IoC 容器和 Dependency Injection 模式

撰文/Martin Fowler

Java 社群近来掀起了一阵轻量级容器的热潮，这些容器能够帮助开发者将来自不同项目的组件组装成为一个内聚的应用程序。在它们的背后有着同一个模式，这个模式决定了这些容器进行组件装配的方式。人们用一个大而化之的名字来称呼这个模式：“控制反转”（Inversion of Control，IoC）。在本文中，我将深入探索这个模式的工作原理，给它一个更能描述其特点的名字——“依赖注入”（Dependency Injection），并将其与“服务定位器”（Service Locator）模式作一个比较。不过，这两者之间的差异并不太重要，更重要的是：应该将组件的配置与使用分离开——两个模式的目标都是这个。

在企业级 Java 的世界里存在一个有趣的现象：有很多人投入很多精力来研究主流 J2EE 技术的替代品——自然，这大多发生在 open source 社群。在很大程度上，这可以看作是开发者对主流 J2EE 技术的笨重和复杂作出的回应，但其中的确有很多极富创意的想法，的确提供了一些可供选择的方案。J2EE 开发者常遇到的一个问题就是如何组装不同的程序元素：如果 web 控制器体系结构和数据库接口是由不同的团队所开发的，彼此几乎一无所知，你应该如何让它们配合工作？很多框架尝试过解决这个问题，有几个框架索性朝这个方向发展，提供了更通用的“组装各层组件”的方案。这样的框架通常被称为“轻量级容器”，PicoContainer 和 Spring 都在此列中。

在这些容器背后，一些有趣的设计原则发挥着作用。这些原则已经超越了特定容器的范畴，甚至已经超越了 Java 平台的范畴。在本文中，我就要初步揭示这些原则。我使用的范例是 Java 代码，但正如我的大多数文章一样，这些原则也同样适用于别的 OO 环境，特别是.NET。

组件和服务

“装配程序元素”，这样的话题立即将我拖进了一个棘手的术语问题：如何区分“服务”（service）和“组件”（component）？你可以毫不费力地找出关于这两个词定义的长篇大论，各种彼此矛盾的定义会让你感受到我所处的窘境。有鉴于此，对于这两个遭到了严重滥用的词汇，我将首先说明它们在本文中的用法。

所谓“组件”是指这样一个软件单元：它将被作者无法控制的其他应用程序使用，但后者不能对组件进行修改。也就是说，使用一个组件的应用程序不能修改组件的源代码，但可以通过作者预留的某种途径对其进行扩展，以改变组件的行为。

服务和组件有某种相似之处：它们都将被外部的应用程序使用。在我看来，两者之间最大的差异在于：组件是在本地使用的（例如 JAR 文件、程序集、DLL、或者源码导入）；而服务是要通过——同步或异步的——远程接口来远程使用的（例如 web service、消息系统、RPC，或者 socket）。

在本文中，我将主要使用“服务”这个词，但文中的大多数逻辑也同样适用于本地组件。实际上，为了方便地访问远程服务，你往往需要某种本地组件框架。不过，“组件或者服务”这样一个词组实在太麻烦了，而且“服务”这个词当下也很流行，所以本文将用“服务”指代这两者。

一个简单的例子

为了更好地说明问题，我要引入一个例子。和我以前用的所有例子一样，这是一个超级简单的例子：它非常小，小得有点不够真实，但足以帮助你看清其中的道理，而不至于陷入真实例子的泥潭中无法自拔。

在这个例子中，我编写了一个组件，用于提供一份电影清单，清单上列出的影片都是由一位特定的导演执导的。实现这个伟大的功能只需要一个方法：

```
class MovieLister...
    public Movie[] moviesDirectedBy(String arg) {
        List allMovies = finder.findAll();
        for (Iterator it = allMovies.iterator(); it.hasNext();) {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(arg)) it.remove();
        }
       return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
    }
```

你可以看到，这个功能的实现极其简单：moviesDirectedBy 方法首先请求 finder（影片搜寻者）对象（我们稍后会谈到这个对象）返回后者所知道的所有影片，然后遍历 finder 对象返回的清单，并返回其中由特定的某个导演执导的影片。非常简单，不过不必担心，这只是整个例子的脚手架罢了。

我们真正想要考察的是finder对象，或者说，如何将MovieLister对象与特定的finder对象连接起来。为什么我们对这个问题特别感兴趣？因为我希望上面这个漂亮的 moviesDirectedBy 方法完全不依赖于影片的实际存储方式。所以，这个方法只能引用一个finder对象，而 finder 对象则必须知道如何对 findAll 方法作出回应。为了帮助读者更清楚地理解，我给 finder定义了一个接口：

```
public interface MovieFinder {
    List findAll();
}
```

现在，两个对象之间没有什么耦合关系。但是，当我要实际寻找影片时，就必须涉及到 MovieFinder 的某个具体子类。在这里，我把“涉及具体子类”的代码放在 MovieLister 类的构造子中。

```
class MovieLister...
    private MovieFinder finder;
    public MovieLister() {
        finder = new ColonDelimitedMovieFinder("movies1.txt");
    }
```

