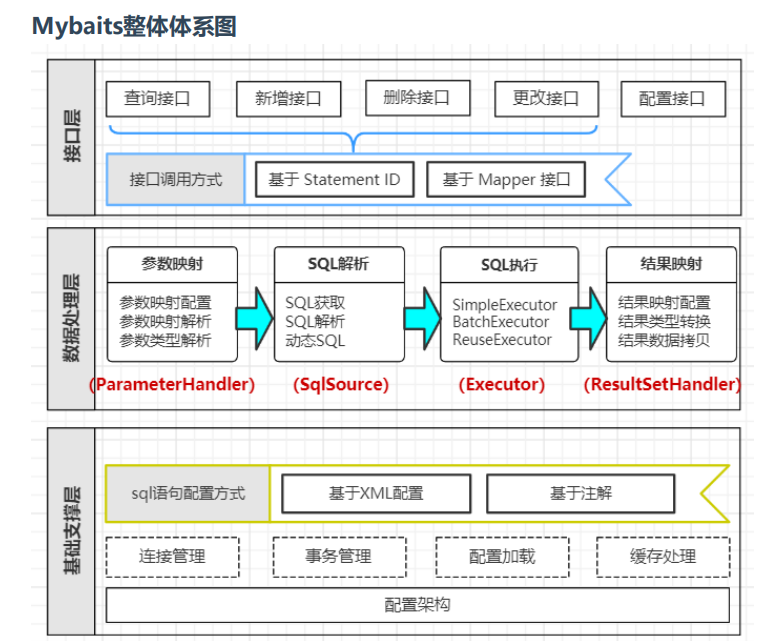
Mybatis在同一事物中多次使用同一SQL查询数据，返回的对象为一个（中间无修改删除新增操作），第一次查询后对象以放入mybatis一级缓存

Mybatis在项目初始化时会将mapper类的类定义注册完成，实例的初始化在service类的autowired注解装配时调用factoryBean的getObject方法时创建动态代理对象

疑问？项目初始化时autowired注解会进行装配吗，还是在类初始化时已经装配过了，验证方式在项目启动后调用mapper方法，断点打在MapperFactoryBean的getObject方法上看会不会执行

# mybatis简介

MyBatis是一个持久层（ORM）框架，使用简单，学习成本较低。可以执行自己手写的SQL语句，比较灵活。但是MyBatis的自动化程度不高，移植性也不高，有时从一个数据库迁移到另外一个数据库的时候需要自己修改配置，所以称只为半自动ORM框架



## mybatis与jdbd的区别

**传统JDBC的问题如下：**

1.数据库连接创建，释放频繁造成系统部资源的浪费，从而影响系统性能，使用数据库连接池可以解决问题。

2.sql语句在代码中硬编码，造成代码的不易维护，实际应用中sql的变化可能较大，sql代码和java代码没有分离开来维护不方便。

3.使用preparedStatement向有占位符传递参数存在硬编码问题因为sql中的where子句的条件不确定，同样是修改不方便

4.对结果集中解析存在硬编码问题，sql的变化导致解析代码的变化，系统维护不方便。

**mybatis对传统的JDBC的解决方案**

1、数据库连接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据连接池，使用连接池管理数据库链接。

2、Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

3、向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis自动将java对象映射至sql语句，通过statement中的parameterType定义输入参数的类型。

4、对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo 对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象，通过statement中的resultType定义输出结果的类型。

## mybatis的配置及启动





总结下就是分为下面四个步骤：

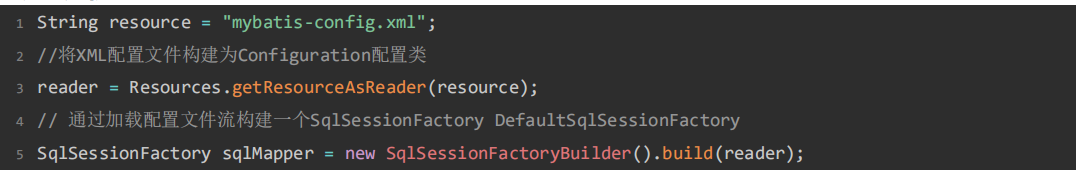
从配置文件（通常是XML文件）得到SessionFactory;

