Alibaba Cloud Table Store

SDK リファレンス

Document Version20190523

目次

1 SDK の概要

Table Store SDK を使用する前に、次のことを行う必要があります。

- *Alibaba Cloud Table Store サービス*を有効にします。
- *AccessKey* を作成します。

このドキュメントでは、Table Store (旧 OTS) 用に Java SDK v4.0.0 以降をインストールして使用する方法について説明します。

言語	参考文献
Java	*Java SDK*
Python	*Python SDK*
C++	*C++ SDK*
Go	*Go SDK*
. .NET	*. .Net SDK*
NodeJS	*NodeJS SDK*

2 Python SDK

2.1 はじめに

このドキュメントは Table Store 用の Python SDK v4.x.x のインストール方法と使用方法を説明します。 Alibaba Cloud Table Store を有効化し、AccessKeyID と AccessKeySecret を作成したことを確認してください。

- Alibaba Cloud Table Store をまだ有効にしていない場合、またはこのサービスについて詳しく知りたい場合は、 *Table Store プロダクトのホームページ*をご参照ください。
- AccessKeyID と AccessKeySecret を作成していない場合は、 *AccessKey コンソール*にログインします。 AccessKeyを作成します。

SDKをダウンロードします。

- *SDK*
- *GitHub*

バージョン履歴の詳細については、 ここをクリックしてご参照ください。

互換性

- Table Store は Python SDK v4.x.x と互換性があります。

- Python SDK v2.x.x は順不同のプライマリキーをサポートしているため、Table Store は Python SDK v2.x.x と互換性がありません。 詳細な非互換性は次のとおりです。
  - SDK の名前が "ots2" から "tablestore" に変更されました。
  - Client.create_table オペレーションは、"TableOptions" パラメーターを使って追加されます。
  - put_row、get_row および update_row オペレーションの "primary_key" パラメーターは、順序を保証するために辞書型からリスト型に変更されています。
  - put_row および update_row オペレーションの "attribute_columns" パラメーターが辞書型からリスト型に変更されました。
  - put_row および update_row オペレーションの "attribute_columns" パラメーターがタイムスタンプ付きで追加されます。
  - get_row および get_range 操作は、"max_version" および"time_range" パラメーターを使用して追加されています。 少なくとも1つのパラメーターが存在する必要があります。
  - put_row、update_row および delete_row 操作は、"return_type" パラメーターを使用して追加されます。 現在サポートされているのは RT_PK のみです。これは、戻り値に現在行の PK 値が含まれていることを示します。
  - put_row、update_row および delete_rowオペレーションによって返された値は、return_row で加算されます。 要求で return_type が RT_PK として指定されている場合、return_row には現在行の PK 値が含まれます。

バージョン

最新バージョン: 4.3.0

2.2 インストール方法

前提条件

Python 2 と Python 3 に適用可能です。

インストール

- 方法 1: pip を使用して Python をインストール

  コマンドは次の通りです。

```
sudo pip install tablestore
```

- 方法 2: GitHub を使用して Python をインストール

  『*git*』がインストールされていることを確認してから、次のコマンドを実行します。

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

- 方法 3: ソースコードを使用して Python をインストール

  1. 『*Python SDK*』をダウンロードしてください。
  2. SDK を解凍してから、次のコマンドを実行します。

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

SDK を確認

"python"と入力します。コマンドラインで [Enter] キーを押してバージョンを確認します。

```
>>> import tablestore
>>> tablestore . __version_ _
' 4 . 3 . 7 '
```

SDK のアンインストール

pip を使用して Python SDK をアンインストールします。

```
sudo pip uninstall tablestore
```

2.3 単一行操作

Table Store SDK には、PutRow、GetRow、UpdateRow、DeleteRow の単一行操作APIが用意されています。

PutRow

指定された行にデータを挿入します。

API

```
        """
        説明: この操作は単一行のデータを書き込みます。 操作によって消費された容量
ユニットを返します。
         table_name は、テーブルの名前です。
```

```
        row は、プライマリキー列と属性列を含む行データを示します。
        condition は、操作が実行される前にチェックされるべき条件を示します。 条件が満たされると、操作が実行されます。 これは tablestore . metadata . Condition クラスのインスタンスです。 現在、この関数は行の存在チェックのみをサポートしています。 チェック条件には、' IGNORE '、' EXPECT_EXI ST ' および ' EXPECT_NOT _EXIST ' があります。
        return_typ e は、戻り値の型を示します。 これは tablestore . metadata . ReturnType クラスのインスタンスです。 現在、 PrimaryKey の戻り値のみがサポートされています。これは通常、プライマリキー列の自動増分で使用されます。

        戻り値: 操作によって消費された CapacityUn its 、および戻される行データです。
        consumed は、消費された CapacityUn its を示します。 これは tablestore . metadata . CapacityUn it クラスのインスタンスです。
        return_row は、戻された行データを示します。これにはプライマリキーと属性の列が含まれます。
        """
        def put_row ( self , table_name , row , condition = None , return_typ e = None )
```

例

データ行を挿入します。

```
    ## プライマリキーの最初のプライマリキー列は 1 の整数値で gid です。 2 番目のプライマリキー列は、整数値 101 で uid です。
    primary_ke y = [(' gid ', 1 ), (' uid ', 101 )]

    ## 4 つの属性列があります。
    ## 最初の属性列は " name " で、文字列値は " John " です。 バージョンは指定されず、現在のシステム時刻がバージョンとして使用されます。
    ## 2 番目の属性列は " mobile " で、整数値は 1510000000 0 です。 バージョンは指定されず、現在のシステム時刻がバージョンとして使用されます。
    ## 3 番目の属性列は " address " で、バイナリ値は " China " です。 バージョンは指定されず、現在のシステム時刻がバージョンとして使用されます。
    ## 4 番目の列は " age " で、値は 29 . 7 です。 バージョンは 1498184687 です。
    attribute_ columns = [(' name ',' John '), (' mobile ', 1510000000 0 ),(' address ', bytearray (' China ')),(' female ', False ), (' age ', 29 . 7 , 1498184687 000 )]

    ## primary_ke y と attribute_ columns を使った Row の構築
    row = Row ( primary_ke y , attribute_ columns )

    # 行条件: 期待行が存在しません。 期待される行が存在する場合は、エラー " Condition Update Update Failed " が戻されます。
    condition = Condition ( RowExisten ceExpectat ion . EXPECT_NOT _EXIST )

    try :
        # put_row メソッドを呼び出します。 ReturnType が指定されていない場合、 return_row は None です。
        consumed , return_row = client . put_row ( table_name , row , condition )

        #このリクエストによって消費された書き込み CU が表示されます。
        print (' put row succeed , consume % s write cu .' % consumed . write )
    # クライアントが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
    except OTSClientE rror as e :
```

```
            print " put row failed , http_statu s :% d ,
 error_mess age :% s " % ( e . get_http_s tatus (), e . get_error_
 message ())
    # サーバーが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが
原因です。
     except OTSService Error as e :
            print " put row failed , http_statu s :% d ,
 error_code :% s , error_mess age :% s , request_id :% s " % ( e
 . get_http_s tatus (), e . get_error_ code (), e . get_error_
 message (), e . get_reques t_id ())
```

注：

- RowExistenceExpectation.IGNORE は、指定された行が存在するかどうかに関係なく、新しいデータが挿入されることを示します。 挿入されたデータが既存のデータと同じ場合、既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_EXIST は、指定された行が存在する場合にのみ新しいデータが挿入されることを示します。 既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_NOT_EXIST は、指定された行が存在しない場合にのみデータが挿入されることを示します。
- 上記のサンプルプログラムでは、属性列 "age" のバージョンは 1498184687000 で、値は 2017年6月23日 です。 現在時刻から max_time_deviation の値 (テーブル作成時に指定) を引いた値が 1498184687000 より大きい場合、PutRow 操作は無効になります。
- 操作が成功した場合、例外はスローされません。操作が失敗した場合は、例外がスローされます。

コードの詳細は、『*PutRow@GitHub*』をご参照ください。

GetRow

指定されたプライマリキーに基づいて単一のデータ行を読み取ります。

API

```
        """
         説明: この操作は単一行のデータを読み取ります。
          table_name は、テーブルの名前です。
          primary_ke y は、プライマリキーを示します。型は dict です。
          columns_to _get は、リスト形式で読み込む列の名前を示すオプションのパ
ラメーターです。 入力しないと、すべての列が読み取られます。
          column_fil ter は、オプションのパラメーターで、指定された条件を満た
す行を読み込むことを示します。
          max_versio n は、オプションのパラメーターで、読み込みバージョンの最大
数を示します。
          time_range は、オプションのパラメーターで、指定された範囲内のバージョ
ンを読むこと、または指定されたバージョンを読むことを示します。 time_range と
 max_versio n は相互に排他的です。
```

```
        戻り値：この操作で消費された CapacityUn its 、プライマリキー列、および属性列です。
        consumed は、消費された CapacityUn its を示します。 これは tablestore . metadata . CapacityUn it クラスのインスタンスです。
        return_row は、リストデータ型の主キー列と属性列を含む行データを示します、たとえば、 [( ' PK0 ', value0 ), ( ' PK1 ', value1 )] です。
        next_token は、ワイド行読み込みの場合、次の読み込み操作の位置を示します。 値はバイナリ形式でエンコードされています。
        """
        def get_row ( self , table_name , primary_ke y , columns_to _get = None ,
                column_fil ter = None , max_versio n = None , time_range = None ,
                start_colu mn = None , end_column = None , token = None ):
```

例

データ行を読み取ります。

```
    # プライマリキーの最初の列は整数値 1 で uid です。 2 列目は整数値 101 で gid です。
    primary_ke y = [(' uid ', 1 ), (' gid ', 101 )]

    # 返される属性列： name 、 growth および type columns_to _get が [] の場合、すべての属性列が返されます。
    columns_to _get = [' name ', ' growth ', ' type ']

    # 列フィルタを追加：この列は、成長列の値が 0 . 9 ではない場合に返されます。
    cond = CompositeC olumnCondi tion ( LogicalOpe rator . AND )
    cond . add_sub_co ndition ( SingleColu mnConditio n (" growth ", 0 . 9 , Comparator Type . NOT_EQUAL ))
    cond . add_sub_co ndition ( SingleColu mnConditio n (" name ", ' Hangzhou ', Comparator Type . EQUAL ))

    try :
        # get_row クエリインターフェイスを呼び出します。 最後のパラメーター 1 は、 1 つのバージョンの値だけを返す必要があることを示します。
        consumed , return_row , next_token = client . get_row ( table_name , primary_ke y , columns_to _get , cond , 1 )
        print (' Read succeed , consume % s read cu .' % consumed . read )
        print (' Value of primary key : % s ' % return_row . primary_ke y )
        print (' Value of attribute : % s ' % return_row . attribute_ columns )
        for att in return_row . attribute_ columns :
            # 各列のキー、値、およびバージョンを表示します。
            print (' name :% s \ tvalue :% s \ ttimestamp :% d ' % ( att [ 0 ], att [ 1 ], att [ 2 ]))
    # クライアントが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
    except OTSClientE rror as e :
        print " get row failed , http_statu s :% d , error_mess age :% s " % ( e . get_http_s tatus (), e . get_error_ message ())
    # サーバが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
    except OTSService Error as e :
        print " get row failed , http_statu s :% d , error_code :% s , error_mess age :% s , request_id :% s " % ( e
```

```
. get_http_s  tatus (),  e . get_error_  code (),  e . get_error_
message (),  e . get_reques  t_id ())
```

注：

データ行をクエリすると、システムはその行のすべての列のデータを戻します。

columns_to_get パラメータを使用して、指定した列のデータを読み取ることができます。 たとえば、col0 と col1 が columns_to_get に挿入されている場合、システムは col0 と col1 のデータのみを戻します。

*GetRow@GitHub*でコードの詳細を取得します。

UpdateRow

指定された行のデータを更新します。 指定された行が存在しない場合は、新しい行が追加されます。 指定された行が存在する場合は、指定された列の値は、リクエスト内容に合わせて、追加、変更、又は削除されます。

API

```
        """
        説明: この操作は単一行のデータを更新します。
        ``  table_name  `` はテーブルの名前です。
        `` row `` はプライマリキー列（リスト型）と属性列（辞書型）を含む更新された行データを示します。
        `` condition `` は操作が実行される前にチェックされるべき条件を示します。 条件が満たされると、操作が実行されます。 これは  tablestore . metadata .
 Condition  クラスのインスタンスです。
        現在、この関数は行の存在チェックのみをサポートしています。 チェック条件には、' IGNORE '、' EXPECT_EXI  ST ' および ' EXPECT_NOT  _EXIST ' があります。
        `` return_typ  e `` は戻り値の型を示します。 これは  tablestore .
 metadata . ReturnType  クラスのインスタンスです。 現在、 PrimaryKey  の戻り値のみがサポートされています。これは通常、主キー列の自動増分で使用されます。
        戻り値: 操作によって消費された  CapacityUn  its  および返される行データ（ return_row によって示される）。
         consumed  は  CapacityUn  its  が消費されたことを示します。 これは
 tablestore . metadata . CapacityUn  it  クラスのインスタンスです。
         return_row  は、返される行データを示します。
        """
         def  update_row ( self ,  table_name ,  row ,  condition ,
 return_typ  e  =  None )
```

例

データ行を更新します。

```
    # プライマリキーの最初の列は整数値  1  で  uid  です。  2  列目は整数値
 101  で  gid  です。
     primary_ke  y  = [(' uid ', 1 ), (' gid ', 101 )]

    # アップデートには、 PUT 、 DELETE 、 DELETE_ALL  の 3 つの部分があります。
```

```
    # PUT : 2 列追加します。 最初の列は " name "、値は David です。 2 列目は " address " で、値は Hongkong です。
    # DELETE : バージョン 1488436949 003 の " address " 列の値を削除します。
    # DELETE_AL : " mobile " 列と " age " 列のすべてのバージョンの値を削除します。
     update_of_ attribute_ columns = {
        ' PUT ' : [(' name ',' David '), (' address ',' Hongkong ')],
        ' DELETE ' : [(' address ', None , 1488436949 003 )],
        ' DELETE_ALL ' : [(' mobile '), (' age ')],
    }
     row = Row ( primary_ke y , update_of_ attribute_ columns )

    # 行条件は Ignore です。これは、指定された行が存在するかどうかに関係なくデータが更新されることを示します。
     condition = Condition ( RowExisten ceExpectat ion . IGNORE ,
 SingleColu mnConditio n (" age ", 20 , Comparator Type . EQUAL
 )) # update row on \
 ly when this row is exist

     try :
         consumed , return_row = client . update_row ( table_name
 , row , condition )
    # クライアントが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
     except OTSClientE rror as e :
         print " update row failed , http_statu s :% d ,
 error_mess age :% s " % ( e . get_http_s tatus (), e . get_error_
 message ())
    # サーバーが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
     except OTSService Error as e :
         print " update row failed , http_statu s :% d ,
 error_code :% s , error_mess age :% s , request_id :% s " % ( e
 . get_http_s tatus (), e . get_error_ code (), e . get_error_
 message (), e . get_reques t_id ())
```

*UpdateRow@GitHub*でコードの詳細を取得します。

DeleteRow

API

```
        """
        説明: この操作は単一行のデータを削除します。
        `` table_name `` はテーブルの名前です。
        `` row `` は行データを示し、 delete_row の場合はプライマリキーのみを含みます。
        `` condition `` は操作が実行される前にチェックされるべき条件を示します。 条件が満たされると、操作が実行されます。 これは tablestore . metadata . Condition クラスのインスタンスです。
        現在、この関数は行の存在チェックのみをサポートしています。 チェック条件には、' IGNORE '、' EXPECT_EXI ST ' および ' EXPECT_NOT _EXIST ' があります。
        戻り値 操作によって消費された CapacityUn its 、および返される行データ ( return_row によって示されます)。
         consumed は CapacityUn its が消費されたことを示します。 これは tablestore . metadata . CapacityUn it クラスのインスタンスです。
         return_row は、返される行データを示します。
        """
```

```
        def  delete_row ( self ,  table_name ,  row ,  condition ,
return_typ  e  =  None ):
```

例

データ行を削除します。

```
    primary_ke  y  = [(' gid ', 1 ), (' uid ',' 101 ')]
    row  =  Row ( primary_ke  y )
    try :
        consumed ,  return_row  =  client . delete_row ( table_name
,  row ,  None )
    # クライアントが異常です。これは通常、誤ったパラメーターまたはネットワークエ
ラーが原因です。
    except  OTSClientE  rror  as  e :
        print  " update  row  failed ,  http_statu  s :% d ,
error_mess  age :% s " % ( e . get_http_s  tatus (),  e . get_error_
message ())
    # サーバーが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが
原因です。
    except  OTSService  Error  as  e :
        print  " update  row  failed ,  http_statu  s :% d ,
error_code :% s ,  error_mess  age :% s ,  request_id :% s " % ( e
. get_http_s  tatus (),  e . get_error_  code (),  e . get_error_
message (),  e . get_reques  t_id ())
    print  (' Delete  succeed ,  consume  % s  write  cu .' %
consumed . write )
```

コードの詳細は、『DeleteRow@GitHub』をご参照ください。

2.4 テーブルレベルの操作

Table Store SDK には、テーブルを操作するため
に、CreateTable、ListTable、UpdateTable、DescribeTable および DeleteTable という API が用意されています。

CreateTable

与えられたテーブル構造情報に基づいてテーブルを作成します。

Table Store でテーブルを作成するときは、テーブルのプライマリキーを指定する必要があります。 プライマリキーには 1 つから 4 つのプライマリキー列があります。 各プライマリキー列には名前と型があります。

API

```
        説明: この操作は、入力されたテーブル情報に基づいてテーブルを作成します。

        table_meta  は  tablestore . metadata . TableMeta  クラスのイン
スタンスです。 テーブル名とプライマリキースキーマが含まれています。
        詳しくは、 TableMeta  クラスのドキュメントをご参照ください。 テーブルが
作成された後、パーティションをロードするのに通常約  1  分かかります。 その後、テー
ブル操作を実行できます。
```

```
        table_opti ons は、 tablestore . metadata . TableOptio
ns クラスのインスタンスです。 time_to_li ve 、 max_versio n および
max_time_d eviation が含まれています。
        reserved_t hroughput は tablestore . metadata .
ReservedTh roughput クラスのインスタンスです。 予約済み読み書きスループットを
示します。

        Return : Null .

        def create_tab le ( self , table_meta , reserved_t
hroughput ):
```

注:

テーブルが Table Store に作成された後、テーブルのロードに数秒かかります。 この間は何の操作もしないでください。

例

プライマリキー列が 2 つ、TTL が 1 年間 (つまり 60 * 60 * 24 * 365 = 31536000 秒)、MaxVersions が 3、MaxTimeDeviation が 1 日 (つまり 86400 秒) および 予約済み読み書きスループットが (0,0) のテーブルを作成します。

```
    # プライマリキーの数、名前、種類など、プライマリキー列のスキーマを作成します。
    # 最初のプライマリキー列（シャーディング列）の名前は pk0 で整数型です。
    # 2 番目のプライマリキー列は pk1 という名前で整数型です。 String 型ま
たは Binary 型も使用できます。
     schema_of_ primary_ke y = [(' pk0 ', ' INTEGER '), (' pk1 ', '
INTEGER ')]

    # テーブル名とプライマリキー列に基づいて tableMeta クラスを作成します。
     table_meta = TableMeta (' SampleTabl e ', schema_of_
primary_ke y )

    # TableOptio ns を作成します。 TTL を 31536000 秒に設定し（期限
切れになるとデータは自動的に消去されます)、 MaxVersion s を 3 に、そして
MaxTimeDev iation を 1 日に設定します。
     table_opti ons = TableOptio ns ( 31536000 , 3 , 86400 )

    # 予約済み読み取りスループットと予約済み書き込みスループットを 0 に設定しま
す。
     reserved_t hroughput = ReservedTh roughput ( CapacityUn it (
0 , 0 ))

    # クライアントの create_tab le API を呼び出します。 例外がスローされ
なければ、実行は成功します。そうでなければ、実行は失敗します。
     try :
         ots_client . create_tab le ( table_meta , table_opti ons
, reserved_t hroughput )
         print " create table succeeded "
    # 例外処理
     except Exception :
         print " create table failed ."
```

コードの詳細は、『CreateTable@GitHub』をご参照ください。

ListTable

現在のインスタンスの下にあるすべてのテーブルの名前を取得します。

API

```
        """
        説明：この操作はすべてのテーブル名をリストします。
        戻り値：テーブル名のリスト。
         table_list はテーブル名のリストを表します。型は tuple です。 たとえば、( ' MyTable1 ', ' MyTable2 ') です。
        """
         def list_table ( self ):
```

例

インスタンスの下にあるすべてのテーブルの名前を取得します。

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

コードの詳細は、『*ListTable@GitHub*』をご参照ください。

UpdateTable

指定されたテーブルの予約済み読み取りまたは書き込みのスループット値を更新します。

API

```
    """
        説明：この操作はテーブル属性を更新します。 現在、この関数は予約済み読み書きのスループットを変更することしかできません。
         table_name は、テーブルの名前です。
         table_opti ons は、 tablestore . metadata . TableOptio ns クラスのインスタンスです。 time_to_li ve 、 max_versio n および max_time_d eviation が含まれています。
         reserved_t hroughput は ots2 . metadata . ReservedTh roughput クラスのインスタンスです。 これは予約済み読み書きスループットを示します。

        戻り値：このテーブルの予約済みスループットが最後に増減した回数、および今日減少した回数。

         update_tab le_respons e は更新結果を示します。 これは ots2 . metadata . UpdateTabl eResponse クラスのインスタンスです。
        """
         def update_tab le ( self , table_name , table_opti ons , reserved_t hroughput ):
```

例

最大バージョンを 5 に更新します。

```
        # 予約済み読み取りスループットを 0 に、予約済み書き込みスループットを
0 に更新します。
        reserved_t hroughput = ReservedTh roughput ( CapacityUn
it ( 0 , 0 ))

        # TableOptio ns を作成します。 TTL を 31536000 秒に設定し
（期限切れになるとデータは自動的に消去されます）、 MaxVersion s を 3 に、そ
して MaxTimeDev iation を 1 日に設定します。
        table_opti ons = TableOptio ns ( 31536000 , 5 , 86400 )

        try :
            # API を呼び出して、テーブルの予約済み読み書きスループットを更新し
ます。
            ots_client . update_tab le (' SampleTabl e ',
reserved_t hroughput )

            # 例外がスローされなければ実行は成功です。
            print " update table succeeded "
        except Exception :
            # 例外がスローされると呼び出しは失敗し、エラーメッセージが表示されま
す。
            print " update table failed "
```

コードの詳細は、『*UpdateTable@GitHub*』をご参照ください。

DescribeTable

指定されたテーブルの構造情報と予約済み読み書きスループット値を問い合わせます。

API

```
        """
        説明: この操作はテーブルの説明を取得します。
        table_name はテーブルの名前です。
        戻り値: テーブルの説明。
        describe_t able_respo nse はテーブルの説明を表します。 これは
ots2 . metadata . DescribeTa bleRespons e クラスのインスタンスです。
        """
        def describe_t able ( self , table_name ):
```

例

テーブルの説明情報を取得します。

```
    try :
        describe_r esponse = ots_client . describe_t able ('
myTable ')
        # 例外がスローされない場合、実行は成功し、次の表の情報が表示されます。
        print " describe table succeeded ."
        print (' TableName : % s ' % describe_r esponse .
table_meta . table_name )
```

```
        print ('PrimaryKey: %s' % describe_response.table_meta.schema_of_primary_key)
        print ('Reserved read throughput: %s' % describe_response.reserved_throughput_details.capacity_unit.read)
        print ('Reserved write throughput: %s' % describe_response.reserved_throughput_details.capacity_unit.write)
        print ('Last increase throughput time: %s' % describe_response.reserved_throughput_details.last_increase_time)
        print ('Last decrease throughput time: %s' % describe_response.reserved_throughput_details.last_decrease_time)
        print ('table options\'s time to live: %s' % describe_response.table_options.time_to_live)
        print ('table options\'s max version: %s' % describe_response.table_options.max_version)
        print ('table options\'s max_time_deviation: %s' % describe_response.table_options.max_time_deviation)
    except Exception:
        # 例外がスローされると、操作は失敗し、例外が処理されます。
        print "describe table failed."
```

コードの詳細は、『*DescribeTable@GitHub*』をご参照ください。

DeleteTable

インスタンス下の指定されたテーブルを削除します。

API

```
        """
        説明：この操作は、指定されたテーブル名に対応するテーブルを削除します。
        table_name はテーブルの名前です。
        戻り値：null 値
        """
        def delete_table(self, table_name):
```

例

テーブルを削除します。

```
try:
    # テーブル SampleTable を削除するために API を呼び出します
    ots_client.delete_table('SampleTable')
    # 例外がスローされなければ、操作は成功です。
    print "delete table succeeded"
    # 例外がスローされた場合、操作は失敗し、例外が処理されます。
except Exception:
    print "delete table failed"
```

コードの詳細は、『*DeleteTable@GitHub*』をご参照ください。

2.5 複数行操作

Table Store SDK は、 BatchGetRow、BatchWriteRow および GetRange の複数行操作 API を提供します。

BatchGetRow

1 つ以上のテーブルから複数のデータ行をバッチで読み取ります。

BatchGetRow 操作は、基本的に複数の GetRow 操作のセットです。 各操作が実行され、結果が返され、容量ユニットが個別に消費されます。

多数の GetRow 操作の実行と比較して、BatchGetRow 操作はリクエストの応答時間を効率的に短縮し、データ読み取り速度を向上させます。

API

```
        """
        説明: このオペレーションバッチは複数の行からデータを読み込みます。
         request  =  BatchGetRo  wRequest ()
         request . add ( TableInBat  chGetRowIt  em ( myTable0 ,
primary_ke  ys ,  column_to_  get = None ,  column_fil  ter = None ))
         request . add ( TableInBat  chGetRowIt  em ( myTable1 ,
primary_ke  ys ,  column_to_  get = None ,  column_fil  ter = None ))
         request . add ( TableInBat  chGetRowIt  em ( myTable2 ,
primary_ke  ys ,  column_to_  get = None ,  column_fil  ter = None ))
         request . add ( TableInBat  chGetRowIt  em ( myTable3 ,
primary_ke  ys ,  column_to_  get = None ,  column_fil  ter = None ))
         response  =  client . batch_get_  row ( request )
         response  は  tablestore . metadata . BatchGetRo  wResponse
 型の返された結果を示します。
        """
        def   batch_get_  row ( self ,  request ):
```

例

この例では、3 行のデータがバッチ読み取りされます。

```
    # 返される列です。
     columns_to  _get  = [' name ', ' mobile ', ' address ', ' age ']

    #  3 行読みます。
     rows_to_ge  t  = []
     for   i   in   range ( 0 ,  3 ):
         primary_ke  y  = [(' gid ', i ), (' uid ', i + 1 )]
         rows_to_ge  t . append ( primary_ke  y )

    # フィルタ条件: 名前は " Johb "、住所は " China " です。
     cond  =  CompositeC  olumnCondi  tion ( LogicalOpe  rator . AND )
     cond . add_sub_co  ndition ( SingleColu  mnConditio  n (" name ",
" John ",  Comparator  Type . EQUAL ))
     cond . add_sub_co  ndition ( SingleColu  mnConditio  n (" address
", ' China ',  Comparator  Type . EQUAL ))

    # バッチ読み込み要求を作成します。
```

```
    request = BatchGetRo wRequest ()

    # テーブル追加: table_name の示すテーブルから読み込む行です。 最後のパラメーター " 1 " は最新バージョンの読み取りを示します。
    request . add ( TableInBat chGetRowIt em ( table_name , rows_to_ge t , columns_to _get , cond , 1 ))

    # テーブル追加: notExistTa ble の示すテーブルから読み込む行。
    request . add ( TableInBat chGetRowIt em (' notExistTa ble ', rows_to_ge t , columns_to _get , cond , 1 ))

    try :
        result = client . batch_get_ row ( request )
        print (' Result status : % s '%( result . is_all_suc ceed ()))

        table_resu lt_0 = result . get_result _by_table ( table_name )
        table_resu lt_1 = result . get_result _by_table (' notExistTa ble ')
        print (' Check first table \' s result :')
        for item in table_resu lt_0 :
            if item . is_ok :
                print (' Read succeed , PrimaryKey : % s , Attributes : % s ' % ( item . row . primary_ke y , item . row . attribute_ columns ))
            else :
                print (' Read failed , error code : % s , error message : % s ' % ( item . error_code , item . error_mess age ))
        print (' Check second table \' s result :')
        for item in table_resu lt_1 :
            if item . is_ok :
                print (' Read succeed , PrimaryKey : % s , Attributes : % s ' % ( item . row . primary_ke y , item . row . attribute_ columns ))
            else :
                print (' Read failed , error code : % s , error message : % s ' % ( item . error_code , item . error_mess age ))
    # The client is abnormal , which is typically due to incorrect parameters or a network error .
    except OTSClientE rror as e :
        print " get row failed , http_statu s :% d , error_mess age :% s " % ( e . get_http_s tatus (), e . get_error_ message ())
    # サーバーが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
    except OTSService Error as e :
        print " get row failed , http_statu s :% d , error_code :% s , error_mess age :% s , request_id :% s " % ( e . get_http_s tatus (), e . get_error_ code (), e . get_error_ message (), e . get_reques t_id ())
```

