Alibaba Cloud Table Store

產品功能

檔案版本：20180929

目錄

1 Table Store的表

建立Table Store的表時，需要指定表名、主鍵和預留讀/寫輸送量。在傳統資料庫中，表具有預定義的結構資訊，比如表的名稱、主鍵、列名和類型，表中的所有行都具有相同的列集合。Table Store是 NoSQL 資料庫，除了主鍵需要格式外，沒有其他的格式資訊。本章將介紹表的概念和使用。

表名

Table Store的表的名稱必須符合以下規範:

- 由英文字元（a-z）或（A-Z）、數字（0-9）和底線（_）組成。
- 首字母必須為英文字母（a-z）、（A-Z）或底線（_）。
- 大小寫敏感。
- 長度在 1~255 字元之間。
- 同一個執行個體下不能有同名的表，但不同執行個體內的表名稱可以相同。

主鍵

建立Table Store的表時必須指定表的主鍵。主鍵包含 1~4 個主鍵列，每個主鍵列都有名字和類型。Table Store對主鍵列的名字和類型都有限制，詳細資料請參考#####。

Table Store根據表的主鍵索引資料，表中的行按照它們的主鍵進行升序排序。

配置預留讀/寫輸送量

為確保應用程式獲得穩定、低延時的服務，應用程式可以在建立表的時候指定預留讀/寫輸送量。如果預留讀/寫輸送量不為 0，Table Store根據預留讀/寫輸送量來為表分配相應的資源，滿足應用程式的預留輸送量需求，同時根據配置的預留讀/寫輸送量收取相應費用。應用程式可以根據自身的業務需求動態上調和下調錶的預留讀/寫輸送量。預留讀/寫輸送量通過讀服務能力單元和寫服務能力單元這兩個數值來設定。


说明：
容量型執行個體下的表不支援預留讀/寫輸送量。

通過 UpdateTable 操作可以更新表的預留讀/寫輸送量。預留讀/寫輸送量的更新有如下規則:

- 一張表上的兩次更新的間隔必須大於 2 分鐘。例如，12:43 AM 更新了某個表的預留讀/寫輸送量，那麼只有在 12:45 AM 之後才能再次更新該表的預留讀/寫輸送量。更新間隔必須大於 2 分鐘的限制是針對錶的，在 12:43 AM ~ 12:45 AM 之間，使用者可以更新其他表的預留讀/寫輸送量。

- 每個自然日內（UTC 時間 00:00:00 到第二天的 00:00:00，北京時間早上 8 點到第二天早上 8 點），上調或者下調預留讀/寫輸送量的總次數不做限制，但是兩次調整之間的時間間隔必須大於 2 分鐘。調整預留讀/寫輸送量的定義是，只要讀服務能力單元或寫服務能力單元配置其中一項進行了更新，則此次操作被視為對錶進行了更新。
- 預留讀/寫輸送量調整完畢後 1 分鐘內生效。

對於訪問消耗的讀/寫輸送量中，超出預留讀/寫輸送量的部分，會計入按量讀/寫輸送量，並根據按量讀/寫輸送量單價進行計費。

由於預留讀/寫輸送量在單價上低於按量讀/寫輸送量，配置合適的預留讀/寫輸送量可以進一步降低成本。例如，在使用者剛建表之後如果需要匯入大量資料，可以設定較大的預留寫輸送量，能夠以較低的寫成本將資料匯入進來，當資料匯入完畢後，再將預留讀/寫輸送量下調。

分區鍵下的資料量限制

Table Store按照分區鍵的範圍對錶的資料進行分區，擁有相同分區鍵的行會被劃分到同一個分區。為了防止分區過大無法切分，建議單個分區索引值下的所有行資料大小之和不要超過 10 GB。

建表後載入時間

Table Store的表在被建立之後需要 1 分鐘進行載入，在此期間對該表的讀/寫資料操作均會失敗。應用程式應該等待表載入完畢後再進行資料操作。

最佳實務

Table Store######

使用**Table Store**的 **SDK** 進行表操作

JAVA-SDK -- ###

Python-SDK -- ###

2 Table Store的資料操作

Table Store的表由行組成，每一行包含主鍵和屬性。本節將介紹Table Store資料的操作方法。

Table Store行簡介

組成Table Store表的基本單位是行，行由主鍵和屬性群組成。其中，主鍵是必須的，且每一行的主鍵列的名稱和類型相同；屬性不是必須的，並且每一行的屬性可以不同。更多資訊請參見Table Store的######。

Table Store的資料操作有以下三種類型：

- 單行操作
  - GetRow：讀取單行資料。
  - PutRow：新插入一行。如果該行內容已經存在，則先刪除舊行，再寫入新行。
  - UpdateRow：更新一行。應用可以增加、刪除一行中的屬性列，或者更新已經存在的屬性列的值。如果該行不存在，則新增一行。
  - DeleteRow：刪除一行。
- 批量操作
  - BatchGetRow：批量讀取多行資料。
  - BatchWriteRow：批量插入、更新或者刪除多行資料。
- 範圍讀取
  - GetRange：讀取表中一個範圍內的資料。

Table Store單行操作

- 單行寫入操作

  Table Store的單行寫操作有三種：PutRow、UpdateRow 和 DeleteRow。下面分別介紹每種操作的行為語義和注意事項:

  - PutRow：新寫入一行。如果這一行已經存在，則這一行舊的資料會先被刪除，再新寫入一行。
  - UpdateRow：更新一行的資料。Table Store會根據請求的內容在這一行中新增列，修改或者刪除指定列的值。如果這一行不存在，則會插入新的一行。但是有一種特殊的情境，若 UpdateRow 請求只包含刪除指定的列，且該行不存在，則該請求不會插入新行。
  - DeleteRow：刪除一行。如果刪除的行不存在，則不會發生任何變化。

應用程式通過佈建要求中的 condition 欄位來指定寫入操作執行時，是否需要對行的存在性進行檢查。condition 有三種類型：

- IGNORE：不做任何存在性檢查。
- EXPECT_EXIST：期望行存在。如果該行存在，則操作成功；如果該行不存在，則操作失敗。
- EXPECT_NOT_EXIST：期望行不存在。如果行不存在，則操作成功；如果該行存在，則操作失敗。

condition 為 EXPECT_NOT_EXIST 的 DeleteRow、UpdateRow 操作是沒有意義的，刪除一個不存在的行是無意義的，如果需要更新不存在的行可以使用 PutRow 操作。

如果操作發生錯誤，如參數檢查失敗、單行資料量過多、行存在性檢查失敗等等，會返回錯誤碼給應用程式。如果操作成功，Table Store會將操作消耗的服務能力單元返回給應用程式。

各操作消耗的寫服務能力單元的計算規則如下：

- PutRow：本次消耗的寫 CU 為修改的行主鍵資料大小與屬性列資料大小之和除以 4 KB 向上取整。若指定條件檢查不為 IGNORE，還需消耗該行主鍵資料大小除以 4 KB 向上取整的讀 CU。如果操作不滿足應用程式指定的行存在性檢查條件，則操作失敗並消耗 1 單位寫服務能力單元和 1 單位讀服務能力單元。請參見 *PutRow* 詳解。
- UpdateRow：本次消耗的寫 CU 為修改的行主鍵資料大小與屬性列資料大小之和除以 4 KB 向上取整。UpdateRow 中包含的需要刪除的屬性列，只有其列名計入該屬性列資料大小。若指定條件檢查不為 IGNORE，還需消耗該行主鍵資料大小除以 4 KB 向上取整的讀 CU。如果操作不滿足應用程式指定的行存在性檢查條件，則操作失敗並消耗 1 單位寫服務能力單元和 1 單位讀服務能力單元。請參見 *UpdateRow* 詳解。
- DeleteRow：被刪除的行主鍵資料大小除以 4 KB 向上取整。若指定條件檢查不為 IGNORE，還需消耗該行主鍵資料大小除以 4 KB 向上取整的讀 CU。如果操作不滿足應用程式指定的行存在性檢查條件，則操作失敗並消耗 1 單位寫服務能力單元。請參見 *DeleteRow* 詳解。

寫操作會根據指定的 condition 情況消耗一定的讀服務能力單元。

樣本

下面舉例說明單行寫操作的寫服務能力單元和讀服務能力單元的計算。

樣本 1，使用 PutRow 進行如下行寫入操作：

```
//PutRow 操作
//row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+
3000Byte=4322Byte
{
```

```
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte), 'value2':String(3000Byte
)}
}

//原來的行
//row_size=len('pk')+len('value2')+8Byte+900Byte=916Byte
//row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
    attributes:{'value2':String(900Byte)}
}
```

讀/寫服務能力單元的消耗情況如下：

- 將 condition 設定為 EXPECT_EXIST 時：消耗的寫服務能力單元為 4322 Byte 除以 4 KB 向上取整，消耗的讀服務能力單元為該行主鍵資料大小 10 Byte 除以 4 KB 向上取整。該 PutRow 操作消耗 2 個單位的寫服務能力單元和 1 個單位的讀服務能力單元。
- 將 condition 設定為 IGNORE 時：消耗的寫服務能力單元為 4322 Byte 除以 4 KB 向上取整，消耗 0 個讀服務能力單元。該 PutRow 操作消耗 2 個單位的寫服務能力單元和 0 個單位的讀服務能力單元。
- 將 condition 設定為 EXPECT_NOT_EXIST 時：指定的行存在性檢查條件檢查失敗，該 PutRow 操作消耗 1 個單位的寫服務能力單元和 1 個單位的讀服務能力單元。

