MyBatis源码相对于Spring应该是层次比较清晰，容易理解的；其实简单来说就是解耦、封装，让开发者更关注业务层的开发，实现简单而又方便的调用。

MyBatis大概两部分内容：xml文件加载初始化、mapper接口动态代理加强。

注：以下介绍主要是基于使用注解的SpringBoot配置。

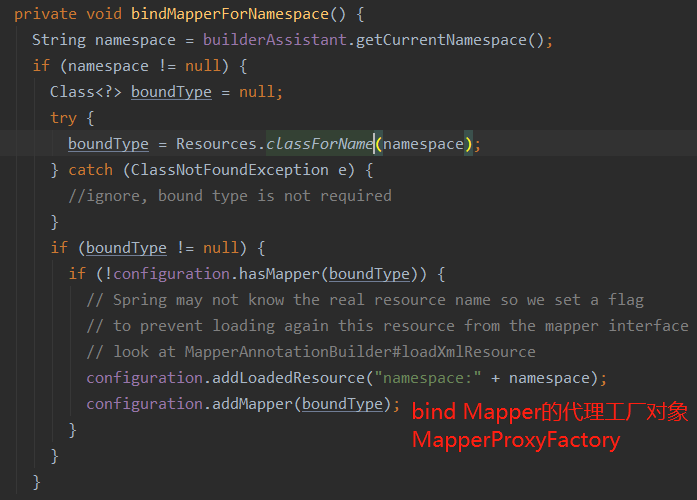
1. xml文件加载初始化

XMLConfigBuilder：解析Mybatis-config.xml文件节点</configuration>；这一块可以忽略，因为现在几乎都是Spring集成Mybatis，通过application.properties来配置，跳过config文件解析直接进入第二阶段XMLMapperBuilder解析mapper.xml文件。

XMLMapperBuilder：解析mapper.xml文件节点</mapper>；Mybatis结合springBoot或者Spring使用时，在配置dataSource创建SqlSessionFactory时启动的XMLMapperBuilder对象。主要有ResultMap、Sql节点等。

XMLStatementBuilder：解析select|insert|update|delete，封装成MappedStatement。

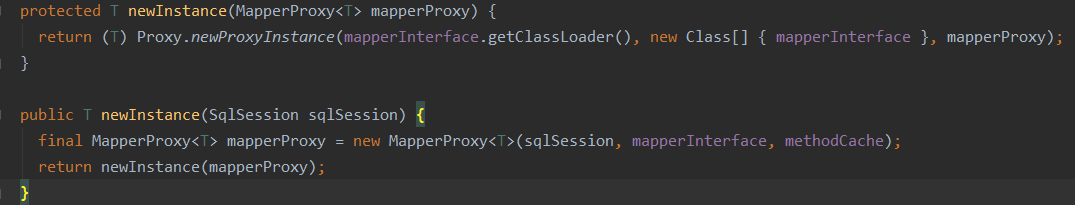
最后绑定mapper class和对代理工厂对象，mapperRegistry是其注册中心；创建Mapper接口的代理工厂类MapperProxyFactory，该类主要为Mapper接口方法生成代理类（MapperProxy（JDK动态代理）），已进行增强。



