开放结构化数据服务（OTS）

开发者手册 &API 参考

Open Table Service Developer Guide & API Reference

API Version:2014-08-08

目录

第 1 章　OTS 简介

欢迎使用阿里云开放结构化数据服务 OTS(Open Table Service)。本文档旨在帮助广大开发者更快速、更高效的使用 OTS 开发应用程序。OTS 是构建在阿里云飞天分布式系统之上的 NoSQL 数据库服务，提供海量结构化数据的存储和实时访问。OTS 以实例和表的形式组织数据，通过数据分片和负载均衡技术，实现规模上的无缝扩展。应用通过调用 OTS API/SDK 或者操作管理控制台来使用 OTS 服务。

如果您是第一次了解 OTS，推荐您从以下章节进行阅读:

- OTS 简介

如果您是应用开发者，以下章节提供了 OTS 的详细信息，让您的应用更好的和 OTS 一起工作:

- OTS 表
- OTS 数据操作
- 访问控制
- 计量计费
- 限制项

如果您是 OTS SDK 的开发者，可以阅读以下进阶章节:

- 使用 OTS API
- API Reference

OTS 特点

OTS 是构建在阿里云飞天分布式系统之上的 NoSQL 数据库服务，提供海量结构化数据的存储和实时访问。OTS 以实例和表的形式组织数据，通过数据分片和负载均衡技术，达到规模的无缝扩展。OTS 向应用程序屏蔽底层硬件平台的故障和错误，能自动从各类错误中快速恢复，提供非常高的服务可用性。OTS 管理的数据全部存储在 SSD 中并具有多个备份，提供了快速的访问性能和极高的数据可靠性。用户在使用 OTS 服务时，只需要按照预留和使用的资源进行付费，无需关心数据库的软硬件升级维护、集群缩容扩容等复杂问题。

OTS 的主要特点:

- 扩展性

- 动态调整预留读写吞吐量

  应用在创建表的时候，需要根据业务访问的情况来配置预留读写吞吐量。OTS 根据表的预留读写吞吐量进行资源的调度和预留，从而保证应用获得可预期的性能。在使用过程中，应用还可以根据应用的情况动态修改预留读写吞吐量

- 无限容量

  OTS 表的数据量没有上限，随着表数据量的不断增大，OTS 会进行数据分区的调整从而为该表配置更多的存储

- 数据可靠性

  OTS 通过存储多个数据备份及备份失效时的快速恢复，提供极高的数据可靠性

- 高可用性

  通过自动的故障检测和数据迁移，OTS 对应用屏蔽了机器和网络的硬件故障，提供高可用性

- 管理便捷

  应用程序无需关心数据分区的管理，软硬件升级，配置更新，集群扩容等繁琐运维任务

- 访问安全性

  OTS 对应用的每一次请求都进行身份认证和鉴权，以防止未授权的数据访问，确保数据访问的安全性

- 强一致性

  OTS 保证数据写入强一致，写操作一旦返回成功，应用就能立即读到最新的数据

- 灵活的数据模型

  OTS 的表无固定格式要求，每行的列数可以不相同，支持多种数据类型 (Integer、Boolean、Double、String、Binary)

- 按量付费

  OTS 根据用户预留和实际使用的资源进行收费，起步门槛低

- 监控集成

  用户可以从 OTS 控制台实时获取每秒请求数、平均响应延时等监控信息

OTS 数据模型概念

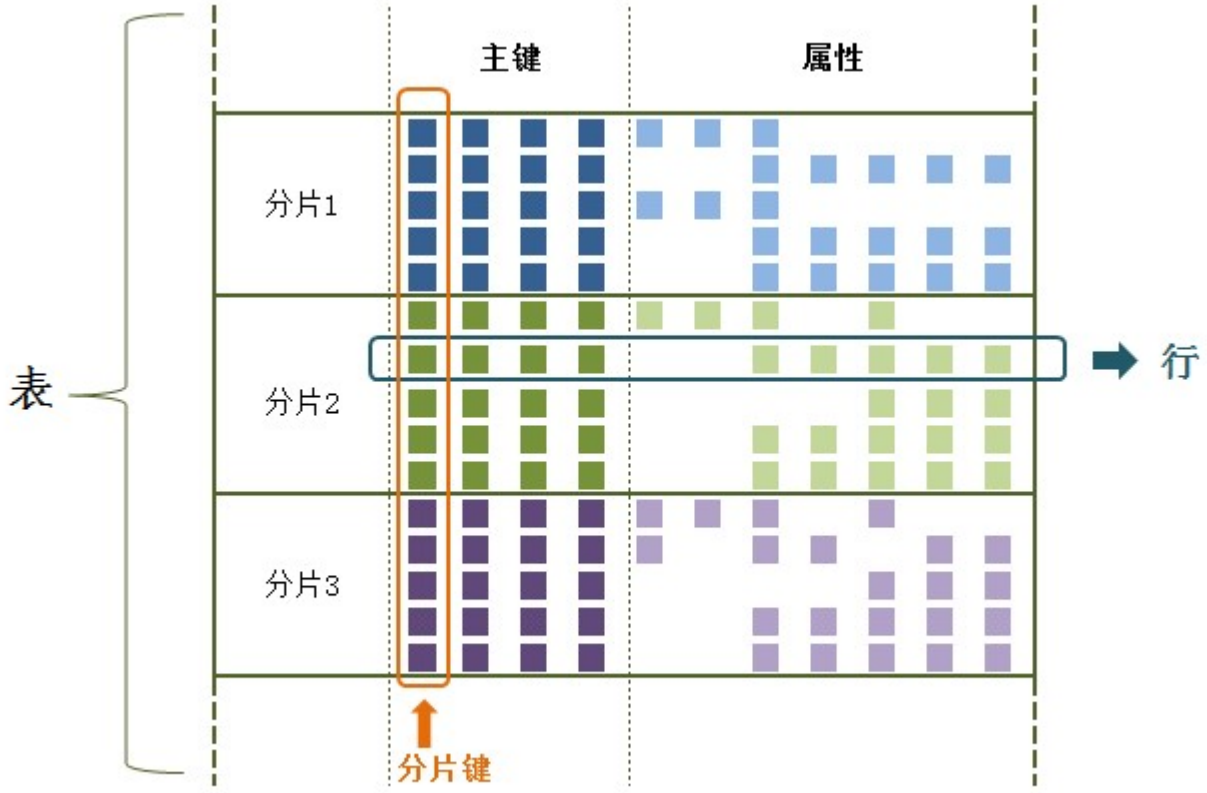

Figure 1: 表结构示意图

OTS 数据模型概念包括表、行、主键和属性。

表是行的集合，行由主键和属性组成。主键列和属性列均由名称和值组成。表中的所有行都必须包含相同数目和名称的主键列，但每行包含的属性列的数目可以不固定，名字和类型也可以不同。

下面的例子展示了同一张表中两行:

id	Type	ISBN	PageCount	Length
'4776'	'Book'	'123-45678912345'	666	空
'6555'	'Music'	空	空	'4:00'

id 是表的主键，id 为'4776' 和'6555' 的行拥有不同的属性，它们可以存在一张表中。

主键

主键是表中每一行的唯一标识。主键由 1 到 4 个主键列组成。应用在创建表的时候，必须明确指定主键的组成，每一个主键列的名字和数据类型以及它们的顺序。主键列的数据类型只能

是 String 和 Integer。如果为 String 类型，长度不超过 1KB。

属性

属性存放行的数据。每一行可以包含 0~128 个属性列。

列名的命名规范

主键列和属性列遵循相同的命名规范:

- 必须由英文字母、数字和下划线组成
- 首字符必须为英文字母、下划线
- 大小写敏感
- 长度在 1~255 个字符之间

合法的列名举例:

- _id
- Message

非法的列名举例:

- sn 序列号 _21

  不能包含中文

- 5store

  首字母不能是数字

- shoping(new)

  不能包含除下划线以外的其他符号

列值类型

OTS 一共支持 5 种类型的列值:

类型	值	能否作为主键	空间占用限制
String	UTF-8 字符串，可以为空字符串	是	不超过 64KB，如果为主键列，不超过 1KB
Integer	范围为 -2^63 ~ 2^63-1 的 64 位整型	是	固定 8Byte
Double	范围为 -10^308 ~ 10^308 的双精度浮点数，提供 15 ~ 16 位有效数字	否	固定 8Byte
Boolean	值为 True 或 False	否	固定 1Byte
Binary	二进制数据，可以为空	否	不超过 64KB

行数据大小的计算

OTS 每行数据都占用一定存储空间，存储空间的计算方式如下:

```
单行数据大小 = 主键的数据大小 + 属性的数据大小
主键的数据大小 = 主键列的名字长度之和 + 主键列的值的数据大小之和
属性的数据大小 = 属性列的名字长度之和 + 属性列的值的数据大小之和
```

值的数据大小的计算方式如下:

- String - UTF-8 字符串占用的字节数。如果字符串为空 (OTS 允许值为空的 String 类型)，则数据大小为 0。
- Integer - 固定 8 字节
- Double - 固定 8 字节
- Boolean - 固定 1 字节
- Binary - 二进制数据占用的字节数

OTS 限制单行数据大小不超过 256KB。

id	Type	ISBN	PageCount	Length
'4776'	'Book'	'123-45678912345'	666	空
'6555'	'Music'	空	空	'4:00'
'8000'	'Music'	空	空	'2:33'

举例 id='4776' 的行数据大小计算，其行数据大小为所有列的名字长度之和与值大小之和。len('id')+len('Type')+len('ISBN')+len('PageCount')+4Byte+4Byte+15Byte+8Byte=50Byte

分片键

组成主键的第一个主键列又称为分片键。OTS 会根据表中每一行分片键的值所属的范围自动将这一行数据分配到对应的分片和机器上，以达到负载均衡的目的。

具有相同分片键的行属于同一个分片，一个分片可能包含多个分片键。OTS 服务会根据特定的规则对分片进行分裂和合并，以达到更好的负载均衡，这个过程是自动的，应用无需关心。

单个分片键下所有行的大小总和不能超过 1GB。

OTS 支持的操作

OTS 支持以下种类操作:

表操作

- ListTable -- 列出实例下的所有表
- CreateTable -- 创建表
- DeleteTable -- 删除表
- DescribeTable -- 获取表的属性信息
- UpdateTable -- 更新表的预留读写吞吐量配置

详细内容可以阅读OTS 表一章

数据操作

OTS 数据操作有以下三种类型:

- 单行操作
  - GetRow -- 读取单行数据
  - PutRow -- 新插入一行。如果该行内容已经存在，先删除旧行，再写入新行
  - UpdateRow -- 更新一行。应用可以增加、删除一行中的属性列，或者更新已经存在的属性列的值。如果该行不存在，那么新增一行
  - DeleteRow -- 删除一行
- 批量操作
  - BatchGetRow -- 批量读取一张或者多张表的多行数据
  - BatchWriteRow -- 批量插入、更新、删除一张表或者多张表的多行数据
- 范围读取
  - GetRange -- 读取表中一个范围内的数据

OTS 写操作特性

OTS 写操作具有以下特性:

- 原子性

  PutRow、UpdateRow、DeleteRow 操作的结果保证原子性，要么全部成功，要么全部失败，不会存在中间状态。

- 强一致性

  应用程序获得写操作成功的响应后，该次操作的修改会立即生效，应用程序可以读取到该行最新的修改

OTS 提供 BatchWriteRow 操作对多个单行写操作进行聚集，应用程序可以将多个 PutRow、UpdateRow、DeleteRow 操作放到一个 BatchWriteRow 操作中。需要特别注意的是，BatchWriteRow 操作是多个单行写操作的聚集，本身不保证原子性，可能会出现部分行操作执行成功，部分行操作执行失败的情况，但是 BatchWriteRow 的子操作具有原子性。

详细内容可以阅读OTS 数据操作一章

预留读写吞吐量简介

预留读写吞吐量是表的一个属性。应用程序在创建表的时候，必须为该表指定预留读写吞吐量。OTS 根据该配置为表分配和预留足够的资源，从而确保可预期的性能。应用程序可以通过 UpdateTable 操作动态修改表的预留读写吞吐量配置。

预留读写吞吐量的单位为读服务能力单元和写服务能力单元。应用程序在配置表的预留读写吞吐量时，需要指定读服务能力单元和写服务能力单元这两个数值。1 单位读能力表示该表每秒可以读 1KB 数据，1 单位写能力表示每秒可以写 1KB 数据。操作数据大小不足 1KB 的部分向上取整，如写入 1.6KB 数据消耗 2 单位写能力，读出 0.1KB 数据消耗 1 单位读能力。

当表上的操作过于频繁导致预留读写吞吐量不足以进行更多操作时，OTS 会返回 OTSQuotaExhausted 错误给应用程序。此时，应用程序需要减少访问该表的频率，或者提升该表预留读写吞吐量的配置。

更多详细信息可以参考OTS 表和计量计费

节点和实例

节点

节点是指阿里云服务节点，OTS 服务会部署在多个阿里云服务节点内，用户可以根据自身的业务需求选择不同节点内的 OTS 服务。

实例

实例是用户使用和管理 OTS 服务的实体，用户在开通 OTS 服务之后，需要通过管理控制台来创建实例，然后在实例内进行表的创建和管理。实例是 OTS 资源管理的基础单元，OTS 对应用程序的访问控制和资源计量都在实例级别完成。

通常用户会为不同的业务创建不同的实例来管理相关的表，也可以为同一个业务的开发测试和生产环境创建不同的实例。OTS 允许一个云账号最多创建 10 个实例，每个实例内最多创建 10 张表。如果您需要增加限额，请联系阿里云客服人员。

实例的名字在单个节点内必须唯一，用户可以在不同的节点内创建名字相同的实例。实例命名规范如下:

- 必须由英文字母、数字和“-”组成

- 首字符必须为英文字母
- 末尾字符不能为“-”
- 大小写不敏感
- 长度在 3~16Byte 之间

服务地址

每一个实例对应一个服务地址，应用程序在进行表和数据操作时需要指定服务地址。

如果应用程序从公网访问 OTS，使用如下格式的服务地址:

```
http://<Instance Name>.<region>.ots.aliyuncs.com
```

如果应用程序从 ECS 服务器访问 OTS，使用如下格式的服务地址:

```
http://<Instance Name>.<region>.ots-internal.aliyuncs.com
```

示例，杭州节点，实例名称为 myInstance 的服务地址:

```
公网服务地址
http://myInstance.cn-hangzhou.ots.aliyuncs.com
内网服务地址
http://myInstance.cn-hangzhou.ots-internal.aliyuncs.com
```

应用程序从 ECS 服务器上通过内网访问 OTS，可以获得更低的响应延迟，也不产生外网流量。

第 2 章　OTS 表

创建 OTS 表时，需要指定表名、主键和预留读写吞吐量。在传统数据库中，表具有预定义的结构信息，比如表的名称、主键、列名和类型，表中的所有行都具有相同的列集合。OTS 是 NoSQL 数据库: 除了主键需要格式外，没有其他的格式信息。本章将介绍表的概念和使用。

表名

OTS 表的名称必须符合以下约束:

- 由英文字符 (a-z)(A-Z)，数字 (0-9)，下划线 (_) 组成
- 首字母必须为英文字母 (a-z)(A-Z) 或下划线 (_)
- 大小写敏感
- 长度在 1~255 字符之间
- 同一个实例下不能有同名的表，但不同实例内的表名字可以相同

主键

创建 OTS 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。OTS 对主键列的名字和类型都有限制，详细信息可以参考 OTS 数据模型的主键一节

OTS 根据表的主键索引数据，表中的行按照它们的主键进行升序排序。

配置预留读写吞吐量

为了确保应用获得稳定、低延时的服务，OTS 要求应用在创建表的时候指定预留读写吞吐量。OTS 根据预留读写吞吐量来为表分配相应的资源，满足应用的吞吐量需求，同时根据配置的预留读写吞吐量收取相应的费用。应用可以根据自身的业务需求动态上调和下调表的预留读写吞吐量。预留读写吞吐量通过读服务能力单元和写服务能力单元这两个数值来设置。

通过 UpdateTable 操作可以更新表的预留读写吞吐量。预留读写吞吐量的更新有如下规则:

- 一张表上的两次更新的间隔必须大于 10 分钟，例如 12:43 AM 更新了某个表的预留读写吞吐量，那么只有在 12:53 AM 之后才能再次更新该表的预留读写吞吐量。更新间隔必须大于 10 分钟的限制是针对表的，12:43 AM ~ 12:53 AM 之间用户还能更新其他表的预留读写吞吐量
- 每个自然日内 (UTC 时间 00:00:00 到第二天的 00:00:00，北京时间早上 8 点到第二天早上 8 点)，下调预留读写吞吐量的总次数不能超过 4 次。下调预留读写吞吐量的定义是，只要读服务能力单元或写服务能力单元配置其中一项进行了下调，则此次操作被视为下调预留读写吞吐量。
- 预留读写吞吐量调整完毕后 1 分钟内生效

用户的应用在初始阶段可能访问压力不大，可以为表设置较小的预留读写吞吐量。当业务量增大后，再提高表的预留读写吞吐量，满足业务需求。用户刚建表之后如果需要快速导入数据，可以设置较大的预留读写吞吐量，让数据能被快速导入进来，当数据导入完毕后，再将预留读写吞吐量下调。

写服务能力单元

写服务能力单元用来表示每秒写入 1KB 数据的数目，不足 1KB 的部分向上取整。举例说明，某张表的写服务能力单元为 10，那么应用程序在每秒中可以向该表写入 10 个 1KB 的行数据，或者向该表写入 5 个 2KB 的行数据。当行的数据大小不足 1KB 时，应用程序每秒最多也只能写入 10 个这样的行。写请求失败也会至少消耗 1 单位写服务能力单元。操作消耗的写服务能力单元的计算方式如下：

- PutRow - 旧的行数据大小 (如不存在则为 0) 和修改后的行数据大小之和，除以 1KB 向上取整。如果操作不满足应用指定的行存在性检查条件，则操作失败并消耗 1 单位写服务能力单元

- UpdateRow - 旧的行数据大小 (如不存在则为 0) 和修改后的行数据大小较大的一方，除以 1KB 向上取整。如果操作不满应用程序指定的行存在性检查条件，则操作失败并消耗 1 单位写服务能力单元
- DeleteRow - 被删除的行数据大小除以 1KB 向上取整。如果操作不满足应用程序指定的行存在性检查条件，则操作失败并消耗 1 单位写服务能力单元

读服务能力单元

读服务能力单元用来表示每秒读出 1KB 数据的数目，不足 1KB 的部分向上取整。举例说明，某张表的读服务能力单元为 10，那么应用程序在每秒中可以向该表读出 10 个 1KB 的行数据，或者向该表读出 5 个 2KB 的行数据。当行的数据大小不足 1KB 时，应用程序每秒最多也只能读出 10 个这样的行。读请求失败也会至少消耗 1 单位读服务能力单元。操作消耗的读服务能力单元的计算方式如下:

- GetRow - 被读取的行数据大小除以 1KB 向上取整。如果行不存在，则消耗 1 单位读服务能力单元
- GetRange - 被读取到的所有行数据大小除以 1KB 向上取整，如果指定的范围内没有数据，则消耗 1 读服务能力单元

分片键下的数据量限制

OTS 按照分片键的范围对表的数据进行分区，拥有相同分片键的行会被划分到同一个分片。为了防止分片过大无法切分，OTS 对单个分片键下的数据量做出了限制。单个分片键下的所有行数据大小之和不能超过 1GB。

