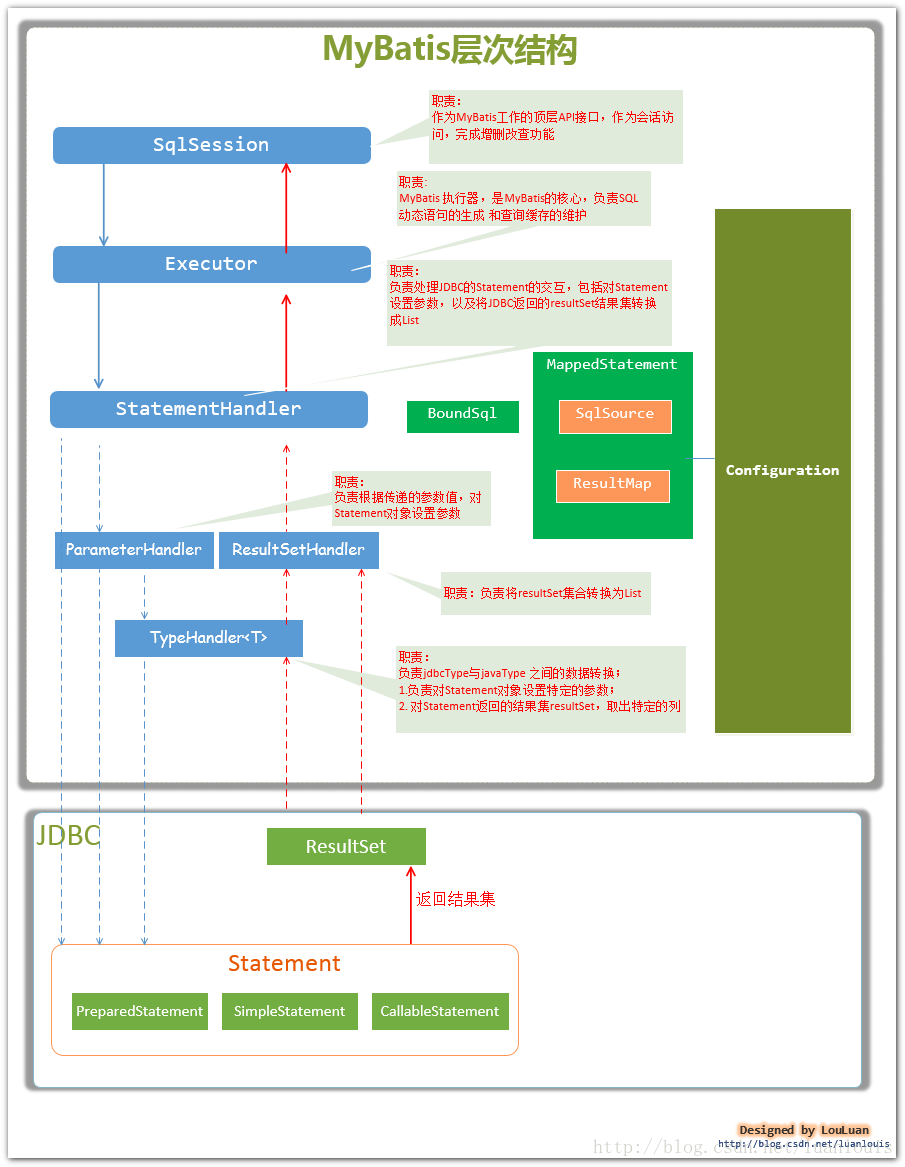
**层次结构**



public class Main {

public static void main(String[] args) throws IOException {

String resource = "mybatis.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

SqlSession sqlSession = sqlSessionFactory.openSession();

try {

ProductMapper productMapper = sqlSession.getMapper(ProductMapper.class);

List<Product> productList = productMapper.selectProductList();

for (Product product : productList) {

System.out.printf(product.toString());

