从零开始写个简单的框架

要实现一个简单的类似于Spring的框架，以下是一些需要实现的基本功能：

1. **IoC容器**：实现IoC容器，用于管理应用程序中的对象及其依赖关系，并提供Di依赖注入功能。

   IoC容器是一个重要的功能模块，用于实现控制反转（IoC）和依赖注入（DI）机制，让应用程序能够松散耦合。

   使用classloader。自定义的类加载器识别到要加载的类

   - Bean定义管理：容器需要支持管理应用程序中的各个组件，将它们称为“Bean”，并提供定义Bean的机制—注解。
   - Bean实例化：容器需要实现Bean的实例化和生命周期管理，包括Bean的创建、初始化和销毁等操作。
   - 我们明确地计划不同条件下创建不同实例时，我们可以使用工厂模式。使用Map装实例对象和类名。
   - Bean依赖注入：容器需要实现依赖注入（DI）机制，即将Bean之间的依赖关系由容器自动注入，而不是由Bean自行管理。
   - Bean作用域管理：容器需要支持管理Bean的作用域，如单例、原型等。
   - AOP（面向切面编程）：容器需要支持AOP机制，实现横切关注点的集中处理，如事务管理、日志记录
   - 异常处理：容器需要提供统一的异常处理机制，以便在应用程序中统一处理各种异常情况。
2. **AOP框架**：实现AOP框架，用于实现横切关注点的功能，例如日志、事务等。
3. **MVC框架**：实现MVC框架，用于管理Web应用程序中的模型、视图和控制器。
   - **要遵循Controller–Service接口–ServiceImpt实现类–Mapper接口模式**
4. **ORM框架**
   - 配置文件解析
     1. MyBatis 框架是通过 XML 配置文件来管理 SQL 语句和映射关系的，因此第一步是实现 XML 配置文件的解析。使用 DOM4J 解析器来解析 XML 文件，并将解析结果存储到内存中。
   - 对象实例化
     1. 配置文件中定义了多个 SQL 语句和映射关系，需要在框架启动时将它们实例化成对应的 Java 对象。这个过程可以使用反射机制来完成。
   - SQL 语句执行
   - 映射关系管理
     1. MyBatis 框架需要管理 SQL 语句和 Java 对象之间的映射关系，这个过程通常使用配置文件来完成。需要设计一种映射关系管理方式，能够在运行时根据配置文件来动态地生成映射关系。
   - 事务管理

流程

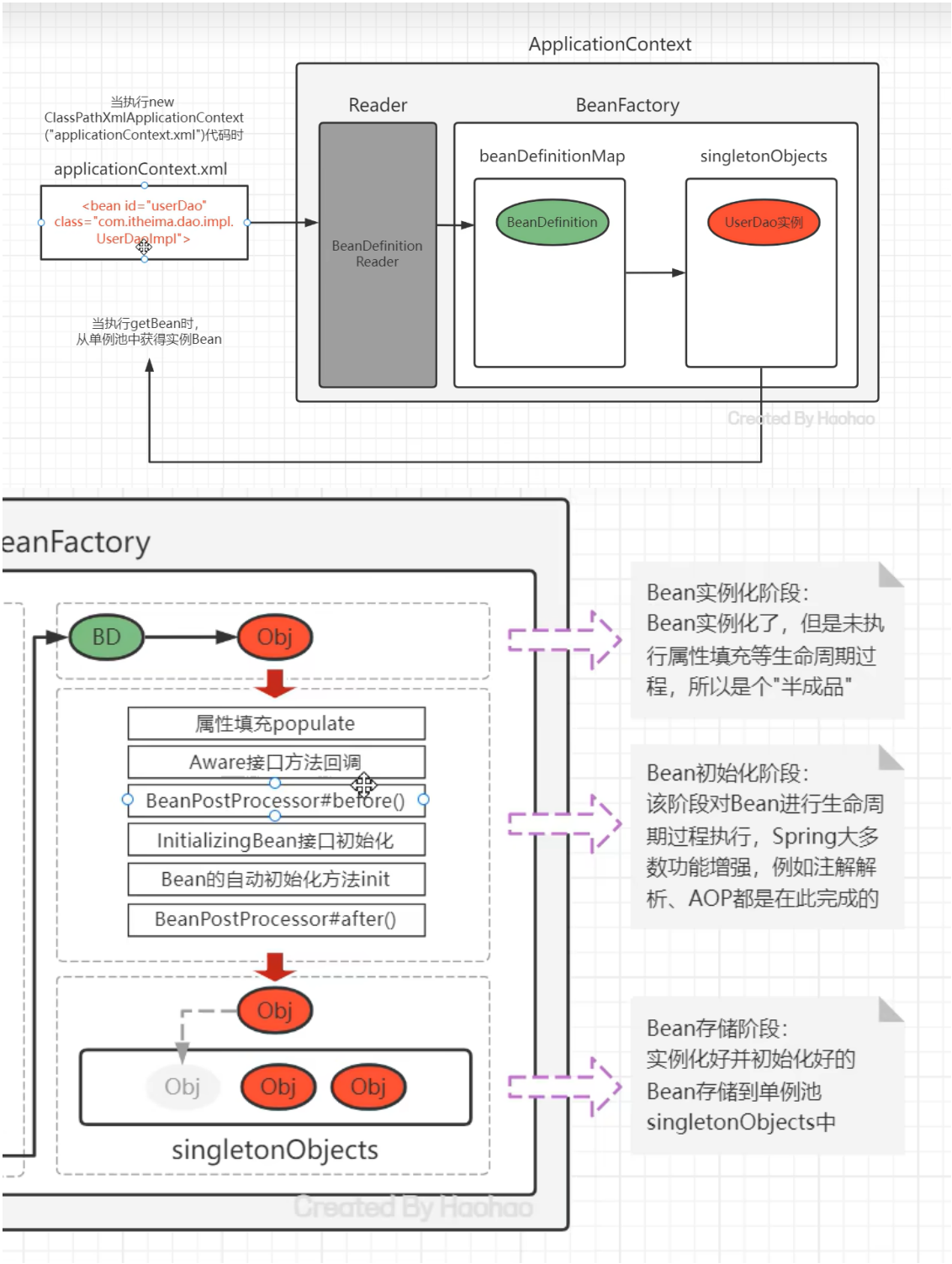
1. 首先，需要创建一个配置文件，用来配置数据库连接、Mapper接口映射、以及一些全局配置。
2. 接下来，需要实现一个SqlSessionFactory类，用来获取SqlSession对象。
3. 实现SqlSession类，用来执行SQL语句，以及对结果进行封装和映射。
4. 实现一个MapperRegistry类，用来管理Mapper接口映射。
5. 实现一个MapperProxy类，用来代理Mapper接口的调用，实现自动映射和转换。

6. 最后，实现Mapper接口，用来定义SQL语句和参数，以及映射结果集。

在实现过程中，需要用到JDBC连接池、反射、注解、XML解析，需要考虑线程安全、异常处理、性能优化等问题。

5. 配置管理：实现配置管理功能，例如读取配置文件、解析配置等

实现IOC

1. 今天先实现IoC容器。
2. 