从SessionFactory得到SqlSession；

通过SqlSession进行CRUD和事务的操作；

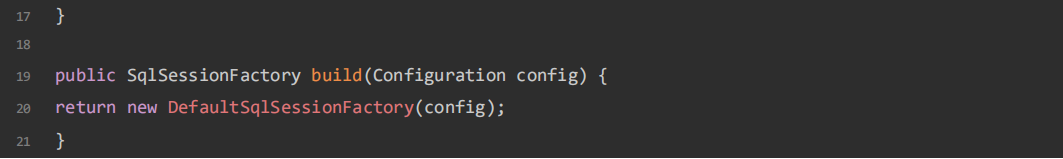
执行完相关操作之后关闭Session。

启动流程分析

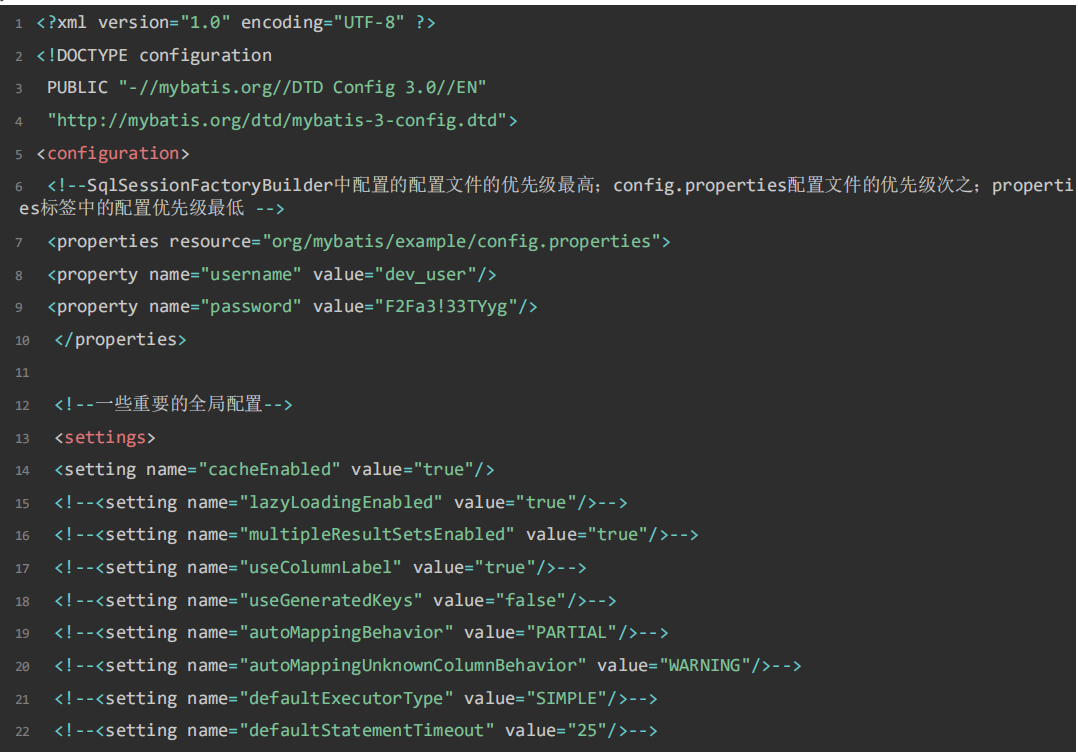


通过上面代码发现，创建SqlSessionFactory的代码在SqlSessionFactoryBuilder中，进去一探究竟：

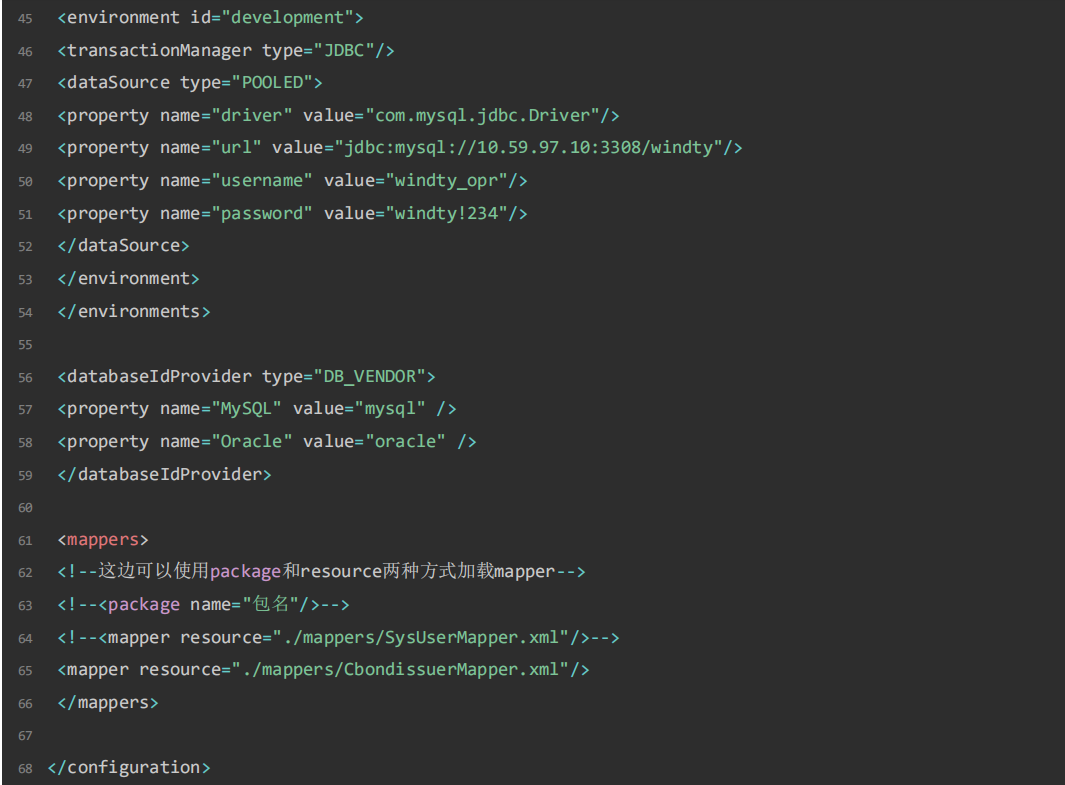




Xml配置文件解析



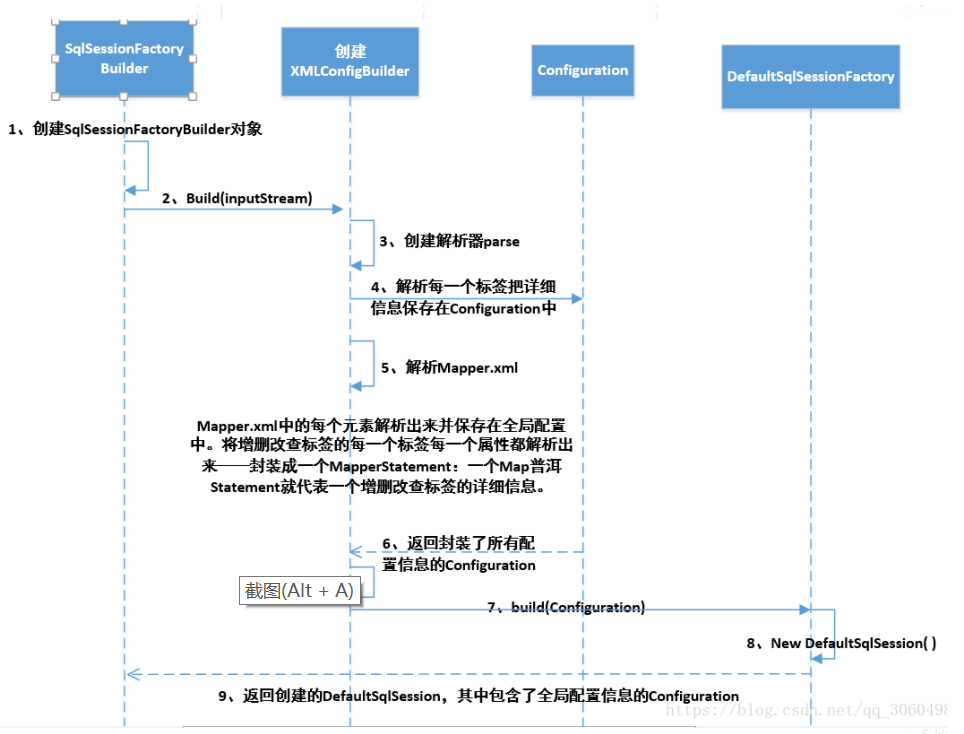




解析配置类

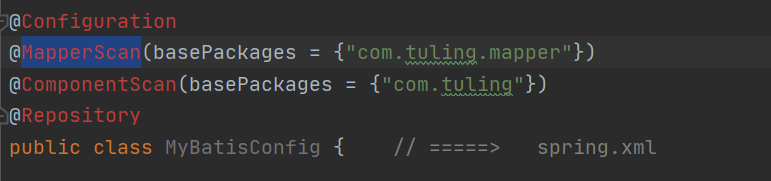






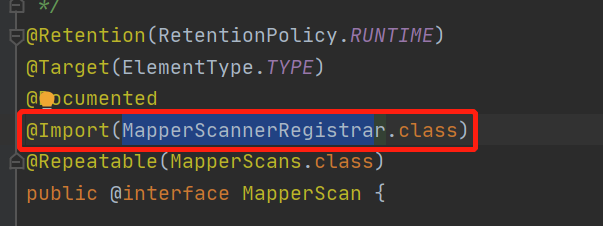
### 生产中mybatis的启动流程

1. 配置类或者配置文件引入mapperScan，解析mapper文件



通过mapperScan引入一个bean定义

这个类是我们spring整合MyBatis的时候 在我们的主配置类上添加了@MapperScann注解 我们研究这个注解的时候，@Import(MapperScannerRegistrar.class) public @interface MapperScan 发现是一个组合注解@Import导入MapperScannerRegistrar 然后我们分析MapperScannerRegistrar组件是实现了ImportBeanDefinitionRegistrar 集合Springbean定义的扫描源码，我们知道ImportBeanDefinitionRegistrar 在bean定义扫描的时会调用registerBeanDefinitions()方法往我们的容器中添加bean定义对象beanDefinitionMap中，该类注册了一个MapperScannerConfigurer类，该类实现了BeanDefinitionRegistryPostProcessor一个bean工厂后置处理器，通过调用后置处理器创建ClassPathMapperScanner对象



Mybatis通过ClassPathMapperScanner类重写isCandidateComponent方法将接口类扫描并且生成bean定义，并且在该类中进行偷天换日将bean定义的class属性替换为

MapperFactoryBean，该类实现FactoryBean，获取对象是通过getObject方法生成动态代理对象

# mybatis深度

## 2.1 mybatis启动初始化

### 1.1 解析配置文件生成SqlSessionFactory

配置文件中创建 SqlSessionFactory时调用afterPropertiesSet()方法中buildSqlSessionFactory()方法开始创建SqlSessionFactory，使用建造者模式解析配置文件 ，sql文件解析为一个个sqlNode，例如ifNode，whereNode当具体调用mapper方法时在拼接出sql语句

### 2.2 openSession过程

**sqlSession为会话级别，每次调用提交后就会关闭，如果使用事务则不会，而是记录引用次数，当事务提交后则关闭session**

在此过程主要是创建执行器，mybatis执行器一共有simpleExecutor、resueExceutor、batchExecutor、cachingExecutor、baseExceutor，当开启二级缓存时，会将执行器装饰为cachingExecutor，simpleExecutor为默认执行器，resueExceutor执行器为可重复使用执行器，batchExecutor批量执行器，baseExceutor执行器实现一级缓存，当二级缓存未启用或者二级缓存未命中，则调用到baseExceutor执行器查找一级缓存，如果没有命中，则使用三种基础的执行器执行具体的查库操作，创建执行器成功后将executor插件添加到拦截器链中

**SimpleExecutor**：每执行一次update或select，就开启一个Statement对象，用完

立刻关闭Statement对象。

**ReuseExecutor**[ˈriːˈjuːz]**：**执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String,

Statement>内，供下一次使用。简言之，就是重复使用Statement对象,Statement对象已经封装好了参数,所以做为key的sql也是带有参数的,会导致Statement的复用率并不高。

**BatchExecutor**：执行update（没有select，JDBC批处理不支持select），将所有

sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多

个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行

executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

CacheExecutor其实是封装了普通的Executor，和普通的区别是在查询前先会查询缓存中

是否存在结果，如果存在就使用缓存中的结果，如果不存在还是使用普通的Executor进行

查询，再将查询出来的结果存入缓存。

## 2.2 mybatis重要类

MapperRegistry：本质上是一个Map，其中的key是Mapper接口的全限定名，

value的MapperProxyFactory；

MapperProxyFactory：这个类是MapperRegistry中存的value值，在通过

sqlSession获取Mapper时，其实先获取到的是这个工厂，然后通过这个工厂创建

Mapper的动态代理类；

MapperProxy：实现了InvocationHandler接口，Mapper的动态代理接口方法的调

用都会到达这个类的invoke方法；

MapperMethod：判断你当前执行的方式是增删改查哪一种，并通过SqlSession执

行相应的操作；

SqlSession：作为MyBatis工作的主要顶层API，表示和数据库交互的会话，完成必

要数据库增删改查功能；

Executor：MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓

存的维护；

StatementHandler:封装了JDBC Statement操作，负责对JDBC statement 的操作，如设

置参数、将Statement结果集转换成List集合。

ParameterHandler:负责对用户传递的参数转换成JDBC Statement 所需要的参数，

ResultSetHandler:负责将JDBC返回的ResultSet结果集对象转换成List类型的集合；

TypeHandler:负责java数据类型和jdbc数据类型之间的映射和转换

MappedStatement:MappedStatement维护了一条<select|update|delete|insert>节点

的封装，

SqlSource:负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到

BoundSql对象中，并返回

BoundSql:表示动态生成的SQL语句以及相应的参数信息

Configuration:MyBatis所有的配置信息都维持在Configuration对象之中。

调试主要关注点

MapperProxy.invoke方法：MyBatis的所有Mapper对象都是通过动态代理生成

的，任何方法的调用都会调到invoke方法，这个方法的主要功能就是创建

MapperMethod对象，并放进缓存。所以调试时我们可以在这个位置打个断点，看下是

否成功拿到了MapperMethod对象，并执行了execute方法。

MapperMethod.execute方法：这个方法会判断你当前执行的方式是增删改查哪一

种，并通过SqlSession执行相应的操作。Debug时也建议在此打个断点看下。

DefaultSqlSession.selectList方法：这个方法获取了获取了MappedStatement对

象，并最终调用了Executor的query方法；

## 2.3 插件

MyBatis 允许你在已映射语句执行过程中的某一点进行拦截调用。默认情况下，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)

ParameterHandler (getParameterObject, setParameters)

ResultSetHandler (handleResultSets, handleOutputParameters)

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

我们看到了可以拦截Executor接口的部分方法，比如update，query，commit，rollback等方法，还有其他接口的一些方法等。

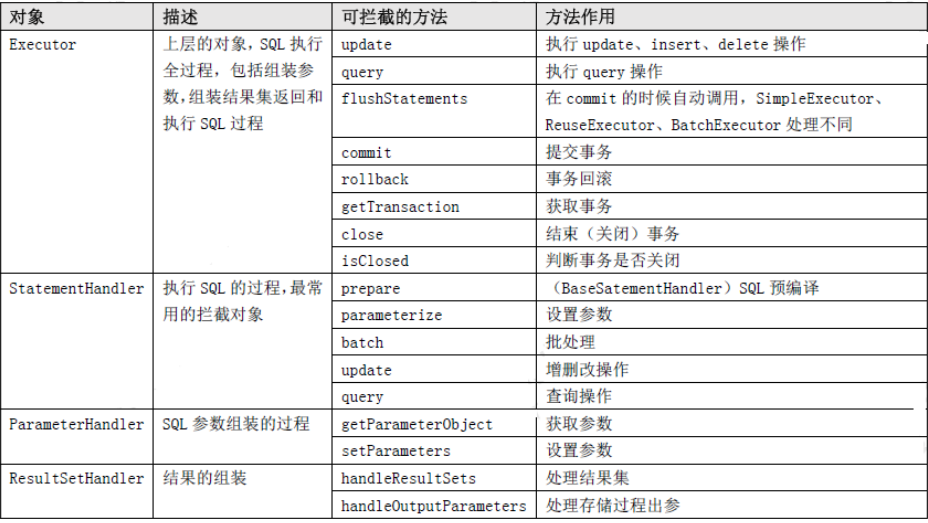
总体概括为：

拦截执行器的方法

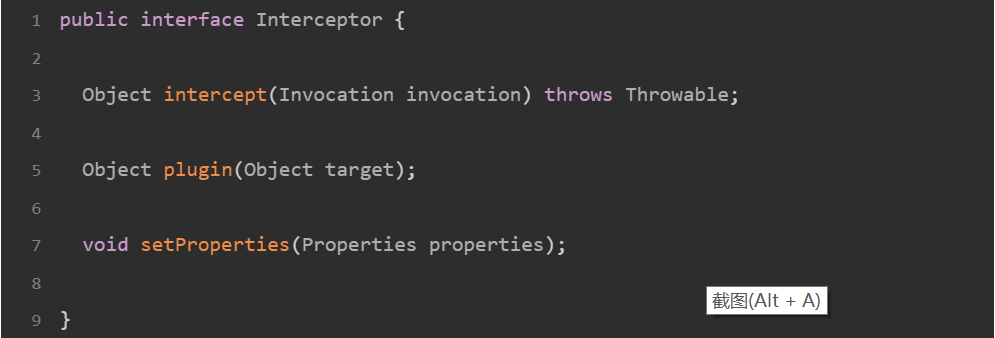
拦截参数的处理

拦截结果集的处理

拦截Sql语法构建的处理



Mybatis拦截器



Intercept方法：

这个方法是拦截使用的，invocation可以获取mybatis中的sql语句。

判断是否通过拦截器：return invocation.proceed();