分片键下的数据量超出 1GB 限制之后，任何导致该分片键下的数据量继续增加的操作都会失败，此时应用程序应该调用删除表数据的操作，清理数据后再继续进行写入。

建表后加载时间

OTS 表在被创建之后需要 1 分钟进行加载，在此期间对该表的读写数据操作均会失败。应用程序应该等待表加载完毕后再进行数据操作。

最佳实践

这一节将会提供一些关于使用 OTS 表的建议。

设计良好的主键

OTS 会根据表的分片键将表的数据切分成多个分片，表上配置的预留读写吞吐量会被均匀的分摊到多个分片上。举例来说，假设表被分成 10 个分片，表上的读服务能力单元为 100，那

么每个分片上分得 10 个读服务能力单元。因此，为了充分利用表上配置的预留读写吞吐量，应用程序需要让数据的分布和访问量的分布尽可能的均匀。

OTS 会对表中的行按主键进行排序，合理地设计主键可以让数据在分片上的分布更加均匀，从而能够充分的利用表上配置的预留读写吞吐量，降低成本。

分片键的选取建议遵循以下几个原则:

- 单个分片键中的数据不宜过大 (不能超过 1GB)
- 同一张表不同分片键中的数据在逻辑上独立
- 访问压力不要集中在小范围连续的分片键中

假设我们现在有这样一张表，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。

那我们该如何设计 OTS 表的主键，才能更高效的利用 OTS 呢？

考虑表的分片键:

- 使用 CardID 作为表的分片键

  使用 CardID 作为表的分片键是一个较好的选择。每天每张卡产生的消费记录数从总体上来讲是均匀的，每一个分片键中的访问压力也应该是均匀的。以 CardID 作为表的分片键可以较好地利用预留读写吞吐量资源

- 使用 SellerID 作为表的分片键

  使用 SellerID 作为表的分片键不是一个较好的选择。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 作为表的分片键

  使用 DeviceID 作为表的分片键是一个较好的选择。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 作为表的分片键也可以保证访问压力的相对均匀。

- 使用 OrderNumber 作为表的分片键

  使用 OrderNumber 作为表的分片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片，预留读写吞吐量没有得到高效的利用。如果必须使用 OrderNumber 作为分片键，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

综上，我们可以根据需求使用 CardID 和和 DeviceID 作为表的分片键，而不应该用 **SellerID** 和 **OrderNumber**。之后再根据应用的实际需求来设计剩余的主键列

并行写入数据

OTS 表会被切分成多个分片，这些分片被分散在多个 OTS 服务器上。如果有一批数据要上传到 OTS 中，同时这批数据是按主键排好顺序的，按顺序写入数据，可能会导致写入压力集中在某个分片中，而其他的分片处于空闲状态，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。保证写入数据均匀的分配在各个分片键上。
- 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

数据往往是具有时效性的。例如在设计良好的主键一节中提到的存储消费记录的表。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 作为分片键，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写服务能力单元，较大的读服务能力单元。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(Open Storage Service) 归档，或直接删除。

使用 OTS Java SDK 进行表操作

使用 OTS Java SDK 进行表操作一般遵循以下几个步骤:

1. 创建 OTSClient 对象，在构造函数中指定 EndPoint，AccessKeyID，AccessKeySecret，实例名字。
2. 构造请求对象。
3. 调用 OTSClient 对象的接口发送操作请求。

CreateTable

应用程序可以通过 createTable 接口创建表。创建表时需要指定表的名字，主键的组成和预留读写吞吐量。

下面的代码展示如何创建一张新表，该表的主键由两个 Integer 类型的主键列组成，**ReservedThroughput 分别为 100 读服务能力单元和 100 写服务能力单元。

```
final String endPoint = "<your endpoint>";
final String accessId = "<your access id>";
final String accessKey = "<your access key>";
final String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String tableName = "myTable";
String COLUMN_GID_NAME = "gid";
String COLUMN_UID_NAME = "uid";

TableMeta tableMeta = new TableMeta(tableName);
tableMeta.addPrimaryKeyColumn(COLUMN_GID_NAME, PrimaryKeyType.INTEGER);
tableMeta.addPrimaryKeyColumn(COLUMN_UID_NAME, PrimaryKeyType.INTEGER);
// 将该表的读写CU都设置为100
CapacityUnit capacityUnit = new CapacityUnit(100, 100);

CreateTableRequest request = new CreateTableRequest();
request.setTableMeta(tableMeta);
request.setReservedThroughput(capacityUnit);
client.createTable(request);

System.out.println("表已创建");
```

createTable 没有返回值。如果创建表失败，对应的异常被抛出。

DeleteTable

应用程序可以通过 deleteTable 接口删除表。删除表时需要指定表的名字。

下面的代码删除了一张名为 myTable 的表。

```
final String endPoint = "<your endpoint>";
final String accessId = "<your access id>";
final String accessKey = "<your access key>";
final String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String tableName = "myTable";

DeleteTableRequest request = new DeleteTableRequest();
request.setTableName(tableName);
client.deleteTable(request);

System.out.println("表已删除");
```

deleteTable 没有返回值。如果删除表失败，对应的异常被抛出。

ListTable

应用程序可以通过 listTable 接口获取实例内的所有表列表。

下面的代码获取实例中的所有表

```
final String endPoint = "<your endpoint>";
final String accessId = "<your access id>";
final String accessKey = "<your access key>";
final String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

ListTableResult result = client.listTable();
System.out.println("表的列表如下：");
for (String tableName : result.getTableNames()) {
    System.out.println(tableName);
}
```

UpdateTable

应用程序可以通过 updateTable 接口更新表的预留读写吞吐量。

下面的代码将 myTable 的读服务能力单元和写服务能力单元分别设置为 50 和 50。

```
final String endPoint = "<your endpoint>";
final String accessId = "<your access id>";
final String accessKey = "<your access key>";
final String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String tableName = "myTable";

// 将表的CU下调为:(50, 50), 但是由于刚创建表，所以需要10分钟之后才能调整
System.out.println("等待10分钟之后调整表的CU设置。");
Thread.sleep(10 * 60000);
UpdateTableRequest request = new UpdateTableRequest();
request.setTableName(tableName);
ReservedThroughputChange cuChange = new ReservedThroughputChange();
cuChange.setReadCapacityUnit(50); // 若想单独调整写CU，则无须设置读CU
cuChange.setWriteCapacityUnit(50); // 若想单独调整读CU，则无须设置写CU
request.setReservedThroughputChange(cuChange);
UpdateTableResult result = client.updateTable(request);

ReservedThroughputDetails reservedThroughputDetails = result.getReservedThroughputDetails
    ();
System.out.println("表的预留读吞吐量："
        + reservedThroughputDetails.getCapacityUnit().getReadCapacityUnit());
System.out.println("表的预留写吞吐量："
        + reservedThroughputDetails.getCapacityUnit().getWriteCapacityUnit());
System.out.println("最后一次上调预留读写吞吐量的时间：" + reservedThroughputDetails.
    getLastIncreaseTime());
System.out.println("最后一次下调预留读写吞吐量的时间：" + reservedThroughputDetails.
    getLastDecreaseTime());
System.out.println("UTC自然日内总的下调预留读写吞吐量的次数："
        + reservedThroughputDetails.getNumberOfDecreasesToday());
```

DescribeTable

应用程序可以通过 describeTable 接口获取表的信息。

下面的代码获取名为 myTable 的表的信息

```
final String endPoint = "<your endpoint>";
final String accessId = "<your access id>";
final String accessKey = "<your access key>";
final String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);
```

```
6
7  String tableName = "myTable";
8
9  DescribeTableRequest request = new DescribeTableRequest();
10 request.setTableName(tableName);
11 DescribeTableResult result = client.describeTable(request);
12 TableMeta tableMeta = result.getTableMeta();
13 System.out.println("表的名称：" + tableMeta.getTableName());
14 System.out.println("表的主键：");
15 for (String keyName : tableMeta.getPrimaryKey().keySet()) {
16     System.out.println(keyName + " : "
17             + tableMeta.getPrimaryKey().get(keyName));
18 }
19 ReservedThroughputDetails reservedThroughputDetails = result.getReservedThroughputDetails
       ();
20 System.out.println("表的预留读吞吐量："
21         + reservedThroughputDetails.getCapacityUnit().getReadCapacityUnit());
22 System.out.println("表的预留写吞吐量："
23         + reservedThroughputDetails.getCapacityUnit().getWriteCapacityUnit());
24 System.out.println("最后一次上调预留读写吞吐量的时间：" + reservedThroughputDetails.
       getLastIncreaseTime());
25 System.out.println("最后一次下调预留读写吞吐量的时间：" + reservedThroughputDetails.
       getLastDecreaseTime());
26 System.out.println("UTC自然日内总的下调预留读写吞吐量的次数："
27         + reservedThroughputDetails.getNumberOfDecreasesToday());
```

使用 OTS Python SDK 进行表操作

使用 OTS Python SDK 进行表操作一般遵循以下几个步骤:

1. 创建 OTSClient 对象，在构造函数中指定 EndPoint，AccessKeyID，AccessKeySecret，实例名字。
2. 构造访问所需的对象。
3. 使用 OTSClient 对象相关接口发送请求。

CreateTable

应用程序可以通过 create_table 接口创建表。创建表时需要设定表的 TableMeta 和预留读写吞吐量。

下面的代码创建了一张拥有两列 PK，第一列为 String 类型，第二列为 Integer 类型，读服务能力单元为 100，写服务能力单元为 100，名字为 myTable 的表。

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

schema_of_primary_key = [('gid', 'INTEGER'), ('uid', 'INTEGER')]
table_meta = TableMeta('myTable', schema_of_primary_key)
reserved_throughput = ReservedThroughput(CapacityUnit(100, 100))
ots_client.create_table(table_meta, reserved_throughput)
print u'表已创建。'
```

create_table 没有返回值。如果创建表失败，对应的异常被抛出。

DeleteTable

应用程序可以通过 delete_table 接口删除表。删除表时需要指定表的名字。

下面的代码删除了一张名为 myTable 的表

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

try:
    ots_client.delete_table('myTable')
except OTSServiceError,e:
    print e.get_error_code()
    print e.get_error_message()
print u'表已删除。'
```

delete_table 没有返回值。如果删除表失败，对应的异常被抛出。

ListTable

应用程序可以通过 list_table 接口获取实例内的所有表名字。

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
```

```
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

list_response = ots_client.list_table()
print u'表的列表如下：'
for table_name in list_response:
    print table_name
```

UpdateTable

应用程序可以通过 update_table 接口更新表的预留读写吞吐量。

下面的代码将 myTable 的读服务能力单元和写服务能力单元分别设置为 50 和 50。

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

try:
    # 由于刚创建表，需要10分钟之后才能调整表的预留读写吞吐量。
    reserved_throughput = ReservedThroughput(CapacityUnit(50, 50))
    update_response = ots_client.update_table('myTable', reserved_throughput)
    print u'表的预留读吞吐量：%s' % update_response.reserved_throughput_details.
        capacity_unit.read
    print u'表的预留写吞吐量：%s' % update_response.reserved_throughput_details.
        capacity_unit.write
    print u'最后一次上调预留读写吞吐量时间：%s' % update_response.
        reserved_throughput_details.last_increase_time
    print u'最后一次下调预留读写吞吐量时间：%s' % update_response.
        reserved_throughput_details.last_decrease_time
    print u'UTC自然日内总的下调预留读写吞吐量次数：%s' % update_response.
        reserved_throughput_details.number_of_decreases_today
except OTSServiceError,e:
    print e.get_error_code()
    print e.get_error_message()
```

返回的结果是一个 ots2.metadata.UpdateTableResponse 类的实例。如果请求失败，对应的异常被抛出。

DescribeTable

应用程序可以通过 describe_table 接口获取表的信息。

下面的代码获取名为 myTable 的表的信息

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

describe_response = ots_client.describe_table('myTable')
print u'表的名称: %s' % describe_response.table_meta.table_name
print u'表的主键: %s' % describe_response.table_meta.schema_of_primary_key
print u'表的预留读吞吐量: %s' % describe_response.reserved_throughput_details.
    capacity_unit.read
print u'表的预留写吞吐量: %s' % describe_response.reserved_throughput_details.
    capacity_unit.write
print u'最后一次上调预留读写吞吐量时间: %s' % describe_response.
    reserved_throughput_details.last_increase_time
print u'最后一次下调预留读写吞吐量时间: %s' % describe_response.
    reserved_throughput_details.last_decrease_time
print u'UTC自然日内总的下调预留读写吞吐量次数: %s' % describe_response.
    reserved_throughput_details.number_of_decreases_today
```

返回的结果是一个 ots2.metadata.DescribeTableResponse 类的实例。如果请求失败，对应的异常被抛出。

第 3 章　OTS 数据操作

OTS 的表由行组成。每一行包含主键和属性。这一章将介绍操作 OTS 数据的方法。

OTS 行简介

组成 OTS 表的基本单位是行，行由主键和属性组成，其中主键是必须的，且每一行的主键列的名称和类型相同，属性不是必须的，并且每一行的属性可以不同。更多信息请参考 OTS 数据模型。

OTS 数据操作有以下三种类型:

- 单行操作
  - GetRow -- 读取单行数据
  - PutRow -- 新插入一行。如果该行内容已经存在，先删除旧行，再写入新行
  - UpdateRow -- 更新一行。应用可以增加、删除一行中的属性列，或者更新已经存在的属性列的值。如果该行不存在，那么新增一行
  - DeleteRow -- 删除一行
- 批量操作
  - BatchGetRow -- 批量读取多行数据
  - BatchWriteRow -- 批量插入、更新、删除多行数据
- 范围读取
  - GetRange -- 读取表中一个范围内的数据

本章将介绍以上几种操作。

OTS 单行操作

单行写入操作

OTS 的单行写操作有三种:PutRow，UpdateRow 和 DeleteRow。下面分别讲述每一种操作的行为语义和注意事项:

- PutRow

  新写入一行。如果这一行已经存在，则这一行旧的数据会先被删除，再新写入一行。

- UpdateRow

  更新一行的数据。OTS 会根据请求的内容在这一行中新增列，修改或者删除指定列的值。如果这一行不存在，则会插入新的一行。但是有一种特殊的场景，UpdateRow 请求只包含删除指定的列，且该行不存在，该请求不会插入新行。

- DeleteRow

  删除一行。如果删除的行不存在，则不会发生任何变化。

应用程序通过设置请求中的 condition 字段来指定写入操作执行时是否要对行的存在性进行检查。condition 有三种类型:

- IGNORE -- 不做任何存在性检查。

- EXPECT_EXIST -- 期望行存在，如果该行存在，则操作成功，如果该行不存在，则操作失败。
- EXPECT_NOT_EXIST -- 期望行不存在，如果行不存在，则操作成功，如果该行存在，则操作失败。

condition 为 EXPECT_NOT_EXIST 的 DeleteRow、UpdateRow 操作是没有意义的，删除一个不存在的行是无意义的，如果需要更新不存在的行可以使用 PutRow 操作。

如果操作发生错误，如参数检查失败，单行数据量过多，行存在性检查失败等等，会返回错误码给应用程序。如果操作成功，OTS 会将操作消耗的服务能力单元返回给应用程序。

各操作消耗的写服务能力单元的计算规则:

- PutRow - 旧的行数据大小 (如不存在则为 0) 和修改后的行数据大小之和，除以 1KB 向上取整。如果操作不满足用户的行存在性检查条件，则操作失败并消耗 1 单位写服务能力单元
- UpdateRow - 旧的行数据大小 (如不存在则为 0) 和修改后的行数据大小取最大值，除以 1KB 向上取整。如果操作不满足用户的行存在性检查条件，则操作失败并消耗 1 单位写服务能力单元
- DeleteRow - 被删除的行数据大小除以 1KB 向上取整。如果操作不满足用户的行存在性检查条件，则操作失败并消耗 1 单位写服务能力单元

写操作均不会消耗读服务能力单元。

下面举例说明单行写操作的写服务能力单元计算。

例子 1，PutRow 覆盖一行:

```
//原来的行
//row_size=len('pk')+len('value1')+8Byte+1300Byte=1316Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte)}
}

//PutRow操作
//row_size=len('pk')+len('value2')+8Byte+900Byte=916Byte
{
    primary_keys:{'pk':1},
    attributes:{'value2':String(900Byte)}
}
```

消耗的写服务能力单元为 1316Byte+916Byte 除以 1KB 向上取整，该 PutRow 操作消耗 3 个单位的写服务能力单元。

例子 2，UpdateRow 新写入一行:

```
//原来的行不存在
//row_size=0

//UpdateRow操作
//row_size=len('pk')+len('value1')+8Byte+900Byte=916Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte), 'value2':Delete}
}
```

消耗的写服务能力单元为 916Byte 除以 1KB 向上取整，该 UpdateRow 操作消耗 1 个单位的写服务能力单元。

例子 3，UpdateRow 更新已存在的行:

```
//row_size=len('pk')+len('value1')+8Byte+1300Byte=1316Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte)}
}
//UpdateRow操作
//row_size=len('pk')+len('value1')+8Byte+900Byte=916Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte)}
}
```

消耗的写服务能力单元为 1316Byte 除以 1KB 向上取整，该 UpdateRow 操作消耗 2 个单位的写服务能力单元。

例子 4，DeleteRow 删除不存在的行:

```
//原来的行不存在
//row_size=0



//DeleteRow操作
//row_size=0
{
    primary_keys:{'pk':1},
}
```

修改前和后修改后的数据大小均为 0，无论读写操作成功还是失败至少消耗一个单位服务能力单元，因此该 DeleteRow 操作消耗 1 个单位的写服务能力单元。

更多的信息可以参看API Reference中的PutRow,UpdateRow,DeleteRow章节

单行读取操作

OTS 的单行读操作只有一种:GetRow

应用程序提供完整的主键和需要返回的列名。列名可以是主键列或属性列。也可以不指定要返回的列名，此时请求返回整行数据。

OTS 根据被读取的行的数据大小计算读服务能力单元。将行数据大小除以 1KB 向上取整作为本次读取操作消耗的读服务能力单元。如果操作指定的行不存在，则消耗 1 单位读服务能力单元。单行读取操作不会消耗写服务能力单元。

例子，GetRow 读取一行消耗的读服务能力单元计算:

