链滴

Latke 源码解析（二）IOC 部分

作者：wthfeng

原文链接：https://ld246.com/article/1493620909167

来源网站：链滴

上篇 Latke源码解析（一）Servlet部分讲解了latke有关web 请求的servlet部分，这次深入了解一下的Ioc部分内容。

前言

本文分析基于latke2.3.6版本，这部分内容主要涉及到了以下2个类库的使用，请留意。

1. javax.enterprise 依赖注入相关(Contexts and Dependency Injection for Java)
2. javassist，操作java字节码的类库

备注： **本人水平有限，若发现文章有误，请积极留言。**

一、从监听器开始

同Spring一样，latke通过配置监听器初始化bean。在latke-demo的web.xml监听器部分如下

```
<listener>
        <listener-class>org.b3log.latke.demo.hello.HelloServletListener</listener-class>
</listener>
```

从这里可以看出，该demo的监听器为HelloServletListener。下面看看HelloServletListener

HelloServletListener.java

```
public class HelloServletListener extends AbstractServletListener {
    @Override
    public void contextInitialized(final ServletContextEvent servletContextEvent) {
        Latkes.setScanPath("org.b3log.latke.demo.hello");  //设置项目扫描的包
        super.contextInitialized(servletContextEvent);  //调用父类的contextInitialized()方法

        //省略其他操作
    }
    //省略其他方法
}
```

HelloServletListener 继承了AbstractServletListener 。而AbstractServletListener是latke提供的默监听器抽象类，在此类中已实现bean的查找注册等许多功能。此类也是我们稍后分析的重点，让我先看一些它的定义。（为简洁，已去掉与ioc无关内容）

AbstractServletListener.java

```
public abstract class AbstractServletListener implements ServletContextListener, ServletReque
tListener, HttpSessionListener {
    /**
     * Servlet context.
     */
    private static ServletContext servletContext;

    @Override
    public void contextInitialized(final ServletContextEvent servletContextEvent) {
      //省略其他配置项
        try {
```

```
            final Collection<Class<?>> beanClasses = Discoverer.discover(Latkes.getScanPath()); //
按需要加载项目中的类文件

            Lifecycle.startApplication(beanClasses); // 创建注册bean
        } catch (final Exception e) {
            //异常处理
        }
        CronService.start();  //定时任务
    }

//其他方法
```

可以知道，HelloServletListener 通过继承AbstractServletListener 实现了ServletContextListener个监听器，可以监听servlet容器创建销毁事件。当项目启动时，contextInitialized()方法执行，bean可以被加载及初始化。从而为servlet部分的请求服务。

须知：HelloServletListener还实现了ServletRequestListener,ServletRequestListener两个监听器用于监听request、session变化，不过这里不是重点。

二、AbstractServletListener

我们已经知道，此类的contextInitialized 方法在项目启动时执行，完成bean的创建工作。

具体的执行逻辑分为两步，对应方法内的两行代码

1. Discoverer.discover(Latkes.getScanPath())

扫描这些指定的类，选出其中包含特定注解的类（@RequestProcessor、@Service等），并加载这特定类。

2. Lifecycle.startApplication(beanClasses);

beanClasses为上步得到的类，解析这些类依赖并创建为bean。

1.discover方法

discover接受字符串形式的类路径，一般是项目的根目录包，也好理解，扫描就要扫描整个项目，把注册的bean的类先找到再说。如demo项目中，HelloServletListener传的就是org.b3log.latke.demo hello。

在discover方法中，要知道：

1. discover的主要任务是找到特定类并加载它们。
2. 除传入的类路径外，discover还要扫描并处理一些内置的包，如 org.b3log.latke.remote 这个。
3. discover使用了javassist类库中的ClassFile，javassist是一个用来处理 Java 字节码的类库。详细参考javassist使用指南

具体的解析过程较为复杂繁琐。这里只看看大致流程的代码

```
    public static Collection<Class<?>> discover(final String scanPath) throws Exception {
        //将传入类包与内置包组合
        final String[] paths = ArrayUtils.concatenate(splitPaths, BUILT_IN_COMPONENT_PKGS);
```

```
        //遍历路径，将其转为磁盘绝对路径，便于javassist解析
        for (String path : paths) {
            if (!AntPathMatcher.isPattern(path)) {
                path = path.replaceAll("\\.", "/") + "/**/*.class";
            }
            urls.addAll(ClassPathResolver.getResources(path));
        }
        for (URL url : urls) {
            //javassist读取class文件，并将文件中类注解信息解析出来
            //遍历一个类下所有注解
            for (final Annotation annotation : annotations){
                //包含`@RequestProcessor，Service，Repository，Named`注解将maybeBeanClass
置true
            }
            if (maybeBeanClass) {
                clz = Thread.currentThread().getContextClassLoader().loadClass(className);  //符合
件，加载此类
                ret.add(clz);
            }
        }
        return ret; //返回Class<?>类型集合信息
    }
```

