ALIBABA CLOUD

阿里云

专有云企业版

表格存储Tablestore

开发指南

产品版本：v3.16.2

文档版本：20220915

阿里云

法律声明

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

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

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

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

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

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

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

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

目录

1.API参考

1.1. API操作

1.1.1. GetRow

调用GetRow接口根据指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;
}
```

名称	类型	是否必选	描述
table_name	string	是	要读取的数据所在的表名。
primary_key	bytes	是	指定行全部的主键列，包含主键名和主键值，由Plainbuffer编码。
columns_to_get	string	否	需要返回的全部列的列名。如果为空，则返回指定行的所有列。columns_to_get中string的个数不应超过128个。 如果指定的列不存在，则不会返回指定列的数据；如果给出了重复的列名，返回结果只会包含一次指定列。
time_range	TimeRange	否，和max_versions必须至少存在一个	读取数据的版本时间戳范围。时间戳的单位为毫秒，取值最小值为0，最大值为INT64.MAX。 如果要查询一个范围，则指定start_time和end_time；如果要查询一个特定时间戳，则指定specific_time。 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。

名称	类型	是否必选	描述
max_versions	int32	否，和time_range至少存在一个	读取数据时，返回的最多版本个数。 如果指定max_versions为2，则每一列最多返回2个版本的数据。
filter	bytes	否	过滤条件表达式。Filter经过protobuf序列化后的二进制数据。
start_column	string	否	指定读取时的起始列，主要用于宽行读。列的顺序按照列名的字典序排序。返回的结果中包含当前起始列。 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	string	否	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。 如果一张表有a、b、c三列，读取时指定end_column为b，则读到b列时会结束，返回a列。
token	bytes	否	宽行读取时指定下一次读取的起始位置，暂不可用。

响应消息结构

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

名称	类型	描述
consumed	ConsumedCapacity	本次操作消耗的服务能力单元。单位为CU。
row	bytes	读取到的数据。如果该行不存在，则数据为空。返回的数据为Plainbuffer格式。

服务能力单元消耗

- 如果请求的行不存在，则消耗1读CU（读服务能力单元）。
- 如果请求的行存在，则消耗读服务能力单元的数值为指定行所有主键列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。
- 如果请求超时，结果未定义，则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。

1.1.2. PutRow

调用PutRow接口插入数据到指定的行，如果该行不存在，则新增一行；如果该行存在，则覆盖原有行。

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;
}
```

名称	类型	是否必选	描述
table_name	string	是	请求写入数据的表名。
row	bytes	是	写入的行数据，包括主键和属性列。Plainbuffer格式。
condition	Condition	是	在数据写入前是否进行行存在性检查。取值范围如下： • 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。 如果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	ReturnContent	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;
}
```

名称	类型	描述
consumed	ConsumedCapacity	本次操作消耗的服务能力单元。
row	bytes	当设置了return_content后，返回的数据。Plainbuffer格式。 如果未设置return_content或者未返回数据，则此处为NULL。

服务能力单元消耗

- 当插入的行不存在时，根据指定的条件检查不同，插入数据消耗的服务能力单元不同。
  - 如果指定条件检查为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。
  - 如果指定条件检查为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。
  - 如果指定条件检查为EXPECT_EXIST，本次插入失败并且消耗1单位写CU和1单位读CU。
- 当插入的行存在时，根据指定的条件检查不同，插入数据消耗的服务能力单元不同。
  - 如果指定条件检查为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。
  - 如果指定条件检查为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。
  - 如果指定条件检查为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）时，如果操作成功，按照上述消耗服务能力单元方式进行计算。如果操作失败，则消耗1单位写CU和1单位读CU。
- 如果请求超时，结果未定义，则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单位写CU。

1.1.3. UpdateRow

调用UpdateRow接口更新指定行的数据。

> **说明** 如果指定行不存在，则新增一行；如果指定行存在，则根据请求的内容在该行中新增、修改或者删除指定列的值。

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;
}
```

名称	类型	是否必选	描述
table_name	string	是	请求更新数据的表名。

名称	类型	是否必选	描述
row_change	bytes	是	更新的数据，包括主键和属性列，由Plainbuffer编码。 该行本次需要更新的全部属性列，表格存储会根据row_change中UpdateType的内容在该行中新增、修改或者删除指定列的值。该行已存在的且不在row_change中的列将不受影响。 UpdateType的取值范围如下： • PUT：此时value必须为有效的属性列值。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。表示删除该列特定版本的数据。 • DELETE_ALL：此时该value和timestamp都必须为空。表示删除该列所有版本的数据。 **说明** 删除本行的全部属性列不等同于删除本行。如果需要删除本行，请使用DeleteRow操作。
condition	Condition	是	在数据更新前是否进行存在性检查。取值范围如下： • IGNORE（默认）：不做行存在性检查。如果忽略该行是否存在，则无论该行是否存在，都不会因此导致本次操作失败。 • EXPECT_EXIST：期望行存在。如果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。
return_content	ReturnContent	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;
}
```

名称	类型	描述
consumed	ConsumedCapacity	本次操作消耗的服务能力单元。单位为CU。

名称	类型	描述
row	bytes	当设置了return_content后，返回的数据。如果未设置return_content或者没返回值，此处为NULL。 返回的数据为Plainbuffer格式。

服务能力单元消耗

- 当更新的行不存在时，根据指定的条件检查不同，更新数据消耗的服务能力单元不同。
  - 如果指定条件检查为IGNORE，则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。
  - 如果指定条件检查为EXPECT_EXIST，则本次插入失败并且消耗1单位写CU和1单位读CU。
- 当更新的行存在时，根据指定的条件检查不同，更新数据消耗的服务能力单元不同。
  - 如果指定条件检查为IGNORE，则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。
  - 如果指定条件检查为EXPECT_EXIST，则除了需要消耗在条件检查为IGNORE情况下的写CU外，还需消耗该行主键数据大小除以4 KB向上取整的读CU。
- 如果请求超时，结果未定义，则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1个写服务能力单元和1个读服务能力单元。

1.1.4. DeleteRow

调用DeleteRow接口删除一行数据。

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;
}
```

名称	类型	是否必选	描述
table_name	string	是	数据表名称。
primary_key	bytes	是	删除行的主键。 主键为Plainbuffer格式。

名称	类型	是否必选	描述
condition	Condition	是	在数据操作前是否进行存在性检查。取值范围如下： • IGNORE（默认）：表示不做行存在性检查。 当忽视该行是否存在时，无论该行实际是否存在，都会操作成功。 • EXPECT_EXIST：表示期望行存在。 当期待该行存在时，如果实际该行存在，则本次删除操作会成功；如果实际该行不存在，则本次删除操作会失败，返回错误。
return_content	ReturnContent	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;
}
```

名称	类型	描述
consumed	ConsumedCapacity	本次操作消耗的服务能力单元。单位为CU。
row	bytes	当设置了return_content后，返回的数据。如果未设置return_content或者没有返回值，此处为NULL。 返回的数据为Plainbuffer格式。

服务能力单元消耗

- 当删除的行不存在时，根据指定的条件检查不同，删除数据消耗的服务能力单元不同。
  - 如果指定条件检查为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。
  - 如果指定条件检查为EXPECT_EXIST，则删除该行失败，且消耗1单位的写CU和1单位的读CU。
- 当删除的行存在时，根据指定的条件检查不同，删除数据消耗的服务能力单元不同。
  - 如果指定条件检查为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。
  - 如果指定条件检查为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。
- 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。

1.1.5. GetRange

调用GetRange接口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;
}
```

名称	类型	是否必选	描述
table_name	string	是	要读取的数据所在的表名。
direction	Direction	是	本次查询的顺序。 • 如果为正序，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。 • 如果为逆序，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	repeated string	否	需要返回的全部列的列名。如果为空，则返回指定行的所有列。columns_to_get中string的个数不应超过128个。 如果给出了重复的列名，返回结果只会包含一次该列。
time_range	TimeRang	否，和max_versions只能存在一个	读取数据的版本时间戳范围。时间戳的单位为毫秒，取值最小值为0，最大值为INT64.MAX。 如果要查询一个范围，则指定start_time和end_time。如果要查询一个特定时间戳，则指定specific_time。 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	int32	否，和time_range只能存在一个	读取数据时，最多返回的版本个数。 如果指定max_versions为2，则每一列最多返回2个版本的数据。

名称	类型	是否必选	描述
limit	int32	否	本次读取最多返回的行数。如果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，表格存储最多返回的行数为5000且总数据大小不超过4 MB。
inclusive_start_primary_key	bytes	是	本次范围读取的起始主键，由Plainbuffer编码。 如果该行存在，则响应中一定会包含此行。
exclusive_end_primary_key	bytes	是	本次范围读取的终止主键，由Plainbuffer编码。 无论该行是否存在，响应中都不会包含此行。 在GetRange中，inclusive_start_primary_key和exclusive_end_primary_key中的Column的type可以使用本操作专用的两个类型INF_MIN和INF_MAX。类型为INF_MIN的Column小于其它Column，类型为INF_MAX的Column大于其它Column。
filter	bytes	否	过滤条件表达式。Filter经过protobuf序列化后的二进制数据。
start_column	string	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。 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	string	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。 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;
}
```

名称	类型	描述
consumed	ConsumedCapacity	本次操作消耗的服务能力单元。单位为CU。

名称	类型	描述
rows	bytes	读取到的所有数据，由Plainbuffer编码。 如果请求中direction为FORWARD，则所有行按照主键由小到大进行排序；如果请求中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，其顺序不保证与请求中的columns_to_get一致；primary_key_columns的顺序也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，则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。
next_start_primary_key	bytes	本次操作的断点信息，由Plainbuffer编码。 如果为空，则本次GetRange的响应消息中已包含了请求范围内的所有数据。 如果不为空，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，如果需要剩下的数据，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。 **说明** 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 MB。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。
- 如果请求超时，结果未定义，则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。

1.1.6. BatchGetRow

行为

批量读取一个或多个表中的若干行数据。

BatchGetRow 操作可视为多个 GetRow 操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

与执行大量的 GetRow 操作相比，使用 BatchGetRow 操作可以有效减少请求的响应时间，提高数据的读取速率。

请求结构

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

tables：

- 类型：repeated TableInBatchGetRowRequest
- 是否必要参数：是
- 指定了需要要读取的行信息。
- 若 tables 中出现了下述情况，则操作整体失败，返回错误。
  - tables 中任一表不存在。
  - tables 中任一表名不符合表名命名规范。
  - tables 中任一行未指定主键、主键名称不符合规范或者主键类型不正确。
  - tables 中任一表的 columns_to_get 内的列名不符合列名命名规范。
  - tables 中包含同名的表。
  - tables 中任一表内包含主键完全相同的行。
  - 所有 tables 中 RowInBatchGetRowRequest 的总个数超过 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。

> **说明** 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。

1.1.7. BatchWriteRow

行为

批量插入、修改或删除一个或多个表中的若干行数据。

BatchWriteRow 操作可视为多个 PutRow、UpdateRow、DeleteRow 操作的集合。各个操作独立执行，独立返回结果，独立计算服务能力单元。

与执行大量的单行写操作相比，使用 BatchWriteRow 操作可以有效减少请求的响应时间，提高数据的写入速率。

请求结构

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

tables：

- 类型：repeated TableInBatchWriteRowRequest
- 是否必要参数：是
- 指定了需要要执行写操作的行信息。
- 以下情况都会返回整体错误：
  - tables 中任一表不存在。
  - tables 中包含同名的表。
  - tables 中任一表名表名不符合表名命名规范。
  - tables 中任一行操作未指定主键、主键列名称不符合规范或者主键列类型不正确。
  - tables 中任一属性列名称不符合列名命名规范。
  - tables 中任一行操作存在与主键列同名的属性列。
  - tables 中任一主键列或者属性列的值大小超过上限。
  - tables 中任一表中存在主键完全相同的请求。
  - 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。

> **说明** 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。

1.1.8. CreateTable

调用CreateTable接口根据给定的表结构信息创建相应的表。

注意事项

- 创建成功的表并不能立刻提供读写服务。通常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1024个表，如果需要提高单实例下表数目的上限，请使用人工服务提高此限额。

请求结构

```
message CreateTableRequest {
    required TableMeta table_meta = 1;
    required ReservedThroughput reserved_throughput = 2;
    optional TableOptions table_options = 3;
    optional StreamSpecification stream_spec = 5;
}
```

参数	类型	是否必选	描述
table_meta	TableMeta	是	数据表的结构信息。其中table_name必须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应符合命名规则和数据类型，type取值只能为STRING、INTEGER或者BINARY。 建表成功后，表的Schema将不能修改。

参数	类型	是否必选	描述
reserved_throughput	ReservedThroughput	是	数据表的预留读吞吐量或者预留写吞吐量。任何表的预留读吞吐量与预留写吞吐量均不能超过100000。单位为CU。 数据表的预留读吞吐量或者预留写吞吐量设定可以通过UpdateTable进行更改。
table_options	TableOptions	否	主要包括TimeToLive和最大版本数的设置。
stream_spec	StreamSpecification	否	描述是否打开Stream相关的属性。

响应消息结构

```
message CreateTableResponse {
}
```

1.1.9. ListTable

行为

获取当前实例下已创建的所有表的表名。

请求结构

```
message ListTableRequest {
}
```

响应消息结构

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

table_names:

- 类型：repeated string
- 当前实例下所有表的表名。

注意事项

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

1.1.10. DeleteTable

行为

删除本实例下指定的表。

请求结构

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

table_name：

- 类型：string
- 是否必要参数：是
- 需要删除的表的表名。

响应消息结构

```
message DeleteTableResponse {
}
```

DeleteTable 的响应中没有任何错误即表示表已经成功删除。

1.1.11. UpdateTable

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。

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。

table_options:

- 类型：TableOptions
- 修改后，最新的table_options参数值。

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。

1.1.12. DescribeTable

行为

查询指定表的结构信息和预留读/写吞吐量设置信息。

请求结构

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

table_name:

- 类型：string
- 是否必要参数：是
- 需要查询的表名。

响应消息结构

```
message DescribeTableResponse {
    required TableMeta table_meta = 1;
    required ReservedThroughputDetails reserved_throughput_details = 2;
    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相关的属性。

shard_splits:

- 类型：bytes
- 当前表所有分区的分裂点。

1.1.13. ComputeSplitPointsBySize

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)、("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。

1.1.14. ListStream

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：repeated 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。

1.1.15. DescribeStream

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。

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。

1.1.16. GetShardIterator

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。

1.1.17. GetStreamRecord

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 raw_string next_shard_iterator = 2;
    optional ConsumedCapacity consumed = 3;
}
```

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。

consumed:

- 类型：ConsumedCapacity
- 本次操作消耗的服务能力单元。
- 读取Stream数据时CU的计算是根据读取所有行总大小除以4 KB向上取整。

1.1.18. 表格存储 ProtocolBuffer 消息定义

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  ############################
```

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

```
    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 on
ly 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 In
placeRowChangeSet, 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
}
```

1.1.19. CreateIndex

在指定的主表上创建索引表。

请求结构

```
message CreateIndexRequest {
    required string main_table_name = 1;
    required IndexMeta index_meta = 2;
    optional bool include_base_data = 3;
}
```

main_table_name

- 类型：string
- 是否必要参数：是
- 索引表所在主表的名字

index_meta

- 类型：IndexMeta
- 是否必要参数：是
- 索引表的schema

include_base_data

- 类型：bool
- 是否必要参数：否
- 索引表中，是否包含在创建索引表前，主表的存量数据。

响应消息结构

```
message CreateIndexResponse {
}
```

1.1.20. DeleteIndex

在指定的主表上删除索引表。

请求结构

```
message DropIndexRequest {
    required string main_table_name = 1;
    required string index_name = 2;
}
```

main_table_name

- 类型：string
- 是否必要参数：是
- 要删除的索引表所在的主表

index_name

- 类型：string
- 是否必要参数：是
- 要删除的索引表名称

响应消息结构

```
message DropIndexResponse {
}
```

1.2. DataType

1.2.1. ActionType

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;
}
```

1.2.2. CapacityUnit

表示一次操作消耗服务能力单元的值或是一个表的预留读/写吞吐量的值。

数据结构

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

read：

- 类型：int32
- 描述：本次操作消耗的读服务能力单元或该表的预留读吞吐量。

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。

1.2.3. ColumnPaginationFilter

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。

1.2.4. ComparatorType

关系运算符，被定义成枚举类型。

- 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;
}
```

1.2.5. CompositeColumnValueFilter

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。

1.2.6. Condition

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。

1.2.7. ConsumedCapacity

表示一次操作消耗的服务能力单元。

数据结构

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

capacity_unit:

- 类型：CapacityUnit
- 描述：本次操作消耗的服务能力单元的值。

1.2.8. Direction

在 GetRange 操作中，查询数据的顺序。

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

枚举取值列表

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

1.2.9. Error

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

数据结构

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

code：

- 类型：string
- 描述：当前单行操作的错误码。

message：

- 类型：string
- 描述：当前单行操作的错误信息。

1.2.10. Filter

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。

1.2.11. PrimaryKeyOption

主键的属性值，目前仅支持AUTO_INCREMENT。

枚举取值列表

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

1.2.12. FilterType

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;
}
```

1.2.13. LogicalOperator

逻辑操作符，被定义成枚举类型。

- 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;
}
```

1.2.14. OperationType

在 UpdateRow 中，表示对一列的修改方式。

- PUT 表示插入一列或覆盖该列的数据。
- 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;
}
```

1.2.15. PartitionRange

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。

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。

1.2.16. PlainBuffer

因为 Protocol Buffer 序列化和解析小对象的性能很差，所以表格存储自定义了 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 和 4 列数据。其中 PK 类型为 string 和 integer；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]
```

1.2.17. PrimaryKeySchema

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。

1.2.18. PrimaryKeyType

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;
}
```

1.2.19. ReservedThroughput

表示一个表设置的预留读/写吞吐量数值。

数据结构

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

capacity_unit：

- 类型：CapacityUnit
- 描述：表当前的预留读/写吞吐量数值。

1.2.20. ReservedThroughputDetails

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。

1.2.21. ReturnContent

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。

1.2.22. ReturnType

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。

1.2.23. RowExistenceExpectation

行存在性判断条件，枚举类型。

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

枚举取值列表

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

1.2.24. RowInBatchGetRowResponse

在 BatchGetRow 操作的返回消息中，表示一行数据。

数据结构

```
message RowInBatchGetRowResponse {
 required bool is_ok = 1 [default = true];
 optional Error error = 2;
 optional ConsumedCapacity consumed = 3;
 optional 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。
- 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。

