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。

1. MyBatis插件原理与自定义插件

1.1. 插件的使用

运行自定义的插件，需要3步，我们以PageHelper为例：

1.1.1. 插件编写

spring-mybatis 工程

编写拦截器类，需要做的有三点：

1.1.1.1. 1）实现Interceptor接口

```
1.  public class PageInterceptor implements Interceptor {
2.  }
```

1.1.1.2. 2）实现相应的方法。

最关键的是intercept()方法里面是拦截的逻辑，需要增强的功能代码就写在这里。

```
public interface Interceptor {

  Object intercept(Invocation invocation) throws Throwable;

  default Object plugin(Object target) {
    return Plugin.wrap(target, this);
  }

  default void setProperties(Properties properties) {
    // NOP
  }

}
```

1.1.1.3. 3）在拦截器类加上注解。

注解签名指定了需要拦截的对象、拦截的方法、参数（因为方法有不同的重载，所以要指定具体的参数）。

例如，这里拦截Executor中的两个query()方法（实际上是同一个）。

```
@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 {
```

1.1.2. 插件配置

在mybatis-config.xml中注册插件，配置属性。

```
    <!--分页插件的注册-->
    <plugins>
        <plugin interceptor="com.github.pagehelper.PageInterceptor">
            <!-- 4.0.0以后版本可以不设置该参数 ,可以自动识别
            <property name="dialect" value="mysql"/>  -->
            <!-- 该参数默认为false -->
            <!-- 设置为true时，会将RowBounds第一个参数offset当成pageNum页码使用 -->
            <!-- 和startPage中的pageNum效果一样-->
            <property name="offsetAsPageNum" value="true"/>
            <!-- 该参数默认为false -->
            <!-- 设置为true时，使用RowBounds分页会进行count查询 -->
            <property name="rowBoundsWithCount" value="true"/>
            <!-- 设置为true时，如果pageSize=0或者RowBounds.limit = 0就会查询出全部的结果 -->
            <!-- （相当于没有执行分页查询，但是返回结果仍然是Page类型）-->
            <property name="pageSizeZero" value="true"/>
            <!-- 3.3.0版本可用 - 分页参数合理化，默认false禁用 -->
            <!-- 启用合理化时，如果pageNum<1会查询第一页，如果pageNum>pages会查询最后一页 -->
            <!-- 禁用合理化时，如果pageNum<1或pageNum>pages会返回空数据 -->
            <property name="reasonable" value="true"/>
            <!-- 3.5.0版本可用 - 为了支持startPage(Object params)方法 -->
            <!-- 增加了一个`params`参数来配置参数映射，用于从Map或ServletRequest中取值 -->
            <!-- 可以配置pageNum,pageSize,count,pageSizeZero,reasonable,orderBy,不配置映射的用默认值 -->
            <!-- 不理解该含义的前提下，不要随便复制该配置 -->
            <property name="params" value="pageNum=start;pageSize=limit;"/>
            <!-- 支持通过Mapper接口参数来传递分页参数 -->
            <property name="supportMethodsArguments" value="true"/>
            <!-- always总是返回PageInfo类型,check检查返回类型是否为PageInfo,none返回Page -->
            <property name="returnPageInfo" value="check"/>
        </plugin>
    </plugins>
```

1.1.3. 插件解析注册

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。

XMLConfigBuilder.pluginElement（）：

```
  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).getDeclaredConstructor().newInstance();
        interceptorInstance.setProperties(properties);
        configuration.addInterceptor(interceptorInstance);
      }
    }
  }
```

在启动解析的时候，把所有的插件全部存到了Configuration的InterceptorChain中，它是一个List。

```
public class InterceptorChain {

  private final List<Interceptor> interceptors = new ArrayList<>();

  public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) {
      target = interceptor.plugin(target);
    }
    return target;
  }

  public void addInterceptor(Interceptor interceptor) {
    interceptors.add(interceptor);
  }

  public List<Interceptor> getInterceptors() {
    return Collections.unmodifiableList(interceptors);
  }

}
```

1.2. 猜想

MyBatis的插件不用修改原jar包的代码，就可以改变核心对象的行为，比如在前面处理参数，在中间处理SQL，在最后处理结果集。

1.2.1. 不修改代码怎么增强功能?

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?

很容易能想到用代理模式，这个也确实是MyBatis插件的实现原理。

1.2.2. 多个插件怎么拦截?

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，执行完了第一个插件的逻辑，要继续执行第二个第三个插件的逻辑。

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？如果是代理模式，已经被代理过的对象，可以再次被代理吗?

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。

1.2.3. 什么对象可以被拦截?

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?

并不是每一个运行的节点都是可以被修改的（如果没有这种规范，就会造成混乱和带来风险）。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。

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

对象		可拦截的方法	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

这里需要注意的是，因为Executor 有可能被二级缓存装饰，那么是先代理再装饰，还是先装饰后代理呢?

Executor 会拦截到CachingExcecutor 或者BaseExecutor。

DefaultSqlSessionFactory.openSessionFromDataSource()：

```
  private SqlSession openSessionFromDataSource(ExecutorType execType,
TransactionIsolationLevel level, boolean autoCommit) {
     ...
      final Executor executor = configuration.newExecutor(tx, execType);
    ...
  }
```

```
  public Executor newExecutor(Transaction transaction, ExecutorType executorType)
{
     ...
    if (ExecutorType.BATCH == executorType) {
      executor = new BatchExecutor(this, transaction);
    } else if (ExecutorType.REUSE == executorType) {
      executor = new ReuseExecutor(this, transaction);
    } else {
      executor = new SimpleExecutor(this, transaction);
    }
    if (cacheEnabled) {
      executor = new CachingExecutor(executor);
    }
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
  }
```

先创建基本类型，再二级缓存装饰，最后插件拦截。所以这里拦截的是CachingExecutor。

1.3. 插件实现原理

如果是代理模式，又有几个问题需要解决：
1、代理类怎么创建？动态代理是JDK Proxy还是Cglib？怎么样创建代理？
2、代理类在什么时候创建？是在解析配置的时候创建，还是获取会话的时候创建，还是在调用的时候创建？
3、核心对象被代理之后，调用的流程是怎么样的？怎么依次执行多个插件的逻辑？在执行完了插件的逻辑之后，怎么执行原来的逻辑？

只要理解了代理模式在这里的运用，就理解了插件的工作流程。

1.3.1. 代理类什么时候创建?

对Executor拦截的代理类是openSession（）的时候创建的。

Configuration.newExecutor（）

```
  public Executor newExecutor(Transaction transaction, ExecutorType executorType) {
    ...
    executor = (Executor) interceptorChain.pluginAll(executor);
    return executor;
  }
```

StatementHandler 是SimpleExecutor.doQuery（）创建的；里面包含了ParameterHandler 和ResultSetHandler的创建和代理。

```
  @Override
  public <E> List<E> doQuery(MappedStatement ms, Object parameter, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) throws SQLException {
    Statement stmt = null;
    try {
      Configuration configuration = ms.getConfiguration();
      StatementHandler handler = configuration.newStatementHandler(wrapper, ms, parameter, rowBounds, resultHandler, boundSql);
      stmt = prepareStatement(handler, ms.getStatementLog());
      return handler.query(stmt, resultHandler);
    } finally {
      closeStatement(stmt);
    }
  }
```

Configuration.newStatementHandler（）、newParameterHandler()、newResultSetHandler（）：

```
  protected BaseStatementHandler(Executor executor, MappedStatement mappedStatement, Object parameterObject, RowBounds rowBounds, ResultHandler resultHandler, BoundSql boundSql) {
    ...
    this.parameterHandler = configuration.newParameterHandler(mappedStatement, parameterObject, boundSql);
    this.resultSetHandler = configuration.newResultSetHandler(executor, mappedStatement, rowBounds, parameterHandler, resultHandler, boundSql);
  }
```

1.3.2. 代理怎么创建?