}

} finally {

sqlSession.close();

}

}

}

**MyBatis的初始化**

1. **获取配置文件**

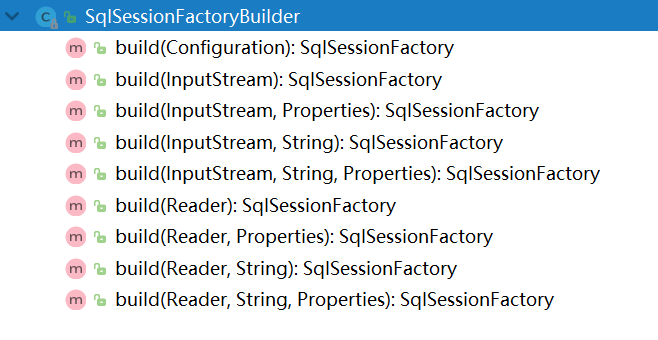
当系统初始化时，首先会**读取配置文件，并将其解析成InputStream**。

String resource = "**mybatis.xml**";

InputStream inputStream = Resources.getResourceAsStream(resource);

1. **创建SqlSessionFactoryBuilder对象**

**SqlSessionFactoryBuilder是用来创建SqlSessionFactory对象的**。



SqlSessionFactoryBuilder中只有一些**重载的build函数，这些build函数的入参都是MyBatis配置文件的输入流，返回值都是SqlSessionFactory**；由此可见，SqlSessionFactoryBuilder的作用就是通过配置文件创建SqlSessionFactory对象的。

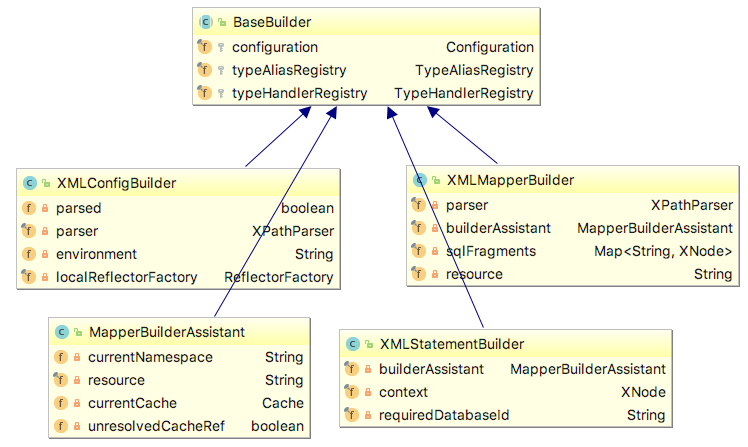
1. **SqlSessionFactory创建过程**

下面具体介绍build函数是如何创建SqlSessionFactory对象的。

**public** SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {  
 **try** {  
 XMLConfigBuilder parser = **new** XMLConfigBuilder(inputStream, environment, properties);  
 **return** build(parser.parse());  
 } **catch** (Exception e) {  
 **throw** ExceptionFactory.*wrapException*(**"Error building SqlSession."**, e);  
 } **finally** {  
 ErrorContext.*instance*().reset();  
 **try** {  
 inputStream.close();  
 } **catch** (IOException e) {  
 *// Intentionally ignore. Prefer previous error.* }  
 }  
}

* 1. **构造XMLConfigBuilder对象**

build函数首先会构造一个XMLConfigBuilder对象，该对象是用来解析XML配置文件的。下图为XMLConfigBuilder的体系结构：



* **XMLConfigBuilder是用来解析XML配置文件**的，不同类型XMLxxxBuilder用来解析MyBatis配置文件的不同部位。比如：XMLConfigBuilder用来解析MyBatis的配置文件，XMLMapperBuilder用来解析MyBatis中的映射文件（如上文提到的ProductMapper.xml），XMLStatementBuilder用来解析映射文件中的SQL语句。
* 这些XMLxxxBuilder都有一个共同的父类——BaseBuilder。**这个父类维护了一个全局的Configuration对象，MyBatis的配置文件解析后就以Configuration对象的形式存储**。
* 当创建XMLConfigBuilder对象时，就会初始化Configuration对象，并且在初始化Configuration对象的时候，一些别名会被注册到Configuration的typeAliasRegistry容器中。