1.2.25. RowInBatchWriteRowRequest

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; // encoded as InplaceRowChangeSet
    required Condition condition = 3;
    optional ReturnContent return_content = 4;
}
```

type:

- 类型：OperationType
- 描述：操作类型。

row_change:

- 类型：bytes
- 描述：行数据，包括主键和属性列，由Plainbuffer编码。

condition:

- 类型：Condition
- 描述：条件更新的值，包括行条件检测和属性列检测。

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。

1.2.26. RowInBatchWriteRowResponse

在 BatchWriteRow 操作的返回消息中，表示一行写入操作的结果。

数据结构

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

is_ok:

- 类型：bool
- 描述：该行操作是否成功。若为 true，则该行写入成功，error 无效；若为 false，则该行写入失败。

error:

- 类型：Error
- 描述：该行操作的错误信息。

consumed:

- 类型：ConsumedCapacity
- 描述：该行操作消耗的服务能力单元。

1.2.27. SingleColumnValueFilter

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protobuf序列化后的值。

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。

1.2.28. StreamDetails

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。

1.2.29. StreamRecord

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。

1.2.30. StreamSpecification

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。

1.2.31. TableInBatchGetRowRequest

在 BatchGetRow 操作中，表示要读取的一个表的请求信息。

数据结构

```
message TableInBatchGetRowRequest {
    required string table_name = 1;
    repeated 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。

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。
- 默认为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。

1.2.32. TableInBatchGetRowResponse

在 BatchGetRow 操作的返回消息中，表示一个表的数据。

数据结构

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

table_name：

- 类型：string
- 描述：该表的表名。

rows：

- 类型：repeated RowInBatchGetRowResponse
- 描述：该表中读取到的全部行数据。

1.2.33. TableInBatchWriteRowRequest

在 BatchWriteRow 操作中，表示要写入的一个表的请求信息。

数据结构

```
message TableInBatchWriteRowRequest {
 required string table_name = 1;
 repeated 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。

1.2.34. TableInBatchWriteRowResponse

在 BatchWriteRow 操作中，表示对一个表进行写入的结果。

数据结构

```
message TableInBatchWriteRowResponse {
 required string table_name = 1;
 repeated RowInBatchWriteRowResponse put_rows = 2;
 repeated RowInBatchWriteRowResponse update_rows = 3;
 repeated RowInBatchWriteRowResponse delete_rows = 4;
}
```

table_name：

- 类型：string
- 描述：该表的表名。

put_rows：

- 类型：RowInBatchWriteRowResponse
- 描述：该表中 PutRow 操作的结果。

update_rows：

- 类型：RowInBatchWriteRowResponse
- 描述：该表中 UpdateRow 操作的结果。

delete_rows：

- 类型：RowInBatchWriteRowResponse
- 描述：该表中 DeleteRow 操作的结果。

1.2.35. TableMeta

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：PrimaryKeySchema
- 描述：该表全部的主键列。

1.2.36. TableOptions

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]。

1.2.37. TimeRange

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。

2.SDK参考

2.1. Java-SDK

2.1.1. 前言

介绍表格存储Java SDK的使用。本文适用于Java SDK 5.0.0以上版本。

前提条件

已获取一个授权账号以及一对AccessKey ID和AccessKey Secret。请参见获取Accesskey在Apsara Uni-manager运营控制台上获取和查看AccessKey。

兼容性

当前最新版本：5.13.5

- 对4.x.x系列SDK：兼容
- 对2.x.x系列SDK：不兼容

2.1.2. 安装

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

环境准备

安装表格存储Java SDK需使用JDK 6及以上版本。

安装方式

您可以通过以下两种方式安装表格存储Java SDK：

- 在Maven项目中加入依赖项

  在Maven工程中使用表格存储Java SDK，只需在pom.xml中加入相应依赖即可。以5.13.5版本为例，在<dependencies>内加入如下内容：

  ```
  <dependency>
      <groupId>com.aliyun.openservices</groupId>
      <artifactId>tablestore</artifactId>
      <version>5.13.5</version>
  </dependency>
  ```

- Eclipse中导入JAR包

  以5.13.5版本为例，详细步骤如下：

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

  其中<versionId>表示表格存储Java SDK的版本，例如5.13.5版本的文件名称为tablestore-5.13.5.jar。

iv. 在Eclipse中选择您的工程，右键选择**Properties > Java Build Path > Add JARs**。

v. 选中您在第3步拷贝的所有JAR文件。

示例程序

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

2.1.3. 初始化

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

确定Endpoint

Endpoint是阿里云表格存储服务各个实例的域名地址，目前支持下列形式。

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

> **说明** 除了公网可以访问外，也支持私网地址。

请按照如下步骤获取实例的Endpoint：

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   访问地址即是该实例的Endpoint。

初始化对接

在获取到AccessKey ID和AccessKey Secret等密钥后，您可以按照下面步骤进行初始化对接。

新建Client

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

表格存储新版的SDK提供了两种Client，SyncClient和AsyncClient，分别对应同步接口和异步接口。同步接口调用完毕后请求即执行完成，使用方便，用户可以先使用同步接口了解表格存储的各种功能。异步接口相比同步接口更加灵活，如果对性能有一定需求，可以在使用异步接口和使用多线程之间做一些取舍。

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

示例代码

- 使用默认配置创建SyncClient。

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

- 使用自定义配置创建SyncClient。

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

HTTPS

升级到java 7后即可。

多线程

- 支持多线程。
- 使用多线程时，建议共用一个OTSClient对象。

2.1.4. 使用手册

2.1.4.1. 表操作

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

创建表（CreateTable）

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

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

- TableMeta

  TableMeta包含表名和表的主键定义。

参数	说明
TableName	数据表名称。

参数	说明
List	表的主键定义。 ○ 表的主键可包含1个~4个主键列。主键列是有顺序的，与用户添加的顺序相同。比如PRIMARY KEY（A, B, C）与PRIMARY KEY（A, C, B）是不同的两个主键结构。表格存储会按照整个主键的大小对行进行排序。 ○ 第一列主键作为分片键。分片键相同的数据会存放在同一个分片内，所以相同分片键下最好不要超过10 GB以上数据，否则会导致单分片过大，无法分裂。另外，数据的读/写访问最好在不同的分片键上均匀分布，有利于负载均衡。 ○ 属性列不需要定义。表格存储每行的数据列都可以不同，属性列的列名在写入时指定。

- TableOptions

参数	说明
TTL	TimeToLive，数据的保存时间。 ○ 系统根据数据的时间戳、当前时间、表的TTL三者判断数据是否过期。当**当前时间-数据的时间戳>表的**`TTL`时，系统会自动清理过期的数据。 ○ 创建数据表时，如果希望数据永不过期，可以设置TTL为-1；创建数据表后，可以通过UpdateTable接口动态修改数据生命周期。 ○ 在使用TTL功能时需要注意写入时是否指定了时间戳以及指定的时间戳是否合理。如果需指定时间戳，建议设置MaxTimeDeviation。 ○ 单位为秒。
MaxTimeDeviation	写入数据的时间戳与系统当前时间的偏差允许最大值。 ○ 默认情况下系统会为新写入的数据生成一个时间戳，数据自动过期功能需要根据这个时间戳判断数据是否过期。用户也可以指定写入数据的时间戳。如果用户写入的时间戳非常小，与当前时间偏差已经超过了表上设置的TTL时间，写入的数据会立即过期。设置MaxTimeDeviation可以避免这种情况。 ○ 创建数据表时，如果未设置有效版本偏差，系统会使用默认值86400；创建数据表后，可以通过UpdateTable接口动态修改有效版本偏差。 ○ 单位为秒。
MaxVersions	数据表中的属性列能够保留数据的最大版本个数。当属性列数据的版本个数超过设置的最大版本数时，系统会自动删除较早版本的数据。 创建数据表时，可以自定义属性列的最大版本数；创建数据表后，可以通过UpdateTable接口动态修改数据表的最大版本数。

- ReservedThroughtput：数据表的预留读/写吞吐量配置。

  容量型实例的预留读/写吞吐量只能设置为0，不允许预留。

示例

```
private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME, PrimaryKeyType
.STRING)); //为主表添加主键列。
        int timeToLive = -1; //数据的过期时间，单位为秒，-1代表永不过期。带索引表的数据表数据过期时
间必须为-1。
        int maxVersions = 3; //保存的最大版本数，带索引表的数据表最大版本数必须为1。
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions);
        request.setReservedThroughput(new ReservedThroughput(new CapacityUnit(0, 0))); //设
置读写预留值，容量型实例只能设置为0，高性能实例可以设置为非零值。
        client.createTable(request);
    }
```

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

示例

获取实例下所有表名。

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

更新表（UpdateTable）

表格存储支持更新表的预留读/写吞吐量（ReservedThroughput）和配置信息（TableOptions）。

关于ReservedThroughput和TableOptions参数说明请参见“创建表”部分内容。ReservedThroughput的调整时间间隔限制为1分钟。

示例

更新表的TTL和最大版本数。

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

查询表描述信息（DescribeTable）

DescribeTable接口可以查询表的结构信息（TableMeta）、配置信息（TableOptions）和预留读/写吞吐量的情况（ReservedThroughputDetails）。

关于TableMeta和TableOptions参数说明请参见“创建表”部分内容，ReservedThroughputDetails除了包含表的预留吞吐量的值外，还包括最近一次上调或者下调的时间。

示例

获取表的描述信息。

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

删除表（DeleteTable）

删除本实例下指定的表。

示例

删除表。

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

2.1.4.2. 单行数据操作

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

插入一行数据（PutRow）

PutRow接口用于插入一行数据，如果原来该行已经存在，会覆盖原来的一行。

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

- 示例1

  写入10列属性列，每列写入1个版本，由服务端指定版本号（时间戳）。

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

- 示例2

  写入10列属性列，每列写入3个版本，由客户端指定版本号（时间戳）。

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

- 示例3

  期望原行不存在时写入。

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

- 示例4

  期望原行存在，且Col0的值大于100时写入。

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

读取一行数据（GetRow）

单行读GetRow接口用于读取一行数据，参数说明请参见下表。

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

- 示例1

  读取一行，设置读取最新版本，设置ColumnsToGet。

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

- 示例2

  设置过滤器。

```
        private static void getRow(SyncClient client, String pkValue) {
            //构造主键。
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuild
er();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromS
tring(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();
            //读取一行数据。
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, prim
aryKey);
            //设置读取最新版本。
            criteria.setMaxVersions(1);
            //设置过滤器，当Col0的值为0时返回该行。
            SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter
("Col0",
                    SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0
));
            //如果不存在Col0列，也不返回。
            singleColumnValueFilter.setPassIfMissing(false);
            criteria.setFilter(singleColumnValueFilter);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row row = getRowResponse.getRow();
            System.out.println("读取完毕，结果为：");
            System.out.println(row);
        }
```

更新一行数据（UpdateRow）

UpdateRow接口用于更新一行数据，如果原行不存在，会新写入一行。

更新操作包括写入某列、删除某列和删除某列的某一版本。

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

- 示例1

  更新一些列，删除某列的某一版本，删除某列。

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

- 示例2

  设置更新的条件。

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

删除一行数据（DeleteRow）

DeleteRow接口用于删除一行。

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

- 示例1

  删除一行。

```
        private static void deleteRow(SyncClient client, String pkValue) {
            //构造主键。
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuild
er();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromS
tring(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();
            RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey)
;
            client.deleteRow(new DeleteRowRequest(rowDeleteChange));
        }
```

- 示例2

  设置删除条件。

```
        private static void deleteRow(SyncClient client, String pkValue) {
            //构造主键。
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuild
er();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromS
tring(pkValue));
            PrimaryKey primaryKey = primaryKeyBuilder.build();
            RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, primaryKey)
;
            //期望原行存在，且Col0的值大于100时删除。
            Condition condition = new Condition(RowExistenceExpectation.EXPECT_EXIST);
            condition.setColumnCondition(new SingleColumnValueCondition("Col0",
                    SingleColumnValueCondition.CompareOperator.GREATER_THAN, ColumnValue.
fromLong(100)));
            rowDeleteChange.setCondition(condition);
            client.deleteRow(new DeleteRowRequest(rowDeleteChange));
        }
```

2.1.4.3. 多行数据操作

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

批量读（BatchGetRow）

批量读接口可以一次请求读取多行数据，参数与GetRow接口参数相同。批量读取的所有行采用相同的参数条件，例如ColumnsToGet=[colA]，则要读取的所有行都只读取colA这一列。

与BatchWriteRow接口类似，使用BatchGetRow接口时也需要检查返回值。当存在部分行失败，而不抛出异常的情况，此时失败行的信息在BatchGetRowResponse中。可以通过BatchGetRowResponse#getFailedRows方法获取失败的行的信息，通过BatchGetRowResponse#isAllSucceed方法可以判断是否所有行都获取成功。

示例

读取10行，设置版本条件、要读取的列、过滤器等。

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

批量写（BatchWriteRow）

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

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

调用BatchWriteRow接口时，需要检查返回值。因为批量写入可能存在部分行成功部分行失败的情况，此时失败行的Index及错误信息在返回的BatchWriteRowResponse中，而并不抛出异常。可通过BatchWriteRowResponse的isAllSucceed方法判断是否全部成功。如果不检查，可能会忽略掉部分操作的失败。另一方面，BatchWriteRow接口也是可能抛出异常的。例如服务端检查到某些操作出现参数错误，可能会抛出参数错误的异常，此时该请求中所有的操作都未执行。

示例

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

```
        private static void batchWriteRow(SyncClient client) {
            BatchWriteRowRequest batchWriteRowRequest = new BatchWriteRowRequest();
            //构造rowPutChange1。
            PrimaryKeyBuilder pk1Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
            pk1Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk
1"));
            RowPutChange rowPutChange1 = new RowPutChange(TABLE_NAME, pk1Builder.build());
            //添加一些列。
            for (int i = 0; i < 10; i++) {
                rowPutChange1.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
            }
            //添加到batch操作中。
            batchWriteRowRequest.addRowChange(rowPutChange1);
            //构造rowPutChange2。
            PrimaryKeyBuilder pk2Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
            pk2Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk
2"));
            RowPutChange rowPutChange2 = new RowPutChange(TABLE_NAME, pk2Builder.build());
            //添加一些列。
            for (int i = 0; i < 10; i++) {
                rowPutChange2.addColumn(new Column("Col" + i, ColumnValue.fromLong(i)));
            }
            //添加到batch操作中。
            batchWriteRowRequest.addRowChange(rowPutChange2);
            //构造rowUpdateChange。
            PrimaryKeyBuilder pk3Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
            pk3Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk
3"));
            RowUpdateChange rowUpdateChange = new RowUpdateChange(TABLE_NAME, pk3Builder.bu
ild());
            //添加一些列。
            for (int i = 0; i < 10; i++) {
                rowUpdateChange.put(new Column("Col" + i, ColumnValue.fromLong(i)));
            }
            //删除一列。
            rowUpdateChange.deleteColumns("Col10");
            //添加到batch操作中。
            batchWriteRowRequest.addRowChange(rowUpdateChange);
            //构造rowDeleteChange。
            PrimaryKeyBuilder pk4Builder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
            pk4Builder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromString("pk
4"));
            RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, pk4Builder.bu
```

```
            RowDeleteChange rowDeleteChange = new RowDeleteChange(TABLE_NAME, pk4Builder.bu
ild());
            //添加到batch操作中。
            batchWriteRowRequest.addRowChange(rowDeleteChange);
            BatchWriteRowResponse response = client.batchWriteRow(batchWriteRowRequest);
            System.out.println("是否全部成功：" + response.isAllSucceed());
            if (!response.isAllSucceed()) {
                for (BatchWriteRowResponse.RowResult rowResult : response.getFailedRows())
{
                    System.out.println("失败的行：" + batchWriteRowRequest.getRowChange(rowR
esult.getTableName(), rowResult.getIndex()).getPrimaryKey());
                    System.out.println("失败原因：" + rowResult.getError());
                }
                /**
                 * 可以通过createRequestForRetry方法再构造一个请求对失败的行进行重试。此处只给出构
造重试请求的部分。
                 * 推荐的重试方法是使用SDK的自定义重试策略功能，支持对batch操作的部分行错误进行重试
。设定重试策略后，调用接口处即不需要增加重试代码。
                 */
                BatchWriteRowRequest retryRequest = batchWriteRowRequest.createRequestForRe
try(response.getFailedRows());
            }
        }
```

范围读（GetRange）

范围读取接口用于读取一个范围内的数据。表格存储表中的行都是按照主键排序的，而主键是由全部主键列按照顺序组成的，所以不能理解为表格存储会按照某列主键排序，这是常见的误区。

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

GetRange请求的参数说明请参见下表。

参数	说明
Direction	枚举类型，包括FORWARD、BACKWARD，分别代表正序、反序。
InclusiveStartPrimaryKey	范围的起始主键（包含）。 如果Direction设置为**BACKWARD**，起始主键要大于结束主键。
ExclusiveEndPrimaryKey	范围的结束主键（不包含）。 如果Direction设置为**BACKWARD**，起始主键要大于结束主键。
Limit	本次请求返回的最大行数。
ColumnsToGet	读取的列的集合。如果不设置此参数，则读取所有列。

参数	说明
MaxVersions	最多读取多少个版本。MaxVersions与TimeRange必须至少设置一个。
TimeRange	读取的版本号的范围。MaxVersions与TimeRange必须至少设置一个。
Filter	过滤器在服务端对读取的结果再进行一次过滤。

示例

正序读取，判断NextStartPrimaryKey是否为null，读取完范围内的全部数据。