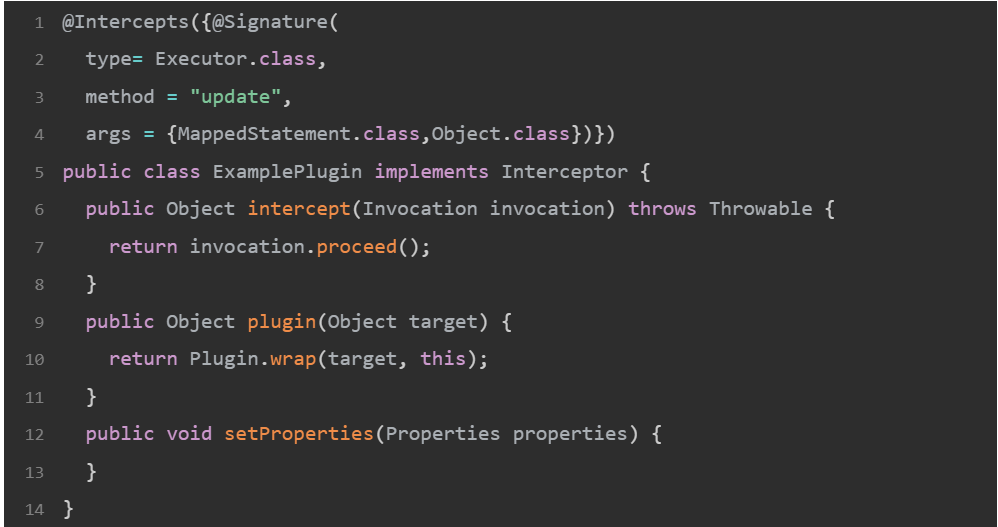
这里几乎可以获取所有向数据库提交的内容。

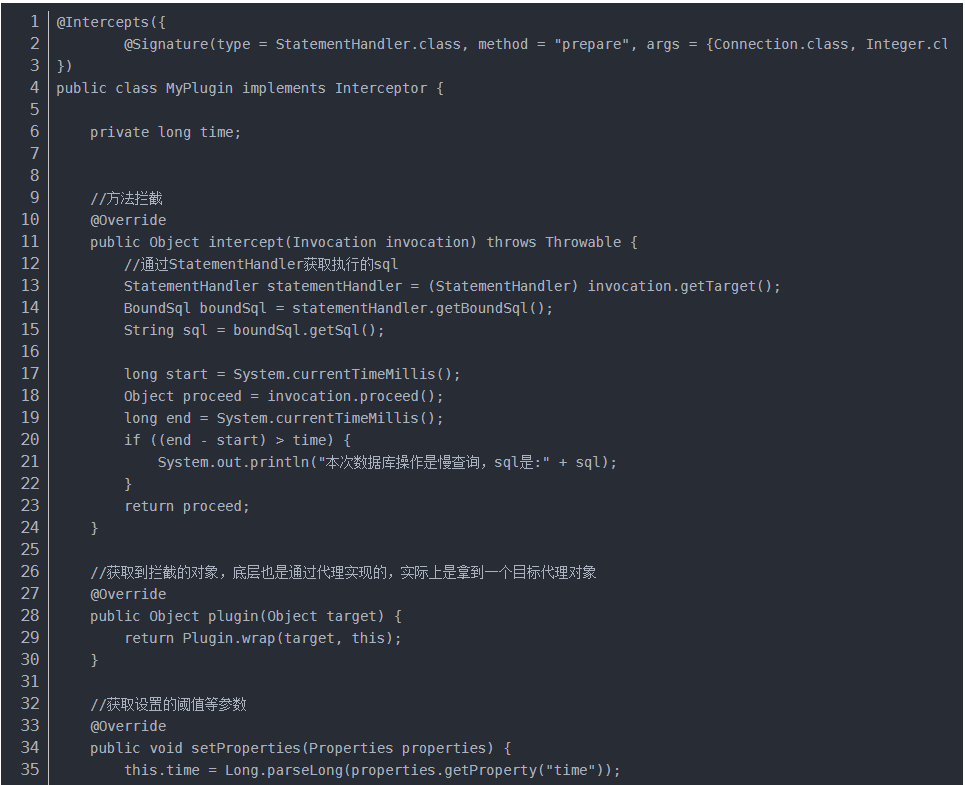
Plugin方法：

setProperties方法：

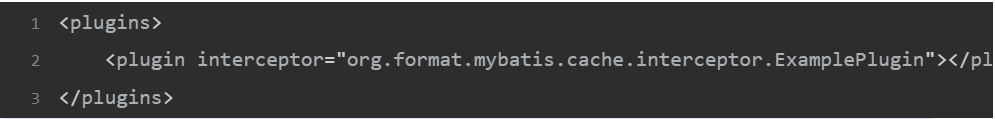
获取mybatis的xml中配置plugin插件配置的数据。

插件示例



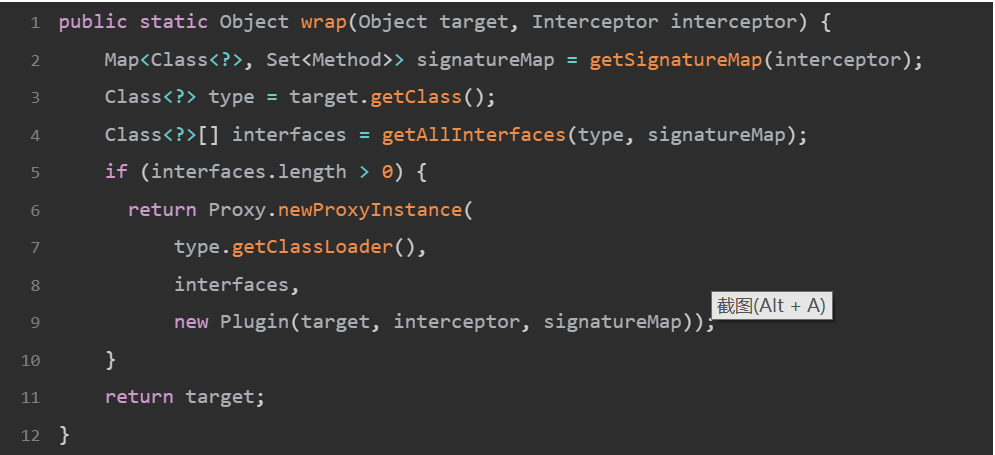


插件配置



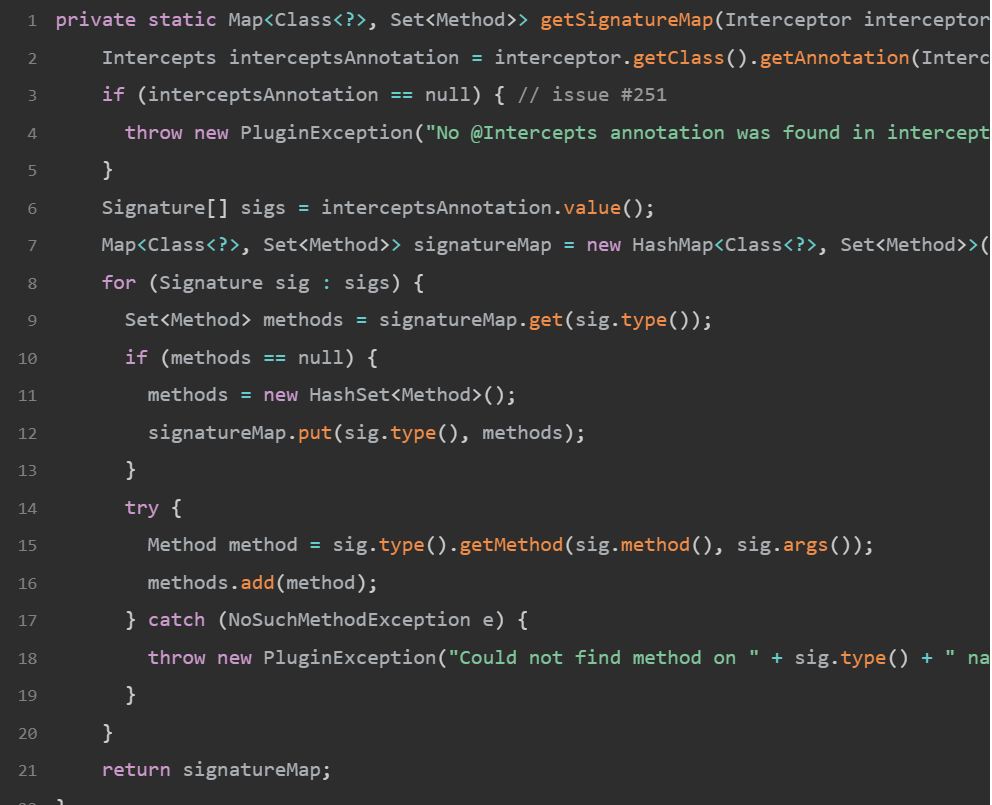
这个拦截器拦截Executor接口的update方法（其实也就是SqlSession的新增，删除，修改操作），所有执行executor的update方法都会被该拦截器拦截到。

Plugin的wrap方法

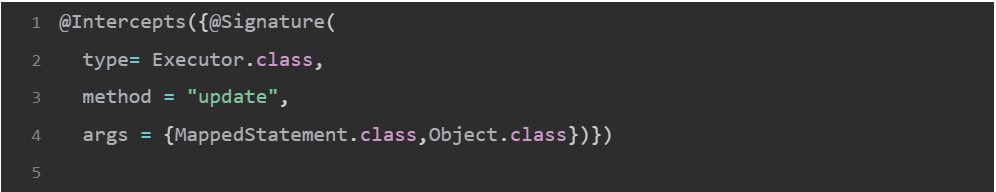


Plugin类实现了InvocationHandler接口，很明显，我们看到这里返回了一个JDK自身提供的动态代理类。我们解剖一下这个方法调用的其他方法：

getSignatureMap方法：



getSignatureMap方法解释：首先会拿到拦截器这个类的 @Interceptors注解，然后拿到这个注解的属性 @Signature注解集合，然后遍历这个集合，遍历的时候拿出 @Signature注解的type属性(Class类型)，然后根据这个type得到带有method属性和args属性的Method。由于 @Interceptors注解的 @Signature属性是一个属性，所以最终会返回一个以type为key，value为Set的Map。



