有道云链接：http://note.youdao.com/noteshare?id=663444a188ed9057dffd07cfbeed43cc&sub=D6060DA6961E42D699A3B7DDB2118F93（复制链接到浏览器的时候注意转行的空格哦）

作者：周瑜

前面两节课，我们大概了解了Spring中的一些概念和底层工作流程，本节课开始将真正讲一些Spring中的概念和工作流程。

本节课的内容，是后续看Spring源码所必备的，防止后续看源码的过程中，遇到不会的概念得单独跳出来学习。

BeanDefinition

BeanDefinition表示Bean定义，BeanDefinition中存在很多属性用来描述一个Bean的特点。比如：

- class，表示Bean类型
- scope，表示Bean作用域，单例或原型等
- lazyInit：表示Bean是否是懒加载
- initMethodName：表示Bean初始化时要执行的方法
- destroyMethodName：表示Bean销毁时要执行的方法
- 还有很多...

在Spring中，我们经常会通过以下几种方式来定义Bean：

1. <bean/>
2. @Bean
3. @Component(@Service,@Controller)

这些，我们可以称之**申明式定义Bean**。

我们还可以**编程式定义Bean**，那就是直接通过BeanDefinition，比如：

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

// 生成一个BeanDefinition对象，并设置beanClass为User.class，并注册到ApplicationContext中
AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.genericBeanDefinition().getBeanDefinition();
beanDefinition.setBeanClass(User.class);
context.registerBeanDefinition("user", beanDefinition);

System.out.println(context.getBean("user"));
```

我们还可以通过BeanDefinition设置一个Bean的其他属性

```
beanDefinition.setScope("prototype"); // 设置作用域
beanDefinition.setInitMethodName("init"); // 设置初始化方法
beanDefinition.setLazyInit(true); // 设置懒加载
```

和申明式事务、编程式事务类似，通过<bean/>，@Bean，@Component等申明式方式所定义的Bean，最终都会被Spring解析为对应的BeanDefinition对象，并放入Spring容器中。

BeanDefinitionReader

接下来，我们来介绍几种在Spring源码中所提供的BeanDefinition读取器（BeanDefinitionReader），这些BeanDefinitionReader在我们使用Spring时用得少，但在Spring源码中用得多，相当于Spring源码的基础设施。

AnnotatedBeanDefinitionReader

可以直接把某个类转换为BeanDefinition，并且会解析该类上的注解，比如

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

AnnotatedBeanDefinitionReader annotatedBeanDefinitionReader = new
AnnotatedBeanDefinitionReader(context);

// 将User.class解析为BeanDefinition
annotatedBeanDefinitionReader.register(User.class);

System.out.println(context.getBean("user"));
```

注意：它能解析的注解是：@Conditional，**@Scope**、@Lazy、@Primary、@DependsOn、@Role、@Description

XmlBeanDefinitionReader

可以解析<bean/>标签

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

XmlBeanDefinitionReader xmlBeanDefinitionReader = new XmlBeanDefinitionReader(context);
int i = xmlBeanDefinitionReader.loadBeanDefinitions("spring.xml");

System.out.println(context.getBean("user"));
```

ClassPathBeanDefinitionScanner

ClassPathBeanDefinitionScanner是扫描器，但是它的作用和BeanDefinitionReader类似，它可以进行扫描，扫描某个包路径，对扫描到的类进行解析，比如，扫描到的类上如果存在@Component注解，那么就会把这个类解析为一个BeanDefinition，比如：

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.refresh();

ClassPathBeanDefinitionScanner scanner = new ClassPathBeanDefinitionScanner(context);
scanner.scan("com.zhouyu");

System.out.println(context.getBean("userService"));
```

BeanFactory

BeanFactory表示Bean**工厂**，所以很明显，BeanFactory会负责创建Bean，并且提供获取Bean的API。

而ApplicationContext是BeanFactory的一种，在Spring源码中，是这么定义的：

```
public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory,
HierarchicalBeanFactory,
        MessageSource, ApplicationEventPublisher, ResourcePatternResolver {

            ...
}
```

