如何实现分库分表+动态数据源+读写分离

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：分库分表 动态数据源 读写分离

题目描述

分布式数据层中间件详解：如何实现分库分表+动态数据源+读写分离

1. 面试题分析

1. 根据题目要求我们可以知道：

    1、分布式数据层中间件：

2、分布式数据层中间件架构设计

3、分布式数据层中间件：具体实现

2. 容易被忽略的坑

- 分析片面
- 没有深入

01分布式数据层中间件：

1.简介：

分布式数据访问层中间件，旨在为供一个通用数据访问层服务，支持MySQL动态数据源、读写分离、分布式唯一主键生成器、分库分表、动态化配置等功能，并且支持从客户端角度对数据源的各方面（比如连接池、SQL等）进行监控，后续考虑支持NoSQL、Cache等多种数据源。

2.功能特性

- 动态数据源
- 读写分离
- 分布式唯一主键生成器
- 分库分表
- 连接池及SQL监控
- 动态化配置

02常见的数据层中间件：

1.TDDL

淘宝根据自己的业务特点开发了TDDL框架，主要解决了分库分表对应用的透明化以及异构数据库之间的数据复制，它是一个基于集中式配置的JDBC datasource实现。

特点

实现动态数据源、读写分离、分库分表。

缺点

分库分表功能还未开源，当前公布文档较少，并且需要依赖diamond（淘宝内部使用的一个管理持久配置的系统）

2.Atlas

Qihoo 360开发维护的一个基于MySQL协议的数据中间层项目。它实现了MySQL的客户端与服务端协议，作为服务端与应用程序通信，同时作为客户端与MySQL通信

特点：

实现读写分离、单库分表

缺点：

不支持分库分表

3.MTDDL（Meituan Distributed Data Layer）

美团点评分布式数据访问层中间件

特点

实现动态数据源、读写分离、分库分表，与tddl类似。

下面我以MTDDL为例，也可以参考淘宝tddl，完整详解分布式数据层中间件的架构设计。

03分布式数据层中间件架构设计

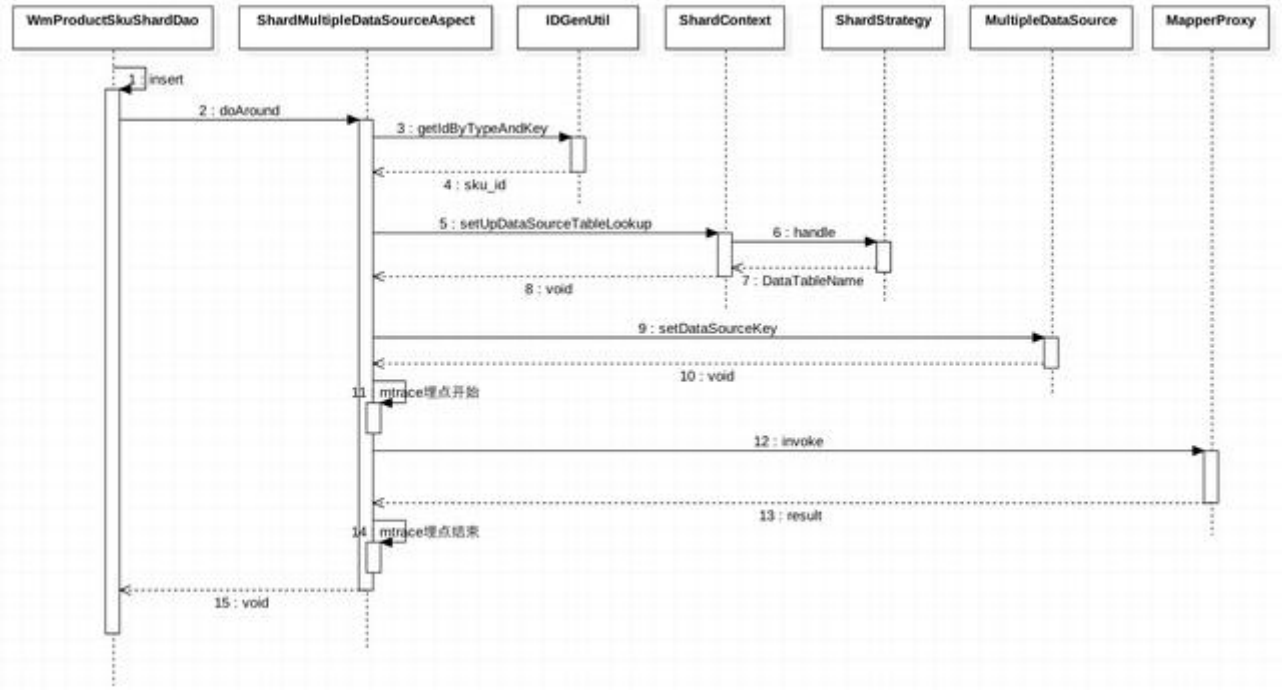
下图是一次完整的DAO层insert方法调用时序图，简单阐述了MTDDL的整个逻辑架构。