```
1  //被读取的行
2  //row_size=len('pk')+len('value1')+len('value2')+8Byte+200Byte+1100Byte=1322Byte
3  {
4      primary_keys:{'pk':1},
5      attributes:{'value1':String(200Byte), 'value2':String(1100Byte)}
6  }
7
8  //GetRow操作
9  {
10     primary_keys:{'pk':1},
11     columns_to_get:{'value1'}
12 }
```

消耗的读服务能力单元为 1322Byte 除以 1KB 向上取整，该 GetRow 操作消耗 2 个单位的读服务能力单元。虽然 GetRow 请求仅读取该行中较小的一列 value1，但是该请求消耗的读服务能力单元是根据整行的数据大小计算的。

更多信息可以参看API Reference 中的GetRow章节。

多行操作

OTS 提供了 BatchWriteRow 和 BatchGetRow 两种多行操作。

BatchWriteRow 用于插入、修改、删除一个表或者多个表中的多行记录。BatchWriteRow 操作由多个 PutRow、UpdateRow、DeleteRow 子操作组成。BatchWriteRow 的各个子操作独立执行，OTS 会将各个子操作的执行结果分别返回给应用程序，可能存在部分请求成功、部分请求失败的现象。即使整个请求没有返回错误，应用程序也必须要检查每个子操作返回的结果，从而拿到正确的状态。BatchWriteRow 的各个子操作单独计算写服务能力单元。

BatchGetRow 用于读取一个表或者多个表中的多行记录。BatchGetRow 各个子操作独立执行，OTS 会将各个子操作的执行结果分别返回给应用程序，可能存在部分请求成功、部分请求

失败的现象。即使整个请求没有返回错误，应用程序也必须要检查每个子操作返回的结果，从而拿到正确的状态。BatchGetRow 的各个子操作单独计算读服务能力单元。

更多信息可以参看 API Reference 中的BatchWriteRow与BatchGetRow章节。

范围读取操作

OTS 提供了范围读取操作 GetRange。该操作将指定主键范围内的数据返回给应用程序。

OTS 表中的行按主键进行从小到大排序，GetRange 的读取范围是一个左闭右开的区间。操作会返回主键属于该区间的行数据，区间的起始点是有效的主键或者是由 INF_MIN 和 INF_MAX 类型组成的虚拟点，虚拟点的列数必须与主键相同。其中，INF_MIN 表示无限小，任何类型的值都比它大，INF_MAX 表示无限大，任何类型的值都比它小。

GetRange 操作需要指定请求列名。请求列名中可以包含多个列名。如果某一行的主键属于读取的范围，但是不包含指定返回的列，那么请求返回结果中不包含该行数据。不指定请求列名，则返回完整的行。

GetRange 操作需要指定读取方向。读取方向可以为正序或逆序。假设同一表中有两个主键 A 和 B，A<B。如正序读取 [A, B)，则按从 A 至 B 的顺序返回主键大于等于 A，小于 B 的行。逆序读取 [B,A)，则按从 B 至 A 的顺序返回大于 A，小于等于 B 的数据。

GetRange 操作可以指定最大返回行数，OTS 按照正序或者逆序最多返回指定的行数之后即结束该操作的执行，即使该区间内仍有未返回的数据。

GetRange 操作可能在以下几种情况下停止执行并返回数据给应用程序：1）返回的行数据大小之和达到 1MB；2）返回的行数等于 5000；3）返回的行数等于最大返回行数。同时 GetRange 请求的返回结果中还包含下一条未读数据的主键，应用程序可以使用该返回值作为下一次 GetRange 操作的起始点继续读取。如果下一条未读数据的主键为空，表示读取区间内的数据全部返回。

OTS 计算读取区间起始点到下一条未读数据的起始点的所有行数据大小之和，如果下一条未读数据的起始点为空，则计算读取区间内的所有行数据大小之和。将数据大小之和除以 1KB 向上取整计算消耗的读服务能力单元。如读取范围中包含 10 行，每行占用数据大小为 330Byte。则消耗的读服务能力单元为 4(数据总和 3.3KB，除以 1KB 向上取整为 4)。

下面举例说明 GetRange 操作的行为。假设表的内容如下，PK1，PK2 是表的主键列，类型分别为 String 和 Integer。A，B 是表的属性列

PK1	PK2	Attr1	Attr2
'A'	2	'Hell'	'Bell'
'A'	5	'Hello'	不存在
'A'	6	不存在	'Blood'
'B'	10	'Apple'	不存在
'C'	1	不存在	不存在

PK1	PK2	Attr1	Attr2
'C'	9	'Alpha'	不存在

例子 1，读取某一范围内的数据

```
//请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 2)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INTEGER, 1)

//响应
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
    attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING, "Bell")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
    attribute_columns:("Attr1", STRING, "Hello")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
    attribute_columns:("Attr2", STRING, "Blood")
  },
  {
    primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
    attribute_columns:("Attr1", STRING, "Apple")
  }
}
```

例子 2，利用 INF_MIN，INF_MAX 读取全表数据

```
//请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", INF_MIN)
exclusive_end_primary_key: ("PK1", INF_MAX)

//响应
cosumed_read_capacity_unit: 1
```

```
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
    attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING, "Bell")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
    attribute_columns:("Attr1", STRING, "Hello")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
    attribute_columns:("Attr2", STRING, "Blood")
  },
  {
    primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
    attribute_columns:("Attr1", STRING, "Apple")
  }
  {
    primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 1)
  }
  {
    primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 9)
    attribute_columns:("Attr1", STRING, "Alpha")
  }
}
```

例子 3，在某些主键列上使用 INF_MIN，INF_MAX

```
//请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)

//响应
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
    attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING, "Bell")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
```

```
    attribute_columns:("Attr1", STRING, "Hello")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
    attribute_columns:("Attr2", STRING, "Blood")
  }
}
```

例子 4，逆序读取

```
//请求
table_name: "table_name"
direction: BACKWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INTEGER, 1)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 5)

//响应
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "C"), ("PK2", INTEGER, 1)
  },
  {
    primary_key_columns:("PK1", STRING, "B"), ("PK2", INTEGER, 10)
    attribute_columns:("Attr1", STRING, "Apple")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
    attribute_columns:("Attr2", STRING, "Blood")
  }
}
```

例子 5，指定列名不包含 PK

```
//请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MAX)
columns_to_get: "Attr1"

//响应
cosumed_read_capacity_unit: 1
rows: {
```

```
  {
    attribute_columns: {"Attr1", STRING, "Alpha"}
  }
}
```

例子 6，指定列名中包含 PK

```
//请求
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "C"), ("PK2", INF_MAX)
columns_to_get: "Attr1", "PK1"

//响应
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "C")
  }
  {
    primary_key_columns:("PK1", STRING, "C")
    attribute_columns:("Attr1", STRING, "Alpha")
  }
}
```

例子 7，使用 limit 和断点

```
//请求1
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MIN)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)
limit: 2

//响应1
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
    attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING, "Bell")
  },
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
```

```
    attribute_columns:("Attr1", STRING, "Hello")
  }
}
next_start_primary_key:("PK1", STRING, "A"), ("PK2", INTEGER, 6)

//请求2
table_name: "table_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", STRING, "A"), ("PK2", INTEGER, 6)
exclusive_end_primary_key: ("PK1", STRING, "A"), ("PK2", INF_MAX)
limit: 2

//响应2
cosumed_read_capacity_unit: 1
rows: {
  {
    primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 6)
    attribute_columns:("Attr2", STRING, "Blood")
  }
}
```

例子 8，GetRange 操作消耗的读服务能力单元计算。在以下表中执行 GetRange 操作，其中 pk 是表的主键列，Attr1、Attr2 是表的属性列。

pk	Attr1	Attr2	row_size
1	不存在	String(1100Byte)	1115Byte
2	8	String(1000Byte)	1028Byte
3	不存在	String(1000Byte)	1015Byte
4	String(1000Byte)	String(1000Byte)	2020Byte

```
//请求
table_name: "table2_name"
direction: FORWARD
inclusive_start_primary_key: ("pk", INTEGER, 1)
exclusive_end_primary_key: ("pk", INTEGER, 2)
columns_to_get: "pk", "Attr1"

//响应
cosumed_read_capacity_unit: 4
rows: {
```

```
  {
    primary_key_columns:("pk", INTEGER, 1)
  },
  {
    primary_key_columns:("pk", INTEGER, 2)
    attribute_columns:("Attr1", INTEGER, 8)
  },
  {
    primary_key_columns:("pk", INTEGER, 3)
  },
}
```

消耗的读服务能力单元为 1115Byte+1028Byte+1015Byte=3158Byte 除以 1KB 向上取整，该 GetRange 操作消耗 4 个单位的读服务能力单元。虽然 GetRange 请求仅读取较小的一列'Attr1'，读到的数据不足 1KB。但是依然使用范围内的行数据大小计算消耗的读服务能力单元。

更多详细信息可以参看 API Reference 中的GetRange章节

最佳实践

这一节将会提供一些关于 OTS 数据操作的建议

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，不同访问频率的属性列放到不同的表中。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。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。

OTS 使用行数据大小计算服务能力单元。如果该行较大，但写入操作仅仅更新一行中小部分数据，访问操作仍然根据该行的总大小计算服务能力单元。拆分表后能减少被访问行的数据大小，减少消耗的服务能力单元。从而降低使用 OTS 的成本。

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 OTS 中。这样做节省了空间，减少了访问的服务能力单元消耗，从而降低使用 OTS 的成本。

将数据量超出限制的属性列存储到 OSS 中

OTS 限制单个属性列值不超过 64KB。如果需要存储单个值超过 64KB 的需求，如图片、音乐、文件等，可以使用 OSS(Open 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 OTS 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 64KB 的单个值拆分成多个行存储在 OTS 中。

错误重试加入时间间隔

OTS 可能遇到软硬件问题，导致应用程序的部分请求失败并返回可重试的错误 (详情见OTS错误信息)。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。

使用 OTS Java SDK 进行数据操作

使用 OTS Java SDK 进行数据操作一般遵循以下几个步骤:

1. 创建 OTSClient 对象，在构造函数中指定 EndPoint，AccessKeyID，AccessKeySecret，实例名字。
2. 构造请求对象。
3. 调用 OTSClient 对象相关接口发送请求。

PutRow

应用程序可以通过 putRow 接口插入或覆盖一行。

下面的代码向表中添加了一行，该表的主键包含两个 Integer 类型的主键列。被添加的行拥有 4 个属性列。行存在性检查条件为 EXPECT_NOT_EXIST，即只有该行不存在时才插入数据.

```
1  String endPoint = "<your endpoint>";
2  String accessId = "<your access id>";
3  String accessKey = "<your access key>";
4  String instanceName = "<your instance name>";
5  OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);
6
7  String COLUMN_GID_NAME = "gid";
8  String COLUMN_UID_NAME = "uid";
9  String COLUMN_NAME_NAME = "name";
10 String COLUMN_ADDRESS_NAME = "address";
11 String COLUMN_AGE_NAME = "age";
12 String COLUMN_MOBILE_NAME = "mobile";
13 String tableName = "myTable";
```

```
14
15 RowPutChange rowChange = new RowPutChange(tableName);
16 RowPrimaryKey primaryKey = new RowPrimaryKey();
17 primaryKey.addPrimaryKeyColumn(COLUMN_GID_NAME, PrimaryKeyValue.fromLong(1));
18 primaryKey.addPrimaryKeyColumn(COLUMN_UID_NAME, PrimaryKeyValue.fromLong(101));
19 rowChange.setPrimaryKey(primaryKey);
20 rowChange.addAttributeColumn(COLUMN_NAME_NAME, ColumnValue.fromString("张三"));
21 rowChange.addAttributeColumn(COLUMN_MOBILE_NAME, ColumnValue.fromString("111111111"));
22 rowChange.addAttributeColumn(COLUMN_ADDRESS_NAME, ColumnValue.fromString("中国A地"));
23 rowChange.addAttributeColumn(COLUMN_AGE_NAME, ColumnValue.fromLong(20));
24 rowChange.setCondition(new Condition(RowExistenceExpectation.EXPECT_NOT_EXIST));
25
26 PutRowRequest request = new PutRowRequest();
27 request.setRowChange(rowChange);
28
29 PutRowResult result = client.putRow(request);
30 int consumedWriteCU = result.getConsumedCapacity().getCapacityUnit().getWriteCapacityUnit
       ();
31
32 System.out.println("成功插入数据，消耗的写CapacityUnit为：" + consumedWriteCU);
```

UpdateRow

应用程序可以通过 updateRow 接口更新一行。

下面的例子更新了一行，插入或修改属性中 name，address，删除 mobile，age。行存在性检查条件为 EXPECT_EXIST，即在该行存在的条件下才更新该行，否则更新失败。

```
1  String endPoint = "<your endpoint>";
2  String accessId = "<your access id>";
3  String accessKey = "<your access key>";
4  String instanceName = "<your instance name>";
5  OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);
6
7  String COLUMN_GID_NAME = "gid";
8  String COLUMN_UID_NAME = "uid";
9  String COLUMN_NAME_NAME = "name";
10 String COLUMN_ADDRESS_NAME = "address";
11 String COLUMN_AGE_NAME = "age";
12 String COLUMN_MOBILE_NAME = "mobile";
13
14 RowUpdateChange rowChange = new RowUpdateChange(tableName);
15 RowPrimaryKey primaryKeys = new RowPrimaryKey();
```

```
primaryKeys.addPrimaryKeyColumn(COLUMN_GID_NAME, PrimaryKeyValue.fromLong(1));
primaryKeys.addPrimaryKeyColumn(COLUMN_UID_NAME, PrimaryKeyValue.fromLong(101));
rowChange.setPrimaryKey(primaryKeys);
// 更新以下三列的值
rowChange.addAttributeColumn(COLUMN_NAME_NAME, ColumnValue.fromString("张三"));
rowChange.addAttributeColumn(COLUMN_ADDRESS_NAME, ColumnValue.fromString("中国B地"));
// 删除mobile和age信息
rowChange.deleteAttributeColumn(COLUMN_MOBILE_NAME);
rowChange.deleteAttributeColumn(COLUMN_AGE_NAME);

rowChange.setCondition(new Condition(RowExistenceExpectation.EXPECT_EXIST));

UpdateRowRequest request = new UpdateRowRequest();
request.setRowChange(rowChange);

UpdateRowResult result = client.updateRow(request);
int consumedWriteCU = result.getConsumedCapacity().getCapacityUnit().getWriteCapacityUnit
    ();

System.out.println("成功更新数据，消耗的写CapacityUnit为：" + consumedWriteCU);
```

DeleteRow

应用程序可以通过 deleteRow 接口删除一行。

下面的例子删除了一行。行存在性检查条件为 IGNORE，即不进行任何行存在性检查。

```
String endPoint = "<your endpoint>";
String accessId = "<your access id>";
String accessKey = "<your access key>";
String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String COLUMN_GID_NAME = "gid";
String COLUMN_UID_NAME = "uid";

RowDeleteChange rowChange = new RowDeleteChange(tableName);
RowPrimaryKey primaryKeys = new RowPrimaryKey();
primaryKeys.addPrimaryKeyColumn(COLUMN_GID_NAME, PrimaryKeyValue.fromLong(1));
primaryKeys.addPrimaryKeyColumn(COLUMN_UID_NAME, PrimaryKeyValue.fromLong(101));
rowChange.setPrimaryKey(primaryKeys);

DeleteRowRequest request = new DeleteRowRequest();
```

```
request.setRowChange(rowChange);

DeleteRowResult result = client.deleteRow(request);
int consumedWriteCU = result.getConsumedCapacity().getCapacityUnit().getWriteCapacityUnit
    ();

System.out.println("成功删除数据，消耗的写CapacityUnit为：" + consumedWriteCU);
```

GetRow

应用程序可以通过 getRow 接口读取一行。

下面的代码读取表中的一行中的 name，address，age 列。

```
String endPoint = "<your endpoint>";
String accessId = "<your access id>";
String accessKey = "<your access key>";
String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String COLUMN_GID_NAME = "gid";
String COLUMN_UID_NAME = "uid";
String COLUMN_NAME_NAME = "name";
String COLUMN_ADDRESS_NAME = "address";
String COLUMN_AGE_NAME = "age";

SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(tableName);
RowPrimaryKey primaryKeys = new RowPrimaryKey();
primaryKeys.addPrimaryKeyColumn(COLUMN_GID_NAME, PrimaryKeyValue.fromLong(1));
primaryKeys.addPrimaryKeyColumn(COLUMN_UID_NAME, PrimaryKeyValue.fromLong(101));
criteria.setPrimaryKey(primaryKeys);
criteria.addColumnsToGet(new String[] {
        COLUMN_NAME_NAME,
        COLUMN_ADDRESS_NAME,
        COLUMN_AGE_NAME
});

GetRowRequest request = new GetRowRequest();
request.setRowQueryCriteria(criteria);
GetRowResult result = client.getRow(request);
Row row = result.getRow();

int consumedReadCU = result.getConsumedCapacity().getCapacityUnit().getReadCapacityUnit();
```

```
System.out.println("本次读操作消耗的读CapacityUnti为：" + consumedReadCU);
System.out.println("name信息：" + row.getColumns().get(COLUMN_NAME_NAME));
System.out.println("address信息：" + row.getColumns().get(COLUMN_ADDRESS_NAME));
System.out.println("age信息：" + row.getColumns().get(COLUMN_AGE_NAME));
```

BatchWriteRow

应用程序可以通过 batchWriteRow 接口插入、更新、删除一张表或多张表中的多行。OTS 会将各个子操作的执行结果分别返回给应用程序，可能存在部分请求成功、部分请求失败的现象。即使整个请求没有返回错误，应用程序也必须要检查每个子操作返回的结果，从而拿到正确的状态。

下面的代码向 5 张表中分别插入 10 行数据，同时对写入失败的行进行重试，最大重试次数为 3。这 5 张表的主键组成相同，均是由 1 列 Integer 类型的主键列与 1 列 String 类型的主键列组成的。

```
String endPoint = "<your endpoint>";
String accessId = "<your access id>";
String accessKey = "<your access key>";
String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String COLUMN_UID_NAME = "uid";
String COLUMN_NAME_NAME = "name";
String COLUMN_ADDRESS_NAME = "address";
String COLUMN_GID_NAME = "groupid";
String tableNamePrefix = "sampleTable_";
int TABLE_COUNT = 5;

System.out.println("\n############# 开始BatchWriteRow操作 #############");
BatchWriteRowRequest request = new BatchWriteRowRequest();
// 向TABLE_COUNT张表中插入数据， 每张表插入10行。
for (int i = 0; i < TABLE_COUNT; ++i) {
    for (int j = 0; j < 10; ++j) {

        RowPutChange rowPutChange = new RowPutChange(tableNamePrefix
                + i);
        RowPrimaryKey primaryKey = new RowPrimaryKey();
        primaryKey.addPrimaryKeyColumn(COLUMN_UID_NAME,
                PrimaryKeyValue.fromLong(1));
        primaryKey.addPrimaryKeyColumn(
                COLUMN_GID_NAME,
                PrimaryKeyValue.fromString(String.format("%03d", j)));
```

```
        rowPutChange.setPrimaryKey(primaryKey);

        rowPutChange.addAttributeColumn(COLUMN_ADDRESS_NAME,
                ColumnValue.fromString("中国某地"));
        rowPutChange.addAttributeColumn(COLUMN_NAME_NAME,
                ColumnValue.fromString("张三" + j));
        request.addRowPutChange(rowPutChange);
    }
}
// batchWriteRow接口会返回一个结果集， 结果集中包含的结果个数与插入的行数相同。
    结果集中的结果不一定都是成功，
// 用户需要自己对不成功的操作进行重试。
BatchWriteRowResult result = client.batchWriteRow(request);
BatchWriteRowRequest failedOperations = null;
int succeedCount = 0;

int retryCount = 0;
do {
    Map<String, List<RowStatus>> status = result.getPutRowStatus();
    failedOperations = new BatchWriteRowRequest();
    for (Entry<String, List<RowStatus>> entry : status.entrySet()) {
        String tableName = entry.getKey();
        List<RowStatus> statuses = entry.getValue();
        for (int index = 0; index < statuses.size(); index++) {
            RowStatus rowStatus = statuses.get(index);
            if (!rowStatus.isSucceed()) {
                // 操作失败， 需要放到重试列表中再次重试
                // 需要重试的操作可以从request中获取参数
                failedOperations.addRowPutChange(request
                        .getRowPutChange(tableName, index));
            } else {
                succeedCount++;
                System.out.println("本次操作消耗的写CapacityUnit为：" + rowStatus.
                    getConsumedCapacity().getCapacityUnit().getWriteCapacityUnit());
            }
        }
    }

    if (failedOperations.isEmpty() || ++retryCount > 3) {
        break; // 如果所有操作都成功了或者重试次数达到上线， 则不再需要重试。
    }

```

