有道云链接：http://note.youdao.com/noteshare?id=bb7e5f810ade4bab72d7af5b9d6c1e97&sub=B26AAB65D991490D9C547529F6AED86B（复制链接到浏览器的时候注意转行的空格）

依赖注入底层原理流程图： https://www.processon.com/view/link/5f899fa5f346fb06e1d8f570

Spring中到底有几种依赖注入的方式?

首先分两种：

1. 手动注入
2. 自动注入

手动注入

在XML中定义Bean时，就是手动注入，因为是**程序员手动给某个属性指定了值。**

```
<bean name="userService" class="com.luban.service.UserService">
    <property name="orderService" ref="orderService"/>
</bean>
```

上面这种底层是通过**set方法**进行注入。

```
<bean name="userService" class="com.luban.service.UserService">
    <constructor-arg index="0" ref="orderService"/>
</bean>
```

上面这种底层是通过**构造方法**进行注入。

所以手动注入的底层也就是分为两种：

1. set方法注入
2. 构造方法注入

自动注入

自动注入又分为两种：

1. XML的autowire自动注入
2. @Autowired注解的自动注入

XML的autowire自动注入

在XML中，我们可以在定义一个Bean时去指定这个Bean的自动注入模式：

1. byType
2. byName
3. constructor
4. default
5. no

比如：

```
<bean id="userService" class="com.luban.service.UserService" autowire="byType"/>
```

这么写，表示Spring会自动的给userService中所有的属性自动赋值（**不需要**这个属性上有@Autowired注解，但需要这个属性有对应的**set方法**）。

在创建Bean的过程中，在填充属性时，Spring会去解析当前类，把**当前类的所有方法**都解析出来，Spring会去解析每个方法得到对应的PropertyDescriptor对象，PropertyDescriptor中有几个属性：

1. **name：这个name并不是方法的名字，而是拿方法名字进过处理后的名字**
    i. **如果方法名字以“get”开头，比如“getXXX”,那么name=XXX**
    ii. **如果方法名字以“is”开头，比如“isXXX”,那么name=XXX**
    iii. **如果方法名字以“set”开头，比如“setXXX”,那么name=XXX**
2. **readMethodRef：表示get方法的Method对象的引用**
3. **readMethodName：表示get方法的名字**
4. **writeMethodRef：表示set方法的Method对象的引用**
5. **writeMethodName：表示set方法的名字**
6. **propertyTypeRef：如果有get方法那么对应的就是返回值的类型，如果是set方法那么对应的就是set方法中唯一参数的类型**

get方法的定义是： 方法参数个数为0个，并且 （方法名字以"get"开头 或者 方法名字以"is"开头并且方法的返回类型为boolean）

set方法的定义是：方法参数个数为1个，并且 （方法名字以"set"开头并且方法返回类型为void）

所以，Spring在通过byName的自动填充属性时流程是：

1. 找到所有set方法所对应的XXX部分的名字
2. 根据XXX部分的名字去获取bean

Spring在通过byType的自动填充属性时流程是：

1. 获取到set方法中的唯一参数的参数类型，并且根据该类型去容器中获取bean
2. 如果找到多个，会报错。

以上，分析了autowire的byType和byName情况，那么接下来分析constructor，constructor表示通过构造方法注入，其实这种情况就比较简单了，没有byType和byName那么复杂。

如果是constructor，那么就可以不写set方法了，当某个bean是通过构造方法来注入时，spring利用构造方法的参数信息从Spring容器中去找bean，找到bean之后作为参数传给构造方法，从而实例化得到一个bean对象，并完成属性赋值（属性赋值的代码得程序员来写）。

我们这里先不考虑一个类有多个构造方法的情况，后面单独讲**推断构造方法。**我们这里只考虑只有一个有参构造方法。

其实构造方法注入相当于byType+byName，普通的byType是根据set方法中的参数类型去找bean，找到多个会报错，而constructor就是通过构造方法中的参数类型去找bean，如果找到多个会根据参数名确定。

另外两个：

1. no，表示关闭autowire
2. default，表示默认值，我们一直演示的某个bean的autowire，而也可以直接在<beans>标签中设置autowire，如果设置了，那么<bean>标签中设置的autowire如果为default，那么则会用<beans>标签中设置的autowire。

