# 深度分析mybatis源码

目录

[深度分析mybatis源码 1](#_Toc17452869)

[1. 前言 2](#_Toc17452870)

[2. 概述 2](#_Toc17452871)

[3. mybatis初始化 2](#_Toc17452872)

[3.1 读取配置文件 3](#_Toc17452873)

[3.2 解析configuration标签 4](#_Toc17452874)

[3.3 解析properties标签 6](#_Toc17452875)

[3.4 解析settings标签 7](#_Toc17452876)

[3.5 解析typeAliases标签 8](#_Toc17452877)

[3.6 解析plugins标签 9](#_Toc17452878)

[3.7 解析environments标签 11](#_Toc17452879)

[3.7.1 解析transactionManager标签 12](#_Toc17452880)

[3.7.2 解析dataSource标签 14](#_Toc17452881)

[3.8 解析typeHandlers标签 14](#_Toc17452882)

[3.9 解析mappers标签 17](#_Toc17452883)

[4. 获取SqlSession实例 21](#_Toc17452884)

[5. 获取代理的Mapper(DAO)接口实例 22](#_Toc17452885)

[6. 执行SQL语句 22](#_Toc17452886)

[7. 参数解析及动态SQL计算 22](#_Toc17452887)

[8. 结果集映射封装 22](#_Toc17452888)

# 前言

至于mybatis是什么，怎么用等基础问题，本书不做过多赘述，可以详细看其官网，

中文官网地址：<http://www.mybatis.org/mybatis-3/zh/index.html>

英文官网地址：<http://www.mybatis.org/mybatis-3/>

源码地址：https://github.com/mybatis/mybatis-3.git

在阅读本书时，假设你已有java基础，使用过mybatis,了解jdbc编程模式

# 概述

为了便于理解mybatis执行SQL语句的整体流程，可以将整个过程粗略的拆解为6个大的模块进行分析，分别是

* mybatis初始化
* 获取SqlSession实例
* 获取Mapper(DAO)接口的代理类实例
* 执行SQL语句
* 参数解析及动态SQL计算
* 结果集映射封装

在这一章节，我将简单的概述一下这6大模块，将你脑海中对mybatis断断续续的理解串联成一条线，也为阅读本书后续章节做好铺垫。

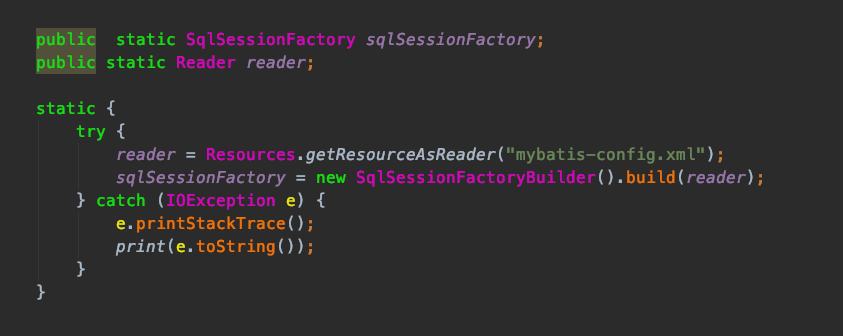
本书将按上述这几个模块为主线，进行递进式的分析，更好的帮助读者理解mybatis整体架构，形成系统化的知识图谱。

最后，本书会给你带来什么，这也是阅读本书的最终价值，初中级开发工程师进阶高级工程师的道路是艰辛的，阅读本书且深入理解经典框架的设计思想可以认为是一条捷径。

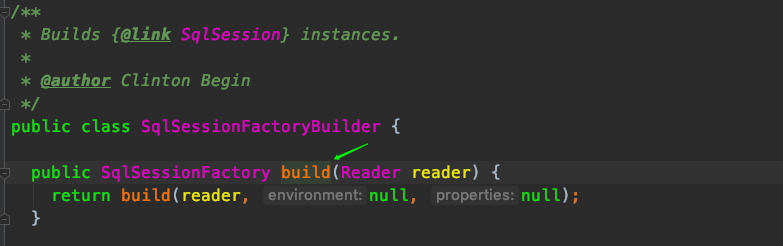
# mybatis初始化

如果你对mybatis万物归一的Configuration类没有深入的了解，那阅读这一章节是很有必要的。面向对象编程模型中抽象和封装的思想，在这一章节得到了最好的诠释。

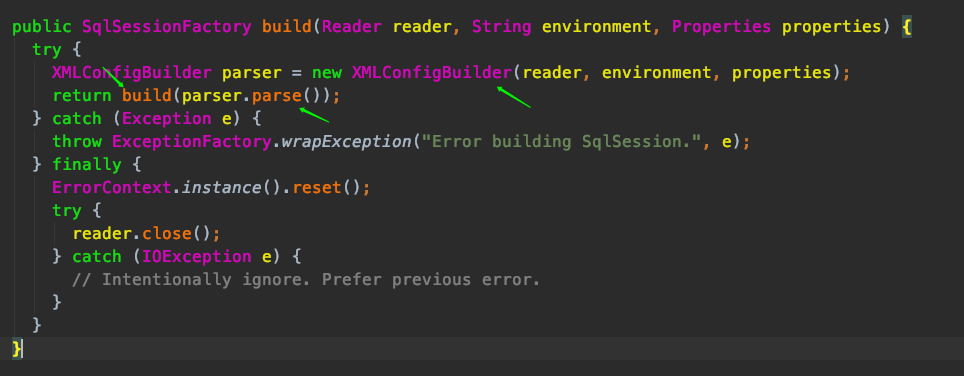
## 读取配置文件



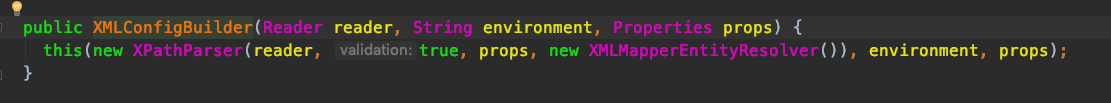
* 1. 使用mybatis提供的资源工具类<Resources>类，能实现很多资源操作，可以在自己应用中广泛使用，此处获取到一个Reader对象。
* 2. 实例化一个SqlSessionFactoryBuilder对象，看到这个命名，应该知道此类涉及到了两种设计模式，分别是Builder模式和工厂模式。
* 3. 通过SqlSessionFactoryBuilder实例的build()方法，处理Reader对象，读取流中的配置信息
* 3.1 build()方法调用如下



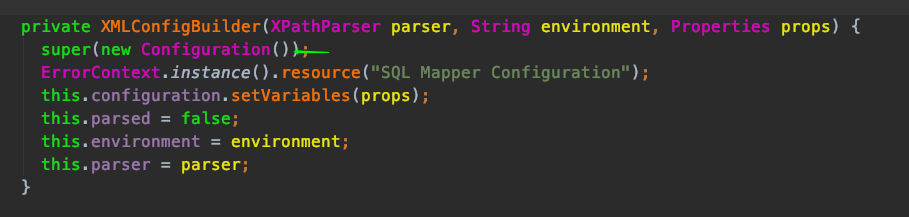
* 3.1.1内部调用了更多参数的重载build()方法，



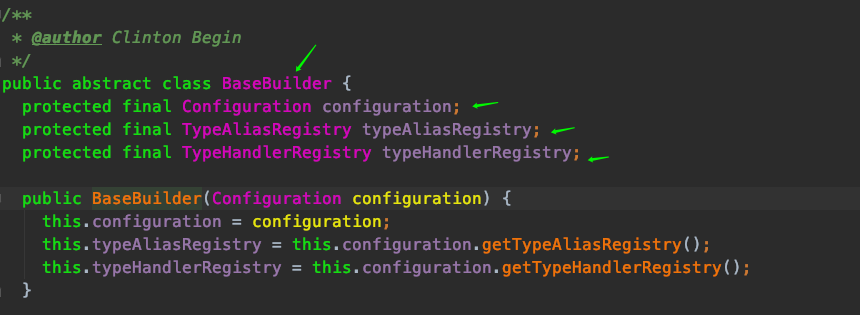
实例化了一个XMLConfigBuilder类的实例，根据\*\*\*Builder的命名模式，可以看出类又使用了builder设计模式，此类的构造方法也看一下



