阿里云
表格存储

SDK 参考

文档版本：20181218

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
	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。	 **禁止：** 重置操作将丢失用户配置数据。
	该类警示信息可能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，恢复业务所需时间约10分钟。
	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。	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。	单击 **确定**。
`courier` 字体	命令。	执行 `cd /d C:/windows` 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。	`swich` *`{stand | slave}`*

目录

1 SDK 概览

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*。

表格存储支持以下主流语言的SDK包。

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

2 Java SDK

2.1 前言

本文档主要介绍Table Store Java SDK的安装和使用，适用4.0.0以上版本。

前提条件

- 您已经开通了表格存储并创建了*Access Key*。
- 4.0.0以上版本SDK支持数据多版本和生命周期。您已经了解了新增数据生命周期*TTL*以及新增数据多版本。

兼容性

当前最新版本：4.8.0

- 对于4.x.x系列的SDK：兼容
- 对于2.x.x系列的SDK：不兼容

历史版本

版本迭代详情参考*Java SDK*历史迭代版本。

2.2 安装

本文主要介绍如何安装表格存储的Java SDK。

环境准备

适用于JDK 6及以上版本。

安装方式

- 在Maven项目中加入依赖项

  在Maven工程中使用Table Store Java SDK，只需在pom.xml中加入相应依赖即可。以 4.8.0 版本为例，在 <dependencies> 内加入如下内容：

  ```
  <dependency>
      <groupId>com.aliyun.openservices</groupId>
      <artifactId>tablestore</artifactId>
      <version>4.8.0</version>
  </dependency>
  ```

- Eclipse中导入JAR包

以4.8.0版本为例，步骤如下：

1. 下载*Java SDK*开发包。
2. 解压该开发包。
3. 将解压后文件夹中的文件 `tablestore-<versionId>.jar` 以及lib文件夹下的所有文件拷贝到您的项目中。
4. 在Eclipse中选择您的工程，右击选择 **Properties** > **Java Build Path** > **Add JARs**。
5. 选中您在第3步拷贝的所有JAR文件。

示例程序

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

2.3 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。

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。

参考如下步骤获取实例的 endpoint：

1. 登录表格存储控制台。
2. 单击实例名称，进行实例详情页，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：

- 主帐号的 AccessKeyId 和 AccessKeySecret。创建步骤如下：
  1. 在阿里云官网注册阿里云帐号。
  2. 登录*AccessKey*管理控制台创建 AccessKeyId 和 AccessKeySecret。
- 被授予访问表格存储权限的子帐号的 AccessKeyId 和 AccesskeySecret。创建步骤如下：
  1. 使用主帐号前往访问控制 *RAM*，创建一个新的子帐号或者使用已经存在的子帐号。
  2. 使用主帐号授予子帐号访问表格存储的权限。

     子帐号被授权后，就可以使用自己的 AccessKeyId 和 AccessKeySecret 访问了。
- 临时访问的 STS token。获取步骤如下：
  1. 应用的服务器通过访问 RAM/STS 服务，获取一个临时的 Accesskeyld、AccesskeySecret 和 token，发送给使用方。
  2. 使用方使用上述临时密钥访问表格存储服务。

初始化对接步骤

在获取到 AccessKeyId 和 AccessKeySecret 等密钥之后，您可以按照下面步骤进行初始化对接。

新建 Client

用户使用表格存储的 SDK时，必须首先构造一个 Client，通过调用这个 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。

表格存储新版的 SDK 提供了两种 Client，SyncClient 和 AsyncClient，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。

示例代码

1. 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);
```

2. 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 ClientConf
iguration();
        // 设置建立连接的超时时间。
        clientConfiguration.setConnectionTimeoutInMillisecond(5000);
        // 设置socket超时时间。
        clientConfiguration.setSocketTimeoutInMillisecond(5000);
        // 设置重试策略，若不设置，采用默认的重试策略。
        clientConfiguration.setRetryStrategy(new AlwaysRetryStrategy
());
        SyncClient client = new SyncClient(endPoint, accessId,
accessKey,
                instanceName, 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。

2.4 表操作

2.4.1 概述

表格存储的 Java SDK 提供了多种表级别的操作接口：

创建表：创建普通表及带索引的表。

创建全局二级索引：为已创建的主表创建索引表。

更新表：更新修改部分表配置息。

查询表描述信息：查看表配置信息。

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：将全表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。

删除全局二级索引：删除一个主表的指定索引表。

2.4.2 创建表

创建表时需要指定表的结构信息（TableMeta）和配置信息（TableOptions），也可以根据需求设置表的预留读/写吞吐量（ReservedThroughput）。

说明：

表格创建好后服务端需要将表的分片加载到某个节点上，需要等待几秒钟才能对表进行读写，否则会出现异常。

参数说明

- **TableMeta**

  TableMeta 包含 TableName 和 List。

参数	定义	说明
TableName	表名	无
List	表的主键定义	• 表格存储可包含多个主键列。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。 • 第一列主键作为分片键。分片键相同的数据会存放在同一个分片内，所以相同分片键下最好不要超过 10 G 以上数据，否则会导致单分片过大，无法分裂。另外，数据的读/写访问最好在不同的分片键上均匀分布，有利于负载均衡。 • 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。

- **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。更多关于时间戳，参见数据模型。 • 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。
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。 • 可在建表时指定，也可通过 UpdateTable 接口修改。
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。

  — 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。

- **IndexMetas**

  索引表的 Meta 信息。使用表格存储创建一张数据表时，可以同时为其创建多张索引表。

参数	定义	说明
IndexName	索引表名字	无

参数	定义	说明
List<String> primaryKey	索引表类型	当前只支持IT_GLOBAL_INDEX。
List<String> definedColumns	索引表的属性列	必需为主表的预定义列组合。
IndexType	索引表类型	当前只支持IT_GLOBAL_INDEX。
IndexUpdateMode	索引表更新模式	当前只支持IUM_ASYNC_INDEX。

示例

- 创建表（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，单位秒, -1代表永不过期，例
如设置过期时间为一年, 即为 365 * 24 * 3600。
        int maxVersions = 3; // 保存的最大版本数，设置为3即代表每列上最多保
存3个最新的版本。
        TableOptions tableOptions = new TableOptions(timeToLive,
maxVersions);
        CreateTableRequestEx request = new CreateTableRequestEx(
tableMeta, tableOptions);
        request.setReservedThroughput(new ReservedThroughput(new
CapacityUnit(0, 0))); // 设置读写预留值，容量型实例只能设置为0，高性能实例可
以设置为非零值。
        client.createTable(request);
    }
```

- 创建表（带索引）

```
   private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(
PRIMARY_KEY_NAME_1, PrimaryKeyType.STRING)); // 为主表添加主键列。
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(
PRIMARY_KEY_NAME_2, PrimaryKeyType.INTEGER)); // 为主表添加主键列。
        tableMeta.addDefinedColumn(new DefinedColumnSchema(
DEFINED_COL_NAME_1, DefinedColumnType.STRING)); // 为主表添加预定义列。
        tableMeta.addDefinedColumn(new DefinedColumnSchema(
DEFINED_COL_NAME_2, DefinedColumnType.INTEGER)); // 为主表添加预定义
列。

        int timeToLive = -1; // 数据的过期时间, 单位秒, -1代表永不过期. 带
索引表的主表数据过期时间必须为-1。
        int maxVersions = 1; // 保存的最大版本数, 带索引表的请表最大版本数
必须为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, indexMetas); // 创建主表时一同创建索引表。

        client.createTable(request);
    }
```

2.4.3 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。更多关于时间戳，参见数据模型。 • 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。
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。

参数	定义	说明
		• 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);
}
```

2.4.4 列出表名称

您可以使用列出表名称（ListTable）接口获取当前实例下已创建的所有表的表名。

示例

```
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);
    }
```

```
}
```

2.4.5 指定大小计算分片

指定大小计算分片（ComputeSplitsBySize）接口一般用于计算引擎规划并发度等执行计划。ComputeSplitsBySize 可以将全表数据逻辑上划分成接近指定大小的若干分片，并返回这些分片之间的分割点以及分片所在机器的提示。

示例

```
private static void describeTable(SyncClient client) {
    // 以200MB划分分片
    ComputeSplitsBySizeRequest request = new ComputeSplitsBySizeR
equest(TABLE_NAME, 2);
    ComputeSplitsBySizeResponse response = client.computeSplitsBySize(
computeSplitsBySizeRequest);
    System.out.println("ConsumedCapacity=" + response.getConsume
dCapacity().jsonize());
    System.out.println("PrimaryKeySchema=" + response.getPrimary
KeySchema());
    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getRange()或createRangeIterator()
        System.out.println("split.getLowerBound()=" + split.getLowerBo
und().jsonize());
        System.out.println("split.getLowerBound()=" + split.getUpperBo
und().jsonize());
    }
}
```

2.4.6 查询表描述信息

您可以使用查询表描述信息（DescribeTable ）接口查询表的结构信息（TableMeta）、配置信息（TableOptions）和预留读/写吞吐量的情况（ReservedThroughputDetails）。

TableMeta 、 TableOptions 、在建表一节已经有过介绍，ReservedThroughputDetails 除了包含表的预留吞吐量的值外，还包括最近一次上调或者下调的时间。

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.getPrimary
KeyList()) {
        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.
getReservedThroughputDetails();
    System.out.println("表的预留读吞吐量："
            + reservedThroughputDetails.getCapacityUnit().getReadCap
acityUnit());
    System.out.println("表的预留写吞吐量："
            + reservedThroughputDetails.getCapacityUnit().getWriteCa
pacityUnit());
}
```

2.4.7 删除表

您可以使用删除表（DeleteTable）接口删除一个实例下的指定表。

示例

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

2.4.8 创建全局二级索引

您可以使用创建索引表（CreateIndex）接口在一张已经存在的主表上创建全局二级索引表。

说明

参数	定义	说明
IndexName	索引表名字	无
List<String> primaryKey	索引表类型	当前只支持`IT_GLOBAL_INDEX`。
List<String> definedColumns	索引表的属性列	必需为主表的预定义列组合。
IndexType	索引表类型	当前只支持`IT_GLOBAL_INDEX`。
IndexUpdateMode	索引表更新模式	当前只支持`IUM_ASYNC_INDEX`。

示例

```
    private static void createIndex(SyncClient client) {
        IndexMeta indexMeta = new IndexMeta(INDEX2_NAME); // 要创建的索
引表名称。
        indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); // 为索引表添
加主键列。
        indexMeta.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2); // 为索引表添
加主键列。
        indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); // 为索引表添加
属性列。
        CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME
, indexMeta, false);
        client.createIndex(request);
    }

}
```

2.4.9 删除全局二级索引

您可以使用删除索引表（DeleteIndex）接口删除一个主表上的指定全局二级索引表，其它索引表不受影响。

示例

```
    private static void deleteIndex(SyncClient client) {
        DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME
, INDEX_NAME); // 要删除的索引表及主表名
        client.deleteIndex(request);
    }
```

2.4.10 创建多元索引

您可以使用创建多元索引（CreateSearchIndex）接口在一张已经存在的主表上创建一个多元索引。一张表上可以创建多个多元索引，在创建多元索引时可以指定索引名和索引结构。

参数说明

- TableName：需要创建多元索引的表名。
- IndexName：多元索引的名字。
- IndexSchema：包含IndexSetting(索引设置)和FieldSchemas(Index的所有字段的设置)。
  - IndexSetting

    RoutingFields(高级功能，可选配置)：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来计算索引数据的分布位置。路由字段值相同的记录会被索引到相同的数据分区中。

- FieldSchemas

参数	是否必需	说明
FieldName	是	• String类型。 • 要索引的字段名，即列名。 • 可以是主键列，也可以是属性列。
FieldType	是	枚举类型，支持： • LONG(长整型) • DOUBLE(浮点数) • BOOLEAN(布尔) • KEYWORD(字符串) • TEXT(分词字符串) • GEO_POINT(地理点) • NESTED(嵌套类型)
Index	否	是否开启索引。
IndexOptions	否	索引的配置选项。
Analyzer	否	分词器设置。
EnableSortAndAgg	否	• Bool类型。 • 是否开启排序与统计功能。
Store	否	• Bool类型。 • 是否在多元索引中附加存储该字段的值。 • 开启后，可以直接从多元索引中读取该字段的值，而不必反查主表，可用于查询性能优化。
Array	否	• Bool类型。 • 是否是数组格式。 • 如果为true，则表示该列是一个数组，在写入时，也必须按照JSON数组格式写入，比如["a","b","c"]。Nested类型本身就是一个数组，所以无须设置Array。

- IndexSort(可选)：索引的预排序方式，可以指定主键排序或者列排序。如果不设置，默认按照主键进行预排序。

示例

示例1

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(KEYWORD)和整型(LONG)。

```
private static void createSearchIndex(SyncClient client) {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(TABLE_NAME); // 设置表名
    request.setIndexName(INDEX_NAME); // 设置索引名
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD) // 设置字段名、类型
                    .setIndex(true) // 设置开启索引
                    .setEnableSortAndAgg(true), // 设置开启排序和统计功能
            new FieldSchema("Col_Long", FieldType.LONG)
                    .setIndex(true)
                    .setEnableSortAndAgg(true)));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request); // 调用client创建SearchIndex
}
```

示例2

指定IndexSort。

```
private static void createSearchIndexWithIndexSort(SyncClient client)
 {
    CreateSearchIndexRequest request = new CreateSearchIndexRequest();
    request.setTableName(TABLE_NAME);
    request.setIndexName(INDEX_NAME);
    IndexSchema indexSchema = new IndexSchema();
    indexSchema.setFieldSchemas(Arrays.asList(
            new FieldSchema("Col_Keyword", FieldType.KEYWORD).setIndex(true).setEnableSortAndAgg(true),
            new FieldSchema("Col_Long", FieldType.LONG).setIndex(true).setEnableSortAndAgg(true),
            new FieldSchema("Col_Text", FieldType.TEXT).setIndex(true),
            new FieldSchema("Timestamp", FieldType.LONG).setIndex(true).setEnableSortAndAgg(true)));
    // 设置按照Timestamp这一列进行预排序, Timestamp这一列必须建立索引，并打开EnableSortAndAgg
    indexSchema.setIndexSort(new Sort(
            Arrays.<Sort.Sorter>asList(new FieldSort("Timestamp", SortOrder.ASC))));
    request.setIndexSchema(indexSchema);
    client.createSearchIndex(request);
```

```
}
```

2.4.11 查询多元索引描述信息

您可以使用询多元索引描述信息（DescribeSearchIndex）接口来查询一个主表的多元索引的描述信息，包括多元索引的字段信息以及索引配置等。

示例

```
private static DescribeSearchIndexResponse describeSearchIndex(
SyncClient client) {
    DescribeSearchIndexRequest request = new DescribeSearchIndexR
equest();
    request.setTableName(TABLE_NAME); // 设置表名
    request.setIndexName(INDEX_NAME); // 设置索引名
    DescribeSearchIndexResponse response = client.describeSearchIndex(
request);
    System.out.println(response.jsonize()); // 输出response的详细信息
    return response;
}
```

2.4.12 列出多元索引

您可以使用列出多元索引（ListSearchIndex）接口列出一个表下的所有多元索引。

示例

```
private static List<SearchIndexInfo> listSearchIndex(SyncClient client
) {
    ListSearchIndexRequest request = new ListSearchIndexRequest();
    request.setTableName(TABLE_NAME); // 设置表名
    return client.listSearchIndex(request).getIndexInfos(); // 返回表下
所有SearchIndex
}
```

2.4.13 删除多元索引

您可以使用删除多元索引（DeleteSearchIndex）接口删除一个不需要的多元索引。

示例

```
private static void deleteSearchIndex(SyncClient client) {
    DeleteSearchIndexRequest request = new DeleteSearchIndexRequest();
    request.setTableName(TABLE_NAME); // 设置表名
    request.setIndexName(INDEX_NAME); // 设置索引名
    client.deleteSearchIndex(request); // 调用client删除对应的多元索引
```

```
}
```

2.5 单行数据操作

2.5.1 概述

表格存储提供了多个单行操作接口：

插入一行数据

读取一行数据

更新一行数据

删除一行数据

2.5.2 插入一行数据

您可以使用 PutRow 接口插入一行数据。

说明：

如某一行已经存在数据，使用 PutRow 接口插入数据会覆盖该行原数据。

PutRow 写入时支持条件更新（Conditional Update），可设置原行的存在性条件或者原行中某列的列值条件。具体参见条件更新。

示例 1

写入 10 列属性列，每列写入 1 个版本，由服务端指定版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    // 构造主键
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrim
aryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKey
Value.fromString(pkValue));
    PrimaryKey primaryKey = primaryKeyBuilder.build();

    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，由客户端指定版本号（时间戳）。

```
private static void putRow(SyncClient client, String pkValue) {
    // 构造主键
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrim
aryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKey
Value.fromString(pkValue));
    PrimaryKey primaryKey = primaryKeyBuilder.build();

    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。

```
private static void putRow(SyncClient client, String pkValue) {
    // 构造主键
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrim
aryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKey
Value.fromString(pkValue));
    PrimaryKey primaryKey = primaryKeyBuilder.build();

    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME,
primaryKey);

    // 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 的值大于 100 时写入。

```
private static void putRow(SyncClient client, String pkValue) {
    // 构造主键
    PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrim
aryKeyBuilder();
    primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKey
Value.fromString(pkValue));
    PrimaryKey primaryKey = primaryKeyBuilder.build();

    RowPutChange rowPutChange = new RowPutChange(TABLE_NAME,
primaryKey);

    // 期望原行存在, 且Col0的值大于100时写入
    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));
}
```

2.5.3 读取一行数据

单行读 GetRow 接口用于读取一行数据。参数说明如下：

参数	定义	说明
PrimaryKey	要读取的行的主键	必须设置。
ColumnsToGet	要读取的列的集合	若不设置，则读取所有列。
MaxVersions	最多读取的版本数	MaxVersions 与 TimeRange 必须至少设置一个。
TimeRange	要读取的版本号的范围	MaxVersions 与 TimeRange 必须至少设置一个。
Filter	过滤器	在服务端对读取的结果再进行一次过滤。

示例 1

读取一行，设置读取最新版本，设置 ColumnsToGet。

```
        private static void getRow(SyncClient client, String pkValue)
 {
            // 构造主键
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();

            // 读一行
            SingleRowQueryCriteria criteria = new SingleRowQueryCriter
ia(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**

设置过滤器。

```
     private static void getRow(SyncClient client, String pkValue) {
            // 构造主键
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();

            // 读一行
            SingleRowQueryCriteria criteria = new SingleRowQueryCriter
ia(TABLE_NAME, primaryKey);
            // 设置读取最新版本
            criteria.setMaxVersions(1);

            // 设置过滤器, 当Col0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这一列, 也不返回.
            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);
        }
```

2.5.4 更新一行数据

您可以使用 UpdateRow 接口更新一行数据。更新操作包括写入某列、删除某列和删除某列的某一版本。

说明：

如果原行不存在，会新写入一行。

UpdateRow 接口支持条件更新（Conditional Update），可设置原行的存在性条件或者原行中某列的列值条件。具体参见条件更新。

示例 1

更新列、删除一个列的某一版本、删除一个列。

```
        private static void updateRow(SyncClient client, String
pkValue) {
            // 构造主键
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();

            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(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();

            RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);

            // 期望原行存在, 且Col0的值大于100时更新
            Condition condition = new Condition(RowExistenceExpectat
ion.EXPECT_EXIST);
            condition.setColumnCondition(new SingleColumnValueCon
dition("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));
        }
```

2.5.5 删除一行数据

您可以使用DeleteRow 接口删除某一行。

DeleteRow 接口支持条件更新（Conditional Update），可设置原行的存在性条件或者原行中某列的列值条件。具体参见条件更新。

示例 **1**

删除一行。

```
        private static void deleteRow(SyncClient client, String
pkValue) {
```

```
            // 构造主键
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();

            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(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();

            RowDeleteChange rowDeleteChange = new RowDeleteChange(
TABLE_NAME, primaryKey);

            // 期望原行存在, 且Col0的值大于100时删除
            Condition condition = new Condition(RowExistenceExpectat
ion.EXPECT_EXIST);
            condition.setColumnCondition(new SingleColumnValueCon
dition("Col0",
                    SingleColumnValueCondition.CompareOperator.
GREATER_THAN, ColumnValue.fromLong(100)));
            rowDeleteChange.setCondition(condition);

            client.deleteRow(new DeleteRowRequest(rowDeleteChange));
        }
```

2.6 多行数据操作

2.6.1 多行数据操作概述

表格存储提供了多行操作接口，方便您对多个行进行批量操作。

批量读：一次请求读取多行数据。

批量写：在一个请求中进行批量的写入操作。

范围读：在一个请求中读取一个范围内的数据。

迭代读：迭代读取数据。

2.6.2 批量读

您可以使用批量读（BatchGetRow）接口一次请求读取多行数据。

检查返回值

调用 BatchGetRow 接口时，需要检查返回值。批量读时，可能存在部分行读取失败，此时失败行的 Index 及错误信息在返回的 BatchGetRowResponse 中，而不抛出异常。

您可通过 BatchGetRowResponse#getFailedRows 方法获取失败行的信息，通过 BatchGetRowResponse#isAllSucceed 方法判断是否所有行都获取成功。

参数说明

说明：

批量读取的所有行会采用相同的参数条件。例如，当设置 ColumnsToGet=[colA]，则要读取的所有行都只读取 colA 这一列。

参数	定义	说明
PrimaryKey	要读取的行的主键	必须设置。
ColumnsToGet	要读取的列的集合	若不设置，则读取所有列。
MaxVersions	最多读取的版本数	MaxVersions 与 TimeRange 必须至少设置一个。
TimeRange	要读取的版本号的范围	MaxVersions 与 TimeRange 必须至少设置一个。
Filter	过滤器	在服务端对读取的结果再进行一次过滤。

示例

以下示例中展示了如何指定读取行、版本、读取列、过滤器等条件。

```
        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 = PrimaryKey
Builder.createPrimaryKeyBuilder();
                primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME
, PrimaryKeyValue.fromString("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 BatchGetRo
wRequest();
            // 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(multiRowQu
eryCriteria);

            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.getTableName(), 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.这里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. 设定重试策略后, 调用接口处即不需要增加重试代码.
                 */
                BatchGetRowRequest retryRequest = batchGetRowRequest.
createRequestForRetry(batchGetRowResponse.getFailedRows());
            }
        }
```

2.6.3 范围读

您可以使用范围读（GetRange）接口读取一个范围内的数据。范围读接口支持正序读取和反序读取。

说明：

- 表格存储表中的行是按照全部主键列排序的，而不是按照某列主键进行排序。
- 如果读取的范围较大，导致已经扫描的行数或者数据量超过一定限制，系统会停止继续扫描，并返回已经获取的行和下一个主键的位置。您可以根据返回的下一个主键位置，继续发起请求，获取范围内剩余的行。

参数说明

参数	定义	说明
Direction	读取顺序	枚举类型，包括： • FORWARD（正序） • BACKWARD（反序）
InclusiveStartPrimaryKey	范围的起始主键（包含）	若为反序，起始主键需大于结束主键。
ExclusiveEndPrimaryKey	范围的结束主键（不包含）	若为反序，起始主键需大于结束主键。
Limit	本次请求返回的最大行数	无
ColumnsToGet	要读取的列的集合	若不设置，则默认读取所有列。
MaxVersions	最多读取多少个版本	MaxVersions 与 TimeRange 必须至少设置一个。
TimeRange	要读取的版本号的范围	MaxVersions 与 TimeRange 必须至少设置一个。
Filter	过滤器	过滤器在服务端对读取的结果再进行一次过滤。

示例

下面的例子展示了在一个指定范围内进行正序读取，并判断 `NextStartPrimaryKey` 是否为 null。

