



21天打卡第七天——第十节，分库分表，路由组件的实现

`DBRouterBase` 这个类好像没有被使用。

宏观认识分库分表

为什么需要分库分表？

因为业务体量随着用户的增长、用户的使用年限逐渐变大，数据增长很快，导致一个数据库中可能承担了多个业务群产生的大量的业务，比如说上万个考勤组的数据随着时间的不断增长导致了好几亿的员工班次历史数据，上万个排班表对应产生的员工班次表可能单表几个月就达到了千万的数据量，半年内就将近到达五六亿的数据量，我们需要把用户产生的数据拆分到不同的库表中，减轻数据库的读写压力，所以需要分库分表，这里又分为垂直拆分和水平拆分两种形式。

垂直拆分和水平拆分

垂直拆分：指的是把业务按照表进行分类，分布到不同的数据库上，这样就把数据压力分担到不同的库上面，最终一个数据库由很多表的构成，每一个表对应不同的业务，换句话说就是专库专用。举个例子，排班引擎业务和考勤组相关的业务原先都在班次库里，此时我需要进行垂直拆分，我把排班引擎的一类表放到排班库中，考勤组相关的一类表放到考勤库中，专库专用。

水平拆分：指的是垂直拆分之后还是遇到了单机瓶颈，单表数据达到七八亿，可以使用水平拆分，相对于垂直拆分的区别是：垂直拆分是把不同的表拆到不同的数据库中，水平拆分是把同一个表拆到不同的数据库中。实际业务中运用的也是水平拆分的场景比较多见。

而拆分的具体实现是由**数据库路由**来实现的，那么又有了接下来的这个问题。

我要完成分库分表具体要怎么做呢？

小傅哥他说实现一个数据库路由的组件给具体要使用的业务服务引入使用就好了，然后他做了这件事，建了一个db-router-spring-boot-starter路由组件。

▼ 什么是Spring Boot Starter呢？

它首先是一个包，一个集合，它把需要用的其他功能组件囊括进来，放到自己的 pom 文件中。

然后它是一个连接，把它引入的组件和我们的项目做一个连接，并且在中间帮我们省去复杂的配置，力图做到使用最简单。

实现一个 starter 有四个要素：

1. starter 命名；
2. 自动配置类，用来初始化相关的 bean；
3. 指明自动配置类的配置文件 spring.factories；
4. 自定义属性实体类，声明 starter 的应用配置属性；

项目资源文件夹里面有个spring.factories文件,它是个啥

实现数据库路由组件

实现水平拆分的数据库路由需要那些知识点呢？

小傅哥告诉我有以下四点：

- AOP 切面拦截的使用，需要给使用数据库路由的方法做上标记，便于处理分库分表逻辑。
- 数据源的切换操作，既然有分库那么就会涉及在多个数据源间进行链接切换，以便把数据分配给不同的数据库。
- 数据库表寻址操作，一条数据分配到哪个数据库，哪张表，都需要进行索引计算。在方法调用的过程中最终通过 ThreadLocal 记录。
- 为了能让数据均匀的分配到不同的库表中去，还需要考虑如何进行数据散列的操作，不能分库分表后，让数据都集中在某个库的某个表，这样就失去了分库分表的意义。

需要用到的技术：[AOP](#)、[数据源切换](#)、[散列算法](#)、[哈希寻址](#)、[ThreadLocal](#)以及[SpringBoot的Starter开发方式](#)等技术。