首先，在Java中，接口是可以**多继承**的，我们发现ApplicationContext继承了ListableBeanFactory和HierarchicalBeanFactory，而ListableBeanFactory和HierarchicalBeanFactory都继承至BeanFactory，所以我们可以认为ApplicationContext继承了BeanFactory，相当于苹果继承水果，宝马继承汽车一样，ApplicationContext也是BeanFactory的一种，拥有BeanFactory支持的所有功能，不过ApplicationContext比BeanFactory更加强大，ApplicationContext还基础了其他接口，也就表示ApplicationContext还拥有其他功能，比如MessageSource表示国际化，ApplicationEventPublisher表示事件发布，EnvironmentCapable表示获取环境变量，等等，关于ApplicationContext后面再详细讨论。

在Spring的源码实现中，当我们new一个ApplicationContext时，其底层会new一个BeanFactory出来，当使用ApplicationContext的某些方法时，比如getBean()，底层调用的是BeanFactory的getBean()方法。

在Spring源码中，BeanFactory接口存在一个非常重要的实现类是：
DefaultListableBeanFactory，也是非常核心的。具体重要性，随着后续课程会感受更深。

所以，我们可以直接来使用**DefaultListableBeanFactory**，而不用使用ApplicationContext的某个实现类，比如：

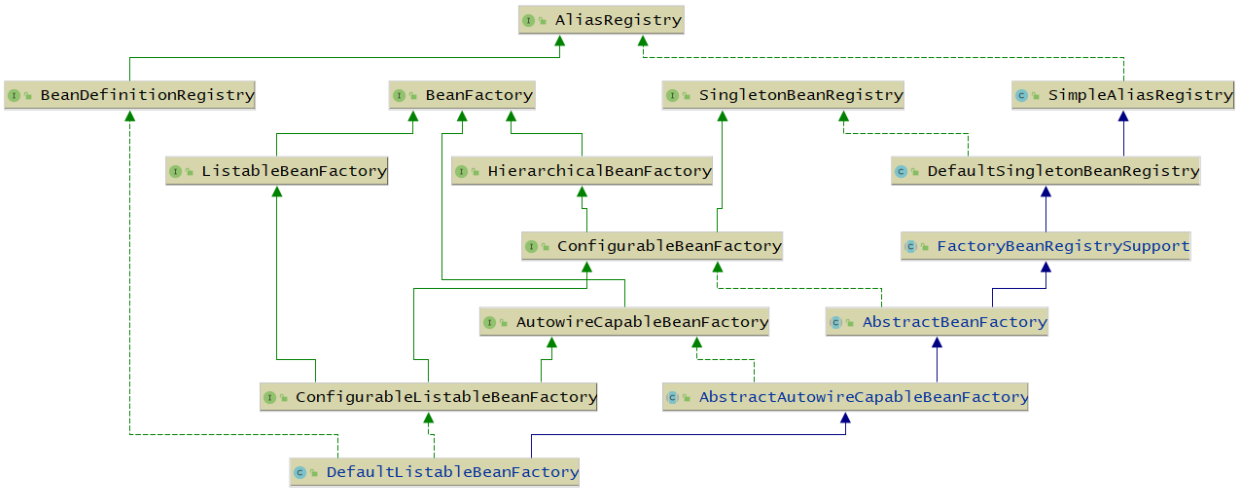
```
DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();

AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.genericBeanDefinition().getBeanDefinition();
beanDefinition.setBeanClass(User.class);

beanFactory.registerBeanDefinition("user", beanDefinition);

System.out.println(beanFactory.getBean("user"));
```

DefaultListableBeanFactory是非常强大的，支持很多功能，可以通过查看DefaultListableBeanFactory的类继承实现结构来看



