Spring基础 - Spring核心之控制反转(IOC)

引入

> 我们在Spring基础 - Spring简单例子引入Spring的核心中向你展示了IoC的基础含义，同时以此发散了一些IoC相关知识点。

1. Spring框架管理这些Bean的创建工作，即由用户管理Bean转变为框架管理Bean，这个就叫**控制反转 - Inversion of Control (IoC)**
2. Spring 框架托管创建的Bean放在哪里呢？ 这便是**IoC Container**;
3. Spring 框架为了更好让用户配置Bean，必然会引入**不同方式来配置Bean？ 这便是xml配置，Java配置，注解配置**等支持
4. Spring 框架既然接管了Bean的生成，必然需要**管理整个Bean的生命周期**等；
5. 应用程序代码从Ioc Container中获取依赖的Bean，注入到应用程序中，这个过程叫 **依赖注入(Dependency Injection，DI)**； 所以说控制反转是通过依赖注入实现的，其实它们是同一个概念的不同角度描述。通俗来说就是**IoC是设计思想，DI是实现方式**
6. 在依赖注入时，有哪些方式呢？这就是构造器方式，@Autowired, @Resource, @Qualifier... 同时Bean之间存在依赖（可能存在先后顺序问题，以及**循环依赖问题**等）

本节将在此基础上进一步解读IOC的含义以及IOC的使用方式；后续的文章还将深入IOC的实现原理：

- Spring进阶- Spring IOC实现原理详解之IOC体系结构设计
- Spring进阶- Spring IOC实现原理详解之IOC初始化流程
- Spring进阶- Spring IOC实现原理详解之Bean的注入和生命周期

如何理解IoC

如果你有精力看英文，首推 Martin Fowler大师的 Inversion of Control Containers and the Dependency Injection pattern；其次IoC作为一种设计思想，不要过度解读，而是应该简化理解，所以我这里也整合了 张开涛早前的博客IoC基础并加入了自己的理解。

Spring Bean是什么

> IoC Container管理的是Spring Bean， 那么Spring Bean是什么呢?

Spring里面的bean就类似是定义的一个组件，而这个组件的作用就是实现某个功能的，这里所定义的bean就相当于给了你一个更为简便的方法来调用这个组件去实现你要完成的功能。

IoC是什么

> Ioc—Inversion of Control，即"控制反转"，**不是什么技术，而是一种设计思想。**在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。

我们来深入分析一下：

- **谁控制谁，控制什么**?

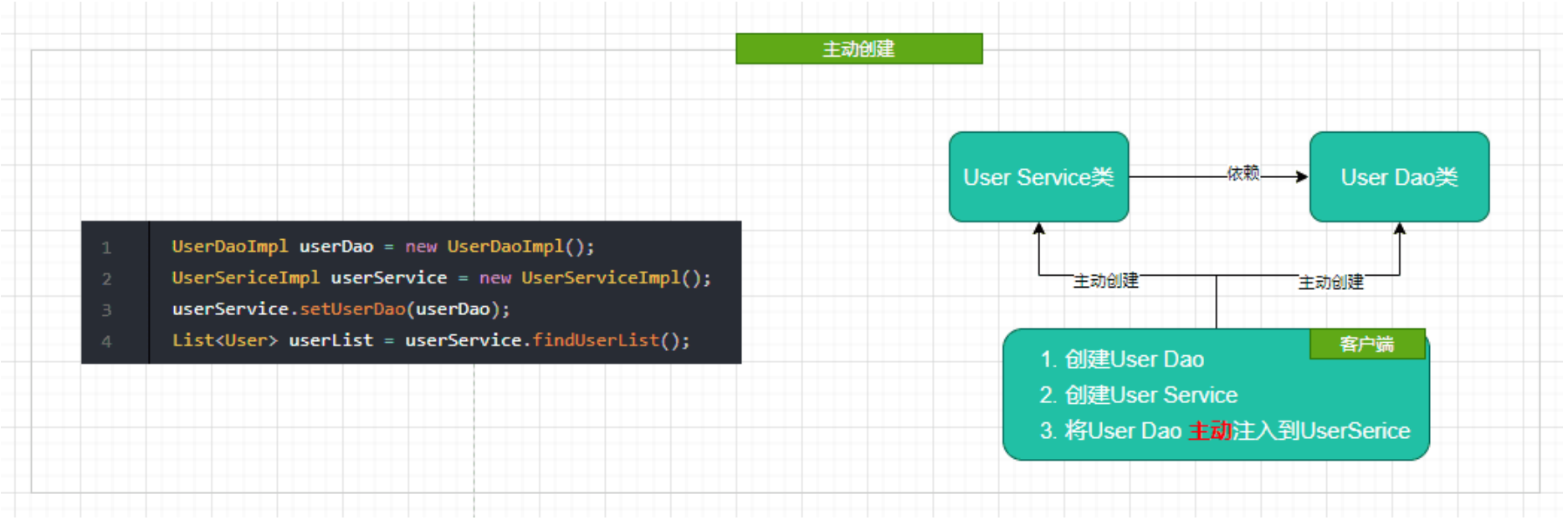
传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对 象的创建；谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