```
       private static void getRange(SyncClient client, String startPkValue, String endPkVal
ue) {
            RangeRowQueryCriteria rangeRowQueryCriteria = new RangeRowQueryCriteria(TABLE_N
AME);
            //设置起始主键。
            PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder
();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromStr
ing(startPkValue));
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(primaryKeyBuilder.build());
            //设置结束主键。
            primaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
            primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME, PrimaryKeyValue.fromStr
ing(endPkValue));
            rangeRowQueryCriteria.setExclusiveEndPrimaryKey(primaryKeyBuilder.build());
            rangeRowQueryCriteria.setMaxVersions(1);
            System.out.println("GetRange的结果为：");
            while (true) {
                GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(ran
geRowQueryCriteria));
                for (Row row : getRangeResponse.getRows()) {
                    System.out.println(row);
                }
                //如果nextStartPrimaryKey不为null，则继续读取。
                if (getRangeResponse.getNextStartPrimaryKey() != null) {
                    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getN
extStartPrimaryKey());
                } else {
                    break;
                }
            }
        }
```

迭代读（createRangeIterator）

迭代读取数据。

示例

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

2.1.4.4. 主键列自增

设置非分区键的主键列为自增列后，在写入一行数据时，自增列无需填值，表格存储会自动生成自增列的值。该值在分区键级别唯一且严格递增。主要用于系统设计中需要使用主键列自增功能的场景，例如电商网站的商品ID、大型网站的用户ID、论坛帖子的ID、聊天工具的消息ID等。

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

特点

主键列自增具有如下特点：

- 自增列的值在分区键级别唯一且严格递增，但不保证连续。
- 自增列的数据类型为64位的有符号长整型。
- 自增列是数据表级别的，同一个实例下可以有自增列或者非自增列的数据表。

> **说明** 无论是否使用主键列自增功能，不影响条件更新的规则。

限制

主键列自增有如下限制：

- 每张数据表最多只能设置一个主键列为自增列，主键中的分区键不能设置为自增列。
- 只能在创建数据表时指定自增列，对于已存在的数据表不能创建自增列。
- 只有整型的主键列才能设置为自增列，系统自动生成的自增列值为64位的有符号长整型。

- 属性列不能设置为自增列。

接口

主键列自增的相关接口说明请参见下表。

接口	说明
CreateTable	创建数据表时，请设置非分区键的主键列为自增列，否则无法使用主键列自增功能。
UpdateTable	数据表创建后，不能通过UpdateTable修改数据表的主键列为自增列。
PutRow	写入数据时，无需为自增列设置具体值，表格存储会自动生成自增列的值。 通过设置ReturnType为RT_PK，可以获取完整的主键值，完整的主键值可以用于GetRow查询数据。
UpdateRow	
BatchWriteRow	
GetRow	使用GetRow时需要完整的主键值，通过设置PutRow、UpdateRow或者BatchWriteRow中的ReturnType为RT_PK可以获取完整的主键值。
BatchGetRow	

示例

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

1. 创建表

   创建表时，只需将自增的主键属性设置为AUTO_INCREMENT。

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

2. 写数据

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

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

2.1.4.5. 条件更新

只有满足条件时，才能对数据表中的数据进行更新；当不满足条件时，更新失败。

条件更新支持算术运算（=、!=、>、>=、<、<=）和逻辑运算（NOT、AND、OR），支持最多10个条件的组合。适用于PutRow、UpdateRow、DeleteRow和BatchWriteRow接口。

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

列判断条件包括列条件和行存在性条件。

列判断条件	说明
列条件	目前支持SingleColumnValueCondition和CompositeColumnValueCondition，是基于某一列或者某些列的列值进行条件判断，与过滤器Filter中的条件类似。

列判断条件	说明
行存在性条件	对数据表进行更改操作时，系统会先检查行存在性条件，如果不满足行存在性条件，则更改失败并给用户报错。 行存在性条件包括如下类型。 • IGNORE：忽略 • EXPECT_EXIS：期望存在 • EXPECT_NOT_EXIST：期望不存在 通过不同接口操作数据表的数据时的行存在性条件更新规则请参见行存在性条件更新规则。

行存在性条件更新规则

> **说明** BatchWriteRow操作由多个PutRow、UpdateRow、DeleteRow子操作组成，所以通过BatchWriteRow接口操作数据表中的数据时，请根据操作类型查看对应接口的更新规则。

接口	IGNORE	EXPECT_EXIST	EXPECT_NOT_EXIST
PutRow（已存在行）	失败	成功	失败
PutRow（不存在行）	成功	失败	失败
UpdateRow（已存在行）	失败	成功	失败
UpdateRow（不存在行）	成功	失败	失败
DeleteRow（已存在行）	成功	成功	失败
DeleteRow（不存在行）	失败	失败	失败

示例

使用列判断条件和乐观锁的示例代码如下：

- 构造SingleColumnValueCondition。

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

- 构造CompositeColumnValueCondition。

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

- 通过Condition实现乐观锁机制，递增一列。

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

2.1.4.6. 过滤器（Filter）

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

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

Filter是对读取后的结果再进行一次过滤，所以SingleColumnValueFilter或者CompositeColumnValueFilter中的参考列必须在读取的结果内。

如果指定的要读取的列中不包含参考列，则过滤器无法获取参考列的值。

示例

- 构造SingleColumnValueFilter。

```
        //设置过滤器，当Col0列的值为0时，返回该行。
        SingleColumnValueFilter singleColumnValueFilter = new SingleColumnValueFilter("Co
l0",
                SingleColumnValueFilter.CompareOperator.EQUAL, ColumnValue.fromLong(0));
        //如果不存在Col0列，也不返回该行。
        singleColumnValueFilter.setPassIfMissing(false);
        //只判断最新版本。
        singleColumnValueFilter.setLatestVersionsOnly(true);
```

- 构造CompositeColumnValueFilter。

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

2.1.4.7. 全局二级索引

介绍全局二级索引的创建、使用和删除操作。

创建全局二级索引

在创建数据表时同时创建全局二级索引或者创建数据表后单独创建全局二级索引，请根据实际情况创建全局二级索引。

- 创建数据表同时创建全局二级索引

```
   private static void createTable(SyncClient client) {
        TableMeta tableMeta = new TableMeta(TABLE_NAME);
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_1, PrimaryKeyType.STRING)); //为数据表添加主键列。
        tableMeta.addPrimaryKeyColumn(new PrimaryKeySchema(PRIMARY_KEY_NAME_2, PrimaryKeyType.INTEGER)); //为数据表添加主键列。
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_1, DefinedColumnType.STRING)); //为数据表添加预定义列。
        tableMeta.addDefinedColumn(new DefinedColumnSchema(DEFINED_COL_NAME_2, DefinedColumnType.INTEGER)); //为数据表添加预定义列。
        int timeToLive = -1; //数据的过期时间，单位为秒，-1代表永不过期。带索引表的数据表数据过期时间必须为-1。
        int maxVersions = 1; //保存的最大版本数，带索引表的数据表最大版本数必须为1。
        TableOptions tableOptions = new TableOptions(timeToLive, maxVersions);
        ArrayList<IndexMeta> indexMetas = new ArrayList<IndexMeta>();
        IndexMeta indexMeta = new IndexMeta(INDEX_NAME);
        indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //为索引表添加主键列。
        indexMeta.addDefinedColumn(DEFINED_COL_NAME_2); //为索引表添加属性列。
        indexMetas.add(indexMeta);
        CreateTableRequest request = new CreateTableRequest(tableMeta, tableOptions, indexMetas); //创建数据表时一同创建索引表。
        client.createTable(request);
    }
```

- 单独创建全局二级索引

> **说明** 本接口支持索引表中包含以及不包含数据表中存量数据。当CreateIndexRequest的最后一个参数为true时，即为包含存量数据；为false时，即为不包含存量数据。

```
private static void createIndex(SyncClient client) {
    IndexMeta indexMeta = new IndexMeta(INDEX_NAME); //新建索引Meta。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_2); //指定DEFINED_COL_NAME_2列为索引表的第一列主键。
    indexMeta.addPrimaryKeyColumn(DEFINED_COL_NAME_1); //指定DEFINED_COL_NAME_1列为索引表的第二列主键。
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, true); //将索引表添加到数据表上，包含存量数据。
    CreateIndexRequest request = new CreateIndexRequest(TABLE_NAME, indexMeta, false); //将索引表添加到数据表上，不包含存量数据。
    /**通过将IncludeBaseData参数设置为true，新建索引表后会开启数据表表中存量数据的同步，之后可以通过索引表查询全部数据，同步时间跟数据量的大小有一定的关系。
    */
    request.setIncludeBaseData(true);
    client.createIndex(request); //创建索引表。
}
```

读取索引表中的数据

读取数据时，如果需要返回的属性列在索引表中，则直接读取索引表中的数据；如果需要返回的属性列不在索引表中，则需要反查数据表。

- 当需要返回的属性列在索引表中时，可以直接读取索引表。

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

- 当需要返回的属性列不在索引表中时，需要反查数据表。

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

```
    rangeRowQueryCriteria.setInclusiveStartPrimaryKey(startPrimaryKeyBuilder.build());
    //设置结束主键。
    PrimaryKeyBuilder endPrimaryKeyBuilder = PrimaryKeyBuilder.createPrimaryKeyBuilder();
    endPrimaryKeyBuilder.addPrimaryKeyColumn(DEFINED_COL_NAME_1, PrimaryKeyValue.INF_MAX)
; //设置需要读取的索引列最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_1, PrimaryKeyValue.INF_MAX)
; //数据表PK最大值。
    endPrimaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME_2, PrimaryKeyValue.INF_MAX)
; //数据表PK最大值。
    rangeRowQueryCriteria.setExclusiveEndPrimaryKey(endPrimaryKeyBuilder.build());
    rangeRowQueryCriteria.setMaxVersions(1);
    while (true) {
        GetRangeResponse getRangeResponse = client.getRange(new GetRangeRequest(rangeRowQ
ueryCriteria));
        for (Row row : getRangeResponse.getRows()) {
            PrimaryKey curIndexPrimaryKey = row.getPrimaryKey();
            PrimaryKeyColumn pk1 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_NAM
E1);
            PrimaryKeyColumn pk2 = curIndexPrimaryKey.getPrimaryKeyColumn(PRIMARY_KEY_NAM
E2);
            PrimaryKeyBuilder mainTablePKBuilder = PrimaryKeyBuilder.createPrimaryKeyBuil
der();
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME1, pk1.getValue());
            mainTablePKBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME2, ke2.getValue());
            PrimaryKey mainTablePK = mainTablePKBuilder.build(); //根据索引表PK构造数据表PK
。
            //反查数据表。
            SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(TABLE_NAME, main
TablePK);
            criteria.addColumnsToGet(DEFINED_COL_NAME3); //读取数据表的DEFINED_COL_NAME3列
。
            //设置读取最新版本。
            criteria.setMaxVersions(1);
            GetRowResponse getRowResponse = client.getRow(new GetRowRequest(criteria));
            Row mainTableRow = getRowResponse.getRow();
            System.out.println(row);
        }
        //如果nextStartPrimaryKey不为null，则继续读取。
        if (getRangeResponse.getNextStartPrimaryKey() != null) {
            rangeRowQueryCriteria.setInclusiveStartPrimaryKey(getRangeResponse.getNextSta
rtPrimaryKey());
        } else {
            break;
        }
    }
}
```

删除全局二级索引

删除索引表。

```
private static void deleteIndex(SyncClient client) {
    DeleteIndexRequest request = new DeleteIndexRequest(TABLE_NAME, INDEX_NAME); //指定数据表名称及索引表名称。
    client.deleteIndex(request); //删除索引表。
}
```

2.1.4.8. 通道服务

2.1.4.8.1. 快速开始

介绍如何使用Java SDK最小化的体验通道服务，完整的代码在文末的附录中。

体验通道服务

使用Java SDK最小化的体验通道服务。

1. 初始化Tunnel Client。

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

2. 创建通道。

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

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

3. 用户自定义数据消费Callback，开始自动化的数据消费。

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

注意事项

- TunnelWorkerConfig中默认会启动读数据和处理数据的线程池。如果使用的是单台机器，则会启动多个TunnelWorker，强烈建议共用一个TunnelWorkerConfig。
- 在创建全量加增量类型的Tunnel时，由于Tunnel的增量日志最多会保留7天（具体的值和数据表的Stream的日志过期时间一致），全量数据如果在7天内没有消费完成，则此Tunnel进入增量阶段会出现OTSTunnelExpired错误，导致增量数据无法消费。
- TunnelWorker的初始化需要预热时间，该值受TunnelWorkerConfig中的heartbeatIntervalInSec参数影响，可以通过TunnelWorkerConfig中的setHeartbeatIntervalInSec方法配置，默认为30s，最小支持调整到5s，具体原理请参见数据消费框架配置详解。
- 当Tunnel从全量切换至增量阶段时，全量的Channel会结束，增量的Channel会启动，此阶段会有初始化时间，该值也受TunnelWorkerConfig中的heartbeatIntervalInSec参数影响。
- 当客户端（TunnelWorker）没有被正常shutdown时（例如异常退出或者手动结束），TunnelWorker会自动进行资源的回收，包括释放线程池，自动调用用户在Channel上注册的shutdown方法，关闭Tunnel连接等。

附录：完整代码

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

```
            worker.connectAndWorking();
        } catch (Exception e) {
            e.printStackTrace();
            worker.shutdown();
            tunnelClient.shutdown();
        }
    }
}
```

2.1.4.8.2. 数据消费框架配置详解

通道服务是基于表格存储数据接口的全增量一体化服务，可以用于对数据表中历史存量数据和新增数据的消费处理。

Tunnel Client为通道服务的自动化数据消费框架。Tunnel Client通过每一轮的定时心跳探测（Heartbeat）进行如下操作：

- 活跃Channel的探测。
- Channel和ChannelConnect状态的更新。
- 数据处理任务的初始化、运行和结束等。

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

配置	参数	说明
Heartbeat的间隔和超时时间	heartbeatTimeoutInSec	Heartbeat的超时间隔。 默认值为300s。 当Heartbeat发生超时，Tunnel服务端会认为当前TunnelClient不可用（失活），客户端需要重新的进行ConnectTunnel。
Heartbeat的间隔和超时时间	heartbeatIntervalInSec	进行Heartbeat的间隔。 Heartbeat用于活跃Channel的探测、Channel状态的更新、（自动化）数据拉取任务的初始化等。 默认值为30s，最小支持配置到5s，单位为s。

配置	参数	说明
记录消费位点的时间间隔	checkpointIntervalInMillis	用户消费完数据后，向Tunnel服务端进行记录消费位点操作（checkpoint）的时间间隔。 默认值为5000ms，单位为ms。 **说明** • 因为读取任务所在机器不同，进程可能会遇到各种类型的错误。例如因为环境因素重启，需要定期对处理完的数据做记录（checkpoint）。当任务重启后，会接着上次的checkpoint继续往后做。在极端情况下，通道服务不保证传给您的记录只有一次，只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。 • 如果希望减少在出错情况下数据的重复处理，可以增加做checkpoint的频率。但是过于频繁的checkpoint会降低系统的吞吐量，请根据自身业务特点决定checkpoint的操作频率。
客户端的自定义标识	clientTag	客户端的自定义标识，用于生成Tunnel Client ID。通过自定义此参数可以用于区分TunnelWorker。
数据处理的自定义Callback	channelProcessor	用户注册的处理数据的Callback，包括process和shutdown方法。
数据读取和数据处理的线程池资源配置	readRecordsExecutor	用于数据读取的线程池资源。无特殊需求，建议使用默认的配置。

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

2.1.4.8.3. 创建新通道

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

请求参数

参数	说明
TableName	创建通道的数据表名称。
TunnelName	通道的名称。
TunnelType	通道的类型，支持全量（BaseData）、增量（Stream）和全量加增量（BaseAndStream）三种。

响应参数

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

示例

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

2.1.4.8.4. 获取表内的通道信息

ListTunnel操作列举了某个数据表内通道的具体信息。

请求参数

参数	说明
TableName	列举通道信息的数据表名称。

响应参数

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

示例

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

2.1.4.8.5. 获取通道的具体信息

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

请求参数

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

响应参数

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

示例

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

2.1.4.8.6. 删除通道

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

请求参数

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

响应参数

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

示例

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

2.1.5. 错误处理

介绍表格存储Java SDK的错误处理方式。

方式

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

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

异常

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

- ClientException：指SDK内部出现的异常，比如参数设置错误等。
- OTSException：指服务器端错误，来自于对服务器错误信息的解析。OTSException包含如下几个成员：
  - getHttpStatus()：HTTP返回码，比如200、404等。
  - getErrorCode()：表格存储返回的错误类型字符串。
  - getRequestId()：用于唯一标识此该次请求的UUID。当您无法解决问题时，记录此RequestId联系表格存储的开发工程师获取帮助。

2.2. Python-SDK

2.2.1. 前言

介绍表格存储Python SDK的安装和使用，本文内容适用于4.x.x版本。

前提条件

已获取一个授权账号以及一对AccessKey ID和AccessKey Secret。请参见获取Accesskey在Apsara Uni-manager运营控制台上获取和查看AccessKey。

SDK下载

通过GitHub下载，具体下载地址请参见SDK包。

兼容性

- 对5.x.x系列的SDK兼容。
- 对4.x.x系列的SDK兼容。
- 对2.x.x系列的SDK不兼容，原因是2.0系列版本中支持主键乱序，而4.0.0版本开始不允许主键乱序，涉及

的不兼容点包括：

- 包名称由ots2变更为tablestore。
- Client.create_table接口新增TableOptions参数。
- put_row、get_row、update_row等接口的primary_key参数由dict类型变更为list类型，目的是保证主键的顺序性。
- put_row、update_row等接口的attribute_columns参数由dict类型变更为list类型。
- put_row、update_row等接口的attribute_columns参数新增timestamp。
- get_row、get_range等接口新增max_version、time_range参数，这两个参数必须存在一个。
- put_row、update_row、delete_row等接口新增return_type参数，目前仅支持RT_PK，表示返回值中包含当前行PK值。
- put_row、update_row、delete_row等接口返回值新增return_row，如果在请求中指定了return_type为RT_PK，则return_row中包含此行的PK值。

版本

当前最新版本为5.2.1。

2.2.2. 安装

本文介绍如何安装表格存储Python SDK。

环境准备

安装表格存储Python SDK需使用Python 2和Python 3。

安装

- 方式一：通过pip安装。

  安装命令如下：

  ```
  sudo pip install tablestore
  ```

- 方式二：通过GitHub安装。

  如果没有安装git，请安装git后，执行如下命令。

  ```
  git clone https://github.com/aliyun/aliyun-tablestore-python-sdk.git
  sudo python setup.py  install
  ```

- 方式三：通过源码安装。

  i. 下载SDK包。

  ii. 解压SDK包后执行如下命令：

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

验证SDK

通过命令行输入python并按回车键，在Python环境下检查SDK的版本。

```
>>> import tablestore
>>> tablestore.__version__
'5.1.0'
```

卸载SDK

直接通过pip卸载。

```
sudo pip uninstall tablestore
```

2.2.3. 初始化

OTSClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、单行数据、多行数据等。

确定Endpoint

使用表格存储实例所在的地域地址，可以通过以下方式查询Endpoint：

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   访问地址即是该实例的 Endpoint。

初始化对接

要接入阿里云表格存储服务，需要拥有一个有效的AccessKey（包括AccessKey ID和AccessKey Secret）用来进行签名认证。

获取到AccessKey ID和AccessKey Secret后，使用表格存储的Endpoint进行初始化对接，示例如下。

- 接口

