依赖注入与JSR-330的参考实现——Guice

2015/05/12 | 分类： [基础技术](http://www.importnew.com/cat/basic) | [0 条评论](http://www.importnew.com/15947.html#respond) | 标签： [GUICE](http://www.importnew.com/tag/guice)

分享到：**1**

原文出处： [wwsun.me](http://wwsun.me/posts/di-guice-post.html)

依赖注入（控制反转的一种形式），它是Java开发主流中一个重要的编程范式（思维方式）。简单的说，使用DI技术可以让对象从别处获得依赖项，而不是由它自己来构造。使用DI有很多好处，它能降低代码之间的耦合度，让代码更易于测试、更易读。Java DI的官方标准是JSR-330，本文我们还会介绍JSR-330的参考实现Guice 3，一个轻量级 的DI框架。

IoC and DI

使用IoC范式编程时，程序逻辑的流程通常是由一个功能中心来控制的。而使用IoC，这个“中心控制”的设计原则会被反转过来。调用者的代码处理程序的执行顺序，而程序逻辑则被封装在接受调用的子流程中。通过一个例子来理解IoC:

在GUI应用中，GUI框架负责控制调用事件处理器，而不是应用逻辑。当用户点击了一个动作，比如“向前”，GUI框架会自动调用 对应的事件处理器，而应用逻辑可以把重点放在处理动作上。程序的控制被反转了，将控制权由应用逻辑转移到了GUI框架。

IoC也被称为好莱坞原则：会有另一段代码拥有最初的控制线程，并且由它来调用你的代码，而不是由你的代码调用它。

*不要给我们打电话，我们会打给你。——好莱坞原则*

IoC有多种不同的实现，包括工厂模式、服务器定位模式，当然还有依赖注入。需要注意的是，DI并不等于IoC，DI只是IoC的一种实现方式，IoC是一种机制。

DI是IoC的一种特定形态，是指寻找依赖项的过程不在当前执行代码的直接控制之下。Java中为DI提供的容器有Guice、Spring、PicoContainer等。DI的好处有：松耦合、易测试、强内聚、可重用、更轻盈的代码。

Example

下面编写代码来解释到底什么是DI，如何使用DI。我们首先编写传统的代码，然后使用工厂模式解耦，进而再使用DI来改进代码，通过这个过程，你将了解到DI的精妙之处。这些改进都基于同一个关键技术，即面向接口编程。

假设你想找到所有对Java可开发人员比较友善的好莱坞经纪人。首先，我们有了一个AgentFinder接口，及其两个实现类SpreadSheetAgentFinder和WebServiceAgentFinder。AgentFinder接口如下：

|  |  |
| --- | --- |
| 1  2  3 | public interface AgentFinder {      public List<Agent> findAllAgents();  } |

传统方式寻找友善经纪人

为了找到经纪人，项目中有个默认的HollywoodService类，它会从SpreadSheetAgentFinder里得到一个经纪人列表，并且过滤出友善的经纪人，最终返回该列表。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class HollywoodService {      public static List<Agent> getFriendlyAgents() {          AgentFinder finder = new SpreadsheetAgentFinder();          List<Agent> agents = finder.findAllAgents();          List<Agent> friendlyAgents = filterAgents(agents,"Java Developers");            return friendlyAgents;      }      //filterAgents  } |

这是最传统的编码方式，显然，HollywoodService被SpreadsheetAgentFinder这个AgentFinder的具体实现死死的绑定住了。

为了改进这个问题，通常我们会想到一个常用的[设计模式](http://www.amazon.cn/gp/product/B001130JN8/ref=as_li_qf_sp_asin_il_tl?ie=UTF8&tag=importnew-23&linkCode=as2&camp=536&creative=3200&creativeASIN=B001130JN8)——工厂模式。工厂模式可以一定程度上解耦程序，它也是IoC的一种实现方式。

工厂模式

利用工厂模式（其实这里用到的是一个简单工厂）重新编写上面的代码，如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public List<Agent> getFriendlyAgents(String agentFinderType) {      AgentFinderFactory factory = AgentFinderFactory.getInstance();      AgentFinder finder = factory.getAgentFinder(agentFinderType);      List<Agent> agents = finder.findAllAgents();      return filterAgents(agents, "Java Developers");  } |

