http://code.google.com/p/google-guice/

用 Guice 写 Java

Guice 1.0 用户指南

（20070326 王咏刚 译自：http://docs.google.com/Doc?id=dd2fhx4z_5df5hw8）

Guice **(**读作**"juice")**是超轻量级的，下一代的，为**Java 5**及后续版本设计的依赖注入容器。

简介

Java企业应用开发社区在连接对象方面花了很大功夫。你的**Web**应用如何访问中间层服务？你的服务如何连接到登录用户和事务管理器？关于这个问题你会发现很多通用的和特定的解决方案。有一些方案依赖于模式，另一些则使用框架。所有这些方案都会不同程度地引入一些难于测试或者程式化代码重复的问题。你马上就会看到，**Guice** 在这方面是全世界做得最好的：非常容易进行单元测试，最大程度的灵活性和可维护性，以及最少的代码重复。

我们使用一个假想的、简单的例子来展示 **Guice** 优于其他一些你可能已经熟悉的经典方法的地方。下面的例子过于简单，尽管它展示了许多显而易见的优点，但其实它还远没有发挥出 **Guice** 的全部潜能。我们希望，随着你的应用开发的深入，**Guice** 的优越性也会更多地展现出来。

在这个例子中，一个客户对象依赖于一个服务接口。该服务接口可以提供任何服务，我们把它称为 **Service**。

```
public interface Service {
  void go();
}
```

对于这个服务接口，我们有一个缺省的实现，但客户对象不应该直接依赖于这个缺省实现。如果我们将来打算使用一个不同的服务实现，我们不希望回过头来修改所有的客户代码。

```
public class ServiceImpl implements Service {
  public void go() {
    ...
  }
}
```

我们还有一个可用于单元测试的伪服务对象。

```
public class MockService implements Service {
  private boolean gone = false;

  public void go() {
    gone = true;
  }

  public boolean isGone() {
    return gone;
  }
}
```

简单工厂模式

在发现依赖注入之前，最常用的是工厂模式。除了服务接口之外，你还有一个既可以向客户提供服务对象，也可以向测试程序传递伪服务对象的工厂类。在这里我们会将服务实现为一个单件对象，以便让示例尽量简化。

```
public class ServiceFactory {

  private ServiceFactory() {}

  private static Service instance = new ServiceImpl();

  public static Service getInstance() {
    return instance;
  }

  public static void setInstance(Service service) {
    instance = service;
  }
}
```

客户程序每次需要服务对象时就直接从工厂获取。

```
public class Client {

  public void go() {
    Service service = ServiceFactory.getInstance();
    service.go();
  }
}
```

客户程序足够简单。但客户程序的单元测试代码必须将一个伪服务对象传入工厂，同时要记得在测试后清理。在我们这个简单的例子里，这不算什么难事儿。但当你增加了越来越多的客户和服务代码后，所有这些伪代码和清理代码会让单元测试的开发一团糟。此外，如果你忘记在测试后清理，其他测试可能会得到与预期不符的结果。更糟的是，测试的成功与失败可能取决于他们被执行的顺序。

```
public void testClient() {
  Service previous = ServiceFactory.getInstance();
  try {
    final MockService mock = new MockService();
    ServiceFactory.setInstance(mock);
    Client client = new Client();
    client.go();
    assertTrue(mock.isGone());
  }
  finally {
    ServiceFactory.setInstance(previous);
  }
}
```

最后，注意服务工厂的**API**把我们限制在了单件这一种应用模式上。即便 `getInstance()` 可以返回多个实例， `setInstance()` 也会束缚我们的手脚。转换到非单件模式也意味着转换到了一套更复杂的**API**。

手工依赖注入

依赖注入模式的目标之一是使单元测试更简单。我们不需要特殊的框架就可以实践依赖注入模式。依靠手工编写代码，你可以得到该模式大约**80%**的好处。

当上例中的客户代码向工厂对象请求一个服务时，根据依赖注入模式，客户代码希望它所依赖的对象实例可以被传入自己。也就是说：不要调用我，我会调用你。

```
public class Client {

  private final Service service;

  public Client(Service service) {
    this.service = service;
  }

  public void go() {
```

```
    service.go();
  }
}
```

这让我们的单元测试简化了不少。我们可以只传入一个伪服务对象，在结束后也不需要多做什么。

```
public void testClient() {
  MockService mock = new MockService();
  Client client = new Client(mock);
  client.go();
  assertTrue(mock.isGone());
}
```