这个实现类的名字就说明：我将要从一个逗号分隔的文本文件中获得影片列表。你不必操心具体的实现细节，只要设想这样一个实现类就可以了。

如果这个类只由我自己使用，一切都没问题。但是，如果我的朋友叹服于这个精彩的功能，也想

使用我的程序，那又会怎么样呢？如果他们也把影片清单保存在一个逗号分隔的文本文件中，并且也把这个文件命名为“ movie1.txt ”，那么一切还是没问题。如果他们只是给这个文件改改名，我也可以从一个配置文件获得文件名，这也很容易。但是，如果他们用完全不同的方式——例如 SQL 数据库、XML 文件、web service，或者另一种格式的文本文件——来存储影片清单呢？在这种情况下，我们需要用另一个类来获取数据。由于已经定义了 MovieFinder接口，我可以不用修改 moviesDirectedBy 方法。但是，我仍然需要通过某种途径获得合适的 MovieFinder 实现类的实例。

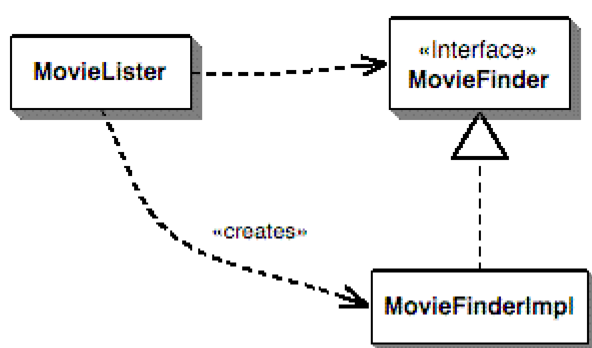

图 1： 在 MovieLister 类中直接创建 MovieFinder 实例 时的依赖关系

图 1 展现了这种情况下的依赖关系：MovieLister 类既依赖于 MovieFinder接口，也依赖于具体的实现类。我们当然希望 MovieLister 类只依赖于接口，但我们要如何获得一个 MovieFinder 子类的实例呢？

在 *Patterns of Enterprise Application Architecture* 一书中，我们把这种情况称为“插件”（plugin）：MovieFinder的实现类不是在编译期连入程序之中的，因为我并不知道我的朋友会使用哪个实现类。我们希望 MovieLister 类能够与 MovieFinder的任何实现类配合工作，并且允许在运行期插入具体的实现类，插入动作完全脱离我（原作者）的控制。这里的问题就是：如何设计这个连接过程，使 MovieLister 类在不知道实现类细节的前提下与其实例协同工作。

将这个例子推而广之，在一个真实的系统中，我们可能有数十个服务和组件。在任何时候，我们总可以对使用组件的情形加以抽象，通过接口与具体的组件交流（如果组件并没有设计一个接口，也可以通过适配器与之交流）。但是，如果我们希望以不同的方式部署这个系统，就需要用插件机制来处理服务之间的交互过程，这样我们才可能在不同的部署方案中使用不同的实现。

所以，现在的核心问题就是：如何将这些插件组合成一个应用程序？这正是新生的轻量级容器所面临的主要问题，而它们解决这个问题的手段无一例外地是控制反转（Inversion of Control）模式。

控制反转

几位轻量级容器的作者曾骄傲地对我说：这些容器非常有用，因为它们实现了“控制反转”。这样的说辞让我深感迷惑：控制反转是框架所共有的特征，如果仅仅因为使用了控制反转就认为这些轻量级容器与众不同，就好象在说“我的轿车是与众不同的，因为它有四个轮子”。

问题的关键在于：它们反转了哪方面的控制？我第一次接触到的控制反转针对的是用户界面的主控权。早期的用户界面是完全由应用程序来控制的，你预先设计一系列命令，例如“输入姓名”、

“输入地址”等，应用程序逐条输出提示信息，并取回用户的响应。而在图形用户界面环境下，UI 框架将负责执行一个主循环，你的应用程序只需为屏幕的各个区域提供事件处理函数即可。在这里，程序的主控权发生了反转：从应用程序移到了框架。

对于这些新生的容器，它们反转的是“如何定位插件的具体实现”。在前面那个简单的例子中，MovieLister 类负责定位 MovieFinder 的具体实现——它直接实例化后者的一个子类。这样一来，MovieFinder 也就不成其为一个插件了，因为它并不是在运行期插入应用程序中的。而这些轻量级容器则使用了更为灵活的办法，只要插件遵循一定的规则，一个独立的组装模块就能够将插件的具体实现“注射”到应用程序中。

因此，我想我们需要给这个模式起一个更能说明其特点的名字——“控制反转”这个名字太泛了，常常让人有些迷惑。与多位 IoC 爱好者讨论之后，我们决定将这个模式叫做“依赖注入”（Dependency Injection）。

下面，我将开始介绍 Dependency Injection 模式的几种不同形式。不过，在此之前，我要首先指出：要消除应用程序对插件实现的依赖，依赖注入并不是唯一的选择，你也可以用 Service Locator 模式获得同样的效果。介绍完 Dependency Injection 模式之后，我也会谈到 Service Locator 模式。

