HybridDB for PostgreSQL

用户指南

用户指南

基础操作

对基于 Greenplum Database 的操作，HybridDB for PostgreSQL 与 Greenplum Database 开源数据库基本一致，包括 schema、类型支持、用户权限等。除 Greenplum Database 开源数据库一些特有的操作，如分布键、AO表等，其它操作皆可参见 PostgreSQL。

本文介绍了 HybridDB for PostgreSQL 的基础操作：创建数据库、创建分布键和查询。

创建数据库

HybridDB for PostgreSQL 中创建数据库的操作与 PostgreSQL 相同，可以通过 SQL 来执行，如在 **psql** 连接到 Greenplum 后执行如下命令：

```
=> create database mygpdb;
CREATE DATABASE

=> \c mygpdb
psql (9.4.4, server 8.3devel)
You are now connected to database "mygpdb" as user "mygpdb".
```

创建分布键

在 HybridDB for PostgreSQL 中，表分布在所有的 Segment 上，其分布规则是 HASH 或者随机。在建表时，指定分布键；当导入数据时，会根据分布键计算得到的 HASH 值分配到特定的 Segment 上。

```
=> create table vtbl(id serial, key integer, value text, shape cuboid, location geometry, comment text) distributed by (key);
CREATE TABLE
```

当不指定分布键时（即不带后面的“distributed by (key)”时），Greenplum 会默认对 id 字段以 round-robin 的方式进行随机分配。

关于分布键

分布键对于查询性能至关重要。“均匀”为分布键选择的第一大原则，选取更有业务意义的字段，将有助于显著提高性能。

尽量选择分布均匀的列、或者多列，以防止数据倾斜。

尽量选择常用于连接运算的字段，对于并发较高的语句，该选择更为重要。

尽量选择高并发查询、过滤性高的条件列。

尽量不要轻易使用随机分布。

详细内容，请参见本文档 参考 中内容。

构造数据

创建函数，用于生成随机字符串。

```
CREATE OR REPLACE FUNCTION random_string(integer) RETURNS text AS $body$
SELECT array_to_string(array
(SELECT substring('0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
FROM (ceil(random()*62))::int
FOR 1)
FROM generate_series(1, $1)), '');
$body$
LANGUAGE SQL VOLATILE;
```

创建分布键。

```
CREATE TABLE tbl(id serial, KEY integer, locate geometry, COMMENT text) distributed by (key);
```

构造数据。

```
INSERT INTO tbl(KEY, COMMENT, locate)
SELECT
KEY,
COMMENT,
ST_GeomFromText(locate) AS locate
FROM
(SELECT
(a + 1) AS KEY,
random_string(ceil(random() * 24)::integer) AS COMMENT,
'POINT(' || ceil(random() * 36 + 99) || ' ' || ceil(random() * 24 + 50) || ')' AS locate
FROM
```

```
generate_series(0, 99999) AS a)
AS t;
```

查询示例

查询

```
  => select * from tbl where key = 751;
| id | key | value | shape | locate | comment |
+-----+-----+-------+-------------+----------------------------------------+----------------+
| 751 | 751 | red | 0101000000000000000000C05B400000000000004A40 | B9hPhjeNWPqV |

(1 row)

Time: 513.101 ms
```

查询计划

```
  => explain select * from tbl where key = 751;
Gather Motion 1:1 (slice1; segments: 1) (cost=0.00..1519.28 rows=1 width=53)
-> Seq Scan on tbl (cost=0.00..1519.28 rows=1 width=53)
Filter: key = 751
Settings: effective_cache_size=8GB; gp_statistics_use_fkeys=on
Optimizer status: legacy query optimizer
```

参考

- Pivotal Greenplum 官方文档
- GP 4.3 Best Practice
- Greenplum 数据分布黄金法则

用户权限管理

用户管理

实例创建过程中，会提示用户指定初始用户名和密码，这个初始用户为“根用户”。实例创建好后，用户可以使用该根用户连接数据库。使用 psql（PostgreSQL 或 Greenplum 的客户端工具）连接数据库后，通过

\du+ 命令可以查看所有用户的信息，示例如下：

注意：除了根用户，有几个用户也会被创建，包括 aurora、replicator 等。这些用户是超级用户，用于内部管理。

```
postgres=> \du+
List of roles
Role name | Attributes | Member of | Description
-------------+-----------------------------------+-----------+---------------
root_user | | | rds_superuser
...
```

