



OceanBase 0.4.2

客户端 用户指南

文档版本: Beta 02

发布日期: 2013.12.30

支付宝（中国）网络技术有限公司·OceanBase 团队

前言

概述

本文档介绍OceanBase 0.4.2的客户端的架构和使用方法。

读者对象

本文档主要适用于：

- 阿里内部开发工程师。
- 阿里内部安装工程师。

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（2013-12-30）	第一次发布Beta版本，适用于OceanBase 0.4.2。

版本和发布日期	说明
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 概述	- 1 -
1.1 客户端简介	- 1 -
1.2 获取安装包	- 1 -
2 OceanBase Java 客户端	- 3 -
2.1 客户端结构	- 3 -
2.2 功能模块	- 5 -
2.2.1 “oceanbase.jar”模块	- 5 -
2.2.2 “oceanbase-core.jar”模块	- 7 -
2.3 访问方式	- 15 -
2.3.1 使用“OB Configure + oceanbase.jar”场景访问 OceanBase	- 15 -
2.3.2 使用“oceanbase-core.jar”场景访问 OceanBase	- 19 -
2.3.3 使用 MergeServer 直接访问 OceanBase	- 21 -
3 OceanBase C 客户端	- 25 -
3.1 客户端结构	- 25 -
3.2 访问流程	- 26 -
3.3 安装	- 26 -
3.4 访问方式	- 28 -
4 附录	- 32 -
4.1 监控系统采集引擎计算规则	- 32 -
4.1.1 表达式求值计算	- 32 -
4.1.2 表达式对比计算	- 33 -
4.1.3 采集周期内平均值	- 36 -
4.1.4 无计算	- 41 -
4.2 数据展示计算规则	- 43 -
4.3 OceanBase Java 客户端流量分配执行实现细节	- 51 -
4.4 OceanBase Java 客户端定时 MergeServer 重新生效流程	- 53 -
4.5 OceanBase Java 客户端日志梳理	- 54 -

4.6 OceanBase C 客户端流量分配设计.....	- 54 -
4.7 OceanBase C 客户端重试设计.....	- 55 -

1 概述

主要介绍客户端的基础知识以及安装包的获取方法。

1.1 客户端简介

OceanBase 客户端主要用于开发人员编程时连接 OceanBase 数据库。

Oceanbase 内置了对 SQL 的支持，用户可以通过 libmysql, JDBC 等方式直接访问 Oceanbase，但由于 OceanBase 是一个分布式数据库，可以由多个节点（MergeServer）同时提供 SQL 服务。而 MySQL 客户端等都是针对单机系统，在连接 OceanBase 时，客户端会绑定其中一台 MergeServer 进行 SQL 操作，而不能有效利用其他 MergeServer 资源。

为了实现了多集群间流量分配和多 MergeServer 间的负载均衡，并给应用开发人员提供一个简单接入方案，我们在 libmysql, JDBC 的基础上封装一个 OceanBase 客户端。

OceanBase 客户端可以根据“流量分配”选择 MergeServer。例如：主集群比例为 40，备集群比例为 60。那么在发送弱一致性读 SQL 语句时，将会有 40% 的概率发送至主集群中的 MergeServer，60% 的概率发送至备集群中的 MergeServer。

流量分配只涉及弱一致性读请求，所有非弱一致性读请求全部发送至主集群。

说明：

- 流量分配：主要指弱一致性读在主备集群中的分配比例。可以在内部表“__all_cluster”中设置。
- 弱一致性读请求包括：hint weak sql、无 hint 的 select，除此之外的 SQL 语句类型为非弱一致性读请求。

OceanBase 客户端主要有以下两种：

- OceanBase C 客户端
为应用程序提供了“libobsql.so”动态库，这个库在二进制接口上与 mysql 的 libmysqlclient 库完全兼容。
- OceanBase Java 客户端
提供了符合 Java 标准的 DataSource，Java 应用程序可以使用 OceanBase Java 客户端获得与 OceanBase 服务器交互的连接。

1.2 获取安装包

OceanBase 客户端安装包的获取方式和说明如[表 1-1](#)所示。

说明： 本文档中使用的安装包版本仅为举例，实际请采用最新安装包。

表 1-1 安装包

类型	安装包	获取地址
OceanBase C 客户端	<p>Linux 版本为 RedHat 5 的安装包：</p> <ul style="list-style-type: none">• curl-7.29.0-1.el5.x86_64.rpm• oceanbase-devel-0.4.2.1-1193.el5.x86_64.rpm <p>Linux 版本为 RedHat 6 的安装包：</p> <ul style="list-style-type: none">• curl-7.29.0-1.el6.x86_64.rpm• oceanbase-devel-0.4.2.1-1193.el6.x86_64.rpm <p>说明： 您可以执行 <code>cat /etc/issue</code> 命令查看 Linux 版本号。</p>	<p>https://github.com/alibaba/oceanbase_client 的“C”文件夹中。</p>
OceanBase Java 客户端	<p>jar 包：</p> <ul style="list-style-type: none">• commons-lang-2.3.jar• commons-logging-1.1.jar• druid-0.2.12.jar• mysql-connector-java-5.1.14.jar• oceanbase-1.0.1.jar• oceanbase-core-1.1.0.jar <p>“oceanbase-core.jar”的源码： oceanbase_core_src</p>	<p>https://github.com/alibaba/oceanbase_client 的“Java”文件夹中。</p>

2 OceanBase Java 客户端

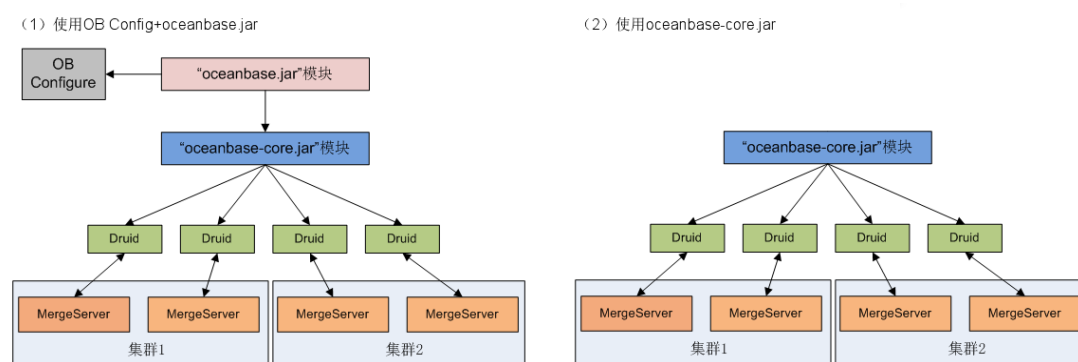
OceanBase Java 客户端主要用于开发人员编写的 Java 程序连接 OceanBase。

2.1 客户端结构

连接 OceanBase 主要有“使用 OB Configure + oceanbase.jar”、“使用 oceanbase-core.jar”和“直接通过 MergeServer 连接”三种场景。其中“直接通过 MergeServer 连接”无需使用 OceanBase 客户端，详细请参见“2.3.3 使用 MergeServer 直接访问 OceanBase”。

OceanBase Java 客户端的结构示意图如[图 2-1](#)所示。

图 2-1 结构示意图



“使用 OB Configure + oceanbase.jar”和“使用 oceanbase-core.jar”两种场景的区别如下：

- “使用 OB Configure + oceanbase.jar”可以自动升级或回滚客户端。
- “使用 oceanbase-core.jar”需要通过手动修改 pom 文件进行升级或回滚客户端。

OceanBase Java 客户端按功能主要可以分为两种使用场景，如[表 2-1](#)所示。

表 2-1 场景说明

模块	说明	是否 在 “OB Configure + oceanbase.jar”场景中使用	是否 在 “oceanbase-core.jar”场景中使用
OB Configure	OceanBase 配置管理中心。OB Configure 保存了所有的 OceanBase Java 客户端的配置参数，包括：数据库集群地址、用户名、密码、以及数据源本身的配置项。其主要功能为：配置信息的集中化管理与操作，简化 DBA 操作流程，同时方便 OceanbBase Java 客户端的升级。详细请参见《OceanBase 0.4.2 配置中心 安装指南》。	是	否
“oceanbase.jar”模块	OceanBase Java 客户端的主要模块之一，使用该模块时需要部署 OB Configure。主要负责集中化管理各个 Server 的配置项，初始化加载 “oceanbase-core.jar”模块，以及完成 “oceanbase-core.jar”模块的版本升级任务。	是	否
“oceanbase-core.jar”模块	OceanBase Java 客户端的主要模块之一，主要封装了 OceanBase 的流量分配、负载均衡的功能。 在使用“oceanbase.jar”模块时，“oceanbase-core.jar”模块从 OB Configure 中自动缓存到本地。	是	是
Druid	数据源模块，是阿里巴巴开发的开源的数据库连接池。每个 Druid 数据源对应一个 MergeServer，用于维护到 MergeServer 的连接。关于 Druid 详细介绍请参见 “https://github.com/alibaba/druid” 。	是	是
MergeSever	OceanBase 中负责提供 SQL 服务的接口模块，用于处理客户端发送的 SQL 请求。	是	是

2.2 功能模块

OceanBase Java 客户端的功能模块主要包括“oceanbase.jar”、“oceanbase-core.jar”和“OB Config”。

2.2.1 “oceanbase.jar”模块

“oceanbase.jar”模块提供服务的接口是“OceanBaseDataSourceProxy”。用户通过使用此接口获取数据库的 Connection，进而完成业务的 SQL 执行。

* 配置中心交互

“oceanbase.jar”模块会定期与 OB Config 交互，对比本地与 OB Config 上“oceanbase-core.jar”模块的版本以及升级配置项，从而确认是否需要进行升级。

用户调用“OceanBaseDataSourceProxy”的接口方法“setConfigURL(String configURL)”传入配置中心地址，获取配置中心信息：

- 集群入口地址（Listener MergeServer）。
- 被托管的“oceanbase-core.jar”模块的 jar 包版本号，并缓存该模块文件。
- 集群帐号信息。

获取集群入口地址后，OceanBase Java 客户端和内部表交互，详细流程请参考“2.2.2 “oceanbase-core.jar”模块”中的“集群内部表交互”。

* 本地缓存

通过配置中心交互模块，获取集群配置中心信息以及“oceanbase-core.jar”模块后，对以下两个信息进行缓存：

- “oceanbase-core.jar”模块
缓存文件为“\$USER_HOME/.obdatasource/\$version”，其中
“\$USER_HOME”为本地用户路径，“\$version”为文件名（版本号）。每一个版本对应都会在本地图缓存一个对应的缓存。这意味着，每一个版本被第一次使用时，都会在本地图缓存一份；再次使用时，将不会再下载。在第一次初始化 Java 应用程序时，“oceanbase-core.jar”模块文件“\$version”从 OB Config 缓存到本地，而不需要手动导入“oceanbase-core-XXX.jar”包。
- 配置信息
缓存文件为“\$USER_HOME/.obdatasource/conf”。配置信息每分钟均随定时任务运行而被更新。

