第16卷 第2期
2006年2月

计算机技术与发展
COMPUTER TECHNOLOGY AND DEVELOPMENT

Vol. 16 No. 2
Feb. 2006

控制反转模式及其应用

吴 刚，郝克刚，葛 玮
（西北大学 计算机科学系，陕西 西安 710069）

摘 要：经过长时间的酝酿，控制反转模式（Inversion of Control）已经得到了广泛的应用。利用面向对象设计方法的原则和特性，控制反转模式可以设计更好的可重用、低藕合、易测试的软件系统。分析了控制反转模式及其各种实现方法（接口注入，设置注入，构造方法注入）的优缺点，最后根据软件工程发展的最新进展，提出了将控制反转模式与面向方面软件开发方法相结合应用，这也是有待进一步研究的问题。

关键词：控制反转；模式；依赖注入
中图分类号：TP391 **文献标识码**：A **文章编号**：1005－3751(2006)02－0171－02

Research of Inversion of Control Pattern and Its Applications

WU Gang, HAO Ke-gang, GE Wei
(Department of Computer Science, Northwest University, Xi'an 710069, China)

Abstract: Inversion of Control(IoC) pattern has moved to the center stage recently after a long gestation period. Using object－oriented design principles and features, such as interface, inheritance, and polymorphism, the IoC pattern enables better software design that facilitates reuse, loose coupling, and easy testing of software components. Analyzes the IoC pattern with the merits and demerits of its implementation methods(Interface injection, Setter injection, Construct injection), moreover, Based on the new development of software engineering, one solution is given, which integrates IoC pattern into aspect－oriented software development, this will be the topics of future research.

Key words: inversion of control; pattern; dependency injection

0 引 言

传统的设计模式为了考虑将来的变化和扩展常常导致当前系统的过度设计，或者由于模块间复杂的依赖关系使得系统变的臃肿笨拙，难以维护和移植。如果有一种可能的变化在设计阶段没有被考虑到，那么这种变化对系统所带来的冲击也不大可能自然的消失，而一种显而易见的事实是设计者不可能穷尽未来所有的变化，这种必然的矛盾毫无疑问将会增加软件开发的成本和风险。

1 控制反转模式的定义

在面向对象设计方法中经常要涉及到一个对象对另一个对象的引用[1]，这种依赖关系可能是因为业务逻辑的需要，也可能是因为数据持久化的需要，这种依赖关系一般是正向进行的，即在A中直接引用控制B(见图1a)。如果由于环境的变化(如数据持久化类型的改变)导致A中的依赖从B迁移到C，那么A不可避免地必须进行相应的变动(见图1b)。然而这种依赖迁移的时间和次数不可预知，如果系统中存在多处这样的依赖关系，那么系统的演化和维护将会变得异常困难。

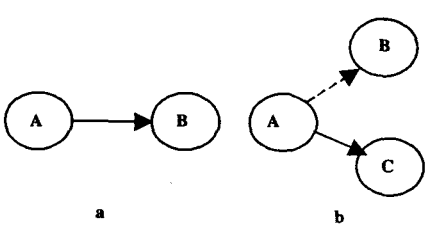


图1 直接控制

但是如果改变一种思路：把A对B的控制反转过来：让A和B各自独立的变化，仅仅当A在需要B的时候通过某种方式B将依赖关系主动注入到A中，完成依赖的传递，使得B对A可用(见图2a)，这样当依赖关系B改变时A可以保持不变，这种对依赖的控制由直接主动变为间接被动的过程大大增加了系统的灵活性、扩展性和可维护性(见图2b)。

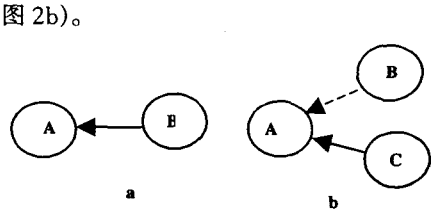


图2 控制反转

收稿日期：2005－05－21
作者简介：吴 刚（1977—），男，陕西咸阳人，硕士研究生，研究方向为软件工程、面向方面软件开发、Web服务；郝克刚，教授，博士生导师，研究方向为构件技术、软件方法学等。

