阿里云
MaxCompute

用户指南

文档版本：20180918

法律声明

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

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。
2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。
3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。
4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的”现状“、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。
5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含”阿里云”、Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。
6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **禁止：** 重置操作将丢失用户配置数据。
	该类警示信息可能导致系统重大变更甚至故障，或者导致人身伤害等结果。	 **警告：** 重启操作将导致业务中断，恢复业务所需时间约10分钟。
	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	 **说明：** 您也可以通过按**Ctrl** + **A**选中全部文件。
>	多级菜单递进。	设置 > 网络 > 设置网络类型
粗体	表示按键、菜单、页面名称等UI元素。	单击 **确定**。
`courier` 字体	命令。	执行 `cd /d C:/windows` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *`Instance_ID`*
[]或者[a|b]	表示可选项，至多选择一个。	`ipconfig` *`[-all|-t]`*
{}或者{a|b}	表示必选项，至多选择一个。	`swich` *`{stand | slave}`*

目录

1 Java沙箱

MaxCompute MapReduce及UDF程序在分布式环境中运行时，受到Java沙箱的限制（MapReduce作业的主程序，例如MR Main则不受此限制），具体限制如下所示。

- 不允许直接访问本地文件，只能通过MaxCompute MapReduce/Graph提供的接口间接访问。
  - 读取resources选项指定的资源，包括文件、Jar包和资源表等。
  - 通过System.out和System.err输出日志信息，可以通过MaxCompute客户端的Log命令查看日志信息。
- 不允许直接访问分布式文件系统，只能通过MaxCompute MapReduce/Graph访问到表的记录。
- 不允许JNI调用限制。
- 不允许创建Java线程，不允许启动子进程执行Linux命令。
- 不允许访问网络，包括获取本地IP地址等，都会被禁止。
- Java反射限制：suppressAccessChecks权限被禁止，无法setAccessible某个private的属性或方法，以达到读取private属性或调用private方法的目的。

在代码中，直接使用下文访问本地文件的方法会报access denied异常。

- java.io.File

```
public boolean delete()
public void deleteOnExit()
public boolean exists()
public boolean canRead()
public boolean isFile()
public boolean isDirectory()
public boolean isHidden()
public long lastModified()
public long length()
public String[] list()
public String[] list(FilenameFilter filter)
public File[] listFiles()
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
public boolean canWrite()
public boolean createNewFile()
public static File createTempFile(String prefix, String suffix)
public static File createTempFile(String prefix,  String suffix,File
 directory)
public boolean mkdir()
public boolean mkdirs()
public boolean renameTo(File dest)
public boolean setLastModified(long time)
```

```
public boolean setReadOnly()
```

- java.io.RandomAccessFile

```
RandomAccessFile(String name, String mode)
RandomAccessFile(File file, String mode)
```

- java.io.FileInputStream

```
FileInputStream(FileDescriptor fdObj)
FileInputStream(String name)
FileInputStream(File file)
```

- java.io.FileOutputStream

```
FileOutputStream(FileDescriptor fdObj)
FileOutputStream(File file)
FileOutputStream(String name)
FileOutputStream(String name, boolean append)
```

- java.lang.Class

```
public ProtectionDomain getProtectionDomain()
```

- java.lang.ClassLoader

```
ClassLoader()
ClassLoader(ClassLoader parent)
```

- java.lang.Runtime

```
public Process exec(String command)
public Process exec(String command, String envp[])
public Process exec(String cmdarray[])
public Process exec(String cmdarray[], String envp[])
public void exit(int status)
public static void runFinalizersOnExit(boolean value)
public void addShutdownHook(Thread hook)
public boolean removeShutdownHook(Thread hook)
public void load(String lib)
public void loadLibrary(String lib)
```

- java.lang.System

```
public static void exit(int status)
public static void runFinalizersOnExit(boolean value)
public static void load(String filename)
public static void loadLibrary( String libname)
public static Properties getProperties()
public static void setProperties(Properties props)
public static String getProperty(String key) // 只允许部分key可以访问
public static String getProperty(String key, String def) // 只允许部分
key可以访问
public static String setProperty(String key, String value)
public static void setIn(InputStream in)
public static void setOut(PrintStream out)
public static void setErr(PrintStream err)
```

```
public static synchronized void setSecurityManager(SecurityManager s
)
```

System.getProperty允许的key列表，如下所示：

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.specification.version
java.specification.vendor
java.specification.name
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
file.encoding
user.timezone
```

- java.lang.Thread

```
Thread()
Thread(Runnable target)
Thread(String name)
Thread(Runnable target, String name)
Thread(ThreadGroup group, ...)
public final void checkAccess()
public void interrupt()
public final void suspend()
public final void resume()
public final void setPriority (int newPriority)
public final void setName(String name)
public final void setDaemon(boolean on)
public final void stop()
public final synchronized void stop(Throwable obj)
public static int enumerate(Thread tarray[])
public void setContextClassLoader(ClassLoader cl)
```

- java.lang.ThreadGroup

```
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name)
public final void checkAccess()
public int enumerate(Thread list[])
public int enumerate(Thread list[], boolean recurse)
public int enumerate(ThreadGroup list[])
public int enumerate(ThreadGroup list[], boolean recurse)
public final ThreadGroup getParent()
public final void setDaemon(boolean daemon)
public final void setMaxPriority(int pri)
public final void suspend()
public final void resume()
```

```
public final void destroy()
public final void interrupt()
public final void stop()
```

- java.lang.reflect.AccessibleObject

```
public static void setAccessible(...)
public void setAccessible(...)
```

- java.net.InetAddress

```
public String getHostName()
public static InetAddress[] getAllByName(String host)
public static InetAddress getLocalHost()
```

- java.net.DatagramSocket

```
public InetAddress getLocalAddress()
```

- java.net.Socket

```
Socket(...)
```

- java.net.ServerSocket

```
ServerSocket(...)
public Socket accept()
protected final void implAccept(Socket s)
public static synchronized void setSocketFactory(...)
public static synchronized void setSocketImplFactory(...)
```

- java.net.DatagramSocket

```
DatagramSocket(...)
public synchronized void receive(DatagramPacket p)
```

- java.net.MulticastSocket

```
MulticastSocket(...)
```

- java.net.URL

```
URL(...)
public static synchronized void setURLStreamHandlerFactory(...)
java.net.URLConnection
public static synchronized void setContentHandlerFactory(...)
public static void setFileNameMap(FileNameMap map)
```

- java.net.HttpURLConnection

```
public static void setFollowRedirects(boolean set)
java.net.URLClassLoader
```

```
URLClassLoader(...)
```

- java.security.AccessControlContext

```
public AccessControlContext(AccessControlContext acc, DomainCombiner
 combiner)
public DomainCombiner getDomainCombiner()
```

2 Lightning

2.1 Lightning概述

MaxCompute Lightning是MaxCompute产品的交互式查询服务，支持以PostgreSQL协议及语法连接访问Maxcompute项目，让您使用熟悉的工具以标准 SQL查询分析MaxCompute项目中的数据，快速获取查询结果。

您可使用主流BI工具（如Tableau、帆软等）或SQL客户端轻松连接到MaxCompute项目，开展BI分析或即席查询。或者利用MaxCompute Lightning的快速查询特性，将项目表数据封装成API对外服务，无需数据迁移就能够支持更丰富的应用场景。

MaxCompute Lightning提供无服务器计算（Serverless）的服务方式，您无需管理任何基础设施，只需为运行的查询付费。

关键特性

- 兼容PostgreSQL

  MaxCompute Lightning提供兼容PostgreSQL协议的JDBC/ODBC接口，所有支持PostgreSQL数据库的工具或应用使用默认驱动都可以轻松地连接到MaxCompute项目。您也可以使用更广泛的PostgreSQL生态工具来分析MaxCompute的数据。

- 显著提升性能

  针对MaxCompute表的快速查询进行了优化，特别是在小数据集、并发场景下有更好的性能表现。从而能够支撑更丰富的应用场景，如固定报表、API开放等。

- 统一的权限管理

  作为MaxCompute产品内的服务，通过MaxCompute Lightning连接到MaxCompute项目的访问完全遵循Maxcompute项目的权限体系，在访问用户权限范围内安全地查询数据。

- 开箱即用，按查询付费

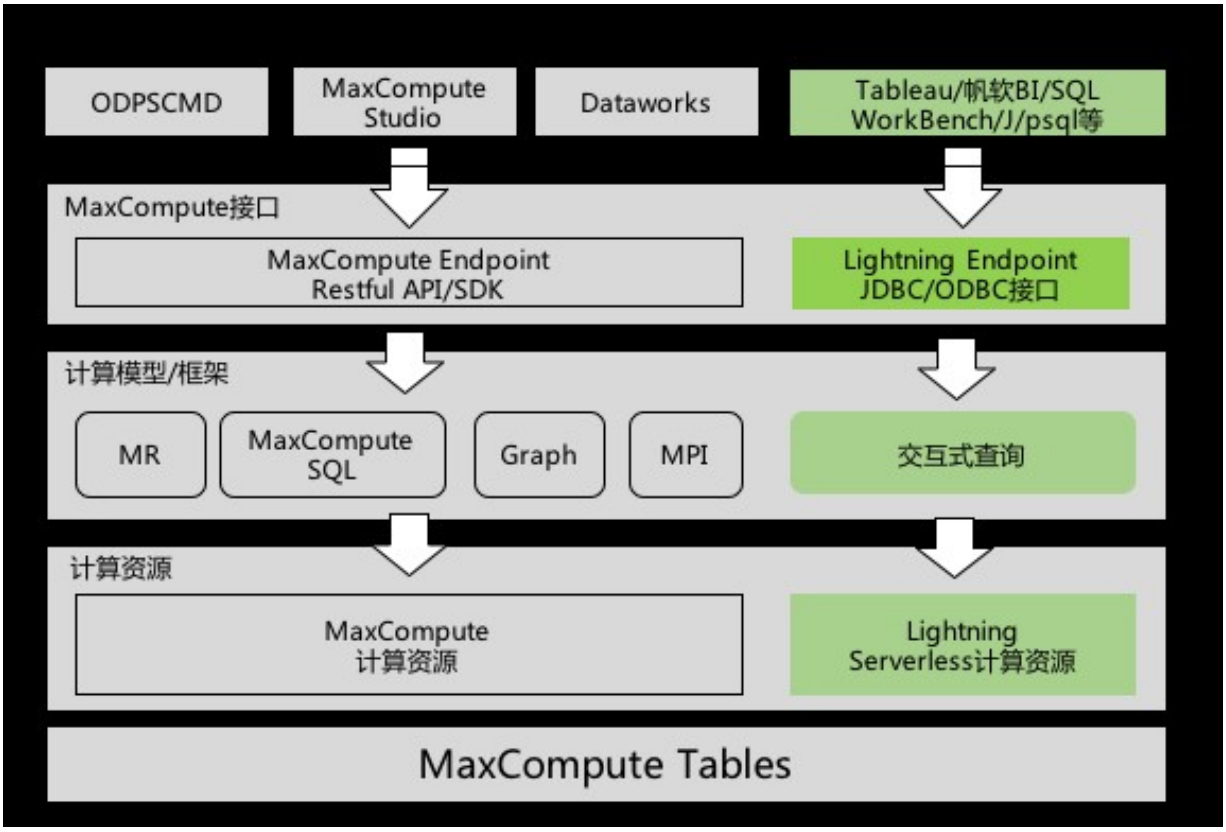
  MaxCompute Lightning是在MaxCompute已有的计算资源之外提供的无服务器的计算服务，您不需要设置、管理或运维MaxCompute Lightning资源，通过MaxCompute Lightning连接后即可开展查询。

  使用MaxCompute Lightning时，只需为每次查询所实际处理的数据量付费，不查询时不产生费用。

系统结构

作为Maxcompute的交互式查询服务，MaxCompute Lightning提供了配套的接入访问域名地址(Endpoint)，客户端工具及应用通过PostgreSQL驱动连接访问Lightning JDBC/ODBC接口服务，在MaxCompute项目统一的权限体系下安全地访问项目数据。

通过该服务接口连接并提交的查询任务，都将使用MaxCompute Lightning的Serverless计算资源以保障交互式查询的服务质量。



应用场景

- 即席查询（Ad Hoc）

  利用MaxCompute Lightning面向小规模数据集（百GB规模内）查询性能优化的特性，使用者可以直接对MaxCompute表开展低时延的查询操作，而不需要再把数据再导入到其它各种系统进行加速（比如ADS、RDS），节约资源和管理成本。

  场景特点：查询的数据对象自由不固定，逻辑相对复杂，期望快速获取查询结果并调整查询逻辑，对查询时延的要求在几十秒内。使用者往往是掌握SQL技能的数据分析师，希望使用熟悉的客户端工具来开展查询分析。

- Reporting报表分析

对MaxCompute项目中通过ETL加工汇总后的结果数据制作分析报表，提供给管理者和业务人员定期查看。

场景特点：查询的数据对象通常为聚合后的结果数据， 数据量较小、查询逻辑固定且较简单。时延要求低：秒级返回（例如大部分查询不超过5秒，不同查询根据其数据规模和查询复杂度有较大差异）。

- 面向在线应用的消费场景

  直接将MaxCompute项目中的数据封装成为restful api，支撑在线应用。

  场景特点：利用MaxCompute Lightning作为加速查询引擎，结合阿里云Dataworks的数据服务组件，零开发、无运维地将MaxCompute的表数据开放为API服务。

2.2 开通Lightning服务

MaxCompute Lightning是MaxCompute提供的交互式查询服务，您需要先开通并创建MaxCompute项目方可使用MaxCompute Lightning。

MaxCompute Lightning服务目前处于公测阶段，未对全网用户开放。如需使用，您可以通过我们在阿里云官网上提供的公测试用申请页面申请公测期间的服务开通。

商业化后，默认为MaxCompute项目开通MaxCompute Lightning服务。

2.3 服务定价

说明：

- MaxCompute Lightning服务目前处于公测阶段，完全免费，公测结束后将正式商业化收费。
- 正式商业化后，使用Lightning服务时按每个查询所扫描的数据量付费（单价待商业化时发布）。

使用MaxCompute Lightning，您只需为您运行的查询付费，根据执行查询过程中，实际扫描的MaxCompute项目表的数据量计费。当不运行查询时，MaxCompute Lightning不收取任何费用。

由于MaxCompute Lightning的使用依赖于您开通创建的MaxCompute项目，因此您还需要关注MaxCompute在数据存储、计算（按量后付费/按CU预付费）、外网下载等方面产生的费用。

计费详情请参见*MaxCompute*计量计费说明。

2.4 快速开始

2.4.1 使用说明

本教程将引导您使用主流的第三方工具连接MaxCompute Lightning服务，查看指定MaxCompute项目下的数据表，进行BI分析。

2.4.2 前提条件

开通MaxCompute并创建项目

使用MaxCompute Lightning需要您已经有创建好的MaxCompute项目。

如果您还没有开通阿里云MaxCompute项目，请参见开通*MaxCompute*进行开通并创建一个MaxCompute项目。

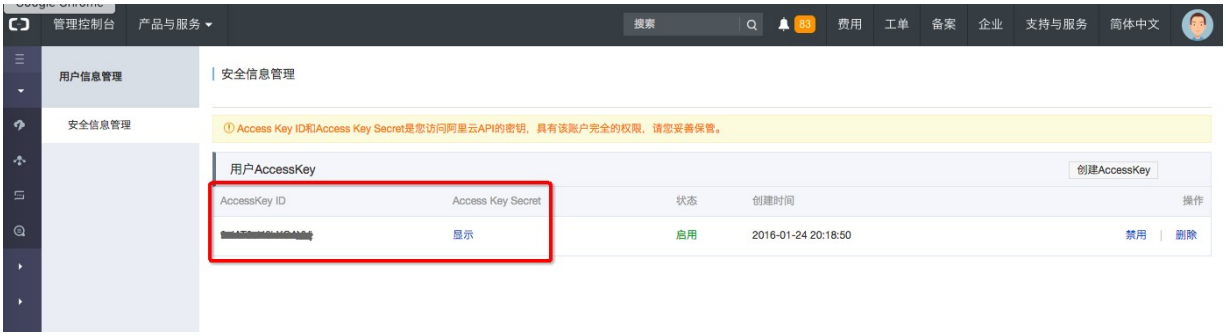
创建表并导入数据

使用MaxCompute Lightning前需要您创建表并导入数据，详情请参见*MaxCompute*快速入门。

获取云账号信息

使用MaxCompute Lightning前需要MaxCompute项目用户的Accesss ID及Access Key云账号信息。

您可登录阿里云官网，进入管理控制台用户信息管理页面进行查看。子账号如果没有查看权限，请联系主账号管理员索取，同时确定该子账号有权限查看指定的数据表。



2.4.3 准备连接的客户端工具

MaxCompute Lightning兼容PostgreSQL接口，支持PostgreSQL数据库连接的客户端工具都可以用于连接MaxCompute Lightning。

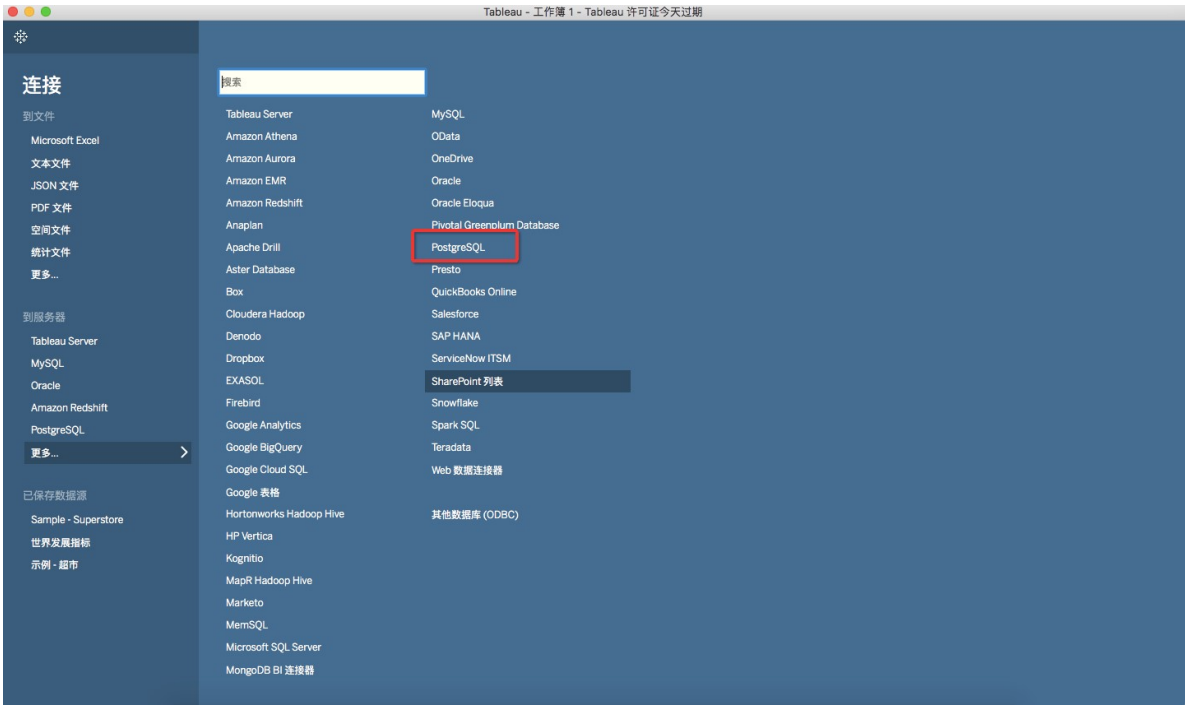
本教程中选择了大家熟悉的BI工具*Tableau Desktop* 进行示例，相关工具请到Tableau官网进行下载。

其他常见客户端工具，如SQL Workbench/J、PSQL、帆软BI、MicroStrategy BI等都可以像连接PostgreSQL数据库一样配置服务连接。

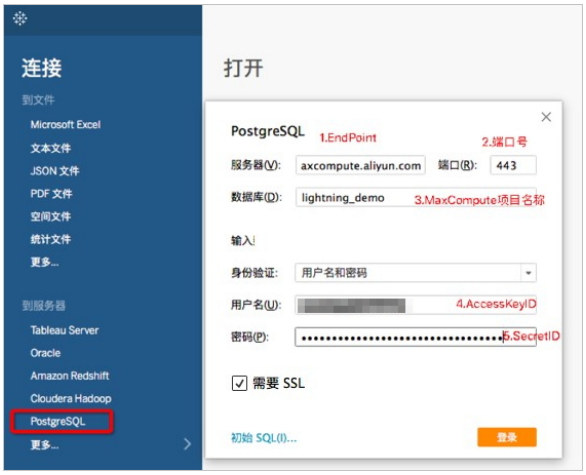
2.4.4 连接服务并开展分析

1. 连接服务器时选择PostgreSQL。

   打开Tableau Desktop，选择连接 > 到服务器 > 更多 > **PostgreSQL**。



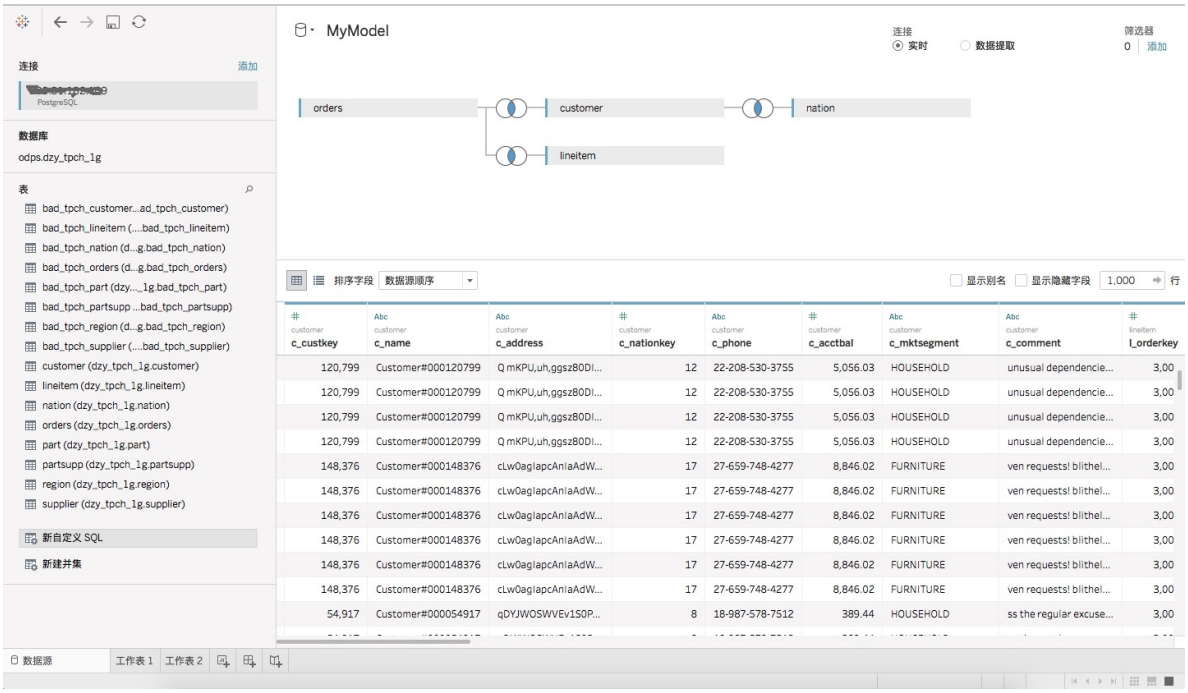
2. 填写服务连接及用户认证信息。



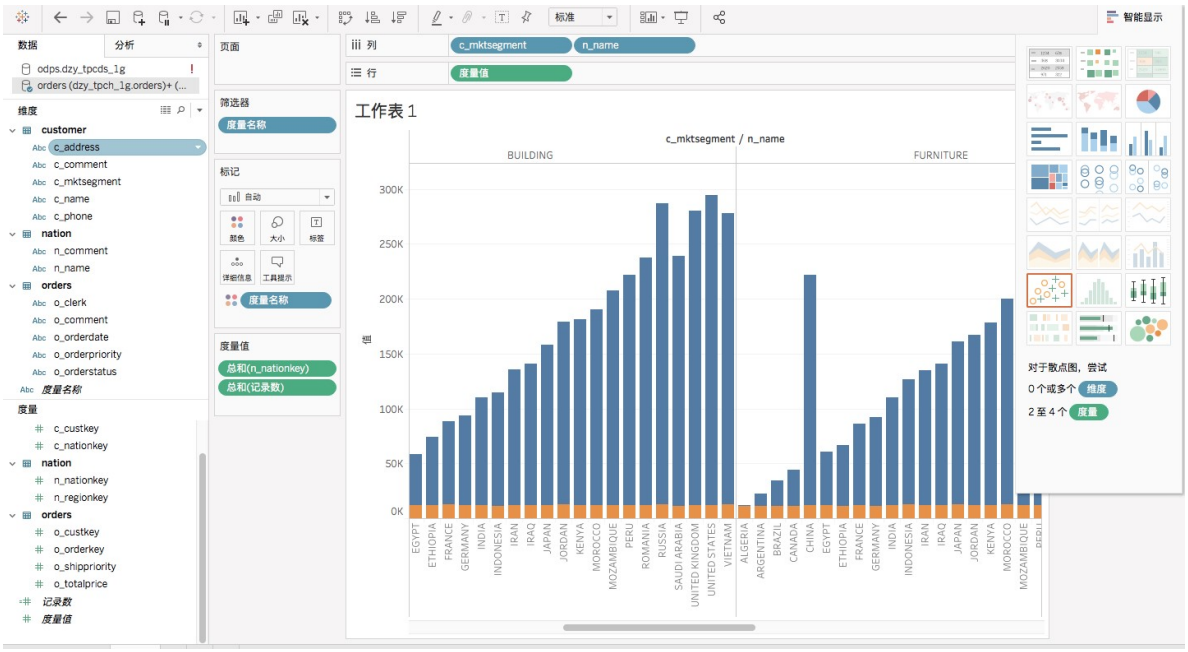
参数	说明
服务器	填写所在区域对应的MaxCompute Lightning EndPoint，如华东2区域的Endpoint为：`lightning.cn-shanghai.maxcompute.aliyun.com`
端口	443
数据库	MaxCompute项目名
身份验证	用户名和密码
用户名/密码	访问用户的Access Key ID/Access Key Secret
SSL连接	勾选需要**SSL**

3. 获取项目表信息，创建数据源/模型。

   配置好联系信息并登录后，Tableau会加载所连接的MaxCompute项目下的表，使用者可以根据需要选择对应的表创建模型和工作表。



   选择特定数据的维度和指标，创建工作表。



至此，您已经使用Tableau工具成功连接MaxCompute Lightning服务，可以对连接到MaxCompute项目内的数据进行BI分析。

> **说明：**
> 为了获得更好的性能和体验，建议您使用Tableau支持的TDC文件方式对Lightning数据源进行连接定制优化，详情请参见*Tableau Desktop*。

2.5 访问域名

MaxCompute Lightning提供了单独的域名访问地址（Endpoint），通过该地址您可以访问到阿里云不同区域的MaxCompute Lightning服务。

MaxCompute Lightning在公共云的不同Region及网络环境下的服务连接对照表如下。

表 2-1: 外网网络下Region和服务连接对照表

Region	开服状态	外网**Endpoint**
华东1	公测中	lightning.cn-hangzhou.maxcompute.aliyun.com
华东2	公测中	lightning.cn-shanghai.maxcompute.aliyun.com
华北2	公测中	lightning.cn-beijing.maxcompute.aliyun.com
亚太东南1	公测中	lightning.ap-southeast-1.maxcompute.aliyun.com
其他区域	暂未开服	-

表 2-2: 经典网络下Region和服务连接对照表

Region	开服状态	经典网络Endpoint
华东1	公测中	lightning.cn-hangzhou.maxcompute.aliyun-inc.com
华东2	公测中	lightning.cn-shanghai.maxcompute.aliyun-inc.com
华北2	公测中	lightning.cn-beijing.maxcompute.aliyun-inc.com
亚太东南1	公测中	lightning.ap-southeast-1.maxcompute.aliyun-inc.com
其他区域	暂未开服	-

表 2-3: VPC网络下Region和服务连接对照表

Region	开服状态	VPC网络Endpoint
华东1	公测中	lightning.cn-hangzhou.maxcompute.aliyun-inc.com
华东2	公测中	lightning.cn-shanghai.maxcompute.aliyun-inc.com
华北2	公测中	lightning.cn-beijing.maxcompute.aliyun-inc.com
亚太东南1	公测中	lightning.ap-southeast-1.maxcompute.aliyun-inc.com
其他区域	暂未开服	-

2.6 通过JDBC连接服务

Maxcompute Lightning查询引擎基于PostgreSQL 8.2，当前仅支持对已有MaxCompute表进行SELECT查询，更多详情请参见查询语法及函数。

如果您的MaxCompute项目还没有数据或需要对现有数据进行加工处理，请参见*MaxCompute*文档，通过*MaxCompute*客户端或*DataWorks*连接MaxCompute项目进行数据对象创建和加工处理。

2.6.1 JDBC驱动程序

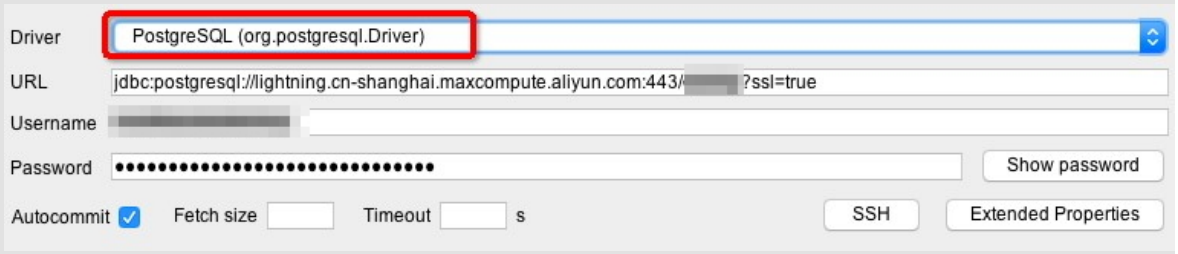
MaxCompute提供完全兼容PostgreSQL消息协议的Java数据库连接（JDBC）接口，使用者可以通过JDBC将SQL客户端工具连接到MaxCompute Lightning服务。

MaxCompute Lightning支持PostgeSQL官方驱动连接，同时也提供了为Lightning服务而优化的驱动程序供选择。

1. 使用PostgeSQL官方提供的*JDBC*驱动程序。

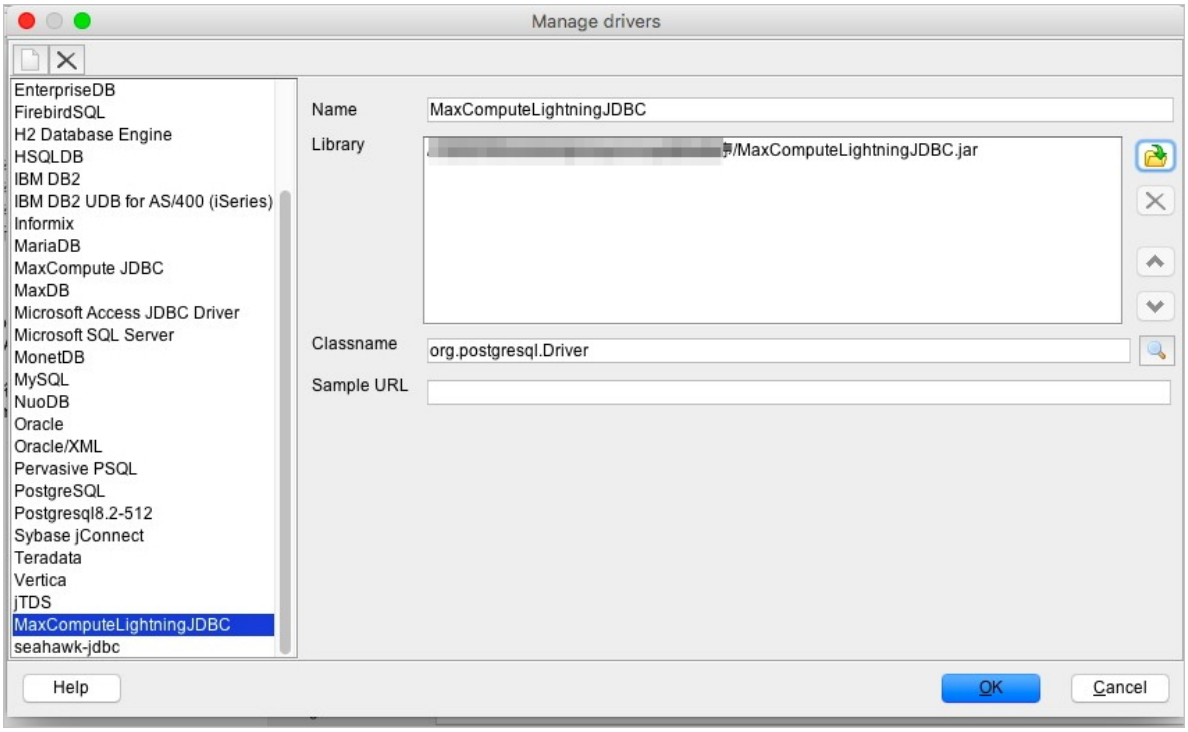
说明：

很多客户端工具默认集成了PostgreSQL数据库的驱动，直接使用工具自带的驱动即可。如果未集成，可从官网下载。以SQL Workbench/J客户端为例，创建连接时可选择PostgreSQL官方驱动。

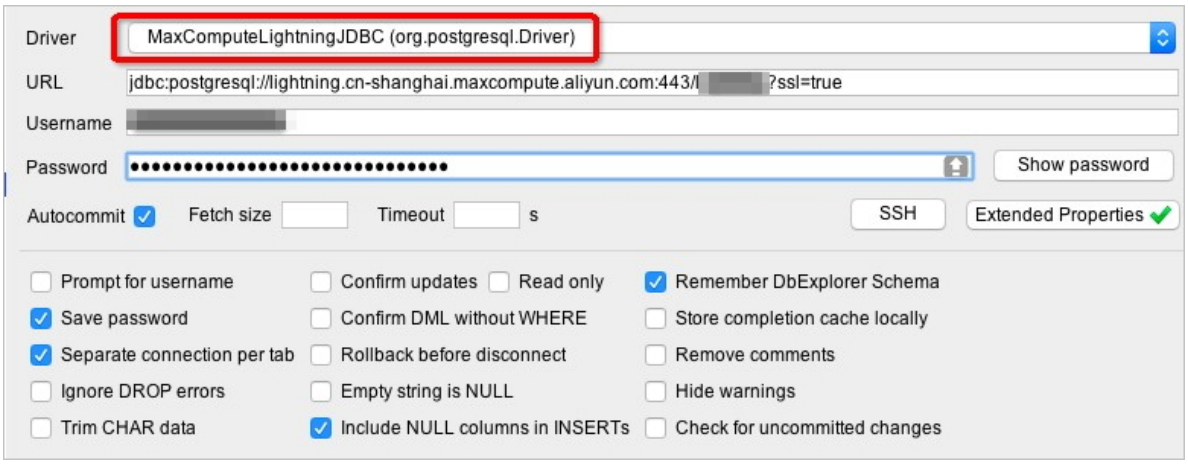


2. 使用阿里云MaxCompute Lightning优化过的*JDBC驱动*，以获取更好的性能。

   下载后的MaxCompute Lightning的JDBC驱动程序保存为MaxComputeLightningJDBC.jar文件。以SQL Workbench/J客户端为例，在驱动管理菜单中，添加MaxCompute Lightning JDBC驱动程序项。



   在创建连接时，从Driver列表中选择刚才添加的MaxCompute Lightning JDBC驱动。



2.6.2 配置JDBC连接

您需要获得您的MaxCompute项目JDBC URL，才能将SQL客户端工具连接到该项目。

JDBC URL命名方式如下：

