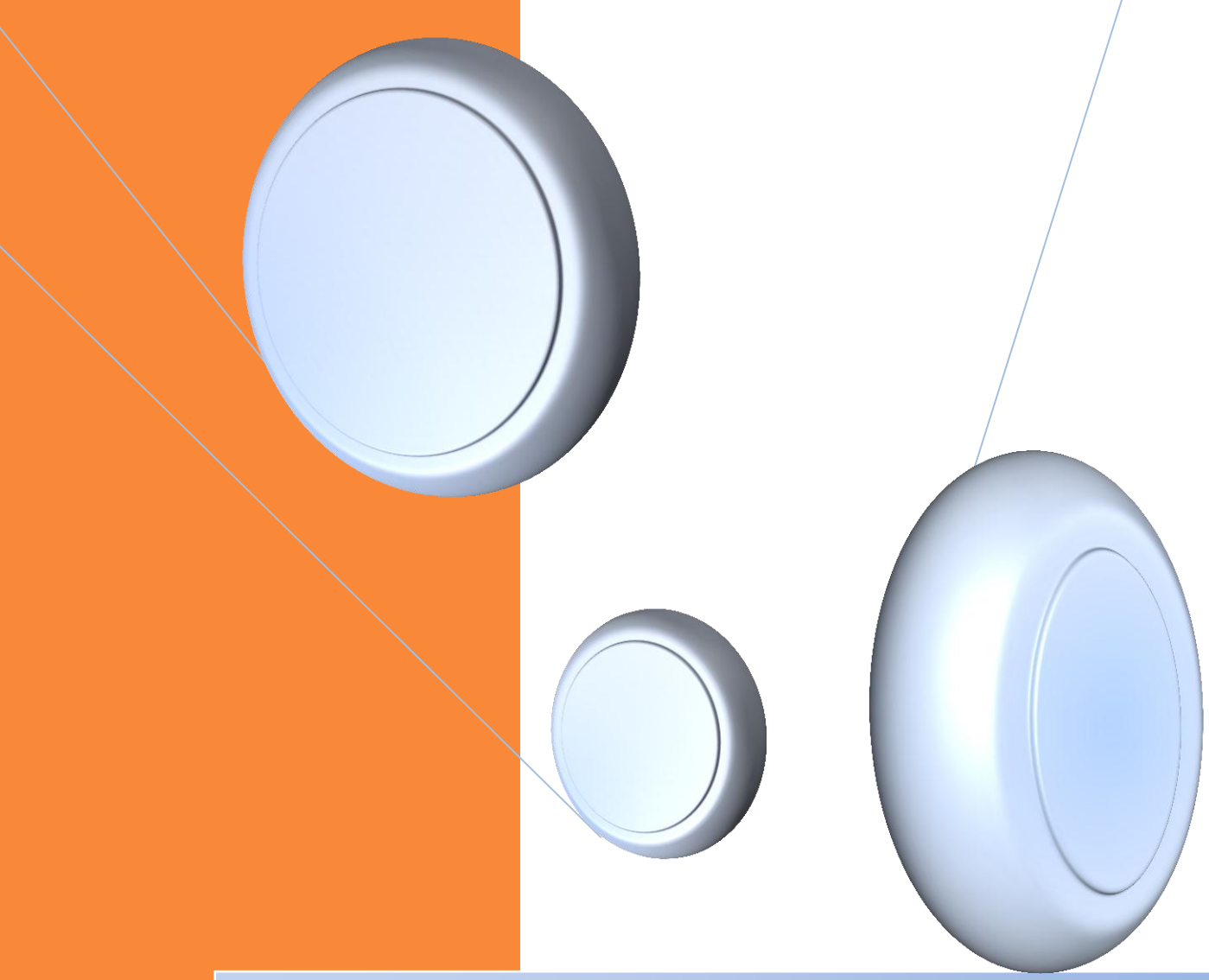


OceanBase 0.4.2
快速入门

文档版本：Beta 02

发布日期：2013.11.30

支付宝（中国）网络技术有限公司·OceanBase 团队

前 言

概述

本文档主要介绍了如何快速入门OceanBase 0.4.2的方法。

读者对象

本文档主要适用于：

- 开发工程师。
- 维护工程师。
- 安装工程师。

通用约定

在本文档中可能出现下列各式，它们所代表的含义如下。

格式	说明
警告	表示可能导致设备损坏、数据丢失或不可预知的结果。
注意	表示可能导致设备性能降低、服务不可用。
小窍门	可以帮助您解决某个问题或节省您的时间。
说明	表示正文的附加信息，是对正文的强调和补充。
宋体	表示正文。
粗体	表示命令行中的关键字（命令中保持不变、必须照输的部分）或者正文中强调的内容。
斜体	用于变量输入。
{ a | b | ... }	表示从两个或多个选项中选取一个。
[ ]	表示用“[ ]”括起来的部分在命令配置时是可选的。

