询语句。支持精确查询、模糊查询、范围查询、地理位置查询、嵌套查询等，功能和Search接口一致。
	limit	扫描数据时一次能返回的数据行数。
	maxParallel	最大并发数。请求支持的最大并发数由用户数据量决定。数据量越大，支持的并发数越多，每次任务前可以通过ComputeSplits API进行获取。
	currentParallelId	当前并发ID。取值范围为[0, maxParallel)。
	token	用于翻页功能。ParallelScan请求结果中有下一次进行翻页的token，使用该token可以接着上一次的结果继续读取数据。

参数		说明
	aliveTime	ParallelScan的当前任务有效时间，也是token的有效时间。默认值为60，建议使用默认值，单位为秒。如果在有效时间内没有发起下一次请求，则不能继续读取数据。持续发起请求会刷新token有效时间。 **说明** 由于服务端采用异步方式清理过期任务，因此当前任务只保证在设置的有效时间内不会过期，但不能保证有效时间之后一定过期。
columnsToGet		ParallelScan目前仅可以扫描多元索引中的数据，需要在创建多元索引时设置附加存储（即store=true）。
sessionId		本次并发扫描数据任务的sessionId。创建Session可以通过ComputeSplits API来创建，同时获得本次任务支持的最大并发数。

示例

单并发扫描数据和多线程并发扫描数据的代码示例如下：

- 单并发扫描数据

  相对于多并发扫描数据，单并发扫描数据的代码更简单，单并发代码无需关心currentParallelId和maxParallel参数。

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ParallelScanResponse;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.SearchRequest.ColumnsToGet;
import com.alicloud.openservices.tablestore.model.search.query.MatchAllQuery;
import com.alicloud.openservices.tablestore.model.search.query.Query;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
public class Test {
    public static List<Row> scanQuery(final SyncClient client) {
        String tableName = "<TableName>";
        String indexName = "<IndexName>";
        //获取sessionId和本次请求支持的最大并发数。
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsR
equest);
        byte[] sessionId = computeSplitsResponse.getSessionId();
```

```
        int splitsSize = computeSplitsResponse.getSplitsSize();
        /*
         *  创建并发扫描数据请求。
         */
        ParallelScanRequest parallelScanRequest = new ParallelScanRequest();
        parallelScanRequest.setTableName(tableName);
        parallelScanRequest.setIndexName(indexName);
        ScanQuery scanQuery = new ScanQuery();
        //该query决定了扫描出的数据范围，可用于构建嵌套的复杂的query。
        Query query = new MatchAllQuery();
        scanQuery.setQuery(query);
        //设置单次请求返回的数据行数。
        scanQuery.setLimit(2000);
        parallelScanRequest.setScanQuery(scanQuery);
        ColumnsToGet columnsToGet = new ColumnsToGet();
        columnsToGet.setColumns(Arrays.asList("col_1", "col_2"));
        parallelScanRequest.setColumnsToGet(columnsToGet);
        parallelScanRequest.setSessionId(sessionId);
        /*
         * 使用builder模式创建并发扫描数据请求，功能与前面一致。
         */
        ParallelScanRequest parallelScanRequestByBuilder = ParallelScanRequest.newBuilder
()
            .tableName(tableName)
            .indexName(indexName)
            .scanQuery(ScanQuery.newBuilder()
                .query(QueryBuilders.matchAll())
                .limit(2000)
                .build())
            .addColumnsToGet("col_1", "col_2")
            .sessionId(sessionId)
            .build();
        List<Row> result = new ArrayList<>();
        /*
         * 使用原生的API扫描数据。
         */
        {
            ParallelScanResponse parallelScanResponse = client.parallelScan(parallelScanR
equest);
            //下次请求的ScanQuery的token。
            byte[] nextToken = parallelScanResponse.getNextToken();
            //获取数据。
            List<Row> rows = parallelScanResponse.getRows();
            result.addAll(rows);
            while (nextToken != null) {
                //设置token。
                parallelScanRequest.getScanQuery().setToken(nextToken);
                //继续扫描数据。
                parallelScanResponse = client.parallelScan(parallelScanRequest);
                //获取数据。
                rows = parallelScanResponse.getRows();
                result.addAll(rows);
                nextToken = parallelScanResponse.getNextToken();
            }
```

```
        }
        /*
         *  推荐方式。
         *  使用iterator方式扫描所有匹配数据。使用方式上更简单，速度和前面方法一致。
         */
        {
            RowIterator iterator = client.createParallelScanIterator(parallelScanRequestB
yBuilder);
            while (iterator.hasNext()) {
                Row row = iterator.next();
                result.add(row);
                //获取具体的值。
                String col_1 = row.getLatestColumn("col_1").getValue().asString();
                long col_2 = row.getLatestColumn("col_2").getValue().asLong();
            }
        }
        /*
         * 关于失败重试的问题，如果本函数的外部调用者有重试机制或者不需要考虑失败重试问题，可以忽略
此部分内容。
         * 为了保证可用性，遇到任何异常均推荐进行任务级别的重试，重新开始一个新的ParallelScan任务。
         * 异常分为如下两种：
         * 1、服务端Session异常OTSSessionExpired。
         * 2、调用者客户端网络等异常。
         */
        try {
            //正常处理逻辑。
            {
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequ
estByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    //处理row，内存足够大时可直接放到list中。
                    result.add(row);
                }
            }
        } catch (Exception ex) {
            //重试。
            {
                result.clear();
                RowIterator iterator = client.createParallelScanIterator(parallelScanRequ
estByBuilder);
                while (iterator.hasNext()) {
                    Row row = iterator.next();
                    //处理row，内存足够大时可直接放到list中。
                    result.add(row);
                }
            }
        }
        return result;
    }
}
```

- 多线程并发扫描数据

```
import com.aliyun.openservices.tablestore.SyncClient;
```

```
import com.alicloud.openservices.tablestore.SyncClient;
import com.alicloud.openservices.tablestore.model.ComputeSplitsRequest;
import com.alicloud.openservices.tablestore.model.ComputeSplitsResponse;
import com.alicloud.openservices.tablestore.model.Row;
import com.alicloud.openservices.tablestore.model.SearchIndexSplitsOptions;
import com.alicloud.openservices.tablestore.model.iterator.RowIterator;
import com.alicloud.openservices.tablestore.model.search.ParallelScanRequest;
import com.alicloud.openservices.tablestore.model.search.ScanQuery;
import com.alicloud.openservices.tablestore.model.search.query.QueryBuilders;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Semaphore;
import java.util.concurrent.atomic.AtomicLong;
public class Test {
    public static void scanQueryWithMultiThread(final SyncClient client, String tableName
, String indexName) throws InterruptedException {
        // 获取机器的CPU数量。
        final int cpuProcessors = Runtime.getRuntime().availableProcessors();
        // 指定客户端多线程的并发数量。建议和客户端的CPU核数一致，避免客户端压力太大，影响查询性能。
        final Semaphore semaphore = new Semaphore(cpuProcessors);
        // 获取sessionId和本次请求支持的最大并发数。
        ComputeSplitsRequest computeSplitsRequest = new ComputeSplitsRequest();
        computeSplitsRequest.setTableName(tableName);
        computeSplitsRequest.setSplitsOptions(new SearchIndexSplitsOptions(indexName));
        ComputeSplitsResponse computeSplitsResponse = client.computeSplits(computeSplitsR
equest);
        final byte[] sessionId = computeSplitsResponse.getSessionId();
        final int maxParallel = computeSplitsResponse.getSplitsSize();
        // 业务统计行数使用。
        AtomicLong rowCount = new AtomicLong(0);
        /*
         *  为了使用一个函数实现多线程功能，此处构建一个内部类继承Thread来使用多线程。
         *  您也可以构建一个正常的外部类，使代码更有条理。
         */
        final class ThreadForScanQuery extends Thread {
            private final int currentParallelId;
            private ThreadForScanQuery(int currentParallelId) {
                this.currentParallelId = currentParallelId;
                this.setName("ThreadForScanQuery:" + maxParallel + "-" + currentParallelI
d);  // 设置线程名称。
            }
            @Override
            public void run() {
                System.out.println("start thread:" + this.getName());
                try {
                    // 正常处理逻辑。
                    {
                        ParallelScanRequest parallelScanRequest = ParallelScanRequest.new
Builder()
                                .tableName(tableName)
                                .indexName(indexName)
                                .scanQuery(ScanQuery.newBuilder()
                                        .query(QueryBuilders.range("col_long").lessThan(1
0_0000)) // 此处的query决定了获取什么数据。
                                        .limit(2000)
```

```
                                        .currentParallelId(currentParallelId)
                                        .maxParallel(maxParallel)
                                        .build())
                                .addColumnsToGet("col_long", "col_keyword", "col_bool")
// 设置要返回的多元索引中的部分字段，或者使用下行注释的内容获取多元索引中全部数据。
                                //.returnAllColumnsFromIndex(true)
                                .sessionId(sessionId)
                                .build();
                        // 使用Iterator形式获取所有数据。
                        RowIterator ltr = client.createParallelScanIterator(parallelScanR
equest);

                        long count = 0;
                        while (ltr.hasNext()) {
                            Row row = ltr.next();
                            // 增加自定义的处理逻辑，此处代码以统计行数为例介绍。
                            count++;
                        }
                        rowCount.addAndGet(count);
                        System.out.println("thread[" + this.getName() + "] finished. this
thread get rows:" + count);
                    }
                } catch (Exception ex) {
                    // 如果有异常，此处需要考虑重试正常处理逻辑。
                } finally {
                    semaphore.release();
                }
            }
        }
        // 多个线程同时执行，currentParallelId取值范围为[0, maxParallel)。
        List<ThreadForScanQuery> threadList = new ArrayList<ThreadForScanQuery>();
        for (int currentParallelId = 0; currentParallelId < maxParallel; currentParallelI
d++) {
            ThreadForScanQuery thread = new ThreadForScanQuery(currentParallelId);
            threadList.add(thread);
        }
        // 同时启动。
        for (ThreadForScanQuery thread : threadList) {
            // 利用semaphore限制同时启动的线程数量，避免客户端瓶颈。
            semaphore.acquire();
            thread.start();
        }
        // 主线程阻塞等待所有线程完成任务。
        for (ThreadForScanQuery thread : threadList) {
            thread.join();
        }
        System.out.println("all thread finished! total rows:" + rowCount.get());
    }
}
```

3.8.21. 虚拟列

使用虚拟列功能时，您可以通过修改多元索引Schema或者新建多元索引来实现新字段新数据类型的查询功能，而无需修改表格存储的存储结构及数据。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。

注意事项

- 虚拟列支持不同类型到字符串类型的相互转换，转换规则请参见下表。

数据表中字段类型	虚拟列字段类型
String	Keyword（含数组）
String	Text（含数组）
String	Long（含数组）
String	Double（含数组）
String	Geo-point（含数组）
Long	Keyword
Long	Text
Double	Keyword
Double	Text

- 虚拟列目前仅支持用在查询语句中，不能用在ColumnsToGet返回列值，如果需要返回列值，可以指定返回该虚拟列的原始列。

参数

更多信息，请参见创建多元索引。

示例

1. 创建多元索引时指定虚拟列。

   创建一个多元索引，多元索引包含Col_Keyword和Col_Long两列，同时创建虚拟列Col_Keyword_Virtual_Long和Col_Long_Virtual_Keyword。Col_Keyword_Virtual_Long映射为数据表中Col_Keyword列，虚拟列Col_Long_Virtual_Keyword映射为数据表中Col_Long列。

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(tableName); //设置数据表名称。
    request.setIndexName(indexName); //设置多元索引名称。
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
        new FieldSchema("Col_Keyword", FieldType.KEYWORD) //设置字段名和类型。
            .setIndex(true) //设置开启索引。
            .setEnableSortAndAgg(true) //设置开启排序和统计功能。
            .setStore(true),
        new FieldSchema("Col_Keyword_Virtual_Long", FieldType.LONG) //设置字段名和类型。
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true) //设置字段是否为虚拟列。
            .setSourceFieldName("Col_Keyword"), //虚拟列对应的数据表中字段。
        new FieldSchema("Col_Long", FieldType.LONG)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true),
        new FieldSchema("Col_Long_Virtual_Keyword", FieldType.KEYWORD)
            .setIndex(true)
            .setEnableSortAndAgg(true)
            .setStore(true)
            .setVirtualField(true)
            .setSourceFieldName("Col_Long")));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); //调用client创建多元索引。
}
```

2. 使用虚拟列查询数据。

   查询表中Col_Long_Virtual_Keyword列的值能够匹配"1000"的数据，返回匹配到的总行数和一些匹配成功的行。

```
private static void query(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    TermsQuery termsQuery = new TermsQuery(); //设置查询类型为TermsQuery。
    termsQuery.setFieldName("Col_Long_Virtual_Keyword"); //设置要匹配的字段。
    termsQuery.addTerm(ColumnValue.fromString("1000")); //设置要匹配的值。
    searchQuery.setQuery(termsQuery);
    searchQuery.setGetTotalCount(true); //设置返回匹配的总行数。
    SearchRequest searchRequest = new SearchRequest("tableName", "indexName", searchQue
ry);
    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setReturnAll(true); //设置返回所有列，不支持返回虚拟列。
    searchRequest.setColumnsToGet(columnsToGet);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount()); //匹配到的总行数，非返回行
数。
    System.out.println("Row: " + resp.getRows());
}
```

3.9. 二级索引

3.9.1. 全局二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

> **说明** 二级索引包括全局二级索引和本地二级索引。关于二级索引的更多信息，请参见二级索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建一个索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
mainTableName	数据表名称。

参数	说明
indexMeta	索引表的结构信息，包括如下内容： ○ indexName：索引表名称。 ○ primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 ○ indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ■ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ■ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。
includeBaseData	索引表中是否包含数据表中已存在的数据。 当CreateIndexRequest中的最后一个参数includeBaseData设置为true时，表示包含存量数据；设置为false时，表示不包含存量数据。

- 示例

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX2_NAME); //设置索引表名称。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列，设置DEFINED_COL_NAME_1列为索引表的第一列主键。
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2); //为索引表添加主键列，设置PRIMARY_KEY_NAME_2列为索引表的第二列主键。
    indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); //为索引表添加属性列，设置DEFINED_COL_NAME_2列为索引表的属性列。
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); //添加索引表到数据表，包含存量数据。
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); //添加索引表到数据表，不包含存量数据。
    /**通过将IncludeBaseData参数设置为true，创建索引表后会开启数据表中存量数据的同步，然后可以通过索引表查询全部数据，
        同步时间和数据量的大小有一定的关系。
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); //创建索引表。
}
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - tableName需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

    当需要返回的属性列在索引表中时，可以直接读取索引表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("扫描索引表的结果为:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

当需要返回的属性列不在索引表中时，请自行反查数据表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
```

```
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBu
ilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, ke2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); //根据索引表主键构造数据
表主键。
            //反查数据表。
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, ma
inTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME_3); //设置读取数据表的DEFINED_COL_N
AME_3列。
            //设置读取最新版本。
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
mainTableName	数据表名称。
indexName	索引表名称。

- 示例

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME, INDEX_NAME); //设置数据表名称和索引表名称。
    client.deleteIndex(request); //删除索引表。
}
```

3.9.2. 本地二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

> **说明** 二级索引包括全局二级索引和本地二级索引。关于二级索引的更多信息，请参见二级索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建一个索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
mainTableName	数据表名称。

参数	说明
indexMeta	索引表的结构信息，包括如下内容： ○ indexName：索引表名称。 ○ primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 ○ indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ■ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ■ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。
includeBaseData	索引表中是否包含数据表中已存在的数据。 当CreateIndexRequest中的最后一个参数includeBaseData设置为true时，表示包含存量数据；设置为false时，表示不包含存量数据。

- 示例

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); //设置索引表名称。
    indexMeta.setIndexType(IT_LOCAL_INDEX);     //设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    indexMeta.setIndexUpdateMode(IUM_SYNC_INDEX);  //设置索引更新模式为同步更新（IUM_SYNC_INDEX）。当索引类型为本地二级索引时，索引更新模式必须为同步更新。
    indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1); //为索引表添加主键列。索引表的第一列主键必须与数据表的第一列主键相同。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_2); //为索引表添加主键列，设置DEFINED_COL_NAME_2列为索引表的第二列主键。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列，设置DEFINED_COL_NAME_1列为索引表的第三列主键。
    //CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); //添加索引表到数据表，包含存量数据。
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); //添加索引表到数据表，不包含存量数据。
    /**通过将IncludeBaseData参数设置为true，创建索引表后会开启数据表中存量数据的同步，然后可以通过索引表查询全部数据，
        同步时间和数据量的大小有一定的关系。
    */
    //request.setIncludeBaseData(true);
    client.createIndex(request); // 创建索引表。
}
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - tableName需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

    当需要返回的属性列在索引表中时，可以直接读取索引表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    System.out.println("扫描索引表的结果为:");
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

当需要返回的属性列不在索引表中时，请自行反查数据表获取数据。

```
private static void scanFromIndex(SyncClient client) {
    RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(INDEX_NAME)
; //设置索引表名称。
    //设置起始主键。
    PrimaryKeyBuilder startPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilde
r();
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_
MIN); //设置需要读取的索引列最小值。
    startPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_
```

```
MIN); //设置数据表主键最小值。
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder(
);
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MA
X); //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MA
X); //设置数据表主键最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRo
wQueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_N
AME_2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBu
ilder();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, pk2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); //根据索引表主键构造数据
表主键。
            //反查数据表。
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, ma
inTablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME3); // 读取主表的DEFINED_COL_NAME3
列
            //设置读取最新版本。
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextS
tartPrimaryKey());
        } else {
            break;
        }
    }
}
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
mainTableName	数据表名称。
indexName	索引表名称。

- 示例

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME, INDEX_NAME); //设置数据表名称和索引表名称。
    client.deleteIndex(request); //删除索引表。
}
```

3.10. 通道服务

3.10.1. 概述

表格存储的Java SDK提供了通道服务的多个接口封装。

接口封装分为两部分：管控接口和自动化的数据消费框架。

- 管控接口请参考以下文档：
  - 创建新通道
  - 获取表内的通道信息
  - 获取通道的具体信息
  - 删除通道
- 自动化的数据消费框架请参考以下文档：
  - 快速开始
  - 数据消费框架原理介绍

3.10.2. 快速开始

使用Java SDK快速体验通道服务。使用前，您需要了解使用通道服务的注意事项等信息。

注意事项

- TunnelWorkerConfig中默认会启动读数据和处理数据的线程池。如果使用的是单台机器，则会启动多个TunnelWorker，建议共用一个TunnelWorkerConfig。
- 在创建全量加增量类型的Tunnel时，由于Tunnel的增量日志最多会保留7天（具体的值和数据表的Stream的日志过期时间一致），全量数据如果在7天内没有消费完成，则此Tunnel进入增量阶段会出现OTSTunnelExpired错误，导致增量数据无法消费。如果您预估全量数据无法在7天内消费完成，请及时联系表格存储技术支持或添加钉钉群23307953进行咨询。
- TunnelWorker的初始化需要预热时间，该值受TunnelWorkerConfig中的heartbeatIntervalInSec参数影响，可以通过TunnelWorkerConfig中的setHeartbeatIntervalInSec方法配置，默认值为30s，最小值为5s。
- 当Tunnel从全量切换至增量阶段时，全量的Channel会结束，增量的Channel会启动，此阶段会有初始化时间，该值也受TunnelWorkerConfig中的heartbeatIntervalInSec参数影响。

- 当客户端（TunnelWorker）没有被正常shutdown时（例如异常退出或者手动结束），TunnelWorker会自动进行资源的回收，包括释放线程池，自动调用用户在Channel上注册的shutdown方法，关闭Tunnel连接等。

体验通道服务

使用Java SDK最小化的体验通道服务。

1. 初始化Tunnel Client。

```
//endPoint为表格存储实例的endPoint，例如https://instance.cn-hangzhou.ots.aliyuncs.com。
//accessKeyId和accessKeySecret分别为访问表格存储服务的AccessKey的Id和Secret。
//instanceName为实例名称。
final String endPoint = "";
final String accessKeyId = "";
final String accessKeySecret = "";
final String instanceName = "";
TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret, in
stanceName);
```

2. 创建通道。

   请提前创建一张测试表或者使用已有的一张数据表。如果需要新建测试表，可以使用SyncClient中的createTable方法或者使用官网控制台等方式创建。

```
//支持创建TunnelType.BaseData（全量）、TunnelType.Stream（增量）、TunnelType.BaseAndStream
（全量加增量）三种类型的Tunnel。
//如下示例为创建全量加增量类型的Tunnel，如果需创建其它类型的Tunnel，则将CreateTunnelRequest中的
TunnelType设置为相应的类型。
final String tableName = "testTable";
final String tunnelName = "testTunnel";
CreateTunnelRequest request = new CreateTunnelRequest(tableName, tunnelName, TunnelType
.BaseAndStream);
CreateTunnelResponse resp = tunnelClient.createTunnel(request);
//tunnelId用于后续TunnelWorker的初始化，该值也可以通过ListTunnel或者DescribeTunnel获取。
String tunnelId = resp.getTunnelId();
System.out.println("Create Tunnel, Id: " + tunnelId);
```

3. 根据业务自定义数据消费Callback函数，开始自动化的数据消费。

```
//根据业务自定义数据消费Callback函数，即实现IChannelProcessor接口（process和shutdown）。
private static class SimpleProcessor implements IChannelProcessor {
    @Override
    public void process(ProcessRecordsInput input) {
        //ProcessRecordsInput中包含有拉取到的数据。
        System.out.println("Default record processor, would print records count");
        System.out.println(
            //NextToken用于Tunnel Client的翻页。
            String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
        try {
            //模拟消费处理。
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void shutdown() {
        System.out.println("Mock shutdown");
    }
}
//TunnelWorkerConfig默认会启动读数据和处理数据的线程池。如果使用的是单台机器，则会启动多个Tunnel
Worker。
//建议共用一个TunnelWorkerConfig，TunnelWorkerConfig中包括更多的高级参数。
TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
//配置TunnelWorker，并启动自动化的数据处理任务。
TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
try {
    worker.connectAndWorking();
} catch (Exception e) {
    e.printStackTrace();
    worker.shutdown();
    tunnelClient.shutdown();
}
```

配置TunnelWorkerConfig

TunnelWorkerConfig提供Tunnel Client的自定义配置，可根据实际需要配置参数，参数说明请参见下表。

配置	参数	说明
Heartbeat的间隔和超时时间	heartbeatTimeoutInSec	Heartbeat的超时间隔。 默认值为300s。 当Heartbeat发生超时，Tunnel服务端会认为当前TunnelClient不可用（失活），客户端需要重新进行ConnectTunnel。

配置	参数	说明
	heartbeatIntervalInSec	进行Heartbeat的间隔。 Heartbeat用于活跃Channel的探测、Channel状态的更新、（自动化）数据拉取任务的初始化等。 默认值为30s，最小支持配置到5s，单位为s。
记录消费位点的时间间隔	checkpointIntervalInMillis	用户消费完数据后，向Tunnel服务端进行记录消费位点操作（checkpoint）的时间间隔。 默认值为5000ms，单位为ms。 **说明** • 因为读取任务所在机器不同，进程可能会遇到各种类型的错误。例如因为环境因素重启，需要定期对处理完的数据做记录（checkpoint）。当任务重启后，会接着上次的checkpoint继续往后做。在极端情况下，通道服务不保证传给您的记录只有一次，只保证数据至少传一次，且记录的顺序不变。如果出现局部数据重复发送的情况，需要您注意业务的处理逻辑。 • 如果希望减少在出错情况下数据的重复处理，可以增加做checkpoint的频率。但是过于频繁的checkpoint会降低系统的吞吐量，请根据自身业务特点决定checkpoint的操作频率。
客户端的自定义标识	clientTag	客户端的自定义标识，可以生成Tunnel Client ID，用于区分TunnelWorker。
数据处理的自定义Callback	channelProcessor	用户注册的处理数据的Callback，包括process和shutdown方法。
	readRecordsExecutor	用于数据读取的线程池资源。无特殊需求，建议使用默认的配置。

配置	参数	说明
数据读取和数据处理的线程池资源配置	processRecordsExecutor	用于处理数据的线程池资源。无特殊需求，建议使用默认的配置。 **说明** • 自定义上述线程池时，线程池中的线程数要和Tunnel中的Channel数尽可能一致，此时可以保障每个Channel都能很快的分配到计算资源（CPU）。 • 在默认线程池配置中，为了保证吞吐量，表格存储进行了如下操作： ◦ 默认预先分配32个核心线程，以保障数据较小时（Channel数较少时）的实时吞吐量。 ◦ 工作队列的大小适当调小，当在用户数据量比较大（Channel数较多）时，可以更快触发线程池新建线程的策略，及时弹起更多的计算资源。 ◦ 设置了默认的线程保活时间（默认为60s），当数据量降下后，可以及时回收线程资源。
内存控制	maxChannelParallel	读取和处理数据的最大Channel并行度，可用于内存控制。 默认值为-1，表示不限制最大并行度。 **说明** 仅Java SDK 5.10.0及以上版本支持此功能。
最大退避时间配置	maxRetryIntervalInMillis	Tunnel的最大退避时间基准值，最大退避时间在此基准值附近随机变化，具体范围为0.75*maxRetryIntervalInMillis~1.25*maxRetryIntervalInMillis。 默认值为2000ms，最小值为200ms。 **说明** • 仅Java SDK 5.4.0及以上版本支持此功能。 • Tunnel对于数据量较小的情况（单次拉取小于900 KB或500条）会进行一定时间的指数退避，直至达到最大退避时间。

附录：完整代码

```
import com.alicloud.openservices.tablestore.TunnelClient;
```

```
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelRequest;
import com.alicloud.openservices.tablestore.model.tunnel.CreateTunnelResponse;
import com.alicloud.openservices.tablestore.model.tunnel.TunnelType;
import com.alicloud.openservices.tablestore.tunnel.worker.IChannelProcessor;
import com.alicloud.openservices.tablestore.tunnel.worker.ProcessRecordsInput;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorker;
import com.alicloud.openservices.tablestore.tunnel.worker.TunnelWorkerConfig;
public class TunnelQuickStart {
    private static class SimpleProcessor implements IChannelProcessor {
        @Override
        public void process(ProcessRecordsInput input) {
            System.out.println("Default record processor, would print records count");
            System.out.println(
                //NextToken用于Tunnel Client的翻页。
                String.format("Process %d records, NextToken: %s", input.getRecords().size(
), input.getNextToken()));
            try {
                //模拟消费处理。
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void shutdown() {
            System.out.println("Mock shutdown");
        }
    }
    public static void main(String[] args) throws Exception {
        //1.初始化Tunnel Client。
        final String endPoint = "";
        final String accessKeyId = "";
        final String accessKeySecret = "";
        final String instanceName = "";
        TunnelClient tunnelClient = new TunnelClient(endPoint, accessKeyId, accessKeySecret
, instanceName);
        //2.创建新通道（此步骤需要提前创建一张测试表，可以使用SyncClient的createTable或者使用官网控制台等方式创建）。
        final String tableName = "testTable";
        final String tunnelName = "testTunnel";
        CreateTunnelRequest request = new CreateTunnelRequest(tableName, tunnelName, Tunnel
Type.BaseAndStream);
        CreateTunnelResponse resp = tunnelClient.createTunnel(request);
        //tunnelId用于后续TunnelWorker的初始化，该值也可以通过ListTunnel或者DescribeTunnel获取。
        String tunnelId = resp.getTunnelId();
        System.out.println("Create Tunnel, Id: " + tunnelId);
        //3.用户自定义数据消费Callback，开始自动化的数据消费。
        //TunnelWorkerConfig中有更多的高级参数。
        TunnelWorkerConfig config = new TunnelWorkerConfig(new SimpleProcessor());
        TunnelWorker worker = new TunnelWorker(tunnelId, tunnelClient, config);
        try {
            worker.connectAndWorking();
        } catch (Exception e) {
            e.printStackTrace();
            worker.shutdown();
```

```
            workerExecutor.shutdown();
            tunnelClient.shutdown();
        }
    }
}
```

3.10.3. 创建通道

CreateTunnel操作为某张数据表创建一个通道，一张数据表上可以创建多个通道。在创建通道时需要指定数据表名称、通道名称和通道类型。

请求参数

参数	说明
tableName	创建通道的数据表名称。
tunnelName	通道的名称。
tunnelType	通道的类型，包括如下取值： • BaseData：全量类型。只能消费处理全量数据。 • Stream：增量类型。只能消费处理增量数据。 • BaseAndStream：全量加增量类型。全量数据消费处理完成后，再消费处理增量数据。

响应参数

参数	说明
TunnelId	通道的ID。
ResponseInfo	返回的一些其它字段。
RequestId	当次请求的Request ID。

示例

- 示例1

  创建全量类型的通道。

```
//支持创建三种类型的通道TunnelType.BaseData（全量）、TunnelType.Stream（增量）和TunnelType.BaseAndStream（全量加增量）。
//本示例为创建全量类型的通道，如果需要创建其它类型的通道，则将CreateTunnelRequest中的TunnelType设置为相应的类型。
private static void createTunnel(TunnelClient client, String tunnelName) {
    CreateTunnelRequest request = new CreateTunnelRequest(TableName, tunnelName, TunnelType.BaseData);
    CreateTunnelResponse resp = client.createTunnel(request);
    System.out.println("RequestId: " + resp.getRequestId());
    System.out.println("TunnelId: " + resp.getTunnelId());
}
```

- 示例2

  创建增量或者全量加增量类型的通道，并指定读取的增量数据时间范围。

```
//创建增量或者全量加增量类型的通道，指定起始时间戳或结束时间戳，表示读取的增量数据时间范围。对于全量类型的通道，StreamTunnelConfig的配置不生效。
private static void createStreamTunnelByOffset(TunnelClient client,String tableName,String tunnelName){
    CreateTunnelRequest createTunnelRequest = new CreateTunnelRequest(tableName,tunnelName, TunnelType.Stream);//创建增量类型通道。
    //CreateTunnelRequest createTunnelRequest = new CreateTunnelRequest(tableName,tunnelName, TunnelType.BaseAndStream);//创建全量加增量类型通道。
    StreamTunnelConfig streamTunnelConfig = new StreamTunnelConfig();
    /*
        指定增量数据的起始时间戳（startTime）和结束时间戳（endTime）。单位为毫秒，取值范围为[CurrentSystemTime - StreamExpiration + 5 minute, CurrentSystemTime)。
        其中CurrentSystemTime为当前系统时间的毫秒单位时间戳，StreamExpiration为增量日志过期时间的毫秒单位时间戳，最大值为7天，您可以在表属性中设置StreamExpiration。
        结束时间戳的取值必须大于起始时间戳。
     */
    streamTunnelConfig.setStartOffset(startTime);
    streamTunnelConfig.setEndOffset(endTime);
    createTunnelRequest.setStreamTunnelConfig(streamTunnelConfig);
    CreateTunnelResponse resp = client.createTunnel(createTunnelRequest);
    System.out.println("RequestId: " + resp.getRequestId());
    System.out.println("TunnelId: " + resp.getTunnelId());
}
```

3.10.4. 获取表内的通道信息

ListTunnel操作列举某个数据表内通道的具体信息。

请求参数

参数	说明
TableName	列举通道信息的数据表名称。

响应参数

参数	说明
List<TunnelInfo>	通道信息的列表，包含如下信息： • TunnelId：通道的ID。 • TunnelType：通道的类型，包括全量（BaseData）、增量（Stream）和全量加增量（BaseAndStream）三种。 • TableName：该通道所在的数据表名称。 • InstanceName：该通道所在的实例名称。 • Stage：该通道所处的阶段，包括初始化(InitBaseDataAndStreamShard)、全量处理（ProcessBaseData）和增量处理（ProcessStream）三种。 • Expired：数据是否超期。 如果该值返回true，请及时通过钉钉联系表格存储技术支持。
ResponseInfo	返回的一些其它字段。
RequestId	当次请求的Request ID。

示例

```
private static void listTunnel(TunnelClient client, String tableName) {
    ListTunnelRequest request = new ListTunnelRequest(tableName);
    ListTunnelResponse resp = client.listTunnel(request);
    System.out.println("RequestId: " + resp.getRequestId());
    for (TunnelInfo info : resp.getTunnelInfos()) {
        System.out.println("TunnelInfo::::::");
        System.out.println("\tTunnelName: " + info.getTunnelName());
        System.out.println("\tTunnelId: " + info.getTunnelId());
        //通道的类型，包括全量(BaseData)、增量(Stream)和全量加增量(BaseAndStream)三种。
        System.out.println("\tTunnelType: " + info.getTunnelType());
        System.out.println("\tTableName: " + info.getTableName());
        System.out.println("\tInstanceName: " + info.getInstanceName());
        //通道所处的阶段，包括初始化(InitBaseDataAndStreamShard)、全量处理(ProcessBaseData)和增
量处理(ProcessStream)三类。
        System.out.println("\tStage: " + info.getStage());
        //数据是否超期。如果该值返回true，请及时通过钉钉联系表格存储技术支持。
        System.out.println("\tExpired: " + info.isExpired());
    }
}
```

3.10.5. 获取通道的具体信息

DescribeTunnel操作描述某个通道里的具体Channel信息。目前一个Channel对应TableStore Stream接口的一个数据分片。

请求参数

参数	说明
TableName	需要获取通道信息的数据表名称。
TunnelName	通道的名称。

响应参数

参数	说明
TunnelConsumePoint	通道消费增量数据的最新时间点，其值等于Tunnel中消费最慢的Channel的时间点，默认值为1970年1月1日（UTC）。
TunnelInfo	通道信息的列表，包含如下信息： • TunnelId：通道的ID。 • TunnelType：通道的类型，包括全量（BaseData）、增量（Stream）和全量加增量（BaseAndStream）三种。 • TableName：该通道所在的数据表名称。 • InstanceName：该通道所在的实例名称。 • Stage：该通道所处的阶段，包括初始化（InitBaseDataAndStreamShard），全量处理（ProcessBaseData）和增量处理（ProcessStream）三种。 • Expired：数据是否超期。 如果该值返回true，请及时通过钉钉联系表格存储技术支持。
List<ChannelInfo>	通道中的Channel信息列表，包含如下信息： • ChannelId：Channel对应的ID。 • ChannelType：Channel的类型，包括全量（BaseData）和增量（Stream）两种。 • ChannelStatus：Channel的状态，包括等待（WAIT）、打开（OPEN）、关闭中（CLOSING）、关闭（CLOSE）和结束（TERMINATED）五种。 • ClientId：通道客户端的ID标识，默认由客户端主机名（可以在TunnelWorkerConfig中自定义）和随机串拼接而成。 • ChannelConsumePoint：Channel消费增量数据的最新时间点，默认值为1970年1月1日（UTC），全量类型无此概念。 • ChannelCount：Channel同步的数据条数。
ResponseInfo	返回的一些其它字段。
RequestId	当次请求的Request ID。

示例

```
//数据消费时间位点（ConsumePoint）和RPO（Recovery Point Objective）为增量类型专用属性值，全量类型无此概念。
//Tunnel增量：TunnelInfo中Stage的值为ProcessStream；Channel增量：ChannelInfo中的ChannelType为Stream。
private static void describeTunnel(TunnelClient client, String tableName, String tunnelName) {
    DescribeTunnelRequest request = new DescribeTunnelRequest(tableName, tunnelName);
    DescribeTunnelResponse resp = client.describeTunnel(request);
    System.out.println("RequestId: " + resp.getRequestId());
    //通道消费增量数据的最新时间点，其值等于Tunnel中消费最慢的Channel的时间点，默认值为1970年1月1日（UTC）。
    System.out.println("TunnelConsumePoint: " + resp.getTunnelConsumePoint());
    System.out.println("TunnelInfo: " + resp.getTunnelInfo());
    for (ChannelInfo ci : resp.getChannelInfos()) {
        System.out.println("ChannelInfo::::::");
        System.out.println("\tChannelId: " + ci.getChannelId());
        //Channel的类型，包括BaseData（全量）和增量（Stream）两种。
        System.out.println("\tChannelType: " + ci.getChannelType());
        //客户端的ID标识，默认由客户端主机名和随机串拼接而成。
        System.out.println("\tClientId: " + ci.getClientId());
        //Channel消费增量数据的最新时间点。
        System.out.println("\tChannelConsumePoint: " + ci.getChannelConsumePoint());
        //Channel同步的数据条数。
        System.out.println("\tChannelCount: " + ci.getChannelCount());
    }
}
```

3.10.6. 删除通道

DeleteTunnel操作为某张数据表删除一个通道，删除时需要指定数据表名称和通道名称。

请求参数

参数	说明
TableName	需要删除通道的数据表名称。
TunnelName	通道的名称。

响应参数

参数	说明
ResponseInfo	返回的一些其它字段。
RequestId	当次请求的Request ID。

示例

```
private static void deleteTunnel(TunnelClient client, String tableName, String tunnelName)
{
    DeleteTunnelRequest request = new DeleteTunnelRequest(tableName, tunnelName);
    DeleteTunnelResponse resp = client.deleteTunnel(request);
    System.out.println("RequestId: " + resp.getRequestId());
}
```

3.11. SQL查询

3.11.1. 概述

本文介绍了通过表格存储Java SDK使用SQL语句时支持的操作。

> **注意** 表格存储Java SDK从5.13.0版本开始支持SQL查询功能。使用SQL查询功能时，请确保获取了正确的Java SDK版本。关于Java SDK历史迭代版本的更多信息，请参见Java SDK历史迭代版本。

操作	说明
创建表及映射关系	通过create table语句创建表及映射关系。如果表存在，则只创建映射关系；如果表不存在，则同时创建同名表。 **注意** 目前执行create table语句暂不支持创建表，只能为已有表创建映射关系。
删除映射关系	通过drop mapping table语句删除表的映射关系。
列出表名称列表	通过show tables语句列出当前数据库中的表名称列表。
查询表的描述信息	通过describe语句查询表的描述信息，例如字段名称、字段类型等。
查询索引描述信息	通过show index语句查询表的索引描述信息。
查询数据	通过select语句查询表中数据。

3.11.2. 创建表及映射关系

通过create table语句创建表及映射关系。如果表存在，则只创建映射关系；如果表不存在，则同时创建同名表。

> **说明** 关于create table语句的更多信息，请参见创建表及映射关系。

前提条件

已初始化Client。具体操作，请参见初始化。

参数

参数	说明
query	SQL语句，请根据所需功能进行设置。

示例

使用 `create table test_table (pk varchar(1024), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primary key(pk))` 语句创建test_table表的映射关系。

```
private static void createTable(SyncClient client) {
    // 创建SQL请求。
    SQLQueryRequest request = new SQLQueryRequest("create table test_table (pk varchar(1024
), long_value bigint, double_value double, string_value mediumtext, bool_value bool, primar
y key(pk))");
    // 获取SQL的响应结果。
    SQLQueryResponse response = client.sqlQuery(request);
}
```

3.11.3. 删除映射关系

通过drop mapping table语句删除表的映射关系。

> **说明** 关于drop mapping table语句的更多信息，请参见删除映射关系。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建映射关系。具体操作，请参见创建表及映射关系。

参数

参数	说明
query	SQL语句，请根据所需功能进行设置。

示例

使用drop mapping table test_table语句删除test_table表的映射关系。

```
private static void dropMappingTable(SyncClient client) {
    // 创建SQL请求。
    SQLQueryRequest request = new SQLQueryRequest("drop mapping table test_table");
    // 获取SQL的响应结果。
    SQLQueryResponse response = client.sqlQuery(request);
}
```

3.11.4. 列出表名称列表

通过show tables语句列出当前数据库中的表名称列表。

> **说明** 关于show tables语句的更多信息，请参见列出表名称列表。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建映射关系。具体操作，请参见创建表及映射关系。

参数

参数	说明
query	SQL语句，请根据所需功能进行设置。

示例

使用 `show tables` 语句列出表名称列表。

```
private static void showTable(SyncClient client) {
    // 创建SQL请求。
    SQLQueryRequest request = new SQLQueryRequest("show tables");
    // 获取SQL的响应结果。
    SQLQueryResponse response = client.sqlQuery(request);
    // 获取SQL返回值的Schema。
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table schema: " + tableMeta.getSchema());
    // 通过SQL ResultSet遍历获取SQL的返回结果。
    System.out.println("response resultset:");
    SQLResultSet resultSet = response.getSQLResultSet();
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0));
    }
    // 通过SQLUtils函数解析Response获取表名称列表。
    System.out.println("response sqlutils resultset:");
    List<String> tables = SQLUtils.parseShowTablesResponse(response);
    for (String table : tables) {
        System.out.print(table + ", ");
    }
}
```

返回结果示例如下：

```
response table schema: [Tables_in_$instanceName:STRING]
response resultset:
test_table
response sqlutils resultset: test_table,
```

3.11.5. 查询表的描述信息

通过describe语句查询表的描述信息，例如字段名称、字段类型等。

> **说明** 关于describe语句的更多信息，请参见查询表的描述信息。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建映射关系。具体操作，请参见创建表及映射关系。

参数

参数	说明
query	SQL语句，请根据所需功能进行设置。

示例

使用 `describe test_table` 语句查询test_table表的描述信息。

```
private static void getTableDesc(SyncClient client) {
    // 创建SQL请求。
    SQLQueryRequest request = new SQLQueryRequest("describe test_table");
    // 获取SQL的响应结果。
    SQLQueryResponse response = client.sqlQuery(request);
    // 获取SQL返回值的Schema。
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table schema: " + tableMeta.getSchema());
    // 通过SQL ResultSet遍历获取SQL的返回结果。
    System.out.println("response resultset:");
    SQLResultSet resultSet = response.getSQLResultSet();
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0) + ", " + row.getString(1) + ", " +
                           row.getString(2) + ", " + row.getString(3) + ", " +
                           row.getString(4) + ", " + row.getString(5));
    }
}
```

返回结果示例如下：

```
response table schema: [Field:STRING, Type:STRING, Null:STRING, Key:STRING, Default:STRING,
Extra:STRING]
response resultset:
pk, varchar(1024), NO, PRI, null,
long_value, bigint(20), YES, , null,
double_value, double, YES, , null,
string_value, mediumtext, YES, , null,
bool_value, tinyint(1), YES, , null,
```

3.11.6. 查询索引描述信息

通过show index语句查询表的索引描述信息。

> **说明** 关于show index语句的更多信息，请参见查询索引描述信息。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建映射关系。具体操作，请参见创建表及映射关系。

参数

参数	说明
query	SQL语句，请根据所需功能进行设置。

示例

使用 `show index in test_table` 语句查询test_table表的索引描述信息。

```
private static void dropMappingTable(SyncClient client) {
    // 创建SQL请求。
    SQLQueryRequest request = new SQLQueryRequest("show index in test_table");
    // 获取SQL的响应结果。
    SQLQueryResponse response = client.sqlQuery(request);
    // 获取SQL返回值的Schema。
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table schema: " + tableMeta.getSchema());
    // 通过SQL ResultSet遍历获取SQL的返回结果。
    System.out.println("response resultset:");
    SQLResultSet resultSet = response.getSQLResultSet();
    while (resultSet.hasNext()) {
        SQLRow row = showIndexResultSet.next();
        System.out.println(row.getString("Table") + ", " + row.getLong("Non_unique") + ", "
+
                           row.getString("Key_name") + ", " + row.getLong("Seq_in_index") +
", " +
                           row.getString("Column_name") + ", " + row.getString("Index_type"
) );
    }
}
```

返回结果示例如下：

```
response table schema: [Table:STRING, Non_unique:INTEGER, Key_name:STRING, Seq_in_index:INTEGER, Column_name:STRING, Is_defined_column:STRING, Collation:STRING, Cardinality:INTEGER, Sub_part:INTEGER, Packed:STRING, Null:STRING, Index_type:STRING, Comment:STRING, Index_comment:STRING, Visible:STRING, Expression:STRING]
response resultset:
test_table, 0, PRIMARY, 1, pk,
test_table, 1, test_table_index, 1, pk, SearchIndex
test_table, 1, test_table_index, 2, bool_value, SearchIndex
test_table, 1, test_table_index, 3, double_value, SearchIndex
test_table, 1, test_table_index, 4, long_value, SearchIndex
test_table, 1, test_table_index, 5, string_value, SearchIndex
```

3.11.7. 查询数据

通过select语句查询表中数据。

> **说明** 关于select语句的更多信息，请参见查询数据。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建映射关系。具体操作，请参见创建表及映射关系。

参数

参数	说明
query	SQL语句，请根据所需功能进行设置。

示例

使用 `select pk, long_value, double_value, string_value, bool_value from test_table limit 20` 语句查询test_table表中数据且最多返回20行数据。系统会返回查询语句的请求类型、返回值Schema、返回结果等信息。

```
private static void queryData(SyncClient client) {
    // 创建SQL请求。
    SQLQueryRequest request = new SQLQueryRequest("select pk, long_value, double_value, str
ing_value, bool_value from test_table limit 20");
    // 获取SQL的响应结果。
    SQLQueryResponse response = client.sqlQuery(request);
    // 获取SQL的请求类型。
    System.out.println("response type: " + response.getSQLStatementType());
    // 获取SQL返回值的Schema。
    SQLTableMeta tableMeta = response.getSQLResultSet().getSQLTableMeta();
    System.out.println("response table meta: " + tableMeta.getSchema());
    // 获取SQL的返回结果。
    SQLResultSet resultSet = response.getSQLResultSet();
    System.out.println("response resultset:");
    while (resultSet.hasNext()) {
        SQLRow row = resultSet.next();
        System.out.println(row.getString(0) + ", " + row.getString("pk") + ", " +
                           row.getLong(1) + ", " + row.getLong("long_value") + ", " +
                           row.getDouble(2) + ", " + row.getDouble("double_value") + ", " +
                           row.getString(3) + ", " + row.getString("string_value") + ", " +
                           row.getBoolean(4) + ", " + row.getBoolean("bool_value"));
    }
}
```

返回结果示例如下：

```
response type: SQL_SELECT
response table meta: [pk:STRING, long_value:INTEGER, double_value:DOUBLE, string_value:STRI
NG, bool_value:BOOLEAN]
response resultset:
binary_null, binary_null, 1, 1, 1.0, 1.0, a, a, false, false
bool_null, bool_null, 1, 1, 1.0, 1.0, a, a, null, null
double_null, double_null, 1, 1, null, null, a, a, true, true
long_null, long_null, null, null, 1.0, 1.0, a, a, true, true
string_null, string_null, 1, 1, 1.0, 1.0, null, null, false, false
```

3.12. 数据湖投递

3.12.1. 创建投递任务

使用CreateDeliveryTask接口创建一个投递任务。通过创建投递任务，您可以将表格存储数据表中的数据投递到OSS Bucket中存储。

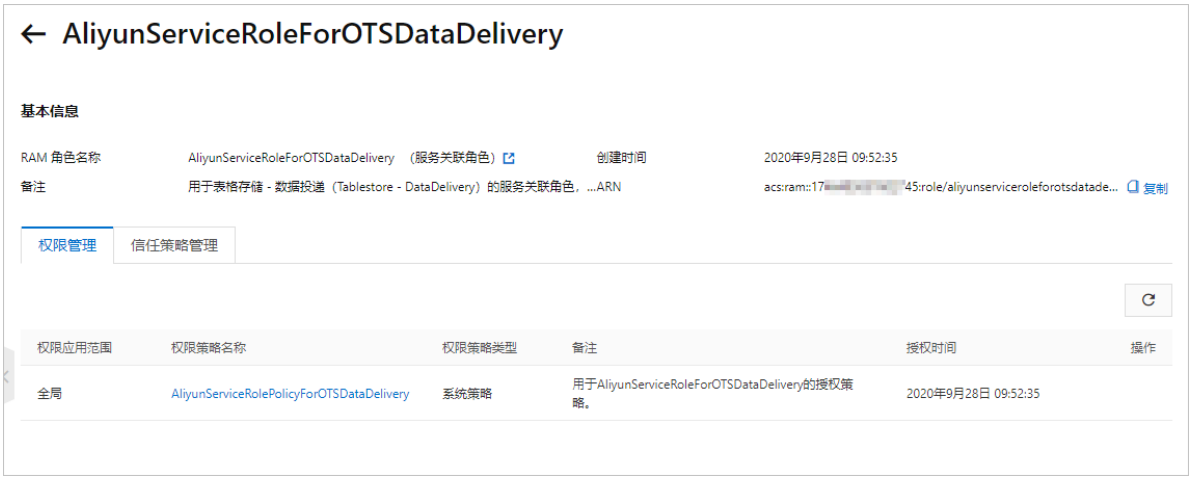
> **注意**
>
> 请确保已安装支持数据湖投递功能的表格存储Java SDK。关于表格存储Java SDK版本的更多信息，请参见Java SDK历史迭代版本。

前提条件

- 已开通OSS服务且在表格存储实例所在地域创建Bucket。具体操作，请参见开通OSS服务。
- 已通过控制台创建表格存储服务关联角色并记录角色的ARN。具体操作，请参见创建投递任务。

  服务关联角色的ARN请通过RAM控制台获取，具体操作如下：

  在**RAM 角色管理**界面，搜索AliyunServiceRoleForOTSDataDelivery后，单击角色名称，在角色详情界面，可以查看和复制角色的ARN信息。



- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。

参数

参数	说明
tableName	数据表名称。
taskName	投递任务名称。 名称只能包含英文小写字母（a~z）、数字和短横线（-），开头和结尾必须为英文小写字母或数字，且长度为3~16字符。

参数	说明

taskConfig	投递任务配置，包括如下选项：
	• ossPrefix：OSS Bucket中的目录前缀，将表格存储的数据投递到该OSS Bucket目录中。投递路径中支持引用$yyyy、$MM、$dd、$HH、$mm五种时间变量。
	  ○ 当投递路径中引用时间变量时，可以按数据的写入时间动态生成OSS目录，实现hive partition naming style的数据时间分区，从而按照时间分区组织OSS中的文件分布。
	  ○ 当投递路径中不引用时间变量时，所有文件会被投递到固定的OSS前缀目录中。
	• ossBucket：OSS Bucket名称。
	• ossEndpoint：OSS Bucket所在地域的服务地址。
	• ossStsRole：表格存储服务关联角色的ARN信息。
	• format：投递的数据的存储以Parquet列存格式存储，数据湖投递默认使用PLAIN编码方式，PLAIN编码方式支持任意类型数据。
	• eventTimeColumn：事件时间列，用于指定按某一列数据的时间进行分区。如果不设置此参数，则按数据写入表格存储的时间进行分区。
	• parquetSchema：指定需要投递的数据列，必须手动配置投递字段的源表字段、目标字段和目标字段类型。
	您可以选择任意字段以任意顺序、名称写入列存文件，OSS的列存数据会按Schema数组中的数据列先后顺序分布。
	注意
	投递数据的字段类型必须与数据源的字段类型匹配，否则会作为脏数据丢弃。字段类型映射详情请参见数据格式映射。

taskType	投递任务的类型，包括如下选项： • INC：表示增量数据投递模式，只同步增量数据。 • BASE：表示全量数据投递模式，一次性全表扫描数据同步。 • BASE_INC（默认）：表示全量&增量数据投递模式，全量数据同步完成后，再同步增量数据。 其中增量数据同步时可以获取最新投递时间和了解当前投递状态。

示例

```
private static void createDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    String taskName = "sampledeliverytask";
    OSSTaskConfig taskConfig = new OSSTaskConfig();
    taskConfig.setOssPrefix("sampledeliverytask/year=$yyyy/month=$MM");
    taskConfig.setOssBucket("datadeliverytest");
    taskConfig.setOssEndpoint("oss-cn-hangzhou.aliyuncs.com");
    taskConfig.setOssStsRole("acs:ram::17************45:role/aliyunserviceroleforotsdatadel
ivery"); //eventColumn为可选配置，指定按某一列数据的时间进行分区。如果不设置此参数，则按数据写入表格
存储的时间进行分区。
    EventColumn eventColumn = new EventColumn("PK1", EventTimeFormat.RFC1123);
    taskConfig.setEventTimeColumn(eventColumn);
    taskConfig.addParquetSchema(new ParquetSchema("PK1", "PK1", DataType.UTF8));
    taskConfig.addParquetSchema(new ParquetSchema("PK2", "PK2", DataType.BOOL));
    taskConfig.addParquetSchema(new ParquetSchema("Col1", "Col1", DataType.UTF8));
    CreateDeliveryTaskRequest request = new CreateDeliveryTaskRequest();
    request.setTableName(tableName);
    request.setTaskName(taskName);
    request.setTaskConfig(taskConfig);
    request.setTaskType(DeliveryTaskType.BASE_INC);
    CreateDeliveryTaskResponse response = client.createDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    System.out.println("create delivery task success");
}
```

3.12.2. 列出投递任务名称

使用ListDeliveryTask接口列出数据表所有的投递任务信息。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建投递任务。使用控制台或SDK的具体操作，请分别参见快速入门或创建投递任务。

请求参数

参数	说明
tableName	数据表名称。

返回参数

参数	说明
tableName	数据表名称，和请求时一致。

参数	说明
taskName	投递任务名称。
taskType	投递任务的类型。

示例

```
public static void listDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    ListDeliveryTaskRequest request = new ListDeliveryTaskRequest(tableName);
    ListDeliveryTaskResponse response = client.listDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    for(DeliveryTaskInfo taskInfo: response.getTaskInfos()) {
        System.out.println("tableName: " + taskInfo.getTableName());
        System.out.println("taskName: " + taskInfo.getTaskName());
        System.out.println("taskType: " + taskInfo.getTaskType().toString());
    }
}
```

3.12.3. 查询投递任务描述信息

使用DescribeDeliveryTask接口查询投递任务描述信息。

请求参数

参数	说明
tableName	数据表名称。
taskName	投递任务名称。

返回参数

参数	说明
taskConfig	投递任务的配置信息。
taskSyncStat	投递任务的同步状态。

参数	说明
taskType	投递任务的类型。

示例

```
public static void describeDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    String taskName  = "sampledeliverytask";
    DescribeDeliveryTaskRequest request = new DescribeDeliveryTaskRequest(tableName, taskNa
me);
    DescribeDeliveryTaskResponse response = client.describeDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    System.out.println("OSSconfig: " + response.getTaskConfig());
    System.out.println("TaskSyncStat: " + response.getTaskSyncStat());
    System.out.println("taskType: " + response.getTaskType());
}
```

3.12.4. 删除投递任务

使用DeleteDeliveryTask接口删除一个投递任务。

参数

参数	说明
tableName	数据表名称。
taskName	投递任务名称。

示例

```
private static void deleteDeliveryTask(SyncClient client) {
    String tableName = "sampleTable";
    String taskName = "sampledeliverytask";
    DeleteDeliveryTaskRequest request = new DeleteDeliveryTaskRequest(tableName, taskName);
    DeleteDeliveryTaskResponse response = client.deleteDeliveryTask(request);
    System.out.println("resquestID: "+ response.getRequestId());
    System.out.println("traceID: " + response.getTraceId());
    System.out.println("delete task delivery success");
}
```

3.13. 时序模型

3.13.1. 概述

表格存储的Java SDK提供了多种时序表级别的功能操作。

功能	描述
创建时序表	创建一张时序表。
列出时序表名称	获取当前实例下所有时序表的名称。
查询时序表描述信息	查询时序表描述信息，例如数据生命周期（Time To Live，简称TTL）配置等。
更新时序表	更新时序表的配置信息。
删除时序表	删除一张时序表。
写入时序数据	批量写入时序数据。
查询时序数据	查询符合指定条件的时序数据。
检索时间线	指定多种条件检索时间线。
更新时间线元数据	批量更新时间线元数据的属性。
删除时间线元数据	批量删除时间线元数据。

3.13.2. 创建时序表

使用CreateTimeseriesTable创建时序表时，需要指定表的配置信息。

前提条件

- 已通过控制台创建实例。具体操作，请参见创建时序模型公测实例。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

注意事项

时序表的名称不能与当前已存在的数据表名称相同。

参数

参数	说明
timeseriesTableName	时序表名。
timeseriesTableOptions	时序表的配置信息，包括如下内容： timeToLive：配置时序表的数据存活时间，单位为秒。如果希望数据永不过期，可以设置为-1。您可以通过UpdateTimeseriesTable接口修改。

示例

创建test_timeseries_table时序表，且该表中数据永不过期。

```
private static void createTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    TimeseriesTableMeta timeseriesTableMeta = new TimeseriesTableMeta(tableName);
    int timeToLive = -1;
    timeseriesTableMeta.setTimeseriesTableOptions(new TimeseriesTableOptions(timeToLive));
    CreateTimeseriesTableRequest request = new CreateTimeseriesTableRequest(timeseriesTable
Meta);
    client.createTimeseriesTable(request);
}
```

3.13.3. 列出时序表名称

使用ListTimeseriesTable接口，您可以获取当前实例下所有时序表的名称。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

示例

获取实例下所有时序表的表名。

```
private static void listTimeseriesTable(TimeseriesClient client) {
    ListTimeseriesTableResponse listTimeseriesTableResponse = client.listTimeseriesTable();
    for (String table : listTimeseriesTableResponse.getTimeseriesTableNames()) {
        System.out.println(table);
    }
}
```

3.13.4. 查询时序表描述信息

使用DescribeTimeseriesTable接口，您可以查询时序表描述信息，例如数据生命周期（Time To Live，简称TTL）配置等。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

参数	说明
timeseriesTableName	时序表名。

示例

查询test_timeseries_table时序表的描述信息。

```
private static void describeTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    DescribeTimeseriesTableResponse describeTimeseriesTableResponse = client.describeTimese
riesTable(new DescribeTimeseriesTableRequest(tableName));
    TimeseriesTableMeta tableMeta = describeTimeseriesTableResponse.getTimeseriesTableMeta(
);
    System.out.println(tableMeta.getTimeseriesTableName()); // 时序表名。
    System.out.println(tableMeta.getStatus()); // 查看时序表状态。
    System.out.println(tableMeta.getTimeseriesTableOptions().getTimeToLive()); // 查看时序表
的TTL配置。
}
```

3.13.5. 更新时序表

使用UpdateTimeseriesTable，您可以更新时序表的配置信息，例如数据生命周期（Time To Live，简称TTL）配置。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

更多信息，请参见创建时序表。

示例

更新test_timeseries_table时序表的TTL为3年。

```
private static void updateTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    UpdateTimeseriesTableRequest updateTimeseriesTableRequest = new UpdateTimeseriesTableRe
quest(tableName);
    updateTimeseriesTableRequest.setTimeseriesTableOptions(new TimeseriesTableOptions(86400
* 365 * 3)); // 更新TTL为3年。
    client.updateTimeseriesTable(updateTimeseriesTableRequest);
    DescribeTimeseriesTableResponse describeTimeseriesTableResponse = client.describeTimese
riesTable(new DescribeTimeseriesTableRequest(tableName));
    TimeseriesTableMeta tableMeta = describeTimeseriesTableResponse.getTimeseriesTableMeta(
);
    System.out.println(tableMeta.getTimeseriesTableOptions().getTimeToLive()); // 查看更新后
时序表的TTL配置。
}
```

3.13.6. 删除时序表

使用DeleteTimeseries接口，您可以删除一张时序表。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。

- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

参数	说明
timeseriesTableName	时序表名。

示例

删除test_timeseries_table时序表。

```
private static void deleteTimeseriesTable(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    DeleteTimeseriesTableRequest deleteTimeseriesTableRequest = new DeleteTimeseriesTableRe
quest(tableName);
    client.deleteTimeseriesTable(deleteTimeseriesTableRequest);
}
```

3.13.7. 写入时序数据

使用PutTimeseriesData接口，您可以批量写入时序数据。一次PutTimeseriesData调用支持写入多行数据。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

一行时序数据（timeseriesRow）包括时间线标识（timeseriesKey）和时间线数据的配置，其中时间线数据包括数据点的时间（timeInUs）和数据点（fields）。详细参数说明请参见下表。

参数	说明
timeseriesKey	时间线标识，包括如下内容： • measurementName：时间线的度量名称。 • dataSource：数据源信息，可以为空。 • tags：时间线的标签信息，为多个字符串的key-value对。
timeInUs	数据点的时间，单位为微秒。
fields	数据点，可以由多个名称（FieldKey）和数据值（FieldValue）对组成。

示例

向test_timeseries_table时序表中写入多个时序数据。

```
private static void putTimeseriesData(TimeseriesClient client) {
    List<TimeseriesRow> rows = new ArrayList<TimeseriesRow>();
    for (int i = 0; i < 10; i++) {
        Map<String, String> tags = new HashMap<String, String>();
        tags.put("region", "hangzhou");
        tags.put("os", "Ubuntu16.04");
        // 通过measurementName、dataSource和tags构建TimeseriesKey。
        TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_" + i, tags);
        // 指定timeseriesKey和timeInUs创建timeseriesRow。
        TimeseriesRow row = new TimeseriesRow(timeseriesKey, System.currentTimeMillis() * 1
000 + i);
        // 增加数据值（field）。
        row.addField("cpu_usage", ColumnValue.fromDouble(10.0));
        row.addField("cpu_sys", ColumnValue.fromDouble(5.0));
        rows.add(row);
    }
    String tableName = "test_timeseries_table";
    PutTimeseriesDataRequest putTimeseriesDataRequest = new PutTimeseriesDataRequest(tableN
ame);
    putTimeseriesDataRequest.setRows(rows);
    // 一次写入多行时序数据。
    PutTimeseriesDataResponse putTimeseriesDataResponse = client.putTimeseriesData(putTimes
eriesDataRequest);
    // 检查是否全部成功。
    if (!putTimeseriesDataResponse.isAllSuccess()) {
        for (PutTimeseriesDataResponse.FailedRowResult failedRowResult : putTimeseriesDataR
esponse.getFailedRows()) {
            System.out.println(failedRowResult.getIndex());
            System.out.println(failedRowResult.getError());
        }
    }
}
```

3.13.8. 查询时序数据

使用GetTimeseriesData接口，您可以查询符合指定条件的时序数据。

前提条件

- 已写入时序数据。具体操作，请参见写入时序数据。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

参数	说明
timeseriesKey	要查询的时间线，包括如下内容： •

参数	说明
timeRange	要查询的时间范围，包括如下内容： • beginTimeInUs：起始时间。 • endTimeInUs：结束时间。
backward	是否按照时间倒序读取数据，可用于获取某条时间线的最新数据。取值范围如下： • true：按照时间倒序读取。 • false（默认）：按照时间正序读取。
fieldsToGet	要获取的数据列列名。如果不指定，则默认获取所有列。 **注意** fieldsToGet中需要指定要获取的每一列的列名和类型。如果类型不匹配，则读取不到对应列的数据。
limit	本次最多返回的行数。 **说明** limit仅限制最多返回的行数，在满足条件行数大于limit时，也可能由于扫描数据量等限制导致返回行数少于limit条，此时可以通过nextToken继续获取后面的行。
nextToken	如果一次查询仅返回了部分符合条件的行，此时response中会包括nextToken，可在下一次请求中指定nextToken用来继续读取数据。

示例

查询test_timeseries_table时序表中满足指定条件的时序数据。

```
private static void getTimeseriesData(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    GetTimeseriesDataRequest getTimeseriesDataRequest = new GetTimeseriesDataRequest(tableN
ame);
    Map<String, String> tags = new HashMap<String, String>();
    tags.put("region", "hangzhou");
    tags.put("os", "Ubuntu16.04");
    // 通过measurementName、dataSource和tags构建TimeseriesKey。
    TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_0", tags);
    getTimeseriesDataRequest.setTimeseriesKey(timeseriesKey);
    // 指定时间范围。
    getTimeseriesDataRequest.setTimeRange(0, (System.currentTimeMillis() + 60 * 1000) * 100
0);
    // 限制返回行数。
    getTimeseriesDataRequest.setLimit(10);
    // 设置是否倒序读取数据，可不设置，默认值为false。如果设置为true，则倒序读取数据。
    getTimeseriesDataRequest.setBackward(false);
    // 设置获取部分数据列，可不设置，默认获取全部数据列。
    getTimeseriesDataRequest.addFieldToGet("string_1", ColumnType.STRING);
    getTimeseriesDataRequest.addFieldToGet("long_1", ColumnType.INTEGER);
    GetTimeseriesDataResponse getTimeseriesDataResponse = client.getTimeseriesData(getTimes
eriesDataRequest);
    System.out.println(getTimeseriesDataResponse.getRows().size());
    if (getTimeseriesDataResponse.getNextToken() != null) {
        // 如果nextToken不为空，可以发起下一次请求。
        getTimeseriesDataRequest.setNextToken(getTimeseriesDataResponse.getNextToken());
        getTimeseriesDataResponse = client.getTimeseriesData(getTimeseriesDataRequest);
        System.out.println(getTimeseriesDataResponse.getRows().size());
    }
}
```

3.13.9. 检索时间线

使用QueryTimeseriesMeta接口，您可以指定多种条件检索时间线。

前提条件

- 已写入时序数据。具体操作，请参见写入时序数据。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

metaQueryCondition表示检索时间线的条件，包括compositeMetaQueryCondition（组合条件）、measurementMetaQueryCondition（度量名称条件）、dataSourceMetaQueryCondition（数据源条件）、tagMetaQueryCondition（标签条件）、attributeMetaQueryCondition（属性条件）和updateTimeMetaQueryCondition（更新时间条件）。详细参数说明请参见下表。

参数	说明

参数	说明
compositeMetaQueryCondition	组合条件，包括如下内容： • operator：逻辑运算符，可选AND、OR、NOT。 • subConditions：子条件列表，通过operator组成复杂查询条件。
measurementMetaQueryCondition	度量名称条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • value：要匹配的度量名称值，类型为字符串。
dataSourceMetaQueryCondition	数据源条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • value：要匹配的数据源值，类型为字符串。
tagMetaQueryCondition	标签条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • value：要匹配的标签值，类型为字符串。
attributeMetaQueryCondition	时间线元数据的属性条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • attributeName：属性名称，类型为字符串。 • value：属性值，类型为字符串。
updateTimeMetaQueryCondition	时间线元数据的更新时间条件，包括如下内容： • operator：关系运算符，可选=、!=、>、>=、<、<=。 • timeInUs：时间线元数据更新时间的时间戳，单位为微秒。

示例

查询test_timeseries_table时序表中度量名称为cpu，标签中含有os标签且标签前缀为"Ubuntu"的所有时间线。

```
private static void queryTimeseriesMeta(TimeseriesClient client) {
    String tableName = "test_timeseries_table";
    QueryTimeseriesMetaRequest queryTimeseriesMetaRequest = new QueryTimeseriesMetaRequest(
tableName);
    // 查询度量名称为cpu，标签中含有os标签且前缀为"Ubuntu"的所有时间线。即measurement_name="cpu" a
nd have_prefix(os, "Ubuntu")
    CompositeMetaQueryCondition compositeMetaQueryCondition = new CompositeMetaQueryConditi
on(MetaQueryCompositeOperator.OP_AND);
    compositeMetaQueryCondition.addSubCondition(new MeasurementMetaQueryCondition(MetaQuery
SingleOperator.OP_EQUAL, "cpu"));
    compositeMetaQueryCondition.addSubCondition(new TagMetaQueryCondition(MetaQuerySingleOp
erator.OP_PREFIX, "os", "Ubuntu"));
    queryTimeseriesMetaRequest.setCondition(compositeMetaQueryCondition);
    queryTimeseriesMetaRequest.setGetTotalHits(true);
    QueryTimeseriesMetaResponse queryTimeseriesMetaResponse = client.queryTimeseriesMeta(qu
eryTimeseriesMetaRequest);
    System.out.println(queryTimeseriesMetaResponse.getTotalHits());
    for (TimeseriesMeta timeseriesMeta : queryTimeseriesMetaResponse.getTimeseriesMetas())
{
        System.out.println(timeseriesMeta.getTimeseriesKey().getMeasurementName());
        System.out.println(timeseriesMeta.getTimeseriesKey().getDataSource());
        System.out.println(timeseriesMeta.getTimeseriesKey().getTags());
        System.out.println(timeseriesMeta.getAttributes());
        System.out.println(timeseriesMeta.getUpdateTimeInUs());
    }
}
```

3.13.10. 更新时间线元数据

使用UpdateTimeseriesMeta接口，您可以批量更新时间线元数据的属性。一次UpdateTimeseriesMeta调用支持更新多个时间线的元数据。

前提条件

- 已写入时序数据。具体操作，请参见写入时序数据。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

timeseriesMeta表示一个时间线元数据，每个timeseriesMeta包括timeseriesKey和attributes。详细参数说明请参见下表，

参数	描述
timeseriesKey	时间线标识。
attributes	时间线的属性信息，内容为字符串类型的key-value对。

示例

检索test_timeseries_table时序表中更新指定条件时间线的属性信息。

```
private static void updateTimeseriesMeta(TimeseriesClient client) {
    List<TimeseriesMeta> timeseriesMetaList = new ArrayList<TimeseriesMeta>();
    for (int i = 0; i < 10; i++) {
        Map<String, String> tags = new HashMap<String, String>();
        tags.put("region", "hangzhou");
        tags.put("os", "Ubuntu16.04");
        // 构造TimeseriesKey。
        TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_" + i, tags);
        TimeseriesMeta meta = new TimeseriesMeta(timeseriesKey);
        // 设置时间线的属性值（attributes）。
        Map<String, String> attrs = new HashMap<String, String>();
        attrs.put("status", "online");
        meta.setAttributes(attrs);
        timeseriesMetaList.add(meta);
    }
    String tableName = "test_timeseries_table";
    UpdateTimeseriesMetaRequest updateTimeseriesMetaRequest = new UpdateTimeseriesMetaReque
st(tableName);
    updateTimeseriesMetaRequest.setMetas(timeseriesMetaList);
    UpdateTimeseriesMetaResponse updateTimeseriesMetaResponse = client.updateTimeseriesMeta
(updateTimeseriesMetaRequest);
    // 检查是否全部成功。
    if (!updateTimeseriesMetaResponse.isAllSuccess()) {
        for (UpdateTimeseriesMetaResponse.FailedRowResult failedRowResult : updateTimeserie
sMetaResponse.getFailedRows()) {
            System.out.println(failedRowResult.getIndex());
            System.out.println(failedRowResult.getError());
        }
    }
}
```

3.13.11. 删除时间线元数据

使用DeleteTimeseriesMeta接口，您可以批量删除时间线元数据。

前提条件

- 已写入时序数据。具体操作，请参见写入时序数据。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

timeseriesKey用于标识一个时间线，您可以传入多个timeseriesKey来删除多条时间线的元数据。详细参数说明请参见下表。

参数	描述
timeseriesKey	时间线标识，包括如下内容： • measurementName：时间线的度量名称。 • dataSource：数据源信息，可以为空。 • tags：时间线的标签信息，为多个字符串的key-value对。

示例

删除test_timeseries_table时序表中的部分时间线。

```
private static void deleteTimeseriesMeta(TimeseriesClient client) {
    List<TimeseriesKey> timeseriesKeyList = new ArrayList<TimeseriesKey>();
    for (int i = 0; i < 10; i++) {
        Map<String, String> tags = new HashMap<String, String>();
        tags.put("region", "hangzhou");
        tags.put("os", "Ubuntu16.04");
        // 构造TimeseriesKey。
        TimeseriesKey timeseriesKey = new TimeseriesKey("cpu", "host_" + i, tags);
        timeseriesKeyList.add(timeseriesKey);
    }
    String tableName = "test_timeseries_table";
    DeleteTimeseriesMetaRequest deleteTimeseriesMetaRequest = new DeleteTimeseriesMetaReque
st(tableName);
    deleteTimeseriesMetaRequest.setTimeseriesKeys(timeseriesKeyList);
    DeleteTimeseriesMetaResponse deleteTimeseriesMetaResponse = client.deleteTimeseriesMeta
(deleteTimeseriesMetaRequest);
    // 检查是否全部成功。
    if (!deleteTimeseriesMetaResponse.isAllSuccess()) {
        for (DeleteTimeseriesMetaResponse.FailedRowResult failedRowResult : deleteTimeserie
sMetaResponse.getFailedRows()) {
            System.out.println(failedRowResult.getIndex());
            System.out.println(failedRowResult.getError());
        }
    }
}
```

3.14. 错误处理

本文介绍表格存储Java SDK的错误处理。

方式

表格存储Java SDK目前采用“异常”的方式处理错误，如果调用接口没有抛出异常，则说明操作成功，否则失败。

> **说明** 批量相关接口，例如BatchGetRow和BatchWriteRow不仅需要判断是否有异常，还需要检查每行的状态是否成功，只有全部成功后才能保证整个接口调用是成功的。

异常

表格存储Java SDK中有ClientException和TableStoreException两种异常，都最终继承自RuntimeException。

- ClientException：指SDK内部出现的异常，例如参数设置错误等。
- TableStoreException：指服务器端错误，来自于对服务器错误信息的解析。TableStoreException包含以下几个成员：
  - getHttpStatus()：HTTP返回码，例如200、404等。

- getErrorCode()：表格存储返回的错误类型字符串。
- getRequestId()：用于唯一标识此次请求的UUID。当您无法解决问题时，记录此RequestId并提交工单。

重试

- SDK中出现错误时会自动重试。默认策略是最大重试时长为10s。对流控类错误以及读操作相关的服务端内部错误进行的重试。
- 您也可以通过继承RetryStrategy类实现自定义重试策略，在构造OTSClient对象时，将自定义的重试策略作为参数传入。

目前SDK中已经实现的重试策略如下：

- DefaultRetryStrategy：默认重试策略，只会对读操作重试，重试间隔时间以10 ms指数增长，最大重试时长为10s。
- AlwaysRetryStrategy：对所有类型的请求进行重试，最大重试3次，重试间隔时间以4 ms指数增长，最大重试间隔为1s。

4.Go SDK

4.1. 前言

本文介绍表格存储Go SDK的安装和使用。

- 已了解和开通表格存储服务，请登录表格存储的产品主页进行了解。
- 已创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。

下载及安装

- SDK下载路径请参见SDK包（包含package、源代码和示例）。
- 安装方式请参见安装。

版本

当前最新版本：5.0.2

4.2. 安装

本文介绍Go SDK的安装。

环境准备

安装Go SDK需使用Go 1.4及以上版本。

安装方式

安装命令如下。

```
go get github.com/aliyun/aliyun-tablestore-go-sdk
```

4.3. 初始化

TableStoreClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、读写单行数据、读写多行数据等。如果要使用时序模型，您需要初始化TimeseriesClient。

确定Endpoint

Endpoint是阿里云表格存储服务在各个区域的域名地址，您可以通过以下方式查询Endpoint：

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

> **说明** 关于Endpoint的更多信息，请参见服务地址。

配置密钥

要接入阿里云的表格存储服务，您需要拥有一个有效的访问密钥进行签名认证。目前支持下面三种方式：

- 阿里云账号的AccessKey ID和AccessKey Secret。创建步骤如下：

i. 在阿里云官网注册阿里云账号。
ii. 创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。

- 被授予访问表格存储权限RAM用户的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 使用阿里云账号前往访问控制RAM，创建一个新的RAM用户或者使用已经存在的RAM用户。
  ii. 使用阿里云账号授予RAM用户访问表格存储的权限。
  iii. RAM用户被授权后，即可使用自己的AccessKey ID和AccessKey Secret访问。
- 从STS获取的临时访问凭证。获取步骤如下：
  i. 应用的服务器通过访问RAM/STS服务，获取一个临时的AccessKey ID、AccessKey Secret和SecurityToken发送给使用方。
  ii. 使用方使用上述临时密钥访问表格存储服务。

初始化TableStoreClient

获取到AccessKey ID和AccessKey Secret后，您可以按照如下示例代码初始化TableStoreClient。

- 接口

```
//初始化``TableStoreClient``实例。
//endPoint是表格存储服务的地址（例如'https://instance.cn-hangzhou.ots.aliyun.com:80'），必须以'https://'或'http://'开头。
//accessKeyId是访问表格存储服务的AccessKeyID，通过官方网站申请或通过管理员获取。
//accessKeySecret是访问表格存储服务的AccessKeySecret，通过官方网站申请或通过管理员获取。
//instanceName是要访问的实例名，通过官方网站控制台创建或通过管理员获取。
func NewClient(endPoint, instanceName, accessKeyId, accessKeySecret string, options ...ClientOption) *TableStoreClient
```

- 示例

```
client = tablestore.NewClient("your_instance_endpoint",  "your_instance_name", "your_user_id", "your_user_key")
```

初始化TimeseriesClient

获取到AccessKey ID和AccessKey Secret后，您可以按照如下示例代码初始化TimeseriesClient。

- 接口

```
//初始化``TimeseriesClient``实例。
//endPoint是表格存储服务的地址（例如'https://instance.cn-hangzhou.ots.aliyun.com:80'），必须以'https://'或'http://'开头。
//accessKeyId是访问表格存储服务的AccessKeyID，通过官方网站申请或通过管理员获取。
//accessKeySecret是访问表格存储服务的AccessKeySecret，通过官方网站申请或通过管理员获取。
//instanceName是要访问的实例名，通过官方网站控制台创建或通过管理员获取。
func NewTimeseriesClient(endPoint, instanceName, accessKeyId, accessKeySecret string , options ...TimeseriesClientOption) *TimeseriesClient
```

- 示例

```
timeseriesClient = tablestore.NewTimeseriesClient("your_instance_endpoint",  "your_instance_name", "your_user_ak_id", "your_user_ak_key")
```

4.4. 表

4.4.1. 创建数据表

使用CreateTable接口创建数据表时，需要指定数据表的结构信息和配置信息，高性能实例中的数据表还可以根据需要设置预留读/写吞吐量。创建数据表的同时支持创建一个或者多个索引表。

> **说明**
> - 创建数据表后需要几秒钟进行加载，在此期间对该数据表的读/写数据操作均会失败。请等待数据表加载完毕后再进行数据操作。
> - 创建数据表时必须指定数据表的主键。主键包含1个~4个主键列，每一个主键列都有名称和类型。

前提条件

- 已通过控制台创建实例。具体操作，请参见创建实例。
- 已初始化Client。具体操作，请参见初始化。

接口

```
//说明：根据指定的表结构信息创建数据表。
//request是CreateTableRequest类的实例，它包含TableMeta和TableOption以及ReservedThroughput。
//请参见TableMeta类的文档。
//当创建一个数据表后，通常需要等待几秒钟时间使partition load完成，才能进行各种操作。
//返回：CreateTableResponse
      CreateTable(request *CreateTableRequest) (*CreateTableResponse, error)
```

参数

参数	说明
TableMeta	数据表的结构信息，包括如下内容： • TableName：数据表名称。 • PrimaryKey：数据表的主键定义。更多信息，请参见主键和属性。 **说明** 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。 ◦ 表格存储可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体请参见表格存储数据模型和查询操作。 ◦ 第一列主键作为分区键。分区键相同的数据会存放在同一个分区内，所以相同分区键下最好不要超过10 GB以上数据，否则会导致单分区过大，无法分裂。另外，数据的读/写访问最好在不同的分区键上均匀分布，有利于负载均衡。 • DefinedColumns：预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。

参数	说明
TableOption	数据表的配置信息。更多信息，请参见数据版本和生命周期。 配置信息包括如下内容： • TimeToAlive：数据生命周期，即数据的过期时间。当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 创建数据表时，如果希望数据永不过期，可以设置数据生命周期为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 单位为秒。 **说明** 如果需要使用索引，则数据生命周期必须设置为-1（数据永不过期）。 • MaxVersion：最大版本数，即属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。 **说明** 如果需要使用索引，则最大版本数必须设置为1。 • DeviationCellVersionInSec：有效版本偏差，即写入数据的时间戳与系统当前时间的偏差允许最大值。只有当写入数据所有列的版本号与写入时时间的差值在数据有效版本偏差范围内，数据才能成功写入。 属性列的有效版本范围为**[数据写入时间-有效版本偏差，数据写入时间+有效版本偏差)**。 创建数据表时，如果未设置有效版本偏差，系统会使用默认值86400；创建数据表后，可以通过UpdateTable接口动态修改有效版本偏差。 单位为秒。
ReservedThroughput	为数据表配置预留读吞吐量或预留写吞吐量。 容量型实例中的数据表的预留读/写吞吐量只能设置为0，不允许预留。 默认值为0，即完全按量计费。 单位为CU。 • 当预留读吞吐量或预留写吞吐量大于0时，表格存储会根据配置为数据表预留相应资源，且数据表创建成功后，将会立即按照预留吞吐量开始计费，超出预留的部分进行按量计费。更多信息，请参见计费概述。 • 当预留读吞吐量或预留写吞吐量设置为0时，表格存储不会为数据表预留相应资源。

参数	说明
IndexMeta	索引表的结构信息，包括如下内容： • IndexName：索引表名称。 • PrimaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 • DefinedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 • IndexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ◦ 当不设置IndexType或者设置IndexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局二级索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ◦ 当设置IndexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。

示例

- 创建数据表（不带索引）

  创建一个含有2个主键列，预留读/写吞吐量为(0, 0)的数据表。

```
func CreateTableSample(client *tablestore.TableStoreClient, tableName string) {
    createTableRequest := new(tablestore.CreateTableRequest)
    //创建主键列的schema，包括PK的个数、名称和类型。
    //第一个PK列为整数，名称是pk0，此列同时也是分区键。
    //第二个PK列为整数，名称是pk1。
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = tableName
    tableMeta.AddPrimaryKeyColumn("pk0", tablestore.PrimaryKeyType_INTEGER)
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1
    tableOption.MaxVersion = 3
    reservedThroughput := new(tablestore.ReservedThroughput)
    reservedThroughput.Readcap = 0
    reservedThroughput.Writecap = 0
    createTableRequest.TableMeta = tableMeta
    createTableRequest.TableOption = tableOption
    createTableRequest.ReservedThroughput = reservedThroughput
    response, err := client.CreateTable(createTableRequest)
    if (err != nil) {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create table finished")
    }
}
```

详细代码请参见CreateTable@GitHub。

- 创建数据表（带索引且索引类型为全局二级索引）

```
func CreateTableWithGlobalIndexSample(client *tablestore.TableStoreClient, tableName stri
ng) {
    createTableRequest := new(tablestore.CreateTableRequest)
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = tableName
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableMeta.AddPrimaryKeyColumn("pk2", tablestore.PrimaryKeyType_INTEGER)
    tableMeta.AddDefinedColumn("definedcol1", tablestore.DefinedColumn_STRING)
    tableMeta.AddDefinedColumn("definedcol2", tablestore.DefinedColumn_INTEGER)
    indexMeta := new(tablestore.IndexMeta) //新建索引表Meta。
    indexMeta.AddPrimaryKeyColumn("definedcol1") //为索引表添加主键列。设置数据表的definedcol
1列作为索引表的主键。
    indexMeta.AddDefinedColumn("definedcol2") //为索引表添加属性列。设置数据表的definedcol2列
作为索引表的属性列。
    indexMeta.IndexName = "indexSample"
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1 //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据
生命周期必须设置为-1。
    tableOption.MaxVersion = 1 //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。
带索引表的数据表最大版本数必须设置为1。
    reservedThroughput := new(tablestore.ReservedThroughput)
    createTableRequest.TableMeta = tableMeta
    createTableRequest.TableOption = tableOption
    createTableRequest.ReservedThroughput = reservedThroughput
    createTableRequest.AddIndexMeta(indexMeta)
    _, err := client.CreateTable(createTableRequest)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create table finished")
    }
}
```

- 创建数据表（带索引且索引类型为本地二级索引）

