通过 Guice 进行依赖项注入

使用较少模板的可测试代码



Nicholas Lesiecki  
2009 年 1 月 08 日发布

[Weibo](http://service.weibo.com/share/share.php?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Fjava%2Fj-guice.html%25&title=%E9%80%9A%E8%BF%87%20Guice%20%E8%BF%9B%E8%A1%8C%E4%BE%9D%E8%B5%96%E9%A1%B9%E6%B3%A8%E5%85%A5&language=zh_cn)[Google+](https://plus.google.com/share?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Fjava%2Fj-guice.html&t=%E9%80%9A%E8%BF%87%20Guice%20%E8%BF%9B%E8%A1%8C%E4%BE%9D%E8%B5%96%E9%A1%B9%E6%B3%A8%E5%85%A5)[用电子邮件发送本页面](mailto:?subject=%E9%80%9A%E8%BF%87%20Guice%20%E8%BF%9B%E8%A1%8C%E4%BE%9D%E8%B5%96%E9%A1%B9%E6%B3%A8%E5%85%A5&body=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Fjava%2Fj-guice.html)

[Comments](https://www.ibm.com/developerworks/cn/java/j-guice.html#icomments)

[0](https://www.ibm.com/developerworks/cn/java/j-guice.html" \l "icomments)

Guice 是一个*依赖项注入*（DI）框架。几年来我一直建议开发人员使用 DI，因为它提高了可维护性、可测试性和灵活性。通过观察工程师对 Guice 的反馈，我发现说服程序员去采用一种新技术的最好方法是使这种技术简单易用。Guice 让 DI 变得很简单，因此 Google 采用了这种方法。我希望本文能帮助您轻松学习 Guice。

**介绍而不是争论**

Guice 紧跟极富创造力的 DI 框架家族的脚步（即 PicoContainer，它是有争议的框架之一。它的支持者通过一个页面解释了它的历史和相互关系；请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-guice.html#artrelatedtopics)）。Guice 后来者极力争辩哪一个框架更好，以及是否需要另一个 DI 框架。与任何技术选择一样，每个库都有它的优点和缺点。我认为 Guice 带来了一些新意，不过在这篇文章中，我只是介绍 Guice 的功能，不会发表任何评论（您可以上网搜索 “guice 对比 spring”，查看一些活跃的讨论）。

Guice 2.0 beta

在写这篇文章时，Guice 开发团队正在奋力编写 Guice 2.0，希望能在 2008 年底之前发布。早期的 beta 发布在 Google 代码下载站点（请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-guice.html#artrelatedtopics)）。这是一个好消息，因为 Guice 团队添加了一些新功能，使 Guice 代码的使用和理解变得更简单。beta 版没有最终版中的一些功能，但是 beta 很稳定，质量也很好。事实上，Google 在产品软件中使用的是 beta 版。我建议您使用 beta 版。这篇文章是专门为 Guice 2.0 编写的，介绍了 Guice 的一些新功能，但没有讨论 1.0 中已经废弃的一些功能。Guice 团队向我保证：这里讨论的功能在最终发行版和当前 beta 版中都是一样的。

如果您已经了解了 DI，而且知道为什么要借助一个框架来使用 DI，那么您可以跳到 [通过 Guice 进行基本注入](https://www.ibm.com/developerworks/cn/java/j-guice.html#basic) 小节。否则，请继续阅读，了解 DI 的好处。

DI 案例

我将以一个例子开始。假设我正在编写一个超级英雄（superhero）应用程序，同时实现一个名为 Frog Man 的 hero（英雄）。清单 1 是相关代码和第一个测试（您一定明白编写单元测试的重要性，这里就不多说了）。

清单 1. 一个基本 hero 及其测试

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class FrogMan {    private FrogMobile vehicle = new FrogMobile();    public FrogMan() {}    // crime fighting logic goes here...  }    public class FrogManTest extends TestCase {   public void testFrogManFightsCrime() {      FrogMan hero = new FrogMan();      hero.fightCrime();      //make some assertions...    }  } |

似乎一切正常，但在运行测试时出现了如清单 2 所示的异常：

清单 2. 依赖项出现问题

|  |  |
| --- | --- |
| 1  2  3  4  5 | java.lang.RuntimeException: Refinery startup failure.    at HeavyWaterRefinery.<init>(HeavyWaterRefinery.java:6)    at FrogMobile.<init>(FrogMobile.java:5)    at FrogMan.<init>(FrogMan.java:8)    at FrogManTest.testFrogManFightsCrime(FrogManTest.java:10) |

