
ODPS 教程

BY 李妹芳

前言	5
第 1 章 ODPS 入门.....	6
1.1 主要功能	6
1.2 基本概念	6
1.2.1 用户空间 Project	6
1.2.2 表 Table	7
1.2.3 分区 Partition.....	7
1.2.4 资源 Resource.....	7
1.2.5 数据类型.....	7
1.3 客户端	8
1.4 使用 ODPS 的准备工作	9
1.4.1 创建账号.....	9
1.4.2 创建 Project	10
1.4.3 授权.....	11
1.5 ODPS CLT.....	11
1.5.1 下载和配置.....	11
1.5.2 运行模式.....	12
第 2 章 ODPS TUNNEL	13
2.1 开发前的准备工作	13
2.1.1 配置开发环境.....	13
2.2 上传数据	14
2.2.1 准备数据.....	14
2.2.2 代码实现和分析.....	15
2.2.3 如何提高并发.....	18
2.2.4 注意事项.....	18
2.2.5 运行和结果输出.....	19
2.3 下载数据	20
2.3.1 代码实现和分析.....	21
2.3.2 如何提高并发.....	22
2.4 核心接口	23
2.4.1 Configuration.....	23
2.4.2 DataTunnel	24
2.4.3 Upload.....	24
2.4.4 Download.....	26

2.4.5 Record	27
2.5 综合示例	28
2.6 FAQ	28
第3章 ODPS MAPREDUCE 入门	29
3.1 初识 ODPS MR	29
3.1.1 输入和输出	29
3.1.2 资源	29
3.2 开发前的准备工作	30
3.2.1 开发流程	30
3.2.2 配置开发环境	30
3.3 入门示例 WordCount	32
3.3.1 准备工作	32
3.3.2 通过 Tunnel 导入数据	33
3.3.3 通过 MR 实现单词计数	36
3.3.4 本地运行调试	40
3.3.5 单元测试	43
3.3.6 集群调试	47
3.4 小结	52
第4章 ODPS MAPREDUCE 进阶探讨	54
4.1 ODPS MAPREDUCE 原理	54
4.1.1 从用户视角谈原理	54
4.1.2 如何实现 map 和 reduce?	56
4.1.3 使用 Combiner?	56
4.1.4 自定义 Comparator?	57
4.1.5 自定义 Partitioner?	57
4.2 核心接口	57
4.2.1 Record	58
4.2.2 Mapper	59
4.2.3 Reducer	60
4.2.4 Partitioner	62
4.2.5 Comparator	63
4.2.6 Combiner	63
4.3 简单示例	64
4.3.1 使用 Combiner	64
4.3.2 自定义 Comparator	68
4.3.1 示例：自定义 Partitioner	70
4.3.2 示例：使用 DistributedCache	74
4.4 综合示例	74
4.4.1 查找淘宝用户共同特征	74
4.4.2 来往好友推荐	74
4.4.3 家庭树	82
4.5 性能调优	88
4.5.1 代码实现逻辑	88

4.5.2 使用 Combiner.....	89
4.5.3 使用 Distributed Cache.....	89
4.5.4 合理配置作业参数.....	90
4.5.5 减少输入的数据量.....	90
4.5.6 数据倾斜怎么办?	91
4.6 FAQ	91
第 5 章 ODPS SQL	92
5.1 保留字和运算符	92
5.1 主要功能	92
5.2 示例场景说明	93
5.3 简单的 DDL 操作.....	93
5.3.1 创建表.....	93
5.3.2 添加分区.....	94
5.3.3 查看表和分区.....	95
5.3.4 修改表.....	95
5.3.5 删除表和分区.....	95
5.4 生成数据	96
5.5 查询分析 DML.....	96
5.5.1 简单查询.....	96
5.5.2 分区表查询.....	97
5.5.3 连接操作 Join.....	99
5.5.4 MAPJOIN.....	101
5.5.5 聚合操作.....	103
5.5.6 窗口函数.....	104
5.5.7 多路输出 (Multi Insert)	105
5.5.8 动态分区 (Dynamic Partition)	106
5.5.9 子查询.....	107
5.5.10 UNION ALL.....	107
5.5.11 排序.....	108
5.5.12 CASE WHEN 表达式.....	110
5.6 UDF.....	110
5.6.1 UDF.....	111
5.6.2 UDAF.....	114
5.6.3 UDTF.....	116
5.7 调优	116
5.7.1 查看执行计划 Explain	116
5.7.2 常见的优化考虑和注意.....	116
5.8 FAQ	117
第 6 章 ODPS 安全	119
第 7 章 ODPS 实战.....	120
7.1 场景和数据说明	120
7.2 通过 TUNNEL 上传数据	121
7.2.1 创建表.....	121

7.2.2 导入数据.....	123
7.3 数据分析需求	124
7.4 通过 MAPREDUCE 按年份分区	124
7.4.1 创建表.....	124
7.4.2 代码实现.....	125
7.4.3 编译和运行	127
7.5 通过 SQL 分析处理	129
7.5.1 统计 PM10 值最大的国家	129
7.5.2 统计指标报表.....	130
7.6 通过 PLSQL 统计分析	136
7.7 通过 XLIB 统计	139

前言

本书面向的读者是 ODPS 用户，而不是 ODPS 开发人员！

我本来很希望把 ODPS 写得虎虎生威，和很多开发人员一样，我迫不及待地想深入了解这个平台及其各种牛逼的技术实现，并和大家一起分享各种前沿思想。有一次谈话，东晖和我（原话记不住了，**rephrase**）“对于一个平台而言，对用户越是透明，说明平台做得越好，用户可以更专注于自己的业务。这本书的目的是教用户怎么用 ODPS，而不是一套套设计思想，你把 ODPS 说得越牛逼，反而吓得用户不敢用了。”

因此，本书是 ODPS 使用教程，而不是开发教程。遵循这个原则，我没有过多介绍 ODPS 各种概念设计等，而是通过各种案例场景引导用户如何使用 ODPS。

< blabla..... >

本书重点通过示例来说明如何使用 ODPS 编程，写得尽量简单、明白。另外，由于是基于示例引导，它展示的仅仅是 ODPS 功能的冰山一角。你可以通过实践和在线手册了解更多。

希望本书能带给你美好的 ODPS 编程之旅！

李妹芳

于阿里

第1章 ODPS 入门

开放数据处理服务（Open Data Processing Service, ODPS）是基于飞天内核构建的海量数据处理和分析的服务平台，它以 RESTful API 形式提供服务，具有 PB 级别的数据处理能力，主要适用于实时性要求不高的海量数据处理，如数据分析、海量数据统计、数据挖掘和商业智能领域。ODPS 访问地址是：<http://www.aliyun.com/product/odps/>。

1.1 主要功能

ODPS 提供了基于云计算的海量数据分析和处理。它提供了以下几大功能模块：

1. **MapReduce (MR)** 编程模型，熟悉 Hadoop 的开发人员可以轻松转到 ODPS 平台，快速开发高效强大的数据处理应用。
2. **SQL** 计算和存储过程，采用类似传统 SQL 语法，传统数据库 SQL 程序员可以轻松入门，使用 ODPS SQL 对数据进行查询分析。ODPS 存储过程支持使用变量、判断语句和循环，适用于较复杂的 SQL 查询逻辑。
3. **DTTask** 和 **Tunnel** 服务，DTTask 适用于集群内和集群之间大数据交换，而 Tunnel 服务适用于各异构数据源（如 mysql、oracle 和本地数据）以及不同的数据同步工具（如 DataX 和 TT）和 ODPS 的数据交互。
4. **Xlib**，提供一系列基于 ODPS 平台的数据挖掘算法，包含统计分析和机器学习算法，适用于对性能要求很高的 BI 数据挖掘。
5. **图 (Graph)** 编程模型，它是面向迭代的分布式图处理框架，支持类似 Google Pregel 的 Java 编程接口，用户可以基于 Graph 编程模型开发高效的机器学习算法或数据挖掘算法。
6. **RODPS**，为了利用 R 工具丰富的算法包和强大的数据可视化功能，ODPS 提供了 RODPS 包，在 R 中安装 RODPS 包后，可以把 R 建模结果自动以 SQL 形式发布到 ODPS 中运行，这样可以充分利用 ODPS 的计算能力。

1.2 基本概念

ODPS 的数据存储单元按粒度划分，依次包括用户空间（Project）、表（Table）和分区（Partition）。

1.2.1 用户空间 Project

用户空间 Project（有时也称项目）是 ODPS 的基本组织单元，它很类似传统数据库的 Database 或 Scheme 的概念，它是进行多用户隔离和访问控制的主要边界。不同用户在独立的用户空间工作，并对其中所有的对象（包括表等）进行管理。此外，它还可以避免命名冲突，不同用户空间下的对象可以有相同的名字。

1.2.2 表 Table

表是 ODPS 的数据存储单元，类似于关系数据库中的表，它在逻辑上也是由行和列组成的二维结构，每行代表一条记录，每列表示相同数据类型的一个字段，一条记录可以包含一个或多个列，各个列的名称和类型构成这张表的 **schema**，比如一条记录包含以下字段

- `user_id` BIGINT，标识唯一用户 ID
- `view_time` BIGINT¹，表示页面访问时间戳
- `page_url` STRING，页面 URL
- `referrer_url` STRING，来源 URL
- `IP` STRING，请求访问的机器 IP

1.2.3 分区 Partition

分区 **Partition** 是指一张表下，根据分区字段（一个或多个组合）对数据存储进行划分。也就是说，如果表没有分区，数据是直接放在表所在的目录下；而如果表有 **Partition**，每个 **Partition** 对应表下的一个目录，数据是分别存储在不同的分区目录下。比如，假设前面给出的各个字段创建一张表，表名为 `page_view`，指定其分区字段为 `dt`（日期）和 `country`，则对于分区 `dt=20130101,country=US` 下的数据，就会存放在 `page_view/dt=20130101/country=US/` 的目录下。分区的最大好处在于可以加快查询，比如要查找满足 `dt=20130101` 且 `country=US` 的数据，只需要扫描相应的分区即 `dt=20130101/country=US/` 目录下的数据即可²；如果没有分区，则需要扫描表 `page_view/` 下的所有数据。

1.2.4 资源 Resource

资源（**Resource**）是 ODPS 特有的概念，用户可以上传本地自定义的 **JAR** 包或文件作为资源，也可以将 **Project** 下的某张表作为资源。比如，把 **UDF**、**MapReduce** 生成的 **JAR** 包，上传本地文件、字典表等。也许这个概念你还是难以理解，别着急，我们将在后面使用时更多描述它。

1.2.5 数据类型

ODPS 支持以下 5 种原始数据类型：

- **BIGINT**，8 字节有符号整型
- **BOOLEAN**，布尔型，包括 **TRUE/FALSE**

¹ `user_id` 表示列名称，**BIGINT** 表示类型，后面类似。另外，在示例介绍中，我参照了 **Hive tutorial** 的示例（<https://cwiki.apache.org/confluence/display/Hive/Tutorial>），这主要是便于那些熟悉 **Hive** 的同学通过对比，更容易切换到 **ODPS SQL**。在示例介绍中，并不假设你了解 **Hive**，但有一定 **SQL** 基础，零 **Hive** 经验也可以通过本教程很快入门 **ODPS SQL**。

² 这个表述不是很严谨，实际上，ODPS 并没有严格限定 `dt=20130101` 且 `country=US` 的数据一定在 `dt=20130101/country=US/` 目录下，这是由用户保证的，一般来说，没有理由不这么做。试想一下，把“`dt=20130101` 且 `country=US`”的数据不导入分区 `dt=20130101/country=US/` 下，而导入到 `dt=20130101/country=CHN/` 目录下，除了造成混乱给自己添麻烦外，能有什么好处呢？

- **DOUBLE**，8 字节双精度浮点数
- **STRING**，字符串，需要注意在 ODPS 中的函数会假设 **STRING** 中存的是 UTF8 编码的字符串，其他编码格式可能会导致异常。
- **DATETIME** 日期类型，格式如 **YYYY-MM-DD HH:mm:ss**，如 **2012-01-02 10:09:25**
- ODPS 不支持如 **Array** 这样复杂的数据类型。

1.3 客户端

ODPS 的客户端有以下几种形式：

1. **Web**：以 **RESTful API** 的形式提供离线数据处理服务；
2. **ODPS SDK**：对 **RESTful API** 的封装，目前提供 **Java** 版本的实现。
3. **ODPS CLT(Command Line Tool)**：也称 **ODPS Console**，它是运行在 **Windows/Linux** 下 客户端工具，通过 **CLT** 可以很方便执行各种命令，比如执行 **Project** 管理、**DDL**、**DML** 等操作。

批注 [a1]: 该图要改

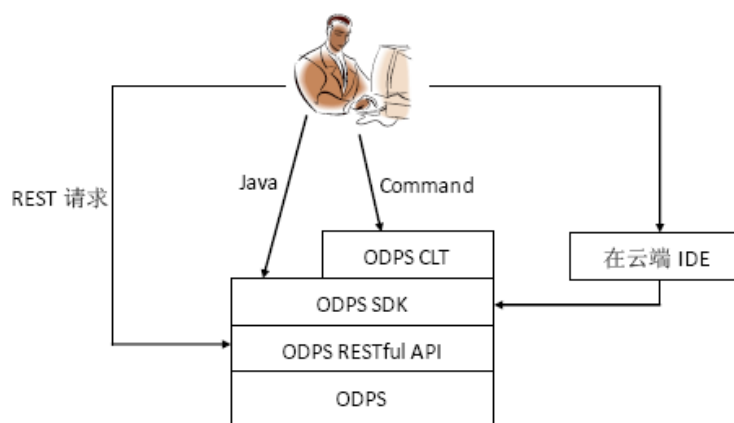


图 1-2 ODPS 应用开发模式

如图所示，ODPS 以 **RESTful API** 方式对外提供服务，用户有三种方式来使用 ODPS 的服务：通过 **REST** 请求访问 **ODPS RESTful API**、**ODPS CLT (Command Line Tool)**和 **ODPS SDK**。

其中，**ODPS SDK** 是对 **ODPS RESTful API** 的封装，但并非一一映射关系，而是提供了更高层次的抽象，以便于用户理解并运用 ODPS 中的概念。**ODPS SDK** 按功能划分了 **tables**、**jobs**、**resources**、**authorization**、**registration** 等包。**ODPS CLT** 是基于 **SDK** 开发的一个 **Windows/Linux** 命令行工具，用户可以命令的方式运行作业。在云端为阿里巴巴内部的一个 **IDE** 产品，通过 **SDK** 访问 **ODPS** 服务。

1.4 使用 ODPS 的准备工作

在使用 ODPS 服务之前，首先需要创建账号，然后要创建一个用户空间（Project）³，如果是使用已有的 Project，则需要申请该 Project 的访问权限。最后，需要通过某个客户端来访问 ODPS 服务。下面咱们先一起来完成这些准备工作吧。

1.4.1 创建账号

为了使用 odps 服务，用户首先需要到 <http://www.aliyun.com> 申请注册账号并获取密钥⁴。一个密钥实际上是个安全加密对：用户名（AccessID）和密码（AccessKey），一个账户可以有多个密钥。对密钥授权后，可以执行相应权限的操作，比如创建表、sql 查询等。创建账户并获取密钥的步骤如下：

1. 登录 <http://www.aliyun.com> 网站，点击“免费注册”。



图1-1. 登录网站

2. 填写注册信息，输入手机号和短信校验码，点击“同意协议并注册”。

图1-2 填写注册信息

注册成功后，会发送邮件到填写的电子邮箱，需要在48小时内点击邮件中的链接才能完成注册。

³ 注：用户空间即 odps 的 Project（项目），它类似于传统数据库的 Database 的概念。

⁴ 注：由于 <http://www.aliyun.com> 网站页面会不时更新，以下截图可能会和你看到的页面不完全一致，但笔者相信这难不倒你。

3. 登录邮箱，激活账户。



图1-3 激活账户

4. 登录aliyun.com，选择“去用户中心看看”



图1-4 查看用户中心

5. 点击“安全认证”，单击“创建Access Key”

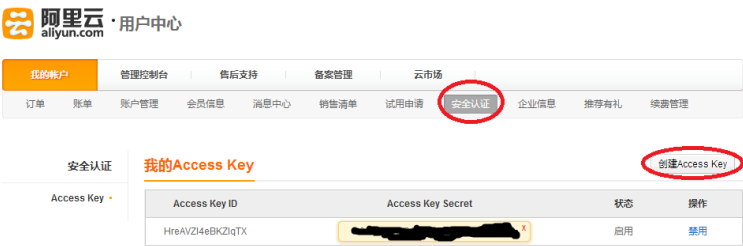


图1-5 创建密钥

则会生成如图所示的 AccessID 和 AccessKey，多次点击“创建 Access Key”可以生成多对密钥，可以分别对这些密钥授权，通过授权后的密钥可以执行相应权限的操作,请妥善保管。

1.4.2 创建 Project

在 aliyun.com 注册后，可以付费购买服务，购买之后就可以创建 ODPS Project 了。（待

补充)

创建 **Project** 后,就成为该 **Project** 的 **Owner** 用户,系统会为生成对应的云账号、**accessId** 和 **accessKey**。稍后我们将会介绍如何使用 **accessId** 和 **accessKey** 配置客户端。

注意,成为 **Project** 的 **Owner** 意味着该 **Project** 内的“所有东西”都是你的,在你给别人授权之前,任何人都无权访问你的空间。

1.4.3 授权

你可以给别的用户授权,这样他就可以访问你的 **Project**,一起工作了。

(待补充)

1.5 ODPS CLT

ODPS CLT 是 ODPS 官方提供的访问 ODPS 的客户端工具,也是目前使用最广泛的。下面我们简单介绍一下,后面各个章节的命令运行都离不开它。

1.5.1 下载和配置

1) 下载

首先,我们下载 ODPS CLT 的安装包⁵,并解压,执行命令如下:

```
wget http://odps.alibaba-inc.com/download/odps_clt_release_64.tar.gz .
mkdir clt
tar zxvf odps_clt_release_64.tar.gz -C clt
```

2) 配置

现在,我们把 **Project** 名称、账号 **accessId** 和 **accessKey** 的值配置到 **odps clt** 的 **odps_config.ini** 中,如下:

```
project_name=odps_book
access_id=*****
access_key=*****
end_point=http://service.odps.aliyun-inc.com/api
```

配置后,运行 **odpscmd**,并运行测试命令“**show tables;**”,结果显示如下:

⁵ 注: ODPS CLT 的下载地址在: <http://odps.alibaba-inc.com/download/>

```

[admin@localhost ~]$ bin/odpscmd
Aliyun ODPS Command Line Tool
Version 1.0
Copyright 2012 Alibaba Cloud Computing Co., Ltd. All rights reserved.
odps@ odps_book>show tables;

ID = 20130911043226255gaav2984
Log view:
http://webconsole.odps.aliyun-inc.com:8080/logview/?h=http://service.odps.aliyun-inc.com/api&p=odps_book&i=20130911043226255gaav2984
odps@ odps_book>

```

可以执行“h;”命令查看 odpscmd 帮助。

注：在 `odps_config.ini` 中文件中，也可以不配置 `project_name` 的值，这样运行时，需要先执行命令 `use odps_book`；。当一直使用某个 Project 时，在 `odps_config.ini` 直接配置 `project name` 使用更方便。

1.5.2 运行模式

odpscmd 支持两种运行模式，一是交互模式，如上面运行的 `show tables;` 命令，二是命令行模式，如 `clt/bin/odpscmd -e "show tables;"`。命令行模式还支持把要执行的命令写到文件中，然后调用 `-f` 来执行，如 `clt/bin/odpscmd -f console/sql.txt`。

一般来说，如果通过脚本调用时，通过命令行模式会比较方便。通过命令行模式执行，可以连续串行执行多个命令，这些命令用分号“;”分隔，如 `clt/bin/odpscmd -e "show tables; desc page_view;"`，如果是多条命令，常见的方式是把这些命令保存到文件中，通过 `-f` 方式来调用。

由于在后面的各个章节中，会介绍相应功能的各个命令示例，所以这里暂时不写。你也可以查看 [ODPS CLI 在线手册6](#) 了解更多。

⁶ http://odps.alibaba-inc.com/doc/prddoc/odps_clt/index.html

第2章 ODPS Tunnel

ODPS Tunnel 是 ODPS 提供的通道服务,支持各种异构数据源和 ODPS 之间的数据交互。它是 ODPS 数据对外的统一通道,提供高吞吐、持续稳定的服务。

Tunnel 也提供了 Restful Web Services 接口。此外，它还提供了 Java SDK，可以方便用户编程。下面，我们将通过示例来说明如何通过 Tunnel Java SDK 上传下载数据。

2.1 开发前的准备工作

2.1.1 配置开发环境

用户可根据自身情况，选择Eclipse或其他工具作为开发环境。由于Eclipse是使用最广泛的Java集成开发环境，下面我们将介绍如何使用Eclipse，基于Tunnel Java SDK实现上传下载：

- 1) 创建Java工程，例如tunnel_example;
- 2) 首先，需要下载Tunnel SDK⁷，下载解压缩后，参照“配置环境”一节，把s-tunnel-sdk-1.0.jar和lib下的jar文件配置到Eclipse环境的Build Path中，步骤：右击tunnel_example,点击Build Path->Configure Build Path, 点击Java Build Path -> Libraries -> Add External JARs, 选中odps-tunnel-sdk-1.0.jar和lib目录下的所有jar文件，如图1-1所示，点击OK后，Eclipse环境就配置好了。

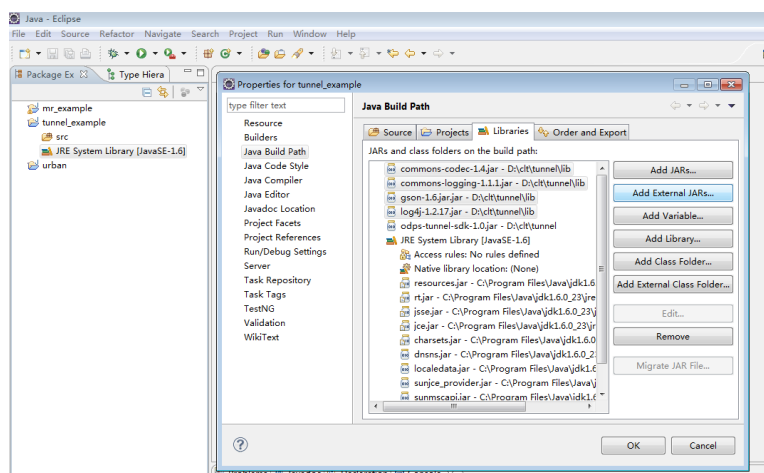


图1-1 一个配置好的项目图

⁷ 下载地址: <http://odps.alibaba-inc.com/download/>

为了便于开发，建议在环境中关联 ODPS MR 的 Java Doc，这样在开发时可以显示引用的 SDK 中类的定义，其好处你懂的。比如，在以上 `tunnel_example` 项目中，关联步骤是：右击 `tunnel_example`，点击最下方的 `Properties`，点击 `Java Build Path` -> `Libraries`，点击 `odps-tunnel-sdk-1.0.jar` -> `Javadoc location`，点击 `Edit`，在 `Javadoc location path` 中输入 http://odps.alibaba-inc.com/doc/prddoc/odps_tunnel/api/，点击 `Validate`，点击 `OK`，就关联好 Java Doc 了。如图 1-2 所示：

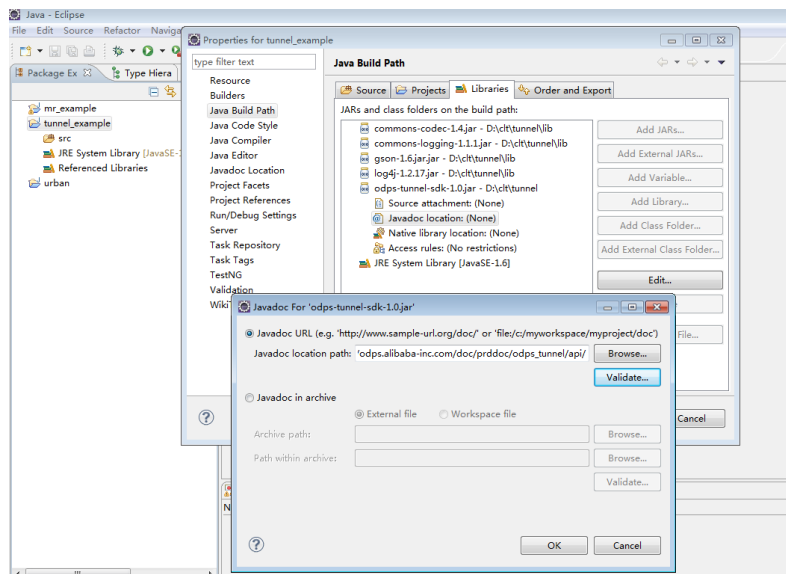


图1-2 关联 ODPS Tunnel JavaDoc

环境配置好了，下面我们通过示例，一起来探讨如何基于 Tunnel SDK 实现相关功能。

2.2 上传数据

“我想利用ODPS进行数据处理和分析，但是数据不在ODPS上，怎么办？”别着急，基于 Tunnel SDK，可以很方便实现数据上传下载功能。

下面，我们先基于一个小小的数据文件入手，了解如何使用Tunnel SDK上传数据。

2.2.1 准备数据

为了方便准备数据⁸，我们从本地文件开始。假设数据文件为`mr_wc_example.txt`，其内容如下：

```
Welcome to ODPS.  
This is a small file for ODPS MapReduce WordCount example.
```

⁸ 该示例数据同时用于 ODPS MapReduce Tutorial 中介绍 ODPS MR 的入门示例 WordCount

ODPS is an Open Data Processing Service. Enjoy it.

现在，先把该数据文件导入到ODPS表中。首先，创建一张ODPS表， sql语句如下：

```
create table mr_wc_in(word string);
```

我们要把文件的所有内容都导入到字段word中。

2.2.2 代码实现和分析

下面，我们基于 Tunnel SDK 中给出的 `sample` 代码⁹，实现 `FileUpload.java` 程序，把数据文件导入到 ODPS 的 `mr_wc_in` 表中，在上传前还做一些处理，把换行替换成空格。`FileUpload.java` 的代码清单如下（这里我们隐去了 `accessId` 和 `accessKey` 的值，请使用你自己的）：

```
package example.upload;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import com.alibaba.odps.tunnel.Column;
import com.alibaba.odps.tunnel.Configuration;
import com.alibaba.odps.tunnel.DataTunnel;
import com.alibaba.odps.tunnel.RecordSchema;
import com.alibaba.odps.tunnel.TunnelException;
import com.alibaba.odps.tunnel.Upload;
import com.alibaba.odps.tunnel.Upload.Status;
import com.alibaba.odps.tunnel.io.Record;
import com.alibaba.odps.tunnel.io.RecordWriter;

public class FileUpload {
    private static String endpoint = "http://dt.odps.aliyun-inc.com";
    private static String accessId = "*****";
    private static String accessKey = "*****";
    private static String project = "odps_book";
```

⁹ 在 Tunnel SDK 解压缩的 `samples` 目录下

```
public static void main(String args[]) {
    if (args.length != 2) {
        System.err.println("Usage: FileUpload <file> <table>");
        System.exit(2);
    }
    String filename = args[0];
    String table = args[1];

    Configuration cfg = new Configuration(accessId, accessKey, endpoint);
    DataTunnel tunnel = new DataTunnel(cfg);

    BufferedReader br = null;
    try {
        // get content
        String content = "";
        String line = "";
        br = new BufferedReader(new FileReader(filename));
        while ((line = br.readLine()) != null) {
            content += line + " ";
        }
        Upload up = tunnel.createUpload(project, table, "");
        String id = up.getUploadId();
        System.out.println("UploadId = " + id);
        Status status = up.getStatus();
        System.out.println("Create Upload Status is: " + status.toString());

        RecordSchema schema = up.getSchema();
        System.out.println("Schema is: " + schema.toJsonString());
        Long blockid = (long) 0;
        RecordWriter writer = up.openRecordWriter(blockid);
        Record r = new Record(schema.getColumnCount());
        if (schema.getColumnType(0) == Column.Type.ODPS_STRING) {
            r.setString(0, content);
        }
        else
        {
            throw new RuntimeException("invalid column type");
        }
    }
}
```



```
        writer.write(r);
        writer.close();

        Long[] blocks = {blockid};
        up.complete(blocks);
    } catch (TunnelException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (br != null)
                br.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

下面，我们来分析一下这块代码。该上传示例主要包括以下几个步骤：

- 1) 创建 **Configuration** 和 **Tunnel** 实例，
- 2) 读取本地文件内容
- 3) 通过 **createUpload()** 函数创建一个 **Upload** 实例，服务端会相应创建一个 **Upload Session** 连接¹⁰，通过 **getStatus()** 查看是否创建成功
- 4) 通过 **getSchema()** 函数，读取在创建上传时服务端返回的 ODPS 表 **Schema**，该 **Schema** 用于后续上传操作时按照数据类型进行转换
- 5) 通过 **openRecordWriter()** 打开一个 **RecordWriter**，该函数的接收参数 **blockid** 唯一标识上传到哪个 **block**。**openRecordWriter()** 调用会创建一个 **Request** 请求连接，相当于打开从客户端到服务端的数据通道。
- 6) 创建一个 **Record** 对象，根据 **Schema** 设置记录中的字段值
- 7) 调用 **write()** 函数写 **Record**，一次 **write()** 调用只写一条 **Record**，**write()** 函数会对 **Record** 进行序列化，并加上 **checksum** 一起发送到网络；服务端接收到数据后，会执行反序列化，校验 **checksum**，确保数据在网络传输中没有丢失，保证数据的正确性。

¹⁰ 注：说明一下 **Session** 和 **Request** 的概念：整个上传下载过程称为一个 **Session**；在一个 **Session** 中，每个 **block** 的上传请求，称为一个 **Request**；一个 **Session** 包含一个或多个 **Request**。

服务端是把数据写到临时目录。

- 8) 调用 `close()`，发送所有数据的 `checksum` 以及总的记录数，服务端会对 `checksum` 和总记录数进行校验，返回 `Response`。
- 9) 调用 `complete()` 函数，该函数的参数是（客户端认为上传成功的）`block` 列表，服务端会校验 `block` 列表是否都上传成功，如果成功，则把临时目录的数据 `move` 到最终目录，即放到结果表所在目录。

实际上，`Tunnel SDK` 中提供了 `FileUploader` 类，对本地文件上传过程进行封装，很方便，有兴趣可以试试。在这里，为了阐述上传过程中涉及的函数调用以及相关步骤说明（因为很多时候你可能不是上传本地文件，比如上传 `MYSQL`、`Oracle` 数据等），需要了解详细过程，所以尽量把各个过程都写出来。

2.2.3 如何提高并发

从客户端角度看，一个 `Upload` 对象对应一个 `Session`¹¹，一个 `RecordWriter` 对象即一个 `Request`。在一个 `Session` 中，可以（通过多线程）创建多个 `RecordWriter`，实现并发上传不同 `block` 数据。

在上面的代码实现中，通过 `getUploadId()` 获取到了 `UploadId`，但没有用它。实际上，为了实现高并发上传，常见的做法是获取到 `UploadId` 后，启动多个进程，每个进程通过 `UploadId`，调用构造函数