```
func CreateTableWithLocalIndexSample(client *tablestore.TableStoreClient, tableName strin
g) {
    createTableRequest := new(tablestore.CreateTableRequest)
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = tableName
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableMeta.AddPrimaryKeyColumn("pk2", tablestore.PrimaryKeyType_INTEGER)
    tableMeta.AddDefinedColumn("definedcol1", tablestore.DefinedColumn_STRING)
    tableMeta.AddDefinedColumn("definedcol2", tablestore.DefinedColumn_INTEGER)
    indexMeta := new(tablestore.IndexMeta) //新建索引表Meta。
    indexMeta.IndexType = IT_LOCAL_INDEX //设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    indexMeta.AddPrimaryKeyColumn("pk1") //为索引表添加主键列。索引表的第一列主键必须与数据表的
第一列主键相同。
    indexMeta.AddPrimaryKeyColumn("definedcol1") //为索引表添加主键列。设置数据表的definedcol
1列作为索引表的主键。
    indexMeta.AddDefinedColumn("definedcol2") //为索引表添加属性列。设置数据表的definedcol2列
作为索引表的属性列。
    indexMeta.IndexName = "indexSample"
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1 //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据
生命周期必须设置为-1。
    tableOption.MaxVersion = 1 //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。
带索引表的数据表最大版本数必须设置为1。
    reservedThroughput := new(tablestore.ReservedThroughput)
    createTableRequest.TableMeta = tableMeta
    createTableRequest.TableOption = tableOption
    createTableRequest.ReservedThroughput = reservedThroughput
    createTableRequest.AddIndexMeta(indexMeta)
    _, err := client.CreateTable(createTableRequest)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create table finished")
    }
}
```

4.4.2. 更新表

使用UpdateTable接口来更新指定表的预留读/写吞吐量。

接口

```
//更改表的tableoptions和reservedthroughput
UpdateTable(request *UpdateTableRequest) (*UpdateTableResponse, error)
```

示例

更新表的最大版本数为5。

```
updateTableReq := new(tablestore.UpdateTableRequest)
updateTableReq.TableName = tableName
updateTableReq.TableOption = new(tablestore.TableOption)
updateTableReq.TableOption.TimeToAlive = -1
updateTableReq.TableOption.MaxVersion = 5
_, err := client.UpdateTable(updateTableReq)
if (err != nil) {
    fmt.Println("failed to update table with error:", err)
} else {
    fmt.Println("update finished")
}
```

详细代码请参见UpdateTable@GitHub。

4.4.3. 列出表名称

使用ListTable接口获取当前实例下已创建的所有表的表名。

> **说明** API说明请参见ListTable。

接口

```
//列出所有的表，如果操作成功，将返回所有表的名称。
ListTable() (*ListTableResponse, error)
```

示例

获取实例下所有表的表名。

```
tables, err := client.ListTable()
if err != nil {
    fmt.Println("Failed to list table")
} else {
    fmt.Println("List table result is")
    for _, table := range (tables.TableNames) {
        fmt.Println("TableName: ", table)
    }
}
```

详细代码请参见ListTable@GitHub。

4.4.4. 预定义列操作

为数据表增加预定义列或删除数据表的预定义列。设置预定义列后，在创建全局二级索引时将预定义列作为索引表的索引列或者属性列。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表。

增加预定义列

使用二级索引时，如果未设置预定义列或者预定义列不满足需求，可以为数据表增加预定义列。

- 参数

参数	说明
TableName	数据表名称。
DefinedColumns	为数据表预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。包含如下设置： ○ Name：预定义列名称。 ○ ColumnType：预定义列的数据类型。

- 示例

  为sampleTable数据表增加预定义列，预定义列分别为definedColumnName01（String类型）、definedColumnName02（INTEGER类型）、definedColumnName03（String类型）。

```
func AddDefinedColumn() {
    client := new(tablestore.TableStoreClient)
    addDefinedColumnRequest := new(tablestore.AddDefinedColumnRequest)
    addDefinedColumnRequest.AddDefinedColumn("definedColumnName01",tablestore.DefinedColumn_STRING)
    addDefinedColumnRequest.AddDefinedColumn("definedColumnName02",tablestore.DefinedColumn_INTEGER)
    addDefinedColumnRequest.AddDefinedColumn("definedColumnName03",tablestore.DefinedColumn_STRING)
    addDefinedColumnRequest.TableName = "sampleTable"
    _, err := client.AddDefinedColumn(addDefinedColumnRequest)
    if (err != nil) {
        fmt.Println("Failed to Add DefinedColumn with error:", err)
    } else {
        fmt.Println("Add DefinedColumn finished")
    }
}
```

删除预定义列

删除数据表上不需要的预定义列。

- 参数

参数	说明
TableName	数据表名称。
DefinedColumns	预定义列名称。

- 示例

删除sampleTable数据表的预定义列definedColumnName01和definedColumnName02。

```
func DeleteDefinedColumn(){
    client := new(tablestore.TableStoreClient)
    deleteDefinedColumnRequest := new(tablestore.DeleteDefinedColumnRequest)
    deleteDefinedColumnRequest.DefinedColumns = []string{"definedColumnName01","definedCo
lumnName02"}
    _, err := client.DeleteDefinedColumn(deleteDefinedColumnRequest)
    if (err != nil) {
        fmt.Println("Failed to delete DefinedColumn with error:", err)
    } else {
        fmt.Println("Delete DefinedColumn finished")
    }
}
```

4.4.5. 查询表描述信息

使用DescribeTable接口可以查询指定表的结构、预留读/写吞吐量详情等信息。

> **说明** API说明请参见DescribeTable。

接口

```
//通过表名查询表描述信息。
DescribeTable(request *DescribeTableRequest) (*DescribeTableResponse, error)
```

参数

参数	说明
TableName	表名。

示例

获取表的描述信息。

```
describeTableReq := new(tablestore.DescribeTableRequest)
describeTableReq.TableName = tableName
describ, err := client.DescribeTable(describeTableReq)
if err != nil {
    fmt.Println("failed to update table with error:", err)
} else {
    fmt.Println("DescribeTableSample finished. Table meta:", describ.TableOption.MaxVersion
, describ.TableOption.TimeToAlive)
}
```

详细代码请参见DescribeTable@GitHub。

4.4.6. 删除数据表

使用DeleteTable接口删除当前实例下指定数据表。

> 说明 API说明请参见DeleteTable。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表。
- 已删除数据表上的索引表和多元索引。

接口

```
DeleteTable(request *DeleteTableRequest) (*DeleteTableResponse, error)
```

参数

参数	说明
TableName	数据表名称。

示例

删除指定数据表。

```
deleteReq := new(tablestore.DeleteTableRequest)
deleteReq.TableName = tableName
_, err := client.DeleteTable(deleteReq)
if (err != nil) {
    fmt.Println("Failed to delete table with error:", err)
} else {
    fmt.Println("Delete table finished")
}
```

详细代码请参见DeleteTable@GitHub。

4.4.7. 主键列自增

设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。

前提条件

已初始化Client，详情请参见初始化。

使用方法

1. 创建表时，将非分区键的主键列设置为自增列。

只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号整型。

2. 写入数据时，无需为自增列设置具体值，只需将相应主键指定为自增主键。

   如果需要获取写入数据后系统自动生成的自增列的值，将ReturnType设置为RT_PK，可以在数据写入成功后返回自增列的值。

   查询数据时，需要完整的主键值。通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为RT_PK可以获取完整的主键值。

示例

主键自增列功能主要涉及创建表（CreateTable）和写数据（PutRow、UpdateRow和BatchWriteRow）两类接口。

1. 创建表

   创建表时，只需将自增的主键属性设置为AUTO_INCREMENT。

```
import (
    "fmt"
    "github.com/aliyun/aliyun-tablestore-go-sdk/tablestore"
)
func CreateTableKeyAutoIncrementSample(client *tablestore.TableStoreClient) {
    createtableRequest := new(tablestore.CreateTableRequest)
    //创建表，数据表中包括三个主键：pk1，String类型；pk2，INTEGER类型，为自增列；pk3，Binary类型。
    tableMeta := new(tablestore.TableMeta)
    tableMeta.TableName = "incrementsampletable"
    tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKeyType_STRING)
    tableMeta.AddPrimaryKeyColumnOption("pk2", tablestore.PrimaryKeyType_INTEGER, tablestore.AUTO_INCREMENT)
    tableMeta.AddPrimaryKeyColumn("pk3", tablestore.PrimaryKeyType_BINARY)
    tableOption := new(tablestore.TableOption)
    tableOption.TimeToAlive = -1
    tableOption.MaxVersion = 3
    reservedThroughput := new(tablestore.ReservedThroughput)
    reservedThroughput.Readcap = 0
    reservedThroughput.Writecap = 0
    createtableRequest.TableMeta = tableMeta
    createtableRequest.TableOption = tableOption
    createtableRequest.ReservedThroughput = reservedThroughput
    client.CreateTable(createtableRequest)
}
```

2. 写数据

   写入数据时，无需为自增列设置具体值，只需将相应主键指定为自增主键。

```
import (
    "fmt"
    "github.com/aliyun/aliyun-tablestore-go-sdk/tablestore"
)
func PutRowWithKeyAutoIncrementSample(client *tablestore.TableStoreClient) {
    fmt.Println("begin to put row")
    putRowRequest := new(tablestore.PutRowRequest)
    putRowChange := new(tablestore.PutRowChange)
    putRowChange.TableName = "incrementsampletable"
    //设置主键，必须按照创建数据表时的顺序添加主键，并且需要指定pk2为自增主键。
    putPk := new(tablestore.PrimaryKey)
    putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
    putPk.AddPrimaryKeyColumnWithAutoIncrement("pk2")
    putPk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    putRowChange.PrimaryKey = putPk
    putRowChange.AddColumn("col1", "col1data1")
    putRowChange.AddColumn("col2", int64(100))
    putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    putRowRequest.PutRowChange = putRowChange
    _, err := client.PutRow(putRowRequest)
    if err != nil {
        fmt.Println("put row failed with error:", err)
    } else {
        fmt.Println("put row finished")
    }
}
```

4.4.8. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过PutRow、UpdateRow、DeleteRow或BatchWriteRow接口更新数据时，可以使用条件更新检查行存在性条件和列条件，只有满足条件时才能更新成功。

条件更新包括行存在性条件和列条件。

- 行存在性条件：包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。

  对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。

- 列条件：包括SingleColumnCondition和CompositeColumnCondition，是基于某一列或者某些列的列值进行条件判断。
  - SingleColumnCondition支持一列（可以是主键列）和一个常量比较。不支持两列或者两个常量相比较。

- CompositeColumnCondition的内节点为逻辑运算，子条件可以是SingleColumnCondition或CompositeColumnCondition。

条件更新可以实现乐观锁功能，即在更新某行时，先获取某列的值，假设为列A，值为1，然后设置条件**列A = 1**，更新行使**列A = 2**。如果更新失败，表示有其他客户端已成功更新该行。

限制

条件更新的列条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。

参数

参数	说明
RowExistenceExpectation	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别用RowExistenceExpectation_IGNORE、RowExistenceExpectation_EXPECT_EXIST、RowExistenceExpectation_EXPECT_NOT_EXIST表示。 • IGNORE：表示忽略，不做任何存在性检查。 • EXPECT_EXIST：表示期望存在，如果该行存在，则满足条件；如果该行不存在，则不满足条件。 • EXPECT_NOT_EXIST：期望行不存在，如果该行不存在，则满足条件；如果该行存在，则不满足条件。
ColumnName	列的名称。
ColumnValue	列的对比值。
Comparator	对列值进行比较的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=），分别用CT_EQUAL、CT_NOT_EQUAL、CT_GREATER_THAN、CT_GREATER_EQUAL、CT_LESS_THAN、CT_LESS_EQUAL表示。
Operator	对多个条件进行组合的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR，分别用LO_NOT、LO_AND、LO_OR表示。 逻辑运算符不同可以添加的子条件个数不同。 • 当逻辑运算符为NOT时，只能添加一个子条件。 • 当逻辑运算符为AND或OR时，必须至少添加两个子条件。
FilterIfMissing	当列在某行中不存在时，条件检查是否通过。类型为bool值，默认值为true，表示如果列在某行中不存在时，则条件检查通过，该行满足更新条件。 当设置FilterIfMissing为false时，如果列在某行中不存在时，则条件检查不通过，该行不满足更新条件。

参数	说明
LatestVersionsOnly	当列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置LatestVersionsOnly为false时，如果列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就条件检查通过，该行满足更新条件。

示例

- 示例1

  根据指定主键更新一行，如果指定的行存在，则更新成功，否则更新失败。

```
    updateRowRequest := new(tablestore.UpdateRowRequest)
    updateRowChange := new(tablestore.UpdateRowChange)
    updateRowChange.TableName = tableName
    updatePk := new(tablestore.PrimaryKey)
    updatePk.AddPrimaryKeyColumn("pk1", "pk1value1")
    updatePk.AddPrimaryKeyColumn("pk2", int64(2))
    updatePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    updateRowChange.PrimaryKey = updatePk
    updateRowChange.DeleteColumn("col1")             //删除col1列。
    updateRowChange.PutColumn("col2", int64(77))     //新增col2列，值为77。
    updateRowChange.PutColumn("col4", "newcol3")     //新增col4列，值为"newcol3"。
    //期望指定行存在。
    updateRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_EXIST)
    updateRowRequest.UpdateRowChange = updateRowChange
    _, err := client.UpdateRow(updateRowRequest)
```

- 示例2

  根据指定主键删除一行，如果指定的行存在，且col2列的值为3，则更新成功，否则更新失败。

```
    deleteRowReq := new(tablestore.DeleteRowRequest)
    deleteRowReq.DeleteRowChange = new(tablestore.DeleteRowChange)
    deleteRowReq.DeleteRowChange.TableName = tableName
    deletePk := new(tablestore.PrimaryKey)
    deletePk.AddPrimaryKeyColumn("pk1", "pk1value1")
    deletePk.AddPrimaryKeyColumn("pk2", int64(2))
    deletePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    deleteRowReq.DeleteRowChange.PrimaryKey = deletePk
    //期望行存在。
    deleteRowReq.DeleteRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_E
XIST)
    //期望列col2的值为3。
    clCondition1 := tablestore.NewSingleColumnCondition("col2", tablestore.CT_EQUAL, int6
4(3))
    deleteRowReq.DeleteRowChange.SetColumnCondition(clCondition1)
    _, err := client.DeleteRow(deleteRowReq)
```

4.4.9. 局部事务

使用局部事务功能，创建数据范围在一个分区键值内的局部事务。对局部事务中的数据进行读写操作后，可以根据实际提交或者丢弃局部事务。局部事务通过悲观锁（Pessimistic Lock）实现并发控制。

目前局部事务功能处于邀测中，默认关闭。如果需要使用该功能，请提交工单进行申请或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。

使用局部事务可以指定某个分区键值内的操作是原子的，对分区键值内的数据进行的操作要么全部成功要么全部失败，并且所提供的隔离级别为读已提交。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

1. 使用StartLocalTransaction在指定的分区键值创建一个局部事务，并获取局部事务ID。
2. 对局部事务范围内的数据进行读写操作。

   支持对局部事务进行操作的接口为GetRow、PutRow、DeleteRow、UpdateRow、BatchWriteRow和GetRange。
3. 使用CommitTransaction提交局部事务或者使用AbortTransaction丢弃局部事务。

限制

- 每个局部事务从创建开始生命周期最长为60秒。

  如果超过60秒未提交或丢弃局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
- 如果创建局部事务时超时，此请求可能在表格存储服务端已执行成功，此时用户需要等待该局部事务超时后重新创建。
- 未提交的局部事务可能失效，如果出现此情况，需要重试该局部事务内的操作。
- 在局部事务中读写数据有如下限制：
  - 不能使用局部事务ID访问局部事务范围（即创建时使用的分区键值）以外的数据。
  - 同一个局部事务中所有写请求的分区键值必须与创建局部事务时的分区键值相同，读请求则无此限制。
  - 一个局部事务同时只能用于一个请求中，在使用局部事务期间，其它使用此局部事务ID的操作均会失败。
  - 每个局部事务中两次读写操作的最大间隔为60秒。

    如果超过60秒未操作局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
  - 每个局部事务中写入的数据量最大为4 MB，按正常的写请求数据量计算规则累加。
  - 如果在局部事务中写入了未指定版本号的Cell，该Cell的版本号会在写入时（而非提交时）由表格存储服务端自动生成，生成规则与正常写入一个未指定版本号的Cell相同。
  - 如果BatchWriteRow请求中带有局部事务ID，则此请求中所有行只能操作该局部事务ID对应的表。
  - 在使用局部事务期间，对应分区键值的数据相当于被锁上，只有持有局部事务ID在局部事务范围内的写请求才会成功，其它不持有局部事务ID在局部事务范围内的写请求均会失败。在局部事务提交、丢弃或超时后，对应的锁也会被释放。
  - 带有局部事务ID的读写请求失败不会影响局部事务本身的存活情况，您可以按照正常的无局部事务ID的读写请求重试规则进行重试，或者主动丢弃当前局部事务。

参数

参数	说明
TableName	数据表名称。
PrimaryKey	数据表主键。 • 创建局部事务时，只需要指定局部事务对应的分区键值。 • 创建局部事务后，对局部事务范围内的数据进行读写操作时，需要指定完整主键。
TransactionId	局部事务ID，用于唯一标识一个局部事务。 创建局部事务后，操作局部事务时均需要带上局部事务ID。

示例

1. 调用StartLocalTransaction方法使用指定分区键值创建一个局部事务，并获取局部事务ID。

```
//局部事务需要指定一个分区键（第一列主键）。
transPk := new(tablestore.PrimaryKey)
transPk.AddPrimaryKeyColumn("userid", userName)
trans := &tablestore.StartLocalTransactionRequest{
    TableName: TableName,
    PrimaryKey: transPk,
}
response, err := client.StartLocalTransaction(trans)
if err != nil {
    fmt.Println("failed to create transaction", err)
    return
}
//获取局部事务ID。
transId := response.TransactionId
```

2. 对局部事务范围内的数据进行读写操作。

   对局部事务范围内数据的读写操作与正常读写数据操作基本相同，只需填入局部事务ID即可。

   - 写入一行数据。

```
putPk := new(tablestore.PrimaryKey)
putPk.AddPrimaryKeyColumn("userid", userName)
putPk.AddPrimaryKeyColumn("age", int64(18))
putRowChange := &tablestore.PutRowChange{
    TableName: tableName,
    PrimaryKey: putPk,
}
putRowChange.AddColumn("col1", "col1data1")
putRowChange.AddColumn("col2", int64(3))
putRowChange.AddColumn("col3", []byte("test"))
putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
//设置局部事务ID，局部事务ID可以通过StartLocalTransactionResponse.TransactionId获取。
putRowChange.TransactionId = transId
putRowRequest := &tablestore.PutRowRequest{
    PutRowChange: putRowChange,
}
_, err = client.PutRow(putRowRequest)
```

- 读取此行数据。

```
getRowPk := new(tablestore.PrimaryKey)
getRowPk.AddPrimaryKeyColumn("userid", userName)
getRowPk.AddPrimaryKeyColumn("age", int64(18))
criteria := &tablestore.SingleRowQueryCriteria{
    PrimaryKey: getRowPk,
    TableName: tableName,
    //设置读取最新版本的数据。
    MaxVersion: 1,
    //设置局部事务ID。
    TransactionId: transId,
}
getRowRequest := &tablestore.GetRowRequest{
    SingleRowQueryCriteria: criteria,
}
getResp, err := client.GetRow(getRowRequest)
```

3. 提交或丢弃局部事务。
   - 提交局部事务，使局部事务中的所有数据修改生效。

```
request := &tablestore.CommitTransactionRequest{
    TransactionId: transId,
}
commitResponse, err := client.CommitTransaction(request)
```

   - 丢弃局部事务，局部事务中的所有数据修改均不会应用到原有数据。

```
request := &tablestore.AbortTransactionRequest{
    TransactionId: transId,
}
abortResponse, err := client.AbortTransaction(request)
```

4.4.10. 原子计数器

将列当成一个原子计数器使用，对该列进行原子计数操作，可用于为某些在线应用提供实时统计功能，例如统计帖子的PV（实时浏览量）等。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

限制

- 只支持对整型列的列值进行原子计数操作。
- 作为原子计数器的列，如果写入数据前该列不存在，则默认值为0；如果写入数据前该列已存在且列值非整型，则产生OTSParameterInvalid错误。
- 增量值可以是正数或负数，但不能出现计算溢出。如果出现计算溢出，则产生OTSParameterInvalid错误。
- 默认不返回进行原子计数操作的列值，可以通过相应操作指定返回进行原子计数操作的列值。
- 在单次更新请求中，不能对某一列同时进行更新和原子计数操作。假设列A已经执行原子计数操作，则列

A不能再执行其他操作（例如列的覆盖写，列删除等）。

- 在一次BatchWriteRow请求中，支持对同一行进行多次更新操作。但是如果某一行已进行原子计数操作，则该行在此批量请求中只能出现一次。
- 原子计数操作只能作用在列值的最新版本，不支持对列值的特定版本做原子计数操作。更新完成后，原子计数操作会插入一个新的数据版本。

接口

updateRowChange类中新增了原子计数器的操作接口，操作接口说明请参见下表。

接口	说明
IncrementColumn(columnName string, value int64)	对列执行增量变更，例如+X，-X等。
AppendIncrementColumnToReturn(name string)	对于进行原子计数操作的列，设置需要返回列值的列名。
SetReturnIncrementValue()	设置返回类型，返回进行原子计数操作的列的新值。

参数

参数	说明
TableName	数据表名称。
ColumnName	进行原子计数操作的列名。只支持对整型列的列值进行原子计数器操作。
Value	对列进行增量变更的值。

示例

写入数据时，使用updateRowChange接口对整型列做列值的增量变更，然后读取更新后的新值。

```
func UpdateRowWithIncrement(client *tablestore.TableStoreClient, tableName string) {
    fmt.Println("begin to update row")
    updateRowRequest = new(tablestore.UpdateRowRequest)
    updateRowChange = new(tablestore.UpdateRowChange)
    //设置数据表名称。
    updateRowChange.TableName = tableName
    updatePk = new(tablestore.PrimaryKey)
    updatePk.AddPrimaryKeyColumn("pk1", "pk1increment")
    updatePk.AddPrimaryKeyColumn("pk2", int64(2))
    updatePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    updateRowChange.PrimaryKey = updatePk
    //将进行原子计数操作的col2列的列值+30，不能设置时间戳。
    updateRowChange.IncrementColumn("col2", int64(30))
    //将进行原子计数操作的列值返回。
    updateRowChange.SetReturnIncrementValue()
    updateRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    updateRowChange.AppendIncrementColumnToReturn("col2")
    updateRowRequest.UpdateRowChange = updateRowChange
    resp, err := client.UpdateRow(updateRowRequest)
    if err != nil {
        fmt.Println("update failed with error:", err)
        return
    } else {
        fmt.Println("update row finished")
        fmt.Println(resp)
        fmt.Println(len(resp.Columns))
        fmt.Println(resp.Columns[0].ColumnName)
        fmt.Println(resp.Columns[0].Value)
        fmt.Println(resp.Columns[0].Timestamp)
    }
}
```

4.4.11. 过滤器

在服务端对读取结果再进行一次过滤，根据过滤器（Filter）中的条件决定返回的行。使用过滤器后，只返回符合条件的数据行。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过GetRow、BatchGetRow或GetRange接口查询数据时，可以使用过滤器只返回符合条件的数据行。

过滤器目前包括SingleColumnValueFilter、SingleColumnValueRegexFilter和CompositeColumnValueFilter。

- SingleColumnValueFilter：只判断某个参考列的列值。
- SingleColumnValueRegexFilter：支持对类型为String的列值，使用正则表达式进行子字符串匹配，然后根据实际将匹配到的子字符串转换为String、Integer或者Double类型，再对子值使用过滤器进行过滤。

- CompositeColumnValueFilter：根据多个参考列的列值的判断结果进行逻辑组合，决定是否过滤某行。

> **说明** 关于过滤器的更多信息，请参见功能介绍中的过滤器。

限制

- 过滤器的条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。
- 过滤器中的参考列必须在读取的结果内。如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。
- 在GetRow、BatchGetRow和GetRange接口中使用过滤器不会改变接口的原生语义和限制项。

  使用GetRange接口时，一次扫描数据的行数不能超过5000行或者数据大小不能超过4 MB。

  当在该次扫描的5000行或者4 MB数据中没有满足过滤器条件的数据时，得到的Response中的Rows为空，但是NextStartPrimaryKey可能不为空，此时需要使用NextStartPrimaryKey继续读取数据，直到NextStartPrimaryKey为空。

参数

参数	说明
ColumnName	过滤器中参考列的名称。
ColumnValue	过滤器中参考列的对比值。
ComparatorType	过滤器中的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=），分别用tablestore.CT_EQUAL、tablestore.CT_NOT_EQUAL、tablestore.CT_GREATER_THAN、tablestore.CT_GREATER_EQUAL、tablestore.CT_LESS_THAN、tablestore.CT_LESS_EQUAL表示。
LogicOperator	过滤器中的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR，分别用tablestore.LO_NOT、tablestore.LO_AND、tablestore.LO_OR表示。
FilterIfMissing	当参考列在某行中不存在时，是否返回该行。类型为bool值，默认值为true，表示如果参考列在某行中不存在，则返回该行。 当设置FilterIfMissing为false时，如果参考列在某行中不存在，则不返回该行。
LatestVersionOnly	当参考列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果参考列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置LatestVersionsOnly为false时，如果参考列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就返回该行。

示例

- 构造SingleColumnValueFilter。

```
func GetRowWithFilter(client *tablestore.TableStoreClient, tableName string) {
    fmt.Println("begin to get row")
    pk := new(tablestore.PrimaryKey)
    pk.AddPrimaryKeyColumn("pk1", "pk1value1")
    pk.AddPrimaryKeyColumn("pk2", int64(2))
    pk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    //设置条件为c1 == "浙江", FilterIfMissing = true。
    condition := tablestore.NewSingleColumnCondition("c1", tablestore.ComparatorType(tabl
estore.CT_EQUAL), "浙江")
    condition.FilterIfMissing = true
    criteria := &tablestore.SingleRowQueryCriteria{
        TableName:     tableName,
        PrimaryKey:    pk,
        MaxVersion:    1,
        Filter:        condition,
    }
    getResp, err := client.GetRow(&tablestore.GetRowRequest{SingleRowQueryCriteria: crite
ria})
    if err != nil {
        fmt.Println("getrow failed with error:", err)
    } else {
        colMap := getResp.GetColumnMap()
        fmt.Println("length is ", len(colMap.Columns))
        fmt.Println("get row col0 result is ", getResp.Columns[0].ColumnName, getResp.Col
umns[0].Value)
    }
}
```

- 构造CompositeColumnValueFilter。

```
func GetRowWithCompositeColumnValueFilter(client *tablestore.TableStoreClient, tableName
string) {
    fmt.Println("begin to get row")
    pk := new(tablestore.PrimaryKey)
    pk.AddPrimaryKeyColumn("pk1", "pk1value1")
    pk.AddPrimaryKeyColumn("pk2", int64(2))
    pk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    //设置条件为(c1 == "浙江") AND (c2 == "杭州")。
    filter := tablestore.NewCompositeColumnCondition(tablestore.LO_AND)
    filter1 := tablestore.NewSingleColumnCondition("c1", tablestore.CT_EQUAL, "浙江")
    filter2 := tablestore.NewSingleColumnCondition("c2", tablestore.CT_EQUAL, "杭州")
    filter.AddFilter(filter2)
    filter.AddFilter(filter1)
    criteria := &tablestore.SingleRowQueryCriteria{
        TableName:  tableName,
        PrimaryKey: pk,
        MaxVersion: 1,
        Filter:     filter,
    }
    getResp, err := client.GetRow(&tablestore.GetRowRequest{SingleRowQueryCriteria: crite
ria})
    if err != nil {
        fmt.Println("getrow failed with error:", err)
    } else {
        colMap := getResp.GetColumnMap()
        fmt.Println("length is ", len(colMap.Columns))
        fmt.Println("get row col0 result is ", getResp.Columns[0].ColumnName, getResp.Col
umns[0].Value)
    }
}
```

4.5. 单行数据操作

表格存储提供了PutRow、GetRow、UpdateRow和DeleteRow等单行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

插入一行数据（PutRow）

PutRow接口用于新写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。

- 接口

```
    // @param PutRowRequest     执行PutRow操作所需参数的封装。
    // @return PutRowResponse
    PutRow(request *PutRowRequest) (*PutRowResponse, error)
```

- 参数

参数	说明
TableName	数据表名称。
PrimaryKey	行的主键。 **说明** ○ 设置的主键个数和类型必须和数据表的主键个数和类型一致。 ○ 当主键为自增列时，只需将相应主键指定为自增主键，详情请参见主键列自增。
Columns	行的属性列。 ○ 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType（可选）、时间戳（可选）。 ○ ColumnType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、BOOLEAN、DOUBLE五种，分别用ColumnType.INTEGER、ColumnType.STRING、ColumnType.BINARY、ColumnType.BOOLEAN、ColumnType.DOUBLE表示，其中BINARY不可省略，其他类型都可以省略。 ○ 时间戳即数据的版本号，详情请参见数据版本和生命周期。 数据的版本号可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。
Condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

- 示例

  插入一行数据。

```
putRowRequest := new(tablestore.PutRowRequest)
putRowChange := new(tablestore.PutRowChange)
putRowChange.TableName = tableName
putPk := new(tablestore.PrimaryKey)
putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
putPk.AddPrimaryKeyColumn("pk2", int64(2))
putPk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
putRowChange.PrimaryKey = putPk
putRowChange.AddColumn("col1", "col1data1")
putRowChange.AddColumn("col2", int64(3))
putRowChange.AddColumn("col3", []byte("test"))
putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
putRowRequest.PutRowChange = putRowChange
_, err := client.PutRow(putRowRequest)
if err != nil {
    fmt.Println("putrow failed with error:", err)
} else {
    fmt.Println("putrow finished")
}
```

详细代码请参见PutRow@GitHub。

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

读取的结果可能有如下两种：

- 如果该行存在，则返回该行的各主键列以及属性列。
- 如果该行不存在，则返回中不包含行，并且不会报错。
- 接口

```
//返回表中的一行数据。
//
// @param GetRowRequest             执行GetRow操作所需参数的封装。
// @return  GetRowResponse          GetRow操作的响应内容。
GetRow(request *GetRowRequest) (*GetRowResponse, error)
```

- 参数

参数	说明
TableName	数据表名称。
PrimaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。

参数	说明
ColumnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ◦ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置ColumnsToGet参数限制。如果将col0和col1加入到ColumnsToGet中，则只返回col0和col1列的值。 ◦ 当ColumnsToGet和Filter同时使用时，执行顺序是先获取ColumnsToGet指定的列，再在返回的列中进行条件过滤。
MaxVersion	最多读取的版本数。 **说明** MaxVersion与TimeRange必须至少设置一个。 ◦ 如果仅设置MaxVersion，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置TimeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置MaxVersion和TimeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。
TimeRange	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** MaxVersion与TimeRange必须至少设置一个。 ◦ 如果仅设置MaxVersion，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置TimeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置MaxVersion和TimeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ◦ 如果查询一个范围的数据，则需要设置Start和End。Start和End分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[Start, End)。 ◦ 如果查询特定版本号的数据，则需要设置Specific。Specific表示特定的时间戳。 Specific和[Start, End)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为INT64.MAX。

参数	说明
Filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当ColumnsToGet和Filter同时使用时，执行顺序是先获取ColumnsToGet指定的列，再在返回的列中进行条件过滤。

- 示例

  读取一行数据。

```
getRowRequest := new(tablestore.GetRowRequest)
    criteria := new(tablestore.SingleRowQueryCriteria);
    putPk := new(tablestore.PrimaryKey)
    putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
    putPk.AddPrimaryKeyColumn("pk2", int64(2))
    putPk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    criteria.PrimaryKey = putPk
    getRowRequest.SingleRowQueryCriteria = criteria
    getRowRequest.SingleRowQueryCriteria.TableName = tableName
    getRowRequest.SingleRowQueryCriteria.MaxVersion = 1
    getResp, err := client.GetRow(getRowRequest)
    if err != nil {
        fmt.Println("getrow failed with error:", err)
    } else {
        fmt.Println("get row col0 result is ",getResp.Columns[0].ColumnName, getResp.Colu
mns[0].Value,)
    }
```

  详细代码请参见Get Row@Git Hub。

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，可以增加和删除一行中的属性列，删除属性列指定版本的数据，或者更新已存在的属性列的值。如果更新的行不存在，则新增一行数据。

说明 当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。

- 接口

```
    // 更新表中的一行数据。
    // @param UpdateRowRequest      执行UpdateRow操作所需参数的封装。
    // @return UpdateRowResponse    UpdateRow操作的响应内容。
    UpdateRow(request *UpdateRowRequest) (*UpdateRowResponse, error)
```

- 参数

参数	说明
TableName	数据表名称。
PrimaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
Columns	行的属性列。 ○ 增加或更新数据时，需要设置属性名、属性值、属性类型（可选）、时间戳（可选）。 时间戳即数据的版本号，可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。详情请参见数据版本和生命周期。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。 ○ 删除属性列特定版本的数据时，只需要设置属性名和时间戳。 时间戳是64位整数，单位为毫秒，表示某个特定版本的数据。 ○ 删除属性列时，只需要设置属性名。 **说明** 删除一行的全部属性列不等同于删除该行，如果需要删除该行，请使用DeleteRow操作。
Condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

- 示例

  更新一行数据。

```
updateRowRequest := new(tablestore.UpdateRowRequest)
updateRowChange := new(tablestore.UpdateRowChange)
updateRowChange.TableName = tableName
updatePk := new(tablestore.PrimaryKey)
updatePk.AddPrimaryKeyColumn("pk1", "pk1value1")
updatePk.AddPrimaryKeyColumn("pk2", int64(2))
updatePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
updateRowChange.PrimaryKey = updatePk
updateRowChange.DeleteColumn("col1")
updateRowChange.PutColumn("col2", int64(77))
updateRowChange.PutColumn("col4", "newcol3")
updateRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_EXIST)
updateRowRequest.UpdateRowChange = updateRowChange
_, err := client.UpdateRow(updateRowRequest)
if err != nil {
    fmt.Println("update failed with error:", err)
} else {
    fmt.Println("update row finished")
}
```

详细代码请参见UpdateRow@GitHub

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行数据。如果删除的行不存在，则不会发生任何变化。

- 接口

```
// 删除表中的一行。
// @param DeleteRowRequest          执行DeleteRow操作所需参数的封装。
// @return DeleteRowResponse        DeleteRow操作的响应内容。
DeleteRow(request *DeleteRowRequest) (*DeleteRowResponse, error)
```

- 参数

参数	说明
TableName	数据表名称。
PrimaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
Condition	支持使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

- 示例

  删除一行数据。

```
    deleteRowReq := new(tablestore.DeleteRowRequest)
    deleteRowReq.DeleteRowChange = new(tablestore.DeleteRowChange)
    deleteRowReq.DeleteRowChange.TableName = tableName
    deletePk := new(tablestore.PrimaryKey)
    deletePk.AddPrimaryKeyColumn("pk1", "pk1value1")
    deletePk.AddPrimaryKeyColumn("pk2", int64(2))
    deletePk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    deleteRowReq.DeleteRowChange.PrimaryKey = deletePk
    deleteRowReq.DeleteRowChange.SetCondition(tablestore.RowExistenceExpectation_EXPECT_E
XIST)
    clCondition1 := tablestore.NewSingleColumnCondition("col2", tablestore.CT_EQUAL, int6
4(3))
    deleteRowReq.DeleteRowChange.SetColumnCondition(clCondition1)
    _, err := client.DeleteRow(deleteRowReq)
    if err != nil {
        fmt.Println("delete failed with error:", err)
    } else {
        fmt.Println("delete row finished")
    }
```

详细代码请参见DeleteRow@GitHub。

4.6. 多行数据操作

表格存储提供了BatchWriteRow、BatchGetRow、GetRange等多行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

批量写（BatchWriteRow）

批量写接口用于在一次请求中进行批量的写入操作，也支持一次对多个数据表进行写入。BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，构造子操作的过程与使用PutRow接口、UpdateRow接口和DeleteRow接口时相同，也支持使用条件更新。

BatchWriteRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量写入可能存在部分行失败的情况，失败行的Index及错误信息在返回的BatchWriteRowResponse中，但并不抛出异常。因此调用BatchWriteRow接口时，需要检查返回值，判断每行的状态是否成功；如果不检查返回值，则可能会忽略掉部分操作的失败。

当服务端检查到某些操作出现参数错误时，BatchWriteRow接口可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。

- 接口

```
// 对多个数据表中的多行数据进行增加、删除或者更新操作。
//
// @param BatchWriteRowRequest              执行BatchWriteRow操作所需参数的封装。
// @return  BatchWriteRowResponse           BatchWriteRow操作的响应内容。
BatchWriteRow(request *BatchWriteRowRequest) (*BatchWriteRowResponse,error)
```

- 参数

  详细参数说明请参见单行数据操作。

- 示例

  批量写入100行数据。

```
batchWriteReq := &tablestore.BatchWriteRowRequest{}
for i := 0; i < 100; i++ {
    putRowChange := new(tablestore.PutRowChange)
    putRowChange.TableName = tableName
    putPk := new(tablestore.PrimaryKey)
    putPk.AddPrimaryKeyColumn("pk1", "pk1value1")
    putPk.AddPrimaryKeyColumn("pk2", int64(i))
    putPk.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    putRowChange.PrimaryKey = putPk
    putRowChange.AddColumn("col1", "fixvalue")
    putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    batchWriteReq.AddRowChange(putRowChange)
}
response, err := client.BatchWriteRow(batchWriteReq)
if err != nil {
    fmt.Println("batch request failed with:", response)
} else {
    fmt.Println("batch write row finished")
}
```

  详细代码请参见BatchWriteRow@GitHub。

批量读（BatchGetRow）

批量读接口用于一次请求读取多行数据，也支持一次对多个数据表进行读取。BatchGetRow由多个GetRow子操作组成。构造子操作的过程与使用GetRow接口时相同，也支持使用过滤器。

批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA列。

BatchGetRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量读取可能存在部分行失败的情况，失败行的错误信息在返回的BatchGetRowResponse中，但并不抛出异常。因此调用BatchGetRow接口时，需要检查返回值，判断每行的状态是否成功。

- 接口

```
//返回数据表（Table）中的多行数据。
//
// @param BatchGetRowRequest              执行BatchGetRow操作所需参数的封装。
// @return  BatchGetRowResponse           BatchGetRow操作的响应内容。
BatchGetRow(request *BatchGetRowRequest) (*BatchGetRowResponse, error)
```

- 参数

详细参数说明请参见单行数据操作。

- 示例

  批量一次读取10行数据。

```
batchGetReq := &tablestore.BatchGetRowRequest{}
mqCriteria := &tablestore.MultiRowQueryCriteria{}
for i := 0; i < 10; i++ {
    pkToGet := new(tablestore.PrimaryKey)
    pkToGet.AddPrimaryKeyColumn("pk1", "pk1value1")
    pkToGet.AddPrimaryKeyColumn("pk2", int64(i))
    pkToGet.AddPrimaryKeyColumn("pk3", []byte("pk3"))
    mqCriteria.AddRow(pkToGet)
    mqCriteria.MaxVersion = 1
}
mqCriteria.TableName = tableName
batchGetReq.MultiRowQueryCriteria = append(batchGetReq.MultiRowQueryCriteria, mqCriteria)
batchGetResponse, err := client.BatchGetRow(batchGetReq)
if err != nil {
    fmt.Println("batchget failed with error:", err)
} else {
    fmt.Println("batchget finished")
}
```

  详细代码请参见BatchGetRow@GitHub。

范围读（GetRange）

范围读接口用于读取一个主键范围内的数据。

范围读接口支持按照确定范围进行正序读取和逆序读取，可以设置要读取的行数。如果范围较大，已扫描的行数或者数据量超过一定限制，会停止扫描，并返回已获取的行和下一个主键信息。您可以根据返回的下一个主键信息，继续发起请求，获取范围内剩余的行。

GetRange操作可能在如下情况停止执行并返回数据。

- 扫描的行数据大小之和达到4 MB。
- 扫描的行数等于5000。
- 返回的行数等于最大返回行数。
- 当前剩余的预留读吞吐量已全部使用，余量不足以读取下一条数据。

> **说明** 表格存储表中的行默认是按照主键排序的，而主键是由全部主键列按照顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

- 接口

```
// 从表中查询一个范围内的多行数据。
//
// @param GetRangeRequest            执行GetRange操作所需参数的封装。
// @return GetRangeResponse          GetRange操作的响应内容。
GetRange(request *GetRangeRequest) (*GetRangeResponse,error)
```

- 参数

参数	说明
TableName	数据表名称。
Direction	读取方向。 ○ 如果值为正序（FORWARD），则起始主键必须小于结束主键，返回的行按照主键由小到大的顺序进行排列。 ○ 如果值为逆序（BACKWARD），则起始主键必须大于结束主键，返回的行按照主键由大到小的顺序进行排列。 例如同一表中有两个主键A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。
StartPrimaryKey EndPrimaryKey	本次范围读取的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由INF_MIN和INF_MAX类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中INF_MIN表示无限小，任何类型的值都比它大；INF_MAX表示无限大，任何类型的值都比它小。 ○ StartPrimaryKey表示起始主键，如果该行存在，则返回结果中一定会包含此行。 ○ EndPrimaryKey表示结束主键，无论该行是否存在，返回结果中都不会包含此行。 数据表中的行按主键从小到大排序，读取范围是一个左闭右开的区间，正序读取时，返回的是大于等于起始主键且小于结束主键的所有的行。
Limit	数据的最大返回行数，此值必须大于 0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。此时可以通过返回结果中的NextStartPrimaryKey记录本次读取到的位置，用于下一次读取。
ColumnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ○ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置ColumnsToGet参数限制。如果将col0和col1加入到ColumnsToGet中，则只返回col0和col1列的值。 ○ 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。 ○ 当ColumnsToGet和Filter同时使用时，执行顺序是先获取ColumnsToGet指定的列，再在返回的列中进行条件过滤。

参数	说明
MaxVersions	最多读取的版本数。 **说明** MaxVersion与TimeRange必须至少设置一个。 ○ 如果仅设置MaxVersion，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置TimeRange，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置MaxVersion和TimeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。
TimeRange	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** MaxVersion与TimeRange必须至少设置一个。 ○ 如果仅设置MaxVersion，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置TimeRange，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置MaxVersion和TimeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 如果查询一个范围的数据，则需要设置Start和End。Start和End分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[Start, End)。 ○ 如果查询特定版本号的数据，则需要设置Specific。Specific表示特定的时间戳。 Specific和[Start, End)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为INT64.MAX。
Filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当ColumnsToGet和Filter同时使用时，执行顺序是先获取ColumnsToGet指定的列，再在返回的列中进行条件过滤。
NextStartPrimaryKey	根据返回结果中的NextStartPrimaryKey判断数据是否全部读取。 ○ 当返回结果中NextStartPrimaryKey不为空时，可以使用此返回值作为下一次GetRange操作的起始点继续读取数据。 ○ 当返回结果中NextStartPrimaryKey为空时，表示读取范围内的数据全部返回。

- 示例

  按照范围读取数据。

```
getRangeRequest := &tablestore.GetRangeRequest{}
    rangeRowQueryCriteria := &tablestore.RangeRowQueryCriteria{}
    rangeRowQueryCriteria.TableName = tableName
    startPK := new(tablestore.PrimaryKey)
    startPK.AddPrimaryKeyColumnWithMinValue("pk1")
    startPK.AddPrimaryKeyColumnWithMinValue("pk2")
    startPK.AddPrimaryKeyColumnWithMinValue("pk3")
    endPK := new(tablestore.PrimaryKey)
    endPK.AddPrimaryKeyColumnWithMaxValue("pk1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk2")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk3")
    rangeRowQueryCriteria.StartPrimaryKey = startPK
    rangeRowQueryCriteria.EndPrimaryKey = endPK
    rangeRowQueryCriteria.Direction = tablestore.FORWARD
    rangeRowQueryCriteria.MaxVersion = 1
    rangeRowQueryCriteria.Limit = 10
    getRangeRequest.RangeRowQueryCriteria = rangeRowQueryCriteria
    getRangeResp, err := client.GetRange(getRangeRequest)
    fmt.Println("get range result is " ,getRangeResp)
    for {
        if err != nil {
            fmt.Println("get range failed with error:", err)
        }
        for _, row := range getRangeResp.Rows {
            fmt.Println("range get row with key", row.PrimaryKey.PrimaryKeys[0].Value, ro
w.PrimaryKey.PrimaryKeys[1].Value, row.PrimaryKey.PrimaryKeys[2].Value)
        }
        if getRangeResp.NextStartPrimaryKey == nil {
            break
        } else {
            fmt.Println("next pk is :", getRangeResp.NextStartPrimaryKey.PrimaryKeys[0].V
alue, getRangeResp.NextStartPrimaryKey.PrimaryKeys[1].Value, getRangeResp.NextStartPrimar
yKey.PrimaryKeys[2].Value)
            getRangeRequest.RangeRowQueryCriteria.StartPrimaryKey = getRangeResp.NextStar
tPrimaryKey
            getRangeResp, err = client.GetRange(getRangeRequest)
        }
        fmt.Println("continue to query rows")
    }
    fmt.Println("range get row finished")
```

详细代码请参见GetRange@GitHub。

4.7. 多元索引

4.7.1. 创建多元索引

使用CreateSearchIndex接口在数据表上创建一个多元索引。一个数据表可以创建多个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（TimeToLive）必须为-1，最大版本数（MaxVersions）必须为

1。

参数

创建多元索引时，需要指定数据表名称（TableName）、多元索引名称（IndexName）和索引的结构信息（IndexSchema），其中IndexSchema包含FieldSchemas（Index的所有字段的设置）、IndexSetting（索引设置）和IndexSort（索引预排序设置）。详细参数说明请参见下表。

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
FieldSchemas	FieldSchema的列表，每个FieldSchema包含如下内容： • FieldName（必选）：创建多元索引的字段名，即列名，类型为String。 多元索引中的字段可以是主键列或者属性列。 • FieldType（必选）：字段类型，类型为tablestore.FieldType_XXX。更多信息，请参见字段。 • Array（可选）：是否为数组，类型为Boolean。 如果设置为true，则表示该列是一个数组，在写入时，必须按照JSON数组格式写入，例如["a","b","c"]。 由于Nested类型是一个数组，当FieldType为Nested类型时，无需设置此参数。 • Index（可选）：是否开启索引，类型为Boolean。 默认为true，表示对该列构建倒排索引或者空间索引；如果设置为false，则不会对该列构建索引。 • Analyzer（可选）：分词器类型。当字段类型为Text时，可以设置此参数；如果不设置，则默认分词器类型为单字分词。关于分词的更多信息，请参见分词。 • EnableSortAndAgg（可选）：是否开启排序与统计聚合功能，类型为Boolean。 只有EnableSortAndAgg设置为true的字段才能进行排序。关于排序的更多信息，请参见排序和翻页。 注意 Nested类型的字段不支持开启排序与统计聚合功能，但是Nested类型内部的子列支持开启排序与统计聚合功能。 • Store（可选）：是否在多元索引中附加存储该字段的值，类型为Boolean。 开启后，可以直接从多元索引中读取该字段的值，而不必反查数据表，可用于查询性能优化。
IndexSetting	索引设置，包含RoutingFields设置。 RoutingFields（可选）：自定义路由字段。可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

参数	说明
IndexSort	索引预排序设置，包含Sorters设置。如果不设置，则默认按照主键排序。 **说明** 含有Nested类型的索引不支持IndexSort，没有预排序。 Sorters（必选）：索引的预排序方式，支持按照主键排序和字段值排序。关于排序的更多信息，请参见排序和翻页。 • PrimaryKeySort表示按照主键排序，包含如下设置： Order：排序的顺序，可按升序或者降序排序，默认为升序。 • FieldSort表示按照字段值排序， 包含如下设置： 只有建立索引且开启排序与统计聚合功能的字段才能进行预排序。 ○ FieldName：排序的字段名。 ○ Order：排序的顺序，可按照升序或者降序排序，默认为升序。 ○ Mode：当字段存在多个值时的排序方式。

示例

- 创建多元索引

  创建一个多元索引。该多元索引包含col1和col2两列，类型分别设置为字符串（Keyword）和整型（Long）。

```
func CreateSearchIndex(client *tablestore.TableStoreClient, tableName string, indexName s
tring) {
    request := &tablestore.CreateSearchIndexRequest{}
    request.TableName = "exampletable" //设置数据表名称。
    request.IndexName = "examplesearchindex" //设置多元索引名称。
    schemas := []*tablestore.FieldSchema{}
    field1 := &tablestore.FieldSchema{
        FieldName: proto.String("Col_Keyword"), //设置字段名，使用proto.String用于获取字符串
指针。
        FieldType: tablestore.FieldType_KEYWORD, //设置字段类型。
        Index:     proto.Bool(true), //设置开启索引。
        EnableSortAndAgg: proto.Bool(true), //设置开启排序与统计聚合功能。
    }
    field2 := &tablestore.FieldSchema{
        FieldName: proto.String("Col_Long"),
        FieldType: tablestore.FieldType_LONG,
        Index:     proto.Bool(true),
        EnableSortAndAgg: proto.Bool(true),
    }
    schemas = append(schemas, field1, field2)
    request.IndexSchema = &tablestore.IndexSchema{
        FieldSchemas: schemas, //设置多元索引包含的字段。
    }
    resp, err := client.CreateSearchIndex(request) //调用client创建多元索引。
    if err != nil {
        fmt.Println("error :", err)
        return
    }
    fmt.Println("CreateSearchIndex finished, requestId:", resp.ResponseInfo.RequestId)
}
```

- 创建多元索引时指定IndexSort。

  创建一个多元索引，同时指定索引预排序。该多元索引包含col1和col2两列，类型分别设置为字符串（Keyword）和整型（Long）。

```
func createSearchIndex_withIndexSort(client *tablestore.TableStoreClient){
    request := &tablestore.CreateSearchIndexRequest{}
    request.TableName = "exampletable" //设置数据表名称。
    request.IndexName = "examplesearchindex" //设置多元索引名称。
    schemas := []*tablestore.FieldSchema{}
    field1 := &tablestore.FieldSchema{
        FieldName: proto.String("col1"), //设置字段名，使用proto.String用于获取字符串指针。
        FieldType: tablestore.FieldType_KEYWORD, //设置字段类型。
        Index:     proto.Bool(true), //设置开启索引。
        EnableSortAndAgg: proto.Bool(true), //设置开启排序与统计聚合功能。
    }
    field2 := &tablestore.FieldSchema{
        FieldName: proto.String("col2"),
        FieldType: tablestore.FieldType_LONG,
        Index:     proto.Bool(true),
        EnableSortAndAgg: proto.Bool(true),
    }
    schemas = append(schemas, field1, field2)
    request.IndexSchema = &tablestore.IndexSchema{
        FieldSchemas: schemas, //设置多元索引包含的字段。
        IndexSort: &search.Sort{ // 指定索引预排序。先按照col2升序，再按照col1降序排序。
            Sorters: []search.Sorter{
                &search.FieldSort{
                    FieldName: "col2",
                    Order:     search.SortOrder_ASC.Enum(),
                },
                &search.FieldSort{
                    FieldName: "col1",
                    Order:     search.SortOrder_DESC.Enum(),
                },
            },
        },
    }
    resp, err := client.CreateSearchIndex(request) //调用client创建多元索引。
    if err != nil {
        fmt.Println("error :", err)
        return
    }
    fmt.Println("CreateSearchIndex finished, requestId:", resp.ResponseInfo.RequestId)
}
```

4.7.2. 列出多元索引列表

创建多元索引后，使用ListSearchIndex接口可以获取当前实例下或某个数据表关联的所有多元索引的列表信息。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称，可以为空。 • 如果设置了数据表名称，则返回该数据表关联的所有多元索引的列表。 • 如果未设置数据表名称，则返回当前实例下所有多元索引的列表。

示例

```
func ListSearchIndex(client *tablestore.TableStoreClient, tableName string) {
    request := &tablestore.ListSearchIndexRequest{}
    request.TableName = tableName //设置数据表名称。
    resp, err := client.ListSearchIndex(request) //获取数据表关联的所有多元索引。
    if err != nil {
        fmt.Println("error: ", err)
        return
    }
    for _, info := range resp.IndexInfo {
        fmt.Printf("%#v\n", info) //打印多元索引的信息。
    }
    fmt.Println("ListSearchIndex finished, requestId:", resp.ResponseInfo.RequestId)
}
```

4.7.3. 查询多元索引描述信息

创建多元索引后，使用DescribeSearchIndex接口可以查询多元索引的描述信息，包括多元索引的字段信息和索引配置等。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。

示例

```
func DescribeSearchIndex(client *tablestore.TableStoreClient, tableName string, indexName s
tring) {
    request := &tablestore.DescribeSearchIndexRequest{}
    request.TableName = tableName  //设置数据表名称。
    request.IndexName = indexName  //设置多元索引名称。
    resp, err := client.DescribeSearchIndex(request)
    if err != nil {
        fmt.Println("error: ", err)
        return
    }
    fmt.Println("FieldSchemas:")
    for _, schema := range resp.Schema.FieldSchemas {
        fmt.Printf("%s\n", schema) //打印多元索引中字段的schema信息。
    }
    fmt.Println("DescribeSearchIndex finished, requestId: ", resp.ResponseInfo.RequestId)
}
```

4.7.4. 删除多元索引

使用DeleteSearchIndex接口可以删除指定数据表的一个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。

示例

```
func DeleteSearchIndex(client *tablestore.TableStoreClient, tableName string, indexName str
ing) {
    request := &tablestore.DeleteSearchIndexRequest{}
    request.TableName = tableName //设置数据表名称。
    request.IndexName = indexName //设置多元索引名称。
    resp, err := client.DeleteSearchIndex(request) //调用client删除多元索引。
    if err != nil {
        fmt.Println("error: ", err)
        return
    }
    fmt.Println("DeleteSearchIndex finished, requestId: ", resp.ResponseInfo.RequestId)
}
```

4.7.5. 精确查询

TermQuery采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。对于Text类型字段，只要分词后有词条可以精确匹配即可。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Query	设置查询类型为TermQuery。
FieldName	要匹配的字段。
Term	查询关键词，即要匹配的值。 该词不会被分词，会被当做完整词去匹配。 对于Text类型字段，只要分词后有词条可以精确匹配即可。例如某个Text类型的字段，值为“tablestore is cool”，如果分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"的数据。
 */
func TermQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.TermQuery{} //设置查询类型为TermQuery。
    query.FieldName = "Col_Keyword" //设置要匹配的字段。
    query.Term = "hangzhou" //设置要匹配的值。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配到的总行数，非返回行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.6. 多词精确查询

类似于TermQuery，但是TermsQuery可以指定多个查询关键词，查询匹配这些词的数据。多个查询关键词中只要有一个词精确匹配，该行数据就会被返回，等价于SQL中的In。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。

参数	说明
IndexName	多元索引名称。
Query	设置查询类型为TermsQuery。
FieldName	要匹配的字段。
Terms	多个查询关键词，即要匹配的值。 多个查询关键词中只要有一个词精确匹配，该行数据就会被返回。
Limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置Limit=0，即不返回任意一行数据。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"或"tablestore"的数据。
 */
func TermsQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.TermsQuery{}   //设置查询类型为TermsQuery。
    query.FieldName = "Col_Keyword" //设置要匹配的字段。
    terms := make([]interface{}, 0)
    terms = append(terms, "hangzhou")
    terms = append(terms, "tablestore")
    query.Terms = terms //设置要匹配的值。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetLimit(100)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll: true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.7. 全匹配查询

MatchAllQuery可以匹配所有行，常用于查询表中数据总行数，或者随机返回几条数据。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Query	设置查询类型为MatchAllQuery。
Limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置Limit=0，即不返回任意一行数据。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
ColumnsToGet	是否返回所有列。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/**
 * 通过MatchAllQuery查询表中数据的总行数。
 */
func MatchAllQuery(client *tablestore.TableStoreClient, tableName string, indexName string)
{
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.MatchAllQuery{} //设置查询类型为MatchAllQuery。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchQuery.SetLimit(0) //设置Limit为0，表示不获取具体数据。
    searchRequest.SetSearchQuery(searchQuery)
    searchResponse, err := client.Search(searchRequest)
    if err != nil { //判断异常。
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess)
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印总行数。
}
```

4.7.8. 匹配查询

MatchQuery采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用MatchPhraseQuery实现高性能的模糊查询。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Query	设置查询类型为MatchQuery。
FieldName	要匹配的列。 匹配查询可应用于Text类型。
Text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被切分成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
Operator	逻辑运算符。默认为OR，表示当分词后的多个词只要有部分匹配时，则行数据满足查询条件。 如果设置Operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。
MinimumShouldMatch	最小匹配个数。 只有当某一行数据的FieldName列的值中至少包括最小匹配个数的词时，才会返回该行数据。 说明 MinimumShouldMatch需要与逻辑运算符OR配合使用。
Offset	本次查询的开始位置。
Limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置Limit=0，即不返回任意一行数据。

参数	说明
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/**
 *  查询表中Col_Keyword列的值能够匹配"hangzhou"的数据，返回匹配到的总行数和一些匹配成功的行。
 */
func MatchQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.MatchQuery{} //设置查询类型为MatchQuery。
    query.FieldName = "Col_Keyword" //设置要匹配的列。
    query.Text = "hangzhou" //设置要匹配的值。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchQuery.SetOffset(0) //设置offset为0。
    searchQuery.SetLimit(20) //设置limit为20，表示最多返回20条数据。
    searchRequest.SetSearchQuery(searchQuery)
    searchResponse, err := client.Search(searchRequest)
    if err != nil { //判断异常。
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //匹配的总行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows)) //返回的行数。
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody)) //不设置columnsToGet，默认只返回主键。
    }
    // 设置返回所有列
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err = client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //匹配的总行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.9. 短语匹配查询

类似于MatchQuery，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。如果查询列的分词类型为模糊分词，则使用MatchPhraseQuery可以实现比WildcardQuery更快的模糊查询。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Query	设置查询类型为MatchPhraseQuery。
FieldName	要匹配的列。 匹配查询可应用于Text类型。
Text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“...，this is tablestore”、“this is a table”，但是无法匹配到“this table is ...”以及“is this a table”。
Offset	本次查询的开始位置。
Limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置Limit=0，即不返回任意一行数据。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false ，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Text列的值能够匹配"hangzhou shanghai"的数据，匹配条件为短语匹配（要求短语完整的按照顺序匹配），返回匹配到的总行数和一些匹配成功的行。
 */
func MatchPhraseQuery(client *tablestore.TableStoreClient, tableName string, indexName stri
ng) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.MatchPhraseQuery{} //设置查询类型为MatchPhraseQuery。
    query.FieldName = "Col_Text"  //设置要匹配的列。
    query.Text = "hangzhou shanghai"  //设置要匹配的值。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchQuery.SetOffset(0) //设置offset为0。
    searchQuery.SetLimit(20) //设置limit为20，表示最多返回20条数据。
    searchRequest.SetSearchQuery(searchQuery)
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //匹配的总行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
    //设置返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err = client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //匹配的总行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.10. 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Query	设置查询类型为PrefixQuery。
FieldName	要匹配的字段。
Prefix	前缀值。 对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Keyword列前缀为"hangzhou"的数据。
 */
func PrefixQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.PrefixQuery{} //设置查询类型为PrefixQuery。
    query.FieldName = "Col_Keyword" //设置要匹配的字段。
    query.Prefix = "hangzhou" //设置前缀值。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配到的总行数，非返回行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.11. 范围查询

RangeQuery根据范围条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足范围条件即可。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。

参数	说明
IndexName	多元索引名称。
Query	设置查询类型为RangeQuery。
FieldName	要匹配的字段。
From	起始位置的值。 设置范围条件时，大于（>）可以使用GT表示，大于等于（>=）可以使用GTE表示。
To	结束位置的值。 设置范围条件时，小于（<）可以使用LT表示；小于等于（<=）可以使用LTE表示。
IncludeLower	结果中是否需要包括From值，类型为Boolean。
IncludeUpper	结果中是否需要包括To值，类型为Boolean。
Sort	按照指定方式排序，详情请参见排序和翻页。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/**
 * 查询表中Col_Long列大于3的数据，结果按照Col_Long列的值逆序排序。
 */
func RangeQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    searchQuery := search.NewSearchQuery()
    rangeQuery := &search.RangeQuery{} //设置查询类型为RangeQuery。
    rangeQuery.FieldName = "Col_Long" //设置要匹配的字段
    rangeQuery.GT(3) //设置该字段的范围条件为大于3。
    searchQuery.SetQuery(rangeQuery)
    //设置按照Col_Long列逆序排序。
    searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.FieldSort{
                FieldName: "Col_Long",
                Order:     search.SortOrder_DESC.Enum(),
            },
        },
    })
    searchRequest.SetSearchQuery(searchQuery)
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.12. 通配符查询

通配符查询中，要匹配的值可以是一个带有通配符的字符串，目前支持星号（*）和问号（?）两种通配符。要匹配的值中可以用星号（*）代表任意字符序列，或者用问号（?）代表任意单个字符，且支持以星号（*）或问号（?）开头。例如查询“table*e”，可以匹配到“tablestore”。

如果查询的模式为*word*，则您可以使用性能更好的模糊查询，具体实现方法如下：

1. 创建多元索引时，设置列为Text类型且设置分词类型为模糊分词。
2. 使用多元索引查询数据时，使用MatchPhraseQuery且设置查询词为word。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	描述
TableName	数据表名称。
IndexName	多元索引名称。
query	设置查询类型为WildcardQuery。
FieldName	列名称。
Value	带有通配符的字符串，字符串长度不能超过20个字符。
ColumnsToGet	是否返回所有列。 ReturnAll默认为false，表示不返回所有列，此时可以通过ColumnsToGet指定返回的列；如果未通过ColumnsToGet指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/**
 * 使用通配符查询，查询表中Col_Keyword列的值匹配"hang*u"的数据。
 */
func WildcardQuery(client *tablestore.TableStoreClient, tableName string, indexName string)
{
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.WildcardQuery{} //设置查询类型为WildcardQuery。
    query.FieldName = "Col_Keyword"
    query.Value = "hang*u"
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配到的总行数，非返回行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.13. 地理位置查询

地理位置查询包括地理距离查询（GeoDistanceQuery）、地理长方形范围查询（GeoBoundingBoxQuery）和地理多边形范围查询（GeoPolygonQuery）三种方式。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

地理距离查询（GeoDistanceQuery）

GeoDistanceQuery根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，满足查询条件。

- 参数

参数	说明
FieldName	列名，类型为Geopoint。
CenterPoint	中心地理坐标点，是一个经纬度值。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。
DistanceInMeter	距离中心点的距离，类型为Double。单位为米。
Query	多元索引的查询语句。设置查询类型为GeoDistanceQuery。
TableName	数据表名称。
IndexName	多元索引名称。

- 示例

  查询表中Col_GeoPoint列的值距离中心点不超过一定距离的数据。

```
func GeoDistanceQuery(client *tablestore.TableStoreClient, tableName string, indexName st
ring) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.GeoDistanceQuery{} //设置查询类型为GeoDistanceQuery。
    query.FieldName = "Col_GeoPoint"
    query.CenterPoint = "5,5"  //设置中心点。
    query.DistanceInMeter = 10000.0 //设置条件为到中心点的距离不超过10000米。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    // 设置返回所有列
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配到的总行数，非返回行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

地理长方形范围查询（GeoBoundingBoxQuery）

GeoBoundingBoxQuery根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时满足查询条件。

- 参数

参数	说明
FieldName	列名，类型为Geopoint。
TopLeft	长方形框的左上角的坐标。
BottomRight	长方形框的右下角的坐标，通过左上角和右下角就可以确定一个唯一的长方形。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。
Query	多元索引的查询语句。设置查询类型为GeoBoundingBoxQuery。
TableName	数据表名称。
IndexName	多元索引名称。

- 示例

  Col_GeoPoint是Geopoint类型，查询表中Col_GeoPoint列的值在左上角为"10,0", 右下角为"0,10"的长方形范围内的数据。

```
func GeoBoundingBoxQuery(client *tablestore.TableStoreClient, tableName string, indexName
string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.GeoBoundingBoxQuery{} //设置查询类型为GeoBoundingBoxQuery。
    query.FieldName = "Col_GeoPoint"  //设置列名。
    query.TopLeft = "10,0"  //设置长方形左上角。
    query.BottomRight = "0,10" //设置长方形右下角。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配到的总行数，非返回行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

地理多边形范围查询（GeoPolygonQuery）

GeoPolygonQuery根据一个多边形范围条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时满足查询条件。

- 参数

参数	说明
FieldName	列名，类型为Geopoint。
Points	组成多边形的距离坐标点。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。
Query	多元索引的查询语句。设置查询类型为GeoPolygonQuery。
TableName	数据表名称。
IndexName	多元索引名称。

- 示例

  查询表中Col_GeoPoint列的值在一个给定多边形范围内的数据。

```
func GeoPolygonQuery(client *tablestore.TableStoreClient, tableName string, indexName str
ing) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.GeoPolygonQuery{} //设置查询类型为GeoDistanceQuery。
    query.FieldName = "Col_GeoPoint"
    query.Points = []string{"0,0","5,5","5,0"} //设置多边形的顶点。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配到的总行数，非返回行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.14. 多条件组合查询

BoolQuery查询条件包含一个或者多个子查询条件，根据子查询条件来判断一行数据是否满足查询条件。每个子查询条件可以是任意一种Query类型，包括BoolQuery。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。

参数	说明
IndexName	多元索引名称。
MustQueries	多个Query列表，行数据必须满足所有的子查询条件才算匹配，等价于And操作符。
MustNotQueries	多个Query列表，行数据必须不能满足任何的子查询条件才算匹配，等价于Not操作符。
FilterQueries	多个Query列表，行数据必须满足所有的子Filter才算匹配，Filter类似于Query，区别是Filter不会根据满足的Filter个数进行相关性算分。
ShouldQueries	多个Query列表，可以满足，也可以不满足，等价于Or操作符。 行数据应该至少满足ShouldQueries子查询条件的最小匹配个数才算匹配。 如果满足的ShouldQueries子查询条件个数越多，则整体的相关性分数更高。
MinimumShouldMatch	ShouldQueries子查询条件的最小匹配个数。当同级没有其他Query，只有ShouldQueries时，默认值为1；当同级已有其他Query，例如MustQueries、MustNotQueries和FilterQueries时，默认值为0。

示例

通过BoolQuery进行多条件组合查询。

```
/**
 * 通过BoolQuery进行多条件组合查询。
 */
func BoolQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    /**
     * 查询条件一：RangeQuery，Col_Long的列值大于3。
     */
    rangeQuery := &search.RangeQuery{}
    rangeQuery.FieldName = "Col_Long"
    rangeQuery.GT(3)
    /**
     * 查询条件二：MatchQuery，Col_Keyword的列值要匹配"hangzhou"。
     */
    matchQuery := &search.MatchQuery{}
    matchQuery.FieldName = "Col_Keyword"
    matchQuery.Text = "hangzhou"
    {
        /**
         * 构造一个BoolQuery，设置查询条件是必须同时满足"查询条件一"和"查询条件二"。
         */
        boolQuery := &search.BoolQuery{
            MustQueries: []search.Query{
                rangeQuery,
                matchQuery,
            },
```

```
        }
        searchQuery := search.NewSearchQuery()
        searchQuery.SetQuery(boolQuery)
        searchRequest.SetSearchQuery(searchQuery)
        searchResponse, err := client.Search(searchRequest)
        if err != nil {
            fmt.Printf("%#v", err)
            return
        }
        fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
        fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配的总行数，非返回行数。
        fmt.Println("RowCount: ", len(searchResponse.Rows))
    }
    {
        /**
         * 构造一个BoolQuery，设置查询条件是至少满足"条件一"和"条件二"中的一个。
         */
        boolQuery := &search.BoolQuery{
            ShouldQueries: []search.Query{
                rangeQuery,
                matchQuery,
            },
            MinimumShouldMatch: proto.Int32(1),
        }
        searchQuery := search.NewSearchQuery()
        searchQuery.SetQuery(boolQuery)
        searchRequest.SetSearchQuery(searchQuery)
        searchResponse, err := client.Search(searchRequest)
        if err != nil {
            fmt.Printf("%#v", err)
            return
        }
        fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
        fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配的总行数，非返回行数。
        fmt.Println("RowCount: ", len(searchResponse.Rows))
    }
}
```

4.7.15. 嵌套类型查询

NestedQuery用于查询嵌套类型字段中子行的数据。嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型字段的路径和一个子查询，其中子查询可以是任意Query类型。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Path	路径名，嵌套类型的列的树状路径。例如news.title表示嵌套类型的news列中的title子列。
Query	嵌套类型的列中子列上的查询，子列上的查询可以是任意Query类型。
ScoreMode	当列存在多个值时基于哪个值计算分数。

示例

类型为Nested的列，子行包含nested_1和nested_2两列，现在查询col_nested.nested_1为"tablestore"的数据。

```
func NestedQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.NestedQuery{ //设置查询类型为NestedQuery。
        Path: "col_nested", //设置嵌套类型字段的路径。
        Query: &search.TermQuery{ //构造NestedQuery的子查询。
            FieldName: "col_nested.nested_1", //设置字段名，注意带有Nested列的前缀。
            Term:      "tablestore",          //设置要查询的值。
        },
        ScoreMode: search.ScoreMode_Avg,
    }
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll: true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.16. 排序和翻页

使用多元索引时，您可以在创建时指定索引预排序和在查询时指定排序方式，在获取返回结果时使用Limit和Offset或者使用Token进行翻页。

索引预排序

多元索引默认按照设置的索引预排序（IndexSort）方式进行排序，使用多元索引查询数据时，IndexSort决定了数据的默认返回顺序。

在创建多元索引时，您可以自定义IndexSort，如果未自定义IndexSort，则IndexSort默认为主键排序。

> **注意** 含有Nested类型字段的多元索引不支持索引预排序。

查询时指定排序方式

只有EnableSortAndAgg设置为true的字段才能进行排序。

在每次查询时，可以指定排序方式，多元索引支持如下四种排序方式（Sorter）。您也可以使用多个Sorter，实现先按照某种方式排序，再按照另一种方式排序的需求。

- ScoreSort

  按照查询结果的相关性（BM25算法）分数进行排序，适用于有相关性的场景，例如全文检索等。

  > **注意** 如果需要按照相关性打分进行排序，必须手动设置ScoreSort，否则会按照索引设置的IndexSort进行排序。

```
    searchQuery := search.NewSearchQuery()
    searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.ScoreSort{
                Order: search.SortOrder_DESC.Enum(), //从得分高到低排序。
            },
        },
    })
```

- PrimaryKeySort

  按照主键进行排序。

```
searchQuery := search.NewSearchQuery()
    searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.PrimaryKeySort{
                Order: search.SortOrder_ASC.Enum(),
            },
        },
    })
```

- FieldSort

  按照某列的值进行排序。

```
    //设置按照Col_Long列逆序排序。
    searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.FieldSort{
                FieldName: "Col_Long",
                Order:     search.SortOrder_DESC.Enum(),
            },
        },
    })
```

先按照某列的值进行排序，再按照另一列的值进行排序。

```
searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.FieldSort{
                FieldName: "col1",
                Order:     search.SortOrder_ASC.Enum(),
            },
            &search.FieldSort{
                FieldName: "col2",
                Order:     search.SortOrder_DESC.Enum(),
            },
        },
    })
```

- GeoDistanceSort

  根据地理点距离进行排序。

```
    searchQuery.SetSort(&search.Sort{
        []search.Sorter{
            &search.GeoDistanceSort{
                FieldName: "location",   //设置Geo点的字段名。
                Points:    []string{"40,-70"}, //设置中心点。
            },
        },
    })
```

翻页方式

在获取返回结果时，可以使用Limit和Offset或者使用Token进行翻页。

- 使用Limit和Offset进行翻页

  当需要获取的返回结果行数小于10000行时，可以通过Limit和Offset进行翻页，即Limit+Offset<=10000，其中Limit的最大值为100。

  > **说明** 如果需要提高Limit的上限，请参见如何将多元索引Search接口查询数据的limit提高到1000。

  如果使用此方式进行翻页时未设置Limit和Offset，则Limit的默认值为10，Offset的默认值为0。

```
searchQuery := search.NewSearchQuery()
searchQuery.SetLimit(10)
searchQuery.SetOffset(10)
```

- 使用Token进行翻页

  由于使用Token进行翻页时翻页深度无限制，当需要进行深度翻页时，推荐使用Token进行翻页。

  当符合查询条件的数据未读取完时，服务端会返回NextToken，此时可以使用NextToken继续读取后面的数据。

  使用Token进行翻页时默认只能向后翻页。由于在一次查询的翻页过程中Token长期有效，您可以通过缓存并使用之前的Token实现向前翻页。

  使用Token翻页后的排序方式和上一次请求的一致，无论是系统默认使用IndexSort还是自定义排序，因此设置了Token不能再设置Sort。另外使用Token后不能设置Offset，只能依次往后读取，即无法跳页。

  > **注意** 由于含有Nested类型字段的多元索引不支持索引预排序，如果使用含有Nested类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回NextToken。

```
/**
 * 使用Token进行翻页读取。
 * 如果SearchResponse返回了NextToken，可以使用此Token发起下一次查询，直到NextToken为空(nil)。
 * NextToken为空(nil)表示所有符合条件的数据已读完。
 */
func QueryRowsWithToken(client *tablestore.TableStoreClient, tableName string, indexName
string) {
    querys := []search.Query{
        &search.MatchAllQuery{},
        &search.TermQuery{
            FieldName: "Col_Keyword",
            Term:      "tablestore",
        },
    }
    for _, query := range querys {
        fmt.Printf("Test query: %#v\n", query)
        searchRequest := &tablestore.SearchRequest{}
        searchRequest.SetTableName(tableName)
        searchRequest.SetIndexName(indexName)
        searchQuery := search.NewSearchQuery()
        searchQuery.SetQuery(query)
        searchQuery.SetLimit(10)
        searchQuery.SetGetTotalCount(true)
        searchRequest.SetSearchQuery(searchQuery)
        searchResponse, err := client.Search(searchRequest)
        if err != nil {
            fmt.Printf("%#v", err)
            return
        }
        rows := searchResponse.Rows
        requestCount := 1
        for searchResponse.NextToken != nil {
            searchQuery.SetToken(searchResponse.NextToken)
            searchResponse, err = client.Search(searchRequest)
            if err != nil {
                fmt.Printf("%#v", err)
                return
            }
            requestCount++
            for _, r := range searchResponse.Rows {
                rows = append(rows, r)
            }
        }
        fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess)
        fmt.Println("TotalCount: ", searchResponse.TotalCount)
        fmt.Println("RowsSize: ", len(rows))
        fmt.Println("RequestCount: ", requestCount)
    }
}
```

4.7.17. 列存在性查询

ExistsQuery也叫NULL查询或者空值查询，一般用于判断稀疏数据中某一行的某一列是否存在。例如查询所有数据中address列不为空的行。

> **说明**
> - 如果需要查询某一列为空，则需要和BoolQuery中的MustNotQueries结合使用。
> - 以下情况会认为某一列不存在，以 "city" 列为例说明。
>   - city列在本行数据中不存在。
>   - city列是空数组，即 "city" = "[]"。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Query	设置查询类型为ExistsQuery。
FieldName	列名。

示例

查询表中city列不为空的行。

```
func ExitsQuery(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)
    searchRequest.SetIndexName(indexName)
    query := &search.ExitsQuery{} //设置查询类型为ExitsQuery。
    query.FieldName = "city" //设置列名。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(query)
    searchQuery.SetGetTotalCount(true)
    searchRequest.SetSearchQuery(searchQuery)
    //设置为返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll:true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    fmt.Println("IsAllSuccess: ", searchResponse.IsAllSuccess) //查看返回结果是否完整。
    fmt.Println("TotalCount: ", searchResponse.TotalCount) //打印匹配到的总行数，非返回行数。
    fmt.Println("RowCount: ", len(searchResponse.Rows))
    for _, row := range searchResponse.Rows {
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.18. 折叠（去重）

当数据查询的结果中含有某种类型的数据较多时，可以使用折叠（Collapse）功能按照某一列对结果集做折叠，使对应类型的数据在结果展示中只出现一次，保证结果展示中类型的多样性。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

注意事项

- 折叠功能只能使用Offset+Limit方式翻页，不能使用Token方式。
- 对结果集同时使用统计聚合与折叠功能时，统计聚合功能只作用于使用折叠功能前的结果集。
- 使用折叠功能后，返回的总分组数取决于Offset+Limit的最大值，目前支持返回的总分组数最大为10000。
- 执行结果中返回的总行数是使用折叠功能前的匹配行数，使用折叠功能后的总分组数无法获取。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
Query	可以是任意Query类型。
Collapse	折叠参数设置，包含FieldName设置。 FieldName：列名，按该列对结果集做折叠，只支持应用于整型、浮点数和Keyword类型的列，不支持数组类型的列。
Offset	本次查询的开始位置。
Limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置Limit=0，即不返回任意一行数据。

示例

```
func QueryWithCollapse(client *tablestore.TableStoreClient, tableName string, indexName str
ing) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.SetTableName(tableName)        //设置数据表名称。
    searchRequest.SetIndexName(indexName)        //设置多元索引名称。
    searchQuery := search.NewSearchQuery()
    searchQuery.SetQuery(&search.MatchQuery{     //构造查询条件。
        FieldName:          "user_id",
        Text:               "00002",
    })
    searchQuery.SetCollapse(&search.Collapse{
        FieldName: "product_name",               //根据"product_name"列对结果集做折叠。
    })
    searchQuery.SetOffset(0)
    searchQuery.SetLimit(100)
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{ReturnAll:true}) //设置为返回所有
列。
    searchRequest.SetSearchQuery(searchQuery)
    searchResponse, err := client.Search(searchRequest)    //查询。
    if err != nil {
        fmt.Println("Failed to search with error:", err)
        return
    }
    for _, row := range searchResponse.Rows {            //打印本次返回的行。
        jsonBody, err := json.Marshal(row)
        if err != nil {
            panic(err)
        }
        fmt.Println("Row: ", string(jsonBody))
    }
}
```

4.7.19. 统计聚合

通过统计聚合接口可以实现求最小值、求最大值、求和、求平均值、统计行数、去重统计行数、按字段值分组、按范围分组、按地理位置分组、按过滤条件分组、嵌套功能；同时支持多个统计聚合功能组合使用，满足复杂的查询需求。

最小值

返回一个字段中的最小值，类似于SQL中的min。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
FieldName	用于统计聚合的字段，仅支持Long和Double类型。
Missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置Missing值，则在统计聚合时会忽略该行。 ○ 如果设置了Missing值，则使用Missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 *  商品库中有每一种商品的价格，求产地为浙江省的商品中，价格最低的商品价格是多少。
 *  等效的SQL语句是SELECT min(column_price) FROM product where place_of_production="浙江省"
;
 */
func min(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "浙江省"}).
            SetLimit(0).    //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高
性能。
            Aggregation(search.NewMinAggregation("min_agg_1", "column_price").Missing(0.0
0)))
    searchResponse, err := client.Search(searchRequest)    //执行查询。
    aggResults := searchResponse.AggregationResults        //获取统计聚合结果。
    agg1, err := aggResults.Min("min_agg_1")        //获取名称为"min_agg_1"的统计聚合结果。
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {        //名称为"min_agg_1"的统计聚合结果是否有Value值。
        fmt.Println(agg1.Value)    //打印统计聚合结果。
    }
}
```

最大值

返回一个字段中的最大值，类似于SQL中的max。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
FieldName	用于统计聚合的字段，仅支持Long和Double类型。
Missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置Missing值，则在统计聚合时会忽略该行。 ○ 如果设置了Missing值，则使用Missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的价格，求产地为浙江省的商品中，价格最高的商品价格是多少。
 * 等效的SQL语句是SELECT max(column_price) FROM product where place_of_production="浙江省"。
 */
func max(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "浙江省"}).
            SetLimit(0).    //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
            Aggregation(search.NewMaxAggregation("max_agg_1", "column_price").Missing(0.00)))
    searchResponse, err := client.Search(searchRequest)    //执行查询。
    aggResults := searchResponse.AggregationResults        //获取统计聚合结果。
    agg1, err := aggResults.Max("max_agg_1")        //获取名称为"max_agg_1"的统计聚合结果。
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {        //名称为"max_agg_1"的统计聚合结果是否有Value值。
        fmt.Println(agg1.Value)    //打印统计聚合结果。
    }
}
```

和

返回数值字段的总数，类似于SQL中的sum。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
FieldName	用于统计聚合的字段，仅支持Long和Double类型。
Missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置Missing值，则在统计聚合时会忽略该行。 ○ 如果设置了Missing值，则使用Missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的售出数量，求产地为浙江省的商品中，一共售出了多少件商品。如果某一件商品没有该值，默认售出了10件。
 * 等效的SQL语句是SELECT sum(column_price) FROM product where place_of_production="浙江省"。
 */
func sum(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "浙江省"}).
            SetLimit(0).    //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
            Aggregation(search.NewSumAggregation("sum_agg_1", "column_price").Missing(0.00)))
    searchResponse, err := client.Search(searchRequest)    //执行查询。
    aggResults := searchResponse.AggregationResults        //获取统计聚合结果。
    agg1, err := aggResults.Sum("sum_agg_1")        //获取名称为"sum_agg_1"的统计聚合结果。
    if err != nil {
        panic(err)
    }
    fmt.Println(agg1.Value)    //打印统计聚合结果。
}
```

平均值

返回数值字段的平均值，类似于SQL中的avg。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
FieldName	用于统计聚合的字段，仅支持Long和Double类型。

