# **MyCAT**

MyCat是基于阿里开源的Cobar产品而研发，Cobar的稳定性、可靠性、优秀的架构和性能以及众多成熟的使用案例使得MYCAT一开始就拥有一个很好的起点，站在巨人的肩膀上，我们能看到更远。业界优秀的开源项目和创新思路被广泛融入到MYCAT的基因中，使得MYCAT在很多方面都领先于目前其他一些同类的开源项目，甚至超越某些商业产品。

MYCAT背后有一支强大的技术团队，其参与者都是5年以上资深软件工程师、架构师、DBA等，优秀的技术团队保证了MYCAT的产品质量。  
MYCAT并不依托于任何一个商业公司，因此不像某些开源项目，将一些重要的特性封闭在其商业产品中，使得开源项目成了一个摆设。  
因此，MyCat你可以认为是从Amoeba->Cobar一路过来的最终版升级者。  
由于MyCat和Corba都是Amoeba框架上发展而来的，如果一个具有Amoeba配置经验的开发者可以几乎不用看任何文档而可以直接使用MyCat来实现mySQL的读写分离更重要的是，基于myCat你可以实现数据的垂直和水平切割，它使得了mySQL据有了真正的“集群”的能力，并为去IOE做好了最终的准备。

myCAT开源项目维护很频繁，目前最新版已经到了1.5 Release(23天前刚维护过)。因此它的维护性、稳定性是得到了保证的。同时myCat的文档极其丰富，参于开发的人员又很多，所以它可以应付很多在以往的Amoeba以及Corba上未能解决的问题，它是一个可以真正被应用在生产环境上的数据库中间件。在下面的章节我们将使用myCat来实现mySQL的读写分离和垂直水平折分的具体案例。

# **myCat介绍**

什么是MyCAT?简单的说，MyCAT就是：

* 一个彻底开源的，面向企业应用开发的“大数据库集群” 支持事务、ACID、可以替代Mysql的加强版数据库
* 一个可以视为“Mysql”集群的企业级数据库，用来替代昂贵的Oracle集群
* 一个融合内存缓存技术、Nosql技术、HDFS大数据的新型SQL Server
* 结合传统数据库和新型分布式数据仓库的新一代企业级数据库产品
* 一个新颖的数据库中间件产品

****目标****

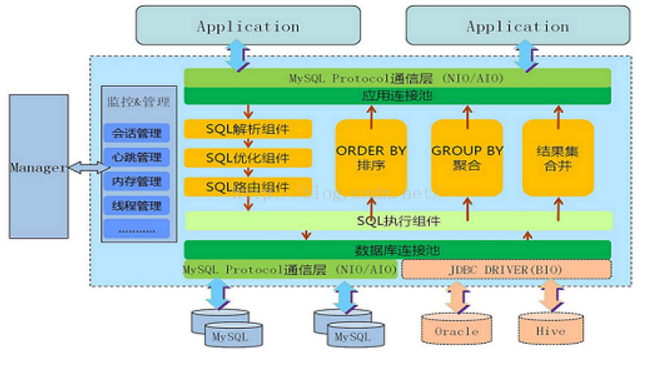
低成本的将现有的单机数据库和应用平滑迁移到“云”端，解决数据存储和业务规模迅速增长情况下的数据瓶颈问题。  
关键特性  
支持 SQL 92标准 支持Mysql集群，可以作为Proxy使用 支持JDBC连接ORACLE、DB2、SQL Server，将其模拟为MySQL Server使用 支持galera for mysql集群，percona-cluster或者mariadb cluster，提供高可用性数据分片集群，自动故障切换，高可用性 ，支持读写分离，支持Mysql双主多从，以及一主多从的模式 ，支持全局表，数据自动分片到多个节点，用于高效表关联查询 ，支持独有的基于E-R 关系的分片策略，实现了高效的表关联查询多平台支持，部署和实施简单。

****优势****

基于阿里开源的Cobar产品而研发，Cobar的稳定性、可靠性、优秀的架构和性能，以及众多成熟的使用案例使得MyCAT一开始就拥有一个很好的起点，站在巨人的肩膀上，我们能看到更远。广泛吸取业界优秀的开源项目和创新思路，将其融入到MyCAT的基因中，使得MyCAT在很多方面都领先于目前其他一些同类的开源项目，甚至超越某些商业产品。MyCAT背后有一只强大的技术团队，其参与者都是5年以上资深软件工程师、架构师、DBA等，优秀的技术团队保证了MyCAT的产品质量。 MyCAT并不依托于任何一个商业公司，因此不像某些开源项目，将一些重要的特性封闭在其商业产品中，使得开源项目成了一个摆设。  
  
  
****长期规划****

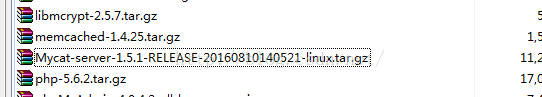
在支持Mysql的基础上，后端增加更多的开源数据库和商业数据库的支持，包括原生支持PosteSQL、FireBird等开源数据库，以及通过JDBC等方式间接支持其他非开源的数据库如Oracle、DB2、SQL Server等实现更为智能的自我调节特性，如自动统计分析SQL，自动创建和调整索引，根据数据表的读写频率，自动优化缓存和备份策略等实现更全面的监控管理功能与HDFS集成，提供SQL命令，将数据库装入HDFS中并能够快速分析集成优秀的开源报表工具，使之具备一定的数据分析的能力。

****MyCat架构****



# **安装myCat**

通过myCat的官方源码网站获得（https://github.com/MyCATApache/Mycat-download）  
本教程中使用的是myCat 8月出的稳定版：Mycat-server-1.5.1-RELEASE-20160810140521。

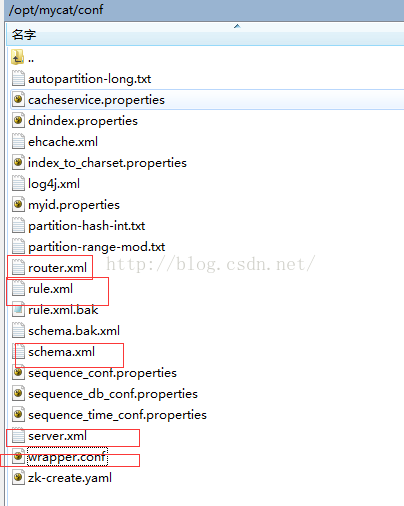


下载后解压至一个目录如：



## **配置myCat**

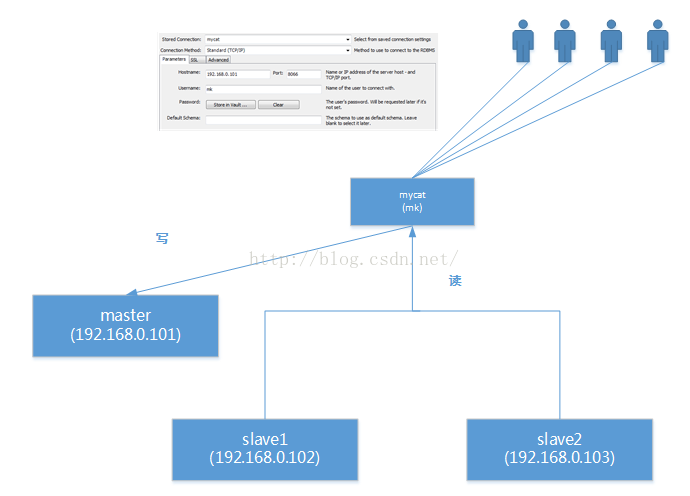
myCat的核心配置文件有如下几个：



* wrapper.conf – 系统环境配置
* server.xml – mycat主配置文件，用于配置mycat对外数据库表、用户名、访问权限等
* schema.xml – 用于配置读写分离、水平垂直折分集群
* rule.xml – 用于配置数据水平垂直折分规则
* router.xml – 配合rule.xml文件使用，当数据折分不符合规则时的走向，类似switch中的default作用