```
Upload(Configuration conf, String project, String table,
        String partition, String uploadId)
```

来构建自己的 `Upload` 实例，这些 `Upload` 实例都对应同一个 `Session`，每个 `Upload` 再分别上传不同 `block` 的数据。通过这种方式，可以实现多个进程（且每个进程又包含多个线程）的方式高并发上传数据。

2.2.4 注意事项

上传数据时，客户端需要注意以下几点（也是经常被提及的）：

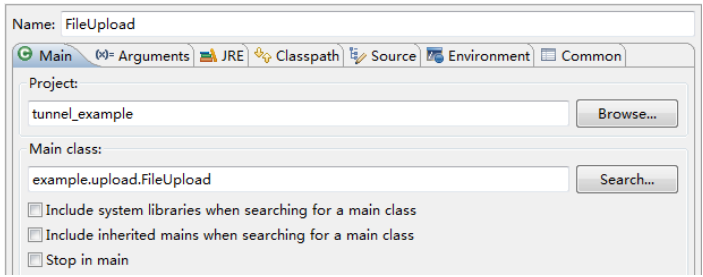
- 1) `Tunnel` 上传数据只支持 `Append` 模式
- 2) 一个 `Session` 的有效期是 24 小时，如果超时，该 `Session` 下的所有操作都是无效的
- 3) `blockid` 的取值范围目前是 `[0,20000)`，这个范围线上可能会根据情况进行调整。
- 4) 一个 `blockid` 数据上传上限是 100GB（序列化后的）
- 5) 同一个 `blockid` 可以重复上传，有效数据是最后一次成功 `upload` 的数据
- 6) 由于上传数据是先写到临时目录，最后确定成功后（客户端调用 `complete` 函数）才写到结果目录，所以在上传过程中，客户端或服务端挂掉，都不会污染结果表原始数据

¹¹ 这个说法不太准确，实际上，相同 `UploadId` 的多个 `Upload` 实例都对应用一个 `Session`，详见下面的分析

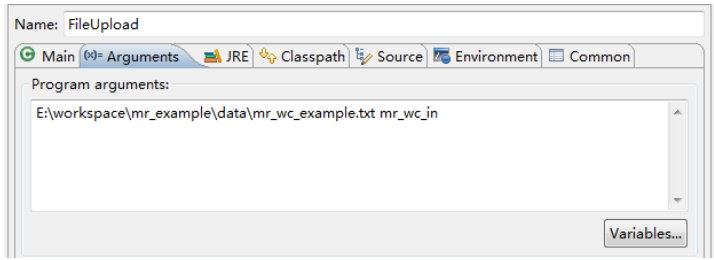
2.2.5 运行和结果输出

2.2.5.1 在 Windows 上运行

客户端程序可以直接在本地运行，右击FileUpload.java，点击Run as -> Run Configurations，在Main标签栏中配置如下：



在Arguments标签页中输入参数本地数据文件地址和ODPS结果表，如下：



点击Run，即可运行。

通过 c!t，查看表结果，输出如下：

```
odps@ odps_book>read mr_wc_in;
+-----+
| word   |
+-----+
| Welcome to ODPS. This is a small file for the ODPS MapReduce WordCount example. ODPS
is an Open Data Processing Service. Enjoy it. |
+-----+
```

可以看到，数据已经导入到 ODPS 表中了。

2.2.5.2 在 Linux 上运行

在某些情况下，在 PC 机上执行时，由于网络不给力或其他原因，在本地运行可能会很慢，所以你可能希望在服务器上运行。因此，这里也介绍一下如何在 Linux（服务器）上运行。

首先，需要把程序生成 Jar 包，可以通过 Eclipse 的 Export 生成 Jar 包，也可以通过 ant 来生成。这里我们 Eclipse 生成 jar 包：tunnel_fileupload.jar，然后上传到 Linux 机器上。当然，也准备好数据文件和 SDK 相关 jar 包。

然后，执行如下命令：

```
java -cp tunnel_fileupload.jar:tunnel_sdk/odps-tunnel-sdk-1.0.jar:tunnel_sdk/lib/commons-cod
ec-1.4.jar:tunnel_sdk/lib/commons-logging-1.1.1.jar:tunnel_sdk/lib/gson-1.6.jar.jar
example.upload.FileUpload /home/admin/book/mr/data/mr_wc_example.txt mr_wc_in
```

（注意，在执行前先 drop 表再创建¹²，否则由于 Tunnel 是 Append 模式，会生成两条记录）
现在，数据导入 ODPS 了，可以对它进行处理分析，比如单词计数等。

2.3 下载数据

假设对导入的数据进行单词进行计数，生成结果表如下：

```
odps@ odps_book>read mr_wc_out;
```

word	count
Data	1
Enjoy	1
MapReduce	1
ODPS	3
Open	1
Processing	1
Service	1
This	1
Welcome	1
WordCount	1
a	1
an	1
example	1
file	1
for	1
is	2
it	1
small	1
the	1

¹² 注：odps sql 不支持 truncate 对表进行清空

to	1	
+-----+-----+		

现在，怎么导出结果表呢？还是让 Tunnel SDK 来帮你吧。

2.3.1 代码实现和分析

下面我们给出 FileDownload.java 的完整程序清单，它实现导出结果表到本地文件。

```
package example.download;

import java.io.File;
import java.io.FileOutputStream;

import com.alibaba.odps.tunnel.Configuration;
import com.alibaba.odps.tunnel.DataTunnel;
import com.alibaba.odps.tunnel.Download;
import com.alibaba.odps.tunnel.io.Record;
import com.alibaba.odps.tunnel.io.RecordReader;
import com.alibaba.odps.tunnel.io.TextRecordWriter;

public class FileDownload {
    private static String endpoint = "http://dt.odps.aliyun-inc.com";
    private static String accessId = "*****";
    private static String accessKey = "*****";
    private static String project = "odps_book";
    private static char colSeparator = '\t';
    private static char recordSeparator = ',';

    public static void main(String args[]) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: FileDownload <file> <table>");
            System.exit(2);
        }
        String fileName = args[0];
        String table = args[1];
        File file = new File(fileName);
```

```

Configuration cfg = new Configuration(accessId, accessKey, endpoint);
DataTunnel tunnel = new DataTunnel(cfg);

Download down = tunnel.createDownload(project, table, "");

RecordReader reader = down.openRecordReader(0L, down.getRecordCount());
FileOutputStream out = new FileOutputStream(file);
TextRecordWriter writer = new TextRecordWriter(out, down.getSchema(), colSeparator,
recordSeparator);
Record r = null;
while ((r = reader.read()) != null) {
    writer.write(r);
}
reader.close();
writer.close();
down.complete();
}
}

```

前面我们详细介绍了上传的各个接口，理解之后，对这个下载示例就很容易明白了。这里重点指出几点区别：

1. 调用 `createDownload()` 创建下载时，Tunnel Server 会对要读的 ODPS 表构建索引，返回总记录数，可以通过 `getRecordCount()` 函数获取。
2. 在调用 `openRecordReader()` 创建 Request 请求连接时，需要指定起始位置 and 要下载的记录数，起始位置的起始值是 0。
3. 这里，我们调用了 `TextRecordWriter` 来写文件，它对根据 schema 字段类型进行转换做了封装，对于下载到本地文件，使用上比较方便。类似地，Tunnel SDK 也提供了 `TextRecordReader` 类。
4. 对于下载而言，`complete()` 函数不需要传递参数。

2.3.2 如何提高并发

对于下载而言，如果希望提高并发，可以对 `getRecordCount()` 函数获取到的总记录数进行均匀切分，通过多进程和（或）多线程，每个进程/线程只下载一个区间范围的数据，各个区间之间连续的方式提高并发。

启动多个进程的方式和上传类似。在创建下载后，可以通过 `getDownloadId()` 获取 `DownloadId`，启动多个进程，每个进程通过 `DownloadId`，调用构造函数

```

Download(Configuration conf, String project, String table,
String partition, String downloadId)

```

来构建自己的 `Download` 对象, 这些 `Download` 对象都对应同一个 `Session`, 每个 `Download` 对象再分别下载不同区间的数据。

2.4 核心接口

在前面的示例说明中, 我没有详细介绍各个接口。为了帮助用户更好地使用 Tunnel SDK 编程, 下面将简要介绍一下 Tunnel SDK 的一些核心接口, 熟悉这些接口可以帮助你更好地驾驭它。为了避免过于冗长而枯燥, 这里只给出一些关键方法, 详细的接口说明请参看 ODPS Tunnel Java Docs¹³。

2.4.1 Configuration

```
public class Configuration {

    public static int DEFAULT_CHUNK_SIZE = 1500 - 4;

    public static int DEFAULT_SOCKET_CONNECT_TIMEOUT = 180; // seconds

    public static int DEFAULT_SOCKET_TIMEOUT = 300; // seconds


    public Configuration(String accessId, String accessKey, String endpoint);

    public String getAccessId();

    public void setAccessId(String accessId);

    public String getAccessKey();

    public void setAccessKey(String accessKey);

    public URI getEndpoint();

    public void setEndpoint(URI endpoint);

    public int getChunkSize();

    public void setChunkSize(int chunkSize);

    public int getSocketConnectTimeout();

    public void setSocketConnectTimeout(int timeout);

    public int getSocketTimeout();

    public void setSocketTimeout(int timeout);

    public void loadFromClasspath(String resource);

}
```

¹³ 注: 访问地址 http://odps.alibaba-inc.com/doc/prddoc/odps_tunnel/api/index.html

Configuration 是 ODPS Tunnel 的配置信息，需要通过 `accessId`（云账号 ID）、`accessKey`（云账号密钥）、`endpoint`（ODPS Tunnel 服务 URL）来创建。它提供一组 `get/set` 方法，除了可以设置或获取 `accessId`、`accessKey` 和 `endpoint` 信息外，还可以获取网络传输的数据块大小、底层 Socket 超时时间等。

2.4.2 DataTunnel

```
public class DataTunnel {  
  
    public DataTunnel(Configuration cfg);  
  
    public Upload createUpload(String project, String table, String partition);  
  
    public Upload createUpload(String project, String table, String partition,  
                               String uploadId) throws TunnelException, IOException;  
  
    public Download createDownload(String project, String table, String partition  
                                   ) throws TunnelException, IOException;  
  
    public Download createDownload(String project, String table, String partition,  
                                   String downloadId) throws TunnelException, IOException;  
  
}
```

DataTunnel 是访问 ODPS Tunnel 服务的总入口类，它是通过前面介绍的 Configuration 创建的。DataTunnel 类提供了创建上传和下载对象的方法，需要特别注意的是，`createUpload` 还提供了通过 `uploadId` 的方法来创建 Upload 实例，`uploadId` 相同的不同实例都是表示同一个 Session。在前面的示例介绍“如何提高并发”一节中，我们已经提到了可以在多个进程中通过这种调用方式，提高数据上传的并发性。下载也类似。

DataTunnel 的生命周期在创建之后一直到程序运行结束。

2.4.3 Upload

```
public class Upload {  
  
    public static enum Status {  
        UNKNOWN, NORMAL, CLOSING, CLOSED, CANCELED, EXPIRED, CRITICAL  
    }  
  
    public Upload(Configuration conf, String project, String table,  
                  String partition) throws TunnelException, IOException;  
  
    public Upload(Configuration conf, String project, String table,  
                  String partition, String uploadId) throws TunnelException, IOException;  
  
    public RecordWriter openRecordWriter(long blockId) throws TunnelException, IOException;  
  
    public void complete(Long[] blocks) throws TunnelException, IOException;  
  
    public void abort() throws TunnelException, IOException;  
  
}
```



```
public void erase();  
public String getUploadId();  
public RecordSchema getSchema();  
public Status getStatus() throws TunnelException, IOException;  
public Long[] getBlockList() throws TunnelException, IOException;  
}
```

Upload 是上传数据的入口类，其相关的逻辑处理包含以下几个方面：

1) 创建 Upload

- a) 请求方式：同步
- b) 可以通过 Upload 构造方法来创建 Upload 实例，也可以通过前面介绍的 DataTunnel 来创建 Upload 实例，其本质是一样的。Tunnel Server 会相应创建一个 Session，生成 uploadId 唯一标识该 Session，通过同一个 uploadId 创建的 Upload 实例都对应同一个 Session，这可以用于提高并发性。

2) 上传数据

- a) 请求方式：异步
- b) 调用 openRecordWriter()，创建一个 Request 请求连接，相当于打开从客户端到服务端的数据通道，其接收参数 blockId 唯一标识上传的数据以及上传的数据在整个表中的位置。当该 blockId 的数据上传失败时，可以只对该 block 重新上传，而不需要重新对整张表上传。

3) 查看上传

- a) 请求方式：同步
- b) 通过 getStatus()方法可以查看 Upload 的状态，可以用过 getBlockList()方法获取成功上传的 block 列表，客户端可以对获取到的 block 列表和自己实际上传的 block 列表进行对比，如果不一致，则说明存在 block 上传失败，可以根据 blockid 对失败的 block 重新上传。比如，客户端获取到的成功上传 block 列表是[0,1,2,3,5]，而实际上传的 block 列表是[0,1,2,3,4,5]，则可以判读 blockid=4 的 block 上传失败，可以对它单独重新上传。此外，获取到的成功上传 block 列表还用于结束上传时（调用 complete 方法），发送给服务端进行验证，见后面的结束上传相关说明。

4) 结束上传

- a) 请求方式：同步
- b) 调用 complete(Long[] blocks)方法，参数 blocks 列表表示已经成功上传的 block 列表，服务端会对该列表进行验证，看这些 blocks 是否真正上传成功，如果验证成功，服务端会把这些 blocks 从临时目录 mv 到结果表所在目录。

5) 取消上传

- a) 请求方式：同步

-
- b) 调用 `abort()`取消上传，这样已经上传的数据还没有 mv 到结果表所在目录，这些数据无效，不可用。服务端的垃圾回收机制会对临时目录执行垃圾回收。

Upload 的生命周期从创建到调用 `complete` 方法结束上传(或调用 `abort()`取消上传)。对 Upload 而言，它通过 `getStatus()`方法从服务端获取的状态包含 7 种，分别说明如下：

- 1) UNKNOWN，服务端刚创建 Session 时设置的初始值，正常情况下，该状态客户端不可见
- 2) NORMAL，创建 Upload 对象成功
- 3) CLOSING，当调用 `Complete` 方法（结束上传）时，服务端会把状态先置为 `closing`
- 4) CLOSED，完成结束上传（即把数据 mv 到结果表所在目录）后，服务端把状态置为 `CLOSED`
- 5) CANCELED，调用 `abort` 取消上传（已上传数据无效，不可用）
- 6) EXPIRED，上传超时
- 7) CRITICAL，服务出错

2.4.4 Download

```
public class Download {  
    public static enum Status {  
        UNKNOWN, NORMAL, CLOSED, EXPIRED  
    }  
  
    public Download(Configuration conf, String project, String table, String partition)  
        throws TunnelException, IOException ;  
  
    public Download(Configuration conf, String project, String table, String partition,  
        String downloadId) throws TunnelException, IOException ;  
  
    public RecordReader openRecordReader(long start, long count) throws TunnelException, IOException;  
    public void complete() throws TunnelException, IOException ;  
    public RecordSchema getSchema();  
    public long getRecordCount() ;  
    public String getDownloadId();  
    public Status getStatus() throws TunnelException, IOException;  
}
```

Download 是下载数据的入口类，其相关的逻辑处理包含以下几个方面：

- 1) 创建 Download
 - a) 请求方式：同步
 - b) 和 Upload 类似，可以通过 Download 构造方法或前面介绍的 DataTunnel 来创建 Download 实例，其本质是一样的。同样，Tunnel Server 会相应创建一个 Session，

生成 `downloadId` 唯一标识该 `Session`，通过同一个 `downloadId` 创建的 `Download` 实例都对应同一个 `Session`，这可以用于提高并发性。

2) 下载数据

- a) 请求方式：异步
- b) 调用 `openRecordWriter()`，创建一个 `Request` 请求连接，相当于打开从客户端到服务端的数据通道，其参数 `start` 标识本次下载的记录的起始位置，从 0 开始；`count` 标识本次下载的记录数。

3) 查看下载

- a) 请求方式：同步
- b) 调用 `getStatus` 可以获取当前 `Download` 状态

4) 结束下载

- a) 请求方式：同步
- b) 调用 `complete` 方法完成下载。和上传不同，对于下载而言，服务端不会做校验，由客户端自己保证。

`Download` 的生命周期从创建到调用 `complete` 方法结束。对 `Download` 而言，它通过 `getStatus()` 方法从服务端获取的状态包含 4 种，分别说明如下：

- 1) `UNKNOWN`，服务端刚创建 `Session` 时设置的初始值，正常情况下，该状态客户端不可见
- 2) `NORMAL`，创建 `Download` 对象成功
- 3) `CLOSED`，下载结束
- 4) `EXPIRED`，下载超时

2.4.5 Record

```
public class Record {  
  
    public Record(int columnCount);  
  
    public int getColumnCount();  
  
    public Long getBigint(int idx);  
  
    public Boolean getBoolean(int idx) ;  
  
    public Date getDatetime(int idx);  
  
    public long getDatetimeInMillis(int idx);  
  
    public Double getDouble(int idx);  
  
    public String getString(int idx);  
}
```

```
public void setBigint(int idx, Long value);

public void setBoolean(int idx, Boolean value);

public void setDatetime(int idx, Date value);

public void setDouble(int idx, Double value);

public void setString(int idx, String value);

public Object get(int idx);

}
```

Record 表示 ODPS 表或分区的一条记录，一条记录由多列组成，列包括列名、类型和值。

Record 支持通过列的序号（0 起始）或列名操作记录的数据。一个 Record 对象：

- 不包括分区列（Partition columns）
- 提供各种类型的 set 和 get 方法

更多关于 ODPS Tunnel 接口说明，可以查看 ODPS 在线手册的 ODPS Tunnel SDK¹⁴。

2.5 综合示例

该示例展示了如何通过获取到的上传 id，多进程并发执行。

2.6 FAQ

待补充

¹⁴ http://odps.alibaba-inc.com/doc/prddoc/odps_tunnel/api/index.html

第3章 ODPS MapReduce 入门

ODPS MapReduce（通常也称为 ODPS MR 或 MRTask）是 ODPS 提供的 Java MapReduce 编程模型。它适用于处理 ODPS 表，其有些编程接口和 Hadoop 的一致，但有些针对 ODPS 场景做了修改。ODPS MR 目标是处理 ODPS 平台上的大规模数据。

3.1 初识 ODPS MR

关于 MapReduce 编程模型及其思想的介绍，网上已经有非常多的资料了。如果你不太了解，建议 google 一下或阅读经典书籍《Hadoop - The Definitive Guide》。这里，我们只简要介绍一下 ODPS MR 特有的一些基本知识，了解它也许可以帮助你更好地编写 ODPS MR 程序。

3.1.1 输入和输出

ODPS MR 作业的输入和输出限制为结构化的表，不允许用户自定义输入输出格式，不提供类似 Hadoop 的 FileSystem 接口。

ODPS MR 的输入输出模型与 Hadoop 的区别如下：

(input) <k1, v1> -> map -> <k2, v2> -> combine -> <k2, v2> -> reduce -> <k3, v3> (output)

Hadoop MR 的输入输出模型

(input) <recordNum, record> -> map -> <k, v> -> combine -> <k, v> -> reduce -> <record> (output)

ODPS MR 的输入输出模型

在 ODPS MR 中，Mapper 的输入是 ODPS 表的记录（record）集合，<k, v>键值对只用于 map 到 reduce 的 shuffle 过程。shuffle 是 MapReduce 的核心，简单地说，它充分考虑“数据局部性（data locality）”原理，是一种把 Map Task 的输出结果有效地发送给 Reduce Task 的机制。了解 shuffle 对于 MR job 的性能调优有很大帮助¹⁵。

Mapper 可以直接生成结果记录写到输出表（这类 MapReduce 程序只包含 Mapper，称为 Map-Only），也可以生成<k, v>键值对集合，输出给 Reducer。Reducer 对每个键（key）所关联的一组数值集（values）进行规约计算，把结果记录输出到结果表中。

由于网络传输是 MapReduce 的一大瓶颈，所以在某些场景下，可以在 Mapper 和 Reducer 之间使用 Combiner，Combiner 又称 LocalReducer，负责对 Mapper 生成的中间结果（键值对）在 Mapper 端进行本地聚集，这有助于降低从 Mapper 到 Reducer 传输的数据量。

3.1.2 资源

资源（Resource）是 ODPS 特有的概念，用户可以上传本地自定义的 JAR 包或文件作为资源，也可以将 Project 下的某张表作为资源。比如，把 UDF、MapReduce 生成的 JAR 包，上传本地文件、字典表等。

¹⁵ 注：如果你想深入了解 shuffle，可以查看《Hadoop - The Definitive Guide》一书。

在集群上运行 MR 作业时，每个计算节点都会分发一份资源副本，这样就可以从本地获取数据，提高处理效率。它类似于 Hadoop 的 DistributedCache 功能。

资源是有限制的，每个资源文件不能超出 64M，单个作业资源总大小不能超出 512M。

目前支持 5 种类型的资源：file、archive、table、py 和 jar。在 clt 创建资源的命令格式分别如下：

```
add file <local_file> [as alias] [comment 'cmt'][-f];  
add archive <local_file> [as alias] [comment 'cmt'][-f];  
add table <table_name> [partition (spec)] [as alias] [comment 'cmt'][-f];  
add py <local_file.py> [comment 'cmt'][-f];  
add jar <local_file.jar> [comment 'cmt'][-f];
```

命令说明：

- -f 参数表示如果同名资源已存在，则强制覆盖，而非报错。
- as alias 参数表示给资源取个别名，如果不加该参数，则默认以资源名作为文件名。
- archive 类型的资源是通过资源名称的后缀来判断压缩类型，支持的压缩类型包括：.zip/.tgz/.tar.gz/.tar/jar。

在 clt 中，可以通过命令 `list resources`；查看当前 Project 下的所有资源。

3.2 开发前的准备工作

3.2.1 开发流程

要开发 ODPS MR 程序，建议开发流程如下：

1. 首先看是否能够用 ODPS SQL 搞定，可以的话尽量用 SQL，简单。
2. 配置环境，编写 MR 代码，使用本地运行模式进行基本测试。
3. 编写 MR 单元测试用例
4. 在集群上调试，验证结果。

后面我们将以 WordCount 为例，说明如何开发和调试。

3.2.2 配置开发环境

用户可根据自身情况，选择 Eclipse 或其他工具作为 ODPS MR 的开发环境。由于 Eclipse 是使用最广泛的 Java 集成开发环境，下面我们将介绍如何使用 Eclipse 开发 ODPS MR：

1) 创建 Java 工程，例如 mr_examples;

2) 添加 ODPS SDK 的 JAR 包（在 ODPS CLT16安装路径的 lib 目录下），需要将其中的 JAR 文件添加到 Eclipse Project 的 Build Path 中，步骤是：右击 mr_examples,点击 Build Path->Configure Build Path, 点击 Java Build Path -> Libraries -> Add External JARs,把 ODPS CLT 的 lib 目录下的所有 jar 文件全部选上，如图 1-1 所示，点击 OK 后，Eclipse 环境就配置好了。

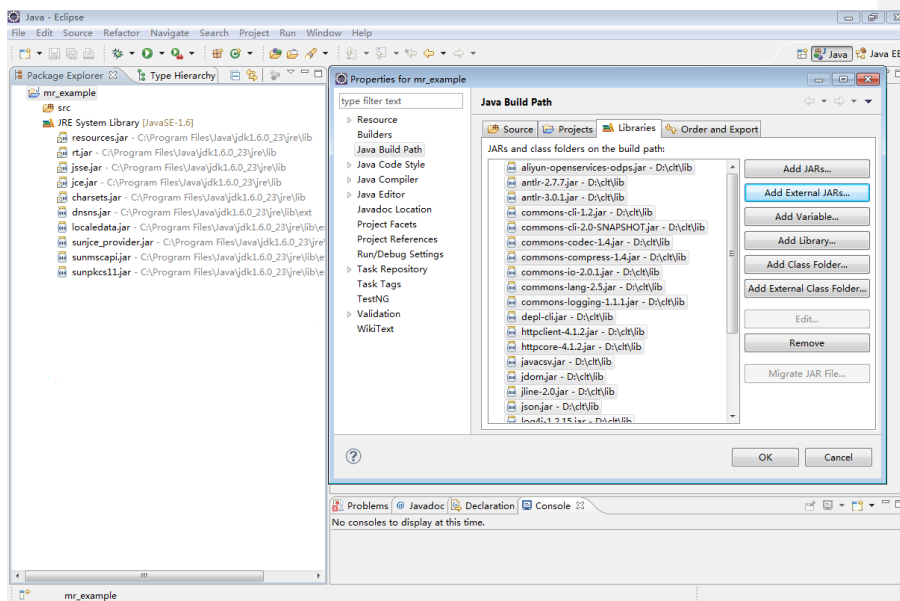


图1-1 一个配置好的项目图

为了便于开发，建议在环境中关联 ODPS MR 的 Java Doc，这样在开发时可以显示引用的 SDK 中类的定义，其好处你懂的。比如，在以上 mr_examples 项目中，关联步骤是：右击 mr_examples，点击最下方的 Properties，点击 Java Build Path ->Libraries，点击 mapreduce-api.jar -> Javadoc location,点击 Edit，在 Javadoc location path 中输入 http://odps.alibaba-inc.com/doc/prddoc/odps_mr/api/，点击 Validate，点击 OK，就关联好 Java Doc 了。如图 1-2 所示：

¹⁶ 注：odps clt 解压后有 lib 目录

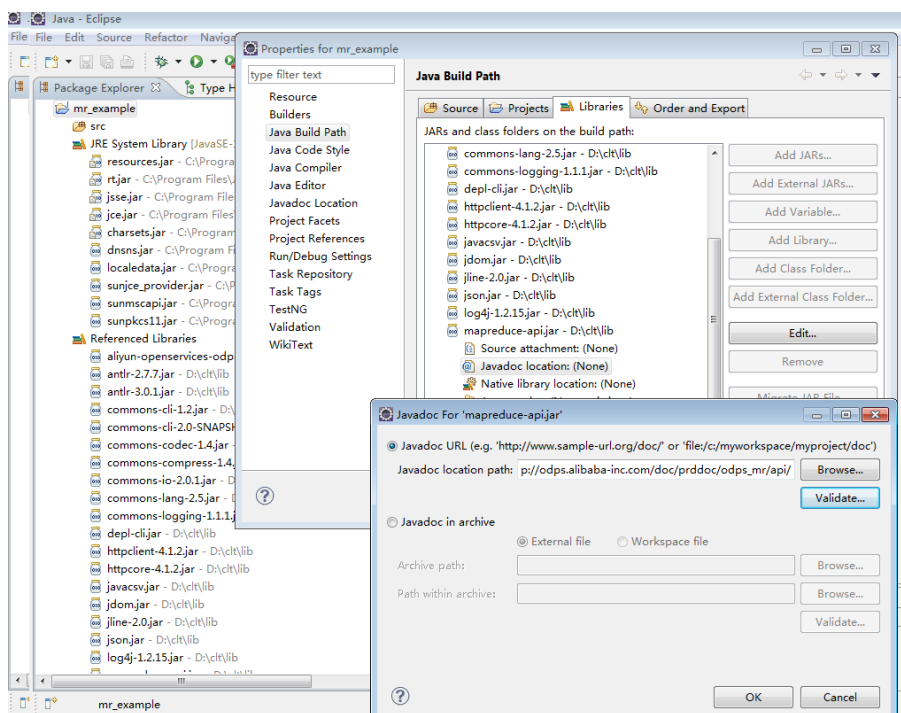


图1-2 关联 ODPS MR Javadoc

如果你曾经编写过 Hadoop MapReduce 应用代码，写 ODPS MR 会比较轻松。如果没有写过也不要紧，下面我们会从零开始，一起来探讨如何基于 ODPS MR 编程模型，实现相关功能。现在，我们一起开始动手体验吧。

3.3 入门示例 WordCount

在学习如何使用 ODPS MR 处理海量数据之前，我们先基于一个小小的数据集入手，了解 ODPS MR 的一些基本功能。WordCount 是编程人员非常熟悉的经典案例，我们就从它出发吧。

3.3.1 准备工作

本节以 WordCount 为例，计算一个文本文件中每个单词出现的次数，一起来初步体验 ODPS MR。由于 ODPS 的输入输出类型为 ODPS 表，一般而言，ODPS MR 程序都是直接处理 ODPS 表。如果该例子直接处理 ODPS 表中的数据，你在动手实践中可能会犯愁如何准备数据。因此，为了使你的 ODPS MR 体验之旅更加顺畅愉悦，我们将从准备本地文件开始。

假设数据文件为 mr_wc_example.txt，其内容如下：

```
Welcome to ODPS.
```

```
This is a small file for ODPS MapReduce WordCount example.
```

```
ODPS is an Open Data Processing Service. Enjoy it.
```

现在，先把该数据文件导入到 ODPS 表中。首先，创建一张 ODPS 表， sql 语句如下：

