[Google-Guice入门介绍](http://blog.csdn.net/derekjiang/article/details/7231490)

标签： [annotations](http://www.csdn.net/tag/annotations)[class](http://www.csdn.net/tag/class)[interface](http://www.csdn.net/tag/interface)[module](http://www.csdn.net/tag/module)[exception](http://www.csdn.net/tag/exception)[string](http://www.csdn.net/tag/string)

2012-02-03 15:08 25610人阅读 [评论](http://blog.csdn.net/derekjiang/article/details/7231490#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/derekjiang/article/details/7231490#report)

 分类：

Java（37）

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/derekjiang/article/details/7231490)

**一. 概述**

Guice是一个轻量级的DI框架。本文对Guice的基本用法作以介绍。

本文的所有例子基于Guice 3.0

本文的很多代码来源于Guice主页：http://code.google.com/p/google-guice/wiki/GettingStarted

考虑到是入门介绍，本文中并未涉及到AOP相关内容，如有需要还请参考上面链接。

**二. 举例说明Guice的用法**

Guice本身只是一个轻量级的DI框架，首先我们通过一个例子来看看怎么使用Guice。

首先有一个需要被实现的接口：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **interface** BillingService {
3. /\*\*
4. \* Attempts to charge the order to the credit card. Both successful and
5. \* failed transactions will be recorded.
6. \*
7. \* @return a receipt of the transaction. If the charge was successful, the
8. \*      receipt will be successful. Otherwise, the receipt will contain a
9. \*      decline note describing why the charge failed.
10. \*/
11. Receipt chargeOrder(PizzaOrder order, CreditCard creditCard);
12. }

然后，有一个实现该接口的实现类：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **class** RealBillingService **implements** BillingService {
2. **private** **final** CreditCardProcessor processor;
3. **private** **final** TransactionLog transactionLog;
5. @Inject
6. RealBillingService(CreditCardProcessor processor,
7. TransactionLog transactionLog) {
8. **this**.processor = processor;
9. **this**.transactionLog = transactionLog;
10. }
12. @Override
13. **public** Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
14. ...
15. }
16. }

现在接口有了，实现类也有了，接下来就是如何将接口和实现类关联的问题了，在Guice中需要定义Module来进行关联

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** BillingModule **extends** AbstractModule {
2. @Override
3. **protected** **void** configure() {
5. /\*
6. \* This tells Guice that whenever it sees a dependency on a TransactionLog,
7. \* it should satisfy the dependency using a DatabaseTransactionLog.
8. \*/
9. bind(TransactionLog.**class**).to(DatabaseTransactionLog.**class**);
11. /\*
12. \* Similarly, this binding tells Guice that when CreditCardProcessor is used in
13. \* a dependency, that should be satisfied with a PaypalCreditCardProcessor.
14. \*/
15. bind(CreditCardProcessor.**class**).to(PaypalCreditCardProcessor.**class**);
16. }
17. }

好了，现在万事俱备，就让我们一起看看怎么使用Guice进行依赖注入吧：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **static** **void** main(String[] args) {
2. /\*
3. \* Guice.createInjector() takes your Modules, and returns a new Injector
4. \* instance. Most applications will call this method exactly once, in their
5. \* main() method.
6. \*/
7. Injector injector = Guice.createInjector(**new** BillingModule());
9. /\*
10. \* Now that we've got the injector, we can build objects.
11. \*/
12. RealBillingService billingService = injector.getInstance(RealBillingService.**class**);
13. ...
14. }

以上就是使用Guice的一个完整的例子，很简单吧，不需要繁琐的配置，只需要定义一个Module来表述接口和实现类，以及父类和子类之间的关联关系的绑定。本文不对比guice和[**spring**](http://lib.csdn.net/base/javaee)，只是单纯介绍Guice的用法。

**三. 绑定方式的介绍**

从上面我们可以看出，其实对于Guice而言，程序员所要做的，只是创建一个代表关联关系的Module，然后使用这个Module即可得到对应关联的对象。因此，主要的问题其实就是在如何关联实现类和接口（子类和父类）。

**1. 在自定义的Module类中进行绑定**

**1.1 在configure方法中绑定**

**1.1.1 链式绑定**

链式绑定是最简单，最直接，也是使用最多的绑定方式。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **protected** **void** configure() {
2. bind(TransactionLog.**class**).to(DatabaseTransactionLog.**class**);
3. }