我们也可以精确地区分出客户代码依赖的**API**。

现在，我们如何连接客户和服务对象呢？手工实现依赖注入的时候，我们可以将所有依赖逻辑都移动到工厂类中。也就是说，我们还需要有一个工厂类来创建客户对象。

```
public static class ClientFactory {

  private ClientFactory() {}

  public static Client getInstance() {
    Service service = ServiceFactory.getInstance();
    return new Client(service);
  }
}
```

手工实现依赖注入需要的代码行数和简单工厂模式差不多。

用 Guice 实现依赖注入

手工为每一个服务与客户实现工厂类和依赖注入逻辑是一件很麻烦的事情。其他一些依赖注入框架甚至需要你显式将服务映射到每一个需要注入的地方。

Guice 希望在不牺牲可维护性的情况下去除所有这些程式化的代码。

使用 **Guice**，你只需要实现模块类。**Guice** 将一个绑定器传入你的模块，你的模块使用绑定器来连接接口和实现。以下模块代码告诉 **Guice** 将 `Service` 映射到单件模式的 `ServiceImpl`：

```
public class MyModule implements Module {
  public void configure(Binder binder) {
    binder.bind(Service.class)
      .to(ServiceImpl.class)
      .in(Scopes.SINGLETON);
  }
}
```

模块类告诉 **Guice** 我们想注入什么东西。那么，我们该如何告诉 **Guice** 我们想把它注入到哪里呢？使用 **Guice**，你可以使用 `@Inject` 标注你的构造器，方法或字段：

```
public class Client {

  private final Service service;

  @Inject
  public Client(Service service) {
    this.service = service;
  }

  public void go() {
    service.go();
```

```
  }
}
```

`@Inject` 标注可以清楚地告诉其他程序员你的类中哪些成员是被注入的。

为了让 **Guice** 向 `Client` 中注入，我们必须直接让 **Guice** 帮我们创建 `Client` 的实例，或者，其他类必须包含被注入的 `Client` 实例。

Guice vs. 手工依赖注入

如你所见，**Guice** 省去了写工厂类的麻烦。你不需要编写代码将客户连接到它们所依赖的对象。如果你忘了提供一个依赖关系，**Guice** 在启动时就会失败。**Guice** 也会自动处理循环依赖关系。

Guice 允许你通过声明指定对象的作用域。例如，你需要编写相同的代码将对象反复存入 `HttpSession`。

实际情况通常是，只有到了运行时，你才能知道具体要使用哪一个实现类。因此你需要元工厂类或服务定位器来增强你的工厂模式。**Guice** 用最少的代价解决了这些问题。

手工实现依赖注入时，你很容易退回到使用直接依赖的旧习惯，特别是当你对依赖注入的概念还不那么熟悉的时候。使用 **Guice** 可以避免这种问题，可以让你更容易地把事情做对。**Guice** 使你保持正确的方向。

更多的标注

只要有可能，**Guice** 就允许你使用标注来替代显式地绑定对象，以减少更多的程式化代码。回到我们的例子，如果你需要一个接口来简化单元测试，而你又不介意编译时的依赖，你可以直接从你的接口指向一个缺省的实现。

```
@ImplementedBy(ServiceImpl.class)
public interface Service {
  void go();
}
```

这时，如果客户需要一个 `Service` 对象，且 **Guice** 无法找到显式绑定，**Guice** 就会注入一个 `ServiceImpl` 的实例。

缺省情况下，**Guice** 每次都注入一个新的实例。如果你想指定不同的作用域规则，你也可以对实现类进行标注。

```
@Singleton
public class ServiceImpl implements Service {
  public void go() {
    ...
  }
}
```