```
    // 如果有需要重试的操作， 则稍微等待一会后再次重试， 否则继续出错的概率很高。
    try {
        Thread.sleep(100); // 100ms后继续重试
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    request = failedOperations;
    result = client.batchWriteRow(request);
} while (true);

System.out.println(String.format("成功插入%d行数据。", succeedCount));
```

BatchGetRow

batchGetRow 用于读取一个表或者多个表中的多行记录。BatchGetRow 各个子操作独立执行，OTS 会将各个子操作的执行结果分别返回给应用程序，可能存在部分请求成功、部分请求失败的现象。即使整个请求没有返回错误，应用程序也必须要检查每个子操作返回的结果，从而拿到正确的状态。

下面的代码从 1 张表中读取 10 行数据，并对读取失败的行进行重试，最大重试次数为 3。表的主键是由 1 列 Integer 类型的主键列与 1 列 String 类型的主键列组成的。

```
String endPoint = "<your endpoint>";
String accessId = "<your access id>";
String accessKey = "<your access key>";
String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String COLUMN_UID_NAME = "uid";
String COLUMN_NAME_NAME = "name";
String COLUMN_ADDRESS_NAME = "address";
String COLUMN_GID_NAME = "groupid";
String tableName = "sampleTable_0";

BatchGetRowRequest request = new BatchGetRowRequest();
MultiRowQueryCriteria tableRows = new MultiRowQueryCriteria(tableName);
for (int j = 0; j < 10; ++j) {
    RowPrimaryKey primaryKeys = new RowPrimaryKey();
    primaryKeys.addPrimaryKeyColumn(COLUMN_UID_NAME,
            PrimaryKeyValue.fromLong(1));
    primaryKeys.addPrimaryKeyColumn(
            COLUMN_GID_NAME,
```

```
            PrimaryKeyValue.fromString(String.format("%03d", j)));

    tableRows.addRow(primaryKeys);
}
tableRows.addColumnsToGet(new String[] { COLUMN_NAME_NAME,
        COLUMN_GID_NAME, COLUMN_UID_NAME, COLUMN_ADDRESS_NAME });
request.addMultiRowQueryCriteria(tableRows);

BatchGetRowResult result = client.batchGetRow(request);
BatchGetRowRequest failedOperations = null;

List<Row> succeedRows = new ArrayList<Row>();

int retryCount = 0;
do {
    failedOperations = new BatchGetRowRequest();

    Map<String, List<com.aliyun.openservices.ots.model.BatchGetRowResult.RowStatus>>
        status = result
            .getTableToRowsStatus();
    for (Entry<String, List<com.aliyun.openservices.ots.model.BatchGetRowResult.RowStatus
        >> entry : status
            .entrySet()) {
        tableName = entry.getKey();
        tableRows = new MultiRowQueryCriteria(tableName);
        List<com.aliyun.openservices.ots.model.BatchGetRowResult.RowStatus> statuses =
            entry
                .getValue();
        for (int index = 0; index < statuses.size(); index++) {
            com.aliyun.openservices.ots.model.BatchGetRowResult.RowStatus rowStatus =
                statuses
                    .get(index);
            if (!rowStatus.isSucceed()) {
                // 操作失败， 需要放到重试列表中再次重试
                // 需要重试的操作可以从request中获取参数
                tableRows.addRow(request
                        .getPrimaryKey(tableName, index));
            } else {
                succeedRows.add(rowStatus.getRow());
                System.out.println("成功读取一行数据:");
                System.out.println("uid信息为: " + rowStatus.getRow().getColumns().get(
                    COLUMN_UID_NAME));
```

```
                System.out.println("gid信息为：" + rowStatus.getRow().getColumns().get(
                    COLUMN_GID_NAME));
                System.out.println("name信息为：" + rowStatus.getRow().getColumns().get(
                    COLUMN_NAME_NAME));
                System.out.println("address信息为：" + rowStatus.getRow().getColumns().get
                    (COLUMN_ADDRESS_NAME));
                System.out.println("本次读操作消耗的读CapacityUnti为：" + rowStatus.
                    getConsumedCapacity().getCapacityUnit().getReadCapacityUnit());
            }
        }

        if (!tableRows.getRowKeys().isEmpty()) {
            tableRows.addColumnsToGet(new String[] { COLUMN_NAME_NAME,
                    COLUMN_GID_NAME, COLUMN_UID_NAME });
            failedOperations.addMultiRowQueryCriteria(tableRows);
        }
    }

    if (failedOperations.isEmpty() || ++retryCount > 3) {
        break; // 如果所有操作都成功了或者重试次数达到上线， 则不再需要重试。
    }

    // 如果有需要重试的操作， 则稍微等待一会后再次重试， 否则继续出错的概率很高。
    try {
        Thread.sleep(100); // 100ms后继续重试
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    request = failedOperations;
    result = client.batchGetRow(request);
} while (true);

System.out.println(String.format("查询成功%d行数据。", succeedRows.size()));
```

GetRange

getRange 将指定主键范围内的数据返回给应用程序。

下面的代码读取 GID 大于等于 1，小于 4 的所有行。

```
String endPoint = "<your endpoint>";
String accessId = "<your access id>";
```

```
String accessKey = "<your access key>";
String instanceName = "<your instance name>";
OTSClient client = new OTSClient(endPoint, accessId, accessKey, instanceName);

String COLUMN_UID_NAME = "uid";
String COLUMN_NAME_NAME = "name";
String COLUMN_ADDRESS_NAME = "address";
String COLUMN_GID_NAME = "groupid";
String COLUMN_AGE_NAME = "age";
String COLUMN_MOBILE_NAME = "mobile";

// 演示一下如何按主键范围查找，这里查找uid从1—4（左开右闭）的数据
RangeRowQueryCriteria criteria = new RangeRowQueryCriteria(tableName);
RowPrimaryKey inclusiveStartKey = new RowPrimaryKey();
inclusiveStartKey.addPrimaryKeyColumn(COLUMN_GID_NAME,
        PrimaryKeyValue.fromLong(1));
inclusiveStartKey.addPrimaryKeyColumn(COLUMN_UID_NAME,
        PrimaryKeyValue.INF_MIN); // 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

RowPrimaryKey exclusiveEndKey = new RowPrimaryKey();
exclusiveEndKey.addPrimaryKeyColumn(COLUMN_GID_NAME,
        PrimaryKeyValue.fromLong(4));
exclusiveEndKey.addPrimaryKeyColumn(COLUMN_UID_NAME,
        PrimaryKeyValue.INF_MAX); // 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

criteria.setInclusiveStartPrimaryKey(inclusiveStartKey);
criteria.setExclusiveEndPrimaryKey(exclusiveEndKey);
GetRangeRequest request = new GetRangeRequest();
request.setRangeRowQueryCriteria(criteria);
GetRangeResult result = client.getRange(request);
List<Row> rows = result.getRows();
for (Row row : rows) {
    System.out.println("name信息为："
            + row.getColumns().get(COLUMN_NAME_NAME));
    System.out.println("address信息为："
            + row.getColumns().get(COLUMN_ADDRESS_NAME));
    System.out.println("mobile信息为："
            + row.getColumns().get(COLUMN_MOBILE_NAME));
    System.out
            .println("age信息为：" + row.getColumns().get(COLUMN_AGE_NAME));
```

```
}

int consumedReadCU = result.getConsumedCapacity().getCapacityUnit()
        .getReadCapacityUnit();
System.out.println("本次读操作消耗的读CapacityUnit为：" + consumedReadCU);
```

使用 OTS Python SDK 进行数据操作

使用 OTS Python SDK 进行数据操作一般遵循以下几个步骤:

1. 创建 OTSClient 对象，在构造函数中指定 EndPoint，AccessKeyID，AccessKeySecret，实例名字。
2. 构造访问所需的对象。
3. 使用 OTSClient 对象相关接口发送请求。

PutRow

应用程序可以通过 put_row 接口插入或覆盖一行。应用程序需要指定该行的主键、属性 (可以为空)，行存在性检查条件。

下面的代码向表中添加了一行，该表的主键包含两个 Integer 类型的主键列。被添加的行拥有 4 个属性列。行存在性检查条件为 EXPECT_NOT_EXIST，即只有该行不存在时才插入数据.

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

primary_key = {'gid':1, 'uid':101}
attribute_columns = {'name':'张三', 'mobile':111111111, 'address':'中国A地', 'age':20}
condition = Condition('EXPECT_NOT_EXIST')
consumed = ots_client.put_row('myTable', condition, primary_key, attribute_columns)
print u'成功插入数据，消耗的写CapacityUnit为：%s' % consumed.write
```

UpdateRow

应用程序可以通过 update_row 接口更新一行。OTS 会根据请求的内容在这一行中新增列，修改或者删除指定列的值。应用程序需要指定该行的主键、要更新的属性列，行存在性检查条件。

下面的例子更新了一行，插入或修改属性中 name，address，删除 mobile，age。行存在性检查条件为 EXPECT_EXIST，即在该行存在的条件下才更新该行，否则更新失败。

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

primary_key = {'gid':1, 'uid':101}
update_of_attribute_columns = {
    'put' : {'name':'张三', 'address':'中国B地'},
    'delete' : ['mobile', 'age'],
}
condition = Condition('EXPECT_EXIST')
consumed = ots_client.update_row('myTable', condition, primary_key,
    update_of_attribute_columns)
print u'成功更新数据，消耗的写CapacityUnit为：%s' % consumed.write
```

DeleteRow

应用程序可以通过 delete_row 接口删除一行。应用程序需要指定该行的主键和行存在性检查条件。

下面的例子删除了一行。行存在性检查条件为 IGNORE，即不进行任何行存在性检查。

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

primary_key = {'gid':1, 'uid':101}
condition = Condition('IGNORE')
consumed = ots_client.delete_row('myTable', condition, primary_key)
print u'成功删除数据，消耗的写CapacityUnit为：%s' % consumed.write
```

GetRow

应用程序可以通过 get_row 接口读取一行。应用程序需要指定该行的主键和读取的列名。

下面的代码读取表中的一行中的 name，address，age 列。

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

primary_key = {'gid':1, 'uid':101}
columns_to_get = ['name', 'address', 'age']
consumed, primary_key_columns, attribute_columns = ots_client.get_row('myTable',
    primary_key, columns_to_get)
print u'成功读取数据，消耗的读CapacityUnit为：%s' % consumed.read
print u'name信息：%s' % attribute_columns.get('name')
print u'address信息：%s' % attribute_columns.get('address')
print u'age信息：%s' % attribute_columns.get('age')
```

BatchWriteRow

应用程序可以通过 batch_write_row 接口插入、更新、删除一张表或多张表中的多行。OTS 会将各个子操作的执行结果分别返回给应用程序，可能存在部分请求成功、部分请求失败的现象。即使整个请求没有返回错误，应用程序也必须要检查每个子操作返回的结果，从而拿到正确的状态。

下面的例子对 myTable 和 notExistTable 中的三行分别进行了 put_row，update_row，delete_row 操作。写入 notExistTable 的子操作会返回表不存在的错误信息。

```
ENDPOINT = "<your endpoint>";
ACCESSID = "<your access id>";
ACCESSKEY = "<your access key>";
INSTANCENAME = "<your instance name>";

ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)

primary_key = {'gid':2, 'uid':202}
attribute_columns = {'name':'李四', 'address':'中国某地', 'age':20}
condition = Condition('EXPECT_NOT_EXIST')
put_row_item = PutRowItem(condition, primary_key, attribute_columns)

primary_key = {'gid':3, 'uid':303}
condition = Condition('IGNORE')
update_of_attribute_columns = {
    'put' : {'name':'张三', 'address':'中国某地'},
```

```
    'delete' : ['mobile', 'age'],
}
update_row_item = UpdateRowItem(condition, primary_key, update_of_attribute_columns)

primary_key = {'gid':4, 'uid':404}
condition = Condition('IGNORE')
delete_row_item = DeleteRowItem(condition, primary_key)

table_item1  = {'table_name':'myTable', 'put':[put_row_item], 'update':[update_row_item],
    'delete':[delete_row_item]}
table_item2  = {'table_name':'notExistTable', 'put':[put_row_item], 'update':[
    update_row_item], 'delete':[delete_row_item]}
batch_list = [table_item1, table_item2]
batch_write_response = ots_client.batch_write_row(batch_list)

# 每一行操作都是独立的，需要分别判断是否成功。对于失败子操作进行重试。
retry_count = 0
operation_list = ['put', 'update', 'delete']
while retry_count < 3:
    failed_batch_list = []
    for i in range(len(batch_write_response)):
        table_item = batch_write_response[i]
        for operation in operation_list:
            operation_item = table_item.get(operation)
            if not operation_item:
                continue
            print u'操作：%s' % operation
            for j in range(len(operation_item)):
                row_item = operation_item[j]
                print u'操作是否成功：%s' % row_item.is_ok
                if not row_item.is_ok:
                    print u'错误码：%s' % row_item.error_code
                    print u'错误信息：%s' % row_item.error_message
                    add_batch_write_item(failed_batch_list, batch_list[i]['table_name'],
                        operation, batch_list[i][operation][j])
                else:
                    print u'本次操作消耗的写CapacityUnit为：%s' % row_item.consumed.write

    if not failed_batch_list:
        break
    retry_count += 1
    batch_list = failed_batch_list
```

```
56      batch_write_response = ots_client.batch_write_row(batch_list)
```

BatchGetRow

应用程序可以通过 batch_get_row 接口读取一个表或者多个表中的多行记录。batch_get_row 各个子操作独立执行，OTS 会将各个子操作的执行结果分别返回给应用程序，可能存在部分请求成功、部分请求失败的现象。即使整个请求没有返回错误，应用程序也必须要检查每个子操作返回的结果，从而拿到正确的状态。

下面的例子从 myTable 和 notExistTable 中读取了三行数据，读取 notExistTable 的子操作会返回表不存在的错误信息。

```
1   ENDPOINT = "<your endpoint>";
2   ACCESSID = "<your access id>";
3   ACCESSKEY = "<your access key>";
4   INSTANCENAME = "<your instance name>";
5
6   ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)
7
8   row1_primary_key = {'gid':1, 'uid':101}
9   row2_primary_key = {'gid':2, 'uid':202}
10  row3_primary_key = {'gid':3, 'uid':303}
11  columns_to_get = ['name', 'address', 'mobile', 'age']
12  batch_list = [('myTable', [row1_primary_key, row2_primary_key, row3_primary_key],
        columns_to_get)]
13  batch_list = [('notExistTable', [row1_primary_key, row2_primary_key, row3_primary_key],
        columns_to_get)]
14  batch_get_response = ots_client.batch_get_row(batch_list)
```

GetRange

应用程序可以通过 get_range 接口将指定主键范围内的数据返回给应用程序。

下面的代码读取 GID 大于等于 1，小于 4 的所有行。

```
1   ENDPOINT = "<your endpoint>";
2   ACCESSID = "<your access id>";
3   ACCESSKEY = "<your access key>";
4   INSTANCENAME = "<your instance name>";
5
6   ots_client = OTSClient(ENDPOINT, ACCESSID, ACCESSKEY, INSTANCENAME)
7
8   # 查询区间：[(1, INF_MIN), (4, INF_MAX))，左闭右开。
9   inclusive_start_primary_key = {'gid':1, 'uid':INF_MIN}
```

```
exclusive_end_primary_key = {'gid':4, 'uid':INF_MAX}
columns_to_get = ['name', 'address', 'mobile', 'age']
consumed, next_start_primary_key, row_list = ots_client.get_range(
            'myTable', 'FORWARD',
            inclusive_start_primary_key, exclusive_end_primary_key,
            columns_to_get, 100
)
for row in row_list:
    attribute_columns = row[1]
    print u'name信息为：%s' % attribute_columns.get('name')
    print u'address信息为：%s' % attribute_columns.get('address')
    print u'mobile信息为：%s' % attribute_columns.get('mobile')
    print u'age信息为：%s' % attribute_columns.get('age')
print u'本次操作消耗的读CapacityUnit为：%s' % consumed.read
print u'下次开始的主键：%s' % next_start_primary_key


def add_batch_write_item(batch_list, table_name, operation, item):
    for table_item in batch_list:
        if table_item.get('table_name') == table_name:
            operation_item = table_item.get(operation)
            if not operation_item:
                table_item[operation] = [item]
            else:
                operation_item.append(item)
            return
    # not found
    table_item = {'table_name':table_name, operation:[item]}
    batch_list.append(table_item)
```

第 4 章　访问控制

OTS 通过 AccessKey 和实例来保证 OTS 的访问安全性。

AccessKey

OTS 根据 AccessKey 对请求进行身份认证和鉴权，每个合法的 OTS 请求都必须携带正确的 AccessKey 信息。每个阿里云账户可以创建至多 5 个 AccessKey。AccessKey 由 AccessKeyID 和 AccessKeySecret 组成。AccessKeyID 用于标识 AccessKey，AccessKeySecret 用于加密 OTS 请求。

AccessKeySecret 是认证请求身份的重要凭证，因此需要保证 AccessKeySecret 的保密和安全。阿里云账户下的 AccessKey 可以被用于访问该阿里云账户中的所有实例。

AccessKey 有启用和禁用两种状态。只有处于启用状态的 AccessKey 可以被用来访问 OTS。在阿里云用户中心可以切换 AccessKey 的启用/禁用状态。AccessKey 切换状态后等待 1 分钟生效。

第 5 章　计量计费

OTS 从三个维度计量应用程序使用的资源并收取相应的费用：数据存储、预留读写吞吐量及外网下行流量。

数据存储

OTS 对实例的数据总量按小时计费。由于用户的数据总量会动态变化，因此 OTS 以固定的时间间隔统计表的数据总量大小，计算每小时数据总量的平均值，将平均值乘以单价进行计费。单价可能发生变化，请参照阿里云官网信息。

实例中所有表的数据大小之和是该实例的数据总量，表的数据大小是表中的所有行数据大小之和，下面举例说明表的数据大小的计算。

假设存在如下表，id 是主键列，其他均为属性列:

id	name	length	comments
Integer(1)	String(10byte)	Integer	String(32Byte)
Integer(2)	String(20byte)	Integer	String(999Byte)
Integer(3)	String(43byte)	Integer	空

对于 id=1 的行，其数据大小:len('id')+len('name')+len('length')+len('comments')+8Byte+10Byte+8Byte+32Byte=78Byte

对于 id=2 的行，其数据大小:len('id')+len('name')+len('length')+len('comments')+8Byte+20Byte+8Byte+999Byte=1055Byte

对于 id=3 的行，其数据大小:len('id')+len('name')+len('length')+8Byte+43Byte+8Byte=71Byte

表的数据大小之和为 78+1055+71=1204Byte。假设一小时内表的数据大小之未发生变化，将会按 1204Byte 进行计费。

OTS 对单表数据大小没有限制，用户可以根据自己的实际需求使用，按需付费。

预留读写吞吐量

预留读写吞吐量是表的一项属性。系统会在后台根据表的预留读写吞吐量配置预留资源，保证您对该表的吞吐量需求。

创建表 (CreateTable) 时需要指定表的预留读写吞吐量。在表创建成功后，还可以使用 UpdateTable 操作更新表的预留读写吞吐量配置。

预留读写吞吐量。OTS 限制单表的预留读写吞吐量必须大于 0(读和写都必须大于 0)，不超过 5000(读和写分别不超过 5000)。如果用户有单表预留读写吞吐量需要超出 5000 的需求，可以通过人工服务提高预留读写吞吐量

预留读写吞吐量的计量单位为写服务能力单元和读服务能力单元，应用程序通过 API 进行 OTS 读写操作时均会消耗对应的写服务能力单元和读服务能力单元。

OTS 对实例中所有表的预留读写吞吐量之和按小时计费。用户配置的预留读写吞吐量可能会动态变化，OTS 以固定的时间间隔统计表的预留读写吞吐量，计算每个小时的预留读写吞吐量平均值，将平均值乘以单价进行计费。预留读写吞吐量单价可能发生变化，请参照阿里云官网信息。

外网下行流量

OTS 对应用访问 OTS 的外网下行流量进行收费。应用程序使用 HTTP 方式访问 OTS 返回的响应是下行流量的主要成分。即使操作失败，OTS 返回操作失败信息，也会产生下行流量。OTS 仅对外网下行流量收费，内网下行流量与上行流量均不收费。流量单价可能发生变化，请参照阿里云官网信息。

第 6 章　使用 OTS API

应用程序可以使用阿里云官方发布的 OTS SDK 来访问 OTS，也可以通过 POST 方法发送 HTTP 请求来访问 OTS。本章将介绍 HTTP 请求结构和数据格式，以及如何构造 HTTP 请求和解析 HTTP 请求的返回结果。最后介绍 OTS 请求返回的错误状态码。Java 和 Python 语言的开发者可使用官方的 Java 和 Python SDK。如果需要使用 Java 和 Python 以外的语言访问 OTS，可以根据本章内容使用 HTTP 消息与 OTS 进行交互，也可以自行编写 SDK。

HTTP 消息

OTS 接收应用程序的 HTTP 请求，执行相应的逻辑并以 HTTP 消息返回处理后的结果数据。HTTP 请求和响应中的数据通过 ProtocolBuffer 协议格式进行组织，关于 ProtocolBuffer 协议的更多内容，请参考ProtocolBuffer 介绍。下面分别介绍 HTTP 请求 header 和 body 以及响应的具体格式。

HTTP Request

HTTP Request URL

访问 OTS 的 URL 由以下方式构成:

