# 1 Dragon简介

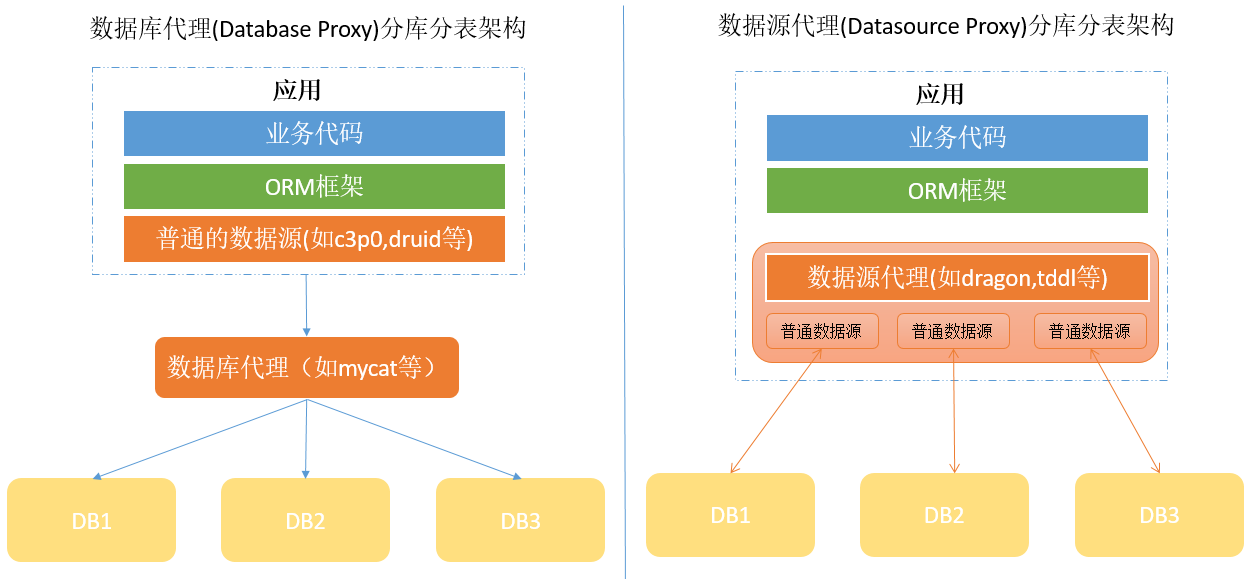
Dragon是一个支持分库分表和读写分离的数据库中间件，以数据源的形式提供这些能力。Dragon实现了JDBC规范定义的接口，可以很容易的与各种ORM框架(如mybatis、hibernate)整合使用。

## 1.1 主流数据库中间件设计方式

目前主流的数据库中间件设计方式有2种：数据库代理(Database Proxy)，数据源代理(Datasource Proxy)。

Dragon采用的是Datasource Proxy的方式提供分库分表，读写分离的功能。

下图对比了数据库代理(Database Proxy)和数据源代理(Datasource Proxy)方式在应用架构上的区别：



以下是关于上图的说明：

1. 数据库代理(Database Proxy)

顾名思义，这种方式是用采用一个中间代理服务器的方式来提供分库分表功能。 应用只需要使用一个普通的数据源如c3p、druid，来与中间代理服务器建立连接，而由代理服务器后面连接多个分库。这样，分库分表对应用来说就是完全屏蔽了。

当执行一个sql时，应用首先将sql交给中间代理服务器。中间代理服务器完成sql的解析，改写，路由，合并结果集的功能，并将合并后的结果集返回给应用。

目前这种方式的实现由很多：如Mysql-proxy、Mycat、Atlas等。

优点：

对应用来说是完全透明的。

缺点：

1. 维护成本高：代理服务器需要单独的机器来部署，增加额外成本。且proxy 需要做集群：由于所有的应用都通过proxy访问数据库，如果只有一个proxy实例，一般其挂了，那么所有的应用都无法访问数据库了。另外通常还需要有专门的人员来进行维护。

2、资源竞争：如果多个不同的应用都进行了分库分表，同时访问同一个proxy，那么彼此必然产生资源竞争。需要做租户隔离。

1. 数据源代理(Datasource Proxy)

数据库代理，代理的是服务端；例如mycat代理多个mysql实例，对外提供分库分表服务。这是从服务端的角度出发的设计思想。

而数据源代理，是从客户端出发的设计思想。普通的数据源(如c3p0，druid，dbcp等)，作为数据库服务的客户端，并不具备分库分表的能力。

数据源代理的思想是，抽象出一个更高层次的数据源，其内部管理多个普通数据源，每个普通数据源与一个分库建立连接。这样就相当于更高层级的数据源管理了所有分库的连接。当要执行一个sql时，首先由这个数据源代理对sql进行解析，判断要操作哪些分库，对sql进行必要的改写，然后选择对应普通数据源去执行sql，最后对多个普通数据源返回的结果进行合并，返回给应用。

目前以数据源代理形式提供分库分表功能的中间件有很多，如： 淘宝的TDDL、大众点评的Zebra、当当网的sharding-jdbc、当然Dragon也是这种形式。

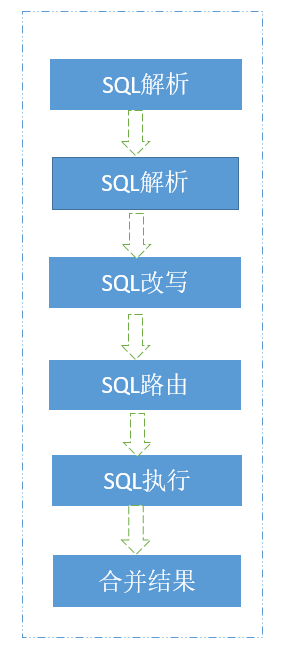
优点：

节省成本：以客户端数据源的形式提供服务，不需要独立的服务器进行部署。

没有资源竞争：因为各个应用都是直连数据库，不会去访问公共的代理，不同的应用之间天然就是隔离的。

缺点：对应用不是完全透明，开发人员需要了解必要的分库分表知识。

总的来说，不管是Database Proxy还是Datasource Proxy，要支持分库分表功能。都要实现以下功能：



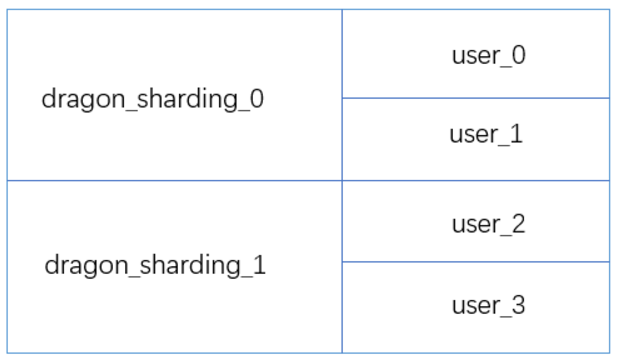
数据库代理，选择的是代理服务端的方式提供分库分表能力。

而数据源代理，选择的是代理以客户端的方式提供分库分表能力。

## 1.2 主流分库分表名称策略

### 1.2.1 sequence-naming

所谓序列命名方式，就是库名和表名按照序列增长的方式进行命名。如下：



即库和表分别按照顺序从0往上依次增加。

这种方式的路由规则如下：

库路由规则：(${id}.toLong()%4).intdiv(2)