樣本 2，使用 UpdateRow 新寫入一行：

```
//UpdateRow 操作
//刪除的屬性列列名長度計入 row_size
//row_size=len('pk')+len('value1')+len('value2')+8Byte+900Byte=
922Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte), 'value2':Delete}
}

//原來的行不存在
//row_size=0
```

讀/寫服務能力單元的消耗情況如下：

- 將 condition 設定為 IGNORE 時：消耗的寫服務能力單元為 922 Byte 除以 4 KB 向上取整，消耗 0 個讀服務能力單元。該 UpdateRow 操作消耗 1個單位的寫服務能力單元和 0 個讀服務能力單元。
- 將 condition 設定為 EXPECT_EXIST 時：指定的行存在性檢查條件檢查失敗，該 PutRow 操作消耗 1 個單位的寫服務能力單元和 1 個單位的讀服務能力單元。

樣本 3，使用 UpdateRow 對存在的行進行更新操作：

```
//UpdateRow 操作
```

```
//row_size=len('pk')+len('value1')+len('value2')+8Byte+1300Byte+
3000Byte=4322Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(1300Byte), 'value2':String(3000Byte
)}
}
//原來的行
//row_size=len('pk')+len('value1')+8Byte+900Byte=916Byte
//row_primarykey_size=len('pk')+8Byte=10Byte
{
    primary_keys:{'pk':1},
    attributes:{'value1':String(900Byte)}
}
```

讀/寫服務能力單元的消耗情況如下：

- 將 condition 設定為 EXPECT_EXIST 時：消耗的寫服務能力單元為 4322 Byte 除以 4 KB 向上取整，消耗的讀服務能力單元為該行主鍵資料大小 10 Byte 除以 4 KB 向上取整。該 UpdateRow 操作消耗 2 個單位的寫服務能力單元和 1 個單位的讀服務能力單元。
- 將 condition 設定為 IGNORE 時：消耗的寫服務能力單元為 4322 Byte 除以 4 KB 向上取整，消耗 0 個讀服務能力單元，該 UpdateRow 操作消耗 2 個單位的寫服務能力單元和 0 個單位的讀服務能力單元。

樣本 4，使用 DeleteRow 刪除不存在的行：

```
//原來的行不存在
//row_size=0

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

修改前後的資料大小均為 0，無論讀寫操作成功還是失敗至少消耗 1 個單位服務能力單元。因此，該 DeleteRow 操作消耗 1 個單位的寫服務能力單元。

讀/寫服務能力單元的消耗情況如下：

- 將 condition 設定為 EXPECT_EXIST 時：消耗的寫服務能力單元為該行主鍵資料大小 10 Byte 除以 4 KB 向上取整，消耗的讀服務能力單元為該主鍵資料大小 10 Byte 除以 4 KB 向上取整。該 DeleteRow 操作消耗 1 個單位的寫服務能力單元和 1 個單位的讀服務能力單元。
- 將 condition 設定為 IGNORE 時：消耗的寫服務能力單元為該行主鍵資料大小 10 Byte 除以 4 KB 向上取整，消耗 0 個讀服務能力單元。該 DeleteRow 操作消耗 1 個單位的寫服務能力單元和 0 個單位的讀服務能力單元。

更多資訊請參見 API Reference 中的 *PutRow*、*UpdateRow* 和 *DeleteRow* 章節。

- 單行讀取操作

  Table Store的單行讀操作只有一種：GetRow。

  應用程式提供完整的主鍵和需要返回的列名。列名可以是主鍵列或屬性列，也可以不指定要返回的列名，此時請求返回整行資料。

  Table Store根據被讀取的行主鍵的資料大小與實際讀取的屬性列資料大小之和計算讀服務能力單元。將行資料大小除以 4 KB 向上取整作為本次讀取操作消耗的讀服務能力單元。如果操作指定的行不存在，則消耗 1 單位讀服務能力單元，單行讀取操作不會消耗寫服務能力單元。

  樣本

  使用 GetRow 讀取一行消耗的寫服務能力單元的計算：

```
//被讀取的行

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

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

  消耗的讀服務能力單元為 1216 Byte 除以 4 KB 向上取整，該 GetRow 操作消耗 1 個單位的讀服務能力單元。

  更多資訊請參見 API Reference 的 *GetRow* 章節。

多行操作

Table Store提供了 BatchWriteRow 和 BatchGetRow 兩種多行操作。

- BatchWriteRow 用於插入、修改、刪除一個表或者多個表中的多行記錄。BatchWriteRow 操作由多個 PutRow、UpdateRow、DeleteRow 子操作組成。BatchWriteRow 的各個子操作獨立執行，Table Store會將各個子操作的執行結果分別返回給應用程式，可能存在部分請求成功、部分請求失敗的現象。即使整個請求沒有返回錯誤，應用程式也必須要檢查每個子操作返回的結果，從而拿到正確的狀態。BatchWriteRow 的各個子操作單獨計算寫服務能力單元。
- BatchGetRow 用於讀取一個表或者多個表中的多行記錄。BatchGetRow 各個子操作獨立執行，Table Store會將各個子操作的執行結果分別返回給應用程式，可能存在部分請求成功、部分請

求失敗的現象。即使整個請求沒有返回錯誤，應用程式也必須要檢查每個子操作返回的結果，從而拿到正確的狀態。BatchGetRow 的各個子操作單獨計算讀服務能力單元。

更多資訊請參見 API Reference 中的 *BatchWriteRow* 與 *BatchGetRow* 章節。

範圍讀取操作

Table Store提供了範圍讀取操作 GetRange，該操作將指定主鍵範圍內的資料返回給應用程式。

Table Store表中的行按主鍵進行從小到大排序，GetRange 的讀取範圍是一個左閉右開的區間。操作會返回主鍵屬於該區間的行資料，區間的起始點是有效主鍵或者是由 INF_MIN 和 INF_MAX 類型組成的虛擬點，虛擬點的列數必須與主鍵相同。其中，INF_MIN 表示無限小，任何類型的值都比它大；INF_MAX 表示無限大，任何類型的值都比它小。

GetRange 操作需要指定請求列名，請求列名中可以包含多個列名。如果某一行的主鍵屬於讀取的範圍，但是不包含指定返回的列，那麼請求返回結果中不包含該行資料。不指定請求列名，則返回完整的行。

GetRange 操作需要指定讀取方向，讀取方向可以為正序或逆序。假設同一表中有兩個主鍵 A 和 B，A < B。如正序讀取 [A, B)，則按從 A 至 B 的順序返回主鍵大於等於 A、小於 B 的行。逆序讀取 [B,A)，則按從 B 至 A 的順序返回大於 A、小於等於 B 的資料。

GetRange 操作可以指定最大返回行數。Table Store按照正序或者逆序最多返回指定的行數之後即結束該操作的執行，即使該區間內仍有未返回的資料。

GetRange 操作可能在以下幾種情況下停止執行並返回資料給應用程式：

- 返回的行資料大小之和達到 4 MB。
- 返回的行數等於 5000。
- 返回的行數等於最大返回行數。
- 當前剩餘的預留讀輸送量已被全部使用，餘量不足以讀取下一條資料。同時 GetRange 請求的返回結果中還包含下一條未讀資料的主鍵，應用程式可以使用該傳回值作為下一次 GetRange 操作的起始點繼續讀取。如果下一條未讀資料的主鍵為空白，表示讀取區間內的資料全部返回。

Table Store計算讀取，區間起始點到下一條未讀資料的起始點的，所有行主鍵資料大小與實際讀取的屬性列資料大小之和。將資料大小之和除以 4 KB 向上取整計算消耗的讀服務能力單元。例如，若讀取範圍中包含 10 行，每行主鍵資料大小與實際讀取到的屬性列資料之和佔用資料大小為 330 Byte，則消耗的讀服務能力單元為 1（資料總和 3.3 KB，除以 4 KB 向上取整為 1）。

樣本

下面舉例說明 GetRange 操作的行為。假設表的內容如下，PK1、PK2 是表的主鍵列，類型分別為 String 和 Integer；Attr1、Attr2 是表的屬性列。

PK1	PK2	Attr1	Attr2
'A'	2	'Hell'	'Bell'
'A'	5	'Hello'	不存在
'A'	6	不存在	'Blood'
'B'	10	'Apple'	不存在
'C'	1	不存在	不存在
'C'	9	'Alpha'	不存在

樣本 1，讀取某一範圍內的資料：

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

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

樣本 2，利用 INF_MIN 和 INF_MAX 讀取全表資料：

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

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

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

樣本 3，在某些主鍵列上使用 INF_MIN 和 INF_MAX：

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

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

樣本 4，逆序讀取：

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

//響應
cosumed_read_capacity_unit: 1
rows: {
      {
```

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

樣本 5，指定列名不包含 PK：

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

//響應
cosumed_read_capacity_unit: 1
rows: {
       {
  attribute_columns: {"Attr1", STRING, "Alpha"}
       }
     }
```

樣本 6，指定列名中包含 PK：

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

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

樣本 7，使用 limit 和斷點：

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

```
//響應 1
cosumed_read_capacity_unit: 1
rows: {
      {
     primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 2)
     attribute_columns:("Attr1", STRING, "Hell"), ("Attr2", STRING, "
Bell")
      },
      {
     primary_key_columns:("PK1", STRING, "A"), ("PK2", INTEGER, 5)
     attribute_columns:("Attr1", STRING, "Hello")
      }
    }
next_start_primary_key:("PK1", STRING, "A"), ("PK2", INTEGER, 6)

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

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

樣本 8，使用 GetRange 操作消耗的讀服務能力單元計算：

在以下表中執行 GetRange 操作，其中 PK 1 是表的主鍵列，Attr1、Attr2 是表的屬性列。

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

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

//響應
cosumed_read_capacity_unit: 1
rows: {
      {
     primary_key_columns:("PK1", INTEGER, 1)
      },
      {
     primary_key_columns:("PK1", INTEGER, 2),
```

```
    attribute_columns:("Attr1", INTEGER, 8)
     },
     {
    primary_key_columns:("PK1", INTEGER, 3),
    attribute_columns:("Attr1", STRING, String(1000Byte))
     },
   }
```