- **为何是反转，哪些方面反转了**?

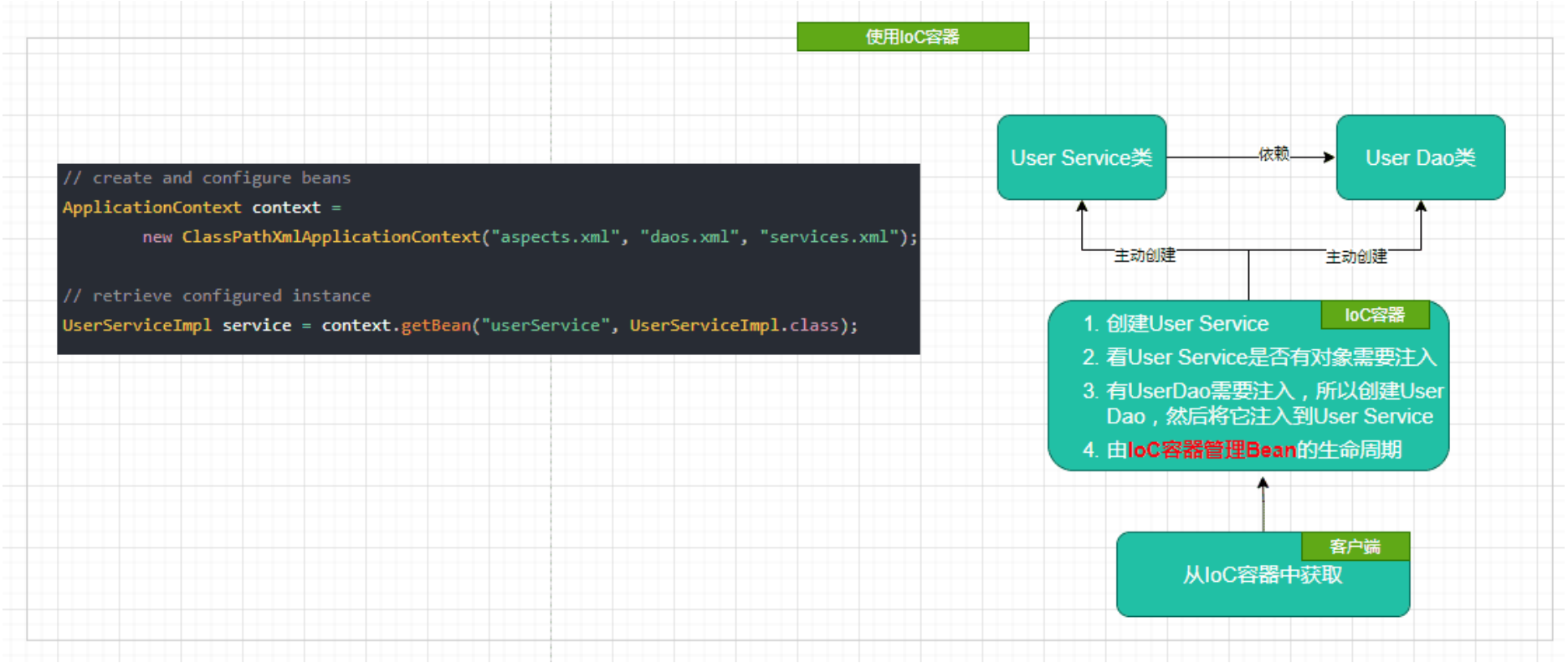
有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

- **用图例说明一下**?