本地缓存模块主要是防止在配置中心故障时，导致业务系统无法运行而影响业务的可用性。同时，能够减少对配置中心的访问，在一定程度上，能够缓减配置中心的压力。

* 加载核心服务

将传入的 OceanBase 密码进行解密，解密完成后，将集群入口地址（Listener MergeServer）、用户名以及密码传入“oceanbase-core.jar”模块，并最终调用“ObGroupDataSource”接口类的 init() 方法，完成核心模块的初始化工作，详情请参见“2.2.1 “oceanbase-core.jar”模块”。

* 自动升级

默认启动升级检测线程，调度周期为 1 分钟。通过扫描配置中心，查看是否已经开启了升级计划。如果开启，则启动升级任务，并完成“oceanbase-core.jar”模块的动态升级。该任务主要为了解决：在不中断业务服务的前提下，完成修复紧急 bug。

自动升级模式主要有以下几种：

- 按白名单升级
在 OceanBase 配置中心设置指定需要升级的客户端，其他客户端不升级。
例如：某业务系统集群的客户端机器列表为“10.10.10.1, 10.10.10.2, 10.10.10.3, 10.10.10.4”。在 OceanBase 配置中心设置白名单为：10.10.10.1，则单独为 10.10.10.1 升级。
- 按比例升级
在 OceanBase 配置中心设置设置集群升级的比例，完成该占比机器的升级。
例如：某业务系统集群的客户端机器列表为“10.10.10.1, 10.10.10.2, 10.10.10.3, 10.10.10.4”。在 OceanBase 配置中心设置升级比例为 50%，则将为 4 台客户端中的 2 台完成升级（理论值）。
同时，可以通过逐渐变化升级的比例。比如第一次设置 25%，完成集群的四分之一升级；然后通过设置比例为 50%，完成集群的二分之一升级；通过逐步提升，到最终完成集群的所有机器升级至指定版本。
- 回滚
对于出现的升级错误，将采用回滚操作来完成版本的还原。

“oceanbase-core.jar”模块每次升级时，原数据源和新数据源同时存在，原数据源在 5 分钟后失效。例如：“oceanbase-core.jar”模块从 A 版本升级到 B 版本。

- 5 分钟内，A 版本和 B 版本的数据源同时存在，此时，“oceanbase-core.jar”模块可以直接回滚到版本 A，不能够升级到其他版本。
- 5 分钟后，A 版本失效，支持升级到其他版本。如需回滚到 A 版本，则视为从 B 版本“升级”到 A 版本。

* 客户端汇报

在 Java 应用程序第一次初始化时，将汇报客户端机器的 IP 以及所使用的“oceanbase-core.jar”的版本号。如果“oceanbase-core.jar”升级，则重新汇报。

* 日志规则初始化

初始化“log4j”组件规则，便于日志的输出。默认日志输出路径为“\$USER_HOME/logs”。

2.2.2 “oceanbase-core.jar”模块

“oceanbase-core.jar”封装了 OceanBase 数据库的流量分配、负载均衡的功能，本章节主要介绍“oceanbase-core.jar”模块的功能和实现。

* 集群内部表交互

“oceanbase-core.jar”模块与 OceanBase 集群交互的内部表主要为“__all_cluster”和“__all_server”，如[表 2-2](#)和[表 2-3](#)所示。

表 2-2 __all_cluster

field	type	nullable	key	default	extra
gm_create	createtime	-	0	NULL	-
gm_modify	modifytime	-	0	NULL	-
cluster_id	int	-	1	NULL	-
cluster_vip	varchar(32)	-	0	NULL	-
cluster_port	int	-	0	NULL	-
cluster_role	int	-	0	NULL	-
cluster_name	varchar(128)	-	0	NULL	-
cluster_info	varchar(128)	-	0	NULL	-
cluster_flow_percent	int	-	0	NULL	-

field	type	nullable	key	default	extra
read_strategy	int	-	0	NULL	-

注：“-”表示无。

“__all_cluster”中决定 OceanBase Java 客户端行为的字段如下：

- cluster_vip 和 cluster_port: 集群入口 Lms 访问地址。
- cluster_role: “1”为主集群，“2”为备集群。
- cluster_flow_percent: 读流量分配占比。
- read_strategy: 隶属于该集群的多个 MergeServer 负载策略，“0”为轮询，“1”为一致性 Hash，“2”为随机。

表 2-3 __all_server

field	type	nullable	key	default	extra
gm_create	createtime	-	0	NULL	-
gm_modify	modifytime	-	0	NULL	-
cluster_id	int	-	1	NULL	-
svr_type	varchar(16)	-	2	NULL	-
svr_ip	varchar(32)	-	3	NULL	-
svr_port	int	-	4	NULL	-
inner_port	int	-	0	NULL	-
svr_role	int	-	0	NULL	-
svr_version	varchar(64)	-	0	NULL	-

注：“-”表示无。

“__all_server”中决定 OceanBase Java 客户端行为的字段如下：

- svr_type: 只查询值为“mergeserver”的记录。

- cluster_id: 集群 id 与__all_cluster 表关联。
- svr_ip, svr_port: MergeServer 访问地址。

OceanBase Java 客户端和内部表互交流程如下：

1. 传入 OceanBase 集群集群的 Listener 入口地址。

说明： 如果“使用 oceanbase-core.jar”场景，则直接提供集群入口地址（Listener MergeServer）；
如果“使用 OB Configure + oceanbase.jar”场景，则提供配置中心的地址，当配置中心交互模块向 OB Configure 传入 configUrl，访问配置中心，获取集群入口地址时，如果集群不可用，则使用本地缓存文件。

2. 将传入的地址按照“,”分裂成多个入口地址（一般是两个）。根据第一个地址访问“__all_cluster”表，并执行“select cluster_vip, cluster_port from __all_cluster where cluster_role = 1”获取主集群的“ip”以及“port”。

如果第一个入口地址不可用，则重试下一个入口地址，依次类推。一般情况下，配置两个入口地址，因此当第二个入口地址不可用时，启动失败。

3. 执行“select cluster_id, cluster_role, cluster_flow_percent, cluster_vip, cluster_port, read_strategy from __all_cluster”获取集群流量分配和该集群的多 MergeServer 负载策略。
4. 根据“步骤 3”中查询到的“cluster_id”，再依次执行“select svr_ip, svr_port from __all_server where svr_type = 'mergeserver' and cluster_id=?”查询所有 MergeServer 信息。
5. 根据 MergeServer 的信息（ip、port）初始化这台 MergeServer 的真实连接池（Druid），默认连接池属性是“maxActive:20,minIdle:1,initialSize:0”。
6. 执行“show variables like 'ob_read_consistency’”，查询集群全局的默认读策略。

* 集群流量分配

集群流量分配只针对所有读服务（即 select 操作），同时综合考虑主集群的写执行量。在实现上，OceanBase Java 客户端根据内部表“__all_cluster”中的“cluster_flow_percent”字段值，构造流量分配使用的数组。默认使用的数组长度是 100，按照“cluster_flow_percent”配置的比例来填充数组的每一个元素，最后随机打散数组。

执行时，每次新请求的 getConnection()，会跟着一个全局唯一的 id，根据 id 定位使用的数组元素下标，以确定使用的集群是哪个。

另外，给主集群单独分配一个全局计数器，每次写操作的 getConnection()，会给计数器执行“+1”操作。当流量分配命中主集群而主集群的计数器大于零时，则重新进行分配，同时给计数器执行“-1”操作，直至分配备机群或者计数器为不大于零。

异常情况下的考虑：如果主集群故障，所有写请求仍然会分配至主集群，所以写请求会执行失败；而此时读请求将分配至备集群，如果备集群正常，则读请求完成执行。相反，如果备集群故障，则读写均不影响。如果主备集群都故障，则读写请求均失败。

目前我们实现的流量分配并不能做到精确的流量分配。在一定的使用场景中分配将无法按照我们实际配置的进行分配。例如：在单个事务中，执行不同数量的 `select`，因为我们只会在每次 `getConnection` 时，进行分配，而之后在该 `connection` 上执行的所有 `SQL` 无法进行分配。关于这点，后续我们将会改进。

* 负载策略

当 `SQL` 请求经过集群分配之后，就流入到某个具体的集群。在同一集群中的 `MergeServer` 选择方式由内部表“`__all_cluster`”中的“`read_strategy`”字段值决定：“0”表示轮询策略，“1”表示一致性策略，“2”表示随机策略。

- 轮询

在默认情况下采用“轮询”策略。该策略为每个申请获取连接的线程分配一个计数器，根据线程计数器变量的值对 `MergeServer` 个数求余得到下标，并定位到具体数组元素。所以，这种实现是在单个线程之中，对经过流量分配之后的集群中的多 `MergeServer` 进行逐个访问。

异常情况分析：当某个 `MergeServer` 被判定为不可用时（详细请参见“* `MergeServer` 不可用判定”），轮询策略将如何工作？

在该算法内部，每次轮询到的 `MergeServer` 首先会判定是否已经处于不可用列表之中。如果不存在，则判定该 `MergeServer` 是否已经不可用，如果已经不可用，则重新选择。而重新选择则需要判定选择的次数是否大于 `MergeServer` 个数。如果都为否，则一直选择直到选择最后一个 `MergeServer` 为止。

- 一致性

一致性散列负载策略，主要是为了更好的利用 `MergeServer cache`，提升单节点 `MergeServer` 的内存 `cache` 命中率。

说明：MergeServer cache 的主要作用为缓存 SQL 操作记录，提高下次读取数据的效率。

1. 准备构造排序映射表。每个 `MergeServer` 会循环存放 100 个，存放规则是：用 `MergeServer` 的 `ip`、`port` 和循环变量 `i`，通过 `Hash` 算法产生一个 `hashcode` 作为映射表的 `key`，而 `value` 则是这个 `MergeServer`。
2. 在每次进行 `MergeServer` 选择前，对 `SQL` 进行还原，将占位符替换为具体的值。再对还原后的 `SQL` 进行 `Hash` 算法取值得到 `hashcode`。
3. 根据 `hashcode` 在排序映射表中寻找大于等于该值的最小值，并最终选择该值对应的 `MergeServer`。

异常情况分析：当某个 `MergeServer` 被判定为不可用时（详细请参见“* `MergeServer` 不可用判定”），一致性策略将如何工作？

在该算法内部,通过 SQL 散列后产生的 `hashcode` 定位出的 `MergeServer`,如果为不可用,会进行重试一次,重试的机制是在 `hashcode` 上采取循环加一(次数不超过 `MergeServer` 总个数),然后再去排序映射表中寻找大于等于该值的最小值,如果不是上次选择的 `MergeServer`,则选中并返回该 `MergeServer`。

- 随机

随机负载实现方式:产生一个在 0 和 `MergeServer` 个数减一之间的随机数,并通过此随机数定位到列表中的 `MergeServer`。

关于随机算法的详细设计,详细请参见“The Art of Computer Programming (Donald Knuth)”中的“Section 3.2.1”。

由于性能方面的考虑,在线程中绑定随机算法实例。详细实现方法请参考 Doug Lea 的 `ThreadLocalRandom`。

异常情况分析:当某个 `MergeServer` 被判定为不可用时(详细请参见“* `MergeServer` 不可用判定”),随机策略将如何工作?