```
"""
初始化``OTSClient``实例。
``end_point``是表格存储服务的地址（例如'https://instance.cn-hangzhou.ots.aliyun.com:80'），必须以'https://'开头。
``access_key_id``是访问表格存储服务的AccessKey ID，通过官方网站申请或通过管理员获取。
``access_key_secret``是访问表格存储服务的AccessKey Secret，通过官方网站申请或通过管理员获取。
``instance_name``是要访问的实例名，通过官方网站控制台创建或通过管理员获取。
``sts_token``是访问表格存储服务的STS token，从阿里云STS服务获取，具有有效期，过期后需要重新获取。
``encoding``请求参数的字符串编码类型，默认值为utf8。
``socket_timeout``是连接池中每个连接的Socket超时，单位为秒，可以为int或float。默认值为50。
``max_connection``是连接池的最大连接数。默认值为50。
``logger_name``用来在请求中打印DEBUG日志，或者在出错时打印ERROR日志。
``retry_policy``定义了重试策略，默认的重试策略为DefaultRetryPolicy。你您可以继承RetryPolicy来实现自己的重试策略，详情请参见DefaultRetryPolicy的代码。
"""
class OTSClient(object):
    def __init__(self, endpoint, access_key_id, access_key_secret, instance_name, **kwargs):
```

- 示例

```
#########    设置日志文件名称和重试策略    #########
# 日志文件名称为table_store.log，重试策略是WriteRetryPolicy，会对写重试。
ots_client = OTSClient('endpoint', 'access_key_id', 'access_key_secret', 'instance_name',
logger_name = 'table_store.log',  retry_policy = WriteRetryPolicy())
#########    使用STS    #########
ots_client = OTSClient('endpoint', 'STS.K8h*******GB77', 'CkuDj******Wn6', 'instance_name
', sts_token = 'CAISjgJ1q6Ft5B2y********OFcsLLuw==')
```

HTTPS

- 从2.0.8版本开始支持HTTPS。
- OpenSSL版本最少为0.9.8j，推荐OpenSSL 1.0.2d。
- Python 2.0.8发布包中包含了certifi包直接安装使用。如果需要更新根证书请从根证书下载最新的根证书。

2.2.4. 使用手册

2.2.4.1. 表操作

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

创建表（CreateTable）

根据指定的表结构信息创建表。

> **说明**
> - 创建表后需要几秒钟进行加载，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。
> - 创建表时必须指定表的主键。主键包含1~4个主键列，每一个主键列都有名称和类型。

- 接口

```
        说明：根据指定表结构信息创建表。
        ``table_meta``是``tablestore.metadata.TableMeta``类的实例，它包含表名和PrimaryKey的schema。
        请参见``TableMeta``类的文档。当创建一个表后，通常需要等待几秒钟时间使partition load完成，才能进行各种操作。
        ``table_options``是``tablestore.metadata.TableOptions``类的实例，它包含time_to_live，max_version和max_time_deviation三个参数。
        ``reserved_throughput``是``tablestore.metadata.ReservedThroughput``类的实例，表示预留读写吞吐量。
        返回：无。
        def create_table(self, table_meta, reserved_throughput):
```

- 示例

  创建一个有2个主键列，数据保留1年（60*60*24*365=31536000秒），最大版本数3，写入时间戳偏移小于1天（86400秒），预留读写吞吐量为(0, 0)的表。

```
    # 创建主键列的schema，包括PK的个数、名称和类型。
    # 第一个PK列为整型，名称是pk0，该列同时也是分区键。
    # 第二个PK列为整型，名称是pk1。其他可选的类型包括STRING和BINARY，此处使用INTEGER。
    schema_of_primary_key = [('pk0', 'INTEGER'), ('pk1', 'INTEGER')]
    # 通过表名和主键列的schema创建一个tableMeta。
    table_meta = TableMeta('SampleTable', schema_of_primary_key)
    # 创建TableOptions，数据保留31536000秒，超过后自动删除；最大3个版本；写入时指定的版本值和当前标准时间相差不能超过1天。
    table_options = TableOptions(31536000, 3, 86400)
    # 设置预留读吞吐量为0，预留写吞吐量为0。
    reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
    # 调用client的create_table接口，如果没有抛出异常，则说明执行成功。
    try:
        ots_client.create_table(table_meta, table_options, reserved_throughput)
        print "create table succeeded"
    # 如果抛出异常，则说明执行失败，处理异常。
    except Exception:
        print "create table failed."
```

详细代码请参见CreateTable@GitHub。

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

- 接口

```
"""
说明：获取所有表名的列表。
返回：表名列表。
``table_list``表示获取的表名列表，类型为tuple，例如('MyTable1', 'MyTable2')。
"""
def list_table(self):
```

- 示例

  获取实例下的所有表名。

```
    try:
        list_response = ots_client.list_table()
        print 'table list: '
        for table_name in list_response:
            print table_name
        print "list table succeeded"
    except Exception:
        print "list table failed."
```

详细代码请参见ListTable@GitHub。

更新表（UpdateTable）

更新指定表的预留读吞吐量、预留写吞吐量、最大版本数等设置。

- 接口

```
    """
        说明：更新表属性，目前只支持修改预留读写吞吐量。
        ``table_name``是对应的表名。
        ``table_options``是``tablestore.metadata.TableOptions``类的示例，它包含time_to_live
，max_version和max_time_deviation三个参数。
        ``reserved_throughput``是``ots2.metadata.ReservedThroughput``类的实例，表示预留读写
吞吐量。
        返回：针对该表的预留读写吞吐量的最近上调时间、最近下调时间和当天下调次数。
        ``update_table_response``表示更新的结果，是ots2.metadata.UpdateTableResponse类的实例
。
        """
        def update_table(self, table_name, table_options, reserved_throughput):
```

- 示例

  更新表的最大版本数为5。

```
        # 设定新的预留读吞吐量为0，写吞吐量为0。
        reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
        # 创建TableOptions，数据保留31536000秒，超过后自动删除；最大5个版本；写入时指定的版本值和
当前标准时间相差不能超过1天。
        table_options = TableOptions(31536000, 5, 86400)
        try:
            # 调用接口更新表的预留读写吞吐量。
            ots_client.update_table('SampleTable', reserved_throughput)
            # 如果没有抛出异常，则说明执行成功。
            print "update table succeeded"
        except Exception:
            # 如果抛出异常，则说明执行失败，处理异常。
            print "update table failed"
```

详细代码请参见UpdateTable@GitHub。

查询表描述信息（DescribeTable）

查询指定表的结构信息和预留读/写吞吐量设置信息。

- 接口

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

- 示例

  获取表的描述信息。

```
    try:
        describe_response = ots_client.describe_table('myTable')
        # 如果没有抛出异常，则说明执行成功，打印如下表信息。
        print "describe table succeeded."
        print ('TableName: %s' % describe_response.table_meta.table_name)
        print ('PrimaryKey: %s' % describe_response.table_meta.schema_of_primary_key)
        print ('Reserved read throughput: %s' % describe_response.reserved_throughput_det
ails.capacity_unit.read)
        print ('Reserved write throughput: %s' % describe_response.reserved_throughput_de
tails.capacity_unit.write)
        print ('Last increase throughput time: %s' % describe_response.reserved_throughpu
t_details.last_increase_time)
        print ('Last decrease throughput time: %s' % describe_response.reserved_throughpu
t_details.last_decrease_time)
        print ('table options\'s time to live: %s' % describe_response.table_options.time
_to_live)
        print ('table options\'s max version: %s' % describe_response.table_options.max_v
ersion)
        print ('table options\'s max_time_deviation: %s' % describe_response.table_option
s.max_time_deviation)
    except Exception:
        # 如果抛出异常，则说明执行失败，处理异常。
        print "describe table failed."
```

详细代码请参见DescribeTable@GitHub。

删除表（DeleteTable）

删除本实例下指定的表。

- 接口

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

- 示例

  删除表。

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

详细代码请参见DeleteTable@GitHub。

2.2.4.2. 单行数据操作

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

插入一行数据（PutRow）

PutRow接口用于插入一行数据，如果原来该行已经存在，会覆盖原来的一行。

- 接口

```
        """
        说明：写入一行数据。返回本次操作消耗的CapacityUnit。
        ``table_name``是对应的表名。
        ``row``是行数据，包括主键和属性列。
        ``condition``表示执行操作前做条件检查，满足条件才执行，是tablestore.metadata.Condition
类的实例。目前只支持对行的存在性进行检查，检查条件包括：'IGNORE'，'EXPECT_EXIST'和'EXPECT_NOT_EXI
ST'。
        ``return_type``表示返回类型，是tablestore.metadata.ReturnType类的实例，目前仅支持返回P
rimaryKey，一般用于主键列自增中。
        返回：本次操作消耗的CapacityUnit和需要返回的行数据。
        ``consumed``表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
        ``return_row``表示返回的行数据，可能包括主键、属性列。
        """
        def put_row(self, table_name, row, condition = None, return_type = None)
```

- 示例

  插入一行数据。

  > **说明**
  >
  > - RowExistenceExpectation.IGNORE表示不管此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。
  > - RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。
  > - RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。
  > - 如下示例中属性列age的版本为1498184687000，此值是2017年06月23日，如果当前时间-max_time_deviation（创建表时指定）大于1498184687000时，则PutRow时会被禁止。

```
    ## 主键的第一个主键列是gid，值是整数1，第二个主键列是uid，值是整数101。
    primary_key = [('gid',1), ('uid',101)]
    ## 属性列包括四个：
    ##                          第一个属性列的名字是name，值是字符串John，版本没有指定，使用系统当前时间作为版本号。
    ##                          第二个属性列的名字是mobile，值是整数15100000000，版本没有指定，使用系统当前时间作为版本号。
    ##                          第三个属性列的名字是address，值是二进制的China，版本没有指定，使用系统当前时间作为版本号。
    ##                          第四个属性列的名字是age，值是29.7，版本号为1498184687。
    attribute_columns = [('name','John'), ('mobile',15100000000),('address', bytearray('China')),('female', False), ('age', 29.7, 1498184687000)]
    ## 通过primary_key和attribute_columns构造Row。
    row = Row(primary_key, attribute_columns)
    # 行条件检查为期望行不存在。如果行存在会出现Condition Update Failed报错。
    condition = Condition(RowExistenceExpectation.EXPECT_NOT_EXIST)
    try :
        # 调用put_row方法，如果没有指定ReturnType，则return_row为None。
        consumed, return_row = client.put_row(table_name, row, condition)
        # 打印此次请求消耗的写CU。
        print ('put row succeed, consume %s write cu.' % consumed.write)
    # 客户端异常，一般为参数错误或者网络异常。
    except OTSClientError as e:
        print "put row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.get_error_message())
    # 服务端异常，一般为参数错误或者流控错误。
    except OTSServiceError as e:
        print "put row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

详细代码请参见Put Row@Git Hub。

读取一行数据（GetRow）

GetRow接口用于读取一行数据。

读取的结果可能有如下两种：

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

```
        """
         说明：获取一行数据。
         ``table_name``是数据表名称。
         ``primary_key``是主键，类型为list。
         ``columns_to_get``是可选参数，表示要获取的列的名称列表，类型为list；如果不填写，表示获取所有列。
         ``column_filter``是可选参数，表示读取指定条件的行。
         ``max_version``是可选参数，表示最多读取的版本数，max_version和time_range必须至少设置一个。
         ``time_range``是可选参数，表示读取版本号范围或特定版本号的数据，max_version和time_range必须至少设置一个。
         返回：本次操作消耗的CapacityUnit、主键列和属性列。
         ``consumed``表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
         ``return_row``表示行数据，包括主键列和属性列，类型都为list，例如[('PK0',value0), ('PK1',value1)]。
         ``next_token``表示宽行读取时下一次读取的位置，编码的二进制。
         """
         def get_row(self, table_name, primary_key, columns_to_get=None,
                 column_filter=None, max_version=None, time_range=None,
                 start_column=None, end_column=None, token=None):
```

- 示例

  读取一行数据。

```
    # 主键的第一列是uid，值是整数1，第二列是gid，值是整数101。
    primary_key = [('uid',1), ('gid',101)]
    # 需要返回的属性列name、growth、type。如果columns_to_get为[]，则返回所有属性列。
    columns_to_get = ['name', 'growth', 'type']
    # 设置过滤器，增加列filter，当growth列的值不等于0.9且name列的值等于'杭州'时，则返回该行。
    cond = CompositeColumnCondition(LogicalOperator.AND)
    cond.add_sub_condition(SingleColumnCondition("growth", 0.9, ComparatorType.NOT_EQUAL)
)
    cond.add_sub_condition(SingleColumnCondition("name", '杭州', ComparatorType.EQUAL))
    try:
        # 调用get_row接口查询，最后一个参数值1表示只需要返回一个版本的值。
        consumed, return_row, next_token = client.get_row(table_name, primary_key, column
s_to_get, cond, 1)
        print ('Read succeed, consume %s read cu.' % consumed.read)
        print ('Value of primary key: %s' % return_row.primary_key)
        print ('Value of attribute: %s' % return_row.attribute_columns)
        for att in return_row.attribute_columns:
            # 打印每一列的key、value和version值。
            print ('name:%s\tvalue:%s\ttimestamp:%d' % (att[0], att[1], att[2]))
    # 客户端异常，一般为参数错误或者网络异常。
    except OTSClientError as e:
        print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(),
e.get_error_message())
    # 服务端异常，一般为参数错误或者流控错误。
    except OTSServiceError as e:
        print "get row failed, http_status:%d, error_code:%s, error_message:%s, request_i
d:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id
())
```

详细代码请参见Get Row@Git Hub。

更新一行数据（UpdateRow）

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

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

- 接口

```
        """
        说明：更新一行数据。
        ``table_name``是数据表名称。
        ``row``表示更新的行数据，包括主键列和属性列，主键列和属性列的类型均是list。
        ``condition``表示执行操作前做条件检查，满足条件才执行，是tablestore.metadata.Condition
类的实例。支持对行的存在性和列条件进行检查，其中行存在性检查条件包括'IGNORE'、'EXPECT_EXIST'和'EXP
ECT_NOT_EXIST'。
        ``return_type``表示返回类型，是tablestore.metadata.ReturnType类的实例。目前仅支持返回P
rimaryKey，一般用于主键列自增中。
        返回：本次操作消耗的CapacityUnit和需要返回的行数据return_row。
        consumed表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
        return_row表示需要返回的行数据。
        """
        def update_row(self, table_name, row, condition, return_type = None)
```

- 示例

  更新一行数据。

```
    # 主键的第一列是uid，值是整数1，第二列是gid，值是整数101。
    primary_key = [('uid',1), ('gid',101)]
    # 更新包括PUT，DELETE和DELETE_ALL三部分。
    # PUT：新增或者更新列值。示例中是新增两列，第一列名字是name，值是David，第二列名字是address，
值是Hongkong。
    # DELETE：删除指定版本号（时间戳）的值。示例中是删除版本为1488436949003的address列的值。
    # DELETE_ALL：删除列。示例中是删除mobile和age两列的所有版本的值。
    update_of_attribute_columns = {
        'PUT' : [('name','David'), ('address','Hongkong')],
        'DELETE' : [('address', None, 1488436949003)],
        'DELETE_ALL' : [('mobile'), ('age')],
    }
    row = Row(primary_key, update_of_attribute_columns)
    # 行条件检查为忽略，无论行是否存在，均会更新。
    condition = Condition(RowExistenceExpectation.IGNORE, SingleColumnCondition("age", 20
, ComparatorType.EQUAL)) # update row only when this row is exist
    try:
        consumed, return_row = client.update_row(table_name, row, condition)
    # 客户端异常，一般为参数错误或者网络异常。
    except OTSClientError as e:
        print "update row failed, http_status:%d, error_message:%s" % (e.get_http_status(
), e.get_error_message())
    # 服务端异常，一般为参数错误或者流控错误。
    except OTSServiceError as e:
        print "update row failed, http_status:%d, error_code:%s, error_message:%s, reques
t_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request
_id())
```

详细代码请参见UpdateRow@GitHub。

删除一行数据（DeleteRow）

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

- 接口

```
        """
        说明：删除一行数据。
        ``table_name``是数据表名称。
        ``row``表示行数据，在delete_row中仅包含主键。
        ``condition``表示执行操作前做条件检查，满足条件才执行，是tablestore.metadata.Condition
类的实例。支持对行的存在性和列条件进行检查，其中行存在性检查条件包括'IGNORE'、'EXPECT_EXIST'和'EXP
ECT_NOT_EXIST'。
        返回：本次操作消耗的CapacityUnit和需要返回的行数据return_row。
        consumed表示消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
        return_row表示需要返回的行数据。
        """
        def delete_row(self, table_name, row, condition, return_type = None):
```

- 示例

  删除一行数据。

```
    primary_key = [('gid',1), ('uid','101')]
    row = Row(primary_key)
    try:
        consumed, return_row = client.delete_row(table_name, row, None)
    # 客户端异常，一般为参数错误或者网络异常。
    except OTSClientError as e:
        print "update row failed, http_status:%d, error_message:%s" % (e.get_http_status(
), e.get_error_message())
    # 服务端异常，一般为参数错误或者流控错误。
    except OTSServiceError as e:
        print "update row failed, http_status:%d, error_code:%s, error_message:%s, reques
t_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request
_id())
    print ('Delete succeed, consume %s write cu.' % consumed.write)
```

  详细代码请参见DeleteRow@GitHub。

2.2.4.3. 多行数据操作

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

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

BatchGetRow操作可视为多个GetRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

与执行大量的GetRow操作相比，使用BatchGetRow操作可以有效减少请求的响应时间，提高数据的读取速率。

- 接口

```
        """
        说明：批量获取多行数据。
        request = BatchGetRowRequest()
        request.add(TableInBatchGetRowItem(myTable0, primary_keys, column_to_get=None, co
lumn_filter=None))
        request.add(TableInBatchGetRowItem(myTable1, primary_keys, column_to_get=None, co
lumn_filter=None))
        request.add(TableInBatchGetRowItem(myTable2, primary_keys, column_to_get=None, co
lumn_filter=None))
        request.add(TableInBatchGetRowItem(myTable3, primary_keys, column_to_get=None, co
lumn_filter=None))
        response = client.batch_get_row(request)
        ``response``为返回的结果，类型为tablestore.metadata.BatchGetRowResponse。
        """
        def batch_get_row(self, request):
```

