表格存储

开发指南

开发指南

表格存储通过 AccessKey 和实例（Instance）来保证访问表格存储的安全性。

实例的访问

对表格存储表数据的访问，首先需要确定该表所在的实例。实例是阿里云用户开通表格存储服务后，管理资源的逻辑容器。用户可以通过表格存储控制台，对表格存储的实例进行创建、管理及删除等操作。成功创建实例后，在实例中管理表格存储的表。其中实例名称（Instance Name）在阿里云节点（Region）范围内是唯一的，在不同的阿里云节点内可以存在同名的实例。实例名称会作为访问地址的前缀：

公网访问地址：http://instanceName.region.ots.aliyuncs.com

阿里云私网地址：http://instanceName.region.ots-internal.aliyuncs.com

例如，华东 1 节点，实例名称为 myInstance 的访问域名为:

公网：http://myInstance.cn-hangzhou.ots.aliyuncs.com

私网：http://myInstance.cn-hangzhou.ots-internal.aliyuncs.com

AccessKey

表格存储根据 AccessKey 对请求进行身份认证和鉴权，所以每个合法的表格存储请求都必须携带正确的 AccessKey 信息。

每个阿里云账户可以创建最多 5 个 AccessKey，用户可以在阿里云用户中心中管理自己的 AccessKey。AccessKey 由 Access Key ID 和 Access Key Secret 组成。Access Key ID 用于标识 AccessKey，Access Key Secret 用于加密表格存储的请求。Access Key Secret 是认证请求身份的重要凭证，因此需要保证 Access Key Secret 的保密和安全。一个账户下的 AccessKey 可以被用于访问该账户中的所有实例。

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

创建表格存储的表时，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。表格存储是 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

表格存储的表的名称必须符合以下规范:

由英文字符（a-z）或（A-Z）、数字（0-9）和下划线（_）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称可以相同。

主键

创建表格存储的表时必须指定表的主键。主键包含 1~4 个主键列，每个主键列都有名字和类型。表格存储对主键列的名字和类型都有限制，详细信息可以参考表格存储数据模型的主键一节。

表格存储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

为确保应用程序获得稳定、低延时的服务，应用程序可以在创建表的时候指定预留读/写吞吐量。如果预留读/写吞吐量不为 0，表格存储根据预留读/写吞吐量来为表分配相应的资源，满足应用程序的预留吞吐量需求，同时根据配置的预留读/写吞吐量收取相应费用。应用程序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。

注意：容量型实例下的表不支持预留读/写吞吐量。

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 2 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 12:45 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 2 分钟的限制是针对表的，在 12:43 AM ~ 12:45 AM 之间，用户可以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），上调或者下调预留读/写吞吐量的总次数不做限制，但是两次调整之间的时间间隔必须大于 2 分

钟。调整预留读/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更新，则此次操作被视为对表进行了更新。

预留读/写吞吐量调整完毕后 1 分钟内生效。

对于访问消耗的读/写吞吐量中，超出预留读/写吞吐量的部分，会计入按量读/写吞吐量，并根据按量读/写吞吐量单价进行计费。

由于预留读/写吞吐量在单价上低于按量读/写吞吐量，配置合适的预留读/写吞吐量可以进一步降低成本。例如，在用户刚建表之后如果需要导入大量数据，可以设置较大的预留写吞吐量，能够以较低的写成本将数据导入进来，当数据导入完毕后，再将预留读/写吞吐量下调。

分区键下的数据量限制

表格存储按照分区键的范围对表的数据进行分区，拥有相同分区键的行会被划分到同一个分区。为了防止分区过大无法切分，建议单个分区键下的所有行数据大小之和不要超过 1 GB。

建表后加载时间

表格存储的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

表格存储表的最佳实践

使用表格存储的 SDK 进行表操作

JAVA-SDK (NEW) — 表操作

Python-SDK — 表操作

表格存储的表由行组成，每一行包含主键和属性。本节将介绍表格存储数据的操作方法。

表格存储行简介

组成表格存储表的基本单位是行，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。更多信息请参见表格存储的数据模型概念。

表格存储的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，则先删除旧行，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，则新增一行。

DeleteRow：删除一行。

批量操作

BatchGetRow：批量读取多行数据。

BatchWriteRow：批量插入、更新或者删除多行数据。

范围读取

- GetRange：读取表中一个范围内的数据。

表格存储单行操作

单行写入操作

表格存储的单行写操作有三种：PutRow、UpdateRow 和 DeleteRow。下面分别介绍每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。表格存储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，若 UpdateRow 请求只包含删除指定的列，且该行不存在，则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，是否需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，表格存储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如下：

PutRow：本次消耗的写 CU 为修改的行主键数据大小与属性列数据大小之和除以 4 KB 向上取整。若指定条件检查不为 IGNORE，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。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和 1 单位读服务能力单元。请参见 PutRow 详解。

UpdateRow：本次消耗的写 CU 为修改的行主键数据大小与属性列数据大小之和除以 4 KB 向上取整。UpdateRow 中包含的需要删除的属性列，只有其列名计入该属性列数据大小。若指定条件检查不为 IGNORE，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。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和 1 单位读服务能力单元。请参见 UpdateRow 详解。

DeleteRow：被删除的行主键数据大小除以 4 KB 向上取整。若指定条件检查不为 IGNORE，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。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。请参见 DeleteRow 详解。

写操作会根据指定的 condition 情况消耗一定的读服务能力单元。

示例

下面举例说明单行写操作的写服务能力单元和读服务能力单元的计算。

示例 1，使用 PutRow 进行如下行写入操作：

```
//PutRow 操作
//row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+3000Byte=4322Byte
{
primary_keys:{'pk':1},
attributes:{'value1':String(1300Byte), 'value2':String(3000Byte)}
}

//原来的行
//row_size=len('pk')+len('value2')+8Byte+900Byte=916Byte
//row_primarykey_size=len('pk')+8Byte=10Byte
{
```

```
primary_keys:{'pk':1},
attributes:{'value2':String(900Byte)}
}
```

读/写服务能力单元的消耗情况如下：

将 condition 设置为 EXPECT_EXIST 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗的读服务能力单元为该行主键数据大小 10 Byte 除以 4 KB 向上取整。该 PutRow 操作消耗 2 个单位的写服务能力单元和 1 个单位的读服务能力单元。

将 condition 设置为 IGNORE 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元。该 PutRow 操作消耗 2 个单位的写服务能力单元和 0 个单位的读服务能力单元。

将 condition 设置为 EXPECT_NOT_EXIST 时：指定的行存在性检查条件检查失败，该 PutRow 操作消耗 1 个单位的写服务能力单元和 1 个单位的读服务能力单元。

示例 2，使用 UpdateRow 新写入一行：

```
//UpdateRow 操作
//删除的属性列列名长度计入 row_size
//row_size=len('pk')+len('value1')+len('value2')+8Byte+900Byte=922Byte
{
primary_keys:{'pk':1},
attributes:{'value1':String(900Byte), 'value2':Delete}
}

//原来的行不存在
//row_size=0
```

读/写服务能力单元的消耗情况如下：

将 condition 设置为 IGNORE 时：消耗的写服务能力单元为 922 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元。该 UpdateRow 操作消耗 1个单位的写服务能力单元和 0 个读服务能力单元。

将 condition 设置为 EXPECT_EXIST 时：指定的行存在性检查条件检查失败，该 PutRow 操作消耗 1 个单位的写服务能力单元和 1 个单位的读服务能力单元。

示例 3，使用 UpdateRow 对存在的行进行更新操作：

```
//UpdateRow 操作
//row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+3000Byte=4322Byte
{
primary_keys:{'pk':1},
attributes:{'value1':String(1300Byte), 'value2':String(3000Byte)}
```

```
}
//原来的行
//row_size=len('pk')+len('value1')+8Byte+900Byte=916Byte
//row_primarykey_size=len('pk')+8Byte=10Byte
{
primary_keys:{'pk':1},
attributes:{'value1':String(900Byte)}
}
```

读/写服务能力单元的消耗情况如下：

将 condition 设置为 EXPECT_EXIST 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗的读服务能力单元为该行主键数据大小 10 Byte 除以 4 KB 向上取整。该 UpdateRow 操作消耗 2 个单位的写服务能力单元和 1 个单位的读服务能力单元。

将 condition 设置为 IGNORE 时：消耗的写服务能力单元为 4322 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元，该 UpdateRow 操作消耗 2 个单位的写服务能力单元和 0 个单位的读服务能力单元。

示例 4，使用 DeleteRow 删除不存在的行：

```
//原来的行不存在
//row_size=0

//DeleteRow 操作
//row_size=0
//row_primarykey_size=len('pk')+8Byte=10Byte
{
primary_keys:{'pk':1},
}
```

修改前后的数据大小均为 0，无论读写操作成功还是失败至少消耗 1 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。

读/写服务能力单元的消耗情况如下：

将 condition 设置为 EXPECT_EXIST 时：消耗的写服务能力单元为该行主键数据大小 10 Byte 除以 4 KB 向上取整，消耗的读服务能力单元为该主键数据大小 10 Byte 除以 4 KB 向上取整。该 DeleteRow 操作消耗 1 个单位的写服务能力单元和 1 个单位的读服务能力单元。

将 condition 设置为 IGNORE 时：消耗的写服务能力单元为该行主键数据大小 10 Byte 除以 4 KB 向上取整，消耗 0 个读服务能力单元。该 DeleteRow 操作消耗 1 个单位的写服务能力单元和 0 个单位的读服务能力单元。

更多信息请参见 API Reference 中的 PutRow、UpdateRow 和 DeleteRow 章节。

单行读取操作

表格存储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。

表格存储根据被读取的行主键的数据大小与实际读取的属性列数据大小之和计算读服务能力单元。将行数据大小除以 4 KB 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。

示例

使用 GetRow 读取一行消耗的写服务能力单元的计算：

```
//被读取的行

//row_size=len('pk')+len('value1')+len('value2')+8Byte+1200Byte+3100Byte=4322Byte
{
primary_keys:{'pk':1},
attributes:{'value1':String(1200Byte), 'value2':String(3100Byte)}
}

//GetRow 操作
//获取的数据 size=len('pk')+len('value1')+8Byte+1200Byte=1216Byte
{
primary_keys:{'pk':1},
columns_to_get:{'value1'}
}
```

消耗的读服务能力单元为 1216 Byte 除以 4 KB 向上取整，该 GetRow 操作消耗 1 个单位的读服务能力单元。

更多信息请参见 API Reference 的 GetRow 章节。

多行操作

表格存储提供了 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，表格存储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，表格存储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。

更多信息请参见 API Reference 中的 BatchWriteRow 与 BatchGetRow 章节。

范围读取操作

表格存储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。

表格存储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。表格存储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：

返回的行数据大小之和达到 1 MB。

返回的行数等于 5000。

返回的行数等于最大返回行数。

当前剩余的预留读吞吐量已被全部使用，余量不足以读取下一条数据。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。

表格存储计算读取，区间起始点到下一条未读数据的起始点的，所有行主键数据大小与实际读取的属性列数据大小之和。将数据大小之和除以 4 KB 向上取整计算消耗的读服务能力单元。例如，若读取范围中包含 10 行，每行主键数据大小与实际读取到的属性列数据之和占用数据大小为 330 Byte，则消耗的读服务能力单元为 1（数据总和 3.3 KB，除以 4 KB 向上取整为 1）。

示例

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；Attr1、Attr2 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
‘C’	9	‘Alpha’	不存在

示例 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")
}
}
```

示例 2，利用 INF_MIN 和 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")
}
}
```

示例 3，在某些主键列上使用 INF_MIN 和 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")
}
}
```

示例 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")
}
}
```

示例 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"}
}
}
```

示例 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")
}
```

```
}
```

示例 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")
}
}
```

示例 8，使用 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 1 是表的主键列，Attr1、Attr2 是表的属性列。

PK1	Attr1	Attr2
1	不存在	String(1000Byte)
2	8	String(1000Byte)
3	String(1000Byte)	不存在
4	String(1000Byte)	String(1000Byte)

```
//请求
table_name: "table2_name"
direction: FORWARD
inclusive_start_primary_key: ("PK1", INTEGER, 1)
exclusive_end_primary_key: ("PK1", INTEGER, 4)
columns_to_get: "PK1", "Attr1"

//响应
cosumed_read_capacity_unit: 1
rows: {
{
primary_key_columns:("PK1", INTEGER, 1)
},
{
primary_key_columns:("PK1", INTEGER, 2),
attribute_columns:("Attr1", INTEGER, 8)
},
{
primary_key_columns:("PK1", INTEGER, 3),
attribute_columns:("Attr1", STRING, String(1000Byte))
},
}
```

此次 GetRange 请求中：

获取的第一行数据大小为：len（‘PK1’）+ 8 Byte = 11 Byte

第二行数据大小为：len（‘PK1’）+ 8 Byte + len（‘Attr1’）+ 8 Byte = 24 Byte

第三行数据大小为：len（‘PK1’）+ 8 Byte + len（‘Attr1’）+ 1000 Byte = 1016 Byte

消耗的读服务能力单元为获取的三行数据之和 11 Byte + 24 Byte + 1016 Byte = 1051 Byte 除以 4 KB 向上取整，该 GetRange 操作消耗 1 个单位的读服务能力单元。

更多详细信息请参见 API Reference 中的 GetRange 章节。

最佳实践

表格存储数据操作的最佳实践

使用表格存储 SDK 进行数据操作

使用 TableStore Java SDK 进行数据操作

使用 TableStore Python SDK 进行数据操作

应用程序可以使用阿里云官方发布的表格存储 SDK 来访问表格存储，也可以通过 POST 方法发送 HTTP 请求来访问表格存储。

以下将介绍 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。最后介绍表格存储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 以外的语言访问表格存储，可以根据本章内容使用 HTTP 消息与表格存储进行交互，也可以自行编写 SDK。

当前表格存储的 API 版本

API Version: 2014-08-08

HTTP 消息

表格存储接收应用程序的 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，请参见表格存储 ProtocolBuffer 消息定义。下面分别介绍 HTTP 请求的 header 和 body， 以及响应的具体格式。

HTTP Request

HTTP Request URL

访问表格存储的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，可以在表格存储控制台查看云账户下拥有的实例信息。大小写不敏感。

region：阿里云服务地域。表格存储服务会部署在多个地理位置不同的阿里云服务地域内。创建实例时需要制定阿里云地域，可以在表格存储控制台查看实例所在的阿里云服务地域名称。大小写不敏感。

operation：表格存储操作名称。可以在 API 操作总览一章查看所有的表格存储操作名。大小写敏感。

如下所示的 URL 是华东 1（杭州）地域，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

表格存储规定 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'
```