就是直接把一种类型的class对象绑定到另外一种类型的class对象，这样，当外界获取TransactionLog时，其实返回的就是一个DatabaseTransactionLog对象。当然，链式绑定也可以串起来，如：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **protected** **void** configure() {
2. bind(TransactionLog.**class**).to(DatabaseTransactionLog.**class**);
3. bind(DatabaseTransactionLog.**class**).to(MySqlDatabaseTransactionLog.**class**);
4. }

这样，当外界请求TransactionLog时，其实返回的就会是一个MySqlDatabaseTransactionLog对象。

**1.1.2 注解（Annotations）绑定**

链式绑定针对于同样的类型都绑定到同一种目标类型时，非常好用，但是对于一个接口有多种实现的时候，链式绑定就不好区分该用哪种实现了。可以把Annotations绑定方式看作是链式绑定的一种扩展，专门用来解决这种同一个接口有多种实现的问题。Annotations绑定又可以分为两种，一种是需要自己写Annotations，另外一种则简化了一些。

**1.1.2.1 自己写Annotations的方式**

首先，写一个注解

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **import** com.google.inject.BindingAnnotation;
2. **import** java.lang.annotation.Target;
3. **import** java.lang.annotation.Retention;
4. **import** **static** java.lang.annotation.RetentionPolicy.RUNTIME;
5. **import** **static** java.lang.annotation.ElementType.PARAMETER;
6. **import** **static** java.lang.annotation.ElementType.FIELD;
7. **import** **static** java.lang.annotation.ElementType.METHOD;
9. @BindingAnnotation @Target({ FIELD, PARAMETER, METHOD }) @Retention(RUNTIME)
10. **public** **@interface** PayPal {}

然后，使用这个注解去修饰目标字段或参数，如：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** RealBillingService **implements** BillingService {
3. @Inject
4. **public** RealBillingService(@PayPal CreditCardProcessor processor,
5. TransactionLog transactionLog) {
6. ...
7. }
8. ｝

或

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** RealBillingService **implements** BillingService {
2. @Inject
3. @Www
4. **private** CreditCardProcessor processor;
5. ...
6. }

最后，在我们进行链式绑定时，就可以区分一个接口的不同实现了，如：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(CreditCardProcessor.**class**)
2. .annotatedWith(PayPal.**class**)
3. .to(PayPalCreditCardProcessor.**class**);

这样，被Annotations PayPal[?](http://wiki.corp.yahoo.com/edit/Main/PayPal?topicparent=Main.Google-GuiceStudy;nowysiwyg=1)修饰的CreditCardProcessor就会被绑定到目标实现类PayPalCreditCardProcessor。如果有其他的实现类，则可把用不同Annotations修饰的CreditCardProcessor绑定到不同的实现类

**1.1.2.2 使用@Named的方式**

使用@Named的方式和上面自己写Annotation的方式很类似，只不过做了相应的简化，不再需要自己去写Annotation了。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** RealBillingService **implements** BillingService {
3. @Inject
4. **public** RealBillingService(@Named("Checkout") CreditCardProcessor processor,
5. TransactionLog transactionLog) {
6. ...
7. }

直接使用@Named修饰要注入的目标，并起个名字，下面就可以把用这个名字的注解修饰的接口绑定到目标实现类了

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(CreditCardProcessor.**class**)
2. .annotatedWith(Names.named("Checkout"))
3. .to(CheckoutCreditCardProcessor.**class**);

**1.1.3 实例绑定**

上面介绍的链式绑定是把接口的class对象绑定到实现类的class对象，而实例绑定则可以看作是链式绑定的一种特例，它直接把一个实例对象绑定到它的class对象上。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(String.**class**)
2. .annotatedWith(Names.named("JDBC URL"))
3. .toInstance("jdbc:mysql://localhost/pizza");
4. bind(Integer.**class**)
5. .annotatedWith(Names.named("login timeout seconds"))
6. .toInstance(10);

需要注意的是，实例绑定要求对象不能包含对自己的引用。并且，尽量不要对那种创建实例比较复杂的类使用实例绑定，否则会让应用启动变慢

**1.1.4 Provider绑定**

在下面会介绍基于@Provides方法的绑定。其实Provider绑定是基于@Provides方法绑定的后续发展，所以应该在介绍完基于@Provides方法绑定之后再来介绍，不过因为Provider绑定也是在configure方法中完成的，而本文又是按照绑定的位置来组织的，因为就把Provider绑定放在这了，希望大家先跳到后面看过基于@Provides方法的绑定再回来看这段。