```
jdbc:postgresql://endpoint:port/database
```

连接参数说明如下：

参数	取值	说明
endpoint	所在区域不同网络环境下的Lightning访问域名	详情请参见访问域名，例如通过外网访问上海Region的服务使用`lightning.cn-shanghai.maxcompute.aliyun.com`
port	443	-
database	填写MaxCompute的项目名称	-
user	访问用户的Access Key ID	-
password	访问用户的Access Key Secret	-
ssl	true	MaxCompute Lightning服务端默认开启SSL服务，客户端需要使用SSL进行连接。
prepareThreshold	0	可选。需要使用JDBC PrepareStatement功能时，建议设置`prepareThreshold=0`。

例如`jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/myproject`

同时，设置user、password、SSL连接参数后进行连接。

您也可以将参数添加到JDBC URL来连接到MaxCompute项目，如下所示：

```
jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/
myproject?ssl=true& prepareThreshold=0&user=xxx&password=yyy
```

说明如下：

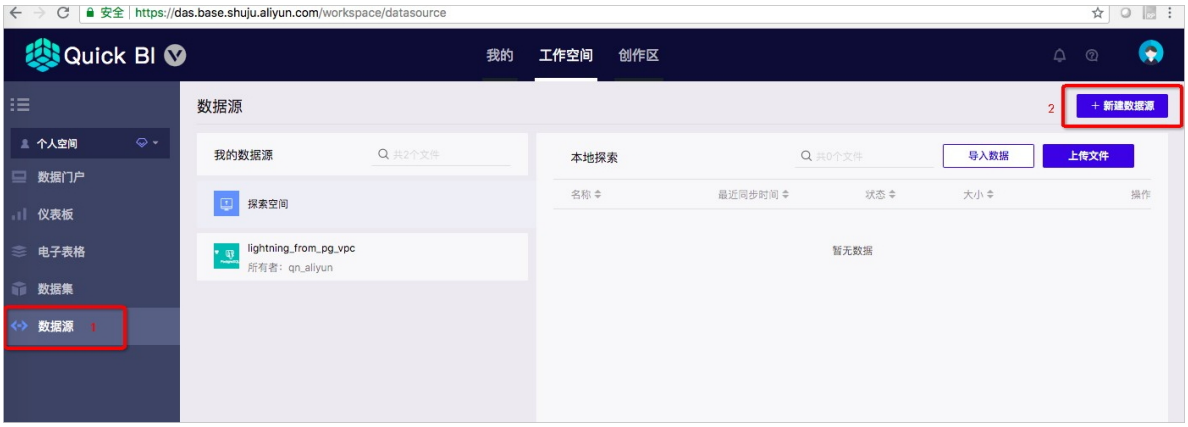
- lightning.cn-shanghai.maxcompute.aliyun.com是华东2区域的Endpoint。
- myproject是需要访问的MaxCompute项目名称。
- ssl=true是指定通过SSL方式连接。
- xxx是访问用户的Access Key ID。
- yyy是访问用户的Access Key Secret。

2.6.3 常见工具的连接

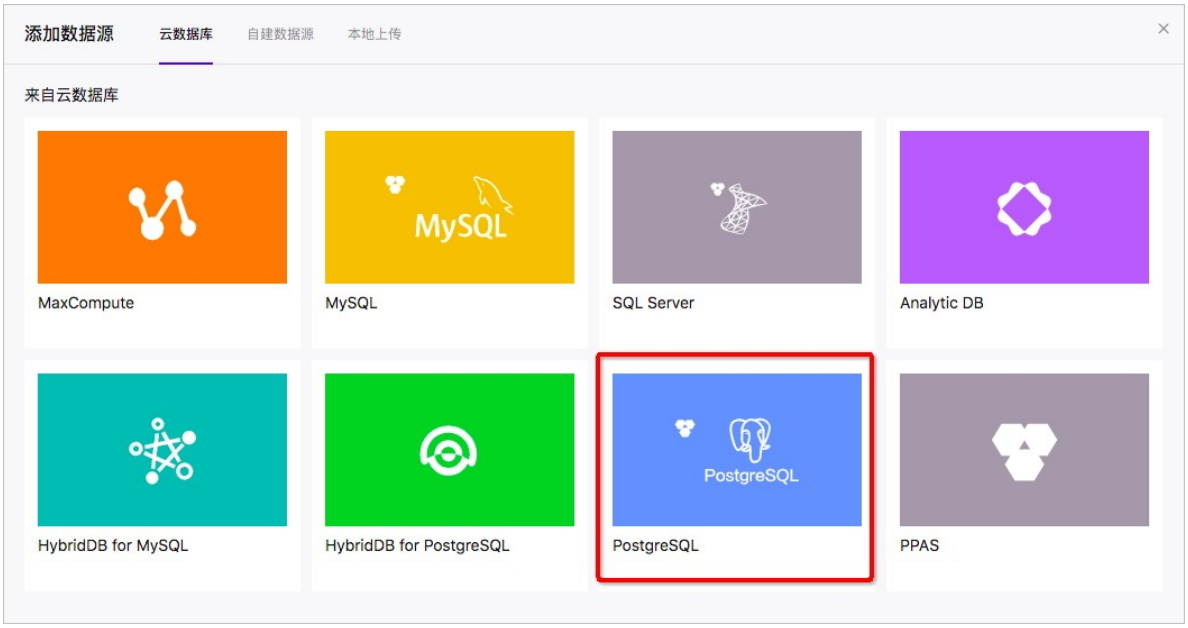
本文将为您介绍几款常见客户端工具的接入说明，除此之外，原则上支持PostgresSQL的工具都可以通过Lightning来对接访问MaxCompute。

阿里云**Quick BI**

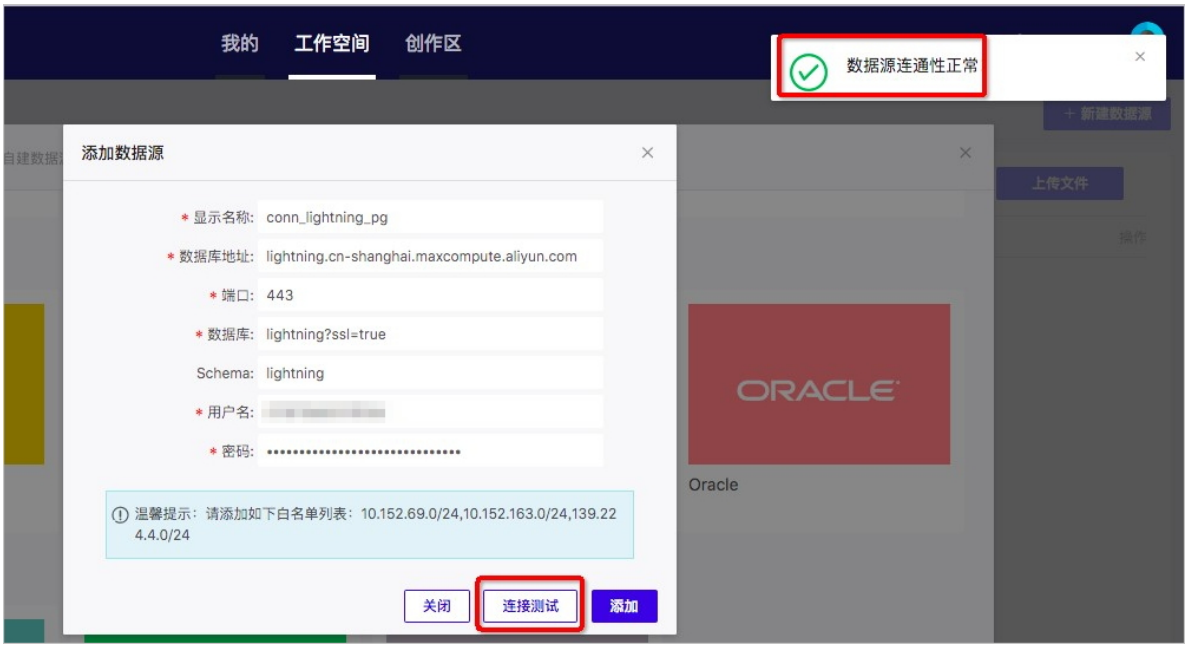
1. 登录Quick BI控制台，单击左侧导航栏中的**数据源**。
2. 单击数据源管理页面右上角的**新建数据源**。



3. 选择云数据库或自建数据源中的PostgeSQL数据库类型添加数据源。



4. 填写对话框中的MaxCompute Lightning的连接信息并测试连接连通状态。



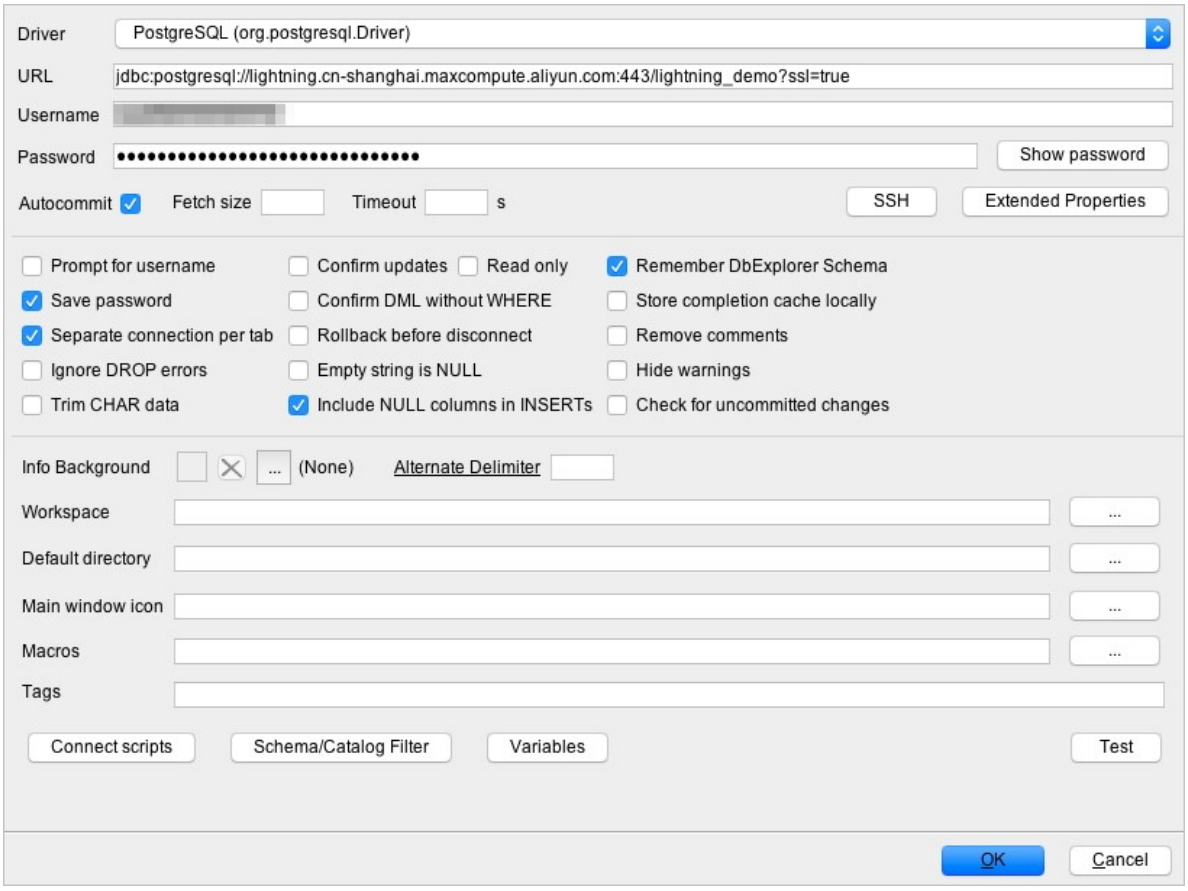
参数	说明
数据库地址	MaxCompute Lightning对应区域的Endpoint，可使用公网访问的Endpoint，也可以使用经典网络及VPC网络访问的Endpoint。
数据库	需要访问的MaxCompute项目的名称加`?ssl=true`，如上图中的`lightning?ssl=true`。
Schema	MaxCompute项目名称。
用户名/密码	用户的Access Key ID/Access Key Secret。

SQL Workbench/J

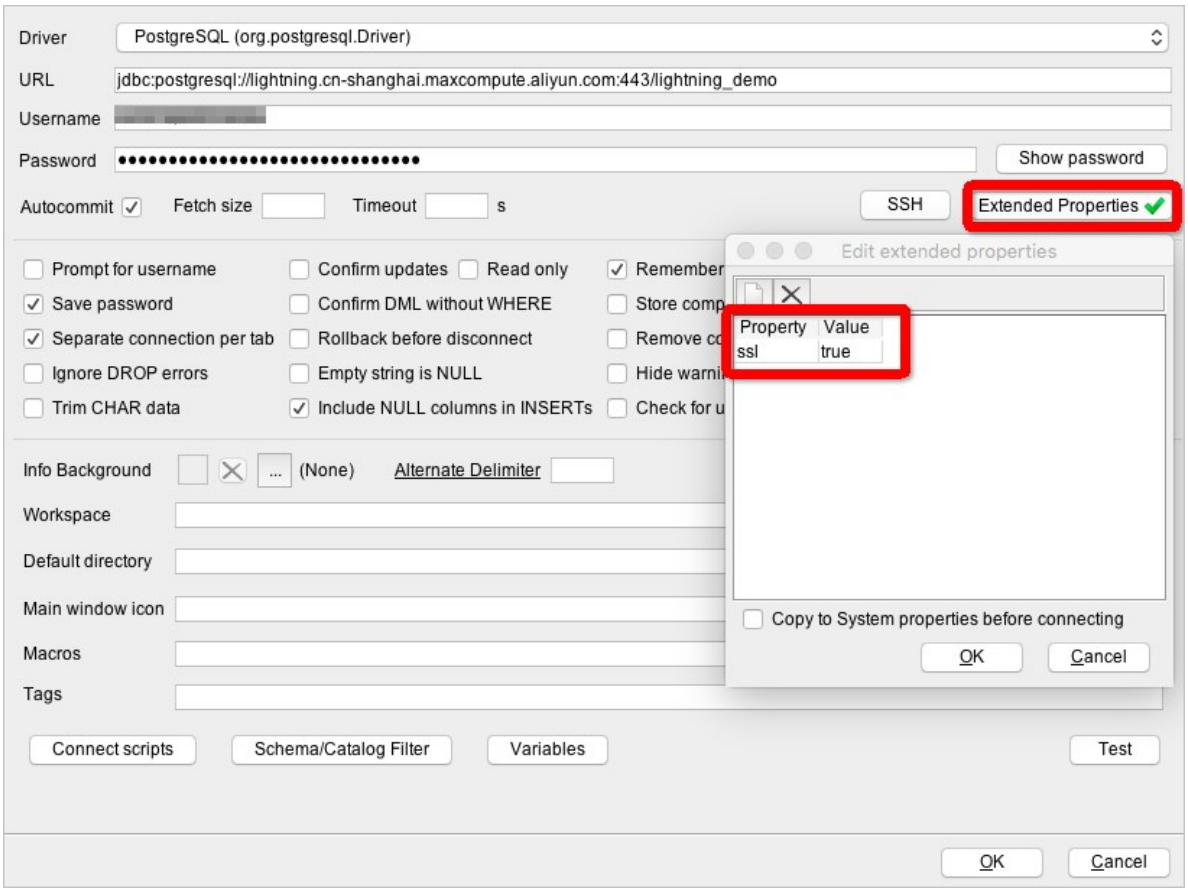
SQL Workbench/J是一款流行的免费、跨平台SQL查询分析工具，使用SQL Workbench/J可以通过PostgreSQL驱动连接MaxCompute Lightning服务。

1. 下载并安装SQL Workbench/J 。
2. 启动SQL Workbench/J，创建数据库连接。

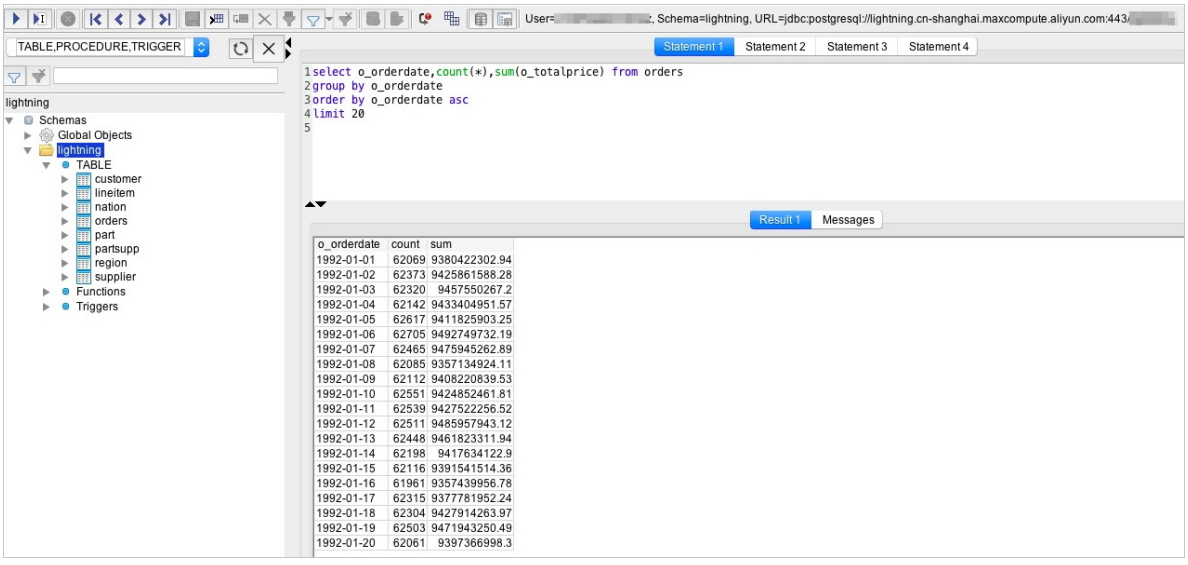
   选择PostgreSQL驱动，连接MaxCompute项目所对应的Lightning URL地址，同时输入访问用户的用户名和密码，即Access Key ID和Access Key Secret。



   您也可通过扩展属性（Extended Properities）设置ssl取值为true。



3. 连接后，在Workbench工作区查看MaxCompute项目的表数据、查询分析。



psql工具连接

psql是PostgreSQL的一个命令行交互式客户端工具，在本机安装PostgreSQL数据库将默认安装psql客户端。

通过psql在命令行下可以连接MaxCompute Lightning，语法与连接PostgreSQL数据库一致。

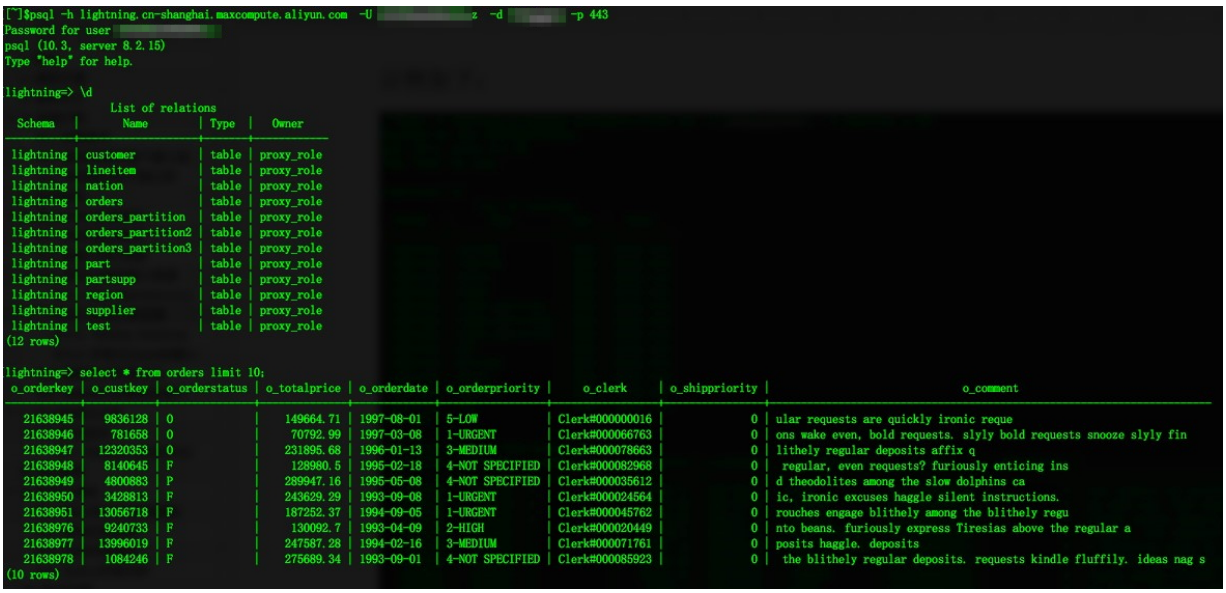
```
psql -h <endpoint> -U <userid> -d <databasename> -p <port>
```

参数说明：

- <endpoint>：MaxCompute Lightning的Endpoint，详情请参见访问域名。
- <userid>：访问用户Access Key ID。
- <databasename>：Maxcompute项目名。
- <port>：443

执行后，在psql密码提示符处，输入<userid>用户的密码，即访问用户的Access Key Secret。

示例如下：



说明：

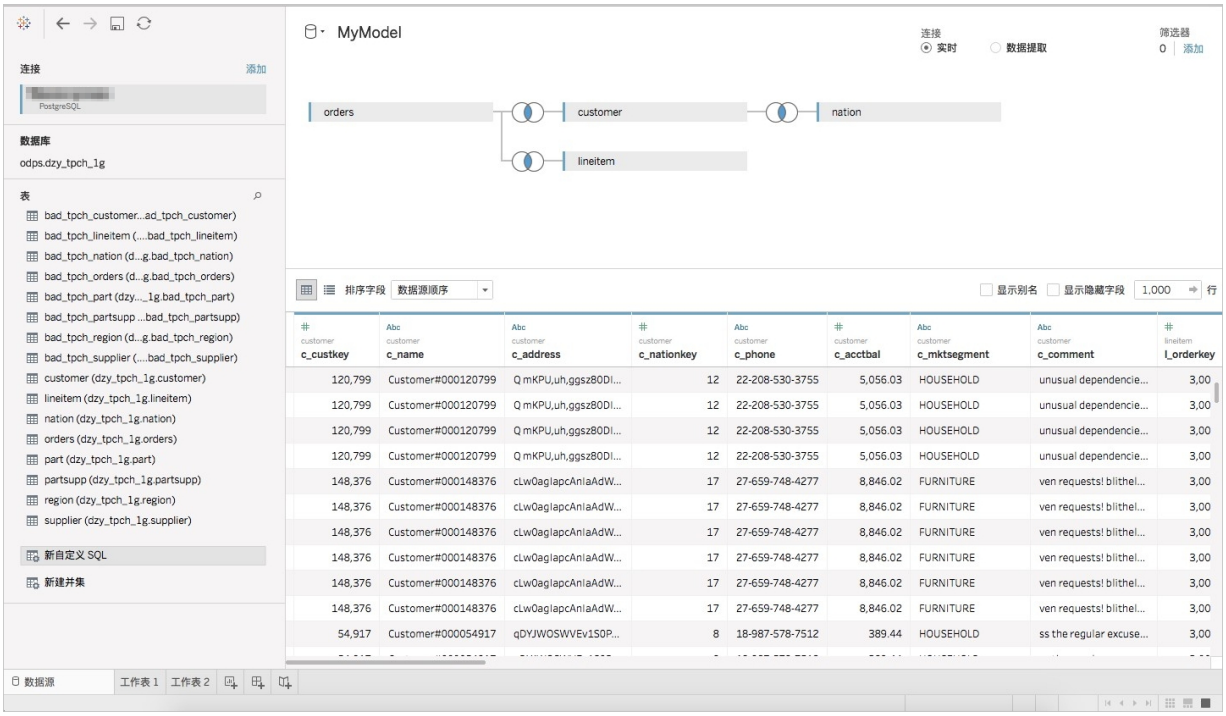
psql默认优先通过ssl方式连接。

Tableau Desktop

使用BI工具，选择PostgreSQL数据源，配置连接。

配置连接时，需勾选需要**SSL**。

登录后，通过Tableau创建工作表进行可视化分析。



说明：

为了获得更好的性能和体验，建议您使用Tableau支持的TDC文件方式，对Lightning数据源进行连接定制优化。具体操作如下：

1. 将如下xml内容保存为postgresql.tdc文件。

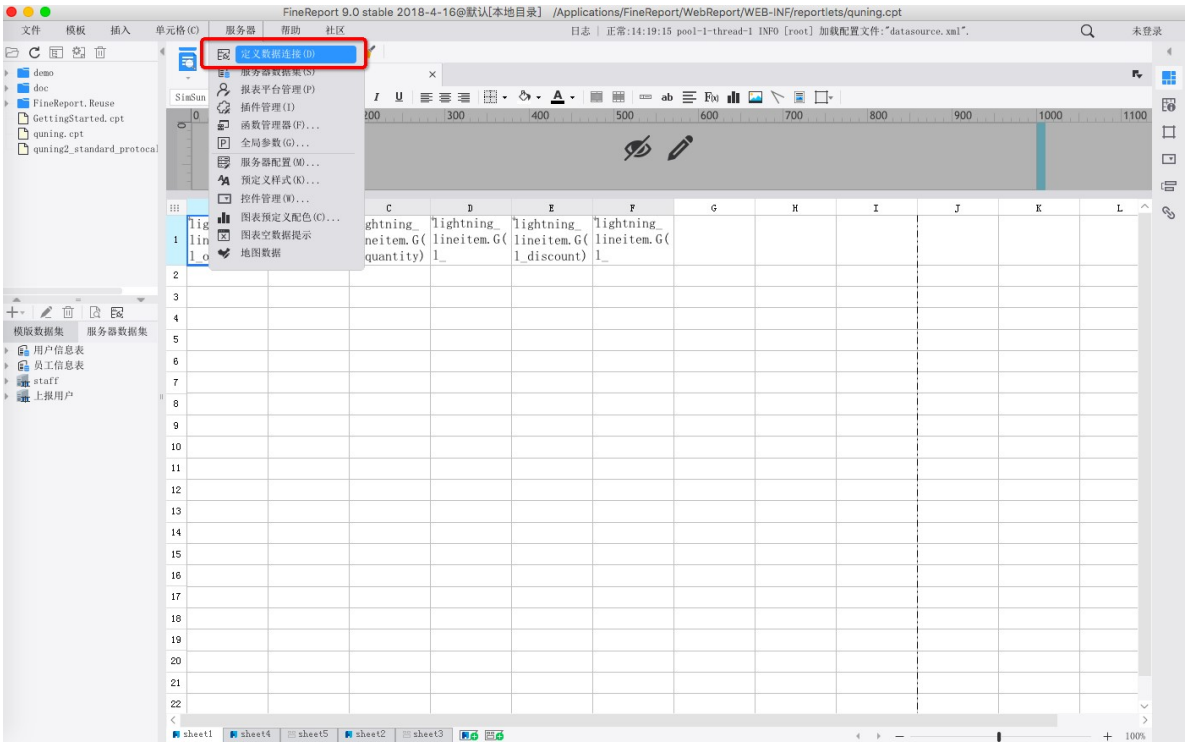
```
<?xml version='1.0' encoding='utf-8' ?>
<connection-customization class='postgres' enabled='true' version='
8.10'>
<vendor name='postgres'/>
<driver name='postgres'/>
```

