TRANSWARP
DATA HUB

Transwarp Data Hub Version 4.3.3
Hyperbase使用手册

星环信息科技（上海）有限公司

Transwarp Data Hub v4.3.3

Hyperbase使用手册

版本：1.0v

发布日期： 2016-03-26

版本号：T001433-04-010

免责声明

本说明书依据现有信息制作，其内容如有更改，恕不另行通知。星环信息科技（上海）有限公司在编写该说明书的时候已尽最大努力保证期内容准确可靠，但星环信息科技（上海）有限公司不对本说明书中的遗漏、不准确或印刷错误导致的损失和损害承担责任。具体产品使用请以实际使用为准。

注释：Hadoop® 和 SPARK® 是Apache™ 软件基金会在美国和其他国家的商标或注册的商标。 Java® 是Oracle公司在美国和其他国家的商标或注册的商标。 Intel® 和Xeon® 是英特尔公司在美国、中国和其他国家的商标或注册的商标。

修订历史记录

文档版本T001433-04-010 (2016-03) 第一次发布。

文档版本T001430-04-010 (2016-01) 第一次发布。

文档版本T001420-04-011 (2015-12) 第一次发布。

文档版本T001420-04-010 (2015-11) 第二次发布。

文档版本T001420-04-010 (2015-06) 第一次发布。

目录

表格清单

范例清单

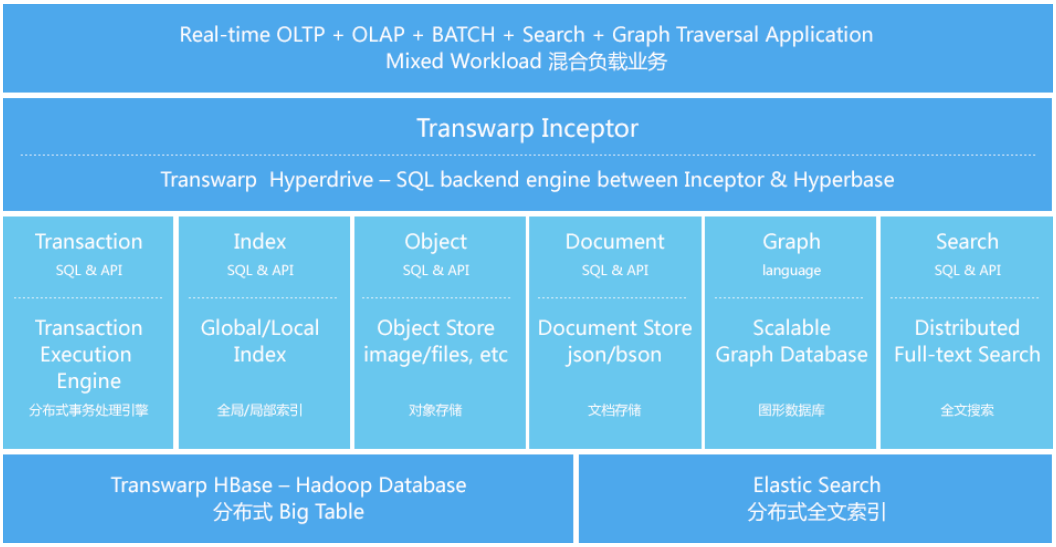
1. 简介

1.1. 公司介绍

星环信息科技(上海)有限公司是目前中国国内极少数掌握企业级大数据Hadoop和Spark核心技术的高科技公司，从事大数据时代核心平台数据库软件的研发与服务。Apache Hadoop技术已成为公认的替代传统数据库的大数据产品。公司产品Transwarp Data Hub (TDH)的整体架构及功能特性比肩硅谷同行，产品性能在业界处于领先水平。

1.2. Transwarp Hyberbase介绍

Transwarp Hyperbase实时数据库是建立在Apache HBase基础之上，融合了多种索引技术、分布式事务处理、全文实时搜索、图形数据库在内的实时NoSQL数据库。Hyperbase可以高效地支持企业的在线OLTP应用、高并发OLAP应用、批处理应用、全文搜索或高并发图形数据库检索应用，结合Inceptor 高速SQL引擎，是企业创建可扩展在线运营数据库（Operational Database）或者实时分析型数据库(ODS - Operational Data Store)的最佳选择。



1.3. Transwarp Hyperbase 功能特性

- SQL支持

  通过Inceptor支持采用SQL进行批处理和高并发查询，批处理比Map/Reduce快10倍。可从Hyperbase的行存储转换成Holodesk的列存储，同时支持在线查询和高速OLAP分析。

- 索引

  支持全局、局部、高维索引和高级过滤器，可用于高并发低延时的OLAP查询。

- CRUD

  支持通过SQL高并发毫秒级数据插入 / 修改 / 查询 / 删除。

- **多数据类型支持**

  支持文档型数据（如JSON/BSON)的存储、索引和搜索，支持对象数据（图片、音视频、二进制文档等）的存储、检索和自动回收。

2. Hyperbase Shell快速入门

在安装了Hyperbase的节点上执行 `hbase shell` 操作可以进入Hyperbase命令行和Hyperbase进行简单交互。本章中，我们介绍如何通过Hyperbase Shell进行一些简单操作，包括建表，填入数据和删除数据。

和Inceptor命令行不同，Hyperbase的命令行指令区分大小写，例如 `create` 指令不能写成 `CREATE`。

2.1. 建表

语法：create

```
create '<table>', '<column_family>' [, '<column_family>', ...]
```

现在我们以先前章节中的银行用户表为例建表：

```
create 'bi', 'ps', 'ct', 'bl'
```

说明：表、列族和列限定符名都要尽量短，以减少对Hyperbase读写时的I/O负载。所以我们将bank_info, personal, contact, balance缩短为bi, ps, ct和bl。建表以后，可以通过describe语句查看表的描述：

```
describe 'bi'
DESCRIPTION
 ENABLED
 'bi',
 {NAME => 'bl', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
 REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1', TTL => 'FOR true
 EVER', MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536',
 IN_MEMORY => 'false', BLOCKCACHE => 'true'},
 {NAME => 'ct', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
 REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1', TTL => 'FOREVER',
 MIN_VERSIONS => '0', KEE
 P_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE
 => 'true'},
 {NAME => 'ps', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
 REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1', TTL => 'FOREVER',
 MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'false', BLOCKSIZE => '65536', IN_MEMORY
 => 'false', BLOCKCACHE => 'true'}
```

如我们所见，在表的描述中，列族有很多个参数，列族与列族间的参数由“{}”分隔开。我们将在之后的章节中解释其中重要的部分。建表时，可以通过直接指定列族参数来定义列族。所有的参数中只有NAME必须指定，其余参数为可选。所以上面的建表语句也可以像下面这样写：

```
hbase(main):001:0> create 'bi', {NAME=>'ps'}, {'NAME'=>'ct'}, {'NAME'=>'bl'}
```

2.2. 填入数据

下面，我们用put指令向表内填入数据：

语法：put

```
put '<table>', '<row_key>', '<column_family:column_qualifier>', '<cell_value>' [,
 <timestamp>]
```

说明 ：在put语句末尾的timestamp是可选的。如果用户不指定timestamp，Hyperbase将根据put的时间给出一个timestamp。我们建议用户不要自己定义timestamp。

举例

```
put 'bi', '0001', 'ps:nm', 'Zhang San'
put 'bi', '0001', 'ps:pw', '1234'
put 'bi', '0001', 'ct:em', 'zs@mail.com'
put 'bi', '0001', 'ct:cp', '12345678912'
put 'bi', '0001', 'bl:bl', '10000.00'
put 'bi', '0002', 'ps:nm', 'Li Si'
put 'bi', '0002', 'ps:pw', '2468'
put 'bi', '0002', 'ct:em', 'ls@school.edu'
put 'bi', '0002', 'ct:cp', '13513572468'
put 'bi', '0002', 'bl:bl', '1000.00'
put 'bi', '0003', 'ps:nm', 'Wang Wu'
put 'bi', '0003', 'ps:pw', '1357'
put 'bi', '0003', 'ct:em', 'ww@hmail.com'
put 'bi', '0003', 'ct:cp', '13612345678'
put 'bi', '0003', 'bl:bl', '500.00'
put 'bi', '0001', 'ps:pw', '5678'          ❶
put 'bi', '0002', 'ct:em', 'ls@transwarp.io'          ❷
put 'bi', '0002', 'bl:bl', 56000          ❸
```

❶ 用户Zhang San修改密码
❷ 用户Li Si修改邮箱
❸ 用户Li Si的存款发生改变

数据全部填写完成后可以用 scan 来查看表中数据

```
scan 'bi'
ROW                             COLUMN+CELL
 0001                           column=bl:bl, timestamp=1422973263541, value=10000.00
 0001                           column=ct:cp, timestamp=1422973240271,
 value=12345678912
 0001                           column=ct:em, timestamp=1422973219713,
 value=zs@mail.com
 0001                           column=ps:nm, timestamp=1422973047378, value=Zhang
 San
 0001                           column=ps:pw, timestamp=1422973504458, value=5678
 0002                           column=bl:bl, timestamp=1422973543652, value=56000
 0002                           column=ct:cp, timestamp=1422973339893,
 value=13513572468
```

```
0002                            column=ct:em, timestamp=1422973526791,
value=ls@transwarp.io
0002                            column=ps:nm, timestamp=1422973288836, value=Li Si
0002                            column=ps:pw, timestamp=1422973299902, value=2468
0003                            column=bl:bl, timestamp=1422973464663, value=500.00
0003                            column=ct:cp, timestamp=1422973432593,
value=13612345678
0003                            column=ct:em, timestamp=1422973416242,
value=ww@hmail.com
0003                            column=ps:nm, timestamp=1422973393169, value=Wang Wu
0003                            column=ps:pw, timestamp=1422973405207, value=1357
```

注意 ， scan时，Hyperbase只会显示单元格值中拥有最新timestamp的版本。

2.3. 删除数据

2.3.1. 删除单元格

为表删除一个单元格中的数据用 `delete` 指令，删除时，需要指定表名、row key，列族和列限定符。

语法：delete

```
delete '<table>', '<row_key>', '<column_family:column_qualifier>'
```

2.3.2. 删除整行数据

将一行中的数据全部删除用deleteall指令，执行时，需要指定表名和row key。

语法：deleteall

```
deleteall '<table_name>', '<row_key>'
```

2.3.3. 删除一个列族

删除一整个列族用以下两个指令，效果相同。

语法：alter … 'delete'…

```
alter '<table>', 'delete'=>'<column_family>'
alter '<table>', NAME => '<column_family>', METHOD => 'delete'
```

举例：删除bi表中的bl列族

```
alter 'bi', 'delete' => 'bl'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
```

2.4. 添加一个列族

建表后，我们还是可以向表添加新的列族，此时需要直接指定列族参数,只有NAME为必选参数，其余参数可选。

语法

```
alter '<table>', {NAME=>'<column_family>',...}
```

举例 下例为bi表添加一个名为bl的列族。

```
alter 'bi', {NAME=>'bl'}
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
```

2.5. 删除表

在删除一张表之前，要先用 `disable` 将这张表下线：

语法：disable

```
disable '<table>'
```

disable之后才能执行删除命令：

语法：drop

```
drop '<table>'
```

举例 我们先试着删除一张名为’ta’的表：

```
drop 'ta'
ERROR: Table ta is enabled. Disable it first.'
Here is some help for this command:
Drop the named table. Table must first be disabled:
  hbase> drop 't1'
  hbase> drop 'ns1:t1'
```

Hyperbase 报错，要先disable表。

```
disable 'ta'
drop 'ta'
```

删除完成。在disable一张表以后，该表即不能使用，无法进行scan, put等操作。重新使用该表，要使用enable命令：

语法：enable

```
enable '<table>'
```

3. 数据模型

3.1. 简介

Hyperbase以“表”为结构来组织数据，“表”中有“行”和“列”，但是Hyperbase中的这些概念和关系型数据库的二维表不同。在谷歌的Bigtable论文中，Bigtable被描述为一个“稀疏的，分布式的，持久的，多维度有序map”。下面，我们围绕这个描述来解说Hyperbase中的数据模型。

Hyperbase中图表对象如下：

- **表（Table）**：Hyperbase以表为单位组织数据。表名的数据类型为string。
- **行（Row）**：：表中数据以行存储。每行数据都有一个独特的RowKey。表中各行数据按RowKey排序。Row key没有数据类型，以byte[]（字节数组）存储。
- **列族（Column Family）**：行中数据以列族分组。各行数据拥有的列族必须相同。但是并不是每个列族中都需要有数据。列族名的数据类型为string。
- **列限定符（Column Qualifier）**：列族中可以有一列或者多列数据。各列根据列限定符识别。各行的拥有的列不一定需要相同。列名没有数据类型，以byte[]存储。
- **单元格（Cell）**：行、列族和列限定符的组合指向独特的单元格。单元格中存放的数据成为单元格的值。单元格的值没有数据类型，以byte[]存储。
- **时间戳（Timestamp）**：单元格的值可以有不同版本。各个版本由版本号区分。默认版本号为单元格值被写入时的时间戳。

为了更好地理解这些概念，下面我们举一个Hyperbase表的例子。

3.2. 例子

Row Key（Account Number）	Column Family - Personal		Column Family - Contact		Column Family - Balance	Timestamp
	column qualifier - name	column qualifier - password	column qualifier - email	column qualifier - cellphone	column qualifier - balance	
0001		5678				t16
					10000.00	t05
				12345678912		t04
			zs@mail.com			t03
		1234				t02
	Zhang San					t01
0002					56000.00	t18
			ls@transwarp.io			t17
					1000.00	t10
				13513572468		t09
			ls@school.edu			t08
		2468				t07
	Li Si					t06
0003					500.00	t15
				13612345678		t14
			ww@hmail.com			t13
		1357				t12
	Wang Wu					t11

该表模拟银行用户表中的部分信息。表的Row Key为账户号码Account Number；各行以账户号码排序。该表有三个列族：Personal，Contact和Balance。Personal列族含有两列，它们的列限定符为name和password。Contact列族含有两列，它们的列限定符为email和cellphone。Balance列族含有一列，它的列限定符为balance。RowKey为0001的行有两个版本，时间戳分别为t3和t1。RowKey为0002的行有三个版本，时间戳分别为t8，t5，t2。RowKey为0003的行有一个版本，时间戳为t6。不同版本的单元格值按时间戳降序排列，方便读取最新的值。表中无值的单元格在系统中是不存在的，表的“稀疏性”就体现在这里。

在Hyperbase中，表不是二维的，而是一个嵌套的map，由多层 **键值对** 组成：

```
{Table: {RowKey: {ColumnFamily: {ColumnQualifier: {Timestamp, Value}}}}}
```

所以，bank_info表可以如下表示：

```
bank_info:{
    0001:{
        Personal: {
            name: {t1: Zhang San}
            password: {t3: 5678
                       t1: 1234}
                 }
        Contact: {
            email: {t1: zs@mail.com}
            cellphone: {t1: 12345678912}
                 }
        Balance: {
            balance: {t1: 10000.00}
                 }
          }
    0002: {
        Personal: {
            name: {t2: Li Si}
            password: {t2: 2468}
                 }
        Contact: {
            email: {t5: ls@transwarp.io
                    t2: ls@school.edu
                   }
            cellphone: {t2: 13513572468}
                 }
        Balance: {
            balance: {t8: 56000.00
                      t2: 1000.00}
                 }
          }
    0003: {
        Personal: {
            name: {t6: Wang Wu}
            password: {t6: 1357}
                 }
        Contact: {
            email: {t6: ww@hmail.com}
            cellphone: {t6: 13612345678}
                 }
        Balance: {
```

```
    balance: {t6: 500.00}
         }
  }
 }
```

4. Hyperbase Shell基本命令

登陆集群中的服务器，执行 `hbase shell` 操作可以进入Hyperbase命令行和Hyperbase进行简单交互。本章中，我们介绍Hyperbase Shell中的各种指令。在Hyperbase Shell中执行命令需要注意以下几点：

1. 所有名字如表名和列名都必须使用单引号进行引用：如 `'table1'`, `'key1'`。
2. 创建和修改表的配置时使用的是Ruby Hashes，如 `{'key1' # 'value1', 'key2' # 'value2', …}`。 它需用 `{` 和 `}` 表明整个对象的开始和结尾，每对key，value之间通过逗号分隔，key和value之间通过 `=>` 分隔。
3. 如果想输入二进制的数值，需用 **双引号** 进行引用，并且使用16进制表示法，如：

```
get 't1', "key\x03\x3f\xcd"
get 't1', "key\003\023\011"
put 't1', "test\xef\xff", 'f1:', "\x01\x33\x40"
```

Hyperbase Shell基本命令大致可以分成6类：表管理命令、数据操作命令、namespace相关命令、通用命令、索引命令和授权命令。 本章会详细解释前5类命令，授权指令将在第 5 章 Hyperbase安全中介绍。如需要了解某个命令的细节可以输入 `help '<command>'`，如 `help 'create'`。

4.1. Hyperbase Shell表管理命令

可以在Hyperbase Shell中执行 `help 'ddl'` 来查看帮助。

4.1.1. list

语法

```
list ['<regex>']
```

列出Hyperbase中所有的表，可以选择列出和正则表达式 `<regex>` 匹配的表。

例 4.1. list

```
list
list 'abc.*'
```

4.1.2. create

语法

```
create '<table>', {NAME => '<column_family>' [, ...]} [, {...}, ...]
```

建表时至少要指定一个列族，列族名通过 `NAME => '<column_family>'` 指定，在建表同时还可以设置列族的其他元数据。如果建表时要创建多个列族，不同列族的元信息要用 `{}` 隔开。

例 4.2. 建表

```
create 't4', {NAME => 'f1'}, {NAME => 'f2'}
```

该建表语句可以简写为：

```
create 't4', 'f1', 'f2'
```

4.1.3. describe

语法

```
describe '<table>'
```

查看 `<table>` 的元数据。

例 4.3. describe

```
describe 'bi'
```

4.1.4. exists

语法

```
exists '<table>'
```

查看 `<table>` 是否存在。

例 4.4. exists

```
exists 'bi'
```

4.1.5. show_filters

语法

```
show_filters
```

列出Hyperbase中所有的filter。

4.1.6. enable

语法

```
enable '<table>'
```

将指定表上线。

4.1.7. disable

语法

```
disable '<table>'
```

将指定表下线。

4.1.8. enable_all

语法

```
enable_all
```

将所有表上线。

4.1.9. is_enabled

语法

```
is_enabled '<table>'
```

查看 <table> 是否上线。

4.1.10. disable_all

语法

```
disable_all
```

将所有表下线。

4.1.11. is_disabled

语法

```
is_disabled '<table>'
```

查看 `<table>` 是否下线。

4.1.12. drop

语法

```
drop '<table>'
```

删除指定表。**注意**，只有下线的表才能被删除。

4.1.13. drop_all

语法

```
drop_all '<regex>'
```

将所有表名和指定正则表达式 `<regex>` 匹配的表删除。

例 4.5. drop_all

```
drop_all 't.*'
```

4.1.14. alter

语法

```
alter '<table>', {<PROPERTY_1> => <value_1>, <PROPERTY_2> => <value_2>, ...},
 {...}, ...
```

`alter` 可以用于添加和修改表或表中列族的元数据，可以同时修改多个列簇。使用 `<PROPERTY> => <value>` 的方式设置表或列族的属性。如果只修改一组元数据，可以不加 `{}`；如果同时修改多组元数据（例如修改多个列族元数据），每组属性需要放在不同的 `{}` 中。

例 4.6. 用 alter 修改列族属性

将表t1中的列簇f1改为常驻内存，并将列簇f2的版本改为5：

```
alter 't1', {NAME => 'f1', IN_MEMORY => true}, {NAME => 'f2', VERSION => 5}
```

例 4.7. 用 alter 删除列族

删除表中某个列族：

```
alter 't1', NAME => 'f1', METHOD => 'delete'
```

上面的指令可以简写为：

```
alter 't1', 'delete' => 'f1'
```

4.1.15. alter_async 和 alter_status

alter_async 和 alter 语法相同，但是语义和 alter 略有不同。alter_async 指令可以立即返回，而 alter 需要等到所有的region都更新完成后才会返回。而要查看更新期间所有regions的更新进度，可以使用 alter_status 命令.