2. startApplication方法

上步中，已经加载了这些bean类，下面就该解析这些bean的依赖关系并放在所谓的bean容器中以便管理了。

startApplication的代码不多，我们来看看

```
 public static void startApplication(final Collection<Class<?>> classes, final BeanModule... be
nModule) {

        beanManager = LatkeBeanManagerImpl.getInstance();  //获得beanManager实例

        applicationContext.setActive(true);

        beanManager.addContext(applicationContext);          //添加上下文
        final Configurator configurator = beanManager.getConfigurator();

        if (null != classes && !classes.isEmpty()) {
            configurator.createBeans(classes);       //classes为discover()的解析值，据此创建bean
        }
        //设置beanModule
    }
```

其中最为关键的是configurator.createBeans(classes);这行代码，查看ConfiguratorImpl的 createBeans()方法，为遍历调用createBean()方法创建。

```
 public <T> LatkeBean<T> createBean(final Class<T> beanClass) {
        try {
            return (LatkeBean<T>) beanManager.getBean(beanClass);  //若存在直接返回bean
        } catch (final Exception e) {
```

```
            LOGGER.log(Level.TRACE, "Not found bean [beanClass={0}], so to create it", beanClass)

        }
        if (!Beans.checkClass(beanClass)) {  //检查是否是符合条件的bean,不能是接口或抽象类
            //抛出异常
        }
        final String name = Beans.getBeanName(beanClass);

        if (null == name) {
            //打印日志，返回
        }
        //获取此bean@Qualifier注解信息
        final Set<Annotation> qualifiers = Beans.getQualifiers(beanClass, name);
        //获取此bean@Scope注解信息
        final Class<? extends Annotation> scope = Beans.getScope(beanClass);
        //获取此bean父类及实现的接口信息
        final Set<Type> beanTypes = Beans.getBeanTypes(beanClass);

        final Set<Class<? extends Annotation>> stereotypes = Beans.getStereotypes(beanClass);
         //创建bean实例，此为真正创建的方法
        final LatkeBean<T> ret = new BeanImpl<T>(beanManager, name, scope, qualifiers, bea
Class, beanTypes, stereotypes);

        beanManager.addBean(ret);  //将bean添加到容器中

        for (final Type beanType : beanTypes) {
            addTypeClassBinding(beanType, beanClass);
        }

        for (final Annotation qualifier : qualifiers) {
            addClassQualifierBinding(beanClass, qualifier);
            addQualifierClassBinding(qualifier, beanClass);
        }

        return ret;
    }
```

三、深入bean创建过程

BeanImpl的构造方法是创建bean的关键，来看看

```
 public BeanImpl(final LatkeBeanManager beanManager, final String name, final Class<? exte
ds Annotation> scope,
        final Set<Annotation> qualifiers, final Class<T> beanClass, final Set<Type> types,
        final Set<Class<? extends Annotation>> stereotypes) {
        this.beanManager = beanManager;
        this.name = name;
        this.scope = scope;
        this.qualifiers = qualifiers;
        this.beanClass = beanClass;
        this.types = types;
        this.stereotypes = stereotypes;
```

```
        this.configurator = beanManager.getConfigurator();

        javassistMethodHandler = new JavassistMethodHandler(beanManager);
        final ProxyFactory proxyFactory = new ProxyFactory();

        proxyFactory.setSuperclass(beanClass);
        proxyFactory.setFilter(javassistMethodHandler.getMethodFilter());
        proxyClass = proxyFactory.createClass();  //得到该bean的Class对象

        // ① 查看这个bean在构造器、方法、字段上的有无@Inject注解，有则存起来
        annotatedType = new AnnotatedTypeImpl<T>(beanClass);

        constructorParameterInjectionPoints = new HashMap<AnnotatedConstructor<T>, List<
arameterInjectionPoint>>();
        constructorParameterProviders = new ArrayList<ParameterProvider<?>>();
        methodParameterInjectionPoints = new HashMap<AnnotatedMethod<?>, List<Paramet
rInjectionPoint>>();
        methodParameterProviders = new HashMap<AnnotatedMethod<?>, List<ParameterPro
ider<?>>>();
        fieldInjectionPoints = new HashSet<FieldInjectionPoint>();
        fieldProviders = new HashSet<FieldProvider<?>>();

        //根据①中结果，处理依赖
        initFieldInjectionPoints();
        initConstructorInjectionPoints();
        initMethodInjectionPoints();
    }
```

主要有两个过程，一是通过javassist得到这个bean的Class对象，以便后续操作。再者初始化该类依。上面注释了流程。我们来看看怎样初始化字段上的@Inject注解