懵懵懂懂的我还是需要实打实的好好的斟酌每一个细节。

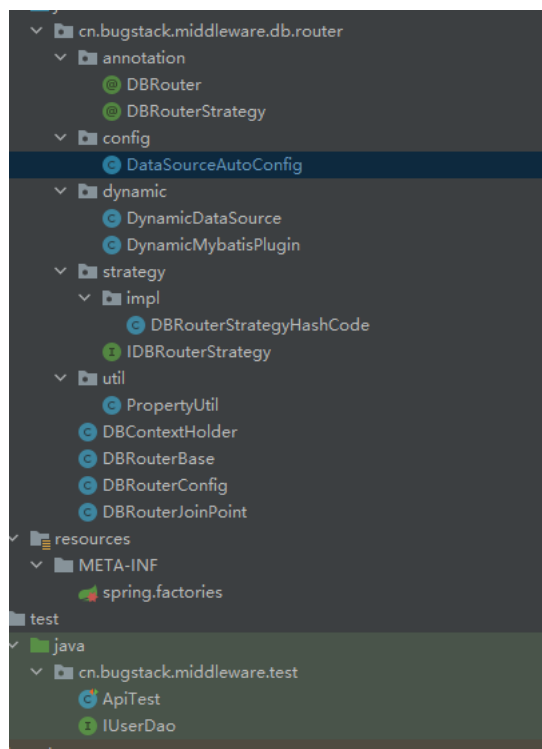
分拆一下路由服务



具体实现逻辑

听听小傅哥是怎么解释的

先看看小傅哥说的数据库路由组件的项目结构：



再了解一些小傅哥用到的Spring的注解：

@Configuration：从Spring3.0，@Configuration用于定义配置类，可替换xml配置文件，被注解的类内部包含有一个或多个被@Bean注解的方法，这些方法将会被AnnotationConfigApplicationContext或AnnotationConfigWebApplicationContext类进行扫描，并用于构建bean定义，初始化Spring容器。

@ConditionalOnMissingBean：它是修饰bean的一个注解，主要实现的是，当你的bean被注册之后，如果而注册相同类型的bean，就不会成功，它会保证你的bean只有一个，即你的实例只有一个，当你注册多个相同的bean时，会出现异常，以此来告诉开发人员。

@Bean是一个方法级别上的注解，主要用在@Configuration注解的类里，也可以用在@Component注解的类里。添加的bean的id为方法名

再回顾以下Spring的事务传播机制：

PROPAGATION_REQUIRED -- 支持当前事务，如果当前没有事务，就新建一个事务。默认的级别就是这个。

PROPAGATION_SUPPORTS -- 支持当前事务，如果当前没有事务，就以非事务方式执行。

PROPAGATION_MANDATORY -- 支持当前事务，如果当前没有事务，就抛出异常。

PROPAGATION_REQUIRES_NEW -- 新建事务，如果当前存在事务，把当前事务挂起。

PROPAGATION_NOT_SUPPORTED -- 以非事务方式执行操作，如果当前存在事务，就把当前事务挂

起。

PROPAGATION_NEVER -- 以非事务方式执行，如果当前存在事务，则抛出异常。

PROPAGATION_NESTED -- 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION_REQUIRED类似的操作。

至于到底是七种还是六种，我就不深究了，大致看了一下什么EJB事务控制（CMT容器管理事务和BMTBean管理事务两种方式）和Java中的事务JDBC事务和JTA事务，有点头疼。

第一步定义路由注解

- 首先我们需要自定义一个注解，用于放置在需要被数据库路由的方法上。
- 它的使用方式是通过方法配置注解，就可以被我们指定的 AOP 切面进行拦截，拦截后进行相应的数据库路由计算和判断，并切换到相应的操作数据源上

这个好理解，自定义注解嘛。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface DBRouter {

    /** 分库分表字段 */
    String key() default "";


}
```

在对应DAO接口的插入数据的方法上加上这个注解就可以实现对应的功能。

第二步解析路由配置

- 数据源配置，在分库分表下的数据源使用中，都需要支持多数据源的信息配置，这样才能满足不同需求的扩展。
- 对于这种自定义较大的信息配置，就需要使用到 `org.springframework.context.EnvironmentAware` 接口，来获取配置文件并提取需要的配置信息。

好家伙，这是干嘛的呢？**Spring Factories**

 Spring Factories是干嘛的？

小傅哥说可以用它来获取引入我们这个数据库路由组件的服务中的配置文件的内容，我们会指定它的多数据源的配置项，要和我们数据源配置提取方法中规定的设置内容一致。

可以看到小傅哥是这么实现的：

```
@Override
public void setEnvironment(Environment environment) {
    String prefix = "router.jdbc.datasource.";

    dbCount = Integer.valueOf(environment.getProperty(prefix + "dbCount"));
    tbCount = Integer.valueOf(environment.getProperty(prefix + "tbCount"));

