

目录

1. MyBatis 流程概述.....	2
2. 第一阶段：配置加载阶段.....	2
2.1 建造者模式.....	2
2.1.1 什么是建造者模式.....	2
2.1.2 与工厂模式区别.....	3
2.2 配置加载的核心类.....	4
2.2.1 建造器三个核心类.....	4
2.2.2 关于 Configuration 对象.....	4
2.3 配置加载过程.....	5
3. 第二阶段：代理封装阶段.....	8
3.1 Mybatis 的接口层.....	9
3.1.1 SqlSession.....	9
3.1.2 策略模式.....	10
3.1.3 SqlSessionFactory.....	10
3.2 binding 模块分析.....	11
3.2.1 binding 模块核心类.....	11
3.2.2 binding 模块运行流程.....	12
4. 第三个阶段：数据访问阶段.....	13
4.1 关于 Executor 组件.....	13
4.2 Executor 中的模板模式.....	13
4.3 Executor 的三个重要小弟.....	15
4.4 关于 StatementHandler.....	16
4.5 关于 ResultHandler.....	17

1. MyBatis 流程概述

通过对快速入门代码的分析，可以把 MyBatis 的运行流程分为三大阶段：

1. 初始化阶段：读取 XML 配置文件和注解中的配置信息，创建配置对象，并完成各个模块的初始化的工作；
2. 代理封装阶段：封装 iBatis 的编程模型，使用 mapper 接口开发的初始化工作；
3. 数据访问阶段：通过 SqlSession 完成 SQL 的解析，参数的映射、SQL 的执行、结果的解析过程；

快速入门代码阶段划分图示：

```
@Before
public void init() throws IOException {
    //-----第一阶段-----
    // 1. 读取mybatis配置文件创SqlSessionFactory
    String resource = "mybatis-config.xml";
    InputStream inputStream = Resources.getResourceAsStream(resource);
    // 1. 读取mybatis配置文件创SqlSessionFactory
    sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
    inputStream.close();
}

@Test
// 快速入门
public void quickStart() throws IOException {
    //-----第二阶段-----
    // 2. 获取sqlSession
    SqlSession sqlSession = sqlSessionFactory.openSession();
    // 3. 获取对应mapper
    TUserMapper mapper = sqlSession.getMapper(TUserMapper.class);

    //-----第三阶段-----
    // 4. 执行查询语句并返回单条数据
    TUser user = mapper.selectByPrimaryKey(2);
    System.out.println(user);

    System.out.println("-----");

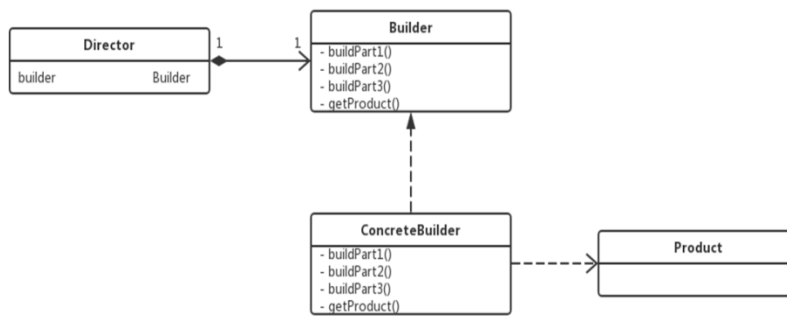
    // 5. 执行查询语句并返回多条数据
    List<TUser> users = mapper.selectAll();
    for (TUser tUser : users) {
        System.out.println(tUser);
    }
}
```

2. 第一阶段：配置加载阶段

2.1 建造者模式

2.1.1 什么是建造者模式

在配置加载阶段大量的使用了建造者模式，首先学习建造者模式。建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。建造者模式类图如下：



各要素如下：

- ✓ **Product**：要创建的复杂对象
- ✓ **Builder**：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建；
- ✓ **ConcreteBuilder**：实现 **Builder** 接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。在建造过程完成后，提供产品的实例；
- ✓ **Director**：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建；

应用举例：红包的创建是个复杂的过程，使用建造起模式进行优化，代码示例：
com.enjoylearning.mybatis.build.Director；

关于建造器模式的扩展知识：流式编程风格越来越流行，如 zookeeper 的 Curator、JDK8 的流式编程等等都是例子。流式编程的优点在于代码编程性更高、可读性更好，缺点在于对程序员编码要求更高、不太利于调试。建造者模式是实现流式编程风格的一种方式；

2.1.2 与工厂模式区别

建造者模式应用场景如下：

- ✓ 需要生成的对象具有复杂的内部结构，实例化对象时要屏蔽掉对象内部的细节，让上层代码与复杂对象的实例化过程解耦，可以使用建造者模式；简而言之，如果“遇到多个构造器参数时要考虑用构建器”；
- ✓ 对象的实例化是依赖各个组件的产生以及装配顺序，关注的是一步一步地组装出目标对象，可以使用建造器模式；

建造者模式与工程模式的区别在于：

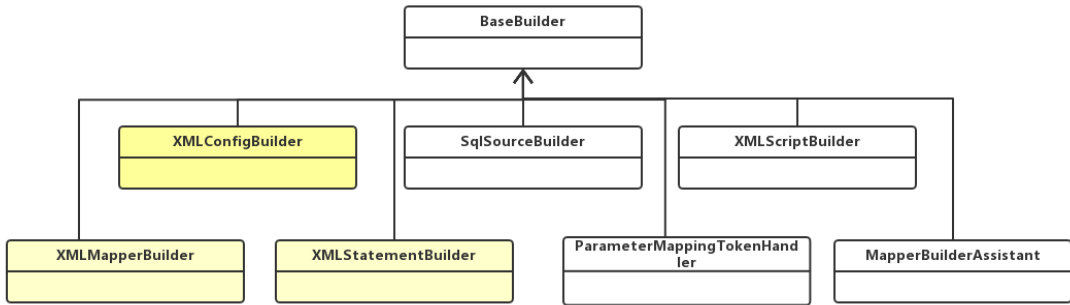
设计模式	形象比喻	对象复杂度	客户端参与程度
工厂模式	生产大众版	关注的是一个产品整体，无须关心产品的各部分是如何创建出来的；	客户端对产品的创建过程参与度低，对象实例化时属性值相对比较固定；
建造者模式	生产定制版	建造的对象更加复杂，是一个复合产品，它由各个部件复合而成，部件不同产品对	客户端参与了产品的创建，决定了产品的类型和内容，参与度高；适合实例化对象时属性