```
       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(startPkValue));
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
primaryKeyBuilder.build());

            // 设置结束主键
            primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuil
der();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(endPkValue));
```

```
            rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKey
Builder.build());

            rangeRowQueryCriteria.setMaxVersions(1);

            System.out.println("GetRange的结果为:");
            while (true) {
                GetRangeResponse getRangeResponse = client.getRange(
new GetRangeRequest(rangeRowQueryCriteria));
                for (Row row : getRangeResponse.getRows()) {
                    System.out.println(row);
                }

                // 若nextStartPrimaryKey不为null, 则继续读取.
                if (getRangeResponse.getNextStartPrimaryKey() != null)
 {
                    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(
getRangeResponse.getNextStartPrimaryKey());
                } else {
                    break;
                }
            }
        }
```

2.6.4 迭代读

您可以使用迭代读（GetRangeByIterator）接口迭代读取数据。

示例

```
        private static void getRangeByIterator(SyncClient client,
String startPkValue, String endPkValue) {
            RangeIteratorParameter rangeIteratorParameter = new
RangeIteratorParameter(TABLE_NAME);

            // 设置起始主键
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(startPkValue));
            rangeIteratorParameter.setInclusiveStartPrimaryKey(
primaryKeyBuilder.build());

            // 设置结束主键
            primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuil
der();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(endPkValue));
            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);
            }
        }
```

2.6.5 批量写

您可以使用批量写（BatchWriteRow）接口在一个请求中进行批量的写入操作。写入操作包括 PutRow、UpdateRow 和 DeleteRow，并支持一次对多张表进行写入。

构造单个操作的过程与使用 *PutRow* 接口、*UpdateRow* 接口和 *DeleteRow* 接口相同，且支持设置更新条件。

检查返回值

调用 BatchWriteRow 接口时，需要检查返回值。批量读时，可能存在部分行写入失败，此时失败行的 Index 及错误信息在返回的 BatchGetRowResponse 中，而不抛出异常。

您可通过 BatchWriteRowResponse 的 isAllSucceed 判断是否全部成功。若不检查，可能会忽略掉部分操作的失败。

说明：

BatchWriteRow 接口在部分情况下会抛出异常。例如，服务端检查到某些操作出现参数错误，可能会抛出参数错误的异常。在抛出异常的情况下该请求中所有的操作都未执行。

示例

下面批量写示例中包含：2 个 PutRow 操作，1 个 UpdateRow 操作，1 个 DeleteRow 操作。

```
        private static void batchWriteRow(SyncClient client) {
            BatchWriteRowRequest batchWriteRowRequest = new BatchWrite
RowRequest();

            // 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
            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:" + batchWrite
RowRequest.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.这里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. 设定重试策略后, 调用接口处即不需要增加重试代码.
                 */
```

```
                BatchWriteRowRequest retryRequest = batchWrite
RowRequest.createRequestForRetry(response.getFailedRows());
            }
        }
```

2.7 条件更新

条件更新是指只有在满足条件时才对表中的数据进行更改，当不满足条件时更新失败，可用于 PutRow、UpdateRow、DeleteRow 和 BatchWriteRow 中。

判断条件包括行存在性条件和列条件。

- 行存在性条件：分为 IGNORE、 EXPECT_EXIST 和 EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。在对表进行更改操作时，会首先检查行存在性条件。若不满足，则更改失败，对用户抛错。
- 列条件：目前支持两种，SingleColumnValueCondition 和 CompositeColumnValueCondition。是基于某一列或者某些列的列值进行条件判断，与过滤器 Filter 中的条件非常类似。

基于条件更新可以实现乐观锁的功能，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”，更新该行同时使“列 A = 2”。若更新失败，代表有其他客户端已经成功更新了该行。

示例1

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这一列, 条件检查不通过.
        singleColumnValueCondition.setPassIfMissing(false);
        // 只判断最新版本
        singleColumnValueCondition.setLatestVersionsOnly(true);
```

示例 2

构造 CompositeColumnValueCondition。

```
// composite1 条件为 (Col0 == 0) AND (Col1 > 100)
        CompositeColumnValueCondition composite1 = new CompositeC
olumnValueCondition(CompositeColumnValueCondition.LogicOperator.AND);
        SingleColumnValueCondition single1 = new SingleColumnValueCon
dition("Col0",
                SingleColumnValueCondition.CompareOperator.EQUAL,
ColumnValue.fromLong(0));
        SingleColumnValueCondition single2 = new SingleColumnValueCon
dition("Col1",
```

```
                SingleColumnValueCondition.CompareOperator.GREATER_TH
AN, ColumnValue.fromLong(100));
        composite1.addCondition(single1);
        composite1.addCondition(single2);
        // composite2 条件为 ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2
 <= 10)
        CompositeColumnValueCondition composite2 = new CompositeC
olumnValueCondition(CompositeColumnValueCondition.LogicOperator.OR);
        SingleColumnValueCondition single3 = new SingleColumnValueCon
dition("Col2",
                SingleColumnValueCondition.CompareOperator.LESS_EQUAL
, ColumnValue.fromLong(10));
        composite2.addCondition(composite1);
        composite2.addCondition(single3);
```

示例3

通过Condition可以实现乐观锁机制，下面例子演示如何通过Condition实现特定列递增功能。

```
        private static void updateRowWithCondition(SyncClient client,
String pkValue) {
            // 构造主键
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();

            // 读一行
            SingleRowQueryCriteria criteria = new SingleRowQueryCriter
ia(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这一列, 使列值+1
            RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);
            Condition condition = new Condition(RowExistenceExpectat
ion.EXPECT_EXIST);
            ColumnCondition columnCondition = new SingleColumnValueCon
dition("Col0", SingleColumnValueCondition.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());
            }
```

```
        }
```

2.8 过滤器（Filter）

过滤器 Filter 可以在服务器端对读取的结果再进行一次过滤，根据 Filter 中的条件决定返回哪些行或者列。Filter 可以用于 GetRow、BatchGetRow 和 GetRange 接口中。

目前表格存储仅支持 SingleColumnValueFilter 和 CompositeColumnValueFilter，这两个 Filter 都是基于参考列的列值决定某行是否会被过滤掉。前者只判断某个参考列的列值，后者会对多个参考列的列值判断结果进行逻辑组合，决定最终是否过滤。

需要注意的是，Filter 是对读取后的结果再进行一次过滤，所以 SingleColumnValueFilter 或者 CompositeColumnValueFilter 中的参考列必须在读取的结果内。如果用户指定了要读取的列，且其中不包含参考列，那么 Filter 无法获得这些参考列的值。当某个参考列不存在时，SingleColumnValueFilter 的 passIfMissing 参数决定此时是否满足条件，即用户可以选择当参考列不存在时的行为。

示例1

构造 SingleColumnValueFilter。

```
        // 设置过滤器, 当Col0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这一列, 也不返回.
        singleColumnValueFilter.setPassIfMissing(false);
        // 只判断最新版本
        singleColumnValueFilter.setLatestVersionsOnly(true);
```

示例2

构造 CompositeColumnValueFilter。

```
        // composite1 条件为 (Col0 == 0) AND (Col1 > 100)
        CompositeColumnValueFilter composite1 = new CompositeC
olumnValueFilter(CompositeColumnValueFilter.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 条件为 ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2
 <= 10)
        CompositeColumnValueFilter composite2 = new CompositeC
olumnValueFilter(CompositeColumnValueFilter.LogicOperator.OR);
        SingleColumnValueFilter single3 = new SingleColumnValueFilter
("Col2",
                SingleColumnValueFilter.CompareOperator.LESS_EQUAL,
ColumnValue.fromLong(10));
        composite2.addFilter(composite1);
        composite2.addFilter(single3);
```

2.9 主键列自增

主键列自增功能是表格存储新推出的一个功能，JAVA SDK 4.2.0 版本开始支持。

主键列自增功能是指若用户指定某一列主键为自增列，在其写入数据时，表格存储会自动为用户在这一列产生一个新的值，且这个值为同一个分区键下该列的最大值。主要用于系统设计中需要使用主键列自增功能的场景，例如电商网站的商品 ID、大型网站的用户 ID、论坛帖子的 ID、聊天工具的消息 ID 等。

特点

- 表格存储目前支持多个主键，第一个主键为分区键，分区键上不允许使用主键列自增功能。
- 除了分区键外，其余主键中的任意一个都可以被设置为递增列。
- 由于是在分区键基础上递增，所以主键列自增是分区键级别的自增。
- 对于每张表，目前只允许设置一个主键列为自增列。
- 自动生成地自增列为 64 位的有符号长整型。

接口

主键自增列功能主要涉及两类接口，创建表和写数据。

- 创建表

  创建表时，只需将需要自增的主键属性设置为 PrimaryKeyOption.AUTO_INCREMENT。

  相关接口：CreateTable。

```
private static void createTable(SyncClient client) {
  TableMeta tableMeta = new TableMeta(“table_name”);

  // 第一列为分区建
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_1",
PrimaryKeyType.STRING));

  // 第二列为自增列，类型为INTEGER，属性为AUTO_INCREMENT
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema("PK_2",
PrimaryKeyType.INTEGER, PrimaryKeyOption.AUTO_INCREMENT));
```

```
        int timeToLive = -1;  // 永不过期
        int maxVersions = 1;  // 只保存一个版本

        TableOptions tableOptions = new TableOptions(timeToLive,
maxVersions);

        CreateTableRequest request = new CreateTableRequest(
tableMeta, tableOptions);

        client.createTable(request);
    }
```

- 第一个主键是分区键，不能设置为自增列。
- 只有整数类型的列才可以设置为自增列。
- 每张表只能设置一个主键自增列。

- 写数据

  写入数据时，只需将自增列的值为设置为占位符 PrimaryKeyValue.AUTO_INCREMENT。

  相关接口：PutRow/UpdateRow/BatchWriteRow。

```
    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("Hangzhou");


  // 第三列是主键递增列，这个值是TableStore产生的，用户在这里不需要填入真实
值，只需要一个占位符：AUTO_INCREMENT 即可。
        primaryKeyBuilder.addPrimaryKeyColumn("PK_2", PrimaryKey
Value.AUTO_INCREMENT);
        PrimaryKey primaryKey = primaryKeyBuilder.build();

        RowPutChange rowPutChange = new RowPutChange("table_name",
primaryKey);

  // 这里设置返回类型为RT_PK，意思是在返回结果中包含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，消息内容
  rowPutChange.addColumn(new Column("content", ColumnValue.
fromString(content)));

  //写数据到TableStore
        PutRowResponse response = client.putRow(new PutRowRequest(
rowPutChange));

  // 打印出返回的PK列
        Row returnRow = response.getRow();
```

```
  if (returnRow != null) {
   System.out.println("PrimaryKey:" + returnRow.getPrimaryKey().
toString());
  }

  // 打印出消耗的CU
  CapacityUnit  cu = response.getConsumedCapacity().getCapacityUnit
();
  System.out.println("Read CapacityUnit:" + cu.getReadCapacityUnit
());
  System.out.println("Write CapacityUnit:" + cu.getWriteCapacityUnit
());
    }
```

- 写入数据时，主键自增列不需要填值，只需填充占位符即可。
- 如果用户想获取向表格存储写入数据后自动生成的主键值，可以将 ReturnType 设置为 RT_PK，这样就能在写入成功后返回主键值。

2.10 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 GetShardIt
eratorRequest(streamId, shardId);
 GetShardIteratorResponse shardIterator = client.getShardIterator(
getShardIteratorRequest);
 }
```

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 GetStreamR
ecordRequest(shardIterator);
  streamRecordRequest.setLimit(100);
  GetStreamRecordResponse streamRecordResponse = client.getStreamR
ecord(streamRecordRequest);
  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());
```

```
}
```

2.11 错误处理

方式

TableStore 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，比如 BatchGetRow 和 BatchWriteRow 不仅需要判断是否有异常，还需要检查每个 row 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

TableStore Java SDK 中有 ClientException 和 OTSException 两种异常，他们都最终继承自 RuntimeException。

- ClientException：指 SDK内部出现的异常，比如参数设置不对等。
- OTSException：指服务器端错误，它来自于对服务器错误信息的解析。OTSException 包含以下几个成员：
  - getHttpStatus()：HTTP 返回码，比如 200、404等。
  - getErrorCode()：表格存储返回的错误类型字符串。
  - getRequestId()：用于唯一标识该次请求的 UUID。当您无法解决问题时，可以记录这个 RequestId 并提交工单。

2.12 多元索引查询操作

2.12.1 精确查询

精确查询（TermQuery）接口采用完整精确匹配的方式查询表中的数据，但是对于分词字符串类型，只要分词后有词条可以精确匹配即可。

比如某个分词字符串类型的字段，值为“tablestore is cool”，假设分词后为“tablestore”、“is”、“cool”三个词条，则查询“tablestore”、“is”、“cool”时都满足查询条件。

示例

```
/**
 * 查询表中Col_Keyword这一列精确匹配"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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    searchRequest.setColumnsToGet(columnsToGet);

    SearchResponse resp = client.search(searchRequest);
    System.out.println("Row: " + resp.getRows());
    // 可检查NextToken是否为空，若不为空，可通过NextToken继续读取。
}
```

2.12.2 匹配查询

MatchAllQuery

MatchAllQuery用于匹配所有行，常用于查询表中数据总行数，或者查看表中任意几条数据。

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();
    searchQuery.setQuery(new MatchAllQuery()); // 设置查询类型为
MatchAllQuery
    /**
     * MatchAllQuery结果中的TotalCount可以表示表中数据的总行数，
     * 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。
     */
    searchQuery.setLimit(0);
    searchQuery.setGetTotalCount(true);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
    /**
     * 判断返回的结果是否是完整的，当isAllSuccess为false时，代表可能有部分节点
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()); // 总行
数
    System.out.println(resp.getRequestId());
}
```

MatchQuery

MatchQuery采用近似匹配的方式查询表中的数据。比如查询的值为"this is", 可以匹配到“... , this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。

示例

```
/**
 * 查询表中Col_Keyword这一列的值能够匹配"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"); // 设置要匹配的字段
    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);
    searchQuery.setSort(new Sort(Arrays.asList(new ScoreSort
())));  // 设置按照匹配得分排序
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount());
    System.out.println("Row: " + resp.getRows()); // 不设置columnsToGet
，默认只返回主键

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    searchRequest.setColumnsToGet(columnsToGet);

    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());
}
```

MatchPhraseQuery

MatchQuery采用近似匹配的方式查询表中的数据。比如查询的值为"this is", 可以匹配到“... , this is tablestore”、“is this tablestore”、“tablestore is cool”、“this"、“is”等。

示例

```
/**
 * 查询表中Col_Text这一列的值能够匹配"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"); // 设置要匹配的字段
    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);
    searchQuery.setSort(new Sort(Arrays.asList(new ScoreSort())));
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
    System.out.println("TotalCount: " + resp.getTotalCount());
    System.out.println("Row: " + resp.getRows()); // 默认只返回主键

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    searchRequest.setColumnsToGet(columnsToGet);

    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());
}
```

2.12.3 前缀查询

PrefixQuery根据前缀条件查询表中的数据。对于分词字符串类型(TEXT)，只要分词后的词条中有词条满足前缀条件即可。

示例

```
/**
 * 查询表中Col_Keyword这一列前缀为"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
    prefixQuery.setFieldName("Col_Keyword");
    prefixQuery.setPrefix("hangzhou");
    searchQuery.setQuery(prefixQuery);
```

```
    searchQuery.setGetTotalCount(true);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
}
```

2.12.4 范围查询

RangeQuery根据范围条件查询表中的数据。对于分词字符串类型(TEXT)，只要分词后的词条中有词条满足范围条件即可。

示例

```
/**
 * 查询表中Col_Long这一列大于3的数据，结果按照Col_Long这一列的值逆序排序。
 * @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");  // 设置针对哪个字段
    rangeQuery.greaterThan(ColumnValue.fromLong(3));  // 设置该字段的范围
条件，大于3
    searchQuery.setGetTotalCount(true);
    searchQuery.setQuery(rangeQuery);

    // 设置按照Col_Long这一列逆序排序
    FieldSort fieldSort = new FieldSort("Col_Long");
    fieldSort.setOrder(SortOrder.DESC);
    searchQuery.setSort(new Sort(Arrays.asList((Sort.Sorter)fieldSort
)));

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    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());
```

```
}
```

2.12.5 通配符查询

使用WildcardQuery进行通配符查询。

要匹配的值可以是一个带有通配符的字符串。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。比如查询“table*e”，可以匹配到“tablestore”。目前不支持以星号开头。

示例

```
/**
 * 使用通配符查询，查询表中Col_Keyword这一列的值匹配"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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
}
```

2.12.6 地理位置查询

多元索引支持三种地理位置查询方式，分别

是GeoBoundingBoxQuery、GeoDistanceQuery和GeoPolygonQuery。

GeoBoundingBoxQuery

使用GeoBoundingBoxQuery进行地理边界框查询。

根据一个矩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矩形范围内时，满足查询条件。

```
/**
 * Col_GeoPoint是GeoPoint类型，查询表中Col_GeoPoint这一列的值在左上角为"10,0", 右下角为"0,10"的矩形范围内的数据。
 * @param client
 */
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"); // 设置比较哪个字段的值
    geoBoundingBoxQuery.setTopLeft("10,0"); // 设置矩形左上角
    geoBoundingBoxQuery.setBottomRight("0,10"); // 设置矩形右下角
    searchQuery.setQuery(geoBoundingBoxQuery);
    searchQuery.setGetTotalCount(true);

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME, INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.ColumnsToGet();
    columnsToGet.setColumns(Arrays.asList("Col_GeoPoint"));  //设置返回Col_GeoPoint这一列
    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());
}
```

GeoDistanceQuery

使用GeoDistanceQuery进行地理距离查询。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。

```
/**
 * 查询表中Col_GeoPoint这一列的值距离中心点不超过一定距离的数据。
 * @param client
 */
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); // 设置到中心点的距离条件，不超过10000米
    searchQuery.setQuery(geoDistanceQuery);
    searchQuery.setGetTotalCount(true);
```

```
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setColumns(Arrays.asList("Col_GeoPoint"));  //设置返回
Col_GeoPoint这一列
    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());
}
```

GeoPolygonQuery

使用GeoPolygonQuery进行地理多边形查询。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内时，满足查询条件。

```
/**
 * 查询表中Col_GeoPoint这一列的值在一个给定多边形范围内的数据。
 * @param client
 */
public static void geoPolygonQuery(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    GeoPolygonQuery geoPolygonQuery = new GeoPolygonQuery();  // 设置查
询类型为GeoPolygonQuery
    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);

    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setColumns(Arrays.asList("Col_GeoPoint"));  //设置返回
Col_GeoPoint这一列
    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());
```

```
}
```

2.12.7 多条件组合查询

使用BoolQuery进行多条件的组合查询。BoolQuery可以包含一个或者多个子查询条件，根据子查询条件是否满足来判断一行数据是否满足查询条件。

可以设置多种组合方式，比如设置多个子查询条件为mustQueries，则要求这些子查询条件必须都满足。设置多个子查询条件为mustNotQueries时，要求这些子查询条件都不能满足。

BoolQuery有以下几个参数：

```
/**
 * 文档必须完全匹配所有的子query
 */
List<Query> mustQueries;
/**
 * 文档必须不能匹配任何子query
 */
List<Query> mustNotQueries;
/**
 * 文档必须完全匹配所有的子filter, filter类似于query，区别是不会进行算分
 */
List<Query> filterQueries;
/**
 * 文档应该至少匹配minimumShouldMatch个should条件，匹配多的得分会高
 */
List<Query> shouldQueries;
/**
 * 定义了至少满足几个should子句，默认是1.
 */
Integer minimumShouldMatch;
```

BoolQuery的子Query可以是任意Query类型，包括BoolQuery本身，因此通过BoolQuery可以实现非常复杂的组合查询类型。

示例

```
/**
 * 通过BoolQuery进行复合条件查询。
 * @param client
 */
public static void boolQuery(SyncClient client) {
    /**
     * 查询条件一：RangeQuery，Col_Long这一列的值要大于3
     */
    RangeQuery rangeQuery = new RangeQuery();
    rangeQuery.setFieldName("Col_Long");
    rangeQuery.greaterThan(ColumnValue.fromLong(3));

    /**
```

```
     * 查询条件二：MatchQuery，Col_Keyword这一列的值要匹配"hangzhou"
     */
    MatchQuery matchQuery = new MatchQuery(); // 设置查询类型为
MatchQuery
    matchQuery.setFieldName("Col_Keyword"); // 设置要匹配的字段
    matchQuery.setText("hangzhou"); // 设置要匹配的值

    SearchQuery searchQuery = new SearchQuery();
    {
        /**
         * 构造一个BoolQuery，设置查询条件是必须同时满足"条件一"和"条件二"
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setMustQueries(Arrays.asList(rangeQuery, matchQuery
));
        searchQuery.setQuery(boolQuery);
        searchQuery.setGetTotalCount(true);
        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
        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());
    }

    {
        /**
         * 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
条件
         */
        BoolQuery boolQuery = new BoolQuery();
        boolQuery.setShouldQueries(Arrays.asList(rangeQuery,
matchQuery));
        boolQuery.setMinimumShouldMatch(1); // 设置最少满足一个条件
        searchQuery.setQuery(boolQuery);
        searchQuery.setGetTotalCount(true);
        SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
        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());
    }
}
```

2.12.8 嵌套类型查询

NestedQuery用于嵌套类型的查询。嵌套类型不能直接查询，需要通过NestedQuery来包装一下，NestedQuery中需要指定嵌套类型的字段路径(path)以及一个子query(可以是任意query)。

示例

```
/**
 * 有一类型为NESTED的列，子文档包含nested_1和nested_2两列，现在查询col_nested
.nested_1为"tablestore"的数据。
 * @param client
```

```
 */
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NESTED字段路径
    TermQuery termQuery = new TermQuery(); // 构造NestedQuery的子查询
    termQuery.setFieldName("col_nested.nested_1"); // 设置字段名，注意带
有Nested列的前缀
    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);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);

    SearchRequest.ColumnsToGet columnsToGet = new SearchRequest.
ColumnsToGet();
    columnsToGet.setReturnAll(true); // 设置返回所有列
    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());
}
```

2.12.9 排序和翻页

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。

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。

```
SearchQuery searchQuery = new SearchQuery();
```

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

PrimaryKeySort

按照主键排序。

```
正序：
SearchQuery searchQuery = new SearchQuery();
searchQuery.setSort(new Sort(Arrays.asList(new PrimaryKeySort())));