    String dataSources = environment.getProperty(prefix + "list");
    for (String dbInfo : dataSources.split(",")) {
        Map<String, Object> dataSourceProps = PropertyUtil.handle(environment, prefix + dbInfo,
        Map.class);
        dataSourceMap.put(dbInfo, dataSourceProps);
    }
}
```

同样的这样也会有之前飞哥在Issue中提到的dataSourceMap会被人为修改的问题：

CSDN-专业IT技术社区-登录

CSDN桌面端登录

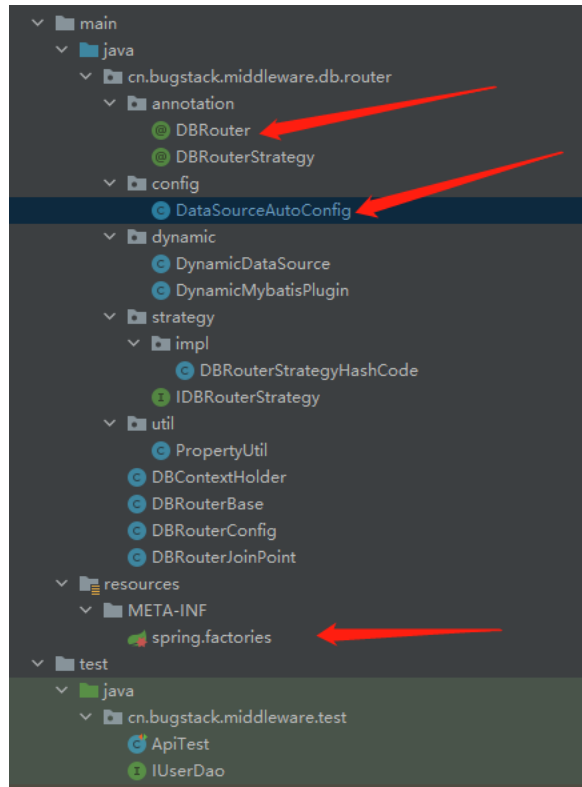
 <https://gitcode.net/KnowledgePlanet/Lottery/-/issues/60>

配置文件做了啥事情呢？

看看 `DataSourceAutoConfig` 类可以知道，它指定了分库的数量，分表的数量，具体的路由字段，数据源的配置组，默认的数据源

在进行第三步数据源处理之前，我们还要重点的分析以下**DataSourceAutoConfig**这个类，它不止完成了配置信息的读取，它还完成了 `dataSource` 数据源的创建和配置，和

到这已经解释清楚了这几个文件是干嘛的：



第三步做好数据源处理

我们已经可以拿到具体的配置信息了，分库分表说白了就是要把数据均匀的散列到各个库表中，那我们肯定要连接好库才可以实现插入数据对不对。这个时候就要解决一个问题：就是要完成动态切换数据源。

那该怎么做呢？

主要就是通过这个抽象类完成的**AbstractRoutingDataSource**

根据用户定义的规则选择当前的数据源，这样我们可以在执行查询之前，设置使用的数据源。实现可动态路由的数据源，在每次数据库查询操作前执行。它的抽象方法 `determineCurrentLookupKey()` 决定使用哪个数据源

官方API大致的意思是：

`AbstractRoutingDataSource`的`getConnection()` 方法根据查找 `lookup key` 键对不同目标数据源的调用，通常是通过(但不一定)某些线程绑定的事物上下文来实现。`AbstractRoutingDataSource`的多数据源动态切换的核心逻辑是：在程序运行时，把数据源数据源通过 `AbstractRoutingDataSource` 动态织入到程序中，灵活的进行数据源切换。

基于`AbstractRoutingDataSource`的多数据源动态切换，可以实现读写分离，这么做缺点也很明显，无法动态的增加数据源。

看一个博主的实现逻辑：

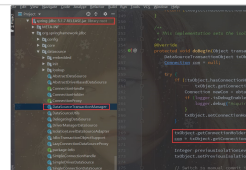
- 定义`DynamicDataSource`类继承抽象类`AbstractRoutingDataSource`，并实现了`determineCurrentLookupKey()`方法。
- 把配置的多个数据源会放在`AbstractRoutingDataSource`的 `targetDataSources`和`defaultTargetDataSource`中，然后通过 `afterPropertiesSet()` 方法将数据源分别进行复制到`resolvedDataSources`和`resolvedDefaultDataSource`中。

- 调用AbstractRoutingDataSource的getConnection()的方法的时候，先调用determineTargetDataSource()方法返回DataSource在进行getConnection()。

spring boot使用AbstractRoutingDataSource实现动态数据源切换_愿风裁尽尘中沙-CSDN博客_abstractroutingdatasource

Spring boot提供了AbstractRoutingDataSource 根据用户定义的规则选择当前的数据源，这样我们可以在执行查询之前，设置使用的数据源。实现可动态路由的数据源，在每次数据库查询操作前执行。它的抽象方法 determineCurrentLookupKey() 决定使用哪个数据源。...