**private** XMLConfigBuilder(XPathParser parser,

String environment, Properties props) {  
 **super**(**new** Configuration());  
 ErrorContext.*instance*().resource(**"SQL Mapper Configuration"**);  
 **this**.**configuration**.setVariables(props);  
 **this**.**parsed** = **false**;  
 **this**.**environment** = environment;  
 **this**.**parser** = parser;  
}

**public** Configuration() {  
 **typeAliasRegistry**.registerAlias(**"JDBC"**, JdbcTransactionFactory.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"MANAGED"**, ManagedTransactionFactory.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"JNDI"**, JndiDataSourceFactory.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"POOLED"**, PooledDataSourceFactory.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"UNPOOLED"**, UnpooledDataSourceFactory.**class**.getName());  
  
 **typeAliasRegistry**.registerAlias(**"PERPETUAL"**, PerpetualCache.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"FIFO"**, FifoCache.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"LRU"**, LruCache.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"SOFT"**, SoftCache.**class**.getName());  
 **typeAliasRegistry**.registerAlias(**"WEAK"**, WeakCache.**class**.getName());  
  
 **typeAliasRegistry**.registerAlias(**"VENDOR"**, VendorDatabaseIdProvider.**class**.getName());  
}

* 1. **解析配置文件**

有了XMLConfigBuilder对象之后，接下来就可以用它来解析配置文件了。

**private void** parseConfiguration(XNode root) {  
 **try** {  
 *// 解析<properties>节点*

propertiesElement(root.evalNode(**"properties"**));

*// 解析<typeAliases>节点*

typeAliasesElement(root.evalNode(**"typeAliases"**));  
 pluginElement(root.evalNode(**"plugins"**));  
 objectFactoryElement(root.evalNode(**"objectFactory"**));  
 objectWrapperFactoryElement(root.evalNode(**"objectWrapperFactory"**));  
 settingsElement(root.evalNode(**"settings"**));  
 environmentsElement(root.evalNode(**"environments"**));  
 databaseIdProviderElement(root.evalNode(**"databaseIdProvider"**));  
 typeHandlerElement(root.evalNode(**"typeHandlers"**));  
 mapperElement(root.evalNode(**"mappers"**));  
 } **catch** (Exception e) {  
 **throw new** BuilderException(**"Error parsing SQL Mapper Configuration. Cause: "** + e, e);  
 }  
}

由上述代码可以看到，XMLConfigBuilder会一次解析配置文件中的<properties>、<typeAliases>、<plugins>、<mappers>等属性。下面介绍几个重要属性的解析过程。

* + 1. <properties>节点的解析过程

**private void** propertiesElement(XNode context) **throws** Exception {  
 **if** (context != **null**) {  
 *// 获取<properties>节点的所有子节点*

Properties defaults = context.getChildrenAsProperties();  
 *// 获取<properties>节点上的resources属性*

String resource = context.getStringAttribute(**"resource"**);  
 *// 获取<properties>节点上的url属性*

String url = context.getStringAttribute(**"url"**);

*// resource和url不能同时存在*  
 **if** (resource != **null** && url != **null**) {  
 **throw new** BuilderException(**"The properties element cannot specify both a URL and a resource based property file reference. Please specify one or the other."**);  
 }  
 **if** (resource != **null**) {

*// 获取resource属性值对应的properties文件中的键值对，并添加至defaults容器中*  
 defaults.putAll(Resources.*getResourceAsProperties*(resource));  
 } **else if** (url != **null**) {