传统程序设计下，都是主动去创建相关对象然后再组合起来：



当有了IoC/DI的容器后，在客户端类中不再主动去创建这些对象了，如图



IoC能做什么

> IoC **不是一种技术，只是一种思想**，一个重要的面向对象编程的法则，它能指导我们如何设计出松耦合、更优良的程序。

传统应用程序都是由我们在类内部主动创建依赖对象，从而导致类与类之间高耦合，难于测试；有了IoC容器后，**把创建和查找依赖对象的控制权交给了容器，由容器进行注入组合对象，所以对象与对象之间是 松散耦合，这样也方便测试，利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。**

其实IoC对编程带来的最大改变不是从代码上，而是从思想上，发生了"主从换位"的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC容器来创建并注入它所需要的资源了。

IoC很好的体现了面向对象设计法则之一—— **好莱坞法则：“别找我们，我们找你”**；即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

IoC和DI是什么关系

> 控制反转是通过依赖注入实现的，其实它们是同一个概念的不同角度描述。通俗来说就是**IoC是设计思想，DI是实现方式。**

DI—Dependency Injection，即依赖注入：组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

我们来深入分析一下：

- **谁依赖于谁**?

当然是应用程序依赖于IoC容器；

- **为什么需要依赖**?

应用程序需要IoC容器来提供对象需要的外部资源；

- **谁注入谁**?

很明显是IoC容器注入应用程序某个对象，应用程序依赖的对象；

- **注入了什么**?

就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

- **IoC和DI有什么关系呢**?

其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字：“依赖注入”，相对IoC 而言，“依赖注入”明确描述了“被注入对象依赖IoC容器配置依赖对象”。通俗来说就是**IoC是设计思想，DI是实现方式。**

Ioc 配置的三种方式

> 在Spring基础 - Spring简单例子引入Spring的核心已经给出了三种配置方式，这里再总结下；总体上目前的主流方式是 **注解 + Java 配置。**

xml 配置

顾名思义，就是将bean的信息配置.xml文件里，通过Spring加载文件为我们创建bean。这种方式出现很多早前的SSM项目中，将第三方类库或者一些配置工具类都以这种方式进行配置，主要原因是由于第三方类不支持Spring注解。

- **优点**： 可以使用于任何场景，结构清晰，通俗易懂
- **缺点**： 配置繁琐，不易维护，枯燥无味，扩展性差

举例：

1. 配置xx.xml文件
2. 声明命名空间和配置bean

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- services -->
```

```
    <bean id="userService" class="tech.pdai.springframework.service.UserServiceImpl">
        <property name="userDao" ref="userDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for services go here -->
</beans>
```

Java 配置

将类的创建交给我们配置的JavcConfig类来完成，Spring只负责维护和管理，采用纯Java创建方式。其本质上就是把在XML上的配置声明转移到Java配置类中

- **优点**：适用于任何场景，配置方便，因为是纯Java代码，扩展性高，十分灵活
- **缺点**：由于是采用Java类的方式，声明不明显，如果大量配置，可读性比较差

举例：

1. 创建一个配置类， 添加@Configuration注解声明为配置类
2. 创建方法，方法上加上@bean，该方法用于创建实例并返回，该实例创建后会交给spring管理，方法名建议与实例名相同（首字母小写）。注：实例类不需要加任何注解