架构概览

我们可以将 **Guice** 的架构分成两个不同的阶段：启动和运行。你在启动时创建一个注入器 `Injector`，在运行时用它来注入对象。

启动

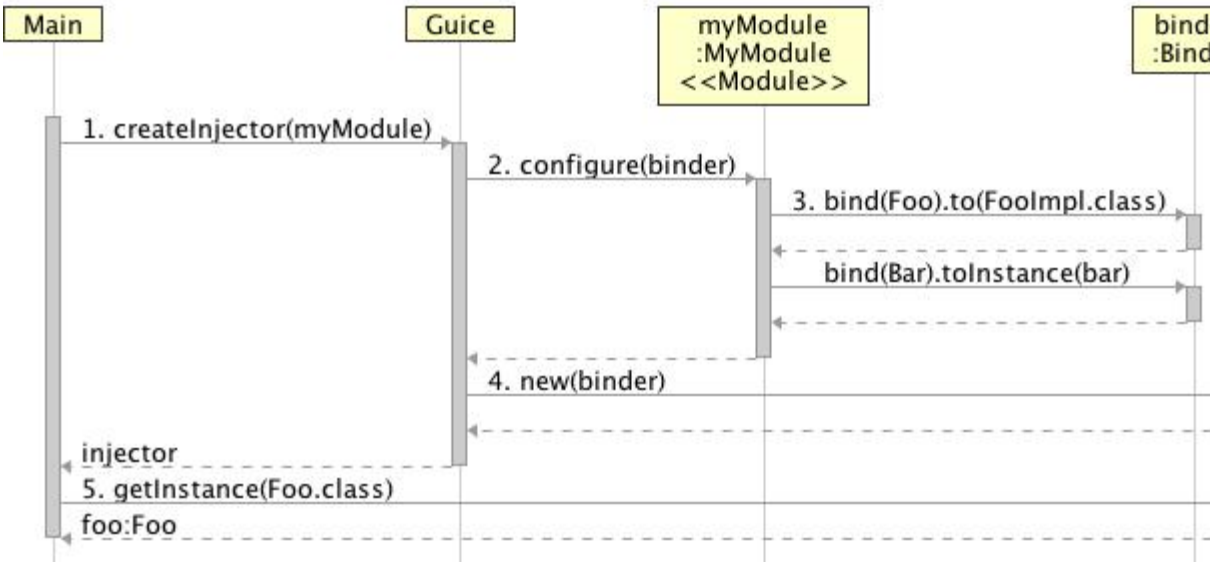
你通过实现 `Module` 来配置 **Guice**。你传给 **Guice** 一个模块对象，**Guice** 则将一个绑定器 `Binder` 对象传入你的模块，然后，你的模块使用绑定器来配置绑定。一个绑定通常包含一个从接口到具体实现的映射。例如：

```
public class MyModule implements Module {
  public void configure(Binder binder) {
    // Bind Foo to FooImpl. Guice will create a new
    // instance of FooImpl for every injection.
    binder.bind(Foo.class).to(FooImpl.class);

    // Bind Bar to an instance of Bar.
    Bar bar = new Bar();
    binder.bind(Bar.class).toInstance(bar);
  }
}
```

在这个阶段，**Guice** 会察看你告诉它的所有类，以及任何与这些类有关系的类，然后通知你是否有依赖关系的缺失。例如，在一个 **Struts 2** 应用中，**Guice** 知道你所有的动作类。**Guice** 会检查你的动作类以及它们依赖的所有类，如果有问题会及早报错。

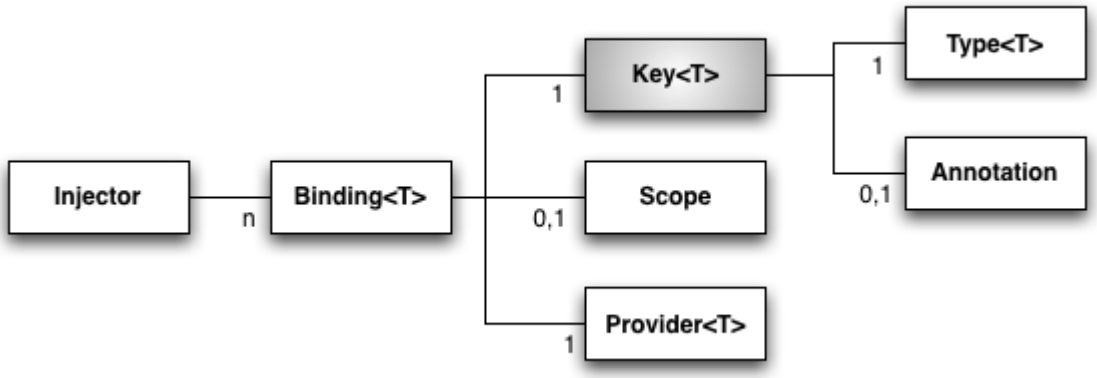
创建一个 `Injector` 涉及以下步骤：



1. 首先创建你的模块类实例，并将其传入 `Guice.createInjector()`.
2. **Guice** 创建一个绑定器 `Binder` 并将其传入你的模块。
3. 你的模块使用绑定器来定义绑定。
4. 基于你所定义的绑定，**Guice** 创建一个注入器 `Injector` 并将其返回给你。
5. 你使用注入器来注入对象。

运行

现在你可以使用第一阶段创建的注入器来注入对象并内省（**introspect**）我们的绑定了。**Guice** 的运行时模型由一个可管理一定数量绑定的注入器组成。



