Google Guice详解

Guice是轻量级的java 依赖注入框架（IoC），与Sping相比速度快，无外部配置，完全基于annotation特性，支持重构和代码静态检查。Guice是类型安全的，能够对构造函数、属性、方法（包含任意个参数的任意方法，而不仅仅是setter方法）进行注入（DI）。

# IoC，控制反转

IoC, Inversion of Control，是一种设计思想，在Java开发过程中，IoC意味着将设计好的对象交给容器控制，而不是在对象内部直接控制，也就是为相互依赖的组件提供抽象，将依赖（底层模块）对象的获得交给第三方系统来控制，即依赖对象不再被依赖模块的类中直接通过new来获取。

IoC一般分为两种类型，依赖注入（Dependency Injection，简称DI）和依赖查找。依赖注入就是将依赖对象的创建和绑定转移到被依赖对象类的外部来实现，其提供一种机制，将需要依赖（低层模块）的对象的引用传递给被依赖（高层模块）对象。

在大型项目中，相互依赖的组件比较多，如果通过手动来创建和注入依赖的话，显然效率很低，因此引入IoC容器，本质上DI框架，包含：

1. 动态创建，注入依赖对象
2. 管理对象声明周期
3. 映射依赖关系

例如Spring BeanFactory就是Ioc容器。

DI，依赖注入的方式有以下三种：

1. 构造函数注入

在构造函数上添加@Inject来达到自动注入的目的，构造函数注入的好处是保证只有一个地方来完成属性注入，这样可以确保在构造函数中完成一些初始化工作。缺点是类的实例化与参数绑定了，限制了实例化类的方式。

1. 字段注入

Guice可以通过以@Inject注解字段进行注入，这是最简洁的方式。

1. 函数注入（setter）

在setter方法上增加@Inject注解

1. 基本属性注入，

通过在接口上添加注解，@ImplemetedBy来自动实现接口与实现类的绑定

本质上是将一个接口绑定到具体的类中，这样客户端就不用关心具体的实现，而只需要获取相应的接口完成其服务即可。

# DIP，依赖倒置原则

类A直接依赖B，假如要将A改为依赖C，则必须修改类A的代码来达成，这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作，假如修改类A，会给程序带来不必要的风险。

因此引入依赖倒置原则：高层模块不依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节，细节依赖抽象。

将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

   依赖倒置原则基于这样一个事实：相对于细节的多变性，抽象的东西要稳定的多。以抽象为基础搭建起来的架构比以细节为基础搭建起来的架构要稳定的多。在java中，抽象指的是接口或者抽象类，细节就是具体的实现类，使用接口或者抽象类的目的是制定好规范和契约，而不去涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成。

依赖倒置原则的核心思想是面向接口编程，用一个例子来说明面向接口编程比相对于面向实现编程好在什么地方。场景是这样的，母亲给孩子讲故事，只要给她一本书，她就可以照着书给孩子讲故事了。代码如下：

[java] view plaincopy

class Book{

public String getContent(){

return "很久很久以前有一个阿拉伯的故事……";

}

}

class Mother{

public void narrate(Book book){

System.out.println("妈妈开始讲故事");

System.out.println(book.getContent());

}

}

public class Client{

public static void main(String[] args){

Mother mother = new Mother();

mother.narrate(new Book());

}

}

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

运行良好，假如有一天，需求变成这样：不是给书而是给一份报纸，让这位母亲讲一下报纸上的故事，报纸的代码如下：

[java] view plaincopy

class Newspaper{

public String getContent(){

return "林书豪38+7领导尼克斯击败湖人……";

}

}

这位母亲却办不到，因为她居然不会读报纸上的故事，这太荒唐了，只是将书换成报纸，居然必须要修改Mother才能读。假如以后需求换成杂志呢？换成网页呢？还要不断地修改Mother，这显然不是好的设计。原因就是Mother与Book之间的耦合性太高了，必须降低他们之间的耦合度才行。

我们引入一个抽象的接口IReader。读物，只要是带字的都属于读物：

[java] view plaincopy

interface IReader{

public String getContent();