此次 GetRange 請求中：

- 擷取的第一行資料大小為：len ('PK1') + 8 Byte = 11 Byte
- 第二行資料大小為：len ('PK1') + 8 Byte + len ('Attr1') + 8 Byte = 24 Byte
- 第三行資料大小為：len ('PK1')+ 8 Byte + len ('Attr1') + 1000 Byte = 1016 Byte

消耗的讀服務能力單元為擷取的三行資料之和 11 Byte + 24 Byte + 1016 Byte = 1051 Byte 除以 4 KB 向上取整，該 GetRange 操作消耗 1 個單位的讀服務能力單元。

更多詳細資料請參見 API Reference 中的 *GetRange* 章節。

最佳實務

Table Store#########

使用**Table Store SDK** 進行資料操作

TableStore Java SDK

TableStore Python SDK

3 使用條件更新

條件更新功能只有在滿足條件時才對錶中的資料變更，當不滿足條件時更新失敗。該功能支援算術運算（=、!=、>、>=、<、<=）和邏輯運算（NOT、AND、OR），支援最多 10 個條件的組合，可用於 *PutRow*、*UpdateRow*、*DeleteRow* 和 *BatchWriteRow* 中。

列判斷條件包括行存在性條件和列條件：

- ######分為 IGNORE、EXPECT_EXIS 和 EXPECT_NOT_EXIST，分別代表忽略、期望存在和期望不存在。對錶變更操作時，會首先檢查行存在性條件，若不滿足，則更改失敗，對使用者拋錯。
- 列條件目前支援 SingleColumnValueCondition 和 CompositeColumnValueCondition，是基於某一列或者某些列的列值進行條件判斷，與過濾器 Filter 中的條件類似。

條件更新可以實現樂觀鎖的功能，即在更新某行時，先擷取某列的值，假設為列 A，值為 1，然後設定條件列 A = 1，更新該行同時使列 A = 2。若更新失敗，代表有其他用戶端已經成功更新了該行。

操作步驟

1. 構造 SingleColumnValueCondition。

```
 // 設定條件為 Col0==0。
 SingleColumnValueCondition singleColumnValueCondition = new
SingleColumnValueCondition("Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL,
ColumnValue.fromLong(0));
 // 如果不存在 Col0 這一列，條件檢查不通過。
 singleColumnValueCondition.setPassIfMissing(false);
 // 只判斷最新版本。
 singleColumnValueCondition.setLatestVersionsOnly(true);
```

2. 構造 CompositeColumnValueCondition。

```
 // composite1 條件為 (Col0 == 0) AND (Col1 > 100)
 CompositeColumnValueCondition composite1 = new CompositeColumnValue
Condition(CompositeColumnValueCondition.LogicOperator.AND);
 SingleColumnValueCondition single1 = new SingleColumnValueCondition
("Col0",
         SingleColumnValueCondition.CompareOperator.EQUAL,
ColumnValue.fromLong(0));
 SingleColumnValueCondition single2 = new SingleColumnValueCondition
("Col1",
         SingleColumnValueCondition.CompareOperator.GREATER_THAN,
ColumnValue.fromLong(100));
 composite1.addCondition(single1);
 composite1.addCondition(single2);

 // composite2 條件為 ( (Col0 == 0) AND (Col1 > 100) ) OR (Col2 <= 10
)
```

```
 CompositeColumnValueCondition composite2 = new CompositeColumnValue
Condition(CompositeColumnValueCondition.LogicOperator.OR);
 SingleColumnValueCondition single3 = new SingleColumnValueCondition
("Col2",
         SingleColumnValueCondition.CompareOperator.LESS_EQUAL,
ColumnValue.fromLong(10));
 composite2.addCondition(composite1);
 composite2.addCondition(single3);
```

3. 通過 Condition 實現樂觀鎖機制, 遞增一列。

```
 private static void updateRowWithCondition(SyncClient client,
String pkValue) {
     // 構造主鍵
     PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
     primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString(pkValue));
     PrimaryKey primaryKey = primaryKeyBuilder.build();

     // 讀一行
     SingleRowQueryCriteria criteria = new SingleRowQueryCriteria(
TABLE_NAME, primaryKey);
     criteria.setMaxVersions(1);
     GetRowResponse getRowResponse = client.getRow(new GetRowRequest
(criteria));
     Row row = getRowResponse.getRow();
     long col0Value = row.getLatestColumn("Col0").getValue().asLong
();

     // 條件更新 Col0 這一列, 使列值 +1
     RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);
     Condition condition = new Condition(RowExistenceExpectation.
EXPECT_EXIST);
     ColumnCondition columnCondition = new SingleColumnValueCon
dition("Col0", SingleColumnValueCondition.CompareOperator.EQUAL,
ColumnValue.fromLong(col0Value));
     condition.setColumnCondition(columnCondition);
     rowUpdateChange.setCondition(condition);
     rowUpdateChange.put(new Column("Col0", ColumnValue.fromLong(
col0Value + 1)));

     try {
         client.updateRow(new UpdateRowRequest(rowUpdateChange));
     } catch (TableStoreException ex) {
         System.out.println(ex.toString());
     }
 }
```

使用情境樣本

對高並發的應用進行更新的操作樣本如下：

```
// 取回當前值
   old_value = Read();
// 對當前值進行計算，比如加 1 操作
   new_value = func(old_value);
// 使用最新值進行更新
```

```
    Update(new_value);
```

在高並發的環境下，old_value 可能被其他用戶端更新，若使用 Conditional Update，就可以做到 Update (new_value) if value 等於 old_value。

说明：

在網頁計數和遊戲等高並發情境下，使用條件更新後會有一定幾率的失敗，需要一定次數的重試。

費用計算

資料寫入或者更新成功，不影響各個介面的 CU 計算規則，如果條件更新失敗，則會消耗 1 單位寫 CU 和 1 單位讀 CU。

4 主鍵列自增

本節介紹Table Store中的主鍵列自增功能，即若使用者佈建某一列主鍵為自增列，在寫入一行資料時，這一列主鍵不用填值，Table Store會自動為使用者產生這一列主鍵的值，這個值在分區鍵上保證唯一，且嚴格遞增。

特點

Table Store的主鍵列自增主要有以下特點：

- Table Store專屬的系統架構和主鍵自增列實現方式，可以保證產生的自增列的值唯一，且嚴格遞增。
- 自動產生的自增列為 64 位元的有符號長整型。
- 分區健層級嚴格遞增。
- 自增列功能是表層級的，同一個執行個體下面可以有自增列的表，也可以有非自增列的表。

使用主鍵列自增功能後，條件更新的邏輯和之前一樣，具體如下表所示：

API	IGNORE	EXPECT_EXIST	EXPECT_NOT_EXIST
PutRow：已存在行	失敗	成功	失敗
PutRow：不存在行	成功	失敗	失敗
UpdateRow： 已存在行	失敗	成功	失敗
UpdateRow：不存在行	成功	失敗	失敗
DeleteRow： 已存在行	成功	成功	失敗
DeleteRow： 不存在行	失敗	失敗	失敗

限制

Table Store的主鍵列自增主要存在以下限制：

- Table Store支援多個主鍵，第一個主鍵為分區鍵，分區鍵不允許設定為自增列。除分區鍵以外，其它任一主鍵都可以設定為自增列。
- 每張表最多隻允許設定一個主鍵為自增列。
- 屬性列不能設定為自增列。
- 僅支援在建立表的時候指定自增列，對於已存在的表不支援建立自增列。

介面

- CreateTable
  - 建表的時候需要設定某一列為自增列，具體設定方式見SDK。
  - 表建立好後，無法再次更改表是否為自增表。
- UpdateTable

  無法通過UpdateTable更改表的自增屬性。
- PutRow/UpdateRow/BatchWriteRow
  - 寫入的時候，需要自增的列不需要設定具體值，只需要設定一個預留位置，比如AUTO_INCREMENT，具體見SDK。
  - 可以設定ReturnContent中的ReturnType為RT_PK，即返回完整主索引值，可以用於GetRow查詢。
- GetRow/BatchGetRow

  GetRow的時候需要完整主鍵列，可以通過設定PutRow、UpdateRow或BatchWriteRow中的ReturnType為RT_PK擷取到。
- 其他介面

  無變化

使用

JAVA SDK ######

計費

主鍵列自增功能不影響現有計費邏輯，返回的主鍵列資料不會額外消耗讀 CU。

5 原子計數器

原子計數器是Table Store提供的一個新特性，即將列當成一個原子計數器來使用。這樣便於為某些線上應用提供即時統計功能，比如統計文章的PV（即時瀏覽量）等。

當要給列值+1或其他增量值時，在傳統設計中（即沒有原子計數器情況下），您需要執行Read-Modify-Write(RMW)操作。首先您需要從服務端讀取列的舊值，然後在用戶端完成列值的修改，最後將修改後的列值寫回服務端。此時，在並發多用戶端修改同一行時會引發一致性的問題。

針對一致性問題，目前的解決方案是：通過一個行級事務，在執行RMW前，首先通過啟動該行的一個行事務獲得行鎖，然後在這個事務內完成RMW的操作。通過該方案可以實現單行多用戶端修改的強一致性，但是原子計數器寫入效能將會受到較大影響。RMW實際對應兩輪網路請求操作，因此會導致比較大的效能開銷。

