# 读写分离

[Spring 实现数据库读写分离](https://www.cnblogs.com/surge/p/3582248.html)

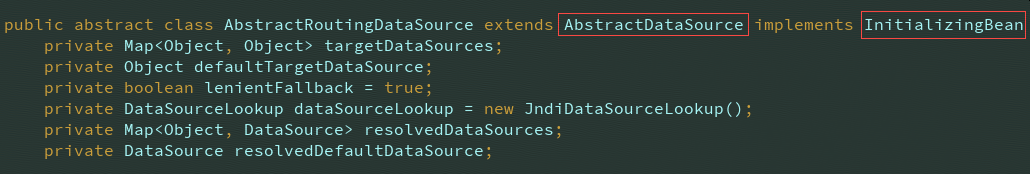
# [Spring AbstractRoutingDataSource 源码分析](https://www.jianshu.com/p/a3bfc6a9aab9)

## 动态数据源切换：

### 原理：

动态数据源切换，就是在程序运行时，把数据源动态织入到程序中，从而选择读取主库还是从库。主要使用的技术是：annotation，Spring AOP ，反射。

AbstractRoutingDataSource类：



其实现了InitializingBean，spring容器启动的时候调用afterPropertiesSet()方法。



其继承了类AbstractDatasource，



其中的方法getConnection在子类AbstractRoutingDataSource中实现，调用了determineTargetDataSource(),内部调用determineCurrentLookupKey()，这个方法是抽象类（获取数据源的key），在子类中实现（这里就是在我们自己写的类DynamicDataSource中实现）。



相当于先把数据源放在resolvedDataSources中，是map结构。通过determineCurrentLookupKey的返回值（map的key）取数据源。

### Flight-base项目中：

1. dao层配置数据源：

<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xmlns="http://www.springframework.org/schema/beans"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <bean id="abstractDataSource" abstract="true" class="com.alibaba.druid.pool.DruidDataSource" init-method="init"  
 destroy-method="close">  
 <property name="maxWait" value="60000"/>  
 <property name="minIdle" value="1"/>  
 <property name="timeBetweenEvictionRunsMillis" value="3000"/>  
 <property name="minEvictableIdleTimeMillis" value="300000"/>  
 <property name="validationQuery" value="SELECT 'x'"/>  
 <property name="testWhileIdle" value="true"/>  
 <property name="testOnBorrow" value="false"/>  
 <property name="testOnReturn" value="false"/>  
 </bean>  
  
 <bean id="dataSourceWrite" parent="abstractDataSource">  
 <property name="url" value="${flight.base.w.jdbc.jdbcUrl}"/>  
 <property name="username" value="${flight.base.w.jdbc.username}"/>  
 <property name="password" value="${flight.base.w.jdbc.password}"/>  
 <!-- 配置初始化大小、最小、最大 -->  
 <property name="initialSize" value="${flight.base.w.jdbc.minConnections}"/>  
 <property name="minIdle" value="5"/>  
 <property name="maxActive" value="${flight.base.w.jdbc.maxConnections}"/>  
 </bean>  
  
 <bean id="dataSourceRead" parent="abstractDataSource">  
 <property name="url" value="${flight.base.r.jdbc.jdbcUrl}"/>  
 <property name="username" value="${flight.base.r.jdbc.username}"/>  
 <property name="password" value="${flight.base.r.jdbc.password}"/>  
 <!-- 配置初始化大小、最小、最大 -->  
 <property name="initialSize" value="${flight.base.r.jdbc.minConnections}"/>  
 <property name="minIdle" value="5"/>  
 <property name="maxActive" value="${flight.base.r.jdbc.maxConnections}"/>  
 </bean>  
  
 <bean id="dataSource" class="com.jd.airplane.infra.mybatis.datasource.DynamicDataSource" lazy-init="true">  
 <property name="writeDataSource" ref="dataSourceWrite"/>  
 <property name="readDataSource" ref="dataSourceRead"/>  
 <!-- 设置默认的目标数据源 -->  
 <property name="defaultTargetDataSource" ref="dataSourceWrite"/>  
 <property name="targetDataSources">  
 <map key-type="java.lang.String" value-type="javax.sql.DataSource">  
 <entry key="WRITE" value-ref="dataSourceWrite"/>  
 <entry key="READ" value-ref="dataSourceRead"/>  
 </map>  
 </property>  
 </bean>  
  
 <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">  
 <property name="dataSource" ref="dataSource"/>  
 <property name="configLocation" value="classpath:mybatis/mybatis-config.xml"/>  
 <property name="mapperLocations" value="classpath:mybatis/mapper/com/jd/airplane/flight/base/model/\*/\*.xml"/>  
 </bean>  
  
 <bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">  
 <constructor-arg index="0" ref="sqlSessionFactory"/>  
 <constructor-arg index="1" value="REUSE"/>  
 </bean>  
</beans>

1. 定义注解：

一个读一个写

*/\*\*  
 \** ***@author*** *yangyanping  
 \** ***@date*** *2018/6/8  
 \** ***@description*** *动态数据源注解  
 \*/*@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
@Documented  
public @interface DataSource {  
 DynamicDataSourceEnum value() default DynamicDataSourceEnum.*READ*;  
}

1. 定义datasourceholder

*/\*\*  
 \* Created by yangyanping on 2018/6/4.  
 \*/*public final class DynamicDataSourceHolder {  
 private static final ThreadLocal<DynamicDataSourceEnum> *holder* = new ThreadLocal<DynamicDataSourceEnum>();  
  
 private DynamicDataSourceHolder() {  
 }  
  
 */\*\*  
 \* 提供给AOP去设置当前的线程的数据源的信息  
 \*/* public static void putDataSource(DynamicDataSourceEnum dataSource) {  
 *holder*.set(dataSource);  
 }  
  