コードの詳細は、『*BatchGetRow@GitHub*』をご参照ください。

BatchWriteRow

1つ以上のテーブル内の複数のデータ行をまとめて挿入、変更、または削除します。

BatchWriteRow 操作は、基本的に複数の PutRow、UpdateRow および DeleteRow 操作のセットです。 各操作が実行され、結果が戻され、容量ユニットが個別に消費されます。

API

```
        """
        説明: この操作バッチは、複数行のデータをバッチ変更します。
         request  =  MiltiTable  InBatchWri  teRowItem ()
         request . add ( TableInBat  chWriteRow  Item ( table0 ,
row_items ))
         request . add ( TableInBat  chWriteRow  Item ( table1 ,
row_items ))
         response  =  client . batch_writ  e_row ( request )
         response  は  tablestore . metadata . BatchWrite  RowRespons
e  型の戻された結果を示します。
        """
         def   batch_writ  e_row ( self ,  request ):
```

例

この例では、データはバッチ書き込みされます。

```
     put_row_it  ems  = []
    ##   PutRow  で行を追加します。
     for   i   in   range ( 0 ,  10 ):
         primary_ke  y  = [(' gid ', i ), (' uid ', i + 1 )]
         attribute_  columns  = [(' name ',' somebody '+ str ( i )),
(' address ',' somewhere '+ str ( i )), (' age ', i )]
         row  =  Row ( primary_ke  y ,  attribute_  columns )
         condition  =  Condition ( RowExisten  ceExpectat  ion .
IGNORE )
         item  =  PutRowItem ( row ,  condition )
         put_row_it  ems . append ( item )

    ##  UpdateRow  で行を追加します。
     for   i   in   range ( 10 ,  20 ):
         primary_ke  y  = [(' gid ', i ), (' uid ', i + 1 )]
         attribute_  columns  = {' put ': [(' name ',' somebody '+ str
( i )), (' address ',' somewhere '+ str ( i )), (' age ', i )]}
         row  =  Row ( primary_ke  y ,  attribute_  columns )
         condition  =  Condition ( RowExisten  ceExpectat  ion .
IGNORE ,  SingleColu  mnConditio  n (" age ",  i ,  Comparator  Type
. EQUAL ))
         item  =  UpdateRowI  tem ( row ,  condition )
         put_row_it  ems . append ( item )

    ##  DeleteRow で行を追加します。
     delete_row  _items  = []
     for   i   in   range ( 10 ,  20 ):
         primary_ke  y  = [(' gid ', i ), (' uid ', i + 1 )]
         row  =  Row ( primary_ke  y )
         condition  =  Condition ( RowExisten  ceExpectat  ion .
IGNORE )
         item  =  DeleteRowI  tem ( row ,  condition )
         delete_row  _items . append ( item )

    # バッチ書き込み要求を作成します。
     request  =  BatchWrite  RowRequest ()
```

```
    request . add ( TableInBat  chWriteRow  Item ( table_name ,
put_row_it  ems ))
    request . add ( TableInBat  chWriteRow  Item (' notExistTa  ble
',  delete_row  _items ))

    # バッチ書き込み操作を実行するには、 batch_writ  e_row  メソッドを呼び出し
ます。 要求パラメーターが正しくない場合は、例外がスローされます。 データが特定の行に
書き込まれなかった場合、内部項目は失敗しますが、例外はスローされません。
    try :
        result  =  client . batch_writ  e_row ( request )
        print  (' Result  status : % s '%( result . is_all_suc  ceed
()))

        ##  PutRow  の結果を確認します。
        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_mess  age ))

        ##  UpdateRow  の結果を確認します。
        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_mess  age ))

        ##  DeleteRow  の結果を確認します。
        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_mess  age ))
    # クライアントが異常です。これは通常、誤ったパラメーターまたはネットワークエ
ラーが原因です。
    except  OTSClientE  rror  as  e :
        print  " get  row  failed ,  http_statu  s :% d ,
error_mess  age :% s " % ( e . get_http_s  tatus (),  e . get_error_
message ())
    # サーバーが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが
原因です。
    except  OTSService  Error  as  e :
        print  " get  row  failed ,  http_statu  s :% d ,
error_code :% s ,  error_mess  age :% s ,  request_id :% s " % ( e
. get_http_s  tatus (),  e . get_error_  code (),  e . get_error_
message (),  e . get_reques  t_id ())
```

コードの詳細は、『*BatchWriteRow@GitHub*』をご参照ください。

GetRange

指定されたプライマリキー範囲内のデータを読み取ります。

API

```
        """
        説明: この操作は、指定された範囲内の複数の行からデータを読み取ります。
         table_name  はテーブルの名前です。
         direction  は範囲の方向を示します。文字列フォーマット、値: ' FORWARD
 ' または  ' BACKWARD '。
         inclusive_  start_prim  ary_key  は 始まりのプライマリキーです（包
括的）。
         exclusive_  end_primar  y_key  は終了のプライマリキーです（排他
的）。
         columns_to  _get  はリスト形式で読み込む列の名前を示すオプションのパラ
メーターです。 入力しないと、すべての列が読み取られます。
         limit  は読み込む行の最大数を示すオプションのパラメーターです。 入力しな
い場合、読み取る行数は制限されません。
         column_fil  ter  はオプションのパラメーターで、指定された条件を満たす
行を読み込むことを示します。
         max_versio  n  はオプションのパラメーターで、返されるバージョンの最大数
を示します。  max_versio  n  と  time_range  は相互に排他的です。
         time_range  はオプションのパラメーターで、返されるバージョンの範囲を示
します。  time_range  と  max_versio  n  は相互に排他的です。
         start_colu  mn  はオプションのパラメーターで、ワイドロー読み込みの場
合、現在の読み込み操作の開始列を示します。
         end_column  はオプションのパラメーターで、ワイドロー読み込みの場合、現
在の読み込み操作の終了列を示します。
         token  はオプションのパラメーターで、ワイドロー読み込みの場合、現在の
リードオペレーションの開始列の位置を示します。 コンテンツはバイナリ形式でエンコード
されており、前のリクエストの結果から返されます。

        戻り値: 条件を満たすデータの結果リスト。
         consumed  は、操作によって消費された  CapacityUn  its  を示しま
す。 これは  tablestore . metadata . CapacityUn  it  クラスのインスタンスです。
         next_start  _primary_k  ey  は、次の  get_range  操作を開始するプ
ライマリキー列の値を示します。型:  dict 。
         row_list  は戻された行データのリストを [ Row , ...] の形式で示しま
す。
        """
    def  get_range ( self ,  table_name ,  direction ,
                inclusive_  start_prim  ary_key ,
                exclusive_  end_primar  y_key ,
                columns_to  _get = None ,
                limit = None ,
                column_fil  ter = None ,
                max_versio  n = None ,
                time_range = None ,
                start_colu  mn = None ,
                end_column = None ,
                token  =  None ):
```

例

指定された範囲のデータを読み込みます。

```
    # クエリ範囲のプライマリキーの先頭
     inclusive_  start_prim  ary_key  = [(' uid ', INF_MIN ), (' gid
', INF_MIN )]

    # クエリ範囲の最後のプライマリキー
     exclusive_  end_primar  y_key  = [(' uid ', INF_MAX ), (' gid ',
INF_MAX )]
```

```
    # すべての列を問い合わせる
     columns_to  _get  = []

    # 一度に最大  90  件の結果が戻されます。 最初のクエリで合計  100  件の結果が戻され、上限が  90  件に設定されている場合、最初に最大  90  件、最小  0  件の結果が戻される可能性がありますが、  next_start  _primary_k  ey  は  None  ではありません。
     limit  =  90

    # フィルターを設定します。
     cond  =  CompositeC  olumnCondi  tion ( LogicalOpe  rator . AND )
     cond . add_sub_co  ndition ( SingleColu  mnConditio  n (" address ", ' China ',  Comparator  Type . EQUAL ))
     cond . add_sub_co  ndition ( SingleColu  mnConditio  n (" age ", 50 ,  Comparator  Type . LESS_THAN ))

     try :
        #  get_range  インターフェイスを呼び出します。
         consumed ,  next_start  _primary_k  ey ,  row_list , next_token   =  client . get_range (
                table_name ,  Direction . FORWARD ,
                inclusive_  start_prim  ary_key ,  exclusive_ end_primar  y_key ,
                columns_to  _get ,
                limit ,
                column_fil  ter  =  cond ,
                max_versio  n  =  1
    )

         all_rows  = []
         all_rows . extend ( row_list )

        #  next_start  _primary_k  ey  の値が  null  値ではない場合、さらにデータが存在し、読み取りが続行されます。
         while  next_start  _primary_k  ey  is  not  None :
             inclusive_  start_prim  ary_key  =  next_start _primary_k  ey
             consumed ,  next_start  _primary_k  ey ,  row_list , next_token  =  client . get_range (
                 table_name ,  Direction . FORWARD ,
                 inclusive_  start_prim  ary_key ,  exclusive_ end_primar  y_key ,
                 columns_to  _get ,  limit ,
                 column_fil  ter  =  cond ,
                 max_versio  n  =  1
        )
             all_rows . extend ( row_list )

        # プライマリキー列と属性列を表示します。
         for  row  in  all_rows :
             print  ( row . primary_ke  y ,  row . attribute_  columns )
         print  (' Total  rows : ',  len ( all_rows ))
    # クライアントが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
     except  OTSClientE  rror  as  e :
         print  " get  row  failed ,  http_statu  s :% d , error_mess  age :% s " % ( e . get_http_s  tatus (),  e . get_error_ message ())
    # サーバーが異常です。これは通常、誤ったパラメーターまたはネットワークエラーが原因です。
     except  OTSService  Error  as  e :
```

```
        print " get row failed , http_statu s :% d ,
error_code :% s , error_mess age :% s , request_id :% s " % ( e
. get_http_s tatus (), e . get_error_ code (), e . get_error_
message (), e . get_reques t_id ())
```

コードの詳細は、『GetRangeRow@GitHub』を参照します。

2.6 エラー処理

方法

現在、Table Store Python SDK は `Exception` メソッドを採用して、エラー処理をしています。 呼び出されたインターフェイスが例外をスローしない場合、操作は成功します。 例外がスローされると、操作は失敗します。

注：
batch_get_row や batch_write_row などのバッチ操作インターフェイスは、システムが各行のステータスが正常であることを確認した場合にのみ正常に呼び出されます。

例外

Table Store Python SDK には、Exception から継承した OTSClientError と OTSService Error の 2 種類の例外があります。

- OTSClientError: 不正なパラメーター値や解析結果の返却失敗など、SDK の内部例外を示します。
- OTSServiceError: サーバーエラーメッセージの解析によって生成されたサーバーエラーを示します。 OTSServiceError には以下のコンポーネントがあります。
  - get_http_status: 返された HTTP コード、例えば 200 または 404です。
  - get_error_code: Table Store から返されたエラータイプの文字列です。
  - get_error_message: Table Store から返されるエラーメッセージ文字列です。
  - get_request_id: リクエストを一意に識別する UUID です。 問題が解決しない場合は、RequestId を保存して チケットを起票し、サポートセンターへお問い合わせください。

再試行

- SDK でエラーが発生した場合、システムは操作を再試行します。 デフォルトでは、操作は最大 3 秒間隔で 20 回再試行されます。 スロットルエラーおよび読み取り関連の内部サーバーエラーに対してシステムが操作を再試行する方法については、 *tablestore / lib / retry . js* をご参照ください。

- RetryPolicy を継承して再試行ポリシーをカスタマイズし、OTSClient オブジェクトを構築するときにそれをパラメーターとして渡すことができます。

SDK には、次の再試行ポリシーがあります。

- DefaultRetryPolicy: デフォルトの再試行ポリシー。 読み取り操作のみが最大 3 秒間隔で 20 回再試行されます。
- NoRetryPolicy: 再試行しません。
- NoDelayRetryPolicy: 遅延のない再試行ポリシーです。 このポリシーは慎重に使用する必要があります。
- WriteRetryPolicy: 書き込み操作はデフォルトの再試行ポリシーに基づいて再試行されます。

3 Java SDK

3.1 はじめに

はじめに

このドキュメントでは、Table Store に Java SDK v4.0.0 以降をインストールして使用する方法について説明します。 このドキュメントでは、Alibaba Cloud Table Store を有効にして AccessKeyId と AccessKeySecret を作成したと想定しています。

- Table Store をまだ有効にしていない場合、または Table Store についてもっと知りたい場合は、 「*Table Store 製品のホームページ*」をご参照ください。
- AccessKeyId と AccessKeySecret を作成していない場合は、 *Alibaba Cloud アクセスキーコンソール*にログインして、 AccessKey を作成します。

特記事項

SDK バージョン 4.0.0 以降では、複数のデータバージョンと TimeToLive がサポートされています。 ただし、これらの SDK バージョンは SDK v2.x.x と互換性がありません。

- *TimeToLive (TTL) の追加*
- 複数のデータバージョンの追加

SDK パッケージのダウンロード

- SDK: "*tablestore-4.7.4-release.zip*"

バージョンサイクルについては、 *[ここ]* をクリックしてください。

バージョン

最新バージョン: 4.7.4

- 互換性

  SDK v4.x.x の場合:

  - 互換性あり

  SDK 2.x.x の場合:

  - 互換性なし

- 変更の説明
  - 主キー列に自動インクリメント機能を追加しました。

3.2 インストール

前提条件

JDK 6 以降に適用されます。

インストール方法

- Maven を使用して Java SDK をインストールする

  Maven で Table Store Java SDK を使用するには、対応する依存関係を `" pom . xml "` ファイルに追加するだけです。 例として Java SDK v4.7.4 が使用されています。 「依存関係」セクションに次の内容を入力します。

```
< dependency >
    < groupId > com . aliyun . openservic  es </ groupId >
    < artifactId > tablestore </ artifactId >
    < version > 4 . 3 . 1 </ version >
</ dependency >
```

- JAR パッケージを Eclipse にインポートして Java SDK をインストールする

  Java SDK v4.7.4 の場合、プロセスは次のとおりです。

  1. *Java SDK* をダウンロードします。
  2. SDK を解凍します。
  3. `" tablestore -< versionId >. jar "` ファイルを解凍したフォルダーにコピーし、"lib" フォルダーにあるすべてのファイルをプロジェクトにコピーします。
  4. Eclipse でプロジェクトを右クリックし、 [プロパティ] > [Java ビルドパス] > [JAR を追加] をコンテキストメニューから選択します。
  5. 手順 3 でコピーしたすべての JAR ファイルを選択します。

  上記の手順を完了したら、Eclipse で Table Store Java SDK を使用できます。

サンプルプログラム

Table Store Java SDK には、参照や使用のためにさまざまなサンプルプログラムが用意されています。 次のようにサンプルプログラムを入手することができます。

1. Table Store Java SDK をダウンロードして解凍します。
2. *" examples "* フォルダにサンプルプログラムがあります。

3.3 初期化

OTSClient は Table Store のクライアントです。 呼び出し側に、テーブルを操作し、単一行または複数の行に対してデータを読み書きするための一連の方法を提供します。 Java SDK を使用して Table Store へのリクエストを開始するには、OTSClient インスタンスを初期化し、必要に応じて ClientConfiguration のデフォルト設定を変更する必要があります。

エンドポイントを決定する

エンドポイントは、リージョン内の Alibaba Cloud Table Store のドメインです。 以下のフォーマットに対応しています。

例	説明
「 http :// sun . cn - hangzhou . ots . aliyuncs . com 」	HTTP プロトコルを使用してインターネット経由で杭州の sun インスタンスにアクセスします。
「 https :// sun . cn - hangzhou . ots . aliyuncs . com 」	HTTPS プロトコルを使用してインターネット経由で杭州の sun のインスタンスにアクセスします。

: インスタンスにはイントラネット経由でもアクセスできます。

Table Store インスタンスが存在するエンドポイントを照会するには、次の手順に従います。

1. *Alibaba Cloud Table Store コンソール*にログインします。
2. 「インスタンスの詳細」にアクセスします。インスタンスのエンドポイントであるインスタンスアクセス URL を見つけます。

AccessKey を設定する

Alibaba Cloud Table Store サービスにアクセスするには、署名認証用の有効な AccessKey が必要です。 以下のタイプのアクセスキーがサポートされています。

- プライマリアカウントの AccessKeyID と AccessKeySecret。 基本的な手順は次の通りです：
  1. Alibaba Cloud Web サイトで *Alibaba Cloud アカウント*を登録します。
  2. *AccessKey コンソール*にログインします。 AccessKey を申請します。

- Table Store へのアクセスを許可されている RAM ユーザーの AccessKeyID および AccessKeySecret。 基本的な手順は次の通りです。
  1. プライマリアカウントを使用して *RAM にアクセス*します。 RAM ユーザーを作成するか、既存の RAM ユーザーを使用します。
  2. プライマリアカウントを使用して、RAM ユーザーに Table Store へのアクセスを許可します。

     認証後、RAM ユーザーの AccessKeyID と AccessKeySecret を使用してTable Store にアクセスできます。
- 一時アクセス用の STS トークン。 トークン取得プロセスは次のとおりです。
  1. アプリケーションサーバーは RAM/STS サーバーにアクセスして、一時的なAccessKeyID、AccessKeySecrets およびトークンを取得し、ユーザーに送信します。
  2. 一時的な AccessKeyID、AccessKeySecret およびトークンを使用して Table Store にアクセスします。

初期化

AccessKeyID と AccessKeySecret を取得したら、OTSClient インスタンスを初期化する必要があります。

クライアントを作成する

Table Store SDK を使用する場合は、クライアントを構築してから、クライアント API を呼び出して Table Store サービスにアクセスする必要があります。 クライアント API 機能は、Table Store が提供する RESTful API に似ています。

Table Store SDK の最新バージョンには、SyncClient と AsyncClient の2種類のクライアントがあり、それぞれ同期 API と非同期 API 用に設計されています。

同期 API 呼び出しが完了すると、リクエスが実行されます。 同期 API を呼び出して、Table Store のさまざまな機能について学ぶことができます。 同期 API と比較して、非同期 API はより優れた柔軟性を提供します。 高いパフォーマンス要件がある場合は、ビジネスニーズに応じて、非同期 API の使用とマルチスレッドの使用のどちらかを選択できます。

> 注 :
> SyncClient と AsyncClient はどちらもセキュアスレッドで、管理スレッドと接続リソースが含まれています。 あまりにも多くのクライアントを作成することは推奨しません。 通常、グローバルクライアント 1 つで十分です。

サンプルコード

1. デフォルト設定を使用して SyncClient を作成します。

```
        final   String   endPoint  = "";
        final   String   accessKeyI  d  = "";
        final   String   accessKeyS  ecret  = "";
        final   String   instanceNa  me  = "";

        SyncClient   client  =  new   SyncClient ( endPoint ,
accessKeyI  d ,  accessKeyS  ecret ,  instanceNa  me );
```

2. カスタム設定を使用して SyncClient を作成します。

```
//  ClientConf  iguration  は複数の設定項目を提供します。 一般的に使用される
項目は次のとおりです。
        ClientConf  iguration   clientConf  iguration  =  new
 ClientConf  iguration ();
       // 接続確立タイムアウトを設定します。
        clientConf  iguration . setConnect  ionTimeout  InMillisec
ond ( 5000 );
       // ソケットのタイムアウトを設定します。
        clientConf  iguration . setSocketT  imeoutInMi  llisecond (
5000 );
       // 再試行ポリシーを設定します。 これが設定されていない場合は、デフォル
トの再試行ポリシーが使用されます。
        clientConf  iguration . setRetrySt  rategy ( new
AlwaysRetr  yStrategy ());
        SyncClient   client  =  new   SyncClient ( endPoint ,
accessId ,  accessKey ,
                instanceNa  me ,  clientConf  iguration );
```

HTTPS

HTTPS 用の Java 7 にアップグレードします。

マルチスレッド

- マルチスレッドがサポートされています。
- そのマルチスレッドは同じ OTSClient オブジェクトを使用することを推奨します。

3.4 単一行操作

Table Store には、PutRow、GetRow、UpdateRow、DeleteRow の単一行操作 API が用意されています。

PutRow

PutRow API は、データ行を挿入するために使用されます。 行が既に存在している場合は上書きされます。

PutRow を実行するときは、条件付き更新機能を使用して、書き込まれる行の存在、または、既存の行の特定の列の値に対する条件を設定できます。 詳しくは、「条件付き更新」をご参照ください。

例 1

この例では、10 個の属性列を持つ行が書き込まれ、各列に 1 つのバージョンが入力されています。 バージョン番号 (タイムスタンプ) はサーバーによって指定されます。

```
private static void putRow ( SyncClient client , String
pkValue ) {
    // プライマリキーを構築します。
     PrimaryKey Builder primaryKey Builder = PrimaryKey Builder
. createPrim aryKeyBuil der ();
     primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE Y_NAME
, PrimaryKey Value . fromString ( pkValue ));
     PrimaryKey primaryKey = primaryKey Builder . build ();

     RowPutChan ge rowPutChan ge = new RowPutChan ge (
TABLE_NAME , primaryKey );

    // 属性列を追加します。
     for ( int i = 0 ; i < 10 ; i ++) {
         rowPutChan ge . addColumn ( new Column (" Col " + i ,
ColumnValu e . fromLong ( i )));
    }

     client . putRow ( new PutRowRequ est ( rowPutChan ge ));
}
```

例 2

この例では、10 個の属性列を持つ行が書き込まれ、各列に 3 つのバージョンが入力されています。 バージョン番号 (タイムスタンプ) はクライアントによって指定されます。

```
private static void putRow ( SyncClient client , String
pkValue ) {
    // プライマリキーを構築します。
     PrimaryKey Builder primaryKey Builder = PrimaryKey Builder
. createPrim aryKeyBuil der ();
     primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE Y_NAME
, PrimaryKey Value . fromString ( pkValue ));
     PrimaryKey primaryKey = primaryKey Builder . build ();

     RowPutChan ge rowPutChan ge = new RowPutChan ge (
TABLE_NAME , primaryKey );

    // 属性列を追加します。
     long ts = System . currentTim eMillis ();
     for ( int i = 0 ; i < 10 ; i ++) {
         for ( int j = 0 ; j < 3 ; j ++) {
             rowPutChan ge . addColumn ( new Column (" Col " + i
, ColumnValu e . fromLong ( j ), ts + j ));
         }
    }

     client . putRow ( new PutRowRequ est ( rowPutChan ge ));
}
```

例 3

この例では、行が存在しない場合、データが書き込まれます。

```
 private   static   void   putRow ( SyncClient   client ,  String
 pkValue ) {
    // プライマリキーを構築します。
     PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey  Builder
 . createPrim  aryKeyBuil  der ();
     primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE  Y_NAME
 ,  PrimaryKey  Value . fromString ( pkValue ));
     PrimaryKey   primaryKey  =  primaryKey  Builder . build ();

     RowPutChan  ge   rowPutChan  ge  =  new   RowPutChan  ge (
 TABLE_NAME ,  primaryKey );

    // 元の行が存在しないと予想します。
     rowPutChan  ge . setConditi  on ( new   Condition ( RowExisten
 ceExpectat  ion . EXPECT_NOT  _EXIST ));

    // 属性列を追加します。
     long   ts  =  System . currentTim  eMillis ();
     for  ( int   i  =  0 ;  i  <  10 ;  i ++) {
         for  ( int   j  =  0 ;  j  <  3 ;  j ++) {
             rowPutChan  ge . addColumn ( new   Column (" Col " +  i
 ,  ColumnValu  e . fromLong ( j ),  ts  +  j ));
         }
    }

     client . putRow ( new   PutRowRequ  est ( rowPutChan  ge ));
}
```

例 4

この例では、元の行は 存在し、Col0 値が 100 より大きい場合にデータを書き込むと予想します。

```
 private   static   void   putRow ( SyncClient   client ,  String
 pkValue ) {
    // プライマリキーを構築します。
     PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey  Builder
 . createPrim  aryKeyBuil  der ();
     primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE  Y_NAME
 ,  PrimaryKey  Value . fromString ( pkValue ));
     PrimaryKey   primaryKey  =  primaryKey  Builder . build ();

     RowPutChan  ge   rowPutChan  ge  =  new   RowPutChan  ge (
 TABLE_NAME ,  primaryKey );

    // 元の行が存在し、 Col0 値が 100 より大きい場合にデータを書き込むと予想します。
     Condition   condition  =  new   Condition ( RowExisten
 ceExpectat  ion . EXPECT_EXI  ST );
     condition . setColumnC  ondition ( new   SingleColu  mnValueCon
 dition (" Col0 ",
             SingleColu  mnValueCon  dition . CompareOpe  rator .
 GREATER_TH  AN ,  ColumnValu  e . fromLong ( 100 )));
     rowPutChan  ge . setConditi  on ( condition );

    // 属性列を追加します。
     long   ts  =  System . currentTim  eMillis ();
     for  ( int   i  =  0 ;  i  <  10 ;  i ++) {
```

```
        for ( int j = 0 ; j < 3 ; j ++) {
            rowPutChan ge . addColumn ( new Column (" Col " + i
, ColumnValu e . fromLong ( j ), ts + j ));
        }
    }

     client . putRow ( new PutRowRequ est ( rowPutChan ge ));
}
```

GetRow

単一行の GetRow API は、単一行のデータの読み取りに使用されます。 以下のパラメーターがあります。

- PrimaryKey: 読み込む行のプライマリキーです。 これは必須パラメーターです。
- ColumnsToGet: 読み込む列のセットです。 設定されていない場合は、すべての列が読み取られます。
- MaxVersions: 読み込むバージョンの最大数です。 MaxVersions パラメーターと TimeRange パラメーターの少なくとも一方を設定する必要があります。
- TimeRange: 読み取るバージョン番号の範囲です。 MaxVersions パラメーターと TimeRange パラメーターの少なくとも一方を設定する必要があります。
- Filter: 適用されたフィルター。 サーバーは、読み取り結果を再度フィルター処理するためにフィルターを使用します。

例 1

この例では、最新バージョンの単一行が読み取られ、特定の ColumnsToGet が設定されています。

```
        private static void getRow ( SyncClient client ,
String pkValue ) {
            // プライマリキーを構築します。
            PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
            primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
            PrimaryKey primaryKey = primaryKey Builder . build
();

            // 1 行読み込みます。
            SingleRowQ ueryCriter ia criteria = new
SingleRowQ ueryCriter ia ( TABLE_NAME , primaryKey );
            // 読み込む最新バージョンを設定します。
            criteria . setMaxVers ions ( 1 );
            GetRowResp onse getRowResp onse = client . getRow (
new GetRowRequ est ( criteria ));
            Row row = getRowResp onse . getRow ();

            System . out . println (" Read complete , result :");
            System . out . println ( row );

            // 読み込み列を設定します。
```

```
            criteria . addColumns  ToGet (" Col0 ");
            getRowResp  onse  =  client . getRow ( new   GetRowRequ
 est ( criteria ));
            row  =  getRowResp  onse . getRow ();

            System . out . println (" Read   complete ,  result :");
            System . out . println ();
        }
```

例 2

この例では、フィルターが設定されています。

```
        private   static   void   getRow ( SyncClient   client ,
 String   pkValue ) {
            // プライマリキーを構築します。
            PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
 Builder . createPrim  aryKeyBuil  der ();
            primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
 Y_NAME ,  PrimaryKey  Value . fromString ( pkValue ));
            PrimaryKey   primaryKey  =  primaryKey  Builder . build
 ();

            //  1  行読み込みます。
            SingleRowQ  ueryCriter  ia   criteria  =  new
 SingleRowQ  ueryCriter  ia ( TABLE_NAME ,  primaryKey );
            // 読み込む最新バージョンを設定します。
            criteria . setMaxVers  ions ( 1 );

            //フィルターを設定します。  Col0  値が  0  の場合、この行が返されま
す。
            SingleColu  mnValueFil  ter   singleColu  mnValueFil  ter
  =  new   SingleColu  mnValueFil  ter (" Col0 ",
                    SingleColu  mnValueFil  ter . CompareOpe  rator .
 EQUAL ,  ColumnValu  e . fromLong ( 0 ));
            //列  Col0  が存在しない場合、データは返されません。
            singleColu  mnValueFil  ter . setPassIfM  issing ( false
 );
            criteria . setFilter ( singleColu  mnValueFil  ter );

            GetRowResp  onse   getRowResp  onse  =  client . getRow (
 new   GetRowRequ  est ( criteria ));
            Row   row  =  getRowResp  onse . getRow ();

            System . out . println (" Read   complete ,  result :");
            System . out . println ( row );
        }
```

UpdateRow

UpdateRow API は、単一行のデータを更新するために使用されます。 行が存在しない場合は、新しい行が追加されます。

更新操作には、列の書き込み、列の削除および列バージョンの削除操作が含まれます。

UpdateRow を実行するときには、条件付き更新機能を使用して、更新する行の存在または既存の行の特定の列の値に関する条件を設定できます。 詳細については、「条件付き更新」をご参照ください。

例 1

この例では、複数の列が更新され、指定された列の指定されたバージョンと指定された列が削除されます。

```
        private   static   void   updateRow ( SyncClient   client ,
String   pkValue ) {
           // プライマリキーを構築します。
            PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
            primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString ( pkValue ));
            PrimaryKey   primaryKey  =  primaryKey  Builder . build
();

            RowUpdateC  hange   rowUpdateC  hange  =  new
RowUpdateC  hange ( TABLE_NAME ,  primaryKey );

           // 列を更新します。
            for  ( int   i  =  0 ;  i  <  10 ;  i ++) {
                rowUpdateC  hange . put ( new   Column (" Col " +  i
,  ColumnValu  e . fromLong ( i )));
           }

           // 指定された列の指定されたバージョンを削除します。
            rowUpdateC  hange . deleteColu  mn (" Col10 ",
1465373223  000L );

           // 指定した列を削除します。
            rowUpdateC  hange . deleteColu  mns (" Col11 ");

            client . updateRow ( new   UpdateRowR  equest (
rowUpdateC  hange ));
       }
```