		象不同，生成的产品粒度细；	变化频繁的场景；
--	--	---------------	----------

2.2 配置加载的核心类

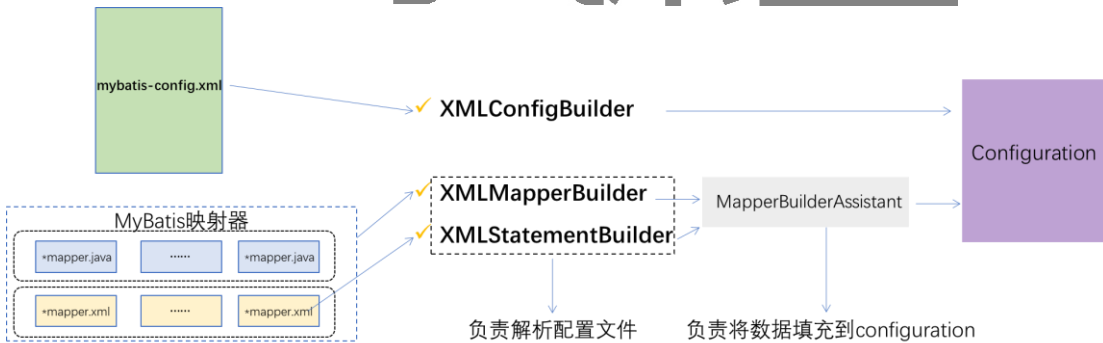
2.2.1 建造器三个核心类

在 MyBatis 中负责加载配置文件的核心类有三个，类图如下：



- ✓ **BaseBuilder**：所有解析器的父类，包含配置文件实例，为解析文件提供一些通用的方法；
- ✓ **XMLConfigBuilder**：主要负责解析 mybatis-config.xml；
- ✓ **XMLMapperBuilder**：主要负责解析映射配置 Mapper.xml 文件；
- ✓ **XMLStatementBuilder**：主要负责解析映射配置文件中的 SQL 节点；

XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuilder 这三个类在配置文件加载过程中非常重要，具体分工如下图所示：



这三个类使用了建造者模式对 configuration 对象进行初始化，但是没有使用建造者模式的“肉体”（流式编程风格），只用了灵魂（屏蔽复杂对象的创建过程），把建造者模式演绎成了工厂模式；后面还会对这三个类源码进行分析；

2.2.2 关于 Configuration 对象

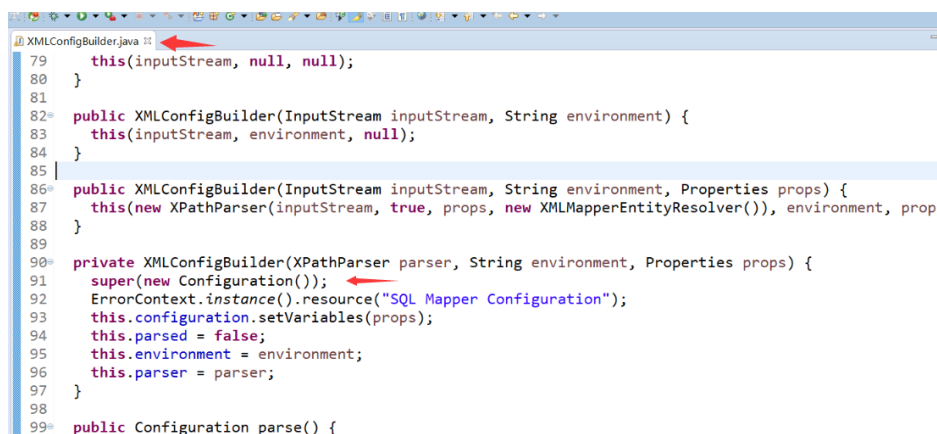
实例化并初始化 Configuration 对象是第一个阶段的最终目的，所以熟悉 configuration 对象是理解第一个阶段代码的核心；configuration 对象的关键属性解析如下：

- ✓ **MapperRegistry**：mapper 接口动态代理工厂类的注册中心。在 MyBatis 中，通过

mapperProxy 实现 InvocationHandler 接口，MapperProxyFactory 用于生成动态代理的实例对象；

- ✓ ResultMap: 用于解析 mapper.xml 文件中的 resultMap 节点，使用 ResultMapping 来封装 id, result 等子元素；
- ✓ MappedStatement: 用于存储 mapper.xml 文件中的 select、insert、update 和 delete 节点，同时还包含了这些节点的很多重要属性；
- ✓ SqlSource: 用于创建 BoundSql，mapper.xml 文件中的 sql 语句会被解析成 BoundSql 对象，经过解析 BoundSql 包含的语句最终仅仅包含？占位符，可以直接提交给数据库执行；

需要特别注意的是 Configuration 对象在 MyBatis 中是单例的，生命周期是应用级的，换句话说只要 MyBatis 运行 Configuration 对象就会独一无二的存在；在 MyBatis 中仅在 org.apache.ibatis.builder.xml.XMLConfigBuilder.XMLConfigBuilder(XPathParser, String, Properties) 中有实例化 configuration 对象的代码，如下图：

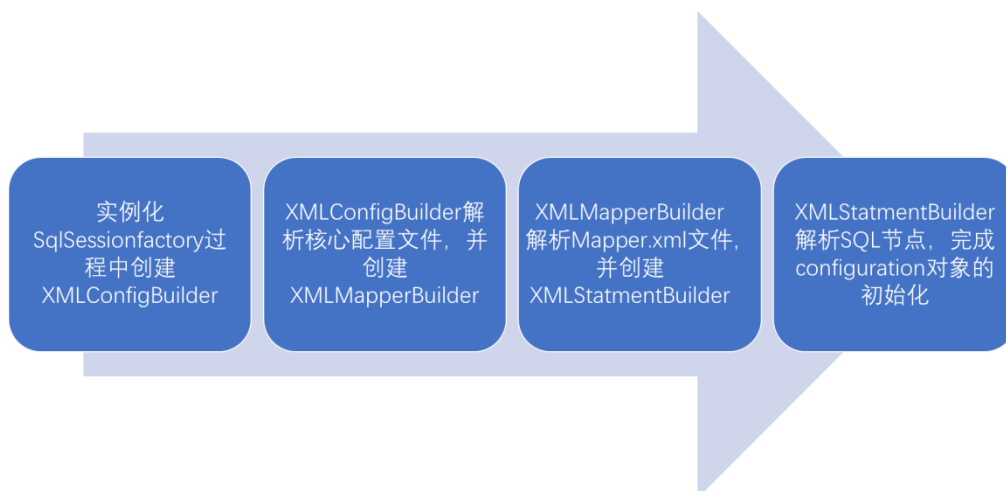


```
79     this(inputStream, null, null);
80 }
81
82 public XMLConfigBuilder(InputStream inputStream, String environment) {
83     this(inputStream, environment, null);
84 }
85
86 public XMLConfigBuilder(InputStream inputStream, String environment, Properties props) {
87     this(new XPathParser(inputStream, true, props, new XMLMapperEntityResolver()), environment, props);
88 }
89
90 private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {
91     super(new Configuration());
92     ErrorContext.instance().resource("SQL Mapper Configuration");
93     this.configuration.setVariables(props);
94     this.parsed = false;
95     this.environment = environment;
96     this.parser = parser;
97 }
98
99 public Configuration parse() {
```