上述伪代码中使用到的函数的说明：

HmacSha1：Hmac-Sha1 加密算法。计算表格存储请求签名时请将 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。如下面示例中，CanonicalURI 为 /ListTable：

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

HTTPRequestMethod：HTTP 请求方法。如 GET、POST 或 PUT 等，表格存储的 HTTP API 只支持 POST 方法。注意 POST 需要大写。

CanonicalHeaders：CanonicalHeaders 是表格存储 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-”开头的表格存储标准头。

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。

表格存储会对 HTTP 请求进行以下验证：

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，表格存储会直接返回身份认证错误。

HTTP Request Body

表格存储规定 HTTP 请求的 Body 部分是表格存储定义的 ProtocolBuffer 消息序列化之后的字符串，Body 长度不超过 2 MB。

关于表格存储请求的 ProtocolBuffer 消息定义，请参见 TableStore ProtocolBuffer 消息定义。

HTTP Response

HTTP Response Header

表格存储规定 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。只有请求的签名被表格存储验证通过的条件下，响应才会包含签名。签名计算方式如下:

```
Authorization = 'OTS ' + AccessKeyID + ':' + base64(HmacSha1(AccessKeySecret, stringToSign))

StringToSign = CanonicalHeaders + CanonicalURI

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

上述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。如下面示例中，CanonicalURI 为 /ListTable：

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

CanonicalHeaders：CanonicalHeaders 是表格存储 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-”开头的表格存储标准头。

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。

客户端应该对表格存储响应进行以下验证：

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，该响应有可能不是表格存储服务返回的。

HTTP Response Content

表格存储规定 HTTP 响应的内容是表格存储定义的 ProtocolBuffer 消息序列化之后的字符串，Body 长度不超过 2 MB。每一个表格存储请求消息对应一个表格存储响应消息，应用程序将响应内容反序列化之后，读取表格存储操作的结果。

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
```

那么表格存储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-db74-4a34-ce0a5df229a1\n/ListTable'

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

本节列举了表格存储的 API 可能出现的错误类型、描述信息与 HTTP 状态码。

以下列表中列出了表格存储可能返回的错误信息。其中部分错误可以通过重试解决，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
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	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。	否
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。如果出现该错误，请提交工单。	是
403	OTSQuotaExhausted	Too frequent table operations.	执行表相关的操作过于频繁。	是
403	OTSQuotaExhausted	Number of tables exceeded the quota.	表的数量超过定额。	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	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 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 Current Length:{CellSize}.	属性列名长度超出命名最大长度限制。	否
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	Duplicated primary key name: '{PKName}'.	主键重复。	否
400	OTSParameterInvalid	The limit must be greater than	limit 参数必须大于 0。	否

		0.		
400	OTSParameterInvalid	Duplicated attribute column name with primary key column: {ColumnName}.	对某行进行操作时，包含与主键列同名的属性列。	否
400	OTSParameterInvalid	Rows count exceeds the upper limit:{limit}.	这次写入的行总数超过了最大限制：limit。	否
400	OTSParameterInvalid	No version condition is specified while querying row.	查询的时候没有指定版本数。	否
400	OTSParameterInvalid	No row is specified in BatchWriteRow	BatchWriteRow操作中没有行数据，目前BatchWriteRow必须至少包含一行。	否
400	OTSParameterInvalid	No operation is specified for table:{TableName}	BatchWriteRow操作中对于表TableName 没有包含任何操作。	否
400	OTSParameterInvalid	The type of row in batch write operation is invalid	BatchWriteRow操作中的行类型不合法。	否
400	OTSParameterInvalid	The modify type is invalid	BatchWriteRow操作中的类型不对，目前仅支持PUT、UPDATE和 DELETE。	否
400	OTSParameterInvalid	The total data size of BatchWriteRow request exceeds the limit, limit size: {LimitSize}, data size:{UserSize}	一次BatchWriteRow操作中的总大小超过了限制。	否
400	OTSParameterInvalid	Time-to-live is missing while creating table	创建表的时候没有指定 TTL 值。	否
400	OTSParameterInvalid	MaxVersions is missing while creating table	创建表的时候没有指定最大版本数。	否
400	OTSParameterI	Start and end	GetRange 操作	否

	nvalid	time must be given at the same time	中需要同时指定 StartTime 和 EndTime。	
400	OTSParameterInvalid	Invalid column type, only STRING,INTEGER,BINARY is allowed.	列类型错误，目前主键列仅支持 STRING、INTEGER 和 BINARY。	否
400	OTSParameterInvalid	Invalid column type: {ColumnType}	列类型不合法。	否
400	OTSParameterInvalid	Invalid return type: {ReturnType}	返回类型不合法。	否
400	OTSParameterInvalid	Invalid condition:{Condition}	指定的条件不合法。	否
400	OTSParameterInvalid	The value of read capacity unit can not be less than {Limit}	读 CU 值不能小于 Limit。	否
400	OTSParameterInvalid	The value of write capacity unit can not be less then {Limit}	写 CU 值不能小于 Limit。	否
400	OTSParameterInvalid	first primary key can’t be AUTO_INCREMENT	第一个主键不能设置为自增列。	否
400	OTSParameterInvalid	AUTO_INCREMENT primary key must be integer	只有整数列可以被设置为自增列。	否
400	OTSParameterInvalid	AUTO_INCREMENT primary key count must <= 1	最多只能设置一个自增列。	否
400	OTSParameterInvalid	Column value cannot be given when type is DELETE_ONE_VERSION,DELETE_ALL_VERSION	当使用了 DELETE_ONE_VERSION 或 DELETE_ALL_VERSION 等删除操作时，不能设置属性值。	否
400	OTSParameterInvalid	Timestamp must be given when type is DELETE_ONE_VERSION	当删除一个版本时必须设置版本号。	否

400	OTSParameterInvalid	Timestamp cannot be given when type is DELETE_ALL_VERSION	当删除所有版本时不能设置版本号。	否
400	OTSParameterInvalid	No name is given for attribute column	属性列缺少名字。	否
400	OTSParameterInvalid	No value is given for column name	属性列缺少值。	否
400	OTSParameterInvalid	OpType cannot be given for column name:{ColumnName} in PutRow	PutRow 时不能指定操作类型，只有 UpdateRow 才需要指定操作类型。	否
400	OTSParameterInvalid	Attribute column is missing	缺少属性列。	否
400	OTSParameterInvalid	The number of attribute columns exceeds the limit, limit count:{Limit} , column count: {Count}	列的个数超过了 Limit 限制。	否
400	OTSParameterInvalid	The length of primary key column: {ColumnName} exceeds the MaxLength: {Limit} with CurrentLength: {Current}	主键的长度超过了限制。	否
400	OTSParameterInvalid	No name is given for primary key	主键列中没有设置名字。	否
400	OTSParameterInvalid	No value is given for primary key name : {PkName}	主键列没有设置值。	否
400	OTSParameterInvalid	OpType cannot be given for primary key name	主键中不能指定操作类型。	否

		：{PkName}		
400	OTSParameterInvalid	Timestamp cannot be given for primary key name：{PkName}	主键中不能指定时间戳。	否
400	OTSParameterInvalid	Duplicated primary key name: {PkName}	主键重复。	否
400	OTSParameterInvalid	Attribute column cannot be given while reading data	读取数据的时候不需要指定列值。	否
400	OTSParameterInvalid	Delete marker cannot be given	非 Delete 时都不能指定 DeleteMarker。	否
400	OTSParameterInvalid	The number of columns from the request exceeds the limit, limit count: {Limit} , column count:{current}	请求的列的个数超过限制。	否
400	OTSParameterInvalid	Timestamp must be in range [0, INT64_MAX/1000)	时间戳必须大于等于 0，小于 INT64_MAX/1000。	否
400	OTSParameterInvalid	TimeToLive cannot be 0 or less than -1	TTL 不能设置为 0 或者小于 -1 的值，但可以设置为 -1。	否
400	OTSParameterInvalid	The maximum versions cannot be less than or equal to 0	最大版本数不能小于等于 0。	否
400	OTSParameterInvalid	Specific timestamp cannot be less than 0	指定的时间戳不能小于 0。	否
400	OTSParameterInvalid	The maximum deviation must be in range [0, INT64_MAX/1000000]	最大版本偏差的值必须位于整个范围内。	否
400	OTSParameterInvalid	Deserialize filter failed	反序列化 filter 失败，原因是	否

			SDK 组装的 filter 格式不对。	
400	OTSParameterInvalid	Offset in ColumnPaginationFilter must be greater than or equal to 0	ColumnPaginationFilter 过滤器的 offset 必须大于等于 0。	否
400	OTSParameterInvalid	Limit in ColumnPaginationFilter must be greater than 0	ColumnPaginationFilter 过滤器的 Limit 必须大于等于 0。	否
400	OTSParameterInvalid	Deserialize relation filter failed	反序列化单列过滤器失败，原因可能是 SDK 中组装的 filter 格式不对。	否
400	OTSParameterInvalid	Deserialize composite filter failed	反序列化组合过滤器失败，原因可能是 SDK 中组装的 filter 格式不对。	否
400	OTSParameterInvalid	Deserialize column pagination filter failed.	反序列化宽行过滤器失败，原因可能是 SDK 中组装的 filter 格式不对。	否
400	OTSParameterInvalid	The count of filter exceeds the max：{Limit}	过滤器的个数超过了限制。	否
400	OTSParameterInvalid	Specific tiemstamp and max versions cannot be given at the same time	单个时间戳和最大版本数不能同时存在，只能存在一个。	否
400	OTSParameterInvalid	Specific tiemstamp and time range cannot be given at the same time	单个时间戳和时间戳范围不能同时存在，只能存在一个。	否
400	OTSParameterInvalid	Specific tiemstamp, time range and max versions cannot be given at the same time	单个时间戳、时间戳范围和最大版本数不能同时存在，只能存在一个。	否
400	OTSParameterI	Duplicated	属性列重复。	否

	nvalid	attribute column name with primary key column:{PkName}		
400	OTSParameterInvalid	The total data size of PutRow request exceeds the limit, limit size: {Limit} , data size: {Current}	PutRow 操作的数据大小超过限制。	否
400	OTSParameterInvalid	The total data size of UpdateRow request exceeds the limit, limit size: {Limit} , data size: {Current}	UpdateRow 操作的数据大小超过限制。	否
400	OTSParameterInvalid	No parameter is specified in table options	TableOption 中没有指定任何参数。	否
400	OTSParameterInvalid	Input encoded PK broken, invalidate cell key type:xx	编码的 PlainBuffer 不合法，cell 类型错误。	否
400	OTSParameterInvalid	Input encoded PK broken, length is shorter than expect	编码的 PlainBuffer 不合法，长度和预期不一致。	否
400	OTSParameterInvalid	PKAutoIncr can’t be used for read operations	读操作时不能指定自增属性。	否
400	OTSParameterInvalid	Begin key must less than end key in FORWARD	前向查询中，开始键必须小于结束键。	否
400	OTSParameterInvalid	Begin key must more than end key in BACKWARD	反向查询中，开始键必须大于结束键。	否
400	OTSParameterInvalid	Repeated rows do not support row exist expectation check in BatchModify	BatchWriteRow 中重复的行操作不支持行存在条件检查。	否

400	OTSParameterInvalid	Repeated rows do not support row condition check in BatchModify	BatchWriteRow操作中不允许重复行存在条件更新。	否
400	OTSParameterInvalid	PK column size not the same for all rows in table: {TableName}	不同行中的主键的个数不一致。	否
400	OTSParameterInvalid	Duplicate rows detected in MultiPut	存在重复行。	否
400	OTSParameterInvalid	Duplicate rows detected in MultiGet	存在重复行。	否
400	OTSParameterInvalid	Column does not exist:{ColumnName}	列不存在。	否
400	OTSParameterInvalid	Invalid max scan size: {Limit}	范围查询的 limit 超过限制。	否
400	OTSParameterInvalid	data format is invalid	PlainBuffer 编码的行数据不合法。	否
400	OTSParameterInvalid	double can not be used as primary key	主键中不能使用double。	否
400	OTSParameterInvalid	data format is invalid, unknown variant type occured	PlainBuffer 编码的行数据不合法，出现了非法的值类型。	否
400	OTSParameterInvalid	Parse PBMessage from RawString failed	反序列化 protobuf 失败，可能是 SDK 序列化代码错误。	否
400	OTSParameterInvalid	Cell data broken, mismatch header, actual: {Header} , expect: {Header}	PlainBuffer 编码的行数据不合法，预期的 Header 不一致。	否
400	OTSParameterInvalid	Cell data broken, PK value cannot be NULL, name: {PkName}	PlainBuffer 编码的行数据不合法，PkName 对应的主键的值为空。	否