為瞭解決由強一致性導致的寫入效能開銷的問題，我們在Table Store中引入了原子計數器。一個RMW操作，通過一次網路請求發送到後端伺服器，後端伺服器使用內部行鎖機制在本地完成RMW的操作。通過原子計數器將分散式運算器的計算邏輯下推到後端伺服器，在保證強一致性的情況下，提升原子計數器的寫入效能。

介面說明

RowUpdateChange類新增原子計數器介面：

- RowUpdateChange increment(Column column)：對列執行增量變更（如：+X，-X等）。
- void addReturnColumn(String columnName)：對於涉及原子加的列，設定需要返回列值的列名。
- void setReturnType(ReturnType.RT_AFTER_MODIFY)：設定返回標誌，返回指定的原子加列的新值。

寫入資料時，使用RowUpdateChange介面對整型列做列值的增量變更，如下所示：

```
private static void incrementByUpdateRowApi(SyncClient client) {
        // 構造主鍵
        PrimaryKeyBuilder primaryKeyBuilder = PrimaryKeyBuilder.
createPrimaryKeyBuilder();
        primaryKeyBuilder.addPrimaryKeyColumn(PRIMARY_KEY_NAME,
PrimaryKeyValue.fromString("pk0"));
        PrimaryKey primaryKey = primaryKeyBuilder.build();

        RowUpdateChange rowUpdateChange = new RowUpdateChange(
TABLE_NAME, primaryKey);

        // 將price列值+10，不允許設定時間戳記
        rowUpdateChange.increment(new Column("price", ColumnValue.
fromLong(10)));

        // 設定ReturnType將原子計數器的結果返回
```

```
        rowUpdateChange.addReturnColumn("price");
        rowUpdateChange.setReturnType(ReturnType.RT_AFTER_MODIFY);

        // 對price列發起原子計數器操作
        UpdateRowResponse response = client.updateRow(new UpdateRowR
equest(rowUpdateChange));

        // 列印出更新後的新值
        Row row = response.getRow();
        System.out.println(row);
    }
```

说明：

- RowUpdateChange.addReturnColumn(列名)：設定需要返回的列。
- RowUpdateChange.setReturnType(RT_AFTER_MODIFY：指定本次修改需要返回上述設定的列值。

使用情境

您可以使用原子計數器對某一行中的資料做即時統計。假設某一使用者需要使用Table Store圖片meta並統計圖片數資訊，表格內每一行對應某使用者ID，行上的其中一列用於儲存上傳的圖片，另一列用於即時統計上傳的圖片數。

- UpdateRow：增加一張新圖片，原子計數器+1。
- UpdateRow：刪除一張舊圖片，原子計數器-1。
- GetRow：讀取原子計數器的值，擷取目前使用者的圖片數。

上述行為具有強一致性，即當您增加一張新圖片時，原子計數器會相應+1，而不會出現-1的情況。

限制

原子計數器主要存在以下幾方面的限制：

- 僅支援Integer類型。
- 作為原子計數器的列，若寫入前該列不存在，則預設值為0。若寫入前該列已存在且列值為非Integer類型，則拋出OTSParameterInvalid錯誤。
- 增量值可以是正數或負數，但不允許計算溢出。若出現計算溢出，則拋出OTSParameterInvalid錯誤。
- 預設不返回原子加列的結果。可以通過addReturnColumn() + setReturnType()介面，指定返回哪些原子加列的結果。
- 在單次更新要求時，不能對某一列同時進行更新和原子計數的操作。假設列A已經執行原子計數器操作，則列A不能再執行其他動作（如列的覆蓋寫，列刪除等）。

- 在一次 BatchWriteRow 請求中，可以支援對同一行的多次更新操作。但若某一行已執行原子計數器操作，則該行在此批量請求中只能出現一次。
- 列上的原子計數器只作用在最新版本上，不支援對列的特定版本做原子計數器。更新完成後，原子計數器會插入一個新的版本。
- 原子計數器操作可能會由於網路逾時、系統錯誤等導致失敗。此時，該應用程式只需重試該操作即可。這會產生更新兩次計數器的風險，導致計數有可能偏多或偏少。針對此類異常情境，建議使用####精確變更列值。

6 Stream增量資料流

6.1 概述

Table Store Stream 是一個資料通道，用於擷取 Table Store 表中的增量資料。

您可以使用 Table Store Stream API 來擷取這些修改內容。增量資料的重要性不言而喻，有了增量資料，可以進行增量資料流的即時增量分析、資料增量同步處理等。

原理

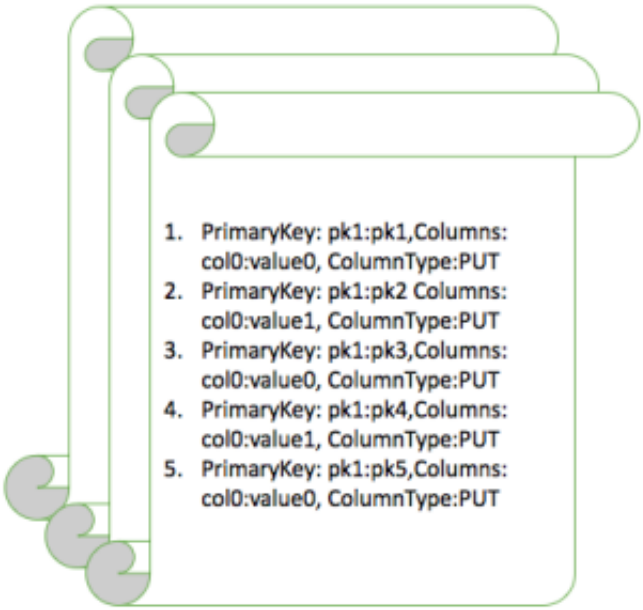
Table Store作為一款分布式 NoSQL 資料庫，當有寫操作（包括 put，delete，update）傳入時，會把對應的修改記錄存放在Table Store的 commit log 中，同時資料庫也會定期做 checkpoint，舊的 commit 日誌會被清除。

開啟 Stream後，記錄檔會被儲存。在設定的儲存期限內，可以通過 Stream 提供的通道讀取這些增量資料。

由於Table Store的分區特性，同一分區的操作會共用一個 commit log，所以擷取增量資料也是按照分區維度擷取。

開啟 Stream 時，會產生一個當前日誌位移量（即 iterator）並記錄下來。使用者可以通過 `GetShardIterator` 介面擷取當前分區的 iterator，在後續讀取該分區增量資料時傳入這個 iterator，Stream 通道就可以知道從哪一行日誌記錄開始返回增量內容。返回增量內容的同時，Stream 也會返回一個新的位移量，用於後續的讀取。整個過程可以類比我們分頁讀取資料，iterator 就是分頁的位移量。

例如，我們有順序地產生一批資料庫記錄檔，如下所示：



當我們在檔案 A，第 3 行開始開啟 Stream，則這個 iterator 就可以用來標識檔案 A 的第 3 行。讀取資料時傳入這個 iterator，就能讀到從上圖中第三個操作 pk3 開始的後續修改操作。

Stream API 也提供了介面關閉這個資料通道。當使用者再次開啟時，Stream 會根據本次開啟時間的日誌位移量重建一個當前分區的 iterator，我們可以用這個 iterator 讀取該分區目前時間點之後的增量資料。

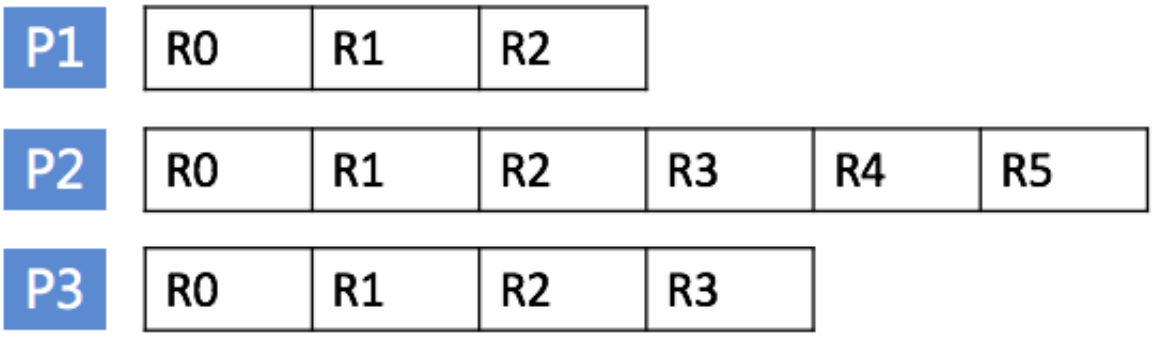
由於寫操作會發生在同一個主鍵上，為了確保資料的一致性，對同一個主鍵的寫操作需要順序讀取。然而，在讀取增量資料之前，我們並不知道在哪些主鍵上發生了修改，所以讀取增量資料的介面按照分區 id 來劃分。使用者可以通過羅列當前表下的所有分區來讀取整張表的增量資料。Stream 通道會保證同分區內的寫操作是順序返回的，即，只要在同一分區內按照順序讀取，就可以保證相同主鍵的資料的一致性。如果持續讀取所有分區的 Stream 資料，也就確保了整張表的增量資料都被會被讀取到。

使用者可以在建立表的時候開啟 Stream，或者通過 `UpdateTable` 來開啟或者關閉 Stream。當有一個 `put`，`update` 或者 `delete` 操作發生，一條修改記錄會被寫入 Stream，資料包含使用者修改行的主鍵以及修改內容。