## **myCat配置读写分离**

请看下面这组读写分离，我们就用myCat来配置出这个实例吧。



### **先在物理上构建主从配置**

先按照“Mysql主从配置”把：

* 192.168.0.101 配成master
* 192.168.0.102 为 192.168.0.101的slave1
* 192.168.0.103 为 192.168.0.103的slave3

### **配置myCat读写分离**

****修改server.xml文件****  
打开server.xml文件，你会发觉好大一陀，全部删了吧，改成下面这个配置：

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<?xml** version="1.0" encoding="UTF-8"**?>**
2. <!DOCTYPE mycat:server SYSTEM "server.dtd"**>**
3. **<mycat:server** xmlns:mycat="http://org.opencloudb/"**>**
4. **<system>**
5. **<property** name="defaultSqlParser"**>**druidparser**</property>**
6. **</system>**
8. **<user** name="mk"**>**
9. **<property** name="password"**>**aaaaaa**</property>**
10. **<property** name="schemas"**>**mycat**</property>**
11. **<property** name="readOnly"**>**false**</property>**
12. **</user>**
13. **</mycat:server>**

上述配置定义了这么一件事：

* 定义了一个可供外部访问的myCat的虚拟数据库
* 它的端口为8806
* schema名为mycat
* 客户端访问时的用户名为mk，密码为aaaaaa（六个a）

****修改schema.xml文件****

打开后也是很大的一陀，全删了吧，改成下面这一段：

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<?xml** version="1.0"**?>**
2. <!DOCTYPE mycat:schema SYSTEM "schema.dtd"**>**
3. **<mycat:schema** xmlns:mycat="http://org.opencloudb/" **>**
5. **<schema** name="mycat" checkSQLschema="false" sqlMaxLimit="100" dataNode="dn1"**/>**
7. **<dataNode** name="dn1" dataHost="virtualHost" database="mk" **/>**
9. **<dataHost** name="virtualHost" maxCon="50" minCon="5" balance="3"
10. writeType="0" dbType="mysql" dbDriver="native" switchType="1" **>**
11. **<heartbeat>**select 1**</heartbeat>**
12. **<writeHost** host="m1" url="192.168.0.101:3306" user="mk" password="aaaaaa"**>**
13. **<readHost** host="s1" url="192.168.0.102:3306" user="mk" password="aaaaaa" **/>**
14. **<readHost** host="s2" url="192.168.0.103:3306" user="mk" password="aaaaaa" **/>**
15. **</writeHost>**
16. **</dataHost>**
18. **</mycat:schema>**

这里有三处需要注意：

****balance="1"与writeType="0" ,switchType=”1”****  
balance 属性负载均衡类型，目前的取值有 4 种：

1. balance="0", 不开启读写分离机制，所有读操作都发送到当前可用的writeHost 上。
2. balance="1"，全部的 readHost 与 stand by writeHost 参与 select 语句的负载均衡，简单的说，当双主双从模式(M1 ->S1 ， M2->S2，并且 M1 与 M2 互为主备)，正常情况下， M2,S1,S2 都参与 select 语句的负载均衡。
3. balance="2"，所有读操作都随机的在 writeHost、 readhost 上分发。
4. balance="3"， 所有读请求随机的分发到 wiriterHost 对应的 readhost 执行,writerHost 不负担读压力，注意 balance=3 只在 1.4 及其以后版本有， 1.3 没有。

****writeType 属性，负载均衡类型，目前的取值有 3 种****：

1. writeType="0", 所有写操作发送到配置的第一个 writeHost，第一个挂了切到还生存的第二个writeHost，重新启动后已切换后的为准，切换记录在配置文件中:dnindex.properties .
2. writeType="1"，所有写操作都随机的发送到配置的 writeHost。
3. writeType="2"，没实现。

****switchType 属性****

1. -1 表示不自动切换
2. - 1 默认值，自动切换
3. 2 基于MySQL 主从同步的状态决定是否切换

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<writeHost** host="m1" url="192.168.0.101:3306" user="mk" password="aaaaaa"**>**
2. **<readHost** host="s1" url="192.168.0.102:3306" user="mk" password="aaaaaa" **/>**
3. **<readHost** host="s2" url="192.168.0.103:3306" user="mk" password="aaaaaa" **/>**
4. **</writeHost>**

这里是配置的我们的myCat后台连接的真实的 1主2从服务器以及它们的连接信息。

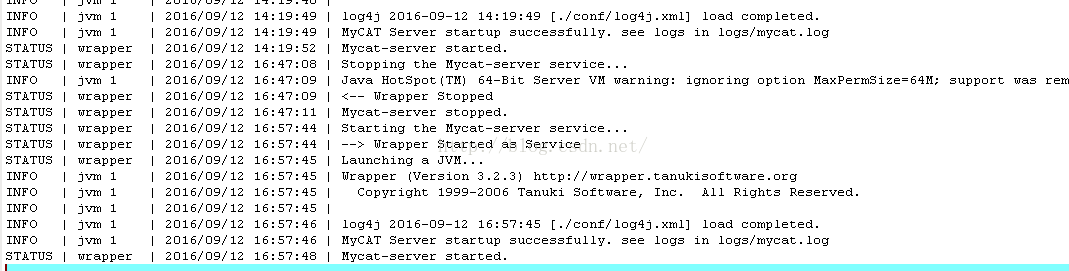
### **测试读写分离**

1. 我们把mycat启动起来

**[plain]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

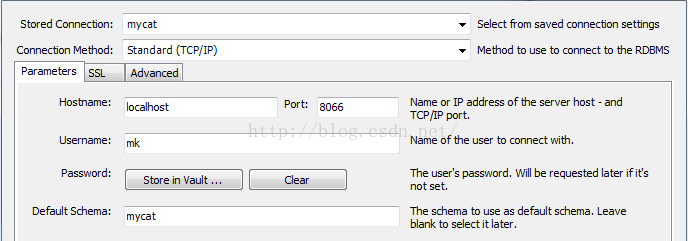
1. ./mycat start

看wrapper.log文件中的内容



看到mycat已经被启动了。‘

2. 于是我们打开一个mysql客户端使用如下方式连接进入mycat

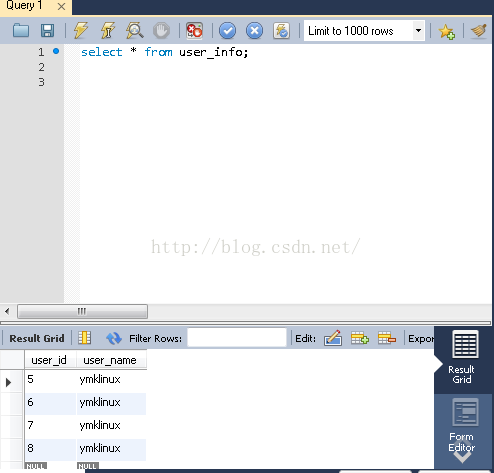


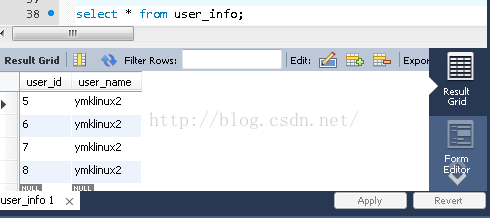
3. 我们往mycat里插入4条数据

**[sql]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **insert** **into** user\_info(user\_name)**values**(@@hostname);
2. **insert** **into** user\_info(user\_name)**values**(@@hostname);
3. **insert** **into** user\_info(user\_name)**values**(@@hostname);
4. **insert** **into** user\_info(user\_name)**values**(@@hostname);
5. **commit**;