目前，HybridDB for PostgreSQL 没有开放 SUPERUSER 权限，对应的是 RDS_SUPERUSER，这一点与云数据库 RDS（PostgreSQL）中的权限体系一致。所以，根用户（如上面的示例中的 root_user）具有 RDS_SUPERUSER 权限，这个权限属性只能通过查看用户的描述（Description）来识别。根用户具有如下权限：

具有 CREATEROLE、CREATEDB 和 LOGIN 权限，即可以用来创建数据库和用户，但没有 SUPERUSER 权限。

查看和修改其它非超级用户的数据表，执行 SELECT、UPDATE、DELTE 或更改所有者（Owner）等操作。

查看其它非超级用户的连接信息、撤销（Cancel）其 SQL 或终止（Kill）其连接。

执行 CREATE EXTENSION 和 DROP EXTENSION 命令，创建和删除插件。

创建其他具有 RDS_SUPERUSER 权限的用户，示例如下：

```
CRATE ROLE root_user2 RDS_SUPERUSER LOGIN PASSWORD 'xyz' ;
```

权限管理

用户可以在数据库（Database）、模式（Schema）、表等多个层次管理权限。例如，赋予一个用户表上的读取权限，但收回修改权限，示例如下。

```
GRANT SELECT ON TABLE t1 TO normal_user1;
REVOKE UPDATE ON TABLE t1 FROM normal_user1;
REVOKE DELETE ON TABLE t1 FROM normal_user1;
```

参考文档

关于具体的用户与权限管理方法，请参见 Managing Roles and Privileges。

插件管理

云数据库 HybridDB for PostgreSQL 基于 Greenplum Database 开源数据库项目开发，由阿里云深度扩展，是一种在线的分布式云数据库，由多个 计算组组成，可提供大规模并行处理（MPP）数据仓库的服务。

插件类型

云数据库 HybridDB for PostgreSQL 支持如下插件：

PostGIS：支持地理信息数据。

MADlib：机器学习方面的函数库。

fuzzystrmatch：字符串模糊匹配。

orafunc：兼容 Oracle 的部分函数。

oss_ext：支持从 OSS 读取数据。

hll：支持用 HyperLogLog 算法进行统计。

pljava：支持使用 PL/Java 语言编写用户自定义函数（UDF）。

pgcrypto：支持加密函数。

intarray：整数数组相关的函数、操作符和索引支持。

创建插件

创建插件的方法如下所示：

```
CREATE EXTENSION <extension name>;
CREATE SCHEMA <schema name>;
```

```
CREATE EXTENSION IF NOT EXISTS <extension name> WITH SCHEMA <schema name>;
```

注意：创建 MADlib 插件时，需要先创建 plpythonu 插件，如下所示：

```
CREATE EXTENSION plpythonu;
CREATE EXTENSION madlib;
```

删除插件

删除插件的方法如下所示：

注意：如果插件被其它对象依赖，需要加入 CASCADE（级联）关键字，删除所有依赖对象。

```
DROP EXTENSION <extension name>;
DROP EXTENSION IF EXISTS <extension name> CASCADE;
```

JSON 数据类型操作

JSON 类型几乎已成为互联网及物联网（IoT）的基础数据类型，其重要性不言而喻，具体协议请参见 JSON 官网。

PostgreSQL 对 JSON 的支持已经比较完善，阿里云深度优化云数据库 HybridDB for PostgreSQL，基于 PostgreSQL 语法进行了 JSON 数据类型的支持。

检查现有版本是否支持 JSON

执行如下命令，检查是否已经支持 JSON：

```
=> SELECT '""'::json;
```

若系统出现如下信息，则说明已经支持 JSON 类型，可以使用实例了。若执行不成功，请重新启动实例。

```
 json
------
""
(1 row)
```

若系统出现如下信息，则说明尚未支持 JSON 类型。

```
ERROR:  type "json" does not exist
```

```
LINE 1: SELECT '""'::json;
^
```