表路由规则：${id}.toLong()%4

其中2表示的表的数量，4表示的是库的数量。这是一个模板，如果你有不同的库和表数量，只要替换成你的库和表的数量即可。

### 1.2.2 every-naming

### 1.2.3 scalable-naming

## 1.3 dragon 与 tddl ，zebra对比

前面提到了，现在已经现在已经有了一些以数据源代理(DataSource Proxy)提供分库分表、读写分离功能的数据库中间件。但是dragon并不是重复造轮子，主要是看到了现有的数据库中间件架构或者功能上的缺陷，因此才有了dragon这个项目。

以下是dragon,tddl,zebra的功能对比。需要注意的是，这里是与tddl、zebra目前已经开源的版本进行对比，有些功能在内部版本中可能已经实现。

### 1.2.1 架构对比

Tddl架构

### 1.2.2 可扩展性对比

### 1.2.3 sql语法支持对比

主要比较的是对mysql语法功能的支持。

INSERT操作：

**单条插入语法**，如insert into user(id,name) values(1，”tianshouzhi”);

tddl、zebra，dragon均支持。

**批量插入语法**，如insert into user(id,name) values(1，”tianshouzhi”),(2,”wangxiaoxiao”);

dragon 完全支持，即支持插入的记录落到多个分表中，但是如果要插入到读个分库不考虑事务一致性。

zebra有限支持：主要指的是，批量插入的记录可能要落到不同的分库分表中，但是zebra要求批量插入的记录必须要是在同一个分表。这样的好处是，不需要考虑某个分库失败，事务回滚的问题。

tddl不支持。

on duplicate key update语法：

除了insert语句，其他的几种类型sql语法select、update、delete都要可能会有一些条件，下面列出首先列出对各种查询条件的支持。

查询条件:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dragon | Tddl | Zebra |
| = | 支持,且可以作为分区字段 | 支持,且可以作为分区字段 | 支持,且可以作为分区字段 |
| In | 支持,且可以作为分区字段 | 支持,且可以作为分区字段 | 支持,且可以作为分区字段 |
| not in | 支持 |  |  |
| Between…and | 支持 |  |  |
| Like | 支持 | 支持 |  |
| not like | 支持 | 支持 |  |
| > 、>=、<、<=，!= | 支持 |  |  |
| Is | 支持 | 支持 |  |
| Is not | 支持 |  |  |

UPDATE操作:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dragon | Tddl | Zebra |
| 单表单条记录更新 | 支持 | 支持 | 支持 |
| 单表批量更新基于in | 支持 | 支持 | 支持 |
| 单表批量更新基于case when | 支持 |  |  |

Select操作：

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dragon | Tddl | Zebra |
| Left join、inner join、right join | 支持 | 有限支持，需要传入hint |  |
| Order by | 支持 | 支持 | 支持 |
| Limit | 支持 | 支持 | 支持 |
| GROUP BY | 支持 |  | 有限支持，必须指定别名 |

聚合函数

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dragon | Tddl | Zebra |
| MAX | 支持 | 支持直接使用列名，如max(id) | 支持直接使用列名 |
| MIN | 支持 | 支持直接使用列名 | 支持直接使用列名 |
| SUM | 支持 | 支持直接使用列名 | 支持直接使用列名 |
| COUNT | 支持 | 支持直接使用列名 | 支持直接使用列名 |
| AVG |  |  |  |
| 混合使用 | 支持 |  |  |

子查询：

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dragon | Tddl | Zebra |
| Where部分子查询 | 支持,不要求表名一致,且可以识别分区字段(对于一些特殊语义不做处理,例如子查询中包含聚合函数) | 支持, 要求嵌套查询内的所有表名全部一致, 同时还需要注意的是，嵌套查询中分库分表的字段只能够出现在一个嵌套层级里，否则会抛异常 | 支持子查询，但无法识别子查询中的分区字段 |
| From部分子查询 |  |

事务：

|  |  |  |
| --- | --- | --- |
| Dragon | Tddl | Zebra |
| 支持单库内的事务 | 支持单库内的事务 |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dragon | Tddl | Zebra |
| 分库分表 | 支持 | 支持 | 支持 |
| 可选ha数据源 | 完全支持，既可以使用DragonHADatasource，也可以直接使用任意其他数据源，如druid、c3p0 | 有限支持，TDataSource强依赖TGroupDataSource |  |
| 第三方数据源整合 | 完全支持，只需配置即可 | 需要自行编写代码 |  |
| 第三方数据源配置属性支持 | 完全支持 | 有限支持，因为tddl定义了atom层标准属性，其他对属性的支持写死在代码里，用户不可自行设置属性 |  |
| 配置分布式自增id生成器 | 有 | 无 | 无 |
| 动态配置中心 | 自带一个，并支持整合任意动态配置中心 | 强依赖diamond |  |
| 监控 |  |  |  |
| 管理后台 |  |  |  |
| 配置复杂性 | 简单 | 极端复杂 |  |

路由规则配置

|  |  |  |
| --- | --- | --- |
|  | 语言 | 备注 |
| Dragon | Groovy | 完全的groovy脚本 |
| Tddl | Groovy | 限制只能使用一行 |
| Zebra | Groovy |  |

Dragon比tddl提供了更好的默认条件，在没有路由条件的情况下，tddl是随机选择一个分库执行sql，dragon的默认行为是没有路由条件，路由到所有库查询。

为了效率，应该只有统计功能路由到所有库，因为统计函数返回的记录较少，虽然dragon目前并没有对此限制，可以直接查询所有库的所有记录。

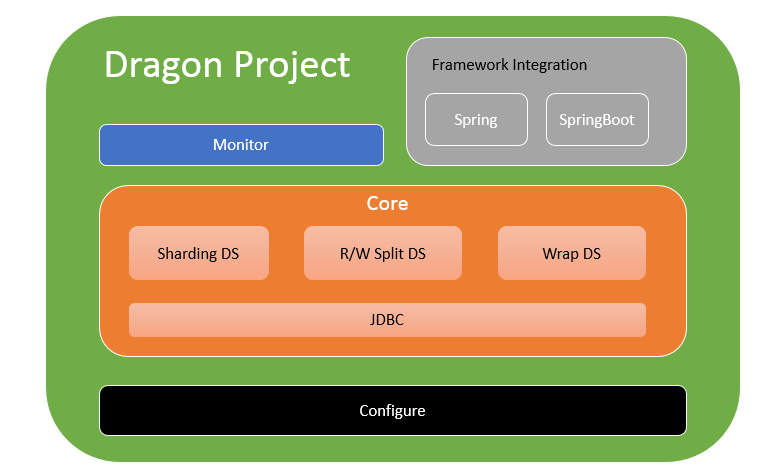
### 1.2.4 配置方式对比(重点配置简易和数据迁移问题到)