参数	说明
Missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置Missing值，则在统计聚合时会忽略该行。 ○ 如果设置了Missing值，则使用Missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 商品库中有每一种商品的售出数量，求产地为浙江省的商品中，平均价格是多少。
 * 等效的SQL语句是SELECT avg(column_price) FROM product where place_of_production="浙江省"。
 */
func avg(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).     //设置数据表名称。
        SetIndexName(indexName).     //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "浙江省"}).
            SetLimit(0).     //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
            Aggregation(search.NewAvgAggregation("avg_agg_1", "column_price").Missing(0.00)))
    searchResponse, err := client.Search(searchRequest)     //执行查询。
    aggResults := searchResponse.AggregationResults         //获取统计聚合结果。
    agg1, err := aggResults.Avg("avg_agg_1")         //获取名称为"avg_agg_1"的统计聚合结果。
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {         //名称为"agg1"的统计聚合结果是否有Value值。
        fmt.Println(agg1.Value)     //打印统计聚合结果。
    }
}
```

统计行数

返回指定字段值的数量或者表中数据总行数，类似于SQL中的count。

> **说明** 通过如下方式可以统计表中数据总行数或者某个query匹配的行数。
> - 使用统计聚合的count功能，在请求中设置count(*)。
> - 使用query功能的匹配行数，在query中设置setGetTotalCount(true)；如果需要统计表中数据总行数，则使用MatchAllQuery。
>
> 如果需要获取表中数据某列出现的次数，则使用count（列名），可应用于稀疏列的场景。

- 参数

参数	说明

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
FieldName	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。

- 示例

```
/**
 * 商家库中有每一种商家的惩罚记录，求浙江省的商家中，有惩罚记录的一共有多少个商家。如果商家没有惩罚记录，则商家信息中不存在该字段。
 * 等效的SQL语句是SELECT count(column_history) FROM product where place_of_production="浙江省"。
 */
func count(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "浙江省"}).
            SetLimit(0).    //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高性能。
            Aggregation(search.NewCountAggregation("count_agg_1", "column_price")))
    searchResponse, err := client.Search(searchRequest)    //执行查询。
    aggResults := searchResponse.AggregationResults        //获取统计聚合结果。
    agg1, err := aggResults.Count("count_agg_1")        //获取名称为"count_agg_1"的统计聚合结果。
    if err != nil {
        panic(err)
    }
    fmt.Println(agg1.Value)    //打印统计聚合结果。
}
```

去重统计行数

返回指定字段不同值的数量，类似于SQL中的count（distinct）。

> **说明** 去重统计行数的计算结果是个近似值。
> - 当去重统计行数小于1万时，计算结果是一个精确值。
> - 当去重统计行数达到1亿时，计算结果的误差为2%左右。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。

参数	说明
FieldName	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。
Missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置Missing值，则在统计聚合时会忽略该行。 ○ 如果设置了Missing值，则使用Missing值作为字段值的默认值参与统计聚合。

- 示例

```
/**
 * 求所有的商品，产地一共来自多少个省份。
 * 等效的SQL语句是SELECT count(distinct column_place) FROM product。
 */
func distinctCount(client *tablestore.TableStoreClient, tableName string, indexName strin
g) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.TermQuery{"place_of_production", "浙江省"}).
            SetLimit(0).    //如果只关心统计聚合结果，不关心具体数据，您可以将limit设置为0来提高
性能。
            Aggregation(search.NewDistinctCountAggregation("distinct_count_agg_1", "colum
n_price").Missing(0.00)))
    searchResponse, err := client.Search(searchRequest)    //执行查询。
    aggResults := searchResponse.AggregationResults        //获取统计聚合结果。
    agg1, err := aggResults.DistinctCount("distinct_count_agg_1")        //获取名称为"dist
inct_count_agg_1"的统计聚合结果。
    if err != nil {
        panic(err)
    }
    fmt.Println(agg1.Value)    //打印统计聚合结果。
}
```

字段值分组

根据一个字段的值对查询结果进行分组，相同的字段值放到同一分组内，返回每个分组的值和该值对应的个数。

> **说明** 当分组较大时，按字段值分组可能会存在误差。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。

参数	说明
FieldName	用于统计聚合的字段，仅支持Long、Double、Boolean和Keyword类型。
Size	返回的分组数量。
GroupBySorters	分组中的item排序规则，默认按照分组中item的数量降序排序，多个排序则按照添加的顺序进行排列。支持的参数如下： ○ 按照值的字典序升序排列 ○ 按照值的字典序降序排列 ○ 按照行数升序排列 ○ 按照行数降序排列 ○ 按照子统计聚合结果中值升序排列 ○ 按照子统计聚合结果中值降序排列
SubAggregation和SubGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 ○ 场景 统计每个类别的商品数量，且统计每个类别价格的最大值和最小值。 ○ 方法 最外层的统计聚合是根据类别进行分组，再添加两个子统计聚合求价格的最大值和最小值。 ○ 结果示例 ■ 水果：5个（其中价格最高15元，最低3元） ■ 洗漱用品：10个（其中价格最高98元，最低1元） ■ 电子设备：3个（其中价格最高8699元，最低2300元） ■ 其它：15个（其中价格最高1000元，最低80元）

- 示例

```
/**
 * 所有商品中每一个类别各有多少个，且统计每一个类别的价格最大值和最小值。
 * 返回结果举例："水果：5个（其中价格最贵15元，最便宜3元），洗漱用品：10个（其中价格最贵98元，最便宜
1元），电子设备：3个（其中价格最贵8699元，最便宜2300元），其它：15个（其中价格最贵1000
 * 元，最便宜80元）"。
 */
func GroupByField(client *tablestore.TableStoreClient, tableName string, indexName string
) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    //匹配所有行。
            SetLimit(0).
            GroupBy(search.NewGroupByField("group1", "column_type").    //按商品类别的字段
值进行分组。
                SubAggregation(search.NewMinAggregation("min_price", "column_price")).
//分组中最便宜的商品。
                SubAggregation(search.NewMaxAggregation("max_price", "column_price"))))
//分组中最贵的商品。
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    //获取统计聚合结果。
    group, err := groupByResults.GroupByField("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    //遍历返回的所有分组。
        //打印分组的值和分组中的记录行数。
        fmt.Println("\tkey: ", item.Key, ", rowCount: ", item.RowCount)
        //打印最低价格。
        minPrice, _ := item.SubAggregations.Min("min_price")
        if minPrice.HasValue() {
            fmt.Println("\t\tmin_price: ", minPrice.Value)
        }
        //打印最高价格。
        maxPrice, _ := item.SubAggregations.Max("max_price")
        if maxPrice.HasValue() {
            fmt.Println("\t\tmax_price: ", maxPrice.Value)
        }
    }
}
```

范围分组

根据一个字段的范围对查询结果进行分组，字段值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
FieldName	用于统计聚合的字段，仅支持Long和Double类型。
Range(fromInclusive float64, toExclusive float64)	分组的范围。 起始值romInclusive可以使用最小值NegInf，结束值toExclusive可以使用最大值Inf。
SubAggregation和SubGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 例如按销量分组后再按省份分组，即可获得某个销量范围内哪个省比重比较大，实现方法是GroupByRange下添加一个GroupByField。

- 示例

```
/**
 * 求商品销量时按[NegInf，1000）、[1000，5000）、[5000，Inf）这些分组计算每个范围的销量。
 */
func GroupByRange(client *tablestore.TableStoreClient, tableName string, indexName string
) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    //匹配所有行。
            SetLimit(0).
            GroupBy(search.NewGroupByRange("group1", "column_number").
                Range(search.NegInf, 1000).
                Range(1000, 5000).
                Range(5000, search.Inf)))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    //获取统计聚合结果。
    group, err := groupByResults.GroupByRange("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    //遍历返回的所有分组。
        fmt.Println("\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)    //
/打印本次分组的行数。
    }
}
```

地理位置分组

根据距离某一个中心点的范围对查询结果进行分组，距离差值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
FieldName	用于统计聚合的字段，仅支持Geo_point类型。
CenterPoint(latitude float64, longitude float64)	起始中心点的经纬度。 latitude是起始中心点坐标纬度，longitude是起始中心点坐标经度。
Range(fromInclusive float64, toExclusive float64)	分组的范围，单位为米。 起始值fromInclusive可以使用最小值NegInf，结束值toExclusive可以使用最大值Inf。
SubAggregation和SubGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

```
/**
 * 求距离万达广场[NegInf，1000）、[1000，5000）、[5000，Inf）这些范围内的人数，单位为米。
 */
func GroupByGeoDistance(client *tablestore.TableStoreClient, tableName string, indexName
string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    //匹配所有行。
            SetLimit(0).
            GroupBy(search.NewGroupByGeoDistance("group1", "Col_GeoPoint", search.GeoPoin
t{Lat: 30.137817, Lon:120.08681}).
                Range(search.NegInf, 1000).
                Range(1000, 5000).
                Range(5000, search.Inf)))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    //获取统计聚合结果。
    group, err := groupByResults.GroupByGeoDistance("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    //遍历返回的所有分组。
        fmt.Println("\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)    /
/打印本次分组的行数。
    }
}
```

过滤条件分组

按照过滤条件对查询结果进行分组，获取每个过滤条件匹配到的数量，返回结果的顺序和添加过滤条件的顺序一致。

- 参数

参数	说明
Name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
Query	过滤条件，返回结果的顺序和添加过滤条件的顺序一致。
SubAggregation和SubGroupBy	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

```
/**
 * 按照过滤条件进行分组，例如添加三个过滤条件（销量大于100、产地是浙江省、描述中包含杭州关键词），然后获取每个过滤条件匹配到的数量。
 */
func GroupByFilter(client *tablestore.TableStoreClient, tableName string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    //匹配所有行。
            SetLimit(0).
            GroupBy(search.NewGroupByFilter("group1").
                Query(&search.RangeQuery{
                    FieldName: "number",
                    From: 100,
                    IncludeLower: true}).
                Query(&search.TermQuery{
                    FieldName: "place",
                    Term:      "浙江省",
                }).
                Query(&search.MatchQuery{
                    FieldName: "description",
                    Text: "杭州",
                })))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    //获取统计聚合结果。
    group, err := groupByResults.GroupByFilter("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    //遍历返回的所有分组。
        fmt.Println("\trowCount: ", item.RowCount)    //打印本次分组的行数。
    }
}
```

嵌套

分组类型的统计聚合功能支持嵌套，其内部可以添加子统计聚合。

主要用于在分组内再次进行统计聚合，以两层的嵌套为例：

- GroupBy+SubGroupBy：按省份分组后再按照城市分组，获取每个省份下每个城市的数据。
- GroupBy+SubAggregation：按照省份分组后再求某个指标的最大值，获取每个省的某个指标最大值。

> **说明** 为了性能、复杂度等综合考虑，嵌套的层级只开放了一定的层数。更多信息，请参见多元索引限制。

示例

```
/**
 * 嵌套的统计聚合示例。
 * 外层GroupByField中添加了2个Aggregation和1个GroupByRange。
 */
func NestedSample(client *tablestore.TableStoreClient, tableName string, indexName string)
{
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    //匹配所有行。
            SetLimit(0).
            GroupBy(search.NewGroupByField("group1", "field1").
                SubAggregation(search.NewMinAggregation("sub_agg1", "sub_field1")).
                SubAggregation(search.NewMaxAggregation("sub_agg2", "sub_field2")).
                SubGroupBy(search.NewGroupByRange("sub_group1", "sub_field3").
                    Range(search.NegInf, 3).
                    Range(3, 5).
                    Range(5, search.Inf))))
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    groupByResults := searchResponse.GroupByResults    //获取统计聚合结果。
    group, err := groupByResults.GroupByField("group1")
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    for _, item := range group.Items {    //遍历返回的所有分组。
        //打印分组的值和分组中的记录行数。
        fmt.Println("\tkey: ", item.Key, ", rowCount: ", item.RowCount)
        //获取名称为"sub_agg1"的统计聚合结果。
        subAgg1, _ := item.SubAggregations.Min("sub_agg1")
        if subAgg1.HasValue() {
            fmt.Println("\t\tsub_agg1: ", subAgg1.Value)
        }
        //获取名称为"sub_agg2"的统计聚合结果。
        subAgg2, _ := item.SubAggregations.Max("sub_agg2")
        if subAgg2.HasValue() {
            fmt.Println("\t\tsub_agg2: ", subAgg2.Value)
        }
        //获取名称为"sub_group1"的统计聚合结果。
        subGroup, _ := item.SubGroupBys.GroupByRange("sub_group1")
        for _, item := range subGroup.Items {
            fmt.Println("\t\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)
        }
    }
}
```

多个统计聚合

多个统计聚合功能可以组合使用。

> **说明** 当多个统计聚合的复杂度较高时可能会影响响应速度。

- 示例1

```
func MultipleAggregations(client *tablestore.TableStoreClient, tableName string, indexNam
e string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).    //设置数据表名称。
        SetIndexName(indexName).    //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).    //匹配所有行。
            SetLimit(0).
            Aggregation(search.NewAvgAggregation("agg1", "Col_Long")).                //
计算Col_Long字段的平均值。
            Aggregation(search.NewDistinctCountAggregation("agg2", "Col_Long")).    //计算
Col_Long字段不同取值的个数。
            Aggregation(search.NewMaxAggregation("agg3", "Col_Long")).                //
计算Col_Long字段的最大值。
            Aggregation(search.NewSumAggregation("agg4", "Col_Long")).                //
计算Col_Long字段的和。
            Aggregation(search.NewCountAggregation("agg5", "Col_Long")))            //计算
存在Col_Long字段的行数。
    //设置返回所有列。
    searchRequest.SetColumnsToGet(&tablestore.ColumnsToGet{
        ReturnAll: true,
    })
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    aggResults := searchResponse.AggregationResults    //获取统计聚合结果。
    //获取求平均值的统计聚合结果。
    agg1, err := aggResults.Avg("agg1")        //获取名称为"agg1"的统计聚合结果，类型为Avg。
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {                            //名称为"agg1"的统计聚合结果是否有Value值
。
        fmt.Println("(avg) agg1: ", agg1.Value)    //打印Col_Long字段平均值。
    } else {
        fmt.Println("(avg) agg1: no value")        //所有行都不存在Col_Long字段时的打印信息。
    }
    //获取去重统计行数的统计聚合结果。
    agg2, err := aggResults.DistinctCount("agg2")    //获取名称为"agg2"的统计聚合结果，类型为
DistinctCount。
    if err != nil {
        panic(err)
    }
```

```
    fmt.Println("(distinct) agg2: ", agg2.Value)        //打印Col_Long字段不同取值的个数。
    //获取求最大值的统计聚合结果。
    agg3, err := aggResults.Max("agg3")         //获取名称为"agg3"的统计聚合结果，类型为Max。
    if err != nil {
        panic(err)
    }
    if agg3.HasValue() {
        fmt.Println("(max) agg3: ", agg3.Value)     //打印Col_Long字段最大值。
    } else {
        fmt.Println("(max) agg3: no value")         //所有行都不存在Col_Long字段时的打印信息。
    }
    //获取求和的统计聚合结果。
    agg4, err := aggResults.Sum("agg4")         //获取名称为"agg4"的统计聚合结果，类型为Sum。
    if err != nil {
        panic(err)
    }
    fmt.Println("(sum) agg4: ", agg4.Value)         //打印Col_Long字段的和。
    //获取统计行数的统计聚合结果。
    agg5, err := aggResults.Count("agg5")         //获取名称为"agg5"的统计聚合结果，类型为Coun
t。
    if err != nil {
        panic(err)
    }
    fmt.Println("(count) agg6: ", agg5.Value)     //打印存在Col_Long字段的个数。
}
```

- 示例2

```
func MultipleAggregationsAndGroupBysSample(client *tablestore.TableStoreClient, tableName
string, indexName string) {
    searchRequest := &tablestore.SearchRequest{}
    searchRequest.
        SetTableName(tableName).     //设置数据表名称。
        SetIndexName(indexName).     //设置多元索引名称。
        SetSearchQuery(search.NewSearchQuery().
            SetQuery(&search.MatchAllQuery{}).     //匹配所有行。
            SetLimit(0).
            Aggregation(search.NewAvgAggregation("agg1", "Col_Long")).                //
计算Col_Long字段的平均值。
            Aggregation(search.NewDistinctCountAggregation("agg2", "Col_Long")).     //计算
Col_Long字段不同取值的个数。
            Aggregation(search.NewMaxAggregation("agg3", "Col_Long")).                //
计算Col_Long字段的最大值。
            GroupBy(search.NewGroupByField("group1", "Col_Keyword").     //对Col_Keyword字
段做GroupByField取值统计聚合。
                GroupBySorters([]search.GroupBySorter{}).     //指定返回结果分组的顺序。
                Size(2).                                      //仅返回前2个分组。
                SubAggregation(search.NewAvgAggregation("sub_agg1", "Col_Long")).     //对
每个分组进行子统计聚合。
                SubGroupBy(search.NewGroupByField("sub_group1", "Col_Keyword2"))).     //
对每个分组进行子统计聚合。
            GroupBy(search.NewGroupByRange("group2", "Col_Long").         //对Col_Long字段
做GroupByRange范围。
                Range(search.NegInf, 3).             //第一个分组包含Col_Long在(NegInf, 3)的
索引行。
```

```
                Range(3, 5).            //第二个分组包含Col_Long在[3, 5)的索引行。
                Range(5, search.Inf)))            //第三个分组包含Col_Long在[5, Inf)的索引行
。
    // 设置返回所有列。
    searchResponse, err := client.Search(searchRequest)
    if err != nil {
        fmt.Printf("%#v", err)
        return
    }
    aggResults := searchResponse.AggregationResults    //获取统计聚合结果。
    //获取求平均值的统计聚合结果。
    agg1, err := aggResults.Avg("agg1")        //获取名称为"agg1"的统计聚合结果，类型为Avg。
    if err != nil {
        panic(err)
    }
    if agg1.HasValue() {                          //名称为"agg1"的统计聚合结果是否有Value值
。
        fmt.Println("(avg) agg1: ", agg1.Value)    //打印Col_Long字段平均值。
    } else {
        fmt.Println("(avg) agg1: no value")        //所有行都不存在Col_Long字段时的打印信息。
    }
    //获取去重统计行数的统计聚合结果。
    agg2, err := aggResults.DistinctCount("agg2")    //获取名称为"agg2"的统计聚合结果，类型为
DistinctCount。
    if err != nil {
        panic(err)
    }
    fmt.Println("(distinct) agg2: ", agg2.Value)        //打印Col_Long字段不同取值的个数。
    //获取求最大值的统计聚合结果。
    agg3, err := aggResults.Max("agg3")        //获取名称为"agg3"的统计聚合结果，类型为Max。
    if err != nil {
        panic(err)
    }
    if agg3.HasValue() {
        fmt.Println("(max) agg3: ", agg3.Value)    //打印Col_Long字段最大值。
    } else {
        fmt.Println("(max) agg3: no value")        //所有行都不存在Col_Long字段时的打印信息。
    }
    groupByResults := searchResponse.GroupByResults    //获取统计聚合结果。
    //获取按字段值分组的统计聚合结果。
    group1, err := groupByResults.GroupByField("group1")    //获取名称为"group1"的统计聚合结
果，类型为GroupByField。
    if err != nil {
        panic(err)
    }
    fmt.Println("group1: ")
    for _, item := range group1.Items {    //遍历返回的所有分组。
        //item
        fmt.Println("\tkey: ", item.Key, ", rowCount: ", item.RowCount)    //打印本次分组的
行数。
        //获取求平均值的子统计聚合结果。
        subAgg1, err := item.SubAggregations.Avg("sub_agg1")    //获取名称为"sub_agg1"的子
统计聚合结果，类型为Avg。
        if err != nil {
```

```
            panic(err)
        }
        if subAgg1.HasValue() {    //如果子统计聚合"sub_agg1"计算出了Col_Long字段的平均值，则HasValue()返回true。
            fmt.Println("\t\tsub_agg1: ", subAgg1.Value)    //打印本次分组中，子统计聚合计算的Col_Long字段的平均值。
        }
        //获取按字段值分组的子统计聚合结果。
        subGroup1, err := item.SubGroupBys.GroupByField("sub_group1")    //获取名称为"sub_group1"的子统计聚合结果，类型为GroupByField。
        if err != nil {
            panic(err)
        }
        fmt.Println("\t\tsub_group1")
        for _, subItem := range subGroup1.Items {    //遍历名称为"sub_group1"的子统计聚合结果。
            fmt.Println("\t\t\tkey: ", subItem.Key, ", rowCount: ", subItem.RowCount)    //打印"sub_group1"的子统计聚合结果分组，即分组中的行数。
            tablestore.Assert(subItem.SubAggregations.Empty(), "")
            tablestore.Assert(subItem.SubGroupBys.Empty(), "")
        }
    }
    //获取按范围分组的统计聚合结果。
    group2, err := groupByResults.GroupByRange("group2")    //获取名称为"group2"的统计聚合结果，类型为GroupByRange。
    if err != nil {
        panic(err)
    }
    fmt.Println("group2: ")
    for _, item := range group2.Items {    //遍历返回的所有分组。
        fmt.Println("\t[", item.From, ", ", item.To, "), rowCount: ", item.RowCount)    //打印本次分组的行数。
    }
}
```

4.7.20. 并发导出数据

当使用场景中不关心整个结果集的顺序时，可以使用并发导出数据（ParallelScan）功能以更快的速度将匹配的数据全部返回。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数		说明
TableName		数据表名称。
IndexName		多元索引名称。
ScanQuery	Query	多元索引的查询语句。支持精确查询、模糊查询、范围查询、地理位置查询、嵌套查询等，功能和Search接口一致。
	Limit	扫描数据时一次能返回的数据行数。
	MaxParallel	最大并发数。请求支持的最大并发数由用户数据量决定。数据量越大，支持的并发数越多，每次任务前可以通过ComputeSplits API进行获取。
	CurrentParallelID	当前并发ID。取值范围为[0, MaxParallel)。
	Token	用于翻页功能。ParallelScan请求结果中有下一次进行翻页的Token，使用该Token可以接着上一次的结果继续读取数据。
	AliveTime	ParallelScan的当前任务有效时间，也是Token的有效时间。默认值为60，建议使用默认值，单位为秒。如果在有效时间内没有发起下一次请求，则不能继续读取数据。持续发起请求会刷新Token有效时间。 **说明** 由于服务端采用异步方式清理过期任务，因此当前任务只保证在设置的有效时间内不会过期，但不能保证有效时间之后一定过期。
ColumnsToGet		ParallelScan目前仅可以扫描多元索引中的数据，需要在创建多元索引时设置附加存储（即Store=true）。
SessionId		本次并发扫描数据任务的SessionId。您可以通过ComputeSplits API创建Session，同时获得本次任务支持的最大并发数。

示例

单并发扫描数据和多线程并发扫描数据的代码示例如下：

- 单并发扫描数据

```
/**
 * ParallelScan单并发扫描数据。
 */
func ParallelScanSingleConcurrency(client *tablestore.TableStoreClient, tableName string,
indexName string) {
        computeSplitsResp, err := computeSplits(client, tableName, indexName)
        if err != nil {
                fmt.Printf("%#v", err)
                return
        }

        query := search.NewScanQuery().SetQuery(&search.MatchAllQuery{}).SetLimit(2)

        req := &tablestore.ParallelScanRequest{}
        req.SetTableName(tableName).
                SetIndexName(indexName).
                SetColumnsToGet(&tablestore.ColumnsToGet{ReturnAllFromIndex: false}).
                SetScanQuery(query).
                SetSessionId(computeSplitsResp.SessionId)

        res, err := client.ParallelScan(req)
        if err != nil {
                fmt.Printf("%#v", err)
                return
        }

        total := len(res.Rows)
        for res.NextToken != nil {
                req.SetScanQuery(query.SetToken(res.NextToken))
                res, err = client.ParallelScan(req)
                if err != nil {
                        fmt.Printf("%#v", err)
                        return
                }

                total += len(res.Rows) //process rows each loop
        }
        fmt.Println("total: ", total)
}
```

- 多线程并发扫描数据

```
/**
 * ParallelScan多并发扫描数据。
 */
func ParallelScanMultiConcurrency(client *tablestore.TableStoreClient, tableName string,
indexName string) {
        computeSplitsResp, err := computeSplits(client, tableName, indexName)
        if err != nil {
                fmt.Printf("%#v", err)
                return
        }

        var wg sync.WaitGroup
```

```
        wg.Add(int(computeSplitsResp.SplitsSize))

        for i := int32(0); i < computeSplitsResp.SplitsSize; i++ {
                current := i
                go func() {
                        defer wg.Done()
                        query := search.NewScanQuery().
                                SetQuery(&search.MatchAllQuery{}).
                                SetCurrentParallelID(current).
                                SetMaxParallel(computeSplitsResp.SplitsSize).
                                SetLimit(2)

                        req := &tablestore.ParallelScanRequest{}
                        req.SetTableName(tableName).
                                SetIndexName(indexName).
                                SetColumnsToGet(&tablestore.ColumnsToGet{ReturnAllFromInd
ex: false}).
                                SetScanQuery(query).
                                SetSessionId(computeSplitsResp.SessionId)

                        res, err := client.ParallelScan(req)
                        if err != nil {
                                fmt.Printf("%#v", err)
                                return
                        }

                        total := len(res.Rows)
                        for res.NextToken != nil {
                                req.SetScanQuery(query.SetToken(res.NextToken))
                                res, err = client.ParallelScan(req)
                                if err != nil {
                                        fmt.Printf("%#v", err)
                                        return
                                }

                                total += len(res.Rows) //process rows each loop
                        }
                        fmt.Println("total: ", total)
                }()
        }
        wg.Wait()
}
```

4.8. 二级索引

4.8.1. 全局二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client。具体操作，请参见初始化。

- 已创建数据表，且数据表的数据生命周期（TimeToLive）必须为-1，最大版本数（MaxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
MainTableName	数据表名称。
IndexMeta	索引表的结构信息，包括如下内容： ○ IndexName：索引表名称。 ○ PrimaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ DefinedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ IndexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置IndexType或者设置IndexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局二级索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置IndexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。
IncludeBaseData	索引表中是否包含数据表中已存在的数据。 当设置IncludeBaseData为true时，表示包含存量数据；设置IncludeBaseData为false时，表示不包含存量数据。

- 示例

  在主键为pk1、pk2的数据表上创建主键列为definedcol1，属性列为definedcol2的索引表。

```
func CreateGlobalIndexSample(client *tablestore.TableStoreClient, tableName string) {
    indexMeta := new(tablestore.IndexMeta) //新建索引表Meta。
    indexMeta.AddPrimaryKeyColumn("definedcol1") //设置数据表的definedcol1列作为索引表的主键
。
    indexMeta.AddDefinedColumn("definedcol2") //设置数据表的definedcol2列作为索引表的属性列。
    indexMeta.IndexName = "indexSample"
    indexReq := &tablestore.CreateIndexRequest{
        MainTableName:tableName, //添加索引表到数据表。
        IndexMeta: indexMeta,
        IncludeBaseData: true, //包含存量数据。
    }
    /**
      通过将IncludeBaseData参数设置为true，创建索引表后会开启数据表中存量数据的同步，然后可以通过
索引表查询全部数据，
      同步时间跟数据量的大小有一定的关系。
    */
    resp, err := client.CreateIndex(indexReq)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create index finished", resp)
    }
}
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - TableName需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - TableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

```
func GetRangeFromIndex(client *tablestore.TableStoreClient, indexName string) {
    getRangeRequest := &tablestore.GetRangeRequest{}
    rangeRowQueryCriteria := &tablestore.RangeRowQueryCriteria{}
    rangeRowQueryCriteria.TableName = indexName
    startPK := new(tablestore.PrimaryKey)
    startPK.AddPrimaryKeyColumnWithMinValue("definedcol1") //索引表的第一主键列。
    startPK.AddPrimaryKeyColumnWithMinValue("pk1") //索引表的第二主键列，此主键列为自动补齐的数据表主键列。
    startPK.AddPrimaryKeyColumnWithMinValue("pk2") //索引表的第三主键列，此主键列为自动补齐的数据表主键列。
    endPK := new(tablestore.PrimaryKey)
    endPK.AddPrimaryKeyColumnWithMaxValue("definedcol1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk2")
    rangeRowQueryCriteria.StartPrimaryKey = startPK
    rangeRowQueryCriteria.EndPrimaryKey = endPK
    rangeRowQueryCriteria.Direction = tablestore.FORWARD
    rangeRowQueryCriteria.MaxVersion = 1
    rangeRowQueryCriteria.Limit = 10
    getRangeRequest.RangeRowQueryCriteria = rangeRowQueryCriteria
    getRangeResp, err := client.GetRange(getRangeRequest)
    fmt.Println("get range result is ", getRangeResp)
    for {
        if err != nil {
            fmt.Println("get range failed with error:", err)
        }
        if len(getRangeResp.Rows) > 0 {
            for _, row := range getRangeResp.Rows {
                fmt.Println("range get row with key", row.PrimaryKey.PrimaryKeys[0].Value, row.PrimaryKey.PrimaryKeys[1].Value, row.PrimaryKey.PrimaryKeys[2].Value)
            }
            if getRangeResp.NextStartPrimaryKey == nil {
                break
            } else {
                fmt.Println("next pk is :", getRangeResp.NextStartPrimaryKey.PrimaryKeys[0].Value, getRangeResp.NextStartPrimaryKey.PrimaryKeys[1].Value, getRangeResp.NextStartPrimaryKey.PrimaryKeys[2].Value)
                getRangeRequest.RangeRowQueryCriteria.StartPrimaryKey = getRangeResp.NextStartPrimaryKey
                getRangeResp, err = client.GetRange(getRangeRequest)
            }
        } else {
            break
        }
        fmt.Println("continue to query rows")
    }
    fmt.Println("putrow finished")
}
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
MainTableName	数据表名称。
IndexName	索引表名称。

- 示例

```
func DeleteIndex(client *tablestore.TableStoreClient, tableName string, indexName string
) {
    deleteIndex := &tablestore.DeleteIndexRequest{ MainTableName:tableName, IndexName: in
dexName }
    resp, err := client.DeleteIndex(deleteIndex)
    if err != nil {
        fmt.Println("Failed to delete index:", err)
    } else {
        fmt.Println("drop index finished", resp)
    }
```

4.8.2. 本地二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（TimeToLive）必须为-1，最大版本数（MaxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
MainTableName	数据表名称。

参数	说明
IndexMeta	索引表的结构信息，包括如下内容： ○ IndexName：索引表名称。 ○ PrimaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ DefinedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ IndexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置IndexType或者设置IndexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局二级索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置IndexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。
IncludeBaseData	索引表中是否包含数据表中已存在的数据。 当设置IncludeBaseData为true时，表示包含存量数据；设置IncludeBaseData为false时，表示不包含存量数据。

- 示例

  在主键为pk1、pk2的数据表上创建主键列为pk1、definedcol1，属性列为definedcol2的索引表。

```
func CreateGlobalIndexSample(client *tablestore.TableStoreClient, tableName string) {
    indexMeta := new(tablestore.IndexMeta) //新建索引表Meta。
    indexMeta.AddPrimaryKeyColumn("pk1") ///索引表的第一列主键必须与数据表的第一列主键相同。
    indexMeta.AddPrimaryKeyColumn("definedcol1") //设置数据表的definedcol1列作为索引表的主键
。
    indexMeta.AddDefinedColumn("definedcol2") //设置数据表的definedcol2列作为索引表的属性列。
    indexMeta.IndexType = IT_LOCAL_INDEX //设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    indexMeta.IndexName = "indexSample"
    indexReq := &tablestore.CreateIndexRequest{
        MainTableName:tableName, //添加索引表到数据表。
        IndexMeta: indexMeta,
        IncludeBaseData: true, //包含存量数据。
    }
    /**
      通过将IncludeBaseData参数设置为true，创建索引表后会开启数据表中存量数据的同步，然后可以通过
索引表查询全部数据，
      同步时间跟数据量的大小有一定的关系。
    */
    resp, err := client.CreateIndex(indexReq)
    if err != nil {
        fmt.Println("Failed to create table with error:", err)
    } else {
        fmt.Println("Create index finished", resp)
    }
}
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - TableName需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - TableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 示例

```
func GetRangeFromIndex(client *tablestore.TableStoreClient, indexName string) {
    getRangeRequest := &tablestore.GetRangeRequest{}
    rangeRowQueryCriteria := &tablestore.RangeRowQueryCriteria{}
    rangeRowQueryCriteria.TableName = indexName
    startPK := new(tablestore.PrimaryKey)
    startPK.AddPrimaryKeyColumnWithMinValue("pk1") //索引表的第一主键列。
    startPK.AddPrimaryKeyColumnWithMinValue("definedcol1") //索引表的第二主键列。
    startPK.AddPrimaryKeyColumnWithMinValue("pk2") //索引表的第三主键列，此主键列为自动补齐的数据表主键列。
    endPK := new(tablestore.PrimaryKey)
    endPK.AddPrimaryKeyColumnWithMaxValue("pk1")
    endPK.AddPrimaryKeyColumnWithMaxValue("definedcol1")
    endPK.AddPrimaryKeyColumnWithMaxValue("pk2")
    rangeRowQueryCriteria.StartPrimaryKey = startPK
    rangeRowQueryCriteria.EndPrimaryKey = endPK
    rangeRowQueryCriteria.Direction = tablestore.FORWARD
    rangeRowQueryCriteria.MaxVersion = 1
    rangeRowQueryCriteria.Limit = 10
    getRangeRequest.RangeRowQueryCriteria = rangeRowQueryCriteria
    getRangeResp, err := client.GetRange(getRangeRequest)
    fmt.Println("get range result is ", getRangeResp)
    for {
        if err != nil {
            fmt.Println("get range failed with error:", err)
        }
        if len(getRangeResp.Rows) > 0 {
            for _, row := range getRangeResp.Rows {
                fmt.Println("range get row with key", row.PrimaryKey.PrimaryKeys[0].Value, row.PrimaryKey.PrimaryKeys[1].Value, row.PrimaryKey.PrimaryKeys[2].Value)
            }
            if getRangeResp.NextStartPrimaryKey == nil {
                break
            } else {
                fmt.Println("next pk is :", getRangeResp.NextStartPrimaryKey.PrimaryKeys[0].Value, getRangeResp.NextStartPrimaryKey.PrimaryKeys[1].Value, getRangeResp.NextStartPrimaryKey.PrimaryKeys[2].Value)
                getRangeRequest.RangeRowQueryCriteria.StartPrimaryKey = getRangeResp.NextStartPrimaryKey
                getRangeResp, err = client.GetRange(getRangeRequest)
            }
        } else {
            break
        }
        fmt.Println("continue to query rows")
    }
    fmt.Println("putrow finished")
}
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
MainTableName	数据表名称。
IndexName	索引表名称。

- 示例

```
func DeleteIndex(client *tablestore.TableStoreClient, tableName string,  indexName string
) {
    deleteIndex := &tablestore.DeleteIndexRequest{ MainTableName:tableName, IndexName: in
dexName }
    resp, err := client.DeleteIndex(deleteIndex)
    if err != nil {
        fmt.Println("Failed to delete index:", err)
    } else {
        fmt.Println("drop index finished", resp)
    }
```

4.9. 通道服务

4.9.1. 安装

本文介绍通道服务的安装。

下载源码包

```
go get github.com/aliyun/aliyun-tablestore-go-sdk/tunnel
```

安装依赖

- 在tunnel目录下使用dep安装依赖。
  - 安装dep
  - dep ensure -v
- 直接使用go get安装依赖包。

```
go get -u go.uber.org/zap
go get github.com/cenkalti/backoff
go get github.com/golang/protobuf/proto
go get github.com/satori/go.uuid
go get github.com/stretchr/testify/assert
go get github.com/smartystreets/goconvey/convey
go get github.com/golang/mock/gomock
go get gopkg.in/natefinch/lumberjack.v2
```

4.9.2. 快速开始

使用Go SDK快速体验通道服务。使用前，您需要了解使用通道服务的注意事项。

注意事项

- TunnelWorkerConfig中默认会启动读数据和处理数据的线程池。如果使用的是单台机器，则会启动多个TunnelWorker，强烈建议共用一个TunnelWorkerConfig。
- 在创建全量加增量类型的Tunnel时，由于Tunnel的增量日志最多会保留7天（具体的值和数据表的Stream的日志过期时间一致），全量数据如果在7天内没有消费完成，则此Tunnel进入增量阶段会出现OTSTunnelExpired错误，导致增量数据无法消费。如果您预估全量数据无法在7天内消费完成，请及时联系表格存储技术支持或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。
- TunnelWorker的初始化需要预热时间，该值受TunnelWorkerConfig中的HeartbeatInterval参数影响，可以通过TunnelWorkerConfig中的time方法配置，默认值为30s，最小值为5s。
- 当Tunnel从全量切换至增量阶段时，全量的Channel会结束，增量的Channel会启动，此阶段会有初始化时间，该值也受TunnelWorkerConfig中的HeartbeatInterval参数影响。
- 当客户端（TunnelWorker）没有被正常shutdown时（例如异常退出或者手动结束），TunnelWorker会自动进行资源的回收，包括释放线程池，自动调用用户在Channel上注册的shutdown方法，关闭Tunnel连接等。

体验通道服务

1. 初始化Tunnel client。

```
//endpoint是表格存储实例endpoint，例如https://instance.cn-hangzhou.ots.aliyun.com。
//instance是实例名称。
//accessKeyId和accessKeySecret分别为访问表格存储服务的AccessKey的Id和Secret。
tunnelClient := tunnel.NewTunnelClient(endpoint, instance,
   accessKeyId, accessKeySecret)
```

2. 创建通道。

```
req := &tunnel.CreateTunnelRequest{
   TableName:  "testTable",
   TunnelName: "testTunnel",
   Type:       tunnel.TunnelTypeBaseStream, //创建全量加增量类型的Tunnel。
}
resp, err := tunnelClient.CreateTunnel(req)
if err != nil {
   log.Fatal("create test tunnel failed", err)
}
log.Println("tunnel id is", resp.TunnelId)
```

3. 根据业务自定义数据消费Callback函数，开始自动化的数据消费。

```
//根据业务自定义数据消费callback函数。
func exampleConsumeFunction(ctx *tunnel.ChannelContext, records []*tunnel.Record) error
{
    fmt.Println("user-defined information", ctx.CustomValue)
    for _, rec := range records {
        fmt.Println("tunnel record detail:", rec.String())
    }
    fmt.Println("a round of records consumption finished")
    return nil
}
//配置callback到SimpleProcessFactory，配置消费端TunnelWorkerConfig。
workConfig := &tunnel.TunnelWorkerConfig{
   ProcessorFactory: &tunnel.SimpleProcessFactory{
      CustomValue: "user custom interface{} value",
      ProcessFunc: exampleConsumeFunction,
   },
}
//使用TunnelDaemon持续消费指定tunnel。
daemon := tunnel.NewTunnelDaemon(tunnelClient, tunnelId, workConfig)
log.Fatal(daemon.Run())
```

4.9.3. 配置项

本文介绍通道服务的配置项。

tunnel client

初始化tunnel client时可以通过NewTunnelClientWithConfig接口自定义客户端配置，如果未指定config进行接口初始化或者config为nil时，则默认配置会使用DefaultTunnelConfig。

```
var DefaultTunnelConfig = &TunnelConfig{
      //最大指数退避重试时间。
      MaxRetryElapsedTime: 45 * time.Second,
      //HTTP请求超时时间。
      RequestTimeout:      30 * time.Second,
      //http.DefaultTransport。
      Transport:           http.DefaultTransport,
}
```

数据消费worker

TunnelWorkerConfig中包含了数据消费worker需要的配置，其中ProcessorFactory为必填项，其余参数如果不填写将使用默认值，通常使用默认值即可。

```
type TunnelWorkerConfig struct {
   //worker同Tunnel服务的心跳超时时间，通常使用默认值即可。
   HeartbeatTimeout  time.Duration
   //worker发送心跳的频率，通常使用默认值即可。
   HeartbeatInterval time.Duration
   //tunnel下消费连接建立接口，通常使用默认值即可。
   ChannelDialer     ChannelDialer
   //消费连接上具体处理器产生接口，通常使用callback函数初始化SimpleProcessFactory即可。
   ProcessorFactory ChannelProcessorFactory
   //zap日志配置，默认值为DefaultLogConfig。
   LogConfig      *zap.Config
   //zap日志轮转配置，默认值为DefaultSyncer。
   LogWriteSyncer zapcore.WriteSyncer
}
```

其中ProcessorFactory为用户注册消费callback函数以及其他信息的接口，建议使用SDK中自带SimpleProcessorFactory实现。

```
type SimpleProcessFactory struct {
   //用户自定义信息，会传递到ProcessFunc和ShutdownFunc中的ChannelContext参数中。
   CustomValue interface{}
   //Worker记录checkpoint的间隔，CpInterval<=0时会使用DefaultCheckpointInterval。
   CpInterval time.Duration
   //worker数据处理的同步调用callback，ProcessFunc返回error时worker会用本批数据退避重试ProcessFunc。
   ProcessFunc  func(channelCtx *ChannelContext, records []*Record) error
   //worker退出时的同步调用callback。
   ShutdownFunc func(channelCtx *ChannelContext)
   //日志配置，Logger为nil时会使用DefaultLogConfig初始化logger。
   Logger *zap.Logger
}
```

日志

默认日志配置和日志轮转配置示例如下：

- 默认日志配置

```
//DefaultLogConfig是TunnelWorkerConfig和SimpleProcessFactory使用的默认日志配置。
var DefaultLogConfig = zap.Config{
   Level:       zap.NewAtomicLevelAt(zap.InfoLevel),
   Development: false,
   Sampling: &zap.SamplingConfig{
      Initial:    100,
      Thereafter: 100,
   },
   Encoding: "json",
   EncoderConfig: zapcore.EncoderConfig{
      TimeKey:        "ts",
      LevelKey:       "level",
      NameKey:        "logger",
      CallerKey:      "caller",
      MessageKey:     "msg",
      StacktraceKey:  "stacktrace",
      LineEnding:     zapcore.DefaultLineEnding,
      EncodeLevel:    zapcore.LowercaseLevelEncoder,
      EncodeTime:     zapcore.ISO8601TimeEncoder,
      EncodeDuration: zapcore.SecondsDurationEncoder,
      EncodeCaller:   zapcore.ShortCallerEncoder,
   },
}
```

- 日志轮转配置

```
//DefaultSyncer是TunnelWorkerConfig和SimpleProcessFactory使用的默认日志轮转配置
var DefaultSyncer = zapcore.AddSync(&lumberjack.Logger{
   //日志文件路径。
   Filename:   "tunnelClient.log",
   //最大日志文件大小。
   MaxSize:    512, //MB
   //压缩轮转的日志文件数。
   MaxBackups: 5,
   //轮转日志文件保留的最大天数。
   MaxAge:     30, //days
   //是否压缩轮转日志文件。
   Compress:   true,
})
```

4.9.4. 创建通道

CreateTunnel操作为某张数据表创建一个通道，一张数据表上可以创建多个通道。在创建通道时需要指定数据表名称、通道名称和通道类型。

请求参数

参数	说明
TableName	创建通道的数据表名称。
TunnelName	通道的名称。

参数	说明
Type	通道的类型，支持全量（BaseData）、增量（Stream）和全量加增量（BaseAndStream）三种。

响应参数

参数	说明
TunnelId	通道的ID。
ResponseInfo	返回的一些其它字段，包括当次请求的RequestId。 RequestId用于唯一标识此次请求。

示例

```
req := &tunnel.CreateTunnelRequest{
   TableName:  "testTable",
   TunnelName: "testTunnel",
   Type:       tunnel.TunnelTypeBaseStream, //创建全量加增量类型的Tunnel。
}
resp, err := tunnelClient.CreateTunnel(req)
if err != nil {
   log.Fatal("create test tunnel failed", err)
}
log.Println("tunnel id is", resp.TunnelId)
```

4.9.5. 获取通道的具体信息

DescribeTunnel操作描述某个通道里的具体Channel信息。目前一个Channel对应Tablestore Stream接口的一个数据分片。

请求参数

参数	说明
TableName	需要获取通道信息的数据表名称。
TunnelName	通道的名称。

响应参数

参数	说明
TunnelRPO	通道消费增量数据的最新时间点，其值等于Tunnel中消费最慢的Channel的时间点，默认值为1970年1月1日（UTC）。

参数	说明
List<TunnelInfo>	通道信息的列表，包含如下信息： • TunnelId：通道的ID。 • TunnelName：通道的名称。 • TunnelType：通道的类型，包括全量（BaseData）、增量（Stream）和全量加增量（BaseAndStream）三种。 • TableName：该通道所在的数据表名称。 • InstanceName：该通道所在的实例名称。 • Stage：该通道所处的阶段，包括初始化（InitBaseDataAndStreamShard），全量处理（ProcessBaseData）和增量处理（ProcessStream）三种。 • Expired：数据是否超期。 如果该值返回true，请及时通过钉钉联系表格存储技术支持。
List<ChannelInfo>	通道中的Channel信息列表，包含如下信息： • ChannelId：Channel对应的ID。 • ChannelType：Channel的类型，包括全量（Base）和增量（Stream）两种。 • ChannelStatus：Channel的状态，包括等待（WAIT）、打开（OPEN）、关闭中（CLOSING）、关闭（CLOSE）和结束（TERMINATED）五种。 • ClientId：通道客户端的ID标识，默认由客户端主机名（可以在TunnelWorkerConfig中自定义）和随机串拼接而成。 • ChannelRPO：Channel消费增量数据的最新时间点，默认值为1970年1月1日（UTC），全量类型无此概念。
ResponseInfo	返回的一些其它字段，包括当次请求的RequestId。 RequestId用于唯一标识此次请求。

示例

```
req := &tunnel.DescribeTunnelRequest{
   TableName:  "testTable",
   TunnelName: "testTunnel",
}
resp, err := tunnelClient.DescribeTunnel(req)
if err != nil {
   log.Fatal("describe test tunnel failed", err)
}
log.Println("tunnel id is", resp.Tunnel.TunnelId)
```

4.9.6. 删除通道

DeleteTunnel操作为某张数据表删除一个通道，删除时需要指定数据表名称和通道名称。

请求参数

参数	说明
TableName	需要删除通道的数据表名称。
TunnelName	通道的名称。

响应参数

参数	说明
ResponseInfo	返回的一些其它字段，包括当次请求的RequestId。 RequestId用于唯一标识此次请求。

示例

```
req := &tunnel.DeleteTunnelRequest {
   TableName: "testTable",
   TunnelName: "testTunnel",
}
_, err := tunnelClient.DeleteTunnel(req)
if err != nil {
   log.Fatal("delete test tunnel failed", err)
}
```

4.10. 数据湖投递

4.10.1. 创建投递任务

使用CreateDeliveryTask接口创建一个投递任务。通过创建投递任务，您可以将表格存储数据表中的数据投递到OSS Bucket中存储。

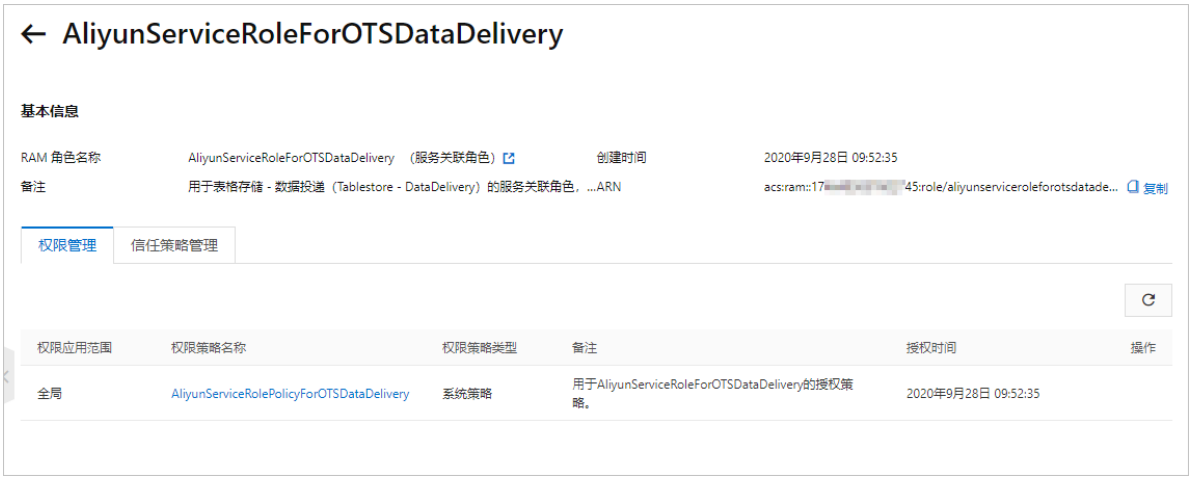
> **注意**
>
> 请确认已安装支持数据湖投递功能的表格存储Go SDK。

前提条件

- 已开通OSS服务且在表格存储实例所在地域创建Bucket。具体操作，请参见开通OSS服务。
- 已通过控制台创建表格存储服务关联角色并记录角色的ARN。具体操作，请参见创建投递任务。

  服务关联角色的ARN请通过RAM控制台获取，具体操作如下：

在**RAM 角色管理**界面，搜索AliyunServiceRoleForOTSDataDelivery后，单击角色名称，在角色详情界面，可以查看和复制角色的ARN信息。



- 已初始化TableStoreClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。

参数

参数	说明
TableName	数据表名称。
TaskName	投递任务名称。 名称只能包含英文小写字母（a~z）、数字和短横线（-），开头和结尾必须为英文小写字母或数字，且长度为3~16字符。

参数	说明

TaskConfig	投递任务配置，包括如下选项：
	• OssPrefix：OSS Bucket中的目录前缀，将表格存储的数据投递到该OSS Bucket目录中。投递路径中支持引用$yyyy、$MM、$dd、$HH、$mm五种时间变量。
	  ◦ 当投递路径中引用时间变量时，可以按数据的写入时间动态生成OSS目录，实现hive partition naming style的数据时间分区，从而按照时间分区组织OSS中的文件分布。
	  ◦ 当投递路径中不引用时间变量时，所有文件会被投递到固定的OSS前缀目录中。
	• OssBucket：OSS Bucket名称。
	• OssEndpoint：OSS Bucket所在地域的服务地址。
	• OssRoleName：表格存储服务关联角色的ARN信息。
	• Format：投递的数据的存储以Parquet列存格式存储，数据湖投递默认使用PLAIN编码方式，PLAIN编码方式支持任意类型数据。
	• EventTimeColumn：事件时间列，用于指定按某一列数据的时间进行分区。如果不设置此参数，则按数据写入表格存储的时间进行分区。
	• Schema：指定需要投递的数据列，必须手动配置投递字段的源表字段、目标字段和目标字段类型。
	您可以选择任意字段以任意顺序、名称写入列存文件，OSS的列存数据会按Schema数组中的数据列先后顺序分布。
	注意
	投递数据的字段类型必须与数据源的字段类型匹配，否则会作为脏数据丢弃。字段类型映射详情请参见数据格式映射。

TaskType	投递任务的类型，包括如下选项： • IncTask：表示增量数据投递模式，只同步增量数据。 • BaseTask：表示全量数据投递模式，一次性全表扫描数据同步。 • BaseIncTask（默认）：表示全量&增量数据投递模式，全量数据同步完成后，再同步增量数据。 其中增量数据同步时可以获取最新投递时间和了解当前投递状态。

示例

```
func CreateTaskSample(client *tablestore.TableStoreClient) {
 createTask := &tablestore.CreateDeliveryTaskRequest{
  TableName: "sampleTable",
  TaskName: "sampledeliverytask",
  TaskType: tablestore.BaseIncTask,
  TaskConfig: &tablestore.OSSTaskConfig{
   OssPrefix:   "sample/year=$yyyy/month=$MM",
   OssBucket:      "datadeliverytest",
   OssEndpoint:    "oss-cn-hangzhou.aliyuncs.com",
   OssRoleName:    "acs:ram::17************45:role/aliyunserviceroleforotsdatadelivery",
   Schema: []*tablestore.TaskSchema{
    {
     ColumnName: "PK1",
     OssColumnName: "PK1",
     Type: tablestore.ParquetInt64,
    },
    {
     ColumnName: "PK2",
     OssColumnName: "PK2",
     Type: tablestore.ParquetUtf8,
    },
    {
     ColumnName: "Col1",
     OssColumnName: "Col1",
     Type: tablestore.ParquetDouble,
    },


   },
  },
 }
 createResp, err := client.CreateDeliveryTask(createTask)
 if err != nil {
  log.Fatal("create delivery task failed ", err)
 }
 fmt.Println("create delivery task success ", createResp.RequestId)
}
```

4.10.2. 列出投递任务名称

使用ListDeliveryTask接口列出数据表所有的投递任务信息。

前提条件

- 已初始化TableStoreClient。具体操作，请参见初始化。
- 已创建投递任务。使用控制台或SDK的具体操作，请分别参见快速入门或创建投递任务。

请求参数

参数	说明
TableName	数据表名称。

返回参数

参数	说明
TableName	数据表名称，和请求时一致。
TaskName	投递任务名称。
TaskType	投递任务的类型，包括如下选项： • 0：表示全量数据投递模式。 • 1：表示增量数据投递模式。 • 2：表示全量&增量数据投递模式。

示例

```
func ListTask(client *tablestore.TableStoreClient, tableName string) {
 resp, err := client.ListDeliveryTask(&tablestore.ListDeliveryTaskRequest{
  TableName: tableName,
 })
 if err != nil {
  log.Fatal("list delivery task failed ", err)
 }
 for _, task := range resp.Tasks {
  fmt.Println("task: ", task)
 }
 fmt.Println("list task finish")
}
```

4.10.3. 查询投递任务描述信息

使用DescribeDeliveryTask接口查询投递任务描述信息。

前提条件

- 已初始化TableStoreClient。具体操作，请参见初始化。
- 已创建投递任务。使用控制台或SDK的具体操作，请分别参见快速入门或创建投递任务。

请求参数

参数	说明
TableName	数据表名称。
TaskName	投递任务名称。

返回参数

参数	说明
TaskConfig	投递任务的配置信息。
TaskSyncStat	投递任务的同步状态。
TaskType	投递任务的类型，包括如下选项： • 0：表示全量数据投递模式。 • 1：表示增量数据投递模式。 • 2：表示全量&增量数据投递模式。

示例

```
func DescribeTaskSample(client *tablestore.TableStoreClient, tableName, taskName string) {
 req := &tablestore.DescribeDeliveryTaskRequest{
  TableName: tableName,
  TaskName: taskName,
 }
 resp, err := client.DescribeDeliveryTask(req)
 if err != nil {
  log.Fatal("describe delivery task failed ", err)
 }
 fmt.Println("TaskConfig: ", *resp.TaskConfig)
 fmt.Println("TaskSyncStat: ", *resp.TaskSyncStat)
 fmt.Println("TaskType: ", resp.TaskType)
 return
}
```

4.10.4. 删除投递任务

使用DeleteDeliveryTask接口删除一个投递任务。

参数

参数	说明
TableName	数据表名称。
TaskName	投递任务名称。

示例

```
func DeleteTaskSample(client *tablestore.TableStoreClient, tableName, taskName string) {
 delResp, err := client.DeleteDeliveryTask(&tablestore.DeleteDeliveryTaskRequest{
  TableName: tableName,
  TaskName: taskName,
 })
 if err != nil {
  log.Fatal("delete delivery task failed ", err)
 }
 fmt.Println("delete task success", delResp.RequestId)
}
```

4.11. 时序模型

4.11.1. 概述

表格存储的Go SDK提供了多种时序表级别的功能操作。

功能	描述
创建时序表	创建一张时序表。
列出时序表名称	获取当前实例下所有时序表的名称。
查询时序表描述信息	查询时序表描述信息，例如数据生命周期（Time To Live，简称TTL）配置等。
更新时序表	更新时序表的配置信息。
删除时序表	删除一张时序表。
写入时序数据	批量写入时序数据。
查询时序数据	查询符合指定条件的时序数据。
检索时间线	指定多种条件检索时间线。
更新时间线元数据	批量更新时间线元数据的属性。

4.11.2. 创建时序表

使用CreateTimeseriesTable创建时序表时，需要指定表的配置信息。

前提条件

- 已通过控制台创建实例。具体操作，请参见创建时序模型公测实例。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

注意事项

时序表的名称不能与当前已存在的数据表名称相同。

参数

参数	说明
timeseriesTableName	时序表名。
timeseriesTableOptions	时序表的配置信息，包括如下内容： timeToLive：配置时序表的数据存活时间，单位为秒。如果希望数据永不过期，可以设置为-1。您可以通过UpdateTimeseriesTable接口修改。

示例

```
/**
 * CreateTimeseriesTableSample用于创建一个时序表，时序表名为timeseriesTableName，TTL为timeToLive。
 */
func CreateTimeseriesTableSample(client *tablestore.TimeseriesClient, timeseriesTableName string , timeToLive int64) {
    fmt.Println("[Info]: Begin to create timeseries table: " , timeseriesTableName)
    timeseriesTableOptions := tablestore.NewTimeseriesTableOptions(timeToLive)        // 构造时序表配置信息。
    // 构造表元数据信息
    timeseriesTableMeta := tablestore.NewTimeseriesTableMeta(timeseriesTableName)     // 设置时序表名。
    timeseriesTableMeta.SetTimeseriesTableOptions(timeseriesTableOptions)             // 设置时序表配置信息
    createTimeseriesTableRequest := tablestore.NewCreateTimeseriesTableRequest()     // 构造创建时序表请求。
    createTimeseriesTableRequest.SetTimeseriesTableMeta(timeseriesTableMeta)
    createTimeseriesTableResponse , err := client.CreateTimeseriesTable(createTimeseriesTableRequest)     // 调用client创建时序表。
    if err != nil {
        fmt.Println("[Error]: Failed to create timeseries table with error: " , err)
        return
    }
    fmt.Println("[Info]: CreateTimeseriesTable finished! RequestId: " , createTimeseriesTableResponse.RequestId)
}
```

4.11.3. 列出时序表名称

使用ListTimeseriesTable接口，您可以获取当前实例下所有时序表的名称以及元数据信息。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

示例

```
/**
 * ListTimeseriesTableSample用于列出实例中所有时序表的表名以及元数据信息。
 */
func ListTimeseriesTableSample(client *tablestore.TimeseriesClient) {
    fmt.Println("[Info]: Begin to list timeseries table !")
    listTimeseriesTableResponse , err := client.ListTimeseriesTable()
    if err != nil {
        fmt.Println("[Info]: List timeseries table failed with error: " , err)
    }
    fmt.Println("[Info]: Timeseries table Meta: ")
    for i := 0; i < len(listTimeseriesTableResponse.GetTimeseriesTableMeta()); i++ {
        curTimeseriesTableMeta := listTimeseriesTableResponse.GetTimeseriesTableMeta()[i]
        fmt.Println("[Info]: Timeseries table name: " , curTimeseriesTableMeta.GetTimeserie
sTableName() , "TTL: " , curTimeseriesTableMeta.GetTimeseriesTableOPtions().GetTimeToLive()
)
    }
    fmt.Println("[Info]: ListTimeseriesTableSample finished !")
}
```

4.11.4. 查询时序表描述信息

使用DescribeTimeseriesTable接口，您可以查询时序表描述信息，例如数据生命周期（Time To Live，简称TTL）配置等。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

参数	说明
timeseriesTableName	时序表名。

示例

```
/**
 * DescribeTimeseriesTableSample用于获取时序表timeseriesTableName的描述信息。
 */
func DescribeTimeseriesTableSample(client *tablestore.TimeseriesClient , timeseriesTableNam
e string) {
    fmt.Println("[Info]: Begin to require timeseries table description!")
    describeTimeseriesTableRequest := tablestore.NewDescribeTimeseriesTableRequset(timeseri
esTableName)  // 构造请求，并设置时序表名。
    describeTimeseriesTableResponse , err := client.DescribeTimeseriesTable(describeTimeser
iesTableRequest)
    if err != nil {
        fmt.Println("[Error]: Failed to require timeseries table description!")
        return
    }
    fmt.Println("[Info]: DescribeTimeseriesTableSample finished. Timeseries table meta: ")
    fmt.Println("[Info]: TimeseriesTableName: " , describeTimeseriesTableResponse.GetTimese
riesTableMeta().GetTimeseriesTableName())
    fmt.Println("[Info]: TimeseriesTable TTL: " , describeTimeseriesTableResponse.GetTimese
riesTableMeta().GetTimeseriesTableOPtions().GetTimeToLive())
}
```

4.11.5. 更新时序表

使用UpdateTimeseriesTable，您可以更新时序表的配置信息，例如数据生命周期（Time To Live，简称TTL）配置。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

更多信息，请参见创建时序表。

示例

```
/**
 * UpdateTimeseriesTableSample用于更新时序表的TTL配置。
 */
func UpdateTimeseriesTableSample(client *tablestore.TimeseriesClient , timeseriesTableName
string) {
    fmt.Println("[Info]: Begin to update timeseries table !")
    // 构造时序表TTL参数配置。
    timeseriesTableOptions := tablestore.NewTimeseriesTableOptions(964000)
    // 构造更新请求。
    updateTimeseriesTableRequest := tablestore.NewUpdateTimeseriesTableRequest(timeseriesTa
bleName)
    updateTimeseriesTableRequest.SetTimeseriesTableOptions(timeseriesTableOptions)
    // 调用时序客户端更新时序表。
    updateTimeseriesTableResponse , err := client.UpdateTimeseriesTable(updateTimeseriesTab
leRequest)
    if err != nil {
        fmt.Println("[Error]: Update timeseries table failed with error: " , err)
        return
    }
    DescribeTimeseriesTableSample(client , timeseriesTableName)
    fmt.Println("[Info]: UpdateTimeseriesTableSample finished! RequestId: " , updateTimeser
iesTableResponse.RequestId)
}
```

4.11.6. 删除时序表

使用DeleteTimeseries接口，您可以删除一张时序表。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

参数	说明
timeseriesTableName	时序表名。

示例

```
/**
 * DeleteTimeseriesTableSample用于删除实例中表名为timeseriesTableName的时序表。
 */
func DeleteTimeseriesTableSample(client *tablestore.TimeseriesClient , timeseriesTableName
string) {
    fmt.Println("[Info]: Begin to delete timeseries table !")
    // 构造删除时序表请求。
    deleteTimeseriesTableRequest := tablestore.NewDeleteTimeseriesTableRequest(timeseriesTa
bleName)
    // 调用时序客户端删除时序表。
    deleteTimeseriesTableResponse , err := client.DeleteTimeseriesTable(deleteTimeseriesTab
leRequest)
    if err != nil {
        fmt.Println("[Error]: Delete timeseries table failed with error: " , err)
        return
    }
    fmt.Println("[Info]: DeleteTimeseriesTableSample finished ! RequestId: " , deleteTimese
riesTableResponse.RequestId)
}
```

4.11.7. 写入时序数据

使用PutTimeseriesData接口，您可以批量写入时序数据。一次PutTimeseriesData调用支持写入多行数据。

前提条件

- 已创建时序表。具体操作，请参见创建时序表。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

一行时序数据（timeseriesRow）包括时间线标识（timeseriesKey）和时间线数据的配置，其中时间线数据包括数据点的时间（timeInUs）和数据点（fields）。详细参数说明请参见下表。

参数	说明
timeseriesKey	时间线标识，包括如下内容： • measurementName：时间线的度量名称。 • dataSource：数据源信息，可以为空。 • tags：时间线的标签信息，为多个字符串的key-value对。
timeInUs	数据点的时间，单位为微秒。
fields	数据点，可以由多个名称（FieldKey）和数据值（FieldValue）对组成。

示例

```
/**
 * PutTimeseriesDataSample用于向时序表中写入一个或多个时序数据。
 */
func PutTimeseriesDataSample(client *tablestore.TimeseriesClient , timeseriesTableName stri
```

```
ng) {
    fmt.Println("[Info]: Begin to PutTimeseriesDataSample !")
    // 构造时序数据行timeseriesRow。
    timeseriesKey := tablestore.NewTimeseriesKey()
    timeseriesKey.SetMeasurementName("CPU")
    timeseriesKey.SetDataSource("127.0.0.1")
    timeseriesKey.AddTag("City" , "Hangzhou")
    timeseriesKey.AddTag("Region" , "Xihu")
    timeseriesRow := tablestore.NewTimeseriesRow(timeseriesKey)
    timeseriesRow.SetTimeInus(time.Now().UnixNano() / 1000)
    timeseriesRow.AddField("temperature" , tablestore.NewColumnValue(tablestore.ColumnType_
INTEGER , 98))
    timeseriesRow.AddField("status" , tablestore.NewColumnValue(tablestore.ColumnType_STRIN
G , "ok"))
    // 构造时序数据行timeseriesRow1。
    timeseriesKey1 := tablestore.NewTimeseriesKey()
    timeseriesKey1.SetMeasurementName("NETWORK")
    timeseriesKey1.SetDataSource("127.0.0.1")
    timeseriesKey1.AddTag("City" , "Hangzhou")
    timeseriesKey1.AddTag("Region" , "Xihu")
    timeseriesRow1 := tablestore.NewTimeseriesRow(timeseriesKey1)
    timeseriesRow1.SetTimeInus(time.Now().UnixNano() / 1000)
    timeseriesRow1.AddField("in" , tablestore.NewColumnValue(tablestore.ColumnType_INTEGER
, 1000))
    timeseriesRow1.AddField("data" , tablestore.NewColumnValue(tablestore.ColumnType_BINARY
, []byte("tablestore")))
    timeseriesRow1.AddField("program" , tablestore.NewColumnValue(tablestore.ColumnType_STR
ING , "tablestore.d"))
    timeseriesRow1.AddField("status" , tablestore.NewColumnValue(tablestore.ColumnType_BOOL
EAN, true))
    timeseriesRow1.AddField("lossrate" , tablestore.NewColumnValue(tablestore.ColumnType_DO
UBLE , float64(1.9098)))
    // 构造写入时序数据的请求。
    putTimeseriesDataRequest := tablestore.NewPutTimeseriesDataRequest(timeseriesTableName)
    putTimeseriesDataRequest.AddTimeseriesRows(timeseriesRow , timeseriesRow1)
    // 调用时序客户端写入时序数据。
    putTimeseriesDataResponse , err := client.PutTimeseriesData(putTimeseriesDataRequest)
    if err != nil {
        fmt.Println("[Error]: Put timeseries data Failed with error: " , err)
        return
    }
    if len(putTimeseriesDataResponse.GetFailedRowResults()) > 0 {
        fmt.Println("[Warning]: Put timeseries data finished ! Some of timeseries row put F
ailed: ")
        for i := 0; i < len(putTimeseriesDataResponse.GetFailedRowResults()); i++ {
            FailedRow := putTimeseriesDataResponse.GetFailedRowResults()[i]
            fmt.Println("[Warning]: Failed Row: Index: " , FailedRow.Index , " Error: " , F
ailedRow.Error)
        }
    } else {
        fmt.Println("[Info]: PutTimeseriesDataSample finished! RequestId: " , putTimeseries
DataResponse.RequestId)
    }
}
```

4.11.8. 查询时序数据

使用GetTimeseriesData接口，您可以查询符合指定条件的时序数据。

前提条件

- 已写入时序数据。具体操作，请参见写入时序数据。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

参数	说明
timeseriesKey	要查询的时间线，包括如下内容： •
timeRange	要查询的时间范围，包括如下内容： • beginTimeInUs：起始时间。 • endTimeInUs：结束时间。
backward	是否按照时间倒序读取数据，可用于获取某条时间线的最新数据。取值范围如下： • true：按照时间倒序读取。 • false（默认）：按照时间正序读取。
fieldsToGet	要获取的数据列列名。如果不指定，则默认获取所有列。 注意 fieldsToGet中需要指定要获取的每一列的列名和类型。如果类型不匹配，则读取不到对应列的数据。
limit	本次最多返回的行数。 说明 limit仅限制最多返回的行数，在满足条件行数大于limit时，也可能由于扫描数据量等限制导致返回行数少于limit条，此时可以通过nextToken继续获取后面的行。
nextToken	如果一次查询仅返回了部分符合条件的行，此时response中会包括nextToken，可在下一次请求中指定nextToken用来继续读取数据。

示例

```
/**
 * GetTimeseriesDataSample用于根据timeseriesKey获取时序表中指定的时间线数据。
 */
func GetTimeseriesDataSample(client *tablestore.TimeseriesClient , timeseriesTableName stri
ng) {
    fmt.Println("[Info]: Begin to get timeseries data !")
    // 构造待查询时间线的timeseriesKey。
    timeseriesKey := tablestore.NewTimeseriesKey()
    timeseriesKey.SetMeasurementName("NETWORK")
    timeseriesKey.SetDataSource("127.0.0.1")
    timeseriesKey.AddTag("City" , "Hangzhou")
    timeseriesKey.AddTag("Region" , "Xihu")
    // 构造查询请求。
    getTimeseriesDataRequest := tablestore.NewGetTimeseriesDataRequest(timeseriesTableName)
    getTimeseriesDataRequest.SetTimeseriesKey(timeseriesKey)
    getTimeseriesDataRequest.SetTimeRange(0 , time.Now().UnixNano() / 1000)    // 指定查询时
间线的范围。
    getTimeseriesDataRequest.SetLimit(-1)
    // 调用时序客户端接口获取时间线数据。
    getTimeseriesResp , err := client.GetTimeseriesData(getTimeseriesDataRequest)
    if err != nil {
        fmt.Println("[Error]: Get timeseries data Failed with error: " , err)
        return
    }
    fmt.Println("[Info]: Get timeseries data succeed ! TimeseriesRows: ")
    for i := 0; i < len(getTimeseriesResp.GetRows()); i++ {
        fmt.Println("[Info]: Row" , i , ": [" , getTimeseriesResp.GetRows()[i].GetTimeserie
sKey().GetMeasurementName(),
            getTimeseriesResp.GetRows()[i].GetTimeseriesKey().GetDataSource(),
            getTimeseriesResp.GetRows()[i].GetTimeseriesKey().GetTagsSlice(), "]",
            getTimeseriesResp.GetRows()[i].GetFieldsSlice(),
            getTimeseriesResp.GetRows()[i].GetTimeInus())
    }
    fmt.Println("[Info]: GetTimeseriesDataSample finished! RequestId: " , getTimeseriesResp
.RequestId)
}
```

4.11.9. 检索时间线

使用QueryTimeseriesMeta接口，您可以指定多种条件检索时间线。

前提条件

- 已写入时序数据。具体操作，请参见写入时序数据。
- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

metaQueryCondition表示检索时间线的条件，包括compositeMetaQueryCondition（组合条件）、measurementMetaQueryCondition（度量名称条件）、dataSourceMetaQueryCondition（数据源条件）、tagMetaQueryCondition（标签条件）、attributeMetaQueryCondition（属性条件）和updateTimeMetaQueryCondition（更新时间条件）。详细参数说明请参见下表。

参数	说明
compositeMetaQueryCondition	组合条件，包括如下内容： • operator：逻辑运算符，可选AND、OR、NOT。 • subConditions：子条件列表，通过operator组成复杂查询条件。
measurementMetaQueryCondition	度量名称条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • value：要匹配的度量名称值，类型为字符串。
dataSourceMetaQueryCondition	数据源条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • value：要匹配的数据源值，类型为字符串。
tagMetaQueryCondition	标签条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • value：要匹配的标签值，类型为字符串。
attributeMetaQueryCondition	时间线元数据的属性条件，包括如下内容： • operator：关系运算符或前缀匹配条件，关系运算符可选=、!=、>、>=、<、<=。 • attributeName：属性名称，类型为字符串。 • value：属性值，类型为字符串。
updateTimeMetaQueryCondition	时间线元数据的更新时间条件，包括如下内容： • operator：关系运算符，可选=、!=、>、>=、<、<=。 • timeInUs：时间线元数据更新时间的时间戳，单位为微秒。

示例

```
/**
 * QueryTimeseriesMetaSample用于根据指定条件查询数据表中特定时间线的measurement、source和tag信息，其中查询条件可为组合条件。
 */
func QueryTimeseriesMetaSample(client *tablestore.TimeseriesClient , timeseriesTableName string) {
    fmt.Println("[Info]: Begin to query timeseries table meta!")
    // 构造多个查询条件。
    measurementMetaQueryCondition := tablestore.NewMeasurementQueryCondition(tablestore.OP_GREATER_EQUAL , "")
    datasourceMetaQueryCondition := tablestore.NewDataSourceMetaQueryCondition(tablestore.OP_GREATER_EQUAL , "")
    tagMetaQueryCondition := tablestore.NewTagMetaQueryCondition(tablestore.OP_GREATER_THAN , "City" , "")
    // 构造组合条件。
    compsiteMetaQueryCondition := tablestore.NewCompositeMetaQueryCondition(tablestore.OP_AND)
    compsiteMetaQueryCondition.AddSubConditions(measurementMetaQueryCondition)
    compsiteMetaQueryCondition.AddSubConditions(datasourceMetaQueryCondition)
    compsiteMetaQueryCondition.AddSubConditions(tagMetaQueryCondition)
    // 构造查询请求。
    queryTimeseriesMetaRequest := tablestore.NewQueryTimeseriesMetaRequest(timeseriesTableName)
    queryTimeseriesMetaRequest.SetCondition(compsiteMetaQueryCondition)
    queryTimeseriesMetaRequest.SetLimit(-1)
    // 调用时序客户端执行查询请求。
    queryTimeseriesTableResponse , err := client.QueryTimeseriesMeta(queryTimeseriesMetaRequest)
    if err != nil {
        fmt.Println("[Error]: Query timeseries table meta failed with error: " , err)
        return
    }
    fmt.Println("[Info]: Query timeseries table meta succeed: ")
    for i := 0; i < len(queryTimeseriesTableResponse.GetTimeseriesMetas()); i++ {
        curTimeseriesMeta := queryTimeseriesTableResponse.GetTimeseriesMetas()[i]
        fmt.Println("[Info]: Meta_" , i , ": " , "Measurement: " , curTimeseriesMeta.GetTimeseriesKey().GetMeasurementName() ,
            "Source: " , curTimeseriesMeta.GetTimeseriesKey().GetDataSource() ,
            "Tags: " , curTimeseriesMeta.GetTimeseriesKey().GetTagsSlice() ,
            "Attrs: " , curTimeseriesMeta.GetAttributeSlice())
    }
    fmt.Println("[Info]: QueryTimeseriesMetaSample finished !")
}
```

4.11.10. 更新时间线元数据

使用UpdateTimeseriesMeta接口，您可以批量更新时间线元数据的属性。一次UpdateTimeseriesMeta调用支持更新多个时间线的元数据。

前提条件

- 已写入时序数据。具体操作，请参见写入时序数据。

- 已初始化TimeseriesClient。具体操作，请参见初始化。

参数

timeseriesMeta表示一个时间线元数据，每个timeseriesMeta包括timeseriesKey和attributes。详细参数说明请参见下表，

参数	描述
timeseriesKey	时间线标识。
attributes	时间线的属性信息，内容为字符串类型的key-value对。

示例

```
/**
 * UpdateTimeseriesMetaSample用于更新时间线中的Attributes信息。
 */
func UpdateTimeseriesMetaSample(tsClient *tablestore.TimeseriesClient, timeseriesTableName
string) {
    fmt.Println("[Info]: Begin to update timeseries meta!")
    PutTimeseriesDataSample(tsClient , timeseriesTableName)
    updateTimeseriesMetaRequest := tablestore.NewUpdateTimeseriesMetaRequest(timeseriesTabl
eName)
    timeseriesKey := tablestore.NewTimeseriesKey()
    timeseriesKey.SetMeasurementName("NETWORK")
    timeseriesKey.SetDataSource("127.0.0.1")
    timeseriesKey.AddTag("City" , "Hangzhou")
    timeseriesKey.AddTag("Region" , "Xihu")
    timeseriesMeta := tablestore.NewTimeseriesMeta(timeseriesKey)
    //timeseriesMeta.SetUpdateTimeInUs(96400)
    timeseriesMeta.AddAttribute("NewRegion" , "Yuhang")
    timeseriesMeta.AddAttribute("NewCity" , "Shanghai")
    updateTimeseriesMetaRequest.AddTimeseriesMetas(timeseriesMeta)
    updateTimeseriesMetaResponse , err := tsClient.UpdateTimeseriesMeta(updateTimeseriesMet
aRequest)
    if err != nil {
        fmt.Println("[Error]: Update timeseries meta failed with error: " , err)
        return
    }
    if len(updateTimeseriesMetaResponse.GetFailedRowResults()) > 0 {
        fmt.Println("[Error]: Update timeseries meta failed row: ")
        for i := 0; i < len(updateTimeseriesMetaResponse.GetFailedRowResults()); i++ {
            fmt.Println("[Error]: " , updateTimeseriesMetaResponse.GetFailedRowResults()[i]
.Index , updateTimeseriesMetaResponse.GetFailedRowResults()[i].Error)
        }
    }
    QueryTimeseriesMetaSample(tsClient , timeseriesTableName)
    fmt.Println("[Info]: UpdateTimeseriesMetaSample finished!")
}
```

5.Python SDK

5.1. 前言

本文介绍表格存储Python SDK的安装和使用。

前提条件

- 已开通表格存储。具体操作，请参见开通表格存储服务。
- 已创建AccessKey。具体操作，请参见获取AccessKey。

SDK下载

- 通过SDK包下载，下载地址请参见SDK包。
- 通过GitHub下载，下载地址请参见GitHub。

关于SDK版本迭代的更多信息，请参见Python SDK 历史迭代版本。

兼容性

- 对5.x.x系列的SDK兼容。5.2.1和5.1.0在如下情况不兼容：
  - Search接口返回结果的类型。

    5.1.0及以前版本的返回结果默认为Tuple类型。从5.2.0开始默认返回结果为SearchResponse对象，SearchResponse已实现__iter__方法，支持遍历；如果需要返回Tuple类型的结果，请使用SearchResponse.v1_response()方法实现。
  - 新增ParallelScan接口。

    默认返回结果为ParallelScanResponse对象。如果需要返回Tuple类型的结果，请使用ParallelScanResponse.v1_response()方法实现。
- 对4.x.x系列的SDK兼容。
- 对2.x.x系列的SDK不兼容，原因是2.0系列版本中支持主键乱序，而4.0.0版本开始不允许主键乱序，涉及的不兼容点包括：
  - 包名称由ots2变更为tablestore。
  - Client.create_table接口新增TableOptions参数。
  - put_row、get_row、update_row等接口的primary_key参数由dict类型变更为list类型，目的是保证主键的顺序性。
  - put_row、update_row等接口的attribute_columns参数由dict类型变更为list类型。
  - put_row、update_row等接口的attribute_columns参数新增timestamp。
  - get_row、get_range等接口新增max_version、time_range参数，这两个参数必须存在一个。
  - put_row、update_row、delete_row等接口新增return_type参数，目前仅支持RT_PK，表示返回值中包含当前行PK值。
  - put_row、update_row、delete_row等接口返回值新增return_row，如果在请求中指定了return_type为RT_PK，则return_row中包含此行的PK值。

版本

当前最新版本为5.2.1。

5.2. 安装

本文介绍如何安装表格存储Python SDK。

环境准备

安装表格存储Python SDK需使用Python 2或Python 3。

安装

- 方式一：通过pip安装。

  安装命令如下。

  ```
  sudo pip install tablestore
  ```

- 方式二：通过GitHub安装。

  如果没有安装Git，请安装Git后，执行如下命令。

  ```
  git clone https://github.com/aliyun/aliyun-tablestore-python-sdk.git
  sudo python setup.py  install
  ```

- 方式三：通过源码安装。

  i. 下载SDK包。

  ii. 解压SDK包后执行如下命令。

  ```
  sudo python setup.py install
  ```

验证SDK

通过命令行输入python并按回车键，在Python环境下检查SDK的版本。

```
>>> import tablestore
>>> tablestore.__version__
'5.2.1'
```

卸载SDK

直接通过pip卸载。

```
sudo pip uninstall tablestore
```

5.3. 初始化

OTSClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、读写单行数据、读写多行数据等。

确定Endpoint

Endpoint是阿里云表格存储服务各个实例的域名地址，目前支持下列形式。

示例	解释
http://sun.cn-hangzhou.ots.aliyuncs.com	HTTP协议，公网网络访问杭州区域的sun实例。
https://sun.cn-hangzhou.ots.aliyuncs.com	HTTPS协议，公网网络访问杭州区域的sun实例。

> **注意** 除了公网可以访问外，也支持私网地址。更多请参见服务地址。

请按照如下步骤获取实例的Endpoint：

1. 登录表格存储管理控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

配置密钥

要接入阿里云的表格存储服务，您需要拥有一个有效的访问密钥进行签名认证。目前支持下面三种方式：

- 阿里云账号的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 在阿里云官网注册阿里云账号。
  ii. 创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。
- 被授予访问表格存储权限的RAM用户的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 使用阿里云账号前往访问控制RAM，创建一个新的RAM用户或者使用已经存在的RAM用户。
  ii. 使用阿里云账号授予RAM用户访问表格存储的权限。
  iii. RAM用户被授权后，即可使用自己的AccessKey ID和AccessKey Secret访问。
- 从STS获取的临时访问凭证。获取步骤如下：
  i. 应用的服务器通过访问RAM/STS服务，获取一个临时的AccessKey ID、AccessKey Secret和SecurityToken发送给使用方。
  ii. 使用方使用上述临时密钥访问表格存储服务。

初始化对接

- 接口

```
        """
        初始化``OTSClient``实例。
        ``end_point``是表格存储服务的地址（例如'https://instance.cn-hangzhou.ots.aliyun.com:
80'），必须以'https://'开头。
        ``access_key_id``是访问表格存储服务的AccessKeyID，通过官方网站申请或通过管理员获取。
        ``access_key_secret``是访问表格存储服务的AccessKeySecret，通过官方网站申请或通过管理员获
取。
        ``instance_name``是要访问的实例名，通过官方网站控制台创建或通过管理员获取。
        ``sts_token``是访问表格存储服务的STS token，从阿里云STS服务获取，具有有效期，过期后需要重
新获取。
        ``encoding``请求参数的字符串编码类型，默认值为utf8。
        ``socket_timeout``是连接池中每个连接的Socket超时，单位为秒，可以为int或float。默认值为50
。
        ``max_connection``是连接池的最大连接数。默认值为50。
        ``logger_name``用来在请求中打印DEBUG日志，或者在出错时打印ERROR日志。
        ``retry_policy``定义了重试策略，默认的重试策略为DefaultRetryPolicy。你您可以继承RetryPo
licy来实现自己的重试策略，详情请参见DefaultRetryPolicy的代码。
        """
        class OTSClient(object):
            def __init__(self, endpoint, access_key_id, access_key_secret, instance_name,
**kwargs):
```

- 示例