4.2. Hyperbase Shell数据操作命令

可以在Hyperbase Shell中执行 help 'dml' 来获取帮助。

4.2.1. count

语法

```
count '<table>' [, CACHE => <n>, INTERVAL => <m>]
```

返回指定表中的行数。默认情况下该命令每次只获取一行，因此对于一张大表来说该命令会执行得很慢。可以通过设置 CACHE 参数来增加每次获取的行数，从而加速该命令的执行. 还可以指定查询到多少行显示一次 count 结果，默认值是1000行，可以通过 INTERVAL 参数进行修改.

例 4.8. count

```
count 't2'
count 't2', CACHE => 1000, INTERVAL => 10000
```

4.2.2. put

语法

```
put '<table>', '<row_key>', '<column_family:column_qualifier>', '<cell_value>',
 [<timestamp>]
```

将值 <cell_value> 填入指定的表（<table>）、行（<row_key>）和列（<column_family:column_qualifier>）对应的单元格中。时间戳 <timestamp> 为可选项。

例 4.9. put

```
put 't4', '1', 'f1:q1', 'a'
```

4.2.3. get

语法

```
get '<table>', '<row_key>' [{<PROPERTY> => <value>, ...}]
```

获取指定的表和行中的数据。可以通过 <PROPERTY> # <value> 指定某些属性来过滤获取的结果。例如 COLUMN # 'column_family:column_qualifier' 指定只获取某一列中的数据。可指定的属性有：COLUMN, TIMESTAMP, TIMERANGE, VERSIONS 和 FILTER。

例 4.10. get

```
get 't4', '1'
get 't4', '1', {COLUMN => 'f1:q1'}
```

4.2.4. scan

语法

```
scan '<table>', [{<PROPERTY> => <value>, ...}]
```

扫描指定的表，批量获取表中数据。可以通过 <PROPERTY> # <value> 指定某些属性来过滤获取的结果。可指定的属性有：TIMERANGE, FILTER, LIMIT, STARTROW, STOPROW, TIMESTAMP, MAXLENGTH, COLUMNS 和 CACHE。

例 4.11. scan

```
scan 't4'
scan 't4', {COLUMNS => 'f1:q1'}
```

4.2.5. delete

语法

```
delete '<table>', '<row_key>', '<column_family:column_qualifier>' [, <timestamp>]
```

删除指定的表（<table>）、行（<row_key>）和列（<column_family:column_qualifier>）对应的单元格。可以加上 <timestamp> 选项指定删除某个时间戳对应的数据。

例 4.12. delete

```
delete 't4','1', 'f1:q1'
```

4.2.6. deleteall

语法

```
deleteall '<table>', '<row_key>' [, '<column_family:column_qualifier>', <timestamp>]
```

删除指定表、指定行中全部的数据。可以加上 '<column_family:column_qualifier>' 和 <timestamp> 选项指定删除某个列或时间戳对应的数据。

例 4.13. deleteall

```
deleteall 't4', '2'
```

4.2.7. truncate 和 truncate_preserve

语法

```
truncate '<table>'
trucnate_preserve '<table>'
```

truncate 类似 delete，但该命令会立即删除表中所有的数据以及region的划分. 它的内部实现是将指定的表下线，删除，并重建. 如果只想立即删除表中所有的数据而不想丢掉原来的region划分，需要使用 truncate_preserve。

4.3. Hyperbase Shell Namespace相关命令

可以在Hyperbase Shell中执行 help 'namespace' 来获取帮助。

4.3.1. alter_namespace

语法

```
alter_namespace '<namespace>', {METHOD => 'set|unset', <PROPERTY> => <value>, ...}
```

修改namespace属性。METHOD 选择 set 为添加或设置属性；选择 unset 为删除属性。

4.3.2. create_namespace

语法

```
create_namespace '<namespace>' [, {<PROPERTY_1> => <value_1>, ...}]
```

创建一个namespace，并可以通过 <PROPERTY_1> => '<value>' 额外指定namespace的属性。

4.3.3. describe_namespace

语法

```
describe_namespace '<namespace>'
```

查看 <namespace> 的元数据。

4.3.4. drop_namespace

语法

```
drop_namespace '<namespace>'
```

删除指定的namespace。 要删除的namespace必须是一个空的namespace，不能存在表。

4.3.5. list_namespace

语法

```
list_namespace ['<regex>']
```

列出Hyperbase中所有的namespace，可以加上正则表达式 <regex> 来对结果进行匹配.

例 4.14. list_namespace

```
list_namespace
list_namespace 'abc.*'
```

4.3.6. list_namespace_tables

语法

```
list_namespace_tables '<namespace>'
```

列出指定的namespace下所有的表。

4.4. Hyperbase Shell索引命令

查询中，当我们需要经常对某一列进行过滤，对这列生成索引可以提高查询效率。在Hyperbase中，您可以用表中的一列或多列建索引。Hyperbase中的索引有四种：全局索引（global index），局部索引（local index），LOB索引（lob index）和全文索引（fulltext index）。全局的索引与与原表独立,以一张表（ **索引表** ）形式存在；局部的索引就在原表中，以一个新的列（ **索引列** ）的形式存在；LOB索引是Object Store中LOB列专用的索引，是LOB列所在表中的一列；而全文索引是一张与原表独立的ES表。

普通的Hyperbase表支持全局索引、LOB索引和全文索引。TDH4.5及以后的版本中新增的Hyperdrive表则支持全部四种索引。本节我们将介绍如何使用Hyperbase Shell指令为普通Hyperbase表创建、生成和删除全局索引。LOB索引的操作需要使用API（请参考第 7 章 Object Store使用方法）或者JSON配置（请参考第 11.6 节 “lob模块”）。全文索引的操作需要使用JSON配置（请参考第 11.3 节 “fulltextindex模块”）。

4.4.1. 创建全局索引

语法

```
add_index '<table>', '<global_index_name>', '<index_definition>'
```

该语句为指定表 `<table>` 添加全局索引，生成一张索引表，索引表的表名将是 `table_global_index_name`（注意区分索引名和索引表名）。`<index_definition>` 为索引的定义，用于指示Hyperbase如何建索引。

索引的定义形式如下：

```
COMBINE_INDEX|INDEXED= <cf>:<cq>:<n>[|<cf>:<cq>:<n>|...]|rowKey:rowKey:<m>,
[UPDATE=true]
```

说明

- `<cf>:<cq>:<n>` 指定生成索引所用的列以及索引长度：`<cf>` 是该列所在的列族，`<cq>` 是该列的列限定符，`<n>` 是索引字段在索引词条中的长度。例如 `f1:q1:n1` 为表中每一行记录用 `f1:q1` 列的数据生成的一段长度为n1字段放在索引词条中。如果指定的 `f1:q1` 列的值长度不足n1，Hyperbase将用0表示空格将长度补足。如果长度超过n1，超过的部分将在索引词条的末尾添上。
- 如果需要使用多列生成索引，可以在定义中添加多组 `<cf>:<cq>:<n>`，组之间用 `|` 隔开。
- 每个生成的索引词条中都会包含一段原表Row Key生成的长度为m的字段，由 `rowKey:rowKey:<m>` 指定。
- `[UPDATE=true]`为可选项，代表在创建索引之前会通过索引列查询数据，如果能查到，就删除前面的索引，重新建索引。

例 4.15. 用单列创建全局索引

对bi表用 ps:nm 列创建名为 psnmindex 的索引。索引词条中 ps:nm 对应字段长度为8，rowKey对应字段长度为10。创建的索引表表名为 bi_psnmindex。

```
add_index 'bi','psnmindex','COMBINE_INDEX|INDEXED=ps:nm:8|rowKey:rowKey:10'
```

例 4.16. 用多列创建全局索引

对bi表用 ps:nm 和 ps:pw 两列创建名为 psnmpwindex 的索引。索引词条中 ps:nm 对应字段长度为8，ps:pw 对应字段长度为9，rowKey对应字段长度为10。创建的索引表表名为 bi_psnmpwindex。

```
add_index 'bi','psnmpwindex','COMBINE_INDEX|INDEXED=ps:nm:8|ps:pw:9|
rowKey:rowKey:10,UPDATE=true'
```

4.4.2. 生成全局索引

第 4.4.1 节 “创建全局索引”创建一张空的索引表，没有索引词条，索引词条在两种情况下生成：

1. 当原表中有新的数据插入时，Hyperbase会 **自动** 为新增数据生成索引词条，写入索引表中。
2. 用户执行 rebuild_global_index 让Hyperbase为原表中数据生成索引词条，写入索引表中。

使用 rebuild_global_index 生成全局索引时要提供表名和索引名。

语法

```
rebuild_global_index '<table>','<index_name>'
```

例 4.17. 生成全局索引

用 rebuild_global_index 生成前文创建的全局索引。

```
rebuild_global_index 'bi', 'psnmindex'
```

4.4.3. 查看全局索引

全局索引在一张表中，我们可以用普通查看表的命令 scan 来查看全局索引所在的索引表。

例 4.18. 查看全局索引

通过 scan 查看例 4.17 “生成全局索引”中生成的全局索引。

```
scan 'bi_psnmindex'
```

4.4.4. 删除全局索引

虽然全局索引存在一张表中，我们不能像删除普通Hyperbase表一样（先 disable 然后 drop）删除它，因为这样Zookeeper还会保留索引表的信息。要彻底删除索引，必须使用删除全局索引专用的命令 delete_global_index。

语法

```
delete_index '<table>', '<index_name>'
```

例 4.19. 删除全局索引

删除例 4.15 “用单列创建全局索引”中建的全局索引：

```
delete_global_index 'bi', 'psnmindex'
```

4.5. Hyperbase Shell通用命令

可以在Hyperbase Shell中执行 help 'general' 来获取帮助。

4.5.1. status

语法

```
status ['<options>']
```

查看集群状态。

Options

- `summary `(default)：总结状态
- simple：简单状态
- detailed：详细状态

4.5.2. version

语法

```
version
```

输出Hyperbase版本。

4.5.3. whoami

语法

```
whoami
```

输出当前Hyperbase用户。Hyperbase根据认证方式识别用户身份。如果Hyperbase使用简单认证模式（除服务器操作系统认证以外没有认证系统），那么当前Hyperbase用户身份和用户在操作系统上的身份一致。如果Hyperbase使用Kerberos认证，用户需要先持Kerberos principal+密码通过Kerberos认证，那么当前Hyperbase用户身份和她的Kerberos principal中的第一个字段一致。例如alice@TDH和alice/admin@TDH都会被Hyperbase识别为用户alice。

例 4.20. whoami

简单认证模式

```
whoami
alice (auth:SIMPLE)
    groups: alice
```

Kerberos认证模式

```
whoami
alice@TDH (auth:KERBEROS)
    groups: alice
```

5. Hyperbase安全

5.1. Hyperbase的认证

安全模式下，Hyperbase通过Kerberos认证。所以在使用Hyperbase Shell之前，您需要确保您的机器上有有效的Kerberos Ticket。如果你的机器上没有有效的Ticket，您需要进行获取。获取指令为：

```
kinit <principal>
```

您需要有能够通过Kerberos认证的用户名（principal）和密码才能获取ticket。这些信息需要您向您的集群管理员索取。

5.2. Hyperbase的权限管理

所以Hyperbase通过用户的principal判断用户身份并使用Access Control Labels （ACLs）在客户端进行授权管理。Hyperbase的超级用户为hbase，可以向其他用户授权。经hbase授权之前，一个普通用户在Hyperbase没有任何权限，她只能 `list` Hyperbase中的表，而不能进行诸如查看表内容、建表、修改表等操作，这些操作权限需要hbase授予。

在Hyperbase中一个用户可以拥有 `RWCA` 权限，即：

- `R (READ)`：读权限，用来进行 `get, scan, exists` 等读操作；
- `W (WRITE)`：写权限，用来进行 `put, delete` 等写操作；
- `X (EXEC)`：执行权限，用来执行coprocessor endpoint（高级操作）；
- `C (CREATE)`：建表权限，用来进行 `create, alter` 等操作；
- `A (ADMIN)`：管理员权限，用来进行 `enable, disable, grant, revoke` 等操作。

默认情况下，一个用户对自己建的表有全部 `RWCA` 权限。

对这些权限的赋予、收回和查看的语法分别如下：

- **授予权限：**

```
grant '<user|@group>', '<permissions>', ['<scope_specification>']
```

- **一次性收回所有权限：**

```
revoke '<user|@group>', ['<scope_specification>']
```

revoke用于批量地收回权限——将一个用户或用户组的 `RWXCA` 权限一次性收回。如果要单独收回某些权限，使用grant重新授予权限，新授的权限中不要包括要收回的权限即可（细节请参见后文）。

指定组时需要在组名前加“@”字符，如 `@groupx`。

- 查看（指定表的）权限：

```
user_permission ['<table>']
```

5.3. Hyperbase中权限作用的级别

Hyperbase提供不同粒度的访问控制，权限作用的 **级别** 由 `<scope_specification>` 指定。Hyperbase支持的权限作用级别如下所列（由高到低排列）：

1. Global：全局
2. Namespace (NS)：命名空间
3. Table：表
4. Column Family (CF)：列族
5. Column Qualifier (CQ)：列

权限从高到低继承，比如如果一个用户对一张表有R权限，那么他对表中的所有列族和列都有R权限。拥有global级别的权限意味着对Hyperbase中所有的命名空间、表、列族、列都有该权限。

`<scope_specification>` 需要按照级别顺序 **从左向右** 指定。

```
@<namespace> <table> <column family> <column qualifier>
```

5.4. Hyperbase中的“角色”

Hyperbase中没有数据库中常见的“角色”概念，而是通过对用户组权限的管理来实现批量授权。Hyperbase没有自己管理的用户组信息，而是将用户的身份映射到 **登陆Hyperbase命令行的服务器** 所在的操作系统上的用户并通过服务器操作系统上的用户组信息判断用户所在的组。映射方式如下：alice/instance@realm和alice@realm都会被映射到操作系统上的alice用户。此时，如果alice在操作系统中没有对应用户，那么alice对Hyperbase来说则没有组信息。

5.5. hbase:acl表

Hyperbase将权限控制信息存放在hbase:acl表中。表中的记录是用户对namespace、表、column family, column qualifier的权限。该表的结构如下：

表 5.1. hbase:acl 结构

Row	CF	CQ	Value
table/@namespace	l	user/@group	permissions
table	l	user/@group,cf	permissions
table	l	user/@group,cf,cq	permissions

其中：

- Row：Hyperbase中的表名或namespace名
- Column Family (CF)：只有一个列族，名为“l”
- Column Qualifier (CQ)：如果row是表名，那么CQ由用户名/用户组名，row对应的表的column family(cf), column qualifier(cq)组成；如果row是namespace名，那么CQ是用户名。
- Value：权限

我们可以用scan指令查看一张实际的hbase:acl中的内容，注意，只有有全局权限：

```
hbase(main):017:0> scan 'hbase:acl'
ROW                                                  COLUMN+CELL
 @ns1                                                column=l:alice, timestamp=1449080847180, value=R
 bi                                                  column=l:alice,ct,cp, timestamp=1448999267124, value=R
 bi1                                                 column=l:usera, timestamp=1446067150397, value=RWXCA
 hbase:acl                                           column=l:hive, timestamp=1445441376184, value=RWC
 hbase:acl                                           column=l:usera, timestamp=1446567820203, value=RWC
 ns1:tb1                                             column=l:alice, timestamp=1449079911226, value=R
 ns1:tb1                                             column=l:hbase, timestamp=1448998136037, value=RWXCA
 ss                                                  column=l:hbase, timestamp=1445442094860, value=RWXCA
 stuff_infor                                         column=l:hive, timestamp=1445357734352, value=RWXCA
 tb                                                  column=l:usera, timestamp=1445524596994, value=RWXCA
 yy                                                  column=l:usera, timestamp=1446054553613, value=RWXCA
9 row(s) in 0.1090 seconds
```