继续调用参数更多的构造方法，就在XMLConfigBuilder类的构造器中，实例化了万物归一的Configuration类，且作为参数继续调用了父类的构造方法



来看看父类BaseBuilder类的构造方法

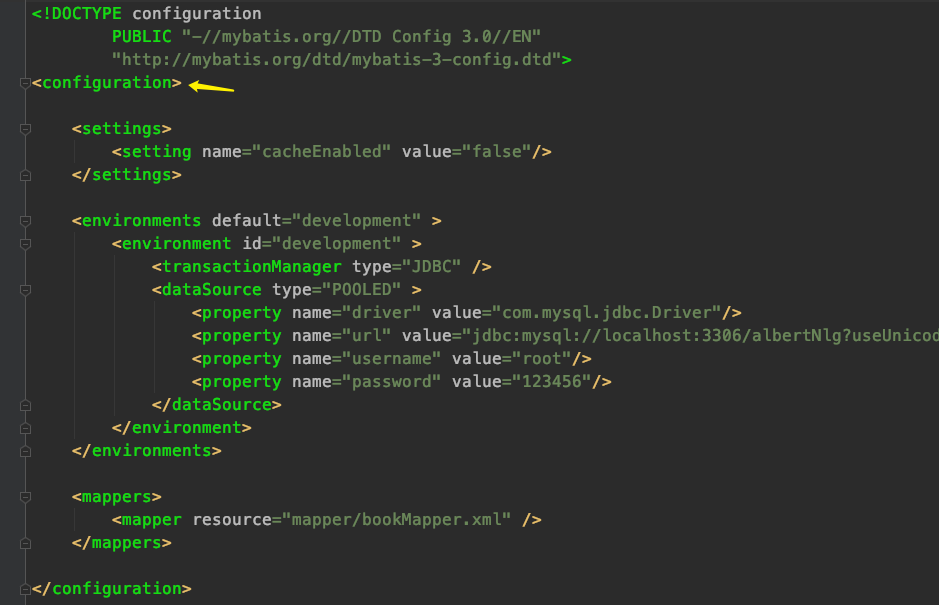


可以看到这是个抽象类，含有三个状态，分别持有Configuration类实例的引用，TypeAliasRegistry类实例的应用，TypeHandlerRegistry类实例的引用

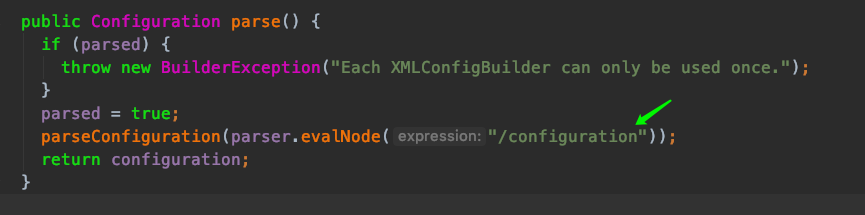
## 解析configuration标签

紧接着3.1.1处，调用了XMLConfigBuilder.parse()方法，内部先解析了config.xml配置文件的顶级标签<configuration/>

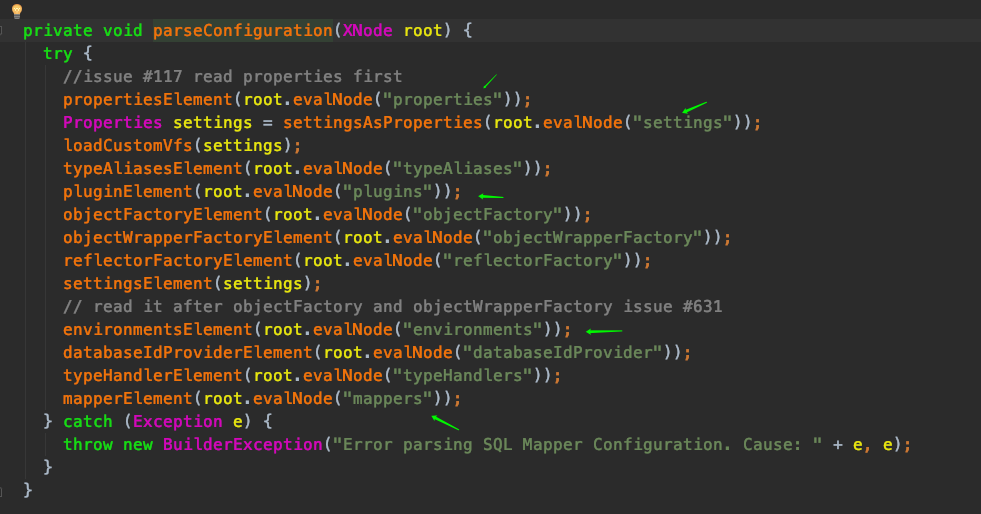
* 先看config.xml文件的配置



* 再看解析配置的代码



接着看parseConfiguration()方法的实现，针对xml配置文件的每一个第二层标签，都对应一个解析的方法，且方法名都命名为\*\*\*Element()。个人认为这也是一整巧妙的设计，值得我们在实际编码过程中使用这种模式。



接下来，详细分析每个element方法的解析，初始化过程。

## 解析properties标签

* 先看config.xml文件的中配置

<configuration>  
 <properties resource="" url="">  
 <property name="id" value="100"></property>  
 </properties>

</configuration>

* 再看写具体解析properties标签的方法：propertiesElement()

private void propertiesElement(XNode context) throws Exception {  
 if (context != null) {  
 Properties defaults = context.getChildrenAsProperties();  
 String resource = context.getStringAttribute("resource");  
 String url = context.getStringAttribute("url");  
 if (resource != null && url != null) {  
 throw new BuilderException("…");  
 }  
 if (resource != null) {  
 defaults.putAll(Resources.*getResourceAsProperties*(resource));  
 } else if (url != null) {  
 defaults.putAll(Resources.*getUrlAsProperties*(url));  
 }  
 Properties vars = configuration.getVariables();  
 if (vars != null) {  
 defaults.putAll(vars);  
 }  
 parser.setVariables(defaults);  
 configuration.setVariables(defaults);  
 }  
}

* 分析：

1. 首先可以看到属性可以引用自外部属性文件可网络上的一个资源，但是二者不能同时配置，不然，会抛出BuilderException
2. 可以发现，此处再次使用了Resources工具类，来获取外部资源和网络资源上的property属性
3. 最终，读取出来的属性存进了万物归一的configuration对象中，这也就意味着，可以在程序运行期，通过configuration对象能使用这些属性配置

## 解析settings标签

* 先看配置

<configuration>  
 <settings>  
 <setting name="cacheEnabled" value="false"/>  
 <setting name="databaseId" value="mysql"/>  
 </settings>

</configuration>

也许你在好奇，settings标签中应该配置哪些值，因为name和value的值是没有编译约束的，意味着可以自定义随便写，但这在程序运行时会抛BuilderException异常。

* 如果你详细看过Configuration的源码，我想你应该已经猜到了，该类中有大量的状态即类变量【注：类变量在本书中统称“状态”，引用自《java concurrence in practice》一书】，这些状态几乎都有默认值，例如控制是否开启二级缓存的状态变量cacheEnable

protected boolean cacheEnabled = true;