```
    #########    设置日志文件名称和重试策略     #########
    # 日志文件名称为table_store.log，重试策略是WriteRetryPolicy，会对写重试。
    ots_client = OTSClient('endpoint', 'access_key_id', 'access_key_secret', 'instance_na
me', logger_name = 'table_store.log',  retry_policy = WriteRetryPolicy())
    #########    使用STS     #########
    ots_client = OTSClient('endpoint', 'STS.K8h*******GB77', 'CkuDj******Wn6', 'instance_
name', sts_token = 'CAISjgJ1q6Ft5B2y********OFcsLLuw==')
```

HTTPS

- 从2.0.8版本开始支持HTTPS。
- OpenSSL版本最少为0.9.8j，推荐OpenSSL 1.0.2d。
- Python 2.0.8发布包中包含了certifi包直接安装使用。如果需要更新根证书请从根证书下载最新的根证书。

5.4. 表

5.4.1. 概述

表格存储的 Python SDK 提供了多种表级别的操作接口：

创建表

列出表名称

更新表

查询表描述

5.4.2. 创建数据表

使用CreateTable接口创建数据表时，需要指定数据表的结构信息和配置信息，高性能实例中的数据表还可以根据需要设置预留读/写吞吐量。创建数据表的同时支持创建一个或者多个索引表。

> **说明**
> - 创建数据表后需要几秒钟进行加载，在此期间对该数据表的读/写数据操作均会失败。请等待数据表加载完毕后再进行数据操作。
> - 创建数据表时必须指定数据表的主键。主键包含1个~4个主键列，每一个主键列都有名称和类型。

前提条件

- 已通过控制台创建实例。具体操作，请参见创建实例。
- 已初始化Client。具体操作，请参见初始化。

接口

```
"""
说明：根据指定表结构信息创建数据表。
``table_meta``是``tablestore.metadata.TableMeta``类的实例，它包含数据表名称和PrimaryKey的schema。
请参见``TableMeta``类的文档。当创建一个数据表后，通常需要等待几秒钟时间使partition load完成，才能进行各种操作。
``table_options``是``tablestore.metadata.TableOptions``类的实例，它包含time_to_live，max_version和max_time_deviation三个参数。
``reserved_throughput``是``tablestore.metadata.ReservedThroughput``类的实例，表示预留读写吞吐量。
``secondary_indexes``是一个数组，可以包含一个或多个``tablestore.metadata.SecondaryIndexMeta``类的实例，表示要创建的全局二级索引。
返回：无。
"""
def create_table(self, table_meta, table_options, reserved_throughput, secondary_indexes=[]):
```

参数

参数	说明

参数	说明
table_meta	数据表的结构信息，包括如下内容： • table_name：数据表名称。 • schema_of_primary_key：数据表的主键定义。更多信息，请参见主键和属性。 **说明** 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。 ◦ 表格存储可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体请参见表格存储数据模型和查询操作。 ◦ 第一列主键作为分区键。分区键相同的数据会存放在同一个分区内，所以相同分区键下最好不要超过10 GB以上数据，否则会导致单分区过大，无法分裂。另外，数据的读/写访问最好在不同的分区键上均匀分布，有利于负载均衡。 • defined_columns：预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。

参数	说明
table_options	数据表的配置信息。更多信息，请参见数据版本和生命周期。 配置信息包括如下内容： • time_to_live：数据生命周期，即数据的过期时间。当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 创建数据表时，如果希望数据永不过期，可以设置数据生命周期为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 单位为秒。 **说明** 如果需要使用索引，则数据生命周期必须设置为-1（数据永不过期）。 • max_versions：最大版本数，即属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。 **说明** 如果需要使用索引，则最大版本数必须设置为1。 • max_time_deviation：有效版本偏差，即写入数据的时间戳与系统当前时间的偏差允许最大值。只有当写入数据所有列的版本号与写入时时间的差值在数据有效版本偏差范围内，数据才能成功写入。 属性列的有效版本范围为**[数据写入时间-有效版本偏差，数据写入时间+有效版本偏差)**。 创建数据表时，如果未设置有效版本偏差，系统会使用默认值86400；创建数据表后，可以通过UpdateTable接口动态修改有效版本偏差。 单位为秒。
reserved_throughput	为数据表配置预留读吞吐量或预留写吞吐量。 容量型实例中的数据表的预留读/写吞吐量只能设置为0，不允许预留。 默认值为0，即完全按量计费。 单位为CU。 • 当预留读吞吐量或预留写吞吐量大于0时，表格存储会根据配置为数据表预留相应资源，且数据表创建成功后，将会立即按照预留吞吐量开始计费，超出预留的部分进行按量计费。更多信息，请参见计费概述。 • 当预留读吞吐量或预留写吞吐量设置为0时，表格存储不会为数据表预留相应资源。

参数	说明
secondary_indexes	索引表的结构信息，每个SecondaryIndexMeta包括如下内容： • index_name：索引表名称。 • primary_key_names：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 • defined_column_names：索引表的属性列，索引表属性列为数据表的预定义列的组合。 • index_type：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ◦ 当不设置index_type或者设置index_type为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ◦ 当设置index_type为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。

示例

- 创建数据表（不带索引）

  创建一个有2个主键列，数据保留1年（60*60*24*365=31536000秒），最大版本数3，写入时间戳偏移小于1天（86400秒），预留读写吞吐量为(0, 0)的数据表。

```
# 创建主键列的schema，包括主键的个数、名称和类型。
# 第一个PK列为整型，名称是pk0，该列同时也是分区键。
# 第二个PK列为整型，名称是pk1。其他可选的类型包括STRING和BINARY，此处使用INTEGER。
schema_of_primary_key = [('pk0', 'INTEGER'), ('pk1', 'INTEGER')]
# 通过数据表名称和主键列的schema创建一个tableMeta。
table_meta = TableMeta('SampleTable', schema_of_primary_key)
# 创建TableOptions，数据保留31536000秒，超过后自动删除；最大3个版本；写入时指定的版本值和当前时间相差不能超过86400秒（即1天）。
table_options = TableOptions(31536000, 3, 86400)
# 设置预留读吞吐量为0，预留写吞吐量为0。
reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
# 调用client的create_table接口，如果没有抛出异常，则说明执行成功。
try:
    ots_client.create_table(table_meta, table_options, reserved_throughput)
    print "create table succeeded"
# 如果抛出异常，则说明执行失败，处理异常。
except Exception:
    print "create table failed."
```

  详细代码请参见CreateTable@GitHub。

- 创建数据表（带索引且索引类型为全局二级索引）

```
schema_of_primary_key = [('gid', 'INTEGER'), ('uid', 'STRING')]
defined_columns = [('i', 'INTEGER'), ('bool', 'BOOLEAN'), ('d', 'DOUBLE'), ('s', 'STRING'
), ('b', 'BINARY')]
table_meta = TableMeta(table_name, schema_of_primary_key, defined_columns)
table_option = TableOptions(-1, 1)
reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
secondary_indexes = [
    SecondaryIndexMeta('index1', ['i', 's'], ['bool', 'b', 'd']),
    ]
client.create_table(table_meta, table_option, reserved_throughput, secondary_indexes)
```

- 创建数据表（带索引且索引类型为本地二级索引）

```
schema_of_primary_key = [('gid', 'INTEGER'), ('uid', 'STRING')]
defined_columns = [('i', 'INTEGER'), ('bool', 'BOOLEAN'), ('d', 'DOUBLE'), ('s', 'STRING'
), ('b', 'BINARY')]
table_meta = TableMeta(table_name, schema_of_primary_key, defined_columns)
table_option = TableOptions(-1, 1)
reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
secondary_indexes = [
    SecondaryIndexMeta('index1', ['gid', 's'], ['bool', 'b', 'd'],index_type= SecondaryIn
dexType.LOCAL_INDEX),
    ]
client.create_table(table_meta, table_option, reserved_throughput, secondary_indexes)
```

5.4.3. 更新表

使用更新表（UpdateTable）接口更新指定表的预留读吞吐量、预留写吞吐量、最大版本数等设置。

接口

```
    """
        说明：更新表属性，目前只支持修改预留读写吞吐量。
        ``table_name``是对应的表名。
        ``table_options``是``tablestore.metadata.TableOptions``类的示例，它包含time_to_live，m
ax_version和max_time_deviation三个参数。
        ``reserved_throughput``是``ots2.metadata.ReservedThroughput``类的实例，表示预留读写吞吐
量。
        返回：针对该表的预留读写吞吐量的最近上调时间、最近下调时间和当天下调次数。
        ``update_table_response``表示更新的结果，是ots2.metadata.UpdateTableResponse类的实例。
        """
        def update_table(self, table_name, table_options, reserved_throughput):
```

示例

更新表的最大版本数为5。

```
        # 设定新的预留读吞吐量为0，写吞吐量为0。
        reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
        # 创建TableOptions，数据保留31536000秒，超过后自动删除；最大5个版本；写入时指定的版本值和当前标准时间相差不能超过1天。
        table_options = TableOptions(31536000, 5, 86400)
        try:
            # 调用接口更新表的预留读写吞吐量。
            ots_client.update_table('SampleTable', reserved_throughput)
            # 如果没有抛出异常，则说明执行成功。
            print "update table succeeded"
        except Exception:
            # 如果抛出异常，则说明执行失败，处理异常。
            print "update table failed"
```

详细代码请参见UpdateTable@GitHub。

5.4.4. 列出表名称

使用ListTable接口获取当前实例下已创建的所有表的表名。

> **说明** API说明请参见ListTable。

接口

```
"""
说明：获取所有表名的列表。
返回：表名列表。
``table_list``表示获取的表名列表，类型为tuple，例如('MyTable1', 'MyTable2')。
"""
def list_table(self):
```

示例

获取实例下所有表的表名。

```
try:
    list_response = ots_client.list_table()
    print 'table list: '
    for table_name in list_response:
        print table_name
    print "list table succeeded"
except Exception:
    print "list table failed."
```

详细代码请参见ListTable@GitHub。

5.4.5. 查询表描述信息

使用DescribeTable接口可以查询指定表的结构、预留读/写吞吐量详情等信息。

> **说明** API说明请参见DescribeTable。

接口

```
def describe_table(self, table_name):
```

返回为表的描述信息。返回结果中describe_table_response表示表的描述信息，是ots2.metadata.DescribeTableResponse类的实例。

参数

参数	说明
table_name	表名。

示例

获取表的描述信息。

```
try:
    describe_response = ots_client.describe_table('myTable')
    # 如果没有抛出异常，则说明执行成功，打印如下表信息。
    print "describe table succeeded."
    print ('TableName: %s' % describe_response.table_meta.table_name)
    print ('PrimaryKey: %s' % describe_response.table_meta.schema_of_primary_key)
    print ('Reserved read throughput: %s' % describe_response.reserved_throughput_details.c
apacity_unit.read)
    print ('Reserved write throughput: %s' % describe_response.reserved_throughput_details.
capacity_unit.write)
    print ('Last increase throughput time: %s' % describe_response.reserved_throughput_deta
ils.last_increase_time)
    print ('Last decrease throughput time: %s' % describe_response.reserved_throughput_deta
ils.last_decrease_time)
    print ('table options\'s time to live: %s' % describe_response.table_options.time_to_li
ve)
    print ('table options\'s max version: %s' % describe_response.table_options.max_version
)
    print ('table options\'s max_time_deviation: %s' % describe_response.table_options.max_
time_deviation)
except Exception:
    # 如果抛出异常，则说明执行失败，处理异常。
    print "describe table failed."
```

详细代码请参见DescribeTable@GitHub。

5.4.6. 删除数据表

使用DeleteTable接口删除当前实例下指定数据表。

> **说明** API说明请参见DeleteTable。

前提条件

- 已初始化OTSClient，详情请参见初始化。
- 已创建数据表。
- 已删除数据表上的索引表和多元索引。

接口

```
def delete_table(self, table_name):
```

参数

参数	说明
table_name	数据表名称。

示例

删除指定数据表。

```
try:
    # 调用接口删除表SampleTable。
    ots_client.delete_table('SampleTable')
    # 如果没有抛出异常，则说明执行成功。
    print "delete table succeeded"
    # 如果抛出异常，则说明执行失败，处理异常。
except Exception:
    print "delete table failed"
```

详细代码请参见DeleteTable@GitHub。

5.4.7. 主键列自增

设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。

前提条件

已初始化Client。具体操作，请参见初始化。

使用方法

1. 创建表时，将非分区键的主键列设置为自增列。

   只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号长整型。

2. 写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符。

   如果需要获取写入数据后系统自动生成的自增列的值，将ReturnType设置为RT_PK，可以在数据写入成功后返回自增列的值。

查询数据时，需要完整的主键值。通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为RT_PK可以获取完整的主键值。

示例

主键自增列功能主要涉及创建表（CreateTable）和写数据（PutRow、UpdateRow和BatchWriteRow）两类接口。

1. 创建表

   创建表时，只需将自增的主键属性设置为PK_AUTO_INCR。

```
from tablestore import *
table_name = 'OTSPkAutoIncrSimpleExample'
def create_table(client):
    # 创建表，表中包括两个主键：gid，INTEGER类型；uid，INTEGER类型，为自增列。
    schema_of_primary_key = [('gid', 'INTEGER'), ('uid', 'INTEGER', PK_AUTO_INCR)]
    table_meta = TableMeta(table_name, schema_of_primary_key)
    table_options = TableOptions()
    reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
    client.create_table(table_meta, table_options, reserved_throughput)
    print ('Table has been created.')
```

2. 写数据

   写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符PK_AUTO_INCR。

```
from tablestore import *
table_name = 'OTSPkAutoIncrSimpleExample'
def put_row(client):
    # 写入主键：gid为1，uid为自增列。uid列必须设置，否则报错。
    primary_key = [('gid',1), ('uid', PK_AUTO_INCR)]
    attribute_columns = [('name','John'), ('mobile',13900006666), ('address','China'),
('age',20)]
    row = Row(primary_key, attribute_columns)
    # 写入属性列。
    row.attribute_columns = [('name','John'), ('mobile',13900006666), ('address','China
'), ('age',25)]
    consumed, return_row = client.put_row(table_name, row)
    print ('Write succeed, consume %s write cu.' % consumed.write)
    consumed, return_row = client.put_row(table_name, row, return_type = ReturnType.RT_
PK)
    print ('Write succeed, consume %s write cu.' % consumed.write)
    print ('Primary key:%s' % return_row.primary_key)
```

5.4.8. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过PutRow、UpdateRow、DeleteRow或BatchWriteRow接口更新数据时，可以使用条件更新检查行存在性条件和列条件，只有满足条件时才能更新成功。

条件更新包括行存在性条件和列条件。

- 行存在性条件：包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。

  对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。

- 列条件：包括SingleColumnCondition和CompositeColumnCondition，是基于某一列或者某些列的列值进行条件判断。
  - SingleColumnCondition支持一列（可以是主键列）和一个常量比较。不支持两列或者两个常量相比较。
  - CompositeColumnCondition的内节点为逻辑运算，子条件可以是SingleColumnCondition或CompositeColumnCondition。

条件更新可以实现乐观锁功能，即在更新某行时，先获取某列的值，假设为列A，值为1，然后设置条件**列A = 1**，更新行使**列A = 2**。如果更新失败，表示有其他客户端已成功更新该行。

限制

条件更新的列条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。

参数

参数	说明
RowExistenceExpectation	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别用RowExistenceExpectation_IGNORE、RowExistenceExpectation_EXPECT_EXIST、RowExistenceExpectation_EXPECT_NOT_EXIST表示。 • IGNORE：表示忽略，不做任何存在性检查。 • EXPECT_EXIST：表示期望存在，如果该行存在，则满足条件；如果该行不存在，则不满足条件。 • EXPECT_NOT_EXIST：期望行不存在，如果该行不存在，则满足条件；如果该行存在，则不满足条件。
column_name	列的名称。
column_value	列的对比值。
comparator	对列值进行比较的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=），分别用CT_EQUAL、CT_NOT_EQUAL、CT_GREATER_THAN、CT_GREATER_EQUAL、CT_LESS_THAN、CT_LESS_EQUAL表示。

参数	说明
combinator	对多个条件进行组合的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR，分别用LO_NOT、LO_AND、LO_OR表示。 逻辑运算符不同可以添加的子条件个数不同。 • 当逻辑运算符为NOT时，只能添加一个子条件。 • 当逻辑运算符为AND或OR时，必须至少添加两个子条件。
pass_if_missing	当列在某行中不存在时，条件检查是否通过。类型为bool值，默认值为True，表示如果列在某行中不存在时，则条件检查通过，该行满足更新条件。 当设置pass_if_missing为False时，如果列在某行中不存在时，则条件检查不通过，该行不满足更新条件。
latest_version_only	当列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为True，表示如果列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置latest_version_only为False时，如果列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就条件检查通过，该行满足更新条件。

示例

根据指定主键更新一行，只有同时满足行存在和“age”列值为20条件时，才能更新成功，否则更新失败。

```
    primary_key = [('gid',1), ('uid',"101")]
    update_of_attribute_columns = {
        'PUT' : [('name','David'), ('address','Hongkong')],
        'DELETE' : [('address', None, 1488436949003)],
        'DELETE_ALL' : [('mobile'), ('age')],
        'INCREMENT' : [('counter', -1)]
    }
    row = Row(primary_key, update_of_attribute_columns)
    # 指定Condition必须同时满足如下2个条件，才能更新成功，否则更新失败。
    # （1）指定的行存在。
    # （2）“age”列值为20。
    condition = Condition(RowExistenceExpectation.EXPECT_EXIST, SingleColumnCondition("age"
, 20, ComparatorType.EQUAL)) # 只有当行存在时，更新该行。
    consumed, return_row = client.update_row(table_name, row, condition)
```

5.4.9. 局部事务

使用局部事务功能，创建数据范围在一个分区键值内的局部事务。对局部事务中的数据进行读写操作后，可以根据实际提交或者丢弃局部事务。局部事务通过悲观锁（Pessimistic Lock）实现并发控制。

目前局部事务功能处于邀测中，默认关闭。如果需要使用该功能，请提交工单进行申请或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。

使用局部事务可以指定某个分区键值内的操作是原子的，对分区键值内的数据进行的操作要么全部成功要么全部失败，并且所提供的隔离级别为读已提交。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

1. 使用start_local_transaction在指定的分区键值创建一个局部事务，并获取局部事务ID。
2. 对局部事务范围内的数据进行读写操作。

   支持对局部事务进行操作的接口为GetRow、PutRow、DeleteRow、UpdateRow、BatchWriteRow和GetRange。
3. 使用commit_transaction提交局部事务或者使用abort_transaction丢弃局部事务。

限制

- 每个局部事务从创建开始生命周期最长为60秒。

  如果超过60秒未提交或丢弃局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
- 如果创建局部事务时超时，此请求可能在表格存储服务端已执行成功，此时用户需要等待该局部事务超时后重新创建。
- 未提交的局部事务可能失效，如果出现此情况，需要重试该局部事务内的操作。
- 在局部事务中读写数据有如下限制：
  - 不能使用局部事务ID访问局部事务范围（即创建时使用的分区键值）以外的数据。
  - 同一个局部事务中所有写请求的分区键值必须与创建局部事务时的分区键值相同，读请求则无此限制。
  - 一个局部事务同时只能用于一个请求中，在使用局部事务期间，其它使用此局部事务ID的操作均会失败。
  - 每个局部事务中两次读写操作的最大间隔为60秒。

    如果超过60秒未操作局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
  - 每个局部事务中写入的数据量最大为4 MB，按正常的写请求数据量计算规则累加。
  - 如果在局部事务中写入了未指定版本号的Cell，该Cell的版本号会在写入时（而非提交时）由表格存储服务端自动生成，生成规则与正常写入一个未指定版本号的Cell相同。
  - 如果BatchWriteRow请求中带有局部事务ID，则此请求中所有行只能操作该局部事务ID对应的表。
  - 在使用局部事务期间，对应分区键值的数据相当于被锁上，只有持有局部事务ID在局部事务范围内的写请求才会成功，其它不持有局部事务ID在局部事务范围内的写请求均会失败。在局部事务提交、丢弃或超时后，对应的锁也会被释放。
  - 带有局部事务ID的读写请求失败不会影响局部事务本身的存活情况，您可以按照正常的无局部事务ID的读写请求重试规则进行重试，或者主动丢弃当前局部事务。

参数

参数	说明
table_name	数据表名称。
key	数据表分区键。 创建局部事务时，只需要指定局部事务对应的分区键值。

参数	说明
primary_key	数据表主键。 创建局部事务后，对局部事务范围内的数据进行读写操作时，需要指定完整主键。
transaction_id	局部事务ID，用于唯一标识一个局部事务。 创建局部事务后，操作局部事务时均需要带上局部事务ID。

示例

1. 调用start_local_transaction方法使用指定分区键值创建一个局部事务，并获取局部事务ID。

```
# 在PK0下创建局部事务。
key = [('PK0', 1)]
# start_local_transaction方法的返回值即为transaction_id。
transaction_id = client.start_local_transaction(table_name, key)
```

2. 对局部事务范围内的数据进行读写操作。

   对局部事务范围内数据的读写操作与正常读写数据操作基本相同，只需填入局部事务ID即可。

   - 写入一行数据。

```
primary_key = [('PK0', 1), ('PK1', 'transaction')]
attribute_columns = [('value', 'origion value')]
row = Row(primary_key, attribute_columns)
condition = Condition(RowExistenceExpectation.IGNORE)
consumed, return_row = client.put_row(table_name, row, condition)
```

   - 读取此行数据。

```
primary_key = [('PK0', 1), ('PK1', 'transaction')]
columns_to_get = ['value']
consumed, return_row, next_token = client.get_row(
    table_name, primary_key, columns_to_get, None, 1, None, None, None, None, transac
tion_id
)
for att in return_row.attribute_columns:
    print ('\tname:%s\tvalue:%s' % (att[0], att[1]))
```

3. 提交或丢弃局部事务。

   - 提交局部事务，使局部事务中的所有数据修改生效。

```
client.commit_transaction(transaction_id)
```

   - 丢弃局部事务，局部事务中的所有数据修改均不会应用到原有数据。

```
client.abort_transaction(transaction_id)
```

5.4.10. 原子计数器

将列当成一个原子计数器使用，对该列进行原子计数操作，可用于为某些在线应用提供实时统计功能，例如统计帖子的PV（实时浏览量）等。

> **说明** 从Python SDK 5.1.0以上版本开始支持原子计数器功能。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

限制

- 只支持对整型列的列值进行原子计数操作。
- 作为原子计数器的列，如果写入数据前该列不存在，则默认值为0；如果写入数据前该列已存在且列值非整型，则产生OTSParameterInvalid错误。
- 增量值可以是正数或负数，但不能出现计算溢出。如果出现计算溢出，则产生OTSParameterInvalid错误。
- 默认不返回进行原子计数操作的列值，可以通过相应操作指定返回进行原子计数操作的列值。
- 在单次更新请求中，不能对某一列同时进行更新和原子计数操作。假设列A已经执行原子计数操作，则列A不能再执行其他操作（例如列的覆盖写，列删除等）。
- 在一次BatchWriteRow请求中，支持对同一行进行多次更新操作。但是如果某一行已进行原子计数操作，则该行在此批量请求中只能出现一次。
- 原子计数操作只能作用在列值的最新版本，不支持对列值的特定版本做原子计数操作。更新完成后，原子计数操作会插入一个新的数据版本。

接口

updateRow接口新增了原子计数器的相关操作，操作说明请参见下表。

操作	说明
update_of_attribute_columns	更新类型为INCREMENT，对列执行增量变更，例如+X，-X等。

参数

参数	说明
table_name	数据表名称。
column_name	进行原子计数操作的列名。只支持对整型列的列值进行原子计数器操作。
value	对列进行增量变更的值。

示例

写入数据时，使用updateRow接口对整型列做列值的增量变更，属性列中对应类型为INCREMENT。

```
def increment_by_update_row(client):
    primary_key = [('pk0', 1)]
    # INCREMENT类型为自增列，如下示例对price属性列值+6。
    update_of_attribute_columns = {
        'INCREMENT': [('price', 6)]
    }
    row = Row(primary_key, update_of_attribute_columns)
    consumed, return_row = client.update_row(table_name, row, None)
    print ('Update succeed, consume %s write cu.' % consumed.write)
```

5.4.11. 过滤器

在服务端对读取结果再进行一次过滤，根据过滤器（Filter）中的条件决定返回的行。使用过滤器后，只返回符合条件的数据行。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过GetRow、BatchGetRow或GetRange接口查询数据时，可以使用过滤器只返回符合条件的数据行。

过滤器目前包括SingleColumnCondition和CompositeColumnCondition。

- SingleColumnCondition：只判断某个参考列的列值。
- CompositeColumnCondition：根据多个参考列的列值的判断结果进行逻辑组合，决定是否过滤某行。

限制

- 过滤器的条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。
- 过滤器中的参考列必须在读取的结果内。如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。
- 在GetRow、BatchGetRow和GetRange接口中使用过滤器不会改变接口的原生语义和限制项。

  使用GetRange接口时，一次扫描数据的行数不能超过5000行或者数据大小不能超过4 MB。

  当在该次扫描的5000行或者4 MB数据中没有满足过滤器条件的数据时，得到的Response中的Rows为空，但是next_start_primary_key可能不为空，此时需要使用next_start_primary_key继续读取数据，直到next_start_primary_key为空。

参数

参数	说明
column_name	过滤器中参考列的名称。
column_value	过滤器中参考列的对比值。

参数	说明
ComparatorType	过滤器中的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=）。
LogicOperator	过滤器中的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR。
pass_if_missing	当参考列在某行中不存在时，是否返回该行。类型为bool值，默认值为True，表示如果参考列在某行中不存在，则返回该行。 当设置pass_if_missing为False时，如果参考列在某行中不存在，则不返回该行。
latest_version_only	当参考列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为True，表示如果参考列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置latest_version_only为False时，如果参考列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就返回该行。

示例

- 构造SingleColumnCondition。

```
def get_row_with_condition(client):
    primary_key = [('uid',1), ('gid',101)]
    columns_to_get = [] # 设置需要返回的列。如果不设置，则表示返回所有列。
    //设置过滤器，当name的值为'杭州'时，返回该行。
    cond = SingleColumnCondition("name", '杭州', ComparatorType.EQUAL, pass_if_missing = T
rue)
    consumed, return_row, next_token = client.get_row(table_name, primary_key, columns_to
_get, cond, 1)
    print ('Read succeed, consume %s read cu.' % consumed.read)
    print ('Value of primary key: %s' % return_row.primary_key)
    print ('Value of attribute: %s' % return_row.attribute_columns)
    for att in return_row.attribute_columns:
        print ('name:%s\tvalue:%s\ttimestamp:%d' % (att[0], att[1], att[2]))
```

- 构造CompositeColumnCondition。

```
def get_row_with_composite_condition(client):
    primary_key = [('uid',1), ('gid',101)]
    columns_to_get = [] # given a list of columns to get, or empty list if you want to ge
t entire row.
    //设置条件为(growth == 0.9) AND (name == '杭州')。
    cond = CompositeColumnCondition(LogicalOperator.AND)
    cond.add_sub_condition(SingleColumnCondition("growth", 0.9, ComparatorType.EQUAL))
    cond.add_sub_condition(SingleColumnCondition("name", '杭州', ComparatorType.EQUAL))
    consumed, return_row, next_token = client.get_row(table_name, primary_key, columns_to
_get, cond, 1)
    print ('Read succeed, consume %s read cu.' % consumed.read)
    print ('Value of primary key: %s' % return_row.primary_key)
    print ('Value of attribute: %s' % return_row.attribute_columns)
    for att in return_row.attribute_columns:
        print ('name:%s\tvalue:%s\ttimestamp:%d' % (att[0], att[1], att[2]))
```

5.5. 单行数据操作

表格存储提供了PutRow、GetRow、UpdateRow和DeleteRow等单行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

插入一行数据（PutRow）

PutRow接口用于新写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。

- 接口

```
"""
说明：写入一行数据。返回本次操作消耗的CapacityUnit。
``table_name``是数据表名称。
``row``是行数据，包括主键和属性列。
``condition``表示执行操作前做条件检查，满足条件才执行，是tablestore.metadata.Condition类的实例。支持对行的存在性和列条件进行检查，其中行存在性检查条件包括'IGNORE'、'EXPECT_EXIST'和'EXPECT_NOT_EXIST'。
``return_type``表示返回类型，是tablestore.metadata.ReturnType类的实例。目前仅支持返回PrimaryKey，一般用于主键列自增中。
返回：本次操作消耗的CapacityUnit和需要返回的行数据。
``consumed``表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
``return_row``表示返回的行数据，可能包括主键、属性列。
"""
def put_row(self, table_name, row, condition = None, return_type = None)
```

- 参数

参数	说明
table_name	数据表名称。

参数	说明
primary_key	行的主键。 **说明** ○ 设置的主键个数和类型必须和数据表的主键个数和类型一致。 ○ 当主键为自增列时，只需将自增列的值设置为占位符。更多信息，请参见主键列自增。
attribute_columns	行的属性列。 ○ 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType（可选）、时间戳（可选）。 ○ ColumnType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、BOOLEAN、DOUBLE五种，分别用ColumnType.INTEGER、ColumnType.STRING、ColumnType.BINARY、ColumnType.BOOLEAN、ColumnType.DOUBLE表示，其中BINARY不可省略，其他类型都可以省略。 ○ 时间戳即数据的版本号。更多信息，请参见数据版本和生命周期。 数据的版本号可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件。更多信息，请参见条件更新。 **说明** ○ RowExistenceExpectation.IGNORE表示无论此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。 ○ RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。 ○ RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。

- 示例

  插入一行数据。

  说明 如下示例中属性列age的版本为1498184687000，此值是2017年06月23日，如果**当前时间-max_time_deviation**（max_time_deviation由创建数据表时指定）大于1498184687000时，则PutRow时会被禁止。

```
## 主键的第一个主键列是gid，值是整数1，第二个主键列是uid，值是整数101。
primary_key = [('gid',1), ('uid',101)]
## 属性列包括五个：
##              第一个属性列的名字是name，值是字符串John，版本号没有指定，使用系统当前时间作为版本号。
##              第二个属性列的名字是mobile，值是整数13900006666，版本号没有指定，使用系统当前时间作为版本号。
##              第三个属性列的名字是address，值是二进制的China，版本号没有指定，使用系统当前时间作为版本号。
##              第四个属性列的名字是female，值是布尔值False，版本号没有指定，使用系统当前时间作为版本号。
##              第五个属性列的名字是age，值是29.7，指定版本号为1498184687000。
attribute_columns = [('name','John'), ('mobile',13900006666),('address', bytearray('China')),('female', False), ('age', 29.7, 1498184687000)]
## 通过primary_key和attribute_columns构造Row。
row = Row(primary_key, attribute_columns)
# 设置条件更新，行条件检查为期望行不存在。如果行存在会出现Condition Update Failed错误。
condition = Condition(RowExistenceExpectation.EXPECT_NOT_EXIST)
try :
    # 调用put_row方法，如果没有指定ReturnType，则return_row为None。
    consumed, return_row = client.put_row(table_name, row, condition)
    # 打印此次请求消耗的写CU。
    print ('put row succeed, consume %s write cu.' % consumed.write)
# 客户端异常，一般为参数错误或者网络异常。
except OTSClientError as e:
    print "put row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.get_error_message())
# 服务端异常，一般为参数错误或者流控错误。
except OTSServiceError as e:
    print "put row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

详细代码请参见Put Row@GitHub。

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

读取的结果可能有如下两种：

- 如果该行存在，则返回该行的各主键列以及属性列。
- 如果该行不存在，则返回中不包含行，并且不会报错。
- 接口

```
"""
说明：获取一行数据。
``table_name``是数据表名称。
``primary_key``是主键，类型为list。
``columns_to_get``是可选参数，表示要获取的列的名称列表，类型为list；如果不填写，表示获取所有列。
``column_filter``是可选参数，表示读取指定条件的行。
``max_version``是可选参数，表示最多读取的版本数，max_version和time_range必须至少设置一个。
``time_range``是可选参数，表示读取版本号范围或特定版本号的数据，max_version和time_range必须至少设置一个。
返回：本次操作消耗的CapacityUnit、主键列和属性列。
``consumed``表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
``return_row``表示行数据，包括主键列和属性列，类型都为list，例如[('PK0',value0), ('PK1',value1)]。
``next_token``表示宽行读取时下一次读取的位置，编码的二进制。
"""
def get_row(self, table_name, primary_key, columns_to_get=None,
        column_filter=None, max_version=None, time_range=None,
        start_column=None, end_column=None, token=None):
```

- 参数

参数	说明
table_name	数据表名称。
primary_key	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
columns_to_get	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ◦ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columns_to_get参数限制。如果将col0和col1加入到columns_to_get中，则只返回col0和col1列的值。 ◦ 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。

参数	说明
max_version	最多读取的版本数。 **说明** max_version与time_range必须至少设置一个。 ○ 如果仅设置max_version，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_version和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。
time_range	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** max_version与time_range必须至少设置一个。 ○ 如果仅设置max_version，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_version和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 如果查询一个范围的数据，则需要设置start_time和end_time。start_time和end_time分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[start_time, end_time)。 ○ 如果查询特定版本号的数据，则需要设置specific_time。specific_time表示特定的时间戳。 specific_time和[start_time, end_time)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为INT64.MAX。
column_filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。

- 示例

  读取一行数据。

```
# 主键的第一列是uid，值是整数1，第二列是gid，值是整数101。
primary_key = [('uid',1), ('gid',101)]
# 需要返回的属性列name、growth、type。如果columns_to_get为[]，则返回所有属性列。
columns_to_get = ['name', 'growth', 'type']
# 设置过滤器，增加列filter，当growth列的值不等于0.9且name列的值等于'杭州'时，则返回该行。
cond = CompositeColumnCondition(LogicalOperator.AND)
cond.add_sub_condition(SingleColumnCondition("growth", 0.9, ComparatorType.NOT_EQUAL))
cond.add_sub_condition(SingleColumnCondition("name", '杭州', ComparatorType.EQUAL))
try:
    # 调用get_row接口查询，最后一个参数值1表示只需要返回一个版本的值。
    consumed, return_row, next_token = client.get_row(table_name, primary_key, columns_to
_get, cond, 1)
    print ('Read succeed, consume %s read cu.' % consumed.read)
    print ('Value of primary key: %s' % return_row.primary_key)
    print ('Value of attribute: %s' % return_row.attribute_columns)
    for att in return_row.attribute_columns:
        # 打印每一列的key、value和version值。
        print ('name:%s\tvalue:%s\ttimestamp:%d' % (att[0], att[1], att[2]))
# 客户端异常，一般为参数错误或者网络异常。
except OTSClientError as e:
    print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.ge
t_error_message())
# 服务端异常，一般为参数错误或者流控错误。
except OTSServiceError as e:
    print "get row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s
" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

详细代码请参见Get Row@GitHub。

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，可以增加和删除一行中的属性列，删除属性列指定版本的数据，或者更新已存在的属性列的值。如果更新的行不存在，则新增一行数据。

> **说明** 当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。

- 接口

```
"""
说明：更新一行数据。
``table_name``是数据表名称。
``row``表示更新的行数据，包括主键列和属性列，主键列和属性列的类型均是list。
``condition``表示执行操作前做条件检查，满足条件才执行，是tablestore.metadata.Condition类的实例。
支持对行的存在性和列条件进行检查，其中行存在性检查条件包括'IGNORE'、'EXPECT_EXIST'和'EXPECT_NOT_E
XIST'。
``return_type``表示返回类型，是tablestore.metadata.ReturnType类的实例。目前仅支持返回PrimaryKe
y，一般用于主键列自增中。
返回：本次操作消耗的CapacityUnit和需要返回的行数据return_row。
consumed表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
return_row表示需要返回的行数据。
"""
def update_row(self, table_name, row, condition, return_type = None)
```

- 参数

参数	说明
table_name	数据表名称。
primary_key	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
update_of_attribute_columns	更新的属性列。 ○ 增加或更新数据时，需要设置属性名、属性值、属性类型（可选）、时间戳（可选）。 时间戳即数据的版本号，可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。详情请参见数据版本和生命周期。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。 ○ 删除属性列特定版本的数据时，只需要设置属性名和时间戳。 时间戳是64位整数，单位为毫秒，表示某个特定版本的数据。 ○ 删除属性列时，只需要设置属性名。 **说明** 删除一行的全部属性列不等同于删除该行，如果需要删除该行，请使用DeleteRow操作。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

- 示例

  更新一行数据。

```
# 主键的第一列是uid，值是整数1，第二列是gid，值是整数101。
primary_key = [('uid',1), ('gid',101)]
# 更新包括PUT，DELETE和DELETE_ALL三部分。
# PUT：新增或者更新列值。示例中是新增两列，第一列名字是name，值是David，第二列名字是address，值是Hongkong。
# DELETE：删除指定版本号（时间戳）的值。示例中是删除版本为1488436949003的address列的值。
# DELETE_ALL：删除列。示例中是删除mobile和age两列的所有版本的值。
update_of_attribute_columns = {
    'PUT' : [('name','David'), ('address','Hongkong')],
    'DELETE' : [('address', None, 1488436949003)],
    'DELETE_ALL' : [('mobile'), ('age')],
}
row = Row(primary_key, update_of_attribute_columns)
# 行条件检查为忽略，无论行是否存在，均会更新。
condition = Condition(RowExistenceExpectation.IGNORE, SingleColumnCondition("age", 20, ComparatorType.EQUAL)) # update row only when this row is exist
try:
    consumed, return_row = client.update_row(table_name, row, condition)
# 客户端异常，一般为参数错误或者网络异常。
except OTSClientError as e:
    print "update row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.get_error_message())
# 服务端异常，一般为参数错误或者流控错误。
except OTSServiceError as e:
    print "update row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

详细代码请参见UpdateRow@GitHub。

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行数据。如果删除的行不存在，则不会发生任何变化。

- 接口

```
"""
说明：删除一行数据。
``table_name``是数据表名称。
``row``表示行数据，在delete_row中仅包含主键。
``condition``表示执行操作前做条件检查，满足条件才执行，是tablestore.metadata.Condition类的实例。支持对行的存在性和列条件进行检查，其中行存在性检查条件包括'IGNORE'、'EXPECT_EXIST'和'EXPECT_NOT_EXIST'。
返回：本次操作消耗的CapacityUnit和需要返回的行数据return_row。
consumed表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
return_row表示需要返回的行数据。
"""
def delete_row(self, table_name, row, condition, return_type = None):
```

- 参数

参数	说明
table_name	数据表名称。

参数	说明
primary_key	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

- 示例

  删除一行数据。

```
primary_key = [('gid',1), ('uid','101')]
row = Row(primary_key)
try:
    consumed, return_row = client.delete_row(table_name, row, None)
# 客户端异常，一般为参数错误或者网络异常。
except OTSClientError as e:
    print "update row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e
.get_error_message())
# 服务端异常，一般为参数错误或者流控错误。
except OTSServiceError as e:
    print "update row failed, http_status:%d, error_code:%s, error_message:%s, request_id
:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id(
))
print ('Delete succeed, consume %s write cu.' % consumed.write)
```

  详细代码请参见DeleteRow@GitHub。

5.6. 多行数据操作

表格存储提供了BatchWriteRow、BatchGetRow、GetRange等多行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

批量写（BatchWriteRow）

批量写接口用于在一次请求中进行批量的写入操作，也支持一次对多个数据表进行写入。BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，构造子操作的过程与使用PutRow接口、UpdateRow接口和DeleteRow接口时相同，也支持使用条件更新。

BatchWriteRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量写入可能存在部分行失败的情况，失败行的Index及错误信息在返回的BatchWriteRowResponse中，但并不抛出异常。因此调用BatchWriteRow接口时，需要检查返回值，判断每行的状态是否成功；如果不检查返回值，则可能会忽略掉部分操作的失败。

当服务端检查到某些操作出现参数错误时，BatchWriteRow接口可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。

- 接口

```
"""
说明：批量修改多行数据。
request = MiltiTableInBatchWriteRowItem()
request.add(TableInBatchWriteRowItem(table0, row_items))
request.add(TableInBatchWriteRowItem(table1, row_items))
response = client.batch_write_row(request)
``response``为返回的结果，类型为tablestore.metadata.BatchWriteRowResponse。
"""
def batch_write_row(self, request):
```

- 参数

  详细参数说明请参见单行数据操作。

- 示例

  批量写数据。

```
put_row_items = []
# 增加PutRow的行。
for i in range(0, 10):
    primary_key = [('gid',i), ('uid',i+1)]
    attribute_columns = [('name','somebody'+str(i)), ('address','somewhere'+str(i)), ('ag
e',i)]
    row = Row(primary_key, attribute_columns)
    condition = Condition(RowExistenceExpectation.IGNORE)
    item = PutRowItem(row, condition)
    put_row_items.append(item)
# 增加UpdateRow的行。
for i in range(10, 20):
    primary_key = [('gid',i), ('uid',i+1)]
    attribute_columns = {'put': [('name','somebody'+str(i)), ('address','somewhere'+str(i
)), ('age',i)]}
    row = Row(primary_key, attribute_columns)
    condition = Condition(RowExistenceExpectation.IGNORE, SingleColumnCondition("age", i,
ComparatorType.EQUAL))
    item = UpdateRowItem(row, condition)
    put_row_items.append(item)
# 增加DeleteRow的行。
delete_row_items = []
for i in range(10, 20):
    primary_key = [('gid',i), ('uid',i+1)]
    row = Row(primary_key)
    condition = Condition(RowExistenceExpectation.IGNORE)
    item = DeleteRowItem(row, condition)
    delete_row_items.append(item)
# 构造批量写请求。
request = BatchWriteRowRequest()
request.add(TableInBatchWriteRowItem(table_name, put_row_items))
request.add(TableInBatchWriteRowItem('notExistTable', delete_row_items))
# 调用batch_write_row方法执行批量写，如果请求参数等错误会抛异常；如果部分行失败，则不会抛异常，但是内部的Item会失败。
```

```
try:
    result = client.batch_write_row(request)
    print ('Result status: %s'%(result.is_all_succeed()))
    # 检查Put行的结果。
    print ('check first table\'s put results:')
    succ, fail = result.get_put()
    for item in succ:
        print ('Put succeed, consume %s write cu.' % item.consumed.write)
    for item in fail:
        print ('Put failed, error code: %s, error message: %s' % (item.error_code, item.e
rror_message))
    # 检查Update行的结果。
    print ('check first table\'s update results:')
    succ, fail = result.get_update()
    for item in succ:
        print ('Update succeed, consume %s write cu.' % item.consumed.write)
    for item in fail:
        print ('Update failed, error code: %s, error message: %s' % (item.error_code, ite
m.error_message))
    # 检查Delete行的结果。
    print ('check second table\'s delete results:')
    succ, fail = result.get_delete()
    for item in succ:
        print ('Delete succeed, consume %s write cu.' % item.consumed.write)
    for item in fail:
        print ('Delete failed, error code: %s, error message: %s' % (item.error_code, ite
m.error_message))
# 客户端异常，一般为参数错误或者网络异常。
except OTSClientError as e:
    print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.ge
t_error_message())
# 服务端异常，一般为参数错误或者流控错误。
except OTSServiceError as e:
    print "get row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s
" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

代码详情请参见BatchWriteRow@GitHub。

批量读（BatchGetRow）

批量读接口用于一次请求读取多行数据，也支持一次对多个数据表进行读取。BatchGetRow由多个GetRow子操作组成。构造子操作的过程与使用GetRow接口时相同，也支持使用过滤器。

批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA列。

BatchGetRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量读取可能存在部分行失败的情况，失败行的错误信息在返回的BatchGetRowResponse中，但并不抛出异常。因此调用BatchGetRow接口时，需要检查返回值，判断每行的状态是否成功。

- 接口

```
"""
说明：批量获取多行数据。
request = BatchGetRowRequest()
request.add(TableInBatchGetRowItem(myTable0, primary_keys, column_to_get=None, column_fil
ter=None))
request.add(TableInBatchGetRowItem(myTable1, primary_keys, column_to_get=None, column_fil
ter=None))
request.add(TableInBatchGetRowItem(myTable2, primary_keys, column_to_get=None, column_fil
ter=None))
request.add(TableInBatchGetRowItem(myTable3, primary_keys, column_to_get=None, column_fil
ter=None))
response = client.batch_get_row(request)
``response``为返回的结果，类型为tablestore.metadata.BatchGetRowResponse。
"""
def batch_get_row(self, request):
```

- 参数

  详细参数说明请参见单行数据操作。

- 示例

  批量一次读取3行数据。

```
# 设置需要返回的列。
columns_to_get = ['name', 'mobile', 'address', 'age']
# 读取3行。
rows_to_get = []
for i in range(0, 3):
    primary_key = [('gid',i), ('uid',i+1)]
    rows_to_get.append(primary_key)
# 过滤条件为name等于John，且address等于China。
cond = CompositeColumnCondition(LogicalOperator.AND)
cond.add_sub_condition(SingleColumnCondition("name", "John", ComparatorType.EQUAL))
cond.add_sub_condition(SingleColumnCondition("address", 'China', ComparatorType.EQUAL))
# 构造批量读请求。
request = BatchGetRowRequest()
# 增加表table_name中需要读取的行，最后一个参数1表示读取最新的一个版本。
request.add(TableInBatchGetRowItem(table_name, rows_to_get, columns_to_get, cond, 1))
# 增加表notExistTable中需要读取的行。
request.add(TableInBatchGetRowItem('notExistTable', rows_to_get, columns_to_get, cond, 1)
)
try:
       result = client.batch_get_row(request)
    print ('Result status: %s'%(result.is_all_succeed()))
    table_result_0 = result.get_result_by_table(table_name)
    table_result_1 = result.get_result_by_table('notExistTable')
    print ('Check first table\'s result:')
    for item in table_result_0:
        if item.is_ok:
            print ('Read succeed, PrimaryKey: %s, Attributes: %s' % (item.row.primary_key
, item.row.attribute_columns))
        else:
            print ('Read failed, error code: %s, error message: %s' % (item.error_code, i
tem.error_message))
    print ('Check second table\'s result:')
    for item in table_result_1:
        if item.is_ok:
            print ('Read succeed, PrimaryKey: %s, Attributes: %s' % (item.row.primary_key
, item.row.attribute_columns))
        else:
            print ('Read failed, error code: %s, error message: %s' % (item.error_code, i
tem.error_message))
# 客户端异常，一般为参数错误或者网络异常。
except OTSClientError as e:
    print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.ge
t_error_message())
# 服务端异常，一般为参数错误或者流控错误。
except OTSServiceError as e:
    print "get row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s
" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

代码详情请参见BatchGetRow@GitHub。

范围读（GetRange）

范围读接口用于读取一个主键范围内的数据。

范围读接口支持按照确定范围进行正序读取和逆序读取，可以设置要读取的行数。如果范围较大，已扫描的行数或者数据量超过一定限制，会停止扫描，并返回已获取的行和下一个主键信息。您可以根据返回的下一个主键信息，继续发起请求，获取范围内剩余的行。

GetRange操作可能在如下情况停止执行并返回数据。

- 扫描的行数据大小之和达到4 MB。
- 扫描的行数等于5000。
- 返回的行数等于最大返回行数。
- 当前剩余的预留读吞吐量已全部使用，余量不足以读取下一条数据。

> **说明** 表格存储表中的行默认是按照主键排序的，而主键是由全部主键列按照顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

- 接口

```
"""
说明：根据范围条件获取多行数据。
``table_name``是数据表名称。
``direction``表示范围读取的读取方向，字符串格式，取值包括'FORWARD'和'BACKWARD'。
``inclusive_start_primary_key``表示范围的起始主键（在范围内）。
``exclusive_end_primary_key``表示范围的结束主键（不在范围内）。
``columns_to_get``是可选参数，表示要获取的列的名称列表，类型为list；如果不填，表示获取所有列。
``limit``是可选参数，表示最多读取多少行；如果不填，则没有限制。
``column_filter``是可选参数，表示读取指定条件的行。
``max_version``是可选参数，表示返回的最大版本数目，max_version与time_range必须至少存在一个。
``time_range``是可选参数，表示返回的版本的范围，max_version与time_range必须至少存在一个。
``start_column``是可选参数，用于宽行读取，表示本次读取的起始列。
``end_column``是可选参数，用于宽行读取，表示本次读取的结束列。
``token``是可选参数，用于宽行读取，表示本次读取的起始列位置，内容被二进制编码，来源于上次请求的返回结果中。
返回：符合条件的结果列表。
``consumed``表示本次操作消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
``next_start_primary_key``表示下次get_range操作的起始点的主健列，类型为dict。
``row_list``表示本次操作返回的行数据列表，格式为：[Row, ...]。
"""
def get_range(self, table_name, direction,
              inclusive_start_primary_key,
              exclusive_end_primary_key,
              columns_to_get=None,
              limit=None,
              column_filter=None,
              max_version=None,
              time_range=None,
              start_column=None,
              end_column=None,
              token = None):
```

- 参数

参数	说明
table_name	数据表名称。

参数	说明
direction	读取方向。 ○ 如果值为正序（FORWARD），则起始主键必须小于结束主键，返回的行按照主键由小到大的顺序进行排列。 ○ 如果值为逆序（BACKWARD），则起始主键必须大于结束主键，返回的行按照主键由大到小的顺序进行排列。 例如同一表中有两个主键A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。
inclusive_start_primary_key exclusive_end_primary_key	本次范围读的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由INF_MIN和INF_MAX类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中INF_MIN表示无限小，任何类型的值都比它大；INF_MAX表示无限大，任何类型的值都比它小。 ○ inclusive_start_primary_key表示起始主键，如果该行存在，则返回结果中一定会包含此行。 ○ exclusive_end_primary_key表示结束主键，无论该行是否存在，返回结果中都不会包含此行。 数据表中的行按主键从小到大排序，读取范围是一个左闭右开的区间，正序读取时，返回的是大于等于起始主键且小于结束主键的所有的行。
limit	数据的最大返回行数，此值必须大于 0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。此时可以通过返回结果中的next_start_primary_key记录本次读取到的位置，用于下一次读取。
columns_to_get	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ○ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columns_to_get参数限制。如果将col0和col1加入到columns_to_get中，则只返回col0和col1列的值。 ○ 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。 ○ 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。

参数	说明
max_version	最多读取的版本数。 **说明** max_version与time_range必须至少设置一个。 ○ 如果仅设置max_version，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_version和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。
time_range	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** max_version与time_range必须至少设置一个。 ○ 如果仅设置max_version，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_version和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 如果查询一个范围的数据，则需要设置start_time和end_time。start_time和end_time分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[start_time, end_time)。 ○ 如果查询特定版本号的数据，则需要设置specific_time。specific_time表示特定的时间戳。 specific_time和[start_time, end_time)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为INT64.MAX。
column_filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。
next_start_primary_key	根据返回结果中的next_start_primary_key判断数据是否全部读取。 ○ 当返回结果中next_start_primary_key不为空时，可以使用此返回值作为下一次GetRange操作的起始点继续读取数据。 ○ 当返回结果中next_start_primary_key为空时，表示读取范围内的数据全部返回。

- 示例

按照范围读取数据。

```
# 设置范围读的起始主键。
inclusive_start_primary_key = [('uid', INF_MIN), ('gid', INF_MIN)]
# 设置范围读的结束主键。
exclusive_end_primary_key = [('uid', INF_MAX), ('gid', INF_MAX)]
# 查询所有列。
columns_to_get = []
# 每次最多返回90行，如果总共有100个结果，首次查询时指定limit=90，则第一次最多返回90，最少可能返回0个结果，但是next_start_primary_key不为None。
limit = 90
# 设置过滤器。
cond = CompositeColumnCondition(LogicalOperator.AND)
cond.add_sub_condition(SingleColumnCondition("address", 'China', ComparatorType.EQUAL))
cond.add_sub_condition(SingleColumnCondition("age", 50, ComparatorType.LESS_THAN))
try:
    # 调用get_range接口。
    consumed, next_start_primary_key, row_list, next_token = client.get_range(
        table_name, Direction.FORWARD,
        inclusive_start_primary_key, exclusive_end_primary_key,
        columns_to_get,
        limit,
        column_filter=cond,
        max_version=1,
        time_range = (1557125059000, 1557129059000)  # start_time大于等于1557125059000，end_time小于1557129059000。
    )
    all_rows = []
    all_rows.extend(row_list)
    # 当next_start_primary_key不为空时，则继续读取。
    while next_start_primary_key is not None:
        inclusive_start_primary_key = next_start_primary_key
        consumed, next_start_primary_key, row_list, next_token = client.get_range(
            table_name, Direction.FORWARD,
            inclusive_start_primary_key, exclusive_end_primary_key,
            columns_to_get, limit,
            column_filter=cond,
            max_version=1
        )
        all_rows.extend(row_list)
    # 打印主键和属性列。
    for row in all_rows:
        print(row.primary_key, row.attribute_columns)
    print('Total rows: ', len(all_rows))
# 客户端异常，一般为参数错误或者网络异常。
except OTSClientError as e:
    print
    "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.get_error_message())
# 服务端异常，一般为参数错误或者流控错误。
except OTSServiceError as e:
    print
    "get row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

代码详情请参见GetRange@GitHub。

5.7. 多元索引

5.7.1. 创建多元索引

使用CreateSearchIndex接口在数据表上创建一个多元索引。一个数据表可以创建多个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（time_to_live）必须为-1，最大版本数（max_versions）必须为1。

参数

创建多元索引时，需要指定数据表名称（table_name）、多元索引名称（index_name）和索引的结构信息（schema），其中schema包含field_schemas（Index的所有字段的设置）、index_setting（索引设置）和index_sort（索引预排序设置）。详细参数说明请参见下表。

组成	说明
table_name	数据表名称。
index_name	多元索引名称。

组成	说明
field_schemas	field_schema的列表，每个field_schema包含如下内容： • field_name（必选）：创建多元索引的字段名，即列名，类型为String。 多元索引中的字段可以是主键列或者属性列。 • field_type（必选）：字段类型，类型为FieldType.XXX。更多信息，请参见字段。 • is_array（可选）：是否为数组，类型为Boolean。 如果设置为true，则表示该列是一个数组，在写入时，必须按照JSON数组格式写入，例如["a","b","c"]。 由于Nested类型是一个数组，当field_type为Nested类型时，无需设置此参数。 • index（可选）：是否开启索引，类型为Boolean。 默认为true，表示对该列构建倒排索引或者空间索引；如果设置为false，则不会对该列构建索引。 • analyzer（可选）：分词器类型。当字段类型为Text时，可以设置此参数；如果不设置，则默认分词器类型为单字分词。关于分词的更多信息，请参见分词。 • enable_sort_and_agg（可选）：是否开启排序与统计聚合功能，类型为Boolean。 只有enable_sort_and_agg设置为true的字段才能进行排序。关于排序的更多信息，请参见排序和翻页。 注意 Nested类型的字段不支持开启排序与统计聚合功能，但是Nested类型内部的子列支持开启排序与统计聚合功能。 • store（可选）：是否在多元索引中附加存储该字段的值，类型为Boolean。 开启后，可以直接从多元索引中读取该字段的值，而不必反查数据表，可用于查询性能优化。 • sub_field_schemas（可选）：当字段类型为Nested类型时，需要通过此参数设置嵌套文档中子列的索引类型，类型为field_schema的列表。
index_setting	索引设置，包含routing_fields设置。 routing_fields（可选）：自定义路由字段。可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

组成	说明
index_sort	索引预排序设置，包含sorters设置。如果不设置，则默认按照主键排序。 说明 含有Nested类型的索引不支持indexSort，没有预排序。 sorters（必选）：索引的预排序方式，支持按照主键排序和字段值排序。关于排序的更多信息，请参见排序和翻页。 • PrimaryKeySort表示按照主键排序，包含如下设置： sort_order：排序的顺序，可按升序或者降序排序，默认为升序（SortOrder.ASC）。 • FieldSort表示按照字段值排序，包含如下设置： 只有建立索引且开启排序与统计聚合功能的字段才能进行预排序。 ○ field_name：排序的字段名。 ○ sort_order：排序的顺序，可按照升序或者降序排序，默认为升序（SortOrder.ASC）。 ○ sort_mode：当字段存在多个值时的排序方式。

示例

```
field_a = FieldSchema('k', FieldType.KEYWORD, index=True, enable_sort_and_agg=True, store=T
rue)
field_b = FieldSchema('t', FieldType.TEXT, index=True, store=True, analyzer=AnalyzerType.SI
NGLEWORD)
field_c = FieldSchema('g', FieldType.GEOPOINT, index=True, store=True)
field_d = FieldSchema('ka', FieldType.KEYWORD, index=True, is_array=True, store=True)
field_e = FieldSchema('la', FieldType.LONG, index=True, is_array=True, store=True)
field_n = FieldSchema('n', FieldType.NESTED, sub_field_schemas=[
    FieldSchema('nk', FieldType.KEYWORD, index=True, store=True),
    FieldSchema('nl', FieldType.LONG, index=True, store=True),
    FieldSchema('nt', FieldType.TEXT, index=True, store=True),
])
fields = [field_a, field_b, field_c, field_d, field_e, field_n]
index_setting = IndexSetting(routing_fields=['PK1'])
index_sort = None # can not set index sort if there is any nested field.
#index_sort = Sort(sorters=[PrimaryKeySort(SortOrder.ASC)])
index_meta = SearchIndexMeta(fields, index_setting=index_setting, index_sort=index_sort)
client.create_search_index(table_name, index_name, index_meta)
```

5.7.2. 查询多元索引描述信息

创建多元索引后，使用DescribeSearchIndex接口可以查询多元索引的描述信息，包括多元索引的字段信息和索引配置等。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。

- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。

示例

```
index_meta, sync_stat = client.describe_search_index(table_name, index_name)
print json.dumps(index_meta, default=lambda x:x.__dict__, indent=2) //打印多元索引的index_meta信息。
print json.dumps(sync_stat, default=lambda x:x.__dict__, indent=2) //打印多元索引数据同步状态。
```

5.7.3. 列出多元索引列表

创建多元索引后，使用ListSearchIndex接口可以获取当前实例下或某个数据表关联的所有多元索引的列表信息。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称，可以为空。 • 如果设置了数据表名称，则返回该数据表关联的所有多元索引的列表。 • 如果未设置数据表名称，则返回当前实例下所有多元索引的列表。

示例

```
for table, index_name in client.list_search_index(table_name):
    print table, index_name
```

5.7.4. 删除多元索引

使用DeleteSearchIndex接口可以删除指定数据表的一个多元索引。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。

- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。

示例

```
client.delete_search_index(table_name, index_name)
```

5.7.5. 精确查询

TermQuery采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。对于Text类型字段，只要分词后有词条可以精确匹配即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
query_type	设置查询类型为TermQuery。
field_name	要匹配的字段。
term	查询关键词，即要匹配的值。 该词不会被分词，会被当做完整词去匹配。 对于Text类型字段，只要分词后有词条可以精确匹配即可。例如某个Text类型的字段，值为“tablestore is cool”，如果分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。
table_name	数据表名称。
index_name	多元索引名称。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。

参数	说明
get_total_count	是否返回匹配的总行数，默认为False，表示不返回。 返回匹配的总行数会影响查询性能。
columns_to_get	是否返回所有列，包含return_type和column_names设置。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中k列的值精确匹配'key000'的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = TermQuery('k', 'key000')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = TermQuery('k', 'key000')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = TermQuery('k', 'key000')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.6. 多词精确查询

类似于TermQuery，但是TermsQuery可以指定多个查询关键词，查询匹配这些词的数据。多个查询关键词中只要有一个词精确匹配，该行数据就会被返回，等价于SQL中的In。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
query_type	设置查询类型为TermsQuery。
field_name	要匹配的字段。
terms	多个查询关键词，即要匹配的值。 多个查询关键词中只要有一个词精确匹配，该行数据就会被返回。
table_name	数据表名称。
index_name	多元索引名称。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为False，表示不返回。 返回匹配的总行数会影响查询性能。
columns_To_get	是否返回所有列，包含return_type和column_names设置。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中k列的值精确匹配'key000'、'key100'、'key888'、'key999'、'key908'或'key1000'中任意一个的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = TermsQuery('k', ['key000', 'key100', 'key888', 'key999', 'key908', 'key1000'])
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = TermsQuery('k', ['key000', 'key100', 'key888', 'key999', 'key908', 'key1000'])
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = TermsQuery('k', ['key000', 'key100', 'key888', 'key999', 'key908', 'key1000'])
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.7. 全匹配查询

MatchAllQuery可以匹配所有行，常用于查询表中数据总行数，或者随机返回几条数据。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	描述
query	设置查询类型为MatchAllQuery。
table_name	数据表名称。
index_name	多元索引名称。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。

参数	描述
columns_to_get	是否返回所有列。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中数据的总行数。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = MatchAllQuery()
all_rows = []
next_token = None
while not all_rows or next_token:
    search_response = client.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED))
    all_rows.extend(search_response.rows)
for row in all_rows:
    print row
print 'Total rows:', len(all_rows)
```

  如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = MatchAllQuery()
all_rows = []
next_token = None
while not all_rows or next_token:
    rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client
.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED)).v1_response()
    all_rows.extend(rows)
for row in all_rows:
    print row
print 'Total rows:', len(all_rows)
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = MatchAllQuery()
all_rows = []
next_token = None
while not all_rows or next_token:
    rows, next_token, total_count, is_all_succeed = client.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFI
ED))
    all_rows.extend(rows)
for row in all_rows:
    print row
print 'Total rows:', len(all_rows)
```

5.7.8. 匹配查询

MatchQuery采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用MatchPhraseQuery实现高性能的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
field_name	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
query	设置查询类型为MatchQuery。
table_name	数据表名称。
index_name	多元索引名称。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。

参数	说明
operator	逻辑运算符。默认为OR，表示当分词后的多个词只要有部分匹配时，则行数据满足查询条件。 如果设置operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。
minimum_should_match	最小匹配个数。 只有当某一行数据的field_name列的值中至少包括最小匹配个数的词时，才会返回该行数据。 说明 minimum_should_match需要与逻辑运算符OR配合使用。
get_total_count	是否返回匹配的总行数，默认为False，表示不返回。 返回匹配的总行数会影响查询性能。
columns_to_get	是否返回所有列。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中t列的值近似匹配'this is 0'的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = MatchQuery('t', 'this is 0')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = MatchQuery('t', 'this is 0')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = MatchQuery('t', 'this is 0')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.9. 短语匹配查询

类似于MatchQuery，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。如果查询列的分词类型为模糊分词，则使用MatchPhraseQuery可以实现比WildcardQuery更快的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
field_name	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“…，this is tablestore”、“this is a table”，但是无法匹配到“this table is …”以及“is this a table”。
query	设置查询类型为MatchPhraseQuery。
table_name	数据表名称。
index_name	多元索引名称。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为False，表示不返回。 返回匹配的总行数会影响查询性能。

参数	说明
columns_to_get	是否返回所有列。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中t列的值按照顺序完整匹配'this is'短语的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = MatchPhraseQuery('t', 'this is')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = MatchPhraseQuery('t', 'this is')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = MatchPhraseQuery('t', 'this is')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.10. 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。

- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
field_name	要匹配的列。
prefix	前缀值。 对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。
query	设置查询类型为PrefixQuery。
table_name	数据表名称。
index_name	多元索引名称。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为False，表示不返回。 返回匹配的总行数会影响查询性能。
columns_to_get	是否返回所有列，包含return_type和column_names设置。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中k列的值中前缀为'key00'的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = PrefixQuery('k', 'key00')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = PrefixQuery('k', 'key00')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = PrefixQuery('k', 'key00')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.11. 范围查询

RangeQuery根据范围条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足范围条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
field_name	要匹配的字段。
range_from	起始位置的值。
range_to	结束位置的值。
include_lower	结果中是否需要包括range_from值，类型为Boolean。
include_upper	结果中是否需要包括range_to值，类型为Boolean。
table_name	数据表名称。
index_name	多元索引名称。
query	设置查询类型为RangeQuery。

参数	说明
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为False，表示不返回。 返回匹配的总行数会影响查询性能。
columns_to_get	是否返回所有列，包含return_type和column_names设置。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中k列的值在'key100'到'key200'之间的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = RangeQuery('k', 'key100', 'key200', include_lower=False, include_upper=False)
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = RangeQuery('k', 'key100', 'key200', include_lower=False, include_upper=False)
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = RangeQuery('k', 'key100', 'key200', include_lower=False, include_upper=False)
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.12. 通配符查询

通配符查询中，要匹配的值可以是一个带有通配符的字符串，目前支持星号（*）和问号（?）两种通配符。要匹配的值中可以用星号（*）代表任意字符序列，或者用问号（?）代表任意单个字符，且支持以星号（*）或问号（?）开头。例如查询“table*e”，可以匹配到“tablestore”。

如果查询的模式为*word*，则您可以使用性能更好的模糊查询，具体实现方法如下：

1. 创建多元索引时，设置列为Text类型且设置分词类型为模糊分词。
2. 使用多元索引查询数据时，使用MatchPhraseQuery且设置查询词为word。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	描述
query	设置查询类型为WildcardQuery。
field_name	列名称。
value	带有通配符的字符串，字符串长度不能超过20个字符。
table_name	数据表名称。
index_name	多元索引名称。
limit	本次查询需要返回的最大数量。
get_total_count	是否返回匹配的总行数，默认为False。 设置get_total_count为True后会影响查询性能。
ColumnsToGet	是否返回所有列。 • 当设置return_type为ColumnReturnType.SPECIFIED时，需要指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

查询表中k列的值匹配'key00*'的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
query = WildcardQuery('k', 'key00*')
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
query = WildcardQuery('k', 'key00*')
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
query = WildcardQuery('k', 'key00*')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.13. 地理位置查询

地理位置查询包括地理距离查询（GeoDistanceQuery）、地理长方形范围查询（GeoBoundingBoxQuery）和地理多边形范围查询（GeoPolygonQuery）三种方式。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

地理距离查询（GeoDistanceQuery）

GeoDistanceQuery根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，满足查询条件。

- 参数

参数	说明
field_name	列名，类型为Geopoint。
center_point	中心地理坐标点，是一个经纬度值。 格式为 **纬度,经度** ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。

参数	说明
distance	距离中心点的距离，类型为Double。单位为米。
query	多元索引的查询语句。设置查询类型为GeoDistanceQuery。
table_name	数据表名称。
index_name	多元索引名称。

- 示例

  查询表中g列的值距离中心点'32.5,116.5'不超过300000米的数据。

  - 5.2.1及之后版本

    使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

    ```
    query = GeoDistanceQuery('g', '32.5,116.5', 300000)
    search_response = client.search(
        table_name, index_name,
        SearchQuery(query, limit=100, get_total_count=True),
        ColumnsToGet(return_type=ColumnReturnType.ALL)
    )
    ```

    如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

    ```
    query = GeoDistanceQuery('g', '32.5,116.5', 300000)
    rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.s
    earch(
        table_name, index_name,
        SearchQuery(query, limit=100, get_total_count=True),
        ColumnsToGet(return_type=ColumnReturnType.ALL)
    ).v1_response()
    ```

  - 5.2.1之前版本

    使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

    ```
    query = GeoDistanceQuery('g', '32.5,116.5', 300000)
    rows, next_token, total_count, is_all_succeed = client.search(
        table_name, index_name,
        SearchQuery(query, limit=100, get_total_count=True),
        ColumnsToGet(return_type=ColumnReturnType.ALL)
    )
    ```

地理长方形范围查询（GeoBoundingBoxQuery）

GeoBoundingBoxQuery根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时，满足查询条件。

- 参数

参数	说明
field_name	列名，类型为Geopoint。

参数	说明
top_left	长方形框的左上角的坐标。
bottom_right	长方形框的右下角的坐标，通过左上角和右下角可以确定一个唯一的长方形。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。
query	多元索引的查询语句。设置查询类型为GeoBoundingBoxQuery。
table_name	数据表名称。
index_name	多元索引名称。

- 示例

  查询表中g列的值在左上角坐标为'30.9,112.0'，右下角坐标为'30.2,119.0'的长方形范围内的数据。

```
query = GeoBoundingBoxQuery('g', '30.9,112.0', '30.2,119.0')
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

地理多边形范围查询（GeoPolygonQuery）

GeoPolygonQuery根据一个多边形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时，满足查询条件。

- 参数

参数	说明
field_name	列名，类型为Geopoint。
points	组成多边形范围的坐标，通过多个坐标可以确定一个唯一的多边形。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。
query	多元索引的查询语句。设置查询类型为GeoPolygonQuery。
table_name	数据表名称。
index_name	多元索引名称。

- 示例

  查询表中g列的值在由'30.9,112.0'、'30.5,115.0'、'30.3, 117.0'和'30.2,119.0'坐标组成的多边形范围内的数据。

```
query = GeoPolygonQuery('g', ['30.9,112.0', '30.5,115.0', '30.3, 117.0', '30.2,119.0'])
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.14. 多条件组合查询

BoolQuery查询条件包含一个或者多个子查询条件，根据子查询条件来判断一行数据是否满足查询条件。每个子查询条件可以是任意一种Query类型，包括BoolQuery。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
must_queries	多个Query列表，行数据必须满足所有的子查询条件才算匹配，等价于And操作符。
must_not_queries	多个Query列表，行数据必须不能满足任何的子查询条件才算匹配，等价于Not操作符。
filter_queries	多个Query列表，行数据必须满足所有的子filter才算匹配，filter类似于query，区别是filter不会根据满足的filter个数进行相关性算分。
should_queries	多个Query列表，可以满足，也可以不满足，等价于Or操作符。 行数据应该至少满足should_queries子查询条件的最小匹配个数才算匹配。 如果满足的should_queries子查询条件个数越多，则整体的相关性分数更高。
minimum_should_match	should_queries子查询条件的最小匹配个数。当同级没有其他Query，只有should_queries时，默认值为1；当同级已有其他Query，例如must_queries、must_not_queries和filter_queries时，默认值为0。
table_name	数据表名称。
index_name	多元索引名称。

示例

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
# k > 'key100' and (l > 110 and l < 200) and not (k = 'key121')
# and should_queries(k > 'key120' or l < 300, minimum_should_match=2)
bool_query = BoolQuery(
    must_queries=[
        RangeQuery('k', range_from='key100', include_lower=False),
        //多条件组合查询。
        BoolQuery(
            //设置需要满足的子查询条件。
            must_queries=[
                RangeQuery('l', range_from=110, include_lower=False),
                RangeQuery('l', range_to=200, include_upper=False)
            ],
        )
    ],
    //设置需要排除的子查询条件。
    must_not_queries=[
        TermQuery('k', 'key121')
    ],
    should_queries=[
        RangeQuery('k', range_from='key120', include_lower=False),
        RangeQuery('l', range_to=300, include_upper=130)
    ],
    minimum_should_match=2
)
//构造完整查询语句，包括排序的列，返回前100行以及返回查询结果总的行数。
search_response = client.search(
    table_name, index_name,
    SearchQuery(
        bool_query,
        sort=Sort(sorters=[FieldSort('l', SortOrder.ASC)]),
        limit=100,
        get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
# k > 'key100' and (l > 110 and l < 200) and not (k = 'key121')
# and should_queries(k > 'key120' or l < 300, minimum_should_match=2)
bool_query = BoolQuery(
    must_queries=[
        RangeQuery('k', range_from='key100', include_lower=False),
        //多条件组合查询。
        BoolQuery(
            //设置需要满足的子查询条件。
            must_queries=[
                RangeQuery('l', range_from=110, include_lower=False),
                RangeQuery('l', range_to=200, include_upper=False)
            ],
        )
    ],
    //设置需要排除的子查询条件。
    must_not_queries=[
        TermQuery('k', 'key121')
    ],
    should_queries=[
        RangeQuery('k', range_from='key120', include_lower=False),
        RangeQuery('l', range_to=300, include_upper=130)
    ],
    minimum_should_match=2
)
//构造完整查询语句，包括排序的列，返回前100行以及返回查询结果总的行数。
rows, next_token, total_count, is_all_succeed, agg_results, group_by_results = client.sea
rch(
    table_name, index_name,
    SearchQuery(
        bool_query,
        sort=Sort(sorters=[FieldSort('l', SortOrder.ASC)]),
        limit=100,
        get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
# k > 'key100' and (l > 110 and l < 200) and not (k = 'key121')
# and should_queries(k > 'key120' or l < 300, minimum_should_match=2)
bool_query = BoolQuery(
    must_queries=[
        RangeQuery('k', range_from='key100', include_lower=False),
        //多条件组合查询。
        BoolQuery(
            //设置需要满足的子查询条件。
            must_queries=[
                RangeQuery('l', range_from=110, include_lower=False),
                RangeQuery('l', range_to=200, include_upper=False)
            ],
        )
    ],
    //设置需要排除的子查询条件。
    must_not_queries=[
        TermQuery('k', 'key121')
    ],
    should_queries=[
        RangeQuery('k', range_from='key120', include_lower=False),
        RangeQuery('l', range_to=300, include_upper=130)
    ],
    minimum_should_match=2
)
//构造完整查询语句，包括排序的列，返回前100行以及返回查询结果总的行数。
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(
        bool_query,
        sort=Sort(sorters=[FieldSort('l', SortOrder.ASC)]),
        limit=100,
        get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.15. 嵌套类型查询

NestedQuery用于查询嵌套类型字段中子行的数据。嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型字段的路径和一个子查询，其中子查询可以是任意Query类型。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
path	路径名，嵌套类型的列的树状路径。例如news.title表示嵌套类型的news列中的title子列。

参数	说明
query	嵌套类型的列中子列上的查询，子列上的查询可以是任意Query类型。
score_mode	当列存在多个值时基于哪个值计算分数。
table_name	数据表名称。
index_name	多元索引名称。

示例

查询表中n.n1列的值大于等于100且小于等于300的数据。

- 5.2.1及之后版本

  使用5.2.1及之后的SDK版本时，默认的返回结果为SearchResponse对象，请求示例如下：

```
nested_query = RangeQuery('n.nl', range_from=100, range_to=300, include_lower=True, inclu
de_upper=True)
query = NestedQuery('n', nested_query)
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

  如果需要返回Tuple类型结果，您可以使用如下请求示例实现。

```
nested_query = RangeQuery('n.nl', range_from=100, range_to=300, include_lower=True, inclu
de_upper=True)
query = NestedQuery('n', nested_query)
rows, next_token, total_count, is_all_succeed, agg_result, group_by_results = client.sear
ch(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
).v1_response()
```

- 5.2.1之前版本

  使用5.2.1之前的SDK版本时，默认的返回结果为Tuple类型，请求示例如下：

```
nested_query = RangeQuery('n.nl', range_from=100, range_to=300, include_lower=True, inclu
de_upper=True)
query = NestedQuery('n', nested_query)
rows, next_token, total_count, is_all_succeed = client.search(
    table_name, index_name,
    SearchQuery(query, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

5.7.16. 排序和翻页

您可以在创建多元索引时指定索引预排序和在使用多元索引查询数据时指定排序方式，以及在获取返回结果时使用limit和offset或者使用token进行翻页。

索引预排序

多元索引默认按照设置的索引预排序（IndexSort）方式进行排序，使用多元索引查询数据时，IndexSort决定了数据的默认返回顺序。

在创建多元索引时，您可以自定义IndexSort，如果未自定义IndexSort，则IndexSort默认为主键排序。

> **注意** 含有Nested类型字段的多元索引不支持索引预排序。

查询时指定排序方式

只有enable_sort_and_agg设置为True的字段才能进行排序。

在每次查询时，可以指定排序方式，多元索引支持如下四种排序方式（Sorter）。您也可以使用多个Sorter，实现先按照某种方式排序，再按照另一种方式排序的需求。

- ScoreSort

  按照查询结果的相关性（BM25算法）分数进行排序，适用于有相关性的场景，例如全文检索等。

  > **注意** 如果需要按照相关性打分进行排序，必须手动设置ScoreSort，否则会按照索引设置的IndexSort进行排序。

  ```
  sort = Sort(
      sorters=[ScoreSort(sort_order=SortOrder.DESC)]
  )
  client.search(
      table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
  ), ColumnsToGet(return_type=ColumnReturnType.ALL)
  )
  ```

- PrimaryKeySort

  按照主键进行排序。

  ```
  sort = Sort(
      sorters=[PrimaryKeySort(sort_order=SortOrder.DESC)]
  )
  client.search(
      table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
  ), ColumnsToGet(return_type=ColumnReturnType.ALL)
  ) = PrimaryKeySort(sort_order=SortOrder.DESC)
  ```

- FieldSort

  按照某列的值进行排序。

  基础类型的字段排序：

```
sort = Sort(
    sorters=[FieldSort('l', SortOrder.ASC)]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

Nested类型的字段排序：

```
sort = Sort(
    sorters=[
        FieldSort(
            'n.nl',
            sort_order=SortOrder.ASC,
            nested_filter=NestedFilter('n', RangeQuery('n.nl', range_from=150, range_to=2
00))
        )
    ]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

先按照某列的值进行排序，再按照另一列的值进行排序。

```
sort = Sort(
    sorters=[
        FieldSort('a', SortOrder.ASC),
        FieldSort('b', SortOrder.ASC)
    ]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

- GeoDistanceSort

  根据地理点距离进行排序。

```
sort = Sort(
    sorters=[GeoDistanceSort('g', ['32.5,116.5', '32.0,116.0'], sort_order=SortOrder.DESC
, sort_mode=SortMode.MAX)]
)
client.search(
    table_name, index_name, SearchQuery(query, sort=sort, limit=100, get_total_count=True
), ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

翻页方式

在获取返回结果时，可以使用limit和offset或者使用token进行翻页。

- 使用limit和offset进行翻页

  当需要获取的返回结果行数小于10000行时，可以使用limit和offset进行翻页，即limit+offset<=10000，其中limit的最大值为100。

  > **说明** 如果需要提高limit的上限，请参见如何将多元索引Search接口查询数据的limit提高到1000。

  如果使用此方式进行翻页时未设置limit和offset，则limit的默认值为10，offset的默认值为0。

```
query = RangeQuery('k', 'key100', 'key500', include_lower=False, include_upper=False)
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, offset=100, limit=100, get_total_count=True),
    ColumnsToGet(return_type=ColumnReturnType.ALL)
)
```

- 使用token进行翻页

  由于使用token进行翻页时翻页深度无限制，当需要进行深度翻页时，推荐使用token进行翻页。

  当符合查询条件的数据未读取完时，服务端会返回next_token，此时可以使用next_token继续读取后面的数据。

  使用token进行翻页时默认只能向后翻页。由于在一次查询的翻页过程中token长期有效，您可以通过缓存并使用之前的token实现向前翻页。

  使用token翻页后的排序方式和上一次请求的一致，无论是系统默认使用IndexSort还是自定义排序，因此设置了token不能再设置Sort。另外使用token后不能设置offset，只能依次往后读取，即无法跳页。

  > **注意** 由于含有Nested类型字段的多元索引不支持索引预排序，如果使用含有Nested类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回next_token。

```
query = MatchAllQuery()
all_rows = []
next_token = None
# first round
search_response = client.search(table_name, index_name,
        SearchQuery(query, next_token=next_token, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFIED))
all_rows.extend(search_response.rows)
# loop
while search_response.next_token:
    search_response = client.search(table_name, index_name,
        SearchQuery(query, next_token=search_response.next_token, sort=None, limit=100, get_total_count=True),
        columns_to_get=ColumnsToGet(['k', 't', 'g', 'ka', 'la'], ColumnReturnType.SPECIFIED))
    all_rows.extend(search_response.rows)
print('Total rows:%d' % len(all_rows))
```

5.7.17. 统计聚合

使用统计聚合功能可以实现求最小值、求最大值、求和、求平均值、统计行数、去重统计行数、按字段值分组、按范围分组、按地理位置分组、按过滤条件分组等；同时多个统计聚合功能可以组合使用，满足复杂的查询需求。

> **说明** 从Python SDK 5.2.1及以上版本开始支持统计聚合功能。

最小值

返回一个字段中的最小值，类似于SQL中的min。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

  统计年龄为18岁的人中得分的最低分数。

```
query = TermQuery('age', 18)
agg = Min('score', name = 'min')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=0, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

最大值

返回一个字段中的最大值，类似于SQL中的max。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long和Double类型。

参数	说明
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

  统计年龄为18岁的人中得分的最高分数。如果某人没有分数，则对应分数的默认值为0。

```
query = TermQuery('age', 18)
agg = Max('score', missing_value = 0, name = 'max')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=0, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

和

返回数值字段的总数，类似于SQL中的sum。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

  统计年龄为18岁的所有人得分的总和。

```
query = TermQuery('age', 18)
agg = Sum('score', name = 'sum')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

平均值

返回数值字段的平均值，类似于SQL中的avg。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long和Double类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置missing值，则在统计聚合时会忽略该行。 ○ 如果设置了missing值，则使用missing值作为字段值的默认值参与统计聚合。

- 示例

  统计年龄为18岁的所有人得分的平均分。

```
query = TermQuery('age', 18)
agg = Avg('score', name = 'avg')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

统计行数

返回指定字段值的数量或者表中数据总行数，类似于SQL中的count。

> **说明** 通过如下方式可以统计表中数据总行数或者某个query匹配的行数。
> - 使用统计聚合的count功能，在请求中设置count(*)。
> - 使用query功能的匹配行数，在query中设置setGetTotalCount(true)；如果需要统计表中数据总行数，则使用MatchAllQuery。
>
> 如果需要获取表中数据某列出现的次数，则使用count（列名），可应用于稀疏列的场景。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。

- 示例

  统计年龄为18岁的人中参加考试有分数的人数。

```
query = TermQuery('age', 18)
agg = Count('score', name = 'count')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

去重统计行数

返回指定字段不同值的数量，类似于SQL中的count（distinct）。

> **说明** 去重统计行数的计算结果是个近似值。
> - 当去重统计行数小于1万时，计算结果是一个精确值。
> - 当去重统计行数达到1亿时，计算结果的误差为2%左右。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long、Double、Boolean、Keyword和Geo_point类型。
missing	当某行数据中的字段为空时，字段值的默认值。 ○ 如果未设置Missing值，则在统计聚合时会忽略该行。 ○ 如果设置了Missing值，则使用Missing值作为字段值的默认值参与统计聚合。

- 示例

  去重统计年龄为18岁的人中一共有多少种不同的姓名。

```
query = TermQuery('age', 18)
agg = DistinctCount('name', name = 'distinct_name')
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, aggs=[agg]
),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for agg_result in search_response.agg_results:
    print('{\n"name":"%s",\n"value":%s\n}\n' % (agg_result.name, str(agg_result.value)))
```