单独连上ymklinux和ymklinux2以及ymklinux3分别作



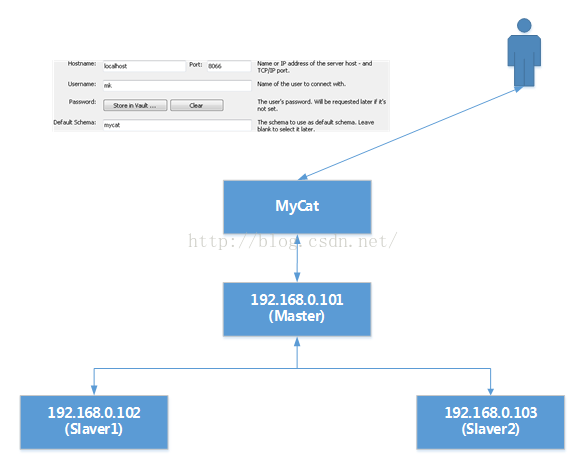


我们可以看到，由于102,103为101的slaver所以当下面这种模式：

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<writeHost** host="m1" url="192.168.0.101:3306" user="mk" password="aaaaaa"**>**
2. **<readHost** host="s1" url="192.168.0.102:3306" user="mk" password="aaaaaa" **/>**
3. **<readHost** host="s2" url="192.168.0.103:3306" user="mk" password="aaaaaa" **/>**
4. **</writeHost>**

writeHost为192.168.0.101（Master）被写入数据时，而又因为192.168.0.101与102, 103为Master-Slaver的关系因此102与103会自动同步192.168.0.101上被insert进入的数据。如下图演示那样。



# **考虑多Master的场景**

我们回到上棕这个场景来看，为什么説需要多Master？

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<writeHost** host="m1" url="192.168.0.101:3306" user="mk" password="aaaaaa"**>**
2. **<readHost** host="s1" url="192.168.0.102:3306" user="mk" password="aaaaaa" **/>**
3. **<readHost** host="s2" url="192.168.0.103:3306" user="mk" password="aaaaaa" **/>**
4. **</writeHost>**

上述这个应用，有一个非致命的缺点：  
即定义的writeHost一旦发生宕机那么其相应对的readHost全部为不可用，换而言之即整个myCat群宕机。

## **熊掌与鱼兼得法**

按照myCat的配置我们可以配置多个writeHost并把writeType这个值设为“0”。  
来看writeType=”0”的含议：  
writeType="0", 所有写操作发送到配置的第一个 writeHost，第一个挂了切到还生存的第二个writeHost，重新启动后已切换后的为准，切换记录在配置文件中:dnindex.properties .

****第一种配置：****

于是我们来考虑下面这样的配置：

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<writeHost** host="m1" url="192.168.0.101:3306" user="root" password="aaaaaa"**>**
2. **</writeHost>**
3. **<writeHost** host="m2" url="192.168.0.104:3306" user="root" password="aaaaaa"**/>**

MyCat群会在Master1发生宕机时自动探寻Master2是否还存活，如果Master2存活那么把数据的读和写全部转向以Master2为代表的读写群。  
但是，这也带来了一个问题。  
如果此时Master1宕机了，Master2被顶了上来，那么数据全部跑入了Master2群内了。  
当：  
Master1再次恢复时。。。对于通过myCat客户端调用者来説这一切是秀明的，数据依然还是那些数据，可是此时，再来一次Master2宕机（刚才是Master1宕机），此时myCat会把Master1作为读写群。  
于是，客户端再次通过myCat代理调用后，会发觉数据有差异了。

****为什么？****

因为Master2内的数据和Master1内的数据没有同步。

****第二种配置：****

于是：

****第一步：****

我们保留第一种配置即

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

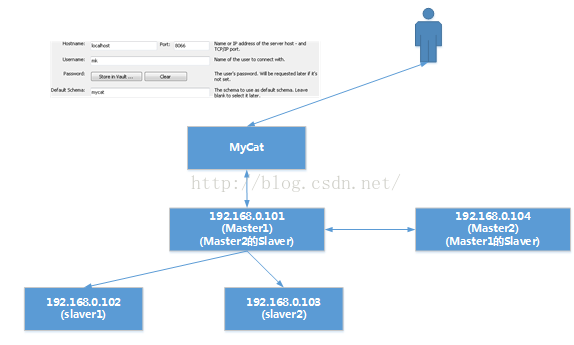
1. **<writeHost** host="m1" url="192.168.0.101:3306" user="root" password="aaaaaa"**>**
2. **</writeHost>**
3. **<writeHost** host="m2" url="192.168.0.104:3306" user="root" password="aaaaaa"**/>**

****第二步：****

我们把Master1和Master2在物理上做成MySQL的主-备结构，现在来看场景推演。  
Master1发生宕机，Master2顶上同时同步了Master1宕机前的数据，对于myCat用户群来説数据everything is ok。  
Master1恢复，Master2宕机客户端发觉通过myCat得到的数据（来自于Master1）有异常，为什么？  
因为Master2是Master1的Slaver，因此它只会“正向同步Master1的数据”，而不能逆向，因此这种配置和第一种配置完全没有区别。  
那么有人説我们把Master1作成Master2的Slaver呢？那无非上第二种情况的场景推演倒一倒而己，还是不能保证任何一点宕机并且在恢复后如何保证数据的强一致性。

****第三种配置：****

于是，我们想到了一个办法，这个办法来源于Master和Slaver在作配置时my.cnf配置文件 中的一个参数，它就是“log-slave-update”。  
对于这个参数的解释，mySQL官方是如下解释的：当你的Master同时又是其它Master’的slaver 时，你需要设置此参数。  
单从上述描述来看这段语句念起来有些晦涩，我们还是实际来看一个架构图吧。



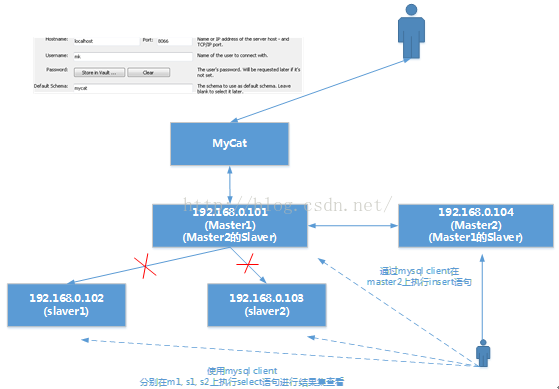
这种结构被称为“双主结构”或者又称为“互为主备”结构。

* Master1是Master2的Slaver；
* Master2是Master1的Slaver；
* 同时，他们又是彼此下面如：Master1拖的S1, S2的Master

# **基于互为主备结构的myCat群搭建方法**

## **第一步(一定要设log-slave-update)**

把Master1作成Master2的Slaver，在Master1配置文件my.cnf中设置log-slave-update。  
如果不设会发生下面这种情况  
1） 通过Master2 insert数据  
2） 在Master1上查看，数据被从Master2上同步过来了  
3） 通过Master1下挂载的它本身的几个Slaver连入并进行查看，结果发觉没有同步Master1的数据  
4） 通过Master1 insert数据  
5） 在Master2上查看，数据被从Master1上同步过来了  
6） 通过Master2下挂载的它本身的几个Slaver连入并进行查看，结果发觉没有同步Master2的数据





## **第二步**

把Master2作成Master1的Slaver，在Master2配置文件my.cnf中设置log-slave-update。  
对于Master1或者是Master2下再外挂的其本身的s1,s2上不需要再设log-slave-update这个开关了。  
如果Master1或者是Master2本身下面还有挂Slaver，请记得此时在其本身的Slaver1上再同步一下它们与Master之间的bin-log。  
因为Master一旦作成了另一个Master的Slaver，因此它的bin-log也改变了，你可以直接在其本身的Slaver上使用change master命令。