3. 其中的核心思想是从配置文件中读取Bean的定义信息，将其加载进容器中，并且完成Bean的自动注入。其中 refresh() 方法是容器初始化的入口，先通过 BeanDefinitionReader 读取配置文件并封装成 BeanDefinition ，再通过 doRegisterBeanDefinition() 方法将其注册进 IOC 容器中。doAutowired() 方法则是在 Bean 注册完成后将不是延迟加载的 Bean 提前初始化，并完成 Bean

的自动注入。将实例化的对象封装到BeanWrapper中，并将BeanWrapper保存到一个Map中，同时使用反射实现属性的注入。

1. 学到的新东西：

   loadclass()和class.forname()的区别：

   - loadclass可以加载类且不初始化其中的静态方法/变量。
   - class.forname初始化其中的静态方法，加载到jvm中。这样可以通过反射实例化

   JarURLConnection

   - 是URLConnection的一个实现类。
   - JarURLConnection的实例可以引用一个JAR的压缩包或者这种包里的某个文件。

   Enumeration

   - 枚举接口，有hasMoreElements，nextElement两个方法。
   - 现在很多已被iterator替代，但是还有一些方法的返回值会使用这个接口。

   ClassLoader

   - 作为类的容器，它起到类隔离的作用
   - ClassLoader一共有三个，BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。
   - BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA_HOME/lib/rt.jar 文件中，我们常用内置库 java.xxx.* 都在里面。称之为根加载器。
   - ExtensionClassLoader 负责加载 JVM 扩展类，比如 swing 系列、内置的 js 引擎、xml 解析器 等等，这些库名通常以 javax 开头，它们的 jar 包位于 JAVA_HOME/lib/ext/*.jar 中，有很多 jar 包。
   - AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。我们自己编写的代码以及使用的第三方 jar 包通常都是由它来加载的。

     所以我们的项目加载我们的框架时要使用AppClassLoader。可以通过getContextClassLoader

     1. 如果没有人工去设置，那么所有的线程的 contextClassLoader 都是 AppClassLoader。
     2. 它可以做到跨线程共享类，只要它们共享同一个 contextClassLoader。父子线程之间会自动传递 contextClassLoader，所以共享起来将是自动化的。
     3. 如果我们不去定制 contextClassLoader，那么所有的线程将会默认使用 AppClassLoader，所有的类都将会是共享的。

     那么为什么不用普通的classLoader呢？不是也是获取面向用户的加载器AppClassLoader吗?

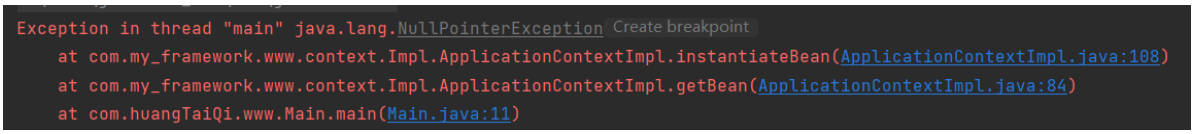
     1. getClassLoader()是当前类加载器,而getContextClassLoader是当前线程的类加载器
     2. getClassLoader是使用双亲委派模型来加载类的,而getContextClassLoader就是为了避开双亲委派模型的加载方式的,也就是说它不是用这种方式来加载类。我的理解就是说在使用getContextClassLoader时AppClassLoader会先自己加载，不行再让他的parent加载。

   getClassLoader().getResources

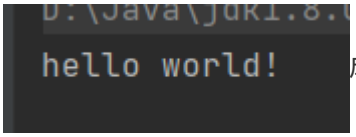
   - 获取url。
   - url.getProtocol()获取协议。网络协议http...文件协议file，jar，一般在路径前面。

     例如 file:/E:/Database/bin/com/test/image.gif

包扫描结果



也是成功地报错了



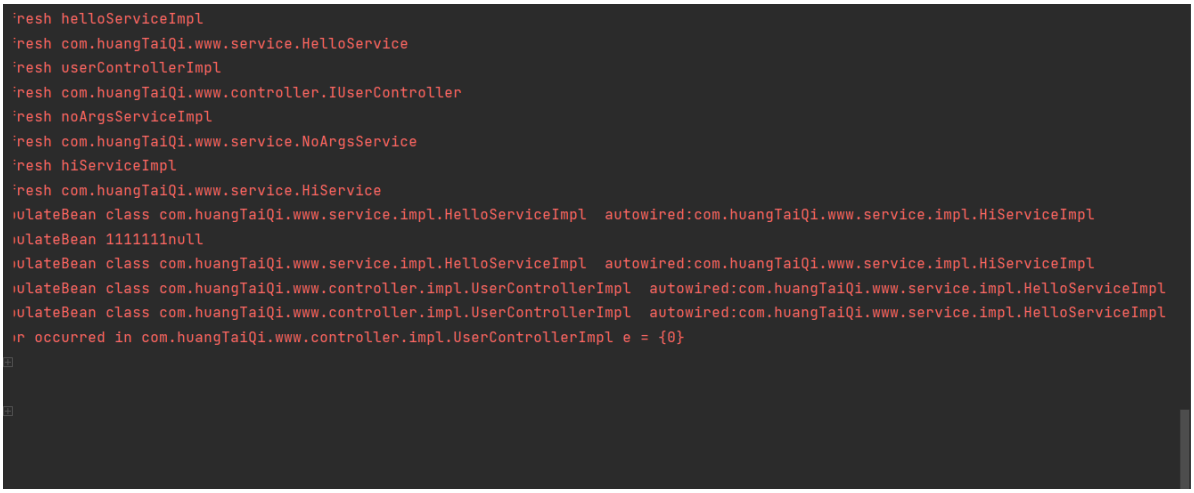
成功运行

注入又有问题

要遵循Controller–Service接口–ServiceImpt实现类–Mapper接口模式

问题：没有处理循环依赖的情况，可能会引发一些问题，需要在开发时小心避免使用循环依赖

运行web项目时，要先包扫描，才可以注入，所以我们要在初始化tomcat的时候加上包扫描



为什么我已经将helloServiceImpl注入进UserController，但是还是有空指针异常？

```
field: "com.huangTaiQi.www.service.impl.HelloServiceImpl com.huangTaiQi.www.controller.impl.UserControllerImpl.helloService"
```