注意表中ROW为hbase:acl的记录（如上图中红框里的记录），拥有对这张表的某个权限意味这这个权限的级别为global。

5.6. 权限的授予

下面我们介绍如何在Hyperbase中向用户和用户组授权。

- 指定用户组时需要在组名前加“@”字符来标识它是组名而不是用户名，如`@groupx`；
- 单独指定namespace时需要在namespace名前加“@”来标识它是namespace名而不是表名，如 `@ns1`；
- 指定namespace下的某张表用namespace:table，如 `ns1:tb1`。

5.6.1. Global（全局）权限的授予

作为管理员，您可以通过查看hbase:acl来查看一个用户的全局权限。hbase:acl表中的全局权限的row就是hbase:acl，意思为如果对hbase:acl有某个权限，则对全局都有该权限。

语法

```
grant <user|@group> <permissions>
```

例 5.1. 授予用户alice全局R权限

```
grant 'alice','R'
```

例 5.2. 授予用户alice全局RW权限

```
grant 'alice','RW'
```

例 5.3. 授予用户组groupx全局W权限

```
grant '@groupx','W'
```

5.6.2. Namespace（命名空间）权限的授予

语法

```
grant '<user|@group>' '<permissions>' '<@namespace>'
```

例 5.4. 授予用户alice对命名空间ns1的R权限

```
grant 'alice','R','@ns1'
```

例 5.5. 授予用户组groupx对命名空间ns1的C权限

```
grant '@groupx','C','@ns1'
```

5.6.3. Table（表）权限的授予

语法

```
grant '<user|@group>' '<permissions>' '<table>'
```

例 5.6. 授予用户alice对表bi的R权限

```
grant 'alice','R','bi'
```

例 5.7. 授予用户alice对命名空间ns1中的表tb1的WC权限

```
grant 'alice','WC','ns1:tb1'
```

例 5.8. 授予用户组groupx对表bi的R权限

```
grant '@groupx','R','bi'
```

5.6.4. 列族权限的授予

语法

```
grant '<user|@group>' '<permissions>' '<table>', '<column_family>'
```

例 5.9. 授予用户alice对表t4中列族f1的R权限

```
grant 'alice', 'R', 't4', 'f1'
```

5.6.5. 列权限的授予

语法

```
grant '<user|@group>' '<permissions>' '<table>', '<column_family>',
 '<column_qualifier>'
```

例 5.10. 授予用户alice对表t4中f2:q1列的R权限

```
grant 'alice', 'R', 't4', 'f2', 'q1'
```

5.7. 权限的收回

权限的收回有两种方式：

- 收回 RWXCA 中的指定权限：使用 grant 命令。
- 一次性收回所有 RWXCA 权限：使用 revoke 命令。

5.7.1. 收回指定权限

当您需要收回用户或用户组的指定权限，您只需要使用 grant 命令重新授权，在授权语句中不包含您需要收回的权限即可。

例 5.11. 从用户alice处收回表t4的 R 权限

我们先向用户alice授予表t4的所有 RWXCA 权限：

```
grant 'alice', 'RWXCA', 't4'
```

现在执行 `user_permission 't4'` 命令，描述alice对表t4权限的输出为：

```
alice          t4,,: [Permission: actions=READ,WRITE,EXEC,CREATE,ADMIN]
```

现在要从alice处收回t4上的 `R` 权限，我们需要执行：

```
grant 'alice', 'WXCA', 't4'
```

再一次执行 `user_permission 't4'` 命令，描述alice对表t4权限的输出会变为：

```
alice          t4,,: [Permission: actions=WRITE,EXEC,CREATE,ADMIN]
```

5.7.2. 一次性收回权限

使用 `revoke` 权限则可以一次性收回所有 `RWXCA` 权限。

语法

```
revoke '<user|@group>', ['<scope_specification>']
```

执行 `revoke` 后，指定用户或用户组将在 `<scope_specification>` 指定的级别上没有任何权限。

例 5.12. 收回用户alice对表bi的所有权限

```
revoke 'alice','t4'
```

5.8. 权限的查看

查看权限的方式有两种：

- 查看hbase:acl表中的信息
- 使用 `user_permission` 命令（只能查看表级和表级以下权限）

两种方式都需要有全局 `R` 权限才能操作。

5.8.1. 查看hbase:acl表

我们来查看一张样例hbase:acl表，并解释其中的一些记录：

```
ROW            COLUMN+CELL
```

```
@ns1            column=l:alice, timestamp=1457636462495, value=R          ❶
bi              column=l:hbase, timestamp=1453427488388, value=RWXCA      ❷
bl              column=l:hive, timestamp=1454513614929, value=RWXCA
hbase:acl       column=l:hive, timestamp=1457629405417, value=R           ❸
t1              column=l:hbase, timestamp=1457528928129, value=RWXCA
```

❶ alice用户对命名空间ns1有 R 权限
❷ hbase用户对表bi有 RWXCA 权限
❸ hive用户有全局 R 权限

5.8.2. user_permission 命令

语法

```
user_permission ['<table>']
```

`user_permission` 查看default命名空间中所有表的表级和表级以下权限。加上 `<table>` 可以只查看指定表的权限。

例 5.13. 查看命名空间ns1中的表tb1上的权限

```
user_permission 'ns1:tb1'
```

6. 在Inceptor中处理Hyperbase表

要在Inceptor中对Hyperbase表进行交互式查询，要先在Inceptor中建一张外表，然后将Hyperbase表通过映射建立和一张二维表的对应关系。映射时，只有最新版本的单元格值会被保存。

举例

将bank_info映射为二维表的逻辑如下图所示：

原表bi

Row Key	ps		ct		b	Timestamp
	nm	ps	em	cp	bl	
0001		5678				t16
					10000.00	t05
				12345678912		t04
			zs@mail.com			t03
		1234				t02
	Zhang San					t01
0002					56000.00	t18
			ls@transwarp.io			t17
					1000.00	t10
				13513572468		t09
			ls@school.edu			t08
		2468				t07
	Li Si					t06
0003					500.00	t15
				13612345678		t14
			ww@hmail.com			t13
		1357				t12
	Wang Wu					t11

映射表 bank_info

acc_num	name	password	email	cellphone	balance
0001	Zhang San	5678	zs@mail.com	12345678912	10000.00
0002	Li Si	2468	ls@transwarp.io	13513572468	56000.00
0003	Wang Wu	1357	ww@hmail.com	13612345678	500.00

语法

```
CREATE EXTERNAL TABLE table_name (row_key_column data_type,
 column_name_1 data_type, column_name_2 data_type, ...)
 STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler|
io.transwarp.hyperbase.HyperbaseStorageHandler' WITH SERDEPROPERTIES
 ("hbase.columns.mapping" = ":row_key_column, column_family:column_qualifer_1,
 column_family:column_qualifier_2, ...") TBLPROPERTIES ("hbase.table.name" =
 "hbase_table_name")
```

说明

- Inceptor表的第一列必须是对应Hyperbase表中的Row Key。
- Inceptor表中剩余的列将是对应Hyperbase表中的所有column_family:column_qualifier组合。
- (row_key_column data_type, column_name_1 data_type, column_name_2 data_type, …) 和 :row_key_column, column_family:column_qualifer_1, column_family:column_qualifier_2, … 中的各个字段有一一对应的关

系。也就是说`row_key_column`对应`:row_key_column`，`column_name_1`对应`column_family:column_qualifer_1`，`column_name_2`对应`column_family:column_qualifer_2`，以此类推。

- hbase_table_name是对应的Hyperbase表的表名。
- 通过这种方式在Inceptor中创建的表称为hbase_table_name的*映射表*。映射关系成立后，在Inceptor Shell中对映射表做的改动和在HBase Shell中对Hyperbase表做的改动都会同步体现在映射表和Hyperbase表中。
- 建映射表时，Hyperbase表中不需要有数据。事实上，我们只要在HBase Shell中定义了Hyperbase表，就可以在Inceptor中建它的映射表。在建映射表时，WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":row_key_column, column_family:column_qualifer_1, column_family:column_qualifier_2, …")子句同时也定义了Hyperbase表各列族中的列。
- 上面，我们说建映射表时，Hyperbase表中不需要有数据。但是在使用HyperbaseStorageHandler作为storage handler的情况下，在建映射表时，Hyperbase表中*不能*有数据。原因是在Hbase Shell中通过put指令写入表中的数据编码方式和HyperbaseStorageHandler的编码方式不同。建映射表时，如果Hyperbase表中已经有通过put写入过数据，这些数据的编码便不能和映射表的编码对应，映射表数据的写入就会发生错误。所以，当选择HyperbaseStorageHandler作为storage handler时，要在Hbase Shell中建表后，通过Inceptor Shell向表内填入数据，而不是在HBase Shell中使用put。事实上，Inceptor Shell的表达能力要远远强于HBase Shell，我们建议如果您的Hyperbase表是主要用途在SQL查询，在HBase Shell建表完成后，所有的操作都在Inceptor Shell中完成。在Inceptor Shell中，我们除了不能对映射表进行分区和分桶，其他Inceptor SQL对映射表都适用。此外，在Inceptor中，单条INSERT, UPDATE和DELETE不能对普通二维表使用。但是可以对Hyperbase映射表使用。

举例

下例是使用HBaseStorageHandler作为storage handler建映射表的例子。在“Hyperbase索引”章节会有使用HyperbaseStorageHandler做storage handler的例子。下面，我们为bi表建映射表：

```
transwarp> CREATE EXTERNAL TABLE bank_info (acc_num STRING, name STRING,
 password STRING, email STRING, cellphone STRING, balance DOUBLE)  STORED
 BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH SERDEPROPERTIES
 ("hbase.columns.mapping"=":key, ps:nm, ps:pw, ct:em, ct:cp, bl:bl") TBLPROPERTIES
 ("hbase.table.name"="bi");
```

让我们查看一下bank_info中的内容:

```
transwarp> select * from bank_info;
0001    Zhang San    5678    zs@mail.com     12345678912     10000.0
0002    Li Si        2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu      1357    ww@hmail.com    13612345678     500.0
```

和预想的一样，映射表中只有最新信息。 下面，我们可以在Inceptor中对bank_info使用Inceptor SQL查询：

举例

```
transwarp> SELECT * FROM bank_info WHERE acc_num = '0003';
0003    Wang Wu      1357    ww@hmail.com    13612345678     500.0
```

6.1. 在Inceptor中对映射表进行操作

在Inceptor中，我们可以对一张映射表做所有除了分区和分桶外所有的InceptorSQL操作。InceptorSQL操作强大而易用，我们建议除了建Hyperbase表以外，所有对表的操作在Inceptor Shell中进行。 如果一张表是映射表，我们还能对其执行一些普通Inceptor表不支持的操作，包括单条INSERT, UPDATE和DELETE。

单条INSERT:

语法

```
INSERT INTO table_name (column_name1, column_name2, ...) VALUES (value1, value2, ...)
```

举例

```
transwarp> INSERT INTO bank_info (acc_num, name, password, email, cellphone, balance)
 VALUES ('0004', 'Sun Liu', '2357', 'zl@qmail.com', '13509876543', 200.00);
transwarp> SELECT * FROM bank_info;
0001    Zhang San       6789    zs@mail.com     12345678912     10000.0
0002    Li Si   2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu 1357    ww@hmail.com    13612345678     500.0
0004    Sun Liu 2357    zl@qmail.com    13509876543     200.0
```

如果字段有struct类型，需要使用named_struct函数来实现。

举例

假设字段col1为struct类型：struct(col1_c1 : string, col1_c2 : string)， 那么需要进行如下操作：

```
INSERT INTO table_name (col1, col2, col3) VALUES (NAMED_STRUCT('col1_c1', 'value1',
 'col1_c2', 'value2'), 'value3', 'value4');
```

UPDATE

使用UPDATE语句可以在Inceptor Shell里对Hyperbase表进行更新。在Inceptor Shell中对映射表UPDATE有两种形式：

形式一：

语法

```
UPDATE table_name SET column_name = value, column_name = value, ... WHERE
 filter_condition;
```

举例

下例更新用户Sun Liu的密码：

```
transwarp> UPDATE bank_info SET password = '1123' WHERE acc_num = '0004';
transwarp> SELECT * FROM bank_info WHERE acc_num = '0004';
0004    Sun Liu 1123    zl@qmail.com    13509876543     200.0
```

形式二：

语法

```
UPDATE table_name SET (col1, col2, col3, ...) = (SELECT col1, col2, col3 FROM [from
 source]);
```

说明

- 括号中必须是一个完整的查询语句。
- 在SET后面的字段列表及查询语句中需要包含对应的rowkey字段（如col1）

举例

以下是一张有不同信息的银行用户表bank_info2:

```
transwarp> SELECT * FROM bank_info2;
0005    Zhou Qi 5813    zq@lmail.com    13497531246     3000.0
```

我们可以通过以下UPDATE语句将bank_info2中的信息并入bank_info中：

```
transwarp> UPDATE bank_info SET (acc_num, name, password, email, cellphone, balance)
 = (SELECT acc_num, name, password, email, cellphone, balance FROM bank_info2);
transwarp> SELECT * FROM bank_info;
0001    Zhang San       6789    zs@mail.com     12345678912     10000.0
0002    Li Si           2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu         1357    ww@hmail.com    13612345678     500.0
0004    Sun Liu         1123    zl@qmail.com    13509876543     200.0
0005    Zhou Qi         5813    zq@lmail.com    13497531246     3000.0
```

DELETE

语法

```
DELETE FROM table_name WHERE filter_condition
```

举例

我们可以用DELETE指令删除上面刚刚并入bank_info中的一条数据：

```
transwarp> DELETE FROM bank_info WHERE acc_num = '0005';
transwarp> SELECT * FROM bank_info;
0001    Zhang San       6789    zs@mail.com     12345678912     10000.0
```

```
0002    Li Si   2468    ls@transwarp.io 13513572468     56000.0
0003    Wang Wu 1357    ww@hmail.com    13612345678     500.0
0004    Sun Liu 1123    zl@qmail.com    13509876543     200.0
```

下面我们举一个在Inceptor Shell中的常见查询的例子：

JOIN

从Inceptor Shell我们还可以对两张Hyperbase表进行JOIN操作。我们在Hyperbase中建一张交易表jy，列族为ac，am；列为ac:ac，am:am。Rowkey是交易流水号。ac:ac记录进行交易的账户，am:am记录交易的金额，记录银行用户表账户的收入和支出。正数为收入，负数为支出。

```
hbase(main):023:0> create 'jy', 'ac','am'
hbase(main):026:0> put 'jy', 'a001', 'ac:ac', '0001'
hbase(main):027:0> put 'jy', 'a001', 'am:am', '-2000'
hbase(main):028:0> put 'jy', 'a002', 'ac:ac', '10000'
hbase(main):029:0> put 'jy', 'a002', 'ac:ac', '0002'
hbase(main):030:0> put 'jy', 'a002', 'am:am', '10000'
hbase(main):031:0> put 'jy', 'a003', 'ac:ac', '0001'
hbase(main):032:0> put 'jy', 'a003', 'am:am', '1000'
hbase(main):033:0> scan 'jy'
ROW                                 COLUMN+CELL
 a001                               column=ac:ac, timestamp=1423136264981, value=0001
 a001                               column=am:am, timestamp=1423136284893,
 value=-2000
 a002                               column=ac:ac, timestamp=1423136319545, value=0002
 a002                               column=am:am, timestamp=1423136334595,
 value=10000
 a003                               column=ac:ac, timestamp=1423136354122, value=0001
 a003                               column=am:am, timestamp=1423136363431, value=1000
```

然后在Inceptor Shell中建jy的映射表jiaoyi:

```
transwarp> CREATE EXTERNAL TABLE jiaoyi (jy_num STRING, acc_num STRING, amount
 DOUBLE) STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler' WITH
 SERDEPROPERTIES ("hbase.columns.mapping"=":key,ac:ac,am:am") TBLPROPERTIES
 ("hbase.table.name"="jy");
transwarp> SELECT * FROM jiaoyi;
a001    0001    -2000.0
a002    0002    10000.0
a003    0001    1000.0
```

现在我们通过对jiaoyi和bank_info的JOIN操作查看各个账户持有人执行过的交易：

```
transwarp> SELECT name, amount FROM bank_info JOIN jiaoyi ON acc_num = acc_num;
Zhang San       -2000.0
Zhang San       1000.0
Li Si           10000.0
```

7. Object Store使用方法

默认情况下，Hyperbase用相同的方法处理结构化和非结构化的数据。但是，由于图片等非结构化数据非常大，在向Hyperbase导入时会让Region大小增长迅速，频繁触发Region的Split和Compaction，在一定程度上卡住客户端的写入，影响Hyperbase的插入性能。因此，对非结构化的大对象采用不同方式处理，在插入时降低Region的Split和Compaction频率，提高插入性能，Object Store就是为了解决这个问题而设计的。我们提供了API以及JSON配置方法来使用Object Store。本章将展示Object Store API的示例代码。关于JSON配置操作请参考第 I 部分 “Hyperbase JSON配置手册”。

- 在使用Object Store建表时一定要预分Region，确保每个Region最多500G数据。
- TDH4.3之后的版本中，您需要在hbase-site.xml的配置项 `hbase.coprocessor.region.classes` 中额外添加一个类 `org.apache.hadoop.hyperbase.coprocessor.LobCoprocessor` 来保证object store的正常使用。