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

FieldSort

按照某列进行排序。

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

先按照某列排序，再按照另一列排序。

```
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这一列为GEOPOINT类型，按照这一列的值距离"0,0"点的距离排序。
Sort.Sorter sorter = new GeoDistanceSort("geo", Arrays.asList("0, 0
"));
searchQuery.setSort(new Sort(Arrays.asList(sorter)));
```

翻页方式

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 <= 200。

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

使用**token**进行翻页

如果符合查询条件的数据没有读完，服务端会返回NextToken，用户可以使用NextToken继续读取后面的数据。使用Token后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)，因此设置了Token不能再设置Sort。\u0008另外，使用Token后不能设置Offset，只能一次一次往后读，即无法跳页。

```
/**
 * 使用Token进行翻页，这个例子会把所有数据读出，放到一个List中。
 * @param client
 */
private static void readMoreRowsWithToken(SyncClient client) {
    SearchQuery searchQuery = new SearchQuery();
    searchQuery.setQuery(new MatchAllQuery());
    searchQuery.setGetTotalCount(true);
    SearchRequest searchRequest = new SearchRequest(TABLE_NAME,
INDEX_NAME, searchQuery);
    SearchResponse resp = client.search(searchRequest);
    if (!resp.isAllSuccess()) {
        throw new RuntimeException("not all success");
    }
    List<Row> rows = resp.getRows();
    while (resp.getNextToken()!=null) { //读到NextToken为null为止，即读出
全部数据
        //把Token设置到下一次请求中
        searchRequest.getSearchQuery().setToken(resp.getNextToken());
        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());
}
```

3 NodeJS SDK

3.1 前言

本文档主要介绍 TableStore NodeJS SDK 的安装和使用，适用 4.x 版本。请确保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。
- 如果您还没有创建 AccessKeyId 和 AccessKeySecret，请到阿里云 *Access Key* 的管理控制台创建 Access Key。

SDK下载

- *SDK 包*
- *GitHub*

版本

当前最新版本：4.0.0

3.2 安装

环境准备

适用于 NodeJS 4.0 及以上版本。

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 *cnpm*。

示例程序

NodeJS SDK 提供丰富的示例程序，方便用户参考或直接使用。您可以通过以下两种方式获取示例程序：

- 下载 Table Store NodeJS SDK 开发包，解压后 examples 为示例程序。
- 访问 Table Store NodeJS SDK 的 *GitHub* 项目。

3.3 初始化

TableStore.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，目前支持下列形式。

Endpoint类型	解释
区域地址	使用表格存储实例（Instance）所在的区域地址，如 `https://instance.cn-hangzhou.ots.aliyuncs.com`

表格存储的区域地址

使用表格存储实例所在的区域地址，可以通过以下方式查询 Endpoint：

1. 登录表格存储控制台。
2. 进入实例详情页，实例访问地址即是该实例的 Endpoint。

配置密钥

要接入阿里云表格存储服务，您需要拥有一个有效的 Access Key（包括AccessKeyId和AccessKeySecret）用来进行签名认证。可以通过如下步骤获得：

1. 在阿里云官网注册阿里云帐号。
2. 登录*AccessKey*管理控制台创建 AccessKeyId 和 AccessKeySecret。

在获取到 AccessKeyId 和 AccessKeySecret 之后，您可以按照下面步骤进行初始化对接。

使用表格存储的 Endpoint 新建 Client。

示例：

```
  var client = new TableStore.Client({
accessKeyId: '<your access key id>',
secretAccessKey: '<your access key secret>',
endpoint: '<your endpoint>',
instancename: '<your instance name>',
maxRetries:20,//默认20次重试，可以省略这个参数。
```

```
});
```

3.4 Long类型

表格存储提供了五种数据类型，与NodeJS SDK的数据类型对应关系如下：

数据类型	NodeJS SDK	描述
String	string	JavaScript语言中的基本数据类型
Integer	int64	NodeJs 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	NodeJS的Buffer对象

int64类型

表格存储的Integer类型是一个64位的有符号整型，这个数据类型在JavaScript中没有相应的数据类型可以对应，所以在NodeJS中也需要一个能表示64位有符号整型的数据类型。

可以做如下转换：

```
var numberA = TableStore.Long.fromNumber(1000);
var numberB = TableStroe.Long.fromString('2000');

var num = numberA.toNumber();
    num = numberA.toString();

var str = numberB.toNumber();
    str = numberB.toString();
```

3.5 表操作

表格存储的 SDK 提供了 CreateTable、ListTable、DeleteTable、UpdateTable 和 DescribeTable 等表级别的操作接口。

创建表（**CreateTable**）

根据给定的表的结构信息创建相应的表。

创建表格存储的表时必须指定表的主键。主键包含 1~4 个主键列，每一个主键列都有名字和类型。

接口

```
  /**
   * 根据给定的表结构信息创建相应的表。
   */
  createTable(params, callback)
```

 说明：

表格存储的表在被创建之后需要几秒钟进行加载，创建成功后需要等待几秒钟后再做其他操作。

示例

创建一个有 2 个主键列，预留读/写吞吐量为 (0,0) 的表。

```
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, 单位秒, -1代表永不过期. 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即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);
});
```

 说明：

详细代码可在 *createTable@GitHub* 获取。

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

接口

```
  /**
   * 获取当前实例下已创建的所有表的表名。
   */
  listTable(params, callback)
```

示例

获取实例下的所有表名。

```
var client = require('./client');

client.listTable({}, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

 说明：

详细代码可在 *listTable@GitHub* 获取。

更新表（UpdateTable）

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);
});
```

说明：

详细代码可在 *updateTable@GitHub* 获取。

查询表描述信息（DescribeTable）

查询指定表的结构信息和预留读/写吞吐量的设置信息。

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)
```

示例

```
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);
});
```

说明：

详细代码可在 *describeTable@GitHub* 获取。

删除表（DeleteTable）

删除本实例下指定的表。

接口

```
  /**
   * 删除本实例下指定的表。
   */
  deleteTable(params, callback)
```

示例

删除表。

```
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);
});
```

 说明：

详细代码可在 *deleteTable@GitHub* 获取。

3.6 单行数据操作

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

插入一行数据（PutRow）

插入数据到指定的行。

接口

```
  /**
   * 插入数据到指定的行，如果该行不存在，则新增一行；若该行存在，则覆盖原有行。
   */
  putRow(params, callback)
```

示例

```
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.RowExistenceExpectat
ion.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);
});
```

- RowExistenceExpectation.IGNORE 表示不管此行是否已经存在，都会插入新数据，如果之前有会被覆盖。
- RowExistenceExpectation.EXPECT_EXIST 表示只有此行存在时，才会插入新数据，此时，原有数据也会被覆盖。
- RowExistenceExpectation.EXPECT_NOT_EXIST 表示只有此行不存在时，才会插入数据，否则不执行。

 说明：

详细代码可在 *PutRow@GitHub* 获取。

读取一行数据（GetRow）

根据给定的主键读取单行数据。

接口

```
  /**
   * 根据给定的主键读取单行数据。
   */
  getRow(params, callback)
```

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
};
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', TableStore.ComparatorType.EQUAL));

params.columnFilter = condition;

client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

说明：

详细代码可在 *GetRow@GitHub* 获取。

更新一行数据（**UpdateRow**）

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这一行中新增、修改或者删除指定列的值。

接口

```
  /**
   * 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这
一行中新增、修改或者删除指定列的值。
   */
  updateRow(params, callback)
```

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.RowExisten
ceExpectation.IGNORE, null),
    primaryKey: [{ 'gid': Long.fromNumber(9) }, { 'uid': Long.
fromNumber(90) }],
```

```
    updateOfAttributeColumns: [
        { 'PUT': [{ 'col4': Long.fromNumber(4) }, { 'col5': '5' }, { '
col6': Long.fromNumber(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);
    });
```

说明：

详细代码可在 *UpdateRow@GitHub* 获取。

删除一行数据（**DeleteRow**）

接口

```
  /**
   * 删除一行数据。
   */
  deleteRow(params, callback)
```

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.RowExisten
ceExpectation.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);
});
```

说明：

详细代码可在 *DeleteRow@GitHub* 获取。

3.7 多行数据操作

表格存储的 SDK 提供了 BatchGetRow、BatchWriteRow、GetRange 和 GetByIterator 等多行操作的接口。

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

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。

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。

接口

```
  /**
   * 批量读取一个或多个表中的若干行数据。
   */
  batchGetRow(params, callback)
```

示例

批量一次读多个表、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);
```

说明：

- 批量读也支持通过条件语句过滤。
- 详细代码可在 *BatchGetRow@GitHub* 获取。

批量写（BatchWriteRow）

批量插入、修改或删除一个或多个表中的若干行数据。

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。

接口

```
/**
 * 批量修改行
 */
batchWriteRow(params, callback)
```

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: 'PUT',
            condition: new TableStore.Condition(TableStore.RowExisten
ceExpectation.IGNORE, null),
            primaryKey: [{ 'gid': Long.fromNumber(8) }, { 'uid': Long.
fromNumber(80) }],
            attributeColumns: [{ 'attrCol1': 'test1' }, { 'attrCol2':
 'test2' }],
            returnContent: { returnType: TableStore.ReturnType.
Primarykey }
        }],
    }],
};

client.batchWriteRow(params, function (err, data) {

    if (err) {
        console.log('error:', err);
        return;
    }

    console.log('success:', data);
```

```
});
```

说明：

- 批量写也支持条件语句。
- 详细代码可在 *BatchWriteRow@GitHub* 获取。

范围读（GetRange）

读取指定主键范围内的数据。

接口

```
  /**
   * 读取指定主键范围内的数据。
   */
  getRange(params, callback)
```

示例

按照范围读取。

```
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，说明需要继续读取
  if (data.next_start_primary_key) {

  }

  console.log('success:', data);
});
```

说明：

- 按范围读也支持通过条件语句过滤。
- 按范围读需要注意数据可能会分页。
- 详细代码可在 *GetRange@GitHub* 获取。

3.8 错误处理

方式

Table Store NodeJS 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。

注意：

批量相关接口比如 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

Table Store NodeJS SDK 中所有的错误都经过了统一的处理，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Table Store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，可以记录这个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，请参考 tablestore/lib/retry.js。

4 Go SDK

4.1 前言

本文档主要介绍 TableStore Go SDK 的安装和使用，适用 4.0.0 版本。并且假设您已经开通了阿里云的表格存储服务，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。
- 如果您还没有创建 AccessKeyId 和 AccessKeySecret，请到阿里云 *Access Key* 的管理控制台创建 Access Key。

SDK 下载

- *SDK 包*（包含package、源代码和示例）
- 更全面的安装方式请参见安装。

版本

当前最新版本：4.1.0

4.2 安装

环境准备

适用于 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、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，目前支持下列形式。

Endpoint类型	解释
区域地址	使用表格存储实例（Instance）所在的区域地址，如 `https://instance.cn-hangzhou.ots.aliyuncs.com`

表格存储的区域地址

使用表格存储实例所在的区域地址，可以通过以下方式查询 Endpoint：

1. 登录表格存储控制台。
2. 进入实例详情页，实例访问地址即是该实例的 Endpoint。

配置密钥

要接入阿里云表格存储服务，您需要拥有一个有效的 Access Key（包括AccessKeyId和AccessKeySecret）用来进行签名认证。可以通过如下步骤获得：

1. 在阿里云官网注册阿里云帐号。
2. 登录*AccessKey*管理控制台创建 AccessKeyId 和 AccessKeySecret。

在获取到 AccessKeyId 和 AccessKeySecret 之后，您可以按照下面步骤进行初始化对接。

使用表格存储的 Endpoint 新建 Client。

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://'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 = NewClient("your_instance_endpoint",  "your_instance_name", "
your_user_id", "your_user_key")
```

4.4 表操作

表格存储的 SDK 提供了 CreateTable、ListTable、DeleteTable、UpdateTable 和 DescribeTable 等表级别的操作接口。

创建表（**CreateTable**）

根据给定的表的结构信息创建相应的表。

创建表格存储的表时必须指定表的主键。主键包含 1~4 个主键列，每一个主键列都有名字和类型。

接口

```
    // 说明：根据表信息创建表。
    // request 是CreateTableRequest类的实例，它包含tablemeta 和
TableOption 以及 ReservedThroughput
   // 请参考TableMeta类的文档。当创建了一个表之后，通常要等待1分钟时间使
partition load
   // 完成，才能进行各种操作。
   // 返回：CreateTableResponse。
  CreateTable(request *CreateTableRequest) (*CreateTableResponse,
error)
```

说明：

表格存储的表在被创建之后需要几秒钟进行加载，创建成功后需要等待几秒钟后再做其他操作。

示例

创建一个含有 2 个主键列，预留读/写吞吐量为 (0,0) 的表。

```
        //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，这个同时也是分片列
        //第二个PK列为整数，名称是pk1

        tableMeta := new(tablestore.TableMeta)
        tableMeta.TableName = tableName
        tableMeta.AddPrimaryKeyColumn("pk0", tablestore.PrimaryKey
Type_INTEGER)
        tableMeta.AddPrimaryKeyColumn("pk1", tablestore.PrimaryKey
Type_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")
        }
```

说明：

详细代码可在 *createTable@GitHub* 获取。

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

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

获取实例下的所有表名。

```
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)
 }
}
```

说明：

详细代码可在 *listTable@GitHub* 获取。

更新表（UpdateTable）

更新指定表的预留读吞吐量或预留写吞吐量的设置。

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")
 }
```

说明：

详细代码可在 *updateTable@GitHub* 获取。

查询表描述信息（**DescribeTable**）

查询指定表的结构信息和预留读/写吞吐量的设置信息。

接口

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

示例

```
 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)
 }
```

说明：

详细代码可在 *describeTable@GitHub* 获取。

删除表（DeleteTable）

删除本实例下指定的表。

接口

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

示例

删除表。

```
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")
}
```

说明：

详细代码可在 *deleteTable@GitHub* 获取。

4.5 单行数据操作

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

插入一行数据（PutRow）

插入数据到指定的行。

接口

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

示例

```
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")
}
```

- RowExistenceExpectation.IGNORE 表示不管此行是否已经存在，都会插入新数据，如果之前有会被覆盖。
- RowExistenceExpectation.EXPECT_EXIST 表示只有此行存在时，才会插入新数据，此时，原有数据也会被覆盖。
- RowExistenceExpectation.EXPECT_NOT_EXIST 表示只有此行不存在时，才会插入数据，否则不执行。

 说明：

详细代码可在 *PutRow@GitHub* 获取。

读取一行数据（GetRow）

根据给定的主键读取单行数据。

接口

```
//返回表（Table）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)
```

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.Columns[0].Value,)
 }
```

说明：

详细代码可在 *GetRow@GitHub* 获取。

更新一行数据（UpdateRow）

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这一行中新增、修改或者删除指定列的值。

接口

```
 // 更新表中的一行
 // @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)
```

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.RowExistenceExpectat
ion_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")
 }
```

说明：

详细代码可在 *UpdateRow@GitHub* 获取。

删除一行数据（DeleteRow）

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)
```

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.RowExisten
ceExpectation_EXPECT_EXIST)
clCondition1 := tablestore.NewSingleColumnCondition("col2",
tablestore.CT_EQUAL, int64(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")
}
```

说明：

详细代码可在 *DeleteRow@GitHub* 获取。

4.6 多行数据操作

表格存储的 SDK 提供了 BatchGetRow、BatchWriteRow、GetRange 和 GetByIterator 等多行操作的接口。

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

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。

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。

接口

```
 //返回表（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
)
```

示例

批量一次读 10 行。

```
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.MultiRowQu
eryCriteria, mqCriteria)
batchGetResponse, err := client.BatchGetRow(batchGetReq)

if err != nil {
 fmt.Println("batachget failed with error:", err)
} else {
 fmt.Println("batchget finished")
}
```

说明：

- 批量读也支持通过条件语句过滤。
- 详细代码可在 *BatchGetRow@GitHub* 获取。

批量写（**BatchWriteRow**）

批量插入、修改或删除一个或多个表中的若干行数据。

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。

接口

```
 // 对多张表（Table）中的多行数据进行增加、删除或者更改操作。
 //
```

```
 // @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)
```

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")
 }
```

说明：

- 批量写也支持条件语句。
- 详细代码可在 *BatchWriteRow@GitHub* 获取。

范围读（GetRange）

读取指定主键范围内的数据。

接口

```
 // 从表（Table）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)
```

示例

按照范围读取。

```
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)
  }
  if (len(getRangeResp.Rows) > 0) {
   for _, row := range getRangeResp.Rows {
    fmt.Println("range get row with key", row.PrimaryKey.PrimaryKey
s[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 = getRangeRe
sp.NextStartPrimaryKey
    getRangeResp, err = client.GetRange(getRangeRequest)
   }
  } else {
   break
  }

  fmt.Println("continue to query rows")
 }
 fmt.Println("putrow finished")
```

 说明：

- 按范围读也支持通过条件语句过滤。
- 按范围读需要注意数据可能会分页。
- 详细代码可在 *GetRange@GitHub* 获取。

5 Python SDK

5.1 简介

本文档主要介绍 Table Store Python SDK 的安装和使用，适用 4.x.x 版本。请确保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。
- 如果您还没有创建 AccessKeyId 和 AccessKeySecret，请到阿里云 *Access Key* 的管理控制台创建 Access Key。

SDK 下载

- *SDK包*
- *GitHub*

关于版本迭代详情，请参见这里。

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，原因是 2.0 系列版本中支持主键乱序，而 4.0 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 接口新增参数 TableOptions。
  - 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 接口，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 中包含这一行的 PK 值。

版本

当前最新版本：4.3.7

5.2 安装

环境准备

适用于 Python 2 和 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，请先安装 *git*，然后执行如下命令：

  ```
  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。

  1. 下载 *SDK包*。
  2. 解压开发包后执行如下命令：

     ```
     sudo python setup.py install
     ```

验证SDK

首先命令行输入python并回车，在Python环境下检查SDK的版本：

```
>>> import tablestore
>>> tablestore.__version__
```

```
'4.3.7'
```

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、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，目前支持下列形式。

Endpoint 类型	解释
区域地址	使用表格存储实例（Instance）所在的区域地址，如 `https://instance.cn-hangzhou.ots.aliyuncs.com`

表格存储的区域地址

使用表格存储实例所在的区域地址，可以通过以下方式查询 Endpoint：

1. 登录表格存储控制台。
2. 进入实例详情页，实例访问地址即是该实例的 Endpoint。

配置密钥

要接入表格存储服务，您需要拥有一个有效的 Access Key（包括 AccessKeyId 和 AccessKeySecret）用来进行签名认证。可以通过如下步骤获得：

1. 在阿里云官网注册阿里云帐号。
2. 登录*AccessKey*管理控制台创建 AccessKeyId 和 AccessKeySecret。

在获取到 AccessKeyId 和 AccessKeySecret 之后，您可以使用表格存储的 Endpoint 进行初始化对接，如下所示：

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,默认是 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。默认为 50。
        ``logger_name``用来在请求中打 DEBUG 日志,或者在出错时打 ERROR 日志。
        ``retry_policy``定义了重试策略,默认的重试策略为 DefaultRetryPolicy
。你可以继承 RetryPolicy 来实现自己的重试策略,请参考 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_name', 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 表操作

表格存储的SDK提供了 CreateTable、ListTable、DeleteTable、UpdateTable 和 DescribeTable 等表级别的操作接口。

创建表（CreateTable）

根据给定的表的结构信息创建相应的表。

创建表格存储的表时必须指定表的主键。主键包含 1~4 个主键列，每一个主键列都有名字和类型。

接口

```
        说明：根据表信息创建表。

        ``table_meta``是``tablestore.metadata.TableMeta``类的实例，它包含表名和PrimaryKey的schema，
        请参考``TableMeta``类的文档。当创建了一个表之后，通常要等待1分钟时间使partition load完成，才能进行各种操作。
        ``table_options``是``tablestore.metadata.TableOptions``类的示例，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。

        返回：无。

  def create_table(self, table_meta, reserved_throughput):
```

 说明：

表格存储的表在被创建之后需要几秒钟进行加载，创建成功后需要等待几秒钟后再做其他操作。

示例

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，这个同时也是分片列
 # 第二个PK列为整数，名称是pk1。其他可选的类型包括STRING，BINARY，这里使用INTEGER。
 schema_of_primary_key = [('pk0', 'INTEGER'), ('pk1', 'INTEGER')]

 # 通过表名和主键列的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标准时间相差不能超过1天。
 table_options = TableOptions(31536000, 3, 86400)
```

```
 # 设定预留读吞吐量为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，则说明成功，否则失败
 try:
  ots_client.create_table(table_meta, table_options, reserved_t
hroughput)
  print "create table succeeded"
 # 处理异常
 except Exception:
  print "create table failed."
```

详细代码请参见*CreateTable@GitHub*。

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

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，如：('MyTable1', '
MyTable2')。
  """
  def list_table(self):
```

示例

获取实例下的所有表名。

```
 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*。

更新表（UpdateTable）

更新指定表的预留读吞吐量或预留写吞吐量设置。

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，max_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_t
hroughput):
```

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
)

            # 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，则说莫执行失败，打印出错误信息
            print "update table failed"
```

详细代码请参见*UpdateTable@GitHub*。

查询表描述信息（DescribeTable）

查询指定表的结构信息和预留读/写吞吐量设置信息。

接口

```
        """
        说明：获取表的描述信息。
        ``table_name``是对应的表名。
        返回：表的描述信息。
        ``describe_table_response``表示表的描述信息，是ots2.metadata.
DescribeTableResponse类的实例。
  """
  def describe_table(self, table_name):
```

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，则认为执行成功，下面打印表格信息
  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_t
hroughput_details.capacity_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_details.last_increase_time)
  print ('Last decrease throughput time: %s' % describe_response.
reserved_throughput_details.last_decrease_time)
  print ('table options\'s time to live: %s' % describe_response.
table_options.time_to_live)
  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:
  # 如果抛出异常则执行失败，处理异常
  print "describe table failed."
```

详细代码请参见*DescribeTable@GitHub*。

删除表（**DeleteTable**）

删除本实例下指定的表。

接口

```
        """
        说明：根据表名删除表。
        ``table_name``是对应的表名。
        返回：无。
  """
  def delete_table(self, table_name):
```

示例

删除表。

```
try:
 # 调用接口删除表SampleTable
    ots_client.delete_table('SampleTable')
 # 如果没有抛出异常，则执行成功
 print "delete table succeeded"
 # 如果抛出异常，则执行失败，处理异常
except Exception:
```

```
    print "delete table failed"
```

详细代码请参见*DeleteTable@GitHub*。

5.5 单行数据操作

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

插入一行数据（PutRow）

插入数据到指定的行。

接口

```
        """
        说明：写入一行数据。返回本次操作消耗的CapacityUnit。
        ``table_name``是对应的表名。
        ``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。目前只支持对行的存在性进行检查，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)
```

示例

插入一行数据。

```
 ## 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)]

 ## 属性列包括四个：
 ##                          第一个属性列的名字是name，值是字符串John，版本没有
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15100000000
，版本没有指定，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age，值是29.7，版本号为
1498184687。
 attribute_columns = [('name','John'), ('mobile',15100000000),('
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)

