spring依赖注入

# 介绍

前一篇文章主要介绍了什么是控制反转，还有控制反转和依赖注入的关系，但是并没有深入讨论依赖注入，本文就详细讨论依赖注入的概念和依赖注入的主要三种方式。上文提到依赖注入只是控制反转的一种具体实现，那么到底依赖注入是个什么东西呢？借用James Shore在Dependency Injection Demystified中的定义：

Dependency injection means giving an object its instance variables.

依赖注入就是给一个对象实例变量，就是这样。上文中用来解释控制反转的例子就使用了依赖注入，我们看到Example作为Client使用了一个Service（Logger）作为依赖，这个Service就成为了Client的状态。依赖注入就是给这个Client它的依赖，而不是让Client自己去建造或者寻找依赖。上文中我们看到，最终我们是在Main方法中实例化的Logger依赖，然后传给了Example类，所以Example不是自己新建或寻找的类，而是由第三方注入的，所以这就是依赖注入。

# 深入分析

我们用一个例子来分析：

|  |
| --- |
| // 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 实例化依赖，并调用Example为其注入依赖  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();  }  } |

以上是我在上文使用的例子，我们看到Client（Example）把提供它依赖的责任委托给了第三方（提供者）。Client不允许调用提供者的代码，而是由提供者构建Service，然后调用Client把Service注入到Client。这就表示Client并不知道注入代码是怎样的，也不知道如何去构建Service，也不知道他所使用的Service的具体实现，它只需要知道它所需要功能的抽象描述（接口）。这就在使用和构建之间实现了分离。也体现了好莱坞原则那句话：

don’t call us, we’ll call you.

这句话怎么理解呢。传统编程的方式是Client自己来实例化自己的依赖，也就是自己主动。现在变成了不需要主动实例化（也就是Don’t call us），而是由提供者来调用Client，把依赖注入给Client（we’ll call you）。所以，John Munsch在回答How to explain dependency injection to a 5-year-old?中有个很好的类比：

When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn’t want you to have. You might even be looking for something we don’t even have or which has expired.

What you should be doing is stating a need, “I need something to drink with lunch,” and then we will make sure you have something when you sit down to eat.

简单翻译一下：

当你想从冰箱里为你自己取一些东西的时候，你可能会引起问题。你可能忘记关门，你可能取出一些妈妈和爸爸不希望你得到的东西。你甚至可能寻求某些我们根本就没有或者已经过期的东西。

你应该做的是陈述一个需求“我需要点饮料在午餐的时候喝”，然后我们会确保你在坐下来吃饭的时候会有喝的饮料。

我想，John Munsch已经说得很明白了。所以，如果你在看Martin Fowler的经典文章Inversion of Control Containers and the Dependency Injection pattern其中关于框架不理解的地方，可以结合这个类比去理解。

# 依赖注入的四个角色

说了这么多，现在理一理上面过程中涉及有哪些角色：

Service：被用来当做依赖的对象

Client：使用Service的对象

Interfaces：定义了Client将如何使用Service

Provider: 负责构建Service并注入给Client

Service和Client，我想大家应该都已经清楚了。Client使用，Service实现。

Interfaces定义了Client希望Service能提供的功能的抽象，正是因为有了这么一层，所以Client和Service才能实现解耦，便于测试和维护。而且，这样设计以后，只要接口的名字和API不改变，无论接口背后如何变化，Client都不需要改变。

Provider把Services注入到Client，它另一方面也构建Client。Provider的作用就是把Service A注入Client A，然后把Client A作为一个Service再注入另一Client中，最终构成一个大的对象图。在Spring中，这个Provider就是IOC容器。所以，Provider有很多叫法：装配工、容器、工厂、构建者、spring或者main。

# 依赖注入的优缺点

## 优点

依赖注入能够用来实现把配置放在代码之外的目的（比如Spring的XML配置），这样如果我们修改某两个部分的联系就不需要修改代码重新编译，只需要修改配置文件就行了。因为Client不依赖具体的Service，所以导致Client更加独立，所以更利于单元测试。这样就能够使用mock对象去模拟哪些并没有处在单元测试环境下的Service实现。这经常是使用依赖注入优先考虑的优点。