 */\*\*  
 \* 提供给AbstractRoutingDataSource的实现类，通过key选择数据源  
 \*/* public static DynamicDataSourceEnum getDataSource() {  
 return *holder*.get();  
 }  
  
 */\*\*  
 \* 使用默认的数据源  
 \*/* public static void clear() {  
 *holder*.remove();  
 }  
}

1. 定义拦截器切面

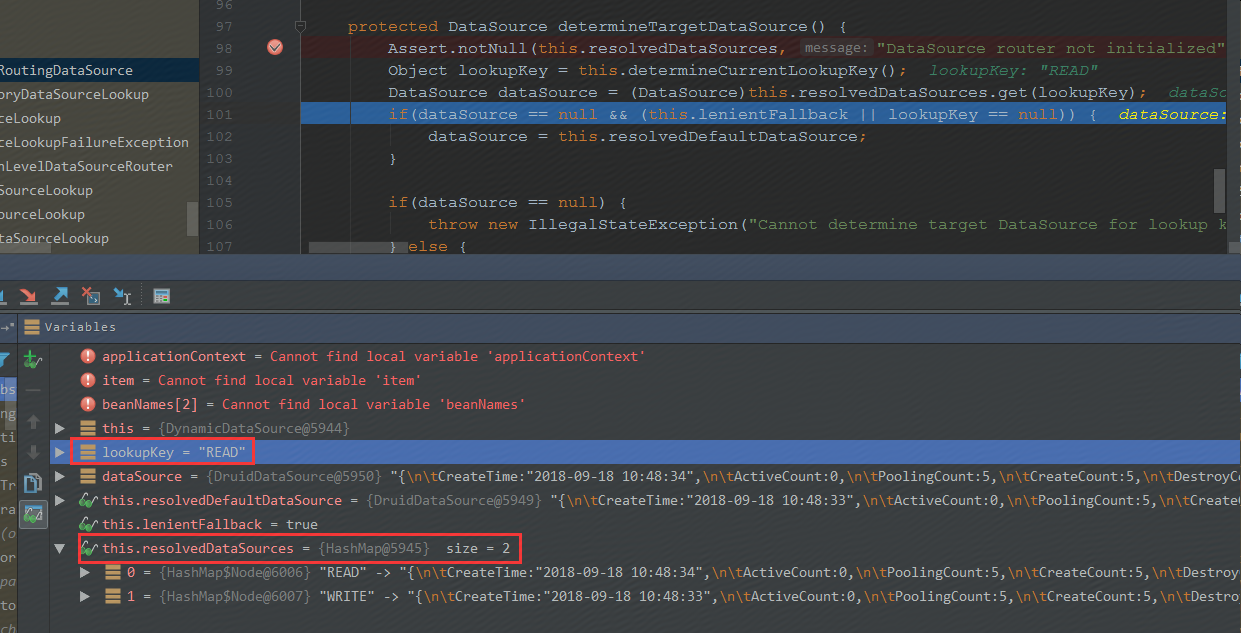
*/\*\*  
 \** ***@author*** *yangyanping.  
 \** ***@date*** *2018/6/8  
 \** ***@description*** *动态数据源拦截器  
 \*/*public class DynamicDataSourceAspect {  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(DynamicDataSourceAspect.class);  
  
 public void pointCut() {  
 }  
  
 public void before(JoinPoint joinPoint) {  
 Method method = ((MethodSignature) joinPoint.getSignature()).getMethod();  
  
 if (method != null && method.isAnnotationPresent(DataSource.class)) {  
 DataSource data = method.getAnnotation(DataSource.class);  
 DynamicDataSourceHolder.*putDataSource*(data.value());  
 }  
 }  
  
 public void after(JoinPoint joinPoint) {  
 DynamicDataSourceHolder.clear();  
 }  
}

1. 实现AbstractRoutingDataSource

*/\*\*  
 \** ***@author*** *yangyanping  
 \** ***@descrption*** *多数据源的选择  
 \** ***@date*** *2018/6/4.  
 \*/*public class DynamicDataSource extends AbstractRoutingDataSource {  
 //写数据源  
 private Object writeDataSource;  
  
 // 读数据源  
 private Object readDataSource;  
  
 public void afterPropertiesSet() {  
 if (this.writeDataSource == null) {  
 throw new IllegalArgumentException("Property 'writeDataSource' is required");  
 }  
  
 Map<Object, Object> targetDataSources = Maps.*newHashMapWithExpectedSize*(2);  
 targetDataSources.put(DynamicDataSourceEnum.*WRITE*.name(), writeDataSource);  
  
 if (readDataSource != null) {  
 targetDataSources.put(DynamicDataSourceEnum.*READ*.name(), readDataSource);  
 }  
  
 // setDefaultTargetDataSource(writeDataSource);  
 // setTargetDataSources(targetDataSources);  
 super.afterPropertiesSet();  
 }  
  