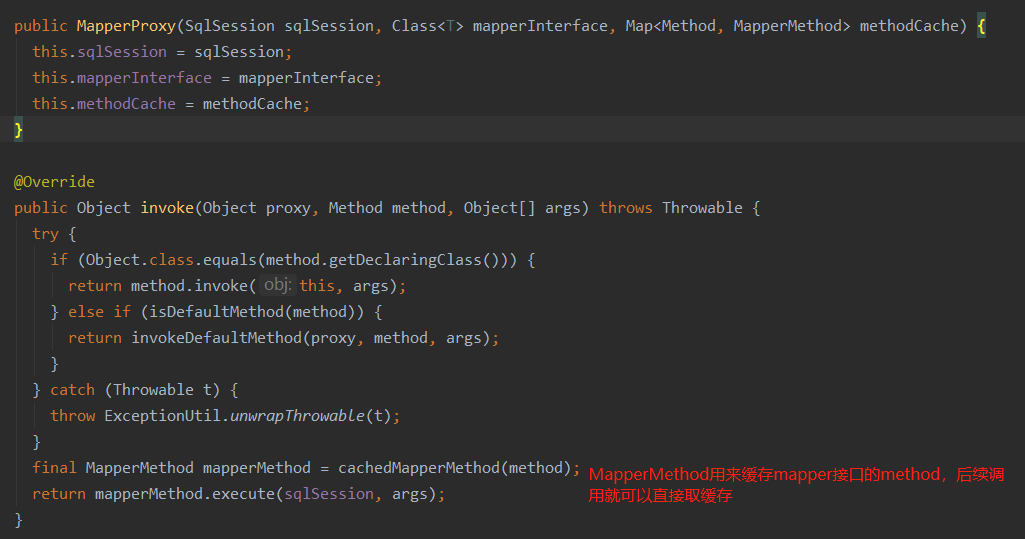
1. Mapper接口动态代理加强

问题思考：Mybatis是如何通过Mapper接口进行sql操作的？

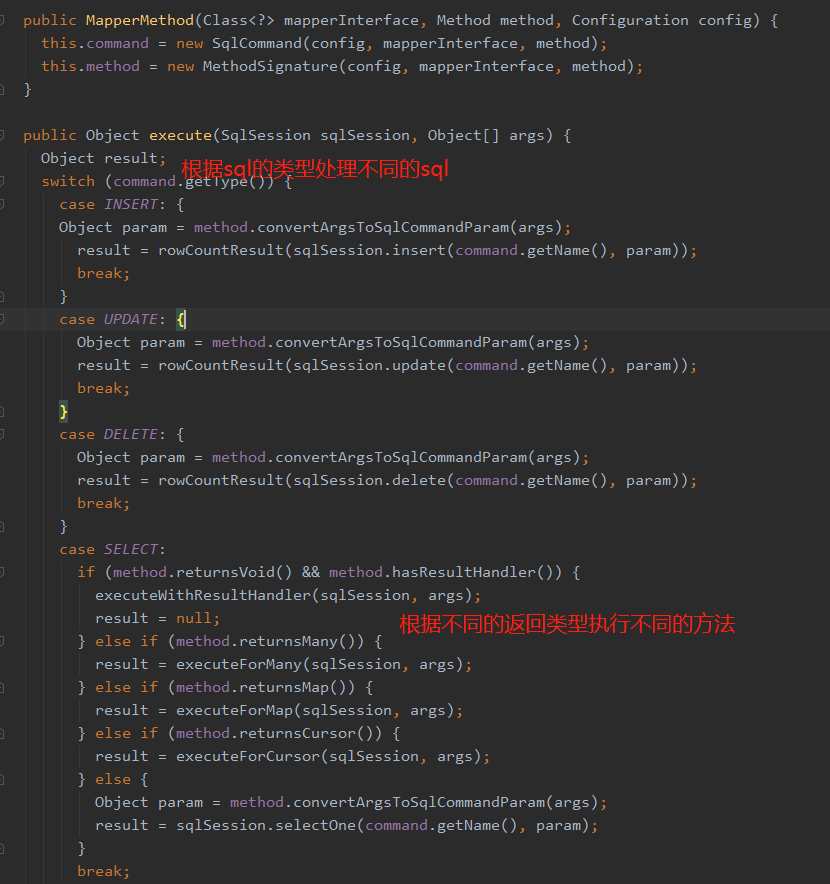
MapperProxy就是Mapper接口的代理类，这个代理类由代理工厂类MapperProxyFactory生成如图所示（此工厂类在第一步中已经将Mapper接口类型作为key进行缓存），MapperProxy的必须属性参数有mapper相关的sqlsession，mapper接口类型Class。



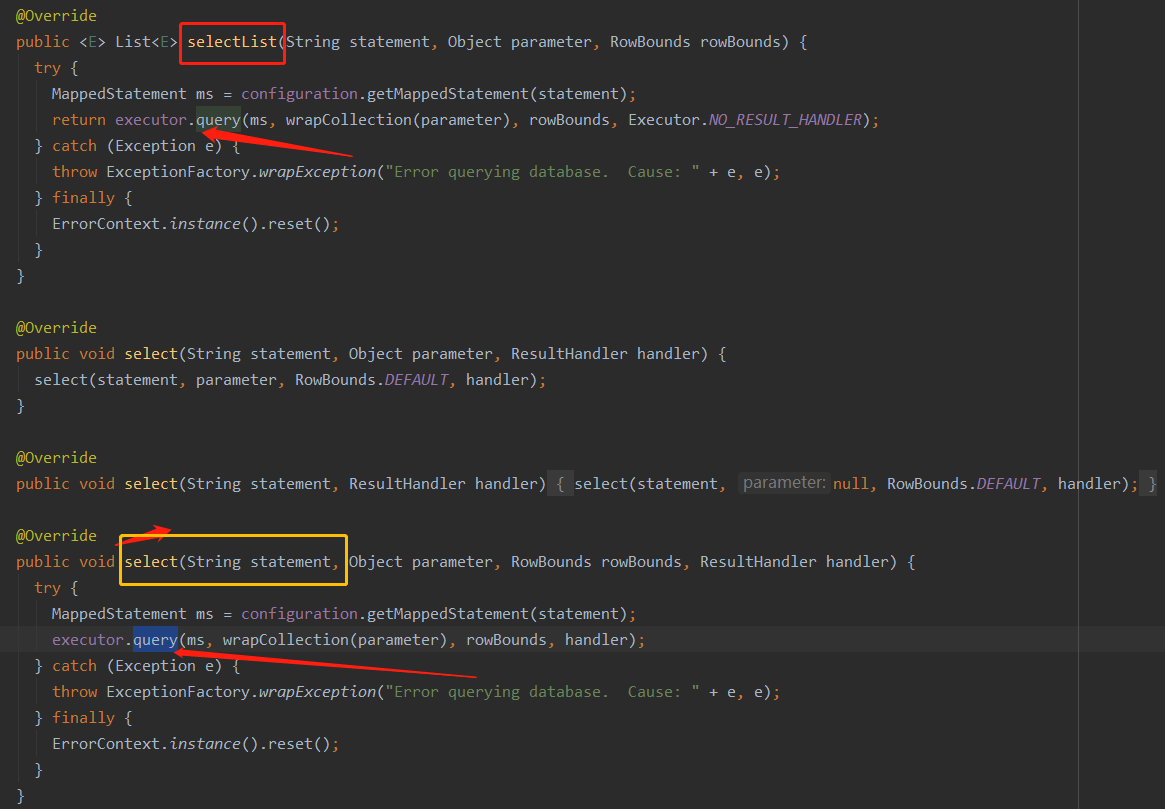
1、动态代理：当调用Mapper接口中的方法就会交由MapperProxy进行代理，



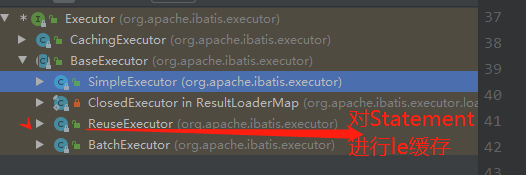
method.execute(sqlsession,args)真正调用sqlSession的地方。走进去看一下sqlSession触发不同的sql类型，但是对于select由于返回类型可能不同，再继续处理。