```
<customizations>
<customization name='CAP_CREATE_TEMP_TABLES' value='no' />
<customization name='CAP_STORED_PROCEDURE_TEMP_TABLE_FROM_BUFFER'
value='no' />
<customization name='CAP_CONNECT_STORED_PROCEDURE' value='no' />
<customization name='CAP_SELECT_INTO' value='no' />
<customization name='CAP_SELECT_TOP_INTO' value='no' />
<customization name='CAP_ISOLATION_LEVEL_SERIALIZABLE' value='yes
' />
<customization name='CAP_SUPPRESS_DISCOVERY_QUERIES' value='yes' />
<customization name='CAP_SKIP_CONNECT_VALIDATION' value='yes' />
<customization name='CAP_ODBC_TRANSACTIONS_SUPPRESS_EXPLICIT_COMMIT
' value='yes' />
<customization name='CAP_ODBC_TRANSACTIONS_SUPPRESS_AUTO_COMMIT'
value='yes' />
<customization name='CAP_ODBC_REBIND_SKIP_UNBIND' value='yes' />
<customization name='CAP_FAST_METADATA' value='no' />
<customization name='CAP_ODBC_METADATA_SUPPRESS_SELECT_STAR' value
='yes' />
<customization name='CAP_ODBC_METADATA_SUPPRESS_EXECUTED_QUERY'
value='yes' />
<customization name='CAP_ODBC_UNBIND_AUTO' value='yes' />
<customization name='SQL_TXN_CAPABLE' value='0' />
<customization name='CAP_ODBC_CURSOR_FORWARD_ONLY' value='yes' />
<customization name='CAP_ODBC_TRANSACTIONS_COMMIT_INVALIDATES
_PREPARED_QUERY' value='yes' />
</customizations>
</connection-customization>
```

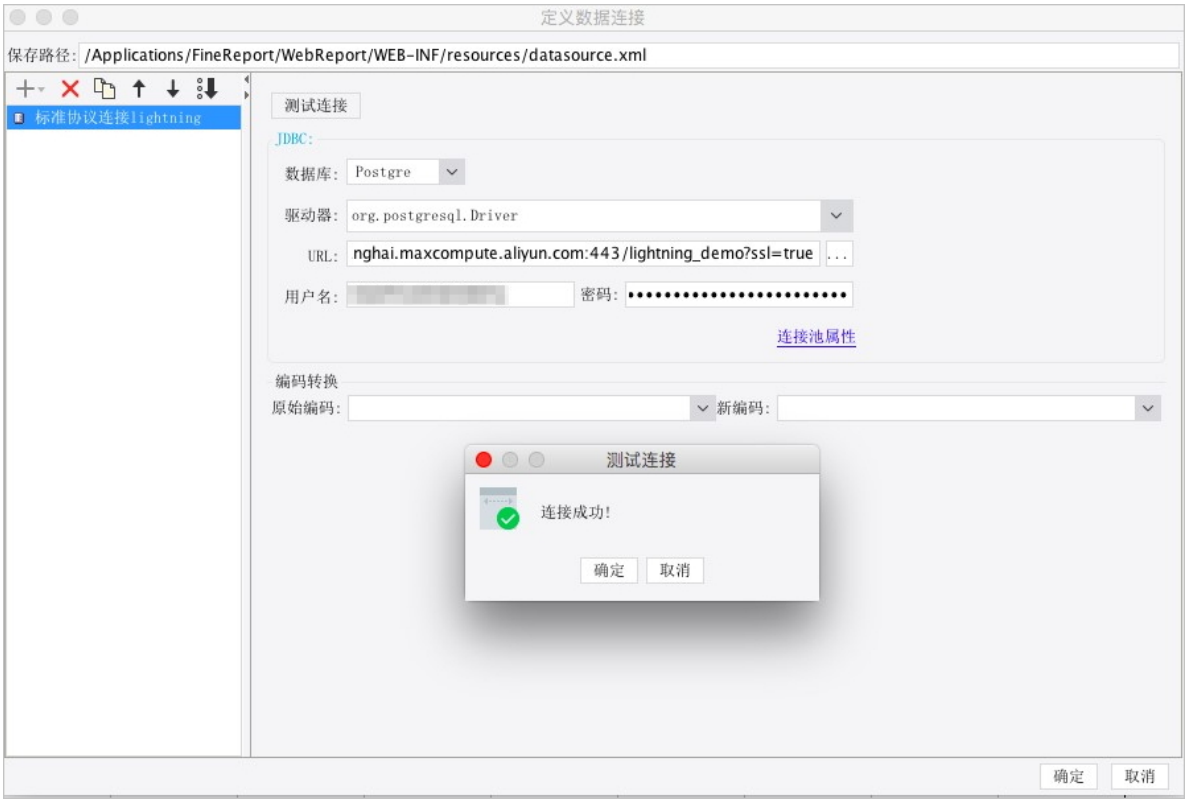
2. 将文件保存到*\My Documents\My Tableau Repository\Datasources*目录下。如果是Tableau Server，Windows下请保存在*C:\ProgramData\Tableau\Tableau Server\data\tabsvc\vizqlserver\Datasources*，Linux下请保存在*/var/opt/tableau/tableau_server/data/tabsvc/vizqlserver/Datasources/*。
3. 重新打开Tableau，使用PostgreSQL数据源连接MaxCompute Lightning服务。关于tdc文件定制数据源的更多内容，请参见*Tableau*官方帮助文档。

帆软Report

1. 打开帆软Report，选择**服务器 > 定义数据库连接**。



2. 添加JDBC连接。



参数说明如下：

参数	说明
数据库	Postgre
驱动器	帆软Report自带的org.postgresql.Driver
URL	`jdbc:postgresql://<MaxCompute Lightning Endpoint>:443/<Project_Name>?ssl=true&prepareThreshold=0` 例如：`jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/lightning_demo?ssl=true&prepareThreshold=0`
用户名/密码	访问用户的Access Key ID和Access Key Secret

2.7 SQL参考

查询语法

MaxCompute Lightning查询引擎基于PostgreSQL 8.2，当前仅支持对已有MaxCompute表进行SELECT查询，相关语法参见*PostgreSQL官方文档*。

函数

MaxCompute Lightning查询引擎基于PostgreSQL 8.2提供内建函数，请参见 *PostgreSQL官方文档*。

在PostgreSQL官方函数基础上，MaxCompute Lightning补充了以下内建函数。

- **MAX_PT**

  命令格式

  `max_pt(table_full_name)`

  命令说明

  对于分区的表，此函数返回该分区表的一级分区的最大值，按字母排序，且该分区下有对应的数据文件。

  参数说明

  table_full_name：String类型，用于指定表名（必须携带project名称，例如prj.src），您必须对此表有读权限。

  返回值

  返回最大的一级分区的值。

示例

假设tbl为分区表，对应分区如下，且都包含数据文件：

```
pt ='20120901'
pt ='20120902'
```

则以下语句中分区max_pt返回值为'20120902'，MaxCompute SQL语句读出pt='20120902'分区下的数据。

`select * from tbl where pt=max_pt('myproject.tbl');`

2.8 查看作业

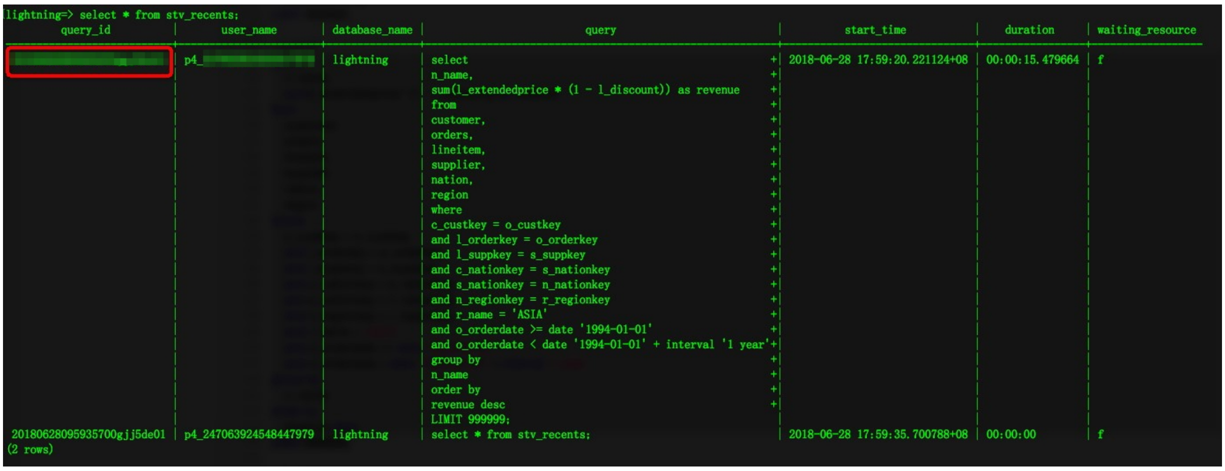
查看运行中的查询

Lightning为用户提供了一个系统视图表stv_recents，通过查询该表能够获取当前用户正在执行中的所有查询作业，查看这些作业的ID、用户、SQL语句、发起开始时间、运行时间、是否等待资源（t为等待资源未执行，f为获取资源并执行中）等信息。

执行查询命令：

```
select * from stv_recents;
```

查询结果如下所示：



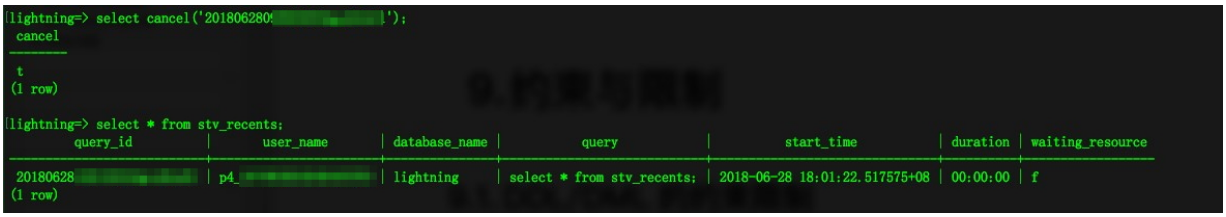
```
lightning=> select * from stv_recents;
        query_id         |      user_name       | database_name |                    query                    |          start_time           |    duration     | waiting_resource
-------------------------+----------------------+---------------+---------------------------------------------+-------------------------------+-----------------+------------------
 [redacted]              | p4_[redacted]        | lightning     | select                                     +| 2018-06-28 17:59:20.221124+08 | 00:00:15.479664 | f
                         |                      |               | n_name,                                    +|                               |                 |
                         |                      |               | sum(l_extendedprice * (1 - l_discount)) as revenue +|                        |                 |
                         |                      |               | from                                       +|                               |                 |
                         |                      |               | customer,                                  +|                               |                 |
                         |                      |               | orders,                                    +|                               |                 |
                         |                      |               | lineitem,                                  +|                               |                 |
                         |                      |               | supplier,                                  +|                               |                 |
                         |                      |               | nation,                                    +|                               |                 |
                         |                      |               | region                                     +|                               |                 |
                         |                      |               | where                                      +|                               |                 |
                         |                      |               | c_custkey = o_custkey                      +|                               |                 |
                         |                      |               | and l_orderkey = o_orderkey                +|                               |                 |
                         |                      |               | and l_suppkey = s_suppkey                  +|                               |                 |
                         |                      |               | and c_nationkey = s_nationkey              +|                               |                 |
                         |                      |               | and s_nationkey = n_nationkey              +|                               |                 |
                         |                      |               | and n_regionkey = r_regionkey              +|                               |                 |
                         |                      |               | and r_name = 'ASIA'                        +|                               |                 |
                         |                      |               | and o_orderdate >= date '1994-01-01'       +|                               |                 |
                         |                      |               | and o_orderdate < date '1994-01-01' + interval '1 year'+|                     |                 |
                         |                      |               | group by                                   +|                               |                 |
                         |                      |               | n_name                                     +|                               |                 |
                         |                      |               | order by                                   +|                               |                 |
                         |                      |               | revenue desc                               +|                               |                 |
                         |                      |               | LIMIT 999999;                               |                               |                 |
 20180628095935700gjj5de01 | p4_247063924548447979 | lightning  | select * from stv_recents;                  | 2018-06-28 17:59:35.700788+08 | 00:00:00        | f
(2 rows)
```

取消正在运行的查询

通过查询stv_recents获得运行中查询的信息，如果想要取消某个查询，可以执行下述语句：

```
select cancel('query_id');
```

其中的query_id，是运行中查询的query_id信息，示例如下：



2.9 约束与限制

DDL/DML的约束限制

MaxCompute Lightning目前不支持update、create、delete、insert操作，仅支持对MaxCompute表进行select，后续版本将陆续开放相关能力。

查询约束限制

- 查询分区表时，扫描分区数的最大值为1024。
- 目前不支持创建和使用View。
- 目前不支持的数据类型：map、array、tinyint和timestamp（陆续支持中）。
- 每个查询中对单张表最大的数据扫描量为1TB。
- 提交的查询语句的长度不超过100KB。
- 查询超时时间为1小时。

UDF约束限制

- 当前不支持在Maxcompute Lightning使用MaxCompute创建的UDF。
- 当前不支持在Maxcompute Lightning中创建和使用PostgreSQL UDF（支持使用*PostgreSQL*内建函数）。

Query并发约束

单个MaxCompute项目的MaxCompute Lightning查询并发数限制为20。

2.10 Lightning常见问题

- Q：还没有建表的情况下，可以用MaxCompute Lightning查什么数据？

  A：您需要先通过DataWorks或odpscmd客户端工具，在MaxCompute项目中创建数据表，加载数据。然后通过MaxCompute Lightning连接到该项目，此时便可查看到项目内的表，并对这些表进行查询。
- Q：MaxCompute Lightning是否限制查询的数据量？查询多大规模的数据性能较好？

A：目前每次查询对单表的扫描数据量限制为1TB，数据量越小查询性能越好。

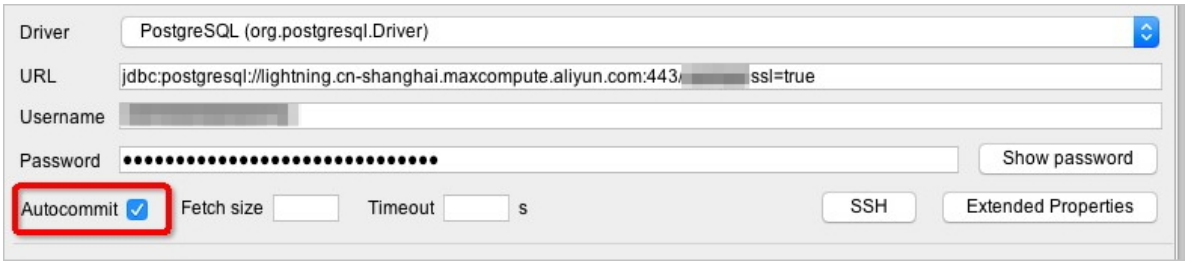
> 说明：
> 建议不要扫描超过100GB的表数据。超过100GB的表数据虽然仍可查询，但查询性能会随数据规模增长逐渐下降。如果需要扫描量超过100GB的查询，建议您根据性能表现考虑使用MaxCompute SQL。

- Q：使用BI工具，通过拖拽方式选择一张分区表进行分析时，提示报错：`ERROR: AXF Exception: specified partitions count in odps table: <project_name.table_name> is: xxx, exceeds the limitation of xxx, please add stricter partition filter`。

  A：MaxCompute Lightning为保障查询性能，对分区表的分区数量进行了限制，一次查询所扫描的单表最大分区数量不能超过1024。由于部分BI工具使用拖拽方式选择表直接进行分析，不能BI前端指定分区条件，导致请求扫描的分区数超限制、触发了Lightning限制而报错提示。建议先对查询的数据表进行加工处理，处理为非分区表或分区小于1024的表再进行分析。
- Q：连接时提示创建数据连接失败: `ERROR: SSL required`。

  A：MaxCompute Lightning要求SSL连接服务，需要客户端指定以SSL方式连接。如果使用客户端工具，可以选择SSL连接选项。如果没有相关选项，可以在JDBC URL连接串中增加SSL参数，需要替换为您项目所在region的endpoint、连接的项目名称，例如`jdbc:postgresql://lightning.cn-shanghai.maxcompute.aliyun.com:443/myproject?ssl=true`。
- Q：使用Workbench/J客户端查询时提示`Error:current transaction is aborted, commands ignored until end of transaction block`。

  A：使用的客户端请勾选**Autocommit**选项。



3 PyODPS

3.1 安装指南

如果能访问外网，推荐使用pip安装，pip安装可以参考 地址 ， 推荐使用 阿里云镜像 加快下载速度。

接着确保 setuptools 和 requests 的版本，对于非 Windows 用户可以安装 Cython 加速 Tunnel 上传下载：

```
pip install setuptools>=3.0
pip install requests>=2.4.0
pip install greenlet>=0.4.10  # 可选，安装后能加速 Tunnel 上传
pip install cython>=0.19.0  # 可选，不建议 Windows 用户安装
```

安装有 合适版本 Visual C++ 和 Cython 的 Windows 用户也可使用 Tunnel 加速功能。

接着就可以安装PyODPS：

```
pip install pyodps
```

检查安装完成：

```
python -c "from odps import ODPS"
```

如果使用的python不是系统默认的python版本，安装完pip则可以：

```
/home/tops/bin/python2.7 -m pip install setuptools>=3.0
```

其余类似。

3.2 工具平台使用指南

3.2.1 工具平台使用指南概述

PyODPS 可在 DataWorks 等数据开发平台中作为节点调用。这些平台提供了 PyODPS 运行环境，不需要手动创建 ODPS 入口对象，免除了手动配置的麻烦，而且还提供了调度执行的能力。对于想从平台迁移到自行部署 PyODPS 环境的用户，下面也提供了迁移注意事项。

- *DataWorks* 用户使用指南
- 从平台到自行部署

3.2.2 从平台到自行部署

如果你需要在本地调试 PyODPS，或者平台中没有提供你所需的包，或者平台的资源限制不能满足你的要求，此时你可能需要 从平台迁移到自己部署的 PyODPS 环境。

安装 PyODPS 的步骤可以参考 安装指南。你需要手动创建先前平台为你创建的 ODPS 入口对象。可以在先前的平台 使用下列语句生成创建 ODPS 入口对象所需要的语句模板，然后手动修改为可用的代码：

```
print("\nfrom odps import ODPS\no = ODPS(%r, '<access-key>', %r, '<
endpoint>')\n" % (o.account.access_id, o.project))
```

其中，<access-key> 和 <endpoint> 需要手动替换成可用的值。access-key 可在 DataWorks 中点击右上角图标 -> 用户信息， 再点击“点击查看”获得。Endpoint 可在 *MaxCompute开通Region和服务连接对照表* 获得， 或者联系项目管理员。

修改完成后，将上述代码放置到所有代码的最顶端即可。

3.2.3 DataWorks 用户使用指南

新建工作流节点

在工作流节点中会包含PYODPS节点。新建即可。

ODPS入口

DataWorks 的 PyODPS 节点中，将会包含一个全局的变量 `odps` 或者 `o` ，即 ODPS 入口。用户不需要手动定义 ODPS 入口。

```
print(o.exist_table('pyodps_iris'))
```

执行SQL

可以参考执行*SQL*文档 。

说明：

Dataworks 上默认没有打开 instance tunnel，即 instance.open_reader 默认走 result 接口（最多一万条）。 打开 instance tunnel，通过 reader.count 能取到记录数，如果要迭代获取全部数据，则需要关闭 limit 限制。

要想全局打开，则：

```
options.tunnel.use_instance_tunnel = True
options.tunnel.limit_instance_tunnel = False  # 关闭 limit 读取全部数据

with instance.open_reader() as reader:
    # 能通过 instance tunnel 读取全部数据
```

或者通过在 open_reader 上添加 `tunnel=True`，来仅对这次 open_reader 打开 instance tunnel；添加 `limit=False`，来关闭 limit 限制从而能下载全部数据。

```
with instance.open_reader(tunnel=True, limit=False) as reader:
    # 这次 open_reader 会走 instance tunnel 接口，且能读取全部数据
```

DataFrame

- 执行

  在 DataWorks 的环境里，*DataFrame*概述 的执行需要显式调用 立即执行的方法#如*execute#head*等# 。

  ```
  from odps.df import DataFrame

  iris = DataFrame(o.get_table('pyodps_iris'))
  for record in iris[iris.sepal_width < 3].execute():  # 调用立即执行的方法
  ```

```
    # 处理每条record
```

如果用户想在print的时候调用立即执行，需要打开 `options.interactive` 。

```
from odps import options
from odps.df import DataFrame

options.interactive = True  # 在开始打开开关

iris = DataFrame(o.get_table('pyodps_iris'))
print(iris.sepal_width.sum())  # 这里print的时候会立即执行
```

- 打印详细信息

  通过设置 `options.verbose` 选项。在 DataWorks 上，默认已经处于打开状态，运行过程会打印 logview 等详细过程。

获取调度参数

与 DataWorks 中的 SQL 节点不同，为了避免侵入代码，PyODPS 节点 不会 在代码中替换 ${param_name} 这样的字符串，而是在执行代码前，在全局变量中增加一个名为 `args` 的 dict，调度参数可以在此获取。例如， 在节点基本属性 -> 参数中设置 `ds=${yyyymmdd}` ，则可以通过下面的方式在代码中获取此参数：

```
print('ds=' + args['ds'])
```

```
ds=20161116
```

特别地，如果要获取名为 `ds=${yyyymmdd}`的分区，则可以使用

```
o.get_table('table_name').get_partition('ds=' + args['ds'])
```

受限功能

由于缺少 matplotlib 等包，所以如下功能可能会受限。

- DataFrame的plot函数

DataFrame 自定义函数需要提交到 MaxCompute 执行。由于 Python 沙箱的原因，第三方库只支持所有的纯 Python 库以及 Numpy， 因此不能直接使用 Pandas，可参考:ref:第三方库支持 <third_party_library> 上传和使用所需的库。DataWorks 中执行的非自定义函数代码可以使用平台预装的 Numpy 和 Pandas。其他带有二进制代码的三方包不被支持。

由于兼容性的原因，在 DataWorks 中，options.tunnel.use_instance_tunnel 默认设置为 False。如果需要全局开启 Instance Tunnel， 需要手动将该值设置为 True。

由于实现的原因，Python 的 atexit 包不被支持，请使用 try - finally 结构实现相关功能。

使用限制

在 DataWorks 上使用 PyODPS，为了防止对 DataWorks 的 gateway 造成压力，对内存和 CPU 都有限制。这个限制由 DataWorks 统一管理。

如果看到 **Got killed**，即内存使用超限，进程被 kill。因此，尽量避免本地的数据操作。

通过 PyODPS 起的 SQL 和 DataFrame 任务 (除 to_pandas) 不受此限制。

3.3 基本操作

3.3.1 基本操作概述

PyODPS 提供直接针对 ODPS 对象的基本操作接口，可通过符合 Python 习惯的编程方式操作 ODPS。

我们会对这几部分来分别展开说明。

- 项目空间
- 表
- *SQL*
- 任务实例
- 资源
- 函数
- *XFlow*和模型

3.3.2 项目空间

项目空间 是ODPS的基本组织单元， 有点类似于Database的概念。

我们通过 ODPS 入口对象的 `get_project` 来取到某个项目空间。

```
project = o.get_project('my_project')  # 取到某个项目
project = o.get_project()              # 取到默认项目
```

如果不提供参数，则取到默认项目空间。

`exist_project` 方法能检验某个项目空间是否存在。

3.3.3 表

表是ODPS的数据存储单元。

基本操作

我们可以用 ODPS 入口对象的 `list_tables`来列出项目空间下的所有表。

```
for table in o.list_tables():
    # 处理每个表
```

通过调用 `exist_table` 来判断表是否存在。

通过调用 `get_table` 来获取表。

```
>>> t = o.get_table('dual')
>>> t.schema
odps.Schema {
  c_int_a                 bigint
  c_int_b                 bigint
  c_double_a              double
  c_double_b              double
  c_string_a              string
  c_string_b              string
  c_bool_a                boolean
  c_bool_b                boolean
  c_datetime_a            datetime
  c_datetime_b            datetime
}
>>> t.lifecycle
-1
>>> print(t.creation_time)
2014-05-15 14:58:43
>>> t.is_virtual_view
False
>>> t.size
1408
>>> t.comment
'Dual Table Comment'
>>> t.schema.columns
[<column c_int_a, type bigint>,
 <column c_int_b, type bigint>,
 <column c_double_a, type double>,
 <column c_double_b, type double>,
 <column c_string_a, type string>,
 <column c_string_b, type string>,
 <column c_bool_a, type boolean>,
 <column c_bool_b, type boolean>,
 <column c_datetime_a, type datetime>,
 <column c_datetime_b, type datetime>]
>>> t.schema['c_int_a']
<column c_int_a, type bigint>
>>> t.schema['c_int_a'].comment
```

```
'Comment of column c_int_a'
```

通过提供 `project`参数，来跨project获取表。

```
>>> t = o.get_table('dual', project='other_project')
```

创建表的Schema

有两种方法来初始化。第一种方式通过表的列、以及可选的分区来初始化。

```
>>> from odps.models import Schema, Column, Partition
>>> columns = [Column(name='num', type='bigint', comment='the column
'),
>>>            Column(name='num2', type='double', comment='the column2
')]
>>> partitions = [Partition(name='pt', type='string', comment='the
partition')]
>>> schema = Schema(columns=columns, partitions=partitions)
>>> schema.columns
[<column num, type bigint>,
 <column num2, type double>,
 <partition pt, type string>]
>>> schema.partitions
[<partition pt, type string>]
>>> schema.names  # 获取非分区字段的字段名
['num', 'num2']
>>> schema.types  # 获取非分区字段的字段类型
[bigint, double]
```

第二种方法是使用 `Schema.from_lists`，这种方法更容易调用，但显然无法直接设置列和分区的注释了。

```
>>> schema = Schema.from_lists(['num', 'num2'], ['bigint', 'double'],
 ['pt'], ['string'])
>>> schema.columns
[<column num, type bigint>,
 <column num2, type double>,
 <partition pt, type string>]
```

创建表

可以使用表 schema 来创建表，方法如下：

```
>>> table = o.create_table('my_new_table', schema)
>>> table = o.create_table('my_new_table', schema, if_not_exists=True)
  # 只有不存在表时才创建
>>> table = o.create_table('my_new_table', schema, lifecycle=7)  # 设置
生命周期
```

更简单的方式是采用“字段名 字段类型”字符串来创建表，方法如下：

```
>>> table = o.create_table('my_new_table', 'num bigint, num2 double',
if_not_exists=True)
>>> # 创建分区表可传入 (表字段列表, 分区字段列表)
```

```
>>> table = o.create_table('my_new_table', ('num bigint, num2 double',
 'pt string'), if_not_exists=True)
```

在未经设置的情况下，创建表时，只允许使用 bigint、double、decimal、string、datetime、boolean、map 和 array 类型。 如果你使用的是位于公共云上的服务，或者支持 tinyint、struct 等新类型，可以设置

`options.sql.use_odps2_extension = True`

打开这些类型的支持，示例如下：

```
>>> from odps import options
>>> options.sql.use_odps2_extension = True
>>> table = o.create_table('my_new_table', 'cat smallint, content
struct<title:varchar(100), body string>')
```

同步表更新

有时候，一个表可能被别的程序做了更新，比如schema有了变化。此时可以调用 `reload` 方法来更新。

```
>>> table.reload()
```

行记录Record

Record表示表的一行记录，我们在 Table 对象上调用 new_record 就可以创建一个新的 Record。

```
>>> t = o.get_table('mytable')
>>> r = t.new_record(['val0', 'val1'])  # 值的个数必须等于表schema的字段数
>>> r2 = t.new_record()  #  也可以不传入值
>>> r2[0] = 'val0' # 可以通过偏移设置值
>>> r2['field1'] = 'val1'  # 也可以通过字段名设置值
>>> r2.field1 = 'val1'  # 通过属性设置值
>>>
>>> print(record[0])  # 取第0个位置的值
>>> print(record['c_double_a'])  # 通过字段取值
>>> print(record.c_double_a)  # 通过属性取值
>>> print(record[0: 3])  # 切片操作
>>> print(record[0, 2, 3])  # 取多个位置的值
>>> print(record['c_int_a', 'c_double_a'])  # 通过多个字段取值
```

获取表数据

有若干种方法能够获取表数据。首先，如果只是查看每个表的开始的小于1万条数据，则可以使用head方法。

```
>>> t = o.get_table('dual')
>>> for record in t.head(3):
```

```
>>>     # 处理每个Record对象
```

其次，在table上可以执行 `open_reader` 操作来打一个reader来读取数据。

使用 with 表达式的写法：

```
>>> with t.open_reader(partition='pt=test') as reader:
>>>     count = reader.count
>>>     for record in reader[5:10]  # 可以执行多次，直到将count数量的
record读完，这里可以改造成并行操作
>>>         # 处理一条记录
```

不使用 with 表达式的写法：

```
>>> reader = t.open_reader(partition='pt=test')
>>> count = reader.count
>>> for record in reader[5:10]  # 可以执行多次，直到将count数量的record读
完，这里可以改造成并行操作
>>>     # 处理一条记录
```

更简单的调用方法是使用 ODPS 对象的 `read_table`方法，例如

```
>>> for record in o.read_table('test_table', partition='pt=test'):
>>>     # 处理一条记录
```

向表写数据

类似于 `open_reader`，table对象同样能执行 `open_writer` 来打开writer，并写数据。

使用 with 表达式的写法：

```
>>> with t.open_writer(partition='pt=test') as writer:
>>>     records = [[111, 'aaa', True],                # 这里可以是list
>>>                [222, 'bbb', False],
>>>                [333, 'ccc', True],
>>>                [444, '中文', False]]
>>>     writer.write(records)  # 这里records可以是可迭代对象
>>>
>>>     records = [t.new_record([111, 'aaa', True]),   # 也可以是Record
对象
>>>                t.new_record([222, 'bbb', False]),
>>>                t.new_record([333, 'ccc', True]),
>>>                t.new_record([444, '中文', False])]
>>>     writer.write(records)
>>>
>>> with t.open_writer(partition='pt=test', blocks=[0, 1]) as writer:
 # 这里同是打开两个block
>>>     writer.write(0, gen_records(block=0))
>>>     writer.write(1, gen_records(block=1))  # 这里两个写操作可以多线程
并行，各个block间是独立的
```

```
>>>
```

不使用 with 表达式的写法：

```
>>> writer = t.open_writer(partition='pt=test', blocks=[0, 1])
>>> writer.write(0, gen_records(block=0))
>>> writer.write(1, gen_records(block=1))
>>> writer.close()  # 不要忘记关闭 writer，否则数据可能写入不完全
```

如果分区不存在，可以使用`create_partition`参数指定创建分区，如

```
>>> with t.open_writer(partition='pt=test', create_partition=True) as
writer:
>>>     records = [[111, 'aaa', True],                # 这里可以是list
>>>                [222, 'bbb', False],
>>>                [333, 'ccc', True],
>>>                [444, '中文', False]]
>>>     writer.write(records)  # 这里records可以是可迭代对象
```

说明：

每次调用 write_table，MaxCompute 都会在服务端生成一个文件。这一操作需要较大的时间开销，同时过多的文件会降低后续的查询效率。因此，我们建议在使用 write_table 方法时，一次性写入多组数据，或者传入一个 generator 对象。

删除表

```
>>> o.delete_table('my_table_name', if_exists=True)  #  只有表存在时删除
>>> t.drop()  # Table对象存在的时候可以直接执行drop函数
```

创建**DataFrame**

PyODPS提供了 *DataFrame*框架 ，支持更方便地方式来查询和操作ODPS数据。 使用`to_df`方法，即可转化为 DataFrame 对象。

```
>>> table = o.get_table('my_table_name')
>>> df = table.to_df()
```

表分区

- 基本操作

  判断是否为分区表：

  ```
  >>> if table.schema.partitions:
  ```

```
>>>     print('Table %s is partitioned.' % table.name)
```

遍历表全部分区：

```
>>> for partition in table.partitions:
>>>     print(partition.name)
>>> for partition in table.iterate_partitions(spec='pt=test'):
>>>     # 遍历二级分区
```

判断分区是否存在：

```
>>> table.exist_partition('pt=test,sub=2015')
```

获取分区：

```
>>> partition = table.get_partition('pt=test')
>>> print(partition.creation_time)
2015-11-18 22:22:27
>>> partition.size
0
```

- 创建分区

```
>>> t.create_partition('pt=test', if_not_exists=True)  # 不存在的时候才创建
```

- 删除分区

```
>>> t.delete_partition('pt=test', if_exists=True)  # 存在的时候才删除
>>> partition.drop()  # Partition对象存在的时候直接drop
```

数据上传下载通道

ODPS Tunnel是ODPS的数据通道，用户可以通过Tunnel向ODPS中上传或者下载数据。

说明：

不推荐直接使用tunnel接口（难用且复杂），推荐直接使用表的写和读接口。 如果安装了

Cython，在安装pyodps时会编译C代码，加速Tunnel的上传和下载。

- 上传

```
from odps.tunnel import TableTunnel

table = o.get_table('my_table')

tunnel = TableTunnel(odps)
upload_session = tunnel.create_upload_session(table.name, partition_
spec='pt=test')

with upload_session.open_record_writer(0) as writer:
    record = table.new_record()
```

```
    record[0] = 'test1'
    record[1] = 'id1'
    writer.write(record)

    record = table.new_record(['test2', 'id2'])
    writer.write(record)

upload_session.commit([0])
```

- 下载

```
from odps.tunnel import TableTunnel

tunnel = TableTunnel(odps)
download_session = tunnel.create_download_session('my_table',
partition_spec='pt=test')

with download_session.open_record_reader(0, download_session.count)
as reader:
    for record in reader:
        # 处理每条记录
```

3.3.4 SQL

PyODPS支持ODPS SQL的查询，并可以读取执行的结果。

`execute_sql` 或者 `run_sql` 方法的返回值是 运行实例 。

说明：

并非所有在 ODPS Console 中可以执行的命令都是 ODPS 可以接受的 SQL 语句。 在调用非 DDL / DML 语句时，请使用其他方法，例如 GRANT / REVOKE 等语句请使用`run_security_query`方法，PAI 命令请使用 `run_xflow` 或 `execute_xflow`方法。

执行SQL

```
>>> o.execute_sql('select * from dual')  #  同步的方式执行，会阻塞直到SQL执行完成
>>>
>>> instance = o.run_sql('select * from dual')  # 异步的方式执行
>>> print(instance.get_logview_address())  # 获取logview地址
```

```
>>> instance.wait_for_success()  # 阻塞直到完成
```

设置运行参数

有时，我们在运行时，需要设置运行时参数，我们可以通过设置 `hints`参数，参数类型是dict。

```
>>> o.execute_sql('select * from pyodps_iris', hints={'odps.sql.mapper
.split.size': 16})
```

我们可以对于全局配置设置sql.settings后，每次运行时则都会添加相关的运行时参数。

```
>>> from odps import options
>>> options.sql.settings = {'odps.sql.mapper.split.size': 16}
>>> o.execute_sql('select * from pyodps_iris')  # 会根据全局配置添加hints
```

读取**SQL**执行结果

运行 SQL 的 instance 能够直接执行 `open_reader`的操作，一种情况是SQL返回了结构化的数据。

```
>>> with o.execute_sql('select * from dual').open_reader() as reader:
>>>     for record in reader:
>>>         # 处理每一个record
```

另一种情况是 SQL 可能执行的比如 `desc`，这时通过 `reader.raw`属性取到原始的SQL执行结果。