依赖注入的几种形式

Dependency Injection 模式的基本思想是：用一个单独的对象（装配器）来获得 MovieFinder 的一个合适的实现，并将其实例赋给 MovieLister 类的一个字段。这样一来，我们就得到了图 2 所示的依赖图：

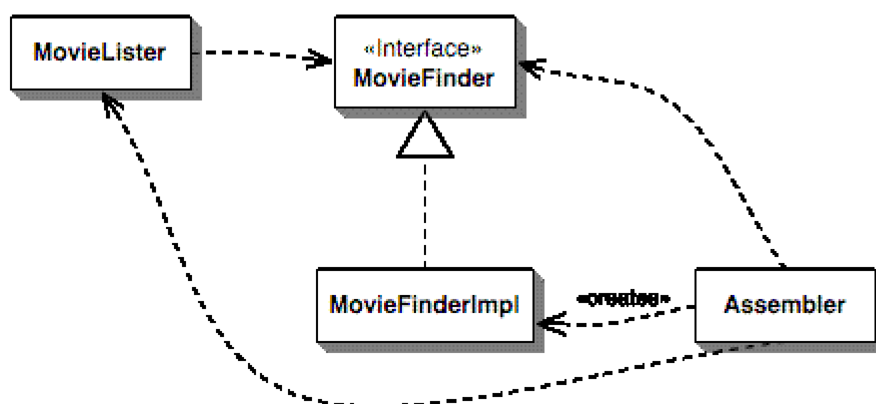

图 2：引入依赖注入器之后的依赖关系

依赖注入的形式主要有三种，我分别将它们叫做构造子注入（Constructor Injection）、设值方法注入（Setter Injection）和接口注入（Interface Injection）。如果读过最近关于 IoC 的一些讨论材料，你不难看出：这三种注入形式分别就是 type 1 IoC （接口注入）、type 2 IoC（设值方法注入）和 type 3 IoC（构造子注入）。我发现数字编号往往比较难记，所以我使用了这里的命名方式。

使用 PicoContainer 进行构造子注入

首先，我要向读者展示如何用一个名为 PicoContainer 的轻量级容器完成依赖注入。之所以从这里开始，主要是因为我在 ThoughtWorks 公司的几个同事在 PicoContainer 的开发社群中非常活跃——没错，也可以说是某种偏袒吧。

PicoContainer 通过构造子来判断"如何将 MovieFinder 实例注入 MovieLister 类"。因此，MovieLister 类必须声明一个构造子，并在其中包含所有需要注入的元素：

```
class MovieLister...
    public MovieLister(MovieFinder finder) {
        this.finder = finder;
    }
```

MovieFinder 实例本身也将由 PicoContainer 来管理，因此文本文件的名字也可以由容器注入：

```
class ColonMovieFinder...
    public ColonMovieFinder(String filename) {
        this.filename = filename;
    }
```

随后，需要告诉 PicoContainer：各个接口分别与哪个实现类关联、将哪个字符串注入 MovieFinder 组件。

```
    private MutablePicoContainer configureContainer() {
        MutablePicoContainer pico = new DefaultPicoContainer();
        Parameter[] finderParams =  {new
ConstantParameter("movies1.txt")};
        pico.registerComponentImplementation(MovieFinder.class,
ColonMovieFinder.class, finderParams);
        pico.registerComponentImplementation(MovieLister.class);
        return pico;
    }
```

这段配置代码通常位于另一个类。对于我们这个例子，使用我的 MovieLister 类的朋友需要在自己的设置类中编写合适的配置代码。当然，还可以将这些配置信息放在一个单独的配置文件中，这也是一种常见的做法。你可以编写一个类来读取配置文件，然后对容器进行合适的设置。尽管 PicoContainer 本身并不包含这项功能，但另一个与它关系紧密的项目 NanoContainer 提供了一些包装，允许开发者使用 XML 配置文件保存配置信息。NanoContainer 能够解析 XML 文件，并对底下的 PicoContainer 进行配置。这个项目的哲学观念就是：将配置文件的格式与底下的配置机制分离开。

使用这个容器，你写出的代码大概会是这样：

```
    public void testWithPico() {
        MutablePicoContainer pico = configureContainer();
        MovieLister lister = (MovieLister)
pico.getComponentInstance(MovieLister.class);
        Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
```

```
        assertEquals("Once Upon a Time in the West",
movies[0].getTitle());
    }
```

尽管在这里我使用了构造子注入，实际上 PicoContainer 也支持设值方法注入，不过该项目的开发者更推荐使用构造子注入。

使用 Spring 进行设值方法注入

Spring 框架是一个用途广泛的企业级 Java 开发框架，其中包括了针对事务、持久化框架、web 应用开发和 JDBC 等常用功能的抽象。和 PicoContainer 一样，它也同时支持构造子注入和设值方法注入，但该项目的开发者更推荐使用设值方法注入——恰好适合这个例子。

