有道云链接： http://note.youdao.com/noteshare?id=e8e3fcbeb1428c2da9e0764c796d2660&sub=A81213EC3FB44E408A12B7273BEA0C37

Spring最重要的功能就是帮助程序员创建对象（也就是IOC），而启动Spring就是为创建Bean对象做准备，所以我们先明白Spring到底是怎么去创建Bean的，也就是先弄明白Bean的生命周期。

Bean的生命周期就是指：**在Spring中，一个Bean是如何生成的，如何销毁的**

Bean生命周期流程图：https://www.processon.com/view/link/5f8588c87d9c0806f27358c1

附带资料JFR介绍：https://zhuanlan.zhihu.com/p/122247741

Bean的生成过程

1. 生成BeanDefinition

Spring启动的时候会进行扫描，会先调用`org.springframework.context.annotation.ClassPathScanningCandidateComponentProvider#scanCandidateComponents(String basePackage)`扫描某个包路径，并得到BeanDefinition的Set集合。

关于Spring启动流程，后续会单独的课详细讲，这里先讲一下Spring扫描的底层实现：

Spring扫描底层流程：https://www.processon.com/view/link/61370ee60e3e7412ecd95d43

1. 首先，通过ResourcePatternResolver获得指定包路径下的所有`.class`文件（Spring源码中将此文件包装成了Resource对象）
2. 遍历每个Resource对象
3. 利用MetadataReaderFactory解析Resource对象得到MetadataReader（在Spring源码中MetadataReaderFactory具体的实现类为CachingMetadataReaderFactory，MetadataReader的具体实现类为SimpleMetadataReader）
4. 利用MetadataReader进行excludeFilters和includeFilters，以及条件注解@Conditional的筛选（条件注解并不能理解：某个类上是否存在@Conditional注解，如果存在则调用注解中所指定的类的match方法进行匹配，匹配成功则通过筛选，匹配失败则pass掉。）
5. 筛选通过后，基于metadataReader生成ScannedGenericBeanDefinition
6. 再基于metadataReader判断是不是对应的类是不是接口或抽象类
7. 如果筛选通过，那么就表示扫描到了一个Bean，将ScannedGenericBeanDefinition加入结果集

MetadataReader表示类的元数据读取器，主要包含了一个AnnotationMetadata，功能有

1. 获取类的名字、
2. 获取父类的名字
3. 获取所实现的所有接口名
4. 获取所有内部类的名字

5. 判断是不是抽象类
6. 判断是不是接口
7. 判断是不是一个注解
8. 获取拥有某个注解的方法集合
9. 获取类上添加的所有注解信息
10. 获取类上添加的所有注解类型集合

值得注意的是，CachingMetadataReaderFactory解析某个.class文件得到MetadataReader对象是利用的**ASM**技术，并没有加载这个类到JVM。并且，最终得到的ScannedGenericBeanDefinition对象，**beanClass属性存储的是当前类的名字，而不是class对象。**（beanClass属性的类型是Object，它即可以存储类的名字，也可以存储class对象）

最后，上面是说的通过扫描得到BeanDefinition对象，我们还可以通过直接定义BeanDefinition，或解析spring.xml文件的<bean/>，或者@Bean注解得到BeanDefinition对象。（后续课程会分析@Bean注解是怎么生成BeanDefinition的）。

2. 合并BeanDefinition

通过扫描得到所有BeanDefinition之后，就可以根据BeanDefinition创建Bean对象了，但是在Spring中支持父子BeanDefinition，和Java父子类类似，但是完全不是一回事。

父子BeanDefinition实际用的比较少，使用是这样的，比如：

```
<bean id="parent" class="com.zhouyu.service.Parent" scope="prototype"/>
<bean id="child" class="com.zhouyu.service.Child"/>
```

这么定义的情况下，child是单例Bean。

```
<bean id="parent" class="com.zhouyu.service.Parent" scope="prototype"/>
<bean id="child" class="com.zhouyu.service.Child" parent="parent"/>
```

但是这么定义的情况下，child就是原型Bean了。

因为child的父BeanDefinition是parent，所以会继承parent上所定义的scope属性。

而在根据child来生成Bean对象之前，需要进行BeanDefinition的合并，得到完整的child的BeanDefinition。

3. 加载类

BeanDefinition合并之后，就可以去创建Bean对象了，而创建Bean就必须实例化对象，而实例化就必须先加载当前BeanDefinition所对应的class，在AbstractAutowireCapableBeanFactory类的createBean()方法中，一开始就会调用：

```
Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
```

这行代码就是去加载类，该方法是这么实现的：

```
if (mbd.hasBeanClass()) {
    return mbd.getBeanClass();
}
if (System.getSecurityManager() != null) {
    return AccessController.doPrivileged((PrivilegedExceptionAction<Class<?>>) () ->
        doResolveBeanClass(mbd, typesToMatch), getAccessControlContext());
    }
else {
    return doResolveBeanClass(mbd, typesToMatch);
}
```

```
public boolean hasBeanClass() {
    return (this.beanClass instanceof Class);
}
```

