控制反转

# 介绍

学习Spring的伙伴都知道控制反转（Inversion of Control）和依赖注入（Dependency Injection），而且这两个概念也是Spring最基础的东西，贯穿了整个Spring框架。但是，初学Spring的人大都会被这两个概念搞混，不知道其异同（我在前一篇文章也把两者认为是同一个东西）。所以，本文将讨论控制反转和依赖注入的概念以及他们的关系。首先，把两者的关系总结出来，因为我们需要带着这个概念去看下面的文章。带着问题去学习效率更高，学习效果更好。

关系：控制反转是更抽象的原则，而依赖注入是更具体的实现。依赖注入只是用以实现控制反转的一种实现方式而已。

# 依赖（Dependency）

既然依赖注入只是控制反转的一种具体的实现，所以我们前面的讨论都只讨论控制反转，最后讨论依赖注入。要理解控制反转，首先我们需要理解什么是依赖。如图1，如果Class A使用了Service A的某个方法或整个Service A，那么说明Service A就是Class A的一个依赖。比如我们在开发Web应用时，在控制器（Controller）中使用某个服务（Service），那么这个服务就是这个控制器的依赖。

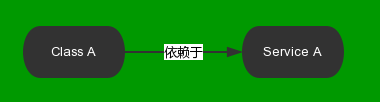


图1 Class A依赖于Service A

# 控制反转

使用来自维基百科的定义：

Inversion of Control is an abstract principal describing an aspect of some software architecture design in which the flow of control of a system is inverted in comparison to procedural programming.

也就是说控制反转其实是把系统中的流程反转了的一种抽象的原则，说了和没说一样。下面就通过例子来解释这个概念。注意：这些都只是代码片段，并不完整。

## 传统编程方式

|  |
| --- |
| // Example.java  public class Example {  private ConsoleLogger logger;  public void doStuff() {  // 对ConsoleLogger的依赖  logger = new ConsoleLogger();  logger.log();  }  }  // ConsoleLogger.java  public class ConsoleLogger {  public void log() {};  }  // TestMain.java 测试Example类  public class TestMain {  public static void main(String[] args){  Example example = new Example();  example.doStuff();  }  } |

我们来分析这种方式下会有那些问题：每一个Example类都拥有一个ConsoleLogger的备份，如果ConsoleLogger的新建非常昂贵的话（比如包含数据库打开，关闭连接操作），那么这是非常影响效率的。因为在Example类中包含ConsoleLogger的实例化操作（使用new操作符），所以如果我们更改了ConsoleLogger的实现方式，那么我们必须去每个Example类中修改源代码。如果我们现在需要修改Example类，不使用ConsoleLogger了，而是使用FileLogger，这样问题就变得复杂，需要去修改涉及的Example源码。

这会让Example类变得难以测试。因为Example类依赖于具体的ConsoleLogger类，这样你就不能仿制一个假的ConsoleLogger类去测试。而且，如果你的ConsoleLogger类依赖于具体的环境的话，那么你的JUnit测试可能在某个环境中可以通过，但是会在其他的环境中失败。

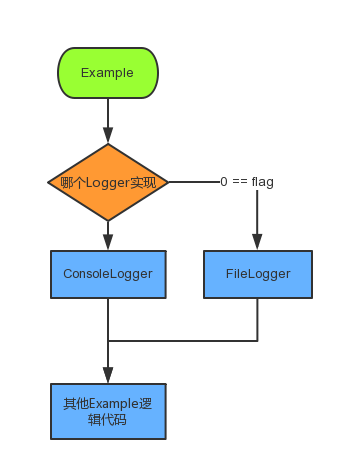
现在，我们就来想办法如何解决这些问题。首先，我们想既然会使用不同的依赖实现方式-ConsoleLogger、FileLogger或者其他方式。所以，我们不应该在Example类中依赖具体的实现类，而应该依赖于抽象。所以，我们建一个抽象的Logger接口：

|  |
| --- |
| // Logger.java  public interface Logger {  void log();  }  // ConsoleLogger和FileLogger类分别继承Logger接口，此处省略  // 修改后的Example.java类  public class Example {  private Logger logger;  public void doStuff() {  // 对ConsoleLogger的依赖  logger = new ConsoleLogger();  logger.log();  }  } |