比如这个 @Interceptors注解会返回一个key为Executor，value为集合(这个集合只有一个元素，也就是Method实例，这个Method实例就是Executor接口的update方法，且这个方法带有MappedStatement和Object类型的参数)。这个Method实例是根据 @Signature的method和args属性得到的。如果args参数跟type类型的method方法对应不上，那么将会抛出异常。

getAllInterfaces方法：

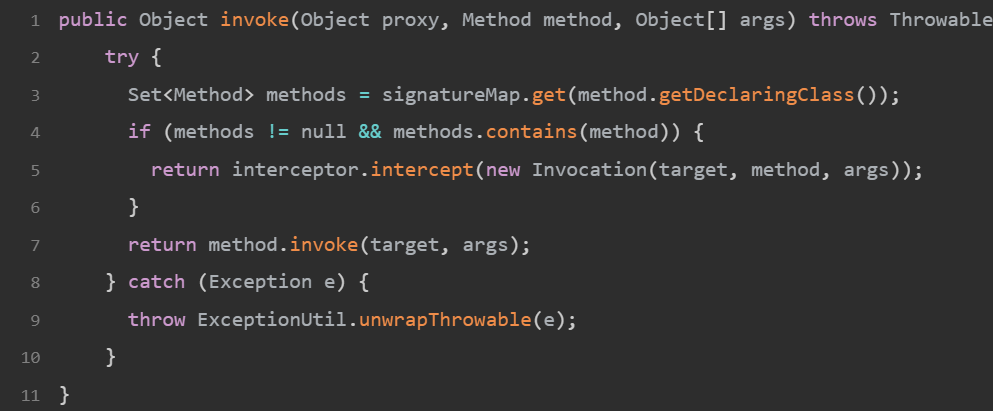


getAllInterfaces方法解释：根据目标实例target(这个target就是之前所说的MyBatis拦截器可以拦截的类，Executor,ParameterHandler,ResultSetHandler,StatementHandler)和它的父类们，返回signatureMap中含有target实现的接口数组。

所以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对象实现的接口决定是否返回一个代理对象替代原先的target对象。

比如MyBatis官网的例子，当Configuration调用newExecutor方法的时候，由于Executor接口的update(MappedStatement ms, Object parameter)方法被拦截器被截获。因此最终返回的是一个代理类Plugin，而不是Executor。这样调用方法的时候，如果是个代理类，那么会执行：

没错，如果找到对应的方法被代理之后，那么会执行Interceptor接口的interceptor方法



MyBatis拦截器接口提供的3个方法中，plugin方法用于某些处理器(Handler)的构建过程。interceptor方法用于处理代理类的执行。setProperties方法用于拦截器属性的设置。

其实MyBatis官网提供的使用 @Interceptors和 @Signature注解以及Plugin类这样处理拦截器的方法，我们不一定要直接这样使用。我们也可以抛弃这3个类，直接在plugin方法内部根据target实例的类型做相应的操作。

总体来说MyBatis拦截器还是很简单的，拦截器本身不需要太多的知识点，但是学习拦截器需要对MyBatis中的各个接口很熟悉，因为拦截器涉及到了各个接口的知识点。

## 2.4 mybatis的一二级缓存

Mybatis默认开启一级缓存，即会话缓存，存于本地，二级缓存默认关闭，需要配置开启

1. 一级缓存

一级缓存只在同一个sqlSession下有效，每次查询都会新建一个session，如果有事务存在，则同一事物内使用同一个session



1. **二级缓存**

二级缓存是全局作用域缓存，默认是不开启的，需要手动进行配置。

Mybatis提供二级缓存的接口以及实现，缓存实现的时候要求实体类实现Serializable接口，二级缓存在sqlSession关闭或提交之后才会生效。

**二级缓存的属性**：

eviction:表示缓存回收策略，默认是LRU

LRU：最近最少使用的，移除最长时间不被使用的对象

FIFO：先进先出，按照对象进入缓存的顺序来移除

SOFT：软引用，移除基于垃圾回收器状态和软引用规则的对象

WEAK：弱引用，更积极地移除基于垃圾收集器状态和弱引用规则的对象

flushInternal:刷新间隔，单位毫秒

默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句时刷新

size：引用数目，正整数

代表缓存最多可以存储多少个对象，太大容易导致内存溢出

readonly：只读，true/false

true：只读缓存，会给所有调用这返回缓存对象的相同实例，因此这些对象不能被修改。

false：读写缓存，会返回缓存对象的拷贝（序列化实现），这种方式比较安全，默认值

**二级缓存的使用范围**：

如果设置了全局的二级缓存配置，那么在使用的时候需要注意，在每一个单独的select语句中，可以设置将查询缓存关闭，以完成特殊的设置

**二级缓存全局设置开启时则可以单独设置sql语句关闭二级缓存，如果全局的二级缓存设置为无效，则不能单独设置二级缓存为开启，因为全局开启二级缓存才会使用cachingExecutor，**

**而在此执行器内会判断单独的sql语句是否开启了二级缓存**

1、在setting中设置，是配置二级缓存开启，一级缓存默认一直开启

<setting name="cacheEnabled" value="true"/>

2、select标签的useCache属性：

在每一个select的查询中可以设置当前查询是否要使用二级缓存，只对二级缓存有效

3、sql标签的flushCache属性

增删改操作默认值为true，sql执行之后会清空一级缓存和二级缓存，而查询操作默认是false

4、sqlSession.clearCache()

只是用来清楚一级缓存

**二级缓存特性**：

1. 默认关闭
2. 作用域：基于全局范围，应用级别。
3. 缓存默认实现类PerpetualCache ,使用map进行存储的但是二级缓存根据不同的mapper命名空间多包了一层map \* : org.apache.ibatis.session.Configuration#caches key:mapper命名空间 value:erpetualCache.map \* \* key==> sqlid+sql
4. 事务提交的时候（sqlSession关闭)
5. 先从二级缓存中获取，再从一级缓存中获取

\* **实现**：

1. 开启二级缓存<setting name="cacheEnabled" value="true"/>
2. 在需要使用到二级缓存的映射文件中加入<cache></cache>,基于Mapper映射文件来实现缓存的，基于Mapper映射文件的命名空间来存储的
3. 在需要使用到二级缓存的javaBean中实现序列化接口implements Serializable \* 配置成功就会出现缓存命中率 同一个sqlId: 从缓存中拿出的次数/查询总次数

**失效**：

1. 同一个命名空间进行了增删改的操作，会导致二级缓存失效 \* 但是如果不想失效：可以将SQL的flushCache 设置为false,但是要慎重设置，因为会造成数据脏读问题，除非你能保证查询的数据永远不会执行增删改
2. 让查询不缓存数据到二级缓存中useCache="false"
3. 如果希望其他mapper映射文件的命名空间执行了增删改清空另外的命名空间就可以设置： \* <cache-ref namespace="cn.tulingxueyuan.mapper.DeptMapper"/>

### 2.4.1 mybatis整合第三方缓存

1、整合redis

添加mybatis-redis适配器依赖



设置mybatis二级缓存实现类



2、整合ehcache

导入ehcache依赖



ehcache配置





属性说明：

l diskStore：指定数据在磁盘中的存储位置。

l defaultCache：当借助CacheManager.add("demoCache")创建Cache时，EhCache便会采用<defalutCache/>指定的的管理策略

以下属性是必须的：

l maxElementsInMemory - 在内存中缓存的element的最大数目

l maxElementsOnDisk - 在磁盘上缓存的element的最大数目，若是0表示无穷大

l eternal - 设定缓存的elements是否永远不过期。如果为true，则缓存的数据始终有效，如果为false那么还要根据timeToIdleSeconds，timeToLiveSeconds判断 l overflowToDisk - 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上

以下属性是可选的：

l timeToIdleSeconds - 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时，这些数据便会删除，默认值是0,也就是可闲置时间无穷大

l timeToLiveSeconds - 缓存element的有效生命期，默认是0.,也就是element存活时间无穷大 diskSpoolBufferSizeMB 这个参数设置DiskStore(磁盘缓存)的缓存区大小.默认是30MB.每个Cache都应该有自己的一个缓冲区.

l diskPersistent - 在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。

l diskExpiryThreadIntervalSeconds - 磁盘缓存的清理线程运行间隔，默认是120秒。每个120s，相应的线程会进行一次EhCache中数据的清理工作

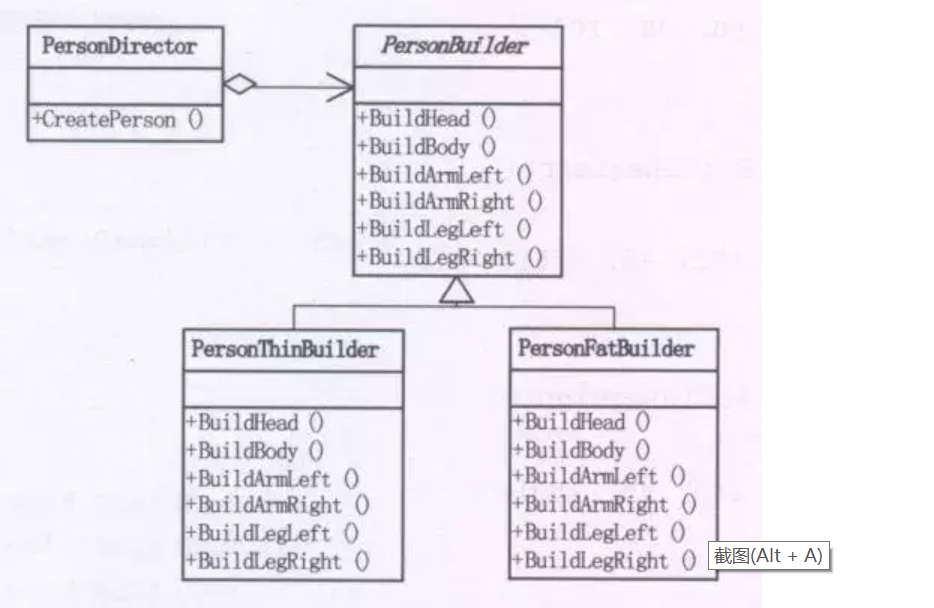
l memoryStoreEvictionPolicy - 当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。默认是LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出）

## 2.5 mybatis所用设计模式

### 1、builder模式

Builder模式，例如SqlSessionFactoryBuilder、XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder、CacheBuilder；

Builder模式的定义是“将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。”，它属于创建类模式，一般来说，如果一个对象的构建比较复杂，超出了构造函数所能包含的范围，就可以使用工厂模式和Builder模式，相对于工厂模式会产出一个完整的产品，Builder应用于更加复杂的对象的构建，甚至只会构建产品的一个部分。

