Dragon项目设计文档

# 1主流的数据库中间件设计方式

主要分为3种Database Proxy、ORM框架集成、实现JDBC规范

## 1.1 Database Proxy

目前主流的以Database Proxy方式设计的中间件有：Mysql-proxy、Mycat、Atlas。

Database Proxy的方式中，应用不直接连接数据库，而是连接proxy，由proxy来完成sql的解析、路由、合并查询结果、返回结果给客户端。

优点：

对应用来说是完全透明的，配置也完全和配置单个数据源一样。

缺点：

1、资源竞争：如果多个应用需要进行分库分表，访问同一个proxy，那么必然对proxy的资源产生竞争，最恶劣情况，如果由于某个应用由于请求非常频繁，导致proxy宕机，那么，其他的应用也受到影响，不能继续访问这个proxy。

2、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。

## 1.2 ORM框架集成

目前以ORM框架集成形式提供的主流的数据库中间件有：hibernate shards、ibatis-sharding。

优点：应用直连数据库，省去了中间的代理，应用之间相互无影响

缺点：必须要使用特定的ORM框架

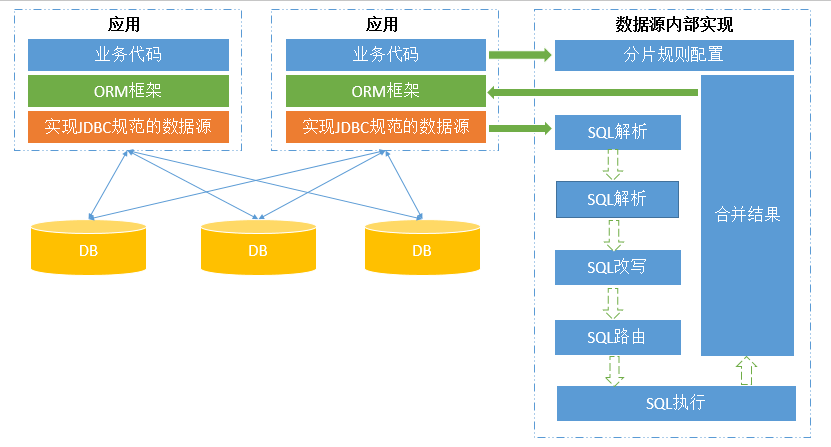
## 1.3 实现JDBC规范的数据源

目前实现jdbc规范形式的数据库中间件主要有：

淘宝的TDDL

大众点评的Zebra

当当网的sharding-jdbc



## 1.4 dragon与tddl、zebra、sharding-jdbc功能对比

架构设计对比

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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 |  |

INSERT操作：

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dragon | Tddl | Zebra |
| 单表单条记录插入 | 支持 | 支持 | 支持 |
| 单表批量插入 | 支持 |  | 有限支持,插入的记录必须都在一个表中,不能路由到多个表 |

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：

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 |  |  |

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。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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：

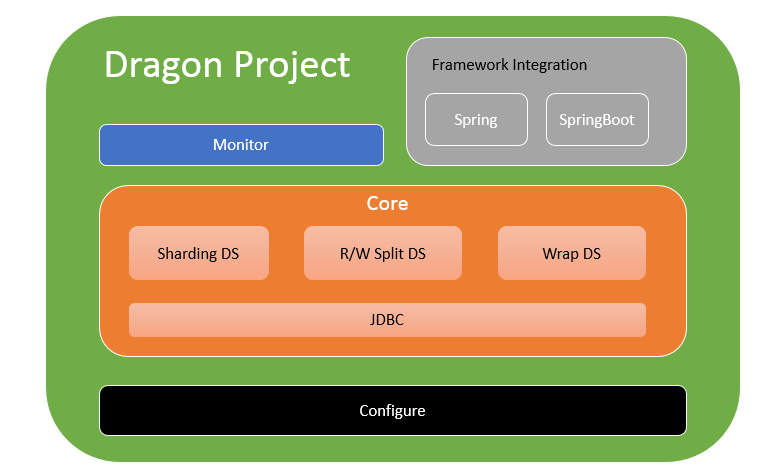
|  |  |  |
| --- | --- | --- |
| Dragon | Tddl | Zebra |
| 支持单库内的事务 | 支持单库内的事务 |  |

