# 阿里云
# 表格存储

## 最佳实践

文档版本：20190918

# 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:**<br>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:**<br>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:**<br>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。 |
| *## * | 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 表操作篇

本文将为您提供关于表设计的最佳实践。

## 设计良好的主键

表格存储会根据表的分区键将表的数据自动切分成多个分区，每个分区调度到一台服务节点上。分区键的值是最小的分区单位，相同的分区键值下的数据无法再做切分。为了防止某一个分区键值的数据成为访问热点达到单机服务能力上限，应用程序需要让数据的分布和访问量的分布尽可能均匀。

表格存储会对表中的行按主键进行排序，合理设计主键可以让数据在分区上的分布更加均匀，从而能够充分地利用表格存储水平扩展的特点。

选取分区键时，建议遵循以下几个原则：

- 单个分区键值中的数据不宜过大，建议不超过 10 GB。

  说明：
  10 GB 是限制为了避免访问热点，而不是数据存储的限制。
- 一张表内，不同分区键值中的数据在逻辑上是独立的。
- 访问压力不要集中在小范围连续的分区键值中。

### 使用示例

例如，有一张表，里面存储的是某大学内所有学生使用学生卡消费的记录。主键列有学生卡 ID（CardID），商家 ID（SellerID），消费终端 ID（DeviceID），订单号（OrderNumber）。同时我们有如下约定：

- 每一张学生卡对应一个 CardID，每一个商家对应一个 SellerID。
- 每一个消费终端对应 DeviceID，DeviceID 在全局是唯一的。
- 在每一个消费终端上产生的每一笔消费记录一个 OrderNumber。一个消费终端产生的 OrderNumber 是唯一的，但是在全局范围内 OrderNumber 不唯一。例如，不同的消费终端有可能产生两条完全不同的消费记录，但是它们的 OrderNumber 相同。
- 同一个消费终端产生的 OrderNumber 按时间排序，新的消费记录比老的消费记录拥有更大的 OrderNumber。
- 每笔消费记录均会被实时写入这张表中。

为高效利用表格存储，在设计表格存储的表的主键时，需考虑表的分区键：

| 分区方式 | 说明 |
| --- | --- |
| 使用 CardID 作为表的分区键 | 使用 CardID 作为表的分区键是一个比较好的选择。每天每张卡产生的消费记录数从总体上来讲是均匀的，每一个分区中的访问压力也应该是均匀的。以 CardID 作为表的分区键可以较好地利用预留读/写吞吐量资源。 |
| 使用 SellerID 作为表的分区键 | 使用 SellerID 作为表的分区键不是一个好的选择。因为学校内的商铺数量相对较少，同时一些商铺可能产生大量的消费记录成为热点，不利于访问压力的均匀分配。 |
| 使用 DeviceID 作为表的分区键 | 使用 DeviceID 作为表的分区键是一个比较好的选择。尽管每家商铺的消费记录数可能相差较大，但是每天每台消费终端上产生的消费记录数是可预期的。消费终端每天产生消费记录的条数取决于收银员操作的速度，这就决定了一台消费终端产生的消费记录数是受限的。因此，使用 DeviceID 作为表的分区键也可以保证访问压力的相对均匀。 |
| 使用 OrderNumber 作为表的分区键 | 使用 OrderNumber 作为表的分区键不是一个好的选择。因为 OrderNumber 是顺序增长的，因此在同一段时间内产生的消费订单的 OrderNumber 的值会集中在一个较小的范围内，这些消费订单记录会集中写入到个别的分区，以致预留读/写吞吐量没能得到高效利用。如果必须使用 OrderNumber 作为分区键，建议在 OrderNumber 上进行哈希散列，将哈希值作为 OrderNumber 的前缀，保证数据和访问压力的均匀。 |

**总结**

可以根据需求将 CardID 和 DeviceID 作为表的分区键，而不应该使用 SellerID 和 OrderNumber。之后再根据应用程序的实际需求来设计剩余的主键列。

### 通过拼接方式使用分区键

建议表格存储中表的同一分区键值下的数据量大小不超过 10 GB。如果您的表中单个分区键值的所有行的总数据量大小可能超过 10 GB，在设计表时可以将原来的多个主键列拼接成分区键。

**使用示例**

例如，上一小节中提到的学生卡消费记录表，假设主键为 [DeviceID, SellerID, CardID, OrderNumber]。DeviceID 是该表的分区键，单个 DeviceID 中所有行的数据量总大小可能超过 10 GB，可以将 DeviceID、SellerID 和 CardID 拼接作为表的第一个主键列（即分区键）。

原来的表如下所示：

| DeviceID | SellerID | CardID | OrderNumber | attrs |
|---|---|---|---|---|
| 16 | 'a100' | 66661 | 200001 | ... |
| 54 | 'a100' | 6777 | 200003 | ... |
| 54 | 'a1001' | 6777 | 200004 | ... |
| 167 | 'a101' | 283408 | 200002 | ... |

将 DeviceID、SellerID 和 CardID 拼接成分区键后的表如下所示：

| CombineDeviceIDSellerIDCardID | OrderNumber | attrs |
|---|---|---|
| '16:a100:66661' | 200001 | ... |
| '167:a101:283408' | 200002 | ... |
| '54:a1001:6777' | 200004 | ... |
| '54:a100:6777' | 200003 | ... |

在原来的表中，Device=54 的两行是属于同一个分区键值为 54 下的两条消费记录。在新的表中，这两条消费记录拥有不同的分区键值。通过拼接多列主键列形成分区键的表减少了单个分区键值下的总数据量大小。

选择将 DeviceID、SellerID 和 CardID 拼接成分区键，而不选择将 DeviceID 和 SellerID 进行拼接的原因是，前一节提到的消费记录表约定所有 DeviceID 相同的消费记录其 SellerID 也相同，因此仅仅拼接 DeviceID 和 SellerID 并不能解决单个分区键值的数据量过大的问题。

但是，拼接主键列形成表有一些小瑕疵。DeviceID 是一个 Integer 类型的主键列，在原来的表中，DeviceID=54 的消费记录在 DeviceID=167 的前面。将前三列主键列拼接成 String 类型的主键列后，DeviceID=54 的消费记录在 DeviceID=167 的后面。假如应用程序需要范围读取 DeviceID 在 [15, 100) 之间所有的消费记录，上面的表无法满足需求。

为了应对这种状况，可以在 DeviceID 高位补 0。补 0 的个数取决于 DeviceID 最大位数。假设 DeviceID 的取值范围是 [0, 999999]，可以将 DeviceID 高位补 0 至 6 位后再进行拼接，得到的表如下所示：

| CombineDeviceIDSellerIDCardID | OrderNumber | attrs |
| --- | --- | --- |
| '000016:a100:66661' | 200001 | ... |
| '000054:a1001:6777' | 200004 | ... |
| '000054:a100:6777' | 200003 | ... |
| '000167:a101:283408' | 200002 | ... |

经过高位补 0 后的表依然有一些问题。在原来的表中，DeviceID=54 的两行，SellerID='a1001' 的行应该在 SellerID='a100' 的后面。产生这种现象的原因是，'000054:a1001' 的字典序小于 '000054:a100:'，但是 'a1001' 的字典序大于 'a100'，连接符 : 影响了字典序。在选取连接符时，应该选取比所有可用字符的 ASCII 码都小的字符作为连接符。在该表中，SellerID 的取值为数字、大小写英文字母。我们可以使用 , 作为连接符，因为 , 比所有 SellerID 可用字符的 ASCII 码都小。

使用 , 拼接后的表如下所示：

| CombineDeviceiDSellerIDCardID | OrderNumber | attrs |
| --- | --- | --- |
| '000016,a100,66661' | 200001 | ... |
| '000054,a100,6777' | 200003 | ... |
| '000054,a1001,6777' | 200004 | ... |
| '000167,a101,283408' | 200002 | ... |

上面经过拼接形成的分区键的表的记录顺序就和原来的表保持一致了。

总结

当表中单个分区键值的所有行的数据量总大小可能超过 10 GB 时，可以将多个主键列拼接成分区键，以避免单分区键值的数据量大小限制。在拼接分区键时需要注意以下事项：

- 选取需要拼接的多个主键列，必须能有效地将原来表中相同的分区键值的记录，变成拥有不同分区键值的记录。
- 拼接 Integer 类型主键列时可以在高位补 0，保持记录的顺序一致。
- 选取连接符时需要考虑连接符对新的分区键的字典序的影响，选取比所有可用字符都小的连接符是一个比较安全的选择。

在分区键中加入哈希前缀

使用示例

在设计良好的主键一节中提到，尽量不要使用 OrderNumber 作为表的分区键。因为 OrderNumber 是顺序增长的，消费记录总是被写入最新的 OrderNumber 范围之内，旧的 OrderNumber 不再有写入压力，造成访问压力不均匀的现象，以致预留读/写吞吐量得不到高效利用。如果必须使用顺序增长的键值作为分区键，我们可以对分区键拼接哈希前缀，让相连的 OrderNumber 在表中随机分布，使访问压力分布均匀。

以 OrderNumber 为分区键的消费记录表如下所示：

| OrderNumber | DeviceID | SellerID | CardID | attrs |
|---|---|---|---|---|
| 200001 | 16 | 'a100' | 66661 | ... |
| 200002 | 167 | 'a101' | 283408 | ... |
| 200003 | 54 | 'a100' | 6777 | ... |
| 200004 | 54 | 'a1001' | 6777 | ... |
| 200005 | 66 | 'b304' | 178994 | ... |

对 OrderNumber 使用 md5 算法计算前缀（您也可以采取其他哈希散列算法），拼接成 HashOrderNumber。因为 md5 算法计算得到的哈希字符串可能过长，我们只需要取前几位就能达到让 OrderNumber 相连的记录在表中随机分布的目的。在这个例子中我们取前 4 位：

| HashOrderNumber | DeviceID | SellerID | CardID | attrs |
|---|---|---|---|---|
| '2e38200004' | 54 | 'a1001' | 6777 | ... |
| 'a5a9200003' | 54 | 'a100' | 6777 | ... |
| 'c335200005' | 66 | 'b304' | 178994 | ... |
| 'db6e200002 | 167 | 'a101' | 283408 | ... |
| 'ddba200001' | 16 | 'a100' | 66661 | ... |

在后续访问消费记录时，使用相同的算法对 OrderNumber 计算哈希前缀，即可得到对应消费记录的 HashOrderNumber。在分区键中加入哈希前缀的弊端是，原来连续的记录会被打散，无法再使用 GetRange 操作读取一段范围内在逻辑上连续的记录。

### 并行写入数据

表格存储的表会被切分成多个分区，这些分区被分散在多个表格存储的服务器上。如果有一批数据要上传到表格存储中，同时这批数据是按主键排列顺序的，若按顺序写入数据，可能会导致写入压力集中在某个分区中，而其他的分区处于空闲状态，无法有效利用预留读/写吞吐量，影响数据导入速度。

可以采取以下任一措施来提升导入数据的速率：

- 将原始数据顺序打乱后再进行导入，以保证写入数据均匀地分配在各个分区上。
- 使用多个工作线程并行导入数据。把大的数据集合切分成很多个小集合，工作线程随机选取小集合进行数据导入。

区分冷数据和热数据

数据往往具有时效性。例如，在设计良好的主键一节中提到的存储消费记录表，近期产生的消费记录被访问的可能性较大，因为应用程序需要及时地对消费记录进行处理和统计，或者查询最近的消费记录。但是年代久远的消费记录被查询的可能性不大，这些数据渐渐成为冷数据，但仍然占用存储空间。

其次，表中存在大量冷数据会导致数据访问压力不均匀，从而导致表上配置的预留读/写吞吐量无法被充分利用。例如，已经毕业的学生的卡片，不会再产生消费记录。假如 CardID 是随着卡片申请时间递增的，以 CardID 作为分区键，会导致已经毕业的学生的 CardID 没有访问压力却被分配到预留读/写吞吐量，造成浪费。

为了解决这种问题，可以用不同的表来区分冷热数据，并设置不同的预留读/写吞吐量。例如，将消费记录按月份分表，每一个新的自然月就换一张新的表。当月的消费记录表需要不停写入新的消费记录，同时有查询操作。当月的消费记录表可以设置一个较大的预留读/写吞吐量配置来满足访问需求。前几个月的表由于不再写入新数据或者写入的新数据量较少，查询的请求较多，因此前几个月的消费记录表可以设置较小的预留写吞吐量，较大的预留读吞吐量。而历史超过一年的消费记录表，由于再被使用的可能性不大，可以设置较小的预留读/写吞吐量配置。已经超出维护年限的消费记录表可以将数据导出，存入 OSS（Object Storage Service）归档，或直接删除。

# 2 数据操作篇

本文将为您提供关于数据操作的最佳实践。

### 拆分属性列访问热度差异大的表

如果行的属性列较多，但是每次操作只访问一部分属性列，可以考虑将表拆分成多个表，将不同访问频率的属性列放到不同的表中。

例如，在商品管理系统中，每行存放商品数量、商品价格和商品简介。商品数量和商品价格均为占用空间较小的 Integer 类型，商品简介是 String 类型占用空间较大。大多数操作仅更新商品数量与商品价格，而不修改商品简介，所以商品简介的修改频率较低。这时候可以考虑将这个表拆分为两个，一个表存储商品数量和商品价格，另一个表存储商品简介。

### 压缩较大的属性列文本

如果属性列是较大的文本，应用程序可以考虑将属性列压缩之后再以 Binary 类型存储到表格存储中。这样做节省了空间、减少了访问的服务能力单元消耗，从而可以降低使用表格存储的成本。

### 将数据量超出限制的属性列存储到 OSS 中

表格存储限制单个属性列值不超过 2 MB。如果需要存储单个值超过 2 MB 的需求，如图片、音乐、文件等，可以使用 OSS（Object Storage Service）对其进行存储。OSS是阿里云提供的开放存储服务，用以应对海量数据的存储和访问。OSS 的存储单价比表格存储更低，更适合存储文件。

如果应用程序不方便使用 OSS，可以将超过 2 MB 的单个值拆分成多个行，存储在表格存储中。

### 错误重试加入时间间隔

表格存储可能会遇到软硬件问题，导致应用程序的部分请求失败，并返回可重试的错误。建议应用程序遇到此类错误时等待一段时间后再进行重试，每两次重试之间应该加大时间间隔或引入随机时间间隔，避免重试的请求堆积到一个时间点上引发雪崩效应。

# 3 亿量级订单管理方案

## 3.1 行业背景

订单系统存在于各行各业，如电商订单、银行流水、运营商话费账单等，是一个非常广泛、通用的系统。随着互联网的发展，以及各企业对数据的重视，需要存储和持久化的订单量越来越大。这些为订单系统管理来了新挑战。

### 需求场景

某电商平台A，需要存储和维护订单数据。同时，基于所有的订单数据，系统又需要向三类人群提供多元化查询服务：消费者、店家、平台。通过订单系统，消费者可以查询自己的历史订单，商家可以统计热销产品，平台也可以分析用户行为、平台交易规模等。主要查询方式涵盖订单的多维度检索，以及订单数据的分析、统计等，例如：

面向消费者：【A消费者】*【近1年】*【卖出电脑】订单查询。

面向售货员：【B售货员】*【近1个月】销售订单。

……

### 技术点

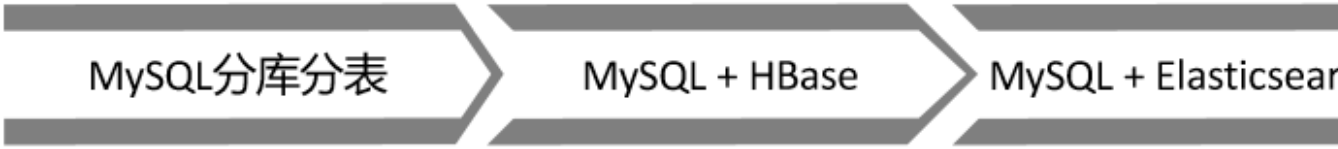
在订单场景中，需要考虑的技术点主要包含下几个方面：

- 查询能力：需要具备丰富的查询类型，如多维度、范围、模糊查询等，同时具备排序、统计等功能。
- 数据量：存储海量数据的同时，满足强一致、高可用、低成本等要求。
- 服务性能：应对高并发请求高并发的同时，保证低延迟。

其中，多维、实时查询功能是订单管理解决方案的核心功能。

## 3.2 方案选择

本章节主要为您介绍针对订单系统的一些传统解决方案，以及面对亿量级订单，表格存储（Table Store）提供的更全面的解决方案。




传统方案一：MySQL分库分表

MySQL自身拥有强大的数据查询、分析功能，基于MySQL创建订单系统，可以应对订单数据多维查询、统计场景。伴随着订单数据量的增加，采取分库、分表方案应对，通过这种伪分布式方案解决数据膨胀带来的问题。但数据一旦达到瓶颈，便出现明显的弊端：

- 数据纵向（数据规模）膨胀：采用分库分表方案，MySQL在部署时需要预估分库规模，数据量一旦达到上限后，重新部署并做数据全量迁移。
- 数据横向（字段维度）膨胀：schema需预定义，迭代新增新字段变更复杂。而维度到达一定量后影响数据库性能。

传统方案二： MySQL+HBase

引入双数据的方案应运而生，通过实时数据、历史数据分存的方案，可以一定程度解决数据量膨胀问题。该方案将数据归类成两部分存储：实时数据、历史数据。

- 实时订单数据（例如：近3个月的订单）：将实时订单存入MySQL数据库。实时订单的总量膨胀的速度得到了限制，同时保证了实时数据的多维查询、分析能力。
- 历史订单数据（例如：3个月以前的订单）：通过数据同步服务，将历史订单数据存入HBase，借助于HBase这一分布式NoSQL数据库，有效应对了订单数据膨胀困扰。也保证了历史订单数据的持久化。

但是，该方案牺牲了历史订单数据对用户、商家、平台的使用价值，假设了历史数据的需求频率极低。但是一旦有需求，便需要全表扫描，查询速度慢、I/O成本很高。而维护数据同步又带来了数据一致性、同步运维成本飙升等难题。

传统方案三：MySQL+Elasticsearch

此方案同样是将数据分两部分存储，可以一定程度解决订单索引维度增长问题。用户自己维护数据同步服务，保证两部分数据的一致性。

- 全量数据：将全量的订单数据存入MySQL数据库，订单ID之外的数据整体存为一个字段。该全量数据作为持久化存储，也用于非索引字段的反查。
- 查询数据：仅将需要检索的字段存入Elasticsearch（基于Lucene分布式索引数据库），借助于Elasticsearch的索引能力，提供可以应付维度膨胀的订单数据，然后必要时反查MySQL获取订单完整信息。

该方案应付了数据维度膨胀带来的困扰，但是随着订单量的不断膨胀，MySQL扩展性差的问题再次暴露出来。同时数据同步至Elasticsearch的方案，开发、运维成本很高，方案选择也存在弊端。

### 表格存储解决方案

由上可以看出，传统的方案并不能完全应对亿量级订单场景。使用Table Store研发的多元索引（SearchIndex）方案，则可以完美地解决亿量级订单系统问题。Table Store具有即开即用，按量收费等特点。多元索引随时创建，是海量电商订单元数据管理的优质方案。

Table Store作为阿里云提供的一款全托管、分布式NoSQL型数据存储服务，具有海量数据存储、热点数据自动分片、海量数据多维检索等功能，天然地解决了订单数据大爆炸这一挑战。同时，多元索引功能在保证用户数据高可用的基础上，提供了数据多维度搜索、统计等能力。针对多种场景创建多种索引，实现多种模式的检索。可以仅在需要的时候创建、开通索引。由Table Store来保证数据同步的一致性，这极大的降低了用户的方案设计、服务运维、代码开发等工作量。

下图为Table Store与传统订单方案使用的工具在各项性能上的对比：

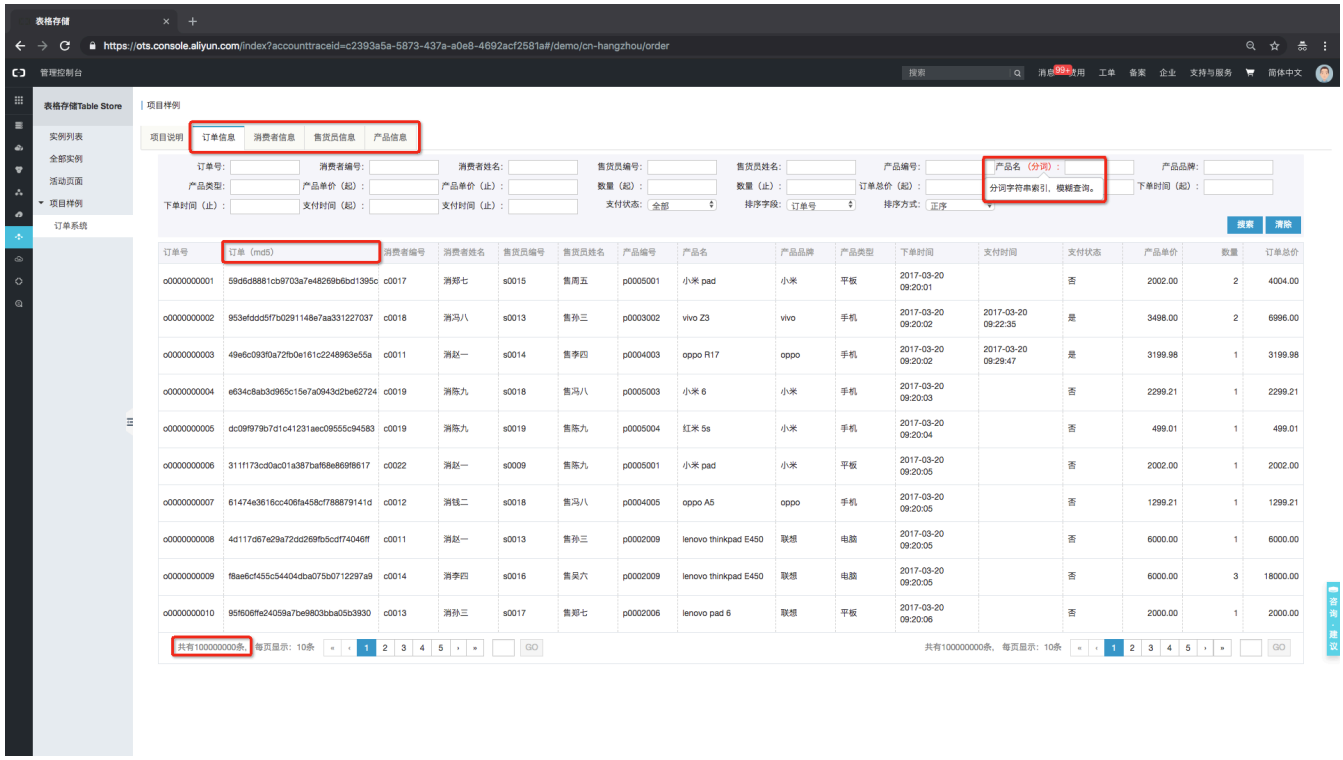
| 能力分析 | MySql | HBase | Elasticsearch | TableStore |
| --- | --- | --- | --- | --- |
| 存储方式 | 行存储 | 列存储 | 索引存储 | 列存储+索引存储 |
| 扩展性 | 单机、扩展性差 | 水平扩展 | 水平扩展 | （自动）水平扩展 |
| 一致性 | 强一致性 | 强一致性、时序一致性 | | 强一致性、时序一致性 |
| 检索 | 较弱的支持 | 不支持 | 支持 | 支持 |
| 数据量 | ~ 1T，~亿行 | ~10 PB，~万亿行 | ~1 PB，~千亿行 | ~10 PB，~万亿行 |

