21 世纪全国高校应用人才培养信息技术类规划教材

软件动态演化技术

李长云　何频捷　李玉龙　编　著

北京大学出版社
PEKING UNIVERSITY PRESS

内 容 简 介

为了适应 Internet 开放环境和用户需求的不断变化，软件系统需要不断调整自身。软件动态演化技术是满足这一变化的有效手段，也是自治计算、网格计算、自适应软件和网构软件的核心技术。本书是国内外第一本对软件动态演化技术进行系统阐述的著作。作者结合多年研究和实践的经验，从开放系统发展动力到动态演化技术产生，从动态演化技术基本原理到动态演化的形态和粒度，从动态配置技术到基于反射的动态演化、动态演化的基础设施，从设计可动态演化的软件系统到动态演化技术的应用以及未来发展趋势，都进行了系统的论述。本书最后部分介绍了作者提出的软件动态演化模型 SASM 及其支持工具和平台，望起到抛砖引玉的作用。

本书内容全面、叙述清楚，注意一些最新的协议、规范及学术界、工业界研究进展，同时还非常注重实用性。本书既适用于本科高年级和研究生的教学，也可供工程技术人员自学参考之用。

图书在版编目（CIP）数据

软件动态演化技术/李长云，何频捷，李玉龙编著．—北京：北京大学出版社，2007.11
（21 世纪全国高校应用人才培养信息技术类规划教材）
ISBN 978-7-301-12989-0

I．软…　II．①李…②何…③李…　III．软件工程－高等学校－教材　IV．TP311.5

中国版本图书馆 CIP 数据核字（2007）第 192065 号

书　　名：软件动态演化技术
著作责任者：李长云　何频捷　李玉龙　编著
责 任 编 辑：卢英华
标 准 书 号：ISBN 978-7-301-12989-0/TP・0918
出　版　者：北京大学出版社
地　　址：北京市海淀区成府路 205 号　100871
电　　话：邮购部 62752015　发行部 62750672　编辑部 62765126　出版部 62754962
网　　址：http://www.pup.cn
电 子 信 箱：xxjs@pup.pku.edu.cn
印　刷　者：
发　行　者：北京大学出版社
经　销　者：新华书店
　　787 毫米×1092 毫米　16 开本　18.75 印张　456 千字
　　2007 年 11 月第 1 版　2007 年 11 月第 1 次印刷
定　　价：38.00 元

前　言

“变化”是现实世界永恒的主题，只有“变化”才能发展。Lehman 认为，现实世界的系统要么变得越来越没有价值，要么进行持续不断的变化以适应环境的变化。杨芙清院士指出，软件是对现实世界中问题空间与解空间的具体描述，是客观事物的一种反映。现实世界是不断变化的，因此，变化性是软件的基本属性。特别是在 Internet 成为主流软件运行环境之后，网络的开放性和动态性使得客户需求与计算环境更加频繁地变化，导致软件的变化性和复杂性进一步增强。有研究指出，Internet 具备如下基本特征：1）网络的开放性；2）资源的动态性；3）计算环境的多样性；4）用户需求的多变性。考察当前的主流软件技术，可以发现目前的软件系统本质都是一种封闭、静态的软件体系框架，难以适应 Internet 开放、动态环境的要求。

为适应这种发展趋势，人们纷纷从不同的视角对软件理论、技术和方法进行研究，例如从软件形态方面，提出了网构软件概念；从网络资源整合方面，提出了网格计算的概念；从计算模式方面，提出了自治计算、自适应软件的概念，以及面向服务的体系结构（SOA），反射中间件等等。纵观所有这些新的概念、思想和理论，都牵涉到一个共同的课题：开放环境下软件系统的动态演化理论和应用研究。软件演化指的是软件进行变化并达到人们所希望的形态的过程，分为静态演化和动态演化两种类型。静态演化是指软件在停机状态下的演化；动态演化是指在软件系统投入执行之后，应新环境条件和需求状况的改变而进行的演化。在传统的软件工程体系中，软件生命周期概念所强调的是从需求提出到软件提交的整个开发过程的重要性，而对于提交使用之后的软件变化过程往往只采用“软件维护”（即静态演化）加以简单概括。这样一种软件生命周期概念对于处于静态封闭环境下的软件系统的开发是合适的，但对处于 Internet 开放、动态和多变环境下软件系统的开发则有局限性。因此，Internet 环境下软件的开发对运行阶段之后的演化过程（即动态演化）的关注是必需的，甚至显得更为重要。

另一方面，在许多重要的应用领域中，系统的持续可用性是一个关键性的要求，运行期间的动态演化可减少因关机和重启而带来的损失和风险。例如对于执行关键任务的一些软件系统，通过停止、更新和重启来实现维护和演化将导致不可接受的延迟、代价和危险。容错系统发生错误时，需要切换到备用冗余部分，确保服务可用。移动计算在环境改变时，需要相应调整计算构件，适应环境变化。诸如交通控制软件、电信交换软件、Internet 服务应用以及高可用性的公共信息系统必须以 7×24 的方式运行，但是又经常需要演化以适应外部环境的变化和满足客户的更多需求，于是它们只能在运行期间进行扩展和升级。此外，越来越多的其他类型的应用软件也提出了运行时刻演化的要求，在不必对应用软件进行重新编译和加载的前提下，为最终用户提供系统定制和扩展的能力。因此，一般的应用软件如果具有运行时修改的特性，将大大提高系统的自适应性和敏捷性，从而延长软件的生命周期，增强企业的竞争力。

目前围绕软件动态演化的相关理论、方法和技术基本还处于起步阶段。结合我们这几年的研究和实践，本书试图对已有动态演化研究成果作一个较系统的整理。从开放系统发展动力到动态演化技术产生，从动态演化技术基本原理到动态演化的形态和粒度，从动态配置技术到基于反射的动态演化、动态演化的基础设施，从设计可动态演化的软件系统到动态演化技术的应用以及未来发展趋势，都进行了较系统的论述。本书最后部分介绍了作者提出的软件动态演化模型 SASM 及其支持工具和平台。

李长云负责本书的统一组织和策划。第 1、2、5、6、9 章由李长云编写，第 3、8、10 章由何频捷编写，第 4、7 章由李玉龙编写，王志兵、梁爱兰、吴岳忠、李伟参与了书稿内容和文字的斟酌、修改、校对及绘图工作。在这里要特别感谢北京航空航天大学计算机学院的马世龙教授、国防科技大学计算机学院的王怀民教授、湖南大学计算机与通信学院的李仁发教授的宝贵意见和指导。本书的研究与写作得到国家自然科学基金“开放环境下的软件动态演化研究”、湖南省教育厅优秀青年科研项目“基于高阶 π 演算的动态体系结构语言研究”、湖南省学位与研究生教改课题“应用导向型的研究生专业课程教学新模式”和湖南工业大学重点教改课题“计算机基础实验教学示范中心建设”的资助。在此我们表示衷心的感谢。

由于作者的认知和水平所限，加上国内外对软件动态演化技术进行系统阐述的资料较少，没有太多可借鉴的经验，本书体系可能是不完整的、不尽合理的，资料的来源也可能不具完全的代表性，敬请读者批评指正。

编　者

2007 年 8 月

目　　录

第 1 章　概　　述

1.1　软件演化的基本概念

“变化”是现实世界永恒的主题，只有“变化”才能发展。Lehman 认为，现实世界的系统要么变得越来越没有价值，要么进行持续不断的演化变化以适应环境的变化。软件是对现实世界中问题空间与解空间的具体描述，是客观事物的一种反映。现实世界是不断演化的，因此，演化性是软件的基本属性。特别是在 Internet 成为主流软件运行环境之后，网络的开放性和动态性使得客户需求与硬件资源更加频繁地变化，导致软件的演化性和复杂性进一步增强。

1.1.1　软件演化、软件维护与软件复用

软件演化（Software Evolution）指在软件系统的生命周期内软件维护和软件更新的行为和过程。在现代软件系统的生命周期内，演化是一项贯穿始终的活动，系统需求改变、功能实现增强、新功能加入、软件体系结构改变、软件缺陷修复、运行环境改变无不均要求软件系统具有较强的演化能力，能够快速适应改变，减少软件维护的代价。

软件演化研究致力于寻找规则、识别模式。这种规则和模式，主要用于管理那些由应用程序的编程人员或者是维护人员所作的变更。这意味着软件演化研究与导致系统不一致的变化有关。软件演化是一个模糊而范围广泛的概念，普遍的理解认为演化就是修改软件，而且意味着需要一个新的开发周期，包括新的测试和生成阶段，这样对于资源消耗来说是很危险的。尤其当调试者不是软件的开发人员的时候，就显得更加复杂了。我们认为软件演化应致力于尽可能限制那些可能被影响的变化本身的性质。

从软件演化的概念来看，软件演化和软件维护有着密切联系，但二者又有本质区别。软件维护是对现有的已交付的软件系统进行修改，使得目标系统能够完成新的功能，或是在新的环境下完成同样的功能，主要是指在软件维护期的修改活动。而软件演化则是着眼于软件的整个生命周期，从系统功能行为的角度来观察系统的变化，这种变化是软件的一种向前的发展过程，主要体现在软件功能的不断完善。在软件维护期，通过具体的维护活动可以使系统不断向前演化。因此，软件维护和软件演化可以归结为这样一种关系：前者是后者特定阶段的活动，并且前者直接是后者的组成部分。

软件演化的过程，也是通过修改软件的组成成分以适应变化的过程。在这一过程中，通常我们会尽可能复用系统已有的部分，降低演化的成本和代价。因此支持软件演化的技术通常也包含了软件复用的部分技术，如基于构件的开发、构件复用技术等等。与软件复

用一样，软件演化也可能发生在时间、平台、应用三维上。

（1）时间维：软件以以前的软件版本作为新版本的基础，适应新需求，加入新功能，不断向前演化。

（2）平台维：软件以某平台上的软件为基础，修改其和运行平台相关的部分，运行在新的平台上，适应环境变化。

（3）应用维：特定领域的软件演化后应用于相近的应用领域。

尽管软件演化和软件复用具有如上的共同特性，但它们研究问题的出发点和手段目的都是不同的。软件复用的目的是避免软件开发的重复劳动，提高软件开发的质量和效率。而软件演化着眼于整个软件的生命周期，研究如何在较低的开发代价的情况下，延长软件的生命周期，提高软件适应改变的能力。

1.1.2 软件演化的分类

软件演化可基本上分为两种：静态演化和动态演化。

（1）静态演化（Static Evolution）：是指软件在停机状态下的演化。其优点是不用考虑运行状态的迁移，同时也没有活动的进程需要处理。然而停止一个应用程序就意味着中断它提供的服务，造成软件暂时失效。

（2）动态演化（Dynamic Evolution）：是指软件在执行期间的软件演化。其优点是软件不会存在暂时的失效，有持续可用性的明显优点。但由于涉及状态迁移等问题，比静态演化从技术上更难处理。

动态演化是最复杂也是最有实际意义的演化形式。动态演化使得软件在运行过程中，可以根据应用需求和环境变化，动态地进行软件的配置、维护和更新，其表现形式包括系统元素数目的可变性、结构关系的可调节性和结构形态的动态可配置性。软件的动态演化特性对于适应未来软件发展的开放性、动态性具有重要意义。

按照变更发生的时机，软件演化可分为以下几类。

（1）设计时演化：设计时演化是指在软件编译前，通过修改软件的设计、源代码，重新编译、部署系统来适应变化。设计时演化是目前在软件开发实践中应用最广泛的演化形式。

（2）装载期演化：装载期演化是指在软件编译后、运行前进行的演化，变更发生在运行平台装载代码期间。因为系统尚未开始执行，这类演化不涉及系统状态的维护问题。

（3）运行时演化：发生在程序执行过程中的任何时刻，部分代码或者对象在执行期间被修改。这种演化是研究领域的一个热点问题。

显而易见，设计时演化是静态演化，运行时演化是一种典型的动态演化，而装载期间的演化既可以被看作是静态演化也可以看作是动态演化，取决于它怎样被平台或提供者使用。事实上，如果是用于装载类和代码，那么装载期演化就是静态演化，因为它其实是类的映射，而实际的装载代码并没有改变；另一种可能是增加一个层，允许在运行时刻动态的装载代码和卸载旧的版本，这样，通过连续的版本来更换代码，最后实现系统的演化，变更本身也可以被认为是动态的演化机制。

另外，演化可以是预设的和非预设的。

（1）预设演化是可以被开发人员所预见的演化。例如，插件技术就是一种允许程序员和维护人员在不更改应用程序核心部分的前提下扩展系统功能的机制。其优点是它可以提供依赖于动态软件演化下简单的装载和卸载机制，同时也可以提供依赖于静态演化下的API；缺点是在一些可能的变更中缺少适应性。

（2）非预设演化是指不能被开发人员所预见的那些演化。变更必须得到语言或者是执行平台（在动态演化中）的支持。其优点是包含了更多可能的演化，缺点是仍然没有一个被普遍接受的演化解决方案。

我们更加关注于非预设的动态演化，也就是说我们想要得到一个在运行期间不受演化可能性限制的演化系统。

1.2　设计时演化

设计时演化是目前在软件开发实践中应用最广泛的演化形式。设计时演化在软件编译前，通过修改软件的设计、源代码，重新编译、部署系统来适应变化。目前有多种技术用来提高软件的设计时演化能力，如基于构件的开发（CBSD）、基于软件框架（Framework）的开发、设计模式（Design Pattern）等等。

1.2.1　设计模式对设计时演化的支持

设计模式是对经过实践检验的、好的设计经验的提炼和总结，它强调了在特定环境下对反复出现的设计问题的一个软件解，侧重于解决软件设计中存在的具体问题。

设计模式解决的核心问题与设计时演化一致，要解决软件如何适应变化的问题。各种设计模式在实际上都从不同侧面封装了变化，有效地提高了软件的设计时演化能力，表 1-1 列出了部分设计模式和它们所封装的变化。

表 1-1　设计模式对设计时演化的支持

变　　化	设 计 模 式
实现算法	Visitor，Strategy
用户接口操作	Command
对象接口	Adapter
对象实现	Bridge
对象之间的交互	Mediator，Facade，Proxy
对象创建过程	Abstract Factory，Factory Method，　Prototype
对象结构建立过程	Builder
对象结构	Composite
遍历算法	Iterator
对象行为	State，Decorator
操作执行过程	Template
对象依赖关系	Publish-Subscribe

1.2.2 构件技术对设计时演化的支持

构件是用来构建软件系统的可复用的软件元素。构件通常表现为不同的形态，形态的差异体现在结构的组织方式和所依赖的软件开发方法。

按照构件的复用方式，构件可以分为以下两种。

- 源代码构件：这是最常见的构件形式，如结构化编程中的函数；面向对象编程中的类、类树；包（Package）等。
- 二进制构件：构件以编译后代码的形式提供，如 ActiveX 控件、EJB 构件、Java 类文件等。
- 按照应用领域，构件可分为以下三种。
- 通用基本构件：这是计算机系统的共性构成成分，如基本的数据结构、用户接口元素等，它们可以存在于各种应用系统中。
- 领域共性构件：这是应用系统所属领域的共性构成成分，它们存在于该领域的各个应用系统中。
- 应用专用构件：这是每个应用系统的特有构成成分。

基于构件的软件开发（Component-Based Software Development，CBSD）是一种利用可复用的软件构件建立应用系统的技术。这些构件由三部分组成：构件实现功能说明；构件的设计实现；构件的开发接口。

CBSD 的基本思路就是通过利用已有构件或自我组织开发的构件来组装应用程序。CBSD 基于 Software IC 的思想，强调软件开发的“组装”特性，即软件开发过程就是技术构件的构造、组织、选取、组装过程。可复用构件为有计划地进行复用提供了手段，是实现软件复用的基石，其生产和使用必须满足两个基本前提，即构件接口的标准化和构件的集成机制。目前从构件的定义和结构来说，已有比较成熟的构件规范，如 COM/DCOM、CORBA、EJB 等，这些规范为基于构件的软件开发、组装过程提供了完善的基础设施。

构件是现代软件系统的主要组成成分，是系统功能实现的载体，因此软件演化的基本单位就是构件演化，实际上构件演化也是软件演化的主要形式。支持构件设计时演化的主要技术如下。

1. 构件包装（Wrapper）

在系统开发过程中，经常出现集成的构件接口与系统设计需要的接口不兼容的情况，常见问题包括接口方法名称不一致、参数类型不一致，这就要求对原构件进行演化。对于这类接口演化问题，通常采用的做法是使用构件包装器（Component Wrapper）。构件包装器（如图 1-1）内封装了原始构件，同时提供了系统需要的接口，这样就解决了构件接口不兼容的问题。

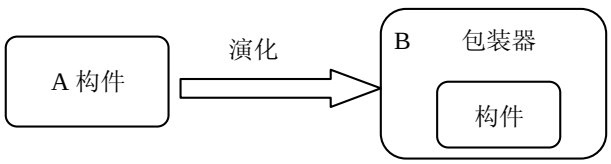


图 1-1 包装器将构件的接口 A 演化为接口 B

构件包装器本身不做核心工作，它只是进行简单的处理后，调用原始构件合适的方法，执行客户的请求，因此这是一种简单有效的构件演化的方法。

构件包装器不仅可以包装单个构件，而且可以包装多个构件，即同时封装多个构件。通过复合多个构件的功能，包装器可以提供更加强大的功能，此时包装器对于包装器的客户，实际本身也已经成为一个构件。一个封装了两个构件功能的包装器的实现如图 1-2 所示。

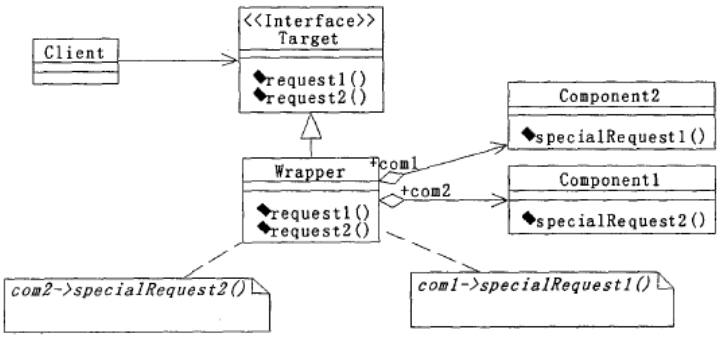


图 1-2　封装了两个构件的构件包装器的实现

2. 基于继承机制的构件演化

基于继承机制的构件演化，是一种广泛使用的构件演化技术。继承（Inheritance）和多态（Polymorphism）是对象建模技术适应变化的强有力武器。通常需要演化的构件，将自己可能需要演化的部分定义为虚函数，并提供默认实现。在新的环境中，当需要对原构件进行演化时，通过创建继承自原构件的子构件，并按照要求重新实现部分虚函数，就可以达到构件演化的目的。

如一个打印构件，负责实现系统数据的打印输出，其输出过程由三部分构成。

beforePrint()：初始化打印机。

print()：打印文档。

afterPrint()：负责打印结束后的后期处理。

对于不同的构件使用环境，打印机的初始化和后期处理可能都不相同。通过将beforePrint()和 afterPrint()定义为虚函数，针对不同的环境，开发不同的子构件，就有效实现了打印构件的演化，具体实现如图 1-3 所示。

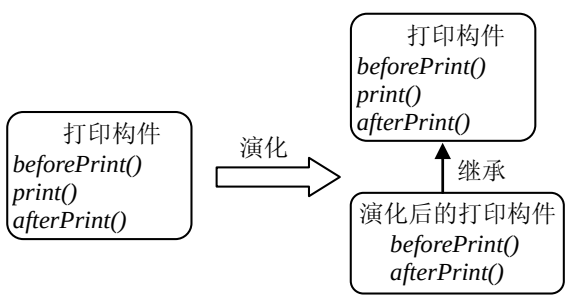


图 1-3　基于继承机制实现打印构件的演化

1.2.3 框架技术对设计时演化的支持

框架（Framework）可以看作一种大粒度的构件，其中包含若干个普通构件，定义了构件之间的协作规约，以实现一个大的系统功能。同时框架也是一个软件产品的半成品，它含可变和不变部分，通过对可变部分的定制得到不同的应用系统。对象框架的复用可以极大地降低应用开发的难度。通过复用一个好的框架可以得到一个设计简洁高效、稳定的系统。

框架是应用系统或子系统的可重用复用设计，在面向对象的系统中，框架通常由一系列的抽象类和具体类组成，系统的行为由抽象类的具体实现类的交互决定。

框架是比设计模式和构件更高层次、更大粒度的复用方式，成功的软件框架设计通常都包括了大量的设计模式（Design Pattern）和可复用构件（Reusable Component），开发者可以复用这些设计模式和构件开发自己具体的应用。

框架都是面向特定领域的，如用户接口设计或分布式通讯。表 1-2 列出了一些流行的软件框架。

表 1-2 一些面向特定领域的软件框架

应 用 框 架	简 介
JARS（Java Authentication and Authorization Service）	JARS 是一个基于 Java 的用户认证和授权安全框架，以可插拔的方式提供安全认证服务，独立于底层具体的安全认证实现
ACE（Adaptive Communication Environment）	ACE 是一个面向对象的基于 C++的分布式通讯框架，它提供了丰富的通讯模式和构件来开发跨平台的高性能分布式系统
Swing	Swing 是 Java 平台的 GUI 开发框架
MFC	MFC 是基于 Microsoft 的 WIN32 GUI 开发框架，它封装了 Win32 API，简化了 WIN32 应用程序的开发
QT	一个基于 C++的 GUI 开发框架，同时支持多个平台（Windows， Unix/Linux， Mac OS X， Embedded Linux）的应用系统开发

设计框架的主要目的是软件复用，框架是一种比构件更大粒度、更高层次的软件复用方式。从某种意义上，可以把框架看作一个大粒度的构件。和普通构件类似，使用环境的改变、软件技术的更新、系统功能的更新都要求框架处在不断的演化过程中。和普通构件不同的是，框架演化对兼容性有更高的要求，因为通常已经存在一些基于旧版本框架开发的软件系统，框架演化应该保证软件系统可以容易地移植到演化后的框架之上，以保护原有的投资。

软件框架按照对设计时演化支持方式的不同，可以分为白箱框架（White Box Framework）和黑箱框架（Black Box Framework）。

1. 白箱框架

白箱框架通常是框架开发经历的第一个阶段，这一阶段的主要特征是框架大量使用基于继承的构件演化技术。继承是框架支持设计时演化的主要手段，因为使用继承机制，需

要演化者对框架的内部机制有清晰的理解，因此演化者通常是框架的设计者。由于使用静态的基于继承的演化技术，白箱框架不支持框架的动态演化。

在白箱演化阶段，为了提高框架的复用能力，通常使用多种设计模式如 Template 和 Factory 等。同时白箱框架也会开发一些构件库（Component Library）供框架使用者使用。

随着基于白箱框架的开发应用系统规模的扩大，白箱框架的缺点逐渐暴露出来。

（1）白箱框架通过继承扩展框架的功能，为了改变一个简单的功能，框架需要派生子类，这样会导致生成太多的子类，增加了系统的复杂度，降低了系统的可理解性和可维护性。

（2）白箱框架要求使用者熟悉框架的结构，复用者熟悉框架的内部实现细节。继承导致构件之间较强的耦合，提高了框架的复用难度。

（3）继承是一种静态演化机制，不支持框架的动态演化。

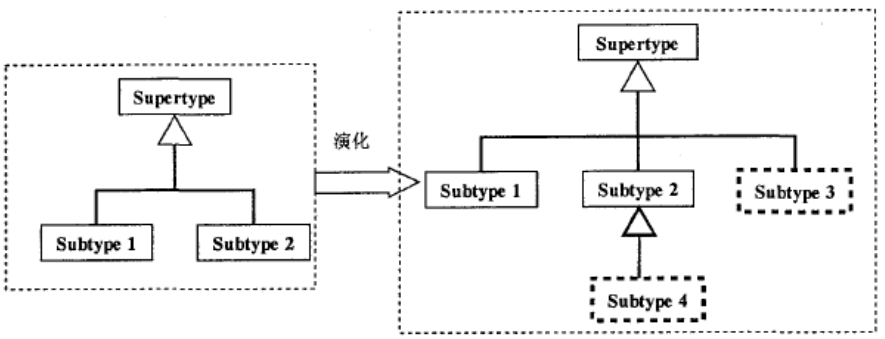


图 1-4　基于继承的白箱框架演化机制

2. 黑箱框架

黑箱框架是框架演化发展的下一阶段，这一阶段的主要特征是黑箱框架使用基于构件组合（Component Composition）的方式进行框架演化。构件组合方式和继承方式最大的不同是构件组合方式支持构件的运行时动态改变，支持框架的动态演化。

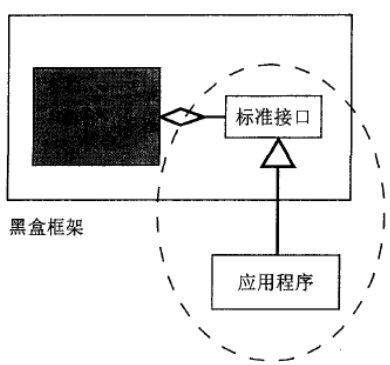


图 1-5　黑箱框架通过可插拔构件实现设计时演化

框架演化的主要方式通过可插拔构件（Pluggable Component）实现。可插拔构件指符合一定接口的构件，在接口不变的情况下，可以实现以不同方式实现的构件互换使用。基于可插拔构件技术，框架的演化者无需理解框架的整体实现细节，只需按照框架说明，实现符合一定接口的可插拔构件，将其静态或动态的插入框架中，就可实现对框架的定制。

1.3 装载时演化

装载时演化一般可以由动态的装载库和 Java 系统的类装载机制完成，也可以要求编译后的软件系统含有足够的系统的运行时信息，如由于 Java 类文件中包含了程序运行的元数据，通过修改元数据，就可以实现 Java 类的演化，包括增添新的方法，添加接口、数据成员等操作。

1. 动态装载

由于在一个编译语言中没有指定的自反模块来处理演化行为，从而导致程序调用困难，因此在一般的程序语言中动态的增加代码和使用这些代码变得十分困难。

早期的一些研究（如 Lisp）致力于创建模块系统来完成不同的程序之间的共享。这使得研究者将精力集中在根据需求装载和编译上。虽然也包含了动态装载机制，但关键是缺少演化，仅仅是简单的装载和卸载模块。

Dorward，Sethi 和 Shopiro 曾提出一个系统，可以向运行的 C++程序动态地增加代码。他们的系统依赖于链接已有的类所生成的类代码，给新载入的类赋予和它的父类相同的标志（Signature）。他们集中在保持类型安全（Type-safety）和得到一个轻便的系统。程序员使用一个预处理程序引入代码以及保持灵活的 C++代码。演化不仅可在动态装载 C++类中进行，同时也可以在程序员将一个对象变成子类的一个对象的时候进行，这是因为它是一个关于 Type-safety 的安全方向。

2. 利用延迟绑定机制进行动态装载

在 Java 中，动态装载被集成在语言中，它主要还有 ClassLoader 类实例来装载类。在 JAVA 语言中，类的查找和装载由 JVM（Java Virtual Machine）通过使用委托机制（Delegation）来进行。委托机制是指在 Java 平台中每个类都由一个类装载器（ClassLoader）查找和装载进内存（每个类装载器都有自己的装载路径，根据此路径来查找和装载类字节码）。除了系统类装载器（System ClassLoader）外，每一个类装载器都有一个父类装载器（Parent ClassLoader）。这样就使得在一个 JVM 中，所有的类装载器之间的关系呈树状，树根则为 JVM 的系统类装载器，如图 1-6 所示。Java 的类委托装载机制有效地解决了类装载中的安全和效率问题。在对构件化软件进行在线演化时，需要将类的实现体更换为新版本的类实现体。对于使用 Java 语言实现的软件系统，可以利用 Java 的上述类装载机制装载新版本的类来实现。

C++是另外一种允许动态装载和支持延迟约束的语言。C++和 Java 在类装载机制方面

有很多相似的特征。C++提供包的抽取，即将一系列相关的类打包，只让它的实例运行。目前还没有标准的抽取方法用来实现运行期间代码的装载和卸载，事实上，它需要由开发者在需要的时候定义。

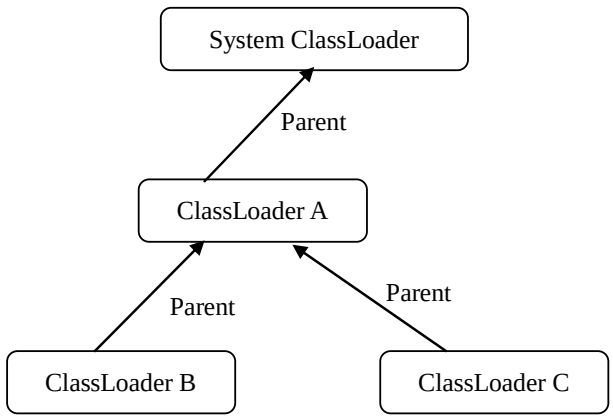


图 1-6　Java 类装载器的层次结构图

1.4　动态演化概念

1.4.1　动态演化、动态配置和软件的演化性

设计时演化技术，作为一种静态演化技术，其演化过程一般比较简单。一般情况下，根据需求开发新的功能实现或更新已有的功能模块，编译链接生成新的应用系统，最后部署更新后的软件系统即可。然而对于某些要求必须连续运行的系统，如空中交通管制系统、全球性的金融交易系统、工业控制系统、网络服务系统等，通过停止、更新和重启来实现维护和演化将导致不可接受的延迟、代价和危险。这就要求软件系统具备动态演化的能力，即可以在不停止系统运行的情况下实现软件的功能更新。

软件动态演化指的是软件进行在线变化并达到所希望形态的过程，其发生在软件运行期间，由一系列复杂的变化活动组成，包括动态更新、增加和删除软件系统的组成部分，动态配置系统结构等。由此可知，动态演化反映的是软件在线变更的过程和形态，而软件动态配置是一个和软件动态演化紧密相关的概念。动态配置是软件动态演化的主要手段和活动，满足系统对动态演化的需求，如图 1-7 所示。

无论系统多么复杂，从软件体系结构的角度出发，系统都由构件根据一定规则连接而成。根据系统的具体实现方式，抽象的构件概念可能对应着进程、对象或者真正的分布式构件，如 EJB 或 CCM 等。系统的构件组成，构件的物理分布及构件之间的连接关系等构成了系统的配置信息。动态配置的目标在于支持系统在运行时刻以尽可能小的代价从一个配置状态演化到另一个配置状态。因此采用动态配置技术改变系统配置时，系统中不受影响的部分仍然可以继续提供服务，从而减少了配置更改对运行系统的影响。

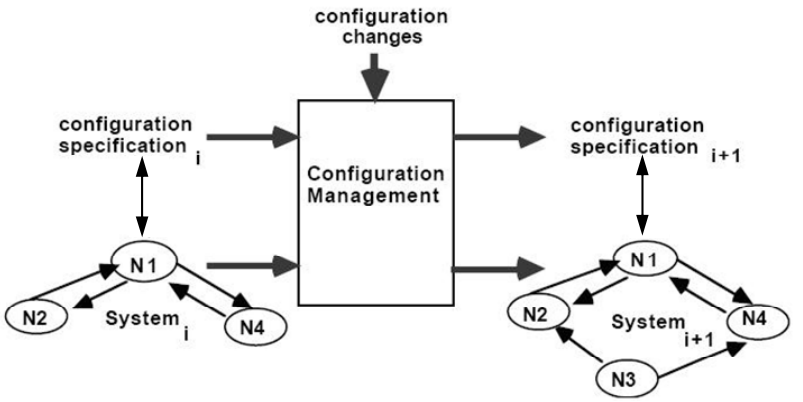


图 1-7 动态演化和动态配置

动态配置技术的研究始于二十世纪七十年代，最初是针对进程或对象进行的。但是由于进程或对象之间紧密耦合，而且涉及编译器和操作系统等底层细节，动态配置实现困难、功能有限、正确性不易保证而且性能较低。进入九十年代，分布式构件技术的出现为动态配置研究提供了良好的实验平台。构件模型的自包含性，其接口的规范性，构件之间、构件与平台间的低耦合性，消除了进程和对象对动态配置研究的种种约束。而分布式构件技术将构件开发、组装和部署相分离的开发模式，以及将业务逻辑和服务支持逻辑相分离的描述方式，不仅提高了构件的可复用性和系统的可扩展性，同时还强烈地推动了其自身对动态配置技术的需求。

另一个和软件演化相关并易混淆的概念是软件的演化性。软件的演化性是指软件系统本身具有的潜在的演化能力。杨芙清院士指出，构造性和演化性是软件的两个基本特性。客观世界本身是有结构的，软件是一个知识性的产品，所以软件应该具有更加严谨的结构性。要把客观事物的活动用软件表达出来，就必须把客观活动的结构性提炼出来。和其他客观事物一样，软件系统也具有演化性，并在不断的演化中促使技术不断发展。同样，技术的发展会反作用于系统，促进系统的演化。

一般说来，构造性高的软件其演化性高。具体说来，软件的演化性主要体现在四个方面：

（1）可分析性（Analyzability）：软件产品根据演化需求定位待修改部分的能力。具有良好演化性的软件应该容易分析、理解，从而可以根据变化迅速定位需要改变的部分。

（2）可修改性（Changeability）：软件产品实现特定部分修改的能力。具有良好演化性的软件可以方便地修改，以支持变化。

（3）稳定性（Stability）：软件产品避免软件修改造成不良后果的能力。

（4）可测试性（Testability）：软件产品验证软件修改有效性的能力。

自然地，软件的动态演化性是指软件系统本身具有的可动态演化的能力。一般来说，动态演化性高的软件要求其设计具有良好的构造性、更强有力的模型表示和更高的计算抽象层次。

1.4.1　动态演化分类

动态演化可分为两种基本类型：预设的和非预设的。在 Web 环境中，软件应用常常需要处理多种类型的信息，因此它们常被设计为可以动态下载并安装插件以处理当前所面临的新类型的信息；而分布式 Web 应用也常常需要增减内部处理节点的数目以适应多变的负载。这些动态改变都是软件设计者能够预先设想到的，可实现为系统的固有功能。另有一些必须对系统配置进行修改和调整的情况是直到系统投入运行以后才发现的，这就要求系统能够处理在原始设计中没有完全预料到的新需求。这种情况下一般需要关闭整个系统，重新开发、重新装入并重新启动系统。然而，为了进行局部的修改而关闭整个系统在某些情况下是不允许的（例如生命维持系统）或者代价太高。精心设计的动态演化技术可以在不关闭整个系统的前提下修改系统的结构配置，并尽量使未受影响的部分继续工作以提高系统的可用度。

从演化的粒度来看，有函数层次的动态演化、类/对象层次的动态演化、构件层次的动态演化、结构层次的动态演化、工作流层次的动态演化。从演化所使用的技术来看，有基于对象的动态演化、基于构件的动态演化、基于 agent 的动态演化，基于动态编程语言的动态、面向方面的动态演化、基于反射的动态演化、基于体系结构的动态演化等等。

另外，Carlos E. Cuesta 等人将软件的动态演化性分为三个级别：最低级别称为交互动态性，仅仅要求固定结构里的动态数据交流；第二级别允许结构的修改，即构件和连接件实例的创建、增加和删除，被称之为结构动态性；第三级别称为体系动态性，允许软件体系结构的基本构造（Infrastructure）的变动，即结构可以被重定义，如新的构件类型的定义。以这个标准衡量，目前的动态演化研究一般仅支持发生在第二个级别上的动态性。

1.5　软件动态演化技术的重要性

1.5.1　Internet 需要软件动态演化

Internet 无疑是 20 世纪末最伟大的技术进展，其普及和发展为我们提供了一种全球范围的信息基础设施，这个不断延伸的网络基础设施，形成了一个资源丰富的计算平台。以 Internet 为主干，各类局域网和无线网为局部设施，伴随着移动通讯技术的发展，再加上各种信息处理设备和嵌入设备作为终端，构成了人类社会的信息化、数字化基础，成为我们学习、生活和工作的必备环境。

Internet 的特点主要体现在下述三个方面。

（1）Internet 提供了广泛的连通性。这里的连通性不仅是指 Internet 物理意义上连接的地域范围，还包含人类生活的各个方面；Internet 通过开放的标准化技术，将规模庞大的各类计算设备连接为一个整体，不仅为信息共享、传播提供了基础，更可以提供巨大的计算能力，并将该能力融合到人类生活的各个方面。可以说，Internet 的这种无所不在的连通能力，已经深深影响了人类生活的各个方面，并将继续产生深远的影响。

（2）Internet 连通起来的整体信息系统是无统一控制的分布性系统结构，各节点具有高度的自治性，实体行为复杂并具有不可测性。一方面系统中相连的人、设备、软件实体和

局部环境具有丰富的多样性和异构性，适应方式也具有个性化和灵活性;另一方面，通过开放性的技术标准又实现了整体复杂的互连和互操作。由此，在基于标准的互连、互操作特性基础上，Internet 系统表现出了显著的动态特性。

（3）Internet 为人类海量信息的处理要求提供了基础平台，互连不仅使信息孤岛互相连接，共享范围得以扩充；更重要的是使得计算设备的能力实现了整合，能够在基本计算单元能力较弱的情况下，实现整体海量的计算能力。Internet 的发展为社会信息化孕育了巨大的创新空间，也为软件技术的发展带来了新的平台、新的模式和新的业务，并为软件产业发展提供了规模空前的基础设施。

总结 Internet 的上述特性，可以为 Internet 提炼出如下的基本特征：

- 无统一控制的“真”分布性；
- 节点连接的开放性和动态性；
- 节点的高度自治性；
- 人、设备和软件实体的多重异构性；
- 实体行为的不可预测性；
- 使用方式的个性化和灵活性；
- 网络连接环境的多样性。

考察当前的主流软件技术，可以发现目前的软件系统本质都是一种封闭、静态的软件体系框架，难以适应 Internet 环境下开放、动态和多变环境的需求。目前的软件结构存在如下问题。

（1）软件实体缺乏动态适应能力：目前，在构建系统的过程中，系统的基本功能和基本结构是按照系统所要解决的问题和问题领域的特性通过系统分析与设计逐步确定的，系统的设计目标在设计开始时确定，在设计过程中逐步细化，直至系统完成部署运行。按照这种模式开发的软件系统由于其实现目标和结构在设计时已经固化，因此对于系统运行时发生的变化难以动态适应。所有运行中的需求，必须进入新一轮的开发过程才能解决。为了适应 Internet 环境开放、多变和动态的特性，软件系统应该具有动态适应特性，系统开发者或维护者可以根据功能、环境的变化迅速改变系统的行为。

（2）软件实体缺乏自主反应能力：面向对象方法中的对象概念通常是静止和被动的，缺乏自主性，它难以在 Internet 开放、动态和多变环境下调整自身的目标和行为，以适应网络环境的变化和用户个性化需求。

（3）软件实体缺乏动态和复杂的协同工作能力：程序实体间传统的协同方式是过程调用（包括进程内调用和 RPC）和对象引用（包括进程内引用和远程引用）。由于其时间和空间耦合的特征，过程调用和对象引用通常只适合于程序实体功能固定、位置固定以及协同逻辑固定的静态和封闭世界，难以满足开放、动态和多变的 Internet 环境对时间或空间的紧密耦合或松散耦合等多种协同方式的需要。

传统的软件理论和开发方法，在解决这些问题上存在内在的困难。面向 Internet 的开放、动态和多变的运行环境，既是对传统的软件理论、方法和技术的挑战，更是一种机遇，促使传统的软件理论和技术发生重大变革，以适应这一变化，其变化可以预测的包括以下变化。

（1）软件基本模型的变化：软件基本模型的发展，表现为一个逐步追求更具表达能力、更符合人类思维模式、更具构造性和演化性的软件结构过程。目前，软件模型已发生了重要变

化，面向对象、面向构件的软件模型已成为主流。对基于 Internet 环境的网构软件系统而言，其基本的计算单元应该是分布自治、异构的“构件”或 Agent，而它们之间的“交互”则可能是从简单的消息传递到复杂的通信协议和协同模式。这种模型将由于所处平台的特性和开放应用的需求而变得比任何传统的计算模型更为复杂，开放、动态协同和演化是其基本属性。

（2）软件生命周期概念的变化：在传统的软件工程体系中，软件生命周期概念所强调的是从问题提出到软件提交的整个开发过程的重要性，而对于提交使用之后的软件变化过程往往只采用“软件维护”加以简单概括。这样一种软件生命周期概念对于处于静态封闭环境下的软件系统的开发是合适的，但对处于 Internet 开放、动态和多变环境下软件系统的开发则有局限性。因此，Internet 环境下软件的开发对运行阶段之后的演化过程的关注是必需的，甚至显得更为重要。其所呈现的是一个不同于传统生命周期概念的“大生命周期概念”，目前，围绕这种新的生命周期的相关理论、方法和技术基本还处于起步阶段。

（3）软件体系结构侧重点的变化:传统软件系统的体系结构更多地关注于软件实体，是以实体为中心的结构模式。虽然最近关于软件体系结构的研究已开始关注软件实体间的连接，但就体系结构的整体拓扑建模来说，仍是实体驱动的。这样的模式适合于封闭和静态环境的需求。在 Internet 开放、动态和多变的环境下，由于存在众多的基础软件资源，软件体系结构研究的出发点将从软件实体转向实体间的动态连接和协同，因此，从基于实体的结构分解到基于协同的实体聚合将会是软件体系结构研究的重要转变。

其他如在软件开发技术和软件开发环境方面，必然也会随之出现新的变化，如目前 Microsoft 的.NET 和新一代 J2EE 对 Web Service 的支持就可以看作面向 Internet 开发环境变化的一个前奏。

综上所述，Internet 及其上应用的快速发展与普及，使计算机软件所面临的环境开始从静态封闭逐步走向开放、动态和多变。软件系统为了适应这样一种发展趋势，将会逐步呈现出自主、反应、演化、协同和多态等特性。针对软件系统呈现出的新的自然特性和当前软件理论、技术和方法的发展趋势及研究热点问题，本书重点论述了软件系统的动态演化理论和应用研究。

1.5.2　动态演化性是网构软件的基本特征

由于软件系统所基于的计算机硬件平台正经历从集中封闭的计算平台向开放的 Internet 平台的转变，软件系统作为计算机系统的灵魂，随着其运行环境的演变也经历了一系列的变革。目前，面向网络的计算环境正由 Client/Server 发展为 Client/Cluster，并正朝着 Client/Virtual Environment 的方向发展。那么，未来的基于 Internet 平台的软件系统又会呈现什么样的形态呢?

从技术的角度看，以软件构件等技术支持的软件实体将以开放、自主的方式存在于 Internet 的各个节点之上，任何一个软件实体可在开放的环境下通过某种方式加以发布，并以各种协同方式与其他软件实体进行跨网络的互连、互通、协作和联盟，从而形成一种与当前的信息 Web 类似的 Software Web。Web 不再仅仅是信息的提供者，而是各种服务（功能）的提供者。由于网络环境的开放与动态性，以及用户使用方式的个性化要求，从而决定了这样一种 Software Web 并不能够像传统软件那样一蹴而就，它应能感知外部网络环境的动态变化，并随着这种变化按照功能指标、性能指标和可信性指标等进行静态的调整和动态的演化，

以使系统具有尽可能高的用户满意度，这样一种新的软件形态被称为网构软件。

网构软件是在 Internet 开放、动态和多变环境下软件系统基本形态的一种抽象，它既是传统软件结构的自然延伸，又具有区别于在集中封闭环境下发展起来的传统软件形态的独有的基本特征。

（1）自主性：网构软件系统中的软件实体具有相对独立性、主动性和自适应性。自主性使其区别于传统软件系统中软件实体的依赖性和被动性;

（2）协同性：网构软件系统中软件实体与软件实体之间可按多种静态连接和动态合作方式在开放的网络环境下加以互连、互通、协作和联盟。协同性使其区别于传统软件系统在封闭集中环境下的单一静态的连接方式；在像 Internet 这样的计算环境下，软件实体可以实现交互模式的动态改变，软件实体间的协同方式不再是静态的，而是根据运行环境和服务质量需求动态地发生变化，从宏观上观察，软件系统的系统拓扑关系在提供最好的可用性的目标指引下，处在不断的变化中。

（3）反应性：网构软件具有感知外部运行和使用环境并对系统演化提供有用信息的能力。反应性使网构软件系统具备了适应 Internet 开放、动态和多变环境的感知能力。

（4）演化性:网构软件结构可根据应用需求和网络环境变化而发生动态演化，主要表现在其实体元素数目的可变性，结构关系的可调节性和结构形态的动态可配置性；演化性使网构软件系统具备了适应 Internet 开放、动态和多变环境的应变能力。

（5）多态性：指网构软件系统的效果体现出相容的多目标性，它可根据某些基本协同原则，在动态变化的网络环境下满足多种相容的目标形态。多态性使网构软件系统在网络环境下具备了一定的柔性和满足个性化需求的能力。

由此可见，对网构软件系统来说，系统提交之后的动态的自适应性和演化性是其基本特征。

1.5.3 追求动态演化能力是自治计算的目的

在很多关键的领域中，我们都需要软件能够感知环境的变化，并根据环境的变化改变自身行为，采取适应性动作以适应资源的可变性、用户需求的变化以及系统错误等。对于软件所具备的这种特性，我们通常称为软件的自适应（Self-adaptation）。为了实现一种自管理的计算模式，人们提出了自治计算概念，它是指一个自动调节以满足正在其中运行的应用需要的基础结构。它以具有模式的人体自治神经系统来命名。通常我们所说的一个自治计算系统是一个具有灵活性的系统，可以采取抢先的和后因子措施，用最小的人工干预来保证高质量的服务，类似于自治神经系统调节人体系统而不需要个体的有意识干预。

根据人们对自治计算系统所赋予的期望，它通常需要具备以下特性。

（1）自保护：一个自保护系统可以检测和识别敌对的行为，采取自治的动作保护自己、抵御入侵行为。我们通常可以通过预定义的策略来保护系统不被破坏，因此，自保护特性要求系统能够进行一种预设的动态演化。

（2）自优化：在一个自治系统中的自优化组件构件能动态地调整自己，用最小化的人为干预来适应端用户或企业的需要。调整动作包括基于负载平衡功能和系统运行状态信息的资源重新分配，以整体提升资源利用率和系统性能。

（3）自修复：自修复是从可以引起系统一些部分出问题的故障中恢复系统的能力。对

一个自恢复的系统来说，它必须能够借助第一次检测从失效部件中恢复，并且隔离有故障的部件，使它离线，修复和重新装入已修好的部件或替换部件投入服务，不会造成任何明显的全面中断。一个自修复系统也需要预报问题并采取动作防止可能会影响到应用的故障发生。为了保持系统正常运转和在所有时间都是可用的，自修复目标是所有的消耗最小化。

（4）自配置：安装、配置和集成一个大的、复杂的系统是具有挑战性的、很耗费时间和经常出错的，甚至对专家也是如此。自配置系统能够自动地适应动态变化的环境，环境中的系统组件构件包括可以动态增加到系统的软件组件构件和硬件组件构件，并且要求不中断系统服务和最小化人工干预。

从以上自治计算系统的基本特性来看，无论是自保护、自优化、自修复或者自配置，它们的最终目标都是要求能实现系统的动态演化。因此，追求动态演化是自治计算最本质的目的，同时也是最基本的出发点。

1.5.4　动态演化技术是网格计算的基础

随着人们日常工作遇到的商业计算越来越复杂，人们越来越需要数据处理能力更强大的计算机，而超级计算机的价格显然妨碍它进入普通人的工作领域。于是，人们开始寻找造价低廉而数据处理能力超强的计算模式，最终找到了一种既能满足大型计算的要求，同时也能进入普及应用的技术——网格计算（Grid Computing）。它主要有以下几个方面的特性。

（1）异构性（Heterogeneity）：网格可以包含多种异构资源，包括跨越地理分布的多个管理域。构成网格计算系统的超级计算机有多种类型，不同类型的超级计算机在体系结构、操作系统及应用软件等多个层次上可能具有不同的结构。

（2）可扩展性（Scalability）：元计算系统初期的规模较小，随着超级计算机系统的不断加入，系统的规模随之扩大。网格可以从最初包含少数的资源发展到具有成千上万资源的大网格。由此可能带来的一个问题是随着网格资源的增加而引起的性能下降以及网格延迟，网格必须能适应规模的变化。

（3）可适应性（Adaptability）：网格中具有很多资源，资源发生故障的概率很高。网格的资源管理或应用必须能动态适应这些情况，调用网格中可用的资源和服务来取得最大的性能。与一般的局域网系统和单机的结构不同，网格计算系统由于地域分布和系统的复杂使其整体结构经常发生变化；网格计算系统的应用必须能适应这种不可预测的结构。

（4）结构的不可预测性：动态和不可预测的系统行为。在传统的高性能计算系统中，计算资源是独占的，因此系统的行为是可以预测的。而在网格计算系统中，由于资源的共享造成系统行为和系统性能经常变化。

（5）多级管理域：由于构成网格计算系统的超级计算机资源通常属于不同的机构或组织并且使用不同的安全机制，因此需要各个机构或组织共同参与解决多级管理域的问题。

网格的引入增强了计算能力，然而其要解决的一个关键问题就是需要远程调用和控制某台计算机的计算资源与存储资源，以及对网络流量的调节。对于开放多变的网络环境而言，这无疑是一大挑战。纵观网格计算平台的异构性、可扩展性、可适应性和结构的不可预测性等特性，它们都需要一种动态演化技术为其提供支持。因此，动态演化技术作为网格计算的基础提出，有着其特殊的作用和意义。

第 2 章　动态演化基础

2.1　基 本 原 理

2.1.1　动态演化过程

一般的动态演化过程模型如图 2-1 所示。根据系统设计 $Design_i$，实现并部署应用系统后，即可得到相应的系统配置信息 Configuration $Information_i$。通过动态配置，系统从当前的配置 Configuration $Information_i$ 增量式地在线演化到一个新的配置 Configuration $Information_{i+1}$。

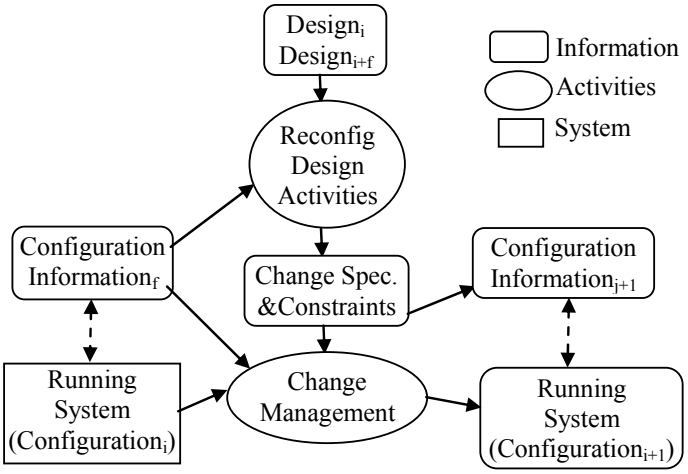


图 2-1　动态演化过程

一次完整的动态演化过程分为三个阶段：动态配置触发、动态配置算法生成、动态配置实施。

为实现系统功能更新或 QoS 管理，动态配置被触发。触发动态配置的主体可能是系统管理员，也可能是系统自身。在实际应用中，更普遍的情况是人与系统一起决定是否触发以及如何触发动态配置。无论以何种模式触发动态配置，都将生成对动态演化意图的描述。动态演化者希望通过实施动态配置而达到的目标称为动态演化意图。动态演化意图分为基本动态演化意图和复杂动态演化意图。基本动态演化意图包括：构件删除、构件添加、构件替换、构件迁移、连接建立、连接删除、连接重定向，以及构件属性设置。复杂动态演化意图由基本动态演化意图组合得到。支持应用系统实现动态演化意图的平台被称为动态

配置平台。触发动态配置时，可能直接向动态配置平台提交动态演化意图，也可能仅提交一个新的系统设计，如 $Design_{i+1}$。对于后者，动态配置平台将通过分析系统的当前设计和配置信息，以及目标设计信息，自动抽取出动态演化意图。但是，动态演化意图本身可能与系统正常运行时所需满足的约束条件相冲突。为杜绝这种情况的发生，动态配置平台需要根据系统约束对动态演化意图本身的合法性进行验证。

动态配置平台根据动态演化意图和系统正确性约束生成动态配置算法。动态配置算法描述了实现动态演化意图的具体步骤。在动态演化意图本身合法的前提下，动态配置算法必须保证系统在动态演化期间和动态演化后处于正常运行状态。在实施动态演化的过程中，系统中的部分构件和连接必定会被影响而无法正常工作。但对于整个系统来说，除了在性能上有所降低外，系统整体行为的逻辑正确性仍然是被满足的或符合预期的。这就是在动态演化期间系统处于正常运行状态的含义。系统处于正常运行状态的性质被称为系统一致性。动态配置必须保证系统一致性。

动态配置平台依据动态配置算法，通过监控构件的行为和状态，实现动态演化意图。由于在动态演化的实施过程中，系统仍然处于运行状态，因此必须尽可能提高动态配置的性能，减少对系统的影响。另外，动态配置算法所保证的系统一致性的内容和程度，以及算法本身设计的合理性也影响着实施动态配置的性能。

动态配置实施的结果必须保证：动态演化意图最终被应用到运行系统上；应用系统仍然可以正确运行；动态演化的约束条件都得到了满足。

在动态演化过程中，有三个需关注的方面。

（1）应用系统配置信息及其变化的描述：只有对当前系统配置信息有详细的了解，才能清晰地描述动态演化意图。无论是系统对动态配置的自动触发、动态演化意图根据系统设计的自动生成，还是动态演化意图的合法性验证，都依赖于对应用系统和配置变化信息的形式化描述和分析。而对应用系统和配置变化信息的描述越完整、越精确，系统一致性就越容易得到保证，动态配置的正确性和性能也就越高。

（2）动态配置性能提高：为了支持动态配置的实现，势必对应用系统的原有运行平台进行扩展。而这部分对平台的功能扩展有可能对系统正常运行时的性能产生影响，应尽量减少这部分的性能损耗。

（3）系统一致性保证：是否保证系统一致性，是衡量动态演化正确性的标准。因此，如何保证系统一致性是动态演化研究和实现中的首要问题。

2.1.2　语言、模型和平台

对动态演化的控制是软件开发者历来追求的目标。为支持软件的动态演化性，人们主要从语言、模型和运行环境三个方面进行研究。

在程序语言的层次上，引进各种机制以支持软件动态演化，例如动态装载技术允许增加代码到已运行的程序中，延迟绑定是在运行时而不是编译时决定类和对象的绑定。Java hotswap 允许在运行时改变方法：当一个方法终止时这个方法的新版本可以有效地替换旧版本，在类层次上代码的二进制兼容被支持。Gilgul 语言也允许更换运行时对象。但程序语言层次上的动态演化机制仅局限于函数、类方法和对象等小粒度的替换，只支持预设的有

限变更，变更由事件触发。

Peterson 等人使用一个基于高阶的类型程序语言来进行运行时变更。这个技术要求程序员预测程序中可能发生变化的部分，并把这部分以函数的形式组织以封装变更。开发者把变化策略和变化范围也编写在源代码中，因此允许在运行时对程序进行细粒度的控制。但是由于变化策略没有从执行计算的源代码中独立出来，单独地更改应用行为变得困难，在大规模系统中管理变更成为很复杂的事情。Polylith 是一个基于软件总线的分布式程序环境，允许异构分布式应用系统的结构变化。在 Polylith 中，结构重配置可发生在指定的运行时刻，但局限于应用源代码层次上。Derra 程序环境支持事件触发的重配置机制，缺陷是仅支持预设的动态演化。

Gorlick 等人表达了一个基于数据流的方法 Weaves。Weaves 是由并发执行的工具片断（tool fragment）构成的网络，工具片断之间以数据对象为中介进行通信。一个工具片断是一个小的软件构件，它按照一个步骤顺序，执行一个单一的、良定义的功能。工具片断消耗对象作为输入，产生对象作为输出。通过附加到“队列”上的“端口”，它把一个对象转换成一个不同的对象。队列把不同工具片断间的通信按照顺序缓存和同步起来，工具片断通过端口连接到队列上。每一个工具片断作为一个独立的线程共享内存。对象在流动中所经过的端口和队列都是被动的，只有工具片断是主动的，它从端口接收对象，调用对象所实现的方法，进行该工具特定的计算，并把对象传递下去。这种机制使得在不扰乱对象流的前提下可以进行运行时重配置，用户使用一个交互式的图形工具可视化地直接重配置 Weaves。但是，Weaves 当前并不提供任何机制进行运行时的一致性检查，也没有显示表达变更策略的方式。

Kramer 和 Magee 提出了一个结构化的分布式系统动态演化方法。在这个方法中，一个配置由使用双向通信链接互连的处理节点构成。当系统运行期间变更发生时，配置管理器强迫受变更直接影响的节点以及其邻接节点进入“静止”状态。在“静止”状态，节点不能发起通信以确保变更期间受变更直接影响的节点的运行状态不变。变更策略以一种声明式语言定义，重配置管理器负责解释和执行变更策略。但是这种方法也不解决演化过程中行为观察一致性问题。

软件体系结构概念的产生是试图在软件需求与软件设计之间架起一座桥梁，着重解决软件系统的结构和需求向实现平滑过渡的问题，软件体系结构已经作为一个明确的文档和中间产品存在于软件开发过程中。可见，软件体系结构技术的产生源于软件开发的需求，原是一个供软件开发人员使用的设计概念和技术。由于软件体系结构清晰地描述了构件及其相互关系和整个系统的框架，自然地，将软件开发中的软件体系结构应用于动态演化也成为一种考虑。

软件体系结构对动态演化的支持，首先体现在把动态配置功能集成在 ADL 中，即动态 ADL 的研究，如 Darwin，Rapide 语言和 Dynamic Wright 等。

动态 ADL 的思路主要是在 ADL 中专门设计用于预设演化的策略表示机制。例如 Darwin 语言运用 π 演算给出体系结构的语义，提供延迟实例化（Lazy instantiation）和直接动态实例化（Direct dynamic instantiation）两种技术支持动态体系结构建模，允许事先规划好的运行时的构件复制、删除和重新绑定；Rapide 基于偏序事件集（Partially Ordered Event Sets）对构件的计算行为和交互行为进行建模，允许在 where 语句中，通过 link 和 unlink

操作符重新建立结构关联；Dynamic Wright 使用标签事件（Tagged events）技术，提供了运用 CSP 对动态体系结构建模的方法。这些语言的动态管理机制中，都要求体系结构运行时的变化必须事先可知。

C2 是一种基于构件和消息的体系结构风格，为体系结构的演化提供了特别的支持。C2 的动态管理机制对体系结构运行时的变化不加限制，专门提供了体系结构变更语言 AML（Architecture Modification Language)。在 AML 中定义了一组在运行时可插入、删除和重新关联体系结构元素的操作。如 addcomponent、weld 等。但 C2 没有严密的形式化基础，不能根据系统的计算行为和状态决定是否实行体系结构动态变化，以确保系统完整地演化。

软件体系结构对动态演化的支持，也体现为以体系结构为核心的应用模型和软件框架的研究。例如，J.Dowling 等人设计的 K-Component 框架元模型通过提供体系结构元模型和适配契约（adaptation contracts）来支持系统动态重配置。适配契约显式地表达适配逻辑，而不将其固化于程序语言或支撑平台之中，从而使得适配逻辑可编程和动态地修改。K-Component 元模型的缺陷是不能严格和全面地表达体系结构的行为语义和结构语义。Walter Cazzola 提出由拓扑元对象和策略元对象形成体系结构反射元模型，但也仅支持预设的演化。

国内北京大学开发的 PKUAS 系统采取了基于运行时软件体系结构（Runtime Software Architecture，简称 RSA)的软件维护与演化方法。通过 RSA，系统的运行状态与行为以体系结构的形式展现，且操纵该体系结构视图能导致运行系统进行相应改变，这种因果关联通过反射式软件中间件实现。他们通过扩展传统的体系结构描述语言 ABC/ADL 描述 RSA，并使之具备继承设计阶段体系结构所富含语义的天然能力。但由于 ABC/ADL 缺乏构件行为和交互的形式化描述，没有解决构件替换时的可观察行为的一致性问题。为了让应用系统能够灵活地动态演化以适应底层因特网计算环境和用户需求的变化，南京大学也提出了一种动态协同架构。该架构引入内置的 RSA 对象来解耦系统中的各个构件，并通过该对象从体系结构的视角来重解释构件之间的引用和交互。这样就把体系结构这一抽象概念具体化为可直接操控的对象，从而可以利用面向对象程序设计语言的继承和多态等整套机制，导出一种面向体系结构的动态演化技术。但这种内置 RSA 增加了应用系统开发的复杂性，也缺乏严格的语义定义，不支持行为分析和一致性检测。

目前一些运行时系统扩展机制已经出现在现代操作系统（Unix 和 Windows 系统中的动态连接库）和构件对象模型（CORBA 和 COM 的动态对象绑定）中。这些机制通过允许在运行时动态的定位、加载和执行新的构件，使得系统可以不需重新编译便可进行演化。

通过标准化运行级构件的规约，依靠构件运行平台（中间件平台）提供的基础设施，使软件在构件层次上的动态演化成为可能。中间件中具有的如命名服务、反射技术和动态适配机制等允许为运行态构件的动态替换和升级提供支撑，推动了软件的动态演化发展。命名服务就是给构件实例提供一个名称，以便客户通过这些名称来获取构件实例。对工业标准构件 EJB 和 CORBA 构件等的引用都可以通过中间件平台的命名服务进行。同一构件的标识可以被映射到多个构件实例，从而根据具体情景可以将某一名字的构件引用导向到不同的构件实例。反射技术是系统的一种自描述（self-representation）和自推理的技术，它提供了关于自身行为的表示，这种表示可以被检查和调整，且与它所描述的系统行为是因果相连（causally connected）的。因果相连，意味着对自表示的改动将立即反映在系统的实

际状态和行为中，反之亦然。将反射性引入中间件能够以可控的方式开放平台内部的实现，从而提高中间件的定制能力和运行时的适应能力。动态适配机制中比较著名的是 CORBA 提供的动态接口服务：动态调用接口 DII 和动态骨架接口 DSI。前者支持动态客户请求调用，而后者支持将请求动态指派（Dispatch）给构件。因此，软件构件化技术使得软件具有良好的构造性，软件演化的粒度更大；中间件技术则为基于构件的软件动态演化提供了坚实的基础设施和方便的操作接口。

2.1.3 动态演化要解决的关键问题

如上节所述，软件动态演化技术还存在一系列的问题没有得到根本解决。在操作系统、分布式对象技术和编程语言（如 Lisp）中提供的运行时更新能力，存在很多不足。这些机制不能保证运行时改变的一致性、正确性以及运行时改变的可控性。变化管理对于进行有效的运行时系统改变具有关键意义。变化管理可以帮助确认需要变化的系统边界，帮助制定有效的运行时系统改变策略和方案，并控制变化过程以维护系统的一致性。如果缺乏变化管理，对系统进行运行时改变引入的风险可能大大超出关闭和重启系统导致的损失。

引起软件演化的原因是多方面的，如基础设施的改变、功能需求的增加、高性能算法的发现、技术环境因素的变化等等。所以，对软件演化进行理解和控制显得比较复杂而又困难。软件动态演化涉及四个方面：

（1）What，变更什么，变更的对象和粒度大小以及变更的结果；

（2）When，变更的时序属性，什么时候变更，变更的频度和历史；

（3）Where，变更发生的地点以及由谁触发变更；

（4）How，如何变更，包括变更的类型，过程和方法。

其中的核心问题是第 4 个方面。在软件动态演化的过程中，应该具备一定的方法和机制来保证和维护应用的正确性，这包含如下几层意思。

（1）能够预先推导变更的结果及其影响范围。在发生软件变更之前，对“变更之后软件是否适应了需求？变更后的软件是否符合应用约束？软件的全局性属性如安全性、可靠性是否受到影响？影响的结果有多大？”这些都应该有合适的机制进行自动的预测和评估，并决定是否进行软件演化。

（2）具有灵活的演化策略定义和处理机制。综合考虑和协调动态演化的诸多因素，给出动态配置的完整方案。不仅能表示和处理预设的演化，也能应对非预设的演化。

（3）对于软件组成成员的替换，能够保证替换前后成员的外部行为的一致性。因为软件系统的各部分相互协作和相互通信，软件的一个组成成员的功能执行可能需要另外成员的配合来完成，每一个成员都对和它进行协作的成员有一个期望的交互方式和行为约束。这意味着在替换软件成员时，不仅要使得它们的接口保持兼容，而且它们的可观察的外部行为也要保持一致性。

（4）具备控制变更过程的手段，以保持应用的完整性。动态演化的全过程都可以得到监视和控制，保障演化完整进行。例如在变更过程中发生了错误，可以采取措施进行回滚，撤销刚做出的变更。

（5）实时、较准确地对变更前后状态进行迁移的机制，以维持演化期间的上下文一致

性。当软件执行到某一断点时，从系统配置上撤换下来，它保持一定的状态信息。当该构件重新连接进入系统时需要从撤换时的断点继续开始执行。

2.2　系统一致性

是否保证系统一致性，是衡量动态演化正确性的标准，因此如何保证系统一致性是动态演化研究和实现中的首要问题。由于系统一致性刻画的是系统的正确性，体现为动态演化在保证系统正确运行时所需满足的约束条件。而应用系统在语义和实现方式上的差别，都会导致约束条件的不同。因此系统一致性的具体内容与应用系统密切相关。我们只能对各个系统共有的一致性提供保证。目前抽取出的公共一致性包括以下六种：相互一致性、本地一致性、构件状态一致性、应用状态一致性、引用一致性和结构一致性，其中相互一致性和本地一致性统一为行为一致性。由于结构一致性一般由应用系统本身及动态演化意图正确性来保证，因此这里我们只对其他五种系统一致性进行讨论和分析，研究保证各种系统一致性的方法。需要强调的是，保证系统一致性的前提是：动态演化者必须保证动态演化意图的语义正确性，即动态演化意图本身不能与系统正确运行时所需满足的约束相冲突。

2.2.1　系统一致性分类

通过对各动态演化方法提出的一致性约束的本质进行分析，根据一致性约束所关心的具体内容，并借用现有研究中对一致性的命名，我们将系统一致性分为以下六类。

（1）相互一致性（Mutual Consistency）：在系统运行过程中，构件间存在大量交互行为。如果在构件 A 响应构件 B 的请求时删除构件 A，将使构件 B 处于无法获得应答的异常状态，破坏了系统一致性。因此，动态演化必须保证构件间交互行为的完整性，这种约束被定义为相互一致性。

（2）本地一致性（Local Consistency）：除了交互行为，构件还存在本地行为，如读写文件的行为。本地行为也对构件状态造成影响。对本地行为的随意中断，也可能引发系统异常，如打断读写文件操作所引发的系统 I/O 异常。因此，动态演化必须保证关键的本地行为的完整性，这种约束被定义为本地一致性。

（3）构件状态一致性（Component State Consistency）：在动态演化过程中，构件间传递状态的情况广泛存在，如构件迁移、替换等。动态配置平台必须保证构件状态被正确收集，并可根据应用需求进行适当转换，从而正确初始化接收状态的构件，这就是构件状态一致性。

（4）应用状态一致性（Application State Consistency）：应用系统可能存在系统断言，对系统中一组构件的状态进行了约定。例如，在基于令牌环结构的系统中，所有构件只能拥有一个令牌的约定。动态演化不能破坏这种全局的状态约束，这就是应用状态一致性。

（5）引用一致性（Reference Consistency）：构件通过持有目标构件的引用，向其发送请求。当由于动态配置的实施而导致目标构件引用被改变时，如构件迁移，动态配置平台必须保证所有发向目标构件的请求不会由于持有过时的引用而无法到达目标构件，此为引用一致性。

（6）其他一致性：除了上述五类一致性外，为保证系统在动态演化期间和动态演化后正常运行，还存在其他一致性约束如结构一致性，又如，在向容错系统中动态添加构件的一个副本时，必须保证多个构件副本间的状态一致性。实际上，系统一致性刻画了系统的正确性，表现为动态配置平台为保证系统正确运行而必须满足的约束条件。而系统在语义和实现方式上的差别，都会导致约束条件的不同，进而导致系统一致性内容的千差万别。

在上述六类一致性中，“其他一致性”包含了大量不可预期的，而且与系统的具体语义和实现紧密相关的一致性约束。由于一致性内容的多样性和不可预测性，任何一个动态演化方法都不可能满足应用系统的所有一致性要求。另外，从系统的复杂性、可靠性和可用性的角度来说，动态配置平台也不能为所有一致性约束提供支持。因此，动态演化方法通常是通过假设动态演化意图合法、假定演化者正确使用动态配置平台提供的机制来保证“其他一致性”的满足。例如，在假设动态演化意图合法的前提下，结构一致性得到保证。

在假定演化者正确使用动态配置平台提供的机制的前提下，演化者在添加冗余的构件副本时，除了利用动态配置平台对构件添加功能的支持外，还必须结合使用容错系统本身提供的日志和恢复机制，从而保证多构件副本间的状态一致性。在上述前提下，动态演化方法只考虑如何保证相互一致性、本地一致性、构件状态一致性、应用状态一致性和引用一致性。

现有一致性研究中对一致性定义的混乱主要体现在相互一致性上。相互一致性最早由Goudarzi 方法提出，要求动态演化必须保证在构件间的交互过程结束后，参与此次交互过程的各构件都到达一致性的状态。由于动态演化方法目前还缺乏获取和分析应用系统语义信息的能力，因此只能通过保证构件交互行为的完整性来保证各构件进入一致性状态，所以我们将相互一致性明确定义为对构件交互行为完整性的保证。在保证构件交互行为完整性时，现有动态演化方法在粒度和方式上存在不同选择。在粒度方面，可通过保证事务或请求的完整性来保证构件交互行为的完整性。在方式上，可通过等待构件完成即将到达的请求或直接阻塞到达构件的请求来保证保证构件交互行为的完整性。因采用不同的粒度和方式组合，导致现有动态演化方法在保证相互一致性的程度方面存在较大差异，也造成不同的一致性命名。

2.2.2 行为一致性

针对现有一致性研究中存在的相互一致性和本地一致性独立定义不合理的问题，本节首先提出了行为一致性的概念。行为一致性的提出，不仅统一了相互一致性和本地一致性的定义，为严格保证构件交互行为和本地行为的完整性提供了支持，同时也简化了系统为保证构件行为完整性而进行的处理。

1. 行为一致性的定义

构件在运行时刻通常会存在大量正在进行的行为，包括构件发送请求、接收应答的交互行为以及构件读写文件等本地行为。行为对系统状态产生影响。对行为的任意中断，可能导致系统处于不一致的状态，破坏系统一致性。例如，删除一个正在响应请求的构件，将导致发出该请求的构件工作异常。中断构件的写文件操作，也将导致系统 I/O 异常。因此，动态演化必须保证包括了构件交互行为和构件本地行为在内的构件行为的完整性，称

之为行为一致性（Behavioral Consistency，BC）。为了定义应用系统中需要保证完整性的行为序列的边界，引入事务概念。由于行为一致性体现为对事务完整性的保证，因此首先分析在保证事务完整性时必须满足的约束，然后给出行为一致性的严格定义。

构件为实现某意图启动一个事务。事务并非仅包括构件交互行为，而是可以由任意构件的任意行为组成，包括构件交互行为和构件本地行为。启动事务的构件被称为事务启动构件，它明确事务何时终止。事务参与构件通过响应请求参与事务，在此过程中还可能向其他构件发送新的请求。事务参与构件在参与事务的过程中向其他构件发送请求的行为，属于它所参与的事务的一部分，而不被视为一个新启动的事务。事务代表了系统从一个一致性状态向另一个一致性状态迁移的过程。而在事务执行的过程中系统状态是不一致的。因此，事务中的行为序列不能被打断。动态演化必须保证系统中所有事务的完整性，从而保证行为一致性。在保证行为一致性时，我们无需关注事务的具体组成，无论它是构件交互行为还是构件本地行为，只要保证事务的完整性，即可保证构件行为的完整性，从而统一了相互一致性和本地一致性，避免了将构件交互行为和本地行为完整性问题分开考虑而带来的复杂性。

事务的执行过程可被抽象为一个自动机。系统从一致性的初始状态出发。随着行为序列中各行为单元的执行，系统在状态间迁移，最终到达一致性的终结状态。所谓事务完整性，即指事务要么没有启动，一旦启动，则系统必须依据自动机的设计按照预定轨迹在状态间进行迁移，并最终到达预定的一致性终结状态。事务执行轨迹的改变，可能导致事务无法完成或进入异常状态。

实际上，在某些动态演化场景中，即使事务不能按照预定轨迹完成，系统也可以到达一个非预期的一致性状态，仍然满足系统的正确性要求，系统一致性不会被破坏。但是，为了判断在何种情况下对事务执行轨迹进行何种改变不会破坏系统一致性，需要对系统行为和动态配置行为进行形式化的描述和分析。而目前对系统形式化描述和分析的研究还远远没有达到实用程度，对动态配置的支持就更是不足。因而通用的动态配置平台尚无能力精确判断事务执行轨迹的改变对系统一致性所造成的影响，只能通过严格要求事务按照设定轨迹执行来保证系统一致性。

事务的完整性约束具体表现为事务原子性约束、事务语义合法性约束、事务保序性约束和事务封闭性约束。

定义 1　事务原子性约束（Transaction Atomicity Constraint，TAC）：事务的执行过程必须是原子的。

事务要么没有启动，一旦启动，则组成事务的所有行为单元都必须完成。

定义 2　事务语义合法性约束（Transaction Valid Constraint，TVC）：事务中各行为单元执行的语义必须符合事务预期。

动态配置的实施可能改变事务中行为单元执行的语义，导致后继行为单元执行的起始状态发生改变，甚至可能使后继的行为单元起始于一个错误的状态。在这种情况下，事务将无法按照预定轨迹完成，甚至可能根本无法完成。例如，实施构件替换意图时，原构件将被新构件所取代。即使新构件可接替原构件参与在动态配置前已启动的事务，完成由于动态配置的实施而被阻塞的请求，但是由于新、旧构件具有不同实现，因此新构件响应请求的结果可能并不符合事务对结果的预期，从而改变了事务后继的执行轨迹。因此，为保证事务正常完成，动态配置必须满足 TVC。

定义 3 事务保序性约束（Transaction Sequence Constraint，TSC）：事务中行为单元的串行执行关系不能被破坏。

事务中若干行为单元的执行顺序可能是并行，也可能是串行。对于并行执行的行为单元而言，相互之间不存在因果关系，因而孰先孰后，不会对事务造成任何影响。但是串行执行的行为单元间存在明确的因果先后关系。对串行执行顺序的改变，将导致事务执行的语义错误。因此动态配置必须满足 TSC。

定义 4 事务封闭性约束（Transaction Closing Constraint，TCC）：事务一旦启动后，则不能向其行为序列中加入新的行为单元。

实施构件添加和连接建立意图时，可能向已启动的事务中添加新的行为单元。新行为单元的加入，可能改变后继行为单元的初始执行状态，导致整个事务执行轨迹的变化。因此，动态配置必须满足 TCC。

满足 TAC 是保证事务完整性的基本要求。事务一旦被启动，TAC，TVC，TSC 和 TCC 将共同保证系统按照预定轨迹运行，并到达预期的一致性状态。因此，行为一致性 BC 被严格定义如下。

定义 5 行为一致性（Behavioral Consistency，BC）：BC＝TAC∧TVC ∧TSC∧TCC。

为保证行为一致性而定义的事务与数据管理系统中的经典事务定义并不冲突。因为前者用于保证系统不会由于动态配置的实施而失效，而后者则侧重于失效恢复。虽然系统在失效时可通过后者进行必要的恢复，但毕竟失效恢复的代价比较大，因此应通过前者尽量避免不必要的系统失效。倘若一个动作序列属于一个经典事务，则其必定属于我们所定义的事务。

对于任意构件 A 而言，系统中的事务分为四类：

（1）构件 A 启动的事务；

（2）由其他构件启动但需要构件 A 参与的事务；

（3）构件 A 不参与但受其状态影响的事务；

（4）构件 A 不参与且不受其状态影响的事务。

下面，我们以负载均衡系统为例，给出四类事务出现的场景。

在图 2-2 所示的负载均衡系统中，负载均衡构件 Balancer 记录了系统中所有服务器构件 Server 的引用和负载情况。客户 Client 为了获得 Server 提供的服务，必须首先启动服务器绑定事务，与 Server 建立绑定关系。Balancer 参与该事务，通过查询负载记录并根据一定的策略将一个 Server 的引用返回给客户，完成绑定。客户利用 Balancer 返回的 Server 引用，启动由 Server 参与的服务使用事务。在每台部署了 Server 的主机上，都有一个负载检测构件 Probe，负责收集部署在该主机上的各 Server 的负载情况，包括服务器的利用率以及该主机的 CPU 和内存等使用情况。Probe 定期启动需要 Server 和 Balancer 参与的负载收集事务，在收集到与其位于同一台主机的每个 Server 的负载后，报告给 Balancer，从而辅助其实现负载均衡。在负载均衡系统中，虽然 Server 没有直接参与服务器绑定事务，但是其是否存在的状态，却直接影响该事务完整性。倘若在删除一个 Server 构件后，Balancer 在服务器绑定事务中仍将该 Server 的引用返回给客户，则显然 Balancer 的这个绑定行为没有正确完成，违背了 TVC 约束，破坏了事务完整性。对于 Server 构件而言，服务器绑定事务即为第三类事务。

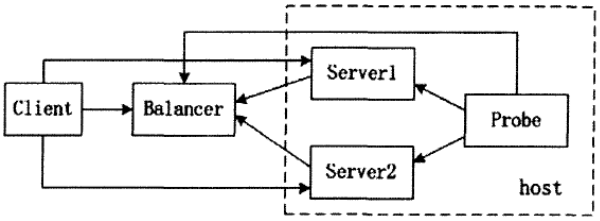


图 2-2　负载均衡系统

对构件 A 实施动态配置时，显然第四类事务的完整性不受影响。第三类事务与构件状态之间的关联关系，通常体现为应用系统对若干构件状态的全局约束。动态配置平台将通过保证应用状态一致性，保证这种全局约束的满足，从而保证第三类事务的完整性。例如，为避免在客户和不存在的 Server 之间建立绑定关系，负载均衡系统存在一个全局的状态约束，只有当 Server 构件存在时，Balancer 构件才能一一记录该 Server 构件的引用。为了满足这个全局的状态约束，在 Server 构件加入和离开系统时，必须对 Balancer 构件记录的 Server 构件的引用信息进行相应修改。关于动态配置平台如何保证应用状态一致性的问题，我们将在下一节详细分析。

综上所述，在保证行为一致性时，我们只需考虑如何保证第一、二类事务的完整性。

2. 系统假设

在介绍行为一致性保证方法前，本节说明行为一致性保证方法的系统假设。

系统假设 1：所有事务都必须在有限时间内完成。

倘若应用系统不满足此条件，则应用系统就没有向动态配置平台提供允许其实施动态配置的时机。动态配置平台为了不破坏事务完整性可能无限期地等待下去，无法完成任何动态演化意图。

系统假设 2：所有事务都必须独立完成，事务间不存在等待依赖关系。

在生产者－消费者系统中，producer 启动数据产生事务，产生数据并发送给 buffer consumer 启动数据消费事务，获取并使用 buffer 中的数据。由于 buffer 存放数据的空间有限，因而数据产生事务完成的前提是 buffer 空间未满。倘若 producer 在 buffer 空间已满时启动数据产生事务，则该事务必须等待 consumer 至少启动并完成一个数据消费事务后才能完成。在这种情况下，数据产生事务依赖于数据消费事务，无法独立完成，这就是所谓的事务间的等待依赖关系。

动态替换 buffer 时，为保证事务完整性，需要首先禁止 buffer 继续参与数据产生事务和数据消费事务，然后等待 buffer 完成已经参与的数据产生事务和数据消费事务后，才能开始实施动态配置。如果在替换前，buffer 空间已满而且在此之后参与了由 producer 启动的数据产生事务，则在成功禁止 consumer 启动数据消费事务后，该数据产生事务将永远无法完成。动态配置平台也将无限期地等待下去。

因而，动态配置平台在保证事务完整性时，必须明确事务间的等待依赖关系在何种情况下出现，并在动态配置过程中避免这种情况的出现，从而避免无限期的等待。但是事务间等待依赖关系出现的情况与应用系统的语义和实现方式密切相关。由于这类语义信息很难被准

确、完整地描述，而且避免这种情况出现的处理方式也与特定的应用系统所绑定，因此，通用的动态配置平台目前尚无能力针对此类极其个性化的情况，提出一个通用的方法解决保证事务完整性时的无限期等待问题。所以，必须要求应用系统中的事务不存在等待依赖关系。

3. 行为一致性保证方法

为保证行为一致性，构件在动态配置前必须进入静止状态。

定义 6　当构件满足如下条件时，构件处于静止状态：

（1）构件当前没有参与自行启动的事务；

（2）构件将来不会自行启动事务；

（3）构件当前没有参与由其他构件启动的事务；

（4）构件将来不会参与由其他构件启动的事务，包括即将启动和已经启动的需要构件参与的事务。

定义 7　当构件满足静止状态定义中的条件 1，2 时，构件处于被动状态。

处于被动状态的构件不存在自行启动的事务，只能被动参与由其他构件启动的事务。

定义 8　构件既可以参与由其他构件启动的事务，又可以自行启动事务的状态为主动状态。

定义 9　PS(Q)是构件 Q 的被动集合（Passive Set），由 Q 和所有可以启动需要 Q 参与的事务的构件组成。

定理 1　构件的被动状态是可达的。

证明：根据前提假设，系统中的任何事务都在有限时间内独立完成。即使参与事务的构件处于被动状态，也不会阻止事务在有限时间内完成。因为处于被动状态的构件只是不具备启动事务的能力，但仍然具有正常参与事务的能力。因此构件自行启动的事务必定在有限时间内结束，静止状态定义中的条件 1 被满足。另外，只要能够在有限时间内禁止构件将来自行启动事务，就可在有限时间内满足静止状态定义中的条件 2。因此构件的被动状态是可达的。

定理 2　当 PS(Q)中的所有构件都进入被动状态时，构件 Q 进入静止状态。

证明：根据静止状态定义，Q 进入静止状态必须满足 4 个条件。当 Q 进入被动状态时，条件 1，2 被直接满足。当 PS(Q)中除 Q 以外的其他构件都进入被动状态时，表明当前和将来都不存在需要 Q 参与的事务，条件 3，4 得以满足。定理成立。

定理 3　构件的静止状态是可达的。

根据定理 1 和定理 2，定理 3 显然成立。

定理 2 实际给出了采用等待方式驱动构件 Q 进入静止状态的算法：首先计算 PS(Q)。在驱动并确认 PS(Q)中的所有构件都进入被动状态后，Q 进入静止状态。通过驱动 PS(Q)中除 Q 以外的所有构件进入被动状态，保证其他构件不再启动需要 Q 参与的事务，并等待已经启动的此类事务的完成，体现了等待方式的处理特点。

采用等待方式驱动目标构件进入静止状态后，目标构件启动和参与的所有事务只可能处于两种状态：没有启动或者已经正常完成。此时对目标构件实施任何动态演化意图都不会破坏事务的完整性。因此，行为一致性强保证方法可在所有动态配置场景下严格保证行为一致性。

每个构件必须实现一个 passivateComp 方法。动态配置管理器（Dynamic Reconfiguration

Manager，DRM）在驱动构件进入被动状态时，调用该方法。构件在 passivateComp 方法中首先禁止将来自行启动事务，然后在确认所有已经启动的事务全部完成后，通知 DRM 自己已经进入被动状态。相对于 passivateComp 方法，构件还必须实现 activateComp 方法，恢复自行启动事务的能力。动态配置结束后，DRM 将调用相应构件的 activateComp 方法，驱动构件从被动状态回到主动状态，从而将系统恢复到正常运行状态。

动态配置的对象除了可以是单个构件外，还可以是由多个构件组成的构件模块。对于目标构件集合 TS(Target Set)来说，PS(TS) = U PS(i)。只有当 PS(TS)中的所有构件都进入被动状态时，模块中的各构件都进入静止状态，整个模块才进入静止状态。

4. PS 计算算法

在行为一致性强保证方法中，为使 TS 中的所有目标构件进入静止状态，必须驱动 PS(TS)中的所有构件进入被动状态。驱动构件进入被动状态，将禁止构件自行启动事务，对系统的正常运行造成较大影响。因此，精确计算 PS，避免将不相关的构件纳入 PS 中，是提高动态配置性能的关键。

PS(Q)由构件 Q 以及所有可以启动需要构件 Q 参与的事务的构件组成。一个构件是否可以启动需要另一个构件参与的事务，隐含在应用语义中。由于目前缺乏描述和分析应用语义的有效手段，PS 尚不能被精确计算。为了获得真实的 PS，虽然可以要求开发者直接为每个构件声明其 PS，但随着系统规模的增大，开发者的负担也呈指数级增长，而且这种方式又不足够灵活以适应系统结构的变化。因此，需要在适当要求开发者提供语义信息的基础上，根据系统的当前结构信息，动态计算出 PS 的替代集合，用于驱动构件进入静止状态。

定义 10　构件启动事务时发送请求的行为被称为主动发送请求（Initiate Request）。

定义 11　请求在响应过程中顺序经过的构件和连接信息构成请求路径。

根据 PS 的定义，若构件 $C \in PS(TS)$ 且 $C \notin ATS$，则 $\exists Q \in TS$，Q 位于 C 主动发送请求的请求路径上。因此，确定主动发送请求的请求路径是 PS 计算的关键。为定义构件间的各种连通关系，从而刻画主动发送请求的请求路径，首先将构件及由构件连接而成的应用系统定义如下：

定义 12　构件是一个四元组<S,I,R,IC>，其中：

（1）S(Send Request Port Set)是构件发送请求的有限端口集合。

（2）I(Initiate Request Port Set)是构件主动发送请求的有限端口集合。

（3）R(Receive Request Port Set)是构件接收请求的有限端口集合。

（4）IC(Implicit Connection)是构件内部在接收请求端口和发送请求端口之间存在的连接关系，$IC \subseteq R \times S$。若构件为响应从端口 p 接收的请求，而从端口 q 向其他构件发送新的请求，其中 $p \in R$，$q \in S$，则<p,q>$\in$IC。

IC 代表的构件端口间的连接关系隐藏在构件内部，因此被称为隐式连接。构件的 S 和 R 信息可从构件接口定义中获得。而 I 和 IC 属于语义信息，必须由构件开发者提供。随着形式化工具的发展，通过对构件源代码进行形式化分析，I 和 IC 信息也可被自动生成。

定义 13　由构件连接而成的应用系统是一个二元组<N, EC>，其中：

（1）N 是有限构件的集合。

（2）EC（Explicit Connection）是在不同构件的发送请求端口和接收请求端口之间存在

的连接关系。$EC \subseteq \bigcup_{n\in N} S_n \times \bigcup_{n\in N} R_n$，其中 S_n 代表构件 n 的发送请求端口集合，R_n 代表构件 n 的接收请求端口集合。如果构件 n 可通过端口 p 直接向构件 m 的端口 q 发送请求，其中 $n, m \in N$ 且 $n \neq m$，$p \in S_n$，$q \in R_m$，则<p,q>∈EC。

EC 刻画的构件端口间的连接关系，体现在系统的结构信息中，因此被称为显式连接。

我们通过图 2-3 的办公服务系统进一步说明上述定义。客户构件 Cl，C2 通过代理构件 A 使用邮件服务 ESV、打印服务 PSV 和传真服务 FSV。 C1 可启动邮件服务使用事务和打印服务使用事务，而 C2 只能启动传真服务使用事务。A 把到达 getl 端口的使用 ESV 的请求通过发送请求端口 emailA 转发给 ESV。而对于到达 get2 端口的请求，A 将根据发送请求的客户端构件，决定将其转发给 PSV 或 FSV。各构件的发送请求端口、主动发送请求端口、接收请求端口、隐式连接信息以及构件间的显式连接信息如图所示。

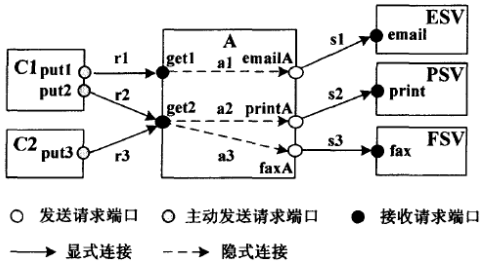


图 2-3 办公服务系统

我们以构件 C1, A 和 ESV 为例，将构件用四元组分别表示为：

C1≡<{putl, put2}, {putl, put2}, ϕ, ϕ>

A≡ <{emailA, print A, faxA}, ϕ, {getl, get2}, {< getl, emailA>，<get2,printA>，<get2, faxA>}>

ESV≡<ϕ, ϕ, {email}, ϕ>

办公服务系统用二元组表示为：

OFFICE ≡< {C1, C2, A, ESV, PSV, FSV},

{<putl, getl>，<put2,get2>，<put3, get2>，<emailA, email>，<print A, print>，<faxA, fax>}>

定义 14 如果构件 A 的发送请求端口 p 与构件 B 的接收请求端口 q 之间存在一条由构件发送请求端口和构件接收请求端口间隔组成的连通路径<s1,r2,s2,...,sm,rm>，并满足以下条件，则称端口 p 在结构级与端口 q 连通，构件 A 与构件 B 在结构级连通：

- $m > 1$
- $p = s_1$，$s_1 \in S_A$
- $q = r_m$，$r_m \in R_B$
- $\forall j \in \{1,\cdots,m-1\} <s_j, r_{j+1}> \in EC$

对于结构级连通而言，构件间连通路径的构造完全是显式连接的机械组合，代表了所有可能存在的请求路径。但在结构级连通的两构件间可能不存在任何交互关系。例如，在办公服务系统中，C2 与 ESV 之间存在连通路径<put3,get2,emailA,email>满足定义 14 中的条件。因而 C2 与 ESV 在结构级连通。但是，根据办公服务系统的语义，ESV 不会接收到任何直接或间接来自 C2 的请求。与 Q 在结构级连通的所有构件组成集合 ECS(Q)(Explicit

Connecting Set)。 ECS(ESV)={C1,C2,A}。ECS(TS)=∪ECS(i)。

定义 15　如果 A 的发送请求端口 p 在结构级与 B 的接收请求端口 q 连通，而且 p、q 间的连通路径<s1,r2,s2,...,sm,rm>满足以下条件，则称端口 p 在语义级与端口 q 连通，构件 A 与构件 B 在语义级连通：

- m=2，或者 m＞2 且 $\forall j \in \{2,\cdots,m-1\}$　$\exists n \in N < r_j, s_j > \in IC_n$

对于语义级连通而言，连通路径或者只包含一条显式连接，或者由若干显式连接和隐式连接间隔组成。由一条显式连接构成的连通路径，直接对应着相邻构件间存在的请求路径。通过加入隐式连接信息，由多条显式连接和隐式连接间隔构成的连通路径在构件内部的轨迹也对应了可能存在的请求路径，而不再是显式连接的机械组合。因而，与结构级连通相比，在语义级连通的两构件间的连通路径更接近真实的请求路径。在办公服务系统中，构件 A 不会为了响应构件 C2 的请求而向构件 ESV 发送请求，因此<get2,emailA＞∉ICA。虽然 C2 与 ESV 在结构级连通，但是两者间的连通路径<put3,get2,emailA,email>不满足定义 15 中的要求，因此 C2 与 ESV 不在语义级连通。这与真实系统中不存在从 C2 到 ESV 的请求路径是一致的。与 Q 在语义级连通的所有构件组成集合 ICS(Q)(Implicit ConnectingSet). ICS(ESV) ＝ {C1, A}. ICS(TS) = $\bigcup_{i \notin TS} ICS(i)$。

根据定义，ICS(Q) ⊆ ECS(Q), ICS(TS) ⊆ ECS(TS)显然成立。

定义 16　如果构件 A 的发送请求端口 p 在语义级与构件 B 的接收请求端口 q 连通，且 p、q 之间的连通路径<s1,r2,s2,...,sm,rm>满足以下条件，则称端口 p 与端口 q 主动连通，构件 A 与构件 B 主动连通：

（1）S1 ∈ IA

语义级连通关系，刻画了构件发送的请求可能经过的请求路径信息。在此基础上，主动连通关系则刻画了构件主动发送的请求可能经过的请求路径信息。在办公服务系统中，C1 与 ESV 主动连通，连通路径为<putl,getl,emailA,email>。与 Q 主动连通的所有构件组成集合 ACS(Q)(Actively Connecting Set). ACS(ESV)={C1}。ACS(TS)=U ACS(i)。根据定义，ACS(Q) ⊆ICS(Q)，ACS(TS) ⊆ICS(TS)显然成立。

定理 4　PS(Q) ⊆ACS(Q) ∪ {Q}

证明略。

综上所述，随着语义信息的逐渐丰富，ECS(TS), ICS(TS), ACS(TS)和 PS(TS)之间存在以下关系：PS(TS) ⊆ ACS(TS) ∪ TS ⊆ICS(TS)∪ TS ⊆ ECS(TS) ∪ TS 。

如果仅根据系统结构信息来计算 PS(TS)的替代集合，则计算出的结果只能是 ECS(TS)∪TS。因为，根据 ECS(TS)的定义，ECS(TS)中的构件都与目标构件在结构级连通，因而都具有启动目标构件参与的事务的可能性。在缺乏语义信息而无法对这种“可能性”进一步精确判定时，为保证正确驱动目标构件进入静止状态，只能驱动 ECS(TS)∪TS 中的所有构件都进入被动状态。但是通常情况下，ECS(TS)∪TS 中的构件个数远远大于 PS(TS)。例如，在办公服务系统中，PS(ESV)＝{ESV,C1}，而 ECS(ESV)∪{ESV}＝{ESV,A,C 1,C2}。当构件间连接关系比较复杂时，这种现象还将更加明显。因此，用 ECS(TS)∪TS 替代 PS(TS)虽然不影响以等待方式驱动目标构件进入静止状态的算法的正确性，但由于 ECS(TS)中包含了过多的不相干构件，扩大了对系统影响的范围，极大地降低了动态配置性能。因此在 PS 替代集合的计算过程中必须引入语义信息，从而使替代集合尽量接近真实 PS。

通过适度要求构件开发者提供构件语义信息，包括构件隐式连接信息和主动发送请求

端口信息，我们可计算出 ACS(TS)。与 ECS(TS)相比，ACS(TS)有效地减少了集合中不相关构件的个数，更接近 PS(TS)。ACS(ESV)∪{ESV}={ESV,C1}＝PS(ESV)。虽然 ACS(TS)∪TS 仍然是 PS(TS)的超集，但要计算出更接近 PS(TS)的替代集合，则需要构件开发者提供过于细节的语义信息。在办公服务系统中，存在一条从 C2 的主动发送请求端口 put3 出发到达 PSV 的连通路径<put3,get2,printA,print>，根据定义 16，C2 ∈ ACS(PSV)。但是根据办公服务系统的语义，C2 不会启动需要 PSV 参与的事务。为了在 PS 替代集合的计算过程中判断出 C2∉PS(PSV)，要求构件开发者必须详细描述代理构件 A 处理请求的语义信息，即 A 将把来自 C2 的请求转发给 FSV 而不是 ESV 的语义信息。这不仅增加了构件开发者的负担，而且对于某些应用来说这类语义信息很难描述。通用的动态配置平台也很难针对不同应用的特定情况进行统一的分析和处理。所以，通过在动态配置性能、动态配置平台的透明度和实现复杂度之间进行权衡，本文选择 ACS(TS)∪TS 作为 PS(TS)的替代集合。

下面给出 PS(TS)替代集合的计算算法。

定义 17　系统中所有具有主动发送请求端口的构件构成了主动发送请求构件集合 IRCS(Initiate Request Component Set)。

以办公服务系统为例，IRCS(OFFICE)＝{C1,C2}。显然，ACS(TS) ⊆IRCS。

根据 ACS 的定义 16，本文计算 ACS(TS)集合的思路是：针对 IRCS 中每个构件 C 的所有主动发送请求端口 p, p∈Ic，利用隐式连接和显式连接关系构造从 p 出发的构件端口连通树，从而判断是否存在一条从 C 出发到达 TS 中构件的连通路径满足定义 16 的要求。其中 Ic 为构件 C 的主动发送请求端口集合。结论为真时，C∈ ACS(TS)。PS 替代集合 RPS(Replacing PS)计算算法如图 2-4 所示。

```
RPS  PSComputing(TS)
     RPS=TS;
  T: foreach (C in IRCS－TS)
       foreach (p in Ic)
         root =p;                          //创建以p为根结点的端口连通树
         current =p;
         while(current ≠ null)
           if (current是发送请求端口)
             foreach (与current显式连接的接收请求端口p)
               if (p在端口连通树中从未出现)
                 将p作为current的子结点加入端口连通树;
           else
             if (current是TS中构件的接收请求端口)
               RPS=RPS∪{C};
               goto T;
             foreach (与current隐式连接的发送请求端口p)
               if (p在端口连通树中从未出现)
                 将p作为current的子结点加入端口连通树;
           if (current存在子结点)
             current = current .leftestChild;
           else
             current = getNextCurrent(root, current);     //end of while
    return RPS;

current  getNextCurrent(root, current)
    m=current.parent;
    n= m.nextLeftChild(current);
    if (n≠null)
      return n;
    else
      if (m≠root)
        return getNextCurrent(root,m);
      else
        return null;
```

图 2-4　RPS 计算算法

利用 RPS 计算算法，我们仍以办公服务系统为例，示意 RPS(ESV)的计算过程，如图 2-5 所示。初始情况下，RPS(ESV) ＝{ESV}。由于 IRCS(OFFICE) ＝{C1, C2}，因此需要分别以 C1 和 C2 的主动发送请求端口作为根节点构造端口连通树。Ic1＝ {putl, put2}。在以 C1 的主动发送请求端口 put1 为根节点构造连通树的过程中，当连通树中出现了 ESV 的接收请求端口 email 时，C1 被加入到 RPS(ESV)中，连通树的构造过程被立即终止。同时也无需再构造以 put2 为根节点的端口连通树。Ic2 = {put3}。在以 C2 的主动发送请求端口 put3 为根节点构造连通树的过程中，始终没有出现 ESV 的接收请求端口。因而连通树的构造过程最终由于没有新节点的加入而告终止。算法执行的最终结果是，RPS(ESV)={ESV, C1}。

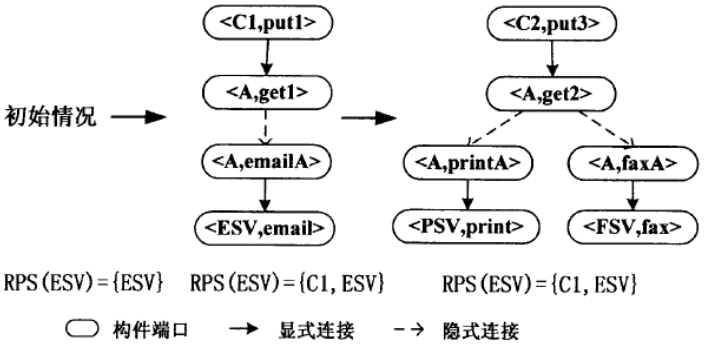


图 2-5　RPS(ESV)计算过程示意图

2.2.3　构件状态一致性

动态配置时，需要在构件间传递状态的情况广泛存在。状态可能从正常工作的构件传递给新加入的冗余构件，从而实现容错或负载均衡。状态也可能从替换或迁移前的构件传递给替换或迁移后的构件。收集并提供状态的构件被称为状态传递的源构件。接受状态并用其初始化自身的构件被称为状态传递的目标构件。状态一致性强调的是，必须将源构件正确收集的状态传给目标构件，而且目标构件通过直接继承该状态或者根据语义约束对其进行转化后正常运行。

在构件状态传递中，源构件和目标构件的对应关系可能是一对一、一对多、多对一或多对多。源构件需要收集什么状态，以及目标构件在接收到来自一个或多个源构件的状态后，是否需要以及如何进行状态转化等问题，都由应用语义决定。因此，每个构件必须自行实现 externalize 和 initialize 方法，分别负责状态的收集和初始化。根据演化者声明的在状态传递过程中源构件与目标构件的对应关系，DRM 将在适当时机调用源构件的 externalize 方法，获取收集的状态，并将其作为参数调用目标构件的 initialize 方法初始化目标构件。

通常，状态传递的源构件在收集状态前必须通过等待方式进入静止状态，从而保证收集的状态是稳定的、一致的，不含有完成一半的事务的状态。但是构件迁移时，由于迁移前后的构件完全相同，只要收集的状态是完整的，即使其中包含了仅完成一半的事务的状态，迁移后的构件也可以此为初始状态完成迁移前构件未完成的事务，并正常运行。因此构件迁移时，只需采用阻塞方式驱动源构件进入静止状态，即可开始状态收集。另外，构件在静止状态时收集的状态中不包含堆、栈和程序执行指针等难以收集和传递的执行状态

信息，而仅包括逻辑状态信息，需要传递的状态集合较小，而且状态易于被接收和转化。

为保证状态传递的目标构件在接收状态后能够正常运行，目标构件与源构件进入静止状态的方式必须保持一致。综上所述，静止状态是传递构件状态的合适时机。应根据具体的动态演化意图，选择合适的方式驱动构件进入静止状态。

2.2.4 应用状态一致性

构件状态的设定除了单纯与自身行为相关，还可能受到系统对状态全局约束的限制。应用状态一致性强调的是：在由于动态配置而导致全局状态约束被破坏时，必须重新设置构件状态，从而满足该约束。任何动态演化意图的实施都可能破坏全局的状态约束，都需要保证应用状态一致性。

目前在保证应用状态一致性时设置构件状态的方式有两种。在此以基于 CCM 实现的动态配置平台 StarDRP 为例，说明两种设置构件状态的方式。一种方式是，StarDRP 支持采用设置构件属性的方式直接设置构件状态。使用这种方式，要求构件状态必须体现为构件属性，而且可从外部明确指定。另一种方式是，StarDRP 利用 CCM 技术的 configuration-complete 机制在构件加入系统时设置构件状态，保证应用状态一致性。每个 CCM 构件根据需要实现 configuration complete 方法。该方法在构件从部署阶段进入运行阶段时被调用。构件通过该方法不仅可设置构件自身状态，还可通过与其他构件交互设置其他构件的状态，从而在构件加入系统时保证应用状态一致性。这种方式实现的状态设置可以十分灵活和复杂，并可同时设置多个构件的状态。当构件状态无法从外界指定时，这种方式更加有效。利用 CCM 构件技术的易扩展性，与 configuration-complete 机制相对称，StarDRP 引入了 run-complete 机制。每个 CCM 构件都有一个 run-complete 方法，构件根据需要自行决定是否实现以及如何实现该方法，该方法在构件离开系统时被调用。构件在该方法中设置系统中剩余构件的状态，从而保证应用状态一致性。

我们把为了保证应用状态一致性，在动态配置过程中被调用了属性设置方法、configuration-complete 方法、run-complete 方法的构件，称为系统状态设置构件。由于对构件属性的设置将改变构件状态，进而影响构件行为。构件在属性设置前开始参与而尚未完成的事务，可能在属性设置后由于行为的改变而无法按照预期继续完成，从而破坏事务的完整性。因此为了给构件状态的设置提供一个稳定的环境，保证行为一致性不被破坏，构件在属性设置方法被调用前必须通过等待方式进入静止状态。configuration-complete 方法和 run-complete 方法的调用时机分别在构件加入系统后和离开系统前。构件开发者必须正确实现这两个方法从而保证系统能够在构件状态设置后正确运行。

我们用 SSS(System State Setting Set)代表系统状态设置构件集合。SSS 又被细分为 ASSS 和 MSSS。其中 ASSS(Attribute-mode State Setting Set)中的构件采用设置构件属性的方式设置构件状态，MSSS(Method-mode State Setting Set)中的构件采用在构件加入或离开系统时调用 configuration-complete 或 run-complete 方法的方式设置构件状态。由于目前缺乏描述和分析应用语义的有效手段，因而当系统对构件状态的全局约束由于动态配置的实施而被破坏时，哪些构件属于 ASSS 以及这些构件的状态应被如何设置都必须由演化者指定。对于在动态配置过程中加入或离开系统的构件而言，构件是否属于 MSSS，即构件在

加入系统后和离开系统前是否需要调用 configuration-complete 方法和 run-complete 方法，在构件开发阶段就已经确定，所以由构件开发者声明。

2.2.5　引用一致性

构件间最常用的一种通讯方式就是通过持有目标构件的引用向其发送请求。在进行构件迁移和替换时，目标构件的引用都会发生变化。如何保证所有发向目标构件的请求不会由于持有过时的引用而无法到达目标构件，就是引用一致性解决的问题。

为了保证引用一致性，需要从两个方面采取措施：更新引用注册中心的引用信息；通过透明的请求重定向机制，将基于旧引用发送的请求正确地定向到新目标构件。在动态配置平台 StarDRP 中，利用了 CORBA 的 Locationee Forward 机制来实现透明的请求重定向。

对引用注册中心的引用更新，实际是通过删除旧引用、添加新引用两个步骤实现。但是在引用的更新过程中，对引用的使用可能是不间断的。如果在旧引用被删除后，而新引用尚未添加前，构件要求从引用注册中心获取目标构件的引用，则将引发不存在该目标构件引用的系统异常。因此，在引用更新过程中，如果构件有使用引用的需求，则旧引用的删除和新引用的添加必须是一组原子性的动作。对应于 CCM 构件平台，构件在被创建后，通过建立构件间的连接，将构件引用注册到其客户端构件用于存储引用信息的数据结构中。相应的，通过删除构件间的连接，注册的引用信息被删除。将连接删除和连接创建过程组成的引用更新过程称为连接重定向，并将连接重定向实现称为原子性操作。连接建立、删除以及非原子性的连接更新过程导致系统异常的情况如图 2-6 所示。

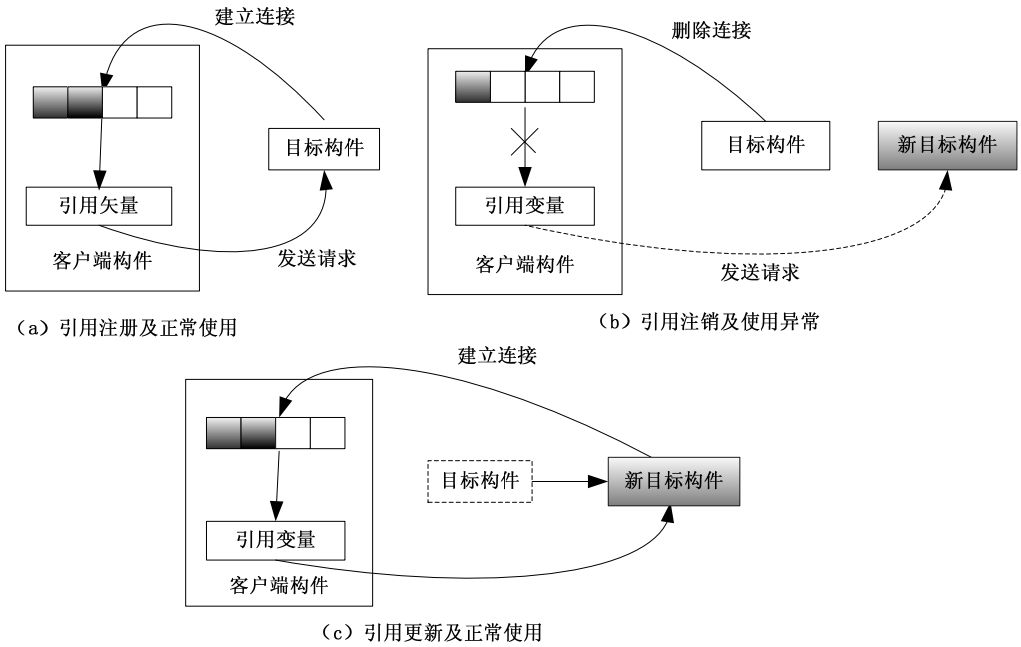


图 2-6　引用注册、注销及更新

2.3 状态迁移方法

从当前态切换到目标态过程中运行状态的迁移是一个重要的问题。一般来说，当软件执行到某一断点时，从系统配置上撤换下来，它保持一定的状态信息。当该构件重新连接进入系统时需要从撤换时的断点继续开始执行。

2.3.1 状态检测

为保证系统一致性，在动态配置过程中，对构件实施的操作可能存在前提条件，要求构件处于特定状态。构件状态检测机制则用于检测构件是否到达特定状态。根据第五章设计的动态配置算法，需要检测的状态包括，构件何时完成所有自行启动的事务，以及构件何时响应完毕所有指定类型的请求。相应的，构件状态检测机制由检测这些特定状态的子机制及辅助机制构成。

在构件自行实现的 passivateComp 方法中，在禁止构件自行启动事务之后，需要等待构件完成所有已经自行启动的事务。为辅助构件检测何时到达此状态，我们扩展容器，提供了构件完成所有自行启动事务的检测机制。作为该机制的辅助机制，我们首先提供了事务边界记录机制，记录构件当前启动的事务个数，具体实现方法是在容器的内部接口中加入 beginTran，commitTran 两个方法，并要求构件在启动事务前和完成事务后分别调用这两个方法。而这两个方法将分别对事务个数进行加 1 和减 1 的操作。基于事务边界记录机制，容器只需检测构件当前启动的事务个数是否为 0，即可判断构件当前是否完成所有自行启动的事务。倘若构件当前尚未完成所有自行启动的事务，则检测构件当前参与的事务个数是否为 0 的过程还将反复进行。此检测过程可以主动方式或被动方式反复进行。在主动方式中，容器将间隔固定的时间启动检测过程。在被动方式中，每当 commitTran 方法被调用，容器就被触发启动一次检测过程。由于很难确定合适的状态检测间隔时间，因而我们选择了被动方式。线程间的同步机制被用于实现触发式的状态检测。

