

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？

2019-12-16 王争

设计模式之美

[进入课程 >](#)



讲述：冯永吉

时长 10:39 大小 9.76M



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
10 public abstract void 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     }
25 }

```

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，便可以被 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。

点击参加小程序学习打卡 

8个月，攻克设计模式



扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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。

上一篇 18 | 理论四：接口隔离原则有哪三种应用？原则中的“接口”该如何理解？

下一篇 20 | 理论六：我为何说KISS、YAGNI原则看似简单，却经常被用错？

精选留言 (67)

写留言



小晏子

2019-12-16

课后思考：

“基于接口而非实现编程”与“依赖注入”的联系是二者都是从外部传入依赖对象而不是在内部去new一个出来。

区别是“基于接口而非实现编程”强调的是“接口”，强调依赖的对象是接口，而不是具体的实现类；而“依赖注入”不强调这个，类或接口都可以，只要是从外部传入不是在...

展开 ▾



32



下雨天

2019-12-16

区别：

1.依赖注入是一种具体编程技巧，关注的是对象创建和类之间关系，目的提高了代码的扩展性，我们可以灵活地替换依赖的类。

2.基于接口而非实现编程是一种设计原则，关注抽象和实现，上下游调用稳定性，目的是降低耦合性，提高扩展性。...

展开 ▾



16



辣么大

2019-12-16

①控制反转是一种编程思想，把控制权交给第三方。依赖注入是实现控制反转最典型的方法。

②依赖注入（对象）的方式要采用“基于接口而非实现编程”的原则，说白了就是依赖倒转。

③低层的实现要符合里氏替换原则。子类的可替换性，使得父类模块或依赖于抽象的高层...

展开 ▾



11



业余爱好者

2019-12-16

原来的模式是一个spring开发的项目放在Tomcat中，控制权在Tomcat手中。现在微服务兴起，大家都用springboot开发。此时是Tomcat在springboot项目当中。控制权在springboot手中，虽然只是表面上。这便是控制反转。

这是一场控制权争夺之战。

展开 ∨

💬 3

👍 9



KIM

2019-12-16

感觉比head first设计模式讲的清晰

展开 ∨

💬

👍 5



Ken张云忠

2019-12-16

区别:

基于接口而非实现编程:是面向对象编程的一种方式.减少对外部的依赖,还可以提升代码的灵活性,扩展及修改时可以控制风险的传播,符合开闭原则.

依赖注入:是一种具体的编码技巧,属于编程规范的范畴.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...

展开 ∨

💬

👍 4



MindController

2019-12-16

深夜打卡

展开 ∨

💬 1

👍 4



帆大肚子

2019-12-16

在我看来,“依赖注入”是“基于接口而非实现编程”的一个实践。

“基于接口而非实现编程”是一条设计原则,可以帮助我们诞生更多类似于“依赖注入”的实践

💬

👍 2



iLeGeND

2019-12-16

有收获

展开 ∨

💬

👍 2



javaadu

2019-12-17

课堂讨论：这两个概念没什么关系，讲的不是一个事。依赖注入讲的是一个对象如何获得它运行所依赖的对象，所谓依赖注入就是不需要自己去new，让框架注入进来；基于接口而不是实现编程讲的是抽象思维的应用，利用编程，可以屏蔽掉底层具体实现改变导致上层改变的问题。

...

展开 ▾



1



阿顺

2019-12-17

区别：依赖注入是不是使用的是多态的特性，基于接口而非实现编程使用了抽象的性，对吗



1



阿冰777

2019-12-16

基于接口而非实现编程（依赖倒置原则）：高层和低层组件都使用了一样的接口，然后让接口去控制整个逻辑，这样高层组件就不会依赖于具体的低层组件实现。简单来讲，就是大家都用接口，彼此不认识。

依赖注入：依赖注入就是一个组件内部依赖一个对象，但是他不自己造，等别人送上来。他们俩的关系就是，在依赖倒置原则指导下的设计里，组件都没有内部创造依赖的对象...

展开 ▾



1



再见孙悟空

2019-12-16

“基于接口而非实现编程” 和 “依赖注入”

联系：

都能实现注入功能，程序依赖的对象都能在外部分先创建而无需程序内部显示 new 。

...

展开 ▾



1



空知

2019-12-16

loc样例代码那里,抽象类TestCase 的doTest方法 应该返回布尔值,而不是void



1



睡觉



2019-12-16

控制反转：控制指的是程序流程的控制，反转是指程序的流程的控制权由程序员转移到框架

依赖注入：上层类依赖底层类执行业务，以前往往将底层类作为上层类的成员变量，在上层类的内部声明底层类。注入就是底层类在外边声明，通过接口的方式注入到上层类中

依赖反转原则：我的理解是模块的解耦。上层模块依赖于低等模块，通过抽象出一套规...

展开 ▾



1



李小四

2019-12-16

设计模式_19

作业

“基于接口而非实现编程”：是一种设计原则。

“依赖注入”：一种对上面原则的应用。

...

展开 ▾



1



Smallfly

2019-12-16

依赖倒置原则概念是高层次模块不依赖于低层次模块。看似在要求高层次模块，实际上是在规范低层次模块的设计。

低层次模块提供的接口要足够的抽象、通用，在设计时需要考虑高层次模块的使用种类和场景。...

展开 ▾



1



墨雨

2019-12-16

感觉依赖反转原则是不是可以叫“依赖抽象原则”？😏，从字面意思来看我觉得可以翻译成：高层模块和低层模块及实现细节都应依赖于抽象。

展开 ▾



1



MarksGui

2019-12-16

这个专栏确实讲解的非常细致！争哥确实是用心做专栏！以前对很多类似的概念都没理解透彻，通过这个专栏完全明白了！

展开 ▾



👍 1



秋天

2019-12-16

区别就是依赖注入属于框架层面，接口编程属于实现层面



👍 1