在该算法内部,每次随机出的 `MergeServer` 首先会判定是否已经处于不可用列表之中。如果不存在,则进行下一步判定,判定此 `MergeServer` 是否已经不可用,如果已经不可用,则重新选择,而重新选择则需要判定选择的次数是否大于 `MergeServer` 个数。如果都为否,则一直选择直到选择最后一个 `MergeServer` 为止。

* `MergeServer` 不可用判定

“`MergeServer` 不可用判定”主要是对 `MergeServer` 的可用性进行判定,对于判定为不可用的 `MergeServer` 会进行状态标识,同时该标识会被负载策略所调用。也就是说,在做负载策略选择 `MergeServer` 时,如果发现该 `MergeServer` 状态标识为“不可用”,则需要重新选择,详细请参见“* 负载策略”。

判定 `MergeServer` 故障方法如下:

当某个 `MergeServer` 在一分钟内的异常次数大于 47.63 次时,则判定该 `MergeServer` 为不可用。“不可用”状态在 1 分钟后失效,即再次进入可用状态。判定一台 `MergeServer` 故障后,在其他可用的 `MergeServer` 中进行重新选择,直至选择到正常的 `MergeServer`。如果该集群已经没有可用的 `MergeServer`,则抛出异常。

`OceanBase` 数据库异常主要有以下几种:

- 连接类异常

客户端和 `OceanBase` 建立底层 TCP 连接时,发生的异常。如:数据库主机故障、数据库服务故障、机房(网络设备)故障等。

- 特殊类异常

客户端和 `OceanBase` 能够建立 TCP 连接,但是由于 `OceanBase` 发生内部错误,导致在应用层所建立的连接失败。如: no memory, login error。

- 执行类异常
客户端和 OceanBase 正确建立连接之后，在执行 SQL 语句时发生的异常。如：SQL 语法异常、无执行权限等。

OceanBase 针对以上的三种异常类型，明确了三种不同的处理逻辑。

- 对于连接类异常，根据执行的 SQL 的类型以及集群状况来选择是否需要重试。
- 对于特殊类异常，将通知物理连接池回收掉腐化的连接。
- 对于执行类异常，不执行任何操作。

目前，我们只针对连接类异常才判定为 fatal 异常，计入 MergeServer 不可用次数之中。fatal 异常判定方法如下：

1. 查看异常信息是否为 SQLException，如果是，则继续“步骤 2”。
2. 获取该异常的 sqlState 跟 errorcode。
 - 如果 sqlState 以“08”开始，则为 fatal。
 - 如果 errorcode 为 1040、1042、1043、1047、1081、1129、1130、1045、1004、1005、1015、1021、1041、1037、1038，则为 fatal。
3. 对异常的描述进行关键字定位，如果包含：“COMMUNICATIONS LINK FAILURE”、“COULD NOT CREATE CONNECTION”、“NO DATASOURCE”、“NO ALIVE DATASOURCE”则为 fatal。

* 集群不可用判定

查看该集群下的所有 MergeServer 是否都为故障。如果全部故障，则判定该集群为故障。当请求为弱一致性读请求时，主集群故障时，会重选集群，将请求发送至备集群。否则，则抛出异常。

* 定时任务

定时任务主要包括“内部表扫描任务”和“日志输出任务”。

内部表扫描任务的调度周期为 60s-120s 之间的任意时刻。选择在 60-120s 之间的随机分布，主要是为了解决应用集中在同一时刻对配置中心进行访问，导致配置中心产生比较大的压力。

内部表扫描任务过程如下：

1. 与内部表交互，详细请参见“集群内部表交互”。
2. 将这次调度执行期间获取的内部表状态与客户端当前内部状态进行比较，对于发生的差异，将促发不同的变更动作。
 - 如果是集群信息发生变化，例如：主备集群角色发生变化、主备集群的 ip 和 port、集群的负载策略等发生变化，则重新初始化数据源。

- 如果是集群的流量比发生变化，则重新构造流量分配数组。
- 集群中的 MergeServer 上下线，则创建或销毁 MergeServer 对应的数据源。

说明：该任务在执行管理器执行 SQL 的过程中，如果发生 **fatal** 异常，同样会被触发，但是，触发周期被控制在 10s 之内。也就是说，10s 内执行管理器只能触发一次。

日志输出任务调度周期为：30s。主要对所有的 MergeServer 的数据源的状态进行输出。目前日志输出格式如下：

```
10.209.144.37-2880[2013-07-09 16-28-11];5;0;0;41;0;38;3;5;1;0;0;0;32;1;278;276;0;0;268;10
10.209.144.38-2880[2013-07-09 16-28-11];7;2;0;35;0;34;1;7;1;0;0;0;32;1;258;258;0;0;244;14
10.209.144.39-2880[2013-07-09 16-28-11];4;0;0;32;0;30;2;4;2;0;0;0;32;1;221;221;0;0;215;8
```

日志输出任务附件说明如下：

```
Name;CreateCount;DestroyCount;CreateErrorCount;ConnectCount;ConnectErrorCount;CloseCount;ActiveCount;
ActivePeak;PoolingCount;LockQueueLength;WaitThreadCount;InitialSize;MaxActive;MinIdle;StartTransactionCount;CommitCount;ErrorCount;CachedPreparedStatementHitCount;CachedPreparedStatementMissCount
```

后续，我们会对客户端的日志进行统一的收集处理，由支付宝监控系统协助完成。这样，可以对客户端的各个真实的数据源进行 30s 的一个准实时监控，更好的帮助监控和定位 Server 产生的问题。

* 执行管理器

执行管理器的主要作用是第一次在虚拟 Connection 上执行 SQL 时，获取一条真实的 Connection，并成功完成整个执行过程。所以它主要管理流量分配以及集群的负载策略，和组合这两种算法的执行过程。

1. 调用流量分配，完成集群的选择，详细请参见“* 集群流量分配”。
2. 调用集群的负载策略分配器，选择 MergeServer，详细请参见“* 负载策略”。

在 MergeServer 执行 SQL 过程中，如果发生了异常，对异常进行判断。

- 如果是 **fatal** 异常，则该 MergeServer 不可用状态计一次，并且再次选择一台 MergeServer 执行。同时激活后台调度线程（激活周期为 10 秒），对“__all_server”表进行扫描，并完成相关操作，详细请参见“* 定时任务”。
- 如果非 **fatal** 异常，则抛出异常。

* JDBC 规范

OceanBase Datasource 严格按照 JDBC 规范进行设计，下面对实现 JDBC 规范的每个接口逐一进行说明。

com.alipay.oceanbase.OBGroupDataSource	-> javax.sql.DataSource
com.alipay.oceanbase.TGroupConnection	-> java.sql.Connection
com.alipay.oceanbase.TGroupStatement	-> java.sql.Statement
com.alipay.oceanbase.TGroupPreparedStatement	-> java.sql.PreparedStatement

- **com.alipay.oceanbase.OBGroupDataSource**
对外提供的主要方法有：**setConfigUrl(String configUrl)**、**setUserName(String userName)**、**setPassword(String password)**、**init()**、**destroy()**和**getConnection()**等。
以上方法可以总结为：参数输入方法；初始化、销毁方法；获取连接方法。
其中，“获取连接”返回的是一个虚拟连接（即 **TGroupConnection**），没有与 **Server** 产生任何真实的连接。
- **com.alipay.oceanbase.TGroupConnection**
对外提供的主要方法有：**createStatement()**和 **preparedSatement(String sql)**，而对应的返回分别是 **TGroupStatement** 和 **TGroupPrepreadStatement**。
同时，此实例会维护真实的连接。当第一次 **execute** 时，会通过执行管理器分配真实的连接，并且完成执行过程。如果此实例的 **close** 没有被调用，则一直保持这条真实连接，并且等待下一次通过此条真实连接执行 **SQL**。目前，通过虚拟连接无法做到真正意义上的读写分离。即读操作完全按照读流量分配执行，写操作则在主集群的 **MergeServer** 上执行。
- **com.alipay.oceanbase.TGroupStatement**
对外提供的主要方法是：任何类型的 **execute()**方法。
当执行 **execute** 方法时候，会检测此次执行的 **SQL**，判断是哪种类型，如 **SELECT**、**INSERT** 等，并调用虚拟连接 **TGroupConnection**，查看是否已经建立连接。如果已经建立，则直接调用执行。如果是第一次执行，则回调 **TGroupDataSource** 中的执行管理器，进行连接分配，然后再执行。
- **com.alipay.oceanbase.TGroupPreparedStatement**
对外提供的主要方法是：**execute()**、**executeQuery()**、**executeUpdate()**等。
当执行这些方法时，会检测此次执行的 **SQL**，调用虚拟连接 **TGroupConnection**，查看是否已经建立连接。如果已经建立连接，则直接执行 **SQL**。如果是第一次执行，则回调 **TGroupDataSource** 中的执行管理器，进行连接分配，然后再执行 **SQL**。

2.3 访问方式

主要介绍 Java 应用程序访问 OceanBase 的三种方式。

2.3.1 使用“OB Configure + oceanbase.jar”场景访问 OceanBase

“oceanbase.jar”模块提供服务的 jar 包为“oceanbase-XXX.jar”，其中“XXX”为版本号。

* 基本信息

在使用“oceanbase.jar”模块访问 OceanBase 时，Java 程序需要初始化“oceanbase.jar”模块。在初始化过程中，“oceanbase.jar”模块会完成如下几件事情：

1. “oceanbase.jar”模块访问 OB Config 配置中心，从配置中心获取 OceanBase 数据库中 Listener 的位置信息，同时 OB Config 也会返回“oceanbase-core.jar”模块的最新版本号、是否允许进行主动升级、升级的机器比例、升级白名单等信息。
2. “oceanbase.jar”模块根据这些信息来判断是否要主动升级“oceanbase-core.jar”模块的版本。如果需要升级，则会将“oceanbase-core.jar”模块的新版本缓存到 Java 程序的本地。
3. “oceanbase.jar”模块访问 OceanBase 数据库中 Listener，从而获取所有可供使用的 MergeServer 的地址列表和流量分配信息，并为每个 MergeServer 建立一个 Druid 数据源。
4. “oceanbase.jar”模块根据流量分配的原则，从一个 Druid 数据源中选择一个连接给 Java 程序使用。等到应用断开连接后，该连接重新回到 Druid 数据源中等待下一次使用。

