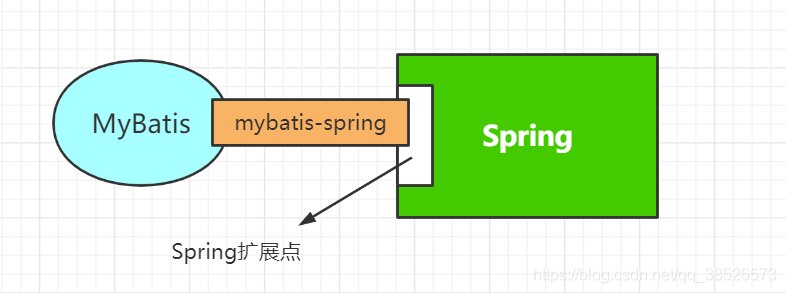
# 透彻掌握Spring整合MyBatis的原理



### MyBatis整合Spring原理

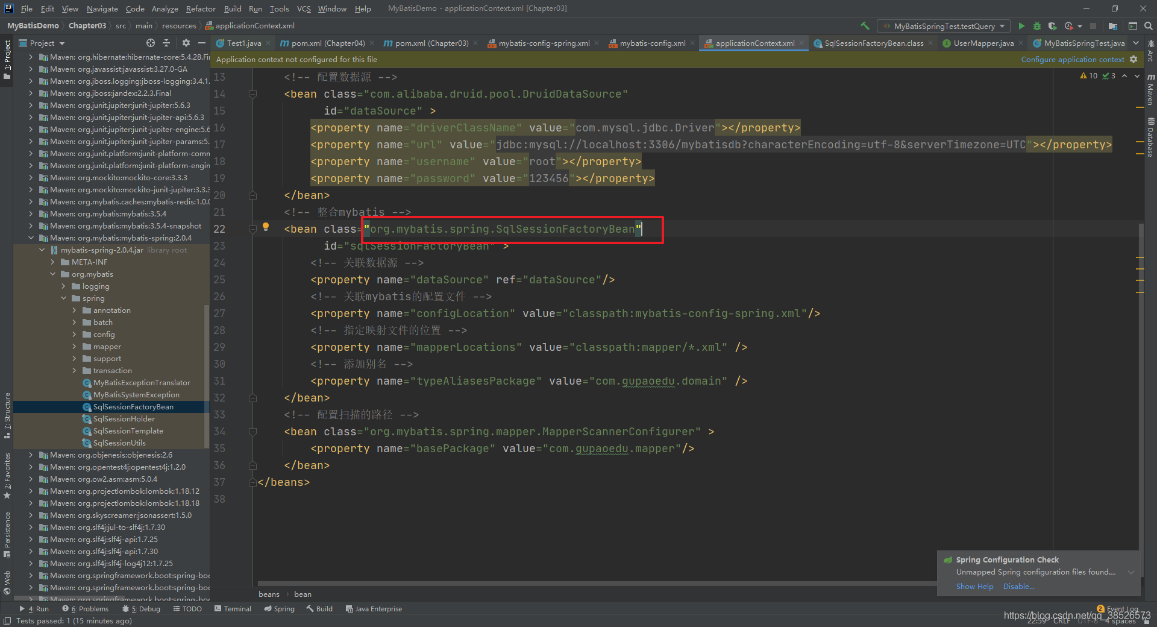
  把MyBatis集成到Spring里面，是为了进一步简化MyBatis的使用，所以只是对MyBatis做了一些封装，并没有替换MyBatis的核心对象。也就是说：MyBatis jar包中的SqlSessionFactory、SqlSession、MapperProxy这些类都会用到。mybatis-spring.jar里面的类只是做了一些包装或者桥梁的工作。

  只要我们弄明白了这三个对象是怎么创建的，也就理解了Spring继承MyBatis的原理。我们把它分成三步：

1. SqlSessionFactory在哪创建的。
2. SqlSession在哪创建的。
3. 代理类在哪创建的。

### 1、SqlSessionFactory

  首先我们来看下在MyBatis整合Spring中SqlSessionFactory的创建过程，查看这步的入口在Spring的配置文件中配置整合的标签中



我们进入SqlSessionFactoryBean中查看源码发现，其实现了InitializingBean 、FactoryBean、ApplicationListener 三个接口

| 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 |
| ApplicationListener | onApplicationEvent() | 对应用的事件进行监听 |

1.1 afterPropertiesSet

  我们首先来看下 afterPropertiesSet 方法中的逻辑

public void afterPropertiesSet() throws Exception {

Assert.notNull(this.dataSource, "Property 'dataSource' is required");

Assert.notNull(this.sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is required");

Assert.state(this.configuration == null && this.configLocation == null || this.configuration == null || this.configLocation == null, "Property 'configuration' and 'configLocation' can not specified with together");

this.sqlSessionFactory = this.buildSqlSessionFactory();