我明白了

```
System.out.println(this);
System.out.println(this.equals(applicationContext.getBean( beanName: "userControllerImpl")));
```

```
com.huangTaiQi.www.controller.impl.UserControllerImpl@4b4cbd35
false
```

不是同一个类。

我试一下能不能用在userController用单例解决

消息 实例化Servlet类[com.huangTaiQi.www.controller.impl.UserControllerImpl]异常

描述 服务器遇到一个意外的情况，阻止它完成请求。

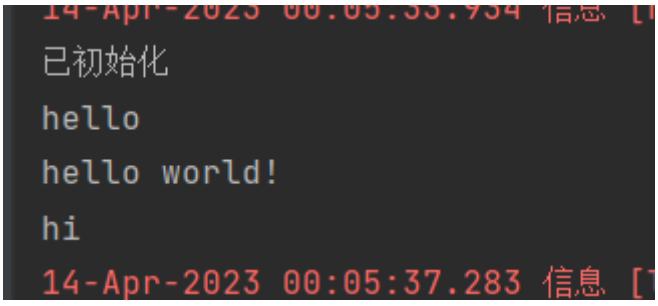
例外情况

```
javax.servlet.ServletException: 实例化Servlet类[com.huangTaiQi.www.controller.impl.UserControllerImpl]异常
        org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
        org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:93)
        org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:682)
        org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:367)
        org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:639)
        org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:63)
        org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:932)
        org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1695)
        org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        org.apache.tomcat.util.threads.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1191)
        org.apache.tomcat.util.threads.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:659)
        org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        java.lang.Thread.run(Thread.java:750)
```

根本原因。

```
java.lang.NoSuchMethodException: com.huangTaiQi.www.controller.impl.UserControllerImpl.<init>()
        java.lang.Class.getConstructor0(Class.java:3082)
        java.lang.Class.getConstructor(Class.java:1825)
        org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:494)
        org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:93)
        org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:682)
        org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:367)
        org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:639)
        org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:63)
        org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:932)
        org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1695)
        org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        org.apache.tomcat.util.threads.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1191)
        org.apache.tomcat.util.threads.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:659)
        org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        java.lang.Thread.run(Thread.java:750)
```

我笑了。我懂了，在转发器改




```
String className = toClassName(this.toString());
Object bean = applicationContext.getBean(className);
try {
    Method method=cls.getMethod(methodName,HttpServletRequest.class,HttpServletResponse.class);
    method.invoke(cls.cast(bean),req,resp);
```

可以的我真的是一个天才，虽然搞了几个小时

又是没有见过的bug

```
Exception in thread "main" java.lang.NoClassDefFoundError Create breakpoint : javax/servlet/http/HttpServlet
    at java.lang.ClassLoader.defineClass1(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:756)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:473)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:74)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:369)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:363) <1 个内部行>
    at java.net.URLClassLoader.findClass(URLClassLoader.java:362)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:418)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:355)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:351)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:348)
    at com.my_framework.www.utils.ClassUtil.loadClass(ClassUtil.java:38)
    at com.my_framework.www.utils.ClassUtil.doAddClass(ClassUtil.java:139)
    at com.my_framework.www.utils.ClassUtil.addClass(ClassUtil.java:115)
    at com.my_framework.www.utils.ClassUtil.addClass(ClassUtil.java:127)
    at com.my_framework.www.utils.ClassUtil.getClassSet(ClassUtil.java:63)
    at com.my_framework.www.beans.BeanDefinitionReader.<init>(BeanDefinitionReader.java:44)
    at com.my_framework.www.context.Impl.ApplicationContextImpl.refresh(ApplicationContextImpl.java:48)
    at com.my_framework.www.context.Impl.ApplicationContextImpl.<init>(ApplicationContextImpl.java:40)
```

排查了一下，发现把

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.1</version>
</dependency>
```

的删了就可以正常运行，但是一直学的就是加上protected啊。

实现AOP

1. 创建通知：在Spring中，支持五种类型的通知，分别是前置通知、后置通知、异常通知、返回通知和环绕通知
2. 创建切点：一个切点是一个连接点的集合，开发者可以在切点中定义一组连接点，从而将通知应用于这组连接点上。
3. 创建切面：切面是通知和切点的集合，根据实际需要来组合通知和切点，从而实现特定的功能。
4. 将切面注入到IOC容器：将创建的切面对象通过Spring的IOC容器进行注入，从而使切面对象成为Spring容器的一个Bean。
5. 创建代理对象：Spring AOP使用JDK动态代理或CGLIB动态代理来实现AOP，我们只实现JDK代理
6. 应用通知：在调用目标方法时，Spring AOP会根据切点信息决定是否需要应用通知。根据不同的通知类型，Spring AOP在特定的方法调用前、后或发生异常时执行相应的通知。

jdk代理是实现了需要代理的对象的接口，然后调用被代理的对象，加上通知就完成了aop。

Aop要在Ioc之前加载，因为首先需要通过Aop获取代理对象，然后才能通过Ioc进行依赖注入。

动态代理和静态代理的区别是代理类不需要程序员来写，程序会自动生成代理对象。这样做的好处是不必为每个类都写一个代理类，一个代理类可以代理多个类。

java动态代理机制中有两个重要的类和接口InvocationHandler（接口）和Proxy（类），这一个类Proxy和接口InvocationHandler是我们实现动态代理的核心；