其中包含了：

1.分布式唯一主键的获取

2.动态数据源的路由

3.以及SQL埋点监控等过程：



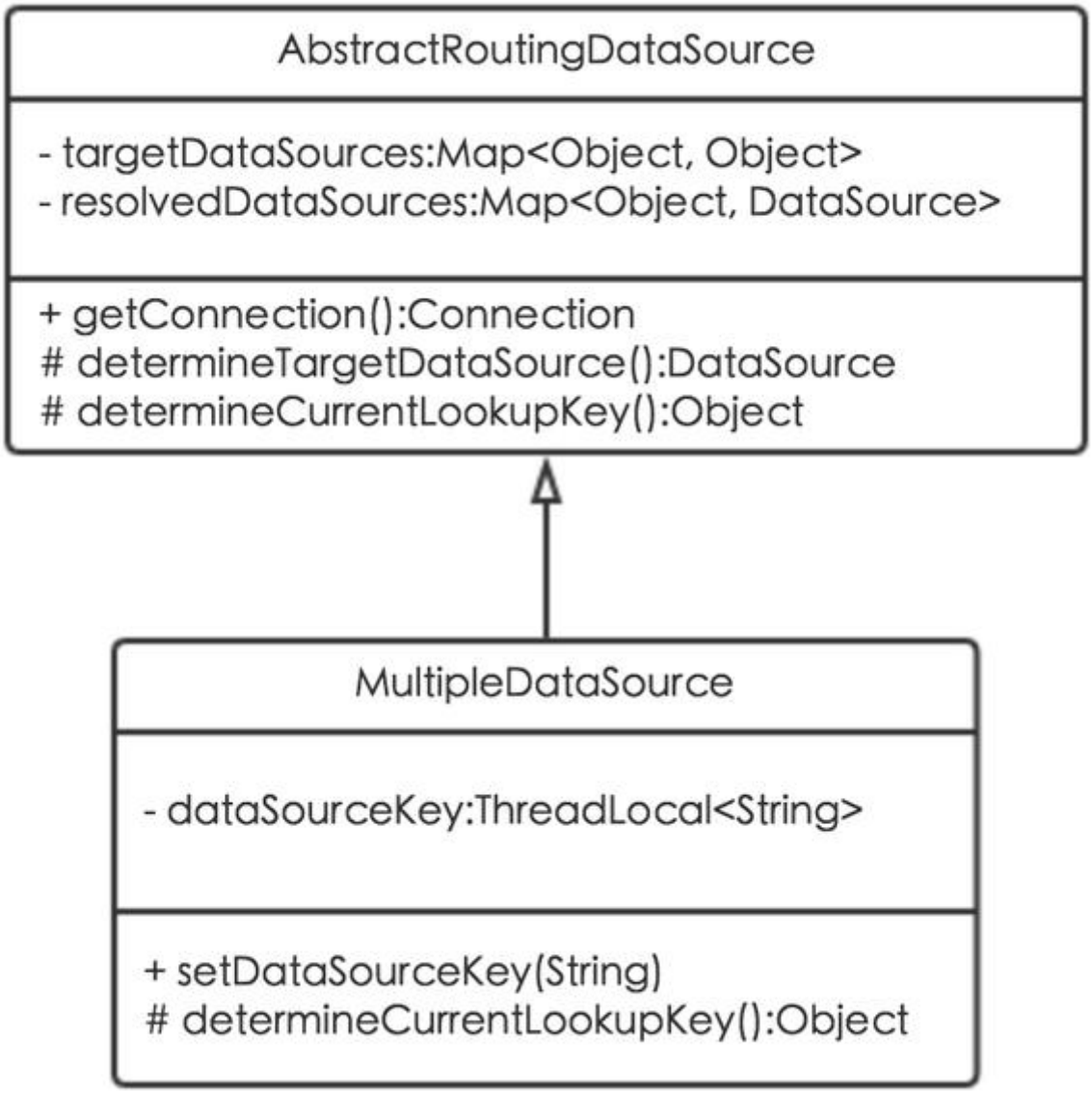
04分布式数据层中间件：具体实现

1.动态数据源及读写分离

在Spring JDBC AbstractRoutingDataSource的基础上扩展出MultipleDataSource动态数据源类，通过动态数据源注解及AOP实现。