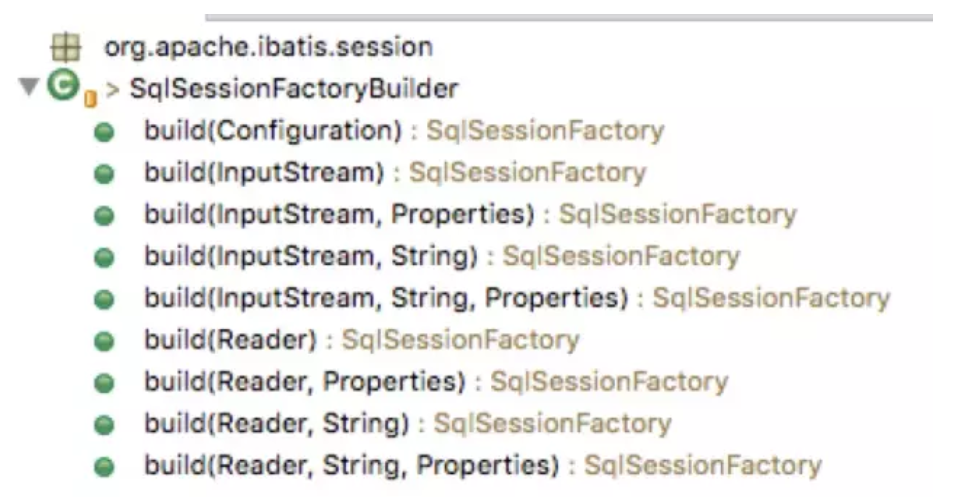


在Mybatis环境的初始化过程中，SqlSessionFactoryBuilder会调用XMLConfigBuilder读取所有的MybatisMapConfig.xml和所有的\*Mapper.xml文件，构建Mybatis运行的核心对象Configuration对象，然后将该Configuration对象作为参数构建一个SqlSessionFactory对象。

其中XMLConfigBuilder在构建Configuration对象时，也会调用XMLMapperBuilder用于读取\*Mapper文件，而XMLMapperBuilder会使用XMLStatementBuilder来读取和build所有的SQL语句。

在这个过程中，有一个相似的特点，就是这些Builder会读取文件或者配置，然后做大量的XpathParser解析、配置或语法的解析、反射生成对象、存入结果缓存等步骤，这么多的工作都不是一个构造函数所能包括的，因此大量采用了Builder模式来解决。

对于builder的具体类，方法都大都用build\*开头，比如SqlSessionFactoryBuilder为例，它包含以下方法：

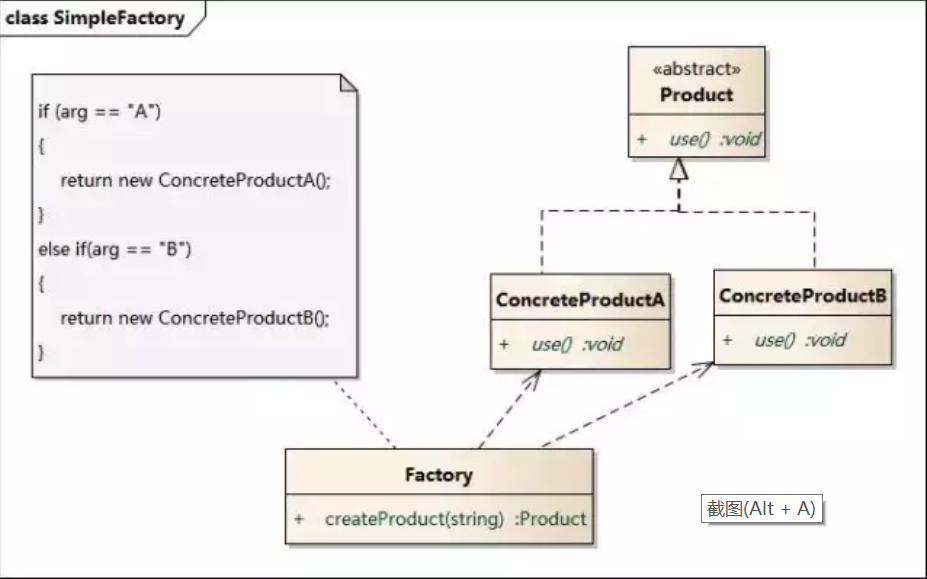


<https://mp.weixin.qq.com/s/127lzD8b1qVQ7lVDYHmd0w> 建造者模式戏说

### 2、工厂模式

在Mybatis中比如SqlSessionFactory使用的是工厂模式，该工厂没有那么复杂的逻辑，是一个简单工厂模式。

简单工厂模式(Simple Factory Pattern)：又称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同类的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

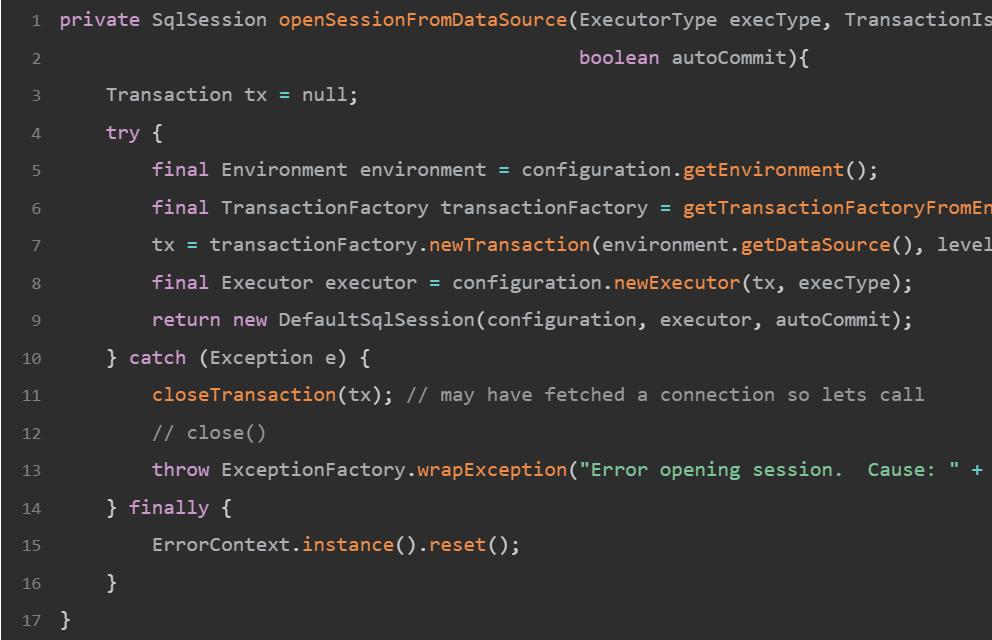


SqlSession可以认为是一个Mybatis工作的核心的接口，通过这个接口可以执行执行SQL语句、获取Mappers、管理事务。类似于连接MySQL的Connection对象。



可以看到，该Factory的openSession方法重载了很多个，分别支持autoCommit、Executor、Transaction等参数的输入，来构建核心的SqlSession对象。

在DefaultSqlSessionFactory的默认工厂实现里，有一个方法可以看出工厂怎么产出一个产品：



这是一个openSession调用的底层方法，该方法先从configuration读取对应的环境配置，然后初始化TransactionFactory获得一个Transaction对象，然后通过Transaction获取一个Executor对象，最后通过configuration、Executor、是否autoCommit三个参数构建了SqlSession。

在这里其实也可以看到端倪，SqlSession的执行，其实是委托给对应的Executor来进行的。

而对于LogFactory，它的实现代码：

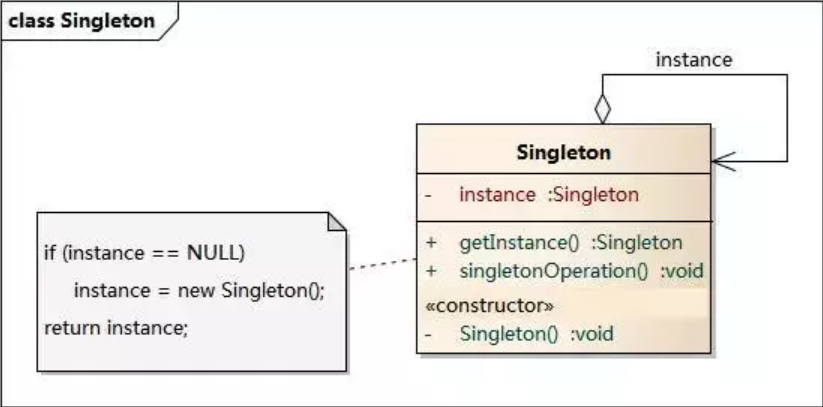


这里有个特别的地方，是Log变量的的类型是Constructorextends Log>，也就是说该工厂生产的不只是一个产品，而是具有Log公共接口的一系列产品，比如Log4jImpl、Slf4jImpl等很多具体的Log。

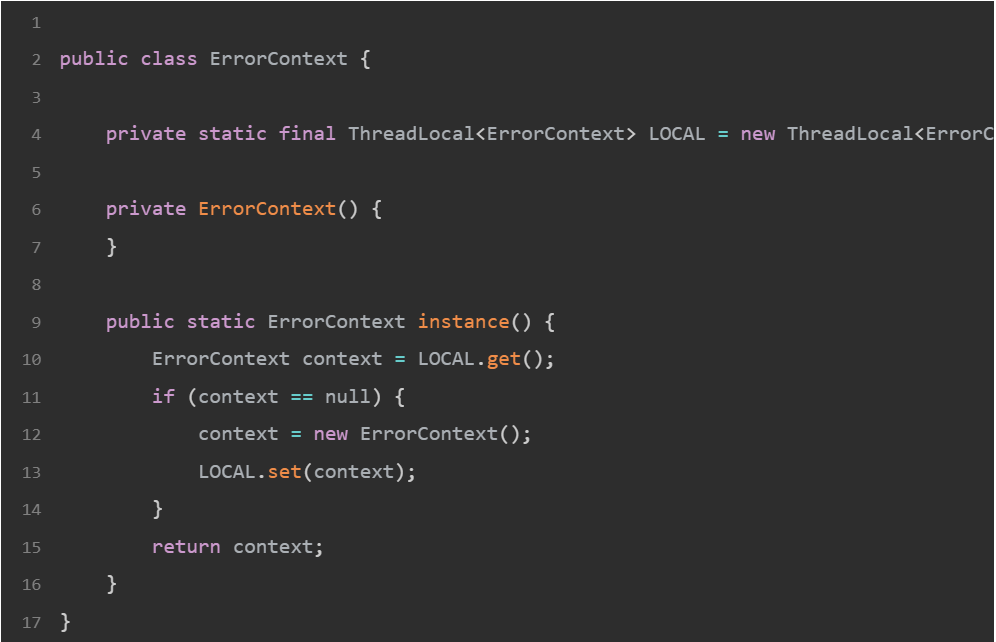
### 3、单例模式

单例模式(Singleton Pattern)：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

单例模式的要点有三个：一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。



在Mybatis中有两个地方用到单例模式，ErrorContext和LogFactory，其中ErrorContext是用在每个线程范围内的单例，用于记录该线程的执行环境错误信息，而LogFactory则是提供给整个Mybatis使用的日志工厂，用于获得针对项目配置好的日志对象。



构造函数是private修饰，具有一个static的局部instance变量和一个获取instance变量的方法，在获取实例的方法中，先判断是否为空如果是的话就先创建，然后返回构造好的对象。

只是这里有个有趣的地方是，LOCAL的静态实例变量使用了ThreadLocal修饰，也就是说它属于每个线程各自的数据，而在instance()方法中，先获取本线程的该实例，如果没有就创建该线程独有的ErrorContext。