但是，有没有想过如果想在运行期改变它们的值，那就需要有外部配置提供一个入口，而

</settings>标签正是作用与此。

* 接下来看解析该标签的过程

Properties settings = settingsAsProperties(root.evalNode("settings"));  
loadCustomVfs(settings);

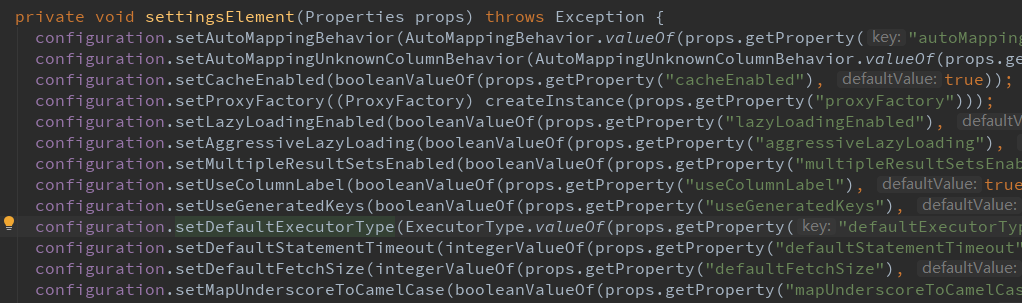
settingsElement(settings);

* 分析：

1. 先将setting的配置读到一个properties对象，读的过程中会校验setting配置的值是否有效，如果不在约束范围内，则会抛出BuilderException异常

private Properties settingsAsProperties(XNode context) {  
 if (context == null) {  
 return new Properties();  
 }  
 Properties props = context.getChildrenAsProperties();  
 // Check that all settings are known to the configuration class  
 MetaClass metaConfig = MetaClass.*forClass*(Configuration.class, localReflectorFactory);  
 for (Object key : props.keySet()) {  
 if (!metaConfig.hasSetter(String.*valueOf*(key))) {  
 throw new BuilderException("The setting " + key + " is not known. Make sure you spelled it correctly (case sensitive).");  
 }  
 }  
 return props;  
}

1. 看下settingsElement()方法内部实现



看到此处，应该恍然大悟了，config.xml文件中</settings>标签真正的用途

## 解析typeAliases标签

先解释下，typeAliase到底是什么，因为在mybatis初始化，运行期等到大量用到了它，举个例子，中国的全程叫中华人民共和国，但大家一致都叫中国，是不是简单又好记，程序也是这样，用一个简短且人更容易识别的别名，来表示哪些需要很长的文字才能标识的事物。

* 看配置

<configuration>  
 <typeAliases>  
 <typeAlias

type="com.peony.bean.ObjectNumberQuestion"

alias="objectNumber">

</typeAlias>  
 </typeAliases>

</configuration>

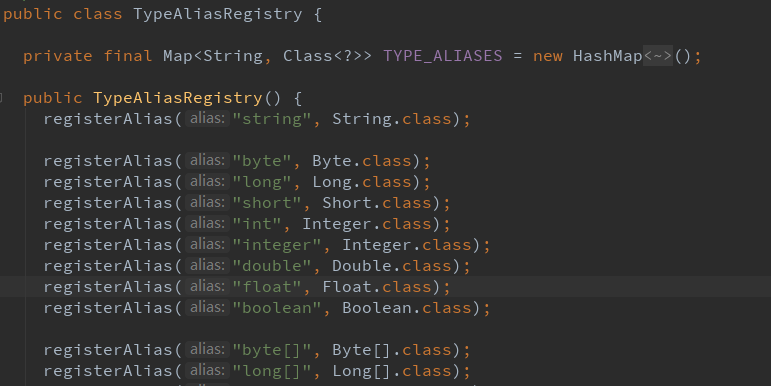
大家都知道，在java程序中，要唯一识别并找到一个class, 需要用相对路径下的全限定名称去找，而给这个起个本系统中不重复的别名，对应这个全限定地址，以后在其他地方就可以直接用别名了，增加了代码的简洁性。

* 再看解析该标签的代码

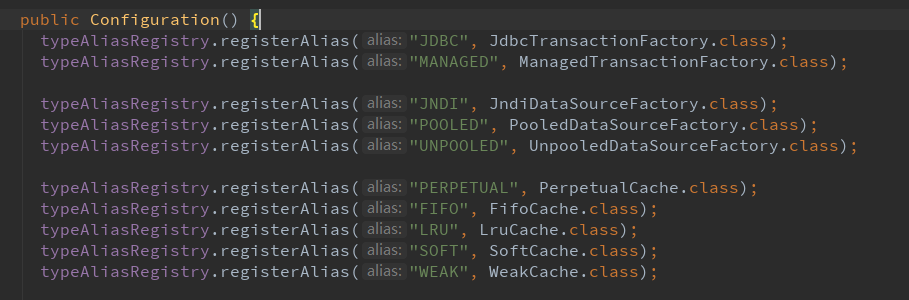
private void typeAliasesElement(XNode parent) {  
 if (parent != null) {  
 for (XNode child : parent.getChildren()) {  
 if ("package".equals(child.getName())) {  
 String typeAliasPackage = child.getStringAttribute("name");  
 configuration.getTypeAliasRegistry().registerAliases(typeAliasPackage);  
 } else {  
 String alias = child.getStringAttribute("alias");  
 String type = child.getStringAttribute("type");  
 try {  
 Class<?> clazz = Resources.*classForName*(type);  
 if (alias == null) {  
 typeAliasRegistry.registerAlias(clazz);  
 } else {  
 typeAliasRegistry.registerAlias(alias, clazz);  
 }  
 } catch (ClassNotFoundException e) {  
 throw new BuilderException("Error registering typeAlias for '" + alias + "'. Cause: " + e, e);  
 }  
 }  
 }  
 }  
}

分析：

1. 使用Resources工具类，将通过类全限定名，反射得到该类的Class对象
2. 将别名和类的Class对象注册到Configuration对象的typeAliasRegistry属性中
3. 除了从配置文件中注册进去的别名，还有两个地方也注册了别名
   1. 在TypeAliasRegistry的构造方法中注册了java基本类的别名，可以看到类型别名的核心是一个私有访问，final修饰的HashMap<String,Class<?>>，所谓注册也就是往map中put键值对



* 1. 在Configuration类的构造方法中，注册了Mybatis框架所需的类型别名



## 解析plugins标签

plugins标签可以配置一些插件，最常见就是分页插件了，通过拦截器的方式，拦截mybatis的指定类，执行一些自己的逻辑。

* 先看配置（此处引用了分页插件PageHelper作为示例，下面会讲到）

<configuration>  
 <plugins>  
 <plugin interceptor="com.github.pagehelper.PageInterceptor">  
 </plugin>  
 </plugins>

</configuration>

* 再看解析代码

是将interceptor标签被表示的类通过反射的方法，获取到实例，再封装properties

* private void pluginElement(XNode parent) throws Exception {  
   if (parent != null) {  
   for (XNode child : parent.getChildren()) {  
   String interceptor = child.getStringAttribute("interceptor");  
   Properties properties = child.getChildrenAsProperties();  
   Interceptor interceptorInstance = (Interceptor) resolveClass(interceptor).newInstance();  
   interceptorInstance.setProperties(properties);  
   configuration.addInterceptor(interceptorInstance);  
   }  
   }  
  }

public void addInterceptor(Interceptor interceptor) {  
 interceptorChain.addInterceptor(interceptor);  
}