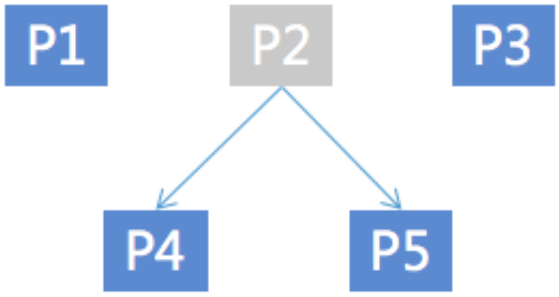
 说明：

- 每個修改記錄在 Stream 中存在且僅存在一次。
- 在每一個 shard 內，Stream 按照使用者的實際操作順序進行修改。但是不同 shard 的資料，不保證順序。

樣本



如上圖所示，當前表有三個分區。圖中的每一行表示一個分區，每一列表示當前分區的一次更新操作。每個分區維護自己的更新記錄。我們可以通過 `DescribeStream` 介面得到分區的資訊，然後順序讀取各自分區的資料。但系統會根據負載進行分區的分裂或者合并。分區的合并、分裂操作會產生新的分區，老的分區也將不會再產生新的增量資料。



上圖中，分區 P2 發生了一次分裂，變成分區 P4 和分區 P5。我們可以並行讀取分區 P4 和分區 P5 的資料，並不互相影響。但是在開始讀取分區 P4 和分區 P5 之前，需要確保分區 P2 的增量資料已經全部讀取完畢。

例如上圖中，當開始讀取分區 P4 的記錄 R6 時，需要保證分區 P2 的 R5 已經被讀取完畢，當 R5 被讀完後，分區 P2 不會再產生新的資料。

6.2 Stream API/SDK

API

- 開啟/關閉 Steam

  使用者可以在建立表的時候設定 Stream 是否開啟，也可以通過 `UpdateTable` 來開啟或者關閉 Stream。`CreateTable` 和 `UpdateTable` 中新增了 `StreamSpecification` 參數，表示 Stream 的相關參數：

  - enable_stream：Stream 是否開啟。
  - expiration_time：Stream 資料的到期時間，較早的修改記錄將會被刪除。
- 讀取修改記錄

  讀取 Stream 資料的流程如下：

  1. 調用 `ListStreams` 擷取當前表的 Stream 資訊，例如 StreamID。詳細資料請參見*ListStream*。
  2. 調用 `DescribeStream` 擷取當前 Stream 的資料分區資訊，例如 shard 的列表，每個 shard 記錄又包含父 shard 資訊、shardID 資訊等。詳細資料請參見*DescribeStream*。
  3. 擷取 StreamID 和 shardID 後，通過 `GetShardIterator` 擷取當前 shard 的讀 iterator 值，這個值標記著讀取該 shard 記錄的起始位置。詳細資料請參見 *GetShardIterator*。

4. 調用 GetStreamRecord 來讀取具體的修改記錄，每次調用會返回新的 iterator，用於下次讀取。詳細資料請參見*GetStreamRecord*。

> 说明：
>
> - 同一個主鍵下的操作必須有序。在同一個 shard 下，Stream 做了這個保證。但是 shard 會出現分裂、合并等操作，所以您在讀取某個 shard 的資料時，需要確保他的父 shard 以及 parent_sibling 的資料已經被讀取。
> - 當讀到空的 NextShardIterator 的時候，說明當前 shard 的增量資料已經全部讀完，通常情況是該 shard 已經進入 inactive 的狀態（發生分裂或者合并）。當一個 shard 已經被全部讀完以後，您可以重新調用 DescribeStream 擷取新的 shard 資訊。

SDK

為了方便您使用 Stream API，Table Store 的 Java SDK 已經支援 Stream 介面。詳細資料請參見*Java SDK*。

6.3 Stream Client

基於 Table Store Stream API 以及 Table Store SDK，您可以使用 API 或者 SDK 讀取 Stream 的記錄。在即時擷取增量資料的時候，需要注意分區的資訊不是靜態。分區可能會分裂、合并。當分區發生變化後，需要處理分區之間的依賴關係以確保單主鍵上資料順序讀取。同時，如果您的資料是由多用戶端並發產生，為了提高匯出增量資料的效率，也需要有多消費端並行讀取各個分區的增量記錄。

Stream Client 可以解決 Stream 資料處理時的常見問題，例如，如何做負載平衡，故障恢複，Checkpoint，分區資訊同步確保分區資訊消費順序等。使用 Stream Client 後，您只需要關心每條記錄的處理邏輯即可。

下面內容將介紹 Stream Client 的原理，以及如何使用 Stream Client 高效打造適合自身業務的資料通道。

Stream Client 原理

為了方便做任務調度，以及記錄當前每個分區的讀取進度，Stream Client 使用了 Table Store 的一張表來記錄這些資訊，您可以自行選定表名，但是要確保該表名沒有其他業務在使用。

Stream Client 中為每個分區定義了一個租約（lease），每個租約的擁有者叫做 worker。租約用來記錄一個分區的增量資料消費者（即 worker）和讀取進度資訊。當一個新的消費端啟動後，worker 會初始化，檢查分區和租約資訊，為沒有相應租約的分區建立租約。當因為分裂或合并產生新的分

區時，Stream Client 會在表中插入一條租約記錄。新的記錄會被某一個 Stream Client 的 worker 搶到並不斷處理，如果有新的 worker 加入則可能會做負載平衡，調度至新 worker 處理。

租約記錄的 schema 如下所示：

參數	說明
主鍵 StreamId	當前處理 Stream 的 Id。
主鍵 StatusType	當前 lease 的 Key。
主鍵 StatusValue	當前 lease 對應的分區的 Id。
屬性 Checkpoint	記錄當前分區 Stream 資料消費的位置（使用者故障恢複）
屬性 LeaseCounter	表示樂觀鎖。每個 lease 的 owner 會持續更新這個 counter 值，用來續租表示繼續佔有當前的 lease。
屬性 LeaseOwner	擁有當前租約的 worker 名。
屬性 LeaseStealer	在負載平衡的時候，表示準備挪至哪個 worker。
屬性 ParentShardIds	當前 shard 的父分割資訊。在 worker 消費當前 shard 時，保證父分割的 stream 資料已經被消費。

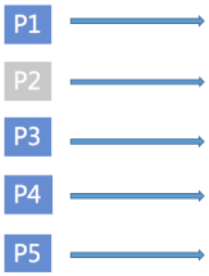
樣本

下圖是典型的使用 Stream Client 消費增量資料的分布式架構：

在這張圖中，worker1 和 worker2 是兩個基於 Stream Client 的消費端，例如在 ECS 上啟動的進程。資料來源不斷地讀寫 Table Store 中的表，這張表初期有分區 P1、P2 和 P3。隨著訪問量和資料量的增大，分區 P2 發生了一次分裂，產生了 P4 和 P5。worker1 在初始階段會消費 P1 的資

料，worker2 消費 P2 和 p3 的資料，當 P2 發生分裂後，新的分區 P4 會被分配給 worker1，分區 P5 分配給 worker2。但 Stream Client 會確保 P2 的 R5 記錄已經被消費完成後，再開始消費 P4 和 P5 的資料。如果這時部署了一個新的消費端 worker3，那可能發生的一件事情是 worker2 上的某個分區會被 worker3 調度走，由此產生負載平衡。

在上述情境下，Stream Client 會在表中產生如下租約資訊：



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

Stream Client 中的 worker 是抽象消費 Stream 資料的載體，每一個分區會被分配一個 worker 即 lease 的擁有者，擁有者會不斷地通過心跳，即更新 LeaseCounter 來續約當前的分區租約，通常每一個 Stream 消費端會起一個 worker，worker 在完成初始化後擷取當前需要處理的分區資訊，同時 worker 會維護自己的線程池，並發地迴圈拉取所擁有的各分區的增量資料。worker 初始化流程如下：

1. 讀取 Table Store 的配置，對內部訪問 Table Store 的用戶端進行初始化。
2. 擷取對應表的 Stream 資訊，並初始化租約管理類。租約管理類會同步租約資訊，為新分區建立租約記錄。
3. 初始化分區同步類，分區同步類會維持當前持有的分區心跳。
4. 迴圈擷取當前 worker 持有的分區的增量資料。

下載 **Stream Client**

- 下載安裝 *JAR #*
- Maven：

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

说明：

Stream Client 的代碼是開源的，您可以####瞭解原理，也歡迎您將基於 Stream 的優秀 Sample 代碼分享給我們。

使用 **Stream Client** 介面

為了方便您使用 Stream Client 消費 Stream 資料、隱藏式磁碟分割讀取邏輯和調度邏輯，Stream Client 提供了 `IRecordProcessor` 介面。Stream Client 的 worker 會在拉取到 stream 資料後調用 `processRecords` 函數來觸發使用者的處理資料邏輯。

```
public interface IRecordProcessor {
    void initialize(InitializationInput initializationInput);
    void processRecords(ProcessRecordsInput processRecordsInput);
    void shutdown(ShutdownInput shutdownInput);
}
```

參數解釋如下：

參數	說明
void initialize(InitializationInput initializationInput);	用於初始化一個讀取任務，表示 stream client 準備開始讀取某個 shard 的資料。
void processRecords(ProcessRecordsInput processRecordsInput);	表示具體讀取到資料後使用者希望如何處理這批記錄。在 `ProcessRecordsInput` 中有一個 `getCheckpointer` 函數可以得到一個 `IRecordProcessorCheckpointer`。這個介面是架構提供給使用者用來做 checkpoint 的介面，使用者可以自行決定多久做一次 checkpoint。
void shutdown(ShutdownInput shutdownInput);	表示結束某個 shard 的讀取任務。