- 示例

  批量一次读3行。

```
    # 设置需要返回的列。
    columns_to_get = ['name', 'mobile', 'address', 'age']
    # 读取3行。
    rows_to_get = []
    for i in range(0, 3):
        primary_key = [('gid',i), ('uid',i+1)]
        rows_to_get.append(primary_key)
    # 过滤条件为name等于John，且address等于China。
    cond = CompositeColumnCondition(LogicalOperator.AND)
    cond.add_sub_condition(SingleColumnCondition("name", "John", ComparatorType.EQUAL))
    cond.add_sub_condition(SingleColumnCondition("address", 'China', ComparatorType.EQUAL
))
    # 构造批量读请求。
    request = BatchGetRowRequest()
    # 增加表table_name中需要读取的行，最后一个参数1表示读取最新的一个版本。
    request.add(TableInBatchGetRowItem(table_name, rows_to_get, columns_to_get, cond, 1))
    # 增加表notExistTable中需要读取的行。
    request.add(TableInBatchGetRowItem('notExistTable', rows_to_get, columns_to_get, cond
, 1))
    try:
           result = client.batch_get_row(request)
        print ('Result status: %s'%(result.is_all_succeed()))
        table_result_0 = result.get_result_by_table(table_name)
        table_result_1 = result.get_result_by_table('notExistTable')
        print ('Check first table\'s result:')
        for item in table_result_0:
            if item.is_ok:
                print ('Read succeed, PrimaryKey: %s, Attributes: %s' % (item.row.primary
_key, item.row.attribute_columns))
            else:
                print ('Read failed, error code: %s, error message: %s' % (item.error_cod
e, item.error_message))
        print ('Check second table\'s result:')
        for item in table_result_1:
            if item.is_ok:
                print ('Read succeed, PrimaryKey: %s, Attributes: %s' % (item.row.primary
_key, item.row.attribute_columns))
            else:
                print ('Read failed, error code: %s, error message: %s' % (item.error_cod
e, item.error_message))
    # 客户端异常，一般为参数错误或者网络异常。
    except OTSClientError as e:
        print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(),
e.get_error_message())
    # 服务端异常，一般为参数错误或者流控错误。
    except OTSServiceError as e:
        print "get row failed, http_status:%d, error_code:%s, error_message:%s, request_i
d:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id
())
```

代码详情请参见BatchGetRow@GitHub。

批量写（BatchWriteRow）

批量插入、修改或删除一个或多个表中的若干行数据。

BatchWriteRow操作可视为多个PutRow、UpdateRow、DeleteRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

- 接口

```
        """
        说明：批量修改多行数据。
        request = MiltiTableInBatchWriteRowItem()
        request.add(TableInBatchWriteRowItem(table0, row_items))
        request.add(TableInBatchWriteRowItem(table1, row_items))
        response = client.batch_write_row(request)
        ``response``为返回的结果，类型为tablestore.metadata.BatchWriteRowResponse。
        """
        def batch_write_row(self, request):
```

- 示例

  批量写数据。

```
    put_row_items = []
    # 增加PutRow的行。
    for i in range(0, 10):
        primary_key = [('gid',i), ('uid',i+1)]
        attribute_columns = [('name','somebody'+str(i)), ('address','somewhere'+str(i)),
('age',i)]
        row = Row(primary_key, attribute_columns)
        condition = Condition(RowExistenceExpectation.IGNORE)
        item = PutRowItem(row, condition)
        put_row_items.append(item)
    # 增加UpdateRow的行。
    for i in range(10, 20):
        primary_key = [('gid',i), ('uid',i+1)]
        attribute_columns = {'put': [('name','somebody'+str(i)), ('address','somewhere'+s
tr(i)), ('age',i)]}
        row = Row(primary_key, attribute_columns)
        condition = Condition(RowExistenceExpectation.IGNORE, SingleColumnCondition("age"
, i, ComparatorType.EQUAL))
        item = UpdateRowItem(row, condition)
        put_row_items.append(item)
    # 增加DeleteRow的行。
    delete_row_items = []
    for i in range(10, 20):
        primary_key = [('gid',i), ('uid',i+1)]
        row = Row(primary_key)
        condition = Condition(RowExistenceExpectation.IGNORE)
        item = DeleteRowItem(row, condition)
        delete_row_items.append(item)
    # 构造批量写请求。
    request = BatchWriteRowRequest()
    request.add(TableInBatchWriteRowItem(table_name, put_row_items))
    request.add(TableInBatchWriteRowItem('notExistTable', delete_row_items))
```

```
    # 调用batch_write_row方法执行批量写，如果请求参数等错误会抛异常；如果部分行失败，则不会抛异常，但是内部的Item会失败。
    try:
        result = client.batch_write_row(request)
        print ('Result status: %s'%(result.is_all_succeed()))
        # 检查Put行的结果。
        print ('check first table\'s put results:')
        succ, fail = result.get_put()
        for item in succ:
            print ('Put succeed, consume %s write cu.' % item.consumed.write)
        for item in fail:
            print ('Put failed, error code: %s, error message: %s' % (item.error_code, item.error_message))
        # 检查Update行的结果。
        print ('check first table\'s update results:')
        succ, fail = result.get_update()
        for item in succ:
            print ('Update succeed, consume %s write cu.' % item.consumed.write)
        for item in fail:
            print ('Update failed, error code: %s, error message: %s' % (item.error_code, item.error_message))
        # 检查Delete行的结果。
        print ('check second table\'s delete results:')
        succ, fail = result.get_delete()
        for item in succ:
            print ('Delete succeed, consume %s write cu.' % item.consumed.write)
        for item in fail:
            print ('Delete failed, error code: %s, error message: %s' % (item.error_code, item.error_message))
    # 客户端异常，一般为参数错误或者网络异常。
    except OTSClientError as e:
        print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.get_error_message())
    # 服务端异常，一般为参数错误或者流控错误。
    except OTSServiceError as e:
        print "get row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

代码详情请参见BatchWriteRow@GitHub。

范围读（GetRange）

读取指定主键范围内的数据。

- 接口

```
        """
        说明：根据范围条件获取多行数据。
        ``table_name``是对应的表名。
        ``direction``表示范围的方向，字符串格式，取值包括'FORWARD'和'BACKWARD'。
        ``inclusive_start_primary_key``表示范围的起始主键（在范围内）。
        ``exclusive_end_primary_key``表示范围的结束主键（不在范围内）。
        ``columns_to_get``是可选参数，表示要获取的列的名称列表，类型为list；如果不填，表示获取所有列。
        ``limit``是可选参数，表示最多读取多少行；如果不填，则没有限制。
        ``column_filter``是可选参数，表示读取指定条件的行。
        ``max_version``是可选参数，表示返回的最大版本数目，与time_range必须存在一个。
        ``time_range``是可选参数，表示返回的版本的范围，于max_version必须存在一个。
        ``start_column``是可选参数，用于宽行读取，表示本次读取的起始列。
        ``end_column``是可选参数，用于宽行读取，表示本次读取的结束列。
        ``token``是可选参数，用于宽行读取，表示本次读取的起始列位置，内容被二进制编码，来源于上次请求的返回结果中。
        返回：符合条件的结果列表。
        ``consumed``表示本次操作消耗的CapacityUnit，是tablestore.metadata.CapacityUnit类的实例。
        ``next_start_primary_key``表示下次get_range操作的起始点的主健列，类型为dict。
        ``row_list``表示本次操作返回的行数据列表，格式为：[Row, ...]。
        """
    def get_range(self, table_name, direction,
                  inclusive_start_primary_key,
                  exclusive_end_primary_key,
                  columns_to_get=None,
                  limit=None,
                  column_filter=None,
                  max_version=None,
                  time_range=None,
                  start_column=None,
                  end_column=None,
                  token = None):
```

- 示例

  范围读取。

```
    # 设置范围查询的起始主键。
    inclusive_start_primary_key = [('uid',INF_MIN), ('gid',INF_MIN)]
    # 设置范围查询的结束主键。
    exclusive_end_primary_key = [('uid',INF_MAX), ('gid',INF_MAX)]
    # 查询所有列。
    columns_to_get = []
    # 每次最多返回90行，如果总共有100个结果，首次查询时指定limit=90，则第一次最多返回90，最少可能返回0个结果，但是next_start_primary_key不为None。
    limit = 90
    # 设置过滤器。
    cond = CompositeColumnCondition(LogicalOperator.AND)
    cond.add_sub_condition(SingleColumnCondition("address", 'China', ComparatorType.EQUAL))
    cond.add_sub_condition(SingleColumnCondition("age", 50, ComparatorType.LESS_THAN))
    try:
        # 调用get_range接口。
        consumed, next_start_primary_key, row_list, next_token  = client.get_range(
                table_name, Direction.FORWARD,
                inclusive_start_primary_key, exclusive_end_primary_key,
                columns_to_get,
                limit,
                column_filter = cond,
                max_version = 1
                time_range = (1557125059000, 1557129059000) # start_time大于等于1557125059000，end_time小于1557129059000。
    )
        all_rows = []
        all_rows.extend(row_list)
        # 当next_start_primary_key不为空时，说明还有数据，继续循环读取。
        while next_start_primary_key is not None:
            inclusive_start_primary_key = next_start_primary_key
            consumed, next_start_primary_key, row_list, next_token = client.get_range(
                table_name, Direction.FORWARD,
                inclusive_start_primary_key, exclusive_end_primary_key,
                columns_to_get, limit,
                column_filter = cond,
                max_version = 1
        )
            all_rows.extend(row_list)
        # 打印主键和属性列。
        for row in all_rows:
            print (row.primary_key, row.attribute_columns)
        print ('Total rows: ', len(all_rows))
    # 客户端异常，一般为参数错误或者网络异常。
    except OTSClientError as e:
        print "get row failed, http_status:%d, error_message:%s" % (e.get_http_status(), e.get_error_message())
    # 服务端异常，一般为参数错误或者流控错误。
    except OTSServiceError as e:
        print "get row failed, http_status:%d, error_code:%s, error_message:%s, request_id:%s" % (e.get_http_status(), e.get_error_code(), e.get_error_message(), e.get_request_id())
```

代码详情请参见GetRange@GitHub。

2.2.4.4. 全局二级索引

介绍全局二级索引的创建和删除操作。

创建全局二级索引

在创建数据表时同时创建全局二级索引或者创建数据表后单独创建全局二级索引，请根据实际情况创建全局二级索引。

全局二级索引中的字段需要在创建表时通过defined_columns预先指定。

- 创建数据表同时创建全局二级索引

```
schema_of_primary_key = [('gid', 'INTEGER'), ('uid', 'STRING')]
defined_columns = [('i', 'INTEGER'), ('bool', 'BOOLEAN'), ('d', 'DOUBLE'), ('s', 'STRING'
), ('b', 'BINARY')]
table_meta = TableMeta(table_name, schema_of_primary_key, defined_columns)
table_option = TableOptions(-1, 1)
reserved_throughput = ReservedThroughput(CapacityUnit(0, 0))
secondary_indexes = [
    SecondaryIndexMeta('index1', ['i', 's'], ['bool', 'b', 'd']),
    ]
client.create_table(table_meta, table_option, reserved_throughput, secondary_indexes)
```

- 单独创建全局二级索引

```
index_meta = SecondaryIndexMeta('index2', ['i', 's'], ['bool', 'b', 'd'])
client.create_secondary_index(table_name, index_meta)
```

删除索引

删除索引表。

```
client.delete_secondary_index(table_name, 'index1')
```

2.2.5. 错误处理

介绍表格存储Python SDK的错误处理方式和重试策略。

方式

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

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

异常

表格存储Python SDK中有OTSClientError和OTSServiceError两种异常，都最终继承自Exception。

- OTSClientError：指SDK内部出现的异常，例如参数设置错误，返回结果解析失败等。
- OTSServiceError：指服务器端错误，来自于对服务器错误信息的解析。OTSServiceError包含以下几个成员：
  - get_http_status：HTTP返回码，例如200、404等。
  - get_error_code：表格存储返回的错误类型字符串。
  - get_error_message：表格存储返回的错误消息字符串。
  - get_request_id：用于唯一标识此次请求的UUID。当您无法解决问题时，记录此RequestId联系表格存储的开发工程师获取帮助。

重试

- SDK中出现错误时会自动重试。默认策略是最大重试次数为20，最大重试间隔为3000毫秒。对流控类错误以及读操作相关的服务端内部错误进行的重试，请参见tablestore/retry.py。
- 您也可以通过继承RetryPolicy类实现自定义重试策略，在构造OTSClient对象时，将自定义的重试策略作为参数传入。

目前SDK中已经实现的重试策略如下。

- DefaultRetryPolicy：默认重试策略，只会对读操作重试，最大重试次数为20，最大重试间隔为3000毫秒。
- NoRetryPolicy：不进行任何重试。
- NoDelayRetryPolicy：没有延时的重试策略，请谨慎使用。
- WriteRetryPolicy：在默认重试策略基础上，会对写操作重试。

2.3. Go-SDK

2.3.1. 前言

本文介绍表格存储Go SDK的安装和使用。

前提条件

已获取一个授权账号以及一对AccessKey ID和AccessKey Secret。请参见获取Accesskey在Apsara Uni-manager运营控制台上获取和查看AccessKey。

下载及安装

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

版本

当前最新版本：5.0.2

2.3.2. 安装

介绍Go SDK的安装。

环境准备

安装Go SDK需使用Go 1.4及以上版本。

安装方式

安装命令如下。

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

2.3.3. 初始化

TableStoreClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、单行数据、多行数据等。

确定Endpoint

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

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

配置密钥

要接入阿里云表格存储服务，需要拥有一个有效的AccessKey（包括AccessKey ID和AccessKey Secret）用来进行签名认证。

获取到AccessKey ID和AccessKey Secret后，使用表格存储的Endpoint进行初始化对接，示例如下。

- 接口

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

- 示例

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

2.3.4. 使用手册

2.3.4.1. 表操作

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

创建表（CreateTable）

根据给定的表的结构信息创建相应的表。

> **说明**
> - 创建表后需要几秒钟进行加载，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。
> - 创建表时必须指定表的主键。主键包含1~4个主键列，每一个主键列都有名称和类型。

- 接口

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

- 示例

  创建一个含有2个主键列，预留读/写吞吐量为(0, 0)的表。

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

  代码详情请参见CreateTable@GitHub。

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

- 接口

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

- 示例

  获取实例下的所有表名。

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

  详细代码请参见ListTable@GitHub。

更新表（UpdateTable）

更新指定表的预留读吞吐量或预留写吞吐量的设置。

- 接口

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

- 示例

  更新表的最大版本数为5。

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

  详细代码请参见UpdateTable@GitHub。

查询表描述信息（DescribeTable）

查询指定表的结构信息和预留读/写吞吐量的设置信息。

- 接口

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

- 示例

  获取表的描述信息。

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

  详细代码请参见DescribeTable@GitHub

删除表（DeleteTable）

删除本实例下指定的表。

- 接口

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

- 示例

  删除表。

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

  详细代码请参见DeleteTable@GitHub。

2.3.4.2. 单行数据操作

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

插入一行数据（PutRow）

插入数据到指定的行。

- 接口

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

- 示例

  插入一行数据。

  > **说明**
  > - RowExistenceExpectation.IGNORE表示不管此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。
  > - RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。
  > - RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。

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

详细代码请参见PutRow@GitHub。

读取一行数据（GetRow）

根据给定的主键读取单行数据。

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

- 示例

  读取一行数据。

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

详细代码请参见GetRow@GitHub。

更新一行数据（UpdateRow）

更新指定行的数据，如果该行不存在，则新增一行；若该行存在，则根据请求的内容在这一行中新增、修改或者删除指定列的值。

- 接口

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

- 示例

  更新一行数据。

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

详细代码请参见UpdateRow@GitHub

删除一行数据（DeleteRow）

删除不需要的数据。

- 接口

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

- 示例

  删除一行数据。

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

详细代码请参见DeleteRow@GitHub。

2.3.4.3. 多行数据操作

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

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

BatchGetRow操作可视为多个GetRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

与执行大量的GetRow操作相比，使用BatchGetRow操作可以有效减少请求的响应时间，提高数据的读取速率。

- 接口

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

- 示例

  批量一次读10行。

  > **说明** 批量读也支持通过条件语句过滤。

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

详细代码请参见BatchGetRow@GitHub。

批量写（BatchWriteRow）

批量插入、修改或删除一个或多个表中的若干行数据。

BatchWriteRow操作可视为多个PutRow、UpdateRow、DeleteRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

- 接口

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

- 示例

  批量写入100行数据。

  > **说明** 批量写也支持通过条件语句过滤。

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

详细代码请参见BatchWriteRow@GitHub。

范围读（GetRange）

读取指定主键范围内的数据。

- 接口

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

- 示例

  按照范围读取数据。

  > **说明**
  > - 按范围读取也支持通过条件语句过滤。
  > - 按范围读取需要注意数据可能会分页。

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

详细代码请参见Get Range@Git Hub。

2.3.4.4. 通道服务

本文主要介绍如何初始化Tunnel client。

2.3.4.4.1. 安装

介绍通道服务的安装。

下载源码包

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

安装依赖

- 在tunnel目录下使用dep安装依赖。
  - 安装dep
  - dep ensure -v
- 直接使用go get安装依赖包。

```
go get -u go.uber.org/zap
go get github.com/cenkalti/backoff
go get github.com/golang/protobuf/proto
go get github.com/satori/go.uuid
go get github.com/stretchr/testify/assert
go get github.com/smartystreets/goconvey/convey
go get github.com/golang/mock/gomock
go get gopkg.in/natefinch/lumberjack.v2
```

2.3.4.4.2. 快速开始

介绍如何使用Go SDK快速体验通道服务、获取通道信息和删除通道。

体验通道服务

使用Go SDK快速体验通道服务。

1. 初始化Tunnel client。

```
//endpoint是表格存储实例endpoint，例如https://instance.cn-hangzhou.ots.aliyun.com。
//instance是实例名称。
//accessKeyId和accessKeySecret分别为访问表格存储服务的AccessKey的Id和Secret。
tunnelClient := tunnel.NewTunnelClient(endpoint, instance,
   accessKeyId, accessKeySecret)
```

2. 创建通道。

```
req := &tunnel.CreateTunnelRequest{
   TableName:  "testTable",
   TunnelName: "testTunnel",
   Type:       tunnel.TunnelTypeBaseStream, //创建全量加增量类型的Tunnel。
}
resp, err := tunnelClient.CreateTunnel(req)
if err != nil {
   log.Fatal("create test tunnel failed", err)
}
log.Println("tunnel id is", resp.TunnelId)
```

3. 用户自定义数据消费Callback，开始自动化的数据消费。