例 7.1. TDH4.2及之前版本的Object Store API

```
package io.transwarp;

import static org.junit.Assert.assertTrue;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.protobuf.generated.HyperbaseProtos;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hyperbase.client.HyperbaseAdmin;
import org.apache.hadoop.hyperbase.metadata.HyperbaseMetadata;
import org.apache.hadoop.hyperbase.secondaryindex.IndexedColumn;
import org.apache.hadoop.hyperbase.secondaryindex.LOBIndex;

public class TestLOB4_3 {

  protected static HyperbaseAdmin admin = null;
  protected static Configuration conf = null;
  static {
    conf = HBaseConfiguration.create();
    //改成对应的集群上面的配置!
```

```
    conf.set("hbase.zookeeper.quorum","transwarp-perf2,transwarp-
perf1,transwarp-perf3");
    conf.set("zookeeper.znode.parent", "/hyperbase1");
    conf.set("hbase.zookeeper.property.clientPort", "2181");
    try {
      admin = new HyperbaseAdmin(conf);
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

  public static void main(String[] args) throws Exception {
    testLOBGet();
  }

  public static void testLOBGet() throws Exception {
    byte[] row = Bytes.toBytes("rowkey01");
    byte[] tableName = Bytes
        .toBytes("SIMPLE_TEST_PUT_SCAN" + System.nanoTime());
    byte[] indexName = Bytes.toBytes("IDX");
    byte[] family1 = Bytes.toBytes("f1");
    byte[] family2 = Bytes.toBytes("f2");
    String path = "/tmp/file1";
    createTable(TableName.valueOf(tableName), family1, family2);
    addLOB(tableName, family1, indexName);
    byte[] value01 = getFileBytes(path);

    HTable htable = new HTable(conf, tableName);

    Put put = new Put(row);
    put.add(family1, Bytes.toBytes("q1"), value01);
    htable.put(put);
    htable.flushCommits();
    TimeUnit.SECONDS.sleep(1);

    Get get = new Get(row);
    Result rs = htable.get(get);
    CellScanner cs = rs.cellScanner();
    while(cs.advance()){
      assertTrue(Bytes.equals(CellUtil.cloneValue(cs.current()), value01));
      System.out.println(Bytes.toString(CellUtil.cloneValue(cs.current())));
    }
    htable.close();
    admin.deleteTable(TableName.valueOf(tableName));
  }

  public static byte[] getFileBytes(String path) throws IOException {
    FileInputStream fis = new FileInputStream(new File(path));// 新建一个
FileInputStream对象
    byte[] b = new byte[fis.available()];// 新建一个字节数组
    fis.read(b);// 将文件中的内容读取到字节数组中
    fis.close();
    return b;
  }

  public static void createTable(TableName tableName, byte[]...
 families) throws Exception {
```

```
    HTableDescriptor tableDescriptor = new HTableDescriptor(tableName);
    for (byte[] family : families) {
      tableDescriptor.addFamily(new HColumnDescriptor(family));
    }
    // create table succ
    admin.createTable(tableDescriptor, null);
    HyperbaseMetadata metadata = admin.getTableMetadata(tableName);
    assertTrue(metadata != null);
    // check metadata
    assertTrue(metadata.getFulltextMetadata() == null);
    assertTrue(metadata.getGlobalIndexes().isEmpty());
    assertTrue(metadata.getLocalIndexes().isEmpty());
    assertTrue(metadata.getLobs().isEmpty());
    assertTrue(metadata.isTransactionTable() == false);
  }

   public static void addLOB(byte[] tableName, byte[] family, byte[]
 LOBFamily) throws
      IOException {
    HyperbaseProtos.SecondaryIndex.Builder LOBBuilder =
 HyperbaseProtos.SecondaryIndex
        .newBuilder();
    LOBBuilder.setClassName(LOBIndex.class.getName());
    LOBBuilder.setUpdate(true);
    LOBBuilder.setDcop(true);
    IndexedColumn column = new IndexedColumn(family, Bytes.toBytes("q1"));
    LOBBuilder.addColumns(column.toPb());
    admin.addLob(TableName.valueOf(tableName), new
 LOBIndex(LOBBuilder.build()), LOBFamily, false, 1);
   }

}
```

例 7.2. TDH4.3及之后版本的Object Store API

```
package io.transwarp;

import static org.junit.Assert.assertTrue;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.concurrent.TimeUnit;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.protobuf.generated.HyperbaseProtos;
import org.apache.hadoop.hbase.util.Bytes;
import org.apache.hadoop.hyperbase.client.HyperbaseAdmin;
import org.apache.hadoop.hyperbase.metadata.HyperbaseMetadata;
import org.apache.hadoop.hyperbase.secondaryindex.IndexedColumn;
```

```
import org.apache.hadoop.hyperbase.secondaryindex.LOBIndex;

public class TestLOB4_3 {

  protected static HyperbaseAdmin admin = null;
  protected static Configuration conf = null;
  static {
    conf = HBaseConfiguration.create();
    //改成对应的集群上面的配置!
    conf.set("hbase.zookeeper.quorum","transwarp-perf2,transwarp-
perf1,transwarp-perf3");
    conf.set("zookeeper.znode.parent", "/hyperbase1");
    conf.set("hbase.zookeeper.property.clientPort", "2181");
    try {
      admin = new HyperbaseAdmin(conf);
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }

  public static void main(String[] args) throws Exception {
    testLOBGet();
  }

  public static void testLOBGet() throws Exception {
    byte[] row = Bytes.toBytes("rowkey01");
    byte[] tableName = Bytes
        .toBytes("SIMPLE_TEST_PUT_SCAN" + System.nanoTime());
    byte[] indexName = Bytes.toBytes("IDX");
    byte[] family1 = Bytes.toBytes("f1");
    byte[] family2 = Bytes.toBytes("f2");
    String path = "/tmp/file1";
    createTable(TableName.valueOf(tableName), family1, family2);
    addLOB(tableName, family1, indexName);
    byte[] value01 = getFileBytes(path);

    HTable htable = new HTable(conf, tableName);

    Put put = new Put(row);
    put.add(family1, Bytes.toBytes("q1"), value01);
    htable.put(put);
    htable.flushCommits();
    TimeUnit.SECONDS.sleep(1);

    Get get = new Get(row);
    Result rs = htable.get(get);
    CellScanner cs = rs.cellScanner();
    while(cs.advance()){
      assertTrue(Bytes.equals(CellUtil.cloneValue(cs.current()), value01));
      System.out.println(Bytes.toString(CellUtil.cloneValue(cs.current())));
    }
    htable.close();
    admin.deleteTable(TableName.valueOf(tableName));
  }

  public static byte[] getFileBytes(String path) throws IOException {
```

```
    FileInputStream fis = new FileInputStream(new File(path));// 新建一个
FileInputStream对象
    byte[] b = new byte[fis.available()];// 新建一个字节数组
    fis.read(b);// 将文件中的内容读取到字节数组中
    fis.close();
    return b;
  }

  public static void createTable(TableName tableName, byte[]...
 families) throws Exception {
    HTableDescriptor tableDescriptor = new HTableDescriptor(tableName);
    for (byte[] family : families) {
      tableDescriptor.addFamily(new HColumnDescriptor(family));
    }
    // 注意，object store一定要预分配region，每个region最好不要超过500G的数据。
    byte[][] splitKeys = new byte[10][];
    for (int i = 0; i < 100; i++) {
      splitKeys[i] =  Bytes.toBytes("rowkey" + i);
    }
    // create table succ
    admin.createTable(tableDescriptor, null, splitKeys);
    HyperbaseMetadata metadata = admin.getTableMetadata(tableName);
    assertTrue(metadata != null);
    // check metadata
    assertTrue(metadata.getFulltextMetadata() == null);
    assertTrue(metadata.getGlobalIndexes().isEmpty());
    assertTrue(metadata.getLocalIndexes().isEmpty());
    assertTrue(metadata.getLobs().isEmpty());
    assertTrue(metadata.isTransactionTable() == false);
  }

  public static void addLOB(byte[] tableName, byte[] family, byte[]
 LOBFamily) throws
      IOException {
    HyperbaseProtos.SecondaryIndex.Builder LOBBuilder =
 HyperbaseProtos.SecondaryIndex
        .newBuilder();
    LOBBuilder.setClassName(LOBIndex.class.getName());
    LOBBuilder.setUpdate(true);
    LOBBuilder.setDcop(true);
    IndexedColumn column = new IndexedColumn(family, Bytes.toBytes("q1"));
    LOBBuilder.addColumns(column.toPb());
    admin.addLob(TableName.valueOf(tableName), new
 LOBIndex(LOBBuilder.build()), LOBFamily, false, 1);
  }

}
```

8. Hyperbase API使用说明

本章介绍Hyperbase中的常用API。

8.1. HBaseConfiguration

作用

通过此接口配置Hyperbase。

常用方法签名及描述

添加Hyperbase配置资源

```
static org.apache.hadoop.conf.Configuration
 addHbaseResources(org.apache.hadoop.conf.Configuration conf)
```

利用classpath中的HbaseResource创建配置对象

```
static org.apache.hadoop.conf.Configuration create()
```

暂无描述

```
static org.apache.hadoop.conf.Configuration
create(org.apache.hadoop.conf.Configuration that)
```

获取属性值,并转换为Int作为返回值

```
static int getInt(org.apache.hadoop.conf.Configuration conf, String name,
 StringdeprecatedName, int defaultValue)
```

从配置实例中获取密码

```
static String getPassword(org.apache.hadoop.conf.Configuration conf, String alias,
 String defPass)
```

获取属性名对应的值

```
string get(String name)
```

获取boolean类型属性值,如果其属性值不是boolean类型的,则返回默认属性值

```
String getBoolean(String name, boolean defaultValue)
```

通过属性名来设置值

```
void set(String name, String value)
```

设置boolean类型的属性值

```
void setBoolean(String name, boolean value)
```

例 8.1. 用 HBaseConfiguration 设置ZooKeeper配置项

```
HBaseConfiguration hc = new HBaseConfiguration();
hc.set("hbase.zookeeper.property.clientPort","2181");
```

8.2. HBaseAdmin

作用

提供了一个接口来管理Hyperbase数据库的表信息。包括：创建表,删除表,列出表 项,使表有效或无效,以及添加或删除表列族成员等。

常用方法签名及描述

修改一列

```
void modifyColumn(byte[] tableName, final HColumnDescriptor descriptor)
```

删除一列

```
void deleteColumn(byte[] tableName, final String columnName)
```

向一个已经存在的表添加列

```
void addColumn(byte[] tableName, HColumnDescriptor column)
```

静态函数,查看Hyperbase是否处于运行状态

```
void checkHBaseAvailable(HBaseConfiguration conf)
```

创建一个表,同步操作

```
void createTable(HTableDescriptor desc)
```

删除一个已经存在的表

```
void deleteTable(byte[] tableName)
```

将表上线

```
void enableTable(byte[] tableName)
```

将表下线

```
void disableTable(byte[] tableName)
```

列出所有用户控件表项

```
HTableDescriptor[] listTables()
```

修改表的模式（异步的操作,可能需要花费一定的时间）

```
void modifyTable(byte[] tableName, HTableDescriptor htd)
```

检查表是否存在

```
boolean tableExists(String tableName)
```

关闭region

```
void closeRegion(final String regionname, final String serverName)
```

表或者region的压缩

```
void compact(final String tableNameOrRegionName)
```

表或者region的flush

```
void flush(final byte[] tableNameOrRegionName)
```

例 8.2. 用 HBaseAdmin 创建、disable和删除表

```
HBaseAdmin admin = HBaseAdmin(HBaseConfiguration.create());
HTableDescriptor desc = new
 HTableDescriptor(TableName.valueOf("default_table"));
HColumnDescriptor colDesc = new HColumnDescriptor("cf1");
desc.addFamily(colDesc);
// 创建表
admin.createTable(desc);
//disable 表
admin.disableTable(desc.getTableName());
// 删除表
admin.deleteTable(desc.getTableName());
```

8.3. HTableDescriptor

作用

描述一个Hyperbase表的所有细节,包括列族等信息。

常用方法签名及描述

获取所有列信息

```
Collection getFamilies()
```

添加一个列族

```
void addFamily(HColumnDescriptor)
```

移除一个列族

```
HColumnDescriptor removeFamily(byte[] column)
```

获取表的名字

```
byte[] getName()
```

获取属性的值

```
byte[] getValue(byte[] key)
```

设置属性的值

```
void setValue(String key, String value)
```

使用示例请参考例 8.2 “用 HBaseAdmin 创建、disable和删除表”。

8.4. HColumnDescriptor

作用

描述了一个Hyperbase表的列族信息,包括：版本, 压缩设置等。

常用方法签名及描述

获取最大版本

```
int getMaxVersions()
```

获取最小版本

```
int getMinVersions()
```

设置最大版本

```
HColumnDescriptor setMaxVersions(int maxVersions)
```

设置最小版本

```
HColumnDescriptor setMinVersions(int minVersions)
```

获取列族名称

```
byte[] getName()
```

获取对应属性的值

```
byte[] getValue(byte[] key)
```

设置对应属性的值

```
void setValue(String key, String value)
```

使用示例请参考例 8.2 “用 HBaseAdmin 创建、disable和删除表”。

8.5. HTable

作用

用于单个表的通信,使用这个类对表进行读写是非线程安全的。

常用方法签名及描述

释放所有的资源或挂起内部缓冲区中的更新

```
void close()
```

检查Get实例所指定的值是否存在于HTable的列中

```
Boolean exists(Get get)
```

获取指定行的某些单元格所对应的值

```
Result get(Get get)
```

获取当前给定列族的scanner实例

```
ResultScanner getScanner(byte[] family)
```

获取当前表的HTableDescriptor实例

```
HTableDescriptor getTableDescriptor()
```

获取表名

```
byte[] getTableName()
```

检查表是否在线

```
static boolean isTableEnabled(HBaseConfiguration conf, String tableName)
```

向表中添加值

```
void put(Put put)
```

获取当前表中所有region的start keys

```
byte [][] getStartKeys()
```

获取当前表中所有region的end keys

```
byte[][] getEndKeys()
```

例 8.3. 使用 HTable

```
byte [] tableName = Bytes.toBytes("tableName");
HTable table = new HTable(conf, Bytes.toBytes(tableName));
byte[] tableName = table.getTableName();
```

8.6. Put

作用

写数据到一行中。

常用方法签名及描述

将指定的列和对应的值添加到Put实例中

```
Put add(byte[] family, byte[] qualifier, byte[] value)
```

将指定的列和对应的值及时间戳添加到Put实例中

```
Put add(byte[] family, byte[] qualifier, long ts, byte[] value)
```

获取Put实例的行

```
byte[] getRow()
```

获取Put实例的时间戳

```
long getTimeStamp()
```

检查familyMap是否为空

```
boolean isEmpty()
```

设置Put实例的时间戳

```
Put setTimeStamp(long timeStamp)
```

例 8.4. 使用 Put

```
byte [] row = new byte [] {'r'};
byte [] tableName = Bytes.toBytes("tableName");
HTable table = new HTable(conf, tableName);
Put p = new Put(row);
```

```
p.add(family,qualifier,value);
table.put(p);
```

8.7. Get

作用

获取单行的相关信息。

常用方法签名及描述

获取指定列族和qualifier对应的列

```
Get addColumn(byte[] family, byte[] qualifier)
```

通过指定的列族获取其对应列的所有列

```
Get addFamily(byte[] family)
```

获取指定取件的列的版本号

```
Get setTimeRange(long minStamp,long maxStamp)
```

当执行Get操作时设置服务器端的过滤器

```
Get setFilter(Filter filter)
```

例 8.5. 使用 Put

```
byte [] row = new byte [] {'r'};
byte [] tableName = Bytes.toBytes("tableName");
HTable table = new HTable(conf, tableName);
Get g = new Get(row);
Result result = table.get(get);
```

8.8. Scan

作用

扫描指定row key范围内的行。

常用方法签名及描述

设置scan的开始rowkey

```
Scan setStartRow(byte[] startRow)
```

设置scan结束rowkey

```
Scan setStopRow(byte[] stopRow)
```

指定filter

```
Scan setFilter(Filter filter)
```

获取指定的列

```
Scan addColumn(byte[] family, byte[] qualifier)
```

获取所有列族下的列

```
Scan addFamily(byte[] family)
```

例 8.6. 使用 Scan

```
SingleColumnValueFilter filter = new SingleColumnValueFilter(
    cf,
    column,
    CompareOp.EQUAL,
    Bytes.toBytes("my value")
    );
scan.setFilter(filter);

byte [] tableName = Bytes.toBytes("tableName");
Configuration conf = HBaseConfiguration.create();
HTable table = new HTable(conf, tableName);

try {
    rs = table.getScanner(scan);
    for (Result r : rs) {
        for (KeyValue kv : r.list()) {
            System.out.println("row:" + Bytes.toString(kv.getRow()));
        }
    }
} finally {
    rs.close();
}
```