键 `Key` 唯一地确定每一个绑定。 `Key` 包含了客户代码所依赖的类型以及一个可选的标注。你可以使用标注来区分指向同一类型的多个绑定。 `Key` 的类型和标注对应于注入时的类型和标注。

每个绑定有一个提供者 **provider**，它提供所需类型的实例。你可以提供一个类，**Guice** 会帮你创建它的实例。你也可以给 **Guice** 一个你要绑定的类的实例。你还可以实现你自己的 **provider**，**Guice** 可以向其中注入依赖关系。

每个绑定还有一个可选的作用域。缺省情况下绑定没有作用域，**Guice** 为每一次注入创建一个新的对象。一个定制的作用域可以使你控制 **Guice** 是否创建新对象。例如，你可以为每一个 `HttpSession` 创建一个实例。

自举（Bootstrapping）你的应用

自举（***bootstrapping***）对于依赖注入非常重要。总是显式地向 `Injector` 索要依赖，这就将 **Guice** 用作了服务定位器，而不是一个依赖注入框架。

你的代码应该尽量少地和 `Injector` 直接打交道。相反，你应该通过注入一个根对象来自举你的应用。容器可以更进一步地将依赖注入根对象所依赖的对象，并如此迭代下去。最终，在理想情况下，你的应用中应该只有一个类知道 `Injector`，每个其他类都应该使用注入的依赖关系。

例如，一个诸如 **Struts 2** 的 **Web** 应用框架通过注入你的所有动作类来自举你的应用。你可以通过注入你的服务实现类来自举一个 **Web** 服务框架。

依赖注入是传染性的。如果你重构一个有大量静态方法的已有代码，你可能会觉得你正在试图拉扯一根没有尽头的线。这是好事情。它表明依赖注入正在帮助你改进代码的灵活性和可测试性。

如果重构工作太复杂，你不想一次性地整理完所有代码，你可以*暂时*将一个 `Injector` 的引用存入某个类的一个静态的字段，或是使用静态注入。这时，请清楚地命名包含该字段的类：比如 `InjectorHack` 和 `GodKillsAKittenEveryTimeYouUseMe`。记住你将来可能不得不为这些类提供伪测试类，你的单元测试则不得不手工安装一个注入器。记住，你将来需要清理这些代码。

绑定依赖关系

Guice 是如何知道要注入什么东西的呢？对启动器来说，一个包含了类型和可选的标注的 `Key` 唯一地指明了一个依赖关系。**Guice** 将 **key** 和实现之间的映射标记为一个 **Binding**。一个实现可以包含一个单独的对象，一个需要由 **Guice** 注入的类，或一个定制的 **provider**。

当注入依赖关系时，**Guice** 首先寻找显式绑定，即你通过绑定器 `Binder` 指明的绑定。`Binder` **API** 使用生成器（**Builder**）模式来创建一种领域相关的描述语言。根据约束适用方法的上下文的不同，不同方法返回不同的对象。

例如，为了将接口 `Service` 绑定到一个具体的实现 `ServiceImpl`，调用：

```
binder.bind(Service.class).to(ServiceImpl.class);
```

该绑定与下面的方法匹配：

```
@Inject
void injectService(Service service) {
  ...
}
```

注：与某些其他的框架相反，**Guice** 并没有给 **"setter"** 方法任何特殊待遇。不管方法有几个参数，只要该方法含有 `@Inject` 标注，**Guice** 就会实施注入，甚至对基类中实现的方法也不例外。

不要重复自己

对每个绑定不断地重复调用 **"binder"** 似乎有些乏味。**Guice** 提供了一个支持 `Module` 的类，名为 `AbstractModule`，它隐含地赋予你访问 `Binder` 的方法的权力。例如，我们可以用扩展 `AbstractModule` 类的方式改写上述绑定：

```
bind(Service.class).to(ServiceImpl.class);
```

在本手册的余下部分中我们会一直使用这样的语法。

标注绑定

如果你需要指向同一类型的多个绑定，你可以用标注来区分这些绑定。例如，将接口 `Service` 和标注 `@Blue` 绑定到具体的实现类 `BlueService` 的代码如下：

```
bind(Service.class)
  .annotatedWith(Blue.class)
  .to(BlueService.class);
```

这个绑定会匹配以下方法：

```
@Inject
void injectService(@Blue Service service) {
  ...
}
```

注意，标注 `@Inject` 出现在方法前，而绑定标注（如 `@Blue`）则出现在参数前。对构造函数也是如此。使用字段注入时，两种标注都直接应用于字段，如以下代码：

```
@Inject @Blue Service service;
```