```java
/**
 * @author pdai
 */
@Configuration
public class BeansConfig {

    /**
     * @return user dao
     */
    @Bean("userDao")
```

```
    public UserDaoImpl userDao() {
        return new UserDaoImpl();
    }

    /**
     * @return user service
     */
    @Bean("userService")
    public UserServiceImpl userService() {
        UserServiceImpl userService = new UserServiceImpl();
        userService.setUserDao(userDao());
        return userService;
    }
}
```

注解配置

通过在类上加注解的方式，来声明一个类交给Spring管理，Spring会自动扫描带有@Component，@Controller，@Service，@Repository这四个注解的类，然后帮我们创建并管理，前提是需要先配置Spring的注解扫描器。

- **优点**：开发便捷，通俗易懂，方便维护。
- **缺点**：具有局限性，对于一些第三方资源，无法添加注解。只能采用XML或JavaConfig的方式配置

举例：

1. 对类添加@Component相关的注解，比如@Controller，@Service，@Repository
2. 设置ComponentScan的basePackage, 比如 `<context:component-scan base-package='tech.pdai.springframework'>` , 或者 `@ComponentScan("tech.pdai.springframework")` 注解，或者 `new AnnotationConfigApplicationContext("tech.pdai.springframework")` 指定扫描的basePackage.

java

```
/**
 * @author pdai
```

```
 */
@Service
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    @Autowired
    private UserDaoImpl userDao;

    /**
     * find user list.
     *
     * @return user list
     */
    public List<User> findUserList() {
        return userDao.findUserList();
    }

}
```

依赖注入的三种方式

> 常用的注入方式主要有三种：构造方法注入（Construct注入），setter注入，基于注解的注入（接口注入）

setter方式

- **在XML配置方式中**，property都是setter方式注入，比如下面的xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- services -->
    <bean id="userService" class="tech.pdai.springframework.service.UserServiceImpl">
        <property name="userDao" ref="userDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for services go here -->
</beans>
```

本质上包含两步：

1. 第一步，需要new UserServiceImpl()创建对象, 所以需要默认构造函数
2. 第二步，调用setUserDao()函数注入userDao的值, 所以需要setUserDao()函数

所以对应的service类是这样的：

```java
/**
 * @author pdai
 */
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    private UserDaoImpl userDao;

    /**
     * init.
```

```
     */
    public UserServiceImpl() {
    }

    /**
     * find user list.
     *
     * @return user list
     */
    public List<User> findUserList() {
        return this.userDao.findUserList();
    }

    /**
     * set dao.
     *
     * @param userDao user dao
     */
    public void setUserDao(UserDaoImpl userDao) {
        this.userDao = userDao;
    }
}
```

- **在注解和Java配置方式下**

```java
/**
 * @author pdai
 */
public class UserServiceImpl {

    /**
     * user dao impl.
     */
```

```
    private UserDaoImpl userDao;

    /**
     * find user list.
     *
     * @return user list
     */
    public List<User> findUserList() {
        return this.userDao.findUserList();
    }

    /**
     * set dao.
     *
     * @param userDao user dao
     */
    @Autowired
    public void setUserDao(UserDaoImpl userDao) {
        this.userDao = userDao;
    }
}
```

在Spring3.x刚推出的时候，推荐使用注入的就是这种, 但是这种方式比较麻烦，所以在Spring4.x版本中推荐构造函数注入。

构造函数

- **在XML配置方式中**，`<constructor-arg>`是通过构造函数参数注入，比如下面的xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
 http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- services -->
    <bean id="userService" class="tech.pdai.springframework.service.UserServiceImpl">
        <constructor-arg name="userDao" ref="userDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for services go here -->
</beans>
```

本质上是new UserServiceImpl(userDao)创建对象, 所以对应的service类是这样的:

```java
/**
 * @author pdai
 */
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    private final UserDaoImpl userDao;

    /**
     * init.
     * @param userDaoImpl user dao impl
     */
    public UserServiceImpl(UserDaoImpl userDaoImpl) {
        this.userDao = userDaoImpl;
    }

    /**
     * find user list.
     *

```

```
     * @return user list
     */
    public List<User> findUserList() {
        return this.userDao.findUserList();
    }

}
```

- **在注解和Java配置方式下**

```java
/**
 * @author pdai
 */
 @Service
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    private final UserDaoImpl userDao;

    /**
     * init.
     * @param userDaoImpl user dao impl
     */
    @Autowired // 这里@Autowired也可以省略
    public UserServiceImpl(final UserDaoImpl userDaoImpl) {
        this.userDao = userDaoImpl;
    }