```
>>> with o.execute_sql('desc dual').open_reader() as reader:
>>>     print(reader.raw)
```

如果 `options.tunnel.use_instance_tunnel == True`，在调用 open_reader 时，PyODPS 会默认调用 Instance Tunnel， 否则会调用旧的 Result 接口。如果你使用了版本较低的 MaxCompute 服务，或者调用 Instance Tunnel 出现了问题，PyODPS 会给出警告并自动降级到旧的 Result 接口，可根据警告信息判断导致降级的原因。如果 Instance Tunnel 的结果不合预期，请将该选项设为 False。在调用 open_reader 时，也可以使用 `tunnel`参数来指定使用何种结果接口，例如

```
>>> # 使用 Instance Tunnel
>>> with o.execute_sql('select * from dual').open_reader(tunnel=True)
as reader:
>>>     for record in reader:
>>>         # 处理每一个record
>>> # 使用 Results 接口
>>> with o.execute_sql('select * from dual').open_reader(tunnel=False
) as reader:
>>>     for record in reader:
```

```
>>>         # 处理每一个record
```

PyODPS 默认不限制能够从 Instance 读取的数据规模。对于受保护的 Project，通过 Tunnel 下载数据受限。此时， 如果 options.tunnel.limit_instance_tunnel 未设置，会自动打开数据量限制。此时，可下载的数据条数受到 Project 配置限制， 通常该限制为 10000 条。如果你想要手动限制下载数据的规模，可以为 open_reader 方法增加 limit 选项， 或者设置 options.tunnel.limit_instance_tunnel = True 。

如果你所使用的 MaxCompute 只能支持旧 Result 接口，同时你需要读取所有数据，可将 SQL 结果写入另一张表后用读表接口读取（可能受到 Project 安全设置的限制）。

设置alias

有时在运行时，比如某个UDF引用的资源是动态变化的，我们可以alias旧的资源名到新的资源，这样免去了重新删除并重新创建UDF的麻烦。

```
from odps.models import Schema

myfunc = '''\
from odps.udf import annotate
from odps.distcache import get_cache_file

@annotate('bigint->bigint')
class Example(object):
    def __init__(self):
        self.n = int(get_cache_file('test_alias_res1').read())

    def evaluate(self, arg):
        return arg + self.n
'''
res1 = o.create_resource('test_alias_res1', 'file', file_obj='1')
o.create_resource('test_alias.py', 'py', file_obj=myfunc)
o.create_function('test_alias_func',
                  class_type='test_alias.Example',
                  resources=['test_alias.py', 'test_alias_res1'])

table = o.create_table(
    'test_table',
    schema=Schema.from_lists(['size'], ['bigint']),
    if_not_exists=True
)

data = [[1, ], ]
# 写入一行数据，只包含一个值1
o.write_table(table, 0, [table.new_record(it) for it in data])

with o.execute_sql(
    'select test_alias_func(size) from test_table').open_reader() as
reader:
```

```
    print(reader[0][0])
```

```
2
```

```
res2 = o.create_resource('test_alias_res2', 'file', file_obj='2')
# 把内容为1的资源alias成内容为2的资源，我们不需要修改UDF或资源
with o.execute_sql(
    'select test_alias_func(size) from test_table',
    aliases={'test_alias_res1': 'test_alias_res2'}).open_reader() as
reader:
    print(reader[0][0])
```

```
3
```

在交互式环境执行 **SQL**

在 ipython 和 jupyter 里支持使用 *SQL* 插件的方式运行 *SQL* ，且支持 参数化查询。

设置 **biz_id**

在少数情形下，可能在提交 SQL 时，需要同时提交 biz_id，否则执行会报错。此时，你可以设置全局 options 里的 biz_id。

```
from odps import options

options.biz_id = 'my_biz_id'
o.execute_sql('select * from pyodps_iris')
```

3.3.5 任务实例

Task如SQLTask是ODPS的基本计算单元，当一个Task在执行时会被实例化， 以 *ODPS实例* 的形式存在。

基本操作

可以调用 `list_instances`来获取项目空间下的所有instance，`exist_instance`能判断是否存在某instance， `get_instance`能获取实例。

```
>>> for instance in o.list_instances():
>>>     print(instance.id)
>>> o.exist_instance('my_instance_id')
```

停止一个instance可以在odps入口使用 `stop_instance`，或者对instance对象调用 `stop`方法。

获取 **LogView** 地址

对于 SQL 等任务，通过调用 `get_logview_address`方法即可。

```
>>> # 从已有的 instance 对象
>>> instance = o.run_sql('desc pyodps_iris')
```

```
>>> print(instance.get_logview_address())
>>> # 从 instance id
>>> instance = o.get_instance('20160426055209459g9k5pvyi2')
>>> print(instance.get_logview_address())
```

对于 XFlow 任务，需要枚举其子任务，再获取子任务的 LogView：

```
>>> instance = o.run_xflow('AppendID', 'algo_public',
                           {'inputTableName': 'input_table', '
outputTableName': 'output_table'})
>>> for sub_inst_name, sub_inst in o.get_xflow_sub_instances(instance
).items():
>>>     print('%s: %s' % (sub_inst_name, sub_inst.get_logview_address
()))
```

任务实例状态

一个instance的状态可以是 `Running`、`Suspended` 或者 `Terminated`，用户可以通过 `status` 属性来获取状态。 `is_terminated` 方法返回当前instance是否已经执行完成，`is_successful` 方法返回当前instance是否正确完成执行， 任务处于运行中或者执行失败都会返回False。

```
>>> instance = o.get_instance('20160426055209459g9k5pvyi2')
>>> instance.status
<Status.TERMINATED: 'Terminated'>
>>> from odps.models import Instance
>>> instance.status == Instance.Status.TERMINATED
True
>>> instance.status.value
'Terminated'
```

调用 `wait_for_completion` 方法会阻塞直到instance执行完成，`wait_for_success`方法同样会阻塞，不同的是， 如果最终任务执行失败，则会抛出相关异常。

子任务操作

一个Instance真正运行时，可能包含一个或者多个子任务，我们称为Task，要注意这个Task不同于ODPS的计算单元。

我们可以通过 `get_task_names` 来获取所有的Task任务，它返回一个所有子任务的名称列表。

```
>>> instance.get_task_names()
['SQLDropTableTask']
```

拿到Task的名称，我们就可以通过 `get_task_result`来获取这个Task的执行结果。

`get_task_results`以字典的形式返回每个Task的执行结果

```
>>> instance = o.execute_sql('select * from pyodps_iris limit 1')
>>> instance.get_task_names()
['AnonymousSQLTask']
>>> instance.get_task_result('AnonymousSQLTask')
```

```
'"sepallength","sepalwidth","petallength","petalwidth","name"\n5.1,3.5
,1.4,0.2,"Iris-setosa"\n'
>>> instance.get_task_results()
OrderedDict([('AnonymousSQLTask',
              '"sepallength","sepalwidth","petallength","petalwidth","
name"\n5.1,3.5,1.4,0.2,"Iris-setosa"\n')])
```

有时候我们需要在任务实例运行时显示所有子任务的运行进程。使用 `get_task_progress`能获得Task当前的运行进度。

```
>>> while not instance.is_terminated():
>>>     for task_name in instance.get_task_names():
>>>         print(instance.id, instance.get_task_progress(task_name).
get_stage_progress_formatted_string())
>>>     time.sleep(10)
20160519101349613gzbzufck2 2016-05-19 18:14:03 M1_Stg1_job0:0/1/1[100
%]
```

3.3.6 资源

资源 在ODPS上常用在UDF和MapReduce中。

列出所有资源还是可以使用 `list_resources`，判断资源是否存在使用 `exist_resource`。 删除资源时，可以调用 `delete_resource`，或者直接对于Resource对象调用 `drop` 方法。

在PyODPS中，主要支持两种资源类型，一种是文件，另一种是表。

文件资源

文件资源包括基础的 `file` 类型、以及 `py`、`jar`、`archive`。

- 创建文件资源

  创建文件资源可以通过给定资源名、文件类型、以及一个file-like的对象（或者是字符串对象）来创建，比如

  ```
  resource = o.create_resource('test_file_resource', 'file', file_obj=
  open('/to/path/file'))  # 使用file-like的对象
  resource = o.create_resource('test_py_resource', 'py', file_obj='
  import this')  # 使用字符串
  ```

- 读取和修改文件资源

  对文件资源调用 `open`方法，或者在odps入口调用 `open_resource`都能打开一个资源， 打开后的对象会是file-like的对象。 类似于Python内置的 `open` 方法，文件资源也支持打开的模式。我们看例子：

  ```
  >>> with resource.open('r') as fp:  # 以读模式打开
  >>>     content = fp.read()  # 读取全部的内容
  >>>     fp.seek(0)  # 回到资源开头
  ```

```
>>>     lines = fp.readlines()  # 读成多行
>>>     fp.write('Hello World')  # 报错，读模式下无法写资源
>>>
>>> with o.open_resource('test_file_resource', mode='r+') as fp:
 # 读写模式打开
>>>     fp.read()
>>>     fp.tell()  # 当前位置
>>>     fp.seek(10)
>>>     fp.truncate()  # 截断后面的内容
>>>     fp.writelines(['Hello\n', 'World\n'])  # 写入多行
>>>     fp.write('Hello World')
>>>     fp.flush()  # 手动调用会将更新提交到ODPS
```

所有支持的打开类型包括：

- `r`，读模式，只能打开不能写
- `w`，写模式，只能写入而不能读文件，注意用写模式打开，文件内容会被先清空
- `a`，追加模式，只能写入内容到文件末尾
- `r+`，读写模式，能任意读写内容
- `w+`，类似于 `r+`，但会先清空文件内容
- `a+`，类似于 `r+`，但写入时只能写入文件末尾

同时，PyODPS中，文件资源支持以二进制模式打开，打开如说一些压缩文件等等就需要以这种模式， 因此 `rb` 就是指以二进制读模式打开文件，`r+b` 是指以二进制读写模式打开。

表资源

- 创建表资源

```
>>> o.create_resource('test_table_resource', 'table', table_name='
my_table', partition='pt=test')
```

- 更新表资源

```
>>> table_resource = o.get_resource('test_table_resource')
>>> table_resource.update(partition='pt=test2', project_name='
my_project2')
```

- 获取表及分区

```
>>> table_resource = o.get_resource('test_table_resource')
>>> table = table_resource.table
>>> print(table.name)
>>> partition = table_resource.partition
>>> print(partition.spec)
```

- 读写内容

```
>>> table_resource = o.get_resource('test_table_resource')
```

```
>>> with table_resource.open_writer() as writer:
>>>     writer.write([0, 'aaaa'])
>>>     writer.write([1, 'bbbbb'])
>>> with table_resource.open_reader() as reader:
>>>     for rec in reader:
>>>         print(rec)
```

3.3.7 函数

函数

ODPS用户可以编写自定义函数 用在ODPS SQL中。

基本操作

可以调用 ODPS 入口对象的 `list_functions` 来获取项目空间下的所有函数，`exist_function`能判断是否存在函数， `get_function` 获取函数对象。

创建函数

可以调用 ODPS 入口对象的 `list_functions`来获取项目空间下的所有函数，`exist_function` 能判断是否存在函数， `get_function`获取函数对象。

创建函数

```
>>> resource = o.get_resource('my_udf.py')
>>> function = o.create_function('test_function', class_type='my_udf.
Test', resources=[resource, ])
```

说明：

注意，公共云由于安全原因，使用 Python UDF 需要申请。

删除函数

```
>>> o.delete_function('test_function')
>>> function.drop()  # Function对象存在时直接调用drop
```

更新函数

只需对函数调用 `update`方法即可。

```
>>> function = o.get_function('test_function')
>>> new_resource = o.get_resource('my_udf2.py')
>>> function.class_type = 'my_udf2.Test'
>>> function.resources = [new_resource, ]
```

```
>>> function.update()  # 更新函数
```

3.3.8 XFlow和模型

XFlow 是 ODPS 对算法包的封装，使用 PyODPS 可以执行 XFlow。

XFlow

对于下面的 PAI 命令：

```
PAI -name AlgoName -project algo_public -Dparam1=param_value1 -Dparam2
=param_value2 ...
```

可以使用如下方法调用：

```
>>> # 异步调用
>>> inst = o.run_xflow('AlgoName', 'algo_public',
                       parameters={'param1': 'param_value1', 'param2':
 'param_value2', ...})
```

或者使用同步调用：

```
>>> # 同步调用
>>> inst = o.execute_xflow('AlgoName', 'algo_public',
                            parameters={'param1': 'param_value1', '
param2': 'param_value2', ...})
```

参数不应包含命令两端的引号（如果有），也不应该包含末尾的分号。

这两个方法都会返回一个 Instance 对象。由于 XFlow 的一个 Instance 包含若干个子 Instance，需要使用下面的方法来获得每个 Instance 的 LogView：

```
>>> for sub_inst_name, sub_inst in o.get_xflow_sub_instances(inst).
items():
>>>     print('%s: %s' % (sub_inst_name, sub_inst.get_logview_address
()))
```

需要注意的是，

`get_xflow_sub_instances`

返回的是 Instance 当前的子 Instance，可能会随时间变化，因而可能需要定时查询。 为简化这一步骤，可以使用

`iter_xflow_sub_instances` 方法

。该方法返回一个迭代器，会阻塞执行直至发现新的子 Instance 或者主 Instance 结束：

```
>>> # 此处建议使用异步调用
>>> inst = o.run_xflow('AlgoName', 'algo_public',
```

```
                          parameters={'param1': 'param_value1', 'param2':
 'param_value2', ...})
>>> for sub_inst_name, sub_inst in o.iter_xflow_sub_instances(inst):
 # 此处将等待
>>>     print('%s: %s' % (sub_inst_name, sub_inst.get_logview_address
()))
```

在调用 run_xflow 或者 execute_xflow 时，也可以指定运行参数，指定的方法与 SQL 类似：

```
>>> parameters = {'param1': 'param_value1', 'param2': 'param_value2
', ...}
>>> o.execute_xflow('AlgoName', 'algo_public', parameters=parameters,
hints={'odps.xxx.yyy': 10})
```

使用 options.ml.xflow_settings 可以配置全局设置：

```
>>> from odps import options
>>> options.ml.xflow_settings = {'odps.xxx.yyy': 10}
>>> parameters = {'param1': 'param_value1', 'param2': 'param_value2
', ...}
>>> o.execute_xflow('AlgoName', 'algo_public', parameters=parameters)
```

PAI 命令的文档可以参考 这份文档 。

XFlow模型

在线模型是 ODPS 提供的模型在线部署能力。

- 在线模型

  用户可以通过 Pipeline 部署自己的模型。详细信息请参考“机器学习平台——在线服务”章节。

  需要注意的是，在线模型的服务使用的是独立的 Endpoint，需要配置 Predict Endpoint。通过

  ```
  >>> o = ODPS('your-access-id', 'your-secret-access-key', 'your-
  default-project',
  >>>           endpoint='your-end-point', predict_endpoint='predict_en
  dpoint')
  ```

  即可在 ODPS 对象上添加相关配置。Predict Endpoint 的地址请参考相关说明或咨询管理员。

- 部署离线模型上线

  PyODPS 提供了离线模型的部署功能。部署方法为：

  ```
  >>> model = o.create_online_model('online_model_name', 'offline_mo
  del_name')
  ```

- 部署自定义 Pipeline 上线

含有自定义 Pipeline 的在线模型可自行构造 ModelPredictor 对象，例子如下：

```
>>> from odps.models.ml import ModelPredictor, ModelProcessor,
BuiltinProcessor, PmmlProcessor, PmmlRunMode
>>> predictor = ModelPredictor(target_name='label')
>>> predictor.pipeline.append(BuiltinProcessor(offline_model_name='
sample_offlinemodel',
>>>                                             offline_model_project
='online_test'))
>>> predictor.pipeline.append(PmmlProcessor(pmml='data_preprocess.
xml',
>>>                                           resources='online_test/
resources/data_preprocess.xml',
>>>                                           run_mode=PmmlRunMode.
Converter))
>>> predictor.pipeline.append(CustomProcessor(class_name='SampleProc
essor',
>>>                                             lib='libsample_
processor.so',
>>>                                             resources='online_test
/resources/sample_processor.tar.gz'))
>>> model = o.create_online_model('online_model_name', predictor)
```

其中，BuiltinProcessor、PmmlProcessor 和 CustomProcessor 分别指 ODPS OfflineModel 形成的 Pipeline 节点、PMML 模型文件形成的 Pipeline 节点和用户自行开发的 Pipeline 节点。

- 在线模型操作

  与其他 ODPS 对象类似，创建后，可列举、获取和删除在线模型：

  ```
  >>> models = o.list_online_models(prefix='prefix')
  >>> model = o.get_online_model('online_model_name')
  >>> o.delete_online_model('online_model_name')
  ```

  可使用模型名和数据进行在线预测，输入数据可以是 Record，也可以是字典或数组和 Schema 的组合：

  ```
  >>> data = [[4, 3, 2, 1], [1, 2, 3, 4]]
  >>> result = o.predict_online_model('online_model_name', data,
  >>>                                 schema=['sepal_length', '
  sepal_width', 'petal_length', 'petal_width'])
  ```

  也可为模型设置 ABTest。参数中的 modelx 可以是在线模型名，也可以是 get_online_model 获得的模型对象本身，而 percentagex 表示 modelx 在 ABTest 中所占的百分比，范围为 0 至 100：

  ```
  >>> result = o.config_online_model_ab_test('online_model_name',
  model1, percentage1, model2, percentage2)
  ```

  修改模型参数可以通过修改 OnlineModel 对象的属性，再调用 update 方法实现，如

  ```
  >>> model = o.get_online_model('online_model_name')
  ```

```
>>> model.cpu = 200
>>> model.update()
```

与其他对象不同的是，在线模型的创建和删除较为耗时。PyODPS 默认 create_online_model 和 delete_online_model 以及 OnlineModel 的 update 方法在整个操作完成后才返回。用户可以通过 async 选项控制是否要在模型创建请求提交后立即返回， 然后自己控制等待。例如，下列语句

```
>>> model = o.create_online_model('online_model_name', 'offline_mo
del_name')
```

等价于

```
>>> model = o.create_online_model('online_model_name', 'offline_mo
del_name', async=True)
>>> model.wait_for_service()
```

而

```
>>> o.delete_online_model('online_model_name')
```

等价于

```
>>> o.delete_online_model('* online_model_name *', async=True)
>>> model.wait_for_deletion()
```

3.4 DataFrame

3.4.1 DataFrame概述

PyODPS 提供了 DataFrame API，它提供了类似 pandas 的接口，但是能充分利用 ODPS 的计算能力； 同时能在本地使用同样的接口，用 pandas 进行计算。

- 快速开始
- 创建 *DataFrame*
- *Sequence*
- *Collection*
- 执行
- *MapReduce API*
- 列运算
- 聚合操作

- 排序、去重、采样、数据变换
- 使用自定义函数
- 数据合并
- 窗口函数
- 绘图
- 调试指南

3.4.2 快速开始

在本例子中，我们拿 *movielens 100K* 来做例子。现在我们已经有三张表了，分别是`pyodps_ml_100k_movies`（电影相关的数据），`pyodps_ml_100k_users`（用户相关的数据），`pyodps_ml_100k_ratings`（评分有关的数据）。

如果你的运行环境没有提供 ODPS 对象，你需要自己创建该对象：

```
>>> from odps import ODPS
>>> o = ODPS('**your-access-id**', '**your-secret-access-key**',
>>>          project='**your-project**', endpoint='**your-end-point
**'))
```

创建一个DataFrame对象十分容易，只需传入Table对象即可。

```
>>> from odps.df import DataFrame
>>> users = DataFrame(o.get_table('pyodps_ml_100k_users'))
```

我们可以通过dtypes属性来查看这个DataFrame有哪些字段，分别是什么类型

```
>>> users.dtypes
odps.Schema {
  user_id             int64
  age                 int64
  sex                 string
  occupation          string
  zip_code            string
}
```

通过head方法，我们能取前N条数据，这让我们能快速预览数据。

```
>>> users.head(10)
   user_id  age  sex     occupation  zip_code
0        1   24    M     technician     85711
1        2   53    F          other     94043
2        3   23    M         writer     32067
3        4   24    M     technician     43537
4        5   33    F          other     15213
5        6   42    M      executive     98101
6        7   57    M  administrator     91344
7        8   36    M  administrator     05201
8        9   29    M        student     01002
```

```
9       10   53    M         lawyer     90703
```

有时候，我们并不需要都看到所有字段，我们可以从中筛选出一部分。

```
>>> users[['user_id', 'age']].head(5)
   user_id  age
0        1   24
1        2   53
2        3   23
3        4   24
4        5   33
```

有时候我们只是排除个别字段。

```
>>> users.exclude('zip_code', 'age').head(5)
   user_id  sex  occupation
0        1    M  technician
1        2    F       other
2        3    M      writer
3        4    M  technician
4        5    F       other
```

又或者，排除掉一些字段的同时，得通过计算得到一些新的列，比如我想将sex为M的置为True，否则为False，并取名叫sex_bool。

```
>>> users.select(users.exclude('zip_code', 'sex'), sex_bool=users.sex
 == 'M').head(5)
   user_id  age  occupation  sex_bool
0        1   24  technician      True
1        2   53       other     False
2        3   23      writer      True
3        4   24  technician      True
4        5   33       other     False
```

现在，让我们看看年龄在20到25岁之间的人有多少个

```
>>> users[users.age.between(20, 25)].count()
195
```

接下来，我们看看男女用户分别有多少。

```
>>> users.groupby(users.sex).agg(count=users.count())
   sex  count
0    F    273
1    M    670
```

用户按职业划分，从高到底，人数最多的前10职业是哪些呢？

```
>>> df = users.groupby('occupation').agg(count=users['occupation'].
count())
>>> df.sort(df['count'], ascending=False)[:10]
      occupation  count
0        student    196
1          other    105
2       educator     95
```

```
3  administrator     79
4       engineer     67
5     programmer     66
6      librarian     51
7         writer     45
8      executive     32
9      scientist     31
```

DataFrame API提供了value_counts这个方法来快速达到同样的目的。

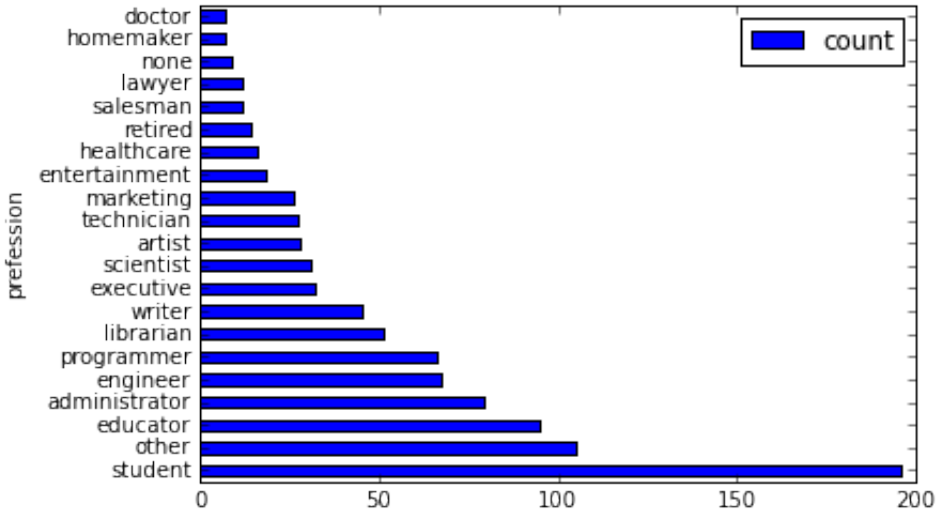
```
>>> users.occupation.value_counts()[:10]
      occupation  count
0        student    196
1          other    105
2       educator     95
3  administrator     79
4       engineer     67
5     programmer     66
6      librarian     51
7         writer     45
8      executive     32
9      scientist     31
```

让我们用更直观的图来看这份数据。

```
>>> %matplotlib inline
```

我们可以用个横向的柱状图来可视化

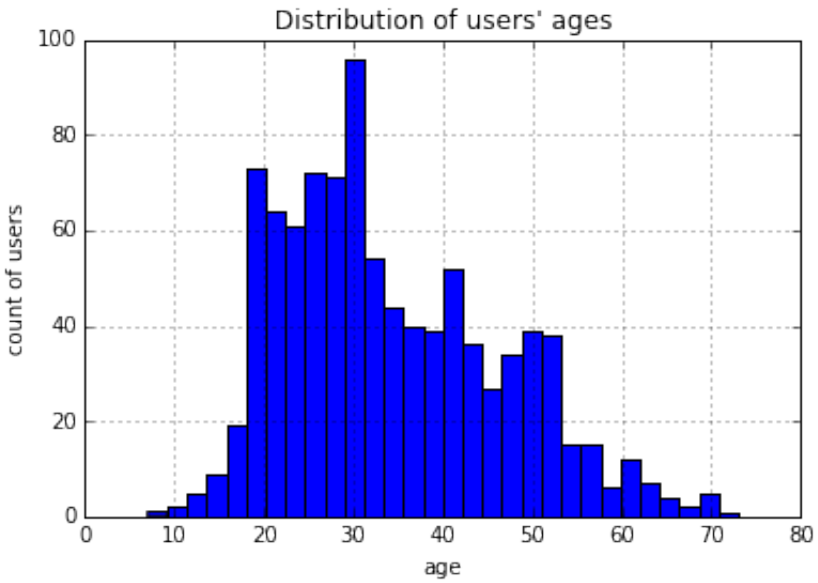
```
>>> users['occupation'].value_counts().plot(kind='barh', x='occupation
', ylabel='prefession')
<matplotlib.axes._subplots.AxesSubplot at 0x10653cfd0>
```



我们将年龄分成30组，来看个年龄分布的直方图

```
>>> users.age.hist(bins=30, title="Distribution of users' ages",
xlabel='age', ylabel='count of users')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10667a510>
```



好了，现在我们把这三张表联合起来，只需要使用join就可以了。join完成后我们把它保存成一张新的表。

```
>>> movies = DataFrame(o.get_table('pyodps_ml_100k_movies'))
>>> ratings = DataFrame(o.get_table('pyodps_ml_100k_ratings'))
>>>
>>> o.delete_table('pyodps_ml_100k_lens', if_exists=True)
>>> lens = movies.join(ratings).join(users).persist('pyodps_ml_
100k_lens')
>>>
>>> lens.dtypes
odps.Schema {
  movie_id                                 int64
  title                                    string
  release_date                             string
  video_release_date                       string
  imdb_url                                 string
  user_id                                  int64
  rating                                   int64
  unix_timestamp                           int64
  age                                      int64
  sex                                      string
  occupation                               string
  zip_code                                 string
}
```

现在我们把年龄分成从0到80岁，分成8个年龄段，

```
>>> labels = ['0-9', '10-19', '20-29', '30-39', '40-49', '50-59', '60-
69', '70-79']
```

```
>>> cut_lens = lens[lens, lens.age.cut(range(0, 81, 10), right=False,
labels=labels).rename('年龄分组')]
```

我们取分组和年龄唯一的前10条看看。

```
>>> cut_lens['年龄分组', 'age'].distinct()[:10]
   年龄分组  age
0       0-9    7
1     10-19   10
2     10-19   11
3     10-19   13
4     10-19   14
5     10-19   15
6     10-19   16
7     10-19   17
8     10-19   18
9     10-19   19
```

最后，我们来看看在各个年龄分组下，用户的评分总数和评分均值分别是多少。

```
>>> cut_lens.groupby('年龄分组').agg(cut_lens.rating.count().rename('评
分总数'), cut_lens.rating.mean().rename('评分均值'))
     年龄分组   评分均值   评分总数
0       0-9  3.767442        43
1     10-19  3.486126      8181
2     20-29  3.467333     39535
3     30-39  3.554444     25696
4     40-49  3.591772     15021
5     50-59  3.635800      8704
6     60-69  3.648875      2623
7     70-79  3.649746       197
```

3.4.3 创建 DataFrame

在使用 DataFrame 时，你需要了解三个对象上的操作：`Collection(DataFrame)`，`Sequence`，`Scalar`。 这三个对象分别表示表结构（或者二维结构）、列（一维结构）、标量。需要注意的是，这些对象仅在使用 Pandas 数据创建后会包含实际数据， 而在 ODPS 表上创建的对象中并不包含实际的数据，而仅仅包含对这些数据的操作，实质的存储和计算会在 ODPS 中进行。

创建 DataFrame

通常情况下，你唯一需要直接创建的 Collection 对象是 *DataFrame Reference*，这一对象用于引用数据源，可能是一个 ODPS 表， ODPS 分区，Pandas DataFrame或sqlalchemy.Table（数据库表）。 使用这几种数据源时，相关的操作相同，这意味着你可以不更改数据处理的代码，仅仅修改输入/输出的指向， 便可以简单地将小数据量上本地测试运行的代码迁移到 ODPS 上，而迁移的正确性由 PyODPS 来保证。

创建 DataFrame 非常简单，只需将 Table 对象、 pandas DataFrame 对象或者 sqlalchemy Table 对象传入即可。

```
>>> from odps.df import DataFrame
>>>
>>> # 从 ODPS 表创建
>>> iris = DataFrame(o.get_table('pyodps_iris'))
>>> iris2 = o.get_table('pyodps_iris').to_df()  # 使用表的to_df方法
>>>
>>> # 从 ODPS 分区创建
>>> pt_df = DataFrame(o.get_table('partitioned_table').get_partition('
pt=20171111'))
>>> pt_df2 = o.get_table('partitioned_table').get_partition('pt=
20171111').to_df()  # 使用分区的to_df方法
>>>
>>> # 从 Pandas DataFrame 创建
>>> import pandas as pd
>>> import numpy as np
>>> df = DataFrame(pd.DataFrame(np.arange(9).reshape(3, 3), columns=
list('abc')))
>>>
>>> # 从 sqlalchemy Table 创建
>>> engine = sqlalchemy.create_engine('mysql://root:123456@localhost/
movielens')
>>> metadata = sqlalchemy.MetaData(bind=engine) # 需要绑定到engine
>>> table = sqlalchemy.Table('top_users', metadata, extend_existing=
True, autoload=True)
>>> users = DataFrame(table)
```

在用 pandas DataFrame 初始化时，对于 numpy object 类型或者 string 类型，PyODPS DataFrame 会尝试推断类型， 如果一整列都为空，则会报错。这时，用户可以指定 unknown_as_string

为True，会将这些列指定为string类型。 用户也可以指定 as_type 参数。若类型为基本类型，会在创建 PyODPS DataFrame 时进行强制类型转换。 如果 Pandas DataFrame 中包含 list 或者 dict 列，该列的类型不会被推断，必须手动使用 as_type 指定。 as_type 参数类型必须是dict。

```
>>> df2 = DataFrame(df, unknown_as_string=True, as_type={'null_col2':
 'float'})
>>> df2.dtypes
odps.Schema {
  sepallength           float64
  sepalwidth            float64
  petallength           float64
  petalwidth            float64
  name                  string
  null_col1             string   # 无法识别，通过unknown_as_string设置成
string类型
  null_col2             float64  # 强制转换成float类型
}
>>> df4 = DataFrame(df3, as_type={'list_col': 'list<int64>'})
>>> df4.dtypes
odps.Schema {
```

```
  id         int64
  list_col   list<int64>  # 无法识别且无法自动转换，通过 as_type 设置
}
```

3.4.4 Sequence

SequenceExpr 代表了二维数据集中的一列。你不应当手动创建 SequenceExpr，而应当从一个 Collection 中获取。

获取列

你可以使用 collection.column_name 取出一列，例如

```
>>> iris.sepallength.head(5)
   sepallength
0          5.1
1          4.9
2          4.7
3          4.6
4          5.0
```

如果列名存储在一个字符串变量中，除了使用 getattr(df, 'column_name') 达到相同的效果外，也可以使用 df[column_name] 的形式，例如

```
>>> iris['sepallength'].head(5)
   sepallength
0          5.1
1          4.9
2          4.7
3          4.6
4          5.0
```

列类型

DataFrame包括自己的类型系统，在使用Table初始化的时候，ODPS的类型会被进行转换。这样做的好处是，能支持更多的计算后端。 目前，DataFrame的执行后端支持ODPS SQL、pandas以及数据库（MySQL和Postgres）。

PyODPS DataFrame 包括以下类型：

`int8`，`int16`，`int32`，`int64`，`float32`，`float64`，`boolean`，`string`，`decimal`，`datetime`，`list`，`dict`

ODPS的字段和DataFrame的类型映射关系如下：

ODPS类型	**DataFrame类型**
bigint	int64
double	float64

ODPS类型	DataFrame类型
string	string
datetime	datetime
boolean	boolean
decimal	decimal
array<value_type>	list<value_type>
map<key_type, value_type>	dict<key_type, value_type>

list 和 dict 必须填写其包含值的类型，否则会报错。目前 DataFrame 暂不支持 MaxCompute 2.0 中新增的 Timestamp 及 Struct 类型，未来的版本会支持。

在 Sequence 中可以通过 sequence.dtype 获取数据类型：