似乎 FrogMobile 构建了一个 HeavyWaterRefinery，假设我不能在测试中构建其中一个依赖项。当然，我可以在生产环境中实现这一点，但是不能保证能在测试中构建第二个提炼厂（refinery）。在现实生活中，您不可能提炼出氧化氘，但您可以依赖远程服务器和强大的数据库。原理是一样的：这些依赖项很难启动，交互起来也很慢，这使得测试比平时更容易失败。

输入 DI

为了避免这个问题，您可以创建一个接口（例如 Vehicle），使 FrogMan 类接受 Vehicle 作为一个构造函数参数，如清单 3 所示：

清单 3. 依赖接口并注入它们

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class FrogMan {    private Vehicle vehicle;      public FrogMan(Vehicle vehicle) {      this.vehicle = vehicle;    }    // crime fighting logic goes here...  } |

这种用法就是 DI 的本质 — 使类通过引用接口而不是构建接口（或使用静态引用）来接受它们的依赖项。清单 4 显示了 DI 如何使测试变得更简单：

清单 4. 测试可以使用 mock 而不是麻烦的依赖项

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | static class MockVehicle implements Vehicle {    boolean didZoom;      public String zoom() {      this.didZoom = true;      return "Mock Vehicle Zoomed.";    }  }    public void testFrogManFightsCrime() {    MockVehicle mockVehicle = new MockVehicle();      FrogMan hero = new FrogMan(mockVehicle);    hero.fightCrime();      assertTrue(mockVehicle.didZoom);    // other assertions  } |

这个测试使用了一个手动编写的 mock 对象来替换 FrogMobile。DI 不仅在测试中省去了麻烦的 refinery 启动过程，而且使测试不用了解 FrogMobile 具体细节。需要的仅是一个 Vehicle 接口。除了使测试变得更简单之外，DI 还有助于提高代码的总体模块性和可维护性。现在，如果想将 FrogMobile 切换为 FrogBarge，可以不修改 FrogMan。所有 FrogMan 都依赖于 Vehicle 接口。

不过这里有一个陷阱。如果您是第一次阅读 DI，可能会想：“这下好了，现在所有 FrogMan 的*调用方* 都必须知道 FrogMobile（refinery、refinery 的依赖项，依此类推……）”。但如果是这样，DI 就永远不会这么流行。您可以不增加调用方的负担，而是编写一些*工厂* 来管理对象及其依赖项的创建。

工厂是存放框架的地方。工厂需要大量冗长重复的代码。工厂往往会让程序员（和读者）很痛苦，他们甚至会嫌它麻烦而放弃编写。Guice 和其他 DI 框架可作为 “超级工厂”，您可以通过配置它们来构建对象。配置 DI 框架比自己编写工厂容易得多。因此，程序员编写的代码大部分是 DI 样式的。测试越多代码就越好，程序员以后也就越省事。

通过 Guice 进行基本注入

我希望在我的介绍之后，您会相信 DI 能为您的设计增加价值，而且使用框架会使工作更轻松。现在让我们从 @Inject 注释和模块开始深入讨论 Guice。

告诉 Guice 给类添加 @Inject

FrogMan 与 Guice 上的 FrogMan 之间的唯一区别是 @Inject。清单 5 显示了 FrogMan 带有注释的构造函数：

清单 5. FrogMan 已经加上 @Inject

|  |  |
| --- | --- |
| 1  2  3  4 | @Inject  public FrogMan(Vehicle vehicle) {    this.vehicle = vehicle;  } |

一些工程师不喜欢给类添加 @Inject。他们更喜欢一个完全忽略 DI 框架的类。这种说法有一定道理，但是我不大赞同。依赖项的注入会使注释的作用更加明显。@Inject 标记只在您要求 Guice 构建类时才有意义。如果不要求 Guice 构建 FrogMan，这个注释对代码行为就没有任何影响。这个注释恰当地指出了 Guice 将参与构建类。但是，使用它需要源代码级别的访问。如果这个注释带来不便，或者正在使用 Guice 创建无法控制其源代码的对象，那么 Guice 就会用一个替代机制（请参阅本文后面的 [provider 方法的其他用法](https://www.ibm.com/developerworks/cn/java/j-guice.html#other) 侧边栏）。