```
create table mr_wc_in(word string);
```

我们要把文件的所有内容都导入到字段 word 中。可以通过 ODPS Tunnel 来完成导入功能，Tunnel 是 ODPS 提供的数据进出的通道服务，它提供了 Java SDK，可以基于 SDK 开发完成数据导入。

3.3.2 通过 Tunnel 导入数据

首先，需要下载 Tunnel SDK¹⁷，下载解压缩后，参照“配置环境”一节，把 odps-tunnel-sdk-1.0.jar 和 lib 下的 gson-1.6.jar 配置到 Eclipse 环境的 Build Path 中：点击 mr_examples，点击 Build Path->Configure Build Path，点击 Java Build Path -> Libraries -> Add External JARs，选中 odps-tunnel-sdk-1.0.jar 和 gson-1.6.jar，选择 OK 后，就完成配置了。

下面，我们基于 Tunnel SDK 中给出的 sample 代码¹⁸，实现 FileUpload.java 程序，把数据文件导入到 ODPS 的 mr_wc_in 表中，FileUpload.java 的代码清单如下（这里我们隐去了 accessId 和 accessKey 的值，请使用你自己的）：

```
package example.wordcount;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import com.alibaba.odps.tunnel.Column;
import com.alibaba.odps.tunnel.Configuration;
import com.alibaba.odps.tunnel.DataTunnel;
import com.alibaba.odps.tunnel.RecordSchema;
import com.alibaba.odps.tunnel.TunnelException;
import com.alibaba.odps.tunnel.Upload;
```

¹⁷ 下载地址：<http://odps.alibaba-inc.com/download/>

¹⁸ 在 Tunnel SDK 解压缩的 samples 目录下

```
import com.alibaba.odps.tunnel.Upload.Status;

import com.alibaba.odps.tunnel.io.Record;

import com.alibaba.odps.tunnel.io.RecordWriter;


public class FileUpload {

    private static String endpoint = "http://dt.odps.aliyun-inc.com";

    private static String accessId = "***";

    private static String accessKey = "***";

    private static String project = "odps_book";


    public static void main(String args[]) {

        if (args.length != 2) {

            System.err.println("Usage: FileUpload <file> <table>");

            System.exit(2);

        }

        String filename = args[0];

        String table = args[1];


        Configuration cfg = new Configuration(accessId, accessKey, endpoint);

        DataTunnel tunnel = new DataTunnel(cfg);


        BufferedReader br = null;

        try {

            // get content

            String content = "";

            String line = "";

            br = new BufferedReader(new FileReader(filename));

            while ((line = br.readLine()) != null) {

                content += line + " ";

            }

        } catch (Exception e) {

            e.printStackTrace();

        } finally {

            if (br != null) {

                try {

                    br.close();

                } catch (Exception e) {

                    e.printStackTrace();

                }

            }

        }

        RecordWriter writer = tunnel.getWriter();

        Record record = new Record(table, content);

        writer.write(record);

    }

}
```

```
}

Upload up = tunnel.createUpload(project, table, "");

RecordSchema schema = up.getSchema();

RecordWriter writer = up.openRecordWriter(0);

Record r = new Record(schema.getColumnCount());

if(schema.getColumnType(0) == Column.Type.ODPS_STRING) {

    r.setString(0, content);

}

else {

    throw new RuntimeException("invalid column type");

}

writer.write(r);

writer.close();

Long[] blocks = {(long) 0};

up.complete(blocks);

} catch (TunnelException e) {

    e.printStackTrace();

} catch (IOException e) {

    e.printStackTrace();

}finally {

    try {

        if (br != null)

            br.close();

    } catch (IOException ex) {

        ex.printStackTrace();

    }

}

}
```

简单起见，直接在本地运行该程序，右击 FileUpload.java，点击 Run as -> Run Configurations，在 Arguments 标签页中输入参数 E:\workspace\mr_example\data\mr_wc_example.txt mr_wc_in，点击 Run，即可运行。

通过 cdt，查看表结果，输出如下：

```
odps@ odps_book>read mr_wc_in;
+-----+
| word      |
+-----+
| Welcome to ODPS. This is a small file for the ODPS MapReduce WordCount example.  ODPS
is an Open Data Processing Service.  Enjoy it.      |
+-----+
```

可以看到，数据已经导入到 ODPS 表中了，这里我们把换行替换成了空格。由于本章的重点在于讨论 ODPS MR，所以这里不多描述如何使用 Tunnel SDK。

3.3.3 通过 MR 实现单词计数

现在，我们要实现 MR 程序 WordCount.java，对 mr_wc_in 表的 word 字段下的文本段内容进行单词计数。在 Mapper 中，通过 Java 的 StringTokenizer 对文本切分成各个单词，把每个单词计数值设为 1，对于之前导入的数据样本，输出<每个单词，1>这样的 <key,value>列表，如下：

```
Welcome, 1
to, 1
ODPS, 1
This, 1
is, 1
a, 1
small, 1
file, 1
for, 1
the, 1
ODPS, 1
MapReduce, 1
...
```

在 Reducer 中，对相同 key 进行合并，对其计数值累加，把结果值赋给 Record，再把 Record 输出到表中。举个例子，在上面的 Mapper 输出中，<ODPS, 1>这样的键值对有 3 条，Reducer 对它处理后，将生成如<ODPS, 3>，把键（ODPS）和值（3）分别赋给 Record 记录的第 1 个字段和第 2 个字段，然后输出 Record。WordCount.java 的代码清单如下：

```
package example.wordcount;

import java.io.IOException;
import java.util.StringTokenizer;

import com.aliyun.odps.Record;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.TableInfo;
import com.aliyun.odps.io.TableInputFormat;
import com.aliyun.odps.io.TableOutputFormat;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.mapreduce.JobClient;
import com.aliyun.odps.mapreduce.JobConf;
import com.aliyun.odps.mapreduce.MapContext;
import com.aliyun.odps.mapreduce.Mapper;
import com.aliyun.odps.mapreduce.ReduceContext;
import com.aliyun.odps.mapreduce.Reducer;

public class WordCount {

    public static class WCMapper extends Mapper<Text, LongWritable> {

        private final static LongWritable one = new LongWritable(1);
        private Text word = new Text();

        @Override
        public void map(LongWritable recordNum, Record record,
            MapContext<Text, LongWritable> context) throws IOException,
            InterruptedException {
            StringTokenizer st = new StringTokenizer(record.get(0).toString());

            while (st.hasMoreElements()) {
```

```
// trim the ending "."
String s = st.nextElement().toString().replace(".", "");
word.set(s);
context.write(word, one);
}
}
}

public static class WReducer extends Reducer<Text, LongWritable> {
    private LongWritable sum = new LongWritable();
    private Record result = null;

    @Override
    protected void setup(ReduceContext<Text, LongWritable> context)
        throws IOException, InterruptedException {
        result = context.createOutputRecord();
    }

    @Override
    public void reduce(Text key, Iterable<LongWritable> values,
        ReduceContext<Text, LongWritable> context) throws IOException,
        InterruptedException {
        int count = 0;
        for (LongWritable val : values) {
            count += val.get();
        }
        sum.set(count);
        result.set(0, key);
        result.set(1, sum);
        context.write(result);
    }
}
```

```
}  
}  
  
public static void main(String[] args) throws Exception {  
    if (args.length != 2) {  
        System.err.println("Usage: wordcount <in_table> <out_table>");  
        System.exit(2);  
    }  
    JobConf job = new JobConf();  
    job.setMapperClass(WCMapper.class);  
    job.setReducerClass(WCReducer.class);  
  
    job.setMapOutputKeyClass(Text.class);  
    job.setMapOutputValueClass(LongWritable.class);  
  
    TableInputFormat.addInput(new TableInfo(args[0]), job);  
    TableOutputFormat.addOutput(new TableInfo(args[1]), job);  
  
    JobClient.runJob(job);  
}  
}
```

现在,这段代码结构很简单,也很典型,它包含一个main函数,一个Mapper类(WCMapper),一个Reducer类(WCReducer)。我们简单分析一下。

- 首先,在main函数中,先实例化一个Jobconf对象,通过一系列set函数进行配置,分别设置Map、Reduce类以及中间结果<Key,Value>类型;然后,通过TableInputFormat和TableOutputFormat设置输入和输出,最后调用JobClient.runJob(job)。

注意,main函数在调用runJob之前的代码都是在客户端运行,它一般完成参数解析、JobConf配置等。调用runJob(job)后,就把配置好的job提交到集群上运行。

- 在WCMapper类继承Mapper类,先定义一个LongWritable对象,值置为1,用于设置map输出每个单词的计数值。然后定义一个Text对象,用于保存单词。

Mapper 类包含四个方法:

- a) `setup(MapContext)`: 在完成 Mapper 构造、`map()`方法开始之前调用;
- b) `cleanup(MapContext)`: 在 `map()`方法之后调用, `setup()`和 `cleanup()`方法用于管理 Mapper 生命周期中的对象;
- c) `map(LongWritable, Record, MapContext)`: `map()`方法用于处理每次调用传入的一条表记录, 第一个参数为当前输入记录的序号, 从 1 开始计数; 第二个参数为当前输入记录, 可通过 `get()/set()`方法操作记录列值;
- d) `run(MapContext)`: MR 框架会自动调用 `run()`方法, 先执行 `setup()`, 然后调用 `map()`方法迭代处理输入表中的所有记录, 最后调用 `cleanup()`。

`MapContext`, 和下面的 `ReduceContext`, 表示 ODPS MR 的上下文对象, 它们都继承自 `Context` 对象, 提供了大量 MR 程序所需要的功能, 比如读取资源文件等。

这里, 在 `map` 函数中先通过 `StringTokenizer` 对文本切, 然后遍历各个元素, 调用 `context.write(word, one)`, 输出 `map` 结果;

- `WCMapper` 类继承 `Reducer` 类, `Reduce` 类包含的方法和 `Map` 类似, 这里不再一一说明。由于要把结果输出到 ODPS 表中, 所以先定义一个 `Record` 变量 `result`, 然后, 在 `setup` 方法中, 创建输出记录, 把返回值赋给 `result`; 在 `reduce` 方法中, 对相同 `key` 的计数值累加, 把结果值赋给 `result`, 然后调用 `context.write(result)`输出结果记录。

好了, 看完这些分析, 你是不是觉得写 ODPS MR 其实可以很简单? 确实如此, 就像看福尔摩斯探案, 了解了如何分析之后, 一切就不再那么神秘了。为了进一步揭开 ODPS MR 的面纱, 我们还需要跑一跑, 看看结果如何。

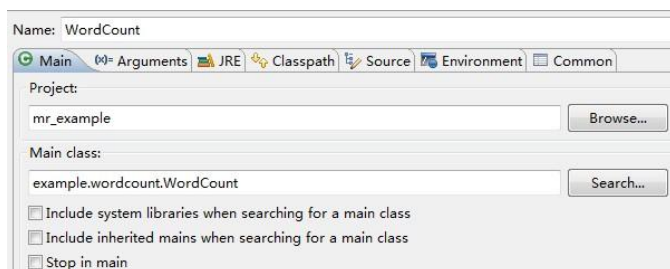
要运行该 MR 程序, 需要给出输入表和输出表。前面已经把数据导入到输入表 `mr_wc_in` 了, 因而这里需要创建输出表, SQL 语句如下:

```
create table mr_wc_out(word string, count bigint);
```

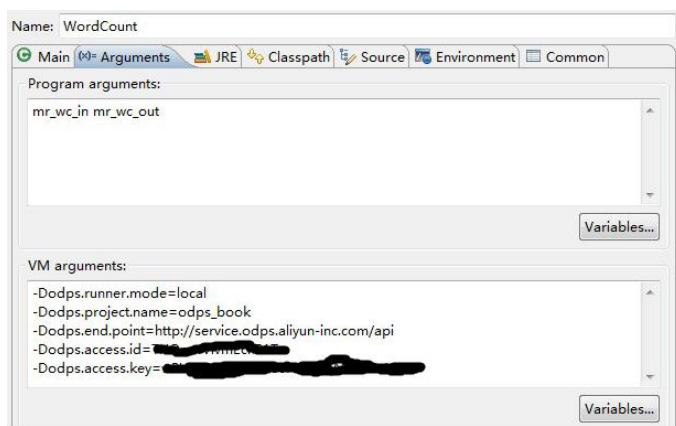
下面, 我们将按照“开发流程”一节建议的步骤顺序, 逐步调试运行该代码。

3.3.4 本地运行调试

在 Eclipse 中, 右键 `WordCount.java`, 点击 `Run As->Run Configurations`, `Main` 标签栏如下:



在 Arguments 标签栏中，分别配置 Program 参数和 VM 参数，如下：



配置好后，点击 Run，程序在 Eclipse Console 中输出如下：

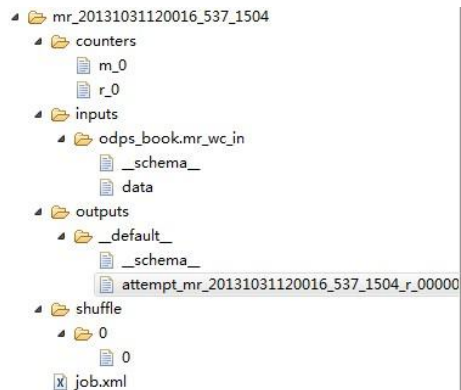
```
Summary:
counters: 23
    map-reduce framework
        combine_input_groups=0
        combine_input_records=0
        combine_output_records=0
        map_input_bytes=134
        map_input_records=1
        map_max_memory=0
        map_max_used_buffer=0
        map_output_bytes=0
        map_output_records=0
        map_shuffle_bytes=311
        map_shuffle_records=23
        map_spill_files=0
        reduce_input_groups=20
        reduce_input_records=23
```

```
reduce_max_memory=0
reduce_output_bytes=174
reduce_output_records=20
map_input_[odps_book.mr_wc_in]_bytes=134
map_input_[odps_book.mr_wc_in]_records=1
reduce_output_[odps_book.mr_wc_out]_bytes=174
reduce_output_[odps_book.mr_wc_out]_records=20
job counters
total_launched_maps=1
total_launched_reduces=1
```

OK

现在，我们来简单分析一下该输出。通过每个 Counter¹⁹名称可以很容易明白其含义，因此这里不逐个说明。由于没有使用 Combine 函数，所有 combine_* 的值都为 0，map 的结果是生成<key,value>键值对给 Reducer，所以 map_out_* 的值也都为 0（后面我们会介绍只有 Mapper 的案例，和这个会有所不同）。在这个例子中，map 只有一个输入，所以 map_input_bytes=134（表示总输入）和 map_input_[odps_book.mr_wc_in]_bytes=134（给出其中一个输入）值相同，其他一些 Counter 分析类似。后面我们会介绍多路输入的案例。

运行一次 MR 作业后²⁰，在 Eclipse 工作空间所关联的数据目录下会生成一个名为 mr_[timestamp]_[random]_[random] 的临时目录，如下图所示：



¹⁹ 注：本地模式运行的 Counters，有一些功能没有实现，集群上运行 Counters 更全。

²⁰ 注：ODPS MR 本地调试还提供了本地 Warehouse 配置方式，输入可以直接读取模拟数据（本地文件），实现真正“pure local”模式。这种方式可以方便修改数据，便于调试。由于实际上开发人员使用不多，这里不做介绍。感兴趣的可以查看 ODPS MR 手册：

http://odps.alibaba-inc.com/doc/prddoc/odps_mr/odps_mr_dev_test.html

该目录下包含如下目录和文件：

- counters - 存放作业运行的一些计数信息
- inputs - 存放作业的输入数据，默认一个 input 只读 100 条数据，可以通过 `-Dodps.mapred.local.record.limit` 参数进行修改，最大值是 10000；
- outputs - 存放作业的输出数据
- resources - 存放作业使用的资源；
- shuffle - 存放 map -> reduce 的中间数据
- job.xml - 作业配置

查看 `outputs->_default_` 目录下的 `attempt_mr_*` 文件，会发现 reduce 输出结果和预期一致。当然，如果输出和预期不同，可以通过 Eclipse 断点调试，或在每一步预期结果前增加 `System.out.print` 输出到屏幕，看哪一步运行结果和预期不同。Java 开发人员对这些调试都非常熟悉，这里不再赘述。

3.3.5 单元测试

本地调试 Ok 后，现在我们来编写 WordCount 单元测试。

ODPS MR 框架提供了一个单元测试基类 `MRUnitTest`，通过继承该类，可以比较简单编写 MR 程序的单元测试用例。先一起来看一下 `MRUnitTest` 的接口定义：

```
public class MRUnitTest<K, V> {  
  
    public MapOutput<K, V> runMapper(JobConf job, MapUTContext context)  
  
        throws IOException, ClassNotFoundException, InterruptedException;  
  
    public ReduceOutput runReducer(JobConf job, ReduceUTContext<K, V> context)  
  
        throws IOException, ClassNotFoundException, InterruptedException;  
  
    ...  
}
```

在 `MRUnitTest` 类中，提供了两个主要的方法用于编写 Mapper 和 Reducer 的单元测试用例：

- `runMapper(JobConf, MapUTContext)`：执行 Mapper 过程
- `runReducer(JobConf, ReduceUTContext)`：执行 Reducer 过程

其中，`MapUTContext` 和 `ReduceUTContext` 分别为 Mapper 和 Reducer 单元测试的上下文，均继承自基类 `UTContext`。

下面，我们一起来实现 WordCount 单元测试代码 `WordCountTest.java`，完整的代码清单如下：

```
package example.wordcount;

import java.io.IOException;
import java.util.List;
import junit.framework.Assert;
import org.junit.Test;
import com.aliyun.odps.Record;
import com.aliyun.odps.io.*;
import com.aliyun.odps.mapreduce.JobConf;
import com.aliyun.odps.mapreduce.unittest.*;
import example.wordcount.WordCount.*;

public class WordCountTest extends MRUnitTest<Text, LongWritable> {

    // 定义输入输出表的 schema

    private final static String INPUT_SCHEMA = "a:string";
    private final static String OUTPUT_SCHEMA = "k:string,v:bigint";

    private JobConf job;

    public WordCountTest() throws IOException {
        // 准备作业配置

        job = new JobConf();
        job.setMapperClass(WCMapper.class);
        job.setReducerClass(WCReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(LongWritable.class);
    }
}
```

```
TableInputFormat.addInput(new TableInfo("mr_wc_in"), job);

TableOutputFormat.addOutput(new TableInfo("mr_wc_out"), job);

}

@Test

public void TestMap() throws IOException, ClassNotFoundException,

    InterruptedException {

    // 准备测试数据

    MapUTContext context = new MapUTContext();

    context.setInputSchema(INPUT_SCHEMA);

    context.setOutputSchema(OUTPUT_SCHEMA);

    Record record = context.createInputRecord();

    record.set(new Text[] { new Text("hello java. hello python. ") });

    context.addInputRecord(record);

    // 运行 map 过程

    MapOutput<Text, LongWritable> output = runMapper(job, context);

    // 验证 map 的结果, 为6组 key/value 对

    List<KeyValue<Text, LongWritable>> rawKvs = output.getRawOutputKeyValues();

    Assert.assertEquals(4, rawKvs.size());

    Assert.assertEquals(new KeyValue<Text, LongWritable>(new Text("hello"),

        new LongWritable(1)), rawKvs.get(0));

    Assert.assertEquals(new KeyValue<Text, LongWritable>(new Text("java"),

        new LongWritable(1)), rawKvs.get(1));

    Assert.assertEquals(new KeyValue<Text, LongWritable>(new Text("hello"),

        new LongWritable(1)), rawKvs.get(2));

    Assert.assertEquals(new KeyValue<Text, LongWritable>(new Text("python"),
```

```
        new LongWritable(1)), rawKvs.get(3));
    }

@Test
public void TestReduce() throws IOException, ClassNotFoundException,
    InterruptedException {
    // 准备测试数据
    ReduceUTContext<Text, LongWritable> context = new ReduceUTContext<Text,
LongWritable>();

    context.setOutputSchema(OUTPUT_SCHEMA);

    context.addInputKeyValue(new Text("hello"), new LongWritable(1));
    context.addInputKeyValue(new Text("java"), new LongWritable(1));
    context.addInputKeyValue(new Text("hello"), new LongWritable(1));
    context.addInputKeyValue(new Text("python"), new LongWritable(1));

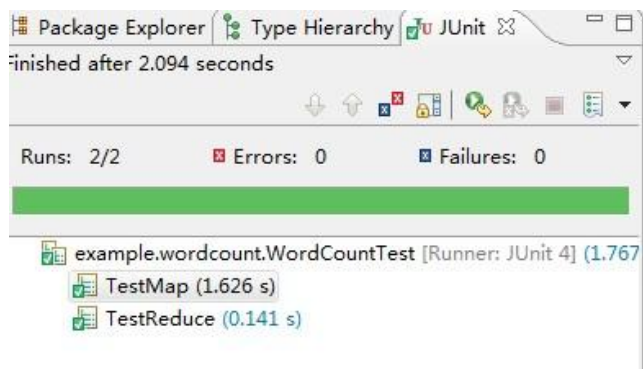
    // 运行 reduce 过程
    ReduceOutput output = runReducer(job, context);

    List<Record> records = output.getOutputRecords();

    // 验证 reduce 结果, 为 3 条 record
    Assert.assertEquals(3, records.size());

    Assert.assertEquals(new Text("hello"), records.get(0).get("k"));
    Assert.assertEquals(new LongWritable(2), records.get(0).get("v"));
    Assert.assertEquals(new Text("java"), records.get(1).get("k"));
    Assert.assertEquals(new LongWritable(1), records.get(1).get("v"));
    Assert.assertEquals(new Text("python"), records.get(2).get("k"));
    Assert.assertEquals(new LongWritable(1), records.get(2).get("v"));
}
}
```

由于用到 `junit.framework` 和 `org.junit`，因此需要下载相应的 jar 包，我们下载了 `junit.jar` 和 `com.springsource.org.junit-4.7.0.jar`，并把它们添加到 `Build Path` 中。配置好后，直接 `Run As -> JUnit Test` 方式运行 `WordCountTest.java` 程序文件，结果如下图所示：



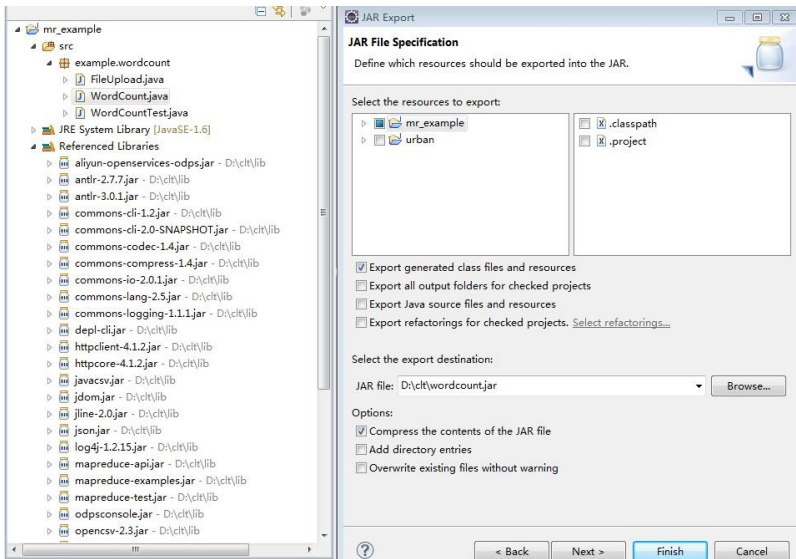
`WordCountTest.java` 代码非常清晰明白，这里不做分析。

3.3.6 集群调试

完成本地调试和单元测试后，现在我们有信心去集群上调试了，一起来体验一下吧。

1. 编译打包

在 Eclipse 中，右击 `WordCount.java`，点击 `Export`，选择 `JAR File`，设置目标路径，如 `D:\c\l\wordcount.jar`，然后点击 `Finish` 即可完成打包。



2. 打开配置好的ODPS CLT，运行bin/odpscmd21，上传wordcount.jar包作为资源：

```
add jar /home/admin/book/lib/wordcount.jar -f;
```

3. 使用jar命令运行WordCount作业，执行命令及输出如下

```
odps@ odps_book>jar -libjars wordcount.jar -classpath
/home/admin/book/lib/wordcount.jar example.wordcount.WordCount mr_wc_in
mr_wc_out;

ID = 20131031085029588gjt52b34

Tracking URL:
http://webconsole.odps.aliyun-inc.com:8080/logview/?h=http://service.odps.aliyun
-inc.com/api&p=odps_book&i=20131031085029588gjt52b34&token=TEROU0crNE9zcmd2NVcrS
G1UMkRaNwdVZXc0PSxPRFBTX09CTzoxNTAxODcwMjYyOTM1NDQ1LDEzODM4MTQyMjksewogICAgI1N0Y
XR1bWVudCI6IFt7CiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgI
kVmZmVjdCI6ICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgI
3RzL29kcHNfYm9vay9pbmN0YW5jZXMvMjAxMzEwODUwMjk1ODhnanQ1MmIzNCJ9XSwwKICAgICJWZ
XJzaW9uIjogIjEifQ==

2013-10-31 16:47:52      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:48:26      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:48:59      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:49:31      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:50:04      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:50:37      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:51:10      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:51:42      MapTask:0/1      ReduceTask:0/0

2013-10-31 16:52:15      MapTask:1/1      ReduceTask:0/1

2013-10-31 16:52:20      MapTask:1/1      ReduceTask:1/1

Summary:

Map-Reduce Framework
```

²¹ odpscmd 可以在 Windows 上运行，也可以在 Linux 上运行，理论上只要是有 JDK1.6 以上即可。笔者习惯在 Linux 下执行命令，所以这里给的都是 Linux 下执行的命令。你可以根据自己的情况选择。当然，在 Linux 下运行，需要把刚生成的包上传到 Linux 机器。

```
combine_input_groups=0
combine_input_records=0
combine_output_records=0
map_avg_time=699
map_cost_cpu=100
map_cost_memory=2048
map_cost_time=1
map_input_[odps_book.mr_wc_in]_bytes=220
map_input_[odps_book.mr_wc_in]_records=1
map_input_bytes=220
map_input_records=1
map_input_records_avg=1
map_input_records_max=1
map_input_records_min=1
map_max_memory=110407912
map_max_time=699
map_max_used_buffer=311
map_output_bytes=0
map_output_records=0
map_output_records_avg=0
map_output_records_max=0
map_output_records_min=0
map_shuffle_bytes=357
map_shuffle_records=23
map_spill_files=0
reduce_avg_time=448
reduce_cost_cpu=100
reduce_cost_memory=2048
reduce_cost_time=1
```

```
reduce_input_bytes=357
reduce_input_groups=20
reduce_input_records=23
reduce_input_records_avg=23
reduce_input_records_max=23
reduce_input_records_min=23
reduce_max_memory=5339848
reduce_max_time=448
reduce_output_[odps_book.mr_wc_out]_bytes=284
reduce_output_[odps_book.mr_wc_out]_records=20
reduce_output_bytes=284
reduce_output_records=20
reduce_output_records_avg=20
reduce_output_records_max=20
reduce_output_records_min=20
File Systems
pangu_read=370
pangu_write=389
Job Counters
total_launched_maps=1
total_launched_reduces=1
OK
```

这里，输出 Counters 和前面本地运行有所差别，比如多了 `map_avg_time=699` 和 `reduce_avg_time=448` 等相关选项。读懂 Counters 对性能调优非常重要，一些值得注意的 Counters 如下：


- `map_avg_time` - map 平均任务执行时间，单位：毫秒
- `map_max_memory` - 最大 map 任务 JVM 使用内存，单位：字节，这个值如果接近于（例如差值小于 300M）设置的 map 内存，则需要加大 map 内存

- **map_max_time** - 最大 **map** 任务运行时间, 如果这个值比 **map_avg_time** 大很多, 可能存在切分 (**split**) 不均匀
- **map_spill_files** - **map** 排序时写到磁盘的临时文件数量, 应尽可能减少这个值
- **reduce_input_records_max** - 最大 **reduce** 任务的输入记录数, 如果这个值比平均值高很多, 说明 **reduce** 任务处理的数据量不均匀, 即所谓的数据倾斜, 会导致长尾
- **reduce_max_memory** - 最大 **reduce** 任务 **JVM** 使用内存, 单位: 字节, 这个值如果接近于 (例如差值小于 300M) 设置的 **reduce** 内存, 则需要加大 **reduce** 内存
- **reduce_max_time** - 最大 **reduce** 任务运行时间, 如果这个值比 **reduce_avg_time** 大很多, 可能存在数据倾斜



也许你现在看到这些**Counters**, 完全没有感觉。没有关系, 在后面的实践中, 如果**MR**作业运行时间很长, 可以看看这些**Counters**, 也许会帮助你解决一些性能问题, 同时也能更好理解这些**Counters**。或许到时你就豁然开朗啦。

此外, 在输出中还包含**Tracking URL**, 可以通过它在浏览器查看作业的详细运行情况, 如下图,

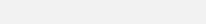

ODPS Instance									
URL	Project	InstanceID	Owner	StartTime	EndTime	Latency (s)	Status	Priority	SourceXML
http://service...	odps_book	20131031085...	ALIYUN\$dxp_757...	2013-10-31 16:50:29	2013-10-31 16:55:27	4:58	Terminated	9	4%






console_mr_201310...

ODPS Tasks									
Name	Type	Status	Result	Detail	StartTime	EndTime	Latency (s)	TimeLine	
console_mr_201310...	MAPR...	Success			2013-10-31 16:50:29	2013-10-31 16:55:26	4:57		

点击**Detail**图标, 在弹出页面中如下点击各个**Tab**, 可以看到**Instance22**日志输出。

Detail for [console_mr_20131031164720_907]									
refresh									
Foxi Jobs Summary JSONSummary									
Foxi Job Name: odps_book_20131031085029588git52b34_MAPREDUCE_0_0									
TaskName	Fatal/InstCount	I/O Records	Progress	Status	StartTime	EndTime	Latency(s)	TimeLine	
1 MapTask	0/1	0/0	100%	Terminated	2013-10-31 16:50:41	2013-10-31 16:55:08	4:27		
2 ReduceTask	0/1	0/0	100%	Terminated	2013-10-31 16:55:08	2013-10-31 16:55:24	16		

MapTask ReduceTask									
Failed(0) Terminated(1) All(1) Long-Tails(0) Latency chart Latency: {"min": "1", "avg": "1", "max": "1"}									
FuxiInstanceID	IP & Path	StdOut	StdErr	Status	StartTime	EndTime	Latency(s)	TimeLine	
1 Odps/odps_b...	10.149.10.18...			Terminated	2013-10-31 16:54:53	2013-10-31 16:54:54	1		

现在, 我们来看看输出结果表:

```
odps@ odps_book>read mr_wc_out;
```

²² 作业提交到集群上, 即对应一个 **Instance** 来运行。作业是个静态概念, **Instance** 相当于作业的实例。

```
+-----+-----+
| word      | count    |
+-----+-----+
| Data      | 1        |
| Enjoy     | 1        |
| MapReduce | 1        |
| ODPS      | 3        |
| Open      | 1        |
| Processing | 1        |
| Service   | 1        |
| This      | 1        |
| Welcome   | 1        |
| WordCount | 1        |
| a         | 1        |
| an        | 1        |
| example   | 1        |
| file      | 1        |
| for       | 1        |
| is        | 2        |
| it        | 1        |
| small     | 1        |
| the       | 1        |
| to        | 1        |
+-----+-----+
```

哈哈，和预期一样，简单的WordCount之旅完美结束啦。

3.4 小结

本章简要介绍ODPS MR的基本知识，通过WordCount示例说明如何开发、调试和运行ODPS MR程序。现在，也许你已经迫不及待地希望通过ODPS MR来解决自己当前的实际需求。好吧，去吧，实践是最好的老师。如果你有时间，或者在实践中遇到一些难题，也许看看后面的内

容对你会有些帮助。

第4章 ODPS MapReduce 进阶探讨

离线数据处理经常会处理一些比较复杂的场景。本章将进一步探讨 ODPS MapReduce 的一些特性和原理，以编写更丰富复杂的 MR 程序来处理不同的需求。

4.1 ODPS MapReduce 原理

本节探讨 ODPS MapReduce 原理的目的在于帮助用户理解 ODPS MR JavaDoc 中各个核心接口类在数据处理过程是什么角色以及在哪里使用。因此，我们将从使用的角度（注意，而不是复杂的内部实现机制²³），简单介绍 ODPS MR 的原理，它可以帮助用户更好地理解 ODPS MR 程序的处理机制，以实现复杂的数据处理逻辑或进行性能调优。

4.1.1 从用户视角谈原理

如下图所示，左侧是数据处理流程，右侧是提供该功能的相应接口类。

²³ 如果你想了解 MapReduce 的内部实现机制，可以查看《Hadoop The Definitive Guide》一书。

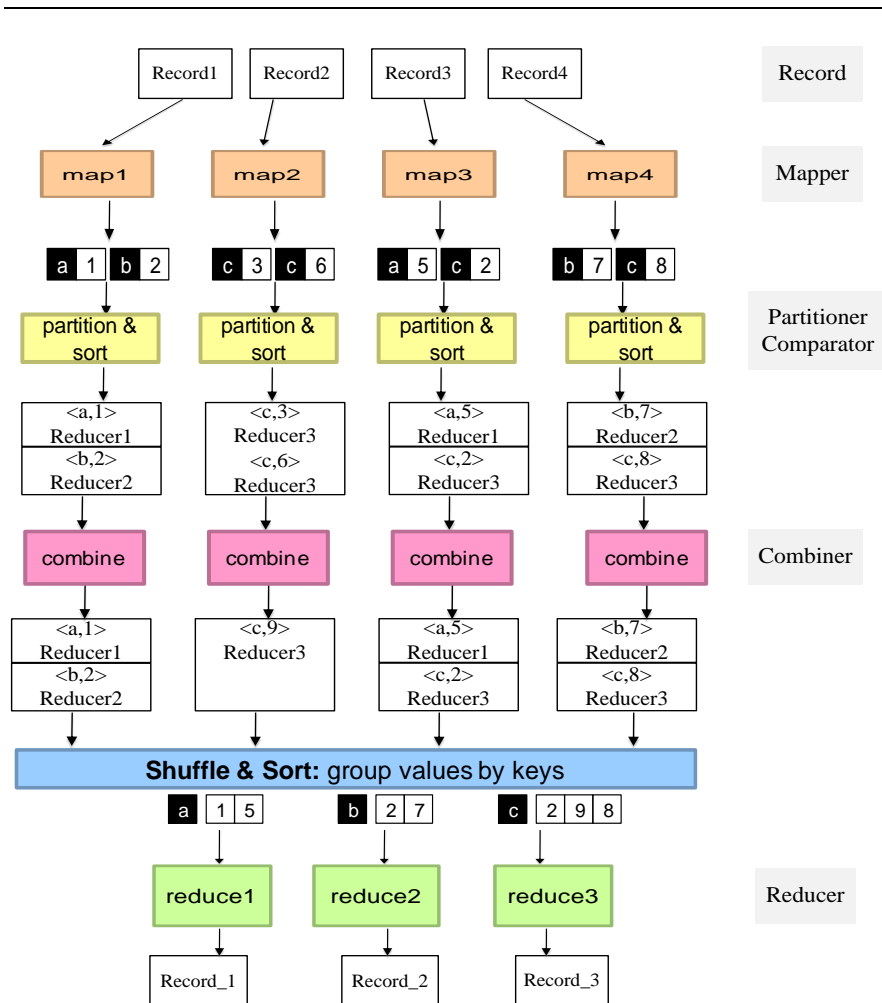


图 2-1 ODPS MapReduce 原理图

如图 2-1 所示，Mapper 的输入是 ODPS Record，通过 map 处理后，生成<key,value>键值对，比如上图中 map1 输出两个键值对<a,1> 和 <b,2>。map 的输出结果会多路输给 Reducer（即输给多个 Reducer），通过 Partitioner 来确定数据应该输出给哪个 Reducer，比如之前提到的 <a,1> 和 <b,2>，分两路输出，分别输出给 Reducer1 和 Reducer2。

排序（Sort）是对同一路（即输出给同一个 Reducer）的数据的 key 进行排序，排序规则是通过 Comparator 定义的，假设这里按字母序排序，map1 输出给 Reducer1 的键值对如下：<a,1>，<you,2>，<me,3>，则排序后结果是<a,1>，<me,3>，<you,2>。想一想，如果 map1 的输出是<a,1>，<you,2>，<me,4>，<m,3>，排序之后，输出一定会是 <a,1>，<me,3>，<me,4>，<you,2>吗？答案是否。排序只对 key 进行排序，相同 key 的输出结果是不确定的。试想一下，如果希望输出严格按照<a,1>，<me,3>，<me,4>，<you,2>次序，应该怎么办？

combine 操作默认是没有的，这里给出是为了说明如果自定义了 Combiner 函数，它是用在哪里。Combiner 是对同一路中的数据进行合并。比如，这里的 Combiner 操作是对键值进行求和，map2 的输出<c,3>和<c,6>，combine 操作后生成<c,9>。由于 Combiner 是在 Map 端本

地合并，所以在某些场景下，设置 Combiner 可以减少通过网络传输传给 Reducer 的数据，从而提高性能。但是，并不是所有场合都适用 Combiner，具体见后面说明。

Reducer 从各个 Mapper 的输出结果中“拉 (shuffle)”相应的数据，比如在图 2-1 中，Reducer1 分别获取 map1 和 map3 的输出结果<a,1> 和 <a,5>，对 key 进行归并排序（因为 map 输出是各路有序的，不同 map 之间是无序的，所以在 Reducer 端会做全局排序），输出结果<a,[1,5]> 给 Reducer 的 reduce 函数，reduce 执行用户实现的逻辑，输出结果 Record。

值得一提的是，在图 2-1 中，在“shuffle”之前的处理是在 Mapper 端执行，从“shuffle”开始是在 Reducer 端处理。在集群上，从 Mapper 端到 Reducer 端的数据传输是通过网络传输来完成（假定 Mapper 和 Reducer 在不同机器，如果在同一机器，相当于在本地执行）。

为了便于理解各个接口的角色，我们在图 2-1 的右侧给出了相应的接口类。比如 partition 功能是通过 Partitioner 实现，sort 功能是通过 Comparator 实现。在 1.2 节核心接口介绍中，会详细说明各个接口类，建议查看时比照该图，也许可以更好地理解它们。

4.1.2 如何实现 map 和 reduce?

如何实现 map 和 reduce，也就是如何基于 MapReduce 编程模型来处理数据，我理解重点在于如何设计 Mapper 的输出和 Reducer 的输入。它可以很简单，比如在前面介绍的 WordCount 示例中，很自然地，map 完成单词切分并计数，输出<单词, 1>，reduce 对相同单词分组聚合，计算单词出现的计数。但是，也有很多场景，map 和 reduce 的处理绝不是这么简单。比如，来往用户好几亿了（迟早的），它有很多好友关系数据，如何基于 MapReduce 编程模型实现来往好友推荐？比如维护了一份家族中人与人关系的数据，如何构建家庭树？或者如何查找淘宝用户共同特征？

遗憾的是，我无法用三言两语回答“如何实现 map 和 reduce”。我们将在后面的示例中具体描述这些场景，说明如何抽象、如何通过 ODPS MR 来实现它们。也许到时你就能对 ODPS MR 编程有自己的心得。

4.1.3 使用 Combiner?

当集群规模很大，对于海量数据处理，Mapper 端和 Reducer 端的数据一般是通过网络传输。我们都知道，网络资源是个宝，所以如何能够减少网络传输的数据量“且不影响 Reduce 最终输出”（这里引号表示强调），那就再好不过了。比如，在前面 WordCount 示例中，假设 MapReduce 处理的数据量非常大（比如爬虫抓取的互联网网页数据），假设一个 map 的输出数据量比较大，如“<the,1>”这样的中间结果就有 100 万条。如何减少数据传输？Combiner。在 Map 端对这些中间结果进行合并。在这个场景中，使用 Combiner 之前，需要传输 100 万条的<the,1>给 Reducer，而采用 Combiner 之后，可以对相同的 key 进行合并，在这个求和计数例子中，就把 100 万条的<the,1>合并成一条记录<the,1000000>，再传输给 Reducer，这样就会大大减少了传输的中间结果数据量。Combiner 这么好，那在所有的 MapReduce 程序中都设置 Combiner？不是这样的。

Combiner 的使用要慎重。一是并非所有的场景使用 Combiner 都会达到优化效果；二是并非所有的场景都适合用 Combiner。

对于第一种情况，比如 map 输出中键值对中相同 key 很少，使用 Combiner 可能对于效率反而适得其反。因为 Combiner 操作需要读取 map 输出并合并，combine 操作本身是有消耗的。如果 combine 操作对数据量影响不大，可能更适合直接输出 map 中间结果给 Reducer，在 Reducer 中再做全局合并操作。

对于第二种情况，使用 Combiner 的原则是不要改变 Reducer 的输出。因此，Combiner 适用于如计算求和、求最大值之类的操作，而对于求平均值，就不适合用 Combiner。

现在，我们看看下面这个例子。假设输出如下键值对，要对相同 key 的 value 求平均值，如下：

<key,1>, <key,5>, <key,8>, <key,6>, <key,11>, <key,35> （这里我们不 care key 的值）

假设 map 输出两路<key,1>, <key,5>和<key,8>, <key,6>, <key,11>, <key,35>, 如果不用 Combiner，最后在 Reducer 归并求平均值，计算结果如下：

$$(1 + 5 + 8 + 6 + 11 + 35) / 6 = 11$$

如果使用 Combiner，先对各路求平均值，最后再 Reducer 再求平均，计算如下：

$$(1 + 5) / 2 = 3$$

$$(8 + 6 + 11 + 35) / 4 = 15$$

$$(3 + 15) / 2 = 9$$

对于这个结果，你是不是惊讶了？也就是说，对于平均值计算，执行 Combiner 会改变最终 Reducer 的输出，不能使用 Combiner。类似的场景还有很多，你能想到吗？

另外，假设现在中间输出结果很多，减少传给 Reducer 的中间结果是“必须”的，该怎么办？是否有其他 combine 方式？答案是肯定的。你能想到吗？我们将在后面 Combine 示例中说明如何实现。

4.1.4 自定义 Comparator?

“什么时候要自定义 Comparator？”如果你问这个问题，一般说明你的应用不需要自定义 Comparator（不要笑，认真的）。当需要的时候，你的问题往往变成“我该如何定义 Comparator？”

简单而言，就是当默认的 Comparator 不满足需求时，就自定义 Comparator。比如，在前面给出的 WordCount 示例中，默认是按单词的字母序排序，如果你希望按照单词的长度排序，那就需要定义一个了。至于如何定义，这完全取决于应用场景。我们在后面的 Comparator 示例中会给出如何实现自定义 Comparator。

4.1.5 自定义 Partitioner?

如果没有定义 Partitioner，ODPS MR 框架默认是按照哈希来划分的，它可以把输出结果均匀划分给各路 Reducer，且保证相同 key 输出到同一路 Reducer。自定义 Partitioner 典型的场景是“group by”和“cluster by”，希望符合某种规则的 key 都输给同一路 Reducer，然后在 Reducer 端执行归并计算。比如二次排序，需要把要排序的两个关键字（key1, key2）一起作为 Mapper 输出的 key，如果采用默认的哈希 Partition 方式，不能保证 key1 相同的都会输出到同一路 Reducer 中，这样在 Reducer 中就无法执行全局 group，所以必须自定义 Partitioner，只对 Mapper 输出 key 中的 key1 进行计算，确保 key1 相同的 key 都输出到同一路 Reducer。在后面的 Partitioner 示例中会给出如何实现自定义 Partitioner。

4.2 核心接口

我曾经尝试在示例中用到时再详细介绍相应接口，发现这样做反而显得有些散乱。因此，这里，我们将简要介绍 ODPS MR 的一些核心接口，理解这些接口是通过 ODPS MR 实现较复杂的数据处理场景的基础。后面在示例说明时只是简单提一下。

也许你一般是基于某个 MR 程序进行修改，来实现自己的业务逻辑，这是很常见的做法。如果你不愿意自己“知其然而不知其所以然”，还是建议看看这些核心接口。

为了避免过于冗长而枯燥，这里只给出这些接口的一些关键方法，详细的接口说明请查看 ODPS MR Java Docs²⁴。

4.2.1 Record

```
public interface Record {
    public int size();
    public Writable get(int index);
    public Writable get(String fieldName) throws IOException;
    public void set(int index, Writable value);
    public void set(String fieldName, Writable value) throws IOException;
    public void set(Writable[] values) throws IOException;
    public FieldSchema getField(int index);
    public FieldSchema[] getFields();
    public Writable[] getAll();
    ...
}

public class FieldSchema {
    public final String name;
    public final String type;
}
```

如图 2-1 所示，ODPS MR 的输入和输出都是 Record。Record 表示 ODPS 表或分区的一条记录，一条记录由多列组成，列包括列名、类型和值。

一个 Record 对象：

- 不包括分区列（Partition columns）
- 每列的属性用 FieldSchema 表示，包含了列名和类型
- 每列的值都是实现 Writable 接口的对象

Record 支持通过列的序号（0 起始）或列名操作记录的数据。

Writable 子类和列类型的对应关系如下：

- LongWritable - int, bigint
- DoubleWritable - double

²⁴ 注：http://odps.alibaba-inc.com/doc/prddoc/odps_mr/api/index.html

- Text - string
- BooleanWritable - boolean
- DatetimeWritable - datetime
- NullWritable 或 null - NULL 值
- InputBlobWritable - blob
- OutputBlobWritable - blob

4.2.2 Mapper

```
public class Mapper<KEYOUT, VALUEOUT> {  
  
    protected void setup(MapContext<KEYOUT, VALUEOUT> context)  
  
        throws IOException, InterruptedException;  
  
    protected void map(LongWritable recordNum, Record record,  
        MapContext<KEYOUT, VALUEOUT> context)  
  
        throws IOException, InterruptedException;  
  
    protected void cleanup(MapContext<KEYOUT, VALUEOUT> context)  
  
        throws IOException, InterruptedException;  
  
    public void run(MapContext<KEYOUT, VALUEOUT> context)  
  
        throws IOException, InterruptedException;  
  
}
```

如图 2-1，Mapper 处理输入表的记录对象 Record，通过 map 方法，加工处理成键值对集合，（若 Reducer 个数不为 0）输出到 Reducer，或者直接输出结果记录。

对于 Mapper，存在两种情况：

- 若 Reducer 数为 0，这种作业也称为 Map-only 作业。
- 若 Reducer 数不为 0，Mapper 生成键值对输出到 Reducer，这些键值对从 Mapper 输出到 Reducer 的中间过程也称为 shuffle。

Mapper 包括四个 protected 方法：

- setup(MapContext) 在任务开始，map 方法之前调用
- cleanup(MapContext) 在任务结束，map 方法之后调用
- map(LongWritable, Record, MapContext) map 方法，每次调用传入一条记录
- run(MapContext) MapReduce 框架会调用 run 方法

MapReduce 框架只会调用 `run(MapContext)` 方法，该方法的默认实现是：

```
public void run(MapContext<KEYOUT, VALUEOUT> context) throws IOException,
    InterruptedException {
    setup(context);
    while (context.nextRecord()) {
        map(context.getCurrentRecordNum(), context.getCurrentRecord(), context);
    }
    cleanup(context);
}
```

从上面代码可以看出，`run` 方法首先调用 `setup` 方法，然后循环读取输入记录 `Record`，传给 `map` 方法进行处理，最后调用 `cleanup` 方法。在大多数情况下，用户不需要重载 `run` 方法，而只需要重载其他三个方法。

`Mapper` 可以通过以下几个方法输出计算结果：

- `TaskAttemptContext.write(Record)` 直接输出计算结果到默认输出表
- `TaskAttemptContext.write(Record, String)` 直接输出计算结果到指定 `label` 的输出表
- `MapContext.write(Object, Object)` 输出中间计算结果（键值对）到 `Reducer`，此方法要求 `Reducer` 数大于 0

中间计算结果（键值对）从 `Mapper` 输出到 `Reducer` 的中间过程也称为 `shuffle`，这是 MapReduce 作业最核心、最复杂的部分，键值对类型通过下面两个方法进行设置：

- `JobConf.setMapOutputKeyClass(Class)`
- `JobConf.setMapOutputValueClass(Class)`

比如，我们在第 1 章的 `WordCount` 示例中，把键值对分别设置成 `Text` 和 `LongWritable` 类型，如下：

```
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(LongWritable.class);
```

4.2.3 Reducer

```
public class Reducer<KEYIN, VALUEIN> {
    protected void setup(ReduceContext<KEYIN, VALUEIN> context)
        throws IOException, InterruptedException;
```

```

protected void reduce(KEYIN key, Iterable<VALUEIN> values,
    ReduceContext<KEYIN, VALUEIN> context)
    throws IOException, InterruptedException;

protected void cleanup(ReduceContext<KEYIN, VALUEIN> context)
    throws IOException, InterruptedException;

public void run(ReduceContext<KEYIN, VALUEIN> context)
    throws IOException, InterruptedException ;
}

```

如图 21-所示，Reducer 对 shuffle & sort 后的键（Key）所关联的一组数值集（Values）进行归约计算。

与 Mapper 类似，Reducer 也包括四个 protected 方法，这里不再一一说明。

run(ReduceContext) 方法的默认实现如下：

```

public void run(ReduceContext<KEYIN, VALUEIN> context) throws IOException,
    InterruptedException {
    setup(context);

    while (context.nextKey()) {
        reduce(context.getCurrentKey(), context.getValues(), context);
    }

    cleanup(context);
}

```

在 Mapper 端，每个 mapper 生成中间结果（键值对）并通过 Partitioner 分发给对应的 Reducer；在 Reducer 端，每个 Reducer 会读入所有 Mapper 输出给它的数，这个过程分三步：

- 归并排序：Reducer 的输入虽然每一路都有序，但是整体并未有序，这时 Reducer 会进行归并排序，使用排序比较器可以由 JobConf.setOutputKeyComparatorClass(Class) 进行指定，如果没有指定，就使用 Writable Key 默认定义的比较器。
- 按 Key 对 Values 分组：经过第一步的归并排序，所有的 Key/Value 对都已经有序，这时会使用 JobConf.setOutputValueGroupingComparator(Class) 指定的分组比较器对排好序的 Key/Value 对进行分组，如果没有指定，则默认使用归并排序所使用的排序比较器。

- 循环调用 `reduce` 方法，传入 `Key` 和 `Value` 迭代器进行规约计算。

这里为什么需要区别 排序比较器 和 分组比较器，原因有两点：

- 两个比较器用于不同的阶段，在某些应用，用户需要分别进行处理，具体参见后面的示例。
- 性能上的考虑，分组比较器只需要对 `Key` 判等，相对于排序比较器可以减少比较次数

批注 [a2]: 给出示例

Reducer 的计算结果可以通过下面两种方法输出到结果表：

- `TaskAttemptContext.write(Record)` 输出计算结果到默认输出表
- `TaskAttemptContext.write(Record, String)` 输出计算结果到指定 `label` 的输出表

`ReduceContext` 继承自 `TaskAttemptContext`，所以也可以如 `WordCount` 示例中那样，调用 `ReduceContext.write(Record)` 输出计算结果。

4.2.4 Partitioner

```
public abstract class Partitioner<KEY, VALUE> {  
  
    public void configure(JobConf job);  
  
    public abstract int getPartition(KEY key, VALUE value, int numPartitions);  
  
    //numPartitions 为 Reducer 数目  
}
```

分区类，决定 Mapper 写出的中间结果（键值对）输出给哪个 Reducer。因为在 MapReduce 中，很可能有多个 Mapper 和 Reducer 任务并发执行，这里假设有 `m` 个 Mapper，`r` 个 Reducer，Mapper 的中间结果键值对输出到哪个 Reducer 进行处理由 Partitioner 决定。

可以通过 `JobConf.setPartitionerClass(Class)` 方法自定义分区类，否则使用默认的分区类：`HashPartitioner`。

`HashPartitioner` 实现如下：

```
public class HashPartitioner<K, V> extends Partitioner<K, V> {  
  
    @Override  
  
    public int getPartition(K key, V value, int numReduceTasks) {  
  
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;  
  
    }  
}
```

4.2.5 Comparator

```
public interface RawComparator<T> extends Comparator<T> {  
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);  
}
```

RawComparator 接口定义了一个基于对象二进制表示的比较方法。WritableComparator 实现了 RawComparator 接口，提供对 WritableComparable 对象的通用比较函数，如果需要自定义比较器，通常可以继承 WritableComparator 类。

ODPS 的 map 和 reduce 阶段均存在排序操作：

- 在 map 阶段，是在每个分区中按照 key 来 **排序** key/value 对，具有相同的 key 的 key/value 对将排在一起（但相同 key 的 value 之间的顺序不固定）；
- 在 reduce 阶段，则要对来自各个 map 的所有输入进行归并 **排序**，然后按照 key 对 value 进行 **分组**。

用户可以通过 JobConf.setOutputKeyComparatorClass() 方法指定按 key 排序的 Comparator，如果没有指定，则使用 key 默认定义的比较器（compareTo() 方法），即升序排列。

用户可以使用 JobConf.setOutputValueGroupingComparator() 方法指定分组比较器，对排好序的键值对进行分组，然后调用 reduce() 方法进行处理。如果没有指定，则默认使用 key 的排序比较器（JobConf.setOutputKeyComparatorClass() 指定的比较器）。

4.2.6 Combiner

```
public class Combiner<KEY, VALUE> {  
    protected void setup(CombineContext<KEY, VALUE> context)  
        throws IOException, InterruptedException;  
    protected void combine(KEY key, Iterable<VALUE> values, CombineContext<KEY, VALUE> context)  
        throws IOException, InterruptedException;  
    protected void cleanup(CombineContext<KEY, VALUE> context)  
        throws IOException, InterruptedException;  
    public void run(CombineContext<KEY, VALUE> context)  
        throws IOException, InterruptedException;  
}
```

Combiner，又叫 Local Reduce，负责对中间结果（键值对）的输出在 Mapper 端进行本地

聚集，这有助于降低从 Mapper 到 Reducer 数据传输量。

可以通过 `JobConf.setCombinerClass(Class)` 指定自定义的 Combiner，它在 Mapper 输出中间结果到 Reducer 前被调用。

与 Mapper, Reducer 类似，Combiner 也包括四个 `protected` 方法，这里不再一一说明。其 `run` 方法默认实现如下：

```
public void run(CombineContext<KEY, VALUE> context) throws IOException,
    InterruptedException {
    setup(context);

    while (context.nextKey()) {
        combine(context.getCurrentKey(), context.getValues(), context);
    }

    cleanup(context);
}
```

理解了 ODPS MapReduce 的原理和了解其核心接口后，你可能有点累了。好吧，先休息一下，下面我们将介绍几个有趣的 ODPS MapReduce 示例，你可以通过它们更好地理解如何应用 ODPS MR 编程模型。

4.3 简单示例

4.3.1 使用 Combiner

在前一章中，我们介绍了 WordCount 示例。在 Mapper 中输出<单词,1>这样的中间结果，在 Reducer 对这些中间结果进行合并。现在，试想一下，假设一个 map 的输出数据量比较大，如“<the,1>”这样的中间结果就有 1 万条。网络带宽就是宝，有没有什么方式可以减少 Mapper 到 Reducer 的数据传输？答案是肯定的。可以通过 Combiner，在 Map 端对这些中间结果进行合并。举个例子，在使用 Combiner 之前，需要传输 1 万条的<the,1>给 Reducer，而采用 Combiner 之后，可以对相同的 key 进行合并，在这个求和计数例子中，就把 1 万条的<the,1>合并成一条记录<the,10000>，再传输给 Reducer，这样就会大大减少了传输的中间结果数据量。

含 Combiner 的 WordCount 代码如下 WordCountWithCombiner.java，新增部分底色采用阴影显示。

```
package example.wordcount;

import java.io.IOException;
import java.util.StringTokenizer;

import com.aliyun.odps.Record;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.TableInfo;
```



```
import com.aliyun.odps.io.TableInputFormat;
import com.aliyun.odps.io.TableOutputFormat;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.mapreduce.CombineContext;
import com.aliyun.odps.mapreduce.Combiner;
import com.aliyun.odps.mapreduce.JobClient;
import com.aliyun.odps.mapreduce.JobConf;
import com.aliyun.odps.mapreduce.MapContext;
import com.aliyun.odps.mapreduce.Mapper;
import com.aliyun.odps.mapreduce.ReduceContext;
import com.aliyun.odps.mapreduce.Reducer;

public class WordCountWithCombiner {

    public static class WCMapper extends Mapper<Text, LongWritable> {

        private final static LongWritable one = new LongWritable(1);
        private Text word = new Text();

        @Override
        public void map(LongWritable recordNum, Record record,
            MapContext<Text, LongWritable> context) throws IOException,
            InterruptedException {
            StringTokenizer st = new StringTokenizer(record.get(0).toString());

            while (st.hasMoreElements()) {
                // trim the ending "."
                String s = st.nextElement().toString().replace(".", "");
                word.set(s);
                context.write(word, one);
            }
        }
    }

    public static class WCCombiner extends Combiner<Text, LongWritable> {
        private LongWritable result = new LongWritable();

        @Override
        protected void combine(Text key, Iterable<LongWritable> values,
            CombineContext<Text, LongWritable> context) throws IOException,
            InterruptedException {
            int count = 0;
            for (LongWritable value : values) {
                count += value.get();
            }
        }
    }
}
```

```

    }
    result.set(count);
    context.write(key, result);
}
}

public static class WReducer extends Reducer<Text, LongWritable> {
    private LongWritable sum = new LongWritable();
    private Record result = null;

    @Override
    protected void setup(ReduceContext<Text, LongWritable> context)
        throws IOException, InterruptedException {
        result = context.createOutputRecord();
    }

    @Override
    public void reduce(Text key, Iterable<LongWritable> values,
        ReduceContext<Text, LongWritable> context) throws IOException,
        InterruptedException {
        int count = 0;
        for (LongWritable val : values) {
            count += val.get();
        }
        sum.set(count);
        result.set(0, key);
        result.set(1, sum);
        context.write(result);
    }
}

public static void main(String[] args) throws Exception {
    if (args.length != 2) {
        System.err.println("Usage: wordcount <in_table> <out_table>");
        System.exit(2);
    }
    JobConf job = new JobConf();
    job.setMapperClass(WMapper.class);
    job.setCombinerClass(WCombiner.class);
    job.setReducerClass(WReducer.class);

    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);
}

```

```

    TableInputFormat.addInput(new TableInfo(args[0]), job);
    TableOutputFormat.addOutput(new TableInfo(args[1]), job);

    JobClient.runJob(job);
}
}

```

注意，在 `main` 方法中不要忘记调用 `job.setCombinerClass(WCCombiner.class)`，否则就不会用上实现的 `WCCombiner`。在 ODPS MR 中，由于 Reducer 的输出是 ODPS Record，而 Combiner 的输出是 `<key, value>` 键值对，所以不能直接把 Reducer 类作为 Combiner，这一点和 Hadoop 不同，属性 Hadoop MapReduce 编程的同学需要特别注意。自定义的 Combiner 需要继承 `Combiner<Text, LongWritable>` 类。

在“使用 Combiner？”一节中，我们讲述了 Combiner 适用和不适用的场景。在最后，我们抛出了一个问题，对于求平均值操作，如何对中间结果进行 `combine`。这里用伪代码²⁵来表示解法：

```

class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
class Combiner
    method Combine(string t, integers [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        Emit(string t, pair (sum, cnt)) _ Separate sum and count
class Reducer
    method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
            sum ← sum + s

```

²⁵ 注：这段伪代码引自《Data-Intensive Text Processing with MapReduce》一书。Emit 表示输出

```
cnt ← cnt + c

ravg ← sum / cnt

Emit(string t, integer ravg)
```

这段伪代码非常清晰，实现逻辑也不复杂，这里就不再给出完整的程序清单了，感兴趣的话可以自己实践一下。

4.3.2 自定义 Comparator

在 WordCount 示例中，我们没有自定义 Comparator，按照默认的字典序排序（因为设置 `job.setMapOutputKeyClass(Text.class);`，Text 默认是字典序）。假设希望根据单词长度进行排序，该怎么做呢？我们可以通过自定义 Comparator 来实现，在原始 WordCount 上，新增代码片段如下所示：

例1. 按单词长度排序

```
public static class WCComparator extends WritableComparator {
    public WCComparator() {
        super(LongWritable.class, true);
    }
    @Override
    public int compare(WritableComparable r1, WritableComparable r2) {
        long len1 = r1.toString().length();
        long len2 = r2.toString().length();
        return len1 == len2 ? 0: (len1 < len2 ? 1 : -1);
    }
}
```

在 main 函数中设置