在使用基于@Provides方法绑定的过程中，如果方法中创建对象的过程很复杂，我们就会考虑，是不是可以把它独立出来，形成一个专门作用的类。Guice提供了一个接口：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **interface** Provider {
2. T get();
3. }

实现这个接口，我们就会得到专门为了创建相应类型对象所需的类：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** DatabaseTransactionLogProvider **implements** Provider {
2. **private** **final** Connection connection;
4. @Inject
5. **public** DatabaseTransactionLogProvider(Connection connection) {
6. **this**.connection = connection;
7. }
9. **public** TransactionLog get() {
10. DatabaseTransactionLog transactionLog = **new** DatabaseTransactionLog();
11. transactionLog.setConnection(connection);
12. **return** transactionLog;
13. }
14. }

这样以来，我们就可以在configure方法中，使用toProvider方法来把一种类型绑定到具体的Provider类。当需要相应类型的对象时，Provider类就会调用其get方法获取所需的对象。

其实，个人感觉在configure方法中使用Provider绑定和直接写@Provides方法所实现的功能是没有差别的，不过使用Provider绑定会使代码更清晰。而且当提供对象的方法中也需要有其他类型的依赖注入时，使用Provider绑定会是更好的选择。

**1.1.5 无目标绑定**

无目标绑定是链接绑定的一种特例，在绑定的过程中不指明目标，如：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(MyConcreteClass.**class**);
2. bind(AnotherConcreteClass.**class**).in(Singleton.**class**);

如果使用注解绑定的话，就不能用无目标绑定，必须指定目标，即使目标是它自己。如：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(MyConcreteClass.**class**).annotatedWith(Names.named("foo")).to(MyConcreteClass.**class**);
2. bind(AnotherConcreteClass.**class**).annotatedWith(Names.named("foo")).to(AnotherConcreteClass.**class**).in(Singleton.**class**);

无目标绑定，主要是用于与被@ImplementedBy 或者 @ProvidedBy修饰的类型一起用。如果无目标绑定的类型不是被@ImplementedBy 或者 @ProvidedBy修饰的话，该类型一定不能只提供有参数的构造函数，要么不提供构造函数，要么提供的构造函数中必须有无参构造函数。因为guice会默认去调用该类型的无参构造函数。

**1.1.6 指定构造函数绑定**

在configure方法中，将一种类型绑定到另外一种类型的过程中，指定目标类型用那种构造函数生成对象。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** BillingModule **extends** AbstractModule {
2. @Override
3. **protected** **void** configure() {
4. **try** {
5. bind(TransactionLog.**class**).toConstructor(
6. DatabaseTransactionLog.**class**.getConstructor(DatabaseConnection.**class**));
7. } **catch** (NoSuchMethodException e) {
8. addError(e);
9. }
10. }
11. }

这种绑定方式主要用于不方便用注解@Inject修饰目标类型的构造函数的时候。比如说目标类型是第三方提供的类型，或者说目标类型中有多个构造函数，并且可能会在不同情况采用不同的构造函数。

**1.1.7 Built-in绑定**

Built-in绑定指的是不用程序员去指定，Guice会自动去做的绑定。目前，Guice所支持的Built-in绑定只有对java.util.logging.Logger的绑定。个人感觉，所谓的Built-in绑定，只是在比较普遍的东西上为大家带来方便的一种做法。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. @Singleton
2. **public** **class** ConsoleTransactionLog **implements** TransactionLog {
4. **private** **final** Logger logger;
6. @Inject
7. **public** ConsoleTransactionLog(Logger logger) {
8. **this**.logger = logger;
9. }
11. **public** **void** logConnectException(UnreachableException e) {
12. /\* the message is logged to the "ConsoleTransacitonLog" logger \*/
13. logger.warning("Connect exception failed, " + e.getMessage());
14. }