方案实现

参见准备工作以及#unique_11。

方案样例

基于表格存储搭建的订单系统样例内嵌在表格存储控制台中，用户可登录控制台体验系统。该样例提供了亿量级订单数据。官网控制台地址：项目样例。



## 3.3 准备工作

开始使用表格存储（Table Store）搭建订单系统前，您需要完成以下准备工作。

### 开通表格存储

具体操作步骤参见#unique_12。

### 创建表格存储实例

#unique_13（Instance）是您使用和管理表格存储服务的实体，每个实例相当于一个数据库。表格存储对应用程序的访问控制和资源计量都在实例级别完成。创建表格存储实例的具体操作步骤参见#unique_14。

#### 下载SDK

选择以下语言的SDK包进行下载安装：

- Java SDK
- Python SDK
- Go SDK
- Node.js SDK
- .NET SDK
- PHP SDK

#### 设计数据表

订单系统不仅仅是订单一张数据表，它应包含：消费者表、售货员表、产品表、供货商表、交易订单表、支付订单表等。在本次搭建示例中，主要使用最基本的四张表（消费者表、售货员表、产品表、交易订单表），如下图：

表名：order_contract

| 列名 | 数据类型 | 索引类型 | 字段说明 |
| --- | --- | --- | --- |
| _id(主键列) | String | | MD5(oId)避免热点 |
| oId(主键列) | String | KEYWORD | 订单编号 |
| pName | String | TEXT | 产品名，TEXT类型索引可模糊查询，但 |
| totalPrice | double | DOUBLE | 订单总价 |
| orderTime | long | LONG | 下单时间（时间戳） |
| ... | ... | ... | ... |

## 3.4 搭建订单系统

本章节主要为您介绍如何使用表格存储（Table Store）搭建亿量级订单管理系统。

#### 前提条件

您已经完成了准备工作。

步骤一 创建数据表

创建四张表：订单表、消费者表、售货员表、产品表。您仅需将四张表创建在同一个实例，您可以通过以下两种方式创建数据表：

- 通过控制台创建、管理数据表，具体参见#unique_22。
- 通过SDK直接创建、管理数据表。

步骤二 创建数据表多元索引

Table Store自动做全量、增量的索引数据同步。您可以通过以下两种方式创建、管理多元索引：

- 通过控制台创建、管理多元索引，具体参见#unique_23。
- 通过SDK创建、管理多元索引。

步骤三 导入数据

插入部分测试数据。

说明：

控制台项目样例中插入了1亿条数据，您可以通过控制台插入少量测试数据。

| 订单号 | 订单（md5）（主键） | 消费者编号 | 消费者姓名 | 售货员编号 | 售货员姓名 | 产品编号 | 产品名 | 产品品牌 | 产品类型 | 下单时间 | 支付时间 | 支付状态 | 产品单价 | 数量 | 总价钱 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| o00000000000000 | c49f5fd5aba33159accae0d3ecd749a7 | c0019 | 消陈九 | s0020 | 售楚十 | p0003004 | vivox21 | vivo | 手机 | 2018-07-17 21:00:00 | | 否 | 2498.999 | 2 | 4997.998 |

| 消费者编号（主键） | 消费者姓名 | 消费者积分 | 注册时间 |
|---|---|---|---|
| c0001 | 消赵一 | 818 | 2018-07-07 14:33:51 |

| 售货员编号（主键） | 售货员姓名 | 售货员积分 | 入职日期 |
|---|---|---|---|
| s0001 | 售赵一 | 613 | 2018-07-07 14:27:59 |

| 产品编号（主键） | 产品名 | 产品品牌 | 产品类型 | 产品单价 | 新增时间 |
|---|---|---|---|---|---|
| p0001001 | iphone 6 | 苹果 | 手机 | 6969.00 | 2018-07-07 14:44 |

步骤四 读取数据

数据读取分为两类：

- 主键读取

  基于原生表格存储的主键列获取：getRow、getRange、 batchGetRow等。主键读取用于索引（自动）反查，用户也可以提供主键（订单md5）的单条查询的页面，亿量级下查询速度极快。单主键查询方式不支持多维度检索。

- 索引读取

  基于新SearchIndex功能Query：search接口。用户可以自由设计索引字段的多维度条件组合查询。通过设置选择不同的查询参数，构建不同的查询条件、不同排序方式；目前支持：精确查询、范围查询、前缀查询、匹配查询、通配符查询、短语匹配查询、分词字符串查询，并通过布尔与、或组合。

  如【c0001号消费者，且消费在99.99以上的订单】组合方式如下：

```
List<Query> mustQueries = new ArrayList<Query>();

TermQuery termQuery = new TermQuery();
termQuery.setFieldName("cId");
termQuery.setTerm(ColumnValue.fromString("c0001"));
mustQueries.add(termQuery);

RangeQuery rangeQuery = new RangeQuery();
rangeQuery.setFieldName("totalPrice");
rangeQuery.setFrom(ColumnValue.fromDouble(99.99)) ;
mustQueries.add(rangeQuery);

BoolQuery boolQuery = new BoolQuery();
boolQuery.setMustQueries(mustQueries);
```

专家服务

表格存储提供专业的免费的技术咨询服务，欢迎加入我们的钉钉讨论群。


表格存储公开交流群
1023人
扫一扫群二维码，立刻加入该群。

# 4 基于多元索引搭建亿量级店铺搜索系统

## 4.1 方案背景

本章节主要为您介绍基于多元索引搭建亿量级店铺搜索系统的需求场景及方案。

亿量级店铺搜索系统的核心点与瓶颈在于数据库是否有海量存储能力、低延迟读能力、高效查询能力以支持GEO+多维度数据检索。表格存储作为阿里云提供的一款全托管、零运维的分布式NoSQL型数据存储服务，具有海量数据存储、热点数据自动分、海量数据多维检索等功能，可以有效的地解决GEO数据量大膨胀这一挑战。

需求场景

某店铺搜索平台，提供了亿量级的店铺信息。用户通过平台提供的PC端、移动端网页，按照自己的需求维度组合，搜索用户心仪的店铺。平台需要在地图上展示店铺的具体位置、店铺详细信息、店铺主页的跳转。

维度一：【距离1km内】【人均100以内】【评分最高】【奶茶店】

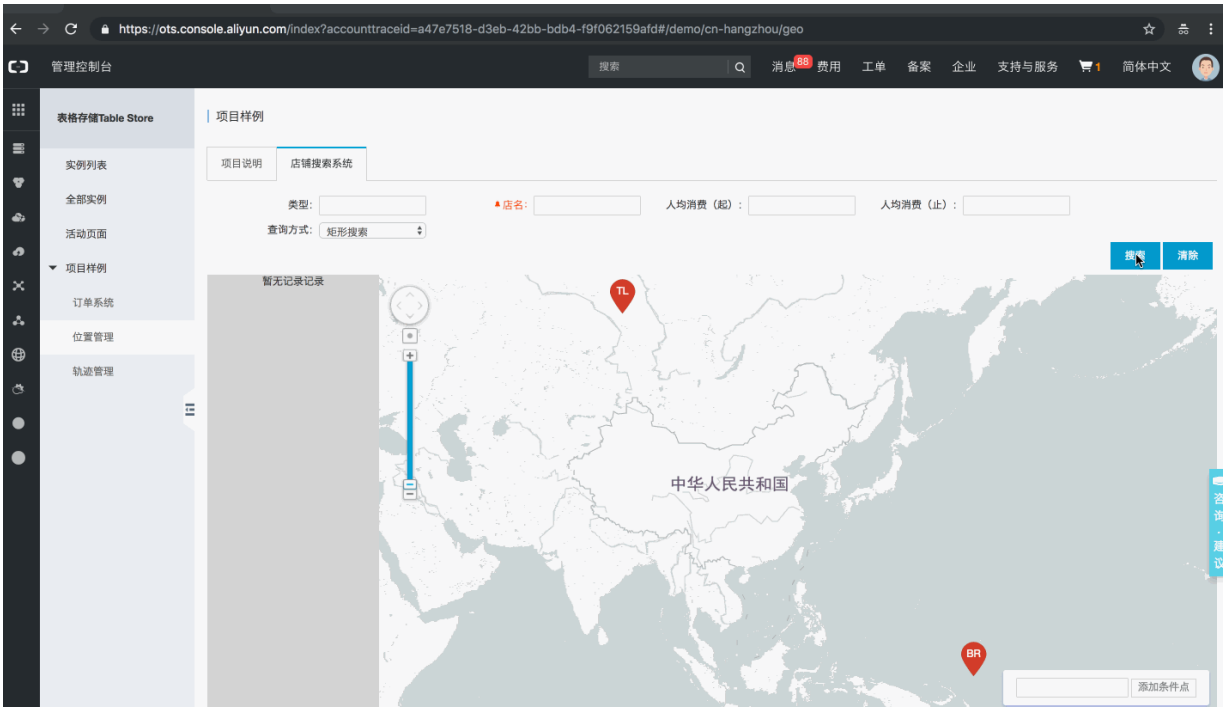
维度二：【杭州市内】【评分最高的】【沈家*】店铺

……

实现快速、多维GEO查询功能，是GEO管理解决方案的核心功能，样例如下：

说明：

该样例提供了【亿量级】店铺数据。具体参见项目样例。




基于表格存储搭建的店铺搜索系统页面一览，样例内嵌在表格存储控制台中，用户可登录控制台体验系统（若为表格存储的新用户，需要点击开通服务后体验，开通免费，订单数据存储在公共实例中，体验不消耗用户存储、流量、CU）。

### 表格存储方案

使用表格存储研发的多元索引（SearchIndex）功能，可以轻松搭建一套亿量级店铺搜索系统。多元索引功能可以创建GEO索引、分词字符串索引等，为用户提供了GEO检索、多维组合检索等能力，用户可随时创建，存量、增量数据自动同步。

用户可以仅在需要的时候创建、开通索引。由表格存储来保证数据同步的一致性，这极大的降低了用户的方案设计、服务运维、代码开发等工作量。

## 4.2 准备工作

开始使用表格存储搭建亿量级店铺搜索系统前，您需要完成以下准备工作。

### 开通表格存储

具体操作步骤参见#unique_12。

### 创建表格存储实例

#unique_13（Instance）是您使用和管理表格存储服务的实体，每个实例相当于一个数据库。表格存储对应用程序的访问控制和资源计量都在实例级别完成。创建表格存储实例的具体操作步骤参见#unique_14。

### 下载SDK

选择以下语言的SDK包进行下载安装：

- Java SDK
- Python SDK
- Go SDK
- Node.js SDK
- .NET SDK
- PHP SDK

### 设计数据表

店铺检索系统样例，仅简易使用一张店铺表，主要包含字段：店铺类型、店铺名称、店铺地理位置、店铺平均评分、人均消费消等。表设计如下：

表名：geo_positon

| 列名 | 数据类型 | 索引类型 | 字段说明 |
| --- | --- | --- | --- |
| _id(主键列) | String | | MD5(pId)避免热点 |
| pId | Stirng | | 店铺编号 |
| type | String | KEYWORD | 类型 |
| name | String | TEXT | 店铺名，TEXT类型索引可模糊查询，但不能排 |
| pos | String | GEO_POINT | 店铺位置："30.132,120.082"(纬度,精度) |
| point | double | DOUBLE | 评分 |
| ... | ... | ... | ... |

## 4.3 方案实现

本章节主要为您介绍如何使用表格存储搭建亿量级店铺搜索系统。

### 前提条件

您已经完成了#unique_28。

步骤一 创建数据表

通过以下两种方式创建一张店铺信息表：

- 通过控制台创建、管理数据表，具体参见#unique_22。
- 通过SDK直接创建、管理数据表。

步骤二 创建数据表索引

表格存储自动进行全量、增量的索引数据同步。您可以通过以下两种方式创建、管理多元索引：

- 通过控制台创建、管理多元索引，具体参见#unique_23。
- 通过SDK创建、管理多元索引。

步骤三 导入数据

控制台项目样例中插入了1亿条数据，您可以通过控制台插入少量测试数据。

| 店铺编号 | 店铺（md5）（主键） | 类型 | 店铺名称 | 店铺位置 | 店铺评分 | 人均消费 | | |
|---|---|---|---|---|---|---|---|---|
| o00570 22192 | 0000000f470ef0f548b9 25ceffe1a7e3 | 杭帮菜 | 韩村杭帮菜 | 36.76613,11 1.41461 | 2.87 | 63.6 7 | | |

步骤四 读取数据

数据读取分为两类：

- 主键读取

  基于原生表格存储的主键列获取。主键读取用于索引（自动）反查，用户也可以提供主键（订单md5）的单条查询的页面，亿量级下查询速度极快。单主键查询方式不支持多维度检索。

- 索引读取（店铺查询）

  基于新SearchIndex功能Query：search接口。用户可以自由设计索引字段的多维度条件组合查询。通过设置选择不同的查询参数，构建不同的查询条件、不同排序方式；目前支持：精确查询、范围查询、前缀查询、匹配查询、通配符查询、短语匹配查询、分词字符串查询，并通过布尔与、或组合。

  如【"36.76613,111.41461"周边1km范围内的奶茶店】，查询条件如下：

```
List<Query> mustQueries = new ArrayList<Query>();

TermQuery termQuery = new TermQuery();
termQuery.setFieldName("type");
termQuery.setTerm(ColumnValue.fromString(奶茶));
```

```
mustQueries.add(termQuery);

GeoDistanceQuery geoDistanceQuery = new GeoDistanceQuery();
geoDistanceQuery.setFieldName("pos");
geoDistanceQuery.setCenterPoint("36.76613,111.41461");
geoDistanceQuery.setDistanceInMeter(1000);
mustQueries.add(geoDistanceQuery);

BoolQuery boolQuery = new BoolQuery();
boolQuery.setMustQueries(mustQueries);
```

## 专家服务

**表格存储提供专业的免费的技术咨询服务，欢迎加入我们的钉钉讨论群。**

# 5 搭建海量智能元数据管理系统

## 5.1 方案背景

本章节主要为您介绍搭建海量智能元数据管理系统的场景、技术点以及搭建方案。

用户存储海量的文档、媒体文件等数据的同时，对文件元数据（Meta）的管理不可或缺。元数据拥有多维度的字段信息，基本信息包含文件大小、创建时间、用户等。随着人工智能的发展，通过AI技术提取文件核心要素也成为文件元数据的重要信息。以图片为例：用户通过智能媒体服务，获取分析图片核心标签并为标签打分，用户还可提取人脸识别相关信息，以及地理位置等信息，提取的信息也需要存储到文件元数据信息中。因而文件元数据的信息量不断增加，格式、类型也不断呈现多元化。

### 需求场景

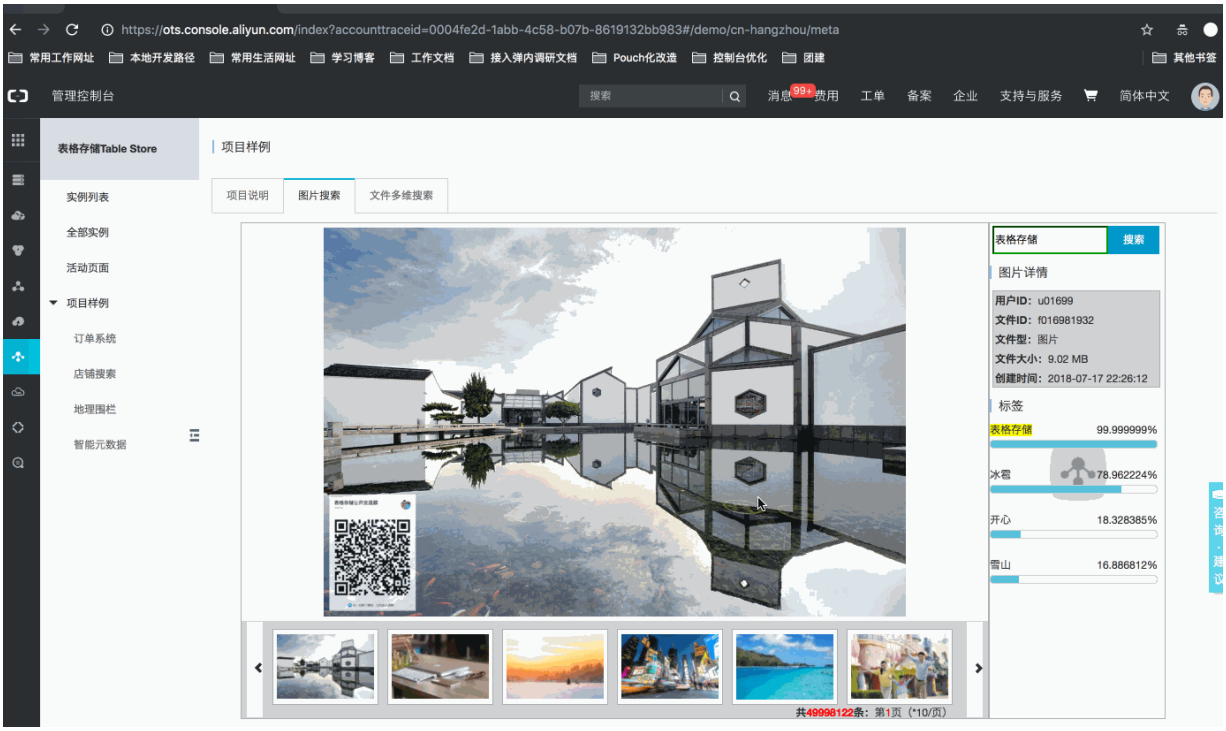
某智能媒体管理平台，为用户提供文件（图片、视频等）管理服务，用户通过自研（或售卖）的智能媒体分析工具，为目标文件进行分析。用分析后的信息丰富原有的元数据信息。因此，平台需要一套有效的元数据管理方案，为用户提供元数据信息的管理、分析、统计功能。例如：

用户A：【用户A的文件】*【近1年】*【标签含[开心]】*的所有图片，按标签分数排序

用户B：【用户B的文件】*【出现某某明星】*的所有视频，按明星相似度排序

……

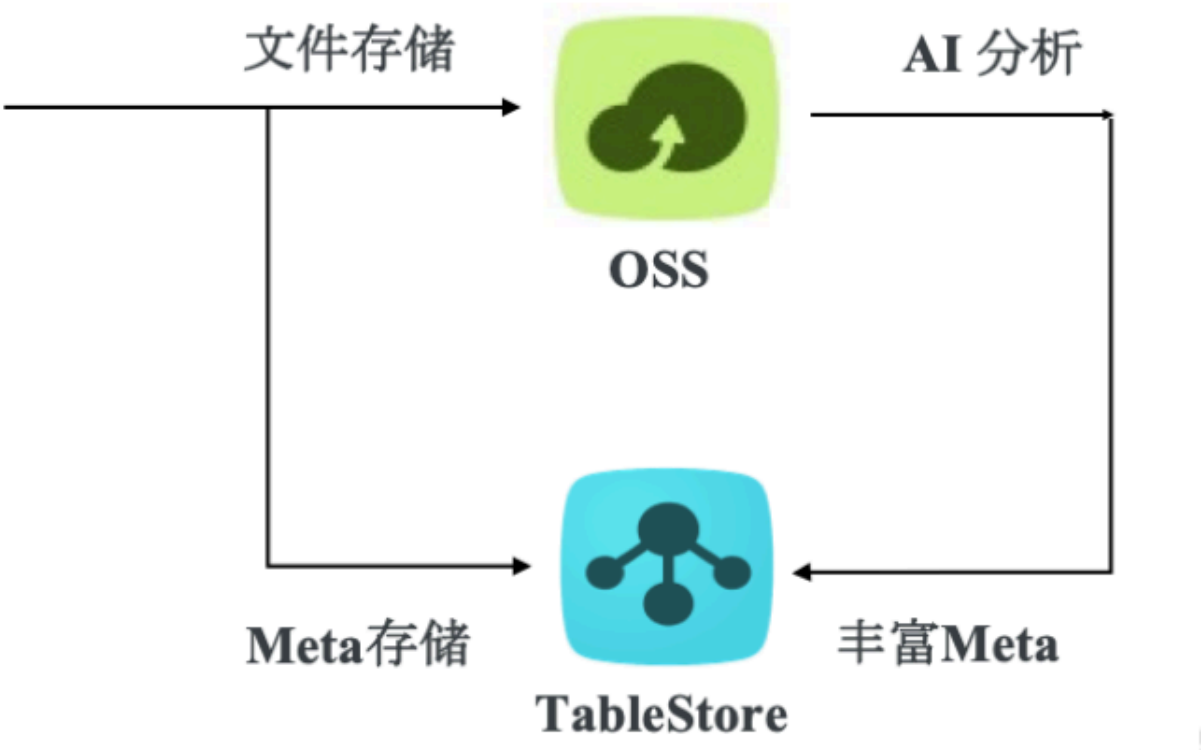
项目样例，如下所示：




## 技术点

对于智能元数据管理系统，通常需要考虑的技术点，包含以下方面：

- 查询能力：具备强大的查询能力，如多类型索引、多维度组合查询等，同时具备排序、统计等功能。
- 横向扩展（多字段）：元数据的字段类型丰富，字段变动、增删频繁，数据库尽量schema free来保证横向扩展能力。
- 纵向扩展（数据量）：海量文件就会对应海量元数据，面对数据膨胀，数据库要满足易扩展、低成本等基本要求。
- 服务性能：应对高并发请的同时，保证低延迟、强一致、高可用。