如果beanClass属性的类型是Class，那么就直接返回，如果不是，则会根据类名进行加载（doResolveBeanClass方法所做的事情）

会利用BeanFactory所设置的类加载器来加载类，如果没有设置，则默认使用**ClassUtils.getDefaultClassLoader()**所返回的类加载器来加载。

ClassUtils.getDefaultClassLoader()

1. 优先返回当前线程中的ClassLoader
2. 线程中类加载器为null的情况下，返回ClassUtils类的类加载器
3. 如果ClassUtils类的类加载器为空，那么则表示是Bootstrap类加载器加载的ClassUtils类，那么则返回系统类加载器

4. 实例化前

当前BeanDefinition对应的类成功加载后，就可以实例化对象了，但是...

在Spring中，实例化对象之前，Spring提供了一个扩展点，允许用户来控制是否在某个或某些Bean实例化之前做一些启动动作。这个扩展点叫**InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation()**。比如：

```
@Component
public class ZhouyuBeanPostProcessor implements InstantiationAwareBeanPostProcessor {

    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName)
throws BeansException {
        if ("userService".equals(beanName)) {
```

```
            System.out.println("实例化前");
        }
        return null;
    }
}
```

如上代码会导致，在userService这个Bean实例化前，会进行打印。

值得注意的是，postProcessBeforeInstantiation()是有返回值的，如果这么实现：

```
@Component
public class ZhouyuBeanPostProcessor implements InstantiationAwareBeanPostProcessor {

    @Override
    public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName)
throws BeansException {
        if ("userService".equals(beanName)) {
            System.out.println("实例化前");
            return new UserService();
        }
        return null;
    }
}
```

userService这个Bean，在实例化前会直接返回一个由我们所定义的UserService对象。如果是这样，表示不需要Spring来实例化了，并且后续的Spring依赖注入也不会进行了，会跳过一些步骤，直接执行初始化后这一步。

5. 实例化

在这个步骤中就会根据BeanDefinition去创建一个对象了。

5.1 Supplier创建对象

首先判断BeanDefinition中是否设置了Supplier，如果设置了则调用Supplier的get()得到对象。

得直接使用BeanDefinition对象来设置Supplier，比如：

```
AbstractBeanDefinition beanDefinition =
BeanDefinitionBuilder.genericBeanDefinition().getBeanDefinition();
beanDefinition.setInstanceSupplier(new Supplier<Object>() {
    @Override
    public Object get() {
        return new UserService();
    }
});
context.registerBeanDefinition("userService", beanDefinition);
```

5.2 工厂方法创建对象

如果没有设置Supplier，则检查BeanDefinition中是否设置了factoryMethod，也就是工厂方法，有两种方式可以设置factoryMethod，比如：

方式一：

```
<bean id="userService" class="com.zhouyu.service.UserService" factory-
method="createUserService" />
```

对应的UserService类为：

```
public class UserService {

    public static UserService createUserService() {
        System.out.println("执行createUserService()");
        UserService userService = new UserService();
        return userService;
    }

    public void test() {
        System.out.println("test");
    }

}
```

方式二：

```
<bean id="commonService" class="com.zhouyu.service.CommonService"/>
<bean id="userService1" factory-bean="commonService" factory-method="createUserService"
/>
```

对应的CommonService的类为：

```
public class CommonService {

    public UserService createUserService() {
        return new UserService();
    }
}
```

Spring发现当前BeanDefinition方法设置了工厂方法后，就会区分这两种方式，然后调用工厂方法得到对象。

值得注意的是，我们通过@Bean所定义的BeanDefinition，是存在factoryMethod和factoryBean的，也就是和上面的方式二非常类似，@Bean所注解的方法就是factoryMethod，AppConfig对象

就是factoryBean。如果@Bean所所注解的方法是static的，那么对应的就是方式一。

5.3 推断构造方法

第一节已经讲过一遍大概原理了，后面有一节课单独分析源码实现。推断完构造方法后，就会使用构造方法来进行实例化了。

额外的，在推断构造方法逻辑中除开会去选择构造方法以及查找入参对象意外，会还判断是否在对应的类中是否存在使用**@Lookup注解**了方法。如果存在则把该方法封装为LookupOverride对象并添加到BeanDefinition中。

在实例化时，如果判断出来当前BeanDefinition中没有LookupOverride，那就直接用构造方法反射得到一个实例对象。如果存在LookupOverride对象，也就是类中存在@Lookup注解了的方法，那就会生成一个代理对象。

@Lookup注解就是**方法注入**，使用demo如下：

```
@Component
public class UserService {

    private OrderService orderService;

    public void test() {
        OrderService orderService = createOrderService();
        System.out.println(orderService);
    }

    @Lookup("orderService")
    public OrderService createOrderService() {
        return null;
    }

}
```