采用阻塞方式驱动构件进入静止状态时，需要在驱动构件进入被动状态并阻塞后继到达构件的请求后，等待构件响应完毕已接收的应用类型请求。边界构件所在的容器实现 passivateClient 时，需要在阻塞或抛弃来自外部实体的请求后，等待边界构件响应完毕已接收的来自外部实体的应用类型请求。因此，我们提出了“构件响应完毕所有指定类型请求的检测机制”。为了区分配置类型请求、发自内部实体的应用类型请求和发自外部实体的应用类型请求，我们首先利用 CORBA 的截获器实现了请求类型标识机制。对于由控制边界内 CCM 构件发出的应用类型请求，容器将在其请求上下文中加入“COMPONENT”标识。DRM 和部署基础设施将把其发出的配置类型请求标记为“CONFIGURATION”。来自外部实体的应用类型请求或者不存在请求上下文，或者请求上下文中没有任何标识。通过这种方式，三类请求被区分开。类似于事务边界记录机制，我们通过对容器进行扩展，在目标构件接收请求和返回应答的相应时机，判断请求类型，并分别记录构件当前正在响应的发自内部实体的应用类型请求个数和发自外部实体的应用类型请求个数，此为构件响应请求信息记录机制。请求类型标识机制和构件响应请求信息记录机制的实现原理如图 2-7 所示。

当 DRM 要求容器检测构件何时响应完毕所有指定类型请求时，容器只需判断当前正在响应的相应类型的请求个数，若为 0 则即刻返回。若不为 0，则容器将同样采用被动方式进行触发式检测。构件每响应完毕一个请求，容器即被触发启动一次检测过程。

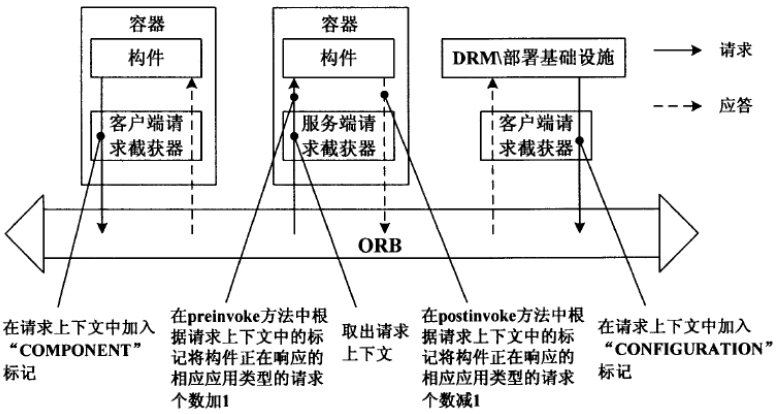


图 2-7　请求类型标识机制和构件响应请求信息记录机制实现原理

2.3.2　状态迁移方法分类

动态演化中对运行系统状态的处理目前有三种方式。

（1）不考虑状态问题，直接进行系统的切换。这种方式优点是快捷简单，缺陷是可能引起系统的部分失效或运行异常，可用来针对非关键系统或状态对外界无关紧要的情况。

（2）状态的直接迁移，直接把系统旧版本的状态写入新版本中。这种方式不仅要求知悉旧版本和新版本的实现细节，而且要求运行时环境能操纵系统实现体的内部。例如通过要求构件开发者根据需要自行实现 externalize 和 initialize 方法，并由 DRM 在合适的时机调用，提供了构件状态外表化机制和构件状态初始化机制，如图 2-8 所示。由于构件状态的表现形式多样，因此使用 any 类型封装具有不同内容和实际类型的构件状态。

```
valuetype ConfigValue
{
        public string name;
        public any value;
};
typedef sequence<ConfigValue> ConfigValues;

ConfigValues* externalize();

void initialize(const ConfigValues& states);
```

图 2-8　构件状态外表化和初始化机制

（3）状态的间接迁移，系统旧版本输出状态，新版本读入接收该状态。在有些情况下，输出状态可以写入磁盘。这种方式存在两个前提或说假定：一是状态要具有标准的表示格

式和可共享的语义，以使新旧版本能够无障碍解析和理解；二是系统中参与的构件必须具有输出状态或输入状态的接口。这两个前提限制了直接迁移方式的应用。

对于静态类型语言来说，因为不同版本之间的通信严格限制于相同的类型之间，因此不同的应用域和类装载器的存在使得变更变得复杂。这样的体系结构造成了变更的瀑布式影响，降低了保持大部分系统或者构件不变的可能性。事实上，一个不能更改的类型 A 引用一个可以更改类型的 B，由于版本障碍，A 将不能使用 B 的新版本。更改所有引用了 B 的新版本的 A 的对象将产生瀑布影响，因为所有引用了 A 的对象也需要更改。对此，Sato 和 Chiba 提出了一种方法，它放宽了在特定环境下的版本限制。还有一种技术从旧版本上改变未发生变更的类型的装载器到新版本的装载器上。这两种方法都需要虚拟机的支持。

如果不依赖于反射和元数据协议，几乎所有的方法都是不可能存在的。不论实实在在的状态迁移算法是怎么产生的，不同版本的源代码都没有提供状态迁移所需要的信息，因此需要增加元数据协议。无论是从旧版本中提取状态还是在新版本中插入都需要反射操作（调整是非预测的，运行着的版本都没有必要的提取功能）。

2.3.3 一种构件间状态迁移的元模型驱动方法

1. 状态描述

状态迁移隐含了需要对一个构件内部状态进行适当的描述，该点需要重视，因为当我们必须找到两个不同的状态之间映射关系的时候，选择描述就有其自身的含义。下面讨论两种可能的描述方式：抽象状态和基于状态的实现。

抽象状态：在这种方法中，建立状态的抽象描述，并且所有实现的具体程序状态都可以映射到这个抽象描述上。在不同的旧版本要更新到最新版本时，该方法具有明显的优点，因为它只需要定义一种映射关系（也就是抽象状态和新的实现之间的映射）。此外，也不用知道旧构件版本的实现信息，因为所有的相关信息都是从抽象状态描述中得到的。有些系统已经在某些特定的领域内应用了这种技术。

基于状态的实现：在这种方法中，从具体的实现版本中提取出来的状态必须能够被迁移函数解析执行。因为每一个构件都有它自己的状态描述，因此状态迁移函数依赖于新旧两个版本的构件。这种状态描述方法的优点是它不需要构件的开发者提供构件状态到抽象状态的映射。

抽象状态看上去是很完美的解决方法，但是在实际中不太实用，不能应用到广泛的领域中去。我们提出了一种混合方法，它使用基于对构件状态描述的实现作为一个整体，但是对于具体的结构却使用抽象的方式（例如，用一个抽象的集合来表示列表、哈希表或者是向量集）。

2. 元模型驱动的状态迁移

在下面描述的步骤中，构件的源代码需要提供必要的功能来允许构件在运行期间被替换。这些功能包括输出状态的函数和输入当前构件版本状态的函数。必要时还要增加代码

来处理诸如将端口从构件的旧的版本移动到新版本之类的管理功能。整个过程如下所述：

（1）在这一步中，新版本的构件源代码被解析并且创建一个元模型。这个元模型是对构件源代码中隐含内部结构的逻辑描述，用这样一个方法来访问构件代码表示的语义信息。类似地，由旧的（当前活动的）构件版本创建另外一个元模型。

（2）有了这两个元模型，当前的构件版本和新的构件版本都是可以被确定了的。SEESCOS 构件系统要求一个构件在替换之前是不活动的，因此，构件的状态可以定位到构成构件的一系列类的实例上。在有些情况中，对文件或者外部数据库的引用也在构件中有一定的描述。这些引用被看作是类的其他成员变量。该方法中的一些相关问题在这里不做详细的介绍。

（3）这一步是要确定两个版本中哪些变量进行了通信。这个过程不可能自动完成，因为并不是所有的语义信息都可以从构件的源代码提取得到。考虑这样一个例子：假如一个构件存储了一个三角形，第一个版本将这个三角形描述成三个点的队列，而新的版本可能理解为三个角和一个边，这二者之间的语义等价则不能从源代码中提取得到。因此，在这个过程中就需要有人的干预。与具有相同名字的变量之间值传递的方法不同，状态的元模型描述允许更多的智能化的算法。

（4）将算法的结果和用户提供的信息结合，新版本构件的元模型被更改，产生允许构件输出或者输入状态的方法。另外做一些额外的调整来保证替换的无误执行，例如重命名类和初始化端口（在基于 Java 的构件系统中不允许装载和已经存在的类重名的类）。

（5）从更改过的元模型得到新版本构件的源代码。此时的构件含有在运行期间实现替换所必要的信息。

整个过程可以用图 2-9 描述。

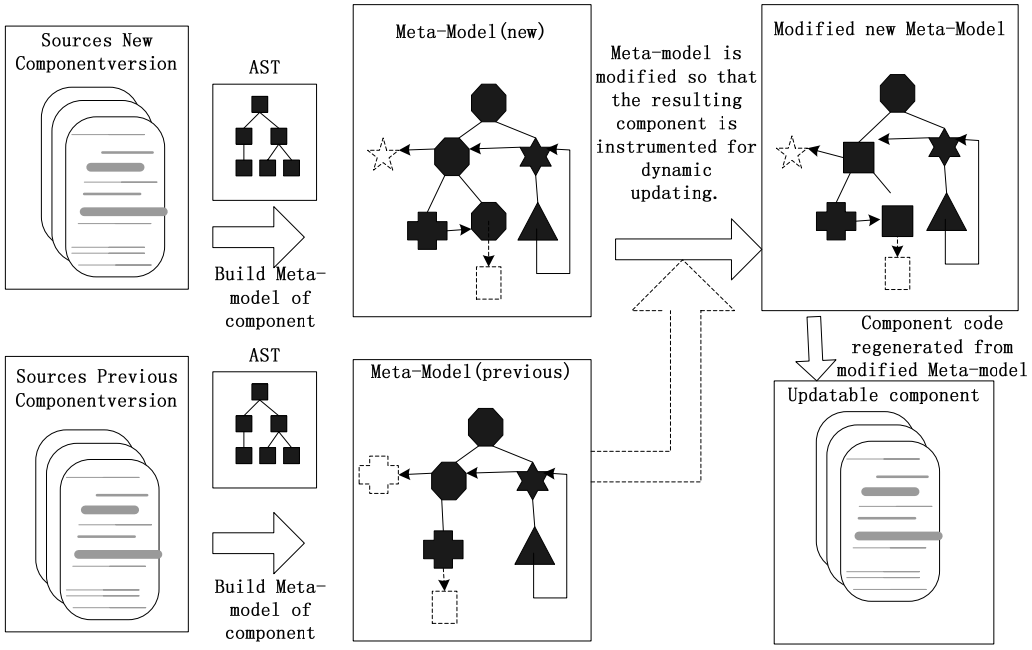


图 2-9　元模型驱动的状态迁移过程

上面的五步中，构件做好在每一个独立阶段被执行完之后进行动态更新的准备。隔离的思想用来避免添加不必要的构件替换功能，保持构件的原始设计不与非功能代码混淆。

使用元模型来完成构件之间的转换是元模型驱动的状态迁移的一个重要方面，它还需要进一步的验证。元模型可以被看作是在构件代码之上的一个抽象的层次，它的主要优点是可以直接使用源代码，从中较容易提取复杂的信息，用于生成比较智能化的映射。另外，元模型允许根据底层的代码独立地定义状态映射。两种类型的模型描述如下：

全局模型：这个模型包含源代码所有的信息，可以生成全部的代码，因此它不依赖于实现语言。和复杂的信息提取类似，全局模型可以轻易地将变更应用到许多不同的类上。全局模型还可以包括一些在源文件中没有提到的信息，例如 JDK。

局部模型：局部模型表示了构件源代码的一部分，它不可能生成全部的代码和隐藏了的、在分析程序结构用不到的那些细节问题（这些问题对生成状态迁移代码是必需的）。另外，局部模型可以生成多种语言。

因为局部模型不是完整的，在使用中需要对步骤 5 做一些适当的调整；步骤 4 需要对源代码做直接修改，步骤 5 则不需要。

第 3 章　设计可动态演化的软件系统

3.1　构造性和演化性

构造性和演化性是软件的两个基本特性。客观世界本身是有结构的，软件是一个知识性的产品，所以软件应该具有更加严谨的结构性。要把客观事物的活动用软件表达出来，就必须把客观活动的结构性提炼出来。和其他客观事物一样，软件系统也具有演变性，并在不断的演化中促使技术不断发展。同样，技术的发展会反作用于系统，促进系统的演化。

如何使软件模型具有更强的表达能力，如何使软件系统更能适应动态变化的环境，在一定意义上来讲，都紧紧围绕了软件的本质特征——构造性和演化性。一般来说，演化性高的软件其设计具有良好的构造性、更强有力的模型表示和更高的计算抽象层次。

汇编语言直接使用指令作为程序单元，表达处理逻辑的主要机制是顺序和转移。显然，这一抽象层次是比较低的，软件难以变动和演化。结构化编程语言使用变量、标识符等概念作为语言的基本构造，并使用 3 种基本控制结构来表达软件模型的计算逻辑，实现了模块化的数据抽象和过程抽象，并使开发的软件具有一定的构造性和演化性。面向对象程序设计语言从现实世界中客观存在的事物（即对象）出发来构造软件系统，提高了人们表达客观世界的抽象层次，使开发的软件具有更好的构造性和演化性。目前，人们更加关注软件复用和演化问题，构建比对象粒度更大、更易于复用和演化的基本单元——构件，并研究以构件为基础的软件构造方法，更好地凸现软件的演化特性。

软件体系结构近年来已经成为软件开发过程中的核心制品，起着主导作用。其研究和实践旨在将一个系统的体系结构显式化，以便在高抽象层次处理诸如全局组织和控制结构、功能到计算元素的分配、计算元素间的高层交互等设计问题。它从系统的总体结构入手，将系统分解为构件和构件之间的交互关系，可以在高层抽象上指导和验证构件组装过程，提供了一种自顶向下、基于构件的软件开发方法，从而为构件组装提供了有力的支持，并使软件的构造性和演化性进一步提高。

3.2　动 态 需 求

3.2.1　具有动态性的需求和需求的动态性

随着软件技术的不断发展和对目标软件系统的期望越来越高，有些与软件需求相关的概念也十分重要。例如，具有动态性要求的需求越来越引起人们的重视，另外，需求本身

也具有动态性的特点，这两个概念容易混淆，下面我们举例说明这些概念。

某公司要开发一套应用系统，用户要求：目标软件系统中与财务相关的子系统应能跟得上国家财务制度的改革，系统能很快按照财务制度改革后的新办法进行财务核算，报表也要符合与该核算方法相对应的格式，而目标软件系统不是每次都要随着国家财务制度的改革而升级。从用户的这个需求不难看出，其中“目标软件系统中与财务相关的子系统应能跟得上国家财务制度的改革”这一要求，表明用户需求是随时间而有所变化的，也就是需求的动态性；而“报表也要符合与该核算方法相对应的格式”这一具体要求，表明用户需求的变化，要用一种相对稳定的方式去跟得上这种变化，那么这种需求是要求目标软件系统具有一定的动态性，能适应可能变化的环境。

具有动态性要求的需求是指需求本身或潜在要求目标软件系统具有动态性，即目标软件系统具有自主适应运行环境和动态改变运行方式的能力。在理想的情况下，软件系统应能自适应需求的变化，能自适应运行环境的变化、运行方式的变化。例如，某应用系统在使用过程中，用户有了新的需求或运行环境变化了，那么该用户只要在其控制中进行简单设置或者智能添加新功能或者去掉废弃不用的子系统。

需求的动态性是指需求本身随时间发生变化，其一，用户不断提出新的要求，而否定过去需求中的一部分内容，难以确定用户的最终需求，其二，目标软件系统已经运行一段时间后，用户才提出新要求，并且发现正在运行的系统中有很多功能、性能、行为不符合其目前的真实要求。最极端的一个例子是，当软件开发人员把软件开发好给用户安装上，用户使用一段时间后，说这根本不是他想要的软件系统。

具有动态性要求的需求和需求的动态性是相关联的，如果用户提出具有动态性要求的需求，那么目标软件系统应具有一定的动态性、适应性，能适应将来需求的动态变化，而用户需求的动态性最好反映具有动态性要求的需求，让目标软件系统能较好地弥补用户需求的变化；但是用户需求的动态性，原因很多，多数是用户也不清楚需要什么，因此，也很难让软件开发人员去帮助用户设想，进而使目标软件系统能动态适应需求的变化。只有当软件开发者具有比较丰富的领域开发经验时，做到这点才有一定的可能性。另外，这种具有动态性要求的需求也有与需求的动态性无关的一层含义，那就是用户从功能要求出发，就应该在目标软件系统中有动态性、适应性的要求。

3.2.2 需求的动态变化性

前面提到的具有动态性要求的需求是指需求本身或潜在要求目标软件系统具有动态性，即目标软件系统具有自主适应运行环境、动态改变运行方式的能力。以设计多级药店管理系统为例，那么这类需求是什么？通过和投资商讨论，调研各类药店，以及分析现有各类系统后，发现目标系统投入运行后，应该能人工或智能地进行定制或重构系统，改变部分子系统的运行方式甚至体系结构。通过分析，根据投资商对业务要求以及用户使用系统时的环境和范围要求，确定以满足多级连锁型药店为目标，决定采用多级分布结构，即对于某一级药店，对上为上级店的分店，对下级店为总店，根据其自身店面规模可选择 B/S、C/S 体系结构，三层/多层体系结构或者是多种体系结构的混合体。由此派生出系统的硬件结构图和软件结构图。

图 3-1 为系统两级间的硬件结构图，图中假设上级店有多个下级店，而每个下级店根据情况又可有多个下级店，因此，用两级店之间的连接关系表示系统的硬件结构。对于某药店内部可采用局域网的形式连接，而连锁的上下级药店之间可选择广域网的连接方式，如用现有的公共 Internet 网。如果某上级店不接受远程客户时，可启用电话拨入服务，让下级店与其连接。上下级店都配备有数据库服务器，二者之间的信息交换也通过服务器进行。

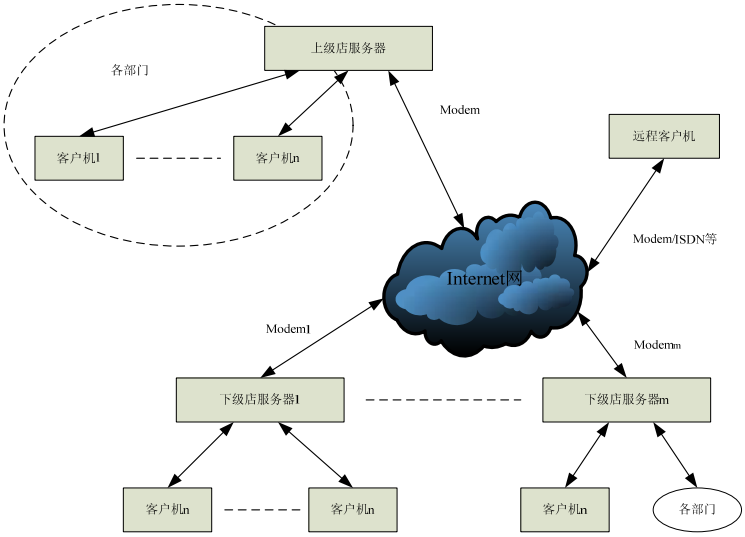


图 3-1　两级药店间硬件结构图

图 3-2 为系统的软件结构图，与硬件结构图紧密对照，总体上可分为控制台、客户端和信息中心三部分分别进行开发。

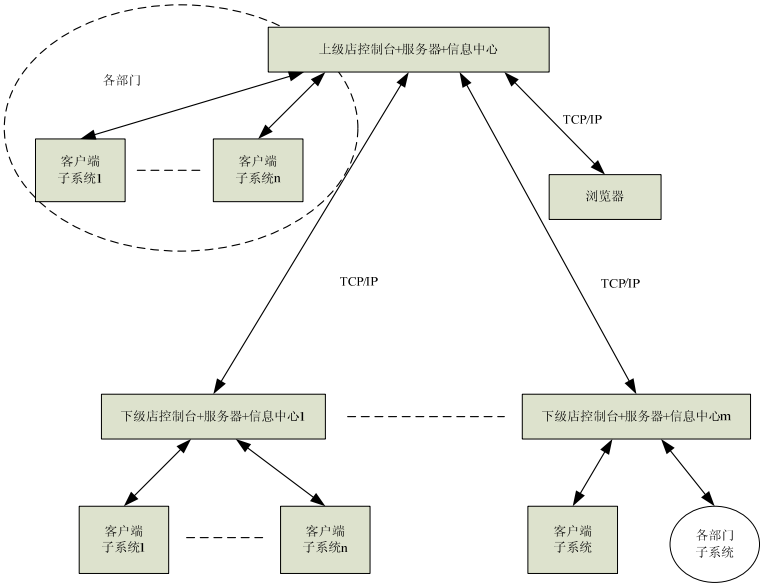


图 3-2　两级药店间软件结构图

在确定硬件和软件的结构方式后，我们可以看出，通过系统的组合以及裁减，基本上可以实现投资商的动态性要求的需求，这一点我们将在下一小节内再进一步澄清，接下来看，系统如何能迎合需求的动态变化。

3.2.3 具有动态性的需求

前面谈到的需求的动态性是指需求本身随时间发生变化，其一，用户不断提出新的要求，而否定过去需求中的一部分内容，很难确定用户的最终需求；其二，目标软件系统已经运行一段时间后，用户才提出新的要求，并且发现正在运行的系统中有很多功能、性能、行为不符合其目前的真实要求。在本例中，不可能分析每一家药店的实际情况，也不可能预测到将来政策、形势的发展变化给药店经营模式带来多大的影响。因此，只能使系统具有较强的开放性，在原有系统不能满足需求的变化时，可以方便快速地升级软件。为此，把软件模型进一步改进如图 3-3 所示。

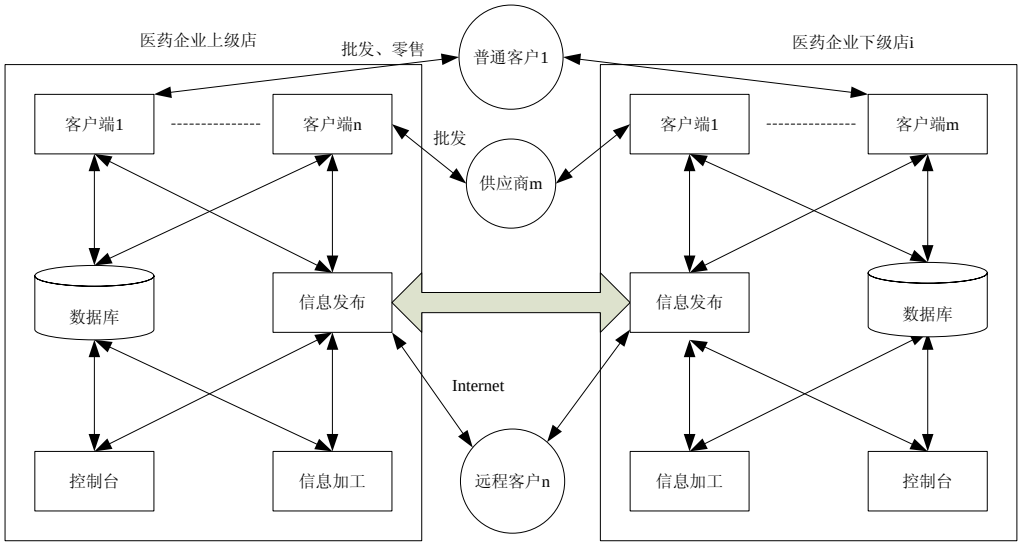


图 3-3 软件模型

图 3-3 所示的是多级连锁型药店上级店和下级店通过信息发布联系，一个上级店管理多个下级店，且各下级店不接受其他店的管理，上级店和下级店的软件模型是完全一致的，只是注册的基本信息、业务构件根据药店的级别不同而有所不同。对于某药店（假设可以完成所有业务类型），可以从供应商或上级药店购买药品，也可以通过上级药店调拨（调入和调出）或向下级药店配送，普通用户（包括来批发药品的客商和在零售柜台买药的患者等）在药店买药或批发，而远程用户通过 Internet（信息发布）来完成各类业务和了解药店、药品信息以及反馈问题等（进行业务时可采用较为安全连接方式，如 Https 等）。该模型采用了控制台－信息－客户端模式的结构，基本上算是一个以 C/S 为主，混合 B/S 以及三层/多层的软件体系结构。

如图 3-4 所示，该模式由三部分组成：控制台、信息中心和客户端。其中，控制台是由可静态配置的基础框架和负责不变业务需求的构件组成，完成如权限管理、数据库维护、备份等

功能；信息中心由信息发布应用程序和信息加工应用程序两部分组成，负责生成发布信息、业务逻辑定义、配置系统信息、构件注册管理等；客户端是由可动态配置的基础框架和负责可变业务需求的构件组成，即包括个性化用户接口框架、各类型业务构件、业务流程助手等。

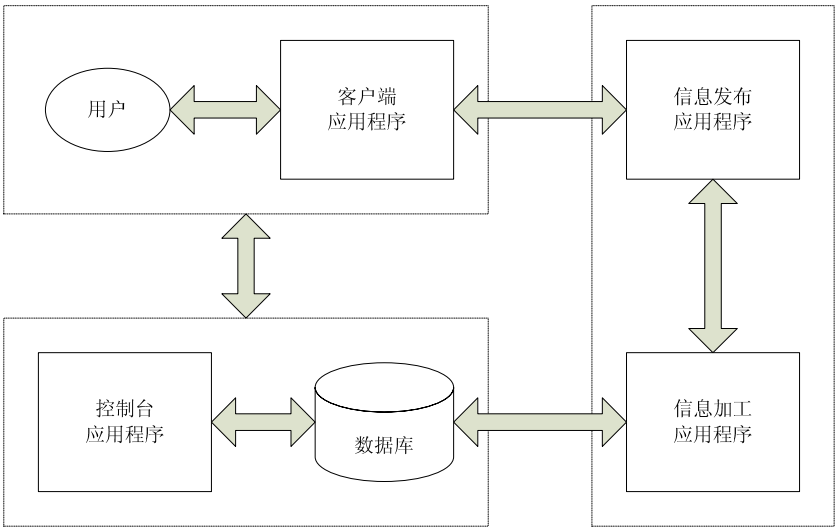


图 3-4　控制台-信息中心-客户端模式

体现能适应需求的动态变化性所在，主要是由信息中心的业务逻辑定义和软件自动升级去完成，而其基础是两个良好的框架，即控制台的可静态配置基础框架和客户端的可动态配置基础框架。

要说明的是，仿佛还没有进行很细致的需求分析就给出了软件的高层设计，这显然不符合需求分析的一般步骤。其实不然，模型的选用是系统分析人员对软件需求有了大致了解后，根据自己成功设计系统的经验选用的，一般来说，这是长期技术和经验积累的结果，而在选用合适的模型后，系统分析人员才进一步细化和分析具体的需求。

3.3　应用设计模式

随着软件工程的日益成熟以及一些大型复杂系统开发的需要，有效重用现有的程序模块和设计经验是当前软件开发人员日益关注的问题。从面向过程的程序设计到基于体系结构的软件开发可以看出，软件的复用粒度越来越大。同时从原本的编程技术逐步转移到更宏观的软件结构设计上；从单纯的代码复用转移到代码、知识、经验的复用上。除此之外，应用设计模式同时还可以让我们从更高层次上来考虑软件设计中的一些问题，提高了软件的抽象层次、便于用更形式化的语言进行描述。

如果说最初的子程序复用以及后来的类库复用是针对代码级的复用，那么构件库的复用则不仅在代码级，还可以在二进制码级和功能级复用。系统设计者复用构件时不需要了解构件的结构和编码，只需知道其功能和组装接口。设计模式库和应用框架库的复用则是针对软件软分析、设计和实现各个环节的成果复用，并使这些环节的工作之间能平滑过渡，使软件

开发过程中的所有活动都可以不同层次、不同粒度地复用软件半成品或以往的软件开发经验。

目前主要存在两种类型的可重用体系结构产品：产品系列以及体系结构模式。其中产品系列是针对特定领域的软件体系结构，定义了通用应用和问题域中的一系列应用，它们主要是按照功能来进行软件体系结构的分类。而体系结构模式是针对特定问题的解决方案，对于某一类问题设计经验的集合。

在很多书上对于模式和风格的概念存在许多争论，有很大一部分人认为设计模式就是风格，也有一部分人认为它们属于两种不同的概念。在本书中将分别对其进行讨论，本节主要介绍设计模式在可动态演化系统设计中的应用，利用已有的设计模式有利于提高软件的生产率、增强软件质量以及降低软件开发与维护成本。在 3.5 节将介绍如何应用风格来设计可动态演化的软件系统。

3.3.1 设计模式的概念和分类

1. 概念

设计模式可以理解为已经被人们证明的在同类问题中有效的解决方案。在这里我们引用当前被广泛认同的一个等式来表示：

Patterns == problem/solution pairs in a context

即：模式是特定背景下的问题与解决方案。模式主要便于重用现有成功的软件体系结构和设计经验。使用模式的好处主要表现在以下三方面。

（1）权衡已经被证实的解决方案。模式是实践的总结，它提供的解决方案是经过在不同时间、不同项目中反复解决了相似的问题后才最终确定的，因此，模式提供了强大的可重用机制，避免开发者和设计者的重复投资；

（2）为交流提供一个共同的词汇表。模式不仅为软件设计者提供一个共同交流的词汇，同时，提供了一种与开发者进行交流的方式。软件设计者和开发者不但可以通过学习和理解模式来创造自己的词汇，而且随着新词汇的记载而增加设计词汇。

（3）约束解决方案的范围。使用模式可以将解决方案约束或限定在一定的范围内，这个范围正好是设计和实现可以实现的范围。模式要求开发者确定实现的边界，超出这个边界就破坏了模式与设计的一致性，就可能导致错误。

设计模式使人们可以更加简单方便地复用成功的设计和体系架构，将已证实是有效的技术实现方式表述为设计模式会使新系统开发者更加容易理解其设计思路，帮助系统设计工程师做出有利于系统复用的选择，避免设计损害了系统复用，进而更好地完成系统设计。

一般来讲，一个模式要有四个基本要素。

（1）模式名称（pattern name）它用一两个词来描述模式的问题、解决方案和效果。命名一个新模式增加了我们的设计词汇。基于一个模式词汇表，软件工程师及其同事之间就可以讨论模式并在文档编写时利用它们。模式名称可以帮助设计者思考、设计者之间交流设计思想及设计结果。

（2）模式问题（pattern problem）描述应该在何时使用模式。它解释了设计问题和问题存在的前因后果，可能描述特定的设计问题，也可能描述了导致不灵活设计的类或对象结构。

（3）解决方案（solution） 描述设计的组成成分，它们之间的相互关系及各自的职责和协作方式。它并不描述一个特定而具体的设计或实现，而是提供抽象问题的描述和怎样用一个具有一般意义的元素（类或对象）组合来解决这个问题。

（4）效果（consequences） 描述了模式应用的效果及使用模式应权衡的问题。模式效果包括它对系统的灵活性、扩充性、可执行性的影响，显式地列出这些效果对理解和评价模式很有帮助。

2. 分类

模式提供了一种有效的软件复用方式。正如不同的设计工程师在设计时复用软件的层次不一样，人们出发点不同会产生对什么是模式和什么不是模式的理解不同。一个人的设计模式对另一个人而言可能只是基本构造部件，同时，模式代表着对特定环境中出现的问题的专业解决方案。因此，设计模式在粒度和抽象层次上各不相同，可以在多个抽象的层次和多个领域内定义模式。常见的软件模式分类有：设计型模式、构架型模式、分析型模式、创建型模式、结构型模式和行为型模式。Erich Gamma 等人提出了两条分类准则。

（1）目的准则，模式是用来完成什么工作的；

（2）范围准则，模式主要用于类还是用于对象。

模式依据其目的可分为：创建型（creational）、结构型（structural）和行为型（behavioral）三种。创建型模式与对象的创建有关，结构型模式处理类或对象的组合，行为型模式对类或对象怎样交互和怎样分配职责进行描述。

模式依据其范围可分为：类模式和对象模式。类模式处理类和子类之间的关系，这些关系通过继承建立，是静态的，在编译时刻就确定下来了；对象模式处理对象间的关系，这些关系在运行时刻是可变的，具有动态性。Erich Gamma 等人即 Gof 在《design pattern》中提出了 23 个设计模式，并对其依据两条分类准则进行分类，如表 3-1 所示。

Gof 定义的模式比较系统、理论地抽象了软件设计及其实现过程中的共性，它成为软件设计的有效手段，同时也是设计、组合其他模式的基础，下面分别作以介绍。

表 3-1　设计模式分类

		目　　的		
		创 建 型	结 构 型	行 为 型
范围	类	工厂方法（Simple Factory）	适配器（类，Adapter）	解释器（Interpreter） 模板方法（Template Method）
	对象	抽象工厂（Abstract Factory 生成器（Builder） 原型（Prototype） 单件（Singleton）	适配器（对象，Adapter） 桥接（Bridge） 组成（Composite） 装饰（Decorator） 外观（façade） 享元（Flyweight） 代理（Proxy）	职责链（Chain of Responsibility） 命　令（Command） 迭代器（Iterator） 中介者（Mediator） 备忘录（Memento） 观察者（Observer） 状　态（State） 策　略（Strategy） 访问者（Visitor）

3. 创建型模式

创建型模式抽象了实例化过程，它帮助一个系统独立于创建、组合和表示它的那些对象，一个类创建型模式使用继承，改变了被实例化的类，而一个对象创建型模式将实例化委托给另外一个对象。

随着软件开发手段不断发展，系统越来越依赖于对象复合而不是继承，创建型模式变得更为重要。此时，软件开发的重心从对一组固定行为的硬编码（hard-coding）转移为定义一个较小的基本行为（构件，可以是类、对象或它们的更高级别的封装）集，这些行为可以被组装成任意数目的更复杂的行为。在创建型模式中首先要考虑两方面问题。

（1）它们都将有关系统使用哪些具体的类的信息封装起来；

（2）它们隐藏了这些类的实例是如何被创建和组装在一起的。

系统对于构成它的对象的了解是通过抽象类定义的接口来实现的。因此，创建型模式在什么场合被创建、谁在创建它、它是怎样创建的以及何时创建等方面给系统设计者提供了很大的灵活性，它允许设计者用结构和功能差别很大的构件配置一个系统，这种配置可以在运行时动态进行，也可以在编译时静态指定。

用一个系统创建的那些对象的类对系统进行参数化有两种常用方法。一种是生成对象的子类，此时使用 Factory Method 模式，其缺点是仅为了改变产品类，就可能需要创建一个新的子类。另一种方法是采用对象复合：定义一个对象负责明确产品对象的类，并把它作为该系统的参数。这就是 Abstract Factory、Builder 和 Prototype 模式的特征，它们都要创建一个新的负责创建产品对象的“工厂对象”，Abstract Factory 由这个工厂对象产生更多类的对象，Builder 由这个工厂对象使用相对复杂的协议，逐步创建一个复杂的产品，Prototype 由这个工厂对象通过拷贝原型对象来创建产品对象。

4. 结构型模式

结构型模式关注如何组合类或对象以获得更大的结构。结构型模式采用继承机制来组合接口或实现，比如采用多重继承方法将两个以上的类组合成一个类，这个类包含了所有父类的性质；结构型对象不是对接口或实现进行组合，而是描述如何对一些对象进行组合，从而实现新功能的一种方法。由于可以在运行时刻改变对象组合关系，所以对象组合关系具有更大的灵活性，而这种机制是用静态类组合不可能实现的。

在结构型模式中，适配器模式和桥接模式具有一些共同的特征，它们都给另一些对象提供一定程度的间接性，都要从自身以外的一个接口向这个对象转发请求。适配器模式主要是解决两个已有接口之间的匹配问题，它不考虑这些接口的实现以及如何演化，它只需要对两个独立设计的类进行重新设计，就能使它们协同工作。桥接模式则对抽象接口与它的实现部分进行桥接。它们被用在软件生命周期的不同部分，适配器模式在类设计好后实施，而桥接模式则在类设计好前实施。外观模式定义一个新接口，适配器是复用一个原有的接口，并且是使原有的两个接口协同工作。

组合模式与装饰模式都是基于递归来组织可变数目的对象，装饰模式是一个退化的组合模式。装饰模式的主要目的是不需要生成子类及可以给对象增加职责，而组合模式则在于构造类，使多个相关的对象能够以统一方式处理，它的重点不在于修饰，而在于表示。

装饰模式与代理模式都描述了怎样为对象提供一定程度的间接应用，都为对象的实现部分保留了一个对象的指针，它们都向这个对象发送请求。但是代理模式不能动态地添加或分离性质，也不是为递归组合而设计的，其目的是当直接访问一个实体不方便或不符合需要时，可以为这个实体提供一个替代者。

5. 行为型模式

行为型模式涉及算法和对象间职责的分配，不仅描述对象或类的模式，还描述了它们之间的通信。这些模式刻画了运行时刻难以跟踪的复杂的控制流，它帮助软件设计者在设计软件时把注意力从控制流转移到对象间的联系上来。

行为型模式使用继承机制在类间分配行为。Erich Gamma 等人在《design pattern》中总结了两种行为型模式：模板方法（Template Method）和解释器（Interpreter），前者是一个算法的抽象定义，它逐步地定义该算法，每一步调用一个抽象操作或原语操作，子类定义抽象操作以具体实现该算法。后者将一个文法表示为一个类层次，并实现一个解释器作为这些实例上的一个操作。

行为型对象模式使用对象复合而不是继承，其中一些模式描述了一组对等的对象怎样互相协作以完成其中一个对象无法完成的任务，另外一些模式常将行为封装在一个对象中并请求指派给它。

各行为型模式之间是相互补充的，一个职责链中的类至少包含一个模板方法的应用，解释器可以使用状态模式定义语义分析上下文，迭代器可以遍历一个聚合，而访问者可以对它的每一个元素进行一个操作。行为型模式的主要特点表现为：

（1）封装的变化。当一个程序某些方面的特征经常发生变化时，行为型模式就定义一个封装这些特征的对象，例如一个 Strategy 对象封装一个算法、一个 State 对象封装与一个状态相关的行为、一个 Interpreter 对象封装对象间的协议、一个 Iterator 对象封装访问和遍历一个聚合对象中各个构件的方法。

（2）对象作为参数。一些模式引用总是被用作参数的对象，例如，一个 visitor 对象是一个多态的 accept 操作的参数，该操作作用于 visitor 对象访问的对象。其他一些模式定义一些作为令牌到处传递的对象，例如，Memento 中令牌代表在一个对象在某时刻的内部状态，Command 中令牌代表一个请求。

（3）通信应该被封装还是被分布。Mediator 和 Observer 使用竞争模式，Observer 通过引入 Observer 和 Subject 对象来分布通信，而 Mediator 对象则封装了其他对象的通信。

（4）对发送者和接受者解耦。当合作的对象直接相互引用时，它们变得互相依赖，这影响了系统在层次和重用性方面的性能。行为型模式通过对接收者和发送者解耦的方式来解决系统中对象的依赖性问题。

6. 小结

由于本节重点介绍支持软件动态演化的几种设计模式，因此只对这三大类设计模式进行简单介绍，便于读者对常见的设计模式有一些简单的了解。设计模式主要应用在软件设计阶段，它不仅可以解决许多设计中的难题（如采用享元设计模式可以减少大量的存储开销，它对每个使用场景都作为一个独立的对象进行存储），同时还便于软件开发人员把握整

个软件体系结构，对这些设计模式有兴趣的开发人员可以参考书目《Design Patterns：Element of Reusable Object-Oriented Software》。

3.3.2 支持动态演化的设计模式

软件的动态演化主要包括两种：预设的动态演化以及非预设的动态演化，其中在设计体系结构时考虑到的演化行为称为预设的演化。比如，在一些安全性要求较高的银行系统需要有两台服务器，其中一台为主服务器，而另一台为预备服务器，当主服务器发生故障或死机后，所有客户端与服务器的交互都要从主服务器上导向从服务器上，而不能出现服务停止故障。这种类型的演化称为预设的动态演化。而另一种是在软件投入使用之后所需要进行的软件动态演化，通常由于系统需求的改变以及系统功能的升级，需要系统在不停机的情况下进行系统更新，这种称为非预设的动态演化。

在软件设计阶段使用部分设计模式有利于软件的动态演化。其中有些设计模式支持预设的动态演化，也有些支持非预设的动态演化，接下来将分别介绍几种典型的支持软件动态演化的设计模式。

1. 简单工厂（Simple Factory）

Simple Factory 模式（又称 Static Factory 模式），一个 Simple Factory 生产成品，而对客户端隐藏产品生产的细节。实作时定义一个产品接口（interface），并通过特定静态方法来建立成品。由于在工厂模式中，客户端只关心其接口，而不关心实现细节，因此，当我们需要在不改变原有功能需求的情况下进行系统功能的动态升级以及修复一些 BUG 时，可以在不改变客户端的情况下只修改具体功能的实现类或构件即可，因此也方便软件的动态演化。在进行构件的替换时，我们需要接口（该接口与面向对象的接口有稍许差别，它主要是指构件对外所提供的功能，同时还定义了构件需要外界所提供的功能）兼容。因此工厂模式主要适用于接口不变的情况下对构件进行功能完善性的动态演化。以下主要以类层次介绍该模式的具体应用。

假设有一个音乐盒工厂，购买音乐盒的客人不用知道音乐盒是如何制作的，他们只要知道如何操作音乐盒就可以了，以 UML 类图来表示以上的概念。

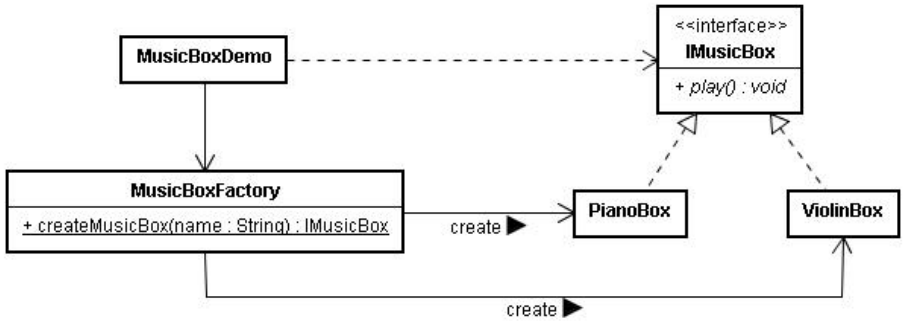


图 3-5 音乐盒 UML 类图

如图 3-5 所示，MusicBoxDemo 代表了客户的角色，它只依赖于 IMusicBox 接口，而

不关心特定的实作，实际如何产生 IMusicBox 的实例由 MusicBoxFactory 完成，以一个简单的程序来实现上面这个 UML 类图：

IMusicBox.java

```
public interface IMusicBox {
    public void play();
}
```

PianoBox.java

```
public class PianoBox implements IMusicBox {
    public void play() {
        System.out.println("播放钢琴音乐:)");
    }
}
```

ViolinBox.java

```
public class ViolinBox implements IMusicBox {
    public void play() {
        System.out.println("播放小提琴音乐^_^");
    }
}
```

MusicBoxFactory.java

```
public class MusicBoxFactory {
    public static IMusicBox createMusicBox(String name)
                            throws InstantiationException,
                                   IllegalAccessException,
                                   ClassNotFoundException {
        // 这边使用的是 Java 的 Reflection 机制来产生实例
        // 不过客户端不用管
        // 以后就算改变了这边的程序，客户端程序是不用更改的
        return (IMusicBox) Class.forName(name).newInstance();
    }
}
```

MusicBoxDemo.java

```
public class MusicBoxDemo {
    public static void main(String[] args) throws Exception {
        playMusicBox(MusicBoxFactory.createMusicBox("PianoBox"));
        playMusicBox(MusicBoxFactory.createMusicBox("ViolinBox"));
    }

    public static void playMusicBox(IMusicBox musicBox) {
```

```
            musicBox.play();
        }
    }
```

由于客户端只依赖于 IMusicBox 接口，所以即使日后改变了 createMusicBox()中的实作方式，对客户端也不会造成影响。

来看看 Simple Factory 的类结构：

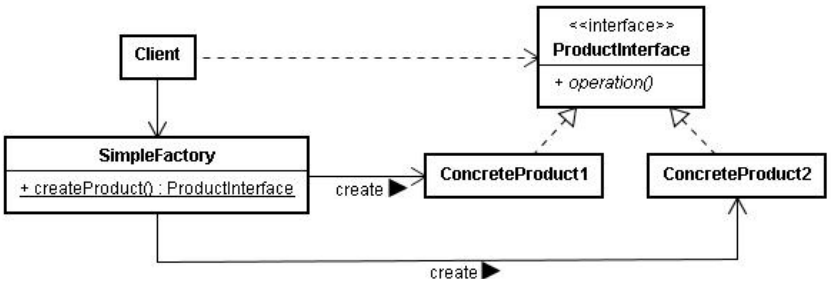


图 3-6 Simple Factory 的类结构

客户只要面对 Factory，客户依赖于产品接口，产品的具体实作是可以与客户隔开的，它们也是可以更换的。

2．适配器（Adapter）

适配器又称包装器（Wrapper），它将一个接口转换成用户希望的另外一个接口，使得原来由于接口不兼容而不能一起工作的那些类可以一起工作。在前面已经介绍过，适配器模式和桥接模式都是通过接口来与其他类进行间接通信，因此它降低了对象与对象之间的耦合。它是用于下列情况：

（1）想使用一个已有的类，但它的接口不符合你的需求；

（2）想创建一个可复用的类，使它与其他不相关的类或不可预见的类（即接口可能不兼容的类）协同工作；

（3）想使用一些已存在的子类，但是不可能对每一个进行子类化以匹配它们的接口（仅适用于对象 Adapter）。

（4）类适配器适用于多重继承对一个接口与另一个接口进行匹配，对象适配器可以适配它的父类接口，它依赖于对象组合。

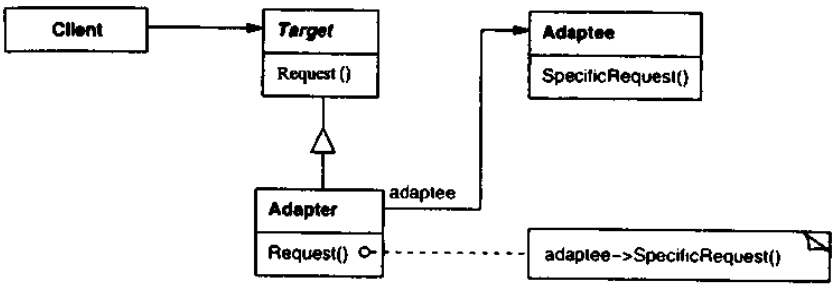


图 3-7 适配器模式图

3. 桥接（Bridge）

桥接又称句柄（handle/body），将抽象部分（即接口）与实现部分分离，使它们可以独立地变化。它适用于以下情况：

（1）不希望抽象部分与其实现部分有一个固定的绑定，这样在程序运行时刻实现部分可以被选择或切换；

（2）通过生成子类的方法进行类的抽象及其实现，对抽象部分和实现部分进行组合，并分别加以扩充；

（3）对一个抽象的实现部分的修改，不会对客户产生影响，客户的代码不必重新编译；

（4）在 C++编程中，想对客户完全隐藏抽象的实现部分；

（5）想在多个对象间共享实现，但又不要求客户知道。

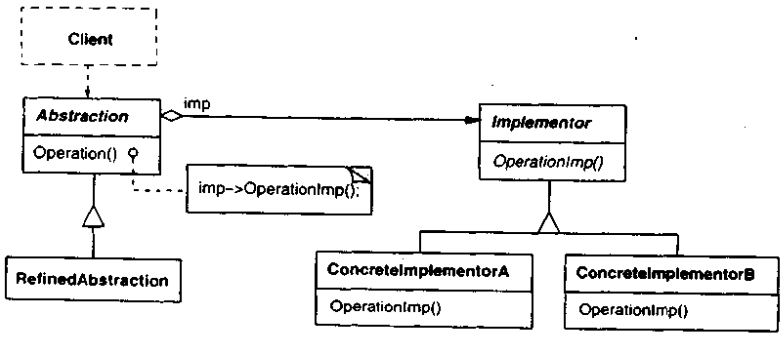


图 3-8　桥接模式图

4. 代理（Proxy）

代理（Proxy）也称为 surrogate，它为其他对象提供一种代理以控制这个对象的访问。常用的代理有：远程代理、虚代理、保护代理和智能指引。

Proxy 保存一个引用使得代理可以访问实体，它提供一个与 Subject 的接口相同的接口，控制对实体的存取，并可能负责创建或删除它。在进行类层次的软件动态演化中可以使用该设计模式，由于两个直接引用的对象之间插入了一个代理，因此可以方便地实现对象在运行期间进行替换。我们也可以将代理理解为一个对象，当我们使用新的对象来替换现有的对象时，就可以直接通过代理，来完成对新对象的调用，这样，所有对原有对象的调用就可以方便地通过代理来将调用导向新的对象中去。

根据不同的代理目的，而有不同的代理情况，Gof 所举的一个例子是 Virtual Proxy，以文件中内嵌图片为例，假设图片是在文件分页的后面，一开始并不用直接加载图片，而使用一个虚代理对象，代替图片被加载，以求开启一个文件的时候，速度能够加快。当卷动文件至该显示图片的页数时，这时再加载图片。

如上图所示，当文档被开启时，ImageProxy 对象代理 Image 对象被加载，在还没卷动至图片显示处时，也就是还没有调用 ImageProxy 的 draw()时，图片并不会被加载，因而可以加速文件的开启与节省内存的使用；如果需要显示图片了，ImageProxy 的 draw()会被

调用，而这时才真正创建 Image 对象，以从硬盘中加载图片。

Proxy 模式的 UML 结构图如下所示：

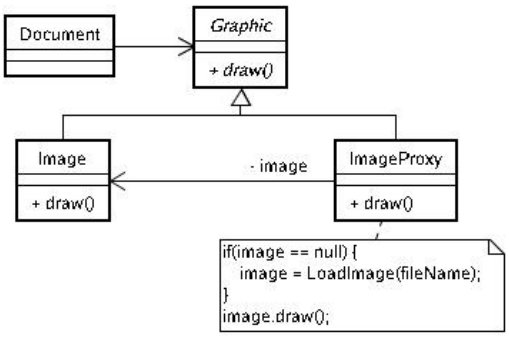


图 3-9 代理模式图

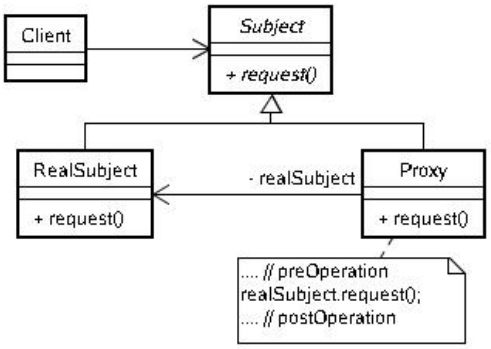


图 3-10 代理模式 UML 结构图

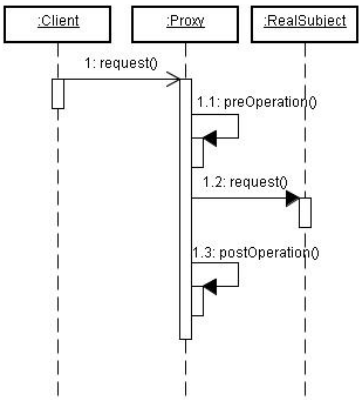


图 3-11 代理模式时序图

在调用 RealSubject 的 request()之前，Proxy 对象也许会有一些预先处理的操作，假设为 preOperation()与 postOperation()，当客户对 Proxy 发出 request()请求后，一个可能的时序图如下图所示。

preOperation()与 postOperation()决定了 Proxy 模式适用于何种情况，例如一个 Remote Proxy 的情况，可以为一个远程真实对象提供一个局部代表；Protection Proxy 控制对象的访问，可以使用它来作不同级别、权限的存取控制；Cache Proxy 为一个对象提供临时的储存，使得许多客户端都能直接存取它，而不用对真实对象直接要求，只有在必要的时候更新这个临时对象，或是让客户直接存取真实对象。

5. 职责链（Chain of Responsibility）

职责链模式就是把涉及的对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止，它使得这条链上的每一个对象都有机会处理请求，从而避免了请求发送者和接受者之间的耦合关系。

在基于体系结构的动态演化中，为了方便构件的添加、删除与替换，提出了多种软件体系结构风格，其中就有消息总线风格，而职责链模式则如同总线风格的体系结构一样，我们把所有的构件都连接在一条总线上进行通讯，构件和构件之间的耦合全部解除。所有构件只要符合总线风格的约束即可添加到整个系统中来，因此，它可以很方便地实现构件的添加、删除和替换。所有构件之间的信息传递都通过总线来完成。

在以下条件下使用职责链。

（1）有许多个对象可以处理一个请求，哪个对象处理该请求在运行时可自行确定；

（2）想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求；

（3）可以处理一个请求的对象集合应被动态指定。

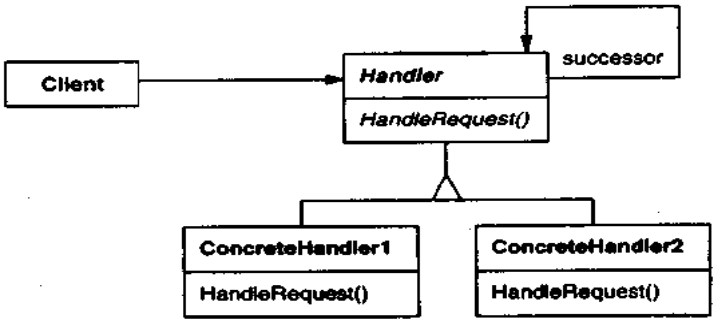


图 3-12　代理模式类图

6. 中介者（Mediator）

面向对象的设计鼓励将行为分布到各个对象中，这种对象间的相互连接会导致其复用性降低，使得对系统的任何较大改变都变得十分困难，中介者模式正是针对这个问题提出来的。中介者是用来封装一系列对象之间的交互，使各对象不需要显式地互相引用，从而使其耦合松散，并且可以独立地改变他们之间的交互。

Mediator 模式用一个中介的对象来封装对象彼此之间的交互，对象之间并不用互相知道另一方，这可以降低对象之间的耦合性，如果要改变对象之间的交互行为，也只需要对 Mediator 加以修改即可。

在 Gof 的书中所举的例子为对话框组件。例如，当一个特定的输入栏为空时，该栏对应的按钮不能使用；在 ListBox 的选项中选择一个项目，将会改变另一个字段的内容；反之，输入字段的内容也会影响 ListBox 的选择等。

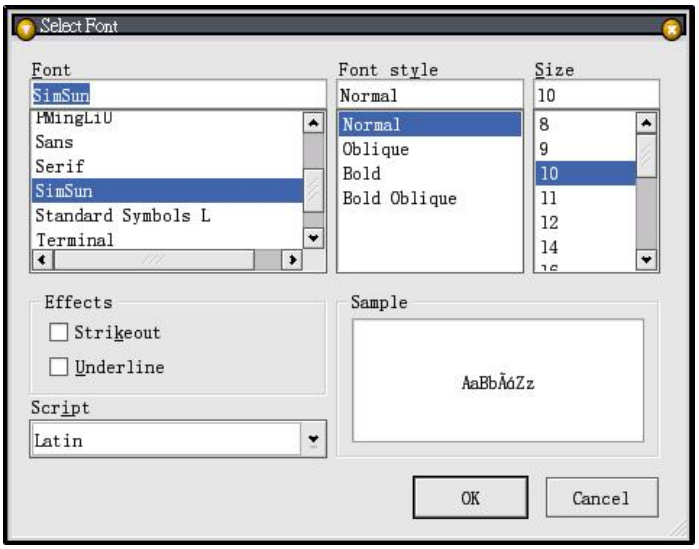


图 3-13　对话框组件图

在这个例子中，可以设计对话框中的组件知道彼此的存在，由一个直接影响另一个（或多个）组件，但最好的方法，还是设计一个 Mediator，由它来协调组件之间的交互，例如

设计一个 FontDialogDirector 类别来作为中介者。

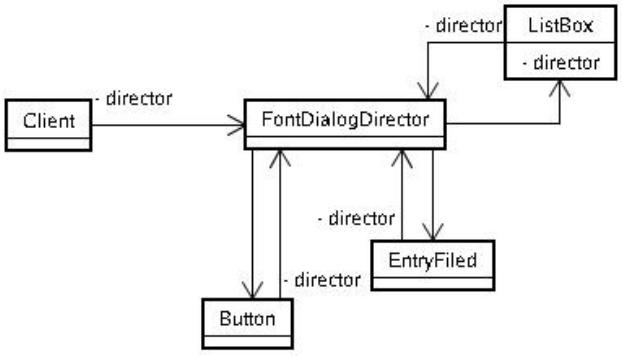


图 3-14 对话框协作图

可以从 Sequence Diagram 来了解 Mediator 的运作：

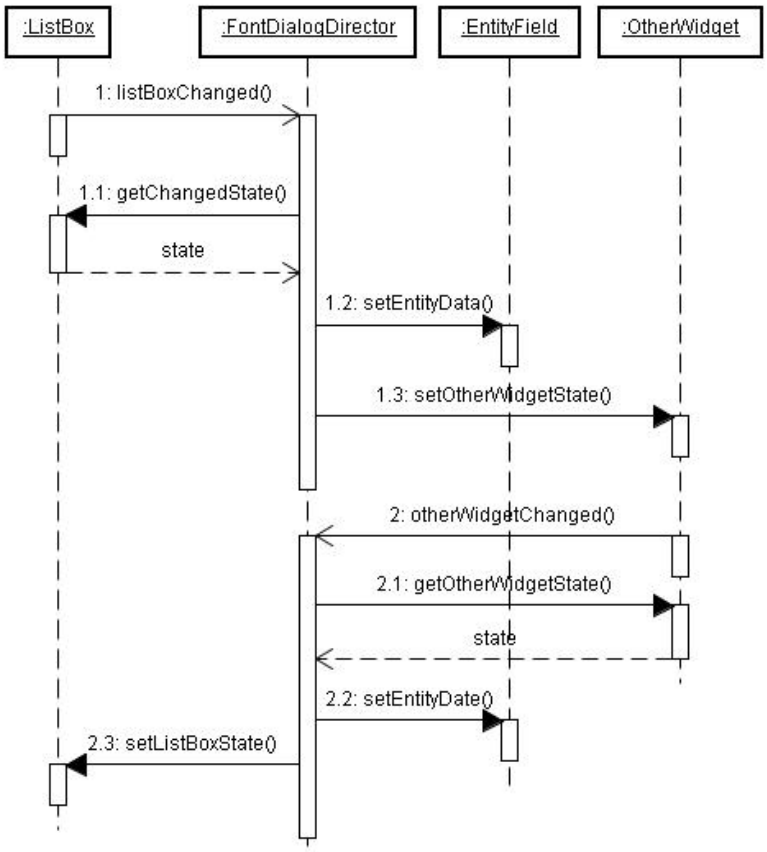


图 3-15 对话框时序图

当 ListBox 发生变化时，它会呼叫 Mediator 的 listBoxChanged()方法，Mediator 取得变化的组件之状态，并重新设定所有与它有交互的组件，同样的，其他的组件发生变化时，也呼叫 Mediator 上对应的方法，由 Mediator 来取得组件变化，并设定其他互动的组件。

简单地说，Mediator 设计有与组件沟通的接口，接口中封装了与其他组件互动细节，组件与组件之间不用知道彼此的存在，它们只要与 Mediator 沟通就好了，利用这种方式，可以切开组件与组件之间的耦合。Mediator 模式的 UML 结构图如图 3-16 所示。

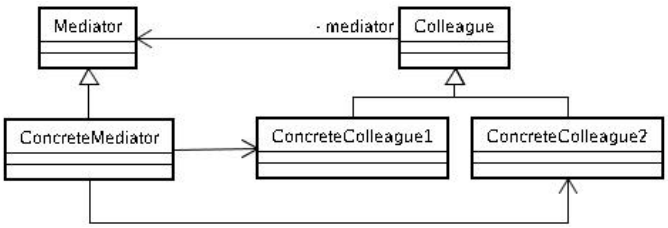


图 3-16　对话框类图

基本上 Mediator 模式使用的弹性很大，由 Sequence Diagram 理解概念，会比从 Class Diagram 了解结构来得重要，不过在 Class Diagram 中可以注意的是类别的名称，Colleague 是同事的意思，将一群共事的组件视为一群共同合作的同事，为了使同事之间的活动独立，并使得团队合作的交互更具弹性，需要一个 Mediator 来协调同事之间的业务行为。

7. 观察者（Observer）

观察者又称依赖（Dependents）或发布—订阅（Publish—Subscribe），它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。观察者的目的是维护对象间交互的一致性而又不使各类紧密耦合。其代表例子就是：分别用图表、柱状图和饼图来表示一组数据，这组数据叫做目标对象，图表、柱状图和饼图就是依赖于目标对象——数据的观察者。一旦数据改变，依赖于数据的图表、柱状图和饼图都得到通知并做出相应的改变。

观察者将数据提供者和它的观察者分离开来，从而更方便在不影响系统其他部分的情况下添加新的观察者。

假设设计一个电子表格程序，当中有一个数据对象，可以用表格图形对象、柱状图形对象、圆饼图形对象等方式来呈现对象，无论用哪种图形对象，重点是若对数据对象的内容作了更改，则图形对象的内容也必须跟着修改，或者程序中有两个以上的图形对象来呈现数据，那么在图形对象上变动数据，则另一个图形对象也必须做出相对应的变化。

主题	数据对象		
观察者	柱状图形	表格图形	圆饼图形

又假设设计一个网络游戏，在服务器上维护一个联机客户端共享的数据对象，当其中一个客户端作了操作，将对此数据对象作修改，则服务器必须通知其他客户端作相对应的变化（例如人物位置走动、建了一个城堡等）。

主题	数据对象		
观察者	客户端一	客户端二	客户端三

在 Observer 模式中的主角为主题（subject）与观察者（observer），观察者订阅它感兴趣的主题，一个主题可以被多个观察者订阅，当主题的状态发生变化时，它必须通知（notify）所有订阅它的观察者，观察者检视主题的状态变化，并做出对应的动作，所以 Observer 模式也称之为 Publish-Subscribe 模式。 Observer 模式的 UML 图如下所示：

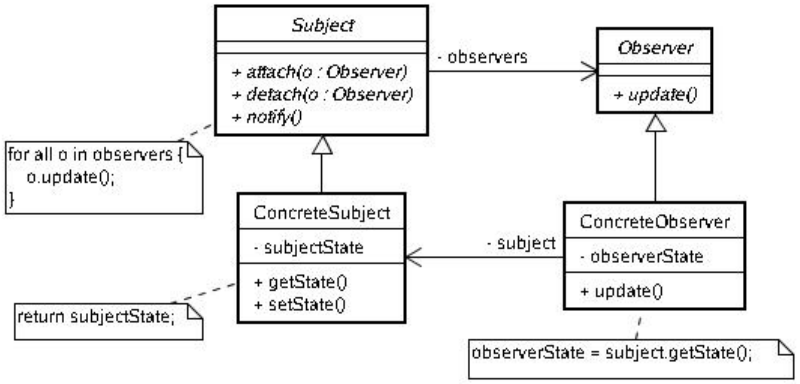


图 3-17 Observer 模式的 UML 图

Subject 类中有一个 notify()方法，通常是在 Subject 的状态发生改变时呼叫它，notify()中会呼叫 Observer 的 update()方法，通常会先取得 Subject 的新状态，然后更新 Observer 的显示或行为，这个过程我们可以透过 Sequence Diagram 来表达：

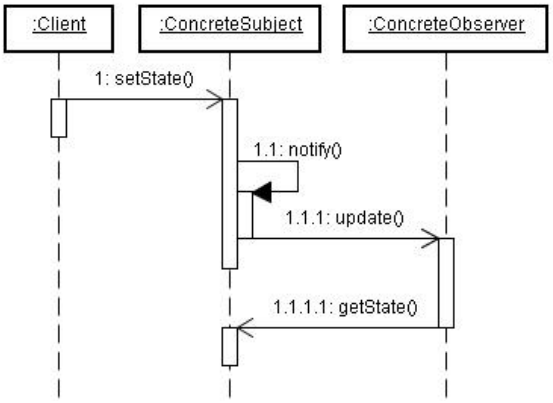


图 3-18 Observer 模式的时序图

在 Java 中支持观察者模式，要成为观察者的类必须实作 Observer 接口，这个接口中定义了一个 update()方法，这个方法会被主题对象在通知状态变化时呼叫，必须在这个方法中实作所想要的对应行为。

主题对象会是 Observable 的子类，在这边注意两个重要的方法：setChanged()与 notifyObserver()。 setChanged()是用来设定主题对象的状态已经被改变，而 notifyObserver()方法会通知所要订阅主题对象的观察者，调用其 update()方法。

有兴趣的话，建议看一下 Java 的 Observable。Java 中是如何实作的，这有助于了解 Observer 模式的运作方式。

8. 状态（State）

状态模式用于描述对象如何在每一种状态下表现出不同的行为，它允许一个对象在内部状态改变时改变它的行为。状态模式又名状态对象（Objects for States）。

状态模式主要可用于设计可预设的软件动态演化中，可以根据不同的对象状态来决定其行为，从而设计出自适应性较强的软件系统。它适用于下面两种状况：

（1）一个对象的行为取决于它的状态，并且它必须在运行时根据状态改变它的行为；

（2）一个操作中含有庞大的多分支条件语句，并且这些分支依赖于该对象的状态，State 将每一个条件分支放入一个独立的类中，根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

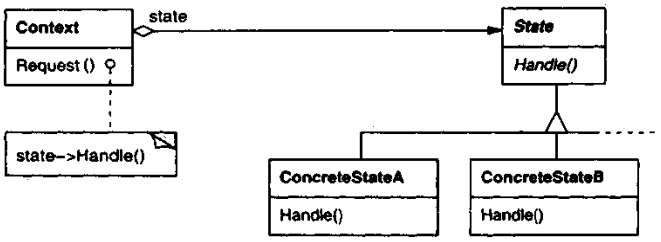


图 3-19　状态模式图

9. 策略（Strategy）

策略模式又称政策（Policy），它定义一系列的算法，并把它们一个个封装起来，使它们可以互相替换。它使得算法可独立于使用它的客户而变化。策略模式和状态模式有许多相似之处，它们都可用在可预设的动态演化中，用户可以定义演化策略然后根据具体情况来选择相应的演化策略，对软件实施动态变更。在下列情况时使用策略模式：

（1）许多相关的类仅仅是行为有异，策略模式提供了一种用多个行为中的一个行为来配置一个类的方法；

（2）需要使用一个算法的变体；

（3）算法使用客户不应该知道的数据，此时可以使用策略模式以避免暴露复杂的、与算法相关的数据结构；

（4）一个类定义了多种行为，并且这些行为在这个类的操作中以多个条件语句的形式出现，将相关的条件分支移入它们各自的策略类中以代替这些条件语句。

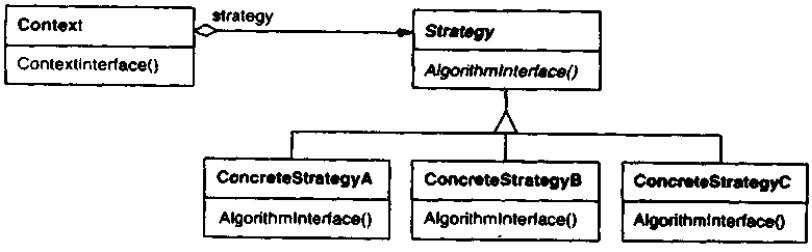


图 3-20　策略模式图

3.3.3 设计模式的应用

在软件设计阶段使用设计模式有助于提高软件的灵活性以及可扩展性。软件需求的多变性导致软件日后演化的需要，以前的软件升级都是经过软件系统的重新编译-停机-替换-重启的过程来完成对现有软件的升级，然而许多安全性较高或执行关键任务的软件系统在一旦停机就会带来无法估量的损失，因此，在软件的设计阶段就应该尽可能地考虑支持动态演化，从而为软件维护阶段提供动态演化基础。结构良好的软件通常更易于软件后期的动态演化。

3.4 应 用 框 架

在上一节主要介绍了设计模式在支持动态演化的软件系统设计中的应用。设计模式的复用主要是从设计经验的复用，它并不涉及到具体代码的应用。设计模式技术的发展和应用，软件开发先行者的经验得以方便复用，使软件开发过程变为“工艺化流程”，出现了专门从事离岸开发（offshore developing）的“软件工厂/车间”。软件框架技术的出现和应用使专门从事提供软件应用框架或基础结构成为可能，它们为软件开发商或系统集成商转向领域应用提供半成品软件，在此基础上针对最终用户的具体需求进行客户化再开发。本节首先简单介绍框架的概念，然后总结现有的一些框架，并对其进行分类，接着介绍一些支持动态演化的框架，最后举例说明框架在设计支持动态演化软件的应用。

3.4.1 框架的概念和分类

1. 概念

工业化的软件复用已经从通用类库进化到了面向领域的应用框架。Gartner Group 认为：“到 2003 年，至少 70%的新应用将主要建立在如软件构件和应用框架这类‘构造块’之上；应用开发的未来就在于提供一开放体系结构，以方便构件的选择、组装和集成”。框架的重用已成为软件生产中最有效的重用方式之一。

框架是构成一类专业领域可复用设计的一组相互协作的类，它规定了应用的体系结构、系统中的类和对象的分割及协作、各部分的主要责任、控制流程。框架预定义了设计参数，以便应用设计者或实现者能集中精力于应用本身的特定细节，框架记录了其应用领域的共同的设计决策，因而更强调设计复用。一个框架就是一个用于构建针对专门客户应用的可重用的“半成品”应用。与其他早期基于类的可重用的面向对象技术相比，框架更专注于具体的业务单元（如数据处理、通信功能单元）和专业领域（如用户界面、实时电子应用）。目前有许多框架系统，例如用户界面方面的框架有 MVC、ET++等，其中 ET++采用 C++ 语言实现，运行于 Unix 等系统中；针对其他领域的则有 FOIBLE、MacApp、FACE (Framework Adaptive Composition Environment)等。

许多采用 J2EE 编程环境的应用框架，如 Jcorporate 公司开发的 Expresso Framework 是

一使用 Java 来建造分布式、重用、基于构件的安全的 Web 应用程序的应用框架。基于框架建立的构件系统具有如下性能：

（1）实现面向产品化、实用性的构件库系统，并具开放性、可扩展性；

（2）支持异构环境中的框架、构件的互联和通信；

（3）实现新旧系统的兼容性；

（4）提供一致的接口分配；

（5）遵循重要构件标准（如 CORBA、J2EE、.NET 等）；

（6）构件具有透明本地化、平台无关性特点；

（7）系统的配置、数据交换基于 XML 和 Java 的标准化格式；

（8）支持个性化信息服务定制和可重构。

基于框架的软件的设计开发方法在很大程度上借鉴了硬件技术发展的思想，通俗地说“它是用硬件生产的方法设计和开发软件”，是构件技术、软件体系结构研究和应用软件开发三者发展结合的产物。框架可以认为是粗粒度的构件的形式，它通常以框架库的形式表现。框架的关键还在于框架内对象间的交互模式和控制流模式。

2. 分类

框架是一类可复用的设计构件，它规定了应用的体系结构，阐明了整个设计、协作构件之间的依赖关系、责任分配和控制流程，表现为一组抽象类以及其实例之间协作的方法，它为构件复用提供了上下文（Context）关系。框架早期应用于图形用户界面设计，微软公司用于设计图形用户界面的类构造器（Microsoft Foundation Classes，MFC）就是典型的代表，MFC 后来成为事实上的工业标准。此时，框架的优点主要表现在为软件工程师提供改善软件模块性、一致性、扩展性和“逆控制”手段，进而提高软件的重用和品质保证。现在框架专注于具体的业务单元和应用领域，特别是 CORBA、J2EE 和微软的.NET 等框架的出现和成熟，他们封装了构架一个分布式应用的基础服务和功能组件，使软件开发工程师把注意力集中在业务分析，不用考虑底层结构的实现，能够更快地开发出高品质的系统以满足应用需求。

框架从其功能结构划分可分为白箱框架和黑箱框架，白箱框架的设计需要设计者了解要设计的框架的具体功能和内部结构，通过类的继承和设计模式的组合来生成框架。白箱框架是一个程序骨架，而衍生出的子类是这个骨架上的附属品。黑箱框架则是利用对象组合或代理机制实现，它的构建比白箱框架更难。

框架从复用的粒度划分，可分为：

（1）系统架构框架。这些框架能够使基础系统的开发变得简单、移植性好、高效，如操作系统的基础架构、通信框架、用户界面框架和语言处理工具，它们用于构建系统的内部组织，而不直接面向最终客户；

（2）中间件集成框架。它们通常用于在分布式系统中集成应用和组件，用于提高软件的模块化、复用性和扩展性，无缝集成分布式应用环境中的应用。例如，ORB 框架、消息中间件和事务中间件；

（3）企业应用框架。这种框架可应用于各行各业，直接面向具体的应用。

框架强调的是软件的设计重用性和系统的可扩充性。在软件开发过程中应用框架技术

以缩短大型应用软件系统的开发周期，提高开发质量。与传统的基于类库的面向对象技术比较，框架更注重于面向专业领域的软件重用，具有领域相关性，可以对现有构件根据框架进行复合，生成可运行的系统。框架的粒度越大，其中包含的领域知识就更加完整。

3.4.2 支持动态演化的框架

目前能很好地在体系结构层次支持软件演化的框架主要有 J.Dowling 等人设计的 K-Component 框架元模型。它主要通过提供体系结构元模型和适配契约来支持系统动态重配置，在 K-Component 元模型中，一个有类型的有向配置图被用来表示应用系统的软件体系结构，其中图节点表示构件接口，类型标签表示构件，有向边表示连接件。一个反射机制被设计用来在这个有类型的有向配置图和目标系统间建立因果连接，使得配置图反映系统的真实软件体系结构。

1. K-Component 模型机制

我们所建立动态软件体系结构的方法主要是采用动态反射技术。一个支持体系结构反射的系统主要依靠它的软件体系结构，例如，它的构件、连接件配置图作为体系结构元模型能够在运行期间显性的被观察和更改。体系结构元模型的改变将导致软件体系结构本身的更改，因此这种体系结构具有反射性。我们也提供了一个单独的自适应约束描述语言（称为自适应约束）用来编写反射程序，它允许程序控制怎么以及什么时候去进行运行时候的软件重配置。对体系结构的重配置操作被实现为图形转换，保证在重配置阶段以及重配置之后体系结构的安全性和完整性。在我们当前的实现工具中，重配置操作允许构件和连接件的替换策略，同时维护了一个软件体系结构的静态配置图。

我们把软件体系结构配置具体化为一个有类型的连通图。里面的顶点表示接口，标签为构件的实现；连线表示连接件，标签为连接件的属性。一个顶点被模型化为接口和实现（构件）。一条边被模型化为一个三角的 i->lj，它包含了源和目标顶点 i 和 j，边的标签为 l。边的标签表示连接件的可重配置属性，例如改变它的交互协议或者整套安装拦截器。配置图的根顶点是一个特殊的顶点类型，是程序的入口点。它们一般为 C++/Java 中 main 函数的实现。在整个配置图中可以出现环结构，同时这些环通常用环连接件来模型表示。一个元层的构件（称为配置管理，如图 3-21 所示）通常负责对软件体系结构的配置图进行存储和管理。

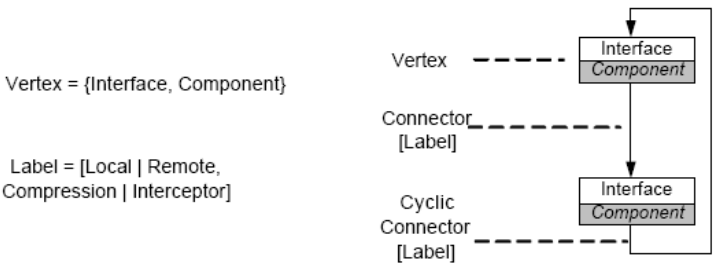


图 3-21 类型配置图

在前面已经提到，我们将动态重配置模型化为带约束的图形变换，明确地讲，在反射

编程中被称为自适应约束。图形变换是一种基于规则的配置图操作。如图 3-21 所示，这些规则定义了怎样以及何时去变换一个图。接口和连接件在我们的图中分别用顶点和边表示，它们描述了系统中在图形变换期间被保护的部分。构件和连接件的属性在图中分别用顶点和边上的标签表示，它们主要描述了图中在图形变换期间被重写的部分。因此，我们动态重配置模型被强迫用来替换系统的重配置图中的构件以及改变连接件的路由策略。在系统运行期间允许加入新的服务将导致一些开放性问题，如系统中已存在的构件和已存在的客户端怎样在运行期间去感知以及访问这些新的服务接口。对自适应软件而言，我们并没有将动态重配置模型看作为过度的约束。实际上，它能通过约束系统可能的动态重配置来帮助编程人员实现想要的动态演化。

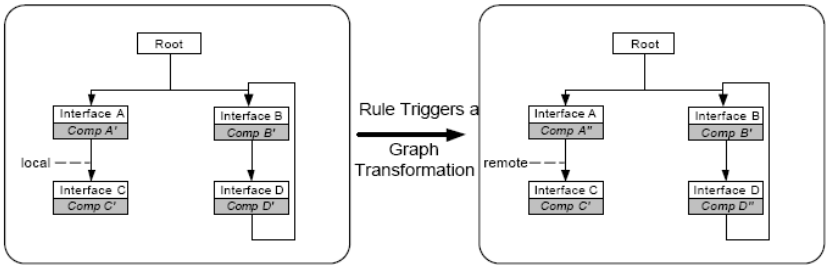


图 3-22　软件体系结构动态重配置转换图

既然图变换能保证变换后的规则结果还是图形，那么如果图规则是在图上的事务处理操作，我们就能保证系统的完整性和一致性。然而，在实际应用中，图变换可能只影响配置图中的一部分而不是锁定整个图，通常用重配置协议来确保只有那些在变换中有影响的定点必须处于安全状态。我们的重配置协议缩短了重配置阶段的时间，我们还允许在重配置阶段客户可以同时请求没有冻结的构件。然后，计算和改编代码当它在构件中冻结计算卷入到一个重配置中的时候通过重配置协议来进行关联。构件状态只能通过计算来更改，而不是重配置操作。

我们的重配置协议有以下这些：

（1）重配置操作调用变换重配置图操作；

（2）关联新的目标配置，给在目标配置中需要更新的源构件和连接件发送一个冻结消息；

（3）执行配置图的变换，例如，删除连接、创建、迁移状态、删除和连接新的构件或者改变连接件策略；

（4）在冻结连接件中的恢复处理。

我们假设一个重配置操作在限定的时间内完成，同时它的发起人（构件管理者）也知道什么时候结束。重配置协议的一个另外的优点就是负责将旧构件的状态值迁移到新的构件状态值上去，从而保证系统状态的完整性。构件状态的成功迁移需要构件开发者为他们的构件实现一个复制构造器接口。一个配置管理需要负责实现重配置操作以及保证重配置协议中操作的正确性。

为了清晰地分离出自适应描述代码和功能代码，我们采用了一种分离语言来将自适应逻辑描述成自适应约束。一个自适应约束集包含了一系列软件体系结构重配置变换条件规

则。自适应约束用来指定系统的体系结构约束，描述怎样以及何时安全地进行软件体系结构重配置。因为体系结构约束代表了配置、构件和连接件的属性或者断言，所以我们的自适应约束需要一个机制来读取这些属性或者断言。我们提供了自适应事件（adaptation events）作为允许自适应约束来自元层配置和基层构件和连接件的体系结构约束信息的机制。自适应事件同时还能从基层的构件和连接件中将元层的自适应规约解耦出来。更有效的是，他们提供了在自适应代码和功能代码中的运行时关注隔离线。因为基层代码不能这样就允许自适应约束被动态的加载和卸载。

自适应约束在自适应约束描述语言（ACDL）中被详细定义了。它们主要由一系列条件语句、自适应事件的测试以及重配置操作和自适应事件的连接组成。一个配置工具实现了 ACDL 中的自适应约束、软件体系结构说明以及用一门具体的语言来实现，例如我们的原型系统用 C++编写。它通过将 K-Component 框架中的抽象和临时类创建专用具体的工具以及将它们绑定到软件体系结构上来产生一个软件体系结构的可执行工具以及它的元模型。自适应约束在运行时通过元层对象来表示，同时通过配置管理来进行部署和管理。

2. K-Component 模型

K-Component 框架模型提供了一个配置管理器用来存储体系结构元模型以及实现体系结构层次的重配置操作。配置管理器同时也是运行时的容器，主要用来部署、计划和执行自适应约束，它还能够随意地为运行时候的自适应约束的加载和卸载提供一个程序接口。它被实现为一个主动对象。

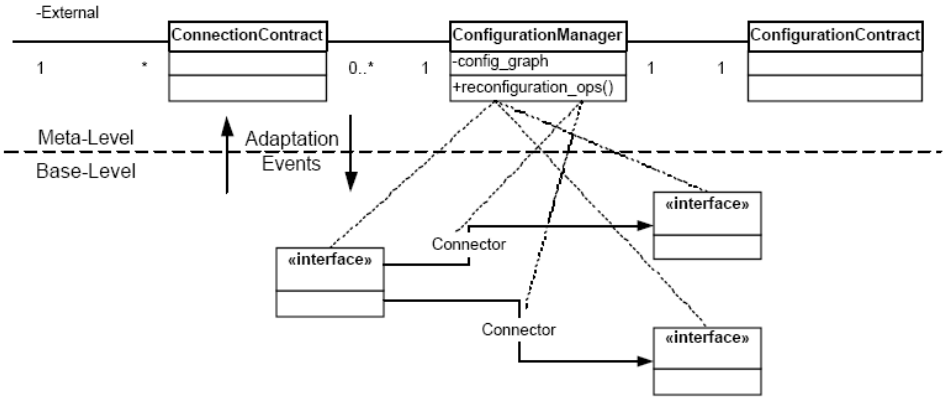


图 3-23 元层配置管理器

（1）构件和连接件

接口定义语言-3（IDL-3）主要用来定义体系结构构件和接口，该定义语言通过 provides 和 users 接口提供了显性依赖管理支持。这些显性的依赖用来帮助产生构件和连接件的配置图。其中的 emits 和 comsumes 事件主要用来说明自适应时间。连接件被实现为与 provides 和 users 接口相关的类型对象，例如构件上的端口。通过在 K-Component 框架中抽象连接件的专用化和模板化，就能够从 IDL-3 构件定义中自动产生连接件。连接件提供一个重配置接口，该接口拥有 link_component 和 unlink_component 操作，配置管理程序通过这个接口来实现图的重写操作。

（2）具体化软件体系结构以及产生体系结构元模型

我们在 K-Component 的元模型工具中采用 C++来替换 ADL 语言描述软件体系结构。这样将有利于编程人员不用为了描述应用程序的体系结构而去学另一种新的语言。然而，也还有许多 ADL 规格说明中的抽象概念以及程序术语需要表示，例如接口、连接件和绑定操作。下面的代码显示了怎样绑定一个连接件到一个构件的接口上。

Connector<Interface>* connector = Factory<Interface>::bind();

配置工具用来通过建立一个构件定义和连接件的依赖图来自动产生体系结构元模型，在系统工具语言中已经被定义了，如 C++。配置工具产生一个具有类型的有向系统配置图，该系统配置图用定点表示接口，边表示连接件来作为 XML 描述符。编程人员能够通过编辑 XML 配置描述符中的接口标签（该标签所指向实际的构件）将接口绑定到实际的构件实例上面。一旦这些构件实例都与所有的接口进行绑定，同时还附加了自适应约束到体系结构上后，那么软件体系结构就能通过配置管理工具进行具体化。

3.4.3　框架的应用

3-in-1 电话是一个基于蓝牙通讯和无线电话用例场景的可自适应应用程序。3-in-1 电话可以动态地使软件系统适应 3 种不同功能的电话。在蓝牙访问点 (BAP) 范围内，作为一个无线电话来计算费用。当用户离开 BAP 范围以外后，电话机就能动态地将它的功能重配置为蜂窝电话，从而改用蜂窝电话的计费方式。最后，当电话进入到另一个蓝牙电话区域内，它将作为“步话机”不收取任何电话费用。

3-in-1 电话能够被模型化为包含了 3 个构件的 K-Component 软件体系结构，这三个构件分别为：电话听筒，传送高质量服务（依靠网络连接的比特率）的声音/多媒体构件以及网络构件（蓝牙或者 GPRS），如图 3-24 所示。一个自适应约束声明描述了何时进行体系结构的重新配置。

if (BAP_Not_Available) change_configuration(“GPRS Config”);

3-in-1 电话体系结构中的动态重配置，如图 3-25 所示，由一个自适应事件 BAP_Not_Available 进行触发。重配置操作将涉及替换蓝牙 N/W 构件和同 GPRS 构件类似的 voice/multimedia 回放构件。重配置协议处理状态的迁移以及构件的装载/卸载，连接和断开操作。

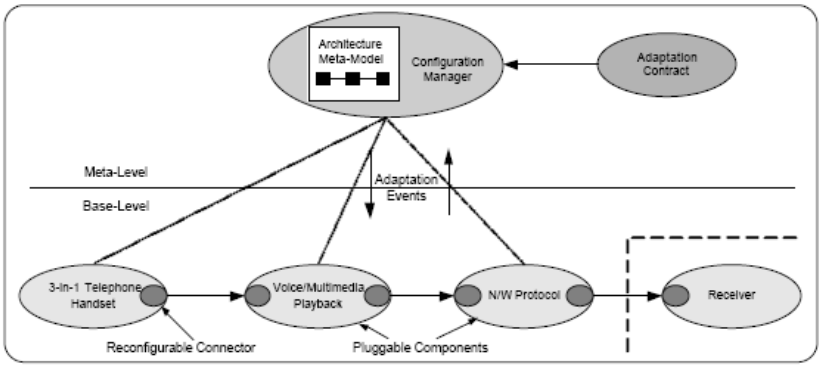


图 3-24　K-Component 在三位一体电话程序中的应用

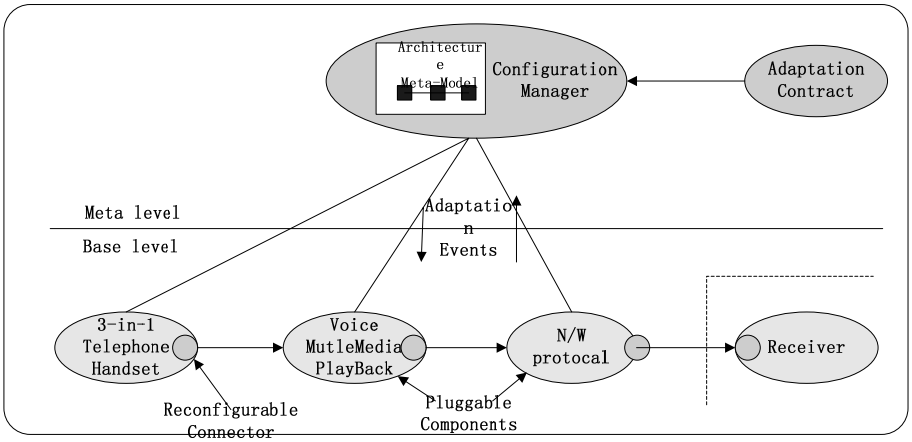


图 3-24 K-Component 在三位一体电话程序中的应用（续）

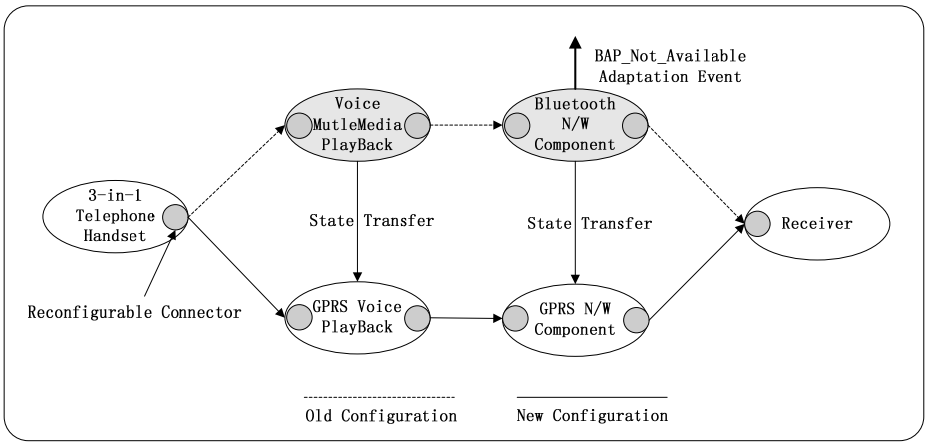


图 3-25 三位一体电话的重配置

3.5 应用软件体系结构风格

在研究动态软件体系结构中，体系结构风格和模式的研究和应用是一个重要的研究领域。体系结构风格和模式作为成功的软件设计经验和知识，它一方面可以被软件设计人员进行重用，从而可以避免大量的重复劳动，同时有利于提高软件质量、降低软件开发成本。另一方面，由于部分软件体系结构风格和模式的本身灵活性和构造性，它有利于软件的动态演化。

本节将首先对软件体系结构风格作简单的介绍，给出体系结构风格的概念以及常见的几种体系结构风格；其次将以流行的 C2 风格为重点，介绍几种支持软件动态演化的体系结构风格；最后介绍体系结构风格在软件设计中的应用。

3.5.1　体系结构风格概念和分类

1. 体系结构风格的概念

在应用设计模式那一节已经提到：目前关于风格和模式概念还没有完全一致的定义，许多书上将模式和风格理解为同一概念，本书将体系结构风格和模式定义为两种不同的概念。通常而言，模式大多数被用在软件的构件层次，主要是类与类之间的关系。而体系结构风格通常被认为是体系结构层次，表现为构件与构件之间的关系。因此，风格的粒度较模式要大，便于体系结构层次的软件复用。

模式并不是简单的某一环境中一组问题的解决方案，某一类问题的解决方案如果不能在软件开发过程中再现，那么也不能称之为模式。因此，如果一个解决方案是多个系统中精炼出来的，并且是一个成功的设计，那么这种解决方案就可以称为模式。

而大多数体系结构风格是以解决某组初始质量属性（软件的质量属性主要包括软件的功能属性和非功能属性，其中功能属性是指软件必须实现的功能部分，而非功能属性是指除软件功能以外的一些属性，如事务处理、QoS 管理、易演化性等）或问题的方式来描述的。因此，从某种意义上讲，风格也可以称为模式，但风格并不是一种解决方案，而是一种解决方案的框架。软件设计师在选择某一风格进行特定软件系统开发时，同时也限定了它必须遵循该框架内的约束。即只能在限定的这些解决方案中进行选择，从而降低了软件开发的复杂性。

体系结构风格通常根据抽象元素（构件和连接件）以及它们的拓扑结构（关系和属性）进行描述。根据 Shaw 和 Garlan 的定义：

软件体系结构={构件，连接件，约束}

因此，软件体系结构可以用来表示体系结构风格。1992 年，Perry 和 Wolf 提出了软件体系结构风格的概念，将那些公认的、被多次成功使用的软件系统结构称为软件体系结构风格。Metayer 给出了一个较为直观的定义：软件体系结构风格是一类具有许多共同特征的软件体系结构的共同描述。软件体系结构可以看作某种软件体系结构风格的具体实例。

2. 体系结构风格的分类

体系结构风格的分类方式有多种，其中最典型的是 Mary 和 David 根据信息交互的不同类型对体系结构风格的分类：

（1）数据流系统：

① 批处理

② 管道-过滤器

（2）调用/返回系统：

① 主程序/子程序

② 面向对象的系统

③ 分层系统

（3）独立构件

① 进程通信

② 基于事件的隐式调用

（4）虚拟机

① 解释器

② 基于规则的系统

（5）以数据为中心的系统

① 数据库

② 超文本系统

③ 黑板

除了这些分类外，另外还有：固定术语派生而来的体系结构风格，如 GenVoca、C2、REST；特定领域体系结构风格，如过程控制、模拟器；特定结构体系结构风格，如分布式处理、状态转移系统；不同体系结构集成的异构体系结构风格等。

3.5.2 支持动态演化的体系结构风格

软件的动态演化需要在运行期间进行功能的升级或修复，因此，在设计期间应用的体系结构风格将直接关系到软件能否支持动态变更。上一小节主要对一些常见的软件体系结构风格进行了分类，其中大多数风格并不支持运行期间系统的演化。在本节主要介绍几种常见的支持软件动态演化的体系结构风格。

1. 管道-过滤器风格

（1）描述

管道-过滤器（pipe and filter）风格最早出现在 UNIX 系统中，它是一种数据流系统，通常以数据如何在系统中交互为特征来进行刻画。数据流体系结构中一般有两个或多个数据流处理构件，每个构件都将输入数据转化为输出数据，并且必须顺序对数据进行转换。在管道－过滤器风格中，一个处理组件通常有两个输出（一个标准输出和一个错误输出）以及一个输入（标准输入）。用来处理输入输出参数的元素称为端口，端口通常负责接收和发送数据。端口通常附属在构件之上，构件与外界的交互都是通过端口来完成。两个彼此相连的端口必须满足管道－过滤器风格约束。一个过滤器通常有三个端口，其中错误输出可以与两个不同的处理组件进行相连。通常把构件看作风格中的过滤器，把连接件看作管道，它位于构件之间，进行数据的传递。在管道－过滤器风格中，构件是作为一个独立的实体存在，各个构件之间不能直接相连，而是通过连接件(即管道)进行连接，因此每个构件都不需要了解其他构件信息，从而避免了构件之间的直接耦合。该风格的结构如图 3-26 所示。

（2）特点

管道过滤器的优点主要有以下几点：

① 每个构件作为独立的实体存在，构件与构件之间不存在直接耦合，因此理解系统和对系统进行演化。

② 支持特殊的分析，如吞吐量分析、死锁分析。

③ 支持并发执行。

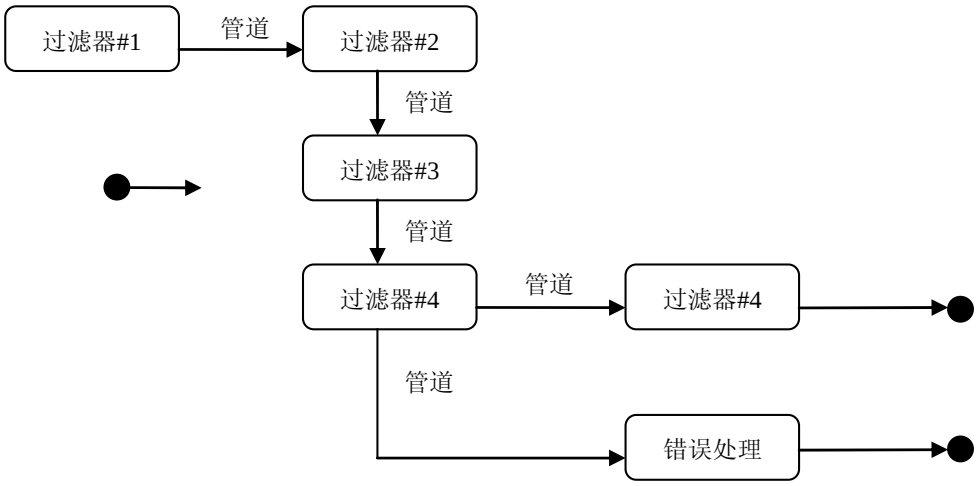


图 3-26　管道-过滤器风格体系结构

管道过滤器的缺点主要有以下几点：

① 经常导致批处理风格的设计。

② 对于交互式的应用程序不适用。

③ 在处理两个独立但是相关的流之间的同步时可能会遇到困难。

④ 可能会强制要求所有过滤器都要能达到公共的最低传输，这导致对于每个过滤器的解析输入和格式化输出要作更多的工作，从而增加了系统的复杂度。

（3）例子

UNIX 的 shell 程序就是使用了管道－过滤器风格。编译器也是一个典型的例子：词法分析→句法分析→语义分析→代码生成。

（4）如何支持动态演化

管道－过滤器风格采用连接件作为管道来传输数据，所有数据的交互都是通过连接件来完成，构件扮演过滤器的角色完成数据的处理。它不需要了解处理后的数据信息将流向哪个构件，同时也不需要知道输入的结果来自哪个构件，它只需要指定输入的格式，凡是满足这种格式的数据，它都可以接收并进行处理。因此，它便于运行时构件的替换。当一个构件需要动态替换现有的一个构件时，它只需要关心原有的构件的端口，由于数据的输入和输出都是通过端口来完成，因此需要用于替换的构件与现有的构件端口兼容。必须满足管道过滤器风格的约束才能替换。由于构件与构件之间不存在直接的相互调用，因此可以不考虑其他构件的运行状态和运行机制。

在动态替换一个构件之前，首先保证端口与现有端口兼容，如果新的构件端口不能兼容，则直接更改与之相连的连接件。由于连接件作为一种显性的实体存在于整个软件的运行当中，可以通过反射机制来实现连接件的变更。然后将连接件的连接从原有构件导向新的构件，这样就完成了构件的替换。当然，这里我们所考虑到的动态替换是假设构件作为一个无需保留状态的计算实体的前提下进行的。构件替换后即处于运行当中。

2. C2 风格

（1）描述

C2 是一种基于构件和消息的体系结构风格，适用于大型频繁交互的层次型图形界面软件的体系结构描述。它的许多思想来源于 Chiron-1 用户界面系统，因此它以 Chiron-2 来命名，简称 C2。

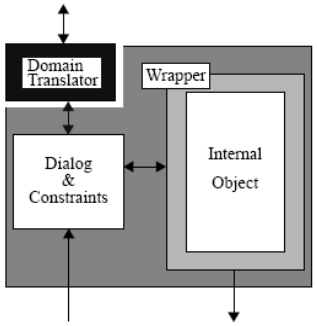


图 3-27 一个 C2 构件的内部体系结构

C2 风格的整个体系结构是一个层次网络，主要由具有计算功能的实体构件以及连接这些构件的连接件构成，构件之间的所有交互信息都必须以消息驱动的方式进行。使用 C2 风格的软件体系结构必须满足 C2 的约束，即下层独立原则：每个构件只能对层次高于自己的构件提供服务，而不能感知到低于自己层次的构件所提供的服务，这样有利于系统的扩展和维护，同时还有利于系统的演化。如图 3-27 为一个 C2 风格实例。

在 C2 风格的体系结构中，所有构件之间的交互都是通过连接件来完成的，它将构件的计算功能和交互功能分离开来，从而减小了构件之间的耦合。构件负责维护状态，进行操作，通过两个接口(顶端接口和底端接口)和其他构件交换消息。该风格同时还存在一些拓扑约束：每一个构件都有一个顶部和底部边界，在两个边界上各有一个端口。这些约束极大地简化了构件的添加、删除以及构件的重连接等操作。C2 中的连接件也有一个顶部和底部，但是其交互端口的数目是由连在它上面的构件数量来确定的。一个连接件能够连接任意多的构件和其他连接件，这样使得 C2 中的连接件能完成系统运行时的重新绑定。最后，所有构件之间都是通过连接件交换信息来完成异步通信。

C2 构件内部包含有 4 个部分（如图 3-27 所示）：内部对象(internal object)、包装器(wrapper)、对话(dialog)和域转换器(domain translator)。内部对象存储构件状态并实现构件所提供的操作。内部对象上的包装器监控所有的操作请求，并通过底端接口发送通知。对话负责把接收到的外部消息映射成内部对象上的操作。域转换器是可选的，它可以修改一些消息使其能被其他构件理解，这样，一个构件就能在特定的体系结构中适用。

（2）C2 风格主要特点

① 基于构件的风格：所有构件以可执行的二进制代码存在，独立于语言和平台，从而方便构件的重用和替换。

② 可扩展性：所有构件都作为独立的实体运行，可以有自己的控制线程，因此它对构件没有共享地址的要求，这就避免了因共享全部变量所带来的复杂性，有利于系统的扩展。

③ 演化性：构件与构件之间是一种松散耦合，都是通过连接件进行连接，因此，该风格本身的体系结构以及运行机制有利于对体系结构的演化。

（3）例子

图 3-28 是一个应用了 C2 风格的音乐播放器系统的音乐添加与删除程序，它所需要实现的功能就是：通过鼠标拖曳来实现音乐的添加与删除，当往播放器中添加一首音乐时，系统需要发出添加音乐的声音；当从播放器中删除一首音乐的时候，系统需要发出删除音

乐的声音。在该图中主要有 4 个构件：音乐操作者、音乐列表绘制器、图形绘制代理器、声音服务等。音乐列表绘制器在接收到音乐操作消息后执行列表显示，并同时创建一个抽象图形模型。图形绘制代理器检测所有对模型的操作，并最终根据检测到的操作消息在屏幕上显示一个图像。为了产生声音效果，在层次结构的底部部署了声音服务构件，它可以根据由音乐列表绘制器和操作者所发送的消息来播放特定的声音效果。图形绘制代理器和音乐列表操作器完全不用感知声音服务构件所提供的服务。同样，音乐操作者也无需感知它所选择的对象是否为可视化的。

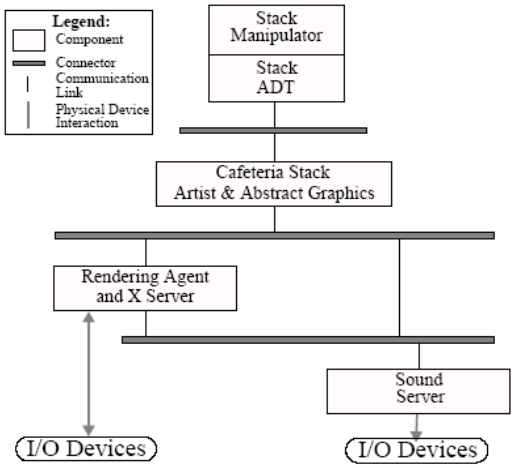


图 3-28　一个可视化音频播放器

（4）如何支持动态演化

体系结构的 3 个组成要素：构件、连接件和体系结构配置，都有一个演化过程。C2 对于构件演化的支持是通过类的子类型实现的。C2 有许多专门的 ADL 用于描述 C2 风格，例如 C2 SADEL（Software Architecture Description & Evolution Language）是用于描述 C2 风格的体系结构的 ADL，它为体系结构的演化提供了特别的支持。C2 的动态管理机制对体系结构运行时的变化不加限制，专门提供了体系结构变更语言 AML（Architecture Modification Language）。在 AML 中定义了一组在运行时可插入、删除和重新关联体系结构元素的操作。如 addcomponent、weld 等。

3.5.3　体系结构风格的应用

这一部分主要讲述体系结构风格在实际例子中的应用，为了突出说明如何使用体系结构风格来支持软件的动态演化，这里主要以 C2 风格为例来讲解。同时使用原型系统来实现系统的动态演化。

本书采用一个货物路由系统来说明基于 C2 风格系统如何进行动态演化。图 3-29 显示了系统的用户接口。顶部的 3 个列表框代表了 3 个货物发送口，它分别代表两个空运跑道和一个火车站。当货物运到发货口的时候，就在这个发货口的列表框中加上一项。系统跟踪每一个装运的物品种类、重量以及在发货口的闲置时长。中间的文本框显示了到目的仓库的可用的装运工具。系统显示了工具的名称、最大速度以及最大装载量。底部的文本框

显示了每一个仓库的名字、最大容量以及当前可用容量。最终用户通过用户接口选择或填写运送货物，然后点击发送按钮。

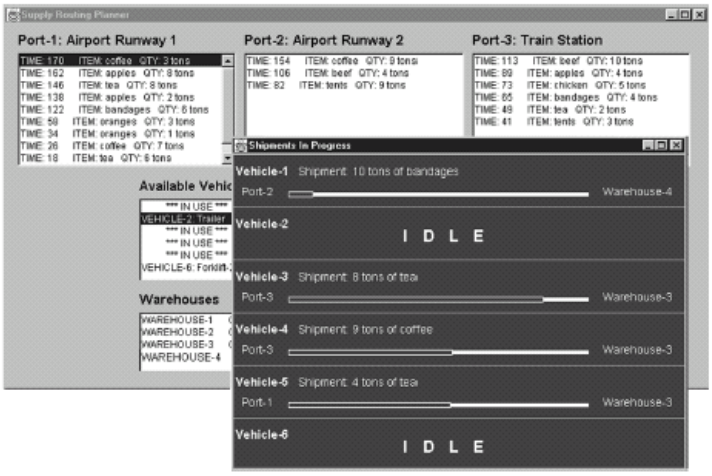


图 3-29 更新后的货物路由系统界面

图 3-30 描述了基于 C2 体系结构风格的货物运送系统的体系结构。Ports, Vehicle 和 Warehouses 构件属于抽象数据类型，分别用来保存发货口、运输工具以及仓库的状态信息。Telemetry 构件确定商品何时到发货口，并跟踪货物从发送开始直到被运送到仓库。Port Artist, Vehicle Artist, and Warehouse Artist 构件主要负责为终端用户图形化表示各自的 ADTs 状态. 当用户点击“Route”按钮时，Router 构件就会发送一条消息给 Telemetry 构件并提供终端用户最后所选择的港口、运输工具以及仓库。Graphics 构件用 JAVA AWT 图形包来绘制设计人员的绘制请求。

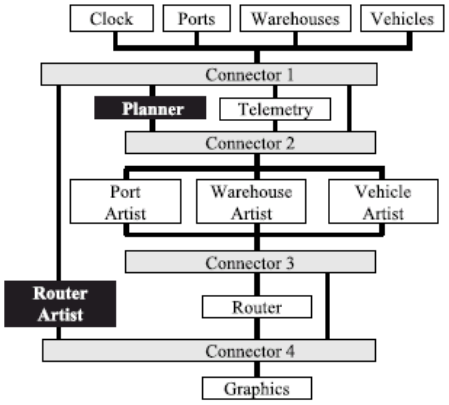


图 3-30 基于 C2 风格的货物路由系统的体系结构

ArchShell 主要用来添加一个新的可视化的图形发货模型，Extension Wizard 脚本主要用来添加自动化的计划构件，该构件帮助用户选择最优的发货决策。所有的变更都在货物发送系统执行的时候进行。我们现在描述 ArchShell 和 Extension Wizard 在添加新的功能到系统的使用方法。

添加一个新的可视化需要添加一个 Router Artist 构件到体系结构中。我们在连接件 1 和连接件 4 中添加一个 Router Artist 构件，因为它用到了由 Port, Warehouse, and Vehicle ADTs 所提供的通知消息和利用 Graphics 构件来绘制图形。架构师可以用 ArchShell 的“add component”命令来添加一个构件，利用“weld”命令将它连接到总线上去，采用“start”命令来向刚刚新建的构件发送信号，通知它应该接收执行周期。

添加自动化的计划需要添加一个 Planner 构件到体系结构中去，新的 Planner 构件加在 Connector 1 下面，因为它主要用来监控 ADTs 的状态从而选择最优的路线。图 3-30 显示了修改脚本的关键部分，当安装了变更时，则由 Extension Wizard 来执行。脚本确定了体系结构模型是否为货物发送系统，然后查看模型来确定构件 Planner 必须连接的连接件的名字。如果操作中的任何一步失败了，就抛出一个异常来取消安装。如果变更所依赖的体系结构元素被其他体系结构修改所改变了，那么这个操作就可能失败。

以上原型系统在 4.4 节中将有详细的介绍，该原型系统专门用于建立 C2 风格软件。它支持对软件的动态演化，通过 C2 风格的拓扑约束，保证演化后系统的完整性与一致性。

3.6　AOP 技 术

软件设计因为引入面向对象思想而逐渐变得丰富起来。“一切皆为对象”的定义，使得程序世界所要处理的逻辑简化，开发者可以用一组对象以及这些对象之间的关系将软件系统形象地表示出来。而从对象的定义，进而到模块，到组件的定义，利用面向对象思想的封装、继承、多态的思想，使得软件系统开发可以像搭建房屋那样循序渐进，从砖石到楼层，进而到整幢大厦的建成。应用面向对象思想，在设计规模更大、逻辑更复杂的系统时，开发周期反而能变得更短。其中自然需要应用到软件工程的开发定义、流程的过程控制，乃至于质量的缺陷管理。但从技术的细节来看，面向对象设计技术居功至伟。然而，面向对象设计的唯一问题是，它本质是静态的、封闭的，任何需求的细微变化都可能对开发进度造成重大影响。

可能解决该问题的方法是设计模式。Gof 将面向对象软件的设计经验作为设计模式记录下来，它使人们可以更加简单方便地复用成功的设计和体系结构，帮助开发人员做出有利于系统复用的选择。设计模式解决特定的设计问题，使面向对象设计更灵活、优雅，最终复用性更好。然而，设计模式虽然给了我们设计的典范与准则，通过最大程度的利用面向对象的特性，诸如利用继承、多态，对责任进行分离、对依赖进行倒置，面向抽象、面向接口，最终设计出灵活、可扩展、可重用的类库、组件，乃至于整个系统的架构。在设计的过程中，通过各种模式体现了对象的行为、暴露的接口、对象间关系，以及对象分别在不同层次中表现出来的形态。然而鉴于对象封装的特殊性，“设计模式”的触角始终在接口与抽象中大做文章，而对于对象内部则无能为力。