```
外网访问URL:
http://<instance>.<region>.ots.aliyuncs.com/<operation>
内网访问URL:
http://<instance>.<region>.ots-internal.aliyuncs.com/<operation>
```

- instance

  实例名称。实例由用户创建，可以在 OTS 控制台查看云账户下拥有的 Instance 信息。大小写不敏感。

- region

  阿里云服务节点。OTS 服务会部署在多个地理位置不同的阿里云服务节点内。创建实例时需要制定阿里云节点，可以在 OTS 控制台查看实例所在的阿里云服务节点名称。大小写不敏感。

- operation

  OTS 操作名称，可以在API Reference一章查看所有的 OTS 操作名。大小写敏感。

如下所示的 URL 是杭州节点，实例名称为 myInstance 的 ListTable 请求的目标 URL。

```
http://myInstance.cn-hangzhou.ots.aliyuncs.com/ListTable
```

HTTP Request Header

OTS 规定 HTTP 请求的 Header 必须包含以下信息:

- x-ots-date

  请求发出时间，日期格式采用 rfc822 标准，并使用 UTC 时间，格式为"%a, %d %b %Y %H:%M:%S GMT"

- x-ots-apiversion

  API 的版本号，版本号是一个日期字符串，本文档对应的 API 版本号为 2014-08-08

- x-ots-accesskeyid

  用户的 AccessKeyID

- x-ots-instancename

  实例名称

- x-ots-contentmd5

  对 HTTP body 中计算出的 MD5，使用 base64 进行编码

- x-ots-signature

  请求的签名。签名计算方式如下:

```
Signature = base64(HmacSha1(AccessKeySecret, StringToSign));

StringToSign = CanonicalURI + '\n' + HTTPRequestMethod + '\n\n' + CanonicalHeaders

CanonicalHeaders = LowerCase (HeaderName1) + ':' + Trim(HeaderValue1) + '\n' + ... +
    LowerCase (HeaderNameN) + ':' + Trim(HeaderValueN) + '\n'
```

  - 伪代码中使用到的函数的说明:
    * HmacSha1 - Hmac-Sha1 加密算法。计算 OTS 请求签名时请将 StringToSign 作为消息，AccessKeySecret 作为密钥
    * base64 - base64 编码算法
    * LowerCase - 将字符串中的字母全部变成小写
    * Trim - 去除字符串首尾处的空白字符
  - CanonicalURI

    HTTP URL 中的路径部分，如"http://myInstance.cn-hangzhou.ots.aliyuncs.com/ListTable" 中，CanonicalURI 为"/ListTable"
  - HTTPRequestMethod

    HTTP 请求方法，如 GET、POST 或 PUT 等，OTS 的 HTTP API 只支持 POST 方法。注意 POST 需要大写
  - CanonicalHeaders

    CanonicalHeaders 是 OTS HTTP header 按照以下规则构造的字符串 (不包括 x-ots-signature 头):
    * 需要包含且只包含所有以'x-ots-' 开头的 OTS 标准头
    * header 项名称全部小写，值必须经过 trim 去除空格
    * header 项按照名字的字典序从小到大排序
    * header 项的名称和值之间以':' 相隔
    * 每个 header 之间以换行符相隔

OTS 会对 HTTP 请求进行验证:

- 验证 x-ots-contentmd5 头的值与 HTTP Body 中所含数据计算出的 MD5 是否一致
- 验证请求头中包含的签名是否正确
- 验证 x-ots-date 包含的时间与服务器时间相差小于 15 分钟

若认证未通过，OTS 会直接返回身份认证错误。

HTTP Request Body

OTS 规定 HTTP 请求的 Body 部分是 OTS 定义的 **ProtocolBuffer** 消息序列化之后的字符串，Body 长度不超过 2MB。

OTS 请求的 **ProtocolBuffer** 消息定义可以参看OTS ProtocolBuffer 消息定义。

HTTP Response

HTTP Response Header

OTS 规定 HTTP 响应的 Header 包含以下信息:

- x-ots-date

  请求发出时间，日期格式采用 rfc822 标准，并使用 UTC 时间，格式为"%a, %d %b %Y %H:%M:%S GMT"

- x-ots-requestid

  本次请求的请求 ID

- x-ots-contenttype

  响应的内容类型。固定为"protocol buffer" 的字符串

- x-ots-contentmd5

  根据响应内容计算出的 MD5 值，使用 Base64 编码

- Authorization

  响应的签名。只有请求的签名被 OTS 验证通过的条件下，响应才会包含签名。签名计算方式如下:

```
1 Authorization = 'OTS ' + AccessKeySecret + ':' + base64(HmacSha1(AccessKeySecret,
      stringToSign))
2
3 StringToSign = CanonicalHeaders + CanonicalURI
4
5 CanonicalHeaders = LowerCase (HeaderName1) + ':' + Trim(HeaderValue1) + '\n' + ... +
      LowerCase (HeaderNameN) + ':' + Trim(HeaderValueN) + '\n'
```

  - 伪代码中使用到的函数的说明:

    与上面请求中所用到的函数相同

  - CanonicalURI

    HTTP URL 中的路径部分，如"http://myInstance.cn-hangzhou.ots.aliyuncs.com/ListTable" 中，CanonicalURI 为"/ListTable"

– CanonicalHeaders

CanonicalHeaders 是 OTS HTTP header 按照以下规则构造的字符串 (不包括 x-ots-signature 头):

* 需要包含且只包含所有以'x-ots-' 开头的 OTS 标准头
* header 项名称全部小写，值必须经过 trim 去除空格
* header 项按照名字的字典序从小到大排序
* header 项的名称和值之间以':' 相隔
* 每个 header 之间以换行符相隔

客户端应该对 OTS 响应进行以下验证:

- 验证响应头中包含的签名是否正确
- 验证 x-ots-date 包含的时间与客户端时间是否相差 15 分钟（正负 15 分钟）
- 验证 x-ots-contentmd5 头的值与响应数据计算出的 MD5 是否一致

如验证不通过，用户应该在代码中拒绝这个响应所包含的数据，该响应有可能不是 OTS 服务返回的。

HTTP Response Content

OTS 规定 HTTP 响应的内容是 OTS 定义的 **ProtocolBuffer** 消息序列化之后的字符串，Body 长度不超过 2MB。每一个 OTS 请求消息对应一个 OTS 响应消息，应用程序将响应内容反序列化之后，读取 OTS 操作的结果。

签名示例

下面提供了两个请求和响应的签名验证示例，用户可以在实现签名算法后用下面的例子测试算法的实现是否正确。

请求签名示例

假设用户的 AccessKeyID 为:'29j2NtzlUr8hjP8b',AccessKeySecret 为: '8AKqXmNBkl85QK70cAOuH4bBd3gS0J'

```
POST /ListTable HTTP/1.0
x-ots-date: Tue, 12 Aug 2014 10:23:03 GMT
x-ots-apiversion:2014-08-08
x-ots-accesskeyid: 29j2NtzlUr8hjP8b
x-ots-contentmd5: 1B2M2Y8AsgTpgAmY7PhCfg==
x-ots-instancename: naketest
```

那么用户请求的签名结果如下

```
stringToSign = '/ListTable\nPOST\n\nx-ots-accesskeyid:29j2NtzlUr8hjP8b\nx-ots-apiversion
    :2014-08-08\nx-ots-contentmd5:1B2M2Y8AsgTpgAmY7PhCfg==\nx-ots-date:Tue, 12 Aug 2014
    10:23:03 GMT\nx-ots-instancename:naketest\n'

signature = base64(HmacSha1('8AKqXmNBkl85QK70cAOuH4bBd3gS0J', stringToSign))
          = '4xap392B7EBpN+RmlHgNowjoG1w='
```

响应签名示例

假设用户的 AccessKeyID 为:'29j2NtzlUr8hjP8b',AccessKeySecret 为: 'AKqXmNBkl85QK70cAOuH4bBd3gS0J'

```
/ListTable
x-ots-contentmd5: 1B2M2Y8AsgTpgAmY7PhCfg==
x-ots-requestid: 0005006c-0e81-db74-4a34-ce0a5df229a1
x-ots-contenttype: protocol buffer
x-ots-date:Tue, 12 Aug 2014 10:23:03 GMT
```

那么 OTS 响应的签名结果如下

```
stringToSign = 'x-ots-contentmd5:1B2M2Y8AsgTpgAmY7PhCfg==\nx-ots-contenttype:protocol
    buffer\nx-ots-date:Tue, 12 Aug 2014 10:23:03 GMT\nx-ots-requestid:0005006c-0e81-db74-4
    a34-ce0a5df229a1\n/ListTable'

authorization = 'OTS ' + AccessKeySecret +  ':' + base64(HmacSha1('8
    AKqXmNBkl85QK70cAOuH4bBd3gS0J', stringToSign))
              = 'OTS 29j2NtzlUr8hjP8b:Y24MHhVti5UhSCW5qsUSDvT9SOk='
```

OTS 错误信息

本节列举了 OTS API 所有可能错误的错误类型、描述信息与 HTTP 状态码

以下列表中列出了所有 OTS 可能返回的错误信息。其中部分错误有可能通过重试解决，此类错误在列表中“重试”列的值为‘是’，反之为‘否’。

权限验证错误

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
403	OTSAuthFailed	The AccessKeyID does not exist.	AccessKeyID 不存在	否
403	OTSAuthFailed	The AccessKeyID is disabled.	AccessKeyID 被禁用	否
403	OTSAuthFailed	The user does not exist.	该用户不存在	否

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
403	OTSAuthFailed	The instance is not found.	该实例不存在	否
403	OTSAuthFailed	The user has no privilege to access the instance.	没有访问该实例的权限	否
403	OTSAuthFailed	The instance is not running.	该实例的状态不是运行中 (Running)	否
403	OTSAuthFailed	The user has no privilege to access the instance.	没有访问该实例的权限	否
403	OTSAuthFailed	Signature mismatch.	签名不匹配	否
403	OTSAuthFailed	Mismatch between system time and x-ots-date: { Date }.	服务器时间与请求 header 中 x-ots-date 的时间相差超过一定范围	否

HTTP 消息错误

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
413	OTSRequestBodyTooLarge	The size of POST data is too large.	用户 Post 请求发送的数据过大	否
408	OTSRequestTimeout	Request timeout.	客户端完成请求的时间过长	否
405	OTSMethodNotAllowed	OTSMethodNotAllowedOnly POST method for requests is supported.	仅支持 POST 方式的请求	否
403	OTSAuthFailed	Mismatch between MD5 value of request body and x-ots-contentmd5 in header.	根据请求 Body 数据计算的 MD5 与请求 header 中包含的 x-ots-contentmd5 值不同	否
400	OTSParameterInvalid	Missing header: '{HeaderName}'.		请求中缺少必要的 header
400	OTSParameterInvalid	Invalid date format: {Date}.	时间格式不合法	否
400	OTSParameterInvalid	Unsupported operation: {Operation}.	请求的 URL 中的操作名不合法	否

API 错误

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
500	OTSInternalServerError	Internal server error.	内部错误	是
403	OTSQuotaExhausted	Too frequent table operations.	执行 CreateTable/ ListTable/DescribeTable/ DeleteTable 这些表相关的操作过于频繁	否
403	OTSQuotaExhausted	Number of tables exceeded the quota.	表的数量超过定额	否
400	OTSParameterInvalid	The name of primary key must be unique.	建表的主键列名不唯一	否
400	OTSParameterInvalid	Failed to parse the ProtoBuf message.	请求 Body 中的 PB 数据反序列化失败	否
400	OTSParameterInvalid	Both read and write capacity unit are required to create table.	建表时必须指定预留读写吞吐量	否
400	OTSParameterInvalid	Neither read nor write capacity unit is set.	更新表时，需要设置读或写预留能力值	否
400	OTSParameterInvalid	Invalid instance name: '{InstanceName}'.	实例名称不合法	否
400	OTSParameterInvalid	Invalid table name: '{TableName}'.	表名不合法	否
400	OTSParameterInvalid	The value of read capacity unit must be in range: [{LowerLimit}, {UpperLimit}]	预留读能力值必须在指定范围内	否
400	OTSParameterInvalid	The value of write capacity unit must be in range: [{LowerLimit}, {UpperLimit}]	预留写能力值必须在指定范围内	否
400	OTSParameterInvalid	Invalid column name: '{ColumnName}'.	列名名称不合法	否
400	OTSParameterInvalid	{ColumnType} is an invalid type for the primary key.	主键列类型不合法	否
400	OTSParameterInvalid	{ColumnType} is an invalid type for the primary key in GetRange.	在 GetRange 请求中，主键列类型不合法	否
400	OTSParameterInvalid	{ColumnType} is an invalid type for the attribute column.	属性列类型不合法	否
400	OTSParameterInvalid	The number of primary key columns must be in range: [1, {Limit}].	主键列的列数不能为 0，不能超出限制	否
400	OTSParameterInvalid	Value of column '{ColumnName}' must be UTF8 encoding.	此列列值必须为 UTF8 编码。	否
400	OTSParameterInvalid	The length of attribute column: '{ColumnName}' exceeded the MaxLength:{MaxSize} with CurrentLength:{CellSize}.	属性列名长度超出命名最大长度限制	否

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
400	OTSParameterInvalid	No row specified in the request of BatchGetRow.	BatchGetRow 请求中未指定任何行	否
400	OTSParameterInvalid	Duplicated table name: '{TableName}'.	在 BatchGetRow 或 BatchWriteRow 操作中包含同名的表	否
400	OTSParameterInvalid	No row specified in table: '{TableName}'.	在 BatchGetRow 操作中某个表上未指定任何行	否
400	OTSParameterInvalid	Duplicated primary key:'{PKName}' of getting row #{RowIndex} in table '{TableName}'.	在 BatchGetRow 操作中某个表的某行包含同名的主键列	否
400	OTSParameterInvalid	Duplicated column name with primary key column: '{PKName}' while putting row #{RowIndex} in table: '{TableName}'.	在 BatchWriteRow 操作中某个表的执行 PutRow 操作的某行包含与主键列同名的属性列	否
400	OTSParameterInvalid	Duplicated column name with primary key column: '{PKName}' while updating row #{RowIndex} in table: '{TableName}'.	在 BatchWriteRow 操作中某个表的执行 UpdateRow 操作的某行包含与主键列同名的属性列	否
400	OTSParameterInvalid	"Duplicated column name: '{ColumnName}' while putting row #{Index} in table: '{TableName}'."	在 BatchWriteRow 操作中某个表的执行 PutRow 操作的某行包含重复的属性列	否
400	OTSParameterInvalid	"Duplicated column name: '{ColumnName}' while updating row #{Index} in table: 'TableName'."	在 BatchWriteRow 操作中某个表的执行 UpdateRow 操作的某行包含重复的属性列	否
400	OTSParameterInvalid	No attribute column specified to update row #{RowIndex} in table '{TableName}'.	在 BatchWriteRow 操作中更新某个表的某行时未指定属性列	否
400	OTSParameterInvalid	Invalid condition: {RowExistence} while updating row #{RowIndex} in table : '{TableName}'.	在 BatchWriteRow 操作中更新某个表的某行时，RowExistence 条件不合法	否
400	OTSParameterInvalid	Duplicated primary key name: '{PKName}'.	主键重复	否
400	OTSParameterInvalid	Invalid condition: {RowExistence} while deleting row.	删除行操作时，RowExistence 条件不合法	否
400	OTSParameterInvalid	The limit must be greater than 0.	limit 参数必须大于 0	否

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
400	OTSParameterInvalid	Duplicated attribute column name with primary key column: '{ColumnName}' while putting row.	对某行进行写入操作时，包含与主键列同名的属性列	否
400	OTSParameterInvalid	"Duplicated column name: '{ColumnName}' while putting row."	对某行进行写入操作时，包含与重复的属性列	否
400	OTSParameterInvalid	Duplicated attribute column name with primary key column: '{ColumnName}' while updating row.	对某行进行更新操作时，包含与主键列同名的属性列	否
400	OTSParameterInvalid	No column specified while updating row.	更新行操作时，没有指定需要更新的列	否
400	OTSParameterInvalid	Duplicated column name: '{ColumnName}' while updating row.	更新行操作时，包含与重复的属性列	否
400	OTSParameterInvalid	Invalid condition: {RowExistence} while updating row.	更新行操作时，RowExistence 条件不合法	否
400	OTSParameterInvalid	Optional field 'v_string' must be set as ColumnType is STRING.	赋值时，传入的列值(可选参数) 必须与参数定义的该列数据类型保持一致均为 String	否
400	OTSParameterInvalid	Optional field 'v_int' must be set as ColumnType is INTEGER.	赋值时，传入的列值(可选参数) 必须与参数定义的该列数据类型保持一致均为 Integer	否
400	OTSParameterInvalid	Optional field 'v_bool' must be set as ColumnType is BOOLEAN.	赋值时，传入的列值(可选参数) 必须与参数定义的该列数据类型保持一致均为 Boolean	否
400	OTSParameterInvalid	Optional field 'v_double' must be set as ColumnType is DOUBLE.	赋值时，传入的列值(可选参数) 必须与参数定义的该列数据类型保持一致均为 Double	否
400	OTSParameterInvalid	Optional field 'v_binary' must be set as ColumnType is BINARY.	赋值时，传入的列值(可选参数) 必须与参数定义的该列数据类型保持一致均为 Binary	否

OTS 存储相关异常

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
503	OTSServerBusy	Server is busy.	OTS 内部服务器繁忙	否
503	OTSPartitionUnavailable	The partition is not available.	内部服务器异常，导致表的部分分区不可服务	否
503	OTSTimeout	Operation timeout.	在 OTS 内部操作超时	否
503	OTSServerUnavailable	Server is not available.	在 OTS 内部有服务器不可访问	否
409	OTSRowOperationConflict	Data is being modified by the other request.	多个并发的请求写同一行数据，导致冲突	是
409	OTSObjectAlreadyExist	Requested table already exists.	请求创建的表已经存在	否
404	OTSObjectNotExist	Requested table does not exist.	请求的表不存在	否
404	OTSTableNotReady	The table is not ready.	表刚被创建还无法立即提供服务	是
403	OTSTooFrequentReservedThroughputAdjustment	Capacity unit adjustment is too frequent.	读写能力调整过于频繁	否
403	OTSNotEnoughCapacityUnit	Remaining capacity unit is not enough.	剩余预留读写能力不足	否
403	OTSConditionCheckFail	Condition check failed.	预查条件检查失败	否
400	OTSOutOfRowSizeLimit	The total data size of columns in one row exceeded the limit.	该行所有列数据大小总和超出限制	否
400	OTSOutOfColumnCountLimit	The number of columns in one row exceeded the limit.	该行总列数超出限制	否
400	OTSInvalidPK	Primary key schema mismatch.	主键不匹配	否

第 7 章　API Reference

Actions

GetRow

行为：

根据给定的主键读取单行数据。

请求结构：