例 2

この例では、更新条件が設定されています。

```
        private   static   void   updateRow ( SyncClient   client ,
String   pkValue ) {
           // プライマリキーを構築します。
            PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
            primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString ( pkValue ));
            PrimaryKey   primaryKey  =  primaryKey  Builder . build
();

            RowUpdateC  hange   rowUpdateC  hange  =  new
RowUpdateC  hange ( TABLE_NAME ,  primaryKey );
```

```
            // 元の行が存在し、 Col0 の値が 100 より大きい場合はデータが更
新されることを期待します。
            Condition condition = new Condition ( RowExisten
ceExpectat ion . EXPECT_EXI ST );
            condition . setColumnC ondition ( new SingleColu
mnValueCon dition (" Col0 ",
                    SingleColu mnValueCon dition . CompareOpe
rator . GREATER_TH AN , ColumnValu e . fromLong ( 100 )));
            rowUpdateC hange . setConditi on ( condition );

            // 列を更新します。
            for ( int i = 0 ; i < 10 ; i ++) {
                rowUpdateC hange . put ( new Column (" Col " + i
, ColumnValu e . fromLong ( i )));
            }

            // 指定された列の指定されたバージョンを削除します。
            rowUpdateC hange . deleteColu mn (" Col10 ",
1465373223 000L );

            // 指定した列を削除します。
            rowUpdateC hange . deleteColu mns (" Col11 ");

            client . updateRow ( new UpdateRowR equest (
rowUpdateC hange ));
        }
```

DeleteRow

DeleteRow API は、単一行を削除するために使用されます。

DeleteRow を実行するときは、条件付き更新機能を使用して、削除する行の存在、または既存の行の特定の列の値に関する条件を設定できます。 詳しくは、 「条件付き更新」をご参照ください。

例 1

この例では、行が削除されます。

```
        private static void deleteRow ( SyncClient client ,
String pkValue ) {
            // プライマリキーを構築します。
            PrimaryKey Builder primaryKey Builder = PrimaryKey
Builder . createPrim aryKeyBuil der ();
            primaryKey Builder . addPrimary KeyColumn ( PRIMARY_KE
Y_NAME , PrimaryKey Value . fromString ( pkValue ));
            PrimaryKey primaryKey = primaryKey Builder . build
();

            RowDeleteC hange rowDeleteC hange = new
RowDeleteC hange ( TABLE_NAME , primaryKey );

            client . deleteRow ( new DeleteRowR equest (
rowDeleteC hange ));
        }
```

例 2

この例では、削除条件が設定されています。

```
        private   static   void   deleteRow ( SyncClient   client ,
String   pkValue ) {
           // プライマリキーを構築します。
            PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
            primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString ( pkValue ));
            PrimaryKey   primaryKey  =  primaryKey  Builder . build
();

            RowPutChan  ge   rowPutChan  ge  =  new   RowPutChan  ge
( TABLE_NAME ,  primaryKey );

           // 元の行が存在し、 Col0  の値が  100  より大きい場合はデータが削
除されることを期待します。
            Condition   condition  =  new   Condition ( RowExisten
ceExpectat  ion . EXPECT_EXI  ST );
            condition . setColumnC  ondition ( new   SingleColu
mnValueCon  dition (" Col0 ",
                    SingleColu  mnValueCon  dition . CompareOpe
rator . GREATER_TH  AN ,  ColumnValu  e . fromLong ( 100 )));
            rowDeleteC  hange . setConditi  on ( condition );

            client . deleteRow ( new   DeleteRowR  equest (
rowDeleteC  hange ));
       }
```

3.5 条件付き更新

条件付き更新機能は、指定された条件が満たされた場合にのみテーブルデータを更新します。 条件が満たされない場合、更新は失敗します。 条件付き更新は、PutRow、UpdateRow、DeleteRow および BatchWriteRow に適用できます。

判定条件には、行の存在条件と列ベースの条件があります。

- 行存在条件: この条件は IGNORE、EXPECT_EXIST、EXPECT_NOT_EXIST の3種類に分けられます。 テーブルを更新する必要がある場合、システムは最初に行の存在状態を確認します。 行の存在条件が満たされていないと、更新は失敗し、システムはエラーをスローします。
- 列ベースの条件: この条件は現在 SingleColumnValueCondition と CompositeColumnValueCondition をサポートしています。 これらは、Table Store フィルタで使用される条件と同様に、1 つ以上の列の値に基づいて条件ベースの判断を下すために使用されます。

条件付き更新を使用して楽観的ロックを実装できます。 行を更新する必要がある場合、システムは最初に列の値を取得します。 たとえば、列 A の値が 1 であるとします。 `" Column  A  =  1 "` という条件を設定し、行を更新し、 `" Column  A  =  2 "` 設定します。 更新が失敗した場合は、その行が別のクライアントによって更新されたことを意味します。

例 1

SingleColumnValueCondition を構築します。

```
        //  Col0  ==  0  に条件を設定します。
         SingleColu  mnValueCon  dition   singleColu  mnValueCon
 dition  =  new   SingleColu  mnValueCon  dition (" Col0 ",
                 SingleColu  mnValueCon  dition . CompareOpe  rator .
 EQUAL ,  ColumnValu  e . fromLong ( 0 ));
        // 列  Col0  が存在しない場合、条件は満たされません。
         singleColu  mnValueCon  dition . setPassIfM  issing ( false
 );
        // 最新バージョンを判定するだけです。
         singleColu  mnValueCon  dition . setLatestV  ersionsOnl  y (
 true );
```

例 2

CompositeColumnValueCondition を構築します。

```
// 条件  composite1  は ( Col0  ==  0 )  AND  ( Col1 >  100 ) です。
         CompositeC  olumnValue  Condition   composite1  =  new
 CompositeC  olumnValue  Condition ( CompositeC  olumnValue  Condition
 . LogicOpera  tor . AND );
         SingleColu  mnValueCon  dition   single1  =  new
 SingleColu  mnValueCon  dition (" Col0 ",
                 SingleColu  mnValueCon  dition . CompareOpe  rator .
 EQUAL ,  ColumnValu  e . fromLong ( 0 ));
         SingleColu  mnValueCon  dition   single2  =  new
 SingleColu  mnValueCon  dition (" Col1 ",
                 SingleColu  mnValueCon  dition . CompareOpe  rator .
 GREATER_TH  AN ,  ColumnValu  e . fromLong ( 100 ));
         composite1 . addConditi  on ( single1 );
         composite1 . addConditi  on ( single2 );
        // 条件  composite2  は、( ( Col0  ==  0 )  AND  ( Col1  >  100
 ) )  OR  ( Col2  <=  10 ) です。
         CompositeC  olumnValue  Condition   composite2  =  new
 CompositeC  olumnValue  Condition ( CompositeC  olumnValue  Condition
 . LogicOpera  tor . OR );
         SingleColu  mnValueCon  dition   single3  =  new
 SingleColu  mnValueCon  dition (" Col2 ",
                 SingleColu  mnValueCon  dition . CompareOpe  rator .
 LESS_EQUAL ,  ColumnValu  e . fromLong ( 10 ));
         composite2 . addConditi  on ( composite1 );
         composite2 . addConditi  on ( single3 );
```

例 3

楽観的ロック機能を実装するために、Condition を使用できます。 次の例は、Condition を使用して列を増分する方法を示しています。

```
         private   static   void   updateRowW  ithConditi  on (
 SyncClient   client ,  String   pkValue ) {
            // プライマリキーを構築します。
             PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
 Builder . createPrim  aryKeyBuil  der ();
             primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
 Y_NAME ,  PrimaryKey  Value . fromString ( pkValue ));
```

```
            PrimaryKey  primaryKey  =  primaryKey  Builder . build
();

           //  1 行読み込みます。
            SingleRowQ  ueryCriter  ia  criteria  =  new
SingleRowQ  ueryCriter  ia ( TABLE_NAME ,  primaryKey );
            criteria . setMaxVers  ions ( 1 );
            GetRowResp  onse  getRowResp  onse  =  client . getRow (
new  GetRowRequ  est ( criteria ));
            Row  row  =  getRowResp  onse . getRow ();
            long  col0Value  =  row . getLatestC  olumn (" Col0 ").
getValue (). asLong ();

           //  Col0  の条件付き更新を設定し、列値 +  1  を使用します。
            RowUpdateC  hange  rowUpdateC  hange  =  new
RowUpdateC  hange ( TABLE_NAME ,  primaryKey );
            Condition  condition  =  new  Condition ( RowExisten
ceExpectat  ion . EXPECT_EXI  ST );
            ColumnCond  ition  columnCond  ition  =  new
SingleColu  mnValueCon  dition (" Col0 ",  SingleColu  mnValueCon
dition . CompareOpe  rator . EQUAL ,  ColumnValu  e . fromLong (
col0Value ));
            condition . setColumnC  ondition ( columnCond  ition );
            rowUpdateC  hange . setConditi  on ( condition );
            rowUpdateC  hange . put ( new  Column (" Col0 ",
ColumnValu  e . fromLong ( col0Value  +  1 )));

            try  {
                client . updateRow ( new  UpdateRowR  equest (
rowUpdateC  hange ));
           }  catch  ( TableStore  Exception  ex ) {
                System . out . println ( ex . toString ());
           }
       }
```

3.6 フィルター

Table Store フィルターは、サーバ側で結果をフィルタリングするために使用されるため、サーバーはフィルター条件に一致する行または列のみを返します。 フィルターは、GetRow、BatchGetRow および GetRange API と共に使用できます。

現在、Table Store は SingleColumnValueFilter と CompositeColumnValueFilter の 2 つのフィルターをサポートしています。これらは、参照列の値に基づいて行のデータをフィルタリングするかどうかを決定するために使用されます。 SingleColumnValueFilter は単一の参照列の値のみをチェックしますが、CompositeColumnValueFilter は複数の参照列の値をチェックし、チェック結果を論理的に組み合わせて行データをフィルタリングするかどうかを決定します。

フィルターは、読み取られたデータをフィルタリングするために使用されます。そのため、SingleColumnValueFilter または CompositeColumnValueFilter によって使用される参照列を読み取り、データに含める必要があります。 データの読み取り元の列を指定しても、これらの

列に参照列が含まれていない場合、フィルターは参照列の値を取得できません。 参照列が存在しない場合、SingleColumnValueFilter は passIfMissing パラメーターを使用してフィルター条件が満たされているかどうかを判断します。 つまり、参照列が存在しない場合でもアクションを選択できます。

例 1

SingleColumnValueFilter を構築します。

```
        // フィルターを設定します。 Col0 値が 0 の場合、この行が返されます。
         SingleColu mnValueFil ter singleColu mnValueFil ter =
new SingleColu mnValueFil ter (" Col0 ",
                SingleColu mnValueFil ter . CompareOpe rator .
EQUAL , ColumnValu e . fromLong ( 0 ));
        // 列 Col0 が存在しない場合、データは返されません。
         singleColu mnValueFil ter . setPassIfM issing ( false );
        // 最新バージョンを判断するだけです。
         singleColu mnValueFil ter . setLatestV ersionsOnl y (
true );
```

例 2

CompositeColumnValueFilter を構築します。

```
        // 条件 composite1 は ( Col0 == 0 ) AND ( Col1 > 100
) です。
        CompositeC olumnValue Filter composite1 = new
CompositeC olumnValue Filter ( CompositeC olumnValue Filter .
LogicOpera tor . AND );
        SingleColu mnValueFil ter single1 = new SingleColu
mnValueFil ter (" Col0 ",
                SingleColu mnValueFil ter . CompareOpe rator .
EQUAL , ColumnValu e . fromLong ( 0 ));
        SingleColu mnValueFil ter single2 = new SingleColu
mnValueFil ter (" Col1 ",
                SingleColu mnValueFil ter . CompareOpe rator .
GREATER_TH AN , ColumnValu e . fromLong ( 100 ));
        composite1 . addFilter ( single1 );
        composite1 . addFilter ( single2 );

        // 条件 composite2 は、 ( ( Col0 == 0 ) AND ( Col1 >
100 ) ) OR ( Col2 <= 10 ) です。
        CompositeC olumnValue Filter composite2 = new
CompositeC olumnValue Filter ( CompositeC olumnValue Filter .
LogicOpera tor . OR );
        SingleColu mnValueFil ter single3 = new SingleColu
mnValueFil ter (" Col2 ",
                SingleColu mnValueFil ter . CompareOpe rator .
LESS_EQUAL , ColumnValu e . fromLong ( 10 ));
        composite2 . addFilter ( composite1 );
        composite2 . addFilter ( single3 );
```

3.7 プライマリキー列の自動インクリメント

プライマリキー列の自動インクリメント機能は、Table Store によって開始された新しい機能であり、Java SDK v4.2.0 以降のバージョンで使用可能です。

プライマリキー列の自動インクリメント機能は、プライマリキー列を自動インクリメント列として指定した場合、データを書き込むときに Table Store がこの列に新しい値を自動的に生成し、生成された値が同じパーティションキー下の列の最大値になることを意味します。この機能は主に、e コマース Web サイトのアイテム ID、大規模 Web サイトのユーザー ID、フォーラムの投稿 ID、チャットツールのメッセージ IDなど、プライマリキー列に自動インクリメント機能を適用する必要があるシステム設計シナリオに適用されます。

機能の特徴

- Table Store は現在複数のプライマリキーをサポートしていますが、最初のプライマリキーはパーティションキーです。これに対して、プライマリキーの自動増分は許可されていません。
- パーティションキーを除いて、他のどのプライマリキーも自動増分列に設定できます。
- 自動インクリメントはパーティションキーに基づいて生成されるため、プライマリキー列の自動インクリメント機能はパーティションキーレベルです。
- 現時点では、各テーブルの自動インクリメント列に設定できるプライマリキー列は 1 つだけです。
- 自動生成された自動インクリメント列は、64 ビット符号付き長整数型です。

インターフェイス

プライマリキー列の自動インクリメント機能には、主に 2 種類のインターフェースが含まれます。テーブルの作成とデータの書き込みです。

- テーブルを作成する

  テーブルを作成するときは、自動インクリメントのプライマリキーの属性を PrimaryKeyOption.AUTO_INCREMENT に設定するだけです。

  関連インターフェイス: CreateTable

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

       // 最初の列はパーティションキーです。
        tableMeta . addPrimary  KeyColumn ( new   PrimaryKey
Schema (" PK_1 ",  PrimaryKey  Type . STRING ));

       //  2 番目の列は  INTEGER 型 の自動インクリメント列で、属性は
AUTO_INCRE  MENT  です。
```

```
        tableMeta . addPrimary  KeyColumn ( new   PrimaryKey
Schema (" PK_2 ",  PrimaryKey  Type . INTEGER ,  PrimaryKey  Option
. AUTO_INCRE  MENT ));

        int   timeToLive  = - 1 ;  // 期限切れになりません。
        int   maxVersion  s  =  1 ;  // 保存されるバージョンは  1  つだ
けです。

        TableOptio  ns   tableOptio  ns  =  new   TableOptio  ns (
timeToLive ,  maxVersion  s );

        CreateTabl  eRequest   request  =  new   CreateTabl
eRequest ( tableMeta ,  tableOptio  ns );

        client . createTabl  e ( request );
    }
```

注：

- **最初のプライマリキーはパーティションキーであり、自動インクリメント列として設定することはできません。**
- **自動増分列として設定できるのは INTEGER 列のみです。**
- **各テーブルに許可されるプライマリキー自動増分列は 1 つだけです。**

- **データを書き込み**

  データを書き込むときは、自動インクリメント列の値をプレースホルダ PrimaryKeyValue.AUTO_INCREMENT として設定するだけです。

  データが Table Store に書き込まれた後にプライマリキー値を自動的に生成する場合は、RT_PK に `ReturnType` を設定し、データが正常に書き込まれたときにプライマリキー値を返すようにできます。

  関連インターフェイス: PutRow / UpdateRow / BatchWriteRow

```
    private   static   void   putRow ( SyncClient   client ,
String   receive_id ) {
       // プライマリキーを作成します
        PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();

       // 最初の列の値は  md5  の最初の  4  桁です ( receive_id )
        primaryKey  Builder . addPrimary  KeyColumn (“ PK_1 ”,
PrimaryKey  Value . fromString (" Hangzhou ");


       //  3  列目は主キーの自動インクリメント列です。 この値は  Table
  Store  によって生成されます。 自動インクリメント列の値をプレースホルダ
AUTO_INCRE  MENT  として設定します。 ここに真の値を入力する必要はありません。
        primaryKey  Builder . addPrimary  KeyColumn (" PK_2 ",
PrimaryKey  Value . AUTO_INCRE  MENT );
        PrimaryKey   primaryKey  =  primaryKey  Builder . build ();
```

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

        // ここで戻り型は RT_PK に設定され、返された結果に PK 列の値が含
まれることを示します。 ReturnType が設定されていない場合、デフォルトでは結果
が返されません。
        rowPutChange.setReturnType(ReturnType.RT_PK);

        // 属性列とメッセージの内容を追加します。
        rowPutChange.addColumn(new Column("content",
ColumnValue.fromString(content)));

        // Table Store にデータを書き込みます。
        PutRowResponse response = client.putRow(new
PutRowRequest(rowPutChange));

        // 返された PK 列を印字します
        Row returnRow = response.getRow();
        if (returnRow != null) {
            System.out.println("PrimaryKey:" + returnRow.
getPrimaryKey().toString());
        }

        // 消費された CU を印字します。
        CapacityUnit cu = response.getConsumedCapacity
().getCapacityUnit();
        System.out.println("ReadCapacityUnit:" + cu.
getReadCapacityUnit());
        System.out.println("WriteCapacityUnit:" + cu.
getWriteCapacityUnit());
    }
```

3.8 エラー処理

方法

現在、Table Store Java SDK は `Exception` メソッドを採用して、エラー処理をしています。呼び出されたインターフェイスが例外をスローしない場合、操作は成功します。 例外がスローされると、操作は失敗します。

注：

BatchGetRow や BatchWriteRow などのバッチ操作インターフェイスは、例外がスローされず、各行のステータスが正常であることがシステムによって確認された場合にのみ正常に呼び出されます。

例外

Table Store Java SDK には、RuntimeException から継承された ClientException と TableStoreException の2種類の例外があります。

- ClientException: 不正なパラメーター値など、SDK 内部の例外を示します。

- TableStoreException: サーバーエラーメッセージの解析によって生成されたサーバーエラーを示します。 TableStoreException には以下のコンポーネントがあります。
  - getHttpStatus(): 返された HTTP コード、例えば 200 または 404。
  - getErrorCode(): Table Store から返されたエラータイプの文字列。
  - getRequestId(): このリクエストの一意の識別子として使用される UUID。 問題を解決できない場合は、RequestId を保存して チケットを起票し、サポートセンターへお問い合わせください。

3.9 インクリメンタルデータ操作

Table Store は、ストリーム用の ListStream 操作と DescribeStream 操作およびシャード用の GetShardIterator 操作と GetStreamRecord 操作を提供します。

ListStream

ListStream は、現在のインスタンスのテーブルで有効になっているすべてのストリームを一覧表示するために使用されます。

例

テーブルに対して有効になっているすべてのストリームに関する情報を一覧表示します。

```
private static void listStream ( SyncClient client , String
  tableName ) {
    ListStream Request listStream Request = new ListStream
Request ( tableName );
    ListStream Response result = client . listStream (
listStream Request );
}
```

DescribeStream

DescribeStream は、ストリームの creationTime、expirationTime、状態、シャード、および (シャードが返されない場合) 次の開始シャードの ID を照会するために使用されます。

例 1

現在のストリームのすべてのシャードに関する情報を入手します。

```
    private static void describeSt ream ( SyncClient client
,  String streamId ) {
        DescribeSt reamReques t desRequest = new DescribeSt
reamReques t ( streamId );
        DescribeSt reamRespon se desStream = client .
describeSt ream ( desRequest );
   }
```

```
```

例 2

InclusiveStartShardId と毎回返されるシャードの最大数を設定します。

```
    private   static   void   describeSt  ream ( SyncClient   client
,  String   streamId ) {
    DescribeSt  reamReques  t   dsRequest  =  new   DescribeSt
reamReques  t ( streamId );
    dsRequest . setInclusi  veStartSha  rdId ( startShard  Id );
    dsRequest . setShardLi  mit ( 10 );
    DescribeSt  reamRespon  se   dscStream  =  client . describeSt
ream ( dsRequest );
   }
```

GetShardIterator

`GetShardIterator` **は、シャードを読み取るための開始反復子を取得するために使用されます。**

例

シャードを読み取るための開始反復子を取得します。

```
    private   static   void   getShardIt  erator ( SyncClient
client ,  String   streamId ,  String   shardId ) {
    GetShardIt  eratorRequ  est   getShardIt  eratorRequ  est  =  new
  GetShardIt  eratorRequ  est ( streamId ,  shardId );
    GetShardIt  eratorResp  onse   shardItera  tor  =  client .
getShardIt  erator ( getShardIt  eratorRequ  est );
   }
```

GetStreamRecord

`GetStreamRecord` **は、シャードの各更新レコードを取得するために使用されます。**

例

シャードの最初の 100 個の更新レコードを入手します。

```
    private   static   void   getShardIt  erator ( SyncClient
client ,  String   shardItera  tor ) {
        GetStreamR  ecordReque  st   streamReco  rdRequest  =  new
GetStreamR  ecordReque  st ( shardItera  tor );
        streamReco  rdRequest . setLimit ( 100 );
        GetStreamR  ecordRespo  nse   streamReco  rdResponse  =
client . getStreamR  ecord ( streamReco  rdRequest );
        List < StreamReco  rd >  records  =  streamReco  rdResponse .
getRecords ();
        for ( int   k = 0 ; k < records . size (); k ++){
            System . out . println (" record   info :" +   records .
get ( k ). toString ());
       }
```

```
        System . out . println (" next   iterator :" +  streamReco
rdResponse . getNextSha  rdIterator ());
   }
```

3.10 複数行操作

Table Store SDK は、BatchGetRow、BatchWriteRow、GetRange、createRangeIterator およびその他の行操作 API を提供します。

BatchGetRow

BatchGetRow API は、単一のリクエストで複数の行からデータを読み取ることができます。パラメーターは GetRow API のものと同じです。 BatchGetRow がすべての行に同じパラメーター条件を使用するように注意します。 たとえば、ColumnsToGet = [colA] の場合、すべての行について colA 値 のみが読み取られます。

BatchGetRow API を使用する場合は、BatchWriteRow API と同様に、戻り値を確認する必要があります。 操作が一部の行で失敗した場合、システムは例外をスローしませんが、失敗した行の情報を BatchGetRowResponse に格納します。 BatchGetRowResponse#getFailedRows メソッドを使用して失敗した行に関する情報を取得するか、BatchGetRowResponse#AllSucceed を使用してすべての行が読み取られたかどうかを判断できます。

例

BatchGetRow を 10 行読み込むように設定します。 バージョン条件、読み込む列、およびフィルタを設定します。

```
        private   static   void   batchGetRo  w ( SyncClient   client
) {
            MultiRowQu  eryCriteri  a   multiRowQu  eryCriteri  a  =
new   MultiRowQu  eryCriteri  a ( TABLE_NAME );
           // 読み込む行を 10 行追加します。
            for  ( int   i  =  0 ;  i  <  10 ;  i ++) {
                PrimaryKey  Builder   primaryKey  Builder  =
PrimaryKey  Builder . createPrim  aryKeyBuil  der ();
                primaryKey  Builder . addPrimary  KeyColumn (
PRIMARY_KE  Y_NAME ,  PrimaryKey  Value . fromString (" pk " +  i ));
                PrimaryKey   primaryKey  =  primaryKey  Builder .
build ();
                multiRowQu  eryCriteri  a . addRow ( primaryKey );
            }
            // 条件を追加します。
            multiRowQu  eryCriteri  a . setMaxVers  ions ( 1 );
            multiRowQu  eryCriteri  a . addColumns  ToGet (" Col0 ");
            multiRowQu  eryCriteri  a . addColumns  ToGet (" Col1 ");
            SingleColu  mnValueFil  ter   singleColu  mnValueFil  ter
 =  new   SingleColu  mnValueFil  ter (" Col0 ",
                    SingleColu  mnValueFil  ter . CompareOpe  rator .
EQUAL ,  ColumnValu  e . fromLong ( 0 ));
            singleColu  mnValueFil  ter . setPassIfM  issing ( false
);
```

```
            multiRowQu  eryCriteri  a . setFilter ( singleColu
mnValueFil  ter );

            BatchGetRo  wRequest   batchGetRo  wRequest  =  new
BatchGetRo  wRequest ();
          //  BatchGetRo  w  を使用すると、複数のテーブルからデータを読み取
ることができます。  A  single  multiRowQu  eryCriteri  a  correspond
s  to  a  query  condition  for  one  table .  You  can  add
  multiRowQu  eryCriteri  a .
            batchGetRo  wRequest . addMultiRo  wQueryCrit  eria (
multiRowQu  eryCriteri  a );

            BatchGetRo  wResponse  batchGetRo  wResponse  =  client
. batchGetRo  w ( batchGetRo  wRequest );

            System . out . println (" Were  all  successful :" +
batchGetRo  wResponse . isAllSucce  ed ());
            if  (!  batchGetRo  wResponse . isAllSucce  ed ()) {
                for  ( BatchGetRo  wResponse . RowResult  rowResult
:  batchGetRo  wResponse . getFailedR  ows ()) {
                    System . out . println (" Failed  rows :" +
batchGetRo  wRequest . getPrimary  Key ( rowResult . getTableNa  me
(),  rowResult . getIndex ()));
                    System . out . println (" Cause  of  failure :"
+  rowResult . getError ());
                }

                /**
                 * 失敗した行を再試行するための別のリクエストを作成するには、
createRequ  estForRetr  y  メソッドを使用できます。 ここでは、再試行リクエスト
の一部のみを構成します。
                 * リトライメソッドには、 SDK  のカスタムリトライポリシー機能を
使用することを推奨します。 この機能を使用すると、バッチ操作後に失敗した行を再試行で
きます。 再試行ポリシーを設定した後は、呼び出し側の  API  に再試行コードを追加する
必要はありません。
                 */
                BatchGetRo  wRequest  retryReque  st  =  batchGetRo
wRequest . createRequ  estForRetr  y ( batchGetRo  wResponse .
getFailedR  ows ());
            }
        }
```

BatchWriteRow

BatchWriteRow API を使用すると、単一の要求で複数の書き込み操作を実行できます。 これらの書き込み操作には、PutRow、UpdateRow および DeleteRowがあります。 この API では、一度に複数のテーブルに書き込むこともできます。

単一の操作を構築するプロセスは、PutRow、UpdateRow または DeleteRow APIを使用するのと同じです。 更新条件を設定することもできます。

BatchWriteRow API を呼び出すときは、必ず戻り値を確認してください。バッチ書き込み中に、他の行が失敗しても、いくつかの行が書き込まれることがあります。 失敗した行インデックスとエラーメッセージは、返された BatchWriteRowResponse に挿入されます。 一部の行が失敗した場合、システムは例外をスローしません。 BatchWriteRowResponse の isAllSucceed

メソッドを使用して、すべての行が正常に書き込まれたかどうかを判断できます。 チェックしない場合、失敗した操作の中には無視されるものがあります。 状況によっては、BatchWriteRow API が例外をスローすることがあります。 たとえば、サーバーが、一部の操作に誤ったパラメーターがあることを検出した場合、システムはパラメーターエラー例外をスローすることがあります。 例外がスローされた場合、これは要求内の操作がどれも完了していないことを意味します。