```
job.setOutputKeyComparatorClass(WCComparator.class);
```

在本地调试运行后，为了便于比较，我们给出了之前按默认字典序排序时，在本地的输出结果，如下：

按默认字典序排序	按例 1 给出的单词长度排序
Data,1	Processing,1
Enjoy,1	MapReduce,2
MapReduce,1	Welcome,3
ODPS,3	small,2
Open,1	ODPS,7
Processing,1	for,2
Service,1	to,5
This,1	a,1
Welcome,1	

WordCount,1	
a,1	
an,1	
example,1	
file,1	
for,1	
is,2	
it,1	
small,1	
the,1	
to,1	

怎么回事？怎么丢失了这么多记录？“Welcome”对应的计数值为什么变成了 3？我们再回来看看上面定义的 Comparator 的 compare 函数的实现：

```
return len1 == len2 ? 0: (len1 < len2 ? 1 : -1);
```

这个含义是什么呢？它是认为长度相同的 key 就是相同的 key。看出问题了吧？相信你已经明白“Welcome”对应的计数值为什么变成了 3 了。

所以对于长度相同，我们还需要比较一下 key 本身是否相同才可以。上面的 compare 方法可以修改如下：

```
public int compare(WritableComparable r1, WritableComparable r2) {
    String k1 = r1.toString();
    String k2 = r2.toString();
    long len1 = k1.length();
    long len2 = k2.length();
    return len1 == len2 ? ((k1.equals(k2)) ? 0 : 1): (len1 < len2 ? 1 : -1);
}
```

运行后，输出如下（还是给出字典序进行参照）：

按默认字典序排序	修正后的单词长度排序
Data,1	Processing,1
Enjoy,1	MapReduce,1
MapReduce,1	WordCount,1
ODPS,3	Welcome,1
Open,1	example,1
Processing,1	Service,1
Service,1	small,1
This,1	Enjoy,1

Welcome,1	This,1
WordCount,1	file,1
a,1	ODPS,3
an,1	Open,1
example,1	Data,1
file,1	for,1
for,1	the,1
is,2	to,1
it,1	is,2
small,1	an,1
the,1	it,1
to,1	a,1

终于 ok 了。这个实践的经验总结是 `Comparator` 其实没那么简单，很容易出错，不能掉以轻心。

试想一下，如果对相同长度的 `key`，希望按字母序排序，怎么办？

4.3.1 示例：自定义 Partitioner

前面已经提到，二级排序（`SecondarySort`）是使用自定义 `Partitioner` 的典型用例，其使用场景大致是输出结果需要按两个列进行归并排序，先按第一个列排序，再按第二个列排序。

下面我们来考虑一个“`Group by`”场景：花花小学要给各个班级评小红花了，每个班级成绩最好的学生获得小红花。原始数据格式（学生姓名，年级，班级，成绩），如下：

```
tinky    grade1    class1    80
dipsy    grade1    class2    90
lala     grade1    class1    70
poe      grade1    class2    71
melanie  grade2    class1    90
winky    grade2    class2    88
linda    grade1    class1    65
jim      grade2    class2    60
```

期望输出（同一个年级的必须在一起连续）：

```
tinky    grade1    class1    80
dipsy    grade1    class2    90

melanie  grade2    class1    90
```

winky grade2 class2 88

这里，map 的输出结果 key 定义为（年级，班级），其值是（学生姓名，成绩），由于同一个年级必须输出到同一路 reducer 中，所以要自定义 Partitioner，否则只能保证相同年级且相同班级的输出会在一路 reducer。代码实现如下：

```
package example.mapreduce;

import java.io.IOException;

import com.aliyun.odps.Record;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.TableInfo;
import com.aliyun.odps.io.TableInputFormat;
import com.aliyun.odps.io.TableOutputFormat;
import com.aliyun.odps.io.Tuple;
import com.aliyun.odps.mapreduce.JobClient;
import com.aliyun.odps.mapreduce.JobConf;
import com.aliyun.odps.mapreduce.MapContext;
import com.aliyun.odps.mapreduce.Mapper;
import com.aliyun.odps.mapreduce.Partitioner;
import com.aliyun.odps.mapreduce.ReduceContext;
import com.aliyun.odps.mapreduce.Reducer;

public class ScoreRank {

    // partition based on (grade), map output key is (grade, class)
    public static class FirstPartitioner extends Partitioner<Tuple, Tuple> {
        @Override
        public int getPartition(Tuple key, Tuple value, int numReduceTasks) {
            return (key.get(0).hashCode() & Integer.MAX_VALUE) % numReduceTasks;
        }
    }

    // input Record: name, grade, class, score
    // output: <grade, class> <name, score>
    public static class ScoreMapper extends Mapper<Tuple, Tuple> {

        private Tuple key = new Tuple(2);
```

```
private Tuple value = new Tuple(2);

@Override
public void map(LongWritable recordCount, Record record,
    MapContext<Tuple, Tuple> context) throws IOException,
    InterruptedException {
    key.set(0,record.get(1));
    key.set(1,record.get(2));
    value.set(0,record.get(0));
    value.set(1,record.get(3));
    context.write(key, value);
}
}

// emit record with maximum score
public static class ScoreReducer extends Reducer<Tuple, Tuple> {
    private Record result = null;

    @Override
    public void setup(ReduceContext<Tuple, Tuple> context)
        throws IOException, InterruptedException {
        result = context.createOutputRecord();
    }

    @Override
    public void reduce(Tuple key, Iterable<Tuple> values,
        ReduceContext<Tuple, Tuple> context) throws IOException,
        InterruptedException {
        long max = 0;
        Tuple name = new Tuple(1);
        for (Tuple value : values) {
            long score = ((LongWritable)value.get(1)).get();
            if(max < score) {
                max = score;
                name.set(0,value.get(0));
            }
        }
        result.set(0, name.get(0));
        result.set(1, key.get(0));
        result.set(2,key.get(1));
    }
}
```



```

        result.set(3, new LongWritable(max));
        context.write(result);
    }
}

public static void main(String[] args) throws Exception {
    if (args.length != 2) {
        System.err.println("Usage: ScoreRank <in> <out>");
        System.exit(2);
    }
    JobConf job = new JobConf();
    job.setMapperClass(ScoreMapper.class);
    job.setReducerClass(ScoreReducer.class);
    // group and partition by the first key in the pair
    job.setPartitionerClass(FirstPartitioner.class);

    // the map output is Tuple<grade, class>, Tuple<name, score>
    job.setMapOutputKeyClass(Tuple.class);
    job.setMapOutputValueClass(Tuple.class);

    TableInputFormat.addInput(new TableInfo(args[0]), job);
    TableOutputFormat.addOutput(new TableInfo(args[1]), job);
    JobClient.runJob(job);
    System.exit(0);
}
}

```

要运行测试该程序，先准备两张表，创建表的 sql 如下：

```

create table mr_score_rank_in (name string, grade string, class string, score bigint);
create table mr_score_rank_out (name string, grade string, class string, score bigint);

```

然后准备以上示例数据，通过 Tunnel SDK 导入到表 `mr_score_rank_in` 中（这里不再详细说明，具体参见第一节）。本地编译运行 ok 后，打包在集群上运行如下：

```

add jar /home/admin/book/mr/score_rank.jar -f;

jar -libjars score_rank.jar -classpath /home/admin/book/mr/score_rank.jar
example.mapreduce.ScoreRank mr_score_rank_in mr_score_rank_out;

```

查看输出如下：

```
odps@ odps_book>read mr_score_rank_out;
```

name	grade	class	score
tinky	grade1	class1	80
dipsy	grade1	class2	90
melanie	grade2	class1	90
winky	grade2	class2	88

如上，结果和预期一致。这个例子只是给出了其中一种实现方式。实际上，由于该自定义的 Partitioner 实际上也是 HashPartitioner，只是它是对 key 的一部分求哈希，而不是对整个 key 求哈希。这里如果用默认 HashPartitioner 也是可以实现，你能想到怎么处理吗？

但是，在某些场景下，则必须自定义 Partitioner，比如执行二分操作对 $\text{key} \% 2$ 等。

下面我们将给出一些接近真实的场景模拟说明，可以帮助你理解如何通过 ODPS MR 处理数据分析问题。

4.3.2 示例：使用 DistributedCache

4.4 综合示例

4.4.1 查找淘宝用户共同特征

4.4.2 来往好友推荐

假设现在有这个场景，“来往”²⁶用户量持续猛涨，在很多情况下，用户不知道自己的好友也来啦，所以“来往”平台就想希望帮用户推荐好友，可以方便他们“say hi”，说不定还可以帮助用户找到“失散”多年的同学呢。生活总是如此充满惊喜。

如何推荐呢？最佳好友推荐往往是通过查找两个非好友之间的共同好友情况来实现的。如果小 A 和小 B 不是好友，但是他们有很多共同好友，那就可以给小 A 推荐小 B，给小 B 推荐小 A。这里，我们假定好友关系是双向的：如果小 A 是小 B 的好友，那么小 B 也是小 A 的好友。假设原始数据每条 Record 包含两个字段，user 和 friends，user 唯一标示一个用户，friends 是该用户的好友（逗号分隔），假设数据如下：

A B,C,D

B A,C

²⁶ 还没用“来往”？你 out 啦。它是阿里推出的为方便保留生活精彩时刻、方便和好友联系来往的社交软件，官方网址是：<https://www.laiwang.com/>

C A,B,D

D A,C,E

E D

期望生成的结果表包含三个字段，如下：

user1 user2 count

输出所有非好友的记录，count 表示 user1 和 user2 之间的共同好友数。在推荐时，我们可以根据共同好友数决定推荐列表中各项的顺序，比如共同好友数越多，排在推荐列表的位置越靠前。

Step 1: Map 阶段，每读取一条 Record，对 friends 进行 split，两两组合生成 key（按字母序），如果是好友，值置为 0，非好友则把值置为 1，如下：

第一条记录输出如下：

(A B), 0 ---- 0 表示已经是好友

(A C), 0

(A D), 0

(B C), 1 ---- 1 表示有一个共同好友

(B D), 1

(C D), 1

第二条记录输出如下：

(A B), 0 ---- 注意这里是 (A B)，而不是 (B A)，即前面提到的“按字母序”

(B C), 0

(A C), 1

第三条记录输出如下：

(A C), 0

(B C), 0

(C D), 0

(A B), 1

(A D), 1

(B D), 1

到这里已经很清楚了，为了节省篇幅，不再一一给出。

这里，key 包含的两个字段进行排序是关键。因为假定好友关系是双向（即无向）的，排序可以确保 (A B) 和 (B A) 是作为同一个 key (A B)，在 Combine 时处理更简单，同时输出到同一路 Reduce 进行归并。

Step 2: 由于这里同一个 map 中，相同 key 的输出很可能很多，所以这里使用 combine 来优化。通过 Combine，对 key 的 value 列表进行遍历，如果存在值为 0，则只输出一条值为 0；如果不存在值为 0，则把值进行相加，具体说明如下。

A B,C,D

B A,C

C A,B,D

D A,C,E

E D

map 输出后，key 为 (A B) 的键值对如下：

(A B), 0 ---- 来自第一条记录

(A B), 0 ---- 来自第二条记录

(A B), 1 ---- 来自第三条记录

combine 在遍历 (A B) 键值时，当读取到 (A B), 0 时，则跳出循环，输出一条 (A B), 0。

key 为 (B D) 的键值对如下：

(B D), 1 ---- 来自第一条记录

(B D), 1 ---- 来自第三条记录

combine 在遍历 (B D) 键值时，因为不存在值为 0 的，所以把键值相加，最后输出一条 (B D), 2。

其他键值对处理也是以此类推。最后，combine 输出的 <key, value> 中，如果 value 为 0，则表示 key 中的两个人是好友；则表示他们不是好友，value 值表示他们有多少个共同好友。

Step 3: Reduce 阶段的处理和 Combine 很类似，只是输出不同。对于同一个 key，当存在 0 值时，则不输出；当不存在 0 值时，则把所有值相加，最后把 key 赋给结果 record 的前两列，把 sum 值赋给结果 record 的 count 列，输出 record。

试想一下：在 Combine 时，对于存在值为 0 的键值对，是否可以不输出？比如，上面前面提到的 combine 输出 (A B), 0，是否也可以像 reduce 那样，不输出呢？为什么呢？

答案是不可以的。我们一起来分析原因。还是以 (A B) 键值对为例，假设有两个 map，map1 处理前两条记录，map2 处理后三条记录，如下：

(A B), 0 ---- 来自第一条记录， map1

(A B), 0 ---- 来自第二条记录， map1

(A B), 1 ---- 来自第三条记录， map2

按前面给出的处理逻辑，经过 combine 后，map1 输出 (A B), 0；map2 输出 (A B), 1；reduce 的输入是：

(A B), 0

(A B), 1

reduce 判断 (A B) 键存在 0 值, 所以不输出。

如果在 combine 时, map1 不输出 (A B), 0, 结果会怎么样呢?

reduce 的输入只有 (A B), 1, reduce 判断 (A B) 键不存在 0 值, 最后输出记录 (A B 1), 这是不对的, 因为 (A B) 本来是好友啊。

现在, 你明白为什么 combine 对于存在 0 值的键值对, 也要输出一条如 <key, 0> 的记录了吧? 因为多路 map 输出可能包含相同的 key, 所以需要保留可以标识 key 中两个用户是好友这个信息。

分析明白以后, 后面的代码实现就很轻松了。你可以试试自己练练手写一下。完整的 FriendRecommendation.java 程序清单如下:

```
package example.mapreduce;

import java.io.IOException;
import java.util.Arrays;

import com.aliyun.odps.Record;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.TableInfo;
import com.aliyun.odps.io.TableInputFormat;
import com.aliyun.odps.io.TableOutputFormat;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.mapreduce.CombineContext;
import com.aliyun.odps.mapreduce.Combiner;
import com.aliyun.odps.mapreduce.JobClient;
import com.aliyun.odps.mapreduce.JobConf;
import com.aliyun.odps.mapreduce.MapContext;
import com.aliyun.odps.mapreduce.Mapper;
import com.aliyun.odps.mapreduce.ReduceContext;
import com.aliyun.odps.mapreduce.Reducer;

public class FriendRecommendation {

    /*
    Record: A B, C, D
    map output:
    (A B), 0 ---- 0 表示已经是好友
    (A C), 0
    */
}
```

```

(A D), 0
(B C), 1 ---- 1 表示有一个共同好友
(B D), 1
(C D), 1
*/
public static class FriendMapper extends Mapper<Text, LongWritable> {

    private final static LongWritable zero = new LongWritable(0);
    private final static LongWritable one = new LongWritable(1);
    // For simpleness, use Text, set key as "A,B" instead of (A B)
    private Text key = new Text();

    @Override
    public void map(LongWritable recordNum, Record record,
        MapContext<Text, LongWritable> context) throws IOException,
        InterruptedException {
        String user = record.get(0).toString();
        String all = user + "," + record.get(1).toString();
        String[] arr = all.split(",");
        Arrays.sort(arr);
        int len = arr.length;
        for (int i=0; i<len-1; ++i) {
            for(int j=i+1; j<len; ++j) {
                key.set(arr[i] + "," + arr[j]);
                if(arr[i].equals(user)) {
                    context.write(key, zero);
                }
                else {
                    context.write(key, one);
                }
            }
        }
    }
}

public static class FriendCombiner extends Combiner<Text, LongWritable> {
    private LongWritable result = new LongWritable();

```

```
@Override
protected void combine(Text key, Iterable<LongWritable> values,
    CombineContext<Text, LongWritable> context) throws IOException,
    InterruptedException {
    long count = 0;
    for (LongWritable value : values) {
        if(0 == value.get()) {
            count = 0;
            break;
        }
        count += value.get();
    }
    result.set(count);
    context.write(key, result);
}

}

public static class FriendReducer extends Reducer<Text, LongWritable> {
    private LongWritable sum = new LongWritable();
    private Text user1 = new Text();
    private Text user2 = new Text();
    private Record result = null;

    @Override
    protected void setup(ReduceContext<Text, LongWritable> context)
        throws IOException, InterruptedException {
        result = context.createOutputRecord();
    }

    @Override
    public void reduce(Text key, Iterable<LongWritable> values,
        ReduceContext<Text, LongWritable> context) throws IOException,
        InterruptedException {
        int count = 0;
        for (LongWritable value : values) {
```

```
        if(0 == value.get()) {
            count = 0;
            break;
        }
        count += value.get();
    }
    // output the record if not friends
    if(count > 0) {
        sum.set(count);
        String user = key.toString();
        String[] users = user.split(",");
        user1.set(users[0]);
        user2.set(users[1]);
        result.set(0, user1);
        result.set(1, user2);
        result.set(2, sum);

        context.write(result);
    }
}

public static void main(String[] args) throws Exception {
    if (args.length != 2) {
        System.err.println("Usage: FriendRecommendation <in_table> <out_table>");
        System.exit(2);
    }
    JobConf job = new JobConf();
    job.setMapperClass(FriendMapper.class);
    job.setCombinerClass(FriendCombiner.class);
    job.setReducerClass(FriendReducer.class);

    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);

    TableInputFormat.addInput(new TableInfo(args[0]), job);
    TableOutputFormat.addOutput(new TableInfo(args[1]), job);
}
```



```
        JobClient.runJob(job);
    }
}
```

要运行该 MR 程序，首先创建两张表，SQL 如下：

```
create table mr_friend_in (user string, friends string);
create table mr_friend_out (user1 string, user2 string, count bigint);
```

然后通过 Tunnel SDK 导入数据（具体见之前的 WordCount 示例），导入后，原始表如下：

```
odps@ odps_book>read mr_friend_in;
```

user	friends
A	B,C,D
B	A,C
C	A,B,D
D	A,C,E
E	D

把 MR 程序打包，通过 odpscmd 执行以下命令，

```
add jar /home/admin/book/mr/mr_friend.jar -f;
jar -libjars mr_friend.jar -classpath /home/admin/book/mr/mr_friend.jar
example.mapreduce.FriendRecommendation mr_friend_in mr_friend_out;
```

执行完成后，生成结果表如下：

```
odps@ odps_book>read mr_friend_out;
```

user1	user2	count
A	E	1
B	D	2
C	E	1

我们在前面已经提到，MR 编程的关键在于设计 Map 的输出和 Reduce 的输入。实际上，在很多情况下，由于设计的 Map 输出不同，所以实现方式也不同。在尝试不同的实现方法过

程中，也许会给你带来很多收获。在这个例子中，你想到其他解决方案了吗？

4.4.3 家庭树

家庭树是个使用 MapReduce Join 的一个非常经典的例子，据说面试 MapReduce 时常常会考它。我觉得这个场景很有趣，而且对于了解本教程下一章要介绍的 ODPS SQL 连接也很有帮助。当然，在实际处理中，这种场景一般选择使用 ODPS SQL 来完成（我也推荐这么做），实际上 SQL 处理幕后也是通过 MapReduce 来完成的，而且做了很多优化。

这里介绍本示例的初衷还在于尝试回答前面提到的“如何实现 Map 和 Reduce？”这一问题。下面，我们一起来看看场景。

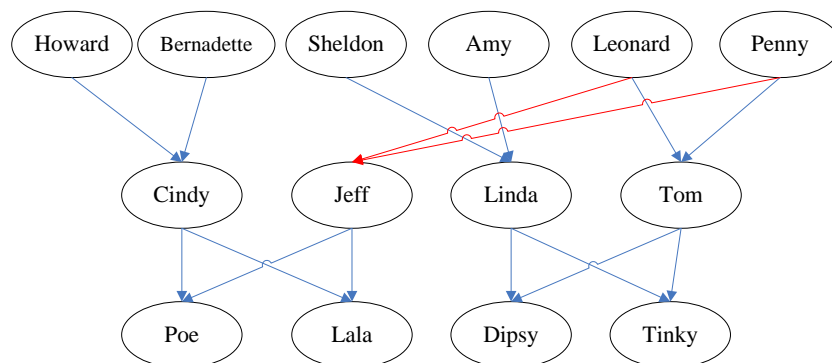
假设原始数据表包含两个字段：child 和 parent，数据如下：

Linda	Sheldon
Linda	Amy
Tom	Penny
Tom	Leonard
Jeff	Penny
Jeff	Leonard
Lala	Jeff
Lala	cindy
Poe	Jeff
Poe	cindy
Tinky	Linda
Tinky	Tom
Dipsy	Linda
Dipsy	Tom
Cindy	Howard
Cindy	Bernadette

希望能够找到家族树，输出结果表包含 child，parent 和 grandparent 三个字段，比如：

Dipsy	Linda Sheldon
Tinky	Linda Sheldon
Tinky	Tom Penny
...	

由于这里样例数据很小，为了便于查看数据及验证后面的输出结果正确，我们先画个族谱图，如下：



下面我们来分析一下。该场景相当于单表连接，比如对于以下三条记录：

Linda Sheldon

Dipsy Linda

Tinky Linda

执行连接后，应该输出：

Dipsy Linda Sheldon

Tinky Linda Sheldon

显然，上面的结果是通过 Linda 这个值来连接的。那么，在程序中，应该如何连接呢？对于单表连接（自连接），一般是实现成两路输入进行关联，通常把这两路分别称为左表和右表。在这个例子中，在左表中，把 child 字段（如 Linda）作为 key，parent 字段（如 Sheldon）作为值；在右表中，把 parent 字段（如 Linda）作为 key，child 字段（如 Dipsy）作为值。由于 map shuffle 处理之后，会把相同的 key 的中间结果输出到同一路 Reducer 中，假设 key 为 Linda 的中间结果输出到 Reducer1 中，如下：

Linda Sheldon

Linda Dipsy

Linda Tinky

问题来了，现在怎么知道 Sheldon 是 Linda 的 parent 呢，还是 child 呢？为了解决这个问题，我们需要区分中间结果是左表和右表。如果对 key 添加标志来区分，比如左表输出表示成 Linda_1 Sheldon，右表输出表示成 Linda_2 Dipsy，这样就会导致 key 不同，无法输出到同一个 Reducer 中进行关联。所以，这里不能按 key 来区分，而是需要对 value 添加标识来区分，比如在左表的 value 值都加上 1，右表加上 2，如下：

Linda Sheldon1

Linda Dipsy2

Linda Tinky2

这样相同 key 输出到同一路 Reducer，我们可以通过 value 值的最后一位判断是该 value 值是 key 的 child 还是 parent，把 value 值最后一位为 2（去掉最后一位）的全部放到数组 childArray 中，最后一位为 1（去掉最后一位）的全部放到数组 grandpaArray 中，最后对两个数组求笛

卡尔积，把 child 项作为结果记录的 child 列，grandpa 项作为结果记录的 grandparent 列，key 作为 parent 列，最后输出结果，如下：

Dipsy Linda Sheldon

Tinky Linda Sheldon

好了，分析到这里已经很明白了，下面我们来练练手，一起实现它吧。FamilyTree.java 的完整的程序清单如下：

```
package example.mapreduce;

import java.io.IOException;
import java.util.ArrayList;
import com.aliyun.odps.Record;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.TableInfo;
import com.aliyun.odps.io.TableInputFormat;
import com.aliyun.odps.io.TableOutputFormat;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.mapreduce.JobClient;
import com.aliyun.odps.mapreduce.JobConf;
import com.aliyun.odps.mapreduce.MapContext;
import com.aliyun.odps.mapreduce.Mapper;
import com.aliyun.odps.mapreduce.ReduceContext;
import com.aliyun.odps.mapreduce.Reducer;

public class FamilyTree {

    private final static String LEFT = "1";
    private final static String RIGHT = "2";

    public static class FamilyMapper extends Mapper<Text, Text> {

        private Text child = new Text();
        private Text parent = new Text();

        @Override
        public void map(LongWritable recordNum, Record record,
            MapContext<Text, Text> context) throws IOException,
            InterruptedException {
```

```
String childName = record.get(0).toString();
String parentName = record.get(1).toString();

//left table: <key,value> = <child,parent>
child.set(childName);
parent.set(parentName + LEFT);
context.write(child, parent);

// right table: <key,value> = <parent, child>
child.set(childName + RIGHT);
parent.set(parentName);
context.write(parent, child);
}
}

public static class FamilyReducer extends Reducer<Text, Text> {
    private Record result = null;
    private Text child = new Text();
    private Text grandpa = new Text();
    private ArrayList<String> childArr = new ArrayList<String>();
    private ArrayList<String> grandpaArr = new ArrayList<String>();

    @Override
    protected void setup(ReduceContext<Text, Text> context)
        throws IOException, InterruptedException {
        result = context.createOutputRecord();
    }

    @Override
    public void reduce(Text key, Iterable<Text> values,
        ReduceContext<Text, Text> context) throws IOException,
        InterruptedException {
        childArr.clear();
        grandpaArr.clear();
        for (Text val : values) {
            String value = val.toString();
            if(value.endsWith(LEFT)) {
```

```
        grandpaArr.add(value.substring(0, value.length()-1));
    }
    else if(value.endsWith(RIGHT)){
        childArr.add(value.substring(0, value.length()-1));
    }
    else {
        throw new IOException("invalid value:" + value + ",key:" + key.toString());
    }
}
for(int i=0; i<childArr.size(); ++i) {
    for(int j=0; j<grandpaArr.size(); ++j) {
        child.set(childArr.get(i));
        grandpa.set(grandpaArr.get(j));
        result.set(0, child);
        result.set(1, key);
        result.set(2, grandpa);
        context.write(result);
    }
}
}
}

public static void main(String[] args) throws Exception {
    if (args.length != 2) {
        System.err.println("Usage: FamilyTree <in_table> <out_table>");
        System.exit(2);
    }
    JobConf job = new JobConf();
    job.setMapperClass(FamilyMapper.class);
    job.setReducerClass(FamilyReducer.class);

    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    TableInputFormat.addInput(new TableInfo(args[0]), job);
    TableOutputFormat.addOutput(new TableInfo(args[1]), job);

    JobClient.runJob(job);
}
```

```
}  
}
```

现在，我们来准备运行该程序。首先，创建两张表，SQL 如下：

```
create table mr_family_in (child string, parent string);  
create table mr_family_out (child string, parent string, grandparent string);
```

然后，通过 Tunnel SDK 导入数据，原始数据表如下：

```
odps@ odps_book>read mr_family_in;  
  
+-----+-----+  
| child   | parent |  
+-----+-----+  
| Tinky   | Linda  |  
| Tinky   | Tom    |  
| Dipsy   | Linda  |  
| Dipsy   | Tom    |  
| Linda   | Amy    |  
| Linda   | Sheldon|  
| Tom     | Penny  |  
| Tom     | Leonard|  
| Jeff    | Penny  |  
| Jeff    | Leonard|  
| Lala    | Jeff   |  
| Lala    | Cindy  |  
| Poe     | Jeff   |  
| Poe     | Cindy  |  
| Cindy   | Howard |  
| Cindy   | Bernadette|  
+-----+-----+
```

MR 程序打包后，运行如下命令：

```
add jar /home/admin/book/mr/mr_family.jar -f;  
jar -libjars mr_family.jar -classpath /home/admin/book/mr/mr_family.jar  
example.mapreduce.FamilyTree mr_family_in mr_family_out;
```

运行成功后，读取结果表如下：

```
odps@ odps_book>read mr_family_out;
```

child	parent	grandparent
Poe	Cindy	Bernadette
Poe	Cindy	Howard
Lala	Cindy	Bernadette
Lala	Cindy	Howard
Lala	Jeff	Penny
Lala	Jeff	Leonard
Poe	Jeff	Penny
Poe	Jeff	Leonard
Tinky	Linda	Amy
Tinky	Linda	Sheldon
Dipsy	Linda	Amy
Dipsy	Linda	Sheldon
Dipsy	Tom	Leonard
Dipsy	Tom	Penny
Tinky	Tom	Leonard
Tinky	Tom	Penny

上面的示例，可以 SQL 实现如下：

```
select a.child, a.parent, b.parent as grandparent from mr_family_in a join mr_family_in b on  
a.parent = b.child;
```

但是实际上，在 ODPS SQL 内部实现中，自连接不是这么处理的，它不需要对数据值添加标识来确定来自哪一路输出，而是通过 DAG 图来确定，具体细节这里就不探讨了。

4.5 性能调优

一般来说，当 ODPS MR Job 运行很慢，或者超出默认设置导致 Job 失败时，需要执行性能调优。性能调优这方面我经验不多，暂时还没有找到合适的例子来逐步说明。以下先简单给出一些说明和建议，可能有些枯燥，不过当你真正遇到性能问题时，我想下面的内容或许会有些帮助。

4.5.1 代码实现逻辑

在我看过的性能问题中，很大一部分是代码实现问题。很多 ODPS MR 的用户代码实现逻辑比较复杂，因此更需要注意以下一些问题：

1. 重用 Writable 对象

尽量不要在 map 和 reduce 方法使用去 new 一个 Writable 对象。

比如在 WordCount 示例中，应该使用如下代码：

```
public static class WCMapper extends Mapper<Text, LongWritable> {  
  
    private final static LongWritable one = new LongWritable(1);  
    private Text word = new Text();  
  