面向方面编程（Aspect-Oriented Programming，AOP）正好可以解决这一问题。它允许开发者动态地修改静态的 OO 模型，构造出一个能够不断增长以满足新增需求的系统，同现实世界中的对象会在其生命周期中不断改变自身一样，应用程序也可以在发展中拥有新的功能。AOP 利用一种称为“横切”的技术，剖开封装的对象内部，并将那些影响了多个

类的行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任，例如事务处理、日志管理、权限控制等封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

AOP 是施乐公司帕洛阿尔托研究中心（Xerox PARC）在上世纪 90 年代发明的一种编程范式。但真正的发展却兴起于近几年对软件设计方兴未艾的研究。由于软件系统越来越复杂，大型的企业级应用越来越需要人们将核心业务与公共业务分离。AOP 技术正是通过编写横切关注点的代码，即“方面”，分离出通用的服务以形成统一的功能架构。它能够将应用程序中的商业逻辑同对其提供支持的通用服务进行分离，使得开发人员从重复解决通用服务的劳动中解脱出来，而仅专注于企业的核心商业逻辑。因此，AOP 技术也就受到越来越多的关注，而应用于各种平台下的 AOP 技术也应运而生。但由于 AOP 技术相对于成熟的 OOP 技术而言，在性能、稳定性、适用性等方面还有待完善，同时 AOP 技术也没有形成一个统一的标准，这使得 AOP 技术的研究更具有前沿性的探索价值。

3.6.1 AOP 技术简介

1. AOP 技术起源

AOP 问世的头十年里，除了极少数时间，在大多数情况下都是归于沉寂。然而这段时间，业内许多人开始谈论 AOP，越来越多的技术人员逐渐意识到 AOP 的巨大优势以及它潜在的力量，而企业的 IT 部门则想了解 AOP 将会对他们的应用程序体系结构产生何种影响。近年来，随着计算机技术的快速发展和应用的不断普及，传统的整体设计和集中控制的软件开发方法也就越来越显示出其固有的局限性。

AOP 技术在 Java 平台下最先得到应用。就在 PARC 对于面向方面编程进行研究的同时，美国 Northeastern University 的博士生 Cristina Lopes 和其同事也开始了类似的思考。最终，美国国防先进技术研究计划署（Defense Advanced Research Projects Agency，DARPA）注意到了这项工作，并提供了科研经费，鼓励将二者的工作成果结合起来。他们通过定义一套 Java 语言的扩展系统，使开发者可以方便地进行面向方面的开发，这套扩展系统被称为 AspectJ。之后，AspectJ 在 2002 年被转让给 Eclipse Foundation，从而成为在开源社区中 AOP 技术的先锋，也是目前最为流行的 AOP 工具。

AspectWerkz 则是基于 Java 的动态的、轻量级 AOP 框架。AspectWerkz 仍然是开源社区中的产品，由 BEA System 提供赞助，开发者则是 BEA 的两名员工 Jonas Bonér 和 Alexandre Vasseur。最近版本是 AspectWerkz 2.0。2005 年 1 月，AspectJ 和 AspectWerkz 达成协议，同意将二者的成果综合到一起，取其精华创建一个单一的工具。他们合作的第一个发布版本为 AspectJ 5，它扩展了 AspectJ 语言，以支持基于 Annotation 开发风格而又支持类似 AspectJ 代码风格。AspectJ 5 也为 Java 5 的语言特性提供完全的 AOP 支持。

在 Java 阵营中，商用软件制造商 JBoss 在其 2004 年推出的 JBoss 4.0 中，引入了 AOP 框架和组件。在 JBoss 4.0 中，用户可以在 JBoss 应用服务器外部单独使用 JBoss AOP，该版本为 JBoss AOP 1.0，是在 2004 年 10 月发布的。在 2005 年，JBoss AOP 框架又发布了

1.3.0 版本，新版本对加载期织入（Weave）和切点（point cut）匹配的性能做了很大的优化，使应用程序的启动时间大大缩短。

作为轻型的 Framework，Spring 在开发轻量级的 J2EE 时，应用是非常广泛的。它通过 IoC 模式（Inversion of Control，控制反转模式）来实现 AOP，通常被称为 Spring AOP。在 2004 年，被作为 Spring 框架的扩展而发布，目前版本已更新到 1.1.3。Spring AOP 作为一种非侵略性的、轻型的 AOP 框架，开发者无需使用预编译器或其他的元标签即可在 Java 程序中应用 AOP。目前，AOP 的功能完全集成到了 Spring 事务管理、日志和其他各种特性的上下文中。

在.Net 的阵营中，AOP 技术的应用远不如 Java 阵营对 AOP 的关注。2005 年 1 月，微软发布的 Enterprise Library 提供了 7 种不同的“应用程序块（application blocks）”。有个别专家认为，这些组件可以被认为是方面。但该观点并未得到一致的认同。事实上，在.Net 平台下，推动 AOP 技术发展的原动力并非微软，而是开源社区。虽然微软的技术专家们也听到了在.Net Framework 中增加 AOP 技术的群众呼声，但作为如此巨大的软件公司，要让它灵活地转变战略方向，显然是不太现实的。正因为此，才赐予了开源社区在 AOP 技术的研究与探索上一个巨大的发展空间。

与 Java 阵营中的 AOP 技术不同，目前在.Net 平台下的各种 AOP 工具，基本上还停留在实验室阶段。但一些在技术上领先且逐渐成熟的 AOP 产品，也在开源社区中渐露峥嵘。这其中主要包括 Aspect#，AspectDNG，Eos AOP 等。

Aspect#是基于 Castle 动态代理技术来实现的。Castle 源于 Apache Avalon 项目，其目的在于实现一个轻量级的 IoC 容器。Aspect#于 2005 年 6 月被收录为 Castle 的一个子项目。它是针对 CLI（.Net 和 Mono）实现的 AOP 框架，利用了反射、代理等机制。目前的 Aspect# 版本为 2.1.1。

AspectDNG 目前的版本为 0.7，仍然处于 beta 版的阶段。它的实现技术是基于 rail 的静态织入。Rail 属于 IL 级别下的代码织入，它自定义的一套 xml 格式的 ILML 语言，能够将原有的程序集拆散成 ILML 格式，以便于对静态程序集进行修改和扩展，从而达到静态织入的目的。因为 AspectDNG 是属于 IL 级别下的代码织入，因此在.Net 平台下，并不受具体的编程语言限制。

Eos AOP 与 AspectDNG 一样，仍然采用静态织入的方式，但从语法定义上，它更近似于 AspectJ 关于 AOP 的实现。它扩展了 C#语法，引入了 aspect、introduce、before、after 等关键字，并且提供了专用的 Eos 编译器。Eos 项目是于 2004 年 9 月开始启动，2005 年 6 月推出的 0.3.3 版本为最新版本，主要的开发人员为 Hridesh Rajan 和 Kevin Sullivan。Hridesh Rajan 为 Virginia 大学计算机系的研究生，Eos 项目最初是由他提出的；而后者则为该计算机系的副教授（Associate Professor）。所以自 Eos 诞生之初，就带有浓厚的学院派特色。

从 AOP 技术的整体发展来看，高性能、稳定、可扩展、易用的 AOP 框架是其趋势与目标。从上述对各种 AOP 技术的分析来看，AOP 技术无疑是具有共同特点的，而各种实现技术就是围绕着这些共性深入与延伸。下面将概要地介绍 AOP 的本质，以及它的技术要素。

2. AOP 技术概览

AOP 可以说是 OOP（Object-Oriented Programming）的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当需要为分散

的对象引入公共行为的时候，OOP 则显得无能为力。也就是说，OOP 允许定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在 OOP 设计中，它导致了大量代码的重复，而不利于各个模块的重用。

而 AOP 技术则恰恰相反，它代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，它们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。AOP 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。正如 Avanade 公司的高级方案构架师 Adam Magee 所说，AOP 的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

实现 AOP 的技术，主要分为两大类：一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。然而殊途同归，实现 AOP 的技术特性却是相同的，分别为：

（1）join point（连接点）：是程序执行中的一个精确执行点，例如类中的一个方法。它是一个抽象的概念，在实现 AOP 时，并不需要去定义一个 join point。

（2）point cut（切入点）：本质上是一个捕获连接点的结构。在 AOP 中，可以定义一个 point cut，来捕获相关方法的调用。

（3）advice（通知）：是 point cut 的执行代码，是执行“方面”的具体逻辑。

（4）aspect（方面）：point cut 和 advice 结合起来就是 aspect，它类似于 OOP 中定义的一个类，但它代表的更多是对象间横向的关系。

（5）introduce（引入）：为对象引入附加的方法或属性，从而达到修改对象结构的目的。有的 AOP 工具又将其称为 mixin。

上述的技术特性组成了基本的 AOP 技术，大多数 AOP 工具均实现了这些技术。它们也可以是研究 AOP 技术的基本术语。

（1）横切关注点。一个关注点（concern）就是一个特定的目的，一块我们感兴趣的区域，一段我们需要的逻辑行为。从技术的角度来说，一个典型的软件系统包含一些核心的关注点和系统级的关注点。举个例子来说，一个信用卡处理系统的核心关注点是借贷/存入处理，而系统级的关注点则是日志、事务完整性、授权、安全及性能问题等，许多关注点——即横切关注点（crosscutting concerns）——会在多个模块中出现。如果使用现有的编程方法，横切关注点会横越多个模块，结果是使系统难以设计、理解、实现和演化。AOP 能够比上述方法更好地分离系统关注点，从而提供模块化的横切关注点。

例如一个复杂的系统，它由许多关注点组合实现，如业务逻辑、性能、数据存储、日志和调度信息、授权、安全、线程、错误检查等，还有开发过程中的关注点，如易懂、易

维护、易追查、易扩展等，图 3-31 演示了由不同模块实现的一批关注点组成一个系统。

通过对系统需求和实现的识别，我们可以将模块中的这些关注点分为：核心关注点和横切关注点。对于核心关注点而言，通常来说，实现这些关注点的模块是相互独立的，它们分别完成了系统需要的商业逻辑，这些逻辑与具体的业务需求有关。而对于日志、安全、持久化等关注点而言，它们却是商业逻辑模块所共同需要的，这些逻辑分布于核心关注点的各处。在 AOP 中，诸如这些模块，都称为横切关注点。应用 AOP 的横切技术，关键就是要实现对关注点的识别。

如果将整个模块比喻为一个圆柱体，那么关注点识别过程可以用三棱镜法则来形容，穿越三棱镜的光束（指需求），照射到圆柱体各处，获得不同颜色的光束，最后识别出不同的关注点。如图 3-32 所示：

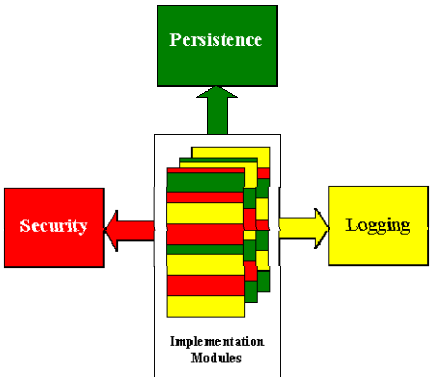


图 3-31　把模块作为一批关注点来实现

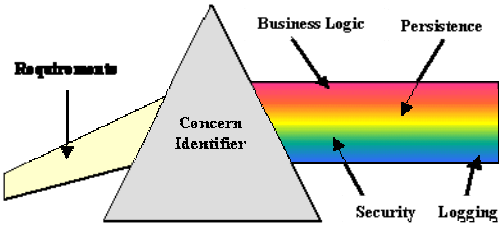


图 3-32　关注点识别：三棱镜法则

上图识别出来的关注点中，Business Logic 属于核心关注点，它会调用到 Security，Logging，Persistence 等横切关注点。

```
public class BusinessLogic
{
public void SomeOperation()
  {
    //验证安全性；Securtity 关注点；
    //执行前记录日志；Logging 关注点；
      DoSomething();
      //保存逻辑运算后的数据；Persistence 关注点；
      //执行结束记录日志；Logging 关注点；
   }
}
```

AOP 的目的，就是要将诸如 Logging 之类的横切关注点从 BusinessLogic 类中分离出来。利用 AOP 技术，可以对相关的横切关注点封装，形成单独的“aspect”。这就保证了横切关注点的复用。由于 BusinessLogic 类中不再包含横切关注点的逻辑代码，为达到调用横切关注点的目的，可以利用横切技术，截取 BusinessLogic 类中相关方法的消息，例如 SomeO-

peration()方法，然后将这些“aspect”织入到该方法中。例如图 3-33：

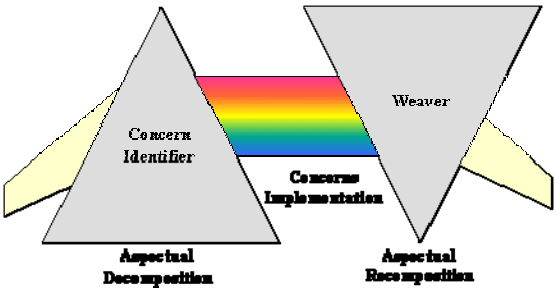


图 3-33　将横切关注点织入到核心关注点中

通过利用 AOP 技术，改变了整个系统的设计方式。在分析系统需求之初，利用 AOP 的思想，分离出核心关注点和横切关注点。在实现了诸如日志、事务管理、权限控制等横切关注点的通用逻辑后，开发人员就可以专注于核心关注点，将精力投入到解决企业的商业逻辑上来。同时，这些封装好了的横切关注点提供的功能，可以最大限度地复用于商业逻辑的各个部分，既不需要开发人员作特殊的编码，也不会因为修改横切关注点的功能而影响具体的业务功能。

为了建立松散耦合的、可扩展的企业系统，AOP 应用到的横切技术，通常分为两种类型：动态横切和静态横切。

（2）动态横切。动态横切是通过切入点和连接点在一个方面中创建行为的过程，连接点可以在执行时横向地应用于现有对象。动态横切通常用于帮助向对象层次中的各种方法添加日志记录或身份认证。在很多应用场景中，动态横切技术基本上代表了 AOP。

动态横切技术的核心主要包括 join point（连接点），point cut（切入点），advice（通知）和 aspect（方面）。在前面，我已经概要地介绍了这些术语分别代表的含义。下面将以一个具体的实例来进一步阐述它们在 AOP 动态横切中实现的意义。

考虑一个电子商务系统，需要对订单进行添加、删除等管理操作。毫无疑问，在实际的应用场景中，这些行为应与权限管理结合，只有获得授权的用户方能够实施这些行为。采用传统的设计方法，其伪代码如下。

```
public class OrderManager
{
    private ArrayList m_Orders;
    public OrderManager()
    {
       m_Orders = new ArrayList();
    }
    public void AddOrder(Order order)
    {
        if (permissions.Verify(Permission.ADMIN))
        {
            m_Orders.Add(order);
```

```
        }
    }
    public void RemoveOrder(Order order)
    {
        if (permissions.Verify(Permission.ADMIN))
        {
            m_Orders.Remove(order);
        }
    }
}
```

同样的，在该电子商务系统中，还需要对商品进行管理，它采用了同样的授权机制。

```
public class ProductManager
{
    private ArrayList m_Products;
    public ProductManager()
    {
        m_Products = new ArrayList();
    }
    public void AddProduct(Product product)
    {
        if (permissions.Verify(Permission.ADMIN))
        {
            m_Products.Add(product);
        }
    }
    public void RemoveProduct(Product product)
    {
        if (permissions.Verify(Permission.ADMIN))
        {
            m_Products.Remove(product);
        }
    }
}
```

如此一来，在整个电子商务系统中，核心业务包括订单管理和商品管理，它们都需要相同的权限管理，如图 3-34 所示。

毫无疑问，利用 AOP 技术，我们可以分离出系统的核心关注点和横切关注点，从横向的角度，截取业务管理行为的内部消息，以达到织入权限管理逻辑的目的。当执行 AddOrder() 等方法时，系统将验证用户的权限，调用横切关注点逻辑，因此该方法即为 AOP 的 join

point。对于电子商务系统而言，每个需要权限验证的方法都是一个单独的 join point。由于权限验证将在每个方法执行前执行，所以对于这一系列 join point，只需要定义一个 point cut。当系统执行到 join point 处时，将根据定义去查找对应的 point cut，然后执行这个横切关注点需要实现的逻辑，即 advice。而 point cut 和 advice，就组合成了一个权限管理 aspect。

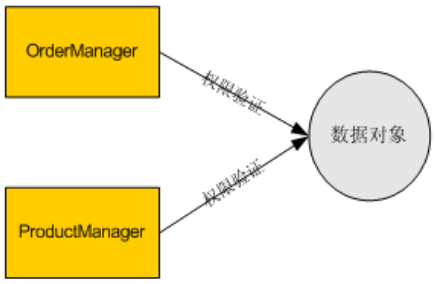


图 3-34　电子商务系统的权限验证实现

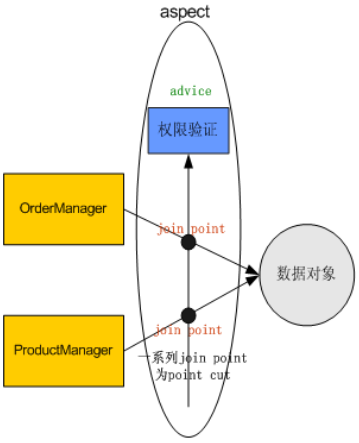


图 3-35　AOP 动态横切的技术实现

由于 aspect 是一个封装的对象，我们可以定义这样一个 aspect：

private static aspect AuthorizationAspect{……}

然后在这个 aspect 中定义 point cut，在 point cut 中，定义了需要截取上下文消息的方法。

```
private pointcut authorizationExecution():
execution(public void OrderManager.AddOrder(Order)) ||
execution(public void OrderManager.DeleteOrder(Order)) ||
execution(public void ProductManager.AddProduct(Product)) ||
execution(public void ProductManager.DeleteProduct(Product));
```

由于权限验证是在订单管理方法执行之前完成，因此在 before advice 中，定义权限检查。

```
before(): authorizationExecution()
{
    if !(permissions.Verify(Permission.ADMIN))
    {
        throw new UnauthorizedException();
}
}
```

通过定义了这样一个完整的 aspect，当系统调用 OrderManager 或 ProductManager 的相关方法时，就触发了 point cut，然后调用相应的 advice 逻辑。这样，OrderManager 和 ProductManager 模块就与权限管理模块完全解除了依赖关系，同时也消除了传统设计中不可避免的权限判断的重复代码。这对于建立一个松散耦合、可扩展的系统软件是非常有利的。

（3）静态横切。静态横切和动态横切的区别在于它不修改一个给定对象的执行行为。相

反，它允许通过引入附加的方法字段和属性来修改对象的结构。此外，静态横切可以把扩展和实现附加到对象的基本结构中。在 AOP 实现中，通常将静态横切称为 introduce 或者 mixin。

静态横切在 AOP 技术中，受到的关注相对较少。事实上，这一技术蕴含的潜力是巨大的。使用静态横切，架构师和设计者能用一种真正面向对象的方法有效地建立复杂系统的模型。静态横切允许您不用创建很深的层次结构，以一种本质上更优雅、更逼真于现实结构的方式，插入跨越整个系统的公共行为。尤其是当开发应用系统时，如果需要在不修改原有代码的前提下，引入第三方产品和 API 库，则静态横切技术将发挥巨大的作用。

举例来说，当前已经实现了一个邮件收发系统，其中类 Mail 完成了收发邮件的功能。但在产品交付后，发现该系统存在缺陷，在收发邮件时，未曾实现邮件地址的验证功能。现在，第三方产品已经提供了验证功能的接口 IValidatable。

```
public interface IValidatable
{
    bool ValidateAddress();
}
```

我们可以利用设计模式中的 Adapter 模式，来完成对第三方产品 API 的调用。我们可以定义一个新的类 MailAdapter，该类实现了 IValidatable 接口，同时继承了 Mail 类。

```
public class MailAdapter:Mail,IValidatable
{
    public bool ValidateAddress()
    {
        if(this.getToAddress() != null)
        {
            return true;
        }
        Else
        {
            return false;
        }
    }
}
```

通过引入 MailAdapter 类，原来 Mail 对象完成的操作，将全部被 MailAdapter 对象取代。然而，此种实现方式虽然能解决引入新接口的问题，但类似下面的代码，却是无法编译通过的。

```
Mail mail = new Mail();
IValidatable validate = ((IValidatable)mail).ValidateAddress();
```

必须将第一行代码作如下修改。

```
Mail mail = new MailAdapter();
```

利用 AOP 的静态横切技术，可以将 IValidatable 接口织入到原有的 Mail 类中，这是一种非常形象的 introduce 功能，其实现仍然是在 aspect 中完成。

```
import com.acme.validate.Validatable;
```

```
public aspect MailValidateAspect
{
    declare parents: Mail implements IValidatable;
    public boolean Mail.validateAddress()
    {
         if(this.getToAddress() != null)
         {
              return true;
         }
         else
         {
              return false;
         }
    }
}
```

静态横切的方法，并没有引入类似 MailAdapter 的新类，而是通过定义的 MailValidateAspect 方面，利用横切技术为 Mail 类 introduce 了新的方法 ValidateAddress()，从而实现了 Mail 的扩展。因此如下的代码完全可行。

```
Mail mail = new Mail();
IValidatable validate = ((IValidatable)mail).ValidateAddress();
```

3.6.2 动态 AOP

所谓动态 AOP，就是通过动态的 Proxy 模式，在目标对象的方法调用前后插入相应的特殊处理代码(如日志、性能报告等)。动态 AOP 机制的实现主要分为两种类型：使用 JDK 的动态代理 API 或字节码 Bytecode 处理技术。

其中用 JDK 的动态代理 API 的主要有 JBOSS4.0，Nanning。基于字节码的项目主要有 AspectWerkz 和 Spring。

下面将以 Spring 为例讲述如何应用动态 AOP 技术的到 Proxy 模式的 UML 图。

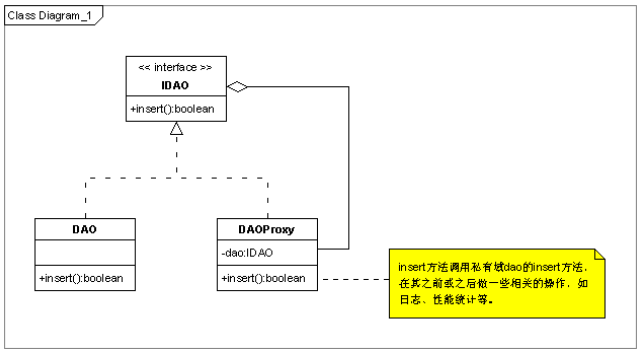


图 3-36 Spring 中的 Proxy 模式图

DAOProxy 类实现了 IDAO 接口，同时有一个 IDAO 类型的私有域，在调用 DAOProxy 的 insert 方法时，其调用私有域 DAO 的 insert 方法，同时做一些相关的操作，这就起到了代理的作用。

这只是静态的 Proxy 模式，但若是假设整个系统有几十个类似于 IDAO 的接口，都要实现 Log 功能，又不能修改接口，在接口中添加 log 函数。那只能再加几十个 Proxy 类了，这显然是一个繁琐的工作。因此，我们需要采用 Dynamic Proxy 模式。Spring 的事务处理，特别是日志和数据库回滚操作，都是基于 Java Dynamic Proxy 的 AOP 原理实现的。

看以下简单的代码就知道了：

TxHandler.java

```
public class TxHandler implements InvocationHandler {
          private Object originalObject;
          public Object bind(Object obj) {
                this.originalObject = obj;
            return Proxy.newProxyInstance(obj.getClass().getClassLoader(),
obj
                        .getClass().getInterfaces(), this);
          }
          public Object invoke(Object proxy, Method method, Object[] args)
                      throws Throwable {
                Object result = null;
            if (!method.getName().startsWith(“save”)) {
                      UserTransaction tx = null;
                      try {
                        tx = (UserTransaction) (new
InitialContext().lookup(“java/tx”));
                            result = method.invoke(originalObject, args);
                            tx.commit();
                  } catch (Exception ex) {
                            if (null != tx) {
                                  try {
                                        tx.rollback();
                                  } catch (Exception e) {
                                  }
                            }
                      }
                } else {
                      result = method.invoke(originalObject, args);
                }
                return result;
```

```
        }
    }
```

上面的代码并没有出现某个特定的接口名和引用类，所以说，它是适用于所有接口的实现的。TxHandler.invoke 方法将在被代理类的方法被调用之前触发。通过这个方法，我们可以在被代理类方法调用的前后进行一些处理，如代码中所示，TxHandler.invoke 方法的参数中传递了当前被调用的方法（Method），以及被调用方法的参数。

只需通过一个 Dynamic Proxy 对所有需要事务管理的 Bean 进行加载，并根据配置，在 invoke 方法中对当前调用的方法名进行判定，并为其加上合适的事务管理代码，那么就实现了 Spring 式的事务管理。

3.6.3 AOP 技术在 Java 平台中的应用

AOP 在实验室应用和商业应用上，Java 平台始终走在前面。从最初也是目前最成熟的 AOP 工具——AspectJ，到目前已经融和在企业级容器 JBoss 中的 JBoss AOP，均建立在 Java 平台上。

前面已经描述到，AOP 的目的就是将核心关注点和横切关注点分离，实际上这就是一种分散关注（seperation of concerns）的思路。在 Java 平台下，如果要开发企业级的应用，非 J2EE 莫属。一个 J2EE 应用系统只有部署在 J2EE 容器中才能运行，那么为什么要划分为 J2EE 容器和 J2EE 应用系统？ 通过对 J2EE 容器运行机制的分析，我们发现：实际上 J2EE 容器分离了一般应用系统的一些通用功能，例如事务机制、安全机制以及对象池或线程池等性能优化机制。这些功能机制是每个应用系统几乎都需要的，因此可以从具体应用系统中分离出来，形成一个通用的框架平台，而且，这些功能机制的设计开发有一定难度，同时运行的稳定性和快速性都非常重要，必须经过长时间调试和运行经验积累而成，因此，形成了专门的 J2EE 容器服务器产品，如 Tomcat JBoss、Websphere、WebLogic 等。

从 J2EE 将应用系统和容器分离的策略，我们能够看到 AOP 的影子。J2EE 应用系统就相当于 AOP 技术中的核心关注点，它的内容主要包括企业系统的商业逻辑；而 J2EE 容器则类似于横切关注点，实现的是通用的功能机制。不过业界在选择 J2EE 容器时，对于 EJB 这种重量级容器服务器而言，虽然欣赏其高效、稳定及企业级的容器服务，但对于整个容器的高开销、高成本以及过于复杂的解决方案均深怀戒心。因此，随着 J2EE 的逐步演化，“轻量级容器架构”通过开源社区如激流一般的驱动力，逐渐占据了 J2EE 技术的强势地位。而所谓“轻量级容器”与 EJB 提供的重量级架构的区别，就在于借助了 AOP 技术和 IoC（Inversion of Control，控制反转模式）机制，降低了代码对于专用接口的依赖性，以简短、轻便、专注、可移植的方式实现业务对象。事实上，我们看到的美好前景是，如果所有企业级服务都可以通过 AOP 机制提供给普通 Java 对象，那么应用服务器就不再有存在的价值了。

正是看到了 AOP 技术在企业级开发中的巨大潜力，而“轻量级容器”也唤起了改革 EJB 容器的呼声（事实上，最新的 EJB V3.0 标准就使用了轻量级容器模型），越来越多的 AOP 工具在 Java 平台下应运而生，从而形成了目前 AOP 工具百家争鸣的局面。其中，应用最为广泛的主要包括 AspectJ、Spring AOP 和 JBoss AOP 等。

第 4 章　动态演化的粒度

4.1　函数层次的动态演化

4.1.1　DLL 简介

Windows 系统平台上提供了一种完全不同的有效的编程方式和运行环境，您可以将独立的程序模块封装为较小的 dll 文件，并可以对它进行单独的编译和测试。在运行的过程中，只有当 EXE 程序确实要调用这些 DLL 模块的情况下，系统才会将他们装载到内存空间中。这种编程方式不仅减少了可执行程序文件的大小和对内存空间的需求，而且可以使得这些 DLL 模块可以同时对几个可执行程序共享调用。Windows 本身就是将一些主要的系统功能以这样的方式来实现的。

一般来说，DLL 模块是以某种磁盘文件的形式存放的，比如.DLL、.DRV、.FON、.SYS、.EXE 等形式的文件。它由全局数据、服务函数和资源组成，在运行过程中对系统加载到进程的虚拟空间里面，成为调用进程的一部分。如果与其他 DLL 模块之间没有冲突，该文件通常映射到进程虚拟空间的同一地址上。DLL 模块包含各种导出函数，用于向外界提供服务，DLL 模块可以利用自己的数据段，但没有自己的堆栈，使用与调用它的应用程序相同的堆栈。一个 DLL 模块通常在内存中只有一个实例，它实现了对代码的封装，而且 DLL 模块的编写与具体的编程语言以及编译器毫无关系。

在 Windows 环境中，每个进程都将自己的读写全局变量复制了下来。如果想要与其他进程共享内存，必须使用内存映射文件或者声明一个共享数据段。DLL 模块需要的堆栈都是从运行期的进程的堆栈中分配出来的。Windows 在加载 DLL 模块时将进程函数调用与 DLL 模块的导出函数相匹配。操作系统对 DLL 模块的操作仅仅是把 DLL 模块映射到需要它的进程的虚拟地址空间里去。DLL 模块中函数的代码所创建的任何对象（包括变量）都归调用它的线程和进程所有。

4.1.2　调用方式

在应用程序或另外一个 DLL 模块能够调用 DLL 模块中的函数之前，DLL 文件映象必须被映射到调用进程的地址空间中去。如果想要完成这样的操作，可以采用两种方法来实现，这就是接下来要提到的加载时的隐含链接和运行期的显式链接。

1. 静态调用方式

由编译系统完成对 DLL 的加载和应用程序结束时 DLL 卸载的编码（如还有其他程序

使用该 DLL，则 Windows 对 DLL 的应用记录减 1，直到所有相关程序都结束对该 DLL 的使用时才释放它），简单实用，但不够灵活，只能满足一般要求。

作为隐式的调用，需要把产生动态连接库时产生的.LIB 文件加入到应用程序的工程中，想使用 DLL 中的函数时，只需说明一下。隐式调用不需要调用 LoadLibrary()和 FreeLibrary()。程序员在建立一个 DLL 文件时，链接程序会自动生成一个与之对应的 LIB 导入文件。该文件包含了每一个 DLL 导出函数的符号名和可选的标识号，但是并不含有实际的代码。LIB 文件作为 DLL 的替代文件被编译到应用程序项目中。当程序员通过静态链接方式编译生成应用程序时，应用程序中的调用函数与 LIB 文件中导出符号相匹配，这些符号或标识号进入到生成的 EXE 文件中。LIB 文件中也包含了对应的 DLL 文件名（但不是完全的路径名），链接程序将其存储在 EXE 文件内部。当应用程序运行过程中需要加载 DLL 文件时，Windows 根据这些信息发现并加载 DLL，然后通过符号名或标识号实现对 DLL 函数的动态链接。所有被应用程序调用的 DLL 文件都会在应用程序 EXE 文件加载时被加载在到内存中。可执行程序链接到一个包含 DLL 输出函数信息的输入库文件（.LIB 文件）。操作系统在加载使用可执行程序时加载 DLL。可执行程序直接通过函数名调用 DLL 的输出函数，调用方法和程序内部其他的函数是一样的。

示例：

```
//Dlltest.h
#pragma comment(lib，"MyDll.lib")
extern "C"_declspec(dllimport) int Max(int a,int b);
extern "C"_declspec(dllimport) int Min(int a,int b);
//TestDll.cpp
#include
#include"Dlltest.h"
void main()
{int a;
a=min(8,10)
printf("比较的结果为%dn"，a);
}
```

2. 动态调用方式

由编程者用 API 函数加载和卸载 DLL 来达到调用 DLL 的目的，使用上较复杂，但能更加有效地使用内存，是编制大型应用程序时的重要方式。

作为显式的调用，是在应用程序中用 LoadLibrary 或 MFC 提供的 AfxLoadLibrary 显式地将自己所做的动态连接库调进来，动态连接库的文件名即是上面两个函数的参数，再用 GetProcAddress()获取想要引入的函数。自此，你就可以象使用如同本应用程序自定义的函数一样来调用此引入函数了。在应用程序退出之前，应该用 FreeLibrary 或 MFC 提供的 AfxFreeLibrary 释放动态连接库。直接调用 Win32 的 LoadLibary 函数，并指定 DLL 的路径作为参数。LoadLibary 返回 HINSTANCE 参数，应用程序在调用 GetProcAddress 函数时使用这一参数。GetProcAddress 函数将符号名或标识号转换为 DLL 内部的地址。程序员可

以决定 DLL 文件何时加载或不加载，显式链接在运行时决定加载哪个 DLL 文件。使用 DLL 的程序在使用之前必须加载（LoadLibrary），加载 DLL 从而得到一个 DLL 模块的句柄，然后调用 GetProcAddress 函数得到输出函数的指针，在退出之前必须卸载 DLL(FreeLibrary)。

示例：

```
#include ……
#include ……
void main(void)
{
typedef int(*pMax)(int a,int b);
typedef int(*pMin)(int a,int b);
HINSTANCE hDLL;
PMax Max
HDLL=LoadLibrary("MyDll.dll");//加载动态链接库 MyDll.dll 文件;
Max=(pMax)GetProcAddress(HDLL,"Max");
A=Max(5,8);
Printf("比较的结果为%dn", a);
FreeLibrary(hDLL);//卸载 MyDll.dll 文件;
}
```

4.1.3　重新编译问题及解决方案

1. 重新编译问题

首先，我们讨论一个在实际案例中可能遇到的问题：

如果已经建立了一个 DLL，导出了 CMyClass 类，客户也能正常使用这个 DLL，假设 CMyClass 对象的大小为 30 字节。现在我们需要修改 DLL 中的 CMyClass 类，让它有相同的函数和成员变量，但是增加了一个私有的成员变量 int 类型，那么 CMyClass 对象的大小就是 34 字节了。当直接把这个新的 DLL 给客户使用替换掉原来 30 字节大小的 DLL，客户应用程序期望的是 30 字节大小的对象，而现在却变成了一个 34 字节大小的对象, 这会导致客户程序出错。

类似的问题，如果不是导出 CMyClass 类，而在导出的函数中使用了 CMyClass，改变对象的大小仍然会有问题的。这个时候修改这个问题的唯一办法就是替换客户程序中的 CMyClass 的头文件，全部重新编译整个应用程序，让客户程序使用大小为 34 字节的对象。

这就是一个严重的问题，有的时候如果没有客户程序的源代码，那么就不能使用这个新版本的 DLL 了。

2. 解决方法

为了能避免重新编译客户程序，这里介绍两个方法：（1）使用接口类。（2）使用创建和销毁类的静态函数。

（1）使用接口类

接口类也就是创建第二个类，它作为要导出类的接口，所以在导出类改变时，也不需要重新编译客户程序，因为接口类没有发生变化。

假设导出的 CMyClass 类有两个函数 Function 和 AFunctionB。现在创建一个接口类 CMyInterface，下面就是在 DLL 中的 CMyInterface 类的头文件的代码：

```
# include "MyClass.h"
class _declspec(dllexport) CMyInterface
{
CMyClass *pmyclass;
CMyInterface();
~CMyInterface();
public:
int FunctionA(int);
int FunctionB(int);
};
```

而在客户程序中的头文件稍不同，不需要 INCLUDE 语句，因为客户程序没有它的拷贝。相反，使用一个 CMyClass 的向前声明，即使没有头文件也能编译：

```
class _declspec(dllimport) CMyInterface
{
class CMyClass;//向前声明
CMyClass *pmyclass;
CMyInterface();
~CMyInterface();
public:
int FunctionA(int);
int FunctionB(int);
};
```

在 DLL 中的 CMyInterface 的实现如下：

```
CMyInterface::CMyInterface()
{
pmyclass = new CMyClass();
}
CMyInterface::~CMyInterface()
{
delete pmyclass;
}
int CMyInterface::FunctionA()
{
return pmyclass->FunctionA();
```

```
}
int CMyInterface::FunctionB()
{
return pmyclass->FunctionB();
}
.....
```

对导出类 CMyClass 的每个成员函数，CMyInterface 类都提供自己的对应的函数。客户程序与 CMyClass 没有联系，这样任意改 CMyClass 也不会有问题，因为 CMyInterface 类的大小没有发生变化。即使为了能访问 CMyClass 中的新增变量而给 CMyInterface 类加了函数也不会有问题的。

但是这种方法也存在明显的问题，对导出类的每个函数和成员变量都要对应实现，有的时候这个接口类会很庞大。同时增加了客户程序调用所需要的时间。增加了程序的开销。

（2）使用静态函数

还可以使用静态函数来创建和销毁类对象。创建一个导出类的时候，增加两个静态的公有函数 CreateMe()/DestroyMe()，头文件如下：

```
class _declspec(dllexport) CMyClass
{
CMyClass();
~CMyClass();
public:
static CMyClass *CreateMe();
static void DestroyMe(CMyClass *ptr);
};
```

实现函数就是：

```
CMyClass * CMyClass:: CreateMe ()
{
return new CMyClass;
}
void CMyClass::DestroyMe(CMyClass *ptr)
{
delete ptr;
}
```

然后像其他类一样导出 CMyClass 类，这个时候在客户程序中使用这个类的方法稍有不同了。如若想创建一个 CMyClass 对象，就应该是：

```
CMyClass x;
CMyClass *ptr = CMyClass::CreateMe();
```

在使用完后删除：

```
CMyClass::DestroyMe(ptr);
```

4.1.4 小结

本节介绍了动态链接库的概念、动态链接库的创建和动态链接库的链接以及一些扩展应用，DLL 的设计方法为应用程序提供一定的可扩展性，但未成为一个规范。为了解决这个问题，微软提出了 COM 构件的设计方法，我们将在 4.3 节进行介绍。

4.2 类/对象层次的动态演化

4.2.1 JAVA 的动态性

Java 语言是一种具有动态性的解释型编程语言，当指定程序运行的时候， Java 虚拟机就将编译生成的.class 文件按照需求和一定的规则加载进内存，并组织成为一个完整的 Java 应用程序。Java 语言把每个单独的类 Class 和接口 Implements 编译成单独的一个.class 文件，这些文件对于 Java 运行环境来说就是一个个可以动态加载的单元。正是因为 Java 的这种特性，我们可以在不重新编译其他代码的情况下，只编译需要修改的单元，并把修改文件编译后的.class 文件放到 Java 的路径当中，等到下次该 Java 虚拟机器重新激活时，这个逻辑上的 Java 应用程序就会因为加载了新修改的.class 文件，自己的功能也做了更新，这就是 Java 的动态性。

4.2.2 隐式加载和显式加载

Java 的加载方式分为隐式加载（implicit）和显示加载（explicit），上面的例子中就是用的隐式加载的方式。所谓隐式加载就是我们在程序中用 new 关键字来定义一个实例变量，JRE 在执行到 new 关键字的时候就会把对应的实例类加载进入内存。隐式加载的方法很常见，用的也很多，JRE 系统在后台自动的帮助用户加载，减少了用户的工作量，也增加了系统的安全性和程序的可读性。

相对于隐式加载的就是我们不经常用到的显式加载。所谓显式加载就是有程序员自己写程序把需要的类加载到内存当中，下面我们看一段程序：

```
class TestClass{
public void method(){
       System.out.println("TestClass-method");
}
}
public class CLTest {
public static void main(String args[]) {
       try{
             Class c = Class.forName("TestClass");
             TestClass object = (TestClass)c.newInstance();
             object.method();
```

```
        }catch(Exception e){
              e.printStackTrace();
        }
    }
    }
```

我们通过 Class 类的 forName（String s）方法把自定义类 TestClass 加载进来，并通过 newInstance()方法把实例初始化。

Class 的 forName()方法还有另外一种形式：Class forName（String s, boolean flag, ClassLoader classloader），s 表示需要加载类的名称，flag 表示在调用该函数加载类的时候是否初始化静态区，classloader 表示加载该类所需的加载器。

forName（String s）是默认通过 ClassLoader.getCallerClassLoader() 调用类加载器的，但是该方法是私有方法，我们无法调用，如果我们想使用 Class forName（String s, boolean flag, ClassLoader classloader）来加载类的话，就必须要指定类加载器，可以通过如下的方式来实现：

```
Test test = new Test();//Test 类为自定义的一个测试类;
ClassLoader cl = test. getClass().getClassLoader();
                        // 获取 test 的类装载器;
Class c = Class.forName("TestClass", true, cl);
```

因为一个类要加载就必须要有加载器，这里我们是通过获取加载 Test 类的加载器 cl 当作加载 TestClass 的类加载器来实现加载的。

4.2.3　自定义类加载机制

在 java.lang 包里有个 ClassLoader 类，ClassLoader 的基本目标是对类的请求提供服务。当 JVM 需要使用类时，它根据名称向 ClassLoader 请求这个类，然后 ClassLoader 试图返回一个表示这个类的 Class 对象。通过覆盖对应于这个过程不同阶段的方法，可以创建定制的 ClassLoader。其中有个 loadClass（String name, boolean resolve）方法，该方法为 ClassLoader 的入口点，在 jdk1.2 以后，loadClass 方法将缺省调用 findClass 方法，详细内容可以参考 API 文档，我们编写的 ClassLoader 主要就是为了覆盖以上两个方法。回到我们刚才的问题，怎样读进字节码文件，并把它构成一个类对象呢？在 ClassLoader 里有个方法，Class defineClass（String name, byte[] b, int off, int len），答案就在这里了，我们可以把 class 字节码文件（如 Hello.class）读进一个字节数组里，byte[] b,并把它转化为 Class 对象，而这些数据可以来源于文件、网络等。

defineClass 管理 JVM 的许多复杂和依赖于实现的方面，它把字节码分析成运行时数据结构、校验有效性等等。我们无需亲自编写它，事实上，即使您想要这么做也不能覆盖它，因为它已被标记成最终的。

还有其他一些方法。

（1）findSystemClass 方法：从本地文件系统装入文件。它在本地文件系统中寻找类文件，如果存在，就使用 defineClass 将原始字节转换成 Class 对象，以将该文件转换成类。

（2）findClass 方法：jdk1.2 以后 loadClass 的缺省实现调用这个新方法。findClass 的用途包含您的 ClassLoader 的所有特殊代码，而无需要复制其他代码（例如，当专门的方法失败时，调用系统 ClassLoader）。

（3）getSystemClassLoader：如果覆盖 findClass 或 loadClass，getSystemClassLoader 使您能以实际 ClassLoader 对象来访问系统 ClassLoader（而不是固定地从 findSystemClass 调用它）。

（4）getParent：为了将类请求委托给父类 ClassLoader，这个新方法允许 ClassLoader 获取它的父类 ClassLoader。当使用特殊方法，定制的 ClassLoader 不能找到类时，可以使用这种方法。

（5）resolveClass：可以不完全地（不带解析）装入类，也可以完全地（带解析）装入类。当编写我们自己的 loadClass 时，可以调用 resolveClass，这取决于 loadClass 的 resolve 参数的值。

（6）findLoadedClass：充当一个缓存，当请求 loadClass 装入类时，它调用该方法来查看 ClassLoader 是否已装入这个类，这样可以避免重新装入已存在类所造成的麻烦。应首先调用该方法。

工作流程。

（1）调用 findLoadedClass（String）来查看是否存在已装入的类，如果没有，那么采用那种特殊的神奇方式来获取原始字节。

（2）通过父类 ClassLoader 调用 loadClass 方法，如果父类 ClassLoader 是 null，那么按缺省方式装入类，即系统 ClassLoader。

（3）调用 findClass（String）去查找类并获取类。

（4）如果 loadClass 的 resolve 参数的值为 true，那么调用 resolveClass 解析 Class 对象。

（5）如果还没有类，返回 ClassNotFoundException。

（6）否则，将类返回给调用程序。

之前我们都是调用系统的类加载器来实现加载的，其实我们是可以自己定义类加载器的。利用 Java 提供的 java.net.URLClassLoader 类就可以实现。下面我们看一段范例：

```
try{
        URL url = new URL("file:/d:/test/lib/");
        URLClassLoader urlCL = new URLClassLoader(new URL[]{url});
        Class c = urlCL.loadClass("TestClassA");
        TestClassA object = (TestClassA)c.newInstance();
        object.method();
   }catch(Exception e){
        e.printStackTrace();
   }
```

我们通过自定义的类加载器实现了 TestClassA 类的加载并调用 method()方法。分析一下这个程序：首先定义 URL 指定类加载器从何处加载类， URL 可以指向网际网络上的任何位置，也可以指向我们计算机里的文件系统（包含 JAR 文件）。上述范例当中我们

从 file:/d:/test/lib/ 处寻找类，然后定义 URLClassLoader 来加载所需的类，最后即可使用该实例了。

4.2.4　类加载器的阶层体系

讨论了这么多以后，接下来我们仔细研究一下 Java 的类加载器的工作原理。

当执行 java ***.class 的时候，java.exe 会帮助我们找到 JRE，接着找到位于 JRE 内部的 jvm.dll，这才是真正的 Java 虚拟机器，最后加载动态库，激活 Java 虚拟机器。虚拟机器激活以后，会先做一些初始化的动作，比如说读取系统参数等。一旦初始化动作完成之后，就会产生第一个类加载器－－ Bootstrap Loader，Bootstrap Loader 是由 C++ 所撰写而成，这个 Bootstrap Loader 所做的初始工作中，除了一些基本的初始化动作之外，最重要的就是加载 Launcher.java 之中的 ExtClassLoader，并设定其 Parent 为 null，代表其父加载器为 BootstrapLoader。然后 Bootstrap Loader 再要求加载 Launcher.java 之中的 AppClassLoader ，并设定其 Parent 为之前产生的 ExtClassLoader 实体。这两个加载器都是以静态类的形式存在的。这里要请大家注意的是，Launcher$ExtClassLoader.class 与 Launcher$AppClassLoader.class 都是由 Bootstrap Loader 所加载，所以 Parent 和由哪个类加载器加载没有关系。

4.2.5　类的动态替换

1. 代理模式

代理模式是一种对象的结构模式，简单地说就是给某个对象提供一个代理对象，并通过代理对象来访问真正的对象。

代理模式的要点就是不直接访问要访问的对象，而是通过代理对象，因此就可以在调用实际对象前或调用后利用消息传递做一些额外的工作。

2. 反射机制与动态编译

java 的反射机制是 java 被视为动态语言的重要特性，主要是通过 java.lang.reflect 包中提供的工具对于给定名称的 class,filed,method 进行访问。这种动态性给程序提供了很多的灵活性，后面要介绍的功能就得益于 java 的这一机制。

动态编译算是 java 反射的加强补充，在 j2se 1.6 以前的版本里边是通过 tools.jar 中的 com.sun.tools.javac 包来提供的，在当前已经发布的 j2se1.6 beta2 中已经将动态编译作为 j2se 的一部分了。

3. Java 的内建代理

J2se 在 1.3 以后提供了 Proxy、InvocationHandler 来支持代理模式。

对于 Java 内建的代理来说，就是 client 不直接访问对象，而是通过 Proxy 对象来访问，而在访问实际对象前后我们可以自己定义一些其他的操作。

具体来讲，client 访问的对象通过 Proxy.newProxyInstance（）给出，client 要访问的实

际的类可以通过 Proxy.getProxyClass 获得，而实际的调用将访问到我们实现的 InvocationHandler 对象的 invoke()方法中来。在 invoke()方法中我们可以为实际的调用添加额外的代码以满足不同的需要。

在后边讲到的具体实现中就可以看到，我们正是在实现 InvocationHandler 的 MyInvocationHandler 的 invoke()方法中来判断 Java 文件的改变，对于改变动态的编译、装载和调用来达到我们预期的目标的，Java 内建的代理模式可谓居功至伟。

4. 替换实例

为了演示运行时的类替换，让我们来定义一种服务。

```
public interface Postman {
   void deliverMessage(String msg);
}
```

具体的实现中，候选的有两种方案：

第一种是将输出字符串到控制台：

```
public class PostmanImpl implements Postman {
   private PrintStream output;
   public PostmanImpl() {
      output = System.out;
   }
   public void deliverMessage(String msg) {
      output.println("[Postman] " + msg);
      output.flush();
   }
}
```

第二种是输出字符串到文本：

```
public class PostmanImpl implements Postman {
   private PrintStream output;
   public PostmanImpl() throws IOException {
     output = new PrintStream(new FileOutputStream("msg.txt"));
   }
   public void deliverMessage(String msg) {
      output.println("[Postman] " + msg);
      output.flush();
   }
}
```

在程序运行中，我们就是要通过动态修改 PostmanImpl 来观察这个类的动态替换的过程。

这里要给出一个访问服务的 main 程序，以便看到这种方式带来的好处。

```
public class PostmanApp {
   public static void main(String[] args) throws Exception {
```

```
        BufferedReader sysin = new BufferedReader(new
InputStreamReader(System.in));
        Postman postman = getPostman();
        while (true) {
            System.out.print("Enter a message: ");
            String msg = sysin.readLine();
            postman.deliverMessage(msg);
        }
    }
    private static Postman getPostman() {
        DynaCode dynacode = new DynaCode();
        dynacode.addSourceDir(new File("dynacode"));
        return (Postman) dynacode.newProxyInstance(Postman.class,
               "sample.PostmanImpl");
    }
}
```

我们可以看到获取 PostMan 对象，只是在初始化的过程中做过一次，后边只是访问其 deliverMessage()方法，而 sample.PostmanImpl 这一实现的动态改变完全不可见。Java 文件改变后的编译、重新载入、对象实例化和方法的调用过程用户感觉不到。

根据上面提到的 Java 的内建代理模式，要实现一个 InvocationHandler，它完成了动态的类替换这么一个行为，下面这段代码展示了 MyInvocationHandler 的主要部分：

```
private class MyInvocationHandler implements InvocationHandler {
        String backendClassName;//实际的类名
        Object backend;//实际的类对象
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            /**
              * 在第一载入的时候通过 loadClass 来载入所需要的服务实现对象，后边每次检
查 java 文件是否被修改过，
              * 如果被修改就 unload 当前的对象，编译并重新载入该类。
              */
            Class clz = loadClass(backendClassName);
            if (backend.getClass() != clz) {
                backend = newDynaCodeInstance(clz);
            }
            // 调用有效的对象的方法
            return method.invoke(backend, args);
        }
    }
```

在 MyInvocationHandler 中保存了实现接口的类名和该类的一个对象，该对象也就是通过 newDynaCodeInstance（）方法得到的对象。在每一次调用该对象的方法的时候，java 的 Proxy 机制保证了系统会自动调用 MyInvocationHandler 的 invoke 方法。

这里采用反射进行动态载入的程序，调用的实际工作是在 InvocationHandler 的 invoke 方法中做的，因此 InvocationHandler 要保存实际的对象。

代理模式的好处是从使用者看来如同调用实际的对象是一样的，而实际上通过代理对象，程序可以动态采用不同的接口实现来完成工作，这一过程只需要在 Proxy.newProxyInstance()中给定不同的实现类即可。

4.2.6 小结

本节给出了关于类级别的动态替换方法，关于类的动态替换方法还有一些，这里只对一种方法进行例证，让读者了解动态的类替换在软件系统的动态性上的体现。

4.3 构件层次的动态演化

4.3.1 构件和基于构件的软件工程

随着对软件工程的研究不断深入，软件开发技术的不断提高，应用程序开发的模式也随之发生了变化。现在，新的程序设计和编写的方式使得开发人员可以通过少量的代码编写，把已有的功能模块进行重用和组合来实现应用的需要，也就是说只需要“组装”已有的功能模块就可以完成应用程序的开发工作了，这“已有的功能模块”就是我们通常所说的“构件”。软件开发变得类似于搭积木，只需要把现有的、前人设计好的构件按我们的需求进行拼装即可完成。

构件是软件系统内可标志的、符合某种标准要求的组成成分，类似于传统工业中的零部件。广义上讲，构件可以是需求分析、设计、代码、测试用例、文档或软件开发过程中的其他产品。狭义来说，一般指对外提供一组规约化接口的、符合一定标准的、可替换的软件系统的程序模块。通常情况下是指后者。

软件构件可分为可复用构件和不可复用构件。一个应用软件系统通常有三类组成成分。

（1）通用的基本构件，如：数据结构、用户界面元素等等。

（2）领域共性构件，指仅在该应用软件所属领域内经常出现的构件。

（3）应用专有构件，指仅在该应用软件中出现的特定构件。

可复用构件指前两者，通常情况下软件构件指可复用构件。

软件构件有两个特征。

（1）有用性。指构件完成的功能是有用的，也就是其功能可以出现在很多应用软件中。

（2）易用性。指构件要有很好的包装，能很方便地使用它。一般来讲，构件的包装要符合一定的标准。

软件构件是软件复用的基本单元，软件构件技术使得软件人员在应用开发时可以使用

其他人的劳动成果，为软件产业进行大规模专业化分工与合作形成了前提。为了更好使用构件技术、更大程度地达到软件复用，基于构件的软件工程 CBSE（Component-Based Software Engineer）得到了关注和发展。CBSE 集软件复用、分布式对象计算、企业级应用开发等技术为一体。CBSE 以软件体系结构为组装蓝图，以可复用软件构件为组装模块，支持组装式软件的复用，大大提高了软件生产效率和软件质量。为此国内外对于这一技术的研究正在不断深入，同时大型的软件公司，例如 Sun、Microsoft 以及软件组织机构，例如 OMG 等，都推出了支持构件技术的软件平台。

4.3.2　当前主要的构件标准规范

1．OMG 的 CORBA

1989 年，由多个在业界享有盛誉的信息产品供应商、软硬件厂商以及最终用户联合成立了一个非盈利性联盟— 对象管理组织 OMG（Object Management Group），该组织的目标是促进分布式系统开发中面向对象技术理论与实践的发展。

为了能开发出具有标准面向对象接口的、可互操作的、可重用的、可移植的软件构件，OMG 组织在基于众多开放系统平台厂商提交的分布对象互操作内容的基础上，于 1991 年 10 月推出了 CORBA 1.0，制定了公共对象请求代理体系规范 CORBA（Common Object Request Broker Architecture），定义了分布式应用程序开发的可重用的复用框架标准。CORBA 现在已经发展到了 3.0 版本。

CORBA 规范是绝大多数分布计算平台厂商所支持和遵循的系统规范技术，具有模型完整、先进，独立于系统平台和开发语言，被支持程度广泛的特点，已逐渐成为分布计算技术的标准。COBRA 标准主要分为 3 个层次[OMG02]:对象请求代理（ORB）、公共对象服务和公共设施。最底层是对象请求代理 ORB，规定了分布对象的定义（接口）和语言映射，实现对象间的通讯和互操作，是分布对象系统中的“软总线”。在 ORB 之上定义了很多公共服务，可以提供诸如并发服务、名字服务、事务（交易）服务、安全服务等各种各样的服务。最上层的公共设施则定义了构件框架，提供可直接为业务对象使用的服务，规定业务对象有效协作所需的协定规则。目前，CORBA 兼容的分布计算产品层出不穷，如 IBM Component Broker、Borland VisiBroker 和 Obacus 等。

为保持 CORBA 的商业中立性和语言独立性，在 CORBA 中使用了接口描述语言（IDL）编写对象接口，使得所有 CORBA 对象以同一种方式被描述，然后由编译器把 IDL 编译成由本地语言（C/C++, CORBA 或 Java 等等)写的 Stub 和 Skeleton. IDL 不同于其他的面向对象程序设计语言，只能用它定义类或方法，不能用它具体实现所定义的类或方法。CORBA 以这种接口定义语言的形式实现了对象内部细节的完整封装，从而降低了软件系统的复杂程度，增加了软件功能的可重用性。又因为语言中立，从而使得软件开发人员可以在较大范围内共享已有成果。

ORB 是一个 CORBA 产品必须提供的、遵从 CORBA 规范的应用程序，没有 ORB 程序，CORBA 应用程序是无法工作的。ORB 最显著的功能就是对应用程序或是其他 ORB 的请求予以响应，实现底层细节对软件开发者的透明性。在 CORBA 应用程序运行期间，

ORB 可能被请求做许多不同的事情，包括查找并调用远程计算机上的对象、负责不同编程语言之间的参数转换（如 C++到 Java），用动态方法调用激活远程对象等等。

为了提出一个请求，客户端可以使用动态调用接口（Dynamic Invocation）或者客户端的 Stub 程序，甚至直接和 ORB 交互。对象实现（Object Im plementation）通过 IDL 静态骨架（IDL Static Skeleton）或动态骨架（Dynamic Skeleton）的调用来接受请求。

CORBA 的种种特点推动了分布式多层软件体系结构的发展。目前，CORBA 技术在银行、电信、保险、电力和电子商务领域都有广泛的应用。

2. Sun 的 J2EE

为了推动基于 Java 的服务器端应用开发，Sun 公司在 1999 年底推出了 Java2 技术及相关的 J2EE 规范，J2EE 的目标是提供平台无关的、可移植的、支持并发访问和安全的、完全基于 Java 的开发服务器端构件的标准。

在 J2EE 中，Sun 给出了完整的基于 Java 语言开发面向企业分布式应用的规范。其中，在分布式互操作协议上，J2EE 同时支持 RMI 和 IIOP（Internet Inter-ORB Protocol）。而在服务器端分布式应用的构造形式，则包括了 Java Servlet, JSP（JavaServer Page）,EJB（Enterprise Java Bean）等多种形式，以支持不同的业务需求，而且 Java 应用程序具有“Write once,run anywhere”的特性，使得 J2EE 技术在分布计算领域得到了快速发展。

J2EE 不仅仅简化了可伸缩的、基于构件服务器端应用的复杂度，更重要的是依据 J2EE 规范，不同的厂家可以实现自己的符合 J2EE 规范的产品。J2EE 规范是众多厂家参与制定的，它不为 Sun 所独有，而且其支持跨平台的开发，目前许多大的分布计算平台厂商都公开支持与 J2EE 兼容技术。

EJB 是 Sun 推出的基于 Java 的服务器端构件规范 J2EE 的一部分，自从 J2EE 推出之后，得到了广泛的发展，已经成为应用服务器端的标准技术。Sun EJB 技术是在 Java Bean 本地构件基础上发展的面向服务器端分布应用构件技术。它基于 Java 语言，提供了基于 Java 二进制字节代码的重用方式。EJB 给出了系统的服务器端分布构件规范，这包括了构件、构件容器的接口规范以及构件打包、构件配置等的标准规范内容。EJB 技术的推出，使得 Java 用基于构件方法开发服务器端分布式应用成为可能。

从企业应用多层结构的角度看，EJB 是业务逻辑层的中间件技术，与 JavaBean 不同，它提供了事务处理的能力，自从三层结构提出以后，中间层，也就是业务逻辑层，是处理事务的核心。中间层从数据存储层分离，取代了存储层的大部分地位。

从分布式计算的角度看，EJB 像 CORBA 一样，提供了分布式技术的基础。提供了对象之间的通讯手段，从 Internet 技术应用的角度看，EJB 和 Servlet，JSP 一起成为新一代应用服务器的技术标准。EJB 中的 Bean 可以分为会话 Bean 和实体 Bean。前者维护会话。后者处理事务。Servlet 负责与客户端通信，访问 EJB，并把结果通过 JSP 产生页面传回客户端。

总的来说 ，J2EE 的特点在于跨平台性。目前，服务器市场的主流还是大型机和 UNIX 平台，这意味着以 Java 开发构件，能够做到“Write once, run anywhere”，开发的应用可以配置到包括 Windows 平台在内的任何服务器端环境中去。

3. Microsoft.Net Framework

.NET Framework 是 Microsoft 推出的一种新的分布式计算框架，它在 Microsoft 原来的分布式计算框架 DNA2000 之上作了重大改进，几乎是一个全新的平台。它简化了在高度分布式 Internet 环境中的应用程序开发。公共语言运行库（CLR）是.NET Framework 的基础。运行库是一个在程序执行时管理代码的代理，它提供核心服务，如管理内存、线程执行、代码执行、代码安全验证、编译以及其他系统服务等，而且还强制实施严格的类型安全以及其他的措施，以确保代码的安全性、可靠性和准确性。CLR 相当于 JAV 的虚拟机的概念，在 CLR 中运行的是由源代码编译而成的微软中语言（MSIL ）代码，相当于 JAVA 的字节码的概念。.Net 把在 CLR 中运行的代码称为托管代码，在 CLR 之外运行的代码称为非托管代码。

.NET Framework 类库是一个与公共语言运行库紧密集成的可重用的类型集合。该类库是面向对象的，托管代码可以利用类库提供的各种强大功能。这不但使.NET Framework 类型易于使用，而且还减少了学习.NET Framework 的新功能所需要的时间。此外，第三方构件可与.NET Framework 中的类无缝集成。数据库、XML、 Web 服务、Windows Form 和 Web Form 都是.Net Framework 类库提供的功能。

.Net Framework 还拥有语言独立的特性。.Net 定义了公共语言规范（CLS）.它是可以满足大多数应用程序所需的一套基本的语言功能。任何语言只要符合 CLS，就可以调用.Net Framework，就可以写出和其他对象完全交互的对象，而不管这些对象是以何种语言实现的。

最重要的是，.Net 是一个 Web 服务的平台，复杂的多层结构系统有能够大幅度提高生产率、相对紧密耦合的特点，而 Web 概念具有面向消息、松散耦合的特点。这二者有机地结合在一起，极大地拓展了应用程序的功能，并实现了软件的动态提供。我们将这种计算风格称为 Web 服务。.NET 正是根据这种 Web 服务原则而创建的，微软目前正着手提供这个基本结构，以便通过.NET 平台的每一部分来实现这种新型的 Web 服务。

4.3.3　构件的动态配置

动态配置的目的是能提供一个能稳定运行而且能不断演化的系统，保证系统的易维护性，最大限度地延长软件的生命周期，这里对构件层次的演化方法进行阐述。

1. 国外的动态配置研究现状

（1）用 ADL（Architecture Description Language，体系结构描述语言）或其他非 ADL 语言实现动态配置。如关注如何动态配置基于 Agent 和用 ADL 描述结构的系统，并实现一个动态重配置的算法。这个动态配置算法主要得益于 agent 执行模型的简单和灵活，算法本身并不需要去考虑状态一致性、多线程和可靠性。还有用非 ADL 的，如用 Common Lisp 实现一个 CORBA 兼容的对象请求代理，并且可以提供动态配置。Common Lisp 有一些特性可以很好地支持动态配置，如动态地修改类和函数定义、生成和最初类不一样的对象实例、多继承、原始闭包以及解释性语义等。

（2）设计了新的框架结构。有很多文献提出新的模型框架来实现动态配置。典型的方法实现了一个支持软件系统动态配置的结构——X-Adapt,X-Adapt 分两层：基础层（base-level）和元层（meta-level）。基础层包括应用程序对象，元层是参与系统动态配置

的主要部分。系统中的构件不是直接调用服务，而是通过一个代理（proxy）对象。代理对象是一个实现了一系列接口的对象。代理对象能重定向调用到实现了相应服务的对象上。有的研究方法描述了一个类似 X-Adapt 的通用结构——K-Component Model，在此结构下也可以进行动态配置。

（3）在 CORBA 和应用程序层之间添加中间层或扩展 CORBA 规范。由于 CORBA 技术越来越成为分布计算的主流技术和工业标准，而 CORBA 规范（2.X）本身缺少直接支持动态配置的特性，于是，在 CORBA 层和应用程序层之间添加能提供动态配置的中间层成了一种解决方案.

（4）其他方法。设计通用连接器（generic connector），再配合 CORBA 来实现动态配置。在动态配置的结构中，连接器也起了关键的作用。再如，起源于建筑工程设计的关于模式的研究，现在已经不断被应用到软件工程的各个方面，如开发组织、软件处理、项目配置管理等方面。其中设计模式和代码模式相对得到了较好的研究。

动态配置的研究大致可分为以上几种，其中第 1、2、4 种是针对通用平台的，第 3 种是针对 CORBA 平台的。

形式化方法也是实现动态配置的一种方法，CMICOO 把监督控制论（Supervisor Control Theory）的思想和技术运用到了分布式软件的设计中，并用 Petri-Net 为对象和对象的交互操作提出了形式化的模型，通过对此模型分析，提出动态配置的策略。

2. 国内对动态配置的研究分类

大致可分为以下几种：

（1）扩展或修改 CORBA 规范。在 CORBA 上增添了称为容错服务模块的系统服务，从而实现动态配置。用基于面向对象高阶 Petri 网的形式化方法描述系统的行为。静态描述仍使用 CORBA IDL（Interface Define Language，接口定义语言），动态描述则采用容错服务提供的支持。其优点是容错服务不改变 ORB 的体系结构和 IDL 的语法映射，应用程序有更高的可移植性。还有类似的方法，扩展和改造 CORBA 框架，容错和动态配置能力以 CORBA 服务的方式出现，并另外提供了多种系统服务以满足实时系统的需求。

（2）基于配置语言或体系结构描述语言。利用 XML 擅长于描述动态配置信息的特点，由 XML 解析器将一组 XML 标签和数据转化为一个 DOM（Document Object Model，文档对象模型）树，通过对树的遍历来动态地发现配置信息。因为 XML 是平台独立的和应用独立的，而且 JAVA 也是平台独立的，因此，用 JAVA 实现解释器，以 XML 表述的配置数据可在平台和应用之间实现真正的可移植性，基本能满足分布式软件动态配置的需求。但目前，这方面还处在初步阶段，他们实现的解释器还不太能适应大型分布式软件的复杂性。当然还有一些传统经典的体系结构语言，有的具有很好的描述动态性的能力，也可以实现动态配置。

（3）模型支持动态配置。大部分研究成果采用了以面向对象技术开发的，具有动态连接特性构件的方法。在一些成果中介绍了 IP（Intelligent Pad，智能型纸片）及其构件模型，并给出了 IP 实现的方法。IP 是一种实用化的软件重用方法，该方法实现了软件构件设计过程中的可视化以及构件分解、合成和重用的可视化。每个构件（PAD）有一个插头和若干插槽，通过插头和插槽的连接实现 PAD 间的互连。构件间的协议有读插槽消息、写插槽消息和更新消息，通过这三种消息使父子 PAD 协同。他们为构件的实现提供了基类，并以动态连接

库的形式给出。这种方法的优点是 PAD 构件之间及构件与基类之间都采用动态连接机制，有效地提高了计算机的内存使用率，还为构件的原型化开发提供支持，简化了构件的开发。

4.3.4　总结

国内、外对动态配置都有不少的研究，相对来说国内的研究较少些，较新的方法也不多见。上述的这些方法都各有优缺点，配置语言（CL）或体系结构描述语言（ADL）虽然对体系结构的动态性有良好的描述，但对程序员提出了较高的要求，除了要考虑应用本身的逻辑编程外，还要另外学习新的语言。再者动态配置的逻辑表示不直观，不易理解。用新的框架或模型实现动态配置，也会由于某些特殊的构件规格，结构标准，给开发者带来很大的困难和额外的负担。基于这些原因，动态配置应该给出一种面向图形的方法，以支持软件体系结构、动态配置和较高级程序设计。

4.4　动态软件体系结构

在本章的前面几节分别从函数层次、类层次以及构件层次对软件的动态演化进行了较详细的阐述。然而函数和类层次的演化往往需要软件维护人员关注软件的源代码，从而不利于软件从整体上进行演化管理。而基于构件层次的动态演化虽然隐去了很多细节，将软件开发人员从原来的细粒度转化为粗粒度演化，但由于构件与构件之间存在较强的耦合，当对某一构件进行替换或者删除时，必须要保证与之相关联的构件始终都能正常地调用它，这对运行中的构件是相当困难的，因而难以保证软件演化的一致性、正确性以及其他一些所期望的特性。体系结构将开发人员的关注点从一行行的代码转移到了粗粒度的构件和整个互连的结构上。这样使得设计者从难以理解的细粒度的软件编程细节中抽出身来关注一些“大图”：系统的结构、构件之间的交互，构件的分派（构件被分派到进程元素中去）等。软件体系结构的一个显著特征就是显性地构造连接件，连接在构件间起着桥梁的作用，它同时还管理构件之间的交互，从而将构件的计算功能从构件通信中分离出来，最大程度减小了构件之间的相互依赖关系，方便对系统的理解、分析和演化。

基于体系结构层次的动态演化方法，它最直接的优点主要有以下 4 个方面：（1）软件工程师使用软件体结构作为描述、理解和推导整个系统行为的工具，同时可以根据软件架构师的知识来管理动态演化。（2）如果在构件内部没有任何限制，就可以对现有的构件进行动态调整。（3）相关变更应用策略都封装在连接件中，由连接件来完成构件与构件之间拓扑结构的管理，这样可以很好地实现与特定构件行为的分离，从而方便对功能行为进行单独的变更。（4）变更应用策略的控制权被移交到系统架构师手中，系统架构师根据对应用需求和语义的理解作出变更决定。

4.4.1　体系结构概念

软件体系结构作为软件工程中的一个新兴研究领域，是随着描述大型、复杂系统结构

的需要和开发人员及计算机科学家在大型软件系统的研制过程中对软件系统理解的逐步深入而发展起来的，它与软件工程技术的发展和计算机发展的需要有着密切的关系。软件体系结构一般指的是系统的组织结构、它们之间的关联关系以及支配系统设计的原则和方针。一个系统的软件体系结构包括构成系统计算单元的构件、规范构件间交互行为的连接件以及构件和连接件如何结合在一起的配置组成。

构件（Component）是体系结构的基本要素之一。一般认为，构件是指具有一定功能、可明确辨识的软件单位，并且具备以下特点：语义完整、语法正确、有可重用价值。这就意味着，在结构上，构件是语义描述、通信接口和实现代码的复合体，是一个计算和数据存储单元。也就是说，构件是计算与状态存在的场所。更具体地，可以把构件视为用于实现某种计算逻辑的相关对象的集合，这些对象或是结构相关或是逻辑相关。在体系结构中，构件可以有不同的粒度。一个构件可以小到只有一个过程，也可以大到包含一个应用程序。它可以包括函数、例程、对象、二进制对象、类库、数据包等。

从抽象程度来看，尽管面向对象技术以类为封装单位达到了类级的代码重用，但这样的重用粒度还太小，仍不足以解决异构互操作和效率更高的重用。构件将抽象的程度提到一个更高的层次，它是对一组类的组合进行封装，并代表完成一个或多个功能的特定服务，也为用户提供了多个接口。构件之间是相对独立的。构件隐藏其具体实现，只通过接口提供服务。如果不用指定的接口与之通信，则外界不会对它的运行造成任何影响。因此，构件可以作为独立单元被使用于不同的体系结构、不同的软件系统中，实现构件的重用。因此构件的定制和规范化十分重要。构件的使用与它的开发也是独立的。

连接件（Connector）是用来建立构件间的交互以及支配这些交互规则的体系结构构造模块。构件之间的交互包括消息或信号量的传递，功能或方法调用，数据的传送和转换，构件之间的同步关系、依赖关系等。在最简单的情况下，构件之间可以直接完成交互，这时体系结构中的连接件就退化为直接连接。在更为复杂的情况下，构件间交互的处理和维持都需要连接件来实现。常见的连接件有管道（pipe）、通信协议或通信机制等。

体系结构级的通信需要有复杂协议来表达，为了抽象这些协议并使之能够重用，可以将连接件构造为类型。构造连接件类型可以将那些用通信协议定义的类型系统化，并独立于实现，或者作为内嵌的、基于它们的实现机制的枚举类型。为了保证体系结构中的构件连接以及它们之间的通信正确，连接件应该导出所期待的服务作为它的接口。为完成接口的有用分析、保证跨体系结构抽象层的细化一致性、强制互联与通信约束等，体系结构技术提供了连接件协议以及变换语法。为了确保执行计划的交互协议，建立起内部连接件依赖关系，强制用途边界，就必须说明连接件约束。

连接件的主要特性有可扩展性、互操作性、动态连接性和请求响应特性。连接件的可扩展性是允许连接件动态改变被关联构件的集合和交互关系的性质。互操作性指的是被连接的构件通过连接件对其他构件进行直接或间接操作的能力。动态连接性即对连接的动态约束，指连接件对于不同的所连接构件实施不同的动态处理方法的能力。请求响应特性包括响应的并发性、时序性。在并行和并发系统中，多个构件有可能并行或并发地提出交互请求，这就要求连接件能够正确协调这些交互请求之间的逻辑关系和时序关系。对于构件而言，连接件是构件的黏合剂，是构件交互的实现。连接件和构件的区别主要在于它们在体系结构中承担着不同的作用。

体系结构配置（Configuration）确定体系结构的构件与连接件的连接关系和拓扑要求。体系结构配置提供限制来确定构件是否正确连接、接口是否匹配、连接件构成的通信是否正确，并说明实现要求行为的组合语义。

4.4.2　演化与体系结构

软件演化旨在通过改变系统的内部结构、行为或者相关属性来实现系统的完善和升级。因此，无论是静态或者动态演化，都需要保证演化后系统的完整性和一致性。软件的结构性和演化性是现代软件的基本特性，根据现有的研究分析，具有良好的结构的软件更有利于软件的演化，所以，演化与体系结构之间有着十分密切的联系。软件体系结构影响演化的进行，而软件演化则可能导致软件体系结构的变更。

在软件系统运行期间，各个系统内部之间的通信都是在系统的底层完成，许多细节信息都被屏蔽在系统内部，从而给整个系统的演化带来很多的问题。在软件体系结构中，连接件作为一种显性实体进行描述，连接件封装了所有构件之间交互的信息，增强了系统的灵活性、分布性以及扩展性。系统在设计初期构件内部的耦合度较低，但随着系统进一步的详细设计，各个模块之间的耦合度就会增强，系统趋向越来越复杂。

不管系统设计初期是什么样子，如果在维护期间不考虑体系结构的影响，那么将会导致结构的退化。这里有一些解决该问题的方法：

（1）使用一个关注系统体系结构的维护过程；

（2）以保持系统结构和“干净”的体系结构的方式进行系统维护的方法来设计系统结构。

随着软件开发的规模日益增大，软件的复杂度也越来越高，因而，如何去重用已有的系统来构建一些大型的系统是解决软件危机的一个很好的方案。基于函数和类的重用粒度太小，不能从宏观层面来把握整个系统的结构和全局行为，同时也不适合用在一些大型软件系统开发中；基于构件的重用增大了软件的重用粒度，然而当构件被部署到特定系统中后，其通常需要针对特定的运行环境来做一定的假设，我们只有在完全了解这些假设之后才能去进行构件的组合，然而对于以“黑盒子”封装的构件而言是相当困难的工作。由于每一个构件都会对环境和其他构件行为做一些不同的假设，从而导致在构件组合时整个体系结构不匹配。目前最常用的解决这种不匹配的方法是封装构件（通过在这些构件之间插入连接代码），从使得构件分离，变换输入输出。

体系结构重用的方法就是产品线结构的概念。它为在以后系统重用已有系统的一部分提供了可能，但是还有很多的工作要做，也较难实现。

4.4.3　动态软件体系结构的描述

ADL 提供了一种形式化机制来描述软件体系结构，这种形式化机制主要通过提供语法和语义描述来模拟构件、连接件和配置。但是大多数 ADLs 只描述系统的静态结构，不支持对体系结构动态性的描述，目前常见的几种支持动态描述的语言主要有 Darwin，Rapide 语言和 Dynamic Wright 等。

动态 ADL 的思路主要是在 ADL 中专门设计用于预设演化的策略表示机制。例如

Darwin 语言运用 π 演算给出体系结构的语义，提供延迟实例化（Lazy instantiation）和直接动态实例化（Direct dynamic instantiation）两种技术支持动态体系结构建模，允许事先规划好的运行时的构件复制、删除和重新绑定；Rapide 基于偏序事件集（Partially Ordered Event Sets）对构件的计算行为和交互行为进行建模，允许在 where 语句中，通过 link 和 unlink 操作符重新建立结构关联；Dynamic Wright 使用标签事件（Tagged events）技术，提供了运用 CSP 对动态体系结构建模的方法。这些语言的动态管理机制中，都要求体系结构运行时的变化必须事先可知。

下面将介绍几种常见的体系结构描述语言：

1. Darwin 语言

Darwin 是由 Magee 和 Kramer 开发的体系结构描述语言。Darwin 通过接口来定义构件类型，接口包括提供服务接口和请求服务接口。系统配置则定义构件实例并在提供服务接口和请求服务接口之间建立绑定（如图 4-1 所示，一个名字为 filter 的构件实例）。Darwin 提供延迟实例化（Lazy instantiation）和直接动态实例化（Direct dynamic instantiation）两种技术支持动态演化。

```
Component filter{
        provide prev<port,int>;
        require next<port,int>;
  }
```

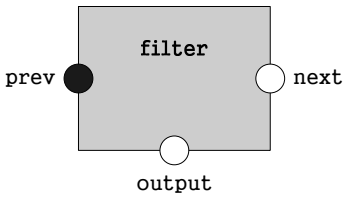


图 4-1 构件类型 filter

Darwin 运用 π 演算提供系统结构规约语义，构件所提供的服务作为一个名（Name），构件的绑定作为一个进程，它传输服务名到请求该服务的构件。π 演算是一种基本演算，用于描述和分析具有演化通信结构的并发系统。Darwin 中用到的是简单的一阶 π 演算。在 π 演算中，一个系统就是一个通过通道（channel）相互通信的独立进程的集合。通道或链接（link）都是通过名（name）来访问的。名在 π 演算中是最基本的实体，它们没有结构。Magee 和 Kramer 应用该语义模型分析实现 Darwin 配置的分布式算法。然而，它并未提供任何方法和技术描述构件或其相关服务的性质，Darwin 服务类型的语义由底层平台实现决定，没有相关语义解释。因此 Darwin 并不能提供体系结构行为分析的基础和方法。同时，Darwin 也未提供显式的连接件概念。它的提供/请求连接模型仅仅支持非对称的交互模式（Asymmetrical Model of Interaction），并且不能独立于构件进行描述。定义体系结构风格的时候，通常是定义它的交互模型，而把构件的定义留给体系结构师（Architect）。因此，Darwin 不能有效支持体系结构风格规约。此外，Darwin 运用参数化配置（(parameterized configurations）方法来描述一类系统的规约。

2. Dynamic Wright

Dynamic Wright 是由 Wright 语言扩展而来的动态体系结构描述语言，它主要完成对软件的动态模拟。通过使用 Wright 作为体系结构元素的描述语言，同时采用一些附加的特征来扩展它，这样就可以很容易地描述动态的环境。

如图 4-2 所示，一种 C-S 模式的软件系统正处于运行当中，我们需要在主服务器出现错误时，客户端能够自动连接到第二个备份的服务器上，这样，设计者就需要采用以下符号来表示这种机制。

上图并不是特别清楚地表达了整个体系结构演化的意思，同时也没有采用其他过多的文本来描述，从而可能导致设计者丢失了一些关键的信息。Dynamic Wright 引入了另外一些符号用来描述体系结构的变更。在 Dynamic Wright 语言中称为“configuror”，通过添加 configuror（C）来控制软件演化，如图 4-3 所示。Configuror 主要由以下几部分组成。

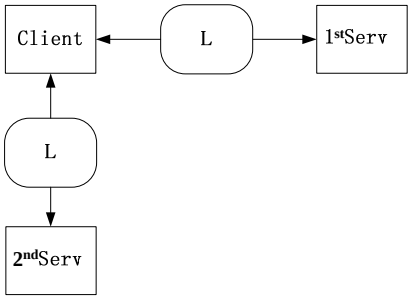


图 4-2　静态描述

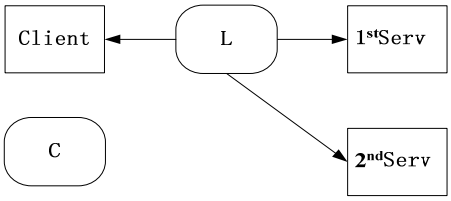


图 4-3　动态描述

（1）体系结构在何时进行重配置。

（2）什么东西能够引发体系结构对其自身进行重配置。

（3）怎样进行重配置。

由于 Dynamic Wright 语言是特意为动态软件体系结构量身定做的，因此它很适合这类动态体系结构设计环境。

3. C2ADL

C2 是一种基于构件和消息的体系结构风格，适用于大型频繁交互的层次型图形界面软件的体系结构描述。有许多专门的 ADL 用于描述 C2 风格，例如 C2SADEL（Software Architecture Description & Evolution Language）是用于描述 C2 风格的体系结构的 ADL，它为体系结构的演化提供了特别的支持。

在 C2 风格的体系结构中，连接件在构件之间转发信息，构件负责维护状态，进行操作，通过两个接口（顶端接口和底端接口）和其他构件交换消息。第一个接口包括一个可以发送的消息集合和一个可以接收的消息集合。构件之间的消息或者是请求构件执行一个操作，或者是一个通知（有关某个构件已经进行了一个操作或已经改变了状态）。

构件之间不能发送消息，必须通过连接件。构件的每一个接口最多只能连接到一个连接件上。一个连接件可以连接多个构件和连接件。请求消息只能在体系结构中自下而上地发送，而通知消息只能自上而下地发送。而且，构件之间的通信只能通过消息传递来实现，不允许使用共享内存方式通信。

体系结构的 3 个组成要素，构件、连接件和体系结构配置，都有一个演化过程。C2 对于构件演化的支持是通过类的子类型实现的。C2 的 ADL 提供体系结构规约的语法标记符号，没有相关语义规约，不支持体系结构风格的规约和分析。

4. ACME

ACME 是一种体系结构互换语言，支持从一种 ADL 向另一种 ADL 规格说明转换。

ACME 的核心概念以 7 种类型的实体为基础：构件、连接件、系统、端口、角色、表述和表述图。ACME 支持体系结构的分级描述。特别是每个构件或连接件都能用一个或多个更详细、更低层的描述来表示。ACME 的 7 种元素足以用来定义体系结构的层次结构，但是，体系结构还包含有许多其他附加信息，而且不同的 ADL 里增加的描述信息不尽相同。作为一种用于交流的 ADL，ACME 为了解决这一问题，使用属性列表来表示对于结构的说明信息。每一个属性由名字、可选类型和值构成。对 7 种设计实体里的任何一种都可以添加注释。从 ACME 的观点来看，属性是不被解释的值。仅当属性被开发工具用于分析、转换、操作等时，它才是有用的。

ACME 已经能够充分定义体系结构实例，并构成了 ACME 的语法分析工具的基础。但是，仍有许多其他方面需要完善。尤其是 ACME 并没有对体系结构的构成进行抽象。这会导致在描述复杂系统时，需要反复对一些公共结构做规格说明。为了处理这个问题，ACME 语言引入了模板机制。它是一种类型化、参数化的宏，用于对反复出现的模式做规格说明。在应用或初始化这些模式时，只要给它们提供适当类型的参数就可以了。模板定义了句法结构，可以被扩展到需要生成新的声明的位置。它们非常灵活，可以像在构件和连接件中一样定义它们的连接。可以对模板功能进行进一步的扩展并把它们归类到体系结构风格中。在 ACME 中，风格定义了相关的模板集合，提供了一种在体系结构设计中捕捉并复用公共结构的机制。

ACME 主要考虑的是体系结构的构造，因此并不包含体系结构的计算语义，而是依靠一个开放的语义框架。这个框架提供了基本的结构语义，它用构建属性的方式，允许特定的 ADL 把体系结构和运行时的行为结合起来。开放语义框架提供了从语言的结构外观对基于关系和约束的逻辑形式的直接映射。在这个框架里，ACME 规格说明表示一个由此推导出的逻辑谓词，称为它的指示规则（prescription）。这个谓词可以用于逻辑推理，或用于和规格说明所试图描述的现实世界中的事物做逼真度的比较。

5. Unicon

Unicon（UNIversal CONnection）是 Shaw 等开发的体系结构描述语言。Unicon 是一种围绕着构件和连接件这两个基本概念组织的体系结构描述语言。构件代表了软件系统中的计算和数据的处所，用于将计算和数据组织成多个部分。这些组成部分都有完善定义的语义和行为。连接件是代表构件间的交互作用的类。它们在构件之间的交互中起中介作用。

Unicon 支持单独存在的、对称和非对称的连接件，用来规定构件之间的交互机制。连接件运用系列角色（Role）定义交互的参与者，而构件接口是系列扮演者（Player）。角色、扮演者具有类型和属性，用来表明构件希望参与交互的性质和相关的交互细节。系统配置的时候，构件的扮演者与连接件的角色相互关联。运用 Unicon 提供的 Connection Expert 和配置工具能够构造运行的软件系统。

然而，Unicon 仅仅支持固定的交互类型和连接件，新型交互类型必须通过编程提供新的 Connection Expert 才能实现。Unicon 没有提供描述一类系统的机制和方法，提供的分析方法依赖于与连接件密切相关的分析工具并且同实现相关。因此，Unicon 在体系结构风格规约方面存在不足。

此外，Unicon 构件和连接件规约与其实现密切关联，体系结构模型与实现相互对应，

与模块互连规约并无本质区别，Medividovic 称其为实现约束语言（Implementation Constraining Language）。

6. Rapide

Rapide 是 Luchham 等开发的体系结构描述语言。Rapide 是一种基于事件（event-based）的、用于定义并执行系统体系结构模型的计算机语言，并包括与这个语言相关的仿真工具集。Rapide 基于偏序事件集（Partially Ordered Event Sets，称为 Posets）对构件的计算行为和交互行为进行建模。Rapide 与 Darwin 和 Unicon 不同，它独立于实现，运用通信事件序列描述构件的行为。Rapide 的通信事件分为两类:外部动作（Extern Action）和公共动作（Public Action）。Rapide 通过观察外部动作，并在它们和公共动作之间建立关联，从而描述构件和系统的计算行为。

Rapide 通过连接（Connection）机制建立构件之间的交互。连接机制在构件接口的外部动作和公共动作之间建立关联，规定构件之间事件发生的因果关系。作为连接的结果，当连接模式左边的事件启动时，它右边的事件随之发生。因此类似于 Darwin 的绑定，Rapide 只能描述构件之间非对称的交互关系。

Rapide 的连接机制嵌入配置定义当中，两者不能分离，Medividovic 把它称为嵌入配置 ADL（In-line Configuration ADL）。因此，Rapide 不允许单独对连接件进行描述和分析，并且没有提供相关机制，把多个连接机制捆绑成为一个整体，构成复杂的交互模式。因此 Rapide 描述构件交互模式的能力存在不足。

Rapide 通过体系结构仿真（Architectural Simulation）执行一致性检测和相关分析，生成系列偏序事件集，检测它们是否与接口、构件行为和连接机制的规约兼容。Rapide 模型的执行结果是一个事件集合，它包括执行过程中发生的事件，以及事件之间的临时和时序关系（casual and timing relation）。其中，作为仿真结果产生的临时历史（casual history）是 Rapide 在所有基于事件的语言中所独有的特点。带有临时历史的事件的集合被称为 poset。而仿真本质上仅仅意味着体系结构的测试，而不是严密的分析。

7. SADL

SADL 语言提供了对软件体系结构的精确的文本表示，同时保留了直观的框线图模型。SADL 语言明确区分了多种体系结构对象，如构件和连接件，并明确了它们的使用目的。SADL 语言不仅提供了定义体系结构的功能，而且能够定义对体系结构的特定类的约束。SADL 的一个独特方面是对体系结构层次体系（architecture hierarchy）的表示和推理。一个垂直层次体系（vertical hierarchy）的作用在于，它能实现体系结构抽象和传统程序设计语言中基本的结构概念之间差异的平稳过渡。通常用不同的语言来描述一个垂直层次体系的每一层，反映出在表示上的变化。SADL 模式支持结构求精（structural refinement），即把一个体系结构系统地转化成另一个包含不同体系结构概念的体系结构。

8. 其他 ADLs

MetaH 的目标是支持实时、容错、安全、多处理、嵌入式的软件系统的分析、验证和生产。MetaH 提供了集成的、可跟踪的体系结构规格说明、分析和实现。MetaH 语言不仅

有文本方式的语法描述，又能以图形方式描述。但 MetaH 缺乏严格的形式化基础，也不能描述动态软件体系结构。

欧盟的 ArchWare 项目旨在以体系结构模型为中心构造软件系统。ArchWare 提出了一个动态体系结构描述语言 π-ADL。π-ADL 是一个基于高阶 π 演算的形式化语言，支持动态体系结构建模、体系结构分析和约束检测。然而，π-ADL 太过抽象和形式化，难以理解和使用。

国内的一些学者也提出了几种比较有特色的体系结构描述语言，如基于框架和角色模型的软件体系结构规约 FRADL，多智能体系统体系结构描述语言 A-ADL，可视化体系结构描述语言 XYZ/ADL，基于主动连接件的体系结构描述语言 Tracer，用于构件组装的 ABC/ADL 等。

4.4.4 动态软件体系结构的实现

软件体系结构概念的产生试图在软件需求与软件设计之间架起一座桥梁，着重解决软件系统的结构和需求向实现平滑过渡的问题，软件体系结构已经作为一个明确的文档和中间产品存在于软件开发过程中。可见，软件体系结构技术的产生源于软件开发的需求，原是一个供软件开发人员使用的设计概念和技术。由于软件体系结构清晰地描述了构件及其相互关系和整个系统的框架，自然地，将软件开发中的软件体系结构应用于动态演化也成为一种考虑。

1. 动态变更类型

本书重点讲解 ArchStudio 原型系统如何实现软件体系结构的动态变更。包括如何实现构件的添加、删除、替换和重配置等。

（1）运行时构件的添加

构件的添加是通过对系统功能的扩展来支持完善演化。一些设计风格很方便构件的添加。例如，观察者设计模式将数据提供者和它的观察者分离开来，从而更方便在不影响系统其他部分的情况下添加新的观察者。在中介者设计方法中，新的中介者可能被引进来维护独立构件之间的关系。采用隐性调用机制的设计方法一般更容易在运行时期添加构件，因为构件调用与所调用的构件数量无关。

在添加构件的时候，需要确保系统处于它的初始状态。典型的就是运行时添加构件，必须能知道系统状态以及执行一些必要操作来将它的内部状态与系统状态保持一致。

体系结构变更说明一般都描述了必要的结构变化来融合这些新的构件。有些时候，体系结构配置变更可能内含在体系结构风格或者应用领域中，或者可以从构件的外部属性中获得。例如，Adobe Photoshop 插件构件导出一个插件类型属性，它的值是直接从一个固定的列表中取得。Photoshop 根据这些值来确定如何去同这些插件进行交互。

（2）运行时构件的删除

构件删除功能使得设计者可以直接删除一些不需要的功能行为，同时还可以通过构件删除操作来完成对原有行为的替换。在适当的情况下管理构件的删除是一种特定的应用。例如，当该构件的任何一个函数或方法在执行栈中的时候，运行环境就可能禁止对该构件

的删除。一些系统，特别是通过内部不可靠的连接进行通信的分布式系统，它们通常被特意设计成能够容忍功能和状态的突然丢失。和构件的添加一样，部分设计方法和风格可能更适合于运行时构件的删除。

（3）运行时构件替换

我们可以将构件的替换理解为构件删除和添加的一种特例。①正在执行的构件状态必须迁移到新的构件上去。②新构件和原有的构件在变更期间不能同时运行。修复演化和适应演化都拥有这种变更特征。

当构件没有状态或者所设计的系统能接受状态丢失的情况下，系统构件的替换就变得相当简单了。这些系统在进行状态恢复的时候通常会首先检测状态的丢失并选择一个低级别的操作模式。另一种方法，单一体系结构风格示例，它在实现工具中就把操作模式包含进去了。当构件不满足显性的执行和精确的需求时，该操作模式将拒绝构件更新。

在没有专门设计容忍状态丢失的系统中，构件的替换需要做一些附加的操作，这超出了构件添加和删除的讨论范畴。目前已经提出了一些方法，它们可以在动态演化中保存构件状态并防止信息丢失。Hofmeister 的方法需要每一个构件提供两个接口方法：一个用来收集状态信息，另一个用来在替换另一个构件的时候执行初始化操作。只有当新构件的显性接口是被替换构件的超类时，这些方法才适用。

（4）动态重配置

体系结构重配置支持重新绑定已经存在的功能来修改整个系统的行为。数据流体系结构，例如 UNIX’s 的管道－过滤器风格和 Weaves，通过对已存在的行为进行静态重配置来提供可靠的灵活性。例如，UNIX’s 的管道－过滤器风格可以通过重新组合已经存在的行为来构造出丰富的行为集。

动态重配置能够通过修改连接件的绑定来实现，因为连接件负责所有构件之间的通讯。和构件替换一样，如果要确保构件的可靠通讯，就必须防止信息的丢失。

2. 动态变更机制

我们所提出的支持基于体系结构层次的动态演化方法主要包括一些相互关联的机制。具体描述如下：

（1）显性的体系结构模型。为了有效的修改系统，我们在系统运行时提供了精确的体系结构模型。为了完成这些，我们采用一些系统的体系结构子集作为系统完整部分进行部署。部署的体系结构模型描述了构件和连接件的内部交互以及到实现模块的映射，映射可以将模型的变更应用到与之关联的实现模块中去。运行时体系结构设施主要维护模型和实现模块之间的关联。

（2）描述动态演化。对体系结构模型进行修改。一个修改策略采用一些特定的算法来新增、删除构件，替换连接件以及构件，更改体系结构拓扑结构等。

这种方法可以灵活地支持模型的演化，在该方法中可以由程序开发中的不同层次的人员提供修改，由最终用户根据特殊的需求，选择应用修改。通过应用不同的修改，一个最终用户能够有效地创建满足他们自身需要的系统。因此，变更策略在这些系统中应该能可靠地进行变更。在整个系统中，还应该提供一个方便查找体系结构模型并用查询结果来指导修改的系统模块来支持体系结构的变更。采用模型来通知和指导修改可以排除很多演化

系统中突发的困难。

（3）管理动态演化。该动态演化方法支持约束变化机制来保持系统的完整性。约束在管理动态演化过程中扮演了一个十分重要的角色，目前许多基于体系结构动态演化的方法都采用了约束机制。而且，管理动态演化的机制也限制了什么时候可能发生特定的变更。

在复杂的修改过程中，系统体系结构在达到最终有效状态之前可能需要历经许多无效的状态。尽管约束可能合法地限制某些修改的路径，但是如果只是基于中间无效状态，那么就可能阻碍一些有效的动态演化。因此，需要提供一个支持可操作的修改机制。

（4）可重用的运行时体系结构基础设施。运行时体系结构基础设施主要包括 3 个方面：①在应用修改策略的时候维护体系结构模型和具体实现的一致性。②将体系结构模型的变更具体化到应用程序中。③防止动态演化违反体系结构约束。因此，运行时体系结构基础设施能够支持不同的构件新增、删除和替换策略，同时还能被订制到特殊的应用领域。运行时体系结构基础设施采用体系结构模型到具体模块的映射关系以及基本环境的支持来实现变更。

3. 动态变更的实现

该部分主要介绍我们的原型系统 Archstudio，它实现了前面所描述的所有机制。该工具 Archstudio 是采用 JAVA 来实现的，它能够修改采用 JAVA-C2 类框架编写的 C2 风格的应用程序。JAVA-C2 类框架提供了一系列基于 C2 概念的可扩张的 JAVA 类，例如构件、连接件和消息。开发人员可以通过对框架类的扩展以及提供特定应用行为来创建构件和连接件。连接件在实现当中被看作分离的实体，同时通过一系列他们输出的功能来支持动态绑定。使用内部通信设施的连接件由框架来提供。

图 4-4 描述了一个 Archstudio 体系结构的概念视图。体系结构模型代表了一个应用程序的当前体系结构。我们当前的实现工具用一个抽象的数据类型（ADT）封装了体系结构模型。ADT 为应用程序的体系结构模型提供了查询和更改操作。

模型采用结构化的 ASCII 码保存，由运行时体系结构基础设施进行维护。模型包括了构件和连接件之间的相互连接，以及到 JAVA 类之间的映射。运行时修改主要包括了一系列对体系结构模型查询和变更请求，他们一般来自许多不同的源。

体系结构演化管理器（Architecture Evolution Manager，AEM）维护了体系结构模型（Architectural Model）到具体实现（Implementation）之间的映射关系。需要通过 AEM 来申请体系结构模型的修改，主要由它来确定模型是否有效。体系结构描述语言（ADL）和运行环境基础设施（Environment Infrastructure）将 AEM 从 ADL 和运行环境的变更中分离出来。AEM 利用体系结构约束机制或者扩展分析工具来判定一个变更是否可以被接受。当前 AEM 的实现工具利用隐性的 C2 风格规则的知识来对变更进行约束；计划在以后加入体系结构约束机制和使用扩展分析工具的能力。如果变更违反了 C2 风格规则，AEM 就会拒绝本次变更。否则，将对体系结构模型进行修改，同时根据映射来修改与模型对应的实际应用程序模块。

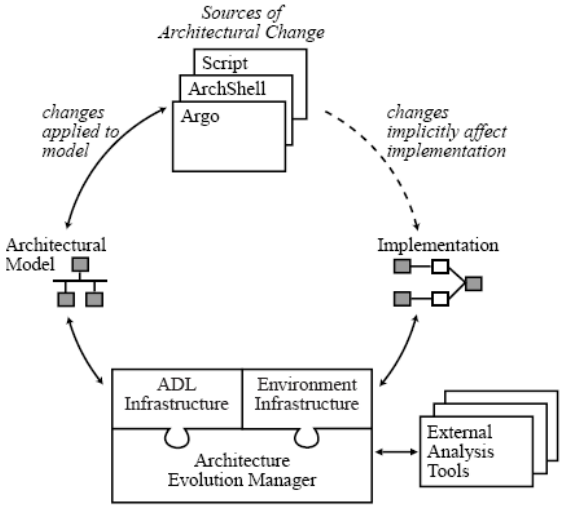


图 4-4　Archstudio 体系结构

Archstudio 目前主要有 3 个部分用来修改体系结构：Argo, ArchShell 和 Extension Wizard（扩展向导）。除了这些，它还包括一个工具，能够存取自己的体系结构模型并采用相同的机制对其进行操作。

Argo 为体系结构模型提供了图形绘制界面，这些体系结构模型可以直接进行操作。从调色板中选择新的构件和连接件，将它们拖曳到设计画布中从而完成构件和连接件的添加。可以选择删除构件和连接件，从而产生一个删除命令。通过直接操作构件和连接件之间的连线来完成配置。

ArchShell 和 Argo 的功能一样提供了一个文本的、命令驱动的接口用来具体化动态修改。主要提供了新增、删除构件和连接件、配置体系结构以及展示体系结构的文本表示命令。

ArchShell 提供了两个 Argo 中没有的命令，一个是能够让体系架构师通过与构件之间发送信息一样的方式任意发送消息到其他任何构件和连接件上，这样就方便调试以及勘测体系结构行为。

作为体系架构师的设计工具，Argo 和 ArchShell 方便快速的进行体系结构设计。在探索提出的动态体系结构演化的同时提供了十分有用的反馈信息。

Argo 和 ArchShell 是一种交互性工具，软件架构师用它来描述体系结构和体系结构修改。一旦一个体系架构师指定和证实了一个修改后，就可以用扩展向导（Extension Wizard）来为最终用户部署修改。扩展向导为制定动态修改提供了最终用户的简单接口，同时被部署为最终用户系统的一部分。扩展向导负责在最终用户安装系统时执行修改脚本。最终用户用一个 Web 浏览器来显示可下载的系统更新文件列表，例如，在应用程序销售商的 Web 站点上提供。一个系统的更新文件是一个压缩文件，它包含了一个运行时修改脚本和任何新的实现模型。选择一个系统更新使得 Web 浏览器去下载文件以及调用扩展向导来处理它。扩展向导对文件进行解压缩，找到它所包含的修改脚本然后执行它。Hall 等人所采用的方法与该方法类似，也是用来部署系统的更新。

4.5 动态工作流

4.5.1 工作流技术简介

1. 工作流技术的产生及发展

工作流（Workflow）作为一个概念最初是出现在图像处理和文档管理领域中。在图像处理和文档管理过程中，图像或文档按一定的顺序被发送到相应的地方进行编辑、核对和审查等活动。在这些业务过程中引入工作流就是用计算机软件系统来控制活动的执行顺序，将活动分配给合适的执行者，并且提供完成活动所需的应用程序和数据。其核心思想就是将组织的活动看作业务过程，并且用计算机软件系统来实现业务过程的自动化管理。所谓业务过程（Business Process），就是在一个具有特定功能角色和关系的组织结构环境中，一组为了实现某个目标而进行的具有一定执行顺序的多个活动。工作流就是业务过程在计算机系统中的表示。这种思想后来被应用到其他领域，从而出现了一批工作流软件。但是这些工作流软件都是分别针对某个具体的应用领域或应用环境，而且受当时的计算机软硬件技术的限制，因此工作流的应用依然很有限。进入 20 世纪 90 年代以来，随着计算机网络、数据库和分布式对象等相关技术的迅速发展和成熟，工作流管理技术也受到了广泛的关注。在全世界范围内，有许多研究团体都在进行工作流管理技术的研究，一些标准化的组织也积极的制定了工作流管理技术的规范，而数以百计的工作流软件产品的出现，则极大地推动了工作流管理技术在商业中的应用。如今，工作流管理技术已经成功地运用到电信、制造业、银行、医疗、保险、软件工程、政府机构等领域。根据 Delphi 组织的调查与研究，工作流市场自 1993 年来一直保持稳步增长，工作流正成为 IT 界一个新的技术与经济的增长点。

目前，作为支持企业经营过程重组（Business Process Reengineering，BPR）、经营过程自动化（Business Process Automation，BPA）的一种手段，工作流技术的研究应用日益受到学术界与企业界的重视。许多大学和研究机构也致力于工作流技术的进一步发展，开展了一系列研究项目，取得了显著的成果。

纵观工作流软件产品由 80 年代的萌芽到 90 年代的繁荣，我们可以把它总结为 3 个阶段。

第 1 阶段，主要是应用于某些特定领域的、相对独立的应用系统，比如图像、文档管理系统；

第 2 阶段，主要表现为具有底层的通信基础结构、能够实现任务协作的应用系统，比如具有消息传递功能的工作流系统；

第 3 段，具有图形用户界面的过程定义工具、用户定义与任务执行完全分离的工作流系统，其体系结构基本上符合工作流管理联盟所提出的标准结构。

2. 工作流的有关定义

十几年来，不同的研究者对工作流分别提出了不同的定义，到目前为止，对工作流仍

没有完全统一的定义。这些不同定义都从不同的角度对工作流进行了描述，可以使我们对工作流的一些基本特征有一定的理解。

（1）工作流管理联盟（Workflow Management Coalition, WFMC）的定义工作流是一类能够完全或者部分自动执行的经营过程。根据一系列过程规则、文档、信息或任务能够在不同的执行者之间传递、执行。

（2）Georgakopoulous 给出的工作流的定义工作流是将一组任务（task）组织起来完成某个经营过程。在工作流中定义了任务的触发顺序和触发条件。每个任务可以由一个或多个软件系统完成，也可以由一个人或一组人完成，还可以由一个人或者多个人与软件系统协作完成。任务的触发顺序和触发条件用来完成并实现任务的触发、任务的同步信息流（数据流）的传递。

（3）IBM Almaden 研究中心给出的工作流定义工作流是经营过程中的一种计算机化的表示模型，定义了完成整个过程所需要的各种参数。这些参数包括对过程中每一个单独步骤的定义、步骤间的执行顺序、条件以及数据流的建立，每一步骤由谁负责以及每个活动所需要的应用程序。

（4）AmitSheth 的定义工作流是经营过程可运转的部分，包括任务的顺序以及由谁执行，支持任务的信息流、评价与控制任务的跟踪、报告机制。

以上都是对工作流的定义的非形式化语言的描述，虽然各有异同。但达成了一个共识：工作流是经营过程的一个计算机实现，可以把工作流简单地看成企业的一个具体经营过程流程的抽象或图示化表示。在实际情况中可以更广泛地把凡是由计算机软件系统（工作流管理系统）控制其执行的过程都称为工作流。

3. 工作流管理系统与实施

在明确了工作流基本概念的基础上，下面介绍工作流管理系统（Workflow Management System，WFMS）的定义。WFMC 给工作流管理系统的定义：工作流管理系统是一个软件系统，它完成工作流的定义和管理，并按照在计算机中预先定义好的工作流逻辑推进工作流实例的执行。

通常，工作流管理系统指运行在一个或多个称为工作流机的软件上的用于定义、实现和管理工作流运行的一套软件系统，它和工作流执行者（人、应用）交互，推进工作流实例的执行，并监控工作流的运行状态。在这里需要强调指出的是工作流管理系统不是企业的业务系统。在很大程度上，工作流管理系统为企业的业务系统运行提供一个软件支撑环境，非常类似于在单个计算机上的操作系统。只不过工作流管理系统支撑的范围比较大、环境比较复杂而已，所以也有人称工作流管理系统是业务操作系统（Business Operating System，BOS）。在工作流管理系统的支撑下，通过集成具体的业务应用软件和操作人员的界面操作，才能够良好地完成对企业经营过程运行的支持。所以，工作流管理系统在一个企业或部门的经营过程中的应用是一个业务应用软件系统的集成与实施过程。

工作流管理系统可以用来定义与执行不同覆盖范围（单个工作者、部门、全企业、企业间）、不同时间跨度（分钟、小时、天、月）的经营过程。这完全取决于实际应用背景的需求。按照经营过程以及组成活动的复杂程度的不同，工作流管理系统可以采取许多种实施方式。在不同的实施方式中，所应用的信息技术、通信技术和支撑系统结构会有很大的差别。

工作流管理系统的实际运行环境可以是在一个工作组内部或者在全企业的所有业务部门。

虽然不同的工作流管理系统具有不同的应用范围和不同的实施方式，它们还是具有许多共同的特征。从比较高的层次上抽象地考察工作流管理系统，可以发现所有的工作流管理系统都提供了三种功能:

（1）建立阶段功能。主要是考虑工作流过程和相关活动的定义和建模功能。在这个阶段，通过使用一个或多个分析、建模和系统定义工具，把实际中的业务过程转化成形式的、计算机可以处理的定义。

（2）运行阶段的控制功能。在一定的环境下，执行工作流过程并完成每个过程活动的排序和调度功能。在运行时期，过程定义由负责创建、控制过程实例的软件所解释，这个软件负责安排各个活动的执行时间。核心构件是基本工作流管理控制软件 Engine（工作流机），它负责过程的创建与删除，控制运行过程中活动的执行时间安排，以及与人、应用工具资源进行交互。

（3）运行阶段的人机交互功能。实现各种活动执行过程中用户与应用工具之间的交互。在活动间转移控制、确定过程的运行状态、调用应用工具、传递适当的数据等，都必须与过程控制软件进行交互。

图 4-5 给出了工作流管理系统三个主要功能之间的关系。

同工作流管理系统的 3 个功能相对应，工作流管理系统在实际系统中的应用一般分为 3 个阶段，即模型建立阶段、模型实例化阶段和模型执行阶段。这 3 个阶段也即工作流系统实施的 3 个阶段。如图 4-6 所示。在模型建立阶段，通过利用工作流建模工具，完成企业经营过程模型的建立，将企业的实际经营过程转化为计算机可处理的工作流模型。模型实例化阶段，给每个过程设定运行所需的参数，并为每个活动分配所需的资源（包括设备、人员、应用）。模型执行阶段，完成经营过程的执行，在这个过程中重要的任务是完成人机交互和应用的执行，并对过程与活动的执行情况进行监控与跟踪。

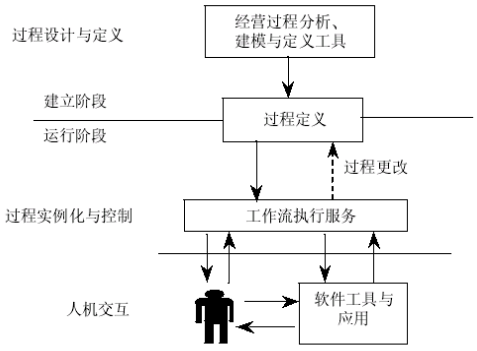


图 4-5 工作流管理系统的特性

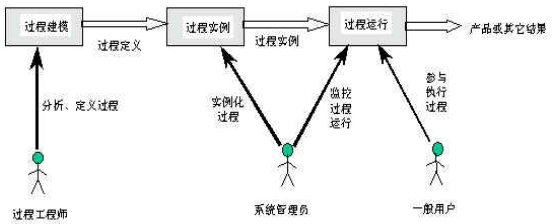


图 4-6 工作流管理系统实施的三个阶段

值得关注的是，工作流管理联盟在工作流管理系统的规范和基础方面做了大量的工作，它提出了工作流管理系统的一些规范，定义了工作流管理系统的结构及其应用、管理工具和其他工作流管理系统之间的应用编程接口。有关文档可参看工作流管理联盟网站 http://www.wfmc.org。

4.5.2　动态工作流概述

随着市场竞争的日益激烈，企业的业务过程不再是一成不变，它需要根据市场的需求不断地做出动态的调整和修改。随着企业环境的持续变化，不确定性和多变性已经成为现代企业流程的内在特点，因此提高工作流管理系统应对流程变化的能力，已经成为工作流领域研究的热点问题之一。

以往工作流管理系统只允许在建模时对过程进行编辑和修改，一旦过程运行之后就不能再对其进行修改。这种限制使得它们仅仅适用于那些结构合理、运行时现实的业务过程与建模时计算机表示的过程完全吻合以及不需要动态扩展的业务过程。但是现实世界复杂多变，在业务过程实际执行过程中，其应用环境经常出现新的法律规则、新的市场需求、新的技术等特例情况。而在建立阶段时完全预知这些特例情况是不可能的，并且对建模人员来说也是不可接受的。此外，实际的业务过程中还可能存在不是静态的、难以预测的意外事件或者异常，这往往会导致实际的业务过程与预定义的工作流过程存在偏差，这种情况的经常发生极大地限制了工作流管理系统的应用范围。为了提高工作流管理系统的灵活性，拓展潜在的市场应用领域，应该在保证工作流基本特性的前提下，研究如何提高工作流系统的灵活性，增强适应各种环境变化的能力。动态工作流正是为此目的提出的。