调用InterceptorChain的pluginAll()方法，做了什么事?

```
  public Object pluginAll(Object target) {
    for (Interceptor interceptor : interceptors) {
      target = interceptor.plugin(target);
    }
    return target;
  }
```

遍历InterceptorChain，使用Interceptor 实现类的plugin（）方法，对目标核心对象进行代理。

```
 public Object plugin(Object target) {
    // 如何实现？
  }
```

这个plugin方法是我们自己实现的，要返回一个代理对象。

如果是JDK动态代理，那我们必须要写一个实现了InvocationHandler接口的触发管理类。然后用Proxy.newProxyInstance（）创建一个代理对象。

Mybatis的插件机制已经把这些类封装好了，它已经提供了一个触发管理类Plugin实现了InvocationHandler。
创建代理对象的newProxyinstance()在这个类里面也进行了封装，就是wrap()方法。

```
  public static Object wrap(Object target, Interceptor interceptor) {
    Map<Class<?>, Set<Method>> signatureMap = getSignatureMap(interceptor);
    Class<?> type = target.getClass();
    Class<?>[] interfaces = getAllInterfaces(type, signatureMap);
    if (interfaces.length > 0) {
      return Proxy.newProxyInstance(
          type.getClassLoader(),
          interfaces,
          new Plugin(target, interceptor, signatureMap));
    }
    return target;
  }
```

在wrap的时候创建了一个Plugin对象，Plugin 是被代理对象、Interceptor的一个封装对象：

```
new Plugin(target, interceptor, signatureMap)
```

持有了被代理对象和interceptor的实例：

```
  private Plugin(Object target, Interceptor interceptor, Map<Class<?>,
Set<Method>> signatureMap) {
    this.target = target;
    this.interceptor = interceptor;
    this.signatureMap = signatureMap;
  }
```

因为这里是for循环代理，所以某个核心对象有多个插件，会返回被代理多次的代理对象。

1.3.3. 被代理之后，调用的流程?

在四大核心对象的一次执行过程中（可能被代理多次），因为已经被代理了，所以会先走到触发管理类Plugin的invoke（）方法。

```
1.   @Override
2.   public Object invoke(Object proxy, Method method, Object[] args) throws
   Throwable {
3.     try {
4.       Set<Method> methods = signatureMap.get(method.getDeclaringClass());
5.       if (methods != null && methods.contains(method)) {
6.         return interceptor.intercept(new Invocation(target, method, args));
7.       }
8.       return method.invoke(target, args);
9.     } catch (Exception e) {
10.      throw ExceptionUtil.unwrapThrowable(e);
11.    }
12.  }
```

如果被拦截的方法不为空，进入Plugin的invoke（）方法，调用interceptor的intercept（）方法：

```
1.  return interceptor.intercept(new Invocation(target, method, args));
```

到了intercept（）方法，也就走到了我们自己实现的拦截逻辑（例如PageInterceptor的intercept()方法）。

注意参数是new出来的Invocation对象，它是对被拦截对象、被拦截方法、被拦截参数的一个封装。为什么要传这样一个参数呢?

```
1.   public Invocation(Object target, Method method, Object[] args) {
2.     this.target = target;
3.     this.method = method;
4.     this.args = args;
5.   }
```

当然，在代理逻辑执行完了之后，比如继续执行被代理对象（四大核心对象）的原方法，也就是说要有一行这样的代码：

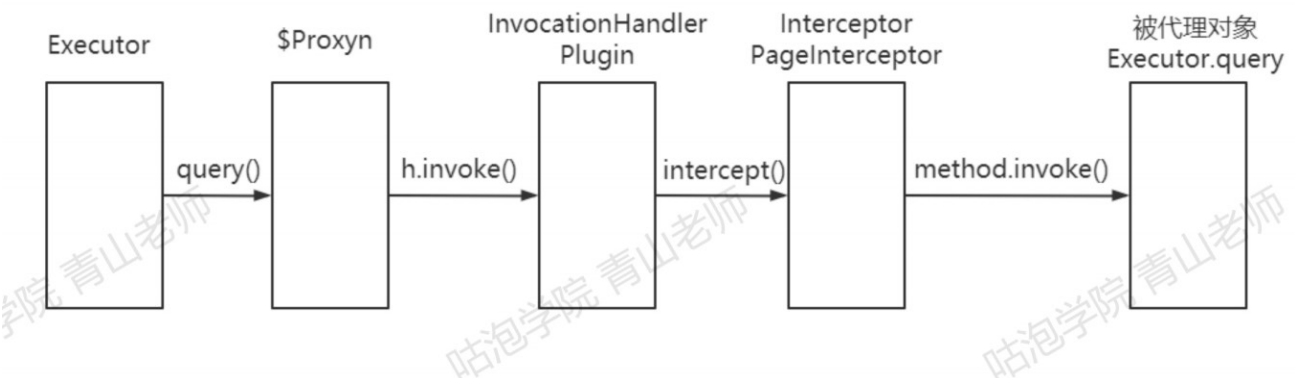
```
1.  return method.invoke(target, args);
```

或者拿到被代理的核心对象，继续执行它的方法（比如executor.query()）。这个时候，我们要拿到被代理对象和它的参数，去哪里拿?

就是上面创建的Invocation对象。它简化了参数的传递，而且直接提供了一个proceed（）方法，也就是说继续执行原方法可以写成：

```
1.  return invocation.proceed();
```