Configuration 对象的初始化（属性复制），是在建造 SqlSessionFactory 的过程中进行的，接下来分析第一个阶段的内部流程：

2.3 配置加载过程

可以把第一个阶段配置加载过程分解为四个步骤，四个步骤如下图：



第一步：通过 SqlSessionFactoryBuilder 建造 SqlSessionFactory，并创建 XMLConfigBuilder 对

象 读 取 MyBatis 核 心 配 置 文 件 ， 见 方 法 ：
org.apache.ibatis.session.SqlSessionFactoryBuilder.build(Reader, String, Properties)， 如下图：

```
2  */
3  public class SqlSessionFactoryBuilder {
4
5  public SqlSessionFactory build(Reader reader) {
6      return build(reader, null, null);
7  }
8
9  public SqlSessionFactory build(Reader reader, String environment) {
10     return build(reader, environment, null);
11 }
12
13 public SqlSessionFactory build(Reader reader, Properties properties) {
14     return build(reader, null, properties);
15 }
16
17 public SqlSessionFactory build(Reader reader, String environment, Properties properties) {
18     try {
19         // 读取配置文件
20         XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);
21         return build(parser.parse()); // 解析配置文件得到configuration对象，并返回SqlSessionFactory
22     } catch (Exception e) {
23         throw ExceptionFactory.wrapException("Error building SqlSession.", e);
24     } finally {
25         ErrorContext.instance().reset();
26         try {
27             reader.close();
28         } catch (IOException e) {
29             // Intentionally ignore. Prefer previous error.
30         }
31     }
32 }
33 }
```

第二步：进入 XMLConfigBuilder 的 parseConfiguration 方法，对 MyBatis 核心配置文件的各个元素进行解析，读取元素信息后填充到 configuration 对象。在 XMLConfigBuilder 的 mapperElement () 方法中通过 XMLMapperBuilder 读取所有 mapper.xml 文件；见方法：org.apache.ibatis.builder.xml.XMLConfigBuilder.parseConfiguration(XNode)； 见下图：

```

XMLConfigBuilder.java 28 SqlSessionFactoryBuilder.java XMLMapperBuilder.java
98
99 public Configuration parse() {
100     if (parsed) {
101         throw new BuilderException("Each XMLConfigBuilder can only be used once.");
102     }
103     parsed = true;
104     parseConfiguration(parser.evalNode("/configuration"));
105     return configuration;
106 }
107
108 private void parseConfiguration(XNode root) {
109     try {
110         //issue #117 read properties first
111         //解析<properties>节点
112         propertiesElement(root.evalNode("properties"));
113         //解析<settings>节点
114         Properties settings = settingsAsProperties(root.evalNode("settings"));
115         loadCustomVfs(settings);
116         //解析<typeAliases>节点
117         typeAliasesElement(root.evalNode("typeAliases"));
118         //解析<plugins>节点
119         pluginElement(root.evalNode("plugins"));
120         //解析<objectFactory>节点
121         objectFactoryElement(root.evalNode("objectFactory"));
122         //解析<objectWrapperFactory>节点
123         objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));
124         //解析<reflectorFactory>节点
125         reflectorFactoryElement(root.evalNode("reflectorFactory"));
126         settingsElement(settings); //将settings填充到configuration
127         // read it after objectFactory and objectWrapperFactory issue #631
128         //解析<environments>节点
129         environmentsElement(root.evalNode("environments"));
130         //解析<databaseIdProvider>节点
131         databaseIdProviderElement(root.evalNode("databaseIdProvider"));
132         //解析<typeHandlers>节点
133         typeHandlerElement(root.evalNode("typeHandlers"));
134         //解析<mappers>节点
135         mapperElement(root.evalNode("mappers"));
136     } catch (Exception e) {
137         throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);
138     }
139 }

```

第三步：XMLMapperBuilder 的核心方法为 configurationElement (XNode)，该方法对 mapper.xml 配置文件的各个元素进行解析，读取元素信息后填充到 configuration 对象。如下图所示：

```

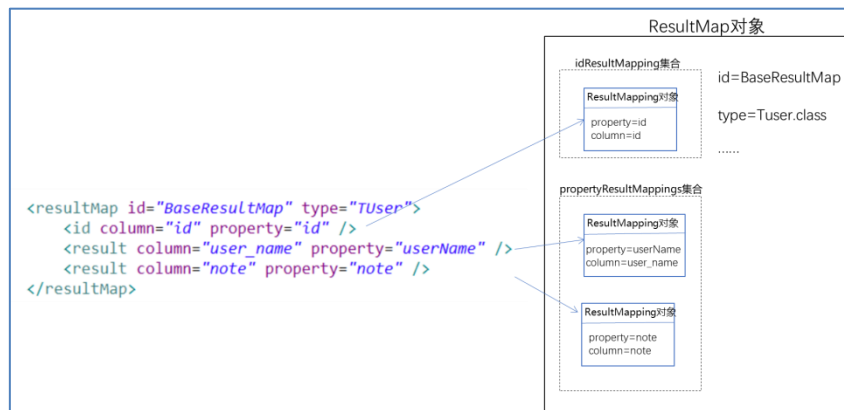
private void configurationElement(XNode context) {
    try {
        //获取mapper节点的namespace属性
        String namespace = context.getStringAttribute("namespace");
        if (namespace == null || namespace.equals("")) {
            throw new BuilderException("Mapper's namespace cannot be empty");
        }
        //设置builderAssistant的namespace属性
        builderAssistant.setCurrentNamespace(namespace);
        //解析cache-ref节点
        cacheRefElement(context.evalNode("cache-ref"));
        //重点分析：解析cache节点-----1-----
        cacheElement(context.evalNode("cache"));
        //解析parameterMap节点（已废弃）
        parameterMapElement(context.evalNodes("/mapper/parameterMap"));
        //重点分析：解析resultMap节点（基于数据结果去理解）-----2-----
        resultMapElements(context.evalNodes("/mapper/resultMap"));
        //解析sql节点
        sqlElement(context.evalNodes("/mapper/sql"));
        //重点分析：解析select、insert、update、delete节点-----3-----
        buildStatementFromContext(context.evalNodes("select|insert|update|delete"));
    } catch (Exception e) {
        throw new BuilderException("Error parsing Mapper XML. The XML location is '" + resource + "'. Cause: " + e, e);
    }
}