# 3、Drangon项目设计

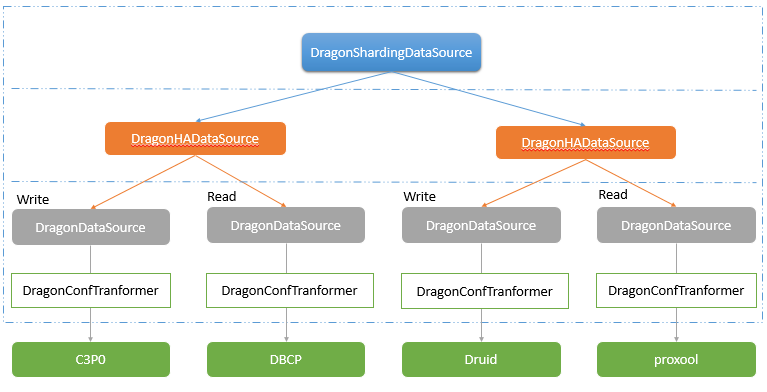
DragonDataSource是一个支持读写分离、分库分表的数据源，实现了JDBC规范。

Drangon项目名称的由来，通常支持分库分表，比支持读写分离复杂的多，而在实现分库分表的过程中，最复杂的部分莫过于sql-parser，这需要掌握一些编译原理相关的知识，编译原理三大圣经中的龙(Dragon)书，是必备的参考书籍，因此项目起名为Dragon。

## 3.1 整体架构



## 3.2 数据源设计



其中：

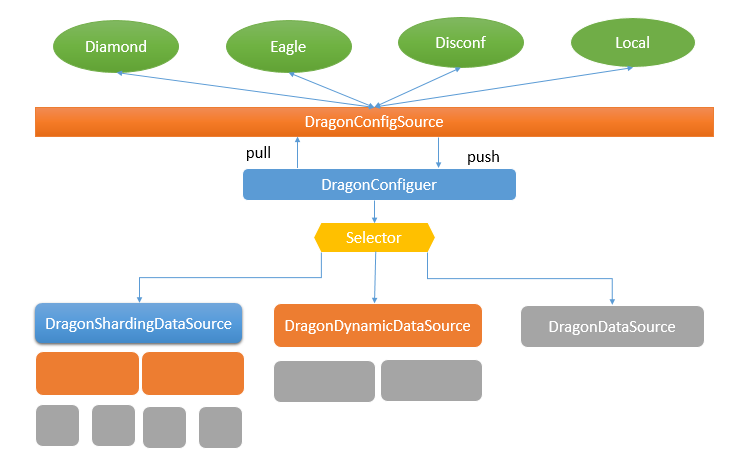
**DragonShardingDataSource**是用于支持分库分表的数据源

**DragonHADataSource**是用于支持读写分离的数据源

**DragonDataSource**是一个真实物理数据源的wrapper(包装设计模式)。

**DragonConfTranformer**是一个数据源配置的转换器。由于每个真实的物理数据源的配置项可能各不相同，DragonDataSource会定一些标准的配置项，通过DragonConfTranformer将标准的配置项转换为某个具体实际数据源的配置项。要支持某个真实的物理数据源，只要增加对应的DragonConfTranformer实现即可。(桥接设计模式)

## 3.3 配置模块



Dragon针对于配置，抽象出一个配置源(DragonConfigSource)的概念，不论配置是放在本地文件中，还是放在远程，Dragon并不关心，因为Dragon是从DragonConfigSource中来获取配置的信息。

DragonConfigSource需要支持通过pull和push的方式获取配置：

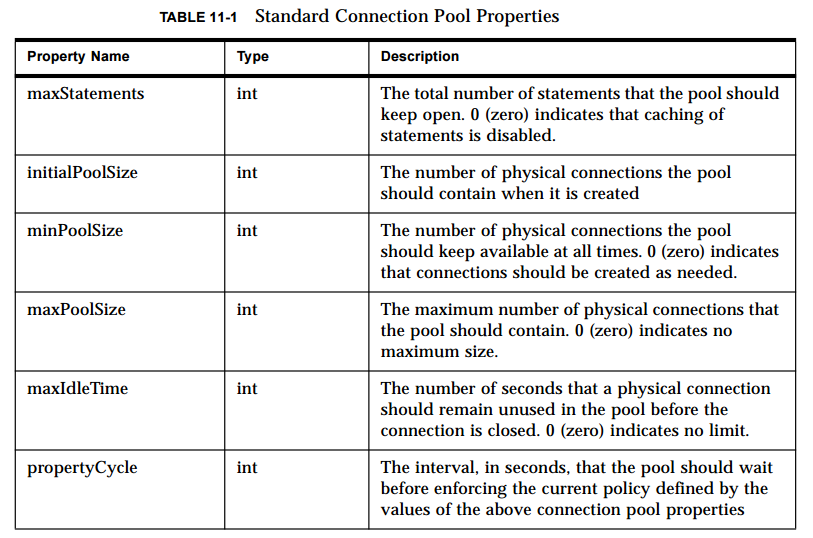
* pull方式指的是主动去外部数据源拉取配置，在Dragon启动的时候，需要主动pull。
* push方式指的是外部数据源主动向Dragon推送配置，例如想再运行时修改配置。

在启动的时候，Dragon会根据主动pull的配置信息，通过DragonConfiguer进行解析，Selector根据解析的结果选择到底初始化哪一个数据源。例如：如果配置信息是分库分表的配置，那么就初始化DragonShardingDataSource；如果是读写分离的配置信息，就初始化DragonHADataSource。

当Dragon在运行时接受到push的配置信息时，会首先解析出变化的部分，例如一个DragonHADataSource下只是某一个DragonConfigSource的配置发生了变化，只将变化的部分传递过来，然后通过Selector选择对应的DragonConfigSource，将变化的部分配置交给其处理。

# 4 DragonDataSource设计

## 4.1 JDBC 3.0规范中的标准连接池属性



除了这些标准属性，一般数据库连接池还会提供以下属性：

url、

driverClassName、

username、

password、

Extra config，由具体的DragonConfTransformer实现完成附加属性的配置。

# 5 DragonHADataSource

## 5.1 DragonHADataSource的使用

因为DragonHADataSource的功能是做读写分离，因此首先我们要搭建一个读写分离的环境。

### 5.1.1 读写分离环境搭建

本节利用mysql\_multi方式搭建一个读写分离环境。也就是在一个机器上启动多个mysql实例，在这多个实例上建立主从关系。步骤如下:

1、下载MySQL压缩包

|  |
| --- |
| wget http://dev.MySQL.com/get/Downloads/MySQL-5.7/mysql-5.7.18-linux-glibc2.5-x86\_64.tar.gz |