```
D:\Java\jdk1.8.0_361\bin\java.exe ...
Invoker Before Method!!!
TargetObject:com.huangTaiQi.www.service.impl.HelloServiceImpl@28a418fc
Args:null
hello world!
出现异常
TargetObject:com.huangTaiQi.www.service.impl.HelloServiceImpl@28a418fc
Args:null
Throws:null
Exception in thread "main" java.lang.reflect.UndeclaredThrowableException <1 个内部行>
    at com.huangTaiQi.www.Main.main(Main.java:12)
Caused by: java.lang.reflect.InvocationTargetException <4 个内部行>
    at com.my_framework.www.aop.intercept.MethodInvocation.proceed(MethodInvocation.java:58)
    at com.my_framework.www.aop.aspect.AfterThrowingAdviceInterceptor.invoke(AfterThrowingAdviceInterceptor.java:26)
    at com.my_framework.www.aop.intercept.MethodInvocation.proceed(MethodInvocation.java:67)
    at com.my_framework.www.aop.aspect.AfterReturningAdviceInterceptor.invoke(AfterReturningAdviceInterceptor.java:23)
    at com.my_framework.www.aop.intercept.MethodInvocation.proceed(MethodInvocation.java:67)
    at com.my_framework.www.aop.aspect.MethodBeforeAdviceInterceptor.invoke(MethodBeforeAdviceInterceptor.java:29)
    at com.my_framework.www.aop.intercept.MethodInvocation.proceed(MethodInvocation.java:67)
    at com.my_framework.www.aop.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:47)
    ... 2 more
Caused by: java.lang.NullPointerException Create breakpoint
    at com.huangTaiQi.www.service.impl.HelloServiceImpl.sayHello(HelloServiceImpl.java:13)
    ... 14 more

进程已结束,退出代码1
```

```
@Service
public class HelloServiceImpl implements HelloService {

    @Autowired
    private HiServiceImpl hiService;   hiService: null

    public void sayHello() {
        System.out.println("hello world!");
        System.out.println(hiService [NullPointerException] .hi());   hiService:
    }
}

null

D:\Java\jdk1.8.0_361\bin\java.exe ...
Invoker Before Method!!!
TargetObject:com.huangTaiQi.www.service.impl.HelloServiceImpl@28a4
Args:null
hello world!
Invoker After Method!!!
TargetObject:com.huangTaiQi.www.service.impl.HelloServiceImpl@28a4
Args:null
use time :0
```

发现了一个严重的bug

在创建代理对象前没有注入

，之前写的jdk代理真的坐牢。

jdk代理实现aop先暂停一下。

用cglib实现@Transaction，使被注释的方法进行事务管理。

把数据库连接池和redis连接池都搬到框架模块了

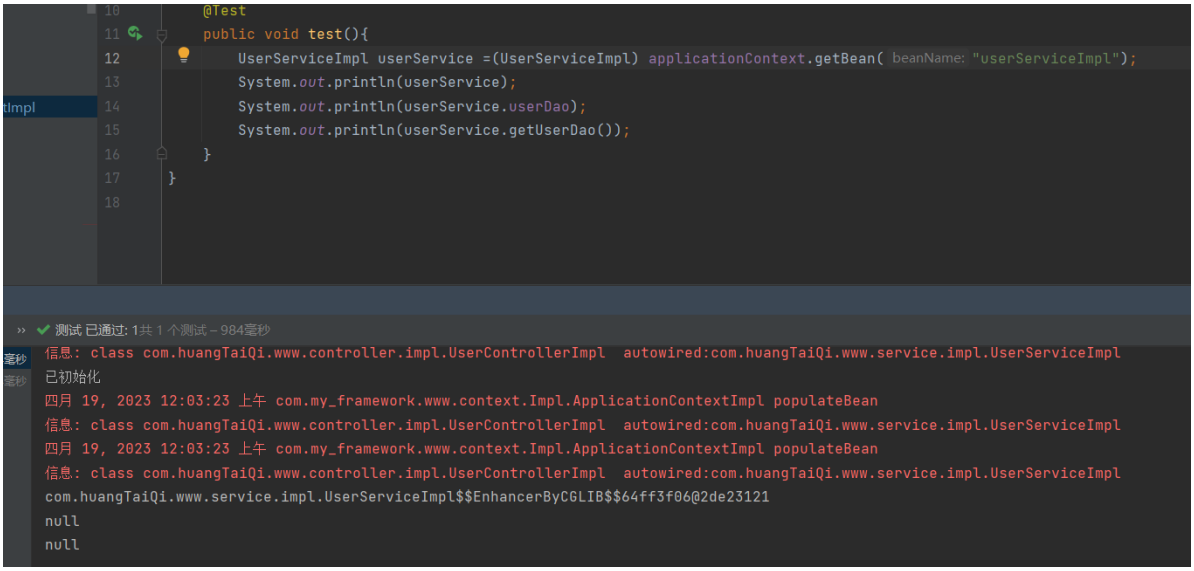
思路：Ioc容器在实例化类时,判断是否是controller。如果是，判断其方法中是否有@Transaction。如果有生成代理对象

使用cglib动态代理实现事务管理后和jdk代理发生了相似的错误。代理对象确实不可以拿到被代理的对象的元素，因为是使用了objenesis，它会绕过构造方法代理，所以成员变量没有初始化。不对不对啊，是因为被注入的成员变量在被代理时没有初始化。

那么如何解决这个问题呢?

```
java.lang.NullPointerException Create breakpoint
    at com.huangTaiQi.www.service.impl.UserServiceImpl.login(UserServiceImpl.java:113)
    at com.huangTaiQi.www.service.impl.UserServiceImpl$$EnhancerByCGLIB$$3221b97b.CGLIB$login$1(<generated>)
    at com.huangTaiQi.www.service.impl.UserServiceImpl$$EnhancerByCGLIB$$3221b97b$$FastClassByCGLIB$$c3b79200.invoke(<generated>)
    at net.sf.cglib.proxy.MethodProxy.invokeSuper(MethodProxy.java:228)
    at com.my_framework.www.Transaction.TransactionHandler.intercept(TransactionHandler.java:22)
    at com.huangTaiQi.www.service.impl.UserServiceImpl$$EnhancerByCGLIB$$3221b97b.login(<generated>)
    at com.huangTaiQi.www.controller.impl.UserControllerImpl.login(UserControllerImpl.java:77) <4 个内部行>
    at com.huangTaiQi.www.controller.BaseController.service(BaseController.java:40)
```

下图是指在*Spring*中*cglib*动态代理不止是用*cglib*，绕开了构造函数，与现在讨论的问题无关，我弄混了



这是代理类的成员方法，我们可以看到并**没有之前我们需要注入的成员变量**，所以我们先生成代理类再扫描成员变量看他有没有需要依赖注入是不可行的

(cglib动态代理实现事务管理，代理类拿不到被代理类被注入的对象，因为被注入的成员变量在被代理时没有初始化，ioc是在获取到实例后再注入的，然后cglib会调用构造方法实现代理，所以cglib动态代理使用被注入的成员变量时报空指针，怎么解决？)这里我之前理解错了

解决了，这个问题起码调试了好多次，想的我头疼。

1. 解决代理类中没有需要注入的成员变量问题：获取父类的成员变量