可以发现XML中的自动注入是挺强大的，那么问题来了，**为什么我们平时都是用的@Autowired注解呢？而没有用上文说的这种自动注入方式呢?**

@Autowired注解相当于XML中的autowire属性的**注解方式的替代。**这是在官网上有提到的。

```
Essentially, the @Autowired annotation provides the same capabilities as described in
Autowiring Collaborators but with more fine-grained control and wider applicability
```

翻译一下： 从本质上讲，@Autowired注解提供了与autowire相同的功能，但是拥有更细粒度的控制和更广泛的适用性。

注意：**更细粒度的控制。**

XML中的autowire控制的是整个bean的所有属性，而@Autowired注解是直接写在某个属性、某个set方法、某个构造方法上的。

再举个例子，如果一个类有多个构造方法，那么如果用XML的autowire=constructor，你无法控制到底用哪个构造方法，而你可以用@Autowired注解来直接指定你想用哪个构造方法。

同时，用@Autowired注解，还可以控制，哪些属性想被自动注入，哪些属性不想，这也是细粒度的控制。

但是@Autowired无法区分byType和byName，@Autowired是先byType，如果找到多个则byName。

那么XML的自动注入底层其实也就是:

1. set方法注入
2. 构造方法注入

@Autowired注解的自动注入

上文说了@Autowired注解，是byType和byName的结合。

@Autowired注解可以写在：

1. 属性上：先根据**属性类型**去找Bean，如果找到多个再根据**属性名**确定一个
2. 构造方法上：先根据方法**参数类型**去找Bean，如果找到多个再根据**参数名**确定一个
3. set方法上：先根据方法**参数类型**去找Bean，如果找到多个再根据**参数名**确定一个

而这种底层到了：

1. 属性注入
2. set方法注入
3. 构造方法注入

寻找注入点

在创建一个Bean的过程中，Spring会利用AutowiredAnnotationBeanPostProcessor的**postProcessMergedBeanDefinition()**找出注入点并缓存，找注入点的流程为：

1. 遍历当前类的所有的属性字段Field
2. 查看字段上是否存在@Autowired、@Value、@Inject中的其中任意一个，存在则认为该字段是一个注入点
3. 如果字段是static的，则不进行注入
4. 获取@Autowired中的required属性的值
5. 将字段信息构造成一个**AutowiredFieldElement对象**，作为一个**注入点对象**添加到currElements集合中。
6. 遍历当前类的所有方法Method
7. 判断当前Method是否是**桥接方法**，如果是找到原方法
8. 查看方法上是否存在@Autowired、@Value、@Inject中的其中任意一个，存在则认为该方法是一个注入点
9. 如果方法是static的，则不进行注入
10. 获取@Autowired中的required属性的值
11. 将方法信息构造成一个**AutowiredMethodElement对象**，作为一个**注入点对象**添加到currElements集合中。
12. 遍历完当前类的字段和方法后，将**遍历父类**的，直到没有父类。
13. 最后将currElements集合封装成一个InjectionMetadata对象，作为当前Bean对于的注入点集合对象，并缓存。

static的字段或方法为什么不支持

```
@Component
@Scope("prototype")
public class OrderService {


}
```

```
@Component
@Scope("prototype")
public class UserService  {

    @Autowired
    private static OrderService orderService;

    public void test() {
        System.out.println("test123");
    }

}
```

看上面代码，UserService和OrderService都是原型Bean，假设Spring支持static字段进行自动注入，那么现在调用两次

1. UserService userService1 = context.getBean("userService")
2. UserService userService2 = context.getBean("userService")

问此时，userService1的orderService值是什么？还是它自己注入的值吗？

答案是不是，一旦userService2 创建好了之后，static orderService字段的值就发生了修改了，从而出现bug。

桥接方法

```
public interface UserInterface<T> {
    void setOrderService(T t);
}
```

```
@Component
public class UserService implements UserInterface<OrderService> {

    private OrderService orderService;

    @Override
    @Autowired
```

```
    public void setOrderService(OrderService orderService) {
        this.orderService = orderService;
    }

    public void test() {
        System.out.println("test123");
    }

}
```