}

Mother类与接口IReader发生依赖关系，而Book和Newspaper都属于读物的范畴，他们各自都去实现IReader接口，这样就符合依赖倒置原则了，代码修改为：

[java] view plaincopy

class Newspaper implements IReader {

public String getContent(){

return "林书豪17+9助尼克斯击败老鹰……";

}

}

class Book implements IReader{

public String getContent(){

return "很久很久以前有一个阿拉伯的故事……";

}

}

class Mother{

public void narrate(IReader reader){

System.out.println("妈妈开始讲故事");

System.out.println(reader.getContent());

}

}

public class Client{

public static void main(String[] args){

Mother mother = new Mother();

mother.narrate(new Book());

mother.narrate(new Newspaper());

}

}

运行结果：

妈妈开始讲故事

很久很久以前有一个阿拉伯的故事……

妈妈开始讲故事

林书豪17+9助尼克斯击败老鹰……

这样修改后，无论以后怎样扩展Client类，都不需要再修改Mother类了。这只是一个简单的例子，实际情况中，代表高层模块的Mother类将负责完成主要的业务逻辑，一旦需要对它进行修改，引入错误的风险极大。所以遵循依赖倒置原则可以降低类之间的耦合性，提高系统的稳定性，降低修改程序造成的风险。

采用依赖倒置原则给多人并行开发带来了极大的便利，比如上例中，原本Mother类与Book类直接耦合时，Mother类必须等Book类编码完成后才可以进行编码，因为Mother类依赖于Book类。修改后的程序则可以同时开工，互不影响，因为Mother与Book类一点关系也没有。参与协作开发的人越多、项目越庞大，采用依赖导致原则的意义就越重大。现在很流行的TDD开发模式就是依赖倒置原则最成功的应用。

# Guice

和Spring有点类似，Guice是一个依赖注入框架，主要使用注解形式来完成依赖注入的过程，而不是像Spring一样主要使用XML。Guice的特点是足够轻量级，对于一些性能要求高的应用来讲，使用Guice时一个很好的选择。

1. Maven依赖

*<dependency>*

*<groupId>com.google.inject</groupId>*

*<artifactId>guice</artifactId>*

*<version>3.0</version>*

*</dependency>*

2）实现接口

*import com.google.inject.ImplementedBy;*

*@ImplementedBy(HelloGuiceImpl.class) //*使用了Guice提供的注解，ImplementedBy，表示该接口由HelloGuiceImpl类实现 这样就可以不手动的去配置依赖关系

*public interface HelloGuice {*

*public void sayHello();*

*}*

3）实现类

*public class HelloGuiceImpl implements HelloGuice {*

*@Override*

*public void sayHello() {*

*System.out.println("Hello Guice !");*

*}*

*}*

1. Guice实现方式1：

*import com.google.inject.Guice;*

*import com.google.inject.Injector;*

*public class TestGuice {*

*public static void main(String[] args) {*

*Injector injector = Guice.createInjector();*

*HelloGuice helloGuice = injector.getInstance(HelloGuice.class);*

*helloGuice.sayHello();*

*}*

*}*

执行结果如下：

*Hello Guice !*

1. 实现方式2：

*public static void main(String[] args) {*

*Injector inj = Guice.createInjector(*

*new Module() {*

*@Override*

*public void configure(Binder binder) {*

*binder.bind(HelloWorld.class).to(HelloWorldImpl.class);*

*}*

*}*

*);*

*HelloWorld hw = inj.getInstance(HelloWorld.class);*

*System.out.println(hw.sayHello());*

*}*

http://blog.csdn.net/u010837612/article/details/50686573#自定义ioc容器的基本架构

http://blog.csdn.net/qq\_24451605/article/details/51355701，设计模式

http://www.cnblogs.com/liuhaorain/p/3747470.html#title\_5，深入理解DIP,IoC,DI等。

http://javabeat.net/introduction-to-google-guice/

http://www.cnblogs.com/zhangchaoyang/articles/4232257.html