以上方式实际运用的就是控制反转(Inversion of Control)模式,控制权的转移,就是所谓的反转。控制反转模式关注的是处理对象或组件间的依赖关系。这种模式借用了桥接模式的基本思想:将抽象部分和它的实现部分相分离,使得它们都可以各自独立的变化,这种分离的过程也就是解耦的过程,不同的是桥接模式利用对象的组合来完成不同依赖关系的接合,而控制反转则是利用依赖注入(Dependency Injection)或依赖查找(Dependency Lookup)完成组件间依赖关系的接合。在实际应用中如果想打破A->B的依赖,可以通过引入A、B之间的交互协议I来办到,也就是将A->B变为(A->I)+(I<-B)。此举同时满足了依赖注入原则(Dependency Injection Principle)和开闭原则(Open-Close Principle),这种交互协议可以引申为外部的容器或框架,由它们来管理对象的生命周期。IoC模式控制依赖关系的粒度可以根据需要自由伸缩,小到对象间的协作,大到容器或框架的实现,然而勿庸置疑的是在容器或框架的管理中控制反转更能发挥强大的作用。

2 控制反转模式的分类及实现

2.1 控制反转模式的分类

控制反转模式根据其实现方式是否具有侵入性分为两种:依赖查找和依赖注入[2]。依赖查找是侵入性的,对象中方法的实现与具体的容器或框架相关,而且不能脱离外部环境独立运行,一般由接口注入(Interface Injection,也称IoC type 1)实现;依赖注入是非侵入性的,对象中方法的实现与具体的容器或框架无关,可以利用编程语言自身的特点实现语言级别的依赖注入,可以脱离外部环境独立运行,一般由设值注入(Setter Injection,也称IoC type 2)或构造方法注入(Constructor Injection,也称IoC type 3)实现。控制反转的分类见图3。

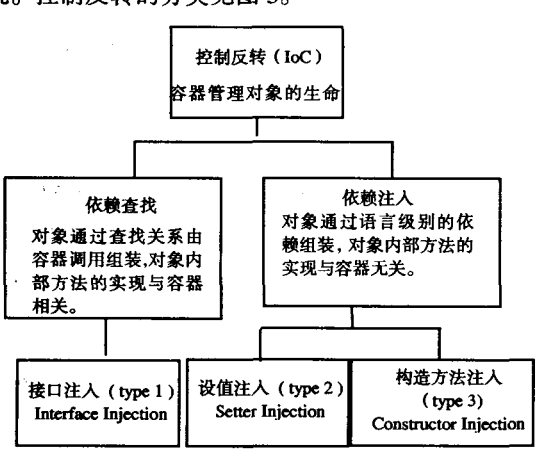


图3 IoC的分类

2.2 各种实现方式的优缺点

(1)设值注入利用属性设置的方法完成依赖注入。首先现今的集成开发环境对涉及属性的方法支持性较好,比如eclipse中可以自动生成javabean属性的getter和setter方法,因此对属性可以用标准的转换编辑器进行统一的类型转换;其次,如果实现属性的getter方法则可以随时查询组件当前的配置状态,可以将其持续化并以数据库或者XML的形式保存,构造方法注入则无此功能;最后,属性可以由子类自动继承,因此现存的模块可以无缝移植到支持控制反转的容器中,从而利用成熟的单元测试框架(JUnit等)构造健壮的软件系统。然而对于属性显式的设置从一定程度上破坏了封装原则,并且属性初始化的先后顺序无法保证,因而导致一些安全隐患,并可能使得程序模块在容器内外的行为不一致。