```
>>> iris.sepallength.dtype
float64
```

如果要修改一列的类型，可以使用 astype 方法。该方法输入一个类型，并返回类型转换后的 Sequence。例如，

```
>>> iris.sepallength.astype('int')
   sepallength
0            5
1            4
2            4
3            4
4            5
```

列名

在 DataFrame 的计算过程中，一个 Sequence 必须要有列名。在很多情况下，DataFrame 会起一个名字。比如：

```
>>> iris.groupby('name').sepalwidth.max()
   sepalwidth_max
0             4.4
1             3.4
2             3.8
```

可以看到，`sepalwidth`取最大值后被命名为`sepalwidth_max`。还有一些操作，比如一个 Sequence 做加法，加上一个 Scalar，这时，会被命名为这个 Sequence 的名字。其它情况下，需要用户去自己命名。

Sequence 提供 rename 方法对一列进行重命名，用法示例如下：

```
>>> iris.sepalwidth.rename('sepal_width').head(5)
   sepal_width
0          3.5
1          3.0
2          3.2
3          3.1
4          3.6
```

简单的列变换

你可以对一个 Sequence 进行运算，返回一个新的 Sequence，正如对简单的 Python 变量进行运算一样。对数值列， Sequence 支持四则运算，而对字符串则支持字符串相加等操作。例如，

```
>>> (iris.sepallength + 5).head(5)
   sepallength
0         10.1
1          9.9
2          9.7
3          9.6
4         10.0
```

而

```
>>> (iris.sepallength + iris.sepalwidth).rename('sum_sepal').head(5)
   sum_sepal
0        8.6
1        7.9
2        7.9
3        7.7
4        8.6
```

注意到两列参与运算，因而 PyODPS 无法确定最终显示的列名，需要手动指定。详细的列变换说明，请参见 列运算。

3.4.5 Collection

DataFrame 中所有二维数据集上的操作都属于 *PyODPS DataFrame指南#DataFrame Reference#*，可视为一张 ODPS 表或一张电子表单，DataFrame 对象也是 CollectionExpr 的特例。

CollectionExpr 中包含针对二维数据集的列操作、筛选、变换等大量操作。

获取类型

`dtypes`可以用来获取 CollectionExpr 中所有列的类型。`dtypes` 返回的是*Schema类型* 。

```
>>> iris.dtypes
odps.Schema {
  sepallength           float64
```

```
  sepalwidth            float64
  petallength           float64
  petalwidth            float64
  name                  string
}
```

列选择和增删

如果要从一个 CollectionExpr 中选取部分列，产生新的数据集，可以使用 expr[columns] 语法。例如，

```
>>> iris['name', 'sepallength'].head(5)
          name  sepallength
0  Iris-setosa          5.1
1  Iris-setosa          4.9
2  Iris-setosa          4.7
3  Iris-setosa          4.6
4  Iris-setosa          5.0
```

说明：

如果需要选择的列只有一列，需要在 columns 后加上逗号或者显示标记为列表，例如 df[df.sepal_length,] 或 df[[df.sepal_length]]，否则返回的将是一个 Sequence 对象，而不是 Collection。

如果想要在新的数据集中排除已有数据集的某些列，可使用 exclude 方法：

```
>>> iris.exclude('sepallength', 'petallength')[:5]
   sepalwidth  petalwidth         name
0         3.5         0.2  Iris-setosa
1         3.0         0.2  Iris-setosa
2         3.2         0.2  Iris-setosa
3         3.1         0.2  Iris-setosa
4         3.6         0.2  Iris-setosa
```

0.7.2 以后的 PyODPS 支持另一种写法，即在数据集上直接排除相应的列：

```
>>> del iris['sepallength']
>>> del iris['petallength']
>>> iris[:5]
   sepalwidth  petalwidth         name
0         3.5         0.2  Iris-setosa
1         3.0         0.2  Iris-setosa
2         3.2         0.2  Iris-setosa
3         3.1         0.2  Iris-setosa
4         3.6         0.2  Iris-setosa
```

如果我们需要在已有 collection 中添加某一列变换的结果，也可以使用 expr[expr, new_sequence] 语法， 新增的列会作为新 collection 的一部分。

下面的例子将 iris 中的 sepalwidth 列加一后重命名为 sepalwidthplus1 并追加到数据集末尾，形成新的数据集：

```
>>> iris[iris, (iris.sepalwidth + 1).rename('sepalwidthplus1')].head(5
)
   sepallength  sepalwidth  petallength  petalwidth         name  \
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

   sepalwidthplus1
0              4.5
1              4.0
2              4.2
3              4.1
4              4.6
```

使用 df[df, new_sequence] 需要注意的是，变换后的列名与原列名可能相同，如果需要与原 collection 合并， 请将该列重命名。

0.7.2 以后版本的 PyODPS 支持直接在当前数据集中追加，写法为

```
>>> iris['sepalwidthplus1'] = iris.sepalwidth + 1
>>> iris.head(5)
   sepallength  sepalwidth  petallength  petalwidth         name  \
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa

   sepalwidthplus1
0              4.5
1              4.0
2              4.2
3              4.1
4              4.6
```

我们也可以先将原列通过 exclude 方法进行排除，再将变换后的新列并入，而不必担心重名。

```
>>> iris[iris.exclude('sepalwidth'), iris.sepalwidth * 2].head(5)
   sepallength  petallength  petalwidth         name  sepalwidth
0          5.1          1.4         0.2  Iris-setosa         7.0
1          4.9          1.4         0.2  Iris-setosa         6.0
2          4.7          1.3         0.2  Iris-setosa         6.4
3          4.6          1.5         0.2  Iris-setosa         6.2
4          5.0          1.4         0.2  Iris-setosa         7.2
```

对于 0.7.2 以后版本的 PyODPS，如果想在当前数据集上直接覆盖，则可以写

```
>>> iris['sepalwidth'] = iris.sepalwidth * 2
>>> iris.head(5)
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.1         7.0          1.4         0.2  Iris-setosa
```

```
1          4.9          6.0          1.4          0.2  Iris-setosa
2          4.7          6.4          1.3          0.2  Iris-setosa
3          4.6          6.2          1.5          0.2  Iris-setosa
4          5.0          7.2          1.4          0.2  Iris-setosa
```

增删列以创建新 collection 的另一种方法是调用 select 方法，将需要选择的列作为参数输入。如果需要重命名，使用 keyword 参数输入，并将新的列名作为参数名即可。

```
>>> iris.select('name', sepalwidthminus1=iris.sepalwidth - 1).head(5)
          name  sepalwidthminus1
0  Iris-setosa               2.5
1  Iris-setosa               2.0
2  Iris-setosa               2.2
3  Iris-setosa               2.1
4  Iris-setosa               2.6
```

此外，我们也可以传入一个 lambda 表达式，它接收一个参数，接收上一步的结果。在执行时，PyODPS 会检查这些 lambda 表达式，传入上一步生成的 collection 并将其替换为正确的列。

```
>>> iris['name', 'petallength'][[lambda x: x.name]].head(5)
          name
0  Iris-setosa
1  Iris-setosa
2  Iris-setosa
3  Iris-setosa
4  Iris-setosa
```

此外，在 0.7.2 以后版本的 PyODPS 中，支持对数据进行条件赋值，例如

```
>>> iris[iris.sepallength > 5.0, 'sepalwidth'] = iris.sepalwidth * 2
>>> iris.head(5)
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.1        14.0          1.4          0.2  Iris-setosa
1          4.9         6.0          1.4          0.2  Iris-setosa
2          4.7         6.4          1.3          0.2  Iris-setosa
3          4.6         6.2          1.5          0.2  Iris-setosa
4          5.0         7.2          1.4          0.2  Iris-setosa
```

引入常数和随机数

DataFrame 支持在 collection 中追加一列常数。追加常数需要使用 *Scalar*，引入时需要手动指定列名，如

```
>>> from odps.df import Scalar
>>> iris[iris, Scalar(1).rename('id')][:5]
   sepallength  sepalwidth  petallength  petalwidth         name  id
0          5.1         3.5          1.4          0.2  Iris-setosa   1
1          4.9         3.0          1.4          0.2  Iris-setosa   1
2          4.7         3.2          1.3          0.2  Iris-setosa   1
3          4.6         3.1          1.5          0.2  Iris-setosa   1
```

```
4          5.0          3.6           1.4          0.2   Iris-setosa    1
```

如果需要指定一个空值列，可以使用*NullScalar* ，需要提供字段类型。

```
>>> from odps.df import NullScalar
>>> iris[iris, NullScalar('float').rename('fid')][:5]
   sepal_length  sepal_width  petal_length  petal_width     category
  fid
0           5.1          3.5           1.4          0.2  Iris-setosa
None
1           4.9          3.0           1.4          0.2  Iris-setosa
None
2           4.7          3.2           1.3          0.2  Iris-setosa
None
3           4.6          3.1           1.5          0.2  Iris-setosa
None
4           5.0          3.6           1.4          0.2  Iris-setosa
None
```

在 PyODPS 0.7.12 及以后版本中，引入了简化写法：

```
>>> iris['id'] = 1
>>> iris
   sepallength  sepalwidth  petallength  petalwidth         name  id
0          5.1         3.5          1.4         0.2  Iris-setosa   1
1          4.9         3.0          1.4         0.2  Iris-setosa   1
2          4.7         3.2          1.3         0.2  Iris-setosa   1
3          4.6         3.1          1.5         0.2  Iris-setosa   1
4          5.0         3.6          1.4         0.2  Iris-setosa   1
```

需要注意的是，这种写法无法自动识别空值的类型，所以在增加空值列时，仍然要使用

```
>>> iris['null_col'] = NullScalar('float')
>>> iris
   sepallength  sepalwidth  petallength  petalwidth         name
null_col
0          5.1         3.5          1.4         0.2  Iris-setosa
None
1          4.9         3.0          1.4         0.2  Iris-setosa
None
2          4.7         3.2          1.3         0.2  Iris-setosa
None
3          4.6         3.1          1.5         0.2  Iris-setosa
None
4          5.0         3.6          1.4         0.2  Iris-setosa
None
```

DataFrame 也支持在 collection 中增加一列随机数列，该列类型为 float，范围为 0 - 1，每行数值均不同。 追加随机数列需要使用 *RandomScalar*，参数为随机数种子，可省略。

```
>>> from odps.df import RandomScalar
>>> iris[iris, RandomScalar().rename('rand_val')][:5]
   sepallength  sepalwidth  petallength  petalwidth         name
rand_val
0          5.1         3.5          1.4         0.2  Iris-setosa  0.
000471
```

```
1           4.9          3.0           1.4          0.2  Iris-setosa  0.
799520
2           4.7          3.2           1.3          0.2  Iris-setosa  0.
834609
3           4.6          3.1           1.5          0.2  Iris-setosa  0.
106921
4           5.0          3.6           1.4          0.2  Iris-setosa  0.
763442
```

过滤数据

Collection 提供了数据过滤的功能，我们试着查询sepallength大于5的几条数据。

```
>>> iris[iris.sepallength > 5].head(5)
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          5.4         3.9          1.7         0.4  Iris-setosa
2          5.4         3.7          1.5         0.2  Iris-setosa
3          5.8         4.0          1.2         0.2  Iris-setosa
4          5.7         4.4          1.5         0.4  Iris-setosa
```

多个查询条件：

```
>>> iris[(iris.sepallength < 5) & (iris['petallength'] > 1.5)].head(5)
   sepallength  sepalwidth  petallength  petalwidth             name
0          4.8         3.4          1.6         0.2      Iris-setosa
1          4.8         3.4          1.9         0.2      Iris-setosa
2          4.7         3.2          1.6         0.2      Iris-setosa
3          4.8         3.1          1.6         0.2      Iris-setosa
4          4.9         2.4          3.3         1.0  Iris-versicolor
```

或条件：

```
>>> iris[(iris.sepalwidth < 2.5) | (iris.sepalwidth > 4)].head(5)
   sepallength  sepalwidth  petallength  petalwidth             name
0          5.7         4.4          1.5         0.4      Iris-setosa
1          5.2         4.1          1.5         0.1      Iris-setosa
2          5.5         4.2          1.4         0.2      Iris-setosa
3          4.5         2.3          1.3         0.3      Iris-setosa
4          5.5         2.3          4.0         1.3  Iris-versicolor
```

说明：

记住，与和或条件必须使用&和|，不能使用and和or。

非条件：

```
>>> iris[~(iris.sepalwidth > 3)].head(5)
   sepallength  sepalwidth  petallength  petalwidth         name
0          4.9         3.0          1.4         0.2  Iris-setosa
1          4.4         2.9          1.4         0.2  Iris-setosa
2          4.8         3.0          1.4         0.1  Iris-setosa
3          4.3         3.0          1.1         0.1  Iris-setosa
```

```
4          5.0         3.0          1.6         0.2  Iris-setosa
```

我们也可以显式调用filter方法，提供多个与条件

```
>>> iris.filter(iris.sepalwidth > 3.5, iris.sepalwidth < 4).head(5)
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.0         3.6          1.4         0.2  Iris-setosa
1          5.4         3.9          1.7         0.4  Iris-setosa
2          5.4         3.7          1.5         0.2  Iris-setosa
3          5.4         3.9          1.3         0.4  Iris-setosa
4          5.7         3.8          1.7         0.3  Iris-setosa
```

同样对于连续的操作，我们可以使用lambda表达式

```
>>> iris[iris.sepalwidth > 3.8]['name', lambda x: x.sepallength + 1]
          name  sepallength
0  Iris-setosa          6.4
1  Iris-setosa          6.8
2  Iris-setosa          6.7
3  Iris-setosa          6.4
4  Iris-setosa          6.2
5  Iris-setosa          6.5
```

对于Collection，如果它包含一个列是boolean类型，则可以直接使用该列作为过滤条件。

```
>>> df.dtypes
odps.Schema {
  a boolean
  b int64
}
>>> df[df.a]
      a  b
0  True  1
1  True  3
```

因此，记住对Collection取单个squence的操作时，只有boolean列是合法的，即对Collection作过滤操作。

```
>>> df[df.a, ]       # 取列操作
>>> df[[df.a]]       # 取列操作
>>> df.select(df.a)  # 显式取列
>>> df[df.a]         # a列是boolean列，执行过滤操作
>>> df.a             # 取单列
>>> df['a']          # 取单列
```

同时，我们也支持Pandas中的`query`方法，用查询语句来做数据的筛选，在表达式中直接使用列名如`sepallength`进行操作， 另外在查询语句中`&`和`and`都表示与操作，`|`和`or`都表示或操作。

```
>>> iris.query("(sepallength < 5) and (petallength > 1.5)").head(5)
   sepallength  sepalwidth  petallength  petalwidth            name
0          4.8         3.4          1.6         0.2     Iris-setosa
1          4.8         3.4          1.9         0.2     Iris-setosa
2          4.7         3.2          1.6         0.2     Iris-setosa
3          4.8         3.1          1.6         0.2     Iris-setosa
```

```
4          4.9         2.4          3.3         1.0  Iris-versicolor
```

当表达式中需要使用到本地变量时，需要在该变量前加一个@前缀。

```
>>> var = 4
>>> iris.query("(iris.sepalwidth < 2.5) | (sepalwidth > @var)").head(5
)
   sepallength  sepalwidth  petallength  petalwidth            name
0          5.7         4.4          1.5         0.4      Iris-setosa
1          5.2         4.1          1.5         0.1      Iris-setosa
2          5.5         4.2          1.4         0.2      Iris-setosa
3          4.5         2.3          1.3         0.3      Iris-setosa
4          5.5         2.3          4.0         1.3  Iris-versicolor
```

目前query支持的语法包括：

语法	说明
name	没有 @前缀的都当做列名处理，有前缀的会获取本地变量
operator	支持部分运算符：+，-，*，/，//，%，**，==，!=，<，<=，>，>=，in，not in
bool	与或非操作，其中 &和 and表示与，|和 or表示或
attribute	取对象属性
index, slice, Subscript	切片操作

并列多行输出

对于 list 及 map 类型的列，explode 方法会将该列转换为多行输出。使用 apply 方法也可以输出多行。 为了进行聚合等操作，常常需要将这些输出和原表中的列合并。此时可以使用 DataFrame 提供的并列多行输出功能， 写法为将多行输出函数生成的集合与原集合中的列名一起映射。

并列多行输出的例子如下：

```
>>> df
   id         a             b
0   1  [a1, b1]  [a2, b2, c2]
1   2      [c1]      [d2, e2]
>>> df[df.id, df.a.explode(), df.b]
   id   a             b
0   1  a1  [a2, b2, c2]
1   1  b1  [a2, b2, c2]
2   2  c1      [d2, e2]
>>> df[df.id, df.a.explode(), df.b.explode()]
   id   a   b
0   1  a1  a2
1   1  a1  b2
2   1  a1  c2
```

```
3   1  b1  a2
4   1  b1  b2
5   1  b1  c2
6   2  c1  d2
7   2  c1  e2
```

如果多行输出方法对某个输入不产生任何输出，默认输入行将不在最终结果中出现。如果需要在结果中出现该行，可以设置

`keep_nulls=True`

。此时，与该行并列的值将输出为空值：

```
>>> df
   id         a
0   1  [a1, b1]
1   2        []
>>> df[df.id, df.a.explode()]
   id   a
0   1  a1
1   1  b1
>>> df[df.id, df.a.explode(keep_nulls=True)]
   id     a
0   1    a1
1   1    b1
2   2  None
```

关于 explode 使用并列输出的具体文档可参考 集合类型相关操作，对于 apply 方法使用并列输出的例子可参考 对一行数据使用自定义函数。

限制条数

```
>>> iris[:3]
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
```

值得注意的是，目前切片对于ODPS SQL后端不支持start和step。我们也可以使用limit方法

```
>>> iris.limit(3)
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
```

说明：

另外，切片操作只能作用在collection上，不能作用于sequence。

3.4.6 执行

延迟执行

DataFrame上的所有操作并不会立即执行，只有当用户显式调用execute方法，或者一些立即执行的方法时（内部调用的就是execute），才会真正去执行。

这些立即执行的方法包括：

方法	说明	返回值
persist	将执行结果保存到ODPS表	PyODPS DataFrame
execute	执行并返回全部结果	ResultFrame
head	查看开头N行数据，这个方法会执行所有结果，并取开头N行数据	ResultFrame
tail	查看结尾N行数据，这个方法会执行所有结果，并取结尾N行数据	ResultFrame
to_pandas	转化为 Pandas DataFrame 或者 Series，wrap 参数为 True 的时候，返回 PyODPS DataFrame 对象	wrap为True返回PyODPS DataFrame，False（默认）返回pandas DataFrame
plot，hist，boxplot	画图有关	

说明：

在交互式环境下，PyODPS DataFrame会在打印或者repr的时候，调用execute方法，这样省去了用户手动去调用execute。

```
>>> iris[iris.sepallength < 5][:5]
   sepallength  sepalwidth  petallength  petalwidth         name
0          4.9         3.0          1.4         0.2  Iris-setosa
1          4.7         3.2          1.3         0.2  Iris-setosa
2          4.6         3.1          1.5         0.2  Iris-setosa
3          4.6         3.4          1.4         0.3  Iris-setosa
4          4.4         2.9          1.4         0.2  Iris-setosa
```

如果想关闭自动调用执行，则需要手动设置

```
>>> from odps import options
>>> options.interactive = False
>>>
>>> iris[iris.sepallength < 5][:5]
```

```
Collection: ref_0
  odps.Table
    name: odps_test_sqltask_finance.`pyodps_iris`
    schema:
      sepallength           : double
      sepalwidth            : double
      petallength           : double
      petalwidth            : double
      name                  : string

Collection: ref_1
  Filter[collection]
    collection: ref_0
    predicate:
      Less[sequence(boolean)]
        sepallength = Column[sequence(float64)] 'sepallength' from
collection ref_0
        Scalar[int8]
          5

Slice[collection]
  collection: ref_1
  stop:
    Scalar[int8]
      5
```

此时打印或者repr对象，会显示整棵抽象语法树。

说明：

ResultFrame是结果集合，不能参与后续计算。

ResultFrame可以迭代取出每条记录：

```
>>> result = iris.head(3)
>>> for r in result:
>>>     print(list(r))
[5.0999999999999996, 3.5, 1.3999999999999999, 0.20000000000000001, u'
Iris-setosa']
[4.9000000000000004, 3.0, 1.3999999999999999, 0.20000000000000001, u'
Iris-setosa']
[4.7000000000000002, 3.2000000000000002, 1.3, 0.20000000000000001, u'
Iris-setosa']
```

ResultFrame 也支持在安装有 pandas 的前提下转换为 pandas DataFrame 或使用 pandas 后端的 PyODPS DataFrame：

```
>>> pd_df = iris.head(3).to_pandas()  # 返回 pandas DataFrame
```

```
>>> wrapped_df = iris.head(3).to_pandas(wrap=True)  # 返回使用 Pandas 后
端的 PyODPS DataFrame
```

保存执行结果为 **ODPS** 表

对 Collection，我们可以调用persist方法，参数为表名。返回一个新的DataFrame对象

```
>>> iris2 = iris[iris.sepalwidth < 2.5].persist('pyodps_iris2')
>>> iris2.head(5)
   sepallength  sepalwidth  petallength  petalwidth            name
0          4.5         2.3          1.3         0.3     Iris-setosa
1          5.5         2.3          4.0         1.3  Iris-versicolor
2          4.9         2.4          3.3         1.0  Iris-versicolor
3          5.0         2.0          3.5         1.0  Iris-versicolor
4          6.0         2.2          4.0         1.0  Iris-versicolor
```

persist可以传入partitions参数，这样会创建一个表，它的分区是partitions所指定的字段。

```
>>> iris3 = iris[iris.sepalwidth < 2.5].persist('pyodps_iris3',
partitions=['name'])
>>> iris3.data
odps.Table
  name: odps_test_sqltask_finance.`pyodps_iris3`
  schema:
    sepallength           : double
    sepalwidth            : double
    petallength           : double
    petalwidth            : double
  partitions:
    name                  : string
```

如果想写入已经存在的表的某个分区，persist可以传入partition参数，指明写入表的哪个分区（如ds=******）。 这时要注意，该DataFrame的每个字段都必须在该表存在，且类型相同。drop_partition和create_partition参数只有在此时有效, 分别表示是否要删除（如果分区存在）或创建（如果分区不存在）该分区。

```
>>> iris[iris.sepalwidth < 2.5].persist('pyodps_iris4', partition='ds=
test', drop_partition=True, create_partition=True)
```

写入表时，还可以指定表的生命周期，如下列语句将表的生命周期指定为10天：

```
>>> iris[iris.sepalwidth < 2.5].persist('pyodps_iris5', lifecycle=10)
```

如果数据源中没有 ODPS 对象，例如数据源仅为 Pandas，在 persist 时需要手动指定 ODPS 入口对象， 或者将需要的入口对象标明为全局对象，如：

```
>>> # 假设入口对象为 o
>>> # 指定入口对象
>>> df.persist('table_name', odps=o)
>>> # 或者可将入口对象标记为全局
>>> o.to_global()
```

```
>>> df.persist('table_name')
```

保存执行结果为 **Pandas DataFrame**

我们可以使用 `to_pandas`方法，如果wrap参数为True，将返回PyODPS DataFrame对象。

```
>>> type(iris[iris.sepalwidth < 2.5].to_pandas())
pandas.core.frame.DataFrame
>>> type(iris[iris.sepalwidth < 2.5].to_pandas(wrap=True))
odps.df.core.DataFrame
```

立即运行设置运行参数

对于立即执行的方法，比如 `execute`、`persist`、`to_pandas` 等，可以设置运行时参数（仅对ODPS SQL后端有效 ）。

一种方法是设置全局参数。详细参考 *SQL*设置运行参数 。

也可以在这些立即执行的方法上，使用 `hints` 参数。这样，这些参数只会作用于当前的计算过程。

```
>>> iris[iris.sepallength < 5].to_pandas(hints={'odps.sql.mapper.split
.size': 16})
```

运行时显示详细信息

有时，用户需要查看运行时instance的logview时，需要修改全局配置：

```
>>> from odps import options
>>> options.verbose = True
>>>
>>> iris[iris.sepallength < 5].exclude('sepallength')[:5].execute()
Sql compiled:
SELECT t1.`sepalwidth`, t1.`petallength`, t1.`petalwidth`, t1.`name`
FROM odps_test_sqltask_finance.`pyodps_iris` t1
WHERE t1.`sepallength` < 5
LIMIT 5
logview:
http://logview

   sepalwidth  petallength  petalwidth         name
0         3.0          1.4         0.2  Iris-setosa
1         3.2          1.3         0.2  Iris-setosa
2         3.1          1.5         0.2  Iris-setosa
3         3.4          1.4         0.3  Iris-setosa
4         2.9          1.4         0.2  Iris-setosa
```

用户可以指定自己的日志记录函数，比如像这样：

```
>>> my_logs = []
>>> def my_logger(x):
>>>     my_logs.append(x)
>>>
>>> options.verbose_log = my_logger
```

```
>>>
>>> iris[iris.sepallength < 5].exclude('sepallength')[:5].execute()
   sepalwidth  petallength  petalwidth         name
0         3.0          1.4         0.2  Iris-setosa
1         3.2          1.3         0.2  Iris-setosa
2         3.1          1.5         0.2  Iris-setosa
3         3.4          1.4         0.3  Iris-setosa
4         2.9          1.4         0.2  Iris-setosa

>>> print(my_logs)
['Sql compiled:', 'SELECT t1.`sepalwidth`, t1.`petallength`, t1.`
petalwidth`, t1.`name` \nFROM odps_test_sqltask_finance.`pyodps_iri
s` t1 \nWHERE t1.`sepallength` < 5 \nLIMIT 5', 'logview:', u'http://
logview']
```

缓存中间Collection计算结果

DataFrame的计算过程中，一些Collection被多处使用，或者用户需要查看中间过程的执行结果，这时用户可以使用 `cache`标记某个collection需要被优先计算。

> **说明：**
> 值得注意的是，`cache`延迟执行，调用cache不会触发立即计算。

```
>>> cached = iris[iris.sepalwidth < 3.5].cache()
>>> df = cached['sepallength', 'name'].head(3)
>>> df
   sepallength         name
0          4.9  Iris-setosa
1          4.7  Iris-setosa
2          4.6  Iris-setosa
>>> cached.head(3)  # 由于cached已经被计算，所以能立刻取到计算结果
   sepallength         name
0          4.9  Iris-setosa
1          4.7  Iris-setosa
2          4.6  Iris-setosa
```

异步和并行执行

DataFrame 支持异步操作，对于立即执行的方法，包括 `execute`、`persist`、`head`、`tail`、`to_pandas`（其他方法不支持），传入 `async` 参数，即可以将一个操作异步执行，`timeout` 参数指定超时时间，异步返回的是 *Future* 对象。

```
>>> from odps.df import Delay
>>> delay = Delay()  # 创建Delay对象
>>>
>>> df = iris[iris.sepal_width < 5].cache()  # 有一个共同的依赖
>>> future1 = df.sepal_width.sum().execute(delay=delay)  # 立即返回
future对象，此时并没有执行
>>> future2 = df.sepal_width.mean().execute(delay=delay)
>>> future3 = df.sepal_length.max().execute(delay=delay)
>>> delay.execute(n_parallel=3)  # 并发度是3，此时才真正执行。
|==========================================|   1 /  1  (100.00%)
  21s
```

```
>>> future1.result()
458.10000000000014
>>> future2.result()
3.0540000000000007
```

可以看到上面的例子里，共同依赖的对象会先执行，然后再以并发度为3分别执行future1到future3。 当 `n_parallel` 为1时，执行时间会达到37s。

`delay.execute` 也接受 `async` 操作来指定是否异步执行，当异步的时候，也可以指定 `timeout` 参数来指定超时时间。

3.4.7 列运算

```
from odps.df import DataFrame
```

```
iris = DataFrame(o.get_table('pyodps_iris'))
lens = DataFrame(o.get_table('pyodps_ml_100k_lens'))
```

对于一个Sequence来说，对它加上一个常量、或者执行sin函数的这类操作时，是作用于每个元素上的。接下来会详细说明。

NULL相关（isnull，notnull，fillna）

DataFrame API提供了几个和NULL相关的内置函数，比如isnull来判断是否某字段是NULL，notnull则相反，fillna是将NULL填充为用户指定的值。

```
>>> iris.sepallength.isnull().head(5)
   sepallength
0        False
1        False
2        False
3        False
4        False
```

逻辑判断（ifelse，switch）

`ifelse`

作用于boolean类型的字段，当条件成立时，返回第0个参数，否则返回第1个参数。

```
>>> (iris.sepallength > 5).ifelse('gt5', 'lte5').rename('cmp5').head(5
)
   cmp5
0   gt5
1  lte5
2  lte5
3  lte5
```

```
4  lte5
```

switch用于多条件判断的情况。

```
>>> iris.sepallength.switch(4.9, 'eq4.9', 5.0, 'eq5.0', default='noeq
').rename('equalness').head(5)
   equalness
0       noeq
1      eq4.9
2       noeq
3       noeq
4      eq5.0
```

```
>>> from odps.df import switch
>>> switch(iris.sepallength == 4.9, 'eq4.9', iris.sepallength == 5.0,
 'eq5.0', default='noeq').rename('equalness').head(5)
   equalness
0       noeq
1      eq4.9
2       noeq
3       noeq
4      eq5.0
```

PyODPS 0.7.8 以上版本支持根据条件修改数据集某一列的一部分值，写法为：

```
>>> iris[iris.sepallength > 5, 'cmp5'] = 'gt5'
>>> iris[iris.sepallength <= 5, 'cmp5'] = 'lte5'
>>> iris.head(5)
   cmp5
0   gt5
1  lte5
2  lte5
3  lte5
4  lte5
```

数学运算

对于数字类型的字段，支持+，-，*，/等操作，也支持log、sin等数学计算。

```
>>> (iris.sepallength * 10).log().head(5)
   sepallength
0     3.931826
1     3.891820
2     3.850148
3     3.828641
4     3.912023
```

```
>>> fields = [iris.sepallength,
>>>           (iris.sepallength / 2).rename('sepallength除以2'),
>>>           (iris.sepallength ** 2).rename('sepallength的平方')]
>>> iris[fields].head(5)
   sepallength  sepallength除以2  sepallength的平方
0          5.1              2.55             26.01
1          4.9              2.45             24.01
2          4.7              2.35             22.09
3          4.6              2.30             21.16
```

```
4           5.0             2.50            25.00
```

算术运算支持的操作包括：

算术操作	说明
abs	绝对值
sqrt	平方根
sin	
sinh	
cos	
cosh	
tan	
tanh	
arccos	
arccosh	
arcsin	
arcsinh	
arctan	
arctanh	
exp	指数函数
expm1	指数减1
log	传入参数表示底是几
log2	
log10	
log1p	log(1+x)
radians	给定角度计算弧度
degrees	给定弧度计算角度
ceil	不小于输入值的最小整数
floor	向下取整，返回比输入值小的整数值
trunc	将输入值截取到指定小数点位置

对于sequence，也支持其于其他sequence或者scalar的比较。

```
>>> (iris.sepallength < 5).head(5)
   sepallength
0        False
1         True
2         True
3         True
4        False
```

值得主意的是，DataFrame API不支持连续操作，比如`3 <= iris.sepallength <= 5`，但是提供了between这个函数来进行是否在某个区间的判断。

```
>>> (iris.sepallength.between(3, 5)).head(5)
   sepallength
0        False
1         True
2         True
3         True
4         True
```

默认情况下，between包含两边的区间，如果计算开区间，则需要设inclusive=False。

```
>>> (iris.sepallength.between(3, 5, inclusive=False)).head(5)
   sepallength
0        False
1         True
2         True
3         True
4        False
```

String 相关操作

DataFrame API提供了一系列针对string类型的Sequence或者Scalar的操作。