```

在 XMLMapperBuilder 解析过程中，有四个点需要注意：

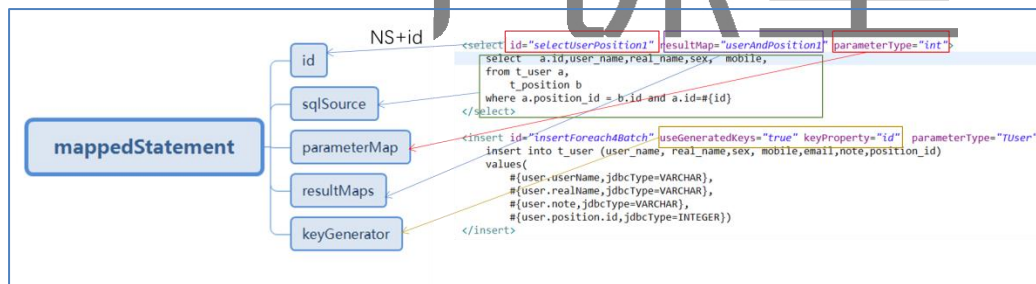
1. resultMapElements(List<XNode>)方法用于解析 resultMap 节点，这个方法非常重要，一定要跟源码理解；解析完之后数据保存在 configuration 对象的 resultMaps 属性中；

如下图所示：



2. XMLMapperBuilder 中在实例化二级缓存(见 `cacheElement(XNode)`)、实例化 `resultMap` (见 `resultMapElements(List<XNode>)`) 过程中都使用了建造者模式，而且是建造者模式的典型应用；
3. XMLMapperBuilder 和 XMLMapperStatementBuilder 有自己的“秘书” `MapperBuilderAssistant`。XMLMapperBuilder 和 XMLMapperStatementBuilder 负责解析读取配置文件里面的信息，`MapperBuilderAssistant` 负责将信息填充到 `configuration`。将文件解析和数据的填充的工作分离在不同的类中，符合单一职责原则；
4. 在 `buildStatementFromContext(List<XNode>)` 方法中，创建 `XMLStatmentBuilder` 解析 `Mapper.xml` 中 `select`、`insert`、`update`、`delete` 节点

第四步：在 `XMLStatmentBuilder` 的 `parseStatementNode()` 方法中，对 `Mapper.xml` 中 `select`、`insert`、`update`、`delete` 节点进行解析，并调用 `MapperBuilderAssistant` 负责将信息填充到 `configuration`。在理解 `parseStatementNod()` 方法之前，有必要了解 `MappedStatement`，这个类用于封装 `select`、`insert`、`update`、`delete` 节点的信息：如下图所示：



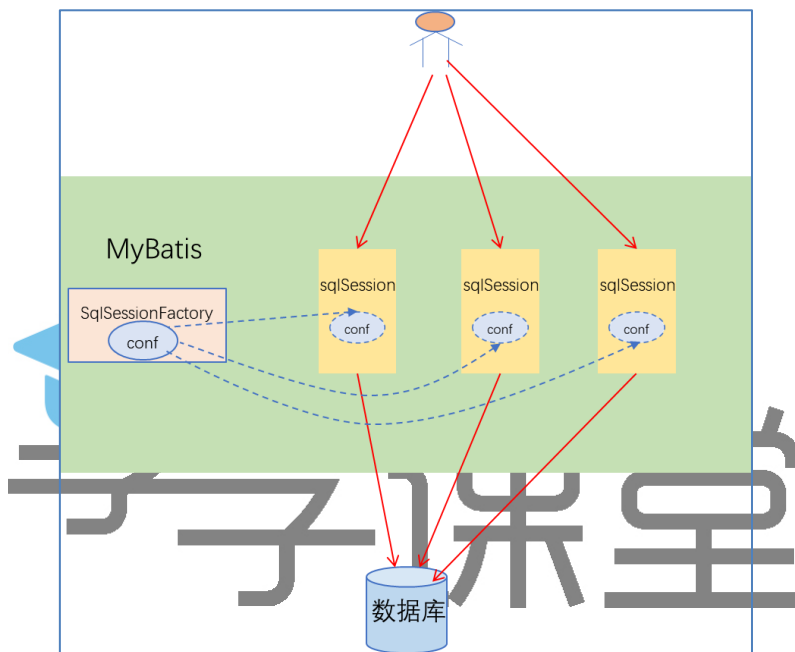
3. 第二阶段：代理封装阶段

第二个阶段是 `MyBatis` 最神秘的阶段，要理解它，就需要对 `Mybatis` 的接口层和 `binding` 模块数据源模块进行深入的学习；

3.1 Mybatis 的接口层

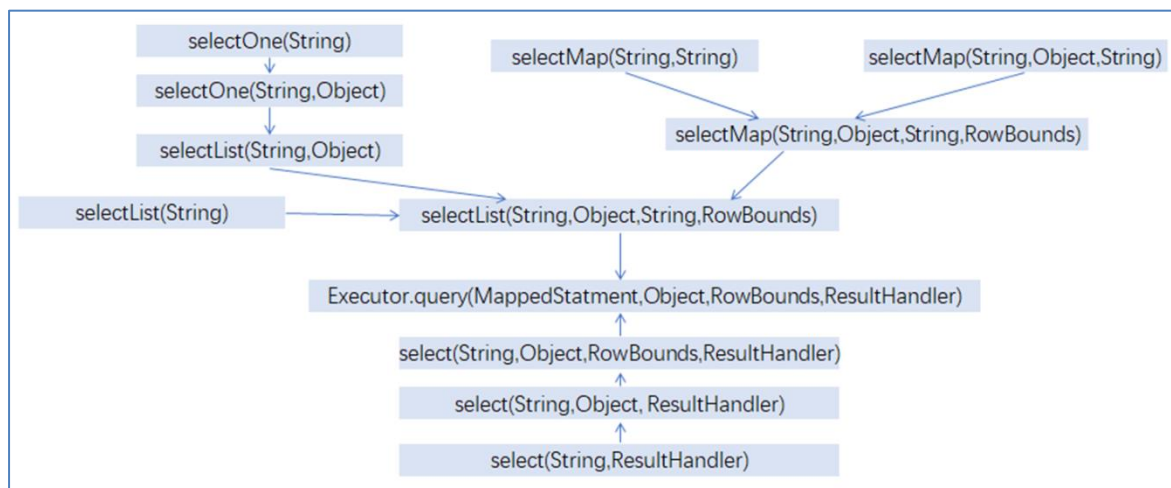
3.1.1 SqlSession

第二个阶段使用到的第一个对象就是 `SqlSession`，`SqlSession` 是 `MyBatis` 对外提供的最核心的接口，通过它可以执行数据库读写命令、获取映射器、管理事务等；`SqlSession` 也意味着客户端与数据库的一次连接，客户端对数据库的访问请求都是由 `SqlSession` 来处理的，`SqlSession` 由 `SqlSessionFactory` 创建，每个 `SqlSession` 都会引用 `SqlSessionFactory` 中全局唯一单例存在的 `configuration` 对象；如下图所示：