UserService对应的字节码为：

```
// class version 52.0 (52)
// access flags 0x21
// signature
Ljava/lang/Object;Lcom/zhouyu/service/UserInterface<Lcom/zhouyu/service/OrderService;>;
// declaration: com/zhouyu/service/UserService implements
com.zhouyu.service.UserInterface<com.zhouyu.service.OrderService>
public class com/zhouyu/service/UserService implements com/zhouyu/service/UserInterface {

  // compiled from: UserService.java

  @Lorg/springframework/stereotype/Component;()

  // access flags 0x2
  private Lcom/zhouyu/service/OrderService; orderService

  // access flags 0x1
  public <init>()V
   L0
    LINENUMBER 12 L0
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V
    RETURN
   L1
    LOCALVARIABLE this Lcom/zhouyu/service/UserService; L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1

  // access flags 0x1
  public setOrderService(Lcom/zhouyu/service/OrderService;)V
  @Lorg/springframework/beans/factory/annotation/Autowired;()
   L0
    LINENUMBER 19 L0
    ALOAD 0
    ALOAD 1
    PUTFIELD com/zhouyu/service/UserService.orderService :
Lcom/zhouyu/service/OrderService;
   L1
    LINENUMBER 20 L1
    RETURN
   L2
    LOCALVARIABLE this Lcom/zhouyu/service/UserService; L0 L2 0
    LOCALVARIABLE orderService Lcom/zhouyu/service/OrderService; L0 L2 1
    MAXSTACK = 2
    MAXLOCALS = 2
```

```
  // access flags 0x1
  public test()V
   L0
    LINENUMBER 23 L0
    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
    LDC "test123"
    INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
   L1
    LINENUMBER 24 L1
    RETURN
   L2
    LOCALVARIABLE this Lcom/zhouyu/service/UserService; L0 L2 0
    MAXSTACK = 2
    MAXLOCALS = 1

  // access flags 0x1041
  public synthetic bridge setOrderService(Ljava/lang/Object;)V
  @Lorg/springframework/beans/factory/annotation/Autowired;()
   L0
    LINENUMBER 11 L0
    ALOAD 0
    ALOAD 1
    CHECKCAST com/zhouyu/service/OrderService
    INVOKEVIRTUAL com/zhouyu/service/UserService.setOrderService
(Lcom/zhouyu/service/OrderService;)V
    RETURN
   L1
    LOCALVARIABLE this Lcom/zhouyu/service/UserService; L0 L1 0
    MAXSTACK = 2
    MAXLOCALS = 2
}
```

可以看到在UserSerivce的字节码中有两个setOrderService方法：

1. public setOrderService(Lcom/zhouyu/service/OrderService;)V
2. public synthetic bridge setOrderService(Ljava/lang/Object;)V

并且都是存在@Autowired注解的。

所以在Spring中需要处理这种情况，当遍历到桥接方法时，得找到原方法。

注入点进行注入

Spring在AutowiredAnnotationBeanPostProcessor的**postProcessProperties()**方法中，会遍历所找到的注入点依次进行注入。

字段注入

1. 遍历所有的**AutowiredFieldElement对象。**
2. 将对应的字段封装为**DependencyDescriptor对象。**

3. 调用BeanFactory的resolveDependency()方法，传入**DependencyDescriptor对象**，进行依赖查找，找到当前字段所匹配的Bean对象。
4. 将**DependencyDescriptor对象**和所找到的**结果对象beanName**封装成一个**ShortcutDependencyDescriptor对象**作为缓存，比如如果当前Bean是原型Bean，那么下次再来创建该Bean时，就可以直接拿缓存的结果对象beanName去BeanFactory中去那bean对象了，不用再次进行查找了
5. 利用反射将结果对象赋值给字段。

Set方法注入

1. 遍历所有的**AutowiredMethodElement对象**
2. 遍历将对应的方法的参数，将每个参数封装成**MethodParameter对象**
3. 将**MethodParameter对象**封装为**DependencyDescriptor对象**
4. 调用BeanFactory的resolveDependency()方法，传入**DependencyDescriptor对象**，进行依赖查找，找到当前方法参数所匹配的Bean对象。
5. 将**DependencyDescriptor对象**和所找到的**结果对象beanName**封装成一个**ShortcutDependencyDescriptor对象**作为缓存，比如如果当前Bean是原型Bean，那么下次再来创建该Bean时，就可以直接拿缓存的结果对象beanName去BeanFactory中去那bean对象了，不用再次进行查找了
6. 利用反射将找到的所有结果对象传给当前方法，并执行。