上述命令是一次从字符串到 JSON 格式的强制转换，这也基本上是其操作上的本质。

JSON 在数据库中的转换

数据库的操作主要分为读和写，JSON 的数据写入一般是字符串到 JSON。字符串中的内容必须符合 JSON 标准，包括字符串、数字、数组、对象等内容。如：

字符串

```
=> SELECT '"hijson"'::json;
json
-------
"hijson"
(1 row)
```

::在 PostgreSQL/Greenplum/HybridDB for PostgreSQL 中代表强制类型转换。在此转换的时候，会调用 JSON 类型的输入函数。因此，类型转换时同样会做 JSON 格式的检查，如下所示：

```
=> SELECT '{hijson:1024}'::json;

ERROR: invalid input syntax for type json
LINE 1: SELECT '{hijson:1024}'::json;
^
DETAIL: Token "hijson" is invalid.
CONTEXT: JSON data, line 1: {hijson...
=>
```

上述"hijson"两边的"是必不可少的，因为在标准中，KEY 值对应的是一个字符串，所以这里的{hijson:1024}在语法上会报错。

除了类型上的强制转换，还有从数据库记录到 JSON 串的转换。

我们正常使用 JSON，不会只用一个 String 或一个 Number，而是一个包含一个或多个键值对的对象。所以，对 Greenplum 而言，支持到对象的转换，即支持了 JSON 的绝大多数场景，如：

```
=> select row_to_json(row('{"a":"a"}', 'b'));
row_to_json
---------------------------------
{"f1":"{\"a\":\"a\"}","f2":"b"}
(1 row)

=> select row_to_json(row('{"a":"a"}'::json, 'b'));
row_to_json
----------------------------
{"f1":{"a":"a"},"f2":"b"}
(1 row)
```

由此也可以看出字符串和 JSON 的区别，这样就可以很方便地将一整条记录转换成 JSON。

JSON 内部数据类型的定义

对象

对象是 JSON 中最常用的，如：

```
  => select '{"key":"value"}'::json;
json
-----------------
{"key":"value"}
(1 row)
```

整数 & 浮点数

JSON 的协议只有三种数字：整数、浮点数和常数表达式，当前 Greenplum 对这三种都有很好的支持。

```
  => SELECT '1024'::json;
json
------
1024
(1 row)

 => SELECT '0.1'::json;
json
------
0.1
(1 row)
```

特殊情况下，需要如下信息：

```
  => SELECT '1e100'::json;
json
-------
1e100
(1 row)

 => SELECT '{"f":1e100}'::json;
json
-------------
{"f":1e100}
(1 row)
```

并且，包括下面这个长度超长的数字：

```
  => SELECT '9223372036854775808'::json;
json
----------------------
9223372036854775808
(1 row)
```

数组

```
  => SELECT '[[1,2], [3,4,5]]'::json;
 json
 ------------------
 [[1,2], [3,4,5]]
 (1 row)
```

操作符

JSON 支持的操作符类型

```
=> select oprname,oprcode from pg_operator where oprleft = 3114;
oprname | oprcode
---------+---------------------------
-> | json_object_field
->> | json_object_field_text
-> | json_array_element
->> | json_array_element_text
#> | json_extract_path_op
#>> | json_extract_path_text_op
(6 rows)
```

基本使用方法

```
=> SELECT '{"f":"1e100"}'::json -> 'f';
?column?
----------
"1e100"
(1 row)

=> SELECT '{"f":"1e100"}'::json ->> 'f';
?column?
----------
1e100
(1 row)

=> select '{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}'::json#>array['f4','f6'];
?column?
-----------
"stringy"
```

```
(1 row)

=> select '{"f2":{"f3":1},"f4":{"f5":99,"f6":"stringy"}}'::json#>'{f4,f6}';
?column?
-----------
"stringy"
(1 row)

=> select '{"f2":["f3",1],"f4":{"f5":99,"f6":"stringy"}}'::json#>>'{f2,0}';
?column?
----------
f3
(1 row)
```