告诉 Guice 您需要哪个依赖项

Guice 知道您的 hero 需要一个 Vehicle 后，它需要知道提供什么 Vehicle。清单 6 包含一个 Module：一个特殊的类，用于告诉 Guice 各个接口对应的实现。

清单 6. HeroModule 将 Vehicle 绑定到 FrogMobile

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class HeroModule implements Module {    public void configure(Binder binder) {      binder.bind(Vehicle.class).to(FrogMobile.class);    }  } |

模块就是一个具有某种单实例对象方法的接口。Guice 传递给模块的 Binder 用于告诉 Guice 您想如何构造对象。绑定程序 API 形成一种 *区域特定语言*（请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-guice.html#artrelatedtopics)）。这种小语言允许您编写表达式代码，比如 bind(X).to(Y).in(Z)。后面将提供更多有关绑定程序作用的例子。每次调用 bind 都会创建一个*绑定*，Guice 将使用绑定集解析注入请求。

使用 Injector 启动

然后，使用 Injector 类启动 Guice。通常需要尽早在程序中创建注入器。这样 Guice 能够帮助您创建大部分对象。清单 7 包含一个以 Injector 开始的示例 main 程序：

清单 7 使用 Injector 启动应用程序

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Adventure {    public static void main(String[] args){      Injector injector = Guice.createInjector(new HeroModule());      FrogMan hero = injector.getInstance(FrogMan.class);      hero.fightCrime();    }  } |

为了获取注入器，需要在 Guice 类上调用 createInjector。向 createInjector 传递一个模块列表，用于配置它本身（本例只有一个，但您可以添加一个配置 evildoer 的 VillainModule）。拥有注入器后，使用 getInstance 向它请求对象，传递您想返回的 .class（细心的读者会注意到您不需要告诉 Guice 有关 FrogMan 的信息。如果您请求一个具体类，而它有一个 @Inject 构造函数或公共非参数构造函数的话，Guice 就会创建这个类，而无需调用 bind）。

这是 Guice 构造对象的第一种方式：显式询问。但是，您不会希望在启动例程之外使用这个操作。更好、更简单的方式是让 Guice 注入依赖项、依赖项的依赖项，依此类推（正如谚语所说：“背起地球的海龟站在另一个海龟的背上”；请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-guice.html#artrelatedtopics)）。最初看来，这似乎比较麻烦，但您很快就会习惯这种用法。例如，清单 8 显示了一个注入了 FuelSource 的 FrogMobile：

清单 8. FrogMobile 接受一个 FuelSource

|  |  |
| --- | --- |
| 1  2  3  4 | @Inject  public FrogMobile(FuelSource fuelSource){    this.fuelSource = fuelSource;  } |

这意味着，当您检索 FrogMan 时，Guice 会构建一个 FuelSource、一个 FrogMobile，最后是一个 FrogMan。即使应用程序与注入器只交互一次，也是如此。

当然，您并不总是有机会控制应用程序的 main 例程。例如，许多 Web 框架自动构建 “操作”、“模板” 或其他一些初始服务。总是可以找到一个地方插入 Guice，不管是使用该框架的一个插件，还是使用一些自己手动编写的代码（例如，Guice 项目发布了一个 Struts 2 插件，它允许 Guice 配置您的 Strut 操作；请参阅 [参考资料](https://www.ibm.com/developerworks/cn/java/j-guice.html#artrelatedtopics)）。

其他注入形式

到目前为止，我展示了 @Inject 应用于构造函数的用法。当 Guice 找到注释时，它会挑选构造函数参数，并试图为每个参数找到一个配置绑定。这称为 *构造函数注入*。根据 Guice 的最佳实践指南，构造函数注入是询问依赖项的首选方式。但这不是唯一的方式。清单 9 显示了配置 FrogMan 类的另一种方式：

清单 9. 方法注入

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class FrogMan{    private Vehicle vehicle;      @Inject    public void setVehicle(Vehicle vehicle) {      this.vehicle = vehicle;    }  //etc. ... |

注意，我没有使用注入的构造函数，而是改用一个带有 @Inject 标记的方法。Guice 会在构造好 hero 之后立即调用此方法。Spring 框架的忠实用户可以将此方法视为 “setter 注入”。不过，Guice 只关心 @Inject；您可以任意命名这个方法，它可以带有多个参数。此方法可以是包保护的，也可以是私有方法。