    # 行条件检查为：期望行不存在。如果行存在就会报错：Condition Update Failed
。
    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)

  # 打印出此次请求消耗的写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_mess
age:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.
get_error_message(), e.get_request_id())
```

 说明：

- RowExistenceExpectation.IGNORE 表示不管此行是否已经存在，都会插入新数据，如果之前有会被覆盖。
- RowExistenceExpectation.EXPECT_EXIST 表示只有此行存在时，才会插入新数据，此时，原有数据也会被覆盖。
- RowExistenceExpectation.EXPECT_NOT_EXIST 表示只有此行不存在时，才会插入数据，否则不执行。
- 上述示例程序中属性列age的版本为1498184687000，这个值2017年06月23日，如果当前时间 - max_time_deviation（创建表时指定）大于1498184687000值，则PutRow的时候会被禁止。
- 若执行成功，则不会抛异常，否则会抛异常。

详细代码请参见*PutRow@GitHub*。

读取一行数据（GetRow）

根据给定的主键读取单行数据。

接口

```
        """
         说明：获取一行数据。
         ``table_name``是对应的表名。
         ``primary_key``是主键，类型为dict。
```

```
        ``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。
        ``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。
        ``time_range``是可选参数，表示读取额版本范围或特定版本，和max_version至少存在一个。

        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，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):
```

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']

 # 增加列filter：如果growth列的值不等于0.9则返回此行。
    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:
   # 打印出每一列的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.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_mess
age:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.
get_error_message(), e.get_request_id())
```

说明：

查询一行数据时，默认返回这一行所有列的数据。如果想只返回特定行，可以通过 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 的值。

详细代码请参见*GetRow@GitHub*。

更新一行数据（UpdateRow）

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这一行中新增、修改或者删除指定列的值。

接口

```
        """
        说明：更新一行数据。
        ``table_name``是对应的表名。
        ``row``表示更新的行数据，包括主键列和属性列，主键列是list；属性列是dict
。
        ``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。
        目前只支持对行的存在性进行检查，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return_row。
        consumed表示消耗的CapacityUnit，是tablestore.metadata.CapacityUn
it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)
```

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101。
    primary_key = [('uid',1), ('gid',101)]

 # 更新包括PUT，DELETE和DELETE_ALL 三部分。
 # PUT：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: 删除版本为1488436949003的address列的值。
 # DELETE_ALL：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)

 # 行条件为：忽略，不管行是否存在，都会更新。
    condition = Condition(RowExistenceExpectation.IGNORE, SingleColu
mnCondition("age", 20, ComparatorType.EQUAL)) # update row on\
ly 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_mess
age:%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**）

接口

```
        """
        说明：删除一行数据。
        ``table_name``是对应的表名。
        ``row``表示行数据，在delete_row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。
        目前只支持对行的存在性进行检查，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.CapacityUn
it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
):
```

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_mess
age:%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

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

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

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。

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。

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_filter=None))
        request.add(TableInBatchGetRowItem(myTable1, primary_keys,
column_to_get=None, column_filter=None))
        request.add(TableInBatchGetRowItem(myTable2, primary_keys,
column_to_get=None, column_filter=None))
        request.add(TableInBatchGetRowItem(myTable3, primary_keys,
column_to_get=None, column_filter=None))
        response = client.batch_get_row(request)
        ``response``为返回的结果，类型为tablestore.metadata.BatchGetRo
wResponse
        """
  def batch_get_row(self, request):
```

示例

批量一次读 3 行。

```
    # 需要返回的列
    columns_to_get = ['name', 'mobile', 'address', 'age']
```

```
 # 读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：name等于Johb，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表示最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, item.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, item.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.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_mess
age:%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*BatchGetRow@GitHub*。

批量写（BatchWriteRow）

批量插入、修改或删除一个或多个表中的若干行数据。

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。

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.BatchWrite
RowResponse
        """
  def batch_write_row(self, request):
```

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)), ('age',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)

 # 构造批量写的请求。
```

```
    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，如果请求参数等有错误会抛异常；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.error_message))

  ## 检查Update的行结果
     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, item.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, item.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.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_mess
age:%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*BatchWriteRow@GitHub*。

范围读（GetRange）

读取指定主键范围内的数据。

接口

```
        """
        说明：根据范围条件获取多行数据。
        ``table_name``是对应的表名。
        ``direction``表示范围的方向，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，与time_range必须存
在一个。
        ``time_range``是可选参数，表示返回的版本的范围，于max_version必须存在
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):
```

示例

范围读取。

```
# 范围查询的起始主键
inclusive_start_primary_key = [('uid',INF_MIN), ('gid',INF_MIN)]

# 范围查询的结束主键
   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，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, Comparator
Type.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
    )

     all_rows = []
     all_rows.extend(row_list)

  # 当next_start_primary_key 不为空的时候，说明还有数据，继续循环读取。
     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_mess
age:%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*GetRange@GitHub*。

5.7 错误处理

方式

Table Store 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：

批量相关接口比如 batch_get_row 和 batch_write_row， 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

Table Store python SDK 中有 OTSClientError 和 OTSServiceError 两种异常，他们都最终继承自 Exception。

- OTSClientError：指 SDK 内部出现的异常，比如参数设置不对，返回结果解析失败等。
- OTSServiceError：指服务器端错误，它来自于对服务器错误信息的解析。包含以下几个成员：
  - get_http_status：HTTP 返回码，比如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：用于唯一标识该次请求的 UUID。当当您无法解决问题时，可以记录这个 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，请参考 tablestore/retry.py。
- 用户也可以通过继承 RetryPolicy 类实现自己的重试策略，在构造 OTSClient 对象的时候，将其作为参数传入。

目前SDK中已经实现的重试策略：

- 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3秒。
- NoRetryPolicy：不进行任何重试。
- NoDelayRetryPolicy：没有延时的重试策略，慎用。
- WriteRetryPolicy：在默认重试策略基础上，会对写操作重试。

6 .NET SDK

6.1 前言

简介

本文档主要介绍 Table Store（原 OTS） C# SDK 的安装和使用，适用 3.0.0 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。
- 如果您还没有创建 AccessKeyId 和 AccessKeySecret，请到阿里云 *Access Key* 的管理控制台创建 Access Key。

SDK 下载

- *SDK 包*
- *GitHub*

版本迭代详情参考这里。

兼容性

对于 2.x.x 系列的 SDK：

- 接口部分不兼容：删除了`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：3.0.0。

变更内容

- 新增 filter。
- 消除编译时的 warnning。
- 清理没用的依赖包。
- 清理没用的代码。
- 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。
- DLL 文件名称由 Aliyun.dll 变更为 Aliyun.TableStore.dll。
- 开源到 GitHub。

6.2 安装

版本依赖

Windows

- 适用于`.NET 4.0`及以上版本。
- 适用于`Visual Studio 2010`及以上版本。

Windows 环境安装

- NuGet 安装步骤
  1. 如果您的 Visual Studio 没有安装 NuGet，请先安装 *NuGet*。
  2. 安装好 NuGet 后，先在 Visual Studio 中新建或者打开已有的项目，然后单击工具 > **NuGet 程序包管理器 > 管理解决方案的 NuGet 程序包**。
  3. 搜索 aliyun.tablestore，在结果中找到 Aliyun.TableStore.SDK。
  4. 选择最新版本，单击安装，成功后会添加到项目应用中。
- GitHub 安装步骤
  1. 如果没有安装 git，请先安装 *git*。
  2. 使用 git 下载源码：*git clone* 。
  3. 下载好源码后，按照本章最后的项目引入方式进行安装即可。
- DLL 引用方式安装步骤
  1. 单击下载 *SDK 安装包*。解压后 bin 目录包括了 Aliyun.TableStore.dll 文件。
  2. 在 Visual Studio 的**解决方案资源管理器**中选择您的项目，然后右击**项目名称 > 引用**，在弹出的菜单中选择**添加引用**。
  3. 弹出添加引用对话框后，单击**浏览**，找到 SDK 包解压的目录。
  4. 在 bin 目录下选中 Aliyun.TableStore.dll 文件，并单击**确定**。
- 项目引入方式安装步骤

1. 如果是下载了 SDK 包或者从 GitHub 上下载了源码，希望源码安装，可以右击**解决方案**，在弹出的菜单中单击**添加 > 现有项目** 。
2. 在弹出的对话框中选择 aliyun-tablestore-sdk.csproj 文件，单击**打开**。
3. 右击**您的项目**，单击**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，然后单击**确定**即可。

6.3 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、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，目前支持下列形式。

Endpoint 类型	解释
区域地址	使用表格存储实例（Instance）所在的区域地址，如`https://instance.cn-hangzhou.ots.aliyuncs.com`。

表格存储的区域地址

使用表格存储实例的所在区域地址，可以通过以下方式查询 Endpoint：

1. 登录表格存储控制台。
2. 进入 Instance 概览页，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 Access Key（包括 AccessKeyId 和 AccessKeySecret），用来进行签名认证。可以通过如下步骤获得：

1. 在阿里云官网注册阿里云帐号。
2. 登录*AccessKey*管理控制台创建 AccessKeyId 和 AccessKeySecret。

在获取到 AccessKeyId 和 AccessKeySecret 之后，您可以按照下面步骤进行初始化对接。

使用表格存储的 Endpoint 新建 Client。

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：

```
 // 构造一个OTSClientConfig对象
 var config = new OTSClientConfig(Endpoint, AccessKeyId, AccessKeyS
ecret, 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
```

说明：

- OTSClientConfig 中还可以设置 ConnectionLimit。如果不设，默认值是 300。
- OTSClientConfig 中的 OTSDebugLogHandler 和 OTSErrorLogHandler 控制日志行为，用户可以自定义。
- OTSClientConfig 中的 RetryPolicy 控制重试逻辑，目前有默认重试策略，用户也可以自定义重试策略。

多线程

- 支持多线程。
- 使用多线程时，建议共用一个 OTSClient 对象。

6.4 表操作

表格存储的 SDK 提供了 CreateTable、ListTable、DeleteTable、UpdateTable 和 DescribeTable 等表级别的操作接口。

创建表（CreateTable）

根据给定的表的结构信息创建相应的表。

创建表格存储的表时必须指定表的主键。主键包含 1~4 个主键列，每一个主键列都有名字和类型。

接口

```
        /// <summary>
        /// 根据表信息（包含表名、主键的设计和预留读写吞吐量）创建表。
        /// </summary>
        /// <param name="request">请求参数</param>
        /// <returns>CreateTable的返回，这个返回实例是空的，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(CreateTabl
eRequest request);
```

说明：

表格存储的表在被创建之后需要几秒钟进行加载，创建成功后需要等待几秒钟后再做其他操作。

示例

创建一个有 2 个主键列，预留读/写吞吐量为 (0,0) 的表。

```
        //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，这个同时也是分片列
        //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
);

        // 设定预留读吞吐量为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, reservedTh
roughput);

            // 调用client的CreateTable接口，如果没有抛出异常，则说明成功，否则
失败
            otsClient.CreateTable(request);

            Console.WriteLine("Create table succeeded.");
        }
        // 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*。

列出表名称（ListTable）

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

获取实例下的所有表名。

```
        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);
```

```
        }
```

更新表（UpdateTable）

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，新设定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(UpdateTabl
eRequest request);
```

示例

更新表的 CU 值为读 1，写 2。

```
        // 设定新的预留读吞吐量为1，写吞吐量为2
        var reservedThroughput = new CapacityUnit(1, 2);

        // 构造UpdateTableRequest对象
        var request = new UpdateTableRequest("SampleTable", reservedTh
roughput);
        try
        {
            // 调用接口更新表的预留读写吞吐量
            otsClient.UpdateTable(request);

            // 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，则说莫执行失败，打印出错误信息
            Console.WriteLine("Update table failed, exception:{0}", ex
.Message);
        }
```

详细代码请参见*UpdateTable@GitHub*。

查询表描述信息（DescribeTable）

查询指定表的结构信息和预留读/写吞吐量设置信息。

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(DescribeTa
bleRequest request);

        /// <summary>
        /// DescribeTable的异步形式。
        /// </summary>
        public Task<DescribeTableResponse> DescribeTableAsync(
DescribeTableRequest request);
```

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.LastIncreaseTime);
            Console.WriteLine("LastDecreaseTime: {0}", response.
ReservedThroughputDetails.LastDecreaseTime);
            Console.WriteLine("NumberOfDecreaseToday: {0}", response.
ReservedThroughputDetails.LastIncreaseTime);
            Console.WriteLine("ReadCapacity: {0}", response.ReservedTh
roughputDetails.CapacityUnit.Read);
            Console.WriteLine("WriteCapacity: {0}", response.
ReservedThroughputDetails.CapacityUnit.Write);
        }
        catch (Exception ex)
        {
            //如果抛出异常，则说明执行失败，打印错误信息
            Console.WriteLine("Describe table failed, exception:{0}",
ex.Message);
        }
```

详细代码请参见*DescribeTable@GitHub*。

删除表（DeleteTable）

删除本实例下指定的表。

接口

```
        /// <summary>
        /// 根据表名删除表。
        /// </summary>
```

```
        /// <param name="request">请求参数，包含表名</param>
        /// <returns>DeleteTable的返回，这个返回实例是空的，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(DeleteTabl
eRequest request);
```

示例

删除表。

```
        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*。

6.5 单行数据操作

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

插入一行数据（**PutRow**）

插入数据到指定的行。

接口

```
        /// <summary>
        /// 指定表名、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
);
```

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表示不
管此行是否存在都执行
            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，打印出错误信息
            Console.WriteLine("Put row failed, exception:{0}", ex.
Message);
        }
```

 **说明：**

- Condition.IGNORE、Condition.EXPECT_EXIST 和 Condition.EXPECT_NOT_EXIST 从 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 表示不管此行是否已经存在，都会插入新数据，如果之前有会被覆盖。
- RowExistenceExpectation.EXPECT_EXIST 表示只有此行存在时，才会插入新数据，此时，原有数据也会被覆盖。

- RowExistenceExpectation.EXPECT_NOT_EXIST 表示只有此行不存在时，才会插入数据，否则不执行。
- 从 2.2.0 版本开始，Condition 不仅支持行条件，也支持列条件。

详细代码请参见*PutRow@GitHub*。

示例 2

设置条件插入一行数据。

下列示例演示：当行存在，且 col1 大于 24 的时候才执行插入操作。

```
        // 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(
RowExistenceExpectation.EXPECT_EXIST),
                                        primaryKey, attribute);

        // 当col0列的值大于24的时候，允许再次put row，覆盖掉原值
        try
        {
            request.Condition.ColumnCondition = new Relational
Condition("col0",
                                                RelationalCondition.
CompareOperator.GREATER_THAN,
                                                new ColumnValue(24));
            otsClient.PutRow(request);

            Console.WriteLine("Put row succeeded.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Put row failed. error:{0}", ex.Message
);
        }
```

 说明：

- 条件不仅支持单个条件，也支持多个条件组合。例如，col1 大于 5 且 pk2 小于'xyz'时插入数据。
- 属性列和主键列都支持条件。

- 当条件中的列在某行不存在时，可以通过 RelationCondition 中的 PassIfMissing 控制，默认是 true。

详细代码请参见*ConditionPutRow@GitHub*。

示例 3

异步插入一行数据。

```
        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(RowExistenceExpectation.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.ConsumedCapacityUnit.Read,
                                  task.Result.ConsumedCapacityUnit.
Write);
            }

            // 如果没有抛出异常，则说明插入数据成功
            Console.WriteLine("Put row async succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则打印出出错信息
            Console.WriteLine("Put row async failed. exception:{0}",
ex.Message);
        }
```

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的时候，可能会出现超时。

详细代码请参见*PutRowAsync@GitHub*。

读取一行数据（GetRow）

根据给定的主键读取单行数据。

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
);
```

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，这里未指定读哪列，默认读整行
            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，这里省略，详见下面GitHub的链接

            // 如果没有抛出异常，则说明成功
            Console.WriteLine("Get row succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，说明执行失败，打印出错误信息
            Console.WriteLine("Update table failed, exception:{0}", ex
.Message);
```

```
        }
```

说明：

- 查询一行数据时，默认返回这一行所有列的数据。如果想只返回特定行，可以通过 columnsToGet 参数限制。如果将 col0 和 col1 加入到 columnsToGet 中，则只返回 col0 和 col1 的值。
- 查询时也支持按条件过滤，比如当 col0 的值大于 24 时才返回结果。
- 当 columnsToGet 和 condition 同时使用时，顺序是 columnsToGet 先生效，然后再去返回的列中进行过滤。
- 当某列不存在时的行为，可以通过 PassIfMissing 控制。

详细代码请参见*GetRow@GitHub*。

示例 2

使用过滤读取一行数据。

下面演示查询数据，但只返回 col0 和 col1 的数据，同时在 col0 上面过滤，要求的条件是 col0=24 。

```
        // 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：col0的值等于5
        var filter1 = new RelationalCondition("col0",
                    RelationalCondition.CompareOperator.EQUAL,
                    new ColumnValue(5));

        // 条件2：col1不等于ff的行
        var filter2 = new RelationalCondition("col1", Relational
Condition.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[col0,col1]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，这里省略

            // 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，打印出错误信息
            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）

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这一行中新增、修改或者删除指定列的值。

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
内容在这一行中新增、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);
```

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表示不管
此行是否存在都执行
            var request = new UpdateRowRequest(TableName, new
Condition(RowExistenceExpectation.IGNORE),
                                        primaryKey, attribute);
            // 调用UpdateRow接口执行
            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，说明执行失败，打印异常信息
            Console.WriteLine("Update row failed, exception:{0}", ex.
Message);
        }
```

 说明：

更新一行数据也支持条件语句。

详细代码请参见*UpdateRow@GitHub*。

删除一行数据（**DeleteRow**）

接口

```
        /// <summary>
        /// 指定表名和主键，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);
```

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，Condition.EXPECT_EXIST表示只有此行存在时才执行
            var deleteRowRequest = new DeleteRowRequest("SampleTable
", Condition.EXPECT_EXIST, primaryKey);

            // 调用DeleteRow接口执行删除
            otsClient.DeleteRow(deleteRowRequest);

            // 如果没有抛出异常，则表示成功
            Console.Writeline("Delete table succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，说明删除失败，打印粗错误信息
            Console.WriteLine("Delete table failed, exception:{0}", ex
.Message);
        }
```

说明：

删除一行数据也支持条件语句。

详细代码请参见*DeleteRow@GitHub*。

6.6 多行数据操作

表格存储的 SDK 提供了 BatchGetRow、BatchWriteRow、GetRange 和 GetRangeIterator 等多行操作的接口。

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

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。

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。

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(BatchGetRo
wRequest request);
```

示例

批量一次读 10 行。

```
        // 构造批量读取请求的对象，设置10行的pk值
        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，查询十行数据
            var response = otsClient.BatchGetRow(request);
            var tableRows = response.RowDataGroupByTable;
            var rows = tableRows[TableName];

            // 输入rows里的数据，这里省略，详见下面GitHub链接

            // 批量操作可能部分成功部分失败，需要为每行检查状态，详见下面GitHub
链接
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则说明执行失败，打印出错误信息
            Console.WriteLine("Batch get row failed, exception:{0}",
ex.Message);
        }
```

 说明：

批量读也支持通过条件语句过滤。

详细代码请参见*BatchGetRow@GitHub*。

批量写（**BatchWriteRow**）

批量插入、修改或删除一个或多个表中的若干行数据。

BatchWriteRow 操作可视为多个 PutRow、UpdateRow 和 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。

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(BatchWrite
RowRequest request);

        /// <summary>
        /// BatchWriteRow的异步形式。
        /// </summary>
        /// <param name="request"></param>
        /// <returns></returns>
        public Task<BatchWriteRowResponse> BatchWriteRowAsync(
BatchWriteRowRequest request);
```

示例

批量导入 100 行数据。

```
        // 构造批量插入的请求对象，包括了这100行数据的pk
        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"));

            // 定义要写入改行的属性列
            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, attribute);
```

```
        }

        request.Add(TableName, rowChanges);

        try
        {
            // 调用BatchWriteRow接口
            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，需要为每行检查状态，详见下面GitHub
链接
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则说明执行失败，打印出错误信息
            Console.WriteLine("Batch put row failed, exception:{0}",
ex.Message);
        }
```

 说明：

批量写也支持条件语句。

详细代码请参见*BatchWriteRow@GitHub*。

范围读（GetRange）

读取指定主键范围内的数据。

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);
```

示例

范围读取。

```
        // 读取 (0, INF_MIN)到(100, INF_MAX)这个范围内的所有行
        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
        {
            // 构造范围查询请求对象
            var request = new GetRangeRequest(TableName, GetRangeDi
rection.Forward,
                                inclusiveStartPrimaryKey, exclusiveE
ndPrimaryKey);

            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, GetRangeDi
rection.Forward,
                                    nextStartPrimaryKey, exclusiveE
ndPrimaryKey);
                response = otsClient.GetRange(request);
                nextStartPrimaryKey = response.NextPrimaryKey;
                foreach (RowDataFromGetRange row in response.
RowDataList)
                {
                    rows.Add(row);
                }
            }

            // 输出Rows的数据，这里省略，详见下面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，打印出错误信息
            Console.WriteLine("Get range failed, exception:{0}", ex.
Message);
        }
```

说明：

按范围读也支持通过条件语句过滤。

详细代码请参见*GetRange@GitHub*。

迭代读（**GetRangeIterator**）

获取一个范围查询的迭代器。

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。