JSON 函数

支持的函数

```
postgres=# \df *json*
List of functions
Schema | Name | Result data type | Argument data types | Type
------------+--------------------------+-----------------+---------------------------------------------------------------+-----
---
pg_catalog | array_to_json | json | anyarray | normal
pg_catalog | array_to_json | json | anyarray, boolean | normal
pg_catalog | json_array_element | json | from_json json, element_index integer | normal
pg_catalog | json_array_element_text | text | from_json json, element_index integer | normal
pg_catalog | json_array_elements | SETOF json | from_json json, OUT value json | normal
pg_catalog | json_array_length | integer | json | normal
pg_catalog | json_each | SETOF record | from_json json, OUT key text, OUT value json | normal
pg_catalog | json_each_text | SETOF record | from_json json, OUT key text, OUT value text | normal
pg_catalog | json_extract_path | json | from_json json, VARIADIC path_elems text[] | normal
pg_catalog | json_extract_path_op | json | from_json json, path_elems text[] | normal
pg_catalog | json_extract_path_text | text | from_json json, VARIADIC path_elems text[] | normal
pg_catalog | json_extract_path_text_op | text | from_json json, path_elems text[] | normal
pg_catalog | json_in | json | cstring | normal
pg_catalog | json_object_field | json | from_json json, field_name text | normal
pg_catalog | json_object_field_text | text | from_json json, field_name text | normal
pg_catalog | json_object_keys | SETOF text | json | normal
pg_catalog | json_out | cstring | json | normal
pg_catalog | json_populate_record | anyelement | base anyelement, from_json json, use_json_as_text boolean |
normal
pg_catalog | json_populate_recordset | SETOF anyelement | base anyelement, from_json json, use_json_as_text
boolean | normal
pg_catalog | json_recv | json | internal | normal
pg_catalog | json_send | bytea | json | normal
pg_catalog | row_to_json | json | record | normal
pg_catalog | row_to_json | json | record, boolean | normal
pg_catalog | to_json | json | anyelement | normal
(24 rows)
```

基本使用方法

```
=> SELECT array_to_json('{{1,5},{99,100}}'::int[]);
array_to_json
------------------
[[1,5],[99,100]]
(1 row)

=> SELECT row_to_json(row(1,'foo'));
row_to_json
---------------------
{"f1":1,"f2":"foo"}
(1 row)

=> SELECT json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]');
json_array_length
-------------------
5
(1 row)

=> select * from json_each('{"f1":[1,2,3],"f2":{"f3":1},"f4":null,"f5":99,"f6":"stringy"}') q;
key | value
-----+-----------
f1 | [1,2,3]
f2 | {"f3":1}
f4 | null
f5 | 99
f6 | "stringy"
(5 rows)

=> select json_each_text('{"f1":[1,2,3],"f2":{"f3":1},"f4":null,"f5":"null"}');
json_each_text
-------------------
(f1,"[1,2,3]")
(f2,"{""f3"":1}")
(f4,)
(f5,null)
(4 rows)

=> select json_array_elements('[1,true,[1,[2,3]],null,{"f1":1,"f2":[7,8,9]},false]');
json_array_elements
-----------------------
1
true
[1,[2,3]]
null
{"f1":1,"f2":[7,8,9]}
false
(6 rows)

create type jpop as (a text, b int, c timestamp);
=> select * from json_populate_record(null::jpop,'{"a":"blurfl","x":43.2}', false) q;
a | b | c
--------+---+---
blurfl | |
```

```
(1 row)

=> select * from json_populate_recordset(null::jpop,'[{"a":"blurfl","x":43.2},{"b":3,"c":"2012-01-20 10:42:53"}]',false) q;
a | b | c
--------+---+---------------------------
blurfl | |
| 3 | Fri Jan 20 10:42:53 2012
(2 rows)
```

完整操作事例

创建表

```
create table tj(id serial, ary int[], obj json, num integer);

=> insert into tj(ary, obj, num) values('{1,5}'::int[], '{"obj":1}', 5);
INSERT 0 1
=> select row_to_json(q) from (select id, ary, obj, num from tj) as q;
row_to_json
-----------------------------------------------
{"f1":1,"f2":[1,5],"f3":{"obj":1},"f4":5}
(1 row)

=> insert into tj(ary, obj, num) values('{2,5}'::int[], '{"obj":2}', 5);
INSERT 0 1

=> select row_to_json(q) from (select id, ary, obj, num from tj) as q;
row_to_json
-----------------------------------------------
{"f1":1,"f2":[1,5],"f3":{"obj":1},"f4":5}
{"f1":2,"f2":[2,5],"f3":{"obj":2},"f4":5}
(2 rows)
```