说明：

- 因為讀取任務所在機器不同，進程可能會遇到各種類型的錯誤，例如因為環境因素重啟，需要定期對處理完的資料做記錄（checkpoint）。當任務重啟後，會接著上次的 checkpoint 繼續往後做。也就是說，在極端情況下，Stream Client 不保證 `ProcessRecordsInput` 裡傳給您的記錄只有一次，只會保證資料至少傳一次，且記錄的順序不變。如果出現局部資料重複發送的情況，需要您注意業務的處理邏輯。
- 如果您希望減少在出錯情況下資料的重複處理，可以增加做 checkpoint 的頻率。但是過於頻繁的 checkpoint 會降低系統的輸送量，請根據自身業務特點決定做 checkpoint 的頻率。

- 如果您發現增量資料不能及時被消費，可以增加消費端的資源，例如用更多的節點來讀取 stream 記錄。

下面給出了一個簡單的樣本，使用 Stream Client 來即時擷取增量資料，並在控制台輸出增量資料。

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

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

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

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

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

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

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

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

   class RecordProcessorFactory implements IRecordProcessorFactory {

       private final String workerIdentifier;
```

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

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

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

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

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

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

7 HBase 支援

7.1 Table Store HBase Client

除了使用現有的 SDK 以及 Restful API 來訪問Table Store，我們還提供了 Table Store HBase Client。使用開源 HBase API 的 JAVA 應用可以通過 Table Store HBase Client 來直接存取Table Store服務。Table Store HBase Client 基於Table Store 4.2.x 以上版本的 JAVA SDK，支援 1.x.x 版本以上的開源 HBase API。

Table Store HBase Client 可以從以下三個途徑擷取：

- GitHub：*tablestore-hbase-client ##*
- *#####*
- Maven

```
<dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
       </dependency>
   </dependencies>
```

由於Table Store是一個全託管的 NoSQL 資料庫服務，當使用 Table Store HBase Client 之後，使用者不再需要關心 HBase Server 的相關事項，只需要通過 Client 暴露出來的介面進行表或者資料的操作即可。

相比自行搭建 HBase 服務，Table Store有著如下的優勢：

	Table Store	自建HBase叢集
成本	根據實際用量進行計費，提供高效能與容量型兩種規格執行個體，適用於不同的應用情境。	需要根據業務峰值進行資源配置，空閑時段資源被閑置，租用及人工營運成本高。
安全	整合阿里雲 RAM 資源許可權管理系統，支援多種鑒權和授權機制及 VPC、主/子帳號功能，授權粒度達到表層級和 API 層級。	需要額外的安全機制。
可靠性	資料自動多重冗餘備份，故障遷移自動完成，可用性不低於 99.9%，資料可靠性達 99.99999999%。	需要自行保障叢集的可用性。

	Table Store	自建HBase叢集
可擴充性	Table Store的自動負載平衡機制支援單表 PB 級資料，即使百萬並發也無需任何人工擴容。	叢集利用率到一定水位之後需要繁瑣的機器上下線流程，影響線上業務。

7.2 Table Store HBase Client 支援的功能

Table Store與 HBase 的 API 區別

作為 NoSQL 資料庫服務，Table Store對使用者屏蔽了資料表分裂、Dump、Compact、Region Server 等底層相關的細節，使用者只需要關心資料的使用。因此，雖然在####及功能上相近，但Table Store並不完全與 HBase 相同，兩者在 API 上有所區別。也正因如此，Table Store Hbase Client 與原生的 HBase API 仍然有一些區別。

支援的功能

- CreateTable

  Table Store不支援列族（ColumnFamily），所有的資料可以認為是在同一個 ColumnFamily 之內，所以Table Store的 TTL 及 Max Versions 都是資料表層級的，支援如下相關功能：

功能	支援情況
family max version	支援表層級 max version，預設為 1
family min version	不支援
family ttl	支援表層級 TTL
is/set ReadOnly	通過 RAM 子帳號支援
預分區	不支援
blockcache	不支援
blocksize	不支援
BloomFilter	不支援
column max version	不支援
cell ttl	不支援
控制參數	不支援

- Put

功能	支援情況
一次寫入多列資料	支援

功能	支援情況
指定一個時間戳記	支援
如果不寫時間戳記，預設用系統時間	支援
單行 ACL	不支援
ttl	不支援
Cell Visibility	不支援
tag	不支援

- Get

  Table Store保證資料的強一致性，在資料寫入 API 收到 HTTP 200 狀態代碼（OK）的回複時，資料即被持久化到所有的備份上，這些資料能夠馬上被 Get 讀到。

功能	支援情況
讀取一行資料	支援
讀取一個列族裡面的所有列	支援
讀取特定列的資料	支援
讀取特定時間戳記的資料	支援
讀取特定個數版本的資料	支援
TimeRange	支援
ColumnfamilyTimeRange	不支援
RowOffsetPerColumnFamily	支援
MaxResultsPerColumnFamily	不支援
checkExistenceOnly	不支援
closestRowBefore	支援
attribute	不支援
cacheblock:true	支援
cacheblock:false	不支援
IsolationLevel:READ_COMMITTED	支援
IsolationLevel:READ_UNCOMMITTED	不支援
IsolationLevel:STRONG	支援
IsolationLevel:TIMELINE	不支援

- Scan

Table Store保證資料的強一致性，在資料寫入 API 收到 HTTP 200 狀態代碼（OK）的回複時，資料即被持久化到所有的備份上，這些資料能夠馬上被 Scan 讀到。

功能	支援情況
指定 start、stop 確定掃描範圍	支援
如果不指定掃描範圍，預設掃描全域	支援
prefix filter	支援
讀取邏輯同 Get	支援
逆序讀	支援
caching	支援
batch	不支援
maxResultSize，返回資料量大小的限制	不支援
small	不支援
batch	不支援
cacheblock:true	支援
cacheblock:false	不支援
IsolationLevel:READ_COMMITTED	支援
IsolationLevel:READ_UNCOMMITTED	不支援
IsolationLevel:STRONG	支援
IsolationLevel:TIMELINE	不支援
allowPartialResults	不支援

- Batch

功能	支援情況
Get	支援
Put	支援
Delete	支援
batchCallback	不支援

- Delete

功能	支援情況
刪除整行	支援

功能	支援情況
刪除特定列的所有版本	支援
刪除特定列的特定版本	支援
刪除特定列族	不支援
指定時間戳記時，deleteColumn 會刪除等於這個時間戳記的版本	支援
指定時間戳記時，deleteFamily 和 deleteColumns 會刪除小於等於這個時間戳記的所有版本	不支援
不指定時間戳記時，deleteColumn 會刪除最近的版本	不支援
不指定時間戳記時，deleteFamily 和 deleteColumns 會刪除當前系統時間的版本	不支援
addDeleteMarker	不支援

- checkAndXXX

功能	支援情況
CheckAndPut	支援
checkAndMutate	支援
CheckAndDelete	支援
檢查列的值是否滿足條件，滿足則刪除	支援
如果不指定值，則表示預設	支援
跨行，檢查 A 行，執行 B 行	不支援

- exist

功能	支援情況
判斷一行或多行是否存在，不返回內容	支援

- Filter

功能	支援情況
ColumnPaginationFilter	不支援 columnOffset 和 count
SingleColumnValueFilter	支援：LongComparator，BinaryComparator，ByteArrayComparable 不支援： RegexStringComparator，SubstringComparator，BitComparator

不支援的方法

- Namespaces

  Table Store上使用__執行個體__對資料表進行管理。執行個體是Table Store最小的計費單元，使用者可以在*Table Store###*上進行執行個體的管理，所以不再支援如下 Namespaces 相關的操作：

  — createNamespace(NamespaceDescriptor descriptor)
  — deleteNamespace(String name)
  — getNamespaceDescriptor(String name)
  — listNamespaceDescriptors()
  — listTableDescriptorsByNamespace(String name)
  — listTableNamesByNamespace(String name)
  — modifyNamespace(NamespaceDescriptor descriptor)

- Region 管理

  Table Store中資料存放區和管理的基本單位為####，Table Store會自動地根據資料分區的資料大小、訪問情況進行分裂或者合并，所以不支援 HBase 中 Region 管理相關的方法。

- Snapshots

  Table Store目前不支援 Snapshots，所以暫時不支援 Snapshots 相關的方法。

- Table 管理

  Table Store會自動對 Table 下的資料分區進行分裂、合并及 Compact 等操作，所以不再支援如下方法：

  — getTableDescriptor(TableName tableName)
  — compact(TableName tableName)
  — compact(TableName tableName, byte[] columnFamily)
  — flush(TableName tableName)
  — getCompactionState(TableName tableName)
  — majorCompact(TableName tableName)
  — majorCompact(TableName tableName, byte[] columnFamily)
  — modifyTable(TableName tableName, HTableDescriptor htd)
  — split(TableName tableName)
  — split(TableName tableName, byte[] splitPoint)

- Coprocessors

Table Store暫時不支援副處理器，所以不支援如下方法：

- coprocessorService()
- coprocessorService(ServerName serverName)
- getMasterCoprocessors()

- Distributed procedures

  Table Store不支援 Distributed procedures，所以不支援如下方法：

  - execProcedure(String signature, String instance, Map props)
  - execProcedureWithRet(String signature, String instance, Map props)
  - isProcedureFinished(String signature, String instance, Map props)

- Increment 與 Append

  暫不支援原子增減和原子 Append。

7.3 Table Store和 HBase 的區別

Table Store HBase Client 的使用方式與 HBase 類似，但存在一些區別。本節內容介紹 Table Store HBase Client 的特點。

Table

不支援多列族，只支援單列族。

Row和Cell

- 不支援設定 ACL
- 不支援設定 Cell Visibility
- 不支援設定 Tag

GET

Table Store只支援單列族，所以不支援列族相關的介面，包括：

- setColumnFamilyTimeRange(byte[] cf, long minStamp, long maxStamp)
- setMaxResultsPerColumnFamily(int limit)
- setRowOffsetPerColumnFamily(int offset)

SCAN

類似於 GET，既不支援列族相關的介面，也不能設定最佳化類的部分介面，包括：

- setBatch(int batch)

- setMaxResultSize(long maxResultSize)
- setAllowPartialResults(boolean allowPartialResults)
- setLoadColumnFamiliesOnDemand(boolean value)
- setSmall(boolean small)

Batch

暫時不支援 BatchCallback。

Mutations 和 Deletions

- 不支援刪除特定列族
- 不支援刪除最新時間戳記的版本
- 不支援刪除小於某個時間戳記的所有版本

Increment 和 Append

暫時不支援

Filter

- 支援 ColumnPaginationFilter
- 支援 FilterList
- 部分支援 SingleColumnValueFilter，比較子僅支援 BinaryComparator
- 其他 Filter 暫時都不支援

Optimization

HBase 的部分介面涉及到訪問、儲存最佳化等，這些介面目前沒有開放：

- blockcache：預設為 true，不允許使用者更改
- blocksize：預設為 64K，不允許使用者更改
- IsolationLevel：預設為 READ_COMMITTED，不允許使用者更改
- Consistency：預設為 STRONG，不允許使用者更改

Admin

HBase 中的介面`org.apache.hadoop.hbase.client.Admin`主要是指管控類的 API，而其中大部分的 API 在Table Store中是不需要的。

由於Table Store是雲端服務，營運、管控類的操作都會被自動執行，使用者不需要關注。其他一些少量介面，目前暫不支援。

- CreateTable

Table Store只支援單列族，在建立表時只允許設定一個列族，列族中支援 MaxVersion 和 TimeToLive 兩個參數。

- Maintenance task

  在Table Store中，下列的任務維護相關介面都會被自動處理：

  - abort(String why, Throwable e)
  - balancer()
  - enableCatalogJanitor(boolean enable)
  - getMasterInfoPort()
  - isCatalogJanitorEnabled()
  - rollWALWriter(ServerName serverName) -runCatalogScan()
  - setBalancerRunning(boolean on, boolean synchronous)
  - updateConfiguration(ServerName serverName)
  - updateConfiguration()
  - stopMaster()
  - shutdown()

- Namespaces

  在Table Store中，執行個體名稱類似於 HBase 中的 Namespaces，因此不支援 Namespaces 相關的介面，包括：

  - createNamespace(NamespaceDescriptor descriptor)
  - modifyNamespace(NamespaceDescriptor descriptor)
  - getNamespaceDescriptor(String name)
  - listNamespaceDescriptors()
  - listTableDescriptorsByNamespace(String name)
  - listTableNamesByNamespace(String name)
  - deleteNamespace(String name)

- Region

  Table Store中會自動處理 Region 相關的操作，因此不支援以下介面：

  - assign(byte[] regionName)
  - closeRegion(byte[] regionname, String serverName)
  - closeRegion(ServerName sn, HRegionInfo hri)
  - closeRegion(String regionname, String serverName)

- closeRegionWithEncodedRegionName(String encodedRegionName, String serverName)
- compactRegion(byte[] regionName)
- compactRegion(byte[] regionName, byte[] columnFamily)
- compactRegionServer(ServerName sn, boolean major)
- flushRegion(byte[] regionName)
- getAlterStatus(byte[] tableName)
- getAlterStatus(TableName tableName)
- getCompactionStateForRegion(byte[] regionName)
- getOnlineRegions(ServerName sn)
- majorCompactRegion(byte[] regionName)
- majorCompactRegion(byte[] regionName, byte[] columnFamily)
- mergeRegions(byte[] encodedNameOfRegionA, byte[] encodedNameOfRegionB, boolean forcible)
- move(byte[] encodedRegionName, byte[] destServerName)
- offline(byte[] regionName)
- splitRegion(byte[] regionName)
- splitRegion(byte[] regionName, byte[] splitPoint)
- stopRegionServer(String hostnamePort)
- unassign(byte[] regionName, boolean force)

Snapshots

不支援 Snapshots 相關的介面。

Replication

不支援 Replication 相關的介面。

Coprocessors

不支援 Coprocessors 相關的介面。

Distributed procedures

不支援 Distributed procedures 相關的介面。

Table management

Table Store自動執行 Table 相關的操作，使用者無需關注，因此不支援以下介面：

- compact(TableName tableName)

- compact(TableName tableName, byte[] columnFamily)
- flush(TableName tableName)
- getCompactionState(TableName tableName)
- majorCompact(TableName tableName)
- majorCompact(TableName tableName, byte[] columnFamily)
- modifyTable(TableName tableName, HTableDescriptor htd)
- split(TableName tableName)
- split(TableName tableName, byte[] splitPoint)

限制項

Table Store是雲端服務，為了整體效能最優，對部分參數做了限制，且不支援使用者通過配置修改，具體限制項請參見*Table Store###*。

7.4 從 HBase 遷移到Table Store

Table Store HBase Client 是基於 HBase Client 的封裝，使用方法和 HBase Client 基本一致，但是也有一些事項需要注意。

依賴

Table Store HBase Client 1.2.0 版本中依賴了 HBase Client 1.2.0 版本和 Table Store JAVA SDK 4.2.1 版本。pom.xml 配置如下：