最终是将该拦截器接口实现类添加到了configuration对象的interceptorChain中了，下面就是interceptorChain的真面目

protected final InterceptorChain interceptorChain = new InterceptorChain();

* 刚才在配置中，我引用了PageHelper插件作为例子，那顺便来看看这个PageHelperInterceptor的源码
* @Intercepts({@Signature(  
   type = Executor.class,  
   method = "query",  
   args = {MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class}  
  ), @Signature(  
   type = Executor.class,  
   method = "query",  
   args = {MappedStatement.class, Object.class, RowBounds.class, ResultHandler.class, CacheKey.class, BoundSql.class}  
  )})  
  public class PageInterceptor implements Interceptor {

可以看到这是个mybatis拦截器接口Interceptor的实现类，用注解标明了要拦截的方法签名，分别拦截Executor类的4个参数的query方法和6个参数的query方法，到了这里，你大概已经明白了分页插件的底层实现，貌似是拦截了mybatis的执行器Executor类，然后改写了SQL语句，再返回给mybatis继续执行，这里我先抛出这个点，等待后续验证。

## 解析environments标签

注明：我省略了<objectFacory>, <objectWrapperFactory>, <refectorFactory>, <databaseIdProvider>这四个标签的解析，这四个标签在实际场景中并不多见

* 先看配置

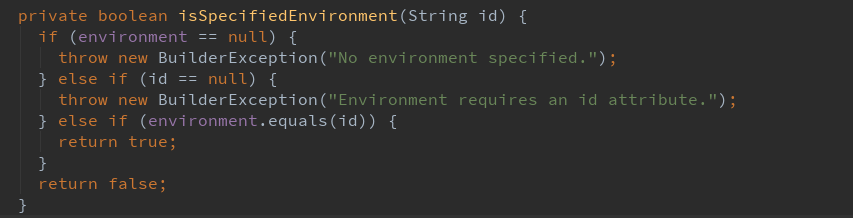


* 再看解析代码



* 分析：

1. 上图中，可以看出，如果environments标签中，没有配置default的值，会抛出BuilderException异常
2. 如果environment标签中，没有配置id的值，也会抛出BuilderException异常



1. 只有当environments-default的值等于environment-id的值是，才会解析environment标签，这也就是说environments标签内可以配置多个environment，mybatis会根据default的值选择其中一个进行解析，即default指定了当前要启用的数据库环境是哪个
2. 最终解析完transactionManager和dataSource后，通过Builder模式，建造Environment对象，并设置transactionManager和dataSource的值
3. 最后，将environment对象设置到万物归一的configuration对象，下面将详细讲解transactionManager和dataSource的解析过程

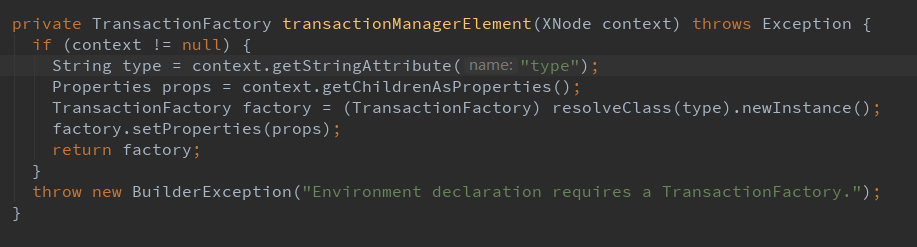
### 解析transactionManager标签

事务通常是应用中最重要的核心部分，而事务管理器提供了事务能力，不同的事务管理器，实现了不同应用环境下的事务要求，此处以JDBC的事务管理器为例

* 先看配置

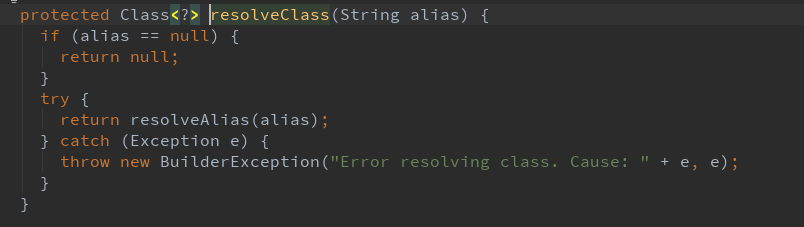


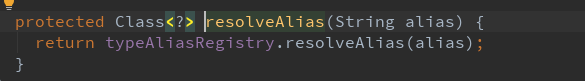
* 再看解析代码

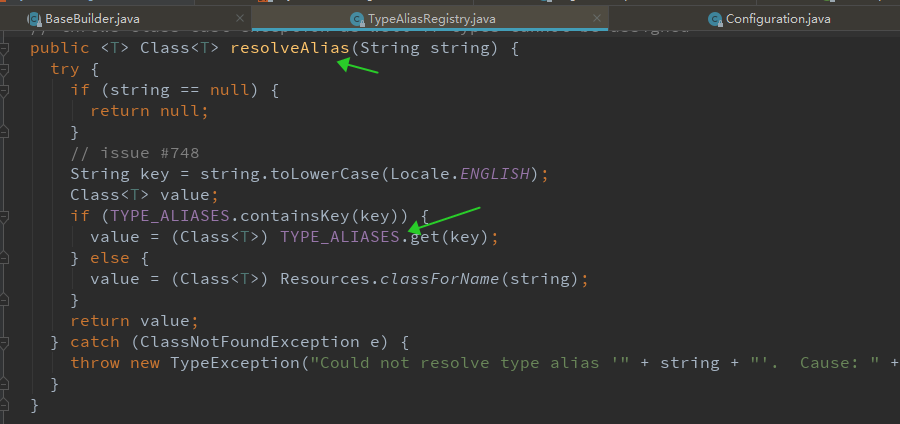


1. 从标签中获取到type的值，此处是”JDBC”
2. 得到type后，通过反射生成了一个事务工厂类的实例，此处使用了工厂模式，但是TransactionFactory是个接口，那猜想生成的实例肯定是它的一个实现类，且应该跟我们配置文件中的type=“JDBC”有关，

扩展一下：通过工厂模式生成实例，mybatis这种通过别名注册器获取类型的方式值得学习，可以替代传统编码中很多的if-else代码，不光光是这种思想，就算是类的命名，也值得学习，\*\*\*Factory,\*\*\*Builder,\*\*\*Registry,\*\*\*Handler,\*\*\*Resolvor等等



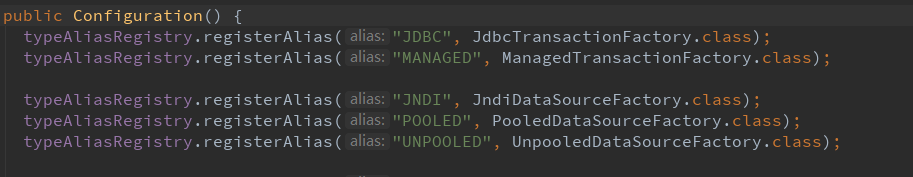




1. 看到上图，如果你还能想起我前面讲述的typeAliasRegistry标签的解析，就能知道我们配置的

<transactionManager type="JDBC" />

type=”JDBC”, JDBC只是个别名，而它正真对应的类型是什么呢？这里仅仅是从类型别名注册器中获取，但是正真的值即JDBC这个别名对应的类型，是什么时候注册进TypeAliasRegistry的呢？请看3.5，讲过别名注册器中注册类型有三个地方，分别是解析别名配置，调用TypeAliasRegistry的构造方法，调用Configuration类的构造方法，而别名JDBC正是在Configuration类的构造方法中注册进去的