“动态工作流”这一名词起源于“动态修改”。所谓动态修改是指在过程运行时对该工作流过程模型进行某种修改，它是相对于工作流建模时（此时过程并未运行）对过程进行的静态修改而言的。其实，目前国内外学术界对于动态工作流并没有给出一个大家公认和接受的明确定义。通俗地说，如果一个工作流管理系统支持对正在运行的工作流过程进行修改，我们称这个工作流管理系统为支持动态修改的工作流管理系统，简称动态工作流系统。而动态工作流技术指与动态修改相关的所有技术，包括动态修改分类、动态修改策略、动态修改实现方法、动态修改带来的问题及怎样解决这些问题等，简称动态工作流。

4.5.3　动态工作流的特征及分类

工作流管理系统是一个综合的大型信息系统，动态变化的内容可以体现在其体系结构上，如图 4-7 所示。

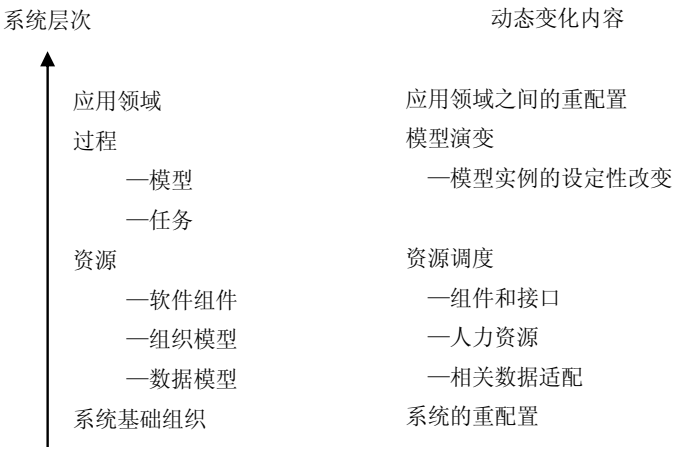


图 4-7　动态变化的层次结构图

从图 4-7 可以看出动态变化的内容随着工作流抽象层次的不同，其动态变化的含义和方式是不同的，从以下 3 个方面来讨论各种动态变化的特征。

1. 应用领域的动态变化

一个工作流管理系统不是独立存在的，它是商业应用系统的一个重要组成部分。一个实际运行的商业系统往往是面向某一特定的领域的，而在这个特定的领域，该工作流系统可以认为是一个“单独”的构件。而一个综合的工作流管理系统必须能够配置和协调这些不同商业领域和不同组织之间的关系。某一商业领域发生变化都会影响应用程序的配置和其嵌入式工作流系统。因此，一个成功的工作流系统应该可以处理和整合这些不同领域和组织之间的异构性，因此就需要在应用领域层次对工作流进行适配。

2. 过程的动态变化

如果一个工作流模型或者其中的一个活动发生变化，那么就需要在过程层次上进行动态适配。这个层次上的动态适配又分为“模型演变”和工作流实例的设定性修改两种：模型演变的概念来自于商业过程重组，一旦商业过程发生变动，那么对应的工作流模型就需要作相应的修改，其变化的核心内容是工作流模型的重设计、替换和版本控制等；而工作流实例的设定性修改随机地发生在一个工作流实例的运行阶段，这种修改往往需要根据结合不同的实际情况并根据设定的修改策略来使得工作流模型保持一致性和正确性。

Vander Aalst 按照变化的影响范围将过程的变化分成两大类。

（1）临时变化（ad-hoc changes）：该变化只影响一个流程实例或一类流程实例，并不影响流程。导致特别变化的原因是错误、稀少事件的发生、用户的某个特别指令等。异常通常会带来这种变化。一个比较典型的例子就是，当情况比较紧急时，需要跳过某个或某些任务执行以减少整个实例的执行时间。

（2）演进变化（evolutionary changes）：该变化对应流程的结构问题，即工作流过程逻辑发生了变化。在某一个时刻，系统中的某个工作流发生变化，则正在运行的该工作流的实例都要受到影响。导致演进变化的主要原因是业务策略发生变化，业务过程重组的结果，外部条件的永久性修改等。

临时变化的一个最显著的特征就是它并不影响定义的流程，只是作用于工作流实例上。由此，就可能产生很多问题。对于流程实例发生的变化，有可能是由于外部条件发生了变化导致不能按照原有的计划进行（比如在医疗流程中，对某个病人进行某种治疗，但他对所开的某种药过敏，则必须停止当前的所有执行），也有可能是由于某个条件不能被满足致使某个节点不能正常执行。这就说明了临时变化的种类是多种多样的，如果不加限制的话，很可能会由于出现各种各样的异常而使系统瘫痪。所以，工作流管理系统很难决定哪种类型的变化是允许的。同时，由于变化的多样性，要预见所有可能的工作流变化几乎是不可能的。这对正确处理临时变化造成了巨大困难。

演进变化则是要更改流程的定义，它通常影响新的流程实例，但也要影响旧的流程实例。在更新流程时，对已经存在的旧流程实例的处理主要有四种方法。

（1）重新启动：所有已存在的流程实例都被终止，并按新的流程重新启动流程实例。这样就保证了在任何时刻，同一流程的所有实例都保持与该流程的一致性。但对于大多数

的工作流应用来说，由于工作的时间和开销比较大，重新启动工作流实例是不可取的。

（2）终止流程实例：所有已存在的工作流实例都被终止，并不再被处理。这是与现实需求不一致的。

（3）继续执行流程实例：每一个流程实例都与一个特定的工作流版本相对应，新的工作流过程模型并不影响旧的流程实例。当前的许多工作流产品都支持这种版本机制。但该方法仍然存在着不足：旧的流程实例不能参考改进后的路由定义。同时，如果工作流的定义频繁更新，则管理众多的版本信息将成为一个相当复杂的问题。

（4）迁移工作流实例：将现存的工作流实例迁移到新定义的工作流上，也就是说，工作流实例直接受到演进变化的影响。

在这四种方法中，由于第一、二两种方法的不可行性，加之第四种方法的复杂性，现阶段工作流管理系统采用最多的是第三种方法，即采用版本机制处理工作流模型变化（演进变化）。业务过程发生变化时，相应地生成一个工作流模型的新版本。系统中的每一个工作流实例都有一个与之对应的工作流模型版本跟它对应。当某个工作流模型生成新的版本时，之前某个版本的工作流实例仍然按照其原先的工作流版本继续执行。这种版本机制在很多情况下是很实用的。例如：一个流出时间比较短的管理过程就比较适合采用版本机制。

但是，在某些实际应用中，并不希望正在执行的工作流实例按照旧的工作流模型的路由执行，这时就不适宜用版本机制来处理工作流的变化问题。在这种情况下，第四种方法就显得尤为重要了。就拿分期付款来说，它的周期一般是 20 到 30 年。如果每年该过程频繁地变化，这就会导致该过程的很多版本同时运行。为了有效地管理这些流程版本信息，同时又减少开销，我们必须要将当前活动的版本（被工作流实例使用的版本）数量控制在一定的数量上，尽量是越少越好，那么就需要将一些旧版本的工作流实例迁移到新版本上执行。事实上，在很多情况下，我们必须将工作流实例迁移到新的工作流模型上执行。

3. 资源的动态变化

资源适配处理的内容主要是各种工作流资源的重新分配，软件系统的应用程序和其接口的替换和修改、组织结构的变化和数据结构的变化等，这里简要讨论组织结构的变化和相关数据的适配。

（1）组织结构的变化。这类变化最典型的就是人员的变动，这样会直接的影响工作流过程的执行。人员的变动、企业组织结构的变动都会直接反映在工作流的组织资源模型上，这时，就需要工作流系统及时有效地处理这种变化。比如大多数工作流系统都采用了用角色来代替某一特定的员工，用组织角色来代替某一类的职能部门等。

（2）工作流数据的变化。当一个工作流过程正在运行时，其数据或数据结构可能会发生变化。通常讲，一个正处于运行阶段的过程是不允许被工作流管理系统或者外部应用程序修改其相关数据的。如果这个过程依赖于某一被详细定义的全局数据，而这个全局数据发生了变化，工作流管理系统就必须作相应的变动从而使其适应这个数据的变化，因此，工作流系统必须对全局的数据结构有相应的感知能力。

（3）系统基础组织的变化。随着工作流系统的发展，新的功能不断增加、新的技术也会被不断采用，这时反映在工作流系统的基础支持软件系统发生了变化，就需要对工作流系统进行重配置。为了及时、准确、方便地对工作流进行再配置，就需要一个柔性的基础

支持系统的适配器，比如采用基于 XML 的配置方法等。

4.5.4 动态修改的策略或处理

1. 常见的几种修改策略

常见的动态修改的策略有 Flush、Abort、Migrate、Adapt 和 Build，其含义如下。

（1）Flush（刷新），指的是清除当前的工作流执行实例，然后修改工作流模型，然后从下一个实例开始执行新版本的工作流模型；

（2）Abort（取消），在这种模式下仅仅取消当前执行的工作流实例，而并不做其他的动态修改；

（3）Migrate（迁移），在这种模式下，当运行的过程发生变化后将转至新修改后的流程接着执行；

（4）Adapt（适配），在适配模式下，当过程发生异常时，将根据发生异常的不同类型而选择不同的候选流程；

（5）Build（编译），在该模式中，系统将根据需要在线重新编译被修改的模型，从而使之添加适当的模型来满足需要。

这五种方式各有特色。Flush 模式、Abort 模式在传统的工作流系统中很容易扩展，而且基本上所有的工作流管理系统都支持这两者，但这两种模式的缺点也是显而易见的，它们都不具备“在线编辑”的能力，即在运行阶段修改的能力，只能是在模型的建立阶段进行修改。Migrate、Adapt 和 Build 模式都在不同的程度提供了运行时修改的能力，但侧重点不同。Migrate 的方式是“此路不通”则“另寻他路”的方式，然后改走新增的流程，不需要提前准备候选路径。而 Adapt 方式则是在“此路不通”的情况下，选择候选路径来进行，否则只好 Abort 了。目前大部分的适应性工作流系统都采用这种模式，而且有了比较广泛的应用；但其缺点也比较明显，即必须事先预测到会发生某种类型的异常，并建立备选流程。Build 模式是一种比较理想的模式，它的思想是提供一种彻底的“在线修改”模式，从而可以明显提高工作流过程的适应能力，但这种模式实现起来非常复杂，目前还处于研究阶段。

2. 面向数据的工作流动态修改模型策略

面向数据的动态修改模型 DOAWFMS（Data Oriented Adaptive Workflow Management System），其核心思想是通过一系列检测算法来保证工作流数据的一致性和完整性，从而确保工作流在动态修改后不会破坏工作流数据的依赖性。

纵观软件系统的发展和信息系统的发展，数据一直是程序处理的核心，也是信息的载体，而工作流系统是信息系统的一种。在工作流系统中，是通过控制流来保证数据被处理的语义正确性的。在对工作流进行动态修改时，Migrate 模式、Adapt 模式和 Build 模式都是通过修改过程的管理员来掌握数据的依赖性的，但在流程比较大，而管理员又对该过程不是非常熟悉时，如何确保工作流数据的一致性是一件非常复杂的问题。这时管理员在对工作流实例作动态修改时就必须掌握每个活动所处理的数据，以及这些数据之间的依赖关

系，因此管理员的工作量非常大，如果修改不当则还会引起更多的问题。DOAWFMS 模型可解决此问题，它可以在管理员对工作流实例作动态修改时，实时地检测每个数据及其之间的依赖性，一旦发生数据不一致则对管理员提出警告。

为了保证数据之间的一致性和完整性，需要对数据依赖性做出相应的定义。在一个应用程序的参数表中，如果存在数据 a 、b 和 c ，并且 a 和 b 是输入数据，而 c 是输出数据，那么我们称数据 a 、b 、c 之间存在偏序依赖关系→：（ a ， b ）→ c ，称为数据 c 依赖数据 a 和 b 。数据依赖关系主要是为了满足数据的连贯性，如果（ a ， b ）→ c ，那么，如果包含 a 或者 b 的节点被删除的话，那么包含 c 的节点也应该作相应的修改，否则数据流中就会出现无源数据。

这种偏序依赖关系“→”可以分为直接依赖和间接依赖两种，即如果 a → b ，而且 b → c ，那么 b 直接依赖 a ， c 直接依赖 b ，而 c 间接依赖 a 。也可以说这种偏序关系存在传递性。

数据依赖关系是 DOAWFMS 模型研究的核心之一，并在该依赖关系的基础上给出了一个检测数据连续性的算法 CheckDataflow。该算法首先对需要修改的工作流过程所在的工作流包作全局扫描，构建所有数据（全局数据和相关数据）的偏序关系集；然后考察每一个需要被修改的活动中的数据，找到它所能影响的所有数据，并检测这些数据之间的依赖性。在该算法中，用 petri 网来对数据之间的依赖性建模，并与工作流过程的 petri 网模型相互结合，用 petri 网的一些理论来判断是否满足数据之间的依赖性。

3. 其他的策略

清华大学的史美林提出了支持动态特性的工作流过程元模型，该模型从时间和工作流组成的过程级别两个角度分析了其动态特性的表现，并提出了相应的解决方法:

（1）在过程建立阶段对过程一级的动态支持表现在对流程控制变化的描述和组成内容不确定的描述，在原有的流程控制类型（顺序执行、并行执行 AND 和选择执行 OR）的基础上添加循环条件控制（DO-WHILE，UNTIL），并在原有的几类活动节点（起始活动、结束活动、原子活动、子活动和块活动等）中添加一个新的节点类型－黑盒活动，用于标志一个语义不确定的节点；

（2）在过程建立阶段对活动一级的动态支持表现在对活动属性的扩充，将传统的活动执行类别增加为必须的、可选的和可重复的，并通过规则来灵活定义活动的行为；

（3）在过程执行阶段对过程一级和活动一级的动态支持表现在对执行阶段动态修改的处理方法与过程的描述，提出了修改过程的 5 个阶段:暂停执行、人工修改、修改后移交、连贯性验证、继续执行。基于该模型，工作流管理系统既有灵活性，又有利于控制变化和操作。

浙江大学的吴朝晖采纳并发展了史美林提出的扩展的工作流元模型的一些想法，在文章《动态工作流建模方法的研究与设计》中提出了一种基于事件－条件－活动规则和活动组合的动态工作流模型。该模型中用柔性活动封装流程中的不确定因素，用选取规则和组合规则来约束柔性活动的具体化过程，并设计了一个追求活动最高并发的活动自动组合算法，以便最大限度地利用系统资源，提高流程执行效率。并且还设计了一个检验活动选择和活动组合的算法，保证了柔性活动具体化后形成的子过程的合法性。

虽然动态工作流得到越来越多的工作流厂商和工作流理论研究者的关注，他们分别从

不同的角度提出自己的解决方案，但由于工作流适应性问题的复杂性等原因，工作流的适应性研究还远远不能满足实际问题的需要，这给工作流的实际应用和大规模的推广带来了很大的限制。

4.5.5 应用示例

本节以 AdaptiveWF 为例介绍工作流过程模型进行动态修改的操作方法，包括调整活动之间的执行依赖关系、添加一些新的活动、删除一些活动，调整活动之间的数据依赖关系等。

1. AdaptiveWF 中动态修改策略

在 AdaptiveWF 中，当动态修改的作用域是过程模型时，我们支持重新启动，继续运行和迁移三种策略，但将迁移策略作为其默认的动态修改策略，让已经运行的过程实例迁移到新的过程模型，并按照新的过程模型继续运行下去。但是在应用迁移策略将已经运行的过程实例迁移到新的过程模型上执行时，这些过程实例一般处于不同的执行进度。这时将多个过程实例生硬地直接转换为同一个过程模型显然是不合适的。因为这有可能造成很多问题，比如某些过程实例在转换后无法继续向下执行等。于是采用了一种称为迁移点的方法来处理演进变化中工作流实例的迁移。该方法为修改前（旧）工作流过程模型中每个节点生成一个迁移规则，不同运行进度的工作流实例根据当前的执行位置选择相应的规则执行，即可完成工作流实例的迁移。

当然，工作流过程模型的动态修改是过程级的修改，它一般是由工作流建模人员在流程定义工具中完成的。所以该类修改操作的正确性保证工作只需由验证工具来完成。如果修改后的过程模型不正确，则撤销所有修改。该过程是一个修改－验证……修改－验证－提交的过程。

2. AdaptiveWF 中动态修改操作

在 AdaptiveWF 的工作流模型的基础上，定义了一个支持工作流演进变化的动态修改操作的集合。工作流的演进变化不仅仅是对单个工作流中的某些节点或连接弧进行修改，也有可能是对工作流本身的修改。前者叫做过程级的修改，后者叫做版本级的修改。例如，添加或删除一个工作流过程模型版本，就属于版本级的修改。修改操作原语定义如下。

（1）版本级的修改

AddProcess（Process p）：添加一个工作流模型。该操作仅仅是添加了一个独立的工作流模型，对现有工作流模型并不会造成影响。

DeleteProcess（Process p, int v）：删除一个工作流模型 P 的第 v 个版本，该操作的前提条件是没有正在运行的该版本的工作流实例。

Modify（Process p, int v）：修改一个工作流过程模型 P 的第 v 个版本，将修改后的过程定义作为新的版本保存。

该类修改操作由版本管理器来完成。

（2）过程级的修改

AddTaskAtt (node t, attribute x, value a) ：向任务节点 t 添加属性 x，其初始值为 a

DelTaskAtt (node t, attribute x) ：删除任务节点 t 的属性 x

AddNode (node t, node a, node b) ：在节点 a 和节点 b 之间添加一个节点 t，则

Succ(a)=t∧Succ(t)=b ∧ t=Prec(b) ∧a=Prec(t)

DeleteNode (node t}：删除一个任务节点 t，则

Succ(Prec(t))=Succ(t) ∧Prec(Succ(t)) =Prec(t)。

AddDL(RL dl, D d, node n, Par par, mode t):在节点 n 和数据 d 之间添加一条类型为 t 的数据连接弧 dl

DeleteDL (RL dl) ：删除数据连接弧 dl

AddData(D x, type t) ：添加类型为 t 的数据元素 x

DeleteData(D x) ：删除数据元素 x

从上述操作可以看出，添加或删除节点操作包括了相关控制连接弧的添加和删除操作，因此没有提供单独的控制连接弧操作。

对于过程级修改，每个修改操作都至少有一个相关的节点，例如与 DeleteTask(Task t) 操作原语相关的节点为任务节点 t。将该相关节点叫做修改点。

对上述各操作的修改点定义如表 4-1，对于数据连接弧和数据元素的修改不影响工作流的控制流结构，因此在此不予考虑。

表 4-1　修改点定义表

修 改 操 作	修 改 点
AddTaskAtt(*node t*,*attribute x*,*value a*)	t
DelTaskAtt(*node t*,*attribute x*)	t
AddTask(*node t*,*node a*,*node b*)	b
DeleteTask(*node t*)	t

若将一个工作流模型进行若干修改，根据每个修改点在工作流模型中是否连续进行分组，将连续的修改点分成一组，每一组对应一个变化区域。将每个变化区域的前一个节点称为该变化区域的转化点（即第一个节点的直接前驱节点）。我们将一个工作流模型经过若干修改操作作用的过程用一个称为“变化模型”的模型来描述。

第 5 章　动态配置技术

5.1　动态配置系统体系结构

动态配置平台由系统信息收集部件、系统信息库和动态配置引擎三大类部件组成，其典型体系结构如图 5-1 所示。系统信息收集部件将收集到的应用系统的体系结构信息和部分有关构件特性的语义信息存放于系统信息库中。利用在系统信息库中存放的对当前系统结构完整的描述信息以及部分语义信息，高层的配置及管理人员可获得应用系统的全局视图，并基于对系统的分析以及参照要达到的特定管理目标，产生相应的动态演化意图，交由动态配置平台实现。动态配置引擎利用系统信息库中的系统结构和语义信息，同时通过系统信息收集部件获取构件实时运行状态信息，从而在适当时机调控构件行为，实现动态演化意图。在智能化的动态配置系统中，动态配置平台本身还可通过分析系统的当前结构和语义信息，自动产生动态配置需求，并予以实施。

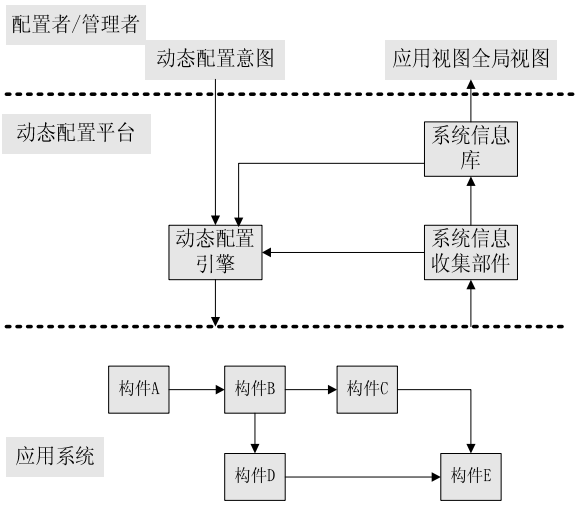


图 5-1　动态配置系统体系结构

5.2　动态配置方法的分类

通过对现有动态配置方法进行总结，我们将其分类，如图 5-2 所示。

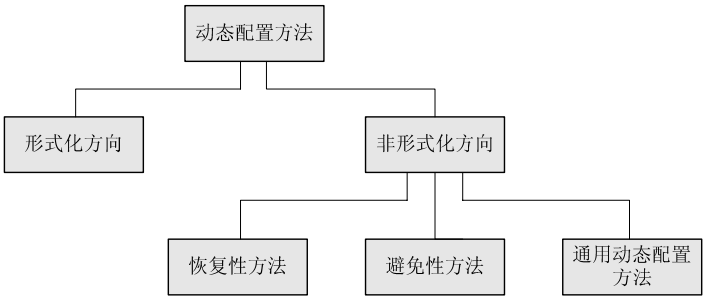


图 5-2　动态配置方法分类

目前动态配置的研究主要从两个方向展开：形式化方向和非形式化方向。

形式化方向的研究重点在于对系统结构、语义、约束和配置变化的形式化描述以及相关性质的形式化验证，包括动态演化意图本身的合法性，动态配置操作序列是否保证系统一致性，以及系统在动态配置后是否具有期望的性质，如不存在死锁、活锁等。形式化方向的主要研究内容是形式化的软件体系结构描述语言。Rapide、Darwin、Wright、Unicon、CHAMP 等形式化的软件体系结构描述语言，不仅支持对系统结构信息的描述，而且利用 CSP、CCS、π 演算等形式化工具还支持对系统行为语义的描述，从而达到精确、无歧义描述系统的目的。形式化分析工具将根据对系统的精确描述，对系统特性进行推理和判定。但是软件体系结构作为一个新兴的研究方向，远未达到其预期的理想目标。利用形式化的软件体系结构描述语言对系统的描述和分析技术，特别是对系统动态特性的描述和分析技术还远未成熟。

非形式化的动态配置方法采用 XML 等非形式化工具描述应用系统及其配置变化。它们或者采用基于 XML 的体系结构描述语言描述系统，如 XADL，或者根据具体需求直接用 XML 描述系统，如分布式构件技术中对应用系统组装部署信息的描述。虽然 XML 不是一种形式化语言，无法做到对系统的精确、无歧义描述，但是 XML 凭借易用性、扩展性和强大、灵活的描述能力，易于被开发者接受，并被广泛使用。另外，由于对系统的形式化分析和验证尚处于研究阶段，大部分系统实际上也并不真正需要形式化的精确描述，而只要求根据使用需求准确、清晰描述所需信息。对于这种要求，XML 足够胜任。因此 XML 成为实际系统中描述系统结构、语义等信息的最佳选择。与形式化的动态配置方法不同，非形式化的动态配置方法主要从工程实践的角度出发，在假设动态演化意图合法的前提下，研究动态配置的关键技术和算法，并最终实现通用的动态配置平台。表 5-1 对动态配置研究的形式化方向和非形式化方向进行了比较。

表 5-1　动态配置方法形式化研究方向和非形式化研究方向的比较

	形式化方向	非形式化方向
应用系统和配置变化描述	形式化描述	非形式化描述
系统一致性保证	形式化验证	动态配置算法的合理设计
通用性	专用	通用
动态配置性能	不考虑	动态配置算法及动态配置平台的合理设计动态配置实现的并行性
实用性	研究阶段	实用动态配置平台

根据保证系统一致性的方式不同，非形式化的动态配置方法又分为恢复性方法、避免性方法和通用动态配置框架方法。

（1）恢复性方法：恢复性方法设计的动态配置算法本身并不能完全保证系统一致性。根据记录的检查点，当动态配置破坏系统一致性时，恢复性方法将系统回滚到特定检查点，从而保证系统一致性。法国格勒诺布尔国立理工学院 L.Oueichek 和东京 NTT 软件实验室的 H.Higaki 等人对此类方法进行了研究。

（2）避免性方法：避免性方法在设计动态配置算法时始终以防止破坏系统一致性为目的，实施动态配置算法时则通常采用暂停构件行为的手段来保证系统一致性。避免性方法的设计理念是，如果系统被动态配置影响的部分在动态配置前被冻结在一致性状态，则系统一致性不会被破坏。基于这个理念，避免性方法的执行过程通常分为三个阶段，首先，确定被动态配置影响的构件集合，驱动这些构件进入预期的冻结状态，这种状态也被称为动态配置安全状态；其后，实施动态演化意图；最后，恢复系统的正常运行。

（3）通用动态配置框架方法：无论是恢复性方法，还是避免性方法，在进行动态配置算法的设计和动态配置平台的实现时，其目的都是为了保证特定的一种或几种系统一致性。与之不同的是，为支持动态配置的灵活实现，通用动态配置框架方法设计了特殊的系统框架和结构。以此为基础，开发者可根据特定的应用语义以及对动态配置功能和系统一致性的要求，定制动态配置的实现方法。美国伊利诺大学的 Fabio 等人在 2K 项目中提出了以 configurator 对象为核心的通用的动态配置框架，而美国罗格斯多大学 Minsky 等人通过为每个构件提供一个控制器，内存构件的当前状态和约束构件交互行为的规则，从而提供了一个通用的动态配置框架。

恢复性方法通过系统回滚保证系统一致性，缺点是代价过大，适用范围小，保证系统一致性的能力有限，而且系统无法回滚以及连锁回滚等问题都无法解决。

通用动态配置框架方法虽然为动态配置提供了个性化支持，不局限于特定的应用系统和系统一致性要求，但这种方法需要开发者的大量参与，而且基于该框架的系统一旦实现，系统支持的动态配置功能和保证的系统一致性也就确定，并不能提供真正的自适应性。

与恢复性方法相比，避免性方法对应用系统本身没有过高要求，适用范围较广，动态配置性能较好，动态配置能力也较强。与通用动态配置框架方法相比，虽然恢复性方法只能保证特定的一种或几种一致性，但它不需要开发者的大量参与，提高了动态配置平台的透明性和可靠性，而且真正满足了系统在线演化的需求。因而，避免性方法是一种比较先进的方法，很多研究都集中在此类方法上。

5.3 避免性动态配置方法

本节重点介绍几种典型的避免性动态配置方法，并对避免性动态配置方法研究中存在的问题进行讨论。

5.3.1　Jeff 方法

Jeff 方法认为系统一致性由相互一致性和应用状态一致性组成。在系统运行过程中，构件之间存在大量交互行为。动态配置必须保证构件间交互行为的完整性，即任意交互行为都不能由于动态配置的实施而被中断，这种约束被定义为相互一致性（mutual consistency）。Jeff.Kramer 和 Jeff.Magee 最早提出相互一致性的概念，同时提出静止状态理论，并基于这套理论给出了保证相互一致性的动态配置算法。Jeff 的静止状态理论十分简洁，但面对纷繁复杂的构件间交互行为，却能很好地理清脉络，保证所有交互行为的完整性。这一切依赖于 Jeff 对构件行为的透彻分析和巧妙分类。Jeff 的静止状态理论成为众多后继动态配置研究的基础。Jeff 方法除了保证相互一致性，还保证了应用状态一致性。一个应用系统可能存在系统断言，对系统中一组构件的状态进行了约定。例如，在基于令牌环结构实现的系统中，所有构件只能拥有一个令牌。动态配置不能破坏这种全局的状态约束，这就是所谓的应用状态一致性。

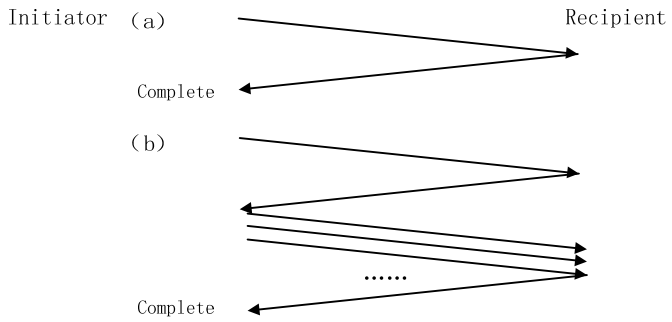


图 5-3　事务的组成

Jeff 将分布式系统定义为由构件和构件之间用于交互的连接构成。连接起始构件向连接终止构件发送请求。两构件间一次完整的信息交互过程被定义为一个事务（transaction）。一个事务可能由若干次构件发送请求并接收应答的过程组成，如图 5-3 所示。Jeff 的事务概念仅定义了一个语法单位，到底事务中包含多少次发送请求和接收应答的过程则由应用语义决定仅用于标志一段构件的执行期间，构件在这个期间处于不一致的状态中。主动启动信息交互过程的构件为事务启动构件，被动参与信息交互过程的构件为事务参与构件。任意事务都在有限时间内完成，而且事务启动构件明确事务何时完成。构件的状态只能通过事务改变。基于事务概念，构件交互行为的完整性具体表现为事务的完整性。当一个事务的完成不依赖于其他事务时，即接收请求的构件无需向其他构件发送新的请求便可响应该请求并返回应答，则这种事务被称为独立性事务，否则称为依赖性事务。引入依赖性事务后，构件在两种情况下启动事务。一种情况是，构件由于内在驱动而启动新的事务。另一种情况则是，构件为了参与由其他构件启动的事务而启动新的事务。为了区别这两种情况，我们将第一种情况称为构件自行启动事务。如果构件自行启动了一个依赖性事务 T，则在该事务的完成过程中，必定有若干新事务被启动。事务 T 的参与构件不仅包括直接参与事务 T 的构件，还包括所有这些新事务的启动和参与构件。Jeff 首先基于独立性事务提出静止状态理论，再针对依赖性事务对静止状态理论进行了扩展。由于独立性事务是依赖

性事务的特例，因此本章仅介绍针对依赖性事务扩展后的静止状态理论。

Jeff 认为构件的状态仅由事务改变。为了确定何时构件处于一致性状态，Jeff 分析了构件的交互行为，并将其分为四类：构件当前参与自行启动的事务的行为（B1）；构件将来自行启动事务的行为（B2）；构件当前参与由其他构件启动的事务的行为（B3）；构件将来参与由其他构件启动的事务的行为（B4）。这四类行为都将导致构件当前或将来处于不一致的状态中。因此，Jeff 要求构件在动态配置前必须处于静止状态（quiescent state ）：

（1）构件当前没有参与自行启动的事务；

（2）构件将来不会自行启动事务；

（3）构件当前没有参与由其他构件启动的事务；

（4）构件将来不会参与由其他构件启动的事务，包括已经启动和即将启动的需要该构件参与的事务，直到动态配置结束。

当条件 1，2 满足时，构件不存在自行启动的事务，只会被动参与由其他构件启动的事务，称为被动状态（passive state）。与之相对的，构件既可以自行启动事务，又可以参与事务的正常运行状态，称为主动状态（active state）。当构件处于静止状态时，构件的状态是一致的，因为不包含完成一半的事务的状态。构件的状态也是“冰冻”的，因其状态不会被任何新的事务所改变。此时对构件实施任何动态配置都不会打断任何事务，从而保证了事务完整性，进而保证了相互一致性。对于为了保证应用状态一致性而需要设置构件状态的情况来说，构件冰冻的一致性状态又为构件状态的读写提供了一个稳定的、一致的环境。因此，在构件处于静止状态时实施动态配置，可以保证相互一致性和应用状态一致性。

基于对构件静止状态和被动状态的定义，Jeff 通过证明其满足的三条性质，给出了驱动构件进入静止状态的具体算法。

性质 1：构件的被动状态是可达的；

构件 Q 的被动集合 PS(Q)，由 Q 以及所有可以自行启动需要 Q 参与的事务的构件组成。

性质 2：当 PS(Q)中的构件都进入被动状态时，构件 Q 进入静止状态；

性质 3：构件的静止状态是可达的。

以上对构件静止状态、被动状态的定义以及相应性质的证明构成了 Jeff 的静止状态理论。根据静止状态理论，Jeff 进一步分析了构件创建（create）、构件删除（remove）、连接建立（link）和连接删除（unlink）四个基本动态配置操作实现的前提条件，最终给出了实现这四个基本动态配置操作的动态配置算法。

remove：在删除构件之前，构件必须是静止的和孤立的。所谓孤立是指，不存在从该构件出发到达其他构件的连接，同时也不存在从其他构件出发到达该构件的连接。对一个孤立构件的任何操作不会对系统造成任何影响，因此孤立构件可以在保证系统一致性的前提下被安全删除。

link & unlink：建立和删除连接的前提是，连接的起始构件必须处于静止状态。根据对构件静止状态的分析，此时删除连接不会破坏相互一致性，而且也为若干构件状态的设定提供了稳定环境，从而保证了应用状态一致性。

create：构件在被创建之初，还未与其他构件建立连接，不可能参与任何事务，必然是静止的和孤立的，因此构件的创建没有任何前提条件。

动态配置算法设计如下：

第一步：根据要执行的动态配置操作，可能包括若干构件创建、构件删除、连接建立和连接删除操作，确定在实施动态配置前需要进入静止状态的构件集合 QS。如果要删除构件，则所有要被删除的构件在孤立化过程中需要被删除的所有连接构成了集合 CS。如果要删除或建立连接，则所有要被删除或建立的连接构成了集合 LS。根据 CS，LS，以及各基本动态配置操作执行的前提条件，确定 QS。各集合具体定义如下：

CS={连接 c|对于一个要被删除的节点，c 是由其启动到达其他构件的连接，或是由其他构件启动到达该构件的连接}

LS={连接 1|1 是要被建立或删除的连接}

QS={构件 n|n 是（LS∪CS）中连接的起始构件，或是要被删除的构件}

第二步：根据 QS，构造动态配置前需要进入被动状态的构件集合 CPS。

CPS =∪PS(i) }，i 是 QS 中的构件

第三步：按照特定顺序实施动态配置：首先驱动 CPS 中的所有构件进入被动状态；然后依次执行删除连接、删除构件、创建构件和建立连接的操作；最后驱动<CPS－被删除的构件＋被创建的构件>中的所有构件进入主动状态。

构件不仅感知动态配置的存在，而且必须主动参与动态配置的实施。实际上，为了支持动态配置算法的实现，系统中的每个构件都必须实现四个方法：驱动并确认自身进入被动状态的 passivate 方法；驱动自身进入主动状态的 activate 方法;建立连接时进行构件状态设定从而保证应用状态一致性的 link 方法;删除连接时进行构件状态设定从而保证应用状态一致性的 unlink 方法。动态配置管理器 CM（Change Management）将在执行动态配置算法的相应时机，调用构件的相应方法。构件在方法完成时，通知 CM，使其继续算法的执行。这四个方法只是为构件提供了时机允许其参与动态配置，从而保证系统一致性。至于这四个方法的具体实现方式，则完全由构件根据语义自行决定。如果构件具备自行启动事务的能力，则在 passivate 方法中，构件将首先禁止将来自行启动事务，然后进入等待状态直到已经自行启动的事务全部完成，通知 CM，方法结束。相应的在 activate 方法中，构件只需解除对将来自行启动事务的禁止即可。但是如果构件根本就不具备自行启动事务的能力，则除了在被调用后即刻通知 CM 外，其 passivate/activate 方法基本为空。同理，unlink/link 方法可能基本为空，也可能需要通过与其他构件进行复杂的交互实现对一组构件状态的设定。

在系统管理和分析中，常常需要对系统进行层次化的抽象。在某个抽象层次上，动态配置的目标是一个构件，而其在底层可能实际对应着多个子构件。这种由子构件构成的构件被称为组合构件，也就是模块。Jeff 方法支持对模块的动态配置。模块进入被动状态，当且仅当所有子构件全部进入被动状态。当且仅当所有子构件全部进入静止状态，模块进入静止状态。

Jeff 的静止状态理论，以及基于该理论提出的动态配置算法，不局限于任何特定的应用语义和实现方式，是一种通用的方法。

Jeff 虽然在静止状态理论中给出了被动集合 PS 的精确定义，却没有给出在实际系统中计算 PS 的方法。但是对于基于静止状态理论的动态配置算法来说，PS 的精确计算是提高动态配置算法性能的关键。

应用系统的封闭性和可控制性，是静止状态理论实现的基础。只有掌握整个系统的体系结构信息，才可能正确计算 PS。只有当系统中的所有构件都实现了 passivate 和 activate

方法并可被调用，目标构件才能进入静止状态，相互一致性才能被保证。但事实上，大多数应用系统并不满足此性质。通常，系统是开放的而且仅有一部分落在配置管理器的控制边界内，典型的如带有客户端的系统。对于这样的系统，如何基于有限的体系结构信息和控制能力，正常驱动控制边界内的构件进入静止状态，就是所谓的边界问题。Jeff 方法没有对边界问题展开分析。

5.3.2 Warren 方法

除了交互行为，构件还存在内部进行的本地行为，如读写文件的行为。构件的本地行为也会对构件状态造成影响。Warren 不仅考虑了构件交互行为的完整性，而且考虑了构件本地行为的完整性，提出了本地一致性。Warren 方法支持构件删除、添加、替换和迁移的实现。

根据 Jeff 方法对构件交互行为的分类，在保证相互一致性时，Warren 方法要求构件在动态配置前必须处于以下状态：

（1）构件当前没有参与对自行发出的请求的处理，即所有发出的请求都已经得到了所需的应答；

（2）构件将来不会自行向外发出请求；

（3）构件当前没有参与响应由其他构件发来的请求；

（4）构件将来不会接收到由其他构件发来的请求，直到动态配置结束。

对应于构件自行启动的事务，构件自行发出的请求是指构件由于内在驱动主动发出的请求，而不是为了响应到达的请求而发出的请求。不难发现，这种状态与 Jeff 方法中定义的构件静止状态十分类似，区别在于前者针对的是构件参与单个请求处理的行为，而后者针对的是构件参与事务的行为。我们将 Warren 方法中定义的这种状态命名为基于请求的静止状态，以示区别。实施构件替换和迁移前，Warren 将暂时阻塞到达构件的请求，以保证构件在动态配置期间不会接收到由其他构件发来的请求。被阻塞的请求将在动态配置后被继续响应。

为了保证本地行为的完整性，Warren 提出了本地一致性：要求实施动态配置时，目标构件在满足相互一致性的基础上，本地没有正在进行的关键操作。构件在完成本地行为时，可能需要同时完成与其他构件的交互行为。例如，在一个生产者－消费者系统中，生产者构件产生数据后，将数据发送到缓冲区中，消费者构件从中——取走数据。生产者构件产生数据的行为属于构件本地行为，发送数据的行为属于构件交互行为。如果存在系统约束，要求一旦生产者构件开始产生数据，就不能中断它发送数据到缓冲区的行为，则构件在完成产生数据的本地行为后，就必须完成发送数据的交互行为。此时称构件的本地行为依赖于构件的交互行为。为了保证这种本地行为的完整性，必须保证其所依赖的构件交互行为的完整性。Warren 提出的解决办法是，一旦构件启动具有依赖性的本地行为，则该行为依赖的所有构件交互行为使用的构件提供服务端口全部被加锁（lock），直到本地行为完成后，所有端口上加的锁才被去掉（unlock）。当构件的所有提供服务端口都没有加锁时，才可对其实施动态配置操作，从而本地行为的完整性得以保证。

实施构件替换和迁移时，构件状态需要在构件间传递。动态配置平台必须保证状态被

正确收集，并可能根据应用需要进行适当转换，从而正确初始化接收状态的构件，这就是构件状态一致性。Warren 为了保证构件状态一致性，要求构件实现状态读写（capture/restore）方法。动态配置管理器将在合适时机调用这些方法，传递状态。

为了保证应用状态一致性，Warren 方法要求构件在构造函数和析构函数（constructor/destructor）中设置自身或其他构件状态的设定，从而在构件加入和离开系统时可保证满足系统对构件状态的全局约束。

Warren 方法仅停留在概念和想法一级，没有给出保证相互一致性和本地一致性的具体可行算法或实现框架。但是 Warren 方法提出的本地一致性，弥补了 Jeff 方法对本地行为完整性方面考虑的不足。

5.3.3　其他方法

1. Almeida 方法

荷兰特文特大学的 J.P.A.Almeida 等人基于 CORBA 平台实现了透明的动态配置服务，支持 CORBA 对象的创建、删除、替换和迁移。为了保证相互一致性，Almeida 方法通过阻塞后继到达对象的请求，并等待对象完成当前参与的请求后，才允许对其实施动态配置。虽然同是保证相互一致性，但 Almeida 方法与 Goudarzi 方法并不相同，最明显的区别在于 Almeida 方法和 Goudarzi 方法分别针对请求和事务进行处理。

在 Almeida 方法实现的动态配置服务中，每个可被动态配置的 CORBA 对象都存在于一个动态配置对象胶囊（capsule）中。每个动态配置对象胶囊包含一个工厂对象、该工厂对象创建的所有动态配置对象，以及一个动态配置代理（Reconfiguration Agent，RA）、RA 负责对到达该胶囊内动态配置对象的请求进行阻塞。基于实现池实现的位置代理（Location Agent，LA）则负责记录对象的引用信息。动态配置时，动态配置管理器（Reconfiguration Manager，RM）首先要求目标对象所在胶囊的 RA 阻塞到达目标对象的请求，然后在驱动目标对象进入没有参与请求处理的动态配置安全状态后，执行动态配置操作，同时修改 LA 中存放的对象引用信息，包括对象引用的注册、更新和删除。RM 利用工厂对象实现对象的创建或删除，并在必要时调用对象实现的状态访问方法（get/set）读写对象状态，从而满足对象替换或迁移过程中在对象间传递状态的需求。为了解决引用完整性问题，即对象在替换和迁移后由于引用的改变而导致其客户端对象无法正常访问的问题，RA 在阻塞请求时，将利用 CORBA 的截获器机制向发送该请求的客户端对象抛出特定异常。该异常被客户端对象所在的 ORB 截获并将请求暂时阻塞。待接收到来自 RM 的动态配置结束消息后，客户端 ORB 将利用 LA 中正确的目标对象引用信息，重新发送被阻塞的请求。客户端对象除了感觉到请求响应时间增大以外，以上的请求阻塞和重发过程对客户端对象完全透明，全部由客户端 ORB 自动实现。此为透明的动态配置服务的含义。

2. XRMI 方法

德国斯图加特大学的 Chen，Xuejun 等人通过扩展 JAVA RMI，引入虚拟 Stub，自动监控构件间的交互行为，从而实现对分布式系统的动态配置。在 XRMI 方法中，构件发送请

求时，获得的不是传统的 RMI Stub，而是虚拟 Stub。虚拟 Stub 在通过 RMI Stub 进行请求发送前和应答接收后，会根据动态配置的需要进行相应处理。虚拟 Stub 记录了构件间随交互行为建立起的动态依赖关系。向外发送请求时，虚拟 Stub 记录请求信息，包括客户方构件名、服务方构件名以及服务方构件被调用的方法名等。接收应答时，虚拟 Stub 将删除所记录的相应请求信息。动态配置时，动态配置代理器利用虚拟 Stub，阻塞其他构件向目标构件发送请求，阻塞目标构件向外发送请求，并根据虚拟 Stub 记录的构件间动态依赖关系来判断目标构件何时完成正在参与的请求，从而在驱动构件进入没有参与任何请求处理的状态后，实施动态配置。在构件替换和迁移后，动态配置管理器还将利用虚拟 Stub 自动更改请求中的服务端对象引用信息，从而保证请求的正确发送，解决引用完整性问题。

3. WebFrame

WebFrame 是中科院软件所自主研制的 J2EE 服务器，实现了 EJB 构件的动态迁移，并在此过程中保证了引用一致性、状态一致性和消息一致性。所谓引用一致性，即指必须解决动态配置过程中的引用完整性问题。所谓消息一致性，则指在删除构件时，构件没有正在处理及等待其处理的请求。消息一致性保证了来自客户的请求不会由于构件的迁移而丢失。状态一致性则强调了构件状态在迁移过程中的正确传递。

不妨将迁移前后的构件分别称为旧目标构件和新目标构件。迁移构件时，WebFrame 首先创建新目标构件，在等待旧目标构件完成正在处理的来自其他构件的请求后，然后将等待其处理的请求转发到新目标构件，进而删除旧目标构件，完成迁移，由此保证了消息一致性。在此过程中，还将进行 JNDI 名的重绑定、构件状态传递等操作，以保证引用一致性和状态一致性。

4. 基于容错 CORBA 平台的方法

基于容错 CORBA，OMG 制定了 CORBA 对象在线升级规范。所谓在线升级，即在系统运行时刻用高版本的对象替换系统中现有版本的对象，对应于动态配置中构件替换的概念。在线升级时，在线升级管理器首先通过与容错 CORBA 的对象组管理器交互，为需要在线升级的对象创建一个对象组。新、旧版本两个对象都是对象组的成员。然后，在线升级管理器阻塞到达旧版本对象的请求，并通过调用旧版本对象实现的特定方法，禁止其主动向外发送请求。在等待旧版本对象完成当前参与的请求后，在线升级管理器调用要求新、旧版本对象实现的状态读写方法，传递对象状态。最后，新版本对象被启用，处理在线升级过程中被阻塞的请求和后继请求。旧版本对象被从对象组中删除。由于新、旧版本对象位于同一个对象组中，基于 Locationes Forward 机制实现的容错 CORBA 的对象组引用机制将保证所有发向旧版本对象的请求都会被透明地正确定位到新版本对象，由此保证了引用一致性。

美国加利福尼亚大学的 L.A.Tewksbury 等人基于特定的容错 CORBA 平台 Eternal 实现了 CORBA 对象的在线升级。在 Tewksbury 方法中，在线升级系统由预处理器和在线升级器两个主要构件组成。在线升级前，预处理器首先离线地分析新、旧版本的源代码，并在开发者的辅助下，生成中间版本对象。中间版本对象是新、旧版本对象的超集，不仅包含新、旧版本对象的所有方法，而且含有状态传递和转换代码。根据新、旧版本对象的接口

变化情况，预处理器确定需要在线升级的对象集合，并生成执行和回滚在线升级过程的脚本，提交给在线升级器。每个在线升级对象都必须至少有一个副本。在线升级时，在线升级器首先用中间版本对象替换旧版本对象。一次只能替换一个对象副本，从而保证至少有一个对象副本处于工作状态，对于整个对象组来说达到向客户端持续提供服务的效果。此时的中间版本对象按照旧版本对象的行为运行。当所有的对象副本都被替换后，通过一个原子性动作，将所有中间版本对象的行为切换到新版本对象。最后再逐个用新版本对象替换中间版本对象。为了保证对象多个副本的一致性，在每次替换完一个副本后，都会用其他副本的状态初始化替换后的副本。为了保证副本读写状态的正确性，以及对象行为切换的原子性，Tewksbury 通过阻塞到达所有对象副本的请求，禁止所有对象副本主动向外发送请求，并等待所有对象副本完成正在处理的请求后，才进行对象副本间状态的传递以及行为的切换。

5.3.4　避免性动态配置方法中存在的不足

关于避免性动态配置方法的研究还有很多，但是它们或者与上述动态配置方法类似，或者仅提供了一个粗略的动态配置框架为系统一致性提供支持，而没有对系统一致性进行深入研究，本书就不一一介绍了。

通过总结现有避免性方法在系统一致性分析、动态配置算法的设计以及动态配置平台实现等各方面的工作，我们发现现有避免性动态配置方法主要存在以下问题：

动态配置模型的缺乏：现有避免性动态配置方法在对系统一致性内容的定义、动态配置功能实现粒度的确定、动态配置算法的设计以及平台的选择和具体实现方面，存在很大差异。多种因素的混杂，使我们很难对现有研究进行准确定位，并从正确性、功能性、性能、复杂性、可靠性和透明性等方面对现有研究进行有效的分析和评价。而另一方面，我们在设计和实现动态配置系统时也缺乏必要的指导原则。因而建立一个刻画动态配置系统本质的动态配置模型十分必要。通过分析动态配置模型中的组成要素以及各要素间的关联关系，从而定位关键性要素，将有助于我们抓住重点，准确定位现有研究中存在的不足，从而对关键技术展开研究，最终设计并实现功能强、可靠性高、性能优良的动态配置系统。

系统一致性体系的混乱：系统一致性内容的定义以及相应一致性保证方法的设计构成了完整的系统一致性体系。对于避免性方法而言，系统一致性体系是动态配置算法设计和动态配置平台实现的依据。但是现有研究中，不同研究团体从不同角度、在不同层次上定义了系统一致性的具体内容。而这些一致性定义中有的互为补充，有的部分重叠，造成了系统一致性概念的不清晰，进而导致系统一致性体系的混乱。因此，有必要通过对现有的各种一致性定义进行深入分析，抓住本质，理清关系，从而对系统一致性内容进行科学分类和严格定义，并提出保证系统一致性的完整、有效的方法。

动态配置平台实现复杂度较大：现有各动态配置方法在实现动态配置平台时，通常要对底层的分布式平台进行较大扩展，如引入工厂对象以及各种形式的动态配置代理，复杂度较大。通过本书第三章对动态配置系统“反射本质”的分析，以及关键要素的定位，我们发现动态配置平台的实现复杂度主要来源于底层分布式平台的反射能力不足。现有动态配置方法通常基于传统的分布式对象平台，如 CORBA 和 RMI 等，实现动态配置平台。与

分布式对象平台相比，分布式构件平台，如 EJB 和 CCM 等，则具有更强的反射能力，也将极大简化动态配置平台的实现。

5.4 动态配置算法

围绕着如何保证行为一致性、引用一致性、构件状态一致性和应用状态一致性，我们对各种基本动态演化意图的实现过程进行分析，并提出相应的动态配置算法。基本动态演化意图包括：构件删除、构件添加、构件替换、构件迁移、连接删除、连接建立、连接重定向和构件属性设置。在动态配置算法的设计过程中，除了考虑系统一致性外，还要考虑针对每种动态演化意图的特点，尽量提高动态配置的性能。

5.4.1 构件删除算法

构件删除意图的目标是从运行的系统中删除一个构件。被删除的构件为目标构件。目标构件的删除可能导致事务被中断，因此需要保证行为一致性。若存在系统对构件状态的全局约束，则构件的删除还可能破坏系统的全局约束，因此构件删除意图的实施还需要保证应用状态一致性。引用一致性和构件状态一致性不会由于构件的删除而被破坏。当存在系统对构件状态的全局约束时，各种动态演化意图的实施都可能导致对全局约束的破坏，因此都需要保证应用状态一致性，我们在对后继动态演化意图的分析中就不再特意说明这点。

为了保证行为一致性，避免事务由于目标构件的删除而被中断，目标构件在被删除前必须进入静止状态。

在保证应用状态一致性时，可以通过直接设置构件属性或者利用 run-complete 机制设置构件状态。相应的系统状态设置构件应根据采用的构件状态设置方式而确定。为了保证应用状态一致性，所有的系统状态设置构件必须进入静止状态。

根据上述分析，实施构件删除意图前，必须进入静止状态的构件集合 QS（Quiescent Set）以及相应的需要进入被动状态的构件集合 CPS（configuration Change Passive Set）为：

QS={构件 n|n 为目标构件或系统状态设置构件}

CPS=∪PS(i);i∈QS

构件删除意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）调用目标构件的 run-complete 方法；

（3）删除目标构件的所有连接，以及目标构件注册的名字信息；

（4）删除目标构件；

（5）为系统状态设置构件中通过属性方式设置状态的构件设置状态；

（6）驱动<CPS－目标构件>中的所有构件从被动状态进入主动状态。

其他构件对目标构件的访问，除了通过直接持有目标构件的引用外，还可能通过在名字服务或交易服务等处的查询间接获取目标构件的引用。因此，构件注册的名字信息可被

看作一种特殊的连接信息，随着目标构件的删除、添加、替换和迁移，也应根据需要被同步删除、添加或更新。

当目标构件实为由多个构件组成的模块时，除了在具体的实现方式上与对单个构件的处理稍有不同外，以上算法同样适用。需要特别说明的是，这种情况下若构件间存在状态依赖关系，在执行算法的第二步时，需要遵循特定的顺序，依次调用组成目标构件的各构件的 run-complete 方法，才能正确设置构件状态。

5.4.2　构件添加算法

构件添加意图的目标是向正在运行的系统中加入一个新构件，从而实现容错、负载均衡或新功能的加入。新加入的构件为目标构件。目标构件可能需要用系统中某构件的状态初始化，因此需要保证构件状态一致性。若存在系统对状态的全局约束，还需要保证应用状态一致性。另外，行为一致性也需要保证，因为目标构件的加入可能为正在运行的事务加入新的行为单元，从而破坏事务完整性。

为了保证行为一致性，避免目标构件中途加入事务的情况，目标构件在被添加到系统后必须处于静止状态，因此，在构件添加前，应根据添加构件后的系统，计算出目标构件的 PS，并驱动其中的构件进入被动状态。目标构件作为新创建的构件，在 configuration-complete 方法被调用之前自然处于被动状态。这是因为构件虽然是一旦被创建并连接到系统中后就可以接收到由系统中其他构件发来的请求，但是却只有在它的 configuration-complete 方法被调用后，它才能开始向外发出请求，启动需要其他构件参与的事务，从而从部署状态进入正常运行状态。因此，可以认为构件在进入正常运行状态前自然处于被动状态。当然构件此时也可以启动本地行为，并非处于绝对的被动状态，但这些行为的存在对于考虑如何保证系统一致性并设计动态配置算法并没有影响。

为了保证构件状态一致性，状态传递的双方构件都必须进入静止状态。对于构件添加意图来说，接收状态的构件为目标构件。

在保证应用状态一致性时，可以通过直接设置构件属性或者利用 configuration-complete 机制设置构件状态。相应的系统状态设置构件应根据采用的构件状态设置方式而确定。为了保证应用状态一致性，所有的系统状态设置构件必须进入静止状态。

根据上述分析，实施构件添加意图前，必须进入静止状态的构件集合 QS 以及相应的需要进入被动状态的构件集合 CPS 为：

QS＝{构件 n|n 为目标构件，或状态传递的源构件，或系统状态设置构件}

CPS = ∪PS(i)- {T} ; i ∈ QS, T 为目标构件

构件添加意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）创建目标构件；

（3）构件状态传递：用源构件收集的状态，初始化目标构件；

（4）为系统状态设置构件中通过属性方式设置状态的构件设置状态；

（5）建立目标构件的连接；

（6）调用目标构件的 configuration-complete 方法；

（7）驱动<CPS－目标构件>中的所有构件从被动状态进入主动状态；

（8）注册目标构件的名字信息。

当目标构件实为由多个构件组成的模块时，除了在具体的实现方式上与对单个构件的处理稍有不同外，以上算法同样适用。此时，目标构件的创建过程实际就是一个应用组装的部署过程：模块内的构件被创建、属性被配置，构件间的连接被建立。而模块将作为一个整体的目标构件，在算法第五步，与系统内的其他构件建立连接关系。

5.4.3 构件替换算法

构件替换意图的目标是在系统运行时刻用一个新创建的构件替换系统中原有的构件。新旧构件具有相同的接口，但可能具有不同的实现或物理位置。系统中原有的构件被称为旧目标构件，新创建的构件被称为新目标构件。旧目标构件的删除，可能破坏事务完整性，因此构件替换意图的实现需要保证行为一致性。新旧目标构件之间可能需要状态传递，因此需要保证构件状态一致性。构件替换后，新目标构件的引用与旧目标构件的不同，因此需要保证引用一致性。若存在系统对状态的全局约束，则还需要保证应用状态一致性。

在实现构件删除意图时，要求构件必须处于静止状态，从而保证事务的完整性。旧目标构件参与的事务被分为两类：一类是替换前可被阻塞的事务，一类是替换前必须完成的事务。大多数动态配置方法在替换构件时，都忽略了这个问题，而是直接阻塞到达旧目标构件的所有事务并延迟到动态配置结束后由新目标构件完成。显然这种方法将在某些动态配置场景下破坏行为一致性。但是，一个事务是否可被阻塞完全由应用语义隐含决定，无法从外界判定。为了严格保证行为一致性，我们要求在构件替换前，旧目标构件参与的所有事务必须全部完成，旧目标构件必须进入静止状态。

为了保证构件状态一致性，进行状态传递的新、旧目标构件必须进入静止状态。假设新、旧目标构件分别为 Told、Tnew。为了保证行为一致性，Told 必须在构件替换前进入静止状态。这意味着 PS(Told)中的所有构件都处于被动状态。对于构件替换来说 PS(Told)-{Told}=PS(Tnew)-{Tnew}。新创建的构件 Tnew 在尚未进入正常运行阶段之前自然处于被动状态，而此时若 PS(Told)-{Told}中的构件也未被恢复到主动状态时，显然 PS(Tnew)中的所有构件都处于被动状态，因此 Tnew 处于静止状态。

在保证应用状态一致性时，可以通过直接设置构件属性或者利用 configuration-complete 和 run-complete 机制设置构件状态。相应的系统状态设置构件应根据采用的构件状态设置方式而确定。为了保证应用状态一致性，所有的系统状态设置构件必须进入静止状态。

为了保证引用一致性，我们利用 CORBA 的 Location Forward 机制实现请求的重定向。

根据上述分析，实施构件替换意图前，必须进入静止状态的构件集合 QS 以及相应的需要进入被动状态的构件集合 CPS 为：

QS ={构件 n|n 为旧目标构件，或系统状态设置构件}

$CPS=\cup PS(i);i\in QS$

构件替换意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）创建新目标构件；
（3）构件状态传递：用旧目标构件收集的状态，初始化新目标构件；
（4）调用旧目标构件 run-complete 方法；
（5）删除旧目标构件的连接；
（6）删除旧目标构件；
（7）建立新目标构件的连接；
（8）更新目标构件注册的名字信息；
（9）为系统状态设置构件中通过属性方式设置状态的构件设置状态；
（10）请求重定向：将到达旧目标构件的请求重定向到新目标构件；
（11）调用新目标构件的 configuration-complete 方法；
（12）驱动<CPS－旧目标构件>中的所有构件从被动状态进入主动状态。

构件的连接关系体现了构件的交互关系，但构件的连接关系与交互关系之间并不是绝对的一一对应。当旧构件已被删除，系统中的其他构件仍使用旧连接关系发送请求时可能就会导致系统异常的出现，因此在建立到达新目标构件的连接关系后，需要将可能被发送到旧目标构件的请求重定向到新目标构件。

在以模块为单位实现构件替换意图时，除了在具体的实现方式上与对单个构件的处理稍有不同外，以上算法同样适用。

以模块为单位的替换可用于实现构件接口的替换。当构件接口发生变化时，可能需要改变其相邻构件的实现，也可能需要增加或删除相邻构件。这种影响还可能继续波及更多的构件。所有这些被影响的构件构成了一个需要被替换的构件模块，由动态配置者指定。在通过替换模块实现对构件接口的替换时，不管模块到底包含多少构件，都必须满足构件替换意图实现算法提出的前提：若将模块抽象为一个构件，则通过替换改变的仅是构件实现，并不改变构件接口。

5.4.4　构件迁移算法

构件迁移意图的目标是在系统运行时刻改变构件被部署的物理位置，从而实现负载均衡等目标。迁移前的构件为旧目标构件，迁移后的构件为新目标构件。构件迁移是构件替换的特例，特殊的地方在于新、旧目标构件是同一个构件，两者除物理位置不同外完全相同。因此，构件迁移也需要保证行为一致性、构件状态一致性、引用一致性和应用状态一致性。其中对引用一致性和应用状态一致性的分析和保证方法与构件替换意图一样，我们在本节就不再重复。

在构件迁移中，旧目标构件参与的事务也被分为两类：迁移前可被阻塞的事务和迁移前必须完成的事务。与构件替换不同的是，构件迁移的语义特殊性使我们能够从外界判定哪些事务可被阻塞。只要一个事务当前执行的状态可被完整、正确地收集，就可以将其暂停，并由新目标构件继续执行，并保证不会破坏事务的完整性。从事务启动者的角度出发，旧目标构件参与的事务可被分为两类，构件自行启动的事务和构件参与的由其他构件启动的事务。构件自行启动的事务必须在旧目标构件迁移前完成，旧目标构件只要进入被动状态即可满足这个要求。构件参与事务的行为表现为若干次请求响应过程。旧目标构件参与

的由其他构件启动的事务在构件迁移前是可被阻塞的，但条件是必须保证单次请求响应过程的完整性。对事务的阻塞具体表现为阻塞到达构件的请求。

根据上述分析，目标构件在迁移前必须进入如下目标状态：目标构件处于被动状态，目标构件响应完毕已接收的请求，后继到达目标构件的请求被阻塞。需要强调的是，通常情况下对请求的阻塞必须是有选择的。原因在于，完全阻塞到达目标构件的请求，可能导致直接或间接来自目标构件的请求被阻塞，进而导致目标构件无法进入目标状态。在某些应用中，目标构件的请求是可重入的。最简单的请求重入情况是，目标构件 A 向构件 B 发送请求，B 为响应该请求又向 A 发送请求。显然，阻塞目标构件 A 的重入请求，将导致无法进入目标状态。除此之外，同时对一组构件实施动态配置时，如果在目标构件 A 处阻塞了直接或间接来自目标构件 B 的请求，则 B 将无法进入目标状态。因此，阻塞到达目标构件的请求时，必须区分并正常响应直接或间接来自其他目标构件的请求和重入请求。对于任意目标构件 A，其接收的直接或间接来自其他目标构件的请求和重入请求可能属于以下两类事务：

（1）目标构件启动的事务：目标构件 B 启动事务，事务中的请求到达构件 A。当 A=B 时，为重入请求。

（2）目标构件参与的由非目标构件启动的事务：非目标构件 C 启动事务，事务中的请求到达目标构件 B，B 为了响应该请求直接或间接向构件 A 发送请求。当 A=B 时，为重入请求。

启动上述两类事务的构件构成了集合 NBS（Not Blocking Set）。驱动 NBS 中的所有构件进入被动状态后，任意目标构件都不会接收到直接或间接来自其他目标构件的请求以及重入请求。此时阻塞所有到达目标构件的请求，不会对目标构件进入静止状态造成任何影响。

需要进入目标状态被迁移的目标构件构成了集合 TS（Target Set）。虽然并非所有目标构件启动的事务都符合第一类事务的要求，但考虑到所有目标构件都必须进入被动状态，因此本节扩展了 NBS，将 TS 作为 NBS 的子集。启动第二类事务的构件构成了集合 ES（Extended Set）。NBS=TS∪ES。TS 作为目标构件的集合，本身就应进入被动状态。而驱动 ES 中构件进入被动状态的目的仅在于实现请求的正确阻塞。一旦开始阻塞所有目标构件，即使 ES 中的构件再启动事务，而且事务中的请求顺序经过多个目标构件，也不会影响目标构件进入静止状态。因为，请求到达第一个目标构件时就被阻塞，不会产生直接或间接发自目标构件的请求被阻塞的现象。此时应立即将 ES 中的构件恢复到主动状态，尽可能减小动态配置对系统的影响。ES 计算算法与 PS 计算算法十分类似。我们利用隐式连接关系、显式连接关系和构件主动发送请求端口等信息构造构件端口间的连通路径。如果发现存在一条从构件 A 的主动发送请求端口出发，多次经过 TS 中目标构件的连通路径，则构件 A 属于 ES。此处我们不再给出详细的 ES 计算算法。

对于构件迁移，由于其特殊的语义，根据上述对行为一致性的分析，只要旧目标构件完成自行启动的事务，而且其参与的由其他构件启动的事务也被正确阻塞时，就可以在保证行为一致性的前提下，正确、完整地收集旧目标构件的状态并初始化新目标构件，而无需驱动旧目标构件进入静止状态。

根据上述分析，实施构件迁移意图前，必须进入静止状态的构件集合 QS 以及相应的

需要进入被动状态的构件集合 CPS 为：

QS={构件 n|n 为系统状态设置构件}

CPS=∪PS(i)∪TS∪ES;i∈QS

构件迁移意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）创建新目标构件；

（3）阻塞到达新目标构件的所有请求；

（4）连接重定向，将所有由系统中其他构件启动的到达旧目标构件的连接重定向到新目标构件，更新目标构件注册的名字信息；

（5）请求重定向，将所有到达旧目标构件的请求重定向到新目标构件；

（6）驱动 ES 中的所有构件从被动状态进入主动状态；

（7）判断并等待旧目标构件响应完毕所有请求；

（8）构件状态传递，用旧目标构件收集的状态，初始化新目标构件；

（9）调用旧目标构件的 run-complete 方法；

（10）删除旧目标构件启动的连接；

（11）删除旧目标构件；

（12）建立新目际构件启动的连接；

（13）为系统状态设置构件中通过属性方式设置状态的构件设置状态；

（14）调用新目标构件的 configuration-complete 方法；

（15）释放新目标构件处被阻塞的所有请求；

（16）驱动<CPS-TS-ES>中的所有构件从被动状态进入主动状态。

连接重定向，实际由连接删除和连接创建过程组成。在目标构件的迁移过程中，系统中的其他构件随时都可以获取连接信息并向目标构件发送请求。如果该构件在它与旧目标构件间的连接已被删除、但是与新构件的连接还未建立之时试图去获取连接信息，将引发异常，破坏行为一致性。因此，连接重定向强调的是必须保证由连接删除和连接建立构成的整个连接更新过程的原子性。

在构件迁移过程中，发向旧目标构件的请求必须被重定向到新目标构件。虽然利用 CORBA 的 Location Forward 机制可以实现透明的请求重定向，但每个请求都要经过请求发送－获取重定向异常－请求再发送－获取应答的过程，大量请求的重定向将是一笔很大的性能开销。为了尽量减少请求重定向的发生，尽早把请求正确定向到新目标构件，从而提高动态配置的性能，我们通过阻塞所有到达新目标构件的请求，重定向到达旧目标构件的连接，并重定向所有到达旧目标构件的请求，实现了对事务的阻塞，而不是将请求先阻塞在旧目标构件处，最后再一起重定向到新目标构件。

由于连接重定向和请求重定向的存在，对于所有到达新目标构件的请求来说，新目标构件响应请求的顺序与这些请求被发出的顺序并不相符。但这并不会破坏事务完整性。一个事务中，顺序启动的若干次请求响应过程组成了一个行为序列。如果行为序列采用同步通讯方式，被重定向的请求的后继请求只有在该请求被成功响应后才能启动，因此行为序列的顺序并没有被破坏，事务完整性得到保证。如果行为序列采用异步通讯方式，同样也不会破坏事务完整性。因为即使在正常运行环境中，中间件也不承诺先发的请求必定会先

被响应，因此在这种情况下破坏行为序列的顺序是被事务语义所允许的。

在基于 CCM 实现的配置管理平台中，构件除了接收到来自其他构件的请求外，还可能接收到来自 DRM 和部署基础设施的请求。我们将前者称为应用类型请求，将后者称为配置类型请求。在一个存在边界问题的系统中，应用类型请求还可被细分为来自外部实体的请求和来自内部实体的请求。在构件迁移意图实现算法中，无论是阻塞请求、重定向请求还是检测构件是否响应完毕所有请求都是指应用类型请求。配置类型请求在任何时候都被正常响应，从而保证各构件可在 DRM 的统一协调和控制下完成动态演化意图。

5.4.5 连接建立算法

连接建立意图的目标是在系统运行时刻建立一条新的连接。新建立的连接为目标连接，启动目标连接的构件为目标构件。类似于构件添加意图，目标连接的加入也可能为正在运行的事务加入新的行为单元，从而破坏事务完整性。因此连接建立需要保证行为一致性和应用状态一致性。

为了保证行为一致性，目标构件在连接建立前必须进入静止状态。

连接建立时，由于没有构件的离开和加入，因此所有系统状态设置构件都是通过设置属性的方式设置状态。为了保证应用状态一致性，系统状态设置构件在设置状态前也必须进入静止状态。

根据上述分析，实施连接建立意图前，必须进入静止状态的构件集合 QS 以及相应的需要进入被动状态的构件集合 CPS 为：

QS ={构件 n|n 为目标构件，或系统状态设置构件}

$CPS=\cup PS(i);\quad i\in QS$

连接意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）创建目标连接；

（3）通过设置属性的方式为系统状态设置构件设置状态；

（4）驱动 CPS 中的所有构件从被动状态进入主动状态。

同时建立多条连接时，以上算法同样适用。

5.4.6 连接删除算法

连接删除意图的目标是在系统运行时刻删除系统中已有的连接。被删除的连接为目标连接，启动目标连接的构件为目标构件。由于连接的删除，事务可能被中断，因此连接删除意图的实现需要保证行为一致性。若存在系统对状态的全局约束，则还需保证应用状态一致性。

为了保证行为一致性，目标构件在删除连接前必须进入静止状态。

连接删除时，由于没有构件的离开和加入，因此所有系统状态设置构件都是通过设置属性的方式设置状态。为了保证应用状态一致性，系统状态设置构件在设置状态前也必须进入静止状态。

根据上述分析，实施连接删除意图前，必须进入静止状态的构件集合 QS 以及相应的

需要进入被动状态的构件集合 CPS 为：

QS={构件 n|n 为目标构件，或系统状态设置构件}

CPS=∪PS(i);i∈QS

连接删除意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）删除目标连接；

（3）通过设置属性的方式为系统状态设置构件设置状态；

（4）驱动 CPS 中的所有构件从被动状态进入主动状态。

同时删除多条连接时，以上算法同样适用。

5.4.7　连接重定向算法

连接重定向意图的目标是在系统运行时刻改变系统中已有连接的目的端，从而实现负载均衡等目标。重定向前的连接被称为旧目标连接，连接的目的端为旧目的端构件。重定向后的连接被称为新目标连接，连接的目的端为新目的端构件。启动新、旧目标连接的构件被称为目标构件。在连接的重定向过程中，旧目标连接将被删除，可能导致事务的中断，因此连接重定向意图的实施需要保证行为一致性。若存在系统对状态的全局约束，则还需要保证应用状态一致性。另外，由于构件间的连接关系和交互关系并不存在绝对的一一对应关系，虽然连接被重定向了，但目标构件仍然可能向旧目的端构件发送请求，违背了连接重定向的语义。因此连接重定向意图的实施还需要保证引用一致性，将到达旧目的端构件的请求重定向到新目的端构件。

虽然连接将被重定向到新目的端构件，某些由于旧目标连接的删除而被中断的事务可由新目的端构件接替完成。但是类似于构件替换，我们无法界定哪些事务可被新目的端构件接替完成。因此，目标构件在连接重定向前必须处于静止状态，从而保证所有可能需要旧目标连接的事务全部完成，进而严格保证行为一致性。

连接重定向时，由于没有构件的离开和加入，因此所有系统状态设置构件都是通过设置属性的方式设置状态。为了保证应用状态一致性，系统状态设置构件在设置状态前也必须进入静止状态。

根据上述分析，实施连接重定向意图前，必须进入静止状态的构件集合 QS 以及相应的需要进入被动状态的构件集合 CPS 为：

Qs={构件 n|n 为目标构件，或系统状态设置构件}

CPS=∪PS(i);　i∈QS

连接重定向意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）连接重定向：将目标构件启动的到达旧目的端构件的连接重定向到新目的端构件；

（3）请求重定向：将所有到达旧目的端构件的请求重定向到新目的端构件；

（4）通过设置属性的方式为系统状态设置构件设置状态；

（5）驱动 CPS 中的所有构件从被动状态进入主动状态。

同时重定向多条连接时，以上算法同样适用。

5.4.8 构件属性设置算法

构件属性设置意图的目标是在系统运行时刻为系统中的构件设置属性，从而改变其运行特性。被设置属性的构件为目标构件。构件的状态影响构件的行为。对构件属性的设置可能导致构件行为的改变。目标构件在属性设置前开始参与的某些事务，可能无法被属性设置后的构件继续完成，从而事务的完整性被破坏。因此构件属性设置意图的实现需要保证行为一致性和应用状态一致性。

为了严格满足行为一致性约束，目标构件在属性设置前必须处于静止状态。

设置构件属性时，由于没有构件的离开和加入，因此所有系统状态设置构件都是通过设置属性的方式设置状态。为了保证应用状态一致性，系统状态设置构件在设置状态前也必须进入静止状态。

根据上述分析，实施构件属性设置意图前，必须进入静止状态的构件集合 QS 以及相应的需要进入被动状态的构件集合 CPS 为：

QS = {构件 n|n 为目标构件，或系统状态设置构件}

$CPS=\cup PS(i);\quad i\in QS$

构件属性设置意图实现算法描述如下：

（1）驱动 CPS 中的所有构件进入被动状态；

（2）设置 QS 集合中构件的属性；

（3）驱动 CPS 集合中的所有构件从被动状态进入主动状态。

同时设置多个构件的属性时，以上算法显然同样适用。

第 6 章　基于反射的动态演化

6.1　反　　射

6.1.1　背景、概念和特征

所有的反射系统都以开放实现为目的。软件工程中的传统观点是通过抽象来处理复杂性的，在这种方式中，实现细节由于特定的抽象而对用户屏蔽。相应结果就是应用组件的开发是典型的“黑箱”模式，这种方式常被认为可以提高代码的重用性。而事实上，常常不可能或也不期望对用户屏蔽所有实现细节。其中根本问题在于：要隐藏实现细节就必须替应用确定（支持应用的系统）实现策略。但是，应用可能会包含一些极重要的信息，如使用某特定模块系统才能更有效地支持应用。开放实现的目的就是通过暴露模块的实现细节来解决这个问题。不过必须有原则地区分模块提供的功能和模块的内部实现。前者被称为一个模块的基接口（base-interface），后者为元接口（meta-interface）。开放（openness）是一个“反射”系统的必要但非充分条件。

反射，抽象地说，是系统的一种推理（reason about）和作用于（act upon）自身的能力。下面通过例子来阐明上述概念。如果一种解释型语言的运行解释器是反射的，且设想这种解释器的 CCSR 表示了解释运行的这几个方面：方法分派、垃圾收集和类装载。那么通过操纵 CCSR 的相应部分，就可以改变程序执行的语义。比如：在每个方法调用的前后插入对高度例程的调用，就可为解释器增加调试功能；可以更改垃圾收集的策略（如使之不起作用或使之支持实时性能等）；也可以在类装载器中插入额外的安全检查等。

从上面的例子中，可以看到反射是如何在运行期检查和调整系统的。通过检查可以监视到系统的当前状态（如确定当前配置的垃圾收集策略等）；而调整可以在运行期改动系统的行为以更好地适应系统当前的执行环境（如：系统是否正在调试，或运行在实时环境，或要求安全的环境等）。

因果相连的自表示，在两个处理“层”之间创建了连接（反射连接）——“基”层，是给定系统的传统计算领域；“元”层，它的计算领域就是系统自身。显然，反射是达到开放实现的一种途径，不过反射最初是独立地发展起来的。

反射语言或系统的主要目的是提供一种有针对性的、有控制的方式以访问下层实现。这用于检查和调整。

（1）检查。反射可以用来检查一种语言或一个系统的内部行为。通过暴露其下层实现，可以很直接地插入模块来监视系统实现，例如可以用来提供具有更好移植性的调试工具（即调试器利用统一的元接口来访问实现细节）。这种方式也可以用来实现其他功能，如性能监

视器、QoS 监视器和记账系统。

（2）调整。反射也可用来改变语言或系统的内部行为，既可以改动一个已存在特性的解释（通过修改或替代），也可以增添新的特性。前者的例子包括：替换消息传输的方式以在无线连接的情况下实现性能优化、若资源不足会降低视频传输的服务质量等。后者的例子包括：在一个运行系统中引入新的分布透明性（如并发、失败透明等）、插入一个过滤对象以减少通信流的带宽要求等。

采用反射技术，反射系统的设计和应用程序的开发都有赖于反射的基本特性：透明性、关注分离、可见性、反射粒度等。

（1）透明性。透明性是指反射系统在逻辑上可以看作是一种多层结构（反射塔），每一层的实体相对其上层的实体而言是独立的。这样就可以透明地对低层进行检查和调整。当将元层与基层集成时，对基层代码的改动程度可以衡量这个反射系统的透明度。

（2）关注分离。反射塔的不同层关注系统的不同方面，即基层执行系统的功能，其他层扩展（由其下分层组成的）系统的不同非功能性方面，比如容错、并发、持久性等。这样，利用反射就可以容易地扩展一个计算系统。将系统的不同方面（功能性属性或各种非功能性属性）委派给不同的分层，就称为关注分离。

（3）可见性。可见性表示计算的范围，即哪些基层实体或基层实体哪些方面涉及了元计算。

（4）反射粒度。反射粒度指一个反射系统中，被不同元实体具体化的基层实体最小的方面。通常的粒度层次是：类、对象、方法、方法调用等。如果反射粒度是在方法这个层次，则同一对象的两个方法可以被两个不同的元实体具体化，从而表现出两种不同的元行为。小的反射粒度使软件系统有更好的灵活性和模块性，但元实体的数量也会激增。

用反射也存在一些缺点。首先，反射可能会导致额外的性能开销。特别是需要额外的代码去确定系统行为的精确解释（因为这些行为会发生变化），因此需要更好地协调灵活性和性能之间的矛盾。另外一个缺点是在允许程序员可以访问下层系统实现的同时，必须保证系统的完整性和一致性。还有，使用反射可能会影响系统的安全性，即要防止源程序对系统不经意的破坏和恶意的访问。

6.1.2 反射的分类

可以从不同的角度对反射进行分类。

根据系统的哪些方面可由元实体检查和调整，可以将反射分为结构反射和行为反射。这两类反射并不是相互排斥的，不过并不是所有的编程语言都有必需的特性，而能以简单的方式支持这两种类型的反射。需要指出的是，这种分类是针对反射的编程语言而言的。

（1）结构反射。结构反射可以定义为语言的这样一种性能：对正在执行的程序和它的抽象数据类型进行了完全的具体化。

结构反射允许检查和调整计算系统的代码。最初的功能性语言（如 Lisp）和逻辑语言（如 prolog）都有一些声明语句去改动程序的表达，这主要基于这些语言的解释特性，解释型语言较容易引入结构反射。但大部分面向对象的语言是编译型的，在运行时没有代码的表示。只有纯面向对象的语言，它们在运行时有代码的表示，正是这些表示可用于实现

结构反射。结构反射在非纯面向对象语言中是以下述方式实现的，但都有一定的局限性，如在编译时 OpenC++或在运行时引入代表（具体化）程序代码的数据结构。前一种方式中所有的结构反射都是静态的，后一种方式的局限性取决于代码的哪些方面被具体化了。

（2）行为反射。行为反射是指语言提供了对其语义和对用于执行当前程序的数据的完全具体化。行为反射允许检查和调整一个系统运行时的行为。通常对编程人员屏蔽的方面被开放出来，比如：代表方法分派机制的数据结构、对象存储的方法等。

行为反射不同于结构反射之处在于，通过执行程序可以改变下层系统某些方面。而结构反射只能增加或改动代码段，而不能改动执行引擎、提供行为反射语言等。

根据系统自表示的不同特性，可以将反射分为过程性反射和说明性反射。

（3）过程性反射。在过程性反射中，系统的自表示是由实现系统的程序给出的。这样，自表示与系统之间的一致性自动地得到保证，因为自表示实际上就是用来实现系统的，因果相连不再是问题。换句话说，这个表示既用于实现系统，又用于推理系统。

（4）说明性反射。另一种自表示仅包括有关系统的说明，而不是系统的实现。这些说明，比如陈述了系统计算必须满足的时间和空间要求。这种自表示不再是系统的过程性表示，而是系统的状态和行为必须满足的一组约束的集合。

说明性反射比过程性反射更抽象，它关心的是系统的实现达到的目标，而不是如何达到的。不足之处在于因果相连的要求较难实现。

6.2　反 射 系 统

6.2.1　反射系统的概念

使用反射技术构造的系统被称为反射系统。反射系统提供了关于自身行为的表示，这种表示可以被检查和调整，且与它所描述的系统行为是因果相连的。因果相连，意味着对自表示的改动将立即反映在系统的实际状态和行为中，反之亦然。因此可以简单地说，反射系统是一种可通过因果相连的途径来推理其自身的计算系统。

在反射系统的实现中一般采用“关注分离”的原则，即系统分为基层和元层。基层一般用于对具体问题领域的抽象，元层则是对基层及系统内部的表示，并且与其因果相连。在基层与元层的相互关系中也涉及具体化与反射的过程。这里的具体化是指将基层的结构行为、系统内部状态表述成可获取元数据的过程。而反射是指通过这样的可获取元数据来观察和调整系统内部及基层相关部分结构行为的过程。一般说来反射系统的实现需解决如下方面的问题：

（1）如何实现具体化。具体化的实现是反射系统实现反射的前提和基础。在反射系统中元层须能将系统自身及基层的结构、行为表述成相关的数据信息，即能以恰当的方式表示出元数据，以供客户应用、程序观察和修改，以及元层进行推理和计算使用。目前的反射系统较多地采用了元模型、元接口等方式实现对基层的具体化。

（2）如何实现元层的推理计算。对元层中元数据的推理、计算是实现反射计算的重要过

程。一般说来有两种方式：一种是通过启动特定的反射过程，让元层的元解释器执行反射代码来实施相关的推理计算；另一种方式则是在相关应用方法调用中包含用于反射所需的元代码，而并不提供特制反射过程。这样可对基层计算实施连续的影响。这两种方式中，前者的优点在于对系统状态的观察、一些精确动作的实施等方面。后者的优点在于对客户应用而言，其对语义方面的反射效果较为持久，且不需要特别的元解释器和相关的反射过程。

如何通过因果相连的元数据来实现对基层的观察和修改？结构反射通过改变基层运行行为的方式来实现，行为反射则通过改变基层功能结构的方式来实现。目前已实现的反射系统较多地采用了后一种方式，对于前者，由于涉及系统运行的环境状态信息及基层运行的语义，因此实现上较难。

6.2.2 面向对象的反射系统

大多数反射语言和系统的研究都采用了面向对象的计算模型。Kiczale 等指出，在反射和面向对象的计算之间，有一种重要的协同性："反射技术使开放一种语言或一个系统的实现成为可能，但不应暴露不必要的实现细节或影响可移植性；而面向对象技术允许语言或系统实现的模型能被局部和渐进性地调整"。而且，鉴于面向对象的概念在分布式系统中特别是中间件中的重要性，本节集中讨论面向对象的语言和系统的反射问题，称之为面向对象的反射。

另外，反射的应用领域，也从最初的编程语言拓展到操作系统、窗口系统、分布式系统等领域。这些系统中采用的反射机制与前述的 3－Lisp 和 CLOS 不同，但都保留了反射的本质要素——显式的自表示。下面从一般的（而不是特定于编程语言）角度，进一步描述反射和相关问题。

对象本身也可用其他对象来表现，后者又称为元对象。元对象的计算（又称为元计算）用于观测和修改它们的指示物（即所代表的对象）。元计算常常是首先捕获它们的指示物，然后由元对象执行正常的计算。换句话说，指示物的行为被元对象所捕获，后者执行元计算，要么替换，要么封装指示物的行为。当然，元对象自身也可由其他对象来表现，即它们是二重元对象的指示物。以此类推，一个反射系统可以是一种多层结构，组成了一个反射塔。在基层的对象称为基对象，它们对应用领域中的实体进行计算；其他层，即元层的对象，对其相邻低层（指示层）的对象执行计算。

对象与元对象之间不一定是一一对应的关联：几个元对象可以共享一个指示物，单个元对象也可有几个指示物。反射塔的相邻层之间的接口通常称为元对象协议。值得注意的是，MOP 有时含义更广，可以泛指一个面向对象的反射系统的组织结构、机制等。可以说，MOP 是反射技术中的一个比较模糊、通用的概念。

每一次反射计算可以被分为两个逻辑部分：计算流上下文切换和元行为。计算从基层的计算流开始，当基层实体执行某个行为时，该行为被元实体捕获，同时计算流上升到元层（称之为换上操作），然后元实体执行它的元计算，当它允许基层实体执行时，计算流又返回到基层（称之为换下操作）。

在所有反射模型中，一个基本概念就是具体化。为了对相邻低层的计算执行计算，每一层维护一组支持这个计算的数据结构，即前述的因果相连的自表示，而创建这种自表示

的过程称为具体化。具体化就是原本隐式的方面显式化。当然，相邻低层的哪些方面被具体化，取决于反射模型（如结构、状态、和行为、通信等）。在任一种情况下，包含自表示的数据结构与系统被具体化的那些方面是因果相连的。在相邻层之间保持这种因果相连系统是反射基础设施的责任，而元对象的设计者和编程人员不必知道如何实现因果相连关系等细节。

反射模型的一个关键特色就是透明性。在反射这个上下文中，透明是指位于任一层的对象完全不知晓其上层对象的存在或工作情况。换句话说，每个元层被加到其指示物层上，而无需改动指示物层本身。即反射系统在元层及其指示物层之间执行因果相连，是以对元层编程人员和指示物编程人员都透明的方式实现的，称之为反射透明性。

6.2.3　反射模型

反射模型分为三类：它们分别是：元类模型、元对象模型和元通信模型。这主要是根据元实体与基层实体的关系来区分的。

1. 元类模型

类描述了它的实例（对象）的结构和行为。在元类模型中，反射塔是通过实例化关系实现的。具体化基层实体的元对象是它的类，而具体化元对象的二重元对象是它的元类（也即元对象），如图 6-1 所示。类很适于控制和改动结构信息，因为它本来就包含着这些信息。这种模型的问题在于很难单独指定某一个实体的元行为，因为同一个类的所有实体有同一个元对象，所以所有实体共享同样的元行为。

2. 元对象模型

这种模型中的元对象，是一个特殊类 MetaObject 或其子类的实例，如图 6-2 所示。反射塔是由调用关系实现的。每个元对象截取（换上操作）送往其指示物的消息，对这些消息执行元计算，然后才将它们交付（换下操作）给指示物。

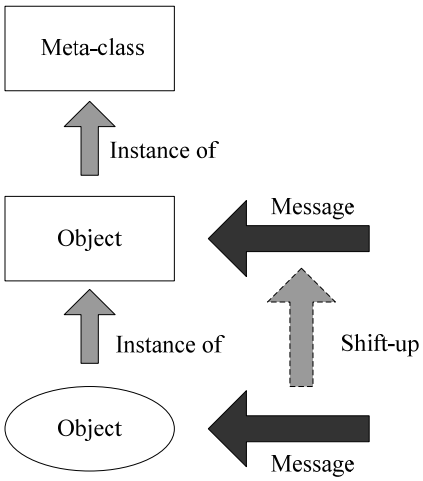


图 6-1　元类模型

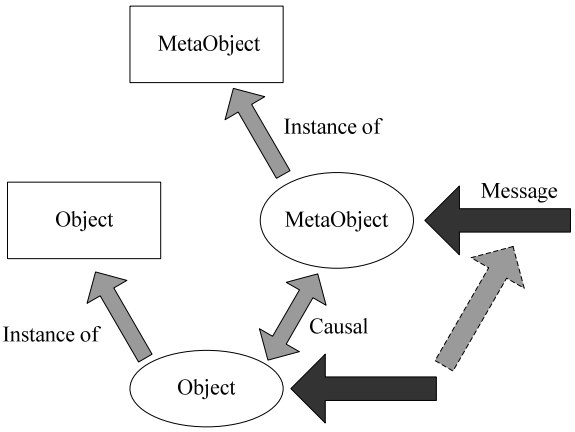


图 6-2　元对象模型

原则上，每个元对象可与多个指示物相关联，每个指示物也可有多个元对象。但是考虑到效率，在大多数实现中，元对象与对象是一一对应的。通常元对象作为基对象的一个实例变量而与对象相关联。

通过开发元对象的新的类，就可以定义新的元行为，这样就可以针对每个对象定制其元行为。元对象模型是应用最广泛的模型，其应用不仅仅局限于编程语言，还涉及操作系统（如 ApertOS）、分布式系统（如 CodA）等。

3. 元通信模型

在这个模型中，元实体是一种特殊的对象，称为消息，它们具体化了基层实体应该执行的操作，如图 6-3 所示。消息的类定义了消息执行的元行为，不同的消息可以有不同的类。每个方法调用，被具体化为一个对象，即消息，消息按照所需元计算的类型管理自身（如分派等），当元计算结束时，这个消息也被破坏。

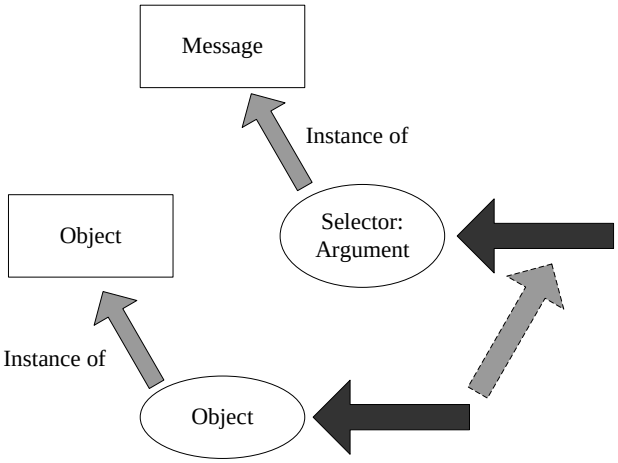


图 6-3 元通信模型

通过指定每个方法的不同类型，可定义对象执行方法调用的不同元行为。每个方法调用创建一个消息，调用完成后消息也被破坏，因此消息的生命周期包含在创建它的方法调用中，也就不可能在元计算之间保持信息。这种模型中的反射塔只包含两层：基层和元层。

6.3 反射和演化

运用反射技术可以提高系统的灵活性，以适应环境的变化和需求的变化。而且由于上述的透明性，一个系统中对象的结构和行为可以被改动、丰富或替代，而不需改动原来的系统代码。这种通过额外的元层来实现系统进化的方法，有的需要重新编译，有的可以动态地进行。

在系统设计中采用反射技术，还可以分离功能属性与非功能属性。一种典型的方式是，基层对象用来满足应用程序的功能性要求，而元层对象用于保证非功能属性（如容错、持

久性、分布性等）。参照对系统的这种划分，有时把基层对象称为功能性对象，把元层对象称为非功能性对象。功能性对象对现实世界的实体进行建模（如 Accout 代表银行账号），非功能性对象对功能性对象的属性进行建模（为反映这个情况，非功能性类常用与这些属性相对应的名字，如 Fault Tolerant Meta Object）。

这种方法的优点在于：首先关注分离增强了系统的可修改性。根据系统要修改部分涉及的是功能性属性还是非功能性属性，可以只改动功能性对象或只改动非功能性属性。比如：一个账号的数据信息改变了，（功能性的）类 Accout 就需要改动；而如果需要更好的容错性，则改动（非功能性的）类 Fault Tolerant Meta Object。

而且，这种方法还从两个方面提高了可重用性。一是同样的功能性对象（如一个 Accout 对象），既可通过与一些元对象相关联而具备某些额外属性，也可独立使用。一个对象的任何额外属性（如：容错、迁移性、持久性等）都会带来额外开销。通过反射技术，这些特性并不是包含在该对象代码中，而是被分离在其多个元对象中；当某个额外属性不需要时，相应的元对象就不必实例化。在反射方法中，一个功能性对象之所以具有持久性，是因为一个元对象透明地修改了它的行为（如：截取其构造/析构函数、增加从文件中恢复/存储到文件中等行为）。如果不采用反射，典型的方法是引入 PersistentObject 类，该类提供了将对象存入文件、再从文件中恢复等操作，其他需要持久性的类则继承 PersistentObject 类，那么它的实例就是持久性的对象了。但是这种方法将非功能属性代码与功能属性代码交织在一起，比如相反操作（通过取消与 PersistentObject 的继承关系而放弃 Accout 的持久性）就较难实现。另一种形式的重用性体现在：非功能性的属性由元对象来实现，且独立于运用它们的功能性对象。即同样的元对象可以被重用于为不同的功能性对象附加同样的非功能性属性，通过反射方式提供了容错、持久性、原子性和认证等。

6.4　反射中间件

6.4.1　背景和概念

当前的中间件，无论 CORBA、DCOM 还是 JAVA　RMI，基本上都采用了黑箱抽象的原则，缺少必要的灵活性和适应性。而中间件处理的是复杂的分布式应用问题，因而常常面对变化的运行环境和不同的用户需求。中间件已被运用到多种多样的应用环境中，如多媒体、实时、嵌入式系统、手持设备、甚至移动网络环境。

当一个便携式计算机从局域网环境移动到无线环境，可用带宽或传输出错率肯定会发生明显变化。理想上希望中间件能切换到一种节约带宽的机制，如对调用消息进行压缩。但大多数中间件不支持这种适应性，而应用受限于只能使用中间件提供的机制。

分布式环境中的不同应用常常有不同的服务质量要求（如性能、安全、可行性、负载平衡等），但中间件对此要么支持得很少，要么将各种属性交织在一起。比如 CORBA 的对象服务，每个对象服务都为核心 ORB 拓展了额外的属性，无论这些服务使用与否，ORB 必须包含对服务的数据结构和协议的支持。这样使 ORB 的实现低效，用户不得不涉及一

些事实上不需要的属性。

为了系统地运用开放实现思想和反射技术，人们开始着手反射中间件的研究和设计。所谓反射中间件，简单地说，是指通过适当的因果相连的自表示而能检查和调整其行为的中间件系统。如果把中间件看作计算系统，那么从反射系统的定义可以得到反射式中间件的定义：反射式中间件是包含自描述信息的中间件，它提供对自描述信息的访问机制，而且自描述信息和中间件的实例之间存在着因果联系，即二者始终是一致的。外界可以通过访问自描述信息来了解中间件的信息，也可以通过修改自描述信息来对中间件进行定制和调整。和传统的中间件相比，反射式中间件具有开放和灵活的特点，因而能够适应动态变化的环境。在中间件设计中运用反射技术，至少可以获得如下好处。

（1）检查系统的结构、状态和行为。例如，CORBA 的接口仓库和动态调用接口，可以使客户程序在运行时得知服务组件的类型，而动态地构造调用请求，以调用这些服务组件的操作。

（2）灵活性和适应性。反射使系统更能适应新的环境，并更好地对付各种变化。这种变化往往发生在当基于中间件的应用程序被部署到动态环境，如多媒体、组通信、实时和移动计算等环境时。

（3）关注分离。在系统设计中采用反射技术，可以分离功能性属性和非功能性属性。如前所述，典型的方式是，基层对象用来满足应用程序的功能性要求，而元层对象用于保证非功能性属性。这种方法的优点在于：首先，关注分离增强了系统的可修改性。根据系统要修改部分涉及的是功能性属性还是哪种非功能性属性，可以只改动对应的分层。而且，一种形式的重用性体现在，同样的元实体可以被重用于为不同的基层实体，附加同样的非功能性属性。

6.4.2 几个典型的反射中间件

目前已存在一些反射中间件，下面介绍几个典型的例子。

1. DynamicTAO

Illinois 大学的研究人员最早开始这方面的研究，他们开发的 DynamicTAO 是 CORBA 兼容的反射 ORB，支持运行期的重配置。DynamicTAO 对其内部结构进行具体化，维持了 ORB 内部组件及它们之间动态交互的自表示。

DynamicTAO 中的具体化是通过一组被称为组件配置器的实体来实现的。一个组件配置器维护了某个组件与系统中其他组件间的依赖关系。每个运行 DynamicTAO 的进程包含了一个被称为 Domain Configurator 的组件配置器的实例，它负责维护在这个进程中运行的 ORB 实例和侍服程序的引用。而且每个 ORB 实例包含一个定制的组件配置器——TAOConfigurator。TAOConfigurator 包含许多接口，这些接口可以看作 ORB 所使用的具体策略实现的安装点。这些策略包括：并发策略、调度策略、安全策略、连接管理策略等。图 6-4 展示了只包含一个 ORB 实例的进程的具体化机制；这些配置器也可以存储依赖于某些策略的客户请求的引用。通过这些信息，就有可能在对策略进行重配置时，保证系统的一致性。

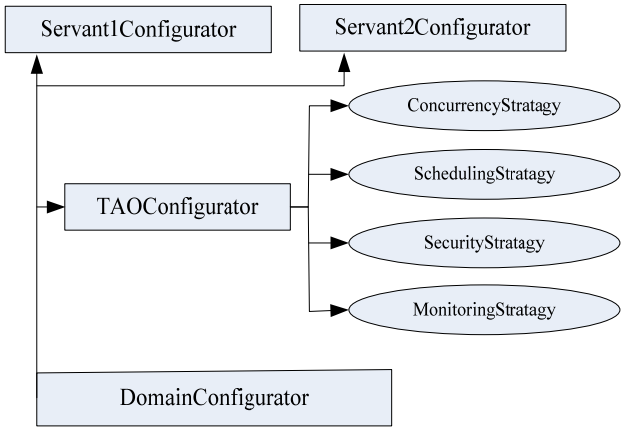


图 6-4　dynamicTAO 的具体化机制

2. OpenORB

Lancaster 大学的研究者正在为构造开放的多媒体平台而开发通用的反射结构（Open-ORB 等项目）。

中间件平台的基层被看作它所提供的一组服务。这些服务包括：对象之间的交互通信及所涉及的活动（消息的编码、分派、优先权机制、资源分配等）、支持性的服务（名字的解析和定位、接口引用的管理、分布式对象生命周期的管理等）以及其他通用的服务（如安全、事务处理、容错、交易服务等）。这些服务通常以对象的形式建模和实现，而平台维护了这些对象的配置。

元层包含了具体化这些服务配置的编程结构，并允许反射计算。Open-ORB 中的元层被细化为四个独立的元模型，即 Encapsulation、Composition、Environment、Resource 元模型。每一个元模型代表了平台的不同方面，并可以进行递归细化，如图 6-5 所示。

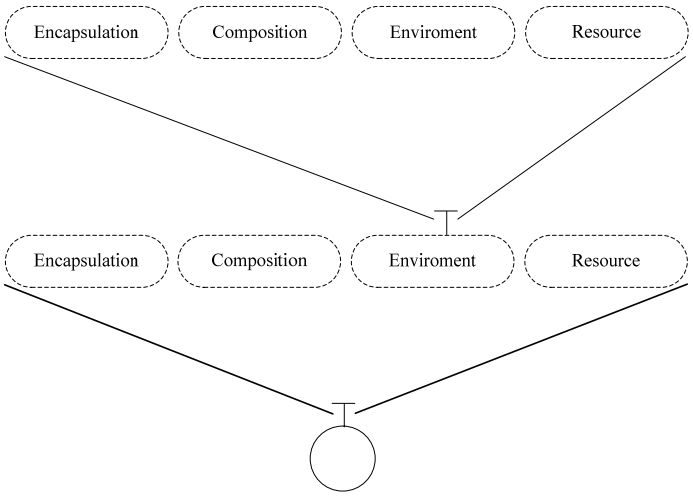


图 6-5　OpenORB 的元空间的结构

Encapsulation 元模型提供了对具体接口的关键性访问，比如这个接口包括哪些方法、接口的继承结构及其他特性。通过 Encapsulation 的 MOP 甚至可以增添或删除接口的某些元素（如方法、属性等）。

Composition 元模型提供了对复杂对象的组件配置的访问。复杂对象的内部结构被表示为对象图的形式，节点代表了组成对象，边表示这些组成对象之间的绑定。而且图中的一些对象可以是（分布式）绑定对象，从而允许创建公布的配置。Composition 的 MOP 提供了查看和操作对象图的结构的设施，并允许访问单个组件及插入和删除组件。这个元模型的重要之处在于具体化了中间件平台的配置，并可以动态地调整（如增加新的服务）。

Environment 元模型代表了平台中对象的执行环境，在分布式环境中，元模型包含消息到达、排队、分派、优先权机制等函数，以及发生方的相应函数。通过 Environment 的 MOP，可以对环境中已有机制的参数进行配置，如存放到达消息的队列的大小和优先权等；也允许添加额外的透明性，或控制这些机制的具体实现。Environment 的一个关键特色是其本身可以是一个复合对象，因此可以递归地利用 Composition 元模型检测和调整它的结构。比如，在其对象图中的某一点上插入一个 QoS 监视器。

Resource 元模型涉及资源的管理，提供了与操作系统无关的资源抽象，如线程、缓冲区等，并将这些高层抽象映射到低层实现上。通过 Resource 的 MOP，可以检查、添加或删除与某个操作关联的资源；对处理资源（线程）的调度参数进行重配置等。

以下用实例展示 OpenORB 如何利用不同的元模型去动态地调整平台所提供的服务。图 6-6 展示了一个两层的绑定对象，它为分布式应用对象之间提供了简单的连接服务。在运行期，如果监控设施检测到带宽下降，则要求对绑定对象进行重配置，以满足协商的 QoS。为此，可以在绑定的两端分别插入压缩和解压的滤波器，以减少数据流量。如图 6-6 所示，Composition 元对象维护了绑定配置的表示（对象图），Composition 的 MOP 提供了调整这个表示的操作，操作的结果将反射到绑定对象的组件的实际配置中。

另一个例子是，考虑在两个流接口之间传输音频的绑定对象。为了更好地控制音频数据流的抖动，连接到接收端的绑定对象按照 Environment 元模型进行具体化。这样，利用 Environment 元对象可以引入一个队列和分派机制，以缓冲音频帧并在恰当的时间交付给接收端，从而满足应用对抖动的要求。而且，通过 Environment 的 MOP 的还可以动态地改变队列的长度和分派算法。

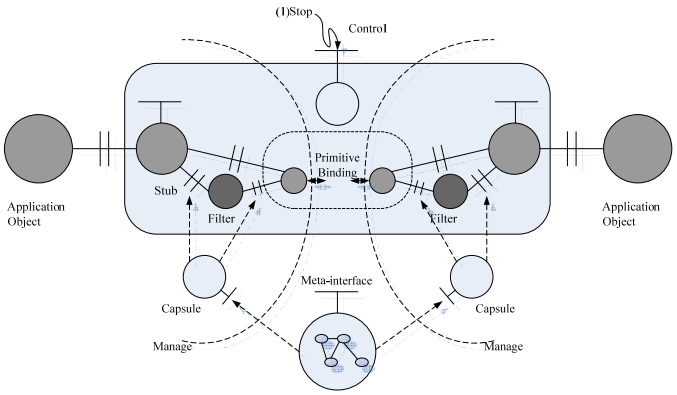


图 6-6　通过 Composition 元模型进行重配置

3. 其他

FlexiNet 是 ANSA 项目中的一部分，主要用来展示面向构件的概念可有效地构造可重配置和可扩展的中间件平台。FlexiNet 是一个基于 Java 的系统，提供了一个通用的绑定框架和组装到这个框架中的一组构件。通过适当选择组装到框架中的构件，可获得不同性能的中间件设施，支持移动对象、持久对象、安全的对象和事务性的对象等。它在协议栈的设计中采用了反射的思想，各协议层是相对独立的构件，可以方便地构成采用不同协议的通信设施。

法国 Ecole des Mines de Nante 的 OpenCorba 是一个反射的 CORBA　ORB，它主要利用反射语言 NeoClasstalk 的 MOP。NeoClasstalk 采用了元类模型，它的 metaclass 定义了关于对象创建、封装、继承规则、消息处理等属性。OpenCorba 利用语言的反射机制具体化 ORB 的内部属性，并可改动和调整运行行为，如远程调用、IDL 类型检查、IR 错误处理等。

6.4.3　中间件中的反射层

典型的中间件系统中 Stub 和 Skeleton 通常由 IDL 编译器生成，下层的通信基础设施由中间件提供。Stub 具有与服务器同样的接口，远程调用由客户调用 Stub 开始，Stub 将调用的方法名、参数等编码成适合网络传输的字节流的形式，再通过中间件通信基础设施传送到远程的服务器节点。Skeleton 对调用请求消息进行解码，得到被调用的方法名和参数，再调用服务器的相应方法。调用返回后，Skeleton 对结果进行编码，同样，中间件通信基础设施传回可互访，Stub 对调用响应消息进行解码，得到调用结果，并返回给客户。这种通信结构可以有效地在客户和服务器之间传递功能性的调用请求，但没有直接提供支持分布式应用非功能性方面的方法，除非修改应用程序或中间件。

而 RECOM 的基于反射技术的分层协议栈中，可以很方便地插入与应用程序独立的反射层，以支持分布式应用的非功能属性，如图 6-7 所示。

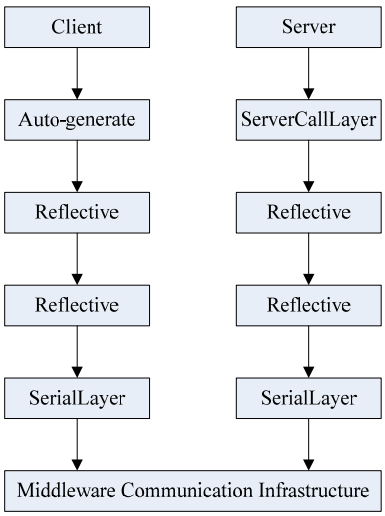


图 6-7　基于 RECOM 的分布式应用

RECOM 的 Stub 实现了与服务器相同的接口，并将具体接口相关的调用转化为通用的形式（一个 Invocation 类的实例）。与一般的中间件不同的是，它没有编码/解码的功能，编码/解码的功能是由独立的一层——SerialLayer 来完成。这样，在 Stub 和 SerialLayer 之间可以插入多个具有统一接口的反射层。同样在服务器方，ServerCallLayer 负责调用服务器的相应方法，而编码/解码的功能由 SerialLayer 完成，在 SerialLayer 和 ServerCallLayer 之间也可以插入多个具有统一接口的反射层。反射层可以独立编码，应用于不同的分布式应用程序。做到反射层截取、检查调用请求，进而执行相关的辅助操作或修改调用请求，以满足分布式应用的一些非功能性要求。

6.4.4 反射层的编程模型

反射层的编程模型，包括 Invocation 类、反射层接口和配置接口等三种抽象。

1. Invocation

第一个抽象是 Invocation 类，它代表由客户发出的远程调用，包含了调用请求在不同阶段的信息：如目标对象、被调用方法、参数和（调用后的）返回值或异常等。另外，为了方便客户方的反射层与服务方的反射层之间交换额外的数据，它还维护了一个数据栈。

```
Public final Class Invocation
{
Invocation( );
Invocation(WrappedMethod method, Object target,Object arg[]);
Object getTarget( ) ;
Void setTarget (Object target );
…
Void invoke( ) throws BadCallException
Void invoke( Object target) throws BadCallException
Void push( Class cls, Object obj);
Object pop( Class cls );
}
```

客户方的 Stub 是根据接口定义自动生成的，在运行时为每次调用实例化一个 Invocation 对象，然后传递给下一层。在服务器方，SerialLayer 对请求消息包中的信息解码，恢复这个 Invocation 对象，然后交给上一层。反射层通过 Invocation 对象，检查、改动或传递应用程序的调用。例如：Invocation 对象中的返回值，一般由 ServerCallLayer 调用实际服务对象，得到返回值后再用 setReturn Value 方法设置。而某些反射层，可能已经知道了调用（它保存了上次调用的结果），就可以直接使用 setReturn Value，并将控制返回客户，这样可以提高性能。

2. 反射层接口

第二个抽象是反射层。配置在客户方的反射层必须实现接口 CltReqRftLayer，而配置在服务器方的反射层要实现 SrvReqRftLayer 接口。

每次客户调用服务器方法时，客户方反射层的 callDown 方法就被调用。callDown 方法接收到 Invocation 对象，可以检查其中的服务对象、被调用的方法和参数。在将调用请求传递给下层之前，它可进行一些辅助性的操作或改动 Invocation 对象。然后，它要执行 i.invoke(down)，以把调用请求交给下一层。该操作返回后，反射层就假定服务器已被调用。i.invoke(down)可能返回某种异常，这表示调用链上某一层失败。反射层可以捕获这种异常，要么自己进行处理，要么简单地重新抛出。服务器方反射层的工作原理与客户方相似，在工作时它的 callUp 方法被调用，将调用请求向上传递的操作是 i.invoke(up)。

具体编写的反射层，可能还要提供一些初始化方法，以便在运用的时候输入特定交互相关的信息。

3. 配置接口

第三个抽象是配置接口。通过配置接口用户可以将反射层插入调用链，或从中删除。这是通过显式绑定协议来实现的，过程如下：

（1）客户通过显式绑定协议获得服务对象、SBM 和 CBM 的引用。

（2）通过 SBM 配置服务器方的反射层。

（3）通过 CBM 配置客户方的反射层。

（4）客户通过服务对象的引用调用服务。

4. 编程模型的特点

通过运用反射层的编程模型，可以满足分布式应用的一些非功能性要求。为此，反射层编程模型具有这样一些特点：

（1）可以捕获异常和抛出异常。因为反射层是显式地调用链中的下一层，所以它可以捕获中间件、其他反射层或服务对象的异常；它可以自己对这些异常进行处理，也可重新抛出；它自己也可能产生新的异常。

（2）可以检查并改动调用请求信息。反射层截取每个调用请求，检查其中的信息，再进行一些辅助性的操作，然后可以重新设置这些信息，如参数、甚至服务对象等。

（3）可以加入和提取额外的信息。Invocation 类维护了一个数据栈，反射层可以向其中写入和读取与方法调用不直接相关的数据，这样可以方便反射层之间交换额外的信息。