# 3、Drangon项目设计

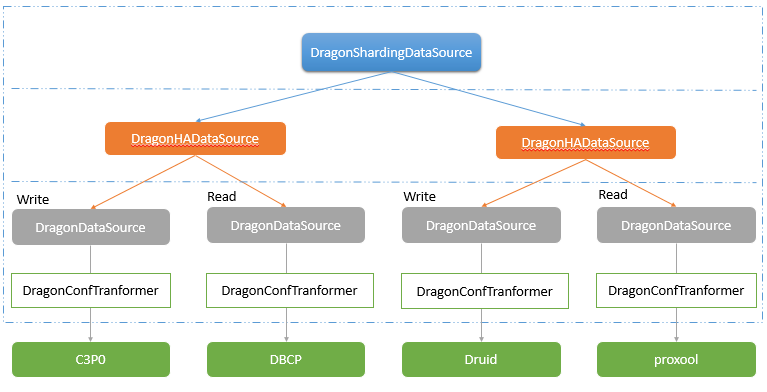
DragonDataSource是一个支持读写分离、分库分表的数据源，实现了JDBC规范。

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：

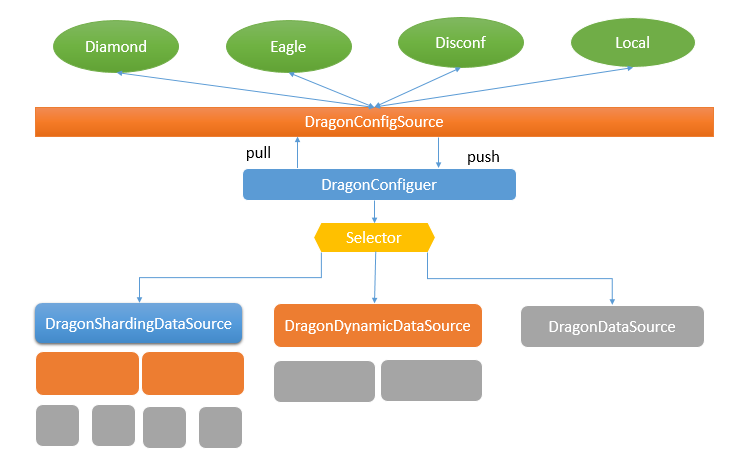
**DragonShardingDataSource**是用于支持分库分表的数据源

**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：

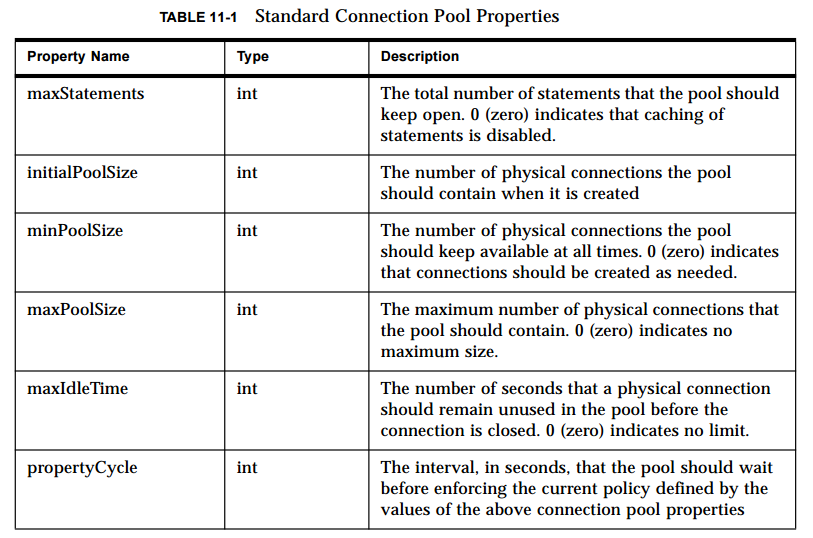
* pull方式指的是主动去外部数据源拉取配置，在Dragon启动的时候，需要主动pull。
* push方式指的是外部数据源主动向Dragon推送配置，例如想再运行时修改配置。

