

19 | 理论五：控制反转、依赖反转、依赖注入，这三者有何区别和联系？

王争 · 设计模式之美



关于 SOLID 原则，我们已经学过单一职责、开闭、里式替换、接口隔离这四个原则。今天，我们再来学习最后一个原则：依赖反转原则。在前面几节课中，我们讲到，单一职责原则和开闭原则的原理比较简单，但是，想要在实践中用好却比较难。而今天我们要讲到的依赖反转原则正好相反。这个原则用起来比较简单，但概念理解起来比较难。比如，下面这几个问题，你看看能否清晰地回答出来：

“依赖反转”这个概念指的是“谁跟谁”的“什么依赖”被反转了？“反转”两个字该如何理解？

我们还经常听到另外两个概念：“控制反转”和“依赖注入”。这两个概念跟“依赖反转”有什么区别和联系呢？它们说的是同一个事情吗？


如果你熟悉 Java 语言，那 Spring 框架中的 IOC 跟这些概念又有什么关系呢？

看了刚刚这些问题，你是不是有点懵？别担心，今天我会带你将这些问题彻底搞个清楚。之后再有人问你，你就能轻松应对。话不多说，现在就让我们带着这些问题，正式开始今天的学习吧！

控制反转（IOC）

在讲“依赖反转原则”之前，我们先讲一讲“控制反转”。控制反转的英文翻译是 Inversion Of Control，缩写为 IOC。此处我要强调一下，如果你是 Java 工程师的话，暂时别把这个“IOC”跟 Spring 框架的 IOC 联系在一起。关于 Spring 的 IOC，我们待会儿还会讲到。

我们先通过一个例子来看一下，什么是控制反转。

 复制代码

```
1 public class UserServiceTest {
2     public static boolean doTest() {
3         // ...
4     }
5
6     public static void main(String[] args) { //这部分逻辑可以放到框架中
7         if (doTest()) {
8             System.out.println("Test succeed.");
9         } else {
10            System.out.println("Test failed.");
11        }
12    }
13 }
```

在上面的代码中，所有的流程都由程序员来控制。如果我们抽象出一个下面这样一个框架，我们再来看，如何利用框架来实现同样的功能。具体的代码实现如下所示：

 复制代码


```
1 public abstract class TestCase {
2     public void run() {
3         if (doTest()) {
4             System.out.println("Test succeed.");
5         } else {
6             System.out.println("Test failed.");
7         }
8     }
9 }
```

```

8    }
9
10   public abstract boolean doTest();
11 }
12
13 public class JunitApplication {
14     private static final List<TestCase> testCases = new ArrayList<>();
15
16     public static void register(TestCase testCase) {
17         testCases.add(testCase);
18     }
19
20     public static final void main(String[] args) {
21         for (TestCase case: testCases) {
22             case.run();
23         }
24     }

```

把这个简化版本的测试框架引入到工程中之后，我们只需要在框架预留的扩展点，也就是 TestCase 类中的 doTest() 抽象函数中，填充具体的测试代码就可以实现之前的功能了，完全不需要写负责执行流程的 main() 函数了。具体的代码如下所示：

 复制代码

```

1 public class UserServiceTest extends TestCase {
2     @Override
3     public boolean doTest() {
4         // ...
5     }
6 }
7
8 // 注册操作还可以通过配置的方式来实现，不需要程序员显示调用register()
9 JunitApplication.register(new UserServiceTest());

```

刚刚举的这个例子，就是典型的通过框架来实现“控制反转”的例子。框架提供了一个可扩展的代码骨架，用来组装对象、管理整个执行流程。程序员利用框架进行开发的时候，只需要往预留的扩展点上，添加跟自己业务相关的代码，就可以利用框架来驱动整个程序流程的执行。

这里的“控制”指的是对程序执行流程的控制，而“反转”指的是在没有使用框架之前，程序员自己控制整个程序的执行。在使用框架之后，整个程序的执行流程可以通过框架来控制。流程的

控制权从程序员“反转”到了框架。


实际上，实现控制反转的方法有很多，除了刚才例子中所示的类似于模板设计模式的方法之外，还有马上要讲到的依赖注入等方法，所以，控制反转并不是一种具体的实现技巧，而是一个比较笼统的设计思想，一般用来指导框架层面的设计。