（4）可以对调用请求进行本地处理。由于对调用链中的下一层的调用是显式进行的，因此反射层也可以完全不调用下一层，而是对调用请求做本地处理后直接返回。

（5）可以访问其他中间件服务。只要反射层获得其他中间件服务的引用（如在初始化时赋予），在运行时就可以访问这个服务。

5. 两种反射层

前面介绍的编程模型是针对反射层配置在 SerialLayer 之上的情况，称之为请求反射层。RECOM 的分层协议栈和显式绑定协议为在其他位置配置反射层提供了可能，典型的是在 RPC 层与传输层之间，如图 6-8 所示，这种反射层称之为消息级反射层，因为这时候反射层截取的是已被编码成字节流形式的调用请求。消息级反射层的编程模型与请求级反射层的在原理上类似，但细节不同，如消息级反射层需要实现 MessageDown 接口（客户方）和 MessageUp 接口

（服务器方）。消息级反射层可用于对请求消息包进行加密/解密、压缩/解压缩等。

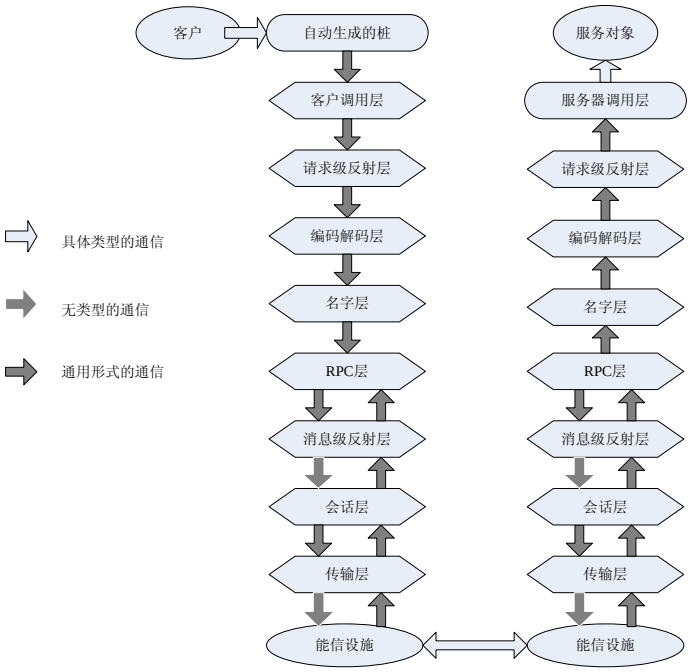


图 6-8　两种反射层

6.4.5　利用反射层实现服务定制

利用反射层的编程模型，可以对特定的客户与服务器交互的协议栈进行配置，以满足 QoS 要求。

1. 利用反射层实现高可用性

系统中硬件或软件组件的失败是不可避免的，作为高可用性的系统，应该尽可能地对客户屏蔽这种失败。为了能够达到此目的，系统应能探测到失败的发生、发现其他可替用的服务器、利用新的服务器继续操作。下面是实现此功能的客户方反射层的伪代码：

用户在初始化 RecoverRefLayer 反射层的时候，可以告知它服务的描述信息以及交易器的引用，在运行时，反射层可以利用这些信息查寻可替用的服务器。反射层的工作过程如下：代码中首先设置一个循环以限定失败后的尝试次数，因为有时可能找不到可替用的服务；如果对服务器的调用出现异常，就捕获这处异常，进而查找是否有其他可替用的服务器；如果有，就调用这个新的服务器，否则重新抛出异常；如果尝试 max_retry 次仍未调用成功，则向上层报告异常，表明这是不可恢复的失败。

2. 利用反射层实现接纳控制

接纳控制在计算机网络中用于限制网络拥堵。接纳控制可以限制某些数据包进入网络，而不影响其他数据包，因而不会降低其他用户的性能。在分布式应用中，可以采用类似方

法限制某些客户对服务器的访问，从而提高对其他客户的 QoS。接纳机制的实现需要服务器方反射层与客户方的反射层配合工作。下面是服务器方反射层的伪代码。

服务器按组将客户归类，每个组都有一个编号，客户方的反射层预先将该客户的组编号压入调用请求的数据栈。用户在初始化 SrvAdmitCtlLayer 时，输入组编号的列表及各组允许的请求。服务器方反射层的工作过程如下：首先从调用请求的数据栈中弹出该客户所属组的编号，进而计算该组当前的请求率；如果该组当前的请求率小于允许的请求率，就继续执行调用；否则，抛出 ReqRejectedException 异常。客户方的反射层捕获到这个异常之后，可以延迟请求。

在初始化客户方反射层的时候，输入客户所属组的编号及允许的最大延迟时间。客户方反射层首先将组编号压入调用请求的数据栈。然后设置一个循环，如果超过允许的最大延迟，则向前一层抛出 ReqRejectedException 异常（不能无限期地延迟）。发出调用请求，如果出现 ReqRejectedException 异常，则计算新的延迟时间（这个时间应该是递增的），然后等待这段时间后再尝试。

从以上例子中可以看出，利用反射层可以为具体的分布式应用定制特定的 QoS。而且值得注意的是，一个协议栈允许配置多个反射层（无论是客户方，还是服务器方），比如同时使用提高可用性和实现接纳控制的反射层。

6.5　基于反射理论的动态配置模型

动态配置系统反射体系如图 6-9 所示：基层是应用系统；元层是动态配置平台；动态配置系统的元控层对应于配置者、管理者；动态配置系统的结构－自省协议和行为－自省协议体现于系统信息收集部件获取系统结构及语义信息并提供给配置者、管理者以及动态配置引擎的过程中；动态配置系统的结构－调控协议和行为－调控协议体现于动态配置引擎接收来自配置者的动态演化意图并调整系统结构、改变系统行为的过程中；动态配置系统的元数据是系统信息库中的系统结构和语义信息以及系统信息收集部件实时收集的构件状态信息。

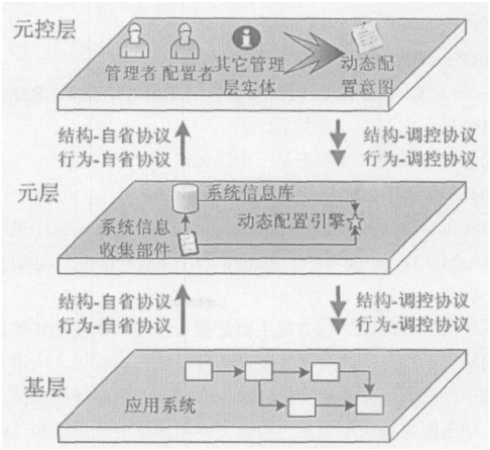


图 6-9　动态配置系统反射体系示意图

在对动态配置系统体现出的各个反射系统的要素进行深入细致的分析后，下面介绍一个反射式动态配置模型 RDRM。

定义 1 反射式动态配置模型 RDRM 是一个三元组：

RDRM=<DRSMA,DRSMD,DRSMP>

其中 DRSMA（Dynamic Reconfiguration System MA）是动态配置系统的元结构，DRSMD（Dynamic Reconfiguration System MD）是动态配置系统的元数据，DRSMP（Dynamic Reconfiguration System MP）是动态配置系统的元协议。

从宏观上而言，RDRM 模型就是一个经典的反射系统模型。从微观上检视，RDRM 模型中的诸要素体现了很强的动态配置系统所特有的特征和性质。因此，必须通过对各要素详细建模来充分描绘这些特性。

定义 2 动态配置系统的元结构是一个三元组：

DRSMA=<TAS,DRP,Reconfigurator>

其中，TAS(Target Application System)是被动态配置的目标应用系统。DRP(Dynamic Reconfiguration Platform)是实施动态配置的动态配置平台。Reconfigurator 是配置者，负责监控 TAS、产生动态演化意图，并要求 DRP 予以实施。TAS, DRP 和 Reconfigurator 分别为动态配置反射体系的基层、元层和元控层。

动态配置的研究现状表明，在构建动态配置平台时，目前通常的做法是基于一个基础平台，如 CORBA, RMI, J2EE 等，并为实现动态配置功能对基础平台进行一定的扩展。

定义 3 动态配置平台是一个二元组：

DRP=<BP, DRFE>

其中，BP 是基础平台（Base Platform），DRFE（Dynamic Reconfiguration Function Extension）是对 BP 的动态配置功能扩展。

定义 4 动态配置系统的元数据是一个二元组：

DRSMD=<DRSStrucMD, DRSSemaMD>

其中，DRSStrucMD（Dynamic Reconfiguration System StrucMD）是动态配置系统的元结构数据，DRSSemaMD（Dynamic Reconfiguration System SemaMD）是动态配置系统的元语义数据。

动态配置系统所关注的元结构数据主要是指目标应用系统的软件体系结构信息，包括组成应用系统的构件、构件间的连接关系及构件的物理位置等。应用系统的结构并非一成不变。很多系统中存在工厂对象或提供对象生命周期管理服务，支持对系统中对象的创建和删除。另外，动态配置的实施显然也将改变应用系统的体系结构。因此，应用系统的结构信息应包含静态体系结构信息和动态体系结构信息两部分的内容。应用系统在运行前的设计、部署信息为静态体系结构信息，应用系统在运行后的结构更改信息为动态体系结构信息。只有元层的动态配置平台完整地获取基层应用系统的静态及动态体系结构信息，才能支持元控层的配置者准确描述动态演化意图。在此基础上，动态配置平台将通过判定构件的依赖连接关系，并根据元语义数据，正确实施对应用系统的配置调整。

定义 5 动态配置系统的元结构数据是一个二元组：

DRSStrucMD=<StaticStrucMD,DynStrucMD>

其中，StaticStrucMD（Static StrucMD）为 TAS 的静态体系结构元数据，DynStrucMD

（Dynamic StrucMD）为 TAS 的动态体系结构元数据。

动态配置系统所关注的元语义数据主要是指应用系统中构件的状态信息以及构件的行为信息。例如，构件是否完成正在响应的请求的信息属于构件状态信息，构件是否可以主动开启事务或发送请求则属于构件的行为信息。我们将在后继章节中对这些语义信息进行进一步的解释。对于一个具体的动态配置系统而言，到底需要哪些语义信息，取决于其提供的动态配置功能，对系统一致性约束的认定和对动态配置算法的设计。

定义 6　动态配置系统的元语义数据是一个二元组：

DRSSemaMD = {StateMD,BehavMD}

其中，StateMD（State MD）为 TAS 中构件的状态语义元数据，BehavMD（Behavioral MD）为 TAS 中构件的行为语义元数据。

在动态配置系统中，通过实施动态演化意图可以改变应用系统的结构，如添加、删除构件，也可以改变构件的运行行为，如对构件属性的设置。无论是改变系统结构还是行为，动态配置平台都将一视同仁，利用元结构数据和元语义数据，在驱动并判定系统到达动态配置安全状态后，再实施具体的动态演化意图。因此，对于动态配置系统而言，区分对系统的调整到底在结构方面还是行为方面，并无意义，反而使问题复杂化。另外两者之间也并非有明显的界限，对系统结构的调整必然改变系统的行为。因此，我们在元协议中，不对结构－调控协议和行为－调控协议进行特别区分。动态配置平台接收动态演化意图，并实施对系统配置调整的过程，就是反射系统中调控协议的执行过程。

定义 7　动态配置系统的元协议是一个三元组：

DRSMP=<DRSStrucIntroProtocol,DRSBehavIntroProtocol, DRSInterProtocol>

其中，DRSStrucIntroProtocol（Dynamic Reconfiguration System StrucIntroProtocol）是动态配置系统的结构－自省协议，DRSBehavIntroProtocol（Dynamic Reconfiguration SystemBehavIntroProtocol）是动态配置系统行为的－自省协议，DRSInterProtocol（Dynamic Reconfiguration System Intercessory Protocol）是动态配置系统的调控协议。动态配置平台通过执行 DRSStrucIntroProtocol 和 DRSBehavIntroProtocol 获得 DRSMD，通过执行 DRSInterProtocol 实现动态配置功能。

动态配置系统在 DRSInterProtocol 协议的执行过程中，必须保证对系统的调控不会导致系统进入异常状态，这就是系统一致性约束。不同的 DRSInterProtocol 协议，在不同的系统一致性约束下，对系统调控的具体内容不同，执行的过程也不同。

定义 8. 动态配置系统的调控协议是一个三元组:

DRSInterProtocol=<SCC,DRF,DR.A>

其中，SCC（System Consistency Constraint）是 DRSInterProtocol 必须满足的正确性约束，具体表现为动态配置必须保证的系统一致性。DRF（Dynamic Reconfiguration Function）是 DRSInterProtocol 对系统的调控能力，具体表现为动态配置平台提供的动态配置功能。DRA（Dynamic Reconfiguration Algorithm）为 DRSInterProtocol 的具体内容，表现为动态配置平台在满足正确性约束的前提下实现动态配置功能的动态配置算法。

根据上述定义，RDRM 模型中各组成要素及要素间的关系如图 6-10 所示。

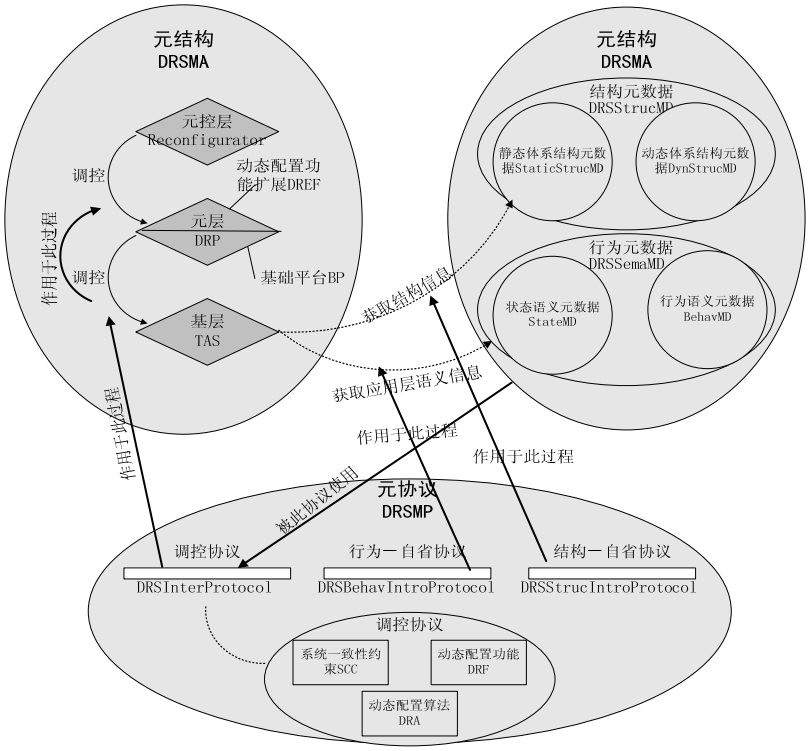


图 6-10　反射式动态配置模型 RDRM

第 7 章　动态演化的基础设施

7.1　COM 构件的演化机制

7.1.1　概述

1. COM 简介

COM（Component Object Model）是微软的构件对象模型。它的实质是一些小的二进制可执行的程序，可以为应用程序和操作系统以及其他的构件提供服务。微软的许多技术如 ActiveX，DirectX，ADO 等都是基于 COM 建立的。COM 以 Win32 动态链接库（DLL）或者以可执行文件（EXE）的形式发布的可执行代码组成。其中以 DLL 形式发布的方式实现的构件程序，在客户调用时会把构件程序和客户程序运行在同一个进程中，所以被称为进程内的构件。以 EXE 的形式发布，在客户调用时，他有自己的独立的进程空间，故称为进程外的构件。总体来看，COM 提供了编写构件的一个标准方法。遵循 COM 标准的构件可以被组合起来以形成应用程序。至于这些构件是谁编写的，是如何实现的并不重要。构件和客户之间通过“接口”来发生联系，使用 COM 技术的关键，乃至整个三层体系结构的关键就是接口。COM 接口是一组逻辑上相互关联的操作，这些操作定义了某种行为，也就是这组操作的规范，而非特定的实现，实质也就是接口代表了接口调用者和实现者之间的一种约定。所有对 COM 的操作都是通过接口指针来进行的。通过接口可把构件的功能展示给调用者。

2. DCOM 和 MTS 简介

DCOM（Distributed Component Object Model）是 COM 发展的新阶段。它扩展了构件对象模型技术，使其能够支持在局域网、广域网甚至 Internet 上不同计算机的对象之间的通讯。使用 DCOM 技术，应用程序就可以在位置上达到分布性，从而满足应用的需求。DCOM 的发展推动了 COM 在网络环境中的发展。

MTS（Microsoft Transaction Server）是一个基于 COM 构件的事务处理系统，提供了对多用户、分布式事务和安全的支持，为开发基于 COM 构件的分布式应用程序提供了环境。在多用户环境中，通过为展开和实施应用程序提供一个设置和运行环境从而简化应用程序的开发，使得开发人员能集中精力开发应用程序的业务逻辑而不是数据库连接和线程同步。MTS 提供的数据连接缓冲、线程管理、分布式事务服务很好地解决了多客户端利用构件频繁访问后台数据库等一系列问题，有助于开发人员开发大规模的、可靠的以及基于

COM 构件的应用程序。MTS 体系结构包括 MTS 环境和 MTS 服务。

3. COM+简介

COM+并不是 COM 的新版本，可以把它理解为 COM 的新发展，或者为 COM 更高层次上的应用。COM+的底层结构仍然以 COM 为基础，它几乎包容了 COM 的所有内容。可以这样认为，COM+是 COM，DCOM 和 MTS 的集成，它综合了这些技术要素。更重要的是，COM+倡导了一种新的概念，它把 COM 构件软件提升到应用层而不再是底层的软件结构，它通过操作系统的各种支持，使构件对象模型建立在应用层上，把所有构件的底层细节留给操作系统，因此，COM+与操作系统的结合更加紧密。

7.1.2 平台设计

若要使得应用系统在 COM 平台上提高可重用性、可维护性、可继承性，关键在于以下 2 点。

（1）设法降低 COM 平台中层与层之间的耦合度。

（2）设法降低业务构件之间的耦合度。

高内聚低耦合一直是软件工程中大力提倡的设计思想，从平台构思、设计到实现都应该遵循。不难想象，若能降低客户层与中间层之间的耦合度，势必能大大提高开发和维护效率，便于 COM 构件的移植。

COM 构件的技术关键在于接口，微软提出的二进制层次上的接口规范使得 COM 构件能满足跨语言调用，同时也带来麻烦，主要是接口版本管理的问题，COM 构件的接口改变会带来很多灾难。倘若能统一所有 COM 业务构件的接口形式，就能真正实现业务构件改变后客户程序无须改变。

构件交互是基于构件的分布式应用系统的核心问题，如何有效解决构件交互问题是提高软件开发质量的关键。目前的中间件平台对构件交互的支持主要是基于传统过程调用的 RPC 机制和面向对象范型的同步式请求应答消息机制，而在后一种机制中构件交互协议的实现嵌入在构件的功能描述代码中。因此构件的交互集成问题比较繁琐，设计出的系统往往结构复杂，缺乏灵活性。业务构件彼此之间的相互调用是紧耦合的、不利于维护的开发模式，若能设法降低构件之间的耦合度，灵活实现构件交互，将有利于提高业务构件的可维护性、可重用性。这里提出的一种独立于构件实现的、支持形式化分析与动态测试的构件交互模型来解决此问题，还提出了基于 XML 建立数据的共享统一模型，实现构件间的松耦合。

另外，工作流技术和业务逻辑的可配置服务都是提高管理类软件灵活性的有效方法。在工作流建模机制中，增加的请求、服务、协调、多实例化等建模元素，提高了工作流模型的描述能力、降低了系统复杂性、提高了系统柔性和适应性；业务逻辑配置机制中，提出了一种业务逻辑配置脚本，有利于基于解释器进行业务逻辑的解析、调度，提高了软件的灵活性。由于从某种角度讲，这里的中间件平台是为了让 COM 构件技术更灵活、更简便地用于软件开发，因此可以借鉴上面两种技术。

这样的中间件平台作为一种基础设施，代表着中间层，为客户端程序提供业务逻辑服

务，其实质是对整个中间层进行了封装，客户层和业务构件取得联系都必须经过本中间件平台。其中，具代表性的基于 COM 的中间件平台是浙江理工大学的 ProBase。

ProBase 平台主要包含以下主要部件和功能：

（1）实现一种基于记录集对象的比 XML 高效的数据交换格式。

（2）实现一种基于流水线思想的自定义容器作为平台内部交换区。

（3）实现一种基于自定义列集技术的统一的分布数据传输载体。

（4）统一平台的业务构件的接口形式，并实现业务构件的原子化以及业务构件间的松耦合。

（5）提供一种基于 XML 的灵活的业务逻辑可配置的脚本服务。

（6）实现一种类似工作流引擎的核心部件。

1. 构件的数据交换

该平台采用封装后的记录集（RecordSet）对象作为通用数据交换标准。记录集是 ADO 构件中的重要对象之一，其实质是二维表，主要优点是易于使用、高速度、低内存支出和占用磁盘空间较少。用它作为参数集合的载体的实现形式，传输效率比基于树型结构的 XML 要高得多。

该平台将封装后的记录集对象命名为 PPA（参数包 Parameter PackAge），它负责将模块间交换的离散数据根据逻辑相关性进行打包，从而形成数量有限的参数集合实体，供业务构件方法使用。因此，PPA 又是 ProBase 平台用于数据建模的基本手段。

2. 数据交换区设计

为了更好地实现平台化开发模式，统一 ProBase 平台对外接口、统一业务构件的接口形式、降低业务构件之间的耦合度、解决构件交互问题，引入了数据交换区这一重要部件，而这一部件的设计思路来源于已经成熟应用于大规模生产的工业流水线思想。与工业流水线的工作方式类似，只是数据交换区取代了流水线，业务构件的方法取代了工序，输入 PPA 集合取代了所需的零件，输出 PPA 集合取代了成品。

一条工业流水线相当于 ProBase 平台中对一段业务逻辑的处理流程。数据交换区在获得输入 PPA 集合后，若干业务构件方法开始顺序工作，每个业务构件方法从数据交换区中取出需要的那个 PPA，当构件完成方法后，将处理完后的那个 PPA，重新放入数据交换区，供其他业务构件方法所用，数据交换区最终剩余的是经过处理的 PPA 集合，也就是输出 PPA 集合。PPA 是若干变量的集合，而数据交换区实质是若干 PPA 的集合。PPA 将某个业务构件的方法所需要的参数封装在一起，统一了业务构件方法这一级别上的参数形式，而数据交换区则更是将一段业务逻辑需要的所有参数都封装在了一起，统一了更高层次的参数形式。

3. ProBase 平台下的业务构件接口形式的统一设计

COM 构件技术的关键在于接口，在传统的基于 COM+的三层架构软件中，COM 构件的接口是限制系统灵活性的关键问题。若能设法统一整个系统中的所有业务构件的接口形

式，势必将使构件间的交互方式变得容易、降低业务构件间的耦合度，从本质上减轻了构件开发、维护的工作量，并使业务构件的原子化成为可能。

如上文所述，一个 PPA 将某个业务构件的方法需要的参数组装在了一起，故一种理想的业务构件对外接口行为可作如下统一的形式化定义：

ResponsePPA Run(behavior,ParameterPPA)

上式中的 Run 是个动词保留字；Behavior 标识业务构件方法的具体行为；ParameterPPA 用于传递行为参数（入参）：ResponsePPA 用于返回行为结果（出参）。逻辑上，上述形式化定义适用于业务构件任意复杂的交互过程。但由于每个业务构件方法所需的参数集合（PPA）都会不同，故需对上式进行调整。由于上一节中引入了“数据交换区”的概念，业务构件接口的形式化描述可更改如下：

Run (Behavior,DEA)

即直接用数据交换区来作为业务构件方法的参数。当然，数据交换区在实现时就必须向 ProBase 平台使用者提供两个方法——从数据交换区里取出所需 PPA 对象的方法和将处理后的 PPA 对象放回数据交换区的方法。形如：

GetPPA([in]PPAName,[out]M ispatch* PPA)//取 PPA

AddPPA ([in]ID ispatch* PPA)//放 PPA

显然，如果所有的业务构件接口都实现 Run( )方法，就能实现业务构件接口形式的统一。具体实现时，可自定义一个包含 Run( )方法的构件接口规范，所有在 ProBase 平台下开发、运行的业务构件都必须实现该接口规范，在实现该接口规范的基础上来编写具体的业务逻辑。由于统一了 ProBase 下所有业务构件接口形式，使得业务构件间的交互变得容易，并使得基于本平台开发的业务构件能实现最大程度的原子化，可将业务逻辑的细分程度达到一个业务构件方法对应一次数据库操作，再结合 ProBase 平台提供的业务逻辑可配置式脚本服务（本章将会在后面详细叙述），ProBase 平台下的业务构件的灵活性和可重用性将大大超过传统的业务构件。

4. 业务逻辑可配置式服务的设计

Pro Service 不能仅仅作为 ProBase 平台提出的一个逻辑概念，ProBase 平台需要设计严格的 ProService 定义语法，并能解析和执行符合该语法的 ProService 文件。此外，为了向 ProBase 平台的使用者提供比传统三层架构开发模式更灵活的业务逻辑服务、进一步提高 ProBase 平台下的业务构件的可重用性，本章试图将 ProService 设计为一种可配置式的脚本文件形式。在现有的研究成果中，结合了构件技术与中间件技术，构建了装配式的软件系统,一种可快速配置、重构的构件化系统框架。由于 ProBase 平台下所有业务构件的接口都被统一、业务构件之间也被设计成松耦合的形式，ProService 的可配置式设计就显得较为轻松。

Pro Base 平台使用文档对象模型（DOM）来实现对 XML 的解析功能。业务逻辑可配置式的脚本文件的格式定义如下：

```
<Boot>
<ProService>
```

```
<Stepl ProgId=" XXXXXX″MethodName="XXX"/>
<Step2 ProgId=" XXXXXX″MethodName="XXX"/>
<Step3 Progld=" XXX.XXX″MethodName="XXX"/>
...
...
<ProService>
<DataBaseConnection>
<Connectionl ConnName="XXX" Conn="Provider=SQLOLEDB.1;Persist Security
Info=False;User ID = XXX;Password=XXX;Initial C ata log=bank;Data
Source=DUXC"/>
<Connection2...>
...
...
</DataBaseConnection>
</Boot>
```

其中，XML 的文件名就代表了 ProService 的名称，即客户端程序希望 ProBase 平台提供的业务逻辑服务名；<ProService>与</ProService>之间的若干属性代表该业务逻辑由哪几步业务构件方法顺序组成：ProgId 是 COM 技术中的专业名词，代表着“业务构件名和接口名”；MethodName 则代表着具体的方法名。业务构件的创建、调用等操作将由 ProBase 平台负责统一管理，从而进一步降低了业务构件间的耦合度和构件交互的难度。

另外，为了进一步降低中间层与数据层的耦合度，以下考虑将数据库连接以及相关的事务功能设计成由 ProBas 平台来统一控制、管理。<DataBaseConnection> 与 </DataBaseConnection>之间的若干属性代表着数据库连接名（ConnName）和数据库连接字符串（Conn），可以只有一个数据库连接也可以有多个数据库连接。由此可见，ProBase 平台在解析 ProService 时，首要任务是根据 XML 文件中提供的数据库连接字符串来创建相应的数据库连接对象（Connection），然后还要将该数据库连接对象放入数据交换区中，以供相关的业务构件提取并使用，同时控制每个数据库连接对象的事务功能，并在适当时负责将其销毁。

7.1.3　ProBase 平台引擎的设计

平台的核心任务是完成对 ProService 的解析执行，从而为客户端程序提供业务逻辑服务，负责完成这部分任务的是 ProBase 平台引擎，它是平台的心脏，是比较复杂的部件，管理、调度着 ProBase 平台内的一切资源。

工作流是一类能够完全或者部分自动执行的业务过程，根据一系列过程规则，文档、信息或任务能够在不同的执行者之间传递、执行。

ProBase 平台并非基于工作流技术，但在平台内部框架的设计中，独创的基于工业流水线思路的业务流程处理机制与工作流技术的确有些相似；此外，ProBase 平台也需要一个对业务逻辑进行解析、执行的引擎，故可将 ProBase 平台引擎归纳为一种类似工作流引擎。

在成功设计了 PPAS 以及其他 ProBase 平台的重要部件后，ProBase 平台引擎需要完成的整体工作已经渐渐明朗。由于引入了数据交换区的概念，其设计、实现难度也比传统的工作流引擎更为容易。因此，对 ProBase 平台引擎的设计要求就是：除了完成对 ProService 的解析执行任务外，还要合理利用平台资源，尽可能做到高效和可靠。

ProBase 平台引擎的工作流程设计图如图 7-1 所示。

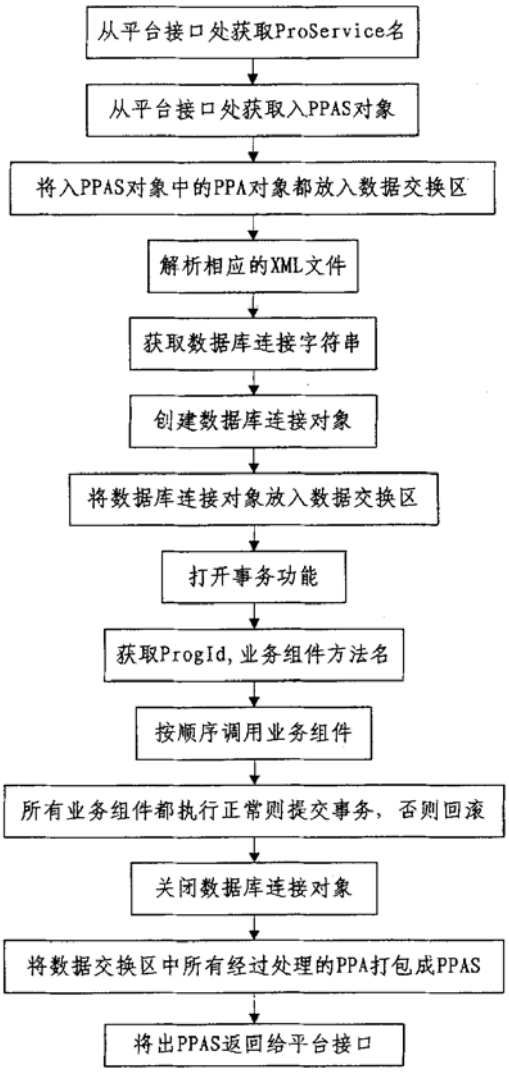


图 7-1 ProBase 平台引擎的工作流程设计图

7.1.4 业务构件交互问题

基于构件技术的软件开发现在已成为软件开发的主流形式，该形式的一个关键问题是如何定义设计中的公共或标准成分。除了构件标准以外，构件间的互操作性需求蕴涵了某种公共机制或特征。它类似于人类交互中需要的一些显式或隐含的约定，如公共语言、上

下文环境等。构件互操作也需要考虑互操作实体的协作控制问题和数据共享问题，即公共的交互机制。

不同的系统使用了不同的方式来定义这类公共交互机制。如分布式对象计算标准 CORBA 是通过接口定义语言 IDL 和相应的支撑设施提供分布构件的互操作；而 Java Bean 和 COM 是通过构件接口来进行构件交互。这种基于服务标准化、以构件接口为中心的互操作方法只是以功能分离的形式提供了服务，而忽略了协同构件工作的关键——构件交互。构件交互是指在多个构件之间，为了达到某个目标而相互交换消息序列来约束互动的动态行为。它不只是简单的两两服务请求，而是包含更多的构件，并经过许多中间状态。这种交互控制不应分隔存在于各个构件，而应分离于构件计算功能，在概念层次或实现层次都成为独立的非功能部件。构件交互已经成为基于构件技术进行系统开发的关键问题。

ProBase 平台定义了业务构件标准，并通过数据交换区、ProService、引擎三者间的协同工作，由数据交换区负责装载业务构件群所需要的所有 PPA ；ProService 负责定义一段业务逻辑中所涉及的业务构件群；引擎负责创建、调用业务构件及业务构件群之间的交互。由此完美地从技术角度解决了业务构件交互困难的问题，使 ProBase 平台真正成为一种可为复杂应用提供良好集成框架的中间件平台，可快速有效地组合业务构件群来构建应用系统，灵活性强。

7.1.5　ProBase 优点总结

该平台设计并实现了一种新颖的基于 COM 构件技术、中间件技术、分布式、工作流等技术实现中间件平台，为三层架构软件的开发、运行提供支撑，使 COM+技术更好、更高效地应用于大规模的企业级应用中。目前，ProBase 平台已投用于杭州创业软件集团，该公司基于本平台成功开发了 ERP 软件——医院资源管理系统（BSHRP3.0），在企业级大数据量、大访问量、复杂业务处理应用中取得了良好的效果，验证 ProBase 平台的有效性。与传统的基于 COM+技术开发的三层架构软件相比，基于 ProBase 平台开发的三层架构软件的可维护性、可重用性、可集成性更高，具体的优点体现在：

（1）采用封装后的记录集对象作为 ProBase 平台的通用数据交换格式，由于记录集的结构是二维表，故在运行效率上将大大超过被广泛使用的具有树型结构的 XML。

（2）提出并实现了一种基于工业流水线思路的自定义对象容器作为 ProBase 平台内部数据交换区，结合自定义的构件接口统一了业务构件的调用接口，降低了业务构件之间的耦合度，实现了业务构件的原子化，解决了构件交互难的问题，提高了系统稳定性、可维护性、可重用性。

（3）将数据库连接从业务构件中分离，由 ProBase 平台统一管理数据库连接的创建、销毁、事务功能，从而降低了中间层与数据层的耦合度，便于数据库移植，并保证了业务构件的安全性。

（4）降低了业务构件的开发、维护难度。

（5）采用自定义列集、散集技术实现了数据交换区对象的可分布高效传输功能。将其作为分布式数据交换载体，并结合 UML 系统分析中对用例的形式化描述实现了 ProBase 平台供外界访问的统一而简约的接口，降低了界面层与中间层的耦合度。

（6）降低了客户端程序的开发、维护难度。

（7）提供一种基于 XML 的灵活的业务逻辑可配置式脚本服务，使业务逻辑的调整更为便利，提高了系统可维护性、灵活性。

（8）平台内部实现了一个复杂而高效的类似工作流引擎。

（9）提出了一种新的三层架构开发模式，为业务逻辑的定义、实现与运行提供统一的规范和环境，为复杂应用软件提供集成框架。任何子系统只要符合 ProBase 平台下的开发规范，都能被很好地兼容，有效提高了系统可集成性。

7.2 CORBA 构件的演化机制

7.2.1 概述

1. CORBA 简介

CORBA 是由 OMG 对象管理组织制定的一个较成熟的对象中间件规范，其全称为通用对象请求代理体系结构（Common Object Request Broker Architecture）。在几种主流的中间件技术中，CORBA 的突出特点在于其跨平台、跨程序语言的特性。J2EE 只能使用 Java 语言，.NET 局限于微软的操作系统，而只有 CORBA 才可以使用几乎任何一种编程语言，运行于任何一种软硬件平台之上。CORBA 的这种特性使其成为许多公司（Oracle,Netscape,Sun,IBM 等）进行系统集成的技术基础。

CORBA 是 OMG 组织所提出的对象管理体系（Object Management Architecture,OMA）的核心平台，OMG 组织将分布式计算环境下的所有对象置于 OMA 的管理之下。OMA 使用对象模型和引用模型来描述如何以平台无关的方式来指定分布式对象，以及分布式对象之间的交互。对象模型用于定义对象描述。在对象模型中，分布式对象被定义为具有唯一标志的实体，这些实体只能通过严格定义的接口访问，且它们的位置和实现细节对访问者是透明的。引用模型则说明了对象之间如何进行交互，它将对象按接口种类分组，各种对象通过对象请求代理 ORB（Object Request Broker）链接在一起，

2. CORBA3.0 规范

为应对分布式对象计算技术所面临的挑战、提供良好的构件模型支持和大规模分布式应用支持，2002 年 OMG 发布了 CORBA3.0 规范族。CORBA3.0 规范族由 CORBA3.0 内核、CCM3.0 规范、配套的语言映射规范、Minimum CORBA 规范、实时 CORBA 规范和一系列 CORBA 服务规范组成。相较于 CORBA2.x 版本，CORBA3.0 规范族在 CORBA 构件模型、Java 与 Internet 集成和异步消息传递等方面作了较大的更新。

（1）CORBA 构件模型。CORBA3.0 规范族的一个重要特征是加入了对 CORBA 构件模型（CORBA Component Model,CCM）的支持。CORBA 构件模型是 OMG 组织制定的用于开发和配置分布式应用的服务器端构件模型。它在服务端引入容器来管理 CORBA 构件，

其体系结构主要包括四大部分：抽象构件模型、构件实现框架、容器编程模型和打包配置模型，以此搭建了分布式企业级服务器计算架构，为创建分布式中间件提供了一个一致化的构件体系。CORBA3.0 规范族中的 CCM3.0 规范详细规定了 CORBA 构件模型的体系结构，CORBA3.0 内核规范和其他规范中也增加了对 CCM 的支持机制。在支持构件级重用、提高服务方软件可复用性的同时，CORBA 构件模型也弥补了传统 CORBA 规范对象生命周期管理能力较弱、与 EJB 的互操作困难等缺陷，并为动态配置 CORBA 应用程序提供了更大的灵活性。

（2）与 Java 和 Internet 集成的能力。CORBA 3.0 规范族增强了 CORBA 与 Java 和 Internet 集成的能力，这主要体现在值类型的语义得到了进一步的完善。值类型的存在不仅实现了 CORBA 中的对象传值语义，使得可以传递如对象图之类的复杂数据类型，而且建立了 CORBA 与 Java 之间互操作的桥梁。值类型的存在使得反向的 Java 到 IDL 映射规范成为可能，使得 Java RMI 对象可以以 CORBA 对象的方式与其他 CORBA 对象交互。值类型也使得 XML 可以直接映射为 CORBA 中的本地数据类型。此外，可互操作的名字服务规范和 CORBA 防火墙规范也增强了 CORBA 与 Internet 集成的能力。

（3）异步消息模型和消息传递 QoS 控制机制。在 CORBA2.4 规范中引入异步消息模型和消息传递 QoS 控制机制，是 CORBA3.0 规范族的一个重要组成部分。CORBA3.0 规范定义了若干种异步和时间独立的调用模式，调用结果可以通过轮询和回调等方式回传。同时，可以通过策略来设定消息传递的 QoS，如优先级、time-to-live 和路由等。MinimumCORBA、容错 CORBA 和实时 CORBA 等规范则适用于各种不同的场合。

3. CORBA 内核规范与 CORBA 内核

随着分布式对象计算技术的发展，CORBA 从单一的规范已经演化为一个规范族，它不仅包括 ORB、GIOP 等传统的 CORBA 构件，还包括衍生出来的实时、嵌入等扩展 CORBA 规范，以及 CCM 规范和大量的服务和设施规范。因此，OMG 组织在发布 CORBA3.0.1 规范时，将传统意义上的 CORBA 规范重命名为 CORBA 内核规范以示区别。

CORBA 内核规范是 CORBA3.0 规范族的核心。它详细规定了由 ORB 和与 ORB 密切相关的构件所组成的 CORBA 内核的行为和接口，从而为 CCM、各项 CORBA 服务和用户程序提供基础设施。CORBA 内核包括包括 ORBCore、POA、GIOP/IIOP、接口池等构件。

（1）对象请求代理 ORB

对象请求代理 ORB 是 CORBA 内核的核心，是对象间的“通信总线”。它可以看作是把应用程序和分布式系统底层细节分离的软件，主要负责客户与服务方之间的通信，将请求传递到服务方，再将应答传回给客户方。ORB 包含了用来识别和定位对象，管理连接和传递数据的必要结构。

ORB 最重要的特征在于提供了客户和目标对象之间的交互透明性。具体而言，ORB 使分布式计算环境中的以下元素对应用程序员透明：

a.对象位置。通过 ORB 传送请求时，客户方无需了解目标对象的具体位置。目标对象可能处于不同机器的不同进程、同一机器的不同进程甚至同一机器的同一进程。

b.对象实现。客户方无需了解目标对象具体是如何实现的。

c.对象的执行状态。即使对象尚未被激活，ORB 也可以透明地激活对象。

d.对象的通信机制。底层的通信对应用程序员是透明的。

ORB 的以上特征使得应用程序员可以专注于领域逻辑，而不是分布式系统的某些低层问题。

ORBCore 具体负责请求的传递。ORBCore 是一个逻辑实体，它可以用各种方法实现（如一个或多个过程，或一个库集合）。为了减轻编写程序的困难，CORBA 规范为 ORB Core 定义了一个统一的接口供用户调用，即 ORB 接口。IDL Stub 和 IDL Skeleton 是 IDL 编译器为每个接口生成的静态存根和框架，它们是客户及服务方应用程序和 ORB 之间的中介，负责调用 ORB 完成请求和结果的编解码和传递。Stub 和 Skeleton 是编译时生成的，因此这种调用方式称为静态调用。

DII 使客户方可以在运行时动态发现服务对象并调用其方法。DSI 则为 Servant 提供运行时的捆绑机制，处理对不具有框架的 Servant 的调用。DII 和 DSI 是存根和框架的动态实现方式，它们提供在运行时动态构造和解释请求的能力，对象适配器 OA 负责在服务方帮助 ORB 把请求传给对象并激活该对象。每个对象适配器将提供对象引用生成与解释的方法、对象实现的激活与休眠手段、对象实现的注册等等。CORBA2.2 以后版本中的对象适配器是可移植对象适配器，可以确保服务器端 Servant 在不同厂商的 ORB 平台之间的移植。

（2）ORB 互操作协议

CORBA 规范将 GIOP（Gerneral Inter-ORB Protocol）定义为 ORB 间基本的互操作性协议。GIOP 并不是一个可直接用于 ORB 之间进行通信的具体协议，它可以看作一个额外的抽象层，从而获得可缩放性、灵活性和架构中立性等益处。ORB 间的互操作协议还包括 ESIOP 等。IIOP 则是 GIOP 在 TCP/IP 协议上的一个具体实现。

（3）接口池

接口池是 ORB 的一个相对独立的组成部分，其中存放的是 IDL 接口的元信息，如接口中有什么操作、参数和返回值的格式等。接口池可以在运行时提供接口信息，从而使应用程序可以与编译时未知的对象交互。接口池与 ORB 的 DII/DSI，DynAny 等一起组成了 CORBA 的动态机制。此外，接口池还被用于管理接口定义的安装和发布、为 CASE 环境提供支持、为编译器等语言绑定机制提供接口信息等。

7.2.2 反射式动态配置模型 RDRM

本节用建模的方法对动态配置系统及其机理进行细致的分析和研究，以发现动态配置的本质、明确动态配置中诸要素之间的关系和运行规律，从而分清其中的重要者和次要者，为我们指出研究的重点。动态配置的过程，在本质上就是一个反射过程；动态配置系统在本质上就是一个反射系统；动态配置系统模型在本质上就是一个特征化的反射系统模型；动态配置系统模型中的强活跃要素则是需要重点研究的对象。这也就是本节所要阐述的核心和结论。后续章节将以本节建立的反射式动态配置模型 RDRM（Reflecting Dynamic Reconfiguration Model）为中介和基础，根据强活跃要素是需要重点研究的对象这一重要结论，对 RDRM 中的各个强活跃要素分别进行研究，针对其中的不足提出新的概念、算法和体系。

在本节中，首先介绍反射的基础理论和基本模型，然后介绍并分析动态配置系统的反射机理和体系，说明其具备构成反射系统的要件，并由此建立反射式动态配置模型 RDRM

这一反射理论的特征化模型，指出 RDRM 中的强活跃要素正是影响动态配置系统优劣的关键因素，复杂软件系统的动态配置是系统在运行过程中，实时收集系统自身的状态信息，对状态信息进行综合、转化和分析，并在系统运行的某个时刻，根据系统外部的管理请求和系统自身状态的实时分析结果，对系统内部的配置进行更改的过程。虽然目前实现动态配置的方法很多，也各有特点，但都具有类似的结构特点、行为模式和特征。从它们这种在系统运行过程中通过实时获取应用系统的体系结构和语义信息，并反过来对应用系统运行行为的调整和控制过程，我们可以清晰地看到其中很好地体现出了反射的思想。

实际上，动态配置系统中的各个要素与反射理论模型中的各个要素存在一一对应的关系，因此动态配置系统可以抽象成为反射理论的一个特征化模型。将动态配置系统及其要素纳入该模型的统一视图进行介绍和讨论，以反射理论为基础考察动态配置的本质结构和机理，有助于更为准确、清晰地描述动态配置的各个要素，使我们能够从本质上阐明不同动态配置方法之间的联系和区别。

1. 反射系统及相关概念

所谓计算系统，是指对某个问题领域进行求解并能将求解结果作用于该领域的系统。因此，计算系统是领域相关的。反射系统的特别之处在于与之相关的领域就是系统本身，而且系统中用于描述系统自身的数据结构和系统本身之间存在着“因果关联”。这里的“因果关联”是指：如果有关系统的描述发生了变化，那么说明系统本身发生了相应的改变；反之，如果系统本身发生了某种改变，那么有关系统的描述也必定发生相应的变化。

在反射系统中，用于描述系统本身的数据结构被称为系统的元数据（Meta-data），它是反射实现的基础。反射系统就是在系统运行时通过监视系统的行为和状态，维护正确的元数据，从而达到根据环境和应用程序的需要对系统进行重新配置的目的。

元系统是指将另一个计算系统作为问题领域的计算系统，而作为问题领域的计算系统被称为目标系统。从这个意义上而言，反射系统就是以自身为目标系统的元系统，而反射性是指该元系统推理并操作自身的能力，反射则是该元系统对自身进行的推理与操纵。

综合上述，一个反射系统通常具有两层结构。一层负责对系统的问题域建模并对其进行推理、计算和操纵，以解决该领域中的问题；另一层则负责对系统自身建模并进行推理、计算和操纵，从而能够自动或在特定要求下改变系统的配置，使之适应某些变化。前者被称为基层（Base-level），后者被称为元层（Meta-level）。这种在逻辑上由基层和元层组成的两层结构即为反射理论中的元层体系结构（Meta-level Architecture），简称元体系结构。通过对元层进行细粒度地划分，某个元层自身可能被更高层的元层所反射，从而得到由多个元层组成的反射塔。

元层对基层的反射能力通过元对象协议（Meta Object Protocol）体现。从对基层实体信息的读、写操作的角度划分，元对象协议可分为自省协议（Introspective Protocol）和调控协议（Intercessory Protocol）。元层通过自省协议读取基层的元数据，这一过程被称为具体化过程（Refication Process）。元层通过调控协议改变基层实体的结构和行为，这一过程被称为吸收过程（Absorption Process）。从反射基层实体的具体内容的角度来划分，元对象协议分为结构反射协议（Structural Protocol）和行为反射协议。其中，结构反射协议反射基层实体的结构信息；行为反射协议则反射基层实体的语义信息。根据上述划分，元对象

协议可具体分为四类：结构自省协议、结构调控协议、行为自省协议、行为调控协议。

2. 反射系统模型

除了元层体系结构中的基层和元层外，在反射系统的元层之上还存在一个元控层（MetaControl-level），或者说应将元层反射塔的顶层作为元控层。实际上我们可以看到在元层体系结构的表述中隐含了该层，但并未显式指出。而将元控层作为反射系统模型要素之一的理由是：从系统完整性角度出发，应把对系统自身进行监控和操纵等管理请求以及控制意图也纳入系统中，作为系统的一部分进行考虑，即把其作为单独的一个元控层。该层实际上和元层类似，只不过它反射的对象是位于反射塔顶端的元层。所以在此视图下元控层实际上是一个特殊的元层，位于反射塔的上方。

我们首先抽取出反射系统模型的要素。

（1）数据要素

元数据：基层的结构及语义信息，依赖于结构自省协议和行为自省协议的执行，或者来源于基层的设计信息。

（2）实体要素

① 基层实体及基层：负责解决领域问题的实体，实现系统对外提供的计算功能和服务，是元层实体观测与调整的目标。所有基层实体的集合构成基层。

② 元层实体及元层：负责监控、收集基层实体的结构及语义信息，并据此对系统自身进行调控的实体。所有元层实体的集合构成元层。

③ 元控层实体及元控层：对元层或通过元层对基层实施监控的实体，位于反射塔的顶端，可能对应于配置管理者。所有元控层实体的集合构成元控层。

（3）实体间关系要素

① 结构自省协议：元层获取基层结构信息的约束、功能和操作集合。

② 结构调控协议：元层调整基层结构的约束、功能和操作集合。

③ 行为自省协议：元层获取基层语义信息的约束、功能和操作集合。

④ 行为调控协议：元层调整基层语义的约束、功能和操作集合。

3. 基于反射理论的动态配置模型

通过剖析动态配置系统，我们发现动态配置系统各要素与反射系统各要素之间存在一一对应的关系，动态配置系统满足构成反射系统的要件。因而，我们可利用反射系统模型对动态配置的本质和机理进行研究和分析。通过探究动态配置系统的反射机理，针对动态配置系统反射体系的特点，我们首先建立了反射式动态配置模型 RDRM（Reflecting Dynamic Reconfiguration Model）。RDRM 是反射系统模型在动态配置领域的细化和特征化。RDRM 中存在诸多要素，其中有较为活跃且对其他要素影响较大的要素，也有不活跃且对其他要素没有影响的要素。因此本节还将对活跃要素进行明确定义，进一步区分强活跃要素和弱活跃要素，并据此界定 RDRM 中不同要素的活跃类型。通过对这些要素活跃性的分析，有助于我们找出影响动态配置系统功能、正确性、性能等方面的关键因素，并抓住其本质，从而得出对现有动态配置方法进行分析、度量和评测的基准，同时为我们开展动态配置技术的研究以及设计并实现功能强、可靠性高、性能优良的动态配置系统提供指导。

7.2.3　动态配置系统反射体系

动态配置平台由三大类部件组成：系统信息收集部件、系统信息库和动态配置引擎。系统信息收集部件将收集到的应用系统的体系结构信息和部分有关构件特性的语义信息存放于系统信息库中。利用在系统信息库中存放的对当前系统结构完整的描述信息以及部分语义信息，高层的配置及管理人员可获得应用系统的全局视图，并基于对系统的分析以及参照要达到的特定管理目标，产生相应的动态配置意图，交由动态配置平台实现。动态配置引擎利用系统信息库中的系统结构和语义信息，同时通过系统信息收集部件获取构件实时运行状态信息，从而在适当时机调控构件行为，实现动态配置意图。

7.2.4　RDRM 模型中的要素活跃性分析

RDRM 模型由诸多要素组成。这些要素可以被分为非活跃要素和活跃要素两类。RDRM 模型在不同平台上的具体映射，改变的仅是活跃要素。而某些活跃要素之间存在一定的因果关联：其中一些活跃要素的改变，将导致若干其他活跃要素的改变。我们将前者定义为强活跃要素，而将后者定义为弱活跃要素：

定义 1　活跃性：模型中诸要素由于 RDRM 模型被映射到不同平台等原因，发生改变与否的性质。

定义 2　活跃要素：由于 RDRM 模型被映射到不同平台等原因而发生改变的要素。

定义 3　强活跃要素：因为自身的改变而导致若干其他活跃要素改变的要素。

定义 4　弱活跃要素：主要由于其他活跃要素的改变而导致自身改变的要素。

不论 RDRM 在哪个平台上进行映射，Reconfigurator 都是普通意义上的配置者，因而属于非活跃要素。由于此处我们关心的 TAS 是逻辑意义上的目标应用系统，而不是基于具体平台实现后的实际系统，因而 TAS 也是非活跃要素。除此之外的其他要素都将或多或少发生改变，属于活跃要素。

调控协议体现了动态配置系统的能力，是 RDRM 模型中的要素，也是在 RDRM 模型的映射过程中主要发生变化的要素，用于区分不同动态配置系统，同时也是评价不同动态配置系统的主要标准。动态配置系统采用的调控协议不同，包括调控协议的正确性约束、功能及协议内容的不同，将决定在动态配置的正确实施过程中所需的元结构和元语义数据的具体内容，从而导致结构自省协议和行为自省协议的改变。而基础平台 BP 则决定了调控协议的能力是否可以被充分体现以及调控协议实现的复杂度。不同的基础平台具有的反射能力不同。反射能力包括自省能力和调控能力。基础平台在自省能力上的区别，将决定结构自省协议和行为自省协议的实施难度及能力，进而影响所收集的元结构数据和元语义数据的内容。基础平台反射能力的不足，将导致调控协议的能力无法体现或无法充分体现。为弥补反射能力的不足，势必对基础平台进行较大扩展，由此增大了调控协议的实现复杂度。显然，基础平台的反射能力越强，为动态配置系统提供的支持也就越好，所需的动态配置功能扩展相应也就越小。

7.2.5　RDRM 模型在 CCM 平台上的映射

在反射式动态配置模型 RDRM 中，基础平台及调控协议作为强活跃要素决定了动态

配置系统的优劣。鉴于 CCM 分布式构件技术以其优良的反射能力和天然的组装部署框架为动态配置的实现提供了良好支持，因此在对调控协议及其正确性约束进行深入研究的基础上，我们实现了基于 StarCCM 的动态配置平台 StarDRP。StarCCM 是国防科学技术大学网络所遵循 CCM 规范研制的 CCM 构件平台，已在 SourceForge 网站上开放了源代码。

以下首先介绍 StarDRP 的平台实现方案，其中特别对 StarDRP 的基础平台 StarCCM 的反射能力进行分析，并通过与其他分布式对象平台进行比较，进一步说明基础平台在动态配置系统的实现复杂度、可用性、可靠性和透明性等方面的关键影响。然后对 StarDRP 进行了功能及性能的测试和分析。

7.2.6 StarDRP 的实现

本节基于 StarCCM 实现了 StarDRP，完成了 RDRM 模型到软件实体的映射。因此，本节首先对 StarCCM 进行简单介绍，通过与其他分布式对象平台进行比较，分析以 StarCCM 为代表的分布式构件平台的优良反射能力及其为动态配置提供的良好支持。然后介绍 StarDRP 的体系结构，并对 RDRM 模型在 CCM 平台上的具体映射过程进行具体说明，最后本节详细介绍 StarDRP 中为调控协议的实现提供支持的若干动态配置机制。

1. StarCCM 平台

CORBA 作为一个分布计算平台，定义了一条软总线，允许客户对远程或本地的对象发出服务请求，同时还定义了一组公共的对象服务，包括名字服务、事务服务、通告服务等。遵循 CORBA 规范，利用对象服务，开发者可将来自不同厂商的复杂、异构的应用和服务集成起来，构造出适合需求的大型分布式企业应用。但是传统的 CORBA 对象模型仍然有很多缺陷，导致 CORBA 对象与底层平台紧密耦合，很难被设计、实现和重用，而基于 CORBA 的应用也很难被部署、维护和扩展。在分布式构件技术发展的大背景下，为解决上述问题，OMG 组织将 CCM 纳入 CORBA3.0。StarCCM 则是遵循 CCM 规范实现的先进的分布式构件平台。

2. CCM 构件模型

在 StarCCM 中，CCM 构件模型定义了 CCM 构件的外观特性。构件 Home 作为构件的工厂对象，对构件进行生命周期管理，实现构件的创建、删除和查询。

构件接口（Component Interface）：类似对象接口，构件接口定义了构件引用对应的提供服务接口。

刻面（Facet）：一个刻面定义一组提供服务的接口。每个构件可以定义多个刻面，向不同类型的客户提供不同的服务，代表构件的不同视图。每个刻面对应一个刻面引用。刻面引用之间、刻面引用与构件引用之间具有导航功能。当构件功能比较简单 ，或不需要提供多个视图时，可以只定义构件接口，无需定义刻面。构件接口的引入还为将普通 CORBA 对象封装成构件提供了支持。

接插口（Receptacle）：一个构件的实现可能依赖于其他构件提供的服务。为了描述构件间的这种依赖关系，CCM 引入了接插口机制。接插口定义了构件依赖的服务接口类型，

最终映射为构件与提供该接口的构件间的一组连接管理操作，包括连接的建立、删除和查询。通过建立连接，构件获取它所依赖的构件的引用。利用该引用，构件采用同步阻塞方式向目标构件发送请求，使用所需服务。因此我们认为接插口机制以同步紧耦合的方式定义了构件间的依赖关系。

事件源（EventSource）和事件槽（Event Sink）：基于 CORBA 的通告服务，事件源、事件槽以异步松耦合的方式定义了构件间的依赖关系。事件源定义了事件的发送点，用于向一个或多个关注特定类型事件的消费者发送该类型事件。事件槽定义了事件的接收点，用于接收外界发送的特定类型事件。基于事件通道，具有依赖关系的构件分别作为事件源和事件槽被连接起来。

属性（Attribute）：属性是构件向外界显露的可被访问或定制的内容，主要用于配置构件在运行时刻的行为特性。例如，计算存款利息的构件可将利率作为一个属性。构件提供服务端口包括构件接口、刻面和事件槽。构件需求服务端口包括接插口和事件源。这种既定义构件提供服务端口、又定义构件需求服务端口的对称的接口定义，清晰地刻画了构件间的依赖连接关系，不仅提高了构件的复用性，为构件的组装提供了支持，而且清楚地展现了整个应用系统的软件体系结构，为系统管理提供了便利。构件开发者可根据构件模型利用 IDL 定义所需构件，并通过 IDL 编译器生成 CORBA 的 Stub 和 Skeleton。

3. 构件实现框架

在 StarCCM 中，构件实现框架支持构件实现的自动生成。构件开发者利用 CIDL（Component Implementation Definition Language）描述构件的实现特性，其中主要基于持久状态服务，描述了构件持久状态存储的实现，并通过 CIDL 编译器生成构件实现框架。构件开发者扩展构件实现框架，加入与业务逻辑相关的代码，并与由 IDL 编译器生成的 CORBA 的 Skeleton 一起，经过与实现语言相关的编译器编译，如 C++编译器，最终生成构件实现。

4. 组装、部署框架

开发完毕的构件以构件包的形式存在。构件根据需求通过组装构成更复杂的应用，以组装包的形式存在。组装包也可以继续参与组装。构件包和组装包中，除了包含仅实现业务逻辑的构件实现文件外，还有大量描述构件包和组装包的 XML 文件。由于这些 XML 文件将在应用部署时被解析，因此被称为部署描述文件（Deployment Descriptor）。部署描述文件不但描述了构件属性的配置信息、构件所需的各种服务及其策略，还描述了多个构件如何连接组装并部署从而构成所需应用的信息，具体包括构件实例和部署位置信息以及构件实例间的连接关系等。StarCCM 的部署基础设施将根据部署描述文件，在指定位置创建构件实例、配置构件属性，并建立构件间的连接，最后驱动整个应用从部署状态进入正常运行状态，从而自动地完成构件组装及应用的分布式部署。

5. 构件运行环境：容器

通过集成事务服务、安全服务、持久状态服务及通告服务，StarCCM 的构件容器为 CCM 构件提供了运行支撑环境。容器在部署过程中获悉构件所需的服务及其策略，并在运行过程中自动提供所需服务。容器截获所有到达构件的请求，根据需要进行相应的处理，

再代理给构件。容器还可调用由构件实现的回调接口（callback interface），与构件进行必要的协调和沟通。通过上述两条途径，容器在系统运行过程中实现了对构件的管理，包括生命周期管理，构件状态激活/去活管理，以及对构件使用的事务服务、安全服务、持久状态服务和通告服务的管理等。由于容器可获取大量的构件运行状态信息，并具有强大的管理和控制构件的能力，因此容器完全可以被扩展以实现更高级的容错、实时、负载平衡等QoS 管理功能，包括动态配置功能。构件将通过内部接口（internal interface），从容器那里获得环境信息以及容器所集成的底层服务接口。

6. StarCCM 的反射能力

基础平台作为 RDRM 模型中的强活跃要素，决定了调控协议的正确性、功能性和性能以及动态配置系统的实现复杂度、可用性、可靠性和透明性。基础平台的反射能力越强，对动态配置系统的正面影响也就越大。目前，动态配置系统在基础平台上有两大类选择：分布式对象平台和分布式构件平台。StarCCM 是遵循 CCM 规范的典型的分布式构件平台。由于 CCM、EJB 和 COM/DOOM/COM+三种主流的分布式构件技术在长期的发展过程中互相借鉴，具有相似的思想、体系结构和实现技术，因此我们仅以 StarCCM 为代表，剖析分布式构件平台的优良反射能力及其为动态配置提供的支持。反射能力具体表现在两个方面：获取目标应用系统结构、语义信息的自省能力；对目标应用系统结构及行为调控的能力。

StarCCM 具有强大的自省能力，支持系统信息的完整获取。CCM 构件模型中不仅定义了构件提供服务端口，而且定义了构件需求服务端口。基于对称的接口定义，部署描述文件描述了构件间的组装连接关系，从而提供了静态体系结构信息。部署描述文件还描述了需要构件开发者提供的行为语义信息，如构件需要的各种服务的策略、动态配置实施过程中所需构件的隐式连接和主动发送请求端口信息。部署基础设施通过解析部署描述文件，获得静态体系结构信息，以及各构件的语义信息。由于容器可截获所有到达构件及构件 Home 的请求，因此可获得构件的实时运行状态信息，通过捕获所有体系结构变化事件，还可获得动态体系结构信息。StarCCM 凭借强大的自省能力，提供了大量动态配置所需的结构及语义元数据，有助于调控协议正确、高效地实施。

借助容器并间接利用底层 CORBA 平台，StarCCM 具有强大的对构件进行调控的能力。容器作为构件的运行环境，可截获所有到达构件的请求，并可通过回调接口和内部接口与构件进行必要的交互，再加上 CORBA 平台通过截获器和 Location Forward 等机制提供的反射能力和控制能力，容器可方便地对构件实施任何所需的管理和控制，如驱动构件进入被动状态，以及对请求的阻塞和重定向等。StarCCM 凭借其强大的调控能力，有助于设计并实现功能强、性能高的调控协议，同时有效降低了调控协议的实现复杂度。

动态配置的本质就是应用系统在运行阶段重新组装并部署的过程。显然，StarCCM 的组装、部署框架，为动态配置的实现提供了天然支持。与分布式构件平台相比，分布式对象平台无论在自省能力还是调控能力方面都显得比较欠缺。例如，在获取元结构数据方面，基于分布式对象平台的动态配置系统通常要求开发者显式提供应用系统的设计结构信息，而无法自动获取此类静态信息，更无法收集体系结构的动态变化信息。元结构数据的过时及不完整，将导致配置者无法正确声明配置意图，动态配置平台则无法正确实现对系统的调控。为了弥补分布式对象平台在自省及调控能力方面的欠缺，各动态配置系统均对底层

的分布式对象平台进行了较大扩展。例如，Almeida 方法将 CORBA 对象扩展成动态配置胶囊，内含工厂对象及动态配置代理等实现系统调控功能的对象。XRMI 方法则在 RMI Stub 和应用对象之间引入了虚拟 Stub，通过控制对象间的通讯并记录相关状态信息，增强自省及调控能力。显然，对于此类动态配置系统而言，由于基础平台的反射能力不足，导致系统的能力有限，透明性、可靠性、可用性不强，实现复杂度较大。

综上所述，以 StarCCM 为代表的分布式构件平台具有比传统的分布式对象平台更强的反射能力。基于分布式构件平台实现动态配置系统，不仅可以极大增强系统的动态配置能力，而且可有效降低实现复杂度。

7.2.7　StarDRP 体系结构

StarDRP 由系统信息库（System Information Library）、动态配置算法库（Reconfiguration Algorithm Library，RAL）、动态配置算法生成器（Reconfiguration Algorithm Generator，RAG）、动态配置管理器（Dynamic Reconfiguration Manager，DRM）、容器和部署基础设施六部分组成。

系统信息库 SIL 中存放了系统结构和语义信息。部署基础设施在部署代表应用的组装包时，通过解析部署描述文件，收集到应用的组成构件、构件的物理位置、构件间的连接关系等系统静态体系结构信息以及各构件的语义信息，并将这些信息存放于 SIL 中。容器在应用运行过程中，捕获体系结构变化事件，收集系统的动态体系结构信息。各容器收集的动态体系结构信息也将通过部署基础设施汇总存储于 SIL 中。为了支持系统信息在 SIL 中的合理组织，我们为 SIL 设计了层次化的系统信息存储结构。为了提高查询效率，我们还在 SIL 中建立了有关构件连接等信息的索引。部署基础设施、容器和 SIL 共同完成了对系统信息的收集、存储和查询的支持。

我们在前面设计的实现各基本动态配置意图的动态配置算法将存放于动态配置算法库 RAL 中，作为动态配置算法模板。动态配置算法生成器 RAG，根据配置者通过图形用户界面 GUI 提交的动态配置意图，从 RAL 中取出对应的动态配置算法模板，利用系统结构和语义信息，生成针对此次动态配置意图的动态配置算法，提交给 DRM 动态配置管理器，DRM 作为中心控制器，负责动态配置算法的解析和执行。容器被扩展成动态配置代理，根据 DRM 的指令，利用 CORBA 平台提供的请求截获和重定向等功能，记录构件响应请求的状态信息，检测构件状态，并实施对请求的阻塞、释放和重定向等控制。由于动态配置就是应用在运行时刻的重新组装和部署，因此，利用部署基础设施已有的组装部署功能，DRM 将完成构件的创建和删除、连接的创建和删除等组装部署行为。构件必须实现那些与语义紧密相关而且很难由第三方从外界实现的控制行为，如构件状态的收集和初始化等。DRM 根据整个动态配置算法的执行进度，在适当时机调用由构件实现的方法，完成相应的控制功能。在 DRM 的协调和控制下，容器、部署基础设施和构件将共同参与动态配置算法的实现，完成动态配置意图。

基于该结构实现的动态配置平台 StarDRP 具有以下特点。

（1）实现简单。该体系结构的设计充分利用了 StarCCM 对动态配置提供的良好支持，因而基于该结构，只需对 StarCCM 进行简单扩展，StarDRP 即可实现，从而避免了在传统分布式对象平台上实现动态配置平台复杂度较大的问题。

（2）扩展性强。SIL 中完整记录了应用系统的体系结构信息和所需的构件语义信息。利用 CORBA 平台的反射能力，容器凭借自身所处的有利位置，可获取所需的构件运行状态信息，并对构件的行为实施必要的控制。平台本身具有的对应用系统大量元数据的获取能力，以及对构件行为的强大控制能力，都使之易于被扩展，从而满足配置者在动态配置功能、动态配置实施过程等方面的个性化要求。基于底层平台的支持，配置者可通过修改或扩充存放于 RAL 中的动态配置算法模板，最终生成满足个性化要求的动态配置算法。

1. 动态配置机制

基于上面提出的体系结构，为实现自省协议和调控协议，StarDRP 提供了动态配置所需的五类机制，简称动态配置机制。

（1）系统信息描述及计算机制：无论是动态配置意图的声明、还是动态配置算法的生成，都依赖于系统信息的完整收集、记录和计算。因而该机制是动态配置正确实现的基础。

（2）构件状态检测机制：动态配置时，为了判断构件是否到达预期的一致性状态或特定状态，从而满足后继操作执行的前提条件，需要对构件状态进行检测。例如，采用阻塞方式驱动构件进入静止状态时，在阻塞到达构件的事务后，需要判断构件何时完成当前参与的事务，从而确定构件已进入静止状态并开始执行后继动态配置动作。

（3）构件行为控制机制：动态配置意图最终通过控制构件行为实现，包括创建、删除构件，建立、删除连接等操作，以及驱动构件进入静止状态和对请求的阻塞、释放等操作。

（4）构件状态传递机制：该机制实现了状态在构件间的传递，用于保证构件状态一致性，具体包括状态的收集和初始化操作。

（5）动态配置算法描述机制：该机制主要支持配置者直接描述动态配置算法，或支持根据配置者对动态配置意图的声明生成动态配置算法。

2. 系统信息描述及计算机制

只有清晰描述应用系统的结构和所需语义信息，配置者才能准确表达动态配置请求，动态配置平台也可能在必要的分析、计算后，实施动态配置，保证系统一致性。丰富的系统信息还可用于系统的管理和监控。系统信息描述机制包括三个子机制：系统信息收集机制、系统信息存储查询机制、系统信息计算机制。

系统信息收集机制用于收集在动态配置分析和实施过程中所需的系统结构和语义信息。完整的系统结构信息由系统在部署时的静态体系结构信息和系统在运行时的动态体系结构信息组成。根据前面对行为一致性强、弱保证方法中 PS、ES 关键集合的计算问题和开放问题的分析以及对动态配置算法的设计，需要收集的语义信息是指必须由构件开发者和将构件组装成应用的组装者提供的信息。构件开发者除了提供构件实现外，还将根据正常部署过程以及动态配置过程的需求，描述构件的自身特性和语义信息。组装者则根据应用需求选择合适的构件组成系统，并对组装包中的构件组成、构件的物理位置以及构件间的连接关系等静态体系结构信息进行描述，同时声明系统中的边界构件信息。我们只需对部署基础设施稍加扩展，即可在其解析各类描述文件并实现分布式部署的过程中收集到所需的语义信息和静态体系结构信息。由于容器可截获所有到达构件的请求，因而我们对容器进行扩展，在其捕获到系统结构变化事件时，包括构件的创建和删除、连接的建立和撤

销，记录相应的系统结构变化信息，从而完成对动态体系结构信息的收集。至此，部署基础设施和容器共同完成了对系统结构和所需语义信息的收集。

系统信息存储查询机制用于合理组织系统信息，并支持对系统信息的查询。根据前面设计的体系结构，静态体系结构信息和所需语义信息由部署基础设施在应用部署时收集并存储于 SIL 中，动态体系结构信息由容器在系统运行时刻收集并通过部署基础设施对 SIL 中的信息进行相应更新。更新系统体系结构信息的方式有两种：推模式、拉模式。所谓推模式，即容器一旦收集到系统结构变化信息，就通过部署基础设施逐层上报，更新 SIL 中的信息。所谓拉模式，则指容器在收集到系统结构变化信息后，并不立即上报，而是保存在本地。在实施动态配置或其他管理目标而需要获取当前体系结构信息时，容器将在上层管理模块的要求下提供系统结构变化信息。推模式实现了对 SIL 中信息的实时更新。但是，这种实时并不彻底。从容器收集到体系结构变化信息到更新 SIL 之间存在时间间隔，若在这段时间内根据系统当前结构信息实施了动态配置则可能引发异常。而且，并非所有的系统结构变化信息都有必要反映到 SIL 中。例如，在电子商务系统中，系统在每个客户到达时都将为其创建一个购物车构件，并在客户离开时删除该构件。在 SIL 中反复更新此类系统结构变化信息，显然是没必要的，而且造成性能的浪费。因此，我们选择拉模式的系统体系结构信息更新方式，只在需要时获取容器收集的动态体系结构信息。

为了支持系统信息在 SIL 中的合理组织，并支持高效的查询，我们还为 SIL 设计了层次化的系统信息存储结构。从上至下依次是体系结构描述对象、构件 Home 描述对象和构件描述对象，对应于 CCM 应用体系结构的层次关系：一个应用直接由若干构件 Home 组成，而每个构件 Home 又管理着若干构件。在系统信息描述对象的层次化结构中，每个上层描述对象都支持对其下层描述对象的创建、删除和查询，每个下层描述对象也支持对其上层描述对象的查询。而每个描述对象中则存放了由部署基础设施解析出的构件的语义信息、构件间的连接信息以及部署的物理位置等信息。因此任何系统信息都可从体系结构描述对象开始逐层查到。为提高查询效率，我们还在 SIL 中建立了有关构件连接等信息的索引。

系统信息计算机制负责根据系统的体系结构信息和语义信息，计算指定构件的 PS 或 ES 集合，从而用于生成动态配置算法。

3. 构件状态检测机制

为保证系统一致性，在动态配置过程中，对构件实施的操作可能存在前提条件，要求构件处于特定状态。构件状态检测机制则用于检测构件是否到达特定状态。根据第五章设计的动态配置算法，需要检测的状态包括，构件何时完成所有自行启动的事务，以及构件何时响应完毕所有指定类型的请求。相应的，构件状态检测机制由检测这些特定状态的子机制及辅助机制构成。

在构件自行实现的 passivateComp 方法中，在禁止构件自行启动事务之后，需要等待构件完成所有已经自行启动的事务。为辅助构件检测何时到达此状态，我们扩展容器，提供了构件完成所有自行启动事务的检测机制。作为该机制的辅助机制，我们首先提供了事务边界记录机制，记录构件当前启动的事务个数，具体实现方法：在容器的内部接口中加入 beginTran、commitTran 两个方法，并要求构件在启动事务前和完成事务后分别调用这两个方法。而这两个方法将分别对事务个数进行加 1 和减 1 的操作。基于事务边界记录机制，

容器只需检测构件当前启动的事务个数是否为 0，即可判断构件当前是否完成所有自行启动的事务。倘若构件当前尚未完成所有自行启动的事务，则检测构件当前参与的事务个数是否为 0 的过程还将反复进行。此检测过程可以主动方式或被动方式反复进行。在主动方式中，容器将间隔固定的时间启动检测过程。在被动方式中，每当 commitTran 方法被调用，容器就被触发启动一次检测过程。由于很难确定合适的状态检测间隔时间，因而我们选择了被动方式。线程间的同步机制被用于实现触发式的状态检测。采用阻塞方式驱动构件进入静止状态时，需要在驱动构件进入被动状态并阻塞后继到达构件的请求后，等待构件响应完毕已接收的应用类型请求。边界构件所在的容器实现 passivateClient 时，需要在阻塞或抛弃来自外部实体的请求后，等待边界构件响应完毕已接收的来自外部实体的应用类型请求。因此，我们提出了“构件响应完毕所有指定类型请求的检测机制”。为了区分配置类型请求、发自内部实体的应用类型请求和发自外部实体的应用类型请求，我们首先利用 CORBA 的截获器实现了请求类型标识机制。对于由控制边界内 CCM 构件发出的应用类型请求，容器将在其请求上下文中加入“COMPONENT”标识。DRM 和部署基础设施将把其发出的配置类型请求标记为“CONFIGURATION”。来自外部实体的应用类型请求或者不存在请求上下文，或者请求上下文中没有任何标识。通过这种方式，三类请求被区分开。类似于事务边界记录机制，我们通过对容器进行扩展，在目标构件接收请求和返回应答的相应时机，判断请求类型，并分别记录构件当前正在响应的发自内部实体的应用类型请求个数和发自外部实体的应用类型请求个数，此为构件响应请求信息记录机制。当 DRM 要求容器检测构件何时响应完毕所有指定类型请求时，容器只需判断当前正在响应的相应类型的请求个数，若为 0 则即刻返回。若不为 0，则容器将同样采用被动方式进行触发式检测。构件每响应完毕一个请求，容器即被触发启动一次检测过程。

4. 构件行为控制机制

构件行为控制机制用于真正实现对构件和连接的操作。由于我们充分利用了 CCM 构件平台的组装机制和部署框架，因此构件行为控制机制中的大量子机制直接来源于 CC 构件平台，包括构件创建/删除机制、连接创建/删除机制、构件属性设置机制和 configuration-complete 机制。另外，与 configuration-complete 机制完全对称的，我们还引入了 run-complete 机制。在此，我们就不对这些机制进行详细介绍了。

连接重定向机制用于在连接起始端构件处原子性的更新所保存的连接终止端的构件引用信息，从而保证引用一致性或实现连接重定向意图。之所以强调引用更新过程的原子性，是为了避免构件在旧引用信息被删除后、新引用信息尚未加入前试图获取引用信息而导致系统异常。本文通过对构件保存引用信息的数据结构进行加锁控制，保证了引用更新过程的原子性。

为了在保证引用一致性时，将基于旧目标构件引用发送的应用类型请求透明地重定向到新目标构件，我们利用 CORBA 的 Location Forward 机制实现了请求重定向机制。DRM 根据请求重定向需求，在容器中设置请求重定向条件，声明新、旧目标构件。容器在 preinvoke 方法中截获到请求时，如果经过判断发现该请求满足请求重定向条件，则抛出 CORBA 标准的系统异常 ForwardRequest 异常，内含新目标构件的引用。发送该请求的客户端 ORB 捕获该异常后，将自动地根据新目标构件引用重新发送请求。即使系统中不再存在满足请求重定向条件的请求，如果不及时删除过时的请求重定向条件，则容器还将在截获到每个应用类型

请求时判定其是否满足请求重定向条件。由于对请求重定向条件的处理位于构件接收请求的必经之路上，因此大量过时的请求重定向条件的存在，虽然不会引发处理错误，但对系统性能造成较大影响。所以为了及时删除过时的请求重定向条件，我们还引入了重定向取消机制。由于构件何时不再使用目标构件引用发送请求由构件实现决定，无法预期，因而我们只能向配置者提供一个删除请求重定向条件的接口，由其决定何时取消对请求的重定向。

在以阻塞方式驱动构件进入静止状态的动态配置算法中，需要阻塞所有应用类型请求。而在 passivateClient 方法的实现中，也需要阻塞来自外部实体的应用类型请求。相应的，在恢复系统正常运行时或在 activateClient 方法的实现中，应解除对请求的阻塞。因此我们引入了请求阻塞机制和解除请求阻塞机制。在阻塞指定类型请求时，DRM 在容器中设置请求阻塞条件，声明目标构件和被阻塞的请求类型。容器进而将请求阻塞条件设置给 ORB 的请求派发模块。从客户端发送出的请求在服务器端 ORB 中形成请求队列。ORB 在从请求队列中取出请求并派发给相应构件之前，将首先判断其是否满足阻塞条件，如果是则将其放置在被阻塞请求的队列中，否则将请求派发给构件响应。在解除被阻塞的请求时，DRM 只需删除相应的请求阻塞条件即可。在实现 passivateClient 方法时，容器利用请求拒绝机制在接收到来自外部实体的请求时返回异常。拒绝请求时，DRM 为容器设置请求拒绝条件，声明发向哪个目标构件的外部实体的请求必须被拒绝。容器在 preinvoke 方法中截获到请求时，如果经过判断发现该请求满足请求拒绝条件，则直接返回异常。类似于请求重定向条件，大量过时的请求拒绝条件的存在也将造成系统性能的极大浪费。因此，为了及时删除过时的请求拒绝条件，我们也引入了拒绝取消机制。但是何时外部实体不再发出到达指定边界构件的请求，则由系统实现决定，无法预期。因而，我们只能向配置者提供一个删除请求拒绝条件的接口，由其决定何时取消对请求的拒绝。

在本部分，驱动构件进入被动/主动状态的机制是动态配置算法实现的基础。引入开放问题后，需要进入被动状态的构件分为两种：内部实体、代表外部实体的 Client 构件。每个内部实体都必须实现 passivateComp 方法和 activateComp 方法。DRM 在合适的时机调用该方法，从而驱动构件进入被动状态和主动状态。构件开发者在实现 passivateComp 方法时，还可利用动态配置平台提供的构件状态检测机制，由容器辅助检测构件何时完成所有自行启动的事务。为了驱动 Client 构件进入被动状态和主动状态，容器必须实现 passivateClient 方法和 activateClient 方法。在 passivateClient 方法中，为了禁止 Client 构件继续发送请求，对于 Client 构件后继向边界构件发送的请求，容器根据配置者的要求或者利用请求拒绝机制直接抛出异常，或者利用请求阻塞机制将其暂时阻塞。然后，容器利用构件状态检测机制，检测构件何时响应完毕所有来自外部实体的应用类型请求，从而确认 Client 构件进入被动状态。若容器采用请求阻塞机制实现 passivateClient 方法，则在对应的 activateClient 方法中，容器只需利用解除请求阻塞机制，允许边界构件继续响应来自 Client 构件的请求即可。若容器采用请求拒绝机制实现 passivateClient 方法，则对应的 activateClient 方法无需执行任何操作。配置者只需在请求拒绝条件过时时将其及时删除即可。

5. 构件状态传递机制

在五类机制中，构件状态传递机制的实现比较简单。通过要求构件开发者根据需要自行实现 externalize 和 initialize 方法，并由 DRM 在合适的时机调用，这里提供了构件状态

外表化机制和构件状态初始化机制。由于构件状态的表现形式多样，因此这里用 CORBA 中的 any 类型封装具有不同内容和实际类型的构件状态。

6. 动态配置算法描述机制

为了支持动态配置意图的声明以及动态配置算法的生成和描述，动态配置算法描述机制由动态配置 GUI 和 RAG 两部分组成。动态配置 GUI 的设计和实现目标是尽可能支持配置者以直观、简单的方式声明动态配置意图，并在此过程中对动态配置意图本身进行合法性验证。利用系统信息收集机制，动态配置平台可获取当前完整的系统结构信息，因而可向配置者提供完整的由点、线组成的系统视图。其中，点代表构件，线代表构件间的连接。通过支持配置者对系统结构进行可视化的修改，可使动态配置意图的声明过程变得直观而简洁。对动态配置意图的合法性验证目前只能在语法层面上进行，如在两个类型不匹配的接口间建立连接将被视为非法。对动态配置意图在语义层 ARE 的深层次的合法性验证依赖于对系统行为的形式化描述和分析，而这方面目前还有待进一步的发展。

由于 CCM 构件平台组装和部署的实现都是基于 XML 格式的描述文件，因此为了最大程度利用 CCM 构件平台的原有功能、简化动态配置平台的实现，采用 XML 描述动态配置算法。文档类型定义文件（Document Type Definition，DTD）作为 XML 文件的模板，定义了 XML 文件中可描述的元素、元素的属性、元素的排列方式和顺序，以及元素中可包含的内容等。因此采用 DTD 文件描述动态配置算法模板，存放于 RAL 中。RAG 根据来自动态配置 GUI 的动态配置意图，读取 DTD 文件，利用 SIL 中的系统结构和语义信息，在算法模板中填入具体内容，从而生成相应的动态配置算法。

7.2.8 小结

本节主要介绍基于 StarCCM 的动态配置平台 StarDRP 的实现。首先对 StarCCM 优良的反射能力进行了分析，说明了选择 StarCCM 作为底层平台的原因及先进性。其后介绍了 StarDRP 的体系结构以及为支持调控协议所实现的动态配置机制，并说明了 StarDRP 于 RDRM 模型的对应关系。通过对负载均衡系统实施多种动态配置意图，验证了 StarDRP 的功能正确性，展示了 StarDRP 的动态配置能力，体现了动态配置技术本身为系统动态特性所提供的良好支持。

7.3 J2EE 平台的演化机制

7.3.1 构件管理框架

JMX（Java Management Extensions，即 Java 管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架。

JMX 是一套标准的代理和服务，实际上，用户可以在任何 Java 应用程序中使用这些代理和服务实现管理。Java 管理扩展 JMX 的前身是 JMAPI。Java 管理扩展 JMX 致力于解决分布

式系统管理的问题，因此，能够适合于各种不同的环境是非常重要的。为了能够利用功能强大的 Java 计算环境解决这一问题，Sun 公司扩充了 Java 基础类库，开发了专用的管理类库。

JMX 是一种应用编程接口，可扩充对象和方法的集合体，可以用于跨越一系列不同的异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统、网络和服务管理应用。它提供了用户界面指导、Java 类和开发集成系统、网络及网络管理应用的规范。管理对象是 JMX 应用程序的核心。JMX 结构包括：支持 Java 的 Web 浏览器用户接口，管理运行模块ARM（Admin Runtime Module）和应用。这三个部件之间通过RMI（Remote Method Invocation）进行通信。这里需要说明的是，RMI 使得一个 Java 虚拟机（JVM）上运行的程序可以调用远程服务器上另一个 JVM 上的对象。 用户接口用来发布管理操作，这些操作可以通过浏览器或通过单独的应用程序来间接激发。管理运行模块用来给应用提供实例化的管理对象。它包括 Agent 对象接口，通知接口和被管数据接口。应用指的是那些被管设备单元。

JMX 是一个完整的网络管理应用程序开发环境，它同时提供了：厂商需要收集的完整的特性清单，可生成资源清单表格，图形化的用户接口；访问 SNMP 的网络 API；主机间远程过程调用；数据库访问方法。

JMX 这一轻型的管理基础结构，价值在于对被管理资源的服务实现了抽象，提供了低层的基本类集合，开发人员在保证大多数的公共管理类的完整性和一致性的前提下，进行扩展以满足特定网络管理应用的需要。JMX 注重于构造管理工具的软件框架，并尽量采用已成熟的技术。

JMX 可以用来管理网络、设备、应用程序等资源。

1. JMX 的优点

（1）可以非常容易地使应用程序具有被管理的功能；

（2）提供具有高度伸缩性的架构；

每个 JMX Agent 服务可以很容易地放入到 Agent 中，每个 JMX 的实现都提供几个核心的 Agent 服务，也可以自己编写服务，服务可以很容易地部署、取消部署。

（3）主要提供接口，允许有不同的实现。

2. JMX 的体系结构

- JMX 的体系结构

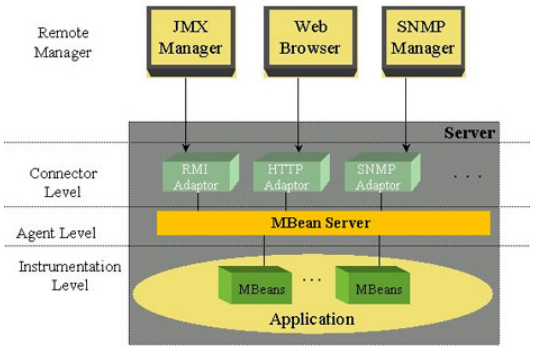


图 7-2　JMX 框架

（1）设备层（Instrumentation Level）：主要定义了信息模型。在 JMX 中，各种管理对象以管理构件的形式存在，需要管理时，向管理构件的 MBean 服务器进行注册。该层还定义了通知机制以及一些辅助元数据类。

（2）代理层（Agent Level）：主要定义了各种服务以及通信模型。该层的核心是一个 MBean 服务器，所有的管理构件都需要向它注册，才能被管理。注册在 MBean 服务器上管理构件并不直接和远程应用程序进行通信，它们通过协议适配器和连接器进行通信。而协议适配器和连接器也必须以管理构件的形式向 MBean 服务器注册才能提供相应的服务。

（3）分布服务层（Distributed Service Level）：主要定义了能对代理层进行操作的管理接口和构件，这样管理者就可以操作代理。然而，当前的 JMX 规范并没有给出这一层的具体规范。

（4）附加管理协议 API：定义的 API 主要用来支持当前已经存在的网络管理协议，如 SNMP、TMN、CIM/WBEM 等。

3. 设备层（Instrumentation Level）

该层定义了如何实现 JMX 管理资源的规范。一个 JMX 管理资源可以是一个 Java 应用、一个服务或一个设备，它们可以用 Java 开发，或者至少能用 Java 进行包装，并且能被置入 JMX 框架中，从而成为 JMX 的一个管理构件（Managed Bean），简称 MBean。管理构件可以是标准的，也可以是动态的，标准的管理构件遵从 JavaBeans 构件的设计模式；动态的管理构件遵从特定的接口，提供了更大的灵活性。该层还定义了通知机制以及实现管理构件的辅助元数据类。

4. 管理构件（MBean）

在 JMX 规范中，管理构件定义如下：它是一个能代表管理资源的 Java 对象，遵从一定的设计模式，并能实现该规范定义的特定接口。该定义保证了所有的管理构件以一种标准的方式来表示被管理资源。

管理接口就是被管理资源暴露出的一些信息，通过对这些信息的修改就能控制被管理资源。一个管理构件的管理接口包括：

（1）能被接触的属性值；

（2）能够执行的操作；

（3）能发出的通知事件；

（4）管理构件的构建器。

管理构件通过公共的方法以及遵从特定的设计模式封装了属性和操作，以便暴露给管理应用程序。例如，一个只读属性在管理构件中只有 Get 方法，既有 Get 又有 Set 方法表示是一个可读写的属性。

其余的 JMX 的构件，例如 JMX 代理提供的各种服务，也必须作为一个管理构件注册到代理中才能提供相应的服务。

JMX 对管理构件的存储位置没有任何限制，管理构件可以存储在运行 JMX 代理的 Java 虚拟机的类路径的任何位置，也可以从网络上的任何位置导入。

JMX 定义了四种管理构件：标准、动态、开放和模型管理构件。每一种管理构件可以

根据不同的环境需要制定。

（1）标准管理构件

标准管理构件的设计和实现是最简单的，它们的管理接口通过方法名来描述。标准管理构件的实现依靠一组命名规则，称之为设计模式。这些命名规则定义了属性和操作。检查标准管理构件接口和应用设计模式的过程被称为内省。JMX 代理通过内省来查看每一个注册在 MBean 服务器上的管理构件的方法和超类，看它是否遵从一定设计模式，决定它是否代表了一个管理构件，并辨认出它的属性和操作。

（2）动态管理构件

动态管理构件提供了更大的灵活性，它可以在运行期暴露自己的管理接口。它的实现是通过实现一个特定的接口 DynamicMBean。

JMX 代理通过 getMBeanInfo 方法来获取该动态管理构件暴露的管理接口，该方法返回的对象是 MbeanInfo 类的实例，包含了属性和操作的签名。由于该方法的调用是发生在动态管理构件向 MBean 服务器注册以后，因此管理接口是在运行期获取的。不同于标准管理构件，JMX 代理不需要通过内省机制来确定动态管理构件的管理接口。由于 DynamicMBean 的接口是不变的，因此可以屏蔽实现细节。由于这种在运行期获取管理接口的特性，动态管理构件提供了更大的灵活性。

（3）开放管理构件

开放管理构件是一种专门化的动态管理构件，其中所有的与该管理构件相关的参数、返回类型和属性都围绕一组预定义的数据类型（String、Integer、Float 等）来建立，并且通过一组特定的接口来进行自我描述。JMX 代理通过获得一个 OpenMBeanInfo 对象来获取开放管理构件的管理接口，OpenMBeanInfo 是 MbeanInfo 的子类。

（4）模型管理构件

模型管理构件也是一种专门化的动态管理构件。它是预制的、通用的和动态的 MBean 类，已经包含了所有必要缺省行为的实现，并允许在运行时添加或覆盖需要定制的那些实现。JMX 规范规定该类必须实现为 javax.management.modelmbean.RequiredModelMBean，管理者要做的就是实例化该类，并配置该构件的默认行为并注册到 JMX 代理中，即可实现对资源的管理。JMX 代理通过获得一个 ModelMBeanInfo 对象来获取管理接口。

模型管理构件具有以下新的特点：

① 持久性。定义了持久机制，可以利用 Java 的序列化或 JDBC 来存储模型 MBean 的状态。

② 通知和日志功能。能记录每一个发出的通知，并能自动发出属性变化通知。

③ 属性值缓存。具有缓存属性值的能力。

（5）通知模型

一个管理构件提供的管理接口允许代理对其管理资源进行控制和配置。然而，对管理复杂的分布式系统来说，这些接口只是提供了一部分功能。通常，管理应用程序需要对状态变化或者当特别情况发生变化时作出反应。

为此，JMX 定义了通知模型。通知模型仅仅涉及了在同一个 JMX 代理中的管理构件之间的事件传播。JMX 通知模型依靠以下几个部分：

① Notification，一个通用的事件类型，该类标识事件的类型，可以被直接使用，也可

以根据传递的事件的需要而被扩展。

② NotificationListener 接口，接受通知的对象需实现此接口。

③ NotificationFilter 接口，作为通知过滤器的对象需实现此接口，为通知监听者提供了一个过滤通知的过滤器。

④ NotificationBroadcaster 接口，通知发送者需实现此接口，该接口允许希望得到通知的监听者注册。

发送一个通用类型的通知，任何一个监听者都会得到该通知。因此，监听者需提供过滤器来选择所需要接受的通知。

任何类型的管理构件，标准的或动态的，都可以作为一个通知发送者，也可以作为一个通知监听者或两者都是。

（6）辅助元数据类

辅助元数据类用来描述管理构件。辅助元数据类不仅被用来内省标准管理构件，也被动态管理构件用来进行自我描述。这些类根据属性、操作、构建器和通告描述了管理接口。JMX 代理通过这些元数据类管理所有管理构件，而不管这些管理构件的类型。

部分辅助元类如下：

① MbeanInfo——包含了属性、操作、构建器和通知的信息。

② MbeanFeatureInfo——为下面类的超类。

③ MbeanAttributeInfo——用来描述管理构件中的属性。

④ MbeanConstructorInfo——用来描述管理构件中的构建器。

⑤ MbeanOperationInfo——用来描述管理构件中的操作。

⑥ MbeanParameterInfo——用来描述管理构件操作或构建器的参数。

⑦ MbeanNotificationInfo——用来描述管理构件发出的通知。

5. 代理层

代理层是一个运行在 Java 虚拟机上的管理实体，它活跃在管理资源和管理者之间，用来直接管理资源，并使这些资源可以被远程的管理程序所控制。代理层由一个 MBean 服务器和一系列处理被管理资源的服务所组成。

（1）MBean 服务器。Mbean 服务器为代理层的核心，设备层的所有管理构件都在其中注册，管理者只有通过它才能访问管理构件。

管理构件可以通过以下三种方法实例化和注册。

① 通过另一个管理构件；

② 管理代理本身；

③ 远程应用程序。

注册一个管理构件时，必须提供一个唯一的对象名。管理应用程序用这个对象名进行标识管理构件并对其操作。这些操作包括：

① 发现管理构件的管理接口；

② 读写属性值；

③ 执行管理构件中定义的操作；

④ 获得管理构件发出的通告；

⑤ 基于对象名和属性值来查询管理构件。

（2）协议适配器和连接器。MBean 服务器依赖于协议适配器和连接器来和运行该代理的 Java 虚拟机之外的管理应用程序进行通信。协议适配器通过特定的协议提供了一张注册在 MBean 服务器的管理构件的视图。例如，一个 HTML 适配器可以将所有注册过的管理构件显示在 Web 页面上。不同的协议，提供不同的视图。

连接器还必须提供管理应用一方的接口以使代理和管理应用程序进行通信，即针对不同的协议，连接器必须提供同样的远程接口来封装通信过程。当远程应用程序使用这个接口时，就可以通过网络透明地和代理进行交互，而忽略协议本身。

适配器和连接器使 MBean 服务器与管理应用程序能进行通信。因此，一个代理要被管理，它必须提供至少一个协议适配器或者连接器。面临多种管理应用时，代理可以包含各种不同的协议适配器和连接器。

当前已经实现和将要实现的协议适配器和连接器包括：

① RMI 连接器；

② SNMP 协议适配器；

③ IIOP 协议适配器；

④ HTML 协议适配器；

⑤ HTTP 连接器。

（3）代理服务。代理服务可以对注册的管理构件执行管理功能。通过引入智能管理，JMX 可以帮助我们建立强有力的管理解决方案。代理服务本身也是作为管理构件而存在，也可以被 MBean 服务器控制。

JMX 规范定义了代理服务有：

① 动态类装载——通过管理小程序服务可以获得并实例化新的类，还可以使位于网络上的类库本地化。

② 监视服务——监视管理构件的属性值变化，并将这些变化通知给所有的监听者。

③ 时间服务——定时发送一个消息或作为一个调度器使用。

④ 关系服务——定义并维持管理构件之间的相互关系。

- 动态类装载

动态类装载是通过 m-let（management applet）服务来实现的，它可以从网络上的任何 URL 处下载并实例化管理构件，然后向 MBean 服务器注册。在一个 M-let 服务过程中，首先是下载一个 m-let 文本文件，该文件是 XML 格式的文件，文件的内容标志了管理构件的所有信息，比如构件名称、在 MBean 服务器中唯一标志该构件的对象名等。然后根据这个文件的内容，m-let 服务完成剩余的任务。

- 监视服务

通过使用监视服务，管理构件的属性值就会被定期监视，从而保证始终处于一个特定的范围。当监视的属性值的变化超出了预期定义的范围，一个特定的通告就会发出。JMX 规范当前规定了三种监视器：

  - ◆ 计数器监视器，监视计数器类型的属性值，通常为整型，且只能按一定规律递增。
  - ◆ 度量监视器，监视度量类型的属性值，通常为实数，值能增能减。

◆ 字符串监视器，监视字符串类型的属性值。

每一个监视器都是作为一个标准管理构件存在的，需要提供服务时，可以由相应的管理构件或远程管理应用程序动态创建并配置注册使用。

● 时间服务

时间服务可以在制定的时间和日期发出通告，也可以周期性地发出通告，依赖于管理应用程序的配置。时间服务也是一个管理构件，它能帮助管理应用程序建立一个可配置的备忘录，从而实现智能管理服务。

● 关系服务

JMX 规范定义了管理构件之间的关系模型。一个关系是用户定义的管理构件之间的 N 维联系。

关系模型定义如下一些术语：

◆ 角色：就是是一个关系中的一类成员身份，它含有一个角色值。
◆ 角色信息：描述一个关系中的一个角色。
◆ 关系类型：由角色信息组成，作为创建和维持关系的模板。
◆ 关系：管理构件之间的当前联系，且必须满足一个关系类型的要求。
◆ 角色值：在一个关系中当前能满足给定角色的管理构件的列表。
◆ 关系服务：是一个管理构件，能接触和维持所有关系类型和关系实例之间的一致性。

在关系服务中，管理构件之间的关系由通过关系类型确定的关系实例来维护。仅仅只有注册到 MBean 服务器上并且能被对象名标志的管理构件才能成为一个关系的成员。关系服务从来就不直接操作它的成员——管理构件，为了方便查找它仅仅提供了对象名。

关系服务能锁定不合理关系类型的创建，同样，不合理的关系的创建也会被锁定。角色值的修正也要遵守一致性检查。

由于关系是定义在注册的管理构件之间的联系，所以当其中的管理构件卸载时，就会更改关系。关系服务会自动更改角色值。所有对关系实例的操作比如创建、更新、删除等都会使关系服务发出通告，通告会提供有关这次操作的信息。

JMX 关系模型只能保证所有的管理构件满足它的设计角色，也就是说，不允许一个管理构件同时出现在许多关系中。

（4）分布服务层。当前，SUN 并没有给出这一层的具体规范，下面给出的只是一个简要描述。

该层规定了实现 JMX 应用管理平台的接口。这一层定义了能对代理层进行操作的管理接口和构件。这些构件能：

① 为管理应用程序提供一个接口，以便它通过一个连接器能透明地和代理层或者 JMX 管理资源进行交互。

② 通过各种协议的映射（如 SNMP、HTML 等），提供一个 JMX 代理和所有可管理构件的视图。

③ 分布管理信息，以便构造一个分布式系统，也就是将高层管理平台的管理信息向其下众多的 JMX 代理发布。

④ 收集多个 JMX 代理端的管理信息并根据管理终端用户的需要筛选用户感兴趣的信

息并形成逻辑视图送给相应的终端用户。

⑤ 提供安全保证。

通过管理应用层和另一管理代理和以及它的设备层的联合，就可以为我们提供一个完整的网络管理的解决方案。这个解决方案为我们带来了独一无二的一些优点：轻便、根据需要部署、动态服务，还有安全性。

（5）附加管理协议 API。该层提供了一些 API 来支持当前已经存在的一些管理协议。

这些附加的协议 API 并没有定义管理应用的功能，或者管理平台的体系结构，他们仅仅定义了标准的 Java API 和现存的网络管理技术通信，例如 SNMP。

网络管理平台和应用的开发者可以用这些 API 来和他们的管理环境进行交互，并将这个交互过程封装在一个 JMX 管理资源中。例如，通过 SNMP 可以对一个运行有 SNMP 代理的交换机进行管理，并将这些管理接口封装成为一个管理构件。在动态网络管理中，可以随时更换这些管理构件以适应需求。

这些 API 可以帮助开发者根据最通常的工业标准来部署他们的管理平台和应用。新的网路管理的解决方案可以和现存的基础结构合为一体，这样，现存的网络管理也能很好的利用基于 Java 技术的网络管理应用。

这些 API 目前在 JCP（Java Community Process）内作为独立的 JSR（Java Specification Request）开发。

它们包括：

① SNMP Manager API

② CIM/WBEM manager and protocol API

（6）JMX 的当前实现及应用

自从 SUN 发布了 JMX 规范，许多大公司纷纷行动起来，实现规范或者实现相应的基于 JMX 的网络管理系统，下面列出了当前的主要实现及应用情况：

① SUN 为 JMX 规范作出了相应的参考实现，并在此基础上开发了一个全新的用于网络管理的产品 JDMK（Java 动态管理工具集），其中定义了资源的开发过程和方法、动态 JMX 代理的实现、远程管理应用的实现。同时，JDMK 也提供了一个完整的体系结构，用来构造分布式的网络管理系统，并提供了多种协议适配器和连接器，如 SNMP 协议适配器、HTML 协议适配器、HTTP 连接器、RMI 连接器。

② IBM Tivoli 实现了 JMX 规范的产品为 TivoliJMX，它为 JAVA 管理应用程序和网络提供了架构、设计模式、一些 API 集和一些服务。

③ Adventnet 开发的关于 JMX 的产品为 AdventNet Agent Toolkit，它使得定义新的 SNMP MIB、开发 JMX 和 Java SNMP Agent 的过程自动化。

④ JBoss 实现的 J2EE 应用服务器以 JMX 为微内核，各个模块以管理构件的形式提供相应的服务。

⑤ BEA 的 Weblogic 应用服务器也将 JMX 技术作为自己的管理基础。

⑥ 金蝶的 Apusic 也是一个以 JMX 为内核开发出的 J2EE 应用服务器。

6. JMX 的好处

（1）可减少对 JAVA 应用实施管理的投资

（2）提供了一个可伸缩的管理框架

（3）集成现有的管理方案：如：WBEM，SNMP，TMN

（4）使用现有的标准 JAVA 技术

（5）能使用未来的一些管理概念：如 Jini 连接技术、通用即插即用、服务定位协议（Service Location Protocol）

（6）只定义了一些可以访问的接口

7.3.2 J2EE 动态演化支撑平台

为了对 J2EE 平台下的构件演化机制做详细阐述，我们在这里介绍一个已有的基于 J2EE 的构件演化平台——PKUAS，该平台为北京大学自主研发、面向电力领域的一个成功案例。为了提供良好的开放性和灵活性，PKUAS 的设计基于以下原则：

（1）基于面向对象中间件：将目前的主流中间件划分为数据访问中间件、远程过程调用中间件、事务中间件、消息中间件和对象中间件。由于面向对象技术具有对构件的自然支持，因此，对象中间件是构建支撑运行平台的必然选择。

（2）结合 J2EE 和 CORBA：PKUAS 的目标运用环境是 INTERNET，这要求 PKUAS 必须具备跨异构平台的能力、多样化的客户端表现能力、强大的互操作能力以及良好的开放性。在现有的主流对象中间件中，J2EE 提供了良好的体系结构和完整的构件省模型，而 CORBA 则提供了强大的跨异构平台的能力和互操作能力。因此 PKUAS 符合 J2EE 体系结构、支持 EJB 构件件模型并集成 CORBA 互操作协议。

（3）支持 ABC 方法：ABC 方法提供了一种基于构件复用的从高层规约到最终实现可运行系统的系统化的开发方法，而特征驱动、软件体系结构指导、中间件支撑的构件组装与运行环境是支持 ABC 方法的主要工具。其中 ABCTOOL 应用于 ABC 的软件体系结构分析与设计、基于体系结构的构件组装阶段，主要功能包括构件池的管理、基于软件体系结构的图形化建模、自动合成系统的面向对象开发模型，作为 ABC 环境的部署和运行平台。PKUAS 允许用户在使用 ABCTOOL 完成系统建模和组装后，简捷、直接地将系统部署到 PKUAS 中。

1. 体系结构

借鉴操作系统微内核思想，PKUAS 通过抽取一组最基本功能形成一个内核，将平台内部的其他功能封装在各个相对独立的模块内，允许用户根据领域特征定制和扩展这些功能模块，在系统启动阶段有内核装配成领域特定的构件运行支撑平台。PKUAS 将平台自身的计算实体划分为四种系统构件。

（1）容器系统：容器是构件运行时所处的空间，负责构件的生命周期管理以及构件运行需要的上下文管理。在 PKUAS 内置的四种 EJB 容器中，一个容器实例管理一个构件的所有实例，而一个应用中所有构件的容器实例组成一个容器系统。这种组织模式有利于实现特定于单个应用的配置和管理，如不同应用使用不同的通信端口、认证机制和安全域。

（2）服务：实现系统的非功能性约束，如通信、安全、事务等。由于这些服务可通过微内核动态增加、更换、删除，因此，为了保证容器或构件正确调用服务并避免服务卸载

的副作用，必须提供服务功能的动态调用机制。对于供容器使用的服务，必须开发相应的截取器作为容器调用服务的执行点，对于供构件使用的服务，必须在命名服务中注册。

（3）工具：辅助用户使用和管理 PKUAS 的工具集合，主要包括部署工具、配置工具与实时监控工具。其中部署工具既能热部署整个应用，也可部署单个构件，从而实现应用的在线演化；配置工具允许用户配置整个服务器和单个应用；而实时监控工具允许用户实时观察系统的运行状态并作出相应调整。

（4）微内核：负责上述系统构件的装置、配置、卸载以及启动、停止、挂起等状态管理。与操作系统微内核不同的是，PKUAS 内核并不负责系统部件之间的通信，从而避免了整个系统性能的降低。

2. 关键技术

与其他 J2EE 产品相比，PKUAS 的特点在于开放、灵活的定制与扩展能力。其关键技术主要包括平台内核、互操作框架和元编程机制。

（1）平台内核。不同的应用领域既可能采用不同类型的容器或者容器的不同实现，也可能采用不同的分布通信协议，还可能使用不同的服务或者同类服务的不同实现。这样要求 PKUAS 平台自身必须具有良好的结构，以允许用户根据领域特点制定和扩展容器、服务、分布通信机制。通过分析现有的 J2EE 实现的体系结构以及其支撑构件运行的基本功能，PKUAS 抽象出一个平台内核，其主要功能是有效地管理容器、服务、管理工具等系统构件。与操作系统微内核负责其他模块之间通信不同，PKUAS 内核并不负责系统构件之间的通信，以免过度降低性能。系统构件之间的通信或者采用 JAVA 语言提供的对象调用机制，或者采用 PKUAS 提供的通信服务实现。

PKUAS 内核符合 JAVA 平台管理标准 JMX，继承了 JMX 可移植、伸缩性强、易于集成其他管理方案、有效利用现有 JAVA 技术、可扩展等优点。PKUAS 内核分为两层。

① 资源层：由可被 JMX 管理的系统构件组成，如容器系统、服务、管理工具等。这些被管理的构件通过 MBean 接口，对外提供与管理相关的属性和操作。MBean 可通过 JAVA 事件机制向外通知自身状态的改变或者其他资源的需求，同时 MBean 也可通过 JAVA 事件机制接受通知并作出相应的管理动作。此外，每个 MBean 还拥有一个元数据对象，以描述与管理接口相关的信息。

② 管理层：由负责注册资源的 MbeanServer 和管理资源的插件组成。MbeanServer 对外提供所有管理资源的接口，隐藏了这些资源的对象实例引用，从而允许管理资源动态的增加和删除。管理插件则是操纵注册资源 MBean 接口的构件，也是一个 MBean，因此，允许用户增加新的管理功能并动态增加到系统中。典型的管理功能包括类的动态装载、监视、定时器、资源关联等。

（2）元编程机制。现有中间件通过提供通用的、对开发者透明的分布系统特性，如分布性、安全性、事务性等，极大简化了分布应用的开发。随着网络环境与应用领域的发展，分布系统的适应性越来越重要，开发者往往需要在不改变现有设计与实现的前提下调整分布系统的行为。元编程机制作为一种有效的解决方案被引入中间件，该机制通过松散系统行为与资源、公用特性的紧耦合关系，提高系统的适应能力。为了提高平台对领域变化性的适应能力，PKUAS 提供了截取器和可编程客户桩两种元编程机制。截取器位于容器内

部，在调用请求进入目标构件之前被激活。多个截取器可以组成一个截取器链，根据请求中包含的信息依次执行特定功能，如认证、授权、审计、日志等。由于构件意识不到截取器的存在，因此可在不修改构件的前提下通过增加截取器来改变系统运行时的行为。截取器对于 PKUAS 服务的扩展至关重要，当增加供容器使用的新服务时，只需增加相应的截取器就可在容器运行时调用新增的服务。与 CORBA 截取器相比，PKUAS 的截取器类似请求截取器，作用于服务器端。而客户端可通过可编程的客户桩实现类似功能，即自己编写一个客户桩以增加新的客户功能或改变客户桩的行为。

7.3.3 小结

目前 PKUAS 主要作为 J2EE 应用服务器：支持三种 EJB 标准，包括无状态会话容器、有状态会话容器和 BMP 实体容器，可通过单个或多个构件的热部署提高应用的在线演化能力；与基于软件体系结构的建模工具 ABCTool 集成；通过反射能力提高平台的动态适应能力；实现电力领域特定的构件运行支撑平台的定制与扩展方案。

7.4 Web Services 和 SOA

Web Services 是自包含的、模块化的应用程序，它可以在网络（通常为 Web） 中被描述、发布、查找以及调用。Web Services 是基于网络的、分布式的模块化构件，它执行特定的任务，遵守具体的技术规范，这些规范使得 Web Services 能与其他兼容的构件进行互操作。

Web Services 定义了应用程序如何在 Internet 实现互操作，它极大地拓展了应用程序的功能，实现了软件的动态提供。同时 Web Services 技术为异构、自治和松散耦合的分布式应用提供了一个集成和交互机制。

传统的应用集成方法，如点对点集成、企业消息总线或 EAI、基于业务流程的集成等，都很复杂、昂贵，而且不灵活。这些集成方法难于快速适应基于企业现代业务变化不断产生的需求。面向服务的体系结构（Service-Oriented Architecture，SOA）是为了解决在 Internet 环境下业务集成的需要，通过连接能完成特定任务的独立动能实体实现的一种软件系统架构。