*// 获取url属性值对应的properties文件中的键值对，并添加至defaults容器中*  
 defaults.putAll(Resources.*getUrlAsProperties*(url));  
 }

*// 获取configuration中原本的属性，并添加至defaults容器中*  
 Properties vars = **configuration**.getVariables();  
 **if** (vars != **null**) {  
 defaults.putAll(vars);  
 }  
 **parser**.setVariables(defaults);

*// 将default容器添加至configuration中*  
 **configuration**.setVariables(defaults);  
 }  
}

首先读取<resources>节点下的所有<resource>节点，并将每个节点的name和value属性存入Properties中；然后读取<resources>节点上的resource、url属性，并获取指定配置文件中的name和value，也存入Properties中（如果resource节点上定义的属性和properties文件中的属性重名，那么properties文件中的属性值会覆盖resource节点上定义的属性值）；最终，**携带所有属性的Properties对象会被存储在Configuration对象中**。

* + 1. <settings>节点的解析过程

解析过程和<properties>属性极为相似，最终，**所有的settings属性都被存储在Configuration对象中**。

* + 1. <typeAliases>属性的解析过程

<typeAliases>属性的定义方式有两种：

方式一：

<typeAliases>

<typeAlias alias="Author" type="domain.blog.Author"/>

<typeAlias alias="Blog" type="domain.blog.Blog"/>

</typeAliases>

方式二：

<typeAliases>

<package name="domain.blog"/>

</typeAliases>

采用这种方式时，MyBatis会为指定包下的所有类起一个别名，该别名为首字母小写的类名。

**private void** typeAliasesElement(XNode parent) {  
 **if** (parent != **null**) {  
 *// 遍历<typeAliases>下所有子节点*

**for** (XNode child : parent.getChildren()) {

*// 若当前节点为<package>*  
 **if** (**"package"**.equals(child.getName())) {

*// 获取<package>上的name属性（包名）*  
 String typeAliasPackage = child.getStringAttribute(**"name"**);  
 *// 为该包下的所有类起个别名，并注册进configuration的typeAliasRegist ry中*

**configuration**.getTypeAliasRegistry().registerAliases(typeAliasPackage);

*// 如果当前结点为<typeAlias>*  
 } **else** {

*// 获取alias和type属性*  
 String alias = child.getStringAttribute(**"alias"**);  
 String type = child.getStringAttribute(**"type"**);

*// 注册进configuration的typeAliasRegistry中*  
 **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);  
 }  
 }  
 }  
 }  
}

这些**别名都会被存入configuration的typeAliasRegistry容器中**。

* + 1. <mappers>节点的解析过程

<mappers>节点的定义方式有四种：

方式一：

<mappers>

<package name="org.mybatis.builder"/>

</mappers>

方式二：

<mappers>

<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>

</mappers>

方式三：

<mappers>

<mapper url="file:///var/mappers/AuthorMapper.xml"/>

</mappers>

方式四：

<mappers>

<mapper class="org.mybatis.builder.AuthorMapper"/>

</mappers>

**private void** mapperElement(XNode parent) **throws** Exception {  
 **if** (parent != **null**) {

*// 遍历<mappers>下所有子节点*  
 **for** (XNode child : parent.getChildren()) {

*// 如果当前节点为<package>*  
 **if** (**"package"**.equals(child.getName())) {

*// 获取<package>的name属性（该属性值为mapper class所在的包名）*  
 String mapperPackage = child.getStringAttribute(**"name"**);

*// 将该包下的所有Mapper Class注册到configuration的mapperRegistry容器中*  
 **configuration**.addMappers(mapperPackage);

*// 如果当前节点为<mapper>*  
 } **else** {

*// 依次获取resource、url、class属性*  
 String resource = child.getStringAttribute(**"resource"**);  
 String url = child.getStringAttribute(**"url"**);  
 String mapperClass = child.getStringAttribute(**"class"**);