8.9. Result

作用

存储Get或者Scan操作后获取表的单行值。

常用方法签名及描述

判断结果集是否存在

```
Boolean getExists()
```

获得最新版本的列值

```
byte[] getValue(byte[] family, byte[] qualifier)
```

判断结果集合是否为空

```
boolean isEmpty()
```

转换为Cell[] 数组返回

```
Cell[] rawCells()
```

扫描一个cell

```
boolean advance()
```

使用示例请参考例 8.6 “使用 Scan”。

9. 程序中调用Hyperbase所需的jar

9.1. 建表所需的jar

- hbase-client-*-transwarp.jar
- hbase-bson-*-transwarp.jar
- hbase-common-*-transwarp.jar
- hbase-protocol-*-transwarp.jar
- hadoop-common-*-transwarp.jar
- hadoop-auth-*-transwarp.jar
- hadoop-annotations-*-transwarp.jar
- hadoop-mapreduce-client-core-*-transwarp.jar
- commons-codec-*.jar
- commons-io-*.jar
- commons-lang-*.jar
- commons-logging-*.jar
- commons-collections-*.jar
- guava-*.jar
- protobuf-java-*.jar
- netty-*.Final.jar
- htrace-core-*.jar
- elasticsearch-*-transwarp.jar
- jackson-mapper-asl-*.jar
- findbugs-annotations-*.jar
- zookeeper-*-transwarp.jar
- log4j-*.jar
- commons-configuration-*.jar
- slf4j-api-*.jar
- slf4j-log4j12-*.jar
- jsch-*.jar
- jzlib-*.jar
- hbase-hyperbase-*-transwarp.jar

jar名称中*位置是包的版本号，这些jar的版本号根据您使用的TDH版本的不同会有所不同，以您集群上正在使用的包为准。要获取这些包，您可以在您集群中的任意一台服务器上运行下面的脚本：

```
#!/bin/bash
rm -rf /tmp/hbase-jars
mkdir /tmp/hbase-jars
cd /tmp/hbase-jars
jardir=`pwd`
#echo $jarsource":/usr/lib/hbase/lib/hbase-client-0.98.6-transwarp.jar"

#jar in /usr/lib/hbase
cp /usr/lib/hbase/lib/hbase-client* ./
cp /usr/lib/hbase/lib/hbase-bson* ./
cp /usr/lib/hbase/lib/hbase-common* ./
cp /usr/lib/hbase/lib/hbase-protocol* ./
cp /usr/lib/hbase/lib/hbase-hyperbase* ./

cp /usr/lib/hbase/lib/commons-codec* ./
cp /usr/lib/hbase/lib/commons-io* ./
cp /usr/lib/hbase/lib/commons-lang* ./
cp /usr/lib/hbase/lib/commons-logging* ./
cp /usr/lib/hbase/lib/commons-collections* ./

cp /usr/lib/hbase/lib/guava*  ./
cp /usr/lib/hbase/lib/protobuf-java* ./
cp /usr/lib/hbase/lib/netty* ./
cp /usr/lib/hbase/lib/htrace-core* ./
cp /usr/lib/hbase/lib/elasticsearch* ./
cp /usr/lib/hbase/lib/jackson-mapper-asl* ./
cp /usr/lib/hbase/lib/findbugs-annotations* ./

cp /usr/lib/hbase/lib/log4j* ./
cp /usr/lib/hbase/lib/commons-configuration* ./
cp /usr/lib/hbase/lib/slf4j-api* ./
cp /usr/lib/hbase/lib/slf4j-log4j12* ./


#jar in /usr/lib/hadoop
cp /usr/lib/hadoop/hadoop-common* ./
cp /usr/lib/hadoop/hadoop-auth* ./
cp /usr/lib/hadoop/hadoop-annotations* ./
cp /usr/lib/hadoop-mapreduce/hadoop-mapreduce-client-core* ./

cp /usr/lib/hadoop/lib/jsch* ./
cp /usr/lib/hadoop/lib/jzlib* ./

#jar in /usr/lib/zookeeper
cp /usr/lib/zookeeper/zookeeper* ./

echo 'Your jars are ready in '$jardir
```

脚本会将上述所列的jar包拷贝到服务器上的/tmp/hbase-jars下。您要在本地进行开发还需要hbase-site.xml文件。这个文件在集群中任意一台服务器上的/etc/hbase/conf/目录下。

部分 I. Hyperbase JSON配置手册

10. JSON配置操作简介

10.1. 表数据 vs 表的扩展数据

索引是Hyperbase的核心功能之一，我们在使用Hyperbase时，常常会为表建各类索引，包括全局索引、局部索引、全文索引和LOB索引，利用索引中的数据提高查询效率。索引中的数据不属于表数据，但是从表数据而来，和表密不可分，所以我们将表数据和它所有索引中的数据合称为 **表的扩展数据**，也就是说，我们做如下定义：

表的扩展数据 = 表数据＋全局索引数据＋局部索引数据＋全文索引数据＋LOB索引数据

10.2. 表的元数据 vs 表的扩展元数据

Hyperbase表的元数据包括表名、列族名、`DATA_BLOCK_ENCODING`、`TTL`、`BLOCKSIZE` 等等。一张Hyperbase表的各个索引也有自己的元数据，和索引数据一样，索引的元数据和表的关系也十分紧密，所以我们将表的元数据和它所有索引的元数据合称为 **表的扩展元数据**：

表的扩展元数据 = 表的元数据＋全局索引元数据＋局部索引元数据＋全文索引元数据＋LOB索引元数据

我们有时也会将表的元数据称为 **基础元数据** 或者 **base元数据**。

10.3. JSON配置的命令行指令

为操作表的扩展数据和扩展元数据服务，Hyperbase提供了 **扩展的命令行指令**：`describeInJson`、`alterUseJson` 和 `truncate_all`。

语法：`describeInJson`

```
describeInJson '<table>', ['true|false'] ,['/<target_dir>/<target_file>'], ['true|false']
```

说明：打印 `<table>` 的扩展元数据（以JSON串形式输出）。第二个参数为可选参数，选择true会将表的扩展元数据以格式化的JSON串打印，默认值为false。第三个参数为可选参数，指定 `/<target_dir>/<target_file>` 可以将表的扩展元数据打印到本地 `<target_dir>` 目录下的 `<target_file>` 文件中，不指定则默认将结果打印到console。如果指定打印到文件，请注意操作的用户需要有本地 `<target_dir>` 的写权限，所以保险起见可以选择使用/tmp目录。第四个参数为可选参数，指定是否将Split Keys以十六进制打印。

例 10.1. `describeInJson`

我们在Hyperbase中的jsondoc namespace下有一张空表jtable，暂时没有任何索引。

将jsondoc:jtable的扩展元数据以未格式化的JSON串形式打印到console：

```
describeInJson 'jsondoc:jtable'
```

将json:jtable的扩展元数据以格式化的JSON串形式打印到console：

```
describeInJson 'jsondoc:jtable', 'true'
```

将json:jtable的扩展元数据以格式化的JSON串形式打印到本地/tmp/jtable.txt文件中：

```
describeInJson 'jsondoc:jtable', 'true', '/tmp/jtable.txt'
```

将json:jtable的扩展元数据以格式化的JSON串形式打印到本地/tmp/jtable.txt文件中，以十六进制打印Split Keys：

```
describeInJson 'jsondoc:jtable', 'true', '/tmp/jtable.txt', 'true'
```

语法：`alterUseJson`

```
alterUseJson '<table>','<json_string>|/<dir>/<json_file>', '[true|false]'
```

说明：如果 `<table>` 存在，根据提供的JSON串配置它的扩展元数据；如果 `<table>` 不存在，则先建表，并根据提供的JSON串配置它的扩展元数据。有两种提供JSON串的方式：直接提供一个JSON串 `<json_string>`，或者提供JSON串所在文件的路径 `/<dir>/<json_file>`。第三个参数为可选参数，指定是否以将JSON串中指定的Split Keys当做十六进制的字符读取。

例 10.2. `alterUseJson`

```
alterUseJson 'jsondoc:jtable', '/tmp/jtable2.txt'
```

将Split Keys当做十六进制字符打印/读取

`describeInJson` 和 `alterUseJson` 两个指令的最后一个参数指定是否将Split Keys以十六进制输出/输入。在Split Keys以十六进制字符表示的情况下，执行 `describeInJson` 和 `alterUseJson` 时必须总是将这个参数设为true，以保证在每次扩展元数据输出/输入时，Hyperbase可以正常地识别Split Keys。如果不进行设置，会发生Split Keys乱码，导致JSON串中的Split Key和表实际的Split Key不同。

语法：`truncate_all`

```
truncate_all '<table>'
```

说明：清空 `<table>` 以及所有 `<table>` 的各类索引中的数据，也就是清除表的扩展数据，保留表的扩展元数据。

- 表的扩展元数据全部都包含在传给Hyperbase的JSON串中，`alterUseJson` 会根据JSON描述来创建和修改表的扩展元数据，将表配置成输入的JSON串描述相同的结构。在第 11 章 JSON配置详解中，我们会详细介绍怎样写JSON串来配置扩展元数据。

我们可以和命令行中的基础命令 `describe`、`alter` 和 `truncate` 大致地做下面的类比：

普通指令	扩展指令
`describe`（打印元数据）	`describeInJson`（打印扩展元数据）
`truncate`（清空表数据，保留基础元数据）	`truncate_all`（清空扩展数据，保留扩展元数据）
`alter`（配置元数据）	`alterUseJson`（配置扩展元数据）

- Hyperbase JSON配置操作面向普通Hyperbase表（也就是非Hyperdrive表）。Hyperdrive表有一套更合适的工具——Hyperdrive SQL——用于创建和删除索引。我们建议不对Hyperdrive表通过本手册中即将介绍的方法配置扩展元数据，而是统一地通过Hyperdrive SQL来操作。
- `alter` 指令不能建表，只能修改表的元数据；而 `alterUseJson` 可以建表。
- 扩展指令可以对表的扩展元数据进行统一操作和管理，而普通指令只能修改表的基础元数据，容易导致表（元）数据和表的扩展（元）数据不统一的情况（譬如，某表被执行 `truncate`后，其索引数据并不会被清空）。所以我们建议尽量使用扩展指令。

11. JSON配置详解

11.1. 扩展元数据JSON串的基本格式

例 10.1 “describeInJson”中的jsondoc:jtable的扩展元数据如下：

```
{
  "tableName" : "jsondoc:jtable",                    ❶
  "base" : {                                         ❷
    "families" : [ {
      "FAMILY" : "f1",
      "DATA_BLOCK_ENCODING" : "NONE",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "NONE",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false
  },
  "fulltextindex" : {                                ❸
  },
  "globalindex" : {                                  ❹
    "indexs" : [ ]
  },
  "lob" : {                                          ❺
    "indexs" : [ ]
  },
  "localindex" : {                                   ❻
    "indexs" : [ ]
  }
}
```

❶ **表名**，可选项。Hyperbase会根据`alterUseJson` 指令后提供的表名来判断应该为哪张表配置扩展元数据。

❷ **基础（base）模块**，必填项，包含了表的基础元数据。

❸ **fulltextindex模块**，选填项，包含了表fulltext index的元数据。

❹ **globalindex模块**，选填项，包含了表global index的元数据。

❺ **lob模块**，选填项，包含了表lob的元数据。

❻ **localindex模块**，选填项，包含了local index的元数据。

这也是任何一张Hyperbase表的扩展元数据的JSON串都具有的基本格式，概括总结为以下几个模块：

```
{
    "tableName" : "<table_name>"
    "base" : {...}
    "fulltextindex" : {...}
    "globalindex" : {...}
    "lob" : {...}
    "localindex" : {...}
}
```

当执行 `alterUseJson` 来根据输入的JSON串来配置表时，Hyperbase会直接将表的扩展元数据修改为和输入的JSON串一致。也就是说：

- 如果输入的JSON串相对于表当前的扩展元数据减少了一部分信息，Hyperbase会删除这部分信息。**注意**，扩展元数据中的所有可选配置项都有默认值，所以“将某配置项删除”等同于“将某配置项的值恢复为默认值”。
- 如果输入的JSON串相对于表当前的扩展元数据增加了一部分信息，Hyperbase会添加这部分信息；
- 如果输入的JSON串相对于表当前的扩展元数据有一部分信息发生了变化，Hyperbase会更新这部分信息。

JSON串的编写规则

关于拼写：

您在编写JSON串时，需要保证配置项的名称和值的拼写和我们提供的 **完全一致**。其中尤其要注意的几点为：

- 除了fulltextindex模块和部分索引模块的配置项，大多数配置项都要放在双引号中，我们建议您使用我们提供的模板编写JSON串。同时您还可以尽量 **复用** JSON串——让 `describeInJson` 打印到文件，在文件基础上修改，减少不必要的错误。
- 除了fulltextindex模块以外，所有模块中的配置项名称都需要 **大写**，除非另外指出，配置项的值都需要放在双引号中。同样，我们建议您使用我们提供的模板，或者复用您自己的JSON串。
- 因为fulltextindex的信息需要和ElasticSearch共享，fulltextindex模块的拼写和其他模块不同，有专门的规定，请参考第 11.3 节 “fulltextindex模块”来了解细节。

关于JSON串中的顺序：

- JSON串中的模块顺序没有规定。
- JSON串中配置项的先后顺序没有规定。
- `"SPLIT_KEYS"` 组中元素出现顺序没有规定。
- 有的配置项组内元素的顺序有规定，包括：

```
# globalindex模块中的 "indexColumnInfos" 配置项
```

```
    # localindex模块中的 "indexColumnInfos" 配置项
```

下面我们分别介绍base、fulltextindex、globalindex和lob模块分别的配置细节。**我们将不介绍localindex模块的配置，因为普通Hyperbase表不支持使用局部索引，而支持局部索引的Hyperdrive表有更灵活的工具Hyperdrive SQL用于创建和管理索引。**

11.2. base模块

base模块是表的扩展元数据中必须的模块，任何一个 `alterUseJson` 传递给Hyperbase的JSON串都必须有该模块。一张普通Hyperbase表（非Hyperdrive表）的base模块具有如下格式：

普通Hyperbase表的base模块格式

```
"base" : {
    "SPLIT_KEYS": ["<split_key1>", "<split_key2>", ...], ❶
    "families" : [{<cf_meta>}, {<cf_meta>}, ...}], ❷
    "THEMIS_ENABLE" : false
}
```

❶ `"SPLIT_KEYS"` 配置项指定该表的Split Key，该配置项为 **选填项**。Split Key只能在建表时指定，建表后无法修改，所以建表后传给Hyperbase的JSON串中这一配置项都无效。

❷ `"families"` 配置项配置表中列族的元数据，该配置项为 **必填项**。它由一组列族元数据组成：每个元素 `{<cf_meta>}` 各包含一个列族的元数据，组中至少要有一个元素。列族的元数据配置 `{<cf_meta>}` 有固定的格式，在`{<cf_meta>}` 的配置中会详细介绍。

`{<cf_meta>}` 的配置

```
{
    "FAMILY":"<column_family>",  // 必选项，指定列族名。
    "DATA_BLOCK_ENCODING":"<encoding_scheme>", // 可选项，默认为NONE，无编码方式
    "BLOOMFILTER":"<bloomfilter>", //可选项，默认为NONE
    "REPLICATION_SCOPE":"<int>", //可选项，默认为0
    "VERSIONS":"<num_versions>", // 可选项，默认为1
    "COMPRESSION":"<compression_scheme>", // 可选项，默认为NONE
    "MIN_VERSIONS":"<num_minversions>", // 可选项，默认为0
    "TTL":"<ttl>", //可选项，默认为Integer.Max
    "KEEP_DELETED_CELLS":"<boolean>", //  可选项，请指定为FALSE
    "BLOCKSIZE":"<blocksize>", // 可选项，默认65535
    "IN_MEMORY":"<boolean>", //可选项，默认为false
    "BLOCKCACHE":"<boolean>"  //可选项，默认为true
}
```

表 11.1. `<cf_meta>` 中的可选配置项信息

配置项	选项	默认值	推荐值
DATA_BLOCK_ENCODING	NONE, PREFIX, DIFF, FAST_DIFF,	NONE	PREFIX 或 PREFIX_TREE

配置项	选项	默认值	推荐值
	PREFIX_TREE, INDEXED_PREFIX		
BLOOMFILTER	NONE, ROW, ROWCOL	NONE	ROW
REPLICATION_SCOPE	0（表示local scope），1（表示global scope）	0	0
VERSIONS	1或更多	1	1
COMPRESSION	LZO, GZ, NONE, SNAPPY, LZ4	NONE	SNAPPY
MIN_VERSIONS	0或更多	0	0
TTL	正整数（表示秒数）	INT最大值2147483647，表示永久	按业务确定表的 TTL，如对于LOB这样的特大列族，设置合理的 TTL 是一个删除过期旧数据的好方法。
KEEP_DELETED_CELLS	true或false	false	false
BLOCKSIZE	正整数（表示一个Hfile中的byte数量）	65536	65536
IN_MEMORY	true或false	false	false
BLOCKCACHE	true或false	false	false

例 11.1. 有两个列族的Hyperbase表的base模块

下面是一张有两个列族（f1, f2）的Hyperbase表的base模块：

```
"base" : {
    "families" : [ {
      "FAMILY" : "f1",  // 列族f1
      "DATA_BLOCK_ENCODING" : "NONE",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "NONE",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    }, {
      "FAMILY" : "f2",  // 列族f2
      "DATA_BLOCK_ENCODING" : "NONE",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "NONE",
```

```
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false
  }
```

Hyperdrive表的扩展元信息

TDH4.5及以后的版本中新增了Hyperdrive表，如果对一张Hyperdrive表执行 `describeInJson`，我们能看到Hyperdrive表的base模块中会有它的 `"schema"` 信息，其中包含了Hyperdrive表Row Key的数据类型，以及表中其他列的column family, column qualifier和数据类型：

Hyperdrive表的base模块

```
"base" : {
    "SPLIT_KEYS": ["<split_key1>", "<split_key2>", ...],
    "families" : [{<cf1_info>}, {<cf2_info>}, ...}],
    "THEMIS_ENABLE" : false,
    "schema" : {...}
}
```

Hyperdrive表的base模块中的schema信息

```
"schema" : {
    "rowkey" : {
      "primary" : {
        "name" : "STRING"
      }
    },
    "columns" : [ {
      "family" : "f",
      "qualifier" : "q1",
      "type" : {
        "primary" : {
          "name" : "STRING"
        }
      }
    }, {
      "family" : "f",
      "qualifier" : "q2",
      "type" : {
        "struct" : [ {
          "primary" : {
            "name" : "STRING"
          },
          "cutLength" : 20
        }, {
          "primary" : {
```

```
            "name" : "INTEGER"
          }
        }, {
          "primary" : {
            "name" : "DOUBLE"
          }
        } ]
      }
    }, {
      "family" : "f",
      "qualifier" : "q3",
      "type" : {
        "primary" : {
          "name" : "DOUBLE"
        }
      }
    } ]
}
```