依赖注入允许Client依赖于抽象接口，所以并不需要Client为Service的修改做出改变。这增加了代码的可重用性、可测试性、可维护性。

减少了很多在Client的模板代码，因为所有的构建和注入都放在了Provider组件中。

依赖注入让一个对象和它的依赖实现了解耦。

## 缺点

使代码变得难懂。因为依赖注入分离了一个类的行为和构造，所以你不得不查看多个文件才能搞懂整个系统是如何运行的。

依赖注入使代码变得难以调试。因为你并不知道如何实例化和注入的依赖，所以Client并不知道自己的依赖是否已经全部被注入了。

依赖注入会使代码变得庞大，如果任何地方都是用依赖注入的话。

所以，我们在平时的开发过程中，一定要学会判断优劣。如果觉得某个模式很好，就一味的使用，而是根据具体的情况，选择不同的解决方案。

# 依赖注入方式

上面讲了这么多概念性东西，现在就来讲讲实质的，依赖注入主要的三种方式。

* 构造器注入：在构造器中传入依赖
* Setter方法注入：使用setter方法注入依赖
* 接口注入：Client继承一个拥有setter方法的setter接口，以便把依赖注入。而依赖自身也要继承一个injector（注入）接口，接受一个setter接口的对象，以便把自己注入给Client

## 构造器注入

这种方式要求Client的构造器提供一个依赖作为参数的构造方法。

|  |
| --- |
| // Example.java 提供Logger接口作为参数的构造方法  public class Example {  // 依赖于接口，依赖于抽象  private Logger logger;  // 使用该构造函数注入Logger依赖  public Example(Logger logger) {  this.logger = logger;  }  public void doStuff() {  ...// 其它的Example代码逻辑  logger.log();  ...  }  } |

## Setter方法注入

这种方式要求Client提供一个依赖的setter方法，以便设置（注入）依赖

|  |
| --- |
| // Example.java 提供Logger接口作为参数的构造方法  public class Example {  // 依赖于接口，依赖于抽象  private Logger logger;  public Example() {  }  // 提供setter方法注入依赖  public void setLogger(Logger logger) {  this.logger = logger;  }  public void doStuff() {  ...// 其它的Example代码逻辑  logger.log();  ...  }  } |

## 接口注入

这种方式需要我们新建两种接口，一种用于Client继承，以便设置它的依赖；一种用于Service（依赖）继承，以便可以把自己设置给Client。这种方式稍显复杂，慢慢解释。

|  |
| --- |
| 首先，我们需要新建一个SetLogger以便Client注入依赖。  // SetLogger.java 用于Client继承  pulbic interface SetLogger {  // setter方法以便注入依赖  void setLogger(Logger logger);  }  // Example.java 修改为继承自SetLogger接口  public class Example implements SetLogger {  // 依赖于接口，依赖于抽象  private Logger logger;  // 实现setter方法，注入依赖  @Override  public void setLogger(Logger logger) {  this.logger = logger;  }  } |
| 接下来，我们需要再建一个注入接口，以便依赖自己继承。然后在实现中调用Client的setter方法把自己注入到Clent中  // Injector.java 用于依赖继承，以便把自己注入给Client（target）  public interface Injector {  // 把依赖注入给targer  void inject(Object target);  }  // ConsoleLogger.java 依赖继承Injector和Logger接口  public class ConsoleLogger implements Logger, Injector {  // 继承Logger  @Override  public void log() {}  // 继承Injector，把自己注入Client（target）  @Override  public void inject(Object target) {  ((SetLogger) target).setLogger(this);  }  } |

## 构造器注入 vs Setter方法注入

首先，我们为什么不使用接口注入呢。因为接口注入更具有侵入性，我们需要写大量的接口去使依赖注入实现。所以，这也是为什么很多轻量级的框架更喜欢构造器注入和setter方法的原因。

首先，在构造器和setter方法之间选择的原则：在构造的时候就创建一个有效的对象。使用构造器注入能够清楚的看到构建一个有效的对象需要的属性。如果，有多种方式可选择就创建多个不同的构造函数。如果你使用setter方法注入，你不能保证构建一个有效对象所需的属性都已经被注入了。而使用构造函数的话，如果你没有提供所有的属性，则不会允许你构建。

另一个使用构造器住的好处就是构建不可改变的对象，这能避免安全性的问题。使用构造器注入能够让你隐藏那些不可编辑的属性，通过不提供setter方法。但是，如果你使用setter方法注入这将是不好实现的。

但是，凡是都有例外。如果你的对象拥有很多属性和依赖，如果都是用构造器注入，那么这个构造函数将变得非常的长，而且看起来非常的混乱。而且，如果你的属性都是一样类型的话，是用构造函数的时候将不能清晰的知道，那个参数对应的那个属性，你不得不去查阅API或者读源码。

到底该如何选择：总体原则就是优先选择构造器注入，如果构造器注入显得很麻烦的时候就选择setter方法注入。