2、Executor执行：其实最终还是要使用Executor执行的

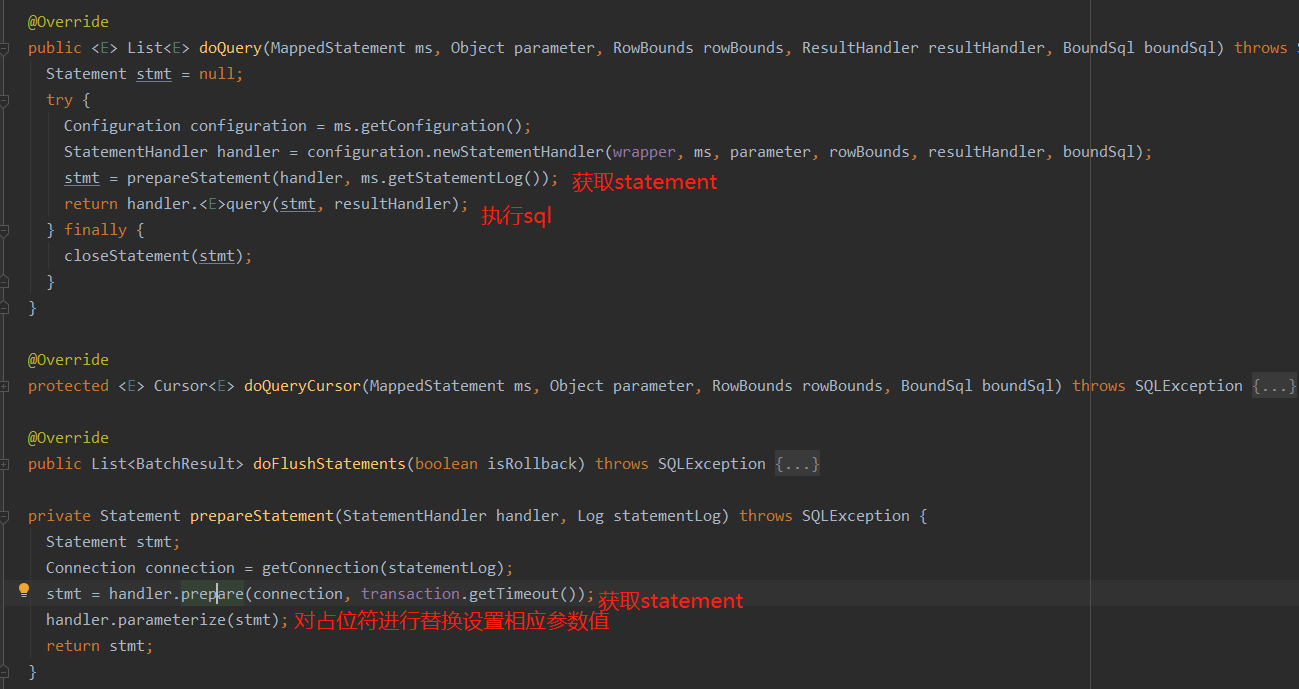


Executor有两个实现类，类图如下；Executor是由Configuration创建的，并且使用了装饰器模式以及动态代理进行加强（interceptorChain.pluginAll(executor)）

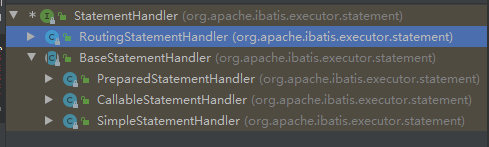


以SimpleExecutor为例看其处理过程：

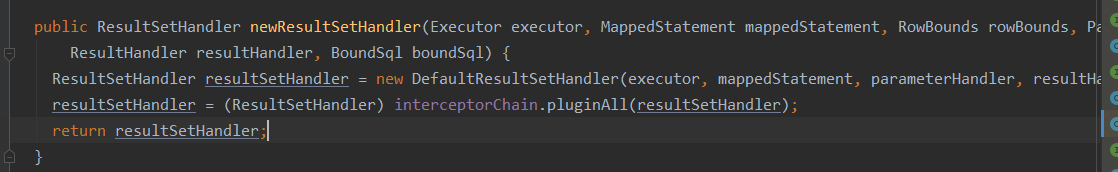
下图中doQuery()方法第四行代码configuration.newStatementHandler():在Configuration类中通过RoutingStatementHandler（静态代理）根据MappedStatement的StatementType（默认PREPARED）生成哪一种StatementHandler（类图如下），



StatementHandler继承关系：



3、ResultSetHandler解析查询结果：结果解析（ResultSetHandler）唯一默认实现类DefaultResultSetHandler，也会被加强；