400	OTSParameterInvalid	Cell data broken, PK is empty	PlainBuffer 编码的行数据不合法，缺少主键。	否
400	OTSParameterInvalid	Cell data broken, no PK follow PKTag	PlainBuffer 编码的行数据不合法，主键标志后面没有按预期出现主键内容。	否
400	OTSParameterInvalid	Cell data broken, attr has no name	PlainBuffer 编码的行数据不合法，属性列缺少名字。	否
400	OTSParameterInvalid	Cell data broken, attr has no value, name:{Name}	PlainBuffer 编码的行数据不合法，属性列缺少值。	否
400	OTSParameterInvalid	Cell data broken, no Cells follow CellTag	PlainBuffer 编码的行数据不合法，Cell 标志后没有按预期出现 Cell 内容。	否
400	OTSParameterInvalid	Cell data broken, no valid element in Cell, name: {Name}	PlainBuffer 编码的行数据不合法，Cell 格式混乱。	否
400	OTSParameterInvalid	Cell data broken, cell is not end with checksum[{0x0A}]”] tag.	PlainBuffer 编码的行数据不合法，Cell 编码结束时没有按照预期出现 CheckSumFlag。	否
400	OTSParameterInvalid	Cell data broken, row is not end with checksum tag	PlainBuffer 编码的行数据不合法，行数据编码结束时没按照预期出现 CheckSumTag。	否
400	OTSParameterInvalid	Cell data broken, mismatch tag, actual tag({TAG}), expect({TAG})	PlainBuffer 编码的行数据不合法，预期的 Tag 没有出现。	否
400	OTSParameterInvalid	Cell data broken, wrong string format, size: {Size}	PlainBuffer 编码的行数据不合法，字符串编码格式错误。	否
400	OTSParameterI	Invalid Capacity	读写 CU 都应该	否

	nvalid	Unit for UpdateTable: Either read({CU}) or write({CU}) should be non-negative	大于等于 0。	
400	OTSParameterInvalid	Auto increment pk must be integer, name: {PkName}	自增列的类型必须是整数。	否
400	OTSParameterInvalid	Cannot deserialize the request data	无法反序列化请求中的数据，原因可能是空值或格式不合法。	否
400	OTSParameterInvalid	Invalid row: missing checksum	PlainBuffer 编码的行数据不合法，缺少行的 checksum 值。	否
400	OTSParameterInvalid	Invalid Column({ColumnName}): missing checksum.	PlainBuffer 编码的列数据不合法，缺少 Cell 的 checksum 值。	否
400	OTSParameterInvalid	Cell data broken, checksum is mismatch, {UserChecksum} ：{SeviceChecksum} ：	PlainBuffer 编码的行数据不合法，用户的 UserChecksum 和服务器计算的 SeviceChecksum 不一致。	否
400	OTSParameterInvalid	Cell data broken, has more data in cell, but they can’t be parsed	PlainBuffer 编码的列数据不合法，存在非预期的过多数据存在。	否
400	OTSParameterInvalid	NaN can’t be set to double value	Double 类型不能设置为 NaN 值。	否
400	OTSParameterInvalid	Infinity can’t be set to double value	Double 类型不能设置为 Infinity 值。	否
400	OTSParameterInvalid	Invalid max versions: {Version} . Reason: Max versions must be positive	最大版本数不合法，应该大于 0 或者等于 -1。	否
400	OTSParameterI	invalid range,	范围查询中的起	否

	nvalid	the begin’s type is different from the end’s, {BeginType} : {EndType}	始键类型和结束键类型不一致。	
400	OTSParameterInvalid	Invalid offset of Filter: {Filter} . Reason: Offset must be nonnegative.	过滤器中的 offset 必须大于等于 0。	否
400	OTSParameterInvalid	Invalid Capacity Unit:{CU} . Capacity Unit must be nonnegative.	CU 值必须大于等于 0。	否
400	OTSParameterInvalid	Unknown or unsupported CellType in primary key: {PkName}	主键的 Cell 类型无法识别。	否
400	OTSParameterInvalid	Invalid boolean real value, value length: {Length}	PlainBuffer 编码的列数据不合法，Bool 类型的值长度不合法。	否
400	OTSParameterInvalid	invalid name of column: empty name	属性列的名字不能为空。	否
400	OTSParameterInvalid	Invalid ModifyType:{ModifyType}	更新类型不合法。	否
400	OTSParameterInvalid	Invalid relation operator: {Operation}	关系运算符不合法。	否
400	OTSParameterInvalid	Invalid type of filter: {Type}	过滤器的类型不合法。	否
400	OTSParameterInvalid	Invalid NOT operator: the number of sub-filters must be 1	NOT 运算符只允许有一个子过滤器。	否
400	OTSParameterInvalid	Invalid AND/OR operator: the number of sub-filters must be 2	AND 和 OR 运算符都只需要有两个子过滤器。	否
400	OTSParameterInvalid	Invalid type of sub-filter: {Filter}	子过滤器的类型不合法。	否

400	OTSParameterInvalid	{Count} pk auto increment exist in row	多个 PK 自增列存在。	否
400	OTSParameterInvalid	Invalid action: {Action}	非法的操作类型。	否
400	OTSParameterInvalid	Invalid row-existence expectation: {Expectation}	指定的行存在性值不合法。	否
400	OTSParameterInvalid	Invalid row to delete which with pk auto increment	删除时不能设置自增列属性。	否
400	OTSParameterInvalid	Invalid expect:{RowExpect} when modify row with pk auto increment	使用主键列自增时，指定的行存在性值不合法。	否
400	OTSParameterInvalid	Can’t set condition when modify row with pk auto increment	使用主键列自增时，不能设置 ConditionUpdate。	否
400	OTSParameterInvalid	Invalid expect: {RowExpect} when modify a specific row for table with pk auto increment	修改主键列自增表的特定一行时，指定的行存在性条件不合法。	否
400	OTSParameterInvalid	Invalid cell: {Cell} . missing timestamp	缺少时间戳。	否
400	OTSParameterInvalid	Invalid cell timestamp: {Timestamp} . Expect: [0, Version::kMax. mVersion / kUsecPerMsec]	指定的时间戳不合法。	否
400	OTSParameterInvalid	Invalid timestamp for cell: {Timestamp} . timestamp is inapplicable to DELETE_ALL_VERSION	当使用 DELETE_ALL_VERSION 删除所有版本时不能指定版本。	否
400	OTSParameterInvalid	Invalid op type of cell: {OpType}	非法的操作类型。	否

400	OTSParameterInvalid	Invalid cell: {Cell}. Reason: missing value	Cell 非法，缺少值。	否
400	OTSParameterInvalid	Invalid request of delete row: missing RowDeleteMarker in request	删除请求中没有包含 RowDeleteMarker。	否
400	OTSParameterInvalid	Invalid request of put/update row: unexpected RowDeleteMarker in request.	PutRow 和 UpdateRow 不能包含 RowDeleteMarker。	否
400	OTSParameterInvalid	Invalid update row request: missing cells in request	UpdateRow 中缺少属性列值。	否
400	OTSParameterInvalid	Invalid delete row request: unexpected cells in request	DeleteRow 中不应该出现属性列的值。	否
400	OTSParameterInvalid	Invalid request of put row: find cells without values	PutRow 中属性列缺少值。	否
400	OTSParameterInvalid	Can not reserve read capacity unit on hybrid storage cluster: {TableName}	混合存储类型实例不能设置预留读 CU。	否
400	OTSParameterInvalid	Can not reserve write capacity unit on hybrid storage cluster: {TableName}	混合存储类型实例不能设置预留写 CU。	否
400	OTSParameterInvalid	First primary key can’ t be auto increment, table: {TableName}	第一个主键不能设置为自增列。	否
400	OTSParameterInvalid	the count of auto-incremental primary keys can not be more than 1	自增列的个数不能超过 1。	否

表格存储存储相关异常

HTTPStatus	ErrorCode	ErrorMsg	描述	重试
503	OTSServerBusy	Server is busy.	TableStore 内部服务器繁忙，稍后重试即可成功。	是
503	OTSPartitionUnavailable	The partition is not available.	一般是分区加载中，稍后重试即可成功。	是
503	OTSTimeout	Operation timeout.	在 TableStore 内部操作超时。	是
503	OTSServerUnavailable	Server is not available.	在 TableStore 内部有服务器不可访问。	是
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.	表新创建后还在做初始化，稍等几秒钟即可使用。	是
403	OTSTooFrequentReservedThroughputAdjustment	Capacity unit adjustment is too frequent.	读写能力调整过于频繁。	是
403	OTSCapacityUnitExhausted	Capacity unit is not enough.	该数据分区读写服务能力耗尽。	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

Stream

Table Store Stream 是一个数据通道，用于获取 Table Store 表中的增量数据。您可以使用 Table Store Stream API 来获取这些修改内容。增量数据的重要性不言而喻，有了增量数据，可以进行增量数据流的实时增量分析、数据增量同步等。

原理

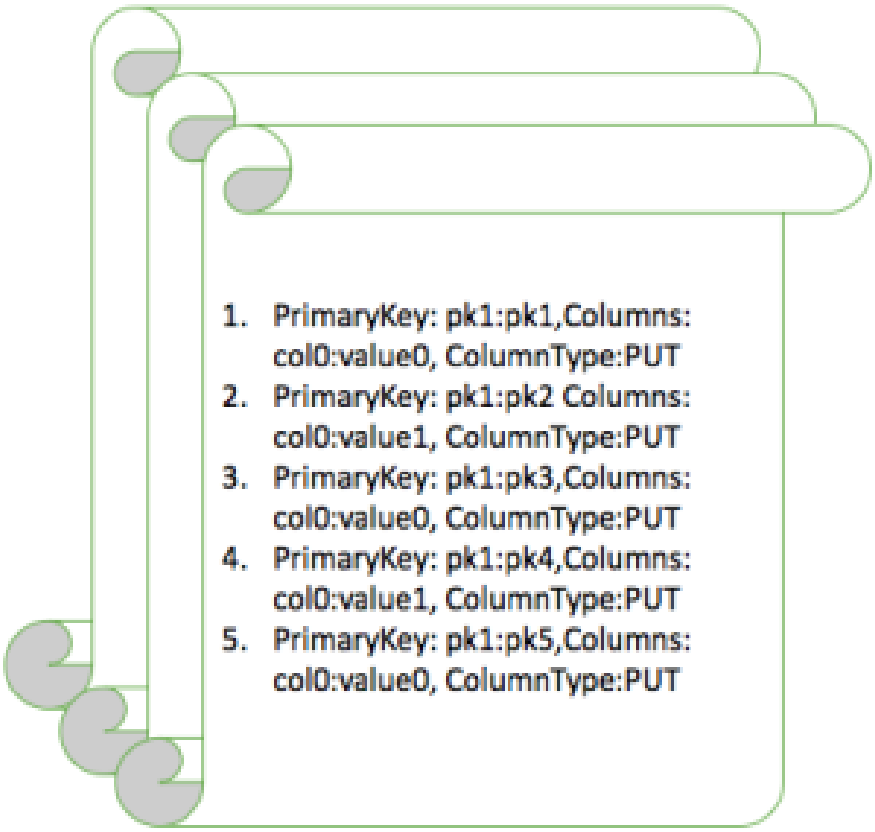
表格存储作为一款分布式 NoSQL 数据库，当有写操作（包括 put，delete，update）传入时，会把对应的修改记录存放在表格存储的 commit log 中，同时数据库也会定期做 checkpoint，旧的 commit 日志会被清除。

开启 Stream后，日志文件会被保存。在设置的保存期限内，可以通过 Stream 提供的通道读取这些增量数据。

由于表格存储的分区特性，同一分区的操作会共享一个 commit log，所以获取增量数据也是按照分区维度获取。

开启 Stream 时，会生成一个当前日志偏移量（即 iterator）并记录下来。用户可以通过 GetShardIterator 接口获取当前分区的 iterator，在后续读取该分区增量数据时传入这个 iterator，Stream 通道就可以知道从哪一行日志记录开始返回增量内容。返回增量内容的同时，Stream 也会返回一个新的偏移量，用于后续的读取。整个过程可以类比我们分页读取数据，iterator 就是分页的偏移量。

例如，我们有顺序地生成一批数据库日志文件，如下所示：



当我们在文件 A，第 3 行开始开启 Stream，则这个 iterator 就可以用来标识文件 A 的第 3 行。读取数据时传入这个 iterator，就能读到从上图中第三个操作 pk3 开始的后续修改操作。

Stream API 也提供了接口关闭这个数据通道。当用户再次开启时，Stream 会根据本次开启时间的日志偏移量重新生成一个当前分区的 iterator，我们可以用这个 iterator 读取该分区当前时间点之后的增量数据。