```
        // 读取 (0, "a")到(1000, "xyz")这个范围内的所有行
        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，这里也支持过滤条件
            var request = new GetIteratorRequest(TableName, GetRangeDi
rection.Forward, inclusiveStartPrimaryKey,
                                                 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);
        }
```

说明：

读数据迭代器也支持通过条件语句过滤

详细代码请参见*GetRangeIterator@GitHub*。

6.7 错误处理

方式

Table Store .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：

批量相关接口比如 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

Table Store .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，比如参数设置不对，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，比如 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，详情请参见 Aliyun.OTS.Retry.DefaultRetryPolicy 类。
- 用户也可以通过修改 OTSClientConfig 中的 RetryPolicy 自定义重试策略。

7 C++ SDK

7.1 前言

本文档主要介绍 Table Store（原 OTS）C++ 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。
- 如果您还没有创建 AccessKeyId 和 AccessKeySecret，请到阿里云 *Access Key* 的管理控制台创建 Access Key。

下载地址：*GitHub*

7.2 安装

由于C++的特殊性，我们建议您先用本文介绍的方法编译，复制`build/release/src/tablestore/core/impl/buildinfo.cpp`备用。然后将C++ SDK源码以及`buildinfo.cpp`复制到您自己的代码库中，并用您的编译系统来进行编译。

编译参数

在编译客户端代码的时候，有些编译器的行为是必须保证的，即，某些编译器参数是必须的。

以下是针对gcc编译器的编译参数及说明。

参数	是否必须	说明
--std=gnu++03	必须	支持C++98 TR1语言版本，带gcc扩展（即typeof）。
-pthread	必须	多线程编程的必要参数。编译或者链接都需要加上该参数。
-fwrapv	必须	整型数据溢出则回转，即，无符号整型向上溢出则成为0，有符号整型向上溢出则成为最小的负数。客户端基于这个行为做溢出检查。
-O2	建议存在	优化级别。一般不建议更高的优化级别。
-fsanitize=address和-fvar-tracking-assignments	建议存在	gcc-4.9之后支持libasan，可以快速而轻量地检测各种内存

参数	是否必须	说明
		使用上的错误。如果需要表格存储的开发人员定位错误，请带上这两个编译参数复现错误，并且在链接的时候也需要带上前一个参数。 注意：libasan和valgrind不兼容。

环境依赖与预编译包

本文以debian8系统为例说明生成预编译包的方法。

1. 打开`docker/debian8/Dockerfile`文件，选择dockerfile的方式输出客户端对系统的环境依赖。这种方式可以自动保证代码和环境的一致性。

   ```
   RUN apt-get install -y scons g++ libboost-all-dev protobuf-compiler
   libprotobuf-dev uuid-dev libssl-dev
   RUN apt-get install -y ca-certificates # for HTTPS support
   ```

   SDK依赖以下几项：

   - scons & gcc: 编译系统
   - boost
   - uuid
   - protobuf：序列化库
   - openssl：签名，以及支持HTTPS所用
   - ca-certificates：仅为支持HTTPS所用。如果您只用表格存储的HTTP地址，可以不安装这个库。建议使用更为安全的HTTPS。

2. 安装以上包之后便可以编译客户端。方法是下载客户端的源码，并在源码目录下执行`scons`。

   ```
   $ git clone https://github.com/aliyun/aliyun-tablestore-cpp-sdk.git
   $ cd aliyun-tablestore-cpp-sdk
   $ scons -j4
   ```

当上述步骤正常结束，一个tar包就编译好了。包名通常可以在`scons`最后的输出中找到。例如debian8系统，包名及所在的路径是：`build/release/pkg/aliyun-tablestore-cpp98-sdk-4.4.1-debian8.9-x86_64.tar.gz`。

- 包名里包含以下几个要素：

- C++版本（C++98）
- SDK版本（4.4.1）
- OS版本（debian8.9）
- OS架构（x86_64）

- 包的内容包括：

```
$ tar -tf build/release/pkg/aliyun-tablestore-cpp98-sdk-4.4.1-
debian8.9-x86_64.tar.gz
version.ini
lib/libtablestore_core.so
lib/libtablestore_core_static.a
lib/libtablestore_util.so
lib/libtablestore_util_static.a
include/tablestore/util/arithmetic.hpp
include/tablestore/util/assert.hpp
include/tablestore/util/foreach.hpp
include/tablestore/util/iterator.hpp
include/tablestore/util/logger.hpp
include/tablestore/util/logging.hpp
include/tablestore/util/mempiece.hpp
include/tablestore/util/mempool.hpp
include/tablestore/util/metaprogramming.hpp
include/tablestore/util/move.hpp
include/tablestore/util/optional.hpp
include/tablestore/util/prettyprint.hpp
include/tablestore/util/random.hpp
include/tablestore/util/result.hpp
include/tablestore/util/security.hpp
include/tablestore/util/seq_gen.hpp
include/tablestore/util/threading.hpp
include/tablestore/util/timestamp.hpp
include/tablestore/util/try.hpp
include/tablestore/util/assert.ipp
include/tablestore/util/iterator.ipp
include/tablestore/util/logging.ipp
include/tablestore/util/mempiece.ipp
include/tablestore/util/move.ipp
include/tablestore/util/prettyprint.ipp
include/tablestore/core/client.hpp
include/tablestore/core/error.hpp
include/tablestore/core/range_iterator.hpp
include/tablestore/core/retry.hpp
include/tablestore/core/types.hpp
```

即，包内容包含以下元素：

- 版本文件：`version.ini`
- 库文件：`lib/`下所有文件。其中`libtablestore_core_static.a`依赖`libtablestore_util_static.a`，动态库也类似。
- 头文件：`include/`下所有文件。

7.3 初始化

表格存储的客户端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Table Store服务在各个区域的域名地址，目前支持下列形式。

示例	描述
`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。

说明：

- 表格存储支持公网访问，也支持私网地址。
- 您可以登录表格存储控制台，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：

- 主帐号的 AccessKeyId 和 AccessKeySecret。创建步骤如下：
  1. 在阿里云官网注册阿里云帐号。
  2. 登录*AccessKey*管理控制台创建 AccessKeyId 和 AccessKeySecret。
- 被授予访问表格存储权限的子帐号的 AccessKeyId 和 AccesskeySecret。创建步骤如下：
  1. 使用主帐号前往访问控制 *RAM*，创建一个新的子帐号或者使用已经存在的子帐号。
  2. 使用主帐号授予子帐号访问表格存储的权限。
  3. 子帐号被授权后，就可以使用自己的 AccessKeyId 和 AccessKeySecret 访问了。
- 临时访问的 STS token。获取步骤如下：
  1. 应用的服务器通过访问 RAM/STS 服务，获取一个临时的 Accesskeyld、AccesskeySecret 和 token，发送给使用方。
  2. 使用方使用上述临时密钥访问表格存储服务。

新建Client

用户使用表格存储的 SDK 时，必须首先构造一个 Client，通过调用这个 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 RESTful API 是一致的。

Client类型

表格存储 C++ SDK 提供两种 Client，SyncClient 和 AsyncClient，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。

同步接口

- 直接创建（即使用Table Store Endpoint新建Client）

```
Endpoint ep("YourEndpoint", "YourInstance");
Credential cr("AccessKeyId", "AccessKeySecret");
ClientOptions opts;
SyncClient* client = NULL;
Optional<OTSError> res = SyncClient::create(client, ep, cr, opts);
```

 说明：

建议您避免使用主账号的AcceessKey来访问表格存储，推荐使用临时令牌或者子账号的AccessKey。如果使用临时令牌STS，上述代码中的Credential对象需要修改为：`Credential cr("AccessKeyId", "AccessKeySecret", "SecurityToken");`。

ClientOptions中包含的配置项说明如下，您可以使用默认值，也可以自定义参数。

mMaxConnections	最大连接数，同时也是最大并发请求数。SDK和表格存储服务端保持着长连接。每次有一个新的请求都会从闲置的连接里随机挑一个来发送请求。

mConnectTimeout	连接超时时间。考虑到DNS解析的时间，建议连接超时时间不短于10秒。
mRequestTimeout	请求超时时间。
mRetryStrategy	重试策略。默认的重试策略会在10秒内重试失败的幂等请求。您可以定义自己的重试策略。
mLogger	日志记录器。默认的日志记录器输出到标准错误上。建议您定义自己的日志记录器。
mActors	线程池。用于执行您回调的线程池。默认10根线程。

- 从AsyncClient创建

```
AsyncClient& async = ...;
SyncClient* sync = SyncClient::create(async);
```

异步接口

异步接口Client的创建和使用，请参见异步接口。

多线程

支持多线程。使用多线程时，建议共用一个客户端对象。

7.4 表操作

SDK提供了CreateTable、ListTable、UpdateTable、DescribeTable和DeleteTable等表级别的操作接口。

> 说明：
> 以下操作为同步接口的示例，异步接口的操作请参见异步接口。

创建表（**CreateTable**）

创建表时必须指定表的名字和主键。主键包含 1~4 个主键列，每一个主键列都有名字和类型。

表格存储的表可以设置自增主键列，详情参见主键列自增。

示例

在本示例中，表名为simple_create_delete_table，主键只含一个主键列，名为pkey，类型为整型（kPKT_Integer）。

```
CreateTableRequest req;
{
```

```
    // immutable configurations of the table
    TableMeta& meta = req.mutableMeta();
    meta.mutableTableName() = "simple_create_delete_table";
    {
        // with exactly one integer primary key column
        Schema& schema = meta.mutableSchema();
        PrimaryKeyColumnSchema& pkColSchema = schema.append();
        pkColSchema.mutableName() = "pkey";
        pkColSchema.mutableType() = kPKT_Integer;
    }
}
CreateTableResponse resp;
Optional<OTSError> res = client.createTable(resp, req);
```

说明：

详细代码在*createTable@GitHub*获取。

可变参数

您可以在数据表上设置若干可变参数。可变参数可以在建表时设定，也可以通过更新表来修改。

可变参数包括以下几项：

可变参数	名称	默认值
mutableTimeToLive()	数据生命周期	-1（即永不过期）
mutableMaxVersions()	最大版本数	1
mutableMaxTimeDeviation()	有效版本偏差	86400秒（即一天）
mutableReservedThroughput()	预留读写吞吐量	0（即全部读写按量计费）

以下是一个建表时设定预留读写吞吐量的示例：

```
CreateTableRequest req;
{
    // immutable configurations of the table
    TableMeta& meta = req.mutableMeta();
    meta.mutableTableName() = "create_table_with_reserved_throughput";
    {
        // with exactly one integer primary key column
        Schema& schema = meta.mutableSchema();
        PrimaryKeyColumnSchema& pkColSchema = schema.append();
        pkColSchema.mutableName() = "pkey";
        pkColSchema.mutableType() = kPKT_Integer;
    }
}
{
    TableOptions& opts = req.mutableOptions();
    {
        // 0 reserved read capacity-unit, 1 reserved write capacity-
unit
        CapacityUnit cu(0, 1);
        opts.mutableReservedThroughput().reset(util::move(cu));
```

```
    }
}
CreateTableResponse resp;
Optional<OTSError> res = client.createTable(resp, req);
```

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

接口

使用SyncClient::listTable()来列举实例下的所有表。

```
SyncClient* client = ...;
ListTableRequest req;
ListTableResponse resp;
Optional<OTSError> res = client->listTable(resp, req);
```

示例

获取实例下的所有表名。

```
const IVector<string>& xs = resp.tables();
for(int64_t i = 0; i < xs.size(); ++i) {
    cout << xs[i] << endl;
}
```

说明：

详细代码在*listTable@GitHub*获取。

更新表（UpdateTable）

更新指定表的可变参数。

示例

更新预留吞吐量。

```
UpdateTableRequest req;
req.mutableTable() = "YourTable";
UpdateTableResponse resp;
{
    TableOptions& opts = req.mutableOptions();
    {
        // 0 reserved read capacity-unit, 1 reserved write capacity-
unit
        CapacityUnit cu(0, 1);
        opts.mutableReservedThroughput().reset(util::move(cu));
    }
}
```

```
Optional<OTSError> res = client.updateTable(resp, req);
```

说明：

详细代码在*updateTable@GitHub*获取。

查询表信息（**DescribeTable**）

通过`describeTable()`接口可以查询如下表信息：

信息项	描述
表的状态	包括： • kTS_Active：表可以正常提供读写服务。 • kTS_Inactive：表上不可读写，但表上数据保留。通常这个状态出现在主备表切换时。 • kTS_Loading：正在建表过程中。表上不可读写。 • kTS_Unloading：正在删表过程中。表上不可读写。 • kTS_Updating：正在更新表可变参数中。表上不可读写。
表meta	参见创建表。
表的可变参数	参见可变参数。
分片之间的分割点	表格存储上的一张表被水平切分成若干分片。通过这个接口可以获取各分片间的分割点。  说明： 由于表格存储会在后台根据负载进行自动分裂与合并，这个接口取到的分割点保证是曾经出现过的分片情况，但不保证与当前情况完全吻合。

示例

```
DescribeTableRequest req;
req.mutableTable() = "YourTable";
DescribeTableResponse resp;
Optional<OTSError> res = client.describeTable(resp, req);
```

说明：

详细代码在*describeTable@GitHub*获取。

删除表（DeleteTable）

删除本实例下指定的表。只需指定表名。

示例

```
DeleteTableRequest req;
req.mutableTable() = "YourTable";
DeleteTableResponse resp;
Optional<OTSError> res = client.deleteTable(resp, req);
```

说明：

详细代码在*deleteTable@GitHub*获取。

7.5 单行数据操作

Table Store提供了PutRow，GetRow，UpdateRow和DeleteRow等单行操作的接口。

说明：

以下操作为同步接口的示例，异步接口的操作请参见异步接口。

插入一行数据（PutRow）

PutRow 接口用于插入一行数据。若该行不存在则插入，如果该行已经存在则覆盖（即原行的所有列以及所有版本的列值都被删除）。

示例

```
PutRowRequest req;
{
    RowPutChange& chg = req.mutableRowChange();
    chg.mutableTable() = "YourTable";
    {
        // set primary key of the row to put
        PrimaryKey& pkey = chg.mutablePrimaryKey();
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("pkey-value"));
    }
    {
        // set attributes of the row to put
        IVector<Attribute>& attrs = chg.mutableAttributes();
        attrs.append() = Attribute(
            "attr",
            AttributeValue::toInteger(123));
    }
}
PutRowResponse resp;
```

```
Optional<OTSError> res = client.putRow(resp, req);
```

读取一行数据（GetRow）

单行读 GetRow 接口用于读取一行数据。

指定表名和一行的主键，读取的结果可能有两种：

- 若该行存在，则GetRowResponse对象返回该行的各主键列以及属性列。
- 若该行不存在，则GetRowResponse对象不含有行，并且不会报错。

示例

```
GetRowRequest req;
{
    PointQueryCriterion& query = req.mutableQueryCriterion();
    query.mutableTable() = “YourTable”;
    {
        PrimaryKey& pkey = query.mutablePrimaryKey();
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("some_key")); // 假设主键只有一列，并且
类型为字符串
    }
    query.mutableMaxVersions().reset(1);
}
GetRowResponse resp;
Optional<OTSError> res = client.getRow(resp, req);
```

更新一行数据（UpdateRow）

UpdateRow 接口用于更新一行数据，如果原行不存在，则新写入一行。

更新操作有以下四种情况：

- 不指定版本写入一个列值，表格存储服务端会自动补上一个版本号，保证此种情况下版本号的递增。
- 指定版本写入一个列值，若该列原先没有该版本列值，则插入数据，否则覆盖原值。
- 删除指定版本的列值。
- 删除整个列的所有版本列值。

示例

```
UpdateRowRequest req;
{
    RowUpdateChange& chg = req.mutableRowChange();
    chg.mutableTable() = "YourTable";
    {
        // set primary key of the row to put
        PrimaryKey& pkey = chg.mutablePrimaryKey();
```

```
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("pkey"));
    }
    {
        // insert a value without specifying version
        RowUpdateChange::Update& up = chg.mutableUpdates().append();
        up.mutableType() = RowUpdateChange::Update::kPut;
        up.mutableAttrName() = "attr0";
        up.mutableAttrValue().reset(AttributeValue::toStr("new value
without specifying version"));
    }
    {
        // insert a value with version
        RowUpdateChange::Update& up = chg.mutableUpdates().append();
        up.mutableType() = RowUpdateChange::Update::kPut;
        up.mutableAttrName() = "attr1";
        up.mutableAttrValue().reset(AttributeValue::toStr("new value
with version"));
        up.mutableTimestamp().reset(UtcTime::now());
    }
    {
        // delete a value with specific version
        RowUpdateChange::Update& up = chg.mutableUpdates().append();
        up.mutableType() = RowUpdateChange::Update::kDelete;
        up.mutableAttrName() = "attr2";
        up.mutableTimestamp().reset(UtcTime::now());
    }
    {
        // delete all values of a attribute column
        RowUpdateChange::Update& up = chg.mutableUpdates().append();
        up.mutableType() = RowUpdateChange::Update::kDeleteAll;
        up.mutableAttrName() = "attr3";
    }
}
UpdateRowResponse resp;
Optional<OTSError> res = client.updateRow(resp, req);
```

删除一行数据（**DeleteRow**）

DeleteRow 接口用于删除一行。无论该行存在与否都不会报错。

示例

```
DeleteRowRequest req;
{
    RowDeleteChange& chg = req.mutableRowChange();
    chg.mutableTable() = "YourTable";
    {
        // set primary key of the row to delete
        PrimaryKey& pkey = chg.mutablePrimaryKey();
        pkey.append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toInteger(1));
    }
}
DeleteRowResponse resp;
```

```
Optional<OTSError> res = client.deleteRow(resp, req);
```

7.6 多行数据操作

Table Store提供了BatchGetRow，BatchWriteRow和GetRange等多行操作的接口。

说明：

以下操作为同步接口的示例，异步接口的操作请参见异步接口。

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

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。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。

BatchGetRow可能出现以下两种错误：

- 请求整体错误，比如网络错误。这类错误存放在batchGetRow()的返回值里。
- 请求整体没有错误，但个别行出错。这类错误存放在相应的行的结果里。

示例

```
BatchGetRowRequest req;
{
    MultiPointQueryCriterion& criterion = req.mutableCriteria().append
();
    IVector<MultiPointQueryCriterion::RowKey>& rowkeys = criterion.
mutableRowKeys();
    {
        MultiPointQueryCriterion::RowKey& exist = rowkeys.append();
        exist.mutableGet().append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("some key"));
        exist.mutableUserData() = &userDataForSomeKey;
    }
    {
        MultiPointQueryCriterion::RowKey& exist = rowkeys.append();
        exist.mutableGet().append() = PrimaryKeyColumn(
            "pkey",
            PrimaryKeyValue::toStr("another key"));
        exist.mutableUserData() = &userDataForAnotherKey;
    }
    criterion.mutableTable() = "YourTable";
    criterion.mutableMaxVersions().reset(1);
}
BatchGetRowResponse resp;
```

```
Optional<OTSError> res = client.batchGetRow(resp, req);
```

说明：

详细代码可在*batchGetRow@GitHub*获取。

批量写（BatchWriteRow）

批量插入、修改或删除一个或多个表中的若干行数据。

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。

BatchWriteRow可能出现以下两种错误：

- 请求整体的错误，比如网络超时。这类错误存放在batchWriteRow()返回值中。
- 单行上的错误，比如主键值不合法。这类错误存放在BatchWriteRowResponse中的每一行上。

示例

```
static const char kPutRow[] = "PutRow";
static const char kUpdateRow[] = "UpdateRow";
static const char kDeleteRow[] = "DeleteRow";

BatchWriteRowRequest req;
{
    // put row
    BatchWriteRowRequest::Put& put = req.mutablePuts().append();
    put.mutableUserData() = kPutRow;
    put.mutableGet().mutableTable() = kTableName;
    PrimaryKey& pkey = put.mutableGet().mutablePrimaryKey();
    pkey.append() = PrimaryKeyColumn(
        "pkey",
        PrimaryKeyValue::toStr("row to put"));
}
{
    // update row
    BatchWriteRowRequest::Update& update = req.mutableUpdates().append
();
    update.mutableUserData() = kUpdateRow;
    update.mutableGet().mutableTable() = kTableName;
    PrimaryKey& pkey = update.mutableGet().mutablePrimaryKey();
    pkey.append() = PrimaryKeyColumn(
        "pkey",
        PrimaryKeyValue::toStr("row to update"));
    RowUpdateChange::Update& attr = update.mutableGet().mutableUpdates
().append();
    attr.mutableType() = RowUpdateChange::Update::kPut;
    attr.mutableAttrName() = "attr0";
    attr.mutableAttrValue().reset(AttributeValue::toStr("some value
"));
}
{
    // delete row
    BatchWriteRowRequest::Delete& del = req.mutableDeletes().append();
```

```
    del.mutableUserData() = kDeleteRow;
    del.mutableGet().mutableTable() = kTableName;
    PrimaryKey& pkey = del.mutableGet().mutablePrimaryKey();
    pkey.append() = PrimaryKeyColumn(
        "pkey",
        PrimaryKeyValue::toStr("row to delete"));
}
BatchWriteRowResponse resp;
Optional<OTSError> res = client.batchWriteRow(resp, req);
```

说明：

详细代码可在*batchWriteRow@GitHub*获取。

范围读（GetRange）

读取指定主键范围内的数据。

建议使用`RangeIterator`。构造RangeIterator需要提供：

- 异步客户端。
- RangeQueryCriterion与单行读的PointQueryCriterion类似，但RangeQueryCriterion还需要：
  - 设定范围的起始点（包含）和终止点（不包含）。除了正常主键键值之外，还可以使用“负无穷大”（严格小于所有正常主键列值）和“正无穷大”（严格大于所有正常主键列值）两个特殊值。
  - 设定正序读取（由小及大）或者反序读取（由大及小）。默认为正序读取。正序读取时，起始点须小于终止点；反序读取时，起始点须大于终止点。

RangeIterator对象是个Iterator，提供三个接口：

- `moveNext()`将RangeIterator对象移动到下一行。刚构造出来的RangeIterator对象必须调用`moveNext()`才可以取值。如果读取数据失败，`moveNext()`会将错误显示在返回值中。
- `valid()`给出RangeIterator对象是否走到了范围的终点。
- 如果`valid()`为true，则可以通过`get()`读取到行对象。您可以将`get()`返回的行对象的内容搬移走以避免内存复制。如果移到他处，紧接着的`get()`将返回搬移后的内容。

示例

```
RangeQueryCriterion query;
query.mutableTable() = "YourTable";
query.mutableMaxVersions().reset(1);
{
    PrimaryKey& start = query.mutableInclusiveStart();
    start.append() = PrimaryKeyColumn(
        "pkey",
        PrimaryKeyValue::toInfMin());
}
```

```
{
    PrimaryKey& end = query.mutableExclusiveEnd();
    end.append() = PrimaryKeyColumn(
        "pkey",
        PrimaryKeyValue::toInfMax());
}
auto_ptr<AsyncClient> aclient(AsyncClient::create(client));
RangeIterator iter(*aclient, query);
for(;;) {
    Optional<OTSError> err = iter.moveNext();
    if (err.present()) {
        // do something with err
        abort();
    }
    if (!iter.valid()) {
        break;
    }
    Row& row = iter.get();
    // do something with row
}
```

说明：

详细代码可在*GetRange@GitHub*获取。

指定列读取

表格存储支持无限宽度的行，但一般无需读取整行，只需指定若干列读取即可。`QueryCriterion`（PointQueryCriterion、MultiPointQueryCriterion和RangeQueryCriterion的基类）提供了`mutableColumnsToGet()`方法来指定需要读取的列，既可以是属性列，也可以是主键列，如果为空则读取整行。

如果指定的列在读取的行上不存在，返回的结果里便缺失这个列。表格存储不提供占位符。

在范围读中，如果指定的列全部是属性列，而范围内某行恰好缺少全部指定的列，那么在结果中并不会出现这一行。如果确实需要感知到该行，可以将主键列加入到指定列之中。

指定版本读取

每个属性列可以包含多个版本，每个版本号（时间戳）对应一个列值。读取的时候可以指定读取多少个版本（`mutableMaxVersions()`）以及读取的版本范围（`mutableTimeRange()`）。

最大版本数和版本范围，至少指定其中一项：

- 如果仅指定版本数，则返回所有版本里从新到旧至多指定数量个数据。
- 如果仅指定版本范围，则返回该范围内所有数据。
- 如果同时指定版本数和版本范围，则返回版本范围内从新到旧至多指定数量个数据。

条件写

条件写是指在写一行之前先检查条件，当条件成立才实际写入。表格存储保证条件检查和写入的原子性。

表格存储支持行存在性条件和列值条件：

- 行存在性条件分为以下几种：
  - Ignore：忽略。默认值，无论行是否存在都写入。
  - ExpectExist：期望存在。行存在则写入。
  - ExpectNotExist：期望不存在。行不存在则写入。
- 列值条件等同于过滤器。

7.7 过滤器（Filter）

表格存储可以在服务端对读取的结果再进行一次过滤，以便减少网络上传输的数据量。

过滤器是一个树形结构，内节点为逻辑运算（CompositeColumnCondition），叶节点为比较判断（SingleColumnCondition）。

- CompositeColumnCondition支持与、或、非。其中与和或可以挂载两个或更多子节点，非只能挂载一个子节点。
- SingleColumnCondition支持全部6种比较条件（等于、不等于、大于、小于、大于等于、小于等于）。
- 每个SingleColumnCondition对象支持一列（可以是主键列）和一个常量比较。不支持两列相比较，也不支持两个常量相比较。
- SingleColumnCondition的latestVersionOnly参数控制多个版本的列值如何参与比较，默认为true。
  - 若为true，则只有版本范围内的最新版本列值参与比较（仅仅是参与比较，如果是过滤器认可该行，其他版本的列值依旧会返回）。
  - 若为false，则任意一个列值满足条件，该节点即认为true。
- SingleColumnCondition的passIfMissing参数控制列值缺失时该节点应当视作何值，默认false。

过滤器作用于其他条件筛选的结果上。所以，如果过滤器中有某一列，而指定列读取中未指定该列，或版本范围内该列没有列值，则过滤器都认为该列缺失。

示例

以下示例从全表中过滤出主键列pkey大于1并且属性列attr等于A的行。