```
//用户定义消费callback函数。
func exampleConsumeFunction(ctx *tunnel.ChannelContext, records []*tunnel.Record) error
{
    fmt.Println("user-defined information", ctx.CustomValue)
    for _, rec := range records {
        fmt.Println("tunnel record detail:", rec.String())
    }
    fmt.Println("a round of records consumption finished")
    return nil
}
//配置callback到SimpleProcessFactory，配置消费端TunnelWorkerConfig。
workConfig := &tunnel.TunnelWorkerConfig{
   ProcessorFactory: &tunnel.SimpleProcessFactory{
      CustomValue: "user custom interface{} value",
      ProcessFunc: exampleConsumeFunction,
   },
}
//使用TunnelDaemon持续消费指定tunnel。
daemon := tunnel.NewTunnelDaemon(tunnelClient, tunnelId, workConfig)
log.Fatal(daemon.Run())
```

创建通道

```
req := &tunnel.CreateTunnelRequest{
   TableName:  "testTable",
   TunnelName: "testTunnel",
   Type:       tunnel.TunnelTypeBaseStream, //创建全量加增量类型的Tunnel。
}
resp, err := tunnelClient.CreateTunnel(req)
if err != nil {
   log.Fatal("create test tunnel failed", err)
}
log.Println("tunnel id is", resp.TunnelId)
```

获取通道的具体信息

```
req := &tunnel.DescribeTunnelRequest{
   TableName:  "testTable",
   TunnelName: "testTunnel",
}
resp, err := tunnelClient.DescribeTunnel(req)
if err != nil {
   log.Fatal("describe test tunnel failed", err)
}
log.Println("tunnel id is", resp.Tunnel.TunnelId)
```

删除通道

```
req := &tunnel.DeleteTunnelRequest {
   TableName: "testTable",
   TunnelName: "testTunnel",
}
_, err := tunnelClient.DeleteTunnel(req)
if err != nil {
   log.Fatal("delete test tunnel failed", err)
}
```

2.3.4.4.3. 配置项

介绍通道服务的配置项。

tunnel client

初始化tunnel client时可以通过NewTunnelClientWithConfig接口自定义客户端配置，使用不指定config初始化接口或者config为nil时会使用DefaultTunnelConfig。

```
var DefaultTunnelConfig = &TunnelConfig{
      //最大指数退避重试时间。
      MaxRetryElapsedTime: 45 * time.Second,
      //HTTP请求超时时间。
      RequestTimeout:      30 * time.Second,
      //http.DefaultTransport。
      Transport:           http.DefaultTransport,
}
```

数据消费worker

TunnelWorkerConfig中包含了数据消费worker需要的配置，其中ProcessorFactory为必填项，其余参数如果不填写将使用默认值，通常使用默认值即可。

```
type TunnelWorkerConfig struct {
   //worker同Tunnel服务的心跳超时时间，通常使用默认值即可。
   HeartbeatTimeout  time.Duration
   //worker发送心跳的频率，通常使用默认值即可。
   HeartbeatInterval time.Duration
   //tunnel下消费连接建立接口，通常使用默认值即可。
   ChannelDialer     ChannelDialer
   //消费连接上具体处理器产生接口，通常使用callback函数初始化SimpleProcessFactory即可。
   ProcessorFactory ChannelProcessorFactory
   //zap日志配置，默认值为DefaultLogConfig。
   LogConfig      *zap.Config
   //zap日志轮转配置，默认值为DefaultSyncer。
   LogWriteSyncer zapcore.WriteSyncer
}
```

其中ProcessorFactory为用户注册消费callback函数以及其他信息的接口，建议使用SDK中自带SimpleProcessorFactory实现。

```
type SimpleProcessFactory struct {
   //用户自定义信息，会传递到ProcessFunc和ShutdownFunc中的ChannelContext参数中。
   CustomValue interface{}
   //Worker记录checkpoint的间隔，CpInterval<=0时会使用DefaultCheckpointInterval。
   CpInterval time.Duration
   //worker数据处理的同步调用callback，ProcessFunc返回error时worker会用本批数据退避重试ProcessFun
C。
   ProcessFunc  func(channelCtx *ChannelContext, records []*Record) error
   //worker退出时的同步调用callback。
   ShutdownFunc func(channelCtx *ChannelContext)
   //日志配置，Logger为nil时会使用DefaultLogConfig初始化logger。
   Logger *zap.Logger
}
```

日志

默认日志配置和日志轮转配置示例如下：

- 默认日志配置

```
//DefaultLogConfig是TunnelWorkerConfig和SimpleProcessFactory使用的默认日志配置。
var DefaultLogConfig = zap.Config{
   Level:       zap.NewAtomicLevelAt(zap.InfoLevel),
   Development: false,
   Sampling: &zap.SamplingConfig{
      Initial:    100,
      Thereafter: 100,
   },
   Encoding: "json",
   EncoderConfig: zapcore.EncoderConfig{
      TimeKey:        "ts",
      LevelKey:       "level",
      NameKey:        "logger",
      CallerKey:      "caller",
      MessageKey:     "msg",
      StacktraceKey:  "stacktrace",
      LineEnding:     zapcore.DefaultLineEnding,
      EncodeLevel:    zapcore.LowercaseLevelEncoder,
      EncodeTime:     zapcore.ISO8601TimeEncoder,
      EncodeDuration: zapcore.SecondsDurationEncoder,
      EncodeCaller:   zapcore.ShortCallerEncoder,
   },
}
```

- 日志轮转配置

```
//DefaultSyncer是TunnelWorkerConfig和SimpleProcessFactory使用的默认日志轮转配置
var DefaultSyncer = zapcore.AddSync(&lumberjack.Logger{
   //日志文件路径。
   Filename:   "tunnelClient.log",
   //最大日志文件大小。
   MaxSize:    512, //MB
   //压缩轮转的日志文件数。
   MaxBackups: 5,
   //轮转日志文件保留的最大天数。
   MaxAge:     30, //days
   //是否压缩轮转日志文件。
   Compress:   true,
})
```

2.3.4.4.4. 错误处理

介绍通道服务的错误处理。

通道服务在收到异常请求后，会返回protobuf格式错误以及HTTP状态码。

格式定义

protobuf错误格式如下：

```
message Error {
    required string code = 1;
    optional string message = 2;
    optional string tunnel_id = 3;
}
```

错误码

使用SDK时，只需要关心处理逻辑为“返回错误”的错误码，其余错误码会被SDK自动处理或重试，建议直接按相应处理逻辑处理错误码。

HTTP状态码	错误码	描述	处理逻辑
400	OTSParameterInvalid	API请求参数错误或数据表不存在。	返回错误。
400	OTSTunnelExpired	增量Tunnel或全量加增量Tunnel日志过期。	返回错误。
403	OTSPermissionDenied	无指定资源的访问权限。	返回错误。
409	OTSTunnelExist	待创建的Tunnel已经在服务端存在。	返回错误。
400	OTSSequenceNumberNotMatch	checkpoint序列号不匹配，通常是序列号滞后或者Channel消费有竞争。	通过checkpoint API重新获取checkpoint和序列号。

HTTP状态码	错误码	描述	处理逻辑
410	OTSResourceGone	Tunnel Client心跳超时。	使用TunnelID重新连接Tunnel Service。
503	OTSTunnelServerUnavailable	Tunnel服务内部错误。	退避重试。

2.4. NodeJS-SDK

2.4.1. 前言

本文介绍表格存储Node.js SDK的使用。本文内容适用于4.x.x及以上版本。

前提条件

已获取一个授权账号以及一对AccessKey ID和AccessKey Secret。请参见获取Accesskey在Apsara Uni-manager运营控制台上获取和查看AccessKey。

SDK下载

通过GitHub下载，具体下载路径请参见GitHub。

版本

当前最新版本：5.1.0

2.4.2. 安装

介绍如何安装表格存储Node.js SDK。

环境准备

适用于Node.js 4.0及以上版本。

> **说明** 由于兼容性问题，建议您不要使用Node.js 12.0版本~12.14版本。

安装

安装命令如下。

```
npm install tablestore
```

> **说明** 如果使用npm遇到网络问题，可以使用淘宝提供的npm镜像。

示例程序

Node.js SDK提供丰富的示例程序，方便参考或直接使用。您可以通过以下两种方式获取示例程序。

- 下载表格存储Node.js SDK开发包，解压后examples为示例程序。
- 访问表格存储Node.js SDK的GitHub项目。

2.4.3. 初始化

TableStore.Client是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、单行数据、多行数据等。

确定Endpoint

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

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

初始化对接

要接入阿里云表格存储服务，需要拥有一个有效的AccessKey（包括AccessKey ID和AccessKey Secret）用来进行签名认证。

获取到AccessKey ID和AccessKey Secret后，使用表格存储的Endpoint进行初始化对接，示例如下。

```
var client = new TableStore.Client({
accessKeyId: '<your access key id>',
accessKeySecret: '<your access key secret>',
endpoint: '<your endpoint>',
instancename: '<your instance name>',
maxRetries:20,//默认20次重试，可以省略此参数。
});
```

2.4.4. 数据类型

介绍表格存储提供的五种数据类型与Node.js SDK数据类型的对应关系。

表格存储数据类型	Node.js SDK数据类型	描述
String	string	JavaScript语言中的基本数据类型
Integer	int64	Node.js SDK封装的数据类型
Dobule	number	JavaScript语言中的基本数据类型
Boolean	boolean	JavaScript语言中的基本数据类型
Binary	Buffer	Node.js的Buffer对象

表格存储的Integer类型是一个64位的有符号整型，此数据类型在JavaScript中没有相应的数据类型可以对应，所以在Node.js中需要一个能表示64位有符号整型的数据类型，可以对表格存储的Integer类型做如下转换。

```
var numberA = TableStore.Long.fromNumber(1000);
var numberB = TableStore.Long.fromString('2000');
var num = numberA.toNumber();
    num = numberA.toString();
var str = numberB.toNumber();
    str = numberB.toString();
```

2.4.5. 使用手册

2.4.5.1. 表操作

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

创建表（CreateTable）

根据给定的表的结构信息创建相应的表。

> **说明**
> - 创建表后需要几秒钟进行加载，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。
> - 创建表时必须指定表的主键。主键包含1~4个主键列，每一个主键列都有名称和类型。

- 接口

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

- 示例

  创建一个有2个主键列，预留读/写吞吐量为(0, 0)的表。

```
var client = require('./client');
var params = {
  tableMeta: {
    tableName: 'sampleTable',
    primaryKey: [
      {
        name: 'gid',
        type: 'INTEGER'
      },
      {
        name: 'uid',
        type: 'INTEGER'
      }
    ]
  },
  reservedThroughput: {
    capacityUnit: {
      read: 0,
      write: 0
    }
  },
  tableOptions: {
    timeToLive: -1,//数据的过期时间，单位为秒，-1代表永不过期。如果设置过期时间为一年，即为365*24*
3600。
    maxVersions: 1//保存的最大版本数，设置为1代表每列上最多保存一个版本（保存最新的版本）。
  }
};
client.createTable(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

详细代码请参见CreateTable@GitHub。

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

- 接口

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

- 示例

  获取实例下的所有表名。

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

详细代码请参见ListTable@GitHub

更新表（UpdateTable）

更新指定表的最大版本数，预留读吞吐量或预留写吞吐量的设置。

- 接口

```
  /**
   * 更新指定表的预留读吞吐量或预留写吞吐量设置。
   */
  updateTable(params, callback)
```

- 示例

  更新表的最大版本数为5。

```
var client = require('./client');
var params = {
    tableName: 'sampleTable',
    tableOptions: {
        maxVersions: 5,
    }
};
client.updateTable(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见UpdateTable@GitHub。

查询表描述信息（DescribeTable）

查询指定表的结构信息和预留读/写吞吐量的设置信息。

- 接口

```
  /**
   * 查询指定表的结构信息和预留读/写吞吐量设置信息。
   */
  describeTable(params, callback)
```

- 示例

  获取表的描述信息。

```
var client = require('./client');
var params = {
    tableName: 'sampleTable'
};
client.describeTable(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

  详细代码请参见DescribeTable@GitHub。

删除表（DeleteTable）

删除本实例下指定的表。

- 接口

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

- 示例

  删除表。

```
var client = require('./client');
var params = {
    tableName: 'sampleTable'
};
client.deleteTable(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

  详细代码请参见DeleteTable@GitHub。

2.4.5.2. 单行数据操作

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

插入一行数据（PutRow）

插入数据到指定的行。

- 接口

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

- 示例

> 说明
>
> - RowExistenceExpectation.IGNORE表示不管此行是否存在均会插入新数据，如果之前行已存在，则写入数据时会覆盖原有数据。
> - RowExistenceExpectation.EXPECT_EXIST表示只有此行存在时才会插入新数据，写入数据时会覆盖原有数据。
> - RowExistenceExpectation.EXPECT_NOT_EXIST表示只有此行不存在时才会插入数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var currentTimeStamp = Date.now();
var params = {
  tableName: "sampleTable",
  condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE, null),
  primaryKey: [{ 'gid': Long.fromNumber(20013) }, { 'uid': Long.fromNumber(20013) }],
  attributeColumns: [
    { 'col1': '表格存储' },
    { 'col2': '2', 'timestamp': currentTimeStamp },
    { 'col3': 3.1 },
    { 'col4': -0.32 },
    { 'col5': Long.fromNumber(123456789) }
  ],
  returnContent: { returnType: TableStore.ReturnType.Primarykey }
};
client.putRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

详细代码请参见PutRow@GitHub

读取一行数据（GetRow）

根据给定的主键读取单行数据。

- 接口

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

- 示例

  读取一行数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sampleTable",
  primaryKey: [{ 'gid': Long.fromNumber(20004) }, { 'uid': Long.fromNumber(20004) }],
  maxVersions: 2 //最多可读取的版本数，设置为2即代表最多可读取2个版本。
};
var condition = new TableStore.CompositeCondition(TableStore.LogicalOperator.AND);
condition.addSubCondition(new TableStore.SingleColumnCondition('name', 'john', TableStore
.ComparatorType.EQUAL));
condition.addSubCondition(new TableStore.SingleColumnCondition('addr', 'china', TableStor
e.ComparatorType.EQUAL));
params.columnFilter = condition;
client.getRow(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  console.log('success:', data);
});
```

  详细代码请参见Get Row@Git Hub

更新一行数据（UpdateRow）

更新指定行的数据，如果该行不存在，则新增一行；如果该行存在，则根据请求的内容在这一行中新增、修改或者删除指定列的值。

- 接口

```
  /**
   * 更新指定行的数据。如果该行不存在，则新增一行；如果该行存在，则根据请求的内容在此行中新增、修改或
者删除指定列的值。
   */
  updateRow(params, callback)
```

- 示例

更新一行数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
    tableName: "sampleTable",
    condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE, null),
    primaryKey: [{ 'gid': Long.fromNumber(9) }, { 'uid': Long.fromNumber(90) }],
    updateOfAttributeColumns: [
        { 'PUT': [{ 'col4': Long.fromNumber(4) }, { 'col5': '5' }, { 'col6': Long.fromNum
ber(6) }] },
        { 'DELETE': [{ 'col1': Long.fromNumber(1496826473186) }] },
        { 'DELETE_ALL': ['col2'] }
    ]
};
client.updateRow(params,
    function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        console.log('success:', data);
    });
```

详细代码请参见UpdateRow@GitHub。

删除一行数据（DeleteRow）

删除不需要的数据。

- 接口

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

- 示例

  删除一行数据。

```
var TableStore = require('../index.js');
var Long = TableStore.Long;
var client = require('./client');
var params = {
    tableName: "sampleTable",
    condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE, null),
    primaryKey: [{ 'gid': Long.fromNumber(8) }, { 'uid': Long.fromNumber(80) }]
};
client.deleteRow(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见DeleteRow@GitHub。

2.4.5.3. 多行数据操作

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

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

BatchGetRow操作可视为多个GetRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

与执行大量的GetRow操作相比，使用BatchGetRow操作可以有效减少请求的响应时间，提高数据的读取速率。

- 接口

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

- 示例

  批量一次读多个表、多行，单行出错时进行重试。

> **说明** 批量读也支持通过条件语句过滤。

```
var client = require('./client');
var TableStore = require('../index.js');
var Long = TableStore.Long;
var params = {
    tables: [{
        tableName: 'sampleTable',
        primaryKey: [
            [{ 'gid': Long.fromNumber(20013) }, { 'uid': Long.fromNumber(20013) }],
```

```
            [{ 'gid': Long.fromNumber(20015) }, { 'uid': Long.fromNumber(20015) }]
        ],
        startColumn: "col2",
        endColumn: "col4"
    },
    {
        tableName: 'notExistTable',
        primaryKey: [
            [{ 'gid': Long.fromNumber(10001) }, { 'uid': Long.fromNumber(10001) }]
        ]
    }
    ],
};
var maxRetryTimes = 3;
var retryCount = 0;
function batchGetRow(params) {
    client.batchGetRow(params, function (err, data) {
        if (err) {
            console.log('error:', err);
            return;
        }
        var isAllSuccess = true;
        var retryRequest = { tables: [] };
        for (var i = 0; i < data.tables.length; i++) {
            var faildRequest = { tableName: data.tables[i][0].tableName, primaryKey: [] }
;
            for (var j = 0; j < data.tables[i].length; j++) {
                if (!data.tables[i][j].isOk && null != data.tables[i][j].primaryKey) {
                    isAllSuccess = false;
                    var pks = [];
                    for (var k in data.tables[i][j].primaryKey) {
                        var name = data.tables[i][j].primaryKey[k].name;
                        var value = data.tables[i][j].primaryKey[k].value;
                        var kp = {};
                        kp[name] = value;
                        pks.push(kp);
                    }
                    faildRequest.primaryKey.push(pks);
                } else {
                    // get success data
                }
            }
            if (faildRequest.primaryKey.length > 0) {
                retryRequest.tables.push(faildRequest);
            }
        }
        if (!isAllSuccess && retryCount++ < maxRetryTimes) {
            batchGetRow(retryRequest);
        }
        console.log('success:', data);
    });
}
batchGetRow(params, maxRetryTimes);
```

详细代码请参见BatchGetRow@GitHub。

批量写（BatchWriteRow）

批量插入、修改或删除一个或多个表中的若干行数据。

BatchWriteRow操作可视为多个PutRow、UpdateRow、DeleteRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