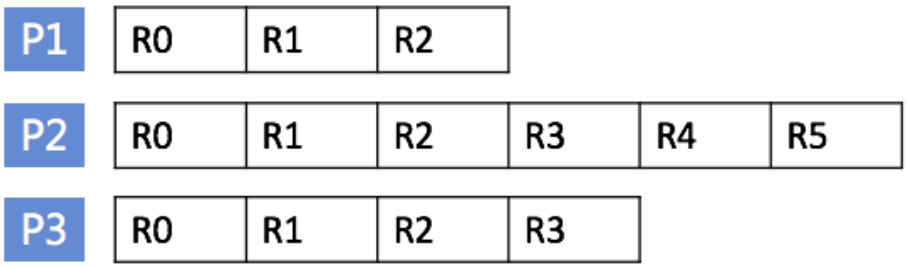
由于写操作会发生在同一个主键上，为了确保数据的一致性，对同一个主键的写操作需要顺序读取。然而，在读取增量数据之前，我们并不知道在哪些主键上发生了修改，所以读取增量数据的接口按照分区 id 来划分。用户可以通过罗列当前表下的所有分区来读取整张表的增量数据。Stream 通道会保证同分区内的写操作是顺序返回的，即，只要在同一分区内按照顺序读取，就可以保证相同主键的数据的一致性。如果持续读取所有分区的 Stream 数据，也就确保了整张表的增量数据都被会被读取到。

用户可以在创建表的时候开启 Stream，或者通过 UpdateTable 来开启或者关闭 Stream。当有一个 put，update 或者 delete 操作发生，一条修改记录会被写入 Stream，数据包含用户修改行的主键以及修改内容。

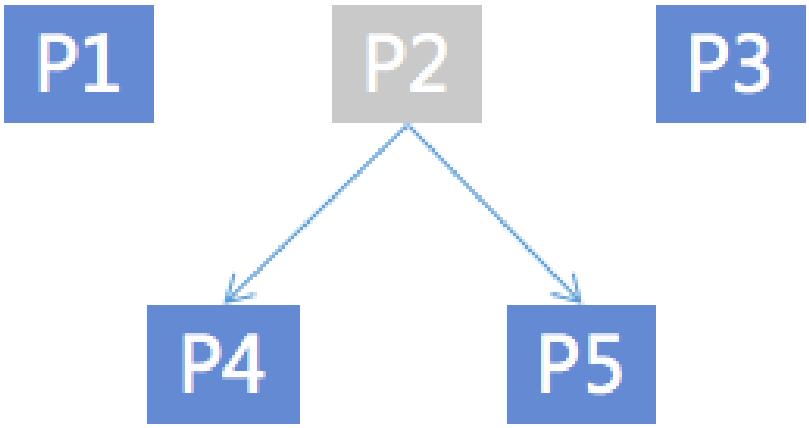
注意：

- 每个修改记录在 Stream 中存在且仅存在一次。
- 在每一个 shard 内，Stream 按照用户的实际操作顺序进行修改。但是不同 shard 的数据，不保证顺序。

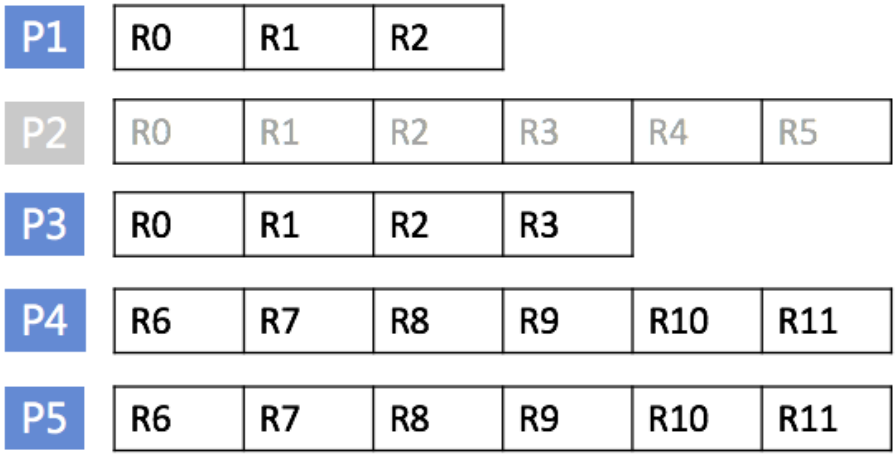
示例



如上图所示，当前表有三个分区。图中的每一行表示一个分区，每一列表示当前分区的一次更新操作。每个分区维护自己的更新记录。我们可以通过 DescribeStream 接口得到分区的信息，然后顺序读取各自分区的数据。但系统会根据负载进行分区的分裂或者合并。分区的合并、分裂操作会生成新的分区，老的分区也将不会再产生新的增量数据。



上图中，分区 P2 发生了一次分裂，变成分区 P4 和分区 P5。我们可以并行读取分区 P4 和分区 P5 的数据，并不互相影响。但是在开始读取分区 P4 和分区 P5 之前，需要确保分区 P2 的增量数据已经全部读取完毕。



例如上图中，当开始读取分区 P4 的记录 R6 时，需要保证分区 P2 的 R5 已经被读取完毕，当 R5 被读完后，分区 P2 不会再生成新的数据。

Stream API

打开/关闭 Stream

用户可以在创建表的时候设置 Stream 是否开启，也可以通过 UpdateTable 来开启或者关闭 Stream。CreateTable 和 UpdateTable 中新增了 StreamSpecification 参数，用来表示 Stream 的相关参数：

- enable_stream：Stream 是否打开
- expiration_time：Stream 数据的过期时间，较早的修改记录将会被删除。

读取修改记录

具体读取 Stream 数据的流程如下：

调用 ListStreams 获取当前表的 Stream 信息，例如 StreamID。具体请参见 ListStream API。

调用 DescribeStream 获取当前 Stream 的数据分片信息，例如 shard 的列表，每个 shard 记录又包含父 shard 信息、shardID 信息等。具体请参见 DescribeStream API。

获取 StreamID 和 shardID 后，通过 GetShardIterator 获取当前 shard 的读 iterator 值，这个值标记着读取该 shard 记录的起始位置。具体请参见 GetShardIterator API。

调用 GetStreamRecord 来读取具体的修改记录，每次调用会返回新的 iterator，用于下次读取。具体请参见 GetStreamRecord API。

注意：

- 因为需要保证同一个主键下的操作要有序，在同一个 shard 下，Stream 做了这个保证。但是 shard 会出现分裂、合并等操作，所以我们在读取某个 shard 的数据时，确保他的父 shard 以及 parent_sibling 的数据已经被读取。
- 当读到空的 NextShardIterator 的时候说明当前 shard 的增量数据已经全部读完，通常情况是该 shard 已经进入 inactive 的状态（分裂或者合并发生）。当一个 shard 已经被全部读完以后，我们可以重新调用 DescribeStream 获取新的 shard 信息。

SDK

为了方便用户使用 Stream API，TableStore 的 Java SDK 已经支持 Stream 接口，具体请参见 JAVA SDK 介绍。

API 参考

API 概览

单行数据操作：

- GetRow
- PutRow
- UpdateRow
- DeleteRow

多行数据操作：

- GetRange
- BatchGetRow
- BatchWriteRow

表操作：

- CreateTable
- ListTable
- DeleteTable
- UpdateTable
- DescribeTable

ComputeSplitPointsBySize

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;
required bytes primary_key = 2; // Plainbuffer编码为二进制
repeated string columns_to_get = 3; // 不指定则读出所有的列
optional TimeRange time_range = 4;
optional int32 max_versions = 5;
optional bytes filter = 7;
optional string start_column = 8;
optional string end_column = 9;
optional bytes token = 10;
}
```

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：bytes

是否必要参数：是。

该行全部的主键列，包含主键名和主键值，由Plainbuffer编码，详见Plainbuffer编码

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。

time_range:

类型：TimeRange。

是否必要参数：和max_versions只能存在一个。

读取数据的版本时间戳范围。

时间戳的取值最小值为0，最大值为INT64.MAX

若要查询一个范围，则指定start_time和end_time

若要查询一个特定时间戳，则指定specific_time

例子：如果指定的time_range为(100, 200)，则返回的列数据的时间戳必须位于[100, 200)范围内，前闭后开区间。

max_versions：

类型：int32。

是否必要参数：和time_range只能存在一个。

读取数据时，返回的最多版本个数。

例子：如果指定max_versions为2，则每一列最多返回2个版本的数据。

filter:

类型：bytes。

是否必要参数：否。

过滤条件表达式。

Filter 经过protobuf序列化后的二进制数据。

start_column：

类型：string。

是否必要参数：否。

指定读取时的起始列，主要用于宽行读。

返回的结果中包含当前起始列。

列的顺序按照列名的字典序排序。

例子：如果一张表有”a”，”b”，”c”三列，读取时指定start_column为“b”，则会从”b”列开始读，返回”b”，”c”两列。

end_column：

类型：string。

是否必要参数：否。

指定读取时的结束列，主要用于宽行读。

返回的结果中不包含当前结束列。

列的顺序按照列名的字典序排序。

例子：如果一张表有”a”，”b”，”c”三列，读取时指定end_column为“b”，则读到”b”列时会结束，返回”a”列。

响应消息结构：

```
message GetRowResponse {
required ConsumedCapacity consumed = 1;
required bytes row = 2; // Plainbuffer编码为二进制
}
```

consumed：

类型：CapacityUnit。

本次操作消耗的服务能力单元。

row：

类型:bytes。

读取到的数据，由Plainbuffer编码，详见Plainbuffer编码

如果该行不存在,则数据为空。

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所有主键列的数据大小与实际读取的属性列数据大小之和除以 4 KB 向上取整。关于数据大小的计算请参见购买指导。

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。

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 bytes row = 2; // Plainbuffer编码为二进制
required Condition condition = 3;
optional ReturnContent return_content = 4;
}
```

table_name：

类型：string。

是否必要参数：是。

请求写入数据的表名。

row

类型：bytes。

是否必要参数：是。

写入的行数据， 包括主键和属性列，Plainbuffer格式，编码详见Plainbuffer编码。

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。

return_content:

类型：ReturnContent。

是否必要参数：否。

写入成功后返回的数据类型，目前仅支持返回主键，主要用于主键列自增功能中。

响应消息结构：

```
message PutRowResponse {
required ConsumedCapacity consumed = 1;
optional bytes row = 2;
}
```

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：

若指定条件检查为 IGNORE，消耗写服务能力单元的数值为本行的主键数据大小与要插入属性列数据大小之和除以 4 KB 向上取整。

若指定条件检查为 EXPECT_NOT_EXIST，除了消耗本行的主键数据大小与要插入属性列数据大小之和除以 4 KB 向上取整的写CU，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。

若指定条件检查为 EXPECT_EXIST，本次插入失败并且消耗 1 单位写 CU 和 1 单位读 CU。

如果该行存在：

若指定条件检查为 IGNORE，消耗写服务能力单元的数值为本行的主键数据大小与要插入属性列数据大小之和除以 4 KB 向上取整。

若指定条件检查为 EXPECT_EXIST，除了消耗本行的主键数据大小与要插入属性列数据大小之和除以 4 KB 向上取整的写 CU，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。

若指定条件检查为 EXPECT_NOT_EXIST，本次插入失败并且消耗 1 单位写 CU 和 1 单位读 CU。

使用条件更新（Conditional Update）：

操作成功，按照上述消耗服务能力单元方式进行计算。

操作失败，则消耗 1 单位写 CU 和 1 单位读 CU。

关于数据大小的计算请参见购买指导。

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 个写服务能力单元。

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。

row：

类型：bytes。

当设置了return_content后，返回的数据。

如果没设置return_content或者没返回数据，此处为NULL。

Plainbuffer格式，编码详见Plainbuffer编码

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 bytes row_change = 2;
required Condition condition = 3;
optional ReturnContent return_content = 4;
}
```

table_name：

类型： string。

是否必要参数： 是。

请求更新数据的表名。

row_change：

类型：bytes。

是否必要参数： 是。

更新的数据，包括主键和属性列，Plainbuffer格式，编码详见Plainbuffer编码。

该行本次想要更新的全部属性列，表格存储会根据 row_change中UpdateType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 row_change 中的列将不受影响。

UpdateType 可以取下面两个值：

PUT：此时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 value 必须为空，需要指定timestamp。语意为删除该列特定版本的数据。

DELETE_ALL：此时该 value 和timestamp 都必须为空。语意为删除该列所有版本的数据。

注意：删除本行的全部属性列不等同于删除本行，若想删除本行，请使用 DeleteRow 操作。

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。

return_content:

类型：ReturnContent

是否必要参数：否。

写入成功后返回的数据类型，目前仅支持返回主键，主要用于主键列自增功能中。

响应消息结构：

```
message UpdateRowResponse {
required ConsumedCapacity consumed = 1;
optional bytes row = 2;
}
```

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：

若指定条件检查为 IGNORE，消耗写服务能力单元的数值为本行的主键数据大小与要更新的属性列数据大小之和除以 4 KB 向上取整。如果 UpdateRow 中包含有需要删除的属性列，只有其列名长度计入该属性列数据大小。

若指定条件检查为 EXPECT_EXIST，本次插入失败并且消耗 1 单位写 CU 和 1 单位读 CU。

如果该行存在：

若指定条件检查为 IGNORE，消耗写服务能力单元的数值为本行的主键数据大小与要更新的属性列数据大小之和除以 4 KB 向上取整。如果 UpdateRow 中包含有需要删除的属性列，只有其列名长度计入该属性列数据大小。

若指定条件检查为 EXPECT_EXIST，除了需要消耗在条件检查为 IGNORE 情况下的写 CU，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。

关于数据大小的计算请参见购买指导。

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 个写服务能力单元和 1 个读服务能力单元。

row：

类型：bytes。

当设置了return_content后，返回的数据。

如果没设置return_content或者没返回值，此处为NULL。

Plainbuffer格式，编码详见Plainbuffer编码

行为：

删除一行数据。

请求消息结构：