https://blog.csdn.net/qq_37502106/article/details/91044952



小傅哥选择的是更加优雅的创建方式。

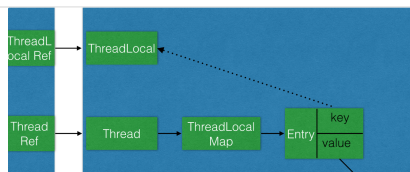


在小傅哥的配置数据源的过程中我们会看到 DBContextHolder 这个用来存放数据源上下文信息的类，它用到了Java中的ThreadLocal，官方的解释大致意思就是ThreadLocal提供了线程内存储变量的能力，这些变量不同之处在于每一个线程读取的变量是对应的互相独立的。通过get和set方法就可以得到当前线程对应的值。说白了就是线程局部变量，每个线程拥有自己的线程局部变量所以是线程安全的。

Java的ThreadLocal详解_tobebetter9527的博客-CSDN博客_java threadlocal

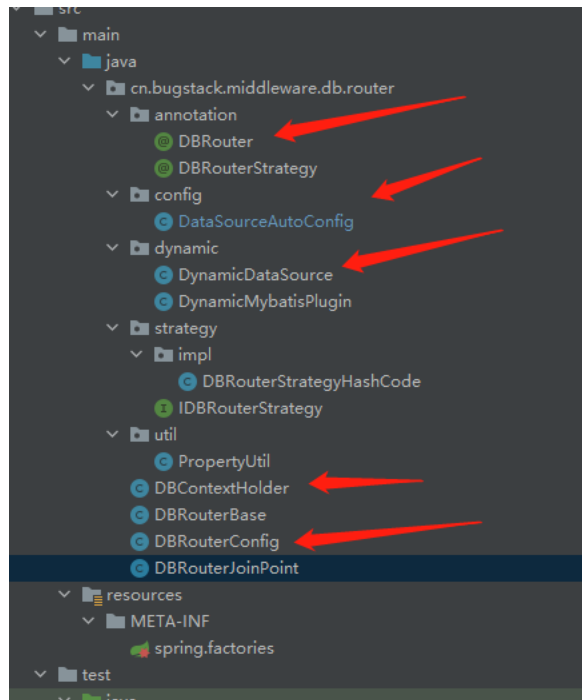
先来看看官方的定义。 This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to

https://blog.csdn.net/qq_39530821/article/details/106042582



到这我们完成了数据源的创建和配置，到第三步为止，我们已经完成了路由注解的定义，路由的配置，数据源的动态配置，那现在还没有做的是针对用了这个注解的方法进行切面操作，然后使用MyBatis完成分表落库的操作。

我们来看看截至目前我们已经分析了那些东西：



第四步完成切面拦截

小傅哥说：

在 AOP 的切面拦截中需要完成；数据库路由计算、扰动函数加强散列、计算库表索引、设置到 ThreadLocal 传递数据源。

- 简化的核心逻辑实现代码如上，首先我们提取了库表乘积的数量，把它当成 HashMap 一样的长度进行使用。
- 接下来使用和 HashMap 一样的扰动函数逻辑，让数据分散的更加散列。
- 当计算完总长度上的一个索引位置后，还需要把这个位置折算到库表中，看看总体长度的索引因为落到哪个库哪个表。
- 最后是把这个计算的索引信息存放到 ThreadLocal 中，用于传递在方法调用过程中可以提取到索引信息。

在具体往下走之前咱复习一下AOP的注解含义：

@Aspect: 把被修饰的类定义一个切面，作用是把被注释修饰的类当成切面给容器读取

@Pointcut：把被修饰的方法定义为一个切点，每个Pointcut的定义包括2部分，一是表达式，二是方法签名。方法签名必须是 public及void型。可以将Pointcut中的方法看作是一个被Advice引用的助记符，因为表达式不直观，因此我们可以通过方法签名的方式为 此表达式命名。因此Pointcut中的方法只需要方法签名，而不需要在方法体内编写实际代码。对应的我们可以知道有十种表达式标签和组成的十二种用法 (<https://zhuanlan.zhihu.com/p/153317556>)

@Around：环绕增强，这里注意它并不是指的是执行两次，

@AfterReturning：后置增强，相当于AfterReturningAdvice，方法正常退出时执行

@Before：标识一个前置增强方法，相当于BeforeAdvice的功能，相似功能的还有

@AfterThrowing：异常抛出增强，相当于ThrowsAdvice

@After: final增强，不管是抛出异常或者正常退出都会执行

其他的都好理解，这个@Around要注意它并不是字面意思切点前执行一次和切点后执行一次，我们来看看官方的解释：

▼ Around advice

runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all.

Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example).

大概的意思是：

它有机会是否在方法执行之前和之后都工作，并确定该方法何时、如何以及是否实际执行。如果您需要在和之前共享状态，则经常使用Around建议方法以线程安全的方式执行(启动和例如停止计时器)。