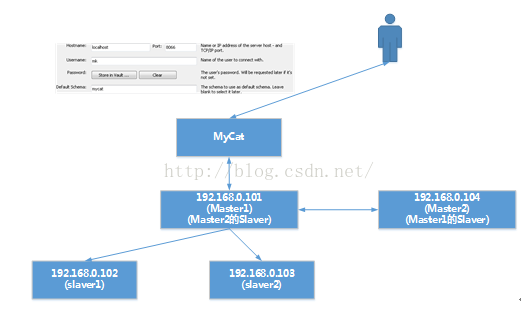
## **第三步**

在myCat的配件文件中作如下设置：

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<dataHost** name="virtualHost" maxCon="50" minCon="5" balance="3"
2. writeType="0" dbType="mysql" dbDriver="native" switchType="2"  slaveThreshold="100"**>**
3. **<heartbeat>**show slave status**</heartbeat>**
4. **<writeHost** host="m1" url="192.168.0.101:3306" user="root" password="aaaaaa"**>**
5. **<readHost** host="s1" url="192.168.0.102:3306" user="root" password="aaaaaa"**/>**
6. **<readHost** host="s2" url="192.168.0.103:3306" user="root" password="aaaaaa"**/>**
7. **</writeHost>**
8. **<writeHost** host="m2" url="192.168.0.104:3306" user="root" password="aaaaaa"**/>**
9. **</dataHost>**

于是，我们就有了这样的结构了



## **测试**

1） 我们连上myCat，对myCat群发送2条insert语句  
2） 我们单独连上m1, s1, s2, m2查看，发觉各个mysql实例中的数据一致  
3） 直接在m1上执行service mysqld stop  
4） 再次连上myCat群，对myCat群再发送2条insert语句  
5） 我们单独连上m2进行查看，现在m2上为4条数据  
6） 我们分别连上s1和s2进行查看，发现s1和s2上还是2条数据  
7） 我们把m1的mysql实例重新启动起来使用命令service mysqld start  
8） 单独连上m1进行查看，发觉m1此时数据条数据为4条  
9）我们分别连上s1和s2进行查看，发现s1和s2上已经从原来的2条数据变为了4条数据

一个高可用的读写分离的集群就此搭建完毕，绝逼完美！

# **mySQL的垂直与水平折分**

在本书系统的《mySQL集群(cluster)》中，我们详细介绍和对比了几种集群的方案，并且在该章节中我详细描述过使用mySQL的分片原则是可以取代集群方案的。并且它比起集群方案来説拥有着更多的优点：

* 廉价，软件免费并且可以使用廉价PC
* 可以大规模铺设不受License的束缚
* 稳定可监控
* 对于已有应用程序来説，它是透明的

## **什么是分片**

一般的replication具有一个限制，即一旦数据库过于庞大，尤其是当写入过于频繁，很难由一台主机支撑的时候，我们还是会面临到扩展瓶颈。  
所谓数据分片(sharding)即是通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）上面，以达到分散单台设备负载的效果。。数据的切分同时还可以提高系统的总体可用性，因为单台设备Crash之后，只有总体数据的某部分不可用，而不是所有的数据。

### **数据的切分（Sharding）模式**

一种是按照不同的表（或者Schema）来切分到不同的数据库（主机）之上，这种切可以称之为数据的垂直（纵向）切分；另外一种则是根据表中的数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上面，这种切分称之为数据的水平（横向）切分。

#### **垂直切分**

一个架构设计较好的应用系统，其总体功能肯定是由很多个功能模块所组成的，而每一个功能模块所需要的数据对应到数据库中就是一个或者多个表。而在架构设计中，各个功能模块相互之间的交互点越统一越少，系统的耦合度就越低，系统各个模块的维护性以及扩展性也就越好。这样的系统，实现数据的垂直切分也就越容易。  
一般来说，如果是一个负载相对不是很大的系统，而且表关联又非常的频繁，那可能数据库让步，将几个相关模块合并在一起减少应用程序的工作的方案可以减少较多的工作量，这是一个可行的方案。一个垂直拆分的例子:

1.用户模块表：user,user\_profile,user\_group,user\_photo\_album  
2.群组讨论表：groups,group\_message,group\_message\_content,top\_message  
3.相册相关表：photo,photo\_album,photo\_album\_relation,photo\_comment  
4.事件信息表：event

群组讨论模块和用户模块之间主要存在通过用户或者是群组关系来进行关联。一般关联的时候都会是通过用户的id或者nick\_name以及group的id来进行关联，通过模块之间的接口实现不会带来太多麻烦；  
相册模块仅仅与用户模块存在通过用户的关联。这两个模块之间的关联基本就有通过用户id关联的内容，简单清晰，接口明确；  
事件模块与各个模块可能都有关联，但是都只关注其各个模块中对象的ID信息，同样可以做到很容易分拆。

****垂直切分的优点****

* 数据库的拆分简单明了，拆分规则明确；
* 应用程序模块清晰明确，整合容易；
* 数据维护方便易行，容易定位；

****垂直切分的缺点****

* 部分表关联无法在数据库级别完成，需要在程序中完成；
* 对于访问极其频繁且数据量超大的表仍然存在性能瓶颈，不一定能满足要求；
* 事务处理相对更为复杂；
* 切分达到一定程度之后，扩展性会遇到限制；
* 过度切分可能会带来系统过渡复杂而难以维护。

#### **水平切分**

将某个访问极其频繁的表再按照某个字段的某种规则来分散到多个表之中，每个表中包含一部分数据。  
对于上面的例子:所有数据都是和用户关联的，那么我们就可以根据用户来进行水平拆分，将不同用户的数据切分到不同的数据库中。  
现在互联网非常火爆的Web2.0类型的网站，基本上大部分数据都能够通过会员用户信息关联上，可能很多核心表都非常适合通过会员ID来进行数据的水平切分。而像论坛社区讨论系统，就更容易切分了，非常容易按照论坛编号来进行数据的水平切分。切分之后基本上不会出现各个库之间的交互。

****水平切分的优点****

表关联基本能够在数据库端全部完成；  
不会存在某些超大型数据量和高负载的表遇到瓶颈的问题；  
应用程序端整体架构改动相对较少；  
事务处理相对简单；  
只要切分规则能够定义好，基本上较难遇到扩展性限制；

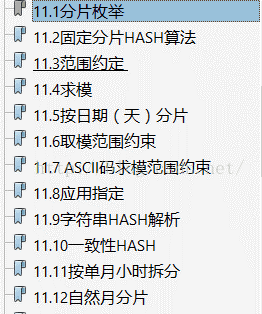
****水平切分的缺点****

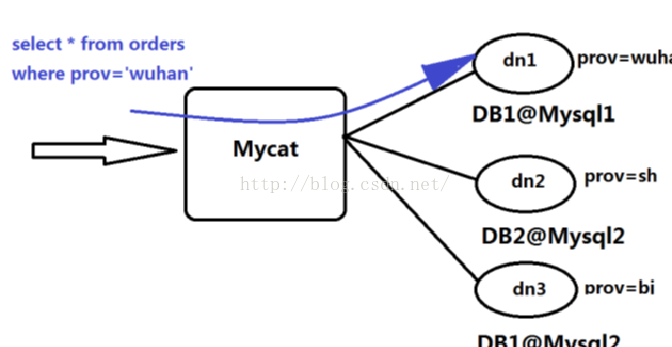
切分规则相对更为复杂，很难抽象出一个能够满足整个数据库的切分规则；  
后期数据的维护难度有所增加，人为手工定位数据更困难；  
应用系统各模块耦合度较高，可能会对后面数据的迁移拆分造成一定的困难。

## **两种切分结合用**

一般来说，我们数据库中的所有表很难通过某一个（或少数几个）字段全部关联起来，所以很难简单的仅仅通过数据的水平切分来解决所有问题。而垂直切分也只能解决部分问题，对于那些负载非常高的系统，即使仅仅只是单个表都无法通过单台数据库主机来承担其负载。我们必须结合“垂直”和“水平”两种切分方式同时使用  
每一个应用系统的负载都是一步一步增长上来的，在开始遇到性能瓶颈的时候，大多数架构师和DBA都会选择先进行数据的垂直拆分，因为这样的成本最先，最符合这个时期所追求的最大投入产出比。然而，随着业务的不断扩张，系统负载的持续增长，在系统稳定一段时期之后，经过了垂直拆分之后的数据库集群可能又再一次不堪重负，遇到了性能瓶颈。  
如果我们再一次像最开始那样继续细分模块，进行数据的垂直切分，那我们可能在不久的将来，又会遇到现在所面对的同样的问题。而且随着模块的不断的细化，应用系统的架构也会越来越复杂，整个系统很可能会出现失控的局面。  
这时候我们就必须要通过数据的水平切分的优势，来解决这里所遇到的问题。而且，我们完全不必要在使用数据水平切分的时候，推倒之前进行数据垂直切分的成果，而是在其基础上利用水平切分的优势来避开垂直切分的弊端，解决系统复杂性不断扩大的问题。而水平拆分的弊端（规则难以统一）也已经被之前的垂直切分解决掉了，让水平拆分可以进行的得心应手。