如果您认为 Guice 访问私有方法不是很好，可以参见清单 10，其中 FrogMan 使用了*字段注入*：

清单 10. 字段注入

|  |  |
| --- | --- |
| 1  2  3  4 | public class FrogMan {    @Inject private Vehicle vehicle;    public FrogMan(){}  //etc. ... |

同样，所有 Guice 都只关心 @Inject 注释。字段注入查找注释的所有字段并试图注入相应的依赖项。

哪种方法最好

三个 FrogMan 版本都展示了相同的行为：Guice 在构建时注入相应的 Vehicle。不过，像 Guice 的作者一样，我更喜欢构造函数注入。下面简单分析这三种方式：

* **构造函数注入** 很简单。因为 Java 技术能保证构造函数调用，您不用担心出现未初始化的对象 — 不管是不是由 Guice 创建的。您还可以将字段标记为 final。
* **字段注入** 会影响可测试性，特别是将字段标记为 private 时。这破坏了使用 DI 的主要目的。应该尽量少使用字段注入。
* **方法注入** 在您不控制类的实例化时很有用。如果您有一个需要某些依赖项的超类，也可以使用方法注入（构造函数注入会使这种情况变得很复杂）。

选择实现

现在，假设应用程序中有多个 Vehicle。一样英勇的 Weasel Girl 无法驾驭 FrogMobile！同时，您不想在 WeaselCopter上硬编码依赖项。清单 11 显示了 Weasel Girl 请求一种更快的传输模式：

清单 11. 使用注释请求某种特定的实现

|  |  |
| --- | --- |
| 1  2  3  4 | @Inject  public WeaselGirl(@Fast Vehicle vehicle) {    this.vehicle = vehicle;  } |

在清单 12 中，HeroModule 使用绑定函数告诉 Guice WeaselCopter 是 “很快” 的：

清单 12. 告诉 Guice Module 中的相关注释

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class HeroModule implements Module {   public void configure(Binder binder) {      binder.bind(Vehicle.class).to(FrogMobile.class);      binder.bind(Vehicle.class).annotatedWith(Fast.class).to(WeaselCopter.class);    }  } |

注意，我选择了一个注释，描述我想以抽象形式描述的工具种类（@Fast），而不是与实现太接近的注释（@WeaselCopter）。如果您使用的注释将想要的实现描述得太精确，就让读者觉得创建一个隐式依赖项。如果使用 @WeaselCopter，而且 Weasel Girl 借用了 Wombat Rocket，就会对程序员阅读和调试代码造成混淆。

要创建 @Fast 注释，需要复制清单 13 中的模板：

清单 13. 复制粘贴这段代码以创建一个绑定注释

|  |  |
| --- | --- |
| 1  2  3  4 | @Retention(RetentionPolicy.RUNTIME)  @Target({ElementType.FIELD, ElementType.PARAMETER})  @BindingAnnotation  public @interface Fast {} |

如果您编写了大量 BindingAnnotations，就会得到许多这样的小文件，每个文件只是注释名称不同。如果您觉得这很繁琐，或者需要执行快速的原型设计，可以考虑 Guice 的内置 @Named 注释，它接受一个字符串属性。清单 14 展示了这种替代方法：

清单 14. 使用 @Named 代替自定义注释

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // in WeaselGirl  @Inject  public WeaselGirl(@Named("Fast") Vehicle vehicle) {    //...  }    // in HeroModule  binder.bind(Vehicle.class)    .annotatedWith(Names.named("Fast")).to(WeaselCopter.class); |

这种方法是可行的，但由于名称只在字符串内有效，所以这不能利用编译时检查和自动补齐。总的来说，我更愿意自己编写注释。

如果您根本不想使用注释，怎么办？即使添加 @Fast 或 @Named("Fast") 都会使类在某种程度上影响配置本身。如果想知道如何解决这个问题，请接着阅读。

provider 方法

**provider 方法的其他用法清单 16. provider 方法允许不注释源**

如果您不放心 Guice 自带的注释，或者无法使用它们（例如，您正在创建一个第三方类），那么可以使用 provider 方法完美地解决这个问题。因为 provider 方法在模块内有效，所以可以注释这个方法，而不用担心其余的源代码会看到注释。例如，清单 16 包含一个针对 BadgerBoy（一个第三方 hero）的 provider 方法：