为了让 MovieLister 类接受注入，我需要为它定义一个设值方法，该方法接受类型为 MovieFinder的参数：

```
class MovieLister...
    private MovieFinder finder;
    public void setFinder(MovieFinder finder) {
        this.finder = finder;
    }
```

类似地，在 MovieFinder的实现类中，我也定义了一个设值方法，接受类型为 String 的参数：

```
class ColonMovieFinder...
    public void setFilename(String filename) {
        this.filename = filename;
    }
```

第三步是设定配置文件。Spring 支持多种配置方式，你可以通过 XML 文件进行配置，也可以直接在代码中配置。不过，XML 文件是比较理想的配置方式。

```
    <beans>
        <bean id="MovieLister" class="spring.MovieLister">
            <property name="finder">
                <ref local="MovieFinder"/>
            </property>
        </bean>
        <bean id="MovieFinder" class="spring.ColonMovieFinder">
            <property name="filename">
                <value>movies1.txt</value>
            </property>
        </bean>
    </beans>
```

于是，测试代码大概就像下面这样：

```
    public void testWithSpring() throws Exception {
        ApplicationContext ctx = new
```

```
FileSystemXmlApplicationContext("spring.xml");
        MovieLister lister = (MovieLister) ctx.getBean("MovieLister");
        Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
        assertEquals("Once Upon a Time in the West",
movies[0].getTitle());
    }
```

接口注入

除了前面两种注入技术，还可以在接口中定义需要注入的信息，并通过接口完成注入。Avalon框架就使用了类似的技术。在这里，我首先用简单的范例代码说明它的用法，后面还会有更深入的讨论。

首先，我需要定义一个接口，组件的注入将通过这个接口进行。在本例中，这个接口的用途是将一个MovieFinder实例注入继承了该接口的对象。

```
public interface InjectFinder {
    void injectFinder(MovieFinder finder);
}
```

这个接口应该由提供 MovieFinder 接口的人一并提供。任何想要使用 MovieFinder 实例的类（例如 MovieLister 类）都必须实现这个接口。

```
class MovieLister implements InjectFinder...
    public void injectFinder(MovieFinder finder) {
        this.finder = finder;
    }
```

然后，我使用类似的方法将文件名注入MovieFinder的实现类：

```
public interface InjectFilename {
    void injectFilename (String filename);
}
class ColonMovieFinder implements MovieFinder, InjectFilename......
    public void injectFilename(String filename) {
        this.filename = filename;
    }
```

现在，还需要用一些配置代码将所有的组件实现装配起来。简单起见，我直接在代码中完成配置，并将配置好的 MovieLister 对象保存在名为 lister的字段中：

```
class IfaceTester...
    private MovieLister lister;
    private void configureLister() {
        ColonMovieFinder finder = new ColonMovieFinder();
        finder.injectFilename("movies1.txt");
        lister = new MovieLister();
        lister.injectFinder(finder);
    }
```

测试代码则可以直接使用这个字段：

```
class IfaceTester...
    public void testIface() {
        configureLister();
        Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
        assertEquals("Once Upon a Time in the West",
movies[0].getTitle());
    }
```

使用 Service Locator

依赖注入的最大好处在于：它消除了MovieLister类对具体 MovieFinder实现类的依赖。这样一来，我就可以把 MovieLister 类交给朋友，让他们根据自己的环境插入一个合适的MovieFinder实现即可。不过，Dependency Injection 模式并不是打破这层依赖关系的唯一手段，另一种方法是使用Service Locator模式。

Service Locator模式背后的基本思想是：有一个对象（即服务定位器）知道如何获得一个应用程序所需的所有服务。也就是说，在我们的例子中，服务定位器应该有一个方法，用于获得一个MovieFinder 实例。当然，这不过是把麻烦换了一个样子，我们仍然必须在 MovieLister 中获得服务定位器，最终得到的依赖关系如图 3 所示：