2、创建安装目录

这里我将Mysql安装在/usr/local/mysql目录里面，也可以安装在其他地方;

|  |
| --- |
| mkdir /usr/local/mysql |

3、解压并复制

|  |
| --- |
| tar -xvf mysql-5.7.18-linux-glibc2.5-x86\_64.tar.gz  mv mysql-5.7.18-linux-glibc2.5-x86\_64/\* /usr/local/mysql/ |

4、创建data目录

|  |
| --- |
| mkdir /usr/local/mysql/data{3306,3307,3308} -p |

这里创建了3个目录，意味着我们要在服务器上同时启动3个mysql实例，通过不同的目录将这3个实例的数据进行隔离。

5、创建mysql用户组及其用户

|  |
| --- |
| groupadd mysql  useradd -r -g mysql mysql |

**6初始化实例的数据库**

mysql\_install\_db 方式已经 deprecated 了，我们使用 mysqld 的 initialize 方法来做初始化

|  |
| --- |
| # --no-defaults 不读取默认的 /etc/my.cnf 全局配置文件 否则可能存在一些冲突问题  # --initialize-insecure 初始化且不需要生成密码，我不太喜欢那一大串字符...  # --basedir mysql 的安装目录  # --datadir 本实例的数据目录  # --user 这样生成的文件用户为 mysql  # --explicit\_defaults\_for\_timestamp timestamp 已经 deprecated 了  #生成 3306 的数据目录  [root@localhost ~]# mysqld --no-defaults \  --initialize-insecure \  --basedir=/usr/local/mysql \  --datadir=/usr/local/mysql/data3306 \  --user=mysql \  --explicit\_defaults\_for\_timestamp  #生成 3307 的数据目录  [root@localhost ~]# mysqld --no-defaults \  --initialize-insecure \  --basedir=/usr/local/mysql \  --datadir=/usr/local/mysql/data3307 \  --user=mysql \  --explicit\_defaults\_for\_timestamp  #生成 3308 的数据目录  [root@localhost ~]# mysqld --no-defaults \  --initialize-insecure \  --basedir=/usr/local/mysql \  --datadir=/usr/local/mysql/data3308 \  --user=mysql \  --explicit\_defaults\_for\_timestamp |

7、将mysql命令加入环境变量

在/etc/profile最后加入两行命令：

|  |
| --- |
| MYSQL\_HOME=/usr/local/mysql  export PATH=$PATH:$MYSQL\_HOME/bin |

**8、在/etc目录下创建 my.cnf文件，配置如下**

|  |
| --- |
| [mysqld\_multi]  mysqld=/usr/local/mysql/bin/mysqld\_safe #mysqld命令的位置，用于启动mysql实例，也可以指定为mysqld\_safe命令的位置  mysqladmin = /usr/local/mysql/bin/mysqladmin #用于停止mysql实例  user=root #调用mysqladmin时使用的账号  #password=123456 调用 mysqladmin时使用的账号的密码，因为我们使用了initialize-insecure，所以root用户最开始是没有密码的  log=/usr/local/mysql/mysql\_multi.log  [mysqld3306]  basedir=/usr/local/mysql  datadir=/usr/local/mysql/data3306  port=3306  socket=/tmp/mysql.sock3306  user=mysql  server\_id=1  log\_bin=mysql-bin #配置了这个选项，说明3306库作为主库  [mysqld3307]  basedir=/usr/local/mysql  datadir=/usr/local/mysql/data3307  port=3307  socket=/tmp/mysql.sock3307  user=mysql  server\_id=2  [mysqld3308]  basedir=/usr/local/mysql  datadir=/usr/local/mysql/data3308  socket=/tmp/mysql.sock3308  port=3308  user=mysql  server\_id=3 |

提示：

1、在mysqld\_multi中配置的user和password是mysql的用户名和密码。为了使用mysqld\_multi命令，**每个mysql实例都必须要提供一个相同的用户名和密码，用于启动和停止服务**。而且要确保这个用户具有停止mysql实例的权限。

2、在[mysqlN]中配置的用户是linux中的用户，是启动mysql服务实例时使用的用户

**mysqld\_multi的使用语法如下:**

|  |
| --- |
| shell> mysqld\_multi [options] {start|stop|reload|report} [GNR[,GNR] ...] |

start, stop, reload (stop and restart)和report 表示要执行的不同操作。你可以针对某个mysql实例执行某个操作，也可以针对多个mysql实例执行某个操作，这依赖于列在命令后的GNR是哪些。如果没有列出任何GNR，则这个操作针对所有mysql实例。需要注意的是GNR列表中，不能有空格。

举例

|  |
| --- |
| [root@localhost mysql]# **mysqld\_multi start**  [root@localhost mysql]# **mysqld\_multi report**  Reporting MySQL servers  MySQL server from group: mysqld3307 is running  MySQL server from group: mysqld3308 is running  [root@localhost mysql]# **mysqld\_multi stop**  [root@localhost mysql]# **mysqld\_multi report**  Reporting MySQL servers  MySQL server from group: mysqld3307 is not running  MySQL server from group: mysqld3308 is not running |

9配置主从关系

为mysql服务器配置复制非常简单，由于场景不同，略微会有所差异

1，在每台server上创建复制帐号

2，配置主库和备库

3，通知备库连接到主库并从主库复制数据

**创建复制账号**

MySQL会给复制线程赋予一些特殊的权限，备库的I/O线程会建立一个到主库的TCP/IP连接，因此必须由主库提供一个用户，并赋予其合适的权限。备库IO线程以该用户名连接到主库并读取其二进制文件。通过如下语句创建账号：

|  |
| --- |
| GRANT REPLICATION SLAVE, REPLICATION CLIENT ON \*.\* TO slave@'localhost' IDENTIFIED BY 'slave'; |

我们在主库和备库都创建该账号。

复制账户事实上只需要有主库上的REPLICATION SLAVE权限，并不一定需要每一端服务器都有REPLICATION CLIENT权限，那么为什么我们要把这两种权限给主/备库都赋予呢？这有两个原因：

1. 用来监控和管理复制的账号需要REPLICATION CLIENT权限，并且针对这两种目的使用同一个账号更加容易。

2. 如果在主库上建立了账号，然后从主库将数据克隆到备库上时，备库也就设置好了——变成主库所需要的配置。这样后续有需要可以方便地交换主备库的角色。

mysql -h127.0.0.1 -uroot -P3307 -p **(注意指定端口登录时，主机名和端口号必须同时指定，否则登录的还是3306端口)**

show variables like 'datadir';

mysql -h127.0.0.1 -uroot -P3308 -p **(注意指定端口登录时，主机名和端口号必须同时指定，否则登录的还是3306端口)**

show variables like 'datadir';