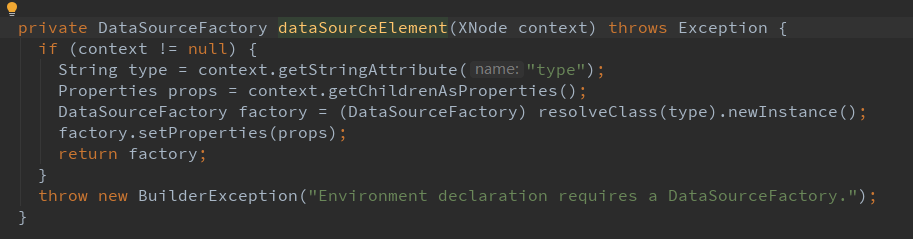
1. 那么，解析完transactionManager标签，就得到了JdbcTransactionFactory类的实例，看到这种类名，就能想到这个类后续是要生产JdbcTransaction类实例的

### 解析dataSource标签

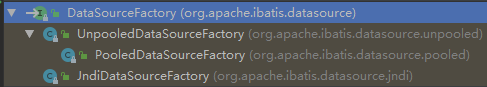
* 先看配置



* 再看解析代码



1. 此处type的解析如上，也是通过别名注册器中找到对应的类型，反射生成对应的实例，此处我配置了POOLED，也就意味着要使用数据库连接池。顺便也看下mybatis提供哪些DataSource



1. 解析properties属性，并赋值到dataSourceFactory实例中

## 解析typeHandlers标签

TypeHandlerRegistry类保存了java类型了JDBC类型之间的对应关系处理的处理类，这也是做参数对象映射到SQL语句的参数，查询返回值处理为应用层的java对象等操作的桥梁

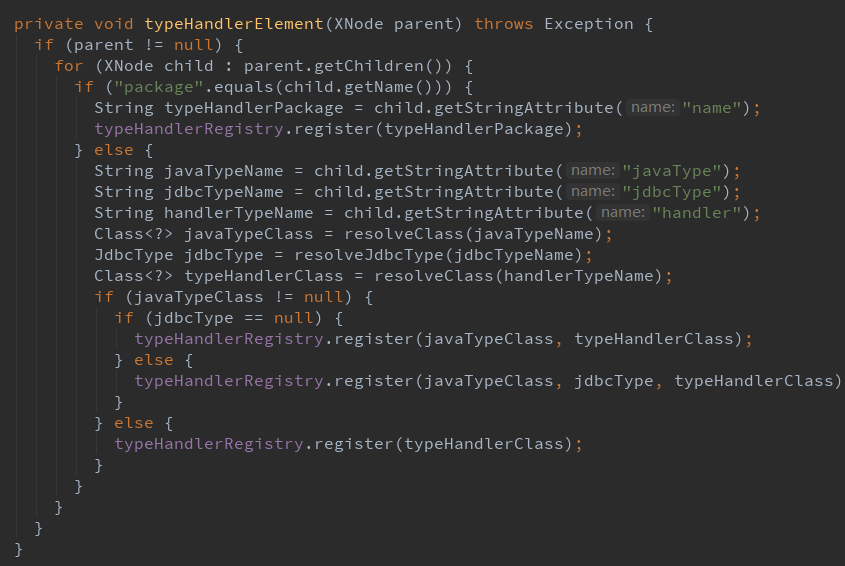
* 先看配置

<configuration>  
<typeHandlers>  
 <typeHandler handler="com.peony.utils.MyStringHandler"  
 jdbcType="VARCHAR"  
 javaType="string">  
 </typeHandler>  
</typeHandlers>

</configuration>

typeHandler标签有三个属性，分别是handler(处理器)，jdbcType，javaType,

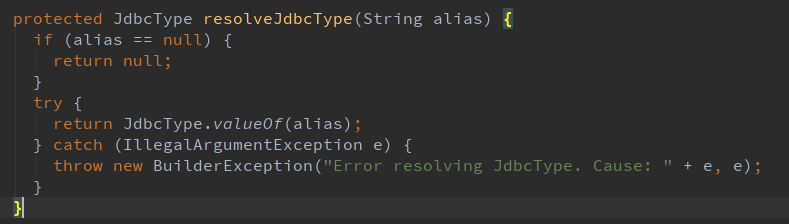
* 分析解析代码

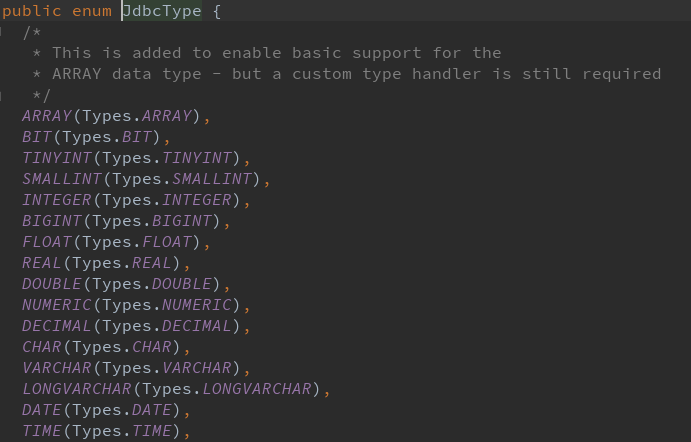


1. 解析javaType, 通过typeAliasRegistry获取到对应的类型。

注意，这里我配置的javaType=string,string就是java.lang.String的别名，解析类型是通过typeAliasRegistry去处理的，别名string会对应到java.lang.String

1. 解析jdbcType，是通过JdbcType枚举类获取的

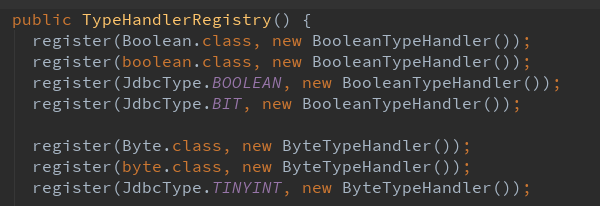




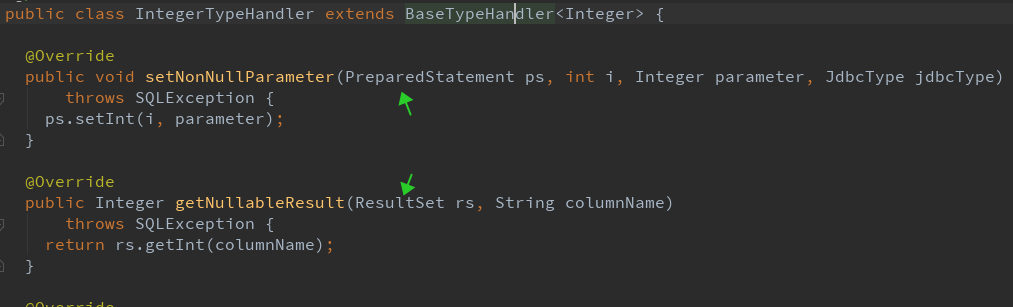
1. 最终将解析出来的java类型，jdbc类型，及其这两种类型的处理器等三者的对应关系缓存到Configuration对象的typeHandlerRegistry中。

或许你在想，java中怎么保存三个维度的对应关系，key-value关系可以用Map就轻而易举的保存了，如果再扩展一层呢，譬如Map<Type,Map<JdbcType,TypeHandler>>

1. TypeHandlerRegistry跟TypeAliasRegistry的设计一样，都是\*\*\*Registry，那除了要解析用户配置信息，还要初始化注册一些系统必须的信息，包括在\*\*\*Registry类本身的构造方法，或者在Configuration类的构造方法中。