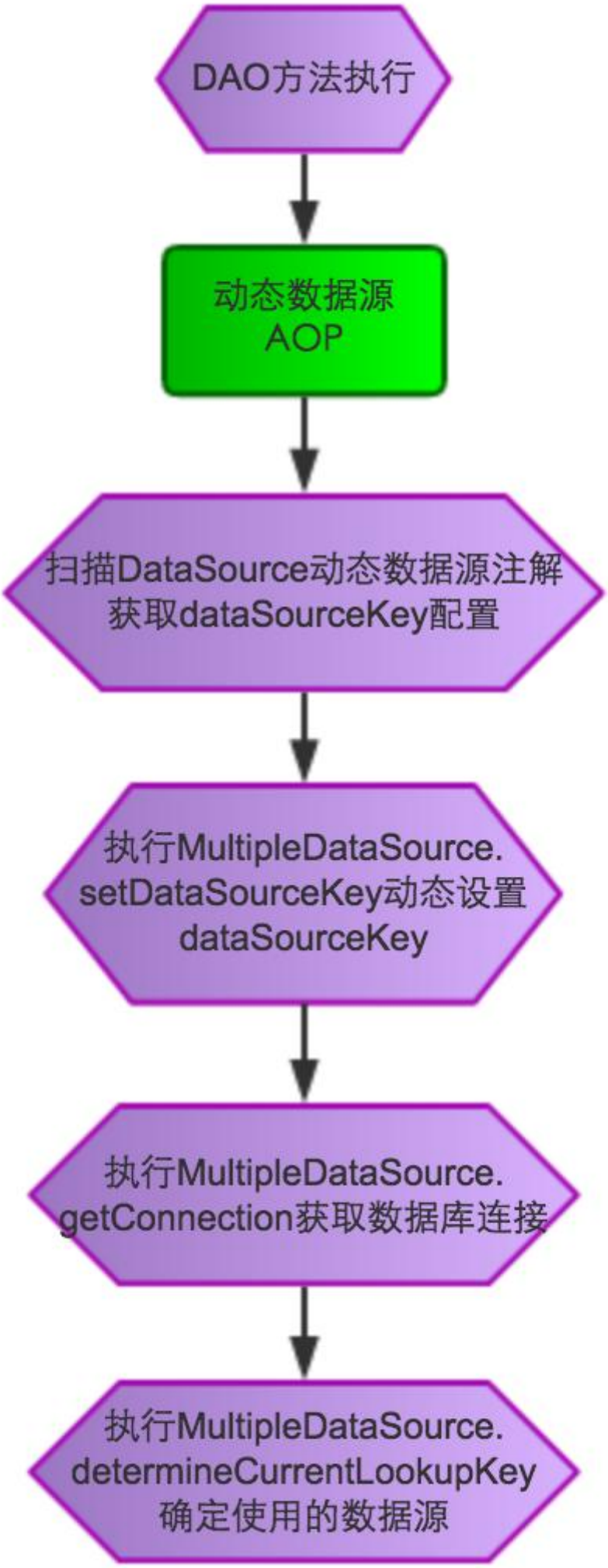
2.动态数据源

MultipleDataSource动态数据源类，继承于Spring JDBC AbstractRoutingDataSource抽象类，实现了determineCurrentLookupKey方法，通过setDataSourceKey方法来动态调整dataSourceKey，进而达到动态调整数据源的功能。其类图如下：



3.动态数据源AOP

ShardMultipleDataSourceAspect动态数据源切面类，针对DAO方法进行功能增强，通过扫描DataSource动态数据源注解来获取相应的dataSourceKey，从而指定具体的数据源。具体流程图如下：


DAO方法执行
动态数据源
AOP
扫描DataSource动态数据源注解
获取dataSourceKey配置
执行MultipleDataSource.
setDataSourceKey动态设置
dataSourceKey
执行MultipleDataSource.
getConnection获取数据库连接
执行MultipleDataSource.
determineCurrentLookupKey
确定使用的数据源

4.配置和使用方式举例

```
/**
 * 参考配置
 */
<bean id="multipleDataSource"
class="com.sankuai.meituan.waimai.datasource
.multi.MultipleDataSource">
 /** 数据源配置 */
 <property name="targetDataSources">
 <map key-type="java.lang.String">
 /** 写数据源 */
 <entry key="dbProductWrite" value-
ref="dbProductWrite"/>
 /** 读数据源 */
 <entry key="dbProductRead" value-
ref="dbProductRead"/>
 </map>
 </property>
</bean>
/**
 * DAO使用动态数据源注解
 */
public interface WmProductSkuDao {
 /** 增删改走写数据源 */
 @DataSource("dbProductWrite")
```

```
    public void insert(WmProductSku sku);
    /** 查询走读数据源 */
    @DataSource("dbProductRead")
    public void getById(long sku_id);
}
```

5.分布式唯一主键生成器

众所周知，分库分表首先要解决的就是分布式唯一主键的问题，业界也有很多相关方案：

序号实现方案优点缺点UUID本地生成，不需要RPC，低延时；

扩展性好，基本没有性能上限无法保证趋势递增；

UUID过长128位，不易存储，往往用字符串表示2Snowflake或MongoDB ObjectId分布式生成，无单点；

趋势递增，生成效率快没有全局时钟的情况下，只能保证趋势递增；

当通过NTP进行时钟同步时可能会出现重复ID；

数据间隙较大3proxy服务+数据库分段获取ID分布式生成，段用完后需要去DB获取，同server有序可能产生数据空洞，即有些ID没有分配就被跳过了，主要原因是在服务重启的时候发生；

