源码总结回顾

MyBatis 核心对象

对象	相关对象	作用
Configuration	MapperRegistry TypeAliasRegistry TypeHandlerRegistry	包含了 MyBatis 的所有的配置信息
SqlSession	SqlSessionFactory DefaultSqlSession	对操作数据库的增删改查的 API 进行了封装，提供给应用层使用
Executor	BaseExecutor SimpleExecutor BatchExecutor ReuseExecutor	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	BaseStatementHandler SimpleStatementHandler PreparedStatementHandler CallableStatementHandler	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	DefaultParameterHandler	把用户传递的参数转换成 JDBC Statement 所需要的参数
ResultSetHandler	DefaultResultSetHandler	把 JDBC 返回的 ResultSet 结果集对象转换成 List 类型的集合
MapperProxy	MapperProxyFactory	代理对象，用于代理 Mapper 接口方法
MappedStatement	SqlSource BoundSql	MappedStatement 维护了一条<select|update|delete|insert>节点的封装，包括了 SQL 信息、入参信息、出参信息

课程目标

掌握插件的使用方法和工作原理

掌握自定义插件的编写方法

掌握 Spring 集成 MyBatis 的原理

内容定位

适合不了解插件工作原理的同学；

适合不清楚 MyBatis 整合到 Spring 的原理的同学。

MyBatis 插件原理与自定义插件

MyBatis 通过提供插件机制，让我们可以根据自己的需要去增强 MyBatis 的功能。

需要注意的是，如果没有完全理解 MyBatis 的运行原理和插件的工作方式，最好不要使用插件，因为它会改变系底层的工作逻辑，给系统带来很大的影响。

1、猜想

MyBatis 的插件可以在不修改原来的代码的情况下，通过拦截的方式，改变四大核心对象的行为，比如处理参数，处理 SQL，处理结果。

第一个问题：

不修改对象的代码，怎么对对象的行为进行修改，比如说在原来的方法前面做一点事情，在原来的方法后面做一点事情？

答案：大家很容易能想到用代理模式，这个也确实是 MyBatis 插件的原理。

第二个问题：

我们可以定义很多的插件，那么这种所有的插件会形成一个链路，比如我们提交一个休假申请，先是项目经理审批，然后是部门经理审批，再是 HR 审批，再到总经理

审批，怎么实现层层的拦截？

答案：插件是层层拦截的，我们又需要用到另一种设计模式——责任链模式。

如果是用代理模式，我们就要解决几个问题：

1） 有哪些对象允许被代理？有哪些方法可以被拦截？

我们应该了解 MyBatis 允许哪些对象的哪些方法允许被拦截，并不是每一个运行的节点都是可以被修改的。只有清楚了这些对象的方法的作用，当我们自己编写插件的时候才知道从哪里去拦截。

在 MyBatis 官网有答案，我们来看一下：

http://www.mybatis.org/mybatis-3/zh/configuration.html#plugins

对象	描述	可拦截的方法	方法作用
Executor	上层的对象，SQL 执行全过程，包括组装参数，组装结果集返回和执行 SQL 过程	update	执行 update、insert、delete 操作
		query	执行 query 操作
		flushStatements	在 commit 的时候自动调用，SimpleExecutor、ReuseExecutor、BatchExecutor 处理不同
		commit	提交事务
		rollback	事务回滚
		getTransaction	获取事务
		close	结束（关闭）事务
		isClosed	判断事务是否关闭
StatementHandler	执行 SQL 的过程，最常用的拦截对象	prepare	（BaseSatementHandler）SQL 预编译
		parameterize	设置参数
		batch	批处理
		update	增删改操作
		query	查询操作
ParameterHandler	SQL 参数组装的过程	getParameterObject	获取参数
		setParameters	设置参数
ResultSetHandler	结果的组装	handleResultSets	处理结果集
		handleOutputParameters	处理存储过程出参

Executor 会拦截到 CachingExcecutor 或者 BaseExecutor。

因为创建 Executor 时是先创建 Executor，再拦截。

2） 怎么样创建代理？

如果我们用 JDK 的动态代理，要有一个实现了 InvocationHandler 的代理类，用来包装被代理对象，这个类是我们自己创建还是谁来创建？

3） 什么时候创建代理对象？是在 MyBatis 启动的时候创建，还是调用的时候创建？

4） 被代理后，调用的是什么方法？怎么调用到原被代理对象的方法（比如 Executor 的 query()方法）？

要解决后面三个问题，我们先看一下别人的插件是怎么工作的。

2、插件编写与注册

（基于 spring-mybatis）运行自定义的插件，需要 3 步，我们以 PageHelper 为例：

1、编写自己的插件类

1）实现 Interceptor 接口

这个是所有的插件必须实现的接口。

2）添加@Intercepts({@Signature()})，指定拦截的对象和方法、方法参数

方法名称+参数类型，构成了方法的签名，决定了能够拦截到哪个方法。

问题：拦截签名跟参数的顺序有关系吗？

3）实现接口的 3 个方法

```
// 用于覆盖被拦截对象的原有方法（在调用代理对象 Plugin 的 invoke()方法时被调用）
Object intercept(Invocation invocation) throws Throwable;

// target 是被拦截对象，这个方法的作用是给被拦截对象生成一个代理对象，并返回它
Object plugin(Object target);

// 设置参数
void setProperties(Properties properties);
```