但是如前文中提到，Hyperdrive表更适合使用Hyperdrive SQL操作，schema通过Hyperdrive SQL设置简单明了，我们不建议通过手工编写JSON串来对Hyperdrive表使用JSON配置，所以本文档不会对Hyperdrive表的JSON配置多加讨论。您可以参考《Hyperdrive SQL手册》来查看Hyperdrive表的操作。但是当您需要直接复用一张Hyperdrive表的扩展元信息时（比如将表从一个集群迁移到另一个集群），Hyperdrive表的JSON串将会非常实用——您可以先使用 `describeInJson` 将Hyperdrive表的扩展元数据输出到一个文件，将文件拷贝到新集群上，通过 `alterUseJson` 用该文件建表并在新集群的Inceptor中建映射表。这样的操作可以帮助您迅速地建好一张拥有完全相同的扩展元信息的表。在例 14.2 “Hyperdrive表的迁移”有详细的操作演示。

11.3. fulltextindex模块

fulltextindex模块是表的扩展元数据中的可选模块，您只需在为表建全文索引时在JSON串中填写该模块。

fulltextindex模块

```
"fulltextindex" : {
    "tableName" : "<affiliated_table>", ❶
    "allowUpdate" : <boolean>, //可选项，是否允许对建索引列进行update操作，默认为true
    "ttl" : <int>,  //可选项，单位为毫秒，默认为0
    "source" : <boolean>,  //可选项，是否在ES中存储_source信息，默认为true
    "all" : <boolean>, //可选项，是否在ES中存储_all信息，默认为false
    "fields" : [{<field_meta>}, {<field_meta>}, ... ] ❷
}
```

❶ **必填项**，指定全文索引所属表的名称。**注意和base模块的同名配置项的意义区分。**

❷ `"fields"` 配置项用于配置建全文索引的各个字段，为必选项。它由一组字段配置信息组成：每个元素 `{<field_meta>}` 各包含一个字段的配置，组中至少要有一个元素。字段的配置 `<field_meta>` 有固定的格式，在`{<field_meta>}` 的配置中会详细介绍。

表 11.2. fulltextindex模块中的配置项

配置项	选项	默认值	推荐值
allowUpdate	true或false	true	true
ttl	正整数（表示毫秒数）	0	如果没有超期限制，使用默认值。
source	true或false	true	true
all	true或false	false	false

和其他模块不同，fulltextindex模块中的配置项都 **必须小写**，并且，配置项的值的拼写，包括大小写和是否放在双引号之间，必须和上面提供的完全一致。为了减少配置出错的情况，我们建议：

- 对Hyperdrive表使用Hyperdrive SQL建索引。
- 对普通Hyperbase表使用我们提供的创建/修改fulltext索引和创建/修改fulltext分索引这两个模板。

{<field_meta>} 的配置

```
{
    "family" : "<column_family_name>", // 必填项，字段所在列族的名称
    "qualifier" : "<column_qualifier>", // 必填项，字段的column qualifier
    "encode_as_string" : <boolean>, ❶
    "attributes" : {
        "index" : "<index_option>", // 可选项，索引方式。
        "store" : "<boolean>", // 可选项，指定是否存储。
        "doc_values" : "<boolean>", //可选项，指定是否使用doc_vales优化。
        "type" : "<data_type>"❷
    }
}
```

❶ 可选项，这个配置项 **非常重要**，它指定了是否将该字段作为STRING类型转换为byte[]——如果它为true，则将该字段作为STRING类型转换；如果它为false，则根据该字段的 **实际类型** 转换。该配置项的默认值为 **false**，即按字段的实际类型转换。但是如果您从Inceptor映射表向Hyperbase导入数据，Hyperbase会默认将数据当做STRING转换（除非使用 #b 关键字显示指定）。所以，您需要格外注意该字段的数据是如何转换的，如果使用默认方式从Inceptor中导入数据（不根据数据实际类型转换），那么您需要将 encode_as_string 值设为true。

❷ 必填项，指定该字段的数据类型。目前支持的数据类型有：string, float, double, byte, short, integer, long, date, boolean和binary。

表 11.3. <field_meta> 中的可选配置项信息

配置项	选项	默认值	推荐值
store	true或false	true	true
index	analyzed, not_analyzed, no	not_analyzed	not_analyzed

配置项	选项	默认值	推荐值
doc_values	true或false	true	true

例 11.2. 用两个字段创建全文索引的表的fulltextindex模块

下面是一张名为 hyper:test 的表的fulltextindex模块。该全文索引使用了两个字段创建。

```
"fulltextindex" : {
  "tableName" : "hyper:test",
  "allowUpdate" : true,
  "ttl" : 0,
  "source" : true,
  "all" : false,
  "storeAsSource" : false,  ❶
  "storeFamily" : "",❷
  "writeConsistencyLevel" : "default", ❸
  "fields" : [ {            // 字段1的field_meta
    "family" : "f",
    "qualifier" : "q1",
    "encode_as_string" : true,
    "attributes" : {
      "index" : "not_analyzed",
      "store" : "true",
      "doc_values" : "true",
      "type" : "string"
    }
  }, {                      // 字段2的field_meta
    "family" : "f",
    "qualifier" : "q3",
    "encode_as_string" : true,
    "attributes" : {
      "index" : "not_analyzed",
      "store" : "true",
      "doc_values" : "true",
      "type" : "double"
    }
  } ]
}
```

❶❷❸ Hyperbase会自动生成这三项，您应当永远使用系统默认值，无需您自己配置。

11.4. globalindex模块

globalindex模块是表的扩展元数据中的可选模块，您只需在为表建全局索引时在JSON串中填写该模块，它的格式如下：

globalindex模块格式

```
"globalindex": {
    "indexs": [{<global_index_meta>}, {<global_index_meta>}, ...]
```

```
}
```

globalindex模块下只有一个配置项 "indexs"，为必填项。它是由表的所有全局索引元数据构成的组，组中的每个元素 {<global_index_meta>} 对应表的一个全局索引的元数据，组中至少要有一个元素。全局索引的元数据 <global_index_meta> 有固定的格式，在{<global_index_meta>} 的配置中会详细介绍。

{<global_index_meta>} 的配置

```
{
    "INDEX_NAME":"<global_index_name>", //必填项，指定索引名称
    "SPLIT_KEYS": [ "<split_key1>", "<split_key2>" ], //可选项，指定Split Key。默认无
Split key，即只有一个region。只能在建索引时指定，建好后不能修改。
    "UPDATE":"<boolean>", //可选项，默认为true
    "DCOP":"<boolean>", //可选项，默认为true
    "INDEX_CLASS":"COMBINE_INDEX", //必填项，值填为COMBINE_INDEX
    "indexColumnInfos": [{<index_column_info>}, {<index_column_info>}, ...,
 {<rowkey_info>}] ❶
}
```

❶ 必填项。它是一个组，组中的元素 {<index_column_info>} 是各个构成该全局索引的列的信息，包括该列的列族，qualifier和在索引词条中所占的长度。组中的最后一个元素 {<rowkey_info>} 是Row Key信息。组中至少要有两个元素：一个属于构成索引的列，另一个属于Row Key。列信息 {<index_column_info>} 如{<index_column_info>} 的配置中所示，Row Key信息 {<rowkey_info>} 如{<rowkey_info>} 的配置中所示。**注意，列信息的先后顺序决定了各列在索引词条中出现的顺序，所以这里组中元素的先后顺序是有意义的，需要您根据自己对索引的设计排列。**

表 11.4. {<global_index_meta>} 中的可选配置项

配置项	选项	默认值	推荐值
UPDATE	true或false （指定是否更新）	true	true
DCOP	true或false	true	true

{<index_column_info>} 的配置

```
{
    "FAMILY" : "<column_family>",  // 必填项，该列所属的列族名
    "QUALIFY" : "<column_qualifier>", //必填项，该列的column qualifier
    "SEGMENT_LENGTH" : <int> // 必填项，该列在全局索引词条中所占长度
}
```

{<rowkey_info>} 的配置

```
{
    "FAMILY" : "rowKey",  // 必填项，rowKey为固定用法
    "QUALIFY" : "rowKey", //必填项，rowKey为固定用法
    "SEGMENT_LENGTH" : <int> //必填项，该列在全局索引词条中所占长度
}
```

注意，SEGMENT_LENGTH 的值不放在引号中。

SEGMENT_LENGTH 的选择

总得来说，列的 SEGMENT_LENGTH 和列值的平均长度相近即可。但是列值的长度如何计算呢？普通Hyperbase表（非Hyperdrive表）中的数据以byte[]形式存储，所有类型的数据在存入Hyperbase时都会被转换成byte[]，列值的长度和 **转换成byte[]** 的方式直接相关。将数据转换成byte[]有两种方式：

方法一将数据的值当做STRING类型转换成byte[]

方法二根据数据的实际类型转换成对应byte[]

如果用方法一转换，那么 SEGMENT_LENGTH 应当设置成列值作为STRING类型的平均长度；如果用方法二转换，那么根据数据类型的不同， SEGMENT_LENGTH 也不同，具体如下表：

表 11.5. 数据类型和对应的byte[]长度

数据类型	byte[]长度
BOOLEAN	1
TINYINT	1
SMALLINT	2
INTEGER	4
BIGINT	8
FLOAT	4
DOUBLE	8
STRING	字符串长度
VARCHAR	字符串长度

所以，**您不仅需要对建索引列的值有大致的了解，还需要知道建索引列的数据存入Hyperbase时是按照哪种方法转换的。**

如果您选择通过Inceptor的映射表向Hyperbase插入数据，可以在建表时指定数据是否按类型转换：

指定数据是否按类型转换

```
CREATE TABLE ... STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping"=":key,f:q1,f:q2#b");
```

在 "hbase.columns.mapping"=":key,f:q1,f:q2#b" 中，列名后添加 #b 为指定按照数据实际类型转换为byte[]（方法二），如果没有添加 #b 则默认将数据当做STRING转换为byte[]（方法一）。

例 11.3. 一张有两个全局索引的表的globalindex模块

下面的globalindex模块属于一张有两个全局索引的表。两个全局索引分别名为 `name_balance_global_index` 和 `name_global_index`。其中，`name_balance_global_index` 由两列建成，`name_global_inex` 由一列建成。

```
"globalindex" : {
    "indexs" : [ { //第一个全局索引
      "INDEX_NAME" : "name_balance_global_index",
      "UPDATE" : "true",
      "DCOP" : "true",
      "INDEX_CLASS" : "COMBINE_INDEX",
      "indexColumnInfos" : [ {  //建name_balance_global_index的列之一
        "FAMILY" : "f",
        "QUALIFY" : "q1",
        "SEGMENT_LENGTH" : 8
      }, {   // 建name_balance_global_index的列之二
        "FAMILY" : "f",
        "QUALIFY" : "q3",
        "SEGMENT_LENGTH" : 9
      }, {
        "FAMILY" : "rowKey",
        "QUALIFY" : "rowKey",
        "SEGMENT_LENGTH" : 16
      } ],
    }, {  //第二个全局索引
      "INDEX_NAME" : "name_global_index",
      "UPDATE" : "true",
      "DCOP" : "true",
      "INDEX_CLASS" : "COMBINE_INDEX",
      "indexColumnInfos" : [ {  //建name_global_index的列
        "FAMILY" : "f",
        "QUALIFY" : "q1",
        "SEGMENT_LENGTH" : 8
      }, {
        "FAMILY" : "rowKey",
        "QUALIFY" : "rowKey",
        "SEGMENT_LENGTH" : 16
      } ],
    } ]
  }
```

11.5. localindex模块

目前，只有Hyperdrive表支持局部索引，我们推荐使用Hyperdrive SQL来创建和管理Hyperdrive表的索引，所以我们不对localindex模块多做讨论。

11.6. lob模块

lob模块是表中LOB列族的lob索引元数据。当表中有LOB列族时，表的base模块中LOB列族的元数据需要多一个配置项，同时表的扩展元数据中 **必须要包含lob模块**。

LOB列族的列族元数据（{<cf_meta>}） 配置

```
{
    "FAMILY":"<column_family>",  // 必选项，指定列族名。
    "DATA_BLOCK_ENCODING":"<encoding_scheme>", // 可选项，默认为NONE，无编码方式
    "BLOOMFILTER":"<bloomfilter>", //可选项，默认为NONE
    "REPLICATION_SCOPE":"<int>", //可选项，默认为0
    "VERSIONS":"<num_versions>", // 可选项，默认为1
    "COMPRESSION":"<compression_scheme>", // 可选项，默认为NONE
    "MIN_VERSIONS":"<num_minversions>", // 可选项，默认为0
    "TTL":"<ttl>", //可选项，默认为Integer.Max
    "KEEP_DELETED_CELLS":"<boolean>", //  必选项，请指定为FALSE
    "BLOCKSIZE":"<blocksize>", // 可选项，默认65535
    "IN_MEMORY":"<boolean>", //可选项，默认为false
    "BLOCKCACHE":"<boolean>"  //可选项，默认为true
}
```

注，LOB列族的 {<cf_meta>} 中的可选配置项和普通列族的 {<cf_meta>} 相同，请参考表 11.1 “<cf_meta> 中的可选配置项信息”。

lob模块的格式

```
"lob" : {
    "indexs" : [{<lob_index_meta>}, {<lob_index_meta>}, ...]
}
```

lob模块下只有一个配置项 "indexs"，为必填项。它是由表的所有LOB索引元数据构成的组，组中的每个元素 {<lob_index_meta>} 对应表的一个LOB索引的元数据，组中至少要有一个元素。：LOB索引的元数据 {<lob_index_meta>} 有固定的格式，在{<lob_index_meta>} 格式中会详细介绍。

{<lob_index_meta>} 格式

```
{
    "INDEX_NAME" : "<lob_index_name>", // 必填项，LOB索引名
    "DCOP" : "<boolean>",
    "INDEX_CLASS" : "LOB_INDEX", //必填项，必须填LOB_INDEX
    "DATA_BLOCK_ENCODING" : "<encoding_scheme>",
    "BLOOMFILTER" : "<bloomfilter>",
    "REPLICATION_SCOPE" : "<int>",
    "COMPRESSION" : "<compression_scheme>",
    "VERSIONS" : "<int>",
    "TTL" : "<int>",
    "MIN_VERSIONS" : "<int>",
    "KEEP_DELETED_CELLS" : "<boolean>",
    "BLOCKSIZE" : "<int>",
    "SPLIT_FAMILY" : "<boolean>",
    "IN_MEMORY" : "<boolean>",
    "BLOCKCACHE" : "<boolean>",
    "indexColumnInfos" : [ {
        "FAMILY" : "<lob_column_family>", // 必填项，LOB列的列族名
        "QUALIFY" : ""  //留为空
    } ]
}
```

注：{<lob_index_meta>} 中的可选配置项和 {<cf_meta>} 以及 {<global_index_meta>} 中的可选配置项重叠，请参考表 11.1 “<cf_meta> 中的可选配置项信息”以及表 11.4 “{<global_index_meta>} 中的可选配置项”。

例 11.4. 一张有LOB列族的表的扩展元数据

```
{
    "tableName" : "test_json",
    "base" : {
    "families" : [ {
        "FAMILY" : "f",
        "DATA_BLOCK_ENCODING" : "PREFIX",
        "BLOOMFILTER" : "ROW",
        "REPLICATION_SCOPE" : "0",
        "COMPRESSION" : "SNAPPY",
        "VERSIONS" : "1",
        "MIN_VERSIONS" : "0",
        "TTL" : "2147483647",
        "hbase.hstore.defaultengine.compactionpolicy.class" :
 "org.apache.hadoop.hbase.regionserver.compactions.ExploringWithLargeHFileCompactionPolicy", (1)
        "KEEP_DELETED_CELLS" : "false",
        "BLOCKSIZE" : "65536",
        "IN_MEMORY" : "false",
        "SPLIT_FAMILY" : "false",
        "BLOCKCACHE" : "true"
        } ]
    },

    "lob" : {
        "indexs" : [ {
            "INDEX_NAME" : "LOBP",
            "DCOP" : "true",
            "INDEX_CLASS" : "LOB_INDEX",
            "DATA_BLOCK_ENCODING" : "PREFIX",
            "BLOOMFILTER" : "ROW",
            "REPLICATION_SCOPE" : "0",
            "COMPRESSION" : "NONE",
            "VERSIONS" : "1",
            "TTL" : "2147483647",
            "MIN_VERSIONS" : "0",
            "KEEP_DELETED_CELLS" : "false",
            "BLOCKSIZE" : "65536",
            "SPLIT_FAMILY" : "true",
            "IN_MEMORY" : "false",
            "BLOCKCACHE" : "true",
            "indexColumnInfos" : [ {
                "FAMILY" : "f",
                "QUALIFY" : ""
            } ]
        } ]
    }

}
```

❶ Hyperbase会根据lob模块下的 `indexColumnInfos` 下的 `FAMILY` 值判断哪一个列族为LOB列族，并自动为该列族加上这个配置项和它的值。您无需在JSON串中设置。

12. JSON配置操作模板

下面提供的模板在TDH4.2及以后的版本中都适用。

创建/修改一张最基础的表

```
{
    "tableName" : "test_json",
    "base" : {
        "families" : [ {
            "FAMILY" : "f",
            "DATA_BLOCK_ENCODING" : "PREFIX",
            "BLOOMFILTER" : "ROW",
            "REPLICATION_SCOPE" : "0",
            "VERSIONS" : "1",
            "COMPRESSION" : "SNAPPY",
            "MIN_VERSIONS" : "0",
            "TTL" : "2147483647",
            "KEEP_DELETED_CELLS" : "false",
            "BLOCKSIZE" : "65536",
            "IN_MEMORY" : "false",
            "BLOCKCACHE" : "true"
        } ]
    }
}
```