*// 解析resource属性（Mapper.xml文件的路径）*  
 **if** (resource != **null** && url == **null** && mapperClass == **null**) {  
 ErrorContext.*instance*().resource(resource);

*// 将Mapper.xml文件解析成输入流*  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);

*// 使用XMLMapperBuilder解析Mapper.xml，并将Mapper Class注册进configuration对向的mapperRegistry容器中*  
 XMLMapperBuilder mapperParser = **new** XMLMapperBuilder(inputStream, **configuration**, resource, **configuration**.getSqlFragments());  
 mapperParser.parse();

*// 解析url属性（Mapper.xml文件的路径）*  
 } **else if** (resource == **null** && url != **null** && mapperClass == **null**) {  
 ErrorContext.*instance*().resource(url);  
 InputStream inputStream = Resources.*getUrlAsStream*(url);  
 XMLMapperBuilder mapperParser = **new** XMLMapperBuilder(inputStream, **configuration**, url, **configuration**.getSqlFragments());  
 mapperParser.parse();

*// 解析class属性（Mapper Class的全限定名）*  
 } **else if** (resource == **null** && url == **null** && mapperClass != **null**) {

*// 将Mapper Class的全限定名转化为Class对象*  
 Class<?> mapperInterface = Resources.*classForName*(mapperClass);

*// 注册进configuration对象的mapperRegistry容器中*  
 **configuration**.addMapper(mapperInterface);  
 } **else** {  
 **throw new** BuilderException(**"A mapper element may only specify a url, resource or class, but not more than one."**);  
 }  
 }  
 }  
 }  
}

MyBatis会遍历<mappers>下所有的子节点，如果当前遍历到的节点是<package>，则MyBatis会将该包下的所有Mapper Class注册到configuration的mapperRegistry容器中；如果当前节点为<mapper>，则会依次湖区resource、url、class属性，解析映射文件，并将映射文件对应的Mapper Class注册到configuration的mapperRegistry容器中。

其中<mapper>节点的解析过程如下：

XMLMapperBuilder mapperParser = **new** XMLMapperBuilder(inputStream,

**configuration**,url,**configuration**.getSqlFragments());  
 mapperParser.parse();

在解析前，首先需要创建XMLMapperBuilder，创建过程如下：

**private** XMLMapperBuilder(XPathParser parser, Configuration configuration, String resource, Map<String, XNode> sqlFragments) {

*// 将configuration赋给BaseBuilder*  
 **super**(configuration);

*// 创建MapperBuilderAssistant对象（该对象为MapperBuilder的协助者）*  
 **this**.**builderAssistant** = **new** MapperBuilderAssistant(configuration, resource);  
 **this**.**parser** = parser;  
 **this**.**sqlFragments** = sqlFragments;  
 **this**.**resource** = resource;  
}

首先会初始化父类BaseBuilder，并将configuration赋给BaseBuilder；然后创建MapperBuilderAssistant对象，该对象为XMLMapperBuilder的协助者，用来协助XMLMapperBuilder完成一些解析映射文件的动作。

有了XMLMapperBuilder后，便可进入解析<mapper>的过程：

**public void** parse() {

*// 若当前的Mapper.xml尚未被解析，则开始解析*

*// 若<mappers>节点下有相同的<mapper>节点，那么就无需再次解析了*  
 **if** (!**configuration**.isResourceLoaded(**resource**)) {

*// 解析<mapper>节点*  
 configurationElement(**parser**.evalNode(**"/mapper"**));

*// 将该Mapper.xml添加至configuration的LoadedResource容器中，下回无需再解析*  
 **configuration**.addLoadedResource(**resource**);

*// 将该Mapper.xml对应的Mapper Class注册进configuration的mapperRegistry容器中*  
 bindMapperForNamespace();  
 }  
  
 parsePendingResultMaps();  
 parsePendingChacheRefs();  
 parsePendingStatements();  
}