字段值分组

根据一个字段的值对查询结果进行分组，相同的字段值放到同一分组内，返回每个分组的值和该值对应的个数。

> **说明** 当分组较大时，按字段值分组可能会存在误差。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long、Double、Boolean和Keyword类型。
size	返回的分组数量。
group_by_sort	分组中的item排序规则，默认按照分组中item的数量降序排序，多个排序则按照添加的顺序进行排列。支持的参数如下： ○ 按照值的字典序升序排列 ○ 按照值的字典序降序排列 ○ 按照行数升序排列 ○ 按照行数降序排列 ○ 按照子统计聚合结果中值升序排列 ○ 按照子统计聚合结果中值降序排列
Ssub_aggs和 sub_group_bys	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 ○ 场景 统计每个类别的商品数量，且统计每个类别价格的最大值和最小值。 ○ 方法 最外层的统计聚合是根据类别进行分组，再添加两个子统计聚合求价格的最大值和最小值。 ○ 结果示例 ■ 水果：5个（其中价格最高15元，最低3元） ■ 洗漱用品：10个（其中价格最高98元，最低1元） ■ 电子设备：3个（其中价格最高8699元，最低2300元） ■ 其它：15个（其中价格最高1000元，最低80元）

- 示例1

  将年龄为18岁的人按分数分组，并获取人数最多的10个分数值以及每个分数的人数。

```
query = TermQuery('age', 18)
group_by = GroupByField('score', size = 10)
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=20, group_bys
= [group_by]),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("key:%s, count:%d" % (item.key, item.row_count))
```

- 示例2

  将年龄为18岁的人按分数分组，并获取人数最少的2个分数值以及每个分数的人数。

```
group_by = GroupByField('score', size = 2, group_by_sort = [RowCountSort(sort_order=SortO
rder.ASC)])
search_response = client.search(table_name, index_name,
                                SearchQuery(TermQuery('age', 18), limit=100, get_total_co
unt=True, group_bys = [group_by]),
                                ColumnsToGet(return_type=ColumnReturnType.ALL_FROM_INDEX)
)
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("key:%s, count:%d" % (item.key, item.row_count))
```

- 示例3

  将年龄为18岁的人按分数分组，并获取人数最多的2个分数值、每个分数的人数以及按主键排序前三的人的信息。

```
sort = RowCountSort(sort_order = SortOrder.DESC)
sub_agg = [TopRows(limit=3,sort=Sort([PrimaryKeySort(sort_order=SortOrder.DESC)]), name =
't1')]
group_by = GroupByField('l', size = 2, group_by_sort = [sort], sub_aggs = sub_agg)
search_response = client.search(table_name, index_name,
                                SearchQuery(TermQuery('age', 18), limit=100, get_total_co
unt=True, group_bys = [group_by]),
                                ColumnsToGet(return_type=ColumnReturnType.ALL_FROM_INDEX)
)
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("\tkey:%s, count:%d" % (item.key, item.row_count))
        for sub_agg in item.sub_aggs:
            print("\t\tname:%s:" % sub_agg.name)
            for entry in sub_agg.value:
                print("\t\t\tvalue:%s" % str(entry))
```

- 示例4

将年龄为18岁的人按分数和性别分组，

```
sort = RowCountSort(sort_order = SortOrder.ASC)
sub_group = GroupByField('sex', size = 10, group_by_sort = [sort])
group_by = GroupByField('score', size = 10, group_by_sort = [sort], sub_group_bys = [sub_
group])
search_response = client.search(table_name, index_name,
                                SearchQuery(TermQuery('age', 18), limit=100, get_total_co
unt=True, group_bys = [group_by]),
                                ColumnsToGet(return_type=ColumnReturnType.ALL_FROM_INDEX)
)
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("\tkey:%s, count:%d" % (item.key, item.row_count))
        for sub_group in item.sub_group_bys:
            print("\t\tname:%s:" % sub_group.name)
            for sub_item in sub_group.items:
                print("\t\t\tkey:%s, count:%s" % (str(sub_item.key), str(sub_item.row_cou
nt)))
```

范围分组

根据一个字段的范围对查询结果进行分组，字段值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Long和Double类型。
range[double_from, double_to)	分组的范围。 起始值double_from可以使用最小值Double.MIN_VALUE，结束值double_to可以使用最大值Double.MAX_VALUE。
sub_aggs和sub_group_bys	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。 例如按销量分组后再按省份分组，即可获得某个销量范围内哪个省比重比较大，实现方法是GroupByRange下添加一个GroupByField。

- 示例

  统计年龄为18岁的人中得分的分数在[80, 90)和[90, 100)两个区间段的人数。

```
query = TermQuery('age', 18)
group_by = GroupByRange(field_name = 'score', ranges = [(80, 90),(90, 100)])
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=0, group_bys
= [group_by]),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("range:%.1f~%.1f, count:%d" % (item.range_from, item.range_to, item.row_cou
nt))
```

地理位置分组

根据距离某一个中心点的范围对查询结果进行分组，距离差值在某范围内放到同一分组内，返回每个范围中相应的item个数。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
field	用于统计聚合的字段，仅支持Geo_point类型。
origin(double lat, double lon)	起始中心点的经纬度。 lat是起始中心点坐标纬度，lon是起始中心点坐标经度。
range[double_from, double_to)	分组的范围，单位为米。 起始值double_from可以使用最小值Double.MIN_VALUE，结束值double_to可以使用最大值Double.MAX_VALUE。
sub_aggs和 sub_group_bys	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

  统计年龄为18岁的人中家庭住址在距离学校一公里以内和一公里到两公里内的人数。其中学校经纬度为(31,116)。

```
query = TermQuery('age', 18)
group_by = GroupByGeoDistance(field_name = 'address', origin=GeoPoint(31, 116), ranges =
[(0, 1000), (1000,2000)])
search_response = client.search(table_name, index_name,
                                SearchQuery(query, next_token = None, limit=2, group_bys
= [group_by]),
                                columns_to_get = ColumnsToGet(return_type = ColumnReturnT
ype.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    for item in group_by.items:
        print("range:%.1f~%.1f, count:%d" % (item.range_from, item.range_to, item.row_cou
nt))
```

过滤条件分组

按照过滤条件对查询结果进行分组，获取每个过滤条件匹配到的数量，返回结果的顺序和添加过滤条件的顺序一致。

- 参数

参数	说明
name	自定义的统计聚合名称，用于区分不同的统计聚合，可根据此名称获取本次统计聚合结果。
filter	过滤条件，返回结果的顺序和添加过滤条件的顺序一致。
sub_aggs和 sub_group_bys	子统计聚合，子统计聚合会根据分组内容再进行一次统计聚合分析。

- 示例

  分别统计年龄为18岁的人中数学考了100分和语文考了100分的人数。

```
query = TermQuery('age', 18)
filter1 = TermQuery('math', 100)
filter2 = TermQuery('chinese', 100)
filters = [filter1, filter2]
group_by = GroupByFilter(filters)
search_response = client.search(
    table_name, index_name,
    SearchQuery(query, next_token = None, limit=2, group_bys = [group_by]),
    columns_to_get = ColumnsToGet(return_type = ColumnReturnType.ALL_FROM_INDEX))
for group_by in search_response.group_by_results:
    print("name:%s" % group_by.name)
    print("groups:")
    i = 0
    for item in group_by.items:
        print("filter:%s=%s, count:%d" % (str(filters[i].field_name), str(filters[i].colu
mn_value), item.row_count))
        i=i+1
```

5.8. 二级索引

5.8.1. 全局二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（time_to_live）必须为-1，最大版本数（max_versions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
main_table_name	数据表名称。
index_meta	索引表的结构信息，包括如下内容： ○ index_name：索引表名称。 ○ primary_key_names：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ defined_column_names：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ index_type：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置index_type或者设置index_type为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置index_type为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。

参数	说明
include_base_data	索引表中是否包含数据表中已存在的数据。 当设置include_base_data为true时，表示包含存量数据；设置include_base_data为false时，表示不包含存量数据。

- 示例

  在主键为pk1、pk2的数据表上创建主键列为definedcol1、pk1，属性列为definedcol2、definedcol3的索引表。

```
index_meta = SecondaryIndexMeta('index2', ['definedcol1', 'pk1'], ['definedcol2', 'define
dcol3'])
client.create_secondary_index(table_name, index_meta)
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - table_name需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  使用GetRange接口读取索引表中数据时有如下注意事项：

  - table_name需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
main_table_name	数据表名称。
index_name	索引表名称。

- 示例

```
client.delete_secondary_index(table_name, 'index1')
```

5.8.2. 本地二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（time_to_live）必须为-1，最大版本数（max_versions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
main_table_name	数据表名称。
index_meta	索引表的结构信息，包括如下内容： ○ index_name：索引表名称。 ○ primary_key_names：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ defined_column_names：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ index_type：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置index_type或者设置index_type为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置index_type为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。

参数	说明
include_base_data	索引表中是否包含数据表中已存在的数据。 当设置include_base_data为true时，表示包含存量数据；设置include_base_data为false时，表示不包含存量数据。

- 示例

  在主键为pk1、pk2的数据表上创建主键列为pk1、definedcol1，属性列为definedcol2、definedcol3的索引表。

```
index_meta = SecondaryIndexMeta('index2', ['pk1', 'definedcol1'], ['definedcol2', 'define
dcol3'],index_type= SecondaryIndexType.LOCAL_INDEX)
client.create_secondary_index(table_name, index_meta)
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - table_name需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  使用GetRange接口读取索引表中数据时有如下注意事项：

  - table_name需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
main_table_name	数据表名称。
index_name	索引表名称。

- 示例

```
client.delete_secondary_index(table_name, 'index1')
```

5.9. 错误处理

本文介绍表格存储Python SDK的错误处理方式和重试策略。

方式

表格存储Python SDK目前采用“异常”的方式处理错误。如果调用接口没有抛出异常，则说明操作成功，否则失败。

> **说明** 批量相关接口，例如BatchGetRow和BatchWriteRow不仅需要判断是否有异常，还需要检查每行的状态是否成功，只有全部成功后才能保证整个接口调用是成功的。

异常

表格存储Python SDK中有OTSClientError和OTSServiceError两种异常，都最终继承自Exception。

- OTSClientError：指SDK内部出现的异常，例如参数设置错误，返回结果解析失败等。
- OTSServiceError：指服务器端错误，来自于对服务器错误信息的解析。OTSServiceError包含以下几个成员：
  - get_http_status：HTTP返回码，例如200、404等。
  - get_error_code：表格存储返回的错误类型字符串。
  - get_error_message：表格存储返回的错误消息字符串。
  - get_request_id：用于唯一标识此次请求的UUID。当您无法解决问题时，记录此requestId并提交工单。

重试

- SDK中出现错误时会自动重试。默认策略是最大重试次数为20，最大重试间隔为3000毫秒。对流控类错误以及读操作相关的服务端内部错误进行的重试，请参见tablestore/retry.py。
- 您也可以通过继承RetryPolicy类实现自定义重试策略，在构造OTSClient对象时，将自定义的重试策略作为参数传入。

目前SDK中已经实现的重试策略如下。

- DefaultRetryPolicy：默认重试策略，只会对读操作重试，最大重试次数为20，最大重试间隔为3000毫秒。
- NoRetryPolicy：不进行任何重试。
- NoDelayRetryPolicy：没有延时的重试策略，请谨慎使用。
- WriteRetryPolicy：在默认重试策略基础上，会对写操作重试。

6.Node.js SDK

6.1. 前言

本文介绍表格存储Node.js SDK的使用。本文内容适用于4.x.x版本。

前提条件

- 已开通表格存储。具体操作，请参见开通表格存储服务。
- 已创建AccessKey。具体操作，请参见获取AccessKey。

SDK下载

- SDK包
- GitHub

版本

当前最新版本：4.1.0

6.2. 安装

本文介绍如何安装表格存储Node.js SDK。

环境准备

适用于Node.js 4.0及以上版本。

> **说明** 由于兼容性问题，建议您不要使用Node.js 12.0版本~12.14版本。

安装

安装命令如下。

```
npm install tablestore
```

> **说明** 如果使用npm遇到网络问题，可以使用淘宝提供的npm镜像，详情请参见cnpm。

示例程序

Node.js SDK提供丰富的示例程序，方便参考或直接使用。您可以通过以下两种方式获取示例程序。

- 下载表格存储Node.js SDK开发包，解压后examples为示例程序。
- 访问表格存储Node.js SDK的GitHub项目。

6.3. 初始化

Client是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、单行数据、多行数据等。

确定Endpoint

Endpoint是阿里云表格存储服务在各个区域的域名地址，您可以通过以下方式查询Endpoint：

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

> **说明** 关于Endpoint的更多信息，请参见服务地址。

配置密钥

要接入阿里云的表格存储服务，您需要拥有一个有效的访问密钥进行签名认证。目前支持下面三种方式：

- 阿里云账号的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 在阿里云官网注册阿里云账号。
  ii. 创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。
- 被授予访问表格存储权限的RAM用户的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 使用阿里云账号前往访问控制RAM，创建一个新的RAM用户或者使用已经存在的RAM用户。
  ii. 使用阿里云账号授予RAM用户访问表格存储的权限。
  iii. RAM用户被授权后，即可使用自己的AccessKey ID和AccessKey Secret访问。
- 从STS获取的临时访问凭证。获取步骤如下：
  i. 应用的服务器通过访问RAM/STS服务，获取一个临时的AccessKey ID、AccessKey Secret和SecurityToken发送给使用方。
  ii. 使用方使用上述临时密钥访问表格存储服务。

初始化对接

获取到AccessKey ID和AccessKey Secret后，您可以按照下如下步骤进行初始化对接。

```
var client = new TableStore.Client({
accessKeyId: '<your access key id>',
accessKeySecret: '<your access key secret>',
endpoint: '<your endpoint>',
instancename: '<your instance name>',
maxRetries:20,//默认20次重试，可以省略此参数。
    });
```

6.4. 数据类型

本文介绍表格存储提供的五种数据类型与Node.js SDK数据类型的对应关系。

> **说明** 表格存储的数据类型请参见命名规则和数据类型。

表格存储数据类型	Node.js SDK数据类型	描述
String	string	JavaScript语言中的基本数据类型
Integer	int64	Node.js SDK封装的数据类型
Dobule	number	JavaScript语言中的基本数据类型
Boolean	boolean	JavaScript语言中的基本数据类型
Binary	Buffer	Node.js的Buffer对象

表格存储的Integer类型是一个64位的有符号整型，此数据类型在JavaScript中没有相应的数据类型可以对应，所以在Node.js中需要一个能表示64位有符号整型的数据类型，可以对表格存储的Integer类型做如下转换。

```
var numberA = TableStore.Long.fromNumber(1000);
var numberB = TableStore.Long.fromString('2000');
var num = numberA.toNumber();
    num = numberA.toString();
var str = numberB.toNumber();
    str = numberB.toString();
```

6.5. 表

6.5.1. 创建数据表

使用CreateTable接口创建数据表时，需要指定数据表的结构信息和配置信息，高性能实例中的数据表还可以根据需要设置预留读/写吞吐量。创建数据表的同时支持创建一个或者多个索引表。

> **说明**
>
> - 创建数据表后需要几秒钟进行加载，在此期间对该数据表的读/写数据操作均会失败。请等待数据表加载完毕后再进行数据操作。
> - 创建数据表时必须指定数据表的主键。主键包含1个~4个主键列，每一个主键列都有名称和类型。

前提条件

- 已通过控制台创建实例。具体操作，请参见创建实例。
- 已初始化Client。具体操作，请参见初始化。

接口

```
/**
 * 根据指定的表结构信息创建相应的数据表。
 */
createTable(params, callback)
```

参数

参数	说明
tableMeta	数据表的结构信息，包括如下内容： • tableName：数据表名称。 • primaryKey：数据表的主键定义。更多信息，请参见主键和属性。 **说明** 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。 ◦ 表格存储可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体请参见表格存储数据模型和查询操作。 ◦ 第一列主键作为分区键。分区键相同的数据会存放在同一个分区内，所以相同分区键下最好不要超过10 GB以上数据，否则会导致单分区过大，无法分裂。另外，数据的读/写访问最好在不同的分区键上均匀分布，有利于负载均衡。 • definedColumn：预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。
tableOptions	数据表的配置信息。更多信息，请参见数据版本和生命周期。 配置信息包括如下内容： • timeToLive：数据生命周期，即数据的过期时间。当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 创建数据表时，如果希望数据永不过期，可以设置数据生命周期为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 单位为秒。 **说明** 如果需要使用索引，则数据生命周期必须设置为-1（数据永不过期）。 • maxVersions：最大版本数，即属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。 **说明** 如果需要使用索引，则最大版本数必须设置为1。

参数	说明
reservedThroughput	为数据表配置预留读吞吐量或预留写吞吐量。 容量型实例中的数据表的预留读/写吞吐量只能设置为0，不允许预留。 默认值为0，即完全按量计费。 单位为CU。 • 当预留读吞吐量或预留写吞吐量大于0时，表格存储会根据配置为数据表预留相应资源，且数据表创建成功后，将会立即按照预留吞吐量开始计费，超出预留的部分进行按量计费。更多信息，请参见计费概述。 • 当预留读吞吐量或预留写吞吐量设置为0时，表格存储不会为数据表预留相应资源。
indexMetas	索引表的结构信息，每个indexMeta包括如下内容： • name：索引表名称。 • primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 • definedColumn：索引表的属性列，索引表属性列为数据表的预定义列的组合。 • includeBaseData：索引表中是否包含数据表中已存在的数据。 当设置includeBaseData为true时，表示包含存量数据；设置includeBaseData为false时，表示不包含存量数据。 • indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ◦ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ◦ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 • indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ◦ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ◦ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。

示例

- 创建数据表（不带索引）

创建一个有2个主键列，预留读/写吞吐量为(0, 0)的数据表。

```
var client = require('./client');
var params = {
  tableMeta: {
    tableName: 'sampleTable',
    primaryKey: [
      {
        name: 'gid',
        type: 'INTEGER'
      },
      {
        name: 'uid',
        type: 'INTEGER'
      }
    ]
  },
  reservedThroughput: {
    capacityUnit: {
      read: 0,
      write: 0
    }
  },
  tableOptions: {
    timeToLive: -1, //数据的过期时间，单位为秒，-1代表永不过期。如果设置过期时间为一年，即为365*24
*3600。
    maxVersions: 1  //保存的最大版本数，设置为1代表每列上最多保存一个版本（保存最新的版本）。
  }
};
client.createTable(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

详细代码请参见CreateTable@GitHub。

- 创建数据表（带索引且索引类型为全局二级索引）

```
var client = require('./client');
var TableStore = require('../index.js');
var params = {
  tableMeta: {
    tableName: 'sdkGlobalTest',
    primaryKey: [
      {
        name: 'pk1',
        type: TableStore.PrimaryKeyType.INTEGER
      },
      {
        name: 'pk2',
        type: TableStore.PrimaryKeyType.INTEGER
```

```
      }
    ],
    definedColumn: [
      {
        "name": "col1",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      },
      {
        "name": "col2",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      }
    ],
  },
  reservedThroughput: {
    capacityUnit: {
      read: 0,
      write: 0
    }
  },
  tableOptions: {
    timeToLive: -1, //数据的过期时间，单位为秒，-1表示永不过期。带索引的数据表数据生命周期必须设置为-1。
    maxVersions: 1  //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引的数据表最大版本数必须设置为1。
  },
  streamSpecification: {
    enableStream: false, //二级索引不支持开启Stream。
  },
  indexMetas: [
    {
      name: "sdkGlobalIndex1",
      primaryKey: ["pk2"],
      definedColumn: ["col1", "col2"]
    },
    {
      name: "sdkGlobalIndex2",
      primaryKey: ["col1"],
      definedColumn: ["col2"]
    }
  ]
};
client.createTable(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

- 创建数据表（带索引且索引类型为本地二级索引）

```
var client = require('./client');
var TableStore = require('../index.js');
var params = {
  tableMeta: {
```

```
    tableName: 'sdkLocalTest',
    primaryKey: [
      {
        name: 'pk1',
        type: TableStore.PrimaryKeyType.INTEGER
      },
      {
        name: 'pk2',
        type: TableStore.PrimaryKeyType.INTEGER
      }
    ],
    definedColumn: [
      {
        "name": "col1",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      },
      {
        "name": "col2",
        "type": TableStore.DefinedColumnType.DCT_INTEGER
      }
    ],
  },
  reservedThroughput: {
    capacityUnit: {
      read: 0,
      write: 0
    }
  },
  tableOptions: {
    timeToLive: -1, //数据的过期时间，单位为秒，-1表示永不过期。带索引的数据表数据生命周期必须设置为-1。
    maxVersions: 1  //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引的数据表最大版本数必须设置为1。
  },
  streamSpecification: {
    enableStream: false, //二级索引不支持开启Stream。
  },
  indexMetas: [
    {
      name: "sdklocalIndex1",
      primaryKey: ["pk1","col1"],//为索引表添加主键列。索引表的第一列主键必须与数据表的第一列主键相同。
      definedColumn: ["col2"],
      indexUpdateMode: TableStore.IndexUpdateMode.IUM_SYNC_INDEX,//设置索引更新模式为同步更新（IUM_SYNC_INDEX）。当索引类型为本地二级索引时，索引更新模式必须为同步更新。
      indexType: TableStore.IndexType.IT_LOCAL_INDEX,//设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    },
    {
      name: "sdklocalIndex2",
      primaryKey: ["pk1","col2"],
      definedColumn: ["col1"],
      indexUpdateMode: TableStore.IndexUpdateMode.IUM_SYNC_INDEX,//设置索引更新模式为同步更新（IUM_SYNC_INDEX）。当索引类型为本地二级索引时，索引更新模式必须为同步更新。
```

```
      indexType: TableStore.IndexType.IT_LOCAL_INDEX,//设置索引类型为本地二级索引（IT_LOCAL_I
NDEX）。
    }
  ]
};
client.createTable(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

6.5.2. 更新表

使用UpdateTable接口更新指定表的最大版本数、预留读吞吐量或预留写吞吐量的设置。

接口

```
/**
 * 更新指定表的预留读吞吐量或预留写吞吐量设置。
 */
updateTable(params, callback)
```

示例

更新表的最大版本数为5。

```
var client = require('./client');
var params = {
    tableName: 'sampleTable',
    tableOptions: {
        maxVersions: 5,
    }
};
client.updateTable(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见UpdateTable@GitHub。

6.5.3. 列出表名称

使用ListTable接口获取当前实例下已创建的所有表的表名。

> **说明** API说明请参见ListTable。

接口

```
/**
 * 获取当前实例下已创建的所有表的表名。
 */
listTable(params, callback)
```

示例

获取实例下所有表的表名。

```
var client = require('./client');
client.listTable({}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见ListTable@GitHub。

6.5.4. 查询表描述信息

使用DescribeTable接口可以查询指定表的结构、预留读/写吞吐量详情等信息。

> **说明** API说明请参见DescribeTable。

接口

```
/**
 * 查询指定表的结构信息和预留读/写吞吐量设置信息。
 */
describeTable(params, callback)
```

参数

参数	说明
tableName	表名。

示例

获取表的描述信息。

```
var client = require('./client');
var params = {
    tableName: 'sampleTable'
};
client.describeTable(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见DescribeTable@GitHub。

6.5.5. 删除数据表

使用DeleteTable接口删除当前实例下指定数据表。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表。
- 已删除数据表上的索引表和多元索引。

接口

```
/**
 * 删除本实例下指定数据表。
 */
deleteTable(params, callback)
```

参数

参数	说明
tableName	数据表名称。

示例

删除指定数据表。

```
var client = require('./client');
var params = {
    tableName: 'sampleTable'
};
client.deleteTable(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见DeleteTable@GitHub。

6.5.6. 主键列自增

设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。

前提条件

已初始化Client，详情请参见初始化。

使用方法

1. 创建表时，将非分区键的主键列设置为自增列。

   只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号长整型。

2. 写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符。

   如果需要获取写入数据后系统自动生成的自增列的值，将ReturnType设置为Primarykey，可以在数据写入成功后返回自增列的值。

   查询数据时，需要完整的主键值。通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为Primarykey可以获取完整的主键值。

示例

主键自增列功能主要涉及创建表（CreateTable）和写数据（PutRow、UpdateRow和BatchWriteRow）两类接口。

1. 创建表

   创建表时，只需将自增的主键属性设置为AUTO_INCREMENT。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var tableName = "autoIncTable";
var pk1 = "stringPK";
var pk2 = "autoIncPK";
function createTableWithAutoIncrementPk() {
    var createParams = {
        tableMeta: {
            tableName: tableName,
            primaryKey: [
                {
                    name: pk1,
                    type: 'STRING'
                },
                {
                    name: pk2,
                    type: 'INTEGER',
                    option: 'AUTO_INCREMENT'//自增列，指定option为AUTO_INCREMENT。
                },
            ]
        },
        reservedThroughput: {
            capacityUnit: {
                read: 0,
                write: 0
            }
        },
        tableOptions: {
            timeToLive: -1,//数据的过期时间，单位秒，-1表示数据永不过期。假如设置过期时间为一年，即为365*24*3600。
            maxVersions: 1//保存的最大版本数，设置为1表示每列上最多保存一个版本(保存最新的版本)。
        },
    };
    client.createTable(createParams, function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('create table success');
    });
}
```

2. 写数据

   写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符PK_AUTO_INCR。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var tableName = "autoIncTable";
var pk1 = "stringPK";
var pk2 = "autoIncPK";
function putRow() {
    var putParams = {
        tableName: tableName,
        condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE,
null),
        primaryKey: [
            { stringPK: 'pk1' },
            { autoIncPK: TableStore.PK_AUTO_INCR }
        ],
        attributeColumns: [
            { 'col1': 'col1val' }
        ],
        returnContent: { returnType: TableStore.ReturnType.Primarykey }
    };
    client.putRow(putParams, function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('put row success,autoIncrement pk value:' + JSON.stringify(data.row
.primaryKey));
    });
}
```

6.5.7. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过PutRow、UpdateRow、DeleteRow或BatchWriteRow接口更新数据时，可以使用条件更新检查行存在性条件和列条件，只有满足条件时才能更新成功。

条件更新包括行存在性条件和列条件。

- 行存在性条件：包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。

  对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。

- 列条件：包括SingleColumnCondition和CompositeCondition，是基于某一列或者某些列的列值进行条件判断。

- SingleColumnCondition支持一列（可以是主键列）和一个常量比较。不支持两列或者两个常量相比较。
- CompositeCondition的内节点为逻辑运算，子条件可以是SingleColumnCondition或CompositeCondition。

条件更新可以实现乐观锁功能，即在更新某行时，先获取某列的值，假设为列A，值为1，然后设置条件**列A = 1**，更新行使**列A = 2**。如果更新失败，表示有其他客户端已成功更新该行。

限制

条件更新的列条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。

参数

参数	说明
RowExistenceExpectation	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别用RowExistenceExpectation_IGNORE、RowExistenceExpectation_EXPECT_EXIST、RowExistenceExpectation_EXPECT_NOT_EXIST表示。 • IGNORE：表示忽略，不做任何存在性检查。 • EXPECT_EXIST：表示期望存在，如果该行存在，则满足条件；如果该行不存在，则不满足条件。 • EXPECT_NOT_EXIST：期望行不存在，如果该行不存在，则满足条件；如果该行存在，则不满足条件。
columnName	列的名称。
columnValue	列的对比值。
comparator	对列值进行比较的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=），分别用CT_EQUAL、CT_NOT_EQUAL、CT_GREATER_THAN、CT_GREATER_EQUAL、CT_LESS_THAN、CT_LESS_EQUAL表示。
combinator	对多个条件进行组合的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR，分别用LO_NOT、LO_AND、LO_OR表示。 逻辑运算符不同可以添加的子条件个数不同。 • 当逻辑运算符为NOT时，只能添加一个子条件。 • 当逻辑运算符为AND或OR时，必须至少添加两个子条件。
filterIfMissing	当列在某行中不存在时，条件检查是否通过。类型为bool值，默认值为true，表示如果列在某行中不存在时，则条件检查通过，该行满足更新条件。 当设置filterIfMissing为false时，如果列在某行中不存在时，则条件检查不通过，该行不满足更新条件。

参数	说明
latestVersionOnly	当列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置latestVersionOnly为false时，如果列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就条件检查通过，该行满足更新条件。

示例

根据指定主键更新一行，只有同时满足行存在，“name”列值为“john”和“addr”列值为“china”条件，才能更新成功；否则更新失败。

```
   var params = {
    tableName: "sampleTable",
    primaryKey: [{ 'gid': Long.fromNumber(20013) }, { 'uid': Long.fromNumber(20013) }],
    updateOfAttributeColumns: [{ 'PUT': [{ 'col1': 'test6' }] }]
};
//指定条件。期望行存在并且name=john，addr=china。
var condition = new TableStore.CompositeCondition(TableStore.LogicalOperator.AND);
condition.addSubCondition(new TableStore.SingleColumnCondition('name', 'john', TableStore.C
omparatorType.EQUAL));
condition.addSubCondition(new TableStore.SingleColumnCondition('addr', 'china', TableStore.
ComparatorType.EQUAL));
params.condition = new TableStore.Condition(TableStore.RowExistenceExpectation.EXPECT_EXIST
, condition);
client.updateRow(params,
    function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('success:', data);
    });
```

6.5.8. 局部事务

使用局部事务功能，创建数据范围在一个分区键值内的局部事务。对局部事务中的数据进行读写操作后，可以根据实际提交或者丢弃局部事务。局部事务通过悲观锁（Pessimistic Lock）实现并发控制。

目前局部事务功能处于邀测中，默认关闭。如果需要使用该功能，请提交工单进行申请或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。

使用局部事务可以指定某个分区键值内的操作是原子的，对分区键值内的数据进行的操作要么全部成功要么全部失败，并且所提供的隔离级别为读已提交。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

1. 使用startLocalTransaction在指定的分区键值创建一个局部事务，并获取局部事务ID。
2. 对局部事务范围内的数据进行读写操作。

   支持对局部事务进行操作的接口为GetRow、PutRow、DeleteRow、UpdateRow、BatchWriteRow和GetRange。
3. 使用commitTransaction提交局部事务或者使用abortTransaction丢弃局部事务。

限制

- 每个局部事务从创建开始生命周期最长为60秒。

  如果超过60秒未提交或丢弃局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
- 如果创建局部事务时超时，此请求可能在表格存储服务端已执行成功，此时用户需要等待该局部事务超时后重新创建。
- 未提交的局部事务可能失效，如果出现此情况，需要重试该局部事务内的操作。
- 在局部事务中读写数据有如下限制：
  - 不能使用局部事务ID访问局部事务范围（即创建时使用的分区键值）以外的数据。
  - 同一个局部事务中所有写请求的分区键值必须与创建局部事务时的分区键值相同，读请求则无此限制。
  - 一个局部事务同时只能用于一个请求中，在使用局部事务期间，其它使用此局部事务ID的操作均会失败。
  - 每个局部事务中两次读写操作的最大间隔为60秒。

    如果超过60秒未操作局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
  - 每个局部事务中写入的数据量最大为4 MB，按正常的写请求数据量计算规则累加。
  - 如果在局部事务中写入了未指定版本号的Cell，该Cell的版本号会在写入时（而非提交时）由表格存储服务端自动生成，生成规则与正常写入一个未指定版本号的Cell相同。
  - 如果BatchWriteRow请求中带有局部事务ID，则此请求中所有行只能操作该局部事务ID对应的表。
  - 在使用局部事务期间，对应分区键值的数据相当于被锁上，只有持有局部事务ID在局部事务范围内的写请求才会成功，其它不持有局部事务ID在局部事务范围内的写请求均会失败。在局部事务提交、丢弃或超时后，对应的锁也会被释放。
  - 带有局部事务ID的读写请求失败不会影响局部事务本身的存活情况，您可以按照正常的无局部事务ID的读写请求重试规则进行重试，或者主动丢弃当前局部事务。

参数

参数	说明
tableName	数据表名称。
primaryKey	数据表主键。 • 创建局部事务时，只需要指定局部事务对应的分区键值。 • 创建局部事务后，对局部事务范围内的数据进行读写操作时，需要指定完整主键。
transactionId	局部事务ID，用于唯一标识一个局部事务。 创建局部事务后，操作局部事务时均需要带上局部事务ID。

示例

- 提交局部事务，使所有数据修改生效。

```
(async () => {
    try {
        //创建局部事务。
        const response = await client.startLocalTransaction({
            tableName,
            primaryKey: [{  //只需要指定局部事务对应的分区键值。
                "id": "partitionKeyValue"
            }]
        });
        //获取局部事务ID。
        const transactionId = response.transactionId;
        //对局部事务范围内的数据进行写操作。
        await client.putRow({
            tableName,
            condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE
, null),
            primaryKey,
            attributeColumns: [{
                col: 'updated'
            }],
            transactionId
        });
        //提交局部事务。
        await client.commitTransaction({
            transactionId
        })
    } catch (e) {
        console.error(e)
    }
})();
```

- 丢弃局部事务，放弃所有数据修改。

```
(async () => {
    try {
        //创建局部事务。
        const response = await client.startLocalTransaction({
            tableName,
            primaryKey: [{  //只需要指定局部事务对应的分区键值。
                "id": "partitionKeyValue"
            }]
        });
        //获取局部事务ID。
        const transactionId = response.transactionId
        //对局部事务范围内的数据进行写操作。
        await client.putRow({
            tableName,
            condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE
, null),
            primaryKey,
            attributeColumns: [{
                col: 'updated'
            }],
            transactionId
        });
        //丢弃局部事务。
        await client.abortTransaction({
            transactionId
    })
  } catch (e) {
    console.error(e)
  }
})();
```

6.5.9. 原子计数器

将列当成一个原子计数器使用，对该列进行原子计数操作，可用于为某些在线应用提供实时统计功能，例如统计帖子的PV（实时浏览量）等。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

限制

- 只支持对整型列的列值进行原子计数操作。
- 作为原子计数器的列，如果写入数据前该列不存在，则默认值为0；如果写入数据前该列已存在且列值非整型，则产生OTSParameterInvalid错误。
- 增量值可以是正数或负数，但不能出现计算溢出。如果出现计算溢出，则产生OTSParameterInvalid错误。
- 默认不返回进行原子计数操作的列值，可以通过相应操作指定返回进行原子计数操作的列值。
- 在单次更新请求中，不能对某一列同时进行更新和原子计数操作。假设列A已经执行原子计数操作，则列A不能再执行其他操作（例如列的覆盖写，列删除等）。

- 在一次BatchWriteRow请求中，支持对同一行进行多次更新操作。但是如果某一行已进行原子计数操作，则该行在此批量请求中只能出现一次。
- 原子计数操作只能作用在列值的最新版本，不支持对列值的特定版本做原子计数操作。更新完成后，原子计数操作会插入一个新的数据版本。

接口

updateRow接口新增了原子计数器的相关操作，操作说明请参见下表。

操作	说明
updateOfAttributeColumns	更新类型为INCREMENT，对列执行增量变更，例如+X，-X等。
returnContent	对于进行原子计数操作的列，设置列名和返回类型，返回进行原子计数操作的列的新值。

参数

参数	说明
tableName	数据表名称。
columnName	进行原子计数操作的列名。只支持对整型列的列值进行原子计数操作。
value	对列进行增量变更的值。
returnColumns	对于进行原子计数操作的列，设置需要返回列值的列名。
returnType	设置返回类型为TableStore.ReturnType.AfterModify，将进行原子计数操作的列值返回。

示例

写入数据时，使用updateRow接口对整型列做列值的增量变更，属性列中对应类型为INCREMENT。

```
var params = {
  tableName: "<Your-Table-Name>",
  condition: new TableStore.Condition(TableStore.RowExistenceExpectation.EXPECT_EXIST, null
),
  primaryKey: [{'pk0': Long.fromNumber(1)}],
  //将进行原子计数操作的price列的列值+10，不能设置时间戳。
  updateOfAttributeColumns: [
    {'INCREMENT': [{'price': Long.fromNumber(10)}]}
  ],
  //设置ReturnType为TableStore.ReturnType.AfterModify，将进行原子计数操作的列值返回。
  returnContent: {
    returnColumns: ["price"],
    returnType: TableStore.ReturnType.AfterModify
  }
};
client.updateRow(params,
  function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
  });
```

6.5.10. 过滤器

在服务端对读取结果再进行一次过滤，根据过滤器（Filter）中的条件决定返回的行。使用过滤器后，只返回符合条件的数据行。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过GetRow、BatchGetRow或GetRange接口查询数据时，可以使用过滤器只返回符合条件的数据行。

过滤器目前包括SingleColumnCondition和CompositeColumnCondition。

- SingleColumnCondition：只判断某个参考列的列值。
- CompositeColumnCondition：根据多个参考列的列值的判断结果进行逻辑组合，决定是否过滤某行。

限制

- 过滤器的条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。
- 过滤器中的参考列必须在读取的结果内。如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。
- 在GetRow、BatchGetRow和GetRange接口中使用过滤器不会改变接口的原生语义和限制项。

  使用GetRange接口时，会受到一次扫描数据的行数不能超过5000行或者扫描数据的数据大小不能大于4 MB的限制。

当在该次扫描的5000行或者4 MB数据中没有满足过滤器条件的数据时，得到的Response中的Rows为空，但是next_start_primary_key可能不为空，此时需要使用next_start_primary_key继续读取数据，直到next_start_primary_key为空。

参数

参数	说明
columnName	过滤器中参考列的名称。
columnValue	过滤器中参考列的对比值。
ComparatorType	过滤器中的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=）。
LogicOperator	过滤器中的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR。
passIfMissing	当参考列在某行中不存在时，是否返回该行。类型为bool值，默认值为true，表示如果参考列在某行中不存在，则返回该行。 当passIfMissing设置为false时，如果参考列在某行中不存在，则不返回该行。
latestVersionOnly	当参考列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果参考列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当latestVersionOnly设置为false时，如果参考列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就返回该行。

示例

- 构造SingleColumnCondition。

```
function getRowWithCondition() {
  //当col1=表格存储时，返回该行。当passIfMissing设置为true时，如果该列不存在，则返回该行；当passIfMissing设置为false时，如果该列不存在，则不返回该行。
  var condition = new TableStore.SingleColumnCondition('col1', '表格存储', TableStore.ComparatorType.EQUAL,true);
  params.columnFilter = condition;
  client.getRow(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    console.log('success:', data);
  });
}
```

- 构造CompositeCondition。

```
function getRowWithCompositeCondition() {
  //设置过滤器，当col1 = 表格存储且col5 = 123456789时，返回该行。
  var condition = new TableStore.CompositeCondition(TableStore.LogicalOperator.AND);
  condition.addSubCondition(new TableStore.SingleColumnCondition('col1', '表格存储', TableStore.ComparatorType.EQUAL));
  condition.addSubCondition(new TableStore.SingleColumnCondition('col5', Long.fromNumber(123456789), TableStore.ComparatorType.EQUAL));
  params.columnFilter = condition;
  client.getRow(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    console.log('success:', data);
  });
}
```

6.6. 单行数据操作

表格存储提供了PutRow、GetRow、UpdateRow和DeleteRow等单行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

插入一行数据（PutRow）

PutRow接口用于新写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。

- 接口

```
  /**
   * 插入数据到指定的行，如果该行不存在，则新增一行；如果该行存在，则覆盖原有行。
   */
  putRow(params, callback)
```

- 参数

参数	说明
tableName	数据表名称。

参数	说明
primaryKey	行的主键。 **说明** ○ 设置的主键个数和类型必须和数据表的主键个数和类型一致。 ○ 当主键为自增列时，只需将自增列的值设置为占位符，详情请参见主键列自增。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。 **说明** ○ RowExistenceExpectation.IGNORE表示无论此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。 ○ RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。 ○ RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。
attributeColumns	行的属性列。 ○ 每一项的顺序是属性名、属性类型（可选）、属性值、时间戳（可选）。 ○ 时间戳即数据的版本号，详情请参见数据版本和生命周期。 数据的版本号可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。
returnContent	表示返回类型。 returnType：设置为TableStore.ReturnType.Primarykey，表示返回主键值，主要用于主键列自增场景。

- 示例

  插入一行数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var currentTimeStamp = Date.now();
var params = {
  tableName: "sampleTable",
  condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE, null),
  primaryKey: [{ 'gid': Long.fromNumber(20013) }, { 'uid': Long.fromNumber(20013) }],
  attributeColumns: [
    { 'col1': '表格存储' },
    { 'col2': '2', 'timestamp': currentTimeStamp },
    { 'col3': 3.1 },
    { 'col4': -0.32 },
    { 'col5': Long.fromNumber(123456789) }
  ],
  returnContent: { returnType: TableStore.ReturnType.Primarykey }
};
client.putRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

详细代码请参见PutRow@GitHub。

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

读取的结果可能有如下两种：

- 如果该行存在，则返回该行的各主键列以及属性列。
- 如果该行不存在，则返回中不包含行，并且不会报错。
- 接口

```
  /**
   * 根据给定的主键读取单行数据。
   */
  getRow(params, callback)
```

- 参数

参数	说明
tableName	数据表名称。

参数	说明
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ◦ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 ◦ 当columnsToGet和columnFilter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 ◦ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。

参数	说明
timeRange	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 ○ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 查询一个范围的数据，则需要设置startTime和endTime。startTime和endTime分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[startTime, endTime)。 ○ 如果查询特定版本号的数据，则需要设置specificTime。specificTime表示特定的时间戳。 specificTime和[startTime, endTime)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Long.MAX_VALUE。
columnFilter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当columnsToGet和columnFilter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。

- 示例

  读取一行数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sampleTable",
  primaryKey: [{ 'gid': Long.fromNumber(20004) }, { 'uid': Long.fromNumber(20004) }],
  maxVersions: 2 //最多可读取的版本数，设置为2即代表最多可读取2个版本。
};
var condition = new TableStore.CompositeCondition(TableStore.LogicalOperator.AND);
condition.addSubCondition(new TableStore.SingleColumnCondition('name', 'john', TableStore
.ComparatorType.EQUAL));
condition.addSubCondition(new TableStore.SingleColumnCondition('addr', 'china', TableStor
e.ComparatorType.EQUAL));
params.columnFilter = condition;
client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

详细代码请参见Get Row@Git Hub。

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，可以增加和删除一行中的属性列，删除属性列指定版本的数据，或者更新已存在的属性列的值。如果更新的行不存在，则新增一行数据。

> **说明** 当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。

- 接口

```
  /**
   * 更新指定行的数据。如果该行不存在，则新增一行；如果该行存在，则根据请求的内容在此行中新增、修改或者删除指定列的值。
   */
  updateRow(params, callback)
```

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。

参数	说明
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。
updateOfAttributeColumns	更新的属性列。 ○ 增加或更新数据时，需要设置属性名、属性值、属性类型（可选）、时间戳（可选）。 时间戳即数据的版本号，可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。详情请参见数据版本和生命周期。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。 ○ 删除属性列特定版本的数据时，只需要设置属性名和时间戳。 时间戳是64位整数，单位为毫秒，表示某个特定版本的数据。 ○ 删除属性列时，只需要设置属性名。 **说明** 删除一行的全部属性列不等同于删除该行，如果需要删除该行，请使用DeleteRow操作。

- 示例

  更新一行数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
    tableName: "sampleTable",
    condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE, null),
    primaryKey: [{ 'gid': Long.fromNumber(9) }, { 'uid': Long.fromNumber(90) }],
    updateOfAttributeColumns: [
        { 'PUT': [{ 'col4': Long.fromNumber(4) }, { 'col5': '5' }, { 'col6': Long.fromNum
ber(6) }] },
        { 'DELETE': [{ 'col1': Long.fromNumber(1496826473186) }] },
        { 'DELETE_ALL': ['col2'] }
    ]
};
client.updateRow(params,
    function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('success:', data);
    });
```

详细代码请参见UpdateRow@GitHub。

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行数据。如果删除的行不存在，则不会发生任何变化。

- 接口

```
/**
 * 删除一行数据。
 */
deleteRow(params, callback)
```

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。

- 示例

删除一行数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
    tableName: "sampleTable",
    condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE, null),
    primaryKey: [{ 'gid': Long.fromNumber(8) }, { 'uid': Long.fromNumber(80) }]
};
client.deleteRow(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见DeleteRow@GitHub。

6.7. 多行数据操作

表格存储提供了BatchWriteRow、BatchGetRow、GetRange等多行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

批量写（BatchWriteRow）

批量写接口用于在一次请求中进行批量的写入操作，也支持一次对多个数据表进行写入。BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，构造子操作的过程与使用PutRow接口、UpdateRow接口和DeleteRow接口时相同，也支持使用条件更新。

BatchWriteRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量写入可能存在部分行失败的情况，失败行的Index及错误信息在返回的BatchWriteRowResponse中，但并不抛出异常。因此调用BatchWriteRow接口时，需要检查返回值，判断每行的状态是否成功；如果不检查返回值，则可能会忽略掉部分操作的失败。

当服务端检查到某些操作出现参数错误时，BatchWriteRow接口可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。

- 接口

```
/**
 * 批量修改行。
 */
batchWriteRow(params, callback)
```

- 参数

  本操作是PutRow、UpdateRow、DeleteRow的组合。

  - 增加了数据表的层级结构，可以一次处理多个数据表。

    tables以数据表为单位组织，后续为各个数据表的操作，设置需要写入、修改或删除的行信息，参数说明请参见单行数据操作。

  - 增加了type参数，用于区分操作类型。

    操作类型可以为PUT、UPDATE、DELETE。

    - 当操作类型为PUT或UPDATE时，primaryKey和attributeColumns有效。
    - 当操作类型为DELETE时，primaryKey有效。

- 示例

  批量写入数据。

```
var client = require('./client');
var TableStore = require('../index.js');
var Long = TableStore.Long;
var params = {
    tables: [
        {
            tableName: 'sampleTable',
            rows: [
                {
                    type: 'UPDATE',
                    condition: new TableStore.Condition(TableStore.RowExistenceExpectatio
n.IGNORE, null),
                    primaryKey: [{ 'gid': Long.fromNumber(20010) }, { 'uid': Long.fromNum
ber(20010) }],
                    attributeColumns: [{ 'PUT': [{ 'col1': 'test3' }, { 'col2': 'test4' }
] }],
                    returnContent: { returnType: 1 }
                },
                {
                    type: 'PUT',
                    condition: new TableStore.Condition(TableStore.RowExistenceExpectatio
n.IGNORE, null),
                    primaryKey: [{ 'gid': Long.fromNumber(20020) }, { 'uid': Long.fromNum
ber(20020) }],
                    attributeColumns: [{ 'col1': 'test1' }, { 'col2': 'test2' }],
                    returnContent: { returnType: TableStore.ReturnType.Primarykey }
                },
                {
                    type: 'DELETE',
                    condition: new TableStore.Condition(TableStore.RowExistenceExpectatio
n.IGNORE, null),
                    primaryKey: [{ 'gid': Long.fromNumber(20018) }, { 'uid': Long.fromNum
ber(20018) }],
                }
            ]
        }
    ],
};
client.batchWriteRow(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见BatchWriteRow@GitHub。

批量读（BatchGetRow）

批量读接口用于一次请求读取多行数据，也支持一次对多个数据表进行读取。BatchGetRow由多个GetRow子操作组成。构造子操作的过程与使用GetRow接口时相同，也支持使用过滤器。

批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA列。

BatchGetRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量读取可能存在部分行失败的情况，失败行的错误信息在返回的BatchGetRowResponse中，但并不抛出异常。因此调用BatchGetRow接口时，需要检查返回值，判断每行的状态是否成功。

- 接口

```
/**
 * 批量读取一个或多个表中的若干行数据。
 */
batchGetRow(params, callback)
```

- 参数

  BatchGetRow和GetRow的区别如下：

  - 增加了数据表的层级结构，可以一次读取多个数据表的数据。

    tables以数据表为单位组织，后续为各个数据表的操作，设置了需要读取的行信息，参数说明请参见单行数据操作。

  - primaryKey支持设置多行的主键，可以一次读取多行数据。

- 示例

  批量一次读多个数据表、多行，单行出错时进行重试。

```
var client = require('./client');
var TableStore = require('../index.js');
var Long = TableStore.Long;
var params = {
    tables: [{
        tableName: 'sampleTable',
        primaryKey: [
            [{ 'gid': Long.fromNumber(20013) }, { 'uid': Long.fromNumber(20013) }],
            [{ 'gid': Long.fromNumber(20015) }, { 'uid': Long.fromNumber(20015) }]
        ],
        startColumn: "col2",
        endColumn: "col4"
    },
    {
        tableName: 'notExistTable',
        primaryKey: [
            [{ 'gid': Long.fromNumber(10001) }, { 'uid': Long.fromNumber(10001) }]
        ]
    }
    ],
};
var maxRetryTimes = 3;
var retryCount = 0;
function batchGetRow(params) {
    client.batchGetRow(params, function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        var isAllSuccess = true;
```

```
        var retryRequest = { tables: [] };
        for (var i = 0; i < data.tables.length; i++) {
            var faildRequest = { tableName: data.tables[i][0].tableName, primaryKey: [] }
;
            for (var j = 0; j < data.tables[i].length; j++) {
                if (!data.tables[i][j].isOk && null != data.tables[i][j].primaryKey) {
                    isAllSuccess = false;
                    var pks = [];
                    for (var k in data.tables[i][j].primaryKey) {
                        var name = data.tables[i][j].primaryKey[k].name;
                        var value = data.tables[i][j].primaryKey[k].value;
                        var kp = {};
                        kp[name] = value;
                        pks.push(kp);
                    }
                    faildRequest.primaryKey.push(pks);
                } else {
                    // get success data
                }
            }
            if (faildRequest.primaryKey.length > 0) {
                retryRequest.tables.push(faildRequest);
            }
        }
        if (!isAllSuccess && retryCount++ < maxRetryTimes) {
            batchGetRow(retryRequest);
        }
        console.log('success:', data);
    });
}
batchGetRow(params, maxRetryTimes);
```

详细代码请参见BatchGetRow@GitHub。

范围读（GetRange）

范围读接口用于读取一个主键范围内的数据。

范围读接口支持按照确定范围进行正序读取和逆序读取，可以设置要读取的行数。如果范围较大，已扫描的行数或者数据量超过一定限制，会停止扫描，并返回已获取的行和下一个主键信息。您可以根据返回的下一个主键信息，继续发起请求，获取范围内剩余的行。

GetRange操作可能在如下情况停止执行并返回数据。

- 扫描的行数据大小之和达到4 MB。
- 扫描的行数等于5000。
- 返回的行数等于最大返回行数。
- 当前剩余的预留读吞吐量已全部使用，余量不足以读取下一条数据。

> **说明** 表格存储表中的行默认是按照主键排序的，而主键是由全部主键列按照顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

- 接口

```
/**
 * 读取指定主键范围内的数据。
 */
getRange(params, callback)
```

- 参数

参数	说明
tableName	数据表名称。
direction	读取方向。 ○ 如果值为正序（FORWARD），则起始主键必须小于结束主键，返回的行按照主键由小到大的顺序进行排列。 ○ 如果值为逆序（BACKWARD），则起始主键必须大于结束主键，返回的行按照主键由大到小的顺序进行排列。 例如同一表中有两个主键A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。
inclusiveStartPrimaryKey exclusiveEndPrimaryKey	本次范围读的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由INF_MIN和INF_MAX类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中INF_MIN表示无限小，任何类型的值都比它大；INF_MAX表示无限大，任何类型的值都比它小。 ○ inclusiveStartPrimaryKey表示起始主键，如果该行存在，则返回结果中一定会包含此行。 ○ exclusiveEndPrimaryKey表示结束主键，无论该行是否存在，返回结果中都不会包含此行。 数据表中的行按主键从小到大排序，读取范围是一个左闭右开的区间，正序读取时，返回的是大于等于起始主键且小于结束主键的所有的行。
limit	数据的最大返回行数，此值必须大于 0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。此时可以通过返回结果中的nextStartPrimaryKey记录本次读取到的位置，用于下一次读取。

参数	说明
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ◦ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 ◦ 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。 ◦ 当columnsToGet和columnFilter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 ◦ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。

参数	说明
timeRange	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 ○ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 查询一个范围的数据，则需要设置startTime和endTime。startTime和endTime分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[startTime, endTime)。 ○ 如果查询特定版本号的数据，则需要设置specificTime。specificTime表示特定的时间戳。 specificTime和[startTime, endTime)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Long.MAX_VALUE。
columnFilter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当columnsToGet和columnFilter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
nextStartPrimaryKey	根据返回结果中的nextStartPrimaryKey判断数据是否全部读取。 ○ 当返回结果中nextStartPrimaryKey不为空时，可以使用此返回值作为下一次GetRange操作的起始点继续读取数据。 ○ 当返回结果中nextStartPrimaryKey为空时，表示读取范围内的数据全部返回。

- 示例

  按照范围读取数据。

```
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sampleTable",
  direction: TableStore.Direction.FORWARD,
  inclusiveStartPrimaryKey: [{ "gid": TableStore.INF_MIN }, { "uid": TableStore.INF_MIN }
],
  exclusiveEndPrimaryKey: [{ "gid": TableStore.INF_MAX }, { "uid": TableStore.INF_MAX }],
  limit: 50
};
client.getRange(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
//如果data.next_start_primary_key不为空，则继续读取。
  if (data.next_start_primary_key) {
  }
  console.log('success:', data);
});
```

详细代码请参见Get Range@Git Hub。

6.8. 多元索引

6.8.1. 创建多元索引

使用CreateSearchIndex接口在数据表上创建一个多元索引。一个数据表可以创建多个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。

参数

创建多元索引时，需要指定数据表名称（tableName）、多元索引名称（indexName）和索引的结构信息（schema），其中schema包含fieldSchemas（Index的所有字段的设置）、indexSetting（索引设置）和indexSort（索引预排序设置）。详细参数说明请参见下表。

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
fieldSchemas	fieldSchemas的列表，每个fieldSchema包含如下内容： • fieldName（必选）：创建多元索引的字段名，即列名，类型为String。 多元索引中的字段可以是主键列或者属性列。 • fieldType（必选）：字段类型，类型为TableStore.FieldType.XXX。更多信息，请参见字段。 • index（可选）：是否开启索引，类型为Boolean。 默认为true，表示对该列构建倒排索引或者空间索引；如果设置为false，则不会对该列构建索引。 • analyzer（可选）：分词器类型。当字段类型为Text时，可以设置此参数；如果不设置，则默认分词器类型为单字分词。关于分词的更多信息，请参见分词。 • enableSortAndAgg（可选）：是否开启排序与统计聚合功能，类型为Boolean。 只有enableSortAndAgg设置为true的字段才能进行排序。关于排序的更多信息，请参见排序和翻页。 注意 Nested类型的字段不支持开启排序与统计聚合功能，但是Nested类型内部的子列支持开启排序与统计聚合功能。 • store（可选）：是否在多元索引中附加存储该字段的值，类型为Boolean。 开启后，可以直接从多元索引中读取该字段的值，而不必反查数据表，可用于查询性能优化。 • isAnArray（可选）：是否为数组，类型为Boolean。 如果设置为true，则表示该列是一个数组，在写入时，必须按照JSON数组格式写入，例如["a","b","c"]。 由于Nested类型是一个数组，当fieldType为Nested类型时，无需设置此参数。 • fieldSchemas（可选）：当字段类型为Nested类型时，需要通过此参数设置嵌套文档中子列的索引类型，类型为fieldSchema的列表。
indexSetting	索引设置，包含routingFields设置。 routingFields（可选）：自定义路由字段。可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

参数	说明
indexSort	索引预排序设置，包含sorters设置。如果不设置，则默认按照主键排序。 说明 含有Nested类型的索引不支持indexSort，没有预排序。 sorters（必选）：索引的预排序方式，支持按照主键排序和字段值排序。关于排序的更多信息，请参见排序和翻页。 • PrimaryKeySort表示按照主键排序，包含如下设置： order：排序的顺序，可按升序或者降序排序，默认为升序（TableStore.SortOrder.SORT_ORDER_ASC）。 • FieldSort表示按照字段值排序，包含如下设置： 只有建立索引且开启排序与统计聚合功能的字段才能进行预排序。 ○ fieldName：排序的字段名。 ○ order：排序的顺序，可按照升序或者降序排序，默认为升序（TableStore.SortOrder.SORT_ORDER_ASC）。 ○ mode：当字段存在多个值时的排序方式。

示例

```
/**
 *创建一个多元索引，包含Col_Keyword、Col_Long、Col_Text、Col_Nested四列。
 *类型分别设置为字符串（Keyword）、整型（Long）、分词字符串（Text）以及嵌套文本（Nested）。
 */
client.createSearchIndex({
    tableName: TABLE_NAME, //设置数据表名称。
    indexName: INDEX_NAME, //设置多元索引名称。
    schema: {
        fieldSchemas: [
            {
                fieldName: "Col_Keyword",
                fieldType: TableStore.FieldType.KEYWORD, //设置字段名和字段类型。
                index: true, //设置开启索引。
                enableSortAndAgg: true, //设置开启排序与统计聚合功能。
                store: false,
                isAnArray: false
            },
            {
                fieldName: "Col_Long",
                fieldType: TableStore.FieldType.LONG,
                index: true,
                enableSortAndAgg: true,
                store: true,
                isAnArray: false
            },
            {
                fieldName: "Col_Text",
                fieldType: TableStore.FieldType.TEXT,
                index: true,
```

```
                enableSortAndAgg: false,
                store: true,
                isAnArray: false,
                analyzer: "single_word"
            },
            // {
            //     fieldName: "Col_Nested",
            //     fieldType: TableStore.FieldType.NESTED,
            //     index: false,
            //     enableSortAndAgg: false,
            //     store: false,
            //     fieldSchemas: [ //嵌套字段中设置子字段索引。
            //         {
            //             fieldName: "Sub_Col_KeyWord",
            //             fieldType: TableStore.FieldType.KEYWORD,
            //             index: true,
            //             enableSortAndAgg: true,
            //             store: false
            //         },
            //         {
            //             fieldName: "Sub_Col_Long",
            //             fieldType: TableStore.FieldType.LONG,
            //             index: true,
            //             enableSortAndAgg: true,
            //             store: false
            //         }
            //     ]
            // }
        ],
        indexSetting: { //索引的配置选项。
            "routingFields": ["Pk_Keyword"], //只支持将主键列设置为routingFields。
            "routingPartitionSize": null
        },
        indexSort: {//含有Nested类型的索引不支持indexSort，没有预排序。
            sorters: [
                // { //不设置indexSort时，默认为PrimaryKeySort（升序）排序。
                //     primaryKeySort: {
                //         order: TableStore.SortOrder.SORT_ORDER_ASC
                //     }
                // },
                {
                    fieldSort: {
                        fieldName: "Col_Keyword",
                        order: TableStore.SortOrder.SORT_ORDER_DESC //设置indexSort排序的顺序
。
                    }
                }
            ]
        }
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
```

```
    }
    console.log('success:', data);
});
```

6.8.2. 列出多元索引列表

创建多元索引后，使用ListSearchIndex接口可以获取当前实例下或某个数据表关联的所有多元索引的列表信息。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称，可以为空。 • 如果设置了数据表名称，则返回该数据表关联的所有多元索引的列表。 • 如果未设置数据表名称，则返回当前实例下所有多元索引的列表。

示例

```
client.listSearchIndex({
    tableName: TABLE_NAME //设置数据表名称。
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.3. 查询多元索引描述信息

创建多元索引后，使用DescribeSearchIndex接口可以查询多元索引的描述信息，包括多元索引的字段信息和索引配置等。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

示例

```
client.describeSearchIndex({
    tableName: TABLE_NAME, //设置数据表名称。
    indexName: INDEX_NAME  //设置多元索引名称。
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.4. 删除多元索引

使用DeleteSearchIndex接口可以删除指定数据表的一个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

示例

```
client.deleteSearchIndex({
    tableName: TABLE_NAME, //设置数据表名称。
    indexName: INDEX_NAME  //设置多元索引名称。
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

6.8.5. 精确查询

TermQuery采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。对于Text类型字段，只要分词后有词条可以精确匹配即可。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
queryType	设置查询类型为TableStore.QueryType.TERM_QUERY。
fieldName	要匹配的字段。
term	查询关键词，即要匹配的值。 该词不会被分词，会被当做完整词去匹配。 对于Text类型字段，只要分词后有词条可以精确匹配即可。例如某个Text类型的字段，值为“tablestore is cool”，如果分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
columnToGet	是否返回所有列，包含returnType和returnNames设置。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，可以通过returnNames指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"的数据。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.TERM_QUERY。
            queryType: TableStore.QueryType.TERM_QUERY,
            query: {
                fieldName: "Col_Keyword",
                term: "hangzhou"
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.6. 多词精确查询

类似于TermQuery，但是TermsQuery可以指定多个查询关键词，查询匹配这些词的数据。多个查询关键词中只要有一个词精确匹配，该行数据就会被返回，等价于SQL中的In。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
offset	本次查询的开始位置。

参数	说明
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
queryType	设置查询类型为TableStore.QueryType.TERMS_QUERY。
fieldName	要匹配的字段。
terms	多个查询关键词，即要匹配的值。 多个查询关键词中只要有一个词精确匹配，该行数据就会被返回。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
columnToGet	是否返回所有列，包含returnType和returnNames设置。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，可以通过returnNames指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 查询表中Col_Keyword列精确匹配"hangzhou"或"shanghai"的数据。
 * TermsQuery可以使用多个Term同时查询。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.TERMS_QUERY。
            queryType: TableStore.QueryType.TERMS_QUERY,
            query: {
                fieldName: "Col_Keyword",
                terms: ["hangzhou", "shanghai"]
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.7. 全匹配查询

MatchAllQuery可以匹配所有行，常用于查询表中数据总行数，或者随机返回几条数据。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
offset	本次查询的开始位置。

参数	说明
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
query	设置查询类型为TableStore.QueryType.MATCH_ALL_QUERY。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
columnToGet	是否返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，可以通过returnNames指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 通过MatchAllQuery查询表中数据的总行数。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: {
            queryType: TableStore.QueryType.MATCH_ALL_QUERY
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_SPECIFIED,
        returnNames: ["Col_1", "Col_2", "Col_3"]
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.8. 匹配查询

MatchQuery采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用MatchPhraseQuery实现高性能的模糊查询。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
queryType	设置查询类型为TableStore.QueryType.MATCH_QUERY。
fieldName	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被切分成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
operator	逻辑运算符。默认为OR，表示当分词后的多个词只要有部分匹配时，则行数据满足查询条件。 如果设置operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。

参数	说明
minimumShouldMatch	最小匹配个数。 只有当某一行数据的fieldName列的值中至少包括最小匹配个数的词时，才会返回该行数据。 说明 minimumShouldMatch需要与逻辑运算符OR配合使用。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
columnToGet	是否返回所有列，包含returnType和returnNames设置。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，可以通过returnNames指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 查询表中Col_Keyword列的值能够匹配"hangzhou"的数据，返回匹配到的总行数和一些匹配成功的行。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了取行数，但不需要具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为MatchQuery。
            queryType: TableStore.QueryType.MATCH_QUERY,
            query: {
                fieldName: "Col_Keyword", //设置要匹配的列。
                text: "hangzhou" //设置要匹配的值。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.9. 短语匹配查询

类似于MatchQuery，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。如果查询列的分词类型为模糊分词，则使用MatchPhraseQuery可以实现比WildcardQuery更快的模糊查询。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
queryType	设置查询类型为TableStore.QueryType.MATCH_PHRASE_QUERY。
fieldName	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“...，this is tablestore”、“this is a table”，但是无法匹配到“this table is ...”以及“is this a table”。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
columnToGet	是否返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，可以通过returnNames指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 查询表中Col_Text列的值能够匹配"hangzhou shanghai"的数据。
 * 匹配条件为短语匹配（要求短语完整的按照顺序匹配），返回匹配到的总行数和一些匹配成功的行。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了取行数，但不需要具体数据，可以设置limit=0，即不返回任意一行数据。
        query: {//设置查询类型为TableStore.QueryType.MATCH_PHRASE_QUERY。
            queryType: TableStore.QueryType.MATCH_PHRASE_QUERY,
            query: {
                fieldName: "Col_Text", //设置要匹配的列。
                text: "hangzhou shanghai" //设置要匹配的值。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.10. 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
queryType	设置查询类型为TableStore.QueryType.PREFIX_QUERY。
fieldName	要匹配的字段。
prefix	前缀值。 对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
columnToGet	是否返回所有列，包含returnType和returnNames设置。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，可以通过returnNames指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 查询表中Col_Keyword列前缀为"hang"的数据，例如"hangzhou"。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.PREFIX_QUERY。
            queryType: TableStore.QueryType.PREFIX_QUERY,
            query: {
                fieldName: "Col_Keyword",
                prefix: "hang" //设置前缀值，可匹配到"hangzhou"、"hangzhoushi"等。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.11. 范围查询

RangeQuery根据范围条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足范围条件即可。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。

参数	说明
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
queryType	设置查询类型为TableStore.QueryType.RANGE_QUERY。
fieldName	要匹配的字段。
rangeFrom	起始位置的值。
rangeTo	结束位置的值。
includeLower	结果中是否需要包括rangeFrom值，类型为Boolean。
includeUpper	结果中是否需要包括rangeTo值，类型为Boolean。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
columnToGet	是否返回所有列，包含returnType和returnNames设置。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，可以通过returnNames指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 查询表中Col_Long列[1, 10)的数据，结果按照Col_Long列的值逆序排序。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.RANGE_QUERY。
            queryType: TableStore.QueryType.RANGE_QUERY,
            query: {
                fieldName: "Col_Long",
                rangeFrom: 1,
                includeLower: true, //包括下边界（即大于等于1）。
                rangeTo: 10,
                includeUpper: false //不包括上边界（即小于10）。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.12. 通配符查询

通配符查询中，要匹配的值可以是一个带有通配符的字符串，目前支持星号（*）和问号（?）两种通配符。要匹配的值中可以用星号（*）代表任意字符序列，或者用问号（?）代表任意单个字符，且支持以星号（*）或问号（?）开头。例如查询“table*e”，可以匹配到“tablestore”。

如果查询的模式为*word*，则您可以使用性能更好的模糊查询，具体实现方法如下：

1. 创建多元索引时，设置列为Text类型且设置分词类型为模糊分词。
2. 使用多元索引查询数据时，使用MatchPhraseQuery且设置查询词为word。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	描述
tableName	数据表名称。
indexName	多元索引名称。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。
queryType	设置查询类型为TableStore.QueryType.WILDCARD_QUERY。
fieldName	列名称。
value	带有通配符的字符串，字符串长度不能超过20个字符。
getTotalCount	是否返回匹配的总行数，默认为false，表示不返回表中数据的总行数。 设置getTotalCount为true后会影响查询性能。
columnToGet	是否返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_SPECIFIED时，需要指定返回的列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_ALL时，表示返回所有列。 • 当设置returnType为TableStore.ColumnReturnType.RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
/**
 * 使用通配符查询，查询表中Col_Keyword列的值匹配"table*e"的数据。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只是为了获取匹配行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.WILDCARD_QUERY。
            queryType: TableStore.QueryType.WILDCARD_QUERY,
            query: {
                fieldName: "Col_Keyword",
                value: "table*e" //wildcardQuery支持通配符。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回表中数据的总行数。
    },
    columnToGet: { //返回列设置，可设置为RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）、RETURN_NONE（不返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.13. 地理位置查询

地理位置查询包括地理距离查询（GeoDistanceQuery）、地理长方形范围查询（GeoBoundingBoxQuery）和地理多边形范围查询（GeoPolygonQuery）三种方式。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

地理距离查询（GeoDistanceQuery）

GeoDistanceQuery根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，满足查询条件。

- 参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
query	多元索引的查询语句。设置查询类型为TableStore.QueryType.GEO_DISTANCE_QUERY。
fieldName	列名，类型为Geopoint。
centerPoint	中心地理坐标点，是一个经纬度值。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。
distance	距离中心点的距离，类型为Double。单位为米。

- 示例

  查询表中Col_GeoPoint列的值距离中心点不超过一定距离的数据。

```
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.GEO_DISTANCE_QUERY。
            queryType: TableStore.QueryType.GEO_DISTANCE_QUERY,
            query: {
                fieldName: "Col_GeoPoint",
                centerPoint: "1,1", //设置中心点。
                distance: 10000 //设置条件为到中心点的距离不超过10000米。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

地理长方形范围查询（GeoBoundingBoxQuery）

GeoBoundingBoxQuery根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时满足查询条件。

- 参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
query	多元索引的查询语句。设置查询类型为TableStore.QueryType.GEO_BOUNDING_BOX_QUERY。
fieldName	列名，类型为Geopoint。
topLeft	长方形框的左上角的坐标。
bottomRight	长方形框的右下角的坐标，通过左上角和右下角就可以确定一个唯一的长方形。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。

- 示例

  Col_GeoPoint是Geopoint类型，查询表中Col_GeoPoint列的值在左上角为"10,0"，右下角为"0,10"的长方形范围内的数据。

```
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.GEO_BOUNDING_BOX_QUERY。
            queryType: TableStore.QueryType.GEO_BOUNDING_BOX_QUERY,
            query: {
                fieldName: "Col_GeoPoint", //设置列名。
                topLeft: "10,0", //设置长方形左上角。
                bottomRight: "0,10" //设置长方形右下角。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

地理多边形范围查询（GeoPolygonQuery）

GeoPolygonQuery根据一个多边形范围条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时满足查询条件。

- 参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
query	多元索引的查询语句。设置查询类型为TableStore.QueryType.GEO_POLYGON_QUERY。
fieldName	列名，类型为Geopoint。
points	组成多边形的距离坐标点。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。

- 示例

查询表中Col_GeoPoint列的值在一个给定多边形范围内的数据。

```
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.GEO_POLYGON_QUERY。
            queryType: TableStore.QueryType.GEO_POLYGON_QUERY,
            query: {
                fieldName: "Col_GeoPoint",
                points: ["0,0","5,5","5,0"] //设置多边形的顶点。
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.14. 多条件组合查询

BoolQuery查询条件包含一个或者多个子查询条件，根据子查询条件来判断一行数据是否满足查询条件。每个子查询条件可以是任意一种Query类型，包括BoolQuery。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
mustQueries	多个Query列表，行数据必须满足所有的子查询条件才算匹配，等价于And操作符。

参数	说明
mustNotQueries	多个Query列表，行数据必须不能满足任何的子查询条件才算匹配，等价于Not操作符。
filterQueries	多个Query列表，行数据必须满足所有的子filter才算匹配，filter类似于query，区别是filter不会根据满足的filter个数进行相关性算分。
shouldQueries	多个Query列表，可以满足，也可以不满足，等价于Or操作符。 行数据应该至少满足shouldQueries子查询条件的最小匹配个数才算匹配。 如果满足的shouldQueries子查询条件个数越多，则整体的相关性分数更高。
minimumShouldMatch	shouldQueries的最小匹配个数。当同级没有其他Query，只有shouldQueries时，默认值为1；当同级已有其他Query，例如mustQueries、mustNotQueries和filterQueries时，默认值为0。

示例

通过构造一个BoolQuery进行多条件组合查询。

```
var client = require('../client');
var TableStore = require('../../index.js');
var Long = TableStore.Long;
/**
 * 使用多条件组合查询实现(col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6))。逻辑如下:
 * boolQuery1 = rangeQuery(col2<4) or rangeQuery(col3<5)
 * boolQuery2 = termQuery(col3=5) or (col3=6)
 * boolQuery3 = termQuery(col2=4) and boolquery2
 * boolQuery4 = boolQuery1 or boolQuery3
 */
client.search({
    tableName: "sampleTable",
    indexName: "sampleSearchIndex",
    searchQuery: {
        offset: 0, //查询偏移量。
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        getTotalCount: false, //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回。
        query: { //构造boolQuery4，设置查询条件为至少满足boolQuery1和boolQuery3中的一个条件。
            queryType: TableStore.QueryType.BOOL_QUERY,
            query: {
                shouldQueries: [ //可选mustQueries、shouldQueries和mustNotQueries。
                    { //构造boolQuery1，设置查询条件为至少满足"查询条件一"和"查询条件二"中的一个条件。
                        queryType: TableStore.QueryType.BOOL_QUERY,
                        query: {
                            //shouldQueries，查询条件为col2的列值小于4或者col3的列值小于5。
                            shouldQueries:[
                                {
                                    //查询条件一：rangeQuery，col2的列值小于4。
                                    queryType: TableStore.QueryType.RANGE_QUERY,
                                    query:{
                                        fieldName: "col2",
```

```
                                    rangeTo: 4
                                }
                            },
                            {
                                //查询条件二：rangeQuery，col3的列值小于5。
queryType: TableStore.QueryType.RANGE_QUERY,
                                query:{
                                    fieldName: "col3",
                                    rangeTo: 5
                                }
                            }
                        ],
                        minimumShouldMatch:1
                    }
                },
                { //构造boolQuery3，设置查询条件为必须同时满足"查询条件三"和boolQuery2。
                    queryType: TableStore.QueryType.BOOL_QUERY,
                    query: {
                        mustQueries: [
                            //mustQueries，查询条件为col2的列值等于4，且col3的列值等于5或者6
。
                            {
                                //查询条件三：termQuery，col2的列值等于4。
                                queryType:TableStore.QueryType.TERM_QUERY,
                                query: {
                                    fieldName : "col2",
                                    term: 4
                                }
                            },
                            { //构造boolQuery2：设置查询条件为至少满足"查询条件四"和"查询条件
五"中的一个条件。
                                queryType: TableStore.QueryType.BOOL_QUERY,
                                query: {
                  TERM                  //shouldQueries，查询条件为col3的列值等于5或者6。
                                    shouldQueries:[
                                        {
                                            //查询条件四：termQuery，col3的列值等于5。
                                            queryType: TableStore.QueryType.TERM_QUERY,
                                            query:{
                                                fieldName:"col3",
                                                term: 5
                                            }
                                        },
                                        {
                                            //查询条件五：termQuery，col3的列值等于6。
queryType: TableStore.QueryType.TERM_QUERY,
                                            query:{
                                                fieldName:"col3",
                                                term: 6
                                            }
                                        }
                                    ],
                                    minimumShouldMatch:1
                                }
```

```
                                }
                            ]
                        }
                    }
                ],
                minimumShouldMatch: 1 //仅shouldQueries时有效，至少满足的条件个数。
            }
        },
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED表示自定义返回列，RETURN_ALL表示返回所有列，RETURN_NONE表示不返回。
        returnType: TableStore.ColumnReturnType.RETURN_SPECIFIED,
        returnNames: ["col2", "col3", "col4"]
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.15. 嵌套类型查询

NestedQuery用于查询嵌套类型字段中子行的数据。嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型字段的路径和一个子查询，其中子查询可以是任意Query类型。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
tableName	数据表名称。
indexName	多元索引名称。
path	路径名，嵌套类型的列的树状路径。例如news.title表示嵌套类型的news列中的title子列。
query	嵌套类型的列中子列上的查询，子列上的查询可以是任意Query类型。
scoreMode	当列存在多个值时基于哪个值计算分数。

示例

查询表中Col_Nested.Sub_Col_Keyword列值为"开心"的数据。

```
/**
 * 嵌套数据示例Col_Nested: '[{Sub_Col_Keyword: "开心"},{Sub_Col_Keyword: "晴天"}]'
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10, //如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
        query: { //设置查询类型为TableStore.QueryType.NESTED_QUERY。
            queryType: TableStore.QueryType.NESTED_QUERY,
            query: {
                path: "Col_Nested",
                query: {
                    queryType: TableStore.QueryType.TERM_QUERY,
                    query: {
                        fieldName: "Col_Nested.Sub_Col_Keyword",
                        term: "开心"
                    }
                },
            }
        },
        getTotalCount: true //结果中的TotalCount可以表示表中数据的总行数，默认为false，表示不返回
。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED（自定义）、RETURN_ALL（所有列）和RETURN_NONE（不
返回）。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

6.8.16. 排序和翻页

您可以在创建多元索引时指定索引预排序和在使用多元索引查询数据时指定排序方式，以及在获取返回结果时使用limit和offset或者使用token进行翻页。

索引预排序

多元索引默认按照设置的索引预排序（IndexSort）方式进行排序，使用多元索引查询数据时，IndexSort决定了数据的默认返回顺序。

在创建多元索引时，您可以自定义IndexSort，如果未自定义IndexSort，则IndexSort默认为主键排序。

> **注意** 含有Nested类型字段的多元索引不支持索引预排序。

查询时指定排序方式

只有enableSortAndAgg设置为true的字段才能进行排序。

在每次查询时，可以指定排序方式，多元索引支持如下四种排序方式（Sorter）。您也可以使用多个Sorter，实现先按照某种方式排序，再按照另一种方式排序的需求。

- ScoreSort

  按照查询结果的相关性（BM25算法）分数进行排序，适用于有相关性的场景，例如全文检索等。

  > **注意** 如果需要按照相关性打分进行排序，必须手动设置ScoreSort，否则会按照索引设置的IndexSort进行排序。

```
sort: {
    sorters: [
        {
            scoreSort: {
                order: TableStore.SortOrder.SORT_ORDER_ASC
            }
        }
    ]
}
```

- PrimaryKeySort

  按照主键进行排序。

```
sort: {
    sorters: [
        {
            primaryKeySort: {
                order: TableStore.SortOrder.SORT_ORDER_DESC //逆序。
                //order: TableStore.SortOrder.SORT_ORDER_ASC //正序。
            }
        }
    ]
}
```

- FieldSort

  按照某列的值进行排序。

```
sort: {
    sorters: [
        {
            fieldSort: {
                fieldName: "Col_Keyword",
                order: TableStore.SortOrder.SORT_ORDER_DESC
            }
        }
    ]
}
```

  先按照某列的值进行排序，再按照另一列的值进行排序。

```
sort: {
    sorters: [
        {
            fieldSort: {
                fieldName: "Col_Keyword",
                order: TableStore.SortOrder.SORT_ORDER_DESC
            }
        },
        {
            fieldSort: {
                fieldName: "Col_Long",
                order: TableStore.SortOrder.SORT_ORDER_DESC
            }
        }
    ]
}
```

- GeoDistanceSort

  根据地理点距离进行排序。

```
sort: {
    sorters: [
        {
            geoDistanceSort: {
                fieldName: "Col_Geo_Point",
                points: ["0,0"],//设置中心点。
                order: TableStore.SortOrder.SORT_ORDER_ASC //距离中心点正序返回。
            }
        }
    ]
}
```

关于代码的更多信息，请参见Search。

翻页方式

在获取返回结果时，可以使用limit和offset或者使用token进行翻页。

- 使用limit和offset进行翻页

  当需要获取的返回结果行数小于10000行时，可以使用limit和offset进行翻页，即limit+offset<=10000，其中limit的最大值为100。

  > **说明** 如果需要提高limit的上限，请参见如何将多元索引Search接口查询数据的limit提高到1000。

  如果使用此方式进行翻页时未设置limit和offset，则limit的默认值为10，offset的默认值为0。

```
/**
 * 通过limit+offset进行翻页，直接展示第10页（第90~99条）数据。
 */
client.search({
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 90,
        limit: 10,
        query: {
            queryType: TableStore.QueryType.MATCH_ALL_QUERY
        }
        getTotalCount: true //结果中的TotalCount表示数据的总行数，默认为false，表示不返回数据的总行数。
    },
    columnToGet: { //返回列设置RETURN_SPECIFIED表示自定义返回列，RETURN_ALL表示返回所有列，RETURN_NONE表示不返回。
        returnType: TableStore.ColumnReturnType.RETURN_ALL
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

- 使用token进行翻页

  由于使用token进行翻页时翻页深度无限制，当需要进行深度翻页时，推荐使用token进行翻页。

  当符合查询条件的数据未读取完时，服务端会返回nextToken，此时可以使用nextToken继续读取后面的数据。

  使用token进行翻页时默认只能向后翻页。由于在一次查询的翻页过程中token长期有效，您可以通过缓存并使用之前的token实现向前翻页。

  使用token翻页后的排序方式和上一次请求的一致，无论是系统默认使用IndexSort还是自定义排序，因此设置了token不能再设置Sort。另外使用token后不能设置offset，只能依次往后读取，即无法跳页。

  > **注意** 由于含有Nested类型字段的多元索引不支持索引预排序，如果使用含有Nested类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回nextToken。

```
/**
 * 使用token翻页示例（同步+异步）。
 */
var params = {
    tableName: TABLE_NAME,
    indexName: INDEX_NAME,
    searchQuery: {
        offset: 0,
        limit: 10,
        token: null,//获取nextToken作为下一页起点（数据类型为字节流）。
        query: {
```

```
            queryType: TableStore.QueryType.MATCH_ALL_QUERY
        },
        getTotalCount: true
    },
    columnToGet: {
        returnType: TableStore.ColumnReturnType.RETURN_SPECIFIED,
        returnNames: ["pic_tag", "pic_description", "time_stemp", "pos"]
    }
};
/**
 * 使用token翻页示例（同步）。
 */
(async () => { //同步示例代码。
  try {
    var data = await client.search(params);
    console.log('success:', JSON.stringify(data, null, 2));
    while (data.nextToken && data.nextToken.length) { //当存在nextToken时，表示还有未读取的数据。
      //token持久化。
      //1）nextToken为buffer，需转换为base64字符串后做持久化。
      //2）持久化的base64字符串，可转换为buffer作为参数重新使用。
      var nextToken = data.nextToken.toString("base64");
      var token =  Buffer.from(nextToken, "base64");
      params.searchQuery.token = token;//翻页更新token值。
      data = await client.search(params);
      console.log('token success:', JSON.stringify(data, null, 2));
    }
  } catch (error) {
      console.log(error);
  }
})()
/**
 * 使用token翻页示例（异步）。
 */
client.search(params, function (err, data) {
    console.log('success:', JSON.stringify(data, null, 2));
    if (data.nextToken && data.nextToken.length) {
        //token持久化。
        //1）nextToken为buffer，需转换为base64字符串后做持久化。
        //2）持久化的base64字符串，可转换为buffer作为参数重新使用。
        var nextToken = data.nextToken.toString("base64");
        var token =  Buffer.from(nextToken, "base64");
        params.searchQuery.token = token;//翻页更新token值。
        client.search(params, function (err, data) {
            console.log('token success:', JSON.stringify(data, null, 2));
        });
    }
});
```

6.9. 二级索引

6.9.1. 全局二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
mainTableName	数据表名称。

参数	说明
indexMeta	索引表的结构信息，包括如下内容： ○ name：索引表名称。 ○ primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ definedColumn：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ includeBaseData：索引表中是否包含数据表中已存在的数据。 当设置includeBaseData为true时，表示包含存量数据；设置includeBaseData为false时，表示不包含存量数据。 ○ indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 ○ indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ■ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ■ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。