```
RangeQueryCriterion query;
query.mutableTable() = kTableName;
query.mutableMaxVersions().reset(1);
{
  PrimaryKey& start = query.mutableInclusiveStart();
  start.append() = PrimaryKeyColumn(
      "pkey",
      PrimaryKeyValue::toInfMin());
}
{
  PrimaryKey& end = query.mutableExclusiveEnd();
  end.append() = PrimaryKeyColumn(
      "pkey",
      PrimaryKeyValue::toInfMax());
}
{
  // set filter
  shared_ptr<ColumnCondition> pkeyCond(
      new SingleColumnCondition(
          "pkey",
          SingleColumnCondition::kLarger,
          AttributeValue::toInteger(1)));
  shared_ptr<ColumnCondition> attrCond(
      new SingleColumnCondition(
          "attr",
          SingleColumnCondition::kEqual,
          AttributeValue::toStr("A")));
  shared_ptr<CompositeColumnCondition> top(new CompositeColumnCondi
tion());
  top->mutableOp() = CompositeColumnCondition::kAnd;
  top->mutableChildren().append() = pkeyCond;
  top->mutableChildren().append() = attrCond;
  query.mutableFilter() = top;
}
```

7.8 主键列自增

C++ SDK支持主键列自增。如果您设置某一列主键为自增列，在写入一行数据时，这一列主键不用填值，表格存储会自动为您生成这一列主键的值，这个值在分区键上保证唯一，且严格递增。详细信息请参见主键列自增。

创建表

注意：

自增主键列必须是整型，并且需要设置PrimaryKeyColumnSchema::AutoIncrement。

```
CreateTableRequest req;
{
    // immutable configurations of the table
    TableMeta& meta = req.mutableMeta();
```

```
    meta.mutableTableName() = kTableName;
    Schema& schema = meta.mutableSchema();
    {
        PrimaryKeyColumnSchema& pkColSchema = schema.append();
        pkColSchema.mutableName() = "ShardKey";
        pkColSchema.mutableType() = kPKT_String;
    }
    {
        PrimaryKeyColumnSchema& pkColSchema = schema.append();
        pkColSchema.mutableName() = "AutoIncrKey";
        pkColSchema.mutableType() = kPKT_Integer;
        pkColSchema.mutableOption().reset(PrimaryKeyColumnSchema::
AutoIncrement);
    }
}
CreateTableResponse resp;
Optional<OTSError> res = client.createTable(resp, req);
```

写入数据

自增主键列的值由表格存储的服务端填入，因此，写入时主键列必须填入一个特殊的占位符，可以由`PrimaryKeyValue::toAutoIncrement()`取得该占位符对象。如果您想要读取这行数据，可以设置返回值包含主键，服务端会将主键中的自增主键列的占位符替换成实际的列值，然后返回，您可以记录下该主键用于后续使用。

本示例以单行覆写为例，其他写入接口类似。

```
PutRowRequest req;
{
    RowPutChange& chg = req.mutableRowChange();
    chg.mutableTable() = kTableName;
    chg.mutableReturnType() = RowChange::kRT_PrimaryKey;
    PrimaryKey& pkey = chg.mutablePrimaryKey();
    pkey.append() = PrimaryKeyColumn(
        "ShardKey",
        PrimaryKeyValue::toStr("shard0"));
    pkey.append() = PrimaryKeyColumn(
        "AutoIncrKey",
        PrimaryKeyValue::toAutoIncrement());
}
```

7.9 异步接口

新建Client

- 直接创建（即使用Table Store Endpoint新建Client）

```
Endpoint ep("YourEndpoint", "YourInstance");
Credential cr("AccessKeyId", "AccessKeySecret");
ClientOptions opts;
AsyncClient* client = NULL;
```

```
Optional<OTSError> res = AsyncClient::create(client, ep, cr, opts);
```

注意：

建议您避免使用主账号的AcceessKey来访问表格存储，推荐使用临时令牌或者子账号的AccessKey。如果使用临时令牌STS，上述代码中的Credential对象需要修改为：`Credential cr("AccessKeyId", "AccessKeySecret", "SecurityToken");`。

配置项的说明参见同步接口。

- 从SyncClient构造

```
SyncClient& sync = ...;
AsyncClient* async = AsyncClient::create(sync);
```

表操作

以表操作为例说明异步接口的使用。

准备工作

需要准备两个函数：

- 请求对象

  `listTable`的函数签名如下所示：

```
void listTable(
  ListTableRequest&,
  const std::tr1::function<void(
      ListTableRequest&, util::Optional<OTSError>&, ListTableR
esponse&)>&);
```

  第一个参数是可变引用（区别于同步接口，同步接口是不可变引用）。 在`listTable()`返回之后（这时整个列举表操作并没有完成），传入的`ListTableRequest`对象可能会被改变或者析构，从而引入一些难以调查的微妙错误。为了避免这类问题，异步客户端会将传入的请求对象里的内容转移（并非复制）到内部保存起来。所以，调用了`listTable()`之后，传入的请求对象有可能被改变。

- 回调函数

  回调函数不能返回任何值，接收参数有以下三种：

  — 请求对象。其内容即用户调用`listTable()`时传入的请求对象。因为回调之后异步客户端也不再需要请求对象，于是以可变引用的方式还给用户的回调函数。这样用户可以将请求对象的内容转移出来。

— 包装在Optional内的错误对象。如果没有错误，则该对象的`present()`方法返回false。

— 响应对象。与请求对象类似，响应对象也是以可变引用的方式交给回调函数。如果有错误，响应对象一定是一个合法的对象（可以正常析构），但是其内容是未定义的。

说明：

- 异步客户端保证每个请求的回调一定会被调用正好一次。
- 理论上，回调函数有可能在`listTable()`返回之前被调用。

示例

```
void listTableCallback(
    ListTableRequest&,
    Optional<OTSError>& err,
    ListTableResponse& resp)
{
    if (err.present()) {
        // 处理错误
    } else {
        const IVector<string>& xs = resp.tables();
        for(int64_t i = 0; i < xs.size(); ++i) {
            cout << xs[i] << endl;
        }
    }
}
void listTable(AsyncClient& client)
{
    ListTableRequest req;
    client.listTable(req, listTableCallback);
}
```

7.10 日志

客户端默认提供一个日志记录器。如果您的应用有自己的日志记录器，为了便于管理日志，建议您使用自己的日志记录器。

日志记录器有以下四个元素，定义在`tablestore/util/logger.hpp`。

- Logger接口

  Logger负责将日志的内容组装成Record对象，并转交给Sinker去写出。同时客户端将Logger组织成树形的结构，其根是用户在ClientOptions中定义的日志记录器，负责请求的逻辑和负责网络的逻辑使用从这个根日志记录器派生出的不同的子日志记录器。

```
class Logger
{
public:
enum LogLevel
{
```

```
    kDebug,
    kInfo,
    kError,
};

virtual ~Logger() {}
virtual LogLevel level() const =0;
virtual void record(LogLevel, const std::string&) =0;
virtual Logger* spawn(const std::string& key) =0;
virtual Logger* spawn(const std::string& key, LogLevel) =0;
};
```

- `level()`返回Logger接受的日志等级，低于该等级的日志不会被传递给Sinker。
- `record()`接受一条日志及其等级，组成Record对象后交给Logger对应的Sinker。
- `spawn()`派生一个子日志记录器。

- Record接口

  Record对象用于Logger向Sinker传递日志内容。

  Record接口本身没有任何方法。具体的Record类提供怎样的方法由Logger与Sinker约定。

- Sinke接口

  Sinker负责将Record对象写出。

```
class Sinker
{
public:
virtual ~Sinker() {}
virtual void sink(Record*) =0;
virtual void flush() =0;
};
```

  - `sink()`，写出一条Record。可以只是写到某个缓存中。
  - `flush()`，刷缓存，确保每条日志都落地。

- SinkerCenter单例对象

  SinkerCenter持有所有的Sinker对象，并将它们和一些键关联起来。

```
class SinkerCenter
{
public:
virtual ~SinkerCenter() {}
static std::tr1::shared_ptr<SinkerCenter> singleton();

virtual Sinker* registerSinker(const std::string& key, Sinker*) =0;
virtual void flushAll() =0;
};
```

  - `singleton()`，获取SinkerCenter单例对象。
  - `registerSinker()`，在SinkerCenter中注册一个Sinker。

- `flushAll()`，将SinkerCenter中的所有Sinker都刷一遍。

7.11 错误处理

TableStore C++ SDK采用返回值的方式处理错误。可能发生错误的接口都会返回`Optional<OTSError>`对象。

- `Optional<T>`是在tablestore/util/optional.hpp中定义的一个模板类。可以把它视为只能存放至多一个`T`对象的箱子。箱子里有两种情况，以此来判断错误是否发生：
  - 如果有一个`T`对象，可以取出这个T对象来用。此时，`Optional<T>::present()`返回true，表示有错误发生。
  - 如果没有`T`对象，此时`Optional<T>::present()`返回false，表示没有错误。
- OTSError对象表示一个具体的错误。它有5个字段：
  - `httpStatus`和`errorCode`：HTTP返回码和错误码。除了错误码参考之外，以下为仅在客户端会发生的错误。

6	OTSCouldntResolveHost	无法解析域名。实例访问地址有错，或者网络不通。
7	OTSCouldntConnect	无法连接服务端。本地host文件配置错误。
28	OTSRequestTimeout	请求超时。
35	OTSSslHandshakeFail	HTTPS握手失败。没有安装本地的证书。
55	OTSWriteRequestFail	网络发送失败。网络中断。
56	OTSCorruptedResponse	响应不完整。
89	OTSNoAvailableConnection	没有可用的连接。通常发生在客户端刚刚构造时，或者因为并发的请求数超过了网络连接数。

  - `message`：错误的详细说明。
  - `requestId`：每个发送到服务端的请求都会由服务端分配一个编号。如果响应正常返回，响应对象里会有requestId。 如果服务端判断请求出错，那么错误对象中会带有requestId。 如果在请求发送之前，或者网络链路上出错，那么错误对象里不会有requestId。

- `traceId`：每个API调用都会由客户端分配traceId。不同的API调用分配的traceId不同。同一个API调用涉及多次重试的，traceId相同，但是requestId可能不同。

7.12 重试

常用重试策略

C++ SDK有以下几种常用的重试策略：

- 默认重试策略

  SDK中出现错误时会自动重试。最大重试间隔为10秒。

- 计数重试策略

  按用户指定的间隔重试，对于可安全重试的策略，最多重试用户指定的次数。

- 不重试

自定义重试策略

您可以通过修改`RetryStrategy`自定义重试策略。

```
class RetryStrategy
{
public:
    virtual ~RetryStrategy() {}

    virtual RetryStrategy* clone() const =0;
    virtual int64_t retries() const throw() =0;
    virtual bool shouldRetry(Action, const OTSError&) const =0;
    virtual util::Duration nextPause() =0;
};
```

- `clone()`，复制一个新的对象。必须和当前对象相同类型，并且重试次数等内部状态也完全一样。
- `retries()`，已重试次数。
- `shouldRetry()`，指定操作和错误，判断是否应该重试。

  为方便操作，表格存储提供了两个工具函数：

  - 第一个retriable将错误分成三类。
    - 重试绝对无害的，例如OTSTableNotReady。
    - 重试有害，或无意义的，例如各种参数错误。
    - 仅凭错误无法判断的，例如OTSRequestTimeout。

- 第二个retriable根据操作的幂等原则，结合操作和错误判断可否重试。即，RETRIABLE类的错误，和读操作的DEPENDS类错误，都判为可重试。

```
enum RetryCategory
{
  UNRETRIABLE,
  RETRIABLE,
  DEPENDS,
};
static RetryCategory retriable(const OTSError&);
static bool retriable(Action, const OTSError&);
```

`nextPause()`，如果可以重试，则指定下次重试的间隔时间。

8 PHP SDK

8.1 前言

简介

本文档主要介绍 Table Store 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。
- 如果您还没有创建 AccessKeyId 和 AccessKeySecret，请到阿里云 *Access Key* 的管理控制台创建 Access Key。

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*

版本迭代详情参考这里。

版本

当前最新版本：4.1.0

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

- 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Google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

环境准备

- 64位PHP 5.5+（必须）

  您可以通过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，而在32位PHP里面，只能用string表示64位的整型，所以暂不支持32位PHP。 由于Windows下面，PHP7之前的版本，不是真正的64位，如果要使用Windows，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） 您可以通过php -m命令查看cURL扩展是否已经安装好。

  

  说明：

  - 在Ubuntu系统中，您可以使用apt-get包管理器安装PHP的cURL扩展 sudo apt-get install php-curl。
  - 在CentOS系统中，您可以使用yum包管理器安装PHP的cURL扩展 sudo yum install php-curl。

- OpenSSL 扩展（建议） 如果您需要使用https，需要安装OpenSSL PHP扩展。

安装方式

composer方式

composer方式安装SDK的步骤如下：

1. 在项目的根目录运行`composer require aliyun/aliyun-tablestore-sdk-php`，或者在您的composer.json中声明对阿里云 tablestore SDK for PHP的依赖：

```
"require": {
    "aliyun/aliyun-tablestore-sdk-php": "~4.0"
}
```

2. 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中国区的镜像源。方法是在命令行执行`composer config -g repositories.packagist composer http://packagist.phpcomposer.com`。

源码包

如果需要源码包，则通过下列方式可以下载：

- 在*GitHub*页面中选择相应版本并下载打包好的压缩文件。
- 在此下载源码包。

示例程序

Table Store PHP SDK 提供丰富的示例程序，方便用户参考或直接使用。您可以通过以下两种方式获取示例程序：

- 下载 Table Store PHP SDK 开发包后，解压后 examples 为示例程序。
- 访问 Table Store PHP SDK 的 GitHub 项目：*aliyun-tablestore-php-sdk*。

您可以通过以下步骤运行示例程序：

- 解压下载的SDK包。
- 修改examples目录中的ExampleConfig.php文件：

  ```
  EXAMPLE_END_POINT：是您从阿里云获得的AccessKeyId。
  EXAMPLE_ACCESS_KEY_ID：是您从阿里云获得的AccessKeySecret。
  EXAMPLE_ACCESS_KEY_SECRET：是您选定的tablestore数据中心访问域名，如 https
  ://sun.cn-hangzhou.ots.aliyuncs.com。
  EXAMPLE_INSTANCE_NAME：是您运行示例程序所使用的Instance。示例程序会在这个
  Instance中进行操作。
  ```

- 在examples目录中单独运行某个示例文件。

示例程序包含以下内容：

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
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

示例文件	示例内容
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
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 TableStore 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。

- 除了公网可以访问外，也支持私网地址。
- 登录表格存储控制台，进入实例详情页，实例访问地址即是该实例的 Endpoint。

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。

说明：

- 除了公网可以访问外，也支持私网地址。
- 登录表格存储控制台，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：

- 主帐号的 AccessKeyId 和 AccessKeySecret。创建步骤如下：
  1. 在阿里云官网注册阿里云帐号。
  2. 登录阿里云的管理控制台申请 *AccessKey*。
- 被授予访问表格存储权限的子帐号的 AccessKeyId 和 AccesskeySecret。创建步骤如下：
  1. 使用主帐号前往访问控制 *RAM*，创建一个新的子帐号或者使用已经存在的子帐号。
  2. 使用主帐号授予子帐号访问表格存储的权限。
  3. 子帐号被授权后，就可以使用自己的 AccessKeyId 和 AccessKeySecret 访问了。
- 临时访问的 STS token。获取步骤如下：

1. 应用的服务器通过访问 RAM/STS 服务，获取一个临时的 AccesskeyId、AccesskeySecret 和 token，发送给使用方。
2. 使用方使用上述临时密钥访问表格存储服务。

初始化对接步骤

在获取到 AccessKeyId 和 AccessKeySecret 等密钥之后，您可以按照下面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 OTSClient(array(
    'EndPoint' => "<your endpoint>",
    'AccessKeyID' => "<your access id>",
    'AccessKeySecret' => "<your access key>"
    'InstanceName' => "<your instance name>"
));
```

提示：

- 除了支持 AccessKeyId 和 AccessKeySecret 外，还支持 STS token 访问。
- 在创建 OTSClient 时，也支持通过 OTSClientConfig 参数设置一些参数，比如超时时间，最大连接数，最多重试次数等。

2. 配置 OTSClient。

如果您需要修改 OTSClient 的一些默认配置，请在构造 OTSClient 的时候传入对应参数，可配置代理、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，null可以关闭重试
ErrorLogHandler	Error级别日志处理函数，用来打印OTS服务端返回错误时的日志。	defaultOTSErrorLogHandler，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，null可以关闭

HTTPS

安装OpenSSL PHP扩展即可。

8.4 表操作

SDK 提供了 CreateTable、ListTable、DeleteTable、UpdateTable、DescribeTable 和 ComputeSplitsBySize 等表级别的操作接口。

创建表（CreateTable）

*API*说明

表格存储建表时需要指定表的结构信息（TableMeta）和配置信息（TableOptions），也可指定表的预留读/写吞吐量（ReservedThroughput）。

建表后服务端需要将表的分片加载到某个节点上，因此需要等待几秒钟才能对表进行读写，否则会抛出异常。注：本SDK遇到这种情况读写会自动重试。

接口

```
     /**
     * 创建表，并设定主键的个数、名称、顺序和类型，以及预留读写吞吐量，TTL，
stream选项。
     * API说明：https://help.aliyun.com/document_detail/27312.html
     * @api
     * @param [] $request 请求参数
     * @return [] 返回为空。CreateTable成功时不返回任何信息，这里返回一个空的
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
     * @throws OTSServerException 当OTS服务端返回错误时
     */
    public function createTable(array $request);
```

请求格式

```
$result = $client->createTable([
    'table_meta' => [                   // REQUIRED
        'table_name' => '<string>',
        'primary_key_schema' => [
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>],
            ['<string>', <PrimaryKeyType>, <PrimaryKeyOption>]
        ]
    ],
    'reserved_throughput' => [          // REQUIRED
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
```

```
    'table_options' => [                // REQUIRED
        '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

- table_meta TableMeta 包含表名和表的主键定义，(必须设置)。
  - table_name 表名。
  - primary_key_schema 表的主键定义。
    - 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如 PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请参见表格存储数据模型和查询操作。
    - 第一列主键列的值作为分片键。分片键相同的数据肯定会在同一个分片内，所以相同分片键下最好不要超过 10 G 以上数据，否则会导致单分片过大（无法分裂）。另一方面，数据和读/写访问最好在不同的分片键上均匀分布，有利于负载均衡，在设计表结构时需要注意这一点。
    - 用户建表时只需要定义表的主键，属性列不需要定义。每行的数据列都可以不同，属性列的列名在写入时指定。因此，表格存储非常适合存储半结构化的数据，在业务发展过程中可以动态添加新的数据列。
    - 每一项的顺序是 主键名，主键类型PrimaryKeyType，主键设置（可选）PrimaryKey Option。
    - 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，详见主键列自增)，分别用PrimaryKeyOptionConst::CONST_PK_AUTO_INCR表示。
- reserved_throughput 表的预留读/写吞吐量配置，与计费相关，(必须设置)。
  - 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，(必须设置)。
  - 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，请参见数据模型概念。
    - 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。比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。
    - 表格存储的新版 API 支持多版本的数据模型，数据模型概念一章对此已经做了一些介绍。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

```
    []
```

结果格式说明

返回为空，出错会抛出异常。

示例

创建一个有 3 个主键列，预留读/写吞吐量为 (0,0) 的表, 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，这个同时也是分片列
        //第二个PK列为字符串，名称是pk1
        //第三个PK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
            ]
        ]);
```

列出表名称（**ListTable**）

获取当前实例下已创建的所有表的表名。

接口

```
    /**
     * 获取该实例下所有的表名。
     * API说明：https://help.aliyun.com/document_detail/27313.html
```

```
     * @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
     * @throws OTSServerException 当OTS服务端返回错误时
     */
    public function listTable(array $request);
```

请求格式

```
$result = $client->listTable([]);
```

请求格式说明

目前请求为空。

结果格式

```
    [
        '<string>',
        '<string>',
        '<string>'
    ]
```

结果格式说明

结果是一个string的list. 每一项是一个表名。

示例

获取实例下的所有表名。

```
 $result = $otsClient->listTable([]);
```

更新表（**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说明：https://help.aliyun.com/document_detail/27315.html
     * @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，请参见数据模型概念。
- 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。比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。

- 表格存储的新版 API 支持多版本的数据模型，数据模型概念一章对此已经做了一些介绍。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
        ]
    ]);
```

查询表描述信息（DescribeTable）

DescribeTable 接口可以查询表的结构信息（TableMeta）、配置信息（TableOptions）、预留读/写吞吐量的情况（ReservedThroughputDetails），stream设置信息（StreamDetails)。TableMeta 和 TableOptions 在“建表”一节已经有过介绍，ReservedThroughputDetails 除了包含表的预留吞吐量的值外，还包括最近一次上调或者下调的时间。 StreamDetails 包含了stream的详细信息。

接口

```
    /**
     * 获取一个表的信息，包括主键设计以及预留读写吞吐量信息。
     * API说明：https://help.aliyun.com/document_detail/27316.html
     * @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 describeTable(array $request);
```

请求格式

```
$result = $client->describeTable([
     'table_name' => '<string>', // REQUIRED
```

```
]);
```

请求格式说明

- table_name 表名。

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>
    ]
]
```

结果格式说明

- table_meta TableMeta 包含表名和表的主键定义，和创建表时的定义是一样的。
- 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。
  - 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，请参见数据模型概念。
    - 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。比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。
    - 表格存储的新版 API 支持多版本的数据模型，数据模型概念一章对此已经做了一些介绍。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_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