修订记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本。

版本和发布日期	说明
Beta 02（2013-11-30）	第一次发布Beta版本，适用于OceanBase 0.4.2。
01（2013-10-30）	第一次正式发布，适用于OceanBase 0.4.1。

联系我们

如果您有任何疑问或是想了解 OceanBase 的最新开源动态消息，请联系我们：

支付宝（中国）网络技术有限公司·OceanBase 团队

地址：杭州市万塘路 18 号黄龙时代广场 B 座；邮编：310099

北京市朝阳区东三环中路 1 号环球金融中心西塔 14 层；邮编：100020

邮箱：alipay-oceanbase-support@list.alibaba-inc.com

新浪微博：http://weibo.com/u/2356115944

技术交流群（阿里旺旺）：853923637

目 录

1 学习资料获取

学习一个产品首先需要从文档入手，因此我们提供了 OceanBase 的各类宝典：

- 产品文档：https://github.com/alibaba/oceanbase/wiki
- 设计文档和其他学习资料：https://github.com/alibaba/oceanbase/tree/oceanbase_0.4/doc，即 OceanBase 源码的“/doc”目录下。

2 关于安装部署

当你获取了学习资料，了解了一些 OceanBase 的知识后，一定迫不及待想要亲自动手实践了吧？那么接下来，我们将为你慢慢道来。

在《OceanBase 0.4.2 安装指南》中我们详细介绍了“采用 RPM 包安装”和“采用源码安装”，此处不再赘述了。

本章节主要介绍的是当我们采用源码安装时，如何使用 deploy 工具完成 OceanBase 部署。

2.1 编译坏境

首先我们需要有一个 OceanBase 能够正常编译和安装的环境：

- Redhat 5u（6u 的编译还有点问题，以后会修复）
- kernel version >= 2.6.18
- gcc version = 4.1.2 (gcc 4.4 目前编译还有点问题)
- automake version >= 1.10.2 (for LIBTOOLFLAGS=--preserve-dup-deps)

2.2 依赖库

要正常编译、安装、运行 OceanBase，还需要依赖一些动态库，详细安装方法请参见《OceanBase 0.4.2 安装指南》的“4.2 安装动态库”章节：

- liblzo2
  压缩库，OceanBase 需要用它来压缩静态数据。
- Snappy
  Google 出品的压缩库，OceanBase 用它来压缩静态数据。Snappy 依赖于 lzo 库，因此安装 Snappy 前请先装 lzo 库。
- Libaio
  Oceanbase 中用到了 AIO，需要 libaio 的支持。
- Libnuma
  Oceanbase 中用到了 NUMA 支持，需要 libnuma 的支持。
- gest 和 gmock
  这两个库为可选。如果你执行./configure –with-test-case=no 不编译 OB 的 test，那么可以不安装。

2.3 源码下载地址

安装 OceanBase 我们需要获取以下源码：

- OceanBase 源码
  http://svn.app.taobao.net/repos/oceanbase/branches/dev
- tbsys 源码
  http://svn.taobao-develop.com/repos/ttsc/branches/V3286_common_20100813/common/tbsys
- libeasy 源码
  http://svn.taobao-develop.com/repos/ttsc/trunk/easy

2.4 安装部署 OceanBase

学习了 OceanBase 知识，准备好编译坏境，安装完依赖库，获取到 OceanBase 源码，那么接下来呢？没错，我们终于可以安装部署 OceanBase 了！

*** 前提条件**

当然，在安装部署 OceanBase 前，还需要再次进行一些检查和设置。请仔细阅读本小节。

使用 deploy 部署 OceanBase 前请完成以下配置，涉及内容可参考《OceanBase 0.4.2 安装指南》：

- 已经设置 deploy 工具所在的主机和所有部署 OceanBase 主机之间的信任关系。
  详细请参见《OceanBase 0.4.2 安装指南》的“3.4 配置免登录”。
- 在所有需要部署 OceanBase 的主机上均完成依赖库安装、tbsys 安装和 libeasy 安装。
  详细请参见《OceanBase 0.4.2 安装指南》的“4.2 安装动态库”、“4.3 安装 tbsys 和 tbnet”和“4.4 安装 libeasy”。
- 已经创建 OceanBase 服务启动所需的目录。
  详细请参见《OceanBase 0.4.2 安装指南》的“2.5 创建数据磁盘挂载点”。