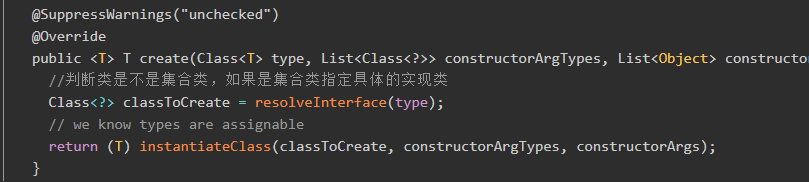


1. Mybatis涉及的模块以及设计模式使用：
2. 反射模块：查**询**的结果ResultSet通过反射构建bean。（实例化目标对象和对象属性赋值需要反射生成）---工厂模式

**反射的核心类：**

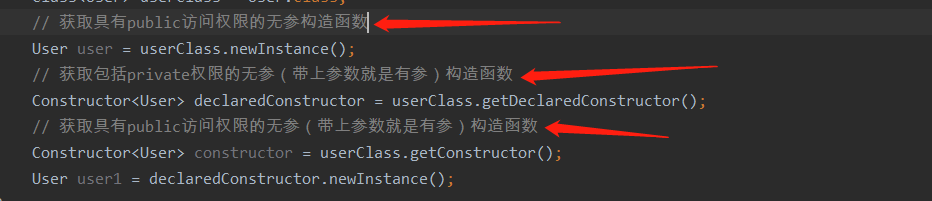
* **Reflector**：存放class定义的属性、方法信息
* **DefaultReflectorFactory**：依据class类新创建Reflector并缓存
* **MetaClass**：引用ReflectorFactory、Reflector，可以获取类的元信息
* **ObjectWrapper**：对bean、MetaClass的封装，可以实现对bean属性的设值以及获取
* **DefaultObjectFactory**：实例化对象
* **MetaObject**：主要封装了**bean**、**ObjectWrapper、DefaultObjectFactory、DefaultReflectorFactory，因此可以通过MetaObject设置和获取bean的属性。（主要是通过ObjectWrapper操作）**
* 用于实例化目标对象的类

ObjectFactory：MyBatis每次创建结果对象的新实例时，使用ObjectFactory构建POJO，默认实现**DefaultObjectFactory（反射实例化对象）**



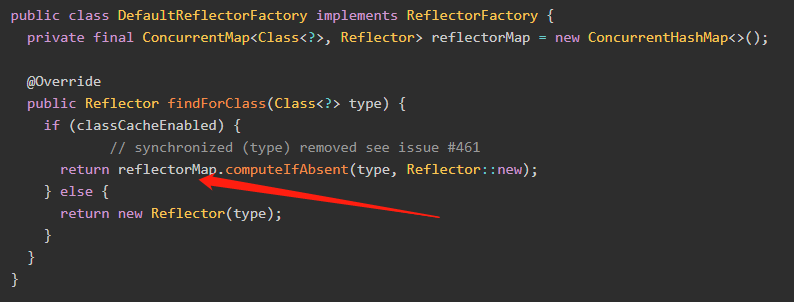


反射创建对象（注意区别）：

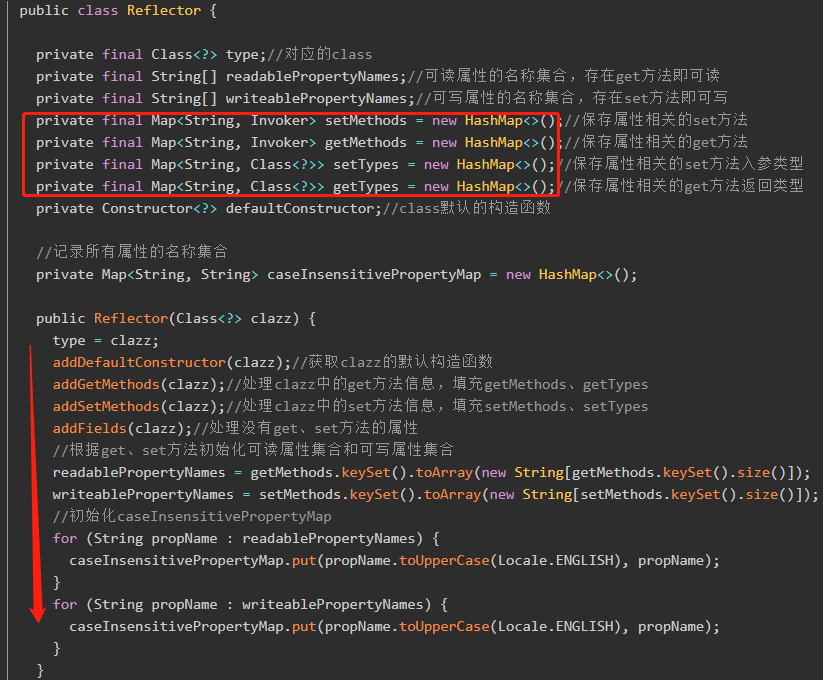


* 用于对象属性赋值的类

ReflectorFactory：创建Reflector的工厂类，Reflector是MyBatis反射模块的基础，每个Reflector对象都对应一个类，在其中缓存了反射操作所需要的类元信息。默认实现类DefaultReflectorFactory（**读取类元信息封装在Reflector中，每一个类都有一个Reflector**）。



Reflector(核心元信息缓存类)：



MetaObject：

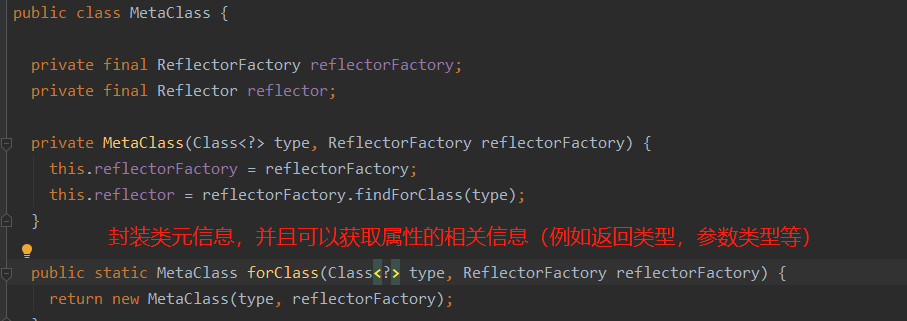
MetaObject封装了对象元信息，包装了MyBatis中的五个核心的反射类，也是提供给外部使用的反射工具类，可以利用它读取或者修改对象的属性信息（其实也是通过ObjectWrapper调用相关的方法，例如BeanWrapper）。