获取表的描述信息。

```
    $result = $client->describeTable([
         'table_name' => 'mySampleTable',
    ]);
    var_dump($result);
```

删除表（DeleteTable）

删除本实例下指定的表。

接口

```
     /**
     * 根据表名删除一个表。
     * API说明：https://help.aliyun.com/document_detail/27314.html
     * @api
     * @param [] $request 请求参数
     * @return [] 返回为空。DeleteTable成功时不返回任何信息，这里返回一个空的
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
     * @throws OTSServerException 当OTS服务端返回错误时
     */
    public function deleteTable(array $request);
```

请求格式

```
$result = $client->deleteTable([
     'table_name' => '<string>', // REQUIRED
]);
```

请求格式说明

table_name 表名。

结果格式

```
    []
```

结果格式说明

返回为空，出错会抛出异常。

示例

删除表。

```
$result = $otsClient->deleteTable([
    'table_name' => 'MyTable'
```

```
]);
```

指定大小计算分片(ComputeSplitsBySize)

将全表数据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。

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说明：https://help.aliyun.com/document_detail/53813.html
     * @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 computeSplitPointsBySize(array $request)
```

请求格式

```
$result = $client->ComputeSplitsBySize([
    'table_name' => '<string>', // REQUIRED
    'split_size' => <integer>   // REQUIRED
]);
```

请求格式说明

- table_name 表名。
- split_size 每个分片的近似大小，以百兆为单位。

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>'
        ],
        // ...
    ]
]
```

结果格式说明

- consumed 本次操作消耗服务能力单元的值。
  - capacity_unit 使用的读写单元量
    - read 读吞吐量
    - write 写吞吐量
- primary_key_schema 表的主键定义，与建表时给出的 Schema 相同。
- splits 分片之间的分割点。
  - lower_bound 主键的区间最小值。注：可以传递给getRange使用。
    - 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。格式同上。
  - location 分割点所在机器的提示。可以为空

示例

指定大小计算分片

```
    $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']);
```

```
    }
```

8.5 单行操作

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

插入一行数据（PutRow）

*API*说明

PutRow 接口用于插入一行数据。若该行不存在则插入，如果该行已经存在则覆盖（即原行的所有列以及所有版本的列值都被删除）。

PutRow 写入时支持条件更新（Conditional Update），可以设置原行的存在性条件或者原行中某列的列值条件，条件更新一章对此有专门介绍。

接口

```
    /**
     * 写入一行数据。如果该行已经存在，则覆盖原有数据。返回该操作消耗的CU。
     * API说明：https://help.aliyun.com/document_detail/27306.html
     * @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 putRow(array $request);
```

请求格式

```
$result = $client->putRow([
    'table_name' => '<string>', // REQUIRED
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                                  // REQUIRED
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => [          // REQUIRED
            ['<string>', <ColumnValue>],
            ['<string>', <ColumnValue>, <ColumnType>],
            ['<string>', <ColumnValue>, <ColumnType>, <integer>]
    ],
    'return_content' => [
        'return_type' => <ReturnType>
    ]
]);
```

请求格式说明

- table_name 表名。必须设置。
- condition 。参加条件更新。
  - row_existence 行存在性条件
  - column_condition 列条件
- primary_key 行的主键值。(必须设置)。
  - 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如 PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请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、主键类型PrimaryKeyType（可选）。
  - PrimaryKeyValue可以是整数和字符串，对于自增主键列，可以设置为null。
  - PrimaryKeyType可以是INTEGER、STRING（UTF-8编码字符串）、BINARY、PK_AUTO_INCR(自增列，详见主键列自增)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。
- attribute_columns 行的属性值(必须设置)。
  - 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，或者设为null。
  - 时间戳是64bit整数，用来表示属性的多个不同的版本，可以设置，也可以不设(由服务器指定)
- return_content 表示返回类型。
  - return_type 目前只需要设置这个。
    - ReturnTypeConst::CONST_PK 表示返回主键值（主要用于主键列自增场景）

结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

结果格式说明

- consumed 本次操作消耗服务能力单元的值。
  - capacity_unit 使用的读写单元量
    - read 读吞吐量
    - write 写吞吐量
- primary_key 主键的值，和请求一致。设置返回PK的时候会有值，主要用于主键列自增。
- attribute_columns 属性的值,和请求一致，目前为空

示例 **1**

写入 10 列属性列，每列写入 1 个版本，由服务端指定版本号（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, EXPECT_EXIST, EXPECT_NOT_EXIST
    'primary_key' => [ // 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，由客户端指定版本号（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, EXPECT_EXIST, EXPECT_NOT_EXIST
    'primary_key' => [ // 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。

```
$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_EXI
ST, // 设置期望不存在时写入
    'primary_key' => [ // 主键
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

示例 4

期望原行存在，且 Col0 的值大于 100 时写入。

```
$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_EXPE
CT_EXIST,// 设置期望存在时写入
        'column_condition' => [                    //条件更新，满足则更新
```

```
            'column_name' => 'Col0',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ]
    ,
    'primary_key' => [ // 主键
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'attribute_columns' => $attr
];
$response = $otsClient->putRow ($request);
```

读取一行数据（GetRow）

API说明

指定表名和一行的主键，读取的结果可能有两种：

- 若该行存在，则返回该行的各主键列以及属性列。
- 若该行不存在，则返回中不含有行，并且不会报错。

接口

```
    /**
     * 读取一行数据。
     * API说明：https://help.aliyun.com/document_detail/27305.html
     * @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 getRow(array $request);
```

请求格式

```
$result = $client->getRow([
    'table_name' => '<string>',                         // REQUIRED
    'primary_key' => [                                  // REQUIRED
        ['<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>
]);
```

请求格式说明

- table_name 表名。必须设置。
- primary_key 行的主键值。(必须设置)。
  - 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如 PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请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、主键类型PrimaryKeyType（可选）。
  - PrimaryKeyValue可以是整数和字符串。
  - 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，对于INTEGER和STRING，可以省略，其它类型不可省略。
- max_versions 最多读取多少个版本（常用）。
- time_range 要读取的版本号的范围。参见*TimeRange*
  - start_time 起始时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。若要查询一个范围，则指定start_time和end_time，前闭后开区间。
  - end_time 结束时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。
  - specific_time 特定的时间戳值，若要查询一个特定时间戳，则指定specific_time。specific_time和[start_time, end_time) 两个中设置一个即可。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。
- max_versions 与 time_range 必须至少设置一个。
  - 如果仅指定 max_versions，则返回所有版本里从新到旧至多指定数量个数据。
  - 如果仅指定 time_range，则返回该范围内所有数据。
  - 如果同时指定 max_versions 和 time_range，则返回版本范围内从新到旧至多指定数量个数据。
- columns_to_get 要读取的列的集合（常用），若不设置，则读取所有列。

- start_column 指定读取时的起始列，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。例子：如果一张表有”a”，”b”，”c”三列，读取时指定start_column为“b”，则会从”b”列开始读，返回”b”，”c”两列。参见宽行读取
- end_column 指定读取时的结束列，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。例子：如果一张表有”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 宽行读取时指定下一次读取的起始位置，暂不可用
- column_filter 过滤条件，满足条件才会返回。和condition里面的column_condition类似。参见过滤器。

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>'
]
```

结果格式说明

- consumed 本次操作消耗服务能力单元的值。
  - capacity_unit 使用的读写单元量
    - read 读吞吐量
    - write 写吞吐量
- primary_key 主键的值，和请求一致。设置返回PK的时候会有值，主要用于主键列自增。
- attribute_columns 属性的值
  - 每一项的顺序是 属性名、属性值ColumnValue、属性类型ColumnType、时间戳。

  - 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。
  - 时间戳是64bit整数，用来表示属性的多个不同的版本
  - 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 request 中的 columns_to_get 一致
- next_token 表示宽行读取时下一次读取的位置，编码的二进制。（暂不可用）
- 如果该行不存在，则 primary_key 和 attribute_columns 均为空列表[].

示例 **1**

读取一行，设置读取最新版本，设置 columns_to_get。

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ // 主键
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'max_versions' => 1,                        // 设置读取最新版本
    'columns_to_get' => ['Col0']                // 设置读取某些列
];
$response = $otsClient->getRow ($request);
```

提示：

- 查询一行数据时，默认返回这一行所有列的数据，如果想只返回特定列，可以通过 columns_to_get 接口限制，如果将 col0 和 col1 加入到 columns_to_get 中，则只返回 col0 和 col1 的值。
- 查询时也支持按条件过滤，比如当 col0 的值大于 24 时才返回结果。
- 当同时使用 columns_to_get 和 column_filter 时，顺序是 columns_to_get 先生效，然后再去返回的列中进行过滤。
- 某列不存在时的行为，可以通过 pass_if_missing 控制。默认值是true，表示列不存在时该条件成立。

示例 **2**

设置过滤器。

```
$request = [
    'table_name' => 'MyTable',
    'primary_key' => [ // 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的值为0
时返回该行.
        'column_name' => 'Col0',
        'value' => 0,
        'comparator' => ComparatorTypeConst::CONST_EQUAL,
        'pass_if_missing' => false          // 如果不存在Col0这一列, 也不
返回.
    ]
];
$response = $otsClient->getRow ($request);
```

更新一行数据（UpdateRow）

*API*说明

UpdateRow 接口用于更新一行数据，如果原行不存在，会新写入一行。

更新操作包括写入某列、删除某列和删除某列的某一版本。

更新操作有以下四种情况：

- 不指定版本写入一个列值，表格存储服务端会自动补上一个版本号，保证此种情况下版本号的递增。
- 指定版本写入一个列值，若该列原先没有该版本列值，则插入数据，否则覆盖原值。
- 删除指定版本的列值。
- 删除整个列的所有版本列值。

UpdateRow 接口支持条件更新（Conditional Update），可以设置原行的存在性条件或者原行中某列的列值条件，条件更新一章对此有专门介绍。

接口

```
    /**
     * 更新一行数据。
     * API说明：https://help.aliyun.com/document_detail/27307.html
     * @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 updateRow(array $request);
```

请求格式

```
$result = $client->updateRow([
    'table_name' => '<string>', // REQUIRED
```

```
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                                  // REQUIRED
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'update_of_attribute_columns' => [          // REQUIRED
        '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>
    ]
]);
```

请求格式说明

- table_name 表名。必须设置。
- condition 条件，满足条件才会生效。参见条件更新。
  - row_existence 行存在性条件
  - column_condition 列条件
- primary_key 行的主键值。(必须设置)。
  - 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如 PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请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、主键类型PrimaryKeyType（可选）。
  - PrimaryKeyValue可以是整数和字符串。
  - 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，对于INTEGER和STRING，可以省略，其它类型不可省略。

- update_of_attribute_columns 行的属性的修改(必须设置，如果没有对应的操作，也可以没有其中的某项)，有三种不同的更新操作，分别如下。
  - PUT 格式和PutRow的attribute_columns一致, 语意为如果该列不存在，则新增一列；如果该列存在，则覆盖该列，多版本则增加一个版本。
    - 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，或者设为null。
    - 时间戳是64bit整数，用来表示属性的多个不同的版本，可以设置，也可以不设(由服务器指定)
  - DELETE 需要指定timestamp。语意为删除该列特定版本的数据。
    - 每一项的顺序是 属性名，时间戳。
    - 时间戳是64bit整数，表示某个特定版本的属性。
  - DELETE_ALL 语意为删除该列所有版本的数据。注意：删除本行的全部属性列不等同于删除本行，若想删除本行，请使用 DeleteRow 操作。
    - 只需要指定属性名即可。
- return_content 需要返回的内容。
  - return_type 目前只需要设置这个。
    - ReturnTypeConst::CONST_PK 表示返回主键值（主要用于主键列自增场景）

结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
```

```
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

结果格式说明

- consumed 本次操作消耗服务能力单元的值。
  - capacity_unit 使用的读写单元量
    - read 读吞吐量
    - write 写吞吐量
- primary_key 主键的值，和请求一致。设置返回PK的时候会有值，主要用于主键列自增。
- attribute_columns 属性的值,和请求一致，目前为空

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' => [ // 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);
```

提示：

- 更新一行数据也支持条件语句。

示例 **2**

设置更新的条件。

```
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => [ // 主键
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPE
CT_EXIST, // 期望原行存在
        'column_filter' => [
       // Col0的值大于100时更新
            '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**）

*API*说明

DeleteRow 接口用于删除一行。无论该行存在与否都不会报错。

DeleteRow 接口支持条件更新（Conditional Update），可以设置原行的存在性条件或者原行中某列的列值条件，[条件更新]一章对此有专门介绍。

接口

```
    /**
     * 删除一行数据。
     * API说明：https://help.aliyun.com/document_detail/27308.html
     * @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 deleteRow(array $request);
```

请求格式

```
$result = $client->deleteRow([
    'table_name' => '<string>', // REQUIRED
    'condition' => [
        'row_existence' => <RowExistence>,
        'column_condition' => <ColumnCondition>
    ],
    'primary_key' => [                                  // REQUIRED
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'return_content' => [
        'return_type' => <ReturnType>
    ]
]);
```

请求格式说明

- table_name 表名。必须设置。
- condition 条件，满足条件才会生效。参见条件更新。
  - row_existence 行存在性条件
  - column_condition 列条件
- primary_key 行的主键值。(必须设置)。
  - 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如 PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请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、主键类型PrimaryKeyType（可选）。
  - PrimaryKeyValue可以是整数和字符串。
  - 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，对于INTEGER和STRING，可以省略，其它类型不可省略。
- return_content 表示返回类型。
  - return_type 目前只需要设置这个。
    - ReturnTypeConst::CONST_PK 表示返回主键值（主要用于主键列自增场景）

结果格式

```
[
    'consumed' => [
        'capacity_unit' => [
            'read' => <integer>,
            'write' => <integer>
        ]
    ],
    'primary_key' => [
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'attribute_columns' => []
]
```

结果格式说明

- consumed 本次操作消耗服务能力单元的值。
  - capacity_unit 使用的读写单元量
    - read 读吞吐量
    - write 写吞吐量
- primary_key 主键的值，和请求一致。设置返回PK的时候会有值，主要用于主键列自增。
- attribute_columns 属性的值,和请求一致，目前为空

示例1

删除一行数据。

```
$request = [
    'table_name' => 'MyTable',
    'condition' => RowExistenceExpectationConst::CONST_IGNORE,
    'primary_key' => [ // 主键
        ['PK0', 123],
        ['PK1', 'abc']
    ],
    'return_content' => [
        'return_type' => ReturnTypeConst::CONST_PK     // 列自增需要主键返回需要设置return_type
    ]
];
$response = $otsClient->deleteRow($request);
```

示例2

设置删除条件。

```
$request = [
    'table_name' => 'MyTable',
```

```
    'condition' => [
        'row_existence' => RowExistenceExpectationConst::CONST_EXPE
CT_EXIST, //期望原行存在
        'column_filter' => [          // Col0的值大于100时删除
            'column_name' => 'Col0',
            'value' => 100,
            'comparator' => ComparatorTypeConst::CONST_GREATER_THAN
        ],
    ],
    'primary_key' => [ // 主键
        ['PK0', 123],
        ['PK1', 'abc']
    ]
];
$response = $otsClient->deleteRow ($request);
```

8.6 多行操作

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

批量读（BatchGetRow）

*API*说明

批量读取一个或多个表中的若干行数据。

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。

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。

参数与 GetRow 接口参数一致。需要注意的是，批量读取的所有行采用相同的参数条件，比如 columns_to_get=[colA], 则要读取的所有行都只读取 colA 这一列。

与 BatchWriteRow 接口类似，使用 BatchGetRow 接口时也需要检查返回值。 存在部分行失败，而不抛出异常的情况，此时失败行的信息在 BatchGetRowResponse 中。

接口

```
    /**
     * 读取指定的多行数据。
     * API说明：https://help.aliyun.com/document_detail/27310.html
     * 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。请参见样例：处理BatchGetRow的返回。
     * @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);
```

请求格式

```
$result = $client->batchGetRow([
    'tables' => [                                           //
REQUIRED
        [
            'table_name' => '<string>',                     //
REQUIRED
            'primary_keys' => [                             //
REQUIRED
                [
                    ['<string>', <PrimaryKeyValue>],
                    ['<string>', <PrimaryKeyValue>],
                    ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
                ],
                // other primary keys
            ]
            '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>
        ],
        // other tables.
    ]
]);
```

请求格式说明

- 和GetRow的区别
  - primary_key变为primary_keys，可以一次读取多行。
  - 增加了表的层级结构，可以一次读取多表。
- tables 以表为单位组织，后续为各个表的操作，指定了需要读取的行信息。
  - table_name 表名。必须设置。
  - primary_keys 行的主键值列表。(必须设置)。列表的每一项都是一个primary_key, 以下是每个primary_key的结构：

- 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请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、主键类型PrimaryKeyType（可选）。
- PrimaryKeyValue可以是整数和字符串。
- 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，对于INTEGER和STRING，可以省略，其它类型不可省略。

— max_versions 最多读取多少个版本（常用）。

— time_range 要读取的版本号的范围。参见*TimeRange* 。

- start_time 起始时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。若要查询一个范围，则指定start_time和end_time，前闭后开区间。
- end_time 结束时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。
- specific_time 特定的时间戳值，若要查询一个特定时间戳，则指定specific_time。specific_time和[start_time, end_time) 两个中设置一个即可。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。

— max_versions 与 time_range 必须至少设置一个。

- 如果仅指定 max_versions，则返回所有版本里从新到旧至多指定数量个数据。
- 如果仅指定 time_range，则返回该范围内所有数据。
- 如果同时指定 max_versions 和 time_range，则返回版本范围内从新到旧至多指定数量个数据。

— columns_to_get 要读取的列的集合（常用），若不设置，则读取所有列。

— start_column 指定读取时的起始列，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。例子：如果一张表有”a”，”b”，”c”三列，读取时指定start_column为“b”，则会从”b”列开始读，返回”b”，”c”两列。参见宽行读取。

— end_column 指定读取时的结束列，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。例子：如果一张表有”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 宽行读取时指定下一次读取的起始位置，暂不可用。

- column_filter 过滤条件，满足条件才会返回。和condition里面的column_condition类似。参见过滤器。

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>, <PrimaryKey
Type>]
                    ],
                    'attribute_columns' => [
                            ['<string>', <ColumnValue>, <ColumnType>,
 <integer>]
                            ['<string>', <ColumnValue>, <ColumnType>,
 <integer>]
                            ['<string>', <ColumnValue>, <ColumnType>,
 <integer>]
                    ],
                    'next_token' => '<string>'
                ],
                // other rows
            ]
        ],
        // other tables.
    ]
]
```

结果格式说明

- *API*说明（介绍了整体失败和部分失败的几种情况，务必查阅。）
- MaxCompute
- tables 以table为单位组织，和请求一一对应。
  - table_name 该表的表名。
  - is_ok 该行操作是否成功。若为 true，则该行读取成功，error 无效；若为 false，则该行读取失败，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 当前单行操作的错误码，具体含义可参考错误码。
  - message 当前单行操作的错误信息，具体含义可参考错误码。
- consumed 本次操作消耗服务能力单元的值。
  - capacity_unit 使用的读写单元量：
    - read 读吞吐量
    - write 写吞吐量
- primary_key 主键的值，和请求一致。
- attribute_columns 属性的值：
  - 每一项的顺序是 属性名、属性值ColumnValue、属性类型ColumnType、时间戳。
  - 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。
  - 时间戳是64bit整数，用来表示属性的多个不同的版本。
  - 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 request 中的 columns_to_get 一致。
- next_token 宽行读取时，下一次读取的起始位置，暂不可用。

示例

批量一次读 30 行。

```
// 从3张表中读取数据， 每张表读取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);

    // 处理返回的每个表
    foreach ($response['tables'] as $tableData) {
      print "Handling table {$tableData['table_name']} ...\n";

      // 处理这个表下的每行数据
      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";
        }
      }
    }
```

更详细的可以参考：

示例文件	示例内容
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

批量写（**BatchWriteRow**）

*API*说明

批量插入、修改或删除一个或多个表中的若干行数据。

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。

BatchWriteRow 接口可以在一次请求中进行批量的写入操作，写入操作包括 PutRow、UpdateRow 和 DeleteRow，也支持一次对多张表进行写入。

构造单个操作的过程与使用 PutRow 接口、UpdateRow 接口和 DeleteRow 接口时相同，也支持设置更新条件。

调用 BatchWriteRow 接口时，需要特别注意的是检查返回值。因为是批量写入，可能存在部分行成功部分行失败的情况，此时失败行的 Index 及错误信息在返回的 BatchWriteRowResponse 中，而并不抛出异常。 若不检查，可能会忽略掉部分操作的失败。 另一方面，BatchWriteRow 接口也是可能抛出异常的。比如，服务端检查到某些操作出现参数错误，可能会抛出参数错误的异常，在抛出异常的情况下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。
     * API说明：https://help.aliyun.com/document_detail/27311.html
     * 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。请参见样例：处理BatchWriteRow的返回。
     * @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);
```

请求格式

```
$result = $client->batchWriteRow([
    'tables' => [                                                   //
REQUIRED
        [
            'table_name' => '<string>',                             //
REQUIRED
            'operation_type' => <OperationType>,
            'condition' => [
                'row_existence' => <RowExistence>,
                'column_condition' => <ColumnCondition>
            ],
            'primary_key' => [                                      //
REQUIRED
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>],
                ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
            ],
            'attribute_columns' => [                               //
REQUIRED in PUT
                    ['<string>', <ColumnValue>],
                    ['<string>', <ColumnValue>, <ColumnType>],
                    ['<string>', <ColumnValue>, <ColumnType>, <integer
>]
            ],
            'update_of_attribute_columns' => [                      //
REQUIRED in UPDATE
```

```
                '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>
            ]
        ],
        // other tables.
    ]
]);
```

请求格式说明

- 本操作是PutRow, UpdateRow, DeleteRow的组合。
  - 增加了operation_type来区分操作类型。
  - 增加了表的层级结构，可以一次处理多个表。
- tables 以表为单位组织，后续为各个表的操作，指定了需要写入/修改/删除的行信息。
  - table_name 表名。必须设置。
  - condition 。参见条件更新。
    - row_existence 行存在性条件
    - column_condition 列条件
  - operation_type 操作类型。可以为PUT(OperationTypeConst::CONST_PUT)，UPDATE(OperationTypeConst::CONST_UPDATE)，DELETE(OperationTypeConst::CONST_DELETE)
    - PUT. primary_key 和 attribute_columns有效。
    - UPDATE. primary_key 和 update_of_attribute_columns有效。
    - DELETE. primary_key有效。
  - return_content 表示返回类型。

- return_type 目前只需要设置这个。
  - ReturnTypeConst::CONST_PK 表示返回主键值（主要用于主键列自增场景）。

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>, <PrimaryKey
Type>]
                      ],
                      'attribute_columns' => []
                  ],
                  // other rows
              ]
          ],
          // other tables.
      ]
  ]
```

结果格式说明

- *API*说明（介绍了整体失败和部分失败的几种情况，务必查阅。）
- tables 以table为单位组织，和请求一一对应。
  - table_name 该表的表名。
  - is_ok 该行操作是否成功。若为 true，则该行写入成功，error 无效；若为 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 当前单行操作的错误码，具体含义可参考错误码。
    - message 当前单行操作的错误信息，具体含义可参考错误码。
  - consumed 本次操作消耗服务能力单元的值。