说明：Listener 是 OceanBase 集群内一个特殊的 MergeServer，主要用来提供获取 MergeServer 列表、集群流量分配的进程。

目前在 taobao、以及 alipay 的 maven 仓库均可搜索获得发布的客户端。pom 坐标如下：

```
<dependency>
  <groupId>com.alipay.oceanbase</groupId>
  <artifactId>oceanbase</artifactId>
  <version>1.1.1</version>
</dependency>
```

目前 druid 的最新版本为 0.2.25。pom 坐标如下：

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>0.2.25</version>
</dependency>
```

说明：如果使用的是 **taobao** 的仓库，只需要添加 **oceanbase** 自身的 **pom** 信息；如果使用的是 **alipay** 的仓库，需要再添加 **druid** 的 **pom** 信息。

MYSQL 的驱动版本为 **5.1.14**。**pom** 坐标如下：

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.14</version>
</dependency>
```

添加完 **pom** 信息之后，就添加了工程的依赖关系。那么下面可以进行编码工作了。现在，编码工作主要分为以下两种方式：

- 配置编码

通过 **id** 为 **dataSource**，向 **Spring IOC** 容器获取配置的 **bean** 后，即可通过调用方法 **getConnection()** 来使用 **OceanBase Java** 客户端。

```
<bean id="dataSource" class="com.alipay.oceanbase.OceanbaseDataSourceProxy"
init-method="init" destroy-method="destroy" />
<property
name="configURL">http://obconsole.test.alibaba-inc.com/ob-config/config.co?dataId=fi
nancial_history</property>
</bean>
```

- 硬编码

通过直接在程序中添加以下语句使用 **OceanBase Java** 客户端。

```
OceanbaseDataSourceProxy obDS = new OceanbaseDataSourceProxy();
obDS.setConfigURL("http://obconsole.test.alibaba-inc.com/ob-config/config.co?dataId=
financial_history");
obDS.init();
Connection conn = obDS.getConnection();
```

configURL 可以分为前缀部分与 **dataId** 部分。例如：**http://obconsole.test.aliba**
ba-inc.com/ob-config/config.co?dataId=financial_history。

- 前缀部分，可以分为以下几种情况：

- 线下环境：
http://obconsole.test.alibaba-inc.com/ob-config/config.co?dataId=

- 线上环境（集团）：
http://obconfigserver.tbsite.net/ob-config/config.co?dataId=
- 线上环境（支付宝）：
http://obconfigserver.db.alipay.com/ob-config/config.co?dataId=
- dataId 部分，由 DBA 操作完成。

在 Druid 数据源内部我们已经进行了配置参数优化，因此，除以下参数外，不建议配置其他的参数项：

```
<!-- Druid 与 MergeServer 的连接。Druid 初始化时的连接数、最小连接数和最大连接数 -->
  <property name="initialSize" value="1" />
  <property name="minIdle" value="1" />
  <property name="maxActive" value="20" />
<!-- 建立一条 Druid 与 MergeServer 新连接的超时时间 -->
  <property name="maxWait" value="60000" />
<!-- SQL 操作 Prepare 语句缓存功能的开关。 -->
  <property name="poolPreparedStatements" value="true" />
<!-- Prepare 语句缓存最多支持缓存的 SQL 语句数量。 -->
  <property name="maxPoolPreparedStatementPerConnectionSize" value="20"/>
```

* 前提条件

- OceanBase 主备集群已经安装且正常运行。
- OB Config 搭建、配置完成且正常运行，详细请参见《OceanBase 0.4.2 配置中心 安装指南》。
- Java 开发环境以及部署完成，如已经安装 JDK、Eclipse 等。
- 以添加 “oceanbase-1.1.1.jar”、“druid-0.2.25.jar”、“mysql-connector-java-5.1.14.jar” 的 pom 坐标，或者已获取这三个 jar 包。
- 已经获取 “commons-logging-1.1.jar” 和 “commons-lang-2.3.jar”。

* 操作流程

1. 连接 OceanBase 数据库。
2. 新建 PreparedStatement 语句，来查询__all_server 表中 svr_port 为某个值的 server 信息。
3. 指定查询__all_server 表中 svr_port 为 2600 的 server 信息。
4. 打印查询结果。

* 预期结果

Eclipse Console 界面将打印出所有端口号为 2600 的 server 信息。

* 代码示例

编辑 Java 代码，如下所示：

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;

public class OceanBaseDataSourceProxyExample {
    public static void main(String[] args) throws Exception {
        useDataSource();
        System.exit(0);
    }

    public static void useDataSource() throws Exception {
        OceanbaseDataSourceProxy datasource = new
OceanbaseDataSourceProxy();

        datasource.setConfigURL("http://obconsole.test.alibaba-inc.com/ob-config/config.co?
dataId=alipay_junbian");

        datasource.setUsername("admin");
        datasource.setPassword("admin");
        datasource.init();
        Connection conn = datasource.getConnection();
        PreparedStatement pstmt = conn
            .prepareStatement("select * from __all_server where
svr_port = ?");

        pstmt.setInt(1, 2600);
        ResultSet rs = pstmt.executeQuery();
        ResultSetMetaData rSetMetaData = rs.getMetaData();
        for (int i = 1; i <= rSetMetaData.getColumnCount(); i++) {
            System.out.print("|" + rSetMetaData.getColumnName(i) + "");
        }
        System.out.println();
        while (rs.next()) {
            for (int i = 1; i <= rSetMetaData.getColumnCount(); i++) {
                System.out.print("|" + rs.getString(i) + "");
            }
        }
    }
}
```

```

        System.out.println();
    }
    if (rs != null) {
        rs.close();
    }
    if (pstmt != null) {
        pstmt.close();
    }
    if (conn != null) {
        conn.close();
    }
    datasource.destroy();
}
}

```

如果 Eclipse Console 界面输出如下信息则表明运行正常。其中第一行为“__all_server”表的 schema 信息，而第二行才是真正的输出结果行。

```

|gm_create|gm_modify|cluster_id|svr_type|svr_ip|svr_port|inner_port|svr_role|svr_version
|2013-05-27 21:54:30.08001|2013-05-29
17:54:45.81237|0|chunkserver|10.209.144.31|2600|0|0|0.4.1.2_13189M(Apr 10 2013
12:40:32)

```

2.3.2 使用“oceanbase-core.jar”场景访问 OceanBase

OceanBase Java 客户端的“oceanbase-core.jar”模块对外提供服务的接口为“OBGroupDataSource”。

“oceanbase-core.jar”模块的 jar 包形式为:“oceanbase-core-XXX.jar”，其依赖的包为“druid-XXX.jar”，其中“XXX”为版本号。

应用程序会根据传入的 Listener 的地址从集群中获取 MergeServer 的列表，并且根据集群流量分配信息从中选择一个 MergeServer 来执行程序中的 SQL 请求。

* 前提条件

- OceanBase 主备集群已经安装且正常运行。
- Java 开发环境以及部署完成，如已经安装 JDK、Eclipse 等。
- 已经获取 OceanBase Java 客户端 Jar 包：“oceanbase-core-1.2.3.jar”和“druid-0.2.12.jar”。
- 已经获取依赖包：“commons-lang-2.3.jar”、“commons-logging-1.1.jar”和“mysql-connector-java-5.1.14.jar”。

* 案例操作流程

1. 连接 OceanBase 数据库。
2. 新建 PreparedStatement 语句，来查询__all_server 表中 svr_port 为某个值的 server 信息。
3. 指定查询__all_server 表中 svr_port 为 2600 的 server 信息。
4. 打印查询结果。

* 预期结果

Eclipse Console 界面将打印出所有端口号为 2600 的 server 信息。

* 示例

假设搭建的 OceanBase 数据库包含两个集群，两个集群中的 Listener 的地址分别为：10.232.36.29:15847,10.232.36.30:15847

*说明：*使用 obGroupDataSource 来访问 Oceanbase 数据库的时候，访问请求会跟进两个集群的流量分配准备来将请求发送给其中一个集群的 MergeServer 上面。

编辑 Java 代码，如下所示：

```
import java.io.*;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;
import java.util.*;
import javax.sql.DataSource;

public class ObGroupDataSourceExample {
    public static void main(String[] args) throws Exception {
        // 初始化 OceaBase DataSource 。
        OBGroupDataSource obGroupDataSource = new OBGroupDataSource();
        obGroupDataSource.setUsername("admin");
        obGroupDataSource.setPassword("admin");
        obGroupDataSource.setLMS("10.232.36.29:15847,10.232.36.30:15847");
        obGroupDataSource.init();
        Connection conn = obGroupDataSource.getConnection();
        PreparedStatement pstmt = conn
            .prepareStatement("select * from __all_server where
svr_port = ?");
        pstmt.setInt(1, 2600);
        ResultSet rs = pstmt.executeQuery();
```

```

        ResultSetMetaData rSetMetaData = rs.getMetaData();
        for (int i = 1; i <= rSetMetaData.getColumnCount(); i++) {
            System.out.print("|" + rSetMetaData.getColumnName(i) + "");
        }
        System.out.println();
        while (rs.next()) {
            for (int i = 1; i <= rSetMetaData.getColumnCount(); i++) {
                System.out.print("|" + rs.getString(i) + "");
            }
            System.out.println();
        }
        if (rs != null) {
            rs.close();
        }
        if (pstmt != null) {
            pstmt.close();
        }
        if (conn != null) {
            conn.close();
        }
    }
}

```

如果 Eclipse Console 界面输出如下信息则表明运行正常。其中第一行为 __all_server 表的 schema 信息，而第二行才是真正的输出结果行。

```

|gm_create|gm_modify|cluster_id|svr_type|svr_ip|svr_port|inner_port|svr_role|svr_version
|2013-05-27 21:54:30.08001|2013-05-29
17:54:45.81237|0|chunkserver|10.209.144.31|2600|0|0|0.4.1.2_13189M(Apr 10 2013
12:40:32)

```

2.3.3 使用 MergeServer 直接访问 OceanBase

利用 MergeServer 访问 OceanBase 的过程最为直接，你只要知道一台 OceanBase 集群中 MergeServer 的地址和端口，就能直接进行访问，你所发出的所有 SQL 请求也都将由这台 MergeServer 进行处理。

该方式没有通过 OceanBase Java 客户端连接 OceanBase，因此，无需导入 OceanBase Java 客户端的 jar 包。

* 前提条件

- OceanBase 主备集群已经安装且正常运行。
- Java 开发环境以及部署完成，如已经安装 JDK、Eclipse 等。

- 已经获取依赖包：“commons-lang-2.3.jar”、“commons-logging-1.1.jar”和“mysql-connector-java-5.1.14.jar”。

* 案例操作流程

1. 连接 OceanBase 数据库。
2. 新建 PreparedStatement 语句，来查询__all_server 表中 svr_port 为某个值的 server 信息。
3. 指定查询__all_server 表中 svr_port 为 2600 的 server 信息。
4. 打印查询结果。

* 预期结果

Eclipse Console 界面将打印出所有端口号为 2600 的 server 信息。

* 示例

编辑 Java 代码，如下所示：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class OceanBaseMergeServerExample {
    public static void main(String[] argv) {
        executeByPreparedStatement();
    }

    public static void executeByPreparedStatement() {
        Connection conn = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection(
                "jdbc:mysql://10.232.36.29:15847",
                "admin", "admin");
            pstmt = conn
                .prepareStatement("select * from
__all_server where svr_port = ?");
```

```

        pstmt.setInt(1, 2600);
        rs = pstmt.executeQuery();
        ResultSetMetaData rSetMetaData = rs.getMetaData();
        for (int i = 1; i <= rSetMetaData.getColumnCount(); i++) {
            System.out.print("|" +
rSetMetaData.getColumnName(i) + "");
        }
        System.out.println();
        while (rs.next()) {
            for (int i = 1; i <= rSetMetaData.getColumnCount();
i++) {
                System.out.print("|" + rs.getString(i) + "");
            }
            System.out.println();
        }
    } catch (Exception e) {
    } finally {
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException ex) {
            }
        }
        if (pstmt != null) {
            try {
                pstmt.close();
            } catch (SQLException ex) {
            }
        }
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException ex) {
            }
        }
    }
}
}
}
}