}

可以发现在afterPropertiesSet中直接调用了buildSqlSessionFactory方法来实现 sqlSessionFactory 对象的创建，在afterPropertiesSet方法中完成了SqlSessionFactory对象的创建，已经相关配置文件和映射文件的解析操作。

**方法小结一下**：通过定义一个实现了InitializingBean接口的SqlSessionFactoryBean类，里面有一个afterPropertiesSet()方法会在bean的属性值设置完的时候被调用。Spring在启动初始化这个Bean的时候，完成了解析和工厂类的创建工作。

1.2 getObject

  另外SqlSessionFactoryBean实现了FactoryBean接口。

  FactoryBean的作用是让用户可以自定义实例化Bean的逻辑。如果从BeanFactory中根据Bean的ID获取一个Bean，它获取的其实是FactoryBean的getObject()返回的对象。

  也就是说，我们获取SqlSessionFactoryBean的时候，就会调用它的getObject()方法。

public SqlSessionFactory getObject() throws Exception {

if (this.sqlSessionFactory == null) {

this.afterPropertiesSet();

}

return this.sqlSessionFactory;

}

getObject方法中的逻辑就非常简单，返回SqlSessionFactory对象，如果SqlSessionFactory对象为空的话就又调用一次afterPropertiesSet来解析和创建一次。

1.3 onApplicationEvent

  实现ApplicationListener接口让SqlSessionFactoryBean有能力监控应用发出的一些事件通知。比如这里监听了ContextRefreshedEvent（上下文刷新事件），会在Spring容器加载完之后执行。这里做的事情是检查ms是否加载完毕。

public void onApplicationEvent(ApplicationEvent event) {

if (this.failFast && event instanceof ContextRefreshedEvent) {

this.sqlSessionFactory.getConfiguration().getMappedStatementNames();

}

}

2、SqlSession

#### **2.1 DefaultSqlSession的问题**

  在前面介绍MyBatis的使用的时候，通过SqlSessionFactory的open方法获取的是DefaultSqlSession，但是在Spring中我们不能直接使用DefaultSqlSession，因为DefaultSqlSession是线程不安全的。所以直接使用会存在数据安全问题，针对这个问题的，在整合的MyBatis-Spring的插件包中给我们提供了一个对应的工具SqlSessionTemplate。