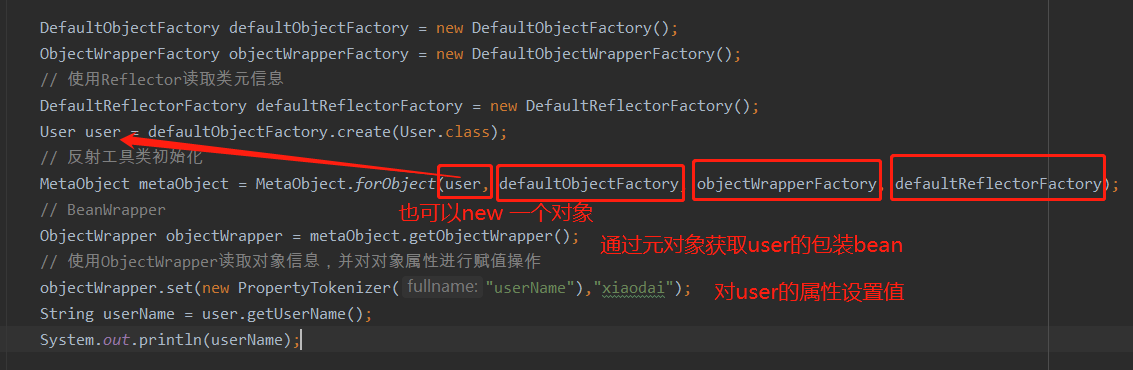
BeanWrapper作用如下:(是对bean的封装可以给bean属性设置值，也可以获取某个属性的值)



MetaClass(封装类元信息,并且也是通过Reflector来获取类的相关属性信息)：



总结：反射模块核心类使用如下



1. 插件模块:

MyBatis 允许使用插件来拦截的方法调用包括：

* Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

-------执行器的拦截-----------------------------------

* ParameterHandler (getParameterObject, setParameters)

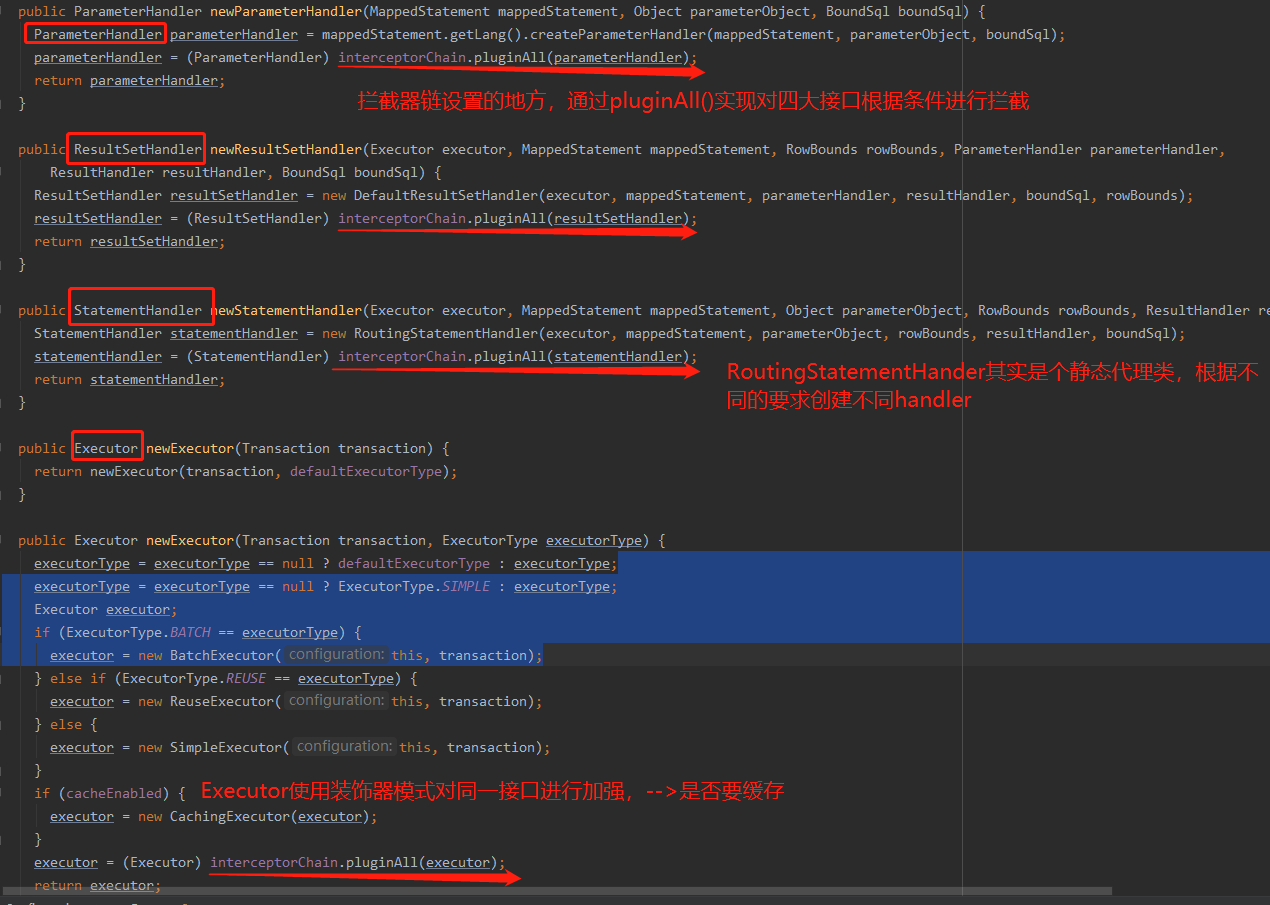
-------参数处理的拦截--------------------------------