*** 编译 OceanBase**

注意：*在所有需要部署 OceanBase 的主机上完成编译 OceanBase。*

1. 进入 OceanBase 源码存放目录。
   cd ~/ob_source_dir
2. 执行以下命令，初始化安装。
   ./build.sh init

3. 执行以下命令，进行编译配置。
   ./configure [--with-release|--with-test-case|
   --with-tblib-root=|--with-easy-root=]
4. 依次执行以下命令，编译 OceanBase。
   make

*** 使用 deploy 部署**

1. 执行以下命令，进入 deploy 工具目录。其中“ob_source_dir”为 OceanBase 源码存放目录。
   cd ~/ob_source_dir/tools/deploy
2. 使用 **vi** 编辑器修改“config.py”文件。

   a. 在单引号之间输入需要部署 OceanBase 的机器 IP。

```
…
ObCfg.default_hosts    = '10.10.10.2 10.10.10.3'.split()
…
```

   b. 添加行。ob1 是你想要部署在目的机器上的安装文件所在的目录。

```
ob1 = OBI(src='ob_source_dir',tpl =
dict(schema_template=read('ob_new_sql_test.schema')))
```

3. 执行以下命令，运行 deploy 工具。
   ./deploy.py.ob1

*** 连接 OceanBase**

成功部署 OceanBase 后，则可使用 MySQL 客户端连接 OceanBase。

mysql -h *merge_server_host* -P *mysql_listen_port* -u admin -padmin

- “merge_server_host”为任意 MergeServer 所在主机的 IP。
- “mysql_listen_port”为 mysql 监听端口，可查看命令行中的设定。
- 初始“用户名/密码”为“admin/admin”。

当出现以下信息时，就说明我们能够连接到 OceanBase，并正常使用了。至此，一切搞定！

当然，如果不小心失败了，那也没关系，我们有强大的后援团——[支付宝（中国）网络技术有限公司·OceanBase 团队]！

```
$ mysql -h 10.10.10.4 -P2880 -uadmin -padmin
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 0
Server version: 5.5.1 OceanBase 0.4.2.5 (r17148) (Built Oct  23 2013 17:59:51)
Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql>
```

3 源码结构

在亲手安装和使用了 OceanBase 后，各位是不是有种想要深入了解 OceaBase 源码的欲望呢？接下来我们将简单介绍下 OceanBase 的源码学习方法。

3.1 源码目录

OceanBase 源码目录说明如表 3-1 所示

表 3-1 OceanBase 源码目录说明

oceanbase 目录	描述
|-- doc	Oceanbase 的相关文档
|-- rpm	Build rpm 包需要的文件
|-- script	主要是部署 ha 需要的脚本
|-- src	Source 目录
|-- tests	测试用例
|-- tools	外围工具

Src 目录说明如表 3-2 所示。

表 3-2 OceanBase 源码目录说明

Src 目录	描述
|-- common	基础库，公用模块
|-- sql	sql 相关(语法解析，执行计划，物理运算符)
|-- sstable	v1 &v2 SSTable
|-- compactsstable	ups 分发的紧凑型 SSTable
|-- compactsstablev2	v3 SSTable
|-- chunkserver	ChunkServer
|-- mergeserver	MergeServer
|-- rootserver	RootServer
|-- updateserver	UpdateServer
|-- importserver	控制旁路导入 importserver
|-- obmysql	OB MySQL Client
|-- lsync	同步 UPS Commit 工具

3.2 编码特征

在开始阅读 OceanBase 代码之前，希望你了解一下 OceanBase 代码的一些主要特征。

参考文档：《OceanBase c&c++编码规范》

常规的方法请参见“[more] effective c++”之类的读本，本文只摘录几条和其他项目明显不同的：

1. 系统中内存分配和释放分别为 ob_malloc()和 ob_free()，这个是对系统内存分配函数的简单包装，调用的时候最好带上 mod_id，系统可以跟踪是哪个模块分配的内存。
2. 尽量不使用 new 和 delete，不得已的时候用 placement new。在调用比较频繁的函数中分配内存是不利于性能的。由此带来很多惯用法：
    - 我们的大对象一般是被重用的，一般都实现一个 reset 方法，每次使用之前会被调用确保对象是干净的。
    - 有一个叫 GET_TSI_MULT 的宏，用于分配线程专用的对象，在线程内部被不停的重复使用。
3. 有一个叫做 PageArena 的内存分配器，用于零星分配内存，统一释放的场景。
4. OB 中不使用 STL 容器，所以会有 ObArray, ObList, ObVector 之类的容器。
5. 所有函数中只能有一个 return 语句，因此你会看到每个函数多数都是此类的代码写成：

```
int ret = OB_SUCCESS;
if (OB_SUCCESS    != (ret = do_something1()))
{
   print_do_something1_error_log();
}
else if (OB_SUCCESS != (ret = do_something2()))
{
   print_do_something2_error_log();
}
else if (OB_SUCCESS != (ret = do_something3()))
{
   print_do_something3_error_log();
}
return ret;
```