例

ここでは、単一の BatchWriteRow 要求が送信されます。 2 つの PutRow オペレーション、1 つの UpdateRow オペレーション、および 1 つの DeleteRow オペレーションが含まれています。

```
        private   static   void   batchWrite  Row ( SyncClient
client ) {
            BatchWrite  RowRequest   batchWrite  RowRequest  =  new
BatchWrite  RowRequest ();

           //  rowPutChan  ge1  を作成します。
            PrimaryKey  Builder   pk1Builder  =  PrimaryKey  Builder
. createPrim  aryKeyBuil  der ();
            pk1Builder . addPrimary  KeyColumn ( PRIMARY_KE  Y_NAME
,  PrimaryKey  Value . fromString (" pk1 "));
            RowPutChan  ge   rowPutChan  ge1  =  new   RowPutChan  ge
( TABLE_NAME ,  pk1Builder . build ());
           // 複数の列を追加します。
            for  ( int   i  =  0 ;  i  <  10 ;  i ++) {
                rowPutChan  ge1 . addColumn ( new   Column (" Col "
+  i ,  ColumnValu  e . fromLong ( i )));
            }
            // バッチ操作に追加します。
            batchWrite  RowRequest . addRowChan  ge ( rowPutChan  ge1
);

           //  rowPutChan  ge2  を作成します。
            PrimaryKey  Builder   pk2Builder  =  PrimaryKey  Builder
. createPrim  aryKeyBuil  der ();
            pk2Builder . addPrimary  KeyColumn ( PRIMARY_KE  Y_NAME
,  PrimaryKey  Value . fromString (" pk2 "));
            RowPutChan  ge   rowPutChan  ge2  =  new   RowPutChan  ge
( TABLE_NAME ,  pk2Builder . build ());
           // 複数の列を追加します。
            for  ( int   i  =  0 ;  i  <  10 ;  i ++) {
                rowPutChan  ge2 . addColumn ( new   Column (" Col "
+  i ,  ColumnValu  e . fromLong ( i )));
            }
            // バッチ操作に追加します。
            batchWrite  RowRequest . addRowChan  ge ( rowPutChan  ge2
);

           //  rowUpdateC  hange  を作成します。
            PrimaryKey  Builder   pk3Builder  =  PrimaryKey  Builder
. createPrim  aryKeyBuil  der ();
            pk3Builder . addPrimary  KeyColumn ( PRIMARY_KE  Y_NAME
,  PrimaryKey  Value . fromString (" pk3 "));
            RowUpdateC  hange   rowUpdateC  hange  =  new
RowUpdateC  hange ( TABLE_NAME ,  pk3Builder . build ());
           // 複数の列を追加します。
```

```
            for ( int i = 0 ; i < 10 ; i ++) {
                rowUpdateC hange . put ( new Column (" Col " + i
, ColumnValu e . fromLong ( i )));
            }
            // 列を削除します。
            rowUpdateC hange . deleteColu mns (" Col10 ");
            // バッチ操作に追加します。
            batchWrite RowRequest . addRowChan ge ( rowUpdateC
hange );

            // rowDeleteC hange を作成します。
            PrimaryKey Builder pk4Builder = PrimaryKey Builder
. createPrim aryKeyBuil der ();
            pk4Builder . addPrimary KeyColumn ( PRIMARY_KE Y_NAME
, PrimaryKey Value . fromString (" pk4 "));
            RowDeleteC hange rowDeleteC hange = new
RowDeleteC hange ( TABLE_NAME , pk4Builder . build ());
            // バッチ操作に追加します。
            batchWrite RowRequest . addRowChan ge ( rowDeleteC
hange );

            BatchWrite RowRespons e response = client .
batchWrite Row ( batchWrite RowRequest );

            System . out . println (" Were all successful :" +
response . isAllSucce ed ());
            if (! response . isAllSucce ed ()) {
                for ( BatchWrite RowRespons e . RowResult
rowResult : response . getFailedR ows ()) {
                    System . out . println (" Failed rows :" +
batchWrite RowRequest . getRowChan ge ( rowResult . getTableNa me
(), rowResult . getIndex ()). getPrimary Key ());
                    System . out . println (" Cause of failure :"
+ rowResult . getError ());
                }
                /**
                 * 失敗した行を再試行するための別のリクエストを作成するには、
createRequ estForRetr y メソッドを使用できます。 ここでは、再試行リクエスト
の一部のみを構成します。
                 * リトライメソッドは、 SDK のカスタムリトライポリシー機能を使
用することを推奨します。 この機能を使用すると、バッチ操作後に失敗した行を再試行でき
ます。 再試行ポリシーを設定した後は、呼び出し側の API に再試行コードを追加する必
要はありません。
                 */
                BatchWrite RowRequest retryReque st =
batchWrite RowRequest . createRequ estForRetr y ( response .
getFailedR ows ());
            }
        }
```

GetRange

GetRange API は、特定の範囲のデータを読み取るために使用されます。 Tabe Store テーブルでは、すべての行はそれらの主キーによってソートされ、プライマリキーはすべてのプライマリキー列によって順番に合成されます。 したがって、行がプライマリキーの特定の列によってソートされると仮定しないでください。

GetRange API を使用すると、特定の範囲に従ってデータを前方または後方に読み取ることができます。 読み取る行数を制限することもできます。 範囲が大きく、スキャンされた行数またはデータ量が制限を超えると、スキャンは停止し、読み取り行と次のプライマリキーが返されます。 すべての行が読み取られない場合は、最後の操作が中断された時点から開始する要求を開始し、前の操作によって戻された次のプライマリキーに基づいて残りの行を読み取ることができます。

GetRange リクエストには、次の主なパラメーターがあります。

- Direction: 列挙型で、値 FORWARD または BACKWARD を含みます。
- InclusiveStartPrimaryKey: 開始プライマリキー (包括的) 方向が逆方向に設定されている場合、開始プライマリキーは終了プライマリキーより大きくなければなりません。
- ExclusiveEndPrimaryKey: 終了プライマリキー（排他的）。 方向が逆方向に設定されている場合、開始プライマリキーは終了プライマリキーより大きくなければなりません。
- Limit: このリクエストの最大行数。
- ColumnsToGet: 読み込む列のセット。 これが設定されていない場合は、すべての列が読み取られます。
- MaxVersions: 読み込むバージョンの最大数。 MaxVersions パラメーターと TimeRange パラメーターの少なくとも一方を設定する必要があります。
- TimeRange: 読み取るバージョン番号の範囲 MaxVersions パラメーターと TimeRange パラメーターの少なくとも一方を設定する必要があります。
- Filter: 適用されたフィルター。 サーバーは、読み取り結果を再度フィルター処理するためにフィルターを使用します。

例

次の操作では、順方向に読み込み、NextStartPrimaryKey が null 値かどうかを判断し、範囲内のすべてのデータを読み込みます。

```
       private   static   void   getRange ( SyncClient   client ,
String   startPkVal  ue ,  String   endPkValue ) {
           RangeRowQu  eryCriteri  a   rangeRowQu  eryCriteri  a  =
new   RangeRowQu  eryCriteri  a ( TABLE_NAME );

          //  StartPrima  ryKey  を設定します。
           PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
           primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString ( startPkVal  ue ));
           rangeRowQu  eryCriteri  a . setInclusi  veStartPri
maryKey ( primaryKey  Builder . build ());

          //  EndPrimary  Key  を設定します。
           primaryKey  Builder  =  PrimaryKey  Builder . createPrim
aryKeyBuil  der ();
```

```
            primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString ( endPkValue ));
            rangeRowQu  eryCriteri  a . setExclusi  veEndPrima  ryKey
( primaryKey  Builder . build ());

            rangeRowQu  eryCriteri  a . setMaxVers  ions ( 1 );

            System . out . println (" GetRange   result :");
            while  ( true ) {
                GetRangeRe  sponse   getRangeRe  sponse  =  client .
getRange ( new   GetRangeRe  quest ( rangeRowQu  eryCriteri  a ));
                for  ( Row   row  :  getRangeRe  sponse . getRows ())
{
                    System . out . println ( row );
                }

                //  nextStartP  rimaryKey  が  null  値でない場合は、読み
続けます。
                 if  ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey
() ! =  null ) {
                    rangeRowQu  eryCriteri  a . setInclusi
veStartPri  maryKey ( getRangeRe  sponse . getNextSta  rtPrimaryK  ey
());
                }  else  {
                    break ;
                }
            }
        }
```

createRangeIterator

次の操作はデータを繰り返し読み込みます。

例

```
        private   static   void   getRangeBy  Iterator ( SyncClient
client ,  String   startPkVal  ue ,  String   endPkValue ) {
            RangeItera  torParamet  er   rangeItera  torParamet  er
=  new   RangeItera  torParamet  er ( TABLE_NAME );

           //  StartPrima  ryKey  を設定します。
            PrimaryKey  Builder   primaryKey  Builder  =  PrimaryKey
Builder . createPrim  aryKeyBuil  der ();
            primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString ( startPkVal  ue ));
            rangeItera  torParamet  er . setInclusi  veStartPri
maryKey ( primaryKey  Builder . build ());

           //  EndPrimary  Key  を設定します。
            primaryKey  Builder  =  PrimaryKey  Builder . createPrim
aryKeyBuil  der ();
            primaryKey  Builder . addPrimary  KeyColumn ( PRIMARY_KE
Y_NAME ,  PrimaryKey  Value . fromString ( endPkValue ));
            rangeItera  torParamet  er . setExclusi  veEndPrima
ryKey ( primaryKey  Builder . build ());

            rangeItera  torParamet  er . setMaxVers  ions ( 1 );

            Iterator < Row >  iterator  =  client . createRang
eIterator ( rangeItera  torParamet  er );
```

```
            System . out . println (" Use   Iterator   to   implement
GetRange   result :");
            while  ( iterator . hasNext ()) {
                Row   row  =  iterator . next ();
                System . out . println ( row );
            }
        }
```

4 Go SDK

4.1 はじめに

このドキュメントでは、Go SDK v4.0.0 以降を Table Store にインストールして使用する方法について説明します。 Alibaba Cloud Table Store を有効化し、AccessKeyId と AccessKeySecret を作成したと仮定します。

- Alibaba Cloud Table Store をまだ有効化していない場合、またはTable Store についてもっと知りたい場合は、 「*Table Store ホームページ*」にアクセスしてください。
- AccessKeyId と AccessKeySecret を作成していない場合は、 *Alibaba Cloud AccessKey コンソール*にログインし、 AccessKey を作成します。

SDK のダウンロード

- *SDK パッケージ*は、パッケージ、ソースコード、サンプルを含みます。
- SDK パッケージのインストール方法の詳細については、「インストール方法」をご参照ください。

バージョン

最新バージョン: 4.1.0

4.2 インストール方法

前提条件

Go SDK v1.4 以降に適用されます。

手順

次のコマンドを実行します。

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

4.3 初期化

TableStoreClient は Table Store のクライアントです。 呼び出し側に、テーブルを操作し、単一行または複数の行に対してデータを読み書きするための一連の方法を提供します。

エンドポイントの決定

エンドポイントは、リージョン内の Alibaba Cloud Table Store のドメイン名アドレスです。次の形式をサポートしています。

エンドポイントの種類	説明
リージョンアドレス	現在の Table Store インスタンスのリージョン。たとえば、`https://instance.cn-hangzhou.ots.aliyuncs.com` となります。

Table Store のリージョンアドレス

Table Store インスタンスが存在するエンドポイントを照会するには、次の手順に従います。

1. *Table Store コンソール*にログインします。
2. インスタンスの詳細ページにアクセスして、インスタンスのエンドポイントであるインスタンスアクセスアドレスを検索します。

AccessKey の設定

Alibaba Cloud Table Store にアクセスするには、署名認証用の有効な AccessKey (AccessKeyId と AccessKeySecret を含む) が必要です。 AccessKey を取得するには、次の手順に従います。

1. *Alibaba Cloud アカウント*を登録します。
2. *AccessKey コンソール*にログインします。AccessKeyId と AccessKeySecret を作成します。

AccessKeyId と AccessKeySecret の取得後、次の手順に従って TableStore.Client インスタンスを初期化します。

Table Store のエンドポイントを使用してクライアントを作成します。

API

```
    // " TableStore  Client " インスタンスを初期化します。
    //  endPoint  は  Table   Store  アドレスです（たとえば、 https ://
 instance . cn - hangzhou . ots . aliyun . com : 80 ) 。  https :// で
始まる必要があります。
    //  accessKeyI  d  は  Table   Store  へのアクセスに使用される
 AccessKeyI  D  です。
    //  accessKeyS  ecret  は、 Table   Store  へのアクセスに使用される
 AccessKeyS  ecret  です。
    //  instanceNa  me  はアクセスするインスタンスの名前です。  Table   Store
  コンソールでインスタンスを作成できます。
 func   NewClient ( endPoint ,  instanceNa  me ,  accessKeyI  d
 ,  accessKeyS  ecret   string ,  options  ...  ClientOpti  on ) *
 TableStore  Client
```

例

```
     client  =  NewClient (" your_insta  nce_endpoi  nt ",  "
 your_insta  nce_name ", " your_user_  id ", " your_user_  key ")
```

4.4 単一行操作

Table Store SDK には、PutRow、GetRow、UpdateRow、DeleteRow の単一行操作 API が用意されています。

PutRow

指定された行にデータを挿入します。

API

```
    // @ param   PutRowRequ  est   PutRow  操作の実行に必要なパラメーターを
カプセル化します。
    // @ return   PutRowResp  onse
     PutRow ( request  * PutRowRequ  est ) (* PutRowResp  onse ,
 error )
```

例

```
     putRowRequ  est  :=  new ( tablestore . PutRowRequ  est )
     putRowChan  ge  :=  new ( tablestore . PutRowChan  ge )
     putRowChan  ge . TableName  =  tableName
     putPk  :=  new ( tablestore . PrimaryKey )
     putPk . AddPrimary  KeyColumn (" pk1 ", " pk1value1 ")
     putPk . AddPrimary  KeyColumn (" pk2 ",  int64 ( 2 ))
     putPk . AddPrimary  KeyColumn (" pk3 ", [] byte (" pk3 "))

     putRowChan  ge . PrimaryKey  =  putPk
     putRowChan  ge . AddColumn (" col1 ", " col1data1 ")
     putRowChan  ge . AddColumn (" col2 ",  int64 ( 3 ))
     putRowChan  ge . AddColumn (" col3 ", [] byte (" test "))
```

```
    putRowChan  ge . SetConditi  on ( tablestore . RowExisten
ceExpectat  ion_IGNORE )
    putRowRequ  est . PutRowChan  ge  =  putRowChan  ge
    _ ,  err  :=  client . PutRow ( putRowRequ  est )

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

- RowExistenceExpectation.IGNORE は、指定された行が存在するかどうかにかかわらず、新しいデータが挿入されることを示します。 挿入されたデータが既存のデータと同じ場合、既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_EXIST は、指定された行が存在する場合にのみ新しいデータが挿入されることを示します。 既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_NOT_EXIST は、指定された行が存在しない場合にのみデータが挿入されることを示します。

 注：

完全なサンプルコードは、『*PutRow@GitHub*』をご参照ください。

GetRow

この API は、指定された主キーに基づいて単一のデータ行を読み取ります。

API

```
    // テーブルからデータの行を返します。
    //
    // @ param  GetRowRequ  est  GetRow  オペレーションの実行に必要なパラ
メーターをカプセル化します。
    // @ return  GetRowResp  onse  GetRow  操作に対するレスポンスの内容で
す。
    GetRow ( request  * GetRowRequ  est ) (* GetRowResp  onse ,
 error )
```

例

データ行を読み取ります。

```
getRowRequ  est  :=  new ( tablestore . GetRowRequ  est )
    criteria  :=  new ( tablestore . SingleRowQ  ueryCriter  ia );
    putPk  :=  new ( tablestore . PrimaryKey )
    putPk . AddPrimary  KeyColumn (" pk1 ", " pk1value1 ")
    putPk . AddPrimary  KeyColumn (" pk2 ",  int64 ( 2 ))
    putPk . AddPrimary  KeyColumn (" pk3 ", [] byte (" pk3 "))

    criteria . PrimaryKey  =  putPk
    getRowRequ  est . SingleRowQ  ueryCriter  ia  =  criteria
```

```
    getRowRequ  est . SingleRowQ  ueryCriter  ia . TableName  =
tableName
    getRowRequ  est . SingleRowQ  ueryCriter  ia . MaxVersion  =  1
    getResp ,  err  :=  client . GetRow ( getRowRequ  est )
    if  err  ! =  nil  {
        fmt . Println (" getrow  failed  with  error :",  err )
    }  else  {
        fmt . Println (" get  row  col0  result  is  ", getResp .
Columns [ 0 ].  ColumnName ,  getResp . Columns [ 0 ].  Value ,)
    }
```

注：

GetRow@GitHub で完全なサンプルコードを入手してください。

UpdateRow

指定された行のデータを更新します。 指定された行が存在しない場合は、新しい行が追加されます。 指定された行が存在する場合、指定された列の値は、リクエストされた内容に合わせて、追加、変更、又は削除されます。

API

```
    // テーブル内のデータ行を更新します。
    // @ param  UpdateRowR  equest  UpdateRow  操作の実行に必要なパラ
メータをカプセル化します。
    // @ return  UpdateRowR  esponse  UpdateRow  オペレーションに対する
応答の内容
    UpdateRow ( request  * UpdateRowR  equest ) (* UpdateRowR
esponse ,  error )
```

例

データ行を更新します。

```
    updateRowR  equest  :=  new ( tablestore . UpdateRowR  equest )
    updateRowC  hange  :=  new ( tablestore . UpdateRowC  hange )
    updateRowC  hange . TableName  =  tableName
    updatePk  :=  new ( tablestore . PrimaryKey )
    updatePk . AddPrimary  KeyColumn (" pk1 ", " pk1value1 ")
    updatePk . AddPrimary  KeyColumn (" pk2 ",  int64 ( 2 ))
    updatePk . AddPrimary  KeyColumn (" pk3 ", [] byte (" pk3 "))
    updateRowC  hange . PrimaryKey  =  updatePk
    updateRowC  hange . DeleteColu  mn (" col1 ")
    updateRowC  hange . PutColumn (" col2 ",  int64 ( 77 ))
    updateRowC  hange . PutColumn (" col4 ", " newcol3 ")
    updateRowC  hange . SetConditi  on ( tablestore . RowExisten
ceExpectat  ion_EXPECT  _EXIST )
    updateRowR  equest . UpdateRowC  hange  =  updateRowC  hange
    _ ,  err  :=  client . UpdateRow ( updateRowR  equest )

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

注：

UpdateRow@GitHub で完全なサンプルコードを入手してください。

DeleteRow

API

```
    // テーブルからデータ行を削除します
    // @ param   DeleteRowR  equest   DeleteRow  操作の実行に必要なパラメーターをカプセル化します。
    // @ return   DeleteRowR  esponse   DeleteRow  操作に対する応答の内容です。
     DeleteRow ( request  * DeleteRowR  equest ) (* DeleteRowR  esponse ,  error )
```

例

この API はデータ行を削除します。

```
    deleteRowR  eq  :=  new ( tablestore . DeleteRowR  equest )
    deleteRowR  eq . DeleteRowC  hange  =  new ( tablestore . DeleteRowC  hange )
    deleteRowR  eq . DeleteRowC  hange . TableName  =  tableName
    deletePk  :=  new ( tablestore . PrimaryKey )
    deletePk . AddPrimary  KeyColumn (" pk1 ", " pk1value1 ")
    deletePk . AddPrimary  KeyColumn (" pk2 ",  int64 ( 2 ))
    deletePk . AddPrimary  KeyColumn (" pk3 ", [] byte (" pk3 "))
    deleteRowR  eq . DeleteRowC  hange . PrimaryKey  =  deletePk
    deleteRowR  eq . DeleteRowC  hange . SetConditi  on ( tablestore . RowExisten  ceExpectat  ion_EXPECT  _EXIST )
    clConditio  n1  :=  tablestore . NewSingleC  olumnCondi  tion (" col2 ",  tablestore . CT_EQUAL ,  int64 ( 3 ))
    deleteRowR  eq . DeleteRowC  hange . SetColumnC  ondition ( clConditio  n1 )
    _ ,  err  :=  client . DeleteRow ( deleteRowR  eq )
    if  err  ! =  nil  {
        fmt . Println (" delete  failed  with  error :",  err )
   }  else  {
        fmt . Println (" delete  row  finished ")
   }
```

注：

完全なサンプルコードは、『DeleteRow@GitHub』をご参照ください。

4.5 複数行操作

Table Store SDK には、BatchGetRow、BatchWriteRow、GetRange および GetByIterator の複数行操作 API が用意されています。

BatchGetRow

1 つ以上のテーブルから複数のデータ行をバッチで読み取ります。

BatchGetRow 操作は、複数の GetRow 操作のセットと見なすことができます。 各操作が実行され、結果が返され、容量ユニットが個別に消費されます。

多数の GetRow 操作の実行と比較して、BatchGetRow 操作はリクエストの応答時間を効果的に短縮し、データ読み取り速度を向上させます。

API

```
    // テーブル内の複数のデータ行を返します。
    //
    // @ param   BatchGetRo  wRequest   BatchGetRo  w  操作の実行に必要なパラメーターをカプセル化します。
    // @ return   BatchGetRo  wResponse   BatchGetRo  w  操作に対するレスポンスの内容です。
     BatchGetRo  w ( request  * BatchGetRo  wRequest ) (* BatchGetRo
 wResponse ,  error )
```

例

10 データ行をまとめて読み取ります。

```
 batchGetRe  q  := & tablestore . BatchGetRo  wRequest {}
 mqCriteria  := & tablestore . MultiRowQu  eryCriteri  a {}

 for   i  :=  0 ;  i  <  10 ;  i ++ {
     pkToGet  :=  new ( tablestore . PrimaryKey )
     pkToGet . AddPrimary  KeyColumn (" pk1 ", " pk1value1 ")
     pkToGet . AddPrimary  KeyColumn (" pk2 ",  int64 ( i ))
     pkToGet . AddPrimary  KeyColumn (" pk3 ", [] byte (" pk3 "))
     mqCriteria . AddRow ( pkToGet )
     mqCriteria . MaxVersion  =  1
}
 mqCriteria . TableName  =  tableName
 batchGetRe  q . MultiRowQu  eryCriteri  a  =  append ( batchGetRe  q
 . MultiRowQu  eryCriteri  a ,  mqCriteria )
 batchGetRe  sponse ,  err  :=  client . BatchGetRo  w ( batchGetRe  q
 )

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

注：

- BatchGetRow は、条件付きステートメントを使用したフィルタリングをサポートしています。
- 完全なサンプルコードは、『BatchGetRow@GitHub』をご参照ください。

BatchWriteRow

1 つ以上のテーブル内の複数のデータ行をまとめて挿入、変更、または削除します。

基本的には、複数の PutRow、UpdateRow および DeleteRow 操作のセットです。 各操作は実行され、結果は独立して返され、容量ユニットは独立して消費されます。

API

```
    // 複数のテーブルの複数のデータ行を追加、削除、または修正します。
    //
    // @ param   BatchWrite  RowRequest   BatchWrite  Row  操作の実行に必要なパラメーターをカプセル化します。
    // @ return   BatchWrite  RowRespons  e   BatchWrite  Row  操作に対するレスポンスの内容です。
     BatchWrite  Row ( request  * BatchWrite  RowRequest ) (* BatchWrite  RowRespons  e , error )
```

例

100 データ行をまとめて書き込みます。

```
    batchWrite  Req  := & tablestore . BatchWrite  RowRequest {}
    for   i  :=  0 ;  i  <  100 ;  i ++ {
        putRowChan  ge  :=  new ( tablestore . PutRowChan  ge )
        putRowChan  ge . TableName  =  tableName
        putPk  :=  new ( tablestore . PrimaryKey )
        putPk . AddPrimary  KeyColumn (" pk1 ", " pk1value1 ")
        putPk . AddPrimary  KeyColumn (" pk2 ",  int64 ( i ))
        putPk . AddPrimary  KeyColumn (" pk3 ", [] byte (" pk3 "))
        putRowChan  ge . PrimaryKey  =  putPk
        putRowChan  ge . AddColumn (" col1 ", " fixvalue ")
        putRowChan  ge . SetConditi  on ( tablestore . RowExisten ceExpectat  ion_IGNORE )
        batchWrite  Req . AddRowChan  ge ( putRowChan  ge )
   }

    response ,  err  :=  client . BatchWrite  Row ( batchWrite  Req )
    if   err  ! =  nil  {
        fmt . Println (" batch   request   failed   with :", response )
   }  else  {
        fmt . Println (" batch   write   row   finished ")
   }
```

注：

- BatchWriteRow supports filtering using conditional statements.
- 完全なサンプルコードは、『BatchWriteRow@GitHub』をご参照ください。

GetRange

この API は、指定されたプライマリキー範囲内のデータを読み取ります。

API

```
    // テーブル内の指定された範囲内の複数のデータ行を読み込みます。
    //
    // @ param   GetRangeRe  quest   GetRange  操作の実行に必要なパラメーターをカプセル化します。
    // @ return   GetRangeRe  sponse   GetRange  操作に対するレスポンスの内容です
     GetRange ( request  * GetRangeRe  quest ) (* GetRangeRe  sponse , error )
```

例

指定範囲内のデータを読み込みます。

```
getRangeRe  quest  := & tablestore . GetRangeRe  quest {}
    rangeRowQu  eryCriteri  a  := & tablestore . RangeRowQu eryCriteri  a {}
    rangeRowQu  eryCriteri  a . TableName  =  tableName

    startPK  :=  new ( tablestore . PrimaryKey )
    startPK . AddPrimary  KeyColumnW  ithMinValu  e (" pk1 ")
    startPK . AddPrimary  KeyColumnW  ithMinValu  e (" pk2 ")
    startPK . AddPrimary  KeyColumnW  ithMinValu  e (" pk3 ")
    endPK  :=  new ( tablestore . PrimaryKey )
    endPK . AddPrimary  KeyColumnW  ithMaxValu  e (" pk1 ")
    endPK . AddPrimary  KeyColumnW  ithMaxValu  e (" pk2 ")
    endPK . AddPrimary  KeyColumnW  ithMaxValu  e (" pk3 ")
    rangeRowQu  eryCriteri  a . StartPrima  ryKey  =  startPK
    rangeRowQu  eryCriteri  a . EndPrimary  Key  =  endPK
    rangeRowQu  eryCriteri  a . Direction  =  tablestore . FORWARD
    rangeRowQu  eryCriteri  a . MaxVersion  =  1
    rangeRowQu  eryCriteri  a . Limit  =  10
    getRangeRe  quest . RangeRowQu  eryCriteri  a  =  rangeRowQu eryCriteri  a

    getRangeRe  sp ,  err  :=  client . GetRange ( getRangeRe  quest )

    fmt . Println (" get  range  result  is  " , getRangeRe  sp )

    for  ; ; {
        if  err  ! =  nil  {
            fmt . Println (" get  range  failed  with  error :", err )
       }
```

```
        if  ( len ( getRangeRe  sp . Rows ) >  0 ) {
            for   _ ,  row  :=  range   getRangeRe  sp . Rows  {
                fmt . Println (" range   get   row   with   key ",
row . PrimaryKey . PrimaryKey  s [ 0 ].  Value ,  row . PrimaryKey .
PrimaryKey  s [ 1 ].  Value ,  row . PrimaryKey . PrimaryKey  s [ 2
].  Value )
            }
             if   getRangeRe  sp . NextStartP  rimaryKey  ==  nil  {
                break
            }  else  {
                fmt . Println (" next   pk   is  :",  getRangeRe  sp
. NextStartP  rimaryKey . PrimaryKey  s [ 0 ].  Value ,  getRangeRe
sp . NextStartP  rimaryKey . PrimaryKey  s [ 1 ].  Value ,
getRangeRe  sp . NextStartP  rimaryKey . PrimaryKey  s [ 2 ].  Value
)
                getRangeRe  quest . RangeRowQu  eryCriteri  a .
StartPrima  ryKey  =  getRangeRe  sp . NextStartP  rimaryKey
                getRangeRe  sp ,  err  =  client . GetRange (
getRangeRe  quest )
            }
       }  else  {
            break
       }

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

注：

- **GetRange は、条件文を使用したフィルタリングをサポートしています。**
- **GetRange 操作を実行すると、データがページングされることがあります。**
- **完全なサンプルコードは、『*GetRange@GitHub*』をご参照ください。**

5 NodeJS SDK

5.1 はじめに

このドキュメントは NodeJS SDK v4.0 以降を Table Store にインストールして使用する方法を説明します。 Alibaba Cloud Table Store を有効にし、AccessKeyID と AccessKeySecret を作成したと仮定します。

- Alibaba Cloud Table Store をまだ有効にしていない場合、または Table Store についてもっと知りたい場合は、 「*Table Store ホームページ*」をご参照ください。
- AccessKeyID と AccessKeySecret を作成していない場合は、 *Alibaba Cloud AccessKey コンソール*にログインします。AccessKey を作成します。

SDK をダウンロードします。

- *SDK パッケージ*
- *GitHub*

バージョン

最新バージョン: 4.0.0

5.2 インストール

前提条件

NodeJS v4.0 以降に適用されます。

手順

次のコマンドを実行します。

```
npm   install   tablestore
```

プログラム例

テーブルストア NodeJS SDK は参照または使用のために多様なプログラム例を提供します。 次のいずれかの方法でサンプルプログラムを入手できます。

- Table Store NodeJS SDK をダウンロードして解凍し、"examples" フォルダにあるサンプルプログラムを見つけます。
- Table Store NodeJS の *GitHub* プロジェクトにアクセスします。

5.3 初期化

TableStore Client は Table Store のクライアントで、テーブルを操作したり、単一行または複数行に対してデータを読み書きするための一連のメソッドを持つ呼び出し元を提供します。

エンドポイントを決定する

エンドポイントは、リージョン内の Alibaba Cloud Table Store のドメイン名アドレスです。次の形式をサポートしています。

エンドポイント種類	説明
リージョンアドレス	`https://instance.cn-hangzhou.ots.aliyuncs.com` のような 現在の Table Store インスタンスのリージョンです。

Table Store のリージョンアドレス

Table Store インスタンスが存在するエンドポイントを照会するには、次の手順に従います。

1. *Table Store コンソール*にログインします。
2. インスタンスの詳細 ページにアクセスします。インスタンスのエンドポイントであるインスタンスアクセスアドレスを見つけます。

AccessKey を設定する

Alibaba Cloud Table Store にアクセスするには、署名認証用の有効な AccessKey (AccessKeyId と AccessKeySecret を含む) が必要です。 AccessKey を取得するには、次の手順に従います。

1. *Alibaba Cloud アカウント*を登録します。
2. *AccessKey コンソール*にログインします。AccessKeyId と AccessKeySecret を作成します。

AccessKeyId と AccessKeySecret を取得したら、Table Store のエンドポイントを使用してクライアントを作成し、TableStore.Client インスタンスを初期化します。

例:

```
        var client = new TableStore.Client({
 accessKeyId: '<your access key id>',
 secretAccessKey: '<your access key secret>',
 endpoint: '<your endpoint>',
 instancename: '<your instance name>',
 maxRetries: 20,// デフォルトの再試行回数は 20 です。 このパラメーターは無視してかまいません
```

```
    });
```

5.4 Long 型

Table Store は 5 つの データ型を提供します。 Table Store データ型と NodeJS SDK データ型の関係は次のとおりです。

Table Store	NodeJS SDK	説明
String	String	JavaScript の基本データ型
Interger 型	int64 型	NodeJs SDK カプセル化のデータ型
Dobule 型	number 型	JavaScript の基本データ型
Boolean 型	Boolean 型	JavaScript の基本データ型
Binary 型	Buffer 型	NodeJS のバッファーオブジェクト

int64 型

Table Store の Integer 型は 64 ビット符号付き整数で、JavaScript では対応するデータ型がありません。 したがって、NodeJS には 64 ビット符号付き整数を表すことができるデータ型が必要です。

次の操作を実行することもできます。

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

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

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

5.5 単一行操作

Table Store SDK には、PutRow、GetRow、UpdateRow、DeleteRow の単一行操作 API が用意されています。

PutRow

指定された行にデータを挿入します。

API

```
  /**
   * 指定された行にデータを挿入します。 指定された行が存在しない場合は、新しい行が
追加されます。 行が存在する場合は、元の行が上書きされます。
   */
   putRow ( params ,  callback )
```

例

```
 var   TableStore  =  require ('../ index . js ');
 var   Long  =  TableStore . Long ;
 var   client  =  require ('./ client ');

 var   currentTim  eStamp  =  Date . now ();
 var   params  = {
   tableName : " sampleTabl  e ",
   condition :  new   TableStore . Condition ( TableStore .
 RowExisten  ceExpectat  ion . IGNORE ,  null ),
   primaryKey : [{ ' gid ':  Long . fromNumber ( 20013 ) }, { ' uid
':  Long . fromNumber ( 20013 ) }],
   attributeC  olumns : [
    { ' col1 ': ' Table   Store ' },
    { ' col2 ': ' 2 ', ' timestamp ':  currentTim  eStamp  },
    { ' col3 ':  3 . 1  },
    { ' col4 ': - 0 . 32  },
    { ' col5 ':  Long . fromNumber ( 123456789 ) }
 ],
   returnCont  ent : {  returnType :  TableStore . ReturnType .
 Primarykey  }
};

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

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

- RowExistenceExpectation.IGNORE は、指定された行が存在するかどうかにかかわらず、新しいデータが挿入されることを示します。 挿入されたデータが既存のデータと同じ場合、既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_EXIST は、指定された行が存在する場合にのみ新しいデータが挿入されることを示します。 既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_NOT_EXIST は、指定された行が存在しない場合にのみデータが挿入されることを示します。

 注：

完全なサンプルコードは、『PutRow@GitHub』をご参照ください。

GetRow

この API は、指定されたプライマリキーに基づいて単一のデータ行を読み取ります。

API

```
/**
 * 与えられたプライマリキーに基づいて単一のデータ行を読み込みます。
 */
getRow ( params , callback )
```

例

Read a data row.