```
 field[] fields = clazz.getDeclaredFields();
Class<?> superclass = clazz.getSuperclass();
Field[] fields1 = superclass.getFields();
Field[] result = new Field[fields.length + fields1.length];
System.arraycopy(fields, 0, result, 0, fields.length);
System.arraycopy(fields1, 0, result, fields.length, fields1.length);
```

2. 解决注入时为空的问题：如果为空直接实例化

```
                logger.log(Level.INFO,"实例化成员变量");
                Object instance = instantiateBean(beanName, definition);
                //2.把这个对象封装到BeanWrapper中
                BeanWrapper beanWrapper1 = new BeanWrapper(instance);
                //3.把BeanWrapper保存到IOC容器中去
                //注册一个类名（首字母小写，如helloService）
                factoryBeanInstanceCache.put(autowiredBeanName,
beanWrapper1);
```

缺点：被注入的对象不能依赖注入

解决问题：

```
                if(definition!=null){
                    //此成员变量是ioc控制的，但是还未实例化
                    if (factoryBeanInstanceCache.get(autowiredBeanName) ==
null) {
                        //被注入的成员变量还没有实例化
                        logger.log(Level.INFO,"实例化成员变量");
                        getBean(autowiredBeanName1);
                    }
                }else {
                    //此成员变量不是ioc控制的
                    continue;
                }
```

写完之后我豁然开朗，这个逻辑也没有这么难嘛！

实现请求转发

```
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) {
        String uri = req.getRequestURI();
        String methodName = uri.substring(uri.lastIndexOf('/') + 1);
        Class<? extends BaseController> cls = this.getClass();
        String className = toClassName(this.toString());
        Object bean = applicationContext.getBean(className);
        try {
            Method
method=cls.getMethod(methodName,HttpServletRequest.class,HttpServletResponse.cla
ss);
            method.invoke(cls.cast(bean),req,resp);
        } catch (Exception e) {
            try {
                if(e instanceof NoSuchMethodException ||e instanceof
IllegalAccessException){
                    logger.log(Level.SEVERE,"不正常的调用，可能方法名错误或调用私有方
法！ " +
                            "An error occurred in BaseServlet e = {0}", e);
                }
                Throwable cause = e.getCause();
                if (cause instanceof SQLException) {
                    if (e.getMessage().contains("doesn't exist")) {
                        // 判断错误信息是否包含数据表不存在异常信息
                        // 给用户提示操作错误，请联系管理员
                        resp.getWriter().write("操作错误，请联系管理员！");
                        resp.setStatus(500);
                        // 设置状态码为 500
                    } else {
                        logger.log(Level.WARNING,"An error occurred in
"+cls.getName()+" e = {0}", e.getMessage());
                    }
                } else if (cause instanceof IOException) {
                    // 处理 IO 异常
                    resp.getWriter().write("系统繁忙，请稍后再试");
                    logger.log(Level.WARNING,"An error occurred in
"+cls.getName()+" e = {0}", e.getMessage());
```

```
 resp.sendRedirect("http://localhost:8080/project_war/html/error/404.html");
                } else {
                    logger.log(Level.WARNING,"An error occurred in
"+cls.getName()+" e = {0}", e);

 resp.sendRedirect("http://localhost:8080/project_war/html/error/404.html");
                }

            }catch (IOException ex){
                // 处理异常时出现异常
                logger.log(Level.WARNING,"An error occurred in "+cls.getName()+"
e = {0}", ex.getMessage());
            }