表格存储方案

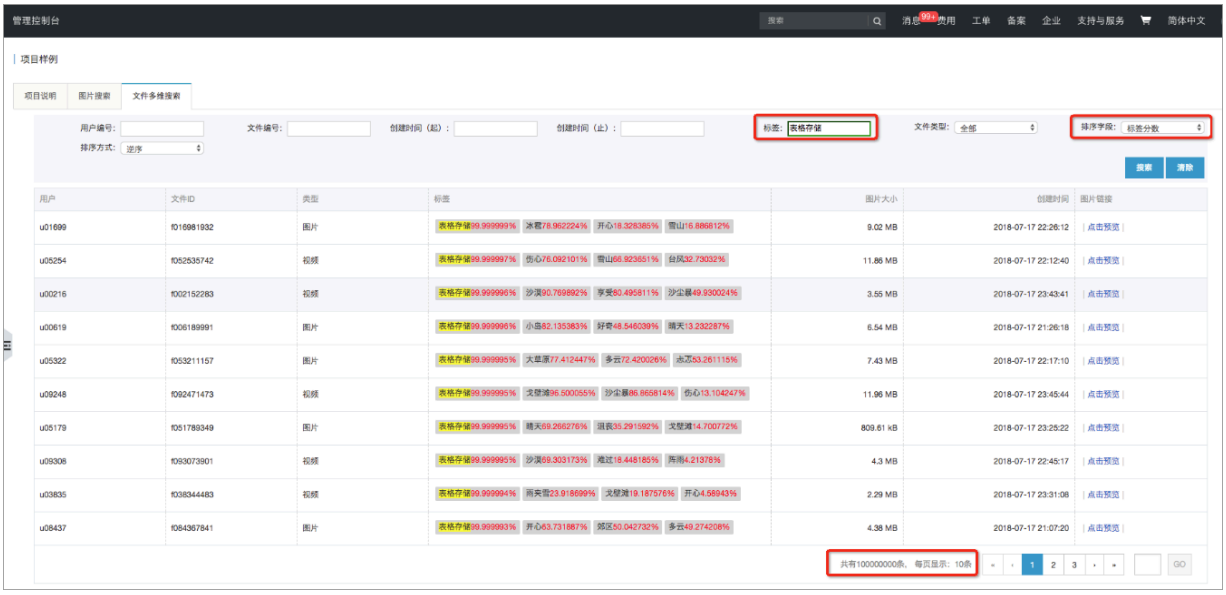
使用表格存储研发的多元索引（SearchIndex）方案，可以有效解决海量元数据的管理问题。表格存储具有即开即用，按量收费等特点。

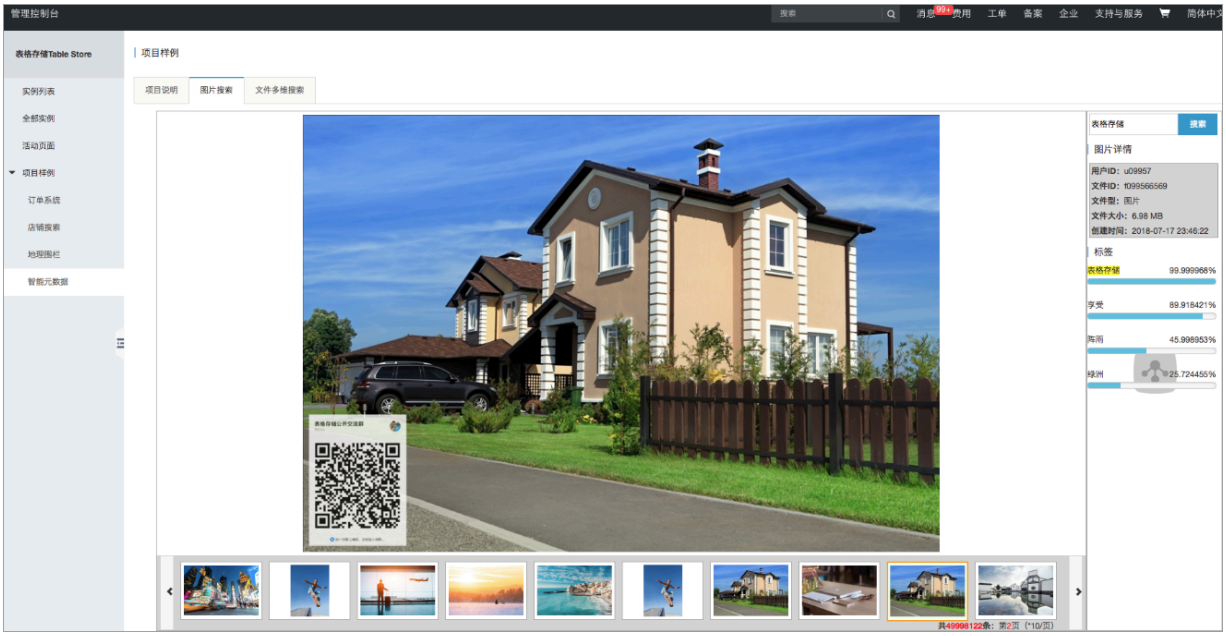
表格存储作为阿里云提供的一款全托管、分布式NoSql型数据存储服务，具有【海量数据存储】、【热点数据自动分片】、【海量数据多维检索】等功能，天然地解决了数据大爆炸这一挑战；在应对数据横向、纵向扩展上，充分发乎其优势。多元索引随时创建，是Meta元数据管理的合适方案。同时，SearchIndex功能在保证用户数据高可用的基础上，提供了数据多维度搜索、统计等能力。针对多种场景创建多种索引，实现多种模式的检索。用户可以仅在需要的时候创建、开通索引。由表格存储来保证数据同步的一致性，这极大的降低了用户的方案设计、服务运维、代码开发等工作量。

基于表格存储搭建的智能元数据管理系统样例

样例内嵌在表格存储控制台中，用户可登录控制台体验系统（若为表格存储的新用户，需要点击开通服务后体验，开通免费，Meta数据存储在公共实例中，体验不消耗用户存储、流量、CU）。


说明:
该样例提供了【亿量级】文件元数据，具体参见项目样例。





## 5.2 准备工作

本章节主要为您介绍使用表格存储（Tablestore）搭建海量智能元数据管理系统前的准备工作。

### 开通表格存储

具体操作步骤参见#unique_12。

### 创建表格存储实例

#unique_13（Instance）是您使用和管理表格存储服务的实体，每个实例相当于一个数据库。表格存储对应用程序的访问控制和资源计量都在实例级别完成。创建表格存储实例的具体操作步骤参见#unique_14。

### 下载SDK

选择以下语言的SDK包进行下载安装：

- Java SDK
- Python SDK
- Go SDK
- Node.js SDK
- .NET SDK
- PHP SDK

### 设计数据表

表名：order_contract

| 列名 | 数据类型 | 索引类型 | 字段说明 |
| --- | --- | --- | --- |
| _id(主键列) | String | | MD5(fId)避免热点 |
| fId | String | KEYWORD | 文件编号 |
| userId | String | KEYWORD | 用户编号 |
| tags | String | Nested: [{ tag: String, score: LONG}] | 多标签使用嵌套索引（数组字符串） '[{"tag":"表格存储","score":97.317251}，{"score":50.770918,"tag":"沙漠"}]' |
| size | long | LONG | 文件大小 |
| createdAt | long | LONG | 创建时间（时间戳） |
| url | String | KEYWORD | 文件链接（存储于oss） |
| ... | ... | ... | ... |

## 5.3 方案实现

本章节主要为您介绍如何使用表格存储（Tablestore）搭建海量智能元数据管理系统。

### 前提条件

您已经完成了准备工作。

### 步骤一 创建数据表

通过以下两种方式创建一张店铺信息表：

- 通过控制台创建、管理数据表，具体参见#unique_22。
- 通过SDK直接创建、管理数据表。

### 步骤二 创建数据表索引

Tablestore自动进行全量、增量的索引数据同步。您可以通过以下两种方式创建、管理多元索引：

- 通过控制台创建、管理多元索引，具体参见#unique_23。
- 通过SDK创建、管理多元索引。

### 步骤三 导入数据

控制台项目样例中插入了1亿条数据，您可以通过控制台插入少量测试数据。

| 文件编号 | 文件ID（md5主键） | 用户编号 | 标签（数组字符串） | 类型 | 链接 | 大小 |
| --- | --- | --- | --- | --- | --- | --- |
| f052535742 | 1bce.... | u052554 | [{"score":99.999999,"tag":"表格存储"},{"score":78.962224,"tag":"冰雹"},{"score":18.328385,"tag":"开心"},{"score":16.886812,"tag":"雪山"}] | image | https://prd-console-demo.oss-cn-hangzhou.aliyuncs.com/image/imm1.jpg | 902206 6 |

### 读取数据

数据读取分为两类：

- 主键读取

  基于原生表格存储的主键列获取：getRow, getRange, batchGetRow等。主键读取用于索引（自动）反查，用户也可以提供主键（文件编号md5）的单条查询的页面，亿量级下查询速度保持在十毫秒量级。单主键查询方式不支持多维度检索。

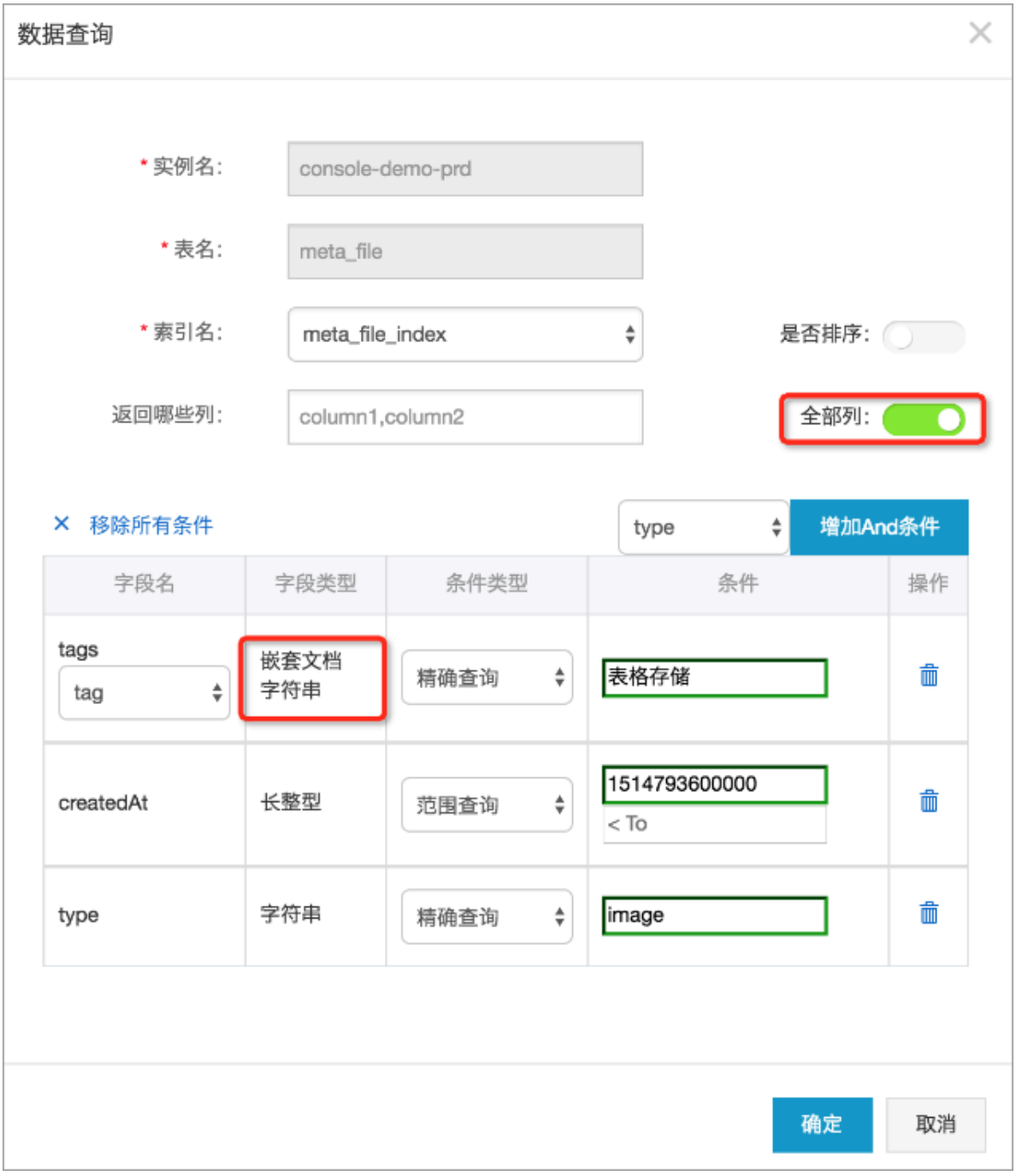
- 索引读取

  基于新SearchIndex功能Query：search接口。用户可以自由设计索引字段的多维度条件组合查询。通过设置选择不同的查询参数，构建不同的查询条件、不同排序方式；目前支持：精确查询、范围查询、前缀查询、匹配查询、通配符查询、短语匹配查询、分词字符串查询、嵌套查询、GEO查询，并通过布尔与、或组合。

  如【标签为：表格存储，创建时间[2018-01-01, 2018-12-01)】文件的信息：（SDK与控制查询）

```
List<Query> mustQueries = new ArrayList<Query>();

//嵌套字段Query
TermQuery termQuery = new TermQuery();
termQuery.setFieldName("tags.tag");
termQuery.setTerm(ColumnValue.fromString("表格存储"));

NestedQuery nestedQuery = new NestedQuery();
nestedQuery.setPath("tags");
nestedQuery.setScoreMode(ScoreMode.Avg);
nestedQuery.setQuery(termQuery);
mustQueries.add(nestedQuery);

//范围Query
RangeQuery rangeQuery = new RangeQuery();
rangeQuery.setFieldName("createdAt");
rangeQuery.setFrom(ColumnValue.fromLong(1514793600000, true);
rangeQuery.setTo(ColumnValue.fromLong(1543651200000, false);
mustQueries.add(rangeQuery);

//精确Query
TermQuery termQuery = new TermQuery();
termQuery.setFieldName("type");
termQuery.setTerm(ColumnValue.fromString("image"));
mustQueries.add(termQuery);

BoolQuery boolQuery = new BoolQuery();
```

```
boolQuery.setMustQueries(mustQueries);
```




## 专家服务

**表格存储提供专业的免费的技术咨询服务，欢迎加入我们的钉钉讨论群。**


表格存储公开交流群
1023人
扫一扫群二维码，立刻加入该群。

# 6 现代IM系统中的消息系统

## 6.1 现代IM系统中的消息系统—架构

本章节主要介绍现代IM系统中的消息系统架构以及基于表格存储（Tablestore）自研的Timeline模型构建的消息系统。基于Timeline构建的现代消息系统能够同时支持消息系统的多种特性，包括多端同步、消息漫游和在线检索，在性能和规模上能够实现全量消息云端存储和索引、百万TPS写入以及毫秒级延迟的消息同步和检索能力。

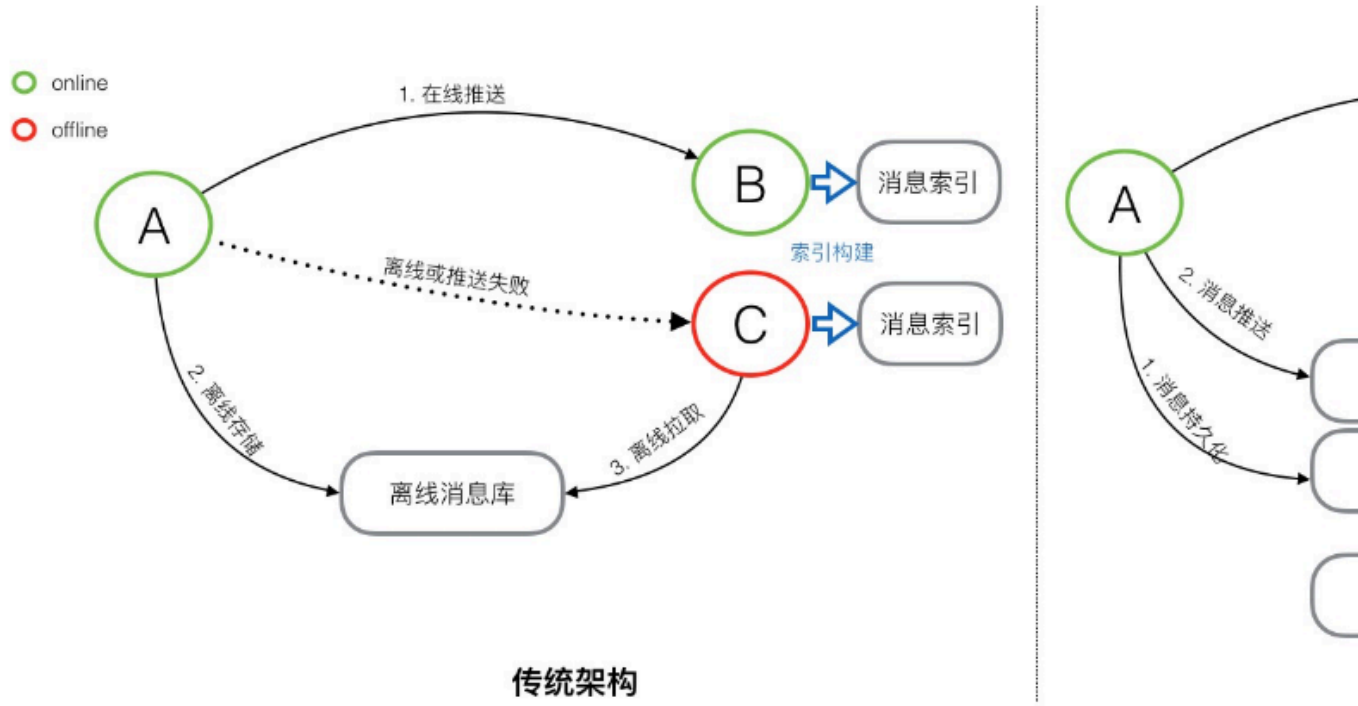
背景

在这个高度信息化的移动互联网时代，IM（Instant Messaging，即时通讯）类产品已经成为生活必备品，例如钉钉、微信、QQ。微信已经成长为一个生态型产品，但其核心功能仍是IM。此外，还有一些非以IM系统为核心的应用也经常出现在我们的娱乐社交生活，例如在线游戏、社交应用。IM系统已经是任何一个带有社交属性的应用需要具备的基础功能。

IM系统在互联网初期即存在，其基础技术架构在这十几年的发展中更新迭代多次，从早期的CS、P2P架构，到现在后台已经演变为一个复杂的分布式系统，涉及移动端、网络通信、协议、安全、存储和搜索等技术的方方面面。IM系统中最核心的部分是消息系统，消息系统中最核心的功能是消息同步、存储和检索。

- 消息同步：指将消息完整的、快速的从发送方传递到接收方。消息同步系统最重要的衡量指标就是消息传递的实时性、完整性以及能支撑的消息规模。在功能上，至少要支持在线和离线推送，有些IM系统还支持多端同步。
- 消息存储：指消息的持久化保存。传统消息系统通常只能支持消息在接收端的本地存储，数据基本不具备可靠性。现代消息系统能支持消息在服务端的在线存储，功能上对应的就是消息漫游，消息漫游的好处是可以实现账号在任意端登录查看所有历史消息。
- 消息检索：消息一般是文本，所以支持全文检索也是必备的能力之一。传统消息系统通常来说也是只能支持消息的本地检索，基于本地存储的消息数据来构建。而现在消息系统在能支持消息的在线存储后，也具备了消息的在线检索能力。

传统架构 vs 现代架构




传统架构

- 传统架构

  传统架构下，消息是先同步后存储。对于在线的用户，消息会直接实时同步到在线的接收方，消息同步成功后，并不会在服务端持久化。而对于离线的用户或者消息无法实时同步成功时，消息会持久化到离线库，当接收方重新连接后，会从离线库拉取所有未读消息。当离线库中的消息成功同步到接收方后，消息会从离线库中删除。传统的消息系统，服务端的主要工作是维护发送方和接收方的连接状态，并提供在线消息同步和离线消息缓存的能力，保证消息一定能够从发送方传递到接收方。服务端不会对消息进行持久化，所以也无法支持消息漫游。消息的持久化存储及索引同样只能在接收端本地实现，数据可靠性极低。

- 现代架构

  现代架构下，消息是先存储后同步。先存储后同步的好处是，如果接收方确认接收到了消息，那这条消息一定是已经在云端保存了。并且消息会有两个库来保存，一个是消息存储库，用于全量保存所有会话的消息，主要用于支持消息漫游。另一个是消息同步库，主要用于接收方的多端同步。消息从发送方发出后，经过服务端转发，服务端会先将消息保存到消息存储库，后保存到消息同步库。完成消息的持久化保存后，对于在线的接收方，会直接选择在线推送。但在线推送并不是一个必须路径，只是一个更优的消息传递路径。对于在线推送失败或者离线的接收方，会有另外一个统一的消息同步方式。接收方会主动的向服务端拉取所有未同步消息，但接收方何时来同步以及会在哪些端来同步消息对服务端来说是未知的，所以要求服务端必须保存所有需要同步到接收方的消息，这是消息同步库的主要作用。对于新的同步设备，会有消息漫游的需求，这是