基于 SOA 的应用程序开发和集成描述了一套完善的开发模式来帮助客户端应用连接到服务上。这些模式定制了系统机制用于描述服务、通知及发现服务、与服务进行通信。从 Web Services 出发，SOA 集合了 Web Services 等基于行业标准的软件构件，灵活地将业务流程元素和基本的 IT 基础架构处理为安全、标准化、松耦合的服务，让企业用户可以重复使用并加以组合，以满足不断变化的业务优先级，从而提升企业的响应能力，并对业务变化做出快速而有效的反应。

7.4.1 Web Services 技术

Web Services 是指由企业发布的完成其特别商务需求的在线应用服务，其他公司、合

作伙伴的应用软件能够通过 Internet 来动态访问并使用这些在线服务。

Web Services 是独立的、模块化的应用，能够通过因特网来描述、发布、定位以及调用。在 Web Services 的体系架构中一般包括三个角色：服务提供者（Service Provider），服务请求者（Service Requestor）和服务注册器（Service Registry）。角色之间主要有三种操作：发布（Publish），查找（Find）和绑定（Bind）。服务注册包括对服务的描述，服务提供者包含服务描述和为服务请求者提供服务，如图 7-3 所示：

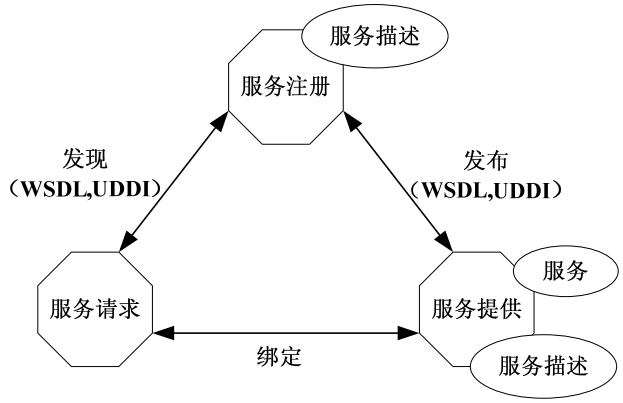


图 7-3　Web Services 模型

Web Services 应用中涉及两个部分：服务本身和对服务的描述。典型的 Web Services 应用过程是服务提供者开发一个通过网络可以被访问的服务，然后将服务的描述注册到服务注册器或者发送给服务请求者；服务请求者通过查找动作在本地或服务注册器中检索服务描述，找到后，通过绑定即可以使用该项服务。

Web Services 的构架可以通过一个层次模型来表示，如整体构架如图 7-4 所示。

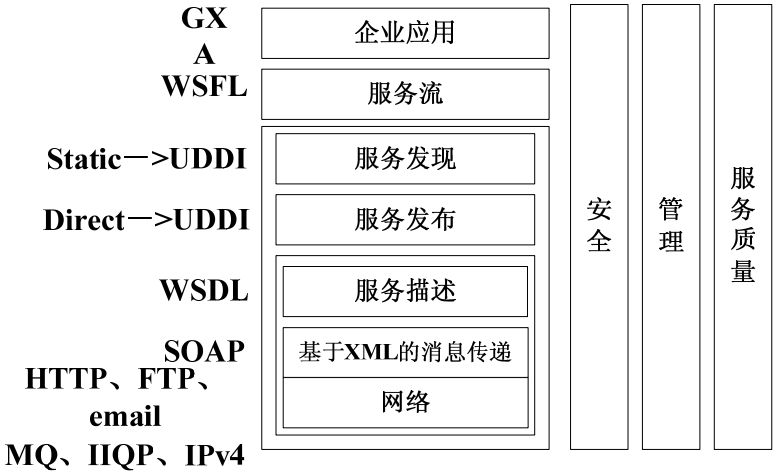


图 7-4　Web Services 协议栈

与 OSI 的七层网络协议构架图类似，Web Services 的整体架构层次图中上一层也需要

下一层的支持。而安全性（Security）、可管理性（Management）、服务质量（QoS）则需要在各个层次都有所体现。

整体架构示意图中涉及一些新的规范，如 UDDI（统一描述、发现和集成）、WSDL（Web Services 描述语言）、WSFL（Web Services Flow Language）、SOAP（简单对象访问协议）、GXA（全局 XML 架构）等。通过一个层次分明的架构，Web Services 力求实现动态的应用集成，将企业系统及电子商务推向智能和更加敏捷及实用的阶段。

语义 Web（Semantic Web）作为当前万维网的扩展，于 1999 年由 Tim Berners- Lee 等人提出，其目的是通过结构化和形式化，以表示 Web 上的资源（不仅限于 HTML 网页，而且包括所有可获取的数据及服务），使得计算机程序能够对网络资源进行分析和推理。图 7-5 是由 Tim Berners- Lee 在 XML2000 大会上提出的语义 Web 体系结构。这一模型得到了语义 Web 研究者的认同。在这个模型中包含了 7 层。第 1 层是 Unicode 和 URI。该层是整个语义 Web 的基础，其中 Unicode 是处理资源的编码，URI 负责标识资源；第 2 层是 XML，用于表示数据的内容和结构。这一层中包括 XML 和定义 XML 语法结构的 XML Schema，以及允许在文档中合成不同词汇的 XML 命名空间（Name Space，NS）。这些只是语义 Web 的语法基础；第 3 层为 RDF（Resource Description Framework）即资源描述框架，作为数据层，用于描述 Web 上的资源及其类型；第 4 层为 Ontology 层，用于描述各种资源之间的联系。语义 Web 中为服务提供语义的是本体 Ontology。Ontology 通过对概念的严格定义和概念与概念之间的关系来确定概念的精确含义，表示共同认可的、可共享的知识。Ontology 是解决语义层次上 Web 信息共享和交换的基础；第 5 层到第 7 层是在下面 4 层的基础上进行逻辑推理操作。其中核心层为 XML，RDF，Ontology。

目前广为使用的语义 Web 标记语言是 DAML+OIL，经过修订，它已被 W3C 接受为国际标准，即 OWL（The Web Ontology Language）。OWL 允许领域本体的创建和使用，为 Web 上的知识共享和集成提供了基础。

OWL 的子语言 OWL-S（前身为 DAML-S）用来描述语义 Web 服务，能够建立服务的分类体系和过程本体论。其中过程本体论是关于业务过程的共享知识，在语义 Web 服务中成为服务描述的基础，刻画服务是如何工作的，为服务是否满足需求执行深层次的分析、在服务合成、活动协调时提供依据。OWL-S 继承了 DAML-S，它也有几个高层本体组成，这些本体包括了构造描述服务的各个方面，表示本体的目标和用法。OWL-S 划分 Web 服务的语义描述为 4 个构件：Resource，ServiceProfile，ServiceModel，ServiceGrounding。图 7-6 是 OWL-S 的上层本体图。

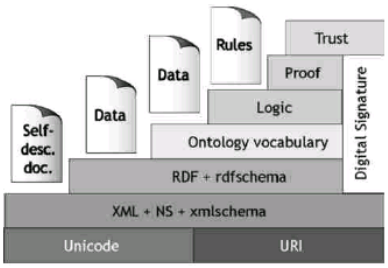


图 7-5 语义 Web 体系结构

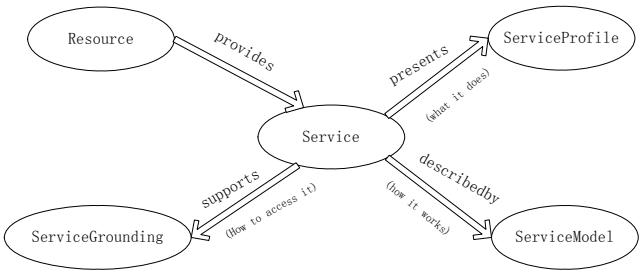


图 7-6 OWL-S 的上层本体

Resource 告诉了服务的来源。

ServiceProfile 告诉服务做什么，即给出发现服务代理需要的信息类型，以便代理决定服务是否满足需要，包括输入输出、前置条件、后置条件和绑定方式。此外，ServiceProfile 也提供服务的非功能性描述，如质量保证层、服务提供者的信息。ServiceProfile 为代理判定服务是否满足需要，是否满足约束提供信息。

ServiceModel 告诉怎么提供服务，即描述服务执行时会发生什么，为服务操作提供更详细的描述。发现服务代理至少可以以 4 种不同的方式使用这个描述：①执行更进一步的分析，看服务是否满足需要；②组合多个服务执行一个专门服务；③在服务设定中协调不同参加者的活动；④监控服务的执行。允许根据一些处理、输入、输出、前置条件和每一处理的结果进行描述。处理结构可能以递归方式建立。如服务可能包括一组子服务。

ServiceGrounding 设置具体的访问服务，主要处理协议和消息格式，序列化（serialization），传送（transport）和选择（addressing）。Grounding 可看作是与服务交互需要的服务描述元素规格从抽象到具体的映射。适合于自动服务选择代理并设置与服务通讯并调用服务。

7.4.2　SOA 基础

早在十年前，Gartner Group 就提出了面向服务软件体系架构概念。但是传统 SOA 的实现采用的都是一种紧耦合、非通用的接口设计，无法满足跨企业的分布式系统的信息共享，无法使软件得到最大限度的重用，不能实现实时系统，因而一直没有得到很好的应用。随着 W3C 对 Web Services 协议的规范化以及 IBM、Microsoft、SUN 等国际顶级 IT 大公司联合制定规范和服务支持，Web Services 技术日趋成熟化。Web Services 采用了一种面向服务（SOA）的开放的、松耦合的架构，所有协议都是基于 XML 具有通用性，并且实现简单。因此，Web Services 给 SOA 软件体系架构带来了新的契机，深入剖析了传统 SOA 架构技术和 Web Services 的原理、实现机制，并且分析了基于 Web 服务的 SOA 的实现，描述了基于 Web 服务的 SOA 的通信结构和实现层次结构。

从本质上来说，SOA 体现的是一种新的系统架构，SOA 的出现，将为整个企业级软件架构设计带来巨大的影响。SOA 本身就是一种面向企业级服务的系统架构，简单来说，SOA 就是一种进行系统开发的新的体系架构，在基于 SOA 架构的系统中，具体应用程序的功能是由一些松耦合并且具有统一接口定义方式的构件（也就是 service）组合构建起来的。因此，基于 SOA 的架构也一定是从企业的具体需求开始构建的。SOA 架构本质上来说体现了一种复合的概念：它不仅为一个企业中商业流程的组织和实现提供了一种指导模式，同时也为具体的底层 Service 开发提供了指导。

SOA 也是一种设计和构建松散耦合的软件解决方案的方法，松散耦合的解决方案能够以程序化的可访问的软件服务形式公开其业务功能，并使其他应用程序可以通过已发布的和可发现的接口来使用这些服务。通过应用 SOA，一个企业可以使用一组分布式服务来构成并组织应用程序，并能构造新的应用程序和修改现有的应用程序。Web 服务代表了面向服务的体系结构的一种实现，但并不能认为所有的 SOA 应用程序都是 Web 服务。

SOA 不是一种技术，更不是一款产品，它是一种企业 IT 架构，是从企业的需求开始

的。SOA 以“服务”为单元来组织企业 IT，SOA 能够帮助企业改善用户体验，并能更快、更有效地响应业务需求和商机。SOA 和其他企业架构方法的不同之处在于 SOA 提供的业务敏捷性。业务敏捷性是指企业对变更快速和有效地进行响应，并且利用变更来得到竞争优势的能力。

要满足这种业务敏捷性，SOA 的实践必须遵循以下原则。

（1）业务驱动服务，服务驱动技术

在抽象层次上，服务位于业务和技术中间。SOA 的设计一方面必须理解在业务需求和可以提供的服务之间的动态关系，另一方面，同样要理解服务与提供这些服务的底层技术之间的关系。

（2）业务敏捷是基本的业务需求

SOA 考虑的是下一个抽象层次：提供响应变化需求的能力是新的“元需求”，而不是处理一些业务上的固定不变的需求。从硬件系统而上的整个架构都必须满足业务敏捷的需求，在 SOA 中任何的瓶颈都会影响到整个工作环境的灵活性。

（3）一个成功的 SOA 总在变化之中

SOA 工作的场景是一个不断变化的活体，而不是一个静态的框架。SOA 的构架设计需要从传统的静态或相对静态的框架设计转向一个动态框架的设计。

SOA 是一种新的分布应用集成示范。在 SOA 体系结构上，它的基本框架由三个角色和三个基本操作构成。三个角色分别为服务提供者、服务请求者和服务注册中心。其中，服务提供者：发布自己的服务，并且对服务请求进行响应；服务注册中心：注册已经发布的 Web Services，对其进行分类，并提供搜索服务；服务请求者：利用服务注册中心查找所需的服务，然后使用该服务。面向服务的体系结构如图 7-7 所示。

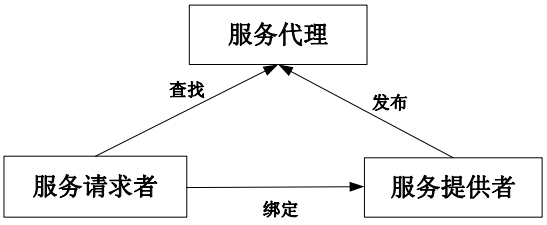


图 7-7　SOA 体系结构

三个基本操作分别为发布操作、查找操作和绑定操作。其中，发布操作：使服务提供者可以向服务注册中心注册自己的功能及访问接口；查找操作：使服务请求者可以通过服务注册中心查找特定种类的服务；绑定操作：使服务请求者能够真正使用服务提供者提供的服务。

为支持结构中的三种操作，SOA 需要对服务进行一定的描述，这种描述应具有下面几个重要特点。

首先，它要声明服务提供者提供的 Web Services 的特征。服务注册中心根据某些特征将服务提供者进行分类，以帮助查找具体服务。服务请求者根据特征来匹配那些满足要求的服务提供者。

其次，服务描述应该声明接口特征，以访问特定的服务。

最后，服务描述还应该声明各种非功能特征，如安全要求、事务要求、使用服务的费

用等。接口特征和非功能特征也可以用来帮助服务请求者查找服务。

7.4.3　SOA 与 Web Services 的联系

随着互联网的发展，网上交易、电子商务的逐渐繁荣，企业内部、企业之间的信息交流越来越依赖于 Internet/Intranet。随之而发展的 Web Services 为分布式计算提供了支持。Web Services 的平台和语言中立性使得跨平台的互操作、系统的整合更加容易。同时 Web Services 技术的成熟化使得 SOA 架构思想得到很好的应用。

Web Services 是当今最适合实现 SOA 的一些技术的集合，Web Services 标准的成熟和应用的普及为广泛实现 SOA 架构提供了基础。SOA 是一种结构模型，它可以根据需求通过网络对松散耦合的粗粒度应用构件进行分布式部署、组合和使用。服务层是 SOA 的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。SOA 的关键是“服务”的概念。

一方面，SOA 概念并没有确切地定义服务具体如何交互，仅仅定义了服务如何相互理解以及如何交互。其中的区别也就是定义如何执行流程的战略与如何执行流程的战术之间的区别。而另一方面，Web 服务在需要交互的服务之间如何传递消息有具体的指导原则；从战术上实现 SOA 模型是通过 HTTP 传递的 SOAP 消息中最常见的 SOA 模型。

Web Services 是面向服务体系结构的一个实例，Web Services 体系结构使用一系列标准和协议实现相关的功能，如使用 Web Services 描述语言（WSDL）描述服务、使用统一描述、发现与集成（UDDI）来发布和查找服务，使用简单对象访问协议（SOAP）来执行服务调用。Web Services 服务较其他分布应用集成方法最大的特点是它是完全低耦合的，服务请求者只有在需要服务时才动态地绑定服务提供者，其次，采用 SOAP 交互协议及 XML 作为消息格式具有跨平台特性，对集成的应用系统也要求较低，只要能够支持 SOAP 协议及处理 XML 文档就可以了。其主要协议内容如下：

SOAP 协议，是一组基于 XML 的无状态、单向、轻量级的消息传递协议，用于无中心、分布式远程过程的调用。

WSDL 协议，是把 Web Services 抽象地用 XML 描述为一组包含在面向文档或面向过程信息的消息上执行操作的端点的集合。

UDDI 协议，是一个基于 SOAP 协议的，为 Web Services 提供信息注册中心的实现标准，同时也包含一组提供 Web Services 注册、发现和调用的访问协议。

目前，SOA 集中在对服务组合、服务协同和服务管理方面的研究。其中，服务组合探讨了包括基于类型、QoS、有向图、Petri 网等多种服务组合方法；在服务协同方面，也出现了相应的标准，如 BPEL，BPEL4WS，WS2Coordination 等；服务管理包括通过包装遗留系统为 Web Services 的方法来协调管理集成系统和管理 Web Services 两方面。其中，管理 Web Services 还包括对基础层管理、应用层管理和业务层管理的研究。在对 SOA 集成方法的研究中，目前，服务协同主要是依据业务流程的方法而建立，具有单调性，还需要扩张多种协同方法，并且服务组合和服务协同的方法常常交叉覆盖。在管理方法方面，还没有相应的标准出现。在 SOA 方面，还有广泛的研究空间和待解决的问题，如在服务建立协同之后，如何保障服务的可靠性，换句话说，采用哪些机制能够保障服务的可靠性；随外

界环境的变化，组合服务又如何进化等。

最后，SOA 体系结构没有对 Web Services 的粒度进行限制，因此一个 Web Services 既可以是一个构件（小粒度），也可以是一个应用程序（大粒度）。

总体上来说，两者区别是把 SOA 看成一个架构模型，该模型被达到概念分解目的的独特方法所定义（与被面向服务的原则定义一样）；而 Web Services 是一种技术平台，它可以实现 SOA。但也存在其他的实现选择，只不过 Web Services 是现在实现 SOA 的最流行最合适的选择。

7.4.4 Web Services 的动态组合

Web Services 技术为异构、自治和松散耦合的分布式应用提供了一个集成和交互机制。但是单一的 Web Services 功能简单、有限，难以满足某些实际应用的需要，因此有必要对现有的单个 Web Services 进行合成，以生成功能更复杂、更强大的 Web Services 来支持各种应用需求。

按照现有的解决方案，可以利用现有的服务技术来完成相关的应用，建立服务并且对外发布，然后利用网络寻找这些服务，并且调用操作。但是，服务的查询、调用、信息的收集、条件判断以及操作流程仍为人工操作，将会耗费大量的时间和人力；另外，如果运用程序执行以上过程，则这个过程要预先定义好，并且一旦编码则不易更改。若情况稍有变化，比如说某个服务变更，则不得不重新编码来构造过程，使得程序流程不够灵活。因此，如果能在提出目标需求后，根据这个需求动态构造一个组合服务，将是非常高效的。

通常，Web Services 的动态合成过程主要涉及以下四个方面：

（1）服务注册。服务提供者向 UDDI 注册中心注册它们所开发的 Web Services。这一步是 Web Services 进行合成的前提，因为只有合成中可能用到的 Web Services 已经注册到 UDDI 注册中心，才可能完成用户所需要的 Web Services 合成任务。

（2）服务建模。由于 Web Services 本身是采用 WSDL 文件来描述的，而各个服务提供商在开发 Web Services 的时候可能会对 Web Services 中 WSDL 的各个元素采用不同的表示方式，所以在进行合成之前，必须先对这些 Web Services 进行统一的建模，以消除合成过程中可能遇到的语义问题，同时也可能需要从 Web Services 的 WSDL 文件中提取一些必要的合成信息。

（3）服务合成。在已经建立的 Web Services 模型的基础上，按照用户的要求利用某种算法自动生成或者手工合成满足需要的 Web Services 合成方案，然后把生成的合成方案转换为某种可执行的代码，如 BPEL4WS。

（4）合成服务的查找与执行。用户可以向合成引擎发出查询请求，而合成引擎则可以根据用户的查询要求查找到所需要的 Web Services，执行并返回结果。

目前国内外对 Web 服务集成的研究可概括为两种模型类型：第一种是从工作流或业务过程的角度，把 Web 服务看作工作流或业务过程中的活动；第二种是从软件工程的角度，把 Web 服务看作分布的软件构件，即基于构件的服务集成。

在 WSDL 基础上，过程定义语言如 BPEL4WS 支持将这些服务集成在一起以成为一个整体。BPEL4WS 的目标是在过程中将服务的引用从真正的服务实现中抽象出来，这将有助

于业务逻辑和服务相分离并实现服务的动态绑定。目前语义 Web 服务研究方兴未艾，直接应用语义 Web 服务语言 OWL-S（DAML-S）的过程模型和推理能力进行服务集成也得到广泛的研究。其中最受推崇的是由斯坦福大学 Mc Iraith 博士提出的一种智能 Web 服务实现模型。该模型建立在 OWL-S 语言之上的，也就是说，提供 Web 服务的系统都采用 OWL-S 来描述自身的 Web 服务，各个系统之间也通过 OWL-S 进行信息交换及服务集成。其优点是对 Web 服务的语义描述和 Web 服务的集成从一个视角看待，使用统一的语言处理。但无论是 BPEL4WS 还是 OWL-S，其本身并不解决从初始的需求构造业务过程规格的问题。

从软件工程的角度，为集成这些服务构件，构造软件体系结构作为集成蓝图成为必要。体系结构是在构件接口层次上刻画系统的集成信息，其中包含了构件接口规约和构件之间连接关系的集成信息。人们在进行软件总体设计时，首先根据用户需求和实现环境的要求，确定系统的体系结构；在体系结构层次上，构件接口规约对外唯一地代表了构件，故从服务构件库中查找符合接口规约的服务构件；最后，通过组装工具把构件接口处的集成映射到服务构件的集成，即把原子服务构件组装成复合服务构件或完整的系统。例如马晓星等使用图来描述体系结构并显式存在于应用环境中，为服务集成提供上下文；胡海涛等提出的 TROLL 方法利用服务组合模板以一种大粒度、可重用的方式组合业务服务，其模板封装的其实就是软件体系结构信息。

另外还有基于 PSM（Problem-Solving Method）的服务集成框架、基于计划的服务集成方法。基本的思想是把用户的需求表示为目标或问题，使用人工智能中的规划和推理技术形成达到目标或解决问题的方案，最终把方案映射到一系列的服务上，从而实现方案。WSMF（Web Services Modeling Framework）是欧盟委员会资助的知识技术领域的研究项目，提供了一个开发和描述语义 Web 服务的概念结构，允许服务的自动发现、互操作和集成。WSMF 计划将 OWL-S 扩展以涵盖 WSMF 的特征，并使用 UPML（The Unified Problem-Solving Method development Language）来构建业务方案和作为服务集成时的协调者（mediators）。UPML 是一种 PSM 开发语言，提供构件、适配器和连接配置的概念来描述和实现 PSM 框架。

总之，基于工作流的服务集成从业务过程的角度定义业务逻辑，关注系统的行为视图和控制流程，具有较好的灵活性和动态性，但忽略了系统的总体结构和难于复用成功的设计方案。基于构件的服务集成从宏观上规定了系统的实体结构和协作行为，易于引进专家经验和知识，提升了软件的复用性。

语义 Web 服务的研究目的是建立机器可以理解的 Web 资源，它的资源因而能够被自动工具如搜索引擎和人类用户共享和处理。当前 Web 服务作为一种新兴的 Web 应用模式和分布式计算模型，是 Web 上业务数据和信息集成的有效机制。SOAP、WSDL 和 UDDI 为 Web Services 的消息传输、服务定义和发现、发布制定了规范。这些规范共同使应用程序遵循一个松散耦合、与平台无关的模型来找到对方并进行交互。但是因为现在的一些 Web 服务仅能提供对文本的描述和提供人类使用的图形信息，本身并不具有智能，而语义 Web 要实现的是信息在知识级的共享和语义上的互操作性，所以在 Web 服务中有效利用本体论领域模型进行服务的概念建模，可以指导 Web 服务应用的设计；在 Web 服务中有效利用语义信息，进行 Web 服务和语义 Web 的有机结合可提高 Web 服务的质量；语义 Web 服务是语义 Web 和 Web 服务的有机结合，可为 Web 服务的发现、执行、解释和组合的自

动化提供有效的支持。通过对语义 Web 服务的研究，可以实现机器的智能化，实现服务的自动检索、动态匹配、集成等。其中，分布式环境下进行语义的提取、语义的表示、探索适合 Web 服务的知识推理方法等方面都是有待进一步研究的开放课题。

Web 服务动态组合是 Web 服务和语义 Web 发展的必然要求。

7.5 多 Agent 系 统

7.5.1 多 Agent 系统简介

Agent 在英语中主要有三个意思：一指能对其行为负责的人；二指能认识环境，并能对环境产生作用的行为者，在物理、化学、生物意义上活跃的东西；三指接受他人委托，并代表其行为的人，通称代理。计算机领域中的 Agent 的概念产生于分布式人工智能 DAI，用于表示构成分布式系统的相互协作的智能主体。分布式人工智能中的 Agent 在初始时期（1956－1985 年）经常被设计成符号 Agents，这种纯粹的表达式方法试图让 Agent 用清晰的逻辑推理来决定如何去完成自己的任务。符号推理的诸多问题引发了人们对它的反对，并形成一种以提倡不使用符号逻辑推理，而使用反应－学习模式构建 Agent 的运动，该运动由 1985 年持续到现在，被人们叫做“反应型 Agent 运动”。在 Agent 诞生的 30 年后（1986 年），MIT 的著名计算机科学家和人工智能的创始人之一 Marvin Minsky 扩展了 Agent 作为 DAI 中的一种人工智能程序的概念，在其“Society of Mind”一书中提出应将“社会”与“社会行为”引入计算机领域，将计算机从传统意义上封闭的、有一致性要求的计算上升到一个社会形态的计算，即开放性的、非一致性计算，从而使计算变为动态的和协作的过程。Minsky 将这样一个计算社会中的个体称为“Agent”。

关于 Agent 的定义在学术界有很多种，目前最被广泛认可的是英国利物浦大学的 Wooldridge 所作的关于 Agent 的定义，内容如下：

Agent 是具有下列特性的计算机系统：

（1）最主要的特性是自治性。能够独立行为，并且对外界展示对其内部状态的控制。

（2）反应性。可以理解周围的环境，并对环境的变化做出实时的响应。

（3）社会性。可以通过某种 Agent 通信语言 ACL 和其他 Agent（包括人）进行信息交流。

（4）能动性。可以主动地做出有目标的行为。

Wooldridge 称这个定义为 Agent 的弱定义，即一个约束比较小的定义。Agent 的强定义为：Agent 除了具备弱定义中所有的特征外，还应具备一些人类才具有的特性，如知识、信念、义务、意向等精神上的观念，还有情感和能力等因素。

作为由单个 Agent 相互协作构成的 MAS 系统，因其组成元素都是自治的，故而作为整体的系统也是自治的。同理 MAS 系统通过其组成元素也能理解周围环境，并对环境的变化做出实时响应，所以也具备反应性。由于 Agent 之间具有社会性行为，所以多个 MAS 系统之间也可以进行社会性协作，所以 MAS 系统也有社会性。同样的道理，MAS 系统也

能够主动的做出有目标的行为即具备能动性。可见 MAS 系统完全继承了 Agent 的必备特性，作为一个满足这些条件的计算机系统而言，根据 Agent 的弱定义，MAS 系统也是一种 Agent，一种特殊的 Agent。

MAS 系统继承了 Agent 的四个基本特征，作为一种协作系统，它还具备单体 Agent 没有的特征即系统体系结构的特征。MAS 系统是一种分布式系统，作为分布式系统节点的 Agent 通过相互间的信息传递来协作完成特定任务，故而它具备分布式系统的结构特征，除此之外 MAS 系统还具备一般分布式系统不具备的特性，主要有以下两点：

（1）动态性。即 MAS 系统的结构是动态的，原因是构成 MAS 系统的 Agent 之间的协作关系是动态的，这种关联随着环境、条件的变动而变动。

（2）适应性。即 MAS 系统结构的变化是为满足特性需求而发生的，通常是因为用户对系统提出了不同的任务要求，针对不同的任务要求系统的协作关系产生相应的变动。

7.5.2　多 Agent 系统的体系结构

随着对多 Agent 系统研究的深入和 MAS 系统应用的不断推广，逐渐出现了一些典型的多 Agent 系统应用，同时针对这些应用，相应的体系结构也成为同类应用的范例。本节主要介绍 FIPA 的 MAS 体系结构标准和 OMG 的 MAS 体系结构标准，其中 FIPA 已得到广泛应用。

1. FIPA 规范的体系结构

FIPA 是一个旨在建立异构软件 Agent 之间互操作标准的非营利性组织，建立于 1996 年，与世界上许多大的企业、组织有合作关系。目前按照 FIPA 标准开发的系统已经成功应用在许多领域。

FIPA 标准的多 Agent 系统结构上分为四个层次，底层是 Agent 消息传输层，它的构成如图 7-8 所示。

<table>
<tr><td>基于 Agent 的应用程序</td><td rowspan="4">抽象结构</td></tr>
<tr><td>Agent 通信</td></tr>
<tr><td>Agent 管理</td></tr>
<tr><td>Agent 消息传输</td></tr>
</table>

图 7-8　FIPA 规范的 Agent 结构

最底层的 Agent 消息传输层定义了一种消息格式，它由消息封套和消息体构成，如图 7-9 所示。它起到如下作用：

<table>
<tr><td>Agent1</td><td>Agent2</td><td rowspan="2"></td><td>Agent3</td><td>Agent4</td></tr>
<tr><td colspan="2">消息传输服务
（Message Transport Service）</td><td colspan="2">消息传输服务
（Message Transport Service）</td></tr>
<tr><td colspan="5">消息传输协议（Message Transport Protocol）</td></tr>
</table>

图 7-9　FIPA 标准的消息传输层

（1）能支持多种传输协议，例如：IIOP，HTTP，WAP 等。

（2）以特定方式套封消息，例如：XML 用于 HTTP 协议下的消息封装，bit-efficient 用于 WAP 下的消息封装。

（3）能够表达 FIPA 的 ACL，例如，使用字符串编码，XML 编码，bit-efficient 编码。

Agent 管理系统（Agent Management System）处理 Agents 的创建、注册、寻址、通信、迁移以及退出等操作，其结构如图 7-10 所示，它提供如下服务。

<table>
<tr><td>Agent1</td><td>Agent 管理系统（Agent Management System）</td><td>目录工具（Directory Faclilitatory）</td><td rowspan="3"></td><td>Agent2</td><td>Agent 管理系统（Agen Management System）</td><td>目录工具（ Directory Faclilitatory）</td></tr>
<tr><td colspan="3">消息传输服务（Message Transport Service）</td><td colspan="3">消息传输服务（Message Transport Service）</td></tr>
<tr><td colspan="7">消息传输协议（Message Transport Protocol）</td></tr>
</table>

图 7-10 FIPA Agent 管理层结构

（4）白页服务，比如 Agent 定位（寻址）、命名和控制访问服务。Agent 的名字被表示成一种灵活的可扩展的结构，这种结构被称为 Agent 标识，它包含了 Agent 的名称、传输地址、名称服务等相关信息。

（5）黄页服务，比如服务定位、注册服务等，此类服务由一个叫做目录 DF 的部分提供。

（6）Agent 消息传输服务。

Agent 的通信机制是一种基于通信谓词又叫通信断言的机制，其结构如图 7-11 所示，支撑这种机制的就是 Agent 通信语言 ACL。ACL 描述两部分内容，其一是通信的行为者，其二是通信的内容，并且支持上下文机制。FIPA 的 ACL 是在早期的 Agent 通信语言 ARCOL 和 KQML 基础上形成的。在内容描述方面，FIPA 使用一种内容语言作为 FIPA 语义语言，这些内容语言就是通常的约束选择语言比如 KIF、RDF 等。FIPA 交互协议描述了通过某些行为或者交互以完成某种目的而进行的对话。

图 7-11 Agent 通信语言（ACL）结构图

以上就是 FIPA 标准的主要构成，这种标准已经成功地应用于一些多 Agent 系统项目，相关的参考可参见 FIPA 官方网站 http://www.fipa.org。

图 7-12 给出了 FIPA 标准的多 Agent 系统应用的一种抽象模型。

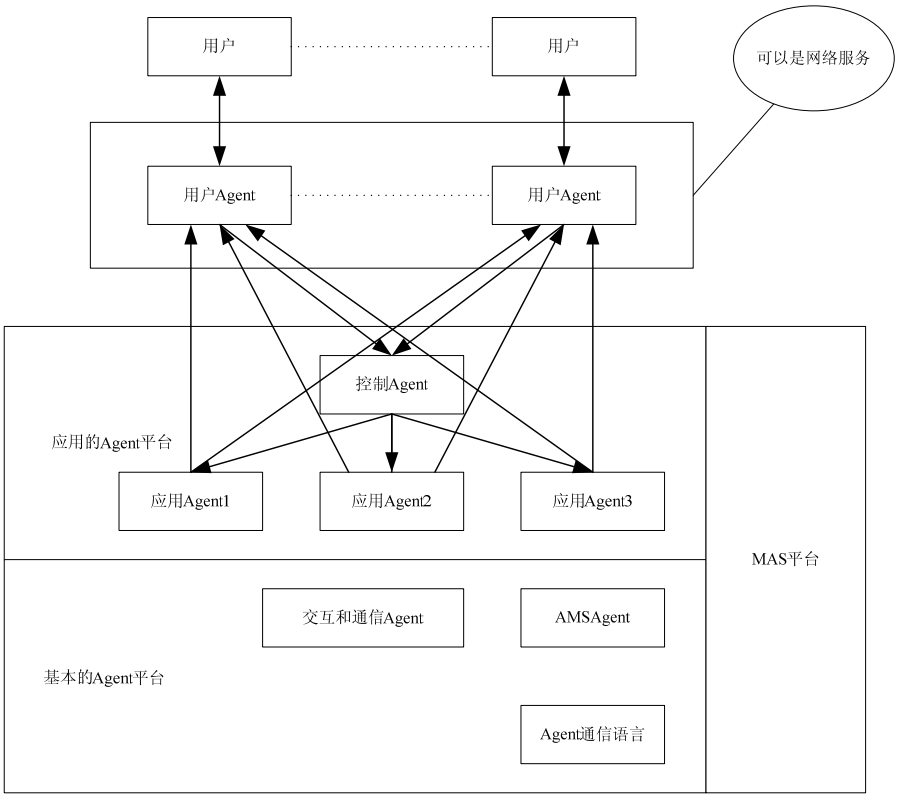


图 7-12　遵循 FIPA 规范的 MAS 抽象体系结构

其中应用层的 Agent 充当了主要的问题解决者，包括问题的分解、协作、综合等，但它们不再考虑如何通信、如何协同、如何理解其他 Agent 的消息等等问题。这就是 FIPA 标准的最大优点，它在 Agent 底层封装了所有应具备的基本功能，就如同人的本能一样。这些通信协同能力随着 Agent 的产生而产生，而对于上层的应用开发者根本不需要知道那些与具体应用毫不相关的通信、协同机制，而仅仅是把自己的 Agent 从标准库中的原始 Agent 继承下来，使用诸如 Send，Receive 等如同命令一样的函数或者操作原语来完成特定应用的需求。这些为多 Agent 系统的推广，以及进一步深入研究创造了良好的条件。

2. OMG 体系结构

OMG 致力于使用 OMG 的方法来标准化 Agent 和多 Agent 系统。他们认为 Agent 技术不是一种独立的、新技术，而是多种技术的集成应用，同样它也不是一种独立的应用，它可以为现有应用增加新的功能。同时未来的 Agent 可能成为操作系统的一支，而人机交互将会因 Agent 的产生而大大加强。

OMG 将多 Agent 应用分为以下几种。

（1）企业级应用，主要包括比如智能文档，面向目标的企业规划，动态从事管理。

（2）交互级企业应用，主要包括产品或者服务的市场拓展，代理商管理，团队管理。

（3）过程控制，包括智能大厦、工厂管理、机器人等。

（4）信息管理任务，包括信息检索，信息过滤，信息监视，数据资源调节，Agents 和个人助手程序间的交互。

这些基本涵盖了目前 Agent 系统的应用范围，同时 OMG 还给出了一种多 Agent 系统的参考结构（http://agent.omg.org），可以认为这就是 OMG 标准的雏形，事实上，由于对 MAS 系统标准化的研究工作是很多相关组织合作进行的，其中就包括 OMG 和 FIPA，当 FIPA 率先推出了自己的标准后，OMG 的研究就受到一些阻碍，因为它必须使自己的标准优于 FIPA，才能可能得到广泛认可，故而目前尚没有 OMG 的相关标准出台。

7.5.3 多 Agent 系统的动态性分析

MAS 由多个独立而又互动的 Agent 构成，每个成员 Agent 仅拥有不完全的信息和问题求解能力（因而其布点是有限的），不存在全局控制，数据是分散或分布的，计算过程是异步、并发或并行的。对于不同的应用系统，Agent 被赋予不同的能力或者功能，依据不同的功能这些 Agent 分别扮演不同的角色。Agent 间的通信建立在一组特定的协议之上，通过特定的语言交互信息，Agent 通过协作完成系统规定的任务，从而在一定的任务条件下，Agent 之间形成某种交互依赖关系。根据软件体系结构的理论，Agent 可以等同于能完成特定功能的大粒度或小粒度的构件，连接件和配置等功能都被封装在了 Agent 实现的底层，它们通过改变 Agent 之间的关系来进行任务调试。在系统运行的某一时刻 MAS 系统的体系结构是确定的，在整个系统运行期间 MAS 系统的体系结构将处于不停的变化当中，而这种变化取决于不同的任务分配，所以 MAS 的体系结构是动态体系结构，它能随不同任务而变化，故而同时具有适应性。

MAS 体系结构的动态性主要表现在：

（1）Agent 自身结构的动态性：Agent 是具有一定智能的实体，具有自适应能力，能通过不断学习和反省，更新自己的知识和能力，调整自身的行为规范及行为。Agent 所扮演的角色由它在 MAS 中所起的作用即向外界提供的服务而定。当发生角色转换时，Agent 向外界提供的服务即 Agent 能力或功能需要进行适当的调整，Agent 自身的结构必须相应的发生变化以适应其角色的转化。

（2）Agent 间通信结构的动态性：在 Agent 的不同阶段，以及在完成不同的任务时，其通信对象都可能发生变化。另一方面，随着 Agent 角色的转换，如从服务者变成被服务者，Agent 与其他 Agent 间的合作关系也会发生相应的变化，也就是说，Agent 间的通信结构发生了变化。

下面从 Agent 间通信结构和系统结构的动态性两方面来分析遵循 FIPA 标准的 MAS 体系结构的动态性。

从 Agent 间通信结构的动态性角度看：在定义用户 Agent（User Agent）的请求时，用户 Agent 必须通过控制 Agent（Controller Agent）才能得到所请求的服务，而控制 Agent 在接收到用户 Agent 的请求后，首先查询各应用 Agent（AppAgent）能提供的服务，然后

将控制 Agent 与应用 Agent 间的连接转接到用户 Agent 与应用 Agent 间。在此过程中，首先是用户 Agent 与控制 Agent 之间在进行通信，用户 Agent 与应用 Agent 间尚不存在连接。然后，用户 Agent 和应用 Agent 间建立起新的连接，此后应用 Agent 将服务传递给用户 Agent 需要服务的地方。这表明在整个通信过程中，用户 Agent、控制 Agent 与应用 Agent 之间的通信结构发生了变化。上面描述的这种通信结构的变化是在通信过程中发生的临时性变更，即用户 Agent 一旦得到应用 Agent 所提供的服务后，与应用 Agent 间的连接就会自动断开。但当用户 Agent 通过学习知道谁能为它提供服务后，就不再需要通过控制 Agent 来查询应用 Agent，而是直接与应用 Agent 建立请求连接，它表明通信结构发生了变化。

从系统结构的动态性角度看：在遵循 FIPA 标准的多 Agent 系统中，系统的结构会随着时间的推移而演变。譬如用户 Agent 通过不断学习，当它充分了解谁能向它提供服务后，就可以直接与应用 Agent 进行通信，而不再经过控制 Agent，这时系统模式就可能退化成典型的客户机/服务器模式。与此同时，应用 Agent 可能会不断向控制 Agent 学习，即请求控制 Agent 提供知识服务；而另一方面，为了保证系统的可靠性，控制 Agent 可能会主动地将自己的知识和能力传授给其他 Agent。这样一旦当某个 Agent 具有与控制 Agent 相当的能力，即能完成控制 Agent 所承担的任务，那么该 Agent 就可以进一步进化为控制 Agent，而进入 Agent 控制层，这样即使出现某个控制 Agent 无法履行其职责的情况，也会有其他 Agent 及时地顶替它来负责相关事务。这种 Agent 角色的转变也是系统结构的一种变化形式。

另外，控制 Agent 根据用户请求，选择可以满足用户需求的应用 Agent，并给出应用 Agent 的协作方式，从而形成通信结构。用户请求内容不同，所需的应用 Agent 及方式也不同，于是在系统运行期间就形成了动态变化的适应性的体系结构。

因此，可以说，这种遵循 FIPA 标准的 MAS 体系结构是动态的。

7.5.4　Web Agent

Web Agent 是在智能 Agent 的理念基础上，结合信息检索、搜索引擎、机器学习、数据挖掘、统计等多个领域知识而产生的用于 Web 导航的工具。

目前已经有许多的 Web Agent 实验系统存在，有些已经出现在人们日常访问的网站中。比较著名的有：Web Watcher 和 Personal Web Watcher，Syskill&Webert，WebMate，Letizia 等。下面以卡耐基-梅隆大学的 Web Watcher 为例对这种系统进行简要介绍。

Web Watcher 搜集训练事例的方式是这样的：Web Watcher 记录用户从登录到服务器开始一直到退出系统或退出服务器，用户浏览过的页面序列，点击过的超链序列，以及它们的时间戳。在退出系统或服务器之前，Web Watcher 会询问用户是否达到目标，即要求用户对此次浏览给出一个二值的评价，即成功或者失败。这种事例对同一时刻连接服务器的成千上万的用户都会产生，Web Watcher 就是通过对这种大量的训练事例的分析，得出当前大多数用户普遍的浏览方式。

当一次新的浏览开始后，Web Watcher 对用户的引导是基于大多数用户过去的浏览经历。即这种向导是建立在一种非常经典的预测理论上的，那就是大多数用户经历了这样一种浏览过程，那么就暗示当前用户很有可能进行类似的选择。Web Watcher 对用户的导航，是对用户浏览当前页面上的超链进行推荐。实验证明，Web Watcher 的推荐精度，远远超

过了不考虑任何用户模型情况下的随机推荐的情况。

此类 Web Agent 系统的体系结构如图 7-13 所示。

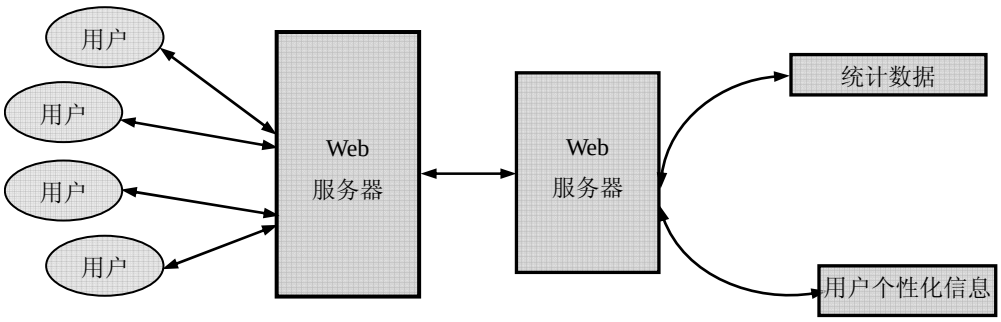


图 7-13 Web Agent 服务结构

这里的 Web Agent 就好像一个过滤器或者一个监控程序一样，从 Web 服务器上获取用户的访问信息，对其进行统计处理，经过算法的加工成为用户访问网页的一种个性化信息，Agent 再拿这些个性化信息反过来服务于用户，而用户在这里无形中起到一种训练作用，在自己访问网络的同时完成了对服务 Agent 的训练。大量的用户访问，使得 Web Agent 能够全面掌握访问网站用户的习惯信息，而且能够在一些新用户刚登入不久就可以提供出用户满意的推荐。故而这种 Agent 在网络中有着很好的应用前景，同时作为 Agent 系统而言很少是单个 Agent 独立完成任务的，对于助手类 Agent 由于其 Agent 本身的协作特性，也存在网络中多个 Web Agent 合作的情况，也就是说一个 Agent 存在于某个特定网站的服务器端或者某个客户端，都不能完全满足用户所有的网络访问需求，它们之间通常都要互通有无，通过协作来提供用户所需要的信息。从整个服务体系或者系统而言，其系统结构仍然是一种动态体系结构，如图 7-14 所示。其中 Web 服务器用虚线表示其为客户端 Agent 与服务端 Agent 之间的一种透明的交互媒介，而作为服务端 Agent 也存在为协作而进行的交互。

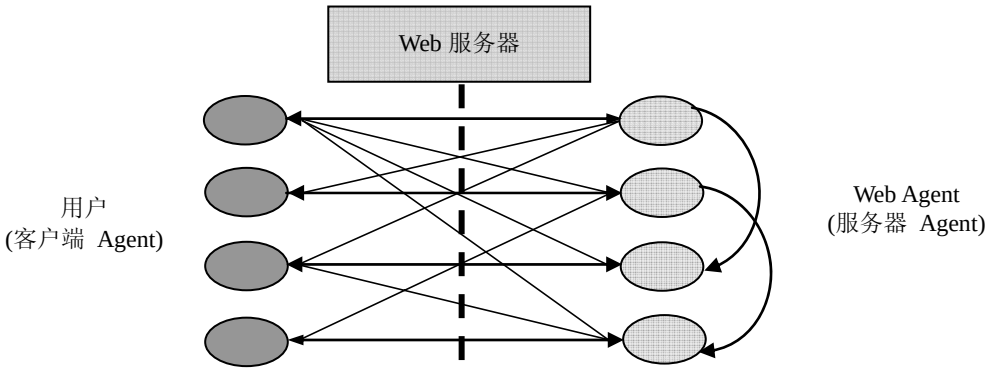


图 7-14 Web Agent 构成的多 Agent 系统结构

第 8 章　与动态演化技术相关的应用

目前软件动态演化技术还处于研究阶段，离产业化应用还有一段距离，但也有很多领域开始应用软件的动态演化技术为其服务，如自治计算、网格计算、普适计算以及自适应中间件等。本章主要介绍软件动态演化技术在以上这些主流研究领域的应用，并简单介绍这些服务的一些基本概念。

8.1　自 治 计 算

2001 年 IBM 发布的一个预测报告显示，未来人类在 IT 业的挑战主要来自于软件复杂性危机。该声明指出管理现在的计算机系统的困难已经超出了单独的软件环境的管理。计算系统的复杂性似乎已经达到了人类的极限，然而对这种大型复杂软件的需求却并未减弱。许多系统为了能提高数据的共享能力，都要求将系统进行整合，从而在很大程度上加大了软件的复杂度，导致系统管理变得异常复杂。

8.1.1　自治计算的概念

随着面向对象技术、构件技术的发展，软件系统的复杂度和规模日益剧增，同时随着 Internet 的发展，软件系统所处的环境动态变化，如计算实体的种类增多，除了传统的 PC 机等还添加了嵌入式实时设备、便携式移动设备，用户需求动态变化受到资源的限制。所以，管理和维护软件系统的负担剧增。为了降低负担，人们考虑让软件系统能够自我维护、自我管理，根据环境的变化调整自己的行为，继续运行，即软件系统具有自适应性。所谓自适应（self-adaptation）就是软件能够感知环境的变化，并根据环境的变化改变自身行为，采取适应性动作，以适应资源的可变性、用户需求的变化以及系统错误。目前，自适应的研究主要是外部适应性（external adaptation），自适应部件位于系统外部，与功能部件分离，通过从运行系统获取和度量变化，分析变化，采取自适应动作作用于系统，从而形成一个封闭的控制环。外部适应性中自适应部件与功能部件分离，所以与应用的耦合度较低，可以修改和复用。同时，由于它是对整个系统进行监控，能实现全局的适应，而不仅仅局限于局部处理。

自治计算的思想来源于人体生物学，自治的神经系统监控人的心跳、检查血糖水平、保持身体温度接近于 37℃，而不需要有意识的努力。同样的，自治计算组件在最少人为干预的情况下参与解决计算机系统的需求和问题求解。然而，在人体中与在计算机系统的自主响应中，它们的自主活动存在重要的区别。人体中自治元素的许多决策是自然发生的，而计算机系统中自治元素的决策是基于人委托给它的特定任务。换句话说，是可修改的政

策而不是硬性的编码，决定计算机系统中自治元素的决策类型和动作。

目前，IBM 的研究者将自治计算中的自我管理机制例示为四个方面的能力（特性）：自配置、自修复、自优化和自保护。这四个能力是相互关联的。例如，如果一个自治计算系统能够通过配置和重配置自己来响应自身所处环境中变化的和不可预测的情况，那么当这些情况是未知的入侵和恶意的攻击时，它应该具备自保护的能力。同时，自配置的能力也需要自治管理系统去学习，这将促使整个计算系统的自优化。另外，当遇到不可预测的事件，这些事件可能引起系统的某些部分失常时，自配置能力应该允许自治管理系统去恢复和修复自己。下面是这四个能力的具体含义。

（1）自配置——自动地适应于动态变化的环境。自配置构件使用 IT 专家提供的政策，自动适应 IT 系统中的变化。这些变化可能包括新构件的部署、现有构件的移除，或者工作负载的急剧增加或减少等。

（2）自修复——发现、诊断和修复故障。自修复构件可以检测系统故障，并启动基于政策的修复活动。

（3）自优化——自动监视和调整资源。自优化构件能够自我调整以满足终端用户或商业的需求。调整活动可能意味着重新分配资源来改善系统的总体效用或者确保特定的商业交易能够及时完成。

（4）自保护——预见、监测、识别、保护来自各处的侵袭。自保护构件能够在敌意行为发生时监测到它们，并采取行动使得自己不易受攻击。敌意行为可能包括未经授权的访问和使用、病毒传染与繁殖，以及服务拒绝入侵等。

在自治计算系统中，以上四个自我管理的能力是通过一个智能控制循环来实现的。这个循环从系统收集信息、决策、然后根据需要调整系统，如图 8-1 所示。

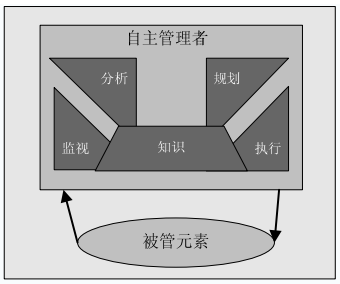


图 8-1　自主元素结构

在图 8-1 中，控制循环（自治元素）由两个主要元素构成：被管元素和自主管理者。被管元素是自主管理者管理的对象，比如服务器、数据库、路由器、Web 服务、网格服务等，也可以是其他自治元素。自主管理者是实现特定控制循环的组件，比如 Agent。它由五个部分组成：监视、分析、规划、执行和知识。其中，“监视”部分提供机制来搜集、合成、过滤、管理和报告来自被管元素的状态细节；“分析”部分实施情景分析；“规划”部分负责组织动作以实现目标；“执行”部分控制规划的执行；以上四个组件使用的数据作为共享的“知识”存储在知识库中，共享的知识包括拓扑信息、系统日志、性能尺度和政策。其中政策由一组行为约束或偏好组成，它们影响自主管理者的决策。明确地，自主管理者的“规划”部分负责解释和翻译政策细节。而“分析”部分则负责决定自主管理者是否能够遵守政策。自主管理者依据高层政策尽可能有效地完成任务。

自主管理系统由自治元素组成。自治元素之间的关系可以是 P2P（ Peer-to-Peer）的或层次结构的。比如，在由 Agent 构成的自主管理系统中，担任自治元素的 Agent 之间可能通过签订合同构成一对一的契约关系，也可能组成嵌套的 Agent 联邦，形成层次关系。

8.1.2　自治计算的特征

一个系统被归类到自治系统，它应具有下列主要特征。

（1）自保护。一个自保护系统可以检测和识别敌对的行为，采取自治的动作保护自己、抵御入侵行为。可以想象，自保护系统能够保护它们自己来抵御两种类型的行为：由人造成的意外错误和恶意的动作。①保护它们自己抵御由人造成的意外错误，例如，当系统管理员发出一个能够中断服务的进程时，自保护系统可以提供一个警告。②保护它们自己抵御恶意的动作，自保护系统将扫描可疑的动作，并且对可疑动作做出反应而用户并不知道在过程中有这样的保护。除了简单地对构成部件出错做出响应或为故障运行周期性检查外，一个自治系统总是保持警惕，预期可能的威胁并预备采取必要的动作。通过基于用户角色和预先建立的策略，确定准予访问的动作，自治系统也用于为授权用户在授权期间提供准确的信息。

（2）自优化。一个自治系统中的自优化组件能动态地调整自己，以最小化的人为干预来适应端用户或企业的需要。调整动作包括基于负载平衡功能和系统运行状态信息的资源重新分配，以提升整个的资源利用率和系统性能。

（3）自修复。自修复是指当系统中的部分部件出现故障后，系统能够进行自我恢复的能力。对一个自恢复的系统来说，它必须能够借助第一次检测从失效部件中恢复，并且隔离有故障的部件，使它离线，修复和重新装入已修好的或替换部件投入服务，而不会造成任何明显的全面中断。一个自修复系统也需要预报问题并采取动作防止可能会影响到应用的故障发生。为了保持系统正常运转和在所有时间都是可用的，自修复目标是使所有的消耗最小化。

（4）自配置。安装、配置和集成一个大的、复杂的系统是非常具有挑战性的。它既耗费时间，同时也很容易出错。自配置系统能够自动地适应动态变化的环境，环境中的系统组件包括可以动态增加到系统的软件组件和硬件组件，并且要求不中断系统服务和最小化人工干预。

（5）开放标准。一个自治系统将基于一个开放的标准，并且提供与其他系统互操作的标准方法。

（6）自学习。一个自治系统将与机器学习构件集成起来，可以基于一个系统运行的某段时间建立知识规则来改进系统性能、健壮性和灵活性，以及预知可能出现的故障。

（7）自适应。一个自治系统能自动找到最佳的方式来与相邻的系统进行交互，例如，它能够向其他系统描述其自身，同时还能找到该运行环境中的其他系统，从而适应环境的变化。

（8）隐藏复杂性。一个自治系统能够隐藏其本身的复杂性，从而提高了系统的透明性。例如它能自动地完成一些基础性的任务，从而减少了用户的管理工作。

8.1.3　动态演化在自治计算中的应用

自治计算本身的特性决定了它必须使用动态演化技术来实现其特有的功能，如自配置、自适应等功能都需要根据外界环境的变化来动态完成。因此构建一个自演化的软件系统（如图 8-2 所示）主要包括以下四个部分：

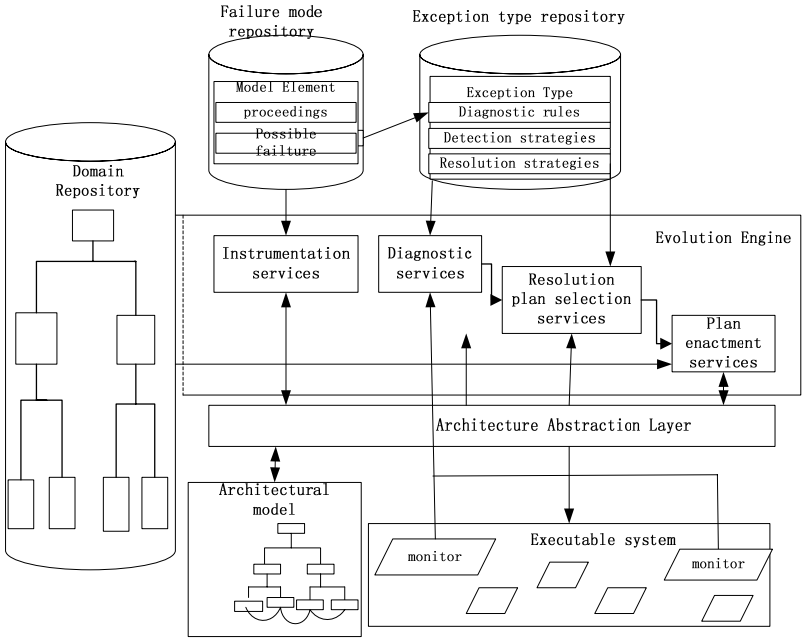


图 8-2 体系结构构成

（1）领域知识库，它包括一个特定的应用领域中可供选择的方法的集合。尽管这些可供选择的方法是给定的，由于在设计期不能提供足够的信息来决定该方法集合中到底哪一个方法是最适用的，因此就需要构建领域知识库。例如，在自动目标识别领域中，只有在导弹发射之后才能够决定哪一个算法是最可靠的。构建这样一个特定领域的自演化系统，前提是在领域知识库中已经包含了足够多软件设计期间的信息。比如在特定应用领域下的软件配置的结构、行为和设计权衡（性能、需求、局限性等等）。

（2）体系结构模型，它反映了运行系统在任何特定时刻的目的、结构和设计原理。在生成实际的软件系统之前，需要做一些初始化设置，将构件组合在一起，产生体系结构模型。除了构建领域知识库之外，体系结构模型需要包括系统当前的配置信息以及系统的目标行为和设计原理。演化引擎充分利用这个模型来推测一个计算想要得到什么样的目的，并监视是否成功。

（3）演化引擎，它监视系统是否达到它的目的，检测异常情况，诊断潜在的问题，并选择解决他们的方法。演化引擎监视系统的执行，并自动的决定什么时候以及怎样对系统进行演化。它依赖于体系结构模型和知识库中包含的信息以及错误的软件模型和异常类型中的可重用信息。演化引擎根据系统设计者的要求来预测在系统操作中可能的异常，并找到合适的解决方法。整个过程包括：预测和检测异常、诊断异常、选择解决策略、演化系统。

（4）结构化抽象层次，体系结构抽象层次的目的主要是将演化引擎从动态重配置中所产生的低层次同步和一致性问题中分离出来，它允许演化引擎可以直接访问运行系统，以及在结构层次上表达动态系统变更，并确保这些变更被完全应用于运行系统。

8.2　网 格 计 算

随着超级计算机的不断发展，它已经成为复杂科学计算领域的主宰。但以超级计算机为中心的计算模式存在明显的不足，而且目前正在经受挑战。超级计算机虽然是一台处理能力强大的“巨无霸”，但它造价极高，通常只有一些国家级的部门，如航天、气象等部门才有能力配置这样的设备。随着人们日常工作遇到的商业计算越来越复杂，人们越来越需要数据处理能力更强大的计算机，而超级计算机的价格显然阻止了它进入普通人的工作领域。于是，人们开始寻找一种造价低廉而数据处理能力超强的计算模式，最终找到了一种既能满足大型计算的要求，同时也能进入普及应用的技术——Grid Computing（网格计算）。

网格的名字来源于电力供应网。电力供应网使得电力供应商能根据用户的需要供应电力，消费者只需支付自己使用的那部分电费。有鉴于此，在计算的性能方面，网格的目标是根据用户的需求通过网络提供必要的计算资源，而用户只需支付相对应的使用费用。

美国阿岗国家实验室的资深科学家、美国计算网格项目的领导人 Ian Foster 这样定义网格：“网格是构筑在因特网上的一组新兴技术，它将高速互联网、高性能计算机、大型数据库、传感器、远程设备等融为一体，为科技人员和普通老百姓提供更多的资源、功能和交互性。因特网主要为人们提供电子邮件、网页浏览等通信功能，而网格功能更多更强，能让人们透明地使用计算、存储等其他资源”。

网格计算的核心是计算机资源的共享。计算机资源包括三个方面：计算资源，存储资源和网络资源。要实现计算机资源的共享，问题的关键在于远程调用和控制某台计算机的计算资源与存储资源，以及对网络流量的调节。

如今，“网格”作为新出现的概念，它代表了一种先进的技术和基础设施，是继 Internet 之后又一次重大的科技进步。“网格”到底代表一种什么样的技术？网格技术追求的是一种什么样的前景和应用效果？什么是网格？如何实现网格服务集成？这些问题都是目前网格技术研究的热点。在此，我们将介绍一些有关网格计算的基础知识以及动态软件演化技术在网格计算中的应用。

8.2.1　网格计算的概念

在科学、工程和商业计算领域，还有很多问题难以使用现有的超级计算机解决，如需要处理的海量数据资源分布在不同的地理区域，所需的特殊计算设施和输入输出设备等不是本地的。基于这种情况，网格计算技术被提了出来，它将网络上的各种资源，包括超级计算机、大规模存储系统、个人计算机、各种设备等组织在一个统一的框架下，从而能够以非常方便的方法解决各种复杂的问题，这种方法又被称为元计算（Meta Computing），无缝、可扩展计算（SeamlessScalable Computing），全局计算（Global Computing）等。

网格计算技术的产生是应用对计算资源和计算能力不断增长的需求的结果。当单台计算机系统不能满足应用的需求时，就需要使用其他计算机系统的资源。但一方面，由于超级计算机系统现在还非常昂贵，不可能添置超级计算机作为解决该应用的专用系统；另一方面，即使可以使用其他超级计算机，由于不具备通用性，也不可能直接利用这些计算机上的资源。

网格计算系统的出现为解决上述问题提供了崭新的途径。从 20 世纪 80 年代末期 Larry Smarr 在 CASA 计划中首先提出这种方案以来，网格计算系统的研究就吸引了众多的注意力。在网格计算领域，已成立了 Global Grid Forum，eGrid: European Grid Computing Initiative 等论坛，以及多家面向网格计算的公司，而且一直不断有新的研究机构或工业组织加入到研究网格计算系统的活动中来。网格计算研究已成了当今软件技术的一个研究热点。

首先，网格是一个集成的计算与资源环境，或者说是一个计算资源池。网格能够充分吸纳各种计算资源，并将它们转化成一种随处可得的、可靠的、标准的同时还是经济的计算能力。除了各种类型的计算机，计算资源还可以包括网络通信能力、数据资料、仪器设备，甚至是人等各种相关的资源。那么，由此我们可以知道网格计算的广义定义就是基于网格的问题的求解。然而，狭义的网格资源主要是指分散的计算机资源，而网格计算就是指将分布的计算机组织起来协同解决复杂的科学与工程计算问题。

网格计算的特性主要有以下几个方面：

（1）异构性（heterogeneity）。网格可以包含多种异构资源，包括跨越地理分布的多个管理域。构成网格计算系统的超级计算机有多种类型，不同类型的超级计算机在体系结构、操作系统及应用软件等多个层次上可能具有不同的结构。

（2）可扩展性（scalability）。元计算系统初期的规模较小，随着超级计算机系统的不断加入，系统的规模随之扩大。网格可以从最初包含少数的资源发展到具有成千上万资源的大网格。由此可能带来的一个问题是随着网格资源的增加而引起的性能下降以及网格延迟，网格必须能适应规模的变化。

（3）可适应性（adaptability）。在网格中，具有很多资源，资源发生故障的概率很高。网格的资源管理或应用必须能动态适应这些情况，调用网格中可用的资源和服务来取得最大的性能。与一般的局域网系统和单机的结构不同，网格计算系统由于地域分布和系统的复杂使其整体结构经常发生变化；网格计算系统的应用必须能适应这种不可预测的结构。

（4）结构的不可预测性。动态和不可预测的系统行为。在传统的高性能计算系统中，计算资源是独占的，因此系统的行为是可以预测的。而在网格计算系统中，由于资源的共享造成系统行为和系统性能经常变化。

（5）多级管理域。由于构成网格计算系统的超级计算机资源通常属于不同的机构或组织并且使用不同的安全机制，因此需要各个机构或组织共同参与解决多级管理域的问题。

8.2.2 网格计算的体系结构

到目前为止，比较重要的网格体系结构有两个，一个就是 Foster 等在早期时候提出的五层沙漏结构。然后就是在以 IBM 为代表的工业界的影响下，在考虑 Web 技术的发展和影响后，Foster 等结合 Web Services 提出的开放网格服务结构 OGSA（Open Grid Services Architecture）。

1. 五层沙漏结构

五层沙漏结构（如图 8-3 所示）是一种影响十分广泛的结构，它的主要特点就是简单，主要侧重于定性的描述而不是具体的协议定义。

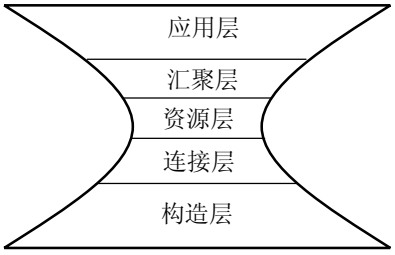


图 8-3　五层沙漏结构图

（1）构造层（Fabric）：控制局部的资源。由物理或逻辑实体组成，目的是为上层提供共享的资源。常用的物理资源包括计算资源、存储系统、目录、网络资源等；逻辑资源包括分布式文件系统、分布计算池、计算机群等。构造层组件的功能受高层需求影响，基本功能包括资源查询连接层和资源管理的 QoS 保证。

（2）连接层（Connectivity）：支持便利安全的通信。该层定义了网格中安全通信与认证授权控制的核心协议。资源间的数据交换和授权认证、安全控制都在这一层控制实现。该层组件提供单点登录、代理委托、同本地安全策略的整合和基于用户的信任策略等功能。

（3）资源层（Resource）：共享单一资源。该层建立在连接层的通信和认证协议之上，满足安全会话、资源初始化、资源运行状况监测、资源使用状况统计等需求，通过调用构造层函数来访问和控制局部资源。

（4）汇集层（Collective）：协调各种资源。该层将资源层提交的受控资源汇集在一起，供虚拟组织的应用程序共享和调用。该层组件可以实现各种共享行为，包括目录服务、资源协同、资源监测诊断、数据复制、负荷控制、账户管理等功能。

（5）应用层（Application）：为网格上用户的应用程序层。应用层是在虚拟组织环境中存在的。应用程序通过各层的应用程序编程接口（API）调用相应的服务，再通过服务调动网格上的资源来完成任务。为便于网格应用程序的开发，需要构建支持网格计算的大型函数库。

五层沙漏结构的一个重要特点就是沙漏的形状，其内在的含义就是因为各部分协议的数量是不同的，对于其最核心的部分，要能够实现上层各种协议向核心协议的映射，同时实现核心协议向下层其他各种协议的映射，核心协议在所有支持网格计算的地点都应该得到支持，因此核心协议的数量不应该太多，这样核心协议就形成了协议层次结构中的一个瓶颈，在五层结构中，资源层和连接层共同组成这一核心的瓶颈部分。

2. OGSA 体系结构

开放网格服务体系结构 OGSA （Open Grid Services Architecture）是由全球网格论坛提出，该结构以一种开放和标准的形式阐述了网格计算的需求，要求分布式系统的框架能够支持集成、虚拟组织和管理。因此，OGSA 框架需要一个核心集合，包括接口、预期的行为、资源模型以及绑定。

如果说五层沙漏结构是以协议为中心的“协议结构”，那么 OGSA 则是以服务为中心的“服务结构”，这里的服务通常是指具有特定功能的网络化实体。在五层沙漏结构中强调的是被共享的物力资源（或者是这些资源所支持的服务），而在 OGSA 中，这种服务所指的范围通常更广，它主要包括各种计算资源、存储资源、网络、数据库等。从资源到服务，这种抽象，将资源、信息、数据等统一起来，十分有利于灵活的、一致的、动态的共享机制的实现，使得分布式系统管理有了标准的接口和行为。

为了使服务的思想更加明确和具体，OGSA 定义了“网格服务”（Grid Service）的概念。网格服务是一种 Web Service，该服务提供了一组接口，这些接口的定义明确并且遵守了特

定的惯例，用来解决服务发现、服务创建、生命周期管理、通知等问题。在 OGSA 中，我们可以将网格看作所有网格服务的集合，即网格={网格服务}。

OGSA 定义了这些核心能力的需求，提供了一个网格计算环境的参考结构。它定义了在网格环境中非常有用的构件以及功能，尽管没有设计到实现的一些接口细节或者其他方面的详细说明，但是它被用在基于需求特定的环境中来识别功能。

8.2.3 网格软件构件

关于网格计算的很多方面是可以通过软件控制的，这些功能可以被复杂软件中的一系列手动的程序处理。软件完成这些功能主要表现在性能和实用性上。这些年来复杂软件比较普遍，但是早期的网格缺少资源支持，使得这些处理在软件中不能得到完全的实现。然而，在下面我们提到的这些软件在设计和配置网格环境时都必须要考虑到的。

1. 管理构件

任何的网格软件都有管理构件。

（1）有一个构件跟踪网格所使用到的那些资源的可用性。这个信息主要用来决定什么样的网格任务将被指派。

（2）还有一些构件用来测试网格节点的容量以及在任意时刻的利用率。这个信息主要用来在网格中安排任务。这个信息还可以用来测试网格的健康情况，发出错误信号，例如资源不足、拥塞或者过量使用。另外这个可以用来对整个的使用情况做出统计，类似于计算网格资源利用率。

（3）高级的网格管理构件可以自动地处理许多与网格有关的问题，就是所谓的自治计算。这样的软件可以自动地从各种网格错误和使用中恢复过来，并找到合适的途径回到工作过程中去。

2. 分布式网格管理

大型的网格通常有一个层次状或者是其他形状的拓扑结构，并且是一个连通的拓扑结构。这样，本地的机器可以连接成一个局域网，形成机器簇。网格可以以簇为单位组成层次型结构，用于管理的工作是分布的，可以增加网格的可测量性。收集、网格操作、资源数据以及任务安排都按照网格的拓扑结构分布其上。例如，不会将提交的工作安排给一个用于执行的机器。被派给较低层次机器的任务可以由多个机器（或簇）来完成。同样的，统计信息的收集也是分布的。较低层次的簇从单个的机器上得到信息，通过整合再发送给较高层次的管理节点。

3. Donor 软件

一般来说，每一个机器都需要在网格中先注册，并安装相应的软件来使用网格资源。通常，在一个机器登录网格之前需要有一些识别验证的过程，这些验证能够确保 Donor 软件、用户和网格本身的标识。

有些网格系统提供自己的注册功能，有些则依赖与本地的操作系统提供用户登录。后

者的情况下，用户 ID 匹配系统需要在不同的机器上对用户的权限进行匹配，这个功能通常由一个网格管理员完成。他决定哪些用户 ID 可以使用网格机器，并将这些 ID 输入到数据库中保存或者在数据库中注册。这样，对同一个用户来说，当任务被提交到不同的机器上时，合适的本地机器将会决定用户的权限。

网格系统记录向网格中新增加的资源信息。Donor 软件有一些监视器用来监测机器的忙闲和资源利用的数量和效率，这些信息被发送到管理软件，以合适地安排资源的使用。例如，在一个净化系统中，这样的信息告诉网格管理系统什么时间机器是闲置的，什么时间是正在工作的。

值得注意的是，安装在机器上的软件能够从管理软件那里接收任务并执行。在网格的任何地方，用户都可以提交需要执行的任务，而管理系统则需要同 donor 软件进行通信，才能使得任务发送到管理软件。Donor 网格软件要求可以接收可执行文件，进行处理并将结果返回给请求者。更高级的处理可以动态地调整正运行任务的优先权，将其挂起随后再恢复运行，还可以检查正在运行的任务在其他不同机器上恢复执行的可能性。这些操作对于反映装载平衡问题、优先权以及变更策略的制订都很有用。

4. 提交管理

通常，在网格中有许多节点机器用来提交任务和初始化网格查询。但是在有的网格系统中，这个功能由一个单独的构件来实现，被称作是提交节点或者是提交客户机。当网格使用独占资源构建时，提交软件需要安装在用户机或者工作站上。

5. 调度

大多数网格系统都有许多种的任务安排软件。这些任务定位到一个个机器上，由它运行由用户提交过来的任务。一般情况下，它只和满足需求的机器绑定，此外使用高级调度还有很多的优点。

许多调度可以用来实现任务优先权系统，通过使用一系列任务队列完成，每一个队列有不同的优先权。当网格单元可以执行任务时，则先从具有较高优先权的队列中取得任务。其中使用到的策略也可以使用调度实现，包括任务、用户和资源之间的各种约束。例如，可能有一个策略来限制在一天中的某个时候不能执行任务。

调度能够及时地反映网格负荷，他们利用当前机器的利用率信息来决定哪一个机器在提交任务之间是空闲的。调度也可以组织成层次结构，例如，一个元调度可以提交一个任务到簇调度上，或者是较的层次的调度上，而不是提交给一个单个的机器。

高级调度可以监视整个任务处理工作流的过程。一旦因为系统或者是网络资源不足造成任务丢失，调度则可以自动地在任何地方重新提交任务。但是，如果一个任务出现在一个死循环中并且已经超时，那么将不能够重新指派。任务有许多不同的实现代码，有的适合重新提交有的则不适合。

预留资源有预留系统管理，它不仅仅是一个调度。首先，它是一个基于时间建立的管理在特定时刻才能使用的资源的系统，防止在同一时刻有其他的任务请求同样的资源。它还可以在资源使用时间没有到达之前，移去或者是挂起使用该资源的任务。

6. 通信

网格系统中含有一些软件来协助任务与其他任务之间的交互通信。例如，一个应用可以分割成许多小的子任务，每一个子任务在网格中完成独立的工作。但是，应用执行的算法可能需要这些子任务进行一些信息的交互。因此，这些子任务应可以找到其他的子任务，并与他们建立通信连接，发送合适的数据。开放标准的信息传递接口（MPI）常常被作为是网格系统的一部分。

7. 观察和测量

我们知道调度可以及时反映当前网格资源的负荷，这些信息不仅仅对任务调度有用，还可以用来绘制整个网格的资源使用情况，统计出对额外硬件资源的使用趋势。另外通过对任务信息的分析可以预测以后的任务对资源的使用要求。

这些测量信息还可以用来作为公平的制订任务调度优先权的依据，以不同的形式展示网格资源的利用情况。

8.2.4 网格服务集成

1. 集成框架

自动地、按需集成 Web Services 要求解决高层次的任务分解、目标规划问题，要求集成是在概念层次上进行。OWL-S 从不同的方面描述 web services，把语义表示引进 web services，使得服务易于查找、匹配和合成。缺陷就是服务操作直接链接具体化参数，过程结构和特定的领域绑定，过程的抽象层次大低。与此相反，我们的过程本体论模型是面向问题解决方案的，过程继承蕴含了任务的分层递解关系。因此，我们提出了图 8-4 所示的网格服务集成框架。

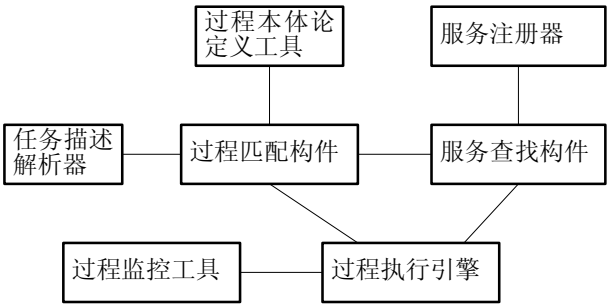


图 8-4 网格服务集成框架

任务描述表达网格用户需要解决的问题和任务，使用带约束的自然语言书写。任务描述解析器对任务描述进行解析，使之转换为规范的任务规格。

过程本体论定义工具定义业务过程，描述用户任务如何被分解为一系列活动，活动间如何通过迁移形成过程。它是面向用户和问题的，用于高层次的目标规划和任务分解。由于网格用户任务的解决可能是跨领域的，过程本体论应可表示抽象的、独立于具体应用的

问题解决过程；为降低任务分解的复杂性，使用分层构造机制；同时过程的表示应独立于 Web Services 的描述。过程匹配构件依据任务规格分类匹配过程本体论，得到可解决和完成任务的过程。

服务注册器使用 OWL-S 语言描述 Web Services 的语义并建立服务的分类体系。过程中的 Simple Process，作为执行任务的原子活动，应该映射到 Web Services 的端口操作上，被 Web Services 实现。服务查找构件按某种策略搜索执行 Simple Process 功能的 Web Services。

过程执行引擎依据过程控制流和数据流调用一系列 web services 共同完成用户任务。过程监控工具记录过程实例状态，如已运行活动轨迹、已运行活动时间、是否挂起等，也提供这些状态的查询操作。

2. 集成过程

在我们的集成框架下，Web 服务的自动化集成过程如图 8-5 所示。由用户使用带约束的自然语言书写的任务描述开始，经过解析，任务描述转换为规范的任务规格；依据任务规格分类匹配过程本体论，得到可解决和完成任务的过程；再按照服务本体论的服务语义描述，搜索筛选能够执行过程中的简单子过程活动的 Web Services；最后通过 SOAP 协议和 WSDL 调用执行 Web Services。

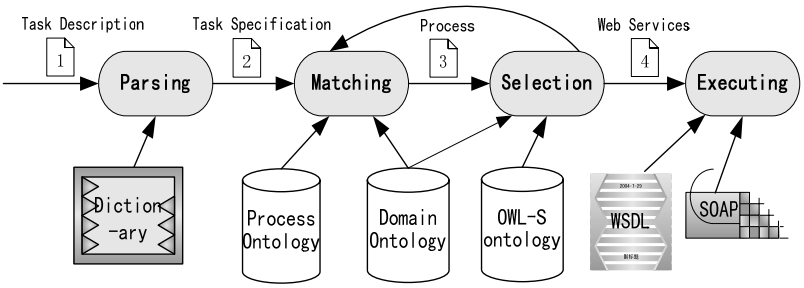


图 8-5　网格服务集成过程

在这里，任务描述使用带定语的动词描述，对任务描述的解析首先是自动切分词语。自动分词通常有机械匹配法、特征词库法、约束矩阵法、语法分析法和理解切分法等。其中，机械匹配法简单易行、速度快、不需涉及具体的汉语知识。由于任务描述使用规范的带定语的动词句法，因此，本文采用基于词典的改进的机械匹配法进行词语切分。然后把结果表示规整为元组（<领域>，<限定>，<基本任务>），得到任务的规格。例如，“银行财产抵押贷款”被解析为<银行，财产抵押，贷款>。

依据任务规格（<领域>，<限定>，<基本任务>）进行过程匹配，首先使用<基本任务>在过程本体论中进行高层匹配,然后在过程分类继承树中使用<领域>或<限定>进行由高到低的层次匹配。最终，有可能不能精准地定位到符合要求的过程，这时使用两种策略：紧缩匹配和宽松匹配。紧缩匹配意指对<领域>或<限定>进行领域本体论搜索，求得它们的下层<派生领域>或<派生限定>，再使用<派生领域>或<派生限定>进行过程的分类层次匹配。由本体论定义知，<派生领域>或<派生限定>是<领域>或<限定>的特殊化，遵从<领域>或<限定>的

所有约束，语义上更紧缩。宽松匹配意指对<领域>或<限定>进行领域本体论搜索，求得它们的上层<超领域>或<超限定>，再使用<超领域>或<超限定>进行过程的分类层次匹配。<超领域>或<超限定>是<领域>或<限定>的普适化，语义上更宽泛。一般说来，应优先使用紧缩匹配。例如在匹配不到“银行财产抵押贷款”过程时，可能匹配“银行房产抵押贷款”或“金融业财产抵押贷款”业务过程，前者为紧缩匹配，后者为宽松匹配。

鉴于虚类不能用来直接实例化出个体，虚过程类不能当作过程实例调度执行的完全依据，当匹配到的是虚过程类时，也使用紧缩匹配策略继续往下搜索，直到得到普通的过程类。一般说来，在过程本体论的分类体系中，增量式继承中的超类和派生类都是普通过程类，参数扩展式继承中的超类是虚过程类。

任务解决过程中的 Simple Process 应该映射到某个 Web Services 端口操作上，由 Web Services 具体执行。OWL-S 分三个层次对 WEB 服务作结构化语义描述：综合信息、输入和输出参数、前置条件（Precondition）和后果（Effect），为开发结构化服务相容查找机制建立了良好基础。所谓相容查找，是依赖于本体论的匹配检查技术，即在一个术语分类体系中，若相应于术语 A 的节点是相应于术语 B 节点的祖先，即 A 在 B 通往根节点的路径上，则称 B 语义相容 A，上面的紧缩匹配即是一种相容查找。相应于这三个层次，我们提供了三个服务相容筛选策略：分类筛选——依据寻求的服务在综合信息中提供的领域分类和 Simple Process 的属性匹配，从服务本体论筛选出相容类别的可提供的服务，称为候选服务；参数筛选——依据 Simple Process 对于输入和输出参数的描述，从候选服务中筛选出参数相容的服务，称为松弛选择的服务；约束筛选——依据 Simple Process 对于前提条件、后果和资源约束的描述，从松弛选择的服务中筛选出约束相容的服务，称为精确选择的服务。从分类筛选，到参数筛选的进一步过滤，到最后约束筛选的精确选择，粒度由粗而细，有良好的选择效率。

分类筛选和参数筛选直接参照领域本体论，比较容易理解。约束筛选涉及条件表达式，其相容性查找较为复杂，以下进一步说明。以前置条件为例，条件表达式旨在约束服务输入参数的取值，表达式中出现的变量只能是参数（以便对参数值作约束）。相容的约束筛选策略面向已通过参数筛选的服务，鉴于对应的参数相容，只要进行适当的参数变换，就可使作比较的 Simple Process 和松弛选择的服务的前置条件中，包含的参数变得完全相同，并仍然保持这些表达式的约束语义不变。此时，如果 Simple Process 的前置条件对于输入参数的约束比松弛选择的服务的前置条件更宽松，则服务的前置条件相容匹配。关于约束筛选更深一步的知识这里不细述。

如果没有查找到符合 Simple Process 规格要求的服务，返回到过程匹配阶段，在过程本体论的分类继承树中从这个 Simple Process 开始，定位其以子过程扩展式继承形式的某个派生过程类，从而转换扩展 Simple Process 为 Composite Process。最终 Composite Process 包含粒度更细的、层次更低的 Simple Process，以这些 Simple Process 为目标活动，再进入服务本体论中搜索能实现其的服务。

最后，通过 SOAP 协议和 WSDL 来调用 Web Services。OWL-S 的 grounding 把语义 Web Services 和服务的 WSDL 描述绑定起来。执行 Web Services 时使用 WSDL 中的消息描述来传输参数，使用 SOAP 进行 Web Services 的存取和通信。

8.3 普 适 计 算

随着用户需求的增长，计算的复杂性也越来强。传统的以计算机为中心的计算模式操作方式过于繁琐，而基于桌面的使用模式限制了用户的使用方式，因此急需寻找一种新的计算模式来满足人们的需要。本节首先介绍普适计算的基本概念，然后对其结构模型进行简单介绍，接下来讲解普适计算的一些关键技术以及动态演化在普适计算中的应用。

8.3.1 普适计算的概念

普适计算（Pervasive Computing）概念最早源自 Xerox PARC（Palo Alto Research Center）计算机科学实验室首席科学家 Mark Weiser 在 1988 年提出的“Ubiquitous Computing 缩写为 Ubicom 或 UC”思想，现在很多文献中又常以“Pervasive Computing”出现。其基本思想是为用户提供服务的普适计算技术将从用户意识中彻底消失，即用户和周围环境（无数大大小小的计算设备）在潜意识上进行交互。用户不会有意识地弄清楚服务来自周围何处的普适计算技术，就好比我们每天重复着开电灯、关电灯动作，却不会有意识地问自己电来自何方发电厂一样。

在普适计算之前的计算模式中，强调“以计算设备为中心”，人机交互的接口不够友好，操作起来繁琐复杂。另外，在整个计算的过程中，需要大量人为的有意识的主动参与，因此工作效率受到了极大的影响。进入 21 世纪后，我们更强调人性化和个性化计算的理念，因而以前传统的计算模式已不能适应当前我们的需求。由此，伴随着新技术的革命性发展，一种新颖的计算模式必然会呼之欲出。

在普适计算环境中，计算本身与周围环境已彻底地融为一体，人们能够在任何时间、任何地点、用任何方式进行信息的获取、处理和应用。计算设备将从我们的视野里消失，一方面是因为这些设备在体积上已经十分微小，我们很难察觉到;另一方面是由于这些计算设备已不是计算本身所关注的重点。因此，我们不用再去关心具体的计算设备和计算形式，而只需关注当前的所需要完成的工作任务本身。

国内清华大学的徐光佑教授和史元春教授对普适计算的研究较早开展，成果也非常显著。他们对普适计算的内涵进行了深入剖析，具体如下。

（1）普适计算是多学科交叉的研究，具有重要的科学意义。普适计算本身包括了计算机软件、通信、多媒体、生物认证、人机交互等技术，还涉及心理学、认知科学、行为学和社会学等多种人文学科；

（2））普适计算是计算模式的革新，代表了信息技术发展的必然趋势。计算模式发展的趋势是：桌面计算将会被普适计算所取代；

（3）普适计算在经济和国防信息化中具有重大的战略意义。我们可以利用普适计算中的具体实现技术，建立面向军事国防的以人为中心的和谐人机环境，为军事信息系统和危机处理系统提供广泛收集的军事信息。因此，普适计算的研究已经成为信息化社会发展的迫切需求，具有很高的学术价值和广泛的应用前景。

我们认为，普适计算必然是以“人”为中心，充分体现高质量、个性化的“为人服务”的思想。人、计算设备和智能空间是普适计算的三要素，人和设备在智能空间中能够自然、

自发以及透明地交互。计算设备在外形上已变得越来越微小，以致被人们很自然地忽略其存在，不再成为关注的重点。人和计算设备交互的接口将变得十分自然便捷和亲近友好：传统的键盘、鼠标式的交互方式将会被以“声纹”为代表的生物特征交互方式所取代。

纵观计算机进入人类社会的历史， 我们可以清楚地发现，计算模式已经经历了主机计算（Mainframe Computing）和桌面计算（Desktop Computing）两个时代。自从上世纪 90 年代初，美国 Weiser 博士提出“普适计算”这一新概念，就开始了人类向未来计算机时代探索的进程。

普适计算（Pervasive Computing），我们认为是指在普适环境下使人们能够使用任意设备、通过任意网络、在任意时间都可以获得一定质量的网络服务的技术。普适计算是在网络技术和移动计算的基础上发展起来的，其重点在于提供面向客户的、统一的、自适应的网络服务，具有间断连接与轻量计算这两个最重要的特征。普适计算所要研究的内容， 包括智能环境和不可见的计算、无缝的可移动性、普遍的信息访问、觉察上下文的计算和可穿戴的计算等。

普适计算的出现将大大改变人类与各种设备交互的方式。它预示着人类身边将出现一个充满各种嵌入设备、传感器、数字传输设备的真实的物理世界， 而这个物理世界能够映射成一个能“感知上下文”（Context-aware）和“携带多种信息”（Information-rich）的普适环境（Pervasive environment）。

8.3.2 普适计算层次化模型

普适计算对原有的计算机科学原理体系提出新的考验。为便于进行讨论和分析普适计算系统，我们有必要提出类似于 OSI 参考模型的普适计算层次化模型（LPC，Layered Pervasive Computing model）。NIST 的 ITL 和德国达姆斯塔特大学分别提出了类似的层次化模型。下面对三层模型作一介绍（如图 8-6 所示）。

L3: UC World	Pervasive computing Things that think Past-PC era Calm technology The Visible computer Embodied rituality Anything-anywhere/momadic computing
L2: Integrations	UC engineering (service discovery, awareness management, AAAS,...) Scalable computing (MOM,Web caching, disconnected-op, components, AOP,..) large-scale computing (organic computing cellular computing, active network, distributed AI, agents,...)
L1:Gadgets	Smarts devices (lables/badges/...) Smarts goods & spaces Smarts paper/cloths/dust/… Location/time/temp/…/awareness Ambient walls /offices/homes/… Augmented reality Wearable computers Pico/ad-hoc networks Internet appliances

图 8-6 普适计算三层模型列表

第一层主要涉及架构普适计算所需的硬件设备和网络设施。比如，智能身份证件（smart badge）、智能汽车、可佩戴计算机、Internet 信息工具、智能空间、自组织微微网（Ad-hocpico-networks）等；第二层描述了基于硬件的系统支撑软件的集成问题；第三层讨论 UC 时代的应用问题。

（1）后 PC 时代（post-PC era）。大型计算机时代，多人共享一台计算机（N:1）；PC 时代，人手一台计算机（1:1）；后 PC 时代（即普适计算），多人使用多台计算设备（N:M）。

（2）Things that think。数不清的专用计算设备和嵌入设备将提供给人们在日常生活中使用。

（3）智能设备（smart devices）。这些设备能够认知现实世界的部分环境内容（比如，时间、温度、人的位置等），这种认知能力使其具有自适应性。

（4）Augmented reality。Mark Weiser 针对 VR（Virtual Reality，以机器为中心）提出的 UC 思想（以人为中心）在实际中难以完全实现。这里所说 augmented reality 实际上是二者调和的产物。

（5）服务发现。混合网络要求节点间在没有设备知识的前提下相互熟悉、协作。为理解节点间的相互认识，术语“服务发现”方法被描述成像 Jini、uPnP、Salutation 和 Bluetooth-SDP 技术一样，致力于开发允许设备在混合网络中申明其存在和了解其他设备的能力。事实上，没有预备知识的相互认知是不可能的，即使人类也需要可供沟通的语言。

（6）认知管理。 智能设备通常是专用的，这些专用智能设备借助部分环境内容可以实现认知。但是，许多应用和设备可能同时访问同样环境（即要求显示同以认知内容），或者若干个智能设备产生更为复杂的认知（比如一组认知），或者存在多种环境内容，等等。这些都会使环境或认知管理变得复杂化，所以，认知管理在开放系统是一个软件工程问题。

8.3.3　普适计算的关键技术

1. 理论模型

普适计算的出现对一些原有的计算机理论体系带来了新的考验与挑战。为了能有效地对普适计算系统进行讨论、分析、设计、评估，非常有必要提出新的普适计算模型。当前在这方面的研究还处于非常初步的阶段。

（1）层次模型

美国国家标准与技术研究院 NIST 提出了一个层次化的普适计算概念模型。该模型参考计算机网络中的 OSI 模型，将普适计算由下而上分为五层，分别为环境层、物理层、资源层、抽象层及意图层。环境层和物理层表示了整个物理环境、用户生理系统及普适计算系统中的物理设备；资源层作为应用软件的基础，包括可获得的计算资源、系统可利用的用户技能等；抽象层则类似于 OSI 模型中的应用层，包括普适计算系统的应用软件和用户内在模型，它需要维护两者的一致性；意图层表示了应用及设备的目的或用户的目标。在该模型中，从设备角度而言，越上层抽象程度越高；从用户角度看，越上层时变特性越强。类似地，德国学者还提出了一个普适计算的三层模型，由下而上分为基件层（gadget）、集成层（Integration）和普适世界层（UC World）。澳大利亚分布系统技术中心的研究人员

则将普适计算归纳为四个元素：设备、用户、软件组件及用户界面，其中软件组件指那些能动态组成完整应用的编程部件，而用户界面则是建立在多个软件组件和设备上负责与用户交互的概念实体。

（2）智能影子模型

普适计算环境是一个开放的动态网络结构的环境，嵌入了计算、信息设备和多模态的传感器等多种异质实体，同时各种普适计算服务的形式各异、内容繁多。用户实体与计算实体之间、计算实体与计算实体之间的相互交互需随用户移动或环境变化而在不同时空动态演变。而层次概念模型仅对普适计算进行大尺度的简单建模，无法作为一个统一工具表达普适计算环境中的高度复杂性与动态性。因此，需要一种通用模型来描述与表达对象属性，刻画和捕获对象活动，为实体互操作和用户服务提供基础。

为此，浙江大学吴朝晖等人提出了基于物理场理论的智能影子模型，它从物理空间和信息空间融合这个背景出发，根据以人为中心的不同实体之间相互作用的基本原理，为普适计算建立了一个统一模型，较准确地表达了普适计算服务“无所不在”的时空特性、普适计算用户界面的“透明”交互模式及普适计算“以人为本”的基本理念。

2. 自然人机交互

随着普适计算中信息空间与物理空间的融合，人与计算环境的交互将从计算机面前扩展至人们生活的整个三维物理空间，人之活动场所，时时处处皆有交互。在普适计算环境中，人机交互的目标是利用人们的日常技能与习惯进行交互，同时尽可能不分散用户对工作本身的注意力。终极目标是使得人与计算环境的交互变得和人与人之间的交互一样自然、一样方便。

人类信息传递的主要渠道为文字、语言、图像。对应地，通过纸笔交互模式、语音以及视觉进行人与计算设备之间和谐交互正成为最有潜力的自然人机交互方式，这也是当前国际上的研究热点。笔式交互可帮助人们进行快速、自然的信息交流与沟通，而在日常生活中，更多的是听觉信息与视觉信息，它们同时可使人们获得更加强烈的存在感和真实感。

（1）笔式交互

笔式交互模拟了人们日常生活中的纸笔交互环境，可以帮助人们在保持自然工作方式的同时，轻松地进行各种信息的交流、记录与处理。笔式交互固有的非精确性和强大的信息表达能力，非常有利于快速、自然的思想表达和交流。

笔式交互较早见于 1992 年 Xerox PARC 研制的一个原型 Liveboard，而后国际上掀起了研究热潮，许多家院校、科研机构及企业纷纷对此展开深入而广泛的研究，国外如 MIT、CMU、Berkeley、日本企业 Sony、Wacom 等，国内如中科院软件所、汉王公司等。2004 年秋季著名人工智能组织 AAAI 举行了笔式交互的专题研讨会。笔式交互商业化的一个里程碑是微软牵头的 Tablet PC 产品的推出，微软为此专门研发了数字墨水、数字笔、笔控操作系统，从而实现了抛弃鼠标和键盘、完全用笔进行所有电脑操作，“用笔来思考和行动”，使人机交互回归到自然的纸笔时代。

（2）基于语音的交互

听觉通道是人与计算设备进行交互的最重要的信息通道之一。在日常生活中人类的相互沟通大约 75%是通过语音来完成的，利用语音进行交互的好处在于其认知负荷较低，不需要一直占据用户的注意力，可与其他通道同时进行工作而不冲突，同时人对声音信号非

常敏感、处理速度快。基于语音的交互试图通过语音识别、语音理解、语音合成等技术，实现人与计算实体在听觉通道的和谐交互。

人机语音交互的关键在于如何实现计算设备的听觉输入和听觉输出。大致步骤如下，人说话的语音数字化后经过计算设备进行语音识别与语言理解，从而让计算设备知道语音的含义，实现计算设备的听觉输入。然后可通过语音合成，用“说”的方式让人们可以听到计算实体的“意图”与“想法”，从而实现计算设备的听觉输出。

（3）基于视觉的交互

虽然人们可不利用视觉进行相互交流沟通，如仅通过谈话或仅用文字，但人们通常更乐于面对面地沟通，原因在于，通过相互所见，沟通可变得更加丰富、更加直接。人的意图和情感通常都是通过行为以及一些形体的微小变化而表现出来，肢体语言如表情、点头、个性化手势、身体移动方式、眼神等都是人与人沟通的重要信息，这些都是人与人交互的重要组成部分。

通过视觉通道感知人的行为与意图，是自然人机交互必然要追求的目标。基于视觉的交互，重点即在于让计算设备也能感知人们通过行为、肢体语言等视觉通道表达出来的意图、情感等高层个体化信息，其远期目标是能理解人们在日常生活中的任何活动与行为，从而使得人的任一动作与行为都成为交互的一部分，达到无时无刻、无处不在的实时感知。

由此可以知，基于视觉的交互其核心在于解决计算设备的视觉通道输入问题。基于视觉的交互可通过多种计算机视觉技术实现交互的视觉通道输入，当前的研究热点包括：

存在与位置判别：场景中是否有人？多少人？具体位置在哪里？

身份识别：用户到底是谁？

视线跟踪：用户正在看什么?