* ResultSetHandler (handleResultSets, handleOutputParameters)

--------结果集处理的拦截----------------------------

* StatementHandler (prepare, parameterize, batch, update, query)

--------sql语法构建的拦截--------------------------



这4个方法实例化了对应的对象之后，都会调用interceptorChain的pluginAll方法，InterceptorChain的pluginAll就是遍历所有的拦截器，然后调用各个拦截器的plugin方法。注意：拦截器的plugin方法的返回值会直接被赋值给原先的对象。

在创建SqlSession的时候最先会执行的Configuration的newExecutor()并进行拦截器配置和加强。例：加入PageInterceptor拦截器，newExecutor()时会调用interceptorChain的pluginAll方法，然后在执行PageInterceptor.plugin(),最后会使用Plugin.wrap()为executor生成Plugin代理对象。最后在执行executor.query()时会被Plugin加强，根据PageInterceptor注解@Signature判断是否是要拦截的方法，PageInterceptor拦截主要是分页处理。

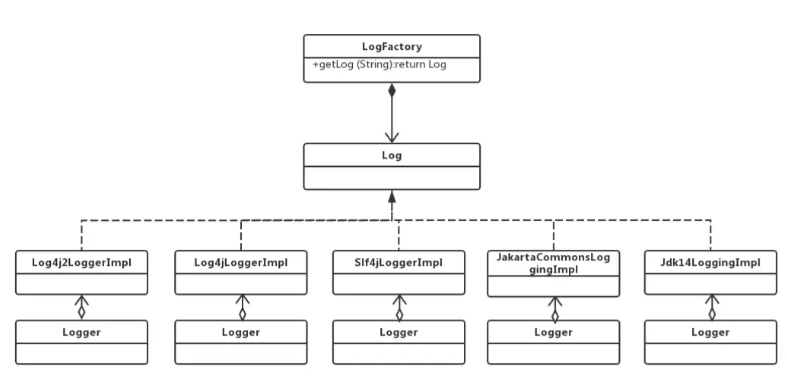
Plugin这个类的作用就是根据 @Interceptors注解，得到这个注解的属性 @Signature数组，然后根据每个 @Signature注解的type，method，args属性使用反射找到对应的Method。最终根据调用的target对象**实现的接口（会和**@Signature的type进行比较**）**决定是否返回一个代理对象替代原先的target对象

总结：拦截的入口其实就是在代理类Plugin的invoke是真实对象之前根据判断@Signature解析的Method和classType是否和代理的对象的类型和方法一致来决定是否执行拦截器。

拦截器链InterceptorChain会对每一个拦截器依次封装在代理对象中（即：realObject --> proxy1RealObject --> prox2Prox1RealObject）只要符合拦截条件都会被依次代理增强。拦截执行的时候会倒序依次执行（prox2Prox1RealObject-->proxy1RealObject -->realObject）。

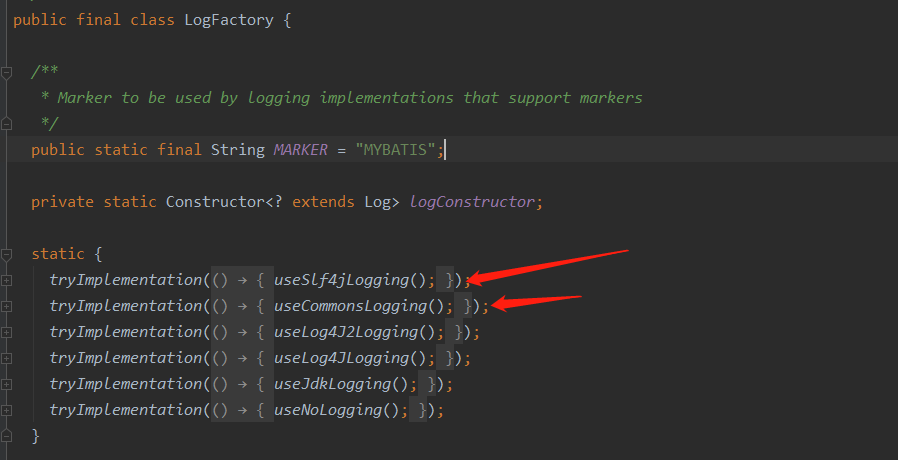
1. Logging模块可以引入第三方例如Slfj----------适配器模式：

mybatis只有属于自己一套的log接口，并没有所有本身的日志,那么如何和市面上面 流行的日志框架做整合呢？这个时候就需要用到适配器模式。



如上图实现mybatis的日志实现类都需要实现这个接口。这使用到LogFactory

==org.apache.ibatis.logging.LogFactory== 日志工厂



如上图 tryImplementation()先判断logConstructor是否为空，只有logConstructor为空的时候tryImplementation里面的方法才会执行。

例如：useSlf4jLogging() ---> 会获取Slf4jImpl的构造器，当使用LogFactory.getLogger()就会实例化，此时logConstructor已经有值，那么其它的try都不会在创建构造器。

1. 缓存模块--------Cache接口（装饰器模式）

实现类-->PerpetualCache：Cache的基本实现类

BlockingCache：它是个阻塞版本的缓存装饰器,保证只有一个线程到数据库中查找 key 对应的数据,使用 ConcurrentHashMap<Object, ReentrantLock> 实现了阻塞功能。

FifoCache：FifoCache 底层维护了一个队列,当需要清除时会清除队列头部的缓存

LruCache：LruCache底层维护了一个 LinkedHashMap 它是一种有序的 HashMap。当我们获取一个对象的时候会把这个对象移动到 LinkedHashMap 的尾部表示最近访问的，需要清除的时候清除头部即可。