多表 JOIN

```
create table tj2(id serial, ary int[], obj json, num integer);

=> insert into tj2(ary, obj, num) values('{2,5}'::int[], '{"obj":2}', 5);
INSERT 0 1

=> select * from tj, tj2 where tj.obj->>'obj' = tj2.obj->>'obj';
id | ary | obj | num | id | ary | obj | num
----+-------+-----------+-----+----+-------+-----------+-----
2 | {2,5} | {"obj":2} | 5 | 1 | {2,5} | {"obj":2} | 5
(1 row)

=> select * from tj, tj2 where json_object_field_text(tj.obj, 'obj') = json_object_field_text(tj2.obj, 'obj');
id | ary | obj | num | id | ary | obj | num
----+-------+-----------+-----+----+-------+-----------+-----
2 | {2,5} | {"obj":2} | 5 | 1 | {2,5} | {"obj":2} | 5
```

```
(1 row)
```

JSON 函数索引

```
CREATE TEMP TABLE test_json (
json_type text,
obj json
);

=> insert into test_json values('aa', '{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}');
INSERT 0 1

=> insert into test_json values('cc', '{"f7":{"f3":1},"f8":{"f5":99,"f6":"foo"}}');
INSERT 0 1

=> select obj->'f2' from test_json where json_type = 'aa';
?column?
----------
{"f3":1}
(1 row)

=> create index i on test_json (json_extract_path_text(obj, '{f4}'));
CREATE INDEX

=> select * from test_json where json_extract_path_text(obj, '{f4}') = '{"f5":99,"f6":"foo"}';
json_type | obj
-----------+--------------------------------------------
aa | {"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}
(1 row)
```

注意：JSON 类型暂时不能支持作为分布键来使用；也不支持 JSON 聚合函数。

下面是 Python 访问的一个例子：

```
#! /bin/env python

import time
import json
import psycopg2

def gpquery(sql):
conn = None
try:
conn = psycopg2.connect("dbname=sanity1x2")
conn.autocommit = True
cur = conn.cursor()
cur.execute(sql)
return cur.fetchall()
except Exception as e:
if conn:
try:
conn.close()
except:
```

```
pass
time.sleep(10)
print e
return None


def main():
sql = "select obj from tj;"
#rows = Connection(host, port, user, pwd, dbname).query(sql)
rows = gpquery(sql)
for row in rows:
print json.loads(row[0])


if __name__ == "__main__":
main()
```

HyperLogLog 的使用

阿里云深度优化云数据库 HybridDB for PostgreSQL，除原生 Greenplum Database 功能外，还支持 HyperLogLog，为互联网广告分析及有类似预估分析计算需求的行业提供解决方案，以便于快速预估 PV、UV 等业务指标。

创建 HyperLogLog 插件

执行如下命令，创建 HyperLogLog 插件：

CREATE EXTENSION hll;

基本类型

执行如下命令，创建一个含有 hll 字段的表：

create table agg (id int primary key,userids hll);

执行如下命令，进行 int 转 hll_hashval：

select 1::hll_hashval;

基本操作符

hll 类型支持 =、!=、<>、|| 和 #。

```
select hll_add_agg(1::hll_hashval) = hll_add_agg(2::hll_hashval);
select hll_add_agg(1::hll_hashval) || hll_add_agg(2::hll_hashval);
select #hll_add_agg(1::hll_hashval);
```

hll_hashval 类型支持 =、!= 和 <>。

```
select 1::hll_hashval = 2::hll_hashval;
select 1::hll_hashval <> 2::hll_hashval;
```

基本函数

hll_hash_boolean、hll_hash_smallint 和 hll_hash_bigint 等 hash 函数。

```
select hll_hash_boolean(true);
select hll_hash_integer(1);
```

hll_add_agg：可以将 int 转 hll 格式。

```
select hll_add_agg(1::hll_hashval);
```

hll_union：hll 并集。

```
select hll_union(hll_add_agg(1::hll_hashval),hll_add_agg(2::hll_hashval));
```

hll_set_defaults：设置精度。

```
select hll_set_defaults(15,5,-1,1);
```

hll_print：用于 debug 信息。

```
select hll_print(hll_add_agg(1::hll_hashval));
```

示例

```
create table access_date (acc_date date unique, userids hll);
insert into access_date select current_date, hll_add_agg(hll_hash_integer(user_id)) from generate_series(1,10000)
t(user_id);
insert into access_date select current_date-1, hll_add_agg(hll_hash_integer(user_id)) from
generate_series(5000,20000) t(user_id);
insert into access_date select current_date-2, hll_add_agg(hll_hash_integer(user_id)) from
generate_series(9000,40000) t(user_id);
postgres=# select #userids from access_date where acc_date=current_date;
?column?
------------------
9725.85273370708
(1 row)
postgres=# select #userids from access_date where acc_date=current_date-1;
?column?
------------------
14968.6596883279
(1 row)
postgres=# select #userids from access_date where acc_date=current_date-2;
?column?
------------------
29361.5209149911
(1 row)
```