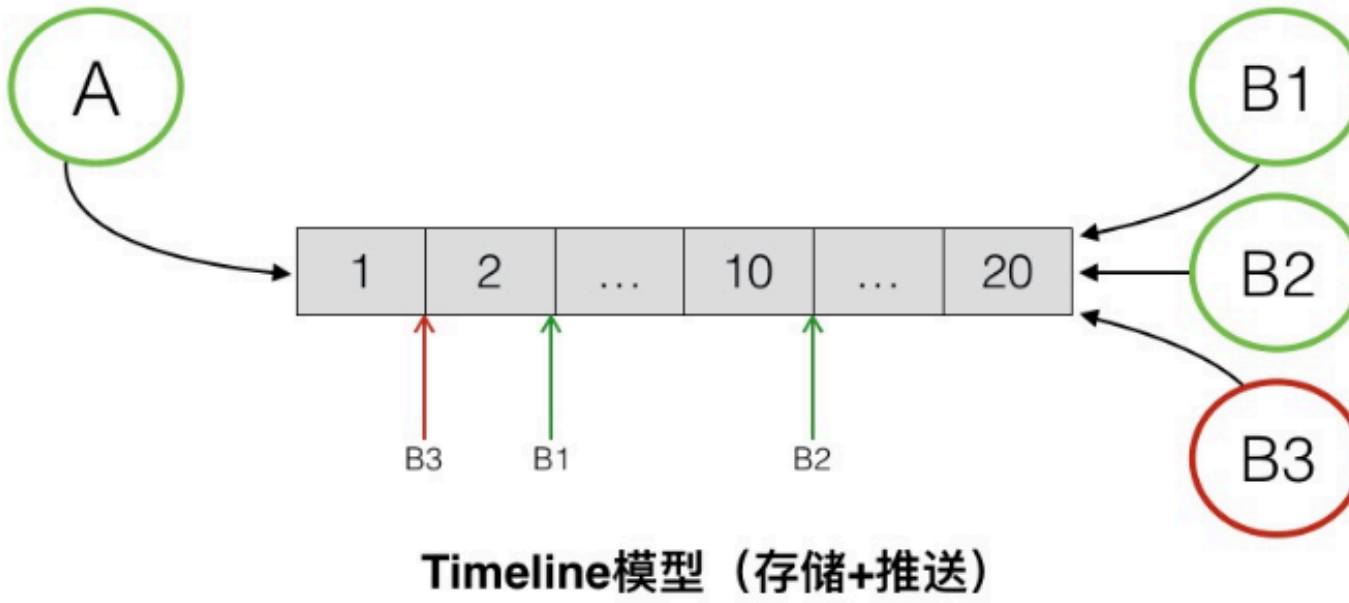
消息存储库的主要作用，在消息存储库中，可以拉取任意会话的全量历史消息。消息检索的实现依赖于对消息存储库内消息的索引，通常是一个近实时（NRT，near real time）的索引构建过程，这个索引同样是在线的。

以上是传统架构和现代架构的一个简单的对比。现代架构上整个消息的同步、存储和索引流程，并没有变复杂太多。现代架构的实现本质上是把传统架构内本地存储和索引都搬到云上，最大挑战是需要集中管理全量消息的存储和索引，优点是能实现多端同步、消息漫游以及在线检索。现代架构中最核心的就是两个消息库：消息同步库、消息存储库，以及对这两个消息库的实现。

### Timeline模型

在深入讲解消息系统的设计和实现之前，需要先了解消息系统内的基本概念和基础模型。表格存储的Timeline是一个对消息系统内消息模型的一个抽象，能简化和更好的让开发者理解消息系统内的消息同步和存储模型，基于此模型我们会再深入探讨消息的同步和存储的选择和实现。

Timeline模型是一个对消息抽象的逻辑模型，该模型会帮助我们简化对消息同步和存储模型的理解，而消息同步库和存储库的设计和实现也是围绕Timeline的特性和需求来展开。




Timeline模型（存储+推送）

上图是Timeline模型的一个抽象表述，Timeline可以简单理解为是一个消息队列，但这个消息队列有如下特性。

- 每条消息对应一个顺序ID：每个消息拥有一个唯一的顺序ID（SequenceId），队列消息按SequenceId排序。
- 新消息写入能自动分配递增的顺序ID，保证永远插入队尾：Timeline中是根据同步位点也就是顺序ID来同步消息，所以需要保证新写入的消息数据的顺序ID绝对不能比已同步的消息的顺序

ID还小，否则会导致数据漏同步，所以需要支持对新写入的数据自动分配比当前已存储的所有消息的顺序ID更大的顺序ID。

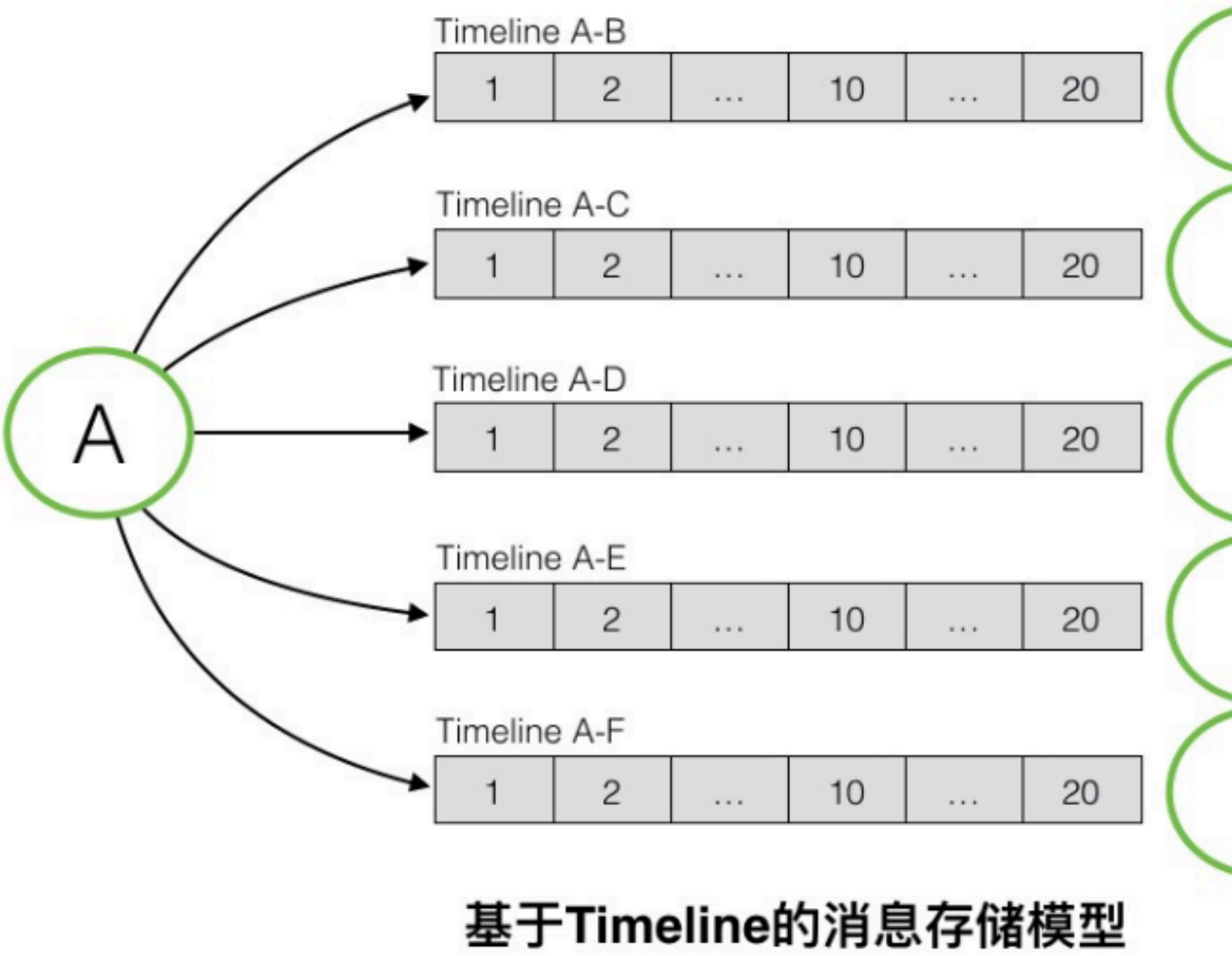
- 新消息写入也能自定义顺序ID，满足自定义排序需求：上面提到的自动分配顺序ID，主要是为了满足消息同步的需求，消息同步要求消息是根据『已同步』或是『已写入』的顺序来排序。而消息的存储，通常要求消息能根据会话顺序来排序，会话顺序通常由端的会话来决定，而不是服务端的同步顺序来定，这是两种顺序要求。
- 支持根据顺序ID的随机定位：可根据SequenceId随机定位到Timeline中的某个位置，从这个位置开始正序或逆序的读取消息，也可支持读取指定顺序ID的某条消息。
- 支持对消息的自定义索引：消息体内数据根据业务不同会包含不同的字段，Timeline需要支持对不同字段的自定义索引，来支持对消息内容的全文索引，或者是任意字段的灵活条件组合查询。

消息同步可以基于Timeline很简单的实现。图中的例子中，消息发送方是A，消息接收方是B，同时B存在多个接收端，分别是B1、B2和B3。A向B发送消息，消息需要同步到B的多个端，待同步的消息通过一个Timeline来进行交换。A向B发送的所有消息，都会保存在这个Timeline中，B的每个接收端都是独立的从这个Timeline中拉取消息。每个接收端同步完毕后，都会在本地记录下最新同步到的消息的SequenceId，即最新的一个位点，作为下次消息同步的起始位点。服务端不会保存各个端的同步状态，各个端均可以在任意时间从任意点开始拉取消息。

消息存储也是基于Timeline实现，和消息同步唯一的区别是，消息存储要求服务端能够对Timeline内的所有数据进行持久化，并且消息采用会话顺序来保存，需要自定义顺序ID。

消息检索基于Timeline提供的消息索引来实现，能支持比较灵活的多字段索引，根据业务的不同可有自由度较高的定制。

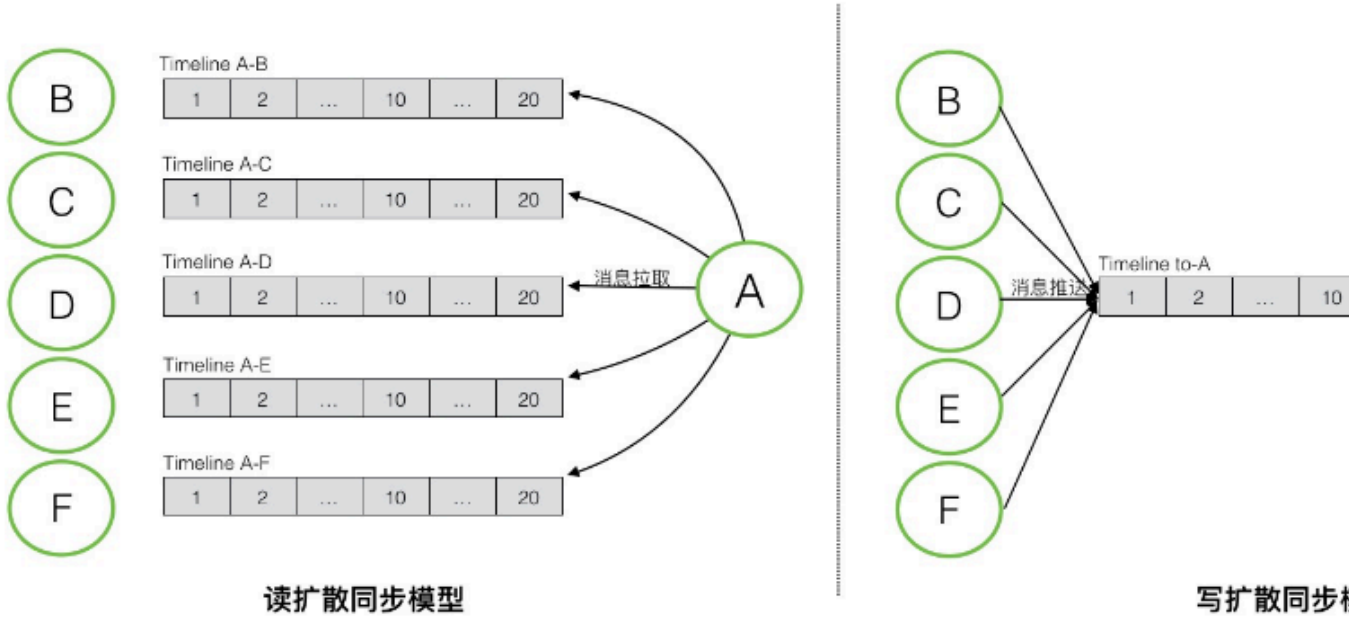
消息存储模型




上图是基于Timeline的消息存储模型，消息存储要求每个会话都对应一个独立的Timeline。如图例子所示，A与B/C/D/E/F均发生了会话，每个会话对应一个独立的Timeline，每个Timeline内存有这个会话中的所有消息，消息根据会话顺序排序，服务端会对每个Timeline进行持久化存储，也就拥有了消息漫游的能力。

消息同步模型

消息同步模型会比消息存储模型稍复杂一些，消息的同步一般有读扩散（也叫拉模式）和写扩散（也叫推模式）两种不同的方式，分别对应不同的Timeline物理模型。



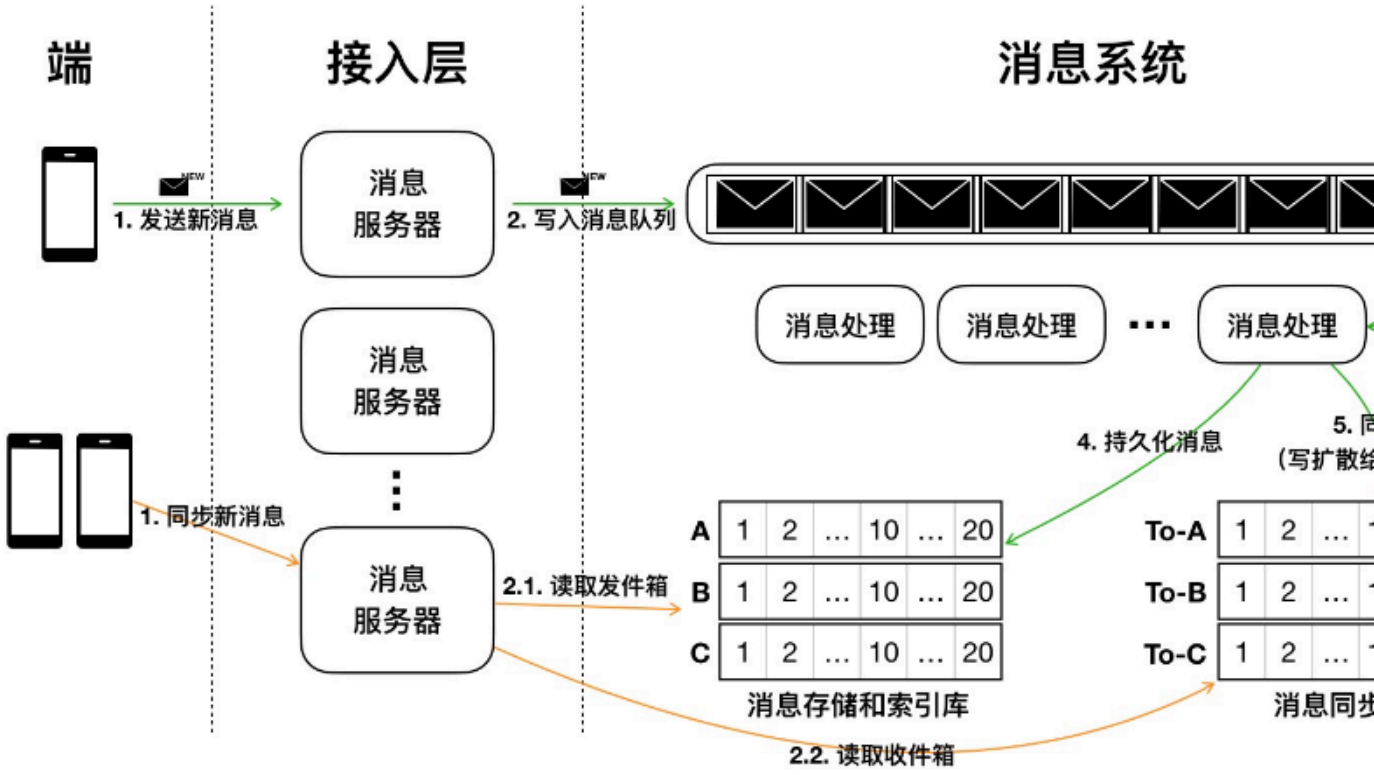

如图是读扩散和写扩散两种不同同步模式下对应的不同的Timeline模型。按图中的示例，A作为消息接收者，其与B/C/D/E/F发生了会话，每个会话中的新的消息都需要同步到A的某个端。

- 读扩散：消息存储模型中，每个会话的Timeline中保存了这个会话的全量消息。读扩散的消息同步模式下，每个会话中产生的新的消息，只需要写一次到其用于存储的Timeline中，接收端从这个Timeline中拉取新的消息。优点是消息只需要写一次，相比写扩散的模式，能够大大降低消息写入次数，特别是在群消息这种场景下。但其缺点也比较明显，接收端去同步消息的逻辑会相对复杂和低效。接收端需要对每个会话都拉取一次才能获取全部消息，读被大大的放大，并且会产生很多无效的读，因为并不是每个会话都会有新消息产生。
- 写扩散：写扩散的消息同步模式，需要有一个额外的Timeline来专门用于消息同步，通常是每个接收端都会拥有一个独立的同步Timeline（或者叫收件箱），用于存放需要向这个接收端同步的所有消息。每个会话中的消息，会产生多次写，除了写入用于消息存储的会话Timeline，还需要写入需要同步到的接收端的同步Timeline。在个人与个人的会话中，消息会被额外写两次，除了写入这个会话的存储Timeline，还需要写入参与这个会话的两个接收者的同步Timeline。而在群这个场景下，写入会被更加的放大，如果这个群拥有N个参与者，那每条消息都需要额外的写N次。写扩散同步模式的优点是，在接收端消息同步逻辑会非常简单，只需要从其同步Timeline中读取一次即可，大大降低了消息同步所需的读的压力。其缺点就是消息写入会被放大，特别是针对群这种场景。

  Timeline模型不会对选择读扩散还是写扩散做约束，而是能同时支持两种模式，因为本质上两种模式的逻辑数据模型并无差别，只是消息数据是用一个Timeline来支持多端读还是复制到多个Timeline来支持多端读的问题。

针对IM这种应用场景，消息系统通常会选择写扩散这种消息同步模式。IM场景下，一条消息只会产生一次，但是会被读取多次，是典型的读多写少的场景，消息的读写比例大概是10:1。若使用读扩散同步模式，整个系统的读写比例会被放大到100:1。一个优化的好的系统，必须从设计上去平衡这种读写压力，避免读或写任意一维触碰到天花板。所以IM系统这类场景下，通常会应用写扩散这种同步模式，来平衡读和写，将100:1的读写比例平衡到30:30。当然写扩散这种同步模式，还需要处理一些极端场景，例如万人大群。针对这种极端写扩散的场景，会退化到使用读扩散。一个简单的IM系统，通常会在产品层面限制这种大群的存在，而对于一个高级的IM系统，会采用读写扩散混合的同步模式，来满足这类产品的需求。采用混合模式，会根据数据的不同类型和不同的读写负载，来决定用写扩散还是读扩散。

典型架构设计




上图是一个典型的消息系统架构，架构中包含几个重要组件。

- 端：作为消息的发送和接收端，通过连接消息服务器来发送和接收消息。
- 消息服务器：一组无状态的服务器，可水平扩展，处理消息的发送和接收请求，连接后端消息系统。
- 消息队列：新写入消息的缓冲队列，消息系统的前置消息存储，用于削峰填谷以及异步消费。
- 消息处理：一组无状态的消费处理服务器，用于异步消费消息队列中的消息数据，处理消息的持久化和写扩散同步。
- 消息存储和索引库：持久化存储消息，每个会话对应一个Timeline进行消息存储，存储的消息建立索引来实现消息检索。

- 消息同步库：写扩散形式同步消息，每个用户的收件箱对应一个Timeline，同步库内消息不需要永久保存，通常对消息设定一个生命周期。

  新消息会由端发出，通常消息体中会携带消息ID（用于去重）、逻辑时间戳（用于排序）、消息类型（控制消息、图片消息或者文本消息等）、消息体等内容。消息会先写入消息队列，作为底层存储的一个临时缓冲区。消息队列中的消息会由消息处理服务器消费，可以允许乱序消费。消息处理服务器对消息先存储后同步，先写入发件箱Timeline（存储库），后写扩散至各个接收端的收件箱（同步库）。消息数据写入存储库后，会被近实时的构建索引，索引包括文本消息的全文索引以及多字段索引（发送方、消息类型等）。

对于在线的设备，可以由消息服务器主动推送至在线设备端。对于离线设备，登录后会主动向服务端同步消息。每个设备会在本地保留有最新一条消息的顺序ID，向服务端同步该顺序ID后的所有消息。

### 专家服务

表格存储提供专业的免费的技术咨询服务，欢迎加入我们的钉钉讨论群。


## 6.2 现代IM系统中的消息系统—模型

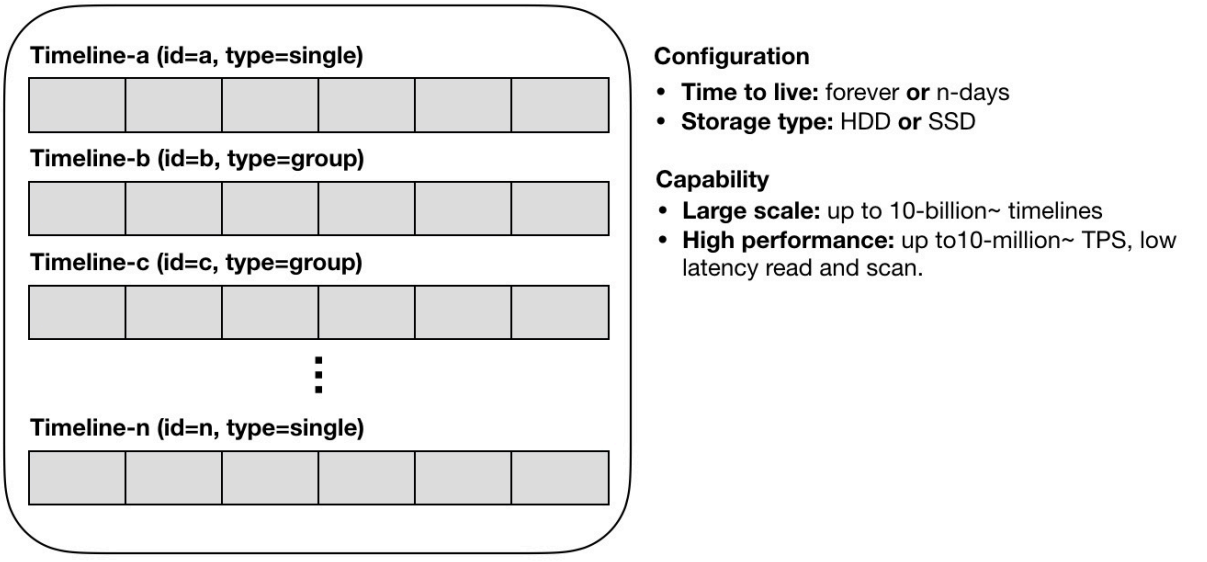
本章节主要介绍表格存储（Tablestore）的Timeline模型基本功能以及核心组件，并且会通过IM消息系统场景，介绍如何基于Timeline实现IM场景下消息同步、存储和索引等基本功能。