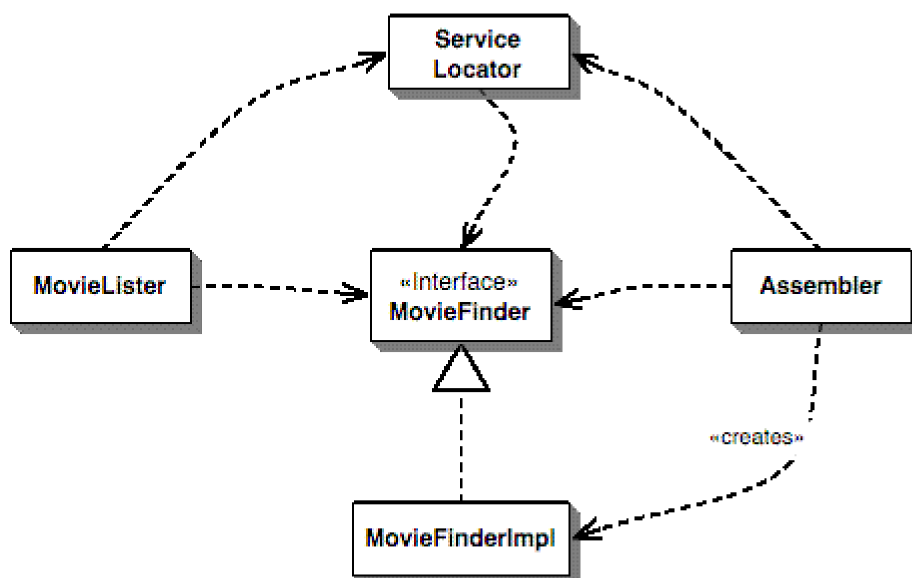

图 3：使用 Service Locator 模式之后的依赖关系

在这里，我把 ServiceLocator 类实现为一个 Singleton 的注册表，于是 MovieLister 就可以在实例化时通过ServiceLocator获得一个 MovieFinder实例。

```
class MovieLister...
    MovieFinder finder = ServiceLocator.movieFinder();
class ServiceLocator...
```

```
    public static MovieFinder movieFinder() {
        return soleInstance.movieFinder;
    }
    private static ServiceLocator soleInstance;
    private MovieFinder movieFinder;
```

和注入的方式一样，我们也必须对服务定位器加以配置。在这里，我直接在代码中进行配置，但设计一种通过配置文件获得数据的机制也并非难事。

```
class Tester...
    private void configure() {
        ServiceLocator.load(new ServiceLocator(new
ColonMovieFinder("movies1.txt")));
    }
class ServiceLocator...
    public static void load(ServiceLocator arg) {
        soleInstance = arg;
    }

    public ServiceLocator(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
```

下面是测试代码：

```
class Tester...
    public void testSimple() {
        configure();
        MovieLister lister = new MovieLister();
        Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
        assertEquals("Once Upon a Time in the West",
movies[0].getTitle());
    }
```

我时常听到这样的论调：这样的服务定位器不是什么好东西，因为你无法替换它返回的服务实现，从而导致无法对它们进行测试。当然，如果你的设计很糟糕，你的确会遇到这样的麻烦；但你也可以选择良好的设计。在这个例子中，ServiceLocator 实例仅仅是一个简单的数据容器，只需要对它做一些简单的修改，就可以让它返回用于测试的服务实现。

对于更复杂的情况，我可以从 ServiceLocator 派生出多个子类，并将子类型的实例传递给注册表的类变量。另外，我可以修改 ServiceLocator 的静态方法，使其调用 ServiceLocator 实例的方法，而不是直接访问实例变量。我还可以使用特定于线程的存储机制，从而提供特定于线程的服务定位器。所有这一切改进都无须修改 ServiceLocator 的使用者。

一种改进的思路是：服务定位器仍然是一个注册表，但不是 Singleton。Singleton 的确是实现注册表的一种简单途径，但这只是一个实现时的决定，可以很轻松地改变它。

为定位器提供分离的接口

上面这种简单的实现方式有一个问题：MovieLister类将依赖于整个ServiceLocator类，但它需要使用的却只是后者所提供的一项服务。我们可以针对这项服务提供一个单独的接口，减少MovieLister 对 ServiceLocator 的依赖程度。这样一来，MovieLister 就不必使用整个的ServiceLocator接口，只需声明它想要使用的那部分接口。

此时，MovieLister类的提供者也应该一并提供一个定位器接口，使用者可以通过这个接口获得MovieFinder实例。

```
public interface MovieFinderLocator {
    public MovieFinder movieFinder();
```

真实的服务定位器需要实现上述接口，提供访问 MovieFinder实例的能力：

```
    MovieFinderLocator locator = ServiceLocator.locator();
    MovieFinder finder = locator.movieFinder();
   public static ServiceLocator locator() {
       return soleInstance;
    }
    public MovieFinder movieFinder() {
       return movieFinder;
    }
    private static ServiceLocator soleInstance;
    private MovieFinder movieFinder;
```

你应该已经注意到了：由于想要使用接口，我们不能再通过静态方法直接访问服务——我们必须首先通过ServiceLocator类获得定位器实例，然后使用定位器实例得到我们想要的服务。

动态服务定位器

上面是一个静态定位器的例子——对于你所需要的每项服务，ServiceLocator类都有对应的方法。这并不是实现服务定位器的唯一方式，你也可以创建一个动态服务定位器，你可以在其中注册需要的任何服务，并在运行期决定获得哪一项服务。

在本例中，ServiceLocator 使用一个 map 来保存服务信息，而不再是将这些信息保存在字段中。此外，ServiceLocator还提供了一个通用的方法，用于获取和加载服务对象。

```
class ServiceLocator...
    private static ServiceLocator soleInstance;
    public static void load(ServiceLocator arg) {
       soleInstance = arg;
    }
    private Map services = new HashMap();
    public static Object getService(String key){
       return soleInstance.services.get(key);
    }
    public void loadService (String key, Object service) {
```

```
        services.put(key, service);
    }
```

同样需要对服务定位器进行配置，将服务对象与适当的关键字加载到定位器中：

```
class Tester...
    private void configure() {
        ServiceLocator locator = new ServiceLocator();
        locator.loadService("MovieFinder", new
ColonMovieFinder("movies1.txt"));
        ServiceLocator.load(locator);
    }
```