```
var TableStore = require ('../ index . js ');
var Long = TableStore . Long ;
var client = require ('./ client ');

var params = {
  tableName : " sampleTabl e ",
  primaryKey : [{ ' gid ': Long . fromNumber ( 20004 ) }, { ' uid
':  Long . fromNumber ( 20004 ) }],
  maxVersion s : 2
};
var condition = new TableStore . CompositeC ondition (
TableStore . LogicalOpe rator . AND );
condition . addSubCond ition ( new TableStore . SingleColu
mnConditio n (' name ', ' john ', TableStore . Comparator Type .
EQUAL ));
condition . addSubCond ition ( new TableStore . SingleColu
mnConditio n (' addr ', ' china ', TableStore . Comparator Type .
EQUAL ));

params . columnFilt er = condition ;

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

注：

完全なサンプルコードは、『GetRow@GitHub』をご参照ください。

UpdateRow

指定された行のデータを更新します。 指定された行が存在しない場合は、新しい行が追加されます。 指定された行が存在する場合は、リクエスト内容に基づいて指定された列の値が追加、変更、削除されます。

API

```
  /**
   * 指定された行のデータを更新します。 指定された行が存在しない場合は、新しい行が追加されます。 指定された行が存在する場合は、指定された列の値は、リクエストに合わせて、追加、変更、又は削除されます。
   */
   updateRow ( params ,  callback )
```

例

データ行を更新します。

```
 var   TableStore  =  require ('../ index . js ');
 var   Long  =  TableStore . Long ;
 var   client  =  require ('./ client ');

 var   params  = {
     tableName : " sampleTabl  e ",
     condition :  new   TableStore . Condition ( TableStore .
 RowExisten  ceExpectat  ion . IGNORE ,  null ),
     primaryKey : [{ ' gid ':  Long . fromNumber ( 9 ) }, { ' uid ':
 Long . fromNumber ( 90 ) }],
     updateOfAt  tributeCol  umns : [
         { ' PUT ': [{ ' col4 ':  Long . fromNumber ( 4 ) }, { ' col5
 ': ' 5 ' }, { ' col6 ':  Long . fromNumber ( 6 ) }] },
         { ' DELETE ': [{ ' col1 ':  Long . fromNumber ( 1496826473
 186 ) }] },
         { ' DELETE_ALL ': [' col2 '] }
    ]
};

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

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

注：

完全なサンプルコードは、『*UpdateRow@GitHub*』をご参照ください。

DeleteRow

API

```
  /**
   * データ行を削除します。
   */
   deleteRow ( params ,  callback )
```

例

データ行を削除します。

```
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);
});
```

注:

完全なサンプルコードは、『DeleteRow@GitHub』をご参照ください。

5.6 複数行操作

Table Store SDK には、BatchGetRow、BatchWriteRow、GetRange および GetByIterator の複数行操作 API が用意されています。

BatchGetRow

1つ 以上のテーブルから複数のデータ行をバッチで読み取ります。

BatchGetRow 操作は、基本的に複数の GetRow 操作のセットです。 各操作が実行され、結果が返され、容量ユニットが個別に消費されます。

多数の GetRow 操作の実行と比較して、BatchGetRow 操作はリクエストの応答時間を効率的に短縮し、データ読み取り速度を向上させます。

API

```
/**
 * 1 つ以上のテーブルからバッチでいくつかのデータ行を読みます。
 */
batchGetRow(params, callback)
```

例

複数のテーブルと複数のデータ行をまとめて読み取り、単一行にエラーが発生した場合は操作を再試行します。

```
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 isAllSucess = 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 ) {
                    isAllSucess = 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 );
                    }
                     faildReque  st . primaryKey . push ( pks );

                } else {
                    // get   success   data
                }
            }

             if  ( faildReque  st . primaryKey . length  >  0 ) {
                 retryReque  st . tables . push ( faildReque  st );
            }
        }

         if  (!  isAllSucce  ss  &&  retryCount ++ <  maxRetryTi  mes
 ) {
             batchGetRo  w ( retryReque  st );
        }

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

 batchGetRo  w ( params ,  maxRetryTi  mes );
```

注：

- BatchGetRow は、条件付きステートメントを使用したフィルタリングをサポートしています。
- 完全なサンプルコードは、『BatchGetRow@GitHub』をご参照ください。

BatchWriteRow

1 つ以上のテーブル内の複数のデータ行をまとめて挿入、変更、または削除します。

BatchWriteRow 操作は、基本的に複数の PutRow、UpdateRow および DeleteRow 操作のセットです。 各操作が実行され、結果が返され、容量ユニットが個別に消費されます。

API

```
  /**
   * データ行をまとめて変更します。
   */
   batchWrite  Row ( params ,  callback )
```

例

データ行をまとめて書き込みます。

```
 var   client  =  require ('./ client ');
 var   TableStore  =  require ('../ index . js ');
```

```
var   Long  =  TableStore . Long ;

var   params  = {
    tables : [{
        tableName : ' sampleTabl  e ',
        rows : [{
            type : ' PUT ',
            condition :  new   TableStore . Condition ( TableStore .
RowExisten  ceExpectat  ion . IGNORE ,  null ),
            primaryKey : [{ ' gid ':  Long . fromNumber ( 8 ) }, { '
uid ':  Long . fromNumber ( 80 ) }],
            attributeC  olumns : [{ ' attrCol1 ': ' test1 ' }, { '
attrCol2 ': ' test2 ' }],
            returnCont  ent : {  returnType :  TableStore .
ReturnType . Primarykey  }
        }],
    }],
};

client . batchWrite  Row ( params ,  function  ( err ,  data ) {

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

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

注:

- BatchWriteRow は、条件文を使用したフィルタリングをサポートしています。
- 完全なサンプルコードは、『*BatchWriteRow@GitHub*』をご参照ください。

GetRange

指定されたプライマリキー範囲内のデータを読み取ります。

API

```
/**
 * 指定されたプライマリキー範囲内のデータを読み取ります。
 */
getRange ( params ,  callback )
```

例

指定範囲内のデータを読み込みます。

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

var   params  = {
  tableName : " sampleTabl  e ",
  direction :  TableStore . Direction . FORWARD ,
```

```
    inclusiveS  tartPrimar  yKey : [{ " gid ":  TableStore . INF_MIN
 }, { " uid ":  TableStore . INF_MIN  }],
    exclusiveE  ndPrimaryK  ey : [{ " gid ":  TableStore . INF_MAX  },
 { " uid ":  TableStore . INF_MAX  }],
    limit :  50
};

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

   // If  the  value  of  data . next_start  _primary_k  ey  is
 not  empty ,  continue  to  read  the  data
    if  ( data . next_start  _primary_k  ey ) {

   }

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

注：

- GetRange は、条件文を使用したフィルタリングをサポートしています。
- GetRange 操作を実行すると、データがページングされる可能性があることにご注意ください。
- 完全なサンプルコードは、『GetRange@GitHub』をご参照ください。

5.7 エラー処理

方法

Table Store NodeJS SDK は 、 `Exception` メソッドを使用して、エラー処理をします。 呼び出された API が例外をスローしない場合、操作は成功します。 例外がスローされると、操作は失敗します。

：

BatchGetRow や BatchWriteRow などのバッチ操作 API は、システムが各行のステータスが成功したことを確認した場合にのみ正常に呼び出されます。

例外

Table Store NodeJS SDK では、すべてのエラーは統一された方法で処理され、コールバックメソッドの "err" パラメーターに返されます。 したがって、返されたデータを取得する前に、"err" パラメーターに値があるかどうかを確認する必要があります。 Table Store サーバーでエラーが

報告されると、RequestId が返され、要求を一意に識別する UUID が指定されます。 問題が解決しない場合は、RequestId を保存して チケットを起票し、サポートセンターへお問い合わせください。

再試行

SDK でエラーが発生すると、システムは自動的に操作を再試行します。 デフォルトのポリシーでは、最大 20 回の再試行と最大 3000ms の再試行間隔が設定されています。システムがスロットルエラーおよび読み取り関連の内部サーバーエラーに対して操作を再試行する方法については、`tablestore / lib / retry . js` をご参照ください。

6 .NET SDK

6.1 はじめに

はじめに

このドキュメントでは、C# SDK 3.0.0 for Table Store のインストール方法と使用方法について説明します。 このドキュメントでは、Alibaba Cloud Table Store を既に有効化して AccessKeyID と AccessKeySecret を作成していることを前提としています。

- Table Store を有効にしていない場合、またはサービスについて知りたい場合は、 *Table Store プロダクトのホームページ*をご参照ください。
- AccessKeyID と AccessKeySecret を作成していない場合は、 *Alibaba Cloud Access Key コンソール*にアクセスし、AccessKey を作成します。

SDK パッケージのダウンロード

- *SDK パッケージ*
- *GitHub*

バージョンサイクルについては、 ここをクリックしてご参照ください。

更新と互換性

最新バージョン 3.0.0 について

- フィルタが追加されました。
- コンパイルプロセス中に生成された警告は消去されます。
- 不要な依存パッケージは消去されます。
- 無用なコードは消去されます。
- テンプレートと呼ばれるコードは単純化されています。
- 無効なパラメーターチェックが追加されました。
- ユーザー設定パラメーターはトリミングされます。
- UserAgent ヘッダーが追加されました。
- DLL ファイル名 Aliyun.dll が Aliyun.TableStore.dll に変更されました。
- オープンソースコードは GitHub にリリースされています。

SDK V2.x.x について

- 部分的インターフェイスとの互換性がありません。`Condition . IGNORE`、`Condition . EXPECT_EXI ST` および `Condition . EXPECT_NOT _EXIST` は削除されます。
- DLL ファイル名 *Aliyun . dll* は、*Aliyun . TableStore . dll* に変更されました。

6.2 初期設定

OTSClient は Table Store のクライアントです。呼び出し側に、テーブルを操作し、単一行または複数の行に対してデータを読み書きするための一連の方法を提供します。

エンドポイントの決定

エンドポイントは、リージョン内の Alibaba Cloud Table Store のドメインです。以下のフォーマットに対応しています。

エンドポイントの種類	説明
リージョンアドレス	Table Store インスタンスが存在するリージョンのアドレス です (例: `https :// instance . cn - hangzhou . ots . aliyuncs . com`)。

テーブルストアのリージョンアドレス

Table Store インスタンスが存在するエンドポイントを照会するには、次の手順に従います。

1. *Table Store コンソール*にログインします。
2. インスタンスの詳細ページにアクセスします。インスタンスのエンドポイントである インスタンスアクセス URL を見つけます。

AccessKey の設定

Alibaba Cloud Table Store サービスにアクセスするには、署名認証用の有効な AccessKey (AccessKeyID と AccessKeySecret を含む) が必要です。AccessKey を取得するには、次の手順に従います。

1. Alibaba Cloud ウェブサイトで *Alibaba Cloud アカウント*を登録します。
2. *AccessKey コンソール*にログインして、AccessKey を申請します。

AccessKeyID と AccessKeySecret を取得したら、Table Store のエンドポイントを使用してクライアントを作成し、OTSClient インスタンスを初期化します。

API:

```
    /// < summary >
    ///  OTSClient  コンストラクタ。
    ///</ summary >
    /// < param   name =" endPoint "> Table   Store  サービスのアドレス
 。たとえば、 https :// instance . cn - hangzhou . ots . aliyun . com :
80  です。 「 https ://」で始める必要があります。 </  param >
    ///< param   name  = " accessKeyI  D ">  Table   Store  の
AccessKeyI  D  です。</ param >
    /// < param   name =" accessKeyS  ecret ">  Table   Store  の
AccessKeyS  ecret  です。 </ param >
    /// < param   name =" instanceNa  me ">  Table   Store   instance
 名前です。  Alibaba   Cloud   Table   Store  コンソールで作成されている必要
があります。 </ param >
public   OTSClient ( string   endPoint ,  string   accessKeyI  D ,
string   accessKeyS  ecret ,  string   instanceNa  me );

    /// < summary >
    ///  OTSClientC  onfig  インスタンスを使用して  OTSClient  インスタンス
を作成します。
    /// </ summary >
    /// < param   name =" config "> OTSClientC  onfig   instance </
param >
     public   OTSClient ( OTSClientC  onfig   config );
```

例

```
    //  OTSClientC  onfig  オブジェクトを作成します。
     var   config  =  new   OTSClientC  onfig ( Endpoint ,
AccessKeyI  d ,  AccessKeyS  ecret ,  InstanceNa  me );

    // ログ出力を無効にします（この機能はデフォルトで有効になっています）。
     config . OTSDebugLo  gHandler  =  null ;
     config . OTSErrorLo  gHandler  =  null ;

    //  OTSClientC  onfig  を使用して  OTSClient  オブジェクト を作成しま
す。
     var   otsClient  =  new   OTSClient ( config );

    //  OTSClient  を使用してデータを挿入またはクエリします。
```

注：

- ConnectionLimit は OTSClientConfig オブジェクトで設定できます。 ConnectionLimit が設定されていない場合、デフォルト値は 300 です。
- OTSClientConfig の OTSDebugLogHandler と OTSErrorLogHandler はロギング動作を制御し、設定可能です。
- OTSClientConfig の RetryPolicy は再試行ロジックを制御します。 デフォルトの再試行ポリシーが提供されています。 独自の再試行ポリシーを定義できます。

マルチスレッド

- マルチスレッドがサポートされています。
- 複数のスレッドが同じ OTSClient オブジェクトを使用することを推奨します。

6.3 テーブルレベルの操作

Table Store SDK には、CreateTable、ListTable、DeleteTable、UpdateTable、DescribeTable のテーブルレベルの操作インターフェイスがあります。

CreateTable

与えられたテーブル構造情報に基づいてテーブルを作成します。

Table Store でテーブルを作成するときは、テーブルのプライマリキーを指定する必要があります。 プライマリキーには 1 つから 4 つのプライマリキー列があります。 各プライマリキー列には名前と型があります。

API

```
        /// < summary >
        /// テーブル情報（テーブル名、プライマリキーのデザイン、予約済み読み書きのスループットなど）に基づいてテーブルを作成します。
        /// </ summary >
        ///< param   name  = " request "> リクエストパラメーター </ param >
        ///< returns >  Create   Table  から返される情報です。 戻されたインスタンスは  null  値です。具体的な情報はありません。
        /// </ returns >
         public   CreateTabl  eResponse   CreateTabl  e ( CreateTabl eRequest   request );

        /// < summary >
        ///  CreateTabl  e  の非同期形式。
        /// </ summary >
         public   Task < CreateTabl  eResponse >  CreateTabl  eAsync ( CreateTabl  eRequest   request );
```

注：
テーブルが Table Store に作成された後、テーブルを完全にロードするのに数秒かかります。この間は何の操作もしないでください。

例

2つのプライマリキー列と (0,0) の予約済み読み書きスループットを持つテーブルを作成します。

```
        // 主キーの数量、名前、タイプなど、主キー列のスキーマを作成します
```

```
        // 最初のプライマリキー列（シャーディング列）は pk0 という名前で、整数
型です。
        // 2 番目のプライマリキー列は pk1 という名前で、 String 型です。
         var primaryKey Schema = new PrimaryKey Schema ();
         primaryKey Schema . Add (" pk0 ", ColumnValu eType .
 Integer );
         primaryKey Schema . Add (" pk1 ", ColumnValu eType .
 String );

        // テーブル名とプライマリキーカラムに基づいて tableMeta を作成します。
         var tableMeta = new TableMeta (" SampleTabl e ",
 primaryKey Schema );

        // 予約済み読み取りスループットと予約書き込みスループットを 0 に設定し
ます
         var reservedTh roughput = new CapacityUn it ( 0 , 0
 );

         try
        {
            // CreateTabl eRequest オブジェクトを作成します
             var request = new CreateTabl eRequest ( tableMeta
 , reservedTh roughput );

            // クライアントの CreateTabl e インターフェイスを呼び出しま
す。 例外がスローされなければ、テーブルは正常に作成されます。 例外がスローされた場
合、テーブルは作成されません。
             otsClient . CreateTabl e ( request );

             Console . WriteLine (" Create table succeeded .") ;
        }
        // 例外を処理します。
         catch ( Exception ex )
        {
             Console . WriteLine (" Create table failed ,
 exception :{ 0 }", ex . Message );
        }
```

コードの詳細は、『*CreateTable@GitHub*』をご参照ください。

ListTable

現在のインスタンスの下にあるすべてのテーブルの名前を取得します。

API

```
        /// < summary >
        /// 現在のインスタンスの下にあるすべてのテーブルの名前を取得します。
        /// </ summary >
        ///< param name = " request "> リクエストパラメーター </ param
 >
        /// < returns > ListTable から戻され、テーブル名リストを取得するた
めに使用される情報です。</ returns > </ returns >
         public ListTableR esponse ListTable ( ListTableR equest
   request );

        /// < summary >
        /// ListTable の非同期形式。
        /// </ summary >
         public Task < ListTableR esponse > ListTableA sync (
 ListTableR equest request );
```

```
```

例

インスタンスの下にあるすべてのテーブルの名前を取得します。

```
         var   request  =  new   ListTableR  equest ();
         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 );
        }
```

UpdateTable

指定されたテーブルの予約された読み取りまたは書き込みのスループット値を更新します。

API

```
        /// < summary >
        /// 指定されたテーブルの予約された読み取りまたは書き込みのスループット値を
更新します。 スループットが正常に更新されてから  1  分以内に新しい値が有効になりま
す。
        /// </ summary >
        ///< param   name  = " request "> リクエストパラメーターです。テーブ
ル名と予約済み読み書きのスループットを含みます。 </ param >
        ///< returns > 更新された予約済みの読み書きスループットとその他の情報を
含みます。 </ returns >
         public   UpdateTabl  eResponse   UpdateTabl  e ( UpdateTabl
 eRequest   request );

        /// < summary >
        ///  UpdateTabl  e  の非同期フォーム。
        /// </ summary >
         public   Task < UpdateTabl  eResponse >  UpdateTabl  eAsync (
 UpdateTabl  eRequest   request );
```

例

テーブルの読み取り CU と書き込み CU をそれぞれ 1 と 2 に更新します。

```
        // 予約読み取りスループットを  1  に、予約書き込みスループットを  2  に
更新します。
         var   reservedTh  roughput  =  new   CapacityUn  it ( 1 ,  2
 );

        //  UpdateTabl  eRequest  オブジェクトを作成します。
```

```
         var   request  =  new   UpdateTabl  eRequest (" SampleTabl  e
",  reservedTh  roughput );
        try
        {
            // テーブルの予約済み読み書きスループットを更新するためにインターフェ
イスを呼び出します
             otsClient . UpdateTabl  e ( request );

            // 例外がスローされなければ、実行は成功します。
             Console . Writeline (" Update   table   succeeded .");
        }
         catch  ( Exception   ex )
        {
            // 例外がスローされてエラーメッセージが表示されると、実行は失敗しま
す。
             Console . WriteLine (" Update   table   failed ,
 exception :{ 0 }",  ex . Message );
        }
```

コードの詳細は、『*UpdateTable@GitHub*』をご参照ください。

DescribeTable

指定されたテーブルの構造情報と予約済み読み書きスループット値を問い合わせます。

API

```
        /// < summary >
        /// 指定されたテーブルの構造情報と予約済み読み書きスループット値を照会しま
す。
        /// </ summary >
        ///< param   name  = " request "> テーブル名を含むリクエストパラメー
ターです。 </ param >
        ///< returns > 指定されたテーブルの構造情報と予約済み読み書きスループッ
ト値を含みます。 </ returns > </ returns >
         public   DescribeTa  bleRespons  e   DescribeTa  ble (
 DescribeTa  bleRequest   request );

        /// < summary >
        ///  DescribeTa  ble  の非同期形式。
        /// </ summary >
         public   Task < DescribeTa  bleRespons  e >  DescribeTa
bleAsync ( DescribeTa  bleRequest   request );
```

例

テーブルの説明情報を取得します。

```
         try
        {
             var   request  =  new   DescribeTa  bleRequest ("
 SampleTabl  e ");
             var   response  =  otsClient . DescribeTa  ble ( request
);

            // 指定されたテーブルの説明情報を表示します。
             Console . Writeline (" Describe   table   succeeded .") ;
```

```
            Console . WriteLine (" LastIncrea  seTime : { 0 }",
response . ReservedTh  roughputDe  tails . LastIncrea  seTime );
            Console . WriteLine (" LastDecrea  seTime : { 0 }",
response . ReservedTh  roughputDe  tails . LastDecrea  seTime );
            Console . WriteLine (" NumberOfDe  creaseToda  y : { 0
}",  response . ReservedTh  roughputDe  tails . LastIncrea  seTime );
            Console . WriteLine (" ReadCapaci  ty : { 0 }",  response
. ReservedTh  roughputDe  tails . CapacityUn  it . Read );
            Console . WriteLine (" WriteCapac  ity : { 0 }",
response . ReservedTh  roughputDe  tails . CapacityUn  it . Write );
        }
         catch  ( Exception  ex )
        {
            //  Execution  fails  if  an  exception  is  thrown
,  and  an  error  message  is  printed .
            Console . WriteLine (" Describe  table  failed ,
exception :{ 0 }",  ex . Message );
        }
```

コードの詳細は、『*DescribeTable@GitHub*』をご参照ください。

DeleteTable

インスタンス下の指定されたテーブルを削除します。

API

```
        /// < summary >
        /// 指定されたテーブル名に対応するテーブルを削除します。
        /// </ summary >
        ///< param  name  = " request "> テーブル名を含むリクエストパラメー
ターです。 </ param >
        ///< returns >  DeleteTabl  e  から戻される情報です。 返されたインス
タンスは  null  値です。具体的な情報はありません。
        /// </ returns >
         public  DeleteTabl  eResponse  DeleteTabl  e ( DeleteTabl
eRequest  request );

        /// < summary >
        ///  DeleteTabl  e  の非同期形式です。
        /// </ summary >
         public  Task < DeleteTabl  eResponse >  DeleteTabl  eAsync (
DeleteTabl  eRequest  request );
```

例

テーブルを削除します。

```
         var  request  =  new  DeleteTabl  eRequest (" SampleTabl  e
");
         try
        {
             otsClient . DeleteTabl  e ( request );
             Console . Writeline (" Delete  table  succeeded .");
        }
         catch  ( Exception  ex )
        {
```

```
            Console . WriteLine (" Delete   table   failed ,
exception :{ 0 }",  ex . Message );
        }
```

コードの詳細は、『DeleteTable@GitHub』をご参照ください。

6.4 単一行操作

Table Store SDK には、PutRow、GetRow、UpdateRow および DeleteRow の単一行操作インターフェイスがあります。

PutRow

指定された行にデータを挿入します。

API

```
        /// < summary >
        /// 指定されたテーブル名、プライマリキー、および属性に基づいてデータ行を書
き込みます。 操作によって消費された  CapacityUn  it  が返されます。
        /// </ summary >
        ///< param   name  = " request "> データ挿入要求 </ param >
        ///< returns > 操作によって消費された  CapacityUn  it  です。 <
returns >
         public   PutRowResp  onse   PutRow ( PutRowRequ  est
request );

        /// < summary >
        ///  PutRow  の非同期形式です。
        /// </ summary >
         public   Task < PutRowResp  onse >  PutRowAsyn  c (
PutRowRequ  est   request );
```

例 1

データ行を挿入します。

```
        // 行のプライマリキーを定義します。これは、テーブル作成時に  TableMeta
で定義されたプライマリキーと一致している必要があります。
         var   primaryKey  =  new   PrimaryKey ();
         primaryKey . Add (" pk0 ",  new   ColumnValu  e ( 0 ));
         primaryKey . Add (" pk1 ",  new   ColumnValu  e (" abc "));

        //  100  行の属性列を定義します。
         var   attribute  =  new   AttributeC  olumns ();
         attribute . Add (" col0 ",  new   ColumnValu  e ( 0 ));
         attribute . Add (" col1 ",  new   ColumnValu  e (" a "));
         attribute . Add (" col2 ",  new   ColumnValu  e ( true ));

         try
        {
            //  RowExisten  ceExpectat  ion . IGNORE  を使用して、指定さ
れた行が存在しない場合でもデータがまだ挿入されていることを示すデータ挿入用のリクエス
トオブジェクトを作成します。
```

```
            var   request  =  new   PutRowRequ  est (" SampleTabl  e
 ",  new   Condition ( RowExisten  ceExpectat  ion . IGNORE ),
                                  primaryKey ,  attribute );

            //  PutRow  を呼び出してデータを挿入します。
             otsClient . PutRow ( request );

            // 例外がスローされなければ、実行は成功します。
             Console . WriteLine (" Put   row   succeeded .");
        }
         catch  ( Exception   ex )
        {
            // 例外がスローされてエラーメッセージが表示されると、実行は失敗しま
す。
             Console . WriteLine (" Put   row   failed ,  exception :{
 0 }",  ex . Message );
        }
```

注：

- Condition.IGNORE、Condition.EXPECT_EXIST および Condition.EXPECT_NOT_EXISTは、v3.0.0 から非推奨になりました。 代わりに、新しい条件 (RowExistenceExpectation.IGNORE)、新しい条件 (RowExistenceExpectation.EXPECT_EXIST) および新しい条件(RowExistenceExpectation.EXPECT_NOT_EXIST) を使用してください。
- RowExistenceExpectation.IGNORE は、指定された行が存在しない場合でも新しいデータが挿入されることを示します。 挿入されたデータが既存のデータと同じ場合、既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_EXIST は、指定された行が存在する場合にのみ新しいデータが挿入されることを示します。 既存のデータは上書きされます。
- RowExistenceExpectation.EXPECT_NOT_EXIST は、指定された行が存在しない場合にのみデータが挿入されることを示します。
- 行条件に加えて、Condition パラメーターは v2.2.0 以降の列条件もサポートします。