### **myCat实现数据分片**





在数据切分处理中，特别是水平切分中，中间件最终要的两个处理过程就是数据的切分、数据的聚合。  
选择合适的切分规则，至关重要，因为它决定了后续数据聚合的难易程度，甚至可以避免跨库的数据聚合  
处理。  
  
  
****myCAT数据分片拆分原则：****

* 避免或减少跨库join。
* 选择最合适的拆分维度。

****Mycat拆分表解决方案：****

* MYCAT 全局表
* ER关系
* 表拆分

              --拆分维度  
              --主键分片vs 非主键分片

myCAT对于后端多个mySQL的数据折分和折分后的聚合，对于myCAT前端调用来説是透明的。

## **以实际例子来操作分片**

在分片前，我们需要事先了解一个重要的知识点，也算是一个需要引起重视的问题。  
A和B互为主从，我们按照userID来实施分片，userID为1，3，5的插入到一台mysql实例，userID为2，4，6的插入到另一个mysql实例中去。然后在mycat前端select时mycat会自动进行聚合，即它会从2个mysql实例中选择数据并合并成1,2,3,4,5,6…这样的数据显示给调用的客户端。

1. 先往A上插入 一条数据（id:1 name: tom, gender: m），于是B上也马上会REPLI（同步）一条数据，因此你在A和B上做SELECT查询，都可以看到两边记录一样

2. 往B上插入 一条数据（id:2 name: tom, gender: m），于是A上也马上会REPLI一条数据，因此你在A和B上做SELECT查询，都可以看到两边记录一样

3. 使用MYCAT读写分离，A为写，B为读，一切OK

4. 使用MYCAT做水平拆分，拆分规则为（id % 2 case 0 then Server A; case 1 then Server B;），于是，规则开始起效。

* 当插入id: 2的数据时，它会被insert到A上，接下去由于A和B互为主备，因此B马上会同步一条数据。
* 接下去再插入一条id:3的数据，由于id为3的数据根据我们预先配置后的规则会被插入到Server B上， 然后A马上也会被REPLI一条数据id为3，此时， 此时A和B两个MYSQL上就都有了同样的记录集，都为1,2,3，即：

|  |  |
| --- | --- |
| Server A | Server B |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |

所以数据分片等于没用。再来看分片读，读的规则也走上述规则，于是当你通过mycat作为mysql proxy时你会发生这样的场景：

Mysql –u root –p aaaaaa –P8066 –h 192.168.0.1

Select \* from person  
此时你得到的结果集为：

1  
2  
3  
1  
2  
3  
或者你的ReadServer始终为一台，那你会得到正确的1，2，3这样的结果集。但是，原先所需要的数据的sharding功能。。。其实没有做到。

#### **myCAT分片前置条件（极其重要）**

****因此：****

****如果要考虑后期的水平和垂直折分，被折分的数据库不可以在其本身已经做了主从结构，而是需要把这个主从结构交由mycat来做。****

### **实际分片演示**

我们只需要修改rule.xml这个文件即可实现myCat的数据分片定义了。  
但是按照前面小节的重要提示，在做分片前，我们一定要让我们参于分片规则的mySQL实例保持独立运行状态，即多个mySQL实例间不要做任何的“主备”结构。

#### **实验前准备**

我们拿192.168.0.101与192.168.0.102两台机器来做分片实验

在192.168.0.101上禁用binlog，关闭master slaver的关联，同时断开与原有192.168.0.104的master slaver关联。

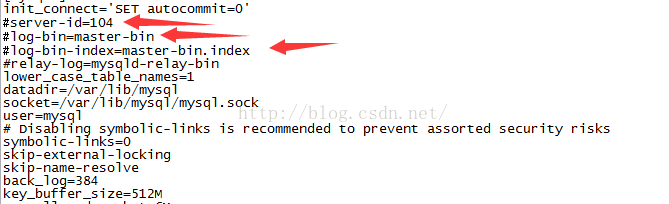
****别忘了把“log-slave-update”也注释掉。****

重启后登录192.168.0.101，输入以下命令

**[plain]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. stop slave;
2. change master to master\_host=' ';

在192.168.0.104上禁用binlog，关闭master slaver的关联，同时断开与原有192.168.0.101的master slaver关联。



****别忘了把“log-slave-update”也注释掉。****

重启后登录192.168.0.104，输入以下命令

**[plain]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. stop slave;
2. change master to master\_host=' ';

#### **配置数据分片**

我们有一张T\_PERSON表，结构如下：

**[sql]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **CREATE** **TABLE** `t\_person` (
2. `person\_id` **int**(11) NOT NULL,
3. `person\_name` **varchar**(20) **DEFAULT** NULL,
4. **PRIMARY** **KEY** (`person\_id`)
5. ) ENGINE=MyISAM **DEFAULT** CHARSET=latin1;
6. show slave status;

数据分片按照PERSON\_ID的奇偶来分。

配置mycat的rule.xml文件

**[plain]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. <?xml version="1.0" encoding="UTF-8"?>
2. <!DOCTYPE mycat:rule SYSTEM "rule.dtd">
3. <mycat:rule xmlns:mycat="http://org.opencloudb/">
4. <tableRule name="mod-long">
5. <rule>
6. <columns>person\_id</columns>
7. <algorithm>mod-long</algorithm>
8. </rule>
9. </tableRule>
10. <function name="mod-long" class="org.opencloudb.route.function.PartitionByMod">
11. <!-- how many data nodes -->
12. <property name="count">2</property>
13. </function>
14. </mycat:rule>

通过配置可以看到我们按照person\_id会把该表数据分在2个物理库内，分片数据包括通过mycat的mysql client端insert时自动分片以及select时自动聚合。

#### **应用分片**

rule.xml文件定义后还需要应用它里面的规则，才能够最终让数据分片在mycat中起效，为此，我们更改schema.xml文件，増加如下一段配置：

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<schema** name="split" checkSQLschema="false" sqlMaxLimit="100"**>**
2. **<table** name="t\_person" dataNode="splitNode1,splitNode2" rule="mod-long"**/>**
3. **</schema>**
4. **<dataNode** name="splitNode1" dataHost="host1" database="mk" **/>**
5. **<dataNode** name="splitNode2" dataHost="host2" database="mk" **/>**
6. **<dataHost** name="host1" maxCon="20" minCon="5" balance="0"
7. writeType="0" dbType="mysql" dbDriver="native" switchType="1" **>**
8. **<heartbeat>**select 1**</heartbeat>**
9. **<writeHost** host="m1" url="192.168.0.101:3306" user="mk" password="aaaaaa"**/>**
10. **</dataHost>**
11. **<dataHost** name="host2" maxCon="20" minCon="5" balance="0"
12. writeType="0" dbType="mysql" dbDriver="native" switchType="1" **>**
13. **<heartbeat>**select 1**</heartbeat>**
14. **<writeHost** host="s1" url="192.168.0.104:3306" user="mk" password="aaaaaa"**/>**
15. **</dataHost>**

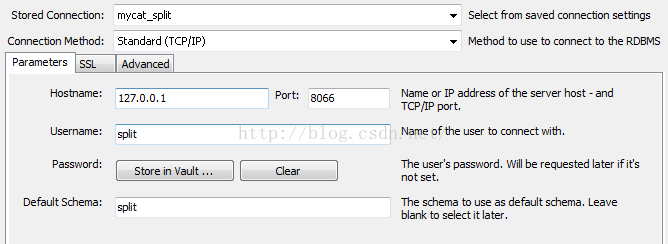
接着我们更改server.xml文件，増加如下一段配置：

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<user** name="split"**>**
2. **<property** name="password"**>**aaaaaa**</property>**
3. **<property** name="schemas"**>**split**</property>**
4. **<property** name="readOnly"**>**false**</property>**
5. **</user>**