使用 Create Library 命令

为支持用户导入自定义软件包，HybridDB for PostgreSQL 引入了 Create Library/Drop Library 命令。使用此命令创建 PL/Java 的 UDF 的示例，请参见 PL/Java UDF 的使用。

本文介绍了 Create/Drop Library 命令的使用方法。

语法

```
CREATE LIBRARY library_name LANGUAGE [JAVA] FROM oss_location OWNER ownername
CREATE LIBRARY library_name LANGUAGE [JAVA] VALUES file_content_hex OWNER ownername
DROP LIBRARY library_name
```

参数说明：

- library_name：要安装的库的名称。若已安装的库与要安装的库的名称相同，必须先删除现有的库，然后再安装新库。
- LANGUAGE [JAVA]：要使用的语言。目前仅支持 PL/Java。
- oss_location：包文件的位置。您可以指定 OSS 存储桶和对象名称，仅可以指定一个文件，且不能为压缩文件。其格式为：

```
oss://oss_endpoint filepath=[folder/[folder/]...]/file_name id=userossid key=userosskey bucket=ossbucket
```

- file_content_hex：文件内容，字节流为 16 进制。例如，73656c6563742031表示” select 1” 的 16 进制字节流。借助这个语法，可以直接导入包文件，不必通过 OSS。
- ownername：指定用户。
- DROP LIBRARY：删除一个库。

示例

示例 1：安装名为 analytics.jar 的 jar 包。

```
create library example language java from 'oss://oss-cn-hangzhou.aliyuncs.com filepath=analytics.jar
id=xxx key=yyy bucket=zzz';
```

示例 2：直接导入文件内容，字节流为 16 进制。

```
create library  pglib LANGUAGE java VALUES '73656c6563742031' OWNER "myuser";
```

示例 3：删除一个库。

```
drop library example;
```

示例 4：查看已经安装的库。

```
select name, lanname from pg_library;
```

使用 PL/Java UDF

HybridDB for PostgreSQL 支持用户使用 PL/Java 语言，编写并上传 jar 软件包，并利用这些 jar 包创建用户自定义函数（UDF）。该功能支持的 PL/Java 语言版本为社区版 PL/Java 1.5.0，使用的 JVM 版本为 1.8。

本文介绍了创建 PL/Java UDF 的示例步骤。更多的 PL/Java 样例，请参见 PL/Java代码（查看 编译方法）。

操作步骤

在 HybridDB for PostgreSQL 中，执行如下命令，创建 PL/Java 插件（每个数据库只需执行一次）。

```
create extension pljava;
```

根据业务需要，编写自定义函数。例如，使用如下代码编写 Test.java 文件：

```
public class Test
{
public static String substring(String text, int beginIndex,
int endIndex)
{
return text.substring(beginIndex, endIndex);
}
}
```

编写 manifest.txt 文件。

```
Manifest-Version: 1.0
Main-Class: Test
Specification-Title: "Test"
Specification-Version: "1.0"
Created-By: 1.7.0_99
Build-Date: 01/20/2016 21:00 AM
```

执行如下命令，将程序编译打包。

```
javac Test.java
jar cfm analytics.jar manifest.txt Test.class
```

将步骤 4 生成的 analytics.jar 文件，通过 OSS 控制台命令上传到 OSS。

```
osscmd put analytics.jar oss://zzz
```

在 HybridDB for PostgreSQL 中，执行 Create Library 命令，将文件导入到 HybridDB for PostgreSQL 中。

注意：Create Library 只支持 filepath，一次导入一个文件。另外，Create Library 还支持字节流形式，可以不通过 OSS 直接导入，详情请参见 Create Library 命令的使用。

```
create library example language java from 'oss://oss-cn-hangzhou.aliyuncs.com filepath=analytics.jar
id=xxx key=yyy bucket=zzz';
```