我使用与服务对象类名称相同的字符串作为服务对象的关键字：

```
class MovieLister...
    MovieFinder finder = (MovieFinder)
ServiceLocator.getService("MovieFinder");
```

总体而言，我不喜欢这种方式。无疑，这样实现的服务定位器具有更强的灵活性，但它的使用方式不够直观明朗。我只有通过文本形式的关键字才能找到一个服务对象。相比之下，我更欣赏“通过一个方法明确获得服务对象”的方式，因为这让使用者能够从接口定义中清楚地知道如何获得某项服务。

用 Avalon 兼顾服务定位器和依赖注入

Dependency Injection 和 Service Locator 两个模式并不是互斥的，你可以同时使用它们，Avalon 框架就是这样的一个例子。Avalon 使用了服务定位器，但“如何获得定位器”的信息则是通过注入的方式告知组件的。

对于前面一直使用的例子，Berin Loritsch 发送给了我一个简单的 Avalon 实现版本：

```
public class MyMovieLister implements MovieLister, Serviceable {
    private MovieFinder finder;

    public void service( ServiceManager manager )
        throws ServiceException
    {
        finder = (MovieFinder)manager.lookup("finder");
    }
```

service 方法就是接口注入的例子，它使容器可以将一个 ServiceManager 对象注入 MyMovieLister 对象。ServiceManager 则是一个服务定位器。在这个例子中，MyMovieLister 并不把 ServiceManager 对象保存在字段中，而是马上借助它找到 MovieFinder 实例，并将后者保存起来。

作出一个选择

到现在为止，我一直在阐述自己对这两个模式（Dependency Injection 模式和 Service

Locator模式）以及它们的变化形式的看法。现在，我要开始讨论他们的优点和缺点，以便指出它们各自适用的场景。

Service Locator vs. Dependency Injection

首先，我们面临Service Locator和 Dependency Injection之间的选择。应该注意，尽管我们前面那个简单的例子不足以表现出来，实际上这两个模式都提供了基本的解耦合能力——无论使用哪个模式，应用程序代码都不依赖于服务接口的具体实现。两者之间最重要的区别在于：这个“具体实现”以什么方式提供给应用程序代码。使用 Service Locator 模式时，应用程序代码直接向服务定位器发送一个消息，明确要求服务的实现；使用 Dependency Injection 模式时，应用程序代码不发出显式的请求，服务的实现自然会出现在应用程序代码中，这也就是所谓“控制反转”

控制反转是框架的共同特征，但它也要求你付出一定的代价：它会增加理解的难度，并且给调试带来一定的困难。所以，整体来说，除非必要，否则我会尽量避免使用它。这并不意味着控制反转不好，只是我认为在很多时候使用一个更为直观的方案（例如 Service Locator 模式）会比较合适。

一个关键的区别在于：使用 Service Locator 模式时，服务的使用者必须依赖于服务定位器。定位器可以隐藏使用者对服务具体实现的依赖，但你必须首先看到定位器本身。所以，问题的答案就很明朗了：选择 Service Locator 还是 Dependency Injection，取决于“对定位器的依赖”是否会给你带来麻烦。

Dependency Injection 模式可以帮助你看清组件之间的依赖关系：你只需观察依赖注入的机制（例如构造子），就可以掌握整个依赖关系。而使用Service Locator模式时，你就必须在源代码中到处搜索对服务定位器的调用。具备全文检索能力的 IDE 可以略微简化这一工作，但还是不如直接观察构造子或者设值方法来得轻松。

这个选择主要取决于服务使用者的性质。如果你的应用程序中有很多不同的类要使用一个服务，那么应用程序代码对服务定位器的依赖就不是什么大问题。在前面的例子中，我要把MovieLister类交给朋友去用，这种情况下使用服务定位器就很好：我的朋友们只需要对定位器做一点配置（通过配置文件或者某些配置性的代码），使其提供合适的服务实现就可以了。在这种情况下，我看不出 Dependency Injection 模式提供的控制反转有什么吸引人的地方。

但是，如果把 MovieLister 看作一个组件，要将它提供给别人写的应用程序去使用，情况就不同了。在这种时候，我无法预测使用者会使用什么样的服务定位器 API，每个使用者都可能有自己的服务定位器，而且彼此之间无法兼容。一种解决办法是为每项服务提供单独的接口，使用者可以编写一个适配器，让我的接口与他们的服务定位器相配合。但即便如此，我仍然需要到第一个服务定位器中寻找我规定的接口。而且一旦用上了适配器，服务定位器所提供的简单性就被大大削弱了。

另一方面，如果使用 Dependency Injection 模式，组件与注入器之间不会有依赖关系，因此组件无法从注入器那里获得更多的服务，只能获得配置信息中所提供的那些。这也是Dependency Injection 模式的局限性之一。

人们倾向于使用 Dependency Injection 模式的一个常见理由是：它简化了测试工作。这里的关键是：出于测试的需要，你必须能够轻松地在“真实的服务实现”与“供测试用的‘伪’组件”之间切换。但是，如果单从这个角度来考虑，Dependency Injection 模式和 Service Locator