**启动复制**

下一步是告诉备库如何连接到主库并重复其二进制日志,这一步不需要通过修改my.cnf来配置,不要在my.cnf中配置master\_port或master\_host，这些老的配置方式已经废弃。而是使用CHANGE MASTER TO 语句,该语句完全替代了my.cnf 中相应的设置,并且允许以后指向别的主库时无须重启备库。下面是复制开始的基本命令：

|  |
| --- |
| mysql>CHANGE MASTER TO \          MASTER\_HOST='localhost',\          MASTER\_USER='slave',\          MASTER\_PORT=3307,\          MASTER\_PASSWORD='slave',\          MASTER\_LOG\_FILE='mysql-bin.000001',\          MASTER\_LOG\_POS=0;  mysql> START SLAVE; |

只需要配置一次即可，之后重启的时候，slave会自动连接到master同步数据，不需要每次都执行这些命令。

10在作为主库的mysql上建库建表，这些库和表信息会自动同步到从库上:

|  |
| --- |
| **CREATE DATABASE** `dragon\_ha`;  **use** dragon\_ha; **CREATE TABLE** `user` (  **`id` int**(11) **NOT NULL AUTO\_INCREMENT**,  **`name` varchar**(255) **NOT NULL**,  **PRIMARY KEY** (**`id`**) ) **ENGINE**=InnoDB **DEFAULT CHARSET**=utf8; |

### 5.1.2 使用DragonHADatasource完成读写分离功能

本节通过一个完整的案例讲解的是DragonHADataSource与mybatis、spring的整合使用。

新建maven项目，pom依赖如下：

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>com.tianshouzhi</**groupId**>  <**artifactId**>dragon-ha-datasource</**artifactId**>  <**version**>0.0.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>4.2.7.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-jdbc</**artifactId**>  <**version**>4.2.7.RELEASE</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis-spring</**artifactId**>  <**version**>1.3.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>3.4.0</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-api</**artifactId**>  </**dependency**>  <**dependency**>  <**groupId**>ch.qos.logback</**groupId**>  <**artifactId**>logback-core</**artifactId**>  <**version**>1.1.7</**version**>  </**dependency**>  <**dependency**>  <**groupId**>ch.qos.logback</**groupId**>  <**artifactId**>logback-classic</**artifactId**>  </**dependency**> </**dependencies**> |

新建Java类com.tianshouzhi.dragon.domain.User;

|  |
| --- |
| **package** com.tianshouzhi.dragon.domain;  **public class** User {  **private int id**;  **private** String **name**;   **public int** getId() {  **return id**;  }   **public void** setId(**int** id) {  **this**.**id** = id;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", name='"** + **name** + **'\''** +  **'}'**;  } } |

定义映射文件classpath:mybatis/mappers/UserMapper.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>*** <**mapper namespace=" com.tianshouzhi.dragon.mappers.UserMapper"**>   <**insert id="insert" useGeneratedKeys="true" keyColumn="id" keyProperty="id" parameterType="com.tianshouzhi.mybatis.quickstart.User"**>  INSERT INTO user(name) VALUES (#{name})  </**insert**>   <**select id="selectById" parameterType="int" resultType="com.tianshouzhi.mybatis.quickstart.User"**>  select id,name from user where id= #{id}  </**select**>   <**update id="updateById" parameterType="com.tianshouzhi.mybatis.quickstart.User"**>  UPDATE user SET name=#{name} WHERE id=#{id}  </**update**>   <**delete id="deleteById" parameterType="int"**>  DELETE FROM user WHERE id=#{id}  </**delete**>  </**mapper**> |

定义com.tianshouzhi.dragon.mappers.UserMapper接口

|  |
| --- |
| **package** com.tianshouzhi.dragon.mappers;  **import** com.tianshouzhi.dragon.domain.User;  **public interface** UserMapper {  **public int** insert(User user);   **public** User selectById(**int** id);   **public int** updateById(User user);   **public int** deleteById(**int** id); } |