(2)构造方法注入则是利用普通的构造方法进行依赖注入,由于要保证参数的完全匹配,因此每一个被管理的组件实例在使用前都可以保证一致的状态,消除设置注入的安全隐患,构造方法注入的主要缺点是构造方法可能不止一个并且参数的顺序和个数容易造成混乱[3],同时继承功能必须显式地指定并且不支持动态配置。

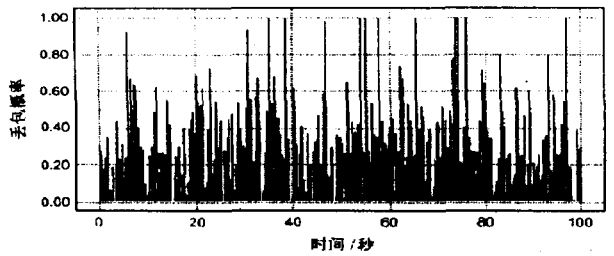
(3)接口注入的主要优点是可以不需要外部的配置文件配置组件,只要实现指定的接口,IoC框架自动实现协作对象的组合,但是由于和框架或容器自身细节关系比较紧密,不利于移植和复用。

以上各注入类型的优缺点都是相对的,可以根据具体情况选择不同的最佳实践,也可以组合使用。现今许多成熟的框架和容器都已经大量地使用了控制反转模式,典型的应用实例包括:利用接口注入实现的JContainer和Avalon容器;利用设置注入实现的Spring Framework[4]和WebWork/XWork;利用构造方法注入实现的Pico和HiveMind容器。

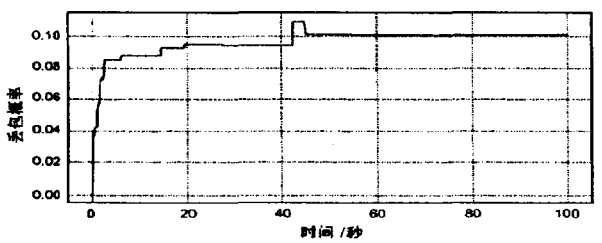
3 控制反转模式与AOP的结合

作为对面向对象技术的重要补充,面向方面编程(aspect-oriented programming)很好地解决了横切关注点(crosscutting concern)的问题[5]。横切关注点指的是传统面向对象程序设计中不能自然地适合单个程序模块或者几个紧密相关的程序模块的行为,如日志记录、安全策略、性能优化等往往要求横切几个模块,其实质是分散的。方面就是实现这些关注点的模块单元。AOP为开发者提供了一种描述横切关注点的机制,并能够通过切入点(Pointcut)自动将横切关注点织入(Weaving)到面向对象的软件系统中,整个过程遵循方面分解(aspectual decomposition)-关注点实现(concern implementation)-方面重组(aspectual recomposition)的步骤,从而实现横切关注点的模块化,使得系统保持简洁优雅、利于扩展和维护的结构。而IoC和AOP的完美结合更加突出了这种优势,AOP中的许多机制如通知、切入点、引入(introductions)等都可以由IoC模式管理和配置。一种典型的情况就是当各个方面织入系统的过程时可以利用通知机制向其注入子类(见图4)。该图的设计要点如下:

(下转第175页)

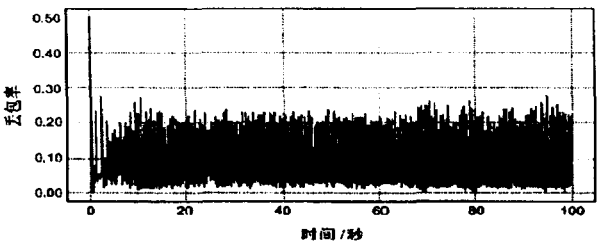


(a) RED 算法

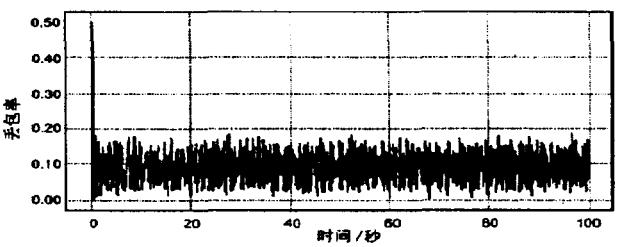


(b) 改进算法

图 4　丢包概率变化



(a) RED 算法



(b) 改进算法

图 5　丢包率变化

参考文献:

[1] Floyd S. A Report on Some Recent Developments in TCP Congestion Control[J/OL]. http://www.icir.org/floyd/, 2001.