姿势识别：在语义上理解头/手/人运动的具体含义。

行为识别：用户正在做什么事情？

表情识别：用户当前的情感状态如何？高兴还是沮丧？愤怒还是哭泣？

3. 无缝的应用迁移

在普适计算环境中用户和计算设备都具有频繁移动的特性，如何为普适计算环境提供支持无缝的应用迁移的软件基础设施是一个亟待解决但又非常困难的问题。其目标是用户的任务可在任何时候任何地方暂停执行，无需用户进行其他主动式操作，而后即可随时在另一地方的不同设备上原样继续执行。例如，用户用家里电脑工作某一时刻因有事要去公司，他无需手动去执行保存数据、关闭各种程序等步骤，其在家里电脑上的整个桌面运行现场即可在公司的电脑上获得原样不变的重现，从而使用户察觉不到因工作地点、计算设备、计算环境的变化带来各种差异，而继续原来的工作。

无缝的应用迁移问题涉及软件基础设施的核心，需要深入研究普适软件的体系结构、服务构建模式、开发环境、软件编制模式等。从软件角度看，一个应用可以看作由一组服务与一组资源组成，从而必须有服务移动与资源动态绑定；另一方面，还需要移动环境下应用的运行现场保存与重构机制。

（1）服务自主发现

为了给移动的用户提供不间断、无缝的服务，需要提供服务漫游，支持在不同的物理

位置以及使用不同的访问终端来获取服务资源。其中，服务的自主发现是关键，需要一种服务发现的机制为用户动态、自主地检索、发现、绑定服务。

目前工业界已有多种适合不同软件平台的解决方案，如，蓝牙服务发现协议、通用即插即用简单服务发现协议（简单的底层抽象）、ETF 的服务位置协议、Salutation 的基于属性的服务检索、Jini 的服务发现协议等。特别地，Jini 提供了服务的广播、发现机制，以及动态代码移动，但 Jini 依赖于 Java 和 JDK。Surrogate 试图在 Jini 上改进，支持非 Java 的设备。

（2）资源动态绑定

在用户和设备的移动过程中，为实现无缝的应用迁移，需要对资源进行动态绑定。根据位置的不同可将获取资源信息的绑定方法分为四类：

① 资源直接移动：随着实体的移动，相关资源随之移动。一般而言，数据库资源不能移动；

② 资源复制移动：随着实体的移动，相关资源被复制。特别地，在多资源的拷贝时，须注意同步修改的冲突问题；

③ 资源远程引用：随着实体的移动，原来需要的资源被修改成远程引用。需要与拥有资源的远端机器进行网络通信；

④ 重新资源绑定：随着实体的移动，使用其他可用资源。

以上四种资源绑定策略的选择依赖运行时的状态、接入设备的属性。对于计算资源充足的移动设备，可以使用资源直接移动和资源复制规则；对于资源受限的移动设备可以使用资源远端引用和重新资源绑定规则。

（3）运行现场的重构

为了实现软件运行现场的保存与重构，目前通常的做法是通过某种机制将应用的执行状态与用户的数据相分离，并借助虚拟机。如“计算胶囊”与 Collective 系统。

“计算胶囊”（compute capsule）由 Stanford 大学的研究人员提出，它是一种新型的抽象机制，可用它重新构建操作系统。计算胶囊提供了一个私有的、虚拟化的、机器无关的系统资源接口，把一个用户任务运行期间的所有状态都封装起来，从而将应用状态与用户数据相分离，为应用迁移提供支持。

Chandra 等采用一种集中式管理的瘦客户端体系结构（称为 Collective）来支持应用迁移，其核心思想是采用基于缓存的系统管理机制，将应用的状态和用户数据分离，分别保存在两个不同的缓存容器中，便于应用移动到新的计算设备上。

4. 上下文感知

由于普适计算系统运行在极其动态和异构的计算环境中，智能实体需要能够感知环境变化，并根据动态场景自适应地调整服务状态。上下文感知计算（context-aware computing）即指能够根据上下文的变化自动地作相适应的改变和配置，为用户提供合适的服务。能够描述环境、用户和应用程序交互过程中相关实体的状态的信息，都可称为上下文（Context），它既可以是静态信息，也可以是动态信息。上下文可分为三类：计算上下文、环境上下文和用户上下文。计算上下文包括计算能力、存储能力、带宽、错误率、连接建立时间、成本、安全要求、竞争、延迟等，这些都会导致协议的变化、应用程序行为的改变、应用程

序功能的改变；环境上下文包括社会条件、物理条件等，例如温度、亮度、湿度等；用户上下文包括人的标志、位置、行为、偏好等。在上下文感知计算中，有四个主要问题：上下文信息的获取、上下文信息的建模、上下文信息的管理、上下文信息的推理。

（1）上下文获取。如何准确地获取各种不同类型的上下文信息是上下文感知中最核心问题之一。通常，具体的获取途径依赖于具体的上下文内容，底层的上下文一般通过传感系统直接获得，而高层上下文则根据底层上下文，并结合先验知识通过推理或融合得到。上下文获取通常是指底层上下文的获取。

传感器具有分布性、异构性、多态性等特性，这使得数据收集遇到了很大挑战，问题在于：如何用一种统一的方式去获取传感器数据？如何把传感器数据转换，并匹配为上下文数据？

当前上下文获取中研究最多的是如何获取设备和用户所处的位置信息。了解设备和用户的位置，非常有助于我们提供更有针对性的服务。这种对位置信息的获取被称为位置感知。位置感知目前已成为普适计算领域中一个非常活跃的研究方向，UbiComp’05 录用的 22 篇论文中有 8 篇是关于位置感知，超过三分之一，其热门程度可见一斑。当前已有多种技术与设备解决不同条件下的位置测定与位置信息的跟踪，如使用红外线、无线射频、超声波、计算机视觉技术等。

（2）上下文建模。上下文的多样性导致它们都有各自的表达方式与不同程度的内在联系，如何对这些繁复庞大异构的信息建立起统一的抽象逻辑模型，从而使得这些信息都能很容易被表达、推理和共享，这是上下文感知中另一个非常重要的问题。目前不同上下文感知系统所采用的建模技术各有不同，主要有表 8-1 所列几种：

表 8-1　上下文建模方法列表

模　　型	特　　点	例　　子
名值对模型	最简单的一种数据结构，易于管理，但对于复杂数据无能为力	Schilit、Capeus
标记模型	在属性和内容上加上标记标签的层次数据结构，经常用其他标签递归定义	CC/PP 扩展
图结构模型	非常易于求得 ER 模型，从而有利于在关系数据库中结构化	ORM 扩展
面向对象模型	利用面向对象技术的封装性和重用性，以解决普适计算中上下文的多变性，要访问上下文信息必须通过指定的接口	TEA 项目
基于逻辑的模型	上下文被定义为事实、表达式和规则，易于推理	Formalizing Context Multi-context
基于本体的模型	本体论是定义概念和相互关系的工具，优点在于容易达到语义上的共享	ASC 模型 Cobra 系统

（3）上下文存储和管理。鉴于上下文的广泛性，随着上下文应用的不断扩大，普适计算系统中的上下文数据必将变得很大，对这些数据需进行合理的存储，并加以一定方式的管理，从而能以最快、最佳的方式供应用程序与用户访问。

在这种移动、分布式的计算环境中，上下文数据存储的主要问题有：数据放置、数据缓存和缓存数据置换。

① 数据放置：数据存储一般放在相联的节点上，这些节点之间形成层次；

② 数据缓存：把数据缓存在移动客户端，当客户收到一个查询时首先从本地缓存中搜索。对于维护上下文感知系统下缓存的一致性，目前已经出现了位置相关的缓存有效性维护研究，通用的解决办法是在缓存数据时同时缓存其有效范围；

③ 缓存数据置换：由于移动设备的缓存空间有限，数据置换技术非常必要。传统置换技术一般置换访问概率最小的数据。

在上下文感知系统下还有另外两个因素：移动客户端当前位置和数据的有效范围。一般来说，距离越大或者有效范围越小，被置换概率越大。

（4）上下文推理。尽管每一时刻上下文感知系统都能从传感器网络中获得大量信息，但是这些信息只是原始未加工的信息，是对环境某一属性在一个时间点上的描述，同时某些传感数据也存在一定的不确定性与不可靠性，这使得我们难以直接获得需要的语义信息。为此，必须通过上下文推理，对这些原始信息进行信息抽取，获得语义知识，依照一定的规则进行推理，得到高层上下文。

当前上下文感知系统采用的推理技术，吸取了人工智能领域的技术，把上下文作为一种知识来推理。目前主要有以下两大类推理技术。

① 基于规则的逻辑推理：这是最传统也是最简单和最常用的一种推理，有很多系统采用了这种技术，主要代表为 GAIA。随着近年来本体理论的升温，语义 web 技术逐渐被应用到各个领域，包括上下文推理。基于本体的上下文推理一般都采用编写推理规则的方式。通常用 RDF 描述元数据的数据模型，用 XML、OWL 语句进行建模，以基于描述逻辑或者 FOL 编写的规则进行推理。

② 基于机器学习的推理：包括贝叶思网络、神经网络和基于马尔可夫模型的学习。贝叶思网络是近年来非常广泛使用的一种技术，尤其在 2000—2005 年期间被广泛用于上下文领域，非常适合处理信息的概率分布，而且已被证明在很多应用领域内有用。神经网络技术也是上下文感知系统中使用非常多的技术，包括上下文识别和预测。

5. 隐私保护

在普适计算环境下，个人信息和环境信息高度结合，一部分智能设备正从人们的周围环境转移到人的身上，人们将很难把自己从信息空间中分离出来。人们所说的、所做的、甚至个性化的感觉与情感都将会被数字化并存储起来。为了提供无干扰、智能化、透视的个性化服务，普适计算系统必须要收集大量与人们活动相关的上下文，如偏好、当前活动状态、未来计划安排等。这使得普适计算中的隐私和安全问题的研究显得极为突出。

如何在设计普适计算系统时为隐私保护提供系统框架级别上的支持，是一个非常重要的问题。瑞士联邦工学院（ETHZ）的研究人员在深入分析普适计算的特性之后，提出了普适计算应用设计的六条指导意见，并构建了一个隐私感知原型系统 PAWS。

（1）声明原则：关于数据收集行为的声明。不仅要声明数据的收集方式，如收集来的数据放在哪儿（即用户到什么地方去取）等等，而且要使用户能够方便地得到这些信息（如这些策略的声明等）。可借鉴 W3C 制定的 Web 隐私信息使用规范 P3P；

（2）可选择原则：要提供一种选择机制，让用户可以选择哪些是愿意提供给别的用户的，哪些数据他们同意被收集或被公开；

（3）匿名或假名机制：让用户不用担心因发布信息而被跟踪或者被识别。目前的一些通信方法很容易暴露用户的身份，需对通信协议进行一些调整，如不使用固定的硬件地址。该机制的缺点在于缺乏个性化信息；

（4）位置关系原则：用根据用户与感知设备或被感知对象的位置远近来决定对相应数据的访问权，从而加强访问约束。是当用声明原则解决问题遇到困难时的一种方法；

（5）增加安全性：由于引入了一些新的约束条件（如有限的能源、计算能力等），使现有的一些安全技术不能直接用于普适计算环境中，有必要对其进行重新审视与研究；

（6）追索机制：应具有追索用户所拥有的数据被使用的情况，相当于数据使用的日志，以满足法律的要求。

8.3.3　动态演化在普适计算中的应用

在普适计算中，用户通常处于移动状态，这导致在特定的空间内用户集合将不断变化；另一方面，移动设备也会动态地进入或退出一个计算环境，这导致计算系统的结构也在发生动态变化。而这种变化都是在系统处于工作期间，系统必须通过自身的调整来适应这种变化，从而保证计算行为的正确性、一致性和完整性。因此在普适计算概念提出之初，就已经包含了动态演化技术。

普适计算的基础设施通常包含了相应的硬件资源和系统软件，一个普适计算的系统软件与其他几类分布式软件不太一样，它主要有两种需求。

（1）物理的集成（physical integration）。在普适计算系统中涉及计算节点与物理世界的某种集成。例如，考虑一个智能会议室（smart meetingroom），其中的椅子嵌入了压力传感器、白板嵌入了可以自动数字化其上的笔迹的传感器、投影机嵌入了网络传输能力，这样房间可以感知会议中用户的存在和位置，当他们坐在桌子旁或在白板旁边发言时能主动提供合适的服务，房间中任一个 PDA 都可以把自己的显示重定向到投影机。

（2）自发的互操作（spontaneous interoperation）。一个普适计算系统可以划分为两部分，一部分是基础设施（infrastructure），一部分是移动设备。移动设备随着人的移动在不同的环境的基础设施中进出，这样就造成了系统环境的不断变化；同样基础设施部件的维护、升级也是造成系统环境变化的另一个原因。普适计算系统中的软件部件应能在这种变化的环境中进行自发的互操作，即不需人为地重新设置和添加软件模块，模块之间就可以自发地建立关联和进行功能上的协作，通过这种动态的自配置和自调整来应对环境的变化。

8.4　自适应中间件

近年来，自适应软件系统越来越成为学术界和工业界研究的热点，对于自适应中间件的研究已成为重中之重。如何让中间件具有正确可靠的自适应特性已成为当前研究的难点和重点。

8.4.1 自适应中间件的概念

下面，我们给出学术界对自适应中间件的相关概念，具体包括自适应的定义、自适应系统的定义以及自适应中间件的定义。

1. 自适应定义

学术界对“自适应”的研究已深入展开多年，并且对“自适应”从不同的角度进行了具体的定义。提出“自适应”的概念最早可以追溯到1971年，Ya.Z. Tsypkin在他的《Adaptation and Learning in Automatic Systems》一书中指出自适应是自动计算的一个方面。他还举了一个简单的自适应应用实例——双模式的手机，在某个物理位置上，该手机可以根据哪种制式信号的强弱程度来自动选择通信制式在GSM和CDMA上的自由切换。另外，随着环境的不断变化，系统需要自适应地调整，以便继续正常地工作。自适应是一种特性，可以使得软件根据操作环境的变化，改变自己的行为以应对变化。

从软件需求角度考虑，目前对于自适应的定义主要有以下几种：1）自适应是指系统整体或部分地适应需求的变化。2）若一个程序是自适应的，则该程序能很容易地改变，并且根据上下文的变化自动地改变行为。3）自适应是软件的一种属性，可以使得软件能够忍受不同的、特定的和变化的环境。4）自适应是软件的特性，软件系统通过这一特性可以进化来满足用户和商业的需求变化。5）自适应是指系统具有一种能够根据外部的刺激作出修改系统内部操作规则的能力。自适应通过修改软件本身，用执行后的结果来适应外部环境的变化。自适应是软件的进化，通过改变软件自身使之在新的环境下继续正常运行。

目前学术界对于自适应提出的多种定义，其共同点就是：自适应是软件的一种能力和特性，随着运行环境上下文的变化，通过自主自动地调整软件的组织结构或是调整软件的功能行为来满足变化的要求，到达“以变应变”的目标。

2. 自适应系统的定义

自适应系统的研究已成为软件工程的难点和热点。目前学术界对于自适应系统的理解也有多种。

（1）认为在复杂的软件系统中，重新获得灵活性和自适应的需求，已成为当前软件工程中基础的挑战；

（2）自适应系统已成为软件体系结构中非常重要的部分，包括在系统设计、协作、检测、评估和实现的各阶段实现无缝的自适应。

Narayana Subramanian在他的博士论文中指出：“自适应系统可以根据其运行场景上下文环境的变化，自动地调整系统内部的组成结构和逻辑算法来适应当前运行情况的变化；并且系统中的实体可以采用进化的方式来平衡环境的变化和系统运行中对功能性和非功能性需求的矛盾。”Narayana Subramanian对自适应的软件体系结构做了深入研究，并使用NFR的方法来生成自适应的软件体系结构，提出“自适应的软件体系结构框架（ASAF:Adaptable Software Architecture Framework）”。

自适应系统具有保护系统逻辑功能在不同环境条件下正常运行的自适应机制。在复杂多变的运行时上下文环境中，自适应系统仍然具有高效稳定的运行性能。和非自适应系统

相比，自适应系统具有高开放、强动态和自演化的软件体系结构。自适应系统的目标具有多重不确定性：自适应系统中的实体具有自主智能性；自适应系统的实体协同方式具有灵活多面性；自适应系统的实体演变具有高度动态性。

3. 自适应中间件定义

目前，在普适计算环境中，传统的中间件平台已不能胜任现有的需求，迫切需要一种新的中间件基础设施。因此，将现有的中间件平台自适应化已成为当前研究的热点和难点。

自适应中间件顾名思义即指中间件内在具有自适应的特性，可以随着外部环境的变化以及软件内部执行情况的变更来进行自适应的配置和重配置，包括组成结构上和行为功能的动态调整，以满足变化的需求，从而达到中间件的“以变应变”的特性。

自适应中间件具有与传统中间件一样的本质特性，都为分布式应用开发提供统一的编程模型，屏蔽底层开发应用平台的异构性等细节。不同之处在于自适应中间件具有开放性、动态性、可重配置性、智能性和可靠性等特征。

总体而言，自适应中间件是具有自适应系统特性的、能够自治地根据内外运行环境的变化做出相应正确可靠调整以适应变化的中间件。特别地，面向普适计算的自适应中间件必须具备以下四个特点。

（1）面向普适计算的自适应中间件（AM，Adaptive Middleware）必然是一个自适应系统 AS。即 AM 具有识别外部运行时上下文环境 EC 变化的能力（外省的能力）；AM 具有感知内部运行时环境信息 IIN 变化的能力（内省的能力）； AM 具有根据 EC 和 IIN 的变化来决策出系统 AM 要做出何种改变的能力；AM 具有执行自适应操作以做出相关改变的能力：AM 中存在一个自适应函数。

（2）面向普适计算的自适应中间件的基本实体组成必然是细粒度的。在普适计算环境中，“构件化”是这种细粒度的实体组成单元发展的必然趋势。以构件为基本实体的中间件的自适应必须体现在符合构件设计规范的全过程中，又集中体现在构件的装配部署（即构件分配）和运行时（即构件组合）两个阶段。

（3）面向普适计算的自适应中间件与上层应用的接口必然是智能化、语义集成的。自适应中间件中的基本单元的组合将对应为具有丰富语义内涵的智能体。这些智能体相互合作，遵循一定的规则，直接为上次应用提供高质量满意的人性化服务。

（4）面向普适计算的自适应中间件中的自适应操作必然是安全高效的。在执行过程中，自适应中间件为了应对感知到的内外情况的变化需做出相应的组成结构级的调整和行为功能级的变化，必然需要执行一组优化高效的以基本实体为单位的自适应操作，并且在此过程中应该确保执行的正确安全。因此，面向普适计算的自适应中间件需要包含自适应操作的约束规范集合。

8.4.2　自适应中间件的分类

根据中间件的体系结构中的不同层，我们将自适应中间件的研究项目按照面向中间件的不同层加以分类综述，主要针对中间件的基础设施层、分布式对象层、分布式构件层和通用服务层。

1. 面向基础设施的自适应中间件

具有典型代表意义的、面向中间件的基础设施层的自适应中间件项目主要包括：自适应通信环境、Adaptive Java 以及 Meta Socket。下面将分别加以介绍。

（1）自适应通信环境

美国华盛顿大学针对分布式软件开发中的危机，提出了面向对象的设计模式和框架，研制开发了自适应的通信环境（ACE-Adaptive Communication Environment ）。ACE 是一个面向对象的工具开发包，它实现了通信软件的基本设计模式。ACE 面向在 UNIX 和 Win32 平台上开发高性能通信服务的开发人员，简化了面向对象的网络应用程序和服务的开发。这些程序和服务用到了进程间通信、事件分离、直接动态链接和并发机制。

ACE 通过在运行时动态链接服务到应用程序和在一个或多个进程或线程中执行这些服务自动完成系统配置和重新配置。ACE 包含了一个丰富的集合，此集合中有可重用的 Wrappers 类以及能够在不同操作系统平台上交互的通用网络编程框架。

ACE 提供的功能具体包括以下六个方面。

① 事件分离和事件处理程序的调度；

② 连接的建立和服务的初始化；

③ 交互通信和共享内存管理；

④ 动态配置和分布式通信服务；

⑤ 并发/并行和同步；

⑥ 高级分布式服务构件（例如名字服务，事件服务，日志服务等）。

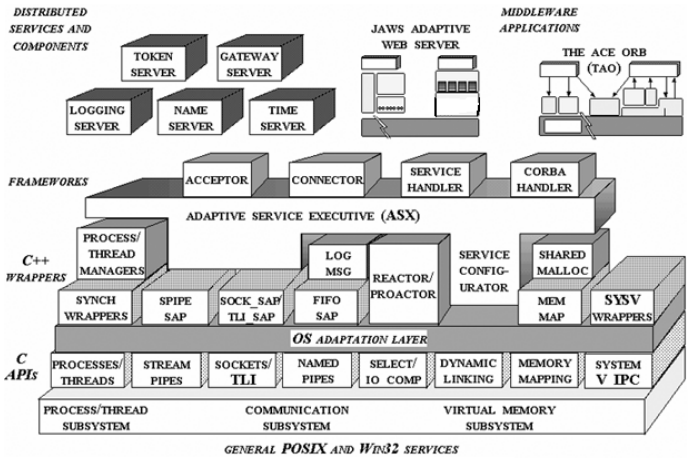


图 8-7 自适应通信环境 ACE 体系结构

（2）Adaptive Java / MetaSocket

针对于普适计算环境中的移动计算设备和无线网络技术的发展，美国密歇根州大学的软件工程与网络系统实验室致力于开发自适应的软件以适应动态、异构的计算环境。他们开发出了自适应 Java 语言（Adaptive Java ），对 SUN 微电子公司的 Java 进行了扩展，提供了自适应 Java 语言的构造器和编译器，用于设计开发自适应软件。在自适应 Java 语言的基础上，他们又开发了一个自适应通信构件——MetaSocket，该构件源于已存在的 Java

Socket 类，但它的结构和行为可以根据外部条件的变化做出自适应的调整。下面分别对自适应 Java 和 Meta Socket 进行简单介绍。

① Adaptive Java

自适应 Java 的核心思想是使用了“反射”的理念，其最基础的组成单元是构件，每个构件都具有三种不同类型的接口，具体如下：

a）常规调用接口（Invocations）：执行该构件中的正常功能行为的接口；

b）内部折射接口（Refractions）：用于观察构件内部行为的接口；

c）执行变化接口（Transmutations）：用于改变构件内部行为的接口。

在自适应 Java 中，如果要将已经存在的“普通的 Java 类”转变为自适应 Java 中的构件，那么要经过两步操作：

a）在基层（Base-level），使用“absorption”操作来构建从普通 Java 类到自适应 Java 构件的转变。“absorption”操作定义使用“absorbs”关键词；

b）在元层〔Meta-level），自适应 Java 构件使用“metafication”操作构建具有内部折射和执行变化接口的元构件。元构件的定义使用“metafy”关键词。

② MetaSocket

为了研究自适应 Java 在支持运行时自适应方面的能力，他们使用了该语言设计开发了称之为“MetaSocket”的构件。MetaSocket 构件中的“absorption”和“metafication”操作过程如图 8-8 所示。

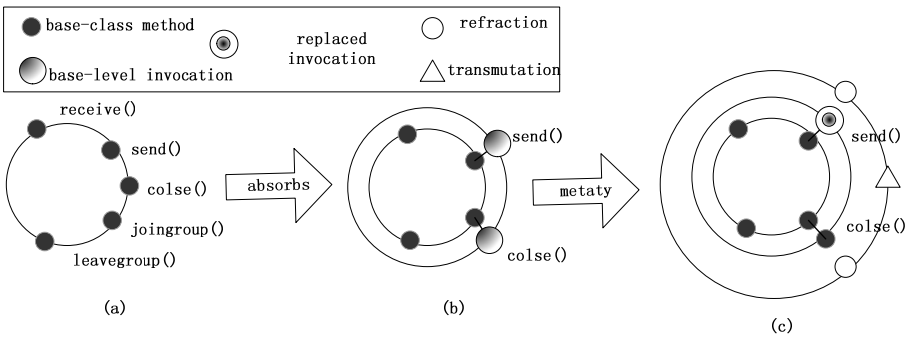


图 8-8　MetaSocket 构件的操作过程

在图 8-8 中，（a）为 Java MulticastSocket ；（b）为 SendMSocket 构件;（c）为 MetaSendMSocket 构件。描述了下面两个步骤。

- 对（a）中的 Java MulticastSocket 执行“absorption”操作，转变为基　层的（b）中的 SendMSocket 构件;
- 对（b）中的 SendMSocket 构件执行“metafication”操作，转变为元层的（c）中的 MetaSendMSocket 构件。

基于此，该实验室中的研究人员将 MetaSocket 应用到了一个“语音流交互”系统中，并通过使用了无线连接的 iPAQ 手持计算机进行运行测试。他们在系统中使用了过滤器来度量和报告无线通信中数据包的丢失情况。实验表明，使用 MetaSocket 的方法在动态变化的环境中具有较好的灵活性和自适应性。

2. 面向分布式对象的自适应中间件

具有典型代表意义的、面向中间件的分布式对象层的自适应中间件项目主要包括TAO，Dynamic TAO，2K，OpenCORBA 和 OpenORB。下面，我们将分别加以具体介绍。

（1）TAO / DynamicTAO / 2K

① TAO

在 ACE 的基础上，DARPA Quorum program 和 NSF 支持并领导开发了 TAO（ TheACE ORB ）。TAO 综合应用了各种软件设计模式和构件技术，是一个基于 CORBA 标准的中间件平台，可以实现远程对象调用，不用关心如何去实现对象定位;同时，可以实现 TAO 应用的跨平台、跨编程语言以及跨硬件平台和通讯协议的特性。一些在 ACE 开发过程中成功的经验和设计模式被应用在 TAO 的开发中，使得 TAO 成为一个高性能和实时的（QoS）的分布式应用平台。TAO 的主要特点包括：

- 建立一个实时 CORBA，可以进行硬（Hard）软（Soft）的 QoS 配置，满足具有关键业务的 DRE 系统需要；
- 将多种实时 I/O 子系统集合到 ORB 中，提供一种垂直方向的集成;
- 集成关键的设计模式和优化模式，用于开发平台兼容、可移植以及可配置的 ORB。

② DynamicTAO

DynamicTAO 是实时 CORBA 系统 TAO 的升级，主要解决了 TAO 无法在线配置的不足。如图 8-9 所示，DynamicTAO 通过配置器（configurator）来表示构件之间的依赖关系，其中，ServantConfigurator 表示 ORB 与应用构件（即 Servant）之间的依赖关系，而 TAOConfigurator 则表示 ORB 内部构件之间的依赖关系。用户可继承实现自己的配置器，以满足特殊的需求。如果需要替换某个构件，用户可以首先通过相应的配置器分析该构件与其他构件的依赖关系，在确保不会产生副作用的前提下替换构件。而 DynamicTAO 可自动分析内置的并发服务、调度服务、安全服务和监测服务之间的依赖关系，并保证相关配置的正确性。DynamicTAO 提供了一个元接口，允许在线装载和卸载构件以及查看或改变 ORB 的配置状态。此外，DynamicTAO 还能够支持通过截取器监测和调整远程调用。

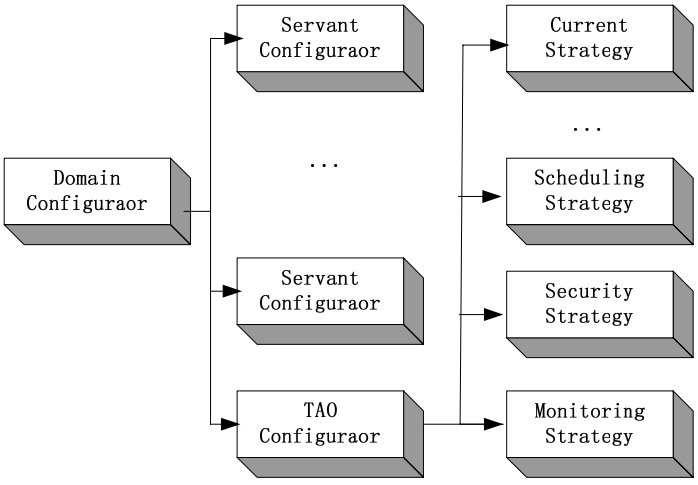


图 8-9　DnamicTAO 的对象配置器体系结构

③ 2K

Kon 等人认为未来的操作系统应该能够适应分布、异构和动态的网络环境。因此他们提出了一个基于中间件和构件的分布式操作系统——2K。之所以在操作系统中引入中间件，是因为现有分布式操作系统致力于分布式资源管理，忽视了主机节点的异构性;而中间件能够有效地解决异构问题，但是其资源地管理却涉及较少。因而，两者的结合可以达到相辅相成的效果。如图 8-10 所示，2K 分为三层：内核层就是目前主流的单机操作系统，如 Solaris，Windows 以及 Palm 操作系统；中间件层解决平台异构问题，强调中间件内部构件的动态配置能力（选择 DynamicTAO 是因其良好的配置能力；而选择 LegORB 是因其内存需求少，适于移动应用）；应用层又可细分为两层，底层为公共服务层，包括 CORBA 本身提供的公共对象服务，如交易服务、安全服务、事务服务等，也包括服务质量敏感的资源管理和自动配置等特色服务。应用层的高层则是实际应用，如视频会议、视频点播等。

因为 2K 的指导思想是“所需即所得”，所以 2K 项目重点考虑了系统构件动态配置的问题。基于系统构件之间的依赖关系，自动配置服务能够自动化地装载（不能自动卸载）需要的系统构件。构件之间的依赖包括构件装载的前提条件（装载该构件所需的硬件资源和软件服务），以及构件装载后与其他构件之间的依赖关系（当该构件发生诸如失效、替换、删除或是增加等变化时通知相关构件）。

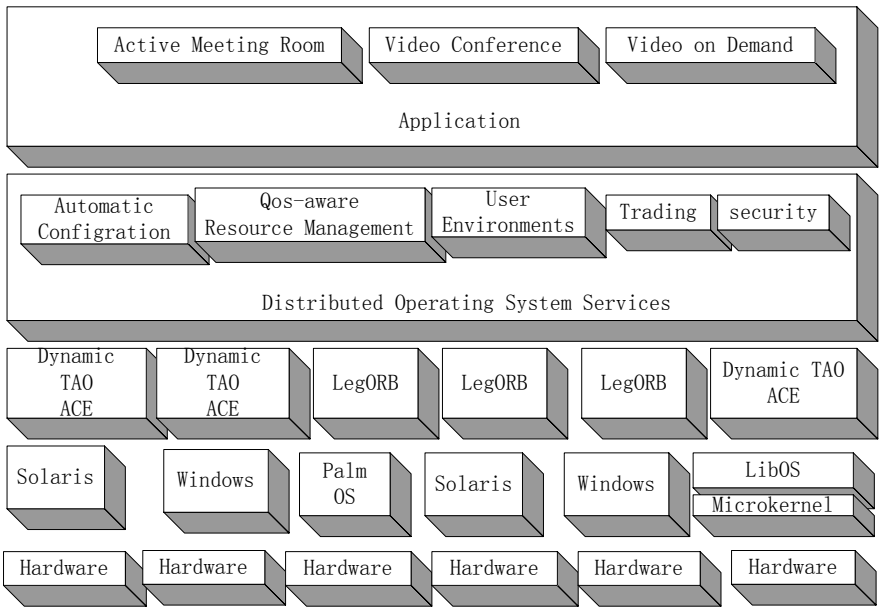


图 8-10　2K 体系结构图

（3）OpenCORBA

OpenCORBA 是采用自省式语言 NeoClasstalk 设计、开发并实现的自省式 CORBA，其中 NeoClasstalk 是 Smalltalk 语言的扩展，通过 MOP 实现了对象的类定义以及类的元类动态替换。基于该 MOP，OpenCORBA 实现了动态修改 IDL 接口行为的自省机制。

如图 8-11 所示，客户端代理（Stub）是远程服务器对象在本地的映像，负责创建并发

送调用请求、接收应答。在 OpenCORBA 中，IDL 编译器自动生成的客户端代理仅仅是 IDL 接口的 Smalltalk 描述，不包含任何实现代码。客户对代理的调用将通过 NeoClasstalk MOP 中的#execute:receiver:arguments:方法转发到与之关联的元类，该元类才真正实现 CORBA Stub 的功能。OpenCORBA 提供的 ProxyRemote 元类利用 DII 实现了 CORBA 标准语义的远程调用。而用户可提供其他的元类以实现不同的远程调用。在服务器端，OpenCORBA 将底层 ORB 对骨架的调用转发给与之关联的元类（如 TypeChecking 元类，负责在调用请求分发给对象实现之前以及调用应答返回给客户之前进行类型检查），然后由骨架将请求分发给目标对象实现。用户通过替换或增加元类来改变服务器端的行为。OpenCORBA 服务器端的元类与 CORBA 截取器类似，但是，前者可在运行时增加、删除或替换，后者在运行时无法定制。此外，OpenCORBA 的元类作用于单个 IDL 接口，而截取器作用于服务器端的所有接口。

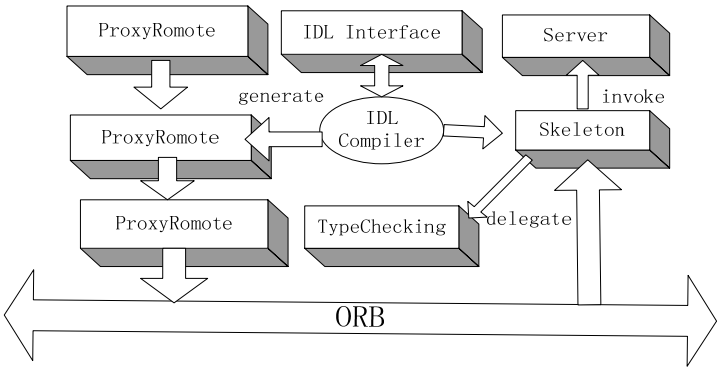


图 8-11　OpenCORBA 体系结构

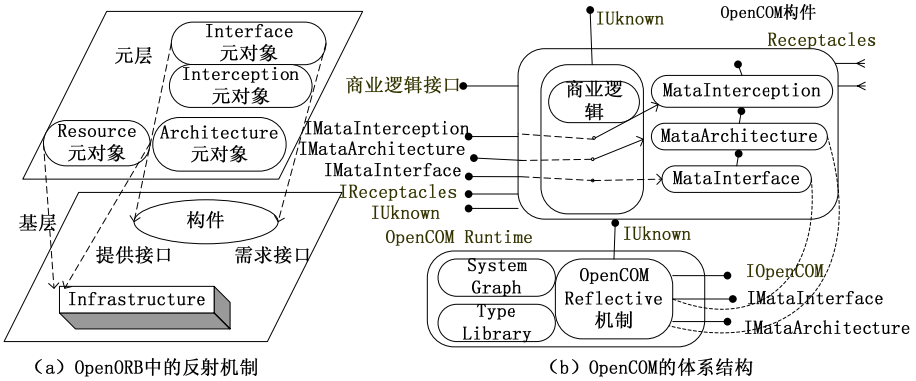
（4）OpenORB

CAS.Blair 与 G.Coulson 在自省式中间件的推广上做出了突出的贡献。他们提出了一个通用的中间件自省体系 OpenORB。该体系侧重于可观测的元信息，这些信息划分为与应用构件相关和与平台相关的两部分信息。

图 8-12（a）详细说明了 OpenORB 中的反射机制，分为：1）与应用构件相关的元信息包括接口、能够截获消息的截取器（Interception ）以及应用构件内部实现的结构信息（ Architecture ）；2）与平台相关的元信息，包括系统资源信息，例如缓冲区、线程和网络接入点等。OpenORB 强调上层应用的信息对基于中间件系统自省的重要性，却弱化了中间件自身的自省。这在其原型系统 OpenCOM 中体现得尤为明显。

图 8-12（b）详细说明了 OpenCOM 的反射体系结构。OpenCOM 是一个开放式构件模型，基于开放实现的思想，一个 OpenCOM 构件应该提供两种接口，一种用于对外提供构件完成的业务功能（称为业务接口）；另一种则对外提供与构件自身实现相关的信息访问与操纵功能（称为管理接口），例如业务接口的管理（IMetaInterface ）、截取器的管理（IMetaInterception）、对其他构件接口依赖的管理（IMetaReceptacles）、构件的管理（IMetaArchitecture ）以及构件生命周期的管理（ILifeCycle）。一个 OpenCOM 构件在实现业务功能的同时，必须实现 IMetaReceptacles 与 ILifeCycle 接口，其他管理接口由一个特殊的、地址空间唯一（Singleton ）的 OpenCOM 构件——OpenCOM Runtime 实现。所有

的 OpenCOM 应用构件必须首先将其实现体注册到 OpenCOM Runtime 中，然后由后者负责装载和实例化。基于应用构件通过 IMetaReceptacles 接口提供的接口依赖信息，OpenCOM Runtime 可以构造出地址空间内所有应用构件的依赖关系，并根据这种全局性的依赖关系考察应用构件的增加、删除和替换。这种开放式构件模型是实现自省系统的一种途径，与基于自省式中间件实现系统自省差异较大。主要体现在，前者需要应用的主动参与并显式提供一些相关信息和操作，而后者无需上层应用的显式支持；前者对中间件平台的自省支持不够，而后者对上层应用的支持相对较弱。OpenORB 的最新版本中逐步增强对中间件平台的自省，但对计算资源管理的自省力度不够。



（a）OpenORB中的反射机制　　（b）OpenCOM的体系结构

图 8-12　OpenORB 体系结构

3. 面向分布式构件的自适应中间件

具有典型代表意义的、面向中间件分布式构件层的自适应中间件项目包括两大类。

（1）基于 CCM 规范的自适应中间件 CIAO 和 Cadena；

（2）基于 EJB 规范的自适应中间件 PKUAS，Cogent 和 ARteMIS-M3C。下面，我们将对这两大类中的典型项目分别加以具体介绍。

① 基于 CCM 规范的自适应中间件

- 构件集成的自适应通信环境的对象请求代理 CIAO

构件集成的自适应通信环境的对象请求代理（CIAO-Component-Integrated ACE ORB ）是建立在 TAO 基础上的 CORBA 构件模型（CCM-CORBA Component Model）的具体实现。

CIAO 具有优良的“传统”：其根基是著名的 C++跨平台并发、网络通讯框架（Framework）“自适应通讯环境”（Adaptive Communication Environment，简称 ACE），其良好的架构设计，优秀的实现使其不但成为广大 C++网络、并发应用开发人员的必备工具，而且其中蕴含的框架、基于模式的软件架构（Pattern-Oriented Software Architecture）等软件设计思想也成为大家进行软件设计的范例和学习的榜样；基于 ACE 开发的“ACE 的对象请求代理”（The ACE ORB，简称 TAO）是利用 ACE 提供的框架和基本实现构件（Building Block）针对分布式、实时、嵌入式系统（Distributed-Real time-Embedded，DRE）开发的 CORBA 对象请求代理软件，从 TAO 的网站上我们可以看到它在学术界、工业界都得到了广泛的应用。

CIAO 已经成为由 COSMIC 支持的缺省的平台具体化模型。COSMIC 是支持一系列模型整合计算（MIC）和 OMG 的模型驱动架构（MDA）的一系列工具。因为很多的非功能性方面能够通过以构件元数据的方式来定义和实现，因此 COSMIC 可以很容易并且很正确地实现这些非功能性需求，而不必去过多担心分布式实时嵌入系统非功能性代码正确性的验证问题。COSMIC 和 CIAO 的结合给下一代软件开发提供了一个很好的范例。

- 基于 CCM 规范的强实时高安全集成开发环境 Cadena

Cadena 项目受美国 NASA 支持，由美国 Kansas 州立大学主要负责研发。美国军事研究办公室、DARPA、Intel 公司和 Honeywell 技术中心也参与该项目的部分研发。

Cadena 为基于构件的中间件系统提供一个集成的开发、分析和验证的环境，便于建模和构建符合 CCM 规范的中间件系统。Cadena 面向实时安全嵌入式应用领域，使用基于构件的开发方法来处理系统的非功能特性，主要包括：实时性、服务质量和安全性等。Cadena 主要应用在 Avionics 航空电子方面，具体使用在波音公司的 Bold Stroke 框架中。

Cadena 具有以下的特性。

a）扩展了 CORBA 的 IDL，加入了轻量级的规范来描述模型变量域、构件之间依赖关系、构件状态变迁语义。该规范允许开发者使用不同的方法来描述同一信息，以提高设计时的精确度；依赖分析能力允许跟踪构件之间的事件和数据依赖关系和基于依赖的构件集成算法的相关信息。

b）提供了一个新颖的模型检查基础设施，用于基于事件的构件间通信中的模型检查。通过 CCM 的 IDL、构件装配描述和注解，可以为中间件的全局属性进行模型检查。

c)提供了一个构件装配框架,用于支持可视化的构件设计开发和部署构件之间的连接。Cadena 使用实时通信通道来建立构件之间的连接,为波音 Bold Stoke 体系结构提供了自动构件部署框架。

② 基于 EJB 模型的中间件自适应项目

- 基于体系结构反射的中间件系统 PKUAS

北京大学计算机系的 PKUAS 系统是基于体系结构反射的中间件系统，其目的是解决现有中间件系统过于注重系统局部或单个实体的反射，而缺乏全局视图的问题，以及过于注重中间件平台内部功能的反射而对上层应用反射不够的问题。对此 PKUAS 系统采用了基于构件化的平台内部体系结构，并引入软件体系结构作为全局视图以实现反射体系对系统整体的表示和控制。

PKUAS 系统中的实体可分为容器系统、公共服务、工具和微内核四类。前三类为系统构件类；微内核则负责对系统构件的加载、配置、卸载以及启动、停止、挂起等状态管理。

PKUAS 系统中对反射的实现主要是通过元模型、元协议和元数据三者进行。PKUAS 的元模型为各种容器和各种服务 MBean，用于反射基层实体，管理部署信息和运行上下文导出元数据;元数据是由设计阶段的 SA、部署阶段的部署信息以及运行阶段的上下文导出并累积而成;元协议包括服务反射元协议和构件反射元协议，前者负责服务元数据的访问和修改，并执行装载、启动、停止、释放、替换和动态配置等动作。后者主要通过构件容器来负责对构件元数据的访问和修改，并维护元数据与构件状态和行为的因果关联。容器是构件运行所处的环境空间，它负责对构件生命周期的管理和构件运行所需上下文的管理。

- Cogent/ARteM1S-M3C

Cogent 系统是南京大学软件新技术国家重点实验室研制的基于移动 Agent 主动构件框架系统，以提高在开放网络环境下对构件组装绑定的灵活性、代码可重用性，解决异构构件之间的互操作等问题，并增强系统对动态变化网络环境的适应性。

在 Cogent 系统框架中，构件的组装机制由组调用表（GroupTable）和定位表（Location Table）两部分，系统底层的移动 Agent 将根据组调用表和定位表所描述的元数据信息完成对相关服务构件的组装调用。

组调用表给出了移动 Agent 对构件调用的执行逻辑信息，Agent 根据组调用表中的功能元数据迁移至提供服务构件的目标节点完成请求并返回结果，其结构为一七元组（调用表名、参数列表、变量声明、私有数据区、调用表体、类型结构、调用条件）；其中调用表体项的结构又为四元组（指引名、结构类型、接口方法调用、后处理定义）。

定位表用于描述服务构件所在的位置信息，Agent 将根据该表迁移至提供服务构件的目标节点完成请求并返回结果，其结构为（定位表名、调用表名、定位表体）。定位表体项的结构为四元组（指引名、节点地址、构件名、构件实例名）。

Cogent 系统对结构反射的支持主要通过对组调用表和定位表的联合解释来实现。组调用表中的类型结构可支持四种构件组装方式：SEQ（顺序）， PAR（并行）、SLOOP（循环）和并行循环（PLOOP），因此通过对结构类型元数据的修改调整，可以实现基层构件运行结构的重配置。通过对接口方法调用的改变可以实现基层构件功能方法的重配置。通过对定位表中构件名、构件实例名等项的修改，可以实现基层构件实例的增加、删除和替换。

ARTeMIS-M3C 则是南京大学计算机软件研究所在 Cogent 系统基础上的进一步拓展。其用于 Internet 环境下软件通信协同中间件系统，以解决开放环境中实体之间协同通信模式的单一性和固定性问题，实现软件服务实体协同模式的多样性、协同方式的灵活性和协同行为的定制性。

ARTeMIS-M3C 系统采用基于 Interceptor 的机制，系统中的 PCM （ Programmable Coordination Media）构成了元层，用于表述基层服务软件实体的协同行为，其核心组成部分包括 Interceptor Manager PCMClinentProxy 表，PCMServerProxy 表以及 SCA （Software Coordination Agent）表。通过对 PCM 的编程配置，可提高对开放网络环境的动态适应性能力。

4. 面向通用服务的自适应中间件

具有典型代表意义的、面向中间件的通用服务层的自适应中间件项目包括：RAPIDWare，ACT 以及 RCSM。下面，我们将对这些典型项目分别加以具体介绍。

（1）RAPIDWare / ACT

① RAPIDWare

RAPIDWare 项目受到美国海军部支持（Grant No.N00014-01-1-0744 ），由美国密歇根州大学的软件工程和网络系统实验室设计开发，起止时间为 2001 年到 2006 年。

RAPIDWare 是基于构件开发的自适应可信中间件，致力于自适应软件的设计和开发，强调高可靠和自适应软件的设计。RAPID Ware 面向动态异构的环境，支持强实时高安全的应用，例如金融网络应用、电力网络、交通系统和命令控制环境，防止外部环境部件的

严重失败和计算机网络攻击。在应用执行异常时，RAPIDWare 使得系统能够运行时自适应，包括动态修改和更新系统部件，从而使应用继续正确执行。RAPIDWare 支持对不可预测的变化的动态自适应，特别支持分布式多媒体应用，提供了高层的自适应和高健壮的多媒体通信服务。

RAPID Ware 项目的主要目标包括：

- 建立统一的方法来设计构件以支持多种非功能方面，包括：容错、安全、能力管理和服务质量。
- 在程序设计中封装了自适应中间件的功能特性，以方便服务质量需求的说明。提供新技术和服务的系统演化，自动检查系统的功能和非功能的属性。
- 提供一套工具来支持中间件的开发，包括新的编程范例和基于构件的软件重用等。

② ACT

RAPID Ware 的子项目 ACT （Adaptive CORBA Template ）是自适应的 CORBA 模板，提供了一个抽象的模型来构造和完善自适应的 CORBA 框架。ACT 框架可以使用不同的编程语言加以实现（例如：Java 和 C++等），而且 ACT 可以被用于扩展已经存在的自适应 CORBA 框架，包括 Qos 等。同时 ACT 可以用来实现异构的 CORBA 框架之间的互操作。一个基于 ACT 的框架可以和一个 CORBA 应用在运行时透明地集成起来，新的自适应特性可以在不需要重新编译的情况下动态地加入到原来的程序中。

ACT 使得 CORBA 应用程序在运行时的性能和灵活性大大加强，使得 CORBA 程序在功能需求和运行环境发生不能预知的变化时可以进行动态的反应。ACT 通过将自适应代码实时透明地“编织”进对象请求代理（ORB）中来加强 CORBA 应用。被编织进的自适应代码对总线中传输的各种请求（Requests）、回应（Replies）和异常进行拦截、调整和传递。ACT 自身是独立于语言和 ORB 的。特别的，ACT 可以被用于在任何语言和任何支持可移植拦截器的 CORBA ORB 条件下，开发面向对象的框架来支持代码的动态加载。而且，ACT 也可以被集成到其他自适应 CORBA 框架中，用来支持和兼容框架的互操作。

实现 ACT 的关键是通用拦截器（Generic Interceptor），是一种特殊的 CORBA 可移植拦截器。通用拦截器必须在启动时注册到 ORB 上，但是它的存在允许其他特定的请求拦截器的注册被推迟到运行时。因此，通用拦截器可以在应用运行时把自适应代码动态地编织进 ORB 中。这些代码拦截对象请求、回应和异常。除了通用拦截器，ACT 还定义了基于规则的拦截器（Rule-based Interceptor），在运行时，可以动态加载规则集合调整拦截得到的对象请求。

（2）RCSM

RCSM（Reconfigurable and Context-Sensitive Middleware）即可重配置的上下文感知中间件，该项目由亚利桑那州大学计算机科学工程系研究开发。

RCSM 致力于为普适计算环境提供位置感知通信的自适应服务。该项目支持普适计算的三个主要特性，具体如下：

① 上下文感知：具有获取上下文信息的能力；具有知晓上下文信息变化的能力；

② 位置感知：为应用提供一种能力，用来捕获和分析上下文信息和用户行为和设备之间的内在关系；

③ Ad hoc 短暂组通信：应用之间的组通信通道可以根据变化的上下文信息、节点的

移动情况和资源的可利用情况进行瞬间的建立和撤销。

另外，RCSM 还提供了以下三种服务。

① 位置感知的通信服务：RCSM 使用位置和相应设备的上下文信息来进行对象之间的通信，开发了位置感知的对象请求代理（Situation-aware Object Request Broker）和位置感知的接口定义语言（Situation-aware Interface Definition Language）以及相关的对象通信框架。

② 瞬间组通信服务：RCSM 使用位置作为协商的参数来建立设备的通信信道。位置触发的组通信服务为 Ad hoc 通信提供了便利。

③ 信息分发的自治协调服务：使用细胞自动计算模型来设计一个简单、节能和可扩展的信息分发服务，允许设备使用简单的协调规则来进行全局复杂的交互。

RCSM 主要使用了可重配置的 FPGA（Field Programmable Gate Arrays）进行中间件的自定制，为异构和嵌入式应用提高中间件的性能。RCSM 的测试应用场景为“智能教室”，为课堂上老师和同学的教学提供了方便，具体的应用包括上课的内容在 PC 和 PDA 上的同步和选择显示：指导学生合作完成某个任务等。

8.4.3　自适应中间件的支撑方法

本节将对实现自适应中间件的相关支撑方法进行综述，具体包括：基于 AOP 的关注分离方法、基于内省和调解的反射方法、基于构件的设计开发方法以及基于静态和动态的软件组合方法。

1. 基于 AOP 的关注分离方法

使用基于 AOP 的关注分离方法，可以使中间件的功能逻辑和非功能逻辑分离。由于 AOP 可以针对多个不同的关注点，例如：QoS、容错、安全等非功能特性，因此基于 AOP 的关注分离方法可以使中间件具有动态组合的能力，从而为中间件带来了可扩展性和自适应性。

前面已对 AOP 方法做了详细的介绍，因此对于该方法的机制和实现在这里不再作过多阐述。

2. 基于内省和调解的反射方法

将反射方法融入中间件的设计中可以使中间件由传统的“黑盒”特性转变为“灰盒”特性，从而使中间件具有部分有序的开放特性。中间件的内省能力可以使其洞悉自身，并且可以在运行上下文发生变化时采用调解的方法调整自身，从而适应新的运行状况。反射方法使中间件具有“内省”的能力。

（1）反射方法简介。Smith 于 1982 年提出了反射（Reflection）的概念，他在博士论文中指出：“既然一个计算过程能够被构造成使用解释器形式化地修改外部世界的表示，从而可以推理外部的世界，那么同样地，一个计算过程也可以设计成使用解释器去形式化地修改其自身的操作和结构，从而可以来推理其自身。”根据 Smith 的定义，我们可以获知：反射方法能够使得一个程序自身含有一个成分可以描述自身，包括自身的数据结构和行为操作，从而可以访问（Access）、推理（Reason About）和改变（Manipulate）其自身的解释。

反射的概念自 20 世纪 80 年代提出以来，陆续应用在程序语言设计和操作系统等领域。Smith 在 Lisp 语言中提供了元接口以感知适应底层虚拟机的信息，用于支持开放、扩展的语言编程。Lisp 语言中有多个解释器，分层排列构成了“反射塔”，塔底层的解释器执行用户的应用程序，上一层的解释器用于执行其下一层解释器提供的反射代码，从而形成了自下而上的层层反射的体系结构。

反射方法中自描述的过程被称之为“具体化（Reification）”，其含义为：假设程序 P 由编程语言 L 所实现，P 中的某个方面信息可以使用 L 加以特征化表示为 MD（称之为元数据），MD 可以被 P 在执行过程中所使用，并且 P 本身与 MD 是“因果相连（Causally Connected）”的，即 P 和 MD 中任何一方的变化将会引起对方的变化。因此，通过具体化过程，P 可以通过 MD 来观察 P 自身的状态，从而实现内省（Introspection）和调解（Intercession）。

下面，我们对内省和调解进行具体说明。

a）内省：即允许应用程序观察到程序内部自身的运行状态和行为信息。

b）调解：即允许程序担任观察者、决策者，并能够根据内省得到的信息修改自身的结构和行为。

综上所述，反射方法提供了一种能力，可以使一个系统对于上层应用暴露其内部的部分实现的结构细节（即元数据）。上层应用通过内省和调解手段，在运行时修改元数据达到动态改变系统行为的目的。由于因果相连的存在，系统的改变也会动态反应到元数据上来。

（2）反射系统简介。使用反射技术的系统称之为反射系统，是一个自表达、自观察和自修改的系统。反射系统采用了“面向 Aspect 的关注分离”的方法，将系统分为两个层次：基层和元层。反射系统中的基层和元层之间的逻辑结构关系如图 8-13 所示。

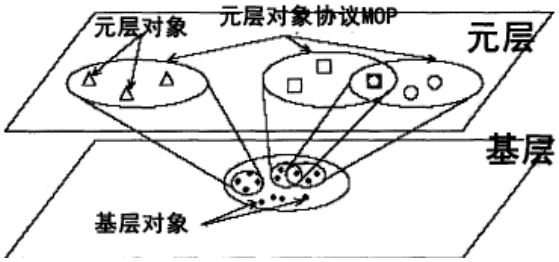


图 8-13 反射系统体系结构

a）基层：对和应用领域相关的具体问题进行抽象，直接反映用户的应用程序特征；

b）元层：对基层和系统内部的结构信息和行为信息加以具体化的表示。

基层和元层之间因果相连。反射系统具体化的过程作用在基层和元层之上，表示为：如何将基层中的组成结构、行为动作和系统内部状态在元层中表示为“元数据”。反射系统的内省和调解也作用在基层和元层之上，表示为：反射系统通过自省元层来观察系统内部状态行为，使用调解来调整修改基层中的相关信息。

反射系统使用了面向对象的方法，元层中的实体为“元对象”封装了基层对象中的元数据。在元对象中，提供了定义良好的接口来访问和操纵基层对象。元对象接口也称之为“元对象协议（MOP Meta Object Protocol）”。MOP 使得反射系统集成具体化、内省和调解

的功能。基于 MOP 的反射系统提供了以下两种反射类型：结构反射和行为反射。

a）结构反射：通过改变基层的组成结构来实现系统的反射。强调反射系统中类的集成、对象的互连、数据类型以及对象方法的检测等组成结构。例如，根据不同的网络安全情况，增加新的加密和解密对象连接，以实现不同的结构组成调整。

b）行为反射：通过改变基层的运行行为来实现系统的反射。强调了反射系统中的运行行为逻辑。例如根据不同的网络连接情况，选择和装载一个通信协议，以实现不同的运行行为。

（3）反射中间件简介。随着分布式计算、移动计算、嵌入式计算直至普适计算的出现，将反射方法集成到中间件系统中已是中间件发展的必然趋势。反射中间件具有传统中间件的本质特性，又提供了中间件内部的有序部分开放、自观察和自调整的特性。因此，反射中间件从某种程度上可以理解为：将中间件设计的“黑盒”原则（完全封闭）转变为“灰盒”原则（部分程序开放）。

Blair 将反射中间件定位在以构件为基本组成单元，他认为：反射中间件由一组可定制的构件集合组成，可以根据应用的需求进行重配置，从而使中间件可以在运行时根据环境的变化做出相应的变化。Coulson 认为：反射中间件具有因果相连的自表示能力，能够实现中间件的内省和自适应性。胡海洋等认为反射中间件是这样一类中间件系统——它将反射系统的反射特性融入中间件中，可以根据外部应用、系统环境及系统内部运行需求的变化，通过对系统自身特定成分的具体化及反射过程来实现系统特定部分的开放性、可配置性和重配置性。

然而，在反射中间件的设计实现中，虽然突出了“具体化”的工作，但是并没有突出自身智能推理的能力。反射中间件只是集中在如何在元层中描述基层的组成结构和动作行为，如何执行基层中相应的结构和行为的调整。

反射中间件和自适应中间件相比有一些不足之处，如，缺少对外部运行环境感知操作的能力；缺少根据元层信息进行推理决策的能力；缺少一个安全的语义约束规范用于指导执行基层中的结构和行为调整的动作。

3. 基于构件的设计开发方法

构件可以由第三方开发并自由部署。在构件技术中，Szyperski 特别强调了构件组合的重要性，他不赞成使用构件继承的方式达到构件的复用性。需要注意的是：在构件的组合过程中，构件之间的接口提供和需求的合约必须事先明确约定。由于构件生存在构件容器中，因此很容易做到“关注分离”，从而可以进行基于构件的面 Aspect 的设计。由此可知：我们可以使用构件技术进行重用；使用构件组合方法来实现中间件的自适应特性。

构件能够被第三方独立地装配部署。构件是自包含的，构件装配部署的配置信息中明确地规定了构件对内所需要和对外所提供的接口信息。基于构件的设计（CBD，Component-Based Design）通过组装不同厂家提供的 COTS（Commodity-Off-The-Shelf）构件以支持大规模的软件重用。独立部署的构件可以提供“迟组合”（或称“迟绑定”），这一机制对于自适应系统非常重要。“迟组合”支持在运行时构件之间通过良好定义的接口进行组合。

基于构件的设计方法可以用于系统的重新组合。对于构件组合，可以分为：静态和动

态构件组合两种方式，具体如图 8-14 所示。

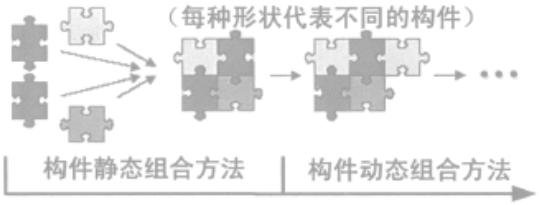


图 8-14 构件在编译和运行时的组合

（1）静态构件组合方法：构件只是在编译开发时、编译时和启动时（包括构件的装配和部署阶段）被静态地加以组合生成应用程序。然而，在运行时构件不能被重新组合；

（2）动态构件组合方法：构件可以在运行时动态的重新组合。

可以使用两种技术来实现动态构件组合，分别是“迟绑定方法”和“间接接口方法”。

（1）迟绑定方法：构件服务提供者和需求者之间通过已定义好的良好接口，使用迟绑定（或称动态绑定）的方法实现构件的动态组合；

（2）间接接口方法：在面向对象的程序设计语言中，可以使用间接接口作为主要手段来支持类的继承和多态，从而使得构件中的方法调用可以在运行时根据不同的情况被重定向到某个合适的方法实现中。

根据以上分析，我们可以综合使用“迟绑定”和“间接接口”结合的方法来实现构件的动态组合，以支持软件的动态自适应。

将基于构件的设计方法应用到中间件的设计开发中，已成为当前中间件发展的必然趋势。基于构件的中间件使用动态迟组合的方法，动态地适应上下文运行环境的变化。基于构件的设计方法为中间件系统提供了灵活、可扩展以及动态特性，中间件可以在部署装配、运行时重配置构件、升级构件和组合构件。通常而言，中间件面向某个应用领域，通过集成面向特定领域的构件，使用第三方的构件达到中间件自身演化和自适应的目的。

4. 基于静态和动态的软件组合方法

在使用软件组合方法之前用的最多的是“参数动态调整方法”。下面我们先简单介绍该参数动态调整方法，再具体介绍基于静态和动态的软件组合方法，最后加以比较和分析。

（1）参数动态调整方法。参数动态调整方法是指：“通过修改程序中的参数来改变程序的行为，以适应变化的发生”。Hiltunen 和 Schlichting 认为使用参数自适应的最好的例子是 Internet 的 TCP 协议，可以通过修改滑动窗口的值来调整其行为，从而适应当前网络的变化。当前，使用参数调整的方法已被广泛运用到上下文感知的系统中。软件的执行受到外部环境变化的直接影响，采用这种方法的自适应仅仅适用于有限的简单变化情况下的低层次调整，很难适用于普适计算环境中复杂变化的上下文环境。这种方法的缺点是明显的，具体包括以下两个方面。

① 参数调整与程序编写紧耦合，调整层次低。在设计开发完成后，不能加入新的控制算法和功能部件。

② 方法本身是封闭的，受限于已知的控制策略。必须明确知道在什么情况下，需要如

何调整有限的参数。

根据以上分析可知：参数动态调整方法不能动态地加入新的自适应控制策略以适应设计时未曾考虑到的方面。

（2）软件组合方法。使用软件组合方法与参数动态调整方法不同，前者综合使用了反射技术、AOP 方法和基于构件的设计方法，软件组合方法具体定义为：“通过互相交换算法级和系统结构级的构件，自主进行构件之间的组合，用来提高系统适应不断变化的环境上下文的能力”。

软件组合方法与参数调整方法相比，优点十分明显，集中体现在前者可以自主加入新的算法构件来处理新的关注点，在运行时可以应对在开发时未曾考虑到的情况。软件组合方法优于简单的参数调整和有限的控制策略选择，可以使软件在运行时动态重组合。例如：在一个内存受限的移动设备中通过控制构件的部署，可以使用系统内部构件的加入和移出，用来控制软件新的行为的变化，从而可以加入新的系统行为。和参数动态调整方法相比，软件组合方法虽然不支持运行时代码优化，但是具有更大的灵活性和自适应的特性，其扩展性非常好。

（3）软件组合自适应。软件组合带来的自适应（Compositional Adaptation）可以改变系统中的结构组成和功能算法，从而能够提高系统适应当前环境变化的能力。与参数自适应相比，采用组合自适应可以使应用在运行时采用新的算法来关注在设计实现时未曾预见的情况，因此具有非常大的灵活性，并且提供了超过参数自适应的简单参数调整和策略选择。

例如在资源受限（通常普适计算中的智能设备能量、计算和存储受限）的情况下，或在系统中增加新的功能行为来部署系统以适应未曾预期的情况和需求（侦测和响应一个新的安全攻击）情况下，相应能够部署的构件的数目受到了限制，因此采用动态组合的方式来部署构件是必须的。

实际上，软件组合自适应的历史可以追溯到 1946 年 ENIAC 的出现。在设计这个计算机时，所有的程序指令和数据都存储在同一个内存中，因此使得指令可同时作为数据或是指令。这个“自修改代码”被用于程序中的动态优化并提供了在有限内存下的解决方法。然而，该方法由于破坏了程序的一致性，因此很难加以调试，不久便受到了批评。然而，动态组合的理念却一直在不同的计算机领域中加以运用和实现。例如：高可靠软件的运行时的升级（电话系统和金融系统等）、Hot-swapbug 修复（On-the-fly 代码修复），软件隐私性保护，避免黑客的程序注入和操控。

最近几年，出现了相关的编程语言直接支持“动态重组”。1970 年出现的 Smalltalk 编程语言中提出了元类（meta class）作为 First-class 对象，将类的信息存储在元类中。1982 年 Smith 开发的 3-List 是第一个具有反射特性的程序设计语言。1984 年，Maes 开发的 3-KRS 是一个基于反射的面向对象语言。1991 年，Kiczales 开发了 CLOS 元对象协议。1995 年，SUN 微电子公司公布的 Java 语言中具有了结构反射的特性。另外，至今已有多个项目在 Java 只具有结构反射的基础上进行了扩展，使之具有行为反射的能力。

在 20 世纪 90 年代，组合自适应得到了相当大的发展，具体原因有二。原因之一，Mark Weiser 在 1992 年提出的普适计算的理念；原因之二，自动计算的兴起，对自管理系统提出了需求。至今，大量有关普适计算、自适应软件以及动态组合的国际会议和论坛纷纷召开，已成为研究的热点和难点；国外的一些研究项目也纷纷展开。

由于软件组合方法涉及的关键问题很多，我们需要从软件组合发生的方式、时间和地点三个方面对软件组合方法进行全面说明，具体分析如下。

① 如何组合

研究学者和开发人员已经提出了许多种方法来支持软件组合自适应，设计开发了相关的项目和商业软件包，分别以不同的形式来实现软件组合自适应。

Aksit 和 Choukair 对实现软件组合的方法进行了详尽的归纳说明。这些方法都采用了程序实体间的间接交互的方式，通过建立一个独立的层实现实体交互。一些项目使用了软件设计模式来实现这种间接交互，另一些项目使用了 AOP 和反射结合的方法，还有一些项目使用了中间件拦截的方法，并且此拦截对用户是不可见的。下面具体介绍这几种方法。

a）综合使用软件设计模式的方法

- 包装模式：包装器（Wapper）把所有的对象都封装起来，负责控制对象中方法的执行。使用该模式的典型项目有 ACE 和 MetaSocket 等。
- 代理模式：在对象中加入了代理，可以重定向到不同的对象实现中。使用该模式的典型项目有 ACE 和 ACT 等。
- 策略模式：每种策略均被封装起来，对于不同的应用透明地更换策略。使用该模式的典型项目有：ACE、TAO 和 CIAO 等。
- 虚拟构件模式：在程序执行过程中，根据需要加入不同的虚拟构件。使用该模式的典型项目有 TAO 和 CIAO 等。

b）AOP 与反射结合的方法

- 有不同关注的 Aspect 可以在运行过程中动态的被编织到程序中。使用该方法的典型项目有 AspectJ 和 TrapJ。
- 使用内省和调解方法，根据 MOP 协议实现结构级和功能级的反射。使用该方法的典型项目有 MetaSocket、DynamicTAO、OpenCOM、OpenORB 和 OpenCORBA 等。

c）在中间件中集成拦截器方法是在中间件内部加入拦截器，对方法的调用请求回复进行解释和重定向。使用该方法的典型项目有 TAO、CIAO、 MetaSocket、DynarnicTAO、OpenCOM、OpenORB 和 OpenCORBA 等。

② 何时组合

何时组合，即何时加入自适应的行为。何时组合取决于自适应中间件对于重配置的“敏感度”，即在某个时刻感知到是否需要做出改变的内在强烈程度。需要注意的是：中间件的自适应对于应用是透明的，而且应该在不改变原中间件代码的情况下获得自适应的特性。

从软件工程的经验上而言，如果软件组合的时间越迟，那么支持动态自适应的能力越强，但对系统的软件架构要求越高，需要非常高的软件一致性；需要确保自适应操作的正确安全，对软件的设计开发增加了复杂性。相反，如果软件组合发生在软件的设计、开发、编译和装载时间段里，那么所提供的动态适应的特性将受到限制，却相对容易实现。

a）静态组合

如果一个程序是在开发时完成软件组合，则该程序所具有的自适应是“硬连”的（Hardwired），自适应的行为不能再发生改变，除非重新编写和编译代码，并且再次执行。如果一个程序是在编译和链接阶段完成软件的组合，则该程序所具有的自适应是“中连”

的。不同的组合可以强调不同的环境需求，例如针对一个新的计算平台或是网络类型。AOP 的方法提供了编译链接阶段的软件组合，允许在编译链接时加载不同的 Aspect。不同的 Aspect 实现了不同策略下的安全和容错机制。这种可定制的应用仅需要重新编译或链接即可适应一个新的环境。

启动时（包括了软件的装配部署）软件组合提供了更大自适应特性，允许在程序执行的最后阶段决定采用哪个软件单元来执行以适应不同的环境。例如 Java 虚拟机对类的装入就采用了这一方法。虽然装入时组合也属于静态组合，但比以上两种静态组合具有更大的灵活性。当应用需要装入一个新的构件时，决策逻辑会从备选构件列表中找出最合适构件用于装入。因而，应用可以在现有的构件基础上灵活配置。例如：一个程序是在手持设备上运行，开始时此设备电能充足，那么一个彩色的显示构件（耗能大）将被加载；当过了一段时间电能将消耗大半，为了继续保持用户应用的较长时间运行，一个黑白的显示构件（耗能小）将替换掉原先的彩色显示构件。

b）动态组合

最为灵活的方法是在运行时实现软件组合。在执行期间，软件的功能逻辑和结构单元可以在不中断和不重启程序的情况下被动态地替换或扩展。

动态组合根据软件的功能（商业逻辑）是否可以修改被分为以下两种。

- 单向动态组合：软件禁止修改其功能，可以使软件根据环境的变化改变相应的关注点，使用于移动计算和普适计算。AspectIX 是具有单向动态组合能力的典型项目。
- 双向动态组合：软件允许其内在的功能代码发生修改，使得一个运行的程序通过动态组合具有新的功能。OpenORB 是使用动态双向组合的典型项目。

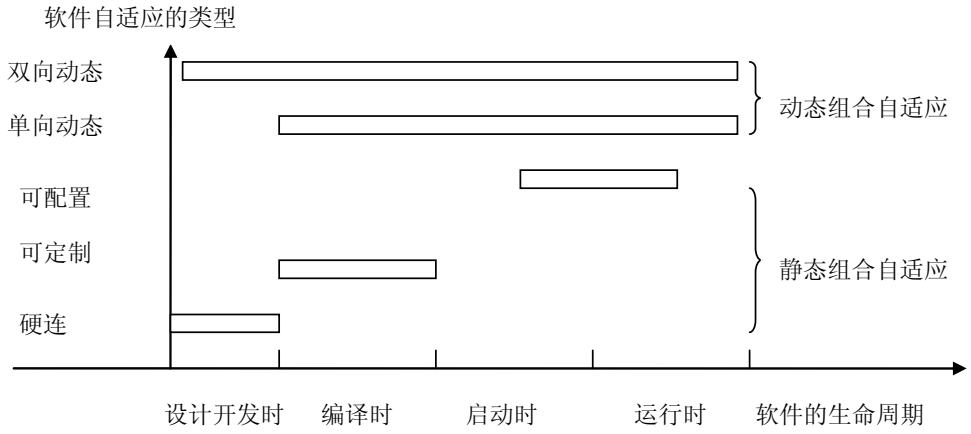


图 8-15　软件何时组合与获得自适应的关系

图 8-15 列出了以组合发生时间为标准来度量软件所获得自适应性的不同情况。纵坐标轴表示应用的类型，包括：静态组合自适应（硬连、可定制、可配置自适应）和动态组合自适应（单向动态和双向动态自适应）。静态组合自适应是指软件组合发生在设计开发、编译和装入（软件的装配和部署阶段）时；动态组合则是指软件组合发生在运行时。

③ 何地组合

根据中间件的分层模型，组合自适应可以发生在对应的不同层中，主要是中间件的主机基础层、分布式对象层（包括构件层）和分布式服务层。

a）中间件的主机基础层通过组合自适应在该层建立了一个自适应通信服务。ACE 是其中的一个典型项目，使用服务配置器支持应用的配置，实现应用服务的动态重配置；使用扩展虚拟机的方法，拦截和重定向功能代码中的交互，主要是在支持 Java 虚拟机的机器上进行。例如 R-Java 可以通过向 Java 解释器加入新的指令使对象的类在运行时发生改变。在该层进行组合自适应的缺点是方法的平台移植性不好。

b）中间件的分布对象和通用服务层

在分布式对象层(包括分布式构件层)和通用服务层进行组合的典型的项目包括:TAO、CIAO、OpenORB 和 DynamicTAO。在这两层进行组合的优点是组合方法的平台移植性较好，自适应的程度更大，可以更好地适用于动态、开放的普适计算环境。

第 9 章　支持动态演化的模型 SASM

9.1　引　　言

软件的可演化性和软件的构造性密切相关，演化性高的软件其设计具有良好的构造性。软件的构造性除了和软件的开发过程有关外，还取决于软件采取的应用程序模型。

例如，对象管理组织（OMG）提出的公共对象请求代理体系结构（CORBA），为分布式异构计算环境提供了一种灵活的通信机制和实用的软件集成技术框架。CORBA 的核心是 ORB，是分布式对象借以相互操作的中介通道。CORBA 还定义了最基本的对象服务构件和公共设施构件。所有这些有效减少了构件的构造复杂度，增强了软件系统间的互操作能力，使构造灵活的分布式应用系统成为可能。

Sun 公司推出的 J2EE 是用于开发基于构件软件的一种基础结构。以 J2EE 技术规范发展的应用服务器中间件为网络环境下的分布式应用提供了一种简洁、可重用的 Java 应用架构，向用户提供包括数据库连接、事务处理、资源管理、远程调用等服务功能，是标准构件（EJB、CORBA 构件等）的运行和集成管理环境，给软件设计带来很大灵活性的同时，降低了软件开发、管理、部署和维护的复杂度。

.NET 框架（.NET Framework）是在 Microsoft .NET 平台上进行开发的基础，为 Web 服务和其他应用程序提供了高效的开发环境和应用程序模型。.NET 框架包含了一个多语言执行环境，可以实现跨语言继承、纠错处理以及程序调试。.NET 框架也提供了一套统一的面向对象、异步、层次结构的可扩展类库，使开发过程变得非常简单，并缩小了将商业逻辑转化成一个可重用的构件而不得不编写的代码数量，极大地提高了软件的构造性。

随着软件技术的发展和软件系统规模的扩大，人们在开发软件系统中关注的重点已经从各个功能的实现逐渐转移到如何将实现具体功能的各个构件组装起来形成完整的系统。在构建分布式的应用时，其首要问题就是确定应用的体系结构。软件系统的体系结构原本是一个较为抽象的规约，最终分散而隐含地体现在系统各个构件及其间的交互行为之中。关键的问题是如何从总体上刻画并实现系统的结构，即所谓“体系结构”或“系统配置”的问题。除此之外，分布式应用系统常常表现出行为动态性和结构灵活性的特点。这些特点使得现有的框架形态和应用程序模型并不能完全适应分布式应用系统的需求，需解决的主要问题有：

（1）如何直观而严格地描述具有较强动态性的软件体系结构，如何描述体系结构在运行时刻的动态演化。

（2）采取什么样的应用程序模型来实现上面描述的体系结构。这样的应用程序模型中，定义怎样的操作才能实现系统体系结构的动态演化，又如何约束这些动态演化。

（3）作为一个在运行时态的软件体系结构（RSA），如何准确地描述目标系统的真实状

态与行为，RSA 的改变又如何直接触发、驱动系统自身的动态调整，以支持分布应用系统的行为与结构演化。

（4）良好的运行环境和平台支持。这样的平台提供控制变更过程的手段，动态演化的全过程都可以得到监视和控制，保障演化完整进行，提供对变更前后状态进行切换的机制，以维持运行期间的上下文一致性。

本章提出一个基于软件体系结构、面向动态演化的应用程序模型，这个模型以 D-ADL 语言为基础，旨在解决上述的前三个问题，关于第四个问题我们在下一章探讨。

9.2 D-ADL 语 言

近年来，Internet 已成为主流的软件运行环境，网络的开放性和动态性使得研究者日益关注具有自适应能力软件的开发。自适应软件的一个基本特征是能够在运行时进行演化以适应需求和环境的变化。软件体系结构（Software Architecture，SA）从全局的角度为系统提供结构、行为和属性等信息，已经成为软件开发过程中的核心制品.体系结构描述语言（Architecture Description Language，ADL）是 SA 研究的核心问题。如何在体系结构层次上刻画软件的运行时动态行为，如何描述动态的 SA，即动态 ADL 的研究已成为设计和实现自适应性软件的基础和关键。

F.Oquendo 等人认为，SA 至少应该从两个不同的视图描述：结构视图和行为视图。结构视图着眼于系统的组成元素及其互连拓扑结构，行为视图着眼于系统的功能和行为。有两种不同类型的行为应该被区分：计算行为和动态行为。计算行为面向系统的商业逻辑，处理的是业务功能中的数据信息；而动态行为面向系统的预定义演化逻辑，使系统能够自适应演化，以体系结构元素为处理对象，如增删构件、建立新的连接等。静态 ADL 如 Wright 和 Darwin 等关注系统的结构视图和计算行为视图，忽略系统的动态行为。而具备对动态行为的表示能力是动态 ADL 的显著标志。因此，设计动态 ADL 的关键之一在于采取什么样的方法表示动态行为。对于动态行为的表示机制，有两种设计思路：一是增强计算行为的表达能力，使之容纳演化逻辑的表示；一是将动态演化逻辑与计算行为相分离，显式地、集中地表达。后者比前者更优越，符合软件工程中关注分离的原则，有助于系统的动态调整。

另一方面，为便于 SA 模型的验证、求精和实现，SA 规约应该具有精确的语义。本书提出的 D-ADL 是一个形式化的基于高阶多型 π 演算的动态 ADL。D-ADL 显式、独立地表示动态行为，并将高阶多型 π 演算作为计算行为和动态行为的统一语义基础。借助高阶多型 π 演算理论，D-ADL 在设计阶段可以精确地刻画系统的交互行为，便于模型验证和求精；在系统投入运行时，有利于推导系统的行为，支持系统的在线演化。

9.2.1 D-ADL 设计原则

D-ADL 借鉴 π-ADL 语言的思想，遵循 ACME、Wright 等给出的已被广泛认同的体系结构描述框架，提供专门的标记符号，围绕体系结构实体如构件、连接件、系统配置、体系结构风格等进行体系结构建模。同时 D-ADL 从一个运行时角度支持软件体系结构的描述。

从运行时的观点看，软件体系结构应该从两个不同的视图描述：结构视图和行为视图。结构视图关注如下内容：（1）构件（系统计算或数据存储单元）；（2）连接件（规范构件间交互行为和约束关系）；（3）配置（构件和连接件构成的连通图结构）。因此，从结构视图来看，一个体系结构描述应该依据构件、连接件和配置提供一个形式化的规格。行为视图关注以下几点：（1）构件计算功能和连接件路由功能；（2）系统的配置活动；（3）系统的的演化行为。

良定义的 ADL 应该有能力描述软件系统的结构视图和行为视图。为支持对动态体系结构的描述，ADL 还应该有能力描述体系结构的结构变更和行为变更。D-ADL 就是为此而设计的一种动态描述语言，它是一种形式化的基于高阶多类型的 π 演算的语言，支持构件、连接件和配置产生变更。D-ADL 的设计遵循以下原则：

（1）形式化：以高阶多类型的 π 演算作为语义基础，使得其可以被机器自动化处理，便于模型检查、求精和演化。

（2）可执行：D-ADL 是一个可执行的语言，可被运行环境解释执行，成为整个系统调度的依据，进行可运行二进制物理构件的发现、执行和通信，从而驱动用户应用的运行。

（3）可演化：D-ADL 旨在提供体系结构的运行时的演化支持，包括结构、行为的运行时演化。

（4）抽象性原则：几乎所有的语法类畴都有着抽象定义，包括行为抽象和数据抽象。

（5）用户友好：除了文本语法外，还包括一个图形化可视文法。

D-ADL 中具有丰富的数据类型，总体上分为两种：常规类型（OrdinaryType）和体系类型（ArchitectureType）。常规类型主要容纳一般意义的数，如整数、自然数、字符串和布尔值等，可作为 π 演算中的名（name）。体系类型用于描述软件体系结构元素，如构件、连接件、端口和通道等。体系类型中最重要的是进程类型（ProcessType），描述体系结构中的各种行为和动作序列。进程类型的值就是进程（Process），对应 π 演算中的进程（process）概念。构件计算功能、连接件路由行为、体系结构配置活动以及系统预设的演化行为等都被模型化为 Process。构件实例和连接件实例也是 Process。高阶多型 π 演算中的 Abstraction 概念被 D-ADL 所直接使用。在 D-ADL 中，Abstraction 是一种特别的进程类型。Abstraction 可以是对 Process 的参数化抽象，这种情况下 Abstraction 用来形式化构件类型、连接件类型。通过对某个 Abstraction 类型的实例化（对其形式参数赋以具体的值），可能转化为 Process，例如构件实例、连接件实例。Abstraction 也可以是对另一个 Abstraction 的参数化抽象，因此 Abstraction 也用来形式化体系结构风格。这种情况下对某个 Abstraction 类型实例化，这个类型转化为抽象程度较低的另一个 Abstraction，例如对体系结构风格的具体化可能形成一个构件类型。

在高阶 π 演算中，不仅是一般的名，进程本身也可以作为参数被其他进程传递和计算。在 D-ADL 中，构件计算行为处理的是系统业务功能逻辑中的数据信息，而体系结构配置活动和预设的演化行为面向的主要就是体系结构本身，以体系结构元素为处理对象，负责体系结构的结构连接、行为变更，使系统能够动态演化。考虑这种情况我们作了不同的安排：在构件计算行为中，只有常规类型值才能够被传递、处理，因此被形式化为一阶进程。而体系结构配置和预设的演化行为却可以使用所有类型值作为参数传递和处理的对象，包括 Pocess，因此被形式化为高阶进程。这种区别设计虽然一定程度上增加了语言的复杂性，但有益于简化对体系结构的分析、求精和演化。

除此之外，体系结构描述语言还应该能够表达属性。属性可分为静态属性和动态属性两类。静态属性表征不随运行进度变化、稳定的属性，反映对体系结构在结构、行为和其他方面上的限制，即一般所说的约束概念。D-ADL 使用一阶谓词逻辑来表示静态属性（约束）。动态属性表征受系统运行过程影响的属性，如运行线程数、缓存大小、构件状态。D-ADL 使用<属性名，属性值>元组来表示动态属性。属性名是动态属性的名称；属性值是这个属性的取值。属性值在系统执行过程中被运行环境赋以值并不断更新，反映系统的当前运行状态。

9.2.2 高阶多型 π 演算简介

1. 高阶多型 π 演算定义

高阶多型 π 演算具有三个基本的实体：进程（Process）、名字（Name）和抽象（Abstraction）。在 π 演算中，进程是并发运行实体的单位，并以名字来统一定义通道（Channel）以及在通道中传送的对象（Object），每个进程都有若干与其他进程联系的通道，进程通过它们共享的通道进行交互。高阶多型 π 演算和一阶 π 演算不同的关键在于对象名字本身也可以是进程，这就使得交互中通信的数据可以是进程，进程可以被传递。通过接受对象名字，进程能够动态得到与其他进程进行交互的能力。因此，整个系统的结构和行为能够伴随时间演化。

抽象建立在进程的基础上，是带参数的进程。对抽象的参数进行具体化，得到一般的进程。在高阶多型 π 演算中，抽象的参数以及通道中传送的对象都具有类型，对抽象的参数进行具体化以及进程交互时都要注意类型兼容。

定义 1：设 N 表示名（name）的集合，名由小写字母表示，$\overline{N}\triangleq\{\overline{\alpha}\mid\alpha\in N\}$，$\alpha\in N\cup\overline{N}$，P，Q 表示高阶 π 演算进程，x 表示名，则高阶多型 π 演算进程可定义为：

$P::=0\mid\alpha(x).P\mid\overline{\alpha}y.P\mid P+Q\mid P\mid Q\mid\nu xP\mid[x=y]P\mid$D<$\tilde{K}$>，其中 D::=（$\tilde{x}$）$P$。

其意义的解释如下：

（1）0 表示非活动进程，它不能与任何进程交互。

（2）$\alpha(x).P$：这里 α 指某一通道的输入端，此进程的行为是先通过通道名 α 接收对象 y，再激活进程 P[y/x]，其中 x 为局部名（local name），[y/x]表示 a 换名操作。$\alpha(x).0$ 常简化为 $\alpha(x)$。

（3）$\overline{\alpha}y.P$：这里 $\overline{\alpha}$ 指某一通道的输出端，此进程的行为是先通过通道名 α 输出对象名 y，再激活进程 P，其中 y 为全域名（global name）。

（4）$P+Q$：这是进程的不确定计算形式，此进程将根据一定的诱因来激活进程 P 或 Q，它的行为仅为这个被激活的进程的行为。

（5）$P\mid Q$：这是一个并行操作形式的进程，此式中进程 P 和 Q 并行地存在，它们可以独立地同其他的进程进行交互操作，也可以彼此通过共享的通道进行通信。

（6）νxP：该进程中 x 是进程 P 的局域名，它使得进程 P 不能通过通道 x 与其他进程进行通信。

（7）[x=y]P：这是一个条件进程，如果 x = y，它将激活进程 P，否则将不进行任何操作，成为非活动进程 0。

（8）D 是抽象，定义为（$\tilde{x}$）P，表示以 $\tilde{x}$（$\tilde{x}$ 代表 $x_1,x_2,..,x_n$）为参数的进程。D<$\tilde{K}$>表示对 D 进行具体化，值 $\tilde{K}$（$\tilde{K}$ 代表 $K_1,K_2,..,K_n$）对参数 $\tilde{x}$ 赋值。

2. 归约和迁移

通过归约（Reduction）关系严格定义高阶多型 π 演算进程的内部演化。归约关系表达单个进程执行内部动作（intraaction）而进行的演化，该种演化独立于进程所在的外部环境，用于解释进程内部的活动。

定义 2（归约关系）：给定进程集合，它的归约关系→由表 9.1 中的规则定义，即能够从该规则推导出来的所有归约集合。

表 9-1　高阶 π 演算归约规则

R-COM	$\left(\overline{\tilde{x}}\tilde{y}P_1+M_1\right)\mid\left(x(\tilde{z}.P_2+M_2\right)\rightarrow P_1\mid P_2\{\tilde{y}/\tilde{z}\}$
R-TAU	$\tau.P+M\rightarrow P$
R-PAR	$\dfrac{P_1\rightarrow P_1'}{P_1\mid P_2\rightarrow P_1'\mid P2}$
R-RES	$\dfrac{P\rightarrow P'}{\nu\tilde{z}P\rightarrow\nu\tilde{z}P'}$
R-STRUCT	$\dfrac{P_1\equiv P_2\rightarrow P_2'\equiv P_1'}{P_1\rightarrow P_1'}$

上述归约规则定义中，$\dfrac{A}{B}$ 表示若 A 成立，则能得出 B。若断言 $P\rightarrow Q$ 能够通过归约规则推导而出，则 P、Q 之间存在归约关系，它反映了进程 P 能够通过单一、自治的步骤演化到 Q。“≡”是结构等同关系（Structural Congruence），两个进程结构等同是指它们只是书写方法不同，但意义完全等同。主要有如下结构等同关系：（1）如果 P 和 Q 具有换名关系，则 P≡Q；（2） $P_1\mid P_2\equiv P_2\mid P_1$；（3）[x=x]P≡P；（4）如果 D 定义为 $(\tilde{x})P$ 且 $\tilde{x}:\tilde{K}$，则 $D<\tilde{K}>\equiv P\{\tilde{K}/\tilde{x}\}$，$\tilde{x}:\tilde{K}$ 表示 $\tilde{x}$ 和 $\tilde{K}$ 类型兼容。

规则 R－COM 是基本的归约规则，它表明一个进程的两个并发成分通过对偶的名字进行交互，并在交互当中传递名字，交互的结果是整个进程执行了一个内部动作并进行演化。交互当中，并发成分 $\left(\overline{\tilde{x}}\tilde{y}.P_1+M_1\right)$ 选择执行它的第一个行为能力，通过名字 x 输出名字元组 $\tilde{y}$ 演化成为 P_1。并发成分 $\left(x(\tilde{z}).P_2+M_2\right)$ 选择执行它的第一个能力，通过名字 x 接受了名字元组 $\tilde{y}$，并用其取代 P_2 中出现在 $\tilde{z}$ 中的名字，得到 $P_2\{\tilde{y}/\tilde{z}\}$。于是整个进程演化成为 $P_1/P_2\{\tilde{y}/\tilde{z}\}$。

规则 R-PAR、R-RES 表明：如果 $P_1\rightarrow P_1'$，则能够得出 $P_1\mid P_2\rightarrow P_1'\mid P_2$ 或者 $\nu\tilde{z}P\rightarrow\nu\tilde{z}P'$，它们与规则 R-TAU，根据直观语义，都易于理解。

规则 R-STRUCT 是指，如果 $P_2\rightarrow P_2'$，并且 P_1 与 P_2 结构等同，P_1' 与 P_2' 结构等同，则能够得到 $P_1\rightarrow P_1'$。

归约关系虽然简单，但它只能表达进程内部演化，不能够表达进程与环境的交互作用。

而迁移关系通过定义进程能够执行的动作（Action），既能描述系统内部活动，也能描述系统与环境的交互活动，但它的代价是关系推导规则复杂。迁移关系运用系列推导规则定义，并由动作进行标记。迁移关系定义进程能够执行的动作，表明进程的行为能力。定义迁移关系时，它的标记动作分为四类，这四类标记动作与进程语法定义中进程的前缀相似，但并不完全相同，现定义如下。

定义 3（动作）：π 进程能够执行的基本动作运用如下的语法进行定义：$a ::= \bar{x}y \mid xy \mid \bar{x}(z) \mid \tau$。动作的集合定义为 Act。

第一类动作 $\bar{x}y$ 称为自由输出（Free Output），表示通过名字 x 发送名字 y。第二类动作称为输入（Input），表示通过名字 x 接受名字 y。第三类动作称为限定输出（Bound Output）表示通过名字 x 发送一个新名字（fresh name），新名字是进程交互中生成局部私有名字并发送该名字，z 是占位符，代表该新名字。第四个动作代表进程的内部动作。

定义迁移关系的时候，需要用到系列有关动作的术语和标记符号，定义如下。

定义 4（动作术语和标记符号）：动作的术语和标记符号定义见表 9-2。

其中，fn(a)定义标记动作 a 的自由名字，bn(a)定义标记动作 a 的限定名字，n(a)定义标记动作 a 中的名字。

表 9-2　高阶 π 演算动作术语和标记符号

a	Kind	fn(a)	bn(a)	n(a)
$\bar{x}y$	free output	{x,y}	Φ	{x,y}
xy	input	{x,y}	Φ	{x,y}
$\bar{x}(y)$	bound output	{x}	{y}	{x,y}
τ	internal	Φ	Φ	Φ

定义 5（迁移关系）：给定进程集合，它的迁移关系 $\{\xrightarrow{a} \mid a \in Act\}$ 由表 9-3 中的规则定义，即能够根据该规则推出的所有迁移的集合。

表 9-3　高阶 π 演算迁移规则

$$\mathrm{ALP}: \frac{P' \xrightarrow{\mu} Q \quad P \text{ and } \alpha-\text{convertible}}{P \xrightarrow{\mu} Q}$$

$$\mathrm{OUT}: \bar{x}\langle \tilde{K} \rangle.P \xrightarrow{\bar{x}(\langle \tilde{K} \rangle)} P \qquad \mathrm{INP}: x(\tilde{U}).P \xrightarrow{x((\tilde{K}))} P\{\tilde{K}/\tilde{U}\}, \text{if} \tilde{K}:\tilde{U}$$

$$\mathrm{SUM}: \frac{P \xrightarrow{\mu} P'}{P+Q \xrightarrow{\mu} P'} \quad \mathrm{PAR}: \frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q} bn(\mu) \cap fn(Q)\theta$$

$$\mathrm{COM}: \frac{P \xrightarrow{(\nu \tilde{y})\bar{x}\langle \tilde{K} \rangle} P' \quad Q \xrightarrow{x\langle \tilde{K} \rangle} Q'}{P \mid Q \xrightarrow{\tau} \nu \tilde{y}} \tilde{y} \cap fn(Q) = \theta$$

$$\mathrm{MATCH}: \frac{P \xrightarrow{\mu} P'}{[x=x]P \mid P \xrightarrow{\mu} P'} \qquad CONST: \frac{P\{\tilde{K}/\tilde{U}\} \xrightarrow{\mu} P'}{D\langle \tilde{K} \rangle \xrightarrow{\mu} P'}, \text{if} D \stackrel{def}{=} (\tilde{U})P$$

$$\mathrm{RES}: \frac{P \xrightarrow{\mu} P'}{\nu x P \xrightarrow{\mu} \nu x P'} x \notin n(\mu) \qquad OPEN: \frac{P \xrightarrow{(\nu \tilde{y})\bar{z}\langle \tilde{K} \rangle} P'}{\nu x P \xrightarrow{(\nu, x\tilde{y})\bar{z}\langle \tilde{K} \rangle} P'} x \neq z, x \in fn(\tilde{K}) - \tilde{y}$$

9.2.3　D-ADL 的语法规约和形式语义

本节讨论 D-ADL 对 SA 的描述方法，重点阐述 D-ADL 对行为的表示及其形式化语义解释。D-ADL 语法使用扩充的 BNF 范式表示。

1. D-ADL 的描述框架和类型系统

一方面，D-ADL 遵循 Wright 等给出的已被广泛认同的 SA 描述框架，围绕 SA 实体如构件、连接件和配置等进行 SA 建模，如图 9-1；另一方面，D-ADL 将高阶多型 π 演算作为行为语义基础。凭借高阶 π 演算描述动态系统的特征，D-ADL 允许构件、连接件和配置产生变更，并使得对 SA 的自动化分析成为可能。

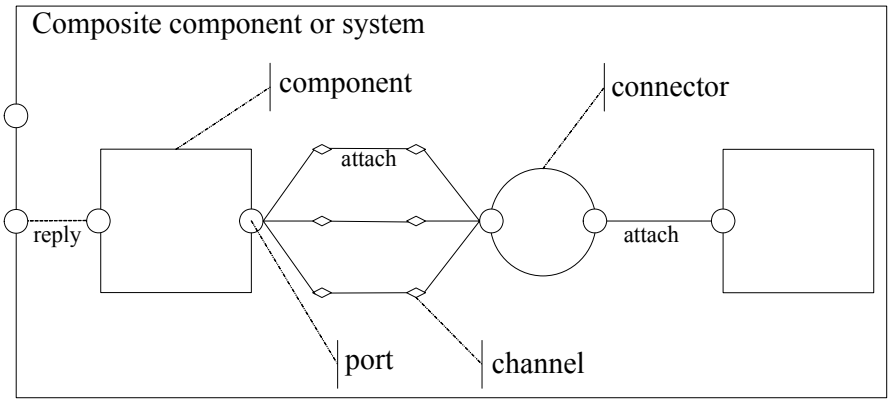


图 9-1　D-ADL 描述框架

构件（component）是 SA 的基本要素之一，依据接口和行为描述。在 SA 中，构件被视为用于实现业务逻辑的实体，执行系统的计算功能，此外构件还描述系统的动态行为。构件作为一个封装的实体，仅通过其接口与外部环境交互，而构件的接口由一组端口（port）组成。每个端口由一组通道（channel）组成，因此端口可看成通道集合。引入端口让构件的交互功能结构化、局部化，从而方便对其进行分析。通道是最基本的交互点，它的角色就是在 SA 元素间提供通信渠道，构件通过通道发送或接收值。

连接件（connector）是一种特殊的构件，旨在建立构件间的交互以及支配这些交互规则。连接件依据接口和路由行为（routing）描述。显然，连接件的接口也由一组端口构成，连接件的端口同样由一组通道组成。路由行为是计算行为的一个变种，指示构件间的通信轨迹和执行交互规则及约束。

复合构件（复合连接件）由外部端口和内部元素构成。应用系统本身可被视作一个特殊的复合构件。复合构件（复合连接件）的内部元素由一系列成员构件和成员连接件组成，它们通过配置链接在一起。配置要求构件端口（通道）与连接件端口（通道）显式连接。一个构件不能直接连接到另一个构件，构件之间通信和协作只能通过连接件进行。

D-ADL 将高阶多型 π 演算作为行为语义基础。D-ADL 拥有丰富的数据类型，总体上分为两种：常规类型（OrdinaryType）和体系类型（ArchitectureType）。常规类型主要容纳一般意义的数，如整数、字符串和布尔值等；体系类型用于描述体系结构元素，如构件、

连接件等。D-ADL 的基本类型和高阶多型 π 演算间的对应关系如表 9-4。

表 9-4 D-ADL 的基本类型和高阶多型 π 演算间的对应关系

<table>
<tr><th colspan="3">D-ADL 标记</th><th colspan="2">高阶多型 π 演算解释</th><th>备注</th></tr>
<tr><td colspan="3">Type</td><td colspan="2">Sort</td><td>类型</td></tr>
<tr><td colspan="3">OrdinaryType</td><td colspan="2">All name except channels in First-Order Process</td><td>通常意义的数值型</td></tr>
<tr><td rowspan="4">Architecture-Type</td><td colspan="2">Channel Type</td><td colspan="2">All channel</td><td>通道型</td></tr>
<tr><td rowspan="3">Process-Type</td><td>computation</td><td rowspan="3">Process Sort</td><td>First-Order Process</td><td>构件业务计算</td></tr>
<tr><td>routing</td><td>High-Order Process</td><td>连接件路由行为</td></tr>
<tr><td>choreographer</td><td>High-Order Process</td><td>体系结构动态行为</td></tr>
<tr><td colspan="2">component、connector、style</td><td colspan="2">Abstraction</td><td>体系结构风格、构件类型和连接件类型</td></tr>
</table>

体系类型中最重要的是进程类型（ProcessType），对应高阶 π 演算中的进程元素，表示体系结构中的各种行为和动作序列。构件的计算行为、连接件的路由行为以及体系结构的动态行为都属于进程类型，分别对应 computation、routing 和 choreographer 类型。计算行为处理的是系统业务功能逻辑中的数据信息，只有常规类型值才能够被传递、处理，因此被形式化为一阶进程；动态行为面向的主要就是体系结构本身，以体系结构元素为处理对象，使系统能够动态演化，而连接件的路由行为也可以使用所有类型值作为参数传递对象，因此都被形式化为高阶进程。高阶多型 π 演算中的 Abstraction 概念被 D-ADL 所直接使用。Abstraction 是对行为的抽象，用来表示构件类型、连接件类型和体系结构风格：当 Abstraction 是对 Process 的直接参数化抽象时，Abstraction 表示构件类型和连接件类型；当 Abstraction 是对另一个 Abstraction 的参数化抽象时，Abstraction 表示体系结构风格。

2. 构件规约及其计算行为语义

D-ADL 将类型与实例区分开来。构件类型（有时直接称之为构件）是实现构件重用的手段。构件具有三个基本组成部分：接口部分、行为部分和属性部分。鉴于本节重点在于讨论 D-ADL 的行为语义，关于构件接口和属性的 D-ADL 表示不再赘述。D-ADL 将构件行为分为两类：计算行为和动态行为。本节主要讨论计算行为规约（computation），动态行为规约（choreographer）见下节。

有两种构件：原子构件和复合构件。原子构件是指不具备内部结构的构件，语法如下：

Atomiccomponent ::=**atomiccomponent** *name*(*name*$_1$**:** Type$_1$,…,*name*$_n$**:** Type$_n$)
{ [0+ *Type definition***.**] [0+ *Variant definition***.**] [01 **port {**[0+ Port definition**.]}**]
computation {computation**} }**[01 **choreographer {** choreographer **} **]
[01 **property {** property **}**]**}**

构件可以参数化，进一步促进重用。computation 部分描述构件的业务计算逻辑，执行各个端口规约的交互行为和内部计算活动，是原子构件计算功能的完整规约。Computation 语法定义如下：

computation ::= prefix. computation | if boolean then { computation1 } [01 else { computation2 }] |
choose { computation1, … ,computationn-1 or computationn } | inaction | replicate |
unobservable. computation | OrdinaryType variant assignment. computation
prefix ::= via port^channel send OrdinaryTypeValue |via port^channel receive OrdinaryTypeVariable

prefix 是输入输出动作，via … receive…通过端口通道接收普通类型数据，via … send…通过端口通道发送普通类型数据；if… then …是条件选择语句；choose … or …是随机选择语句；inaction 是哑活动，可作为进程的结束符或 choose 子句的空选择；replicate 是计算的复制或重复；unobservable 是不可观察内部行为；assignment 是变量赋值语句，语法规格同一般程序语言相同。

为了支持系统的层次化分解，引入复合构件的概念。复合构件由多个构件实例和连接件实例组装而成，在规约层次上表达了成员之间的组合。下面是复合构件语法规格：

compositecomponent ::= **compositecomponent** *name*(*name*$_1$:Type$_1$, …,name$_n$:Type$_n$)
{[0+ type definition.][0+ variant definition.][01 **port {** [0+ port.]}] [0+ Variant assignment.]
instance {[0+ Instance.] **}**
configuration {[01 configuration]**}**[01 **choreographer** {choreographer}]
[01 **property {** property **}**]**}**

复合构件的参数声明、类型和变量定义都与原子构件相同。instance 段声明成员元素(含构件实例和连接件实例)，configuration 表示复合构件内的拓扑关系。Configuration 的语法规格如下：

configuration::=rely.configuration |attach.configuration |**inaction**
rely ::= *ComponenInstance^Port* **rely** *Port* | *ComponenInstance^Port^Channel* **rely** P*ort^Channel*
attach ::= **attach** *ComponenInstance^Port* **to** C*onnectorInstance^Eport*|
attach *ComponenInstance^Port^Channel* **to** C*onnectoInstance^Eport^Echannel*

rely 将复合构件外部端口（通道）映射到成员构件端口（通道）上，attach 将成员构件端口（通道）与成员连接件端口（通道）连接起来。

上述构件计算行为语法规约的 π 演算语义解释见表 9-5 所示。

表 9-5　构件计算行为规约的 π 演算语义解释

D-ADL 标记		π 演算解释		备　　注
computation	**if** ... **then** …	First-Order Process	*[x=y] P*	条件选择
	choose … **or** …		*P+Q*	不确定选择
	unobservable		τ	内部不可见行为
	inaction		*0*	哑活动
	replicate		*!*	计算的复制或重复
prefix	**via** …**receive** ...	Action	αx	通过端口通道接收普通类型数据
	via ...**send** ...		αy	通过端口通道发送普通类型数据
configuration	**…reply…**	α 换名操作		由 rely 或 attach 关联的通道被 α 换名为同一个名字
	attach..to…			

构件的计算行为（computation）和动态行为（Choreographer）的关系为选择组合。当将构件行为解释为 π 演算进程时，若构件行为用进程 P 表示，计算行为用进程 P_a 表示，动态行为用进程 P_b 表示，则 $P=P_a+P_b$。另外，复合构件处理业务逻辑的计算行为来自其成员行为的并发组合，其执行也被委托给成员构件和成员连接件。若复合构件计算行为用进程 P_a 表示，成员行为分别为进程 $P_{a1},P_{a2},\cdots\cdots P_{an}$,则 $P_a=P_{a1}|\ P_{a2}|\cdots\cdots|P_{an}$。

连接件是一种特殊的构件，同样地，连接件也分为原子连接件和复合连接件两类。原子连接件语法规约类似于原子构件，仅仅是计算行为（computation）描述换成了路由行为（routing，计算行为的一个变种）。通过结构化组合，多个连接件和构件也能形成复合连接件。

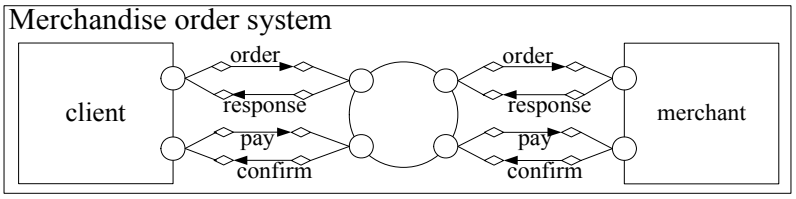


图 9-2 商品订购系统体系结构

我们以一个商品订购系统为例说明 D-ADL 的使用，因为重点是讨论如何使用 D-ADL 进行 SA 建模，所以对商品订购系统适当作了一些简化，省略了细节问题。其体系结构如图 9-2 所示。

工作流程如下：客户首先向商家发出货物订购（order）请求，商家根据情况确定是否能够满足客户的订购请求，向客户发出响应（response）。响应信息包含是否同意请求，可能还包含货款信息；若商家同意订购请求，客户支付货款并向商家发出款已付信息（pay），商家确认已付帐（confirm）并发货，订购完成。D-ADL 描述如下：

```
atomiccomponent Tclient () {
port { portc1: Tcaccess. portc2: Tmaccess.}
computation  {
via portc1^order send orderdata.via portc1^response receive replydate.
if replydata^accept then
{ via portc2^pay send replydata^payment. via portc2^confirm receive confirmation }
replicate } }
atomiccomponent Tmerchant () {
port { portm1:Tcaccess. portm2: Tmaccess. }
computation  {
choose {
{ via portm1^order receive orderdata.unobservable.via port1m^response send replydate },
{ via portm2^pay receive payment.unobservable.via portm2^confirm send
confirmation }}
replicate  }  }
atomicconnector TOrderLink() {
port { portc-l1,portl-m1:Tcaccess. portc-l2,portl-m2: Tmaccess.}
```

```
        routing  {
choose  {
{ via portc-l1^order receive orderdata.via portl-m1^order send orderdata.
via portl-m1^response receive replydata.via portc-l1^response send replydate },
{ via portc-l2^pay receive payment.via portl-m1^pay send payment.
via portl-m2^confirm receive confrimation.via portc-l2^confirm send confirmation }}
replicate  }  }
compositecomponent  TOrderSystem() {
instance  {client:Tclient(). merchant:Tmerchant().cmlink:TOrderLink().}
configuration {
attach client^port1 to cmlink^portc-l1.attach cmlink^portl-m1 to merchant^port1.
attach client^port2 to cmlink^portc-l2.attach cmlink^portl-m2 to merchant^port2.
inaction } }
```

上述描述很容易根据表 9-3 转换为 π 演算进程，例如 Tmerchant 的行为可表达为进程 $P=(\alpha x.\tau.\overline{\beta}y+\Psi z.\tau.\overline{\eta}k).P$ 。

3. *动态行为规约及其形式化解释*

在 D-ADL 中，动态行为规约是通过 choreographer 处理的。动态行为本质上是对体系结构的动态重配置，涉及以下体系结构变动：（1）动态创建新的构件实例和连接件实例，以及新的端口和通道；（2）动态删除构件实例和连接件实例，以及端口和通道；（3）体系结构元素之间连接的改变。choreographer 的语法规格如下：

```
choreographer ::= attach.choreographer|detach.choreographer|create.choreographer |
destroy.choreographer |Ecomputation.choreographer|inaction | replicate
detach ::= detach ComponenInstance^Port from ConnectorInstance^Eport |
detach ComponenInstance^Port^Channel from ConnectoInstance^Eport^Echannel
create ::= new [01 ComponentInstanc:]ComponentName[01 Actual-parameter list]|
new [01 ConnectorInstanc:]ConnectorName[01 Actual-parameter list] |
new [01 Port:]PortTypeName|new [01 Port^Channel:] ChannelTypeName
destroy::=delete ComponentInstance| delete ConnectorInstance| delete Port| delete
Port^Channel
```

attach 起建立新连接的作用，语法规格参见上一节；detach 起解除连接的作用；create 使用 **new** 关键字动态创建新的构件实例和连接件实例，以及新的端口和通道，其中，Actual-parameter 是实际参数列表，PortTypeName 是端口类型名，ChannelTypeName 是通道类型名；destroy 使用 **delete** 关键字动态删除构件实例和连接件实例，以及端口和通道；Ecomputation 是 computation 的扩展版本，由于动态行为以体系结构元素为处理对象，Ecomputation 可以处理包含体系类型数据在内的任意类型数据，而 computation 只能处理常规类型数据；Eprefix 是 prefix 的扩展版本，它们的区别与 Ecomputation 和 computation 之间区别是一样的。动态行为规约的高阶 π 演算解释见表 9-6。

表 9-6 动态行为规约的高阶 π 演算语义解释

<table>
<tr><th colspan="2">D-ADL 标记</th><th colspan="2">高阶多型 π 演算解释</th><th>备注</th></tr>
<tr><td>detach</td><td>detach…from…</td><td colspan="2">α 换名操作</td><td>关联的通道被 α 换名为不同的名字</td></tr>
<tr><td rowspan="2">create</td><td>new…ComponentName…
或
new…ConnectorName…</td><td rowspan="4">P 表示构件(连接件)的行为,P_a 表示构件(连接件)的计算行为(路由行为),P_b 表示构件(连接件)的动态行为，P_c 表示待创建构件(连接件)的行为,Q 表示运行环境行为,ζ 表示构件和运行环境进行通信的虚拟通道,θ 表示待创建或删除的端口(通道).P_{a1}，P_{a2},…,P_{an} 分别表示复合构件成员行为，P_{ak} 表示待删除的成员构件(或成员连接件)的行为.</td><td>$P=P_a+P_b$
$P_b=\zeta\chi.(\chi|P_a+P_b)$
$Q=\bar{\zeta}P_c.Q$</td><td>复合构件(复合连接件)中增加新的构件实例或新的连接件实例</td></tr>
<tr><td>new…PortTypeName…
或
new…ChannelTypeName…</td><td>$P=P_a+P_b$
$P_b=\zeta\chi.(P_a+P_b)$
$Q=\bar{\zeta}\theta.Q$</td><td>创建新的端口
或
创建新的通道</td></tr>
<tr><td rowspan="2">destroy</td><td>delete componentInstance
或
delete ConnectorInstance</td><td>$P=P_a+P_b$
$P_b=\bar{\zeta}P_{ak}.(P_{a1}|\ldots|P_{a(k-1)}|P_{a(k+1)}|\ldots|P_{an}+P_b)$
$Q=\zeta\chi.Q$</td><td>从复合构件(复合连接件)中删除构件实例或删除连接件实例</td></tr>
<tr><td>delete Port
或
delete Port^Channel</td><td>$P=P_a+P_b$
$P_b=\bar{\zeta}\theta.(P_a+P_b)$
$Q=\zeta\chi.Q$</td><td>删除端口
或
删除通道</td></tr>
</table>

对于增删构件实例、连接件实例以及端口和通道的行为，在进行形式语义映射时存在一个难点：π 演算没有对应 new 和 delete 关键字的操作符，因此不能将它们直接变换为 π 演算进程。我们的方法是将运行环境的行为纳入考虑。这是因为，在软件实际执行过程中，体系结构的变更行为总是由软件本身和运行环境共同参与完成，例如构件实例的创建可能需要中间件提供的服务或程序语言运行时库的支持。对运行环境行为的完整描述和形式化是一个复杂的问题，所幸这里只需对运行环境参与上述变更行为的相关动作进行形式化描述即可。

以复合构件增加一个新成员构件实例为例说明（**new** …*ComponentName*…），由表 9-5 知下述公式成立：

$$P = P_a+P_b \tag{1}$$

$$P_b = \zeta\chi.(\chi|P_a+P_b) \tag{2}$$

$$Q = \bar{\zeta}P_c.Q \tag{3}$$

（1）式说明复合构件行为是其计算行为和动态行为的选择组合；（2）式说明动态行为是首先经由 ζ 通道输入变量 χ，然后执行进程（$\chi|P_a+P_b$）；（3）式说明运行环境的相关动作序列：反复执行经由 ζ 通道输出进程 P_c 的动作。由于系统行为是体系结构行为和运行环境相关行为的并行组合，因此系统行为是：

$$P|Q = (P_a+P_b)|\bar{\zeta}P_c.Q = (P_a+\zeta\chi.(\chi|P_a+P_b))|\bar{\zeta}P_c.Q$$

按照 R-COM 归约规则，上式可归约为$(P_c|P_a+P_b)|Q$。即复合构件行为演化为$(P_c|P_a+P_b)$，而复合构件的计算行为由 P_a 演化为 $P_c|P_a$，意味着复合构件创建了一个行为为 P_c 的新成员构件实例。仿照以上处理的思路也能将体系结构的其他变更行为解释为高阶 π 演算进程。

如何使用 D-ADL 描述动态行为，我们也以前例进一步说明。假设订购服务器（merchant）发生错误而死机或崩溃时，系统需要自动重新启动一个服务器实例，并将客户请求导向新的服务器使服务不至于中断。这种具有自动切换功能的商品订购系统的体系

结构 D-ADL 描述如下：

```
    compositecomponent TDynamicOrderSystem () {
    port { environment: Tenvironment. }
    .
    .
    .
      choreographer {
    via environment^servermessage receive sign.
    if sign=0 then {
    detach merchant^port1 from cmlink^portl-m1.detach merchant^port2 from
cmlink^portl-m2.
    delete merchant.
    new merchant:Tmerchant().
    attach merchant^port1 to cmlink^portl-m1.attach merchant^port2 to
cmlink^portl-m2.  }
    replicate }  }
```

TDynamicOrderSystem 和 TOrderSystem 相比，增加了一个和运行环境进行通信的端口 environment 和动态行为规约 choreographer。前者是因为检测 merchant 发生错误的任务应由运行环境完成，后者描述订购服务器的动态切换过程：当运行环境检测到 merchant 发生故障时（sign 值为 0），通过 environment^servermessage 通道向 TDynamicOrderSystem 发送消息。TDynamicOrderSystem 得知消息后，解除 merchant 和 cmlink 之间的连接然后销毁 merchant，接着创建新的服务器（取名还是叫 merchant）并使之和 cmlink 建立连接。

9.2.4　D-ADL 对系统联机演化和 SA 求精的形式化支持

使用形式化工具的目标是协助系统设计、验证和演化。π 演算理论核心是行为的模拟和等价理论。对于行为模拟和等价概念的详细讨论超出了本书的范围，请参见文献。下面我们使用这些理论来探讨 D-ADL 对系统联机演化和体系结构模型求精的支持。

首先我们使用弱等价概念来讨论系统联机演化时构件替换的正确性。软件系统的各部分相互协作和相互通信，每一个成员都对和它进行协作的成员有一个期望的交互方式和行为约束。这意味着即使两个构件具有不同的内部结构和不同的内部行为，但从任何环境中取出其中一个而放入另外一个，环境不会感知替换发生。换句话说，在软件运行期间构件之间的相互替换不仅要使得它们的接口保持兼容，而且它们的外部行为也要保持一致。当然，无论是构件还是连接件，其外部可见行为是发生在接口处的交互活动，即在通道处定义的接收（receive）或发送（send）活动，其他活动都被视作内部活动。在 π 演算中定义了两种行为等价关系：强等价关系和弱等价关系。强等价关系既考虑系统的内部行为，又考虑系统的外部行为；而弱等价关系是一种不考虑系统的内部行为的行为等价理论，意指如果系统 P 和 Q 弱互模拟，则 P 和 Q 表现出来的外部行为是等价的。因此有：

规则 1：设构件 A 和 B 的行为分别被形式化为进程 P_a 和 P_b，则要使得 A 和 B 能够相互联机替换，必要的条件就是 P_a 和 P_b 具备弱等价关系。

D-ADL 以高阶多型 π 演算作为行为语义基础，因此构件的行为规约能够从 D-ADL 描述出发转换为高阶 π 演算进程，进而利用规则 1 和高阶 π 演算弱等价判定理论及其工具推导出构件

能否替换。以商品订购系统为例，若客户接收到商家同意其订购请求后，有两种选择：购买商品（向商家发出付款信息）和放弃购买（仅有内部处理动作）。新的客户构件描述如下：

```
atomiccomponent Tnewclient () {
port { portc1: Tcaccess. portc2: Tmaccess.}
computation  {
via portc1^order send orderdata.via portc1^response receive replydate.
if replydata^accept then
 { choose
{ via portc2^pay send replydata^payment. via portc2^confirm receive confirmation }, unobservable. }
replicate  } }
```

Tclient 的行为规约和 Tnewclient 的行为规约可分别转换为进程 P 和 Q:

$$P = \overline{\alpha}x.\beta y.[i=1]\overline{\delta}z.\eta h.P, \quad Q = \overline{\alpha}x.\beta y.[i=1](\overline{\delta}z.\eta h + \tau).Q$$

根据弱等价判定理论或使用 π 演算工具可推导出 P 弱等价 Q。因此，运行期间，由 Tnewclient 实例化的构件具备替换由 Tclient 实例化的构件的能力。

下面，我们再利用行为弱模拟理论来讨论 SA 行为求精问题。求精是软件开发中的基本活动，在 SA 的构造过程中，如果 SA 的抽象粒度过大，就需要对 SA 进行求精、细化，行为求精又是 SA 求精活动的核心。行为求精的基本原则是维持不同层次行为信息的一致性。具体来说，虽然求精结果可能表现出一些新的行为，但环境对求精对象的行为期望应该被求精结果所维持。使用高阶 π 演算的行为模拟术语表示，就是求精结果可以弱模拟求精对象的行为，但求精对象无需弱模拟求精结果的行为。这种弱模拟关系在构件行为求精和连接件行为求精上又有不同的具体体现。

在 π 演算中，弱模拟实际上还可细分为观察弱模拟和分支弱模拟两种，通常所说的弱模拟概念指的是观察弱模拟。观察弱模拟意指对不可观察的活动序列进行抽象，并跟踪每一个可观察的活动。对构件的行为求精,着眼点在于外部观察行为之间的观察弱模拟关系。

规则 2：设进程 PA 表示构件 A 的行为，进程 PB 表示构件 B 的行为，若构件 A 是对构件 B 的求精，则 PA 观察弱模拟 PB。

分支弱模拟意指对不可观察的活动序列进行抽象,在跟踪每一个可观察的活动的同时，保持进程的所有分支轨迹。连接件旨在建立构件间的交互，而交互的核心体现为路由行为，连接件求精应该保持路由行为的一致。

规则 3: 设进程 PC 表示连接件 C 的行为，进程 PD 表示连接件 D 的行为，若连接件 C 是对连接件 D 的求精，则 PC 分支弱模拟 PD。

还以商品订购系统为例说明行为求精规则。在接收到客户订购请求后，商家根据情况确定是否能够满足订购请求的实际过程是：订购服务器向仓储服务器查询是否有足够供货。因此，Tmerchant 求精如下：

```
atomiccomponent Tmerchant () {
port { portm1:Tcaccess. portm2: Tmaccess.portm3:Tinquire}
computation  {
choose {
```

```
    { via portm1^order receive orderdata.   via portm3^inquire send orderdata.
      via portm3^answer receive result.
      if result then
    { unobservable. via port1m^response send record(true,payment)}
    else
    { unobservable. via port1m^response send record(false,0)} },
    {  via  portm2^pay  receive  payment.unobservable.via  portm2^confirm  send
confirmation }}
    replicate  }  }
```

求精后，Tmerchant 添加了一个新的端口 portm3:Tinquire，用于向仓储服务器查询是否有足够供货（inquire)。当 Tmerchant 接收到仓储服务器的肯定答复时（result 值为真），将向客户发出同意订购请求和应付款信息，否则拒绝客户请求。求精前后的 Tmerchant 行为规约可分别转换为进程 P 和 Q：

$$P=(\alpha x.\tau.\overline{\beta}y+\Psi z.\tau.\overline{\eta}k).P,\quad Q=(\alpha x.\overline{\delta}e.\eta f.([f=1]\overline{\beta}g+[f=0]\overline{\beta}h)+\Psi z.\tau.\overline{\eta}k).Q$$

根据观察弱模拟判定理论或使用 Workbench 可以推导出 Q 观察弱模拟 P，但 P 并不观察弱模拟 Q。由此可知，求精活动符合规则 2，保持了求精前后行为的一致性。

9.2.5　D-ADL 和其他相关工作的比较

动态 ADL 的研究首先体现在将动态配置功能集成在 ADL 中。为开发能随着环境和需求变化而动态自适应的系统，J.Dowling 等人设计了 K-Component 框架元模型。在 K-Component 元模型中，一个有类型的配置图被用来表示 SA，其中节点表示构件接口，类型标签表示构件，边表示连接件。基于关注分离这一软件工程基本原则，尽量使演化逻辑与计算逻辑解耦，K-Component 元模型提供一个专门的机制“adaptation contracts”来显式地表达演化逻辑，而不将其固化于程序语言或支撑平台之中，从而使得演化逻辑可编程和动态地修改。K-Component 元模型的缺陷是配置图虽然直观，但并不能严格和全面地表达 SA 的行为语义和结构语义。C2 是一种基于构件和消息的体系结构风格，为体系结构的演化提供了特别的支持。C2 的动态管理机制使用专门的体系结构变更语言 AML（Architecture Modification Language）。在 AML 中定义了一组在运行时可插入、删除和重新关联体系结构元素的操作，如 addcomponent、weld 等。但 C2 也没有严密的形式化基础，不能对体系结构的动态行为进行严格的分析和推演.Rapide 是 Luchham 等开发的体系结构描述语言，基于偏序事件集（Partially Ordered Event Sets）对构件的计算行为和交互行为进行建模，允许在 where 语句中通过 link 和 unlink 操作符重新建立结构关联。Rapide 的连接机制嵌入配置定义当中，两者不能分离，Medividovic 将它称为嵌入配置 ADL（In-line Configuration ADL）。因此，Rapide 不允许单独对连接件进行描述和分析，并且没有提供相关机制，将多个连接机制捆绑成为一个整体，构成复杂的交互模式。因此 Rapide 描述构件交互模式的能力存在不足。

Wright 是 R.Allen 等开发的静态 ADL，它的形式语义基础是通信顺序进程 CSP（Communication Sequence Process），支持 SA 的静态分析。Dynamic Wright 使用标签事件（Tagged events）技术对 Wright 进行扩展从而支持对动态 SA 建模和分析，虽然 Dynamic Wright 长

于进行诸如死锁检测、模型一致性验证等工作，但对动态行为的表达力不足，同时基于 CSP 的特点，难以进行行为模拟和等价判定工作。Darwin 是 J.Magee 和 J.Kramer 开发的动态 ADL，提供延迟实例化（Lazy instantiation）和直接动态实例化（Direct dynamic instantiation）两种技术支持动态 SA 建模，允许事先规划好的运行时构件复制、删除和重新绑定。Darwin 虽然运用简单的一阶 π 演算提供构件计算和交互规约的语义，但其动态机制并不提供任何 π 演算语义，因此 Darwin 不能提供动态行为形式化分析的基础。

欧盟的 ArchWare 项目旨在以体系结构模型为中心构造可演化的软件系统。ArchWare 提出了一个动态体系结构描述语言 π-ADL。 π-ADL 也是一个基于高阶 π 演算的形式化语言，支持动态体系结构建模和验证。D-ADL 借鉴了 π-ADL 的一些思想，但和 π-ADL 有如下不同：（1）D-ADL 显性定义了体系结构的动态行为操作符“new”和“delete”，π-ADL 则借助 π 进程的输入输出动作间接实现动态行为，虽然 D-ADL 的动态行为操作符的语义解释也归之于 π 进程的输入输出动作，但从语用学角度看使用 D-ADL 更便于动态行为的描述和理解；（2）D-ADL 将动态行为与计算行为相分离，使得动态体系结构的行为视图更加清晰，同时由于动态行为具备高阶 π 演算语义，能够预先推导动态行为的结果，因此可采取措施使这种分离不影响计算行为的正确性；（3）π-ADL 直接支持体系结构的演化（规约变化，通过 Composition /Decomposition 算子实现），D-ADL 本身仅实现了对体系结构动态行为（规约不变）的描述，但对体系结构的演化也有间接的支持。

9.3 SASM 模 型

9.3.1 相关研究

近年来，许多研究者关注在软件体系结构的指导下维护和演化系统，注意到在系统运行时刻显式地维护体系结构信息的重要性。其中的关键原因是：网络的开放性和动态性导致软件的演化性和复杂性进一步增强，越来越多的系统需要能够对自身进行动态调整；而体系结构包括构成系统计算单元的构件、规范构件间交互行为和约束关系的连接件、构件和连接件形成的连通图结构，因此软件体系结构揭示了系统的变化性，提供了一种在较高抽象层次观察系统并推理系统行为和性质的方式。可见，运行时刻软件体系结构相关信息的改变可用来触发、驱动系统自身的动态调整。

为开发能随着环境和需求变化而动态自适应的系统，J.Dowling 等人设计了 K-Component 框架元模型。在 K-Component 元模型中，一个有类型的有向配置图被用来表示应用系统的软件体系结构，其中图节点表示构件接口，类型标签表示构件，有向边表示连接件。一个反射机制被设计用来在这个有类型的有向配置图和目标系统间建立因果连接，使得配置图反映系统的真实软件体系结构。

基于关注分离（separation of concerns）这一软件工程基本原则，尽量使适配逻辑与计算逻辑解耦，K-Component 元模型提供一个专门的机制“adaptation contracts”来显式地表达适配逻辑，而不将其固化于程序语言或支撑平台之中。“adaptation contracts”使用一个