### Timeline模型

Timeline模型以简单为设计目标，核心模块主要包括：

| 模块 | 说明 |
| --- | --- |
| Store | Timeline存储库，类似数据库表的概念。 |
| Identifier | 用于区分Timeline的唯一标识。 |
| Meta | 用于描述Timeline的元数据，元数据描述采用free-schema结构，可自由包含任意列。 |
| Queue | 一个Timeline内所有Message存储在Queue内。 |
| Message | Timeline内传递的消息体，也是一个free-schema的结构，可自由包含任意列。 |
| Index | 包含Meta Index和Message Index，可对Meta或Message内的任意列自定义索引，提供灵活的多条件组合查询和搜索。 |

Timeline Store




Timeline Store

Timeline Store是Timeline的存储库，对应于数据库内表的概念。上图是Timeline Store的结构图，Store内会存储所有的Timeline数据。Timeline是一个面向海量消息的数据模型，同时用于消息存储库和同步库，需要满足多种要求。

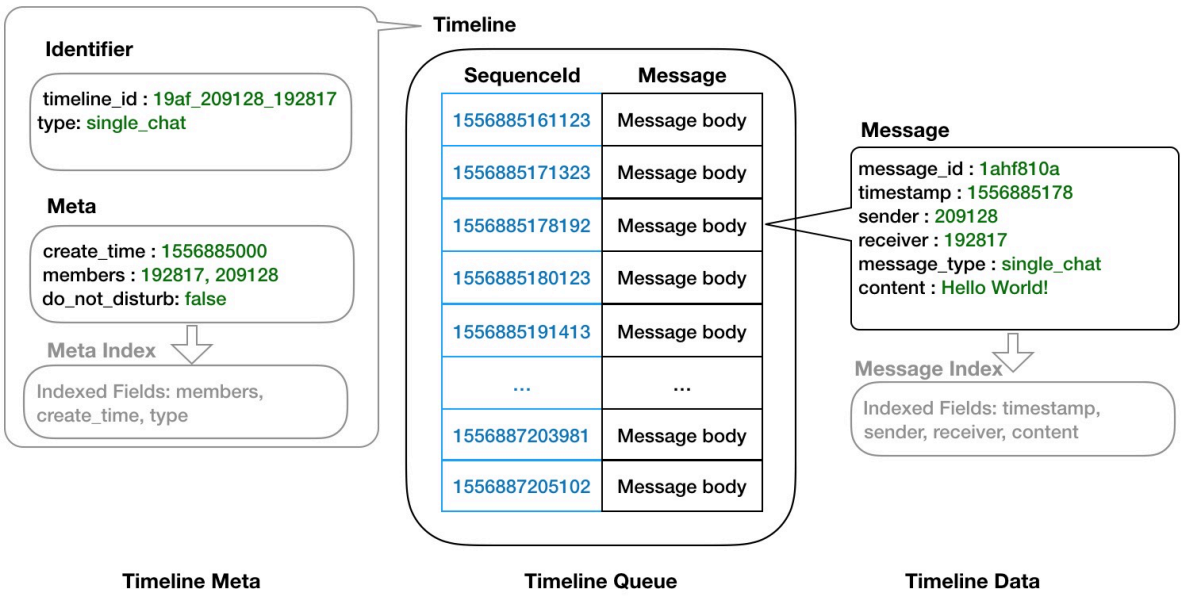
- 支撑海量数据存储：对于消息存储库来说，如果需要消息永久存储，则随着时间的积累，数据规模会越来越大，需要存储库能应对长时间积累的海量消息数据存储，需要能达到PB级容量。
- 低存储成本：消息数据的冷热区分是很明显的，大部分查询都会集中在热数据，所以对于冷数据需要有一个比较低成本的存储方式，否则随着时间的积累数据量不断膨胀，存储成本会非常大。
- 数据生命周期管理：不管是对于消息数据的存储还是同步，数据都需要定义生命周期。存储库是用于在线存储消息数据本身，通常需要设定一个较长周期的保存时间。而同步库是用于写扩散模式的在线或离线推送，通常设定一个较短的保存时间。

- 极高的写入吞吐：各类场景下的消息系统，除了类似微博、头条这种类型的Feeds流系统，像绝大部分即时通讯或朋友圈这类消息场景，通常是采用写扩散的消息同步模式，写扩散要求底层存储具备极高的写入吞吐能力，以应对消息洪峰。
- 低延迟的读：消息系统通常是应用在在线场景，所以对于查询要求低延迟。

Tablestore Timeline的底层是基于LSM存储引擎的分布式数据库，LSM的最大优势就是对写入非常友好，天然适合消息写扩散的模式。同时对查询也做了极大优化，例如热数据进缓存、bloom filter等等。数据表采用Range Partition的分区模式，能提供水平扩展的服务能力，以及能自动探测并处理热点分区的负载均衡策略。为了满足同步库和存储库对存储的不同要求，也提供了一些灵活的自定义配置，主要包括：

| 配置 | 说明 |
| --- | --- |
| Time to live（数据生命周期） | 可自定义数据生命周期，例如永久保存，或者保存N天。 |
| Storage type（存储类型） | 自定义存储类型，对存储库来说，HDD是最好的选择，对同步库来说，SSD是最好的选择。 |

Timeline Module




Timeline Store内能存储海量的Timeline，单个Timeline的详细结构图如上，可以看到Timeline主要包含了Timeline Meta、Timeline Queue、Timeline Data三大部分。

- Timeline Meta

  元数据部分，用于描述Timeline

  - Identifier：用于唯一标识Timeline，可包含多个字段。
  - Meta：用于描述Timeline的元数据，可包含任意个数任意类型的字段。
  - Meta Index：元数据索引，可对元数据内任意属性列建索引，支持多字段条件组合查询和检索。

- Timeline Queue

  用于存储和同步消息的队列，队列中元素由以下两部分组成。

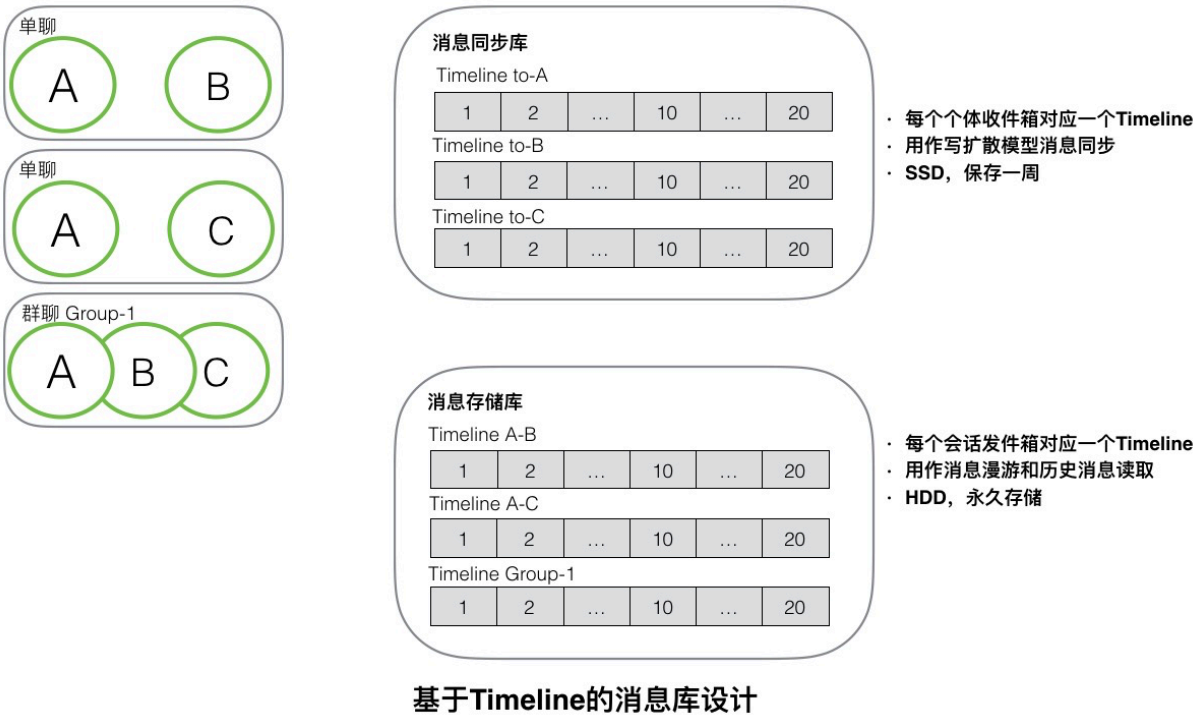
  - Sequence Id：顺序ID，队列中用于定位Message的位点信息，在队列中顺序ID保持递增。
  - Message：队列中承载消息的实体，包含了消息的完整内容。

- Timeline Data

  Timeline的数据部分就是Message，Message主要包含以下两部分。

  - Message：消息实体，其内部也可以包含任意数量任意类型字段。
  - Message Index：消息数据索引，可对消息实体内任意列做索引，支持多字段条件组合查询和检索。

IM消息系统建模




基于Timeline的消息库设计

以一个简易版IM系统为例，来看如何基于Tablestore Timeline模型建模。按照上图中的例子，存在A、B、C三个用户，A与B发生单聊，A与C发生单聊，以及A、B、C组成一个群聊，来看下在这个场景下消息同步、存储以及读写流程分别如何基于Tablestore Timeline建模。

消息同步模型

消息同步选择写扩散模型，能完全利用Tablestore Timeline的优势，以及针对IM消息场景读多写少的特性，通过写扩散来平衡读写，均衡整个系统的资源。写扩散模型下，每个接收消息的个体均拥有一个收件箱，所有需要同步至该个体的消息需要投递到其收件箱内。图上例子中，A、B、C三个用户分别拥有收件箱，每个用户不同的设备端，均从同一个收件箱内拉取新消息。

消息同步库

收件箱存储在同步库内，同步库中每个收件箱对应一个Timeline。根据图上的例子，总共存在3个Timeline作为收件箱。每个消息接收端保存有本地最新拉取的消息的SequenceID，每次拉取新消息均是从该SequenceID开始拉取消息。对同步库的查询会比较频繁，通常是对最新消息的查询，所以要求热数据尽量缓存在内存中，能提供高并发低延迟的查询。所以对同步库的配置，一般是需要SSD存储。消息如果已经同步到了所有的终端，则代表收件箱内的该消息已经被消费完毕，理论上可以清理。但设计上来说不做主动清理，而是给数据定义一个较短的生命周期来自动过期，一般定义为一周或者两周。数据过期之后，如果仍要同步拉取新消息，则需要退化到读扩散的模式，从存储库中拉取消息。

消息存储库

消息存储库中保存有每个会话的消息，每个会话的发件箱对应一个Timeline。发件箱内的消息支持按会话维度拉取消息，例如浏览某个会话内的历史消息则通过读取发件箱完成。一般来说，新消息通过在线推送或者查询同步库可投递到各个接收端，所以对存储库的查询会相对来说较少。而存储库用于长期存储消息，例如永久存储，相对同步库来说数据量会较大。所以存储库的选择一般是HDD，数据生命周期根据消息需要保存的时间来定，通常是一个较长的时间。

消息索引库

消息索引库依附于存储库，使用了Timeline的Message Index，可以对存储库内的消息进行索引，例如对文本内容的全文索引、收件人、发件人以及发送时间的索引等，能支持全文检索等高级查询和搜索。

## 专家服务

表格存储提供专业的免费的技术咨询服务，欢迎加入我们的钉钉讨论群。


## 6.3 现代IM系统中的消息系统—实现

本章节主要为本章节以钉钉（DingTalk）的功能为参照，详细说明如何基于表格存储（Tablestore）的Timeline模型实现钉钉的IM功能。

以下内容按照聊天系统的消息存储、关系维护、即时感知、多端同步这四个功能模块分段，分别介绍每一部分的功能、方案介绍、表设计以及实现代码等。

### 功能模块

#### 消息存储

消息系统中，消息存储是最基本的功能。对于消息存储（提供消息的读、写、持久化），一方面需要持久化写入，保证消息数据的不丢失，另一方面，适合用户的快速、高效查询。在IM场景中，写入方式通常是单行、批量写入，而读取需要按照消息队列范围读取。有时用户还有对于历史消息的模糊查询需求，这时就需要使用多维检索、全文检索的能力。

消息的存储都是基于Timeline模型，具体模型参见Tablestore发布Timeline 2.0模型。

表设计：im_timeline_store_table

数据源：im_timeline_store_table　　表格数据最多显示50行。

| 详细数据 | timeline_id(主键) | sequence_id(主键) | conversation | send_time | sender | text | type |
|---|---|---|---|---|---|---|---|
| 详细数据 | group_1 | 1563422738697000 | group_1 | 1563422738694 | user_1 | 今天谁使用表格存储了 | GROUP |
| 详细数据 | group_1 | 1563422738781000 | group_1 | 1563422738779 | user_2 | 表格+1 | GROUP |
| 详细数据 | group_1 | 1563422738871000 | group_1 | 1563422738868 | user_3 | 表格+1 | GROUP |
| 详细数据 | group_1 | 1563422738982000 | group_1 | 1563422738979 | user_4 | 表格+1 | GROUP |
| 详细数据 | single_2 | 1563422738479000 | user_2 | 1563422738424 | user_2 | 我们是好朋友了2 | SINGLE |
| 详细数据 | single_3 | 1563422738618000 | user_3 | 1563422738614 | user_3 | 我们是好朋友了3 | SINGLE |
| 详细数据 | single_4 | 1563422738646000 | user_4 | 1563422738643 | user_4 | 我们是好朋友了4 | SINGLE |

共有7条，每页显示：10条

### 存储库

**功能：会话窗口消息展示**

存储库是聊天会话消息所对应的存储表，消息以会话分类存储，每个会话是一个消息队列。单个消息队列（TimelineQueue）通过timelineId唯一标识，所有消息基于sequenceId有序排列。消息体中含有发送人、消息id（消息去重）、消息发送时间、消息体内容、消息类型（类型包含图片、文件、普通文本，本文仅适用文本）等。


如上图，当用户点击某一个会话时，窗口会展示相应会话的最新一页消息。图片里的消息都是从存储库拉取的，通过timelineId获取该会话的Queue实例，然后调用Queue的scan接口与ScanParam参数（sequenceId范围+倒序）拉取最新的一页消息。当用户向上滚动，展示完这一页消息后，客户端会基于第一次请求的最小sequncId发起第二次请求，获取第二页消息记录，单页消息数通常选择20-30条。会话的消息可以选择在客户端持久化，然后在感知到新消息之后更新本地消息，增加缓存减少网络IO。

**核心代码**

```
    public List<AppMessage> fetchConversationMessage(String timelineId
, long sequenceId) {
        TimelineStore store =  timelineV2.getTimelineStoreTabl
eInstance();

        TimelineIdentifier identifier = new TimelineIdentifier.Builder
()
                .addField("timeline_id", timelineId)
                .build();

        ScanParameter parameter = new ScanParameter()
                .scanBackward(sequenceId)
                .maxCount(30);

        Iterator<TimelineEntry> iterator = store.createTimelineQueue(
identifier).scan(parameter);

        List<AppMessage> appMessages = new LinkedList<AppMessage>();
        while (iterator.hasNext() && counter++ <= 30) {
            TimelineEntry timelineEntry = iterator.next();
            AppMessage appMessage = new AppMessage(timelineId,
timelineEntry);

            appMessages.add(appMessage);
        }

        return appMessages;
    }
```

```
[Scan Conversation Message]: -> group_1
    [user_4 say @2019-07-17 21:26:42]: 表格+1
    [user_3 say @2019-07-17 21:26:42]: 表格+1
    [user_2 say @2019-07-17 21:26:42]: 表格+1
    [user_1 say @2019-07-17 21:26:42]: 今天谁使用表格存储了
    [Total Count]: 4, [Update SyncCheckpoint]: 1563370002181000
```

**存储库的消息需要永久保存，是整个应用的全量消息存储。存储库数据过期时间（TTL）需要设为-1。**

**功能：多维组合、全文检索**

**全文检索能力就是对存储库的消息内容做模糊查询，因而需要对存储库的数据建立多元索引。具体索引字段，需要根据设计需求设计。如钉钉公开群的检索，需要对群ID、消息发送人、消息类型、消息内容、以及时间建立索引，其中消息内容需要使用分词字符串类型，从而提供模糊查询的能力。**


**核心代码**

```
public List<AppMessage> fetchConversationMessage(String timelineId,
long sequenceId) {
    TimelineStore store =  timelineV2.getTimelineStoreTableInstance();

    TimelineIdentifier identifier = new TimelineIdentifier.Builder()
            .addField("timeline_id", timelineId)
            .build();

    ScanParameter parameter = new ScanParameter()
            .scanBackward(sequenceId)
            .maxCount(30);

    Iterator<TimelineEntry> iterator = store.createTimelineQueue(
identifier).scan(parameter);

    List<AppMessage> appMessages = new LinkedList<AppMessage>();
    int counter = 0;
    while (iterator.hasNext() && counter++ <= 30) {
        TimelineEntry timelineEntry = iterator.next();
        AppMessage appMessage = new AppMessage(timelineId, timelineEn
try);

        appMessages.add(appMessage);
    }

    return appMessages;
```

```
}
```

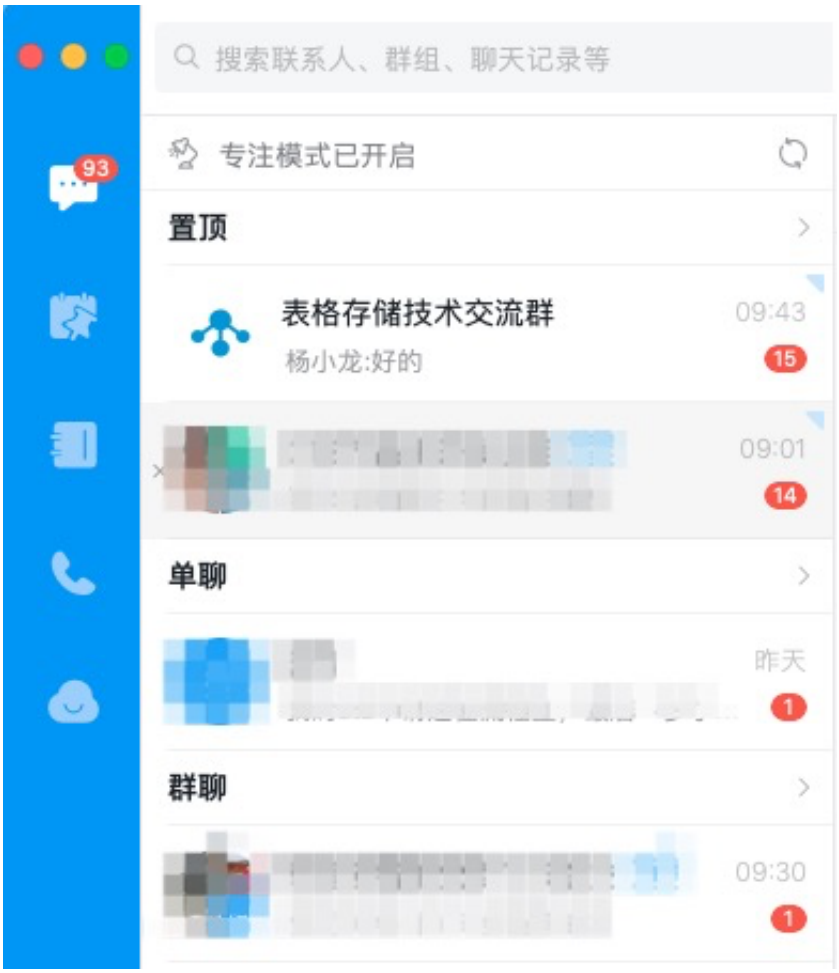
```
[Search Message in Conversation]: -> group_1 -> '表格'
    [user_1 say @2019-07-17 21:26:42]: 今天谁使用表格存储了
    [user_2 say @2019-07-17 21:26:42]: 表格+1
    [user_3 say @2019-07-17 21:26:42]: 表格+1
    [user_4 say @2019-07-17 21:26:42]: 表格+1
    [Total Count]: 4
```

另外，为了做消息的权限管理，仅允许用户检索自己有权限查看的消息，可在消息体字段中扩展接收人ID数组，这样对所有群做检索时，需要增加接收人字段为自己的用户ID这一必要条件，即可实现消息内容的权限限制。 样例中没有实现这一功能，用户可根据需求自己增加、修改。

同步库

功能：新消息即时统计

当客户端在线时，应用的系统服务会维护客户端的长连接，因而可以感知客户端在线。当用户的同步库有新消息写入时（即有新消息），应用会发出信号通知客户端有新消息，然后客户端会基于同步库checkpoint点，拉取同步库中该sequenceId之后的所有新消息，统计各会话的新消息数，并更新checkpoint点。