 */\*\*  
 \* 根据Key获取数据源的信息，上层抽象函数的钩子  
 \*/* @Override  
 protected Object determineCurrentLookupKey() {  
 DynamicDataSourceEnum dynamicDataSourceEnum = DynamicDataSourceHolder.*getDataSource*();  
 if (dynamicDataSourceEnum == null) {  
 return DynamicDataSourceEnum.*WRITE*.name();  
 }  
  
 return dynamicDataSourceEnum == DynamicDataSourceEnum.*READ* ?  
 DynamicDataSourceEnum.*READ*.name() : DynamicDataSourceEnum.*WRITE*.name();  
 }  
  
 public void setWriteDataSource(Object writeDataSource) {  
 this.writeDataSource = writeDataSource;  
 }  
  
 public Object getWriteDataSource() {  
 return writeDataSource;  
 }  
  
 public Object getReadDataSource() {  
 return readDataSource;  
 }  
  
 public void setReadDataSource(Object readDataSource) {  
 this.readDataSource = readDataSource;  
 }  
}

运行断点：

启动时先调用DynamicDataSource的afterPropertiesSet方法，再调用父类的afterPropertiesSet方法（因为配置了DynamicDataSource的bean，配置见上面的xml数据源配置，只执行一次）。然后调用determineTargetDataSource(),内部调用determineCurrentLookupKey()【每次执行数据库操作都会调用】



## DefaultObjectFactory

[MyBatis中ObjectFactory简介](https://www.jianshu.com/p/4578322a27d8)

MyBatis 每次创建结果对象实例时,它使用一个 ObjectFactory 实例来完成。默认的ObjectFactory仅会按照配置结果类型的默认构造方法或者指定构造方法来创建对象实例。如果想按照其他方法创建对象，可以重写DefaultObjectFactory或者实现ObjectFactory接口

## 枚举字段与数据表字段的映射、

[**MyBatis里字段到枚举类型的转换/映射**](https://www.cnblogs.com/jeffen/p/6380982.html)

项目中使用：

*/\*\*  
 \* 用于枚举类型和数据库的自动映射  
 \* EnumTypeHandler : 使用枚举字符串名称传递参数  
 \* Created by yangyanping on 2018/3/4.  
 \*/*public class EnumCodeTypeHandler <E extends Enum<E> & IEnumCode<E>> extends EnumTypeHandler<E> {  
 private Class<E> type;  
 public EnumCodeTypeHandler(Class<E> type) {  
 super(type);  
 this.type = type;  
 }  
 @Override  
 public void setNonNullParameter(PreparedStatement preparedStatement, int i, E parameter, JdbcType jdbcType) throws SQLException {  
 preparedStatement.setInt(i, parameter.getCode());  
 }  
 @Override  
 public E getNullableResult(ResultSet resultSet, String columnName) throws SQLException {  
 //使用列名获取数据  
 Integer code = resultSet.getInt(columnName);  
  
 return parse(code);  
 }  
 @Override  
 public E getNullableResult(ResultSet resultSet, int columnIndex) throws SQLException {  
 //使用列下标获取数据  
 Integer code = resultSet.getInt(columnIndex);  
  
 return parse(code);  
 }  
 @Override  
 public E getNullableResult(CallableStatement callableStatement, int columnIndex) throws SQLException {  
 Integer code = callableStatement.getInt(columnIndex);  
  
 return parse(code);  
 }  
 private E parse(Integer code) {  
 if (null == code) {  
 return null;  
 }  
 for (E e : type.getEnumConstants()) {  
 if (code.equals(e.getCode())) {  
 return e;  
 }  
 }  
 return null;  
 }  
}

具体枚举类使用上面handler

public enum UserLevelEnum implements IEnumCode<UserLevelEnum> {

public interface IEnumCode<E extends Enum<E>> {  
 int getCode();  
}

public class UserLevelEnumCodeHandler extends EnumCodeTypeHandler<UserLevelEnum> {  
 public UserLevelEnumCodeHandler() {  
 super(UserLevelEnum.class);  
 }  
}

## <choose>与<if>

[MyBatis-动态SQL的if、choose、when、otherwise、trim、where、set、foreach使用](https://blog.csdn.net/qq_32588349/article/details/51541871)

[mybatis】IF判断的坑](http://cheng-xinwei.iteye.com/blog/2008200)

[MyBatis中的OGNL教程](https://blog.csdn.net/isea533/article/details/50061705)

有时候我们并不想应用所有的条件，而只是想从多个选项中选择一个。而使用if标签时，只要test中的表达式为 true，就会执行 if 标签中的条件。MyBatis 提供了 choose 元素。if标签是与(and)的关系，而 choose 是或(or)的关系。

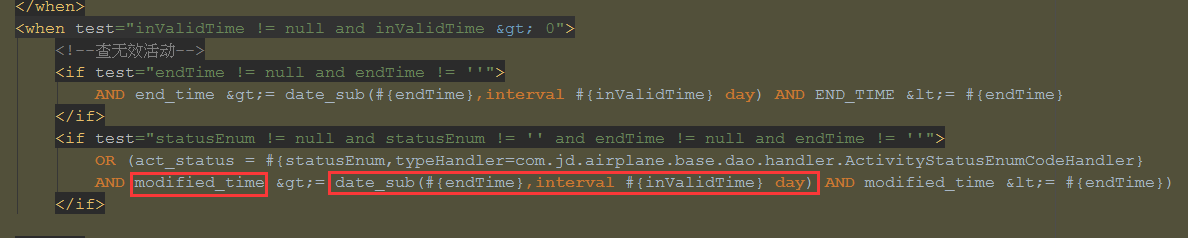
choose标签是按顺序判断其内部when标签中的test条件出否成立，如果有一个成立，则 choose 结束。当 choose 中所有 when 的条件都不满则时，则执行 otherwise 中的sql。类似于Java 的 switch 语句，choose 为 switch，when 为 case，otherwise 则为 default。