#### **测试分片**

我们使用split/aaaaaa用户连上mycat实例

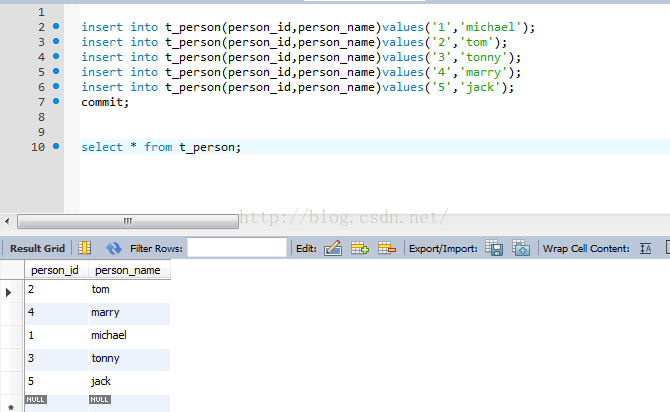


我们插入5条数据

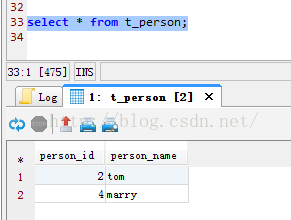
**[sql]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **insert** **into** t\_person(person\_id,person\_name)**values**('1','michael');
2. **insert** **into** t\_person(person\_id,person\_name)**values**('2','tom');
3. **insert** **into** t\_person(person\_id,person\_name)**values**('3','tonny');
4. **insert** **into** t\_person(person\_id,person\_name)**values**('4','marry');
5. **insert** **into** t\_person(person\_id,person\_name)**values**('5','jack');
6. **commit**;

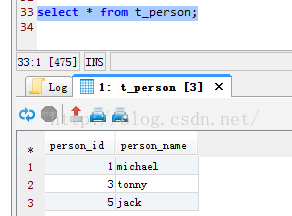
然后我们接着做一次查询



连入192.168.0.101上进行查看，我们可以得到2条数据，person\_id为偶数倍。



连入192.168.0.104上进行查看，我们可以得到3条数据，person\_id为奇数倍。



****分片成功！！！****

myCAT的分片功能相当的强大，完全可以应付亿万级的数据，如果再结合适当的读写分离机制是完全可以让你的网站飞起来的。  
myCAT的分片规则还有很多，文后的附件中会给出一个myCAT分片规则大全。

# **附件 myCAT分片规则大全**

常用的根据主键或非主键的分片规则配置：

## **1. 枚举法**

通过在配置文件中配置可能的枚举id，自己配置分片，使用规则：'

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="sharding-by-intfile"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**hash-int**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="hash-int" class="org.opencloudb.route.function.PartitionByFileMap"**>**
8. **<property** name="mapFile"**>**partition-hash-int.txt**</property>**
9. **<property** name="type"**>**0**</property>**
10. **<property** name="defaultNode"**>**0**</property>**
11. **</function>**
13. partition-hash-int.txt 配置：
14. 10000=0
15. 10010=1
16. DEFAULT\_NODE=1

上面columns 标识将要分片的表字段，algorithm 分片函数，  
其中分片函数配置中，mapFile标识配置文件名称，type默认值为0，0表示Integer，非零表示String，  
所有的节点配置都是从0开始，及0代表节点1

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. /\*\*
2. \*  defaultNode 默认节点:小于0表示不设置默认节点，大于等于0表示设置默认节点
3. \*
4. 默认节点的作用：枚举分片时，如果碰到不识别的枚举值，就让它路由到默认节点
5. \*                如果不配置默认节点（defaultNode值小于0表示不配置默认节点），碰到
6. \*                不识别的枚举值就会报错，
7. \*                like this：can't find datanode for sharding column:column\_name val:ffffffff
8. \*/

## **2. 固定分片hash算法**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="rule1"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**func1**</algorithm>**
5. **</rule>**
6. **</tableRule>**
8. **<function** name="func1" class="org.opencloudb.route.function.PartitionByLong"**>**
9. **<property** name="partitionCount"**>**2,1**</property>**
10. **<property** name="partitionLength"**>**256,512**</property>**
11. **</function>**

配置说明：  
上面columns 标识将要分片的表字段，algorithm 分片函数，  
partitionCount 分片个数列表，partitionLength 分片范围列表  
分区长度:默认为最大2^n=1024 ,即最大支持1024分区  
约束 :  
count,length两个数组的长度必须是一致的。  
1024 = sum((count[i]\*length[i])). count和length两个向量的点积恒等于1024  
用法例子：  
本例的分区策略：希望将数据水平分成3份，前两份各占25%，第三份占50%。（故本例非均匀分区）

**[plain]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. // |<---------------------1024------------------------>|
2. // |<----256--->|<----256--->|<----------512---------->|
3. // | partition0 | partition1 | partition2 |
4. // | 共2份,故count[0]=2 | 共1份，故count[1]=1 |
5. int[] count = new int[] { 2, 1 };
6. int[] length = new int[] { 256, 512 };
7. PartitionUtil pu = new PartitionUtil(count, length);
9. // 下面代码演示分别以offerId字段或memberId字段根据上述分区策略拆分的分配结果
10. int DEFAULT\_STR\_HEAD\_LEN = 8; // cobar默认会配置为此值
11. long offerId = 12345;
12. String memberId = "qiushuo";
14. // 若根据offerId分配，partNo1将等于0，即按照上述分区策略，offerId为12345时将会被分配到partition0中
15. int partNo1 = pu.partition(offerId);
17. // 若根据memberId分配，partNo2将等于2，即按照上述分区策略，memberId为qiushuo时将会被分到partition2中
18. int partNo2 = pu.partition(memberId, 0, DEFAULT\_STR\_HEAD\_LEN);

如果需要平均分配设置：平均分为4分片，partitionCount\*partitionLength=1024

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<function** name="func1" class="org.opencloudb.route.function.PartitionByLong"**>**
2. **<property** name="partitionCount"**>**4**</property>**
3. **<property** name="partitionLength"**>**256**</property>**
4. **</function>**

## **3. 范围约定**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="auto-sharding-long"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**rang-long**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="rang-long" class="org.opencloudb.route.function.AutoPartitionByLong"**>**
8. **<property** name="mapFile"**>**autopartition-long.txt**</property>**
9. **</function>**
10. # range start-end ,data node index
11. # K=1000,M=10000.
12. 0-500M=0
13. 500M-1000M=1
14. 1000M-1500M=2
15. 或
16. 0-10000000=0
17. 10000001-20000000=1

配置说明：  
上面columns 标识将要分片的表字段，algorithm 分片函数，  
rang-long 函数中mapFile代表配置文件路径  
所有的节点配置都是从0开始，及0代表节点1，此配置非常简单，即预先制定可能的id范围到某个分片

## **4. 求模法**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="mod-long"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**mod-long**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="mod-long" class="org.opencloudb.route.function.PartitionByMod"**>**
8. <!-- how many data nodes  -->
9. **<property** name="count"**>**3**</property>**
10. **</function>**

配置说明：  
上面columns 标识将要分片的表字段，algorithm 分片函数，  
此种配置非常明确即根据id进行十进制求模预算，相比方式1，此种在批量插入时需要切换数据源，id不连续

## **5. 日期列分区法**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="sharding-by-date"**>**
2. **<rule>**
3. **<columns>**create\_time**</columns>**
4. **<algorithm>**sharding-by-date**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="sharding-by-date" class="org.opencloudb.route.function.PartitionByDate"**>**
8. **<property** name="dateFormat"**>**yyyy-MM-dd**</property>**
9. **<property** name="sBeginDate"**>**2014-01-01**</property>**
10. **<property** name="sPartionDay"**>**10**</property>**
11. **</function>**