创建绑定标注

刚才提到的标注 `@Blue` 是从哪里来的？你可以很容易地创建这种标注，但不幸的是，你必须使用略显复杂的标准语法：

```
/**
 * Indicates we want the blue version of a binding.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@BindingAnnotation
public @interface Blue {}
```

幸运的是，我们不需要理解这些代码，只要会用就可以了。对于好奇心强的朋友，下面是这些程式化代码的含义：

- `@Retention(RUNTIME)` 使得你的标注在运行时可见。
- `@Target({FIELD, PARAMETER})` 是对用户使用的说明；它不允许 `@Blue` 被用于方法、类型、局部变量和其他标注。
- `@BindingAnnotation` 是 **Guice** 特定的信号，表示你希望该标注被用于绑定标注。当用户将多于一个的绑定标注应用于同一个可注入元素时，**Guice** 会报错。

有属性的标注

如果你已经会写有属性的标注了，请跳到下一节。

你也可以绑定到标注实例，即，你可以有多个绑定指向同样的类型和标注类型，但每个绑定拥有不同的标注属性值。如果 **Guice** 找不到拥有特定属性值的标注实例，它会去找一个绑定到该标注类型的绑定。

例如，我们有一个绑定标注 `@Named`，它有一个字符串属性值。

```
@Retention(RUNTIME)
@Target({ FIELD, PARAMETER })
@BindingAnnotation
public @interface Named {
  String value();
}
```

如果我们希望绑定到 `@Named("Bob")`，我们首先需要一个 `Named` 的实现。我们的实现必须遵守关于 `Annotation` 的约定，特别是 `hashCode()` 和 `equals()` 的实现。

```
class NamedAnnotation implements Named {

  final String value;

  public NamedAnnotation(String value) {
    this.value = value;
  }

  public String value() {
    return this.value;
  }

  public int hashCode() {
    // This is specified in java.lang.Annotation.
    return 127 * "value".hashCode() ^ value.hashCode();
  }

  public boolean equals(Object o) {
    if (!(o instanceof Named))
      return false;
    Named other = (Named) o;
    return value.equals(other.value());
  }

  public String toString() {
    return "@" + Named.class.getName() + "(value=" + value + ")";
  }

  public Class<? extends Annotation> annotationType() {
    return Named.class;
  }
}
```

现在我们可以使用这个标注实现来创建一个指向 `@Named` 的绑定。

```
bind(Person.class)
  .annotatedWith(new NamedAnnotation("Bob"))
  .to(Bob.class);
```

与其它框架使用基于字符串的标识符相比，这显得有些繁琐，但记住，使用基于字符串的标识符，你根本无法这样做。而且，你会发现你可以大量复用已有的绑定标注。

因为通过名字标记一个绑定非常普遍，以至于 **Guice** 在 `com.google.inject.name` 中提供了一个十分有用的 `@Named` 的实现。

隐式绑定

正如我们在简介中看到的那样，你并不总是需要显式地声明绑定。如果缺少显式绑定，**Guice** 会试图注入并创建一个你所依赖的类的新实例。如果你依赖于一个接口，**Guice** 会寻找一个指向具体实现的 `@ImplementedBy` 标注。例如，下例中的代码显式绑定到一个具体的、可注入的名为 `Concrete` 的类。它的含义是，将 `Concrete` 绑定到 `Concrete`。这是显式的声明方式，但也有些冗余。

```
bind(Concrete.class);
```

删除上述绑定语句不会影响下面这个类的行为：

```
class Mixer {

  @Inject
  Mixer(Concrete concrete) {
    ...
  }
}
```

好吧，你自己来选择：显式的或简略的。无论何种方式，**Guice** 在遇到错误时都会生成有用的信息。

注入提供者

有时对于每次注入，客户代码需要某个依赖的多个实例。其它时候，客户可能不想在一开始就真地获取对象，而是等到注入后的某个时候再获取。对于任意绑定类型 `T`，你可以不直接注入 `T` 的实例，而是注入一个 `Provider<T>`，然后在需要的时候调用 `Provider<T>.get()`，例如：

```
@Inject
void injectAtm(Provider<Money> atm) {
  Money one = atm.get();
  Money two = atm.get();
  ...
}
```

正如你所看到的那样， `Provider` 接口简单得不能再简单了，它不会为简单的单元测试添加任何麻烦。

注入常数值

对于常数值，**Guice** 对以下几种类型做了特殊处理：

- 基本类型**(int, char, ...)**
- 基本封装类型**(Integer, Character, ...)**
- **Strings**
- **Enums**
- **Classes**