总结一下：

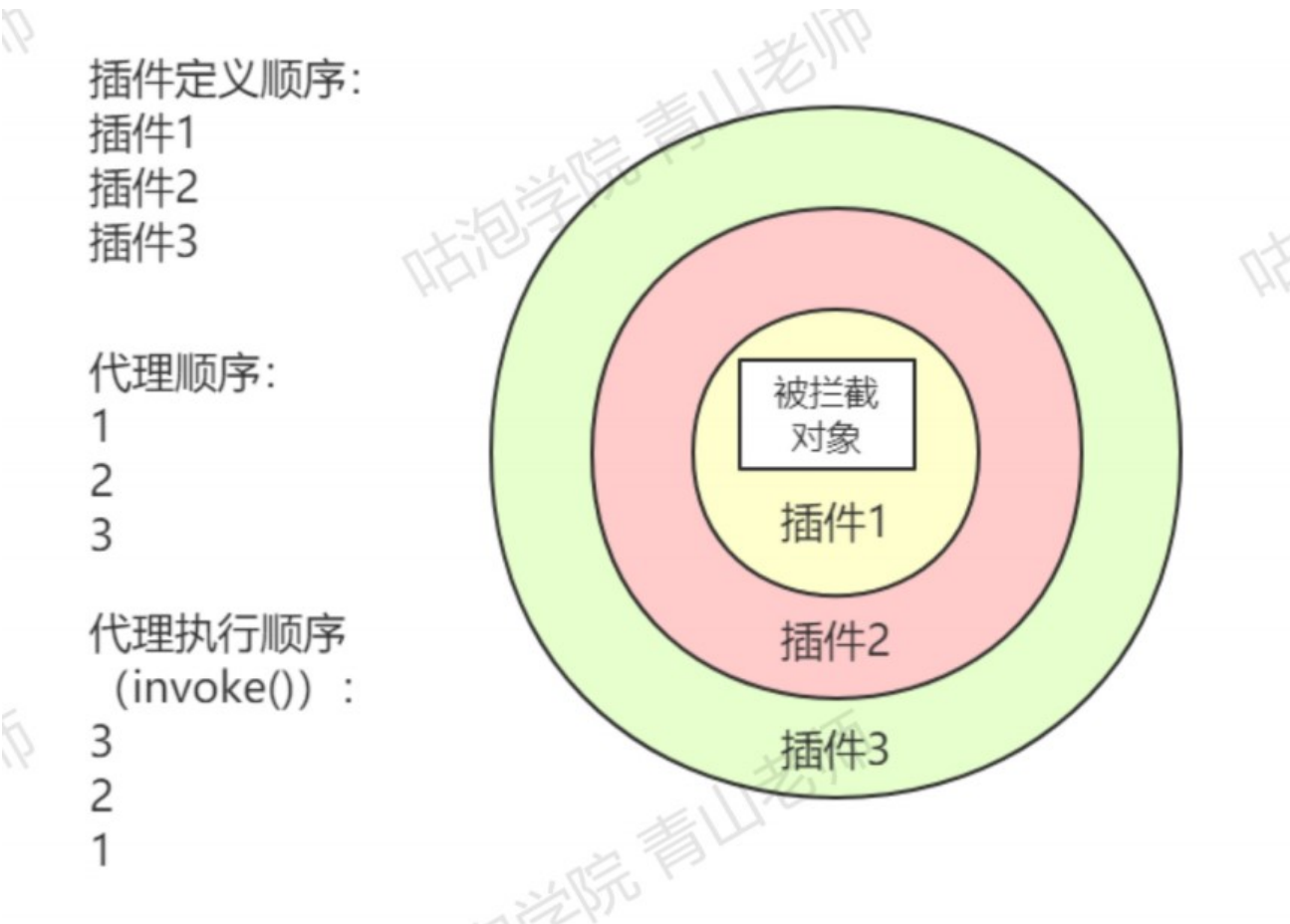


如果说对象被代理了多次，这里会继续调用下一个插件的逻辑，再走一次Plugin的invoke（）方法。这里我们需要关注一下有多个插件的时候的运行顺序。

1.3.4. 配置的顺序和执行的顺序?

配置的顺序和执行的顺序是相反的。InterceptorChain的List是按照插件从上往下的顺序解析、添加的。

而创建代理的时候也是按照 list的顺序代理。执行的时候当然是从最后代理的对象开始。



总结:

对象	作用
Interceptor	自定义插件需要实现接口，实现 4 个方法
InterceptChain	配置的插件解析后会保存在 Configuration 的 InterceptChain 中
Plugin	触发管理类，还可以用来创建代理对象
Invocation	对被代理类进行包装，可以调用 proceed() 调用到被拦截的方法

1.4. PageHelper原理

1.4.1. 逻辑翻页

使用RowBounds翻页，在内存中筛选数据。

> 参见mybatis-standalone3-lesson工程cn.sitedev.MyBatisTest

```
    @Test
    public void testSelectByRowBounds() throws IOException {
        SqlSession session = sqlSessionFactory.openSession();
        try {
            BlogMapper mapper = session.getMapper(BlogMapper.class);
            int start = 0; // offset
            int pageSize = 5; // limit
            RowBounds rb = new RowBounds(start, pageSize);
            List<Blog> list = mapper.selectBlogList(rb); // 使用逻辑分页
            for (Blog b : list) {
                System.out.println(b);
            }
        } finally {
            session.close();
        }
    }
```

```
    <select id="selectBlogList" resultMap="BaseResultMap">
        SELECT bid, name, author_id authorId from blog
    </select>
```

1.4.2. 使用和参数传递

在引入了pageHelper的依赖，配置了插件之后，如果我们需要翻页，需要用到相关的工具类：

> 参见spring-mybatis3-lesson工程cn.sitedev.crud.controller.EmployeeController

```
    /**
     * 查询员工数据 分页
     *
     * @param pn
     * @param model
     * @return
     * @RequestMapping("/emps")
     */
    public String getEmps(@RequestParam(value = "pn", defaultValue = "1") Integer pn, Model model) {
        PageHelper.startPage(pn, 10);
        List<Employee> emps = employeeService.getAll();
        PageInfo page = new PageInfo(emps, 10);
        //连续显示的页数是10页
        //包装查出来的结果，只需要将pageInfo交给页面，封装了详细的分页信息
        //包括查询出来的数据
        model.addAttribute("pageInfo", page);

        return "list";
    }
```

在PageHelper 类中指定页码和每页数量。MyBatis的方法不用做任何的修改，最后可以把结果包装成一个PageInfo返回给前端。

如果不需要翻页，把这两行代码去掉就行了。插件的优点就是不用修改MyBatis本身的代码。

1.4.3. SQL改写的实现

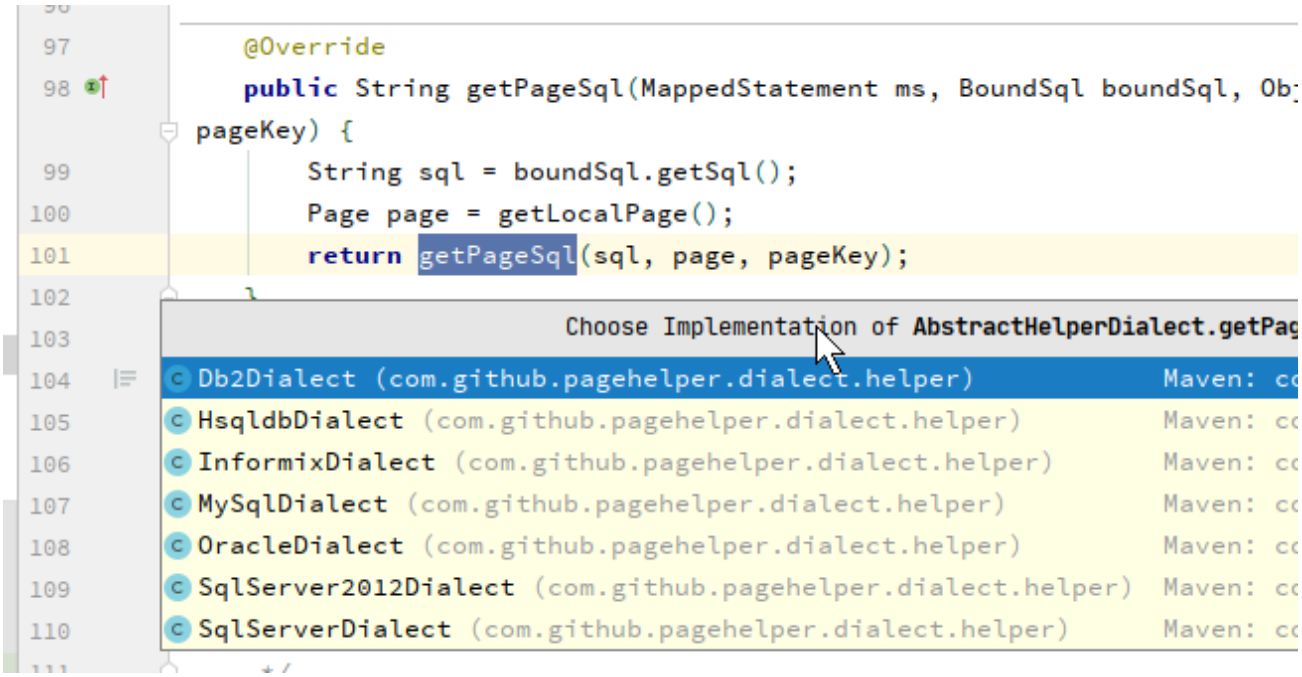
PageHelper 到底是怎么通过拦截实现翻页的呢？首先看一下拦截器，PageInterceptor类。