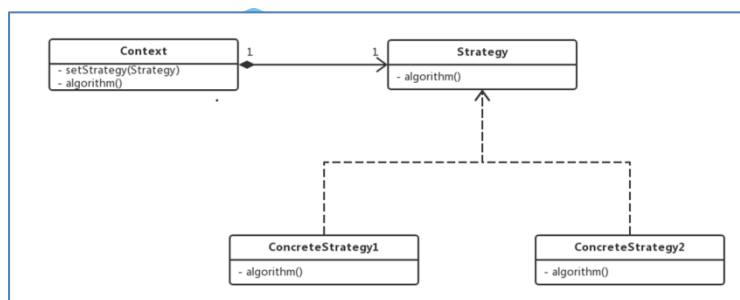
要深入理解 `SqlSession` 就得深入到源码进行学习，`SqlSession` 默认实现类为 `org.apache.ibatis.session.defaults.DefaultSqlSession`，解读如下：

- (1) `SqlSession` 是 `MyBatis` 的门面，是 `MyBatis` 对外提供数据访问的主要 API，实例代码：`com.enjoylearning.mybatis.MybatisDemo.originalOperation()`
- (2) 实际上 `SqlSession` 的功能都是基于 `Executor` 来实现的，遵循了单一职责原则，例如在 `SqlSession` 中的各种查询形式，最终会把请求转发到 `Executor.query` 方法，如下图所示：



3.1.2 策略模式

策略模式 (Strategy Pattern) 策略模式定义了一系列的算法，并将每一个算法封装起来，而且使他们可以相互替换，让算法独立于使用它的客户而独立变化。Spring 容器中使用配置可以灵活的替换掉接口的实现类就是策略模式最常见的应用。类图如下：



- ✓ Context：算法调用者，使用 setStrategy 方法灵活的选择策略（strategy）；
- ✓ Strategy：算法的统一接口；
- ✓ ConcreteStrategy：算法的具体实现；

策略模式的使用场景：

- （1）针对同一类型问题的多种处理方式，仅仅是具体行为有差别时；
- （2）出现同一抽象类有多个子类，而又需要使用 if-else 或者 switch-case 来选择具体子类时。

3.1.3 SqlSessionFactory

SqlSessionFactory 使用工厂模式创建 SqlSession，其默认的实现类为 DefaultSqlSessionFactory，其中获取 SqlSession 的核心方法为 openSessionFromDataSource(ExecutorType, TransactionIsolationLevel, boolean)，在这个方法中从 configuration 中获取的 TransactionFactory 是典型的策略模式的应用。运行期，TransactionFactory 接口的实现，是由配置文件配置决定的，可配置选项包括：JDBC、Managed，可根据需求灵活的替换 TransactionFactory 的实现；配置文件截图如下：

```

<!--配置environment环境-->
<environments default="development">
  <!-- 环境配置1, 每个SqlSessionFactory对应一个环境 -->
  <environment id="development">
    <transactionManager type="JDBC" />
    <dataSource type="POOLED">
      <property name="driver" value="${jdbc_driver}" />
      <property name="url" value="${jdbc_url}" />
      <property name="username" value="${jdbc_username}" />
      <property name="password" value="${jdbc_password}" />
    </dataSource>
  </environment>

```

灵活配置根据需要
替换实现

3.2 binding 模块分析

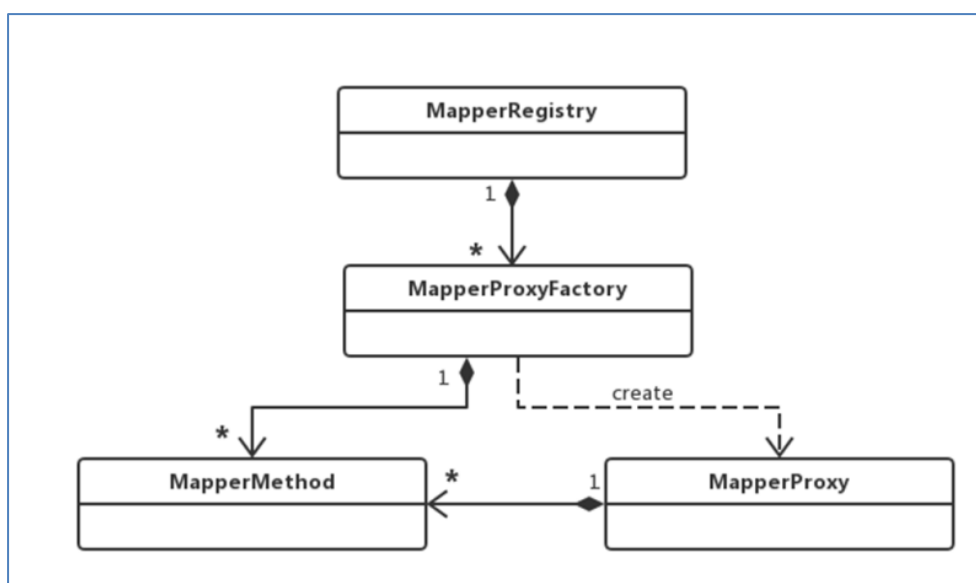
上面提到 `SqlSession` 是 MyBatis 对外提供数据库访问最主要的 API，但是因为直接使用 `SqlSession` 进行数据库开发存在代码可读性差、可维护性差的问题，所以我们很少使用，而是使用 `Mapper` 接口的方式进行数据库的开发。表面上我们在使用 `Mapper` 接口编程，实际上 MyBatis 的内部，将对 `Mapper` 接口的调用转发给了 `SqlSession`，这个请求的转发是建立在配置文件解读、动态代理增强的基础之上实现的，实现的过程有三个关键要素：