这部分现在看不懂没关系，源码熟悉一点后回来再来看都可以。

它实现了很多接口，表示，它拥有很多功能：

1. AliasRegistry：支持别名功能，一个名字可以对应多个别名
2. BeanDefinitionRegistry：可以注册、保存、移除、获取某个BeanDefinition
3. BeanFactory：Bean工厂，可以根据某个bean的名字、或类型、或别名获取某个Bean对象
4. SingletonBeanRegistry：可以直接注册、获取某个**单例**Bean
5. SimpleAliasRegistry：它是一个类，实现了AliasRegistry接口中所定义的功能，支持别名功能
6. ListableBeanFactory：在BeanFactory的基础上，增加了其他功能，可以获取所有BeanDefinition的beanNames，可以根据某个类型获取对应的beanNames，可以根据某个类型获取{类型：对应的Bean}的映射关系
7. HierarchicalBeanFactory：在BeanFactory的基础上，添加了获取父BeanFactory的功能
8. DefaultSingletonBeanRegistry：它是一个类，实现了SingletonBeanRegistry接口，拥有了直接注册、获取某个**单例**Bean的功能

9. ConfigurableBeanFactory：在HierarchicalBeanFactory和SingletonBeanRegistry的基础上，添加了设置父BeanFactory、类加载器（表示可以指定某个类加载器进行类的加载）、设置Spring EL表达式解析器（表示该BeanFactory可以解析EL表达式）、设置类型转化服务（表示该BeanFactory可以进行类型转化）、可以添加BeanPostProcessor（表示该BeanFactory支持Bean的后置处理器），可以合并BeanDefinition，可以销毁某个Bean等等功能
10. FactoryBeanRegistrySupport：支持了FactoryBean的功能
11. AutowireCapableBeanFactory：是直接继承了BeanFactory，在BeanFactory的基础上，支持在创建Bean的过程中能对Bean进行自动装配
12. AbstractBeanFactory：实现了ConfigurableBeanFactory接口，继承了FactoryBeanRegistrySupport，这个BeanFactory的功能已经很全面了，但是不能自动装配和获取beanNames
13. ConfigurableListableBeanFactory：继承了ListableBeanFactory、AutowireCapableBeanFactory、ConfigurableBeanFactory
14. AbstractAutowireCapableBeanFactory：继承了AbstractBeanFactory，实现了AutowireCapableBeanFactory，拥有了自动装配的功能
15. DefaultListableBeanFactory：继承了AbstractAutowireCapableBeanFactory，实现了ConfigurableListableBeanFactory接口和BeanDefinitionRegistry接口，所以DefaultListableBeanFactory的功能很强大

ApplicationContext

上面有分析到，ApplicationContext是个接口，实际上也是一个BeanFactory，不过比BeanFactory更加强大，比如：

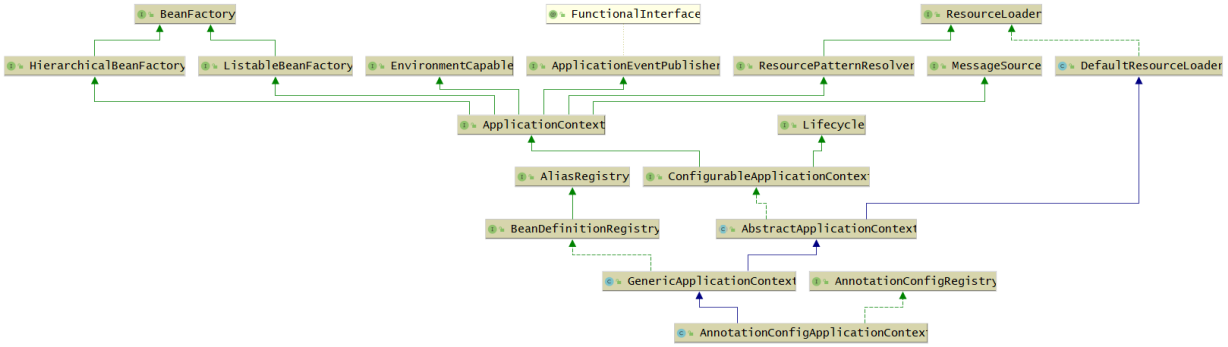
1. HierarchicalBeanFactory：拥有获取父BeanFactory的功能
2. ListableBeanFactory：拥有获取beanNames的功能
3. ResourcePatternResolver：资源加载器，可以一次性获取多个资源（文件资源等等）
4. EnvironmentCapable：可以获取运行时环境（没有设置运行时环境功能）
5. ApplicationEventPublisher：拥有广播事件的功能（没有添加事件监听器的功能）
6. MessageSource：拥有国际化功能