        }
    }
```

这是我之前写的请求转发，我暂时觉得有两大痛点

1. url不能进一步细分，如果要细分需要再写一个controller。

   比如我写很多方法获取user的某一个参数我在原来的转发器可以这样写：

   ```
   /user/参数名
   ```

   如果我想在userController写

   ```
   user/info/参数名
   ```

   暂时没想到怎么写。

2. 不能自动填充参数。

   在这个转发器中只能使用request和response两个参数，因为我是通过反射调用的

   ```
   Method
   method=cls.getMethod(methodName,HttpServletRequest.class,HttpServletResponse
   .class);
   ```

   在这里写死了参数列表，所以无法自动填充参数。

   为了解决这两个痛点，我决定写一个新的请求转发器

   思考了一下又引发了几个问题

   1. 参数中还是要有req和resp，（获取session，请求头和转发）

      又是一个大坑啊，填完可能SpringMVC都写得差不多了

      所以还是保留req和resp

   2. post的请求如果不是表单数据不能用getparameter拿。

思路：实例化后将context传给转发器。转发器扫描，如果是controller获取RequestMapping和其方法，

生成url pattern。请求进来看哪个匹配就反射调用哪个方法。

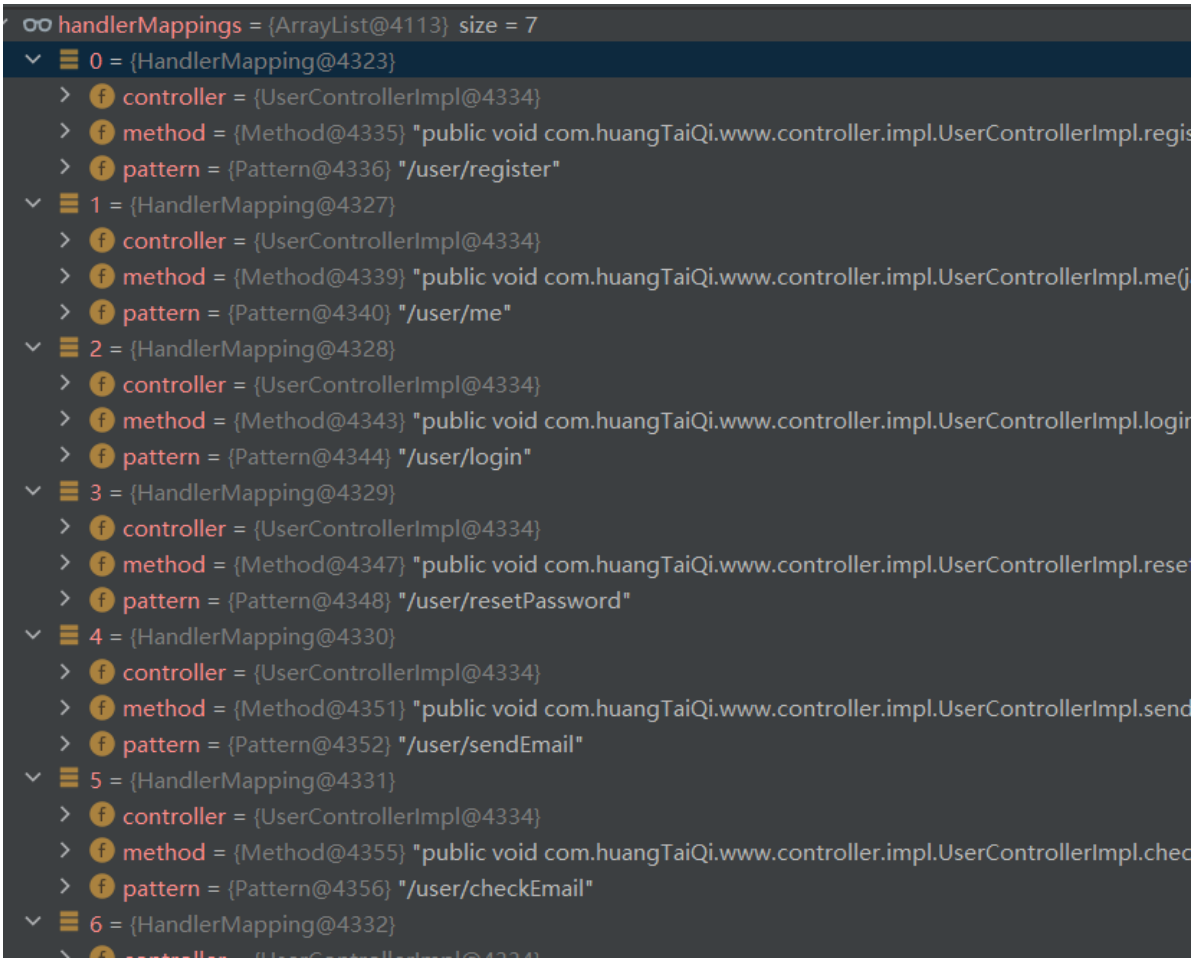
又发现一个很大的问题。

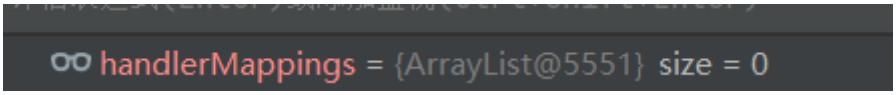
我实现转发器要使用到applicationContext，但是原来我是在我业务的项目中实例化的。

我现在是在请求转发器的init方法实例化applicationContext的，但是尴尬的是我无法包扫描了

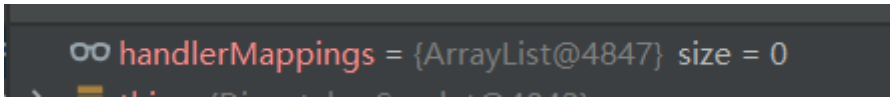
所以我applicationContext在filter的init实例再传

下图的bug是static变量不同步





这个bug明天再看



明明是静态变量，为什么会是不一样的实例呢

（一）静态变量：线程非安全

1、静态变量：使用static关键字定义的变量。static可以修饰变量和方法，也有static静态代码块。被static修饰的成员变量和成员方法独立于该类的任何对象。也就是说，它不依赖类特定的实例，被**类的所有实例共享**。只要这个类被加载，Java虚拟机就能根据类名在运行时数据区的方法区内定找到他们。因此，static对象可以在它的任何对象创建之前访问，无需引用任何对象。

用public修饰的static成员变量和成员方法本质是全局变量和全局方法，当声明它的类的对象时，不生成static变量的副本，而是类的所有实例共享同一个static变量。

2、静态变量使用的场景：

(1)对象间共享值时

(2)方便访问变量时

3、静态方法使用注意事项：

Thread[RMI TCP Connection(5)-127.0.0.1,5,RMI Runtime]

打印了初始化时的线程

这是一个Java RMI (远程方法调用) 线程。

Java RMI 是用于实现远程方法调用的Java API。它使得在Java应用程序之间进行远程方法调用变得简单，可以方便地像调用本地方法一样调用远程方法。

在这个例子中，线程名称为 `RMI TCP Connection(5)-127.0.0.1`，表示这个线程是由RMI网络传输层创建的，而 `127.0.0.1` 表示该线程连接到本地 RMI 服务器。

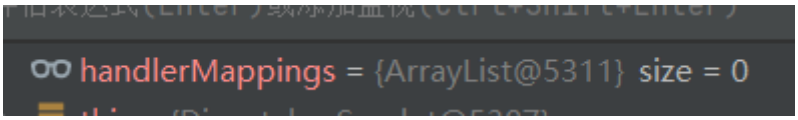
5,RMI Runtime表示这个线程的 ID 是 5，它的线程组是 RMI Runtime。

Thread[http-nio-8080-exec-3,5,main]

执行请求的线程

在Java中，静态变量是在类的初始加载时创建的，属于类级别的变量，拥有全局唯一性的特点。这意味着，无论哪个线程修改了静态变量，它对其他所有线程都是可见的。也就是说，两个线程所访问的静态变量是同一个实例，并且它们之间的修改是共享的。

`Thread[RMI TCP Connection(5)-127.0.0.1,5,RMI Runtime]` 线程修改了静态变量后，另一个线程 `Thread[http-nio-8080-exec-3,5,main]` 无法拿到修改后的值，可能是因为该线程在修改之前已经读取了静态变量的旧值，**容易出现竞态条件的情况。这也是Java多线程编程中需要注意的问题之一。**



怎么加了volatile还是这样

在Java中，对象的地址是唯一的，每个对象都有自己的地址。当多个线程访问同一个静态变量时，它们所拿到的静态变量引用是同一个，但是它们在内存中的存储地址是不同的。每个线程都有自己的运行栈，它们对静态变量的访问都是基于自己线程的运行栈的。

Thread[RMI TCP Connection(5)-127.0.0.1,5,RMI Runtime] 线程和 Thread[http-nio-8080-exec-3,5,main] 线程虽然都访问了同一个静态变量，但它们所拿到的静态变量引用指向的内存地址是不同的，这是因为它们在不同的线程运行栈中。

简单来说，一个静态变量：它的地址是唯一的，引用是唯一的，但是线程栈的内存地址是不同的。

project项目：