- ✓ 找到 `SqlSession` 中对应的方法执行；
- ✓ 找到命名空间和方法名（两维坐标）
- ✓ 传递参数

要实现上述的步骤，必须对 `binding` 模块有深入的分析：

3.2.1 binding 模块核心类

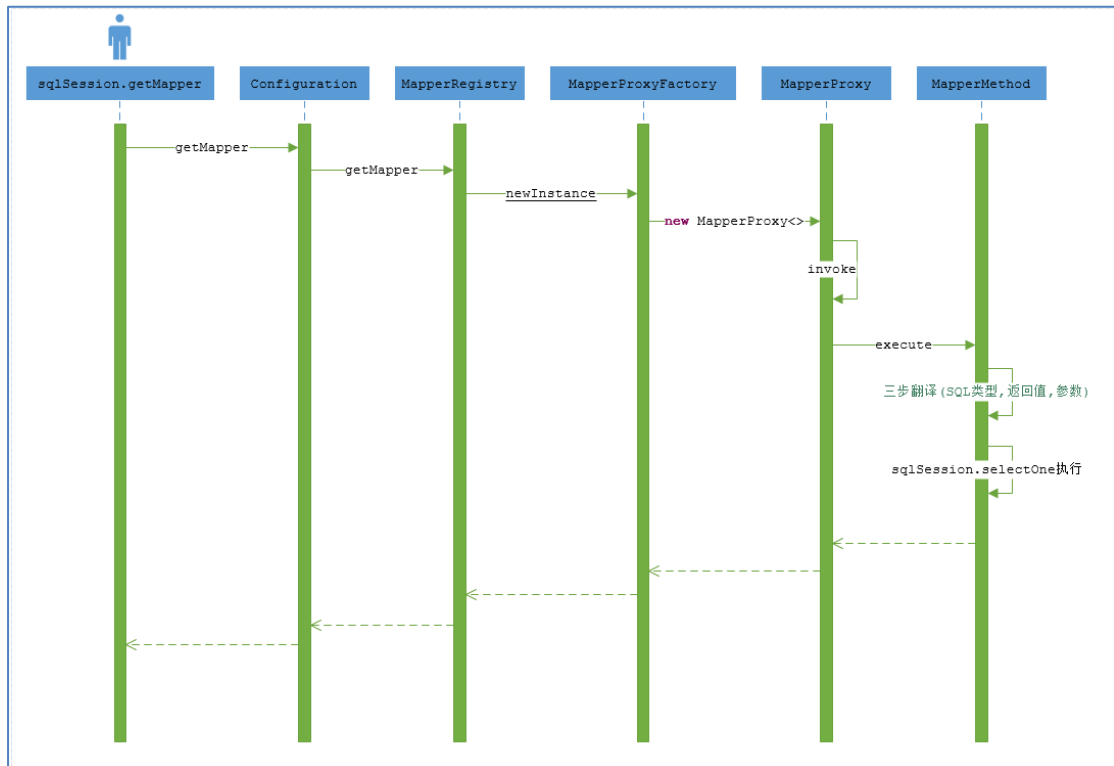
`binding` 模块核心类结构如下：



- ✓ **MapperRegistry**: mapper 接口和对应的代理对象工厂的注册中心;
- ✓ **MapperProxyFactory**: 用于生成 mapper 接口动态代理的实例对象; 保证 Mapper 实例对象是局部变量;
- ✓ **MapperProxy**: 实现了 **InvocationHandler** 接口, 它是增强 mapper 接口的实现;
- ✓ **MapperMethod**: 封装了 Mapper 接口中对应方法的信息, 以及对应的 sql 语句的信息; 它是 mapper 接口与映射配置文件中 sql 语句的桥梁; **MapperMethod** 对象不记录任何状态信息, 所以它可以在多个代理对象之间共享; **MapperMethod** 内几个关键数据结构:
 - **SqlCommand** : 从 configuration 中获取方法的命名空间.方法名以及 SQL 语句的类型;
 - **MethodSignature**: 封装 mapper 接口方法的相关信息 (入参, 返回类型);
 - **ParamNameResolver**: 解析 mapper 接口方法中的入参, 将多个参数转成 Map;

3.2.2 binding 模块运行流程

从 `SqlSession.getMapper(Class<T>)` 方法开始跟踪, 画出 binding 模块的时序图如下所示:



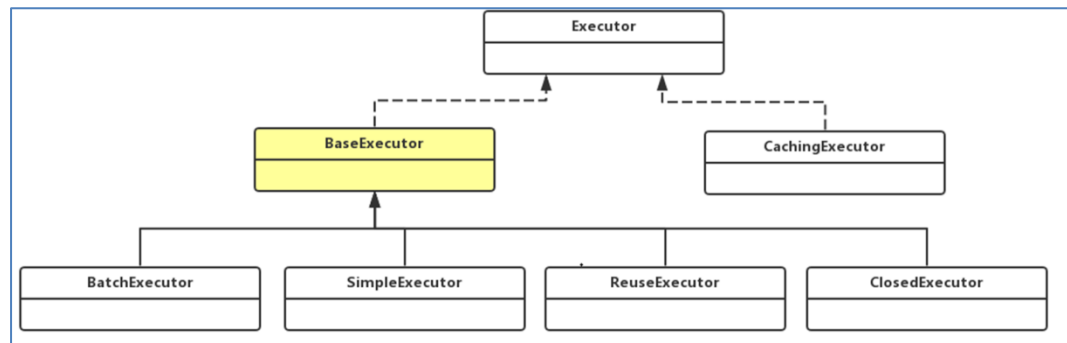
在 binding 模块的运行流程中实现三步翻译的核心方法是 **MapperMethod.execute(SqlSession, Object[])**, 翻译的过程描述如下:

- ✓ 通过 **Sql 语句的类型** (**MapperMethod.SqlCommand.type**) 和 **mapper 接口方法的返回参数** (**MapperMethod.MethodSignature.returnType**) 确定调用 **SqlSession** 中的某个方法;
- ✓ 通过 **MapperMethod.SqlCommand.name** 生成两维坐标;
- ✓ 通过 **MapperMethod.MethodSignature.paramNameResolve** 将传入的多个参数转成 **Map** 进行参数传递;