Mybatis中等于是==或eq

项目中使用

Java中属性是long类型，数据库中字段是dateTime类型，比较时候使用unix\_timestamp函数将dateTime转成时间戳

Java中属性时Date类型，数据库中是dateTime类型，比较时候直接使用>=符号，运算使用函数



<choose>  
 <when test="inValidTime == null or inValidTime == 0">  
 <!--查有效活动-->  
 <if test="endTime != null and endTime != ''">  
 AND END\_TIME &gt;= #{endTime}  
 </if>  
 </when>  
 <when test="inValidTime != null and inValidTime &gt; 0">  
 <!--查无效活动-->  
 <if test="endTime != null and endTime != ''">  
 AND END\_TIME &gt;= #{endTime}-#{inValidTime}\*86400000 AND END\_TIME &lt;= #{endTime}  
 </if>  
 <if test="statusEnum != null and statusEnum != '' and endTime != null and endTime != ''">  
 OR (ACT\_STATUS = #{statusEnum,typeHandler=com.jd.airplane.base.dao.handler.ActivityStatusEnumCodeHandler}  
 AND unix\_timestamp(MODIFIED)\*1000 &gt;= #{endTime}-#{inValidTime}\*86400000 AND unix\_timestamp(MODIFIED)\*1000 &lt;= #{endTime})  
 </if>  
 </when>  
 <otherwise>  
 <!--不满足前两个条件则查全部活动-->  
 <if test="beginTime != null and beginTime != ''">  
 AND BEGIN\_TIME = #{beginTime}  
 </if>  
 <if test="endTime != null and endTime != ''">  
 AND END\_TIME = #{endTime}  
 </if>  
 </otherwise>  
</choose>

## #与$符号区别

1. 介绍  
    mybatis 中使用 Mapper.xml里面的配置进行 sql 查询，经常需要动态传递参数，例如我们需要根据用户的姓名来筛选用户时，sql 如下：  
   select \* from user where name = "Jack";    
   上述 sql 中，我们希望 name 后的参数 "Jack" 是动态可变的，即不同的时刻根据不同的姓名来查询用户。在 Mapper.xml文件中使用如下的 sql 可以实现动态传递参数 name：  
   select \* from user where name = #{name};    
   或者是：  
   select \* from user where name = ${name};    
   二、$ 与 #  
   1. 区别 :  
   动态 SQL 是 mybatis 的强大特性之一，也是它优于其他 ORM 框架的一个重要原因。mybatis 在对 sql 语句进行预编译之前，会对 sql 进行动态解析，解析为一个 BoundSql 对象，也是在此处对动态 SQL 进行处理的。在动态 SQL 解析阶段， #{ } 和 ${ } 会有不同的表现。  
   #{ }：解析为一个 JDBC 预编译语句（prepared statement）的参数标记符。  
   例如，Mapper.xml中如下的 sql 语句：  
   select \* from user where name = #{name};    
   动态解析为：  
   select \* from user where name = ?;    
   一个 #{ } 被解析为一个参数占位符 ? 。  
   而${ } 仅仅为一个纯碎的 string 替换，在动态 SQL 解析阶段将会进行变量替换。  
   例如，Mapper.xml中如下的 sql：  
   select \* from user where name = ${name};    
   当我们传递的参数为 "Jack" 时，上述 sql 的解析为：  
   select \* from user where name = "Jack";    
   预编译之前的 SQL 语句已经不包含变量了，完全已经是常量数据了。  
   综上所得， ${ } 变量的替换阶段是在动态 SQL 解析阶段，而 #{ }变量的替换是在 DBMS 中。  
   三、用法  
   1、能使用 #{ } 的地方就用 #{ }  
   首先这是为了性能考虑的，相同的预编译 sql 可以重复利用。其次，${ } 在预编译之前已经被变量替换了，这会存在 sql 注入问题。例如，如下的 sql：  
   select \* from ${tableName} where name = #{name}    
   假如，我们的参数 tableName 为 user; delete user; --，那么 SQL 动态解析阶段之后，预编译之前的 sql 将变为：  
   select \* from user; delete user; -- where name = ?;    
   -- 之后的语句将作为注释，不起作用，因此本来的一条查询语句偷偷的包含了一个删除表数据的 SQL。  
   2. 表名作为变量时，必须使用 ${ }  
   这是因为，表名是字符串，使用 sql 占位符替换字符串时会带上单引号 ''，这会导致 sql 语法错误，例如：  
   select \* from #{tableName} where name = #{name};    
   预编译之后的sql 变为：  
   select \* from ? where name = ?;    
   假设我们传入的参数为 tableName = "user" , name = "Jack"，那么在占位符进行变量替换后，sql 语句变为：  
   select \* from 'user' where name='Jack';    
   上述 sql 语句是存在语法错误的，表名不能加单引号 ''（注意，反引号 ``是可以的）。  
   四、sql预编译  
   1. 定义：  
   sql 预编译指的是数据库驱动在发送 sql 语句和参数给 DBMS 之前对 sql 语句进行编译，这样 DBMS 执行 sql 时，就不需要重新编译。  
   2. 为什么需要预编译  
   JDBC 中使用对象 PreparedStatement 来抽象预编译语句，使用预编译。预编译阶段可以优化 sql 的执行。预编译之后的 sql 多数情况下可以直接执行，DBMS 不需要再次编译，越复杂的sql，编译的复杂度将越大，预编译阶段可以合并多次操作为一个操作。预编译语句对象可以重复利用。把一个 sql 预编译后产生的 PreparedStatement 对象缓存下来，下次对于同一个sql，可以直接使用这个缓存的 PreparedState 对象。mybatis 默认情况下，将对所有的 sql 进行预编译。