```
message GetRowRequest {
    required string table_name = 1;
    repeated Column primary_key = 2;
```

```
    repeated string columns_to_get = 3;
}
```

table_name:

类型: string

是否必要参数: 是

要读取的数据所在的表名。

primary_key:

类型: repeated Column

是否必要参数: 是

该行全部的主键列。

columns_to_get:

类型: repeated string

是否必要参数: 否

需要返回的全部列的列名。若为空，则返回该行的所有列。

如果指定的列不存在，则不会返回该列的数据。

如果给出了重复的列名，返回结果只会包含一次该列。

columns_to_get 中 string 的个数不应超过 128 个。

响应消息结构：

```
message GetRowResponse {
    required ConsumedCapacity consumed = 1;
    required Row row = 2;
}
```

consumed:

类型: ConsumedCapacity

本次操作消耗的服务能力单元。

row:

类型: Row

该行需要返回的的列数据集合。其中 primary_key_columns 和 attribute_columns 分别存放读取到的主键列和属性列，其顺序不保证与 GetRowRequest 中的 columns_to_get 一致。如果该行不存在，primary_key_columns 和 attribute_columns 均为空。

服务能力单元消耗：

如果请求的行不存在，消耗 1 读服务能力单元。

如果请求的行存在，消耗读服务能力单元的数值为这该行全部数据大小除以 1KB 向上取整。关于数据大小的计算请参见相关章节。

如果请求超时，结果未定义，服务能力单元有可能被消耗，也可能未被消耗。

如果返回内部错误（HTTP 状态码:5XX），则此次操作不消耗服务能力单元，其他错误情况消耗 1 读服务能力单元。

请求示例：

```
GetRowRequest {
  table_name: "consume_history"
  primary_key {
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  primary_key {
    name: "SellerID"
    value {
      type: STRING
      v_string: "00022"
    }
  }
  primary_key {
    name: "DeviceID"
    value {
      type: STRING
      v_string: "061104"
    }
  }
  primary_key {
    name: "OrderNumber"
    value {
```

```
      type: INTEGER
      v_int: 142857
    }
  }
  columns_to_get: "CardID"
  columns_to_get: "SellerID"
  columns_to_get: "DeviceID"
  columns_to_get: "OrderNumber"
  columns_to_get: "Amount"
  columns_to_get: "Remarks"
}
```

响应示例：

```
GetRowResponse {
  consumed {
    capacity_unit {
      read: 1
    }
  }
  row {
    primary_key_columns {
      name: "CardID"
      value {
        type: STRING
        v_string: "2007035023"
      }
    }
    primary_key_columns {
      name: "SellerID"
      value {
        type: STRING
        v_string: "00022"
      }
    }
    primary_key_columns {
      name: "DeviceID"
      value {
        type: STRING
        v_string: "061104"
      }
    }
```

```
    primary_key_columns {
      name: "OrderNumber"
      value {
        type: INTEGER
        v_int: 142857
      }
    }
    attribute_columns {
      name: "Amount"
      value {
        type: DOUBLE
        v_double: 2.5
      }
    }
    attribute_columns {
      name: "Remarks"
      value {
        type: STRING
        v_string: "ice cream"
      }
    }
  }
}
```

PutRow

行为:

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。

请求消息结构:

```
message PutRowRequest {
    required string table_name = 1;
    required Condition condition = 2;
    repeated Column primary_key = 3;
    repeated Column attribute_columns = 4;
}
```

table_name:

类型: string

是否必要参数: 是

请求写入数据的表名。

condition:

类型: Condition

是否必要参数: 是

在数据写入前是否进行行存在性检查，可以取下面三个值:

- IGNORE 表示不做行存在性检查。
- EXPECT_EXIST 表示期望行存在。
- EXPECT_NOT_EXIST 表示期望行不存在。

若期待该行不存在但该行已存在，则会插入失败, 返回错误；反之亦然。

primary_key:

类型: repeated Column

是否必要参数: 是

请求写入的行全部的主键列。

attribute_columns:

类型: repeated Column

是否必要参数: 否

请求写入的行的属性，如果 attribute_columns 为空，则表示写入的行没有任何属性列。

attribute_columns 中 Column 的个数不能超过 128 个。

响应消息结构:

```
message PutRowResponse {
    required ConsumedCapacity consumed = 1;
}
```

consumed:

类型 ConsumedCapacity

本次操作消耗的服务能力单元。

服务能力单元消耗：

如果该行不存在，消耗写服务能力单元的数值为要插入的全部数据大小除以 1KB 向上取整。关于数据大小的计算请参见相关章节。

如果该行存在，消耗写服务能力单元的数值为该行原有全部数据大小除以 1KB 向上取整 + 要插入的全部数据大小除以 1KB 向上取整。

如果返回内部错误（HTTP 状态码:5XX），则此次操作不消耗服务能力单元；其他错误情况消耗 1 写服务能力单元。

如果请求超时，结果未定义，服务能力单元有可能被消耗，也可能未被消耗。

请求示例：

```
PutRowRequest {
  table_name: "consume_history"
  condition {
    row_existence: EXPECT_NOT_EXIST
  }
  primary_key {
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  primary_key {
    name: "SellerID"
    value {
      type: STRING
      v_string: "00022"
    }
  }
  primary_key {
    name: "DeviceID"
    value {
      type: STRING
      v_string: "061104"
    }
  }
  primary_key {
    name: "OrderNumber"
    value {
```

```
      type: INTEGER
      v_int: 142857
    }
  }
  attribute_columns {
    name: "Amount"
    value {
      type: DOUBLE
      v_double: 2.5
    }
  }
  attribute_columns {
    name: "Remarks"
    value {
      type: STRING
      v_string: "ice cream"
    }
  }
}
```

响应示例：

```
PutRowResponse {
  consumed {
    capacity_unit {
      write: 1
    }
  }
}
```

UpdateRow

行为：

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。

请求消息结构：

```
message UpdateRowRequest {
    required string table_name = 1;
```

```
    required Condition condition = 2;
    repeated Column primary_key = 3;
    repeated ColumnUpdate attribute_columns = 4;
}
```

table_name:

类型: string

是否必要参数: 是

请求更新数据的表名。

condition:

类型: Condition

是否必要参数: 是

在数据更新前是否进行存在性检查，可以取下面两个值:

- IGNORE 表示不做行存在性检查。
- EXPECT_EXIST 表示期望行存在。

若期待该行存在但该行不存在，则本次更新操作会失败, 返回错误；若忽视该行是否存在，则无论该行是否存在，都不会因此导致本次操作失败。

primary_key:

类型: repeated Column

是否必要参数: 是

请求更新的行全部的主键列。

attribute_columns:

类型: repeated ColumnUpdate

是否必要参数: 是

该行本次想要更新的全部 **AttributColumn**，OTS 会根据 attribute_columns 中每个 ColumnUpdate 的内容在这一行中新增、修改或者删除指定列的值。

该行已经存在的不在 attribute_columns 列表中的列将不受影响。

attribute_columns 中应至少含有一个 ColumnUpdate 对象，否则请求失败，返回错误。

若 attribute_columns 中出现了相同列名的列，则请求失败，返回错误。

attribute_columns 中 ColumnUpdate 的个数不能超过 128 个；更新完成后，每行的属性列亦不能超过 128 个。

ColumnUpdate 的 type 可以取下面两个值:

- PUT，此时该 ColumnUpdate 的 value 必须为有效的属性列值。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。
- DELETE，此时该 ColumnUpdate 的 value 必须为空。语意为删除该列。

如果该行不存在，且 attribute_columns 中只含有 type 为 DELETE 的 ColumnUpdate，则 UpdateRow 操作完成后该行仍然不存在。

注意: 删除本行的全部 **AttributColumn** 不等同于删除本行，若想删除本行，请使用 DeleteRow 操作。

响应消息结构:

```
message UpdateRowResponse {
    required ConsumedCapacity consumed = 1;
}
```

consumed:

类型 ConsumedCapacity

本次操作消耗的服务能力单元。

服务能力单元消耗:

如果该行不存在，消耗写服务能力单元的数值为要更新的全部数据大小除以 1KB 向上取整。关于数据大小的计算请参见相关章节。

如果该行存在，消耗写服务能力单元的数值为该行原有全部数据大小除以 1KB 向上取整和更新完成后该行的全部数据大小除以 1KB 向上取整两者中的较大值。

如果请求超时，结果未定义，服务能力单元有可能被消耗，也可能未被消耗。

如果返回内部错误（HTTP 状态码:5XX），则此次操作不消耗服务能力单元；其他错误情况消耗 1 写服务能力单元。

请求示例:

```
UpdateRowRequest {
```

```
  table_name: "consume_history"
  condition {
    row_existence: EXPECT_EXIST
  }
  primary_key {
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  primary_key {
    name: "SellerID"
    value {
      type: STRING
      v_string: "00022"
    }
  }
  primary_key {
    name: "DeviceID"
    value {
      type: STRING
      v_string: "061104"
    }
  }
  primary_key {
    name: "OrderNumber"
    value {
      type: INTEGER
      v_int: 142857
    }
  }
  attribute_columns {
    type: PUT
    name: "Amount"
    value {
      type: DOUBLE
      v_double: 3.5
    }
  }
  attribute_columns {
    type: DELETE
```

```
    name: "Remarks"
  }
}
```

响应示例:

```
UpdateRowResponse {
  consumed {
    capacity_unit {
      write: 1
    }
  }
}
```

DeleteRow

行为:

删除一行数据。

请求消息结构:

```
message DeleteRowRequest {
    required string table_name = 1;
    required Condition condition = 2;
    repeated Column primary_key = 3;
}
```

table_name:

类型: string

是否必要参数: 是

请求删除数据的表名。

condition:

类型: Condition

是否必要参数: 是

在删除本行前是否进行行存在性检查，可以取下面两个值:

- IGNORE 表示不做行存在性检查。
- EXPECT_EXIST 表示期望行存在。

若期待该行存在但该行不存在，则本次删除操作会失败, 返回错误；若忽视该行是否存在，则无论该行实际是否存在，都不会因此导致操作失败。

primary_key:

类型: repeated Column

是否必要参数: 是

请求删除的行全部的主键列。

响应消息结构:

```
message DeleteRowResponse {
    required ConsumedCapacity consumed = 1;
}
```

consumed:

类型 ConsumedCapacity

本次操作消耗的服务能力单元。

服务能力单元消耗:

如果该行不存在，消耗 1 写服务能力单元。

如果该行存在，消耗写服务能力单元的数值为该行原有全部数据大小除以 1KB 向上取整。关于数据大小的计算请参见相关章节。

如果返回内部错误（HTTP 状态码:5XX），则此次操作不消耗服务能力单元；其他错误情况消耗 1 写服务能力单元。

如果请求超时，结果未定义，服务能力单元有可能被消耗，也可能未被消耗。

请求示例:

```
DeleteRowRequest {
  table_name: "consume_history"
  condition {
    row_existence: IGNORE
  }
```

```
  primary_key {
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  primary_key {
    name: "SellerID"
    value {
      type: STRING
      v_string: "00022"
    }
  }
  primary_key {
    name: "DeviceID"
    value {
      type: STRING
      v_string: "061104"
    }
  }
  primary_key {
    name: "OrderNumber"
    value {
      type: INTEGER
      v_int: 142857
    }
  }
}
```

响应示例：

```
DeleteRowResponse {
  consumed {
    capacity_unit {
      write: 1
    }
  }
}
```

GetRange

行为：

读取指定主键范围内的数据。

请求结构：

```
message GetRangeRequest {
    required string table_name = 1;
    required Direction direction = 2;
    repeated string columns_to_get = 3;
    optional int32 limit = 4;
    repeated Column inclusive_start_primary_key = 5;
    repeated Column exclusive_end_primary_key = 6;
}
```

table_name:

类型: string

是否必要参数: 是

要读取的数据所在的表名。

direction:

类型: Direction

是否必要参数: 是

本次查询的顺序，若为正序，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，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。

columns_to_get:

类型: repeated string

是否必要参数: 否

需要返回的全部列的列名。若为空，则返回读取结果中每行的所有列。

如果给出了重复的列名，返回结果只会包含一次该列。

columns_to_get 中 string 的个数不应超过 128 个。

limit:

类型: int32

是否必要参数: 否

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，OTS 最多返回行数为 5000 且总数据大小不超过 1M。

inclusive_start_primary_key:

类型: repeated Column

是否必要参数: 是

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。

exclusive_end_primary_key:

类型: repeated Column

是否必要参数: 是

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。

在 GetRange 中，inclusive_start_primary_key 和 exclusive_end_primary_key 中的 Column 的 type 可以使用本操作专用的两个类型 INF_MIN 和 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。

响应消息结构:

```
message GetRangeResponse {
    required ConsumedCapacity consumed = 1;
    repeated Column next_start_primary_key = 2;
    repeated Row rows = 3;
}
```

consumed:

类型: ConsumedCapacity

本次操作消耗的服务能力单元。

next_start_primary_key:

类型: repeated Column

本次 GetRange 操作的断点信息。

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。

注意:OTS 系统中限制了 GetRange 操作的响应消息中数据不超过 5000 行，大小不超过 1M。即使在 GetRange 请求中未设定 limit，在响应中仍可能出现 next_start_primary_key。因此在使用 GetRange 时一定要对响应中是否有 next_start_primary_key 进行处理。

rows:

类型: repeated Row

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 primary_key_columns 和 attribute_columns 均只包含在 columns_to_get 中指定的列，其顺序不保证与 request 中的 columns_to_get 一致；primary_key_columns 的顺序亦不保证与建表时指定的顺序一致。

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。

服务能力单元消耗:

GetRange 操作消耗读服务能力单元的数值为查询范围内所有行数据大小除以 1KB 向上取整。关于数据大小的计算请参见相关章节。

如果请求超时，结果未定义，服务能力单元有可能被消耗，也可能未被消耗。

如果返回内部错误（HTTP 状态码:5XX），则此次操作不消耗服务能力单元，其他错误情况消耗 1 读服务能力单元。

请求示例:

```
GetRangeRequest {
  table_name: "consume_history"
  direction: FORWARD
  columns_to_get: "CardID"
  columns_to_get: "SellerID"
  columns_to_get: "DeviceID"
  columns_to_get: "OrderNumber"
  columns_to_get: "Amount"
  columns_to_get: "Remarks"
  limit: 2
  inclusive_start_primary_key {
```

```
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  inclusive_start_primary_key {
    name: "SellerID"
    value {
      type: INF_MIN
    }
  }
  inclusive_start_primary_key {
    name: "DeviceID"
    value {
      type: INF_MIN
    }
  }
  inclusive_start_primary_key {
    name: "OrderNumber"
    value {
      type: INF_MIN
    }
  }
  exclusive_end_primary_key {
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  exclusive_end_primary_key {
    name: "SellerID"
    value {
      type: INF_MIN
    }
  }
  exclusive_end_primary_key {
    name: "DeviceID"
    value {
      type: INF_MIN
    }
```

```
  }
  exclusive_end_primary_key {
    name: "OrderNumber"
    value {
      type: INF_MIN
    }
  }
}
```

响应示例：

```
consumed {
  capacity_unit {
    read: 1
  }
}
next_start_primary_key {
  name: "CardID"
  value {
    type: STRING
    v_string: "2007035023"
  }
}
next_start_primary_key {
  name: "SellerID"
  value {
    type: STRING
    v_string: "00026"
  }
}
next_start_primary_key {
  name: "DeviceID"
  value {
    type: STRING
    v_string: "065499"
  }
}
next_start_primary_key {
  name: "OrderNumber"
  value {
    type: INTEGER
    v_int: 166666
```

```
  }
}
rows {
  primary_key_columns {
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  primary_key_columns {
    name: "SellerID"
    value {
      type: STRING
      v_string: "00022"
    }
  }
  primary_key_columns {
    name: "DeviceID"
    value {
      type: STRING
      v_string: "061104"
    }
  }
  primary_key_columns {
    name: "OrderNumber"
    value {
      type: INTEGER
      v_int: 142857
    }
  }
  attribute_columns {
    name: "Amount"
    value {
      type: DOUBLE
      v_double: 2.5
    }
  }
  attribute_columns {
    name: "Remarks"
    value {
      type: STRING
```

```
      v_string: "ice cream"
    }
  }
}
rows {
  primary_key_columns {
    name: "CardID"
    value {
      type: STRING
      v_string: "2007035023"
    }
  }
  primary_key_columns {
    name: "SellerID"
    value {
      type: STRING
      v_string: "00026"
    }
  }
  primary_key_columns {
    name: "DeviceID"
    value {
      type: STRING
      v_string: "065499"
    }
  }
  primary_key_columns {
    name: "OrderNumber"
    value {
      type: INTEGER
      v_int: 153846
    }
  }
  attribute_columns {
    name: "Amount"
    value {
      type: DOUBLE
      v_double: 0.5
    }
  }
}
```

BatchGetRow

行为：

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。

请求结构：

```
message BatchGetRowRequest {
    repeated TableInBatchGetRowRequest tables = 1;
}
```

tables

类型: repeated TableInBatchGetRowRequest

是否必要参数: 是

指定了需要要读取的行信息。

若 tables 中出现了下述情况，则操作整体失败，返回错误。

- tables 中任一表不存在。
- tables 中任一表名不符合表名命名规范。
- tables 中任一行未指定主键、主键名称不符合规范或者主键类型不正确。
- tables 中任一表的 columns_to_get 内的列名不符合列名命名规范。
- tables 中包含同名的表。
- tables 中任一表内包含主键完全相同的行。
- 所有 tables 中 RowInBatchGetRowRequest 的总个数超过 10 个。
- tables 中任一表内不包含任何 RowInBatchGetRowRequest。
- tables 中任一表的 columns_to_get 超过 128 列。

响应消息结构：