```
>>> fields = [
>>>     iris.name.upper().rename('upper_name'),
>>>     iris.name.extract('Iris(.*)', group=1)
>>> ]
>>> iris[fields].head(5)
    upper_name     name
0  IRIS-SETOSA  -setosa
1  IRIS-SETOSA  -setosa
2  IRIS-SETOSA  -setosa
3  IRIS-SETOSA  -setosa
4  IRIS-SETOSA  -setosa
```

string相关操作包括：

string 操作	说明
capitalize	

string 操作	说明
contains	包含某个字符串，如果 regex 参数为 True，则是包含某个正则表达式，默认为 True
count	指定字符串出现的次数
endswith	以某个字符串结尾
startswith	以某个字符串开头
extract	抽取出某个正则表达式，如果 group 不指定，则返回满足整个 pattern 的子串；否则，返回第几个 group
find	返回第一次出现的子串位置，若不存在则返回-1
rfind	从右查找返回子串第一次出现的位置，不存在则返回-1
replace	将某个 pattern 的子串全部替换成另一个子串，n 参数若指定，则替换n次
get	返回某个位置上的字符串
len	返回字符串的长度
ljust	若未达到指定的 `width` 的长度，则在右侧填充 `fillchar` 指定的字符串（默认空格）
rjust	若未达到指定的 `width` 的长度，则在左侧填充 `fillchar` 指定的字符串（默认空格）
lower	变为全部小写
upper	变为全部大写
lstrip	在左侧删除空格（包括空行符）
rstrip	在右侧删除空格（包括空行符）
strip	在左右两侧删除空格（包括空行符）
split	将字符串按分隔符拆分为若干个字符串（返回 list<string> 类型）
pad	在指定的位置（left，right 或者 both）用指定填充字符（用 `fillchar` 指定，默认空格）来对齐
repeat	重复指定 n 次

string 操作	说明
slice	切片操作
swapcase	对调大小写
title	同 str.title
zfill	长度没达到指定 `width` ，则左侧填充0
isalnum	同 str.isalnum
isalpha	同 str.isalpha
isdigit	是否都是数字，同 str.isdigit
isspace	是否都是空格，同 str.isspace
islower	是否都是小写，同 str.islower
isupper	是否都是大写，同 str.isupper
istitle	同 str.istitle
isnumeric	同 str.isnumeric
isdecimal	同 str.isdecimal
todict	将字符串按分隔符拆分为一个 dict，传入的两个参数分别为项目分隔符和 Key-Value 分隔符（返回 dict<string, string> 类型）
strptime	按格式化读取成时间，时间格式和Python标准库相同，详细参考 *Python* 时间格式化

时间相关操作

对于datetime类型Sequence或者Scalar，可以调用时间相关的内置函数。

```
>>> df = lens[[lens.unix_timestamp.astype('datetime').rename('dt')]]
>>> df[df.dt,
>>>    df.dt.year.rename('year'),
>>>    df.dt.month.rename('month'),
>>>    df.dt.day.rename('day'),
>>>    df.dt.hour.rename('hour')].head(5)
                   dt  year  month  day  hour
0  1998-04-08 11:02:00  1998      4    8    11
1  1998-04-08 10:57:55  1998      4    8    10
2  1998-04-08 10:45:26  1998      4    8    10
3  1998-04-08 10:25:52  1998      4    8    10
4  1998-04-08 10:44:19  1998      4    8    10
```

与时间相关的属性包括：

时间相关属性	说明
year	
month	
day	
hour	
minute	
second	
weekofyear	返回日期位于那一年的第几周。周一作为一周的第一天
weekday	返回日期当前周的第几天
dayofweek	同 weekday
strftime	格式化时间，时间格式和 Python 标准库相同，详细参考 *Python* 时间格式化

PyODPS 也支持时间的加减操作，比如可以通过以下方法得到前3天的日期。两个日期列相减得到相差的毫秒数。

```
>>> df
                            a                           b
0 2016-12-06 16:43:12.460001 2016-12-06 17:43:12.460018
1 2016-12-06 16:43:12.460012 2016-12-06 17:43:12.460021
2 2016-12-06 16:43:12.460015 2016-12-06 17:43:12.460022
>>> from odps.df import day
>>> df.a - day(3)
                            a
0 2016-12-03 16:43:12.460001
1 2016-12-03 16:43:12.460012
2 2016-12-03 16:43:12.460015
>>> (df.b - df.a).dtype
int64
>>> (df.b - df.a).rename('a')
         a
0  3600000
1  3600000
2  3600000
```

支持的时间类型包括：

属性	说明
year	
month	
day	

属性	说明
hour	
minute	
second	
millisecond	

集合类型相关操作

PyODPS 支持的集合类型有 List 和 Dict。这两个类型都可以使用下标获取集合中的某个项目，另有 len 方法，可获得集合的大小。

同时，两种集合均有 explode 方法，用于展开集合中的内容。对于 List，explode 默认返回一列，当传入参数 pos 时， 将返回两列，其中一列为值在数组中的编号（类似 Python 的 enumerate 函数）。对于 Dict，explode 会返回两列， 分别表示 keys 及 values。explode 中也可以传入列名，作为最后生成的列。

示例如下：

```
>>> df
   id         a                              b
0   1  [a1, b1]  {'a2': 0, 'b2': 1, 'c2': 2}
1   2      [c1]           {'d2': 3, 'e2': 4}
>>> df[df.id, df.a[0], df.b['b2']]
   id   a    b
0   1  a1    1
1   2  c1  NaN
>>> df[df.id, df.a.len(), df.b.len()]
   id  a  b
0   1  2  3
1   2  1  2
>>> df.a.explode()
    a
0  a1
1  b1
2  c1
>>> df.a.explode(pos=True)
   a_pos   a
0      0  a1
1      1  b1
2      0  c1
>>> # 指定列名
>>> df.a.explode(['pos', 'value'], pos=True)
   pos value
0    0    a1
1    1    b1
2    0    c1
>>> df.b.explode()
  b_key  b_value
0    a2        0
1    b2        1
```

```
2    c2        2
3    d2        3
4    e2        4
>>> # 指定列名
>>> df.b.explode(['key', 'value'])
  key  value
0  a2      0
1  b2      1
2  c2      2
3  d2      3
4  e2      4
```

explode 也可以和并列多行输出 结合，以将原有列和 explode 的结果相结合，例子如下：

```
>>> df[df.id, df.a.explode()]
   id   a
0   1  a1
1   1  b1
2   2  c1
>>> df[df.id, df.a.explode(), df.b.explode()]
   id   a b_key  b_value
0   1  a1    a2        0
1   1  a1    b2        1
2   1  a1    c2        2
3   1  b1    a2        0
4   1  b1    b2        1
5   1  b1    c2        2
6   2  c1    d2        3
7   2  c1    e2        4
```

除了下标、len 和 explode 两个共有方法以外，List 还支持下列方法：

list 操作	说明
contains(v)	列表是否包含某个元素
sort	返回排序后的列表（返回值为 List）

Dict 还支持下列方法：

dict 操作	说明
keys	获取 Dict keys（返回值为 List）
values	获取 Dict values（返回值为 List）

其他元素操作（**isin**，**notin**，**cut**）

`isin`给出Sequence里的元素是否在某个集合元素里。`notin`是相反动作。

```
>>> iris.sepallength.isin([4.9, 5.1]).rename('sepallength').head(5)
   sepallength
0         True
1         True
2        False
3        False
```

```
4        False
```

cut提供离散化的操作，可以将Sequence的数据拆成几个区段。

```
>>> iris.sepallength.cut(range(6), labels=['0-1', '1-2', '2-3', '3-4',
 '4-5']).rename('sepallength_cut').head(5)
   sepallength_cut
0             None
1              4-5
2              4-5
3              4-5
4              4-5
```

include_under和include_over可以分别包括向下和向上的区间。

```
>>> labels = ['0-1', '1-2', '2-3', '3-4', '4-5', '5-']
>>> iris.sepallength.cut(range(6), labels=labels, include_over=True).
rename('sepallength_cut').head(5)
   sepallength_cut
0               5-
1              4-5
2              4-5
3              4-5
4              4-5
```

3.4.8 聚合操作

```
from odps.df import DataFrame
```

```
iris = DataFrame(o.get_table('pyodps_iris'))
```

首先，我们可以使用describe函数，来查看DataFrame里数字列的数量、最大值、最小值、平均值以及标准差是多少。

```
>>> print(iris.describe())
    type  sepal_length  sepal_width  petal_length  petal_width
0  count    150.000000   150.000000    150.000000   150.000000
1   mean      5.843333     3.054000      3.758667     1.198667
2    std      0.828066     0.433594      1.764420     0.763161
3    min      4.300000     2.000000      1.000000     0.100000
4    max      7.900000     4.400000      6.900000     2.500000
```

我们可以使用单列来执行聚合操作：

```
>>> iris.sepallength.max()
7.9
```

如果要在消除重复后的列上进行聚合，可以先调用 unique方法，再调用相应的聚合函数：

```
>>> iris.name.unique().cat(sep=',')
```

```
u'Iris-setosa,Iris-versicolor,Iris-virginica'
```

如果所有列支持同一种聚合操作，也可以直接在整个 DataFrame 上执行聚合操作：

```
>>> iris.exclude('category').mean()
   sepal_length  sepal_width  petal_length  petal_width
1      5.843333     3.054000      3.758667     1.198667
```

需要注意的是，在 DataFrame 上执行 count 获取的是 DataFrame 的总行数：

```
>>> iris.count()
150
```

PyODPS 支持的聚合操作包括：

聚合操作	说明
count（或size）	数量
nunique	不重复值数量
min	最小值
max	最大值
sum	求和
mean	均值
median	中位数
quantile(p)	p分位数，仅在整数值下可取得准确值
var	方差
std	标准差
moment	n 阶中心矩（或 n 阶矩）
skew	样本偏度（无偏估计）
kurtosis	样本峰度（无偏估计）
cat	按sep做字符串连接操作
tolist	组合为 list

需要注意的是，与 Pandas 不同，对于列上的聚合操作，不论是在 ODPS 还是 Pandas 后端下，PyODPS DataFrame 都会忽略空值。这一逻辑与 SQL 类似。

分组聚合

DataFrame API提供了groupby来执行分组操作，分组后的一个主要操作就是通过调用agg或者aggregate方法，来执行聚合操作。

```
>>> iris.groupby('name').agg(iris.sepallength.max(), smin=iris.
sepallength.min())
             name  sepallength_max  smin
0      Iris-setosa              5.8   4.3
1  Iris-versicolor              7.0   4.9
2   Iris-virginica              7.9   4.9
```

最终的结果列中会包含分组的列，以及聚合的列。

DataFrame 提供了一个`value_counts`操作，能返回按某列分组后，每个组的个数从大到小排列的操作。

我们使用 groupby 表达式可以写成：

```
>>> iris.groupby('name').agg(count=iris.name.count()).sort('count',
ascending=False).head(5)
             name  count
0   Iris-virginica     50
1  Iris-versicolor     50
2      Iris-setosa     50
```

使用value_counts就很简单了：

```
>>> iris['name'].value_counts().head(5)
             name  count
0   Iris-virginica     50
1  Iris-versicolor     50
2      Iris-setosa     50
```

对于聚合后的单列操作，我们也可以直接取出列名。但此时只能使用聚合函数。

```
>>> iris.groupby('name').petallength.sum()
   petallength_sum
0            73.2
1           213.0
2           277.6
```

```
>>> iris.groupby('name').agg(iris.petallength.notnull().sum())
             name  petallength_sum
0      Iris-setosa               50
1  Iris-versicolor               50
2   Iris-virginica               50
```

分组时也支持对常量进行分组，但是需要使用Scalar初始化。

```
>>> from odps.df import Scalar
>>> iris.groupby(Scalar(1)).petallength.sum()
   petallength_sum
```

```
0            563.8
```

编写自定义聚合

对字段调用agg或者aggregate方法来调用自定义聚合。自定义聚合需要提供一个类，这个类需要提供以下方法：

- buffer()：返回一个mutable的object（比如 list、dict），buffer大小不应随数据而递增。
- __call__(buffer, *val)：将值聚合到中间 buffer。
- merge(buffer, pbuffer)：将 pbuffer 聚合到 buffer 中。
- getvalue(buffer)：返回最终值。

让我们看一个计算平均值的例子。

```
class Agg(object):

    def buffer(self):
        return [0.0, 0]

    def __call__(self, buffer, val):
        buffer[0] += val
        buffer[1] += 1

    def merge(self, buffer, pbuffer):
        buffer[0] += pbuffer[0]
        buffer[1] += pbuffer[1]

    def getvalue(self, buffer):
        if buffer[1] == 0:
            return 0.0
        return buffer[0] / buffer[1]
```

```
>>> iris.sepalwidth.agg(Agg)
3.0540000000000007
```

如果最终类型和输入类型发生了变化，则需要指定类型。

```
>>> iris.sepalwidth.agg(Agg, 'float')
```

自定义聚合也可以用在分组聚合中。

```
>>> iris.groupby('name').sepalwidth.agg(Agg)
   petallength_aggregation
0                    3.418
1                    2.770
2                    2.974
```

当对多列调用自定义聚合，可以使用agg方法。

```
class Agg(object):

    def buffer(self):
```

```
        return [0.0, 0.0]

    def __call__(self, buffer, val1, val2):
        buffer[0] += val1
        buffer[1] += val2

    def merge(self, buffer, pbuffer):
        buffer[0] += pbuffer[0]
        buffer[1] += pbuffer[1]

    def getvalue(self, buffer):
        if buffer[1] == 0:
            return 0.0
        return buffer[0] / buffer[1]r[0] / buffer[1]
```

```
>>> from odps.df import agg
>>> to_agg = agg([iris.sepalwidth, iris.sepallength], Agg, rtype='
float')  # 对两列调用自定义聚合
>>> iris.groupby('name').agg(val=to_agg)
              name       val
0      Iris-setosa  0.682781
1  Iris-versicolor  0.466644
2   Iris-virginica  0.451427
```

要调用 ODPS 上已经存在的 UDAF，指定函数名即可。

```
>>> iris.groupby('name').agg(iris.sepalwidth.agg('your_func'))  # 对单
列聚合
>>> to_agg = agg([iris.sepalwidth, iris.sepallength], 'your_func',
rtype='float')
>>> iris.groupby('name').agg(to_agg.rename('val'))  # 对多列聚合
```

说明：

目前，受限于 Python UDF，自定义聚合无法支持将 list / dict 类型作为初始输入或最终输出结果。

HyperLogLog 计数

DataFrame 提供了对列进行 HyperLogLog 计数的接口 `hll_count`，这个接口是个近似的估计接口， 当数据量很大时，能较快的对数据的唯一个数进行估计。

这个接口在对比如海量用户UV进行计算时，能很快得出估计值。

```
>>> df = DataFrame(pd.DataFrame({'a': np.random.randint(100000, size=
100000)}))
>>> df.a.hll_count()
63270
>>> df.a.nunique()
63250
```

提供 `splitter`参数会对每个字段进行分隔，再计算唯一数。

3.4.9 排序、去重、采样、数据变换

```
from odps.df import DataFrame
```

```
iris = DataFrame(o.get_table('pyodps_iris'))
```

排序

排序操作只能作用于Collection。我们只需要调用sort或者sort_values方法。

```
>>> iris.sort('sepalwidth').head(5)
   sepallength  sepalwidth  petallength  petalwidth            name
0          5.0         2.0          3.5         1.0  Iris-versicolor
1          6.2         2.2          4.5         1.5  Iris-versicolor
2          6.0         2.2          5.0         1.5   Iris-virginica
3          6.0         2.2          4.0         1.0  Iris-versicolor
4          5.5         2.3          4.0         1.3  Iris-versicolor
```

如果想要降序排列，则可以使用参数ascending，并设为False。

```
>>> iris.sort('sepalwidth', ascending=False).head(5)
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.7         4.4          1.5         0.4  Iris-setosa
1          5.5         4.2          1.4         0.2  Iris-setosa
2          5.2         4.1          1.5         0.1  Iris-setosa
3          5.8         4.0          1.2         0.2  Iris-setosa
4          5.4         3.9          1.3         0.4  Iris-setosa
```

也可以这样调用，来进行降序排列：

```
>>> iris.sort(-iris.sepalwidth).head(5)
   sepallength  sepalwidth  petallength  petalwidth         name
0          5.7         4.4          1.5         0.4  Iris-setosa
1          5.5         4.2          1.4         0.2  Iris-setosa
2          5.2         4.1          1.5         0.1  Iris-setosa
3          5.8         4.0          1.2         0.2  Iris-setosa
4          5.4         3.9          1.3         0.4  Iris-setosa
```

多字段排序也很简单：

```
>>> iris.sort(['sepalwidth', 'petallength']).head(5)
   sepallength  sepalwidth  petallength  petalwidth            name
0          5.0         2.0          3.5         1.0  Iris-versicolor
1          6.0         2.2          4.0         1.0  Iris-versicolor
2          6.2         2.2          4.5         1.5  Iris-versicolor
3          6.0         2.2          5.0         1.5   Iris-virginica
4          4.5         2.3          1.3         0.3      Iris-setosa
```

多字段排序时，如果是升序降序不同，ascending参数可以传入一个列表，长度必须等同于排序的字段，它们的值都是boolean类型

```
>>> iris.sort(['sepalwidth', 'petallength'], ascending=[True, False]).
head(5)
```

```
   sepallength  sepalwidth  petallength  petalwidth            name
0          5.0         2.0          3.5         1.0  Iris-versicolor
1          6.0         2.2          5.0         1.5   Iris-virginica
2          6.2         2.2          4.5         1.5  Iris-versicolor
3          6.0         2.2          4.0         1.0  Iris-versicolor
4          6.3         2.3          4.4         1.3  Iris-versicolor
```

下面效果是一样的：

```
>>> iris.sort(['sepalwidth', -iris.petallength]).head(5)
   sepallength  sepalwidth  petallength  petalwidth            name
0          5.0         2.0          3.5         1.0  Iris-versicolor
1          6.0         2.2          5.0         1.5   Iris-virginica
2          6.2         2.2          4.5         1.5  Iris-versicolor
3          6.0         2.2          4.0         1.0  Iris-versicolor
4          6.3         2.3          4.4         1.3  Iris-versicolor
```

说明：

由于 ODPS 要求排序必须指定个数，所以在 ODPS 后端执行时， 会通过 `options.df.odps.sort.limit` 指定排序个数，这个值默认是 10000， 如果要排序尽量多的数据，可以把这个值设到较大的值。不过注意，此时可能会导致 OOM。

去重

去重在Collection上，用户可以调用distinct方法。

```
>>> iris[['name']].distinct()
             name
0     Iris-setosa
1 Iris-versicolor
2  Iris-virginica
```

```
>>> iris.distinct('name')
             name
0     Iris-setosa
1 Iris-versicolor
2  Iris-virginica
```

```
>>> iris.distinct('name', 'sepallength').head(3)
          name  sepallength
0 Iris-setosa          4.3
1 Iris-setosa          4.4
2 Iris-setosa          4.5
```

在Sequence上，用户可以调用unique，但是记住，调用unique的Sequence不能用在列选择中。

```
>>> iris.name.unique()
             name
0     Iris-setosa
1 Iris-versicolor
```

```
2   Iris-virginica
```

下面的代码是错误的用法。

```
>>> iris[iris.name, iris.name.unique()]  # 错误的
```

采样

要对一个 collection 的数据采样，可以调用 `sample` 方法。PyODPS 支持四种采样方式。

说明：

除了按份数采样外，其余方法如果要在 ODPS DataFrame 上执行，需要 Project 支持 XFlow，否则，这些方法只能在 Pandas DataFrame 后端上执行。

- 按份数采样

在这种采样方式下，数据被分为 `parts` 份，可选择选取的份数序号。

```
>>> iris.sample(parts=10)  # 分成10份，默认取第0份
>>> iris.sample(parts=10, i=0)  # 手动指定取第0份
>>> iris.sample(parts=10, i=[2, 5])   # 分成10份，取第2和第5份
>>> iris.sample(parts=10, columns=['name', 'sepalwidth'])  # 根据name和
sepalwidth的值做采样
```

- 按比例 / 条数采样

在这种采样方式下，用户指定需要采样的数据条数或采样比例。指定 `replace` 参数为 True 可启用放回采样。

```
>>> iris.sample(n=100)  # 选取100条数据
>>> iris.sample(frac=0.3)  # 采样30%的数据
```

- 按权重列采样

在这种采样方式下，用户指定权重列和数据条数 / 采样比例。指定 `replace` 参数为 True 可启用放回采样。

```
>>> iris.sample(n=100, weights='sepal_length')
>>> iris.sample(n=100, weights='sepal_width', replace=True)
```

- 分层采样

在这种采样方式下，用户指定用于分层的标签列，同时为需要采样的每个标签指定采样比例（`frac` 参数）或条数（`n` 参数）。暂不支持放回采样。

```
>>> iris.sample(strata='category', n={'Iris Setosa': 10, 'Iris
Versicolour': 10})
>>> iris.sample(strata='category', frac={'Iris Setosa': 0.5, 'Iris
Versicolour': 0.4})
```

数据缩放

DataFrame 支持通过最大/最小值或平均值/标准差对数据进行缩放。例如，对数据

```
name  id  fid
0  name1   4  5.3
1  name2   2  3.5
2  name2   3  1.5
3  name1   4  4.2
4  name1   3  2.2
5  name1   3  4.1
```

使用 min_max_scale 方法进行归一化：

```
>>> df.min_max_scale(columns=['fid'])
    name  id       fid
0  name1   4  1.000000
1  name2   2  0.526316
2  name2   3  0.000000
3  name1   4  0.710526
4  name1   3  0.184211
5  name1   3  0.684211
```

min_max_scale 还支持使用 feature_range 参数指定输出值的范围，例如，如果我们需要使输出值在 (-1, 1) 范围内，可使用

```
>>> df.min_max_scale(columns=['fid'], feature_range=(-1, 1))
    name  id       fid
0  name1   4  1.000000
1  name2   2  0.052632
2  name2   3 -1.000000
3  name1   4  0.421053
4  name1   3 -0.631579
5  name1   3  0.368421
```

如果需要保留原始值，可以使用 preserve 参数。此时，缩放后的数据将会以新增列的形式追加到数据中， 列名默认为原列名追加“_scaled”后缀，该后缀可使用 suffix 参数更改。例如，

```
>>> df.min_max_scale(columns=['fid'], preserve=True)
    name  id  fid  fid_scaled
0  name1   4  5.3    1.000000
1  name2   2  3.5    0.526316
2  name2   3  1.5    0.000000
3  name1   4  4.2    0.710526
4  name1   3  2.2    0.184211
```

```
5  name1   3  4.1    0.684211
```

min_max_scale 也支持使用 group 参数指定一个或多个分组列，在分组列中分别取最值进行缩放。例如，

```
>>> df.min_max_scale(columns=['fid'], group=['name'])
    name  id       fid
0  name1   4  1.000000
1  name1   4  0.645161
2  name1   3  0.000000
3  name1   3  0.612903
4  name2   2  1.000000
5  name2   3  0.000000
```

可见结果中，name1 和 name2 两组均按组中的最值进行了缩放。

std_scale 可依照标准正态分布对数据进行调整。例如，

```
>>> df.std_scale(columns=['fid'])
    name  id       fid
0  name1   4  1.436467
1  name2   2  0.026118
2  name2   3 -1.540938
3  name1   4  0.574587
4  name1   3 -0.992468
5  name1   3  0.496234
```

std_scale 同样支持 preserve 参数保留原始列以及使用 group 进行分组，具体请参考 min_max_scale，此处不再赘述。

空值处理

DataFrame 支持筛去空值以及填充空值的功能。例如，对数据

```
id   name   f1   f2   f3   f4
0   0  name1  1.0  NaN  3.0  4.0
1   1  name1  2.0  NaN  NaN  1.0
2   2  name1  3.0  4.0  1.0  NaN
3   3  name1  NaN  1.0  2.0  3.0
4   4  name1  1.0  NaN  3.0  4.0
5   5  name1  1.0  2.0  3.0  4.0
6   6  name1  NaN  NaN  NaN  NaN
```

使用 dropna 可删除 subset 中包含空值的行：

```
>>> df.dropna(subset=['f1', 'f2', 'f3', 'f4'])
   id   name   f1   f2   f3   f4
0   5  name1  1.0  2.0  3.0  4.0
```

如果行中包含非空值则不删除，可以使用 how='all'：

```
>>> df.dropna(how='all', subset=['f1', 'f2', 'f3', 'f4'])
   id   name   f1   f2   f3   f4
```

```
0   0  name1  1.0  NaN  3.0  4.0
1   1  name1  2.0  NaN  NaN  1.0
2   2  name1  3.0  4.0  1.0  NaN
3   3  name1  NaN  1.0  2.0  3.0
4   4  name1  1.0  NaN  3.0  4.0
5   5  name1  1.0  2.0  3.0  4.0
```

你也可以使用 thresh 参数来指定行中至少要有多少个非空值。例如：

```
>>> df.dropna(thresh=3, subset=['f1', 'f2', 'f3', 'f4'])
   id   name   f1   f2   f3   f4
0   0  name1  1.0  NaN  3.0  4.0
2   2  name1  3.0  4.0  1.0  NaN
3   3  name1  NaN  1.0  2.0  3.0
4   4  name1  1.0  NaN  3.0  4.0
5   5  name1  1.0  2.0  3.0  4.0
```

使用 fillna 可使用常数或已有的列填充未知值。下面给出了使用常数填充的例子：

```
>>> df.fillna(100, subset=['f1', 'f2', 'f3', 'f4'])
   id   name     f1     f2     f3     f4
0   0  name1    1.0  100.0    3.0    4.0
1   1  name1    2.0  100.0  100.0    1.0
2   2  name1    3.0    4.0    1.0  100.0
3   3  name1  100.0    1.0    2.0    3.0
4   4  name1    1.0  100.0    3.0    4.0
5   5  name1    1.0    2.0    3.0    4.0
6   6  name1  100.0  100.0  100.0  100.0
```

你也可以使用一个已有的列来填充未知值。例如：

```
>>> df.fillna(df.f2, subset=['f1', 'f2', 'f3', 'f4'])
   id   name   f1   f2   f3   f4
0   0  name1  1.0  NaN  3.0  4.0
1   1  name1  2.0  NaN  NaN  1.0
2   2  name1  3.0  4.0  1.0  4.0
3   3  name1  1.0  1.0  2.0  3.0
4   4  name1  1.0  NaN  3.0  4.0
5   5  name1  1.0  2.0  3.0  4.0
6   6  name1  NaN  NaN  NaN  NaN
```

特别地，DataFrame 提供了向前 / 向后填充的功能。通过指定 method 参数为下列值可以达到目的：

取值	含义
bfill / backfill	向前填充
ffill / pad	向后填充

例如：

```
>>> df.fillna(method='bfill', subset=['f1', 'f2', 'f3', 'f4'])
   id   name   f1   f2   f3   f4
0   0  name1  1.0  3.0  3.0  4.0
```

```
1   1  name1  2.0  1.0  1.0  1.0
2   2  name1  3.0  4.0  1.0  NaN
3   3  name1  1.0  1.0  2.0  3.0
4   4  name1  1.0  3.0  3.0  4.0
5   5  name1  1.0  2.0  3.0  4.0
6   6  name1  NaN  NaN  NaN  NaN
>>> df.fillna(method='ffill', subset=['f1', 'f2', 'f3', 'f4'])
   id   name   f1   f2   f3   f4
0   0  name1  1.0  1.0  3.0  4.0
1   1  name1  2.0  2.0  2.0  1.0
2   2  name1  3.0  4.0  1.0  1.0
3   3  name1  NaN  1.0  2.0  3.0
4   4  name1  1.0  1.0  3.0  4.0
5   5  name1  1.0  2.0  3.0  4.0
6   6  name1  NaN  NaN  NaN  NaN
```

你也可以使用 ffill / bfill 函数来简化代码。ffill 等价于 fillna(method='ffill')， bfill 等价于 fillna(method='bfill')

3.4.10 对所有行/列调用自定义函数

要对一行数据使用自定义函数，可以使用 apply 方法，axis 参数必须为 1，表示在行上操作。

对一行数据使用自定义函数

apply 的自定义函数接收一个参数，为上一步 Collection 的一行数据，用户可以通过属性、或者偏移取得一个字段的数据。

```
>>> iris.apply(lambda row: row.sepallength + row.sepalwidth, axis=1,
reduce=True, types='float').rename('sepaladd').head(3)
   sepaladd
0       8.6
1       7.9
2       7.9
```

`reduce`为 True 时，表示返回结果为Sequence，否则返回结果为Collection。 `names`和 `types`参数分别指定返回的Sequence或Collection的字段名和类型。 如果类型不指定，将会默认为string类型。

在 apply 的自定义函数中，reduce 为 False 时，也可以使用 `yield`关键字来返回多行结果。

```
>>> iris.count()
150
>>>
>>> def handle(row):
>>>     yield row.sepallength - row.sepalwidth, row.sepallength + row.
sepalwidth
>>>     yield row.petallength - row.petalwidth, row.petallength + row.
petalwidth
>>>
>>> iris.apply(handle, axis=1, names=['iris_add', 'iris_sub'], types
=['float', 'float']).count()
```

```
300
```

我们也可以在函数上来注释返回的字段和类型，这样就不需要在函数调用时再指定。

```
>>> from odps.df import output
>>>
>>> @output(['iris_add', 'iris_sub'], ['float', 'float'])
>>> def handle(row):
>>>     yield row.sepallength - row.sepalwidth, row.sepallength + row.
sepalwidth
>>>     yield row.petallength - row.petalwidth, row.petallength + row.
petalwidth
>>>
>>> iris.apply(handle, axis=1).count()
300
```

也可以使用 map-only 的 map_reduce，和 axis=1 的apply操作是等价的。

```
>>> iris.map_reduce(mapper=handle).count()
300
```

如果想调用 ODPS 上已经存在的 UDTF，则函数指定为函数名即可。

```
>>> iris['name', 'sepallength'].apply('your_func', axis=1, names=['
name2', 'sepallength2'], types=['string', 'float'])
```

使用 apply 对行操作，且 reduce为 False 时，可以使用 使用自定义函数 与已有的行结合，用于后续聚合等操作。

```
>>> from odps.df import output
>>>
>>> @output(['iris_add', 'iris_sub'], ['float', 'float'])
>>> def handle(row):
>>>     yield row.sepallength - row.sepalwidth, row.sepallength + row.
sepalwidth
>>>     yield row.petallength - row.petalwidth, row.petallength + row.
petalwidth
>>>
>>> iris[iris.category, iris.apply(handle, axis=1)]
```

对所有列调用自定义聚合

调用apply方法，当我们不指定axis，或者axis为0的时候，我们可以通过传入一个自定义聚合类来对所有sequence进行聚合操作。

```
class Agg(object):

    def buffer(self):
        return [0.0, 0]

    def __call__(self, buffer, val):
        buffer[0] += val
        buffer[1] += 1
```

```
    def merge(self, buffer, pbuffer):
        buffer[0] += pbuffer[0]
        buffer[1] += pbuffer[1]

    def getvalue(self, buffer):
        if buffer[1] == 0:
            return 0.0
        return buffer[0] / buffer[1]
```

```
>>> iris.exclude('name').apply(Agg)
   sepallength_aggregation  sepalwidth_aggregation  petallengt
h_aggregation  petalwidth_aggregation
0                 5.843333                   3.054                3.
758667               1.198667
```

说明：

目前，受限于 Python UDF，自定义函数无法支持将 list / dict 类型作为初始输入或最终输出结果。

引用资源

类似于对*map* 方法的resources参数，每个resource可以是ODPS上的资源（表资源或文件资源），或者引用一个collection作为资源。

对于axis为1，也就是在行上操作，我们需要写一个函数闭包或者callable的类。 而对于列上的聚合操作，我们只需在 __init__ 函数里读取资源即可。

```
>>> words_df
                    sentence
0                Hello World
1               Hello Python
2  Life is short I use Python
>>>
>>> import pandas as pd
>>> stop_words = DataFrame(pd.DataFrame({'stops': ['is', 'a', 'I']}))
>>>
>>> @output(['sentence'], ['string'])
>>> def filter_stops(resources):
>>>     stop_words = set([r[0] for r in resources[0]])
>>>     def h(row):
>>>         return ' '.join(w for w in row[0].split() if w not in
stop_words),
>>>     return h
>>>
>>> words_df.apply(filter_stops, axis=1, resources=[stop_words])
                sentence
0            Hello World
1           Hello Python
2  Life short use Python
```

可以看到这里的stop_words是存放于本地，但在真正执行时会被上传到ODPS作为资源引用。

使用第三方Python库

使用方法类似*map中使用第三方Python库* 。

可以在全局指定使用的库：

```
>>> from odps import options
>>> options.df.libraries = ['six.whl', 'python_dateutil.whl']
```

或者在立即执行的方法中，局部指定：

```
>>> df.apply(my_func, axis=1).to_pandas(libraries=['six.whl', '
python_dateutil.whl'])
```

说明：

由于字节码定义的差异，Python 3 下使用新语言特性（例如 `yield from`）时，代码在使用 Python 2.7 的 ODPS Worker 上执行时会发生错误。因而建议在 Python 3 下使用 MapReduce API 编写生产作业前，先确认相关代码是否能正常 执行。

3.4.11 使用自定义函数

DataFrame函数支持对Sequence使用map，它会对它的每个元素调用自定义函数。

DataFrame函数：

```
>>> iris.sepallength.map(lambda x: x + 1).head(5)
   sepallength
0          6.1
1          5.9
2          5.7
3          5.6
4          6.0
```

说明：

目前，受限于 Python UDF，自定义函数无法支持将 list / dict 类型作为输入或输出。

如果map前后，Sequence的类型发生了变化，则需要显式指定map后的类型。

```
>>> iris.sepallength.map(lambda x: 't'+str(x), 'string').head(5)
   sepallength
0         t5.1
1         t4.9
2         t4.7
3         t4.6
```

```
4          t5.0
```

如果在函数中包含闭包，需要注意的是，函数外闭包变量值的变化会引起函数内该变量值的变化。例如，

```
>>> dfs = []
>>> for i in range(10):
>>>     dfs.append(df.sepal_length.map(lambda x: x + i))
```

结果为 dfs 中每个 SequenceExpr 均为 `df.sepal_length + 9`。为解决此问题，可以将函数作为另一函数的返回值，或者使用 partial，如