具体的功能演示，后面会有。

我们先来看ApplicationContext两个比较重要的实现类：

1. AnnotationConfigApplicationContext
2. ClassPathXmlApplicationContext

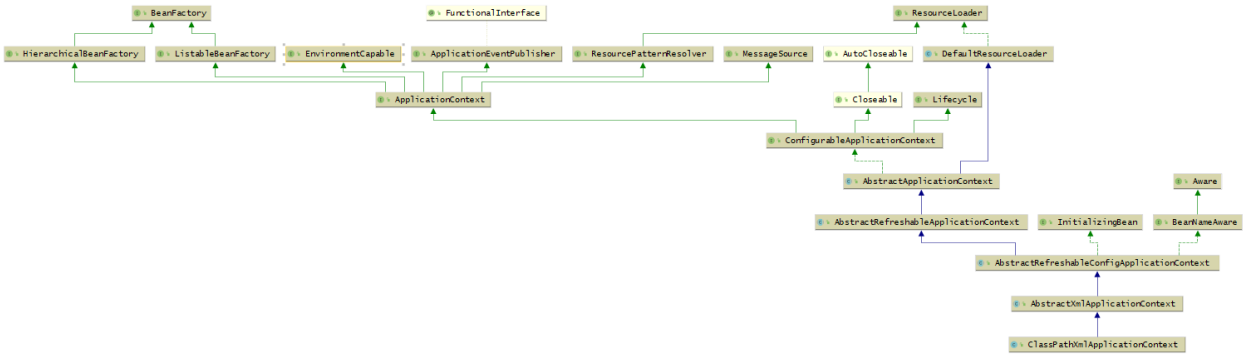
AnnotationConfigApplicationContext



这部分现在看不懂没关系，源码熟悉一点后回来再来看都可以。

1. ConfigurableApplicationContext：继承了ApplicationContext接口，增加了，添加事件监听器、添加BeanFactoryPostProcessor、设置Environment，获取ConfigurableListableBeanFactory等功能
2. AbstractApplicationContext：实现了ConfigurableApplicationContext接口
3. GenericApplicationContext：继承了AbstractApplicationContext，实现了BeanDefinitionRegistry接口，拥有了所有ApplicationContext的功能，并且可以注册BeanDefinition，注意这个类中有一个属性(DefaultListableBeanFactory **beanFactory**)
4. AnnotationConfigRegistry：可以单独注册某个为类为BeanDefinition（可以处理该类上的**@Configuration注解**，已经可以处理**@Bean注解**），同时可以扫描
5. AnnotationConfigApplicationContext：继承了GenericApplicationContext，实现了AnnotationConfigRegistry接口，拥有了以上所有的功能

ClassPathXmlApplicationContext



它也是继承了AbstractApplicationContext，但是相对于AnnotationConfigApplicationContext而言，功能没有AnnotationConfigApplicationContext强大，比如不能注册BeanDefinition

国际化

先定义一个MessageSource:

```
@Bean
public MessageSource messageSource() {
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
    messageSource.setBasename("messages");
```

```
    return messageSource;
}
```

有了这个Bean，你可以在你任意想要进行国际化的地方使用该MessageSource。 同时，因为ApplicationContext也拥有国家化的功能，所以可以直接这么用：

```
context.getMessage("test", null, new Locale("en_CN"))
```