```

如果 Eclipse Console 界面输出如下信息则表明运行正常。其中第一行为 __all_server 表的 schema 信息，而第二行才是真正的输出结果行。

```
|gm_create|gm_modify|cluster_id|svr_type|svr_ip|svr_port|inner_port|svr_role|svr_version
```

|2013-05-27 21:54:30.08001|2013-05-29
17:54:45.81237|0|chunkserver|10.209.144.31|2600|0|0|0.4.1.2_13189M(Apr 10 2013
12:40:32)

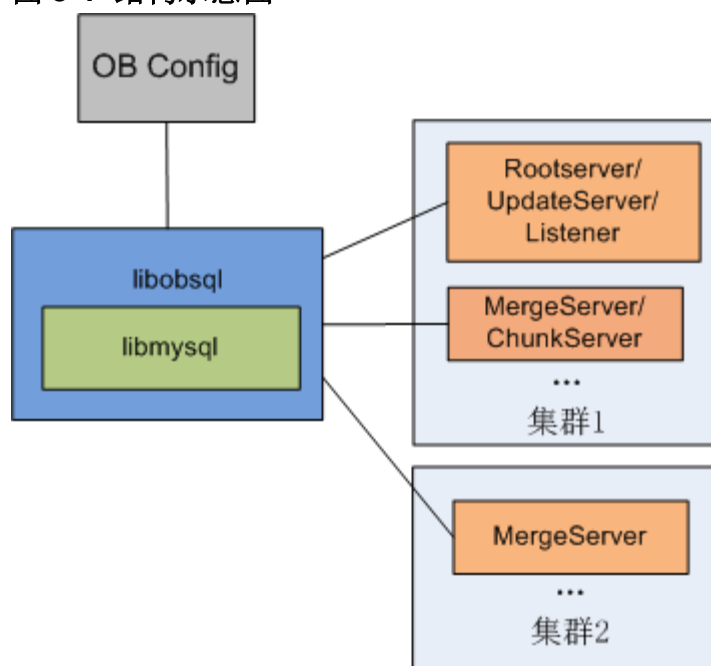
3 OceanBase C 客户端

OceanBase C 客户端主要用于开发人员编写 C 程序连接 OceanBase 数据库。它通过动态库“libobsql.so”的形式提供给用户使用。

3.1 客户端结构

OceanBase C 客户端的结构示意图如[图 3-1](#)所示。

图 3-1 结构示意图



OceanBase C 客户端的结构介绍如下：

- **OB Config**（可选）
OceanBase 配置管理中心，仅供阿里内部使用。主要负责客户端获取 OceanBase 集群地址、管理多个版本客户端库和元数据和通过配置项管理客户端的升级等功能。用户只需要获取到 **OB Config** 的地址以及 OceanBase 的用户名和密码就能获取到数据库的访问地址以及流量分配等信息，从而初始化整个 OceanBase C 客户端。详细请参见《OceanBase 0.4.2 配置中心 安装指南》。
- **libmysqlclient**
MySQL 提供的 C 语言客户端库，用来访问 MySQL 数据库。
- **libobsql**
OceanBase 提供的 C 语言客户端库。libobsql 封装了 libmysqlclient，它实现了所有 libmysqlclient 的接口。用户可以通过使用“LD_PRELOAD”环

境变量，调用“libmysqlclient”动态库中的接口，但是实际上是调用的 libobsql 的接口。

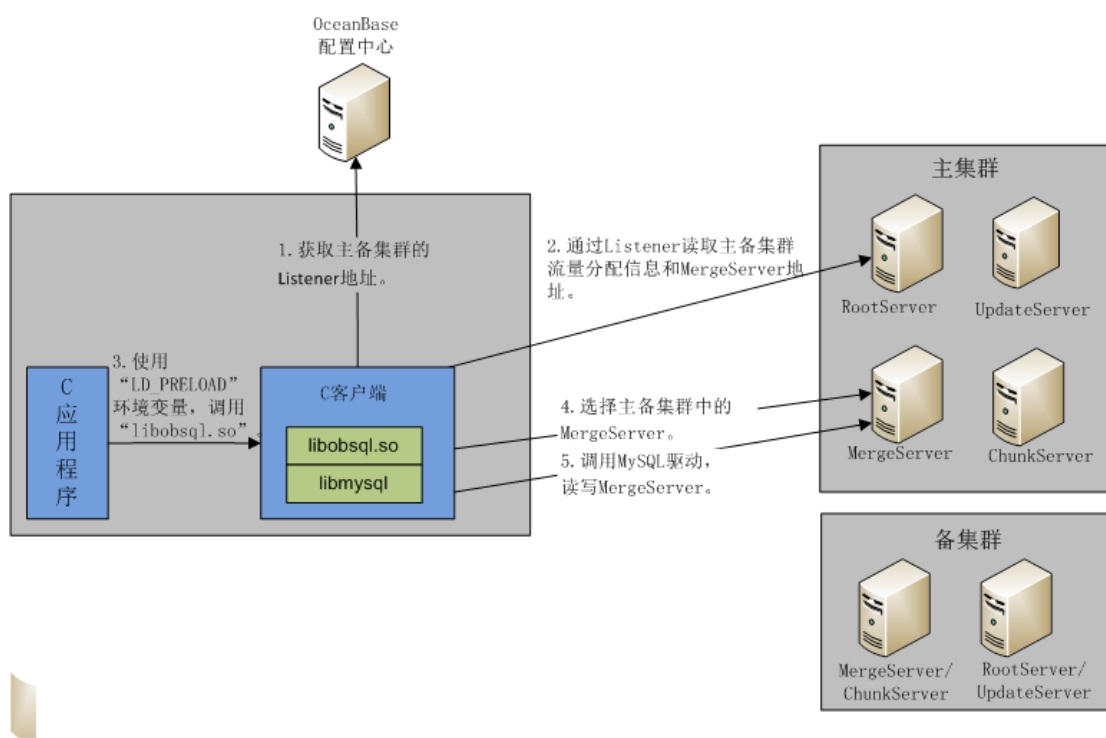
- RootServer/UpdateServer/MergeServer/ChunkServer
OceanBase 集群的组成模块，具体功能介绍请参见《OceanBase 0.4.2 描述》。
- Listener
OceanBase 集群内的特殊角色，是客户端获取访问 OceanBase 中 MergeServer 列表、集群流量分配的接口。

3.2 访问流程

应用程序可以使用 libmysqlclient 提供的 API 接口访问 OceanBase，通过 libmysqlclient 编译的应用程序无需重新编译即可连接 OceanBase。

OceanBase C 客户端的访问流程如图 3-2 所示。

图 3-2 访问流程



3.3 安装

假设本地计算机的用户为 **sqluser**。安装 OceanBase C 客户端的操作步骤如下：

1. 以 **sqluser** 用户登录本地计算机。
2. 执行以下命令，检出 Oceanbase 源码。
git clone https://github.com/alibaba/oceanbase oceanbase_install

3. 执行以下命令，进入“oceanbase_0.4”分支。
git checkout oceanbase_0.4
4. 执行以下命令，进入 OceanBase C 客户端安装包的存放目录。
cd ~/oceanbase_install/client_package/c
5. 执行以下命令，安装依赖库 curl。
sudo rpm -Uvh curl-7.29.0-1.el6.x86_64.rpm
6. 执行以下命令，安装 OceanBase C 客户端。
sudo rpm -Uvh oceanbase-devel-0.4.2.1-1193.el6.x86_64.rpm
7. 使用 vi 编辑器在“~/.bashrc”文件中添加以下内容。

```
export LD_PRELOAD=/home/admin/oceanbase/lib/libobsq1.so.0.0.0
export OB_SQL_CONFIG_DIR=/home/admin/oceanbase/etc/
```

8. 执行以下命令，使环境变量生效。
source ~/.bashrc
9. 使用 vi 编辑器修改“~/oceanbase/etc/libobsq1.conf”文件，参数说明如[表 3-1](#)所示。

```
logfile=/tmp/obsq1.log
#initurl=http://10.232.102.182:8080/diamond-server/config.co?dataId=197
loglevel=DEBUG
minconn=2
maxconn=50
ip=10.10.10.2
port=2828
username=admin
passwd=admin
```

表 3-1 参数说明

参数名称	说明
logfile	客户端日志路径。
initurl	OB Config 的 URL 地址，向 DBA 申请获取。需要搭建 OB Config，详细请参见《OceanBase 0.4.2 配置中心 安装指南》。 使用 OB Config 获取主备集群的 Listener 时，需要配置。
minconn	客户端到 MergeServer 的最小连接数。

参数名称	说明
maxconn	客户端到 MergeServer 的最大连接数。
ip	主集群中主 RootServer 的 IP。 使用 OB Config 获取主备集群的 Listener 时，不需要配置。
port	Listener 的 MySQL 的协议端口。 使用 OB Config 获取主备集群的 Listener 时，不需要配置。
username	连接 OceanBase 的用户名。 缺省值：admin
passwd	连接 OceanBase 的密码。 缺省值：admin

3.4 访问方式

主要通过一个示例介绍如何使用 OceanBase C 客户端连接 OceanBase。

* 前提条件

- OceanBase 主备集群已经安装且正常运行。
- 配置“__all_cluster”中主备集群的流量分配“cluster_flow_percent”，主集群为“0”，备集群为“100”。
- C 开发环境已经部署完成，如已安装 g++，gcc 等。
- 客户端服务器用户：**sqluser**。

* 案例操作流程

编写一段 C 程序，使用 OceanBase C 客户端进行如下操作：

1. 连接 OceanBase 数据库。
2. 创建表“test”：c1 列为主键，int 类型；c2 列为 varchar 类型。
3. 插入两行数据“1”、“hello ob”和“2”、“hello java”。