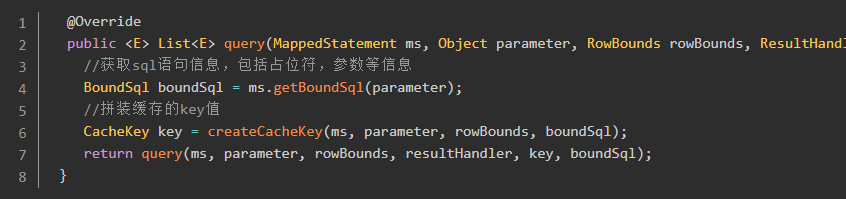
SoftCache&WeakCache：是 JVM 中定义的引用类型,是为了更方便的进行JVM垃圾回收的，针对那些只用一次的对象进行更高效率的回收。

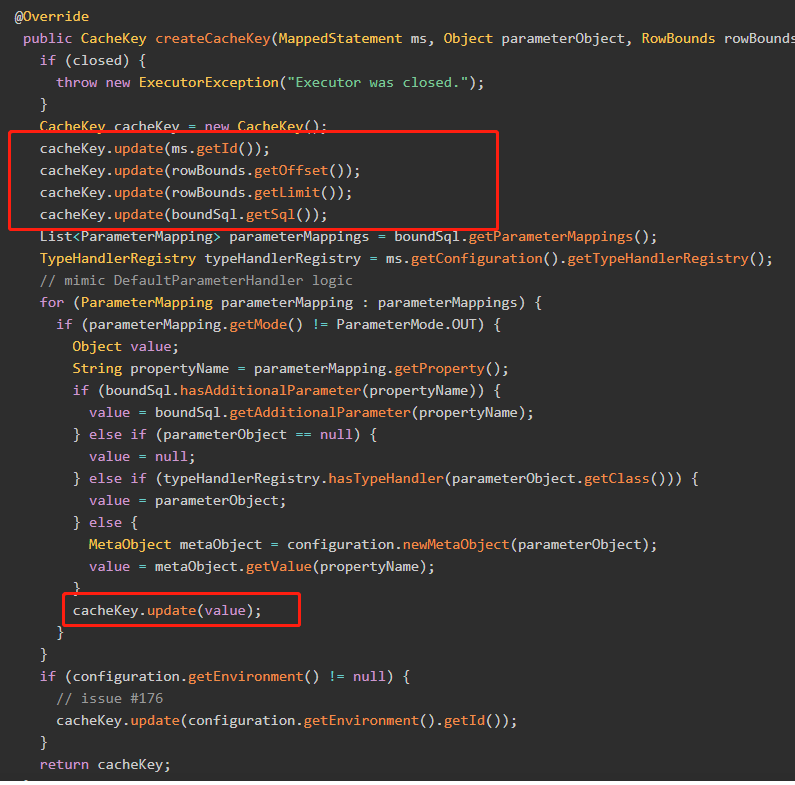
MyBatis中涉及到动态SQL需要通过CacheKey来封装缓存的key值。

构成CacheKey对象的要素（如下图代码）：

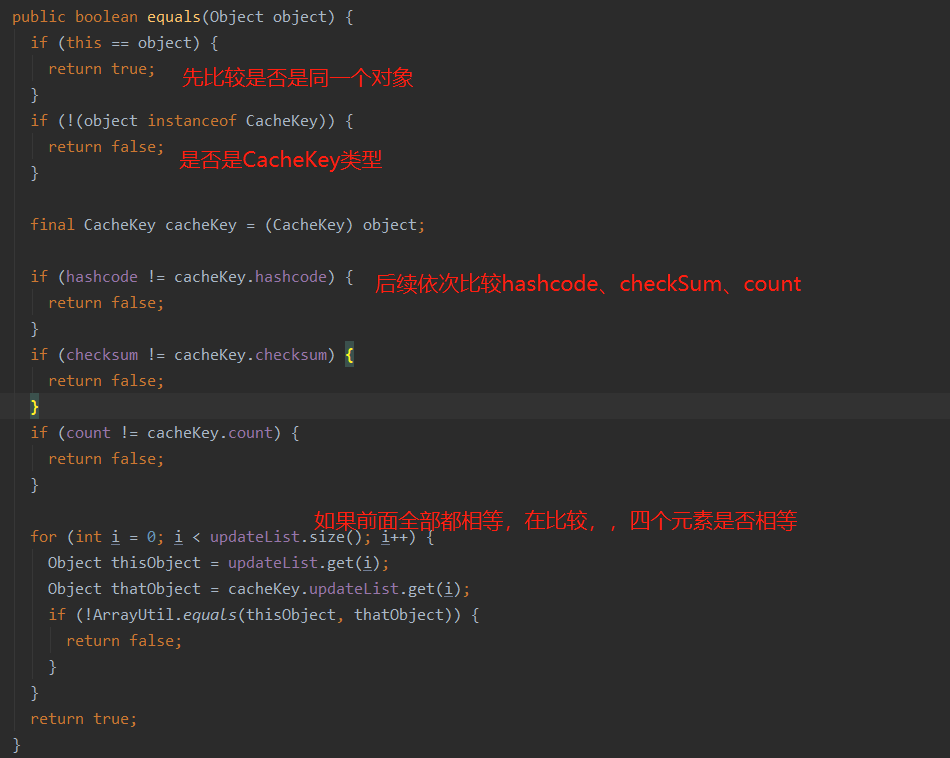
* mappedStatement的id
* 指定查询结果集的范围（分页信息）
* 查询所使用的SQL语句
* 用户传递给SQL语句的实际参数值