就是说它只会执行一次，它是为了保证切点执行前后能保证在共享的状态而产生的。这是官方设计这个环绕通知的初衷，它能保证线程安全的方式执行。

具体来看看代码：


定义了路由切面 `DBRouterJoinPoint` 又发现了新的东西，

`@Around ("aopPoint() && @annotation(dbRouter)")` 这种写法

搜了一下看到了这几种写法都是可以的

spring aop 中@annotation()的使用，关于自定义注解，绝壁原创文章_请叫我大师兄-CSDN博客

在自定义个注解之后，通过这个注解，标注需要切入的方法，同时把需要的参数传到切面去。那么我们怎么在切面使用这个注解。我们使用这个自定义注解一方面是为了传一些参数，另一方面也是为了省事。具体怎么省事，看下面的例子就造啦。一般，别人的切面都是这么写的 先声明一个切入点。//切入点签名 @Pointcut("execution(*

 https://blog.csdn.net/qq_27093465/article/details/78804793

原创

小傅哥给出了切面的执行逻辑：

第一步：计算路由

第二步：扰动函数计算具体散列值

第三步：库表索引到哪

第四步：设置到数据源上下文对象中（线程局部变量）

第五步：返回数据源上下文对象

第五步MyBatis拦截器处理分表

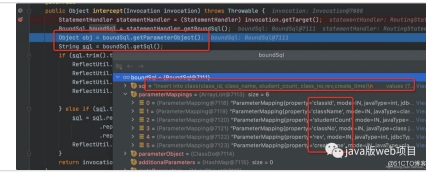
要理解这个我们首先要知道要实现MyBatis自定义拦截器的过程：

自定义实现拦截器mybatis插件,让你为所欲为!【附源码】_wx59a761fc8c542_51CTO博客

首先熟悉一下Mybatis的执行过程,如下图:先说明Mybatis中可以被拦截的类型具体有以下四种:1.Executor:拦截执行器的方法。2.ParameterHandler:拦截参数的处理。3.ResultHandler:拦截结果集的处理。

4.StatementHandler:拦截Sql语法构建的处理。1234 Intercepts注解需要一个Signature(拦截点)参数数组。通过

https://blog.51cto.com/u_13260163/3072097



Mybatis中可以被拦截的类型具体有以下四种:

- 1.Executor:拦截执行器的方法。
- 2.ParameterHandler:拦截参数的处理。
- 3.ResultHandler:拦截结果集的处理。
- 4.StatementHandler:拦截Sql语法构建的处理。

▼ StatementHandler的具体方法

prepare: 用于创建一个具体的 Statement 对象的实现类或者是 Statement 对象

parametersize: 用于初始化 Statement 对象以及对sql的占位符进行赋值

update: 用于通知 Statement 对象将 insert、update、delete 操作推送到数据库

query: 用于通知 Statement 对象将 select 操作推送数据库并返回对应的查询结果

我们具体操作的是第四种, StatementHandler 的prepare对创建的Statement做处理。

说明: @Intercepts: 标识该类是一个拦截器; @Signature: 指明自定义拦截器需要拦截哪一个类型, 哪一个方法; - type: 上述四种类型中的一种; - method: 对应接口中的哪类方法 (因为可能存在重载方法); - args: 对应哪一个方法的入参;

- 实现 Interceptor 接口的 intercept 方法, 获取StatementHandler、通过自定义注解判断是否进行分表操作、获取SQL并替换SQL表名 USER 为 USER_03、最后通过反射修改SQL语句
- 此处会用到正则表达式拦截出匹配的sql, `(from|into|update)[\s]{1,}(\w{1,})`

```
private Pattern pattern = Pattern.compile(" ( from|into|update ) [ \\s ] { 1, } ( \\w { 1, } ) ", Pattern.CASE_INSENSITIVE);
```

像这些正则的写法也是用到了预编译的方式, 比直接用String的方式好很多。