* 1. 创建SqlSessionFactory对象

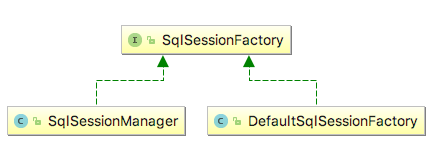
**public** SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {  
 **try** {  
 XMLConfigBuilder parser = **new** XMLConfigBuilder(inputStream, environment, properties);  
 **return** build(parser.parse());  
 } **catch** (Exception e) {  
 **throw** ExceptionFactory.*wrapException*(**"Error building SqlSession."**, e);  
 } **finally** {  
 ErrorContext.*instance*().reset();  
 **try** {  
 inputStream.close();  
 } **catch** (IOException e) {  
 *// Intentionally ignore. Prefer previous error.* }  
 }  
}

回过头再来看一下SqlSessionFactory的build函数，上面介绍了XMLConfigBuilder解析映射文件的过程，解析完成之后parser.parse()函数会返回一个包含了映射文件解析结果的configuration对象，紧接着，这个对象将作为参数传递给另一个build函数，如下：

**public** SqlSessionFactory build(Configuration config) {  
 **return new** DefaultSqlSessionFactory(config);  
}

这个函数将configuration作为参数，创建了DefaultSqlSessionFactory对象。

DefaultSqlSessionFactory是接口SqlSessionFactory的一个实现类，SqlSessionFactory的体系结构如下：



此时，SqlSessionFactory创建完毕。

初始化的过程就是**解析配置文件**和**初始化Configuration**的过程，MyBatis的初始化过程可用以下几行代码来表述：

String resource = "mybatis.xml";

// 加载mybatis的配置文件（它也加载关联的映射文件）

InputStream inputStream = null;

try {

inputStream = Resources.getResourceAsStream(resource);

} catch (IOException e) {

e.printStackTrace();

}

// 构建sqlSession的工厂

sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

首先会**创建SqlSessionFactoryBuilder建造者对象**，然后**由它进行创建SqlSessionFactory**。这里用到的是**建造者模式**，建造者模式最简单的理解就是不手动new对象，而是由其他类来进行对象的创建。

*// SqlSessionFactoryBuilder类*

**public** SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {  
 **try** {  
 XMLConfigBuilder parser = **new** XMLConfigBuilder(inputStream, environment, properties);  
 **return** build(parser.parse()); *// 开始进行解析了。*  
 } **catch** (Exception e) {  
 **throw** ExceptionFactory.*wrapException*(**"Error building SqlSession."**, e);  
 } **finally** {  
 ErrorContext.*instance*().reset();  
 **try** {  
 inputStream.close();  
 } **catch** (IOException e) {  
 *// Intentionally ignore. Prefer previous error.* }  
 }  
}

**XMLConfigBuilder对象会进行XML配置文件的解析，实际为configuration节点的解析操作**。

*// XMLConfigBuilder类*

**public** Configuration parse() {  
 **if** (**parsed**) {  
 **throw new** BuilderException(**"Each MapperConfigParser can only be used once."**);  
 }  
 **parsed** = **true**;  
 parseConfiguration(**parser**.evalNode(**"/configuration"**));  
 **return configuration**;  
}  
  
**private void** parseConfiguration(XNode root) {  
 **try** {  
 propertiesElement(root.evalNode(**"properties"**)); *//issue #117 read properties first* typeAliasesElement(root.evalNode(**"typeAliases"**));  
 pluginElement(root.evalNode(**"plugins"**));  
 objectFactoryElement(root.evalNode(**"objectFactory"**));  
 objectWrapperFactoryElement(root.evalNode(**"objectWrapperFactory"**));  
 settingsElement(root.evalNode(**"settings"**));