2、插件注册，在 mybatis-config.xml 中注册插件

```
<plugins>
    <plugin interceptor="com.github.pagehelper.PageInterceptor">
        <property name="offsetAsPageNum" value="true"/>
            ……后面全部省略……
    </plugin>
</plugins>
```

3、插件登记

MyBatis 启动时扫描 <plugins> 标签，注册到 Configuration 对象的 InterceptorChain 中。property 里面的参数，会调用 setProperties()方法处理。

以上就是编写和使用自定义插件的全部步骤。

3、代理和拦截是怎么实现的？

问题 1：四大对象什么时候被代理，也就是：代理对象是什么时候创建的？

问题 2：多个插件的情况下，代理能不能被代理？代理顺序和调用顺序的关系？

问题 3：谁来创建代理对象？

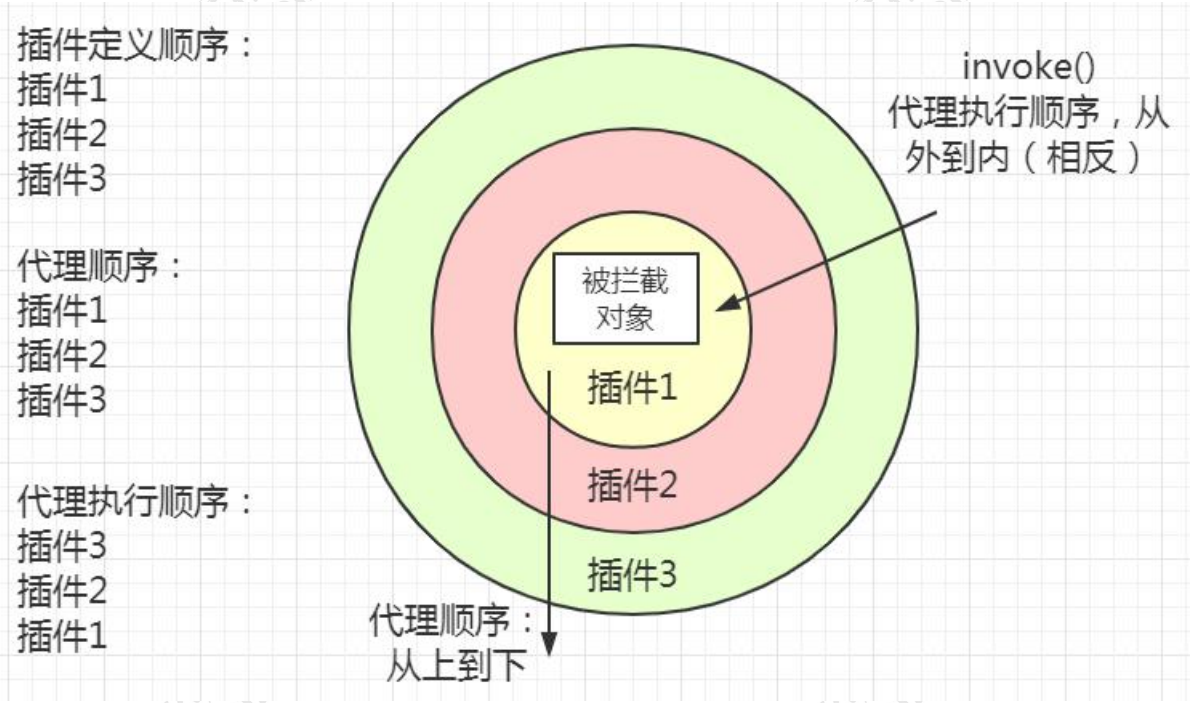
问题 4：被代理后，调用的是什么方法？怎么调用到原被代理对象的方法？

问题 1：

Executor 是 openSession() 的时候创建的；StatementHandler 是 SimpleExecutor.doQuery()创建的；里面包含了处理参数的 ParameterHandler 和处理结果集的 ResultSetHandler 的创建，创建之后即调用 InterceptorChain.pluginAll()，返回层层代理后的对象。