@Provides @Inject  
private BadgerBoy provideBadgerBoy(WeaselCopter copter) {  
return new BadgerBoy(copter);  
}

通过这种 provider 方法，您可以使用 Guice 配置 BadgerBoy，甚至选择它需要的 Vehicle，这一切操作完全不用更改 BadgerBoy。这意味着您完全可以不使用 @Inject 或绑定注释，比如 @Fast。还可以使用 provider 方法选择依赖项。

选择哪种方法取决于您的喜好，不管您是否使用第三方类，或者希望以多紧密的程度将类和它的依赖项联系起来。provider 方法允许自由选择。注释的依赖项允许配置信息更接近类，这使得源更易于理解，同时能保持一定的灵活性和可测试性。应用程序可能会根据环境同时使用两种方法来指定依赖项。

如果每次探险都派遣 Frog Man，您可能会厌烦。您喜欢在每个场景中出现的 hero 是随机的。但是，Guice 的默认绑定程序 API 不允许出现 “每次调用时将 Hero 类绑定到一个不同的实现” 这样的调用。不过，您*可以* 告诉 Guice 使用一种特殊的方法来创建每个新的 Hero。清单 15 显示了将一个新方法添加到 HeroModule 中，并用特殊的 @Provides 注释进行注释：

清单 15. 使用 provider 编写自定义创建逻辑

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Provides  private Hero provideHero(FrogMan frogMan, WeaselGirl weaselGirl) {    if (Math.random() > .5) {      return frogMan;    }    return weaselGirl;  } |

Guice 会自动发现具有 @Provides 注释的 Module 中的所有方法。根据 Hero 的返回类型，在您请求某个 hero 时，Guice 会进行计算，它应该调用 provider 方法来提供 hero。您可以为 provider 方法添加逻辑以构建对象并在缓存中查询它，或者通过其他方式获得它。provider 方法是将其他库集成到 Guice 模块中的很好方式。它们也是从 Guice 2.0 开始提供的新方法（Guice 1.0 中只编写自定义 provider 类，这比较乏味，而且更加繁琐。如果您已经决定使用 Guice 1.0，用户指南中有这种旧方法的文档，而且在本文随附的 [示例代码](https://www.ibm.com/developerworks/cn/java/j-guice.html#artdownload) 中，您可以找到一个自定义 provider）。

在清单 15 中，Guice 自动使用正确的参数注入 provider 方法。这意味着 Guice 将从它的绑定列表中找到 WeaselGirl 和 FrogMan，您无需在 provider 方法中手动构建它们。这演示了 “海龟背地球” 原则（海龟背地球，哪海龟下面是什么呢？是由另一只海龟背着，如此反复）。您依靠 Guice 来提供依赖项，即使是配置 Guice 模块本身。

请求一个 Provider 而不是一个依赖项

假设一个故事（Saga）中要有多个 hero。如果要求 Guice 注入一个 Hero，只会得到一个 hero。但如果您请求一个 “hero provider”，就可以根据需要创建任意多的 hero，如清单 17 所示：

清单 17. 注入 provider 来控制实例化

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Saga {    private final Provider<Hero> heroProvider;      @Inject    public Saga(Provider<Hero> heroProvider) {      this.heroProvider = heroProvider;    }      public void start() throws IOException {      for (int i = 0; i < 3; i++) {        Hero hero = heroProvider.get();        hero.fightCrime();      }    }  } |

提供者也可以推迟英雄的出场时间，直到传奇真正开始。如果英雄依赖于时间敏感或上下文敏感的数据，这就会很方便。

Provider 接口有一个方法：get<T>。要访问提供的对象，调用这个方法即可。每次有没有获取新对象以及对象如何配置取决于 Guice 是如何配置的（参阅下面的 [作用域](https://www.ibm.com/developerworks/cn/java/j-guice.html#scopes) 部分，了解单实例对象和其他长生命周期对象的详细信息）。在本例中，Guice 使用 @Provides 方法，因为它是构建新 Hero 的注册方式。这意味着该传奇应该由任意三位英雄混合而成。