配置说明：  
上面columns 标识将要分片的表字段，algorithm 分片函数，  
配置中配置了开始日期，分区天数，即默认从开始日期算起，分隔10天一个分区  
  
  
  
  
Assert.assertEquals(true, 0 == partition.calculate("2014-01-01"));  
Assert.assertEquals(true, 0 == partition.calculate("2014-01-10"));  
Assert.assertEquals(true, 1 == partition.calculate("2014-01-11"));  
Assert.assertEquals(true, 12 == partition.calculate("2014-05-01"));

## **6. 通配取模**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="sharding-by-pattern"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**sharding-by-pattern**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="sharding-by-pattern" class="org.opencloudb.route.function.PartitionByPattern"**>**
8. **<property** name="patternValue"**>**256**</property>**
9. **<property** name="defaultNode"**>**2**</property>**
10. **<property** name="mapFile"**>**partition-pattern.txt**</property>**
12. **</function>**

partition-pattern.txt   
# id partition range start-end ,data node index  
###### first host configuration  
1-32=0  
33-64=1  
65-96=2  
97-128=3  
######## second host configuration  
129-160=4  
161-192=5  
193-224=6  
225-256=7  
0-0=7  
配置说明：  
上面columns 标识将要分片的表字段，algorithm 分片函数，patternValue 即求模基数，defaoultNode 默认节点，如果配置了默认，则不会按照求模运算  
mapFile 配置文件路径  
配置文件中，1-32 即代表id%256后分布的范围，如果在1-32则在分区1，其他类推，如果id非数据，则会分配在defaoultNode 默认节点

String idVal = "0";  
Assert.assertEquals(true, 7 == autoPartition.calculate(idVal));  
idVal = "45a";  
Assert.assertEquals(true, 2 == autoPartition.calculate(idVal));

## **7.  ASCII码求模通配**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="sharding-by-prefixpattern"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**sharding-by-prefixpattern**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="sharding-by-pattern" class="org.opencloudb.route.function.PartitionByPattern"**>**
8. **<property** name="patternValue"**>**256**</property>**
9. **<property** name="prefixLength"**>**5**</property>**
10. **<property** name="mapFile"**>**partition-pattern.txt**</property>**
12. **</function>**
14. partition-pattern.txt
16. # range start-end ,data node index
17. # ASCII
18. # 48-57=0-9
19. # 64、65-90=@、A-Z
20. # 97-122=a-z
21. ###### first host configuration
22. 1-4=0
23. 5-8=1
24. 9-12=2
25. 13-16=3
26. ###### second host configuration
27. 17-20=4
28. 21-24=5
29. 25-28=6
30. 29-32=7
31. 0-0=7
32. 配置说明：
33. 上面columns 标识将要分片的表字段，algorithm 分片函数，patternValue 即求模基数，prefixLength ASCII 截取的位数
34. mapFile 配置文件路径
35. 配置文件中，1-32 即代表id%256后分布的范围，如果在1-32则在分区1，其他类推
37. 此种方式类似方式6只不过采取的是将列种获取前prefixLength位列所有ASCII码的和进行求模sum%patternValue ,获取的值，在通配范围内的
38. 即 分片数，
39. /\*\*
40. \* ASCII编码：
41. \* 48-57=0-9阿拉伯数字
42. \* 64、65-90=@、A-Z
43. \* 97-122=a-z
44. \*
45. \*/
46. 如
48. String idVal="gf89f9a";
49. Assert.assertEquals(true, 0==autoPartition.calculate(idVal));
51. idVal="8df99a";
52. Assert.assertEquals(true, 4==autoPartition.calculate(idVal));
54. idVal="8dhdf99a";
55. Assert.assertEquals(true, 3==autoPartition.calculate(idVal));

## **8. 编程指定**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="sharding-by-substring"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**sharding-by-substring**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="sharding-by-substring" class="org.opencloudb.route.function.PartitionDirectBySubString"**>**
8. **<property** name="startIndex"**>**0**</property>** <!-- zero-based -->
9. **<property** name="size"**>**2**</property>**
10. **<property** name="partitionCount"**>**8**</property>**
11. **<property** name="defaultPartition"**>**0**</property>**
12. **</function>**
13. 配置说明：
14. 上面columns 标识将要分片的表字段，algorithm 分片函数
15. 此方法为直接根据字符子串（必须是数字）计算分区号（由应用传递参数，显式指定分区号）。
16. 例如id=05-100000002
17. 在此配置中代表根据id中从startIndex=0，开始，截取siz=2位数字即05，05就是获取的分区，如果没传默认分配到defaultPartition

## **9. 字符串拆分hash解析**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. **<tableRule** name="sharding-by-stringhash"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**sharding-by-stringhash**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="sharding-by-substring" class="org.opencloudb.route.function.PartitionDirectBySubString"**>**
8. **<property** name=length**>**512**</property>** <!-- zero-based -->
9. **<property** name="count"**>**2**</property>**
10. **<property** name="hashSlice"**>**0:2**</property>**
11. **</function>**
12. 配置说明：
13. 上面columns 标识将要分片的表字段，algorithm 分片函数
14. 函数中length代表字符串hash求模基数，count分区数，hashSlice hash预算位
16. 即根据子字符串 hash运算


20. hashSlice ： 0 means str.length(), -1 means str.length()-1
22. /\*\*
23. \* "2" -**>** (0,2)**<br/>**
24. \* "1:2" -**>** (1,2)**<br/>**
25. \* "1:" -**>** (1,0)**<br/>**
26. \* "-1:" -**>** (-1,0)**<br/>**
27. \* ":-1" -**>** (0,-1)**<br/>**
28. \* ":" -**>** (0,0)**<br/>**
29. \*/

例子：

**[java]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

1. String idVal=**null**;
2. rule.setPartitionLength("512");
3. rule.setPartitionCount("2");
4. rule.init();
5. rule.setHashSlice("0:2");
6. //      idVal = "0";
7. //      Assert.assertEquals(true, 0 == rule.calculate(idVal));
8. //      idVal = "45a";
9. //      Assert.assertEquals(true, 1 == rule.calculate(idVal));


13. //last 4
14. rule = **new** PartitionByString();
15. rule.setPartitionLength("512");
16. rule.setPartitionCount("2");
17. rule.init();
18. //last 4 characters
19. rule.setHashSlice("-4:0");
20. idVal = "aaaabbb0000";
21. Assert.assertEquals(**true**, 0 == rule.calculate(idVal));
22. idVal = "aaaabbb2359";
23. Assert.assertEquals(**true**, 0 == rule.calculate(idVal));

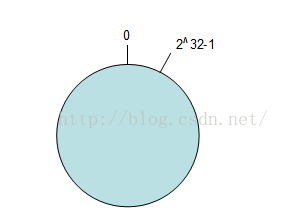
## **10. 一致性hash**

**[html]** [view plain](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "view plain) [copy](http://blog.csdn.net/lifetragedy/article/details/52919984" \o "copy)