4. 第三个阶段：数据访问阶段

4.1 关于 Executor 组件

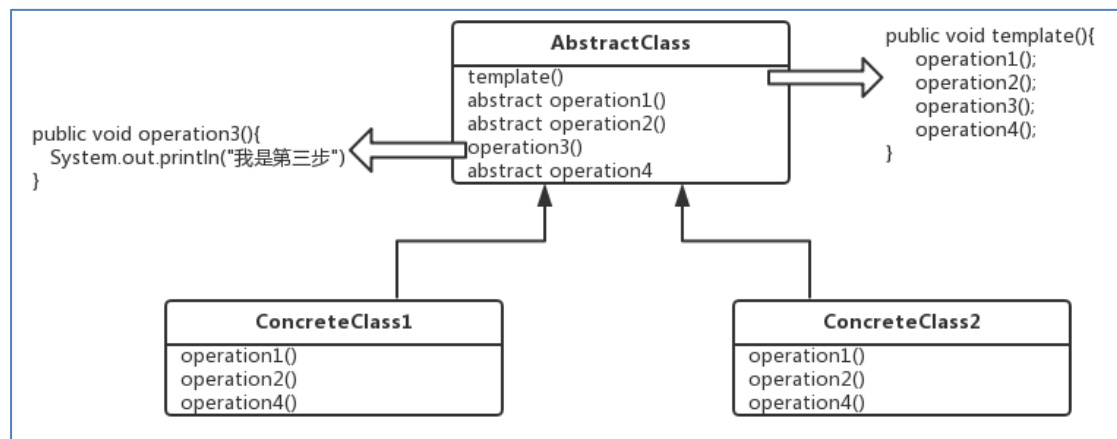
在讲第二阶段的时候，提到 `SqlSession` 的功能都是基于 `Executor` 来实现的，`Executor` 是 `MyBatis` 核心接口之一，定义了数据库操作最基本的方法，在其内部遵循 `JDBC` 规范完成对数据库的访问；`Executor` 类继承机构如下图所示：



- ✓ `Executor`: `MyBatis` 核心接口之一，定义了数据库操作最基本的方法；
- ✓ `CachingExecutor`: 使用装饰器模式，对真正提供数据库查询的 `Executor` 增强了二级缓存的能力；二级缓存初始化位置：`DefaultSqlSessionFactory.openSessionFromDataSource(ExecutorType, TransactionIsolationLevel, boolean)`；
- ✓ `BaseExecutor`: 抽象类，实现了 `executor` 接口的大部分方法，主要提供了缓存管理和事务管理的能力，其他子类需要实现的抽象方法为：`doUpdate`, `doQuery` 等方法；
- ✓ `BatchExecutor`: 批量执行所有更新语句，基于 `JDBC` 的 `batch` 操作实现批处理；
- ✓ `SimpleExecutor`: 默认执行器，每次执行都会创建一个 `statement`，用完后关闭。；
- ✓ `ReuseExecutor`: 可重用执行器，将 `statement` 存入 `map` 中，操作 `map` 中的 `statement` 而不会重复创建 `statement`；

4.2 Executor 中的模板模式

模板模式：一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。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。模板方法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实现；类结构如下：

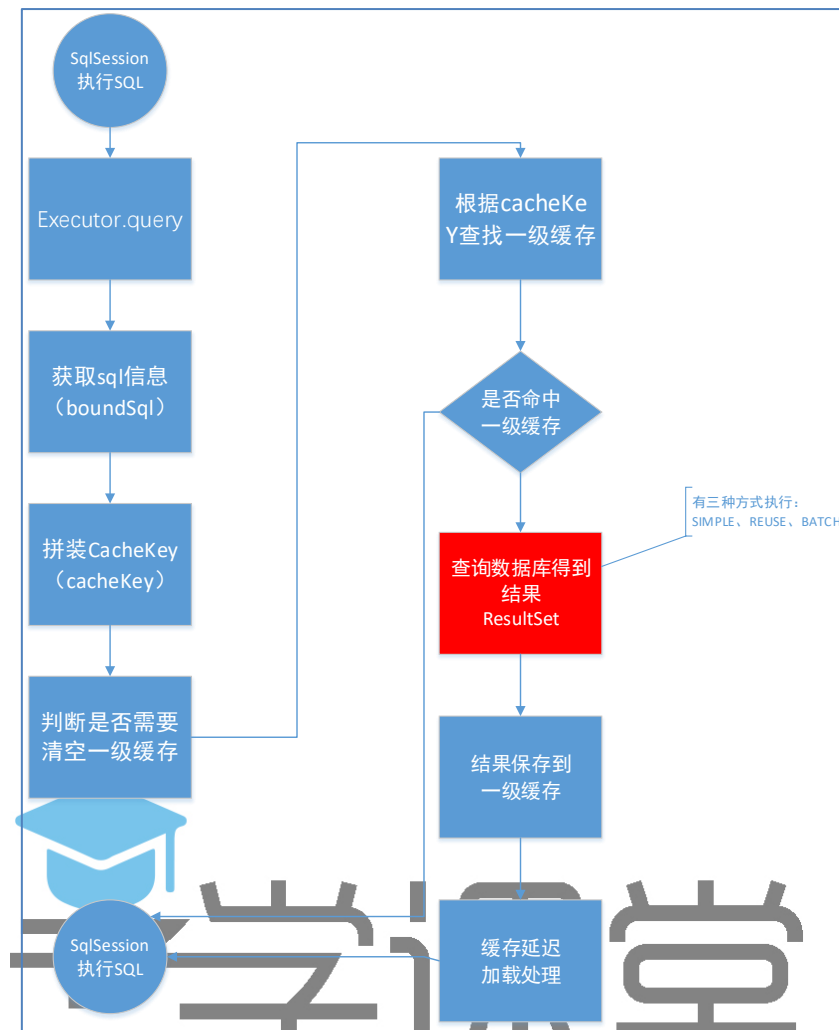


AbstractClass 中模板方法 **template()**定义了功能实现的多个步骤，抽象父类只会对其中几个通用的步骤有实现，而一些可定制化的步骤延迟到子类 **ConcreteClass1**、**ConcreteClass2** 中实现，子类只能定制某几个特定步骤的实现，而不能改变算法的结构；

应用场景：遇到由一系列步骤构成的过程需要执行，这个过程从高层次上看是相同的，但是有些步骤的实现可能不同，这个时候就需要考虑用模板模式了；

MyBatis 的执行器组件是使用模板模式的典型应用，其中 **BaseExecutor**、**BaseStatementHandler** 是模板模式的最佳实践；**BaseExecutor** 执行器抽象类，实现了 **executor** 接口的大部分方法，主要提供了缓存管理和事务管理的能力，其他子类需要实现的抽象方法为：**doUpdate**、**doQuery** 等方法；在 **BaseExecutor** 中进行一次数据库查询操作的流程如下：

学课堂



如上图所示，doQuery 方法是查询数据的结果的子步骤，doQuery 方法有 SIMPLE、REUSER、BATCH 三种实现，这三种不同的实现是在子类中定义的：