首先，当绑定到这些类型的常数值的时候，你不需要指定你要绑定到的类型。**Guice** 可以根据值判断类型。例如，一个绑定标注名为 `TheAnswer`：

```
bindConstant().annotatedWith(TheAnswer.class).to(42);
```

它的效果等价于：

```
bind(int.class).annotatedWith(TheAnswer.class).toInstance(42);
```

当需要注入这些类型的数值时，如果 **Guice** 找不到指向基本数据类型的显式绑定，它会找一个指向相应的封装类型的绑定，反之亦然。

转换字符串

如果 **Guice** 仍然无法找到一个上述类型的显式绑定，它会去找一个拥有相同绑定标注的常量 `String` 绑定，并试图将字符串转换到相应的值。例如：

```
bindConstant().annotatedWith(TheAnswer.class).to("42"); // String!
```

会匹配：

```
@Inject @TheAnswer int answer;
```

转换时，**Guice** 会用名字去查找枚举和类。**Guice** 在启动时转换一次，这意味着它提前做了类型检查。这个特性特别有用，例如，当绑定值来自一个属性文件的时候。

定制的提供者

有时你需要手工创建你自己的对象，而不是让 **Guice** 创建它们。例如，你可能不能为来自第三方的实现类添加 `@Inject` 标注。在这种情况下，你可以实现一个定制的 `Provider`。**Guice** 甚至可以注入你的提供者类。例如：

```
class WidgetProvider implements Provider<Widget> {

  final Service service;

  @Inject
  WidgetProvider(Service service) {
    this.service = service;
  }

  public Widget get() {
    return new Widget(service);
  }
}
```

你可以像这样把 `Widget` 绑定到 `WidgetProvider`：

```
bind(Widget.class).toProvider(WidgetProvider.class);
```

注入定制的提供者可以使 **Guice** 提前检查类型和依赖关系。定制的提供者可以在任意作用域中使用，而不依赖于他们所创建的类的作用域。缺省情况下，**Guice** 为每一次注入创建一个新的提供者实例。在上例中，如果每个 `Widget` 需要它自己的 `Service` 实例，我们的代码也没有问题。通过在工厂类上使用作用域标注，或为工厂类创建单独的绑定，你可以为定制的工厂指定不同的作用域。

示例：与 JNDI 集成

例如我们需要绑定从 **JNDI** 得到的对象。我们可以仿照下面的代码实现一个可复用的定制的提供者。注意我们注入了 **JNDI** Context：

```
package mypackage;

import com.google.inject.*;
import javax.naming.*;

class JndiProvider<T> implements Provider<T> {

  @Inject Context context;
  final String name;
  final Class<T> type;

  JndiProvider(Class<T> type, String name) {
    this.name = name;
    this.type = type;
  }

  public T get() {
    try {
      return type.cast(context.lookup(name));
    }
    catch (NamingException e) {
      throw new RuntimeException(e);
    }
  }

  /**
   * Creates a JNDI provider for the given
   * type and name.
   */
  static <T> Provider<T> fromJndi(
      Class<T> type, String name) {
    return new JndiProvider<T>(type, name);
  }
}
```

感谢泛型擦除（**generic type erasure**）技术。我们必须在运行时将依赖传入类中。你可以省略这一步，但在今后跟踪类型转换错误会比较棘手（当 **JNDI** 返回错误类型的对象的时候）。

我们可以使用定制的 JndiProvider 来将 DataSource 绑定到来自 **JNDI** 的一个对象：

```
    import com.google.inject.*;
    import static mypackage.JndiProvider.fromJndi;
    import javax.naming.*;
    import javax.sql.DataSource;

    ...

    // Bind Context to the default InitialContext.
    bind(Context.class).to(InitialContext.class);

    // Bind to DataSource from JNDI.
    bind(DataSource.class)
        .toProvider(fromJndi(DataSource.class, "..."));
```

限制绑定的作用域

缺省情况下，**Guice** 为每次注入创建一个新的对象。我们把它称为"无作用域"。你可以在配制绑定时指明作用域。例如，每次注入相同的实例：

```
bind(MySingleton.class).in(Scopes.SINGLETON);
```

另一种做法是，你可以在实现类中使用标注来指明作用域。**Guice** 缺省支持 `@Singleton`：

```
@Singleton
class MySingleton {
  ...
}
```