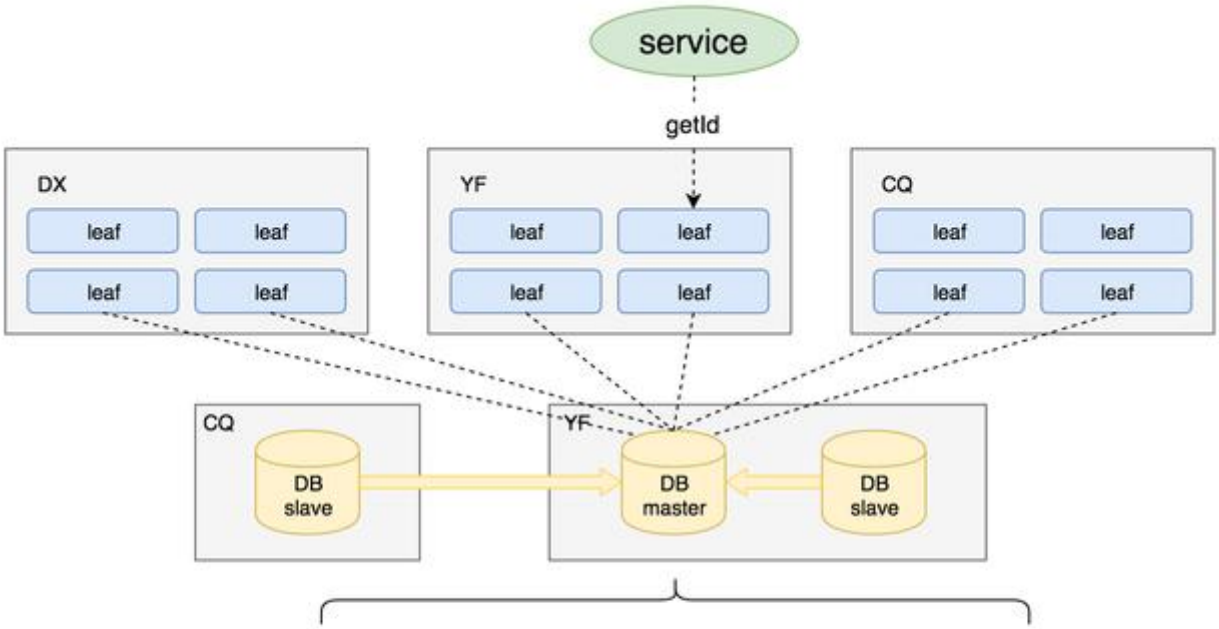
无法保证有序，需要未来解决，可能会通过其他接口方案实现

综上，方案3的缺点可以通过一些手段避免，但其他方案的缺点不好处理，所以选择第3种方案：分布式ID生成系统Leaf。

6.分布式ID生成系统Leaf

分布式ID生成系统Leaf，其实是一种基于DB的Ticket服务，通过一张通用的Ticket表来实现分布式ID的持久化，执行update更新语句来获取一批Ticket，这些获取到的Ticket会在内存中进行分配，分配完之后再从DB获取下一批Ticket。

整体架构图如下：



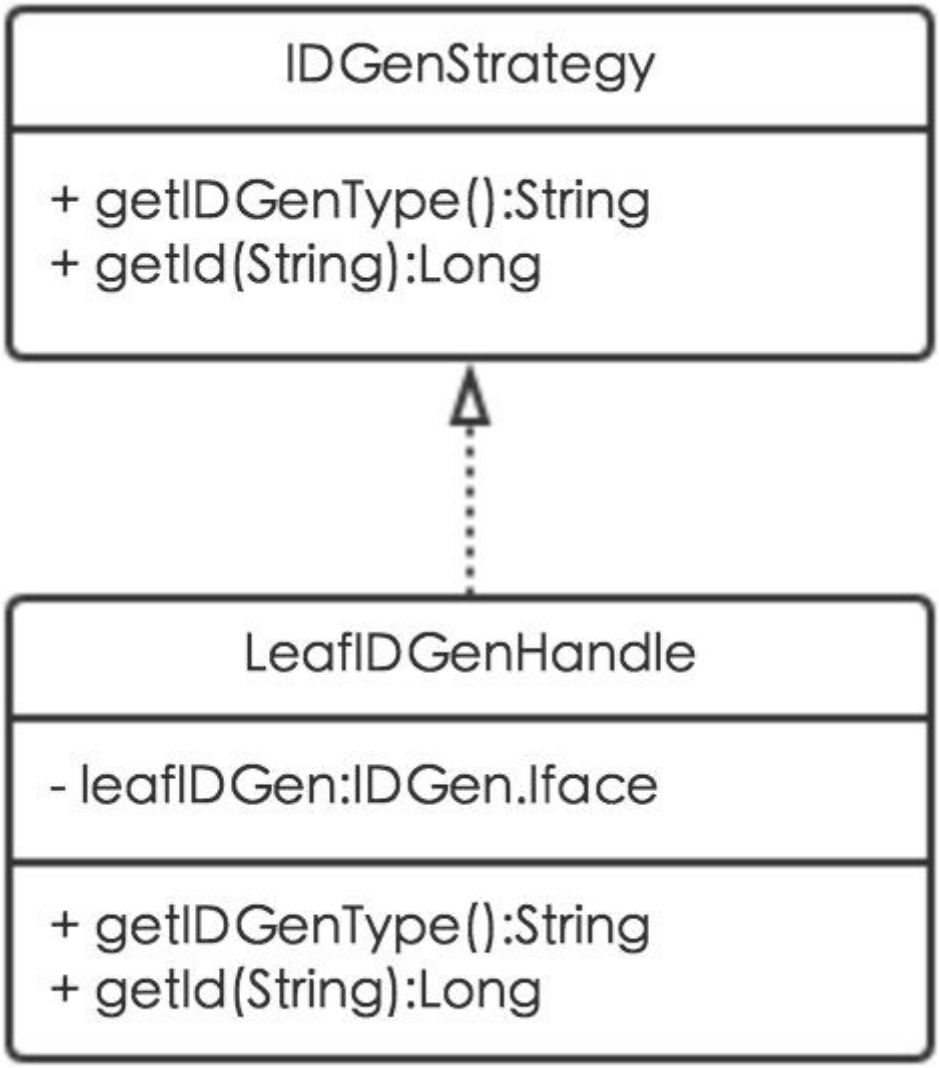
biz_tag	max_id	step	desc	update_time
pay_order	12384124	10000		
waimai_order	892345	1000		
banma_order	20000	9000		
test	50000000	2000		