```
message BatchGetRowResponse {
    repeated TableInBatchGetRowResponse tables = 1;
}
```

tables

类型: repeated TableInBatchGetRowResponse

对应了每个 table 下读取到的数据。

响应消息中 TableInBatchGetRowResponse 对象的顺序与 BatchGetRowRequest 中的 TableInBatchGetRowRequest 对象的顺序相同；每个 TableInBatchGetRowResponse 下面的 RowInBatchGetRowResponse 对象的顺序与 TableInBatchGetRowRequest 下面的 RowInBatchGetRowRequest 相同。

如果某行不存在或者某行在指定的 columns_to_get 下没有数据，仍然会在 TableInBatchGetRowResponse 中有一条对应的 RowInBatchGetRowResponse，但其 row 下面的 primary_key_columns 和 attribute_columns 将为空。

若某行读取失败，该行所对应的 RowInBatchGetRowResponse 中 is_ok 将为 false，此时 row 将为空。

注意:BatchGetRow 操作可能会在行级别部分失败，此时返回的 HTTP 状态码仍为 200。应用程序必须对 RowInBatchGetRowResponse 中的 error 进行检查确认每一行的执行结果，并进行相应的处理。

服务能力单元消耗：

如果本次操作整体失败，不消耗任何服务能力单元。

如果请求超时，结果未定义，服务能力单元有可能被消耗，也可能未被消耗。

其他情况将每个 RowInBatchGetRowRequest 视为一个 GetRow 操作独立计算读服务能力单元。

请求示例：

```
BatchGetRowRequest {
  tables {
    table_name: "consume_history"
    rows {
      primary_key {
        name: "CardID"
        value {
```

```
          type: STRING
          v_string: "2007035023"
        }
      }
      primary_key {
        name: "SellerID"
        value {
          type: STRING
          v_string: "00022"
        }
      }
      primary_key {
        name: "DeviceID"
        value {
          type: STRING
          v_string: "061104"
        }
      }
      primary_key {
        name: "OrderNumber"
        value {
          type: INTEGER
          v_int: 142857
        }
      }
    }
    rows {
      primary_key {
        name: "CardID"
        value {
          type: STRING
          v_string: "2007035023"
        }
      }
      primary_key {
        name: "SellerID"
        value {
          type: STRING
          v_string: "00026"
        }
      }
      primary_key {
```

```
        name: "DeviceID"
        value {
          type: STRING
          v_string: "065499"
        }
      }
      primary_key {
        name: "OrderNumber"
        value {
          type: INTEGER
          v_int: 153846
        }
      }
    }
    columns_to_get: "CardID"
    columns_to_get: "SellerID"
    columns_to_get: "DeviceID"
    columns_to_get: "OrderNumber"
    columns_to_get: "Amount"
    columns_to_get: "Remarks"
  }
}
```

响应示例：

```
BatchGetRowResponse {
  tables {
    table_name: "consume_history"
    rows {
      is_ok: true
      consumed {
        capacity_unit {
          read: 1
        }
      }
      row {
        primary_key_columns {
          name: "CardID"
          value {
            type: STRING
            v_string: "2007035023"
          }
```

```
        }
        primary_key_columns {
          name: "SellerID"
          value {
            type: STRING
            v_string: "00022"
          }
        }
        primary_key_columns {
          name: "DeviceID"
          value {
            type: STRING
            v_string: "061104"
          }
        }
        primary_key_columns {
          name: "OrderNumber"
          value {
            type: INTEGER
            v_int: 142857
          }
        }
        attribute_columns {
          name: "Amount"
          value {
            type: DOUBLE
            v_double: 2.5
          }
        }
        attribute_columns {
          name: "Remarks"
          value {
            type: STRING
            v_string: "ice cream"
          }
        }
      }
    }
    rows {
      is_ok: true
      consumed {
        capacity_unit {
```

```
          read: 1
        }
      }
      row {
        primary_key_columns {
          name: "CardID"
          value {
            type: STRING
            v_string: "2007035023"
          }
        }
        primary_key_columns {
          name: "SellerID"
          value {
            type: STRING
            v_string: "00026"
          }
        }
        primary_key_columns {
          name: "DeviceID"
          value {
            type: STRING
            v_string: "065499"
          }
        }
        primary_key_columns {
          name: "OrderNumber"
          value {
            type: INTEGER
            v_int: 153846
          }
        }
        attribute_columns {
          name: "Amount"
          value {
            type: DOUBLE
            v_double: 0.5
          }
        }
      }
    }
  }
```

```
}
```

BatchWriteRow

行为：

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。

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。

请求结构：

```
message BatchWriteRowRequest {
    repeated TableInBatchWriteRowRequest tables = 1;
}
```

tables

类型: repeated TableInBatchWriteRowRequest

是否必要参数: 是

指定了需要要执行写操作的行信息。

以下情况都会返回整体错误:

- tables 中任一表不存在。
- tables 中包含同名的表。
- tables 中任一表名表名不符合表名命名规范。
- tables 中任一行操作未指定主键、主键列名称不符合规范或者主键列类型不正确。
- tables 中任一属性列名称不符合列名命名规范。
- tables 中任一行操作存在与主键列同名的属性列。
- tables 中任一主键列或者属性列的值大小超过上限。
- tables 中任一表中存在主键完全相同的 。
- tables 中所有表总的行操作个数超过 100 个，或者其含有的总数据大小超过 1M。

- tables 中任一表内没有包含行操作, 则返回 OTSParameterInvalidException 的错误
- tables 中任一 PutRowInBatchWriteRowRequest 包含的 Column 个数超过 128 个。
- tables 中任一 UpdateRowInBatchWriteRowRequest 包含的 ColumnUpdate 个数超过 128 个。

响应消息结构:

```
message BatchWriteRowResponse {
    repeated TableInBatchWriteRowResponse tables = 1;
}
```

tables

类型: repeated TableInBatchWriteRowResponse

对应了每个 table 下各操作的响应信息，包括是否成功执行，错误码和消耗的服务能力单元。

响应消息中 TableInBatchWriteRowResponse 对象的顺序与 BatchWriteRowRequest 中的 TableInBatchWriteRowRequest 对象的顺序相同；每个 TableInBatchWriteRowRequest 中 put_rows、update_rows、delete_rows 包含的 RowInBatchWriteRowResponse 对象的顺序分别与 TableInBatchWriteRowRequest 中 put_rows、update_rows、delete_rows 包含的 PutRowInBatchWriteRowRequest，UpdateRowInBatchWriteRowRequest 和 DeleteRowInBatchWriteRowRequest 对象的顺序相同。

若某行读取失败，该行所对应的 RowInBatchWriteRowResponse 中 is_ok 将为 false。

注意:BatchWriteRow 操作可能会在行级别部分失败，此时返回的 HTTP 状态码仍为 200。应用程序必须对 RowInBatchWriteRowResponse 中的 error 进行检查，确认每一行的执行结果并进行相应的处理。

服务能力单元消耗:

如果本次操作整体失败，不消耗任何服务能力单元。

如果请求超时，结果未定义，服务能力单元有可能被消耗，也可能未被消耗。

其他情况将每个 PutRowInBatchWriteRowRequest、UpdateRowInBatchWriteRowRequestDelete、RowInBatchWriteRowRequest 依次视作相对应的写操作独立计算读服务能力单元。

请求示例:

```
BatchWriteRowRequest {
  tables {
    table_name: "consume_history"
    put_rows {
      condition {
```

```
      row_existence: IGNORE
    }
    primary_key {
      name: "CardID"
      value {
        type: STRING
        v_string: "2007035023"
      }
    }
    primary_key {
      name: "SellerID"
      value {
        type: STRING
        v_string: "00022"
      }
    }
    primary_key {
      name: "DeviceID"
      value {
        type: STRING
        v_string: "061104"
      }
    }
    primary_key {
      name: "OrderNumber"
      value {
        type: INTEGER
        v_int: 142857
      }
    }
    attribute_columns {
      name: "Amount"
      value {
        type: DOUBLE
        v_double: 2.5
      }
    }
    attribute_columns {
      name: "Remarks"
      value {
        type: STRING
        v_string: "ice cream"
```

```
        }
      }
    }
    update_rows {
      condition {
        row_existence: EXPECT_EXIST
      }
      primary_key {
        name: "CardID"
        value {
          type: STRING
          v_string: "2007035023"
        }
      }
      primary_key {
        name: "SellerID"
        value {
          type: STRING
          v_string: "00026"
        }
      }
      primary_key {
        name: "DeviceID"
        value {
          type: STRING
          v_string: "065499"
        }
      }
      primary_key {
        name: "OrderNumber"
        value {
          type: INTEGER
          v_int: 153846
        }
      }
      attribute_columns {
        type: PUT
        name: "Amount"
        value {
          type: DOUBLE
          v_double: 1
        }
```

```
      }
      attribute_columns {
        type: DELETE
        name: "Remarks"
      }
    }
    delete_rows {
      condition {
        row_existence: IGNORE
      }
      primary_key {
        name: "CardID"
        value {
          type: STRING
          v_string: "2007035023"
        }
      }
      primary_key {
        name: "SellerID"
        value {
          type: STRING
          v_string: "00026"
        }
      }
      primary_key {
        name: "DeviceID"
        value {
          type: STRING
          v_string: "065499"
        }
      }
      primary_key {
        name: "OrderNumber"
        value {
          type: INTEGER
          v_int: 166666
        }
      }
    }
  }
}
```

响应示例：

```
tables {
  table_name: "consume_history"
  put_rows {
    is_ok: true
    consumed {
      capacity_unit {
        write: 1
      }
    }
  }
  update_rows {
    consumed {
      capacity_unit {
        write: 1
      }
    }
  }
  delete_rows {
    consumed {
      capacity_unit {
        write: 1
      }
    }
  }
}
```

CreateTable

行为：

根据给定的表结构信息创建相应的表。

请求结构：

```
message CreateTableRequest {
    required TableMeta table_meta = 1;
    required ReservedThroughput reserved_throughput = 2;
}
```

table_meta:

类型: TableMeta

是否必要参数: 是

将要创建的表的结构信息，其中 table_name 应在本实例范围内唯一；primary_key 的中 ColumnSchema 的个数应在 1~4 个范围内；primary_key 中的 ColumnSchema 的 name 应符合列名命名规范，type 取值只能为 STRING 或 INTEGER。

建表成功后，表的 Schema 将不能修改。

reserved_throughput:

类型: ReservedThroughput

是否必要参数: 是

将要创建的表的初始预留读写吞吐量设定，任何表的读服务能力单元与写服务能力单元均不能超过 5000。表的预留读写吞吐量设定可以通过 UpdateTable 进行动态更改。

响应消息结构:

```
message CreateTableResponse {
}
```

注意事项:

创建成功的表并不能立刻提供读写服务。一般来讲，在建表成功后一分钟左右，即可对新创建的表进行读写操作。

单个实例下不能超过 10 个表，如果需要提高单实例下表数目的上限，请使用人工服务提高此限额。

请求示例:

```
CreateTableRequest {
  table_meta {
    table_name: "consume_history"
    primary_key {
      name: "CardID"
      type: STRING
    }
    primary_key {
      name: "SellerID"
```

```
      type: STRING
    }
    primary_key {
      name: "DeviceID"
      type: STRING
    }
    primary_key {
      name: "OrderNumber"
      type: INTEGER
    }
  }
  capacity_unit {
    read: 100
    write: 100
  }
}
```

响应示例：

```
CreateTableResponse {
}
```

ListTable

行为：

获取当前实例下已创建的所有表的表名。

请求结构：

```
message ListTableRequest {
}
```

响应消息结构：

```
message ListTableResponse {
    repeated string table_names = 1;
}
```

table_names:

类型: repeated string

当前实例下所有表的表名。

注意事项:

若一个表刚刚创建成功，其表名会出现在 ListTableResponse 里，但此时该表不一定能够进行读写。

请求示例:

```
ListTableRequest {
}
```

响应示例:

```
ListTableResponse {
  table_names: "consume_history"
  table_names: "card_info"
  table_names: "seller_info"
}
```

DeleteTable

行为:

删除本实例下指定的表。

请求结构:

```
message DeleteTableRequest {
    required string table_name = 1;
}
```

table_name:

类型: string

是否必要参数: 是

需要删除的表的表名。

响应消息结构：

```
message DeleteTableResponse {
}
```

DeleteTable 的响应中没有任何错误即表示表已经成功删除。

请求示例：

```
DeleteTableRequest {
  table_name: "consume_history"
}
```

响应示例：

```
DeleteTableResponse {
}
```

UpdateTable

行为：

更新指定表的读服务能力单元或写服务能力单元设置，新设定将于更新成功一分钟内生效。

请求结构：

```
message UpdateTableRequest {
    required string table_name = 1;
    required ReservedThroughput reserved_throughput = 2;
}
```

table_name:

类型: string

是否必要参数: 是

需要更改预留读写吞吐量设置的表的表名。

reserved_throughput:

类型: ReservedThroughput

是否必要参数: 是

将要更改的表的预留读写吞吐量设定，该设定将于一分钟后生效。

可以只更改表的读服务能力单元设置或只更改表的写服务能力单元设置，也可以一并更改。

capacity_unit 中 read 和 write 应至少有一个非空，否则请求失败，返回错误。

响应消息结构:

```
message UpdateTableResponse {
    required ReservedThroughputDetails reserved_throughput_details = 1;
}
```

capacity_unit_details:

类型: ReservedThroughputDetails

更新后该表的预留读写吞吐量设置信息，除了包含当前的预留读写吞吐量设置值之外，还包含了最近一次更新该表的预留读写吞吐量设置的时间和当日已下调预留读写吞吐量的次数。

tips

调整每个表预留读写吞吐量的最小时间间隔为 10 分钟，如果本次 UpdateTable 操作距上次不到 10 分钟将被拒绝。

每个自然日 (UTC 时间 00:00:00 到第二天的 00:00:00) 内每个表上调预留读写吞吐量次数不限，但下调预留读写吞吐量次数不能超过 4 次。下调写服务能力单元或者读服务能力单元其中之一即视为下调预留读写吞吐量。

请求示例:

```
UpdateTableRequest {
  table_name: "consume_history"
  capacity_unit {
    read: 10
    write: 150
  }
}
```

响应示例:

```
UpdateTableResponse {
  capacity_unit_details {
    capacity_unit {
```

```
      read: 10
      write: 150
    }
    last_increase_time: 1407507306
    last_decrease_time: 1407507306
    number_of_decreases_today: 2
  }
}
```

DescribeTable

行为：

查询指定表的结构信息和预留读写吞吐量设置信息。

请求结构：

```
message DescribeTableRequest {
    required string table_name = 1;
}
```

table_name:

类型: string

是否必要参数: 是

需要查询的表名。

响应消息结构：

```
message DescribeTableResponse {
    required TableMeta table_meta = 1;
    required ReservedThroughputDetails reserved_throughput_details = 2;
}
```

table_meta:

类型: TableMeta

该表的 Schema，与建表时给出的 Schema 相同。

reserved_throughput_details:

类型: ReservedThroughputDetails

该表的预留读写吞吐量设置信息，除了包含当前的预留读写吞吐量设置值之外，还包含了最近一次更新该表的预留读写吞吐量设置的时间和当日已下调预留读写吞吐量的次数。

请求示例:

```
DescribeTableRequest {
  table_name: "consume_history"
}
```

响应示例:

```
DescribeTableResponse {
  table_meta {
    table_name: "consume_history"
    primary_key {
      name: "CardID"
      type: STRING
    }
    primary_key {
      name: "SellerID"
      type: STRING
    }
    primary_key {
      name: "DeviceID"
      type: STRING
    }
    primary_key {
      name: "OrderNumber"
      type: INTEGER
    }
  }
  capacity_unit_details {
    capacity_unit {
      read: 10
      write: 150
    }
    last_increase_time: 1407507306
    last_decrease_time: 1407507306
    number_of_decreases_today: 2
```

```
  }
}
```

Data Type

CapacityUnit

表示一次操作消耗服务能力单元的值或是一个表的预留读写吞吐量的值。

数据结构

```
message CapacityUnit {
    optional int32 read = 1;
    optional int32 write = 2;
}
```

read

类型: int32

描述: 本次操作消耗的读服务能力单元或该表的读服务能力单元。

write

类型: int32

描述: 本次操作消耗的写服务能力单元或该表的写服务能力单元。

相关操作

UpdateRow

BatchWriteRow

Column

表示一列。

数据结构

```
message Column {
    required string name = 1;
    required ColumnValue value = 2;
}
```

name

类型: string

描述: 该列的列名。

value

类型: ColumnValue

描述: 该列的列值。

ColumnSchema

定义一列，只用于主键列。

数据结构

```
message ColumnSchema {
    required string name = 1;
    required ColumnType type = 2;
}
```

name

类型: string

描述: 该列的列名。

type

类型: ColumnType

描述: 该列的数据类型。

相关操作

CreateTable

DescribeTable

ColumnType

表示一列的数据类型，枚举类型。

INF_MIN 和 INF_MAX 为 GetRange 操作专用类型，value 的 type 为 INF_MIN 的 Column 为小于其它所有 Column，value 的 type 为 INF_MAX 的 Column 大于其它所有 Column。

枚举取值列表

```
enum ColumnType {
    INF_MIN = 0;
    INF_MAX = 1;
    INTEGER = 2;
    STRING = 3;
    BOOLEAN = 4;
    DOUBLE = 5;
    BINARY = 6;
}
```

ColumnUpdate

在 UpdateRow 中，表示更新一列的信息。

数据结构

```
message ColumnUpdate {
    required OperationType type = 1;
    required string name = 2;
    optional ColumnValue value = 3;
}
```

type

类型: OperationType

描述: 对该列的修改方式，更新或删除。

name

类型: string

描述: 该列的列名。

value

类型: ColumnValue

描述: 该列更新后的列值，在 type 为 PUT 时有效。

相关操作

UpdateRow

BatchWriteRow

ColumnValue

表示一列的列值。

数据结构

```
message ColumnValue {
    required ColumnType type = 1;
    optional int64 v_int = 2;
    optional string v_string = 3;
    optional bool v_bool = 4;
    optional double v_double = 5;
    optional bytes v_binary = 6;
}
```

type

类型: ColumnType

描述: 该列的数据类型。

v_int

类型: int64

描述: 该列的数据，只在 type 为 INTEGER 时有效。

v_string

类型: string

描述: 该列的数据，只在 type 为 STRING 时有效，必须为 UTF-8 编码。

v_bool

类型: bool

描述: 该列的数据，只在 type 为 BOOLEAN 时有效。

v_double

类型: double

描述: 该列的数据，只在 type 为 DOUBLE 时有效。

v_binary

类型: bytes

描述: 该列的数据，只在 type 为 BINARY 时有效。

Condition

在 PutRow，UpdateRow 和 DeleteRow 中使用的行判断条件，目前只含有 row_existence 一项。

数据结构

```
message Condition {
    required RowExistenceExpectation row_existence = 1;
}
```

row_existence

类型: RowExistenceExpectation

描述: 对该行进行行存在性检查的设置。

相关操作

PutRow

UpdateRow

DeleteRow

BatchWriteRow

ConsumedCapacity

表示一次操作消耗的服务能力单元。

数据结构