## 无参构造函数

因为mybatis中，查询数据表时，会生成对象的cglib代理（通过.class字节码生成子类），所以需要无参构造函数

## parameterType

selectList方法：

public <E> List<E> selectList(String statement, Object parameter, RowBounds rowBounds) {  
 return this.sqlSessionProxy.selectList(statement, parameter, rowBounds);  
}

参数：parameter类型是Object，可以是map，对象，list，数组，及基本类型

传入参数与，mapper.xml文件中的使用关系见下面链接

可以传递一个List或Array类型的对象作为参数,MyBatis会自动的将List或Array对象包装到一个Map对象中,List类型对象会使用list作为键名,而Array对象会用array作为键名。

[MyBatis传入参数与parameterType](https://www.cnblogs.com/ruiati/p/6410339.html)

Foreach属性

| **属性** | **描述** |
| --- | --- |
| item | 循环体中的具体对象。支持属性的点路径访问，如item.age,item.info.details。 具体说明：在list和数组中是其中的对象，在map中是value。 该参数为必选。 |
| collection | 要做foreach的对象，作为入参时，List<?>对象默认用list代替作为键，数组对象有array代替作为键，Map对象没有默认的键。 当然在作为入参时可以使用@Param("keyName")来设置键，设置keyName后，list,array将会失效。 除了入参这种情况外，还有一种作为参数对象的某个字段的时候。举个例子： 如果User有属性List ids。入参是User对象，那么这个collection = "ids" 如果User有属性Ids ids;其中Ids是个对象，Ids有个属性List id;入参是User对象，那么collection = "ids.id" 上面只是举例，具体collection等于什么，就看你想对那个元素做循环。 该参数为必选。 |
| separator | 元素之间的分隔符，例如在in()的时候，separator=","会自动在元素中间用“,“隔开，避免手动输入逗号导致sql错误，如in(1,2,)这样。该参数可选。 |
| open | foreach代码的开始符号，一般是(和close=")"合用。常用在in(),values()时。该参数可选。 |
| close | foreach代码的关闭符号，一般是)和open="("合用。常用在in(),values()时。该参数可选。 |
| index | 在list和数组中,index是元素的序号，在map中，index是元素的key，该参数可选。 |

### 项目中使用：

Java调用：参数是

Map<String, DictEnum> parameter = Collections.*singletonMap*("dict", dict);

而DictEnum中的属性

public enum DictEnum {  
 //机场  
 *AIRPORT*("dict\_airport", new HashMap<String, String>() {  
 {  
 put(*CODE*, "code");  
 put(*CHINESE\_NAME*, "name");  
 put(*CITY\_CODE*, "city\_code");  
 put(*CITY\_NAME*, "city\_name");  
 put(*PINYIN*, "spell\_name");  
 put(*SHORT\_PINYIN*, "simple\_spell\_name");  
 }  
 })

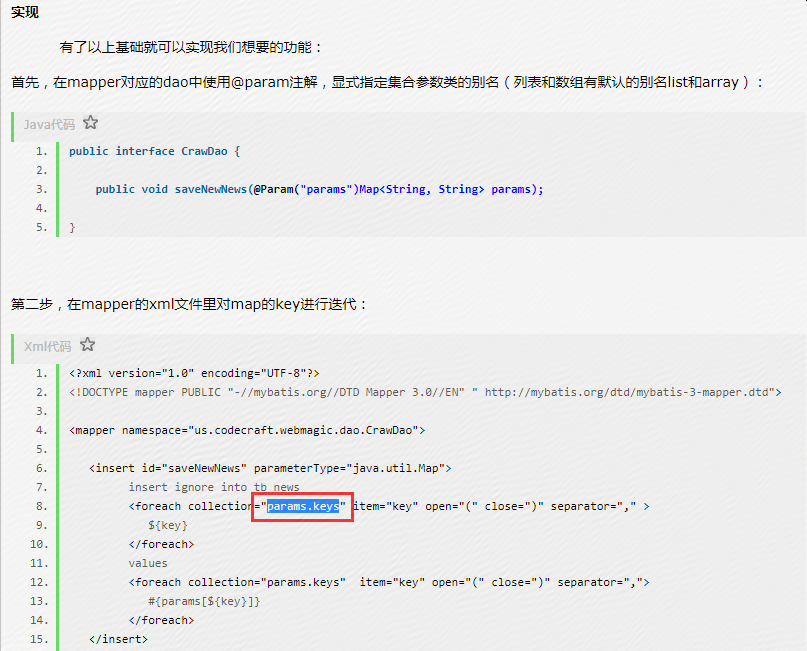
DictEnum(String table, Map<String, String> fieldMap) {  
 this.table = table;  
 this.fieldMap = Collections.*unmodifiableMap*(fieldMap);  
}

调用

list = dictSqlSessionTemplate.selectList(*SELECT\_ITEM\_LIST*, parameter, new RowBounds(offset, limit));

mapper.xml中collection="dict.fieldMap.keySet()"也可以

<select id="selectItemList" parameterType="map" resultType="java.util.Map">  
 SELECT id  
 <foreach collection="dict.fieldMap.keys" item="key">  
 ,${dict.fieldMap[key]} as ${key}  
 </foreach>  
 FROM ${dict.table}  
 WHERE  
 is\_delete = 0  
 ORDER BY id  
</select>



## 分页

[Mybatis Interceptor 拦截器原理 源码分析](https://www.cnblogs.com/daxin/p/3541922.html)