每个业务tag对应一条DB记录，DB MaxID字段记录当前该Tag已分配出去的最大ID值。

IDGenerator服务启动之初向DB申请一个号段，传入号段长度如 genStep = 10000，DB事务置 MaxID = MaxID + genStep，DB设置成功代表号段分配成功。每次IDGenerator号段分配都通过原子加的方式，待分配完毕后重新申请新号段。

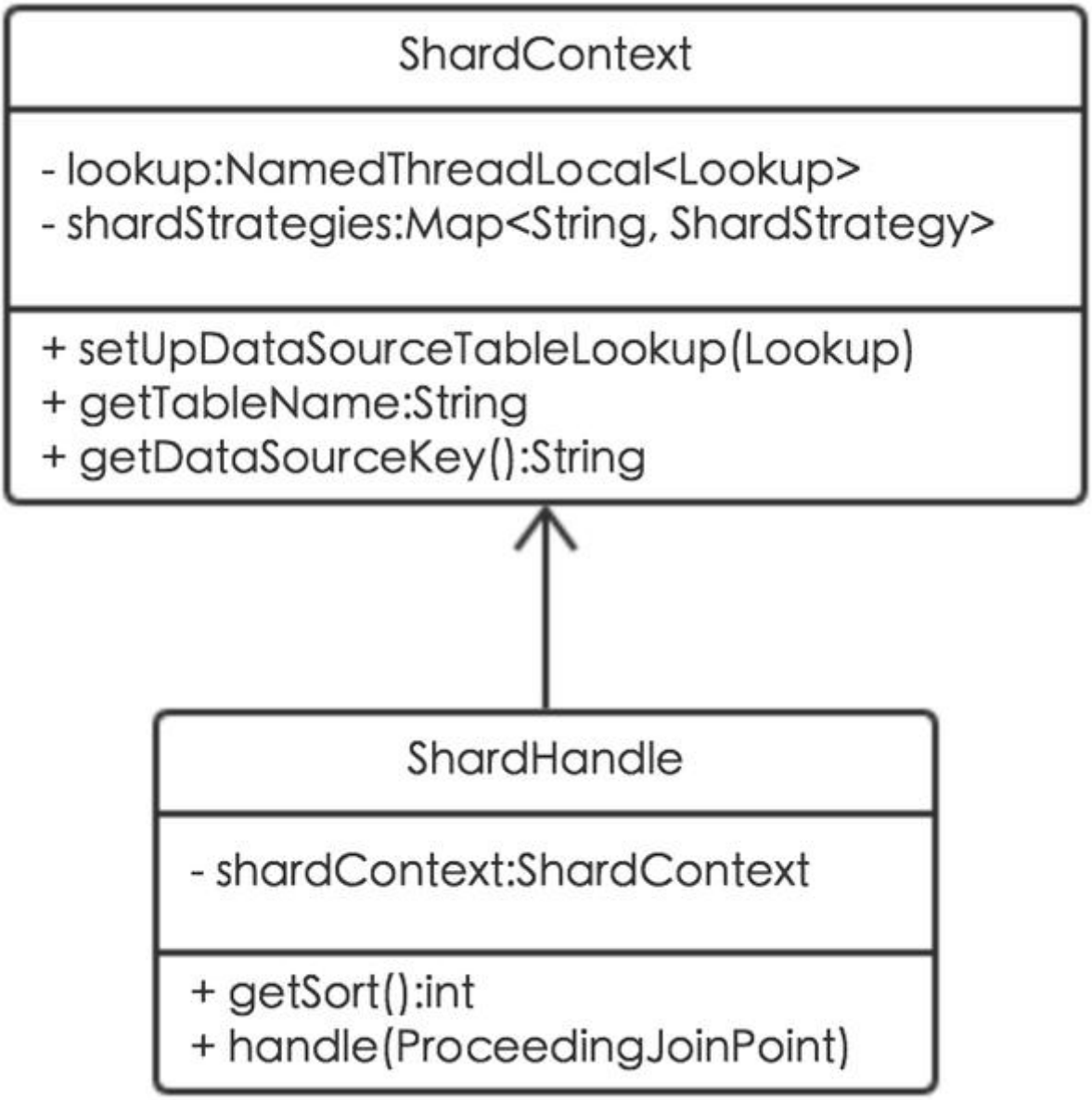
7.唯一主键生成算法扩展

MTDDL不仅集成了Leaf算法，还支持唯一主键算法的扩展，通过新增唯一主键生成策略类实现IDGenStrategy接口即可。IDGenStrategy接口包含两个方法：getIDGenType用来指定唯一主键生成策略，getId用来实现具体的唯一主键生成算法。其类图如下：