    /**
     * find user list.
     *
```

```
     * @return user list
     */
    public List<User> findUserList() {
        return this.userDao.findUserList();
    }

}
```

在Spring4.x版本中推荐的注入方式就是这种， 具体原因看后续章节。

注解注入

以@Autowired（自动注入）注解注入为例，修饰符有三个属性：Constructor，byType，byName。默认按照byType注入。

- **constructor**：通过构造方法进行自动注入，spring会匹配与构造方法参数类型一致的bean进行注入，如果有一个多参数的构造方法，一个只有一个参数的构造方法，在容器中查找到多个匹配多参数构造方法的bean，那么spring会优先将bean注入到多参数的构造方法中。
- **byName**：被注入bean的id名必须与set方法后半截匹配，并且id名称的第一个单词首字母必须小写，这一点与手动set注入有点不同。
- **byType**：查找所有的set方法，将符合符合参数类型的bean注入。

比如：

```java
/**
 * @author pdai
 */
@Service
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    @Autowired
    private UserDaoImpl userDao;
```

```
12
13      /**
14       * find user list.
15       *
16       * @return user list
17       */
18      public List<User> findUserList() {
19          return userDao.findUserList();
20      }
21
22  }
```

IoC和DI使用问题小结

> 这里总结下实际开发中会遇到的一些问题：

为什么推荐构造器注入方式?

先来看看Spring在文档里怎么说：

> The Spring team generally advocates constructor injection as it enables one to implement application components as immutable objects and to ensure that required dependencies are not null. Furthermore constructor-injected components are always returned to client (calling) code in a fully initialized state.

简单的翻译一下：这个构造器注入的方式**能够保证注入的组件不可变，并且确保需要的依赖不为空。**此外，构造器注入的依赖总是能够在返回客户端（组件）代码的时候保证完全初始化的状态。

下面来简单的解释一下：

- **依赖不可变**：其实说的就是final关键字。
- **依赖不为空**（省去了我们对其检查）：当要实例化UserServiceImpl的时候，由于自己实现了有参数的构造函数，所以不会调用默认构造函数，那么就需要Spring容器传入所需要的参数，所以就两种情况：1、有该类型的参数->传入，OK 。2：无该类型的参数->报错。
- **完全初始化的状态**：这个可以跟上面的依赖不为空结合起来，向构造器传参之前，要确保注入的内容不为空，那么肯定要调用依赖组件的构造方法完成实例化。而在Java类加载实例化的过程中，构造方法是最后一步（之前如果有父类先初始化父类，然后自己的成员变量，最后才是构造方法），所以返回来的都是初始化之后的状态。

所以通常是这样的

```java
/**
 * @author pdai
 */
 @Service
public class UserServiceImpl {

    /**
     * user dao impl.
     */
    private final UserDaoImpl userDao;

    /**
     * init.
     * @param userDaoImpl user dao impl
     */
    public UserServiceImpl(final UserDaoImpl userDaoImpl) {
        this.userDao = userDaoImpl;
    }

}
```

如果使用setter注入，缺点显而易见，对于IOC容器以外的环境，除了使用反射来提供它需要的依赖之外，**无法复用该实现类**。而且将一直是个潜在的隐患，因为你不调用将一直无法发现NPE的存在。

```java
// 这里只是模拟一下，正常来说我们只会暴露接口给客户端，不会暴露实现。
UserServiceImpl userService = new UserServiceImpl();
userService.findUserList(); // -> NullPointerException, 潜在的隐患
```

循环依赖的问题：使用field注入可能会导致循环依赖，即A里面注入B，B里面又注入A：

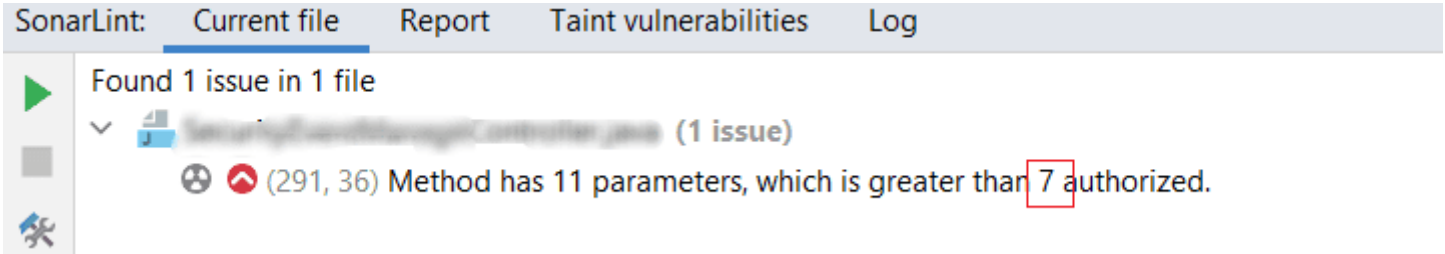
```java
public class A {
    @Autowired
    private B b;
}