不要把提供者与 provider 方法弄混淆了（在 Guice 1.0，这两者更难区分开来）。尽管该 Saga 是从自定义 @Provides 方法中获得它的英雄，但您可以请求任意 Guice 实例化依赖项的一个 Provider。如果需要，可以根据清单 18 重新编写 FrogMan 的构造函数：

清单 18. 请求 Provider 而不是依赖项

|  |  |
| --- | --- |
| 1  2  3  4 | @Inject  public FrogMan(Provider<Vehicle> vehicleProvider) {    this.vehicle = vehicleProvider.get();  } |

（注意您完全不用更改这个模块代码）。重新编写没有任何作用；只是说明您总是可以请求 Provider，而不用直接请求依赖项。

作用域

默认情况下，Guice 为每个请求的依赖项创建一个新实例。如果对象是轻量级的，这个策略可以很好地工作。但是，如果有一个创建开销很大的依赖项，就可能需要在几台客户机之间共享实例。在清单 19 中，HeroModule 将 HeavyWaterRefinery作为一个单实例对象绑定：

清单 19. 将 HeavyWaterRefinery 绑定为一个单实例对象

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class HeroModule implements Module {    public void configure(Binder binder) {      //...      binder.bind(FuelSource.class)        .to(HeavyWaterRefinery.class).in(Scopes.SINGLETON);    }  } |

**单实例对象是恶魔吗？**

如果您在网上搜索 “单实例对象是恶魔”，会找到许多有关单实例对象的帖子、文章和评论。似乎 “应用程序单实例对象” 和 “JVM 单实例对象” 之间存在某种区别（评论员激烈讨论的话题）。JVM 单实例对象通过使用语言技巧强制它的 “单实例对象性质”，并强制客户机使用静态引用去引用它。另一方面，应用程序单实例对象不会强制使用这个策略。另外的层级（例如 Guice）会强制要求*每个应用程序* 存在一个类实例。客户机代码不知道这个规则，这使得设计非常灵活，测试也容易编写。

这意味着 Guice 会一直保持 “提炼厂” 可用，只要另一个实例需要燃料源，Guice 就会注入*相同* 的 “提炼厂”。这避免了在应用程序中启动多个 “提炼厂”。

在选择作用域时，Guice 提供了一个选项。可以使用绑定程序配置它们，或者直接注释依赖项，如清单 20 所示：

清单 20. 改用注释选择作用域

|  |  |
| --- | --- |
| 1  2 | @Singleton  public class HeavyWaterRefinery implements FuelSource {...} |

Guice 提供了超出范围的 Singleton 作用域，但它允许您定义自己的作用域（如果您愿意）。例如，Guice servlet 包提供了两个其他作用域：Request 和 Session，它们为 servlet 请求和 servlet 会话提供类的一个独特实例。

常量绑定和模块配置

HeavyWaterRefinery 需要一个许可密钥才能启动。Guice 可以绑定常量值和新实例。请查看清单 21：

清单 21. 在模块中绑定常量值

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class HeavyWaterRefinery implements FuelSource {    @Inject    public HeavyWaterRefinery(@Named("LicenseKey") String key) {...}  }    // in HeroModule:  binder.bind(String.class)    .annotatedWith(Names.named("LicenseKey")).toInstance("QWERTY"); |

这里有必要使用绑定注释，否则 Guice 将不能区分不同的 String。

注意，尽管前面不推荐使用，我还是选择使用 @Named 注释。因为我想显示清单 22 所示的代码：

清单 22. 使用属性文件配置模块

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | //In HeroModule:  private void loadProperties(Binder binder) {    InputStream stream =      HeroModule.class.getResourceAsStream("/app.properties");    Properties appProperties = new Properties();    try {      appProperties.load(stream);      Names.bindProperties(binder, appProperties);    } catch (IOException e) {      // This is the preferred way to tell Guice something went wrong      binder.addError(e);    }  }    //In the file app.properties:  LicenseKey=QWERTY1234 |

这段代码使用 Guice Names.bindProperties 实用函数，通过恰当的 @Named 注释将 app.properties 文件中的每个属性与一个常量绑定。这本身就很好，而且还显示了您可以使模块代码更复杂。如果喜欢，可以从数据库或 XML 文件加载绑定信息。模块是纯 Java 代码，这提供了*很大的* 灵活性。