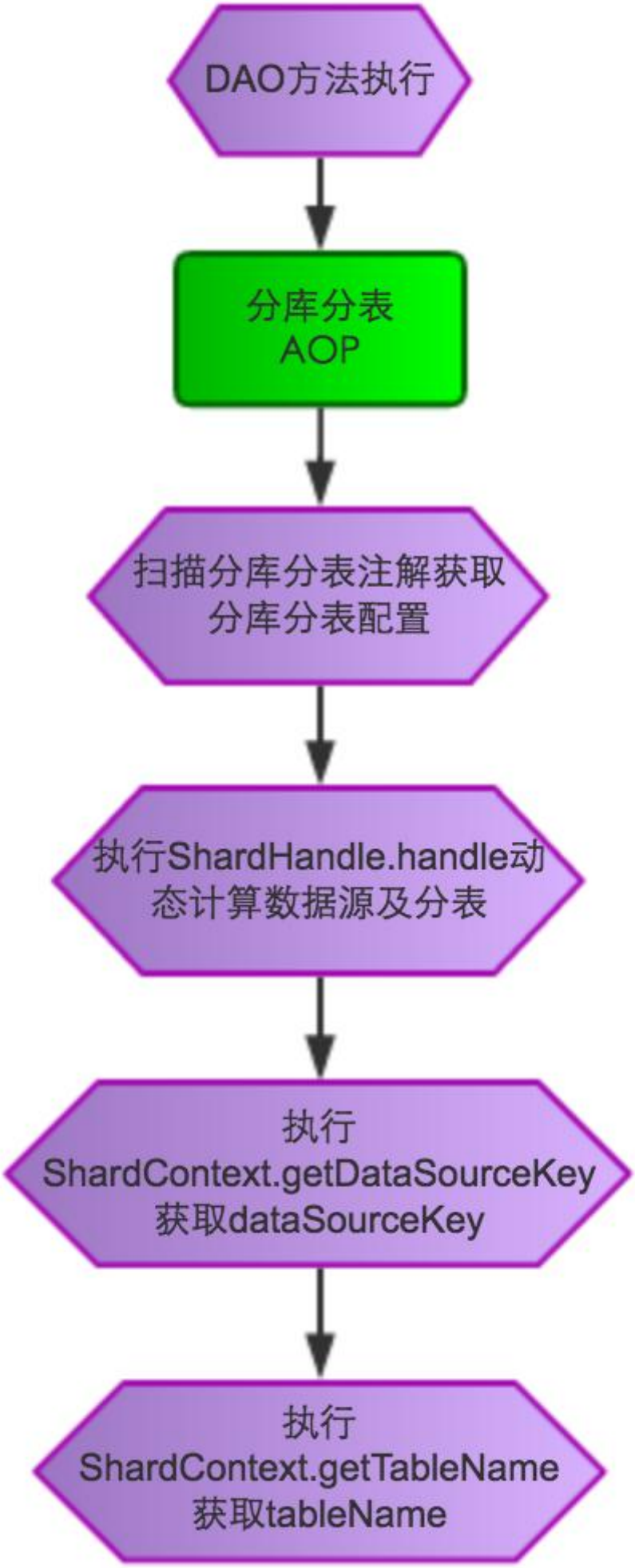


8.分库分表

在动态数据源AOP的基础上扩展出分库分表AOP，通过分库分表ShardHandle类实现分库分表数据源路由及分表计算。ShardHandle关联了分库分表上下文ShardContext类，而ShardContext封装了所有的分库分表算法。其类图如下：


ShardContext
- lookup:NamedThreadLocal<Lookup>
- shardStrategies:Map<String, ShardStrategy>
+ setUpDataSourceTableLookup(Lookup)
+ getTableName:String
+ getDataSourceKey():String
ShardHandle
- shardContext:ShardContext
+ getSort():int
+ handle(ProceedingJoinPoint)

分库分表流程图如下：


DAO方法执行
分库分表
AOP
扫描分库分表注解获取
分库分表配置
执行ShardHandle.handle动
态计算数据源及分表
执行
ShardContext.getDataSourceKey
获取dataSourceKey
执行
ShardContext.getTableName
获取tableName

9.分库分表取模算法

分库分表目前默认使用的是取模算法，分表算法为(#shard_key % (group_shard_num * table_shard_num))，分库算法为 (#shard_key % (group_shard_num * table_shard_num)) / table_shard_num，其中group_shard_num为分库个数，table_shard_num为每个库的分表个数。