<https://mp.weixin.qq.com/s/dDAAH-Vkxi8xoj6uT0XRtg> 戏说单例

### 4、代理模式

戏说代理模式 <https://mp.weixin.qq.com/s/3wdp73HlmVb-XsmHWYHTfw>

代理模式可以认为是Mybatis的核心使用的模式，正是由于这个模式，我们只需要编写Mapper.java接口，不需要实现，由Mybatis后台帮我们完成具体SQL的执行。

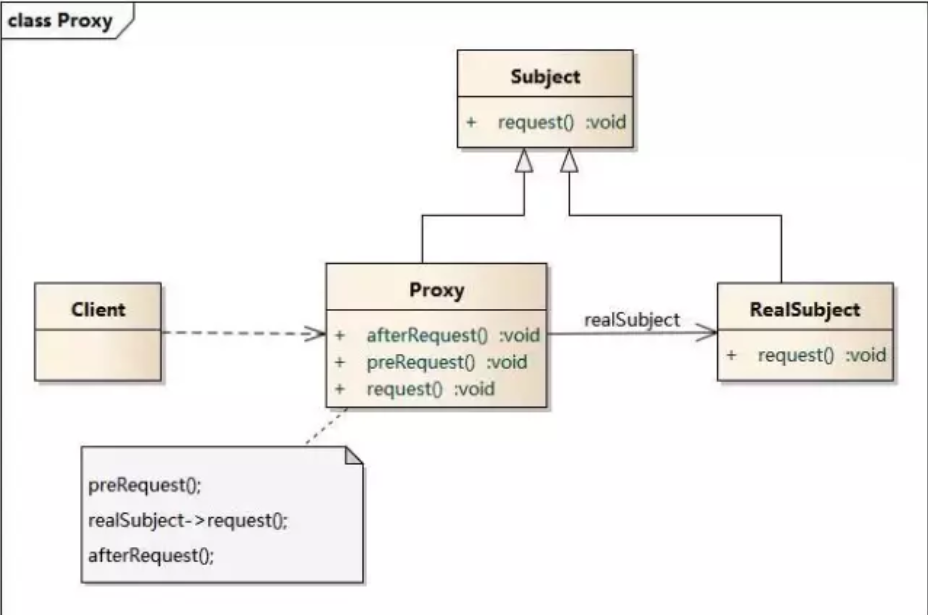
代理模式(Proxy Pattern) ：给某一个对象提供一个代 理，并由代理对象控制对原对象的引用。代理模式的英 文叫做Proxy或Surrogate，它是一种对象结构型模式。

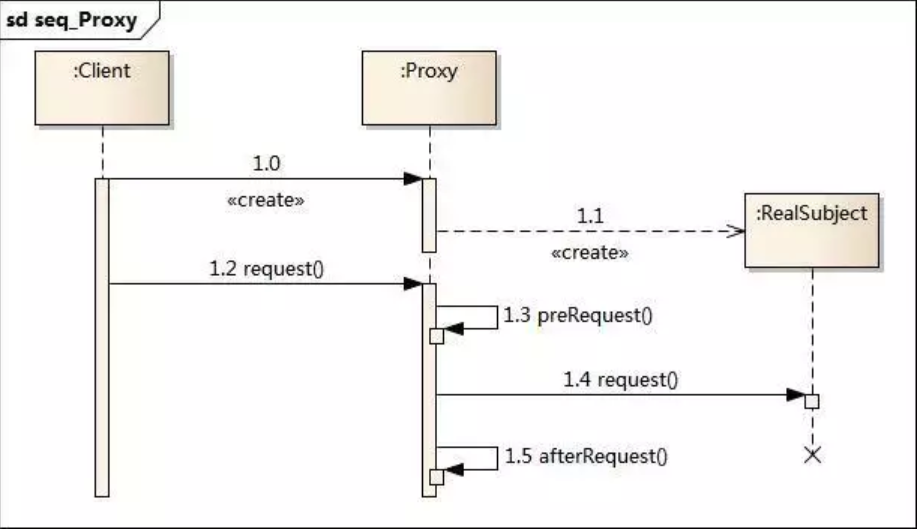
代理模式包含如下角色：

Subject: 抽象主题角色

Proxy: 代理主题角色

RealSubject: 真实主题角色



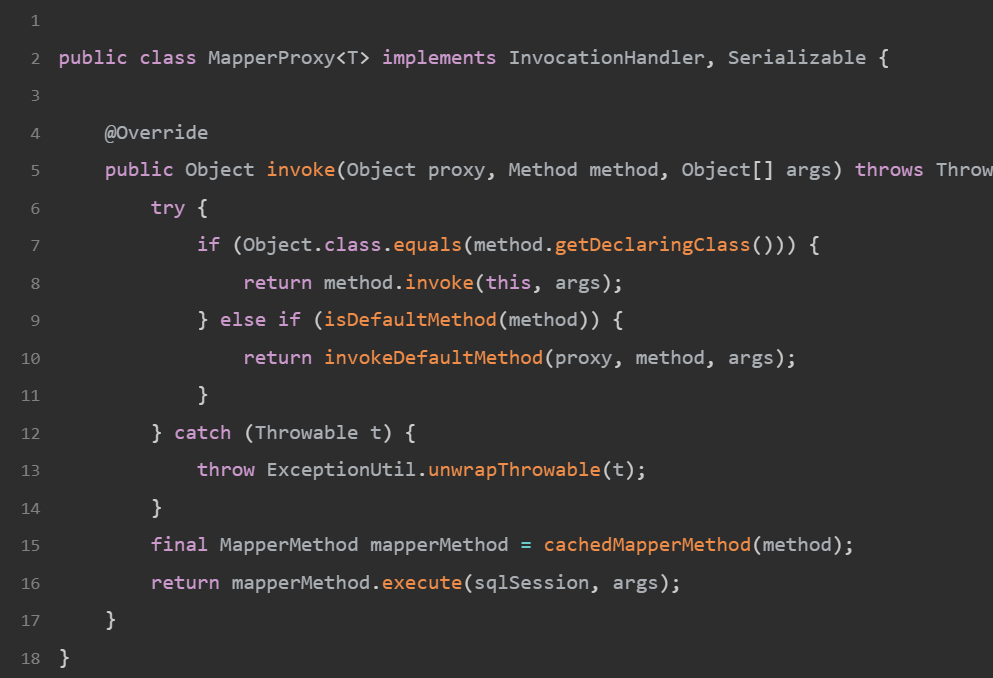


这里有两个步骤，第一个是提前创建一个Proxy，第二个是使用的时候会自动请求Proxy，然后由Proxy来执行具体事务；

当我们使用Configuration的getMapper方法时，会调用mapperRegistry.getMapper方法，而该方法又会调用mapperProxyFactory.newInstance(sqlSession)来生成一个具体的代理：

在这里，先通过T newInstance(SqlSession sqlSession)方法会得到一个MapperProxy对象，然后调用T newInstance(MapperProxy mapperProxy)生成代理对象然后返回。

而查看MapperProxy的代码，可以看到如下内容：



非常典型的，该MapperProxy类实现了InvocationHandler接口，并且实现了该接口的invoke方法。

通过这种方式，我们只需要编写Mapper.java接口类，当真正执行一个Mapper接口的时候，就会转发给MapperProxy.invoke方法，而该方法则会调用后续的sqlSession.cud>executor.execute>prepareStatement等一系列方法，完成SQL的执行和返回。

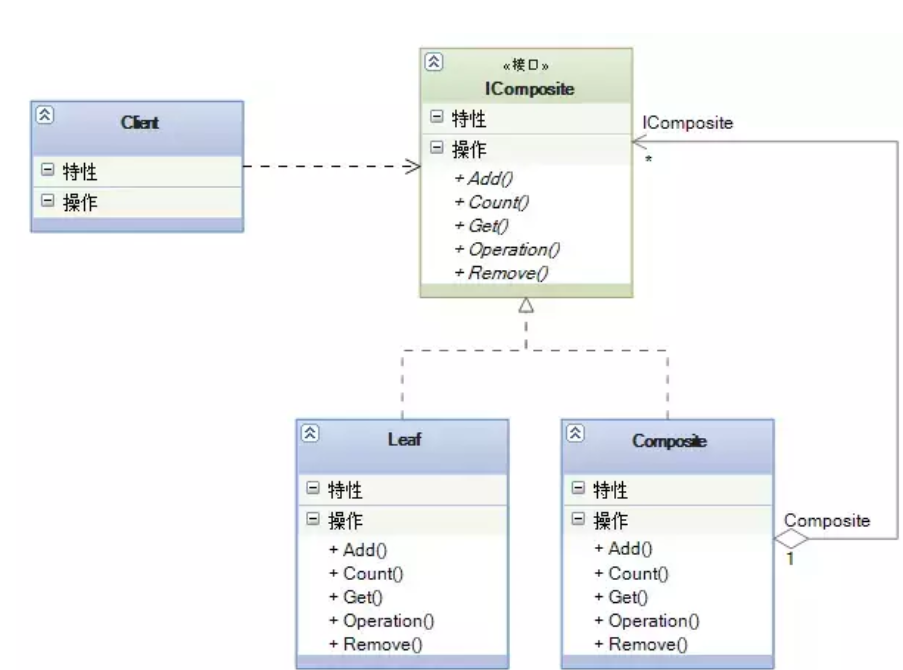
### 组合模式

<https://note.youdao.com/ynoteshare1/index.html?id=4975cd9e83f1e73e14a369598a232abe&type=note> 戏说组合

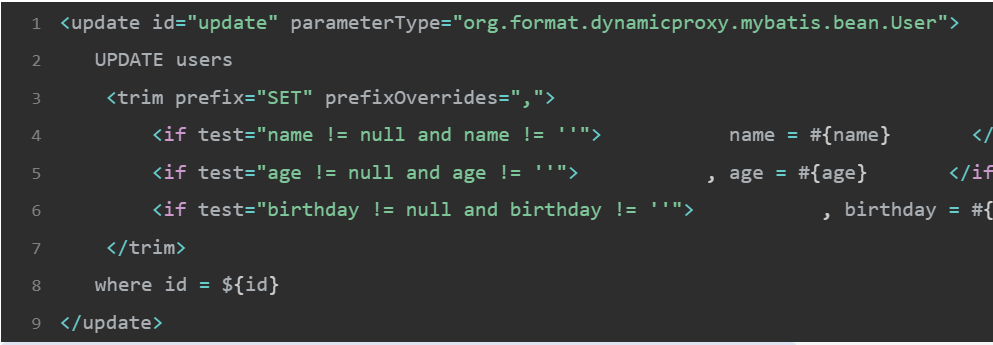
组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次。

组合模式对单个对象(叶子对象)和组合对象(组合对象)具有一致性，它将对象组织到树结构中，可以用来描述整体与部分的关系。同时它也模糊了简单元素(叶子对象)和复杂元素(容器对象)的概念，使得客户能够像处理简单元素一样来处理复杂元素，从而使客户程序能够与复杂元素的内部结构解耦。