使用标注的方法对于隐式绑定也同样有效，但需要 **Guice** 来创建你的对象。另一方面，调用 `in()` 适用于几乎所有绑定类型（显然，绑定到一个单独的实例是个例外）并且会忽略已有的作用域标注。如果你不希望引入对于作用域实现的编译时依赖，`in()` 还可以接受标注。

可以使用 `Binder.bindScope()` 为定制的作用域指定标注。例如，对于标注 `@SessionScoped` 和一个 `Scope` 的实现 `ServletScopes.SESSION`：

```
binder.bindScope(SessionScoped.class, ServletScopes.SESSION);
```

创建作用域标注

用于指定作用域的标注必须：

- 有一个 `@Retention(RUNTIME)` 标注，从而使我们可以在运行时看到该标注。
- 有一个 `@Target({TYPE})` 标注。作用域标注只用于实现类。
- 有一个 `@ScopeAnnotation` 元标注。一个类只能使用一个此类标注。

例如：

```
/**
 * Scopes bindings to the current transaction.
 */
@Retention(RUNTIME)
@Target({TYPE})
@ScopeAnnotation
public @interface TransactionScoped {}
```

尽早加载绑定

Guice 可以等到你实际使用对象时再加载单件对象。这有助于开发，因为你的应用程序可以快速启动，只初始化你需要的对象。但是，有时你总是希望在启动时加载一个对象。你可以告诉 **Guice**，让它总是尽早加载一个单件对象，例如：

```
bind(StartupTask.class).asEagerSingleton();
```

我们经常在我们的应用程序中使用这个方法实现初始化逻辑。你可以通过在 **Guice** 必须首先初始化的单件对象上创建依赖关系来控制初始化顺序。

在不同作用域间注入

你可以安全地将来自大作用域的对象注入到来自小作用域或相同作用域的对象中。例如，你可以将一个作用域为 **HTTP** 会话的对象注入到作用域为 **HTTP** 请求的对象中。但是，向较大作用域的对象中注入就是另一件事了。例如，如果你把一个作用域为 **HTTP** 请求的对象注入到一个单件对象中，最好情况下，你会得到无法在 **HTTP** 请求中运行的错误信息，最坏情况下，你的单件对象会总是引用来自第一个 **HTTP** 请求的对象。在这些时候，你应该注入一个 `Provider<T>`，然后在需要的时候使用它从

较小的作用域中获取对象。这时，你必须确保，在 `T` 的作用域之外，永远不要调用这个提供者（例如，当目前没有 `HTTP` 请求且 `T` 的作用域为 `HTTP` 请求的时候）。

开发阶段

Guice 明白你的应用开发需要经历不同的阶段。你可以在创建容器时告诉它应用程序运行在哪一个阶段。**Guice** 目前支持"开发"和"产品"两个阶段。我们发现测试通常属于其中某一个阶段。

在开发阶段，**Guice** 会根据需要加载单件对象。这样，你的应用程序可以快速启动，只加载你正在测试的部分。

在产品阶段，**Guice** 会在启动时加载全部单件对象。这帮助你尽早捕获错误，提前优化性能。

你的模块也可以使用方法拦截和其他基于当前阶段的绑定。例如，一个拦截器可能会在开发阶段检查你是否在作用域之外使用对象。

拦截方法

Guice 使用 **AOP Alliance API** 支持简单的方法拦截。你可以在模块中使用 `Binder` 绑定拦截器。例如，对标注有 `@Transactional` 的方法应用事务拦截器：

```
import static com.google.inject.matcher.Matchers.*;

...

binder.bindInterceptor(
  any(),                                // Match classes.
  annotatedWith(Transactional.class), // Match methods.
  new TransactionInterceptor()        // The interceptor.
);
```

尽量让匹配代码多做些过滤工作，而不是在拦截器中过滤。因为匹配代码只在启动时运行一次。

静态注入

静态字段和方法会增加测试和复用的难度，但有的时候你唯一的选择就是保留一个 `Injector` 的静态引用。

在这些情况下，**Guice** 支持注入可访问性较少的静态方法。例如，**HTTP** 会话对象经常需要被串行化，以支持复制机制。但是，如果你的会话对象依赖于一个作用域为容器生命周期的对象，该怎么办呢？我们可以保留一个该对象的临时引用，但在反串行化的时候，我们该如何再次找到该对象呢？

我们发现更实用的解决方案是使用静态注入：

```
@SessionScoped
class User {

  @Inject
  static AuthorizationService authorizationService;
  ...
}
```

Guice 从不自动实施静态注入。你必须使用 `Binder` 显式请求 `Injector` 在启动后注入你的静态成员：

```
binder.requestStaticInjection(User.class);
```