    @Override  
    public void map(LongWritable recordNum, Record record,  
        MapContext<Text, LongWritable> context) throws IOException,  
        InterruptedException {  
        ...  
    }  
}
```

注意，阴影部分显示的 new 两个 Writable 实例对象，不要放到 map 方法中执行，否则就会创建 recordNum 个存活时间非常短的对象，直到 Java 垃圾回收机制来收集处理。

2. 尽量减少使用 string.split 方法

由于前面给出的示例为了尽量简单明白，都是构造小数据集来说明，可能没有太注意这些细节问题，但是在实际处理大数据时，还是需要特别注意。

3. string 的 += 连接方法，在字符串非常大时，可能会导致内存溢出，建议使用 StringBuilder 来处理。在简短的 string 连接中，建议不要使用 StringBuffer.append 方式处理。
4. 使用最紧凑的 Writable 来存储数据类型，比如在处理 protobuf 数据时，中间结果可以采用 BytesWritable 类型，最后结果用 Text 类型来保存 String 类型数据。

4.5.2 使用 Combiner

对于 Combiner，相信你已经不陌生了，它相当于在 Mapper 端执行局部 Reduce，合并结果，减少从 Mapper 端到 Reducer 端的数据传输。具体示例参见前面的带 Combiner 的 WordCount 示例。

4.5.3 使用 Distributed Cache

我们知道，对于 MapReduce 操作而言，中间结果 Shuffle 数据，当数据量很大时，往往会成为性能瓶颈，因为 shuffle 需要通过网络传输数据。在执行一个大表和一个表（字典表或资源文件）的表连接操作时，往往会使用 Distributed Cache 把小表加载到缓存中，通过这种方式，ODPS MR 在 Job 执行之前会把把这些小表（或资源文件）拷贝到所有的 worker 上并加载到缓存中，这样在后续连接操作时就不再需要拷贝大表，对于大表相当于直接在本地执行，不用 shuffle 数据，这样可以很大提高性能。

注意，小表的限制是 512M。在实际应用中，Distributed Cache 应用很广泛。示例 1.3.4 给出

了 DistributedCache 的使用方式。

4.5.4 合理配置作业参数

对性能有所影响的 JobConf 配置项包括：

- setMinSplitSize(long) // 最小输入表切分大小，单位 MB，大于 0，默认 256，不能大于 max split size
- setMaxSplitSize(long) // 最大输入表切分大小，单位 MB，大于 0，默认 640，不能小于 min split size
- setNumMapTasks(int) // Map 个数，【0， 100000】之间，默认-1
- setNumReduceTasks(int) // Reduce 个数，【0， 2000】之间，默认为 Map 数的 1/4
- setCPUForMapTask(int) // Map CPU 资源，100 为 1cpu 核，【50， 800】之间，默认 100
- setCPUForReduceTask(int) // Reduce CPU 资源，100 为 1cpu 核，【50， 800】之间，默认 100
- setMemoryForMapTask(long) // Map 内存资源，单位 MB，【256M， 12G】之间，默认 2048M
- setMemoryForReduceTask(long) // Reduce 内存资源，单位 MB，【256M， 12G】之间，默认 2048M
- setSortMB(int) // Map 做分区排序时内存 Buffer 大小，单位 MB，【64， Map 内存】之间，默认为 100
- setIndexPercentForSortMB(float) // Map 排序内存 Buffer 索引区的比例，用于调优，取值范围：(0.0, 1.0)，默认 0.05

通常情况下：

1. 如果 map 阶段慢，可以考虑使用 setMinSplitSize 方法减少最小切分大小，提高 map 数解决；
2. 如果作业返回的 Counters 中 map_spill_files 值很大，可以考虑使用 setSortMB 方法提升 Map 做分区排序时的内存大小，或者使用 setIndexPercentForSortMB 方法调整排序内存索引区的比例，减少 dump 到外存的次数；
3. 如果 reduce 阶段慢，可以考虑通过 setNumReduceTasks 方法设大 reduce 数量；
4. 默认 map/reduce 都是 2G 内存，如果 Counters 中的 map_max_memory 和 reduce_max_memory 接近 2G 时，可以考虑提高内存；

下面这个示例给出了如何查看 Job 运行慢的原因以及如何调整参数配置。

批注 [a3]: 示例

4.5.5 减少输入的数据量

数据量大时，读取磁盘中的数据可能耗费一部分处理时间，因此，减少需要读取的数据字节数可以提高总体的吞吐量，从而提高作业性能。可供选择的方法有如下几种：

-
1. 减少输入数据量：对某些决策性质的应用，处理数据采样后子集所得到的结果只可能影响结果的精度，而并不会影响整体的准确性，因此可以考虑先对数据进行特定采样后再导入输入表中进行处理，可以尝试使用 采样功能；
 2. 避免读取用不到的字段：ODPS M/R 框架的 TableInfo 类支持读取指定的列（以列名数组方式传入），而非整个表或表分区，这样也可以减少输入的数据量，提高作业性能。

4.5.6 数据倾斜怎么办？

另外一种常见的性能问题：**数据倾斜**，反应到 Counters 就是 instance 的最大运行时间远远超出平均运行时间：`reduce_max_time >> reduce_avg_time` 这时可以查看作业日志 (Logview)，找到运行时间长的 instance，跟其他 instance 的 counters 进行对比，就可以确认是否发生数据倾斜了。

数据倾斜的原因通常是某些 key 对应的记录数远远超出其他 key，这些 key 被分到少量的 reduce 处理，从而导致这些 reduce 相对于其他运行时间长很多，解决方法：

1. 可以试试 Combiner，把这些 key 对应的记录进行 Local-Reduce，减少后面 reduce 处理的记录数；
2. 改进业务逻辑。

4.6 FAQ

以下是和 ODPS MapReduce 相关的一些常见用户问题：

1. MR 任务的输入表在指定分区的时候，只能用 `pt=1` 这种形式么？能否用 `pt like %` 或者 `pt>1` 这样？

只能是 `pt=1` 这种形式。

2. 是否支持动态分区？

不支持。

（待补充）

第5章 ODPS SQL

ODPS 是基于飞天内核构建的海量数据处理和分析的服务平台，和传统的数据库不同，没有数据库的事务、主键约束这些特性。ODPS SQL 是 ODPS 提供的数据分析处理模块，是一种用于查询和分析存储在 ODPS 中的大规模数据的机制。

ODPS SQL 采用类似 SQL 的语法，通过底层飞天 MapReduce 框架完成数据计算，适用于数据量大（TB 级别）、实时性要求不高的场合，因为作业准备、提交等阶段需要花费数秒的时间。

ODPS SQL 它在语法上和 Hive HQL 非常接近，熟悉 SQL 或 HQL 的编程人员会发现 ODPS SQL 很容易上手。

5.1 保留字和运算符

ODPS SQL 定义了一些保留字，对表、列或分区命名时请不要使用。保留字不区分大小写。它支持关系运算符、算术运算符、位运算符、逻辑运算符。

为了避免过于枯燥，这里不一一列出保留字和运算符，具体请参考 ODPS SQL 在线手册²⁷。

5.1 主要功能

本章最开头泛泛几个字：“用于查询和分析存储在 ODPS 中的大规模数据”，也许你已经迫不及待地想知道“如何通过 ODPS SQL 来查询分析？如何执行？”别着急，我们将在后面的示例介绍中逐步手把手教你。这里先简单介绍一下 ODPS SQL 的主要功能，从而有个初步认识：

- 通过 Create、Drop 和 Alter 对表和 Partition 进行管理
- 通过 Select 选择表中的某几条记录
- 通过 Where 语句查看满足条件的记录，实现过滤功能
- 通过等值连接 Join 实现两张表的关联
- 通过对某些列 Group By，实现聚合操作，如求 Sum
- 通过 Insert 把结果记录插入到另一张表中
- 通过内置函数和自定义函数（UDF）来实现计算
- 支持收集表的统计信息和设置表生命周期
- 支持正则表达式

批注 [a4]: 没有例子说明

批注 [a5]: 例子说明

下面，我们将通过各个示例分析来详细说明。你可以照着这些示例练练手，这样就可以很快实现 ODPS SQL 入门。

²⁷ ODPS SQL 在线手册地址：http://odps.alibaba-inc.com/doc/prddoc/odps_sql/index.html

5.2 示例场景说明

为了使后面的示例说明尽量生动些，这里我不想从 ODPS SQL 语法角度来逐个介绍各个用法，而是构造场景和需求，逐步介绍如何通过 ODPS SQL 实现。为了用户体验更顺畅，这里所有用到的表都给出建表语句。

假设场景是有这样一份 Web log，log 数据主要包含前面介绍的各个字段，数据每天都上传到 ODPS 中。还有一份用户表 user，包括如下信息：

- user_id BIGINT，用户 ID，唯一标识一个用户
- gender BIGINT， 性别，0 未知， 1 男， 2 女
- age BIGINT，用户年龄
- active BIGINT， 活跃度，0 未知，1 活跃， 2 不活跃

5.3 简单的 DDL 操作

后面会介绍通过 ODPS SQL 完成一些简单的 DDL 操作，包括创建表、添加分区、查看表和分区、修改表、删除表和分区。

5.3.1 创建表

首先，创建前面提到的 page_view 表，按照 dt（日期）和 country（国家）进行分区，建表语句如下：

```
CREATE TABLE page_view(  
    user_id BIGINT,  
    view_time BIGINT,  
    page_url STRING,  
    referrer_url STRING,  
    ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING);
```

在创建表时，需要注意的几点是：

- 表名、列名以及 SQL 保留字都是大小写不敏感，只能包含 a-z,A-Z 以及数字和下划线_，且必须以字母开头
- 如果 page_view 表已经存在，以上建表语句会报错；如果希望不要报错（即不存在才创建），可以使用 IF NOT EXISTS 选项，如 CREATE TABLE IF NOT EXISTS page_view
- PARTITIONED BY 关键字指定表的分区，这里分区键是 dt 和 country 的组合，一级分区是 dt，二级分区是 country；目前分区键只支持 STRING 类型

如果要创建一个和 page_view 有相同 Schema 的表，比如创建 page_view_test 用于测试，可以通过 Like 执行，SQL 语句如下：

```
CREATE TABLE page_view_test LIKE page_view;
```

通过 Create ... Like 方式建表很简单，新创建的表会复制对应表的 Schema，但不会复制任何数据。

如果希望选中已有表的某几个字段，比如想对上述示例的 url 进行分析，选中 page_url 和 referrer_url 两个字段创建一张新表 page_view_url，SQL 语句如下：

```
CREATE TABLE page_view_url AS  
SELECT page_url, referrer_url FROM page_view;
```

和 Create ... Like 方式不同，Create ... AS Select 方式建表会把数据复制到新表中，但它不会复制原有表结构。在上面的建表语句中，page_view_url 表会包含 2 个字段，page_url STRING 和 referrer_url STRING，没有分区。Create ... AS Select 不支持对生成的结果表创建分区。

由于 ODPS 表没有主键，所以 Create ... AS Select 可能会存在重复记录，比如原有 page_view 表包含以下两条记录：

```
123 1386213971 http://www.tmall.com http://www.taobao.com/ 192.91.189.6  
798 1387213003 http://www.tmall.com http://www.taobao.com/ 112.92.142.7
```

则 page_view_url 表中会包含以下两条重复记录：

```
http://www.tmall.com http://www.taobao.com/  
http://www.tmall.com http://www.taobao.com/
```

这里，我们也创建一张前面场景描述的提到用户表，SQL 如下：

```
CREATE TABLE user(  
    user_id BIGINT,  
    gender BIGINT COMMENT '0 Unknown, 1 Male, 2 Female',  
    age BIGINT,  
    active BIGINT);
```

5.3.2 添加分区

假设要往 page_view 的表导入 dt='2011-12-17',country='US'的数据，我们应该先给 page_view 表添加分区，SQL 如下：

```
ALTER TABLE page_view ADD PARTITION (dt='2011-12-17',country='US');
```

和创建表类似，如果分区已存在，以上命令会报错。如果希望仅在分区不存在的情况下创建，可以添加 IF NOT EXISTS，执行如下 SQL：

```
ALTER TABLE page_view ADD IF NOT EXISTS PARTITION (dt='2011-12-17',country='US');
```

5.3.3 查看表和分区

查看 Project 下的所有表:

```
SHOW TABLES;
```

它会列出所有表, 当表很多时, 往往会非常长, 如果想只列出前缀为 `page` 的表, 可以通过正则表达式, 如下:

```
SHOW TABLES 'page.*';
```

要想查看 `page_view` 表下的所有分区, SQL 如下:

```
SHOW PARTITIONS page_view;
```

要想查看表 `page_view` 的结构信息, 可以执行 SQL:

```
DESCRIBE page_view; 或
```

```
DESC page_view;
```

5.3.4 修改表

修改表使用关键字 `ALTER TABLE`, 其实前面给出的添加分区也是用 `ALTER TABLE`, 除此之外, 还可以通过 `ALTER TABLE` 修改列名、添加列、修改注释等。假设对于 `user` 表, 想给它添加 `info` 列, 保存用户的其他信息, 可以执行如下 SQL:

```
ALTER TABLE user ADD COLUMNS(info STRING);
```

前面在创建 `user` 表时, 没有说明 `active` 字段的含义, 为了避免以后看到该字段值 “0、1、2” 不知啥含义, 我们现在就赶紧给它加个注释吧:

```
ALTER TABLE user CHANGE COLUMN active active BIGINT COMMENT '0 Unknown, 1 Active, 2 Not-active';
```

注释在 `DESC` 查看表信息时会显示出来。

5.3.5 删除表和分区

删除表操作比较简单, 直接使用 `DROP TABLE tablename;` 即可。比如 SQL 如下:

```
DROP TABLE user;
```

在 `console` 交互模式下运行该 SQL, 会弹出确认信息: `Confirm to "DROP TABLE user;" (yes/no)?`, 输入 `y` 回车就确认删除。

在非交互模式下执行该 SQL, 则不会弹出确认信息, 执行删表。

执行删除表之后, 会清除该表的所有数据, 所以务必三思而后行, “Think before you type!”

和删除表不同, 删除分区也需要用 `ALTER TABLE`, 假设要删除 `dt='20130111',country='US'`, 执行 SQL 如下:

```
ALTER TABLE page_view DROP PARTITION(dt='20130111',country='US');
```

5.4 生成数据

可以通过 ODPS Tunnel 导入数据，具体详见本教程的姐妹篇 ODPS Tunnel Tutorial。

在学习 ODPS SQL 时，除非仅仅是为了验证语法正确性，否则我非常不建议在空表上跑 SQL，因为这样根本无法判断写的 SQL 是否正确，是否达到预期效果。而且，执行任何 SQL，（如果语法正确的话）输出几乎都是空（除了 Count 之类），学起来也会比较无聊。所以，强烈建议还是花些时间弄点数据进去。这里，出于简单，我们通过 INSERT INTO 来造几条测试数据，这也在简单测试时经常被用到的一种方式。

先给 user 表造数据，执行 SQL 如下：

```
INSERT INTO TABLE user
SELECT 123, 1, 15, 1, "test1"
FROM (SELECT count(*) FROM user)a;28
```

该 SQL 会插入一条记录 123, 1, 15, 1, "test1" 到 user 表中。注意，这里，FROM 的表中有几条数据，就会插入几条数据到 user 表中，由于 count(*) 结果必然是一条数据，所以当插入一条数据时，经常这么写。你可以多次修改 SELECT 后的给定值，插入不同记录。

给 page_view 表造数据，执行 SQL 如下：

```
INSERT INTO TABLE page_view PARTITION(dt='2011-12-17', country='US')
SELECT 123, 1386213971, "http://www.tmall.com", "http://www.taobao.com/", "192.91.189.6"
FROM (SELECT count(*) FROM user)a;
```

5.5 查询分析 DML

这里，从简单查询开始，一步一步逐渐提高复杂度，介绍了在查询分析中使用的各种函数以及一些关键字的区别。

5.5.1 简单查询

假设要查看所有活跃用户的信息，执行 SQL 如下：

```
SELECT user.*
FROM user
WHERE user.active=1;
```

²⁸ 这里 From 不是直接跟表名，而是一个 SELECT 查询，这称为子查询，具体见后面的“子查询”一节说明。

该查询会在屏幕上输出结果，如果是查询调试小表，经常会这么做，但是很多时候查询结果太大，直接输出没法看，需要把它写到另一张表中，可以执行如下命令：

```
CREATE TABLE user_active LIKE user;

INSERT OVERWRITE TABLE user_active
SELECT user.*
FROM user
WHERE user.active=1;
```

这里，我们先创建 `user_active` 表，然后通过 `INSERT OVERWRITE` 把结果数据保存到该表中。`INSERT OVERWRITE` 在写入数据之前，会清空目标表的原始数据。我们在前面介绍了 `CREATE ... AS SELECT`，这里通过它也可以实现同样的目的：

```
CREATE TABLE IF NOT EXISTS user_active AS

    SELECT user.*
    FROM user
    WHERE user.active=1;
```

不过，并不是所有 `INSERT OVERWRITE` 都可以用 `CREATE ... AS SELECT` 来替换，`INSERT OVERWRITE` 还支持写入某个 `Partition`。另外，ODPS SQL 还支持 `INSERT INTO`，它会向表或表的 `Partition` 中追加数据。

5.5.2 分区表查询

假设要查询 2011-12-17 这一天 `referrer_url` 来自 `taobao.com` 这个域名的 PV 情况，可以执行如下 SQL：

```
SELECT page_view.*
FROM page_view
WHERE page_view.dt='2011-12-17' AND
      page_view.referrer_url LIKE '%taobao.com';
```

注意，对于分区表的 `<table>.*`（比如这里 `page_view.*`）查询，会包含分区表中的 `Partition key`（`dt` 和 `country`），因此这里 `SELECT page_view.*` 输出内容包含 7 列（其中 `Partition key` 占两列），而不是 5 列。因为传统数据库没有 `Partition`，所以习惯传统 SQL 的用户，使用 ODPS SQL 需要特别注意这点。下面先创建两张表，一张带 `Partition`，一张不带 `Partition`，SQL 如下：

```
CREATE TABLE from_taobao_no_partition(
    user_id BIGINT,
    view_time BIGINT,
```

```
page_url STRING,  
referrer_url STRING,  
ip STRING);
```

```
CREATE TABLE from_taobao_with_partition LIKE page_view;
```

如果执行下面的 SQL，把查询结果保存到表 `taobao_no_partition`，如下：

```
INSERT OVERWRITE TABLE from_taobao_no_partition  
SELECT page_view.*  
FROM page_view  
WHERE page_view.dt='2011-12-17' AND  
page_view.referrer_url LIKE '%taobao.com/';
```

执行该 SQL，会报以下错误信息：

FAILED: ODPS-0130071:Semantic analysis exception - Cannot insert into target table because column number/types are different : line 1:23 'from_taobao_no_partition': Table insclause-0 has 5 columns, but query has 7 columns.

这里，由于 `from_taobao_no_partition` 表只有 5 个列（没有 Partition 列），所以需要把 SQL 改成如下：

```
INSERT OVERWRITE TABLE from_taobao_no_partition  
SELECT pv.user_id, pv.view_time, pv.page_url, pv.referrer_url, pv.ip  
FROM page_view pv  
WHERE pv.dt='2011-12-17' AND  
pv.referrer_url LIKE '%taobao.com/';
```

如果要把查询结果保存到前面创建的带 Partition 的表 `from_taobao_with_partition` 中，执行 SQL 如下：

```
INSERT OVERWRITE TABLE from_taobao_with_partition  
PARTITION(dt='2011-12-17',country='US')  
SELECT pv.user_id, pv.view_time, pv.page_url, pv.referrer_url, pv.ip  
FROM page_view pv  
WHERE pv.dt='2011-12-17' AND pv.country='US' AND  
pv.referrer_url LIKE '%taobao.com/';
```

```

INSERT OVERWRITE TABLE from_taobao_with_partition
    PARTITION(dt='2011-12-17',country='China')
SELECT pv.user_id, pv.view_time, pv.page_url, pv.referrer_url, pv.ip
FROM page_view pv
WHERE pv.dt='2011-12-17' AND pv.country='China' AND
    pv.referrer_url LIKE '%taobao.com/';

```

这里假定 dt='2011-12-17'下只有 country='US'和 country='China'两个国家，如果包含的国家更多，这么写是不是过于繁琐？另外，如果预先不知道dt='2011-12-17' 的分区下有哪些 Partition 呢？ODPS SQL 的动态分区近乎完美地解决了这个问题，我们将在后面详细说明它。

5.5.3 连接操作 Join

假设要查看 2011-12-17 这一天不同性别和年龄的网站 PV 访问量，就需要把 page_view 表和 user 表通过 Join 进行连接，SQL 语句如下：

```

CREATE TABLE pv_users(
    user_id BIGINT,
    view_time BIGINT,
    page_url STRING,
    referrer_url STRING,
    ip STRING COMMENT 'IP Address of the User',
    gender BIGINT COMMENT '0 Unknown, 1 Male, 2 Female',
    age BIGINT)
PARTITIONED BY(dt STRING, country STRING);

INSERT OVERWRITE TABLE pv_users PARTITION(dt,country)
SELECT pv.user_id, pv.view_time, pv.page_url, pv.referrer_url, pv.ip,
    u.gender, u.age,
    pv.dt, pv.country
FROM user u JOIN page_view pv ON (pv.user_id = u.user_id)
WHERE pv.dt = '2011-12-17';

```

诚如前面提到的，这里给出建表语句是为了当用户真正去体验运行这些示例时，会更顺畅，后面如有类似情况不再一一说明。

和 Hive 类似，ODPS SQL 的多表连接需要用 Join...On 语句，不支持以下这样的传统数据库 SQL：

```

FROM user u, page_view pv
WHERE pv.dt = '2011-12-17' and pv.user_id = u.user_id

```

此外，On 后面的条件必须是等值判断，比如以下写法也不对，

ON (pv.user_id < u.user_id)

以上 Join ... On 连接通常称为内连接，对两个表中 user_id 相同的记录进行连接。和传统 SQL 类似，ODPS SQL 也提供了三种外连接，分别是左连接(LEFT OUTER JOIN)、右连接(RIGHT OUTER JOIN) 和全连接 (FULL OUTER JOIN)。内连接只返回两张表中满足条件的记录，外连接相当于是对内连接的扩展，左连接会返回坐标中所有记录，即使右表没有对应记录；右连接会返回右表中的所有记录，即使左表中没有对应记录；全连接会返回两张表的所有记录。举个简单的例子，假设有以下两张表 t1 和 t2，结构和数据分别如下

表 t1

id	name
1	a1
2	b1

表 t2

id	score
1	80
3	90

执行内连接 Join，SQL 和结果分别如下：

SELECT t1.*, t2.score FROM t1 JOIN t2 ON (t1.id = t2.id);

id	name	score
1	a1	80

执行左连接，SQL 和结果分别如下：

SELECT t1.*, t2.score FROM t1 RIGHT OUTER JOIN t2 ON (t1.id = t2.id);

id	name	score
1	a1	80
2	b1	NULL

执行右连接，SQL 和结果分别如下：

SELECT t1.*, t2.score FROM t1 RIGHT OUTER JOIN t2 ON (t1.id = t2.id);

id	name	score
1	a1	80
NULL	NULL	90

是不是觉得和期望不太一致，本来期望第二条输出是 3, NULL, 90。为什么结果是 NULL, NULL, 90 呢？原来是因为 SELECT t1.*，它选择的 t1 的 id，所以我们改一下 SQL：

SELECT t2.id, t1.name, t2.score FROM t1 RIGHT OUTER JOIN t2 ON (t1.id = t2.id);

其输出结果就和期望的一致。

同样，执行全连接，SQL 和结果分别如下：

SELECT t1.*, t2.score FROM t1 FULL OUTER JOIN t2 ON (t1.id = t2.id);

id	name	score
1	a1	80
2	b1	NULL
NULL	NULL	90

也许这里第三条记录输出和期望又不一致了，怎么办？感兴趣的话，你可以自己练练手试试。

通过以上简单的示例说明，相信你已经很明白这些连接的区别了。

此外，如果要连接两张以上的表，可以执行如下 SQL：

```
INSERT OVERWRITE TABLE pv_friends
SELECT pv.user_id, pv.view_time, pv.page_url, pv.referrer_url, pv.ip,
       u.gender, u.age, f.friends
FROM user u JOIN page_view pv ON (pv.user_id = u.user_id)
       JOIN friend_list f ON (u.user_id = f.uid)
WHERE pv.dt = '2011-12-17';
```

5.5.4 MAPJOIN

当一个大表和一个或多个小表 JOIN 时，可以使用 MAPJOIN，性能上会比前面介绍的普通 JOIN 快很多。对于普通 JOIN，对两张表都执行 Mapper 后，对 Mapper 结果都进行 Shuffle，给 Reducer 进一步执行，Shuffle 是执行网络传输，所以对于大表小表 JOIN，普通 JOIN 方式在网络上传输的数据比较大。而对于 MAPJOIN，它直接把小表拷贝到各台计算机（TaskWorker）上，这样大表就直接在本地和小表进行 JOIN 计算，生成结果，因而，通过 MAPJOIN 方式，大表数据不需要进行网络传输。顾名思义，MAPJOIN 是没有 Reduce 操作的。

批注 [a6]: 周一找 garr 了解各 join 实现原理细节

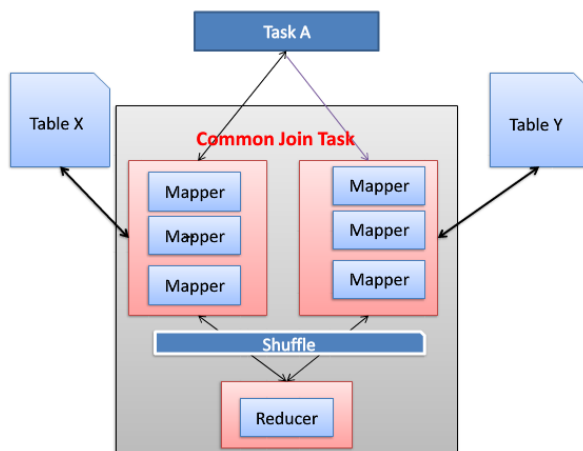


图 1-1 普通 JOIN 原理图

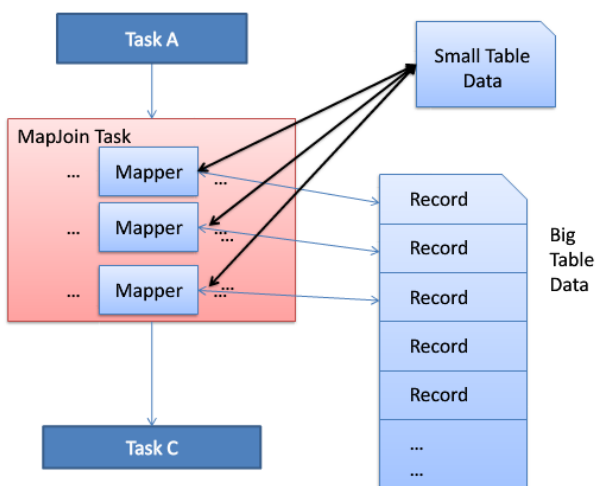


图 1-2 MAPJOIN 原理图

批注 [a7]: 基于自己的理解来画?

因此，MAPJOIN 经常用于优化，希望大家能够理解并记住它，下面我们看看如何使用它。假定前面给出的 page_view 表是个大表，user 表是个小表，通过 MAPJOIN 对它们进行连接，SQL 语句如下：

```
SELECT /*+ MAPJOIN(u) */
    u.user_id, u.gender, u.age,
    pv.view_time, pv.page_url, pv.referrer_url, pv.ip
FROM user u JOIN page_view pv
```

```
ON u.user_id = pv.user_id;
```

注意，这里 `/*+ MAPJOIN(u)*/` 是必须的，它标识该 JOIN 是 MAPJOIN，而且小表是 `u`（user 表）。如果去掉它，就变成普通的 JOIN 操作了。

需要注意的是：

- LEFT OUTER JOIN，应该把大表放在左边
- RIGHT OUTER JOIN，应该把大表放在右边
- (INNER) JOIN 和 FULL OUTER JOIN，则和大表位置无关

此外，MAPJOIN 对小表大小目前限制为 512M（解压后在内存中的大小）。前面我们已经指出，ODPS SQL 只支持等值连接，它是针对普通连接而言；对于 MAPJOIN，还支持不等值表达式，OR 逻辑等复杂的 JOIN 条件。

批注 [a8]: 多张小表 example?

批注 [a9]: example?

5.5.5 聚合操作

假设有这样一个需求，需要查看访问页面的男女用户分别是多少。显然，要按性别进行聚合。SQL 如下：

```
SELECT pv_users.gender, count (DISTINCT pv_users.user_id)
FROM pv_users
GROUP BY pv_users.gender;
```

对于前面构造的两条数据，查询输出结果如下：

gender	_c1
1	1
2	1

这里，由于没有对聚合计算结果 `count (DISTINCT pv_users.user_id)` 指定别名，所以系统会指定如 `“_c1”` 这样的名字，使用时需要加上反引号才能正确引用，如 `SELECT ` _c1 ` FROM t1`；这样使用上相对麻烦，另外过段时间看，可能完全不知道这一列的含义了。因此，强烈建议对这种列通过 AS 指定别名。上面的 SQL 可以改写如下：

```
SELECT pv_users.gender, count (DISTINCT pv_users.user_id) AS cnt
FROM pv_users
GROUP BY pv_users.gender;
```

ODPS SQL 支持 “Multi-Distinct”，即支持 DISTINCT 操作作用于不同的列，比如下面这个 SQL，DISTINCT 作用的列分别是 `user_id` 和 `ip`，也是可以正常工作的：

```
SELECT pv_users.gender,
```

```
count(DISTINCT pv_users.user_id) AS user_count,  
count(DISTINCT pv_users.ip) AS ip_count  
FROM pv_users  
GROUP BY pv_users.gender;
```

常见的聚合函数包括 COUNT、SUM、MAX、MIN 等，聚合函数必须使用关键字 GROUP BY。需要注意的是，SELECT 中的各个列（除了聚合计算），都必须包含在 GROUP BY 中，比如上面的 SQL 中，由于 count(DISTINCT pv_users.user_id) 是聚合计算，所以可以不在 GROUP BY 列中，而下面这个 SQL 就是非法的：

```
SELECT pv_users.gender, count (DISTINCT pv_users.user_id) as count, pv_users.ip as count  
FROM pv_users  
GROUP BY pv_users.gender;
```

执行该 SQL 语句，会抛出如下异常信息：

```
FAILED: ODPS-0130071:Semantic analysis exception - Expression not in GROUP BY key : line  
1:68 'ip'
```

下面举个简单的例子，就不难明白其中原因了。

假设只查询 gender 和 ip，按 gender 进行 Group By，实际上，SQL 语句转换成 MapReduce 作业时，reducer 的输入就是按 Group By 后的字段（这里是 gender）进行排序分组的，假设某个用户在一台机器上访问网站，可能数据如下：

```
1 192.168.1.122  
1 192.168.3.86  
1 112.172.1.244
```

由于 Group By 的 key 是 gender（值为 1），相同 key 只能输出一条结果，那么 gender 为 1 的结果应该输出哪个 ip 呢？在这个例子中，如果希望输出多条结果，则应该把 gender 和 ip 一起作为 Group By 的 key，如下