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、

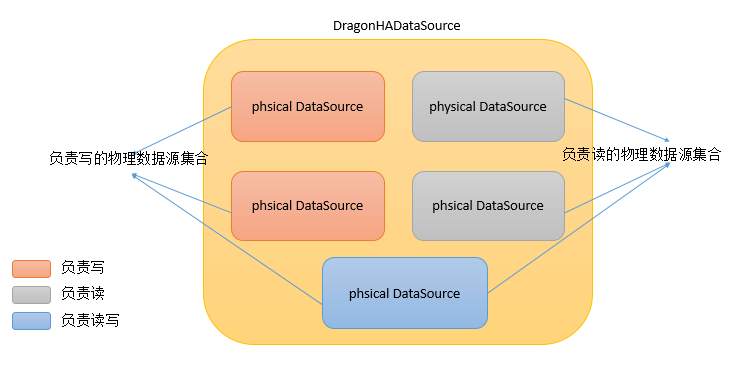
username、

password、

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

# 5 DragonHADataSource设计

## 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。

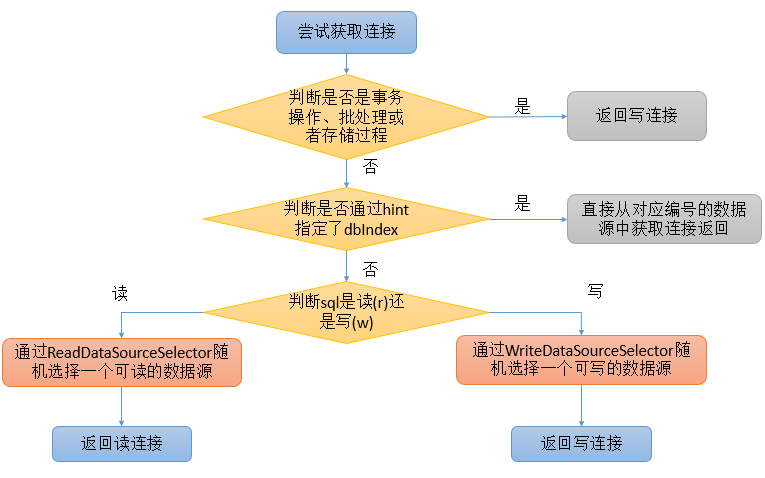
每个物理数据源都支持权重。

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

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

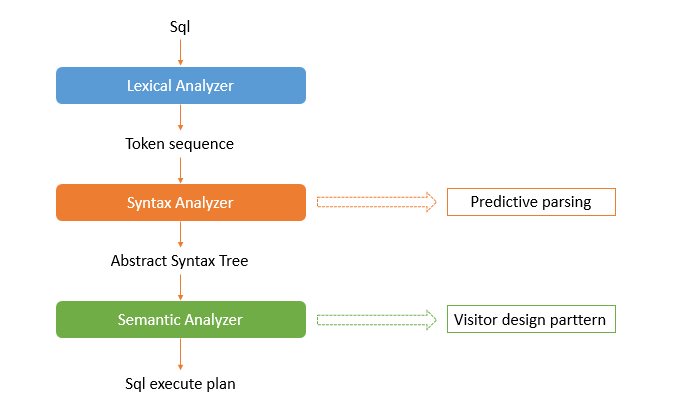
Java中的SqlException除了包含sqlstate，还包含一个vendor code。这是特定数据库供应商定义的错误。针对这类问题也要处理。

# 6 DragonShardingDataSource设计

## 6.1 规则引擎

## 6.2 Sql Parser

### 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