1. TypeHandlerRegistry顾名思义是注册了一堆TypeHandler, 随便找一个来看看源码



发现类型处理器就是在处理SQL查询的传入参数和结果集的类型问题

## 解析mappers标签

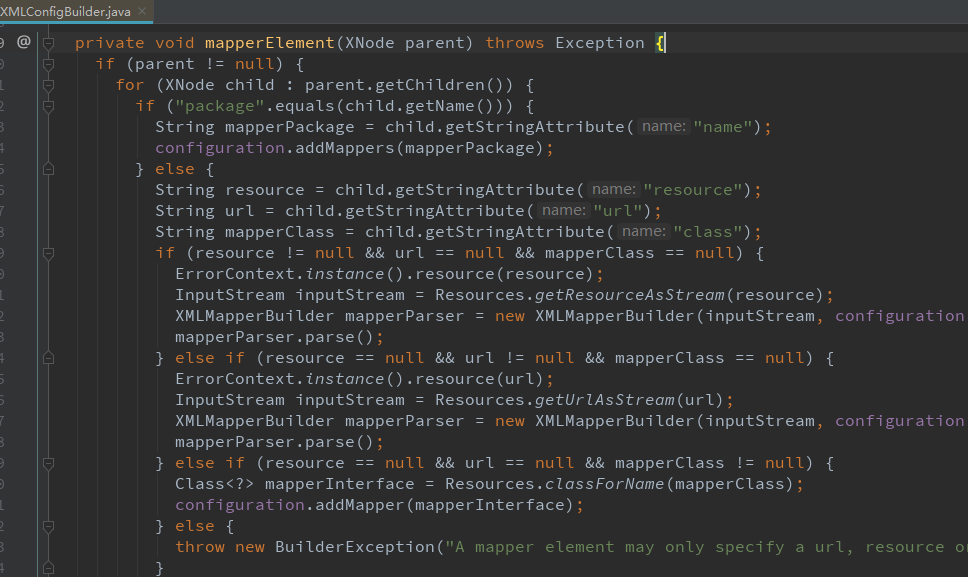
* 先看配置

<configuration>

<mappers>  
 <mapper resource="mapper/bookMapper.xml"   
 url="http://192.168.111.111"   
 class=""/>  
 </mappers>  
</configuration>

看一看到mapper标签有三个属性，分别是resource, url, class

* 再看解析代码

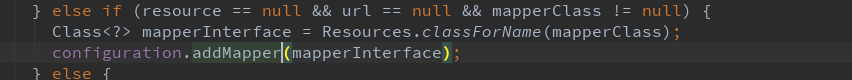


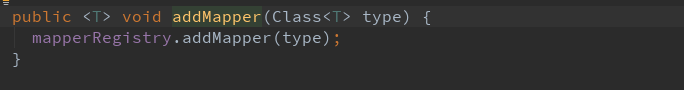
可以看到3个if-else，分别解析resource, url, class类型的配置，这三种属性最多只能配置一个，否则会抛出BuilderException。

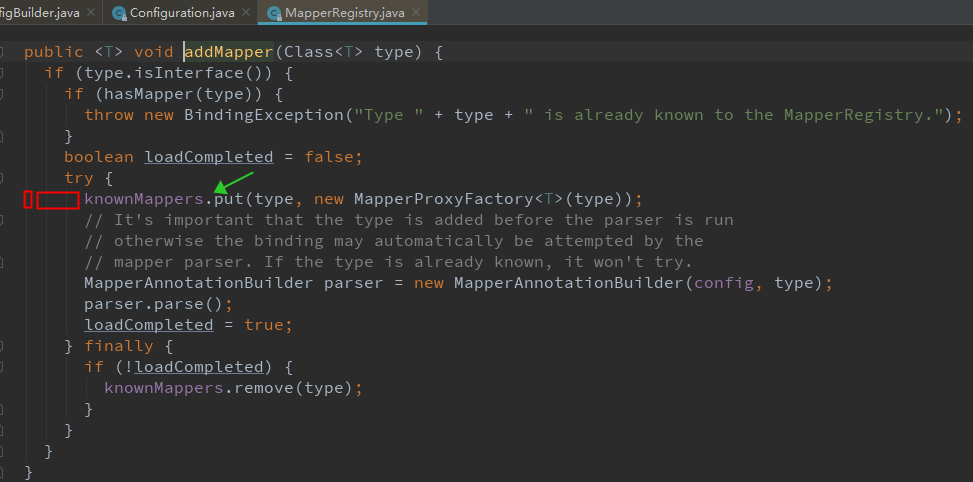
### class属性解析

1. 先看class的解析，因为这个比较简单

其实class属性应该配置Mapper(DAO)接口的全限定名，也就是mapper.xml文件中namespace属性配置的接口，这时你应该想到了，如果配置resource, 则指定的是mapper.xml文件的相对路径，mybatis解析它的时候也正是在解析它的namespace属性

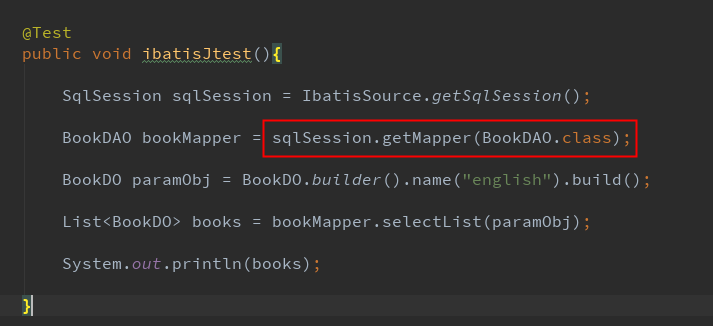






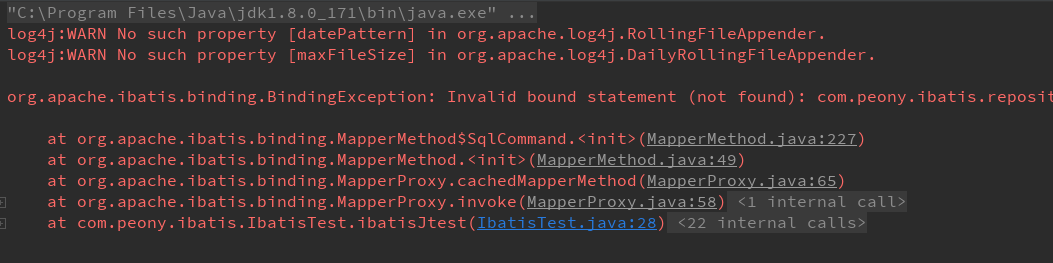
此处我用矩形标红，说明这块是比较重要的，因为在后面执行阶段，这里解析的内容需要用到

* + 解析到的type先put到一个knownMappers中了，其key就是这个Mapper(DAO)接口类的类型，**value是一个泛型的MapperProxyFactory, 通过其命名，就大概能想到这个类使用了代理模式和工厂模式，为后面执行阶段在调用接口的方法时，程序居然还能正确执行做了充足的准备**
  + **做个猜想，在执行阶段，通过**sqlSession.getMapper(BookDAO.class)获取到了一个BookDAO接口的实例，大概也是因为这里的配置起了作用，首先在**knownMappers**中找到该DAO接口类型对应的**MapperProxyFactory，再通过其生成MapperProxy实例，在通过MapperProxy生成一个**DAO接口的代理类。到这里，都是猜想，先留下伏笔，等到第5章节，通过源代码一步一步验证



