

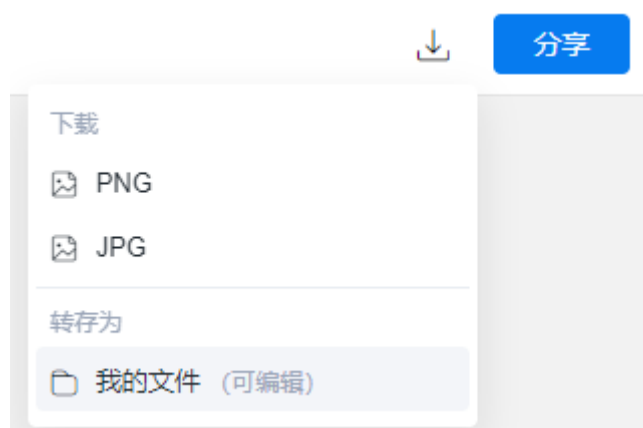
## Spring容器加载流程图：

<https://www.processon.com/view/link/5f15341b07912906d9ae8642?cid=5f15341b7d9c081beac17a19>

课上图（可转存自己编辑）：

<https://www.processon.com/view/link/662dd98a21fb06109ba2e316?cid=6624d18e404949098c6c4526>

访问密码：1nfk



## 1.读取配置

如果配置了这样的Bean:

```
@Component
@Lazy
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@DependsOn
public class UserService {
```

或者

```
<bean class="com.xushu.service.UserService" id="userService" scope="prototype"
lazy-init="true" />
```

```
ans>
factory-bean
factory-method
name
autowire
parent
init-method
abstract
autowire-candidate
depends-on
destroy-method
```

或者

```
@Bean()
@Scope
@Lazy
public UserService userService(){
    return new UserService();
}
```

这些是不同定义bean的方式，他们最终都会生成bean。那Spring为了生成bean代码复用，使用统一的创建流程，所以通过多态方式读取不同的配置会有不同的读取器，读取完后后续创建bean的流程是通用的。

不同的spring容器会使用不同的读取器：

1. AnnotationConfigApplicationContext-**AnnotatedBeanDefinitionReader**
2. ClassPathXmlApplicationContext-**XmlBeanDefinitionReader**

## 1.读取器：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源码中用得更多，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

```
1 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(App
  Config.class);
2
3 AnnotatedBeanDefinitionReader annotatedBeanDefinitionReader = new AnnotatedBeanDefiniti
  onReader(context);
4
5 // 将User.class解析为BeanDefinition
6 annotatedBeanDefinitionReader.register(User.class);
7
8 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

```
1 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(App
  Config.class);
2
3 XmlBeanDefinitionReader xmlBeanDefinitionReader = new XmlBeanDefinitionReader(context);
4 int i = xmlBeanDefinitionReader.loadBeanDefinitions("spring.xml");
5
6 System.out.println(context.getBean("user"));
```


## 2. 扫描器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：

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

```
3
4 ClassPathBeanDefinitionScanner scanner = new ClassPathBeanDefinitionScanner(context);
5 scanner.scan("com.xs");
6
7 System.out.println(context.getBean("userService"));
```

### 3.注册BeanDefinition

 Spring为了使用通用的创建bean流程，不同的配置最终会成为通用的对象：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，我们可以称之为**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：

```
1 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(App
  Config.class);
2
3 // 生成一个BeanDefinition对象，并设置beanClass为用户.class，并注册到ApplicationContext中
4 AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.genericBeanDefinition().g
  etBeanDefinition();
5 beanDefinition.setBeanClass(User.class);
6 context.registerBeanDefinition("user", beanDefinition);
```

7

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

我们还可以通过BeanDefinition设置一个Bean的其他属性

```
1 beanDefinition.setScope("prototype"); // 设置作用域
2 beanDefinition.setInitMethodName("init"); // 设置初始化方法
3 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。

## 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：

```
1 public class Test {
2
3     public static void main(String[] args) throws IOException {
4         SimpleMetadataReaderFactory simpleMetadataReaderFactory = new SimpleMetadataReaderFactory();
5
6         // 构造一个MetadataReader
7         MetadataReader metadataReader = simpleMetadataReaderFactory.getMetadataReader("com.xs.service.UserService");
8
9         // 得到一个ClassMetadata，并获取了类名
10        ClassMetadata classMetadata = metadataReader.getClassMetadata();
11
12        System.out.println(classMetadata.getClassName());
13
14        // 获取一个AnnotationMetadata，并获取类上的注解信息
15        AnnotationMetadata annotationMetadata = metadataReader.getAnnotationMetadata();
16        for (String annotationType : annotationMetadata.getAnnotationTypes()) {
```

```
17         System.out.println(annotationType);
18     }
19
20 }
21 }
```