```
GROUP BY pv_users.gender, pv_users.ip
```

这样，由于 Group By 的 Key 是（gender, ip）组合，以上三条结果就是唯一的，都会输出。

ODPS SQL 除了支持一些聚合函数外，还支持一些数据函数，如求绝对值（abs）、四舍五入（round）、生成随机数（rand）等，以及一些字符串处理函数、日期函数等。特别地，ODPS SQL 还支持窗口函数，由于 Hive 中没有，下面我们特别介绍一下 ODPS SQL 的窗口函数。

5.5.6 窗口函数

传统数据库如 Oracle 等提供了窗口函数（在 Oracle 中称分析函数），为了支持复杂的数据分析需求，ODPS SQL 也提供了语法非常类似的窗口函数功能，如常见的 Row_number、Rank 等，窗口函数只能用在 SELECT 子句中。窗口函数相当于给查询结果增加一个列，比如假设有这个场景，对于之前创建的 user 表，希望输出年龄最小的 10 个用户的年龄和性别信息，并且给出排序序号，如下：

user_id	age	gender	rk
123	15	1	1
777	16	2	2

即多出一个 rk 字段，表示年龄排序，这可以通过窗口函数 rank()来实现，执行 SQL 如下：

```
SELECT *
FROM
(SELECT user_id, age, gender,
      rank() over (partition by 1 order by age) as rk
From user
) t
WHERE t.rk <=10;
```

这里，简单说明一下该 SQL 语句：

- 1) partition by 1 是指 Partition 的 key 都设置为 1，这样所有记录就会输出到一个 Reducer 进行归并排序
- 2) 嵌套查询是因为 WHERE 条件会比 SELECT 中的各个字段计算先执行，所有在 WHERE 中不能引用 SELECT 中的字段计算结果 rk，比如下面的 SQL 是错误的：

```
SELECT user_id, age, gender,
      rank() over (partition by 1 order by age) as rk
From user
WHERE rk <=10;
```

5.5.7 多路输出（Multi Insert）

假设有这样一个场景，需要扫描 pv_users 表，分别按性别和年龄查看页面访问情况，把结果相应输出到两张结果表中。应该怎么做呢？最简单最直接的方式就是执行两次 SQL 查询，分别输出对应的结果表中。如果对同一张表的查询分析需要输出到 50 张结果表呢？如果执行 50 个 SQL，意味着需要扫描 50 次 pv_users 表。因为是同一张表，是否可以通过某种方式，只扫描一次原始数据表，输出到不同的结果表中呢？对于 ODPS SQL，答案是肯定的，其多路输出功能就是为此“而生”的。

比如对于该场景，执行 SQL 如下：

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_sum
  SELECT pv_users.gender, count(DISTINCT pv_users.user_id)
  GROUP BY pv_users.gender
```

```
INSERT INTO TABLE pv_age_sum
SELECT pv_users.age, count(DISTINCT pv_users.user_id)
GROUP BY pv_users.age;
```

在该 SQL 语句中，从 pv_users 表每读取一条记录，就会对 SELECT 语句的条件进行判断。不同 SELECT 之间是独立运行的，而不是 IF...THEN...ELSE...这种模式。

此外，多路输出不仅支持输出到多张表，还支持输出到同一张表的不同 Partition，或者混合方式。另外，在一个 SQL 中，还同时支持 Insert Into 和 Insert Overwrite。

5.5.8 动态分区 (Dynamic Partition)

在“分区表查询”一节中，曾提到这样的场景：“假设要查询 2011-12-17 这一天 referrer_url 来自 taobao.com 这个域的 PV 情况”，并且提到如果要把查询结果保存到带 Partition 的表 from_taobao_with_partition 中，如果 Partition 数很多或者预先不知道的话，应该如何处理。这里先给出动态分区的解决方案，执行如下 SQL：

```
INSERT OVERWRITE TABLE from_taobao_with_partition
PARTITION(dt='2011-12-17',country)
SELECT pv.user_id, pv.view_time, pv.page_url, pv.referrer_url, pv.ip, pv.country
FROM page_view pv
WHERE pv.dt='2011-12-17' AND
pv.referrer_url LIKE '%taobao.com/';
```

注意，PARTITION 中只给出 dt='2011-12-17'，而没有给出 country 的值，具体的 country 分区是通过判断每条满足条件的记录应该输出到哪个分区而确定的。这里，我们称 country 是“动态分区键”，相反，分区键 dt 值已经给出，称为“静态分区键”。动态分区键必须通过 SELECT 的最后一个（或几个）列指定，而静态分区键则不需要在 SELECT 中给出。

此外，对于动态分区，还需要注意以下几点：

- 1) 可以指定多个动态分区键，比如 PARTITION(dt,country)；静态分区键（如果有）必须在动态分区键的前面，比如 PARTITION(dt,country='US')是不支持的。
- 2) 动态分区键必须在 SELECT 选定列的最后一个（或几个，个数和动态分区键数一致），它是按选定列的序号来确定的，而不是按列名称，比如这里动态分区键 country 是由 SELECT 的最后一列 pv.country 确定的，如果 pv.country AS country_name，也是可以的；而如果 SELECT 写成 SELECT pv.country, pv.user_id, pv.view_time, pv.page_url, pv.referrer_url, pv.ip 则是不可以的。
- 3) 动态分区可能会在瞬间生成非常多的分区（这往往也属于误用 SQL），这会造成资源消耗非常大，所以 ODPS SQL 对动态分区数设置了上限，目前单个节点上限是 1024，所有节点上限值是 2048。
- 4) 由于动态分区键的值是根据满足条件的记录来确定的，比如在这个例子中，page_view 包含分区 dt=2011-12-17/country=Japan，但是该分区下没有满足 referrer_url LIKE '%taobao.com/' 条件的记录，那结果表中就不会创建该分区。同样道理，如果结果表中已

有该分区，则数据不会被覆盖（这和 Insert Overwrite 相关）；如果结果表中已有某个分区，而查询也有输出到该分区的记录，那么该分区会被覆盖。

5) 动态分区键的值不能是 NULL，否则会报异常。

动态分区实质上和多路输出很相似，可以把动态分区理解成一种特殊形态的多路输出。如果不采用动态分区方式，需要写多个 Insert 语句，每个 Insert 语句可能对应一个 MapReduce 任务，这在性能上会差很多。此外，使用动态分区，程序的扩展性会更好，如果没有采用动态分区，每次在源表添加新的分区，就得更改程序，添加新的 Insert，而使用动态分区，则没有这些问题。

批注 [a10]: 这个理解对么

5.5.9 子查询

在前面给出的查询中，大部分是 SELECT...FROM table;的形式，即 From 的对象是一张已存在的表。如果 From 的对象是另一个 SELECT 查询语句，比如在“导入数据”一节中为了造数据提到的：

```
SELECT 123, 1386213971, "http://www.tmall.com", "http://www.taobao.com/", "192.91.189.6"
FROM (SELECT count(*) FROM user)a;
```

我们称这类查询为子查询。这里要注意的是，子查询的 SELECT 子句（如 SELECT count(*) FROM user）需要有别名（比如这里别名为 a）。实质上，可以把子查询的结果当作一张表，和其他表或子查询进行 Join 操作，比如下面这个 SQL：

```
SELECT a.user_id, b.cnt
FROM (SELECT user_id FROM user) a
JOIN
    (SELECT pv.user_id, count(*) as cnt FROM page_view pv group by pv.user_id) b
ON a.user_id = b.user_id;
```

在传统 SQL 中，子查询往往可以通过 in 子句实现，ODPS SQL 也支持 where 条件中包含子查询，比如下面这个 SQL：

```
SELECT pv.*
FROM page_view pv
WHERE pv.user_id IN
(SELECT user.user_id from user);
```

注意，这里 WHERE 子查询不能用别名，比如(SELECT user.user_id from user) u 会报错。也就是说，FROM 子查询必须别名，WHERE 子查询不能别名。

5.5.10 UNION ALL

Union All 可以把多个 SELECT 操作返回的结果联合成一个数据集，它会返回所有的结果，不会执行去重。

ODPS SQL 不支持直接对顶级的两个查询结果执行 UNION 操作，需要写成子查询的形式，如以下 SQL 会报错：

```
SELECT pv.user_id as uid
FROM page_view pv
UNION ALL
SELECT u.user_id as uid
FROM user u;
```

执行报错如下：

FAILED: ODPS-0130071:Semantic analysis exception - line 2:5 Top level UNION is not supported currently; use a subquery for the UNION. Error encountered near token 'uid'

改成子查询形式如下：

```
SELECT * FROM(
    SELECT pv.user_id as uid
    FROM page_view pv
    UNION ALL
    SELECT u.user_id as uid
    FROM user u) a;
```

另外需要注意的是，UNION ALL 连接的两个 SELECT 查询语句，两个 SELECT 的列个数、列名称、列类型必须严格一致，如果原始名称不一致，可以通过别名设置成相同名称。由于这里是 FROM 子查询，所以必须设置别名（如 a）。

5.5.11 排序

ODPS SQL 提供的排序功能相关的关键字主要有四个，这里先简要说明一下，再给出一些示例。

- 1) Order By: 全局排序，和数据库的 Order By 功能一致，最后所有数据都必须在一个 Reducer 执行，当数据量很大时，会导致出现执行过慢，无法输出结果的情况。因此，在 ODPS SQL 中，Order By 必须和 Limit N 一起用，这样 Reducer 只需要对 N*mapper 数 条记录进行归并排序。
- 2) Sort By: 局部排序，只对同一路 Reducer 中的数据进行排序，不同 Reducer 之间的数据无序
- 3) Distribute By: 如何划分数据，按指定的字段把数据划分到不同的 Reducer 中
- 4) Cluster By: Distribute By + Sort By

Distribute By 实质上是指定 Mapper 的 key，和排序无关，因为它经常和 Sort By 一起用（不了解的往往也容易把它和排序方式混淆起来），所以这里一起介绍。

下面是个简单的 ORDER BY 例子：

```
SELECT pv.*
FROM page_view pv
ORDER BY pv.user_id desc
Limit 10;
```

另一个示例：

```
SELECT pv.*
FROM page_view pv
DISTRIBUTE BY pv.user_id
SORT BY pv.user_id, pv.view_time DESC;
```

一般来说，从业务角度，**SORT BY** 的字段往往会包含 **DISTRIBUTE BY** 字段，这样可以保证相同 **key** 可以归并在一起。举个例子，假设某路 **Reducer** 输出中包含以下几条记录，因为 **DISTRIBUTE BY** 只保证相同 **key**（这里是 **user_id**）会输出到同一路 **Reducer** 中，但并不保证相同 **Key** 会聚合在一起，如下，

表 1-1 Distribute By 结果

user_id	view_time
1	123
2	124
1	125
2	125

如果只执行 **SORT BY pv.view_time** 的话，按 **view_time** 进行排序后，结果还是表 1-1 那样。所以如果希望把相同 **user_id** 归并在一起再按 **view_time** 进行排序，输出如下：

表 1-2 期望的 Sort By 结果

user_id	view_time
1	123
1	125
2	124
2	125

则需要执行 **SORT BY pv.user_id, pv.view_time**。

Cluster By 不支持指定排序方式，只支持倒序形式，由于以上 SQL 指定 **pv.view_time DESC** 即倒序方式，所以可以用 **Cluster By** 改写如下：

```
SELECT pv.*
FROM page_view pv
CLUSTER BY pv.user_id, pv.view_time;
```

值得一提的是，不是所有的 `DISTRIBUTE BY+ SORT BY` 都可以用 `CLUSTER BY` 改写，比如这里如果 `SORT BY` 没有指定降序，`SORT BY pv.user_id, pv.view_time`，则不可改写。

5.5.12 CASE WHEN 表达式

在 `SELECT` 查询时，也可以通过 `CASE ...WHEN` 表达式，根据表达式结果灵活返回不同的值。比如我们之前在 `user` 表中，定义 1 表示男性，2 表示女性，0 未知，所以在 `user` 表中，`gender` 列的值都是 0,1,2 之类，如果希望查询结果该列显示 `male`、`female` 之类，便于理解，可以通过 `CASE WHEN` 来实现，SQL 如下：

```
SELECT u.user_id,
       CASE
         WHEN u.gender=1 THEN 'male'
         WHEN u.gender=2 THEN 'female'
         ELSE "unknown"
       END
AS gender
FROM user u;
```

`CASE WHEN` 其实相当于 `SELECT` 的一个列，`AS gender` 表示别名。

5.6 UDF

ODPS SQL 还支持用户自定义函数（UDF）来扩展系统内置函数，实现特定功能。

具体来说，UDF 包含三种类型：

- **UDF（User Defined Function）**：用户自定义的 `Scalar Function`，可以在 `SQL` 表达式中使用，并且只返回一个值。
- **UDAF（User Defined Aggregation Function）**：用户自定义聚合函数，将多条记录聚合成一条结果。
- **UDTF（User Defined Table Function）**：用户自定义表函数，对多条记录进行转换后再输出，输出结果个数和输入记录个数不需要一一对应。**UDTF** 是唯一可以返回多条结果的自定义函数。

从广义上说，UDF 是以上三种的统称。

UDF 目前支持 `Java` 和 `Python` 两种语言接口，用户编写的 UDF 程序可以以资源的方式上传到 `Project` 中，通过 `Create Function` 创建函数，然后就可以在 `SQL` 中调用该函数了。

下面，我们一起动手实现一个 UDF 吧。

5.6.1 UDF

在前面给出的 `page_view` 表中，我们定义了其 `ip` 字段是 `String` 类型如 “192.91.189.6”，假设有另一张表的 `ip` 是数字形式，需要关联查询，期望把 `page_view` 表的 `ip` `String` 类型转换成数字形式便于后续关联，我们就可以实现一个 `ip2num` 的自定义函数，来完成这个功能。下面我们按照实际步骤分别说明。

5.6.1.1 Python 版 UDF

1. 代码实现

首先，Python 版的 `udf` 实现 `ip2num.py` 的完整程序清单如下：

```
#!/usr/bin/env python
#coding:utf-8
import sys
from odps.udf import annotate

@annotate("string->bigint")
class ip2num(object):
    def evaluate(self, ipString):
        if ipString is None:
            raise Exception("Invalid IP")
        try:
            octets = [octet.strip() for octet in ipString.split('.')]
        except Exception,e:
            raise e
        num = (int(octets[0])<<24) + (int(octets[1])<<16) + (int(octets[2])<<8) + int(octets[3])
        return num
def main():
    test = ip2num()
    res = test.evaluate("192.168.7.2")
    print res
if __name__ == "__main__":
    main()
```

代码比较简单，这里有几点需要注意：

- 1) 必须通过 `@annotate` 指定函数签名²⁹，比如这里 `@annotate("string->bigint")`，表示接收参数是 `string` 类型，函数返回类型是 `bigint`。
- 2) 定义一个类，并在类中实现 `evaluate` 方法。

²⁹ 函数签名是指 接收输入参数类型，以及函数返回类型

3) udf 不需要 main 函数，这里定义 main()是为了简单测试该 udf 的正确性。

2. 代码测试

要测试代码，最简单的是实现main，在main中调用实例化一个类对象，调用evaluate函数，给出测试数据，看结果是否一致。

如果直接运行以上代码，python ip2num.py，会报错：SyntaxError: invalid syntax，这是因为@annotate("string->bigint")是odps udf库定义的，而不是Python自带的。一个简单的测试方式是注释掉以下两行代码：

```
#from odps.udf import annotate
#@annotate("string->bigint")
```

然后运行 python ip2num.py 程序即可输出结果。

此外，ODPS UDF 手册³⁰还给出了更完善但稍微复杂一些的本地调试方式。这里不再赘述。

3. 上传

要使用该 UDF，首先需要把代码作为资源上传到 Project 中，在 clt 中执行如下：

```
odps@ odps_book>add py /home/admin/book/sql/udf_ip2num.py -f;
```

```
OK: Resource 'udf_ip2num.py' have been updated.
```

-f 选项表示如果 Project 中已上传过 udf_ip2num.py，则覆盖它。

4. 创建函数

执行如下命令：

```
odps@ odps_book>create function ip2num as 'udf_ip2num.ip2num' using 'udf_ip2num.py';
```

```
Success: Function 'ip2num' have been created.
```

这里，create function ... as ... using 是创建 udf 函数的关键字，第一个 ip2num 表示 udf 的函数名称，SQL 在引用该函数时即使用该名称；'udf_ip2num.ip2num'表示 Python 脚本名.类名，必须用引号引起来。'udf_ip2num.py'表示资源名称，即上一步上传的资源，同样必须用引号引起来。

如果函数已存在，则会报错，可以执行 drop function ip2num；先删除它后再创建。

5. 在 SQL 中使用该函数

比如，执行个简单的 SQL 如下：

```
SELECT user_id, ip2num(ip) AS ip from page_view;
```

以上我们一起完成了创建 UDF 到使用 UDF 的整个过程，是不是很简单？也许，你要说“我

³⁰ http://odps.alibaba-inc.com/doc/prddoc/odps_udf/odps_udf.html#udf-local-debug

习惯用 Java 开发,可以给个 Java 版本示例吗?”好的,那下面我们也给个 Java 版示例说明。

5.6.1.2 Java 版 UDF

现在,我们一起来看看如何通过 Java UDF 实现以上步骤。

1. 代码实现

我们在 Eclipse 下实现该代码。首先创建 Project, 因为 Java UDF 需要继承 `com.aliyun.odps.udf.UDF` 类, 该类在 `mapreduce-api.jar` 包³¹。所以要先配置 path, 步骤是右击 Project 名称, 选择“Build Path -> Configure Build Path”, 在 Libraries 标签栏中, 选择“Add External JARs...”, 选择 `mapreduce-api.jar`, 然后点击 ok, 就配置完成。

Java 代码实现如下:

```
package example;
import com.aliyun.odps.udf.UDF;

public final class IP2num extends UDF{
    public Long evaluate(String ip) {
        long result = 0;
        String[] ipArray = ip.split("\\.");
        for(int i=3; i>=0; i--) {
            long n = Long.parseLong(ipArray[3-i]);
            result |= n << (i*8);
        }
        return result;
    }
}
```

这里, `evaluate` 方法必须是非 `static` 的 `public` 方法。`evaluate` 方法的参数类型和返回值类型作为该 UDF 的函数签名。另外需要注意的是, `evaluate` 返回的是对象, 而不是基本数据类型, 这里必须是 `LONG`, 而不是 `long`, 否则会报 “but no valid method” 的错误。

对于 Java UDF, 需要打包生成 jar 文件, 我们通过 Eclipse 的 `Export->JAR File`, 生成 `udf_ip2num.jar`。

2. 代码测试

首先, 为以上 `IP2num` 类实现一个简单的测试代码, 如下:

```
package example;
import example.IP2num;

public class Test {
    static IP2num t = new IP2num();
}
```

³¹ 从 <http://odps.alibaba-inc.com/download/> 下载 `odps_clt_release_64.tar.gz`, 解压目录下有

```
public static void main(String[] args) {  
    String s = "192.168.7.2";  
    System.out.println(t.evaluate(s));  
}  
}
```

然后运行它，查看结果。当然，ODPS UDF 提供了更好的调试方式。

3. 上传

在 clt 中执行命令如下：

```
odps@ odps_book>add jar /home/admin/book/sql/udf_ip2num.jar -f;  
OK: Resource 'udf_ip2num.jar' have been updated.
```

4. 创建函数

执行命令如下：

```
odps@ odps_book>drop function ip2num;  
Confirm to "drop function ip2num;" (yes/no)? yes  
odps@ odps_book>create function ip2num as 'example.IP2num' using 'udf_ip2num.jar';  
Success: Function 'ip2num' have been created.
```

这里，由于之前已经创建了 ip2num 函数，所以先 Drop 掉。'example.IP2num'是 Java 类的全名，'udf_ip2num.jar'是上传的 JAR 包资源文件，都必须用引号引起来。

5. 在 SQL 中使用函数

和之前一样，执行如下 SQL：

```
SELECT user_id, ip2num(ip) AS ip from page_view;
```

输出结果应该和之前用 Python 版 UDF 的一样。

5.6.2 UDAF

比如现在有这个场景，要查询用户都在哪台机器(ip)访问，并且每个用户只输出一条结果，把 ip 连接起来，相同 ip 进行去重，如下：

```
user_id  ip1, ip2,  ip3,  ip4
```

由于 ODPS 自带的 wm_concat 函数没有去重功能，我们自己写个函数实现它。

显然，这属于 UDAF 场景：通过 Group By 按 user_id 进行分组，对同一个分组（即 user_id 相同）下的多条记录，进行聚合，输出一条结果。

代码实现如下：

```
#!/usr/bin/env python
#coding:utf-8

from odps.udf import annotate
from odps.udf import BaseUDAF

@annotate('string, string->string')
class WmDistinctConcat(BaseUDAF):

    def new_buffer(self):
        self.SEPERATOR = ""
        return {}

    def iterate(self, adict, seperator, element) :
        if not adict.has_key(element):
            adict[element] = seperator

    def merge(self, adict, pdict):
        for key in pdict.keys():
            if not adict.has_key(key):
                adict[key] = pdict[key]

    def terminate(self, adict):
        if len(adict) == 0:
            return
        sep = ""
        res = list()
        for key in adict.keys():
            sep = adict[key]
            res.append(str(key))

        return sep.join(res)
```

然后，在 console 中执行以下命令和 SQL，得到期望的输出结果。

```
add py /home/admin/book/sql/udaf_wm_distinct_concat.py -f;
```

```
create function wm_distinct_concat as 'udaf_wm_distinct_concat.WmDistinctConcat' using
'udaf_wm_distinct_concat.py';
```

```
SELECT user_id, wm_distinct_concat(',', ip) as ip_list
FROM page_view
GROUP BY user_id;
```

5.6.3 UDTF

前面，我们对 UDF 和 UDAF 进行了比较详细的叙述，以帮助用户入门。UDTF 的开发运行类似，这里不再多写了。另外，ODPS UDF 手册给出了很详尽的说明。

5.7 调优

对于 ODPS SQL 计算而言，由于它是在分布式计算框架上计算的，就是面向海量数据处理，所以数据量大不一定会造成性能问题，但数据倾斜则会导致作业运行很慢。这里，我尝试对性能调优给出一些见解。如果你有更好的经验分享，非常欢迎交流。

5.7.1 查看执行计划 Explain

要对 ODPS SQL 进行调优，首先有必要了解一下 ODPS SQL 的执行过程。可以通过 Explain 关键字查看 ODPS SQL 的执行计划。比如在 console 中输入：

```
EXPLAIN select count(*) as cnt from page_view;
```

则可以查看 SQL 语句 `select count(*) as cnt from page_view;` 生成作业的执行计划。ODPS SQL 手册给出了非常详尽的执行计划说明³²，很值得看看。

5.7.2 常见的优化考虑和注意

这里，我非常简略地给出一些经验总结，可能显得有些空洞，但或许对你面临的调优问题也能有些帮助，以后争取给出一些更详尽的示例说明。

1. 数据模型

从业务角度考虑，好的数据模型是成功的一半！

2. 对大表分 Partition

这样后面查询条件很可能只需要遍历单个 partition，而不是整张表

3. 大小表连接

尽量用 MAPJOIN

³² http://odps.alibaba-inc.com/doc/prddoc/odps_sql/odps_sql_grmr.html#id10

4. 作业数太多问题

由于 ODPS SQL 计算很多是生成 MapReduce 作业（一般来说，直接在本地运行的作业也不会造成性能问题），而 MapReduce 作业初始化需要一定的时间，所以如果作业数太大，比如很多 JOIN 和聚合操作，可能也会耗时很长。解决方案是从业务角度减少作业数。

5. 数据倾斜问题

- 了解数据情况，从业务角度避免，比如对于 Count(distinct)操作，因为 Count(distinct)是先按照 Group By 字段进行分组，再按照 Distinct 字段进行排序，所以很容易造成数据倾斜，要慎用
- 设置 `odps.sql.groupby.skewindata=true`

6. 参数配置

几个非常常见（且有效）的性能调优参数配置说明如下表所示。

参数名称	作用	默认值
<code>odps.sql.groupby.skewindata</code>	遇到数据倾斜时，可以设置该属性为 true，它只对 Group By 造成的倾斜有用，默认为 false	false
<code>odps.sql.mapper.split.size</code>	设置一个 mapper 的最大数据输入量，可以通过该变量调整 mapper 数	256（MB）
<code>odps.sql.reducer.instances</code>	设置 Reducer 的数量，当中间结果数据量比较大时，可以调整它（增大）	-1（表示 mapper 数的 1/4）
<code>odps.sql.joiner.instances</code>	设置 Join Task 的 instance 数	-1（表示 join 上游的两个 task 数相加），上限 2000

7. 整体调优

在性能调优中，还应该从整体考虑，单个作业最优不如整体最优。

5.8 FAQ

1. explain 输出不全？

输出结果最大 4KB，超出会被截断。暂时无解。

2. 报错：FAILED: ODPS-0010000:System internal error - compiling failed

SQL 大小不能超出 2MB，如果超出，编译时内存占用太大会被 kill。可以对 SQL 进行分解。

3. odps sql 创建的表默认的字段分隔符是什么？

ODPS 表没有分隔符，底层基于列存储。

4. ODPS SQL 如何对数据抽样?

5. 报错: ODPS-0121145:Data overflow - Div func result is nan, two params are -71 and nan
这可能是分母为 0 或空导致

待补充。。。

第6章 ODPS 安全

第7章 ODPS 实战

在本章，我们将通过一个真实数据示例教程，了解如何通过 ODPS 提供的各种服务来满足各种需求。在前面的各个章节中，我们已经详细介绍了 ODPS 提供的主要功能以及如何使用，而在该实战教程中，我们将一起探讨如何通过 ODPS 服务分析处理真实数据，有兴趣的话，你还是可以参照本示例教程自己动手演练一遍。

7.1 场景和数据说明

本节所展示的示例教程是基于一份真实的数据集，数据来源是世界银行网站上提供的关于“城市发展”这一专题的数据集³³。它涵盖全世界各个国家和地区、从 1960 年 2012 年关于城市发展的各个指标的数据。对于每个国家或区域，都有 21 个相同的指标，如下³⁴：

INDICATOR_CODE	INDICATOR_NAME
EN. ATM. PM10. MC. M3	PM10，国家级（每立方米微克）
EN. POP. DNST	人口密度（每公里土地面积人数）
EN. URB. LCTY	最大城市中的人口
EN. URB. LCTY. UR. ZS	最大城市中的人口（占城市人口的百分比）
EN. URB. MCTY	人口超过 100 万的城市群中的人口。
EN. URB. MCTY. TL. ZS	人口超过 100 万的城市群中的人口（占总人口的百分比）
EP. PMP. DESL. CD	柴油的市场价格（美元/每升）
EP. PMP. SGAS. CD	汽油的市场价格（美元/每升）
IS. ROD. DESL. PC	道路部门人均燃料消耗量（千吨石油当量）
IS. ROD. ENGY. ZS	道路部门能源消耗量（占总能源消耗量的百分比）
IS. ROD. SGAS. PC	道路部门人均汽油消耗量（千吨石油当量）
IS. VEH. NVEH. P3	机动车（每千人）
IS. VEH. PCAR. P3	客车（每千人）
IS. VEH. ROAD. K1	车辆数量（每千米道路）
SH. H2O. SAFE. UR. ZS	城市改善的水源（获得改善水源的城市人口所占百分比）
SH. STA. ACSN. UR	城市改善的卫生设施（获得经改善卫生设施的城市人口所占百分比）
SI. POV. URGP	贫困差距，按城市贫困线衡量的（百分比）
SI. POV. URHC	贫困人口比例，按城市贫困线衡量的（占人口的百分比）
SP. URB. GROW	城镇人口增长率（年增长率）

³³ 注：可以登录 <http://data.worldbank.org/topic/urban-development> 查看更多关于该专题的说明。数据下载 url：http://api.worldbank.org/datafiles/16_Topic_MetaData_en_EXCEL.xls

³⁴ 注：这些指标的含义说明在下载 excel 文件的 sheet2。

SP.URB.TOTL	城镇人口
SP.URB.TOTL.IN.ZS	城镇人口（占总人口比例）

我们下载的是 excel 格式数据，在 Excel 中打开，通过“另存为”菜单栏命令保存为以 tab 分隔的纯文本文件³⁵，生成文件命名为 metadata.txt。

在生成的数据文件中，每一行包含 57 个字段，分别是 Country Name、Country Code、Indicator Name、Indicator Code 以及从 1960 年到 2012 年的所有年份（共 43 个年份），数据示例如下：

（第 1 行：字段名）				Country Name	Country Code		Indicator Name		Indicator Code		1960	1961	1962	1963
1964	1965	1966	1967	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	
1978	1979	1980	1981	1982	1983	1984	1985	1986	1987	1988	1989	1990	1991	
1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	
2006	2007	2008	2009	2010	2011	2012								
（第 2 行：字段值）				Arab World	ARB	"PM10, country level (micrograms per cubic meter)"				EN.ATM.PM10.MC.M3				
149.8373646	152.1199551			154.6523814	152.815989		145.5228696		138.5398091		133.4297122			
137.0123045	129.4388408			120.8656106	111.612253		110.6137047		110.8483421		103.6972135			
105.6099798	94.90057264			87.3701309	82.73166359		77.26622055		76.42026712		69.67364899			

其中 Country Name 是国家的全称³⁶，Country Code 唯一标识某个国家；Indicator Name 表示指标信息，Indicator Code 唯一标识某个指标；1960 到 2012 表示某个国家某个指标在对应年份的值，这个值有很多为空。

7.2 通过 Tunnel 上传数据

首先，我们需要把这份数据导入到 ODPS 中，才能执行后续的处理分析操作。

这里采用 **DataX+Tunnel** 的方式导入数据。Tunnel 是 ODPS 提供的支持多种数据交换场景的通用服务，它提供了 Java SDK。DataX 是离线数据同步工具，实现各种异构数据源之间的数据同步功能，它把不同数据源抽象成相应的 Reader、Writer 插件。因此，可以通过其 StreamReader 和 ODPSWriter 插件，把本地数据通过 Tunnel 服务导入到 ODPS 中。

批注 [a11]: 改成用 Tunnel SDK 方式

7.2.1 创建表

创建原始数据表 urban_data，sql 语句如下：

```
create table if not exists urban_data(
    country_name string,
    country_code string,
    indicator_name string,
    indicator_code string,
    y1960 double,
    y1961 double,
    y1962 double,
```

³⁵ 注：这种在 Excel 中的“另存为”方式存在丢失精度可能。由于该示例教程重点在于说明如何使用 ODPS，所以采用这种简单蛮力的方式来生成原始数据文件。在真实数据分析中，丢失精度可能是无法接受的。