首先判断是否需要count 获取总数，默认是true。获得count之后，判断是否需要分页，如果pageSize>0，就分页。

这里通过 getPageSql()方法生成了一个新的BoundSql：

```
    @Override
    public Object intercept(Invocation invocation) throws Throwable {
        ...
                    String pageSql = dialect.getPageSql(ms, boundSql, parameter, rowBounds, pageKey);
                    BoundSql pageBoundSql = new BoundSql(ms.getConfiguration(), pageSql, boundSql.getParameterMappings(), parameter);
        ...
```

getPageSql（）对于不同的数据库有不同的实现：



以MySQL为例，实际上是添加了LIMIT语句，加上了起始位置和结束位置：

```
public class MySqlDialect extends AbstractHelperDialect {

    @Override
    public String getPageSql(String sql, Page page, CacheKey pageKey) {
        StringBuilder sqlBuilder = new StringBuilder(sql.length() + 14);
        sqlBuilder.append(sql);
        if (page.getStartRow() == 0) {
            sqlBuilder.append(" LIMIT ");
            sqlBuilder.append(page.getPageSize());
        } else {
            sqlBuilder.append(" LIMIT ");
            sqlBuilder.append(page.getStartRow());
            sqlBuilder.append(",");
            sqlBuilder.append(page.getPageSize());
            pageKey.update(page.getStartRow());
        }
        pageKey.update(page.getPageSize());
        return sqlBuilder.toString();
    }

}
```

问题是，插件是怎么获取到页码和每页数量，是怎么传递给插件的?

回头看看PageHelper.startPage（）方法，startPage（）调用了PageMethod的setLocalPage（）方法，包装了一个Page对象，并且把这个对象放到了ThreadLocal变量中。

```java
public abstract class PageMethod {
    protected static final ThreadLocal<Page> LOCAL_PAGE = new ThreadLocal<Page>();

    /**
     * 设置 Page 参数
     *
     * @param page
     */
    protected static void setLocalPage(Page page) {
        LOCAL_PAGE.set(page);
    }

    /**
     * 开始分页
     *
     * @param pageNum  页码
     * @param pageSize 每页显示数量
     */
    public static <E> Page<E> startPage(int pageNum, int pageSize) {
        return startPage(pageNum, pageSize, true);
    }

    /**
     * 开始分页
     *
     * @param pageNum  页码
     * @param pageSize 每页显示数量
     * @param count    是否进行count查询
     */
    public static <E> Page<E> startPage(int pageNum, int pageSize, boolean count)
{
        Page<E> page = new Page<E>(pageNum, pageSize, count);
        setLocalPage(page);
        return page;
    }
```

而在AbstractHelperDialect中，Page 对象中的翻页信息是通过 getLocalPage（）取出来的：

```java
    @Override
    public String getPageSql(MappedStatement ms, BoundSql boundSql, Object
parameterObject, RowBounds rowBounds, CacheKey pageKey) {
        String sql = boundSql.getSql();
        Page page = getLocalPage();
        return getPageSql(sql, page, pageKey);
    }
```

它调用的正是PageHelper的getLocalPage()，从ThreadLocal中获取到了翻页信息。

```java
public abstract class AbstractHelperDialect extends AbstractDialect {

    /**
     * 获取分页参数
     *
     * @param <T>
     * @return
     */
    public <T> Page<T> getLocalPage() {
        return PageHelper.getLocalPage();
    }
```

```java
public abstract class PageMethod {
    protected static final ThreadLocal<Page> LOCAL_PAGE = new ThreadLocal<Page>();

    /**
     * 获取 Page 参数
     *
     * @return
     */
    public static <T> Page<T> getLocalPage() {
        return LOCAL_PAGE.get();
    }
```

所以，每次查询（每一个线程）都会有一个线程私有的Page对象，它里面有页码和每页数量。

1.4.4. 关键类

对象	作用
PageInterceptor	自定义拦截器，实现了 Interceptor 接口
Page	包装分页参数，比如每页数量，当前页码等等
PageInfo	包装结果
PageHelper	工具类，设置翻页信息

1.5. 应用场景分析

作用	描述	实现方式
水平分表	一张费用表按月度拆分为 12 张表。fee_202001-202012。 当查询条件出现月度（tran_month）时，把 select 语句中的逻辑表名修改为对应的月份表。	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
数据脱敏	手机号和身份证在数据库完整存储。但是返回给用户，屏蔽手机号的中间四位。屏蔽身份证号中的出生日期。	query——对结果集脱敏
菜单权限控制	不同的用户登录，查询菜单权限表时获得不同的结果，在前端展示不同的菜单	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

2. 与Spring整合分析

官网：

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

https://github.com/mybatis/spring

在MyBatis的原生API中，有三个对外提供的核心对象：SqlSessionFactory、SqlSession、getMapper（）返回的代理对象（里面有一个h对象MapperProxy的实例）。

> 参见mybatis-standalone3-lesson工程cn.sitedev.MyBatisTest

```
1.      private SqlSessionFactory sqlSessionFactory;
2.
3.      @Before
4.      public void prepare() throws IOException {
5.          String resource = "mybatis-config.xml";
6.          InputStream inputStream = Resources.getResourceAsStream(resource);
7.          sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
8.      }
9.
10.     @Test
11.     public void testSelect() throws IOException {
12.         SqlSession session = sqlSessionFactory.openSession(); //
    ExecutorType.BATCH
13.         try {
14.             BlogMapper mapper = session.getMapper(BlogMapper.class);
15.             Blog blog = mapper.selectBlogById(1);
16.             System.out.println(blog);
17.         } finally {
18.             session.close();
19.         }
20.     }
```

虽然MyBatis对JDBC进行封装之后，已经大大地简化了我们对于数据库的操作，但是在我们的业务代码里面，不断地创建、释放 SqlSession也是一件很麻烦的事情。

所以，在Spring的工程里面，有没有更加简单的使用MyBatis的方法呢?

实际上，MyBatis 基于Spring的扩展接口，对原生API中的操作进一步进行了简化，这个也是为什么为什么在Spring 里面使用MyBatis的时候，没有看到这三个对象在代码里面的出现的原因。我们只需要把Mapper接口注入到需要的地方，调用它的方法就 OK了

> 参见spring-mybatis3-lesson工程cn.sitedev.crud.service.EmployeeService

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

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

这里我们以传统的Spring XML配置为例来分析一下MyBatis 集成到Spring的原理，因为XML的配置封装更少，更直观。

当然，使用注解的效果和本质都是一样的，对于Spring来说只是解析方式的差异。

在Spring中，有几个关键的问题，我们要弄清楚：
1、如果用@Autowired注入一个接口，调用接口的方法就可以找到SQL执行，这个接口在IoC容器中也是一个代理对象吗?
2、如果是代理对象，还是不是SqlSession用getMapper()获得的代理对象？SqlSession又是什么时候创建的?
3、每个会话都要产生一个SqlSession，单例的SqlSessionFactory 是什么时候创建的?