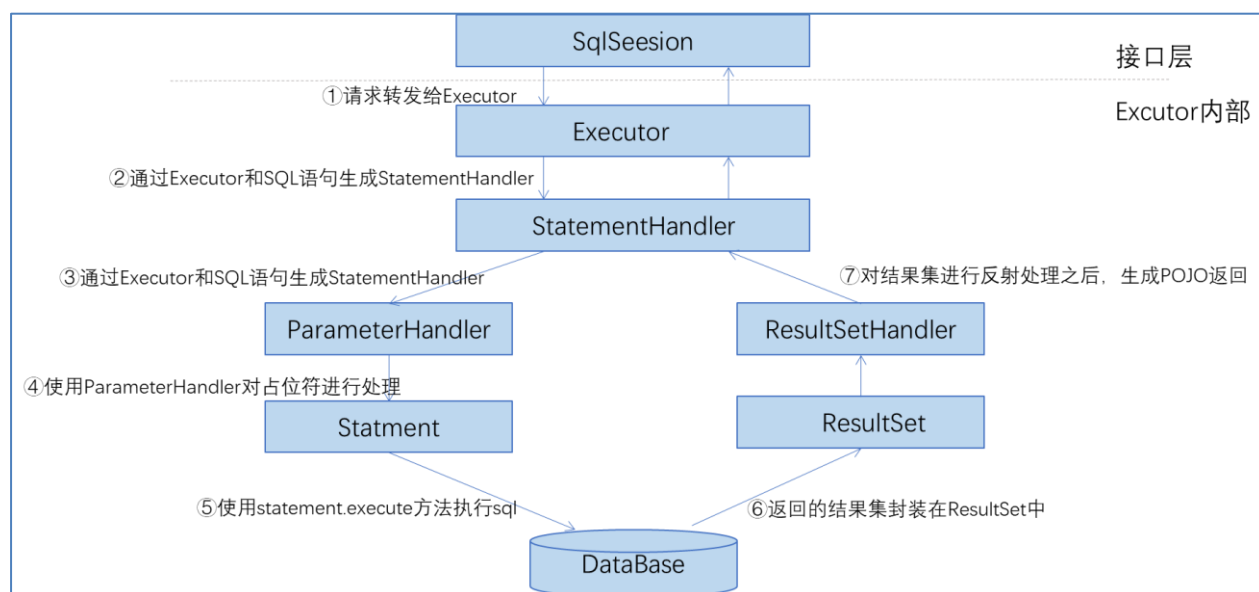
- ✓ SimpleExecutor: 默认配置，在 doQuery 方法中使用 PreparedStatement 对象访问数据库，每次访问都要创建新的 PreparedStatement 对象；
- ✓ ReuseExecutor: 在 doQuery 方法中，使用预编译 PreparedStatement 对象访问数据库，访问时，会重用缓存中的 statement 对象；
- ✓ BatchExecutor: 在 doQuery 方法中，实现批量执行多条 SQL 语句的能力；

4.3 Executor 的三个重要小弟

通过对 SimpleExecutor doQuery()方法的解读发现，Executor 是个指挥官，它在调度三个小弟工作，这三个小弟分别为：

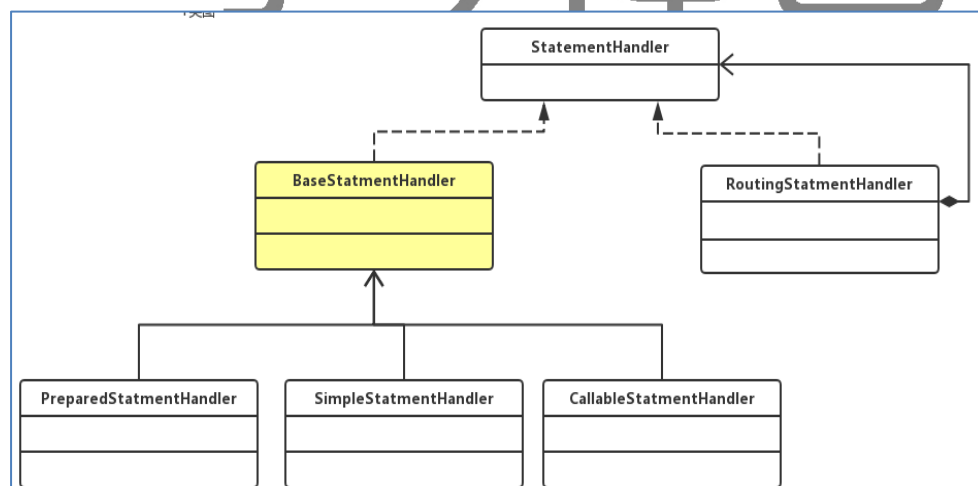
- ✓ StatementHandler: 它的作用是使用数据库的 Statement 或 PreparedStatement 执行操作，启承上启下作用；
- ✓ ParameterHandler: 对预编译的 SQL 语句进行参数设置，SQL 语句中的占位符 “？” 都对应 BoundSql.parameterMappings 集合中的一个元素，在该对象中记录了对应的参数名称以及该参数的相关属性
- ✓ ResultSetHandler: 对数据库返回的结果集（ResultSet）进行封装，返回用户指定的实体类型；

Executor 三小弟内部运作流程如下图所示：



4.4 关于 StatementHandler

StatementHandler 完成 Mybatis 最核心的工作，也是 Executor 实现的基础；功能包括：创建 statement 对象，为 sql 语句绑定参数，执行增删改查等 SQL 语句、将结果映射集进行转化；StatementHandler 的类继承关系如下图所示：



- ✓ **BaseStatementHandler**：所有子类的抽象父类，定义了初始化 statement 的操作顺序，由子类实现具体的实例化不同的 statement（模板模式）；
- ✓ **RoutingStatementHandler**：Excutor 组件真正实例化的子类，使用静态代理模式，根据上下文决定创建哪个具体实体类；
 - (1) RoutingStatementHandler 是在 Configuration 的 newStatementHandler 中创建的，见下图：

(2) `RoutingStatementHandler` 中使用动态代理的方式进行请求转发，在构造方法中，根据上下文（配置）决定创建具体实现类；如下图：

- ✓ SimpleStatmentHandler : 使用 statement 对象访问数据库, 无须参数化;
- ✓ PreparedStatmentHandler : 使用预编译 PrepareStatement 对象访问数据库;
- ✓ CallableStatmentHandler : 调用存储过程;

ResultSetHandler 将从数据库查询得到的结果按照映射配置文件的映射规则，映射成相应的结果集对象；在 **ResultSetHandler** 内部实际是做三个步骤：找到映射匹配规则 → 反射实例化目标对象 → 根据规则填充属性值，为完成这三部 **ResultHandler** 内部调用的流程图如下：