BeanImpl.java

```
  private void initFieldInjectionPoints() {
      //此bean中含有@Inject注解的set集合
        final Set<AnnotatedField<? super T>> annotatedFields = annotatedType.getFields();

        for (final AnnotatedField<? super T> annotatedField : annotatedFields) {
            final Field field = annotatedField.getJavaMember();

            if (field.getType().equals(Provider.class)) { // by provider
                final FieldProvider<T> provider = new FieldProvider<T>(beanManager, annotatedF
eld);

                fieldProviders.add(provider);

                final FieldInjectionPoint fieldInjectionPoint = new FieldInjectionPoint(this, annotate
Field);

                fieldInjectionPoints.add(fieldInjectionPoint);
            } else { // 字段类型不是Provider走这个流程
                //构造一个FieldInjectionPoint对象，加入此bean的fieldInjectionPoints中
                final FieldInjectionPoint fieldInjectionPoint = new FieldInjectionPoint(this, annotate
Field);
```

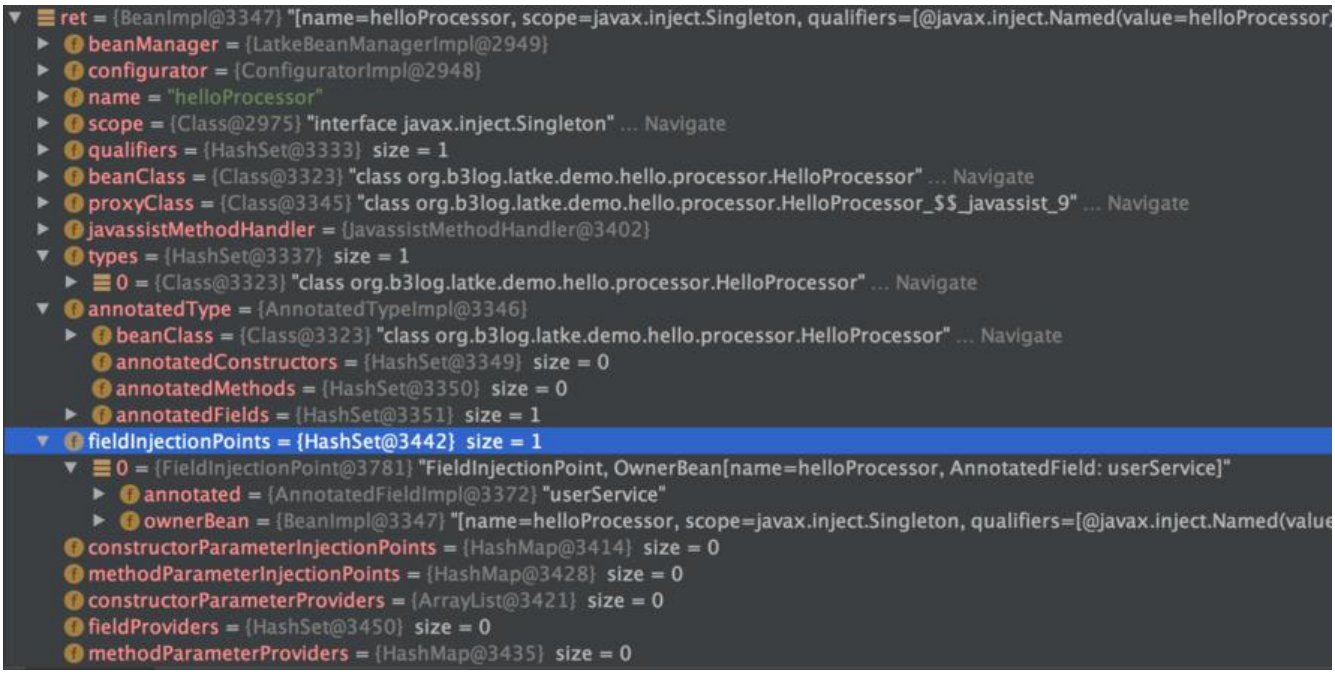
```
                fieldInjectionPoints.add(fieldInjectionPoint);
            }
        }
    }
```

至此，bean的属性基本已构造完毕，以HelloProcessor 这个类对应的bean，我添加了一个Service赖，如下

```
@RequestProcessor
public class HelloProcessor {
    @Inject
    private UserService userService;

    @Before(adviceClass = HelloAdvice.class)
    @RequestProcessing(value = {"/", "/index", "/index.*", "/**/ant/*/path"}, method = HTTPRe
uestMethod.GET)
    public void index(final HTTPRequestContext context)     {
      //省略
    }
}
```

则此bean到目前分析这，其属性构造如下图



亮的部分fieldInjectionPoints 属性表示该bean字段上的依赖集合，可以看到userService这个依赖。此bean就算基本创建完毕了。其他部分诸如分析依赖的工作，等下次有时间再分析吧。

参考文章：

1. Javassist 使用指南（一）
2. Javassist 使用指南（三）
3. 3.11.3 JSR-330标准注解的限制
4. Java 依赖注入标准（JSR-330）简介

5. CDI是什么?