4. 查询表“test”内容。

5. 删除表“test”。

* 预期结果

根据客户端特性，create table、insert 和 drop table 在主集群中执行，select 操作在备集群中执行。

* 示例

1. 编写一段 SQL 的 C 代码，保存为“test.c”文件，并上传至客户端服务器的“/home/sqluser”目录。

*说明：*API 的完整说明请参考 [《MySQL C API 文档》](#)。

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mysql/mysql.h> //注意，此处为“mysql.h”
int main(int argc, char *argv[])
{
    (void)argc;
    (void)argv;
    const char* HOST = "127.0.0.1";
    int PORT = 2828;

    // 1. 初始化 libmysqlclient
    if (0 != mysql_library_init(0, NULL, NULL))
    {
        fprintf(stderr, "could not init mysql library\n");
        exit(1);
    }
    MYSQL my_;
    mysql_init(&my_);

    // 2. 连接 OceanBase SQL 服务
    fprintf(stderr, "Connecting server %s:%d...\n", HOST, PORT);
    mysql_real_connect(&my_, HOST, "admin", "admin", "", PORT, NULL, 0);

    // 3. 建表
    int ret = 0;
    ret = mysql_query(&my_, "drop table if exists test");
    if (0 != ret)
    {
        fprintf(stderr, "%sn", mysql_error(&my_));
    }
}
```

```

    }
    ret = mysql_query(&my_, "create table if not exists test (c1 int primary key, c2
varchar)");
    if (0 != ret)
    {
        fprintf(stderr, "%sn", mysql_error(&my_));
    }

    //建表后预留用于主备同步的时间。
    sleep(60);

    //4. 插入数据
    ret = mysql_query(&my_, "insert into test (c1, c2) values (1, 'hello ob'), (2, 'hello
java')");
    if (0 != ret)
    {
        fprintf(stderr, "%sn", mysql_error(&my_));
    }

    // 5. 查询数据
    ret = mysql_query(&my_, "select * from test");
    if (0 != ret)
    {
        fprintf(stderr, "%sn", mysql_error(&my_));
    }
    MYSQL_RES *result = mysql_store_result(&my_);

    // 6. 删表
    ret = mysql_query(&my_, "drop table if exists test");
    if (0 != ret)
    {
        fprintf(stderr, "%sn", mysql_error(&my_));
    }

    // 7. destroy
    mysql_close(&my_);
    mysql_library_end();
    return 0;
}

```

2. 以 **sqluser** 用户登录客户端服务器。
3. 执行以下命令，编译“test.c”。
g++ test.c -o atest -I /usr/include -L /usr/lib64/mysql -lmysqlclient

4. 执行以下命令，运行“test.c”。

./test

5. 分别登录主备集群中安装 MergeServer 的服务器，查看 MergeServer 的日志文件“/home/admin/oceanbase/log/mergeserver.log”。

如果出现如下日志，则说明客户端运行正常。

- 主集群

```
[2013-06-08 10:51:16.467612] INFO  ob_mysql_server.cpp:1054
[139693026940672] start query: "create table test (c1 int primary key, c2
varchar)" real_query_len=50, peer=10.14.19.17:53472
[2013-06-08 10:52:16.764349] INFO  ob_mysql_server.cpp:1054
[139693440083712] start query: "insert into test values (1, 'hello ob'), (2, 'hello
java')" real_query_len=58, peer=10.14.19.17:53472
[2013-06-08 10:52:16.810628] INFO  ob_mysql_server.cpp:1054
[139693062612736] start query: "drop table test" real_query_len=15,
peer=10.14.19.17:53472
```

- 备集群