模式其实并没有太大区别：两者都能够很好地支持“伪”组件的插入。之所以很多人有“Dependency Injection 模式更利于测试”的印象，我猜是因为他们并没有努力保证服务定位器的可替换性。这正是持续测试起作用的地方：如果你不能轻松地用一些“伪”组件将一个服务架起来以便测试，这就意味着你的设计出现了严重的问题。

当然，如果组件环境具有非常强的侵略性（就像EJB框架那样），测试的问题会更加严重。我的观点是：应该尽量减少这类框架对应用程序代码的影响，特别是不要做任何可能使“编辑-执行”的循环变慢的事情。用插件（plugin）机制取代重量级组件会对测试过程有很大帮助，这正是测试驱动开发（Test Driven Development，TDD）之类实践的关键所在。

所以，主要的问题在于：代码的作者是否希望自己编写的组件能够脱离自己的控制、被使用在另一个应用程序中。如果答案是肯定的，那么他就不能对服务定位器做任何假设——哪怕最小的假设也会给使用者带来麻烦。

构造子注入 vs. 设值方法注入

在组合服务时，你总得遵循一定的约定，才可能将所有东西拼装起来。依赖注入的优点主要在于：它只需要非常简单的约定——至少对于构造子注入和设值方法注入来说是这样。相比于这两者，接口注入的侵略性要强得多，比起 Service Locator 模式的优势也不那么明显。

所以，如果你想要提供一个组件给多个使用者，构造子注入和设值方法注入看起来很有吸引力：你不必在组件中加入什么希奇古怪的东西，注入器可以相当轻松地把所有东西配置起来。

设值函数注入和构造子注入之间的选择相当有趣，因为它折射出面向对象编程的一些更普遍的问题：应该在哪里填充对象的字段，构造子还是设值方法？

一直以来，我首选的做法是尽量在构造阶段就创建完整、合法的对象——也就是说，在构造子中填充对象字段。这样做的好处可以追溯到 Kent Beck 在 *Smalltalk Best Practice Patterns* 一书中介绍的两个模式：Constructor Method 和 Constructor Parameter Method。带有参数的构造子可以明确地告诉你如何创建一个合法的对象。如果创建合法对象的方式不止一种，你还可以提供多个构造子，以说明不同的组合方式。

构造子初始化的另一个好处是：你可以隐藏任何不可变的字段——只要不为它提供设值方法就行了。我认为这很重要：如果某个字段是不应该被改变的，“没有针对该字段的设值方法”就很清楚地说明了这一点。如果你通过设值方法完成初始化，暴露出来的设值方法很可能成为你心头永远的痛。（实际上，在这种时候我更愿意回避通常的设值方法约定，而是使用诸如 initFoo 之类的方法名，以表明该方法只应该在对象创建之初调用。）

不过，世事总有例外。如果参数太多，构造子会显得凌乱不堪，特别是对于不支持关键字参数的语言更是如此。的确，如果构造子参数列表太长，通常标志着对象太过繁忙，理应将其拆分成几个对象，但有些时候也确实需要那么多的参数。

如果有不止一种的方式可以构造一个合法的对象，也很难通过构造子描述这一信息，因为构造子之间只能通过参数的个数和类型加以区分。这就是 Factory Method 模式适用的场合了，工厂方法可以借助多个私有构造子和设值方法的组合来完成自己的任务。经典 Factory Method 模式的问题在于：它们往往以静态方法的形式出现，你无法在接口中声明它们。你可以创建一个工厂类，但那又变成另一个服务实体了。“工厂服务”是一种不错的技巧，但你仍然需要以某种方式实例化这个工厂对象，问题仍然没有解决。

如果要传入的参数是像字符串这样的简单类型，构造子注入也会带来一些麻烦。使用设值方法注入时，你可以在每个设值方法的名字中说明参数的用途；而使用构造子注入时，你只能靠参数的位置来决定每个参数的作用，而记住参数的正确位置显然要困难得多。

如果对象有多个构造子，对象之间又存在继承关系，事情就会变得特别讨厌。为了让所有东西都正确地初始化，你必须将对子类构造子的调用转发给超类的构造子，然后处理自己的参数。这可能造成构造子规模的进一步膨胀。

尽管有这些缺陷，但我仍然建议你首先考虑构造子注入。不过，一旦前面提到的问题真的成了问题，你就应该准备转为使用设值方法注入。

在将 Dependecy Injection 模式作为框架的核心部分的几支团队之间，“构造子注入还是设值方法注入”引发了很多的争论。不过，现在看来，开发这些框架的大多数人都已经意识到：不管更喜欢哪种注入机制，同时为两者提供支持都是有必要的。

代码配置 vs. 配置文件

另一个问题相对独立，但也经常与其他问题牵涉在一起：如何配置服务的组装，通过配置文件还是直接编码组装？对于大多数需要在多处部署的应用程序来说，一个单独的配置文件会更合适。配置文件几乎都是XML 文件，XML 也的确很适合这一用途。不过，有些时候直接在程序代码中实现装配会更简单。譬如一个简单的应用程序，也没有很多部署上的变化，这时用几句代码来配置就比 XML 文件要清晰得多。