³⁶ 注：原始数据除了国家的数据统计外，还包含区域范围（按某个特征）的数据统计，比如“Country Name”值可能为“East Asia & Pacific (all income levels)”。这里为了简单，没有做特别处理。

y1963 double,
y1964 double,
y1965 double,
y1966 double,
y1967 double,
y1968 double,
y1969 double,
y1970 double,
y1971 double,
y1972 double,
y1973 double,
y1974 double,
y1975 double,
y1976 double,
y1977 double,
y1978 double,
y1979 double,
y1980 double,
y1981 double,
y1982 double,
y1983 double,
y1984 double,
y1985 double,
y1986 double,
y1987 double,
y1988 double,
y1989 double,
y1990 double,
y1991 double,
y1992 double,
y1993 double,
y1994 double,
y1995 double,
y1996 double,
y1997 double,
y1998 double,
y1999 double,
y2000 double,

```
y2001 double,  
y2002 double,  
y2003 double,  
y2004 double,  
y2005 double,  
y2006 double,  
y2007 double,  
y2008 double,  
y2009 double,  
y2010 double,  
y2011 double,  
y2012 double);
```

把该 sql 保存到 create.sql 文件中，通过以下命令执行创建表：

```
clt/bin/odpscmd -f create.sql
```

7.2.2 导入数据

我们把本地数据以 stream 方式，通过 datax 导入到 odps。

首先，先下载 datax，

批注 [a12]: 改掉，改成 Tunnel SDK

生成 datax 配置文件³⁷，如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<jobs>  
  <job id="tutorial_import ">  
    <reader>  
      <plugin>streamreader</plugin>  
      <param key="field-delimiter" value="\t"/>  
      <param key="concurrency" value="1"/>  
    </reader>  
    <writer>  
      <plugin>odpswriter</plugin>  
      <param key="project" value="odps_book"/>  
      <param key="table" value="urban_data"/>  
      <param key="access-id" value="***/>  
      <param key="access-key" value="***/>
```

³⁷ 注：如何配置请参考 <http://gitlab.taobao.ali.com/atcloud/datax/wikis/datax-plugin-streamreader>

```
<param key="concurrency" value="1"/>

<param key="truncate" value="true"/>

<param key="error-limit" value="0"/>

</writer>

</job>

</jobs>
```

然后执行以下命令：

```
cat data/metadata.txt | python datax/bin/datax.py streamreader_odpswriter.xml
```

注意：在执行命令之前，我们删除了 metadata.txt 中第 1 行字段名称。

批注 [a13]: 补充：通过在云端，使用 DTTask 来完成

7.3 数据分析需求

数据导入到 ODPS 表中后，可以通过 ODPS 提供的 MapReduce（MR）或 SQL 等进行分析处理。假设我们要完成如下分析：

1. 把数据按年份分区存储
2. 统计 1960 年到 2012 年间每年 PM10 值最大的国家
3. 统计每个国家在各个年份的所有指标报表
4. 比较 2010 年低收入、中等收入和高收入国家在人口密度、城镇人口增长率和城镇人口（占总人口比例）三个指标的情况，输出每个指标值最大的国家类型
5. 对中、美、日、法、德、英这六个国家每年的能源消耗情况进行排序

7.4 通过 MapReduce 按年份分区

在大数据处理中，可以把大表按分区来存储，这样可以利用局部性原理，在查找分析某个分区的数据时，只需要扫描单个分区，而不需要遍历整张大表。这里，由于年份是固定的（1960 年到 2012 年），我们可以实现一个 Map 程序，支持多路输出，每个分区相当于一路输出，把数据写到 urban 表的各个分区。需要注意的是，MapReduce 的输出路数上限是 512。

7.4.1 创建表

要创建 urban 表，sql 语句如下：

```
create table if not exists urban(country_name string, country_code string, indicator_name string, indicator_code string, value double) partitioned by (year string);
```

7.4.2 代码实现

实现 MRTask，实现把原表按年份分区³⁸，代码如下：

```
package urban;

import java.io.IOException;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import com.aliyun.odps.Record;
import com.aliyun.odps.io.LongWritable;
import com.aliyun.odps.io.DoubleWritable;
import com.aliyun.odps.io.TableInfo;
import com.aliyun.odps.io.TableInputFormat;
import com.aliyun.odps.io.TableOutputFormat;
import com.aliyun.odps.io.Text;
import com.aliyun.odps.mapreduce.JobClient;
import com.aliyun.odps.mapreduce.JobConf;
import com.aliyun.odps.mapreduce.MapContext;
import com.aliyun.odps.mapreduce.Mapper;

public class UrbanMap {

    public static final Log LOG = LogFactory.getLog(UrbanMap.class);

    public static class UrbanMapper extends
        Mapper<Text, LongWritable> {
        @Override
        protected void setup(MapContext<Text, LongWritable> context)
            throws IOException, InterruptedException {
            super.setup(context);
        }

        private void setUrban(Record record, int year, MapContext<Text, LongWritable> context) throws
        IOException {
            Record r_urban = context.createOutputRecord("urban" + year);
```

³⁸ 也可以通过 SQL+UDTF 实现该功能，我们将在后面 UDTF 一节中介绍完成。

```

        r_urban.set("country_name", new Text(record.get("country_name").toString()));
        r_urban.set("country_code", new Text(record.get("country_code").toString()));
        r_urban.set("indicator_name", new Text(record.get("indicator_name").toString()));
        r_urban.set("indicator_code", new Text(record.get("indicator_code").toString()));
        if(! record.isNull("y" + year) ) {
            r_urban.set("value", new DoubleWritable(((DoubleWritable) record.get("y" + year)).get()));
            context.write(r_urban, "urban" + year);
        }
    }
}

@Override
public void map(LongWritable recordNum, Record record,
                MapContext<Text, LongWritable> context) throws IOException {

    // urban data (partitioned)
    for(int year=1960; year<=2012; ++year) {
        setUrban(record, year, context);
    }

}

}

public static void main(String args[]) throws IOException {

    JobConf job = new JobConf();
    job.setMapperClass(UrbanMapper.class);
    job.setNumReduceTasks(0);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(LongWritable.class);

    TableInputFormat.addInput(new TableInfo("urban_data"), job);

    boolean overwrite = true;
    for(int year=1960; year<=2012; ++year) {
        TableOutputFormat.addOutput(new TableInfo("urban", "year=" + year, "urban"+year,
        overwrite, job);
    }

    JobClient.runJob(job);

```

```
}  
}
```

这个 **MRTask** 非常简单，它只包含一个 **Mapper**，根据年份创建 **Output** 标签以及获取字段值，并相应写到各个分区中。如果字段值为空，则不写这条记录。实际上，**MRTask** 只能实现静态分区（即分区是已知的），对于动态分区（即分区是未知的），目前只能通过 **SQL** 实现，我们将在后面探讨如何创建动态分区。

7.4.3 编译和运行

1) 配置 **build.xml** 文件，通过 **ant** 编译，如下：

```
<?xml version="1.0" encoding="UTF-8"?>  
<project name="mr_example" default="package" basedir=".">  
  <property name="root" location="${basedir}" />  
  <property name="src.dir" location="${basedir}/src" />  
  <property name="build.dir" location="${root}/build/${ant.project.name}" />  
  <property name="build.classes" location="${build.dir}/classes" />  
  <property name="build.encoding" value="UTF-8" />  
  <property name="build.number" value="1.0.0" />  
  <property name="urban_example.jar.name" value="urban_example" />  
  <property name="javac.debug" value="false" />  
  <property name="javac.deprecation" value="false" />  
  
  <path id="classpath">  
    <pathelement location="${build.classes}" />  
    <fileset dir="${basedir}/lib" includes="*.jar" />  
  </path>  
  
  <target name="create-dirs">  
    <mkdir dir="${build.dir}" />  
    <mkdir dir="${build.classes}" />  
  </target>  
  
  <target name="init" depends="create-dirs">  
    <echo message="Initing: ${ant.project.name}" />  
  </target>  
  
  <available file="${src.dir}" property="src.dir.exist" />
```

```
<target name="compile-src" if="src.dir.exist">
    <javac encoding="${build.encoding}" source="1.6" target="1.6" srcdir="${src.dir};"
    destdir="${build.classes}" debug="${javac.debug}" includeantruntime="false"
    deprecation="${javac.deprecation}">
        <classpath refid="classpath" />
    </javac>
    <copy todir="${build.classes}">
        <fileset dir="${src.dir}" includes="**/*.xml" />
    </copy>
    <copy todir="${build.classes}">
        <fileset dir="${src.dir}" includes="**/*.properties" />
    </copy>
</target>

<target name="compile" depends="init">
    <echo message="Compiling: ${ant.project.name}" />
    <antcall target="compile-src" />
</target>

<target name="package" depends="compile">
    <echo message="Packaging: ${urban_example.jar.name}" />
    <echo message="${urban_example.classes}" />
    <jar jarfile="${build.dir}/${urban_example.jar.name}.jar" basedir="${build.classes}"
    includes="urban/**/*.properties">
        <manifest>
            <attribute name="Created-By" value="Alibaba CDO" />
            <attribute name="Implementation-Vendor" value="Alibaba CDO" />
            <attribute name="Implementation-Title" value="CDO DataProcess MapReduce" />
            <attribute name="Implementation-Version" value="${build.number}" />
            <attribute name="url" value="http://www.alidata.org" />
        </manifest>
    </jar>
</target>

<target name="clean">
    <echo message="Cleaning: ${ant.project.name}" />
    <delete dir="${build.dir}" />
</target>

</project>
```


这些代码都在目录 `urban_example` 下，`urban_example` 目录下的目录和文件结构大致如下：

```
./src/urban/UrbanMap.java
./lib
./build.xml
```

其中 `lib` 目录是从解压 `odps_clt_release_64.tar.gz` 包获取的。

运行 `ant`，会生成 `urban_example.jar` 包。

2) 在 `console` 中运行如下命令，就可以把原表数据导入到按年份分区的分区表中了：

```
add jar /home/admin/book/example/mapreduce/urban_example/build/mr_example/urban_example.jar -f;

jar -libjars urban_example.jar -classpath /home/admin/book/example/mapreduce/urban_example/build/mr_example/urban_example.jar urban.UrbanMap;
```

7.5 通过 SQL 分析处理

7.5.1 统计 PM10 值最大的国家

要通过 SQL 查询来统计 1960 年到 2012 年间每年 PM10 值最大的国家，首先按年份 `group`，查找每一年 PM10 最大的值，然后通过连接 `join` 操作，得到每年 PM10 值最大的国家，最后再根据年份进行排序，如下：

```
select a.country_name,a.value,a.year from urban a right outer join (select max(value) as v1,year from urban
where indicator_code='EN.ATM.PM10.MC.M3' group by year) b on
(a.indicator_code='EN.ATM.PM10.MC.M3' and a.value=b.v1 and a.year=b.year) order by a.year limit 43;
```

可以看到，ODPS SQL 语句语法和传统 SQL 语法非常相似，这是为了方便开发者而设计的。也许细心的读者已经注意到了，这里和传统 SQL 的一点区别在于，在排序 `order by` 时，需要给出 `limit` 关键字，这是因为如果没有给出 `limit`，SQL 会对全表进行排序，这样代价会非常大，可能会运行很长时间。由于一共有 43 个年份，所以这里给出 `limit 43`。

该 SQL 查询结果如下：

country_name	value	year
Sudan	317.1211699	1990
Sudan	297.5599326	1991
Armenia	366.5486228	1992
Iraq	272.1034244	1993

Iraq	270.3622488 1994
Iraq	246.4300016 1995
Sudan	237.4880022 1996
Iraq	305.3185137 1997
Iraq	240.4694128 1998
Sudan	292.5370567 1999
Sudan	231.8002765 2000
...	
+-----+-----+-----+	

由于有些年份该指标都没有值，所以只获取到该指标有值的年份的最大值。从结果可以看出，PM10 值最大的国际主要是苏丹国和伊拉克，可能是长期战争导致的吧。

7.5.2 统计指标报表

要统计每个国家在各个年份的所有指标报表，我们先创建一个表，把指标作为字段名称，各个指标的值保存到相应字段中。这样后续可以便于根据字段名称进行查询统计。由于 ODPS 表的字段名称全部都是小写，而且不支持 “.” 号，我们把 21 指标名称转换成全小写，且把 “.” 号替换成 “_” 作为字段名称。创建表的 SQL 语句如下：

```
create table urban_indicator (  
    country_name string,  
    country_code string,  
    en_atm_pm10_mc_m3 double,  
    en_pop_dnst double,  
    en_urb_lcty double,  
    en_urb_lcty_ur_zs double,  
    en_urb_mcty double,  
    en_urb_mcty_tl_zs double,  
    ep_pmp_desl_cd double,  
    ep_pmp_sgass_cd double,  
    is_rod_desl_pc double,  
    is_rod_engy_zs double,  
    is_rod_sgass_pc double,  
    is_veh_nveh_p3 double,  
    is_veh_pcar_p3 double,  
    is_veh_road_k1 double,  
    sh_h2o_safe_ur_zs double,  
    sh_sta_acsn_ur double,  
    si_pov_urgp double,  
    si_pov_urhc double,
```

```
sp_urb_grow double,  
sp_urb_totl double,  
sp_urb_totl_in_zs double) partitioned by (year string);
```

对于该场景，相当于要读取多条记录，获取其字段值，最后输出一条记录，结果记录包含多个字段。该功能可以通过 SQL+UDTF 来实现。UDTF 代码如下：

```
#!/coding=utf8  
from odps.udf import *  
  
@annotate('string,string,string,string,double,string->string,string,double,double,double,double,double,double,dou  
ble,double,double,double,double,double,double,double,double,double,double,double,double,double,string'  
)  
class UrbanIndicators(BaseUDTF):  
    def __init__(self):  
        self.country_name = None  
        self.country_code = None  
        self.year = None  
        self.en_atm_pm10_mc_m3= None  
        self.en_pop_dnst= None  
        self.en_urb_lcty= None  
        self.en_urb_lcty_ur_zs= None  
        self.en_urb_mcty= None  
        self.en_urb_mcty_tl_zs= None  
        self.ep_pmp_desl_cd= None  
        self.ep_pmp_sgas_cd= None  
        self.is_rod_desl_pc= None  
        self.is_rod_engy_zs= None  
        self.is_rod_sgas_pc= None  
        self.is_veh_nveh_p3= None  
        self.is_veh_pcar_p3= None  
        self.is_veh_road_k1= None  
        self.sh_h2o_safe_ur_zs= None  
        self.sh_sta_acsn_ur= None  
        self.si_pov_urgp= None  
        self.si_pov_urhc= None  
        self.sp_urb_grow= None  
        self.sp_urb_totl= None  
        self.sp_urb_totl_in_zs= None
```

```
self.idict = { "EN.ATM.PM10.MC.M3": "en_atm_pm10_mc_m3",

               "EN.POP.DNST": "en_pop_dnst",

               "EN.URB.LCTY": "en_urb_lcty",

               "EN.URB.LCTY.UR.ZS": "en_urb_lcty_ur_zs",

               "EN.URB.MCTY": "en_urb_mcty",

               "EN.URB.MCTY.TL.ZS": "en_urb_mcty_tl_zs",

               "EP.PMP.DESL.CD": "ep_pmp_desl_cd",

               "EP.PMP.SGAS.CD": "ep_pmp_sgas_cd",

               "IS.ROD.DESL.PC": "is_rod_desl_pc",

               "IS.ROD.ENG.Y.ZS": "is_rod_engy_zs",

               "IS.ROD.SGAS.PC": "is_rod_sgas_pc",

               "IS.VEH.NVEH.P3": "is_veh_nveh_p3",

               "IS.VEH.PCAR.P3": "is_veh_pcar_p3",

               "IS.VEH.ROAD.K1": "is_veh_road_k1",

               "SH.H2O.SAFE.UR.ZS": "sh_h2o_safe_ur_zs",

               "SH.STA.ACSN.UR": "sh_sta_acsn_ur",

               "SI.POV.URGP": "si_pov_urgp",

               "SI.POV.URHC": "si_pov_urhc",

               "SP.URB.GROW": "sp_urb_grow",

               "SP.URB.TOTL": "sp_urb_totl",

               "SP.URB.TOTL.IN.ZS": "sp_urb_totl_in_zs" }

def process(self, country_name, country_code, indicator_name, indicator_code, value, year):

    if (self.country_name and country_name != self.country_name) or (self.country_name and country_name

    == self.country_name and year and year != self.year):

self.forward(self.country_name, self.country_code, self.en_atm_pm10_mc_m3, self.en_pop_dnst, self.en_urb_lcty, s

elf.en_urb_lcty_ur_zs, self.en_urb_mcty, self.en_urb_mcty_tl_zs, self.ep_pmp_desl_cd, self.ep_pmp_sgas_cd, self.is

_rod_desl_pc, self.is_rod_engy_zs, self.is_rod_sgas_pc, self.is_veh_nveh_p3, self.is_veh_pcar_p3, self.is_veh_road

_k1, self.sh_h2o_safe_ur_zs, self.sh_sta_acsn_ur, self.si_pov_urgp, self.si_pov_urhc, self.sp_urb_grow, self.sp_urb_t

otl, self.sp_urb_totl_in_zs, self.year)

self.country_name = None

self.country_code = None

self.year = None

self.en_atm_pm10_mc_m3 = None

self.en_pop_dnst = None

self.en_urb_lcty = None

self.en_urb_lcty_ur_zs = None

self.en_urb_mcty = None
```

```
self.en_urb_mcty_tl_zs= None

self.ep_pmp_desl_cd= None

self.ep_pmp_sgas_cd= None

self.is_rod_desl_pc= None

self.is_rod_engy_zs= None

self.is_rod_sgas_pc= None

self.is_veh_nveh_p3= None

self.is_veh_pcar_p3= None

self.is_veh_road_k1= None

self.sh_h2o_safe_ur_zs= None

self.sh_sta_acsn_ur= None

self.si_pov_urgp= None

self.si_pov_urhc= None

self.sp_urb_grow= None

self.sp_urb_totl= None

self.sp_urb_totl_in_zs= None


# set the values

self.country_name = country_name

self.country_code = country_code

self.year = year

if indicator_code == 'EN.ATM.PM10.MC.M3':

    self.en_atm_pm10_mc_m3 = value

elif indicator_code == 'EN.POP.DNST':

    self.en_pop_dnst = value

elif indicator_code == 'EN.URB.LCTY':

    self.en_urb_lcty = value

elif indicator_code == 'EN.URB.LCTY.UR.ZS':

    self.en_urb_lcty_ur_zs = value

elif indicator_code == 'EN.URB.MCTY':

    self.en_urb_mcty = value

elif indicator_code == 'EN.URB.MCTY.TL.ZS':

    self.en_urb_mcty_tl_zs = value

elif indicator_code == 'EP.PMP.DESL.CD':

    self.ep_pmp_desl_cd = value

elif indicator_code == 'EP.PMP.SGAS.CD':

    self.ep_pmp_sgas_cd = value

elif indicator_code == 'IS.ROD.DESL.PC':
```

```

        self.is_rod_desl_pc = value
    elif indicator_code == 'IS.ROD.ENGZ.ZS':
        self.is_rod_engz_zs = value
    elif indicator_code == 'IS.ROD.SGAS.PC':
        self.is_rod_sgass_pc = value
    elif indicator_code == 'IS.VEH.NVEH.P3':
        self.is_veh_nveh_p3 = value
    elif indicator_code == 'IS.VEH.PCAR.P3':
        self.is_veh_pcar_p3 = value
    elif indicator_code == 'IS.VEH.ROAD.K1':
        self.is_veh_road_k1 = value
    elif indicator_code == 'SH.H2O.SAFE.UR.ZS':
        self.sh_h2o_safe_ur_zs = value
    elif indicator_code == 'SH.STA.ACSN.UR':
        self.sh_sta_acsn_ur = value
    elif indicator_code == 'SI.POV.URGP':
        self.si_pov_urgp = value
    elif indicator_code == 'SI.POV.URHC':
        self.si_pov_urhc = value
    elif indicator_code == 'SP.URB.GROW':
        self.sp_urb_grow = value
    elif indicator_code == 'SP.URB.TOTL':
        self.sp_urb_totl = value
    elif indicator_code == 'SP.URB.TOTL.IN.ZS':
        self.sp_urb_totl_in_zs = value

def close(self):
    if self.country_name:

self.forward(self.country_name,self.country_code,self.en_atm_pm10_mc_m3,self.en_pop_dnst,self.en_urb_lcty,s
elf.en_urb_lcty_ur_zs,self.en_urb_mcty,self.en_urb_mcty_tl_zs,self.ep_pmp_desl_cd,self.ep_pmp_sgass_cd,self.is
_rod_desl_pc,self.is_rod_engz_zs,self.is_rod_sgass_pc,self.is_veh_nveh_p3,self.is_veh_pcar_p3,self.is_veh_road
_k1,self.sh_h2o_safe_ur_zs,self.sh_sta_acsn_ur,self.si_pov_urgp,self.si_pov_urhc,self.sp_urb_grow,self.sp_urb_t
otl,self.sp_urb_totl_in_zs, self.year)

```

把该 udtf 作为资源添加到 odps 中，如下：

```

add py /home/admin/book/example/mapreduce/udf_example/lib/udtf_indicators.py -f;
create function get_indicator as 'udtf_indicators.UrbanIndicators' using 'udtf_indicators.py';

```

再执行如下 SQL 语句：

```
insert overwrite table urban_indicator partition(year)

select get_indicator(country_name, country_code, indicator_name, indicator_code,value, year) as
(country_name, country_code, en_atm_pm10_mc_m3, en_pop_dnst, en_urb_lcty, en_urb_lcty_ur_zs,
en_urb_mcty,
en_urb_mcty_tl_zs,ep_pmp_desl_cd,ep_pmp_sgas_cd,is_rod_desl_pc,is_rod_engy_zs,is_rod_sgas_pc,is_veh_nv
eh_p3,is_veh_pcar_p3,is_veh_road_k1,sh_h2o_safe_ur_zs,sh_sta_acsn_ur,si_pov_urgp,si_pov_urhc,sp_urb_gro
w,sp_urb_totl,sp_urb_totl_in_zs, year)

from (

select * from (

select country_name,country_code, indicator_name, indicator_code,value, year from urban group
by country_name, year, country_code, indicator_name, indicator_code,value) t1

cluster by t1.country_name, t1.year) t2;
```

这样，就生成一张结果表 urban_indicator，它表示在各个年份的每个国家的所有指标。

现在，要查询中国在 2009 年的所有指标报表，SQL 语句如下：

```
select * from urban_indicator where country_code='CHN' and year='2009';
```

假如要查看中国和美国的 PM10 每年的变化情况比较，SQL 查询如下：

```
select t1.year, t1.en_atm_pm10_mc_m3 as china_pm10, t2.en_atm_pm10_mc_m3 as usa_pm10 from (select
country_code, year, en_atm_pm10_mc_m3 from urban_indicator where country_code='CHN') t1 right outer join
(select country_code, year, en_atm_pm10_mc_m3 from urban_indicator where country_code='USA') t2 on
t1.year=t2.year where t1.en_atm_pm10_mc_m3 is not NULL or t2.en_atm_pm10_mc_m3 is not NULL order by
t1.year limit 50;
```

该查询输出结果如下：

```
+-----+-----+-----+
| year | china_pm10 | usa_pm10 |
+-----+-----+-----+
| 1990 | 113.7111087 | 29.61434134 |
| 1991 | 111.7143966 | 28.14684198 |
| 1992 | 108.6854553 | 27.80046752 |
| 1993 | 104.1039217 | 27.33393185 |
| 1994 | 93.99340055 | 27.02428916 |
| 1995 | 87.86843985 | 25.76125932 |
| 1996 | 89.80956437 | 25.31688046 |
| 1997 | 83.19823843 | 25.08196163 |
| 1998 | 77.75132905 | 24.74551158 |
| 1999 | 78.87803164 | 24.28498371 |
| 2000 | 87.85521238 | 23.8068895 |
| 2001 | 82.40943699 | 24.30621139 |
| 2002 | 81.39831421 | 22.87420047 |
```

```
| 2003 | 81.94416266 | 22.4847698 |
| 2004 | 81.97528535 | 22.27449004 |
| 2005 | 77.5277191 | 21.66618949 |
| 2006 | 74.62295048 | 20.71275543 |
| 2007 | 68.79011544 | 20.23678892 |
| 2008 | 62.30584965 | 19.09671328 |
| 2009 | 60.22997912 | 18.0857484 |
| 2010 | 58.85967547 | 17.78026461 |
+-----+-----+-----+
```

可以看到，PM10 值都随年份呈下跌趋势，中国的 PM10 值是美国的 3 ~ 4 倍，不过和 1990 年相比，中国的 PM10 值在 2010 年已经下降了接近一半，虽然还不够理想，但还是得到了显著改善！

7.6 通过 PLSQL 统计分析

ODPS SQL 存储过程支持使用变量、条件判断和循环控制，便于完成较复杂的逻辑处理。要比较 2010 年低收入、中等收入和高收入国家在人口密度、城镇人口增长率和城镇人口（占总人口比例）三个指标的情况，输出每个指标值最大的国家类型，我们可以通过存储过程完成这个功能，虽然代码有些冗长，但逻辑很简单清晰。代码如下：

```
declare
    low string;
    middle string;
    high string;

    low_dnst double;
    middle_dnst double;
    high_dnst double;

    low_grow double;
    middle_grow double;
    high_grow double;

    low_urb double;
    middle_urb double;
    high_urb double;

    max_dnst string;
    max_grow string;
    max_urb string;
```

```
begin

    low      := 'LIC';
    middle   := 'MIC';
    high     := 'HIC';

    max_dnst := low;
    max_grow := low;
    max_urb  := low;

    select en_pop_dnst, sp_urb_grow, sp_urb_totl_in_zs into $low_dnst, $low_grow, $low_urb from
    urban_indicator where country_code=$low and year='2010';

    select en_pop_dnst, sp_urb_grow, sp_urb_totl_in_zs into $middle_dnst, $middle_grow, $middle_urb from
    urban_indicator where country_code=$middle and year='2010';

    select en_pop_dnst, sp_urb_grow, sp_urb_totl_in_zs into $high_dnst, $high_grow, $high_urb from
    urban_indicator where country_code=$high and year='2010';

    if low_dnst > middle_dnst then
        if low_dnst < high_dnst then
            max_dnst := high;
        end if;
    else
        if middle_dnst > high_dnst then
            max_dnst := middle;
        else
            max_dnst := high;
        end if;
    end if;

    if low_grow > middle_grow then
        if low_grow < high_grow then
            max_grow := high;
        end if;
    else
        if middle_grow > high_grow then
            max_grow := middle;
        else
            max_grow := high;
        end if;
    end if;

end if;
```

```
if low_urb > middle_urb then
    if low_urb < high_urb then
        max_urb := high;
    end if;
else
    if middle_urb > high_urb then
        max_urb := middle;
    else
        max_urb := high;
    end if;
end if;

print max_dnst;
print max_grow;
print max_urb;
end;
```

保存为 rank.pl 文件，执行命令如下：

```
clt/bin/odpscmd -x /home/admin/book/example/mapreduce/pl_example/rank.pl
```

输出结果如下：

```
max_dnst
-----
MIC

max_grow
-----
LIC

max_urb
-----
HIC
```

从输出结果可以看到，在 2010 年，中等收入国家人口密度最大，低收入国家城镇人口增长率最大，而高收入国家城镇人口（占总人口比例）最大。

在存储过程中，一种很常见的使用方式是将其逻辑处理结果作为某个字段写入到一张结果表中，用于后续分析处理。

7.7 通过 XLib 统计

要对中、美、日、法、德、英这六个国家每年的能源消耗情况进行排序，这里，我们将考虑三个指标：道路部门能源消耗量、道路部门人均燃料消耗量、道路部门人均汽油消耗量。对这三个列在各个年份的值按国家进行排序，由于 SQL 本身不支持多列排序，如果实现 udf 的话，比较复杂，而且对于大数据量，可能会存在性能问题。幸运的是，ODPS Xlib 轻松为我们解决了这个问题。Xlib 是 ODPS 平台内专用于数据挖掘算法的模块，重点实现性能要求很高的统计分析、机器学习算法，以支持各种复杂数据挖掘场景。

首先，我们先创建一张表，只包含中、美、日、法、德、英这六个国家的以上三个指标在各个年份的值。创建表 SQL 如下：

```
create table urban_road(country_name string,
    is_rod_engy_zs double,
    is_rod_desl_pc double,
    is_rod_sgas_pc double,
    year string);
```

插入数据，SQL 如下：

```
insert overwrite table urban_road
select country_name, is_rod_engy_zs, is_rod_desl_pc, is_rod_sgas_pc, year from urban_indicator
where country_code = 'CHN'
or    country_code = 'USA'
or    country_code = 'GBR'
or    country_code = 'DEU'
or    country_code = 'JPN'
or    country_code = 'FRA';
```

下面，我们通过 Xlib 提供的多列排序和分位（sort_rank）算法来实现排序，执行命令如下：

```
sort_rank -i urban_road -o urban_road_s, urban_road_r -c 1,2,3 -g 4 -a 0;
```

该命令会生成两张表：排序表（urban_road_s）和 分位表（urban_road_r）。在这个例子中，顾名思义，排序表是对各个指标的排序，分位表是对各个指标的实际值在从 0 到 100 各个分位值的比例。排序这个概念很容易理解，而分位则不然，我们将在 Xlib 专题中详细说明。

urban_road_s 的部分结果显示如下：

```
odps@ odps_book>read urban_road_s 6;
+-----+-----+-----+-----+
| country_name | year | is_rod_engy_zs_rk | is_rod_desl_pc_rk | is_rod_sgas_pc_rk |
+-----+-----+-----+-----+
| Japan       | 2006 | 2.0                | 2.0                | 5.0                |
| United States | 2006 | 6.0                | 5.0                | 6.0                |
```

China	2006	1.0	1.0	1.0	
Germany	2006	3.0	3.0	3.0	
France	2006	4.0	6.0	2.0	
United Kingdom	2006	5.0	4.0	4.0	
+-----+-----+-----+-----+-----+					

可以看到，在 2006 年，中国在能源消耗的三个指标都居第一，可以看到中国在节约能源上还有相当大的发展空间。

以上排序分位是对每个年份，比较不同国家各指标的排序。试着想想看，如果希望对每个国家，比较其在不同年份的各个指标排序，该如何实现呢？有兴趣的话，可以自己动手试试。

除了本示例教程提到的 Tunnel、MapReduce、SQL/PL、Xlib 之外，ODPS 还提供了 ODPS R、ODPS Graph 等产品服务，这里不再一一示例。