```
[2013-06-08 10:52:16.797199] INFO  ob_mysql_server.cpp:1056
[140177548736256] start query: "select * from test" real_query_len=18,
peer=10.14.19.17:53527
```

4 附录

4.1 监控系统采集引擎计算规则

本小节主要介绍监控系统采集引擎计算规则，包括表达式求值计算、表达式对比计算、表达式对比计算且求采集周期内平均值和无计算。

4.1.1 表达式求值计算

该计算类型仅仅只针对某个指标的表达式进行计算并获得计算后的具体值。如缓存命中率： $\text{block_cache_hit} / (\text{block_cache_hit} + \text{block_cache_miss})$ 。

详细规则条目如下：

Item Name	Item Compute Express	Item Type
memory_total	memory_used_default + memory_used_network + memory_used_thread_buffer + memory_used_tablet + memory_used_bi_cache + memory_used_block_cache + memory_used_bi_cache_unserving + memory_used_block_cache_unserving + memory_used_join_cache + memory_used_sstable_row_cache + memory_used_merge_buffer + memory_used_merge_split_buffer	express
block_index_cache_miss	block_index_cache_miss - block_index_cache_miss_org	express
tablet_merge_percent	old_ver_merged_tablets_num / old_ver_tablets_num	express
split_count	new_ver_tablets_num - old_ver_merged_tablets_num	express

Item Name	Item Compute Express	Item Type
sql_active_session	sql_succ_login_count - sql_logout_count + sql_fail_login_count	express
location_cache_percent	location_cache_hit / (location_cache_hit + location_cache_miss)	express
sql_ps_allocator_count	sql_ps_allocator_count - 0	express

4.1.2 表达式对比计算

该计算类型需要将当前采集周期采集所获得的某个指标值与上一个采集周期所采集到的该指标值进行求差计算并最终获得计算后的具体值。如 **SELECT** 请求平均耗时： $(\text{sql_select_time} - \text{sql_select_time_orgi}) / (\text{sql_select_count} - \text{sql_select_count_orgi})$ 。

- sql_select_time: 本次采集的指标值
- sql_select_time_orgi: 上次采集的指标值
- sql_select_count: 本次采集的指标值
- sql_select_count_orgi: 上次采集的指标值

详细规则条目如下：

Item Name	Item Compute Express	Item Type
sql_avg_select_time	$(\text{sql_select_time} - \text{sql_select_time_orgi}) / (\text{sql_select_count} - \text{sql_select_count_orgi})$	avg
sql_avg_insert_time	$(\text{sql_insert_time} - \text{sql_insert_time_orgi}) / (\text{sql_insert_count} - \text{sql_insert_count_orgi})$	avg
sql_avg_replace_time	$(\text{sql_replace_time} - \text{sql_replace_time_orgi}) / (\text{sql_replace_count} - \text{sql_replace_count_orgi})$	avg

Item Name	Item Compute Express	Item Type
sql_avg_update_time	$(\text{sql_update_time} - \text{sql_update_time_orgi}) / (\text{sql_update_count} - \text{sql_update_count_orgi})$	avg
sql_avg_delete_time	$(\text{sql_delete_time} - \text{sql_delete_time_orgi}) / (\text{sql_delete_count} - \text{sql_delete_count_orgi})$	avg
avg_get_time	$(\text{get_time} - \text{get_time_orgi}) / (\text{get_count} - \text{get_count_orgi})$	avg
avg_scan_time	$(\text{scan_time} - \text{scan_time_orgi}) / (\text{scan_count} - \text{scan_count_orgi})$	avg
avg_apply_time	$(\text{apply_time} - \text{apply_time_orgi}) / (\text{apply_count} - \text{apply_count_orgi})$	avg
avg_batch_time	$(\text{batch_time} - \text{batch_time_orgi}) / (\text{batch_count} - \text{batch_count_orgi})$	avg
avg_merge_time	$(\text{merge_time} - \text{merge_time_orgi}) / (\text{merge_count} - \text{merge_count_orgi})$	avg
block_cache_hit	$\text{block_cache_hit} - \text{block_cache_hit_orgi}$	avg
block_cache_miss	$\text{block_cache_miss} - \text{block_cache_miss_orgi}$	avg
block_index_cache_hit	$\text{block_index_cache_hit} - \text{block_index_cache_hit_orgi}$	avg
sstable_row_cache_hit	$\text{sstable_row_cache_hit} - \text{sstable_row_cache_hit_orgi}$	avg

Item Name	Item Compute Express	Item Type
sstable_row_cache_miss	$\text{sstable_row_cache_miss} - \text{sstable_row_cache_miss_orgi}$	avg
avg_get_time	$(\text{get_time} - \text{get_time_orgi}) / (\text{get_count} - \text{get_count_orgi})$	avg
avg_scan_time	$(\text{scan_time} - \text{scan_time_orgi}) / (\text{scan_count} - \text{scan_count_orgi})$	avg
block_cache_percent	$(\text{block_cache_hit} - \text{block_cache_hit_orgi}) / (\text{block_cache_hit} - \text{block_cache_hit_orgi} + \text{block_cache_miss} - \text{block_cache_miss_orgi})$	avg
block_index_cache_percent	$(\text{block_index_cache_hit} - \text{block_index_cache_hit_orgi}) / (\text{block_index_cache_hit} - \text{block_index_cache_hit_orgi} + \text{block_index_cache_miss} - \text{block_index_cache_miss_orgi})$	avg
sstable_row_cache_percent	$(\text{sstable_row_cache_hit} - \text{sstable_row_cache_hit_orgi}) / (\text{sstable_row_cache_hit} - \text{sstable_row_cache_hit_orgi} + \text{sstable_row_cache_miss} - \text{sstable_row_cache_miss_orgi})$	avg
queue_wait_time	$(\text{queue_wait_time} - \text{queue_wait_time_orgi}) / (\text{request_count} - \text{request_count_orgi})$	avg
avg_get_event_time	$(\text{get_event_time} - \text{get_event_time_orgi}) / (\text{get_event_count} - \text{get_event_count_orgi})$	avg

Item Name	Item Compute Express	Item Type
avg_scan_event_time	$(\text{scan_event_time} - \text{scan_event_time_orgi}) / (\text{scan_event_count} - \text{scan_event_count_orgi})$	avg
location_cache_hit	$\text{location_cache_hit} - \text{location_cache_hit_orgi}$	avg
location_cache_miss	$\text{location_cache_miss} - \text{location_cache_miss_orgi}$	avg
sql_avg_modify_time	$(\text{sql_insert_time} - \text{sql_insert_time_orgi} + \text{sql_replace_time} - \text{sql_replace_time_orgi} + \text{sql_update_time} - \text{sql_update_time_orgi} + \text{sql_delete_time} - \text{sql_delete_time_orgi}) / (\text{sql_insert_count} - \text{sql_insert_count_orgi} + \text{sql_replace_count} - \text{sql_replace_count_orgi} + \text{sql_update_count} - \text{sql_update_count_orgi} + \text{sql_delete_count} - \text{sql_delete_count_orgi})$	avg

4.1.3 采集周期内平均值

在“1.2 表达式对比计算”中的计算结束后，会除以采集周期，从而得到当前采集周期之内的一个相对平均值。采集周期可以用户自定义，默认值为“1min”。如请求行数： $(\text{sstable_get_rows} - \text{sstable_get_rows_orgi}) + (\text{sstable_scan_rows} - \text{sstable_scan_rows_orgi})$ 。

- sstable_get_rows: 本次采集的指标值
- sstable_get_rows_orgi: 上次采集的指标值
- sstable_scan_rows: 本次采集的指标值
- sstable_scan_rows_orgi: 上次采集的指标值

详细规则条目如下：

Item Name	Item Compute Express	Item Type
commit_log_size	commit_log_size - commit_log_size_org	all
sstable_disk_io_num	sstable_disk_io_num - sstable_disk_io_num_org	all
sstable_disk_io_bytes	sstable_disk_io_bytes - sstable_disk_io_bytes_org	all
rpc_bytes_in	rpc_bytes_in - rpc_bytes_in_org	all
rpc_bytes_out	rpc_bytes_out - rpc_bytes_out_org	all
succ_get_count	succ_get_count - succ_get_count_org	all
succ_scan_count	succ_scan_count - succ_scan_count_org	all
fail_get_count	fail_get_count - fail_get_count_org	all
fail_scan_count	fail_scan_count - fail_scan_count_org	all
migrate_count	migrate_count - migrate_count_org	all
copy_count	copy_count - copy_count_org	all
rpc_bytes_in	rpc_bytes_in - rpc_bytes_in_org	all
rpc_bytes_out	rpc_bytes_out - rpc_bytes_out_org	all

Item Name	Item Compute Express	Item Type
request_count	request_count - request_count_org	all
request_rows	(sstable_get_rows - sstable_get_rows_org) + (sstable_scan_rows - sstable_scan_rows_org)	all
get_count	get_count - get_count_org	all
scan_count	scan_count - scan_count_org	all
sql_insert_count	sql_insert_count - sql_insert_count_org	all
sql_replace_count	sql_replace_count - sql_replace_count_org	all
sql_update_count	sql_update_count - sql_update_count_org	all
sql_delete_count	sql_delete_count - sql_delete_count_org	all
sql_succ_query_count	(sql_succ_query_count - sql_succ_query_count_org) + (sql_succ_exec_count - sql_succ_exec_count_org)	all
sql_fail_query_count	sql_fail_query_count - sql_fail_query_count_org	all
sstable_disk_io_num	sstable_disk_io_num - sstable_disk_io_num_org	all
sstable_disk_io_bytes	sstable_disk_io_bytes - sstable_disk_io_bytes_org	all

Item Name	Item Compute Express	Item Type
sql_select_count	sql_select_count - sql_select_count_org	all
all_sql_select_count	sql_select_count - sql_select_count_org	all
sql_succ_prepare_count	sql_succ_prepare_count - sql_succ_prepare_count_org	all
sql_fail_prepare_count	sql_fail_prepare_count - sql_fail_prepare_count_org	all
sql_succ_exec_count	sql_succ_exec_count - sql_succ_exec_count_org	all
sql_fail_exec_count	sql_fail_exec_count - sql_fail_exec_count_org	all
sql_succ_close_count	sql_succ_close_count - sql_succ_close_count_org	all
sql_fail_close_count	sql_fail_close_count - sql_fail_close_count_org	all
rpc_bytes_out	rpc_bytes_out - rpc_bytes_out_org	all
rpc_bytes_in	rpc_bytes_in - rpc_bytes_in_org	all
sql_modify_count	(sql_insert_count - sql_insert_count_org) + (sql_replace_count - sql_replace_count_org) + (sql_update_count - sql_update_count_org) + (sql_delete_count - sql_delete_count_org)	all

Item Name	Item Compute Express	Item Type
block_cache_hit_per_second	block_cache_hit - block_cache_hit_org	all
block_cache_miss_per_second	block_cache_miss - block_cache_miss_org	all
block_index_cache_hit_per_second	block_index_cache_hit - block_index_cache_hit_org	all
block_index_cache_miss_per_second	block_index_cache_miss - block_index_cache_miss_org	all
sstable_row_cache_hit_per_second	sstable_row_cache_hit - sstable_row_cache_hit_org	all
sstable_row_cache_miss_per_second	sstable_row_cache_miss - sstable_row_cache_miss_org	all
get_event_count	get_event_count - get_event_count_org	all
scan_event_count	scan_event_count - scan_event_count_org	all
rpc_bytes_in	rpc_bytes_in - rpc_bytes_in_org	all
rpc_bytes_out	rpc_bytes_out - rpc_bytes_out_org	all
get_count	get_count - get_count_org	all
scan_count	scan_count - scan_count_org	all
apply_count	apply_count - apply_count_org	all

Item Name	Item Compute Express	Item Type
batch_count	batch_count - batch_count_org i	all
merge_count	merge_count - merge_count_o rgi	all
get_bytes	get_bytes - get_bytes_orgi	all
scan_bytes	scan_bytes - scan_bytes_orgi	all

4.1.4 无计算

仅仅采集指标值，不进行任何操作。

详细规则条目如下：

Item Name	Item Compute Express	Item Type
sql_logout_count	-	none
sql_succ_login_count	-	none
all_table_count	-	none
all_tablet_count	-	none
all_row_count	-	none
all_data_size	-	none
old_ver_tablets_num	-	none
memory_used_block_cache	-	none
memory_used_bi_cache	-	none

Item Name	Item Compute Express	Item Type
memory_used_sstable_row_cache	-	none
memory_used_join_cache	-	none
total_rows	-	none
active_total_rows	-	none
frozen_total_rows	-	none
memory_limit	-	none
memory_total	-	none
memtable_total	-	none
active_memtable_limit	-	none
active_memtable_total	-	none
frozen_memtable_limit	-	none
frozen_memtable_total	-	none
load1	-	none
load5	-	none
load15	-	none
cpu_sys	-	none
cpu_user	-	none
cpu_iowait	-	none

Item Name	Item Compute Express	Item Type
mem_used	-	none
mem_free	-	none
swap	-	none
network_in	-	none
network_out	-	none
ms_memory_limit	-	none
ms_memory_total	-	none
ms_memory_parser	-	none
ms_memory_transformer	-	none
ms_memory_ps_plan	-	none
ms_memory_rpc_request	-	none
ms_memory_sql_array	-	none
ms_memory_expression	-	none
ms_memory_row_store	-	none
ms_memory_session	-	none

4.2 数据展示计算规则

主要为了解决在数据展示时，单位不一致的情况。如采集的内存数值，是一个以 **Byte** 为单位的值。而这样的数值，不适合查看。所以根据配置的规则，将该值转换为以 **GB** 为单位的值。

详细配置条目如下：

Item Name	Item Show Express	Value Type
sql_avg_select_time	-	微妙/次
sql_avg_insert_time	-	微妙/次
sql_avg_replace_time	-	微妙/次
sql_avg_update_time	-	微妙/次
sql_avg_delete_time	-	微妙/次
avg_get_time	-	微妙/次
avg_scan_time	-	微妙/次
avg_apply_time	-	微妙/次
avg_batch_time	-	微妙/次
avg_merge_time	-	微妙/次
block_cache_hit	-	次/分
block_cache_miss	-	次/分
block_index_cache_hit	-	次/分
sstable_row_cache_hit	-	次/分
sstable_row_cache_miss	-	次/分
avg_get_time	-	微妙/次
avg_scan_time	-	微妙/次
block_cache_percent	block_cache_percent*100	%

Item Name	Item Show Express	Value Type
block_index_cache_percent	block_index_cache_percent*100	%
sstable_row_cache_percent	sstable_row_cache_percent*100	%
queue_wait_time	-	微秒/请求
avg_get_event_time	-	微秒/次
avg_scan_event_time	-	微秒/次
location_cache_hit	-	次/分
location_cache_miss	-	次/分
sql_avg_modify_time	-	微妙/次
memory_total	memory_total/(1024*1024*1024)	GB
tablet_merge_percent	-	%
split_count	-	个
sql_active_session	-	个
location_cache_percent	location_cache_percent*100	%
sql_ps_allocator_count	-	个
block_index_cache_miss	-	-
commit_log_size	commit_log_size/1024	KB/秒
sstable_disk_io_num	-	个/秒
sstable_disk_io_bytes	-	Byte/秒

Item Name	Item Show Express	Value Type
rpc_bytes_in	-	KB/秒
rpc_bytes_out	-	KB/秒
succ_get_count	-	次/秒
succ_scan_count	-	次/秒
fail_get_count	-	次/秒
fail_scan_count	-	次/秒
migrate_count	-	次/秒
copy_count	-	次/秒
rpc_bytes_in	rpc_bytes_in/1024	Byte/秒
rpc_bytes_out	rpc_bytes_out/1024	Byte/秒
request_count	-	次/秒
request_rows	-	行/秒
get_count	-	次/秒
scan_count	-	次/秒
sql_insert_count	-	次/秒
sql_replace_count	-	次/秒
sql_update_count	-	次/秒
sql_delete_count	-	次/秒

Item Name	Item Show Express	Value Type
sql_succ_query_count	-	次/秒
sql_fail_query_count	-	次/秒
sstable_disk_io_num	-	个/秒
sstable_disk_io_bytes	sstable_disk_io_bytes/1024	KB/秒
sql_select_count	-	次/秒
all_sql_select_count	-	次/秒
sql_succ_prepare_count	-	次/秒
sql_fail_prepare_count	-	次/秒
sql_succ_exec_count	-	次/秒
sql_fail_exec_count	-	次/秒
sql_succ_close_count	-	次/秒
sql_fail_close_count	-	次/秒
rpc_bytes_out	rpc_bytes_out/1024	KB/秒
rpc_bytes_in	rpc_bytes_in/1024	KB/秒
sql_modify_count	-	次/秒
block_cache_hit_per_second	-	次/秒
block_cache_miss_per_second	-	次/秒
block_index_cache_hit_per_second	-	次/秒

Item Name	Item Show Express	Value Type
block_index_cache_miss_per_second	-	次/秒
sstable_row_cache_hit_per_second	-	次/秒
sstable_row_cache_miss_per_second	-	次/秒
get_event_count	-	次/秒
scan_event_count	-	次/秒
rpc_bytes_in	rpc_bytes_in/1024	KB/秒
rpc_bytes_out	rpc_bytes_out/1024	KB/秒
get_count	-	次/秒
scan_count	-	次/秒
apply_count	-	次/秒
batch_count	-	次/秒
merge_count	-	次/秒
get_bytes	get_bytes/1024	KB/秒
scan_bytes	-	Byte/秒
sql_logout_count	-	次
sql_succ_login_count	-	次/分
all_table_count	-	个

Item Name	Item Show Express	Value Type
all_tablet_count	-	个
all_row_count	-	行
all_data_size	$\text{all_data_size}/(1024*1024*1024)$	GB
old_ver_tablets_num	-	个
memory_used_block_cache	$\text{memory_used_block_cache}/(1024*1024*1024)$	GB
memory_used_bi_cache	$\text{memory_used_bi_cache}/(1024*1024)$	MB
memory_used_sstable_row_cache	$\text{memory_used_sstable_row_cache}/(1024*1024*1024)$	GB
memory_used_join_cache	$\text{memory_used_join_cache}/(1024*1024)$	MB
total_rows	$\text{total_rows}/1000000$	百万行
active_total_rows	$\text{active_total_rows}/1000000$	百万行
frozen_total_rows	$\text{frozen_total_rows}/1000000$	百万行
memory_limit	$\text{memory_limit}/(1024*1024*1024)$	GB
memory_total	$\text{memory_total}/(1024*1024*1024)$	GB
memtable_total	$\text{memtable_total}/(1024*1024*1024)$	GB
active_memtable_limit	$\text{active_memtable_limit}/(1024*1024*1024)$	GB

Item Name	Item Show Express	Value Type
active_memtable_total	$\text{active_memtable_total}/(1024*1024*1024)$	GB
frozen_memtable_limit	$\text{frozen_memtable_limit}/(1024*1024*1024)$	GB
frozen_memtable_total	$\text{frozen_memtable_total}/(1024*1024*1024)$	GB
load1	-	-
load5	-	-
load15	-	-
cpu_sys	-	-
cpu_user	-	-
cpu_iowait	-	-
mem_used	-	GB
mem_free	-	GB
swap	-	-
network_in	-	KB/s
network_out	-	KB/s
ms_memory_limit	$\text{ms_memory_limit}/(1024*1024*1024)$	次/秒
ms_memory_total	$\text{ms_memory_total}/(1024*1024*1024)$	GB

Item Name	Item Show Express	Value Type
ms_memory_parser	ms_memory_parser/(1024*1024*1024)	GB
ms_memory_transformer	ms_memory_transformer/(1024*1024*1024)	GB
ms_memory_ps_plan	ms_memory_ps_plan/(1024*1024*1024)	GB
ms_memory_rpc_request	ms_memory_rpc_request/(1024*1024*1024)	GB
ms_memory_sql_array	ms_memory_sql_array/(1024*1024*1024)	GB
ms_memory_expression	ms_memory_expression/(1024*1024*1024)	GB
ms_memory_row_store	ms_memory_row_store/(1024*1024*1024)	GB
ms_memory_session	ms_memory_session/(1024*1024*1024)	GB

4.3 OceanBase Java 客户端流量分配执行实现细节

OceanBase Java 客户端中，SQL 操作发送原则如下：

- “select /*+ read_consistency(weak) */ * from ”和“select * from”语句根据集群流量分配发送至主备集群。
- “select /*+read_cluster(master)*/ * from”和“select /*+read_cluster(slave)*/ * from”，根据指定集群发送。
- 其余操作均发送至主集群。

OceanBase Java 客户端流量分配的主要依据为“__all_cluster”表中“cluster_flow_percent”列。

OceanBase Java 客户端流量分配执行流程如下：

1. 解析 SQL 语句，查看该语句类型，主要包括以下三个方面：
 - a. 查看该 SQL 属于 DML 还是 DDL 语句。
 - b. 如果是 DML 语句，那么该 SQL 是否为 SELECT 语句。
 - c. 如果 SELECT 操作，那么该语句是否带有 hint。
2. 如果 SQL 带有 hint，且为 read_cluster 类型的 hint，则依据 hint 中指定的集群，进行分配。例如：
 - /*+ read_cluster(master)*/，直接发送至主集群。
 - /*+ read_cluster(slave) */，直接发送至备集群。
3. 判断 SQL 类型，如果是修改操作，则发送至主集群；如果是查询操作，则按照以下规则：
 - 带有/*+ read_consistency(strong) */，则发送至主集群。
 - 带有/*+ read_consistency(weak) */，则参与流量分配。
 - 不带有任何 hint，则依据 Server 配置项“ob_read_consistency”的值进行分配。
 - 如果 ob_read_consistency < 4，则为弱一致性读取操作，参与流量分配。
 - 如果 ob_read_consistency >= 4，则为强一致性读取操作，直接发送至主集群。
4. 对于参与流量分配的 SQL 操作，OceanBase Java 客户端按照“__cluster_flow_percent”的值，获取各个集群所占有的流量比例，并构建长度为 100 的数组，用于存放主集群引用或是备集群引用（主要是标识为主集群还是备集群）。然后执行 SHUFFLE 操作，随机分布数组中的各个元素。

说明：在“步骤 2”和“步骤 3”中，SQL 操作发送到主集群时，主集群操作计数器执行“+1”操作（该计数器主要为了进一步均衡主集群的流量，对于按流量分配到主集群的非一致性读操作，根据此参数转移至备集群）。

 - 根据构建的数组，如果 SQL 操作被分配到的的是备集群，则直接发送至备集群。
 - 根据构建的数组，如果 SQL 操作被分配到的的是主集群，则先对主集群计数器执行“-1”操作，并查看“-1”之后的值是否为小于 0。如果是小于 0，说明主集群目前流量为 0，可以发送至主集群；如果是大于或等于 0，则根据构建的数组重新分配主备集群。

假设 1: 主集群的 cluster_flow_percent 值为 50，备集群的 cluster_flow_percent 值为 50，ob_read_consistency 值为 3。