    - 多数完成一般功能的函数会以 OB_SUCCESS 返回成功，其他值返回错误代码。

- 一个函数如果要分几个步骤完成，请把每个步骤封装成一个函数，符合以上返回值规范。
- 每个错误处理都需打印尽可能助于分析错误的日志，包括输入参数，状态信息。

3.3 common

common 下面的都是一些公用代码，有几种：

- 基础类，替代 STL 里面的一些容器，通用数据结构（ObArray, ObVector, hash, btree）。
- 内存分配器(ob_malloc.cpp, ob_memory_pool.cpp, page_arena.cpp)。
- 各个模块都会使用的公共数据结构、接口。
- 序列化/反序列化方法，远程接口封装。
- 网络接口封装(ObClientManager, ObBaseClient)。
- schema 封装。

3.4 网络框架封装

网络框架封装分为服务端和客户端两部分。

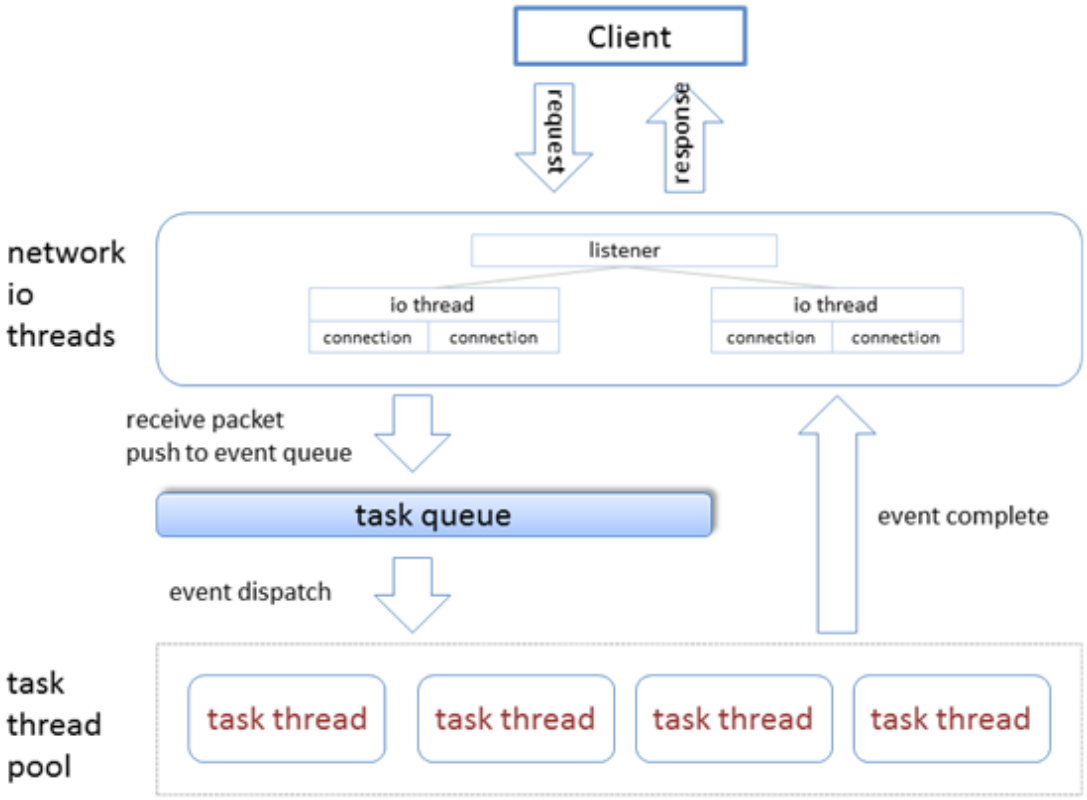
3.4.1 服务端

我们的每个 Server（RS,UPS,CS,MS）都是使用相同的网络 io 模型进行服务（UPS 有少许差别），都是基于 libeasy 库的封装而成。

如果你想理解 libeasy 的工作细节请参见立德同学的分析：“OceanBase 使用 libeasy 原理源码分析.doc”

server 网络 io 处理模型如图 3-1 所示。

图 3-1 io 处理模型



处理流程如下：

1. libeasy 的 io 处理线程从 connection 上读到客户端发送的请求数据。
2. 调用预先设置好的 handler.decode 方法创建 OB 需要的 packet。
3. 调用 handler.process 方法，process 有两种：
   - 直接利用 io 线程处理完成，发送 response。
   - 绝大多数采用方式是将 packet 放入任务队列，等待任务线程池中的某个线程处理完毕，构造返回包(reponse)，挂在 request->opacket 上，调用 easy_request_wakeup 唤醒 io 线程发送响应给客户端。