如上图，对于一个在线客户端，每个会话都会维护一个未读消息的计数（小红点），也会有一个总未读数的计数，这个数量一般会存储在客户端本地，或者通过redis持久化。这些未读消息，指的就是通过同步库拉取并统计过，但是还未被用户点开的消息数量。在拉取到新消息列表后，客户端（或应用层）会遍历所有新消息，然后将新消息所对应会话的未读计数累加1，这样实现了未读消息的即时感知与更新。只有当用户点开会话后，会话的未读计数才会清零。

在更新未读数的同时，会话列表中还会有最新消息的简短摘要信息以及最新消息的发送时间等。这些可以在遍历新消息列表时不断更新。这些统计、摘要都是依托同步库，而非存储库实现的。

核心代码

```
public List<AppMessage> fetchSyncMessage(String userId, long
lastSequenceId) {
    TimelineStore sync =  timelineV2.getTimelineSyncTableInstance();

    TimelineIdentifier identifier = new TimelineIdentifier.Builder()
            .addField("timeline_id", userId)
            .build();

    ScanParameter parameter = new ScanParameter()
            .scanForward(lastSequenceId)
            .maxCount(30);

    Iterator<TimelineEntry> iterator = sync.createTimelineQueue(
identifier).scan(parameter);

    List<AppMessage> appMessages = new LinkedList<AppMessage>();
    int counter = 0;
    while (iterator.hasNext() && counter++ <= 30) {
        AppMessage appMessage = new AppMessage(userId, iterator.next
());
        appMessages.add(appMessage);
    }

    return appMessages;
}
```

在统计到会话列表中不存在的会话时，客户端会做一次额外请求。通过timelineID获取会话的基本描述信息，如群头像或好友的头像、群名称等，并初始化未读数计时器0，然后累加新消息数、更新最新消息摘要等。

```
[Count Unread Message]
    conversation: user_3, newUnread: 1, latestMessage: 我们是好朋友了3
    conversation: user_2, newUnread: 1, latestMessage: 我们是好朋友了2
    conversation: user_4, newUnread: 1, latestMessage: 我们是好朋友了4
    conversation: group_1, newUnread: 4, latestMessage: 表格+1
    [Total Count]: 7, [Update SyncCheckpoint]: 1563370002491000
```

同步库对于IM场景下的新消息即时感知统计这一核心功能，就是通过写入冗余的方式，提升新消息读取统计的效率与速度。对于IM场景没有收件箱的概念，因而同步库中冗余消息并没有永久保存的

价值，提供7天过期时间已经足够保证功能正常。用户可以根据自身需求，调整同步库的数据过期时间（TTL）。

功能：异步写扩散

在本文的样例中，单聊会话的消息在写完存储库后同时写入了同步库，只有两行的写入开销很小。但是对于群会话，写完存储库后要获取群用户列表，然后依次写入相应用户的同步库。这种方式在群少、用户少时不会有问题，但随着用户体量、活跃度的增加，同步的写的方式就会面临性能问题，因此建议用户对群写扩散使用异步任务实现。

用户可以基于表格存储实现一个任务队列，将写扩散任务写入队列中后直接返回，然后由其他进程保证任务队列的执行。任务队列保存了群ID、消息的完整信息，消费进程不断轮询读取新任务，获取任务后，才会从群关系表中获取完整的群成员列表，并做相应的写扩散。

任务队列可以直接基于Tablestore实现，表设计为两列主键，第一列为topic，第二列为自增列，一个topic对应一个队列，任务会被有序写入单个队列中。当并发量持续膨胀后，可对任务做hash分桶，随机写入多个topic。这样可以增加消费者数量（消费并发量），提升写扩散效率。对应任务队列消费，用户只需要维护每个topic的checkpoint点。checkpoint点之前的为已完成任务，通过getRange的方式顺序获取checkpoint点之后未执行的新任务，保证任务的执行。失败的任务可以重新写入任务队列来提升容错，并增加重试计数。出现多次失败后放弃重写，然后将该任务写入特殊的问题队列，方便应用的开发者们查询、定位问题。

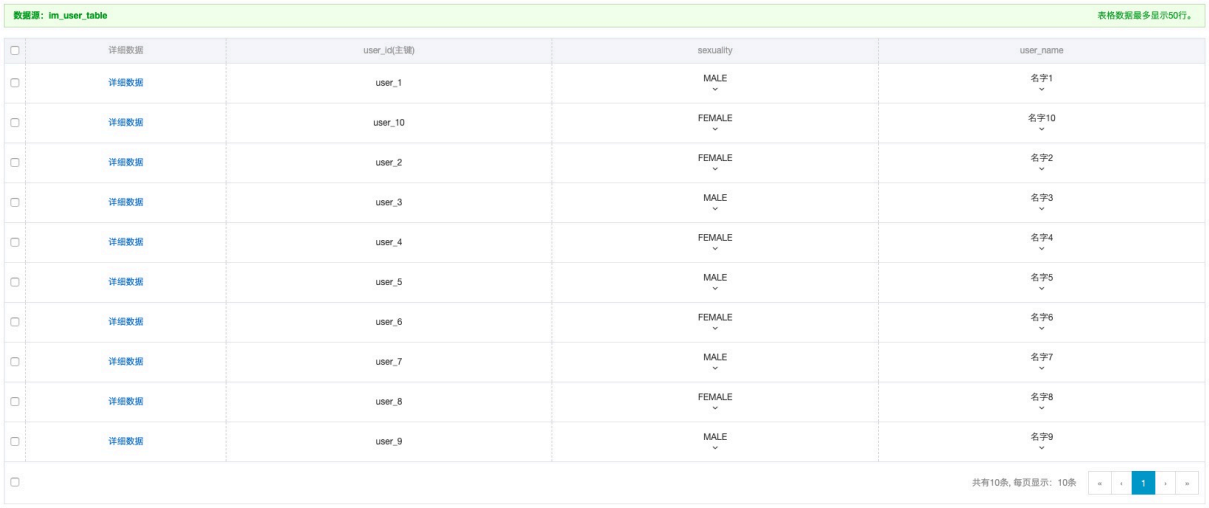
元数据管理

所谓元数据，就是描述数据的数据。在这里主要体现为两类：用户元数据、会话元数据。这里群的元数据信息：群ID（复用群的timelineId）、群名称、创建时间等信息，可以直接基于timelineMeta的管理表完成实现，所有Group类型的TimelineMeta可以映射为一个Group。但是用户的元数据却不能复用TimelineMeta，所以需要单独的表实现。

用户元数据

即用户的属性信息，通过用户ID识别特定用户。在上面提到的用户关系中，通过用户的标识ID确认用户身份，但用户的属性信息，如：性别、签名、头像等信息，还是需要单独维护。因此需要单独维护。

表设计：im_user_table



数据源：im_user_table　　表格数据最多显示50行。

| 详细数据 | user_id(主键) | sexuality | user_name |
| --- | --- | --- | --- |
| 详细数据 | user_1 | MALE | 名字1 |
| 详细数据 | user_10 | FEMALE | 名字10 |
| 详细数据 | user_2 | FEMALE | 名字2 |
| 详细数据 | user_3 | MALE | 名字3 |
| 详细数据 | user_4 | FEMALE | 名字4 |
| 详细数据 | user_5 | MALE | 名字5 |
| 详细数据 | user_6 | FEMALE | 名字6 |
| 详细数据 | user_7 | MALE | 名字7 |
| 详细数据 | user_8 | FEMALE | 名字8 |
| 详细数据 | user_9 | MALE | 名字9 |

共有10条, 每页显示：10条　« ‹ 1 › »

用户元数据以user_id为标识，与同步库中的timeline_id一一对应。用户同步新消息时，只会拉取同步库中自己对应的单个消息队列（TimelineQueue）。因此，为了唯一ID的方便管理，我们可以选择user_id与用户同步库的timeline_id使用同一个值。这样一来，在消息写扩散时，只需知道群内用户的user_id列表回好友user_id，即可以完成写扩算。

功能：用户检索

对于用户，添加好友的需求有很多种，本样例中我们只需要维护用户表，并且创建多元索引，即可轻松实现。样例中没有实现，您可以根据自己需求配置不同的索引字段设置，这里我们仅简单分析一下需求。

- 通过用户ID：主键查询
- 二维码（含用户ID信息）：主键查询
- 用户姓名：多元索引，用户名字段设置分词字符串
- 用户标签：多元索引，数组字符串索引提供签检索、嵌套索引提供多标签打分检索排序
- 附近的人：多元索引，GEO索引查询附近、特定地理围栏的人

更多关于多元索引，参见多元索引简介。

会话元数据

即会话的属性信息，通过唯一会话ID识别特定会话，属性信息会包含：会话类别（群、单聊、公众号等）、群名称、公告、创建时间等。同时，通过群名称模糊查找群，也会是会话元数需要的重要能力。

在Timeline模型中，提供了Timeline Meta的管理能力，只需通过相应的接口便可实现会话meta的管理。

数据源：im_timeline_meta_table　　表格数据最多显示50行。

| 详细数据 | timeline_id(主键) | create_time | group_name | type | users |
|---|---|---|---|---|---|
| 详细数据 | group_1 | 1563422737593 | 表格存储群1 | GROUP | |
| 详细数据 | group_2 | 1563422737782 | 表格存储群2 | GROUP | |
| 详细数据 | group_3 | 1563422737819 | 表格存储群3 | GROUP | |
| 详细数据 | group_4 | 1563422737852 | 表格存储群4 | GROUP | |
| 详细数据 | group_5 | 1563422737886 | 表格存储群5 | GROUP | |
| 详细数据 | single_2 | 1563422737945 | | SINGLE | [user_1, user_2] |
| 详细数据 | single_3 | 1563422738041 | | SINGLE | [user_1, user_3] |
| 详细数据 | single_4 | 1563422738072 | | SINGLE | [user_1, user_4] |
| 详细数据 | single_5 | 1563422738100 | | SINGLE | [user_1, user_5] |

共有9条, 每页显示：10条　« ‹ 1 › »

存储库中管理的是会话的消息队列（TimelineQueue），这里与会话元数据中的行一一对应。客户端用户选中特定会话后，应用从相应的消息队列倒序批量拉取消息展示到客户端，群聊单聊的使用方式一样，因而并不做会话类型的区分。

功能：群检索

用户如果有加入群的需求，首先需要查询到特定的群。查询群的方式与用户查询方式类似，功能也可以做相同的实现。用户可以根据自己需求定制不同的索引字段设置，需求实现方式如下。

- 群ID：主键查询
- 二维码（含用户ID信息）：主键查询
- 群名：多元索引，用户名字段设置分词字符串
- 群标签：多元索引，数组字符串索引提供签检索、嵌套索引提供多标签打分检索排序

说明：

会话元数据可以直接维护单聊会话与人的映射关系。对于单聊的meta增加一列users字段，存放两个用户ID，这样不用额外维护关系表（基于单聊关系表im_user_relation_table创建timeline_id为第一列主键的二级索引）。

关系维护

完成了元数据管理以及用户和群的检索，剩下的就是如何添加好友、加入群聊了。这里就涉及到IM体统中另一个重要的功能点。关系维护包含：人与人的关系、人与群的关系以及人与会话的。下面我们介绍如何基于Tablestore解决这一关系维护的需求。

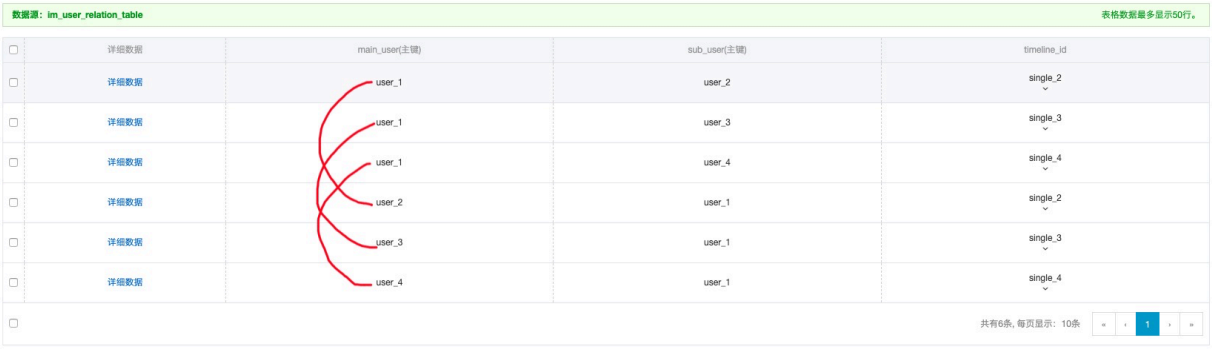
单聊关系

功能：人与单聊会话的关系

单聊场景下，参与者仅有两个人，同时不考虑顺序。无论是我联系小明或是小明联系我，对应的会话必须有且仅有一个。如果使用表格存储维护这个关系，建议用如下的设计方式。

第一列为主用户ID、第二列为次用户ID，在两个人成为好友后，关系表中需要插入两行数据，分别以自己的用户ID为main_user，以好友的用户ID为sub_user，然后将共同的会话timline_id作为属性列，并且可以维护相互之间不同的昵称、显示。

表设计：im_user_relation_table




基于该单聊关系表，还可以建立多元索引，方便用户好友列表的获取，同时支持加好友时间排序、昵称排序等功能。如果考虑到延时、费用等因素，即时使用多元索引，直接通过getRange接口也可以快速拉、高效的获取自己所有好友列表，实现好友关系的维护与查询。

功能：人与人的关系

借助以上表，人与人的关系可以很简单实现，比如我判断我与小明的好友关系，直接通过单行查询知道我们的好友关系是否存在，如果存在就不会展示加好友按钮。而如果非好友，这是完成好友添加后，写入两行不同主键顺序行，并生成一个唯一的timelineId即可。这个设计的好处在于用户可以直接通过自己的ID与好友的ID快速获取会话信息。只要用户在写入两行时做好一致性维护。

如果好友关系一旦解除，可以直接拼出关系表中两行主键对用户关系，通过做物理删除（删除行）或逻辑删除（属性列状态修改）结束两两个人的好友关系即可。

核心代码

```
public void establishFriendship(String userA, String userB, String
timelineId) {
    PrimaryKey primaryKeyA = PrimaryKeyBuilder.createPrimaryKeyBuilder
()
            .addPrimaryKeyColumn("main_user", PrimaryKeyValue.
fromString(userA))
            .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.
fromString(userB))
            .build();

    RowPutChange rowPutChangeA = new RowPutChange(userRelationTable,
primaryKeyA);
    rowPutChangeA.addColumn("timeline_id", ColumnValue.fromString(
timelineId));

    PrimaryKey primaryKeyB = PrimaryKeyBuilder.createPrimaryKeyBuilder
()
            .addPrimaryKeyColumn("main_user", PrimaryKeyValue.
fromString(userB))
```

```
            .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.
fromString(userA))
            .build();

    RowPutChange rowPutChangeB = new RowPutChange(userRelationTable,
primaryKeyB);
    rowPutChangeB.addColumn("timeline_id", ColumnValue.fromString(
timelineId));

    BatchWriteRowRequest request = new BatchWriteRowRequest();
    request.addRowChange(rowPutChangeA);
    request.addRowChange(rowPutChangeB);

    syncClient.batchWriteRow(request);
}

public void breakupFriendship(String userA, String userB) {
    PrimaryKey primaryKeyA = PrimaryKeyBuilder.createPrimaryKeyBuilder
()
            .addPrimaryKeyColumn("main_user", PrimaryKeyValue.
fromString(userA))
            .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.
fromString(userB))
            .build();

    RowDeleteChange rowPutChangeA = new RowDeleteChange(userRelati
onTable, primaryKeyA);

    PrimaryKey primaryKeyB = PrimaryKeyBuilder.createPrimaryKeyBuilder
()
            .addPrimaryKeyColumn("main_user", PrimaryKeyValue.
fromString(userB))
            .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.
fromString(userA))
            .build();

    RowDeleteChange rowPutChangeB = new RowDeleteChange(userRelati
onTable, primaryKeyB);

    BatchWriteRowRequest request = new BatchWriteRowRequest();
    request.addRowChange(rowPutChangeA);
    request.addRowChange(rowPutChangeB);

    syncClient.batchWriteRow(request);
}
```

```
[Create New SingleConversation]
    [Establish Friendship] user_1 make friend with: user_2 to user_5
        [Friends List] user_id: user_1, friends: [single_2:SINGLE:user_2:名字2, single_3:SINGLE:user_3:名字3, single_4:SINGLE:user_4:名字4, single_5:SINGLE:user_5:名字5]
        [Friends List] user_id: user_2, friends: [single_2:SINGLE:user_1:名字1]
    [Breakup Friendship] user_1 breakup with user_5
        [Friends List] user_id: user_1, friends: [single_2:SINGLE:user_2:名字2, single_3:SINGLE:user_3:名字3, single_4:SINGLE:user_4:名字4]
        [Friends List] user_id: user_5, friends: []
```

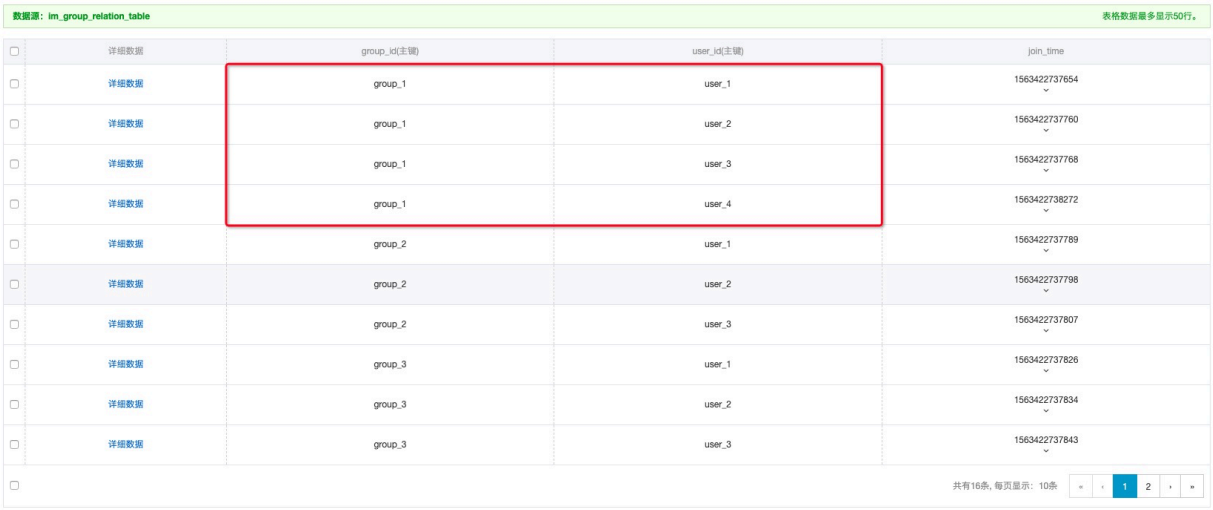
**群聊关系**

**功能：群聊会话与人的关系**

**群聊时，主要的查询需求还是获取当前群内用户的列表。一方面方便群属性的展示，另一方面为应用做写扩散提供快速获取收件人列表的查询。因而在表设计上，我们会建议用户使用两列主键：第一列为群ID，第二列为用户ID。通过这样的设计，可以直接给予getRange接口拉取群所有用户的信息。**

群聊关系表解决了群到用户的映射关系，但我们还需要用户到群的映射关系。如果为了查询用户所在群的列表而新键一张表，冗余成本、一致性维护成本就很高。这里可以使用两种索引来解决反向的映射关系。样例中，我们使用了二级索引，将用户ID字段作为索引主键，从而可以直接基于索引查询单用户的群列表。同步实时性更好，成本更低。当然用户也可以使用多元索引：对群、用户、入群时间做索引，可以查询到某用户的所有在群列表，并且基于入群时间排序。

表设计：im_group_relation_table



数据源：im_group_relation_table　　表格数据最多显示50行。

| 详细数据 | group_id(主键) | user_id(主键) | join_time |
|---|---|---|---|
| 详细数据 | group_1 | user_1 | 1563422737654 |
| 详细数据 | group_1 | user_2 | 1563422737760 |
| 详细数据 | group_1 | user_3 | 1563422737768 |
| 详细数据 | group_1 | user_4 | 1563422738272 |
| 详细数据 | group_2 | user_1 | 1563422737789 |
| 详细数据 | group_2 | user_2 | 1563422737798 |
| 详细数据 | group_2 | user_3 | 1563422737807 |
| 详细数据 | group_3 | user_1 | 1563422737826 |
| 详细数据 | group_3 | user_2 | 1563422737834 |
| 详细数据 | group_3 | user_3 | 1563422737843 |

共有16条, 每页显示：10条　« ‹ 1 2 › »

基于群关系表，可以直接基于关系主表通过getRange的方式获取单个群内所有的用户。在做写扩散时，可以直接获取群内用户ID列表，提升写扩散的效率。同时，也方便展示群内用户列表。

核心代码

```
public List<Conversation> listMySingleConversations(String userId) {
    PrimaryKey start = PrimaryKeyBuilder.createPrimaryKeyBuilder()
            .addPrimaryKeyColumn("main_user", PrimaryKeyValue.
fromString(userId))
            .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.INF_MIN)
            .build();

    PrimaryKey end = PrimaryKeyBuilder.createPrimaryKeyBuilder()
            .addPrimaryKeyColumn("main_user", PrimaryKeyValue.
fromString(userId))
            .addPrimaryKeyColumn("sub_user", PrimaryKeyValue.INF_MAX)
            .build();

    RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(
userRelationTable);
    criteria.setInclusiveStartPrimaryKey(start);
    criteria.setExclusiveEndPrimaryKey(end);
    criteria.setMaxVersions(1);
    criteria.setLimit(100);
    criteria.setDirection(Direction.FORWARD);
    criteria.addColumnsToGet(new String[] {"timeline_id"});

    GetRangeRequest request = new GetRangeRequest(criteria);
    GetRangeResponse response = syncClient.getRange(request);

    List<Conversation> singleConversations = new ArrayList<Conversati
on>(response.getRows().size());
```