资源加载

ApplicationContext还拥有资源加载的功能，比如，可以直接利用ApplicationContext获取某个文件的内容：

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

Resource resource = context.getResource("file://D:\\IdeaProjects\\spring-
framework\\luban\\src\\main\\java\\com\\luban\\entity\\User.java");
System.out.println(resource.contentLength());
```

你可以想想，如果你不使用ApplicationContext，而是自己来实现这个功能，就比较费时间了。

还比如你可以：

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

Resource resource = context.getResource("file://D:\\IdeaProjects\\spring-framework-
5.3.10\\tuling\\src\\main\\java\\com\\zhouyu\\service\\UserService.java");
System.out.println(resource.contentLength());
System.out.println(resource.getFilename());

Resource resource1 = context.getResource("https://www.baidu.com");
System.out.println(resource1.contentLength());
System.out.println(resource1.getURL());

Resource resource2 = context.getResource("classpath:spring.xml");
System.out.println(resource2.contentLength());
System.out.println(resource2.getURL());
```

还可以一次性获取多个：

```
Resource[] resources = context.getResources("classpath:com/zhouyu/*.class");
for (Resource resource : resources) {
    System.out.println(resource.contentLength());
    System.out.println(resource.getFilename());
}
```

获取运行时环境

```
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

Map<String, Object> systemEnvironment = context.getEnvironment().getSystemEnvironment();
System.out.println(systemEnvironment);

System.out.println("=======");

Map<String, Object> systemProperties = context.getEnvironment().getSystemProperties();
System.out.println(systemProperties);

System.out.println("=======");

MutablePropertySources propertySources = context.getEnvironment().getPropertySources();
System.out.println(propertySources);

System.out.println("=======");

System.out.println(context.getEnvironment().getProperty("NO_PROXY"));
System.out.println(context.getEnvironment().getProperty("sun.jnu.encoding"));
System.out.println(context.getEnvironment().getProperty("zhouyu"));
```

注意，可以利用

```
@PropertySource("classpath:spring.properties")
```

来使得某个properties文件中的参数添加到运行时环境中

事件发布

先定义一个事件监听器

```
@Bean
public ApplicationListener applicationListener() {
    return new ApplicationListener() {
        @Override
        public void onApplicationEvent(ApplicationEvent event) {
            System.out.println("接收到了一个事件");
        }
    };
}
```

然后发布一个事件：

```
context.publishEvent("kkk");
```

类型转化

在Spring源码中，有可能需要把String转成其他类型，所以在Spring源码中提供了一些技术来更方便的做对象的类型转化，关于类型转化的应用场景， 后续看源码的过程中会遇到很多。

PropertyEditor

这其实是JDK中提供的类型转化工具类

```
public class StringToUserPropertyEditor extends PropertyEditorSupport implements
PropertyEditor {

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        User user = new User();
        user.setName(text);
        this.setValue(user);
    }
}
```

```
StringToUserPropertyEditor propertyEditor = new StringToUserPropertyEditor();
propertyEditor.setAsText("1");
User value = (User) propertyEditor.getValue();
System.out.println(value);
```

如何向Spring中注册PropertyEditor：

```
@Bean
public CustomEditorConfigurer customEditorConfigurer() {
    CustomEditorConfigurer customEditorConfigurer = new CustomEditorConfigurer();
    Map<Class<?>, Class<? extends PropertyEditor>> propertyEditorMap = new HashMap<>();

    // 表示StringToUserPropertyEditor可以将String转化成User类型，在Spring源码中，如果发现当前
对象是String，而需要的类型是User，就会使用该PropertyEditor来做类型转化
    propertyEditorMap.put(User.class, StringToUserPropertyEditor.class);
    customEditorConfigurer.setCustomEditors(propertyEditorMap);
    return customEditorConfigurer;
}
```

假设现在有如下Bean:

```
@Component
public class UserService {

    @Value("xxx")
    private User user;

    public void test() {
```

```
        System.out.println(user);
    }

}
```

那么test属性就能正常的完成属性赋值

ConversionService

Spring中提供的类型转化服务，它比PropertyEditor更强大

```
public class StringToUserConverter implements ConditionalGenericConverter {

    @Override
    public boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType) {
        return sourceType.getType().equals(String.class) &&
targetType.getType().equals(User.class);
    }

    @Override
    public Set<ConvertiblePair> getConvertibleTypes() {
        return Collections.singleton(new ConvertiblePair(String.class, User.class));
    }

    @Override
    public Object convert(Object source, TypeDescriptor sourceType, TypeDescriptor
targetType) {
        User user = new User();
        user.setName((String)source);
        return user;
    }
}
```

```
DefaultConversionService conversionService = new DefaultConversionService();
conversionService.addConverter(new StringToUserConverter());
User value = conversionService.convert("1", User.class);
System.out.println(value);
```