好了，说完mapper标签中如果配置了classs属性，解析完后将其填加到了configuration对象的knownMapper中，

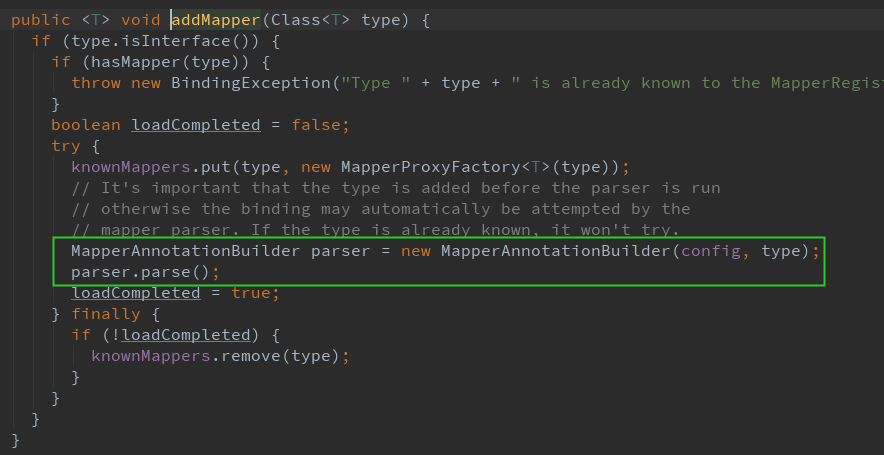
但是，如果这样配置，程序是不知道从哪里找到SQL语句了，因为没有任何mapper.xml的位置信息暴露给mybatis，如果直接运行，我认为肯定会报错，不妨试一试



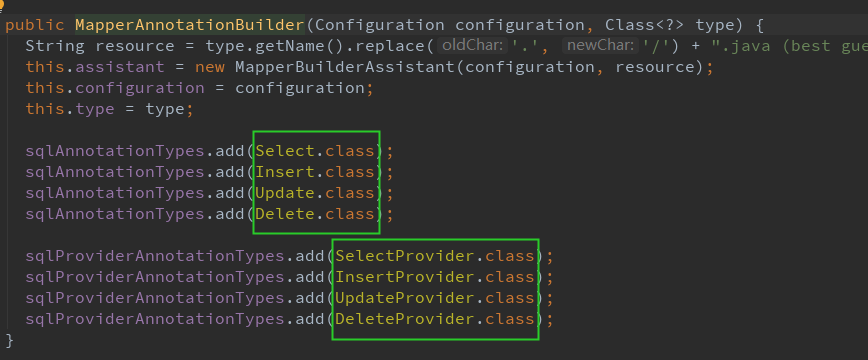
果然报错了，意思大概是该DAO接口没有绑定合适的statement，大意就是没有sql与之对应，

或许此时应该想一想了，既然mybatis在mapper标签中提供了class这么一个属性可以让我们配置DAO接口，class,url,resource等三个属性同时只能使用一个，那怎么让它知道该接口到底要执行什么SQL呢？我想到了注解，你呢？

接着看源码，其实很清楚的就能看到是在解析一些注解，其实mybatis的代码有很强的自解释性，有些源码不用去看内部实现，大概也能猜到一二。



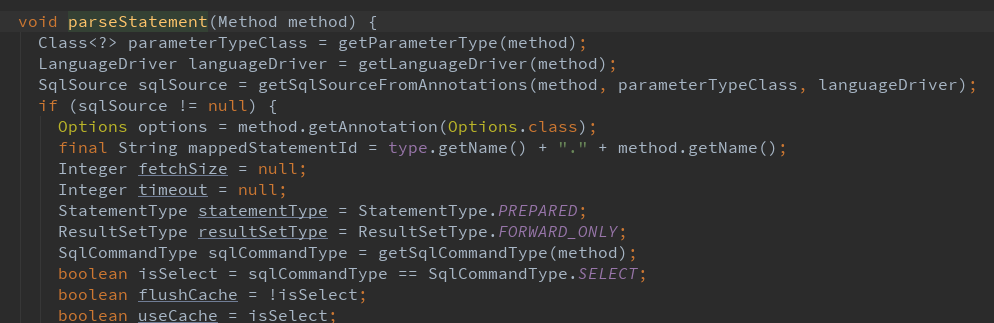
获取到MapperAnnotationBuilder类的实例，可以看到它的构造方法中出现了8中注解



再来看其parse方法

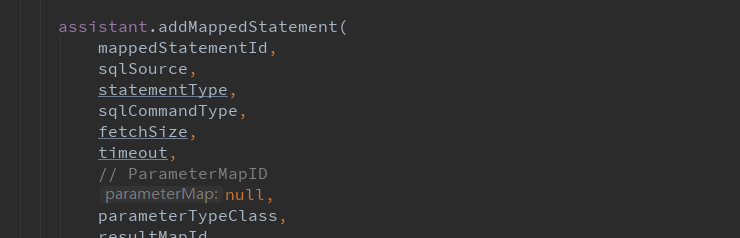


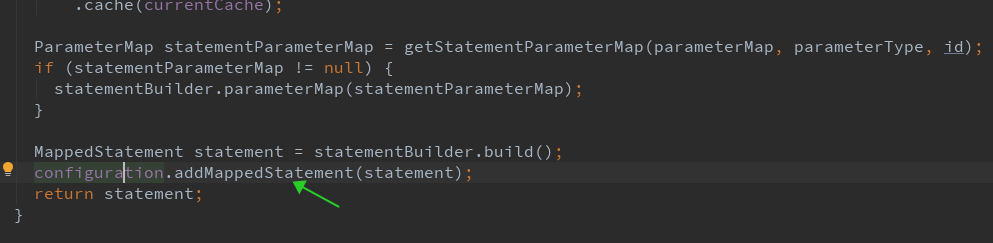
可以看到，先处理该类被加载的标识信息，后在循环中是在遍历DAO接口中的方法，然后逐个解析