问题 2：

可以被代理，debug 看一下。



问题 3：

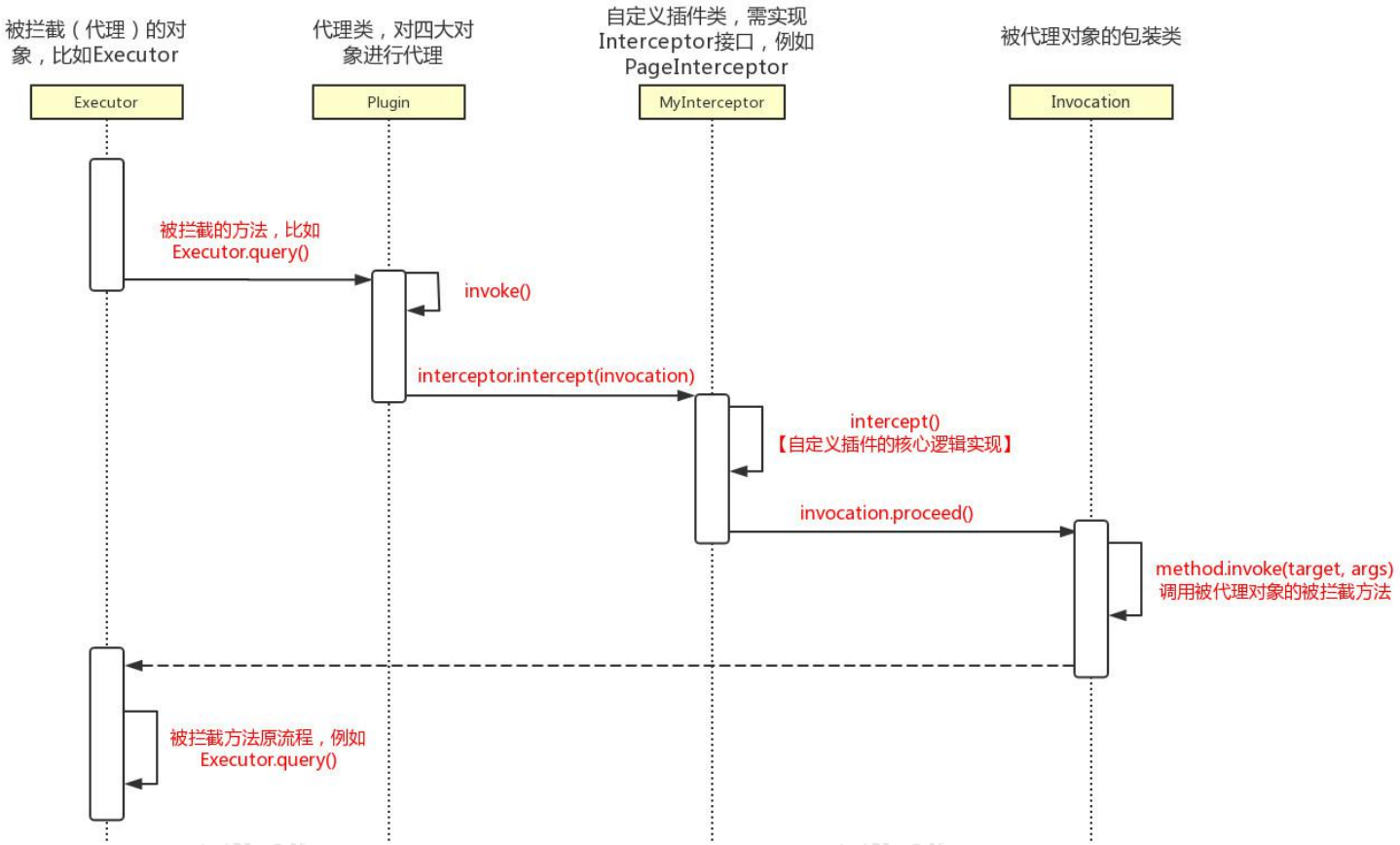
Plugin 类。在我们重写的 plugin() 方法里面可以直接调用 return Plugin.wrap(target, this);返回代理对象。

问题 4：

因为代理类是 Plugin，所以最后调用的是 Plugin 的 invoke()方法。它先调用了定义的拦截器的 intercept()方法。可以通过 invocation.proceed()调用到被代理对象被拦截的方法。

总结流程如下：

MyBatis插件调用流程



总结：

对象	作用
Interceptor	自定义插件需要实现接口，实现 3 个方法
InterceptChain	配置的插件解析后会保存在 Configuration 的 InterceptChain 中
Plugin	用来创建代理对象，包装四大对象
Invocation	对被代理类进行包装，可以调用 proceed()调用到被拦截的方法

4、PageHelper 原理

（基于 spring-mybatis）PageInterceptor 类

1、用法（EmployeeController.getEmpsWithJson()）

```
PageHelper.startPage(pn, 10);
List<Employee> emps = employeeService.getAll();
PageInfo page = new PageInfo(emps, 10);
```

先看 PageHelper jar 包中 PageInterceptor 的源码。拦截的是 Executor 的两个 query()方法。在这里对 SQL 进行了改写：

```
//调用方言获取分页 sql
String pageSql = dialect.getPageSql(ms, boundSql, parameter, rowBounds, pageKey);
```