**1.2 在@Provides方法中进行绑定**

当你只是需要在需要的时候，产生相应类型的对象的话，@Provides Methods是个不错的选择。方法返回的类型就是要绑定的类型。这样当需要创建一个该类型的对象时，该provide方法会被调用，从而得到一个该类型的对象。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** BillingModule **extends** AbstractModule {
2. @Override
3. **protected** **void** configure() {
4. ...
5. }
7. @Provides
8. TransactionLog provideTransactionLog() {
9. DatabaseTransactionLog transactionLog = **new** DatabaseTransactionLog();
10. transactionLog.setJdbcUrl("jdbc:mysql://localhost/pizza");
11. transactionLog.setThreadPoolSize(30);
12. **return** transactionLog;
13. }
14. }

需要注意的是Provide方法必须被@Provides所修饰。同时，@Provides方法绑定方式是可以和上面提到的注解绑定混合使用的，如：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. @Provides @PayPal
2. CreditCardProcessor providePayPalCreditCardProcessor(
3. @Named("PayPal API key") String apiKey) {
4. PayPalCreditCardProcessor processor = **new** PayPalCreditCardProcessor();
5. processor.setApiKey(apiKey);
6. **return** processor;
7. }

这样一来，只有被@PayPal修饰的CreditCardProcessor对象才会使用provide方法来创建对象，同时

**2. 在父类型中进行绑定**

**2.1 @ImplementedBy**

在定义父类型的时候，直接指定子类型的方式。  
这种方式其实和链式绑定的效果是完全一样的，只是声明绑定的位置不同。  
和链式绑定不同的是它们的优先级，@ImplementedBy实现的是一种default绑定，当同时存在@ImplementedBy和链式绑定时，链式绑定起作用。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. @ImplementedBy(PayPalCreditCardProcessor.**class**)
2. **public** **interface** CreditCardProcessor {
3. ChargeResult charge(String amount, CreditCard creditCard)
4. **throws** UnreachableException;
5. }

等价于：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(CreditCardProcessor.**class**).to(PayPalCreditCardProcessor.**class**);

**2.2 @ProvidedBy**

在定义类型的时候直接指定子类型的Provider类。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. @ProvidedBy(DatabaseTransactionLogProvider.**class**)
2. **public** **interface** TransactionLog {
3. **void** logConnectException(UnreachableException e);
4. **void** logChargeResult(ChargeResult result);
5. }

等价于:

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(TransactionLog.**class**)
2. .toProvider(DatabaseTransactionLogProvider.**class**);

并且，和@ImplementedBy类似，@ProvidedBy的优先级也比较低，是一种默认实现，当@ProvidedBy和toProvider函数两种绑定方式并存时，后者有效。

**3. 在子类型中进行注入**

**3.1 构造函数注入**

在构造函数绑定中，Guice要求目标类型要么有无参构造函数，要么有被@Inject注解修饰的构造函数。这样，当需要创建该类型的对象时，Guice可以帮助进行相应的绑定，从而生成对象。在Guice创建对象的过程中，其实就是调用该类型被@Inject注解修饰的构造函数，如果没要@Inject注解修饰的构造函数，则调用无参构造函数。在使用构造函数绑定时，无需再在Module中定义任何绑定关系。

这里需要注意的是，Guice在创建对象的过程中，无法初始化该类型的内部类（除非内部类有static修饰符），因为内部类会有隐含的对外部类的引用，Guice无法处理。

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** PayPalCreditCardProcessor **implements** CreditCardProcessor {
2. **private** **final** String apiKey;
4. @Inject
5. **public** PayPalCreditCardProcessor(@Named("PayPal API key") String apiKey) {
6. **this**.apiKey = apiKey;
7. }

**3.2 属性注入**

属性绑定的目的是告诉Guice，当创建该类型的对象时，哪些属性也需要进行依赖注入。用一个例子来看：  
首先，有一个接口：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **package** guice.test;
3. **import** com.google.inject.ImplementedBy;;
5. @ImplementedBy (SayHello.**class**)
6. **public** **interface** Talk {
7. **public** **void** sayHello();
8. }

该接口指明了它的实现类SayHello

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **package** guice.test;
3. **public** **class** SayHello **implements** Talk{
4. @Override
5. **public** **void** sayHello() {
6. System.out.println("Say Hello!");
8. }
9. }

接下来就是属性注入的例子：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **package** guice.test;
3. **import** com.google.inject.Inject;
5. **public** **class** FieldDI {
6. @Inject
7. **private** Talk bs;
9. **public** Talk getBs() {
10. **return** bs;
11. }
12. }