依赖注入（DI）

接下来，我们再来看依赖注入。依赖注入跟控制反转恰恰相反，它是一种具体的编码技巧。依赖注入的英文翻译是 Dependency Injection，缩写为 DI。对于这个概念，有一个非常形象的说法，那就是：依赖注入是一个标价 25 美元，实际上只值 5 美分的概念。也就是说，这个概念听起来很“高大上”，实际上，理解、应用起来非常简单。

那到底什么是依赖注入呢？我们用一句话来概括就是：不通过 new() 的方式在类内部创建依赖类对象，而是将依赖的类对象在外部创建好之后，通过构造函数、函数参数等方式传递（或注入）给类使用。

我们还是通过一个例子来解释一下。在这个例子中，Notification 类负责消息推送，依赖 MessageSender 类实现推送商品促销、验证码等消息给用户。我们分别用依赖注入和非依赖注入两种方式来实现一下。具体的实现代码如下所示：

 复制代码


```
1 // 非依赖注入实现方式
2 public class Notification {
3     private MessageSender messageSender;
4
5     public Notification() {
6         this.messageSender = new MessageSender(); //此处有点像hardcode
7     }
8
9     public void sendMessage(String cellphone, String message) {
10         //...省略校验逻辑等...
11         this.messageSender.send(cellphone, message);
12     }
13 }
14
15 public class MessageSender {
16     public void send(String cellphone, String message) {
```

```

17     //....
18 }
19 }
20 // 使用Notification
21 Notification notification = new Notification();
22
23 // 依赖注入的实现方式
24 public class Notification {
25     private MessageSender messageSender;
26
27     // 通过构造函数将messageSender传递进来
28     public Notification(MessageSender messageSender) {
29         this.messageSender = messageSender;
30     }
31
32     public void sendMessage(String cellphone, String message) {
33         //...省略校验逻辑等...
34         this.messageSender.send(cellphone, message);
35     }
36 }
37 //使用Notification
38 MessageSender messageSender = new MessageSender();
39 Notification notification = new Notification(messageSender);

```

通过依赖注入的方式来将依赖的类对象传递进来，这样就提高了代码的扩展性，我们可以灵活地替换依赖的类。这一点在我们之前讲“开闭原则”的时候也提到过。当然，上面代码还有继续优化的空间，我们还可以把 MessageSender 定义成接口，基于接口而非实现编程。改造后的代码如下所示：

 复制代码

```

1 public class Notification {
2     private MessageSender messageSender;
3
4     public Notification(MessageSender messageSender) {
5         this.messageSender = messageSender;
6     }
7
8     public void sendMessage(String cellphone, String message) {
9         this.messageSender.send(cellphone, message);
10    }
11 }
12
13 public interface MessageSender {
14     void send(String cellphone, String message);

```



```

15 }
16
17 // 短信发送类
18 public class SmsSender implements MessageSender {
19     @Override
20     public void send(String cellphone, String message) {
21         //....
22     }
23 }
24
25 // 站内信发送类
26 public class InboxSender implements MessageSender {
27     @Override
28     public void send(String cellphone, String message) {
29         //....
30     }
31 }
32
33 //使用Notification
34 MessageSender messageSender = new SmsSender();
35 Notification notification = new Notification(messageSender);


```

实际上，你只需要掌握刚刚举的这个例子，就等于完全掌握了依赖注入。尽管依赖注入非常简单，但却非常有用，在后面的章节中，我们会讲到，它是编写可测试性代码最有效的手段。

依赖注入框架（DI Framework）

弄懂了什么是“依赖注入”，我们再来看一下，什么是“依赖注入框架”。我们还是借用刚刚的例子来解释。

在采用依赖注入实现的 Notification 类中，虽然我们不需要用类似 hard code 的方式，在类内部通过 new 来创建 MessageSender 对象，但是，这个创建对象、组装（或注入）对象的工作仅仅是被移动到了更上层代码而已，还是需要我们程序员自己来实现。具体代码如下所示：

 复制代码

```

1 public class Demo {
2     public static final void main(String args[]) {
3         MessageSender sender = new SmsSender(); //创建对象
4         Notification notification = new Notification(sender); //依赖注入