BaseExecutor的query：





cacheKey比较是否相等判断逻辑：



总结：MyBatis 的缓存接口 Cache 的基础实现 PerpetualCache 维护了一份 HashMap 作为缓存的实现。同时它还有很多装饰类,比如阻塞式缓存BlockingCache 内部维护了带对象同步器的 ConcurrentHashMap、带清除策略的 FifoCache/LruCache，FifoCache 内部维护了一个队列Dqueue,LruCache内部维护了一份 LinkedHashMap 。这些都使得缓存模块的功能很多,业务方可以根据自己的需求进行灵活组装。

**一级缓存和二级缓存：**

MyBatis 中的缓存就是说 MyBatis 在执行一次SQL查询或者SQL更新之后，这条SQL语句并不会消失，而是被MyBatis 缓存起来，当再次执行相同SQL语句的时候，就会直接从缓存中进行提取，而不是再次执行SQL命令。

一级缓存：

mybatis的一级缓存是基于sqlsession的缓存，默认开启，不同sqlsession之间的缓存数据区域是不会相互影响的，如果会话失效，则缓存失效。

清除缓存：

当commit或者rollback的时候会清除缓存，并且当执行insert、update、delete的时候也会清除缓存。

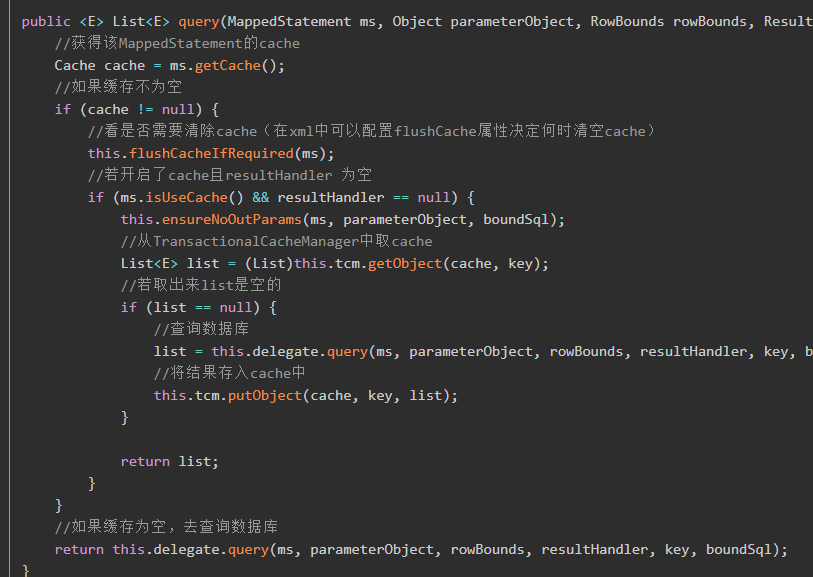
mybatis一级缓存失效的四种情况

* sqlsession变了 缓存失效
* sqlsession不变,查询条件不同，一级缓存失效
* sqlsession不变,中间发生了增删改操作，一级缓存失败
* sqlsession不变,手动清除缓存，一级缓存失败

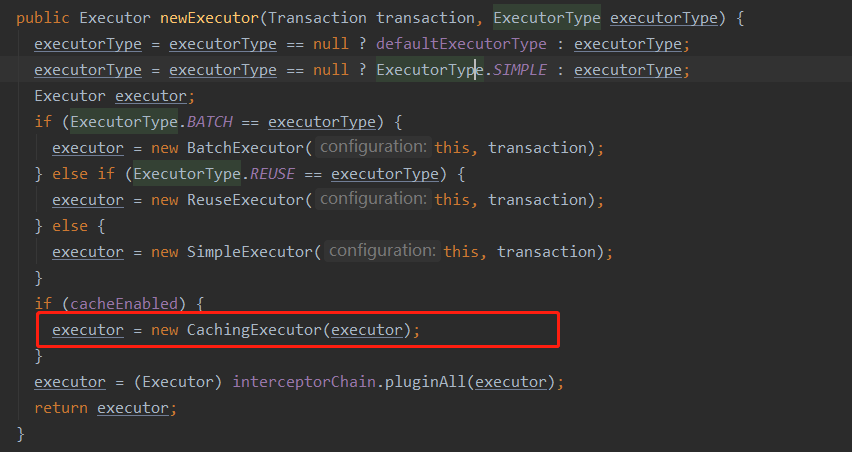
二级缓存：

二级缓存是手动开启的，作用域为mapper级别（也可以说MapperStatement级缓存，也就是一个namespace就会有一个缓存），不同的sqlsession两次执行相同的namespace下的sql，且向sqlq中传递的参数也相同，即最终执行相同的sql，则第一次中会从数据库查并存缓存，第二次会从内存查；因为二级缓存的数据不一定都是存储到内存中，它的存储介质多种多样，实现二级缓存的时候，MyBatis要求返回的POJO必须是可序列化的，也就是要求实现Serializable接口，如果存储在内存中的话，实测不序列化也可以的。

如果开启了二级缓存的话，你的Executor将会被装饰成CachingExecutor，缓存是通过CachingExecutor来操作的，查询出来的结果会存在statement中的cache中，若有更新，删除类的操作默认就会清空该MapperStatement的cache（也可以通过修改xml中的属性，让它不执行），不会影响其他的MapperStatement。



Executor使用的装饰器模式：Configuration创建Executor时如果开启缓存则CachingExecutor会装饰BaseExecutor并会被interceptorChain.pluginAll(executor)进行加强。



**5、建造者模式：**在解析mapper.xml时会通过MapperBuilderAssistant助理进行每一阶段的属性值的builder（例如熟悉addResultMap、addMappedStatement）。