每个 server 都会有一个 Ob*Server 的类，继承于 ObSingleServer(common/ob_single_server.cpp)，后者继承于 ObBaseServer；对于有多个任务队列的 server(比如 ObUpdateServer 来说)，直接继承自 ObBaseServer(common/ob_base_server.cpp)，自己创建任务队列。

ObSingleServer 适用于一个 Server 只有一个任务队列的情况(CS,MS,RS)；完成一系列的初始化工作（创建 io 线程，设置 handler，监听网络端口，启动任务线程池等）。

ObSingleServer 实现了一个 handPacket 接口，这个接口会在 easy 的 io thread 读到请求并转化为 packet 以后 push 到任务队列中；ObSingleServer 有一个 ObPacketQueueThread((common/ob_packet_queue_thread.cpp))的成员，里面封装了一个任务队列和一堆任务线程，任务线程会等待在一个信号量上，一旦

queue 中有了 packet，信号量会被 singal，会有某个线程被唤醒。ObSingleServer 会同时实现 ObPacketQueueThread 的一个 handler，任务线程拿到这个 packet 进行处理，就调用这个 handler. handle_request ()方法，而这个 handle_request 最终会调用一个 do_request 的模板方法，由具体的 Ob*Server 类进行实现。具体过程请参考“ObPacketQueueThread 实现”。

Ob*Server 一般会有一个 Ob*Service 的成员，用于处理具体的消息。以 ObChunkServer(chunkserver/ob_chunk_server.cpp)为例，有一个对应的 ObChunkService 类。ObChunkServer 实现的消息处理函数 do_request 会调用 ObChunkService::do_request，这个函数就是具体的消息处理过程了。

```
easy.handler.process()
  |→  ObSingleServer:: handPacket
       |→  ObPacketQueueThread::push
            |→  wakeup task thread
                 |→  ObSingleServer::hand_request()
                      |→  ObSingleServer::do_request
                           |→  Ob*Server::do_request
                                |→  Ob*Service::do_request
```

3.4.2 客户端

具体的发送消息的类是由 ObClientManager(common/ob_client_manager.cpp)封装的，这个类中有一系列(send_packet,post_packet)的方法，如果是在 Server 里想要与别的 Server 进行消息交互，那么直接使用这个 ObClientManager 即可，每个 Ob*Server 类中都会有一个 ObClientManager 成员，初始化过程中会设定好，直接使用即可。

如果要一个独立的客户端，可以使用另外的一个简单的类 ObBaseClient 封装。

因为 ObClientManager 发送的都是 ObPacket，发送消息之前还需要进行打包输入参数，并解包返回的 response packet，使用起来多是一些重复的代码，因此有一个类 ObRpcStub 用来封装上述过程。

ObRpcStub 利用宏和模板实现了如下成员函数：

```
ObRpcStub::send_x_return_y(server, timeout, request_packet_code, rpc_version, input_params, output_params);
```

这一个函数系列，其中 x,y 分别表示输入和输出参数的个数； 0 <= x <= 6; 0 <= y <= 3;

如果你要往指定的 server 发送一个消息，并得到返回的结果，都可以通过调用这个函数（我们称为远程调来实现用），其中前 4 个参数是固定的：

- server: 远程调用的目的 Server。
- timeout: 远程调用等待响应的超时时间。
- request_packet_code：发送的消息代码。

- rpc_version：远程调用的协议版本。
- input_params： 如果你有多个输入参数，请将这些参数依次传入，传入顺序与 server 解析的顺序相同。
- out_params: 如果你有多个输出参数，请将输出参数依次传入，传入顺序与 server 打包顺序相同。

对于 input_params 和 output_params 中的每个参数有些限制，要么是基本类型（如 int , double, float, bool, const char*）；要么是一个普通类，实现了 serialize 和 deserialize 方法。

3.5 OBMySQL

讲完了网络框架部分以后，请求已经以包的形式传入到 Server 端了；以一个具体的查询请求为例，我们可以流转每个 Server，讲述 Server 之间是如何交互的。