在 HybridDB for PostgreSQL 中，执行如下命令，创建和使用相应 UDF。

```
create table temp (a varchar) distributed randomly;
insert into temp values ('my string');
create or replace function java_substring(varchar, int, int) returns varchar as 'Test.substring' language
java;
select java_substring(a, 1, 5) from temp;
```

重启实例

为了更好地满足用户的需求，HybridDB for PostgreSQL 会不断更新数据库内核版本，详细的发布历史见 **发布历史**。您在实例在创建时，默认使用的是最新版本的数据库内核。当新的版本发布之后，您可以通过重启实例来更新数据库内核版本，从而使用新版本中扩展的功能。本文介绍了重启实例的方法。

操作步骤

您可以按照下面的步骤来重启实例：

登录 云数据库 HybridDB for PostgreSQL 管理控制台。

选择要操作实例所在的地域。例如，华东1。

单击目标实例右侧操作栏中的 **管理** 按钮，进入基本信息页面。

单击页面右上角的 **重启实例**，并在确认框中单击 **确定**。如果您绑定了手机，还需要进行手机验证码验证。

注意：重启过程一般耗时3到30分钟，在此过程中该实例不能对外提供服务，请您提前做出调整。当实例重启结束，对应实例恢复运行中状态，您可以正常访问数据库。

完成了上述操作后，您可以回到控制台查看目标实例的运行状态。如果实例重启操作完成，则实例运行状态为 **运行中**，否则为 **重启中**。

升级实例规格

在使用云数据库 HybridDB for PostgreSQL 过程中，随着您的数据量和计算量的动态增长，一些计算资源如CPU、磁盘、内存以及数据处理节点数量将成为数据处理速度的瓶颈。为了支持实例的动态扩展，HybridDB for PostgreSQL 提供在线升级实例规格的功能，目前暂不支持对实例的降级操作。本文介绍了升级实例规格的相关操作。

查看当前实例规格

HybridDB for PostgreSQL 实例规格包括 计算组规格 和 计算组数量，各种计算组规格详细说明见 实例规格。

您可以通过以下的步骤来查看当前的实例规格：

登录 云数据库 HybridDB for PostgreSQL 管理控制台。

选择要操作实例所在的地域。例如，华东1。

单击目标实例右侧操作栏中的 **管理** 按钮，进入基本信息页面。

在基本信息页面的 **配置信息** 中显示了当前的实例规格，包括计算组规格、计算组详情、计算组数量、计算资源汇总。

HybridDB for PostgreSQL 目前提供两类规格：

- 高性能，规格名称以gpdb.group.**segsdx**开始，提供更好的I/O能力，带来更高的性能。
- 高容量，规格名称以gpdb.group.**seghdx**开始，提供更大、更实惠的空间，满足更高的存储需要。

升级实例规格

当确认需要升级实例规格后，您可以按照下面的步骤进行实例规格的升级：

登录 云数据库 HybridDB for PostgreSQL 管理控制台。

选择要操作实例所在的地域。例如，华东1。

单击目标实例右侧操作栏中的 **升级** 按钮，进入 **确认订单** 页面。

选择合适的计算组规格和计算组数量，点击 **去开通** 按钮。

目前，HybridDB for PostgreSQL 支持不同的计算组规格和计算组数量组合，详情参见 **规则组合速查表**。在选择新的计算组规格和计算组数量组合时，要满足以下约束：

- 新旧计算组规格必须为同一类计算组规格，而且新的计算组规格 >= 旧的计算组规格
- 如果新的计算组规格 = 旧的计算组规格，新的计算组数量 > 旧的计算组数量

在选择计算组规格和计算组数量的时候，除了满足以上约束，您还需要对自己的数据量和计算量进行评估，从而选择合适的计算组规格和计算组数量组合，详情可参考 **如何选择实例规格**。

注意：根据您数据量的不同，实例规格升级的过程大约需要30分钟到3个小时的时间。在此过程中，为了保证数据的一致性，您实例只对外提供只读服务，并且会闪断两次，请您提前做出调整。当升级实例规格结束，对应实例恢复运行中状态，您可以正常访问数据库，而且实例的数据库内核版本自动升级为最新。

在完成了上述操作之后，您可以回到控制台，查看目标实例的运行状态。如果实例规格升级操作完成，则实例运行状态为 **运行中**，否则为 **升降级中**。