如何向Spring中注册ConversionService：

```
@Bean
public ConversionServiceFactoryBean conversionService() {
    ConversionServiceFactoryBean conversionServiceFactoryBean = new
ConversionServiceFactoryBean();
    conversionServiceFactoryBean.setConverters(Collections.singleton(new
StringToUserConverter()));

    return conversionServiceFactoryBean;
}
```

TypeConverter

整合了PropertyEditor和ConversionService的功能，是Spring内部用的

```
SimpleTypeConverter typeConverter = new SimpleTypeConverter();
typeConverter.registerCustomEditor(User.class, new StringToUserPropertyEditor());
//typeConverter.setConversionService(conversionService);
User value = typeConverter.convertIfNecessary("1", User.class);
System.out.println(value);
```

OrderComparator

OrderComparator是Spring所提供的一种比较器，可以用来根据@Order注解或实现Ordered接口来执行值进行笔记，从而可以进行排序。

比如：

```
public class A implements Ordered {

    @Override
    public int getOrder() {
        return 3;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName();
    }
}
```

```
public class B implements Ordered {

    @Override
    public int getOrder() {
        return 2;
    }

    @Override
    public String toString() {
        return this.getClass().getSimpleName();
    }
}
```

```
public class Main {

    public static void main(String[] args) {
```

```
        A a = new A(); // order=3
        B b = new B(); // order=2

        OrderComparator comparator = new OrderComparator();
        System.out.println(comparator.compare(a, b));  // 1

        List list = new ArrayList<>();
        list.add(a);
        list.add(b);

        // 按order值升序排序
        list.sort(comparator);

        System.out.println(list);  // B, A
    }
}
```

另外，Spring中还提供了一个OrderComparator的子类：
AnnotationAwareOrderComparator，它支持用@Order来指定order值。比如：

```
@Order(3)
public class A {

    @Override
    public String toString() {
        return this.getClass().getSimpleName();
    }

}
```

```
@Order(2)
public class B {

    @Override
    public String toString() {
        return this.getClass().getSimpleName();
    }

}
```

```
public class Main {

    public static void main(String[] args) {
        A a = new A(); // order=3
        B b = new B(); // order=2

        AnnotationAwareOrderComparator comparator = new AnnotationAwareOrderComparator();
        System.out.println(comparator.compare(a, b)); // 1

        List list = new ArrayList<>();
        list.add(a);
        list.add(b);
```

```
        // 按order值升序排序
        list.sort(comparator);

        System.out.println(list); // B, A
    }
}
```

BeanPostProcessor

BeanPostProcess表示Bena的后置处理器，我们可以定义一个或多个BeanPostProcessor，比如通过一下代码定义一个BeanPostProcessor：

```
@Component
public class ZhouyuBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        if ("userService".equals(beanName)) {
            System.out.println("初始化前");
        }

        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        if ("userService".equals(beanName)) {
            System.out.println("初始化后");
        }

        return bean;
    }
}
```

一个BeanPostProcessor可以在**任意一个Bean**的**初始化之前**以及**初始化之后**去额外的做一些用户自定义的逻辑，当然，我们可以通过判断beanName来进行针对性处理（针对某个Bean，或某部分Bean）。

我们可以通过定义BeanPostProcessor来干涉Spring创建Bean的过程。

BeanFactoryPostProcessor

BeanFactoryPostProcessor表示Bean工厂的后置处理器，其实和BeanPostProcessor类似，BeanPostProcessor是干涉Bean的创建过程，BeanFactoryPostProcessor是干涉BeanFactory的创建过程。比如，我们可以这样定义一个BeanFactoryPostProcessor：

```
@Component
public class ZhouyuBeanFactoryPostProcessor implements BeanFactoryPostProcessor {

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
throws BeansException {
        System.out.println("加工beanFactory");
    }
}
```