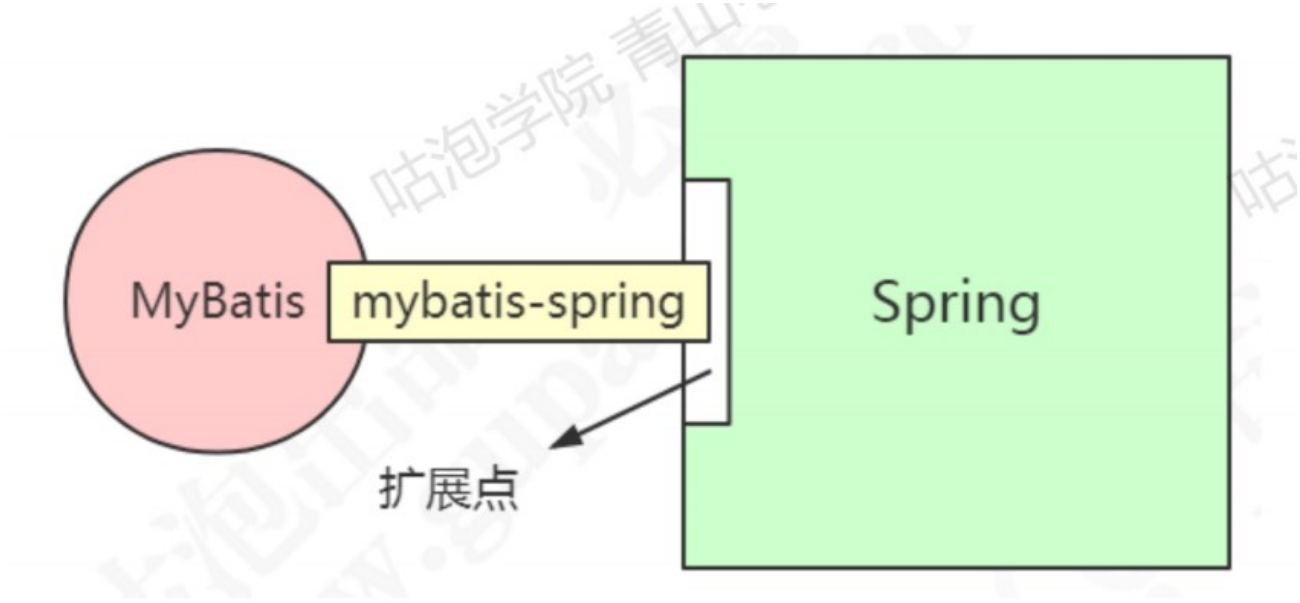
2.1. 1、关键配置

我们先回到用户的视角，看看把MyBatis 集成到Spring中要做的几件事情。

2.1.1. pom 依赖

首先，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。

这里要注意两点：这个包名叫mybatis-spring，而不是spring-mybatis，是因为它是MyBatis 利用Spring的接口开发的。



其次：mybatis的版本和mybatis-spring的版本有兼容关系，版本要对应。

```
        <!--mybatis 和Spring整合 -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis-spring</artifactId>
            <version>2.0.4</version>
        </dependency>

        <!-- mybatis -->
        <dependency>
            <groupId>org.mybatis</groupId>
            <artifactId>mybatis</artifactId>
            <version>3.5.1</version>
        </dependency>
```

2.1.2. SqlSessionFactoryBean

然后在Spring的applicationContext.xml 里面配置 SqlSessionFactoryBean。很明显，这个Bean会初始化一个SqlSessionFactory，用来帮我们创建 SqlSession。

它的属性还要指定全局配置文件mybatis-config.xml和Mapper映射器文件的路径。数据源也是由Spring来管理。

```
    <!-- 在Spring启动时创建 sqlSessionFactory -->
    <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>
```

2.1.3. MapperScannerConfigurer

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

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="cn.sitedev.crud.dao"/>
    </bean>
```

第二种是配置一个 `<scan>` 标签：

```
    <mybatis-spring:scan base-package="cn.sitedev.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("cn.sitedev.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。

经过这两步（SqlSessionFactoryBean+MapperScannerConfigurer）配置以后，Mapper 就可以注入到Service层使用了，MyBatis其他的代码和配置不需要做任何的改动。

它是怎么实现的呢?

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。

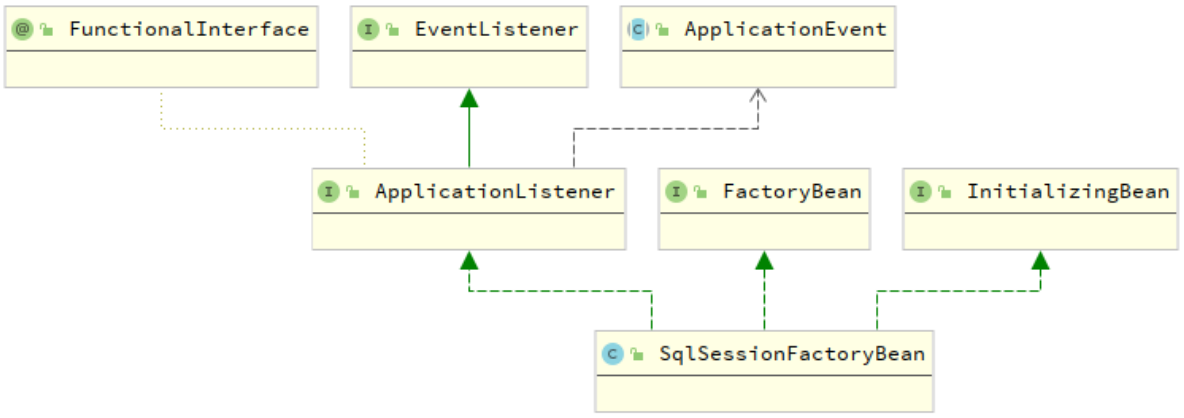
2.2. 2、创建会话工厂SqlSessionFactory

第一步，我们看一下在Spring里面，SqlSessionFactory 是怎么创建的。

我们在Spring的配置文件中配置了一个SqlSessionFactoryBean，看下这个类的内容：

```
public class SqlSessionFactoryBean
    implements FactoryBean<SqlSessionFactory>, InitializingBean,
ApplicationListener<ApplicationEvent> {
```

它实现了三个接口：InitializingBean、FactoryBean、ApplicationListener。



实现这三个接口意味着什么呢?

2.2.1. InitializingBean

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。

```
  @Override
  public void afterPropertiesSet() throws Exception {
    notNull(dataSource, "Property 'dataSource' is required");
    notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is
required");
    state((configuration == null && configLocation == null) || !(configuration !=
null && configLocation != null),
        "Property 'configuration' and 'configLocation' can not specified with
together");

    this.sqlSessionFactory = buildSqlSessionFactory();
  }
```

在afterPropertiesSet()方法里面，经过一些检查之后，调用了buildSqlSessionFactory（）方法。

这里创建了一个Configuration对象，叫做targetConfiguration。还创建了一个用来解析全局配置文件的XMLConfigBuilder。

```
  protected SqlSessionFactory buildSqlSessionFactory() throws Exception {

    final Configuration targetConfiguration;

    XMLConfigBuilder xmlConfigBuilder = null;
      ...
```

接下来判断 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。

```
  protected SqlSessionFactory buildSqlSessionFactory() throws Exception {

      ...
    if (this.configuration != null) {
      targetConfiguration = this.configuration;
      if (targetConfiguration.getVariables() == null) {
        targetConfiguration.setVariables(this.configurationProperties);
      } else if (this.configurationProperties != null) {
        targetConfiguration.getVariables().putAll(this.configurationProperties);
      }
        ...
```

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。

```
  protected SqlSessionFactory buildSqlSessionFactory() throws Exception {
    ...
    } else if (this.configLocation != null) {
      xmlConfigBuilder = new
XMLConfigBuilder(this.configLocation.getInputStream(), null,
this.configurationProperties);
      targetConfiguration = xmlConfigBuilder.getConfiguration();
        ...
```

applicationContext.xml

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

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。

```
  protected SqlSessionFactory buildSqlSessionFactory() throws Exception {
    ...
    } else {
      LOGGER.debug(
          () -> "Property 'configuration' or 'configLocation' not specified, using default MyBatis Configuration");
      targetConfiguration = new Configuration();

Optional.ofNullable(this.configurationProperties).ifPresent(targetConfiguration::setVariables);
    }
    ...
```

后面就是基于当前SqlSessionFactoryBean对象里面已有的属性，对targetConfiguration 对象里面属性的赋值。
Optional.ofNullable（）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，把解析出来的属性，赋值给SqlSessionFactoryBean的属性。

```
  protected SqlSessionFactory buildSqlSessionFactory() throws Exception {
      ...

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

Optional.ofNullable(this.objectWrapperFactory).ifPresent(targetConfiguration::setObjectWrapperFactory);
    Optional.ofNullable(this.vfs).ifPresent(targetConfiguration::setVfsImpl);
      ...
```