结束语

Guice 主要概念小结：

* 使用 @Inject 请求依赖项。
* 将依赖项与 Module 中的实现绑定。
* 使用 Injector 引导应用程序。
* 使用 @Provides 方法增加灵活性。

需要了解的 Guice 知识还很多，但您应该先掌握这篇文章中讨论的内容。我建议下载它，以及本文的 [示例代码](https://www.ibm.com/developerworks/cn/java/j-guice.html#artdownload)。当然，您也可以创建自己的示例应用程序，这就更好了。通过示例深入了解概念但又不用考虑生产代码是很有意思的。如果要了解更多 Guice 高级功能（比如面向方面编程支持），建议您访问 [参考资料](https://www.ibm.com/developerworks/cn/java/j-guice.html#artrelatedtopics) 中的一些链接。

说到生产代码，DI 的一个缺点是它可能感染病毒。注入一个类后，它会导致注入下一个类，依此类推。不过这很好，因为 DI 使代码更好。另一方面，这需要大量重构现有代码。为了使工作易于管理，可以将 Guice Injector 存储在某处并直接调用它。应该将这当作一根临时需要的拐杖，但最后一定可以摆脱它。

Guice 2.0 即将推出。有一些功能我还没有讨论，它可以使模块的配置更简单，而且能支持更大、更复杂的配置方案。可以访问 [参考资料](https://www.ibm.com/developerworks/cn/java/j-guice.html#artrelatedtopics) 中的链接了解即将面世的功能。

我希望您会考虑将 Guice 添加到工具包中。根据我的经验，DI 对于实现灵活的可测试代码库特别有用。Guice 使 DI 变得简单而有趣。还有什么比容易编写的、灵活的、可测试的代码更好呢？

下载资源

* [本文的 Java 文件](http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=362635&filename=j-guice.zip&method=http&locale=zh_CN) (j-guice.zip | 19KB)

相关主题

* 您可以参阅本文在 developerWorks 全球网站上的 [英文原文](http://www.ibm.com/developerworks/java/library/j-guice.html)。
* [Guice](http://code.google.com/p/google-guice/)：Guice 的主页是一个不错的起点，[Guice 用户指南](http://google-guice.googlecode.com/files/Guice%201.0%20User%27s%20Guide.pdf) 讨论了大量高级 Guice 特性。
* [依赖项注入](http://martinfowler.com/articles/injection.html) 和 [领域特定语言](http://www.martinfowler.com/bliki/DomainSpecificLanguage.html)：Martin Fowler 总是能提供非常棒的参考资料。请访问他的网站，查看关于这些主题的讨论。
* [Inversion of Control History](http://www.picocontainer.org/inversion-of-control-history.html)：阅读 DI 框架之间的历史和相互关系，比如 [Spring](http://www.springframework.org/)、[PicoContainer](http://www.picocontainer.org/) 和 [HiveMind](http://hivemind.apache.org/)。
* “[Expert Guice: 50 some odd ways to Guice up your Java](http://parleys.com/display/PARLEYS/Home#slide=1;title=Guice;talk=14123111)”：通过录像演示稿查看 Guice 创建者 Bob Lee 对 Guice 的说明，以及使用它的最好方式。
* [Guice 2.0](http://code.google.com/p/google-guice/wiki/Changes20)：了解新版 Guice 中的新功能。
* [Turtles all the way down](http://en.wikipedia.org/wiki/Turtles_all_the_way_down)：Wikipedia 对此进行了解释。
* “[使用模仿对象进行单元测试](http://www.ibm.com/developerworks/cn/java/j-mocktest/)”（Alexander Chaffee 和 William Pietri，developerWorks，2002 年 11 月）和 “[Mock Roles, not Objects](http://www.jmock.org/oopsla2004.pdf)”（Steve Freeman et al.，jmock.org，2004）：了解使用 mock 对象进行单元测试的更多信息。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：提供了几百篇有关 Java 编程各个方面的文章，包括关于 [依赖项注入](http://www.ibm.com/developerworks/search/searchResults.jsp?searchType=1&pageLang=&displaySearchScope=dW&searchSite=dW&query=dependency+injection&searchScope=dW) 的许多其他文章。
* [Guice](http://code.google.com/p/google-guice/downloads/list)：下载 Guice 2 beta 发行版。
* [Guice 插件](http://cwiki.apache.org/S2PLUGINS/guice-plugin.html)：获取 Strut 的 Guice 插件。