```
message ConsumedCapacity {
    required CapacityUnit capacity_unit = 1;
}
```

capacity_unit

类型: CapacityUnit

描述: 本次操作消耗的服务能力单元的值。

DeleteRowInBatchWriteRowRequest

在 BatchWriteRow 操作中，表示要删除的一行信息。

数据结构

```
message DeleteRowInBatchWriteRowRequest {
    required Condition condition = 1;
    repeated Column primary_key = 2;
}
```

condition

类型: Condition

描述: 在数据删除前是否进行存在性检查。

primary_key

类型: repeated Column

描述: 请求删除的行的全部主键列。

相关操作

BatchWriteRow

Direction

在 GetRange 操作中，查询数据的顺序。

- FORWARD 表示此次查询按照主键由小到大的顺序进行。
- BACKWARD 表示此次查询按照主键由大到小的顺序进行。

枚举取值列表

```
enum Direction {
    FORWARD = 0;
    BACKWARD = 1;
}
```

相关操作

GetRange

Error

用于在操作失败时的响应消息中表示错误信息，以及在 BatchGetRow 和 BatchWriteRow 操作的响应消息中表示单行请求的错误。

数据结构

```
Error {
    required string code = 1;
    optional string message = 2;
}
```

code

类型: string

描述: 当前单行操作的错误码，具体含义可参考 OTS 存储相关异常。

message

类型: string

描述: 当前单行操作的错误信息，具体含义可参考 OTS 存储相关异常。

相关操作

BatchGetRow

BatchWriteRow

OperationType

在 UpdateRow 中，表示对一列的修改方式。

- PUT 表示插入一列或覆盖该列的数据。
- DELETE 表示删除该列。

枚举取值列表

```
enum OperationType {
    PUT = 1;
    DELETE = 2;
}
```

PutRowInBatchWriteRowRequest

在 BatchWriteRow 操作中，表示要插入的一行信息。

数据结构

```
message PutRowInBatchWriteRowRequest {
    required Condition condition = 1;
    repeated Column primary_key = 2;
    repeated Column attribute_columns = 3;
}
```

condition

类型: Condition

描述: 在数据插入前是否进行行存在性检查。

primary_key

类型: repeated Column

描述: 请求插入的行全部主键列。

attribute_columns

类型: repeated Column

描述: 请求插入的行的属性。

相关操作

BatchWriteRow

ReservedThroughput

表示一个表设置的预留读写吞吐量数值。

数据结构

```
message ReservedThroughput {
    required CapacityUnit capacity_unit = 1;
}
```

capacity_unit

类型: CapacityUnit

描述: 表当前的预留读写吞吐量数值。

相关操作

CreateTable

UpdateTable

DescribeTable

ReservedThroughputDetails

数据结构

表示一个表的预留读写吞吐量信息。

```
message ReservedThroughputDetails {
    required CapacityUnit capacity_unit = 1;
    required int64 last_increase_time = 2;
    optional int64 last_decrease_time = 3;
    required int32 number_of_decreases_today = 4;
}
```

capacity_unit

类型: CapacityUnit

描述: 该表的预留读写吞吐量的数值。

last_increase_time

类型: int64

描述: 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

类型: int64

描述: 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。

number_of_decreases_today

类型: int32

描述: 本个自然日内已下调该表的预留读写吞吐量设置的次数。

相关操作

UpdateTable

DescribeTable

Row

在 GetRow 和 GetRange 的响应消息中，表示一行数据。

数据结构

```
message Row {
    repeated Column primary_key_columns = 1;
    repeated Column attribute_columns = 2;
}
```

primary_key_columns

类型: repeated Column

描述: 表示该行需要返回的全部主键列。

attribute_columns

类型: repeated Column

描述: 表示该行需要返回的全部属性列。

相关操作

GetRow

GetRange

[BatchGetRow]](#BatchGetRow)

RowExistenceExpectation

行存在性判断条件，枚举类型。

- IGNORE 表示不做行存在性检查。
- EXPECT_EXIST 表示期待该行存在。
- EXPECT_NOT_EXIST 表示期待该行不存在。

枚举取值列表

```
enum RowExistenceExpectation {
    IGNORE = 0;
    EXPECT_EXIST = 1;
    EXPECT_NOT_EXIST = 2;
}
```

相关操作

PutRow

UpdateRow

DeleteRow

BatchWriteRow

RowInBatchGetRowResponse

在 BatchGetRow 操作的返回消息中，表示一行数据。

数据结构

```
message RowInBatchGetRowResponse {
    required bool is_ok = 1 [default = true];
    optional Error error = 2;
    optional ConsumedCapacity consumed = 3;
    optional Row row = 4;
}
```

is_ok

类型: bool

描述: 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，row 无效。

error

类型: Error

描述: 该行操作的错误信息。

consumed

类型: ConsumedCapacity

描述: 该行操作消耗的服务能力单元。

row

类型: row

描述: 该行需要返回的列数据集合。

相关操作

BatchGetRow

RowInBatchGetRowRequest

在 BatchGetRow 操作中，表示要读取的一行请求信息。

数据结构

```
message RowInBatchGetRowRequest {
    repeated Column primary_key = 1;
}
```

primary_key

类型: repeated Column

描述: 需要读取的行的全部主键列。

相关操作

BatchGetRow

RowInBatchWriteRowResponse

在 BatchWriteRow 操作的返回消息中，表示一行写入操作的结果。

数据结构

```
message RowInBatchWriteRowResponse {
    required bool is_ok = 1 [default = true];
```

```
    optional Error error = 2;
    optional ConsumedCapacity consumed = 3;
}
```

is_ok

类型: bool

描述: 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

类型: Error

描述: 该行操作的错误信息。

consumed

类型: ConsumedCapacity

描述: 该行操作消耗的服务能力单元。

相关操作

BatchWriteRow

TableInBatchGetRowRequest

在 BatchGetRow 操作中，表示要读取的一个表的请求信息。

数据结构

```
message TableInBatchGetRowRequest {
    required string table_name = 1;
    repeated RowInBatchGetRowRequest rows = 2;
    repeated string columns_to_get = 3;
}
```

table_name

类型: string

描述: 该表的表名。

rows

类型: repeated RowInBatchGetRowRequest

描述: 该表中需要读取的全部行的信息。

columns_to_get

类型: repeated string

描述: 该表中需要返回的全部列的列名。

相关操作

BatchGetRow

TableInBatchGetRowResponse

在 BatchGetRow 操作的返回消息中，表示一个表的数据。

数据结构

```
message TableInBatchGetRowResponse {
    required string table_name = 1;
    repeated RowInBatchGetRowResponse rows = 2;
}
```

table_name

类型: string

描述: 该表的表名。

rows

类型: repeated RowInBatchGetRowResponse

描述: 该表中读取到的全部行数据。

相关操作

BatchGetRow

TableInBatchWriteRowRequest

在 BatchWriteRow 操作中，表示要写入的一个表的请求信息。

数据结构

```
message TableInBatchWriteRowRequest {
    required string table_name = 1;
    repeated PutRowInBatchWriteRowRequest put_rows = 2;
    repeated UpdateRowInBatchWriteRowRequest update_rows = 3;
    repeated DeleteRowInBatchWriteRowRequest delete_rows = 4;
}
```

table_name

类型: string

描述: 该表的表名。

put_rows

类型: repeated PutRowInBatchWriteRowRequest

描述: 该表中请求插入的行信息。

update_rows

类型: repeated UpdateRowInBatchWriteRowRequest

描述: 该表中请求更新的行信息。

delete_rows

类型: repeated DeleteRowInBatchWriteRowRequest

描述: 该表中请求删除的行信息。

相关操作

BatchWriteRow

TableInBatchWriteRowResponse

在 BatchWriteRow 操作中，表示对一个表进行写入的结果。

数据结构

```
message TableInBatchWriteRowResponse {
    required string table_name = 1;
```

```
    repeated RowInBatchWriteRowResponse put_rows = 2;
    repeated RowInBatchWriteRowResponse update_rows = 3;
    repeated RowInBatchWriteRowResponse delete_rows = 4;
}
```

table_name

类型: string

描述: 该表的表名。

put_rows

类型: RowInBatchWriteRowResponse

描述: 该表中 PutRow 操作的结果。

update_rows

类型: RowInBatchWriteRowResponse

描述: 该表中 UpdateRow 操作的结果。

delete_rows

类型: RowInBatchWriteRowResponse

描述: 该表中 DeleteRow 操作的结果。

相关操作

BatchWriteRow

TableMeta

表示一个表的结构信息。

数据结构

```
message TableMeta {
    required string table_name = 1;
    repeated ColumnSchema primary_key = 2;
}
```

table_name

类型: string

描述: 该表的表名。

primary_key

类型: repeated ColumnSchema

描述: 该表全部的主键列。

相关操作

CreateTable

DescribeTable

UpdateRowInBatchWriteRowRequest

在 BatchWriteRow 操作中，表示要更新的一行信息。

数据结构

```
message UpdateRowInBatchWriteRowRequest {
    required Condition condition = 1;
    repeated Column primary_key = 2;
    repeated ColumnUpdate attribute_columns = 3;
}
```

condition

类型: Condition

描述: 在数据更新前是否进行行存在性检查。

primary_key

类型: repeated Column

描述: 请求更新的行的全部主键列。

attribute_columns

类型: repeated ColumnUpdate

描述: 请求更新的全部 **AttributColumn**，

相关操作

BatchWriteRow

第 8 章　限制项

限制项	限制范围	说明
一个阿里云用户帐号下可以保有实例数	不超过 10	如有需求提高上限，请联系客服
一个实例中表的个数	不超过 10	如有需求提高上限，请联系客服
实例名字长度	3-16 bytes	字符集为 [a-zA-Z0-9] 和连词符 (-), 首字符必须是字母且末尾字符不能为连词符 (-)
表名长度	1-255 bytes	字符集为 [a-zA-Z0-9_]，首字符必须是字母或 (_)
列名长度限制	1-255 bytes	字符集为 [a-zA-Z0-9_]，首字符必须是字母或 (_)
主键包含的列数	1-4	至少 1 列，至多 4 列
String 类型主键列列值大小	不超过 1KB	单一主键列 String 类型的列列值大小上限 1KB
String 类型属性列列值大小	不超过 64KB	单一属性列 String 类型的列列值大小上限 64KB
Binary 类型列列值大小	不超过 64KB	单一列 Binary 类型的列列值大小
一行中属性列的个数	不超过 128	单一行最多拥有 128 个属性列
一行中属性列列值总大小	不超过 256KB	单一行中所有属性列的列值总和大小上限 256KB
单表的预留读写吞吐量	1-5000	如有需求提高上限，请联系客服
读请求中 columns_to_get 参数的列的个数	0-128	读请求
单表 UpdateTable 的次数	上调：无限制，下调：一个自然日内 4 次	自然日从 UTC 时间 00:00:00 开始到第二天的 00:00:00 结束，即北京时间早上 8 点到第二天早上 8 点。如有需求一天内下调超过 4 次，请联系客服。
单表 UpdateTable 的频率	每 10 分钟 1 次	单表在 10 分钟之内，最多允许调整 1 次预留读/写能力值
BatchGetRow 一次操作请求读取的行数	不超过 10	N/A
BatchWriteRow 一次操作请求写入行数	不超过 100	N/A
GetRange 一次返回的数据	5000 行或者 1MB	一次返回数据的行数超过 5000 行，或者返回数据的数据大小大于 1MB。以上任一条件满足时，超出上限的数据将会被截掉

限制项	限制范围	说明
一次 HTTP 请求 Request Body 的数据大小	不超过 2MB	

第 9 章　附录

OTS ProtocolBuffer 消息定义

```
package com.aliyun.cloudservice.ots2;

message Error {
    required string code = 1;
    optional string message = 2;
}


enum ColumnType {
    INF_MIN = 0; // only for GetRange
    INF_MAX = 1; // only for GetRange
    INTEGER = 2;
    STRING = 3;
    BOOLEAN = 4;
    DOUBLE = 5;
    BINARY = 6;
}

message ColumnSchema {
    required string name = 1;
    required ColumnType type = 2;
}

message ColumnValue {
    required ColumnType type = 1;
    optional int64 v_int = 2;
    optional string v_string = 3;
    optional bool v_bool = 4;
    optional double v_double = 5;
    optional bytes v_binary = 6;
```

```
}

message Column {
    required string name = 1;
    required ColumnValue value = 2;
}

message Row {
    repeated Column primary_key_columns = 1;
    repeated Column attribute_columns = 2;
}

message TableMeta {
    required string table_name = 1;
    repeated ColumnSchema primary_key = 2;
}

enum RowExistenceExpectation {
    IGNORE = 0;
    EXPECT_EXIST = 1;
    EXPECT_NOT_EXIST = 2;
}

message Condition {
    required RowExistenceExpectation row_existence = 1;
}

message CapacityUnit {
    optional int32 read = 1;
    optional int32 write = 2;
}

message ReservedThroughputDetails {
    required CapacityUnit capacity_unit = 1;
    required int64 last_increase_time = 2;
    optional int64 last_decrease_time = 3;
    required int32 number_of_decreases_today = 4;
}

message ReservedThroughput {
    required CapacityUnit capacity_unit = 1;
}
```

```
73
74  message ConsumedCapacity {
75      required CapacityUnit capacity_unit = 1;
76  }
77
78  /* CreateTable */
79  message CreateTableRequest {
80      required TableMeta table_meta = 1;
81      required ReservedThroughput reserved_throughput = 2;
82  }
83
84  message CreateTableResponse {
85  }
86
87  /* UpdateTable */
88
89  message UpdateTableRequest {
90      required string table_name = 1;
91      required ReservedThroughput reserved_throughput = 2;
92  }
93
94  message UpdateTableResponse {
95      required ReservedThroughputDetails reserved_throughput_details = 1;
96  }
97
98  /* DescribeTable */
99  message DescribeTableRequest {
100     required string table_name = 1;
101 }
102
103 message DescribeTableResponse {
104     required TableMeta table_meta = 1;
105     required ReservedThroughputDetails reserved_throughput_details = 2;
106 }
107
108 /* ListTable */
109 message ListTableRequest {
110 }
111
112 message ListTableResponse {
113     repeated string table_names = 1;
114 }
```

```
115
116 /* DeleteTable */
117 message DeleteTableRequest {
118     required string table_name = 1;
119 }
120
121 message DeleteTableResponse {
122 }
123
124 /* GetRow */
125 message GetRowRequest {
126     required string table_name = 1;
127     repeated Column primary_key = 2;
128     repeated string columns_to_get = 3;
129 }
130
131 message GetRowResponse {
132     required ConsumedCapacity consumed = 1;
133     required Row row = 2;
134 }
135
136 /* UpdateRow */
137 enum OperationType {
138     PUT = 1;
139     DELETE = 2;
140 }
141
142 message ColumnUpdate {
143     required OperationType type = 1;
144     required string name = 2;
145     optional ColumnValue value = 3;
146 }
147
148 message UpdateRowRequest {
149     required string table_name = 1;
150     required Condition condition = 2;
151     repeated Column primary_key = 3;
152     repeated ColumnUpdate attribute_columns = 4;
153 }
154
155 message UpdateRowResponse {
156     required ConsumedCapacity consumed = 1;
```

```
}

/* PutRow */
message PutRowRequest {
    required string table_name = 1;
    required Condition condition = 2;
    repeated Column primary_key = 3;
    repeated Column attribute_columns = 4;
}

message PutRowResponse {
    required ConsumedCapacity consumed = 1;
}

/* DeleteRow */
message DeleteRowRequest {
    required string table_name = 1;
    required Condition condition = 2;
    repeated Column primary_key = 3;
}

message DeleteRowResponse {
    required ConsumedCapacity consumed = 1;
}

/* BatchGetRow */
message RowInBatchGetRowRequest {
    repeated Column primary_key = 1;
}

message TableInBatchGetRowRequest {
    required string table_name = 1;
    repeated RowInBatchGetRowRequest rows = 2;
    repeated string columns_to_get = 3;
}

message BatchGetRowRequest {
    repeated TableInBatchGetRowRequest tables = 1;
}

message RowInBatchGetRowResponse {
    required bool is_ok = 1 [default = true];
```

```
    optional Error error = 2;
    optional ConsumedCapacity consumed = 3;
    optional Row row = 4;
}

message TableInBatchGetRowResponse {
    required string table_name = 1;
    repeated RowInBatchGetRowResponse rows = 2; // same indices w.r.t. request
}

message BatchGetRowResponse {
    repeated TableInBatchGetRowResponse tables = 1; // same indices w.r.t. request
}

/* BatchWriteRow */
message PutRowInBatchWriteRowRequest {
    required Condition condition = 1;
    repeated Column primary_key = 2;
    repeated Column attribute_columns = 3;
}

message UpdateRowInBatchWriteRowRequest {
    required Condition condition = 1;
    repeated Column primary_key = 2;
    repeated ColumnUpdate attribute_columns = 3;
}

message DeleteRowInBatchWriteRowRequest {
    required Condition condition = 1;
    repeated Column primary_key = 2;
}

message TableInBatchWriteRowRequest {
    required string table_name = 1;
    repeated PutRowInBatchWriteRowRequest put_rows = 2;
    repeated UpdateRowInBatchWriteRowRequest update_rows = 3;
    repeated DeleteRowInBatchWriteRowRequest delete_rows = 4;
}

message BatchWriteRowRequest {
    repeated TableInBatchWriteRowRequest tables = 1; // same indices w.r.t. request
}
```

```
241
242 message RowInBatchWriteRowResponse {
243     required bool is_ok = 1 [default = true];
244     optional Error error = 2;
245     optional ConsumedCapacity consumed = 3;
246 }
247
248 message TableInBatchWriteRowResponse {
249     required string table_name = 1;
250     repeated RowInBatchWriteRowResponse put_rows = 2; // same indices w.r.t. request
251     repeated RowInBatchWriteRowResponse update_rows = 3; // same indices w.r.t. request
252     repeated RowInBatchWriteRowResponse delete_rows = 4; // same indices w.r.t. request
253 }
254
255 message BatchWriteRowResponse {
256     repeated TableInBatchWriteRowResponse tables = 1;
257 }
258
259 /* GetRange */
260 enum Direction {
261     FORWARD = 0;
262     BACKWARD = 1;
263 }
264
265 message GetRangeRequest {
266     required string table_name = 1;
267     required Direction direction = 2;
268     repeated string columns_to_get = 3;
269     optional int32 limit = 4;
270     repeated Column inclusive_start_primary_key = 5; // required all PKs, possibly filled
        with INF_MIN/INF_MAX
271     repeated Column exclusive_end_primary_key = 6; // required all PKs, possibly filled
        with INF_MIN/INF_MAX
272 }
273
274 message GetRangeResponse {
275     required ConsumedCapacity consumed = 1;
276     repeated Column next_start_primary_key = 2; // missing means hitting the end
277     repeated Row rows = 3;
278 }
```

OTS 术语中英对照表

中文	英文
节点	Region
实例	Instance
表	Table
分片	Partition
行	Row
主键	PrimaryKey
主键列	PrimaryKeyColumn
分片键	PartitionKey
属性	Attributes
属性列	AttributeColumn
预留读写吞吐量	ReservedThroughput
服务能力单元	CapacityUnit
服务地址	EndPoint