コードの詳細は、『*PutRow@GitHub*』をご参照ください。

例 2

指定した条件に基づいてデータ行を挿入します。

次のコード例は、指定された行が存在し、col1 の値が 24 より大きい場合にのみデータを挿入します。

```
        // 行のプライマリキーを定義します。これは、テーブル作成時に  TableMeta
で定義されたプライマリキーと一致している必要があります。
        var   primaryKey  =  new   PrimaryKey ();
        primaryKey . Add (" pk0 ",  new   ColumnValu  e ( 0 ));
        primaryKey . Add (" pk1 ",  new   ColumnValu  e (" abc "));
```

```
        //  100  行の属性列を定義します。
         AttributeC  olumns  attribute  =  new  AttributeC  olumns
();
         attribute . Add (" col0 ",  new  ColumnValu  e ( 0 ));
         attribute . Add (" col1 ",  new  ColumnValu  e (" a "));
         attribute . Add (" col2 ",  new  ColumnValu  e ( true ));

         var  request  =  new  PutRowRequ  est ( tableName ,  new
Condition ( RowExisten  ceExpectat  ion . EXPECT_EXI  ST ),
                                    primaryKey ,  attribute );

        //  col0  の値が  24  より大きい場合は、 PutRow  が再度実行されて元の
値が上書きされます。
         try
        {
             request . Condition . ColumnCond  ition  =  new
Relational  Condition (" col0 ",
                                                Relational  Condition
. CompareOpe  rator . GREATER_TH  AN ,
                                                new  ColumnValu  e (
24 ));
             otsClient . PutRow ( request );

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

注：

- 単一の条件または条件の組み合わせを設定できます。 たとえば、col1 > 5 および pk2 < 'xyz' として2つのデータ挿入条件を設定できます。
- 属性列と主キー列は、Condition パラメーターをサポートしています。
- 条件で指定された列が行に存在しない場合は、RelationCondition の PassIfMissing が実行するアクションを制御します。 デフォルト値は true です。

コードの詳細は、*ConditionPutRow@GitHub*』をご参照ください。

例 3

データ行を非同期に挿入します。

```
         try
        {
             var  putRowTask  List  =  new  List < Task <
PutRowResp  onse >>();
             for  ( int  i  =  0 ;  i  <  100 ;  i ++)
            {
                //  行のプライマリキーを定義します。これは、テーブル作成時に
TableMeta  で定義されたプライマリキーと一致している必要があります。
                 var  primaryKey  =  new  PrimaryKey ();
```

```
                primaryKey . Add (" pk0 ",  new   ColumnValu  e ( i
));
                primaryKey . Add (" pk1 ",  new   ColumnValu  e ("
abc "));

               //  100  行の属性列を定義します
                var   attribute  =  new   AttributeC  olumns ();
                attribute . Add (" col0 ",  new   ColumnValu  e ( i
));
                attribute . Add (" col1 ",  new   ColumnValu  e (" a
"));
                attribute . Add (" col2 ",  new   ColumnValu  e (
true ));

                var   request  =  new   PutRowRequ  est ( TableName
,  new   Condition ( RowExisten  ceExpectat  ion . IGNORE ),
                                                  primaryKey ,
attribute );

                putRowTask  List . Add ( TabeStoreC  lient .
PutRowAsyn  c ( request ));
            }

            // 各非同期呼び出しが結果を返すまで待機し、消費された  CU  を表示します。
             foreach  ( var   task   in   putRowTask  List )
            {
                task . Wait ();
                Console . WriteLine (" consumed   read :{ 0 },  write
:{ 1 }",  task . Result . ConsumedCa  pacityUnit . Read ,
                                    task . Result . ConsumedCa
pacityUnit . Write );
            }

            // 例外がスローされない場合、データは正常に挿入されます。
             Console . WriteLine (" Put   row   async   succeeded .");
        }
         catch  ( Exception   ex )
        {
            // 例外がスローされるとエラーメッセージが表示されます。
             Console . WriteLine (" Put   row   async   failed .
exception :{ 0 }",  ex . Message );
        }
```

:

各非同期呼び出しはスレッドを開始します。非常に多くの非同期呼び出しが連続して開始され、各呼び出しが長時間を消費すると、タイムアウトエラーが発生する可能性があります。

コードの詳細は、『PutRowAsync@GitHub』をご参照ください。

GetRow

指定された主キーに基づいて単一のデータ行を読み取ります。

API

```
        /// < summary >
        /// 指定されたプライマリキーに基づいて単一のデータ行を読み取ります。
```

```
        /// </ summary >
        /// < param   name =" request "> データクエリリクエスト </ param >
        ///< returns >  GetRow  から返された応答 </ returns >
         public   GetRowResp  onse   GetRow ( GetRowRequ  est
request );

        /// < summary >
        ///  GetRow  の非同期形式
        /// </ summary >
         public   Task < GetRowResp  onse >  GetRowAsyn  c (
GetRowRequ  est   request );
```

例 1

データ行を読み取ります。

```
        // 行のプライマリキーを定義します。これは、テーブル作成時に  TableMeta
で定義されたプライマリキーと一致している必要があります。
         PrimaryKey   primaryKey  =  new   PrimaryKey ();
         primaryKey . Add (" pk0 ",  new   ColumnValu  e ( 0 ));
         primaryKey . Add (" pk1 ",  new   ColumnValu  e (" abc "));

         try
        {
            // クエリリクエストオブジェクトを作成します。 列が指定されていない場
合は、行全体が読み取られます。
             var   request  =  new   GetRowRequ  est ( TableName ,
primaryKey );

            // データを問い合わせるために  GetRow  インターフェイスを呼び出しま
す。
             var   response  =  otsClient . GetRow ( request );

            // 行のデータを出力します。 出力データはここでは省略されています。 詳
しくは、 GitHub  のリンクをご参照ください。

            // 例外がスローされなければ、データは正常に読み取られます。
             Console . WriteLine (" Get   row   succeeded .");
        }
         catch  ( Exception   ex )
        {
            // 例外がスローされてエラーメッセージが表示されると、実行は失敗します
             Console . WriteLine (" Update   table   failed ,
exception :{ 0 }",  ex . Message );
        }
```

注 :

- データ行をクエリすると、システムはその行のすべての列のデータを返します。 columnsToGet パラメーターを使用して、指定した列のデータを読み取ることができます。 たとえば、col0 と col1 が columnsToGet に挿入された場合、システムは col0 と col1 のデータのみを返します。
- 条件付きフィルタがサポートされています。 たとえば、col0 の値が 24 より大きい場合にのみ結果を返すようにシステムを設定できます。

- columnsToGet パラメーターと Condition パラメーターの両方が使用されている場合、システムは最初に columnsToGet に基づいて結果を返し、次に Condition パラメーターに基づいて返された列をフィルタリングします。
- 指定された列が存在しない場合、PassIfMissing が次のアクションを決定します。

コードの詳細は、『*GetRow@GitHub*』をご参照ください。

例 2

フィルター機能を使用してデータ行を読み取ります。

次のコード例は、データを照会し、col0 と col1 のデータのみを返し、col0 の値が 24 であるという条件に基づいて col0 のデータをフィルタリングするようにシステムを設定します。

```
        // 行のプライマリキーを定義します。これは、テーブル作成時に TableMeta
で定義されたプライマリキーと一致している必要があります。
        PrimaryKey primaryKey = new PrimaryKey ();
        primaryKey . Add (" pk0 ", new ColumnValu e ( 0 ));
        primaryKey . Add (" pk1 ", new ColumnValu e (" abc "));

        var rowQueryCr iteria = new SingleRowQ ueryCriter ia
(" SampleTabl e ");
        rowQueryCr iteria . RowPrimary Key = primaryKey ;

       // 条件 1 は col0 の値が 5 であることです。
        var filter1 = new Relational Condition (" col0 ",
                    Relational Condition . CompareOpe rator . EQUAL
,
                    new ColumnValu e ( 5 ));

       // 条件 2 は col1 の値が ff ではないことです。
        var filter2 = new Relational Condition (" col1 ",
Relational Condition . CompareOpe rator . NOT_EQUAL , new
ColumnValu e (" ff "));

       // 条件 1 と条件 2 の組み合わせを OR 関係で構築します。
        var filter = new CompositeC ondition ( CompositeC
ondition . LogicOpera tor . OR );
        filter . AddConditi on ( filter1 );
        filter . AddConditi on ( filter2 );

        rowQueryCr iteria . Filter = filter ;

       // 条件 [ col0 , col1 ] に基づいてクエリする行を設定し、指定された条
件に基づいてクエリされたデータをフィルタリングします。
        rowQueryCr iteria . AddColumns ToGet (" col0 ");
        rowQueryCr iteria . AddColumns ToGet (" col1 ");

       // GetRowRequ est を構築します。
        var request = new GetRowRequ est ( rowQueryCr iteria
);

        try
       {
           // クエリデータ
            var response = otsClient . GetRow ( request );

           // データの出力または関連する論理演算の実行をします（省略）。
```

```
            // 例外がスローされなければ、実行は成功します。
             Console . WriteLine (" Get   row   with   filter
succeeded .");
        }
         catch  ( Exception   ex )
        {
            // 例外がスローされてエラーメッセージが表示されると、実行は失敗します
             Console . WriteLine (" Get   row   with   filter   failed
,  exception :{ 0 }",  ex . Message );
        }
```

コードの詳細は、『GetRowWithFilter@GitHub』をご参照ください。

UpdateRow

指定された行のデータを更新します。 指定された行が存在しない場合は、新しい行が追加されます。 それ以外の場合は、指定された列の値は、リクエストに合わせて、追加、変更、又は削除されます。

API

```
        /// < summary >
        /// 指定された行のデータを更新します。 指定された行が存在しない場合は、新
しい行が追加されます。 指定された行が存在する場合は、指定された列の値は、リクエスト
に合わせて、追加、変更、又は削除されます。
        /// </ summary >
        ///< param   name  = " request "> リクエストインスタンス </ param
>
         public   UpdateRowR  esponse   UpdateRow ( UpdateRowR  equest
  request );

        /// < summary >
        ///  UpdateRow  の非同期形式です。
        /// </ summary >
        /// < param   name =" request "></ param >
        /// < returns ></ returns >
         public   Task < UpdateRowR  esponse >  UpdateRowA  sync (
UpdateRowR  equest   request );
```

例

指定された行のデータを更新します。

```
        // 行のプライマリキーを定義します。これは、テーブル作成時に  TableMeta
 で定義されたプライマリキーと一致している必要があります。
        PrimaryKey   primaryKey  =  new   PrimaryKey ();
        primaryKey . Add (" pk0 ",  new   ColumnValu  e ( 0 ));
        primaryKey . Add (" pk1 ",  new   ColumnValu  e (" abc "));

        //  100  行の属性列を定義します。
        UpdateOfAt  tribute   attribute  =  new   UpdateOfAt  tribute
();
        attribute . AddAttribu  teColumnTo  Put (" col0 ",  new
ColumnValu  e ( 0 ));
```

```
        attribute . AddAttribu  teColumnTo  Put (" col1 ",  new
ColumnValu  e (" b ")); //  Change   the   original   value  ' a '
to  ' b '
        attribute . AddAttribu  teColumnTo  Put (" col2 ",  new
ColumnValu  e ( true ));

        try
        {
            // 指定された行が存在しない場合でもデータがまだ更新されていることを示
す  RowExisten  ceExpectat  ion . IGNORE  を使用して、行更新用の要求オブジェ
クトを作成します。
            var   request  =  new   UpdateRowR  equest ( TableName ,
new   Condition ( RowExisten  ceExpectat  ion . IGNORE ),
                                    primaryKey ,  attribute );
            //  UpdateRow  インターフェイスを呼び出します。
            otsClient . UpdateRow ( request );

            // 例外がスローされなければ、実行は成功します。
            Console . Writeline (" Update   row   succeeded .");
        }
        catch  ( Exception   ex )
        {
            // 例外がスローされてエラーメッセージが表示されると、実行は失敗します
            Console . WriteLine (" Update   row   failed ,  exception
:{ 0 }",  ex . Message );
        }
```

注：

行更新は条件付きステートメントの使用をサポートします。

コードの詳細は、『*UpdateRow@GitHub*』をご参照ください。

DeleteRow

API

```
        ( delete )/// < summary >( delete )
        /// 指定されたテーブル名とプライマリキーに基づいてデータ行を削除します。
        /// </ summary >
        /// < param   name  = " request "> リクエストインスタンス </ param
>
        /// < returns > レスポンスインスタンス </ returns />
        public   DeleteRowR  esponse   DeleteRow ( DeleteRowR  equest
  request );

        /// < summary >
        ///  DeleteRow  の非同期形式です。
        /// </ summary >
        public   Task < DeleteRowR  esponse >  DeleteRowA  sync (
DeleteRowR  equest   request );
```

例

データ行を削除します。

```
        // 削除する行のプライマリキー列の値は  0  と " abc " です。
```

```
            var   primaryKey  =  new   PrimaryKey ();
            primaryKey . Add (" pk0 ",  new   ColumnValu  e ( 0 ));
            primaryKey . Add (" pk1 ",  new   ColumnValu  e (" abc "));

            try
           {
               // 行が存在する場合にのみ行が削除されることを示す  Condition .
 EXPECT_EXI  ST  を使用して要求を作成します。
               var   deleteRowR  equest  =  new   DeleteRowR  equest ("
 SampleTabl  e ",  Condition . EXPECT_EXI  ST ,  primaryKey );

               //  DeleteRow  インターフェイスを呼び出して行を削除します。
               otsClient . DeleteRow ( deleteRowR  equest );

               // 例外がスローされない場合、行は正常に削除されます。
               Console . Writeline (" Delete   table   succeeded .");
           }
            catch  ( Exception   ex )
           {
               // 例外がスローされた場合、行は削除されず、エラーメッセージが表示されます。
               Console . WriteLine (" Delete   table   failed ,
 exception :{ 0 }",  ex . Message );
           }
```

注：

行の削除は条件付きステートメントの使用をサポートします。

コードの詳細は、『*DeleteRow@GitHub*』をご参照ください。

6.5 複数行操作

Table Store SDK には、BatchGetRow、BatchWriteRow、GetRange および GetRangeIterator の複数行操作インターフェイスがあります。

BatchGetRow

バッチは、1 つ以上のテーブルから複数のデータ行を読み取ります。 これは基本的に複数の GetRow 操作のセットです。 各操作は実行され、結果を返し、容量ユニットを個別に消費します。

多数の GetRow 操作の実行と比較して、BatchGetRow 操作はリクエストの応答時間を短縮し、データ読み取り速度を上げます。

API

```
           /// < summary >
           /// < para >   BatchGetRo  w  オペレーションは、基本的に複数の
 GetRow  オペレーションのセットです。</ para >
           /// < para > 各操作は実行され、結果を返し、容量ユニットを個別に消費します。 </ para >
           /// 多数の  GetRow  オペレーションの実行と比較して、 BatchGetRo  w
  オペレーションはリクエストの応答時間を短縮し、データ読み取り速度を上げます。
```

```
        /// </ summary >
        /// < param   name =" request "> リクエストインスタンス </ param >
        /// < returns > レスポンスインスタンス </ returns >
         public   BatchGetRo  wResponse   BatchGetRo  w ( BatchGetRo
 wRequest   request );

        /// < summary >
        ///  BatchGetRo  w  の非同期形式です。
        /// </ summary >
         public   Task < BatchGetRo  wResponse >  BatchGetRo  wAsync (
 BatchGetRo  wRequest   request );
```

例

10 データ行をバッチ読み取りします。

```
        //  10  行のプライマリキー値を含む、バッチ読み取り要求オブジェクトを作成
します。
         List < PrimaryKey >  primaryKey  s  =  new   List <
 PrimaryKey >();
         for  ( int   i  =  0 ;  i  <  10 ;  i ++)
        {
             PrimaryKey   primaryKey  =  new   PrimaryKey ();
             primaryKey . Add (" pk0 ",  new   ColumnValu  e ( i ));
             primaryKey . Add (" pk1 ",  new   ColumnValu  e (" abc
 "));
             primaryKey  s . Add ( primaryKey );
        }

         try
        {
             BatchGetRo  wRequest   request  =  new   BatchGetRo
 wRequest ();
             request . Add ( TableName ,  primaryKey  s );

            //  BatchGetRo  w  を呼び出して  10  データ行を読み取ります。
             var   response  =  otsClient . BatchGetRo  w ( request );
             var   tableRows  =  response . RowDataGro  upByTable ;
             var   rows  =  tableRows [ TableName ];

            // 行のデータを入力します。 入力データは省略しています。 詳しくは、
 GitHub  のリンクをご参照ください。

            //  BatchGetRo  w  は部分的に失敗する可能性があります。 したがっ
て、各行の状況を確認する必要があります。 詳しくは、 GitHub  のリンクをご参照くださ
い。
        }
         catch  ( Exception   ex )
        {
            // 例外がスローされてエラーメッセージが表示された場合、実行は失敗しま
す。
             Console . WriteLine (" Batch   get   row   failed ,
 exception :{ 0 }",  ex . Message );
        }
```

注:

BatchGetRow は、条件文を使用したフィルタをサポートしています。

コードの詳細は、『BatchGetRow@GitHub』をご参照ください。

BatchWriteRow

BatchWriteRow は、1 つ以上のテーブル内の複数のデータ行を挿入、変更、または削除します。 これは、複数の PutRow、UpdateRow および DeleteRow 操作のセットです。 各操作は実行され、独立して結果を戻し、容量ユニットを独立して消費します。

API

```
        /// < summary >
        /// < para > Batch   inserts ,  modifies ,  or   deletes
 several   data   rows   in   one   or   more   tables . </ para >
        /// < para >  BatchWrite  Row  オペレーションは、基本的に複数の
 PutRow 、 UpdateRow 、 DeleteRow  オペレーションのセットです。 各操作は実行さ
れ、結果を戻し、容量単位を個別に消費します。 </ para >
        /// < para > 多数の書き込み操作の実行と比較して、 BatchWrite  Row
  操作はリクエスト応答時間を短縮し、データ書き込み速度を増加させます。 </ para >
        /// </ summary >
        /// < param   name =" request ">リクエストインスタンス</ param >
        /// < returns >レスポンスインスタンス</ returns >
         public   BatchWrite  RowRespons  e   BatchWrite  Row (
 BatchWrite  RowRequest   request );

        /// < summary >
        ///  BatchWrite  Row  の非同期行です。
        /// </ summary >
        /// < param   name =" request "></ param >
        /// < returns ></ returns >
         public   Task < BatchWrite  RowRespons  e >  BatchWrite
 RowAsync ( BatchWrite  RowRequest   request );
```

例

100データ行を一括インポートします。

```
        //  100  行のプライマリキー値を含むバッチインポート要求オブジェクトを作成
します。
         var   request  =  new   BatchWrite  RowRequest ();
         var   rowChanges  =  new   RowChanges ();
         for  ( int   i  =  0 ;  i  <  100 ;  i ++)
        {
             PrimaryKey   primaryKey  =  new   PrimaryKey ();
             primaryKey . Add (" pk0 ",  new   ColumnValu  e ( i ));
             primaryKey . Add (" pk1 ",  new   ColumnValu  e (" abc
 "));

            //  100  行の属性列を定義します。
             UpdateOfAt  tribute   attribute  =  new   UpdateOfAt
 tribute ();
             attribute . AddAttribu  teColumnTo  Put (" col0 ",  new
 ColumnValu  e ( 0 ));
             attribute . AddAttribu  teColumnTo  Put (" col1 ",  new
 ColumnValu  e (" a "));
             attribute . AddAttribu  teColumnTo  Put (" col2 ",  new
 ColumnValu  e ( true ));
```

```
            rowChanges . AddUpdate ( new   Condition ( RowExisten
 ceExpectat  ion . IGNORE ),  primaryKey ,  attribute );
        }

         request . Add ( TableName ,  rowChanges );

         try
        {
            //  Call   BatchWrite  Row
             var   response  =  otsClient . BatchWrite  Row ( request
 );
             var   tableRows  =  response . TableRespo  nes ;
             var   rows  =  tableRows [ TableName ];

            //  BatchGetRo  w  の部分的な操作が失敗する可能性があります。したがって、各行の状況を確認する必要があります。 詳しくは、 GitHub  のリンクをご参照ください。
        }
        }  catch  ( Exception   e ) {
        {
            // 例外がスローされてエラーメッセージが表示された場合、実行は失敗します。
             Console . WriteLine (" Batch   put   row   failed ,
 exception :{ 0 }",  ex . Message );
        }
```

注 :

BatchWriteRow は、条件文を使用したフィルタをサポートしています。

コードの詳細は、『*BatchWriteRow@GitHub*』をご参照ください。

GetRange

指定したプライマリキー範囲のデータを読み込みます。

API

```
        /// < summary >
        /// 指定された範囲の複数の行からデータを取得します。
        /// </ summary >
        /// < param   name =" request ">リクエストインスタンス</ param >
        /// < returns >レスポンスインスタンス</ returns >
         public   GetRangeRe  sponse   GetRange ( GetRangeRe  quest
 request );

        /// < summary >
        ///  GetRange の非同期形式です。
        /// </ summary >
        /// < param   name =" request "></ param >
        /// < returns ></ returns >
         public   Task < GetRangeRe  sponse >  GetRangeAs  ync (
 GetRangeRe  quest   request );
```

例

指定された範囲のデータを読み込みます。

```
        // ( 0 ,  INF_MIN ) から ( 100 ,  INF_MAX ) までの範囲のすべての
行を読み込みます。
         var  inclusiveS  tartPrimar  yKey  =  new  PrimaryKey ();
         inclusiveS  tartPrimar  yKey . Add (" pk0 ",  new
 ColumnValu  e ( 0 ));
         inclusiveS  tartPrimar  yKey . Add (" pk1 ",  ColumnValu  e .
 INF_MIN );

         var  exclusiveE  ndPrimaryK  ey  =  new  PrimaryKey ();
         exclusiveE  ndPrimaryK  ey . Add (" pk0 ",  new  ColumnValu
 e ( 100 ));
         exclusiveE  ndPrimaryK  ey . Add (" pk1 ",  ColumnValu  e .
 INF_MAX );

         try
        {
            // 指定された範囲の読み込み要求オブジェクトを作成します。
             var  request  =  new  GetRangeRe  quest ( TableName ,
 GetRangeDi  rection . Forward ,
                              inclusiveS  tartPrimar  yKey ,
 exclusiveE  ndPrimaryK  ey );

             var  response  =  otsClient . GetRange ( request );

            // 部分的なデータしか返されない場合、読み取り操作を続行します。
             var  rows  =  response . RowDataLis  t ;
             var  nextStartP  rimaryKey  =  response . NextPrimar
 yKey ;
             while  ( nextStartP  rimaryKey  ! =  null ) {
            {
                 request  =  new  GetRangeRe  quest ( TableName ,
 GetRangeDi  rection . Forward ,
                                 nextStartP  rimaryKey ,  exclusiveE
 ndPrimaryK  ey );
                 response  =  otsClient . GetRange ( request );
                 nextStartP  rimaryKey  =  response . NextPrimar  yKey
 ;
                 foreach  ( RowDataFro  mGetRange  row  in
 response . RowDataLis  t )
                {
                     rows . Add ( row );
                }
            }

            // 行のデータを出力します。 出力データはここでは省略されています。 詳
しくは、 GitHub  のリンクを参照してください。

            // 例外がスローされなければ、実行は成功します。
             Console . WriteLine (" Get  range  succeeded ");
        }
        }  catch  ( Exception  e ) {
        {
            // 例外がスローされてエラーメッセージが表示されると、実行は失敗しま
す。
             Console . WriteLine (" Get  range  failed ,  exception
 :{ 0 }",  ex . Message );
        }
```

注：

GetRange は条件文を使ったフィルタをサポートしています。

コードの詳細は、『*GetRange@GitHub*』をご参照ください。

GetRangeIterator

指定された範囲の反復子を取得します。

API

```
        /// < summary >
        /// 指定された範囲の複数の行からデータを取得します。 システムは、各データ
行の反復処理に使用される反復子を返します。
        /// </ summary >
        /// < param   name =" request ">< see   cref =" GetIterato
 rRequest "/></ param >
        /// < returns > 反復子を戻します。 < see   cref =" RowDataFro
 mGetRange "/> </ returns >
         public   IEnumerabl  e < RowDataFro  mGetRange >  GetRangeIt
 erator ( GetIterato  rRequest   request );
```

例

反復子を取得します。

```
        // ( 0 ,  " a ") から ( 1000 ,  " xyz ") の範囲のすべての行を読み込
みます。
         PrimaryKey   inclusiveS  tartPrimar  yKey  =  new
 PrimaryKey ();
         inclusiveS  tartPrimar  yKey . Add (" pk0 ",  new
 ColumnValu  e ( 0 ));
         inclusiveS  tartPrimar  yKey . Add (" pk1 ",  new
 ColumnValu  e (" a "));

         PrimaryKey   exclusiveE  ndPrimaryK  ey  =  new   PrimaryKey
 ();
         exclusiveE  ndPrimaryK  ey . Add (" pk0 ",  new   ColumnValu
 e ( 1000 ));
         exclusiveE  ndPrimaryK  ey . Add (" pk1 ",  new   ColumnValu
 e (" xyz "));

        // 反復によって消費された  CU  を記録するために  CapacityUn  it  を構
築します。
         var   cu  =  new   CapacityUn  it ( 0 ,  0 );

         try
        {
            //  GetIterato  rRequest  を構築します。 フィルタ基準がサポートさ
れています。
             var   request  =  new   GetIterato  rRequest ( TableName
 ,  GetRangeDi  rection . Forward ,  inclusiveS  tartPrimar  yKey ,
                                                 exclusiveE
 ndPrimaryK  ey ,  cu );
```

```
            var   iterator   =  otsClient . GetRangeIt  erator (
request );
           // 走査メソッドでデータを読み込む反復子です。
            foreach  ( var   row   in   iterator )
           {
               //  Processing   logic
           }

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

注：

GetRangeIterator は条件文を使ったフィルタをサポートしています。

コードの詳細は、『*GetRangeIterator@GitHub*』をご参照ください。

6.6 エラー処理

方法

現在、Table Store C# SDKは `Exception` メソッドを採用して、エラーを処理します。 呼び出されたインターフェイスが例外をスローしない場合、操作は成功します。 例外がスローされると、操作は失敗します。

注：

BatchGetRow や BatchWriteRow などのバッチ操作インターフェイスは、システムが各行のステータスが成功したことを確認した場合にのみ正常に呼び出されます。

例外

Table Store C# SDK には、Exception から継承した OTSClientException と OTSServerException の 2 種類の例外があります。

- OTSClientException: 不正なパラメータ値や解析結果を返さないなど、SDK の内部例外を示します。

- OTSServerException: サーバーエラーメッセージの解析によって生成されたサーバーエラーを示します。 OTSServerException には、以下のコンポーネントがあります。
  - HttpStatusCode: 戻された HTTP コード、例えば 200 または 404。
  - ErrorCode: Table Store から返されたエラータイプの文字列。
  - ErrorMessage: Table Store から返されるエラーメッセージ文字列。
  - RequestId: リクエストを一意に識別する UUID。 問題が解決しない場合は、RequestId を保存してチケットを起票し、サポートセンターへお問い合わせください。

再試行

SDK でエラーが発生した場合、システムは操作を再試行します。 デフォルトでは、操作は最大 2 秒間隔で 3 回再試行されます。 詳しくは、Aliyun.OTS.Retry.DefaultRetryPolicy クラスをご参照ください。

OTSClientConfig で RetryPolicy を使用して再試行ポリシーをカスタマイズできます。

7 C++ SDK

7.1 はじめに

このドキュメントでは、Table Store C++ SDK のインストール方法と使用方法について説明します。 バージョン 4.0.0 以降に適用されます。 Table Store サービスに登録し、AccessKeyId と AccessKeySecret を作成したと仮定します。

- Alibaba Cloud Table Store サービスについての詳細は、「*Table Store プロダクトホームページ*」をご参照ください。
- AccessKeyId と AccessKeySecret をまだ作成していない場合は、*AccessKey コンソール*にアクセスして、 AccessKey を作成します。

ダウンロードアドレス: *GitHub*

7.2 インストール方法

C++ は特殊なので、このトピックで説明されている方法を使用してコンパイルし、`build / release / src / tablestore / core / impl / buildinfo . cpp` をバックアップのためにコピーすることを推奨します。 次に、C++ SDK のソースコードと `buildinfo . cpp` を自分のコードライブラリにコピーして、自分のコンパイルシステムを使ってコンパイルします。

コンパイルパラメーター

クライアント側のコードをコンパイルするときは、いくつかのコンパイラ動作を保証する必要があります。 これは、いくつかのコンパイラパラメーターが必要であることを意味します。

GCC コンパイラパラメーターは次のとおりです。

パラメーター	必須項目	説明
--std=gnu++03	はい	GCC 拡張子を持つ C++98 TR1 言語、つまり typeof をサポートします。
-pthread	はい	マルチスレッドプログラミングに必要なパラメーターです。 このパラメーターはコンパイルとリンクの両方に必要です。

パラメーター	必須項目	説明
-fwrapv	はい	整数オーバーフロー時に入れ替わります。 具体的には、符号なし整数オーバーフローの場合は 0、符号付き整数オーバーフローの場合は最小の負の数になります。 クライアントはこの動作に基づいてオーバーフローをチェックします。
-O2	推奨	最適化グレードです。 一般に、より高い最適化グレードは推奨されません。
-fsanitize=address and-fvar-tracking-assignments	推奨	gcc-4.9 以降のバージョンでは、libasan をサポートしています。これは、メモリ使用量のエラーをすばやく検出できる軽量の検出機能です。 Table Store の開発者がエラーを見つける必要がある場合は、エラーが再現されるように、これら2つのコンパイルエラーを取ります。 リンク時にも前のパラメーターを使用する必要があります。  : libasan と valgrind は互換性がありません。