```
//1、初始化ApplicationContext，从web.xml中获取参数
ApplicationContextImpl context = new ApplicationContextImpl( configLocation: "application.properties");
//2、初始化Spring MVC
DispatcherServlet.initStrategies(context);
```

framework项目：

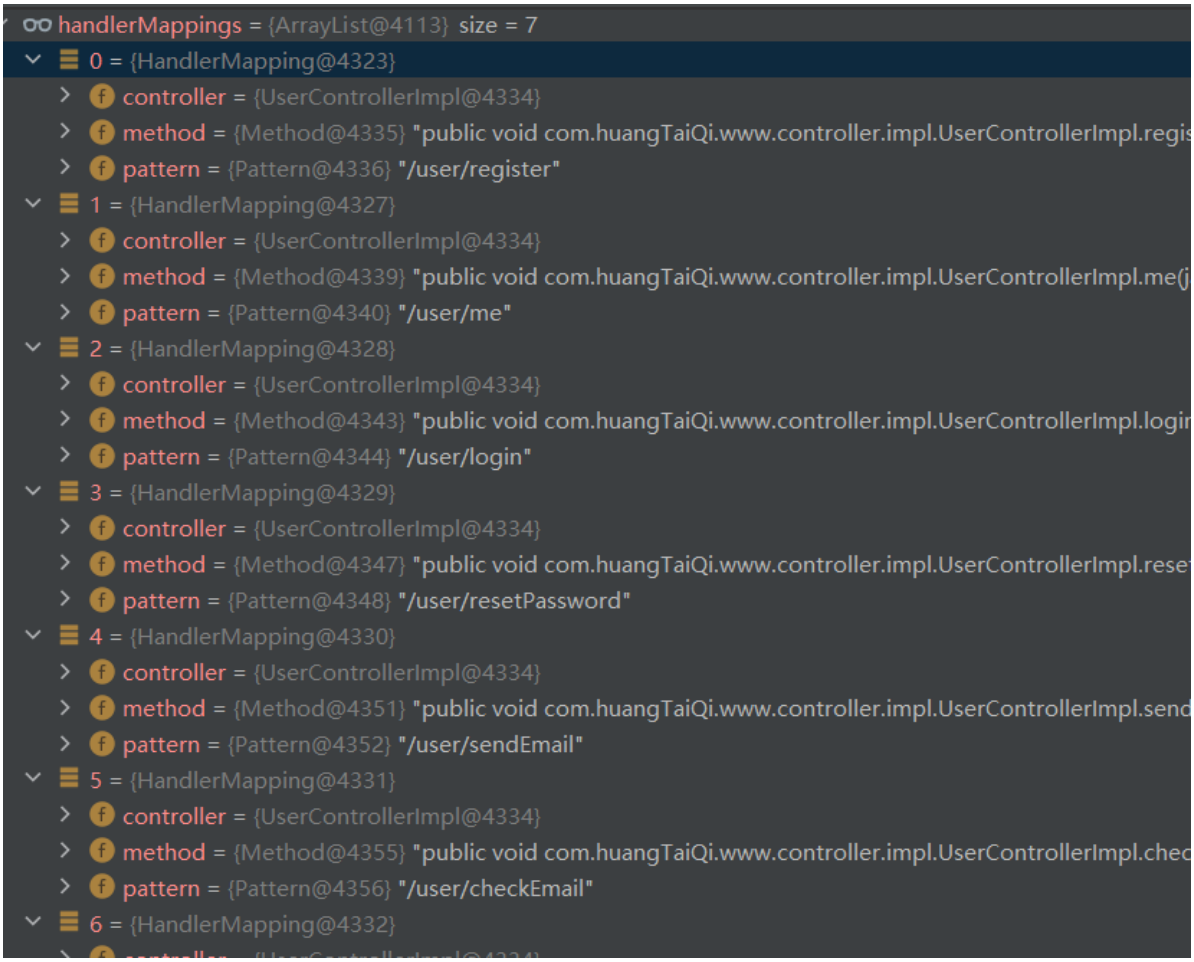
```
private static Logger logger = Logger.getLogger(DispatcherServlet.class.getName());
3 个用法
private static volatile Map<HandlerMapping, HandlerAdapter> handlerAdapters = new HashMap<>();
9 个用法
private static volatile List<HandlerMapping> handlerMappings = new ArrayList<>();
```

```
public static void initStrategies(ApplicationContextImpl context) {
    //handlerMapping，必须实现
    initHandlerMappings(context);
    //初始化参数适配器，必须实现
    initHandlerAdapters();
}
```

下面相当于填充handlerMappings，handlerMappings是需要请求转发的方法

```
private static void initHandlerMappings(ApplicationContextImpl context) {
    //获取ioc容器的类名
    String[] beanNames = context.getBeanDefinitionNames();
    try {
        for (String beanName : beanNames) {
            Object controller = context.getBean(beanName);
            Class<?> clazz = controller.getClass();
            if (!clazz.isAnnotationPresent(Controller.class)) {
                //不是controller跳过
                continue;
            }
            String baseUrl = "";
            //获取Controller的url配置
            if (clazz.isAnnotationPresent(RequestMapping.class)) {
                RequestMapping requestMapping = clazz.getAnnotation(RequestMapping.class);
                baseUrl = requestMapping.value();
            }
            //获取Method的url配置
            Method[] methods = clazz.getMethods();
            for (Method method : methods) {
                //没有加RequestMapping注解的直接忽略
```

填充后：



请求过来执行dodispatch

```
3 个用法  Qiu168
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    doPost(req, resp);
}

2 个用法  Qiu168 *
@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws IOException {
    try {
        logger.log(Level.INFO,Thread.currentThread().toString());
        this.doDispatch(req, resp);
    } catch (Exception e) {
        resp.getWriter().write( s: "500 Exception,Details:\r\n"
                + Arrays.toString(e.getStackTrace()).replaceAll( regex: "\\[|\\]",  replacement: "")
                .replaceAll( regex: ",\\s",  replacement: "\r\n"));
        e.printStackTrace();
    }
}
```

```
1 个用法  Qiu168 *
public void doDispatch(HttpServletRequest req, HttpServletResponse resp) throws Exception {
    //1、通过从request中拿到URL，去匹配一个HandlerMapping
    HandlerMapping handler = getHandler(req);
    if (handler == null) {
        //没有找到handler返回404
        resp.sendRedirect( location: "http://localhost:8080/project_war/html/error/404.html");
        return;
    }
    //2、准备调用前的参数
    HandlerAdapter ha = getHandlerAdapter(handler);
    if(ha==null){
        resp.sendRedirect( location: "http://localhost:8080/project_war/html/error/404.html");
        return;
    }
    //3、真正的调用controller的方法
    ha.handle(req, resp, handler);
}
```

此时handlerMappings为空