静态注入是一个很难避免的祸害，它会使测试难度加大。如果有办法避开它，你多半会很高兴的。

可选注入

有时你的代码应该在无论绑定是否存在的时候都能工作。在这些情况下，你可以使用 `@Inject(optional=true)`，**Guice** 会在有绑定可用时，用一个绑定实现覆盖你的缺省实现。例如：

```
@Inject(optional=true) Formatter formatter = new DefaultFormatter();
```

如果谁为 `Formatter` 创建了一个绑定，**Guice** 会基于该绑定注入实例。否则，如果 `Formatter` 不能被注入**(**参见*隐式绑定*)，**Guice** 会忽略可选成员。

可选注入只能应用于字段和方法，而不能用于构造函数。对于方法，如果一个参数的绑定找不到，**Guice** 就不会注入该方法，即便其他参数的绑定是可用的。

绑定到字符串

只要有可能，我们就尽量避免使用字符串，因为它们容易被错误拼写，对工具不友好，等等。但使用字符串而不是创建定制的标注对于"快而脏"的代码来说仍是有用的。在这些情况下，**Guice** 提供了 `@Named` 和 `Names`。例如，一个到字符串名字的绑定：

```
import static com.google.inject.name.Names.*;

...

bind(named("bob")).to(10);
```

会匹配下面的注入点：

```
@Inject @Named("bob") int score;
```

Struts 2支持

要在 **Struts 2.0.6** 或更高版本中安装 **Guice Struts 2** 插件，只要将 `guice-struts2-plugin-1.0.jar` 包含在你的 **Web** 应用的 **classpath** 中，并在 `struts.xml` 文件中选择 **Guice** 作为你的 `ObjectFactory` 实现即可：

```
<constant name="struts.objectFactory" value="guice" />
```

Guice 会注入所有你的 **Struts 2** 对象，包括动作和拦截器。你甚至可以设置动作类的作用域。你也可以在你的 `struts.xml` 文件中指定 **Guice** 的 `Module`：

```
<constant name="guice.module" value="mypackage.MyModule"/>
```

如果你的所有绑定都是隐式的，你就根本不用定义模块了。

一个计数器的例子

例如，我们试图统计一个会话中的请求数目。定义一个在会话中存活的 `Counter` 对象：

```
@SessionScoped
public class Counter {

  int count = 0;

  /** Increments the count and returns the new value. */
  public synchronized int increment() {
    return count++;
  }
}
```

接下来，我们可以将我们的计数器注入到动作中：

```
public class Count {

  final Counter counter;

  @Inject
  public Count(Counter counter) {
    this.counter = counter;
  }

  public String execute() {
    return SUCCESS;
  }

  public int getCount() {
    return counter.increment();
  }
}
```

然后在 **struts.xml** 文件中为动作类创建映射：

```
<action name="Count"
    class="mypackage.Count">
  <result>/WEB-INF/Counter.jsp</result>
</action>
```

以及一个用于显示结果的 **JSP** 页面：

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
  <body>
    <h1>Counter Example</h1>
    <h3><b>Hits in this session:</b>
      <s:property value="count"/></h3>
  </body>
</html>
```

我们实际上把这个例子做得比需求更复杂，以便展示更多的概念。在现实中，我们不需要使用单独的 `Counter` 对象，只要把 `@SessionScoped` 直接应用于我们的动作类即可。

JMX 集成

参见 com.google.inject.tools.jmx.

附录：注入器如何解决注入请求

注入器解决注入请求的过程依赖于已有的绑定和相关类型中的标注。这里是关于如何解决注入请求的一个概要描述：

1. 观察被注入元素的 **Java** 类型和可选的“绑定标注”。如果类型是 `com.google.inject.Provider<T>`，就使用类型 `T` 解决注入请求。对于（类型，标注）对，寻找一个绑定。如果找不到，则跳到步骤**4**。
2. 沿着绑定链检查。如果该绑定连接到另一个绑定，则沿着这条边继续检查，直到到达一个没有连接到任何后续绑定的绑定为止。现在我们就为该注入请求找到了最明确的显式绑定。
3. 如果绑定指明一个实例或一个 `Provider` 实例，所有事情都做完了；使用这个实例来满足请求即可。
4. 此时，如果注入请求使用了标注类型或值，我们就报告错误。

5. 否则，检查绑定的 **Java** 类型；如果找到了 `@ImplementedBy` 标注，就实例化该类型。 如果找到了 `@ProvidedBy` 标注，就实例化提供者类并用它来获取想要的对象。否则试图实例化类型本身。