```

```
5     notification.sendMessage("13918942177", "短信验证码: 2346");
6 }
7 }
```

在实际的软件开发中，一些项目可能会涉及几十、上百、甚至几百个类，类对象的创建和依赖注入会变得非常复杂。如果这部分工作都是靠程序员自己写代码来完成，容易出错且开发成本也比较高。而对象创建和依赖注入的工作，本身跟具体的业务无关，我们完全可以抽象成框架来自动完成。

你可能已经猜到，这个框架就是“依赖注入框架”。我们只需要通过依赖注入框架提供的扩展点，简单配置一下所有需要创建的类对象、类与类之间的依赖关系，就可以实现由框架来自动创建对象、管理对象的生命周期、依赖注入等原本需要程序员来做的事情。

实际上，现成的依赖注入框架有很多，比如 Google Guice、Java Spring、Pico Container、Butterfly Container 等。不过，如果你熟悉 Java Spring 框架，你可能会说，Spring 框架自己声称是**控制反转容器**（Inversion Of Control Container）。

实际上，这两种说法都没错。只是控制反转容器这种表述是一种非常宽泛的描述，DI 依赖注入框架的表述更具体、更有针对性。因为我们前面讲到实现控制反转的方式有很多，除了依赖注入，还有模板模式等，而 Spring 框架的控制反转主要是通过依赖注入来实现的。不过这点区分并不是很明显，也不是很重要，你稍微了解一下就可以了。

依赖反转原则（DIP）

前面讲了控制反转、依赖注入、依赖注入框架，现在，我们来讲一讲今天的主角：依赖反转原则。依赖反转原则的英文翻译是 Dependency Inversion Principle，缩写为 DIP。中文翻译有时候也叫依赖倒置原则。

为了追本溯源，我先给出这条原则最原汁原味的英文描述：

High-level modules shouldn't depend on low-level modules. Both modules should depend on abstractions. In addition, abstractions shouldn't depend on details. Details depend on abstractions.

我们将它翻译成中文，大概意思就是：高层模块（high-level modules）不要依赖低层模块（low-level）。高层模块和低层模块应该通过抽象（abstractions）来互相依赖。除此之外，抽象（abstractions）不要依赖具体实现细节（details），具体实现细节（details）依赖抽象（abstractions）。

所谓高层模块和低层模块的划分，简单来说就是，在调用链上，调用者属于高层，被调用者属于低层。在平时的业务代码开发中，高层模块依赖底层模块是没有任何问题的。实际上，这条原则主要还是用来指导框架层面的设计，跟前面讲到的控制反转类似。我们拿 Tomcat 这个 Servlet 容器作为例子来解释一下。

Tomcat 是运行 Java Web 应用程序的容器。我们编写的 Web 应用程序代码只需要部署在 Tomcat 容器下，便可以由 Tomcat 容器调用执行。按照之前的划分原则，Tomcat 就是高层模块，我们编写的 Web 应用程序代码就是低层模块。Tomcat 和应用程序代码之间并没有直接的依赖关系，两者都依赖同一个“抽象”，也就是 Servlet 规范。Servlet 规范不依赖具体的 Tomcat 容器和应用程序的实现细节，而 Tomcat 容器和应用程序依赖 Servlet 规范。

重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要掌握的重点内容。

1. 控制反转

实际上，控制反转是一个比较笼统的设计思想，并不是一种具体的实现方法，一般用来指导框架层面的设计。这里所说的“控制”指的是对程序执行流程的控制，而“反转”指的是在没有使用框架之前，程序员自己控制整个程序的执行。在使用框架之后，整个程序的执行流程通过框架来控制。流程的控制权从程序员“反转”给了框架。

2. 依赖注入

依赖注入和控制反转恰恰相反，它是一种具体的编码技巧。我们不通过 new 的方式在类内部创建依赖类的对象，而是将依赖的类对象在外部创建好之后，通过构造函数、函数参数等方式传递（或注入）给类来使用。

3. 依赖注入框架

我们通过依赖注入框架提供的扩展点，简单配置一下所有需要的类及其类与类之间依赖关系，就可以实现由框架来自动创建对象、管理对象的生命周期、依赖注入等原本需要程序员来做的事情。