跟踪到最后，是在 MySqlDialect.getPageSql()对 SQL 进行了改写，翻页参数是从一个 Page 对象中拿到的，那么 Page 对象是怎么传到这里的呢？

上一步，AbstractHelperDialect.getPageSql()中：

```
Page page = getLocalPage();
return getPageSql(sql, page, pageKey);
```

Page 对象是从一个 ThreadLocal<>变量中拿到的，那它是什么时候赋值的？

回到 EmployeeController.getEmpsWithJson()中，PageHelper.startPage()方法，把分页参数放到了 ThreadLocal<>变量中。

```
protected static void setLocalPage(Page page) {
    LOCAL_PAGE.set(page);
}
```

关键类总结：

对象	作用
PageInterceptor	自定义拦截器
Page	包装分页参数
PageInfo	包装结果
PageHelper	工具类

5、应用场景分析

作用	实现方式
水平分表	对 query update 方法进行拦截 在接口上添加注解，通过反射获取接口注解，根据注解上配置的参数进行分表，修改原 SQL，例如 id 取模，按月分表
数据加解密	update——加密；query——解密 获得入参和返回值
菜单权限控制	对 query 方法进行拦截 在方法上添加注解，根据权限配置，以及用户登录信息，在 SQL 上加上权限过滤条件

6、需求实现

【作业】现在我们有两个简单需求：

1、 当我们传入 RowBounds 做翻页查询的时候，使用 limit 物理分页，代替原来的逻辑分页。

基于 mybatis-standalone

MyBatisTest.java —— testSelectByRowBounds()

2、 在未启用日志组件的情况下，输出执行的 SQL（先实现查询的拦截），并且统计 SQL 的执行时间

与 Spring 整合分析

http://www.mybatis.org/spring/zh/index.html

这里我们以传统的 Spring 为例，因为配置更直观，在 Spring 中使用配置类注解是一样的。

在前面的课程里面，我们基于编程式的工程已经弄清楚了 MyBatis 的工作流程、核心模块和底层原理。编程式的工程，也就是 MyBatis 的原生 API 里面有三个核心对象：

SqlSessionFactory、SqlSession、MapperProxy。

大部分时候我们不会在项目中单独使用 MyBatis 的工程 ,而是集成到 Spring 里面使用，但是却没有看到这三个对象在代码里面的出现。我们直接注入了一个 Mapper 接口，调用它的方法。

所以有几个关键的问题，我们要弄清楚：

1、 SqlSessionFactory 是什么时候创建的？

2、 SqlSession 去哪里了？为什么不用它来 getMapper？

3、 为什么@Autowired 注入一个接口 ,在使用的时候却变成了代理对象？在 IOC 的容器里面我们注入的是什么？ 注入的时候发生了什么事情？

关键配置

我们先看一下把 MyBatis 集成到 Spring 中要做的几件事情。

为了让大家看起来更直观，这里我们依旧用传统的 xml 配置给大家来做讲解，当然

使用配置类@Configuration 效果也是一样的，对于 Spring 来说只是解析方式的差异。

除了 MyBatis 的依赖之外，我们还需要在 pom 文件中引入 MyBatis 和 Spring 整合的 jar 包（注意版本！mybatis 的版本和 mybatis-spring 的版本有兼容关系）。

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis-spring</artifactId>
    <version>2.0.0</version>
</dependency>
```

然后在 Spring 的 applicationContext.xml 里面配置 SqlSessionFactoryBean，它是用来帮助我们创建会话的，其中还要指定全局配置文件和 mapper 映射器文件的路径。

```
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <property name="configLocation" value="classpath:mybatis-config.xml"></property>
    <property name="mapperLocations" value="classpath:mapper/*.xml"></property>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

然后在 applicationContext.xml 配置需要扫描 Mapper 接口的路径。

在 Mybatis 里面有几种方式，第一种是配置一个 MapperScannerConfigurer。

```
<bean id="mapperScanner" class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <property name="basePackage" value="com.gupaoedu.crud.dao"/>
</bean>
```

第二种是配置一个<scan>标签：

```
<mybatis-spring:scan base-package="com.gupaoedu.crud.dao"/>
```

还有一种就是直接用@MapperScan 注解，比如我们在 Spring Boot 的启动类上加上一个注解：

```
@SpringBootApplication
@MapperScan("com.gupaoedu.crud.dao")
public class MybaitsApp {
   public static void main(String[] args) {
      SpringApplication.run(MybaitsApp.class, args);
   }
```

```
}
```