Mybatis采用责任链模式，通过动态代理组织多个拦截器（插件），通过这些拦截器可以改变Mybatis的默认行为（诸如SQL重写之类的），由于插件会深入到Mybatis的核心，因此在编写自己的插件前最好了解下它的原理，以便写出安全高效的插件。

Mybatis支持对Executor、StatementHandler、PameterHandler和ResultSetHandler进行拦截，也就是说会对这4种对象进行代理。

### 项目中应用

[MyBatis实现物理分页的实例](https://www.jb51.net/article/103323.htm)

#### 实现intercepter的插件

*\*分页拦截器，用于拦截需要进行分页查询的操作，然后对其进行分页处理。  
\* 利用拦截器实现Mybatis分页的原理：  
\* 要利用JDBC对数据库进行操作就必须要有一个对应的Statement对象，Mybatis在执行Sql语句前就会产生一个包含Sql语句的Statement对象，而且对应的Sql语句  
\* 是在Statement之前产生的，所以我们就可以在它生成Statement之前对用来生成Statement的Sql语句下手。在Mybatis中Statement语句是通过RoutingStatementHandler对象的  
\* prepare方法生成的。所以利用拦截器实现Mybatis分页的一个思路就是拦截StatementHandler接口的prepare方法，然后在拦截器方法中把Sql语句改成对应的分页查询Sql语句，之后再调用  
\* StatementHandler对象的prepare方法，即调用invocation.proceed()。  
\* 对于分页而言，在拦截器里面我们还需要做的一个操作就是统计满足当前条件的记录一共有多少，这是通过获取到了原始的Sql语句后，把它改为对应的统计语句再利用Mybatis封装好的参数和设置参数的功能把Sql语句中的参数进行替换，之后再执行查询记录数的Sql语句进行总记录数的统计。*

要实现分页拦截器，首先创建类PageInterceptor实现intercepter接口，在类PageInterceptor上需要通过打标注解

@Intercepts({@Signature(type = StatementHandler.class, method = "prepare", args = {Connection.class})})

拦截器的实现：

public class PageInterceptor implements Interceptor {  
 private static final Logger *LOGGER* = LoggerFactory.*getLogger*(PageInterceptor.class);  
 private static final ObjectFactory *DEFAULT\_OBJECT\_FACTORY* = new DefaultObjectFactory();  
 private static final ObjectWrapperFactory *DEFAULT\_OBJECT\_WRAPPER\_FACTORY* = new DefaultObjectWrapperFactory();  
 private static final Pattern *hasLimit* = Pattern.*compile*("^.\*\\sLIMIT\\s+\\d+(,\\s\*\\d+)?$");  
  
 public Object intercept(Invocation invocation) throws Throwable {  
 StatementHandler statementHandler = (StatementHandler) invocation.getTarget();  
  
 MetaObject metaStatementHandler = MetaObject.*forObject*(statementHandler, *DEFAULT\_OBJECT\_FACTORY*,  
 *DEFAULT\_OBJECT\_WRAPPER\_FACTORY*);  
 RowBounds rowBounds = (RowBounds) metaStatementHandler.getValue("delegate.rowBounds");  
  
 if (null == rowBounds || RowBounds.*NO\_ROW\_LIMIT* == rowBounds.getLimit()) {  
 return invocation.proceed();  
 }  
  
 // 分离代理对象链(由于目标类可能被多个拦截器拦截，从而形成多次代理，通过下面的两次循环可以分离出最原始的的目标类)  
 while (metaStatementHandler.hasGetter("h")) {  
 Object object = metaStatementHandler.getValue("h");  
 metaStatementHandler = MetaObject.*forObject*(object, *DEFAULT\_OBJECT\_FACTORY*, *DEFAULT\_OBJECT\_WRAPPER\_FACTORY*);  
 }  
  
 // 分离最后一个代理对象的目标类  
 while (metaStatementHandler.hasGetter("target")) {  
 Object object = metaStatementHandler.getValue("target");  
 metaStatementHandler = MetaObject.*forObject*(object, *DEFAULT\_OBJECT\_FACTORY*, *DEFAULT\_OBJECT\_WRAPPER\_FACTORY*);  
 }  
  
 //通过反射获取delegate父类BaseStatementHandler的mappedStatement属性  
 MappedStatement mappedStatement = (MappedStatement) metaStatementHandler.getValue("delegate.mappedStatement");  
  
 // 只重写需要分页的sql语句。只匹配SELECT  
 if (SqlCommandType.*SELECT*.equals(mappedStatement.getSqlCommandType())) {  
 BoundSql boundSql = (BoundSql) metaStatementHandler.getValue("delegate.boundSql");  
 String sql = boundSql.getSql().trim().replaceAll(";\*$", "");  
 if (!*hasLimit*.matcher(sql.toUpperCase()).matches()) {  
 // 重写sql  
 String pageSql = sql + " LIMIT " + rowBounds.getOffset() + "," + rowBounds.getLimit();  
  
 metaStatementHandler.setValue("delegate.boundSql.sql", pageSql);  
 // 采用物理分页后，就不需要mybatis的内存分页了，所以重置下面的两个参数  
 metaStatementHandler.setValue("delegate.rowBounds.offset", RowBounds.*NO\_ROW\_OFFSET*);  
 metaStatementHandler.setValue("delegate.rowBounds.limit", RowBounds.*NO\_ROW\_LIMIT*);  
 }  
 }  
  