创建/修改global索引

```
{
    "tableName" : "test_json",
    "base" : {
        "families" : [ {
            "FAMILY" : "f",
            "DATA_BLOCK_ENCODING" : "PREFIX",
            "BLOOMFILTER" : "ROW",
            "REPLICATION_SCOPE" : "0",
            "VERSIONS" : "1",
            "COMPRESSION" : "SNAPPY",
            "MIN_VERSIONS" : "0",
            "TTL" : "2147483647",
            "KEEP_DELETED_CELLS" : "false",
            "BLOCKSIZE" : "65536",
            "IN_MEMORY" : "false",
            "BLOCKCACHE" : "true"
        } ]
    },
    "globalindex" : {
        "indexs" : [ {
            "INDEX_NAME" : "index_q",
            "UPDATE" : "true",
            "DCOP" : "true",
            "INDEX_CLASS" : "COMBINE_INDEX",
            "indexColumnInfos" : [ {
                "FAMILY" : "f",
```

```
                "QUALIFY" : "q",
                "SEGMENT_LENGTH" : 16
            }, {
                "FAMILY" : "rowKey",
                "QUALIFY" : "rowKey",
                "SEGMENT_LENGTH" : 16
            } ]
        } ]
    }
}
```

注意，SEGMENT_LENGTH 的值不放在双引号中。

创建/修改LOB索引

```
{
    "tableName" : "test_json",
    "base" : {
        "families" : [ {
            "FAMILY" : "f",
            "DATA_BLOCK_ENCODING" : "PREFIX",
            "BLOOMFILTER" : "ROW",
            "REPLICATION_SCOPE" : "0",
            "COMPRESSION" : "SNAPPY",
            "VERSIONS" : "1",
            "MIN_VERSIONS" : "0",
            "TTL" : "2147483647",
            "hbase.hstore.defaultengine.compactionpolicy.class" :
 "org.apache.hadoop.hbase.regionserver.compactions.ExploringWithLargeHFileCompactionPolicy",
            "KEEP_DELETED_CELLS" : "false",
            "BLOCKSIZE" : "65536",
            "IN_MEMORY" : "false",
            "SPLIT_FAMILY" : "false",
            "BLOCKCACHE" : "true"
        } ]
    },
    "lob" : {
        "indexs" : [ {
            "INDEX_NAME" : "LOBP",
            "DCOP" : "true",
            "INDEX_CLASS" : "LOB_INDEX",
            "DATA_BLOCK_ENCODING" : "PREFIX",
            "BLOOMFILTER" : "ROW",
            "REPLICATION_SCOPE" : "0",
            "COMPRESSION" : "NONE",
            "VERSIONS" : "1",
            "TTL" : "2147483647",
            "MIN_VERSIONS" : "0",
            "KEEP_DELETED_CELLS" : "false",
            "BLOCKSIZE" : "65536",
            "SPLIT_FAMILY" : "true",
            "IN_MEMORY" : "false",
            "BLOCKCACHE" : "true",
            "indexColumnInfos" : [ {
                "FAMILY" : "f",
                "QUALIFY" : ""
            } ]
        } ]
```

```
    }
}
```

创建/修改fulltext索引

```
{
    "tableName" : "test_json",
    "base" : {
        "families" : [ {
            "FAMILY" : "f",
            "DATA_BLOCK_ENCODING" : "PREFIX",
            "BLOOMFILTER" : "ROW",
            "REPLICATION_SCOPE" : "0",
            "VERSIONS" : "1",
            "COMPRESSION" : "SNAPPY",
            "MIN_VERSIONS" : "0",
            "TTL" : "2147483647",
            "KEEP_DELETED_CELLS" : "false",
            "BLOCKSIZE" : "65536",
            "IN_MEMORY" : "false",
            "BLOCKCACHE" : "true"
        } ]
    },
    "fulltextindex" : {
        "tableName" : "test_json",
        "allowUpdate" : "true",
        "ttl" : 0,
        "source" : true,
        "fields" : [ {
            "family" : "f",
            "qualifier" : "a",
            "encode_as_string" : true,
            "attributes" : {
                "index" : "not_analyzed",
                "store" : "true",
                "doc_values" : "true",
                "type" : "string"
            }
        }, {
            "family" : "f",
            "qualifier" : "b",
            "encode_as_string" : true,
            "attributes" : {
                "index" : "not_analyzed",
                "store" : "true",
                "doc_values" : "true",
                "type" : "string"
            }
        }]
    }
}
```

创建/修改fulltext分索引

```
{
    "tableName" : "test_json",
    "base" : {
```

```
        "families" : [ {
            "FAMILY" : "f",
            "DATA_BLOCK_ENCODING" : "PREFIX",
            "BLOOMFILTER" : "ROW",
            "REPLICATION_SCOPE" : "0",
            "VERSIONS" : "1",
            "COMPRESSION" : "SNAPPY",
            "MIN_VERSIONS" : "0",
            "TTL" : "2147483647",
            "KEEP_DELETED_CELLS" : "false",
            "BLOCKSIZE" : "65536",
            "IN_MEMORY" : "false",
            "BLOCKCACHE" : "true"
        } ]
    },
    "fulltextindex" : {
        "tableName" : "test_json",
        "allowUpdate" : "true",
        "ttl" : 0,
        "source" : true,
        "fields" : [ {
            "family" : "f",
            "qualifier" : "a",
            "encode_as_string" : true,
            "attributes" : {
                "index" : "not_analyzed",
                "store" : "true",
                "doc_values" : "true",
                "type" : "string"
            }
        }, {
            "family" : "f",
            "qualifier" : "b",
            "encode_as_string" : true,
            "attributes" : {
                "index" : "not_analyzed",
                "store" : "true",
                "doc_values" : "true",
                "type" : "string"
            }
        } ],
        "splits" : {
            "family" : "f",
            "qualifier" : "b",
            "splitKeys" : ["9223370633623114807",
 "9223370647781835807" ,"9223370647781839807" ]
        }
    }
}
```

13. JSON配置简单使用实例

本章我们将用一些实际操作来演示怎样使用Hyperbase JSON配置操作来完成一些常见的简单任务。操作前，我们指出下面几个要点：

- 传给Hyperbase的JSON串必须有base模块。如果base模块中有LOB列族，那么JSON串中还必须有lob模块。
- 如果您需要使用可选配置项的默认值，请直接将该配置项在JSON串中省略。
- 我们建议使用JSON操作修改扩展元数据时，总是先将当前的扩展元数据输出到文件，在该文件的基础上修改，再将修改好的文件重新传给Hyperbase，以减少不必要的错误。**在实际生产中，我们推荐您根据您的应用场景在建表时一次性建好表所需的所有索引，尽量避免建表后，尤其是表中有数据后，对元数据的修改。**

为了表述的简洁，我们做出下面规定：

- 当我们说“对表jsondoc:jtable使用jtable.txt文件”，我们指的是执行：

```
alterUseJson 'jsondoc:jtable', '/tmp/jtable.txt'
```

  我们将所有的JSON串文件存在本地的/tmp目录下。
- 当我们说“查看表test_table的扩展元数据”，我们指的是执行：

```
describeInJson 'jsondoc:jtable', 'true', '/tmp/file.txt'
```

  然后查看文件file.txt中的内容。

例 13.1. 建表

假设Hyperbase中没有名为jsondoc:jtable的表。将下面的JSON串存在本地的/tmp/jtable.txt文件中（这个JSON串会建一张有一个列族f1的表，表的Split Keys为1, 2, 3。）：

```
{
    "tableName": "jsondoc:jtable",
    "base": {
        "SPLIT_KEYS": ["1","2","3"],
        "families": [
            {
                "FAMILY": "f1"
            }
        ]
    }
}
```

然后执行：

```
alterUseJson 'jsondoc:jtable', '/tmp/jtable.txt'
```

现在查看jsondoc:jtable的扩展元数据：

```
{
  "tableName" : "jsondoc:jtable",
  "base" : {
    "SPLIT_KEYS" : [ "1", "2", "3" ],
    "families" : [ {
      "FAMILY" : "f1",
      "DATA_BLOCK_ENCODING" : "NONE",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "NONE",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false
  },
  "fulltextindex" : {
  },
  "globalindex" : {
    "indexs" : [ ]
  },
  "lob" : {
    "indexs" : [ ]
  },
  "localindex" : {
    "indexs" : [ ]
  }
}
```

例 13.2. 添加和删除列族

对例 13.1 “建表”中建的表jsondoc:jtable使用下面的JSON串，将表中的f1列族删除，增加f2列族：

```
{
  "tableName" : "jsondoc:jtable",
  "base" : {
    "families" : [ {
      "FAMILY" : "f2",
      "DATA_BLOCK_ENCODING" : "NONE",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "NONE",
      "MIN_VERSIONS" : "0",
```

```
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false
  },
  "fulltextindex" : {
  },
  "globalindex" : {
    "indexs" : [ ]
  },
  "lob" : {
    "indexs" : [ ]
  },
  "localindex" : {
    "indexs" : [ ]
  }
}
```

例 13.3. 修改列族属性

对例 13.2 “添加和删除列族”中的表jsondoc:jtable使用下面的JSON串，会将表中列族f2的 COMPRESSION 配置项设为 SNAPPY，DATA_BLOCK_ENCODING 设为 PREFIX：

```
{
  "tableName" : "jsondoc:jtable",
  "base" : {
    "families" : [ {
      "FAMILY" : "f2",
      "DATA_BLOCK_ENCODING" : "PREFIX",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "SNAPPY",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false
  },
  "fulltextindex" : {
  },
  "globalindex" : {
    "indexs" : [ ]
  },
  "lob" : {
    "indexs" : [ ]
  },
  "localindex" : {
```

```
    "indexs" : [ ]
  }
}
```

例 13.4. 建表的同时创建全局索引

假设Hyperbase中没有jsondoc:jtable_gi这张表，使用下面的JSON串会创建这张表，同时为表建全局索引index_g：

```
{
    "tableName": "jsondoc:jtable_gi",
    "base" : {
        "families" : [ {
            "FAMILY" : "f1",
        } ]
    },
    "globalindex": {
        "indexs": [
            {
                "INDEX_NAME": "index_g",
                "SPLIT_KEYS": ["a", "b" ],
                "UPDATE": false,
                "DCOP": true,
                "INDEX_CLASS": "COMBINE_INDEX",
                "indexColumnInfos": [
                    {
                        "FAMILY": "f1",
                        "QUALIFY": "g",
                        "SEGMENT_LENGTH": 8
                    },
                    {
                       "FAMILY": "rowKey",
                       "QUALIFY": "rowKey",
                       "SEGMENT_LENGTH": 10
                    }
                ]
            }
        ]
    }
}
```

例 13.5. 建表的同时创建全文索引

假设Hyperbase中没有jsondoc:jtable_fu这张表，使用下面的JSON串会创建这张表，同时为表建全文索引：

```
{
    "tableName": "jsondoc:jtable_fu",
    "base" : {
        "families" : [ {
            "FAMILY" : "f1",
```

```
        } ]
    },

    "fulltextindex": {
        "tableName":"jsondoc:jtable_fu",
        "fields":[{
            "family":"f1",
            "qualifier":"q1",
            "encode_as_string" : true,
            "attributes" : {
                "type" : "string"
            }
   }
}
```

例 13.6. 建表的同时创建LOB索引

假设Hyperbase中没有jsondoc:jtable_lob这张表，使用下面的JSON串会创建这张表，同时为表建LOB索引LOBP：

```
{
    "tableName": "jsondoc:jtable_lob",
    "base" : {
      "families" : [ {
        "FAMILY" : "f1",
    } ]
    },

    "lob": {
        "indexs": [
            {
                "INDEX_NAME": "LOBP",
                "DCOP": true,
                "INDEX_CLASS": "LOB_INDEX",
                "indexColumnInfos": [
                    {
                        "FAMILY": "f1",
                        "QUALIFY": ""
                    }
                ]
            }
        ]
    }
}
```

JSON串的复用可以帮助您轻松迁移表的扩展元信息，也就是使用一张已有表的JSON串来创建一张扩展元信息完全相同的表——您只需将 `describeInJson` 的结果输出到文件中，拷贝这个文件，再通过 `alterUseJson` 用这个文件建表，这在集群迁移中非常实用，

14. JSON配置迁移扩展元数据

在将表从一个集群迁移到另一个集群的时候，我们常常不仅希望能够迁移表的元数据，还希望迁移表索引的元数据。现在，通过复用表的JSON串就可以轻松地做到这一点——您只需将 `describeInJson` 的结果输出到文件中，拷贝这个文件，再通过 `alterUseJson` 用这个文件建表，便可以在新集群中建一张扩展元数据完全相同的表。下面我们分别演示如何迁移普通Hyperbase表和Hyperdrive表的扩展元数据。

例 14.1. 迁移普通Hyperbase表的扩展元数据

假设我们要迁移一张名为jsondoc:jtable的表。

1. 将jsondoc:jtable的扩展元信息输出到本地的/tmp/jtable.txt文件中：

```
describeInJson 'jsondoc:jtable', 'true', '/tmp/jtable.txt', 'true'
```

请格外留意最后一个参数的设置，保证Split Keys的正常输出。

我们可以打开文件查看jtable.txt中的信息：

```
{
  "tableName" : "jsondoc:jtable",
  "base" : {
    "SPLIT_KEYS" : [ "31", "32", "33" ],
    "families" : [ {
      "FAMILY" : "f1",
      "DATA_BLOCK_ENCODING" : "NONE",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "NONE",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false
  },
  "fulltextindex" : {
  },
  "globalindex" : {
    "indexs" : [ ]
  },
  "lob" : {
    "indexs" : [ ]
  },
  "localindex" : {
    "indexs" : [ ]
```

```
  }
}
```

2. 将jtable.txt文件拷贝到新集群的/tmp目录下。
3. 在新集群的Hyperbase命令行中执行下面指令建表：

```
alterUseJson 'jtable_copy', '/tmp/jtable.txt', 'true'
```

请再次留意最后一个参数的设置，保证Split Keys的正常输入。

4. 现在jtable_copy表已经在新集群中建好，它的扩展元数据和jtable完全相同。我们可以将它的扩展元数据打印到文件jtable_copy.txt：

```
describeInJson 'jtable_copy', '/tmp/jtable_copy.txt', 'true'
```

然后查看文件中的信息：

```
{
  "tableName" : "jtable_copy",
  "base" : {
    "SPLIT_KEYS" : [ "31", "32", "33" ],
    "families" : [ {
      "FAMILY" : "f1",
      "DATA_BLOCK_ENCODING" : "NONE",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "NONE",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false
  },
  "fulltextindex" : {
  },
  "globalindex" : {
    "indexs" : [ ]
  },
  "lob" : {
    "indexs" : [ ]
  },
  "localindex" : {
    "indexs" : [ ]
  }
}
```

我们发现除了表名以外，其他所有的信息和jtable.txt完全一致。

5. 扩展元数据迁移成功。

例 14.2. Hyperdrive表的迁移

前面我们提到不推荐使用JSON配置来手工配置一张Hyperdrive表的扩展元数据，但是如果您已经有一张使用Hyperdrive SQL配置好的Hyperdrive表，下面将介绍的操作将使得迁移Hyperdrive表的扩展元数据变得简单。

假设我们想要迁移Inceptor的hyper数据库一张名为test_table的Hyperdrive表，它的Hyperdrive SQL中的建表语句如下：

```
CREATE TABLE test_table(
    key STRING,
    name STRING,
    info STRUCT<
        add:STRING LENGTH 20,
        age:INT,
        height:DOUBLE
    >,
    balance DOUBLE
)
STORED AS HYPERDRIVE
TBLPROPERTIES('hbase.table.splitkey'='10,20,30'); // 表test_table的Split Keys
```

在Inceptor中可以用 DESCRIBE FORMATTED test_table 查看它的元数据：

Inceptor中显示的test_table元数据

```
DESCRIBE FORMATTED test_table;
# col_name              data_type             comment
 notNull_constraint   unique_constraint

key                   string    // 从这里往下四行是test_table表的schema。
name                  string
info                  struct<add:string,age:int,height:double>
balance               double

# Detailed Table Information
Database:             hyper
Owner:                hive
CreateTime:           Thu Mar 24 19:36:42 CST 2016
LastAccessTime:       UNKNOWN
Protect Mode:         None
Retention:            0
Location:             hdfs://service/inceptorsql1/user/hive/warehouse/hyper.db/
hive/test_table
Table Type:           MANAGED_TABLE
Table Parameters:
 hbase.table.name      hyper:test_table
 hyperdrive.virtual.column _vc
 hyperdrive.virtual.family f
 storage_handler       io.transwarp.hyperdrive.HyperdriveStorageHandler
 transient_lastDdlTime 1458819402

# Storage Information
SerDe Library:        io.transwarp.hyperdrive.serde.HyperdriveSerDe
InputFormat:          io.transwarp.hyperdrive.HyperdriveInputFormat
```

```
OutputFormat:           org.apache.hadoop.hive.ql.io.HivePassThroughOutputFormat
Compressed:             No
Num Buckets:            -1
Bucket Columns:         []
Sort Columns:           []
Storage Desc Params:
 hbase.columns.mapping :key,f:q1,f:q2,f:q3 //和Hyperbase中的hyper:test_table表列之间的映射。
 hyperdrive.structstring.length.info.add 20
 serialization.format 1

# Hyperbase Information
> LocalIndex   //局部索引信息
    IndexName: name_balance_local_index
    IndexInfo: Index Columns: [name(8), balance(9)] | Attach Columns: []

> GlobalIndex  //全局索引信息
    IndexName: name_balance_global_index
    IndexInfo: Index Columns: [name(8), balance(9)] | Attach Columns: []

    IndexName: name_global_index
    IndexInfo: Index Columns: [name(8)] | Attach Columns: []

> FulltextIndex  //全文索引信息
    IndexName: elasticsearch_hyper:test_table
    IndexInfo: Index Columns: [name (doc_values: false), balance
 (doc_values: false)] | shards:
```