```
<dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
       </dependency>
   </dependencies>
```

如果需要使用其他版本的 HBase Client 或 Table Store JAVA SDK，可以使用 exclusion 標籤。下面樣本中使用 HBase Client 1.2.1 版本和 Table Store JAVA SDK 4.2.0 版本。

```
  <dependencies>
       <dependency>
           <groupId>com.aliyun.openservices</groupId>
           <artifactId>tablestore-hbase-client</artifactId>
           <version>1.2.0</version>
           <exclusions>
               <exclusion>
                   <groupId>com.aliyun.openservices</groupId>
                   <artifactId>tablestore</artifactId>
               </exclusion>
               <exclusion>
                   <groupId>org.apache.hbase</groupId>
                   <artifactId>hbase-client</artifactId>
```

```
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
            <version>1.2.1</version>
        </dependency>
        <dependency>
            <groupId>com.aliyun.openservices</groupId>
            <artifactId>tablestore</artifactId>
            <classifier>jar-with-dependencies</classifier>
            <version>4.2.0</version>
        </dependency>
    </dependencies>
```

HBase Client 1.2.x 和 其他版本（如 1.1.x）存在介面變化，而 Table Store HBase Client 1.2.x 版本只能相容 HBase Client 1.2.x。

如果需要使用 HBase Client 1.1.x 版本，請使用 Table Store HBase Client 1.1.x 版本。

如果需要使用 HBase Client 0.x.x 版本，請參考####### *HBase*。

設定檔

從 HBase Client 遷移到 Table Store HBase Client，需要在設定檔中修改以下兩點。

- HBase Connection類型

  Connection 需要配置為 TableStoreConnection。

  ```
      <property>
          <name>hbase.client.connection.impl</name>
          <value>com.alicloud.tablestore.hbase.TablestoreConnection</
  value>
      </property>
  ```

- Table Store的配置項

  Table Store是雲端服務，提供了嚴格的許可權管理。要訪問Table Store，需要配置秘鑰等資訊。

  ▬ 必須配置以下四個配置項才能成功訪問Table Store：

  ```
  <property>
    <name>tablestore.client.endpoint</name>
    <value></value>
   </property>
   <property>
    <name>tablestore.client.instancename</name>
    <value></value>
   </property>
   <property>
    <name>tablestore.client.accesskeyid</name>
    <value></value>
   </property>
  ```

```
<property>
 <name>tablestore.client.accesskeysecret</name>
 <value></value>
</property>
```

- 下面為可選配置項：

```
<property>
 <name>hbase.client.tablestore.family</name>
 <value>f1</value>
</property>
<property>
 <name>hbase.client.tablestore.family.$tablename</name>
 <value>f2</value>
</property>
<property>
 <name>tablestore.client.max.connections</name>
 <value>300</value>
</property>
<property>
 <name>tablestore.client.socket.timeout</name>
 <value>15000</value>
</property>
<property>
 <name>tablestore.client.connection.timeout</name>
 <value>15000</value>
</property>
<property>
 <name>tablestore.client.operation.timeout</name>
 <value>2147483647</value>
</property>
<property>
 <name>tablestore.client.retries</name>
 <value>3</value>
</property>
```

- hbase.client.tablestore.family 與 hbase.client.tablestore.family.$tablename
  - Table Store只支援單列族，使用 HBase API 時，需要有一項 family 的內容，因此通過配置來填充此項 family 的內容。

    其中，`hbase.client.tablestore.family`為全域配置，`hbase.client.tablestore.family.$tablename`為單個表的配置。
  - 規則為：對錶名為 T 的表，先找`hbase.client.tablestore.family.T`，如果不存在則找`hbase.client.tablestore.family`，如果依然不存在則取預設值 f。
- tablestore.client.max.connections

  最大連結數，預設是 300。
- tablestore.client.socket.timeout

  Socket 逾時時間，預設是 15 秒。
- tablestore.client.connection.timeout

連線逾時時間，預設是 15 秒。

- tablestore.client.operation.timeout

  API 逾時時間，預設是 Integer.MAX_VALUE，類似於永不逾時。

- tablestore.client.retries

  請求失敗時，重試次數，預設是 3 次。

7.5 遷移較早版本的 HBase

Table Store HBase Client 目前支援 HBase Client 1.0.0 及以上版本的 API。

HBase Client 1.0.0 版本相對於之前版本有一些較大的變化，這些變化是不相容的。

為了協助一些使用老版本 HBase 的使用者能方便地使用Table Store，本節我們將介紹 HBase 1.0 相較於舊版本的一些較大變化，以及如何使其相容。

Connection 介面

HBase 1.0.0 及以上的版本中廢除了 HConnection 介面，並推薦使用`org.apache.hadoop.hbase.client.ConnectionFactory`類，建立一個實現 Connection 介面的類，用 ConnectionFactory 取代已經廢棄的 ConnectionManager 和 HConnectionManager。

建立一個 Connection 的代價比較大，但 Connection 是安全執行緒的。使用時可以在程式中只產生一個 Connection 對象，多個線程可以共用這一個對象。

HBase 1.0.0 及以上的版本中，使用者需要管理 Connection 的生命週期，並在使用完以後將它 close。

最新的代碼如下所示：