在使用组合模式中需要注意一点也是组合模式最关键的地方：叶子对象和组合对象实现相同的接口。这就是组合模式能够将叶子节点和对象节点进行一致处理的原因

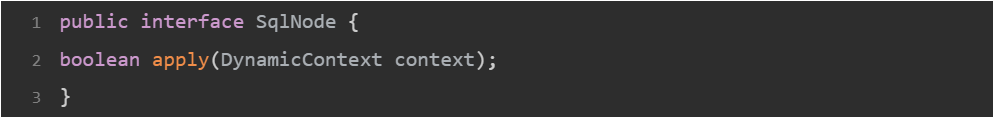
Mybatis支持动态SQL的强大功能，比如下面的这个SQL：

XHTML

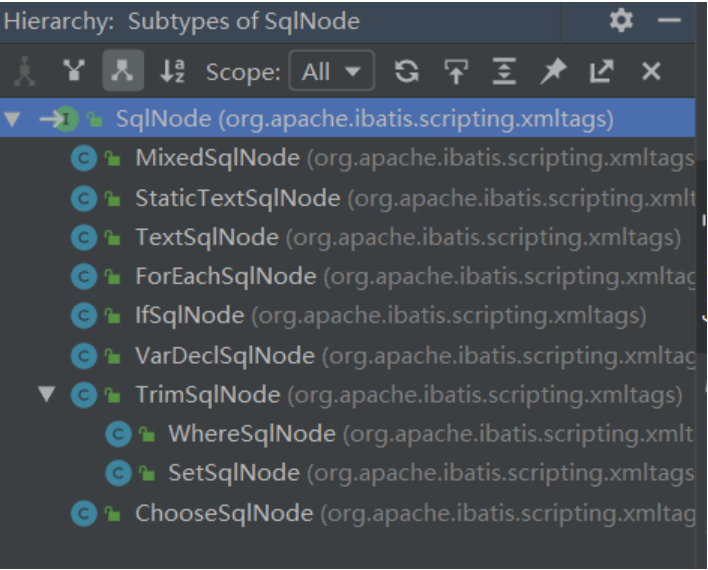


在这里面使用到了trim、if等动态元素，可以根据条件来生成不同情况下的SQL；

在DynamicSqlSource.getBoundSql方法里，调用了rootSqlNode.apply(context)方法，apply方法是所有的动态节点都实现的接口：



对于实现该SqlNode 接口的所有节点，就是整个组合模式树的各个节点：



组合模式的简单之处在于，所有的子节点都是同一类节点，可以递归的向下执行，比如对于TextSqlNode，因为它是最底层的叶子节点，所以直接将对应的内容append到SQL语句中：



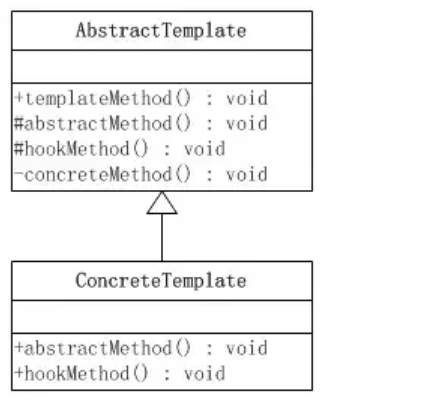
### 模板模式

<https://mp.weixin.qq.com/s/iRnihhHtvx1JYcI1hrd9vg> 戏说模板

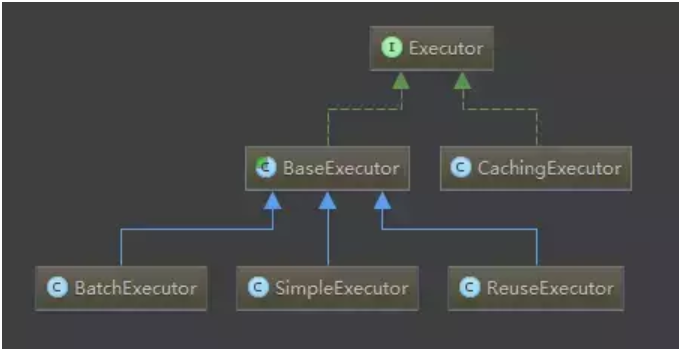
模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。

模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法(primitive method)；而将这些基本方法汇总起来的方法叫做模板方法(template method)，这个设计模式的名字就是从此而来。

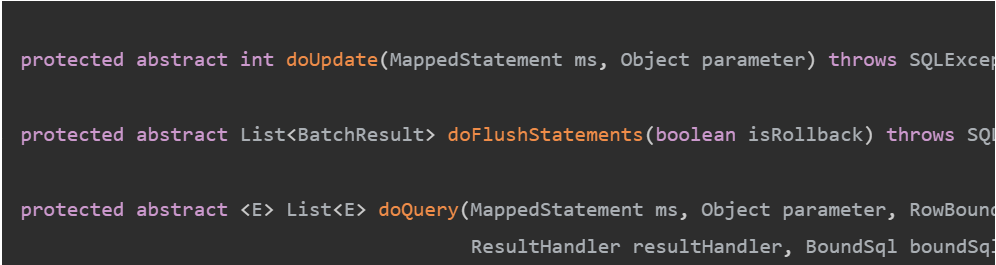
模板类定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。



在Mybatis中，sqlSession的SQL执行，都是委托给Executor实现的，Executor包含以下结构：



其中的BaseExecutor就采用了模板方法模式，它实现了大部分的SQL执行逻辑，然后把以下几个方法交给子类定制化完成：



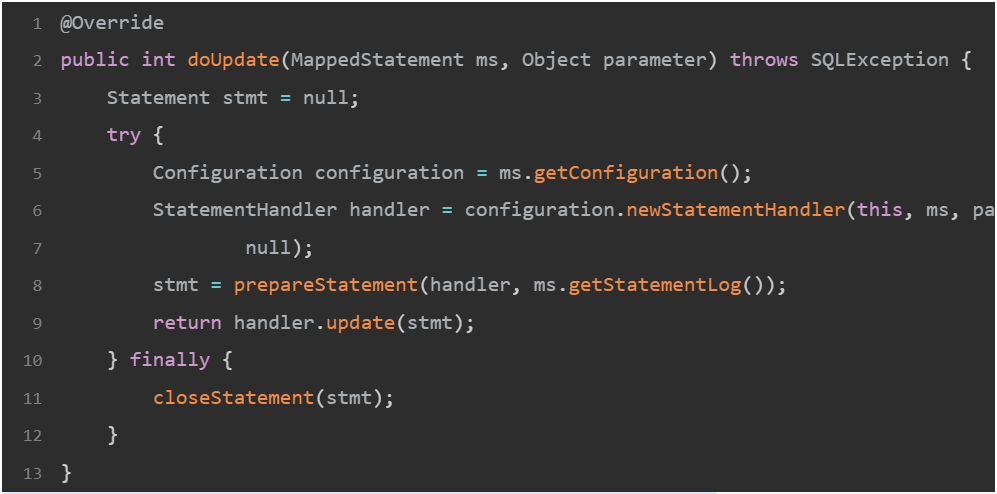
该模板方法类有几个子类的具体实现，使用了不同的策略：

简单SimpleExecutor：每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。（可以是Statement或PrepareStatement对象）

重用ReuseExecutor：执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。（可以是Statement或PrepareStatement对象）

批量BatchExecutor：执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理的；BatchExecutor相当于维护了多个桶，每个桶里都装了很多属于自己的SQL，就像苹果蓝里装了很多苹果，番茄蓝里装了很多番茄，最后，再统一倒进仓库。（可以是Statement或PrepareStatement对象）

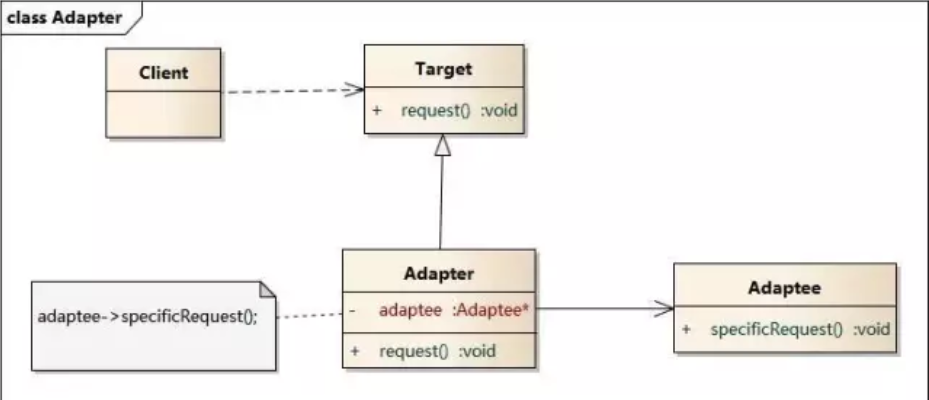
比如在SimpleExecutor中这样实现update方法：



### 适配器模式

<https://mp.weixin.qq.com/s/NBF5IvkboC8wt_DhYaHhvA>

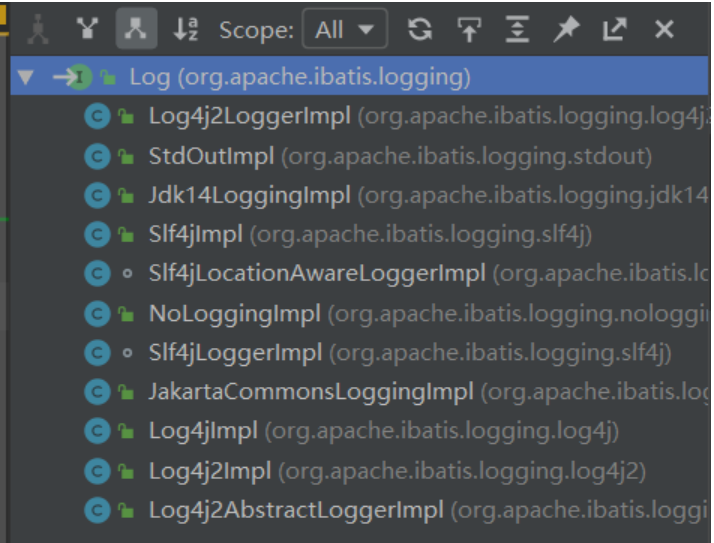
适配器模式(Adapter Pattern) ：将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为适配器(Adapter )。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。



在Mybatsi的logging包中，有一个Log接口：



该接口定义了Mybatis直接使用的日志方法，而Log接口具体由谁来实现呢？Mybatis提供了多种日志框架的实现，这些实现都匹配这个Log接口所定义的接口方法，最终实现了所有外部日志框架到Mybatis日志包的适配：