4. 依赖反转原则

依赖反转原则也叫作依赖倒置原则。这条原则跟控制反转有点类似，主要用来指导框架层面的设计。高层模块不依赖低层模块，它们共同依赖同一个抽象。抽象不要依赖具体实现细节，具体实现细节依赖抽象。

课堂讨论

从 Notification 这个例子来看，“基于接口而非实现编程”跟“依赖注入”，看起来非常类似，那它俩有什么区别和联系呢？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

控制反转（IOC）和依赖注入（DI）是软件开发中重要的概念。控制反转指的是将程序执行流程的控制权从程序员转移到框架，而依赖注入则是一种具体的编码技巧，通过构造函数、函数参数等方式将依赖的类对象传递给类使用，提高了代码的扩展性和灵活性。本文通过例子详细解释了控制反转和依赖注入的概念和实现方式，帮助读者理解这两个概念的区别和联系。同时，还提到了Spring框架中的IOC和依赖注入的关系，为读者提供了更多实际应用的参考。通过本文的学习，读者可以更好地理解和应用控制反转和依赖注入，为编写可测试性代码提供有效的手段。文章还介绍了依赖注入框架和依赖反转原则，为读者提供了更全面的知识体系。整体而言，本文内容丰富，涵盖了控制反转、依赖注入、依赖注入框架和依赖反转原则，对于想深入了解软件开发中的这些重要概念的读者来说，是一篇值得阅读的文章。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (222)

最新 精选



thomas

2019-12-16

DIP原则有点嘎然而止的感觉，缺少了一个具体的例子。

作者回复: tomcat的例子不算啊

共 3 条评论 >

👍 27



JRich

2020-11-22

控制反转定义里是程序执行流程的控制权反转，而依赖注入讲的是对象的创建由外部创建好通过构造方法或setter方法注入进来，感觉两个讲的不是一个意思，虽然依赖注入也有控制反转的意思，但是对象的创建和获取的权利被反转，更确切的理解应该是2个场景吧。

作者回复: 🤔🤔🤔🤔🤔



👍 1



港岛妹夫

2020-06-13

想知道争哥的英文原文都是从哪里读来的. 如果是书的话, 可以推荐一些嘛~

作者回复: 都是实践中来的，自己动脑子思考的比较多，书的话也就是看看而已，你可以看我写的这篇文章：

<https://mp.weixin.qq.com/s/uKkQMIWTtAmvsEYZCxvZeg>



👍 1



青子

2020-01-04

`JunitApplication.register(new UserServiceTest());`
在控制反转中执行这句话会执行该类中的final修饰的main方法吗

作者回复: 代码稍微有点问题，我改下，抱歉！

共 3 条评论 >

👍 1



|·ω·`)

2019-12-18

关于最后一个依赖反转能再举个简单的代码例子吗？Tomcat的案例没懂`(`

作者回复: 你网上搜下 开关的例子 那个更简单



堵车

2019-12-17

老师，今天怎么没更新，我已经迫不及待了。隔壁那本DDD好多词汇难理解，我受了打击，过来找安慰。

作者回复: 周一三五更新的



沉淀的梦想

2019-12-16

SOLID 的最后一个原则D，我看好多书上说是 迪米特法则 啊，为什么文章里没有提呢？

作者回复: d是dip 不是lod的 lod后面有讲到



JRich

2020-11-20

基于接口而非实现编程使用了依赖注入编程技巧。因为基于接口而非实现编程使用了面向对象的多态特性来提高代码扩展性，必然不可能在类内部创建对象，只能从外部注入。区别就是依赖注入不仅可以使使用接口，还可以使用类。

作者回复: ？？？？？



开心小毛

2020-01-05

底层模块依赖上层模块的抽象是否提倡？

作者回复: 上层模块的抽象你是指什么呢?

共 3 条评论 >



程晓擘

2020-01-05

高层模块不依赖低层模块，它们共同依赖同一个抽象。抽象不要依赖具体实现细节，具体实现细节依赖抽象。 不太明白，为什么叫依赖倒置呀？倒置啥呢？ 我可能会取名，依赖抽象原则，哈哈。

作者回复: 倒置就是反转，我在文章中不是解释了为啥叫反转吗😂