*// 处理environments节点数据*  
 environmentsElement(root.evalNode(**"environments"**));  
 databaseIdProviderElement(root.evalNode(**"databaseIdProvider"**));  
 typeHandlerElement(root.evalNode(**"typeHandlers"**));  
 mapperElement(root.evalNode(**"mappers"**));  
 } **catch** (Exception e) {  
 **throw new** BuilderException(**"Error parsing SQL Mapper Configuration. Cause: "** + e, e);  
 }  
}

在configuration节点下会一次解析properties/settings/.../mappers等节点配置。在解析environments节点时，会根据transactionManager的配置来创建事务管理器，根据dataSource的配置来创建DataSource对象，这里面包含了数据库登录的相关信息。在解析mappers节点时，会读取该节点下所有的mapper文件，然后进行解析，并将解析后的结果存到configuration对象中。

*// XMLConfigBuilder类*

**private void** environmentsElement(XNode context) **throws** Exception {  
 **if** (context != **null**) {  
 **if** (**environment** == **null**) {  
 **environment** = context.getStringAttribute(**"default"**);  
 }  
 **for** (XNode child : context.getChildren()) {  
 String id = child.getStringAttribute(**"id"**);  
 **if** (isSpecifiedEnvironment(id)) {

*// 创建事务管理器*  
 TransactionFactory txFactory = transactionManagerElement(child.evalNode(**"transactionManager"**));  
 DataSourceFactory dsFactory = dataSourceElement(child.evalNode(**"dataSource"**));  
 DataSource dataSource = dsFactory.getDataSource();

*// 建造者模式，设计模式*  
 Environment.Builder environmentBuilder = **new** Environment.Builder(id)  
 .transactionFactory(txFactory)  
 .dataSource(dataSource);  
 **configuration**.setEnvironment(environmentBuilder.build());  
 }  
 }  
 }  
}

*// 解析单独的mapper文件*

**private void** mapperElement(XNode parent) **throws** Exception {  
 **if** (parent != **null**) {  
 **for** (XNode child : parent.getChildren()) {  
 **if** (**"package"**.equals(child.getName())) {  
 String mapperPackage = child.getStringAttribute(**"name"**);  
 **configuration**.addMappers(mapperPackage);  
 } **else** {  
 String resource = child.getStringAttribute(**"resource"**);  
 String url = child.getStringAttribute(**"url"**);  
 String mapperClass = child.getStringAttribute(**"class"**);  
 **if** (resource != **null** && url == **null** && mapperClass == **null**) {  
 ErrorContext.*instance*().resource(resource);  
 InputStream inputStream = Resources.*getResourceAsStream*(resource);  
 XMLMapperBuilder mapperParser = **new** XMLMapperBuilder(inputStream, **configuration**, resource, **configuration**.getSqlFragments());  
 mapperParser.parse(); *// 开始解析mapper文件了*  
 } **else if** (resource == **null** && url != **null** && mapperClass == **null**) {  
 ErrorContext.*instance*().resource(url);  
 InputStream inputStream = Resources.*getUrlAsStream*(url);  
 XMLMapperBuilder mapperParser = **new** XMLMapperBuilder(inputStream, **configuration**, url, **configuration**.getSqlFragments());  
 mapperParser.parse();  
 } **else if** (resource == **null** && url == **null** && mapperClass != **null**) {  
 Class<?> mapperInterface = Resources.*classForName*(mapperClass);  
 **configuration**.addMapper(mapperInterface);  
 } **else** {  
 **throw new** BuilderException(**"A mapper element may only specify a url, resource or class, but not more than one."**);  
 }  
 }  
 }  
 }  
}

解析完MyBatis配置文件后，configuration就初始化完成了，然后根据configuration对象来创建SqlSession，到这里时，MyBatis的初始化的征程已经走完了。

*// SqlSessionFactoryBuilder类*

**public** SqlSessionFactory build(Configuration config) {  
 **return new** DefaultSqlSessionFactory(config);  
}