这里面，指明熟悉Talk类型的bs将会被注入。使用的例子是：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **package** guice.test;
3. **import** com.google.inject.Guice;
4. **import** com.google.inject.Injector;
6. **public** **class** Test {
7. **public** **static** **void** main(String[] args) {
9. Injector injector = Guice.createInjector(**new** BillingModule());

12. FieldDI fdi = injector.getInstance(FieldDI.**class**);
13. fdi.getBs().sayHello();
14. }
15. }

如果我们没有用@Inject修饰Talk bs的话，就会得到如下错误：

Exception in thread "main" java.lang.NullPointerException

如果我们用@Inject修饰Talk bs了，但是Talk本身没有被@ImplementedBy修饰的话，会得到如下错误：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. Exception in thread "main" com.google.inject.ConfigurationException: Guice configuration errors:
3. 1) No implementation **for** guice.test.Talk was bound.
4. **while** locating guice.test.Talk
5. **for** field at guice.test.FieldDI.bs(FieldDI.java:5)
6. **while** locating guice.test.FieldDI
8. 1 error
9. at com.google.inject.internal.InjectorImpl.getProvider(InjectorImpl.java:1004)
10. at com.google.inject.internal.InjectorImpl.getProvider(InjectorImpl.java:961)
11. at com.google.inject.internal.InjectorImpl.getInstance(InjectorImpl.java:1013)
12. at guice.test.Test.main(Test.java:24)

另外，属性注入的一种特例是注入provider。如：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **class** RealBillingService **implements** BillingService {
2. **private** **final** Provider processorProvider;
3. **private** **final** Provider transactionLogProvider;
5. @Inject
6. **public** RealBillingService(Provider processorProvider,
7. Provider transactionLogProvider) {
8. **this**.processorProvider = processorProvider;
9. **this**.transactionLogProvider = transactionLogProvider;
10. }
12. **public** Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
13. CreditCardProcessor processor = processorProvider.get();
14. TransactionLog transactionLog = transactionLogProvider.get();
16. /\* use the processor and transaction log here \*/
17. }
18. }

其中，Provider的定义如下：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **public** **interface** Provider {
2. T get();
3. }

**3.3 Setter方法注入**

有了上面的基础我们再来看Setter注入就非常简单了，只不过在setter方法上增加一个@Inject注解而已。 还是上面的例子，只是有一点修改：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. **package** guice.test;
3. **import** com.google.inject.Inject;
5. **public** **class** FieldDI {
7. @Inject
8. **public** **void** setBs(Talk bs) {
9. **this**.bs = bs;
10. }
12. **private** Talk bs;
14. **public** Talk getBs() {
15. **return** bs;
16. }
17. }

**四. 对象产生的Scopes**

在默认情况下，每次通过Guice去请求对象时，都会得到一个新的对象，这种行为是通过Scopes去配置的。  
Scope的配置使得我们重用对象的需求变得可能。目前，Guice支持的Scope有@Singleton, @SessionScoped, @RequestScoped。  
同样，程序员也可以自定义自己的Scope，本文不涉及自定义scope，如果有兴趣，请参考：http://code.google.com/p/google-guice/wiki/CustomScopes

下面我们就以@Singleton举例说明怎么来告诉Guice我们要以@Singleton的方式产生对象：1. 在定义子类型时声明

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. @Singleton
2. **public** **class** InMemoryTransactionLog **implements** TransactionLog {
3. /\* everything here should be threadsafe! \*/
4. }

2.在module的configure方法中做绑定时声明

bind(TransactionLog.class).to(InMemoryTransactionLog.class).in(Singleton.class);

3.在module的Provides方法里声明

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. @Provides @Singleton
2. TransactionLog provideTransactionLog() {
3. ...
4. }

这里有一点需要注意的是，如果发生下面的情况：

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(Bar.**class**).to(Applebees.**class**).in(Singleton.**class**);
2. bind(Grill.**class**).to(Applebees.**class**).in(Singleton.**class**);

这样一共会生成2个Applebees对象，一个给Bar用，一个给Grill用。如果在上面的配置的情况下，还有下面的配置，

**[java]** [view plain](http://blog.csdn.net/derekjiang/article/details/7231490) [copy](http://blog.csdn.net/derekjiang/article/details/7231490)

1. bind(Applebees.**class**).in(Singleton.**class**);

这样一来，Bar和Grill就会共享同样的对象了。