假设有如下查询的 SQL:

```
Select t1.c1, t2.c1, sum(t2.qty)
 From t1, t2
 Where t1.c2 = t2.c2
 And t1.c3 > 5
 And t2.c3 < sin10°
 Group by t1.c1, t2.c1
 Having count(t2.qty) < 3
 Order by t2.c1
 Limit 2, 5;
```

通过 MySQL 客户端发送到 MergeServer 端，首先被 MergeServer 中的 ObMySQLServer::handlePacketQueue(obmysql/ob_mysql_server.cpp)获取，注意 ObMySQLServer 不是从 ObSingleServer 继承的，因为要处理 mysql command，有两个任务队列(ObMySQLCommandQueueThread), 但处理过程基本相同。

```
ObMySQLServer::handlePacketQueue
    |→ ObMySQLServer::do_com_query
        |→ ObSql::direct_execute?
        |→ ObMySQLServer::send_response
```

ObMySQLServer 在执行 SQL 之前还处理了 mysql client login 过程，这里面有些特殊的方法，具体说明可参考“OceanBase 使用 libeasy 原理源码分析.doc”，具体细节请参见代码。

3.6 SQL

SQL 模块下包含了整个 SQL 语句前期的处理过程，包括 SQL 语法解析，物理执行计划生成，执行计划优化。

物理执行计划，是由若干物理运算符（Physical Operator）组成的树形结构。每个物理操作符会完成一个特定的数据代数运算，如投影(Project)、过滤(Filter)、排序(Sort)、聚集(GroupBy)等。

物理执行计划执行的过程就是树形结构的根节点执行迭代数据的过程。根节点会驱动字节子节点进行数据迭代。每层操作符之间利用行接口进行数据传递。

物理运算符的设计与实现细节，请参见竹翁编写的“OceanBase SQL 物理运算符详细设计”。

接“3.5 OBMySQL”，SQL 的执行过程：

```
ObSql::direct_execute
    |→ ObSql::process_special_stmt_hook?     //处理一些 mysql 的命令的(show warnings)
    |→ ObSql::generate_logical_plan          //生成逻辑执行计划
    |→ ObSql::generate_physical_plan         //生成物理执行计划
    |→ ObResultSet:: set_physical_plan?      //这里只是把物理执行计划挂上去了，还没执行。
```

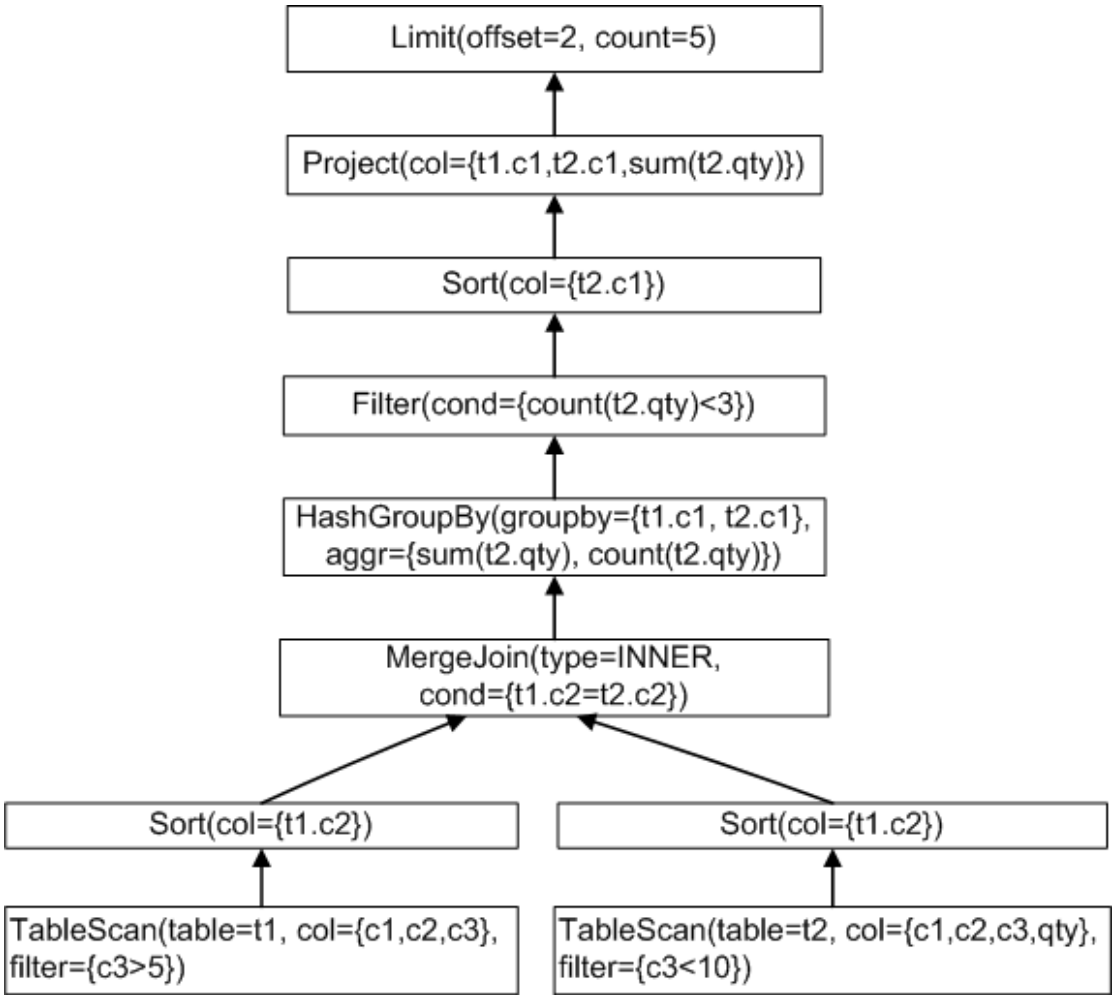
真正的物理计划是在哪儿执行的呢？在“3.5 OBMySQL”中调用的最后一步：ObMySQLServer::send_response。

```
ObMySQLServer::send_response
    |→ ObMySQLResultSet::open ObPhyOperator::open() //调用根节点 open，一次调用各个字节点 open，完成一些初始化工作
    |→ ObMySQLServer ::send_row_packets
            |→ ObMySQLResultSet::next_row                    //这个是不停迭代的过程，每次迭代一行数据， 一直到 OB_ITER_END
                    |→ ObResultSet::get_next_row
    |→ ObMySQLResultSet::close()                             //关闭物理执行计划，释放资源，完成善后工作
```