```
message DeleteRowRequest {
required string table_name = 1;
required bytes primary_key = 2; // Plainbuffer编码为二进制
required Condition condition = 3;
optional ReturnContent return_content = 4;
}
```

table_name：

类型：string。

是否必要参数：是。

请求更新数据的表名。

primary_key

类型：bytes。

是否必要参数：是。

删除行的主键，Plainbuffer格式，编码详见Plainbuffer编码

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。

return_content:

类型：ReturnContent

是否必要参数：否。

写入成功后返回的数据类型，目前仅支持返回主键，主要用于主键列自增功能中。

响应消息结构：

```
message DeleteRowResponse {
required ConsumedCapacity consumed = 1;
optional bytes row = 2;
}
```

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：

若指定条件检查为 IGNORE，消耗写服务能力单元的数值为该行主键数据大小除以 4 KB 向上取整。

若指定条件检查为 EXPECT_EXIST，删除该行失败，消耗 1 单位的写 CU 和 1 单位的读 CU。

如果该行存在：

若指定条件检查为 IGNORE，消耗写服务能力单元的数值为该行主键数据大小除以 4 KB 向上取整。

若指定条件检查为 EXPECT_EXIST，除了消耗该行主键数据大小除以 4 KB 向上取整的写 CU，还需消耗该行主键数据大小除以 4 KB 向上取整的读 CU。

关于数据大小的计算请参见购买指导。

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 个写服务能力单元。

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。

row：

类型：bytes。

当设置了return_content后，返回的数据。

如果没设置return_content或者没返回值，此处为NULL。

Plainbuffer格式，编码详见Plainbuffer编码

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; // 不指定则读出所有的列
optional TimeRange time_range = 4;
optional int32 max_versions = 5;
optional int32 limit = 6;
required bytes inclusive_start_primary_key = 7; // Plainbuffer编码为二进制
required bytes exclusive_end_primary_key = 8; // Plainbuffer编码为二进制
optional bytes filter = 10;
optional string start_column = 11;
optional string end_column = 12;
}
```

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。

time_range:

类型：TimeRange。

是否必要参数：和max_versions只能存在一个。

读取数据的版本时间戳范围。

时间戳的取值最小值为0，最大值为INT64.MAX。

若要查询一个范围，则指定start_time和end_time。

若要查询一个特定时间戳，则指定specific_time。

例子：如果指定的time_range为(100, 200)，则返回的列数据的时间戳必须位于[100, 200)范围内，前闭后开区间。

max_versions：

类型：int32。

是否必要参数：和time_range只能存在一个。

读取数据时，返回的最多版本个数。

例子：如果指定max_versions为2，则每一列最多返回2个版本的数据。

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，表格存储最多返回行数为 5000 且总数据大小不超过 1 M 。

inclusive_start_primary_key：

类型: bytes。

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。由Plainbuffer编码，详见Plainbuffer编码。

exclusive_end_primary_key：

类型：bytes。

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。由Plainbuffer编码，详见Plainbuffer编码。

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。

filter:

类型：bytes。

是否必要参数：否。

过滤条件表达式。

Filter 经过protobuf序列化后的二进制数据。

start_column：

类型：string。

是否必要参数：否。

指定读取时的起始列，主要用于宽行读。

返回的结果中包含当前起始列。

列的顺序按照列名的字典序排序。

例子：如果一张表有”a”，”b”，”c”三列，读取时指定start_column为“b”，则会从”b”列开始读，返回”b”，”c”两列。

end_column：

类型：string。

是否必要参数：否。

指定读取时的结束列，主要用于宽行读。

返回的结果中不包含当前结束列。

列的顺序按照列名的字典序排序。

例子：如果一张表有”a”，”b”，”c”三列，读取时指定end_column为“b”，则读到”b”列时会结束，返回”a”列。

响应消息结构：

```
message GetRangeResponse {
required ConsumedCapacity consumed = 1;
required bytes rows = 2;
```

```
optional bytes next_start_primary_key = 3;
}
```

consumed：

类型：ConsumedCapacity。

本次操作消耗的服务能力单元。

rows：

类型：bytes。

由Plainbuffer编码，详见Plainbuffer编码。

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。

next_start_primary_key：

类型：bytes。

本次 GetRange 操作的断点信息。由Plainbuffer编码，详见Plainbuffer编码。

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：表格存储系统中限制了 GetRange 操作的响应消息中数据不超过 5000 行，大小不超过 4 M，并且返回的数据量不超过当前剩余的预留读吞吐量。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。

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主键数据大小与实际读取的属性列数据大小之和除以 4 KB 向上取整。关于数据大小的计算请参见购买指导。

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 个读服务能力单元。

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 100 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 操作独立计算写服务能力单元，GetRow服务能力单元消耗。

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请求。

tables 中所有表总的行操作个数超过 200 个，或者其含有的总数据大小超过 4 M。

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 1024 个。

tables 中任一 UpdateRowInBatchWriteRowRequest 包含的 ColumnUpdate 个数超过 1024 个。

响应消息结构：

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

tables：

类型：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。PutRow 服务能力单元消耗、UpdateRow 服务能力单元消耗、DeleteRow 服务能力单元消耗。

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;
optional TableOptions table_options = 3;
repeated PartitionRange partitions = 4;
optional StreamSpecification stream_spec = 5;
```

```
}
```

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 应符合表名命名规范，type 取值只能为 STRING， INTEGER或BINARY。

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，任何表的预留读吞吐量与预留写吞吐量均不能超过 5000。

表的预留读/写吞吐量设定可以通过 UpdateTable 进行动态更改。

table_options：

类型：TableOptions。

是否必要参数：是。

主要设置TimeToLive和最大版本数。

StreamSpecification

- 类型: StreamSpecification
- 是否必要参数：否。

- 描述是否打开Stream等Stream相关的属性。

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 64 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。

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。

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。

行为：

更新指定表的预留读吞吐量或预留写吞吐量设置，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;
optional ReservedThroughput reserved_throughput = 2;
optional TableOptions table_options = 3;
optional StreamSpecification stream_spec = 4;
}
```

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。

可以只更改表的预留读吞吐量的设置或只更改表的预留写吞吐量的设置，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。

table_options：

类型：TableOptions。

是否必要参数：是。

主要设置TimeToLive和最大版本数。

StreamSpecification

- 类型: StreamSpecification

- 是否必要参数：否。
- 描述是否打开Stream等Stream相关的属性。

响应消息结构：

```
message UpdateTableResponse {
required ReservedThroughputDetails reserved_throughput_details = 1;
required TableOptions table_options = 2;
}
```

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。

注意事项：

调整每个表预留读/写吞吐量的最小时间间隔为 2 分钟，如果本次 UpdateTable 操作距上次不到 2 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）内每个表上调和下调预留读写吞吐量次数不限。

table_options：

类型：TableOptions。

修改后，最新的table_options参数值。

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;
required TableOptions table_options = 3;
optional StreamDetails stream_details = 5;
repeated bytes shard_splits = 6;
}
```

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

table_options：

类型：TableOptions。

当前最新的table_options参数值。

StreamSpecification

- 类型: StreamSpecification
- 是否必要参数：否。
- 描述是否打开Stream等Stream相关的属性。

shard_splits：

类型：bytes。

当前表所有分区的分裂点。

行为

将全表的数据在逻辑上划分成接近指定大小的若干分片，返回这些分片之间的分割点以及分片所在机器的提示。一般用于计算引擎规划并发度等执行计划。

请求结构

```
message ComputeSplitPointsBySizeRequest {
required string table_name = 1;
required int64 split_size = 2; // in 100MB
}
```

table_name：

- 类型：string。
- 是否必要参数：是。
- 要切分的数据所在的表名。

split_size:

- 类型：int64

- 是否必要参数：是。
- 每个分片的近似大小，以百兆为单位。

响应消息结构

```
message ComputeSplitPointsBySizeResponse {
required ConsumedCapacity consumed = 1;
repeated PrimaryKeySchema schema = 2;

/**
* Split points between splits, in the increasing order
*
* A split is a consecutive range of primary keys,
* whose data size is about split_size specified in the request.
* The size could be hard to be precise.
*
* A split point is an array of primary-key column w.r.t. table schema,
* which is never longer than that of table schema.
* Tailing -inf will be omitted to reduce transmission payloads.
*/
repeated bytes split_points = 3;

/**
* Locations where splits lies in.
*
* By the managed nature of TableStore, these locations are no more than hints.
* If a location is not suitable to be seen, an empty string will be placed.
*/
message SplitLocation {
required string location = 1;
required sint64 repeat = 2;
}
repeated SplitLocation locations = 4;
}
```

consumed:

- 类型：ConsumedCapacity
- 本次请求消耗的服务能力单元。

schema:

- 类型：PrimaryKeySchema
- 该表的 Schema，与建表时给出的 Schema 相同。

split_points:

- 类型：repeated bytes
- 分片之间的分割点。分割点之间保证单调增。每个分割点都是以Plainbuffer编码的行，并且只有primary-key项。为了减少传输的数据量，分割点最后的-inf不会传输。

locations：

- 类型：repeated SplitLocation
- 分割点所在机器的提示。可以为空。

举个例子，如果有一张表有三列主键，其中首列主键类型为string。调用这个API后得到5个分片，分别为(-inf,-inf,-inf)到("a",-inf,-inf)、("a",-inf,-inf)到("b",-inf,-inf)、("b",-inf,-inf)到("c",-inf,-inf)、("c",-inf,-inf)到("d",-inf,-inf)和("d",-inf,-inf)到(+inf,+inf,+inf)。前三个落在”machine-A”，后两个落在”machine-B”。那么，split_points为（示意）[("a"),("b"),("c"),("d")]，而locations为（示意）"machine-A"*3, "machine-B"*2。

服务能力单元消耗

消耗的读服务能力单元数量与分片的数量相同。不消耗写服务能力单元。

行为：

获取当前实例下所有表的stream信息。

请求结构：

```
message ListStreamRequest {
optional string table_name = 1;
}
```

table_name：

类型：optional string。

当前stream所属的表名。

响应消息结构：

```
message ListStreamResponse {
repeated Stream streams = 1;
}

message Stream {
required string stream_id = 1;
required string table_name = 2;
required int64 creation_time = 3;
```

```
}
```

stream_id：

类型：required string。

当前stream的id。

table_name：

类型：required string。

当前stream所属的表名。

creation_time：

类型：required int64。

当前stream enable的时间。

行为：

获取当前stream的shard信息。

请求结构：

```
message DescribeStreamRequest {
required string stream_id = 1;
optional string inclusive_start_shard_id = 2;
optional int32 shard_limit = 3;
}
```

stream_id：

类型：required string。

当前stream的id。

inclusive_start_shard_id：

类型：required string。

查询起始shard的id。

shard_limit：

类型：required string。

单次查询返回shard数目的上限。

响应消息结构：

```
message DescribeStreamResponse {
required string stream_id = 1;
required int32 expiration_time = 2;
required string table_name = 3;
required int64 creation_time = 4;
required StreamStatus stream_status = 5;
repeated StreamShard shards = 6;
optional string next_shard_id = 7;
}

message StreamShard {
required string shard_id = 1;
optional string parent_id = 2;
optional string parent_sibling_id = 3;
}
```

stream_id：

类型：required string。

当前stream的id。

expiration_time：

类型：required int32。

Stream的过期时间。

table_name：

类型：required string。

当前stream 所属的table名字。

creation_time：

类型：required int32。

当前stream创建的时间。

stream_status：

类型：required StreamStatus。

当前stream的状态，包括enabling和active。

stream_status：

类型：required StreamStatus。

当前stream的状态，包括enabling和active。

shards：

类型：required StreamShard。

streamShard的信息，包括shard的id，父shard的id，父shard的邻居shard信息（适用于父shard发生merge）。

next_shard_id：

类型：optional string。

分页查询下一个shard的起始id。

注意事项：

读取当前shard的数据时需要确保父shard的数据已经全部读取完毕。

行为：

获取读取当前shard记录的起始iterator。

请求结构：

```
message GetShardIteratorRequest {
required string stream_id = 1;
required string shard_id = 2;
}
```

stream_id：

类型：required string。

当前stream的id。

shard_id：

类型：required string。

当前shard的id。

响应消息结构：

```
message GetShardIteratorResponse {
required string shard_iterator = 1;
}
```

shard_iterator：

类型：required string。

读取当前shard记录的起始iterator。

行为：

读取当前shard的增量内容。

请求结构：

```
message GetStreamRecordRequest {
required string shard_iterator = 1;
optional int32 limit = 2;
}
```

shard_iterator：

类型：required string。

当前shard读取的iterator。

响应消息结构：

```
message GetStreamRecordResponse {
message StreamRecord {
required ActionType action_type = 1;
required bytes record = 2;
}
repeated StreamRecord stream_records = 1;
optional string next_shard_iterator = 2;
}
```

StreamRecord：

类型：repeated StreamRecord。

读取当前shard记录的record entry。

shard_iterator：

类型：required string。

下次读取此shard的iterator。

下面是table_store.proto和table_store_filter.proto的详细定义。

table_store.proto

```
package com.alicloud.openservices.tablestore.core.protocol;