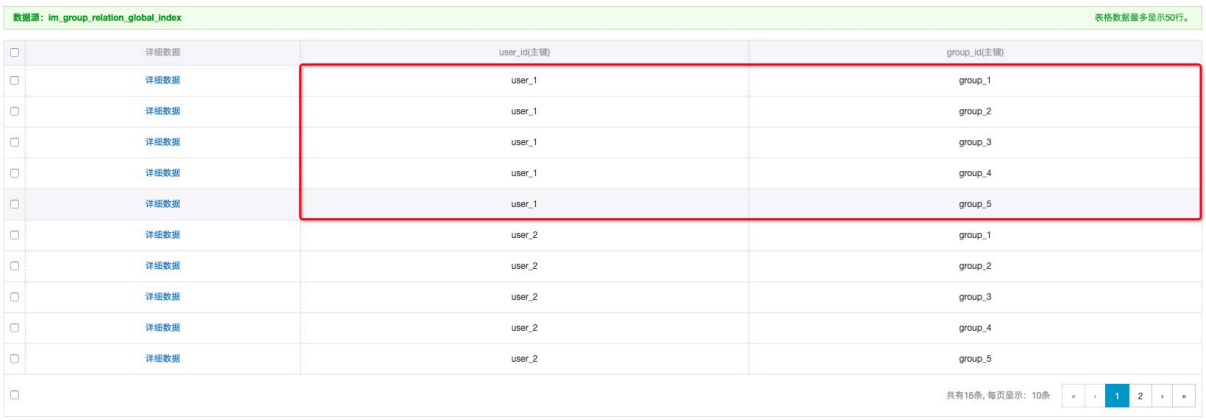
```
    for (Row row : response.getRows()) {
        String timelineId = row.getColumn("timeline_id").get(0).
getValue().asString();
        String subUserId = row.getPrimaryKey().getPrimaryKeyColumn("
sub_user").getValue().asString();
        User friend = describeUser(subUserId);

        Conversation conversation = new Conversation(timelineId,
friend);

        singleConversations.add(conversation);
    }

    return singleConversations;
}
```

```
[List User Single Conversation]:  -> user_1
    [timelineId: single_2]: type: SINGLE, userId: user_2, userName: 名字2, sexuality: FEMALE
    [timelineId: single_3]: type: SINGLE, userId: user_3, userName: 名字3, sexuality: MALE
    [timelineId: single_4]: type: SINGLE, userId: user_4, userName: 名字4, sexuality: FEMALE
    [Total Count]: 3
```

**功能：人与群聊会话的关系**

**获取单用户所有加入群列表，可以基于主表创建二级索引，将用户字段设为索引的第一列主键。索引的数据结构见下图。这样基于二级索引，可以直接通过getRange的方式获取单用户加入的群的TimlineId列表。**

**二级索引：im_group_relation_global_index**



| 详细数据 | user_id(主键) | group_id(主键) |
|---|---|---|
| 详细数据 | user_1 | group_1 |
| 详细数据 | user_1 | group_2 |
| 详细数据 | user_1 | group_3 |
| 详细数据 | user_1 | group_4 |
| 详细数据 | user_1 | group_5 |
| 详细数据 | user_2 | group_1 |
| 详细数据 | user_2 | group_2 |
| 详细数据 | user_2 | group_3 |
| 详细数据 | user_2 | group_4 |
| 详细数据 | user_2 | group_5 |

**核心代码**

```
public List<Conversation> listMyGroupConversations(String userId) {
    PrimaryKey start = PrimaryKeyBuilder.createPrimaryKeyBuilder()
            .addPrimaryKeyColumn("user_id", PrimaryKeyValue.fromString
(userId))
            .addPrimaryKeyColumn("group_id", PrimaryKeyValue.INF_MIN)
            .build();

    PrimaryKey end = PrimaryKeyBuilder.createPrimaryKeyBuilder()
            .addPrimaryKeyColumn("user_id", PrimaryKeyValue.fromString
(userId))
            .addPrimaryKeyColumn("group_id", PrimaryKeyValue.INF_MAX)
```

```
            .build();

    RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(
groupRelationGlobalIndex);
    criteria.setInclusiveStartPrimaryKey(start);
    criteria.setExclusiveEndPrimaryKey(end);
    criteria.setMaxVersions(1);
    criteria.setLimit(100);
    criteria.setDirection(Direction.FORWARD);
    criteria.addColumnsToGet(new String[] {"group_id"});

    GetRangeRequest request = new GetRangeRequest(criteria);
    GetRangeResponse response = syncClient.getRange(request);

    List<Conversation> groupConversations = new ArrayList<Conversation
>(response.getRows().size());

    for (Row row : response.getRows()) {
        String timelineId = row.getPrimaryKey().getPrimaryKeyColumn("
group_id").getValue().asString();
        Group group = describeGroup(timelineId);

        Conversation conversation = new Conversation(timelineId, group
);

        groupConversations.add(conversation);
    }

    return groupConversations;
}
```

```
[List User Group Conversation]:  -> user_1
    [timelineId: group_1]: type: GROUP, groupName: 表格存储群1, createAt: 2019-07-17 21:26:41
    [timelineId: group_2]: type: GROUP, groupName: 表格存储群2, createAt: 2019-07-17 21:26:41
    [timelineId: group_3]: type: GROUP, groupName: 表格存储群3, createAt: 2019-07-17 21:26:41
    [timelineId: group_4]: type: GROUP, groupName: 表格存储群4, createAt: 2019-07-17 21:26:41
    [timelineId: group_5]: type: GROUP, groupName: 表格存储群5, createAt: 2019-07-17 21:26:41
    [Total Count]: 5
```

**即时感知**

**让客户端即时感知消息的实现方案，可以参考Feed 流设计总纲中会话池的维护方式，这里作简要描述，不会在样例中实现。**

### 会话池方案

**即时感知新消息正是IM（Instant Message）场景下核心所在。让客户端及时感知到新信息的到来，然后客户端接收到通知后才会从同步库中拉取更新的消息，让用户更快速、更及时地提醒用户阅读新消息。可是，接受者如何才能快速感知到自己有了新消息呢?**

**让在线的客户端周期性的刷新拉取？这样的方式毫无疑问可以满足需求，但伴随而来的是大量无效的网络资源浪费。同时应用的压力也会随着用户量的不断增长变得更沉重。而当白天大量非活跃用户在线时，压力更为明显。面对这一问题，应用通常会维护一个推送会话池。会话池记录了在线客户端与用户信息，当在线用户有新的消息写入，通过推送池获取该用户的会话，然后通知客户端拉**

取同步库新消息。这样同步消息的压力只会随着真实消息量而增长，避免了大量不必要的同步库查询请求。

实现会话推送池的方案很多，可以使用内存型数据库，也可以直接使用表格存储，同时保证会话推送池的持久化。

在即时感知上，最直观的就是会话表中变动的未读消息数统计了。统计新消息的实现方式上，已在本文的【消息存储 > 第二类：同步库 > 新消息即时统计】部分做了详尽描述，不理解的可返回去重新看一下。持久化未读消息数是很必要的，否则在更换设备或重新登录后。未读消息数被清零，将会忽略很多新消息提醒，这是我们不能接受的。

### 多端同步

实现了以上功能，IM系统的基本需求已经完成。但实现多端数据同步上，还有两个注意事项。

其一，我们对于单客户端情况下，用户同步库做了一个checkpoint点的持久化，对应的概念是：“已读最新消息的sequenceId”。此时，checkpoint点无客户端的区分，如果使用本地做持久化，多端同步时就会出现问题，不同客户端统计的未读消息数就会不一致。这是需要通过应用服务端维护checkpoint点，同时会话的未读消息数也需要在应用服务侧维护，这样才能保证多端统计数一致。同时，当有未读消息的会话被点击，会话未读数清0时，要让服务有感知，然后通知到其他在线端，维护实时一致性。

其二，多端情况下，自己在一个客户端发送了新消息，其他客户端在没有其他新消息时，是无法感知并刷新自己的发送消息，这在多端同步中也是要解决的小问题。这时，简单的解决方案就是将自己发送的消息，也写入自己的同步库。只要再统计未读信息时，对自己的信息不计数，但在最新消息摘要中需要做更新。这样，多端同步问题很容易实现。

### 添加好友、入群申请

添加好友或入群，不是主动发起请求就会直接完成的，这里需要主动方申请后，审核方完成统一才会真实完成。因而只有在审核方才会有权限发起关系的创建。

那如何让被添加用户或群主感知到申请？当然是借助同步库，作为一种新的消息类型或者特殊的会话，让用户即时感知到新申请，尽早完成审批。申请列表如果需要持久化，也可单独建表维护，只要保证用户新申请的即时感知即可。

### 样例实操

本位为了与用户一起梳理IM系统应用的功能点，基于Tablestore实现的样例简单功能，完整的样例代码已完成开源，代码地址：源码链接。用户可以结合文章、代码一起阅读。代码在本地运行，使用前请确保您已经完成了以下操作：

- #unique_12、#unique_14

- 获取AK
- 设置样例配置文件
- 实例支持二级索引（需要通过工单主动申请）

样例配置

在home目录下创建tablestoreCong.json文件，填写相应参数如下：

```
# mac 或 linux系统下: /home/userhome/tablestoreCong.json
# windows系统下: C:\Documents and Settings\%用户名%\tablestoreCong.json
{
  "endpoint": "http://instanceName.cn-hangzhou.ots.aliyuncs.com",
  "accessId": "***********",
  "accessKey": "**********************",
  "instanceName": "instanceName"
}
```

endpoint：实例的接入地址，控制台实例详情页获取

accessId：AK的ID，获取AK链接提供

accessKey：AK的密码，获取AK链接提供

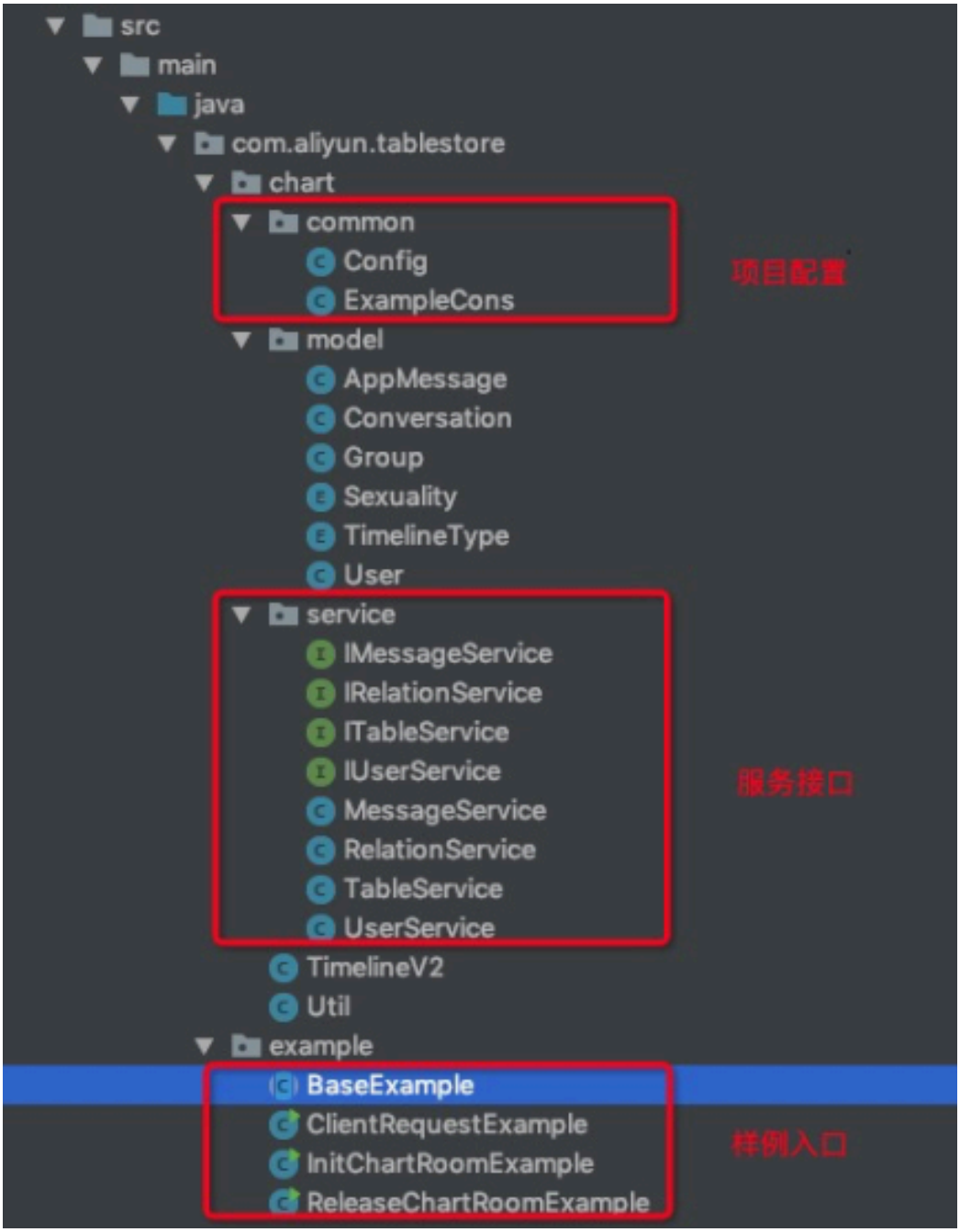
instanceName：使用的实例名

样例入口

样例中共有三个入口，用户需要根据先后顺序执行，使用后及时释放资源，避免不必要的费用浪费。

| 入口 | 入口类名 | 功能 |
| --- | --- | --- |
| 初始化 | InitChartRoomExample | 创建所有需要的表，同时根据配置创建相应的多元索引与二级索引 |
| 模拟调用 | ClientRequestExample | 应用的接口使用，样例未做前后端联调调用，用户可通过接口返回数据的打印了解使用方式。 |
| 释放资源 | ReleaseChartRoomExample | 释放所有资源，先释放索引后删表 |

### 项目结构




### 专家服务

**表格存储提供专业的免费的技术咨询服务，欢迎加入我们的钉钉讨论群。**


表格存储公开交流群
1023人
扫一扫群二维码，立刻加入该群。

# 7 海量气象格点数据解决方案

## 7.1 方案背景

本文主要为您介绍基于表格存储的海量气象格点数据解决方案的背景及挑战。

背景

气象数据是一类典型的大数据，具有数据量大、时效性高、数据种类丰富等特点。气象数据中大量的数据是时空数据，记录了时间和空间范围内各个点的各个物理量的观测量或者模拟量，每天产生的数据量常在几十TB到上百TB的规模，且在爆发性增长。如何存储和高效的查询这些气象数据越来越成为一个难题。

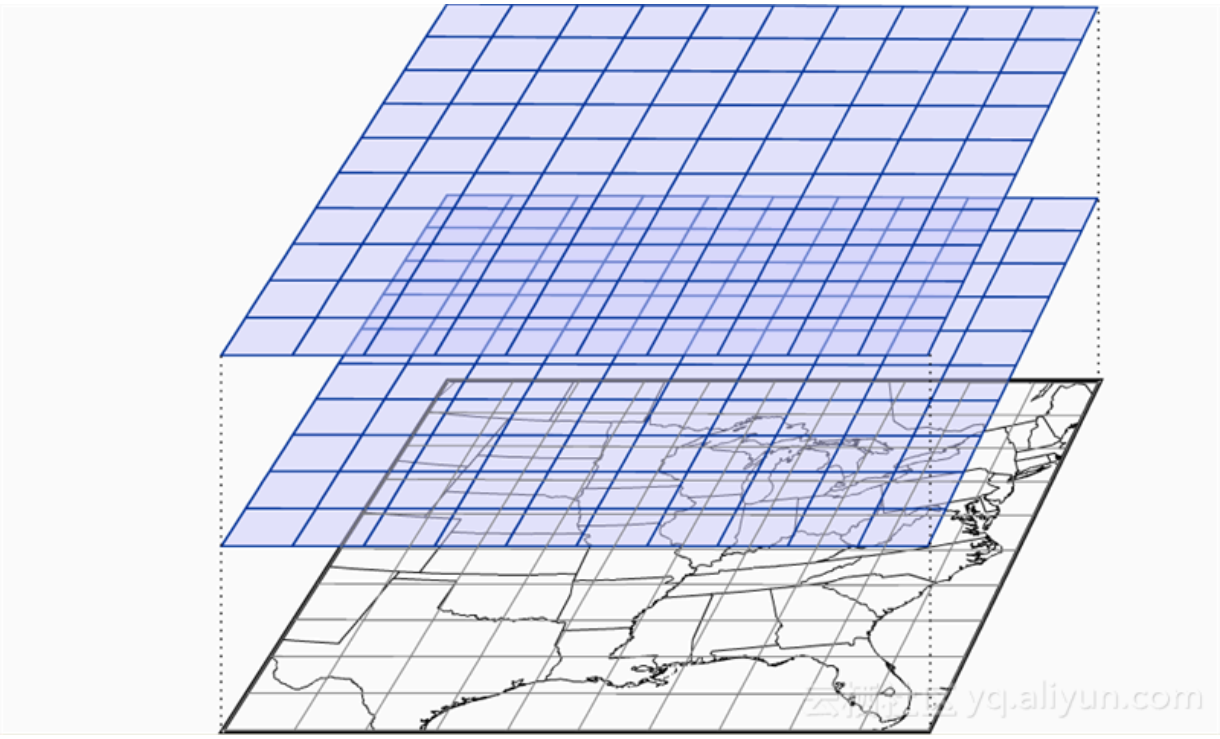
传统的方案采用关系型数据库加文件系统的方式实现这类气象数据的存储和实时查询。 传统方案在可扩展性、可维护性和性能上都有缺陷，并且随着数据规模的增大，缺点越来越明显。表格存储是一款阿里云自研的分布式NoSQL服务，可以提供超大规模的存储容量，支撑超大规模的并发访问和低延迟的性能，可以很好的解决气象数据的规模和查询性能问题。本方案主要通过表格存储来实现气象格点数据的存储和查询。

挑战一：数据规模大

格点数据具有明显的多维特点，以模式系统每次产生的数据为例，一般包含以下五个维度。

- 物理量（或者称为要素，例如温度、湿度、风向、风速等）
- 时间（例如气象中的预报时效，未来3小时、6小时、9小时等）
- 高度
- 经度
- 纬度

当我们固定某一要素某一预报时效，那么高度、经度、纬度就构成一个三维网格数据，如下图所示。每个格点代表了一个三维空间上的点，上面的数值为该点在某一预报时效（例如未来三小时）下，某一物理量（例如温度）的预报值。



假设一个三维格点空间包含10个不同高度的平面，每个平面为一个2880 x 570的格点，每个格点保存一个4字节数据，那么这三维的数据量为2880 x 570 x 4 x 10，大约64MB。这仅仅是某个模式系统对某个物理量某一时效下的一次预报，可见模式数据的总量非常巨大，这对数据库的存储提出了一个不小的挑战。

挑战二：查询维度多

预报员会通过页面的形式浏览各种模式数据（格点数据），并进行数值模式预报。这个页面需要提供多种模式数据的查询方式，例如：

- 查询一个经纬度平面的格点数据

  例如未来三小时全球地面温度的格点数据，或者未来三小时浙江省地面温度数据。
- 查询某个格点的时间序列数据

  例如阿里云公司所在地未来3小时、未来6小时、一直到未来72小时的温度。
- 查询不同物理量的数据

  例如查询某一预报时效、某一高度、某一点的全部物理量的预报数据。
- 查询不同模式系统产生的数据

  例如同时查询欧洲中心的某一模式数据和中国气象机构产生的对应数据等。

一个格点数据集一般是一个五维结构，各种查询方式实际上就是对这个五维数据进行切分，例如查询某个平面、每个剖面，某个点序列、某个三维、四维子空间等等。方案设计要保证在各种查询条件的查询性能，这是数据查询方面的主要技术挑战。

## 7.2 方案设计

本章节主要为您介绍基于表格存储的海量气象格点数据解决方案的模型及方案设计。

### 标准化格点数据模型

一个规整的五维网格数据为一个网格的数据集（GridDataSet），按照维度顺序五维分别为：

| 维度 | 说明 |
| --- | --- |
| variable | 变量，例如各种物理量 |
| time | 时间维度 |
| z | z轴，一般表示空间高度 |
| x | x轴，一般表示经度或纬度 |
| y | y轴，一般表示经度或纬度 |

GridDataSet = F(variable, time, z, x, y)

一个GridDataSet除了包含五维数据，以及各个维度的长度等外，还包含一些其他信息：

| 名称 | 说明 |
| --- | --- |
| GridDataSetId | 唯一标记这个GridDataSet的ID。 |
| Attributes | 自定义属性信息，例如该数据的产生时间、数据来源、预报类型等等。<br>您可以自定义属性，也可以给某些属性建立索引，建立索引后就可以通过各种组合条件来查询符合条件的数据集。 |

例如，假设某种气象预报每次预报未来72小时的每个整点的各个高度、各个经纬度的各种物理量，则这次预报就是一个标准的五维数据，是一个单独的GridDataSet，下一次相同的预报则是另一个数据集。这两个数据集需要有不同的GridDataSetId。这两个数据集比较类似，只是起报时间不同，但是因为起报时间不在五维模型中（五维内的时间为一次预报中的未来不同时刻），所以属于不同的数据集，起报时间可以作为数据集的自定义属性。本方案中，也支持对自定义属性设置条件进行检索。

### 数据存储方案

表格存储设计了两张表分别存储数据集的meta和data：

- meta表示这个数据集的元数据，例如GridDataSetId、各维度长度、自定义属性等。
- data表示这个数据集里实际的网格数据。data相比meta在数据大小上要大很多。

将数据集的meta和data分开存储，主要是出于以下考虑：

- 用户会有根据多种条件查询数据集的要求，例如查询最近有哪些数据集已经完成入库，或者查询表中有哪些某种类型的数据集等。传统方案中主要是通过MySQL等关系型数据库来存储，在本方案中我们通过单独的meta表来存储，并通过表格存储的多元索引功能来实现多条件的组合查询和多种排序方式，相比传统方案更加易用。
- 在查询格点数据之前，一般要知道格点数据中各维度的长度等信息，这些信息就是存储在meta表中的，即需要先查询meta表，再查询data表。因为meta数据一般都很小，因此查询效率相比查询data要高，多一次查询并不会明显增加延迟。

### meta表设计

由于GridDataSetId可以唯一标记一个GridDataSet，所以meta表的主键只有一列，用于记录GridDataSetId。各种系统属性和自定义属性保存在meta表的属性列中。