環境依存性とコンパイル済みパッケージ

ここでは、Debian 8 システムを例として、コンパイル済みパッケージの生成方法を説明します。

1. `docker / debian8 / Dockerfile` ファイルを開いて、dockerfile を選択肢し、クライアントの環境依存関係をシステムにエクスポートします。 このメソッドは自動的にコードと環境の間の一貫性を保証します。

```
RUN apt-get install -y scons g++ libboost-all-dev
  protobuf-compiler libprotobuf-dev uuid-dev libssl-
dev
RUN apt-get install -y ca-certificates # HTTPS をサ
ポートするため。
```

SDK は以下に依存します。

- scons & gcc: コンパイルシステムです。
- boost
- uuid
- protobuf: serialization library
- openssl: signature, used to support HTTPS
- ca-certificates: used to support HTTPS only. Table Store の HTTP アドレスのみを使用する場合は、このライブラリをインストールしないことを選択できます。 より安全な HTTPS を使用することを推奨します。

2. これらのパッケージをインストールした後でクライアントをコンパイルできます。 クライアントのソースコードをダウンロードし、ソースコードディレクトリで `scons` を実行します。

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

これらのステップが完了すると、tar パッケージがコンパイルされます。 通常、パッケージ名は `scons` の最終出力にあります。 たとえば、Debian 8 システムのパッケージ名とパスは `build / release / pkg / aliyun - tablestore - cpp98 - sdk - 4 . 4 . 1 - debian8 . 9 - x86_64 . tar . gz` です。

- パッケージ名には、以下の要素が含まれています。
  - C++ version (C++98)
  - SDK version (4.4.1)
  - OS version (Debian 8.9)
  - OS architecture (x86_64)
- パッケージの内容は次のとおりです。

```
$ tar - tf build / release / pkg / aliyun - tablestore - cpp98 -
 sdk - 4 . 4 . 1 - debian8 . 9 - x86_64 . tar . gz
 version . ini
 lib / libtablest ore_core . so
 lib / libtablest ore_core_s tatic . a
 lib / libtablest ore_util . so
 lib / libtablest ore_util_s tatic . a
 include / tablestore / util / arithmetic . hpp
 include / tablestore / util / assert . hpp
 include / tablestore / util / foreach . hpp
```

```
include / tablestore / util / iterator . hpp
include / tablestore / util / logger . hpp
include / tablestore / util / logging . hpp
include / tablestore / util / mempiece . hpp
include / tablestore / util / mempool . hpp
include / tablestore / util / metaprogra  mming . hpp
include / tablestore / util / move . hpp
include / tablestore / util / optional . hpp
include / tablestore / util / prettyprin  t . hpp
include / tablestore / util / random . hpp
include / tablestore / util / result . hpp
include / tablestore / util / security . hpp
include / tablestore / util / seq_gen . hpp
include / tablestore / util / threading . hpp
include / tablestore / util / timestamp . hpp
include / tablestore / util / try . hpp
include / tablestore / util / assert . ipp
include / tablestore / util / iterator . ipp
include / tablestore / util / logging . ipp
include / tablestore / util / mempiece . ipp
include / tablestore / util / move . ipp
include / tablestore / util / prettyprin  t . ipp
include / tablestore / core / client . hpp
include / tablestore / core / error . hpp
include / tablestore / core / range_iter  ator . hpp
include / tablestore / core / retry . hpp
include / tablestore / core / types . hpp
```

このパッケージには現在、次の要素が含まれています。

- バージョンファイル: `version . ini`
- ライブラリファイル: `lib /` 以下のすべてのファイル。 `libtablest  ore_core_s  tatic . a` は、 `libtablest  ore_util_s  tatic . a` に依存しますが、Moments ライブラリに類似しています。
- ヘッダファイル: `include /` 以下のすべてのファイル。

7.3 初期化

Table Store クライアントは、テーブル、単一のデータ行、複数のデータ行などを操作するための一連の方法を呼び出し元に提供します。

エンドポイントの決定

エンドポイントは、各リージョンの Alibaba Cloud Table Store サービスのドメイン名アドレスです。 現在サポートされている形式は次のとおりです。

例	説明
`http :// sun . cn - hangzhou . ots  . aliyuncs . com`	HTTP を使用した杭州の sun インスタンスへのパブリックネットワークアクセスです。

例	説明
`https :// sun . cn - hangzhou . ots . aliyuncs . com`	HTTPS を使用して杭州の sun インスタンスへのパブリックネットワークアクセスです。

注：

- Table Store は、パブリックおよびプライベートネットワークアクセスをサポートしています。
- *Table Store コンソール*にログインし、インスタンスの詳細ページを確認することができます。 インスタンスアクセス URL はそのインスタンスのエンドポイントです。

AccessKey の設定

Alibaba Cloud Table Store サービスにアクセスするには、署名認証用の有効なAccessKey が必要です。 AccessKey を構成する以下の 3 つの方法が現在サポートされています。

- プライマリアカウントの AccessKeyId と AccessKeySecret を使用します。 手順は次のとおりです。
  1. *Alibaba Cloud アカウント*を登録します。
  2. *AccessKey コンソール*にログインし、 AccessKeyId と AccessKeySecret を作成します。
- Table Store へのアクセスを許可されている RAM ユーザーの AccessKeyId および AccesskeySecret を使用します。 手順は次のとおりです。
  1. プライマリアカウントを使用して *リソースアクセス管理 (RAM)* にアクセスして、新しい RAM ユーザーを作成するか、既存のものを使用します。
  2. プライマリアカウントを使用して、RAM ユーザーに Table Store へのアクセスを許可します。
  3. その後、認証された RAM ユーザーの AccessKeyId と AccessKeySecret を使用して Table Store にアクセスできます。
- 一時アクセスには STS トークンを使用します。 手順は次のとおりです。
  1. アプリケーションサーバーは RAM/STS サービスにアクセスし、一時的な AccesskeyId、AccesskeySecret およびトークンを取得して、それらをユーザーに送信します。
  2. 一時キーを使用して Table Store サービスにアクセスできます。

クライアントを作成する

Table Store SDK を使用するときは、最初にクライアントを構築し、次にそのインターフェイスを呼び出して Table Store サービスにアクセスする必要があります。 クライアントインターフェイスは、Table Store が提供する RESTful API と一致しています。

クライアントの種類

Table Store C++ SDK には、SyncClient と AsyncClient の2種類のクライアントがあります。 SyncClient と AsyncClient はそれぞれ同期インターフェイスと非同期インターフェイスに対応します。

- 同期インターフェイス: 呼び出しが完了するとすぐに要求が実行されるので、非常に便利な方法です。 同期インターフェイスを使用して、さまざまな Table Store 機能について詳しく知ることができます。
- 非同期インターフェイス: これらは同期インターフェイスよりも高い柔軟性を提供します。 高いパフォーマンス要件がある場合は、非同期インターフェイスの使用とマルチスレッドの使用との間で妥協点を見つけることができます。

> 注:
> SyncClient と AsyncClient はどちらもスレッドセーフであり、スレッドと接続リソースを自動的かつ内部的に管理できます。 スレッドやリクエストごとにクライアントを作成する必要はありません。 代わりに、グローバルクライアントを作成します。

同期インターフェイス

- 直接作成する (Table Store エンドポイントを使用してクライアントを作成する)

```
Endpoint   ep (" YourEndpoi  nt ", " YourInstan  ce ");
Credential   cr (" AccessKeyI  d ", " AccessKeyS  ecret ");
ClientOpti  ons   opts ;
SyncClient *  client  =  NULL ;
Optional < OTSError >  res  =  SyncClient :: create ( client ,  ep
,  cr ,  opts );
```

> 注:
> プライマリアカウントの AccessKey を使用して Table Store にアクセスしないでください。 一時的なトークンまたはサブアカウントの AccessKey を使用することを推奨します。 一時的 STS トークンを使用する場合は、前述のコードの資格情報オブジェクトを

次のように変更する必要があります。 `Credential cr (" AccessKeyI d ", " AccessKeyS ecret ", " SecurityTo ken ");`

ClientOptions に含まれる設定項目は以下のとおりです。 デフォルト値を使用することも、自分で定義することもできます。

mMaxConnections	合計接続数の最大数および同時要求の最大数。 永続的な接続は、SDK と Table Store サービスの間で維持されます。 新しい要求はそれぞれ、ランダムに選択されたアイドル接続によって送信されます。
mConnectTimeout	接続タイムアウト。 DNS 解決時間を考慮して、推奨される接続タイムアウトは少なくとも 10 秒です。
mRequestTimeout	タイムアウト時間を要求します。
mRetryStrategy	再試行の戦略。 デフォルトでは、再試行戦略は失敗したべき等倍要求を 10 秒以内に再試行します。 独自の再試行戦略を定義することができます。
mLogger	ロガー。 デフォルトでは、ロガーはログを標準エラーにエクスポートします。 独自のロガーを定義することを推奨します。
mActors	スレッドプール。 コールバックを実行するために使用されたるレッドプールです。 デフォルトは 10 スレッドです。

- AsyncClient から作成

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

非同期インターフェイス

非同期インターフェイスクライアントの作成方法と使用方法の詳細については、「非同期インターフェイス」をご参照ください。

マルチスレッド

マルチスレッドがサポートされています。 マルチスレッド化する場合、1 つのクライアントオブジェクトを共有することを推奨します。

7.4 テーブルレベルの操作

SDK は、CreateTable、ListTable、UpdateTable、DescribeTable、DeleteTable などのテーブル操作インターフェイスを提供します。


注：
次の動作例は同期インターフェイスに基づいています。 非同期インターフェイスの操作方法について詳しくは、「非同期インターフェイス」をご参照ください。

テーブルを作成する (CreateTable)

作成するテーブルの名前とプライマリキーを常に指定する必要があります。 プライマリキーには 1 つから 4 つのプライマリキー列があり、各列には名前と型があります。

Table Store テーブルでは、プライマリキー列に自動増分を設定できます。 詳細については、「プライマリキー列の自動インクリメント」をご参照ください。

例

この例では、テーブル名は simple_create_delete_table です。 プライマリキーには、pkey という名前の 1 つのプライマリキー列しかなく、整数型 (kPKT_Integer) です。

```
 CreateTabl  eRequest   req ;
{
    // テーブルの不変の設定
     TableMeta &  meta  =  req . mutableMet  a ();
     meta . mutableTab  leName () = " simple_cre  ate_delete  _table
 ";
    {
        // 厳密に  1  つの整数プライマリキー列を持ちます。
         Schema &  schema  =  meta . mutableSch  ema ();
         PrimaryKey  ColumnSche  ma &  pkColSchem  a  =  schema .
 append ();
         pkColSchem  a . mutableNam  e () = " pkey ";
         pkColSchem  a . mutableTyp  e () =  kPKT_Integ  er ;
    }
}
 CreateTabl  eResponse   resp ;
 Optional < OTSError >  res  =  client . createTabl  e ( resp ,  req
 );
```


注：
詳細なコードは *createTable@GitHub* にあります。

可変パラメーター

データテーブルにはいくつかの可変パラメーターを設定できます。 可変パラメーターは、テーブルの作成時に設定することも、テーブルのアップデートを使用して変更することもできます。

可変パラメーターには次のものがあります。

可変パラメーター	名前	デフォルト値
mutableTimeToLive()	*Time to live*	-1 (データの有効期限が切れないことを意味します)
mutableMaxVersions()	*Max Versions*	1
mutableMaxTimeDeviation()	*Max Version Offset*	86,400s (または 1 日)
mutableReservedThroughput()	*Reserved read/write throughput*	0 (すべての読み書きは従量課金制)

以下は、テーブルを作成するときに予約済み読み書きスループットを設定する例です。

```
 CreateTabl  eRequest   req ;
{
    // テーブルの不変の設定
     TableMeta &  meta  =  req . mutableMet  a ();
     meta . mutableTab  leName () = " create_tab  le_with_re
 served_thr  oughput ";
    {
        // 厳密に 1 つの整数プライマリキー列を持ちます。
         Schema &  schema  =  meta . mutableSch  ema ();
         PrimaryKey  ColumnSche  ma &  pkColSchem  a  =  schema .
 append ();
         pkColSchem  a . mutableNam  e () = " pkey ";
         pkColSchem  a . mutableTyp  e () =  kPKT_Integ  er ;
    }
}
{
     TableOptio  ns &  opts  =  req . mutableOpt  ions ();
    {
        // 予約済み読み取り容量単位は  0 、予約書き込み容量単位は  1  です。
         CapacityUn  it   cu ( 0 ,  1 );
         opts . mutableRes  ervedThrou  ghput (). reset ( util :: move
 ( cu ));
    }
}
 CreateTabl  eResponse   resp ;
 Optional < OTSError >  res  =  client . createTabl  e ( resp ,  req
 );
```

テーブル名を一覧表示する (ListTable)

現在のインスタンスで作成されたすべてのテーブルの名前を取得するために使用されます。

API

`SyncClient :: listTable ()` をインスタンス下にあるすべてのテーブルをリストするために使用します。

```
 SyncClient *  client  = ... ;
 ListTableR  equest   req ;
```

```
ListTableR  esponse   resp ;
Optional < OTSError >  res  =  client -> listTable ( resp ,  req );
```

例

インスタンスの下にあるすべてのテーブルの名前を取得するために使用されます。

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

注：

詳細なコードは、『*listTable@GitHub*』をご参照ください。

テーブルを更新する (UpdateTable)

指定されたテーブルの変数パラメーターを更新します。

例

予約済みスループットを更新します。

```
UpdateTabl  eRequest   req ;
req . mutableTab  le () = " YourTable ";
UpdateTabl  eResponse   resp ;
{
    TableOptio  ns &  opts  =  req . mutableOpt  ions ();
    {
        // 予約済み読み取り容量単位は  0 、予約済み書き込み容量単位は  1  です。
        CapacityUn  it   cu ( 0 ,  1 );
        opts . mutableRes  ervedThrou  ghput (). reset ( util :: move
( cu ));
    }
}
Optional < OTSError >  res  =  client . updateTabl  e ( resp ,  req
);
```

注：

詳細なコードは、『*updateTable@GitHub*』をご参照ください。

テーブル情報を照会する (DescribeTable)

以下のテーブル情報は、 `describeTa  ble ()` インターフェイスを介して照会できます。

情報アイテム	説明
テーブルの状態	以下を含みます。 ・ kTS_Active: テーブルは通常の読み書きサービスを提供できます。 ・ kTS_Inactive: テーブルは読み書きできませんが、テーブルデータは予約されています。 このステータスは通常、プライマリバックアップテーブルの切り替え中に発生します。 ・ - kTS_Loading: テーブルは作成中です。テーブルは読み書きできません。 ・ kTS_Unloading: テーブルは削除中です。テーブルは読み書きできません。 ・ kTS_Updating: 変数テーブルパラメーターは更新中です。 テーブルは読み書きできません。
テーブルメタ	「テーブルの作成」をご参照ください。
可変テーブルパラメーター	「可変パラメーター」をご参照ください。
シャード間でポイントを分割する	Table Store テーブルは、水平方向にいくつかのシャードに分割されています。 分割ポイントはこのインターフェイスを通して得られます。  注 : Table Store は、負荷に応じてバックグラウンドで自動的に分割およびマージできます。そのため、このインターフェイスが受け取る分割ポイントは過去のシャードを反映することが保証されていますが、現在進行中のものと必ずしも一致するわけではありません。

例

```
DescribeTa  bleRequest   req ;
req . mutableTab  le () = " YourTable ";
DescribeTa  bleRespons  e   resp ;
Optional < OTSError >  res  =  client . describeTa  ble ( resp ,  req
);
```

 注 :

詳細なコードは、『*describeTable@GitHub*』をご参照ください。

テーブルを削除する (DeleteTable)

このインスタンスの下にある指定されたテーブルを削除します。 唯一の要件は、テーブル名を指定することです。

例

```
DeleteTabl eRequest req ;
req . mutableTab le () = " YourTable ";
DeleteTabl eResponse resp ;
Optional < OTSError > res = client . deleteTabl e ( resp , req
);
```

注 :

詳細なコードは、『*deleteTable@GitHub*』をご参照ください。

7.5 単一行操作

Table Store は、PutRow、GetRow、UpdateRow、DeleteRow などの単一行操作用のインターフェイスを提供します。

注 :

次の動作例は同期インターフェイスに基づいています。 非同期インターフェイスの操作方法について詳しくは、「非同期インターフェイス」をご参照ください。

データ行を入れる (PutRow)

PutRow インターフェイスは、データ行を挿入するために使用されます。 行が存在しない場合は、この操作によって行が挿入されます。 行が存在する場合、この操作はそれを上書きします。これは、元の行のすべてのバージョンのすべての列と列値が削除されることを意味します。

例

```
PutRowRequ est req ;
{
    RowPutChan ge & chg = req . mutableRow Change ();
    chg . mutableTab le () = " YourTable ";
    {
        // 配置する行の主キーを設定します。
        PrimaryKey & pkey = chg . mutablePri maryKey ();
        pkey . append () = PrimaryKey Column (
            " pkey ",
            PrimaryKey Value :: toStr (" pkey - value "));
    }
    {
        // 配置する行の属性を設定します。
        IVector < Attribute >& attrs = chg . mutableAtt ributes
 ();
        attrs . append () = Attribute (
```

```
            " attr ",
             AttributeV  alue :: toInteger ( 123 ));
    }
}
 PutRowResp  onse   resp ;
 Optional < OTSError >  res  =  client . putRow ( resp ,  req );
```

行のデータを取得する (GetRow)

GetRow インターフェイスは、単一行のデータを読み込むために使用されます。

行のテーブル名とプライマリキーを指定してください。 2 つの考えられる読み取り結果は次のとおりです。

- この行が存在する場合、GetRowResponse オブジェクトはその行の各プラマリキー列と属性列を返します。
- この行が存在しない場合、GetRowResponse オブジェクトに行は含まれず、エラーも報告されません。

例

```
 GetRowRequ  est   req ;
{
     PointQuery  Criterion &  query  =  req . mutableQue  ryCriterio
 n ();
     query . mutableTab  le () = “ YourTable ”;
    {
         PrimaryKey &  pkey  =  query . mutablePri  maryKey ();
         pkey . append () =  PrimaryKey  Column (
            " pkey ",
             PrimaryKey  Value :: toStr (" some_key ")); // テーブルに
は、文字列型のプライマリキーが  1  つしかないと仮定します。
    }
     query . mutableMax  Versions (). reset ( 1 );
}
 GetRowResp  onse   resp ;
 Optional < OTSError >  res  =  client . getRow ( resp ,  req );
```

行のデータを更新する (UpdateRow)

UpdateRow インターフェイスは、データ行を更新するために使用されます。 指定された行が存在しない場合は、新しい行が追加されます。

更新操作には、以下の 4 つのシナリオが考えられます。

- バージョン番号を指定せずに列値を書き込みます。 Table Store サービスは自動的にバージョン番号を提示して、バージョン番号が確実に増分されるようにします。
- 指定されたバージョン番号で列値を書き込みます。 列にこのバージョンの列値がない場合は、データが挿入されます。 列にある場合は、元の値が上書きされます。
- 指定されたバージョンの列値を削除します。

・ 列全体のすべてのバージョンの列値を削除します。

例

```
 UpdateRowR  equest   req ;
{
     RowUpdateC  hange &  chg  =  req . mutableRow  Change ();
     chg . mutableTab  le () = " YourTable ";
    {
        // 配置する行のプライマリキーを設定します。
         PrimaryKey &  pkey  =  chg . mutablePri  maryKey ();
         pkey . append () =  PrimaryKey  Column (
            " pkey ",
             PrimaryKey  Value :: toStr (" pkey "));
    }
    {
        // バージョンを指定せずに値を挿入します。
         RowUpdateC  hange :: Update &  up  =  chg . mutableUpd  ates
 (). append ();
         up . mutableTyp  e () =  RowUpdateC  hange :: Update :: kPut
 ;
         up . mutableAtt  rName () = " attr0 ";
         up . mutableAtt  rValue (). reset ( AttributeV  alue :: toStr
 (" new   value   without   specifying   version "));
    }
    {
        // バージョンとともに値を挿入します。
         RowUpdateC  hange :: Update &  up  =  chg . mutableUpd  ates
 (). append ();
         up . mutableTyp  e () =  RowUpdateC  hange :: Update :: kPut
 ;
         up . mutableAtt  rName () = " attr1 ";
         up . mutableAtt  rValue (). reset ( AttributeV  alue :: toStr
 (" new   value   with   version "));
         up . mutableTim  estamp (). reset ( UtcTime :: now ());
    }
    {
        // 特定のバージョンの値を削除します。
         RowUpdateC  hange :: Update &  up  =  chg . mutableUpd  ates
 (). append ();
         up . mutableTyp  e () =  RowUpdateC  hange :: Update ::
 kDelete ;
         up . mutableAtt  rName () = " attr2 ";
         up . mutableTim  estamp (). reset ( UtcTime :: now ());
    }
    {
        // 属性列のすべての値を削除します。
         RowUpdateC  hange :: Update &  up  =  chg . mutableUpd  ates
 (). append ();
         up . mutableTyp  e () =  RowUpdateC  hange :: Update ::
 kDeleteAll ;
         up . mutableAtt  rName () = " attr3 ";
    }
}
 UpdateRowR  esponse   resp ;
 Optional < OTSError >  res  =  client . updateRow ( resp ,  req );
```

行のデータを削除する (DeleteRow)

DeleteRow インターフェイスは、行を削除するために使用されます。 この行が存在するかどうかにかかわらず、エラーは報告されません。

例

```
 DeleteRowR  equest   req ;
{
     RowDeleteC  hange &  chg  =  req . mutableRow  Change ();
     chg . mutableTab  le () = " YourTable ";
    {
        // 削除する行のプライマリキーを設定します。
         PrimaryKey &  pkey  =  chg . mutablePri  maryKey ();
         pkey . append () =  PrimaryKey  Column (
            " pkey ",
             PrimaryKey  Value :: toInteger ( 1 ));
    }
}
 DeleteRowR  esponse   resp ;
 Optional < OTSError >  res  =  client . deleteRow ( resp ,  req );
```

7.6 複数行操作

Table Store は、BatchGetRow、BatchWriteRow、GetRange などの複数行操作用のインターフェイスを提供します。

 注 :
次の動作例は同期インターフェイスに基づいています。 非同期インターフェイスの操作方法について詳しくは、「非同期インターフェイス」をご参照ください。

BatchGetRow

バッチは、1 つ以上のテーブルから複数行のデータを読み取ります。

BatchGetRow は、複数の GetRow 操作のセットと見なすことができます。 各操作の実行、結果の返し、およびサービスキャパシティーユニットの計算はすべて独立して行われます。 多数の GetRow 操作を実行する場合と比較して、BatchGetRow を使用すると、要求の応答時間を効果的に短縮し、データ読み取り速度を向上させることができます。

BatchGetRow を使用すると、2つの考えられるエラーが発生する可能性があります。

- ネットワークエラーなどの全体的な要求エラーがある場合。 これらのエラーは batchGetRow() の戻り値に格納されます。

- 全体的な要求エラーはなく、個々の行にエラーが含まれている場合。 これらのエラーは対応する行の結果に格納されます。

例

```
 BatchGetRo  wRequest   req ;
{
     MultiPoint  QueryCrite  rion &  criterion  =  req . mutableCri
 teria (). append ();
     IVector < MultiPoint  QueryCrite  rion :: RowKey >&  rowkeys  =
 criterion . mutableRow  Keys ();
    {
         MultiPoint  QueryCrite  rion :: RowKey &  exist  =  rowkeys .
 append ();
         exist . mutableGet (). append () =  PrimaryKey  Column (
            " pkey ",
             PrimaryKey  Value :: toStr (" some   key "));
         exist . mutableUse  rData () = & userDataFo  rSomeKey ;
    }
    {
         MultiPoint  QueryCrite  rion :: RowKey &  exist  =  rowkeys .
 append ();
         exist . mutableGet (). append () =  PrimaryKey  Column (
            " pkey ",
             PrimaryKey  Value :: toStr (" another   key "));
         exist . mutableUse  rData () = & userDataFo  rAnotherKe  y ;
    }
     criterion . mutableTab  le () = " YourTable ";
     criterion . mutableMax  Versions (). reset ( 1 );
}
 BatchGetRo  wResponse   resp ;
 Optional < OTSError >  res  =  client . batchGetRo  w ( resp ,  req
 );
```

注：

詳細なコードは、『batchGetRowGitHub』をご参照ください。

BatchWriteRow

1 つ以上のテーブルから複数行のデータを一括して挿入、変更、または削除します。

BatchWriteRow は、複数の PutRow、UpdateRow および DeleteRow 操作のセットと見なすことができます。 各操作の実行、結果の戻し、およびサービスキャパシティーユニットの計算はすべて独立して行われます。

BatchWriteRow を使用すると、2 つの考えられるエラーが発生する可能性があります。

- ネットワークタイムアウトなどの全体的な要求エラーの場合。 これらのエラーは batchWriteRow() の戻り値に格納されています。
- 無効な主キー値など、個々の行のエラーの場合。 これらのエラーは BatchWriteRowResponse のすべての行に格納されています。

例

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

 BatchWrite  RowRequest   req ;
{
    // 行を挿入します。
     BatchWrite  RowRequest :: Put &  put  =  req . mutablePut  s ().
 append ();
     put . mutableUse  rData () =  kPutRow ;
     put . mutableGet (). mutableTab  le () =  kTableName ;
     PrimaryKey &  pkey  =  put . mutableGet (). mutablePri  maryKey
 ();
     pkey . append () =  PrimaryKey  Column (
        " pkey ",
         PrimaryKey  Value :: toStr (" row   to   put "));
}
{
    // 行を更新します。
     BatchWrite  RowRequest :: Update &  update  =  req . mutableUpd
 ates (). append ();
     update . mutableUse  rData () =  kUpdateRow ;
     update . mutableGet (). mutableTab  le () =  kTableName ;
     PrimaryKey &  pkey  =  update . mutableGet (). mutablePri
 maryKey ();
     pkey . append () =  PrimaryKey  Column (
        " pkey ",
         PrimaryKey  Value :: toStr (" row   to   update "));
     RowUpdateC  hange :: Update &  attr  =  update . mutableGet ().
 mutableUpd  ates (). append ();
     attr . mutableTyp  e () =  RowUpdateC  hange :: Update :: kPut ;
     attr . mutableAtt  rName () = " attr0 ";
     attr . mutableAtt  rValue (). reset ( AttributeV  alue :: toStr
 (" some   value "));
}
{
    // 行を削除します。
     BatchWrite  RowRequest :: Delete &  del  =  req . mutableDel
 etes (). append ();
     del . mutableUse  rData () =  kDeleteRow ;
     del . mutableGet (). mutableTab  le () =  kTableName ;
     PrimaryKey &  pkey  =  del . mutableGet (). mutablePri  maryKey
 ();
     pkey . append () =  PrimaryKey  Column (
        " pkey ",
         PrimaryKey  Value :: toStr (" row   to   delete "));
}
 BatchWrite  RowRespons  e   resp ;
 Optional < OTSError >  res  =  client . batchWrite  Row ( resp ,  req
 );
```

注:

詳細なコードは、『*batchWriteRowGitHub*』をご参照ください。

GetRange

指定された主キー範囲内のデータを読み取ります。

`RangeItera tor` を推奨します。 RangeIterator を構築するには、

- *AsyncClient* を提供する必要があります。
- RangeQueryCriterion は PointQueryCriterion に似ていますが、RangeQueryCriterion は以下も必要とします。
  - 範囲の始点 (inclusive) と終点 (exclusive) を設定します。 通常のプライマリキー値に加えて、2 つの特別な値、「負の無限大」 (すべての通常のプライマリキー列の値より厳密に小さい値) および「正の無限大」 (すべてのプライマリキーの列値より厳密に大きい値) も使用できます。
  - 読み値をポジティブシーケンス (小さい値から大きい値へ) またはリバースシーケンス (大きい値から小さい値へ) に設定します。 デフォルト設定はポジティブシーケンスです。 ポジティブシーケンスで読むときは、始点は終点より小さくなければなりません。 リバースシーケンスでは、始点は終点より大きくなければなりません。

RangeIterator オブジェクトは、3 つのインターフェイスを提供する反復子です。

- `moveNext ()` は、RangeIterator オブジェクトを次の行に移動します。 新しく構築された RangeIterator オブジェクトは `moveNext ()` を呼び出して、値を取得する必要があります。 データの読み取りに失敗した場合は、 `moveNext ()` は、戻り値にエラーを表示します。
- `valid ()` は、RangeIterator オブジェクトが範囲の終点に達したかどうかを示します。
- `valid ()` が true の場合、行オブジェクトは `get ()` を使用して読み取ることができます。 `get ()` によって戻される行オブジェクトの内容を削除すると、メモリの複製を回避できます。 コンテンツを削除した後、すぐ後に続く `get ()` は削除されたコンテンツを返します。

例

```
 RangeQuery  Criterion   query ;
 query . mutableTab  le () = " YourTable ";
 query . mutableMax  Versions (). reset ( 1 );
{
     PrimaryKey &  start  =  query . mutableInc  lusiveStar  t ();
     start . append () =  PrimaryKey  Column (
        " pkey ",
         PrimaryKey  Value :: toInfMin ());
}
{
     PrimaryKey &  end  =  query . mutableExc  lusiveEnd ();
     end . append () =  PrimaryKey  Column (
```