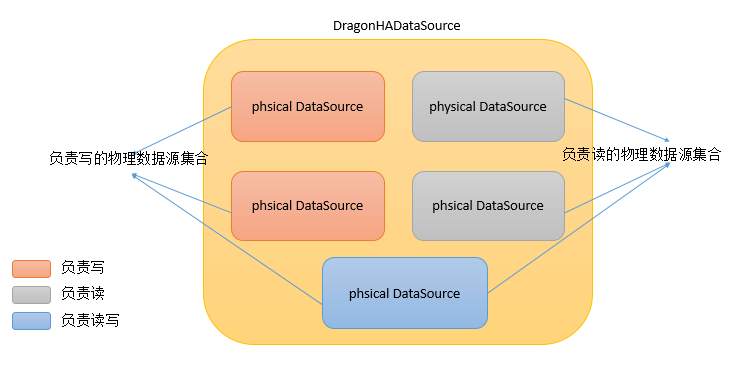
创建mybatis与spring整合配置文件classpath：ha/dragon-ha-mybatis-spring.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"**> *<!--==========================================DragonHADataSource配置=================================================-->  <!--定义主库和从库数据源共同的属性，这里使用的是durid数据源，注意这个bean的abstract=true  对于主库和从库不同的配置，例如url、username、password等，单独进行配置  -->* <**bean id="druidParent" class="com.alibaba.druid.pool.DruidDataSource" abstract="true"**>  <**property name="driverClassName" value="com.mysql.jdbc.Driver"**></**property**>  <**property name="initialSize" value="10"**></**property**>  <**property name="maxActive" value="50"**></**property**>  <**property name="maxWait" value="60000"**></**property**>  <**property name="minIdle" value="1"** />  <**property name="testWhileIdle" value="true"** />  <**property name="testOnBorrow" value="false"** />  <**property name="testOnReturn" value="false"** />  <**property name="validationQuery" value="select 'x'"**></**property**>  </**bean**>   *<!--主库数据源配置，这里设置parent="druidParent"，因此druidParent这个bean中配置的属性都会被继承-->* <**bean id="master" parent="druidParent"**>  <**property name="username" value="root"**></**property**>  <**property name="password" value="shxx12151022"**></**property**>  <**property name="url" value="jdbc:mysql://localhost:3306/dragonr?characterEncoding=UTF-8&amp;useSSL=false"**></**property**>  </**bean**>   *<!--从库数据源配置，一般从来从库配置的用户名/密码，只具有只读权限-->* <**bean id="slave1" parent="druidParent"**>  <**property name="username" value="root"**></**property**>  <**property name="password" value="shxx12151022"**></**property**>  <**property name="url" value="jdbc:mysql://localhost:3307/dragon\_ha?characterEncoding=UTF-8&amp;useSSL=false"**></**property**>  </**bean**>   <**bean id="slave2" parent="druidParent"**>  <**property name="username" value="root"**></**property**>  <**property name="password" value="shxx12151022"**></**property**>  <**property name="url" value="jdbc:mysql://localhost:3308/dragon\_ha?characterEncoding=UTF-8&amp;useSSL=false"**></**property**>  </**bean**>   *<!--定义DragonHADatasource，前面定义的几个主库和从库数据源都要交给DragonHADatasource进行管理-->* <**bean id="dragon-ha-datasource" class="com.tianshouzhi.dragon.ha.jdbc.datasource.DragonHADatasource"**>  *<!--  将之前定义的主、从数据源都保存到一个Map中。  Map的key：是为这个数据源起的一个名字，没有特殊含义，主要是为了方便调试。一般与bean的id保持一致。  Map的Value：类型是DatasourceWrapper，其实是对前面配置的主从数据源进行的一层封装，封装的目的是为了配置读写权重  -->* <**property name="indexDsMap"**>  <**map**>  *<!--将主库数据源交给DragonHADatasource管理，注意这里的属性key=master只是一个名字在而已，并不是真正代表主库。  DatasourceWrapper的physicalDataSource属性引用的bean，如果是主库，才表示这个entry真正的是主库数据源-->* <**entry key="master"**>  <**bean class="com.tianshouzhi.dragon.ha.jdbc.datasource.dbselector.DatasourceWrapper"**>  *<!--注意这里的ref="master"才真正表示的是引用前面定义的主库数据源-->* <**property name="physicalDataSource" ref="master"**></**property**>  *<!--写权重：主库应该是可写的，因此写权重>0-->* <**property name="writeWeight" value="10"**></**property**>  *<!--读权重: 如果希望读操作不走主库，将读权重设置为0。需要注意的是，即使读权重为0，对于开启事务或者使用了hint的情况，还是走主库-->* <**property name="readWeight" value="0"**></**property**>   </**bean**>  </**entry**>  *<!--从库配置，一般来说，从库都是只读的，因此writeWeight=0。  这里配置了两个从库，readWeight值都为10，二者比例为1:1。表示当有一个查询请求时，在确定从哪一个从库进行查询时，二者选中的比例都是50%-->* <**entry key="slave1"**>  <**bean class="com.tianshouzhi.dragon.ha.jdbc.datasource.dbselector.DatasourceWrapper"**>  <**property name="physicalDataSource" ref="slave1"**></**property**>  <**property name="readWeight" value="10"**></**property**>  <**property name="writeWeight" value="0"**></**property**>  </**bean**>  </**entry**>  <**entry key="slave2"**>  <**bean class="com.tianshouzhi.dragon.ha.jdbc.datasource.dbselector.DatasourceWrapper"**>  <**property name="physicalDataSource" ref="slave2"**></**property**>  <**property name="readWeight" value="10"**></**property**>  <**property name="writeWeight" value="0"**></**property**>  </**bean**>  </**entry**>  </**map**>  </**property**>  </**bean**>   *<!--=====================================Mybatis Spring整合配置=================================-->* <**bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>  *<!--注意这里使用数据源是dragon-ha-datasource，因为DragonHADataSource实现了jdbc规范定义的接口，因此可以直接交给SqlSessionFactory进行管理-->* <**property name="dataSource" ref="dragon-ha-datasource"**></**property**>  *<!--指定Mapper映射文件的位置-->* <**property name="mapperLocations" value="classpath\*:mybatis/mappers/\*\*/\*.xml"**></**property**>  <**property name="typeAliasesPackage" value="com.tianshouzhi.dragon.domain"**></**property**>  </**bean**>  <**bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"**>  *<!--指定Mapper接口类所在的包-->* <**property name="basePackage" value="com.tianshouzhi.dragon.mappers"** />  *<!--指定SqlSessionFactoryBean这个bean的id属性-->* <**property name="sqlSessionFactoryBeanName" value="sqlSessionFactory"** />  </**bean**> </**beans**> |

建立测试类：

|  |
| --- |
| **package** com.tianshouzhi.dragon.ha;  **import** com.tianshouzhi.dragon.domain.User; **import** com.tianshouzhi.dragon.mappers.UserMapper; **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext;  **public class** DragonHAMybatisSpringTest {  **public static void** main(String[] args) {  ApplicationContext context=**new** ClassPathXmlApplicationContext(**"classpath:ha/dragon-ha-mybatis-spring.xml"**);  UserMapper userMapper = context.getBean(UserMapper.**class**);  User user = **new** User();  user.setName(**"tianshozhi"**);  **int** insertCount = userMapper.insert(user);  **assert** insertCount==1;   Integer id = user.getId();  user = userMapper.selectById(id);  System.***out***.println(user);   user.setName(**"wangxiaoxiao"**);  **int** updateCount = userMapper.updateById(user);  **assert** updateCount==1;   **int** deleteCount = userMapper.deleteById(user.getId());  **assert** deleteCount==1;  } } |

## 5.1 整体设计



设计一个数据源DragonHADataSource，这个数据源下面可以管理多个真实物理数据源。不同的真实物理数据源职责不同，有的负责读(R)，有的负责写(W)，也可以同时负责读写。

总的来说，DragonHADataSource下面管理的物理数据源，大致就是分为2类：

用于读的物理数据源：当有一个读的sql请求时，会从这个集合中选择一个数据源执行这个sql，这是通过一个类ReadDataSourceSelector实现的。

用于写的物理数据源：当有一个写的sql请求时，会从这个集合中选择一个数据源执行这个sql，这是通过一个类WriteDataSourceSelector实现的。

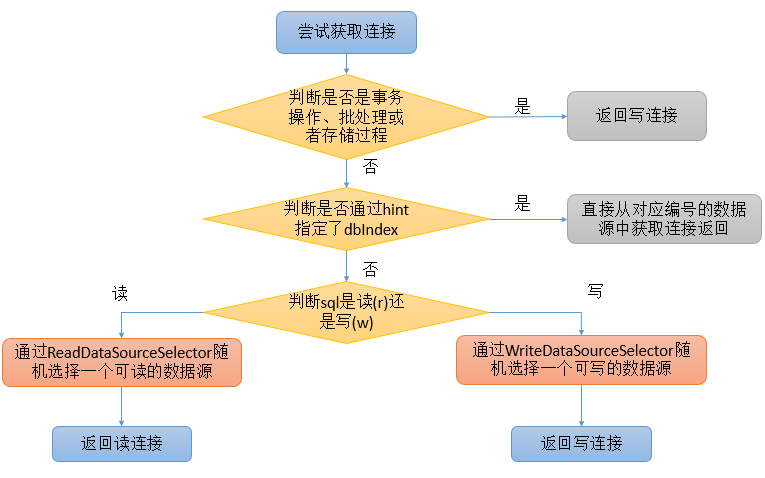
每个物理数据源都支持权重。

一般情况，主库都是配置成可读可写，从库配置可读。

需要注意的是：

为了便于区分，每个物理数据源都需要指定的一个唯一的dbIndex。

## 5.2 ****连接获取过程****



在获取一个连接时，执行的步骤如下所示：

1、首先判断是否是实务操作、批处理或者存储过程，如果是，获取一个写连接，否则，进入第2步

2、判断有没有通过sql或者threadlocal的方式传递了hint，如果传递了hint，则从hint中指定的dbIndex数据源中获取连接，否则进去第三步

3、判断sql是读还是写，如果是写返回写连接，如果是读，返回度连接。这个步骤会对sql是读还是写的结果进行缓存。对于Statement中使用的sql不会缓存，每次都需要判断，对于PrepareStatement中的sql，会进行缓存。