public class B {
    @Autowired
    private A a;
}
```

如果使用构造器注入，在spring项目启动的时候，就会抛出：BeanCurrentlyInCreationException：Requested bean is currently in creation: Is there an unresolvable circular reference？从而提醒你避免循环依赖，如果是field注入的话，启动的时候不会报错，在使用那个bean的时候才会报错。

我在使用构造器注入方式时注入了太多的类导致Bad Smell怎么办？

比如当你一个Controller中注入了太多的Service类，Sonar会给你提示相关告警



对于这个问题，说明你的类当中有太多的责任，那么你要好好想一想是不是自己违反了类的单一性职责原则，从而导致有这么多的依赖要注入。

（pdai： 想起来一句话：**所有困难问题的解决方式，都在另外一个层次**）

@Autowired和@Resource以及@Inject等注解注入有何区别?

> @Autowired和@Resource以及@Inject等注解注入有何区别? 这时平时在开发中，或者常见的面试题。

@Autowired

- **Autowired注解源码**

在Spring 2.5 引入了 @Autowired 注解

```java
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD, ElementType.PARAMETER, ElementType.FIELD, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Autowired {
  boolean required() default true;
}
```

从Autowired注解源码上看，可以使用在下面这些地方：

```java
@Target(ElementType.CONSTRUCTOR) #构造函数
@Target(ElementType.METHOD) #方法
@Target(ElementType.PARAMETER) #方法参数


```

```
@Target(ElementType.FIELD) #字段、枚举的常量
@Target(ElementType.ANNOTATION_TYPE) #注解
```

还有一个value属性，默认是true。

- **简单总结**:

1、@Autowired是Spring自带的注解，通过AutowiredAnnotationBeanPostProcessor 类实现的依赖注入

2、@Autowired可以作用在CONSTRUCTOR、METHOD、PARAMETER、FIELD、ANNOTATION_TYPE

3、@Autowired默认是根据类型（byType ）进行自动装配的

4、如果有多个类型一样的Bean候选者，需要指定按照名称（byName ）进行装配，则需要配合@Qualifier。

指定名称后，如果Spring IOC容器中没有对应的组件bean抛出NoSuchBeanDefinitionException。也可以将@Autowired中required配置为false，如果配置为false之后，当没有找到相应bean的时候，系统不会抛异常

- **简单使用代码**:

在字段属性上。

```java
@Autowired
private HelloDao helloDao;
```

或者

```java
private HelloDao helloDao;
public HelloDao getHelloDao() {
 return helloDao;
}
@Autowired
```

```
6    public void setHelloDao(HelloDao helloDao) {
7     this.helloDao = helloDao;
8    }
```

或者

```java
1    private HelloDao helloDao;
2    //@Autowired
3    public HelloServiceImpl(@Autowired HelloDao helloDao) {
4     this.helloDao = helloDao;
5    }
6    // 构造器注入也可不写@Autowired，也可以注入成功。
```

将@Autowired写在被注入的成员变量上，setter或者构造器上，就不用再xml文件中配置了。

如果有多个类型一样的Bean候选者，则默认根据设定的属性名称进行获取。如 HelloDao 在Spring中有 helloWorldDao 和 helloDao 两个Bean候选者。