```
>>> dfs = []
>>> def get_mapper(i):
>>>     return lambda x: x + i
>>> for i in range(10):
>>>     dfs.append(df.sepal_length.map(get_mapper(i)))
```

或

```
>>> import functools
>>> dfs = []
>>> for i in range(10):
>>>     dfs.append(df.sepal_length.map(functools.partial(lambda v, x:
x + v, i)))
```

map也支持使用现有的UDF函数，传入的参数是str类型（函数名）或者 *Function*对象 。

map传入Python函数的实现使用了ODPS Python UDF，因此，如果用户所在的Project不支持Python UDF，则map函数无法使用。除此以外，所有 Python UDF 的限制在此都适用。

目前，第三方库（包含C）只能使用`numpy`，第三方库使用参考 使用第三方*Python*库。

除了调用自定义函数，DataFrame还提供了很多内置函数，这些函数中部分使用了map函数来实现，因此，如果用户所在Project未开通Python UDF，则这些函数也就无法使用（注：阿里云公共服务暂不提供Python UDF支持）。

说明：

由于字节码定义的差异，Python 3 下使用新语言特性（例如 `yield from`）时，代码在使用 Python 2.7 的 ODPS Worker 上执行时会发生错误。因而建议在 Python 3 下使用 MapReduce API 编写生产作业前，先确认相关代码是否能正常 执行。

引用资源

自定义函数也能读取ODPS上的资源（表资源或文件资源），或者引用一个collection作为资源。 此时，自定义函数需要写成函数闭包或callable的类。

```
>>> file_resource = o.create_resource('pyodps_iris_file', 'file',
file_obj='Iris-setosa')
>>>
>>> iris_names_collection = iris.distinct('name')[:2]
>>> iris_names_collection
       sepallength
0      Iris-setosa
1  Iris-versicolor
```

```
>>> def myfunc(resources):  # resources按调用顺序传入
>>>     names = set()
>>>     fileobj = resources[0] # 文件资源是一个file-like的object
>>>     for l in fileobj:
>>>         names.add(l)
>>>     collection = resources[1]
>>>     for r in collection:
>>>         names.add(r.name)  # 这里可以通过字段名或者偏移来取
>>>     def h(x):
>>>         if x in names:
>>>             return True
>>>         else:
>>>             return False
>>>     return h
>>>
>>> df = iris.distinct('name')
>>> df = df[df.name,
>>>         df.name.map(myfunc, resources=[file_resource, iris_names
_collection], rtype='boolean').rename('isin')]
>>>
>>> df
              name   isin
0      Iris-setosa   True
1  Iris-versicolor   True
2   Iris-virginica  False
```

说明：

分区表资源在读取时不包含分区字段。

使用第三方Python库

现在我们有个DataFrame，只有一个string类型字段。

```
>>> df
               datestr
0  2016-08-26 14:03:29
```

```
1  2015-08-26 14:03:29
```

全局配置使用到的三方库：

```
>>> from odps import options
>>>
>>> def get_year(t):
>>>     from dateutil.parser import parse
>>>     return parse(t).strftime('%Y')
>>>
>>> options.df.libraries = ['six.whl', 'python_dateutil.whl']
>>> df.datestr.map(get_year)
   datestr
0     2016
1     2015
```

或者，通过立即运行方法的 `libraries`参数指定：

```
>>> def get_year(t):
>>>     from dateutil.parser import parse
>>>     return parse(t).strftime('%Y')
>>>
>>> df.datestr.map(get_year).execute(libraries=['six.whl', 'python_dat
eutil.whl'])
   datestr
0     2016
1     2015
```

PyODPS 默认支持执行纯 Python 且不含文件操作的第三方库。在较新版本的 MaxCompute 服务下，PyODPS 也支持执行带有二进制代码或带有文件操作的 Python 库。这些库的后缀必须是 cp27-cp27m-manylinux1_x86_64， 以 archive 格式上传，whl 后缀的包需要重命名为 zip。同时，作业需要开启 `odps.isolation.session.enable` 选项，或者在 Project 级别开启 Isolation。下面的例子展示了如何上传并使用 scipy 中的特殊函数：

```
>>> # 对于含有二进制代码的包，必须使用 Archive 方式上传资源，whl 后缀需要改为
zip
>>> odps.create_resource('scipy.zip', 'archive', file_obj=open('scipy-
0.19.0-cp27-cp27m-manylinux1_x86_64.whl', 'rb'))
>>>
>>> # 如果 Project 开启了 Isolation，下面的选项不是必需的
>>> options.sql.settings = { 'odps.isolation.session.enable': True }
>>>
>>> def psi(value):
>>>     # 建议在函数内部 import 第三方库，以防止不同操作系统下二进制包结构差异造
成执行错误
>>>     from scipy.special import psi
>>>     return float(psi(value))
>>>
```

```
>>> df.float_col.map(psi).execute(libraries=['scipy.zip'])
```

对于只提供源码的二进制包，可以在 Linux Shell 中打包成 Wheel 再上传，Mac 和 Windows 中生成的 Wheel 包无法在 MaxCompute 中使用：

```
python setup.py bdist_wheel
```

使用计数器

```
from odps.udf import get_execution_context

def h(x):
    ctx = get_execution_context()
    counters = ctx.get_counters()
    counters.get_counter('df', 'add_one').increment(1)
    return x + 1

df.field.map(h)
```

logview 的 JSONSummary 中即可找到计数器值。

调用**ODPS**内建或者已定义函数

要想调用 ODPS 上的内建或者已定义函数，来生成列，我们可以使用 `func`接口，该接口默认函数返回值为 String， 可以用 rtype 参数指定返回值。

```
>>> from odps.df import func
>>>
>>> iris[iris.name, func.rand(rtype='float').rename('rand')][:4]
>>> iris[iris.name, func.rand(10, rtype='float').rename('rand')][:4]
>>> # 调用 ODPS 上定义的 UDF，列名无法确定时需要手动指定
>>> iris[iris.name, func.your_udf(iris.sepalwidth, iris.sepallength,
rtype='float').rename('new_col')]
>>> # 从其他 Project 调用 UDF，也可通过 name 参数指定列名
>>> iris[iris.name, func.your_udf(iris.sepalwidth, iris.sepallength,
rtype='float', project='udf_project',
>>>                               name='new_col')]
```

 说明：

注意：在使用 Pandas 后端时，不支持执行带有 `func`的表达式。

3.4.12 MapReduce API

PyODPS DataFrame也支持MapReduce API，用户可以分别编写map和reduce函数（map_reduce可以只有mapper或者reducer过程）。

首先我们来看个简单的wordcount的例子。

```
>>> def mapper(row):
>>>     for word in row[0].split():
```

```
>>>         yield word.lower(), 1
>>>
>>> def reducer(keys):
>>>     cnt = [0]
>>>     def h(row, done):  # done表示这个key已经迭代结束
>>>         cnt[0] += row[1]
>>>         if done:
>>>             yield keys[0], cnt[0]
>>>     return h
>>>
>>> words_df.map_reduce(mapper, reducer, group=['word', ],
>>>                     mapper_output_names=['word', 'cnt'],
>>>                     mapper_output_types=['string', 'int'],
>>>                     reducer_output_names=['word', 'cnt'],
>>>                     reducer_output_types=['string', 'int'])
     word  cnt
0   hello    2
1       i    1
2      is    1
3    life    1
4  python    2
5   short    1
6     use    1
7   world    1
```

group参数用来指定reduce按哪些字段做分组，如果不指定，会按全部字段做分组。

其中对于reducer来说，会稍微有些不同。它需要接收聚合的keys初始化，并能继续处理按这些keys聚合的每行数据。 第2个参数表示这些keys相关的所有行是不是都迭代完成。

这里写成函数闭包的方式，主要为了方便，当然我们也能写成callable的类。

```
class reducer(object):
    def __init__(self, keys):
        self.cnt = 0

    def __call__(self, row, done):  # done表示这个key已经迭代结束
        self.cnt += row.cnt
        if done:
            yield row.word, self.cnt
```

使用 output来注释会让代码更简单些。

```
>>> from odps.df import output
>>>
>>> @output(['word', 'cnt'], ['string', 'int'])
>>> def mapper(row):
>>>     for word in row[0].split():
>>>         yield word.lower(), 1
>>>
>>> @output(['word', 'cnt'], ['string', 'int'])
>>> def reducer(keys):
>>>     cnt = [0]
>>>     def h(row, done):  # done表示这个key已经迭代结束
>>>         cnt[0] += row.cnt
>>>         if done:
>>>             yield keys.word, cnt[0]
>>>     return h
```

```
>>>
>>> words_df.map_reduce(mapper, reducer, group='word')
     word  cnt
0   hello    2
1       i    1
2      is    1
3    life    1
4  python    2
5   short    1
6     use    1
7   world    1
```

有时候我们在迭代的时候需要按某些列排序，则可以使用 `sort`参数，来指定按哪些列排序，升序降序则通过 `ascending`参数指定。 `ascending` 参数可以是一个bool值，表示所有的 `sort`字段是相同升序或降序， 也可以是一个列表，长度必须和 `sort`字段长度相同。

指定combiner

combiner表示在map_reduce API里表示在mapper端，就先对数据进行聚合操作，它的用法和reducer是完全一致的，但不能引用资源。 并且，combiner的输出的字段名和字段类型必须和mapper完全一致。

上面的例子，我们就可以使用reducer作为combiner来先在mapper端对数据做初步的聚合，减少shuffle出去的数据量。

```
>>> words_df.map_reduce(mapper, reducer, combiner=reducer, group='word
')
```

引用资源

在MapReduce API里，我们能分别指定mapper和reducer所要引用的资源。

如下面的例子，我们对mapper里的单词做停词过滤，在reducer里对白名单的单词数量加5。

```
>>> white_list_file = o.create_resource('pyodps_white_list_words', '
file', file_obj='Python\nWorld')
>>>
>>> @output(['word', 'cnt'], ['string', 'int'])
>>> def mapper(resources):
>>>     stop_words = set(r[0].strip() for r in resources[0])
>>>     def h(row):
>>>         for word in row[0].split():
>>>             if word not in stop_words:
>>>                 yield word, 1
>>>     return h
>>>
>>> @output(['word', 'cnt'], ['string', 'int'])
>>> def reducer(resources):
>>>     d = dict()
>>>     d['white_list'] = set(word.strip() for word in resources[0])
>>>     d['cnt'] = 0
>>>     def inner(keys):
>>>         d['cnt'] = 0
```

```
>>>         def h(row, done):
>>>             d['cnt'] += row.cnt
>>>             if done:
>>>                 if row.word in d['white_list']:
>>>                     d['cnt'] += 5
>>>                 yield keys.word, d['cnt']
>>>         return h
>>>     return inner
>>>
>>> words_df.map_reduce(mapper, reducer, group='word',
>>>                     mapper_resources=[stop_words], reducer_re
sources=[white_list_file])
     word  cnt
0   hello    2
1    life    1
2  python    7
3   world    6
4   short    1
5     use    1
```

使用第三方**Python**库

使用方法类似 *map*中使用第三方*Python*库 。

可以在全局指定使用的库：

```
>>> from odps import options
>>> options.df.libraries = ['six.whl', 'python_dateutil.whl']
```

或者在立即执行的方法中，局部指定：

```
>>> df.map_reduce(mapper=my_mapper, reducer=my_reducer, group='key').
execute(libraries=['six.whl', 'python_dateutil.whl'])
```

说明：

由于字节码定义的差异，Python 3 下使用新语言特性（例如 `yield from`）时，代码在使用 Python 2.7 的 ODPS Worker 上执行时会发生错误。因而建议在 Python 3 下使用 MapReduce API 编写生产作业前，先确认相关代码是否能正常 执行。

重排数据

有时候我们的数据在集群上分布可能是不均匀的，我们需要对数据重排。调用 `reshuffle`接口即可。

```
>>> df1 = df.reshuffle()
```

默认会按随机数做哈希来分布。也可以指定按那些列做分布，且可以指定重排后的排序顺序。

```
>>> df1.reshuffle('name', sort='id', ascending=False)
```

布隆过滤器

PyODPS DataFrame提供了 `bloom_filter` 接口来进行布隆过滤器的计算。

给定某个collection，和它的某个列计算的sequence1，我们能对另外一个sequence2进行布隆过滤，sequence1不在sequence2中的一定会过滤， 但可能不能完全过滤掉不存在于sequence2中的数据，这也是一种近似的方法。

这样的好处是能快速对collection进行快速过滤一些无用数据。

这在大规模join的时候，一边数据量远大过另一边数据，而大部分并不会join上的场景很有用。 比如，我们在join用户的浏览数据和交易数据时，用户的浏览大部分不会带来交易，我们可以利用交易数据先对浏览数据进行布隆过滤， 然后再join能很好提升性能。

```
>>> df1 = DataFrame(pd.DataFrame({'a': ['name1', 'name2', 'name3', '
name1'], 'b': [1, 2, 3, 4]}))
>>> df1
       a  b
0  name1  1
1  name2  2
2  name3  3
3  name1  4
>>> df2 = DataFrame(pd.DataFrame({'a': ['name1']}))
>>> df2
       a
0  name1
>>> df1.bloom_filter('a', df2.a) # 这里第0个参数可以是个计算表达式如: df1.a
 + '1'
       a  b
0  name1  1
1  name1  4
```

这里由于数据量很小，df1中的a为name2和name3的行都被正确过滤掉了，当数据量很大的时候，可能会有一定的数据不能被过滤。

如之前提的join场景中，少量不能过滤并不能并不会影响正确性，但能较大提升join的性能。

我们可以传入 `capacity` 和 `error_rate` 来设置数据的量以及错误率，默认值是 `3000` 和 `0.01` 。

说明：

要注意，调大 `capacity` 或者减小 `error_rate`会增加内存的使用，所以应当根据实际情况选择一个合理的值。

透视表（**pivot_table**）

PyODPS DataFrame提供透视表的功能。我们通过几个例子来看使用。

```
>>> df
     A    B      C  D  E
0  foo  one  small  1  3
1  foo  one  large  2  4
2  foo  one  large  2  5
3  foo  two  small  3  6
4  foo  two  small  3  4
5  bar  one  large  4  5
6  bar  one  small  5  3
7  bar  two  small  6  2
8  bar  two  large  7  1
```

最简单的透视表必须提供一个`rows` 参数，表示按一个或者多个字段做取平均值的操作。

```
>>> df['A', 'D', 'E'].pivot_table(rows='A')
     A  D_mean  E_mean
0  bar     5.5    2.75
1  foo     2.2    4.40
```

rows可以提供多个，表示按多个字段做聚合。

```
>>> df.pivot_table(rows=['A', 'B', 'C'])
     A    B      C  D_mean  E_mean
0  bar  one  large     4.0     5.0
1  bar  one  small     5.0     3.0
2  bar  two  large     7.0     1.0
3  bar  two  small     6.0     2.0
4  foo  one  large     2.0     4.5
5  foo  one  small     1.0     3.0
6  foo  two  small     3.0     5.0
```

我们可以指定 `values`来显示指定要计算的列。

```
>>> df.pivot_table(rows=['A', 'B'], values='D')
     A    B    D_mean
0  bar  one  4.500000
1  bar  two  6.500000
2  foo  one  1.666667
```

```
3  foo  two  3.000000
```

计算值列时，默认会计算平均值，用户可以指定一个或者多个聚合函数。

```
>>> df.pivot_table(rows=['A', 'B'], values=['D'], aggfunc=['mean', '
count', 'sum'])
     A    B    D_mean  D_count  D_sum
0  bar  one  4.500000        2      9
1  bar  two  6.500000        2     13
2  foo  one  1.666667        3      5
3  foo  two  3.000000        2      6
```

我们也可以把原始数据的某一列的值，作为新的collection的列。 这也是透视表最强大的地方。

```
>>> df.pivot_table(rows=['A', 'B'], values='D', columns='C')
     A    B  large_D_mean  small_D_mean
0  bar  one           4.0           5.0
1  bar  two           7.0           6.0
2  foo  one           2.0           1.0
3  foo  two           NaN           3.0
```

我们可以提供 `fill_value` 来填充空值。

```
>>> df.pivot_table(rows=['A', 'B'], values='D', columns='C',
fill_value=0)
     A    B  large_D_mean  small_D_mean
0  bar  one             4             5
1  bar  two             7             6
2  foo  one             2             1
3  foo  two             0             3
```

Key-Value 字符串转换

DataFrame 提供了将 Key-Value 对展开为列，以及将普通列转换为 Key-Value 列的功能。

我们的数据为

```
>>> df
    name                kv
0  name1  k1=1,k2=3,k5=10
1  name1    k1=7.1,k7=8.2
2  name2    k2=1.2,k3=1.5
3  name2      k9=1.1,k2=1
```

可以通过 extract_kv 方法将 Key-Value 字段展开：

```
>>> df.extract_kv(columns=['kv'], kv_delim='=', item_delim=',')
   name   kv_k1  kv_k2  kv_k3  kv_k5  kv_k7  kv_k9
0  name1    1.0    3.0    NaN   10.0    NaN    NaN
1  name1    7.0    NaN    NaN    NaN    8.2    NaN
2  name2    NaN    1.2    1.5    NaN    NaN    NaN
```

```
3  name2    NaN    1.0    NaN    NaN    NaN    1.1
```

其中，需要展开的字段名由 columns 指定，Key 和 Value 之间的分隔符，以及 Key-Value 对之间的分隔符分别由 kv_delim 和 item_delim 这两个参数指定，默认分别为半角冒号和半角逗号。输出的字段名为原字段名和 Key 值的组合，通过“_”相连。缺失值默认为 None，可通过 `fill_value` 选择需要填充的值。例如，相同的 df，

```
>>> df.extract_kv(columns=['kv'], kv_delim='=', fill_value=0)
   name   kv_k1  kv_k2  kv_k3  kv_k5  kv_k7  kv_k9
0  name1    1.0    3.0    0.0   10.0    0.0    0.0
1  name1    7.0    0.0    0.0    0.0    8.2    0.0
2  name2    0.0    1.2    1.5    0.0    0.0    0.0
3  name2    0.0    1.0    0.0    0.0    0.0    1.1
```

DataFrame 也支持将多列数据转换为一个 Key-Value 列。例如，

```
>>> df
   name    k1   k2   k3    k5   k7   k9
0  name1  1.0  3.0  NaN  10.0  NaN  NaN
1  name1  7.0  NaN  NaN   NaN  8.2  NaN
2  name2  NaN  1.2  1.5   NaN  NaN  NaN
3  name2  NaN  1.0  NaN   NaN  NaN  1.1
```

可通过 to_kv 方法转换为 Key-Value 表示的格式：

```
>>> df.to_kv(columns=['k1', 'k2', 'k3', 'k5', 'k7', 'k9'], kv_delim
='=')
    name               kv
0  name1  k1=1,k2=3,k5=10
1  name1    k1=7.1,k7=8.2
2  name2    k2=1.2,k3=1.5
3  name2      k9=1.1,k2=1
```

3.4.13 数据合并

```
from odps.df import DataFrame
```

```
movies = DataFrame(o.get_table('pyodps_ml_100k_movies'))
ratings = DataFrame(o.get_table('pyodps_ml_100k_ratings'))
```

```
movies.dtypes
```

```
odps.Schema {
  movie_id                                 int64
  title                                    string
  release_date                             string
  video_release_date                       string
  imdb_url                                 string
```

```
}
```

```
ratings.dtypes
```

```
odps.Schema {
  user_id                  int64
  movie_id                 int64
  rating                   int64
  unix_timestamp           int64
}
```

Join 操作

DataFrame 也支持对两个 Collection 执行 join 的操作，如果不指定 join 的条件，那么 DataFrame API会寻找名字相同的列，并作为 join 的条件。

```
>>> movies.join(ratings).head(3)
   movie_id              title  release_date  video_release_date
                                      imdb_url  user_id  rating
unix_timestamp
0         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...       49       3
888068877
1         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      621       5
881444887
2         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      291       3
874833936
```

我们也可以显式指定join的条件。有以下几种方式：

```
>>> movies.join(ratings, on='movie_id').head(3)
   movie_id              title  release_date  video_release_date
                                      imdb_url  user_id  rating
unix_timestamp
0         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...       49       3
888068877
1         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      621       5
881444887
2         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      291       3
874833936
```

在join时，on条件两边的字段名称相同时，只会选择一个，其他类型的join则会被重命名。

```
>>> movies.left_join(ratings, on='movie_id').head(3)
   movie_id_x              title  release_date  video_release_date
                                        imdb_url  user_id  movie_id_y
rating  unix_timestamp
0           3  Four Rooms (1995)   01-Jan-1995
http://us.imdb.com/M/title-exact?Four%20Rooms%...       49           3
       3       888068877
```

```
1           3  Four Rooms (1995)   01-Jan-1995
http://us.imdb.com/M/title-exact?Four%20Rooms%...      621           3
       5       881444887
2           3  Four Rooms (1995)   01-Jan-1995
http://us.imdb.com/M/title-exact?Four%20Rooms%...      291           3
       3       874833936
```

可以看到，`movie_id`被重命名为movie_id_x，以及movie_id_y，这和`suffixes`参数有关（默认是`('_x', '_y')`），当遇到重名的列时，就会被重命名为指定的后缀。

```
>>> ratings2 = ratings[ratings.exclude('movie_id'), ratings.movie_id.
rename('movie_id2')]
>>> ratings2.dtypes
odps.Schema {
  user_id                  int64
  rating                   int64
  unix_timestamp           int64
  movie_id2                int64
}
>>> movies.join(ratings2, on=[('movie_id', 'movie_id2')]).head(3)
   movie_id              title  release_date  video_release_date
                                      imdb_url  user_id  rating
unix_timestamp  movie_id2
0         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...       49       3
888068877          3
1         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      621       5
881444887          3
2         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      291       3
874833936          3
```

也可以直接写等于表达式。

```
>>> movies.join(ratings2, on=[movies.movie_id == ratings2.movie_id2]).
head(3)
   movie_id              title  release_date  video_release_date
                                      imdb_url  user_id  rating
unix_timestamp  movie_id2
0         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...       49       3
888068877          3
1         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      621       5
881444887          3
2         3  Four Rooms (1995)   01-Jan-1995                      http
://us.imdb.com/M/title-exact?Four%20Rooms%...      291       3
874833936          3
```

self-join的时候，可以调用`view`方法，这样就可以分别取字段。

```
>>> movies2 = movies.view()
>>> movies.join(movies2, movies.movie_id == movies2.movie_id)[movies,
movies2.movie_id.rename('movie_id2')].head(3)
   movie_id            title_x release_date_x video_release_date_x  \
0         2   GoldenEye (1995)    01-Jan-1995                 True
1         3  Four Rooms (1995)    01-Jan-1995                 True
```

```
2         4  Get Shorty (1995)    01-Jan-1995                 True

                                          imdb_url_x  movie_id2
0  http://us.imdb.com/M/title-exact?GoldenEye%20(...          2
1  http://us.imdb.com/M/title-exact?Four%20Rooms%...          3
2  http://us.imdb.com/M/title-exact?Get%20Shorty%...          4
```

除了`join`以外，DataFrame还支持`left_join`，`right_join`，和`outer_join`。在执行上述外连接操作时， 默认会将重名列加上 _x 和 _y 后缀，可通过在 suffixes 参数中传入一个二元 tuple 来自定义后缀。

如果需要在外连接中避免对谓词中相等的列取重复列，可以指定 merge_columns 选项，该选项会自动选择两列中的非空值作为新列的值：

```
>>> movies.left_join(ratings, on='movie_id', merge_columns=True)
```

要使用 **mapjoin**也很简单，只需将mapjoin设为True，执行时会对右表做mapjoin操作。

用户也能join分别来自ODPS和pandas的Collection，或者join分别来自ODPS和数据库的Collection，此时计算会在ODPS上执行。

Union操作

现在有两张表，字段和类型都一致（可以顺序不同），我们可以使用union或者concat来把它们合并成一张表。

```
>>> mov1 = movies[movies.movie_id < 3]['movie_id', 'title']
>>> mov2 = movies[(movies.movie_id > 3) & (movies.movie_id < 6)]['
title', 'movie_id']
>>> mov1.union(mov2)
   movie_id              title
0         1   Toy Story (1995)
1         2   GoldenEye (1995)
2         4  Get Shorty (1995)
3         5     Copycat (1995)
```

用户也能union分别来自ODPS和pandas的Collection，或者union分别来自ODPS和数据库的Collection，此时计算会在ODPS上执行。

3.4.14 窗口函数

DataFrame API也支持使用窗口函数。

```
>>> grouped = iris.groupby('name')
>>> grouped.mutate(grouped.sepallength.cumsum(), grouped.sort('
sepallength').row_number()).head(10)
          name  sepallength_sum  row_number
0  Iris-setosa            250.3           1
1  Iris-setosa            250.3           2
2  Iris-setosa            250.3           3
```

```
3  Iris-setosa            250.3           4
4  Iris-setosa            250.3           5
5  Iris-setosa            250.3           6
6  Iris-setosa            250.3           7
7  Iris-setosa            250.3           8
8  Iris-setosa            250.3           9
9  Iris-setosa            250.3          10
```

窗口函数可以使用在列选择中：

```
>>> iris['name', 'sepallength', iris.groupby('name').sort('sepallength
').sepallength.cumcount()].head(5)
          name  sepallength  sepallength_count
0  Iris-setosa          4.3                  1
1  Iris-setosa          4.4                  2
2  Iris-setosa          4.4                  3
3  Iris-setosa          4.4                  4
4  Iris-setosa          4.5                  5
```

窗口函数按标量聚合时，和分组聚合的处理方式一致。

```
>>> from odps.df import Scalar
>>> iris.groupby(Scalar(1)).sort('sepallength').sepallength.cumcount()
```

DataFrame API支持的窗口函数包括：

窗口函数	说明
cumsum	计算累积和
cummean	计算累积均值
cummedian	计算累积中位数
cumstd	计算累积标准差
cummax	计算累积最大值
cummin	计算累积最小值
cumcount	计算累积和
lag	按偏移量取当前行之前第几行的值，如当前行号为rn，则取行号为rn-offset的值
lead	按偏移量取当前行之后第几行的值，如当前行号为rn则取行号为rn+offset的值
rank	计算排名
dense_rank	计算连续排名
percent_rank	计算一组数据中某行的相对排名
row_number	计算行号，从1开始

窗口函数	说明
qcut	将分组数据按顺序切分成n片，并返回当前切片值，如果切片不均匀，默认增加第一个切片的分布
nth_value	返回分组中的第n个值
cume_dist	计算分组中值小于等于当前值的行数占分组总行数的比例

其中，rank、dense_rank、percent_rank 和 row_number 支持下列参数：

参数名	说明
sort	排序关键字，默认为空
ascending	排序时，是否依照升序排序，默认 True

lag 和 lead 除 rank 的参数外，还支持下列参数：

参数名	说明
offset	取数据的行距离当前行的行数
default	当 offset 指定的行不存在时的返回值

而 cumsum、cummax、cummin、cummean、cummedian、cumcount 和 cumstd 除 rank 的上述参数外，还支持下列参数：

参数名	说明
unique	是否排除重复值，默认 False
preceding	窗口范围起点
following	窗口范围终点

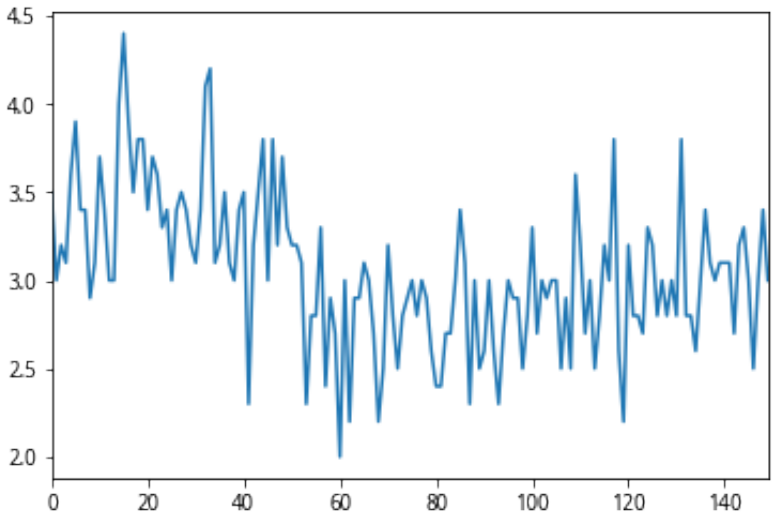
3.4.15 绘图

PyODPS DataFrame提供了绘图的方法。如果要使用绘图，需要 pandas 和 matplotlib 的安装。

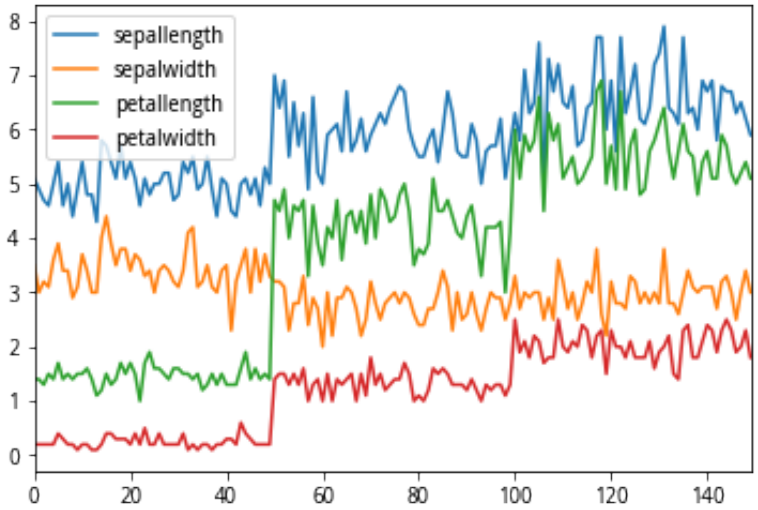
接下来的例子都是在jupyter中运行。

```
>>> from odps.df import DataFrame
>>> iris = DataFrame(o.get_table('pyodps_iris'))
>>> %matplotlib inline
>>> iris.sepalwidth.plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10c2b3510>
```

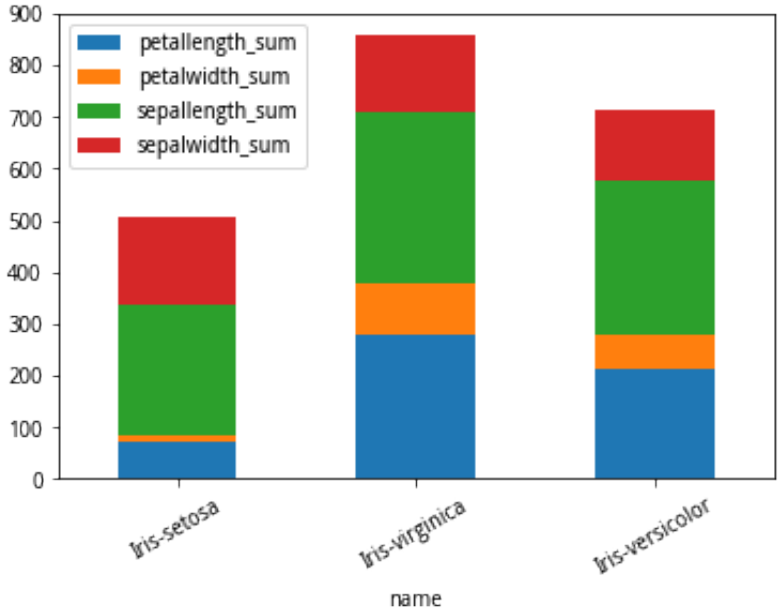



```
>>> iris.plot()
<matplotlib.axes._subplots.AxesSubplot at 0x10db7e690>
```

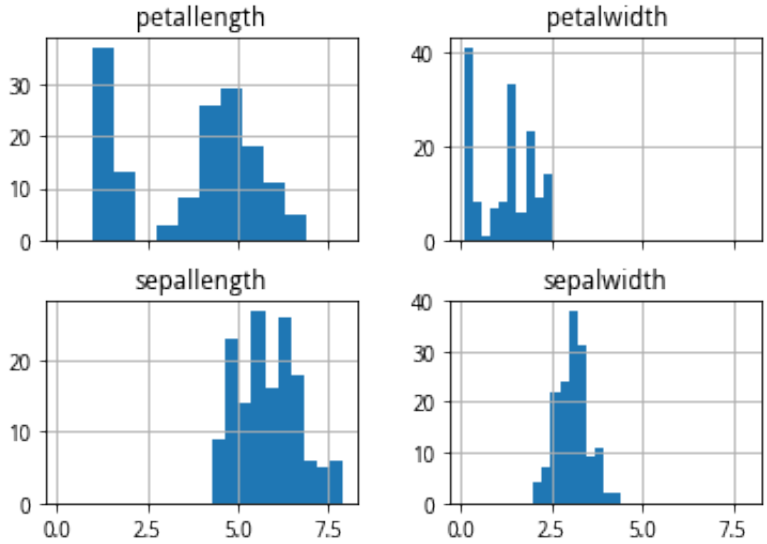