以上描述的只是第一次获取真实连接的过程，由于通常情况下，我们在获取一个连接的时候，可能会执行多个sql，有读有写。这个时候的判断逻辑如下：

1. 如果当前连接是写连接，不管之后的sql是查询、更新、批处理、或者事务，总是返回当前连接
2. 如果当前连接是读连接，之后的sql需要写连接，则切换真实连接
3. 如果sql是读类型，总是返回当前连接
4. 如果传递了hint，会先判断当前连接是否是从对应的DBIndex中获取的连接，如果是，直接返回当前连接，否则切换连接。

## ****5.3 Hint****

支持使用Hint的方式直接指定使用某个数据源。一些场景下，可能需要直接指定使用某个数据源，例如写完数据，立即读的情况。主库的数据同步到从库可能会存在延迟，导致读不到数据。因此需要可以手工影响路由方式的办法，这就是Hint的作用。

### 5.3.1 Sql hint

如

|  |
| --- |
| /\*DRAGON\_HA ( PHYSICAL\_DS\_INDEXES= yourDbIndex [,...] )\*/ SELECT \* FROM user" |

其中/\*...\*/部分就是所谓的Hint，不区分大小写。

红色部分是固定的，绿色部分填写配置的dbIndex，一个sql可以指定多个dbIndex，用逗号”，”进行分割。

如：

|  |
| --- |
| **/\*DRAGON\_HA ( PHYSICAL\_DS\_INDEXES = slave1,slave2)\*/ SELECT \* FROM user** |

### 5.3.2 ThreadLocal Hint

除了通过sql的方式传递hint，还可以通过threadlocal的方式进行传递，传递完成一定要记得清理，dragon不会主动帮用户清理hint。

调用ThreadLocalHintUtil的set方法设置dbIndex，通过get方法获取当前的dbIndex，通过remove方法移除hint。

## ****5.4 事务支持****

一旦开启了事务，事务中所有的sql语句，不论是读还是写，都是在写库(主库)上执行。

## ****5.5 读失败重试****

对于一个sql请求失败了，支持进行自动重试。例如，有2个DruidDataSource用于读，当第一个读取失败的时候，那么会尝试使用第二个进行读。默认最多重试3次(包含第1次)，且只会通过未尝试过的数据源进行重试。

只对读操作进行重试，写操作不会进行重试，如果开启了事务，则也不会进行重试。另外如果通过hint或者threadlocal方式指定了db，则也不会进行重试。

## 5.6 动态移除不可用的数据源

对于一些致命的异常，例如某个数据库server实例宕机，导致某个物理数据源不可用的情况，会将这个物理数据源标记为不可用，之后有sql的时候，不会再转发到这个数据源上。会有一个线程，定时测试被标记为不可用的数据源，一旦测试成功，重新标记为可用。

## 5.7 ExeptionSorter

读失败重试的基础是对SqlException进行排序，如果是读取超时异常，可以考虑重试，但是如果是致命异常，例如某个mysql实例挂了，那么重试也没有意义，不光不应该重试，而且以后的请求都不应该路由到对应的数据源上。这就涉及到对异常进行排序。

### 5.7.1 SqlState

http://blog.csdn.net/hyx1990/article/details/22806441

根据 X/Open 和 SQL Access Group SQL CAE 规范 (1992) 所进行的定义，SQLERROR 返回 SQLSTATE 值。SQLSTATE 值是包含五个字符的字符串 。五个字符包含数值或者大写字母， 代表各种错误或者警告条件的代码。SQLSTATE 有个层次化的模式：头两个字符标识条件的通常表示错误条件的类别， 后三个字符表示在该通用类中的子类。成功的状态是由 00000 标识的。SQLSTATE 代码在大多数地方都是定义在 SQL 标准里的。

表 7. 类代码 08：连接异常

| **SQLSTATE 值** | **含义** |
| --- | --- |
| 08001 | 应用程序请求器不能建立连接。 |
| 08002 | 连接已存在。 |
| 08003 | 连接不存在。 |
| 08004 | 应用程序服务器拒绝建立连接。 |
| 08007 | 事务解析未知。 |
| 08502 | 用 TWOPHASE 的 SYNCPOINT 运行的应用程序进程发出的 CONNECT 语句无效，因为无事务管理器可用。 |
| 08504 | 当处理指定的路径重命名配置文件时遇到错误。 |

### 5.7.2 vendor code

Java中的SqlException除了包含sqlstate，还包含一个vendor code。这是特定数据库供应商定义的错误。针对这类问题也要处理。

# 6 DragonShardingDataSource设计

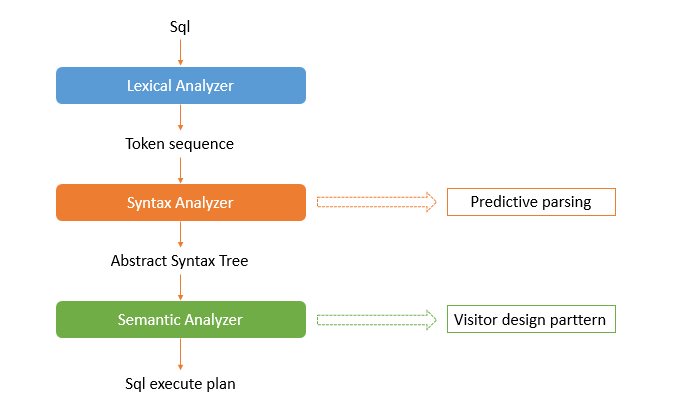
## 6.1 规则引擎

## 6.2 Sql Parser

建议都使用preparestatement,这样会缓存可以解析出来AST。

特别的，使用mybatis的情况下，建议都使用#，这样会使用占位符？。如果使用$，mybatis依然会创建preparestatement，但是不使用？，直接将值填充进sql。这样缓存命中率降低。

### 6.2.1 Sql 解析流程



### 6.2.2 主流的开源sql-parser

目前开源的sql-parser有很多，例如：

**FoundationDB SQL Parser：**

项目地址https://github.com/FoundationDB/sql-parser

**Jsqlparser**

项目地址：https://github.com/JSQLParser/JSqlParser

**Druid SQL Parser**

项目地址：<https://github.com/alibaba/druid/wiki/SQL-Parser>

其中Fdbparser和jsqlparser都是基于javacc实现的。

#### 6.2.2.1 sql解析性能对比

Mycat团队曾经做一个性能测试，对同一个sql语句，使用3种解析器解析出ast语法树，执行10万次、100万次的时间对比。测试结果表明：

1. 10万次：druid比fdbparser快10倍，比JSQLParser快6倍
2. 100万次：druid比fdbparser快15倍，比JSQLParser快近10倍
3. 如果sql越长，其快的倍数会越多，之前在公司有一条长的select语句，druid解析能比fdbparser快40倍。