```
select * from t1 where t1 = 1; //参与主备流量分配，一半概率至主，一半概率至备
select /*+read_consistency(strong)*/ from t1 where t1 = 1; //直接发送至主集群，计数器“+1”
```

```

select /*+read_consistency(weak)*/ from t1 where t1 = 1; //参与主备流量分配，一半概率至主，一半概率至备
select * from t1 where t1=1 for update; //直接发送至主集群，计数器“+1”
select /*+read_cluster(master)*/ * from t1 where t1=1; //直接发送至主集群，计数器“+1”
select /*+read_cluster(slave)*/ * from t1 where t1=1; //直接发送至备集群
insert into t1 values (1,1); //发送至主集群，计数器“+1”
select * from t1 where t1=1; //根据数组重新分配主备集群（由于前一条主集群计数器执行了加一操作）

```

假设 2: 主集群的 cluster_flow_percent 值为 50, 备集群的 cluster_flow_percent 值为 50, ob_read_consistency 值为 4。

```

select * from t1 where t1 = 1; //发送至主集群，计数器“+1”
select /*+read_consistency(strong)*/ from t1 where t1 = 1; //直接发送至主集群，计数器“+1”
select /*+read_consistency(weak)*/ from t1 where t1 = 1; //参与流量分配，一半概率至主，一半概率至备
select * from t1 where t1=1 for update; //直接发送至主集群，计数器“+1”
select /*+read_cluster(master)*/ * from t1 where t1=1; //直接发送至主集群，计数器“+1”
select /*+read_cluster(slave)*/ * from t1 where t1=1; //直接发送至备集群
insert into t1 values (1,1); //发送至主集群，计数器“+1”
select /*+read_consistency(weak)*/ from t1 where t1=1; //根据数组重新分配主备集群（前一条主集群计数器执行了加一操作）

```

4.4 OceanBase Java 客户端定时 MergeServer 重新生效流程

OceanBase Java 客户端会启动一个定时线程，该线程的主要任务为：扫描内部表“__all_cluster”和“__all_server”，并查看这两个表中的数据是否发生了变化。如果变化，则将生效新的流量规则并上下线 MergeServer。

实现细节如下：

1. 任务调度周期 60-120s（考虑是，将调度时间分散，避免在某个集中时刻，对配置中心的访问，导致配置中心发生异常）。
每次调度时，将首先访问 OceanBase 配置中心，获取访问入口。如果配置中心发生故障，则无法获取访问入口，调度任务将失败。但是，不影响客户端的处理外部请求的功能。
2. 获得到访问入口后，将读取内部表“__all_cluster”和“__all_server”的内容。
3. 与当前使用的信息进行对比，如果发现改变，则开始生效。
 - 流量发生变化，则对决定流量的数组进行重新分配。
 - MergeServer 列表发生变化，则对底层的 Druid 数据源进行创建或销毁。

4.5 OceanBase Java 客户端日志梳理

OceanBase Java 客户端日志主要包以下内容：

- 将每个 MergeServer 的 druid 数据源连接池的状态进行输出，其中包括：当前正在使用的连接个数、历史创建的连接总个数、历史销毁的连接总个数、历史最大连接使用个数、最大可创建的个数等信息。
- 输出 OceanBase Java 客户端与 MergeServer 连接异常信息以及此次异常所完成的动作（重试 MergeServer，直接抛出等）。
- 完全独立 ob-datasource 的日志信息。后续将考虑从日志信息中获取并将信息展示至监控中心。

4.6 OceanBase C 客户端流量分配设计

流量分配只涉及弱一致性读请求，非弱一致性读请求全部发送至主集群，所以 OceanBase C 客户端的流量分配方案采用了一个记账的模型：

每次更新集群信息时都会根据最新的流量比列，构造出一个定长数组，并且用变量“OFFSET”表述数据的当前偏移，变量“MASTER”记录当前一致性请求的次数。

1. 在每次发起读请求时，如果是一致性请求，则直接发往主集群（不查询上述构造的数组），且“MASTER++”，“OFFSET”不变；如果是非一致性请求，则继续步骤 2。
2. 查询上述构造的数组，如果当前偏移是主集群，则继续步骤 3；如果是备集群则直接发送，且“OFFSET++”。
3. 查看 MASTER 值，如果“MASTER==0”，则直接发送请求到该主集群，且“OFFSET++”，“MASTER”不变；否则“MASTER--”，“OFFSET++”，并重复步骤 2。

由于强一致性请求固定发往主集群，弱一致性通过查询数组按流量比发送，因此会造成主集群流量偏大。通过 MASTER 这个变量就有效抵消了强一致性请求到主集群的流量，从而达到服务端设置的流量比列。

Ex: 假设有集群 A 和 B，A 是主集群，B 是备集群，流量比列是“4:6”，请求序列“cq1 Nq1 Cq2 Nq2 Nq3 Nq4”。其中“Cq”表示一致性请求，“Nq”表示非一致性请求。

构造的一个定长数组如下：

A	B	B	A	B	A	B	B	A	B
---	---	---	---	---	---	---	---	---	---

MASTER = 0, OFFSET=0

则流量分配如下：

请求	发往的集群	MASTER	OFFSET
Cq1	A	1	0
Nq1	B	0	2
Cq2	A	1	2
Nq2	B	1	3
Nq3	B	0	5
Nq4	A	0	6

4.7 OceanBase C 客户端重试设计

OceanBase 数据库为分布式系统，单点失效时需要保持对用户透明，所以在客户端需要在访问失败的时候增加重试策略。

对于普通请求，当客户端访问某 MergeServer 失败时，就给它打上标签并持续一段时间，然后随机重试该集群中的另一台 MergeServer 处理。重试过程中不修改内部的 OBGroupDataSource。

对于绑定变量的请求的，访问失败后并不重试，直接把错误交由应用处理，因为在客户端内部无法得知连接上绑定过哪些 statement。