- 示例

```
var client = require('./client');
var TableStore = require('../index.js');
client.createIndex({
    mainTableName: "sdkGlobalIndexTest",
    indexMeta: {
        name: "sdkGlobalIndex",
        primaryKey: ["col1"],
        definedColumn: ["col2"],
        includeBaseData: false,
        indexUpdateMode: TableStore.IndexUpdateMode.IUM_ASYNC_INDEX,//索引更新模式默认为异步更新（IUM_ASYNC_INDEX）。
        indexType: TableStore.IndexType.IT_GLOBAL_INDEX,//索引类型默认为全局二级索引（IT_GLOBAL_INDEX）。
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  - 参数

    使用GetRow接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "index1",
  primaryKey: [ {'pk2': Long.fromNumber(2)}, {'pk1': Long.fromNumber(1)}]
};
client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sdkIndex1", //假定pk2是索引表的第一个主键列，pk1是索引表的第二个主键列。
  direction: TableStore.Direction.FORWARD,
  maxVersions: 10,
  inclusiveStartPrimaryKey: [{ "pk2": TableStore.INF_MIN }, { "pk1": TableStore.INF_MIN
}],
  exclusiveEndPrimaryKey: [{ "pk2": TableStore.INF_MAX }, { "pk1": TableStore.INF_MAX }
],
  limit: 2
};
var resultRows = []
var getRange = function () {
  client.getRange(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    resultRows = resultRows.concat(data.rows)
    //如果data.next_start_primary_key不为空，则继续读取。
    if (data.nextStartPrimaryKey) {
      params.inclusiveStartPrimaryKey = [
        { "pk2": data.nextStartPrimaryKey[0].value },
        { "pk1": data.nextStartPrimaryKey[1].value }
      ];
      getRange()
    } else {
      console.log(JSON.stringify(resultRows));
    }
  });
}
getRange()
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
mainTableName	数据表名称。
indexName	索引表名称。

- 示例

```
var client = require('./client');
client.dropIndex({
  mainTableName: "sdkGlobalTest",
  indexName: "sdkIndex1"
}, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

6.9.2. 本地二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client。具体操作，请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表。具体操作，请参见创建数据表。

- 参数

参数	说明
mainTableName	数据表名称。

参数	说明
indexMeta	索引表的结构信息，包括如下内容： ○ name：索引表名称。 ○ primaryKey：索引表的索引列，索引列为数据表主键和预定义列的组合。 使用本地二级索引时，索引表的第一个主键列必须与数据表的第一个主键列相同。 ○ definedColumn：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ includeBaseData：索引表中是否包含数据表中已存在的数据。 当设置includeBaseData为true时，表示包含存量数据；设置includeBaseData为false时，表示不包含存量数据。 ○ indexType：索引类型。可选值包括IT_GLOBAL_INDEX和IT_LOCAL_INDEX。 ■ 当不设置indexType或者设置indexType为IT_GLOBAL_INDEX时，表示使用全局二级索引。 使用全局索引时，表格存储以异步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，正常情况下同步延迟达到毫秒级别。 ■ 当设置indexType为IT_LOCAL_INDEX时，表示使用本地二级索引。 使用本地二级索引时，表格存储以同步方式将数据表中被索引的列和主键列的数据自动同步到索引表中，当数据写入数据表后，即可从索引表中查询到数据。 ○ indexUpdateMode：索引更新模式。可选值包括IUM_ASYNC_INDEX和IUM_SYNC_INDEX。 ■ 当不设置indexUpdateMode或者设置indexUpdateMode为IUM_ASYNC_INDEX时，表示异步更新。 使用全局二级索引时，索引更新模式必须设置为异步更新（IUM_ASYNC_INDEX）。 ■ 当设置indexUpdateMode为IUM_SYNC_INDEX时，表示同步更新。 使用本地二级索引时，索引更新模式必须设置为同步更新（IUM_SYNC_INDEX）。

- 示例

```
var client = require('./client');
var TableStore = require('../index.js');
client.createIndex({
    mainTableName: "sdkLocalIndexTest",
    indexMeta: {
        name: "sdkLocalIndex",
        primaryKey: ["pk1","col1"],//为索引表添加主键列。索引表的第一列主键必须与数据表的第一列主键相同。
        definedColumn: ["col2"],
        includeBaseData: false,
        indexUpdateMode: TableStore.IndexUpdateMode.IUM_SYNC_INDEX,//设置索引更新模式为同步更新（IUM_SYNC_INDEX）。当索引类型为本地二级索引时，索引更新模式必须为同步更新。
        indexType: TableStore.IndexType.IT_LOCAL_INDEX,//设置索引类型为本地二级索引（IT_LOCAL_INDEX）。
    }
}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', JSON.stringify(data, null, 2));
});
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  更多信息，请参见单行数据操作。

  - 参数

    使用GetRow接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 示例

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sdkLocalIndex",
  primaryKey: [ {'pk1': Long.fromNumber(1)},{'col1': Long.fromNumber(2)}, {'pk2': Long.
fromNumber(2)}]
};
client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

- 范围读索引表中数据

  更多信息，请参见多行数据操作。

  - 参数

    使用GetRange接口读取索引表中数据时有如下注意事项：

    - tableName需要设置为索引表名称。
    - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

  - 示例

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sdkLocalIndex",
  direction: TableStore.Direction.FORWARD,
  maxVersions: 10,
  inclusiveStartPrimaryKey: [{ "pk1": TableStore.INF_MIN }, { "col1": TableStore.INF_MI
N }, { "pk2": TableStore.INF_MIN }],
  exclusiveEndPrimaryKey: [{ "pk1": TableStore.INF_MIN }, { "col1": TableStore.INF_MIN
}, { "pk2": TableStore.INF_MAX }],
  limit: 2
};
var resultRows = []
var getRange = function () {
  client.getRange(params, function (err, data) {
    if (err) {
      console.log('error:', err);
      return;
    }
    resultRows = resultRows.concat(data.rows)
    //如果data.next_start_primary_key不为空，则继续读取。
    if (data.nextStartPrimaryKey) {
      params.inclusiveStartPrimaryKey = [
        { "pk1": data.nextStartPrimaryKey[0].value },
        { "col1": data.nextStartPrimaryKey[1].value }
        { "pk2": data.nextStartPrimaryKey[2].value }
      ];
      getRange()
    } else {
      console.log(JSON.stringify(resultRows));
    }
  });
}
getRange()
```

删除索引表（DeleteIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
mainTableName	数据表名称。
indexName	索引表名称。

- 示例

```
var client = require('./client');
client.dropIndex({
  mainTableName: "sdkGlobalTest",
  indexName: "sdkIndex1"
}, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', JSON.stringify(data, null, 2));
});
```

6.10. 错误处理

本文介绍表格存储Node.js SDK的错误处理方式和重试策略。

方式

表格存储Node.js SDK目前采用“异常”的方式处理错误。如果调用接口没有抛出异常，则说明操作成功，否则失败。

> **说明** 批量相关接口，例如BatchGetRow和BatchWriteRow不仅需要判断是否有异常，还需要检查每行的状态是否成功，只有全部成功后才能保证整个接口调用是成功的。

异常

表格存储Node.js SDK中所有的错误均经过了统一的处理，最终会返回到callback方法的err参数中，所以在获取返回数据前，需要检查err参数是否有值。如果是表格存储服务端报错，会返回requestId。requestId用于唯一标识该次请求的UUID。当您无法解决问题时，记录此requestId并提交工单。

重试

SDK中出现错误时会自动重试。默认策略是最大重试次数为20，最大重试间隔为3000毫秒。对流控类错误以及读操作相关的服务端内部错误进行的重试，请参见tablestore/lib/retry.js。

7..NET SDK

7.1. 前言

本文介绍表格存储.NET SDK的使用。

前提条件

- 已开通表格存储。具体操作，请参见开通表格存储服务。
- 已创建AccessKey。具体操作，请参见获取AccessKey。

SDK下载

- 从NuGet下载SDK安装包，具体下载路径请参见.NET SDK 4.1.4。
- 从Github下载源码，具体下载路径请参见GitHub。

历史版本迭代详情请参见.NET SDK历史迭代版本。

兼容性

对于3.x.x系列的SDK兼容。

对于2.x.x系列的SDK不兼容处如下。

- 接口部分不兼容：删除Condition.IGNORE、Condition.EXPECT_EXIST和Condition.EXPECT_NOT_EXIST。
- DLL文件名称由Aliyun.dll变更为Aliyun.TableStore.dll。

版本

当前最新版本为4.1.4。

7.2. 安装

本文介绍如何安装表格存储.NET SDK。

版本依赖

Windows

- 适用于.NET 4.x版本。
- 适用于Visual Studio 2010及以上版本。

Windows环境安装

- NuGet安装
  i. 在Visual Studio中新建或者打开已有的项目后，选择**工具 > NuGet程序包管理器 > 管理解决方案的NuGet程序包**。

  > **说明** 如果Visual Studio未安装NuGet，请下载并安装NuGet，具体下载路径请参见NuGet。

  ii. 搜索aliyun.tablestore，在结果中找到Aliyun.TableStore.SDK。
  iii. 选择最新版本，单击**安装**。

安装成功后，表格存储.NET SDK会添加到项目应用中。

- 项目引入方式安装
  i. 使用Git从GitHub下载源码，具体下载路径请参见GitHub。

  > **说明** 如果未安装Git，请下载并安装Git，具体下载路径请参见Git。

  ii. 在Visual Studio中右键选择**解决方案**，在弹出的菜单中选择**添加 > 现有项目** 。
  iii. 在弹出的对话框中选择*aliyun-tablestore-sdk.csproj*文件，单击**打开**。
  iv. 右键选择**您的项目**，选择**引用 > 添加引用**，在弹出的对话框选择**项目**选项卡，并选中aliyun-tablestore-sdk项目。
  v. 单击**确定**。

7.3. 初始化

OTSClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、读写单行数据、读写多行数据等。

确定Endpoint

Endpoint是阿里云表格存储服务在各个区域的域名地址，您可以通过以下方式查询Endpoint：

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

> **说明** 关于Endpoint的更多信息，请参见服务地址。

配置密钥

要接入阿里云的表格存储服务，您需要拥有一个有效的访问密钥进行签名认证。目前支持下面三种方式：

- 阿里云账号的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 在阿里云官网注册阿里云账号。
  ii. 创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。
- 被授予访问表格存储权限的RAM用户的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 使用阿里云账号前往访问控制RAM，创建一个新的RAM用户或者使用已经存在的RAM用户。
  ii. 使用阿里云账号授予RAM用户访问表格存储的权限。
  iii. RAM用户被授权后，即可使用自己的AccessKey ID和AccessKey Secret访问。
- 从STS获取的临时访问凭证。获取步骤如下：
  i. 应用的服务器通过访问RAM/STS服务，获取一个临时的AccessKey ID、AccessKey Secret和SecurityToken发送给使用方。
  ii. 使用方使用上述临时密钥访问表格存储服务。

初始化对接

获取到AccessKey ID和AccessKey Secret后，您可以按照下如下步骤进行初始化对接。

- 接口

```
    /// <summary>
    /// OTSClient的构造函数。
    /// </summary>
    /// <param name="endPoint">OTS服务的地址（例如'https://instance.cn-hangzhou.ots.aliyun.com:80'），必须以'https://'开头。</param>
    /// <param name="accessKeyID">OTS的Access Key ID，通过官方网站申请。</param>
    /// <param name="accessKeySecret">OTS的Access Key Secret，通过官方网站申请。</param>
    /// <param name="instanceName">OTS实例名，通过官方网站控制台创建。</param>
public OTSClient(string endPoint, string accessKeyID, string accessKeySecret, string instanceName);
    /// <summary>
    /// 通过客户端配置OTSClientConfig的实例来创建OTSClient实例。
    /// </summary>
    /// <param name="config">客户端配置实例</param>
    public OTSClient(OTSClientConfig config);
```

- 示例

> 说明
> - OTSClientConfig中还可以设置ConnectionLimit。如果不设置，默认值为300。
> - OTSClientConfig中的OTSDebugLogHandler和OTSErrorLogHandler控制日志行为，可以自定义。
> - OTSClientConfig中的RetryPolicy控制重试逻辑，目前有默认重试策略，也可以自定义重试策略。

```
    // 构造一个OTSClientConfig对象。
    var config = new OTSClientConfig(Endpoint, AccessKeyId, AccessKeySecret, InstanceName);
    // 禁止输出日志，默认是打开的。
    config.OTSDebugLogHandler = null;
    config.OTSErrorLogHandler = null;
    // 使用OTSClientConfig创建一个OtsClient对象。
    var otsClient = new OTSClient(config);
    // 使用otsClient插入或者查询数据。
```

多线程

- 支持多线程。
- 使用多线程时，建议共用一个OTSClient对象。

7.4. 表

7.4.1. 创建数据表

使用CreateTable接口创建数据表时，需要指定数据表的结构信息和配置信息，高性能实例中的数据表还可以根据需要设置预留读/写吞吐量。创建数据表的同时支持创建一个或者多个索引表。

> **说明**
> - 创建数据表后需要几秒钟进行加载，在此期间对该数据表的读/写数据操作均会失败。请等待数据表加载完毕后再进行数据操作。
> - 创建数据表时必须指定数据表的主键。主键包含1个~4个主键列，每一个主键列都有名称和类型。

前提条件

- 已通过控制台创建实例，详情请参见创建实例。
- 已初始化Client，详情请参见初始化。

接口

```
/// <summary>
 /// 根据表信息（包含数据表名称、主键的定义和预留读写吞吐量）创建数据表。
 /// </summary>
 /// <param name="request">请求参数</param>
 /// <returns>CreateTable的返回，此返回实例是空的，不包含具体信息。
 /// </returns>
        public CreateTableResponse CreateTable(CreateTableRequest request);
        /// <summary>
        /// CreateTable的异步形式。
        /// </summary>
        public Task<CreateTableResponse> CreateTableAsync(CreateTableRequest request);
```

参数

参数	说明
tableMeta	数据表的结构信息，包括如下内容： • tableName：数据表名称。 • primaryKeySchema：数据表的主键定义，详情请参见主键和属性。 **说明** 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。 ◦ 表格存储可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体参见表格存储数据模型和查询操作。 ◦ 第一列主键作为分区键。分区键相同的数据会存放在同一个分区内，所以相同分区键下最好不要超过10 GB以上数据，否则会导致单分区过大，无法分裂。另外，数据的读/写访问最好在不同的分区键上均匀分布，有利于负载均衡。 • definedColumnSchema：预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。

参数	说明
tableOptions	数据表的配置信息，详情请参见数据版本和生命周期。 配置信息包括如下内容： • timeToLive：数据生命周期，即数据的过期时间。当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 创建数据表时，如果希望数据永不过期，可以设置数据生命周期为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 单位为秒。 **说明** 如果需要使用索引，则数据生命周期必须设置为-1（数据永不过期）。 • maxVersions：最大版本数，即属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。 **说明** 如果需要使用索引，则最大版本数必须设置为1。 • DeviationCellVersionInSec：有效版本偏差，即写入数据的时间戳与系统当前时间的偏差允许最大值。只有当写入数据所有列的版本号与写入时时间的差值在数据有效版本偏差范围内，数据才能成功写入。 属性列的有效版本范围为**[数据写入时间-有效版本偏差，数据写入时间+有效版本偏差)**。 创建数据表时，如果未设置有效版本偏差，系统会使用默认值86400；创建数据表后，可以通过UpdateTable接口动态修改有效版本偏差。 单位为秒。
reservedThroughput	为数据表配置预留读吞吐量或预留写吞吐量。 容量型实例中的数据表的预留读/写吞吐量只能设置为0，不允许预留。 默认值为0，即完全按量计费。 单位为CU。 • 当预留读吞吐量或预留写吞吐量大于0时，表格存储会根据配置为数据表预留相应资源，且数据表创建成功后，将会立即按照预留吞吐量开始计费，超出预留的部分进行按量计费。详情请参见计费概述。 • 当预留读吞吐量或预留写吞吐量设置为0时，表格存储不会为数据表预留相应资源。

参数	说明
indexMetas	索引表的结构信息，每个indexMeta都包括如下内容： • indexName：索引表名称。 • primaryKey：索引表的索引列，索引列为数据表主键和预定义列的任意组合。 • definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 • indexUpdateMode：索引表更新模式，当前只支持IUM_ASYNC_INDEX。 • indexType：索引表类型，当前只支持IT_GLOBAL_INDEX。

示例

- 创建数据表（不带索引）

  创建一个有2个主键列，预留读/写吞吐量为(0, 0)的数据表。

```
//创建主键列的schema，包括PK的个数、名称和类型。
//第一个PK列为整数，名称是pk0，这个同时也是分区键。
//第二个PK列为字符串，名称是pk1。
var primaryKeySchema = new PrimaryKeySchema();
primaryKeySchema.Add("pk0", ColumnValueType.Integer);
primaryKeySchema.Add("pk1", ColumnValueType.String);
//通过表名和主键列的schema创建一个tableMeta。
var tableMeta = new TableMeta("SampleTable", primaryKeySchema);
//设置预留读吞吐量为0，预留写吞吐量为0。
var reservedThroughput = new CapacityUnit(0, 0);
try
{
    //构造CreateTableRequest对象。
    var request = new CreateTableRequest(tableMeta, reservedThroughput);
    //调用client的CreateTable接口，如果没有抛出异常，则说明执行成功。
    otsClient.CreateTable(request);
    Console.WriteLine("Create table succeeded.");
}
//如果抛出异常，则说明失败，处理异常。
catch (Exception ex)
{
    Console.WriteLine("Create table failed, exception:{0}", ex.Message);
}
```

  详细代码请参见CreateTable@GitHub。

- 创建数据表（带索引）

```
        public static void CreateTableWithGlobalIndex()
        {
            //创建数据表，两列主键为Pk1、Pk2，预定义列为Col1、Col2。
            //创建索引表，索引表中Col1放Pk0。
            OTSClient otsClient = Config.GetClient();
            Console.WriteLine("Start create table with globalIndex...");
            PrimaryKeySchema primaryKeySchema = new PrimaryKeySchema
                {
                    { Pk1, ColumnValueType.String },
                    { Pk2, ColumnValueType.String }
                };
            TableMeta tableMeta = new TableMeta(TableName, primaryKeySchema);
            tableMeta.DefinedColumnSchema = new DefinedColumnSchema {
                   { Col1, DefinedColumnType.STRING},
                   { Col2, DefinedColumnType.STRING}
                };
            IndexMeta indexMeta = new IndexMeta(IndexName);
            indexMeta.PrimaryKey = new List<string>() { Col1 };
            indexMeta.DefinedColumns = new List<string>() { Col2 };
            //indexMeta.IndexType = IndexType.IT_GLOBAL_INDEX;
            //indexMeta.IndexUpdateModel = IndexUpdateMode.IUM_ASYNC_INDEX;
            List<IndexMeta> indexMetas = new List<IndexMeta>() { };
            indexMetas.Add(indexMeta);
            CapacityUnit reservedThroughput = new CapacityUnit(0, 0);
            CreateTableRequest request = new CreateTableRequest(tableMeta, reservedThroug
hput, indexMetas);
            otsClient.CreateTable(request);
            Console.WriteLine("Table is created: " + TableName);
        }
```

7.4.2. 更新表

更新指定表的预留读吞吐量或预留写吞吐量设置。

接口

```
/// <summary>
/// 更新指定表的预留读吞吐量或预留写吞吐量，新设置将于更新成功一分钟内生效。
/// </summary>
/// <param name="request">请求参数，包含表名以及预留读写吞吐量</param>
/// <returns>包含更新后的预留读写吞吐量等信息</returns>
public UpdateTableResponse UpdateTable(UpdateTableRequest request);
/// <summary>
/// UpdateTable的异步形式。
/// </summary>
public Task<UpdateTableResponse> UpdateTableAsync(UpdateTableRequest request);
```

示例

更新表的预留读吞吐量为1，预留写吞吐量为2。

```
//设置新的预留读吞吐量为1，预留写吞吐量为2。
var reservedThroughput = new CapacityUnit(1, 2);
//构造UpdateTableRequest对象。
var request = new UpdateTableRequest("SampleTable", reservedThroughput);
try
{
    //调用接口更新表的预留读写吞吐量。
    otsClient.UpdateTable(request);
    //如果没有抛出异常，则说明执行成功。
    Console.Writeline("Update table succeeded.");
}
catch (Exception ex)
{
    //如果抛出异常，则说明执行失败，处理异常。
    Console.WriteLine("Update table failed, exception:{0}", ex.Message);
}
```

详细代码请参见UpdateTable@GitHub。

7.4.3. 列出表名称

使用ListTable接口获取当前实例下已创建的所有表的表名。

接口

```
/// <summary>
/// 获取当前实例下已创建的所有表的表名。
/// </summary>
/// <param name="request">请求参数</param>
/// <returns>ListTable的返回，用来获取表名列表。</returns>
public ListTableResponse ListTable(ListTableRequest request);
/// <summary>
/// ListTable的异步形式。
/// </summary>
public Task<ListTableResponse> ListTableAsync(ListTableRequest request);
```

示例

获取实例下所有表的表名。

```
var request = new ListTableRequest();
try
{
    var response = otsClient.ListTable(request);
    foreach (var tableName in response.TableNames)
    {
        Console.WriteLine("Table name:{0}", tableName);
    }
    Console.WriteLine("List table succeeded.");
}
catch (Exception ex)
{
    Console.WriteLine("List table failed, exception:{0}", ex.Message);
}
```

7.4.4. 查询表描述信息

使用DescribeTable接口可以查询指定表的结构、预留读/写吞吐量详情等信息。

> **说明** API说明请参见DescribeTable。

接口

```
/// <summary>
/// 查询指定表的结构信息和预留读写吞吐量设置信息。
/// </summary>
/// <param name="request">请求参数，包含表名</param>
/// <returns>包含表的结构信息和预留读写吞吐量等信息。</returns>
public DescribeTableResponse DescribeTable(DescribeTableRequest request);
/// <summary>
/// DescribeTable的异步形式。
/// </summary>
public Task<DescribeTableResponse> DescribeTableAsync(DescribeTableRequest request);
```

参数

参数	说明
tableName	表名。

示例

获取表的描述信息。

```
 try
 {
     var request = new DescribeTableRequest("SampleTable");
     var response = otsClient.DescribeTable(request);
     //打印表的描述信息。
     Console.Writeline("Describe table succeeded.");
     Console.WriteLine("LastIncreaseTime: {0}", response.ReservedThroughputDetails.LastIncr
easeTime);
     Console.WriteLine("LastDecreaseTime: {0}", response.ReservedThroughputDetails.LastDecr
easeTime);
     Console.WriteLine("NumberOfDecreaseToday: {0}", response.ReservedThroughputDetails.Las
tIncreaseTime);
     Console.WriteLine("ReadCapacity: {0}", response.ReservedThroughputDetails.CapacityUnit
.Read);
     Console.WriteLine("WriteCapacity: {0}", response.ReservedThroughputDetails.CapacityUni
t.Write);
 }
 catch (Exception ex)
 {
     //如果抛出异常，则说明执行失败，处理异常。
     Console.WriteLine("Describe table failed, exception:{0}", ex.Message);
        }
```

详细代码请参见DescribeTable@GitHub。

7.4.5. 删除数据表

使用DeleteTable接口删除当前实例下指定数据表。

> **说明** API说明请参见DeleteTable。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表。
- 已删除数据表上的索引表和多元索引。

接口

```
/// <summary>
/// 根据数据表名称删除数据表。
/// </summary>
/// <param name="request">请求参数，包含数据表名称</param>
/// <returns>DeleteTable的返回，返回实例为空，不包含具体信息。
/// </returns>
public DeleteTableResponse DeleteTable(DeleteTableRequest request);
/// <summary>
/// DeleteTable的异步形式。
/// </summary>
public Task<DeleteTableResponse> DeleteTableAsync(DeleteTableRequest request);
```

示例

删除指定数据表。

```
var request = new DeleteTableRequest("SampleTable");
try
{
    otsClient.DeleteTable(request);
    Console.Writeline("Delete table succeeded.");
}
catch (Exception ex)
{
    Console.WriteLine("Delete table failed, exception:{0}", ex.Message);
}
```

详细代码请参见DeleteTable@GitHub。

7.4.6. 主键列自增

设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。

前提条件

已初始化Client，详情请参见初始化。

使用流程

1. 创建表时，将非分区键的主键列设置为自增列。

   只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号长整型。

2. 写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符。

   如果需要获取写入数据后系统自动生成的自增列的值，将ReturnType设置为RT_PK，可以在数据写入成功后返回自增列的值。

   查询数据时，需要完整的主键值。通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为RT_PK可以获取完整的主键值。

示例

主键自增列功能主要涉及创建表（CreateTable）和写数据（PutRow、UpdateRow和BatchWriteRow）两类接口。

- 创建表时，只需将自增的主键属性设置为AUTO_INCREMENT。
- 写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符AUTO_INCREMENT。

```
using System;
using System.Collections.Generic;
using Aliyun.OTS.DataModel;
using Aliyun.OTS.Request;
using Aliyun.OTS.Response;
namespace Aliyun.OTS.Samples.Samples
{
    /// <summary>
    /// 主键列自增示例。
    /// </summary>
    public class AutoIncrementSample
    {
        private static readonly string TableName = "AutoIncrementSample";
        private static readonly string Pk1 = "Pk1";
        private static readonly string Pk2 = "Pk2_AutoIncrement";
        static void Main(string[] args)
        {
            Console.WriteLine("AutoIncrementSample");
            //创建一个带自增列的表。
            CreateTableWithAutoIncrementPk();
            //写入10行数据。
            for (int i = 0; i < 10; i++)
            {
                PutRow(i.ToString());
            }
            Console.ReadLine();
        }
        /// <summary>
        /// 创建一个带自增列的表。
        /// </summary>
        private static void CreateTableWithAutoIncrementPk()
        {
            OTSClient otsClient = Config.GetClient();
            IList<string> tables = otsClient.ListTable(new ListTableRequest()).TableNames;
            if (tables.Contains(TableName))
            {
                return;
            }
            PrimaryKeySchema primaryKeySchema = new PrimaryKeySchema
            {
                { Pk1, ColumnValueType.String },
                //指定Pk2为自增主键。
                { Pk2, ColumnValueType.Integer, PrimaryKeyOption.AUTO_INCREMENT}
            };
            TableMeta tableMeta = new TableMeta(TableName, primaryKeySchema);
            CapacityUnit reservedThroughput = new CapacityUnit(0, 0);
            CreateTableRequest request = new CreateTableRequest(tableMeta, reservedThroughp
```

```
            CreateTableRequest request = new CreateTableRequest(tableMeta, reservedThroughp
ut);
            otsClient.CreateTable(request);
        }
        public static void PutRow(string pk1Value)
        {
            Console.WriteLine("Start put row...");
            OTSClient otsClient = Config.GetClient();
            //定义行的主键，必须与创建表时的TableMeta中定义的一致。
            PrimaryKey primaryKey = new PrimaryKey
            {
                { Pk1, new ColumnValue(pk1Value) },
                { Pk2,  ColumnValue.AUTO_INCREMENT }
            };
            //定义要写入该行的属性列。
            AttributeColumns attribute = new AttributeColumns
            {
                { "Col1", new ColumnValue(0) }
            };
            PutRowRequest request = new PutRowRequest(TableName, new Condition(RowExistence
Expectation.IGNORE), primaryKey, attribute);
            request.RowPutChange.ReturnType = ReturnType.RT_PK;
            var response =  otsClient.PutRow(request);
            Console.WriteLine("Put row succeed, autoIncrement Pk value:"+ response.Row.GetP
rimaryKey()[Pk2].IntegerValue);
        }
    }
}
```

7.4.7. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过PutRow、UpdateRow、DeleteRow或BatchWriteRow接口更新数据时，可以使用条件更新检查行存在性条件和列条件，只有满足条件时才能更新成功。

条件更新包括行存在性条件和列条件。

- 行存在性条件：包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。

  对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。

- 列条件：包括RelationalCondition和CompositeCondition，是基于某一列或者某些列的列值进行条件判断。
  - RelationalCondition支持一列（可以是主键列）和一个常量比较。不支持两列或者两个常量相比较。

- CompositeCondition的内节点为逻辑运算，子条件可以是RelationalCondition或CompositeCondition。

条件更新可以实现乐观锁功能，即在更新某行时，先获取某列的值，假设为列A，值为1，然后设置条件**列A = 1**，更新行使**列A = 2**。如果更新失败，表示有其他客户端已成功更新该行。

限制

条件更新的列条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。

参数

参数	说明
RowExistenceExpectation	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST。 • IGNORE：表示忽略，不做任何存在性检查。 • EXPECT_EXIST：表示期望存在，如果该行存在，则满足条件；如果该行不存在，则不满足条件。 • EXPECT_NOT_EXIST：期望行不存在，如果该行不存在，则满足条件；如果该行存在，则不满足条件。
ColumnName	列的名称。
ColumnValue	列的对比值。
Operator	对列值进行比较的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_THAN（>=）、LESS_THAN（<）和LESS_EQUAL（<=）。
LogicOperator	对多个条件进行组合的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR。 逻辑运算符不同可以添加的子条件个数不同。 • 当逻辑运算符为NOT时，只能添加一个子条件。 • 当逻辑运算符为AND或OR时，必须至少添加两个子条件。
PassIfMissing	当列在某行中不存在时，条件检查是否通过。类型为bool值，默认值为true，表示如果列在某行中不存在时，则条件检查通过，该行满足更新条件。 当设置PassIfMissing为false时，如果列在某行中不存在时，则条件检查不通过，该行不满足更新条件。
LatestVersionsOnly	当列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置LatestVersionsOnly为false时，如果列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就条件检查通过，该行满足更新条件。

示例

```
    //定义行的主键，必须与创建表时的TableMeta中定义的一致。
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    //定义行的属性列。
    AttributeColumns attribute = new AttributeColumns();
    attribute.Add("col0", new ColumnValue(0));
    attribute.Add("col1", new ColumnValue("a"));
    attribute.Add("col2", new ColumnValue(true));
    PutRowRequest request = new PutRowRequest(tableName, new Condition(RowExistenceExpectat
ion.IGNORE), primaryKey, attribute);
    //不带condition时putrow，预期更新成功。
    try
    {
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    } catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
    //当col0列的值不等于5，将再次putrow，覆盖原有值，预期更新成功。
    try
    {
        request.Condition.ColumnCondition = new RelationalCondition("col0",
                                            CompareOperator.NOT_EQUAL,
                                            new ColumnValue(5));
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    } catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
    //当col0列的值等于5，将再次putrow，覆盖原有值，预期更新失败。
    try
    {
        //新增col0列的值等于5条件。
        request.Condition.ColumnCondition = new RelationalCondition("col0",
                                            CompareOperator.EQUAL,
                                            new ColumnValue(5));
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    }
    catch (OTSServerException)
    {
        //由于条件不满足，抛出OTSServerException。
        Console.WriteLine("Put row failed  because condition check failed. but expected");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
```

7.4.8. 原子计数器

将列当成一个原子计数器使用，对该列进行原子计数操作，可用于为某些在线应用提供实时统计功能，例如统计帖子的PV（实时浏览量）等。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

限制

- 只支持对整型列的列值进行原子计数操作。
- 作为原子计数器的列，如果写入数据前该列不存在，则默认值为0；如果写入数据前该列已存在且列值非整型，则产生OTSParameterInvalid错误。
- 增量值可以是正数或负数，但不能出现计算溢出。如果出现计算溢出，则产生OTSParameterInvalid错误。
- 默认不返回进行原子计数操作的列值，可以通过相应操作指定返回进行原子计数操作的列值。
- 在单次更新请求中，不能对某一列同时进行更新和原子计数操作。假设列A已经执行原子计数操作，则列A不能再执行其他操作（例如列的覆盖写，列删除等）。
- 在一次BatchWriteRow请求中，支持对同一行进行多次更新操作。但是如果某一行已进行原子计数操作，则该行在此批量请求中只能出现一次。
- 原子计数操作只能作用在列值的最新版本，不支持对列值的特定版本做原子计数操作。更新完成后，原子计数操作会插入一个新的数据版本。

接口

rowUpdateChange类中新增了原子计数器的操作接口，操作接口说明请参见下表。

接口	说明
RowUpdateChange Increment(Column column)	对列执行增量变更，例如+X，-X等。
List<String> ReturnColumnNames	对于进行原子计数操作的列，设置需要返回列值的列名。
ReturnType ReturnType	设置返回类型，返回进行原子计数操作的列的新值。

参数

参数	说明
TableName	数据表名称。
ColumnName	进行原子计数操作的列名。只支持对整型列的列值进行原子计数操作。
Value	对列进行增量变更的值。
ReturnColumnNames	对于进行原子计数操作的列，设置需要返回列值的列名。
ReturnType	设置返回类型为ReturnType.RT_AFTER_MODIFY，将进行原子计数操作的列值返回。

示例

写入数据时，使用rowUpdateChange接口对整型列做列值的增量变更，然后读取更新后的新值。

```
public static void Increment(int incrementValue)
{
    Console.WriteLine("Start set increment column...");
    OTSClient otsClient = Config.GetClient();
    //定义行的主键，必须与创建表时TableMeta中定义的一致。
    PrimaryKey primaryKey = new PrimaryKey
    {
        { Pk1, new ColumnValue(0) },
        { Pk2, new ColumnValue("abc") }
    };
    RowUpdateChange rowUpdateChange = new RowUpdateChange(TableName, primaryKey); //设置数据表名称。
    //设置ReturnType为ReturnType.RT_AFTER_MODIFY，将进行原子计数操作的列值返回。
    rowUpdateChange.ReturnType = ReturnType.RT_AFTER_MODIFY;
    rowUpdateChange.ReturnColumnNames = new List<string>() { IncrementCol};
    //设置进行原子计数操作的列，该列从0开始自增，每次增加incrementValue。
    rowUpdateChange.Increment(new Column(IncrementCol, new ColumnValue(incrementValue)));
    UpdateRowRequest updateRowRequest = new UpdateRowRequest(rowUpdateChange);
    var response = otsClient.UpdateRow(updateRowRequest);
    Console.WriteLine("set Increment column succeed, Increment result:" + response.Row.GetColumns()[0].Value);
}
```

7.4.9. 过滤器

在服务端对读取结果再进行一次过滤，根据过滤器（Filter）中的条件决定返回的行。使用过滤器后，只返回符合条件的数据行。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过GetRow、BatchGetRow或GetRange接口查询数据时，可以使用过滤器只返回符合条件的数据行。

过滤器目前包括RelationalCondition和CompositeCondition。

- RelationalCondition：只判断某个参考列的列值。
- CompositeCondition：根据多个参考列的列值的判断结果进行逻辑组合，决定是否过滤某行。

限制

- 过滤器的条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。
- 过滤器中的参考列必须在读取的结果内。如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。
- 在GetRow、BatchGetRow和GetRange接口中使用过滤器不会改变接口的原生语义和限制项。

使用GetRange接口时，一次扫描数据的行数不能超过5000行或者数据大小不能超过4 MB。

当在该次扫描的5000行或者4 MB数据中没有满足过滤器条件的数据时，得到的Response中的Rows为空，但是nextStartPrimaryKey可能不为空，此时需要使用nextStartPrimaryKey继续读取数据，直到nextStartPrimaryKey为空。

参数

参数	说明
ColumnName	过滤器中参考列的名称。
ColumnValue	过滤器中参考列的对比值。
ComparatorType	过滤器中的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=）。
LogicOperator	过滤器中的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR。
PassIfMissing	当参考列在某行中不存在时，是否返回该行。类型为bool值，默认值为true，表示如果参考列在某行中不存在，则返回该行。 当设置PassIfMissing为false时，如果参考列在某行中不存在，则不返回该行。
LatestVersionsOnly	当参考列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果参考列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置LatestVersionsOnly为false时，如果参考列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就返回该行。

示例

- 构造RelationalCondition。

```
public void GetRowWithRelationalCondition(OTSClient otsClient)
{
    //定义行的主键，必须与创建表时的TableMeta中定义的一致。
    PrimaryKey primaryKey = new PrimaryKey
    {
        { "pk0", new ColumnValue(0) },
        { "pk1", new ColumnValue("abc") }
    };
    var rowQueryCriteria = new SingleRowQueryCriteria(TableName)
    {
        RowPrimaryKey = primaryKey
    };
    //只返回col0的值等于5的行。
    var filter = new RelationalCondition("col0",CompareOperator.EQUAL,new ColumnValue(5))
    {
        PassIfMissing = true
    };
    rowQueryCriteria.Filter = filter.ToFilter();
    rowQueryCriteria.AddColumnsToGet("col0");
    rowQueryCriteria.AddColumnsToGet("col1");
    GetRowRequest request = new GetRowRequest(rowQueryCriteria);
    //查询。
    GetRowResponse response = otsClient.GetRow(request);
    PrimaryKey primaryKeyRead = response.PrimaryKey;
    AttributeColumns attributesRead = response.Attribute;
    Console.WriteLine("Primary key read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in primaryKeyRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Attributes read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in attributesRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Get row with filter succeed.");
}
```

- 构造CompositeCondition。

```
public void GetRowWithCompositeCondition(OTSClient otsClient)
{
    //定义行的主键，必须与创建表时的TableMeta中定义的一致。
    PrimaryKey primaryKey = new PrimaryKey
    {
        { "pk0", new ColumnValue(0) },
        { "pk1", new ColumnValue("abc") }
    };
    var rowQueryCriteria = new SingleRowQueryCriteria(TableName)
    {
        RowPrimaryKey = primaryKey
        };
    //只返回col0的值等于5的行或者col1不等于ff的行。
    var filter1 = new RelationalCondition("col0",
                                          CompareOperator.EQUAL,
                                          new ColumnValue(5));
    var filter2 = new RelationalCondition("col1", CompareOperator.NOT_EQUAL, new ColumnVa
lue("ff"));
    var filter = new CompositeCondition(LogicOperator.OR);
    filter.AddCondition(filter1);
    filter.AddCondition(filter2);
    rowQueryCriteria.Filter = filter.ToFilter();
    rowQueryCriteria.AddColumnsToGet("col0");
    rowQueryCriteria.AddColumnsToGet("col1");
    GetRowRequest request = new GetRowRequest(rowQueryCriteria);
    //查询。
    GetRowResponse response = otsClient.GetRow(request);
    PrimaryKey primaryKeyRead = response.PrimaryKey;
    AttributeColumns attributesRead = response.Attribute;
    Console.WriteLine("Primary key read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in primaryKeyRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Attributes read: ");
    foreach (KeyValuePair<string, ColumnValue> entry in attributesRead)
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("Get row with filter succeed.");
}
```

7.5. 单行数据操作

表格存储提供了PutRow、GetRow、UpdateRow和DeleteRow等单行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

插入一行数据（PutRow）

PutRow接口用于新写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。

- 接口

```
/// <summary>
/// 指定数据表名称、主键和属性，写入一行数据。返回本次操作消耗的CapacityUnit。
/// </summary>
/// <param name="request">插入数据的请求</param>
/// <returns>本次操作消耗的CapacityUnit</returns>
public PutRowResponse PutRow(PutRowRequest request);
/// <summary>
/// PutRow的异步形式。
/// </summary>
public Task<PutRowResponse> PutRowAsync(PutRowRequest request);
```

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** ○ 设置的主键个数和类型必须和数据表的主键个数和类型一致。 ○ 当主键为自增列时，只需将自增列的值设置为占位符。更多信息，请参见主键列自增。
attribute	行的属性列。 ○ 每一项的顺序是属性名、属性类型（可选）、属性值、时间戳（可选）。 ○ 时间戳即数据的版本号。更多信息，请参见数据版本和生命周期。 数据的版本号可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。

参数	说明
condition	支持使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件。更多信息，请参见条件更新。 **说明** ◦ 从.NET SDK 2.2.0版本开始，Condition不仅支持行条件，也支持列条件。 ◦ Condition.IGNORE、Condition.EXPECT_EXIST和Condition.EXPECT_NOT_EXIST从.NET SDK 3.0.0版本开始被废弃，请替换为new Condition (RowExistenceExpectation.IGNORE)、new Condition (RowExistenceExpectation.EXPECT_EXIST)和new Condition (RowExistenceExpectation.EXPECT_NOT_EXIST)。 ◦ RowExistenceExpectation.IGNORE表示无论此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。 ◦ RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。 ◦ RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。

- 示例1

  插入一行数据。

```
    //定义行的主键，必须与创建表时的TableMeta中定义的一致。
    var primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    //定义要写入该行的属性列。
    var attribute = new AttributeColumns();
    attribute.Add("col0", new ColumnValue(0));
    attribute.Add("col1", new ColumnValue("a"));
    attribute.Add("col2", new ColumnValue(true));
    try
    {
        //构造插入数据的请求对象，RowExistenceExpectation.IGNORE表示无论此行是否存在均会插入新数据。
        var request = new PutRowRequest("SampleTable", new Condition(RowExistenceExpectation.IGNORE),
                                    primaryKey, attribute);
        //调用PutRow接口插入数据。
        otsClient.PutRow(request);
        //如果没有抛出异常，则说明执行成功。
        Console.WriteLine("Put row succeeded.");
    }
    catch (Exception ex)
    {
        //如果抛出异常，则说明执行失败，处理异常。
        Console.WriteLine("Put row failed, exception:{0}", ex.Message);
    }
```

详细代码请参见PutRow@GitHub。

- 示例2

  设置条件插入一行数据。如下示例的条件为当行存在且col0大于24时才执行插入操作。

```
    //定义行的主键，必须与创建表时的TableMeta中定义的一致。
    var primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    //定义要写入该行的属性列。
    AttributeColumns attribute = new AttributeColumns();
    attribute.Add("col0", new ColumnValue(0));
    attribute.Add("col1", new ColumnValue("a"));
    attribute.Add("col2", new ColumnValue(true));
            var request = new PutRowRequest(tableName, new Condition(RowExistenceExpectat
ion.EXPECT_EXIST),
                                    primaryKey, attribute);
    //当col0列的值大于24时，允许再次插入行，覆盖掉原值。
    try
    {
        request.Condition.ColumnCondition = new RelationalCondition("col0",
                                            RelationalCondition.CompareOperator.GREATER_T
HAN,
                                            new ColumnValue(24));
        otsClient.PutRow(request);
        Console.WriteLine("Put row succeeded.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Put row failed. error:{0}", ex.Message);
    }
```

详细代码请参见ConditionPutRow@GitHub。

- 示例3

  异步插入一行数据。

  > **注意** 每一个异步调用都会启动一个线程，如果连续启动了很多异步调用，且每个都耗时比较大时，可能会出现超时。

```
    try
    {
        var putRowTaskList = new List<Task<PutRowResponse>>();
        for (int i = 0; i < 100; i++)
        {
            //定义行的主键，必须与创建表时的TableMeta中定义的一致。
            var primaryKey = new PrimaryKey();
            primaryKey.Add("pk0", new ColumnValue(i));
            primaryKey.Add("pk1", new ColumnValue("abc"));
            //定义要写入该行的属性列。
            var attribute = new AttributeColumns();
            attribute.Add("col0", new ColumnValue(i));
            attribute.Add("col1", new ColumnValue("a"));
            attribute.Add("col2", new ColumnValue(true));
            var request = new PutRowRequest(TableName, new Condition(RowExistenceExpectat
ion.IGNORE),
                                            primaryKey, attribute);
            putRowTaskList.Add(TabeStoreClient.PutRowAsync(request));
        }
        //等待每个异步调用返回，并打印出消耗的CU值。
        foreach (var task in putRowTaskList)
        {
            task.Wait();
            Console.WriteLine("consumed read:{0}, write:{1}", task.Result.ConsumedCapacit
yUnit.Read,
                                    task.Result.ConsumedCapacityUnit.Write);
        }
        //如果没有抛出异常，则说明执行成功。
        Console.WriteLine("Put row async succeeded.");
    }
    catch (Exception ex)
    {
        //如果抛出异常，则说明执行失败，处理异常。
        Console.WriteLine("Put row async failed. exception:{0}", ex.Message);
    }
```

详细代码请参见PutRowAsync@GitHub。

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

读取的结果可能有如下两种：

- 如果该行存在，则返回该行的各主键列以及属性列。
- 如果该行不存在，则返回中不包含行，并且不会报错。
- 接口

```
/// <summary>
/// 根据给定的主键读取单行数据。
/// </summary>
/// <param name="request">查询数据的请求</param>
/// <returns>GetRow的响应</returns>
public GetRowResponse GetRow(GetRowRequest request);
/// <summary>
/// GetRow的异步形式。
/// </summary>
public Task<GetRowResponse> GetRowAsync(GetRowRequest request);
```

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ◦ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 ◦ 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 ◦ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ◦ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ◦ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。

参数	说明
timeRange	读取版本号范围或特定版本号的数据。更多信息，请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 ○ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 如果查询一个范围的数据，则需要设置StartTime和EndTime。StartTime和EndTime分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[StartTime, EndTime)。 ○ 如果查询特定版本号的数据，则需要设置SpecificTime。SpecificTime表示特定的时间戳。 SpecificTime和[StartTime, EndTime)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Int64.MaxValue。
filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行。更多信息，请参见过滤器。 **说明** 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。

- 示例1

  读取一行数据。

```
//定义行的主键，必须与创建表时的TableMeta中定义的一致。
PrimaryKey primaryKey = new PrimaryKey();
primaryKey.Add("pk0", new ColumnValue(0));
primaryKey.Add("pk1", new ColumnValue("abc"));
try
{
    //构造查询请求对象，此处未指定读取的列，默认读取整行数据。
    var request = new GetRowRequest(TableName, primaryKey);
    //调用GetRow接口查询数据。
    var response = otsClient.GetRow(request);
    //输出此行的数据，此处省略，详见示例代码的GitHub链接。
    //如果没有抛出异常，则说明执行成功。
    Console.WriteLine("Get row succeeded.");
}
catch (Exception ex)
{
    //如果抛出异常，则说明执行失败，处理异常。
    Console.WriteLine("Update table failed, exception:{0}", ex.Message);
}
```

详细代码请参见Get Row@Git Hub。

- 示例2

  使用过滤器读取一行数据。

  如下示例为查询数据后，只返回col0和col1的数据，同时在col0列和col1列进行过滤，过滤条件是col0等于5或者col1不等于ff。

```
    //定义行的主键，必须与创建表时的TableMeta中定义的一致。
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    var rowQueryCriteria = new SingleRowQueryCriteria("SampleTable");
    rowQueryCriteria.RowPrimaryKey = primaryKey;
    //条件1为col0列的值等于5。
    var filter1 = new RelationalCondition("col0",
                RelationalCondition.CompareOperator.EQUAL,
                new ColumnValue(5));
    //条件2为col1列的值不等于ff。
    var filter2 = new RelationalCondition("col1", RelationalCondition.CompareOperator.NOT
_EQUAL, new ColumnValue("ff"));
    //构造组合条件，包括条件1和条件2，关系是OR。
    var filter = new CompositeCondition(CompositeCondition.LogicOperator.OR);
    filter.AddCondition(filter1);
    filter.AddCondition(filter2);
    rowQueryCriteria.Filter = filter;
    //设置要查询和返回的行，查询和过滤的顺序是先在行的[col0,col1]列上查询，然后再按条件过滤。
    rowQueryCriteria.AddColumnsToGet("col0");
    rowQueryCriteria.AddColumnsToGet("col1");
    //构造GetRowRequest。
    var request = new GetRowRequest(rowQueryCriteria);
    try
    {
        //查询。
        var response = otsClient.GetRow(request);
        //输出数据或者相关逻辑操作，此处省略。
        //如果没有抛出异常，则说明执行成功。
        Console.WriteLine("Get row with filter succeeded.");
    }
    catch (Exception ex)
    {
        //如果抛出异常，则说明执行失败，处理异常。
        Console.WriteLine("Get row with filter failed, exception:{0}", ex.Message);
    }
```

详细代码请参见GetRowWithFilter@GitHub。

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，可以增加和删除一行中的属性列，删除属性列指定版本的数据，或者更新已存在的属性列的值。如果更新的行不存在，则新增一行数据。

> **说明** 当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。

- 接口

```
    /// <summary>
    /// 更新指定行的数据，如果该行不存在，则新增一行；若该行存在，则根据请求的内容在该行中新增、修改或者删除指定列的值。
    /// </summary>
    /// <param name="request">请求实例</param>
    public UpdateRowResponse UpdateRow(UpdateRowRequest request);
    /// <summary>
    /// UpdateRow的异步形式。
    /// </summary>
    /// <param name="request"></param>
    /// <returns></returns>
    public Task<UpdateRowResponse> UpdateRowAsync(UpdateRowRequest request);
```

- 参数

参数	说明
tableName	数据表名称。
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件。更多信息，请参见条件更新。
attribute	更新的属性列。 ○ 增加或更新数据时，需要设置属性名、属性值、属性类型（可选）、时间戳（可选）。 时间戳即数据的版本号，可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。详情请参见数据版本和生命周期。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。 ○ 删除属性列特定版本的数据时，只需要设置属性名和时间戳。 时间戳是64位整数，单位为毫秒，表示某个特定版本的数据。 ○ 删除属性列时，只需要设置属性名。 **说明** 删除一行的全部属性列不等同于删除该行，如果需要删除该行，请使用DeleteRow操作。

- 示例

  更新一行数据。

```
    //定义行的主键，必须与创建表时的TableMeta中定义的一致。
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    //定义要写入该行的属性列。
    UpdateOfAttribute attribute = new UpdateOfAttribute();
    attribute.AddAttributeColumnToPut("col0", new ColumnValue(0));
    attribute.AddAttributeColumnToPut("col1", new ColumnValue("b")); // 将原先的值'a'改为'b
'
    attribute.AddAttributeColumnToPut("col2", new ColumnValue(true));
    try
    {
        //构造更新行的请求对象，RowExistenceExpectation.IGNORE表示无论此行是否存在都执行更新。
        var request = new UpdateRowRequest(TableName, new Condition(RowExistenceExpectati
on.IGNORE),
                                    primaryKey, attribute);
        //调用UpdateRow接口更新数据。
        otsClient.UpdateRow(request);
        //如果没有抛出异常，则说明执行成功。
        Console.Writeline("Update row succeeded.");
    }
    catch (Exception ex)
    {
        //如果抛出异常，则说明执行失败，处理异常。
        Console.WriteLine("Update row failed, exception:{0}", ex.Message);
    }
```

详细代码请参见UpdateRow@GitHub。

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行数据。如果删除的行不存在，则不会发生任何变化。

- 接口

```
    /// <summary>
    /// 指定数据表名称和行的主键，删除一行数据。
    /// </summary>
    /// <param name="request">请求实例</param>
    /// <returns>响应实例</returns>
    public DeleteRowResponse DeleteRow(DeleteRowRequest request);
    /// <summary>
    /// DeleteRow的异步形式。
    /// </summary>
    public Task<DeleteRowResponse> DeleteRowAsync(DeleteRowRequest request);
```

- 参数

参数	说明
tableName	数据表名称。

参数	说明
primaryKey	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件。更多信息，请参见条件更新。

- 示例

  删除一行数据。

```
    //要删除的行的PK列分别为0和"abc"。
    var primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(0));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    try
    {
        //构造请求，RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才执行删除。
        var deleteRowRequest = new DeleteRowRequest("SampleTable", new Condition(RowExist
enceExpectation.EXPECT_EXIST), primaryKey);
        //调用DeleteRow接口删除数据。
        otsClient.DeleteRow(deleteRowRequest);
        //如果没有抛出异常，则说明执行成功。
        Console.Writeline("Delete table succeeded.");
    }
    catch (Exception ex)
    {
        //如果抛出异常，则说明执行失败，处理异常。
        Console.WriteLine("Delete table failed, exception:{0}", ex.Message);
    }
```

  详细代码请参见DeleteRow@GitHub。

7.6. 多行数据操作

表格存储提供了BatchGetRow、BatchWriteRow、GetRange和GetRangeIterator等多行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

批量写（BatchWriteRow）

批量写接口用于在一次请求中进行批量的写入操作，也支持一次对多个数据表进行写入。BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，构造子操作的过程与使用PutRow接口、UpdateRow接口和DeleteRow接口时相同，也支持使用条件更新。

BatchWriteRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量写入可能存在部分行失败的情况，失败行的Index及错误信息在返回的BatchWriteRowResponse中，但并不抛出异常。因此调用BatchWriteRow接口时，需要检查返回值，判断每行的状态是否成功；如果不检查返回值，则可能会忽略掉部分操作的失败。

当服务端检查到某些操作出现参数错误时，BatchWriteRow接口可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。

- 接口

```
/// <summary>
/// <para>批量插入，修改或删除一个或多个表中的若干行数据。</para>
/// <para>BatchWriteRow操作可视为多个PutRow、UpdateRow、DeleteRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。</para>
/// <para>与执行大量的单行写操作相比，使用BatchWriteRow操作可以有效减少请求的响应时间，提高数据的写入速率。</para>
/// </summary>
/// <param name="request">请求实例</param>
/// <returns>响应实例</returns>
public BatchWriteRowResponse BatchWriteRow(BatchWriteRowRequest request);
/// <summary>
/// BatchWriteRow的异步形式。
/// </summary>
/// <param name="request"></param>
/// <returns></returns>
public Task<BatchWriteRowResponse> BatchWriteRowAsync(BatchWriteRowRequest request);
```

- 参数

  详细参数说明请参见单行数据操作。

- 示例

  批量写入100行数据。

```
//构造批量写的请求对象，设置100行数据的主键。
var request = new BatchWriteRowRequest();
var rowChanges = new RowChanges();
for (int i = 0; i < 100; i++)
{
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(i));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    //定义要写入该行的属性列。
    UpdateOfAttribute attribute = new UpdateOfAttribute();
    attribute.AddAttributeColumnToPut("col0", new ColumnValue(0));
    attribute.AddAttributeColumnToPut("col1", new ColumnValue("a"));
    attribute.AddAttributeColumnToPut("col2", new ColumnValue(true));
    rowChanges.AddUpdate(new Condition(RowExistenceExpectation.IGNORE), primaryKey, attri
bute);
}
request.Add(TableName, rowChanges);
try
{
    //调用BatchWriteRow接口写入数据。
    var response = otsClient.BatchWriteRow(request);
    var tableRows = response.TableRespones;
    var rows = tableRows[TableName];
    //批量操作可能部分成功部分失败，需要检查每行的状态是否成功，详见示例代码的GitHub链接。
}
catch (Exception ex)
{
    //如果抛出异常，则说明执行失败，处理异常。
    Console.WriteLine("Batch put row failed, exception:{0}", ex.Message);
}
```

详细代码请参见BatchWriteRow@GitHub。

批量读（BatchGetRow）

批量读接口用于一次请求读取多行数据，也支持一次对多张表进行读取。BatchGetRow由多个GetRow子操作组成。构造子操作的过程与使用GetRow接口时相同，也支持使用过滤器。

批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA列。

BatchGetRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量读取可能存在部分行失败的情况，失败行的错误信息在返回的BatchGetRowResponse中，但并不抛出异常。因此调用BatchGetRow接口时，需要检查返回值，判断每行的状态是否成功。

- 接口

```
/// <summary>
/// <para>批量读取一个或多个表中的若干行数据。</para>
/// <para>BatchGetRow操作可视为多个GetRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。</para>
/// 与执行大量的GetRow操作相比，使用BatchGetRow操作可以有效减少请求的响应时间，提高数据的读取速率。
/// </summary>
/// <param name="request">请求实例</param>
/// <returns>响应实例</returns>
public BatchGetRowResponse BatchGetRow(BatchGetRowRequest request);
/// <summary>
/// BatchGetRow的异步形式。
/// </summary>
public Task<BatchGetRowResponse> BatchGetRowAsync(BatchGetRowRequest request);
```

- 参数

  详细参数说明请参见单行数据操作。

- 示例

  批量一次读取10行。

```
//构造批量读的请求对象，设置10行数据的主键。
List<PrimaryKey> primaryKeys = new List<PrimaryKey>();
for (int i = 0; i < 10; i++)
{
    PrimaryKey primaryKey = new PrimaryKey();
    primaryKey.Add("pk0", new ColumnValue(i));
    primaryKey.Add("pk1", new ColumnValue("abc"));
    primaryKeys.Add(primaryKey);
}
try
{
    BatchGetRowRequest request = new BatchGetRowRequest();
    request.Add(TableName, primaryKeys);
    //调用BatchGetRow接口查询10行数据。
    var response = otsClient.BatchGetRow(request);
    var tableRows = response.RowDataGroupByTable;
    var rows = tableRows[TableName];
    //输出rows中的数据，此处省略，详见示例代码的GitHub链接。
    //批量操作可能部分成功部分失败，需要检查每行的状态是否成功，详见示例代码的GitHub链接。
}
catch (Exception ex)
{
    //如果抛出异常，则说明执行失败，处理异常。
    Console.WriteLine("Batch get row failed, exception:{0}", ex.Message);
}
```

  详细代码请参见BatchGetRow@GitHub。

范围读（GetRange）

范围读接口用于读取一个主键范围内的数据。

范围读接口支持按照确定范围进行正序读取和逆序读取，可以设置要读取的行数。如果范围较大，已扫描的行数或者数据量超过一定限制，会停止扫描，并返回已获取的行和下一个主键信息。您可以根据返回的下一个主键信息，继续发起请求，获取范围内剩余的行。

GetRange操作可能在如下情况停止执行并返回数据。

- 扫描的行数据大小之和达到4 MB。
- 扫描的行数等于5000。
- 返回的行数等于最大返回行数。
- 当前剩余的预留读吞吐量已全部使用，余量不足以读取下一条数据。

> **说明** 表格存储表中的行默认是按照主键排序的，而主键是由全部主键列按照顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

- 接口

```
/// <summary>
/// 根据范围条件获取多行数据。
/// </summary>
/// <param name="request">请求实例</param>
/// <returns>响应实例</returns>
public GetRangeResponse GetRange(GetRangeRequest request);
/// <summary>
/// GetRange的异步版本。
/// </summary>
/// <param name="request"></param>
/// <returns></returns>
public Task<GetRangeResponse> GetRangeAsync(GetRangeRequest request);
```

- 参数

参数	说明
tableName	数据表名称。
direction	读取方向。 ○ 如果值为正序（FORWARD），则起始主键必须小于结束主键，返回的行按照主键由小到大的顺序进行排列。 ○ 如果值为逆序（BACKWARD），则起始主键必须大于结束主键，返回的行按照主键由大到小的顺序进行排列。 例如同一表中有两个主键A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。

参数	说明
inclusiveStartPrimaryKey	本次范围读的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由INF_MIN和INF_MAX类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中INF_MIN表示无限小，任何类型的值都比它大；INF_MAX表示无限大，任何类型的值都比它小。 ○ inclusiveStartPrimaryKey表示起始主键，如果该行存在，则返回结果中一定会包含此行。 ○ exclusiveEndPrimaryKey表示结束主键，无论该行是否存在，返回结果中都不会包含此行。 数据表中的行按主键从小到大排序，读取范围是一个左闭右开的区间，正序读取时，返回的是大于等于起始主键且小于结束主键的所有的行。
exclusiveEndPrimaryKey	
limit	数据的最大返回行数，此值必须大于 0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。此时可以通过返回的断点记录本次读取到的位置，用于下一次读取。
columnsToGet	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ○ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columnsToGet参数限制。如果将col0和col1加入到columnsToGet中，则只返回col0和col1列的值。 ○ 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。 ○ 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。

参数	说明
maxVersions	最多读取的版本数。 **说明** maxVersions与timeRange必须至少设置一个。 ○ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。
timeRange	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** maxVersions与timeRange必须至少设置一个。 ○ 如果仅设置maxVersions，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置timeRange，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置maxVersions和timeRange，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 如果查询一个范围的数据，则需要设置StartTime和EndTime。StartTime和EndTime分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[StartTime, EndTime)。 ○ 如果查询特定版本号的数据，则需要设置SpecificTime。SpecificTime表示特定的时间戳。 SpecificTime和[StartTime, EndTime)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为Int64.MaxValue。
filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当columnsToGet和filter同时使用时，执行顺序是先获取columnsToGet指定的列，再在返回的列中进行条件过滤。

参数	说明
nextStartPrimaryKey	根据返回结果中的nextStartPrimaryKey判断数据是否全部读取。 ○ 当返回结果中nextStartPrimaryKey不为空时，可以使用此返回值作为下一次GetRange操作的起始点继续读取数据。 ○ 当返回结果中nextStartPrimaryKey为空时，表示读取范围内的数据全部返回。

- 示例

  按照范围读取数据。

```
//读取(0, INF_MIN)到(100, INF_MAX)范围内的所有行。
var inclusiveStartPrimaryKey = new PrimaryKey();
inclusiveStartPrimaryKey.Add("pk0", new ColumnValue(0));
inclusiveStartPrimaryKey.Add("pk1", ColumnValue.INF_MIN);
var exclusiveEndPrimaryKey = new PrimaryKey();
exclusiveEndPrimaryKey.Add("pk0", new ColumnValue(100));
exclusiveEndPrimaryKey.Add("pk1", ColumnValue.INF_MAX);
try
{
    //构造范围读的请求对象。
    var request = new GetRangeRequest(TableName, GetRangeDirection.Forward,
                    inclusiveStartPrimaryKey, exclusiveEndPrimaryKey);
    var response = otsClient.GetRange(request);
    //如果一次没有返回所有数据，则需要继续查询。
    var rows = response.RowDataList;
    var nextStartPrimaryKey = response.NextPrimaryKey;
    while (nextStartPrimaryKey != null)
    {
        request = new GetRangeRequest(TableName, GetRangeDirection.Forward,
                        nextStartPrimaryKey, exclusiveEndPrimaryKey);
        response = otsClient.GetRange(request);
        nextStartPrimaryKey = response.NextPrimaryKey;
        foreach (RowDataFromGetRange row in response.RowDataList)
        {
            rows.Add(row);
        }
    }
    //输出rows中的数据，此处省略，详见示例代码的GitHub链接。
    //如果没有抛出异常，则说明执行成功。
    Console.WriteLine("Get range succeeded");
}
catch (Exception ex)
{
    //如果抛出异常，则说明执行失败，处理异常。
    Console.WriteLine("Get range failed, exception:{0}", ex.Message);
}
```

详细代码请参见GetRange@GitHub。

迭代读（GetRangeIterator）

迭代读取数据。

- 接口

```
/// <summary>
/// 根据范围条件获取多行数据，返回用来迭代每一行数据的迭代器。
/// </summary>
/// <param name="request"><see cref="GetIteratorRequest"/></param>
/// <returns>返回<see cref="RowDataFromGetRange"/>的迭代器。</returns>
public IEnumerable<RowDataFromGetRange> GetRangeIterator(GetIteratorRequest request);
```

- 示例

  迭代读取(0, "a")到(1000, "xyz")范围内的所有行。

```
//读取(0, "a")到(1000, "xyz")范围内的所有行。
PrimaryKey inclusiveStartPrimaryKey = new PrimaryKey();
inclusiveStartPrimaryKey.Add("pk0", new ColumnValue(0));
inclusiveStartPrimaryKey.Add("pk1", new ColumnValue("a"));
PrimaryKey exclusiveEndPrimaryKey = new PrimaryKey();
exclusiveEndPrimaryKey.Add("pk0", new ColumnValue(1000));
exclusiveEndPrimaryKey.Add("pk1", new ColumnValue("xyz"));
//构造一个CapacityUnit，用于记录迭代过程中消耗的CU值。
var cu = new CapacityUnit(0, 0);
try
{
    //构造一个GetIteratorRequest，也支持使用过滤条件。
    var request = new GetIteratorRequest(TableName, GetRangeDirection.Forward, inclusiveS
tartPrimaryKey,
                                                exclusiveEndPrimaryKey, cu);
    var iterator = otsClient.GetRangeIterator(request);
    //遍历迭代器，读取数据。
    foreach (var row in iterator)
    {
        //处理逻辑。
    }
    Console.WriteLine("Iterate row succeeded");
}
catch (Exception ex)
{
    Console.WriteLine("Iterate row failed, exception:{0}", ex.Message);
}
```

详细代码请参见GetRangeIterator@GitHub。

7.7. 多元索引

7.7.1. 创建多元索引

使用CreateSearchIndex接口在数据表上创建一个多元索引。一个数据表可以创建多个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。

- 已创建数据表，且数据表的数据生命周期（TimeToLive）必须为-1，最大版本数（MaxVersions）必须为1。

参数

创建多元索引时，需要指定数据表名称（TableName）、多元索引名称（IndexName）和索引的结构信息（IndexSchema），其中IndexSchema包含FieldSchemas（Index的所有字段的设置）、IndexSetting（索引设置）和IndexSort（索引预排序设置）。详细参数说明请参见下表。

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。
FieldSchemas	FieldSchema的列表，每个FieldSchema包含如下内容： • FieldName（必选）：创建多元索引的字段名，即列名，类型为String。 多元索引中的字段可以是主键列或者属性列。 • FieldType（必选）：字段类型，类型为FieldType.XXX。更多信息，请参见字段。 • Array（可选）：是否为数组，类型为Boolean。 如果设置为true，则表示该列是一个数组，在写入时，必须按照JSON数组格式写入，例如["a","b","c"]。 由于Nested类型是一个数组，当FieldType为Nested类型时，无需设置此参数。 • Index（可选）：是否开启索引，类型为Boolean。 默认为true，表示对该列构建倒排索引或者空间索引；如果设置为false，则不会对该列构建索引。 • Analyzer（可选）：分词器类型。当字段类型为Text时，可以设置此参数；如果不设置，则默认分词器类型为单字分词。关于分词的更多信息，请参见分词。 • EnableSortAndAgg（可选）：是否开启排序与统计聚合功能，类型为Boolean。 只有EnableSortAndAgg设置为true的字段才能进行排序。关于排序的更多信息，请参见排序和翻页。 注意 Nested类型的字段不支持开启排序与统计聚合功能，但是Nested类型内部的子列支持开启排序与统计聚合功能。 • Store（可选）：是否在多元索引中附加存储该字段的值，类型为Boolean。 开启后，可以直接从多元索引中读取该字段的值，而不必反查数据表，可用于查询性能优化。
IndexSetting	索引设置，包含RoutingFields设置。 RoutingFields（可选）：自定义路由字段。可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

参数	说明
IndexSort	索引预排序设置，包含Sorters设置。如果不设置，则默认按照主键排序。 **说明** 含有Nested类型的索引不支持IndexSort，没有预排序。 Sorters（必选）：索引的预排序方式，支持按照主键排序和字段值排序。关于排序的更多信息，请参见排序和翻页。 • PrimaryKeySort表示按照主键排序，包含如下设置： Order：排序的顺序，可按升序或者降序排序，默认为升序（DataModel.Search.Sort.SortOrder.ASC）。 • FieldSort表示按照字段值排序，包含如下设置： 只有建立索引且开启排序与统计聚合功能的字段才能进行预排序。 ○ FieldName：排序的字段名。 ○ Order：排序的顺序，可按照升序或者降序排序，默认为升序（DataModel.Search.Sort.SortOrder.ASC）。 ○ Mode：当字段存在多个值时的排序方式。

示例

- 示例1

  创建一个多元索引。

```
/// <summary>
/// 创建一个多元索引，包含Keyword_type_col、Long_type_col、Text_type_col三个属性列，类型分别设置为不分词字符串(Keyword)、整型(Long)、分词字符串(Text)。
/// </summary>
/// <param name="otsClient"></param>
public static void CreateSearchIndex(OTSClient otsClient)
{
    //设置数据表名称和多元索引名称。
    CreateSearchIndexRequest request = new CreateSearchIndexRequest(TableName, IndexName)
;
    List<FieldSchema> FieldSchemas = new List<FieldSchema>() {
        new FieldSchema(Keyword_type_col,FieldType.KEYWORD){ //设置字段名和字段类型。
            index =true, //设置开启索引。
            EnableSortAndAgg =true //设置开启排序与统计聚合功能。
        },
        new FieldSchema(Long_type_col,FieldType.LONG){ index=true,EnableSortAndAgg=true},
        new FieldSchema(Text_type_col,FieldType.TEXT){ index=true}
    };
    request.IndexSchame = new IndexSchema()
    {
        FieldSchemas = FieldSchemas
    };
    //调用client创建多元索引。
    CreateSearchIndexResponse response = otsClient.CreateSearchIndex(request);
    Console.WriteLine("Searchindex is created: " + IndexName);
}
```

- 示例2

  创建多元索引时指定IndexSort。

```
/// <summary>
/// 创建一个多元索引，包含Keyword_type_col、Long_type_col、Text_type_col三个属性列，类型分别设置为不分词字符串(Keyword)、整型(Long)、分词字符串(Text)。
/// </summary>
/// <param name="otsClient"></param>
public static void CreateSearchIndexWithIndexSort(OTSClient otsClient)
{
    //设置数据表名称和多元索引名称。
    CreateSearchIndexRequest request = new CreateSearchIndexRequest(TableName, IndexName)
;
    List<FieldSchema> FieldSchemas = new List<FieldSchema>() {
        new FieldSchema(Keyword_type_col,FieldType.KEYWORD){ //设置字段名和字段类型。
            index =true, //设置开启索引。
            EnableSortAndAgg =true //设置开启排序与统计聚合功能。
        },
        new FieldSchema(Long_type_col,FieldType.LONG){ index=true,EnableSortAndAgg=true},
        new FieldSchema(Text_type_col,FieldType.TEXT){ index=true}
    };
    request.IndexSchame = new IndexSchema()
    {
        FieldSchemas = FieldSchemas,
        //按照Long_type_col列进行预排序，Long_type_col列必须建立索引且开启EnableSortAndAgg。
        IndexSort = new DataModel.Search.Sort.Sort()
        {
            Sorters = new List<DataModel.Search.Sort.ISorter>
            {
                new DataModel.Search.Sort.FieldSort(Long_type_col, DataModel.Search.Sort.
SortOrder.ASC)
            }
        }
    };
    CreateSearchIndexResponse response = otsClient.CreateSearchIndex(request);
    Console.WriteLine("Searchindex is created: " + IndexName);
}
```

7.7.2. 查询多元索引描述信息

创建多元索引后，使用DescribeSearchIndex接口可以查询多元索引的描述信息，包括多元索引的字段信息和索引配置等。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。

示例

```
/// <summary>
/// 查询多元索引的描述信息。
/// </summary>
/// <param name="otsClient"></param>
public static void DescribeSearchIndex(OTSClient otsClient)
{
    //设置数据表名称和多元索引名称。
    DescribeSearchIndexRequest request = new DescribeSearchIndexRequest(TableName, IndexNam
e);
    DescribeSearchIndexResponse response = otsClient.DescribeSearchIndex(request);
    string serializedObjectString = JsonConvert.SerializeObject(response);
    Console.WriteLine(serializedObjectString); //打印response的详细信息。
}
```

7.7.3. 列出多元索引列表

创建多元索引后，使用ListSearchIndex接口可以获取当前实例下或某个数据表关联的所有多元索引的列表信息。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称，可以为空。 • 如果设置了数据表名称，则返回该数据表关联的所有多元索引的列表。 • 如果未设置数据表名称，则返回当前实例下所有多元索引的列表。

示例

```
/// <summary>
/// 列出多元索引名称。
/// </summary>
/// <param name="otsClient"></param>
public static void ListSearchIndex(OTSClient otsClient)
{
    //设置数据表名称。
    ListSearchIndexRequest request = new ListSearchIndexRequest(TableName);
    ListSearchIndexResponse response = otsClient.ListSearchIndex(request);
    //获取数据表关联的所有多元索引。
    foreach (var index in response.IndexInfos)
    {
        Console.WriteLine("indexname:" + index.IndexName);
        Console.WriteLine("tablename:" + index.TableName);
    }
}
```

7.7.4. 删除多元索引

使用DeleteSearchIndex接口可以删除指定数据表的一个多元索引。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
TableName	数据表名称。
IndexName	多元索引名称。

示例

```
/// <summary>
/// 删除多元索引。
/// </summary>
/// <param name="otsClient"></param>
public static void DeleteSearchIndex(OTSClient otsClient)
{
    //设置数据表名和多元索引名称。
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest(TableName, IndexName);
    //调用otsClient删除多元索引。
    DeleteSearchIndexResponse response = otsClient.DeleteSearchIndex(request);
}
```

7.7.5. 精确查询

TermQuery采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。对于Text类型字段，只要分词后有词条可以精确匹配即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
FieldName	要匹配的字段。
Term	查询关键词，即要匹配的值。 该词不会被分词，会被当做完整词去匹配。 对于Text类型字段，只要分词后有词条可以精确匹配即可。例如某个Text类型的字段，值为“tablestore is cool”，如果分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
Query	设置查询类型为TermQuery。
TableName	数据表名称。
IndexName	多元索引名称。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/// <summary>
/// 精确查询Keyword_type_col列中值为"SearchIndex"的行。
/// </summary>
/// <param name="otsClient"></param>
public static void TermQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    //设置返回匹配的总行数。
    searchQuery.GetTotalCount = true;
    //设置查询的类型为TermQuery，设置要匹配的字段为Keyword_type_col，要匹配的值为"SearchIndex"。
    searchQuery.Query = new TermQuery(Keyword_type_col, new ColumnValue("SearchIndex"));
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true //设置为返回所有列。
    };
    var response = otsClient.Search(request);
    //可检查NextToken
```

7.7.6. 全匹配查询

MatchAllQuery可以匹配所有行，常用于查询表中数据总行数，或者随机返回几条数据。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
Query	设置查询类型为MatchAllQuery。
TableName	数据表名称。
IndexName	多元索引名称。
Limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置Limit=0，即不返回任意一行数据。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。

参数	说明
ColumnsToGet	是否返回所有列。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/// <summary>
/// 查询所有行，返回行数。
/// </summary>
/// <param name="otsClient"></param>
public static void MatchAllQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    searchQuery.Query = new MatchAllQuery();
    searchQuery.GetTotalCount = true; //设置GetTotalCount = true时，才会返回满足条件的数据总行数
。
    /*
    * MatchAllQuery结果中的Totalcount可以表示数据的总行数。
    * 如果只为了获取行数，但不需要具体数据，可以设置limit=0，即不返回任意一行数据。
    */
    searchQuery.Limit = 0;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    var response = otsClient.Search(request);
    //判断返回的结果是否完整，当isAllSuccess为false时，表示可能存在部分节点查询失败，返回的是部分数据
。
    Console.WriteLine("IsAllSuccess:" + response.IsAllSuccess);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

7.7.7. 匹配查询

MatchQuery采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用MatchPhraseQuery实现高性能的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明

参数	说明
FieldName	要匹配的列。 匹配查询可应用于Text类型。
Text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
Query	设置查询类型为MatchQuery。
Operator	逻辑运算符。默认为OR，表示当分词后的多个词只要有部分匹配时，则行数据满足查询条件。 如果设置Operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。
MinimumShouldMatch	最小匹配个数。 只有当某一行数据的FieldName列的值中至少包括最小匹配个数的词时，才会返回该行数据。 说明 MinimumShouldMatch需要与逻辑运算符OR配合使用。
TableName	数据表名称。
IndexName	多元索引名称。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
ColumnsToGet	是否返回所有列。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/// <summary>
/// 模糊匹配和短语或邻近查询，返回指定列。
/// </summary>
/// <param name="otsClient"></param>
public static void MatchQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    //设置查询类型为MatchQuery，要匹配的列为Text_type_col，要匹配的值为"SearchIndex"。
    searchQuery.Query = new MatchQuery(Text_type_col, "SearchIndex");
    searchQuery.GetTotalCount = true;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        Columns = new List<string>() { Long_type_col, Text_type_col, Keyword_type_col }
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

7.7.8. 短语匹配查询

类似于MatchQuery，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。如果查询列的分词类型为模糊分词，则使用MatchPhraseQuery可以实现比WildcardQuery更快的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
FieldName	要匹配的列。 匹配查询可应用于Text类型。
Text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“..., this is tablestore”、“this is a table”，但是无法匹配到“this table is ...”以及“is this a table”。
Query	设置查询类型为MatchPhraseQuery。
TableName	数据表名称。

参数	说明
IndexName	多元索引名称。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
ColumnsToGet	是否返回所有列。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/// <summary>
/// 类似MatchQuery（MatchQuery仅匹配某个词即可），但是MatchPhraseQuery会匹配所有的短语。
/// </summary>
/// <param name="otsClient"></param>
public static void MatchPhraseQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    //设置查询类型为MatchPhraseQuery。
    searchQuery.Query = new MatchPhraseQuery(Text_type_col, "TableStore SearchIndex");
    searchQuery.GetTotalCount = true;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        Columns = new List<string>() { Long_type_col, Text_type_col, Keyword_type_col }
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

7.7.9. 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
FieldName	要匹配的字段。
Prefix	前缀值。 对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
Query	设置查询类型为PrefixQuery。
TableName	数据表名称。
IndexName	多元索引名称。
ColumnsToGet	是否返回所有列，包含ReturnAll和Columns设置。 ReturnAll默认为false，表示不返回所有列，此时可以通过Columns指定返回的列；如果未通过Columns指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/// <summary>
/// 前缀查询。
/// </summary>
/// <param name="otsClient"></param>
public static void PrefixQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    //设置查询类型为PrefixQuery，匹配字段为Keyword_type_col，前缀值为"Search"。
    searchQuery.Query = new PrefixQuery(Keyword_type_col, "Search");
    searchQuery.GetTotalCount = true;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true //设置为返回所有列。
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

7.7.10. 范围查询

RangeQuery根据范围条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足范围条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
Query	设置查询类型为RangeQuery。
FieldName	要匹配的字段。
From	起始位置的值。
To	结束位置的值。
IncludeLower	结果中是否需要包括From值，类型为Boolean。
IncludeUpper	结果中是否需要包括To值，类型为Boolean。
TableName	数据表名称。
IndexName	多元索引名称。

示例

```
/// <summary>
/// 范围查询。通过设置一个范围（From，To），查询该范围内的所有数据。
/// </summary>
/// <param name="otsClient"></param>
public static void RangeQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    searchQuery.GetTotalCount = true;
    var rangeQuery = new RangeQuery(Long_type_col, new ColumnValue(0), new ColumnValue(6));
    //包括下边界（即大于等于0）。
    rangeQuery.IncludeLower = true;
    searchQuery.Query = rangeQuery;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

7.7.11. 通配符查询

通配符查询中，要匹配的值可以是一个带有通配符的字符串，目前支持星号（*）和问号（?）两种通配符。要匹配的值中可以用星号（*）代表任意字符序列，或者用问号（?）代表任意单个字符，且支持以星号（*）或问号（?）开头。例如查询“table*e”，可以匹配到“tablestore”。

如果查询的模式为*word*，则您可以使用性能更好的模糊查询，具体实现方法如下：

1. 创建多元索引时，设置列为Text类型且设置分词类型为模糊分词。
2. 使用多元索引查询数据时，使用MatchPhraseQuery且设置查询词为word。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	描述
Query	设置查询类型为WildcardQuery。
FieldName	列名称。
Value	带有通配符的字符串，字符串长度不能超过20个字符。
TableName	数据表名称。
IndexName	多元索引名称。
GetTotalCount	是否返回匹配的总行数，默认为false，表示不返回表中数据的总行数。 设置GetTotalCount为true后会影响查询性能。
ColumnsToGet	是否返回所有列。 ReturnAll默认为false，表示不返回所有列，此时可以通过ColumnsToGet指定返回的列；如果未通过ColumnsToGet指定返回的列，则只返回主键列。 当设置ReturnAll为true时，表示返回所有列。

示例

```
/// <summary>
/// 通配符查询。支持*（任意0或多个）和?（任意1个字符）两种通配符。
/// </summary>
/// <param name="otsClient"></param>
public static void WildcardQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    //设置查询类型为WildcardQuery，要匹配的值支持通配符。
    searchQuery.Query = new WildcardQuery(Keyword_type_col, "*Search*");
    searchQuery.GetTotalCount = true;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

7.7.12. 地理位置查询

地理位置查询包括地理距离查询（GeoDistanceQuery）、地理长方形范围查询（GeoBoundingBoxQuery）和地理多边形范围查询（GeoPolygonQuery）三种方式。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

地理距离查询（GeoDistanceQuery）

GeoDistanceQuery根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，满足查询条件。

- 参数

参数	说明
FieldName	列名，类型为Geopoint。
CenterPoint	中心地理坐标点，是一个经纬度值。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。
DistanceInMeter	距离中心点的距离，类型为Double。单位为米。
Query	多元索引的查询语句。设置查询类型为GeoDistanceQuery。
TableName	数据表名称。

参数	说明
IndexName	多元索引名称。

- 示例

```
/// <summary>
///  查询表中geo_type_col列的值距离中心点不超过一定距离的数据。
/// </summary>
/// <param name="client"></param>
public static void GeoDistanceQuery(OTSClient client)
{
    SearchQuery searchQuery = new SearchQuery();
    GeoDistanceQuery geoDistanceQuery = new GeoDistanceQuery();  //设置查询类型为GeoDistanceQuery。
    geoDistanceQuery.FieldName = Geo_type_col;
    geoDistanceQuery.CenterPoint = "10,11"; //设置中心点。
    geoDistanceQuery.DistanceInMeter = 10000; //设置条件为到中心点的距离不超过10000米。
    searchQuery.Query = geoDistanceQuery;
    SearchRequest searchRequest = new SearchRequest(TableName, IndexName, searchQuery);
    ColumnsToGet columnsToGet = new ColumnsToGet();
    columnsToGet.Columns = new List<string>() { Geo_type_col };  //设置返回Col_GeoPoint列。
    searchRequest.ColumnsToGet = columnsToGet;
    SearchResponse response = client.Search(searchRequest);
    Console.WriteLine(response.TotalCount);
}
```

地理长方形范围查询（GeoBoundingBoxQuery）

GeoBoundingBoxQuery根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时满足查询条件。

- 参数

参数	说明
FieldName	列名，类型为Geopoint。
TopLeft	长方形框的左上角的坐标。
BottomRight	长方形框的右下角的坐标，通过左上角和右下角就可以确定一个唯一的长方形。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。
Query	多元索引的查询语句。设置查询类型为GeoBoundingBoxQuery。
TableName	数据表名称。
IndexName	多元索引名称。

- 示例