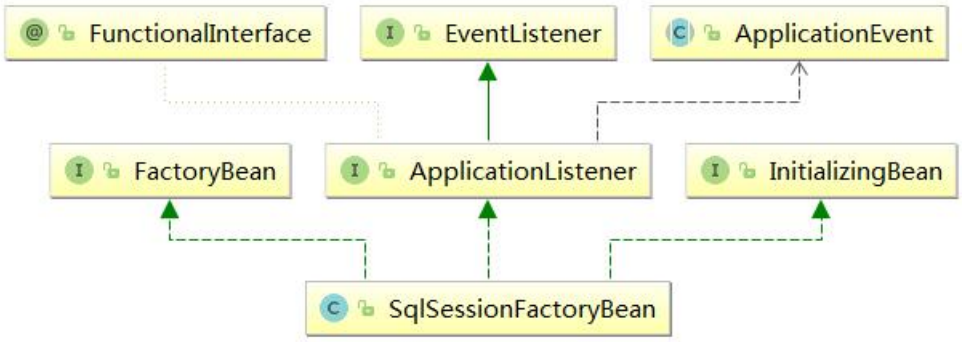
这三种方式实现的效果是一样的。

1、创建会话工厂

Spring 对 MyBatis 的对象进行了管理，但是并不会替换 MyBatis 的核心对象。也就意味着：MyBatis jar 包中的 SqlSessionFactory、SqlSession、MapperProxy 这些都会用到。而 mybatis-spring.jar 里面的类只是做了一些包装或者桥梁的工作。

所以第一步，我们看一下在 Spring 里面，工厂类是怎么创建的。

我们在 Spring 的配置文件中配置了一个 SqlSessionFactoryBean，我们来看一下这个类。



它实现了 InitializingBean 接口，所以要实现 afterPropertiesSet()方法，这个方法会在 bean 的属性值设置完的时候被调用。

另外它实现了 FactoryBean 接口，所以它初始化的时候，实际上是调用 getObject()方法，它里面调用的也是 afterPropertiesSet()方法。

在 afterPropertiesSet()方法里面：

第一步是一些标签属性的检查，接下来调用了 buildSqlSessionFactory()方法。

然后定义了一个 Configuration，叫做 targetConfiguration。

426 行，判断 Configuration 对象是否已经存在，也就是是否已经解析过。如果已经有对象，就覆盖一下属性。

433 行，如果 Configuration 不存在，但是配置了 configLocation 属性，就根据 mybatis-config.xml 的文件路径，构建一个 xmlConfigBuilder 对象。

436 行，否则，Configuration 对象不存在，configLocation 路径也没有，只能使用默认属性去构建去给 configurationProperties 赋值。

后面就是基于当前 factory 对象里面已有的属性 ,对 targetConfiguration 对象里面属性的赋值。

```
Optional.ofNullable(this.objectFactory).ifPresent(targetConfiguration::setObjectFactory);
```

这个方法是 Java8 里面的一个判空的方法。如果不为空的话，就会调用括号里面的对象的方法，做赋值的处理。

在第 498 行，如果 xmlConfigBuilder 不为空，也就是上面的第二种情况，调用了 xmlConfigBuilder.parse()去解析配置文件，最终会返回解析好的 Configuration 对象。

在第 507 行，如果没有明确指定事务工厂，默认使用 SpringManagedTransactionFactory。它创建的 SpringManagedTransaction 也有 getConnection()和 close()方法。

```
<property name="transactionFactory" value="" />
```

在 520 行，调用 xmlMapperBuilder.parse()，这个步骤我们之前了解过了，它的作用是把接口和对应的 MapperProxyFactory 注册到 MapperRegistry 中。

最后调用 sqlSessionFactoryBuilder.build() 返回了一个 DefaultSqlSessionFactory。

OK，在这里我们完成了编程式的案例里面的第一步，根据配置文件获得一个工厂类，它是单例的，会在后面用来创建 SqlSession。

用到的 Spring 扩展点总结：

接口	方法	作用
FactoryBean	getObject()	返回由 FactoryBean 创建的 Bean 实例
InitializingBean	afterPropertiesSet()	bean 属性初始化完成后添加操作
BeanDefinitionRegistryPostProcessor	postProcessBeanDefinitionRegistry()	注入 BeanDefination 时添加操作

2、创建 SqlSession

Q1：可以直接使用 DefaultSqlSession 吗？

我们现在已经有一个 DefaultSqlSessionFactory，按照编程式的开发过程，我们接下来就会创建一个 SqlSession 的实现类，但是在 Spring 里面，我们不是直接使用 DefaultSqlSession 的，而是对它进行了一个封装，这个 SqlSession 的实现类就是 SqlSessionTemplate。这个跟 Spring 封装其他的组件是一样的，比如 JdbcTemplate，RedisTemplate 等等，也是 Spring 跟 MyBatis 整合的最关键的一个类。

为什么不用 DefaultSqlSession？它是线程不安全的，注意看类上的注解：

```
Note that this class is not Thread-Safe.
```

而 SqlSessionTemplate 是线程安全的。

```
* Thread safe, Spring managed, {@code SqlSession} that works with Spring
```

回顾一下 SqlSession 的生命周期：

对象	生命周期
SqlSessionFactoryBuiler	方法局部（method）
SqlSessionFactory（单例）	应用级别（application）