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。

## 4. 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：

```
1 public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory,
2     HierarchicalBeanFactory,
3
4     MessageSource, ApplicationEventPublisher, ResourcePatternResolver {
5
6     ...
7 }
```

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。

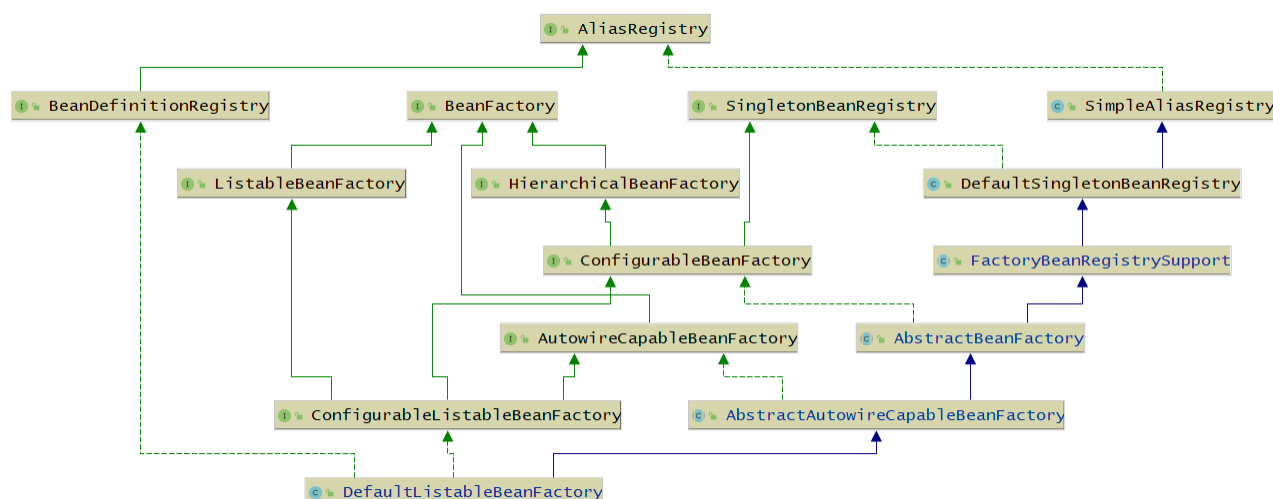
在Spring的源码实现中，当我们new一个ApplicationContext时，其底层会new一个BeanFactory出来，当使用ApplicationContext的某些方法时，比如getBean()，底层调用的是BeanFactory的getBean()方法。

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。

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：

```
1 DefaultListableBeanFactory beanFactory = new DefaultListableBeanFactory();
2
3 AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.genericBeanDefinition().getBeanDefinition();
4 beanDefinition.setBeanClass(User.class);
5
6 beanFactory.registerBeanDefinition("user", beanDefinition);
7
8 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

## 5. 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)



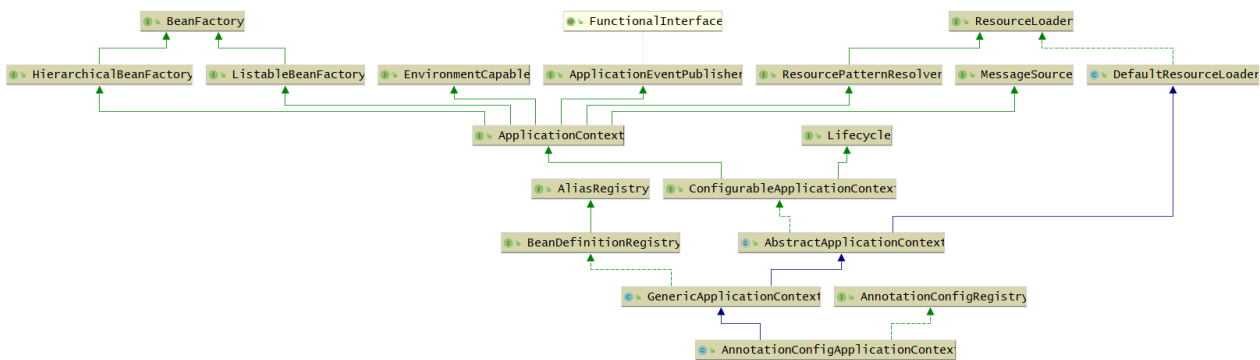
## 6. MessageSource：拥有国际化功能

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



除了这些基本的以外， 还有国际化、资源加载、运行时环境、事件、BeanPostProcessor、BeanFactoryPostProcessor

## 国际化



先定义一个MessageSource:

```
1 @Bean
2 public MessageSource messageSource() {
3     ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
4     messageSource.setBasename("messages");
5     return messageSource;
6 }
```

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：

```
1 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：

```
1 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(App
  Config.class);
2
3 Resource resource = context.getResource("file://D:\\IdeaProjects\\spring-
  framework\\luban\\src\\main\\java\\com\\luban\\entity\\User.java");
4 System.out.println(resource.getLength());
```

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：

```
1 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(App
  Config.class);
2
3 Resource resource = context.getResource("file:///D:\\IdeaProjects\\spring-framework-
  5.3.10\\tuling\\src\\main\\java\\com\\xs\\service\\UserService.java");
4 System.out.println(resource.getLength());
5 System.out.println(resource.getFilename());
6
7 Resource resource1 = context.getResource("https://www.baidu.com");
8 System.out.println(resource1.getLength());
9 System.out.println(resource1.getURL());
10
11 Resource resource2 = context.getResource("classpath:spring.xml");
12 System.out.println(resource2.getLength());
13 System.out.println(resource2.getURL());
```

还可以一次性获取多个：


```
1 Resource[] resources = context.getResources("classpath:com/xs/*.class");
2 for (Resource resource : resources) {
3     System.out.println(resource.getLength());
4     System.out.println(resource.getFilename());
5 }
```

## 获取运行时环境


## Build and run

[Modify options](#)  **Alt+M**



**JRE Alt+J**

java 8 SDK of 'spring.tuling.main' module 


**Use classpath of module Alt+O**

-cp spring.tuling.main 


**Add VM options Alt+V**

-Dfile.encoding=UTF-8 -Dxs=1  

**Main class Alt+C**

com.xushu.all.MainStart 

**Program arguments Alt+R**

Program arguments  

Press Alt for field hints

```
1 AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(App
  Config.class);
2
3 Map<String, Object> systemEnvironment =
  context.getEnvironment().getSystemEnvironment();
4 System.out.println(systemEnvironment);
5
6 System.out.println("=====");
7
8 Map<String, Object> systemProperties = context.getEnvironment().getSystemProperties();
9 System.out.println(systemProperties);
10
11 System.out.println("=====");
12
13 MutablePropertySources propertySources = context.getEnvironment().getPropertySources();
14 System.out.println(propertySources);
15
16 System.out.println("=====");
17
18 System.out.println(context.getEnvironment().getProperty("NO_PROXY"));
19 System.out.println(context.getEnvironment().getProperty("sun.jnu.encoding"));
20 System.out.println(context.getEnvironment().getProperty("xs"));
```

注意，可以利用

```
1 @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

```
1 @Bean
2 public ApplicationListener applicationListener() {
3     return new ApplicationListener() {
4         @Override
5         public void onApplicationEvent(ApplicationEvent event) {
6             System.out.println("接收到了一个事件");
7         }
8     };
9 }
```

然后发布一个事件：

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

## 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：

```
1 @Component
2 public class ZhouyuBeanPostProcessor implements BeanPostProcessor {
3
4     @Override
5     public Object postProcessBeforeInitialization(Object bean, String
        beanName) throws BeansException {
6         if ("userService".equals(beanName)) {
7             System.out.println("初始化前");
8         }
9     }
10 }
```

```

9
10     return bean;
11 }
12
13 @Override
14 public Object postProcessAfterInitialization(Object bean, String
    beanName) throws BeansException {
15     if ("userService".equals(beanName)) {
16         System.out.println("初始化后");
17     }
18
19     return bean;
20 }
21 }

```

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：

```

1 @Component
2 public class ZhouyuBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
3
4     @Override
5     public void postProcessBeanFactory(ConfigurableListableBeanFactory
        beanFactory) throws BeansException {
6         System.out.println("加工beanFactory");
7     }
8 }

```

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BeanPostProcessor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：

```
1  @Component
2  public class ZhouyuFactoryBean implements FactoryBean {
3
4      @Override
5      public Object getObject() throws Exception {
6          UserService userService = new UserService();
7
8          return userService;
9      }
10
11     @Override
12     public Class<?> getObjectType() {
13         return UserService.class;
14     }
15 }
```

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.xs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。



```
1 @ComponentScan(value = "com.xs",
2   excludeFilters = {@ComponentScan.Filter(
3     type = FilterType.ASSIGNABLE_TYPE,
4     classes = UserService.class)})
5 public class AppConfig {
6 }
```

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。

```
1 @ComponentScan(value = "com.xs",
2   includeFilters = {@ComponentScan.Filter(
3     type = FilterType.ASSIGNABLE_TYPE,
4     classes = UserService.class)})
5 public class AppConfig {
6 }
```

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：表示是否是符合某个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。

有道云链接：<https://note.youdao.com/s/JHOqotQ9>