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

enum PrimaryKeyType {
INTEGER = 1;
STRING = 2;
BINARY = 3;
}

enum PrimaryKeyOption {
AUTO_INCREMENT = 1;
}

message PrimaryKeySchema {
required string name = 1;
required PrimaryKeyType type = 2;
optional PrimaryKeyOption option = 3;
}

message TableOptions {
optional int32 time_to_live = 1; // 可以动态更改
optional int32 max_versions = 2; // 可以动态更改
optional int64 deviation_cell_version_in_sec = 5; // 可以动态修改
}
```

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

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

message Condition {
required RowExistenceExpectation row_existence = 1;
optional bytes column_condition = 2;
}

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

message ReservedThroughputDetails {
required CapacityUnit capacity_unit = 1; // 表当前的预留吞吐量的值。
required int64 last_increase_time = 2; // 最后一次上调预留吞吐量的时间。
optional int64 last_decrease_time = 3; // 最后一次下调预留吞吐量的时间。
}

message ReservedThroughput {
required CapacityUnit capacity_unit = 1;
}

message ConsumedCapacity {
required CapacityUnit capacity_unit = 1;
}

/* ############################################### CreateTable
############################################### */
/**
* table_meta用于存储表中不可更改的schema属性，可以更改的ReservedThroughput和TableOptions独立出来，作为
UpdateTable的参数。
* message CreateTableRequest {
* required TableMeta table_meta = 1;
* required ReservedThroughput reserved_throughput = 2;
* required TableOptions table_options = 3;
* }
*/
message CreateTableRequest {
required TableMeta table_meta = 1;
required ReservedThroughput reserved_throughput = 2;
optional TableOptions table_options = 3;
}

message CreateTableResponse {
}

/*
```

```
##########################################################################################
################### */


/* ############################################## UpdateTable
############################################## */
message UpdateTableRequest {
required string table_name = 1;
optional ReservedThroughput reserved_throughput = 2;
optional TableOptions table_options = 3;
}

message UpdateTableResponse {
required ReservedThroughputDetails reserved_throughput_details = 1;
required TableOptions table_options = 2;
}
/*
##########################################################################################
################### */

/* ############################################## DescribeTable
############################################## */
message DescribeTableRequest {
required string table_name = 1;
}

message DescribeTableResponse {
required TableMeta table_meta = 1;
required ReservedThroughputDetails reserved_throughput_details = 2;
required TableOptions table_options = 3;
repeated bytes shard_splits = 6;
}
/*
##########################################################################################
##################### */

/* ############################################## ListTable
############################################## */
message ListTableRequest {
}

/**
* 当前只返回一个简单的名称列表
*/
message ListTableResponse {
repeated string table_names = 1;
}
/*
##########################################################################################
################ */

/* ############################################## DeleteTable
############################################## */
message DeleteTableRequest {
required string table_name = 1;
}
```

```
message DeleteTableResponse {
}

/* ############################################## UnloadTable
############################################## */
message UnloadTableRequest {
required string table_name = 1;
}

message UnloadTableResponse {

}
/*
##########################################################################################
################### */

/**
* 时间戳的取值最小值为0，最大值为INT64.MAX
* 1. 若要查询一个范围，则指定start_time和end_time
* 2. 若要查询一个特定时间戳，则指定specific_time
*/
message TimeRange {
optional int64 start_time = 1;
optional int64 end_time = 2;
optional int64 specific_time = 3;
}

/* ############################################## GetRow
############################################## */

enum ReturnType {
RT_NONE = 0;
RT_PK = 1;
}

message ReturnContent {
optional ReturnType return_type = 1;
}

/**
* 1. 支持用户指定版本时间戳范围或者特定的版本时间来读取指定版本的列
* 2. 目前暂不支持行内的断点
*/
message GetRowRequest {
required string table_name = 1;
required bytes primary_key = 2; // encoded as InplaceRowChangeSet, but only has primary key
repeated string columns_to_get = 3; // 不指定则读出所有的列
optional TimeRange time_range = 4;
optional int32 max_versions = 5;
optional bytes filter = 7;
optional string start_column = 8;
optional string end_column = 9;
optional bytes token = 10;
}
```

```
message GetRowResponse {
required ConsumedCapacity consumed = 1;
required bytes row = 2; // encoded as InplaceRowChangeSet
optional bytes next_token = 3;
}
/*
##############################################################################################
############## */

/* ############################################## UpdateRow
############################################## */
message UpdateRowRequest {
required string table_name = 1;
required bytes row_change = 2;
required Condition condition = 3;
optional ReturnContent return_content = 4;
}

message UpdateRowResponse {
required ConsumedCapacity consumed = 1;
optional bytes row = 2;
}
/*
##############################################################################################
################ */

/* ############################################## PutRow
############################################## */
/**
* 这里允许用户为每列单独设置timestamp，而不是强制整行统一一个timestamp。
*/
message PutRowRequest {
required string table_name = 1;
required bytes row = 2; // encoded as InplaceRowChangeSet
required Condition condition = 3;
optional ReturnContent return_content = 4;
}

message PutRowResponse {
required ConsumedCapacity consumed = 1;
optional bytes row = 2;
}
/*
##############################################################################################
############## */

/* ############################################## DeleteRow
############################################## */
/**
* OTS只支持删除该行的所有列所有版本，不支持：
* 1. 删除所有列的所有小于等于某个版本的所有版本
*/
message DeleteRowRequest {
required string table_name = 1;
required bytes primary_key = 2; // encoded as InplaceRowChangeSet, but only has primary key
required Condition condition = 3;
```

```
optional ReturnContent return_content = 4;
}

message DeleteRowResponse {
required ConsumedCapacity consumed = 1;
optional bytes row = 2;
}
/*
##############################################################################################
################## */

/* ############################################## BatchGetRow
############################################## */
message TableInBatchGetRowRequest {
required string table_name = 1;
repeated bytes primary_key = 2; // encoded as InplaceRowChangeSet, but only has primary key
repeated bytes token = 3;
repeated string columns_to_get = 4; // 不指定则读出所有的列
optional TimeRange time_range = 5;
optional int32 max_versions = 6;
optional bytes filter = 8;
optional string start_column = 9;
optional string end_column = 10;
}

message BatchGetRowRequest {
repeated TableInBatchGetRowRequest tables = 1;
}

message RowInBatchGetRowResponse {
required bool is_ok = 1;
optional Error error = 2;
optional ConsumedCapacity consumed = 3;
optional bytes row = 4; // encoded as InplaceRowChangeSet
optional bytes next_token = 5;
}

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

message BatchGetRowResponse {
repeated TableInBatchGetRowResponse tables = 1;
}
/*
##############################################################################################
################## */

/* ############################################## BatchWriteRow
############################################## */

enum OperationType {
PUT = 1;
UPDATE = 2;
DELETE = 3;
```

```
}

message RowInBatchWriteRowRequest {
required OperationType type = 1;
required bytes row_change = 2; // encoded as InplaceRowChangeSet
required Condition condition = 3;
optional ReturnContent return_content = 4;
}

message TableInBatchWriteRowRequest {
required string table_name = 1;
repeated RowInBatchWriteRowRequest rows = 2;
}

message BatchWriteRowRequest {
repeated TableInBatchWriteRowRequest tables = 1;
}

message RowInBatchWriteRowResponse {
required bool is_ok = 1;
optional Error error = 2;
optional ConsumedCapacity consumed = 3;
optional bytes row = 4;
}

message TableInBatchWriteRowResponse {
required string table_name = 1;
repeated RowInBatchWriteRowResponse rows = 2;
}

message BatchWriteRowResponse {
repeated TableInBatchWriteRowResponse tables = 1;
}
/*
##############################################################################################
###################### */

/* ############################################## GetRange
############################################## */
enum Direction {
FORWARD = 0;
BACKWARD = 1;
}

message GetRangeRequest {
required string table_name = 1;
required Direction direction = 2;
repeated string columns_to_get = 3; // 不指定则读出所有的列
optional TimeRange time_range = 4;
optional int32 max_versions = 5;
optional int32 limit = 6;
required bytes inclusive_start_primary_key = 7; // encoded as InplaceRowChangeSet, but only has primary key
required bytes exclusive_end_primary_key = 8; // encoded as InplaceRowChangeSet, but only has primary key
optional bytes filter = 10;
optional string start_column = 11;
optional string end_column = 12;
```

```
optional bytes token = 13;
}

message GetRangeResponse {
required ConsumedCapacity consumed = 1;
required bytes rows = 2; // encoded as InplaceRowChangeSet
optional bytes next_start_primary_key = 3; // 若为空，则代表数据全部读取完毕. encoded as InplaceRowChangeSet,
but only has primary key
optional bytes next_token = 4;
}

/* ##################### ComputeSplitPointsBySize ################### */
message ComputeSplitPointsBySizeRequest {
required string table_name = 1;
required int64 split_size = 2; // in 100MB
}

message ComputeSplitPointsBySizeResponse {
required ConsumedCapacity consumed = 1;
repeated PrimaryKeySchema schema = 2;

/**
* Split points between splits, in the increasing order
*
* A split is a consecutive range of primary keys,
* whose data size is about split_size specified in the request.
* The size could be hard to be precise.
*
* A split point is an array of primary-key column w.r.t. table schema,
* which is never longer than that of table schema.
* Tailing -inf will be omitted to reduce transmission payloads.
*/
repeated bytes split_points = 3;

/**
* Locations where splits lies in.
*
* By the managed nature of TableStore, these locations are no more than hints.
* If a location is not suitable to be seen, an empty string will be placed.
*/
message SplitLocation {
required string location = 1;
required sint64 repeat = 2;
}
repeated SplitLocation locations = 4;
}
```

table_store_filter.proto

```
package com.alicloud.openservices.tablestore.core.protocol;

enum FilterType {
FT_SINGLE_COLUMN_VALUE = 1;
FT_COMPOSITE_COLUMN_VALUE = 2;
```

```
FT_COLUMN_PAGINATION = 3;
}

enum ComparatorType {
CT_EQUAL = 1;
CT_NOT_EQUAL = 2;
CT_GREATER_THAN = 3;
CT_GREATER_EQUAL = 4;
CT_LESS_THAN = 5;
CT_LESS_EQUAL = 6;
}

message SingleColumnValueFilter {
required ComparatorType comparator = 1;
required string column_name = 2;
required bytes column_value = 3;
required bool filter_if_missing = 4;
required bool latest_version_only = 5;
}

enum LogicalOperator {
LO_NOT = 1;
LO_AND = 2;
LO_OR = 3;
}

message CompositeColumnValueFilter {
required LogicalOperator combinator = 1;
repeated Filter sub_filters = 2;
}

message ColumnPaginationFilter {
required int32 offset = 1;
required int32 limit = 2;
}

message Filter {
required FilterType type = 1;
required bytes filter = 2; // Serialized string of filter of the type
}
```

DataType

在 GetStreamRecord 的响应消息中，表示操作类型。

PUT_ROW 表示此次操作类型是PutRow。

UPDATE_ROW 表示此次操作类型是UpdateRow

DELETE_ROW 表示此次操作类型是DeleteRow

枚举取值列表

```
enum ActionType {
PUT_ROW = 1;
UPDATE_ROW = 2;
DELETE_ROW = 3;
}
```

相关操作

GetStreamRecord

CapacityUnit

Column

ColumnCondition

ColumnConditionType

ColumnSchema

ColumnPaginationFilter

ColumnType

ColumnUpdate

ColumnValue

CompositeColumnValueFilter

Condition

ComparatorType

CompositeColumnValueFilter

ConsumedCapacity

Direction

Error

Filter

FilterType

LogicalOperator

OperationType

ReservedThroughput

ReservedThroughputDetails

Row

RowExistenceExpectation

RowInBatchGetRowResponse

RowInBatchGetRowRequest

RowInBatchWriteRowResponse

TableInBatchGetRowRequest

TableInBatchGetRowResponse

TableInBatchWriteRowRequest

TableInBatchWriteRowResponse

TableMeta

SingleColumnValueFilter

PrimaryKeyOption

ReturnContent

ReturnType

TimeRange

TableOptions

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预留读吞吐量。

write：

类型：int32。

描述：本次操作消耗的写服务能力单元或该表的预留写吞吐量。

相关操作

UpdateRow

BatchWriteRow

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。

列判断条件，适用于条件更新（Conditional Update）和过滤器（Filter）功能。

数据结构

```
message ColumnCondition {
required ColumnConditionType type = 1;
required bytes condition = 2;
}
```

type：

类型：ColumnConditionType。

描述：列条件类型。

condition：

类型：bytes。

描述：CompositeCondition 或者 RelationCondition 类型的条件语句通过 Protobuf 序列化后的二进制数据。

相关操作

ConditionUpdate

PutRow

UpdateRow

DeleteRow

BatchWriteRow

Filter

GetRow

GetRange

BatchGetRow

列条件，枚举类型。

CCT_RELATION 表示关系型，仅有一个条件，比如 column_a > 5。

CCT_COMPOSITE 表示组合型，包含多个条件，比如 column_a > 5 OR column_b < 10。

枚举取值列表

```
enum ColumnConditionType {
```

```
CCT_RELATION = 1;
CCT_COMPOSITE = 2;
}
```

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

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;
}
```

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

在 PutRow、UpdateRow 和 DeleteRow 中使用的行判断条件，目前含有 row_existence 和 column_condition 两项。

数据结构

```
message Condition {
required RowExistenceExpectation row_existence = 1;
optional bytes column_condition = 2;
}
```

row_existence：

类型：RowExistenceExpectation。

描述：对该行进行行存在性检查的设置。

column_condition：

类型：bytes。

描述：对列条件的设置。Filter经过protobuf序列化后的bytes。

相关操作

PutRow

UpdateRow

DeleteRow

BatchWriteRow

组合类型，枚举类型。

CT_EQUAL 表示相等。

CT_NOT_EQUAL 表示不相等。

CT_GREATER_THAN 表示大于。

CT_GREATER_EQUAL 表示大于等于。

CT_LESS_THAN 表示小于。

CT_LESS_EQUAL 表示小于等于。

枚举取值列表