Mycat团队同时还给出了druid解析更加快的原因。

1. Druid的解析过程使用“统筹方法”，就是几件事情同时一起做。Druid解析包含词法解析和语法解析，这两件事情它只需要从字符的第一个到最后一个遍历一遍，就同时完成了词法解析和语法解析，语法树也已经构造完成。
2. Fdbparser由于依赖于javacc解析，做了很多无用功，它是解析完所有的词法，中间的对象得到后最后再构造出ast语法树对象。

#### 6.2.2.2 数据库方言支持对比

SQL-92、SQL-99等都是标准SQL，mysql/oracle/pg/sqlserver/odps等都是方言，sql-parser需要针对不同的方言进行特别处理。

Druid的sql parser是目前支持各种数据语法最完备的SQL Parser。目前对各种数据库的支持如下：

|  |  |  |
| --- | --- | --- |
| 数据库 | DML | DDL |
| mysql | 完全支持 | 支持大部分 |
| oracle | 大部分 | 支持大部分 |
| postgresql | 完全支持 | 支持大部分 |
| sql server | 支持常用的 | 支持常用的ddl |
| db2 | 支持常用的 | 支持常用的ddl |

druid还缺省支持sql-92标准的语法，所以也部分支持其他数据库的sql语法。

Fdbparser虽然有很多弊端，但是也有它存在的理由，它更偏向于标准sql。

更能引导大家写标准sql，而不是mysql方言，这样应用程序才有可能支持多种数据库，领导说要换数据库时，不至于因为都是非标准sql要大改才能迁移到其他数据库。

结论：Druid支持的方言比fdbparser多。

需要注意的是，并不是偏向于sql标准，就一定好。例如limit是mysql数据库方言，如果不支持这个关键字，估计开发者要抓狂了。

## 6.3 跨库操作与事务

# 7 配置模块

## 7.1 properties配置说明

|  |
| --- |
| *#dragon-sharding配置总共分为三个部分：数据源配置、逻辑表配置、其他配置 #===============================数据源配置开始，配置项key以datasource开头===================================== #数据源的命名格式，会利用此配置项创建一个java.text.MessageFormat对象* **datasource.namePattern**=**dragon\_sharding\_{0,number,#00}** *#数据源名称列表，注意列出的每一个名称都要满足datasource.namePattern规定的格式* **datasource.list**=**dragon\_sharding\_00,dragon\_sharding\_01** *#提示：如果分库过多，那么我们可能要配置之的对应的数据源也越多，可以使用以下datasource.indexRange配置项替代 #以下配置项会将0-2之间的数字，即0,1,2逐一当做参数传递给datasource.namePattern配置项构造的MessageFormat对象的format方法 #datasource.indexRange=[0,2]  #对datasource.list列出的数据源进行配置，以下为这些数据源提供默认的配置* **datasource.datasocueClass**=**com.alibaba.druid.pool.DruidDataSource** *#默认配置key命名方式datasource.default.{property} ，其中{property}表示datasource.datasocueClass定义的物理数据源具有的属性 #不同的物理数据源属性不同，例如：以下属性都是DruidDataSource具有的属性，如果使用其他的数据源，则可能需要使用不同的属性名* **datasource.default.username**=**root datasource.default.password**=**shxx12151022 datasource.default.driverClassName**=**com.mysql.jdbc.Driver datasource.default.initialSize**=**10 datasource.default.maxActive**=**50 datasource.default.maxIdle**=**8 datasource.default.validationQuery**=**select 'x' datasource.default.testOnBorrow**=**false datasource.default.testWhileIdle**=**true** *#针对每个数据源的特殊配置，每个数据源的url应该都不同的，因此需要单独配置，其他部分会继承默认的配置，只要用数据源的名称替换掉default即可* **datasource.dragon\_sharding\_00.url**=**jdbc:mysql://localhost:3306/dragon\_sharding\_00 datasource.dragon\_sharding\_01.url**=**jdbc:mysql://localhost:3306/dragon\_sharding\_01** *#如果想覆盖其他默认配置，例如对于dragon\_sharding\_02，假设我们希望其initialSize=5，可以使用以下配置* **datasource.dragon\_sharding\_02.initialSize**=**5** *#默认库，主要用于处理不要分库，只需要分表的情况 #datasource.defaultDSName=dragon\_sharding\_00 #===============================数据源配置结束================================================================  #==============================逻辑表配置开始,配置项key以logicTable开头======================================= #所有需要拆分的逻辑表* **logicTable.list**=**user,user\_account** *#表与库的映射关系，如果不配置，说明每个分库里面只有一个分表，表名就是逻辑表名，如果配置了* **logicTable.default.namePattern**=**#logicTable#\_{0,number,#0000}** *#提示：这段配置说明在dragon\_sharding\_00和dragon\_sharding\_01两个分库里面都有四个表user\_00，user\_01,user\_account\_00,user\_account\_01 #如果每个表拆分后，在每个库中的映射规则都相同，可以使用以下配置简化 #逻辑表的db默认路由规则，多个规则用分号";"进行分割* **logicTable.default.dbRouteRules**=**${id}.toLong().intdiv(100)%100** *#逻辑表的tb默认路由规则* **logicTable.default.tbRouteRules**=**${id}.toLong()%10000** *#logicTable.default.everydb.mapping=#logicTable#\_[00,01] #如果每个库和表的映射规则不同* **logicTable.default.dragon\_sharding\_00.mapping**=**#logicTable#\_[0000,0001] logicTable.default.dragon\_sharding\_01.mapping**=**#logicTable#\_[0100,0101]** *#如果某个表不使用默认的路由规则，可以使用以下方式覆盖，假设user\_account* **logicTable.user\_account.dbRouteRules**=**${user\_id}.toLong().intdiv(100)%100 logicTable.user\_account.tbRouteRules**=**${user\_id}.toLong()%100** *#==============================逻辑表配置结束=======================================  #==============================其他配置，可以省略======================================= #并发执行线程池配置 #核心线程池数，默认值为datasource.list的个数* **dragon.executor.corePoolSize**=**10** *#默认为datasource.list\*10* **dragon.executor.maxPoolSize**=**30** *#任务队列最大容量，默认为所有分表的个数，当执行sql操作时，如果队列已满，会阻塞任务提交线程，直至超时或者队列不为空* **dragon.executor.workQueueSize**=**10000** *#默认为3秒* **dragon.executor.timeout**=**3000** *#用于上传监控信息，如果没有，可以不配* **dragon.appName**=**draong-sharding** |

# 8 分布式id生成器

# 9 使用案例

见dragon项目子模块:dragon-demo

Spring+mybatis+dragon 整合,参见: com.tianshouzhi.dragon.demo.DragonSpringMybatisTest

jdbc api demo,参见: com.tianshouzhi.dragon.demo.DragonAPITest