6. BeanDefinition的后置处理

Bean对象实例化出来之后，接下来就应该给对象的属性赋值了。在真正给属性赋值之前，Spring又提供了一个扩展点**MergedBeanDefinitionPostProcessor.postProcessMergedBeanDefinition()**，可以对此时的BeanDefinition进行加工，比如：

```
@Component
public class ZhouyuMergedBeanDefinitionPostProcessor implements
MergedBeanDefinitionPostProcessor {

    @Override
    public void postProcessMergedBeanDefinition(RootBeanDefinition beanDefinition,
Class<?> beanType, String beanName) {
```

```
        if ("userService".equals(beanName)) {
            beanDefinition.getPropertyValues().add("orderService", new OrderService());
        }
    }
}
```

在Spring源码中，AutowiredAnnotationBeanPostProcessor就是一个MergedBeanDefinitionPostProcessor，它的postProcessMergedBeanDefinition()中会去查找注入点，并缓存在AutowiredAnnotationBeanPostProcessor对象的一个Map中（injectionMetadataCache）。

7. 实例化后

在处理完BeanDefinition后，Spring又设计了一个扩展点：**InstantiationAwareBeanPostProcessor.postProcessAfterInstantiation()**，比如：

```
@Component
public class ZhouyuInstantiationAwareBeanPostProcessor implements
InstantiationAwareBeanPostProcessor {

    @Override
    public boolean postProcessAfterInstantiation(Object bean, String beanName) throws
BeansException {

        if ("userService".equals(beanName)) {
            UserService userService = (UserService) bean;
            userService.test();
        }

        return true;
    }
}
```

上述代码就是对userService所实例化出来的对象进行处理。

这个扩展点，在Spring源码中基本没有怎么使用。

8. 自动注入

这里的自动注入指的是Spring的自动注入，后续依赖注入课程中单独讲

9. 处理属性

这个步骤中，就会处理@Autowired、@Resource、@Value等注解，也是通过**InstantiationAwareBeanPostProcessor.postProcessProperties()**扩展点来实现的，比如我们甚至可以实现一个自己的自动注入功能，比如：

```
@Component
public class ZhouyuInstantiationAwareBeanPostProcessor implements
InstantiationAwareBeanPostProcessor {

    @Override
    public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String
beanName) throws BeansException {
        if ("userService".equals(beanName)) {
            for (Field field : bean.getClass().getFields()) {
                if (field.isAnnotationPresent(ZhouyuInject.class)) {
                    field.setAccessible(true);
                    try {
                        field.set(bean, "123");
                    } catch (IllegalAccessException e) {
                        e.printStackTrace();
                    }
                }
            }
        }

        return pvs;
    }
}
```

关于@Autowired、@Resource、@Value的底层源码，会在后续的依赖注入课程中详解。

10. 执行Aware

完成了属性赋值之后，Spring会执行一些回调，包括：

1. BeanNameAware：回传beanName给bean对象。
2. BeanClassLoaderAware：回传classLoader给bean对象。
3. BeanFactoryAware：回传beanFactory给对象。

11. 初始化前

初始化前，也是Spring提供的一个扩展点：
BeanPostProcessor.postProcessBeforeInitialization()，比如

```
@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;
```

```
    }
}
```

利用初始化前，可以对进行了依赖注入的Bean进行处理。

在Spring源码中：

1. InitDestroyAnnotationBeanPostProcessor会在初始化前这个步骤中执行@PostConstruct的方法，
2. ApplicationContextAwareProcessor会在初始化前这个步骤中进行其他Aware的回调：
    i. EnvironmentAware：回传环境变量
    ii. EmbeddedValueResolverAware：回传占位符解析器
    iii. ResourceLoaderAware：回传资源加载器
    iv. ApplicationEventPublisherAware：回传事件发布器
    v. MessageSourceAware：回传国际化资源
    vi. ApplicationStartupAware：回传应用其他监听对象，可忽略
    vii. ApplicationContextAware：回传Spring容器ApplicationContext

12. 初始化

1. 查看当前Bean对象是否实现了InitializingBean接口，如果实现了就调用其afterPropertiesSet()方法
2. 执行BeanDefinition中指定的初始化方法

13. 初始化后

这是Bean创建生命周期中的最后一个步骤，也是Spring提供的一个扩展点：**BeanPostProcessor.postProcessAfterInitialization()**，比如：

```
@Component
public class ZhouyuBeanPostProcessor implements BeanPostProcessor {

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

        return bean;
    }
}
```

可以在这个步骤中，对Bean最终进行处理，Spring中的**AOP就是基于初始化后实现**的，**初始化后返回的对象才是最终的Bean对象。**

总结BeanPostProcessor

1. InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation()
2. 实例化
3. MergedBeanDefinitionPostProcessor.postProcessMergedBeanDefinition()
4. InstantiationAwareBeanPostProcessor.postProcessAfterInstantiation()
5. 自动注入
6. InstantiationAwareBeanPostProcessor.postProcessProperties()
7. Aware对象
8. BeanPostProcessor.postProcessBeforeInitialization()
9. 初始化
10. BeanPostProcessor.postProcessAfterInitialization()