```
        " pkey ",
         PrimaryKey  Value :: toInfMax ());
}
 auto_ptr < AsyncClien  t >  aclient ( AsyncClien  t :: create (
 client ));
 RangeItera  tor   iter (* aclient ,  query );
 for (;;) {
     Optional < OTSError >  err  =  iter . moveNext ();
     if  ( err . present ()) {
        //  err  で何かをします。
         abort ();
    }
     if  (!  iter . valid ()) {
         break ;
    }
     Row &  row  =  iter . get ();
    // 行で何かをします。
}
```

注：

詳細なコードは、『*batchRangeRowGitHub*』にあります。

指定された列を読み取る

Table Store は、無限の行幅をサポートしています。 通常、指定された列を読み取るだけで十分なので、行全体を読み取る必要はありません。 クエリ基準 (PointQueryCriterion、MultiPointQueryCriterion および RangeQueryCriterion) は、`mutableCol umnsToGet ()` を提供して、読み込む必要がある列を指定します。 それらは属性列またはプライマリキー列にすることができます。 値が空の場合は、行全体が読み取られます。

指定された列が読み取り行に存在しない場合、その列は返された結果から欠落しています。 Table Store はプレースホルダを提供しません。

GetRange では、指定されたすべての列が属性列であり、範囲内の行が指定されたすべての列を見逃した場合、この行は結果に表示されません。 この行に注目する必要がある場合は、指定した列にプライマリキー列を追加してください。

指定されたバージョンを読む

各属性列には複数のバージョンを含めることができ、各バージョン番号 (タイムスタンプ) は列の値に対応します。 読み取るときに、バージョンの数 `mutableMax Versions ()`) とバージョンの範囲 `mutableTim eRange ()` を定義して、読み取ることができます。

maxversions とバージョンの範囲の少なくとも一方を定義する必要があります。

- バージョン数のみを定義した場合は、すべてのバージョンで定義されているデータ量が最新のものから順に返されます。

- バージョンの範囲のみを定義した場合は、この範囲内のすべてのデータが返されます。
- バージョン数とバージョン範囲の両方を定義した場合は、バージョンの範囲内で定義されているデータ量が最新のものから古いものの順に返されます。

条件付き書き込み

条件付き書き込みでは、行を書き込む前に特定の条件を確認して満たす必要があります。 Table Store は、条件チェックと書き込みの原子性を保証します。

Table Store は行の存在条件と列の値条件をサポートします。

- 行の存在条件は、次の種類に分類できます。
  - Ignore: 行が存在するかどうかにかかわらず、これがデフォルト設定です。
  - ExpectExist: 存在すると予想します。 行が存在する場合は書き込みます。
  - ExpectNotExist: 存在しないと予想します。 行が存在しない場合は書き込みます。
- 列値の条件は、「フィルター」と同じです。

7.7 フィルター

Table Store は、サービス側で読み取り結果を再フィルタリングして、ネットワーク上で送信されるデータ量を制限することができます。

ツリー構造フィルタの内部ノードは論理演算子 (CompositeColumnCondition) で、そのリーフノードは比較演算子 (SingleColumnCondition) です。

- CompositeColumnCondition は、 `AND` 、 `OR` および `NOT` をサポートします。 `AND` および `OR` に複数のサブノードをマウントできます。 `NOT` にマウントできるノードは1つだけです。
- SingleColumnCondition は、6つすべての比較条件 (EQUAL TO、NOT EQUAL TO、GREATER THAN、LESS THAN、GREATER THAN OR EQUAL TO、LESS THAN OR EQUAL TO) をすべてサポートします。
- 各 SingleColumnCondition オブジェクトは、列 (プライマリキー列にすることができます) と定数との比較をサポートします。 2 つの列または 2 つの定数を比較することはサポートされていません。

- SingleColumnCondition の latestVersionOnly パラメーターは、列値の複数のバージョンを比較に含める方法を制御します。 デフォルト値は "true" です。
  - true の場合、指定されたバージョンの範囲内にある最新の列値のみが比較に含まれます。(これらの列値は比較にのみ関係します。 フィルタがこの行を受け入れる場合でも、他のバージョンの列値が返されます。)
  - false の場合、いずれかの列値が条件を満たすと、ノードはそれを true と見なします。
- SingleColumnCondition の passIfMissing パラメーターは、列値が欠落している場合にノードが考慮する値を制御します。 デフォルト値は "false" です。

フィルタは他の条件付きフィルタの結果に作用します。 したがって、フィルター内の列が指定された読み取り列に指定されていない場合、またはバージョンの範囲内に列値が存在しない場合、フィルターはこの列が欠落していると見なします。

例

次の例では、プライマリキー列 pkey が 1 より大きく、属性列 attr が "A" に等しいという条件を満たすすべての行をテーブルから除外します。

```
 RangeQuery  Criterion   query ;
 query . mutableTab  le () =  kTableName ;
 query . mutableMax  Versions (). reset ( 1 );
{
   PrimaryKey &  start  =  query . mutableInc  lusiveStar  t ();
   start . append () =  PrimaryKey  Column (
      " pkey ",
       PrimaryKey  Value :: toInfMin ());
}
{
   PrimaryKey &  end  =  query . mutableExc  lusiveEnd ();
   end . append () =  PrimaryKey  Column (
      " pkey ",
       PrimaryKey  Value :: toInfMax ());
}
{
  // フィルタを設定します。
   shared_ptr < ColumnCond  ition >  pkeyCond (
       new   SingleColu  mnConditio  n (
          " pkey ",
           SingleColu  mnConditio  n :: kLarger ,
           AttributeV  alue :: toInteger ( 1 )));
   shared_ptr < ColumnCond  ition >  attrCond (
       new   SingleColu  mnConditio  n (
          " attr ",
           SingleColu  mnConditio  n :: kEqual ,
           AttributeV  alue :: toStr (" A ")));
   shared_ptr < CompositeC  olumnCondi  tion >  top ( new
 CompositeC  olumnCondi  tion ());
   top -> mutableOp () =  CompositeC  olumnCondi  tion :: kAnd ;
   top -> mutableChi  ldren (). append () =  pkeyCond ;
   top -> mutableChi  ldren (). append () =  attrCond ;
   query . mutableFil  ter () =  top ;
}
```

7.8 プライマリキー列の自動増分機能

C++ SDK は、プライマリキー列の自動インクリメントをサポートしています。 プライマリキー列を自動インクリメント列に設定した場合、1 行のデータを書き込むときは、Table Store が自動的にこの値を生成するため、この列の値を空のままにします。 この生成された値はパーティションキー上で一意であり、厳密な増分順序に従います。 詳細については、「プライマリキー列の自動インクリメント」をご参照ください。

テーブルを作成する

 :

自動インクリメントプライマリキーは、整数型である必要があり、 `PrimaryKey ColumnSche ma :: AutoIncrem ent` を設定する必要があります。

```
 CreateTabl eRequest  req ;
{
    // テーブルの不変の設定
     TableMeta &  meta  =  req . mutableMet  a ();
     meta . mutableTab  leName () =  kTableName ;
     Schema &  schema  =  meta . mutableSch  ema ();
    {
         PrimaryKey  ColumnSche  ma &  pkColSchem  a  =  schema .
 append ();
         pkColSchem  a . mutableNam  e () = " ShardKey ";
         pkColSchem  a . mutableTyp  e () =  kPKT_Strin  g ;
    }
    {
         PrimaryKey  ColumnSche  ma &  pkColSchem  a  =  schema .
 append ();
         pkColSchem  a . mutableNam  e () = " AutoIncrKe  y ";
         pkColSchem  a . mutableTyp  e () =  kPKT_Integ  er ;
         pkColSchem  a . mutableOpt  ion (). reset ( PrimaryKey
 ColumnSche  ma :: AutoIncrem  ent );
    }
}
 CreateTabl eResponse  resp ;
 Optional < OTSError >  res  =  client . createTabl  e ( resp ,  req
 );
```

データを書き込む

自動インクリメントのプライマリキー列の値は、Table Store サービスによって入力されます。したがって、書き込み時にプライマリキー列に特別なプレースホルダを入力する必要があります。 `PrimaryKey Value :: toAutoIncr ement ()` は、このプレースホルダオブジェクトを取得することができます。 この行のデータを読み取るために、戻り値がプライマリキーを含むように設定することができます。 Table Store サービスは、プライマリキー内の自動

インクリメントプライマリキー列のプレースホルダを実際の列値で置き換えてから返します。 このプライマリキーを後で使用するために記録できます。

この例は、単一行の上書きに基づいています。 他のインターフェイスでの記述も類似しています。

```
 PutRowRequ  est   req ;
{
     RowPutChan  ge &  chg  =  req . mutableRow  Change ();
     chg . mutableTab  le () =  kTableName ;
     chg . mutableRet  urnType () =  RowChange :: kRT_Primar  yKey ;
     PrimaryKey &  pkey  =  chg . mutablePri  maryKey ();
     pkey . append () =  PrimaryKey  Column (
        " ShardKey ",
         PrimaryKey  Value :: toStr (" shard0 "));
     pkey . append () =  PrimaryKey  Column (
        " AutoIncrKe  y ",
         PrimaryKey  Value :: toAutoIncr  ement ());
}
```

7.9 非同期インターフェイス

クライアントの作成

- 直接作成 (Table Store エンドポイントを使用してクライアントを作成)。

```
 Endpoint   ep (" YourEndpoi  nt ", " YourInstan  ce ");
 Credential   cr (" AccessKeyI  d ", " AccessKeyS  ecret ");
 ClientOpti  ons   opts ;
 AsyncClien  t *  client  =  NULL ;
 Optional < OTSError >  res  =  AsyncClien  t :: create ( client ,
 ep ,  cr ,  opts );
```

:

プライマリアカウントの AccessKey を使用して Table Store にアクセスしないでください。 一時的なトークンまたはサブアカウントの AccessKey を使用することを推奨します。 一時的な STS トークンを使用する場合は、前述のコードの資格情報オブジェクトを次のように変更する必要があります。 `Credential   cr (" AccessKeyI  d ", " AccessKeyS  ecret ", " SecurityTo  ken ");`

設定アイテムの詳細については「同期インターフェイス」をご参照ください。

- SyncClient から構築します。

```
 SyncClient &  sync  = ... ;
```

```
AsyncClien t * async = AsyncClien t :: create ( sync );
```

テーブル操作

非同期インターフェイスの使用方法を示すために、ここでは例としてテーブル操作を使用します。

準備

2 つの関数が用意されている必要があります。

- リクエストオブジェクト

  関数シグネチャ `listTable` は以下の通りです。

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

  最初のパラメーターは可変参照です (対照的に、同期インターフェイスは不変参照です)。`listTable ()` が戻された後 (この時点では、リストテーブル操作は完了していません)、渡された `ListTableR equest` オブジェクトは変更されていたり破壊されたりして、調べるのが難しいわかりにくいエラーが紛れ込む可能性があります。 この種の問題を回避するために、非同期クライアントは渡された要求オブジェクトの内容を (コピーせずに) 内部ストレージに転送します。 つまり、 `listTable ()` を呼び出した後、渡されたオブジェクトを変更できます。
- コールバック関数

  コールバック関数は値を返しません。 3 種類の受信パラメーターは次のとおりです。

  - リクエストオブジェクト 。これは、ユーザーが `listTable ()` を呼び出したときに渡されるリクエストオブジェクトです。 コールバックの後、非同期クライアントはリクエストオブジェクトを必要としなくなり、可変参照の形でユーザーのコールバック関数に戻されます。 これにより、ユーザーはリクエストオブジェクトの内容を転送できます。
  - オプションにラップされているエラーオブジェクト。 エラーが存在しない場合は、このオブジェクトの `present ()` メソッドは false を返します。
  - レスポンスオブジェクト。 リクエストオブジェクトと同様に、レスポンスオブジェクトも可変参照の形でコールバック関数に返されます。 エラーが存在する場合、レスポンスオブ

ジェクトは有効なオブジェクト (破壊される可能性があるもの) でなければなりません。 ただし、レスポンスオブジェクトの内容は未定義です。

注：

- 非同期クライアントは、すべてのリクエストコールバックが一度だけ呼び出されるようにします。
- 理論的には、 `listTable()` が返される前にコールバック関数を呼び出すことができます。

例

```
 void   listTableC  allback (
     ListTableR  equest &,
     Optional < OTSError >&  err ,
     ListTableR  esponse &  resp )
{
     if  ( err . present ()) {
        // 処理中のエラー
    }  else  {
         const   IVector < string >&  xs  =  resp . tables ();
         for ( int64_t   i  =  0 ;  i  <  xs . size (); ++ i ) {
             cout  <<  xs [ i ] <<  endl ;
        }
    }
}
 void   listTable ( AsyncClien  t &  client )
{
     ListTableR  equest   req ;
     client . listTable ( req ,  listTableC  allback );
}
```

7.10 ログ

クライアントはデフォルトでロガーを提供します。 アプリケーションに独自のロガーがある場合は、ログ管理が容易になるため、そちらを使用することを推奨します。

ロガーには、 `tablestore / util / logger . hpp` で定義されているように、次の4つの要素があります。

・ Logger インターフェイス

Logger は、ログの内容を Record オブジェクトに組み立てるために使用されます。この Record オブジェクトは、Sinker に転送され、Sinker によって書き出されます。 クライアントは Logger をツリー構造に編成します。そのルートは ClientOptions でユーザーが定義

したロガーです。リクエストロジックとネットワーキングロジックは、このルートロガーから派生した異なるサブロガーを使用します。

```
 class   Logger
{
 public :
 enum   LogLevel
{
     kDebug ,
     kInfo ,
     kError ,
};

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

- `level ()` は、Logger によって受け入れられたロググレードを返します。低いグレードのログは Sinker に転送されません。
- `record ()` は、ログとそのグレードを受け入れ、それらを使用して、対応する Logger の Sinker に送信される Record オブジェクトを形成します。
- `spawn ()` は、サブロガーを派生させます。

· Record インターフェイス

Logger は Record オブジェクトを使用してログの内容を Sinker に転送します。

Record インターフェイスはメソッド自体を提供しません。Logger と Sinker は、Record クラスが提供するメソッドを決定します。

· Sinker インターフェイス

Sinker は Record オブジェクトの書き出しを担当しています。

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

- `sink ()` は、レコードを書き出します。レコードはキャッシュにしか書き込めません。
- `flush ()` は、すべてのログが保存されていることを確認するためにキャッシュをクリアします。

- SinkerCenter シングルトンオブジェクト

  SinkerCenter は、すべての Sinker オブジェクトを保持し、それらを特定のキーに関連付けます。

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

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

  - `singleton ()` は、SinkerCenter シングルトンオブジェクトを取得します。
  - `registerSi  nker ()` は、Sinker を SinkerCenter に登録します。
  - `flushAll ()` は、SinkerCenter のすべての Sinker をクリアします。

7.11 エラー処理

TableStore C++ SDK は戻り値を使用してエラーを処理します。 エラーを生成する可能性のあるすべてのインターフェイスは `Optional < OTSError >` オブジェクトを返します。

- `Optional < T >` は "tablestore/util/optional.hpp" で定義されているテンプレートクラスです。 ただ 1 つの `T` オブジェクトを保存できる箱として考えます。 ボックス内には、エラーが発生したかどうかを判断するために使用できる 2 つのシナリオがあります。
  - `T` オブジェクトが存在する場合は、それを取り出して利用することができます。 この場合、`Optional < T >:: present ()` は true を返して、エラーが発生したことを示します。
  - `T` オブジェクトが存在しない場合、`Optional < T >:: present ()` は false を返して、エラーが発生していないことを示します。
- OTSError オブジェクトは特定のエラーを示します。 5 つのフィールドがあります。
  - `httpStatus` および `errorCode` は、HTTP リターンコードとエラーコードです。 と *Error messages* に加えて、以下のエラーはクライアントでのみ発生します。

6	OTSCouldntResolveHost	ドメイン名を解決できません。 インスタンスアクセスアドレスが間違っているか、ネットワークが切断されています。

7	OTSCouldntConnect	サービスに接続できません。 ローカルホストファイルの設定エラーです。
28	OTSRequestTimeout	リクエストがタイムアウトしました。
35	OTSSslHandshakeFail	HTTPS ハンドシェイクに失敗しました。 ローカル証明書がインストールされていません。
55	OTSWriteRequestFail	ネットワーク配信に失敗しました。 ネットワークの中断です。
56	OTSCorruptedResponse	不完全な応答です。
89	OTSNoAvailableConnection	利用可能な接続がありません。 これは通常、新しく構築されたクライアントまたは同時要求がネットワーク接続の総数を超えると発生します。

- `message` : エラーの詳細です。
- `requestId` : サービスに送信された各要求には、サービスによって番号が割り当てられます。 応答が正常に返された場合、レスポンスオブジェクトには requestId が含まれています。 リクエストにエラーがあるとサービスが判断した場合、エラーオブジェクトには requestId が含まれます。 要求が送信される前にエラーが発生した場合、またはネットワークリンクでエラーが発生した場合、エラーオブジェクトには requestId は含まれません。
- `traceId` : 各 API 呼び出しには、クライアントによって traceId が割り当てられます。 異なる API 呼び出しには異なる traceIdが 割り当てられています。 同じ API 呼び出しが繰り返し試行された場合、traceId は同じままですが、 requestId は異なる可能性があります。

7.12 再試行

一般的な再試行戦略

SDK は、次の一般的な再試行方法を提供します。

- デフォルトの再試行戦略

  SDK エラーは自動的に再試行を引き起こします。 最大再試行間隔は 10 秒です。

- 再試行戦略のカウント

  ユーザー定義の間隔での再試行。 再試行戦略が安全な場合、最大再試行回数はユーザー定義の数を超えることはできません。

- 再試行なし

カスタム再試行戦略

`RetryStrat egy` を変更して、再試行戦略をカスタマイズできます。

```
class RetryStrat egy
{
public :
    virtual ~ RetryStrat egy () {}

    virtual RetryStrat egy * clone () const = 0 ;
    virtual int64_t retries () const throw () = 0 ;
    virtual bool shouldRetr y ( Action , const OTSError &)
const = 0 ;
    virtual util :: Duration nextPause () = 0 ;
};
```

- `clone ()` は、新しいオブジェクトを複製します。 新しいオブジェクトは、現在のオブジェクトと同じタイプおよび内部ステータス (再試行回数を含む) でなければなりません。
- `retries ()` は、既に完了した再試行の回数です。
- `shouldRetr y ()` は、操作とエラーを指定して再試行が必要かどうかを判断します。

  Table Store には、操作を簡単にするための 2 つのツール機能があります。

  - 最初の再試行では、エラーを 3 つのグループに分けます。

    ■ OTSTableNotReady など、完全に無害な再試行。

    ■ パラメーターエラーなどによる有害または無意味な再試行。

    ■ OTSRequestTimeout などのエラーだけでは判断できないもの。

  - 2 回目の再試行では、操作とエラーも考慮しながら、操作のべき等性原則に基づいて再試行を実行できるかどうかを決定します。 つまり、読み取り操作の RETRIABLE エラーと DEPENDS エラーの両方が再試行可能と判断されます。

    ```
    enum RetryCateg ory
    {
       UNRETRIABL E ,
       RETRIABLE ,
       DEPENDS ,
    };
    static RetryCateg ory retriable ( const OTSError &);
    ```

```
static bool retriable ( Action , const OTSError &);
```

`nextPause ()` は、再試行可能である場合の次の再試行までの間隔です。

8 SDK のダウンロード

8.1 Java SDK

バージョン 4.0.0 以降の SDK は TTL および Max Version を提供するため、バージョン 2.x.x の SDK とは互換性がありません。

SDK バージョン: 4.7.4

公開日: 2018/09/27

ダウンロード: *tablestore-4.7.4-release.zip*

更新内容

- SearchIndex を追加しました。
  - マルチフィールド検索
  - 範囲クエリ
  - ワイルドカード検索
  - 入れ子になったクエリ
  - 全文検索
  - ランキング
- グローバルセカンダリインデックスを追加しました。

SDK バージョン: 4.1.0

公開日: 2016/10/11

ダウンロード: *aliyun_tablestore_java_sdk_4.1.0.zip*

更新内容: パーティションの分割点は DescribeTable の応答から取得できます。

SDK バージョン: 4.0.0

公開日: 2016/08/01

ダウンロード: *aliyun_tablestore_java_sdk_4.0.0.zip*

更新内容

- *Time To Live* を提供します。
- *Max Version* を提供します。

SDK バージョン: 2.2.4

公開日: 2016/05/12

ダウンロード: *aliyun_tablestore_java_sdk_2.2.4.zip*

更新内容

- API 条件の更新を追加しました。
- フィルタを追加しました。

SDK バージョン 2.1.0

公開日: 2015/12/11

ダウンロード: *aliyun-ots-java-sdk-2.1.0.zip*

更新内容

- 非同期ネットワーク伝送とパフォーマンスチューニング: CPU 使用率が同じ場合、 QPS は数倍になります。
- 柔軟で使いやすい非同期インターフェイス: コールバックが導入され、同時に Future が返されます。
- OSS SDK からバンドル解除: 新しいバージョンには、TableStore SDK のコードのみが含まれています。 ディレクトリがわずかに調整されています。
- 最適化された再試行ロジック: デフォルトの再試行ロジックが最適化されています。 誤った単一行は、バッチ操作中に独立して再試行できます。 再試行ロジックのカスタムメソッドが明確になりました。
- 最適化されたログ: 要求送信から要求受信までの各ステップについてログが記録されます。 遅い要求のログが記録されます。 SDK からバックエンドサービスまでのチェーン全体のログは、TraceId を使用して記録されます。
- バッチデータインポートをサポートする OTSWriter インタフェイス: このインタフェイスは、ユーザに使いやすく効率的なデータインポートサービスを提供することを目的としています。
- その他の最適化機能: さまざまなデータクラスのためのツールボックス機能が充実しており、データサイズ計算のためのインターフェイスが提供されています。

 注 :

以下の制限により、バージョン 2.1.0 はバージョン 2.0.4 と若干互換性に欠ける部分があります。

- 古い SDK を新しいものに置き換えるときは、いくつかのクラスのインポートパスを変更する必要があります。これらのデータクラスのパッケージは調整されているためです。 たとえば、"ClientConfiguration" のパッケージは "com.aliyun.openservices" から "com.aliyun.openservices.ots" に変更されています。 パッケージが調整される主な理由は、Table Store SDK が OSS SDK からアンバンドルされているためです。 したがって、一般的に使用されるデータのクラスを Table Store のパッケージに入れる方が適切です。
- OTSClient インスタンスを使用しなくなったら (たとえば、プログラムが終了する前に)、OTSClient の shutdown メソッドを呼び出して、OTSClient オブジェクトが占有していたスレッドと接続リソースを解放します。
- "ClientConfiguration" の一部の設定項目の名前を調整しました。 例えば、設定の単位として時間単位が追加されます。
- 新しい SDK のパッケージ間の依存関係が変更されました。 たとえば、"HttpAsyncClient" と "Jodatime" が使用されます。 SDK の実行中に問題が発生した場合は、競合する依存関係が発生していないか確認します。

SDK バージョン: 2.0.4

公開日: 2015/09/25

ダウンロード: *aliyun-ots-java-sdk-2.0.4.zip*

8.2 NodeJS SDK

バージョン: 4.0.6

公開日: 2016/10/09

ダウンロード: *aliyun-tablestore-nodejs-sdk-4.0.6.tar.gz*

GitHub: *v4.0.6*

更新内容

- 新機能 `async` `await` および関連サンプルコードのサポートを追加しました。
- 中国語のテキストを読み取るときのエラーを修正しました。
- STSToken のサポートを追加しました。
- 基準 maxTimeDeviation のサポートを UpdateTable オペレーションに追加します。
- バージョン、ビルド、カバレッジ、およびライセンスのロゴを Readme に追加します。

バージョン: 4.0.3

公開日: 2016/08/27

ダウンロード: *aliyun-tablestore-nodejs-sdk-4.0.3.tar.gz*

GitHub: *v4.0.3*

更新内容

・ バージョン情報を統一します。

バージョン: 4.0.1

公開日: 2017/08/27

ダウンロード: *aliyun-tablestore-nodejs-sdk-4.0.1.tar.gz*

GitHub: *v4.0.1*

更新内容

・ 不要なコードを削除します。
・ GetRange が 0 個のデータを返すときの値を修正します。

バージョン: 4.0.0

公開日: 2017/07/07

ダウンロード: *aliyun-tablestore-nodejs-sdk-4.0.0.tar.gz*

GitHub: *v4.0.0*

8.3 Python SDK

Python SDK **開発キットバージョン** 4.3.0

公開日: 2017/12/12

ダウンロード: *aliyun-tablestore-python-sdk-4.3.0.tar.gz*

更新内容

・ Python 3 (Python 3.3、Python 3.4、Python 3.5、Python 3.6) をサポートします。

Python SDK **開発キットバージョン** 4.2.0

公開日: 2017/09/12

ダウンロード: *aliyun-tablestore-python-sdk-4.2.0.tar.gz*

更新内容

・ STS をサポートします。

Python SDK 開発キットバージョン 4.1.0

公開日: 2017/06/28

ダウンロード: *aliyun-tablestore-python-sdk-4.1.0.tar.gz*

更新内容

- python 2.6 をサポートしています。

Python SDK 開発キットバージョン 4.0.0

公開日: 2017/06/27

ダウンロード: *aliyun-tablestore-python-sdk-4.0.0.tar.gz*

更新内容

- マルチバージョンをサポートしています。
- TTL をサポートしています。
- プライマリキー列の自動インクリメント機能をサポートします。

Python SDK 開発キットバージョン 2.1.0

公開日: 2016/10/15

ダウンロード: *aliyun-ots-python-sdk-2.1.0.zip*

更新内容

- 条件付き更新とフィルタを提供します。
- 再試行のバグを修正しました。

Python SDK 開発キットバージョン 2.0.8

公開日: 2016/03/30

ダウンロード: *aliyun-ots-python-sdk-2.0.8.zip*

更新内容

- HTTPS アクセスと証明書検証を提供します。

Python SDK 開発キットバージョン 2.0.7

公開日: 2015/12/30

ダウンロード: *aliyun-ots-python-sdk-2.0.7.zip*

更新内容

・ サンプルコードを改良します。

Python SDK **開発キットバージョン** 2.0.6

公開日: 2015/10/23

ダウンロード: *aliyun-ots-python-sdk-2.0.6.zip*

更新内容

・ 指定された例外に対するバックオフの再試行方法を調整します。

Python SDK **開発キットバージョン** 2.0.5

公開日: 2015/09/25

ダウンロード: *ots_python_sdk-2.0.5.zip*

8.4 .Net SDK

バージョン: 3.0.0

公開日: 2016/05/05

ダウンロード: *aliyun_table_store_dotnet_sdk_3.0.0.zip*

更新内容

・ フィルタが追加されました。
・ コンパイルプロセス中に生成された警告は消去されます。
・ 不要な依存パッケージは消去されます。
・ 無なコードはクリアされます。
・ テンプレートと呼ばれるコードは単純化されています。
・ 無効なパラメーターチェックが追加されました。
・ ユーザー設定パラメーターはトリミングされます。
・ UserAgent ヘッダーが追加されました。
・ Condition.IGNORE、Condition.EXPECT_EXIST および Condition.EXPECT_NOT_EXIST は削除されます。
・ DLL ファイル名が Aliyun.dll から Aliyun.TableStore.dll に変更されました。
・ オープンソースコードが GitHub にリリースされました。

バージョン: 2.2.1

公開日: 2016/04/14

ダウンロード: *aliyun-ots-dotnet-sdk-2.2.1.zip*

更新内容

- SDK が HTTP 応答ヘッダーを内部的にチェックするとき、文字の大文字と小文字の区別は無視されます。

バージョン: 2.2.0

公開日: 2016/04/05

ダウンロード: *aliyun-ots-dotnet-sdk-2.2.0.zip*

更新内容

- 接続プールのデフォルトの接続数が 50 から 300 に増えました。
- 条件更新機能を追加しました。

バージョン: 2.1.0

公開日: 2015/12/30

ダウンロード: *aliyun-ots-dotnet-sdk-2.1.0.zip*

更新内容

- サンプルコードの予約済みの容量単位設定は、従量課金方式に基づいて調整されます。
- より広範囲の BatchWriteRow および BatchGetRow テストケースを追加します。

8.5 PHP SDK

PHP SDK バージョン 2.1.1

公開日: 2017/01/14

ダウンロード: *aliyun-tablestore-php-sdk-2.1.1.zip*

更新内容

- 32 ビットオペレーションシステムをサポートします。

PHP SDK バージョン 2.1.0

公開日: 2016/11/16

ダウンロード: *aliyun-ots-php-sdk-2.1.0.zip*

更新内容

- 条件付き更新とフィルタを提供します。

- PHP バージョン 5.5 および 5.6 と互換性があります。

PHP SDK バージョン 2.0.3

公開日: 2016/05/18

ダウンロード: *aliyun-ots-php-sdk-2.0.3.zip*

更新内容

- サンプルコードからテーブル削除を削除します。

PHP SDK バージョン 2.0.2

公開日: 2016/04/11

ダウンロード: *aliyun-ots-php-sdk-2.0.2.zip*

更新内容

- pb デコードのバグを修正しました。

PHP SDK バージョン 2.0.0

公開日: 2015/09/22

ダウンロード: *aliyun-ots-php-sdk-2.0.0.zip*

更新内容

- Table Store API のすべてをサポートしています。
- PHP バージョン 5.3、5.4、5.5 および 5.6 と互換性があります。
- 標準の再試行方法が含まれています。
- ネットワークライブラリとして Guzzle Http Client を使用します。
- 依存関係管理およびエンジニアリング組織ツールとして composer を使用します。
- HTML 形式のプログラミングドキュメントを生成するために phpDocumentor 2 を使います。