```
/// <summary>
/// geo_type_col是Geopoint类型，查询表中geo_type_col列的值在左上角为"10,0"，右下角为"0,10"的长方形范围内的数据。
/// </summary>
/// <param name="client"></param>
public static void GeoBoundingBoxQuery(OTSClient client)
{
    SearchQuery searchQuery = new SearchQuery();
    GeoBoundingBoxQuery geoBoundingBoxQuery = new GeoBoundingBoxQuery(); //设置查询类型为GeoBoundingBoxQuery。
    geoBoundingBoxQuery.FieldName = Geo_type_col; //设置列名。
    geoBoundingBoxQuery.TopLeft = "10,0"; //设置长方形左上角。
    geoBoundingBoxQuery.BottomRight = "0,10"; //设置长方形右下角。
    searchQuery.Query = geoBoundingBoxQuery;
    SearchRequest searchRequest = new SearchRequest(TableName, IndexName, searchQuery);
    var columnsToGet = new ColumnsToGet();
    columnsToGet.Columns = new List<string> { Geo_type_col };  //设置返回Col_GeoPoint列。
    searchRequest.ColumnsToGet = columnsToGet;
    SearchResponse response = client.Search(searchRequest);
    Console.WriteLine(response.TotalCount);
}
```

地理多边形范围查询（GeoPolygonQuery）

GeoPolygonQuery根据一个多边形范围条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时满足查询条件。

- 参数

参数	说明
FieldName	列名，类型为Geopoint。
Points	组成多边形的距离坐标点。 格式为“纬度,经度”，纬度在前，经度在后，且纬度范围为-90～+90，经度范围-180～+180。例如“35.8,-45.91”。
Query	多元索引的查询语句。设置查询类型为GeoPolygonQuery。
TableName	数据表名称。
IndexName	多元索引名称。

- 示例

```
/// <summary>
/// 查询表中geo_type_col列的值在一个给定多边形范围内的数据。
/// </summary>
/// <param name="client"></param>
public static void GeoPolygonQuery(OTSClient client)
{
    SearchQuery searchQuery = new SearchQuery();
    GeoPolygonQuery geoPolygonQuery = new GeoPolygonQuery();  //设置查询类型为GeoPolygonQuery。
    geoPolygonQuery.FieldName = Geo_type_col;
    geoPolygonQuery.Points = new List<string>() { "0,0", "10,0", "10,10" }; //设置多边形的顶点。
    searchQuery.Query = geoPolygonQuery;
    SearchRequest searchRequest = new SearchRequest(TableName, IndexName, searchQuery);
    ColumnsToGet columnsToGet = new ColumnsToGet();
    columnsToGet.Columns = new List<string>() { Geo_type_col };  //设置返回Col_GeoPoint列。
    searchRequest.ColumnsToGet = columnsToGet;
    SearchResponse response = client.Search(searchRequest);
    Console.WriteLine(response.TotalCount);
}
```

7.7.13. 多条件组合查询

BoolQuery查询条件包含一个或者多个子查询条件，根据子查询条件来判断一行数据是否满足查询条件。每个子查询条件可以是任意一种Query类型，包括BoolQuery。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
MustQueries	多个Query列表，行数据必须满足所有的子查询条件才算匹配，等价于And操作符。
MustNotQueries	多个Query列表，行数据必须不能满足任何的子查询条件才算匹配，等价于Not操作符。
FilterQueries	多个Query列表，行数据必须满足所有的子Filter才算匹配，Filter类似于Query，区别是Filter不会根据满足的Filter个数进行相关性算分。
ShouldQueries	多个Query列表，可以满足，也可以不满足，等价于Or操作符。 行数据应该至少满足ShouldQueries子查询条件的最小匹配个数才算匹配。 如果满足的ShouldQueries子查询条件个数越多，则整体的相关性分数更高。

参数	说明
MinimumShouldMatch	ShouldQueries子查询条件的最小匹配个数。当同级没有其他Query，只有ShouldQueries时，默认值为1；当同级已有其他Query，例如MustQueries、MustNotQueries和FilterQueries时，默认值为0。
TableName	数据表名称。
IndexName	多元索引名称。

示例

```
/// <summary>
///多条件组合查询BoolQuery由一个或者多个子查询条件组成，每个子查询条件都有特定的类型。
///MustQueries：行数据必须完全匹配条件。
///ShouldQueries：ShouldQueries中会带有一个以上的条件，至少满足一个条件，此行数据就符合查询条件。
///MustNotQueries：行数据必须不匹配条件。
///MinimumShouldMatch：should查询的条件至少满足的个数。
/// </summary>
/// <param name="otsClient"></param>
public static void BoolQuery(OTSClient otsClient)
{
    Console.WriteLine("\n Start bool query...");
    var searchQuery = new SearchQuery();
    searchQuery.GetTotalCount = true;
    var boolQuery = new BoolQuery();
    var shouldQuerys = new List<IQuery>();
    shouldQuerys.Add(new TermQuery(Keyword_type_col, new ColumnValue("SearchIndex")));
    shouldQuerys.Add(new TermQuery(Keyword_type_col, new ColumnValue("TableStore")));
    boolQuery.ShouldQueries = shouldQuerys;
    boolQuery.MinimumShouldMatch = 1;
    searchQuery.Query = boolQuery;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
    foreach (var row in response.Rows)
    {
        PrintRow(row);
    }
}
```

7.7.14. 嵌套类型查询

NestedQuery用于查询嵌套类型字段中子行的数据。嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型字段的路径和一个子查询，其中子查询可以是任意Query类型。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
Path	路径名，嵌套类型的列的树状路径。例如news.title表示嵌套类型的news列中的title子列。
Query	嵌套类型的列中子列上的查询，子列上的查询可以是任意Query类型。
ScoreMode	当列存在多个值时基于哪个值计算分数。
TableName	数据表名称。
IndexName	多元索引名称。

示例

```
/// <summary>
/// 类型为NESTED的col1_nested列，子行包含nested_1和nested_2两列，现在查询col1_nested.nested_1为"tablestore"的数据。
/// </summary>
/// <param name="otsClient"></param>
public static void NestedQuery(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    searchQuery.GetTotalCount = true;
    var nestedQuery = new NestedQuery();
    nestedQuery.Path = "col1_nested"; //设置嵌套类型字段的路径。
    TermQuery termQuery = new TermQuery("col1_nested.nested_1",new ColumnValue("tablestore"));//构造NestedQuery的子查询。
    nestedQuery.Query = termQuery;
    nestedQuery.ScoreMode = ScoreMode.None;
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    request.ColumnsToGet = new ColumnsToGet()
    {
        ReturnAll = true
    };
    var response = otsClient.Search(request);
    Console.WriteLine("Total Count:" + response.TotalCount);
}
```

7.7.15. 排序和翻页

使用多元索引时，您可以在创建时指定索引预排序和在查询时指定排序方式，在获取返回结果时使用Limit和Offset或者使用Token进行翻页。

索引预排序

多元索引默认按照设置的索引预排序（IndexSort）方式进行排序，使用多元索引查询数据时，IndexSort决定了数据的默认返回顺序。

在创建多元索引时，您可以自定义IndexSort，如果未自定义IndexSort，则IndexSort默认为主键排序。

> **注意** 含有Nested类型字段的多元索引不支持索引预排序。

查询时指定排序方式

只有EnableSortAndAgg设置为true的字段才能进行排序。

在每次查询时，可以指定排序方式，多元索引支持如下四种排序方式（Sorter）。您也可以使用多个Sorter，实现先按照某种方式排序，再按照另一种方式排序的需求。

- ScoreSort

  按照查询结果的相关性（BM25算法）分数进行排序，适用于有相关性的场景，例如全文检索等。

  > **注意** 如果需要按照相关性打分进行排序，必须手动设置ScoreSort，否则会按照索引设置的IndexSort进行排序。

  ```
  var searchQuery = new SearchQuery();
  searchQuery.Sort = new Sort(new List<ISorter>() { new ScoreSort() });
  ```

- PrimaryKeySort

  按照主键进行排序。

  ```
  //正序。
  var searchQuery = new SearchQuery();
  searchQuery.Sort = new Sort(new List<ISorter>() { new PrimaryKeySort() });
  //逆序。
  var searchQuery = new SearchQuery();
  searchQuery.Sort = new Sort(new List<ISorter>() { new PrimaryKeySort(SortOrder.DESC) });
  ```

- FieldSort

  按照某列的值进行排序。

  ```
  var searchQuery = new SearchQuery();
  var fieldSort = new FieldSort("col", SortOrder.ASC);
  searchQuery.Sort = new Sort(new List<ISorter>() { fieldSort });
  ```

  先按照某列的值进行排序，再按照另一列的值进行排序。

  ```
  var searchQuery = new SearchQuery();
  var col1Sort = new FieldSort("col", SortOrder.ASC);
  var col2Sort = new FieldSort("co2", SortOrder.ASC);
  searchQuery.Sort = new Sort(new List<ISorter>() { col1Sort, col2Sort });
  ```

- GeoDistanceSort

  根据地理点距离进行排序。

```
var searchQuery = new SearchQuery();
var geoDistanceSort = new GeoDistanceSort("geoCol", new List<string>(){"0","0"});
searchQuery.Sort = new Sort(new List<ISorter>() { geoDistanceSort });
```

翻页方式

在获取返回结果时，可以使用Limit和Offset或者使用Token进行翻页。

- 使用Limit和Offset进行翻页

  当需要获取的返回结果行数小于10000行时，可以通过Limit和Offset进行翻页，即Limit+Offset<=10000，其中Limit的最大值为100。

  > **说明** 如果需要提高Limit的上限，请参见如何将多元索引Search接口查询数据的limit提高到1000。

  如果使用此方式进行翻页时未设置Limit和Offset，则Limit的默认值为10，Offset的默认值为0。

  ```
  var searchQuery = new SearchQuery();
  searchQuery.Query = new MatchAllQuery();
  searchQuery.Limit = 100;
  searchQuery.Offset = 100;
  ```

- 使用Token进行翻页

  由于使用Token进行翻页时翻页深度无限制，当需要进行深度翻页时，推荐使用Token进行翻页。

  当符合查询条件的数据未读取完时，服务端会返回NextToken，此时可以使用NextToken继续读取后面的数据。

  使用Token进行翻页时默认只能向后翻页。由于在一次查询的翻页过程中Token长期有效，您可以通过缓存并使用之前的Token实现向前翻页。

  使用Token翻页后的排序方式和上一次请求的一致，无论是系统默认使用IndexSort还是自定义排序，因此设置了Token不能再设置Sort。另外使用Token后不能设置Offset，只能依次往后读取，即无法跳页。

  > **注意** 由于含有Nested类型字段的多元索引不支持索引预排序，如果使用含有Nested类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回NextToken。

```
/// <summary>
/// 使用Token进行翻页，此示例将读取所有数据，放到一个List中。
/// </summary>
/// <param name="otsClient"></param>
public static SearchResponse ReadMoreRowsWithToken(OTSClient otsClient)
{
    var searchQuery = new SearchQuery();
    searchQuery.Query = new MatchAllQuery();
    var request = new SearchRequest(TableName, IndexName, searchQuery);
    var response = otsClient.Search(request);
    var rows = response.Rows;
    while (response.NextToken != null) //直到读取到NextToken为null，即读出全部数据。
    {
        request.SearchQuery.Token = response.NextToken;
        response = otsClient.Search(request);
        rows.AddRange(response.Rows);
    }
    return response;
}
```

7.8. 全局二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateGlobalIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表，详情请参见创建数据表。

- 参数

参数	说明
mainTableName	数据表名称。

参数	说明
indexMeta	索引表的结构信息，包括如下内容： ○ indexName：索引表名称。 ○ primaryKey：索引表的索引列，索引列为数据表主键和预定义列的任意组合。 ○ definedColumns：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ indexUpdateMode：索引表更新模式，当前只支持IUM_ASYNC_INDEX。 ○ indexType：索引表类型，当前只支持IT_GLOBAL_INDEX。
includeBaseData	索引表中是否包含数据表中已存在的数据。 当设置includeBaseData为true时，表示包含存量数据；设置includeBaseData为false时，表示不包含存量数据。

- 示例

```
public static void CreateGlobalIndex()
{
    OTSClient otsClient = Config.GetClient();
    Console.WriteLine("Start create globalIndex...");
    IndexMeta indexMeta = new IndexMeta(IndexName2);
    indexMeta.PrimaryKey = new List<string>() { Col2 };
    indexMeta.DefinedColumns = new List<string>() { Pk1 };
    CapacityUnit reservedThroughput = new CapacityUnit(0, 0);
    CreateGlobalIndexRequest request = new CreateGlobalIndexRequest(TableName, indexMeta)
;
    otsClient.CreateGlobalIndex(request);
    Console.WriteLine("Global Index is created,tableName: " + TableName + ",IndexName:" +
IndexName2);
 }
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  详情请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  ○ tableName需要设置为索引表名称。

  ○ 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  详情请参见多行数据操作。

- 参数

  使用GetRange接口读取索引表中数据时有如下注意事项：

  - tableName需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 示例

```
public static void GetRangeFromIndexTable()
{
    Console.WriteLine("Start getRange from index...");
    OTSClient otsClient = Config.GetClient();
    //指定第一主键Col1的值，进行扫描。
    PrimaryKey inclusiveStartPrimaryKey = new PrimaryKey
    {
        { Col1, new ColumnValue("Col1Value") },
        { Pk1,  ColumnValue.INF_MIN },
        { Pk2, ColumnValue.INF_MIN }
    };
    PrimaryKey exclusiveEndPrimaryKey = new PrimaryKey
    {
        { Col1, new ColumnValue("Col1Value") },
        { Pk1,  ColumnValue.INF_MAX },
        { Pk2, ColumnValue.INF_MAX }
    };
    GetRangeRequest request = new GetRangeRequest(IndexName, GetRangeDirection.Forward,
inclusiveStartPrimaryKey, exclusiveEndPrimaryKey);
    GetRangeResponse response = otsClient.GetRange(request);
    IList<Row> rows = response.RowDataList;
    PrimaryKey nextStartPrimaryKey = response.NextPrimaryKey;
    while (nextStartPrimaryKey != null)
    {
        request = new GetRangeRequest(TableName, GetRangeDirection.Forward, nextStartPr
imaryKey, exclusiveEndPrimaryKey);
        response = otsClient.GetRange(request);
        nextStartPrimaryKey = response.NextPrimaryKey;
        foreach (var row in response.RowDataList)
        {
            rows.Add(row);
        }
    }
    foreach (var row in rows)
    {
        PrintRow(row);
    }
    Console.WriteLine("TotalRowsRead: " + rows.Count);
}
private static void PrintRow(Row row)
{
    Console.WriteLine("-----------------");
    foreach (KeyValuePair<string, ColumnValue> entry in row.GetPrimaryKey())
    {
        Console.WriteLine(entry.Key + ":" + PrintColumnValue(entry.Value));
    }
    foreach (Column entry in row.GetColumns())
    {
        Console.WriteLine(entry.Name + ":" + PrintColumnValue(entry.Value));
    }
    Console.WriteLine("-----------------");
}
```

删除索引表（DeleteGlobalIndex）

使用DeleteIndex接口删除数据表上指定的索引表。

- 参数

参数	说明
mainTableName	数据表名称。
indexName	索引表名称。

- 示例

```
public static void DeleteGlobalIndex()
{
    OTSClient otsClient = Config.GetClient();
    Console.WriteLine("Start delete globalIndex...");
    DeleteGlobalIndexRequest request = new DeleteGlobalIndexRequest(TableName, IndexName)
;
    otsClient.DeleteGlobalIndex(request);
    DeleteGlobalIndexRequest request2 = new DeleteGlobalIndexRequest(TableName, IndexName
2);
    otsClient.DeleteGlobalIndex(request2);
    Console.WriteLine("Global Index is deleted,tableName: " + TableName + ",IndexName:" +
IndexName + "," + IndexName2);
}
```

7.9. 错误处理

本文主要为您介绍表格存储 .NET SDK的错误处理。

方式

表格存储 .NET SDK 目前采用 `异常` 的方式处理错误，如果调用接口没有抛出异常，则说明操作成功，否则失败。

> **说明** 批量相关接口例如 BatchGetRow 和 BatchWriteRow， 需要检查每个 row 的状态都是成功后才能保证整个接口调用是成功的。

异常

表格存储 .NET SDK 中有 OTSClientException 和 OTSServerException 两种异常，他们都最终继承自 Exception。

- OTSClientException：指 SDK 内部出现的异常，例如参数设置不对，返回结果解析失败等。
- OTSServerException：指服务器端的错误，它来自于对服务器错误信息的解析。OTSServerException 一般有以下几个成员：
  - HttpStatusCode：HTTP 返回码，例如 200、404等。
  - ErrorCode：表格存储返回的错误类型字符串。
  - ErrorMessage：表格存储返回的错误消息字符串。

- RequestId：用于唯一标识该次请求的 UUID。当您无法解决问题时，可以凭这个 RequestId 来请求表格存储开发工程师的帮助。

重试

- SDK 中出现错误时会自动重试。默认策略是最多重试3次，重试间隔最大2秒。
- 用户也可以通过修改 OTSClientConfig 中的 RetryPolicy 自定义重试策略。

8.PHP SDK

8.1. 前言

本文档主要介绍表格存储PHP SDK的安装和使用，适用4.0.0以上版本。并且假设您已经开通了阿里云表格存储服务，并创建了AccessKeyId和AccessKeySecret。

- 如果您还没有开通或者还不了解阿里云的表格存储服务，请登录表格存储的产品主页进行了解。
- 已创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。

特别注意

4.0.0以上版本SDK支持数据多版本和生命周期，但是该版本SDK不兼容2.x.x系列的SDK。

- 新增数据生命周期TTL
- 新增数据多版本
- 主键列自增

SDK下载

- SDK源码包
- GitHub

版本迭代详情请参见PHP SDK历史迭代版本。

版本

当前最新版本：5.0.0

兼容性

- 对于4.x.x系列的SDK兼容
- 对于2.x.x系列的SDK不兼容

变更内容

- 5.0.0
  - 新功能：支持局部事务
  - 新功能：支持多元索引
  - 新功能：支持全局二级索引
- 4.1.0

  支持Stream基础接口
- 4.0.0
  - 支持5.5以上PHP版本，包括5.5、5.6、7.0、7.1、7.2等版本，只支持64位的PHP系统，推荐使用PHP7。
  - 新功能：支持TTL设置，createTable、updateTable新增table_options参数。
  - 新功能：支持多版本，putRow、updateRow、deleteRow、batchGetRow均支持timestamp设置，getRow、getRange、BatchGet等接口支持max_versions过滤。
  - 新功能：支持主键列自增功能，接口新增return_type，返回新增primary_key，返回对应操作的primary_key。

- 变更：底层protobuf升级成谷歌官方版本protobuf-php库。
- 变更：各接口的primary_key变更成list类型，保证顺序性。
- 变更：各接口的attribute_columns变更成list类型，以支持多版本功能。

8.2. 安装

本文主要为您介绍表格存储PHP SDK包安装。

环境准备

- 64位PHP 5.5+（必须）

  通过**php -v**命令查看当前的PHP版本。

  由于表格存储里的整型是64位的，而32位PHP只能用string表示64位的整型，所以暂不支持32位PHP；由于Windows系统中PHP7之前的版本整型不是真正的64位，如果要使用Windows系统，请升级至PHP7或者自行改造，强烈建议使用PHP7以获得最佳性能。

- cURL扩展（建议）

  通过**php -m**命令查看cURL扩展是否已经安装好。

  > **说明**
  > - 在Ubuntu系统中，使用apt-get包管理器安装PHP的cURL扩展sudo apt-get install php-curl。
  > - 在CentOS系统中，使用yum包管理器安装PHP的cURL扩展sudo yum install php-curl。

- OpenSSL扩展（建议）

  当需要使用HTTPS时，请安装OpenSSL PHP扩展。

安装方式

- composer方式

  composer方式安装SDK的步骤如下：

  i. 在项目的根目录运行 `composer require aliyun/aliyun-tablestore-sdk-php` ，或者在composer.json中声明对阿里云tablestore SDK for PHP的依赖。

  ```
  "require": {
      "aliyun/aliyun-tablestore-sdk-php": "~5.0"
  }
  ```

  ii. 通过composer install安装依赖。安装完成后，目录结构如下：

  ```
  .
  ├── app.php
  ├── composer.json
  ├── composer.lock
  └── vendor
  ```

  其中app.php是用户的应用程序，vendor/目录下包含了所依赖的库。您需要在app.php中引入依赖。

```
require_once __DIR__ . '/vendor/autoload.php';
```

> **说明**
>
> - 如果您的项目中已经引用过autoload.php，则加入了SDK的依赖之后，不需要再次引入。
> - 如果使用composer出现网络错误，可以使用composer中国区的镜像。方法是在命令行执行 `composer config -g repo.packagist composer https://developer.aliyun.com/composer` 。

- 源码包

  如果需要源码包，请通过如下方式下载。

  - 通过GitHub选择相应版本并下载源码压缩文件。具体路径请参见GitHub。
  - 通过SDK包获取源码，具体路径请参见SDK源码包。

示例程序

Tablestore PHP SDK提供丰富的示例程序，方便用户参考或直接使用。

您可以通过如下两种方式获取示例程序。

- 下载Tablestore PHP SDK开发包后，解压后examples为示例程序。
- 访问Tablestore PHP SDK的GitHub项目。具体路径请参见aliyun-tablestore-php-sdk。

您可以通过如下步骤运行示例程序。

1. 解压下载的SDK包。
2. 修改examples目录中的ExampleConfig.php文件。

   ```
   EXAMPLE_END_POINT：是访问表格存储服务中Instance的服务地址，例如https://sun.cn-hangzhou.ots.aliyuncs.com。
   EXAMPLE_ACCESS_KEY_ID：是从阿里云获取的AccessKeyId。
   EXAMPLE_ACCESS_KEY_SECRET：是从阿里云获取的AccessKeySecret。
   EXAMPLE_INSTANCE_NAME：是运行示例程序使用的Instance，示例程序会在该Instance中进行操作。
   ```

3. 在examples目录中单独运行某个示例文件。

示例程序包含如下内容。

示例文件	示例内容
NewClient.php	展示了设置默认Client的用法。
NewClient2.php	展示了设置Client的自定义配置用法。
NewClientLogClosed.php	展示了Client关闭Log的用法。
NewClientLogDefined.php	展示了Client设置自定义Log的用法。
CreateTable.php	展示了CreateTable的用法。
DeleteTable.php	展示了DeleteTable的用法。

示例文件	示例内容
DescribeTable.php	展示了DescribeTable的用法。
ListTable.php	展示了ListTable的用法。
UpdateTable.php	展示了UpdateTable的用法。
ComputeSplitPointsBySize.php	展示了ComputeSplitPointsBySize的用法。
PutRow.php	展示了PutRow的用法。
PutRowWithColumnFilter.php	展示了PutRow条件更新的用法。
UpdateRow1.php	展示了UpdateRow中PUT的用法。
UpdateRow2.php	展示了UpdateRow中DELETE_ALL的用法。
UpdateRow3.php	展示了UpdateRow中DELETE的用法。
UpdateRowWithColumnFilter.php	展示了UpdateRow条件更新的用法。
GetRow.php	展示了GetRow的用法。
GetRow2.php	展示了GetRow中设置column_to_get的用法。
GetRowWithSingleColumnFilter.php	展示了GetRow进行条件过滤的用法。
GetRowWithMultipleColumnFilter.php	展示了GetRow进行复杂条件过滤的用法。
DeleteRow.php	展示了DeleteRow的用法。
DeleteRowWithColumnFilter.php	展示了DeleteRow进行条件删除的用法。
PKAutoIncrment.php	展示了自增列的完整用法。
BatchGetRow1.php	展示了BatchGetRow获取单表多行的用法。
BatchGetRow2.php	展示了BatchGetRow获取多表多行的用法。
BatchGetRow3.php	展示了BatchGetRow获取单表多行同时制定获取特定列的用法。
BatchGetRow4.php	展示了BatchGetRow如何处理返回结果的用法。
BatchGetRowWithColumnFilter.php	展示了BatchGetRow的同时进行条件过滤的用法。
BatchWriteRow1.php	展示了BatchWriteRow中多个PUT的用法。
BatchWriteRow2.php	展示了BatchWriteRow中多个UPDATE的用法。
BatchWriteRow3.php	展示了BatchWriteRow中多个DELETE的用法。

示例文件	示例内容
BatchWriteRow4.php	展示了BatchWriteRow中混合进行UPDATE、PUT、DELETE的用法。
BatchWriteRowWithColumnFilter.php	展示了BatchWriteRow的同时进行条件更新的用法。
GetRange1.php	展示了GetRange的用法。
GetRange2.php	展示了GetRange指定获取列的用法。
GetRange3.php	展示了GetRange指定获取行数限制的用法。
GetRangeWithColumnFilter.php	展示了GetRange同时进行条件过滤的用法。

8.3. 初始化

OTSClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、读写单行数据、读写多行数据等。使用PHP SDK发起请求，您需要初始化一个OTSClient实例，并根据需要修改OTSClientConfig的默认配置项。

确定Endpoint

Endpoint是阿里云表格存储服务各个实例的域名地址，目前支持下列形式。

示例	解释
http://sun.cn-hangzhou.ots.aliyuncs.com	HTTP协议，公网网络访问杭州区域的sun实例。
https://sun.cn-hangzhou.ots.aliyuncs.com	HTTPS协议，公网网络访问杭州区域的sun实例。

> **注意** 除了公网可以访问外，也支持私网地址。更多请参见服务地址。

请按照如下步骤获取实例的Endpoint：

1. 登录表格存储管理控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

配置密钥

要接入阿里云的表格存储服务，您需要拥有一个有效的访问密钥进行签名认证。目前支持下面三种方式：

- 阿里云账号的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 在阿里云官网注册阿里云账号。
  ii. 创建AccessKey ID和AccessKey Secret。具体操作，请参见获取AccessKey。
- 被授予访问表格存储权限的RAM用户的AccessKey ID和AccessKey Secret。创建步骤如下：
  i. 使用阿里云账号前往访问控制RAM，创建一个新的RAM用户或者使用已经存在的RAM用户。
  ii. 使用阿里云账号授予RAM用户访问表格存储的权限。

iii. RAM用户被授权后，即可使用自己的AccessKey ID和AccessKey Secret访问。

- 从STS获取的临时访问凭证。获取步骤如下：
  i. 应用的服务器通过访问RAM/STS服务，获取一个临时的AccessKey ID、AccessKey Secret和SecurityToken发送给使用方。
  ii. 使用方使用上述临时密钥访问表格存储服务。

初始化对接

获取到AccessKey ID和AccessKey Secret后，您可以按照如下步骤进行初始化对接。

1. 使用表格存储的Endpoint新建Client。

   使用示例

```
$otsClient = new Aliyun\OTS\OTSClient(array(
    'EndPoint' => "<your endpoint>",
    'AccessKeyID' => "<your access id>",
    'AccessKeySecret' => "<your access key>",
    'InstanceName' => "<your instance name>"
));
```

2. 配置OTSClient。

   如果您需要修改OTSClient的一些默认配置，请在构造OTSClient时传入对应参数，例如代理、连接超时、最大连接数等参数。 具体设置的参数见下表。

参数	描述	默认值
ConnectionTimeout	与OTS建立连接的最大延时。	2.0秒
StsToken	临时访问的token。	null
SocketTimeout	每次请求响应最大延时。	2.0秒，传输量比较大的时候，建议设置大些。
RetryPolicy	重试策略。	DefaultRetryPolicy，设置为null可以关闭。
ErrorLogHandler	Error级别日志处理函数，用来打印表格存储服务端返回错误时的日志。	defaultOTSErrorLogHandler，设置为null可以关闭。
DebugLogHandler	Debug级别日志处理函数，用来打印正常的请求和响应信息。	defaultOTSDebugLogHandler，设置为null可以关闭。

HTTPS

安装OpenSSL PHP扩展即可。

8.4. 表操作

8.4.1. 创建数据表

使用CreateTable接口创建数据表时，需要指定数据表的结构信息和配置信息，高性能实例中的数据表还可以根据需要设置预留读/写吞吐量。创建数据表的同时支持创建一个或者多个索引表。

> 说明
> - 创建数据表后需要几秒钟进行加载，在此期间对该数据表的读/写数据操作均会失败。请等待数据表加载完毕后再进行数据操作。
> - 创建数据表时必须指定数据表的主键。主键包含1个~4个主键列，每一个主键列都有名称和类型。

前提条件

- 已通过控制台创建实例。具体操作，请参见创建实例。
- 已初始化Client。具体操作，请参见初始化。

接口

```
/**
 * 创建数据表，并设置主键的个数、名称、顺序和类型，以及预留读写吞吐量、TTL和stream选项。
 * @api
 * @param [] $request 请求参数。
 * @return [] 返回为空。CreateTable成功时不返回任何信息，此处返回一个空的array，与其他API保持一致。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 */
public function createTable(array $request);
```

参数

- 请求参数

参数	说明

参数	说明
table_meta	数据表的结构信息，包括如下内容： ○ table_name：数据表名称。 ○ primary_key_schema：数据表的主键定义。更多信息，请参见主键和属性。 **说明** 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。 ■ 每一项的顺序是主键名，主键类型PrimaryKeyType，主键设置PrimaryKeyOption（可选）。 PrimaryKeyType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY三种，分别用PrimaryKeyTypeConst::CONST_INTEGER、PrimaryKeyTypeConst::CONST_STRING、PrimaryKeyTypeConst::CONST_BINARY表示。 PrimaryKeyOption可以是PK_AUTO_INCR（自增列），用PrimaryKeyOptionConst::CONST_PK_AUTO_INCR表示。更多信息，请参见主键列自增。 ■ 表格存储可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体参见表格存储数据模型和查询操作。 ■ 第一列主键作为分区键。分区键相同的数据会存放在同一个分区内，所以相同分区键下最好不要超过10 GB以上数据，否则会导致单分区过大，无法分裂。另外，数据的读/写访问最好在不同的分区键上均匀分布，有利于负载均衡。 ○ defined_column：预先定义一些非主键列以及其类型，可以作为索引表的属性列或索引列。

参数	说明
table_options	数据表的配置信息。更多信息，请参见数据版本和生命周期。 配置信息包括如下内容： ○ time_to_live：数据生命周期，即数据的过期时间。当数据的保存时间超过设置的数据生命周期时，系统会自动清理超过数据生命周期的数据。 数据生命周期至少为86400秒（一天）或-1（数据永不过期）。 创建数据表时，如果希望数据永不过期，可以设置数据生命周期为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 单位为秒。 **说明** 如果需要使用索引，则数据生命周期必须设置为-1（数据永不过期）。 ○ max_versions：最大版本数，即属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。 **说明** 如果需要使用索引，则最大版本数必须设置为1。 ○ deviation_cell_version_in_sec：有效版本偏差，即写入数据的时间戳与系统当前时间的偏差允许最大值。只有当写入数据所有列的版本号与写入时时间的差值在数据有效版本偏差范围内，数据才能成功写入。 属性列的有效版本范围为**[数据写入时间-有效版本偏差，数据写入时间+有效版本偏差)**。 创建数据表时，如果未设置有效版本偏差，系统会使用默认值86400；创建数据表后，可以通过UpdateTable接口动态修改有效版本偏差。 单位为秒。
reserved_throughput	为数据表配置预留读吞吐量或预留写吞吐量。 容量型实例中的数据表的预留读/写吞吐量只能设置为0，不允许预留。 默认值为0，即完全按量计费。 单位为CU。 ○ 当预留读吞吐量或预留写吞吐量大于0时，表格存储会根据配置为数据表预留相应资源，且数据表创建成功后，将会立即按照预留吞吐量开始计费，超出预留的部分进行按量计费。更多信息，请参见计费概述。 ○ 当预留读吞吐量或预留写吞吐量设置为0时，表格存储不会为数据表预留相应资源。

参数	说明
stream_spec	Stream相关设置（可选配置）。 ○ enable_stream：数据表是否打开Stream。 ○ expiration_time：Stream数据的过期时间，较早的修改记录将会被删除，单位为小时。 只有当设置enable_stream为true时才能设置此参数。
index_metas	索引表的结构信息，每个index_meta都包括如下内容： ○ name：索引表名称。 ○ primary_key：索引表的索引列，索引列为数据表主键和预定义列的任意组合。 ○ defined_column：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ index_update_mode：索引表更新模式，当前只支持IUM_ASYNC_INDEX。 ○ index_type：索引表类型，当前只支持IT_GLOBAL_INDEX。

- 请求格式

```
$result = $client->createTable([
    'table_meta' => [                   //设置数据表结构信息，必须配置。
        'table_name' => '<string>',
        'primary_key_schema' => [
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>, <PrimaryKeyOption>]
        ]
    ],
    'reserved_throughput' => [          //设置预留读写吞吐量，必须配置。
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'table_options' => [                //设置数据表配置信息，必须配置。
        'time_to_live' => <integer>,
        'max_versions' => <integer>,
        'deviation_cell_version_in_sec' => <integer>
    ],
    'stream_spec' => [
        'enable_stream' => true || false,
        'expiration_time' => <integer>
    ]
]);
```

- 响应参数

  返回为空，出错时会抛出异常。

- 结果格式

```
[]
```

示例

- 创建数据表（不带索引）

  创建一个有3个主键列，预留读/写吞吐量为(0, 0)的数据表，TTL永不过期，存储两个版本的数据，同时打开Stream。

```
//创建主键列的schema，包括主键的个数、名称和类型。
//第一个主键列为整数，名称是pk0，此主键列也是分区键。
//第二个主键列为字符串，名称是pk1。
//第三个主键列为二进制，名称是pk2。
$result = $client->createTable([
    'table_meta' => [
        'table_name' => 'SampleTable',
        'primary_key_schema' => [
            ['PK0', PrimaryKeyTypeConst::CONST_INTEGER],
            ['PK1', PrimaryKeyTypeConst::CONST_STRING],
            ['PK2', PrimaryKeyTypeConst::CONST_BINARY]
        ]
    ],
    'reserved_throughput' => [
        'capacity_unit' => [
            'read' => 0,
            'write' => 0
        ]
    ],
    'table_options' => [
        'time_to_live' => -1,
        'max_versions' => 2,
        'deviation_cell_version_in_sec' => 86400
    ],
    'stream_spec' => [
        'enable_stream' => true,
        'expiration_time' => 24
    ]
]);
```

- 创建数据表（带索引）

```
$request = array (
    'table_meta' => array (
        'table_name' => self::$tableName, //数据表名称为testGlobalTableName。
        'primary_key_schema' => array (
            array('PK0', PrimaryKeyTypeConst::CONST_INTEGER),
            array('PK1', PrimaryKeyTypeConst::CONST_STRING)
        ),
        'defined_column' => array(
            array('col1', DefinedColumnTypeConst::DCT_STRING),
            array('col2', DefinedColumnTypeConst::DCT_INTEGER)
        )
    ),
    'reserved_throughput' => array (
        'capacity_unit' => array (
            'read' => 0, //设置预留读写吞吐量为0个读CU和0个写CU。
            'write' => 0
        )
    ),
    'table_options' => array(
        'time_to_live' => -1,    //数据的过期时间，单位为秒，-1表示永不过期。带索引表的数据表数据生命周期必须设置为-1。
        'max_versions' => 1,     //保存的最大版本数，1表示每列上最多保存一个版本即保存最新的版本。带索引表的数据表最大版本数必须设置为1。
        'deviation_cell_version_in_sec' => 86400  //数据有效版本偏差，单位为秒。
    ),
    'index_metas' => array(
        array(
            'name' => self::$indexName1,
            'primary_key' => array('col1'),
            'defined_column' => array('col2')
        ),
        array(
            'name' => self::$indexName2,
            'primary_key' => array('PK1'),
            'defined_column' => array('col1', 'col2')
        )
    )
);
$this->otsClient->createTable($request);
```

8.4.2. 更新表

本文主要为您介绍更新表的接口。

UpdateTable

API说明：UpdateTable

表格存储支持更新表的预留读/写吞吐量（ReservedThroughput）、配置信息（TableOptions）以及stream配置（StreamSpecification）。

关于 ReservedThroughput、TableOptions 以及 StreamSpecification，在本章开始的“创建表”部分已经有过介绍。ReservedThroughput 的调整有时间间隔限制，目前为 1 分钟。

接口

```
    /**
     * 更新一个表，包括这个表的预留读写吞吐量，配置信息，stream配置。
     * 这个API可以用来上调或者下调表的预留读写吞吐量。
     * @api
     * @param [] $request 请求参数
     * @return [] 请求返回
     * @throws OTSClientException 当参数检查出错或服务端返回校验出错时
     * @throws OTSServerException 当OTS服务端返回错误时
     */
    public function updateTable(array $request);
```

请求格式

```
$result = $client->updateTable([
    'table_name' => '<string>',         // REQUIRED
    'reserved_throughput' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'table_options' => [
        'time_to_live' => <integer>,
        'max_versions' => <integer>,
        'deviation_cell_version_in_sec' => <integer>
    ],
    'stream_spec' => [
        'enable_stream' => true || false,
        'expiration_time' => <integer>
    ]
]);
```

请求格式说明

- 和 CreateTable的区别只有tableMeta。除了TableMeta以外都是可以更新的，而且含义和CreateTable保持一致。同时除了table_name外，都是可选的。
- table_name 表名（必须设置）。
- reserved_throughput 表的预留读/写吞吐量配置，与计费相关（可选配置）。
  - capacity_unit 当预留读/写吞吐量大于 0 时，会按照预留量和持续时间进行计费，超出预留的部分进行按量计费。默认预留读写吞吐量为 0，即完全按量计费，如果要设置为大于 0 的值，请仔细阅读表格存储的计费相关文档，以免产生未期望的费用。容量型实例的预留读/写吞吐量只能设置为 0，不允许预留。
    - read 预留读吞吐量
    - write 预留写吞吐量
- table_options TableOptions 包含表的 TTL、MaxVersions 和 MaxTimeDeviation 配置（可选配置）。

- time_to_live TimeToLive，数据存活时间，单位秒。
  - 表格存储的新版 API 支持数据自动过期。如果期望永不过期，TTL 可设置为 -1。
  - 数据是否过期是根据“数据的时间戳“、“当前时间”、“表的 TTL”三者进行判断的。当“当前时间”减去“数据的时间戳”大于“表的 TTL”时，数据会过期并被表格存储服务器端清理。有关数据的时间戳的更多信息。
  - 当设置 TTL 后，由于判断过期涉及数据的时间戳，如果用户指定时间戳写入，且指定的时间戳严重偏离当前时间，那么可能导致未预料的数据过期行为。例如指定的数据时间戳很小，可能导致数据一写入就被过期回收了。当指定的数据时间戳很大时，又可能导致期望过期的数据过期不掉。因此在使用 TTL 功能时需要注意写入时是否指定了时间戳，以及指定的时间戳是否合理。
- max_versions 每个属性列保留的最大版本数。
  - MaxVersions 即用来指定每个属性列最多保存多少个版本的数据，如果写入的版本数超过 MaxVersions，服务端只会保留版本号最大的 MaxVersions 个版本。
- deviation_cell_version_in_sec 指定版本写入数据时所指定的版本与系统当前时间偏差允许的最大值，单位为秒。
  - 表格存储支持多版本，默认情况下系统会为新写入的数据生成一个版本号，是写入时间的毫秒单位时间戳，数据自动过期功能需要根据这个时间戳判断数据是否过期。另一方面，用户可以指定写入数据的时间戳，因此如果用户写入的时间戳非常小，与当前时间偏差已经超过了表上设置的 TTL 时间，写入的数据会立即过期。出于保护的目的，在表上增加了 MaxTimeDeviation 设置，限制写入数据的时间戳与系统当前时间的偏差，该值的单位为秒，可在建表时指定，也可通过 UpdateTable 接口修改。

- stream_spec Stream相关设置（可选配置）。
  - enable_stream Stream 是否打开
  - expiration_time Stream 数据的过期时间，较早的修改记录将会被删除，单位小时

结果格式

```
[
    'capacity_unit_details' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ],
        'last_increase_time' => <integer>,
        'last_decrease_time' => <integer>
    ],
    'table_options' => [
        'time_to_live' => <integer>,
        'max_versions' => <integer>,
        'deviation_cell_version_in_sec => <integer>
    ],
    'stream_details' => [
        'enable_stream' => true || false,
        'stream_id' => '<string>',
        'expiration_time' => <integer>,
        'last_enable_time' => <integer>
    ]
]
```

结果格式说明

- capacity_unit_details 表的预留读/写吞吐量配置，与计费相关。
  - capacity_unit 当预留读/写吞吐量大于 0 时，会按照预留量和持续时间进行计费，超出预留的部分进行按量计费。默认预留读写吞吐量为 0，即完全按量计费，如果要设置为大于 0 的值，请仔细阅读表格存储的计费相关文档，以免产生未期望的费用。容量型实例的预留读/写吞吐量只能设置为 0，不允许预留。
    - read 预留读吞吐量
    - write 预留写吞吐量
  - last_increase_time 最近一次上调该表的预留读/写吞吐量设置的时间，使用 UTC 秒数表示。
  - last_decrease_time 最近一次下调该表的预留读/写吞吐量设置的时间，使用 UTC 秒数表示。
- table_options TableOptions 包含表的 TTL、MaxVersions 和 MaxTimeDeviation 配置。和请求一致。
- stream_details 表的stream信息。
  - enable_stream 该表是否打开stream
  - stream_id 该表的stream的id
  - expiration_time 该表的stream的过期时间，较早的修改记录将会被删除，单位小时
  - last_enable_time 该stream的打开的时间

示例

更新表的 CU 值为读 1，写 2。

```
$result = $client->updateTable([
    'table_name' => 'SampleTable',
    'reserved_throughput' => [
        'capacity_unit' => [
            'read' => 1,            // 可以单独更新读或者写
            'write' => 2
        ]
    ]
]);
```

更新表的TTL为一天（86400），保留版本2， 最大偏差10s。

```
$result = $client->updateTable([
    'table_name' => 'SampleTable',
    'table_options' => [
        'time_to_live' => 86400,
        'max_versions' => 2,
        'deviation_cell_version_in_sec' => 10
    ]
]);
```

打开表的Stream，并设置过期时间24小时。

```
    $result = $client->updateTable([
        'table_name' => 'SampleTable',
        'stream_spec' => [
            'enable_stream' => true,
            'expiration_time' => 24
        ]
    ]);
```

8.4.3. 列出表名称

使用ListTable接口获取当前实例下已创建的所有表的表名。

> **说明** API说明请参见ListTable。

接口

```
/**
 * 获取该实例下所有的表名。
 * @api
 * @param [] $request 请求参数，为空。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 */
public function listTable(array $request);
```

请求格式

请求参数为空，即无需设置请求参数。

```
$result = $client->listTable([]);
```

结果格式

返回结果是一个String类型的list，list中的每一项均是一个表名。

```
[
    '<string>',
    '<string>',
    '<string>'
]
```

示例

获取实例下所有表的表名。

```
$result = $otsClient->listTable([]);
```

8.4.4. 指定大小计算分片

使用ComputeSplitsBySize接口可以将全表数据逻辑上划分成若干接近指定大小的分片，并返回这些分片之间的分割点以及分片所在机器的提示。一般用于计算引擎规划并发度等执行计划。

> **说明** API说明请参见ComputeSplitPointsBySize。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

接口

```
    /**
     * 将全表的数据在逻辑上划分成接近指定大小的若干分片，返回这些分片之间的分割点以及分片所在机器的提示
。
     * 一般用于计算引擎规划并发度等执行计划。
     * @api
     * @param [] $request 请求参数。
     * @return [] 请求返回。
     * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
     * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
     */
    public function computeSplitPointsBySize(array $request)
```

参数

- 请求参数

参数	说明
table_name	数据表名称。
split_size	每个分片的近似大小。 单位为百兆（即100 MB）。

- 请求格式

```
$result = $client->ComputeSplitsBySize([
    'table_name' => '<string>', //设置数据表名称，必须设置。
    'split_size' => <integer>   //设置分片大小，必须设置。
]);
```

- 响应参数

参数	说明

参数	说明
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写单元。 ○ read：读吞吐量 ○ write：写吞吐量
primary_key_schema	数据表的主键定义，与创建数据表时的主键定义相同。
splits	分片之间的分割点，包括如下内容： ○ lower_bound：主键的区间最小值。 此值可以传递给GetRange用于范围读数据。 ■ 每一项的顺序是主键名、主键值PrimaryKeyValue、主键类型PrimaryKeyType。 ■ PrimaryKeyType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、INF_MIN(-inf)、 INF_MAX(inf)五种，分别用PrimaryKeyTypeConst::CONST_INTEGER、PrimaryKeyTypeConst::CONST_STRING、PrimaryKeyTypeConst::CONST_BINARY、PrimaryKeyTypeConst::CONST_INF_MIN、PrimaryKeyTypeConst::CONST_INF_MAX表示。 ○ upper_bound：主键的区间最大值。格式与lower_bound相同。 此值可以传递给GetRange用于范围读数据。 ○ location：分割点所在机器的提示，可以为空。

- 结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key_schema' => [
        ['<string>', <PrimaryKeyType>],
        ['<string>', <PrimaryKeyType>, <PrimaryKeyOption>]
    ]
    'splits' => [
        [
            'lower_bound' => [
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'upper_bound' => [
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'location' => '<string>'
        ],
        // ...
    ]
]
```

示例

将全表的数据在逻辑上划分成接近100 MB的若干分片。

```
$result = $client->ComputeSplitsBySize([
    'table_name' => 'MyTable',
    'split_size' => 1
]);
foreach($result['splits'] as $split) {
    print_r($split['location']);
    print_r($split['lower_bound']);
    print_r($split['upper_bound']);
}
```

8.4.5. 查询表描述信息

使用DescribeTable接口可以查询指定表的结构、预留读/写吞吐量详情等信息。

> **说明** API说明请参见DescribeTable。

接口

```
/**
 * 获取一个表的信息，包括表的结构信息、配置信息、预留读/写吞吐量详情和Stream设置信息。
 * @api
 * @param [] $request 请求参数。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 */
public function describeTable(array $request);
```

参数

- 请求参数

参数	说明
table_name	表名。

- 请求格式

```
$result = $client->describeTable([
    'table_name' => '<string>', //设置表名，必须设置。
]);
```

- 返回参数

参数	说明
table_meta	表的结构信息，包括如下内容： ○ table_name：表名称。 ○ primary_key_schema：表的主键定义，与创建数据表时的主键定义相同。
capacity_unit_details	表的预留读/写吞吐量配置详情，包括如下内容： ○ capacity_unit表示表的预留读/写吞吐量配置信息，与计费相关。 ■ read：预留读吞吐量 ■ write：预留写吞吐量 ○ last_increase_time：最近一次上调该数据表的预留读/写吞吐量设置的时间，使用UTC秒数表示。 ○ last_decrease_time：最近一次下调该数据表的预留读/写吞吐量设置的时间，使用UTC秒数表示。
table_options	表的配置信息，包括如下内容： ○ time_to_live：数据生命周期，即数据的过期时间。 ○ max_versions：最大版本数，即属性列能够保留数据的最大版本个数。 ○ deviation_cell_version_in_sec：有效版本偏差，即写入数据的时间戳与系统当前时间的偏差允许最大值。

参数	说明
stream_details	数据表的Stream信息，包括如下内容： ○ enable_stream：数据表是否打开Stream。 ○ stream_id：数据表的Stream ID。 ○ expiration_time：Stream的过期时间，较早的修改记录将会被删除，单位为小时。 ○ last_enable_time：Stream的打开的时间。

- 结果格式

```
[
    'table_meta' => [
        'table_name' => '<string>',
        'primary_key_schema' => [
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>, <PrimaryKeyOption>]
        ]
    ],
    'capacity_unit_details' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ],
        'last_increase_time' => <integer>,
        'last_decrease_time' => <integer>
    ],
    'table_options' => [
        'time_to_live' => <integer>,
        'max_versions' => <integer>,
        'deviation_cell_version_in_sec => <integer>
    ],
    'stream_details' => [
        'enable_stream' => true || false,
        'stream_id' => '<string>',
        'expiration_time' => <integer>,
        'last_enable_time' => <integer>
    ]
]
```

示例

获取表的描述信息。

```
$result = $client->describeTable([
        'table_name' => 'mySampleTable',
        ]);
        var_dump($result);
```

8.4.6. 删除数据表

使用DeleteTable接口删除当前实例下指定数据表。

> **说明** API说明请参见DeleteTable。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表。
- 已删除数据表上的索引表和多元索引。

接口

```
/**
 * 根据数据表名称删除数据表。
 * @api
 * @param [] $request 请求参数
 * @return [] 返回为空。DeleteTable成功时不返回任何信息，此处返回一个空的array，与其他API保持一致。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 */
public function deleteTable(array $request);
```

参数

- 请求参数

参数	说明
table_name	数据表名称。

- 请求格式

```
$result = $client->deleteTable([
    'table_name' => '<string>', // REQUIRED
]);
```

- 响应参数

  返回为空，出错会抛出异常。

- 结果格式

```
    []
```

示例

删除指定数据表。

```
$result = $otsClient->deleteTable([
    'table_name' => 'MyTable'
]);
```

8.4.7. 主键列自增

设置非分区键的主键列为自增列后，在写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。

> **说明** 从PHP SDK 4.0.0版本开始支持主键列自增功能。

前提条件

已初始化Client，详情请参见初始化。

使用流程

1. 创建表时，将非分区键的主键列设置为自增列。

   只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号长整型。

2. 写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符。

   如果需要获取写入数据后系统自动生成的自增列的值，将ReturnType设置为ReturnTypeConst::CONST_PK，可以在数据写入成功后返回自增列的值。

   查询数据时，需要完整的主键值。通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为ReturnTypeConst::CONST_PK可以获取完整的主键值。

示例

主键自增列功能主要涉及创建表（CreateTable）和写数据（PutRow、UpdateRow和BatchWriteRow）两类接口。

1. 创建表

   创建表时，只需将自增的主键属性设置为PrimaryKeyOptionConst::CONST_PK_AUTO_INCR。

```
function createTable($otsClient)
{
    $request = [
        'table_meta' => [
            'table_name' => 'table_name',        //设置数据表名称。
            'primary_key_schema' => [
                ['PK_1', PrimaryKeyTypeConst::CONST_STRING],    //第一个主键列（又叫分区键）名称为PK_1，类型为STRING。
                ['PK_2', PrimaryKeyTypeConst::CONST_INTEGER, PrimaryKeyOptionConst::CONST_PK_AUTO_INCR]
                //第二个主键列名称为PK_2，类型为INTEGER，并且设置为主键列。
            ]
        ],
        'reserved_throughput' => [
            'capacity_unit' => [          //预留读写吞吐量设置为0个读CU和0个写CU。
                'read' => 0,
                'write' => 0
            ]
        ],
        'table_options' => [
            'time_to_live' => -1,              //设置为数据永不过期。
            'max_versions' => 1,               //只保存一个版本。
            'deviation_cell_version_in_sec' => 86400    //数据有效版本偏差，单位秒。
        ]
    ];
    $otsClient->createTable($request);
}
```

2. 写数据

   写入数据时，无需为自增列设置具体值，只需将自增列的值设置为占位符PrimaryKeyTypeConst::CONST_PK_AUTO_INCR。

```
function putRow($otsClient)
{
    $row = [
        'table_name' => 'table_name',
        'primary_key' => [
            ['PK_1', 'Hangzhou'],                          //主键名和主键值，此处为list。
            ['PK_2', null, PrimaryKeyTypeConst::CONST_PK_AUTO_INCR]     //主键自增列，此处无需填入具体值，只需要一个占位符PrimaryKeyTypeConst::CONST_PK_AUTO_INCR，表格存储会自动生成此值。
        ],
        'attribute_columns' => [              //属性列，此处为list。
            ['name', 'John'],                  //[属性名，属性值，属性类型，时间戳]，未设置时可以忽略。
            ['age', 20],
            ['address', 'Alibaba'],
            ['product', 'OTS'],
            ['married', false]
        ],
        'return_content' => [
            'return_type' => ReturnTypeConst::CONST_PK      //返回自增列的主键值需要设置return_type。
        ]
    ];
    $ret = $otsClient->putRow($row);
    print_r($ret);
    $primaryKey = $ret['primary_key'];   //此处获取到的primaryKey可以传递给GetRow、UpdateRow、DeleteRow等API使用。
    return $primaryKey;
}
```

8.4.8. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过PutRow、UpdateRow、DeleteRow或BatchWriteRow接口更新数据时，可以使用条件更新检查行存在性条件和列条件，只有满足条件时才能更新成功。

条件更新包括行存在性条件和列条件。

- 行存在性条件：包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。

  对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。

- 列条件：包括SingleColumnValueCondition和CompositeColumnValueCondition，是基于某一列或者某些列的列值进行条件判断。

- SingleColumnValueCondition支持一列和一个常量比较。不支持两列或者两个常量相比较。
- CompositeColumnValueCondition的内节点为逻辑运算，子条件可以是SingleColumnValueCondition或CompositeColumnValueCondition。

条件更新可以实现乐观锁功能，即在更新某行时，先获取某列的值，假设为列A，值为1，然后设置条件**列A = 1**，更新行使**列A = 2**。如果更新失败，表示有其他客户端已成功更新该行。

限制

条件更新的列条件支持关系运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。

参数

条件更新可以用于PutRow、UpdateRow、DeleteRow和BatchWriteRow的condition中。

```
'condition' => [
    'row_existence' => <RowExistenceExpectation>
    'column_condition' => <ColumnCondition>
]
```

当只有行存在性条件时，可以简写为如下结构。

```
'condition' => <RowExistenceExpectation>
```

- SingleColumnValueCondition结构

```
[
    'column_name' => '<string>',
    'value' => <ColumnValue>,
    'comparator' => <ComparatorType>
    'pass_if_missing' => true || false
    'latest_version_only' => true || false
]
```

- CompositeColumnValueFilter结构

```
[
    'logical_operator' => <LogicalOperator>
    'sub_conditions' => [
        <ColumnCondition>,
        <ColumnCondition>,
        <ColumnCondition>,
        // other conditions
    ]
]
```

参数	说明

参数	说明
row_existence	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括IGNORE、EXPECT_EXIST和EXPECT_NOT_EXIST，分别用RowExistenceExpectationConst::CONST_IGNORE、RowExistenceExpectationConst::CONST_EXPECT_EXIST、RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST表示。 • IGNORE：表示忽略，不做任何存在性检查。 • EXPECT_EXIST：表示期望存在，如果该行存在，则满足条件；如果该行不存在，则不满足条件。 • EXPECT_NOT_EXIST：期望行不存在，如果该行不存在，则满足条件；如果该行存在，则不满足条件。
column_name	列的名称。
value	列的对比值。 格式为[Value, Type]。Type可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、BOOLEAN、DOUBLE五种，分别用ColumnTypeConst::CONST_INTEGER、ColumnTypeConst::CONST_STRING、ColumnTypeConst::CONST_BINARY、ColumnTypeConst::CONST_BOOLEAN、ColumnTypeConst::CONST_DOUBLE表示，其中BINARY不可省略，其他类型均可省略。 当Type不是BINARY时，可以简写为Value。
comparator	对列值进行比较的关系运算符，类型详情请参见ComparatorType。 关系运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=），分别用ComparatorTypeConst::CONST_EQUAL、ComparatorTypeConst::CONST_NOT_EQUAL、ComparatorTypeConst::CONST_GREATER_THAN、ComparatorTypeConst::CONST_GREATER_EQUAL、ComparatorTypeConst::CONST_LESS_THAN、ComparatorTypeConst::CONST_LESS_EQUAL表示。
logical_operator	对多个条件进行组合的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR，分别用LogicalOperatorConst::CONST_NOT、LogicalOperatorConst::CONST_AND、LogicalOperatorConst::CONST_OR表示。 逻辑运算符不同可以添加的子条件个数不同。 • 当逻辑运算符为NOT时，只能添加一个子条件。 • 当逻辑运算符为AND或OR时，必须至少添加两个子条件。
pass_if_missing	当列在某行中不存在时，条件检查是否通过。类型为bool值，默认值为true，表示如果列在某行中不存在时，则条件检查通过，该行满足更新条件。 当设置pass_if_missing为false时，如果列在某行中不存在时，则条件检查不通过，该行不满足更新条件。

参数	说明
latest_version_only	当列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当设置latest_version_only为false时，如果列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就条件检查通过，该行满足更新条件。

示例

- 构造SingleColumnValueCondition。

```
    //设置条件更新，当Col0的值为0时，条件检查通过。
    $column_condition = [
        'column_name' => 'Col0',
        'value' => 0,
        'comparator' => ComparatorTypeConst::CONST_EQUAL
        'pass_if_missing' => false                          //如果不存在Col0列，条件检查不通过。
        'latest_version_only' => true                       //只判断最新版本。
    ];
```

- 构造CompositeColumnValueCondition。

```
//composite1的条件为(Col0 == 0) AND (Col1 > 100)。
    $composite1 = [
        'logical_operator' => LogicalOperatorConst::CONST_AND,
        'sub_conditions' => [
            [
                'column_name' => 'Col0',
                'value' => 0,
                'comparator' => ComparatorTypeConst::CONST_EQUAL
            ],
            [
                'column_name' => 'Col1',
                'value' => 100,
                'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
            ]
        ]
    ];
    //composite2的条件为( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10)。
    $composite2 = [
        'logical_operator' => LogicalOperatorConst::CONST_OR,
        'sub_conditions' => [
            $composite1,
            [
                'column_name' => 'Col2',
                'value' => 10,
                'comparator' => ComparatorTypeConst::CONST_LESS_EQUAL
            ]
        ]
    ];
```

- 通过Condition实现乐观锁机制，递增一列。

```
//读取一行数据。
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ //主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'max_versions' => 1
];
$response = $otsClient->getRow ($request);
$columnMap = getColumnValueAsMap($response['attribute_columns']);
$col0Value = $columnMap['col0'][0][1];
//条件更新Col0列，使列值加1。
$request = [
    'table_name' => 'MyTable',
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST,
        'column_condition' => [                    //满足条件，则更新数据。
            'column_name' => 'col0',
            'value' => $col0Value,
            'comparator' => ComparatorTypeConst::CONST_EQUAL
        ]
    ],
    'primary_key' => [ //主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'update_of_attribute_columns'=> [
        'PUT' => [
            ['col0', $col0Value+1]
        ]
    ]
];
$response = $otsClient->updateRow ($request);
```

8.4.9. 局部事务

使用局部事务功能，创建数据范围在一个分区键值内的局部事务。对局部事务中的数据进行读写操作后，可以根据实际提交或者丢弃局部事务。局部事务通过悲观锁（Pessimistic Lock）实现并发控制。

目前局部事务功能处于邀测中，默认关闭。如果需要使用该功能，请提交工单进行申请或者加入钉钉群23307953（表格存储技术交流群-2）进行咨询。

使用局部事务可以指定某个分区键值内的操作是原子的，对分区键值内的数据进行的操作要么全部成功要么全部失败，并且所提供的隔离级别为读已提交。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

1. 使用startLocalTransaction在指定的分区键值创建一个局部事务，并获取局部事务ID。
2. 对局部事务范围内的数据进行读写操作。

   支持对局部事务进行操作的接口为GetRow、PutRow、DeleteRow、UpdateRow、BatchWriteRow和GetRange。
3. 使用commitTransaction提交局部事务或者使用abortTransaction丢弃局部事务。

限制

- 每个局部事务从创建开始生命周期最长为60秒。

  如果超过60秒未提交或丢弃局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
- 如果创建局部事务时超时，此请求可能在表格存储服务端已执行成功，此时用户需要等待该局部事务超时后重新创建。
- 未提交的局部事务可能失效，如果出现此情况，需要重试该局部事务内的操作。
- 在局部事务中读写数据有如下限制：
  - 不能使用局部事务ID访问局部事务范围（即创建时使用的分区键值）以外的数据。
  - 同一个局部事务中所有写请求的分区键值必须与创建局部事务时的分区键值相同，读请求则无此限制。
  - 一个局部事务同时只能用于一个请求中，在使用局部事务期间，其它使用此局部事务ID的操作均会失败。
  - 每个局部事务中两次读写操作的最大间隔为60秒。

    如果超过60秒未操作局部事务，表格存储服务端会认为此局部事务超时，并将局部事务丢弃。
  - 每个局部事务中写入的数据量最大为4 MB，按正常的写请求数据量计算规则累加。
  - 如果在局部事务中写入了未指定版本号的Cell，该Cell的版本号会在写入时（而非提交时）由表格存储服务端自动生成，生成规则与正常写入一个未指定版本号的Cell相同。
  - 如果BatchWriteRow请求中带有局部事务ID，则此请求中所有行只能操作该局部事务ID对应的表。
  - 在使用局部事务期间，对应分区键值的数据相当于被锁上，只有持有局部事务ID在局部事务范围内的写请求才会成功，其它不持有局部事务ID在局部事务范围内的写请求均会失败。在局部事务提交、丢弃或超时后，对应的锁也会被释放。
  - 带有局部事务ID的读写请求失败不会影响局部事务本身的存活情况，您可以按照正常的无局部事务ID的读写请求重试规则进行重试，或者主动丢弃当前局部事务。

接口

- startLocalTransaction

```
/**
 * 创建局部事务，获取局部事务ID。
 * @api
 * @param [] $request
 *              请求参数：数据表名称、分区键。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 * @example "src/examples/StartLocalTransaction.php" 50
 */
public function startLocalTransaction(array $request)
```

- commitTransaction

```
/**
 * 提交局部事务。
 * @api
 *
 * @param [] $request
 *            请求参数：局部事务ID。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 * @example "src/examples/CommitTransaction.php" 50
 */
public function commitTransaction(array $request)
```

- abortTransaction

```
/**
 * 丢弃局部事务。
 * @api
 *
 * @param [] $request
 *           请求参数：局部事务ID。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时。
 * @throws OTSServerException 当OTS服务端返回错误时。
 * @example "src/examples/AbortTransaction.php" 20
 */
public function abortTransaction(array $request)
```

参数

参数	说明
table_name	数据表名称。
key	数据表分区键。 创建局部事务时，只需要指定局部事务对应的分区键值。
primary_key	数据表主键。 创建局部事务后，对局部事务范围内的数据进行读写操作时，需要指定完整主键。
transaction_id	局部事务ID，用于唯一标识一个局部事务。 创建局部事务后，操作局部事务时均需要带上局部事务ID。

示例

1. 调用startLocalTransaction方法使用指定分区键值创建一个局部事务，并获取局部事务ID。

```
$response = $this->otsClient->startLocalTransaction (array (
    'table_name' => 'TransactionTable',
    'key' => array(  //主键为[PK0:INTEGER,PK1:STRING]。
        array('PK0', 0)
    )
));
```

2. 对局部事务范围内的数据进行读写操作。

   对局部事务范围内数据的读写操作与正常读写数据操作基本相同，只需填入局部事务ID即可。

```
$updateRequest = array(
    'table_name' => 'TransactionTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => array (
        array('PK0', 0),
        array('PK1', '1')
    ),
    'update_of_attribute_columns'=> array(
        'PUT' => array (
            array('attr0', 'new value')
        )
    ),
    'transaction_id' => $response['transaction_id']
);
$this->otsClient->updateRow($updateRequest);
```

3. 提交或丢弃局部事务。
   - 提交局部事务，使局部事务中的所有数据修改生效。

```
$this->otsClient->commitTransaction(array(
    'transaction_id' => $response['transaction_id']
));
```

   - 丢弃局部事务，局部事务中的所有数据修改均不会应用到原有数据。

```
$this->otsClient->abortTransaction(array(
    'transaction_id' => $response['transaction_id']
));
```

8.4.10. 过滤器

在服务端对读取结果再进行一次过滤，根据过滤器（Filter）中的条件决定返回的行。使用过滤器后，只返回符合条件的数据行。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

使用方法

在通过GetRow、BatchGetRow或GetRange接口查询数据时，可以使用过滤器只返回符合条件的数据行。

过滤器目前包括SingleColumnValueFilter、SingleColumnValueRegexFilter和CompositeColumnValueFilter。

- SingleColumnValueFilter：只判断某个参考列的列值。
- SingleColumnValueRegexFilter：支持对类型为String的列值，使用正则表达式进行子字符串匹配，然后根据实际将匹配到的子字符串转换为String、Integer或者Double类型，再对子值使用过滤器进行过滤。
- CompositeColumnValueFilter：根据多个参考列的列值的判断结果进行逻辑组合，决定是否过滤某行。

> **说明** 关于过滤器的更多信息，请参见功能介绍中的过滤器。

限制

- 过滤器的条件支持算术运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），最多支持10个条件的组合。
- 过滤器中的参考列必须在读取的结果内。如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。
- 在GetRow、BatchGetRow和GetRange接口中使用过滤器不会改变接口的原生语义和限制项。

  使用GetRange接口时，会受到一次扫描数据的行数不能超过5000行或者扫描数据的数据大小不能大于4 MB的限制。

  当在该次扫描的5000行或者4 MB数据中没有满足过滤器条件的数据时，得到的Response中的Rows为空，但是next_start_primary_key可能不为空，此时需要使用next_start_primary_key继续读取数据，直到next_start_primary_key为空。

参数

过滤器可以用于GetRow、BatchGetRow和GetRange接口的column_filter中。

```
'column_filter' => <ColumnFilter>
```

- SingleColumnValueFilter结构

  SingleColumnValueFilter支持一列（可以是主键列）和一个常量比较。不支持两列或者两个常量相比较。

  ```
  [
      'column_name' => '<string>',
      'value' => <ColumnValue>,
      'comparator' => <ComparatorType>
      'pass_if_missing' => true || false
      'latest_version_only' => true || false
  ]
  ```

- CompositeColumnValueFilter结构

  CompositeColumnValueFilter是一个树形结构，内节点为逻辑运算（logical_operator），叶节点为SingleColumnValueFilter。

```
[
    'logical_operator' => <LogicalOperator>
    'sub_filters' => [
        <ColumnFilter>,
        <ColumnFilter>,
        <ColumnFilter>,
        // other conditions
    ]
]
```

参数	说明
column_name	过滤器中参考列的名称。
value	过滤器中参考列的对比值。 格式为[Value, Type]。Type可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、BOOLEAN、DOUBLE五种，分别用ColumnTypeConst::CONST_INTEGER、ColumnTypeConst::CONST_STRING、ColumnTypeConst::CONST_BINARY、ColumnTypeConst::CONST_BOOLEAN、ColumnTypeConst::CONST_DOUBLE表示，其中BINARY不可省略，其他类型均可省略。 当Type不是BINARY时，可以简写为Value。
comparator	过滤器中的算术运算符，类型详情请参见ComparatorType。 算术运算符包括EQUAL（=）、NOT_EQUAL（!=）、GREATER_THAN（>）、GREATER_EQUAL（>=）、LESS_THAN（<）和LESS_EQUAL（<=），分别用ComparatorTypeConst::CONST_EQUAL、ComparatorTypeConst::CONST_NOT_EQUAL、ComparatorTypeConst::CONST_GREATER_THAN、ComparatorTypeConst::CONST_GREATER_EQUAL、ComparatorTypeConst::CONST_LESS_THAN、ComparatorTypeConst::CONST_LESS_EQUAL表示。
logical_operator	过滤器中的逻辑运算符，类型详情请参见LogicalOperator。 逻辑运算符包括NOT、AND和OR，分别用LogicalOperatorConst::CONST_NOT、LogicalOperatorConst::CONST_AND、LogicalOperatorConst::CONST_OR表示。
pass_if_missing	当参考列在某行中不存在时，是否返回该行。类型为bool值，默认值为true，表示如果参考列在某行中不存在，则返回该行。 当pass_if_missing设置为false时，如果参考列在某行中不存在，则不返回该行。
latest_version_only	当参考列存在多个版本的数据时，是否只使用最新版本的值做比较。类型为bool值，默认值为true，表示如果参考列存在多个版本的数据时，则只使用该列最新版本的值进行比较。 当latest_version_only设置为false时，如果参考列存在多个版本的数据时，则会使用该列的所有版本的值进行比较，此时只要有一个版本的值满足条件，就返回该行。

参数	说明
sub_filters	子节点可以是SingleColumnValueFilter或CompositeColumnValueFilter。 内节点的逻辑运算符不同可以挂载的子节点个数不同。 • 当内节点的逻辑运算符为NOT时，只能挂载一个子节点。 • 当内节点的逻辑运算符为AND或OR时，可以挂载多个子节点。

示例

- 构造SingleColumnValueFilter。

```
//设置过滤器，当Col0的值为0时，返回该行。
$column_filter = [
    'column_name' => 'Col0',
    'value' => 0,
    'comparator' => ComparatorTypeConst::CONST_EQUAL
    'pass_if_missing' => false                          //如果不存在Col0列，也不返回该行。
    'latest_version_only' => true                       //只判断最新版本。
];
```

- 构造CompositeColumnValueFilter。

```
//composite1的条件为(Col0 == 0) AND (Col1 > 100)。
$composite1 = [
    'logical_operator' => LogicalOperatorConst::CONST_AND,
    'sub_filters' => [
        [
            'column_name' => 'Col0',
            'value' => 0,
            'comparator' => ComparatorTypeConst::CONST_EQUAL
        ],
        [
            'column_name' => 'Col1',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ]
    ]
];
//composite2的条件为( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10)。
$composite2 = [
    'logical_operator' => LogicalOperatorConst::CONST_OR,
    'sub_filters' => [
        $composite1,
        [
            'column_name' => 'Col2',
            'value' => 10,
            'comparator' => ComparatorTypeConst::CONST_LESS_EQUAL
        ]
    ]
];
```

8.5. 单行数据操作

表格存储提供了PutRow、GetRow、UpdateRow和DeleteRow等单行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

插入一行数据（PutRow）

PutRow接口用于新写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。

- 接口

```
    /**
     * 写入一行数据。如果该行已存在，则先删除原行数据（原行的所有列以及所有版本的数据），再写入新行数据。返回该操作消耗的CU。
      * @api
     * @param [] $request 请求参数。
     * @return [] 请求返回。
     * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
     * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
     */
    public function putRow(array $request);
```

- 请求参数

参数	说明
table_name	数据表名称。

参数	说明
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。 ○ row_existence：行存在性条件。 说明 ■ RowExistenceExpectationConst::CONST_IGNORE表示无论此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。 ■ RowExistenceExpectationConst::CONST_EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。 ■ RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。 ○ column_condition：列条件。
primary_key	行的主键。 说明 ○ 设置的主键个数和类型必须和数据表的主键个数和类型一致。 ○ 当主键为自增列时，只需将自增列的值设置为占位符，详情请参见主键列自增。 ○ 数据表可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同，例如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照主键的大小为行排序，具体参见表格存储数据模型和查询操作。 ○ 每一项的顺序是主键名、主键值PrimaryKeyValue、主键类型PrimaryKeyType（可选）。 ○ PrimaryKeyValue可以是整数、二进制和字符串。 ○ PrimaryKeyType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、PK_AUTO_INCR（主键列自增）四种，分别用PrimaryKeyTypeConst::CONST_INTEGER，PrimaryKeyTypeConst::CONST_STRING，PrimaryKeyTypeConst::CONST_BINARY，PrimaryKeyTypeConst::CONST_PK_AUTO_INCR表示，对于INTEGER和STRING，可以省略，其它类型不可省略。

参数	说明
attribute_columns	行的属性列。 ○ 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType（可选）、时间戳（可选）。 ○ ColumnType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、BOOLEAN、DOUBLE五种，分别用ColumnTypeConst::CONST_INTEGER、ColumnTypeConst::CONST_STRING、ColumnTypeConst::CONST_BINARY、ColumnTypeConst::CONST_BOOLEAN、ColumnTypeConst::CONST_DOUBLE表示，其中BINARY不可省略，其他类型都可以省略，或者设置为null。 ○ 时间戳即数据的版本号，详情请参见数据版本和生命周期。 数据的版本号可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。
return_content	表示返回类型。 return_type：设置为ReturnTypeConst::CONST_PK，表示返回主键值，主要用于主键列自增场景。

- 请求格式

```
$result = $client->putRow([
    'table_name' => '<string>', //设置数据表名称。
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                                  //设置主键。
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => [          //设置属性列。
            ['<string>', <ColumnValue>],
            ['<string>', <ColumnValue>, <ColumnType>],
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
    ],
    'return_content' => [
        'return_type' => <ReturnType>
    ]
]);
```

- 响应参数

参数	说明
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写能力单元。 ○ read：读吞吐量 ○ write：写吞吐量
primary_key	主键的值，和请求时一致。 **说明** 当在请求中设置return_type为ReturnTypeConst::CONST_PK时，会返回完整的主键，主要用于主键列自增。
attribute_columns	属性列的值，和请求时一致，目前为空。

- 响应格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

- 示例1

  写入10列属性列，每列写入1个版本，由系统自动生成数据的版本号（时间戳）。

```
$attr = array();
for($i = 0; $i < 10; $i++) {
    $attr[] = ['Col'. $i, $i];
}
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE, //condition可以为IGNORE、EX
PECT_EXIST和EXPECT_NOT_EXIST。
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

- 示例2

  写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
$attr = array();
$timestamp = getMicroTime();
for($i = 0; $i < 10; $i++) {
    for($j = 0; $j < 3; $j++) {
        $attr[] = ['Col'. $i, $j, null, $timestamp+$j];
    }
}
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE, //condition可以为IGNORE、EX
PECT_EXIST和EXPECT_NOT_EXIST。
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

- 示例3

  期望原行不存在时，写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
$attr = array();
$timestamp = getMicroTime();
for($i = 0; $i < 10; $i++) {
    for($j = 0; $j < 3; $j++) {
        $attr[] = ['Col'. $i, $j, null, $timestamp+$j];
    }
}
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST, //设置期望原行不存在时，写入数据。
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

- 示例4

  期望原行存在且Col0列的值大于100时，写入10列属性列，每列写入3个版本，自定义数据的版本号（时间戳）。

```
$attr = array();
$timestamp = getMicroTime();
for($i = 0; $i < 10; $i++) {
    for($j = 0; $j < 3; $j++) {
        $attr[] = ['Col'. $i, $j, null, $timestamp+$j];
    }
}
$request = [
    'table_name' => 'MyTable',
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST,//期望原行存在。
        'column_condition' => [                    //使用条件更新，满足条件则更新。
            'column_name' => 'Col0',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ]
    ,
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

读取的结果可能有如下两种：

- 如果该行存在，则返回该行的各主键列以及属性列。
- 如果该行不存在，则返回中不包含行，并且不会报错。
- 接口

```
/**
 * 读取一行数据。
 * @api
 * @param [] $request 请求参数。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 */
public function getRow(array $request);
```

- 请求参数

参数	说明
table_name	数据表名称。
primary_key	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
max_versions	最多读取的版本数。 **说明** max_versions与time_range必须至少设置一个。 ○ 如果仅设置max_versions，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_versions和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。

参数	说明
time_range	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** max_versions与time_range必须至少设置一个。 ○ 如果仅设置max_versions，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_versions和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 如果查询一个范围的数据，则需要设置start_time和end_time。start_time和end_time分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[start_time, end_time)。 ○ 如果查询特定版本号的数据，则需要设置specific_time。specific_time表示特定的时间戳。 specific_time和[start_time, end_time)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为INT64.MAX。
columns_to_get	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ○ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columns_to_get参数限制。如果将col0和col1加入到columns_to_get中，则只返回col0和col1列的值。 ○ 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。
start_column	读取的起始列，主要用于宽行读，返回的结果中包含当前起始列。 列的顺序按照列名的字典序排序。例如一张表有“a”，”b”，”c”三列，读取时设置start_column为“b”，则会从”b”列开始读，返回”b”，”c”两列。
end_column	读取时的结束列，主要用于宽行读，返回的结果中不包含当前结束列。 列的顺序按照列名的字典序排序。例如一张表有”a”，”b”，”c”三列，读取时指定end_column为“b”，则读到”b”列时会结束，返回”a”列。
token	宽行读取时下一次读取的起始位置，暂不可用。

参数	说明
column_filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。

- 请求格式

```
$result = $client->getRow([
    'table_name' => '<string>',                    //设置数据表名称。
    'primary_key' => [                             //设置主键。
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'max_versions' => <integer>,
    'time_range' => [
        'start_time' => <integer>,
        'end_time' => <integer>,
        'specific_time' => <integer>
    ],
    'start_column' => '<string>',
    'end_column' => '<string>',
    'token' => '<string>',
    'columns_to_get' => [
        '<string>',
        '<string>',
        //...
    ],
    'column_filter' =>  <ColumnCondition>
]);
```

- 响应参数

参数	说明
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写能力单元。 ○ read：读吞吐量 ○ write：写吞吐量
primary_key	主键的值，和请求时一致。 **说明** 如果该行不存在，则primary_key为空列表[]。

参数	说明
attribute_columns	属性列的值。 **说明** 如果该行不存在，则attribute_columns为空列表[]。 ○ 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType、时间戳。 时间戳为64位整数，用于表示属性列数据的多个不同的版本。 ○ 返回结果中的属性会按照属性名的字典序升序，属性的多个版本按时间戳降序。 ○ 其顺序不保证与请求中的columns_to_get一致。
next_token	宽行读取时下一次读取的位置，暂不可用。

- 结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => [
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
    ],
    'next_token' => '<string>'
]
```

- 示例1

  读取一行，设置读取最新版本的数据和读取的列。

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'max_versions' => 1,                    //设置读取最新版本。
    'columns_to_get' => ['Col0']            //设置读取的列。
];
$response = $otsClient->getRow ($request);
```

- 示例2

  在读取一行数据时使用过滤器。

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'max_versions' => 1,                    //设置读取最新版本。
    'column_filter' => [                    //设置过滤器，当Col0列的值为0时，返回该行。
        'column_name' => 'Col0',
        'value' => 0,
        'comparator' => ComparatorTypeConst::CONST_EQUAL,
        'pass_if_missing' => false          //如果Col0列不存在，则不返回该行。
    ]
];
$response = $otsClient->getRow ($request);
```

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，可以增加和删除一行中的属性列，删除属性列指定版本的数据，或者更新已存在的属性列的值。如果更新的行不存在，则新增一行数据。

> **说明** 当UpdateRow请求中只包含删除指定的列且该行不存在时，则该请求不会新增一行数据。

- 接口

```
    /**
     * 更新一行数据。
     * @api
     * @param [] $request 请求参数。
     * @return [] 请求返回。
     * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
     * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
     */
    public function updateRow(array $request);
```

- 请求参数

参数	说明
table_name	数据表名称。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。
primary_key	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
update_of_attribute_columns	更新的属性列。 ○ 增加或更新数据时，需要设置属性名、属性值、属性类型（可选）、时间戳（可选）。 时间戳即数据的版本号，可以由系统自动生成或者自定义，如果不设置此参数，则默认由系统自动生成。详情请参见数据版本和生命周期。 ■ 当由系统自动生成数据的版本号时，系统默认将当前时间的毫秒单位时间戳（从1970-01-01 00:00:00 UTC计算起的毫秒数）作为数据的版本号。 ■ 当自定义数据的版本号时，版本号需要为64位的毫秒单位时间戳且在有效版本范围内。 ○ 删除属性列特定版本的数据时，只需要设置属性名和时间戳。 时间戳是64位整数，单位为毫秒，表示某个特定版本的数据。 ○ 删除属性列时，只需要设置属性名。 **说明** 删除一行的全部属性列不等同于删除该行，如果需要删除该行，请使用DeleteRow操作。
return_content	表示返回类型。 return_type：目前只需要设置为ReturnTypeConst::CONST_PK，表示返回主键值，主要用于主键列自增场景。

- 请求格式

```
$result = $client->updateRow([
    'table_name' => '<string>', //设置数据表名称。
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                                  //设置主键。
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'update_of_attribute_columns' => [          //设置更新的属性列。
        'PUT' => [
            ['<string>', <ColumnValue>],
            ['<string>', <ColumnValue>, <ColumnType>],
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
        ],
        'DELETE' => [
            ['<string>', <integer>],
            ['<string>', <integer>],
            ['<string>', <integer>],
            ['<string>', <integer>]
        ],
        'DELETE_ALL' => [
            '<string>',
            '<string>',
            '<string>',
            '<string>'
        ],
    ],
    'return_content' => [
        'return_type' => <ReturnType>
    ]
]);
```

- 响应参数

参数	说明
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写能力单元。 ○ read：读吞吐量 ○ write：写吞吐量
primary_key	主键的值，和请求时一致。 **说明** 当在请求中设置return_type为ReturnTypeConst::CONST_PK时，会返回完整的主键，主要用于主键列自增。

参数	说明
attribute_columns	属性列的值，和请求时一致，目前为空。

- 结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

- 示例1

  更新一些列，删除某列的某一版本，删除某列。

```
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'update_of_attribute_columns' => [
        'PUT' => [                          //更新一些列。
            ['Col0', 100],
            ['Col1', 'Hello'],
            ['Col2', 'a binary', ColumnTypeConst::CONST_BINARY],
            ['Col3', 100, null, 1526418378526]
        ],
        'DELETE' => [                       //删除某列的某一版本。
            ['Col10', 1526418378526]
        ],
        'DELETE_ALL' => [
            'Col11'                         //删除某一列。
        ]
    ]
];
$response = $otsClient->updateRow($request);
```

- 示例2

  设置更新的条件。

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST, //期望原行存在。
        'column_filter' => [                                                 //当Col0列的值大于100时更新数据。
            'column_name' => 'Col0',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ]
    ],
    'update_of_attribute_columns' => [
        'PUT' => [                      //更新一些列。
            ['Col0', 100],
            ['Col1', 'Hello'],
            ['Col2', 'a binary', ColumnTypeConst::CONST_BINARY],
            ['Col3', 100, null, 1526418378526]
        ],
        'DELETE' => [                   //删除某列的某一版本。
            ['Col10', 1526418378526]
        ],
        'DELETE_ALL' => [
            'Col11'                     //删除某一列。
        ]
    ]
];
```

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行数据。如果删除的行不存在，则不会发生任何变化。

- 接口

```
    /**
     * 删除一行数据。
     * @api
     * @param [] $request 请求参数。
     * @return [] 请求返回。
     * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
     * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
     */
    public function deleteRow(array $request);
```

- 请求参数

参数	说明
table_name	数据表名称。
condition	使用条件更新，可以设置原行的存在性条件或者原行中某列的列值条件，详情请参见条件更新。
primary_key	行的主键。 **说明** 设置的主键个数和类型必须和数据表的主键个数和类型一致。
return_content	表示返回类型。 return_type：目前只需要设置为ReturnTypeConst::CONST_PK，表示返回主键值，主要用于主键列自增场景。

- 请求格式

```
$result = $client->deleteRow([
    'table_name' => '<string>', //设置数据表名称。
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                                  //设置主键。
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'return_content' => [
        'return_type' => <ReturnType>
    ]
]);
```

- 响应参数

参数	说明
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写能力单元。 ○ read：读吞吐量 ○ write：写吞吐量

参数	说明
primary_key	主键的值，和请求时一致。 **说明** 当在请求中设置return_type为ReturnTypeConst::CONST_PK时，会返回完整的主键，主要用于主键列自增。
attribute_columns	属性列的值，和请求时一致，目前为空。

- 结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

- 示例1

  删除一行数据。