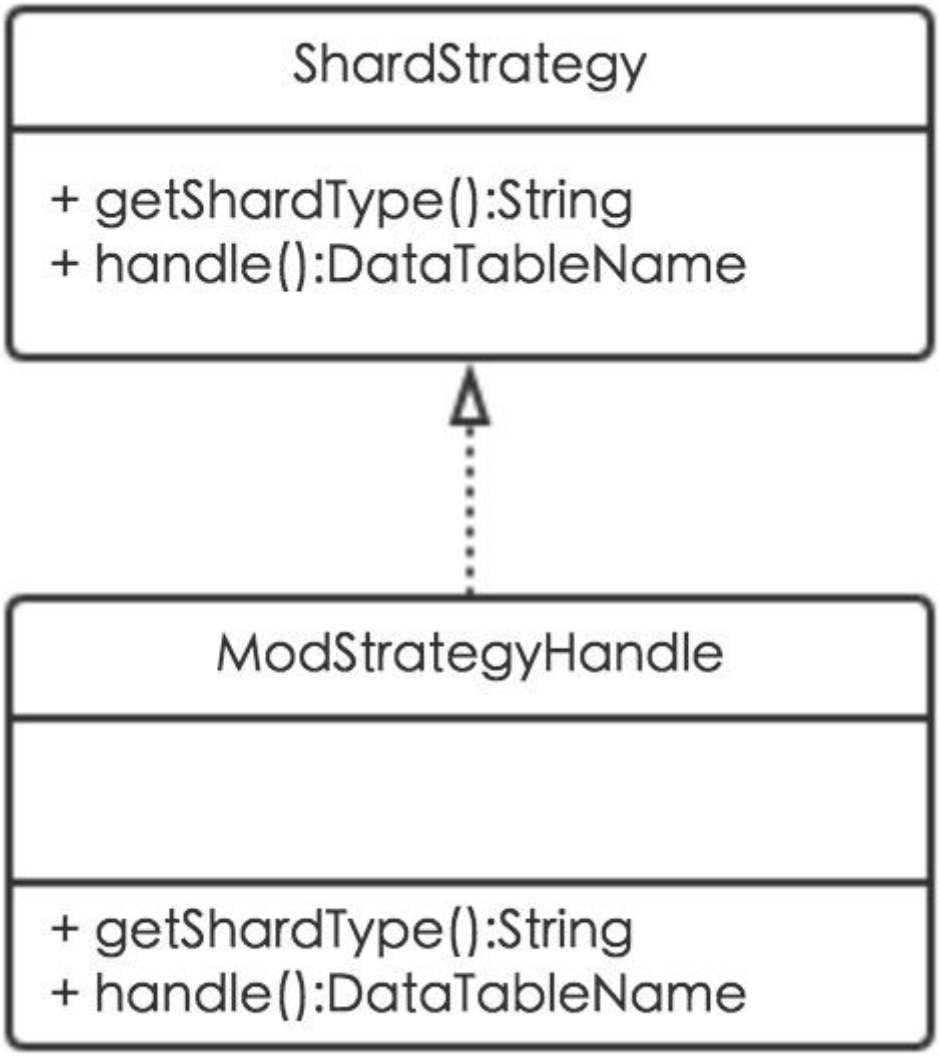
例如把一张大表分成100张小表然后散到2个库，则0-49落在第一个库、50-99落在第二个库。核心实现如下：

```
public class ModStrategyHandle implements
ShardStrategy {
 @Override
 public String getShardType() {
 return "mod";
 }
 @Override
 public DataTableName handle(String
tableName, String dataSourceKey, int
tableShardNum,
 int dbShardNum, Object shardValue) {
 /** 计算散到表的值 */
 long shard_value =
Long.valueOf(shardValue.toString());
```

```
 long tablePosition = shard_value %
tableShardNum;
 long dbPosition = tablePosition /
(tableShardNum / dbShardNum);
 String finalTableName = new
StringBuilder().append(tableName).append("_"
).append(tablePosition).toString();
 String finalDataSourceKey = new
StringBuilder().append(dataSourceKey).append
(dbPosition).toString();
 return new DataTableName(finalTableName,
finalDataSourceKey);
 }
}
```

10.分库分表算法扩展

MTDDL不仅支持分库分表取模算法，还支持分库分表算法的扩展，通过新增分库分表策略类实现ShardStrategy接口即可。ShardStrategy接口包含两个方法：getShardType用来指定分库分表策略，handle用来实现具体的数据源及分表计算逻辑。其类图如下：

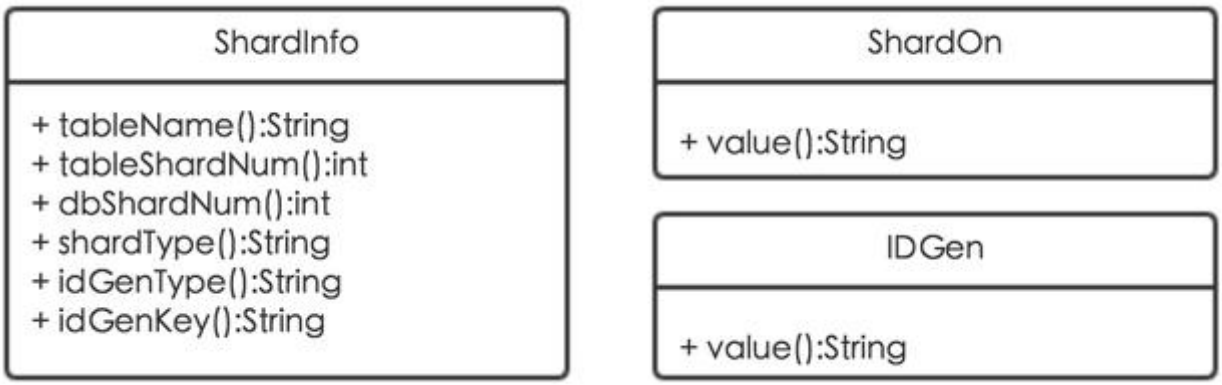


11.全注解方式接入

为了尽可能地方便业务方接入，MTDDL采用全注解方式使用分库分表功能，通过ShardInfo、ShardOn、IDGen三个注解实现。

ShardInfo注解用来指定具体的分库分表配置：包括分表名前缀tableName、分表数量tableShardNum、分库数量dbShardNum、分库分表策略shardType、唯一键生成策略idGenType、唯一键业务方标识idGenKey；ShardOn注解用来