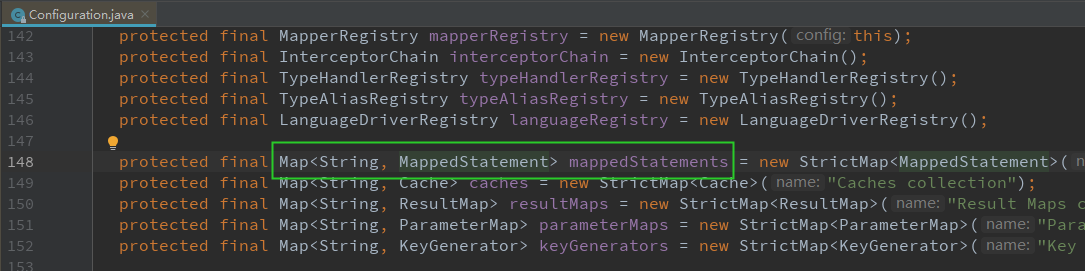


从注解中获取到SQL语句对象，sql执行类型（select,update…）,stementType,cache等信息

最终，将解析出来的信息，填加到了configuration对象的MappentStatements容器中

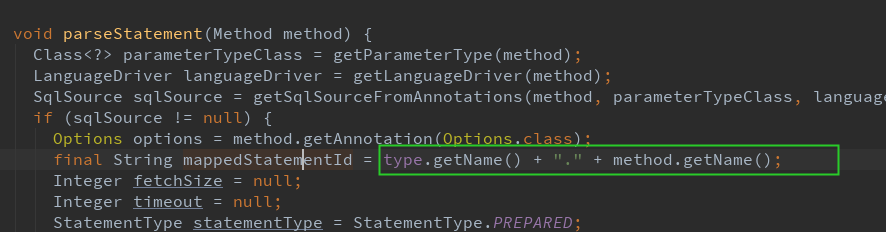




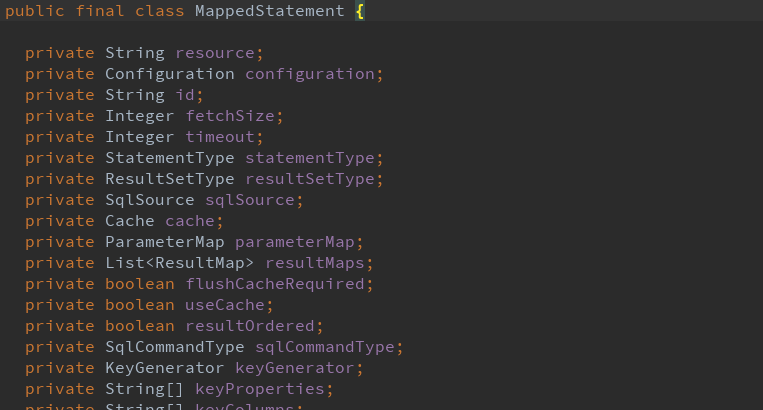


MappentStatements容器的key就是DAO接口的全限定名+方法名

例如：com.peony.ibatis.repository.BookDAO.selectList

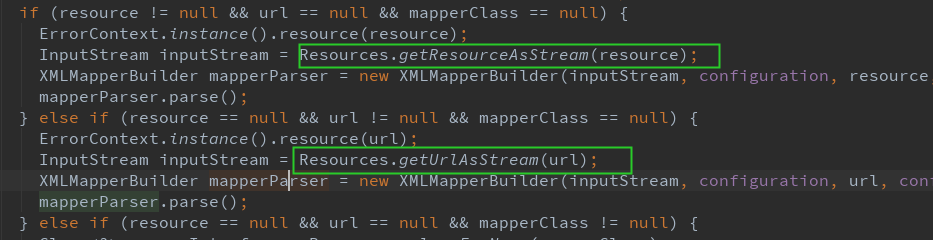


value就是该方法要执行的SQL信息对象MappedStatement,这是个高度抽象的类，一个mappedStatement对象对应到一个DAO接口的一个方法。



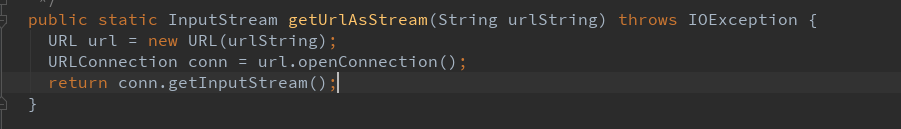
到了这里，我想你大概也明白了，在执行阶段，为什么可以通过接口的方法，就能执行一段SQL语句了（就是通过接口名+方法名，找到其对应的mappedStatement对象，获取到sql的，这在本书的这个位置还只是猜想，需要通过细度第6章节来验证）

### resource/url属性解析



首先说明, resource/url属性在获取资源是用了Resources工具类的不同方法，分别是

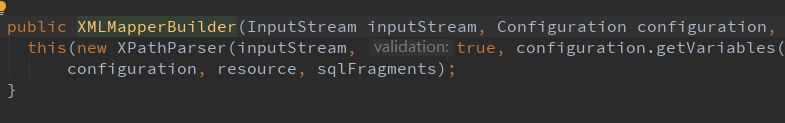
但是，获取到资源后，解析的过程是一致的，因此，可以归并解析，顺便看下通过一个url是怎么获取网络上的资源的

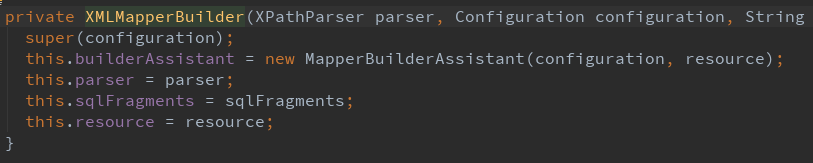


surprise！居然这么简单，是的，就是这么简单，请相信自己的眼睛，当然了，这也是阅读源码带给你的一点点惊喜, 拨开云雾见日的感觉是不是油然而生。

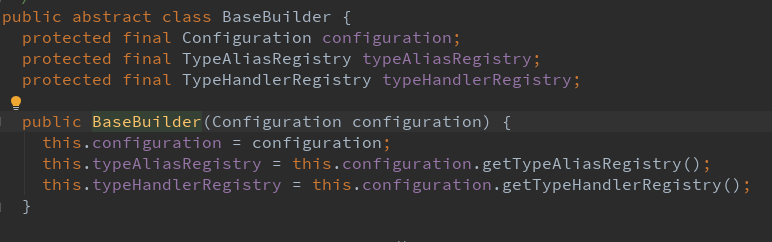
1. 获取XMLMapperBuilder实例

这里我想说一下，一个类重载很多中不同参数的构造方法，在类中封装一些参数的默认值，提供全参数的构造方法，也提供短参数甚至无参的构造方法，调用无参或者短参的构造方法时，内部最终都调用参数最多的终极构造器，这是一种简单却又很考验人的设计思想，它能体现你对一种事物的抽象能力和对面向对象编程的本质的认识

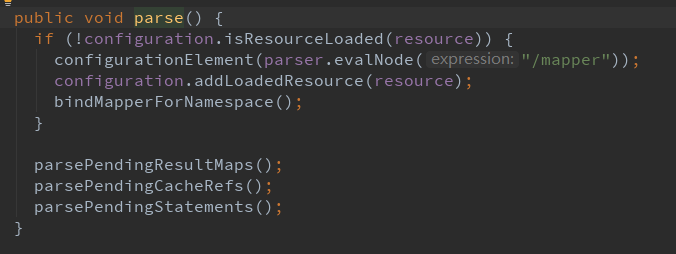




实例化了MapperBuilderAssistant实例（Mapper构建助手）

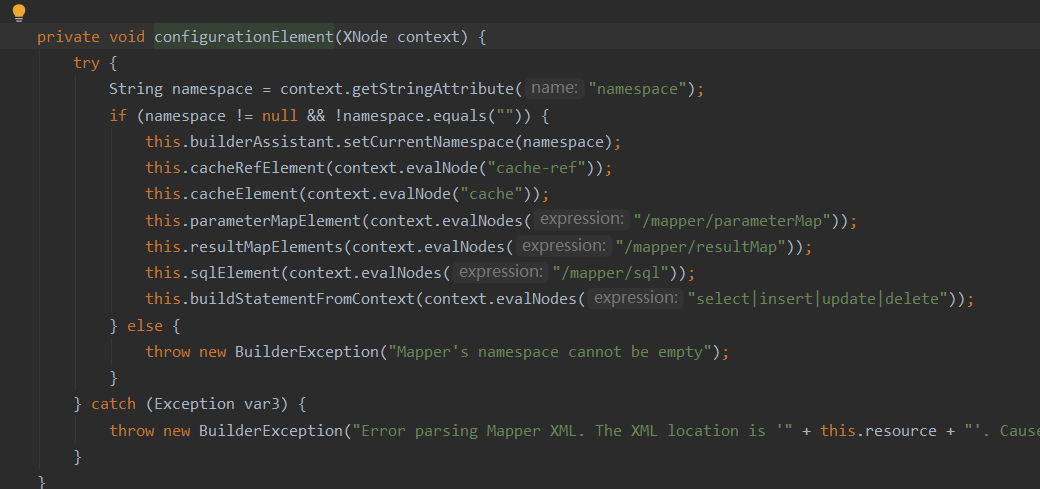


1. 解析Mapper标签



首选判断是否加载过，再进行加载，加载完后再标记已加载

看看configurationElement()方法，分别解析了mapper.xml文件中的各种一级标签,代码自解释性非常强



可以看到namespace如果为空，则会抛出BuilderException异常

# 获取SqlSession实例

# 获取代理的Mapper(DAO)接口实例

# 执行SQL语句

# 参数解析及动态SQL计算

# 结果集映射封装