```
enum ComparatorType {
CT_EQUAL = 1;
CT_NOT_EQUAL = 2;
CT_GREATER_THAN = 3;
CT_GREATER_EQUAL = 4;
CT_LESS_THAN = 5;
CT_LESS_EQUAL = 6;
}
```

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。

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 PrimaryKeyColumn。

相关操作

BatchWriteRow

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

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，具体含义可参考表格存储的存储相关异常。

message：

类型：string。

描述：当前单行操作的错误信息，具体含义可参考表格存储的存储相关异常。

相关操作

BatchGetRow

BatchWriteRow

逻辑操作符，枚举类型。

LO_NOT 表示非。

LO_AND 表示并。

LO_OR 表示或。

枚举取值列表

```
enum LogicalOperator {
LO_NOT = 1;
LO_AND = 2;
LO_OR = 3;
}
```

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。

UPDATE 表示更新一列数据。

DELETE 表示删除该列。

枚举取值列表

```
enum OperationType {
PUT = 1;
UPDATE = 2;
DELETE = 3;
}
```

在 BatchWriteRow 操作中，表示要插入、更新和删除的一行信息。

数据结构

```
message RowInBatchWriteRowRequest {
required OperationType type = 1;
required bytes row_change = 2; // Plainbuffer编码
required Condition condition = 3;
optional ReturnContent return_content = 4;
}
```

type：

类型：OperationType

是否必要参数：是。

描述：操作类型，包括插入、更新和删除。

row_change：

类型：bytes。

是否必要参数：是。

描述：请求插入、更新或删除的行数据。

- 如果是插入和更新，行数据包括了主键和属性列。
- 如果是删除，行数据包括了主键。

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。

return_content:

类型：ReturnContent。

是否必要参数：否。

写入成功后返回的数据类型，目前仅支持返回主键，主要用于主键列自增功能中。

相关操作

BatchWriteRow

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

UpdateRow

DescribeTable

表示一个表的预留读/写吞吐量信息。

数据结构

```
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

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

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

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 bytes row = 4;
optional bytes next_token = 5;
}
```

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：bytes。

读取到的数据，由Plainbuffer编码，详见Plainbuffer编码

如果该行不存在,则数据为空。

next_token：

类型：bytes。

宽行读取时，下一次读取的起始位置，暂不可用。

相关操作

BatchGetRow

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

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

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 bytes primary_key = 2; //Plainbuffer编码
repeated bytes token = 3;
repeated string columns_to_get = 4; // 不指定则读出所有的列
optional TimeRange time_range = 5;
optional int32 max_versions = 6;
optional bool cache_blocks = 7 [default = true]; // 本次读出的数据是否进入BlockCache
optional bytes filter = 8;
optional string start_column = 9;
optional string end_column = 10;
}
```

table_name：

类型：string。

描述：该表的表名。

primary_key：

类型：repeated bytes

是否必要参数：是。

该行全部的主键列，包含主键名和主键值，由Plainbuffer编码，详见Plainbuffer编码

token：

类型：repeated bytes。

是否必要参数：否。

宽行读取时指定下一次读取的起始位置，暂不可用。

columns_to_get：

类型：repeated string。

描述：该表中需要返回的全部列的列名。

time_range：

类型：TimeRange。

是否必要参数：和max_versions必须至少存在一个。

读取数据的版本时间戳范围。

时间戳的单位是毫秒，取值最小值为0，最大值为INT64.MAX

若要查询一个范围，则指定start_time和end_time

若要查询一个特定时间戳，则指定specific_time

例子：如果指定的time_range为(100, 200)，则返回的列数据的时间戳必须位于[100, 200)范围内，前闭后开区间。

max_versions：

类型：int32。

是否必要参数：和time_range必须至少存在一个。

读取数据时，返回的最多版本个数。

例子：如果指定max_versions为2，则每一列最多返回2个版本的数据。

cache_blocks：

类型：bool。

是否必要参数：否。

本次读出的数据是否进入BlockCache。

默认是true。

当前暂不支持设置为false。

filter：

类型：bytes。

是否必要参数：否。

过滤条件表达式。

Filter经过protobuf序列化后的二进制数据。

start_column：

类型：string。

是否必要参数：否。

指定读取时的起始列，主要用于宽行读。

返回的结果中包含当前起始列。

列的顺序按照列名的字典序排序。

例子：如果一张表有”a” ，” b” ，” c” 三列，读取时指定start_column为“b” ，则会从” b” 列开始读，返回” b” ，” c” 两列。

end_column：

类型：string。

是否必要参数：否。

指定读取时的结束列，主要用于宽行读。

返回的结果中不包含当前结束列。

列的顺序按照列名的字典序排序。

例子：如果一张表有”a” ，” b” ，” c”三列，读取时指定end_column为“b” ，则读到”b”列时会结束，返回”a”列。

相关操作

BatchGetRow

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

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 RowInBatchWriteRowRequest rows = 2;
}
```

table_name：

类型：string。

描述：该表的表名。

rows：

类型：repeated RowInBatchWriteRowRequest。

描述：该表中请求插入、更新和删除的行信息。

相关操作

BatchWriteRow

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

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：请求更新的全部属性列。

相关操作

BatchWriteRow

表示一个表的结构信息。

数据结构

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

table_name：

类型：string。

描述：该表的表名。

primary_key：

类型：repeated PrimaryKeySchema。

描述：该表全部的主键列。

相关操作

CreateTable

DescribeTable

表的参数值，包括TimeToLive，最大版本数等。

数据结构

```
message TableOptions {
optional int32 time_to_live = 1; // 可以动态更改
optional int32 max_versions = 2; // 可以动态更改
optional int64 deviation_cell_version_in_sec = 5; // 可以动态修改
}
```

time_to_live：

类型：int32。

描述：本张表中保存的数据的存活时间，单位秒。

max_versions：

类型：int32.

描述：本张表保留的最大版本数。

deviation_cell_version_in_sec：

类型：int64

描述：最大版本偏差。目的主要是为了禁止写入与预期较大的数据，比如设置deviation_cell_version_in_sec为1000，当前timestamp如果为10000，那么允许写入的timestamp范围为[10000 - 1000, 10000 + 1000]。

分区的范围信息。

数据结构

```
message PartitionRange {
required bytes begin = 1; // encoded as SQLVariant
required bytes end = 2; // encoded as SQLVariant
}
```

begin：

类型：bytes。

描述：分区的起始键，分区的区间为前闭后开，包含起始键。Plainbuffer格式，编码详见PlainBuffer。

end：

类型：bytes。

描述：分区的结束键，分区的区间为前闭后开，不包含结束键。Plainbuffer格式，编码详见PlainBuffer。

主键的属性值，目前仅支持AUTO_INCREMENT。

枚举取值列表

```
enum PrimaryKeyOption {
AUTO_INCREMENT = 1;
}
```

返回的数据内容。

数据结构

```
message ReturnContent {
optional ReturnType return_type = 1;
}
```

return_type：

类型：ReturnType

描述：返回数据的类型。

返回数据的类型。

枚举数据类型

```
enum ReturnType {
RT_NONE = 0;
RT_PK = 1;
}
```

RT_NONE：默认值，不返回任何值。

RT_PK：返回主键列。

查询数据时指定的时间戳范围或特定时间戳值。

数据结构

```
message TimeRange {
optional int64 start_time = 1;
optional int64 end_time = 2;
optional int64 specific_time = 3;
}
```

start_time：

类型：int64。

描述：起始时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。

end_time：

类型：int64。

描述：结束时间戳。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。

specific_time：

类型：int64。

描述：特定的时间戳值。specific_time和[start_time, end_time) 两个中设置一个即可。单位是毫秒。时间戳的取值最小值为0，最大值为INT64.MAX。

因为 Protocol Buffer 序列化和解析小对象的性能很差,所以 OTS 自定义了 PlainBuffer 数据格式用来表示行数据。

格式定义

```
plainbuffer = tag_header row1  [row2]  [row3]
row = ( pk [attr] | [pk] attr | pk attr ) [tag_delete_marker] row_checksum;
pk = tag_pk cell_1 [cell_2] [cell_3]
attr = tag_attr cell1 [cell_2] [cell_3]
cell = tag_cell cell_name [cell_value] [cell_op] [cell_ts] cell_checksum
```

```
cell_name = tag_cell_name formated_value
cell_value = tag_cell_value formated_value
cell_op = tag_cell_op cell_op_value
cell_ts = tag_cell_ts cell_ts_value
row_checksum = tag_row_checksum row_crc8
cell_checksum = tag_cell_checksum row_crc8

formated_value = value_type value_len value_data
value_type = int8
value_len = int32

cell_op_value = delete_all_version | delete_one_version
cell_ts_value = int64
delete_all_version = 0x01 (1byte)
delete_one_version = 0x03 (1byte)
```

Tag取值

```
tag_header = 0x75 (4byte)
tag_pk = 0x01 (1byte)
tag_attr = 0x02 (1byte)
tag_cell = 0x03 (1byte)
tag_cell_name = 0x04 (1byte)
tag_cell_value = 0x05 (1byte)
tag_cell_op = 0x06 (1byte)
tag_cell_ts = 0x07 (1byte)
tag_delete_marker = 0x08 (1byte)
tag_row_checksum = 0x09 (1byte)
tag_cell_checksum = 0x0A (1byte)
```

ValueType 取值

formated_value 中 value_type 的取值如下：

```
VT_INTEGER = 0x0
VT_DOUBLE = 0x1
VT_BOOLEAN = 0x2
VT_STRING = 0x3
VT_NULL = 0x6
VT_BLOB = 0x7
VT_INF_MIN = 0x9
VT_INF_MAX = 0xa
VT_AUTO_INCREMENT = 0xb
```

计算 Checksum

计算 checksum 的基本逻辑是：

- 针对每个 cell 的 name/value/type/timestamp 计算。
- 针对 row 里面的 delete 计算，其中有 delete mark 补单字节1；若没有，补单字节0。
- 因为对每个 cell 计算了 checksum,所以计算 row 的 checksum 的时候直接利用 cell 的checksum 来计算, 即 row 的 crc 只对 cell 的 checksum 做 crc, 不对数据做 crc。

C++实现：

```
void GetChecksum(uint8_t* crc, const InplaceCell& cell)
{
Crc8(crc, cell.GetName());
Crc8(crc, cell.GetValue().GetInternalSlice());
Crc8(crc, cell.GetTimestamp());
Crc8(crc, cell.GetOpType());
}

void GetChecksum(uint8_t* crc, const InplaceRow& row)
{
const std::deque<InplaceCell>& pk = row.GetPrimaryKey();
for (size_t i = 0; i < pk.size(); i++) {
uint8_t* cellcrc;
*cellcrc = 0;
GetChecksum(cellcrc, pk[i]);
Crc8(crc, *cellcrc);
}
for (int i = 0; i < row.GetCellCount(); i++) {
uint8_t* cellcrc;
*cellcrc = 0;
GetChecksum(cellcrc, row.GetCell(i));
Crc8(crc, *cellcrc);
}

uint8_t del = 0;
if (row.HasDeleteMarker()) {
del = 1;
}
Crc8(crc, del);
}
```

举例

假设一行数据有两列 PK,4 列数据,其中 PK 类型为 string 和 integer,而 4 列数据中分别是 string/int/double,版本分别是 1001、1002 和 1003,还有一列是删除所有版本。

- 主键列：
  - [pk1:string:iampk]
  - [pk2:integer:100]
- 属性列：
  - [column1:string:bad:1001]
  - [column2:integer:128:1002]

- [column3:double:34.2:1003]
- [column4:del_all_versions]

编码：

```
    <Header开始>[0x75]
<主键列开始>[0x1]
<Cell1>[0x3][0x4][3][pk1][0x5][3][5][iampk]
<Cell2>[0x3][0x4][3][pk2][0x5][0][100]
<属性列开始>[0x2]
<Cell1>[0x3][0x4][7][column1][0x5][0x3][3][bad][0x7][1001]
<Cell2>[0x3][0x4][7][column2][0x5][0x0][128][0x7][1002]
<Cell3>[0x3][0x4][7][column3][0x5][0x1][34.2][0x7][1003]
<Cell4>[0x3][0x4][7][column4][0x5][0x6][1]
```

宽行读取过滤条件，适用于filter。

数据结构

```
message ColumnPaginationFilter {
required int32 offset = 1;
required int32 limit = 2;
}
```

offset：

类型：int32。

描述：起始列的位置，表示从第几列开始读。

limit：

类型：int32。

描述：读取的列的个数。

相关操作

Filter

GetRow

GetRange

BatchGetRow

多个组合条件，比如 column_a > 5 AND column_b = 10 等。适用于 ConditionUpdate 和 Filter 功能。

数据结构

```
message CompositeColumnValueFilter {
required LogicalOperator combinator = 1;
repeated Filter sub_filters = 2;
}
```

combinator：

类型：LogicalOperator。

描述：逻辑操作符。

sub_filter：

类型：Filter。

描述：子条件表达式。

相关操作

ConditionUpdate

PutRow

UpdateRow

DeleteRow

BatchWriteRow

Filter

GetRow

GetRange

BatchGetRow

列判断条件，适用于条件更新（Conditional Update）和过滤器（Filter）功能。

数据结构

```
message Filter {
required FilterType type = 1;
required bytes filter = 2;
}
```

type：

类型：FilterType。

描述：列条件类型。

filter：

类型：bytes。

描述：CompositeColumnValueFilter 、ColumnPaginationFilter或者SingleColumnValueFilter类型的条件语句通过 Protobuf 序列化后的二进制数据。

相关操作

ConditionUpdate

PutRow

UpdateRow

DeleteRow

BatchWriteRow

Filter

GetRow

GetRange

BatchGetRow

条件更新或过滤的类型。

FT_SINGLE_COLUMN_VALUE：单列条件。

FT_COMPOSITE_COLUMN_VALUE：多列组合条件。

FT_COLUMN_PAGINATION：宽行读取条件。

枚举取值列表