指定分库分表字段；IDGen注解用来指定唯一键字段。具体类图如下：



12.配置和使用方式举例

```
// 动态数据源
@DataSource("dbProductSku")
// tableName：分表名前缀，tableShardNum：分表数量，dbShardNum：分库数量，shardType：分库分表策略，idGenType：唯一键生成策略，idGenKey：唯一键业务方标识
@ShardInfo(tableName="wm_food",
tableShardNum=100, dbShardNum=1,
shardType="mod", idGenType=IDGenType.LEAF,
idGenKey=LeafKey.SKU)
@Component
public interface WmProductSkuShardDao {
 // @ShardOn("wm_poi_id") 将该注解修饰的对象的wm_poi_id字段作为shardValue
 // @IDGen("id") 指定要设置唯一键的字段
 public void insert(@ShardOn("wm_poi_id")
@IDGen("id") WmProductSku sku);
 // @ShardOn 将该注解修饰的参数作为shardValue
 public List<WmProductSku>
getSkusByWmPoiId(@ShardOn long wm_poi_id);
}
```

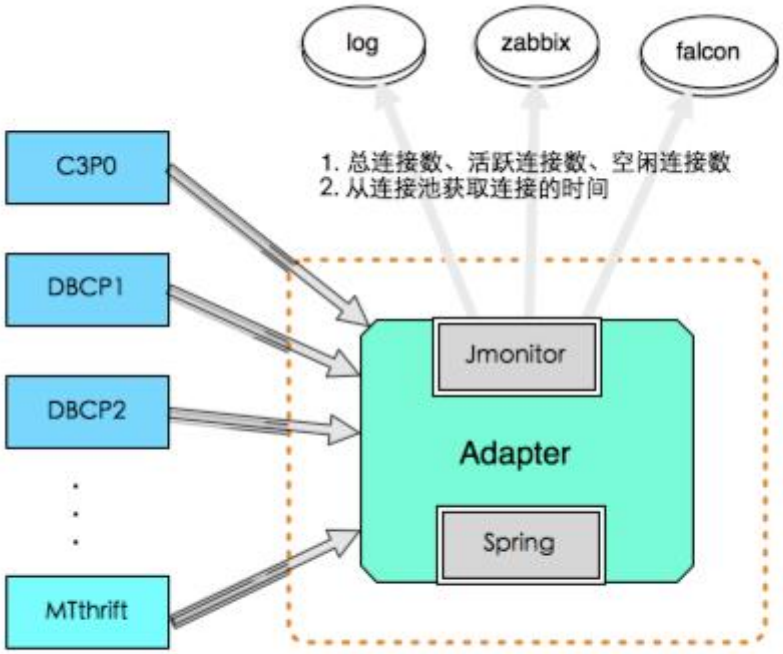
05连接池及SQL监控

DB连接池使用不合理容易引发很多问题，如连接池最大连接数设置过小导致线程获取不到连接、获取连接等待时间设置过大导致很多线程挂起、空闲连接回收器运行周期过长导致空闲连接回收不及时等等，如果缺乏有效准确的监控，会造成无法快速定位问题以及追溯历史。

连接池监控

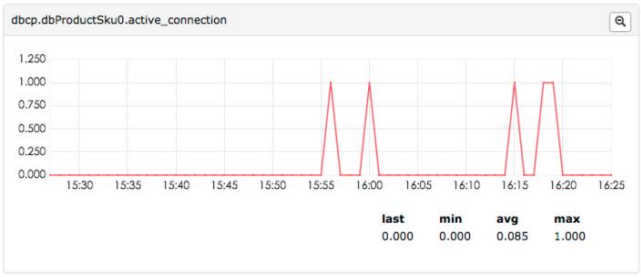
实现方案

结合Spring完美适配c3p0、dbcp1、dbcp2、mtthrift等多种方案，自动发现新加入到Spring容器中的数据源进行监控，通过美团点评统一监控组件JMonitor上报监控数据。整体架构图如下：



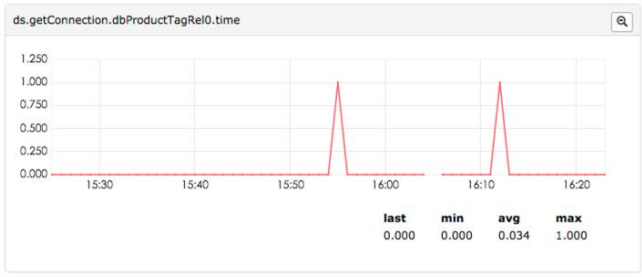
连接数量监控

监控连接池active、idle、total连接数量，Counter格式：（连接池类型.数据源.active/idle/total_connection），效果图如下：



获取连接时间监控

监控获取空闲连接时间，Counter格式：（ds.getConnection.数据源.time），效果图如下：



2. 扩展内容

- 分布式数据库数据一致性的原理、与技术实现方案
- 分布式事务的解决方案，以及原理、总结
- 分布式锁的由来、及Redis分布式锁的实现详解