后面对于typeAliases、plugins、typeHandlers的解析，都是调用Configuration类的方法。

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。

```
  protected SqlSessionFactory buildSqlSessionFactory() throws Exception {
      ...
    if (xmlConfigBuilder != null) {
      try {
        xmlConfigBuilder.parse();
        LOGGER.debug(() -> "Parsed configuration file: '" + this.configLocation +
"'");
      } catch (Exception ex) {
        throw new NestedIOException("Failed to parse config resource: " +
this.configLocation, ex);
      } finally {
        ErrorContext.instance().reset();
      }
    }
      ...
```

这个方法在讲解源码的时候已经详细分析过了，这里不再重复。

先判断事务工厂是否存在。这个事务工厂，是解析 environments标签的时候，根据 `<transactionManager>` 子标签创建的。

```
<transactionManager type="JDBC"/>
```

只有JDBC或者MANAGED这两个值。如果没有配置，比如数据源交给Spring管理的时候，这里就是空的。

此时会new一个SpringManagedTransactionFactory，放在Environment里面。

这个事务工厂获取的事务是SpringManagedTransaction对象，定义了getConnection()、close()、commit（）、rollback()等方法。

```
    targetConfiguration.setEnvironment(new Environment(this.environment,
        this.transactionFactory == null ? new SpringManagedTransactionFactory() :
this.transactionFactory,
        this.dataSource));
```

创建了一个用来解析 Mapper.xml的XMLMapperBuilder，调用了它的parse（）方法。这个步骤我们之前了解过了，主要做了两件事情，一个是把增删改查标签注册成MappedStatement对象。第二个是把接口和对应的MapperProxyFactory工厂类注册到MapperRegistry中。

```
            XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(mapperLocation.getInputStream(),
                targetConfiguration, mapperLocation.toString(),
targetConfiguration.getSqlFragments());
            xmlMapperBuilder.parse();
```

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

```
    return this.sqlSessionFactoryBuilder.build(targetConfiguration);
```

总结一下：通过定义一个实现了InitializingBean 接口的SaqlSessionFactoryBean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。

2.2.2. FactoryBean

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。

```
  @Override
  public SqlSessionFactory getObject() throws Exception {
    if (this.sqlSessionFactory == null) {
      afterPropertiesSet();
    }

    return this.sqlSessionFactory;
  }
```

而 getObject()方法也是调用了afterPropertiesSet（）方法，去做MyBatis 解析配置文件的工作，返回一个DefaultSqlSessionFactory。

2.2.3. `ApplicationListener<ApplicationEvent>`

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。

```
  @Override
  public void onApplicationEvent(ApplicationEvent event) {
    if (failFast && event instanceof ContextRefreshedEvent) {
      // fail-fast -> check all statements are completed
      this.sqlSessionFactory.getConfiguration().getMappedStatementNames();
    }
  }
```

SqlSessionFactoryBean 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
ApplicationListener	onApplicationEvent()	对应用的时间进行监听

2.3. 3、创建会话SqlSession

2.3.1. 为什么不能直接使用DefaultSqlSession?

我们现在已经有一个DefaultSqlSessionFactory，按照编程式的开发过程，我们接下来就会用openSession（）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。为什么不用DefaultSqlSession？它是线程不安全的，注意看类上的注释：

```
/**
 * The default implementation for {@link SqlSession}.
 * Note that this class is not Thread-Safe.
 *
 * @author Clinton Begin
 */
public class DefaultSqlSession implements SqlSession {
```

https://mybatis.org/mybatis-3/zh/getting-started.html

《作用域（Scope）和生命周期》一节也有提到：

> 每个线程都应该有它自己的SqlSession实例。SqlSession的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域。绝对不能将SqlSession实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。也绝不能将SqlSession实例的引用放在任何类型的托管作用域中，比如Servlet框架中的HttpSession。如果你现在正在使用一种Web框架，考虑将SqlSession放在一个和HTTP请求相似的作用域中。
> 换句话说，每次收到HTTP请求，就可以打开一个SqlSession，返回一个响应后，就关闭它。这个关闭操作很重要，为了确保每次都能执行关闭操作，你应该把这个关闭操作放到finally块中。

所以，在Spring里面，我们要保证 SqlSession实例的线程安全，必须为每一次请求单独创建一个SqlSession。但是每一次请求用openSession()自己去创建，又会比较麻烦。

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）。

```java
/**
 * Thread safe, Spring managed, {@code SqlSession} that works with Spring transaction management to ensure that that the
 * actual SqlSession used is the one associated with the current Spring transaction. In addition, it manages the session
 * life-cycle, including closing, committing or rolling back the session as necessary based on the Spring transaction
 * configuration.
 * <p>
 * The template needs a SqlSessionFactory to create SqlSessions, passed as a constructor argument. It also can be
 * constructed indicating the executor type to be used, if not, the default executor type, defined in the session
 * factory will be used.
 * <p>
 * This template converts MyBatis PersistenceExceptions into unchecked DataAccessExceptions, using, by default, a
 * {@code MyBatisExceptionTranslator}.
 * <p>
 * Because SqlSessionTemplate is thread safe, a single instance can be shared by all DAOs; there should also be a small
 * memory savings by doing this. This pattern can be used in Spring configuration files as follows:
 *
 * <pre class="code">
 * {@code
 * <bean id="sqlSessionTemplate" class="org.mybatis.spring.SqlSessionTemplate">
 *   <constructor-arg ref="sqlSessionFactory" />
 * </bean>
 * }
 * </pre>
 *
 * @author Putthiphong Boonphong
 * @author Hunter Presnall
 * @author Eduardo Macarron
 *
 * @see SqlSessionFactory
 * @see MyBatisExceptionTranslator
 */
public class SqlSessionTemplate implements SqlSession, DisposableBean {
```

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。

```
1.   @Override
2.   public void select(String statement, ResultHandler handler) {
3.     this.sqlSessionProxy.select(statement, handler);
4.   }
```

这个代理对象是怎么来的？在构造方法里面通过JDK动态代理创建：

```
1.   public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType
     executorType,
2.       PersistenceExceptionTranslator exceptionTranslator) {
3.
4.     notNull(sqlSessionFactory, "Property 'sqlSessionFactory' is required");
5.     notNull(executorType, "Property 'executorType' is required");
6.
7.     this.sqlSessionFactory = sqlSessionFactory;
8.     this.executorType = executorType;
9.     this.exceptionTranslator = exceptionTranslator;
10.    this.sqlSessionProxy = (SqlSession)
     newProxyInstance(SqlSessionFactory.class.getClassLoader(),
11.        new Class[] { SqlSession.class }, new SqlSessionInterceptor());
12.  }
```

它是对SqlSession 实现类DefaultSqlSession的代理。既然是JDK动态代理，那对代理类任意方法的调用都会走到（第三个参数）实现了InvocationHandler接口的触发管理类SqlSessionInterceptor的invoke()方法。