 // 将执行权交给下一个拦截器  
 return invocation.proceed();  
 }  
  
 */\*\*  
 \* Mybatis拦截器只能拦截四种类型的接口：Executor、StatementHandler、ParameterHandler和ResultSetHandler。  
 \* 当目标类是StatementHandler类型时，才包装目标类，否者直接返回目标本身,减少目标被代理的次数  
 \*/* public Object plugin(Object target) {  
 if (!(target instanceof StatementHandler)) {  
 return target;  
 }  
  
 return Plugin.*wrap*(target, this);  
 }  
  
 */\*\*  
 \* setProperties方法是用于在Mybatis配置文件中指定一些属性的  
 \*/* public void setProperties(Properties properties) {  
  
 }

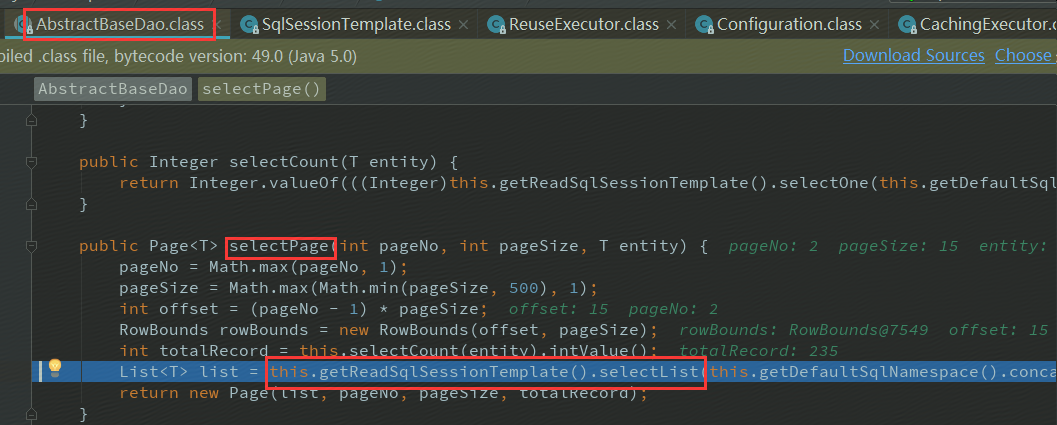
#### 源码执行过程

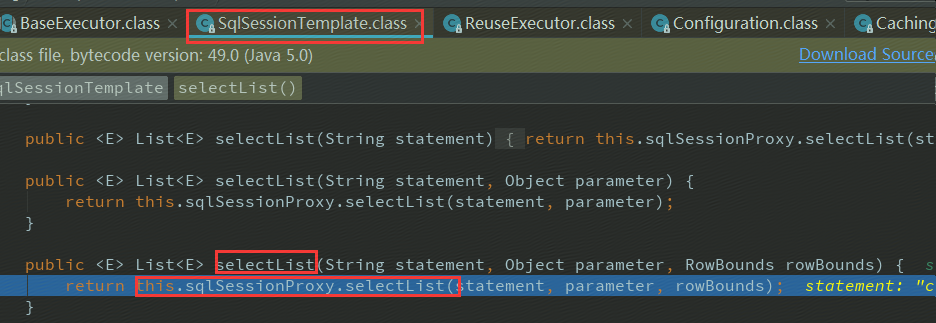
Intercepts注解的源码：

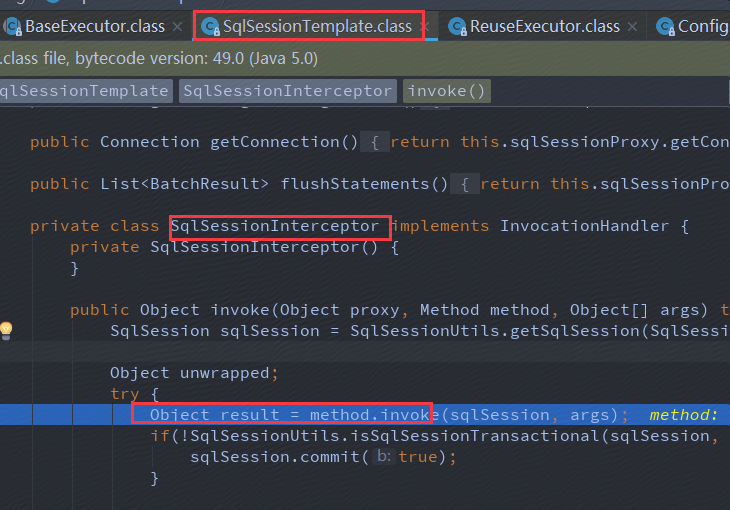
package org.apache.ibatis.plugin;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE})  
public @interface Intercepts {  
 Signature[] value();  
}