```
private HandlerMapping getHandler(HttpServletRequest req) {
    logger.log(Level.INFO,handlerMappings.toString());
    System.out.println(handlerMappings);
    if (handlerMappings.isEmpty()) {
        return null;
    }

    String url = req.getRequestURI();
    String contextPath = req.getContextPath();
    url = url.replace(contextPath,  replacement: "").replaceAll( regex: "/+",  replacement: "/");
    for (HandlerMapping handler : handlerMappings) {
        Matcher matcher = handler.getPattern().matcher(url);
        //如果没有匹配上继续下一个匹配
        if (!matcher.matches()) {
            continue;
        }
        return handler;
    }
    return null;
}
```

```
oo handlerMappings = {ArrayList@5551} size = 0
```

我都不知道我看了多少博客才知道哪里错了

```
<groupId>org.example</groupId>
<artifactId>my-framework</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>war</packaging>
```

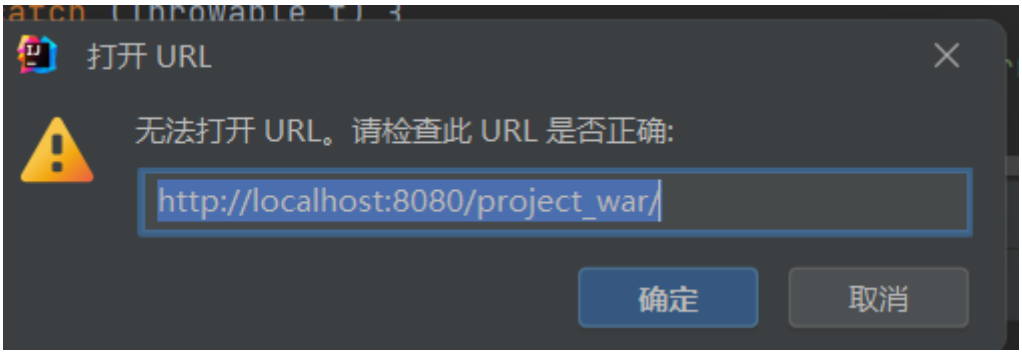
这里不对啊，不应该是war，应该是写的一个jar包给project引用

web.xml不是在framework写是在project写。

web.xml要指定servlet。

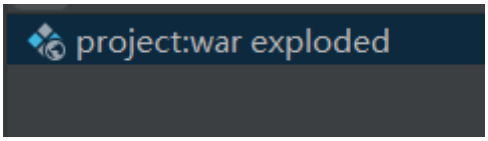
```
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
21-Apr-2023 16:24:30.912 信息 [http-nio-8080-exec-9] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.913 信息 [http-nio-8080-exec-10] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exe
21-Apr-2023 16:24:30.914 信息 [http-nio-8080-exec-1] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.915 信息 [http-nio-8080-exec-3] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.916 信息 [http-nio-8080-exec-2] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.916 信息 [http-nio-8080-exec-5] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.917 信息 [http-nio-8080-exec-4] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.918 信息 [http-nio-8080-exec-6] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.919 信息 [http-nio-8080-exec-7] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.920 信息 [http-nio-8080-exec-8] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.921 信息 [http-nio-8080-exec-9] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:30.922 信息 [http-nio-8080-exec-10] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exe
doFilter
21-Apr-2023 16:24:31.428 信息 [http-nio-8080-exec-2] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.430 信息 [http-nio-8080-exec-5] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.431 信息 [http-nio-8080-exec-4] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.432 信息 [http-nio-8080-exec-6] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.433 信息 [http-nio-8080-exec-7] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.434 信息 [http-nio-8080-exec-8] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.434 信息 [http-nio-8080-exec-9] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.435 信息 [http-nio-8080-exec-10] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exe
21-Apr-2023 16:24:31.436 信息 [http-nio-8080-exec-1] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.437 信息 [http-nio-8080-exec-3] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.438 信息 [http-nio-8080-exec-2] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.439 信息 [http-nio-8080-exec-5] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.440 信息 [http-nio-8080-exec-4] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.440 信息 [http-nio-8080-exec-6] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.441 信息 [http-nio-8080-exec-8] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.442 信息 [http-nio-8080-exec-10] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exe
21-Apr-2023 16:24:31.443 信息 [http-nio-8080-exec-1] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.444 信息 [http-nio-8080-exec-3] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.445 信息 [http-nio-8080-exec-2] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.445 信息 [http-nio-8080-exec-5] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
doFilter
```

```
doFilter
doFilter
doFilter
doFilter
21-Apr-2023 16:24:31.949 信息 [http-nio-8080-exec-7] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.951 信息 [http-nio-8080-exec-8] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.952 信息 [http-nio-8080-exec-9] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.953 信息 [http-nio-8080-exec-10] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exe
21-Apr-2023 16:24:31.954 信息 [http-nio-8080-exec-1] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.955 信息 [http-nio-8080-exec-3] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.956 信息 [http-nio-8080-exec-2] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.957 信息 [http-nio-8080-exec-6] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.957 信息 [http-nio-8080-exec-4] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.958 信息 [http-nio-8080-exec-7] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.959 信息 [http-nio-8080-exec-8] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.960 信息 [http-nio-8080-exec-9] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.961 信息 [http-nio-8080-exec-10] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exe
21-Apr-2023 16:24:31.962 信息 [http-nio-8080-exec-3] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.963 信息 [http-nio-8080-exec-2] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.963 信息 [http-nio-8080-exec-5] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.964 信息 [http-nio-8080-exec-6] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.965 信息 [http-nio-8080-exec-7] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.966 信息 [http-nio-8080-exec-9] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exec
21-Apr-2023 16:24:31.967 信息 [http-nio-8080-exec-10] com.my_framework.www.webmvc.DispatcherServlet.doPost Thread[http-nio-8080-exe
doFilter
doFilter
```



又是奇奇怪怪的错误

要取这个url，然后filter也要过滤？？然后打不开这个url就一直请求，然后redis池撑不住了。



改成exploded就好了。为什么呢，就因为热部署？还是因为url改变了？

完善：前面因为如果application.properties在framework中会找不到包扫描的包，而且不方便修改。然后如果application.properties在project项目framework无法读取到。在filter的init实例又感觉很奇怪。这一事情在web.xml的到来迎来了转机。servlet部署的时候会读取web.xml。web.xml里有application.properties的地址就解决了。