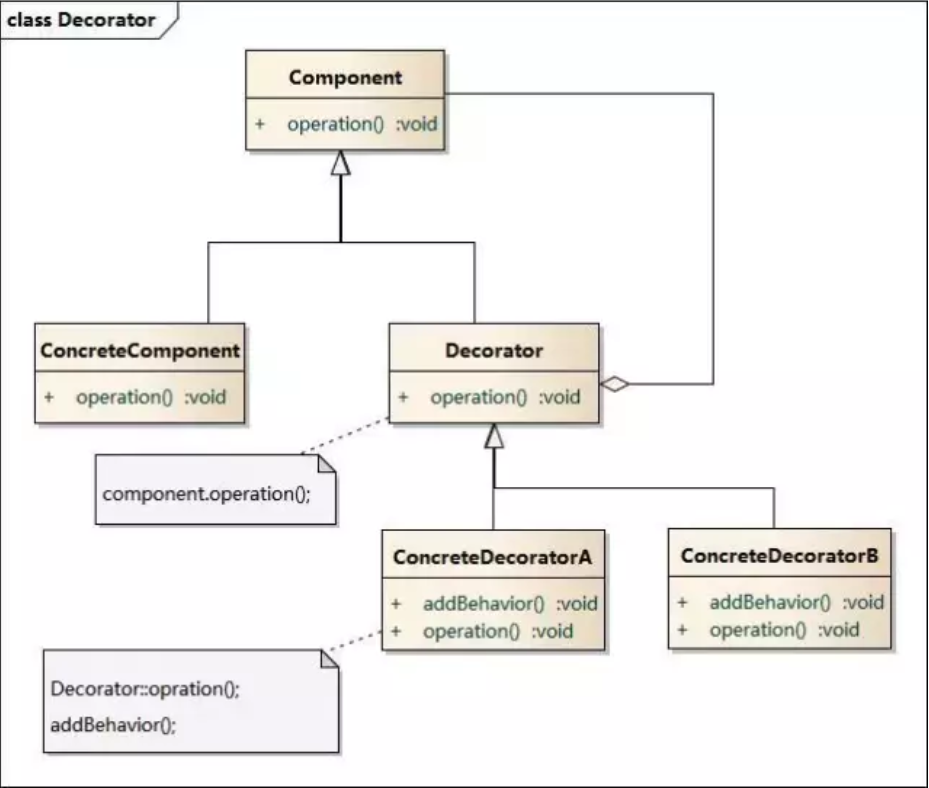
比如对于Log4jImpl的实现来说，该实现持有了org.apache.log4j.Logger的实例，然后所有的日志方法，均委托该实例来实现。

### 装饰器模式

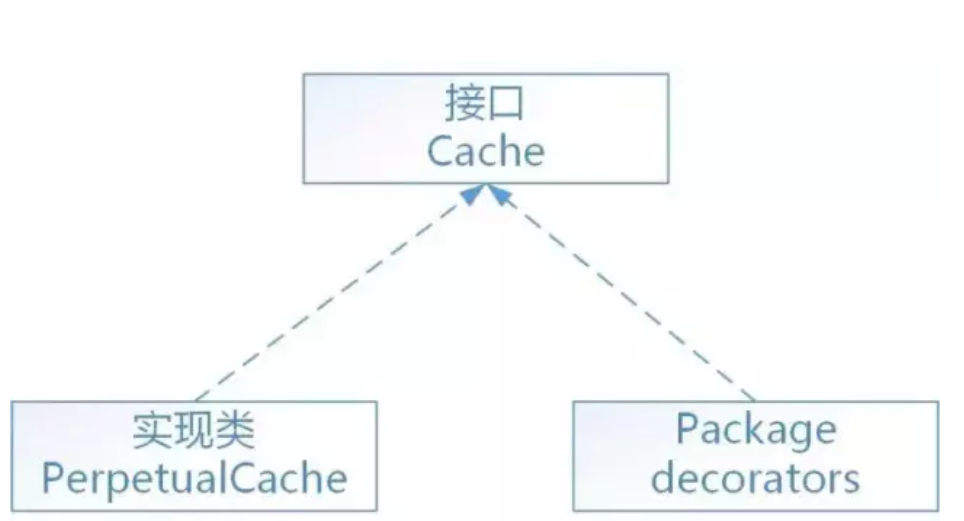
<https://mp.weixin.qq.com/s/v4jEeegy911hqhqnIC8s5w>

装饰模式(Decorator Pattern) ：动态地给一个对象增加一些额外的职责(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper、Decorator)。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

\* 1，不改变原类文件。 \* 2，不使用继承。 \* 3，动态扩展。



在mybatis中，缓存的功能由根接口Cache（org.apache.ibatis.cache.Cache）定义。整个体系采用装饰器设计模式，数据存储和缓存的基本功能由PerpetualCache（org.apache.ibatis.cache.impl.PerpetualCache）永久缓存实现，然后通过一系列的装饰器来对PerpetualCache永久缓存进行缓存策略等方便的控制。如下图：



用于装饰PerpetualCache的标准装饰器共有8个（全部在org.apache.ibatis.cache.decorators包中）：

FifoCache：先进先出算法，缓存回收策略

LoggingCache：输出缓存命中的日志信息

LruCache：最近最少使用算法，缓存回收策略

ScheduledCache：调度缓存，负责定时清空缓存

SerializedCache：缓存序列化和反序列化存储

SoftCache：基于软引用实现的缓存管理策略

SynchronizedCache：同步的缓存装饰器，用于防止多线程并发访问

WeakCache：基于弱引用实现的缓存管理策略

另外，还有一个特殊的装饰器TransactionalCache：事务性的缓存

正如大多数持久层框架一样，mybatis缓存同样分为一级缓存和二级缓存

一级缓存，又叫本地缓存，是PerpetualCache类型的永久缓存，保存在执行器中（BaseExecutor），而执行器又在SqlSession（DefaultSqlSession）中，所以一级缓存的生命周期与SqlSession是相同的。

二级缓存，又叫自定义缓存，实现了Cache接口的类都可以作为二级缓存，所以可配置如encache等的第三方缓存。二级缓存以namespace名称空间为其唯一标识，被保存在Configuration核心配置对象中。

二级缓存对象的默认类型为PerpetualCache，如果配置的缓存是默认类型，则mybatis会根据配置自动追加一系列装饰器。

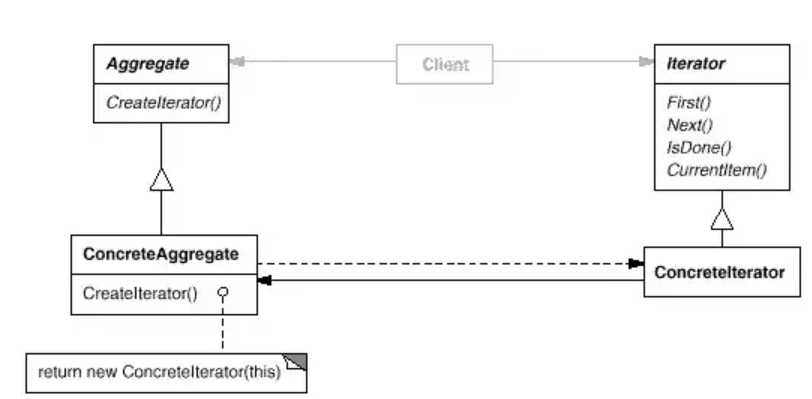
Cache对象之间的引用顺序为：

SynchronizedCache–>LoggingCache–>SerializedCache–>ScheduledCache–>LruCache–>PerpetualCache

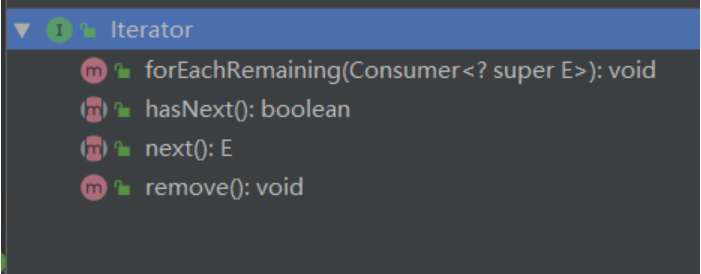
### 迭代器模式

<https://mp.weixin.qq.com/s/gxIbhqjMZKx3PwOHUMeTYw>

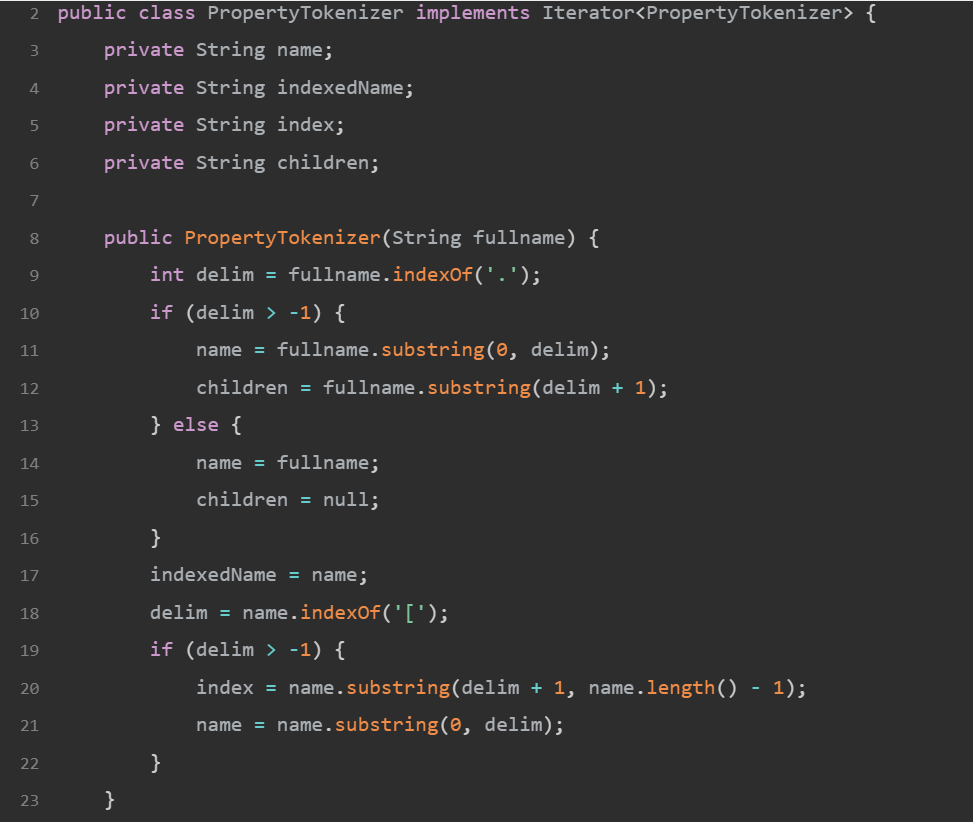
迭代器（Iterator）模式，又叫做游标（Cursor）模式。GOF给出的定义为：提供一种方法访问一个容器（container）对象中各个元素，而又不需暴露该对象的内部细节。



Java的Iterator就是迭代器模式的接口，只要实现了该接口，就相当于应用了迭代器模式：



比如Mybatis的PropertyTokenizer是property包中的重量级类，该类会被reflection包中其他的类频繁的引用到。这个类实现了Iterator接口，在使用时经常被用到的是Iterator接口中的hasNext这个函数。



可以看到，这个类传入一个字符串到构造函数，然后提供了iterator方法对解析后的子串进行遍历，是一个很常用的方法类。