控制反转 (IoC) 和依赖注入 (DI)

2012 年 04 月 04 日

说来惭愧，听说 控制反转（Inversion of Control） 和 依赖注入（Dependency Injection） 这两个名词已经是大二的事情了，时至今日才想明白了具体的含义。我之前知道依赖注入的用法，一直将控制反转将其等同化，现在看来二者其实还是存在差别的。于是今天就此也顺便查阅了一些资料，将一些思考记录下来。

框架和库（Framework and Library）的区别

最早看到这个问题，是在一份面试题上：“框架（Framework）和库（Library）有什么区别？”，那篇博文的作者给出的是一个很口头的说法：客户程序员使用库，框架使用客户程序员。我当时只是粗略地接受了这个说法，并没有意识到，这就是控制是否被反转了的区别。今天看 limodou 在 PyCon 2011 的 Presentation 《Web 框架开发思考与实践》，再次提到了这个问题。limodou 引用了 Martin Fowler 对控制反转的定义：

> Inversion of Control is a key part of what makes a framework different to a library. A library is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.
>
> A framework embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.

在 Martin Fowler 的看法中，控制反转是框架和库的关键区别所在。对于一个库而言，用户程序员使用的方式是主动调用它，这是通常情况的做法，也就是“正向”控制；而对于一个框架，往往将用户程序员编写的代码注册到框架中，最后由框架来调用用户程序员编写的代码，这就构成了控制反转。也就是说，控制反转的关键在于“控制者”是谁。对于一个库而言，复用的可能只是算法和数据结构；而对于一个框架而言，复用的往往还有控制流逻辑，这也是控制反转的结果。

看 Python Web 框架中的控制反转

我最喜欢的一个 Python Web 框架是 Flask，从这里我们可以轻易的找出控制反转的应用。比如编写一个 Hello World 演示页面，Flask 的做法是：

> hello.py

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def show_helloworld():
    return "hello, world"

app.run()
```

在这里，用户程序员编写了产生视图输出的函数 `show_helloworld` 。当用户不用管 WSGI 是如何控制执行流的，只要将它用 `@app.route` 装饰器注册到 `Flask` 的实例 `app` 中，调用 `app.run` 之后 `Flask` 将在每次请求的 URI 命中路由 / 时，调用 `show_helloworld` 。

所以，用 Flask 编写 Web 应用的方式是控制反转的应用，Flask 是一个框架而不是一个库。

而对比 Flask 底层所使用的基础库 Werkzeug，可以很明显的看出区别。同样的 Hello World 演示，原始 WSGI 这样写：

wsgi.py

```
def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello World!']
```

Werkzeug 则在 HTTP 请求、响应方面提供了更为方便的包装：

wsgi.py

```
from werkzeug.wrappers import Request, Response

def application(environ, start_response):
    request = Request(environ)
    text = 'Hello %s!' % request.args.get('name', 'World')
    response = Response(text, mimetype='text/plain')
    return response(environ, start_response)
```

这种包装和 Flask 最大的不同，是这里并未将控制权交给 Werkzeug，而是仍然把握在 `application` 函数内，用户程序员调用了 Werkzeug 的组件，但并未委托控制权。如果要实现和上述 Flask 版 Hello World 等价，这里还需要判断 URI 是否为 `'/'` ，如果不是还需要返回 HTTP 404 错误。但如果要实现这些，都需要用户程序员在 `application` 函数中自己实现。实现过程可能可以调用 Werkzeug 简化细节，但控制权仍然在用户程序员自己手里。所以这里没有应用控制反转，Werkzeug 是一个库而不是一个框架。

依赖注入

依赖注入（Dependency Injection） 则是和控制反转相关的一个话题，最初的目的是解除程序组件之间的一些直接耦合。具体的做法是，将原本的 A 依赖 B，改成 A 依赖一个接口和一个 DI 容器（B 实现了 A 依赖的那个接口），然后将 B 的创建工厂（其中可能使用了创建型设计模式，并将类型转化为 A 所依赖的接口才返回）注册到 DI 容器，最后 A 只需要给 DI 容器一个标识索取 B，DI 容器将回调 B 的创建工厂，生产出 B 的实例之后交给 A。

如果在动态语言（比如 Python 比如 Ruby）里面这么做，则是一件非常第三类青年的事情。为什么不能直接让 A 去调用 B 的工厂呢？中途插入一个注册表有区别吗？所以我不做第三类青年，在使用 Python 的时候我会选择直接调用工厂，但这在静态语言中行不通。一来静态语言有着静态的变量类型，手动构造带有类型转换的工厂，还不如交托给一个自动构造的容器；二来静态语言的编译期、装载期、运行期是严格分离的，要想运行期做装载的事情，必须运用反射来处理，这个处理语言并没有内置（否则就近似动态语言了），必须编写专门的组件来做，DI 容器就是专门的组件。

归根结底，依赖注入的本质是运用控制反转。前面说了，要生产对象的工厂是被注册到 DI 容器的（有的 DI 容器可以自动构造常用的工厂），也就是说 DI 容器在对象生产方面就是“框架”，是控制权的主导者。

依赖注入在静态语言中的价值

使用依赖注入，对于静态语言编写的应用来说最大的好处是增加可测试性。前文说了，依赖注入对于静态语言编写的应用一个重要价值是将对象的索取从编译期、装载期移到了运行期。如此一来，即使是已经经过编译的组件，也可以通过更改 DI 容器的配置，将一个被依赖对象替换。

在做单元测试的时候，用 Stub、Mock 隔离依赖的组件是常用的手法。一般的做法是让 Stub、Mock 对象和真实对象实现相同的接口，以达到“以假乱真”的目的。DI 容器在这个过程中，则负责“替换”这个过程。在生产环境，DI 容器使用生产环境的配置文件，注入到目标对象中的是真实对象；而在测试环境，DI 容器使用测试环境的配置文件，将 Stub 或 Mock 注入到目标对象。

DI 容器在动态语言中的不必要

前文说到，依赖注入对于静态语言构建的项目来说很有价值的一点是增强可测试性。这是由于静态语言本身在编译期、装载期处理依赖。对于动态语言来说，编译期、装载期、运行期是一体的，所以依赖注入在动态语言中可以有着原生的支持，独立的 DI 容器在动态语言构建的项目中没有存在的必要性。

还是前文说的单元测试 Stub、Mock 的例子，假如在 Python 这样的动态语言中，需要对一个已经编译好的模块使用 Stub、Mock，只需要直接替换其属性。

```
import unittest
import urllib2
import fakeurllib2

def install_mock(urllib2):
    urlllib2.Request = fakeurllib2.Request

def install_stub(urllib2):
    urllib2.urlopen = fakeurllib2.urlopen


install_mock(urllib2)
install_stub(urllib2)


class MyTest(unittest.TestCase):

    # ...
```

可见，动态语言的运行时环境本身就是一个强大的依赖注入容器。

Posted by Jiangge Zhang 2012 年 04 月 04 日 IoC DI Design Pattern Note