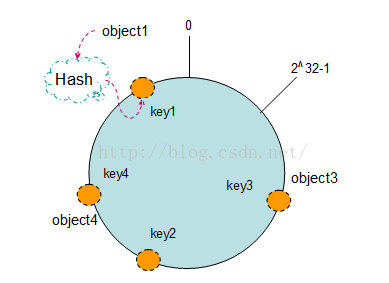
1. **<tableRule** name="sharding-by-murmur"**>**
2. **<rule>**
3. **<columns>**user\_id**</columns>**
4. **<algorithm>**murmur**</algorithm>**
5. **</rule>**
6. **</tableRule>**
7. **<function** name="murmur" class="org.opencloudb.route.function.PartitionByMurmurHash"**>**
8. **<property** name="seed"**>**0**</property>**<!-- 默认是0-->
9. **<property** name="count"**>**2**</property>**<!-- 要分片的数据库节点数量，必须指定，否则没法分片-->
10. **<property** name="virtualBucketTimes"**>**160**</property>**<!-- 一个实际的数据库节点被映射为这么多虚拟节点，默认是160倍，也就是虚拟节点数是物理节点数的160倍-->
11. <!--
12. **<property** name="weightMapFile"**>**weightMapFile**</property>**
13. 节点的权重，没有指定权重的节点默认是1。以properties文件的格式填写，以从0开始到count-1的整数值也就是节点索引为key，以节点权重值为值。所有权重值必须是正整数，否则以1代替 --**>**
14. <!--
15. **<property** name="bucketMapPath"**>**/etc/mycat/bucketMapPath**</property>**
16. 用于测试时观察各物理节点与虚拟节点的分布情况，如果指定了这个属性，会把虚拟节点的murmur hash值与物理节点的映射按行输出到这个文件，没有默认值，如果不指定，就不会输出任何东西 --**>**
17. **</function>**

****一致性hash预算有效解决了分布式数据的扩容问题，前1-9中id规则都多少存在数据扩容难题，而10规则解决了数据扩容难点  
关于一致性hash详细：****

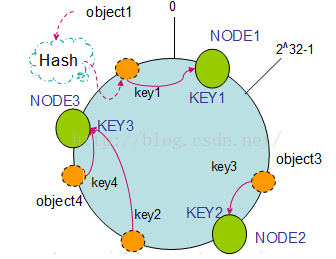
一致性哈希算法在1997年由麻省理工学院提出的一种分布式哈希（DHT）实现算法，设计目标是为了解决因特网中的热点(Hot spot)问题，初衷和CARP十分类似。一致性哈希修正了CARP使用的简 单哈希算法带来的问题，使得分布式哈希（DHT）可以在P2P环境中真正得到应用。   
一致性hash算法提出了在动态变化的Cache环境中，判定哈希算法好坏的四个定义：  
1、平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。  
2、单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。   
3、分散性(Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。   
4、负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。  
在分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的hash(object)%N算法，那么在有机器添加或者删除后，很多原有的数据就无法找到了，这样严重的违反了单调性原则。接下来主要讲解一下一致性哈希算法是如何设计的：  
环形Hash空间  
按照常用的hash算法来将对应的key哈希到一个具有2^32次方个桶的空间中，即0~(2^32)-1的数字空间中。现在我们可以将这些数字头尾相连，想象成一个闭合的环形。如下图



把数据通过一定的hash算法处理后映射到环上  
现在我们将object1、object2、object3、object4四个对象通过特定的Hash函数计算出对  
应的key值，然后散列到Hash环上。如下图：  
    Hash(object1) = key1；  
    Hash(object2) = key2；  
    Hash(object3) = key3；  
    Hash(object4) = key4；

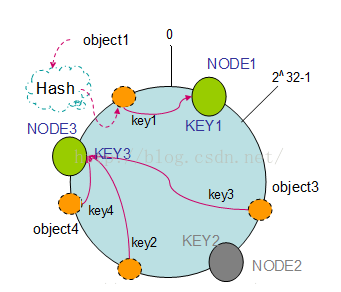


将机器通过hash算法映射到环上  
在采用一致性哈希算法的分布式集群中将新的机器加入，其原理是通过使用与对象存储一样的Hash算法将机器也映射到环中（一般情况下对机器的hash计算是采用机器的IP或者机器唯一的别名作为输入值），然后以顺时针的方向计算，将所有对象存储到离自己最近的机器中。  
假设现在有NODE1，NODE2，NODE3三台机器，通过Hash算法得到对应的KEY值，映射到环中，其示意图如下：  
Hash(NODE1) = KEY1;  
Hash(NODE2) = KEY2;  
Hash(NODE3) = KEY3;

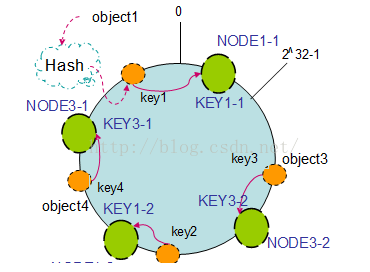


通过上图可以看出对象与机器处于同一哈希空间中，这样按顺时针转动object1存储到了NODE1中，object3存储到了NODE2中，object2、object4存储到了NODE3中。在这样的部署环境中，hash环是不会变更的，因此，通过算出对象的hash值就能快速的定位到对应的机器中，这样就能找到对象真正的存储位置了。

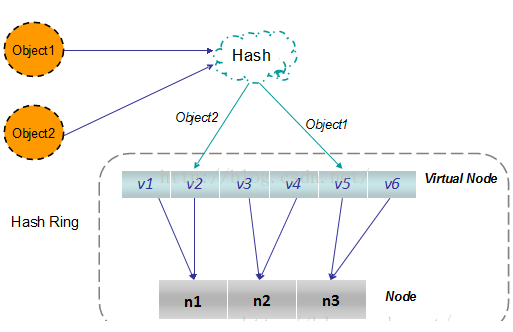
机器的删除与添加  
普通hash求余算法最为不妥的地方就是在有机器的添加或者删除之后会照成大量的对象存储位置失效，这样就大大的不满足单调性了。下面来分析一下一致性哈希算法是如何处理的。  
1. 节点（机器）的删除  
以上面的分布为例，如果NODE2出现故障被删除了，那么按照顺时针迁移的方法，object3将会被迁移到NODE3中，这样仅仅是object3的映射位置发生了变化，其它的对象没有任何的改动。如下图：  
                                                   2. 节点（机器）的添加   
如果往集群中添加一个新的节点NODE4，通过对应的哈希算法得到KEY4，并映射到环中，如下图：



通过按顺时针迁移的规则，那么object2被迁移到了NODE4中，其它对象还保持这原有的存储位置。通过对节点的添加和删除的分析，一致性哈希算法在保持了单调性的同时，还是数据的迁移达到了最小，这样的算法对分布式集群来说是非常合适的，避免了大量数据迁移，减小了服务器的的压力。  
  
  
平衡性  
根据上面的图解分析，一致性哈希算法满足了单调性和负载均衡的特性以及一般hash算法的分散性，但这还并不能当做其被广泛应用的原由，因为还缺少了平衡性。下面将分析一致性哈希算法是如何满足平衡性的。hash算法是不保证平衡的，如上面只部署了NODE1和NODE3的情况（NODE2被删除的图），object1存储到了NODE1中，而object2、object3、object4都存储到了NODE3中，这样就照成了非常不平衡的状态。在一致性哈希算法中，为了尽可能的满足平衡性，其引入了虚拟节点。  
    ——“虚拟节点”（ virtual node ）是实际节点（机器）在 hash 空间的复制品（ replica ），一实际个节点（机器）对应了若干个“虚拟节点”，这个对应个数也成为“复制个数”，“虚拟节点”在 hash 空间中以hash值排列。  
以上面只部署了NODE1和NODE3的情况（NODE2被删除的图）为例，之前的对象在机器上的分布很不均衡，现在我们以2个副本（复制个数）为例，这样整个hash环中就存在了4个虚拟节点，最后对象映射的关系图如下：



根据上图可知对象的映射关系：object1->NODE1-1，object2->NODE1-2，object3->NODE3-2，object4->NODE3-1。通过虚拟节点的引入，对象的分布就比较均衡了。那么在实际操作中，正真的对象查询是如何工作的呢？对象从hash到虚拟节点到实际节点的转换如下图：



 “虚拟节点”的hash计算可以采用对应节点的IP地址加数字后缀的方式。例如假设NODE1的IP地址为192.168.1.100。引入“虚拟节点”前，计算 cache A 的 hash 值：  
Hash(“192.168.1.100”);  
引入“虚拟节点”后，计算“虚拟节”点NODE1-1和NODE1-2的hash值：  
Hash(“192.168.1.100#1”); // NODE1-1  
Hash(“192.168.1.100#2”); // NODE1-2  
****以上所有规则每种都有特定使用场景，可以选择性使用!****