```
enum FilterType {
FT_SINGLE_COLUMN_VALUE = 1;
FT_COMPOSITE_COLUMN_VALUE = 2;
FT_COLUMN_PAGINATION = 3;
}
```

主键的属性值。

数据结构

```
message PrimaryKeySchema {
required string name = 1;
required PrimaryKeyType type = 2;
optional PrimaryKeyOption option = 3;
}
```

name：

类型：string。

描述：该列的列名。

type：

类型：PrimaryKeyType。

描述：该列的类型。

option：

类型：PrimaryKeyOption。

描述：该列的附加属性值。

主键的类型。

枚举数据类型

INTEGER：整数。

STRING：字符串。

BINARY：二进制。

```
enum PrimaryKeyType {
INTEGER = 1;
STRING = 2;
BINARY = 3;
}
```

单个条件，比如 column_a > 5 等。适用于 ConditionUpdate 和 Filter 功能。

数据结构

```
message SingleColumnValueFilter {
required ComparatorType comparator = 1;
required string column_name = 2;
required bytes column_value = 3;
```

```
required bool filter_if_missing = 4;
required bool latest_version_only = 5;
}
```

comparator：

类型：ComparatorType。

描述：比较类型。

column_name：

类型：string。

描述：列名称。

column_value：

类型：bytes。

描述：ColumnValue经过Plainbuffer编码后的值。

filter_if_missing：

类型：bool。

描述：当某行的这一列不存在时，设置条件是否过滤。比如条件是 column_a>0，filter_if_missing 是 true，当某一行没有列 column_a 时，这一行的条件判断就会通过。

latest_version_only：

类型：bool。

描述：是否只对最新版本有效。如果为true，则表示只检测最新版本的值是否满足条件；如果是false，则会检测所有版本的值是否满足条件。

相关操作

ConditionUpdate

PutRow

UpdateRow

DeleteRow

BatchWriteRow

Filter

GetRow

GetRange

BatchGetRow

在 GetStreamRecord 的响应消息中，表示一行数据。

数据结构

```
 message StreamRecord {
required ActionType action_type = 1;
required bytes record = 2;
}
```

action_type：

类型：required ActionType。

描述：表示该行的操作类型。

record：

类型：required bytes。

描述：表示该行的数据内容。

相关操作

GetStreamRecord

表示一个表的stream信息。

数据结构

```
message StreamSpecification {
required bool enable_stream = 1;
optional int32 expiration_time = 2;
}
```

enable_stream：

类型：bool。

描述：该表是否打开stream。

expiration_time：

类型：int32

描述：该表的stream过期时间。

相关操作

CreateTable

DescribeTable

UpdateTable

表示一个表的stream信息。

数据结构

```
message StreamDetails {
required bool enable_stream = 1;
optional string stream_id = 2;
optional int32 expiration_time = 3;
optional int64 last_enable_time = 4;
}
```

enable_stream：

类型：required bool。

描述：该表是否打开stream。

stream_id：

类型：optional string

描述：该表的stream的id。

expiration_time：

类型：optional int32

描述：该表的stream的过期时间。

last_enable_time：

类型：optional int64

描述：该stream的打开的时间。

相关操作

DescribeTable

基于 Table Store Stream API 以及 Table Store SDK，您可以使用 API 或者 SDK 读取 Stream 的记录。在实时获取增量数据的时候，需要注意分区的信息不是静态的。分区可能会分裂、合并。当分区发生变化后，需要

处理分区之间的依赖关系以确保单主键上数据顺序读取。同时，如果您的数据是由多客户端并发生成，为了提高导出增量数据的效率，也需要有多消费端并行读取各个分区的增量记录。

Stream Client 可以解决 Stream 数据处理时的常见问题，例如，如何做负载均衡，故障恢复，Checkpoint，分区信息同步确保分区信息消费顺序等。使用 Stream Client 后，您只需要关心每条记录的处理逻辑即可。

下面内容将介绍 Stream Client 的原理，以及如何使用 Stream Client 高效打造适合自身业务的数据通道。

Stream Client 原理

本节介绍 Stream Client 的内部逻辑。

为了方便做任务调度，以及记录当前每个分区的读取进度，Stream Client 使用了 Table Store 的一张表来记录这些信息，您可以自行选定表名，但是要确保该表名没有其他业务在使用。

Stream Client 中为每个分区定义了一个租约（lease），每个租约的拥有者叫做 worker。租约用来记录一个分区的增量数据消费者（即 worker）和读取进度信息。当一个新的消费端启动后，worker 会初始化，检查分区和租约信息，为没有相应租约的分区创建租约。当因为分裂或合并产生新的分区时，Stream Client 会在表中插入一条租约记录。新的记录会被某一个 Stream Client 的 worker 抢到并不断处理，如果有新的 worker 加入则可能会做负载均衡，调度至新 worker 处理。

租约记录的 schema 如下所示：

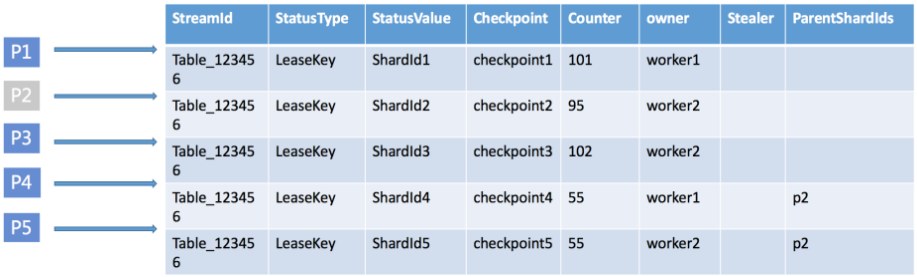
参数	说明
主键 StreamId	当前处理 Stream 的 Id。
主键 StatusType	当前 lease 的 Key。
主键 StatusValue	当前 lease 对应的分区的 Id。
属性 Checkpoint	记录当前分区 Stream 数据消费的位置（用户故障恢复）
属性 LeaseCounter	表示乐观锁。每个 lease 的 owner 会持续更新这个 counter 值，用来续租表示继续占有当前的 lease。
属性 LeaseOwner	拥有当前租约的 worker 名。
属性 LeaseStealer	在负载均衡的时候，表示准备挪至哪个 worker。
属性 ParentShardIds	当前 shard 的父分区信息。在 worker 消费当前 shard 时，保证父分区的 stream 数据已经被消费。

示例

下图是典型的使用 Stream Client 消费增量数据的分布式架构：

在这张图中，worker1 和 worker2 是两个基于 Stream Client 的消费端，例如在 ECS 上启动的进程。数据源不断地读写 Table Store 中的表，这张表初期有分区 P1、P2 和 P3。随着访问量和数据量的增大，分区 P2 发生了一次分裂，产生了 P4 和 P5。worker1 在初始阶段会消费 P1 的数据，worker2 消费 P2 和 p3 的数据，当 P2 发生分裂后，新的分区 P4 会被分配给 worker1，分区 P5 分配给 worker2。但 Stream Client 会确保 P2 的 R5 记录已经被消费完成后，再开始消费 P4 和 P5 的数据。如果这时部署了一个新的消费端 worker3，那可能发生的一件事情是 worker2 上的某个分区会被 worker3 调度走，由此产生负载均衡。

在上述场景下，Stream Client 会在表中生成如下租约信息：



StreamId	StatusType	StatusValue	Checkpoint	Counter	owner	Stealer	ParentShardIds
Table_123456	LeaseKey	ShardId1	checkpoint1	101	worker1		
Table_123456	LeaseKey	ShardId2	checkpoint2	95	worker2		
Table_123456	LeaseKey	ShardId3	checkpoint3	102	worker2		
Table_123456	LeaseKey	ShardId4	checkpoint4	55	worker1		p2
Table_123456	LeaseKey	ShardId5	checkpoint5	55	worker2		p2

Stream Client 中的 worker 是抽象消费 Stream 数据的载体，每一个分区会被分配一个 worker 即 lease 的拥有者，拥有者会不断地通过心跳，即更新 LeaseCounter 来续约当前的分区租约，通常每一个 Stream 消费端会起一个 worker，worker 在完成初始化后获取当前需要处理的分区信息，同时 worker 会维护自己的线程池，并发地循环拉取所拥有的各分区的增量数据。worker 初始化流程如下：

1. 读取 Table Store 的配置，对内部访问 Table Store 的客户端进行初始化。
2. 获取对应表的 Stream 信息，并初始化租约管理类。租约管理类会同步租约信息，为新分区创建租约记录。
3. 初始化分区同步类，分区同步类会维持当前持有的分区心跳。
4. 循环获取当前 worker 持有的分区的增量数据。

下载 Stream Client

下载安装 JAR 包

Maven：

```
<dependency>
<groupId>com.aliyun.openservices</groupId>
<artifactId>tablestore-streamclient</artifactId>
<version>1.0.0</version>
</dependency>
```

说明：Stream Client 的代码是开源的，您可以下载源码了解原理，也欢迎您将基于 Stream 的优秀 Sample 代码分享给我们。

使用 Stream Client 接口

为了方便您使用 Stream Client 消费 Stream 数据、隐藏分区读取逻辑和调度逻辑，Stream Client 提供了 IRecordProcessor 接口。Stream Client 的 worker 会在拉取到 stream 数据后调用 processRecords 函数来触发用户的处理数据逻辑。

```
public interface IRecordProcessor {

void initialize(InitializationInput initializationInput);

void processRecords(ProcessRecordsInput processRecordsInput);

void shutdown(ShutdownInput shutdownInput);
}
```

参数解释如下：

参数	说明
void initialize(InitializationInput initializationInput);	用于初始化一个读取任务，表示 stream client 准备开始读取某个 shard 的数据。
void processRecords(ProcessRecordsInput processRecordsInput);	表示具体读取到数据后用户希望如何处理这批记录。在 ProcessRecordsInput 中有一个 getCheckpointer 函数可以得到一个 IRecordProcessorCheckpointer。这个接口是框架提供给用户用来做 checkpoint 的接口，用户可以自行决定多久做一次 checkpoint。
void shutdown(ShutdownInput shutdownInput);	表示结束某个 shard 的读区任务。

需要注意的是：

> 因为读取任务所在机器不同，进程可能会遇到各种类型的错误，例如因为环境因素重启，需要定期对处理完的数据做记录（checkpoint）。当任务重启后，会接着上次的 checkpoint 继续往后做。也就是说，在极端情况下，Stream Client 不保证 ProcessRecordsInput 里传给您的记录只有一次，只会保证数据至少传一次，且记录的顺序不变。如果出现局部数据重复发送的情况，需要您注意业务的处理逻辑。

如果您希望减少在出错情况下数据的重复处理，可以增加做 checkpoint 的频率。但是过于频繁的 checkpoint 会降低系统的吞吐量，请根据自身业务特点决定做 checkpoint 的频率。

如果您发现增量数据不能及时被消费，可以增加消费端的资源，例如用更多的节点来读取 stream 记录。

下面给出了一个简单的示例，使用 Stream Client 来实时获取增量数据，并在控制台输出增量数据。

```
 public class StreamSample {
class RecordProcessor implements IRecordProcessor {

private long creationTime = System.currentTimeMillis();
private String workerIdentifier;

public RecordProcessor(String workerIdentifier) {
this.workerIdentifier = workerIdentifier;
}

public void initialize(InitializationInput initializationInput) {
// Trace some info before start the query like stream info etc.
}

public void processRecords(ProcessRecordsInput processRecordsInput) {
List<StreamRecord> records = processRecordsInput.getRecords();

if(records.size() == 0) {
// No more records we can wait for the next query
System.out.println("no more records");
}
for (int i = 0; i < records.size(); i++) {
System.out.println("records:" + records.get(i));
}

// Since we don't persist the stream record we can skip blow step
System.out.println(processRecordsInput.getCheckpointer().getLargestPermittedCheckpointValue());
try {
processRecordsInput.getCheckpointer().checkpoint();
} catch (ShutdownException e) {
e.printStackTrace();
} catch (StreamClientException e) {
e.printStackTrace();
} catch (DependencyException e) {
e.printStackTrace();
}
}

public void shutdown(ShutdownInput shutdownInput) {
// finish the query task and trace the shutdown reason
System.out.println(shutdownInput.getShutdownReason());
}
}

class RecordProcessorFactory implements IRecordProcessorFactory {
```

```
private final String workerIdentifier;

public RecordProcessorFactory(String workerIdentifier) {
this.workerIdentifier = workerIdentifier;
}

public IRecordProcessor createProcessor() {
return new StreamSample.RecordProcessor(workerIdentifier);
}
}

public Worker getNewWorker(String workerIdentifier) {
// Please replace with your table info
final String endPoint = "";
final String accessId = "";
final String accessKey = "";
final String instanceName = "";

StreamConfig streamConfig = new StreamConfig();
streamConfig.setOTSClient(new SyncClient(endPoint, accessId, accessKey,
instanceName));
streamConfig.setDataTableName("teststream");
streamConfig.setStatusTableName("statusTable");

Worker worker = new Worker(workerIdentifier, new ClientConfig(), streamConfig,
new StreamSample.RecordProcessorFactory(workerIdentifier), Executors.newCachedThreadPool(), null);
return worker;
}

public static void main(String[] args) throws InterruptedException {
StreamSample test = new StreamSample();
Worker worker1 = test.getNewWorker("worker1");
Thread thread1 = new Thread(worker1);
thread1.start();
}
}
```