SqlSession	请求和操作（request/method）
Mapper	方法（method）

在编程式的开发中，SqlSession 我们会在每次请求的时候创建一个，但是 Spring 里面只有一个 SqlSessionTemplate（默认是单例的），多个线程同时调用的时候怎么保证线程安全？

思考：为什么 SqlSessionTemplate 是线程安全的？

思考：在编程式的开发中，有什么方法保证 SqlSession 的线程安全？

SqlSessionTemplate 里面有 DefaultSqlSession 的所有的方法：selectOne()、selectList()、insert()、update()、delete()，不过它都是通过一个代理对象实现的。这个代理对象在构造方法里面通过一个代理类创建：

```
this.sqlSessionProxy = (SqlSession) newProxyInstance(
    SqlSessionFactory.class.getClassLoader(),
    new Class[] { SqlSession.class },
    new SqlSessionInterceptor());
```

所有的方法都会先走到内部代理类 SqlSessionInterceptor 的 invoke()方法：

```
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    SqlSession sqlSession = getSqlSession(
        SqlSessionTemplate.this.sqlSessionFactory,
        SqlSessionTemplate.this.executorType,
        SqlSessionTemplate.this.exceptionTranslator);
    try {
      Object result = method.invoke(sqlSession, args);
```

首先会使用工厂类、执行器类型、异常解析器创建一个 sqlSession，然后再调用 sqlSession 的实现类，实际上就是在这里调用了 DefaultSqlSession 的方法。

Q2：怎么拿到一个 SqlSessionTemplate？

我们知道在 Spring 里面会用 SqlSessionTemplate 替换 DefaultSqlSession，那么接下来看一下怎么在 DAO 层拿到一个 SqlSessionTemplate。

不知道用过 Hibernate 的同学还记不记得，如果不用注入的方式，我们在 DAO 层注入一个 HibernateTemplate 的一种方法是什么？

——让我们 DAO 层的实现类去继承 HibernateDaoSupport。

MyBatis 里面也是一样的，它提供了一个 SqlSessionDaoSupport，里面持有一个 SqlSessionTemplate 对象，并且提供了一个 getSqlSession()方法，让我们获得一个 SqlSessionTemplate。

```
public abstract class SqlSessionDaoSupport extends DaoSupport {

  private SqlSessionTemplate sqlSessionTemplate;

  public SqlSession getSqlSession() {
    return this.sqlSessionTemplate;
  }
  前面和后面省略.....
```

也就是说我们让 DAO 层的实现类继承 SqlSessionDaoSupport，就可以获得 SqlSessionTemplate，然后在里面封装 SqlSessionTemplate 的方法。

当然，为了减少重复的代码，我们通常不会让我们的实现类直接去继承 SqlSessionDaoSupport，而是先创建一个 BaseDao 继承 SqlSessionDaoSupport。在 BaseDao 里面封装对数据库的操作，包括 selectOne()、selectList()、insert()、delete()这些方法，子类就可以直接调用。

```
public  class BaseDao extends SqlSessionDaoSupport {
    //使用 sqlSessionFactory
    @Autowired
    private SqlSessionFactory sqlSessionFactory;

    @Autowired
```

```
    public void setSqlSessionFactory(SqlSessionFactory sqlSessionFactory) {
        super.setSqlSessionFactory(sqlSessionFactory);
    }

    public Object selectOne(String statement, Object parameter) {
        return getSqlSession().selectOne(statement, parameter);
    }
    后面省略.....
```

然后让我们的实现类继承 BaseDao 并且实现我们的 DAO 层接口，这里就是我们的 Mapper 接口。实现类需要加上@Repository 的注解。

在实现类的方法里面，我们可以直接调用父类(BaseDao)封装的 selectOne()方法，那么它最终会调用 sqlSessionTemplate 的 selectOne()方法。

```
@Repository
public class EmployeeDaoImpl extends BaseDao implements EmployeeMapper {
    @Override
    public Employee selectByPrimaryKey(Integer empId) {
        Employee emp = (Employee)
this.selectOne("com.gupaoedu.crud.dao.EmployeeMapper.selectByPrimaryKey",empId);
        return emp;
    }
    后面省略.....
```

然后在需要使用的地方，比如 Service 层，注入我们的实现类，调用实现类的方法就行了。我们这里直接在单元测试类里面注入：

```
@Autowired
EmployeeDaoImpl employeeDao;

@Test
public void EmployeeDaoSupportTest() {
    System.out.println(employeeDao.selectByPrimaryKey(1));
}
```

最终会调用到 DefaultSqlSession 的方法。

Q3：有没有更好的拿到 SqlSessionTemplate 的方法？

这么做有一个问题：我们的每一个 DAO 层的接口（Mapper 接口也属于），如果要拿到一个 SqlSessionTemplate，去操作数据库，都要创建实现一个实现类，加上@Repository 的注解，继承 BaseDao，这个工作量也不小。