与之相对的，有时应用程序的组装非常复杂，涉及大量的条件步骤。一旦编程语言中的配置逻辑开始变得复杂，你就应该用一种合适的语言来描述配置信息，使程序逻辑变得更清晰。然后，你可以编写一个构造器（builder）类来完成装配工作。如果使用构造器的情景不止一种，你可以提供多个构造器类，然后通过一个简单的配置文件在它们之间选择。

我常常发现，人们太急于定义配置文件。编程语言通常会提供简捷而强大的配置管理机制，现代编程语言也可以将程序编译成小的模块，并将其插入大型系统中。如果编译过程会很费力，脚本语言也可以在这方面提供帮助。

通常认为，配置文件不应该用编程语言来编写，因为它们需要能够被不懂编程的系统管理人员编辑。但是，这种情况出现的几率有多大呢？我们真的希望不懂编程的系统管理人员来改变一个复杂的服务器端应用程序的事务隔离等级吗？只有在非常简单的时候，非编程语言的配置文件才有最好的效果。如果配置信息开始变得复杂，就应该考虑选择一种合适的编程语言来编写配置文件。

在 Java 世界里，我们听到了来自配置文件的不和谐音——每个组件都有它自己的配置文件，而且格式还各各不同。如果你要使用一打这样的组件，你就得维护一打的配置文件，那会很快让你烦死。

在这里，我的建议是：始终提供一种标准的配置方式，使程序员能够通过同一个编程接口轻松地完成配置工作。至于其他的配置文件，仅仅把它们当作一种可选的功能。借助这个编程接口，开发者可以轻松地管理配置文件。如果你编写了一个组件，则可以由组件的使用者来选择如何管理配置信息：使用你的编程接口、直接操作配置文件格式，或者定义他们自己的配置文件格式，并将其与你的编程接口相结合。

分离配置与使用

所有这一切的关键在于：服务的配置应该与使用分开。实际上，这是一个基本的设计原则——分离接口与实现。在面向对象程序里，我们在一个地方用条件逻辑来决定具体实例化哪一个类，以后的条件分支都由多态来实现，而不是继续重复前面的条件逻辑，这就是“分离接口与实现”的原则。

如果对于一段代码而言，接口与实现的分离还只是“有用”的话，那么当你需要使用外部元素（例如组件和服务）时，它就是生死攸关的大事。这里的第一个问题是：你是否希望将“选择具体实现类”的决策推迟到部署阶段。如果是，那么你需要使用插入技术。使用了插入技术之后，插件的装配原则上是与应用程序的其余部分分开的，这样你就可以轻松地针对不同的部署替换不同的配置。这种配置机制可以通过服务定位器来实现（Service Locator 模式），也可以借助依赖注入直接完成（Dependency Injection 模式）。

更多的问题

在本文中，我关注的焦点是使用 Dependency Injection 模式和 Service Locator 模式进行服务配置的基本问题。还有一些与之相关的话题值得关注，但我已经没有时间继续申发下去了。特别值得注意的是生命周期行为的问题：某些组件具有特定的生命周期事件，例如“停止”、“开始”等等。另一个值得注意的问题是：越来越多的人对“如何在这些容器中运用面向方面（aspect oriented）的思想”产生了兴趣。尽管目前还没有认真准备过这方面的材料，但我也很希望以后能在这个话题上写一些东西。

关于这些问题，你在专注于轻量级容器的网站上可以找到很多资料。浏览 PicoContainer（http://www.picocontainer.org）或者 Spring（http://www.springframework.org）的网站，你可以找到大量相关的讨论，并由此引申出更多的话题。

结论和思考

在时下流行的轻量级容器都使用了一个共同的模式来组装应用程序所需的服务，我把这个模式称为 Dependency Injection，它可以有效地替代 Service Locator 模式。在开发应用程序时，两者不相上下，但我认为 Service Locator 模式略有优势，因为它的行为方式更为直观。但是，如果你开发的组件要交给多个应用程序去使用，那么 Dependency Injection 模式会是更好的选择。

如果你决定使用 Dependency Injection 模式，这里还有几种不同的风格可供选择。我建议你首先考虑构造子注入；如果遇到了某些特定的问题，再改用设值方法注入。如果你要选择一个容器，在其之上进行开发，我建议你选择同时支持这两种注入方式的容器。

Service Locator 模式和 Dependency Injection 模式之间的选择并是最重要的，更重要的是：应该将服务的配置和应用程序内部对服务的使用分离开。

致谢

在此，我要向帮助我理解本文中所提到的问题、并对本文提出宝贵意见的几个人表示感谢，他们是 Rod Johnson、Paul Hammant、Joe Walnes、Aslak Hellesoy、Jon Tirsen 和 Bill Caputo。

另外，Berin Loritsch和Hamilton Verissimo de Oliveira在Avalon方面给了我非常有用的建议，一并向他们表示感谢。