我们可以在postProcessBeanFactory()方法中对BeanFactory进行加工。

FactoryBean

上面提到，我们可以通过BeanPostPorcessor来干涉Spring创建Bean的过程，但是如果我们想一个Bean完完全全由我们来创造，也是可以的，比如通过FactoryBean：

```
@Component
public class ZhouyuFactoryBean implements FactoryBean {

    @Override
    public Object getObject() throws Exception {
        UserService userService = new UserService();

        return userService;
    }

    @Override
    public Class<?> getObjectType() {
        return UserService.class;
    }
}
```

通过上面这段代码，我们自己创造了一个UserService对象，并且它将成为Bean。但是通过这种方式创造出来的UserService的Bean，**只会经过初始化后**，其他Spring的生命周期步骤是不会经过的，比如依赖注入。

有同学可能会想到，通过@Bean也可以自己生成一个对象作为Bean，那么和FactoryBean的区别是什么呢？其实在很多场景下他俩是可以替换的，但是站在原理层面来说的，区别很明显，@Bean定义的Bean是会经过完整的Bean生命周期的。

ExcludeFilter和IncludeFilter

这两个Filter是Spring扫描过程中用来过滤的。ExcludeFilter表示**排除过滤器**，IncludeFilter表示**包含过滤器**。

比如以下配置，表示扫描com.zhouyu这个包下面的所有类，但是排除UserService类，也就是就算它上面有@Component注解也不会成为Bean。

```
@ComponentScan(value = "com.zhouyu",
        excludeFilters = {@ComponentScan.Filter(
                type = FilterType.ASSIGNABLE_TYPE,
                classes = UserService.class)}.)
public class AppConfig {
}
```

再比如以下配置，就算UserService类上没有@Component注解，它也会被扫描成为一个Bean。

```
@ComponentScan(value = "com.zhouyu",
        includeFilters = {@ComponentScan.Filter(
                type = FilterType.ASSIGNABLE_TYPE,
                classes = UserService.class)})
public class AppConfig {
}
```

FilterType分为：

1. ANNOTATION：表示是否包含某个注解
2. ASSIGNABLE_TYPE：表示是否是某个类
3. ASPECTJ：表示否是符合某个Aspectj表达式
4. REGEX：表示是否符合某个正则表达式
5. CUSTOM：自定义

在Spring的扫描逻辑中，默认会添加一个AnnotationTypeFilter给includeFilters，表示默认情况下Spring扫描过程中会认为类上有@Component注解的就是Bean。

MetadataReader、ClassMetadata、AnnotationMetadata

在Spring中需要去解析类的信息，比如类名、类中的方法、类上的注解，这些都可以称之为类的元数据，所以Spring中对类的元数据做了抽象，并提供了一些工具类。

MetadataReader表示类的元数据读取器，默认实现类为**SimpleMetadataReader**。比如：

```
public class Test {

    public static void main(String[] args) throws IOException {
        SimpleMetadataReaderFactory simpleMetadataReaderFactory = new
SimpleMetadataReaderFactory();

        // 构造一个MetadataReader
        MetadataReader metadataReader =
```

```
simpleMetadataReaderFactory.getMetadataReader("com.zhouyu.service.UserService");

        // 得到一个ClassMetadata，并获取了类名
        ClassMetadata classMetadata = metadataReader.getClassMetadata();

        System.out.println(classMetadata.getClassName());

        // 获取一个AnnotationMetadata，并获取类上的注解信息
        AnnotationMetadata annotationMetadata = metadataReader.getAnnotationMetadata();
        for (String annotationType : annotationMetadata.getAnnotationTypes()) {
            System.out.println(annotationType);
        }

    }
}
```

需要注意的是，SimpleMetadataReader去解析类时，使用的**ASM技术**。

为什么要使用ASM技术，Spring启动的时候需要去扫描，如果指定的包路径比较宽泛，那么扫描的类是非常多的，那如果在Spring启动时就把这些类全部加载进JVM了，这样不太好，所以使用了ASM技术。