- 接口

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

- 示例

  批量写入数据。

  > **说明** 批量写也支持通过条件语句过滤。

```
var client = require('./client');
var TableStore = require('../index.js');
var Long = TableStore.Long;
var params = {
    tables: [{
        tableName: 'sampleTable',
        rows: [{
            type: 'PUT',
            condition: new TableStore.Condition(TableStore.RowExistenceExpectation.IGNORE
, null),
            primaryKey: [{ 'gid': Long.fromNumber(8) }, { 'uid': Long.fromNumber(80) }],
            attributeColumns: [{ 'attrCol1': 'test1' }, { 'attrCol2': 'test2' }],
            returnContent: { returnType: TableStore.ReturnType.Primarykey }
        }],
    }],
};
client.batchWriteRow(params, function (err, data) {
    if (err) {
        console.log('error:', err);
        return;
    }
    console.log('success:', data);
});
```

详细代码请参见BatchWriteRow@GitHub。

范围读（GetRange）

读取指定主键范围内的数据。

- 接口

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

- 示例

  按照范围读取数据。

  > **说明**
  > - 按范围读取也支持通过条件语句过滤。
  > - 按范围读取需要注意数据可能会分页。

```
var Long = TableStore.Long;
var client = require('./client');
var params = {
  tableName: "sampleTable",
  direction: TableStore.Direction.FORWARD,
  inclusiveStartPrimaryKey: [{ "gid": TableStore.INF_MIN }, { "uid": TableStore.INF_MIN }
],
  exclusiveEndPrimaryKey: [{ "gid": TableStore.INF_MAX }, { "uid": TableStore.INF_MAX }],
  limit: 50
};
client.getRange(params, function (err, data) {
  if (err) {
    console.log('error:', err);
    return;
  }
  //如果data.next_start_primary_key不为空，说明需要继续读取
  if (data.next_start_primary_key) {
  }
  console.log('success:', data);
});
```

详细代码请参见GetRange@GitHub。

2.4.6. 错误处理

介绍表格存储Node.js SDK的错误处理方式和重试策略。

方式

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

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

异常

表格存储Node.js SDK中所有的错误均经过了统一的处理，最终会返回到callback方法的err参数中，所以在获取返回数据前，需要检查err参数是否有值。如果是表格存储服务端报错，会返回requestId。requestId用于唯一标识该次请求的UUID。当您无法解决问题时，记录此requestId联系表格存储的开发工程师获取帮助。

重试

SDK中出现错误时会自动重试。默认策略是最大重试次数为20，最大重试间隔为3000毫秒。对流控类错误以及读操作相关的服务端内部错误进行的重试，请参见tablestore/lib/retry.js。

2.5. .NET-SDK

2.5.1. 前言

本文介绍表格存储.NET SDK的使用。

前提条件

已获取一个授权账号以及一对AccessKey ID和AccessKey Secret。请参见获取Accesskey在Apsara Uni-manager运营控制台上获取和查看AccessKey。

SDK下载

- 从NuGet下载SDK安装包，具体下载路径请参见https://www.nuget.org/packages/Aliyun.TableStore.SDK/4.1.4。
- 从Github下载源码，具体下载路径请参见GitHub。

兼容性

对于3.x.x系列的SDK兼容。

对于2.x.x系列的SDK不兼容处如下。

- 接口部分不兼容：删除Condition.IGNORE、Condition.EXPECT_EXIST和Condition.EXPECT_NOT_EXIST。
- DLL文件名称由Aliyun.dll变更为Aliyun.TableStore.dll。

版本

当前最新版本为4.1.4。

2.5.2. 安装

介绍如何安装表格存储.NET SDK。

版本依赖

Windows

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

Windows环境安装

- NuGet安装
  i. 在Visual Studio中新建或者打开已有的项目后，选择**工具 > NuGet 程序包管理器 > 管理解决方案的 NuGet 程序包**。

> **说明** 如果Visual Studio未安装NuGet，请下载并安装NuGet，具体下载路径请参见NuGet。

ii. 搜索aliyun.tablestore，在结果中找到Aliyun.TableStore.SDK。

iii. 选择最新版本，单击**安装**。

安装成功后，表格存储.NET SDK会添加到项目应用中。

- 项目引入方式安装

i. 使用git从GitHub下载源码，具体下载路径请参见GitHub。

> **说明** 如果未安装git，请下载并安装git，具体下载路径请参见git。

ii. 在Visual Studio中右键选择**解决方案**，在弹出的菜单中选择**添加 > 现有项目** 。

iii. 在弹出的对话框中选择*aliyun-tablestore-sdk.csproj*文件，单击**打开**。

iv. 右键选择**您的项目**，选择**引用 > 添加引用**，在弹出的对话框选择**项目**选项卡，并选中aliyun-tablestore-sdk项目。

v. 单击**确定**。

2.5.3. 初始化

TableStoreClient是表格存储服务的客户端，它为调用者提供了一系列的方法，可以用来操作表、单行数据、多行数据等。

确定Endpoint

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

1. 登录表格存储控制台。
2. 单击实例名称进入**实例详情**页。

   实例访问地址即是该实例的Endpoint。

配置密钥

要接入阿里云表格存储服务，需要拥有一个有效的AccessKey（包括AccessKey ID和AccessKey Secret）用来进行签名认证。

获取到AccessKey ID和AccessKey Secret后，使用表格存储的Endpoint进行初始化对接，示例如下。

- 接口

```
    /// <summary>
    /// OTSClient的构造函数。
    /// </summary>
    /// <param name="endPoint">表格存储实例的服务地址（例如'https://instance.cn-hangzhou.ots.aliyun.com:80'），必须以'https://'开头。</param>
    /// <param name="accessKeyID">表格存储的AccessKey ID，通过官方网站申请。</param>
    /// <param name="accessKeySecret">表格存储的AccessKey Secret，通过官方网站申请。</param>
    /// <param name="instanceName">表格存储的实例名，通过官方网站控制台创建。</param>
public OTSClient(string endPoint, string accessKeyID, string accessKeySecret, string instanceName);
    /// <summary>
    /// 通过客户端配置OTSClientConfig的实例来创建OTSClient实例。
    /// </summary>
    /// <param name="config">客户端配置实例</param>
    public OTSClient(OTSClientConfig config);
```

- 示例

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

```
    // 构造一个OTSClientConfig对象。
    var config = new OTSClientConfig(Endpoint, AccessKeyId, AccessKeySecret, InstanceName);
    // 禁止输出日志，默认是打开的。
    config.OTSDebugLogHandler = null;
    config.OTSErrorLogHandler = null;
    // 使用OTSClientConfig创建一个OtsClient对象。
    var otsClient = new OTSClient(config);
    // 使用otsClient插入或者查询数据。
```

多线程

- 支持多线程。
- 使用多线程时，建议共用一个OTSClient对象。

2.5.4. 使用手册

2.5.4.1. 表操作

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

创建表（CreateTable）

根据给定的表的结构信息创建相应的表。

> **说明**
>
> - 创建表后需要几秒钟进行加载，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。
> - 创建表时必须指定表的主键。主键包含1~4个主键列，每一个主键列都有名称和类型。

- 接口

```
/// <summary>
 /// 根据表信息（包含表名、主键的设计和预留读写吞吐量）创建表。
 /// </summary>
 /// <param name="request">请求参数</param>
 /// <returns>CreateTable的返回，此返回实例是空的，不包含具体信息。
 /// </returns>
        public CreateTableResponse CreateTable(CreateTableRequest request);
        /// <summary>
        /// CreateTable的异步形式。
        /// </summary>
        public Task<CreateTableResponse> CreateTableAsync(CreateTableRequest request);
```

- 示例

  创建一个有2个主键列，预留读/写吞吐量(0, 0)的表。

```
//创建主键列的schema，包括PK的个数、名称和类型。
//第一个PK列为整数，名称是pk0，这个同时也是分区键。
//第二个PK列为字符串，名称是pk1。
var primaryKeySchema = new PrimaryKeySchema();
primaryKeySchema.Add("pk0", ColumnValueType.Integer);
primaryKeySchema.Add("pk1", ColumnValueType.String);
//通过表名和主键列的schema创建一个tableMeta。
var tableMeta = new TableMeta("SampleTable", primaryKeySchema);
//设置预留读吞吐量为0，预留写吞吐量为0。
var reservedThroughput = new CapacityUnit(0, 0);
try
{
    //构造CreateTableRequest对象。
    var request = new CreateTableRequest(tableMeta, reservedThroughput);
    //调用client的CreateTable接口，如果没有抛出异常，则说明执行成功。
    otsClient.CreateTable(request);
    Console.WriteLine("Create table succeeded.");
}
//如果抛出异常，则说明失败，处理异常。
catch (Exception ex)
{
    Console.WriteLine("Create table failed, exception:{0}", ex.Message);
}
```

  详细代码请参见CreateTable@GitHub。

列出表名称（ListTable）

获取当前实例下已创建的所有表的表名。

- 接口

```
/// <summary>
/// 获取当前实例下已创建的所有表的表名。
/// </summary>
/// <param name="request">请求参数</param>
/// <returns>ListTable的返回，用来获取表名列表。</returns>
public ListTableResponse ListTable(ListTableRequest request);
/// <summary>
/// ListTable的异步形式。
/// </summary>
public Task<ListTableResponse> ListTableAsync(ListTableRequest request);
```

- 示例

  获取实例下的所有表名。

```
var request = new ListTableRequest();
try
{
    var response = otsClient.ListTable(request);
    foreach (var tableName in response.TableNames)
    {
        Console.WriteLine("Table name:{0}", tableName);
    }
    Console.Writeline("List table succeeded.");
}
catch (Exception ex)
{
    Console.WriteLine("List table failed, exception:{0}", ex.Message);
}
```

更新表（UpdateTable）

更新指定表的预留读吞吐量或预留写吞吐量设置。

- 接口

```
/// <summary>
/// 更新指定表的预留读吞吐量或预留写吞吐量，新设置将于更新成功一分钟内生效。
/// </summary>
/// <param name="request">请求参数，包含表名以及预留读写吞吐量</param>
/// <returns>包含更新后的预留读写吞吐量等信息</returns>
public UpdateTableResponse UpdateTable(UpdateTableRequest request);
/// <summary>
/// UpdateTable的异步形式。
/// </summary>
public Task<UpdateTableResponse> UpdateTableAsync(UpdateTableRequest request);
```

- 示例

  更新表的CU值为读1，写2。

```
//设置新的预留读吞吐量为1，预留写吞吐量为2。
var reservedThroughput = new CapacityUnit(1, 2);
//构造UpdateTableRequest对象。
var request = new UpdateTableRequest("SampleTable", reservedThroughput);
try
{
    //调用接口更新表的预留读写吞吐量。
    otsClient.UpdateTable(request);
    //如果没有抛出异常，则说明执行成功。
    Console.Writeline("Update table succeeded.");
}
catch (Exception ex)
{
    //如果抛出异常，则说明执行失败，处理异常。
    Console.WriteLine("Update table failed, exception:{0}", ex.Message);
}
```

详细代码请参见UpdateTable@GitHub。

查询表描述信息（DescribeTable）

查询指定表的结构信息和预留读/写吞吐量设置信息。

- 接口

```
/// <summary>
/// 查询指定表的结构信息和预留读写吞吐量设置信息。
/// </summary>
/// <param name="request">请求参数，包含表名</param>
/// <returns>包含表的结构信息和预留读写吞吐量等信息。</returns>
public DescribeTableResponse DescribeTable(DescribeTableRequest request);
/// <summary>
/// DescribeTable的异步形式。
/// </summary>
public Task<DescribeTableResponse> DescribeTableAsync(DescribeTableRequest request);
```

- 示例

  获取表的描述信息。

```
try
{
    var request = new DescribeTableRequest("SampleTable");
    var response = otsClient.DescribeTable(request);
    //打印表的描述信息。
    Console.Writeline("Describe table succeeded.");
    Console.WriteLine("LastIncreaseTime: {0}", response.ReservedThroughputDetails.LastIncreaseTime);
    Console.WriteLine("LastDecreaseTime: {0}", response.ReservedThroughputDetails.LastDecreaseTime);
    Console.WriteLine("NumberOfDecreaseToday: {0}", response.ReservedThroughputDetails.LastIncreaseTime);
    Console.WriteLine("ReadCapacity: {0}", response.ReservedThroughputDetails.CapacityUnit.Read);
    Console.WriteLine("WriteCapacity: {0}", response.ReservedThroughputDetails.CapacityUnit.Write);
}
catch (Exception ex)
{
    //如果抛出异常，则说明执行失败，处理异常。
    Console.WriteLine("Describe table failed, exception:{0}", ex.Message);
       }
```

详细代码请参见DescribeTable@GitHub。

删除表（DeleteTable）

删除本实例下指定的表。

- 接口

```
/// <summary>
/// 根据表名删除表。
/// </summary>
/// <param name="request">请求参数，包含表名</param>
/// <returns>DeleteTable的返回，这个返回实例是空的，不包含具体信息。
/// </returns>
public DeleteTableResponse DeleteTable(DeleteTableRequest request);
/// <summary>
/// DeleteTable的异步形式。
/// </summary>
public Task<DeleteTableResponse> DeleteTableAsync(DeleteTableRequest request);
```

- 示例

  删除表。

```
var request = new DeleteTableRequest("SampleTable");
try
{
    otsClient.DeleteTable(request);
    Console.Writeline("Delete table succeeded.");
}
catch (Exception ex)
{
    Console.WriteLine("Delete table failed, exception:{0}", ex.Message);
}
```

详细代码请参见DeleteTable@GitHub。

2.5.4.2. 单行数据操作

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

插入一行数据（PutRow）

插入数据到指定的行。

接口

```
        /// <summary>
        /// 指定表名、主键和属性，写入一行数据。返回本次操作消耗的CapacityUnit。
        /// </summary>
        /// <param name="request">插入数据的请求</param>
        /// <returns>本次操作消耗的CapacityUnit</returns>
        public PutRowResponse PutRow(PutRowRequest request);
        /// <summary>
        /// PutRow的异步形式。
        /// </summary>
        public Task<PutRowResponse> PutRowAsync(PutRowRequest request);
```

示例 1

插入一行数据。

```
        // 定义行的主键，必须与创建表时的TableMeta中定义的一致
        var primaryKey = new PrimaryKey();
        primaryKey.Add("pk0", new ColumnValue(0));
        primaryKey.Add("pk1", new ColumnValue("abc"));
        // 定义要写入改行的属性列
        var attribute = new AttributeColumns();
        attribute.Add("col0", new ColumnValue(0));
        attribute.Add("col1", new ColumnValue("a"));
        attribute.Add("col2", new ColumnValue(true));
        try
        {
            // 构造插入数据的请求对象，RowExistenceExpectation.IGNORE表示不管此行是否存在都执行
            var request = new PutRowRequest("SampleTable", new Condition(RowExistenceExpect
ation.IGNORE),
                                    primaryKey, attribute);
            // 调用PutRow接口插入数据
            otsClient.PutRow(request);
            // 如果没有抛出异常，则说明执行成功
            Console.WriteLine("Put row succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则说明执行失败，打印出错误信息
            Console.WriteLine("Put row failed, exception:{0}", ex.Message);
        }
```

说明

- Condition.IGNORE、Condition.EXPECT_EXIST 和 Condition.EXPECT_NOT_EXIST 从 3.0.0 版本开始被废弃，请替换为 new Condition (RowExistenceExpectation.IGNORE)、new Condition (RowExistenceExpectation.EXPECT_EXIST) 和 new Condition (RowExistenceExpectation.EXPECT_NOT_EXIST)。
- RowExistenceExpectation.IGNORE 表示不管此行是否已经存在，都会插入新数据，如果之前有会被覆盖。
- RowExistenceExpectation.EXPECT_EXIST 表示只有此行存在时，才会插入新数据，此时，原有数据也会被覆盖。
- RowExistenceExpectation.EXPECT_NOT_EXIST 表示只有此行不存在时，才会插入数据，否则不执行。
- 从 2.2.0 版本开始，Condition 不仅支持行条件，也支持列条件。
- 详细代码：PutRow@GitHub。

示例 2

设置条件插入一行数据。

下列示例演示：当行存在，且 col1 大于 24 的时候才执行插入操作。

```
        // 定义行的主键，必须与创建表时的TableMeta中定义的一致
        var primaryKey = new PrimaryKey();
        primaryKey.Add("pk0", new ColumnValue(0));
        primaryKey.Add("pk1", new ColumnValue("abc"));
        // 定义要写入改行的属性列
        AttributeColumns attribute = new AttributeColumns();
        attribute.Add("col0", new ColumnValue(0));
        attribute.Add("col1", new ColumnValue("a"));
        attribute.Add("col2", new ColumnValue(true));
        var request = new PutRowRequest(tableName, new Condition(RowExistenceExpectation.EX
PECT_EXIST),
                                        primaryKey, attribute);
        // 当col0列的值大于24的时候，允许再次put row，覆盖掉原值
        try
        {
            request.Condition.ColumnCondition = new RelationalCondition("col0",
                                                RelationalCondition.CompareOperator.GREATER
_THAN,
                                                new ColumnValue(24));
            otsClient.PutRow(request);
            Console.WriteLine("Put row succeeded.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Put row failed. error:{0}", ex.Message);
        }
```

说明

- 条件不仅支持单个条件，也支持多个条件组合。例如，col1 大于 5 且 pk2 小于'xyz'时插入数据。
- 属性列和主键列都支持条件。
- 当条件中的列在某行不存在时，可以通过 RelationCondition 中的 PassIfMissing 控制，默认是 true。
- 详细代码：ConditionPutRow@GitHub。

示例 3

异步插入一行数据。

```
        try
        {
            var putRowTaskList = new List<Task<PutRowResponse>>();
            for (int i = 0; i < 100; i++)
            {
                // 定义行的主键，必须与创建表时的TableMeta中定义的一致
                var primaryKey = new PrimaryKey();
                primaryKey.Add("pk0", new ColumnValue(i));
                primaryKey.Add("pk1", new ColumnValue("abc"));
                // 定义要写入改行的属性列
                var attribute = new AttributeColumns();
                attribute.Add("col0", new ColumnValue(i));
                attribute.Add("col1", new ColumnValue("a"));
                attribute.Add("col2", new ColumnValue(true));
                var request = new PutRowRequest(TableName, new Condition(RowExistenceExpect
ation.IGNORE),
                                                primaryKey, attribute);
                putRowTaskList.Add(TabeStoreClient.PutRowAsync(request));
            }
            // 等待每个异步调用返回，并打印出消耗的CU值
            foreach (var task in putRowTaskList)
            {
                task.Wait();
                Console.WriteLine("consumed read:{0}, write:{1}", task.Result.ConsumedCapac
ityUnit.Read,
                                        task.Result.ConsumedCapacityUnit.Write);
            }
            // 如果没有抛出异常，则说明插入数据成功
            Console.WriteLine("Put row async succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则打印出出错信息
            Console.WriteLine("Put row async failed. exception:{0}", ex.Message);
        }
```