```
>>> iris.groupby('name').sum().plot(kind='bar', x='name', stacked=True
, rot=30)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x10c5f2090>
```




```
>>> iris.hist(sharex=True)
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x10e013f90>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10e2d1c10
>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x10e353f10>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x10e3c4410
>]], dtype=object)
```



参数kind表示了绘图的类型，支持的包括：

kind	说明
line	线图

kind	说明
bar	竖向柱状图
barh	横向柱状图
hist	直方图
box	boxplot
kde	核密度估计
density	和kde相同
area	
pie	饼图
scatter	散点图
hexbin	

详细参数可以参考Pandas文档：*pandas.DataFrame.plot*

除此之外，plot函数还增加了几个参数，方便进行绘图。

参数	说明
xlabel	x轴名
ylabel	y轴名
xlabelsize	x轴名大小
ylabelsize	y轴名大小
labelsize	轴名大小
title	标题
titlesize	标题大小
annotate	是否标记值

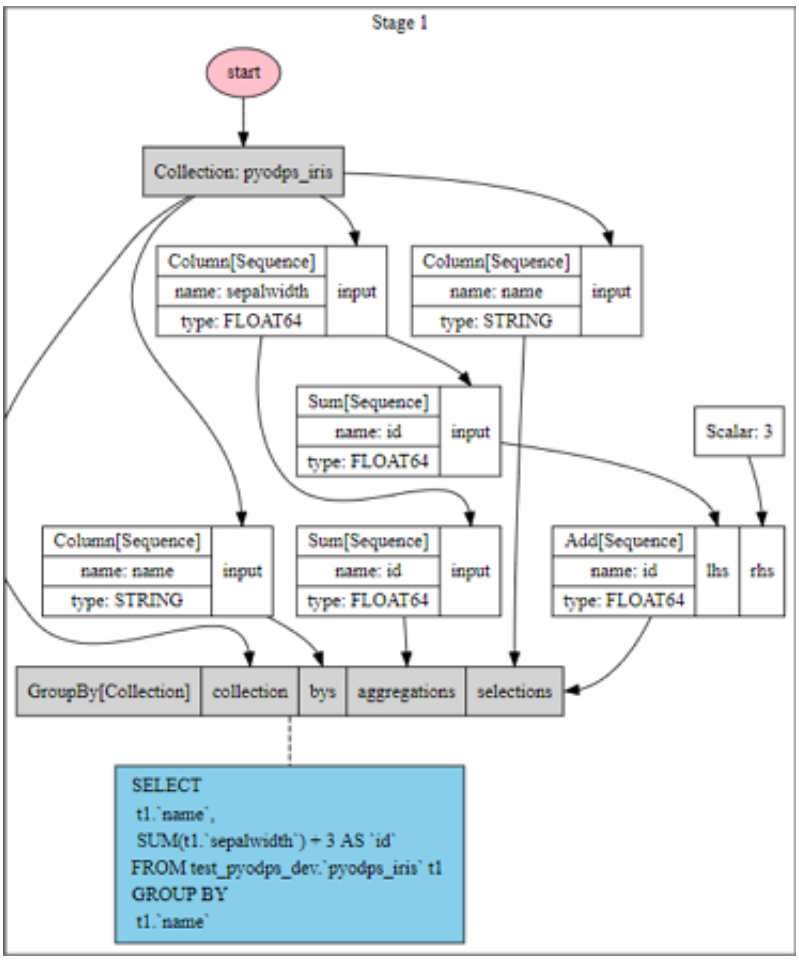
3.4.16 调试指南

可视化DataFrame

由于PyODPS DataFrame本身会对整个操作执行优化，为了能直观地反应整个过程， 我们可以使用可视化的方式显示整个表达式的计算过程。

值得注意的是，可视化需要依赖 *graphviz 软件* 和 **graphviz** Python 包。

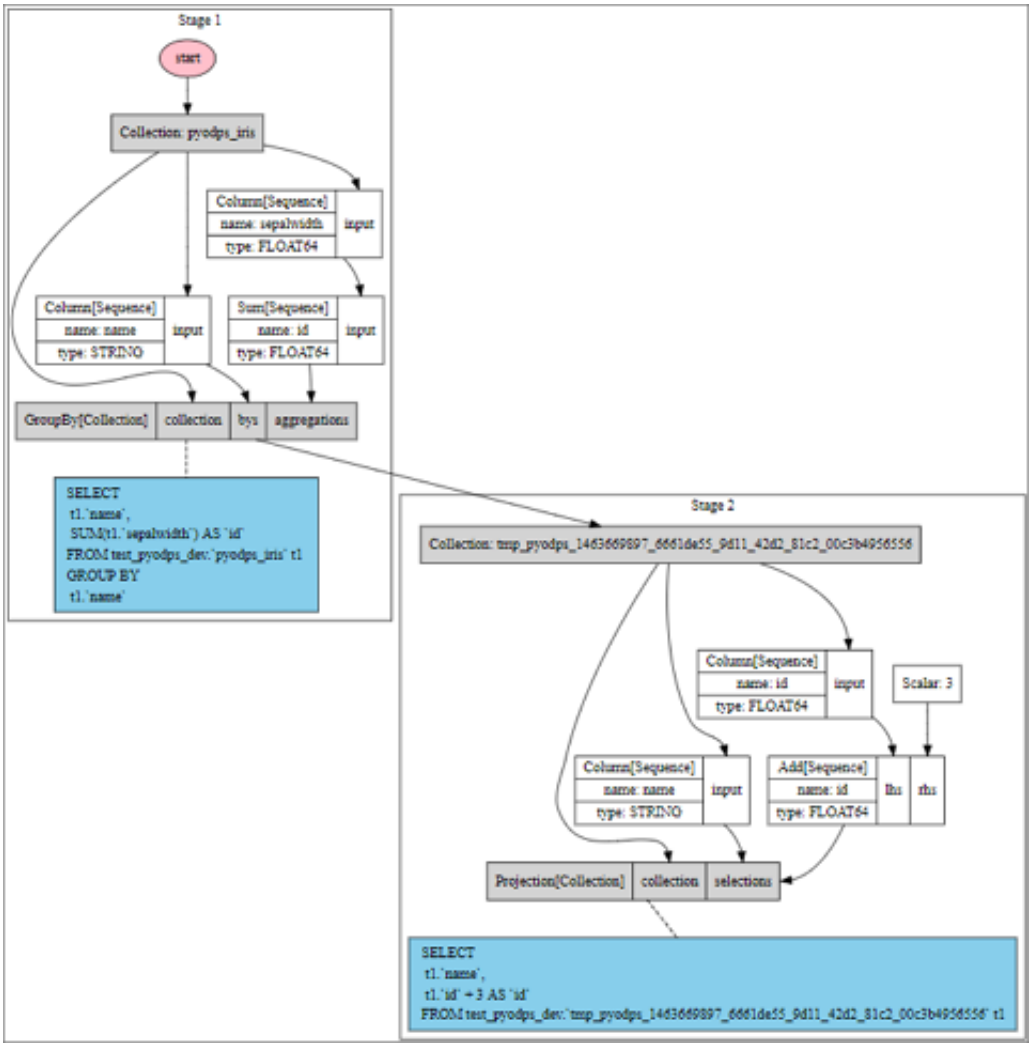
```
>>> df = iris.groupby('name').agg(id=iris.sepalwidth.sum())
>>> df = df[df.name, df.id + 3]
>>> df.visualize()
```



可以看到，这个计算过程中，PyODPS DataFrame将GroupBy和列筛选做了操作合并。

```
>>> df = iris.groupby('name').agg(id=iris.sepalwidth.sum()).cache()
>>> df2 = df[df.name, df.id + 3]
```

```
>>> df2.visualize()
```



此时，由于用户执行了cache操作，这时整个执行计划将会分成两步来执行。

ODPS SQL后端查看编译结果

我们可以直接调用 `compile`方法来查看ODPS SQL后端编译到SQL的结果。

```
>>> df = iris.groupby('name').agg(sepalwidth=iris.sepalwidth.max())
>>> df.compile()
Stage 1:

SQL compiled:

SELECT
  t1.`name`,
  MAX(t1.`sepalwidth`) AS `sepalwidth`
FROM test_pyodps_dev.`pyodps_iris` t1
GROUP BY
```

```
  t1.`name`
```

使用**pandas**计算后端执行本地调试

对于来自ODPS表的DataFrame，一些操作不会compile到ODPS SQL执行，而是会使用Tunnel下载， 这个过程是很快的，且无需等待ODPS SQL任务的调度。 利用这个特性，我们能快速下载小部分ODPS数据到本地，使用pandas计算后端来进行代码编写和调试。

这些操作包括：

- 对非分区表进行选取整个或者有限条数据、或者列筛选的操作（不包括列的各种计算），以及计算其数量
- 对分区表不选取分区或筛选前几个分区字段，对其进行选取全部或者有限条数据、或者列筛选的操作，以及计算其数量

如我们的iris这个DataFrame的来源ODPS表是非分区表，以下操作会使用tunnel进行下载。

```
>>> iris.count()
>>> iris['name', 'sepalwidth'][:10]
```

对于分区表，如有个DataFrame来源于分区表（有三个分区字段，ds、hh、mm），以下操作会使用tunnel下载。

```
>>> df[:10]
>>> df[df.ds == '20160808']['f0', 'f1']
>>> df[(df.ds == '20160808') & (df.hh == 3)][:10]
>>> df[(df.ds == '20160808') & (df.hh == 3) & (df.mm == 15)]
```

因此我们可以使用 `to_pandas` 方法来将部分数据下载到本地来进行调试，我们可以写出如下代码：

```
>>> DEBUG = True
```

```
>>> if DEBUG:
>>>     df = iris[:100].to_pandas(wrap=True)
>>> else:
>>>     df = iris
```

这样，当我们全部编写完成时，再把 `DEBUG`设置为False就可以在ODPS上执行完整的计算了。

 说明：

由于沙箱的限制，本地调试通过的程序不一定能在ODPS上也跑通。

3.5 交互体验增强

3.5.1 Jupyter Notebook 增强

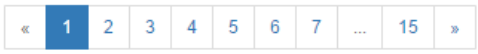
PyODPS 针对 Jupyter Notebook下的探索性数据分析进行了增强， 包括结果探索功能以及进度展示功能。

结果探索

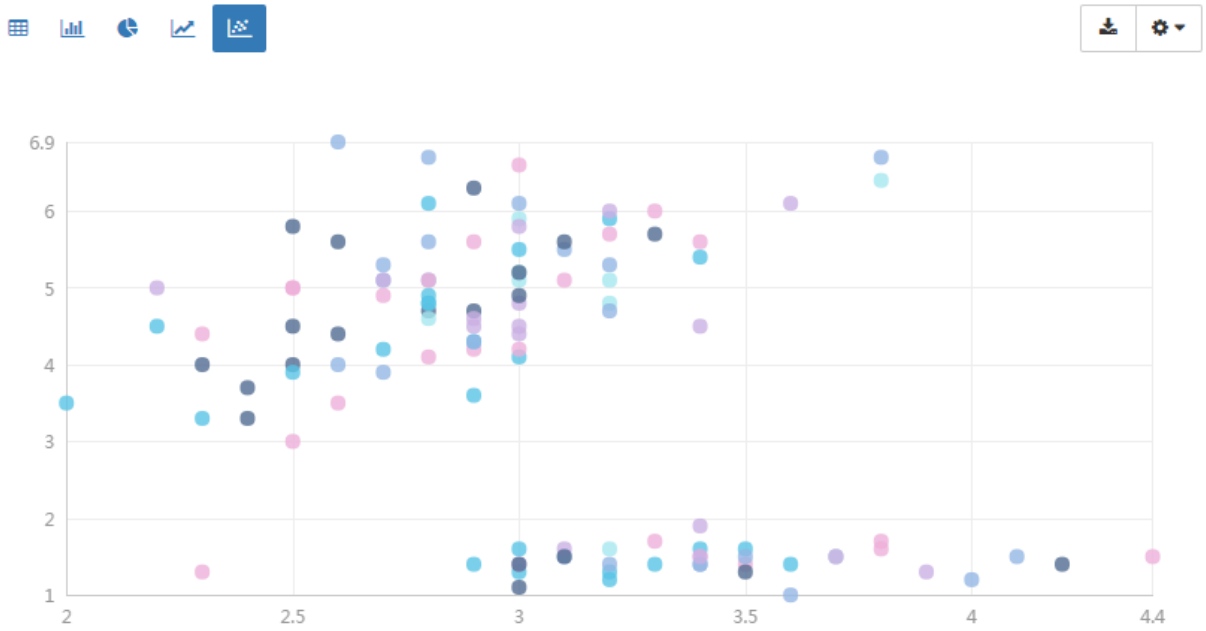
PyODPS 在 Jupyter Notebook 中为 SQL Cell 和 DataFrame 提供了数据探索功能。对于已拉到本地的数据，可使用交互式的数据探索工具 浏览数据，交互式地绘制图形。

当执行结果为 DataFrame 时，PyODPS 会读取执行结果，并以分页表格的形式展示出来。单击页号或前进 / 后退按钮可在数据中导航， 如下图。

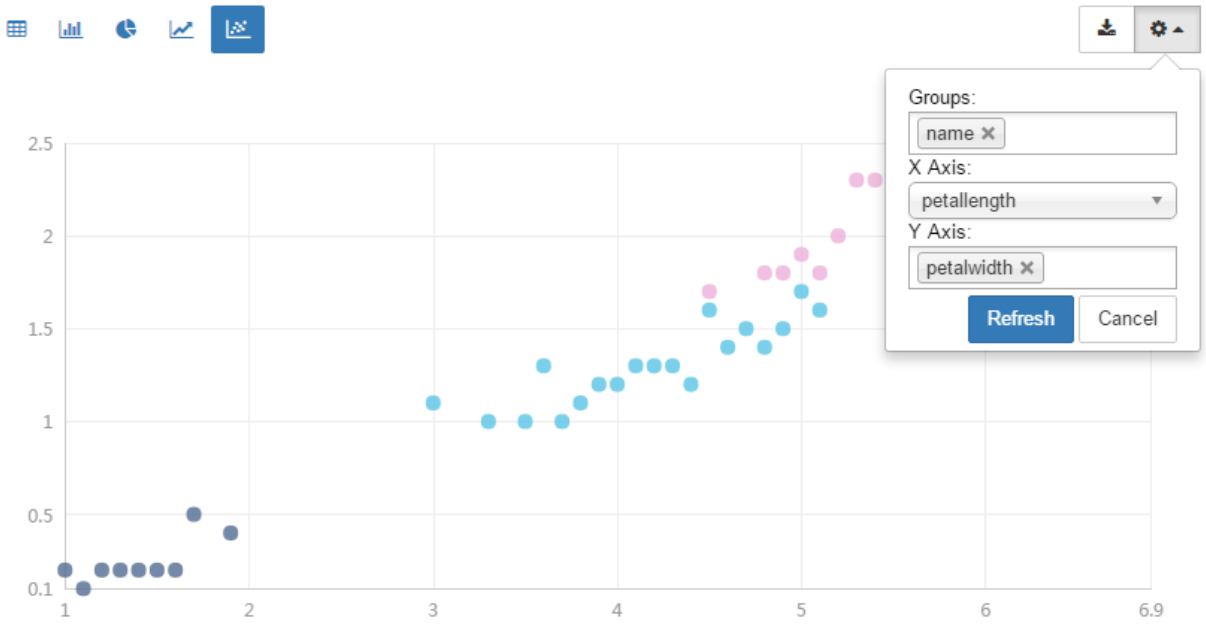
sepallength	sepalwidth	petallength	petalwidth	name
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa



结果区的顶端为模式选择区。除数据表外，也可以选择柱状图、饼图、折线图和散点图。下图为使用默认字段选择（即前三个字段）绘制的散点图。

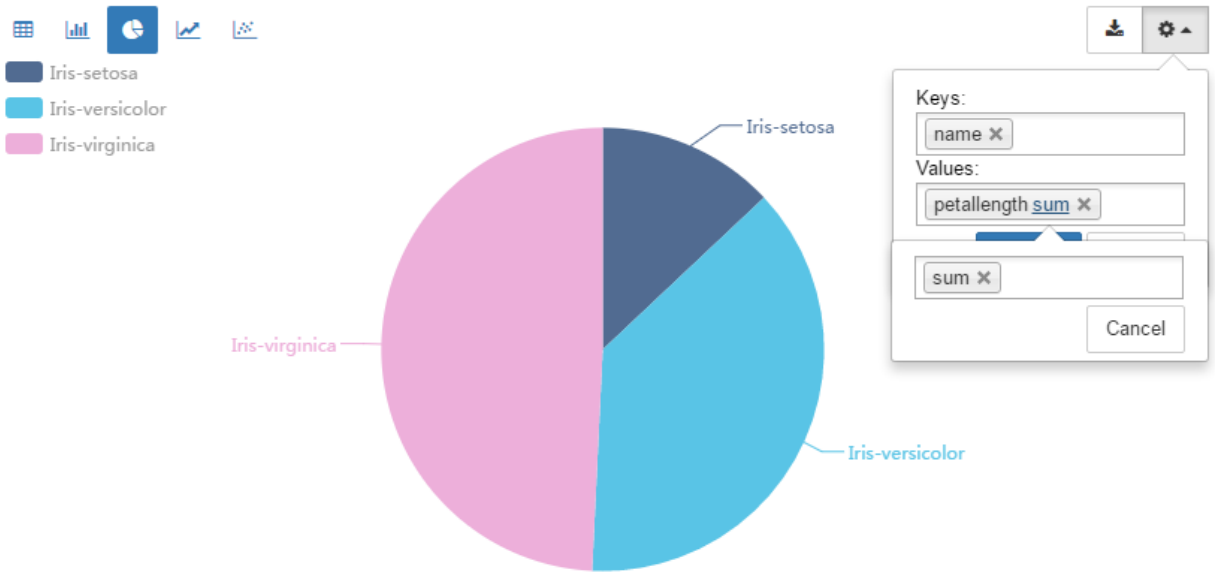


在绘图模式下，单击右上角的配置按钮可以修改图表设置。如下图中，将 name 设置为分组列，X 轴选择为 petallength，Y 轴选择为 petalwidth，则图表变为下图。可见在 petallength - petalwidth 维度下，数据对 name 有较好的区分度。



对于柱状图和饼图，值字段支持选择聚合函数。PyODPS 对柱状图的默认聚合函数为 sum，对饼图则为 count。如需修改聚合函数， 可在值字段名称后的聚合函数名上单击，此后选择所需的聚合函数即可。

对于折线图，需要避免 X 轴包含空值，否则图像可能不符合预期。



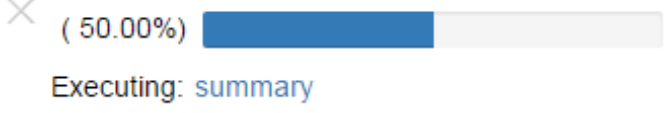
完成绘图后，可单击“下载”保存绘制的图表。

> **说明：**
> 注意：使用此功能需要安装 Pandas ，并保证 ipywidgets 被正确安装。

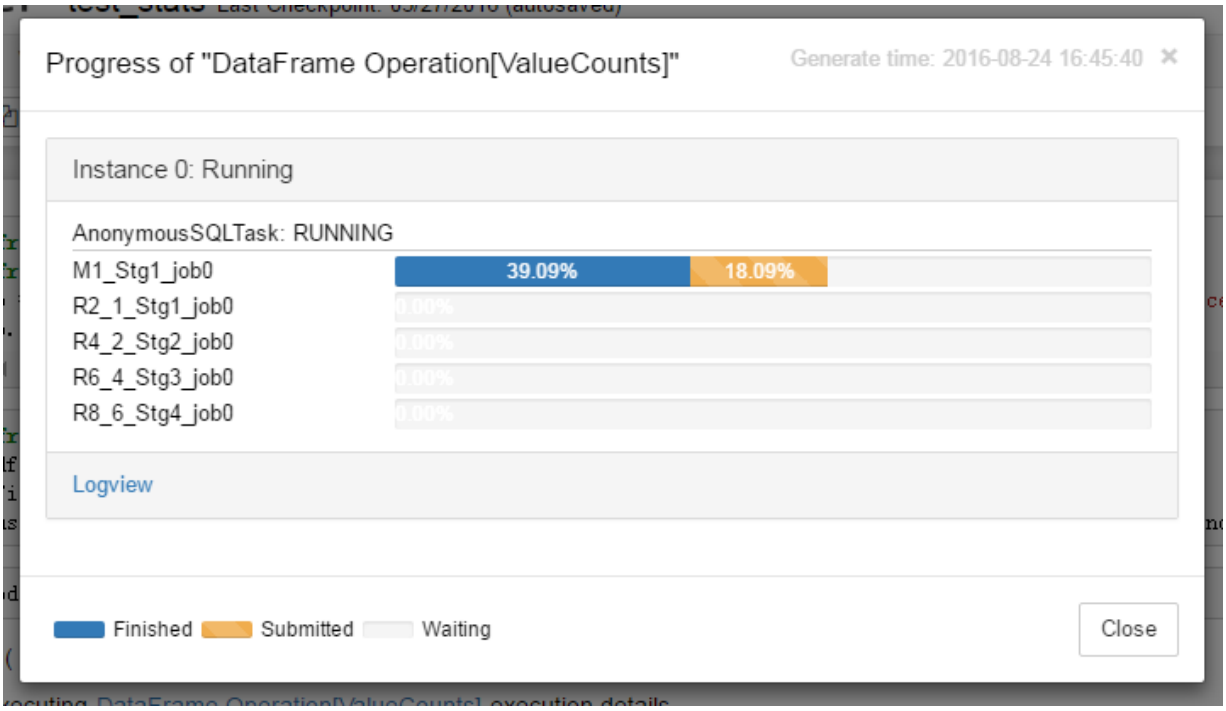
进度展示

大型作业执行通常需要较长的时间，因而 PyODPS 提供了进度展示功能。当 DataFrame、机器学习作业或通过 %sql 编写的 SQL 语句在 Jupyter Notebook 中执行作业时，会显示当前正在执行的作业列表及总体进度，如下图：

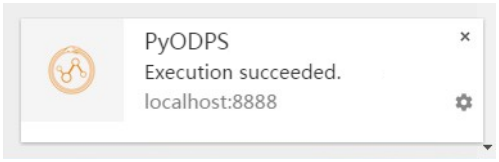
```
In [*]: from odps import ODPS
from odps.df.examples import create_ionosphere
o = ODPS(access_id, secret_access_key, project=project, endpoint=endpoint)
df = create_ionosphere(o)['a01', 'a02', 'a03', 'a04', 'class']
df.calc_summary()
```



当点击某个作业名称上的链接时，会弹出一个对话框，显示该作业中每个 Task 的具体执行进度，如图：



当作业运行成功后，浏览器将给出提醒信息，告知作业是否成功：



3.5.2 IPython增强

PyODPS 还提供了 IPython 的插件，来更方便得操作 ODPS。

首先，针对命令行增强，也有相应的命令。让我们先加载插件：

```
%load_ext odps
```

```
%enter
```

```
<odps.inter.Room at 0x11341df10>
```

此时全局会包含o和odps变量，即ODPS入口。

```
o.get_table('dual')
odps.get_table('dual')
```

```
odps.Table
  name: odps_test_sqltask_finance.`dual`
  schema:
    c_int_a                 : bigint
    c_int_b                 : bigint
    c_double_a              : double
```

```
    c_double_b              : double
    c_string_a              : string
    c_string_b              : string
    c_bool_a                : boolean
    c_bool_b                : boolean
    c_datetime_a            : datetime
    c_datetime_b            : datetime
```

```
%stores
```

default	desc
name	
iris	安德森鸢尾花卉数据集

对象名补全

PyODPS 拓展了 IPython 原有的代码补全功能，支持在书写 `o.get_xxx` 这样的语句时，自动补全对象名。

例如，在 IPython 中输入下列语句（ <tab>不是实际输入的字符，而是当所有输入完成后，将光标移动到相应位置， 并按 Tab 键 ）：

```
o.get_table(<tab>
```

如果已知需要补全对象的前缀，也可以使用

```
o.get_table('tabl<tab>
```

IPython 会自动补全前缀为 tabl 的表。

对象名补全也支持补全不同 Project 下的对象名。下列用法都被支持：

```
o.get_table(project='project_name', name='tabl<tab>
o.get_table('tabl<tab>', project='project_name')
```

如果匹配的对象有多个，IPython 会给出一个列表，其最大长度由 `options.completion_size` 给出， 默认为 10。

SQL命令

PyODPS 还提供了 SQL 插件，来执行 ODPS SQL。下面是单行 SQL：

```
%sql select * from pyodps_iris limit 5
```

	sepallength	sepalwidth	petallength	petalwidth	name
0	5.1	3.5	1.4	0.2	Iris-setosa

	sepallength	sepalwidth	petallength	petalwidth	name
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

多行SQL可以使用%%sql的命令

```
%%sql

select * from pyodps_iris
where sepallength < 5
limit 5
```

	sepallength	sepalwidth	petallength	petalwidth	name
0	4.9	3.0	1.4	0.2	Iris-setosa
1	4.7	3.2	1.3	0.2	Iris-setosa
2	4.6	3.1	1.5	0.2	Iris-setosa
3	4.6	3.4	1.4	0.3	Iris-setosa
4	4.4	2.9	1.4	0.2	Iris-setosa

如果想执行参数化SQL查询，则需要替换的参数可以使用:参数的方式。

```
In [1]: %load_ext odps

In [2]: mytable = 'dual'

In [3]: %sql select * from :mytable
|==========================================|   1 /  1  (100.00%)
   2s
Out[3]:
   c_int_a  c_int_b  c_double_a  c_double_b  c_string_a  c_string_b
c_bool_a  \
0        0        0       -1203           0           0       -1203
  True

  c_bool_b         c_datetime_a         c_datetime_b
0    False  2012-03-30 23:59:58  2012-03-30 23:59:59
```

设置SQL运行时参数，可以通过 %set设置到全局，或者在sql的cell里用SET进行局部设置。

```
In [17]: %%sql
         set odps.sql.mapper.split.size = 16;
```

```
          select * from pyodps_iris;
```

这个会局部设置，不会影响全局的配置。

```
In [18]: %set odps.sql.mapper.split.size = 16
```

这样设置后，后续运行的SQL都会使用这个设置。

持久化 pandas DataFrame 到 ODPS 表

PyODPS 还提供把 pandas DataFrame 上传到 ODPS 表的命令:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.arange(9).reshape(3, 3), columns=list('abc'))
```

```
%persist df pyodps_pandas_df
```

这里的第0个参数df是前面的变量名，pyodps_pandas_df是ODPS表名。

3.5.3 命令行增强

PyODPS 提供了命令行下的增强工具。

首先，用户可以在任何地方配置了帐号以后，下次就无需再次输入帐号信息。

```
from odps.inter import setup, enter, teardown
```

接着就可以配置帐号

```
setup('**your-access_id**', '**your-access-key**', '**your-project
**', endpoint='**your-endpoint**')
```

在不指定room这个参数时，会被配置到叫做default的room里。

以后，在任何命令行打开的地方，都可以直接调用：

```
room = enter()
```

我们可以拿到ODPS的入口：

```
o = room.odps
```

```
o.get_table('dual')
```

```
odps.Table
  name: odps_test_sqltask_finance.`dual`
  schema:
    c_int_a                 : bigint
    c_int_b                 : bigint
```

```
    c_double_a                : double
    c_double_b                : double
    c_string_a                : string
    c_string_b                : string
    c_bool_a                  : boolean
    c_bool_b                  : boolean
    c_datetime_a              : datetime
    c_datetime_b              : datetime
```

说明：

在重新 setup room 后，ODPS 入口对象并不会自动替换，需要再次调用 enter() 以获得新的 Room 对象。

我们可以把常用的ODPS表或者资源都可以存放在room里。

```
room.store('存储表示例', o.get_table('dual'), desc='简单的表存储示例')
```

我们可以调用`display`方法，来把已经存储的对象以表格的形式打印出来：

```
room.display()
```

default	desc
name	
存储表示例	简单的表存储示例
iris	安德森鸢尾花卉数据集

我们通过`room['存储表示例']`，或者像`room.iris`，就可以取出来存储的对象了。

```
room['存储表示例']
```

```
odps.Table
  name: odps_test_sqltask_finance.`dual`
  schema:
    c_int_a                   : bigint
    c_int_b                   : bigint
    c_double_a                : double
    c_double_b                : double
    c_string_a                : string
    c_string_b                : string
    c_bool_a                  : boolean
    c_bool_b                  : boolean
    c_datetime_a              : datetime
```

```
    c_datetime_b            : datetime
```

删除也很容易，只需要调用drop方法

```
room.drop('存储表示例')
```

```
room.display()
```

default	desc
name	
iris	安德森鸢尾花卉数据集

要删除某个room，只需要调用teardown就可以了，不传参数时删除默认room。

```
teardown()
```

3.6 配置选项

PyODPS 提供了一系列的配置选项，可通过 `odps.options` 获得，如下面的例子：

```
from odps import options
# 设置所有输出表的生命周期（lifecycle 选项）
options.lifecycle = 30
# 使用 Tunnel 下载 string 类型时使用 bytes（tunnel.string_as_binary 选项）
options.tunnel.string_as_binary = True
# PyODPS DataFrame 用 ODPS 执行时，参照下面 dataframe 相关配置，sort 时设置 limit 到一个比较大的值
options.df.odps.sort.limit = 100000000
```

下面列出了可配的 ODPS 选项。

通用配置

选项	说明	默认值
end_point	ODPS Endpoint	None
default_project	默认 Project	None
log_view_host	LogView 主机名	None
log_view_hours	LogView 保持时间（小时）	24
local_timezone	使用的时区，True 表示本地时间，False 表示 UTC， 也可用 pytz 的时区	1

选项	说明	默认值
lifecycle	所有表生命周期	None
temp_lifecycle	临时表生命周期	1
biz_id	用户 ID	None
verbose	是否打印日志	False
verbose_log	日志接收器	None
chunk_size	写入缓冲区大小	1496
retry_times	请求重试次数	4
pool_connections	缓存在连接池的连接数	10
pool_maxsize	连接池最大容量	10
connect_timeout	连接超时	5
read_timeout	读取超时	120
api_proxy	API 代理服务器	None
data_proxy	数据代理服务器	None
completion_size	对象补全列举条数限制	10
notebook_repr_widget	使用交互式图表	True
sql.settings	ODPS SQL运行全局hints	None
sql.use_odps2_extension	启用 MaxCompute 2.0 语言扩展	False

数据上传/下载配置

选项	说明	默认值
tunnel.endpoint	Tunnel Endpoint	None
tunnel.use_instance_tunnel	使用 Instance Tunnel 获取执行结果	True
tunnel.limit_instance_tunnel	是否限制 Instance Tunnel 获取结果的条数	None
tunnel.string_as_binary	在 string 类型中使用 bytes 而非 unicode	False

DataFrame 配置

选项	说明	默认值
interactive	是否在交互式环境	根据检测值
df.analyze	是否启用非 ODPS 内置函数	True
df.optimize	是否开启DataFrame全部优化	True
df.optimizes.pp	是否开启DataFrame谓词下推优化	True
df.optimizes.cp	是否开启DataFrame列剪裁优化	True
df.optimizes.tunnel	是否开启DataFrame使用tunnel优化执行	True
df.quote	ODPS SQL后端是否用``来标记字段和表名	True
df.libraries	DataFrame运行使用的第三方库（资源名）	None
df.supersede_libraries	使用自行上传的numpy替换服务中的版本	False
df.odps.sort.limit	DataFrame有排序操作时，默认添加的limit条数	10000

机器学习配置

选项	说明	默认值
ml.xflow_settings	Xflow 执行配置	None
ml.xflow_project	默认 Xflow 工程名	algo_public
ml.use_model_transfer	是否使用 ModelTransfer 获取模型 PMML	False
ml.model_volume	在使用 ModelTransfer 时使用的 Volume 名称	pyodps_volume

3.7 API Reference

3.7.1 API概述

我们将为您提供自动生成的PyODPS API文档。

- *ODPS详解#Definitions#*
- *PyODPS DataFrame指南#DataFrame Reference#*

4 MaxCompute管家

当开通MaxCompute预付费后，会遇到账号购买了150CU，但是经常看到使用预付费的项目很多任务依然要排队很长时间，管理员或运维人员希望能看到具体哪些任务抢占了资源，从而对任务进行合理的管控，如按任务对应的业务优先级进行调度时间调整，重要和次要任务错开调度的问题。

MaxCompute 管家就是解决这个预付费计算资源监控管理的问题。目前MaxCompute 管家主要提供三个功能，系统状态、资源组分配、任务监控。具体使用说明请参考DataWorks文档*MaxCompute*预付费资源监控工具-*CU*管家。

说明：

MaxCompute管家使用须知：

- 您需要购买了MaxCompute预付费CU资源，且购买数量为60CU或以上。CU过小无法发挥计算资源及管家的优势。