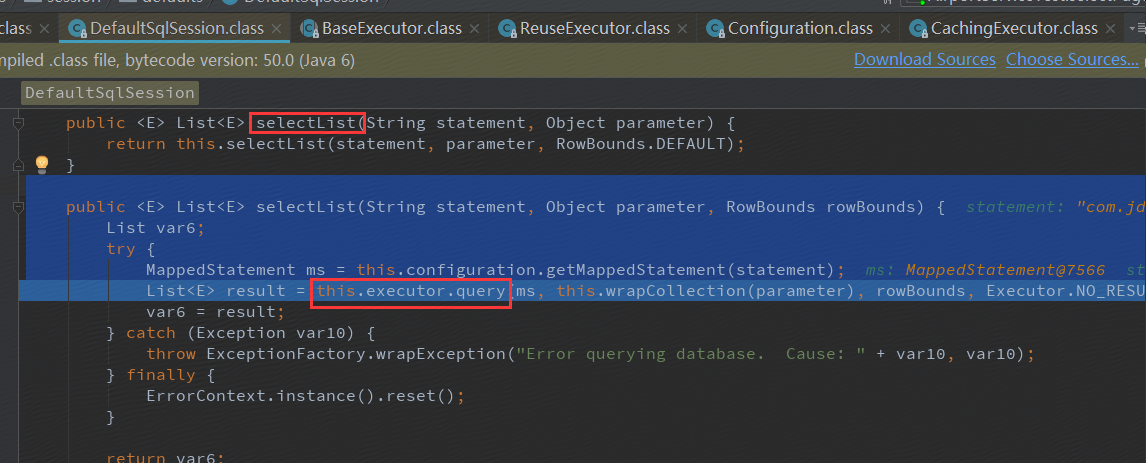
Signature注解源码：

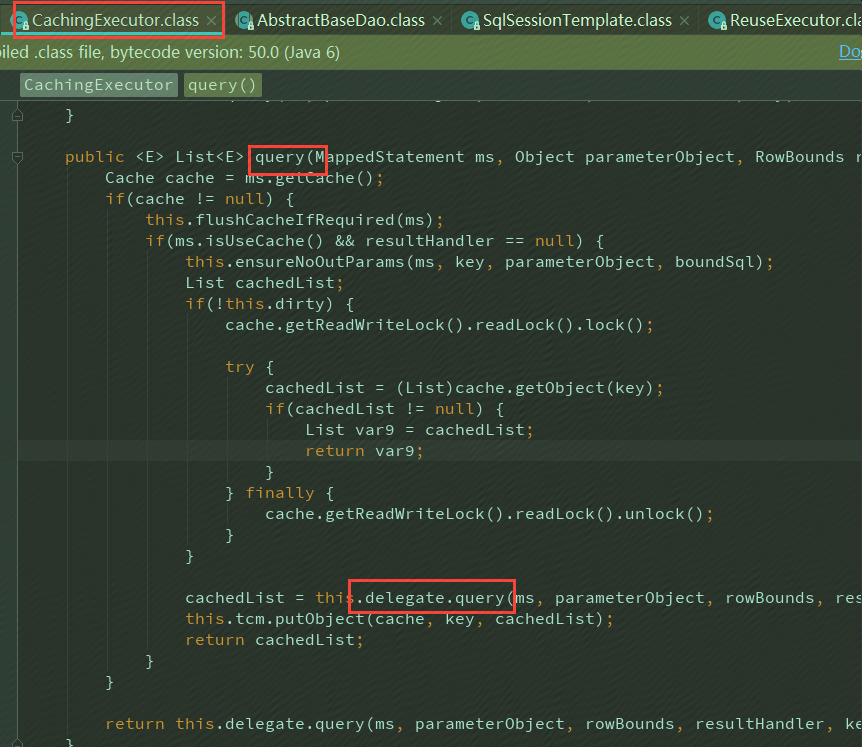
package org.apache.ibatis.plugin;  
  
import java.lang.annotation.ElementType;  
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE})  
public @interface Signature {  
 Class<?> type();  
  
 String method();  
  
 Class<?>[] args();  
}



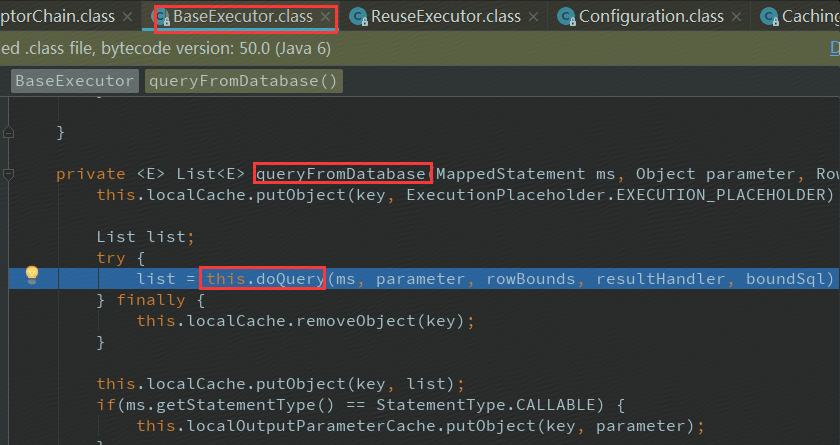


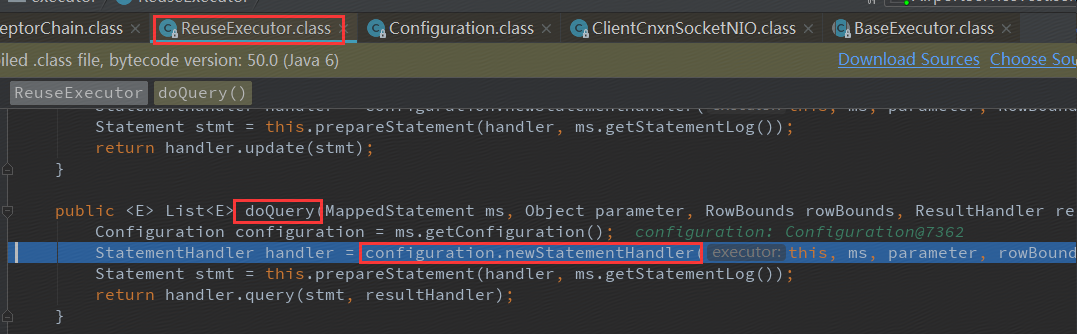
















## 打印sql语句

[mybatis 打印SQL语句](https://www.cnblogs.com/zsg88/p/7903735.html)