如你所见，这里不再有具体的实现来黏住你，而是通过注入agentFinderType的方式让你选择想要的AgentFinder。但这里仍然还有问题：

1. 代码注入的仅仅是一个引用凭据（agentFinderType），而不是真正实现AgentFinder的对象
2. 方法getFriendlyAgents中还有获取其依赖的代码，达不到只关注自身智能的理想状态

我们需要通过DI来达成这两个目标。

手工实现DI

|  |  |
| --- | --- |
| 1  2  3  4 | public static List<Agent> getFriendlyAgents(AgentFinder finder){      List<Agent> agents = finder.findAllAgents();      return filterAgents(agents,"Java Developers");  } |

上面的代码是一个纯手工打造的DI方案，AgentFinder被直接注入到getFriendlyAgents方法中。现在这个getFriendlyAgents方法干净利落，只专注于纯业务逻辑。

但是，这种手工方式的DI显然存在问题，如何配置AgentFinder具体实现的问题并没有解决，原本AgentFinderFactory要做的工作还是要找一个地方去做。解决这个问题的方式是借助DI框架，而DI框架就是把你的代码打包起来的运行时环境，在你需要的时候注入依赖项。

使用Guice

Java中DI方面的标准规范是JSR-330，它提供了统一的DI通用功能的标准，而且提供了你需要了解的幕后规则及限制。这里不具体介绍JSR-330，有兴趣的可以查找相关资料阅读。我们重点介绍JSR-330的参考实现Guice，它是一个由Google实现的针对Java 6以上版本的流行的、轻量级的DI框架。

下面我们使用Guice来解决手工方式实现DI的不足问题：

首先我们需要创建一个定义绑定关系的AgentFinderModule模块（配置类）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class AgentFinderModule extends AbstractModule{      @Override      protected void configure() {          bind(AgentFinder.class).to(WebServiceAgentFinder.class);      }  } |

绑定关系的确立在调用Guice的bind方法时发生，把要绑定的类（AgentFinder)传给它，然后调用to方法指明要注入到哪个实现类。

下面我们来Guice版本的HollywoodService：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class HollywoodServiceGuice {      private AgentFinder finder = null;        @Inject      public HollywoodServiceGuice(AgentFinder finder){          this.finder = finder;      }        public List<Agent> getFriendlyAgents(){          List<Agent> agents = finder.findAllAgents();          return filterAgents(agents, "Java Developers");      }        //...  } |

我们在模块中声明了绑定关系，下面我们就可以让注入器构建对象关系图了。接下来我们看看在独立Java程序和Web应用程序这两种情况下分别要如何实现：

1.构建Guice对象关系图——独立Java程序

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class HollywoodServiceClient {       public static void main(String[] args) {           Injector injector = Guice.createInjector(new AgentFinderModule());           HollywoodServiceGuice hollywoodServiceGuice = injector.getInstance(HollywoodServiceGuice.class);           List<Agent> agents = hollywoodServiceGuice.getFriendlyAgents();       }   } |

2.构建Guice对象关系图——Web应用程序 把Gucie-servlet.jar加到web应用的类库中，然后再web.xml中添加下面的配置项：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <filter>      <filter-name>guiceFilter</filter-name>      <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>    </filter>      <filter-mapping>      <filter-name>guiceFilter</filter-name>      <url-pattern>/\*</url-pattern>    </filter-mapping> |

3.然后是标准动作，扩展ServletContextListener以便使用Guice的ServeltModule：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class MyGuiceServletConfig extends GuiceServletContextListener {       @Override     protected Injector getInjector() {       return Guice.createInjector(new ServletModule());     }   } |

4.最好，将其加入到web.xml文件中，以便servlet容器在部署应用时触发该类：

|  |  |
| --- | --- |
| 1  2  3 | <listener>     <listener-class>com.example.MyGuiceServletConfig</listener-class>   </listener> |

经由注入器创建HollywoodServiceGuice，你得到一个配置完备的类，马上就可以调用其中的getFriendlyAgents方法了。

关于JSR-330和Guice的更多用法，请参考其[Github页面](https://github.com/google/guice)。

References

1. https://github.com/google/guice/wiki/GettingStarted
2. The Well-Grounded Java Developer, chapter 3, Dependency Injection