物理执行计划的生成是一个比较复杂的过程，目前我们并没太多的描述文档，主要还是要靠阅读代码（上面提到的几个函数）。

以上面的 SQL 为例，最终生成的物理执行计划如图 3-2所示。。

图 3-2 物理执行计划



物理执行计划涉及到的就是一个个的物理操作符，物理操作符的功能相对比较单一，可以直接阅读相关的类实现，都在源码的“sr\csql”目录下。

3.7 MergeServer

对于查询语句来说，一般执行计划的最底层都是一个 TableScan(sql/ob_table_scan.h)，MergeServer 模块有很大部分都是在实现这个操作符。

TableScan 的功能就是完成对指定 table，指定范围的数据扫描。为了提高效率，TableScan 同时还会完成一些 filter、groupby 之类的操作。

TableScan 的主体是 ObTableRpcScan(sql/ob_table_rpc_scan.h)，最终完成实际工作的是 ObMsSqlScanRequest(mergeserver/ob_ms_sql_scan_request.h)。

```
TableScan -> ObTableRpcScan ->    ObRpcScan -> ObMsSqlScanRequest
```

ObMsSqlScanRequest 进行 scan 的过程是一个并发执行的过程：

1. 将设置要扫描的 range 根据 RootServer 的 RootTable 里面的分片信息，拆分为 1 个或多个 sub range，每个 sub range 对应一个 ChunkServer 上的 Tablet。
2. 为每个 sub range 构造一个 ObMsSqlSubScanRequest(mergeserver/ob_ms_sql_sub_scan_request.h);并发将每个 SubRequest 发送给对应的个 ChunkServer，最终阻塞在线程上等待个 ChunkServer 返回结果。
3. MergeServer 会检查返回的结果，一旦返回的结果满足了操作符的要求(比如 Limit 拿到了足够的行)，返回给上层操作符。

当然实际的过程比上面描述的复杂，每个 SubRequest 无法一次返回完整的数据就会涉及到流式接口，另外每次请求还有并发量限制，具体细节请参见晓楚编写的“MergeServer 请求流程.pptx”。

3.8 ChunkServer

ObMsSqlSubScanRequest 是针对一个 Tablet 的查询。ChunkServer 只处理针对单个 Tablet 的查询请求。ObMsSqlSubScanRequest 的实现实际上是发送了一个 CS_SQL_SCAN 的 packet，ChunkServer 在收到这个类型的 packet 以后，会构造一个操作符 ObTabletScan(sql/ob_tablet_scan.cpp)执行实际的查询请求。

```
ObChunkService::do_request
    |→  ObChunkService:: cs_sql_scan
          |→ ObChunkService::cs_sql_read
                |→ ObTabletService::open
                      |→ ObTabletScan::create_plan
                      |→ ObTabletScan::open
                |→ ObTabletService::fill_scan_data
                      |→ ObTabletScan::get_next_row
```