```
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'return_content' => [
        'return_type' => ReturnTypeConst::CONST_PK     //使用主键列自增时，可以设置return_type返回主键。
    ]
];
$response = $otsClient->deleteRow($request);
```

- 示例2

  设置删除条件。

```
$request = [
    'table_name' => 'MyTable',
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPECT_EXIST, //期望原行存在。
        'column_filter' => [          //当Col0列的值大于100时删除数据。
            'column_name' => 'Col0',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ],
    ],
    'primary_key' => [ //设置主键。
        ['PK0', 123],
        ['PK1', 'abc']
    ]
];
$response = $otsClient->deleteRow ($request);
```

8.6. 多行数据操作

表格存储提供了BatchWriteRow、BatchGetRow、GetRange等多行操作的接口。

如果需要了解表格存储各场景的应用案例，请参见快速玩转Tablestore入门与实战。

前提条件

- 已初始化Client，详情请参见初始化。
- 已创建数据表并写入数据。

批量写（BatchWriteRow）

批量写接口用于在一次请求中进行批量的写入操作，也支持一次对多个数据表进行写入。BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，构造子操作的过程与使用PutRow接口、UpdateRow接口和DeleteRow接口时相同，也支持使用条件更新。

BatchWriteRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

- 由于批量写入可能存在部分行失败的情况，失败行的Index及错误信息在返回的BatchWriteRowResponse中，但并不抛出异常。因此调用BatchWriteRow接口时，需要检查返回值，判断每行的状态是否成功；如果不检查返回值，则可能会忽略掉部分操作的失败。
- 当服务端检查到某些操作出现参数错误时，BatchWriteRow接口可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。
- 接口

```
/**
 * 写入、更新或者删除指定的多行数据。
 * 请注意BatchWriteRow在部分行读取失败时，会在返回的$response中表示，而不是抛出异常。详情请参见处理BatchWriteRow的返回样例。
 * @api
 * @param [] $request 请求参数。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时。
 * @throws OTSServerException 当OTS服务端返回错误时。
 */
public function batchWriteRow(array $request);
```

- 请求参数

  本操作是PutRow、UpdateRow、DeleteRow的组合。

  - 增加了数据表的层级结构，可以一次处理多个数据表。

    tables以数据表为单位组织，后续为各个数据表的操作，设置需要写入、修改或删除的行信息，参数说明请参见单行数据操作。

  - 增加了operation_type参数，用于区分操作类型。

    操作类型可以为PUT、UPDATE、DELETE，分别用OperationTypeConst::CONST_PUT、OperationTypeConst::CONST_UPDATE、OperationTypeConst::CONST_DELETE表示。

    - 当操作类型为PUT时，primary_key和attribute_columns有效。
    - 当操作类型为UPDATE时，primary_key和update_of_attribute_columns有效。
    - 当操作类型为DELETE时，primary_key有效。

- 请求格式

```
$result = $client->batchWriteRow([
    'tables' => [                                                   //设置数据表的层级结构。
        [
            'table_name' => '<string>',                             //设置数据表名称。
            'operation_type' => <OperationType>,
            'condition' => [
                'row_existence' => <RowExistence>,
                'column_condition' => <ColumnCondition>
            ],
            'primary_key' => [                                      //设置主键。
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'attribute_columns' => [                                //当操作类型为PUT时必须设置。
                    ['<string>', <ColumnValue>],
                    ['<string>', <ColumnValue>, <ColumnType>],
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ],
            'update_of_attribute_columns' => [                      //当操作类型为UPDATE时必须设置
。
                'PUT' => [
                    ['<string>', <ColumnValue>],
                    ['<string>', <ColumnValue>, <ColumnType>],
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                ],
                'DELETE' => [
                    ['<string>', <integer>],
                    ['<string>', <integer>],
                    ['<string>', <integer>],
                    ['<string>', <integer>]
                ],
                'DELETE_ALL' => [
                    '<string>',
                    '<string>',
                    '<string>',
                    '<string>'
                ],
            ],
            'return_content' => [
                'return_type' => <ReturnType>
            ]
        ],
        //其他数据表。
    ]
]);
```

- 响应参数

  tables以table为单位组织，和请求一一对应，参数说明请参见下表。

参数	说明
table_name	数据表名称。

参数	说明
is_ok	该行操作是否成功。 ○ 如果值为true，则该行写入成功，此时error无效。 ○ 如果值为false，则该行写入失败。
error	用于在操作失败时的响应消息中表示错误信息。 ○ code表示当前单行操作的错误码。 ○ message表示当前单行操作的错误信息。
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写单元。 ○ read：读吞吐量 ○ write：写吞吐量
primary_key	主键的值，和请求时一致。 设置return_type时会有值，主要用于主键列自增。
attribute_columns	属性列的值，和请求时一致，目前为空。

- 结果格式

```
[
    'tables' => [
        [
            'table_name' => '<string>',
            'rows' => [
                [
                    'is_ok' => true || false,
                    'error' => [
                        'code' => '<string>',
                        'message' => '<string>',
                    ]
                    'consumed' => [
                        'capacity_unit' => [
                            'read' => <integer>,
                            'write' => <integer>
                        ]
                    ],
                    'primary_key' => [
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
                    ],
                    'attribute_columns' => []
                ],
                //其他行。
            ]
        ],
        //其他数据表。
    ]
]
```

- 示例

  批量写入30行数据，分别向3个数据表中写入数据，每个数据表中写入10行。

```
//向3个数据表中写入数据，每个数据表中写入10行。
$tables = array();
for($i = 0; $i < 3; $i++) {
    $rows = array();
    for($j = 0; $j < 10; $j++) {
        $rows[] = [
            'operation_type' => OperationTypeConst::CONST_PUT, //设置操作类型为PUT。
            'condition' => RowExistenceExpectationConst::CONST_IGNORE,
            'primary_key' => [
                ['pk0', $i],
                ['pk1', $j]
            ],
            'attribute_columns' => [
                ['Col0', 4],
                ['Col2', '成杭京']
            ]
        ];
    }
    $tables[] = [
        'table_name' => 'SampleTable' . $i,
        'rows' => $rows
    ];
}
$request = [
    'tables' => $tables
];
$response = $otsClient->batchWriteRow ($request);
//处理返回的每个数据表。
foreach ($response['tables'] as $tableData) {
  print "Handling table {$tableData['table_name']} ...\n";
  //处理该数据表下的PutRow返回的结果。
  $putRows = $tableData['rows'];
  foreach ($putRows as $rowData) {
    if ($rowData['is_ok']) {
      //写入成功。
      print "Capacity Unit Consumed: {$rowData['consumed']['capacity_unit']['write']}\n";
    } else {
      //处理出错。
      print "Error: {$rowData['error']['code']} {$rowData['error']['message']}\n";
    }
  }
}
```

详细代码示例请参见下表。

示例	说明
BatchWriteRow1.php	展示了BatchWriteRow中多个PUT的用法。
BatchWriteRow2.php	展示了BatchWriteRow中多个UPDATE的用法。
BatchWriteRow3.php	展示了BatchWriteRow中多个DELETE的用法。

示例	说明
BatchWriteRow4.php	展示了BatchWriteRow中组合使用UPDATE、PUT和DELETE的用法。
BatchWriteRowWithColumnFilter.php	展示了BatchWriteRow中同时使用条件更新的用法。

批量读（BatchGetRow）

批量读接口用于一次请求读取多行数据，也支持一次对多张表进行读取。BatchGetRow由多个GetRow子操作组成。构造子操作的过程与使用GetRow接口时相同，也支持使用过滤器。

批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA列。

BatchGetRow的各个子操作独立执行，表格存储会分别返回各个子操作的执行结果。

由于批量读取可能存在部分行失败的情况，失败行的错误信息在返回的BatchGetRowResponse中，但并不抛出异常。因此调用BatchGetRow接口时，需要检查返回值，判断每行的状态是否成功。

- 接口

```
/**
 * 读取指定的多行数据。
 * 请注意BatchGetRow在部分行读取失败时，会在返回的$response中表示，而不是抛出异常。详情请参见处理BatchGetRow的返回样例。
 * @api
 * @param [] $request 请求参数。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时。
 * @throws OTSServerException 当OTS服务端返回错误时。
 */
public function batchGetRow(array $request);
```

- 请求参数

  BatchGetRow和GetRow的区别如下：

  - 增加了数据表的层级结构，可以一次读取多个数据表的数据。

    tables以数据表为单位组织，后续为各个数据表的操作，设置了需要读取的行信息，参数说明请参见单行数据操作。

  - primary_key变为primary_keys，支持设置多行的主键，可以一次读取多行数据。

- 请求格式

```
$result = $client->batchGetRow([
    'tables' => [                                                    //设置数据表的层级结构。
        [
            'table_name' => '<string>',                              //设置数据表名称。
            'primary_keys' => [                                      //设置主键。
                [
                    ['<string>', <PrimaryKeyValue>],
                    ['<string>', <PrimaryKeyValue>],
                    ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
                ],
                //其他主键。
            ]
            'max_versions' => <integer>,
            'time_range' => [
                'start_time' => <integer>,
                'end_time' => <integer>,
                'specific_time' => <integer>
            ],
            'start_column' => '<string>',
            'end_column' => '<string>',
            'token' => '<string>',
            'columns_to_get' => [
                '<string>',
                '<string>',
                //...
            ],
            'column_filter' =>  <ColumnCondition>
        ],
        //其他数据表。
    ]
]);
```

- 响应参数

  tables以table为单位组织，和请求一一对应， 参数说明请参见下表。

参数	说明
table_name	数据表名称。
is_ok	该行操作是否成功。 ○ 如果值为true，则该行读取成功，此时error无效。 ○ 如果值为false，则该行读取失败，此时consumed、primary_key、attribute_columns无效。
error	用于在操作失败时的响应消息中表示错误信息。 ○ code表示当前单行操作的错误码。 ○ message表示当前单行操作的错误信息。

参数	说明
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写能力单元。 ○ read：读吞吐量 ○ write：写吞吐量
primary_key	主键的值，和请求时一致。 **说明** 如果该行不存在，则primary_key为空列表[]。
attribute_columns	属性列的值。 **说明** 如果该行不存在，则attribute_columns为空列表[]。 ○ 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType、时间戳。 时间戳为64位整数，用于表示属性列数据的多个不同的版本。 ○ 返回结果中的属性会按照属性名的字典序升序，属性的多个版本按时间戳降序。 ○ 其顺序不保证与请求中的columns_to_get一致。
next_token	宽行读取时下一次读取的位置，暂不可用。

- 结果格式

```
[
    'tables' => [
        [
            'table_name' => '<string>',
            'rows' => [
                [
                    'is_ok' => true || false,
                    'error' => [
                        'code' => '<string>',
                        'message' => '<string>',
                    ]
                    'consumed' => [
                        'capacity_unit' => [
                            'read' => <integer>,
                            'write' => <integer>
                        ]
                    ],
                    'primary_key' => [
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>],
                        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
                    ],
                    'attribute_columns' => [
                            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                    ],
                    'next_token' => '<string>'
                ],
                //其他行。
            ]
        ],
        //其他数据表。
    ]
]
```

- 示例

  批量一次读取30行，分别从3个数据表中读取数据，每个数据表读取10行。

```
//从3个数据表中读取数据，每个数据表读取10行。
$tables = array();
for($i = 0; $i < 3; $i++) {
    $primary_keys = array();
    for($j = 0; $j < 10; $j++) {
        $primary_keys[] = [
            ['pk0', $i],
            ['pk1', $j]
        ];
    }
    $tables[] = [
        'table_name' => 'SampleTable' . $i,
        'max_versions' => 1,
        'primary_keys' => $primary_keys
    ];
}
$request = [
    'tables' => $tables
];
$response = $otsClient->batchGetRow ($request);
//处理返回的每个数据表。
foreach ($response['tables'] as $tableData) {
  print "Handling table {$tableData['table_name']} ...\n";
  //处理该数据表下的每行数据。
  foreach ($tableData['rows'] as $rowData) {
    if ($rowData['is_ok']) {
      //处理读取到的数据。
        $row = json_encode($rowData['primary_key']);
        print "Handling row: {$row}\n";
    } else {
      //处理出错。
      print "Error: {$rowData['error']['code']} {$rowData['error']['message']}\n";
    }
  }
}
```

详细代码示例请参见下表。

示例	说明
BatchGetRow1.php	展示了BatchGetRow获取单个数据表多行的用法。
BatchGetRow2.php	展示了BatchGetRow获取多个数据表多行的用法。
BatchGetRow3.php	展示了BatchGetRow获取单个数据表多行同时指定获取特定列的用法。
BatchGetRow4.php	展示了BatchGetRow处理返回结果的用法。
BatchGetRowWithColumnFilter.php	展示了BatchGetRow同时使用过滤器的用法。

范围读（GetRange）

范围读接口用于读取一个主键范围内的数据。

范围读接口支持按照确定范围进行正序读取和逆序读取，可以设置要读取的行数。如果范围较大，已扫描的行数或者数据量超过一定限制，会停止扫描，并返回已获取的行和下一个主键信息。您可以根据返回的下一个主键信息，继续发起请求，获取范围内剩余的行。

GetRange操作可能在如下情况停止执行并返回数据。

- 扫描的行数据大小之和达到4 MB。
- 扫描的行数等于5000。
- 返回的行数等于最大返回行数。
- 当前剩余的预留读吞吐量已全部使用，余量不足以读取下一条数据。

> **说明** 表格存储表中的行默认是按照主键排序的，而主键是由全部主键列按照顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

- 接口

```
/**
 * 范围读起始主键和结束主键间的数据。
 * 请注意服务端可能会截断此范围，需要判断返回中的next_start_primary_key来判断是否继续调用GetRange。
 * 可以指定最多读取多少行。
 * 在指定开始主键和结束主键时，可以使用INF_MIN和INF_MAX分别表示最小值和最大值，详情请参见如下代码样例。
 * @api
 * @param [] $request 请求参数。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时。
 * @throws OTSServerException 当OTS服务端返回错误时。
 */
public function getRange(array $request);
```

- 参数

  GetRange和GetRow的区别如下：

  - primary_key变成inclusive_start_primary_key和exclusive_end_primary_key，前闭后开区间。
  - 增加direction表示读取方向。
  - 增加limit限制返回行数。

参数	说明
table_name	数据表名称。

参数	说明
inclusive_start_primary_key	本次范围读的起始主键和结束主键，起始主键和结束主键需要是有效的主键或者是由INF_MIN和INF_MAX类型组成的虚拟点，虚拟点的列数必须与主键相同。 其中INF_MIN表示无限小，任何类型的值都比它大；INF_MAX表示无限大，任何类型的值都比它小。 ○ inclusive_start_primary_key表示起始主键，如果该行存在，则返回结果中一定会包含此行。 ○ exclusive_end_primary_key表示结束主键，无论该行是否存在，返回结果中都不会包含此行。 数据表中的行按主键从小到大排序，读取范围是一个左闭右开的区间，正序读取时，返回的是大于等于起始主键且小于结束主键的所有的行。
exclusive_end_primary_key	
direction	读取方向。 ○ 如果值为正序（DirectionConst::CONST_FORWARD），则起始主键必须小于结束主键，返回的行按照主键由小到大的顺序进行排列。 ○ 如果值为逆序（DirectionConst::CONST_BACKWARD），则起始主键必须大于结束主键，返回的行按照主键由大到小的顺序进行排列。 例如同一表中有两个主键A和B，A<B。如正序读取[A, B)，则按从A至B的顺序返回主键大于等于A、小于B的行；逆序读取[B, A)，则按从B至A的顺序返回大于A、小于等于B的数据。
limit	数据的最大返回行数，此值必须大于 0。 表格存储按照正序或者逆序返回指定的最大返回行数后即结束该操作的执行，即使该区间内仍有未返回的数据。此时可以通过返回结果中的next_start_primary_key记录本次读取到的位置，用于下一次读取。
max_versions	最多读取的版本数。 **说明** max_version与time_range必须至少设置一个。 ○ 如果仅设置max_version，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_version和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。

参数	说明
time_range	读取版本号范围或特定版本号的数据。详情请参见TimeRange。 **说明** max_version与time_range必须至少设置一个。 ○ 如果仅设置max_version，则最多返回所有版本中从新到旧指定数量版本的数据。 ○ 如果仅设置time_range，则返回该范围内所有数据或指定版本数据。 ○ 如果同时设置max_version和time_range，则最多返回版本号范围内从新到旧指定数量版本的数据。 ○ 如果查询一个范围的数据，则需要设置start_time和end_time。start_time和end_time分别表示起始时间戳和结束时间戳，范围为前闭后开区间，即[start_time, end_time)。 ○ 如果查询特定版本号的数据，则需要设置specific_time。specific_time表示特定的时间戳。 specific_time和[start_time, end_time)中只需要设置一个。 时间戳的单位为毫秒，最小值为0，最大值为INT64.MAX。
columns_to_get	读取的列集合，列名可以是主键列或属性列。 如果不设置返回的列名，则返回整行数据。 **说明** ○ 查询一行数据时，默认返回此行所有列的数据。如果需要只返回特定列，可以通过设置columns_to_get参数限制。如果将col0和col1加入到columns_to_get中，则只返回col0和col1列的值。 ○ 如果某行数据的主键属于读取范围，但是该行数据不包含指定返回的列，那么返回结果中不包含该行数据。 ○ 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。
start_column	读取的起始列，主要用于宽行读，返回的结果中包含当前起始列。 列的顺序按照列名的字典序排序。例如一张表有“a”，”b”，”c”三列，读取时设置start_column为“b”，则会从”b”列开始读，返回”b”，”c”两列。

参数	说明
end_column	读取时的结束列，主要用于宽行读，返回的结果中不包含当前结束列。 列的顺序按照列名的字典序排序。例如一张表有”a”，”b”，”c”三列，读取时指定end_column为“b”，则读到”b”列时会结束，返回”a”列。
token	宽行读取时下一次读取的位置，暂不可用。
column_filter	使用过滤器，在服务端对读取结果再进行一次过滤，只返回符合过滤器中条件的数据行，详情请参见过滤器。 **说明** 当columns_to_get和column_filter同时使用时，执行顺序是先获取columns_to_get指定的列，再在返回的列中进行条件过滤。

- 请求格式

```
$result = $client->getRange([
    'table_name' => '<string>',                                    //设置表名称。
    'inclusive_start_primary_key' => [                             //设置起始主键。
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'exclusive_end_primary_key' => [                               //设置结束主键。
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'direction' => <Direction>,                                    //设置读取方向。
    'limit' => <Direction>,
    'max_versions' => <integer>,
    'time_range' => [
        'start_time' => <integer>,
        'end_time' => <integer>,
        'specific_time' => <integer>
    ],
    'start_column' => '<string>',
    'end_column' => '<string>',
    'token' => '<string>',
    'columns_to_get' => [
        '<string>',
        '<string>',
        //...
    ],
    'column_filter' =>  <ColumnCondition>
]);
```

- 响应参数

参数	说明
consumed	本次操作消耗服务能力单元的值。 capacity_unit表示使用的读写能力单元。 ○ read：读吞吐量 ○ write：写吞吐量
primary_key	主键的值，和请求时一致。
attribute_columns	属性列的值。 ○ 每一项的顺序是属性名、属性值ColumnValue、属性类型ColumnType、时间戳。 时间戳为64位整数，单位为毫秒，用于表示属性列数据的多个不同的版本。 ○ 返回结果中的属性会按照属性名的字典序升序，属性的多个版本按时间戳降序。 ○ 其顺序不保证与请求中的columns_to_get一致。
next_start_primary_key	根据返回结果中的next_start_primary_key判断数据是否全部读取。 ○ 当返回结果中next_start_primary_key不为空时，可以使用此返回值作为下一次GetRange操作的起始点继续读取数据。 ○ 当返回结果中next_start_primary_key为空时，表示读取范围内的数据全部返回。

- 结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'next_start_primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'rows' => [
        [
            'primary_key' => [
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'attribute_columns' => [
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
                    ['<string>', <ColumnValue>, <ColumnType>, <integer>]
            ]
        ],
        //其他行。
    ]
]
```

- 示例

  按照范围读取数据。

```
//查找PK0从1到4（左闭右开）的数据。
//范围的边界需要提供完整的主键，如果查询的范围不涉及到某一列值的范围，则需要将该列设置为无穷大（INF_MAX）或者无穷小（INF_MIN）。
$startPK = [
    ['PK0', 1],
    ['PK1', null, PrimaryKeyTypeConst::CONST_INF_MIN]  //'INF_MIN'表示最小值。
];
//范围的边界需要提供完整的主键，如果查询的范围不涉及到某一列值的范围，则需要将该列设置为无穷大（INF_MAX）或者无穷小（INF_MIN）。
$endPK = [
    ['PK0', 4],
    ['PK1', null, PrimaryKeyTypeConst::CONST_INF_MAX]  //'INF_MAX'表示最大值。
];
$request = [
    'table_name' => 'SampleTable',
    'max_versions' => 1,                            //设置读取最新版本。
    'direction' => DirectionConst::CONST_FORWARD, //设置正序查询。
    'inclusive_start_primary_key' => $startPK,     //设置开始主键。
    'exclusive_end_primary_key' => $endPK,         //设置结束主键。
    'limit' => 10                                   //设置最多返回10行数据。
];
$response = $otsClient->getRange ($request);
print "Read CU Consumed: {$response['consumed']['capacity_unit']['read']}\n";
foreach ($response['rows'] as $rowData) {
  //处理每一行数据。
}
```

详细代码示例请参见下表。

示例	说明
GetRange1.php	展示了GetRange的用法。
GetRange2.php	展示了GetRange指定获取列的用法。
GetRange3.php	展示了GetRange指定获取行数限制的用法。
GetRangeWithColumnFilter.php	展示了GetRange同时使用过滤器的用法。

8.7. 多元索引

索引预排序(IndexSort)

多元索引默认会按照索引中配置的IndexSort进行排序（含有NESTED类型的索引不支持IndexSort，没有预排序），默认的IndexSort为主键排序，用户可以在创建索引时自定义预排序方式。IndexSort决定了多元索引查询时默认的返回顺序，若用户未自定义IndexSort，即按照主键顺序返回。

> **说明** IndexSort不支持ScoreSort、GeoDistanceSort。

示例

```
"index_sort" => array(
    array(
        'field_sort' => array(
            'field_name' => 'keyword',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
    array(
        'pk_sort' => array(
            'order' => SortOrderConst::SORT_ORDER_ASC
        )
    ),
)
```

查询时指定排序方式

在每次查询时，用户也可以指定排序方式，多元索引支持以下四种排序方式(Sorter)。用户也可以使用多个Sorter，实现先按照某种方式排序，再按照某种方式排序的需求。

ScoreSort

按照分数进行排序，应用在全文索引等有相关性的场景下。需要注意的时，必须手动设置ScoreSort，才能按照相关性打分进行排序，否则会按照索引设置的IndexSort进行排序返回。sort字段设置：

```
'sort' => array(
    array(
        'score_sort' => array(
            'order' => SortOrderConst::SORT_ORDER_DESC
        )
    ),
)
```

PrimaryKeySort

按照主键排序。sort字段设置：

```
'sort' => array(
    array(
        'pk_sort' => array(
            'order' => SortOrderConst::SORT_ORDER_ASC
        )
    ),
)
```

FieldSort

按照某列进行排序。sort字段设置：

```
'sort' => array(
    array(
        'field_sort' => array(
            'field_name' => 'keyword',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
)
```

GeoDistanceSort

根据地理点距离进行排序。sort字段设置：

```
'sort' => array(
    array(
        'geo_distance_sort' => array(
            'field_name' => 'geo',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
            'points' => array('0.6,0.6')
        )
    ),
)
```

多类型组合排序

先按照某列排序，再按照另一列排序。sort字段设置：

```
'sort' => array(
    array(
        'field_sort' => array(
            'field_name' => 'keyword',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
    array(
        'pk_sort' => array(
            'order' => SortOrderConst::SORT_ORDER_ASC
        )
    ),
)
```

翻页方式

只有enableSortAndAgg设置为true的字段才能进行排序，排序方式支持多条件设置

使用limit和offset

当需要获取的总条数小于2000行时，可以通过limit和offset进行翻页，limit+offset <= 2000。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 10,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    )
);
$response = $otsClient->search($request);
```

使用token进行翻页

服务端在一次查询之后会返回NextToken，用户可以使用NextToken继续读取后面的数据，排序方式会跟上一次请求一致(不管是系统默认采用IndexSort排序还是用户自定义排序)。使用Token后不能设置Offset，只能一次一次往后读，即无法跳页。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::FUNCTION_SCORE_QUERY,
            'query' => array(
                'query' => array(
                    'query_type' => QueryTypeConst::TERM_QUERY,
                    'query' => array(
                        'field_name' => 'keyword',
                        'term' => 'keyword'
                    )
                ),
                'field_value_factor' => array(
                    'field_name' => 'long'
                )
            )
        ),
        'sort' => array(
            array(
                'score_sort' => array(
                    'order' => SortOrderConst::SORT_ORDER_DESC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
print "total_hits: " . $response['total_hits'] . "\n";
print json_encode($response['rows'], JSON_PRETTY_PRINT);
while($response['next_token'] != null) {
    $request['search_query']['token'] = $response['next_token'];
    $request['search_query']['sort'] = null;//有next_token时，不设置sort，token中含sort信息
    $response = $otsClient->search($request);
    print json_encode($response['rows'], JSON_PRETTY_PRINT);
}
```

8.7.1. 创建多元索引

使用CreateSearchIndex接口在数据表上创建一个多元索引。一个数据表可以创建多个多元索引。

前提条件

- 已初始化Client。具体操作，请参见初始化。

- 已创建数据表，且数据表的数据生命周期（time_to_live）必须为-1，最大版本数（max_versions）必须为1。

接口

```
/**
 * 创建多元索引。
 * @api
 *
 * @param [] $request
 *            请求参数、数据表名称、索引配置等。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 * @example "src/examples/CreateSearchIndex.php"
 */
public function createSearchIndex(array $request)
```

参数

创建多元索引时，需要指定数据表名称（table_name）、多元索引名称（index_name）和索引的结构信息（schema），其中schema包含field_schemas（Index的所有字段的设置）、index_setting（索引设置）和index_sort（索引预排序设置）。详细参数说明请参见下表。

参数	说明
table_name	数据表名称。
index_name	多元索引名称。

参数	说明
field_schemas	field_schema的列表，每个field_schemas包含如下内容： • field_name（必选）：建立索引的字段名，即列名，类型为String。 多元索引中的字段可以是主键列或者属性列。 • field_type（必选）：字段类型，类型为FieldTypeConst::XXX。更多信息，请参见字段。 • is_array（可选）：是否为数组，类型为Boolean。 如果设置为true，则表示该列是一个数组，在写入时，必须按照JSON数组格式写入，例如["a","b","c"]。 由于Nested类型是一个数组，当field_type为Nested类型时，无需设置此参数。 • index（可选）：是否开启索引，类型为Boolean。 默认为true，表示对该列构建倒排索引或者空间索引；如果设置为false，则不会对该列构建索引。 • analyzer（可选）：分词器类型。当字段类型为Text时，可以设置此参数；如果不设置，则默认分词器类型为单字分词。关于分词的更多信息，请参见分词。 • enable_sort_and_agg（可选）：是否开启排序与统计聚合功能，类型为Boolean。 只有enable_sort_and_agg设置为true的字段才能进行排序。关于排序的更多信息，请参见排序和翻页。 注意 Nested类型的字段不支持开启排序与统计聚合功能，但是Nested类型内部的子列支持开启排序与统计聚合功能。 • store（可选）：是否在多元索引中附加存储该字段的值，类型为Boolean。
index_setting	索引设置，包含routing_fields设置。 routing_fields（可选）：自定义路由字段。可以选择部分主键列作为路由字段，在进行索引数据写入时，会根据路由字段的值计算索引数据的分布位置，路由字段的值相同的记录会被索引到相同的数据分区中。

参数	说明
index_sort	索引预排序设置，包含sorters设置。如果不设置，则默认按照主键排序。 **说明** 含有Nested类型的索引不支持index_sort，没有预排序。 sorters（必选）：索引的预排序方式，支持按照主键排序和字段值排序。关于排序的更多信息，请参见排序和翻页。 • PrimaryKeySort表示按照主键排序，包含如下设置： order：排序的顺序，可按升序或者降序排序，默认为升序（SortOrderConst::SORT_ORDER_ASC）。 • FieldSort表示按照字段值排序，包含如下设置： 只有建立索引且开启排序与统计聚合功能的字段才能进行预排序。 ○ field_name：排序的字段名。 ○ order：排序的顺序，可按照升序或者降序排序，默认为升序（SortOrderConst::SORT_ORDER_ASC）。 ○ mode：当字段存在多个值时的排序方式。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'schema' => array(
        'field_schemas' => array(
            array(
                'field_name' => 'keyword',
                'field_type' => FieldTypeConst::KEYWORD,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'text',
                'field_type' => FieldTypeConst::TEXT,
                'analyzer' => 'single_word',
                'index' => true,
                'enable_sort_and_agg' => false,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'geo',
                'field_type' => FieldTypeConst::GEO_POINT,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
```

```
            ),
            array(
                'field_name' => 'long',
                'field_type' => FieldTypeConst::LONG,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'double',
                'field_type' => FieldTypeConst::DOUBLE,
                'index' => true,
                'enable_sort_and_agg' => true,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'boolean',
                'field_type' => FieldTypeConst::BOOLEAN,
                'index' => true,
                'enable_sort_and_agg' => false,
                'store' => true,
                'is_array' => false
            ),
            array(
                'field_name' => 'array',
                'field_type' => FieldTypeConst::KEYWORD,
                'index' => true,
                'enable_sort_and_agg' => false,
                'store' => true,
                'is_array' => true
            ),
            array(
                'field_name' => 'nested',
                'field_type' => FieldTypeConst::NESTED,
                'index' => false,
                'enable_sort_and_agg' => false,
                'store' => false,
                'field_schemas' => array(
                    array(
                        'field_name' => 'nested_keyword',
                        'field_type' => FieldTypeConst::KEYWORD,
                        'index' => false,
                        'enable_sort_and_agg' => false,
                        'store' => false,
                        'is_array' => false
                    )
                )
            ),
        ),
        'index_setting' => array(
            'routing_fields' => array("pk1")
        ),
//        "index_sort" => array(//含有Nested类型的索引不支持index_sort，没有预排序。
```

```
//            array(
//                'field_sort' => array(
//                    'field_name' => 'keyword',
//                    'order' => SortOrderConst::SORT_ORDER_ASC,
//                    'mode' => SortModeConst::SORT_MODE_AVG,
//                )
//            ),
//            array(
//                'pk_sort' => array(
//                    'order' => SortOrderConst::SORT_ORDER_ASC
//                )
//            ),
//        )
    )
);
$response = $otsClient->createSearchIndex();
```

8.7.2. 删除多元索引

使用DeleteSearchIndex接口可以删除指定数据表的一个多元索引。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

接口

```
/**
 * 删除多元索引。
 * @api
 *
 * @param [] $request
 *            请求参数，数据表名称、多元索引名称。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 * @example "src/examples/DeleteSearchIndex.php"
 */
public function deleteSearchIndex(array $request)
```

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index'
);
$response = $otsClient->deleteSearchIndex($request);
```

8.7.3. 查询多元索引描述信息

创建多元索引后，使用DescribeSearchIndex接口可以查询多元索引的描述信息，包括多元索引的字段信息和索引配置等。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

接口

```
    /**
     * 获取数据表的某个多元索引的详细信息。
     * @api
     *
     * @param [] $request
     *             请求参数，数据表名称。
     * @return [] 请求返回
     * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
     * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
     * @example "src/examples/DescribeSearchIndex.php" 20
     */
    public function describeSearchIndex(array $request)
```

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index'
);
$response = $otsClient->describeSearchIndex($request);
```

8.7.4. 列出多元索引列表

创建多元索引后，使用ListSearchIndex接口可以获取当前实例下或某个数据表关联的所有多元索引的列表信息。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

接口

```
/**
 * 获取该数据表下所有的多元索引名称。
 * @api
 *
 * @param [] $request
 *              请求参数，数据表名称。
 * @return [] 请求返回。
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 * @example "src/examples/ListSearchIndex.php"
 */
public function listSearchIndex(array $request)
```

参数

参数	说明
table_name	数据表名称，可以为空。 • 如果设置了数据表名称，则返回该数据表关联的所有多元索引的列表。 • 如果未设置数据表名称，则返回当前实例下所有多元索引的列表。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
);
$response = $otsClient->listSearchIndex($request);
```

8.7.5. 精确查询

TermQuery采用完整精确匹配的方式查询表中的数据，类似于字符串匹配。对于Text类型字段，只要分词后有词条可以精确匹配即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
query_type	设置查询类型为QueryTypeConst::TERM_QUERY。
field_name	要匹配的字段。
term	查询关键词，即要匹配的值。 该词不会被分词，会被当做完整词去匹配。 对于Text类型字段，只要分词后有词条可以精确匹配即可。例如某个Text类型的字段，值为“tablestore is cool”，如果分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。
sort	按照指定方式排序，详情请参见排序和翻页。
columns_to_get	是否返回所有列，包含return_type和return_names设置。 • 当设置return_type为ColumnReturnTypeConst::RETURN_SPECIFIED时，可以通过return_names指定返回的列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::TERM_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'term' => 'keyword'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_ALL,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

8.7.6. 多词精确查询

类似于TermQuery，但是TermsQuery可以指定多个查询关键词，查询匹配这些词的数据。多个查询关键词中只要有一个词精确匹配，该行数据就会被返回，等价于SQL中的In。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
offset	本次查询的开始位置。

参数	说明
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
query_type	设置查询类型为QueryTypeConst::TERMS_QUERY。
field_name	要匹配的字段。
terms	多个查询关键词，即要匹配的值。 多个查询关键词中只要有一个词精确匹配，该行数据就会被返回。
sort	按照指定方式排序，详情请参见排序和翻页。
columns_to_get	是否返回所有列，包含return_type和return_names设置。 • 当设置return_type为ColumnReturnTypeConst::RETURN_SPECIFIED时，可以通过return_names指定返回的列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 5,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::TERMS_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'terms' => array(
                    "keyword",
                    "php"
                )
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'long',
                    'order' => SortOrderConst::SORT_ORDER_DESC,
                    'mode' => SortModeConst::SORT_MODE_AVG
                )
            )
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

8.7.7. 全匹配查询

MatchAllQuery可以匹配所有行，常用于查询表中数据总行数，或者随机返回几条数据。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。

参数	说明
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置Limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
collapse	按照指定列对返回结果进行去重。
query_type	设置查询类型为QueryTypeConst::MATCH_ALL_QUERY。
sort	按照指定方式排序，详情请参见排序和翻页。
columns_to_get	是否返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_SPECIFIED时，可以通过return_names指定返回的列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'collapse' => array(
            'field_name' => 'keyword'
        ),
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
//        'sort' => array(//如果需要特定排序。
//            array(
//                'field_sort' => array(
//                    'field_name' => 'keyword',
//                    'order' => SortOrderConst::SORT_ORDER_ASC
//                )
//            ),
//        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    )
);
$response = $otsClient->search($request);
```

8.7.8. 匹配查询

MatchQuery采用近似匹配的方式查询表中的数据。对Text类型的列值和查询关键词会先按照设置好的分词器做切分，然后按照切分好后的词去查询。对于进行模糊分词的列，建议使用MatchPhraseQuery实现高性能的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
offset	本次查询的开始位置。

参数	说明
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
query_type	设置查询类型为QueryTypeConst::MATCH_QUERY。
field_name	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如当要匹配的列为Text类型时，分词类型为单字分词，则查询词为"this is"，可以匹配到“...，this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。
operator	逻辑运算符。默认为OR，表示当分词后的多个词只要有部分匹配时，则行数据满足查询条件。 如果设置operator为AND，则只有分词后的所有词都在列值中时，才表示行数据满足查询条件。
minimum_should_match	最小匹配个数。 只有当某一行数据的field_name列的值中至少包括最小匹配个数的词时，才会返回该行数据。 **说明** minimum_should_match需要与逻辑运算符OR配合使用。
columns_to_get	是否返回所有列，包含return_type和return_names设置。 • 当设置return_type为ColumnReturnTypeConst::RETURN_SPECIFIED时，可以通过return_names指定返回的列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_QUERY,
            'query' => array(
                'field_name' => 'text',
                'text' => 'ots text php keyword',
//              'operator' => QueryOperatorConst::PBAND,
                'operator' => QueryOperatorConst::PBOR,//minimum_should_match与OR配合使用。
                'minimum_should_match' => 3
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('text')
    )
);
$response = $otsClient->search($request);
```

8.7.9. 短语匹配查询

类似于MatchQuery，但是分词后多个词的位置关系会被考虑，只有分词后的多个词在行数据中以同样的顺序和位置存在时，才表示行数据满足查询条件。如果查询列的分词类型为模糊分词，则使用MatchPhraseQuery可以实现比WildcardQuery更快的模糊查询。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。

参数	说明
index_name	多元索引名称。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置Limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
query_type	设置查询类型为QueryTypeConst::MATCH_PHRASE_QUERY。
field_name	要匹配的列。 匹配查询可应用于Text类型。
text	查询关键词，即要匹配的值。 当要匹配的列为Text类型时，查询关键词会被分词成多个词，分词类型为创建多元索引时设置的分词器类型。如果创建多元索引时未设置分词器类型，则默认分词类型为单字分词。 例如查询的值为“this is”，可以匹配到“...，this is tablestore”、“this is a table”，但是无法匹配到“this table is ...”以及“is this a table”。
columns_to_get	是否返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_SPECIFIED时，可以通过return_names指定返回的列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_PHRASE_QUERY,
            'query' => array(
                'field_name' => 'text',
                'text' => 'text keyword'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('text')
    )
);
$response = $otsClient->search($request);
```

8.7.10. 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
offset	本次查询的开始位置。

参数	说明
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回。 返回匹配的总行数会影响查询性能。
query_type	设置查询类型为QueryTypeConst::PREFIX_QUERY。
field_name	要匹配的字段。
prefix	前缀值。 对于Text类型字段，只要分词后的词条中有词条满足前缀条件即可。
sort	按照指定方式排序，详情请参见排序和翻页。
columns_to_get	是否返回所有列，包含return_type和return_names设置。 • 当设置return_type为ColumnReturnTypeConst::RETURN_SPECIFIED时，可以通过return_names指定返回的列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::PREFIX_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'prefix' => 'key'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_ALL,
        'return_names' => array('keyword')
    )
);
$response = $otsClient->search($request);
```

8.7.11. 范围查询

RangeQuery根据范围条件查询表中的数据。对于Text类型字段，只要分词后的词条中有词条满足范围条件即可。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
offset	本次查询的开始位置。

参数	说明
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需获取具体数据，可以设置limit=0，即不返回任意一行数据。
get_total_count	是否返回匹配的总行数，默认为False，表示不返回。 返回匹配的总行数会影响查询性能。
query_type	设置查询类型为QueryTypeConst::RANGE_QUERY。
field_name	要匹配的字段。
range_from	起始位置的值。
range_to	结束位置的值。
include_lower	结果中是否需要包括range_from值，类型为Boolean。
include_upper	结果中是否需要包括range_to值，类型为Boolean。
sort	按照指定方式排序，详情请参见排序和翻页。
columns_to_get	是否返回所有列，包含return_type和column_names设置。 • 当设置return_type为ColumnReturnType.SPECIFIED时，可以通过column_names指定返回的列。 • 当设置return_type为ColumnReturnType.ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnType.NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::RANGE_QUERY,
            'query' => array(
                'field_name' => 'long',
                'range_from' => 100,
                'include_lower' => true,
                'range_to' => 101,
                'include_upper' => false
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('double', 'long', 'keyword')
    )
);
$response = $otsClient->search($request);
```

8.7.12. 通配符查询

通配符查询中，要匹配的值可以是一个带有通配符的字符串，目前支持星号（*）和问号（?）两种通配符。要匹配的值中可以用星号（*）代表任意字符序列，或者用问号（?）代表任意单个字符，且支持以星号（*）或问号（?）开头。例如查询“table*e”，可以匹配到“tablestore”。

如果查询的模式为*word*，则您可以使用性能更好的模糊查询，具体实现方法如下：

1. 创建多元索引时，设置列为Text类型且设置分词类型为模糊分词。
2. 使用多元索引查询数据时，使用MatchPhraseQuery且设置查询词为word。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	描述
table_name	数据表名称。
index_name	多元索引名称。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。
get_total_count	是否返回匹配的总行数，默认为false，表示不返回表中数据的总行数。 设置get_total_count为true后会影响查询性能。
query_type	设置查询类型为QueryTypeConst::WILDCARD_QUERY。
field_name	列名称。
value	带有通配符的字符串，字符串长度不能超过20个字符。
sort	按照指定方式排序，详情请参见排序和翻页。
columns_to_get	是否返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_SPECIFIED时，需要通过return_names指定返回的列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_ALL时，表示返回所有列。 • 当设置return_type为ColumnReturnTypeConst::RETURN_NONE时，表示不返回所有列，只返回主键列。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::WILDCARD_QUERY,
            'query' => array(
                'field_name' => 'keyword',
                'value' => 'key*'
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword')
    )
);
$response = $otsClient->search($request);
```

8.7.13. 地理位置查询

地理位置查询包括地理距离查询（GeoDistanceQuery）、地理长方形范围查询（GeoBoundingBoxQuery）和地理多边形范围查询（GeoPolygonQuery）三种方式。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

地理距离查询（GeoDistanceQuery）

GeoDistanceQuery根据一个中心点和距离条件查询表中的数据，当一个地理位置点到指定的中心点的距离不超过指定的值时，满足查询条件。

- 参数

参数	说明
table_name	数据表名称。

参数	说明
index_name	多元索引名称。
query	多元索引的查询语句。设置查询类型为QueryTypeConst::GEO_DISTANCE_QUERY。
field_name	列名，类型为Geopoint。
center_point	中心地理坐标点，是一个经纬度值。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。
distance	距离中心点的距离，类型为Double。单位为米。

- 示例

  查询表中geo列的值距离中心点 `30.001,120.001` 不超过1000米的数据。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::GEO_DISTANCE_QUERY,
            'query' => array(
                'field_name' => 'geo',
                'center_point' => '30.001,120.001',
                'distance' => 1000
            )
        ),
        'sort' => array(
            array(
                'geo_distance_sort' => array(
                    'field_name' => 'geo',
                    'order' => SortOrderConst::SORT_ORDER_ASC,
                    'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
                    'points' => array('30,120')
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('geo')
    )
);
$response = $otsClient->search($request);
```

地理长方形范围查询（GeoBoundingBoxQuery）

GeoBoundingBoxQuery根据一个长方形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在给出的长方形范围内时，满足查询条件。

- 参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
query	多元索引的查询语句。设置查询类型为QueryTypeConst::GEO_BOUNDING_BOX_QUERY。
field_name	列名，类型为Geopoint。
top_left	长方形框的左上角的坐标。
bottom_right	长方形框的右下角的坐标，通过左上角和右下角的坐标可以确定一个唯一的长方形。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。

- 示例

  查询表中geo列的值在左上角坐标为 `31,119` ，右下角坐标为 `29,121` 的长方形范围内的数据。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::GEO_BOUNDING_BOX_QUERY,
            'query' => array(
                'field_name' => 'geo',
                'top_left' => '31,119',
                'bottom_right' => '29,121'
            )
        ),
        'sort' => array(
            array(
                'geo_distance_sort' => array(
                    'field_name' => 'geo',
                    'order' => SortOrderConst::SORT_ORDER_ASC,
                    'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
                    'points' => array('30,120')
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('geo')
    )
);
$response = $otsClient->search($request);
```

地理多边形范围查询（GeoPolygonQuery）

GeoPolygonQuery根据一个多边形范围的地理位置边界条件查询表中的数据，当一个地理位置点落在指定的多边形范围内时，满足查询条件。

- 参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
query	多元索引的查询语句。设置查询类型为QueryTypeConst::GEO_POLYGON_QUERY。
field_name	列名，类型为Geopoint。

参数	说明
points	组成多边形范围的坐标，通过多个坐标可以确定一个唯一的多边形。 格式为 `纬度,经度` ，纬度在前，经度在后，且纬度范围为[-90,+90]，经度范围[-180,+180]。例如 `35.8,-45.91` 。

- 示例

  查询表中geo列的值在由 `31,120` 、 `29,121` 和 `29,119` 坐标组成的多边形范围内的数据。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::GEO_POLYGON_QUERY,
            'query' => array(
                'field_name' => 'geo',
                'points' => array(
                    "31,120",
                    "29,121",
                    "29,119"
                )
            )
        ),
        'sort' => array(
            array(
                'geo_distance_sort' => array(
                    'field_name' => 'geo',
                    'order' => SortOrderConst::SORT_ORDER_ASC,
                    'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
                    'points' => array('30,120')
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('geo')
    )
);
$response = $otsClient->search($request);
```

8.7.14. 列存在性查询

ExistsQuery也叫NULL查询或者空值查询，一般用于判断稀疏数据中某一行的某一列是否存在。例如查询所有数据中address列不为空的行。

> **说明**
> - 如果需要查询某一列为空，则需要和BoolQuery中的must_not_queries结合使用。
> - 以下情况会认为某一列不存在，以 "city" 列为例说明。
>   - city列在本行数据中不存在。
>   - city列是空数组，即 "city" = "[]"。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
field_name	列名。
query_type	设置查询类型为QueryTypeConst::EXISTS_QUERY。 **说明** 嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型的列的路径和一个子查询，其中子查询可以是任意Query类型。详情请参见嵌套类型查询。

示例

- 示例1

  查询表中text列的值不为空的数据。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::EXISTS_QUERY,
            'query' => array(
                'field_name' => 'text'
            )
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long', 'array')
    )
);
$response = $otsClient->search($request);
```

- 示例2

  查询嵌套类型的nested列中的nested_long子列的值不为空的数据。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::NESTED_QUERY,
            'query' => array(
                'path' => "nested",
                'query' => array(
                    'query_type' => QueryTypeConst::EXISTS_QUERY,
                    'query' => array(
                        'field_name' => 'nested.nested_long',
                    )
                ),
                'score_mode' => ScoreModeConst::SCORE_MODE_AVG
            )
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('nested')
    )
);
$response = $this->otsClient->search($request);
```

8.7.15. 折叠（去重）

当数据查询的结果中含有某种类型的数据较多时，可以使用折叠（Collapse）功能按照某一列对结果集做折叠，使对应类型的数据在结果展示中只出现一次，保证结果展示中类型的多样性。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

注意事项

- 折叠功能只能使用offset+limit方式翻页，不能使用token方式。
- 对结果集同时使用统计聚合与折叠功能时，统计聚合功能只作用于使用折叠功能前的结果集。
- 使用折叠功能后，返回的总分组数取决于offset+limit的最大值，目前支持返回的总分组数最大为10000。
- 执行结果中返回的总行数是使用折叠功能前的匹配行数，使用折叠功能后的总分组数无法获取。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
query	可以是任意Query类型。
collapse	折叠参数设置，包含field_name设置。 field_name：列名，按该列对结果集做折叠，只支持应用于整型、浮点数和Keyword类型的列，不支持数组类型的列。
offset	本次查询的开始位置。
limit	本次查询需要返回的最大数量。 如果只为了获取行数，无需具体数据，可以设置limit=0，即不返回任意一行数据。

示例

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'collapse' => array(
            'field_name' => 'keyword'
        ),
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
//        'sort' => array(//如果需要特定排序。
//            array(
//                'field_sort' => array(
//                    'field_name' => 'keyword',
//                    'order' => SortOrderConst::SORT_ORDER_ASC
//                )
//            ),
//        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    )
);
$response = $otsClient->search($request);
```

8.7.16. 多条件组合查询

BoolQuery查询条件包含一个或者多个子查询条件，根据子查询条件来判断一行数据是否满足查询条件。每个子查询条件可以是任意一种Query类型，包括BoolQuery。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
must_queries	多个Query列表，行数据必须满足所有的子查询条件才算匹配，等价于And操作符。

参数	说明
must_not_queries	多个Query列表，行数据必须不能满足任何的子查询条件才算匹配，等价于Not操作符。
filter_queries	多个Query列表，行数据必须满足所有的子filter才算匹配，filter类似于query，区别是filter不会根据满足的filter个数进行相关性算分。
should_queries	多个Query列表，可以满足，也可以不满足，等价于Or操作符。 行数据应该至少满足should_queries子查询条件的最小匹配个数才算匹配。 如果满足的should_queries子查询条件个数越多，则整体的相关性分数更高。
minimum_should_match	should_queries子查询条件的最小匹配个数。当同级没有其他Query，只有should_queries时，默认值为1；当同级已有其他Query，例如must_queries、must_not_queries和filter_queries时，默认值为0。

示例

如下示例中条件A为keyword列值精确匹配'keyword'，条件B为long列值大于等于100且小于101。

- 示例1

  通过BoolQuery进行And条件查询，查询条件为Must Queries(A AND B)，即查询同时满足条件A和条件B的行。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::BOOL_QUERY,
            'query' => array(
                'must_queries' => array(
                    array(
                        'query_type' => QueryTypeConst::TERM_QUERY,
                        'query' => array(
                            'field_name' => 'keyword',
                            'term' => 'keyword'
                        )
                    ),
                    array(
                        'query_type' => QueryTypeConst::RANGE_QUERY,
                        'query' => array(
                            'field_name' => 'long',
                            'range_from' => 100,
                            'include_lower' => true,
                            'range_to' => 101,
                            'include_upper' => false
                        )
                    )
                ),
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- 示例2

  通过BoolQuery进行Not条件查询，查询条件为MustNotQueries(!A AND !B)，即查询同时不满足条件A和条件B的行。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::BOOL_QUERY,
            'query' => array(
                'must_not_queries' => array(
                    array(
                        'query_type' => QueryTypeConst::TERM_QUERY,
                        'query' => array(
                            'field_name' => 'keyword',
                            'term' => 'keyword'
                        )
                    ),
                    array(
                        'query_type' => QueryTypeConst::RANGE_QUERY,
                        'query' => array(
                            'field_name' => 'long',
                            'range_from' => 100,
                            'include_lower' => true,
                            'range_to' => 101,
                            'include_upper' => false
                        )
                    )
                ),
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- 示例3

  通过BoolQuery进行Filter条件查询，查询条件为FilterQueries(A AND B)，即查询同时满足条件A和条件B的行，行数据不会根据满足条件个数进行相关性算分。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::BOOL_QUERY,
            'query' => array(
                'filter_queries' => array(
                    array(
                        'query_type' => QueryTypeConst::TERM_QUERY,
                        'query' => array(
                            'field_name' => 'keyword',
                            'term' => 'keyword'
                        )
                    ),
                    array(
                        'query_type' => QueryTypeConst::RANGE_QUERY,
                        'query' => array(
                            'field_name' => 'long',
                            'range_from' => 100,
                            'include_lower' => true,
                            'range_to' => 101,
                            'include_upper' => false
                        )
                    )
                ),
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- ShouldQueries(A OR B)

  查询满足条件A和条件B中至少一个条件的行。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::BOOL_QUERY,
            'query' => array(
                'should_queries' => array(
                    array(
                        'query_type' => QueryTypeConst::TERM_QUERY,
                        'query' => array(
                            'field_name' => 'keyword',
                            'term' => 'keyword'
                        )
                    ),
                    array(
                        'query_type' => QueryTypeConst::RANGE_QUERY,
                        'query' => array(
                            'field_name' => 'long',
                            'range_from' => 100,
                            'include_lower' => true,
                            'range_to' => 101,
                            'include_upper' => false
                        )
                    )
                ),
                'minimum_should_match' => 1
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
```

- 示例4

  含有子BoolQuery的组合查询。使用多条件组合查询实现(col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6))，每个and或or相当于一个BoolQuery，多个表达式的and或or就是多个BoolQuery的组合。

```
$request = array(
    'table_name' => 'php_sdk_test',
```

```
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_index',
    'search_query' => [
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => false,
        'query' => [
            'query_type' => QueryTypeConst::BOOL_QUERY,
            'query' => [
                //总组合查询条件为(col2<4 or col3<5) or (col2 = 4 and (col3 = 5 or col3 =6)
)。
                'should_queries' => [
                    [
                        'query_type' => QueryTypeConst::BOOL_QUERY,
                        'query' => [
                            //组合条件1为(col2<4 or col3<5)。
                            'should_queries' => [
                                [
                                    'query_type' => QueryTypeConst::RANGE_QUERY,
                                    //查询条件1为col2<4。
                                    'query' => [
                                        'field_name' => 'col2',
                                        'range_to' => 4
                                    ]
                                ],
                                [
                                    'query_type' => QueryTypeConst::RANGE_QUERY,
                                    //查询条件2为col3<5。
                                    'query' => [
                                        'field_name' => 'col3',
                                        'range_to' => 5
                                    ]
                                ]
                            ]
                        ]
                    ],
                    [
                        'query_type' => QueryTypeConst::BOOL_QUERY,
                        //组合条件2为(col2 = 4 and (col3 = 5 or col3 =6))。
                        'query' => [
                            'must_queries' => [
                                [
                                    'query_type' => QueryTypeConst::TERM_QUERY,
                                    //查询条件3为col2=4。
                                    'query' => [
                                        'field_name' => 'col2',
                                        'term' => 4
                                    ]
                                ],
                                [
                                    'query_type' => QueryTypeConst::BOOL_QUERY,
                                    //组合条件3为(col3 = 5 or col3 =6))。
                                    'query' => [
                                        'should_queries' => [
                                            [
```

```
                                                'query_type' => QueryTypeConst::TERM_QUER
Y,
                                                //查询条件4为col3=5。
                                                'query' => [
                                                    'field_name' => 'col3',
                                                    'term' => 5
                                                ]
                                            ],
                                            [
                                                'query_type' => QueryTypeConst::TERM_QUER
Y,
                                                //查询条件5为col3=6。
                                                'query' => [
                                                    'field_name' => 'col3',
                                                    'term' => 6
                                                ]
                                            ]
                                        ],
                                        'minimum_should_match' => 1
                                    ]
                                ]
                            ]
                        ]
                    ],
                    'minimum_should_match' => 1
                ]
            ]
    ],
    //通过设置columns_to_get参数可以指定返回的列或返回所有列，如果不设置此参数，则默认只返回主键列
。
    'columns_to_get' => [
        //'return_type' => ColumnReturnTypeConst::RETURN_ALL //设置为返回所有列。
        //设置为返回指定列。
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    ]
);
```

8.7.17. 嵌套类型查询

NestedQuery用于查询嵌套类型字段中子行的数据。嵌套类型不能直接查询，需要通过NestedQuery包装，NestedQuery中需要指定嵌套类型字段的路径和一个子查询，其中子查询可以是任意Query类型。

前提条件

- 已初始化OTSClient。具体操作，请参见初始化。
- 已创建数据表并写入数据。
- 已在数据表上创建多元索引。具体操作，请参见创建多元索引。

参数

参数	说明
table_name	数据表名称。
index_name	多元索引名称。
path	路径名，嵌套类型的列的树状路径。例如news.title表示嵌套类型的news列中的title子列。
query	嵌套类型的列中子列上的查询，子列上的查询可以是任意Query类型。
score_mode	当列存在多个值时基于哪个值计算分数。

示例

查询表中nested.nested_keyword列值为'sub'的数据。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 2,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::NESTED_QUERY,
            'score_mode' => ScoreModeConst::SCORE_MODE_AVG,
            'query' => array(
                'path' => "nested",
                'query' => array(
                    'query_type' => QueryTypeConst::TERM_QUERY,
                    'query' => array(
                        'field_name' => 'nested.nested_keyword',
                        'term' => 'sub'
                    )
                )
            )
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'nested.nested_long',
                    'order' => SortOrderConst::SORT_ORDER_DESC,
                    'nested_filter' => array(
                        'path' => "nested",
                        'query' => array(
                            'query_type' => QueryTypeConst::TERM_QUERY,
                            'query' => array(
                                'field_name' => 'nested.nested_keyword',
                                'term' => 'sub'
                            )
                        )
                    )
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('nested')
    )
);
$response = $otsClient->search($request);
```

8.7.18. 排序和翻页

您可以在创建多元索引时指定索引预排序和在使用多元索引查询数据时指定排序方式，以及在获取返回结果时使用limit和offset或者使用token进行翻页。

索引预排序

多元索引默认按照设置的索引预排序（IndexSort）方式进行排序，使用多元索引查询数据时，IndexSort决定了数据的默认返回顺序。

在创建多元索引时，您可以自定义IndexSort，如果未自定义IndexSort，则IndexSort默认为主键排序。

> **注意** 含有Nested类型字段的多元索引不支持索引预排序。

查询时指定排序方式

只有enable_sort_and_agg设置为true的字段才能进行排序。

在每次查询时，可以指定排序方式，多元索引支持如下四种排序方式（Sorter）。您也可以使用多个Sorter，实现先按照某种方式排序，再按照另一种方式排序的需求。

- ScoreSort

  按照查询结果的相关性（BM25算法）分数进行排序，适用于有相关性的场景，例如全文检索等。

  > **注意** 如果需要按照相关性打分进行排序，必须手动设置ScoreSort，否则会按照索引设置的IndexSort进行排序。

  ```
  'sort' => array(
      array(
          'score_sort' => array(
              'order' => SortOrderConst::SORT_ORDER_DESC
          )
      ),
  )
  ```

- PrimaryKeySort

  按照主键进行排序。

  ```
  'sort' => array(
      array(
          'pk_sort' => array(
              'order' => SortOrderConst::SORT_ORDER_ASC
          )
      ),
  )
  ```

- FieldSort

  按照某列的值进行排序。

```
'sort' => array(
    array(
        'field_sort' => array(
            'field_name' => 'keyword',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
)
```

- GeoDistanceSort

  根据地理点距离进行排序。

```
'sort' => array(
    array(
        'geo_distance_sort' => array(
            'field_name' => 'geo',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'distance_type' => GeoDistanceTypeConst::GEO_DISTANCE_PLANE,
            'points' => array('0.6,0.6')
        )
    ),
)
```

- 多类型组合排序

  先按照某列进行排序，再按照另一列进行排序。

```
'sort' => array(
    array(
        'field_sort' => array(
            'field_name' => 'keyword',
            'order' => SortOrderConst::SORT_ORDER_ASC,
            'mode' => SortModeConst::SORT_MODE_AVG,
        )
    ),
    array(
        'pk_sort' => array(
            'order' => SortOrderConst::SORT_ORDER_ASC
        )
    ),
)
```

翻页方式

在获取返回结果时，可以使用limit和offset或者使用token进行翻页。

- 使用limit和offset进行翻页

  当需要获取的返回结果行数小于10000行时，可以使用limit和offset进行翻页，即limit+offset<=10000，其中limit的最大值为100。

  > **说明** 如果需要提高limit的上限，请参见如何将多元索引Search接口查询数据的limit提高到1000。

如果使用此方式进行翻页时未设置limit和offset，则limit的默认值为10，offset的默认值为0。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 10,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::MATCH_ALL_QUERY
        ),
        'sort' => array(
            array(
                'field_sort' => array(
                    'field_name' => 'keyword',
                    'order' => SortOrderConst::SORT_ORDER_ASC
                )
            ),
        ),
        'token' => null,
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('col1', 'col2')
    )
);
$response = $otsClient->search($request);
```

- 使用token进行翻页

  由于使用token进行翻页时翻页深度无限制，当需要进行深度翻页时，推荐使用token进行翻页。

  当符合查询条件的数据未读取完时，服务端会返回next_token，此时可以使用next_token继续读取后面的数据。

  使用token进行翻页时默认只能向后翻页。由于在一次查询的翻页过程中token长期有效，您可以通过缓存并使用之前的token实现向前翻页。

  使用token翻页后的排序方式和上一次请求的一致，无论是系统默认使用IndexSort还是自定义排序，因此设置了token不能再设置Sort。另外使用token后不能设置offset，只能依次往后读取，即无法跳页。

  > **注意** 由于含有Nested类型字段的多元索引不支持索引预排序，如果使用含有Nested类型字段的多元索引查询数据且需要翻页，则必须在查询条件中指定数据返回的排序方式，否则当符合查询条件的数据未读取完时，服务端不会返回next_token。

```
$request = array(
    'table_name' => 'php_sdk_test',
    'index_name' => 'php_sdk_test_search_index',
    'search_query' => array(
        'offset' => 0,
        'limit' => 10,
        'get_total_count' => true,
        'query' => array(
            'query_type' => QueryTypeConst::FUNCTION_SCORE_QUERY,
            'query' => array(
                'query' => array(
                    'query_type' => QueryTypeConst::TERM_QUERY,
                    'query' => array(
                        'field_name' => 'keyword',
                        'term' => 'keyword'
                    )
                ),
                'field_value_factor' => array(
                    'field_name' => 'long'
                )
            )
        ),
        'sort' => array(
            array(
                'score_sort' => array(
                    'order' => SortOrderConst::SORT_ORDER_DESC
                )
            ),
        )
    ),
    'columns_to_get' => array(
        'return_type' => ColumnReturnTypeConst::RETURN_SPECIFIED,
        'return_names' => array('keyword', 'long')
    )
);
$response = $otsClient->search($request);
print "total_hits: " . $response['total_hits'] . "\n";
print json_encode($response['rows'], JSON_PRETTY_PRINT);
while($response['next_token'] != null) {
    $request['search_query']['token'] = $response['next_token'];
    $request['search_query']['sort'] = null;//当有next_token时，不能再设置Sort，token中包含sort信息。
    $response = $otsClient->search($request);
    print json_encode($response['rows'], JSON_PRETTY_PRINT);
}
```

8.8. 全局二级索引

在数据表上创建索引表后，可根据需要读取索引表中的数据或者删除数据表上指定的索引表。

前提条件

- 已初始化Client，详情请参见初始化。

- 已创建数据表，且数据表的数据生命周期（timeToLive）必须为-1，最大版本数（maxVersions）必须为1。
- 数据表已设置预定义列。

创建索引表（CreateIndex）

使用CreateIndex接口在已存在的数据表上创建索引表。

> **说明** 您也可以使用CreateTable接口在创建数据表的同时创建一个或者多个索引表，详情请参见创建数据表。

- 接口

```
/**
 * 创建二级索引。
 * @api
 *
 * @param [] $request
 *            请求参数，数据表名称。
 * @return [] 请求返回
 * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
 * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
 * @example "src/examples/CreateIndex.php"
 */
public function createIndex(array $request)
```

- 参数

参数	说明
main_table_name	数据表名称。
index_meta	索引表的结构信息，包括如下内容： ○ name：索引表名称。 ○ primary_key：索引表的索引列，索引列为数据表主键和预定义列的任意组合。 ○ defined_column：索引表的属性列，索引表属性列为数据表的预定义列的组合。 ○ index_update_mode：索引表更新模式，当前只支持IUM_ASYNC_INDEX。 ○ index_type：索引表类型，当前只支持IT_GLOBAL_INDEX。
include_base_data	索引表中是否包含数据表中已存在的数据。 当设置include_base_data为true时，表示包含存量数据；设置include_base_data为false时，表示不包含存量数据。

- 示例

```
$request = array(
    'table_name' => 'MainTableName',
    'index_meta' => array(
        'name' => self::$indexName1,
        'primary_key' => array('col1'),
        'defined_column' => array('col2')
    )
);
$this->otsClient->createIndex($request);
```

读取索引表中数据

从索引表中单行或者范围读取数据，当返回的属性列在索引表中时，可以直接读取索引表获取数据，否则请自行反查数据表获取数据。

- 单行读取索引表中数据

  详情请参见单行数据操作。

  使用GetRow接口读取索引表中数据时有如下注意事项：

  - table_name需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置行的主键时，需要同时设置索引表索引列和补齐的数据表主键。

- 范围读索引表中数据

  详情请参见多行数据操作。

  使用GetRange接口读取索引表中数据时有如下注意事项：

  - table_name需要设置为索引表名称。
  - 由于系统会自动将未出现在索引列中的数据表主键补齐到索引表主键中，所以设置起始主键和结束主键时，需要同时设置索引表索引列和补齐的数据表主键。

删除索引表（DropIndex）

使用DropIndex接口删除数据表上指定的索引表。

- 接口

```
    /**
     * 删除全局二级索引。
     * @api
     *
     * @param [] $request
     *            请求参数，数据表名称。
     * @return [] 请求返回。
     * @throws OTSClientException 当参数检查出错或服务端返回校验出错时抛出异常。
     * @throws OTSServerException 当OTS服务端返回错误时抛出异常。
     * @example "src/examples/DropIndex.php"
     */
    public function dropIndex(array $request)
```

- 参数

参数	说明
table_name	数据表名称。
index_name	索引表名称。

- 示例

```
$request = array (
    'table_name' => 'MainTableName',
    'index_name' => 'IndexName'
);
$otsClient->dropIndex( $request );
```

8.9. 错误处理

本文介绍表格存储PHP SDK的错误处理。

方式

表格存储PHP SDK目前采用“异常”的方式处理错误，如果调用接口没有抛出异常，则说明操作成功，否则失败。

> **说明** 批量相关接口，例如BatchGetRow和BatchWriteRow不仅需要判断是否有异常，还需要检查每行的状态是否成功，只有全部成功后才能保证整个接口调用是成功的。

异常

表格存储PHP SDK中有OTSClientException和OTSServerException两种异常，都最终继承自Exception。

- OTSClientException：指SDK内部出现的异常，例如参数设置错误等。
- OTSServerException：指服务器端错误，来自于对服务器错误信息的解析。OTSServerException包含以下几个成员：
  - getHttpStatus()：HTTP返回码，例如200、404等。
  - getOTSErrorCode()：表格存储返回的错误类型字符串。常见错误请参见错误码参考。
  - getOTSErrorMessage()：表格存储返回的错误详细描述。
  - getRequestId()：用于唯一标识此次请求的UUID。当您无法解决问题时，记录此RequestId并提交工单。

9.历史版本 SDK 下载

9.1. Java SDK历史迭代版本

本文介绍表格存储Java SDK的历史迭代版本。

> **说明** 4.0.0以上版本的SDK支持数据多版本和生命周期，且不兼容2.x.x系列的SDK。

版本号：5.13.5

发布时间：2022-01-06

下载地址：5.13.5

更新日志：

- 时序模型中支持删除时间线元数据。具体操作，请参见删除时间线元数据。
- 时序模型中查询时序数据时支持倒序读取。具体操作，请参见查询时序数据。

版本号：5.13.0

发布时间：2021-11-17

下载地址：5.13.0

更新日志：新增SQL查询功能。更多信息，请参见SQL查询。

版本号：5.12.0

发布时间：2021-10-27

下载地址：5.12.0

更新日志：新增时序模型。更多信息，请参见时序模型概述。

版本号：5.11.0

发布时间：2021-07-23

下载地址：5.11.0

更新日志：

- 多元索引新增虚拟列功能。具体操作，请参见虚拟列。
- 多元索引新增百分位统计和直方图统计功能。具体操作，请参见统计聚合。

版本号：5.10.3

发布时间：2020-10-26

下载地址：5.10.3

更新日志：

- 新增数据湖投递功能。通过控制台或SDK的具体操作，请分别参见快速入门或者使用SDK。
- 多元索引新增获取统计聚合分组内的行功能。具体操作，请参见统计聚合。
- 新增对配置参数的格式检查。
- SDK内部优化与计算引擎的对接。

版本号：5.7.0

发布时间：2020-04-07

下载地址：5.7.0

更新日志：

- 多元索引新增并发导出数据功能。具体操作，请参见并发导出数据。
- 优化TableStoreWriter。

版本号：5.4.0

发布时间：2019-11-18

下载地址：5.4.0

更新日志：

- 支持已有的表新增或删除预定义列。
- 通道功能进行了SDK内部优化。

版本号：5.3.0

发布时间：2019-09-10

下载地址：5.3.0

更新日志：多元索引新增统计聚合功能。

版本号：5.1.0

发布时间：2019-07-01

下载地址：5.1.0

更新日志：多元索引新增更多分词类型。

版本号：5.0.0

发布时间：2019-06-10

下载地址：5.0.0

更新日志：

- 为了各SDK版本号的统一，版本号提升到5.x，与4.x版本SDK接口兼容。
- 多元索引新增exists query。

版本号：4.12.1

发布时间：2019-05-08

下载地址：4.12.1

更新日志：新增TimelineV2模型。

版本号：4.11.0

发布时间：2019-04-12

下载地址：4.11.0

更新日志：新增Timestream模型。

版本号：4.10.0

发布时间：2019-02-20

下载地址：tablestore-4.10.0

更新日志：新增通道服务功能

- 新增通道服务管控接口。
- 新增全增量一体化数据处理框架。

版本号：4.8.0

发布时间：2018-12-17

下载地址：tablestore-4.8.0-release.zip

更新日志：

- 多元索引新增和优化的功能。
  - 支持设置IndexSort。
  - 使用Token替换SearchAfter进行翻页。
  - 新增TermsQuery。
  - 支持设置TotalCount是否返回（默认不返回）。
- 日志配置和线程名调整。

版本号：4.7.4

发布时间：2018-09-27

下载地址：tablestore-4.7.4-release.zip

更新日志：

- 新增多元索引功能。
  - 多字段检索
  - 范围查询
  - 通配符查询
  - 嵌套查询
  - 全文检索
  - 排序
- 新增全局二级索引。

版本号：4.3.1

发布时间：2017-04-27

下载地址：tablestore-4.3.1-release.zip

更新日志：提供按照指定大小将全表数据逻辑分片的功能。

版本号：4.2.1

发布时间：2017-01-18

下载地址：aliyun_tablestore_java_sdk_4.2.1.zip

更新日志：修复PrimaryKeyValue和PrimaryKeySchema无法放入HashMap的问题。

版本号：4.2.0

发布时间：2016-11-29

下载地址：aliyun_tablestore_java_sdk_4.2.0.zip

更新日志：新增主键列自增功能。

版本号：4.1.0

发布时间：2016-10-11

下载地址：aliyun_tablestore_java_sdk_4.1.0.zip

更新日志：提供DescribeTable接口返回分片间的分隔点，可用于确定分片范围。

版本号：4.0.0

发布时间：2016-08-01

下载地址：aliyun_tablestore_java_sdk_4.0.0.zip

更新日志：

- 新增数据生命周期TTL。
- 新增数据多版本。

版本号：2.2.5

发布时间：2016-08-23

下载地址：aliyun_tablestore_java_sdk_2.2.5.zip

更新日志：解决OTSWriter中可能导致程序Hang的一个bug。

版本号：2.2.4

发布时间：2016-05-12

下载地址：aliyun_tablestore_java_sdk_2.2.4.zip

更新日志：

- 新增condition update。
- 新增filter。

版本号：2.1.2

发布时间：2015-12-31

下载地址：aliyun_ots_java_sdk_2.1.2.zip

更新日志：根据按量计费方式，重新调整了示例代码中的预留CU设置。

版本号：2.1.1

发布时间：2015-12-30

下载地址：aliyun-ots-java-sdk-2.1.1.zip

更新日志：由于JodaTime 2.4在Java 8下有序列化时间格式不正确的bug，所以将SDK依赖的JodaTime版本从2.4提升到2.9.1。

版本号：2.1.0

发布时间：2015-11-12

下载地址：aliyun-ots-java-sdk-2.1.0.zip

更新日志：

- 网络传输异步化及一系列性能调优：同等CPU使用情况下QPS提升数倍。
- 提供灵活易用的异步接口：支持传入callback，同时返回future。
- 解除与OSS SDK的绑定：新的SDK只包含表格存储SDK相关代码，目录结构有细微调整。
- 重试逻辑优化：优化默认的重试逻辑；支持batch操作中单行错误单独重试；提供更清晰的重试逻辑自定义方式。
- 日志优化：支持请求发送到接收的各个环节的日志记录；支持记录慢请求日志；通过TraceId打通SDK与后端服务的全链条日志。
- 提供支持批量数据导入的OTSWriter接口：旨在提供易用、高效的数据导入体验。
- 其他优化：丰富各种数据类型的工具函数；提供计算数据大小的接口等。

重要提示：

2.1.0与2.0.4版本有细微不兼容之处，如下所示。更多信息，请参见老版SDK迁移教程。

- 替换新的SDK后需要更改少数几个类的import路径。因为有几个类的package有调整，例如ClientConfiguration的package由com.aliyun.openservices调整为com.aliyun.openservices.ots。调整package的主要原因是表格存储的SDK已经与OSS SDK分离，之前公用的几个类放在ots的package下更为合理。
- 当您不再使用OTSClient实例时（例如程序结束前），需要主动调用OTSClient的shutdown方法，释放OTSClient对象占有的线程和连接资源。
- ClientConfiguration中部分配置项的名称有调整，例如在配置项名称中加入了时间单位。
- 新SDK的包依赖有变化，例如使用了HttpAsyncClient和Jodatime等，如果您在运行中有遇到问题，需要考虑是否引入了冲突的依赖。

版本号：2.0.4

发布时间：2015-09-25

下载地址：aliyun-ots-java-sdk-2.0.4.zip

9.2. Node.js SDK历史迭代版本

本文介绍表格存储Node.js SDK历史迭代版本。

版本号：4.0.6

发布时间：2016-10-09

下载地址：aliyun-tablestore-nodejs-sdk-4.0.6.tar.gz

GitHub地址：v4.0.6

更新日志：

- 添加对新特性async await的支持，添加相关示例代码。

- 修复部分中文读取出错的问题。
- 添加对STSToken的支持。
- UpdateTable接口添加对maxTimeDeviation条件的支持。
- Readme文档中添加version、build、coverage、license徽标。

版本号：4.0.3

发布时间：2016-08-27

下载地址：aliyun-tablestore-nodejs-sdk-4.0.3.tar.gz

GitHub地址：v4.0.3

更新日志：

- 统一版本信息。

版本号：4.0.1

发布时间：2017-08-27

下载地址：aliyun-tablestore-nodejs-sdk-4.0.1.tar.gz

GitHub地址：v4.0.1

更新日志：

- 移除部分无用的代码。
- 修复当GetRange返回0条数据时的值。

版本号：4.0.0

发布时间：2017-07-07

下载地址：aliyun-tablestore-nodejs-sdk-4.0.0.tar.gz

GitHub地址：v4.0.0

9.3. Python SDK 历史迭代版本

本文介绍表格存储 Python SDK的历史迭代版本。

版本号：5.2.1

发布时间：2021-02-20

下载地址：aliyun-tablestore-python-sdk-5.2.1.tar.gz

更新日志：多元索引新增统计聚合功能。具体操作，请参见统计聚合。

版本号：5.1.0

发布时间：2019-07-22

下载地址：aliyun-tablestore-python-sdk-5.1.0.tar.gz

更新日志：新增局部事务功能。具体操作，请参见局部事务。

版本号：5.0.0

发布时间：2019-05-29

下载地址：aliyun-tablestore-python-sdk-5.0.0.tar.gz

更新日志：

- 多元索引新增列存在性查询。
- 解决next_token不存在时的问题。

版本号：4.6.0

发布时间：2019-02-15

下载地址：aliyun-tablestore-python-sdk-4.6.0.tar.gz

更新日志：

- 新增多元索引功能。
- 新增全局二级索引功能。

版本号：4.3.7

发布时间：2018-05-10

下载地址： aliyun-tablestore-python-sdk-4.3.7.tar.gz

更新日志：解决bytearray类型的编码问题。

版本号：4.3.4

发布时间：2018-03-12

下载地址： aliyun-tablestore-python-sdk-4.3.4.tar.gz

更新日志：使用官方protobuf包替换第三方的protobuf-py3包。

版本号：4.3.2

发布时间：2018-01-08

下载地址： aliyun-tablestore-python-sdk-4.3.2.tar.gz

更新日志：移除对crcmod的依赖。

版本号：4.3.0

发布时间：2017-12-12

下载地址：aliyun-tablestore-python-sdk-4.3.0.tar.gz

更新日志：新增支持python 3（python3.3、python3.4、python3.5和python3.6）。

版本号：4.2.0

发布时间：2017-09-12

下载地址：aliyun-tablestore-python-sdk-4.2.0.tar.gz

更新日志：新增支持STS。

版本号：4.1.0

发布时间：2017-06-28

下载地址：aliyun-tablestore-python-sdk-4.1.0.tar.gz

更新日志：新增支持python 2.6。

版本号：4.0.0

发布时间：2017-06-27

下载地址：aliyun-tablestore-python-sdk-4.0.0.tar.gz

更新日志：

- 新增数据多版本功能。更多信息，请参见数据版本和生命周期。
- 新增数据生命周期功能。更多信息，请参见数据版本和生命周期。
- 新增主键列自增功能。具体操作，请参见主键列自增。

版本号：2.1.0

发布时间：2016-10-15

下载地址：aliyun-ots-python-sdk-2.1.0.zip

更新日志：

- 新增条件更新功能。具体操作，请参见条件更新。
- 新增过滤器功能。具体操作，请参见过滤器。
- 修复SDK内部重试无效的Bug。

版本号：2.0.8

发布时间：2016-03-30

下载地址：aliyun-ots-python-sdk-2.0.8.zip

更新日志：调整连接池参数支持 HTTPS 访问和证书验证。

版本号：2.0.7

发布时间：2015-12-30

下载地址：aliyun-ots-python-sdk-2.0.7.zip

更新日志：根据按量计费方式，重新调整了示例代码中的预留CU设置。

版本号：2.0.6

发布时间：2015-10-23

下载地址：aliyun-ots-python-sdk-2.0.6.zip

更新日志：调整了部分异常情况下的重试退避策略。

版本号：2.0.5

发布时间：2015-09-25

下载地址：ots_python_sdk-2.0.5.zip

9.4. .NET SDK历史迭代版本

本文介绍表格存储.NET SDK的历史迭代版本。

版本号：4.1.4

发布时间：2019-06-24

下载地址：4.1.4

更新日志：修复多元索引中分词器默认值为SingleWord类型的问题。

版本号：4.1.2

发布时间：2019-05-29

下载地址：4.1.2

更新日志：修复多元索引中非TEXT类型可以创建分词的问题。

版本号：4.1.1

发布时间：2019-03-06

下载地址：4.1.1

更新日志：修复多元索引部分场景乱码的问题。

版本号：4.1.0

发布时间：2018-12-17

下载地址：4.1.0

更新日志：

- 新增原子计数器功能。更多信息，请参见原子计数器。
- 新增全局二级索引功能。更多信息，请参见全局二级索引。
- 新增多元索引功能。更多信息，请参见创建多元索引。

版本号：4.0.0

发布时间：2018-09-18

下载地址：4.0.0

更新日志：

- 新增数据版本和生命周期功能。更多信息，请参见数据版本和生命周期。
- 新增主键列自增功能。更多信息，请参见主键列自增。

版本号：3.0.0

发布时间：2016-05-05

下载地址：aliyun_table_store_dotnet_sdk_3.0.0.zip

更新日志：

- 新增filter。
- 消除编译时的warning。
- 清理没用的依赖包、没用的代码。
- 精简了模板调用相关的代码。
- 增加非法参数检查。
- trim用户的配置参数。

- 增加UserAgent头部。
- 删除了Condition.IGNORE、Condition.EXPECT_EXIST和Condition.EXPECT_NOT_EXIST。
- DLL文件名称由Aliyun.dll更名为Aliyun.TableStore.dll。
- 开源到GitHub。

版本号：2.2.1

发布时间：2016-04-14

下载地址：aliyun-ots-dotnet-sdk-2.2.1.zip

更新日志：SDK内部在对HTTP响应头做校验的时候，对大小写进行忽略。

版本号：2.2.0

发布时间：2016-04-05

下载地址：aliyun-ots-dotnet-sdk-2.2.0.zip

更新日志：

- 连接池的默认连接个数从50调整到300。
- 新增conditional update功能。

版本号：2.1.0

发布时间：2015-12-30

下载地址：aliyun-ots-dotnet-sdk-2.1.0.zip

更新日志：

- 根据按量计费方式，重新调整了示例代码中的预留CU设置。
- 丰富了BatchGetRow和BatchWriteRow测试用例。

9.5. PHP SDK历史迭代版本

本文介绍表格存储PHP SDK历史迭代版本。

版本号5.0.0

发布时间：2019-05-30

下载地址：aliyun_tablestore_php_sdk_5.0.0.tar.gz

更新日志

- 新功能：支持局部事务。
- 新功能：支持多元索引。
- 新功能：支持二级索引。

版本号4.1.0

发布时间：2018-07-24

下载地址：aliyun_tablestore_php_sdk_4.1.0.tar.gz

更新日志：

支持Stream基础接口。

版本号4.0.0

发布时间：2018-06-25

下载地址：aliyun_tablestore_php_sdk_4.0.0.tar.gz

更新日志：

- 支持5.5以上PHP版本，包括5.5、5.6、7.0、7.1、7.2等版本，只支持64位的PHP系统，推荐使用PHP7。
- 新功能：支持TTL设置，createTable, updateTable新增table_options参数。
- 新功能：支持多版本，putRow, updateRow, deleteRow, batchGetRow均支持timestamp设置，getRow, getRange, BatchGet等接口支持max_versions过滤。
- 新功能：支持主键列自增功能，接口新增return_type，返回新增primary_key，返回对应操作的primary_key。
- 变更：底层protobuf升级成谷歌官方版本protobuf-php库。
- 变更：各接口的primary_key变更成list类型，保证顺序性。
- 变更：各接口的attribute_columns变更成list类型，以支持多版本功能。

版本号2.1.1

发布时间：2017-01-14

下载地址：aliyun-tablestore-php-sdk-2.1.1.zip

更新日志：

支持32位操作系统。

版本号2.1.0

发布时间：2016-11-16

下载地址：aliyun-ots-php-sdk-2.1.0.zip

更新日志：

- 支持ConditionalUpdate和Filter功能。
- 新增了方便SDK使用的常量类。
- 兼容PHP 5.5以上版本。

版本号2.0.3

发布时间：2016-05-18

下载地址：aliyun-ots-php-sdk-2.0.3.zip

更新日志：

删除示例中循环删除表的代码。

版本号2.0.2

发布时间：2016-04-11

下载地址：aliyun-ots-php-sdk-2.0.2.zip

更新日志：

修复了pb decode的时候，会将有符号的整数解释为无符号的整数导致无法写入负数的情况。

版本号2.0.1

发布时间：2015-12-30

下载地址：aliyun-ots-php-sdk-2.0.1.zip

更新日志：

根据按量计费方式，重新调整了示例代码中的预留CU设置。

版本号2.0.0

发布时间：2015-09-22

下载地址：aliyun-ots-php-sdk-2.0.0.zip

更新日志：

- 包含表格存储的所有接口。
- 兼容PHP 5.3、5.4、5.5和5.6版本。
- 包含标准的重试策略。
- 使用Guzzle Http Client作为网络库。
- 使用composer作为依赖管理和工程组织工具。
- 使用phpDocumentor 2生成HTML格式的编程文档。