SqlSessionInterceptor 是一个内部类：

```
  private class SqlSessionInterceptor implements InvocationHandler {
    @Override
    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);
        if (!isSqlSessionTransactional(sqlSession, SqlSessionTemplate.this.sqlSessionFactory)) {
          // force commit even on non-dirty sessions because some databases require
          // a commit/rollback before calling close()
          sqlSession.commit(true);
        }
        return result;
      } catch (Throwable t) {
        Throwable unwrapped = unwrapThrowable(t);
        if (SqlSessionTemplate.this.exceptionTranslator != null && unwrapped instanceof PersistenceException) {
          // release the connection to avoid a deadlock if the translator is no loaded. See issue #22
          closeSqlSession(sqlSession, SqlSessionTemplate.this.sqlSessionFactory);
          sqlSession = null;
          Throwable translated = SqlSessionTemplate.this.exceptionTranslator
              .translateExceptionIfPossible((PersistenceException) unwrapped);
          if (translated != null) {
            unwrapped = translated;
          }
        }
        throw unwrapped;
      } finally {
        if (sqlSession != null) {
          closeSqlSession(sqlSession, SqlSessionTemplate.this.sqlSessionFactory);
        }
      }
    }
  }
```

这里会先用getSqlSession()方法创建一个SqlSession对象，把SqlSessionFactory、执行器类型、异常解析器传进去。

获得 SqlSession 实例（实际上是DefaultSqlSession）之后，再调用它的增删改查的方法。

总结一下：因为DefaultSqlSession自己做不到每次请求调用产生一个新的实例，我们干脆创建一个代理类，也实现SqlSession，提供跟 DefaultSqlSession一样的方法，在任何一个方法被调用的时候都先创建一个DefaultSqlSession实例，再调用被代理对象的相应方法。

MyBatis 还自带了一个线程安全的SqlSession实现：SqlSessionManager，实现方式一样，如果不集成到Spring要保证线程安全，就用SqlSessionManager。

跟JdbcTemplate，RedisTemplate 一样，SqlSessionTemplate 可以简化MyBatis在Spring中的使用，也是Spring跟MyBatis 整合的最关键的一个类。

2.3.2. 怎么拿到一个SqlSessionTemplate?

因为SqlSessionTemplate是线程安全的，可以替换DefaultSqlSession，那在DAO层怎么拿到一个SqlSessionTemplate呢?

在applicationContext.xml里面配置一个bean，用@Autowired注入到需要使用的地方不就好了?

但是，我们这里并没有显式地定义一个SqlSessionTemplate的bean，是不能直接注入的。所以，问题是，如果不用注入的方式，怎么获得一个SqlSession Template?

再new一个可以吗？也可以，它有三个重载的构造函数，例如：

```
  public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory) {
    this(sqlSessionFactory, sqlSessionFactory.getConfiguration().getDefaultExecutorType());
  }

  /**
   * Constructs a Spring managed SqlSession with the {@code SqlSessionFactory} provided as an argument and the given
   * {@code ExecutorType} {@code ExecutorType} cannot be changed once the {@code SqlSessionTemplate} is constructed.
   *
   * @param sqlSessionFactory
   *          a factory of SqlSession
   * @param executorType
   *          an executor type on session
   */
  public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType executorType) {
    this(sqlSessionFactory, executorType,
        new MyBatisExceptionTranslator(sqlSessionFactory.getConfiguration().getEnvironment().getDataSource(), true));
  }

  /**
   * Constructs a Spring managed {@code SqlSession} with the given {@code SqlSessionFactory} and {@code ExecutorType}. A
   * custom {@code SQLExceptionTranslator} can be provided as an argument so any {@code PersistenceException} thrown by
   * MyBatis can be custom translated to a {@code RuntimeException} The {@code SQLExceptionTranslator} can also be null
   * and thus no exception translation will be done and MyBatis exceptions will be thrown
   *
   * @param sqlSessionFactory
   *          a factory of SqlSession
   * @param executorType
   *          an executor type on session
   * @param exceptionTranslator
   *          a translator of exception
   */
  public SqlSessionTemplate(SqlSessionFactory sqlSessionFactory, ExecutorType executorType,
      PersistenceExceptionTranslator exceptionTranslator) {

    notNull(sqlSessionFactory, "Property 'sqlSessionFactory' is required");
    notNull(executorType, "Property 'executorType' is required");

    this.sqlSessionFactory = sqlSessionFactory;
```

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

new 出来是可以，但是这个单例的SqlSessionTemplate必须存起来放在一个地方，可以在任何需要替代DefaultSqlSession的地方都可以拿到，不能重复new，否则就不是单例了。

所以要存在一个什么地方呢？或者说，我们是不是提供一个工具类用来获取单例的SqlSessionTemplate呢？

在Hibernate中，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，它提供了一个抽象的支持类SqlSessionDaoSupport。

官网：http://mybatis.org/spring/zh/sqlsession.html

SqlSessionDaoSupport 类中持有一个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;
  }
```

也就是说我们让DAO层（实现类）继承抽象类SqlSessionDaoSupport，就自动拥有了getSqlSession()方法。调用getSqlSession（）就能拿到共享的SqlSessionTemplate。

在DAO层执行SQL格式如下：

```
getSqlSession().selectOne(statement, parameter);
getSqlSession().insert(statement);
getSqlSession().update(statement);
getSqlSession().delete(statement);
```

还是不够简洁。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。

```java
/**
 * 通过继承SqlSessionDaoSupport 获得一个 SqlSessionTemplate
 */
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) {
        return getSqlSession().selectOne(statement);
    }

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

然后让我们的DAO层实现类继承BaseDao并且实现我们的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。

```java
@Repository
public class EmployeeDaoImpl extends BaseDao implements EmployeeMapper {
    /**
     * 暂时只实现了这一个方法
     *
     * @param empId
     * @return
     */
    @Override
    public Employee selectByPrimaryKey(Integer empId) {
        Employee emp = (Employee)
this.selectOne("cn.sitedev.crud.dao.EmployeeMapper.selectByPrimaryKey", empId);
        return emp;
    }
```

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DaoSupportTest.java里面注入：

> 参见spring-mybatis3-lesson工程cn.sitedev.DaoSupportTest

```java
package cn.sitedev;

import cn.sitedev.crud.daosupport.EmployeeDaoImpl;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:applicationContext.xml"})
public class DaoSupportTest {
    @Autowired
    EmployeeDaoImpl employeeDao;

    @Test
    public void EmployeeDaoSupportTest() {

        System.out.println(employeeDao.selectByPrimaryKey(1));
    }

}
```

最终会调用到DefaultSqlSession的方法。

2.3.3. 有没有更好的拿到SqlSessionTemplate的方法?

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。

也就是说它没有用到JDK动态代理，在调用接口方法的时候通过MapperProxy自动找到对应的Statement ID。

怎么办呢？不继承SqlSessionDaoSupport就拿不到SqlSessionTemplate了吗?

SqlSession Template硬编码的问题又怎么解决?

但是，我们在实际的Spring 项目里面也没有这么做。我们是直接注入了一个Mapper接口，调用它的方法就OK了。

那这个Mapper接口是怎么拿到SqlSessionTemplate的？当我们调用方法的时候，还会不会通过MapperProxy?

这个Mapper 接口可以@Autowired注入到任何地方的话，它肯定是在容器BeanFactory（比如XmlWebApplicationContext）中注册过了。

问题是：

1、什么时候注册到容器中的?