ObTabletScan 的主要工作是将两部分的数据进行合并的过程，一部分数据是通过操作符 ObSSTableScan(sql/ob_sstable_scan.h)从本地 SSTable 文件中读取的基准数据，另一部分是通过操作符 ObUpsScan(sql/ob_ups_scan.h)从 UPS 远程读取的增量数据，然后通过 ObTabletScanFuse(sql/ob_tablet_scan_fuse.h)进行合并。如果 scan 的表配置为内置 join 功能，那么还需要在 ObTabletScanFuse 上架一层 ObTabletJoin(sql/ob_tablet_join.h)。

ObTabletScan::create_plan 就是根据需要构造上面这些操作符。

ChunkServer 除了负责查询以外，还有一大部分工作是进行数据每日合并。每日合并就是将本地 SSTable 的静态数据与 UPS 的冻结表数据进行合并的过程。这个合并以 Tablet 为单位来进行的，其中读取数据合并的工作也是由 ObTabletScan 来实现的。

每日合并代码请参见: ObChunkMerge(chunkserver/ob_chunk_merge.cpp), ObTabletMergerV1(chunkserver/ob_tablet_merger_v1.cpp), ObTabletMergerV2(chunkserver/ob_tablet_merger_v2.cpp)。

ChunkServer 利用一个 tablet image 文件对 Tablet 进行管理，参考类实现为 ObTabletImage<ob_tablet_image.h>，文件格式请参见文档“chunkserver_tablet 索引元数据结构.docx”

其他参考文档“chunkserver 概述.ppt”。

3.9 SSTable/compactsstablev2

SSTable 模块主要实现了静态数据 SSTable 文件的读写过程。

“src\sstable”目录下是 v1,v2 的 SSTable 实现，“src\compactsstablev2”目录下是 v3 的 SSTable 实现。两者实现了相同的接口 ObRowkeyPhyOperator(sql/ob_rowkey_phy_operator.h)。

SSTable 的代码包含了几部分:

- SSTableScanner (sstable/ob_sstable_scanner.cpp)：实现 ObSSTableScan 的接口，读 SSTable
- SSTableGetter(sstable/ob_sstable_getter.cpp)：实现读单条数据
- SSTableWriter (sstable/ob_sstable_writer.cpp)：生成 sstable 文件，合并的时候写 SSTable 文件
- BlockCache, BlockIndexCache (sstable/ob_blockcache.cpp，sstable/ob_block_index_cache.cpp)：分别缓存 SSTable block, SSTable index

说明:*ob0.4 以前的版本都是基于 cell 接口的，每次迭代一个 cell, 而新的 ob0.4 采用的行接口 get_next_row。新的接口是以 physical operator 的形式出现，针对老版本(v1,v2)SSTable 的读操作的行接口都实现在 sql 目录中。新版本的 SSTable 读操作实现在 compactsstablev2 中。*

SSTable 读写的代码都与 SSTable 的具体格式相关，阅读代码之前必须熟读相关的格式文档：

- Sstable_format_and_SstableWriter_interface.doc (v1&v2)
- OceanBase 紧凑型格式 SSTable 写入详细设计.doc (v3)
- OceanBase 紧凑型格式 SSTable 读详细设计.doc (v3)

接“3.8 ChunkServer”的流程：

```
ObTabletScan::create_plan
    |→ ObSSTableScan::open_scan_context
ObTabletScan::open
    |→ ObTabletScanFuse::open
            |→ ObSSTableScan::open
                    |→ ObSSTableScan::init_sstable_scanner    //这里会根据不同的版本实现不同的 SSTable scanner。
```

3.10 RootServer

rootserver 管理 root table，提供 root table 查询功能：

- 管理各个 Server 的上下线，对 UPS 进行选主操作
- 维护内部表
- Tablet 的复制与迁移
- Tablet 的合并
- 每日合并过程控制

参考文档：

- RootServer 启动注意事项.docx
- RS 状态转换及 Tablet 合并与核对.docx
- OceanBase 内部表定义.docx

3.11 UpdateServer

UpdateServer 是更新数据服务器，实现了多索引(btree,hash)的 MemTable 更新与读取、redo log 机制、事务管理机制。

参考文档：updateserver 概述.ppt