```
Connection connection = ConnectionFactory.createConnection(config);
// ...
connection.close();
```

TableName 類

1.0.0 之前版本的 HBase 中，使用者在建立表時可以使用 String 類型的表名，但是 1.0.0 之後需要使用類 `org.apache.hadoop.hbase.TableName`。

最新的代碼如下所示：

```
String tableName = "MyTable";
// or byte[] tableName = Bytes.toBytes("MyTable");
```

```
TableName tableNameObj = TableName.valueOf(tableName);
```

Table，**BufferedMutator** 和 **RegionLocator** 介面

從 HBase Client 1.0.0 開始，HTable 介面已經廢棄，取而代之的是 Table、BufferedMutator 和 RegionLocator 三個介面。

- `org.apache.hadoop.hbase.client.Table`：用於操作單張表的讀寫等請求
- `org.apache.hadoop.hbase.client.BufferedMutator`：用於非同步批量寫，對應於舊版本 HTableInterface 介面中的`setAutoFlush(boolean)`
- `org.apache.hadoop.hbase.client.RegionLocator`：表分區資訊

Table、BufferedMutator 和 RegionLocator 三個介面都不是安全執行緒的，但比較輕量，可以為每個線程建立一個對象。

Admin 介面

從 HBase Client 1.0.0 開始，HBaseAdmin 類被新介面`org.apache.hadoop.hbase.client.Admin`取代。由於Table Store是一個雲端服務，大多數營運類介面都是自動處理的，所以 Admin 介面中的眾多介面都不會被支援，具體區別請參見*Table Store# HBase ###*。

通過 Connection 執行個體建立 Admin 執行個體：

```
Admin admin = connection.getAdmin();
```

7.6 Hello World

本節描述如何使用 Table Store HBase Client 實現一個簡單的 Hello World 程式，主要包括下列操作：

- 配置依賴
- 串連Table Store
- 建立表
- 寫資料
- 讀資料
- 掃描資料
- 刪表

代碼位置

當前樣本程式使用了 HBase API 訪問Table Store服務，完整的樣本程式位於 Github 的*hbase* 項目中，目錄位置是 src/test/java/samples/HelloWorld.java。

使用 HBase API

- 設定項目依賴

  Maven 的依賴配置如下：

```
   <dependencies>
        <dependency>
            <groupId>com.aliyun.openservices</groupId>
            <artifactId>tablestore-hbase-client</artifactId>
            <version>1.2.0</version>
        </dependency>
    </dependencies>
```

  更進階的配置請參考*# HBase ###Table Store*。

- 設定檔

  hbase-site.xml 中增加下列配置項：

```
<configuration>
    <property>
        <name>hbase.client.connection.impl</name>
        <value>com.alicloud.tablestore.hbase.TablestoreConnection</
value>
    </property>
    <property>
        <name>tablestore.client.endpoint</name>
        <value>endpoint</value>
    </property>
    <property>
        <name>tablestore.client.instancename</name>
        <value>instance_name</value>
    </property>
    <property>
        <name>tablestore.client.accesskeyid</name>
        <value>access_key_id</value>
    </property>
    <property>
        <name>tablestore.client.accesskeysecret</name>
        <value>access_key_secret</value>
    </property>
    <property>
        <name>hbase.client.tablestore.family</name>
        <value>f1</value>
    </property>
    <property>
        <name>hbase.client.tablestore.table</name>
        <value>ots_adaptor</value>
    </property>
</configuration>
```

  更進階的配置請參考*# HBase ###Table Store*。

- 串連Table Store

通過建立一個 TableStoreConnection 對象來連結資料表格儲存服務。

```
  Configuration config = HBaseConfiguration.create();

  // 建立一個 Tablestore Connection
  Connection connection = ConnectionFactory.createConnection(config
);

  // Admin 負責建立、管理、刪除等
        Admin admin = connection.getAdmin();
```

- 建立表

  通過指定表名建立一張表，MaxVersion 和 TimeToLive 使用預設值。

```
  // 建立一個 HTableDescriptor，只有一個列族
  HTableDescriptor descriptor = new HTableDescriptor(TableName.
valueOf(TABLE_NAME));

  // 建立一個列族，MaxVersion 和 TimeToLive使用預設值，MaxVersion 預設值
是 1，TimeToLive 預設值是 Integer.INF_MAX
  descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));

  // 通過 Admin 的 createTable 介面建立表
  System.out.println("Create table " + descriptor.getNameAsString
());
  admin.createTable(descriptor);
```

- 寫資料

  寫入一行資料到Table Store。

```
  // 建立一個 TablestoreTable，用於單個表上的讀寫更新刪除等操作
  Table table = connection.getTable(TableName.valueOf(TABLE_NAME));

  // 建立一個 Put 對象，主鍵是 row_1
  System.out.println("Write one row to the table");
        Put put = new Put(ROW_KEY);

  // 增加一列，Table Store只支援單列族，列族名稱在 hbase-site.xml 中配置，
如果沒有配置則預設是“f”，所以寫入資料時 COLUMN_FAMILY_NAME 可以是空值
  put.addColumn(COLUMN_FAMILY_NAME, COLUMN_NAME, COLUMN_VALUE);

  // 執行 Table 的 put 操作，使用 HBase API 將這一行資料寫入Table Store
        table.put(put);
```

- 讀資料

  讀取指定行的資料。

```
  // 建立一個 Get 對象，讀取主鍵為 ROW_KEY 的行
  Result getResult = table.get(new Get(ROW_KEY));
  Result result = table.get(get);

  // 列印結果
  String value = Bytes.toString(getResult.getValue(COLUMN_FAM
ILY_NAME, COLUMN_NAME));
  System.out.println("Get one row by row key");
```

```
   System.out.printf("\t%s = %s\n", Bytes.toString(ROW_KEY), value);
```

- 掃描資料

  範圍讀取資料。

```
掃描全表所有行資料
System.out.println("Scan for all rows:");
Scan scan = new Scan();

ResultScanner scanner = table.getScanner(scan);

// 迴圈列印結果
for (Result row : scanner) {
 byte[] valueBytes = row.getValue(COLUMN_FAMILY_NAME, COLUMN_NAME);
 System.out.println('\t' + Bytes.toString(valueBytes));
}
```

- 刪表

  使用 Admin API 刪除一張表。

```
   print("Delete the table");
   admin.disableTable(table.getName());
   admin.deleteTable(table.getName());
```

完整代碼

```
package samples;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

import java.io.IOException;

public class HelloWorld {

    private static final byte[] TABLE_NAME = Bytes.toBytes("HelloTable
store");
    private static final byte[] ROW_KEY = Bytes.toBytes("row_1");
    private static final byte[] COLUMN_FAMILY_NAME = Bytes.toBytes("f
");
    private static final byte[] COLUMN_NAME = Bytes.toBytes("col_1");
    private static final byte[] COLUMN_VALUE = Bytes.toBytes("
col_value");

    public static void main(String[] args) {
        helloWorld();
    }

    private static void helloWorld() {

        try  {
            Configuration config = HBaseConfiguration.create();
```

```
            Connection connection = ConnectionFactory.createConnection
(config);
            Admin admin = connection.getAdmin();

            HTableDescriptor descriptor = new HTableDescriptor(
TableName.valueOf(TABLE_NAME));
            descriptor.addFamily(new HColumnDescriptor(COLUMN_FAM
ILY_NAME));

            System.out.println("Create table " + descriptor.getNameAsS
tring());
            admin.createTable(descriptor);

            Table table = connection.getTable(TableName.valueOf(
TABLE_NAME));

            System.out.println("Write one row to the table");
            Put put = new Put(ROW_KEY);
            put.addColumn(COLUMN_FAMILY_NAME, COLUMN_NAME, COLUMN_VAL
UE);
            table.put(put);

            Result getResult = table.get(new Get(ROW_KEY));
            String value = Bytes.toString(getResult.getValue(
COLUMN_FAMILY_NAME, COLUMN_NAME));
            System.out.println("Get a one row by row key");
            System.out.printf("\t%s = %s\n", Bytes.toString(ROW_KEY),
value);

            Scan scan = new Scan();

            System.out.println("Scan for all rows:");
            ResultScanner scanner = table.getScanner(scan);
            for (Result row : scanner) {
                byte[] valueBytes = row.getValue(COLUMN_FAMILY_NAME,
COLUMN_NAME);
                System.out.println('\t' + Bytes.toString(valueBytes));
            }

            System.out.println("Delete the table");
            admin.disableTable(table.getName());
            admin.deleteTable(table.getName());

            table.close();
            admin.close();
            connection.close();
        } catch (IOException e) {
            System.err.println("Exception while running HelloTable
store: " + e.toString());
            System.exit(1);
        }
    }
}
```