如此修改之后，如果我们需要修改使用不同的Logger实现，只需要把上述Example类中的new ConsoleLogger()换成其他Logger实现就行了。但是，这样也不能满足不修改源码实现使用不同Logger实现的需求。既然需要不同的实现，我们能不能使用一个标识符来进行判断，从而达到选择不同Logger实现的目的。

|  |
| --- |
| // Example.java 添加标识符判断使用不同Logger实现  public class Example {  private Logger logger;  public Example(int flag) {  if (0 == flag) {  logger = new FileLogger();  } else {  logger = new ConsoleLogger();  }  }  public void doStuff() {  // 其它的Example代码逻辑  ...  logger.log();  ...  }  }  // TestMain.java 还需要修改使用Example类的地方  public class TestMain {  public static void main(String[] args){  // 使用FileLogger  Example example = new Example(0);  example.doStuff();  // 使用默认的ConsoleLogger  Example example = new Example();  example.doStuff();  }  } |

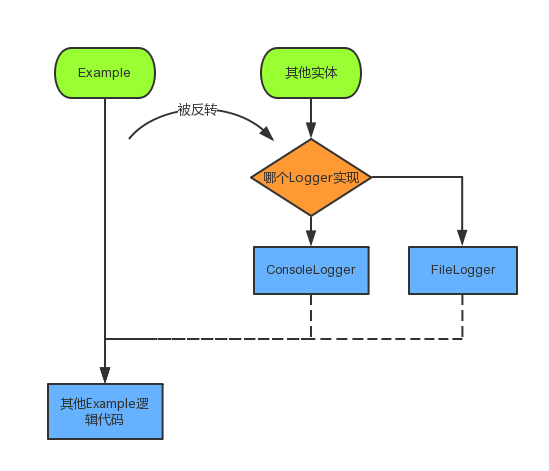
通过Flag控制依赖实现，这样修改以后，是不是觉得好多了。通过传入不同的参数就可以选择不同的Logger实现。但是，这样还是没有解决上面提到的那些问题。Example自己决定依赖的实现方式，自己实例化依赖。这样的结果就是Client(Example类）和Service（Logger的具体实现）之间的耦合，也就导致了以上我们列出的问题。所以，如何解决这个问题呢，方案就是控制反转。



## 控制反转

下面我们使用控制反转来解决这些问题

|  |
| --- |
| // Example.java 修改Example类，不在让其决定使用哪个Logger实现  public class Example {  private Logger logger;  public Example(Logger logger) {  this.logger = logger;  }  public void doStuff() {  ...// 其它的Example代码逻辑  logger.log();  ...  }  }  // TestMain.java 还需要修改使用Example类的地方  public class TestMain {  public static void main(String[] args){  // 使用FileLogger  Example example = new Example(new FileLogger());  example.doStuff();  // 使用ConsoleLogger  Example example = new Example(new ConsoleLogger());  example.doStuff();  }  } |



控制反转：从上面的代码我们可以看到，决定具体使用哪个Logger实现已经不是Example自己来决定了，而是由其他的实体（在Spring中是IOC容器）来决定，这样就实现了控制的反转，这就是所谓的控制反转。既然，控制反转搞清楚了，就来看看到底解决我们的问题没有。

因为在运行之前，Example都不知道Logger的具体实现，这样我们就可以重复使用现成的Logger实现，节省了资源。

因为现在Example根本不知道Logger的实现，所以，我们可以根据需要随时在使用时更换使用哪种Logger实现。实现了Client（Example）和Service（Logger）的解耦，提高了代码的可重用性。

因为在运行时期才确定Logger实现，所以我们可以使用仿制（mock）的Service去测试Example，使Example易于测试，达到了独立测试的目的。

减少了很多重复代码。当你使用传统模式实现时，当项目规模变得越来越大后，代码的重复就越多。而使用了控制反转就减少了很多重复的代码。

我们已经清楚的明白了什么是控制反转，但是什么是依赖注入呢？它和控制反转的具体关系是什么？

# 控制反转的实现方式

前面，我们讨论了什么是控制反转。所以，任何可以达到控制反转的实现都属于控制反转这个原则之下，那么到底有哪些方式呢。

* 使用工厂模式（Factory pattern）
* 使用服务定位器（Service Locator pattern）
* 使用依赖注入（Dependency Injection）
* 构造器注入（Constructor Injection）
* Setter方法注入
* 接口注入（Interface Injection）
* 使用上下文查找（contextualized lookup）
* 使用模板方法设计模式（template method design pattern）
* 使用策略设计模式（strategy design pattern）

从上面的方式我们看到依赖注入只是控制反转的一种方式，而我们在讨论控制反转例子中使用的实现控制反转的方式就是依赖注入中的构造器注入。至于什么是依赖注入我会在下一篇文章中介绍，也会介绍依赖注入的三种方式。