查询meta表有两种方式：一种是通过GridDataSetId直接查询，另外一种是通过多元索引。可以根据多种属性条件组合进行查询，例如筛选某种类型的数据，按照入库时间从新到老返回等。

### data表设计

data表的设计要解决五维数据在不同的切分模式下的查询效率问题，不能简单直接的对数据进行存储。

为了高效查询，需要尽量减少一次查询需要扫描的数据量。一个数据集的数据量可能在几GB的级别，但是一次查询往往只需要其中的几MB的数据，如果无法高效的定位要查询的数据，那么就要扫描全部的几GB的数据，从中筛选出符合某个范围的数据，显然效率是很低的。如何才能高效定位到需要的数据之中是提高查询效率的关键。

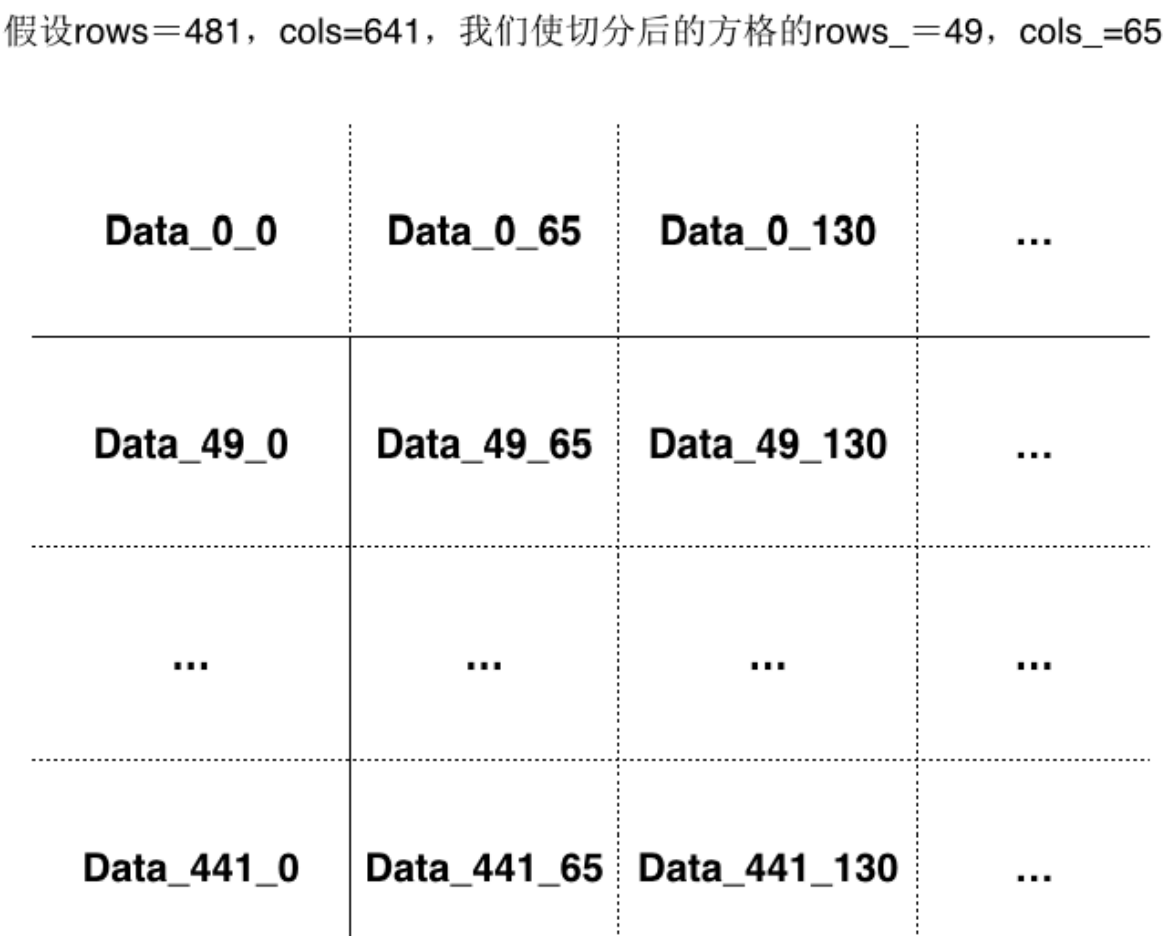
合理的表结构设计可以提高查询效率，在设计表data时，使用四个主键列：

| 主键 | 说明 |
| --- | --- |
| GridDataSetId | 数据集Id，唯一标记这个数据集。 |
| Variable | 变量名，即五维模型中的第一维。 |
| Time | 时间，即五维模型中的第二维。 |
| Z | 高度，即五维模型中的第三维。 |


这四列主键列标记一行表格存储中的数据，这行数据需要保存后两维的数据，即一个格点平面。

这种设计下，五维中的前三维都可以通过主键列的值来定位，即对于前三维的每一种情况，都对应表格存储中的一行。由于前三维分别代表变量、时间和高度，一般不会很多，每个维度在几个到几十个的级别，可以通过一些并行查询的方法来加速查询速度。

后两维代表了一个水平的平面，一般是一个经纬度网格，这两维的大小是比前三维要大很多的，每维在几百到几千的级别，随着数值预报越来越精细化，这个网格的大小还会成倍增加。这样的一个稠密的网格数据，不能把每个格点都用一列来保存，这样列的数量会非常多，存储效率也会非常的低。另一方面，如果我们把一个平面的格点数据存储到一列中，在整读整取时效率比较高，但是如果只读取某个点，就会读取很多的无效数据，效率又会变得比较低。因此我们采取一种折中的方案，对平面的二维数据再次进行切分，切分成更小的平面数据块，这样就可以做到只读取部分数据块，而不总是读取整个平面，因此极大的提高了查询性能。




## 7.3 方案实现

**本章节主要为您介绍如何基于表格存储实现海量气象格点数据解决方案。**

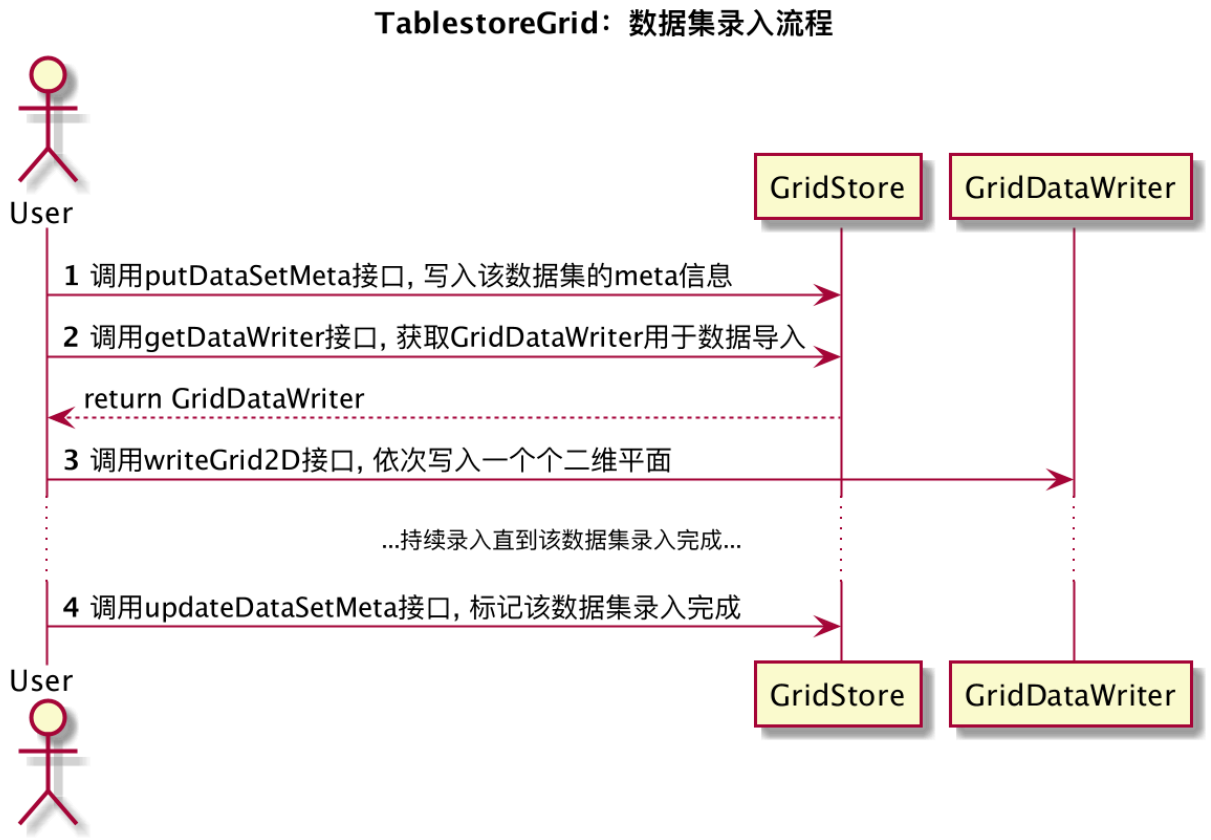
**基于方案设计，表格存储实现了一个Tablestore-Grid的library，基于这个Library您可以非常方便的实现气象格点数据的存储、查询和管理。提供以下接口：**

```
public interface GridStore {
    /**
     * 创建相关的meta、data表，数据录入前调用。
     * @throws Exception
     */
    void createStore() throws Exception;

    /**
     * 写入gridDataSet的meta信息。
     * @param meta
     * @throws Exception
     */
    void putDataSetMeta(GridDataSetMeta meta) throws Exception;

    /**
     * 更新meta信息。
     * @param meta
     * @throws Exception
     */
    void updateDataSetMeta(GridDataSetMeta meta) throws Exception;
```

```
    /**
     * 通过gridDataSetId获取meta。
     * @param dataSetId
     * @return
     * @throws Exception
     */
    GridDataSetMeta getDataSetMeta(String dataSetId) throws Exception;

    /**
     * // 创建meta表的多元索引。
     * @param indexName
     * @param indexSchema
     * @throws Exception
     */
    void createMetaIndex(String indexName, IndexSchema indexSchema)
throws Exception;

    /**
     * 通过多种查询条件来查询符合条件的数据集。
     * @param indexName 多元索引名。
     * @param query 查询条件，可以通过QueryBuilder构建。
     * @param queryParams 查询相关参数，包括offset、limit、sort等。
     * @return
     * @throws Exception
     */
    QueryGridDataSetResult queryDataSets(String indexName, Query query
, QueryParams queryParams) throws Exception;

    /**
     * 获取GridDataWriter用于写入数据。
     * @param meta
     * @return
     */
    GridDataWriter getDataWriter(GridDataSetMeta meta);

    /**
     * 获取GridDataFetcher用于读取数据。
     * @param meta
     * @return
     */
    GridDataFetcher getDataFetcher(GridDataSetMeta meta);

    /**
     * 释放资源。
     */
    void close();
}

public interface GridDataWriter {
    /**
     * 写入一个二维平面。
     * @param variable 变量名。
     * @param t 时间维的值。
     * @param z 高度维的值。
     * @param grid2D 平面数据。
     * @throws Exception
     */
    void writeGrid2D(String variable, int t, int z, Grid2D grid2D)
throws Exception;
}

public interface GridDataFetcher {
    /**
```

```
     * 设置要查询的变量。
     * @param variables
     * @return
     */
    GridDataFetcher setVariablesToGet(Collection<String> variables);

    /**
     * 设置要读取的各维度起始点和大小。
     * @param origin 各维度起始点。
     * @param shape 各维度大小。
     * @return
     */
    GridDataFetcher setOriginShape(int[] origin, int[] shape);

    /**
     * 获取数据。
     * @return
     * @throws Exception
     */
    GridDataSet fetch() throws Exception;
}
```

**数据录入**




**数据录入流程可以分为三部分：**

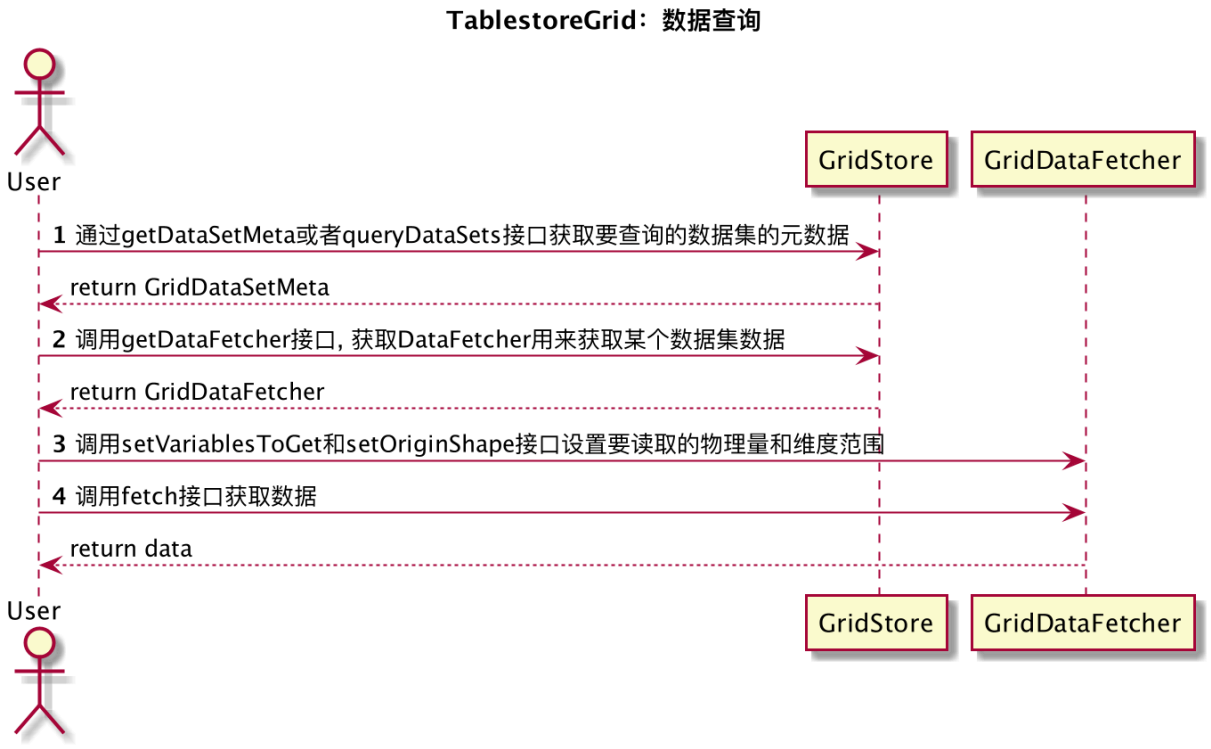
- **写入putDataSetMeta接口写入数据集的meta信息。**
- **通过GridDataWriter录入整个数据集的数据。**
- **通过updateDataSetMeta接口更新数据集的meta信息，标记数据已经录入完成。**

**示例如下：**

**以下示例中读取一个NetCDF（气象格点数据常用的格式）文件，然后将其中的数据通过GridDataWriter录入到表格存储中。通过GridDataWriter每次写入时，只能写入一个二维平面，所以需要在外层进行3层循环，分别枚举变量维、时间维、高度维的值，然后读取对应的二维平面的数据进行录入。**

```
public void importFromNcFile(GridDataSetMeta meta, String ncFileName)
throws Exception {
    GridDataWriter writer = tableStoreGrid.getDataWriter(meta);
    NetcdfFile ncFile = NetcdfFile.open(ncFileName);
    List<Variable> variables = ncFile.getVariables();
    for (Variable variable : variables) {
        if (meta.getVariables().contains(variable.getShortName())) {
            for (int t = 0; t < meta.gettSize(); t++) {
                for (int z = 0; z < meta.getzSize(); z++) {
                    Array array = variable.read(new int[]{t, z, 0, 0
}, new int[]{1, 1, meta.getxSize(), meta.getySize()});
                    Grid2D grid2D = new Grid2D(array.getDataAsB
yteBuffer(), variable.getDataType(),
                                new int[] {0, 0}, new int[] {meta.getxSize
(), meta.getySize()});
                    writer.writeGrid2D(variable.getShortName(), t, z,
grid2D);
                }
            }
        }
    }
}
```

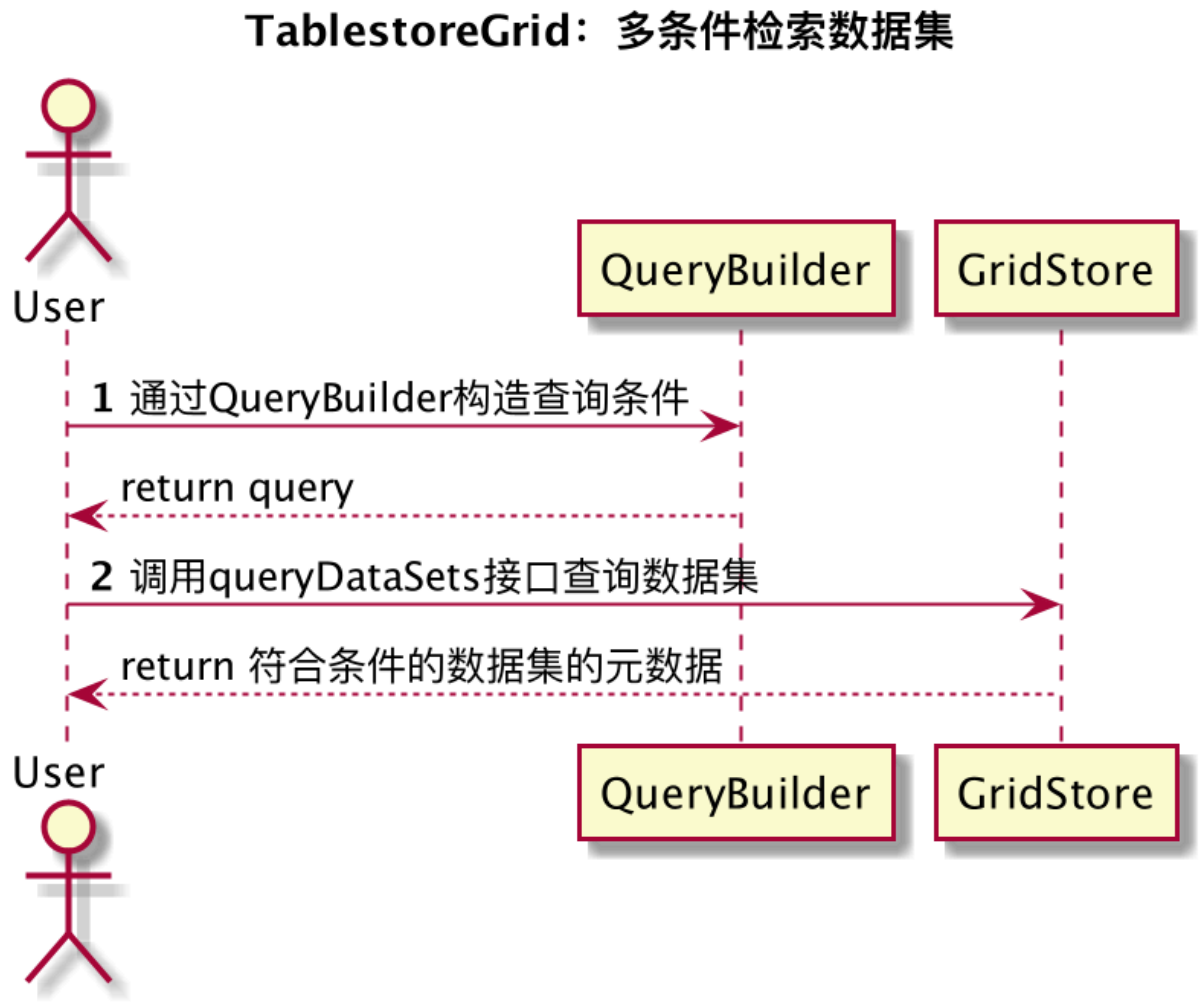
**数据查询**



GridDataFetcher支持对五维数据进行任意维度的查询。第一维是变量维，通过setVariablesToGet接口设置要读取哪些变量，其余四维通过设置起始点（origin）和读取的大小（shape）就可以实现任意维度读取。

```
public Array queryByTableStore(String dataSetId, String variable, int
[] origin, int[] shape) throws Exception {
      GridDataFetcher fetcher = this.tableStoreGrid.getDataFetcher(
this.tableStoreGrid.getDataSetMeta(dataSetId));
      fetcher.setVariablesToGet(Arrays.asList(variable));
      fetcher.setOriginShape(origin, shape);
      Grid4D grid4D = fetcher.fetch().getVariable(variable);
      return grid4D.toArray();
}
```

多条件检索数据集




本方案中，对meta表建立多元索引后，可以支持通过各种组合条件来进行数据集检索，查询出符合条件的数据集，这个功能对于气象管理系统来说非常重要。

示例如下：

假设我们要查询已经完成入库的，创建时间为最近一天的，来源为ECMWF（欧洲中期天气预报中心）或者NMC（全国气象中心），精度为1km的气象预报，并按照创建时间从新到老排序，可以用以下代码实现：

查询条件：(status == DONE) and (create_time > System.currentTimeMillis - 86400000) and (source == "ECMWF" or source == "NMC") and (accuracy == "1km")

```
QueryGridDataSetResult result = tableStoreGrid.queryDataSets(
        ExampleConfig.GRID_META_INDEX_NAME,
        QueryBuilder.and()
                .equal("status", "DONE")
                .greaterThan("create_time", System.currentTimeMillis()
 - 86400000)
                .equal("accuracy", "1km")
                .query(QueryBuilder.or()
                        .equal("source", "ECMWF")
                        .equal("source", "NMC")
                        .build())
                .build(),
        new QueryParams(0, 10, new Sort(Arrays.<Sort.Sorter>asList(new
 FieldSort("create_time", SortOrder.DESC)))));
```

这一部分功能利用了表格存储的多元索引，多元索引可以实现多字段组合查询、模糊查询、全文检索、排序、范围查询、嵌套查询、空间查询等功能，给元数据管理场景提供了强大的底层能力。

### 代码的获取

可以在github上获取Tablestore-Grid的实现代码和示例代码。

### 专家服务

表格存储提供专业的免费的技术咨询服务，欢迎加入我们的钉钉讨论群。