- capacity_unit 使用的读写单元量
  - read 读吞吐量
  - write 写吞吐量

- primary_key 主键的值，和请求一致。设置返回PK的时候会有值，主要用于主键列自增。
- attribute_columns 属性的值, 目前为空。

示例

批量导入 30 行数据。

```
    // 向3张表中插入数据， 每张表插入10行。
    $tables = array();
    for($i = 0; $i < 3; $i++) {
        $rows = array();
        for($j = 0; $j < 10; $j++) {
            $rows[] = [
                'operation_type' => OperationTypeConst::CONST_PUT,
        //操作是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);
    // 处理返回的每个表
    foreach ($response['tables'] as $tableData) {
      print "Handling table {$tableData['table_name']} ...\n";

      // 处理这个表下的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";
        }
      }
    }
```

更详细的可以参考：

示例文件	示例内容
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
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

范围读（GetRange）

*API*说明

读取指定主键范围内的数据。

范围读取接口用于读取一个范围内的数据。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 接口支持按照给定范围正序读取和反序读取，可以限定要读取的行数。如果范围较大，已经扫描的行数或者数据量超过一定限制，会停止继续扫描，返回已经获取的行和下一个主键的位置。用户可以根据返回的下一个主键位置，继续发起请求，获取范围内剩余的行。

接口

```
    /**
     * 范围读取起始主键和结束主键之间的数据。
     * 请注意，这个范围有可能会被服务端截断，你需要判断返回中的 next_start
_primary_key 来决定是否继续调用 GetRange。
     * 你可以指定最多读取多少行。
     * 在指定开始主键和结束主键时，你可以用 INF_MIN 和 INF_MAX 来代表最大值和最
小值，详见下面的代码样例。
     * API说明：https://help.aliyun.com/document_detail/27309.html
     * @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);
```

请求格式

```
$result = $client->getRange([
    'table_name' => '<string>',
  // REQUIRED
    'inclusive_start_primary_key' => [
  // REQUIRED
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'exclusive_end_primary_key' => [
  // REQUIRED
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>],
        ['<string>', <PrimaryKeyValue>, <PrimaryKeyType>]
    ],
    'direction' => <Direction>,
  // REQUIRED
    '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>
]);
```

请求格式说明

- 和GetRow的区别
  - primary_key 变成 inclusive_start_primary_key，exclusive_end_primary_key，前闭后开区间。
  - 增加 direction 表示方向。
  - 增加 limit 限制返回行数。
- table_name 表名。必须设置。

- inclusive_start_primary_key 本次范围读取的起始主键，若该行存在，则响应中一定会包含此行。(必须设置)。
  - 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如 PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请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、主键类型PrimaryKeyType（可选）。
  - PrimaryKeyValue可以是整数和字符串。
  - 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，对于INTEGER和STRING，可以省略，其它类型不可省略。
  - PrimaryKeyType有两种特定类型，INF_MIN, INF_MAX。类型为 INF_MIN 的 Column 永远小于其它 Column；类型为 INF_MAX 的 Column 永远大于其它 Column。
  - 分别用PrimaryKeyTypeConst::CONST_INF_MIN，PrimaryKeyTypeConst::CONST_INF_MAX表示。对应的primaryKeyValue可以设置为null。
- exclusive_end_primary_key 本次范围读取的终止主键，无论该行是否存在，响应中都不会包含此行。(必须设置)。
  - 表的主键可包含多个主键列。主键列是有顺序的，与用户添加的顺序相同。比如 PRIMARY KEY (A, B, C) 与 PRIMARY KEY (A, C, B) 是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序，具体请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、主键类型PrimaryKeyType（可选）。
  - PrimaryKeyValue可以是整数和字符串。
  - 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，对于INTEGER和STRING，可以省略，其它类型不可省略。
  - PrimaryKeyType有两种特定类型，INF_MIN, INF_MAX。类型为 INF_MIN 的 Column 永远小于其它 Column；类型为 INF_MAX 的 Column 永远大于其它 Column。
  - 分别用PrimaryKeyTypeConst::CONST_INF_MIN，PrimaryKeyTypeConst::CONST_INF_MAX表示。对应的primaryKeyValue可以设置为null.

- direction 本次查询的顺序(必须设置)。若为正序(DirectionConst::CONST_FORWARD)，则 inclusive_start_primary 应小于 exclusive_end_primary，响应中各行按照主键由小到大的顺序进行排列；若为逆序(DirectionConst::CONST_BACKWARD)，则 inclusive_start_primary 应大于 exclusive_end_primary，响应中各行按照主键由大到小的顺序进行排列。
- limit 本次读取最多返回的行数。
  - 若查询到的行数超过此值，则通过响应中包含的断点记录本次读取到的位置，以便下一次读取。此值必须大于 0。
  - 无论是否设置此值，表格存储最多返回行数为 5000 且总数据大小不超过 4 M 。
- max_versions 最多读取多少个版本（常用）。
- time_range 要读取的版本号的范围。参见*TimeRange* 。
  - start_time 起始时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。若要查询一个范围，则指定start_time和end_time，前闭后开区间。
  - end_time 结束时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。
  - specific_time 特定的时间戳值，若要查询一个特定时间戳，则指定specific_time。specific_time和[start_time, end_time) 两个中设置一个即可。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。
- max_versions 与 time_range 必须至少设置一个。
  - 如果仅指定 max_versions，则返回所有版本里从新到旧至多指定数量个数据。
  - 如果仅指定 time_range，则返回该范围内所有数据。
  - 如果同时指定 max_versions 和 time_range，则返回版本范围内从新到旧至多指定数量个数据。
- columns_to_get 要读取的列的集合（常用），若不设置，则读取所有列。
- start_column 指定读取时的起始列，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。例子：如果一张表有”a”，”b”，”c”三列，读取时指定start_column为“b”，则会从”b”列开始读，返回”b”，”c”两列。参见宽行读取。
- end_column 指定读取时的结束列，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。例子：如果一张表有”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 宽行读取时指定下一次读取的起始位置，暂不可用

- column_filter 过滤条件，满足条件才会返回。和condition里面的column_condition类似。参见过滤器。

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
>]
            ]
        ],
        // other rows
    ]
]
```

结果格式说明

- *API*说明（详细解说，务必查阅）
- consumed 本次操作消耗服务能力单元的值。
  - capacity_unit 使用的读写单元量：
    - read 读吞吐量
    - write 写吞吐量
- rows
  - 读取到的所有数据，若请求中 direction 为 FORWARD，则所有行按照主键由小到大进行排序；若请求中 direction 为 BACKWARD，则所有行按照主键由大到小进行排序。

- 其中每行的 attribute_columns 只包含在 columns_to_get 中指定的列，其顺序不保证与 request 中的 columns_to_get 一致。
- 如果请求中指定的 columns_to_get 不含有任何主键列，那么其主键在查询范围内。但没有任何一个属性列在 columns_to_get 中的行将不会出现在响应消息里。
- primary_key 主键的值，和请求一致。设置返回PK的时候会有值，主要用于主键列自增。
- attribute_columns 属性的值：
  - 每一项的顺序是 属性名、属性值ColumnValue、属性类型ColumnType、时间戳。
  - 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。
  - 时间戳是64bit整数，用来表示属性的多个不同的版本。
  - 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。

- next_start_primary_key 本次 GetRange 操作的断点信息。
  - 格式和primary_key一样。
  - 若为空，则本次 GetRange 的响应消息中已包含了请求范围内的所有数据。
  - 若不为空，则表示本次 GetRange 的响应消息中只包含了 [inclusive_start_primary_key, next_start_primary_key) 间的数据，若需要剩下的数据，需要将 next_start_primary_key 作为 inclusive_start_primary_key，原始请求中的 exclusive_end_primary_key 作为 exclusive_end_primary_key 继续执行 GetRange 操作。

示例

按照范围读取。

```
    // 这里查找uid从1-4（左闭右开）的数据
    // 范围的边界需要提供完整的PK，若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或者无穷小
    $startPK = [
        ['PK0', 1],
        ['PK1', null, PrimaryKeyTypeConst::CONST_INF_MIN]  // 'INF_MIN' 用来表示最小值
    ];
    // 范围的边界需要提供完整的PK，若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或者无穷小
    $endPK = [
        ['PK0', 4],
        ['PK1', null, PrimaryKeyTypeConst::CONST_INF_MAX]  // 'INF_MAX' 用来表示最大值
```

```
    ];
    $request = [
        'table_name' => 'SampleTable',
        'max_versions' => 1,                                // 设置读取最新版本
        'direction' => DirectionConst::CONST_FORWARD, // 前向查询
        'inclusive_start_primary_key' => $startPK,     // 开始主键
        'exclusive_end_primary_key' => $endPK,         // 结束主键
        'limit' => 10                                       // 最多返回10行
    ];
    $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
    }
```

更详细的可以参考：

示例文件	示例内容
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

8.7 条件更新

条件更新功能是指只有在满足条件时才对表中的数据进行更改，当不满足条件时更新失败，可用于PutRow、UpdateRow、DeleteRow 和 BatchWriteRow 的condition 中。

判断条件包括行存在性条件和列条件。

- 行存在性条件：分为 IGNORE、 EXPECT_EXIST 和 EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。可以分别设置为RowExistenceExpectationConst::CONST_IGNORE、RowExistenceExpectationConst::CONST_EXPECT_EXIST和RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST。 在对表进行更改操作时，会首先检查行存在性条件。若不满足，则更改失败，对用户抛错。
- 列条件：目前支持两种，SingleColumnValueCondition 和 CompositeColumnValueCondition。是基于某一列或者某些列的列值进行条件判断，与过滤器 Filter 中的条件非常类似。

基于条件更新可以实现乐观锁的功能，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”，更新该行同时使“列 A = 2”。若更新失败，代表有其他客户端已经成功更新了该行。

格式

```
    'condition' => [
        'row_existence' => <RowExistenceExpectation>
        'column_condition' => <ColumnCondition>
    ]
```

其中 SingleColumnValueCondition 结构如下

```
    [
        'column_name' => '<string>',
        'value' => <ColumnValue>,
        'comparator' => <ComparatorType>
        'pass_if_missing' => true || false
        'latest_version_only' => true || false
    ]
```

CompositeColumnValueCondition结构如下

```
    [
        'logical_operator' => <LogicalOperator>
        'sub_conditions' => [
            <ColumnCondition>,
            <ColumnCondition>,
            <ColumnCondition>,
            // other conditions
        ]
    ]
```

或者当只有行存在性条件时（大部分情况是这样），可以简写为：

```
    'condition' => <RowExistenceExpectation>
```

格式说明

- row_existence 行存在性条件
  - 分为 IGNORE、 EXPECT_EXIST 和 EXPECT_NOT_EXIST，分别代表忽略、期望存在和期望不存在。
  - 可以分别设置为RowExistenceExpectationConst::CONST_IGNORE、RowExistenceExpectationConst::CONST_EXPECT_EXIST和RowExistenceExpectationConst::CONST_EXPECT_NOT_EXIST。
  - 在对表进行更改操作时，会首先检查行存在性条件。若不满足，则更改失败，对用户抛错。
- column_condition 列条件，
  - SingleColumnValueCondition 支持一列（可以是主键列）和一个常量比较。不支持两列相比较，也不支持两个常量相比较。

- column_name 列名称
- value 列值
  - 格式[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，其他类型都可以省略。
  - 当Type不是BINARY时，可以简写为Value.
- comparator 比较类型
  - EQUAL 表示相等，用 ComparatorTypeConst::CONST_EQUAL 表示
  - NOT_EQUAL 表示不相等，用 ComparatorTypeConst::CONST_NOT_EQUAL 表示
  - GREATER_THAN 表示大于，用 ComparatorTypeConst::CONST_GREATER_THAN 表示
  - GREATER_EQUAL 表示大于等于，用 ComparatorTypeConst::CONST_GREATER_EQUAL 表示
  - LESS_THAN 表示小于，用 ComparatorTypeConst::CONST_LESS_THAN 表示
  - LESS_EQUAL 表示小于等于，用 ComparatorTypeConst::CONST_LESS_EQUAL 表示
- pass_if_missing 由于OTS一行的属性列不固定，有可能存在有condition条件的列在该行不存在的情况，这时参数控制在这种情况下对该行的检查结果。
  - 如果设置为true，则若列在该行中不存在，该节点即认为true。
  - 如果设置为false，则若列在该行中不存在，该节点即认为false。
  - 默认值为true。
- latest_version_only 是否只对最新版本有效。
  - 如果为true，则表示只检测最新版本的值是否满足条件；
  - 如果是false，则会检测所有版本的值是否满足条件，任意一个列值满足条件，该节点即认为true。
  - 默认值为true。

— CompositeColumnValueCondition 一个树形结构，内节点为逻辑运算（logical_operator），叶节点为比较判断SingleColumnValueCondition

- logical_operator 逻辑操作符，枚举类型
  - NOT 表示非，用 LogicalOperatorConst::CONST_NOT 表示。
  - AND 表示并，用 LogicalOperatorConst::CONST_AND 表示。
  - OR 表示或，用 LogicalOperatorConst::CONST_OR 表示。
- sub_conditions 递归下去, 可以继续是SingleColumnValueCondition 或 CompositeColumnValueCondition
  - 其中并和或可以挂载两个或更多子节点，非只能挂载一个子节点。

示例1

构造 SingleColumnValueCondition。

```
    // 设置过滤器, 当Col0的值为0时返回该行.
    $column_condition = [
        'column_name' => 'Col0',
        'value' => 0,
        'comparator' => ComparatorTypeConst::CONST_EQUAL
        'pass_if_missing' => false                          // 如果不存在Col0这一列, 也不返回.
        'latest_version_only' => true                       // 只判断最新版本
    ];
```

示例2

构造 CompositeColumnValueCondition。

```
// composite1 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 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
            ]
        ]
    ];
```

示例3

通过Condition可以实现乐观锁机制，下面例子演示如何通过Condition实现特定列递增功能。

```
// 读一行
    $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这一列, 使列值+1
    $request = [
        'table_name' => 'MyTable',
        'condition' => [
            'row_existence' => RowExistenceExpectationConst::
CONST_EXPECT_EXIST,
            'column_condition' => [                  //条件更新，满足则更
新
                '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);
```

8.8 过滤器

过滤器 Filter 可以在服务器端对读取的结果再进行一次过滤，根据 Filter 中的条件决定返回哪些行或者列。Filter 可以用于 GetRow、BatchGetRow 和 GetRange 接口的column_filter中。

目前表格存储仅支持 SingleColumnValueFilter 和 CompositeColumnValueFilter，这两个 Filter 都是基于参考列的列值决定某行是否会被过滤掉。前者只判断某个参考列的列值，后者会对多个参考列的列值判断结果进行逻辑组合，决定最终是否过滤。

需要注意的是，Filter 是对读取后的结果再进行一次过滤，所以 SingleColumnValueFilter 或者 CompositeColumnValueFilter 中的参考列必须在读取的结果内。如果用户指定了要读取的列，且其中不包含参考列，那么 Filter 无法获得这些参考列的值。 当某个参考列不存在时，SingleColumnValueFilter 的 passIfMissing 参数决定此时是否满足条件，即用户可以选择当参考列不存在时的行为。

格式

```
    'column_filter' => <ColumnFilter>
```

其中 SingleColumnValueFilter 结构如下

```
    [
        'column_name' => '<string>',
        'value' => <ColumnValue>,
        'comparator' => <ComparatorType>
        'pass_if_missing' => true || false
        'latest_version_only' => true || false
    ]
```

CompositeColumnValueFilter结构如下

```
    [
        'logical_operator' => <LogicalOperator>
        'sub_filters' => [
            <ColumnFilter>,
            <ColumnFilter>,
            <ColumnFilter>,
            // other conditions
        ]
    ]
```

格式说明

- column_filter 过滤条件，

- SingleColumnValueFilter 支持一列（可以是主键列）和一个常量比较。不支持两列相比较，也不支持两个常量相比较。
  - column_name 列名称
  - value 列值
    - 格式[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，其他类型都可以省略。
    - 当Type不是BINARY时，可以简写为Value.
  - comparator 比较类型
    - EQUAL 表示相等，用 ComparatorTypeConst::CONST_EQUAL 表示
    - NOT_EQUAL 表示不相等，用 ComparatorTypeConst::CONST_NOT_EQUAL 表示
    - GREATER_THAN 表示大于，用 ComparatorTypeConst::CONST_GREATER_THAN 表示
    - GREATER_EQUAL 表示大于等于，用 ComparatorTypeConst::CONST_GREATER_EQUAL 表示
    - LESS_THAN 表示小于，用 ComparatorTypeConst::CONST_LESS_THAN 表示
    - LESS_EQUAL 表示小于等于，用 ComparatorTypeConst::CONST_LESS_EQUAL 表示
  - pass_if_missing 由于OTS一行的属性列不固定，有可能存在有condition条件的列在该行不存在的情况，这时参数控制在这种情况下对该行的检查结果。
    - 如果设置为true，则若列在该行中不存在，该节点即认为true。
    - 如果设置为false，则若列在该行中不存在，该节点即认为false。
    - 默认值为true。
  - latest_version_only 是否只对最新版本有效。
    - 如果为true，则表示只检测最新版本的值是否满足条件；
    - 如果是false，则会检测所有版本的值是否满足条件，任意一个列值满足条件，该节点即认为true。
    - 默认值为true。

- CompositeColumnValueFilter 一个树形结构，内节点为逻辑运算（logical_operator），叶节点为比较判断SingleColumnValueFilter
  - logical_operator 逻辑操作符，枚举类型
    - NOT 表示非，用 LogicalOperatorConst::CONST_NOT 表示。
    - AND 表示并，用 LogicalOperatorConst::CONST_AND 表示。
    - OR 表示或，用 LogicalOperatorConst::CONST_OR 表示。
  - sub_filters 递归下去, 可以继续是SingleColumnValueFilter 或 CompositeColumnValueFilter
    - 其中并和或可以挂载两个或更多子节点，非只能挂载一个子节点。

示例 **1**

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这一列, 也不返回.
        'latest_version_only' => true                       // 只判断
最新版本
    ];
```

示例 **2**

构造 CompositeColumnValueFilter。

```
    // composite1 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_GREA
TER_THAN
            ]
        ]
    ];
    // composite2 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
            ]
        ]
    ];
```

8.9 主键列自增

主键列自增功能是表格存储新推出的一个功能，PHP SDK 4.0.0 版本开始支持。

主键列自增功能是指若用户指定某一列主键为自增列，在其写入数据时，表格存储会自动为用户在这一列产生一个新的值，且这个值为同一个分区键下该列的最大值。主要用于系统设计中需要使用主键列自增功能的场景，例如电商网站的商品 ID、大型网站的用户 ID、论坛帖子的 ID、聊天工具的消息 ID 等。

特点

- 表格存储目前支持多个主键，第一个主键为分区键，分区键上不允许使用主键列自增功能。
- 除了分区键外，其余主键中的任意一个都可以被设置为递增列。
- 由于是在分区键基础上递增，所以主键列自增是分区键级别的自增。
- 对于每张表，目前只允许设置一个主键列为自增列。
- 自动生成地自增列为 64 位的有符号长整型。

接口

主键自增列功能主要涉及两类接口，创建表和写数据。

- 创建表

  创建表时，只需将需要自增的主键属性设置为 PrimaryKeyOptionConst::CONST_PK_AUTO_INCR。

  相关接口：CreateTable。

```
function createTable($otsClient)
{
    $request = [
        'table_meta' => [
            'table_name' => 'table_name',       // 设置表名
            'primary_key_schema' => [
```

```
                ['PK_1', PrimaryKeyTypeConst::CONST_STRING],
  // 第一个主键列(又叫分片键)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, 并且设置成列自
增主键
            ]
        ],
        '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,              // 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);
}
```

- 第一个主键是分区键，不能设置为自增列。
- 只有整数类型的列才可以设置为自增列。
- 每张表只能设置一个主键自增列。

- 写数据

  写入数据时，只需将自增列的值为设置为占位符 PrimaryKeyTypeConst::CONST_PK_AUTO_INCR。

  相关接口：PutRow/UpdateRow/BatchWriteRow。

```
function putRow($otsClient)
{
    $row = [
        'table_name' => 'table_name',
        'primary_key' => [
            ['PK_1', 'Hangzhou'],                        // 主键名,主键
值, 注意是个list
            ['PK_2', null, PrimaryKeyTypeConst::CONST_PK_AUTO_INCR]
   // 主键递增列,这个值是TableStore产生的,用户在这里不需要填入真实值,只需要
一个占位符:PrimaryKeyTypeConst::CONST_PK_AUTO_INCR 即可.
        ],
        'attribute_columns' => [              // 属性列,注意是个list
            ['name', 'John'],                  // [属性名,属性值,属性
类型,时间戳], 没有设置可以忽略
            ['age', 20],
            ['address', 'Alibaba'],
            ['product', 'OTS'],
            ['married', false]
```

```
        ],
        'return_content' => [
            'return_type' => ReturnTypeConst::CONST_PK      // 列自增
需要主键返回需要设置return_type
        ]
    ];
    $ret = $otsClient->putRow($row);
    print_r($ret);

    $primaryKey = $ret['primary_key'];                    // 这里获取到
的primaryKey可以传递给GetRow, UpdateRow, DeleteRow等API使用
    return $primaryKey;
}
```

- 写入数据时，主键自增列不需要填值，只需填充占位符即可。
- 如果用户想获取向表格存储写入数据后自动生成的主键值，可以将 ReturnType 设置为 ReturnTypeConst::CONST_PK，这样就能在写入成功后返回主键值。

8.10 错误处理

TableStore 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，比如 BatchGetRow 和 BatchWriteRow 不仅需要判断是否有异常，还需要检查每个 row 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

TableStore PHP 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，比如参数设置不对等。
- OTSServerException 指服务器端错误，它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，比如 200、404 等。
  - getOTSErrorCode()：表格存储返回的错误类型字符串。参考错误码。
  - getOTSErrorMessage()：表格存储返回的错误详细描述。
  - getRequestId()：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。

8.11 常见问题

如果前面的文档还不能解决您的问题，您可以使用以下途径联系我们的工程师团队：

联系我们

- 前往云栖社区问答板块提问题。
- 前往工单系统提交工单。
- 扫码加入TableStore讨论群，和我们直接交流讨论

9 历史版本 SDK 下载

9.1 Java SDK 历史迭代版本

4.0.0 以上版本的 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。

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：

SearchIndex相关

- 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SearchIndex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：增加主键列自增功能。

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：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，详情请参见老版 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，比如 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。
- 当您不再使用一个 OTSClient 实例时（比如程序结束前），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，比如在配置项名称中加入了时间单位。
- 新 SDK 的包依赖有变化，比如使用了 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 NodeJs 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

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：

- Fix 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：

- 支持多版本
- 支持TTL
- 支持主键列自增功能

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：

- 支持ConditionUpdate和Filter功能
- 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

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。
- 清理没用的代码。
- 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

版本号 **4.1.0**

发布时间：2018-7-24

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：2018-6-25

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Google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：2017-1-14

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。