> **说明**
>
> - 每一个异步调用都会启动一个线程，如果连续启动了很多异步调用，且每个都耗时比较大的时候，可能会出现超时。
> - 详细代码：PutRowAsync@GitHub。

读取一行数据（GetRow）

根据给定的主键读取单行数据。

接口

```
/// <summary>
/// 根据给定的主键读取单行数据。
/// </summary>
/// <param name="request">查询数据的请求</param>
/// <returns>GetRow的响应</returns>
public GetRowResponse GetRow(GetRowRequest request);
/// <summary>
/// GetRow的异步形式。
/// </summary>
public Task<GetRowResponse> GetRowAsync(GetRowRequest request);
```

示例 1

读取一行数据。

```
// 定义行的主键，必须与创建表时的TableMeta中定义的一致
PrimaryKey primaryKey = new PrimaryKey();
primaryKey.Add("pk0", new ColumnValue(0));
primaryKey.Add("pk1", new ColumnValue("abc"));
try
{
    // 构造查询请求对象，这里未指定读哪列，默认读整行
    var request = new GetRowRequest(TableName, primaryKey);
    // 调用GetRow接口查询数据
    var response = otsClient.GetRow(request);
    // 输出此行的数据，这里省略，详见下面GitHub的链接
    // 如果没有抛出异常，则说明成功
    Console.WriteLine("Get row succeeded.");
}
catch (Exception ex)
{
    // 如果抛出异常，说明执行失败，打印出错误信息
    Console.WriteLine("Update table failed, exception:{0}", ex.Message);
}
```

说明

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

示例 2

使用过滤读取一行数据。

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

```
        // 定义行的主键，必须与创建表时的TableMeta中定义的一致
        PrimaryKey primaryKey = new PrimaryKey();
        primaryKey.Add("pk0", new ColumnValue(0));
        primaryKey.Add("pk1", new ColumnValue("abc"));
        var rowQueryCriteria = new SingleRowQueryCriteria("SampleTable");
        rowQueryCriteria.RowPrimaryKey = primaryKey;
        // 条件1：col0的值等于5
        var filter1 = new RelationalCondition("col0",
                    RelationalCondition.CompareOperator.EQUAL,
                    new ColumnValue(5));
        // 条件2：col1不等于ff的行
        var filter2 = new RelationalCondition("col1", RelationalCondition.CompareOperator.N
OT_EQUAL, new ColumnValue("ff"));
        // 构造组合条件，包括条件1和条件2，关系是OR
        var filter = new CompositeCondition(CompositeCondition.LogicOperator.OR);
        filter.AddCondition(filter1);
        filter.AddCondition(filter2);
        rowQueryCriteria.Filter = filter;
        // 设置要查询和返回的行，查询和过滤的顺序是：先在行[col0,col1]上查询，然后再按条件过滤
        rowQueryCriteria.AddColumnsToGet("col0");
        rowQueryCriteria.AddColumnsToGet("col1");
        // 构造GetRowRequest
        var request = new GetRowRequest(rowQueryCriteria);
        try
        {
            // 查询
            var response = otsClient.GetRow(request);
            // 输出数据或者相关逻辑操作，这里省略
            // 如果没有抛出异常，则说明执行成功
            Console.WriteLine("Get row with filter succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则说明执行失败，打印出错误信息
            Console.WriteLine("Get row with filter failed, exception:{0}", ex.Message);
        }
```

> **说明** 详细代码：GetRowWithFilter@GitHub。

更新一行数据（UpdateRow）

更新指定行的数据。如果该行不存在，则新增一行；若该行存在，则根据请求的内容在这一行中新增、修改或者删除指定列的值。

接口

```
        /// <summary>
        /// 更新指定行的数据，如果该行不存在，则新增一行；若该行存在，则根据请求的内容在这一行中新增、修改或者删除指定列的值。
        /// </summary>
        /// <param name="request">请求实例</param>
        public UpdateRowResponse UpdateRow(UpdateRowRequest request);
        /// <summary>
        /// UpdateRow的异步形式。
        /// </summary>
        /// <param name="request"></param>
        /// <returns></returns>
        public Task<UpdateRowResponse> UpdateRowAsync(UpdateRowRequest request);
```

示例

更新一行数据。

```
        // 定义行的主键，必须与创建表时的TableMeta中定义的一致
        PrimaryKey primaryKey = new PrimaryKey();
        primaryKey.Add("pk0", new ColumnValue(0));
        primaryKey.Add("pk1", new ColumnValue("abc"));
        // 定义要写入改行的属性列
        UpdateOfAttribute attribute = new UpdateOfAttribute();
        attribute.AddAttributeColumnToPut("col0", new ColumnValue(0));
        attribute.AddAttributeColumnToPut("col1", new ColumnValue("b")); // 将原先的值'a'改为'b'
        attribute.AddAttributeColumnToPut("col2", new ColumnValue(true));
        try
        {
            // 构造更新行的请求对象，RowExistenceExpectation.IGNORE表示不管此行是否存在都执行
            var request = new UpdateRowRequest(TableName, new Condition(RowExistenceExpectation.IGNORE),
                                        primaryKey, attribute);
            // 调用UpdateRow接口执行
            otsClient.UpdateRow(request);
            // 如果没有抛出异常，则说明执行成功
            Console.Writeline("Update row succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，说明执行失败，打印异常信息
            Console.WriteLine("Update row failed, exception:{0}", ex.Message);
        }
```

> 说明
>
> - 更新一行数据也支持条件语句。
> - 详细代码：UpdateRow@GitHub。

删除一行数据（DeleteRow）

接口

```
        /// <summary>
        /// 指定表名和主键，删除一行数据。
        /// </summary>
        /// <param name="request">请求实例</param>
        /// <returns>响应实例</returns>
        public DeleteRowResponse DeleteRow(DeleteRowRequest request);
        /// <summary>
        /// DeleteRow的异步形式。
        /// </summary>
        public Task<DeleteRowResponse> DeleteRowAsync(DeleteRowRequest request);
```

示例

删除一行数据。

```
        // 要删除的行的PK列分别为0和"abc"
        var primaryKey = new PrimaryKey();
        primaryKey.Add("pk0", new ColumnValue(0));
        primaryKey.Add("pk1", new ColumnValue("abc"));
        try
        {
            // 构造请求，Condition.EXPECT_EXIST表示只有此行存在时才执行
            var deleteRowRequest = new DeleteRowRequest("SampleTable", Condition.EXPECT_EXI
ST, primaryKey);
            // 调用DeleteRow接口执行删除
            otsClient.DeleteRow(deleteRowRequest);
            // 如果没有抛出异常，则表示成功
            Console.Writeline("Delete table succeeded.");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，说明删除失败，打印粗错误信息
            Console.WriteLine("Delete table failed, exception:{0}", ex.Message);
        }
```

> 说明
>
> - 删除一行数据也支持条件语句。
> - 详细代码：DeleteRow@GitHub。

2.5.4.3. 多行数据操作

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

批量读（BatchGetRow）

批量读取一个或多个表中的若干行数据。

BatchGetRow 操作可视为多个 GetRow 操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

与执行大量的 GetRow 操作相比，使用 BatchGetRow 操作可以有效减少请求的响应时间，提高数据的读取速率。

接口

```
        /// <summary>
        /// <para>批量读取一个或多个表中的若干行数据。</para>
        /// <para>BatchGetRow操作可视为多个GetRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。</para>
        /// 与执行大量的GetRow操作相比，使用BatchGetRow操作可以有效减少请求的响应时间，提高数据的读取速率。
        /// </summary>
        /// <param name="request">请求实例</param>
        /// <returns>响应实例</returns>
        public BatchGetRowResponse BatchGetRow(BatchGetRowRequest request);
        /// <summary>
        /// BatchGetRow的异步形式。
        /// </summary>
        public Task<BatchGetRowResponse> BatchGetRowAsync(BatchGetRowRequest request);
```

示例

批量一次读 10 行。

```
        // 构造批量读取请求的对象，设置10行的pk值
        List<PrimaryKey> primaryKeys = new List<PrimaryKey>();
        for (int i = 0; i < 10; i++)
        {
            PrimaryKey primaryKey = new PrimaryKey();
            primaryKey.Add("pk0", new ColumnValue(i));
            primaryKey.Add("pk1", new ColumnValue("abc"));
            primaryKeys.Add(primaryKey);
        }
        try
        {
            BatchGetRowRequest request = new BatchGetRowRequest();
            request.Add(TableName, primaryKeys);
            // 调用BatchGetRow，查询十行数据
            var response = otsClient.BatchGetRow(request);
            var tableRows = response.RowDataGroupByTable;
            var rows = tableRows[TableName];
            // 输入rows里的数据，这里省略，详见下面GitHub链接
            // 批量操作可能部分成功部分失败，需要为每行检查状态，详见下面GitHub链接
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则说明执行失败，打印出错误信息
            Console.WriteLine("Batch get row failed, exception:{0}", ex.Message);
        }
```

> 说明
>
> - 批量读也支持通过条件语句过滤。
> - 详细代码：BatchGetRow@GitHub。

批量写（BatchWriteRow）

批量插入、修改或删除一个或多个表中的若干行数据。

BatchWriteRow 操作可视为多个 PutRow、UpdateRow 和 DeleteRow 操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。

接口

```
        /// <summary>
        /// <para>批量插入，修改或删除一个或多个表中的若干行数据。</para>
        /// <para>BatchWriteRow操作可视为多个PutRow、UpdateRow、DeleteRow操作的集合，各个操作独立执行，独立返回结果，独立计算服务能力单元。</para>
        /// <para>与执行大量的单行写操作相比，使用BatchWriteRow操作可以有效减少请求的响应时间，提高数据的写入速率。</para>
        /// </summary>
        /// <param name="request">请求实例</param>
        /// <returns>响应实例</returns>
        public BatchWriteRowResponse BatchWriteRow(BatchWriteRowRequest request);
        /// <summary>
        /// BatchWriteRow的异步形式。
        /// </summary>
        /// <param name="request"></param>
        /// <returns></returns>
        public Task<BatchWriteRowResponse> BatchWriteRowAsync(BatchWriteRowRequest request)
;
```

示例

批量导入 100 行数据。

```
        // 构造批量插入的请求对象，包括了这100行数据的pk
        var request = new BatchWriteRowRequest();
        var rowChanges = new RowChanges();
        for (int i = 0; i < 100; i++)
        {
            PrimaryKey primaryKey = new PrimaryKey();
            primaryKey.Add("pk0", new ColumnValue(i));
            primaryKey.Add("pk1", new ColumnValue("abc"));
            // 定义要写入改行的属性列
            UpdateOfAttribute attribute = new UpdateOfAttribute();
            attribute.AddAttributeColumnToPut("col0", new ColumnValue(0));
            attribute.AddAttributeColumnToPut("col1", new ColumnValue("a"));
            attribute.AddAttributeColumnToPut("col2", new ColumnValue(true));
            rowChanges.AddUpdate(new Condition(RowExistenceExpectation.IGNORE), primaryKey,
attribute);
        }
        request.Add(TableName, rowChanges);
        try
        {
            // 调用BatchWriteRow接口
            var response = otsClient.BatchWriteRow(request);
            var tableRows = response.TableRespones;
            var rows = tableRows[TableName];
            // 批量操作可能部分成功部分失败，需要为每行检查状态，详见下面GitHub链接
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则说明执行失败，打印出错误信息
            Console.WriteLine("Batch put row failed, exception:{0}", ex.Message);
        }
```

> **说明**
>
> - 批量写也支持条件语句。
> - 详细代码：BatchWriteRow@GitHub。

范围读（GetRange）

读取指定主键范围内的数据。

接口

```
        /// <summary>
        /// 根据范围条件获取多行数据。
        /// </summary>
        /// <param name="request">请求实例</param>
        /// <returns>响应实例</returns>
        public GetRangeResponse GetRange(GetRangeRequest request);
        /// <summary>
        /// GetRange的异步版本。
        /// </summary>
        /// <param name="request"></param>
        /// <returns></returns>
        public Task<GetRangeResponse> GetRangeAsync(GetRangeRequest request);
```

示例

范围读取。

```
        // 读取 (0, INF_MIN)到(100, INF_MAX)这个范围内的所有行
        var inclusiveStartPrimaryKey = new PrimaryKey();
        inclusiveStartPrimaryKey.Add("pk0", new ColumnValue(0));
        inclusiveStartPrimaryKey.Add("pk1", ColumnValue.INF_MIN);
        var exclusiveEndPrimaryKey = new PrimaryKey();
        exclusiveEndPrimaryKey.Add("pk0", new ColumnValue(100));
        exclusiveEndPrimaryKey.Add("pk1", ColumnValue.INF_MAX);
        try
        {
            // 构造范围查询请求对象
            var request = new GetRangeRequest(TableName, GetRangeDirection.Forward,
                            inclusiveStartPrimaryKey, exclusiveEndPrimaryKey);
            var response = otsClient.GetRange(request);
            // 如果一次没有返回所有数据，则需要继续查询
            var rows = response.RowDataList;
            var nextStartPrimaryKey = response.NextPrimaryKey;
            while (nextStartPrimaryKey != null)
            {
                request = new GetRangeRequest(TableName, GetRangeDirection.Forward,
                                nextStartPrimaryKey, exclusiveEndPrimaryKey);
                response = otsClient.GetRange(request);
                nextStartPrimaryKey = response.NextPrimaryKey;
                foreach (RowDataFromGetRange row in response.RowDataList)
                {
                    rows.Add(row);
                }
            }
            // 输出Rows的数据，这里省略，详见下面GitHub链接
            // 如果没有抛出异常，则说明执行成功
            Console.WriteLine("Get range succeeded");
        }
        catch (Exception ex)
        {
            // 如果抛出异常，则说明执行失败，打印出错误信息
            Console.WriteLine("Get range failed, exception:{0}", ex.Message);
        }
```

> 说明
>
> - 按范围读也支持通过条件语句过滤。
> - 详细代码：GetRange@GitHub。

迭代读（GetRangeIterator）

获取一个范围查询的迭代器。

接口

```
        /// <summary>
        /// 根据范围条件获取多行数据，返回用来迭代每一行数据的迭代器。
        /// </summary>
        /// <param name="request"><see cref="GetIteratorRequest"/></param>
        /// <returns>返回<see cref="RowDataFromGetRange"/>的迭代器。</returns>
        public IEnumerable<RowDataFromGetRange> GetRangeIterator(GetIteratorRequest request
);
```

示例

迭代读取。

```
        // 读取 (0, "a")到(1000, "xyz")这个范围内的所有行
        PrimaryKey inclusiveStartPrimaryKey = new PrimaryKey();
        inclusiveStartPrimaryKey.Add("pk0", new ColumnValue(0));
        inclusiveStartPrimaryKey.Add("pk1", new ColumnValue("a"));
        PrimaryKey exclusiveEndPrimaryKey = new PrimaryKey();
        exclusiveEndPrimaryKey.Add("pk0", new ColumnValue(1000));
        exclusiveEndPrimaryKey.Add("pk1", new ColumnValue("xyz"));
        // 构造一个CapacityUnit，用于记录迭代过程中消耗的CU值
        var cu = new CapacityUnit(0, 0);
        try
        {
            // 构造一个GetIteratorRequest，这里也支持过滤条件
            var request = new GetIteratorRequest(TableName, GetRangeDirection.Forward, incl
usiveStartPrimaryKey,
                                                 exclusiveEndPrimaryKey, cu);
            var iterator = otsClient.GetRangeIterator(request);
            // 遍历迭代器，读取数据
            foreach (var row in iterator)
            {
                // 处理逻辑
            }
            Console.WriteLine("Iterate row succeeded");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Iterate row failed, exception:{0}", ex.Message);
        }
```

> **说明**
> - 读数据迭代器也支持通过条件语句过滤
> - 详细代码：GetRangeIterator@GitHub。

2.5.5. 错误处理

方式

TableStore C# SDK 目前采用异常的方式处理错误，如果调用接口没有抛出异常，则说明操作成功，否则失败。

> **说明** 批量相关接口，比如 BatchGetRow 和 BatchWriteRow 需要检查每个 row 的状态都是成功后才能保证整个接口调用是成功的。

异常

TableStore C# SDK 中有 OTSClientException 和 OTSServerException 两种异常，他们都最终继承自 Exception。

- OTSClientException：指 SDK 内部出现的异常，比如参数设置不对，返回结果解析失败等。
- OTSServerException：指服务器端的错误，它来自于对服务器错误信息的解析。OTSServerException 一般有以下几个成员：
  - HttpStatusCode：HTTP 返回码，比如 200、404等。
  - ErrorCode：表格存储返回的错误类型字符串。
  - ErrorMessage：表格存储返回的错误消息字符串。
  - RequestId：用于唯一标识该次请求的 UUID。当您无法解决问题时，可以凭这个 RequestId 来请求表格存储开发工程师的帮助。

重试

- SDK 中出现错误时会自动重试。默认策略是最多重试3次，重试间隔最大2秒，详情请参见 Aliyun.OTS.Retry.DefaultRetryPolicy 类。
- 用户也可以通过修改 OTSClientConfig 中的 RetryPolicy 自定义重试策略。

2.6. 获取Accesskey

AccessKey支持RAM和STS两种授权模式，在发起调用时选择使用其中一种AccessKey即可。本文将为您介绍如何获取通过RAM授权的AccessKey。

获取个人账号AccessKey

获取个人账号AccessKey的方法如下：

1. 登录Apsara Uni-manager运营控制台。
2. 在系统界面右上角，单击当前登录用户头像，单击**个人信息**。
3. 在**阿里云AccessKey**区域，您可以查看个人账户的AccessKey信息。

> **说明** Accesskey ID和AccessKey Secret是您访问云资源时的密钥，具有该账号完整的权限，请您妥善保管。

获取组织AccessKey

获取组织AccessKey的方法如下：

1. 管理员登录Apsara Uni-manager运营控制台。
2. 在顶部菜单栏，单击**企业**。
3. 在左侧导航栏中，选择**资源管理 > 组织管理**。
4. 在组织结构中，单击目标一级组织名称。
5. 单击**管理Accesskey**。
6. 在弹出的对话框中，查看组织AccessKey信息。