```java
1    @Autowired
2    private HelloDao helloDao;
```

首先根据类型获取，发现多个HelloDao，然后根据helloDao进行获取，如果要获取限定的其中一个候选者，结合@Qualifier进行注入。

```java
1    @Autowired
2    @Qualifier("helloWorldDao")
3    private HelloDao helloDao;
```

注入名称为helloWorldDao 的Bean组件。@Qualifier("XXX") 中的 XX是 Bean 的名称，所以 @Autowired 和 @Qualifier 结合使用时，自动注入的策略就从 byType 转变成 byName 了。

多个类型一样的Bean候选者，也可以@Primary进行使用，设置首选的组件，也就是默认优先使用哪一个。

注意：使用@Qualifier 时候，如何设置的指定名称的Bean不存在，则会抛出异常，如果防止抛出异常，可以使用：

```java
@Qualifier("xxxxyyyy")
@Autowired(required = false)
private HelloDao helloDao;
```

在SpringBoot中也可以使用@Bean+@Autowired进行组件注入，将@Autowired加到参数上，其实也可以省略。

```java
@Bean
public Person getPerson(@Autowired Car car){
 return new Person();
}
// @Autowired 其实也可以省略
```

@Resource

- **Resource注解源码**

```java
@Target({TYPE, FIELD, METHOD})
@Retention(RUNTIME)
public @interface Resource {
    String name() default "";
    // 其他省略
}
```

从Resource注解源码上看，可以使用在下面这些地方：

```java
@Target(ElementType.TYPE) #接口、类、枚举、注解
@Target(ElementType.FIELD) #字段、枚举的常量
@Target(ElementType.METHOD) #方法
```

name 指定注入指定名称的组件。

- **简单总结**:

1、@Resource是JSR250规范的实现，在javax.annotation包下

2、@Resource可以作用TYPE、FIELD、METHOD上

3、@Resource是默认根据属性名称进行自动装配的，如果有多个类型一样的Bean候选者，则可以通过name进行指定进行注入

- **简单使用代码**:

```java
@Component
public class SuperMan {
    @Resource
    private Car car;
}
```

按照属性名称 car 注入容器中的组件。如果容器中BMW还有BYD两种类型组件。指定加入BMW。如下代码:

```java
@Component
public class SuperMan {
    @Resource(name = "BMW")
    private Car car;
}
```

name 的作用类似 @Qualifier

@Inject

- **Inject注解源码**

```java
@Target({ METHOD, CONSTRUCTOR, FIELD })
@Retention(RUNTIME)
@Documented
public @interface Inject {}
```

从Inject注解源码上看，可以使用在下面这些地方：

```java
@Target(ElementType.CONSTRUCTOR) #构造函数
@Target(ElementType.METHOD) #方法
@Target(ElementType.FIELD) #字段、枚举的常量
```

- **简单总结**：

1、@Inject是JSR330 (Dependency Injection for Java)中的规范，需要导入javax.inject.Inject jar包 ，才能实现注入

2、@Inject可以作用CONSTRUCTOR、METHOD、FIELD上

3、@Inject是根据类型进行自动装配的，如果需要按名称进行装配，则需要配合@Named；

- **简单使用代码**：

```java
@Inject
private Car car;
```

指定加入BMW组件。

```java
@Inject
@Named("BMW")
private Car car;
```

@Named 的作用类似 @Qualifier!

总结

1、@Autowired是Spring自带的，@Resource是JSR250规范实现的，@Inject是JSR330规范实现的

2、@Autowired、@Inject用法基本一样，不同的是@Inject没有required属性

3、@Autowired、@Inject是默认按照类型匹配的，@Resource是按照名称匹配的

4、@Autowired如果需要按照名称匹配需要和@Qualifier一起使用，@Inject和@Named一起使用，@Resource则通过name进行指定

如果你还期望源码层理解，我给你找了一篇文章Spring源码分析@Autowired、@Resource注解的区别

参考文章

Inversion of Control Containers and the Dependency Injection pattern

https://www.iteye.com/blog/jinnianshilongnian-1413846

https://blog.csdn.net/qq_35634181/article/details/104276056

https://www.cnblogs.com/diandianquanquan/p/11518365.html