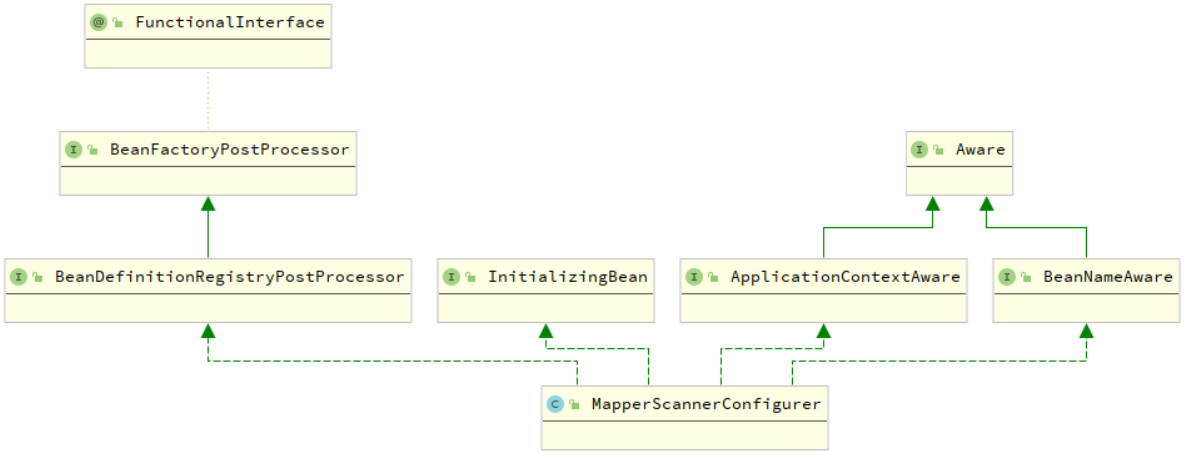
2、注册的时候，注册的是什么对象？是代理对象吗?

2.4. 4、接口的扫描注册

回顾一下，我们在applicationContext.xml里面配置了一个MapperScannerConfigurer，它是用来扫描Mapper接口的。

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。



MapperScannerConfigurer 重写了postProcessBeanDefinitionRegistry（），那它要做什么呢?

在这个方法里面：

创建了一个scanner对象，然后设置属性：

```
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) {
        if (this.processPropertyPlaceHolders) {
            this.processPropertyPlaceHolders();
        }

        ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);
        scanner.setAddToConfig(this.addToConfig);
        scanner.setAnnotationClass(this.annotationClass);
        scanner.setMarkerInterface(this.markerInterface);
        scanner.setSqlSessionFactory(this.sqlSessionFactory);
        scanner.setSqlSessionTemplate(this.sqlSessionTemplate);
        scanner.setSqlSessionFactoryBeanName(this.sqlSessionFactoryBeanName);
        scanner.setSqlSessionTemplateBeanName(this.sqlSessionTemplateBeanName);
        scanner.setResourceLoader(this.applicationContext);
        scanner.setBeanNameGenerator(this.nameGenerator);
        scanner.setMapperFactoryBeanClass(this.mapperFactoryBeanClass);
        if (StringUtils.hasText(this.lazyInitialization)) {

scanner.setLazyInitialization(Boolean.valueOf(this.lazyInitialization));
        }

        scanner.registerFilters();
        scanner.scan(StringUtils.tokenizeToStringArray(this.basePackage, ",;
\t\n"));
    }
```

ClassPathBeanDefinitionScanner的scan（）方法：

```
1.      public int scan(String... basePackages) {
2.          int beanCountAtScanStart = this.registry.getBeanDefinitionCount();
3.
4.          doScan(basePackages);
5.
6.          // Register annotation config processors, if necessary.
7.          if (this.includeAnnotationConfig) {
8.
    AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
9.          }
10.
11.         return (this.registry.getBeanDefinitionCount() - beanCountAtScanStart);
12.     }
13.
14.     /**
15.      * Perform a scan within the specified base packages,
16.      * returning the registered bean definitions.
17.      * <p>This method does <i>not</i> register an annotation config processor
18.      * but rather leaves this up to the caller.
19.      * @param basePackages the packages to check for annotated classes
20.      * @return set of beans registered if any for tooling registration purposes
    (never {@code null})
21.      */
22.     protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
23.         Assert.notEmpty(basePackages, "At least one base package must be
    specified");
24.         Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
25.         for (String basePackage : basePackages) {
26.             Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
27.             for (BeanDefinition candidate : candidates) {
28.                 ScopeMetadata scopeMetadata =
    this.scopeMetadataResolver.resolveScopeMetadata(candidate);
29.                 candidate.setScope(scopeMetadata.getScopeName());
30.                 String beanName =
    this.beanNameGenerator.generateBeanName(candidate, this.registry);
31.                 if (candidate instanceof AbstractBeanDefinition) {
32.                     postProcessBeanDefinition((AbstractBeanDefinition) candidate,
    beanName);
33.                 }
34.                 if (candidate instanceof AnnotatedBeanDefinition) {
35.
    AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition)
    candidate);
36.                 }
37.                 if (checkCandidate(beanName, candidate)) {
38.                     BeanDefinitionHolder definitionHolder = new
    BeanDefinitionHolder(candidate, beanName);
39.                     definitionHolder =
```

```
                AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder,
                this.registry);
                    beanDefinitions.add(definitionHolder);
                    registerBeanDefinition(definitionHolder, this.registry);
                }
            }
        }
        return beanDefinitions;
    }
```

这里会调用它的子类ClassPathMapperScanner的doScan方法：

```
  @Override
  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;
  }
```

1、子类ClassPathMapperScanner又调用了父类ClassPathBeanDefinitionScanner的 doScan()所有的接口，把接口全部添加到beanDefinitions中。

```
    protected Set<BeanDefinitionHolder> doScan(String... basePackages) {
        Assert.notEmpty(basePackages, "At least one base package must be specified");
        Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
        for (String basePackage : basePackages) {
            Set<BeanDefinition> candidates = findCandidateComponents(basePackage);
            for (BeanDefinition candidate : candidates) {
                ScopeMetadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(candidate);
                candidate.setScope(scopeMetadata.getScopeName());
                String beanName = this.beanNameGenerator.generateBeanName(candidate, this.registry);
                if (candidate instanceof AbstractBeanDefinition) {
                    postProcessBeanDefinition((AbstractBeanDefinition) candidate, beanName);
                }
                if (candidate instanceof AnnotatedBeanDefinition) {
                    AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition) candidate);
                }
                if (checkCandidate(beanName, candidate)) {
                    BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(candidate, beanName);
                    definitionHolder =
                    AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, this.registry);
                    beanDefinitions.add(definitionHolder);
                    registerBeanDefinition(definitionHolder, this.registry);
                }
            }
        }
        return beanDefinitions;
    }
```

2、processBeanDefinitions（）方法里面，在注册 beanDefinitions的时候，BeanClass 被改为 MapperFactoryBean

```
  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.mapperFactoryBeanClass);
        ...
  }
```

也就是说，所有的Mapper接口，在容器里面都被注册成一个支持泛型的MapperFactoryBean了。

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？

点开看看。

看一下这个类：

```
public class MapperFactoryBean<T> extends SqlSessionDaoSupport implements
FactoryBean<T> {
```

继承了抽象类SqlSessionDaoSupport，这不就解决了我们的第一个问题了，现在每一个注入Mapper的地方，都可以拿到SqlSessionTemplate。

现在只剩下最后一个问题了，有没有用到MapperProxy？如果注册的是MapperFactoryBean，难道注入使用的也是MapperFactoryBean吗？这个类并不是代理类。

2.5. 5、接口注入使用

所以注入的到底是一个什么对象？注意看MapperFactoryBean也实现了FactoryBean，我们已经见过一次了。它可以在getObject()中修改获取Bean实例的行为。

```
  @Override
  public T getObject() throws Exception {
    return getSqlSession().getMapper(this.mapperInterface);
  }
```

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。

学到了什么?
1、为组件预留扩展接口。
2、利用Spring的扩展机制，把组件集成到MyBatis中。

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

3. 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 插件：Plugin Spring 集成 MyBaits：SqlSessionTemplate 的内部 SqlSessionInterceptor MyBatis 自带连接池：PooledConnection 日志打印：ConnectionLogger、StatementLogger
适配器模式	Log，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、SimpleExecutor、BatchExecutor、ReuseExecutor
装饰器模式	LoggingCache、LruCache 对 PerpetualCache CachingExecutor 对其他 Executor
责任链模式	Interceptor、InterceptorChain