1. 现在在Hyperbase Shell中用 `describeInJson` 将hyper:test_table表的扩展元信息打印到文件中：

```
describeInJson 'test_table', 'true', '/tmp/test_table.txt', 'true'
```

我们可以打开test_table.txt文件查看test_table的扩展元数据，您可以将下面的信息和Inceptor中显示的test_table元数据对比：

test_table的扩展元数据

```
{
  "tableName" : "hyper:test_table",
  "base" : {
    "SPLIT_KEYS" : [ "3130", "3230", "3330" ],  // test_table的Split Keys，以十六进制字符表示。
    "families" : [ {
      "FAMILY" : "f",
      "DATA_BLOCK_ENCODING" : "PREFIX",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "SNAPPY",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
```

```
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false,
    "schema" : {  // test_table表base模块的的schema信息。
      "rowkey" : {
        "primary" : {
          "name" : "STRING"
        }
      },
      "columns" : [ {
        "family" : "f",
        "qualifier" : "q1",
        "type" : {
          "primary" : {
            "name" : "STRING"
          }
        }
      }, {
        "family" : "f",
        "qualifier" : "q2",
        "type" : {
          "struct" : [ {
            "primary" : {
              "name" : "STRING"
            },
            "cutLength" : 20
          }, {
            "primary" : {
              "name" : "INTEGER"
            }
          }, {xinxi
            "primary" : {
              "name" : "DOUBLE"
            }
          } ]
        }
      }, {
        "family" : "f",
        "qualifier" : "q3",
        "type" : {
          "primary" : {
            "name" : "DOUBLE"
          }
        }
      } ]
    }
  },
  "fulltextindex" : {  // 全文索引信息
    "tableName" : "hyper:test_table",
    "allowUpdate" : true,
    "ttl" : 0,
    "source" : true,
    "all" : false,
    "storeAsSource" : false,
    "storeFamily" : "",
    "writeConsistencyLevel" : "default",
    "fields" : [ {
      "family" : "f",
```

```
      "qualifier" : "q1",
      "encode_as_string" : false,
      "attributes" : {
        "index" : "not_analyzed",
        "store" : "true",
        "doc_values" : "false",
        "type" : "string"
      }
    }, {
      "family" : "f",
      "qualifier" : "q3",
      "encode_as_string" : false,
      "attributes" : {
        "index" : "not_analyzed",
        "store" : "true",
        "doc_values" : "false",
        "type" : "double"
      }
    } ]
  },
  "globalindex" : {  //全局索引信息
    "indexs" : [ {
      "INDEX_NAME" : "name_balance_global_index",
      "UPDATE" : "true",
      "DCOP" : "true",
      "INDEX_CLASS" : "COMBINE_INDEX",
      "indexColumnInfos" : [ {
        "FAMILY" : "f",
        "QUALIFY" : "q1",
        "SEGMENT_LENGTH" : 8
      }, {
        "FAMILY" : "f",
        "QUALIFY" : "q3",
        "SEGMENT_LENGTH" : 9
      }, {
        "FAMILY" : "rowKey",
        "QUALIFY" : "rowKey",
        "SEGMENT_LENGTH" : 16
      } ],
      "attachingColumnInfos" : [ ]
    }, {
      "INDEX_NAME" : "name_global_index",
      "UPDATE" : "true",
      "DCOP" : "true",
      "INDEX_CLASS" : "COMBINE_INDEX",
      "indexColumnInfos" : [ {
        "FAMILY" : "f",
        "QUALIFY" : "q1",
        "SEGMENT_LENGTH" : 8
      }, {
        "FAMILY" : "rowKey",
        "QUALIFY" : "rowKey",
        "SEGMENT_LENGTH" : 16
      } ],
      "attachingColumnInfos" : [ ]
    } ]
  },
  "lob" : {
```

```
    "indexs" : [ ]
  },
  "localindex" : {  // 局部索引信息
    "indexs" : [ {
      "INDEX_NAME" : "name_balance_local_index",
      "UPDATE" : "true",
      "DCOP" : "true",
      "INDEX_CLASS" : "COMBINE_INDEX",
      "DATA_BLOCK_ENCODING" : "PREFIX",
      "hbase.half.hfile.reader.class" :
 "org.apache.hadoop.hbase.io.LocalIndexSplitHalfStoreFileReader",
      "BLOOMFILTER" : "ROW",
      "TTL" : "2147483647",
      "SPLIT_FAMILY" : "false",
      "IN_MEMORY" : "false",
      "REPLICATION_SCOPE" : "0",
      "COMPRESSION" : "SNAPPY",
      "VERSIONS" : "1",
      "MIN_VERSIONS" : "0",
      "hbase.half.hfile.reader.parse.class" :
 "\n7org.apache.hadoop.hyperbase.secondaryindex.CombineIndex
\u0010\u0001\u0018\u0001\"K\nA\n\u0001f
\u0012\u0002q1\u001A8\n6\n4org.apache.hadoop.hyperbase.datatype.StringHDataType
\u0010####\u000F\u0018\b\"J\n@\n\u0001f
\u0012\u0002q3\u001A7\n5\n3org.apache.hadoop.hyperbase.datatype.DoubleDataType
\u0010####\u000F\u0018\t\"T\nJ\n\u0006rowKey\u0012\u0006rowKey
\u001A8\n6\n4org.apache.hadoop.hyperbase.datatype.StringHDataType\u0010####
\u000F\u0018\u0010",
      "KEEP_DELETED_CELLS" : "FALSE",
      "FILL_FAMILY_READ_WRITE_DEFAULT" : "\u0000",
      "BLOCKSIZE" : "65536",
      "BLOCKCACHE" : "true",
      "indexColumnInfos" : [ {
        "FAMILY" : "f",
        "QUALIFY" : "q1",
        "SEGMENT_LENGTH" : 8
      }, {
        "FAMILY" : "f",
        "QUALIFY" : "q3",
        "SEGMENT_LENGTH" : 9
      }, {
        "FAMILY" : "rowKey",
        "QUALIFY" : "rowKey",
        "SEGMENT_LENGTH" : 16
      } ],
      "attachingColumnInfos" : [ ]
    } ]
  }
}
```

2. 将文件test_table.txt拷贝到新的集群上的/tmp目录下。
3. 在新集群中使用 `alterUseJson` 建表：

```
alterUseJson 'test_table_2', 'true', '/tmp/test_table_2.txt', 'true'
```

我们可以打开文件test_table_2.txt查看其中的信息：

```
{
  "tableName" : "test_table_2",
  "base" : {
    "SPLIT_KEYS" : [ "3130", "3230", "3330" ],
    "families" : [ {
      "FAMILY" : "f",
      "DATA_BLOCK_ENCODING" : "PREFIX",
      "BLOOMFILTER" : "ROW",
      "REPLICATION_SCOPE" : "0",
      "VERSIONS" : "1",
      "COMPRESSION" : "SNAPPY",
      "MIN_VERSIONS" : "0",
      "TTL" : "2147483647",
      "KEEP_DELETED_CELLS" : "FALSE",
      "BLOCKSIZE" : "65536",
      "IN_MEMORY" : "false",
      "BLOCKCACHE" : "true"
    } ],
    "THEMIS_ENABLE" : false,
    "schema" : {
      "rowkey" : {
        "primary" : {
          "name" : "STRING"
        }
      },
      "columns" : [ {
        "family" : "f",
        "qualifier" : "q1",
        "type" : {
          "primary" : {
            "name" : "STRING"
          }
        }
      }, {
        "family" : "f",
        "qualifier" : "q2",
        "type" : {
          "struct" : [ {
            "primary" : {
              "name" : "STRING"
            },
            "cutLength" : 20
          }, {
            "primary" : {
              "name" : "INTEGER"
            }
          }, {
            "primary" : {
              "name" : "DOUBLE"
            }
          } ]
        }
      }, {
        "family" : "f",
        "qualifier" : "q3",
        "type" : {
```

```
        "primary" : {
          "name" : "DOUBLE"
        }
      }
    } ]
  }
},
"fulltextindex" : {
  "tableName" : "hyper:test_table",
  "allowUpdate" : true,
  "ttl" : 0,
  "source" : true,
  "all" : false,
  "storeAsSource" : false,
  "storeFamily" : "",
  "writeConsistencyLevel" : "default",
  "fields" : [ {
    "family" : "f",
    "qualifier" : "q1",
    "encode_as_string" : false,
    "attributes" : {
      "index" : "not_analyzed",
      "store" : "true",
      "doc_values" : "false",
      "type" : "string"
    }
  }, {
    "family" : "f",
    "qualifier" : "q3",
    "encode_as_string" : false,
    "attributes" : {
      "index" : "not_analyzed",
      "store" : "true",
      "doc_values" : "false",
      "type" : "double"
    }
  } ]
},
"globalindex" : {
  "indexs" : [ {
    "INDEX_NAME" : "name_balance_global_index",
    "UPDATE" : "true",
    "DCOP" : "true",
    "INDEX_CLASS" : "COMBINE_INDEX",
    "indexColumnInfos" : [ {
      "FAMILY" : "f",
      "QUALIFY" : "q1",
      "SEGMENT_LENGTH" : 8
    }, {
      "FAMILY" : "f",
      "QUALIFY" : "q3",
      "SEGMENT_LENGTH" : 9
    }, {
      "FAMILY" : "rowKey",
      "QUALIFY" : "rowKey",
      "SEGMENT_LENGTH" : 16
    } ],
    "attachingColumnInfos" : [ ]
```

```
    }, {
      "INDEX_NAME" : "name_global_index",
      "UPDATE" : "true",
      "DCOP" : "true",
      "INDEX_CLASS" : "COMBINE_INDEX",
      "indexColumnInfos" : [ {
        "FAMILY" : "f",
        "QUALIFY" : "q1",
        "SEGMENT_LENGTH" : 8
      }, {
        "FAMILY" : "rowKey",
        "QUALIFY" : "rowKey",
        "SEGMENT_LENGTH" : 16
      } ],
      "attachingColumnInfos" : [ ]
    } ]
  },
  "lob" : {
    "indexs" : [ ]
  },
  "localindex" : {
    "indexs" : [ {
      "INDEX_NAME" : "name_balance_local_index",
      "UPDATE" : "true",
      "DCOP" : "true",
      "INDEX_CLASS" : "COMBINE_INDEX",
      "DATA_BLOCK_ENCODING" : "PREFIX",
      "hbase.half.hfile.reader.class" :
 "org.apache.hadoop.hbase.io.LocalIndexSplitHalfStoreFileReader",
      "BLOOMFILTER" : "ROW",
      "TTL" : "2147483647",
      "SPLIT_FAMILY" : "false",
      "IN_MEMORY" : "false",
      "REPLICATION_SCOPE" : "0",
      "COMPRESSION" : "SNAPPY",
      "VERSIONS" : "1",
      "MIN_VERSIONS" : "0",
      "hbase.half.hfile.reader.parse.class" :
 "\n7org.apache.hadoop.hyperbase.secondaryindex.CombineIndex
\u0010\u0001\u0018\u0001\"K\nA\n\u0001f
\u0012\u0002q1\u001A8\n6\n4org.apache.hadoop.hyperbase.datatype.StringHDataType
\u0010####\u000F\u0018\b\"J\n@\n\u0001f
\u0012\u0002q3\u001A7\n5\n3org.apache.hadoop.hyperbase.datatype.DoubleDataType
\u0010####\u000F\u0018\t\"T\nJ\n\u0006rowKey\u0012\u0006rowKey
\u001A8\n6\n4org.apache.hadoop.hyperbase.datatype.StringHDataType\u0010####
\u000F\u0018\u0010",
      "KEEP_DELETED_CELLS" : "FALSE",
      "FILL_FAMILY_READ_WRITE_DEFAULT" : "\u0000",
      "BLOCKSIZE" : "65536",
      "BLOCKCACHE" : "true",
      "indexColumnInfos" : [ {
        "FAMILY" : "f",
        "QUALIFY" : "q1",
        "SEGMENT_LENGTH" : 8
      }, {
        "FAMILY" : "f",
        "QUALIFY" : "q3",
        "SEGMENT_LENGTH" : 9
```

```
      }, {
        "FAMILY" : "rowKey",
        "QUALIFY" : "rowKey",
        "SEGMENT_LENGTH" : 16
      } ],
      "attachingColumnInfos" : [ ]
    } ]
  }
}
```

对比test_table的扩展元数据，我们发现两份信息除了表名以外，完全相同。

4. 现在test_table的扩展元数据已经顺利迁移进了新集群的Hyperbase中的test_table_2上，但是新集群的Inceptor中还没有登记test_table_2，所以我们还不能在Inceptor中通过Hyperdrive SQL操作test_table_2。我们需要 **在新集群上的Inceptor中用Hyperdrive SQL登记test_table_2（建外表映射到test_table_2）：**

```
CREATE EXTERNAL TABLE test_table_2(
    key STRING,
    name STRING,
    info STRUCT<
        add:STRING LENGTH 20,
        age:INT,
        height:DOUBLE
    >,
    balance DOUBLE
)
STORED BY 'io.transwarp.hyperdrive.HyperdriveStorageHandler'
WITH SERDEPROPERTIES('hbase.columns.mapping'=':key,f:q1,f:q2,f:q3')
TBLPROPERTIES('hbase.table.name'='test_table_2');
```

5. 我们可以使用 `DESCRIBE FORMATTED test_table_2` 查看test_table_2在Inceptor中显示的元数据：

```
DESCRIBE FORMATTED test_table_2;
# col_name              data_type           comment
 notNull_constraint   unique_constraint

key                   string
name                  string
info                  struct<add:string,age:int,height:double>
balance               double

# Detailed Table Information
Database:             hyper
Owner:                hive
CreateTime:           Thu Mar 24 22:12:21 CST 2016
LastAccessTime:       UNKNOWN
Protect Mode:         None
Retention:            0
Location:             hdfs://service/inceptorsql1/user/hive/warehouse/
hyper.db/hive/test_table_2
Table Type:           EXTERNAL_TABLE
Table Parameters:
 EXTERNAL              TRUE
```

```
 hbase.table.name     test_table_2
 hyperdrive.virtual.column _vc
 hyperdrive.virtual.family f
 storage_handler      io.transwarp.hyperdrive.HyperdriveStorageHandler
 transient_lastDdlTime 1458828741

# Storage Information
SerDe Library:       io.transwarp.hyperdrive.serde.HyperdriveSerDe
InputFormat:         io.transwarp.hyperdrive.HyperdriveInputFormat
OutputFormat:
 org.apache.hadoop.hive.ql.io.HivePassThroughOutputFormat
Compressed:          No
Num Buckets:         -1
Bucket Columns:      []
Sort Columns:        []
Storage Desc Params:
 hbase.columns.mapping :key,f:q1,f:q2,f:q3
 hyperdrive.structstring.length.info.add 20
 serialization.format 1

# Hyperbase Information
> LocalIndex
    IndexName: name_balance_local_index
    IndexInfo: Index Columns: [name(8), balance(9)] | Attach Columns: []

> GlobalIndex
    IndexName: name_balance_global_index
    IndexInfo: Index Columns: [name(8), balance(9)] | Attach Columns: []

    IndexName: name_global_index
    IndexInfo: Index Columns: [name(8)] | Attach Columns: []

> FulltextIndex
    IndexName: elasticsearch_hyper:test_table
    IndexInfo: Index Columns: [name (doc_values: false), balance
 (doc_values: false)] | shards:
```

对比Inceptor中显示的test_table元数据，我们可以看到test_table_2和test_table的元数据除了表名、建表时间和是否为外表以外完全相同，尤其是 **各类索引的信息完全相同——得益于JSON串的复用，我们省去了用Hyperdrive SQL重新创建索引的步骤**。

6. 扩展元数据迁移成功。

客户服务

技术支持

感谢你使用星环信息科技（上海）有限公司的产品和服务。如您在产品使用或服务中有任何技术问题，可以通过以下途径找到我们的技术人员给予解答。

Email：support@transwarp.io

技术支持热线电话：4008 079 976

技术支持QQ专线：3221723229, 3344341586

官方网址：www.transwarp.io

意见反馈

如果你在系统安装，配置和使用中发现任何产品问题，可以通过以下方式反馈：

Email：support@transwarp.io

感谢你的支持和反馈，我们一直在努力！

关于我们：

星环信息科技（上海）有限公司是一家大数据领域的高科技公司，致力于大数据基础软件的研发。星环科技目前掌握的企业级Hadoop和Spark核心技术在国内独树一帜，其产品Transwarp Data Hub (TDH)的整体架构及功能特性堪比硅谷同行，在业界居于领先水平，性能大幅领先Apache Hadoop，可处理从GB到PB级别的数据。星环科技的核心开发团队参与部署了国内最早的Hadoop集群，并在中国的电信、金融、交通、政府等领域的落地应用拥有丰富经验，是中国大数据核心技术企业化应用的开拓者和实践者。星环科技同时提供存储、分析和挖掘大数据的高效数据平台 和服务，立志成为国内外领先的大数据核心技术厂商。

行业地位：

来自知名外企的创业团队，成功完成近千万美元的A轮融资，经验丰富的企业级Hadoop发行版开发团队，国内最多落地案例。

核心技术：

高性能、完善的SQL on Hadoop、R语言的并行化支持，为企业数据分析与挖掘提供优秀选择。

应用案例：

已成功部署多个关键行业领域，包括电信、电力、智能交通、工商管理、税务、金融、广电、电商、物流等。

地址：上海市徐汇区桂平路481号18幢3层301室（漕河泾新兴技术开发区）
邮编：200233
电话：4008-079-976
网址：www.transwarp.io