[2] Allman M. Ongoing TCP Research Related to Satelites RFC [EB/OL]. http://roland.grc.nasa.gov/~mallma, 2000-02.

[3] Balakrishnan H, Padmanabhan V N, Seshan S, et al. A Comparison Of Mechanisms For Improving TCP Performance Over Wireless Links[EB/OL]. http://nms.lcs.mit.edu/papers/hari-phd/, 1996.

[4] Floyd S, Jacobson V. Random early detection gateway for congestion avoidance[J]. IEEE/ACM Transactions on Networking, 1993, 1(4): 397-413.

[5] McCanne S, Floyd S. ns-LBNL Network Simulator[CP/OL]. http://www-nrg.ee.lbl.gov/ns/, 1996.

(上接第 172 页)

Join points(联结点):在程序结构或者执行过程中明确定义的可以联结附加行为的点,可以是一个方法的调用,或者是一个特别的被抛出的异常。

Advice(通知):在结合点执行的行为,其作用类似拦截器。不同的通知类型包括:around advice, before advice, after advice, throws advice。

Pointcut(切入点):指定一个通知将被引发的一系列 Join points 的集合。

IoC模式和 AOP 方法的紧密结合首先会在支持 IoC 模式的容器和框架中得到广泛的应用,然而如何使得两者更加有机而无缝的结合并得到大量实用工具的有力支持仍需要进一步的研究和实践。

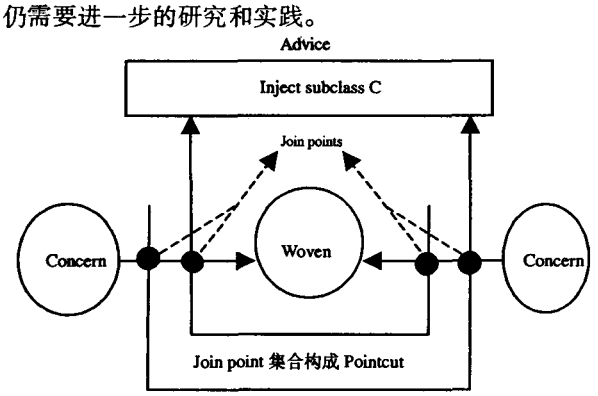


图 4　AOP 与 IoC

4　结　论

讨论了 IoC 及其分类和应用,总结了各种实现 IoC 方法的优缺点。由于 IoC 模式概念清晰,对依赖关系有着强大的分解和操作能力,使得 IoC 成为一种构建轻型容器和框架必要而可行的模式,相信未来其与 AOP 的结合可以最大限度地发挥功用,从而更简单地构建高质量的软件系统。

参考文献:

[1] Malarvannan M. Design better software with the inversion of control pattern[EB/OL]. http://www.devx.com/Java/Article/27583/0/page/1, 2005-03-18.

[2] Johnson R, Hoeller J. Expert one-on-one J2EE development without EJB[M]. Indiana: Wiley Publishing Inc., 2004.

[3] Fowler M. Inversion of control containers and the dependency injection pattern[EB/OL]. http://www.martinfowler.com/articles/injection.html, 2004-01-23.

[4] Johnson R, Hoeller J, Arendsen A, et al. Spring-Java/J2EE Application Framework Reference Documentation[EB/OL]. http://www.springframework.org/docs/reference/index.html, 2005.

[5] Kiczales G, Lamping J, Mendhekar A, et al. Aspect-Oriented Programming[M]. Finland: Springer-Verlag, 1997.