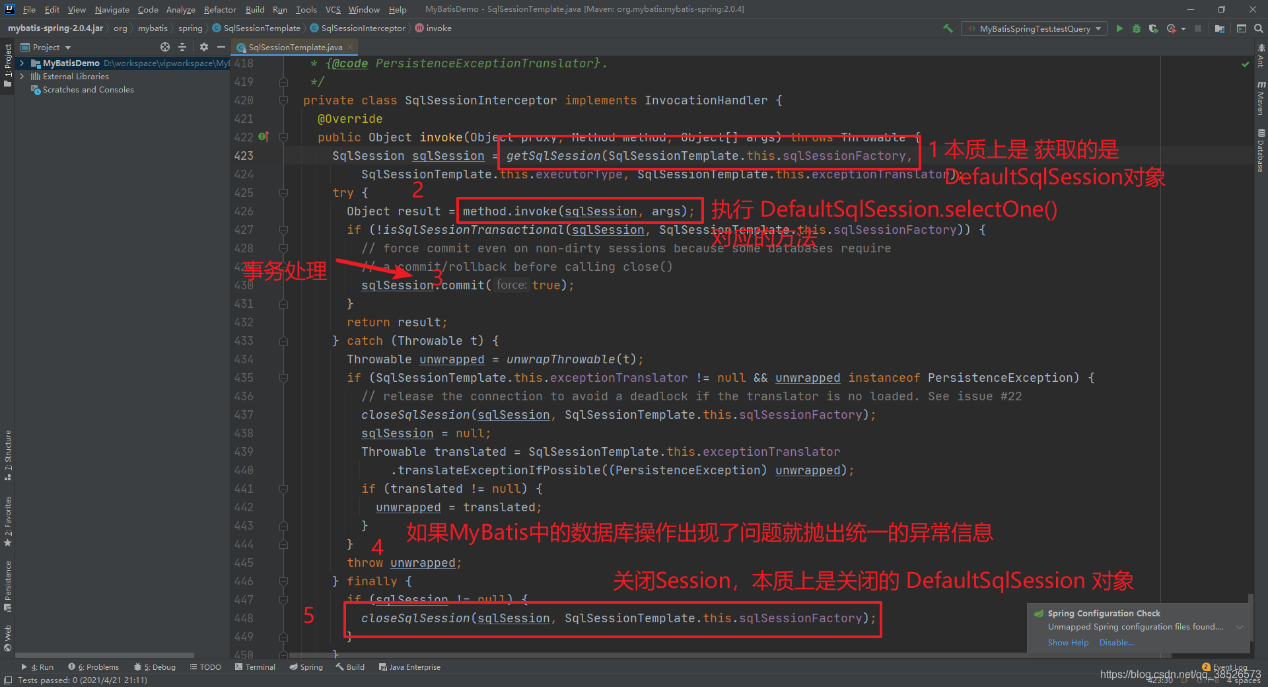
**2.2 SqlSessionTemplate**

  在mybatis-spring的包中，提供了一个线程安全的SqlSession的包装类，用来替代SqlSession，这个类就是SqlSessionTemplate。因为它是线程安全的，所以可以在所有的DAO层共享一个实例（默认是单例的）。

  SqlSessionTemplate虽然跟DefaultSqlSession一样定义了操作数据的selectOne()、selectList()、insert()、update()、delete()等所有方法，但是没有自己的实现，全部调用了一个代理对象的方法。

那么SqlSessionProxy是怎么来的呢？在SqlSessionTemplate的构造方法中有答案

通过上面的介绍那么我们应该进入到 SqlSessionInterceptor 的 invoke 方法中。



上面的代码虽然看着比较复杂，但是本质上就是下面的操作

SqlSession session = null;

try {

session = sqlSessionFactory.openSession();

// 你的应用逻辑代码

}finally{

session.close();

}

**2.3 SqlSessionDaoSupport**

  通过上面的介绍我们清楚了在Spring项目中我们应该通过SqlSessionTemplate来执行数据库操作，那么我们就应该首先将SqlSessionTemplate添加到IoC容器中，然后我们在Dao通过@Autowired来获取

**2.4 MapperScannerConfigurer**

  上面我们介绍了SqlSessionTemplate和SqlSessionDaoSupport，也清楚了他们的作用，但是我们在实际开发的时候，还是能够直接获取到 Mapper 的代理对象，并没有创建Mapper的实现类，这个到底是怎么实现的呢？这个我们就要注意在整合MyBatis的配置文件中除了SqlSessionFactoryBean以外我们还设置了一个MapperScannerConfigurer，我们来分析下这个类

MapperScannerConfigurer实现了BeanDefinitionRegistryPostProcessor接口。BeanDefinitionRegistryPostProcessor 是BeanFactoryPostProcessor的子类，里面有一个postProcessBeanDefinitionRegistry()方法。

  实现了这个接口，就可以在Spring创建Bean之前，修改某些Bean在容器中的定义。Spring创建Bean之前会调用这个方法。

**2.5 MapperFactoryBean**

  为什么要注册成它呢？那注入使用的时候，也是这个对象，这个对象有什么作用？首先来看看他们的类图结构

从类图中我们可以看到MapperFactoryBean继承了SqlSessionDaoSupport,那么每一个注入Mapper的地方，都可以拿到SqlSessionTemplate对象了。然后我们还发现MapperFactoryBean实现了 FactoryBean接口，也就意味着，向容器中注入MapperFactoryBean对象的时候，本质上是把getObject方法的返回对象注入到了容器中，它并没有直接返回一个MapperFactoryBean。而是调用了SqlSessionTemplate的getMapper()方法。SqlSessionTemplate的本质是一个代理，所以它最终会调用DefaultSqlSession的getMapper()方法。后面的流程我们就不重复了。也就是说，最后返回的还是一个JDK的动态代理对象。

  所以最后调用Mapper接口的任何方法，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

**总结一下**，Spring是怎么把MyBatis继承进去的？

1. 提供了SqlSession的替代品SqlSessionTemplate，里面有一个实现了实现了InvocationHandler的内部SqlSessionInterceptor，本质是对SqlSession的代理。
2. 提供了获取SqlSessionTemplate的抽象类SqlSessionDaoSupport。
3. 扫描Mapper接口，注册到容器中的是MapperFactoryBean，它继承了SqlSessionDaoSupport，可以获得SqlSessionTemplate。
4. 把Mapper注入使用的时候，调用的是getObject()方法，它实际上是调用了SqlSessionTemplate的getMapper()方法，注入了一个JDK动态代理对象。
5. 执行Mapper接口的任意方法，会走到触发管理类MapperProxy，进入SQL处理流程。

**核心对象：**

| 对象 | 生命周期 |
| --- | --- |
| SqlSessionTemplate | Spring中SqlSession的替代品，是线程安全的 |
| SqlSessionDaoSupport | 用于获取SqlSessionTemplate |
| SqlSessionInterceptor（内部类） | 代理对象，用来代理DefaultSqlSession，在SqlSessionTemplate中使用 |
| MapperFactoryBean | 代理对象，继承了SqlSessionDaoSupport用来获取SqlSessionTemplate |
| SqlSessionHolder | 控制SqlSession和事务 |