另外一个，我们去直接调用 selectOne()方法，还是出现了 Statement ID 的硬编码，MapperProxy 在这里根本没用上。

我们可以通过什么方式，不创建任何的实现类，就可以把 Mapper 注入到别的地方使用，并且可以拿到 SqlSessionTemplate 操作数据库呢？

这个也确实是我们在 Spring 中的用法。那我们就必要弄清楚，我们只是注入了一个接口，在对象实例化的时候，是怎么拿到 SqlSessionTemplate 的？当我们调用方法的时候，还是不是用的 MapperProxy？

3、接口的扫描注册

在 Service 层可以使用@Autowired 自动注入的 Mapper 接口，需要保存在 BeanFactory（比如 XmlWebApplicationContext）中。也就是说接口肯定是在 Spring 启动的时候被扫描了，注册过的。

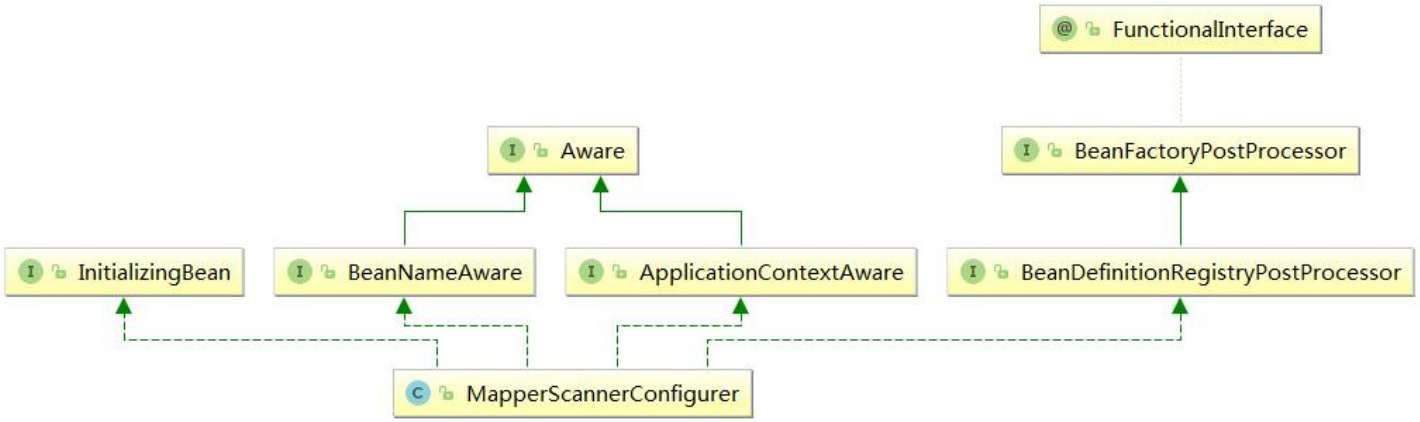
1、 什么时候扫描的？

2、 注册的时候，注册的是什么？这个决定了我们拿到的是什么实际对象。

回顾一下，我们在 applicationContext.xml 里面配置了一个 MapperScannerConfigurer。

MapperScannerConfigurer 实现了 BeanDefinitionRegistryPostProcessor 接口，BeanDefinitionRegistryPostProcessor 是 BeanFactoryPostProcessor 的子类，可以

通过编码的方式修改、新增或者删除某些 Bean 的定义。



我们只需要重写 postProcessBeanDefinitionRegistry()方法，在这里面操作 Bean 就可以了。

在这个方法里面：

scanner.scan()方法是 ClassPathBeanDefinitionScanner 中的，而它的**子类** ClassPathMapperScanner 覆盖了 doScan()方法，在 doScan()中调用了 processBeanDefinitions：

```
public Set<BeanDefinitionHolder> doScan(String... basePackages) {
  Set<BeanDefinitionHolder> beanDefinitions = super.doScan(basePackages);

  if (beanDefinitions.isEmpty()) {
    LOGGER.warn(() -> "No MyBatis mapper was found in '" + Arrays.toString(basePackages) +
"' package. Please check your configuration.");
  } else {
    processBeanDefinitions(beanDefinitions);
  }

  return beanDefinitions;
}
```

它先调用父类的 doScan()扫描所有的接口。

processBeanDefinitions 方法里面，在注册 beanDefinitions 的时候，BeanClass 被改为 MapperFactoryBean（注意灰色的注释）。

```
private void processBeanDefinitions(Set<BeanDefinitionHolder> beanDefinitions) {
  GenericBeanDefinition definition;
  for (BeanDefinitionHolder holder : beanDefinitions) {
    definition = (GenericBeanDefinition) holder.getBeanDefinition();
    String beanClassName = definition.getBeanClassName();
    LOGGER.debug(() -> "Creating MapperFactoryBean with name '" + holder.getBeanName()
        + "' and '" + beanClassName + "' mapperInterface");

    // the mapper interface is the original class of the bean
    // but, the actual class of the bean is MapperFactoryBean
    definition.getConstructorArgumentValues().addGenericArgumentValue(beanClassName); // 
issue #59
    definition.setBeanClass(this.mapperFactoryBean.getClass());
```

问题：

为什么要把 BeanClass 修改成 MapperFactoryBean，这个类有什么作用？

MapperFactoryBean 继承了 SqlSessionDaoSupport，可以拿到 SqlSessionTemplate。

4、接口注入使用

我们使用 Mapper 的时候，只需要在加了 Service 注解的类里面使用@Autowired 注入 Mapper 接口就好了。

```
@Service
public class EmployeeService {
    @Autowired
    EmployeeMapper employeeMapper;

    public List<Employee> getAll() {
        return employeeMapper.selectByMap(null);
    }
}
```

Spring 在启动的时候需要去实例化 EmployeeService。

EmployeeService 依赖了 EmployeeMapper 接口(是 EmployeeService 的一个属性）。

Spring 会根据 Mapper 的名字从 BeanFactory 中获取它的 BeanDefination，再从 BeanDefination 中获取 BeanClass，EmployeeMapper 对应的 BeanClass 是 MapperFactoryBean（上一步已经分析过）。

接下来就是创建 MapperFactoryBean，因为实现了 FactoryBean 接口，同样是调用 getObject()方法。

```
// MapperFactoryBean.java
public T getObject() throws Exception {
  return getSqlSession().getMapper(this.mapperInterface);
}
```

因为 MapperFactoryBean 继承了 SqlSessionDaoSupport，所以这个 getSqlSession()就是调用父类的方法，返回 SqlSessionTemplate。

```
// SqlSessionDaoSupport.java
public SqlSession getSqlSession() {
  return this.sqlSessionTemplate;
}
```

第二步，SqlSessionTemplate 的 getMapper()方法，里面又有两个方法：

```
// SqlSessionTemplate.java
public <T> T getMapper(Class<T> type) {
  return getConfiguration().getMapper(type, this);
}
```

第一步：SqlSessionTemplate 的 getConfiguration()方法：

```
// SqlSessionTemplate.java
public Configuration getConfiguration() {
  return this.sqlSessionFactory.getConfiguration();
}
```

进入方法，通过 DefaultSqlSessionFactory，返回全部配置 Configuration：

```
// DefaultSqlSessionFactory.java
public Configuration getConfiguration() {
  return configuration;
}
```

第二步：Configuration 的 getMapper()方法：

```
// Configuration.java
public <T> T getMapper(Class<T> type, SqlSession sqlSession) {
  return mapperRegistry.getMapper(type, sqlSession);
}
```

这一步我们很熟悉，跟编程式使用里面的 getMapper 一样，通过工厂类 MapperProxyFactory 获得一个 MapperProxy 代理对象。

也就是说 ,我们注入到 Service 层的接口 ,实际上还是一个 MapperProxy 代理对象。所以最后调用 Mapper 接口的方法，也是执行 MapperProxy 的 invoke()方法，后面的流程就跟编程式的工程里面一模一样了。

总结：

对象	生命周期
SqlSessionTemplate	Spring 中 SqlSession 的替代品，是线程安全的，通过代理的方式调用 DefaultSqlSession 的方法
SqlSessionInterceptor（内部类）	代理对象，用来代理 DefaultSqlSession，在 SqlSessionTemplate 中使用
SqlSessionDaoSupport	用于获取 SqlSessionTemplate，只要继承它即可
MapperFactoryBean	注册到 IOC 容器中替换接口类，继承了 SqlSessionDaoSupport 用来获取 SqlSessionTemplate，因为注入接口的时候，就会调用它的 getObject()方法
SqlSessionHolder	控制 SqlSession 和事务

思考：@MapperScan 注解是怎么解析的？

设计模式总结

设计模式	类
工厂	SqlSessionFactory、ObjectFactory、MapperProxyFactory
建造者	XMLConfigBuilder、XMLMapperBuilder、XMLStatementBuidler
单例模式	SqlSessionFactory、Configuration、ErrorContext
代理模式	绑定：MapperProxy 延迟加载：ProxyFactory（CGLIB、JAVASSIT） 插件：Plugin Spring 集成 MyBaits：SqlSessionTemplate 的内部类 SqlSessionInterceptor MyBatis 自带连接池：PooledDataSource 管理的 PooledConnection 日志打印：ConnectionLogger、StatementLogger
适配器模式	logging 模块，对于 Log4j、JDK logging 这些没有直接实现 slf4j 接口的日志组件，需要适配器
模板方法	BaseExecutor 与子类 SimpleExecutor、BatchExecutor、ReuseExecutor
装饰器模式	LoggingCache、LruCache 等对 PerpectualCache 的装饰 CachingExecutor 对其他 Executor 的装饰
责任链模式	InterceptorChain

作者：咕泡学院-青山

最后更新时间：2019 年 5 月 16 日 12:21:24