叫做“adaptation contract description language”的描述语言书写，从而使得适配逻辑可编程和动态地修改。配置管理器提供“adaptation contracts”的部署和运行环境，允许“adaptation contracts”的动态装载、修改和卸载。一旦需要动态调整系统，适配事件产生，配置管理器依据“adaptation contracts”调用重配置操作，而重配置操作的结果将导致系统配置图的修改和迁移，进而更改系统本身。K-Component 元模型的缺陷是配置图虽然直观，但并不能严格和全面地表达软件体系结构的行为语义和结构语义。

ArchStudio 是基于软件体系结构的开发和运行环境，支持 C2 体系结构风格软件的动态修改。ArchStudio 所定义的系统动态演化方法是如何使在体系结构层面表达的动态调整在具体系统中得以实施的一个典型代表，图 9-3 说明了这个方法。

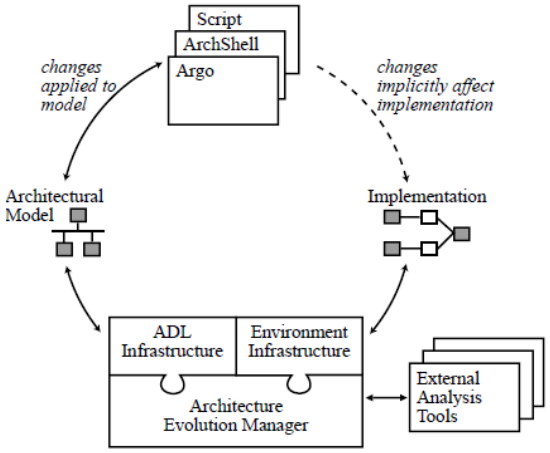


图 9-3　ArchStudio 的动态演化模型

在 ArchStudio 中，运行时修改由一系列的查询和变更组成。AEM（Architecture Evolution Manager）维护在体系结构模型（Architectural Model）和系统实现之间的一致性。AEM 决定对系统的更改是否有效，使用体系结构约束机制或外部分析工具来决定变更是否可接受。一旦 AEM 接受变更请求，首先通过 ADL 基础设施（ADL infrastructure）对体系结构进行修改，进而通过环境基础设施（Environment infrastructure）把体系结构层次的更改映射到系统实现上，从而驱动系统自身的动态调整。

目前，ArchStudio 包含三种体系结构变更源（Sources of Architectural Modification）工具：Argo、ArchShell 和扩展向导（Extension Wizard）。Argo 提供一个体系结构的图形描述和操作手段，ArchShell 提供一个文本的、命令式的体系结构变更语言，扩展向导提供一个可执行的脚本更改语言用来对体系结构进行连续演化。

但 ArchStudio 采用的软件体系结构是设计阶段的制品，而且仅仅作为一种辅助文档，无法刻画系统真实的运行状态和行为，而且仅限于使用 Java 编写的遵循 C2 体系结构风格的软件动态修改，演化的合理性与正确性难以保证。

北京大学的 PKUAS 系统是基于体系结构反射的中间件系统，其目的是解决现有中间件系统过于注重系统局部或单个实体的反射而缺乏全局视图的问题，及过于注重中间件平台内部功能的反射而对上层应用反射不够的问题。对此 PKUAS 采用基于构件化的平台内部体系结构，并引入运行时软件体系结构（RSA）作为全局视图以实现反射体系对系统整

体的表示和控制。

PKUAS 系统中对反射的实现主要通过元模型、元协议和元数据三者进行。PKUAS 的元模型为各种容器和各种服务 Mbean，用于反射基层实体，管理部署信息和运行上下文导出元数据；元数据是由设计阶段的软件体系结构，部署阶段的部署信息以及运行阶段的上下文导出并累积而成的 RSA；元协议包括服务反射元协议和构件反射元协议，前者负责服务元数据的访问和修改，并执行装载、启动、停止、释放、替换和动态配置等动作。后者主要通过构件容器来负责对构件元数据的访问和修改，并维护元数据与构件状态和行为的因果关联。PKUAS 通过扩展传统的体系结构描述语言 ABC/ADL 描述 RSA，并使之具备继承设计阶段体系结构所富含语义的天然能力。但由于 ABC/ADL 缺乏构件行为和交互的形式化描述，它并没有解决构件替换时的可观察行为的一致性问题。

还存在其他的一些基于体系结构的动态演化模型，如 Cazzol 等人提出的体系反射，着重考察软件体系结构与运行系统之间的关系；Crarlan 等人利用度量计评估运行系统的状态和行为然后调整系统；南京大学提出的一种内置运行时体系结构对象动态协同架构。但所有这些工作都存在一些问题，主要表现为对 RSA 的行为语义描述不够严密，不能保证系统的完整演化和演化过程的一致性。

9.3.2 基本原理

实际上软件体系结构不仅仅只是作为设计概念，任何可运行的软件系统都存在客观上的软件体系结构，这种客观存在的运行时软件体系结构是作为设计概念的软件体系结构的具体实现。但是目前存在的一个主要问题，就是在软件体系结构概念和具体实现之间存在语义鸿沟。主要表现为许多结构问题被分散实现在应用代码中，甚至被嵌入到中间件平台和操作系统内部，以及语言的运行时支持库中。连接件嵌入在构件的代码片断内，而它依赖于中间件提供的 API 和操作系统原语，那样的接口被实现在平台内部，细节对用户不可见。因此，结构的问题被传播到操作系统和中间件内，而不再平台独立。这导致了以下几个问题：

（1）软件体系结构不能在应用层次上被观察和控制，用户无法获知软件系统的体系结构信息如拓扑结构、部件状态和数目。

（2）软件系统难以在运行时动态重配置，使得系统的可扩展性和灵活性不够，难以联机演化。例如插入新的部件、修改部件间通信协议等。

（3）难以适应灵活多变的复杂需求。从管理层到操作层的各种角色的需求变化，抽象层次和视图各不相同，要求同时满足和适应这些需求变动，对软件系统的构造提出了更高的要求。

究其原因，这是因为在实现和运行阶段软件体系结构本身从整体上来说在系统中没有对应的一类显性实体，而以隐性的方式存在。即使在设计阶段软件体系结构被良好地构造，在实现阶段和运行时也不能被清晰可见。因此，我们工作的基本出发点就是：使得软件体系结构在可运行应用系统中以一类实体显性地表示，并且被整个运行环境共享，作为整个系统运行的依据。也就是说，运行时刻体系结构相关信息的改变可用来触发、驱动系统自身的动态调整。

软件演化产生的动力主要来源于两个方面：一是需求的变化；二是设计的变化。这两方面又是密切相关的。一个较复杂的系统，需要满足从管理层到操作层不同层次的用户角色的需求，而这些需求抽象层次和视图各不相同，具有不同的表现方式：目标、任务、功能和过程等。考察一般的软件开发过程，软件开发过程应当是在不同抽象级别考虑和处理问题的过程，开发过程的每一步都是对较高一级抽象的解作一次较具体化的描述。由于用户的需求是主观的和多层次的，因此不可能一开始就将它们进行完整描述，总是从最初的较抽象的、宏观的需求出发。随着开发进程的不断发展，需求逐渐被细化、具体化。在每个阶段，依据某种设计策略，这个阶段的需求驱动本阶段软件制品的产生。当然，不同的设计策略，即使相同的需求也可能产生不同的软件制品。但总的说来，某个阶段的软件制品总是为了满足相应的需求而开发的。因此，软件开发过程中各个阶段的软件制品也具有不同的抽象层次，也是不同抽象层次的用户需求的映射。

进一步考察软件的静态演化过程，也就是一般所说的软件维护过程。一方面，为满足和适应管理层或操作层的不同层次角色的需求变化，总是从这个层次的需求所对应的软件制品入手，对其进行变更，并使这种变更逐次迁移繁殖，影响下阶段的软件制品，使之做出响应上阶段变更要求的设计变更。另一方面，由设计策略的变化而导致的静态演化，最终也落实到某阶段的软件制品的变更，进行和由需求变化引起的静态演化相同的过程。由此可见，软件静态演化必须依赖丰富的语义信息，尤其是那些软件设计过程中产生的信息，否则，静态演化的合理性与正确性难以保证。

本节提出的面向动态演化、基于体系结构的软件模型 SASM（Software Architecture Space-Based Model）正是基于以上认识提出来的。SASM 的出发点是在可运行系统中尽量多地保留设计期的建模信息。由于软件体系结构模型集中反映了开发者的设计决策，提供了一种观察系统并推理系统行为和性质的方式，SASM 使用软件体系结构模型作为系统设计期的建模信息载体。又由于设计的层次性，我们在构造这种体系结构模型时也使之具有多层表示。不同层次的体系结构反映不同用户角色的要求，提供给用户不同抽象级别的体系结构视图和管理手段。这样的体系结构模型是一个体系结构空间，同时又是运行时可操作的，我们称之为运行时体系结构空间（Runtime Software Architecture Space，RSAS）。RSAS 作为一个在运行时刻有状态、有行为、可访问的对象，在系统实现中直接地、集中地反映软件体系结构规约，从而在最终实现中保留了体系结构决策，极大地提高了系统的构造性，为系统投入运行以后的演化行为提供了必要的依据和支持。

存在的关键问题是如何使 RSAS 准确地描述目标系统的真实状态与行为，如何使对 RSAS 的改变能够直接导致运行系统的相应变化。反射技术是一个自然的选择。在 SASM 中，RSAS 以元层形式存在，是关于系统本身的表示，这种表示可以被用户检查和调整；可运行的二进制物理构件（可能是 EJB、COM 构件、CORBA 构件或 Web Services）是基对象，一系列基对象形成基层，执行应用领域中的功能。元层和基层因果相联，形成一个反射体系，从而解决上述问题。

9.3.3　SASM 框架

基于以上思想，SASM 模型的总体框架被设计为如图 9-4 所示。整个框架由因果相联

的元层和基层组成，形成一个反射系统。

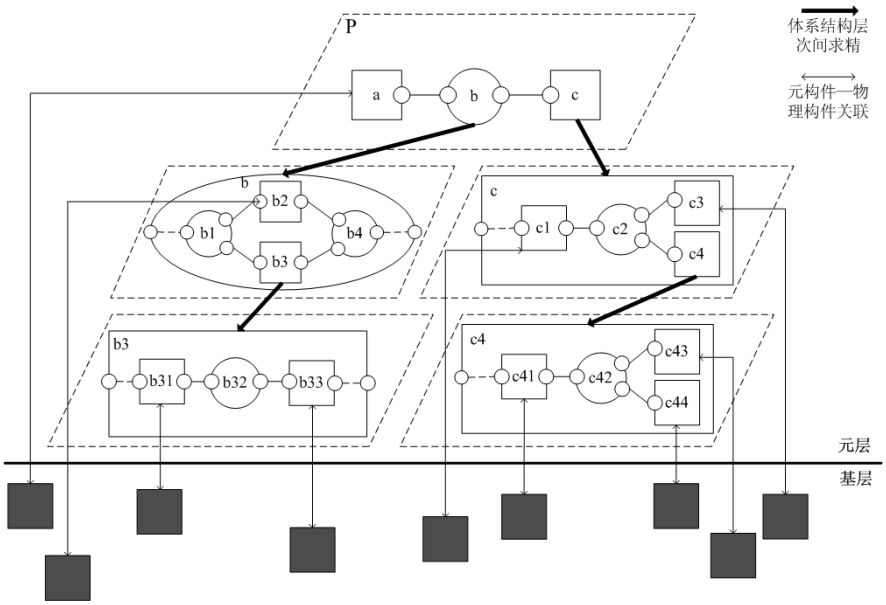


图 9-4　SASM 模型框架

元层由 RSAS 模型构成，描述系统软件体系结构语义，是应用领域分析设计的结果，使用 D-ADL 语言描述。作为元层的 RSAS 模型是一个由不同层次的体系结构形成的信息空间。其中，不同抽象层次的体系结构之间存在求精关系，高层次体系结构是低层次体系结构的抽象、约束，低层次体系结构是高层次体系结构的具体化和实现。鉴于隔离求精方式的特点，RSAS 模型采取隔离求精方式构造。隔离求精使得高层次体系结构中某单一构件（单一连接件）可能被求精为低层次中具有内部结构的某一复合构件（连接件），但后者的外部观察行为（路由行为）和前者保持一致，后者的属性信息满足前者的属性约束。

依据隔离求精的特点，RSAS 模型的层次性表现为树或森林的形状，树或森林的所有叶子节点可联结在一起形成应用系统的最具体体系结构。在 RSAS 模型中，构件叶子节点被称之为元构件，是软件功能的领域概念单元，描述了应用领域的计算语义和约束；连接件叶子节点被称之为元连接件，显性表示构件之间的交互行为和协作语义。例如图 9-4 中 a,b1,b2,b31,b33,c1, c3,c41,c43,c44 为元构件，b1,b4,b32,c2,c42 为元连接件。

基层由可运行的二进制物理构件形成，真正执行应用领域中的单元计算。在 RSAS 中的每一个元构件都绑定一个物理构件，元构件是对应物理构件的信息映象，物理构件是其对应元构件的实现承载体。

在开放的环境下，交互模式应该根据应用和环境的特点而设，并可能需要动态调整。对于普通的二进制物理构件，如 EJB、CORBA 构件、DCOM 构件等，构件之间交互是通过构件或对象引用进行，导致构件互相依赖和通信协议被固化在构件内部。这个缺陷使得整个应用难于联机演化，尤其是结构动态重配置。SASM 模型中的物理构件，也遵循构件工业标准实现，如 EJB 规范、CCM 规范，因此在一定条件下也可以直接部署到 J2EE、CORBA 中间件平台或 DCOM 平台上，向外界提供服务。但这些物理构件实现时其内部不包含其他

构件的接口信息，不相互直接引用，是相对独立的。这使得构件之间耦合度降低，对物理构件的替换可以不影响到其他构件。

所有要求其他物理构件提供的服务都依据元连接件规约进行。交互事件发生时，运行环境从物理构件中接过控制权，分析传输过来的数据，解析执行元连接件规约所定义的构件间通信协议、信息交流和数据转换等功能。同时运行环境依据 RSAS 描述语义，定位、调用其他物理构件。图 9-5 说明了这个过程。

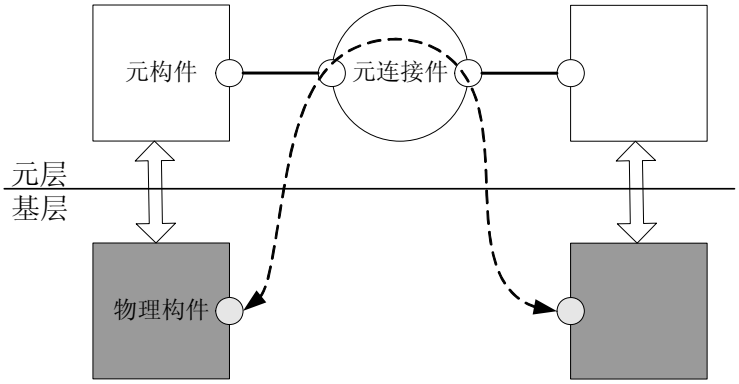


图 9-5　物理构件之间的交互

因此，元连接件规约实质是物理构件之间的黏结剂和交互的线索。变更元连接件规约将导致物理构件间的交互和协调过程随之改变，交互可根据当前的体系结构配置情况动态地决定含义。同时元连接件规约也可以附加信任控制、事务处理、QoS 管理等非功能属性。

9.3.4　SASM 中的反射机制

实现反射的关键是设计如何在运行期检查和调整系统的机制。通过检查可以观察到系统的当前状态；而调整可以在运行期改动系统的行为以更好地适应系统当前的执行环境。

要考察 SASM 模型中反射机制的具体情形，我们先从符合 SASM 模型框架的应用系统的构建过程谈起。图 9-6 说明了这个构建过程。

从初始宏观需求出发，随着对系统认识和分析的不断深化和细化，需求不断被求精而具体化。需求驱动着体系结构的设计，宏观需求转化为高层体系结构设计，微观需求转化为低层体系结构设计。同时，低层体系结构也是高层体系结构的求精（采取隔离求精方式）结果。所有层次的体系结构规约形成一个体系结构空间。把这个体系结构空间部署到面向动态演化、体系结构驱动的运行环境中，就初步得到了应用系统的元层：RSAS 模型。对每一个 RSAS 中元构件依据其规约设计实现一个物理构件，或者从物理构件库中检索符合元构件规约的物理构件，所有这些物理构件形成应用系统的基层。至此，符合 SASM 模型的应用系统得到建立。

可见，一方面，作为元层的 RSAS 模型描述了系统的体系结构规约，在系统中保留了体系结构设计决策。通过对 RSAS 的观察，可获知系统的结构、行为知识。另一方面，物理构件是其对应元构件的实现承载体，元构件是对应物理构件的信息映象。因此，在元层

和基层中建立起了初步的反射关系，这种反射既有结构的反射又有行为的反射，为系统投入运行以后的演化行为提供了必要的依据和支持。

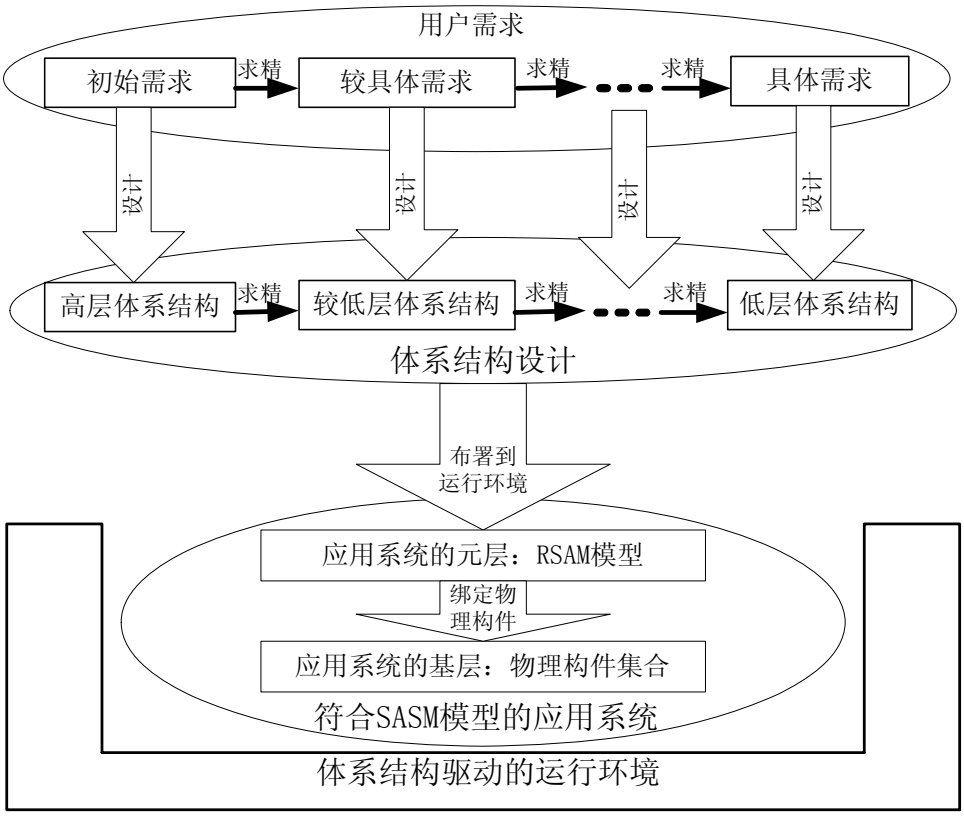


图 9-6　SASM 应用系统的构建过程

进一步考察系统的运行时情况。当某一个物理构件正处于运行态时，运行环境跟踪、监控这个物理构件的执行，收集它的运行信息，如运行线程数、缓存大小、构件状态等，并把这些信息实时地存入其对应元构件规约中的动态属性值中，以反映系统的当前运行情况。运行环境也对这些信息进行统计和分析，以判断物理构件的运行是否违反了元层 RSAS 规约中的行为约束，并向用户提出警报和建议。同时由上节可知，物理构件间的通信通过对元连接件规约的解析执行完成，可理解为通信发生在元层，从而建立了一个元通信反射机制。

现在考虑元层 RSAS 发生变动的情形。当元构件规约改变时，要重新进行物理构件的绑定过程，新的物理构件将引起系统单元计算过程的变更；当元连接件规约改变时，发生在物理构件间的交互和协调过程随之改变。总之，对元层 RSAS 的调整，毫无疑问将导致系统的演化。

可见，在运行期间，通过元层 RSAS 模型实现了对基层的观察和修改，元层和基层之间因果相联。一方面，通过检查元层 RSAS 模型，除获得系统的结构知识和行为规约外，还可获知系统的运行状态和信息。另一方面，对元层 RSAS 的调整实现了对基层进而系统的修改。

9.3.5　SASM 系统开发

在 SASM 模型中，物理构件是遵循某种工业标准（如 EJB、DCOM 或 CORBA）的二进制构件，因此其开发和一般 EJB、DCOM 构件或 CORBA 构件的开发几乎是一样的。一般 EJB、DCOM 构件或 CORBA 构件的开发可参阅相关资料，这里不再赘述。

其不同点在于：普通的 CORBA 构件、DCOM 构件、EJB 等，构件之间交互是通过构件或对象引用进行。物理构件实现时内部包含其所调用的其他构件的接口信息。编译器引入其他构件的接口信息如 IDL，然后生成调用桩模块或骨架模块，最后编译进构件内部。这使得物理构件的实现、运行紧耦合于被调用物理构件所部署的位置、构件标志，这些信息稍有更动，物理构件将不能正常运行。但 SASM 物理构件实现时不直接引用其他构件，其内部不包含其他构件的接口信息，是相对独立的。物理构件之间的交互行为和协作语义被元连接件规约显性表示，独立于物理构件的实现。

因此，为开发 SASM 物理构件，关键在于实现 SASM 物理构件对外要求功能的调用点。实际上，对于 SASM 物理构件来说，在调用点处真正需要的只是传出参数，获得被调用物理构件的运算结果。因此，调用点只定义要传出的参数和要获取的结果类型以及其他要说明的语义约束，把实际调用委托给运行环境完成，由运行环境根据元连接规约，完成被调用的构件寻址和实际的调用工作。对程序员来说，构件运行环境提供“调用点环境化”编程模型，程序员使用这种编程接口调用对外要求的功能不需要知道功能提供者位置，甚至不需要知道被调用构件名，只需传递操作名、输入参数和返回数据类型及其他可选信息。为开发符合这样规格的 EJB，我们扩展 J2EE 平台实现了这样的编程接口，提供实施实际的调用类 Transferspot 及其类方法 Transferspotdefine，Transferspotdefine 原型如下：

Transferspotdefine（String interfaceName，String operateName，Array inputData，Array resultData）；

InterfaceName 指示构件对外要求的功能接口名，operateName 指示构件要调用的操作名，Array 是 java 数组类，可动态构造和容纳各种类的对象，inputData 和 resultData 分别表示调用点输入参数列表和返回结果，元素类型为 Variant，一种存储可变数据类型的 java 类，以适应输入输出参数的类型多样性。

Transferspotdefine 方法并不指定目标构件名。调用发生时，运行环境截获 Transferspotdefine 方法，借助中间件平台（如 J2EE 平台）提供的动态调用服务，依据元连接件规约定位目标构件进行实际交互过程。调用结果置于 Resultdata 数组中返回给物理构件。

这样做，使得物理构件间的交互过程都统一为由运行环境实施，调用点没有硬绑定到被调用物理构件上，被调用物理构件可进行动态替换和位置迁移，降低了物理构件的耦合度，有利于应用的动态演化。

9.4　SASM 动态演化方法

本节讨论在 SASM 软件模型中是如何对系统演化进行支持的，主要是从方法层次进行阐述，着重动态演化的方法，有关一些实现机制留待下一章讨论。

9.4.1 简单的系统演化

简单的系统演化是变更仅仅发生在基层实体上的演化。在 SASM 模型中，每一个元构件在基层都应该有一个作为其实现载体的可执行物理构件。对于一个特定的元构件，只要物理构件遵循它的规约，包括接口定义和行为方式，就可以成为它的基层载体。显而易见，一个元构件规约可以以不同的细节和算法来实现，一个元构件可以有多个符合要求的候选物理构件。因此，使用一个新的候选物理构件替换掉原来的物理构件将引起系统的演化。例如物理构件 a1、a2 都符合元构件 A 的规约，元构件 A 首先使用物理构件 a1 作为基层实现载体，其后使用物理构件 a2 替换掉 a1，则系统因为物理构件的替换发生了演化。这种演化有可能使系统的运行性能提高，资源消耗更小，因为 a2 比 a1 的内部算法可能更为精巧。但这种演化并不改变系统的结构特性和功能特性，例如既不增加功能也不减少功能。这是因为这种演化并不涉及到元层 RSAS 规约的变更。

当运行时进行基层物理构件的替换时，要考虑的情况稍微多些。不仅要评估用来替换的物理构件是否符合元构件规约，还要处理进行替换的物理构件的运行时状态问题。后者又分为两个部分：（1）被替换物理构件在什么状态下才能被替换，即变更发生的时间；（2）替换时被替换物理构件的状态如何迁移到用来替换的物理构件中。由于本章主旨是讨论 SASM 模型本身，这些问题的解决涉及运行环境的设计原理和运行机制，我们留待下章讨论。

9.4.2 由 RSAS 变更引起的动态演化

在 SASM 系统中，更常见的演化情形是由元层中 RSAS 的变更而引起的动态演化。RSAS 作为一个在运行时刻有状态、可访问的对象，在系统中直接地、集中地反映软件体系结构规约，从而促使系统开发人员从系统整体的层面理解和支持演化行为，在 RSAS 上所施行的更改操作通过 SASM 的反射机制最终影响到基层的变更，导致系统的演化。

RSAS 是一个由不同层次体系结构规约形成的体系结构空间，不同层次的体系结构规约反映了不同抽象级别的体系结构视图。为提供给用户不同抽象级别的系统演化管理手段，RSAS 上的变更可以发生在每个层次上，每一层次都可进行构件、连接件的增删、替换。例如图 9-7 可视化了图 9-4 中 RSAS 模型的树形层次结构的变更，图 9-8 可视化了对应体系结构空间的实际变更。

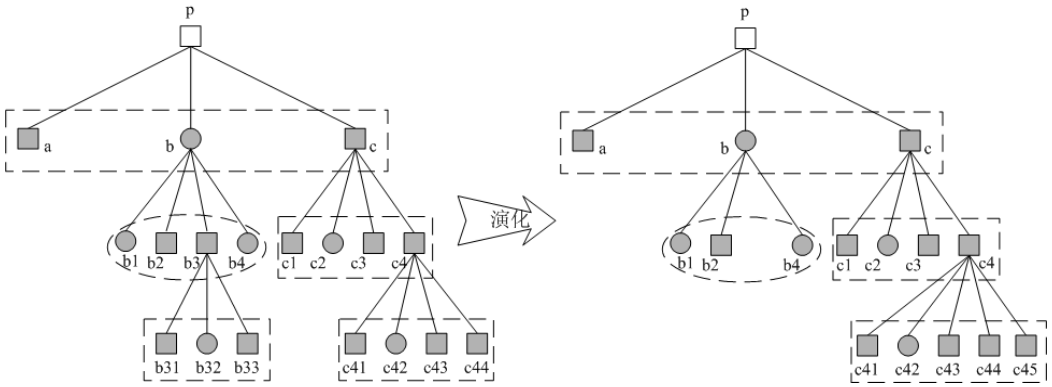


图 9-7 RSAS 模型的树形层次结构的变更

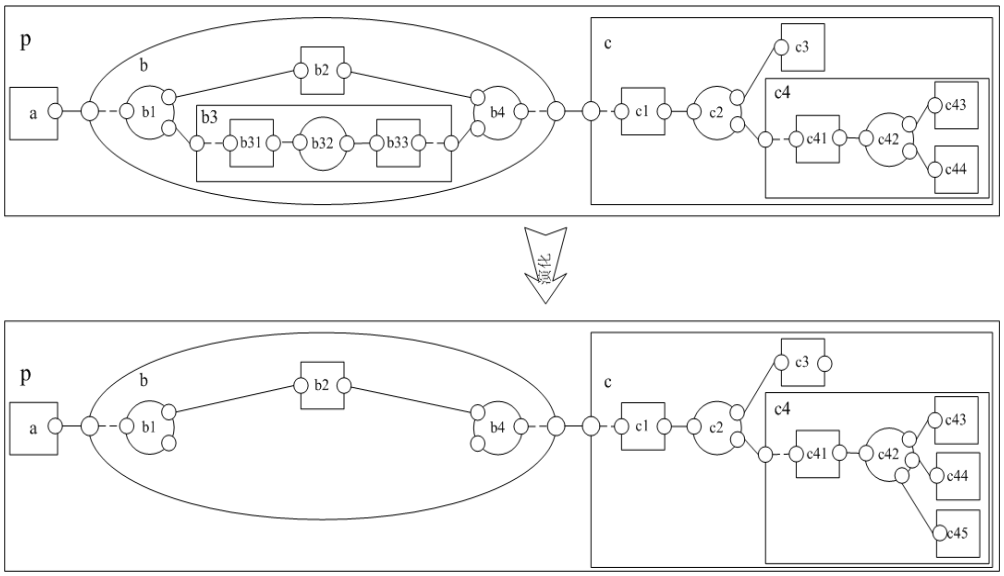


图 9-8　体系结构空间的实际变更

这种变更是以一种受控的方式进行：一方面，由于在构造过程中，我们知道 RSAS 模型的层次结构是一个树形结构，其中每个节点本身都是单一构件或单一连接件，由某个节点求精所得的所有下层节点共同可组成一个复合构件或复合连接件。在 RSAS 上发生的变更正是以这些复合构件（复合连接件）为变更对象，对其包含的成员构件、成员连接件做增删、替换操作，以及在成员构件、成员连接件之间作关联或解关联操作，这些操作的结果还是复合构件（复合连接件）。另一方面，由于层次之间存在着求精关系，高层次体系结构是低层次体系结构的抽象、约束，低层次体系结构是高层次体系结构的具体化和实现。因此发生在某个层次上的变更不仅受到上一层次的约束，也将影响繁殖到下一层次。上一层次的约束意指变更发生前后的复合构件（复合连接件）都应该是其上层对应单一构件（单一连接件）的求精结果；影响繁殖到下一层次意指当复合构件（复合连接件）包含的某个元素删除时，由其衍生的下层节点都应删除，当复合构件（复合连接件）中增加或替换某个元素时，如果这个元素有进一步的求精，求精结果应该反映在这个元素衍生的下层节点上。

例如图 9-7 中 b3 的删除将引起 b31、b32、b33 的删除，同时删除 b3 后由 b1、b2、b4 组成的复合连接件还是 b 连接件的求精结果，即 b1、b2、b4 组成的复合连接件行为分支弱模拟 b 连接件行为。同样的道理，增加 c45 构件后，由 c41、c42、c43、c44 和 c45 组成的复合构件还是 c4 构件的求精结果，即这个复合构件计算行为观察弱模拟 c4 构件计算行为。

针对产生机制的不同，RSAS 上的变更可分为两种形式：预设变更和非预设变更。

预设的变更意指事先可知的变动，变动可实现为系统的固有功能，成为应用的一部分。在 SASM 模型应用中，通过在 RSAS 上定义预设的变更策略，以应对预设的动态演化。RSAS 规格使用 D-ADL 语言描述，而在 D-ADL 中，这种变更的表示是通过复合构件（复合连接件）的 choreographer 段处理的（请参见 9.2 D-ADL 语言）。choreographer 段首先定义触发变更产生的事件（通常是某个输入事件），然后定义变更的处理策略（动态创建、删除构件、连接件、端口以及通道，变更体系结构元素之间的连接）。需注意的是，choreographer 段规

约本身是一个高阶 π 演算进程，而借助高阶 π 演算进程的迁移公式，可归约得到进程执行结果。因此，choreographer 段规约对复合构件（复合连接件）进行变更或重配置，可事先通过计算得到这种变更或重配置的结果（新的复合构件或复合连接件）。这种结果可用来评估是否符合其上层的约束，即是否还是上层单一构件（单一连接件）的求精结果，以决定这种演化计划是否有效。

在 SASM 模型中，虽然元构件行为规约是通过一个实现载体——基层物理构件执行完成的，但复合构件（复合连接件）的 choreographer 段规约是通过运行环境对其直接解析执行的，这和元连接件规约的执行方式相同。运行过程中，当变更事件发生时，运行环境首先执行变更处理策略，修改这个复合构件（复合连接件）的配置，然后把这种变更向 RSAS 的下层繁殖。

非预设的 RSAS 变更是通过对单一构件（单一连接件）的结构和行为直接进行不同的求精来完成的，为非预设的软件动态演化提供支持。对于 RSAS 树形层次结构中任意的一个节点（代表单一构件或单一连接件），不同的求精方案，可得到不同的求精结果（复合构件或复合连接件）。因此，对某个非叶子节点采取和原来不同的求精方案得到一个新的求精结果，使这个新的求精结果取代原来的求精结果，从而可得到新的 RSAS 模型。表现在 RSAS 树形层次结构上就是从这个非叶子节点上衍生的代表原来的求精结果的子节点被代表新的求精结果的子节点所取代。求精方案的改变，既可源于用户某个需求的变动，也可源于设计策略的变动，而这些变动可能是事先无法预估的或不能完全预估的。

无论是预设的还是非预设的变更，在 RSAS 上的变更发生后，新增加的元构件需要绑定、载入符合其规约的物理构件，已删除的元构件需要从基层中销毁其对应的物理构件。

对照 Carlos E. Cuesta 等人三个级别的软件体系结构动态性的划分来看，本文的方法支持所有级别的动态性。不仅允许结构动态性，即构件和连接件实例的创建、增加和删除，也支持体系动态性，即结构可以被重定义，如新的构件类型的定义。这超越了一般的动态软件体系结构研究仅支持结构动态性的局限。

9.5 小　　结

演化性高的软件，其设计具有良好的构造性，而构造性和软件采取的应用模型密切相关。本章提出的软件模型 SASM 旨在提高软件的构造性，以更好地支持软件动态演化。

SASM 使用反射技术构造，其元层由一个具有树形层次的体系结构空间 RSAS 组成，以反映不同用户角色的需求，保留体系结构设计和决策信息，从而为系统投入运行以后的演化行为提供充分的依据和支持。基层由可运行的物理构件形成，真正执行应用领域中的单元计算。物理构件彼此独立，不相互直接引用，其交互通过元连接件完成。

元层和基层因果相联，对元层 RSAS 的在线调整可实现对基层的修改进而实现系统的动态演化。RSAS 提供给用户不同抽象级别的体系结构视图和管理手段，不同层次之间存在的求精关系保证了应用的完整性和演化的一致性。SASM 也支持所有级别的体系结构动态性。

第 10 章　SASM 支撑平台和工具

本章主要介绍我们目前正在设计的原型系统——SASM 支撑平台，主要包括软件的开发、部署、验证、演化及其基本的支撑环境。为了从宏观层面上对整个系统有更好的理解，在接下来的小节中我们将首先介绍该支撑平台的总体架构设计，根据软件设计的步骤以及整个软件的生命周期将整个框架分成以下三大块：构造工具集、执行工具集和演化工具集。然后分析软件运行和监控阶段的一些关键技术。在 10.4 节将着重对体系结构层次动态演化管理进行分析，从而为后面的设计提供支撑原理与机制。10.5 节分析如何根据整体框架、动态演化的基本原理及一些相关的机制来实现我们的原型系统。

10.1　引　　言

在软件工程领域中，软件体系结构概念自提出以来已经发生了很大的变化，学术界首先提出软件体系结构概念是为了弥补从软件需求到软件设计之间的鸿沟，从而便于人们去理解整个系统的框架与结构。然而，经过近几年的发展，软件体系结构已经不仅仅停留在软件的分析阶段，而是逐步贯穿于整个软件的生命周期，从需求分析到软件维护都以软件体系结构描述作为蓝本。本书主要讨论软件的动态演化技术，因此我们主要关注运行期间的软件体系结构，从而更好地为软件的动态演化提供可靠的支持。因此，后面几节所提出的 SASM 原型系统的设计都以体系结构为核心来驱动软件的动态演化，并为开发人员提供一个可视化的操作环境。

纵观整个软件开发领域的发展历史，其整体趋势就是尽量使得设计方法和开发工具越来越符合人的习惯。自第一台存储程序计算机设计以来，程序最初是采用二进制指令建立的，所有的程序都是通过手工翻译成机器能够识别的二进制码。在 20 世纪 50 年代后期，开始了自动程序设计系统方面的研究，将所有翻译工作交由计算机来完成。在随后几十年的发展过程中，出现了高级程序语言，其形式越来越符合人的思维习惯，从而也降低了软件开发的门槛。然而仅仅采用一种计算机能够精确识别的程序语言来进行程序设计，显然不能满足人们的需要，特别是随着软件在企业中的逐渐普及，很多软件需要特定领域知识的人来参与开发。随着面向对象程序语言 C++的出现，关于应用程序的设计有了一个根本的转变，即从基于文本的程序设计到基于图形界面（GUI）的程序设计上来了，这样就很直观地将软件的使用界面展现在用户面前。而在最近几年的时间里，人们开始把目光放到模型语言上。目前，模型驱动的体系结构（MDA）研究正在如火如荼地进行，它使用模型直接生成程序代码和整个软件框架，从而使程序设计人员的目光由原来在代码上转移到整个软件的架构上来，它更倾向于对软件的整体设计，而不是具

体细节的实现，这样更有利于程序设计人员从系统的整体架构上作全局的把握。模型作为开发语言的升级，它更符合人类的思维习惯。SASM 支撑平台中的 SA 可视化构造器旨在为不同的用户提供不同的软件开发视角：软件架构师可以直接使用它来搭建软件整体框架；程序员依据其整体框架可以进行模型的细化，包括属性设置以及具体功能的细化；而一些拥有领域知识的专家则可以在该框架上进行业务流程设计。整个实体以模型存在，便于不同角色之间的理解与交流。

软件进入运行期间后，还需要进行软件维护与升级。许多安全性要求较高或者执行关键任务的软件需要在运行期间进行动态升级，如银行系统、电力系统、卫星软件等。如果对这些软件进行停机升级将带来难以估量的损耗。软件动态演化技术正是为迎合这些领域的需要而出现的。为了支持软件的动态升级，目前还有许多工作要做，除了一些演化理论之外，还需要实际平台的支持。

目前这类平台主要有卡耐基－梅隆大学计算机科学学院（School of Computer Science，Carnegie Mellon University）、ABLE（Architecture Based Languages and Environments）项目组开发的 Acme 工具。Acme 所提供的功能主要有两个：1）提供现有的几种 ADL 之间的转换（如 Rapide、Wright、Unicon 和 Aesop 等）；2）进行体系结构设计和分析。目前该工具的最新版本为 3.0.0 版，该工具以及相应的说明文档都可以直接从 ABLE 项目组的网站（http://www.cs.cmu.edu/~acme/）上获得。另外还有加州大学（UCI，the University of California，Irvine）欧文分校的软件研究院开发的基于软件体系结构的开发环境 ArchStudio4。ArchStudio4 强调从软件体系结构角度进行软件开发，而不注重软件实现细节的代码编写，它支持基于 C2 体系结构风格的软件开发。该工具可以直接从他们的网站（http://www.isr.uci.edu/projects/archstudio/）下载。

10.2 支撑平台的总体架构设计

为了更好地支持软件体系结构建模及动态演化，我们所设计的 SASM 支撑平台主要由一系列执行相关任务的工具集组成。所有工具集通过相互协作来进行软件的动态演化管理。其总体可以分为三个部分：构造工具集、执行工具集和动态演化工具集。其中构造工具集旨在方便用户按照 SASM 模型可视化地进行应用系统的开发；执行工具集为系统提供运行和监控环境，包括物理构件的调度和执行、构件交互事件的截取和管理、元连接件规约的解析执行等。动态演化工具集支持基于体系结构的系统动态演化，包括演化分析、评价、解决从当前态切换到目标态过程中运行状态的迁移问题，保证演化期间的上下文一致性，进而保障软件演化完整进行。其总体框架如图 10-1 所示。下面将对这些工具集的各个功能模块作详细的说明。

1. 构造工具集

主要包括：SA 可视化构造器，D-ADL 语言编辑器、D-ADL 语言语法解析器、求精检测器、物理构件发现器、物理构件绑定器、模型验证器。

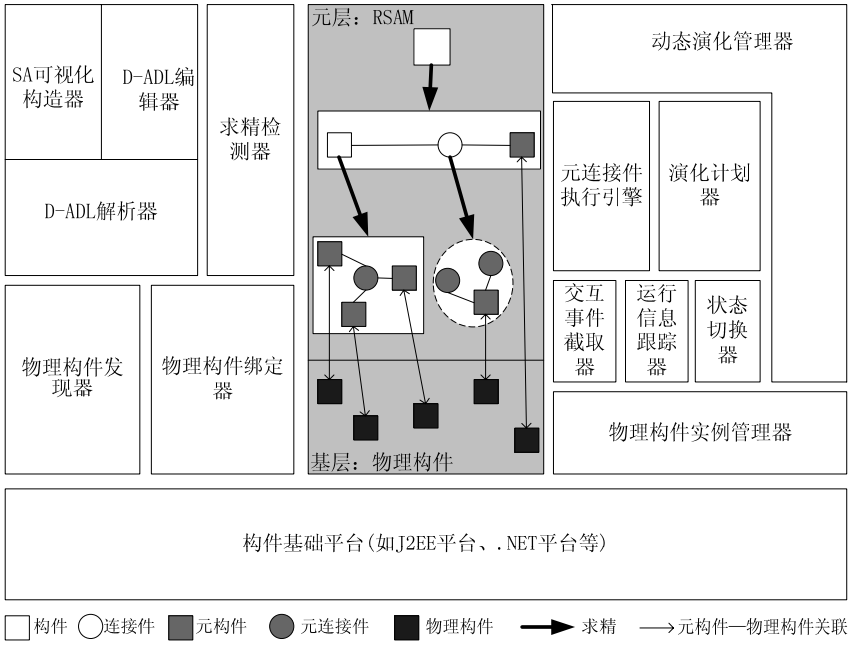


图 10-1　SASM 支撑平台的总体架构

（1）SA 可视化构造器。该构造器提供一个体系结构构造环境，它可以根据用户的需求生成体系结构模型。该框架中基本模型元素主要有连接件、构件、端口以及通道。构造器利用这些基本的元素进行体系结构建模，建模过程中可以产生不同层次的体系结构视图，其中底层的体系结构模型保持了上层体系结构的行为和约束，低层次的体系结构是高层体系结构的逐步细化。在运行过程中，我们可以从不同的视角来观察不同层次体系结构的行为和属性。底层体系结构的外部观察行为和顶层保持一致，其属性也满足顶层体系结构的属性约束。因此，为了方便不同类型的用户能从不同层次的体系结构视图进行系统的观察与实施，需要引进复合构件和复合连接件元素。在体系结构的拓扑结构中，构件与构件之间必须通过连接件进行连接，同时应该满足接口的连接规则，用于保证系统正确地构建与演化。该构造器同时还提供一个可视化界面用于设置构件、连接件以及端口的属性。

（2）D-ADL 语言编辑器。该编辑器主要提供一个文本编辑界面用于完成对 D-ADL 语言的浏览与编辑，用户除了可以通过 SA 可视化构造器进行体系结构建模之外，还能通过所生成的 D-ADL 语言进行编辑来完成对体系结构模型的修改。

（3）D-ADL 语言语法解析器。该语法解析器主要对采用 D-ADL 语言编写的体系结构描述进行解析。D-ADL 语言语法解析器和其他编程语言解析器类似，它也有自己的关键字、语法和语义。当用户在语言编辑器中编写的体系结构描述不符合 D-ADL 语法时，该解析器可以正确地定位语法错误，并提出修正意见。

（4）求精检测器。软件体系结构在设计初期并不能一次将所有细节表示出来，因此需要进行体系结构求精操作。体系结构求精的本质在于：不同层次的体系结构信息应该保持一致。体系结构求精主要分为结构求精、行为求精和属性求精。构件的行为求精应该满足进程观察弱模拟关系，连接件的行为求精应该满足进程分支弱模拟关系。而求精检测器则依据这些准

则来验证求精后体系结构是否依旧保持了抽象体系结构的行为、结构及其他方面属性。

（5）模型验证器。在进行体系结构建模期间同时还生成了 D-ADL 语言。在 D-ADL 中，构件、连接件和体系结构风格被模型化为高阶多型π演算中的抽象（abstraction）类型，系统行为被模型化为进程（process），构件和连接件的交互点则被模型化为通道（channel）。因此将生成的 D-ADL 转换为高阶多型π演算中所对应的元素后可以进行模型检查、进程死锁、进程跟踪等一系列校验，从而保证体系结构规约与系统实现保持一致。

（6）物理构件发现器。软件架构师在设计软件体系结构时采用 D-ADL 形式化描述语言对体系结构中的构件接口进行了描述，接口描述通常包括了该构件对外所提供的服务以及需要外界为其所提供的服务。为了便于软件的动态演化，我们采用反射机制来进行体系结构重配置，该反射框架分为两个层次：元层和基层。其中元层主要包括元构件和元连接件，基层主要包括可运行的实体构件。元层的构件和基层的物理构件一一对应，从而完成元层到基层的映射。因此，当整个软件体系结构被部署到运行环境后，物理构件发现器根据构件的接口描述在分布式环境下查找满足该接口的实体构件，构件的发现通常还包括一系列的接口匹配算法，查找最合适的构件进行绑定。

（7）物理构件绑定器。物理构件绑定器主要在软件体系结构模型被部署到运行环境时进行物理构件的绑定，将基层物理构件绑定到元层构件上，进而完成体系结构的部署工作，将软件投入运行。

根据绑定时机的不同，物理构件绑定可以分为静态绑定和动态绑定。静态绑定是指在软件体系结构部署期间直接由用户手动绑定物理构件。动态绑定是指运行环境在软件运行期间根据用户的配置信息重新发现满足条件的物理构件并进行绑定。该平台提供一个可视化的绑定窗口来供用户绑定物理构件。

2. 执行工具集

主要为系统提供运行和监控环境，包含构件基础平台、物理构件管理器、运行信息跟踪器、交互事件截取器和元连接件执行引擎。

（1）构件基础平台。为了保证构件能按照用户的期望有序执行，运行环境必须为构件提供一些必要的基础设施。目前主流的构件执行平台主要有 J2EE 和.Net 平台，我们的原型系统以 Weblogic 应用服务器为基础进行扩展，使得它能提供一个构件运行环境，并能很好地对构件进行管理。

（2）物理构件管理器。物理构件管理器主要利用构件基础平台的接口实现对物理构件的生命周期管理，包括物理构件实例的创建、调度、执行、挂起、激活、销毁等活动。

（3）运行信息跟踪器。为了能实时收集软件运行期间的一些状态参数，平台专门提供一个构件运行信息跟踪器，用于跟踪系统的运行状态，并实时将状态值结果返回到元层的构件属性上。

（4）交互事件截取器。软件系统中构件交互事件存在部分序列现象，即部分事件之间具有时序和因果关系，如事务处理中事务提交必发生在开始事务事件之后。交互事件截取器旨在捕获构件之间的交互事件，并把物理构件之间的交互导向元连接件引擎，使得用户能对特定事件进行显性监控，进而在应用层次上具有联机分析分布式系统行为语义的能力。

（5）元连接件执行引擎。元连接件执行引擎实际是软件体系结构描述中连接件语义的

解释器。不同的元连接件执行引擎通过调用不同的中间件平台提供的通信 API 分别解释执行该类别的协议和连接件其他语义，包含信息交流、协调和数据转换等功能，同时依据 RSAS 描述语义，定位、调用其他物理构件。例如 IIOP 元连接件引擎借助 CORBA 平台、RMI 元连接件引擎建立在 J2EE 平台基础上。所有元连接件执行引擎注册到连接件执行管理器上，被连元接件执行管理器管理和调度，类似操作系统的设备驱动管理。当用户对元连接件进行变更后，元接件执行管理器将调用相应的元连接件执行引擎来解析元连接件拓扑语义信息，进而完成构件之间的信息交互。

3. 动态演化工具集

主要提供控制变更过程的手段，监视和控制动态演化的全过程，保证演化完整运行。它主要包括演化计划器、状态切换器和动态演化管理器。

（1）演化计划器。演化计划器预先推导变更的结果及其影响范围。在发生软件变更之前，演化计划器预测和评估变更行为，决定是否进行软件演化，并制定运行系统从当前态切换到目标态的方案。

（2）状态切换器。状态切换器主要负责提供对变更前后状态进行切换的机制，以维持运行期间的上下文一致性。

（3）动态演化管理器。动态演化管理器主要负责调用其他工具进行体系结构配置，管理和监控所有的动态演化活动。

10.3　运行和监控的关键技术

软件动态演化是指软件在运行期间进行软件的升级，与传统的静态演化不同，由于软件在整个升级过程中始终处于运行阶段，因此，如何处理运行期间的状态迁移问题？如何保证软件朝着用户设想的方向演化？如何保证演化的完整性？这些在软件的动态演化过程中都是值得关注的问题。

本节将主要提出解决以上问题的两个关键技术——运行信息跟踪器的机理分析以及元连接件引擎的设计。

10.3.1　运行信息跟踪器的机理分析

物理构件的运行信息除了线程数、缓存大小等外，最重要的就是构件状态了。因为在动态演化期间，物理构件替换时的状态迁移是维持上下文一致性的关键问题。所以对物理构件状态的跟踪和监控是运行信息跟踪器的一个重要任务。

在 SASM 模型中，物理构件是遵循某种工业标准（如 EJB、COM 或 CORBA）的二进制构件，运行在诸如 J2EE 服务器这样的基础平台中，因此，对运行信息跟踪器来说，物理构件是一个黑盒子，其内部计算并不可见。然而，物理构件和外界发生的交互事件，无论是请求事件还是响应事件却都可以被交互事件截取器截获从而被运行信息跟踪器所知悉。如果把物理构件的状态理解为定义在两个相继发生的交互事件之间的时间段，仅仅因为交

互事件才能触发状态的转换，那么这种状态对于运行信息跟踪器来说是可观察的和可跟踪的。这与一般所说的构件状态有所不同，后者指的是构件内部变量值确定的状态。从粒度上来看，前者比后者要大，而后者由内部计算触发状态转换，难以被观察到。所幸在软件动态演化期间，对变更前后状态进行迁移时，仅需前者这种大粒度的状态定义就够了。因为软件动态演化时只需维持上下文的大体一致性。

当物理构件运行时，运行信息跟踪器收集它的运行信息，如线程数、缓存大小等，并把这些信息实时地存入其对应元构件规约的动态属性值中，以维持元层—基层的反射关系。借助交互事件截取器的支持，运行信息跟踪器也跟踪、监控物理构件之间的交互事件，并周期性地按时间段（时间段长短可自定义）把交互事件轨迹存储下来。每一个交互事件都包含如下内容：时刻、事件名、事件参数、事件类型（send 或 receive）、物理构件名。

运行信息跟踪器也对这些信息进行统计和分析，以判断物理构件的运行是否违反了元层 RSAS 规约中的行为约束，并向用户提出警报和建议。例如，某构件规约中定义了如下行为约束：

```
every array {
true* . via Open send { 1 } .
(not via Close send { 1 })* . via Open send { 2 }
} leads to state {false}
```

表示在两个进程间的独占式资源共享规则：在 1 个或多个任意活动后（true*），如果打开了资源 1(via Open send { 1 })，则在没有经历 1 个或多个关闭资源 1 的动作后((not via Close send { 1 })*)，不允许再打开资源 2 (via Open send { 2 })。如果这个构件作为一个元构件，其绑定的物理构件运行时的交互事件轨迹如下：

```
(…,<09/08/058:31:19,open,<1>,send,a3>,<09/08/058:31:23,open,<2>,send,a3>,…)
```

则运行信息跟踪器可判定这个物理构件的运行违反了元层 RSAS 规约中的行为约束，用户可根据运行信息跟踪器的警报采取对策，例如绑定一个新的物理构件到这个元构件上。

在软件系统中构件交互事件存在部分序列现象，即部分事件之间具有时序和因果关系，如事务处理中事务提交必发生在开始事务事件之后。通过行为约束表示，我们的 D-ADL 语言也可以描述交互事件的部分序列现象。由上可见，SASM 支撑平台在应用运行时捕获所有的交互事件并可提供给用户监控的手段，所以在应用层次上具有联机分析分布式系统行为语义的能力。

10.3.2 元连接件引擎的设计

在 SASM 模型中，物理构件间的交互通过对元连接件规约的解析执行完成。一旦交互事件截取器拦截到物理构件之间的交互事件，就把控制权交给元连接件引擎。元连接件引擎分析物理构件传输过来的数据，解释执行元连接件规约所定义的构件间通信协议、信息交流和数据转换等功能，并定位、调用其他物理构件。因此，元连接件引擎实际是元连接件规约的解释器和执行器，其设计如图 10-2。

开放、动态的运行环境意味着应用系统可能横跨异构的多种中间件平台，组成系统的

各个构件运行在不同的中间件平台上，遵循不同的构件标准，使用不同的构件通信协议。如何屏蔽各种构件的异构性和通信方式的差异性，成为一个需要解决的问题。理论上，SASM 支撑平台可建立在异构的多个中间件平台和专有通信中间件上，包括已有的诸如 CORBA—DCOM 桥接器，元连接件引擎进行通信协议和数据格式的转换，从而可屏蔽各种构件的异构性，保证参数传递的一致性和正确性。

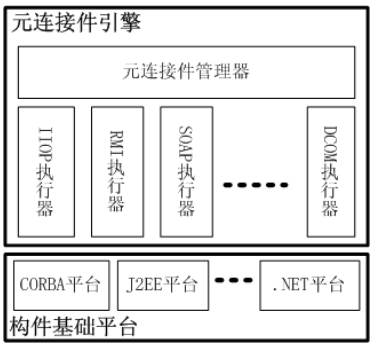


图 10-2　元连接件引擎的设计

Nikunj R. Mehta 等人提出了连接件分类框架，认为连接件执行构件之间的控制和数据的传输和转换，依据连接件服务的不同分为四大类：信息交流（Communication）、协调（Coordination）、转换（Conversion）和促进（Facilitation），根据连接件运行机制分为不同类别：过程调用、事件、流和数据存取等。考虑到分布式环境中构件之间主要是远程过程调用形式的交互，应用系统可能横跨异构的多种中间件平台，元连接件主要依据过程调用协议分类：IIOP、SOAP、DCOM、RMI 等。当然这并不意味不能有其他类型如事件机制的连接件执行引擎，元连接件引擎被设计为一个开放平台，用户可以遵循一定规范设计其他的连接件执行器。

所有连接件执行器注册到元连接件管理器上，被元连接件管理器管理和调度，类似操作系统的设备驱动管理。目前我们已实现了基于 J2EE 平台的 RMI 连接件执行引擎和基于 Web Services 的 SOAP 连接件执行引擎，中间件平台为 BEA 的 Weblogic 8.0。

10.4　动态演化管理

如何保证软件动态演化的完整性、一致性及其可追溯性是软件进行动态演化必须解决的问题，动态演化管理主要为软件的动态演化提供一个可操作的平台，使得用户能及时掌握软件演化过程中的异常信息，并采取相应的策略进行调整，保证软件能朝向用户指定的方向进行演化。

本节主要说明 SASM 支撑平台对应用系统进行动态演化的管理，重点在于 SASM 支撑平台提供的工具如何支持动态演化过程，以及动态演化过程中的状态切换机制。

10.4.1　动态演化过程中的平台支持

在 SASM 支撑平台中，动态演化管理器调用其他工具，管理和监控所有的动态演化活动。在其支持下，经典的由 RSAS 变更引起的动态演化过程如图 10-3 所示，分为三个大的阶段：变更 RSAS 的备份、演化分析和规划、系统切换。

演化从变更应用的 SASM 模型元层 RSAS 的备份开始。由于 RSAS 作为一个在运行时刻有状态、可访问的对象，在 RSAS 上所施行的更改操作通过 SASM 的反射机制最终影响到基层的变更，因此将导致系统的演化。这里着重阐述平台在其中所起的作用。对于预设的变更，动态演化管理器依据 RSAS 规约中预设的演化行为确定 RSAS 中的变更结果。对

于非预设的变更，动态演化管理器调用 SA 可视化构造器或 D-ADL 编辑器供用户对 RSAS 规约进行修改。须强调注意的是，在整个演化过程中，对 RSAS 备份的这种变更直到系统切换阶段才真正投入到系统的元层，在此之前，原来的 RSAS 模型一直起着作用。

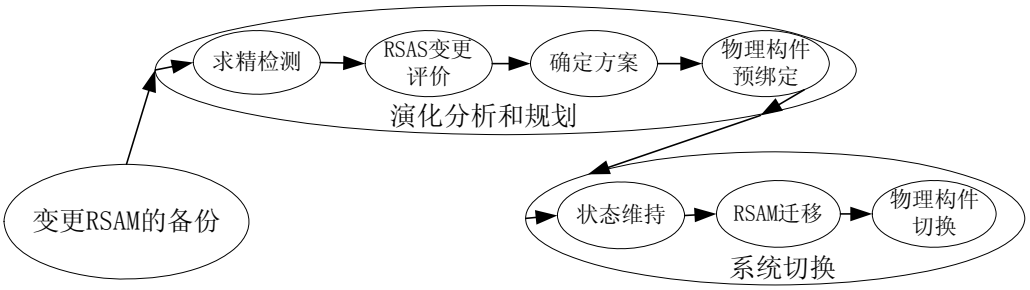


图 10-3　平台支持下的动态演化过程

变更 RSAS 备份后，动态演化管理器调用演化计划器对这种变更进行推导、评价，并制定运行系统从当前态切换到目标态的方案。由于 RSAS 是一个由不同层次体系结构规约形成的体系结构空间，即使 RSAS 发生变更后，层次之间还应该存在着求精关系，不同层次体系结构信息应该保持一致。因此，演化计划器首先调用求精检测器对已变更的备份 RSAS 进行层次求精关系检测。若检测没通过，则取消备份 RSAS 已发生的变更，终止这次演化过程。

若检测通过，则演化计划器对备份 RSAS 变更进行评价。考虑到元构件是 RSAS 中最具体的单元计算规约和物理构件的直接映象，元连接件是物理构件交互的线索和规约，评价使用简单而有效的策略：首先统计变更涉及元构件和元连接件的数量，包括增加了多少新的元构件和元连接件，删除了多少元构件和元连接件。然后计算涉及的元构件和元连接件数量占总元构件和元连接件数量的百分比。如果这种百分比大于已定义的阈值（阈值可由用户根据应用的特点自定义，经验值表明，一般低于 30%），则意味变更大太，超出了运行系统演化的承载力，可能导致运行系统的不稳定和崩溃。此时，取消备份 RSAS 已发生的变更，终止演化过程。

如果备份 RSAS 的变更是可接受的，演化计划器制定运行系统从当前态切换到目标态的方案。这是一个用户参与的交互式过程。演化计划器从物理构件库中发现符合新增或被需要更改绑定的元构件规约的候选物理构件，推荐给用户，用户从这些候选物理构件中进行最终的选择和确定。在用户的指示下，演化计划器也确定运行系统从当前态切换到目标态的策略，例如在什么时候进行切换，切换时是否需要处理状态的迁移，被解绑的物理构件实例是销毁还是放入实例池中以待重用等。切换方案制订后，演化计划器就依据切换方案调用物理构件管理器创建新的物理构件实例，并使新增的元构件或需要更改绑定的元构件预绑定新的物理构件实例，把两者关联起来，以预备系统的切换。

在运行系统从当前态切换到目标态阶段，动态演化管理器首先调用状态切换器进行状态维持活动（如果需要），尽量使运行上下文一致，具体情形见下节。然后动态演化管理器进行 RSAS 迁移活动，元层原来的 RSAS 用更改了的备份 RSAS 所取代。依赖于物理构件管理器，动态演化管理器最后把系统切换到新的物理构件实例的运行上（对于新增或被替换的物理构件），把解绑的物理构件实例销毁或放入实例池中。

10.4.2　运行状态维持机制

从当前态切换到目标态过程中运行状态的维持是一个重要的问题。一般来说，当软件执行到某一断点时，从系统配置上撤换下来，它保持一定的状态信息。当该构件重新连接进入系统时需要从撤换时的断点继续开始执行。

动态演化中对运行系统状态的处理目前有三种方式：（1）不考虑状态问题，直接进行系统的切换。这种方式的优点是快捷简单，缺陷是可能引起系统的部分失效或运行异常，可用来针对非关键系统或状态对外界无关紧要的情况。（2）状态的直接迁移，直接把系统旧版本的状态写入新版本中。这种方式不仅要求知悉旧版本和新版本的实现细节，而且要求运行时环境能操纵系统实现体的内部。这种机制除了极少数的如 NeoClasstalk 语言 （smalltalk 语言的一种扩展）环境支持外，不适用于大多数情况。（3）状态的间接迁移，系统旧版本输出状态，新版本读入接收该状态。这种方式存在两个前提或假定：一是状态要具有标准的表示格式和可共享的语义，以使新旧版本能够无障碍解析和理解；二是系统中参与的构件必须具有输出状态或输入状态的接口。这两个前提限制了直接迁移方式的应用。

在按照 SASM 模型构造的应用系统中，运行状态的维持主要是指替换物理构件间的状态迁移，因为，虽然系统规约由元层 RSAS 指定，但功能实现体和单元计算却落实在物理构件上。考虑到两方面的因素，上述状态迁移方式在这里并不适用。一方面，物理构件只是遵循工业标准（如 EJB、COM 或 CCM）的二进制构件，运行在诸如 J2EE 服务器这样的基础平台中，其内部计算并不可见，因此直接状态迁移方式不易实现。另一方面，间接迁移状态方式的两个前提在 SASM 模型中并没有强制规定，因此也不完全适用。

根据 10.3.1 节的分析得知，执行工具集跟踪、监控物理构件之间的交互事件，并周期性地把交互事件轨迹存储下来。也就是说，物理构件最近一段时间内的输入输出历史被记录下来。正是基于这一点，我们的状态维持方法的基本思想是：新物理构件重新执行一次依据对应旧物理构件的最近一段输入序列，以期望达到后者的状态。这和前面的方法都不同，倒有点类似数据库管理系统从日志中恢复数据的机制。这种方法要解决两个关键的问题：一是确定执行的第一个输入，也就是在输入历史序列确定起始点；一是为恢复状态而执行的过程中其输出如何处置。

第一个问题的解决依赖于物理构件对应的元构件规约。我们知道，元构件规约以 D-ADL 描述，D-ADL 以 π 演算的进程概念定义了物理构件的行为语义，而 π 演算进程可以表示成一个标签迁移系统。由于物理构件的实际运行过程必须遵循元构件规约，因此，物理构件的输入、输出事件都可以对应元构件规约的进程标签迁移系统中的输入、输出动作，而且其先后顺序符合迁移规则的约束。新物理构件被创建后从进程标签迁移系统的起点开始运行，因此，其第一个输入应该定位在从离进程标签迁移系统起点出发碰到的第一个输入动作所对应的输入事件。同时考虑到应尽量缩短恢复状态的时间，第一个输入动作所对应的输入事件可能有多个，我们选择最近的一个。

以示例说明，在动态服务器的 C-S 体系结构系统中处理这种情况：服务器发生错误死机或崩溃，系统需要自动重新启动一个新的服务器，并把客户端请求导向新的服务器以不致服务中断。服务器规约如下描述：

```
singlecomponent server () {
```

```
port { saccess : Tsaccess. }
computation  {
        via saccess^request receive requestdata.
        unobservable.
        via saccess^reply send replydate.
        replicate
             }
                  }
```

其标签迁移系统如下图所示：

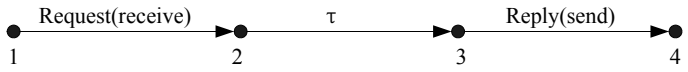


图 10-4　服务器规约的标签迁移系统

原来的物理服务器的交互事件历史轨迹假设如下：

```
(…,<09/08/069:30:01,request,<10>,receive>,<09/08/069:30:02,reply,<20>,send>,
<09/08/069:30:05,request,<100>,receive>,<09/08/069:30:06,reply,<200>,send>,
 <09/08/06 9:30:09,request,<1000>,receive>)
```

从交互事件历史轨迹可看出，原来的物理服务器在接收到数据 1000 时出现故障。这时启动新服务器 server2，server2 为保持 server1 的状态，应该重新执行第一个输入动作所对应的输入事件并且是最近的一个，即执行（<09/08/06 9:30:09,request,<1000>,receive>）事件。

关于第二个问题，即如何处置新物理构件为恢复状态而执行过程中的输出，其解决办法较为简单。对于那些在旧物理构件的交互事件历史轨迹中已存在对应的输出动作，交互事件截取器截获后直接抛弃掉它们。对于那些在旧物理构件的交互事件历史轨迹中还没有对应的输出动作，交互事件截取器截获后把它们导向元连接件引擎以依据元连接件规约发往目的物理构件。上例中新服务器执行（<09/08/06 9:30:09,request,<1000>,receive>）事件后的输出应该被交互事件截取器截获后导向元连接件引擎。如果原来的物理服务器的交互事件历史轨迹如下：

```
(…,<09/08/069:30:01,request,<10>,receive>,<09/08/069:30:02,reply,<20>,send>,
<09/08/069:30:05,request,<100>,receive>,<09/08/06 9:30:06,reply,<200>,send>,
<09/08/069:30:09,request,<1000>,receive>,<09/08/069:30:10,reply,<2000>,send>)
```

则新服务器还是从执行（<09/08/06 9:30:09,request,<1000>,receive>）事件开始，但紧接其后的这个输出则被交互事件截取器截获后舍弃。

10.5　支撑平台的一个原型实现

本节将主要从以下几方面介绍 SASM 支撑平台原型的实现：首先介绍该原型系统的开

发环境，然后介绍 SA 可视化构造器如何表示体系结构元素，接着介绍该原型系统的设计与实现，最后使用原型系统对物理构件之间的通信进行测试并给出测试结果。

10.5.1　原型系统的开发环境

目前比较流行的图形开发插件主要有两类，一类是 JGraph 组件，它本身自带构件和端口模型元素，便于扩展成动态体系结构建模工具。另一类是基于 Eclipse 平台的 GEF（Graphical Editing Framework）和 EMF（Eclipse Modeling Framework）插件，该框架采用 MVC 框架进行开发，其中 GEF 插件主要用于完成业务逻辑处理部分，EMF 主要用于完成模型部分。动态体系结构建模工具需要提供一个模型构建视图，一个动态体系结构语言编辑视图以及从元层到基层的绑定视图。JGraph 组件和 JEdit 组件的配合可以完成模型和语言的构件，但对于物理构件的绑定以及所有功能的整合还有较多的工作。而目前已有的几款体系结构支撑平台如 AcmeStudio 都是基于 Eclipse 进行开发的，由于 Eclipse 开发的应用程序可以扩展 Eclipse 自身框架，使得用户可以不必关心各个视图之间的关联细节，方便用户对整个框架进行扩展，从而节省了许多工作量。根据 SASM 支撑平台集成了多个工具集的特点，同时它所用到的执行工具集主要由 Weblogic 应用服务器扩展而来，而 Weblogic 在 Eclipse 开发环境中开发了一系列的应用接口，便于其他程序对其进行扩展。因此，我们选择在 Eclipse 环境下进行开发，构造器主要依据 GEF 插件扩展而来。

1．Eclipse 简介

Eclipse 是一个非常优秀的集成开发环境，这个在 IBM 支持下的开放源码项目经过一段时间的发展完善，已经为广大 Java 开发者所熟悉。Eclipse 的出现，为 Java 开发者提供了免费使用强大的 Java IDE 的机会，通过集成大量的插件（plugin），Eclipse 的功能可以不断扩展，以支持各种不同的应用。Eclipse 开发环境主要由 Eclipse 项目、Eclipse 工具项目和 Eclipse 技术项目三个项目组成，每一个项目由一个项目管理委员会监督，并由它的项目章程管理。每一个项目由其自身的子项目组成，并且使用 Common Public License（CPL）版本 1.0 许可协议。

Eclipse Platform 是一个开放的可扩展的 IDE。Eclipse Platform 提供建造块和构造并运行集成软件开发工具的基础。Eclipse Platform 允许工具建造者独立开发与他人工具无缝集成的工具，你无须分辨一个工具功能在哪里开始，而另一个工具功能在哪里结束。

Eclipse SDK（软件开发者包）是 3 个 Eclipse 项目的子项目（Platform、JDT、PDE）所生产的组件合并，它们可以一次下载。这些部分在一起提供了一个具有丰富特性的开发环境，允许开发者有效地建造可以无缝集成到 Eclipse Platform 中的工具。Eclipse SDK 由 Eclipse 项目生产的工具和来自其他开放源代码的第三方软件组合而成。Eclipse 项目生产的软件以 CPL 发布，第三方组件有各自自身的许可协议。

目前许多较流行的开源项目以及商业项目都在 Eclipse 环境下开发了应用插件，以便用户能更方便地将其功能进行集成和整合。因此作为研究性质的原型系统可以从这些开源项目中节省大量的时间。同时也能更好的将自己所需要的功能以插件的方式集成在 Eclipse 平台下，实现与其他项目的无缝集成，从而节省大量的资源，加快了系统的开发进度。

2. GEF 插件简介

GEF（Graphical Editor Framework）是一个图形化编辑框架，它允许开发人员以图形化的方式显示和编辑模型，从而更易于用户对编辑对象的理解和操作。目前存在很多由 GEF 构建的各种 Eclipse 的应用程序，其中包括状态图、活动图、类图，用于 AWT、Swing 和 SWT 的 GUI 构建器以及过程流编辑器。同 Eclipse 一样，它作为一种开放源代码的插件被众多开发人员所喜爱，通过对源代码的分析可以更好地扩展出满足自己需要的软件制品。

如果你需要以图形方式来显示和编辑模型，那么 GEF 是一个很好的选择。它提供了一个可在 Eclipse 工作台中任何地方使用的查看器（类型为 EditPartViewer）。作为流行的图形编辑框架，它拥有很好的设计模式和方便的扩展接口。其中有许多对于图模型的操作功能都已经实现了，其中包括图形的显示和编辑，开发人员只需集成其相应的接口即可完成简单的模型操作。

GEF 采用了流行的 MVC 框架进行开发，其中控制器作为视图和模型之间的桥梁（请参阅图 10-5）。每个控制器（即所谓的 EditPart）负责将模型映射到它的视图，也负责对模型进行更改。EditPart 还观察模型并更新视图，以反映模型状态中的变化。EditPart 是一种对象，用户将与这种对象进行交互。该框架的协作图如 10-5 所示。

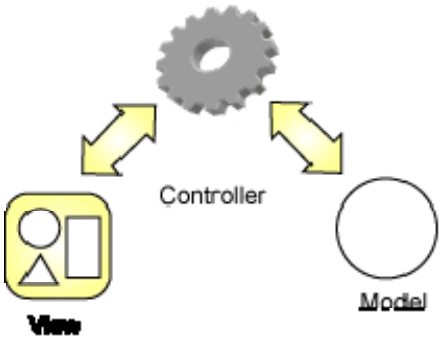


图 10-5 模型-视图-控制器

GEF 同时还提供了两种查看器类型：图形的和基于树的。每种查看器都主管一种不同类型的视图。图形查看器使用了在 SWT 画布（Canvas）上绘制的图形（figure）。图形是在 Draw2D 插件中定义的，该插件是 GEF 的一部分。TreeViewer 将 SWT 树和 TreeItem 用于其视图。

10.5.2 体系结构元素的表示

在 D-ADL 中，体系结构元素主要包括构件、连接件、端口、通道。为了给不同用户提供不同的视图，我们还引入了复合连接件和复合构件两种元素，各个元素的表示如图 10-6 所示。

构件（component）作为体系结构的基本要素之一，依据外部端口和内部行为描述。用户可以根据自己的需要定制属性名和类型，除此之外，还包括了规则、结构、描述等信息。在 SA 可视化构造器中，构件模型采用蓝色圆角矩形表示。

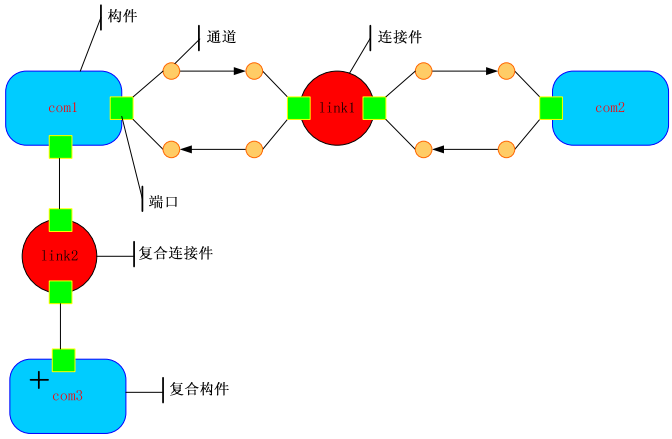


图 10-6　SASM 中体系结构元素表示

连接件作为一种特殊的构件，它依据外部端口和内部路由行为（routing）描述。连接件主要用于建立构件间的交互以及支配这些交互规则，所有构件都通过连接件相连。为了与构件的表示区分开来，我们采用红色圆形来表示连接件模型。

端口表示了构件和外部环境的交互场所，定义了构件能够提出的计算委托。每个端口可以被施加诸如通信协议等约束而形成端口类型。构件接口由端口组成，因此，不同类型的端口代表构件所提供的不同服务。端口不能单独存在，它只能作为构件或者连接件对外交互的一部分，附属在这两种元素之上。在 SA 可视化构造器中，端口模型采用绿色正方形表示。

端口通常由一组通道构成，通道是体系结构中基本的通信接口，它的体系结构角色就是在端口之间提供通信通道，为构件与构件之间提供可靠的信息通信。通道也不能单独存在，每一个通道都必须属于某一特定的端口，在 SA 可视化构造器中，通道模型采用橘红色圆形表示。

复合构件的内部结构是由构件和连接件组成，与复合构件端口相连的内部元素必须是构件端口，我们称之为端口映射。复合既可作为一种用于组合更高一层复合构件或连接件的元素，也可作为一个单独运行的系统。很多时候我们直接把系统看作为一个独立运行的复合构件。复合构件除了拥有端口和内部描述外，还可能有动态演化计划定义（choreographer）。复合连接件与复合构件类似，不同的是与复合连接件端口相连的内部元素必须是连接件端口。在体系结构模型中，复合构件模型采用带“+”的圆角矩形表示。复合连接件模型采用带“+”的圆形表示。

10.5.3　原型系统的设计与实现

原型系统的设计主要分 3 大块进行，其中包括构造工具集、执行工具集以及演化工具集。这三大块所包含的各个分模块在 10.2 节已经做了详细的介绍。这一部分重点从两个方面进行介绍：SASMLib 体系结构和 SASM 设计结构。

1. SASMLib 体系结构

SASMLib 属于 SASM 中的核心部分，是一个可重用的类库，用于表示和操作 SASM 的

设计，目前我们只实现了 Java SASMLib。它为用 JAVA 语言编写体系结构工具提供一个面向对象的框架。它可以读、写、操作用 D-ADL 定义的软件体系结构设计。SASMLib 应用程序体系结构如下图所示。

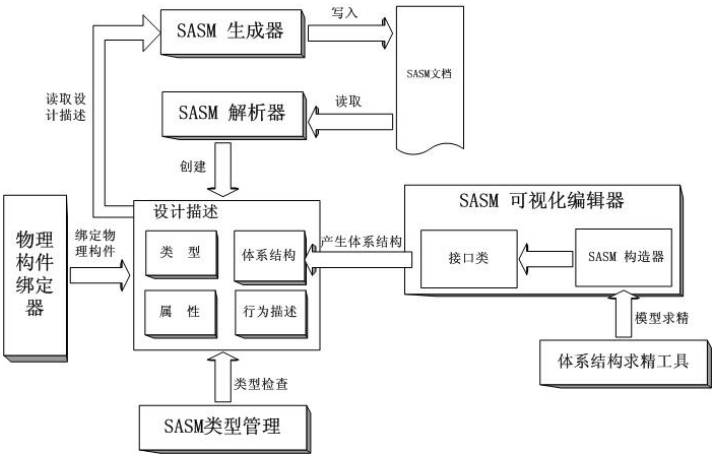


图 10-7 SASMLib 应用体系结构

2. SASM 设计结构

构造工具集主要涉及可视化建模工具的设计，以及 D-ADL 语言编辑器的设计。同一个体系结构通常包括两个不同的表示方式：对象模型和文本化的 D-ADL 语言。系统在开发人员建立对象模型的同时也生成了对应的 D-ADL 语言。在建立体系结构模型的时候需要遵循该平台所定义的规约，如端口作为构件和连接件的一部分只能放置在其边上；构件与构件之间的端口不能直接相连，必须通过连接件来完成通讯；通道只能与通道进行连接，且根据通道的类型来确定通道之间连线的方向（通道有 3 种类型：in, out, inout）。

SA 可视化构造器的整体设计结构如图 10-8 所示：

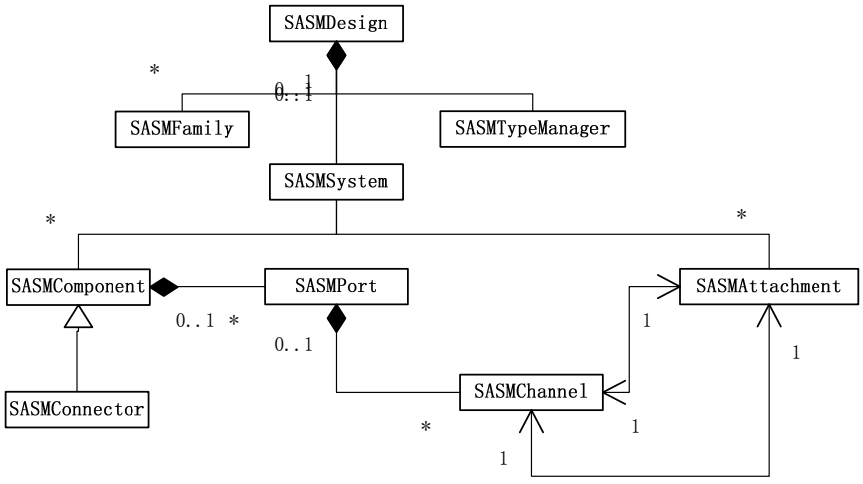


图 10-8 SASM 设计结构

构造器的设计与 ACME 雷同，一个 SASMDesign 包括一个全局体系结构元素类型集合以及属性类型定义（SASMElementType 和 SASMPropertyType 类实例），一个 SASMTypeManager 实例，多个 SASMFamily 实例，多个 SASMComponent 实例以及多个 SASMAttachment 实例。在 D-ADL 语言中，连接件被看作一种特殊的构件，它拥有端口及通道。所不同的是构件作为计算单元具有计算行为，而连接件作为构件与构件之间的通信桥梁，主要支配构件之间的交互规则，其路由行为可以看作是计算行为的变种。因此，在进行对象结构设计时我们将 SASMConnector 看作 SASMComponent 的一个子类。一个 SASMComponent 对象包含了 SASMPort 类的实例的集合，该集合作为构件的接口描述了该构件对外所提供的服务以及需要外界为其所提供的服务。通道是构件与构件之间通信的基本交互点，且每一个通道只能与一个通道相连，因此 SASMChannel 与 SASMAttachment 之间为二元连接关系。

一个 SASM 系统通常是由构件、连接件、端口和通道实例所构成的集合，它们分别对应体系结构描述语言中的构件、连接件、端口和通道。为了给不同的用户提供不同的视图，SASM 系统引入了体系结构空间概念，从高层的体系结构模型到低层的体系结构模型，是逐步细化求精的过程，这意味着高层次体系结构中某一构件（连接件）可能被求精为低层次中具有内部结构的某一复合构件（连接件），但后者的外部观察行为（路由行为）应该和前者保持一致，后者的属性信息应该满足前者的属性约束。

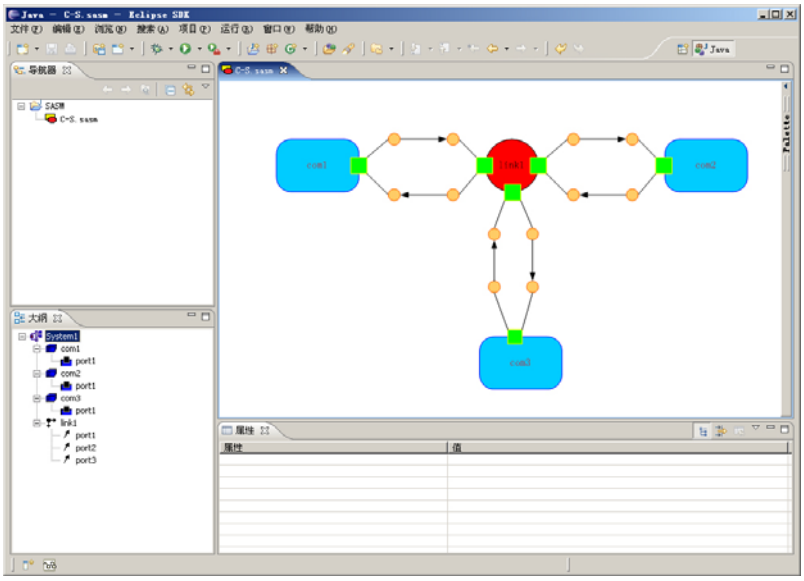


图 10-9　构造工具的 SA 可视化构造器

执行工具集作为系统的运行环境，主要为构件运行起调度作用，同时可以跟踪构件的运行，截获构件交互事件等基本操作。它的原型系统是在 BEA 的 Weblogic 8.0 中间件平台上实现的，并以插件的方式运行在 Eclipse 平台下。执行工具集中一些基本的功能如元连接件引擎、交互事件截取器、运行信息跟踪器、物理构件管理器等都是由一系列协作的 EJB 组成。运行的构件信息都直接反映在元层体系结构上，并利用 SA 可视化构造器进行跟踪管理。元连接件引擎暂时只支持基于 J2EE 平台的 RMI 通信和基于 Web Services 的 SOAP 通信。

演化工具集和执行工具集一样，也是实现在基于 Weblogic 8.0 中间件平台上。同时采用一系列协作的 EJB 来完成系统的动态演化管理以及状态迁移等功能。根据反射框架，可以实现元层构件到基层物理构件的映射。在元层所进行的体系结构调整将直接影响基层的体系结构，如构件的添加、删除和替换等操作都直接影响基层的物理构件；而基层的体系结构也将直接反映到元层的体系结构模型上，这样就很方便对运行的系统实施动态跟踪和演化。

10.5.4 对原型环境中物理构件间的通信测试

与直接通过 Weblogic 8.0 进行的构件引用交互方式相比，通过元连接件引擎进行构件交互的额外开销主要耗费在元连接件执行管理器的派发操作和连接件执行器对元连接件规约语义的解析执行上。对原型系统作了一些基本的性能测试，结果表明其带来的额外开销被控制在一个合理的可以接受的范围之内。

在实验中我们使用如下环境：6 台配置 800MHz Intel 奔腾处理器、256MB 内存的 Windows2000 PC 工作站，一台配置 2.1G Intel 奔腾处理器、1GB 内存的 Windows2000 server 服务器，整个环境为 100M 以太网。工作站安装 J2EE 客户端用户软件和原型系统交互点运行时 Java 支持类，服务器安装 Weblogic 和原型系统的执行工具集 EJB 以及用户 EJB。在实验中，J2EE 客户软件向服务器里的用户 EJB 发送消息，后者收到消息后立即将这个消息送回前者。我们对 SASM 原型系统与直接使用 RMI 通过 Weblogic 在处理消息传递时引起的额外开销进行对比，并测量不同大小的消息往返所需的时间。结果数据显示如图 10.10。从这个图中可以看到，原型系统相对 RMI 的额外开销平均大致为 55ms 左右。而且，尽管原型系统消息传递的总开销随着消息长度的增长而线性增长，原型系统本身导致的额外开销却增长十分缓慢，消息长度从 1 字节增长到 100000 字节，而这个额外开销仅从 40ms 增长到 75ms。

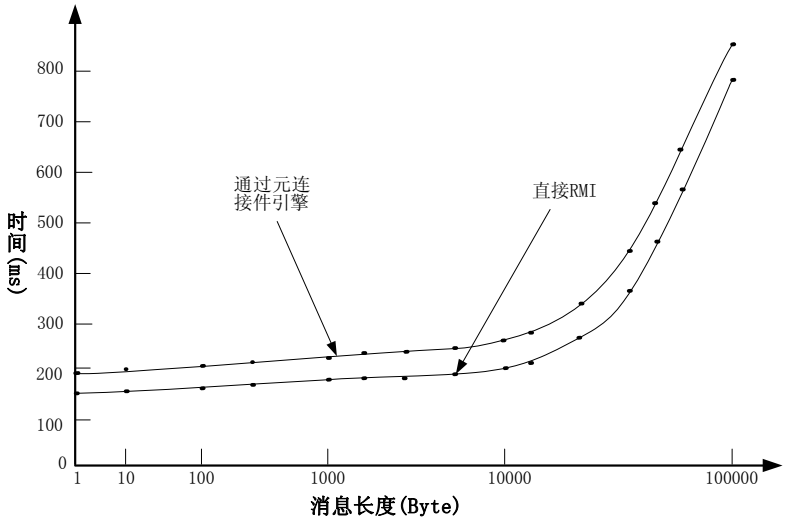


图 10-10 物理构件间的通信测试

基本性能测试表明 SASM 这种分布式应用模型在性能方面具有较好的可行性，与带来的优点比较，SASM 的额外开销在大多数的应用上是可以接受的。

过了不考虑任何用户模型情况下的随机推荐的情况。

此类 Web Agent 系统的体系结构如图 7-13 所示。

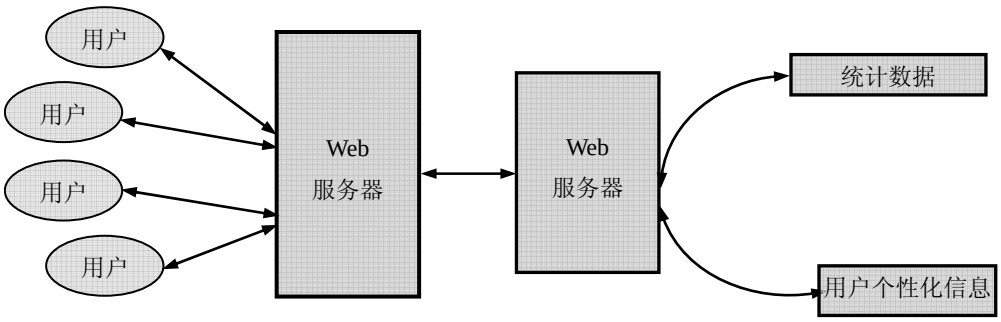


图 7-13 Web Agent 服务结构

这里的 Web Agent 就好像一个过滤器或者一个监控程序一样，从 Web 服务器上获取用户的访问信息，对其进行统计处理，经过算法的加工成为用户访问网页的一种个性化信息，Agent 再拿这些个性化信息反过来服务于用户，而用户在这里无形中起到一种训练作用，在自己访问网络的同时完成了对服务 Agent 的训练。大量的用户访问，使得 Web Agent 能够全面掌握访问网站用户的习惯信息，而且能够在一些新用户刚登入不久就可以提供出用户满意的推荐。故而这种 Agent 在网络中有着很好的应用前景，同时作为 Agent 系统而言很少是单个 Agent 独立完成任务的，对于助手类 Agent 由于其 Agent 本身的协作特性，也存在网络中多个 Web Agent 合作的情况，也就是说一个 Agent 存在于某个特定网站的服务器端或者某个客户端，都不能完全满足用户所有的网络访问需求，它们之间通常都要互通有无，通过协作来提供用户所需要的信息。从整个服务体系或者系统而言，其系统结构仍然是一种动态体系结构，如图 7-14 所示。其中 Web 服务器用虚线表示其为客户端 Agent 与服务端 Agent 之间的一种透明的交互媒介，而作为服务端 Agent 也存在为协作而进行的交互。

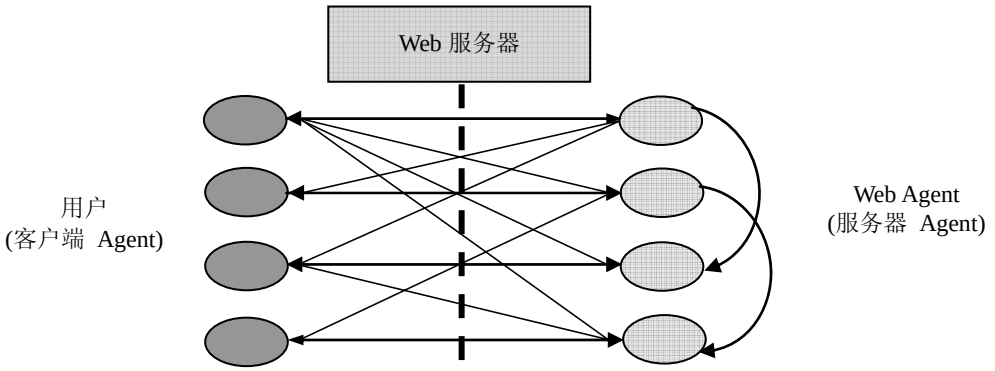


图 7-14 Web Agent 构成的多 Agent 系统结构

- Cogent/ARteM1S-M3C

Cogent 系统是南京大学软件新技术国家重点实验室研制的基于移动 Agent 主动构件框架系统，以提高在开放网络环境下对构件组装绑定的灵活性、代码可重用性，解决异构构件之间的互操作等问题，并增强系统对动态变化网络环境的适应性。

在 Cogent 系统框架中，构件的组装机制由组调用表（GroupTable）和定位表（Location Table）两部分，系统底层的移动 Agent 将根据组调用表和定位表所描述的元数据信息完成对相关服务构件的组装调用。

组调用表给出了移动 Agent 对构件调用的执行逻辑信息，Agent 根据组调用表中的功能元数据迁移至提供服务构件的目标节点完成请求并返回结果，其结构为一七元组（调用表名、参数列表、变量声明、私有数据区、调用表体、类型结构、调用条件）；其中调用表体项的结构又为四元组（指引名、结构类型、接口方法调用、后处理定义）。

定位表用于描述服务构件所在的位置信息，Agent 将根据该表迁移至提供服务构件的目标节点完成请求并返回结果，其结构为（定位表名、调用表名、定位表体）。定位表体项的结构为四元组（指引名、节点地址、构件名、构件实例名）。

Cogent 系统对结构反射的支持主要通过对组调用表和定位表的联合解释来实现。组调用表中的类型结构可支持四种构件组装方式：SEQ（顺序）， PAR（并行）、SLOOP（循环）和并行循环（PLOOP），因此通过对结构类型元数据的修改调整，可以实现基层构件运行结构的重配置。通过对接口方法调用的改变可以实现基层构件功能方法的重配置。通过对定位表中构件名、构件实例名等项的修改，可以实现基层构件实例的增加、删除和替换。

ARTeMIS-M3C 则是南京大学计算机软件研究所在 Cogent 系统基础上的进一步拓展。其用于 Internet 环境下软件通信协同中间件系统，以解决开放环境中实体之间协同通信模式的单一性和固定性问题，实现软件服务实体协同模式的多样性、协同方式的灵活性和协同行为的定制性。

ARTeMIS-M3C 系统采用基于 Interceptor 的机制，系统中的 PCM （ Programmable Coordination Media）构成了元层，用于表述基层服务软件实体的协同行为，其核心组成部分包括 Interceptor Manager PCMClinentProxy 表，PCMServerProxy 表以及 SCA （Software Coordination Agent）表。通过对 PCM 的编程配置，可提高对开放网络环境的动态适应性能力。

1 1 1 1
0..1 0..1 0..1 0..1

4. 面向通用服务的自适应中间件

具有典型代表意义的、面向中间件的通用服务层的自适应中间件项目包括：RAPIDWare，ACT 以及 RCSM。下面，我们将对这些典型项目分别加以具体介绍。

（1）RAPIDWare / ACT

① RAPIDWare

RAPIDWare 项目受到美国海军部支持（Grant No.N00014-01-1-0744 ），由美国密歇根州大学的软件工程和网络系统实验室设计开发，起止时间为 2001 年到 2006 年。

RAPIDWare 是基于构件开发的自适应可信中间件，致力于自适应软件的设计和开发，强调高可靠和自适应软件的设计。RAPID Ware 面向动态异构的环境，支持强实时高安全的应用，例如金融网络应用、电力网络、交通系统和命令控制环境，防止外部环境部件的

设计，目前我们只实现了 Java SASMLib。它为用 JAVA 语言编写体系结构工具提供一个面向对象的框架。它可以读、写、操作用 D-ADL 定义的软件体系结构设计。SASMLib 应用程序体系结构如下图所示。

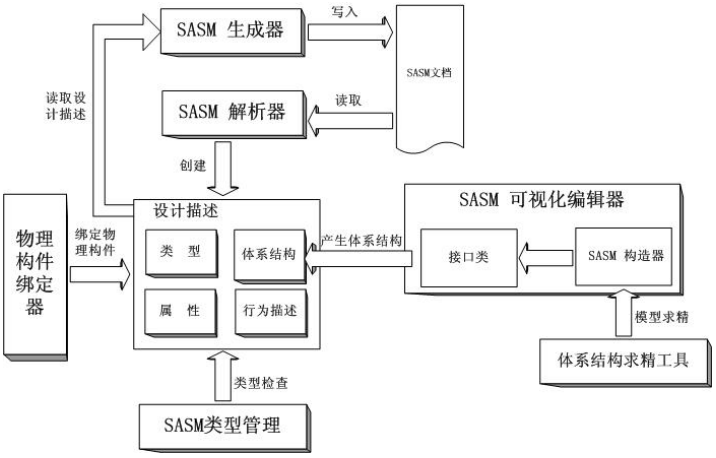


图 10-7 SASMLib 应用体系结构

2. SASM 设计结构

构造工具集主要涉及可视化建模工具的设计，以及 D-ADL 语言编辑器的设计。同一个体系结构通常包括两个不同的表示方式：对象模型和文本化的 D-ADL 语言。系统在开发人员建立对象模型的同时也生成了对应的 D-ADL 语言。在建立体系结构模型的时候需要遵循该平台所定义的规约，如端口作为构件和连接件的一部分只能放置在其边上；构件与构件之间的端口不能直接相连，必须通过连接件来完成通讯；通道只能与通道进行连接，且根据通道的类型来确定通道之间连线的方向（通道有 3 种类型：in, out, inout）。

SA 可视化构造器的整体设计结构如图 10-8 所示：

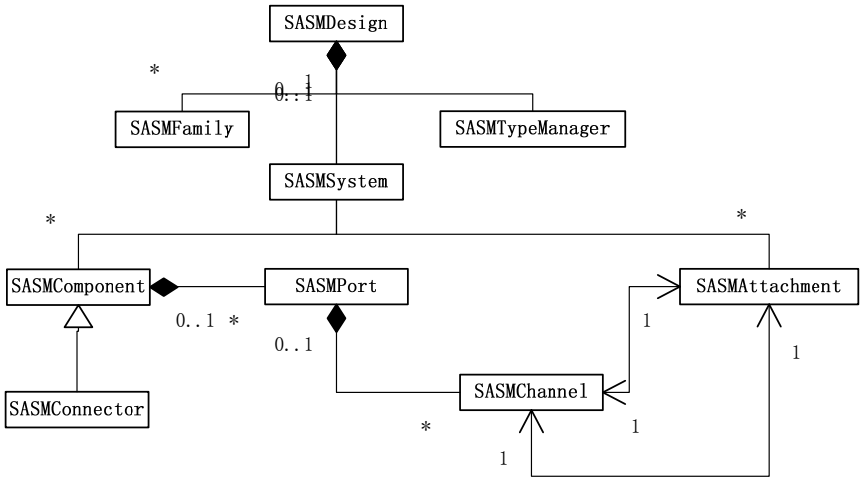


图 10-8 SASM 设计结构

参 考 文 献

［1］马晓星，余萍，陶先平，吕建．一种面向服务的动态协同架构及其支撑平台［J］．计算机学报，2005，28(4):467-477．

［2］贾向阳，应时，张韬，余晓峰．一个支持业务过程动态演化的可反射框架［J］．计算机工程，2006，32(10): 71-73，54．

［3］张世琨，王立福，常欣，杨芙清．基于层次消息总线的软件体系结构描述语言［J］．电子学报，2001，29(5): 581-584．

［4］李长云，李莹，吴健，吴朝晖．一个面向服务的支持动态演化的软件模型［J］．计算机学报，2006，29(7):1020-1028．

［5］黄罡，王千祥，梅宏，等．基于软件体系结构的反射式中间件研究［J］．软件学报，2003，14(11)：1819-1826．

［6］徐光佑，史元春，谢伟凯．普适计算［J］．计算机学报，2003，26(9):1042-1051．

［7］郑增威，吴朝晖．普适计算综述［J］．计算机科学，2003，30(4):18-22．

［8］李允，罗蕾，熊光泽，面向普适计算的自适应技术研究［J］．电子学报，2004，32(5)：470-474．

［9］徐磊．普适计算中间件技术的研究［J］．计算机工程，2004，30(20):113-115．

［10］Dynamic component and code co-evolution. In 5th Int. Workshop on Principles of Software Evolution IWPSE 2002, pages 71-75, Orlando, FL, May 2002. ACM SIGSOFT.

［11］钱德沛，桂小林．基于Internet的网格计算模型研究［J］．西安交通大学学报，2001，35(10):1008-1011．

［12］肖连兵，黄林鹏．网格计算综述［J］．计算机工程，2002，28(3):1-3，50．

［13］张秋余，袁占亭，萧海东．网格计算技术的应用［J］．甘肃工业大学学报，2003，29(1):94-97．

［14］都志辉，陈渝，刘鹏．网格技术综述［J］．微型电脑应用，2002，(18)：62-64［2］．

［15］李长云，李赣生，何频捷．一种形式化的动态体系结构描述语言［J］．软件学报，2006，17(6):1349-1359．

［16］J.Kramer, J.Magee. The Evolving Philosophers Problem［J］. IEEE Transactions on Software Engineering,15(1):1293-1306,November 1990.

［17］李长云，阳爱民，满君丰，应晶．一种面向按需集成服务的业务模型构造方法［J］．计算机学报，2006，29(7):1095-1104.

［18］Peyman Oreizy. Issues in Modeling and Analyzing Dynamic Software Architectures［J］. Information and Computer Science.

［19］ J.Philipps, B.Rumpe. Refinement of Pipe and Filter Architectures［J］. Proceedings of FM’99, 1999, LNCS 1708:96-115.

［20］ A.Tsalgatidou, T. Pilioura. An Overview of Standards and Related Technology in Web services［J］. Distributed and Parallel Databases,12(3), Kluwer Academic Publishers, 2002.

［21］ 吕建，马晓星，陶先平，徐锋，胡昊．网构软件的研究与进展．南京大学计算机软件新技术

国家重点实验室南京大学计算机软件研究所.

［22］ Oreizy P,Taylor R. On the role of software architectures in runtime system reconfiguration［J］. In:Proc the 4th Int Conf Configurable Distributed Systems.Annapolis:IEEE Computer Society Press,1998.61-70.

［23］Garlan D,Cheng S W,Huang A C. Rainbow:Architecture-based self-adaptation with reusable infrastructure［J］. Comput,2004,37(10):46-54.

［24］梅宏，陈锋，冯耀东等．ABC：基于体系结构、面向构件的软件开发方法［J］．软件学报，2003，14(4):721-732．

［25］Vitharana P.Risks. challenges of component-based software development[J]. Commun ACM，2003，46:67-72.

［26］叶新林，赵文耘，蒋韬．支持演化与重配置的动态构架技术的研究［J］．计算机工程与应用，2004.7．

［27］窦蕾．面向构件的复杂软件系统中动态配置技术的研究［博士学位论文］．国防科学技术大学，2005．

［28］王树风．基于CORBA构件模型的动态配置平台的研究与实现［硕士学位论文］．国防科学技术大学，2004．

［29］余萍，马晓星，吕建，陶先平．一种面向动态软件体系结构的在线演化方法［J］．软件学报，2006，17(6):1360-1371．

［30］王映辉，王立福，张世琨，王琼芳．一种软件需求变化追踪方法［J］．电子学报，2006，34(8).

［31］汪洋，葛叶冰，李丽燕．软件构架在系统动态演化中的应用［J］．计算机应用研究，2000，(1).

［32］Dowling J,Cahill V. The K-component architecture meta-model for self-adaptive software［J］. In:Proceedings of the 3rd International Conference(REFLECTION 2001),LNCS 2192.Berlin:Springer,89-109.

［33］Dowling J,Cahill V. Dynamic software evolution and the k-component model［J］. In:Proc.of the Workshop on Software Evolution,OOPSLA.2001.

［34］Dellarocas, C., Klein, M., Shrobe, H. An Architecture for Constructing Self-Evolving Software Systems［J］. Proc. of the 3rd International Workshop on Software Architecture,1998.

［35］黄罡，梅宏，杨芙清．基于反射式软件中间件的运行时软件体系结构［J］．中国科学E辑技术科学，2004，34(2):121-138．

［36］骆华俊，唐稚松，郑建丹．可视化体系结构描述语言XYZ/ADL［J］．软件学报，2000，11(8):1024-1029．

［37］W Emmerich. Software engineering and middleware:A roadmap[A]. ICSE-Future of SE Track 2000［C］.2000.117-129.

［38］ Mary Shaw,David Garlan. Software Architecture (Perspectives on an Emerging Discipine)［M］. USA: Prentice Hall,1996.

［39］Object Management Group［EB/OL］. Model-Driven Architecture. www.omg.org/mda/.

［40］Perry DE, Wolf AL. Foundations for the study of software architecture[J]. ACM SIGSOFT Software Engineering Notes, 1992,17(4):44-52.

［41］Metayer DL. Describing software architecture styles using graph grammars. IEEE Trans. on Software Engineering, 1998,24(7):521-533.

［42］P. Oreizy, N. Medvidovic, R.N. Taylor. Architecture-Based Runtime Software Evolution［J］. Proceedings of ICSE'20, Kyoto, Japan, IEEE Computer Society Press, 1998:177-186.

［43］ E. M. Clarke, J. M. Wing. Formal methods: State of the art and future directions［J］. ACM Computing Surveys,1996,28(4):626-843.

［44］G. Abowd, R. Allen, D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture［J］. ACM Transactions on Software Engineering and Methodology, 1995,4(4):319-364.

［45］ N. Medvidovic. Formal definition of the chiron-2 software architectural style［J］. Technical Report UCI-ICS-95-24, University of California, Irvine, August 1995.

［46］K. J. Sullivan, M. Marchukov, and J. Socha. Analysis of a conict between aggregation and interface negotiation in microsoft's component object model［J］. IEEE Transactions on Software Engineering, 1999,25(4):584-599.

［47］ M. Moriconi, X. Qian, R.A. Riemenschneider. Correct architecture refinement［J］. IEEE Transactions on Software Engineering.1995,21(4):356-372.

［48］R. Allen，D. Garlan. A formal basis for architectural connection［J］. ACM Transactions on Software Engineering and Methodology,1997,6(3):213-249.

［49］ M .Bemardo, P. Ciancarini, and L. Donatiello. On the formalization of architectural types with process algebras［J］. In D. S. Rosenblum, editor, Proc. of the 8th ACM Int. Symp. on the Foundations of Software Engineering (FSE-8): 140148. ACM Press, November 2000.

［50］J. Magee, N. Dulay, S. Eisenbach, and J. Kramec. Specifying distributed software architectures[J]. In Proceedings of the Fifth European Software Engineering Conference, ESEC' 95, LNCS989, pages 137153. Springer-Verlag, September 1995.

［51］ F. Achermann, M. Lumpe, Jean-Guy, and 0: Neirstrasz. Piccola: a small composition language［M］. In H. Bowman and J. Derrick, eds. Formal Methods for Distributed Processing, an Object Oriented Approach. Cambridge University Press, 2001.

［52］DC. Luckham, J. Vera. An event-based architecture definition language［J］. IEEE Transactions on Software Engineering, 1995,21(9):717-734.

［53］G. Berry and G Boudol. The Chemical abstract machine［J］. Theoretical Computer Science, 1992(96):217-248.

［54］http://autofocus.in.tum.de/nelli/englisch/html/index.html

［55］P. Inverardi, A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model［J］. IEEE Transactions on Software Engineering, 1995,21(4):373-386.

［56］J .Philipps, B. Rumpe. Refinement of Pipe-and-Filter Architectures[J]. In: J. M. Wing, J. Woodcock, J. Davies (eds.).FM'99—Formal Methods, Proceedings of the World Congress on Formal Methods in the Development of Computing System. LNCS 1708: 96-115 Springer, 1999.

［57］C. Jansen, D. Bjomer. Where do architectures come from? From domain via requirements to software［J］. Technical report, Technical University of Denmark, April 1998.

［58］S .Nakajima, T. Tamai. Behavioural analysis of the enterprise javabeans component architecture. In M. B. Dwyer, editor, Model Checking Software: Proceedings of the 8th International SPIN Workshop, LNCS2057, 2001.

［59］J. Goguen. Parameterized programming and software architecture［J］. In Proceedings, Reuse'96, pages 2-11. IEEE Computer Society, April 1996.

［60］M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in Casl［J］. In Proc. of the 7th Intl. Conference on Algebraic Methodology and Software Technology, LNCS1548:341-357. Springer-Verlag, 1999.

［61］ J. Pennix, P. Alexander, and K. Havelund. Declarative specification of software architectures［J］. In the Proceedings of the 12th International Automated Software Engineering Conference., November 1997.

［62］ D.Garlan, R.Monroe, D.Wile. ACME:Architectural Description of Component-Based Systems［M］. Foundations of Component-Based Systems, G.T. Leavens, and M .Sitaraman, Cambridge University Press, 2000.

［63］ 孙昌爱，金茂忠，刘超．软件体系结构研究综述［J］．软件学报，2002，13(7):1229-1237．

［64］ 杨芙清．软件工程技术发展思索［J］．软件学报，2005,16(1)：1-7．

［65］ 李长云，邬惠峰，应晶，李赣生．软件体系结构驱动的运行环境［J］．小型微型计算机系统，2005,26(8):1358-1363．

［66］ 张云勇,张智江,刘锦德,刘韵洁．中间件技术原理与应用［M］．清华大学出版社，2004.10．

［67］ R.Allen, R.Douence, D.Garlan. Specifying and Analyzing Dynamic Software Architectures［J］. Proceedings on Fundamental Approaches to Software Engineering, Lisbon, Portugal,March 1998,LNCS 1382:21-37.

［68］ 胡海洋，马晓星，陶先平，吕建．反射中间件的研究与进展［J］．计算机学报，2005，28(9):1408-1420．

［69］冯铁，张家晨，陈伟等．基于框架和角色模型的软件体系结构规约［J］．软件学报，2000，11(8):1078-1086．

［70］ 马俊涛，傅韶勇，刘积仁．A-ADL：一种多智能体系统体系结构描述语言［J］．软件学报，2000，11(10):1382-1389．

［71］ 张家晨，冯铁，陈伟等．基于主动连接的软件体系结构及其描述方法［J］．软件学报，2000，11(8):1047-1052．

［72］M.A.Wermelinger. Specification of Software Architecture Reconfiguration［M］. PhD thesis, Universidade Nova de Lisboa, 1999.

［73］N. Medvidovic. ADLs and dynamic architecture changes［J］. Second International Software Architecture Workshop(ISAW-2), San Francisco, October 1996:24--27.

［74］马晓星，曹建农，吕建．一种面向图的分布web应用架构技术［J］．计算机学报，2003，26(9):1104-1115．

［75］ 黄罡，王千祥，曹东刚，梅宏．PKUAS:一种面向领域的构件运行支撑平台［J］．电子学报，2002，30(12A):39-43．

［76］任洪敏．基于π演算的软件体系结构形式化研究［博士学位论文］．上海，复旦大学，2003．

［77］ R. Milner, J. Parrow, D. Walker, A Calculus Of Mobile Processes［J］. Information and Computation, 1992,100(1):1-40.

［78］ R. Milner. Communicating and Mobile Systems:The Pi-Calculus［M］. Cambridge University Press, 1999.

［79］D.Sangiorgi. Expressing Mobility in Process Algebras:First-Order and Higher-Order Paradigms. PhD Thesis, University of Edinburgh, 1992.

［80］余雪丽．软件体系结构及实例分析［M］． 科学出版社,2004.8．

［81］J.R.Abrial. The B-Book:Assigning Programs to Meanings［M］. Cambridge University Press, 1996.

［82］J.Davies, J.Woodcock. Using Z: Specification, Refinement and Proof［M］. Prentice Hall International Series in Computer Science, 1996.

［83］J.Fitzgerald, P.Larsen. Modelling Systems: Practical Tools and Techniques for Software Development［M］. Cambridge University Press, 1998.

［84］M.G.Hinchey, S.A.Jarvis. Concurrent Systems: Formal Development in CSP［M］. McGraw-Hill, 1995.

［85］李长云，李莹，胡军，李赣生．基于软件体系结构的反射工作流［J］．计算机辅助设计与图形学报，2005，17(8):1844-1850．

［86］D.F.Souza, A.C Wills. Objects, Components and Frameworks with UML: The Catalysis Approach［M］. Addison-Wesley, 1998.

［87］F.Oquendo. π-ARL:An Architecture Refinement Language for Formally Modelling theStepwise Refinement of Software Architectures［J］. ACM SIGSOFT Software Engineering Notes, September 2004,29(5):1-20.

［88］A.Kerschbaumer. Behavioral Refinement of Software Architectures［J］. PhD thesis, Institute for Software Technology, Graz University of Technology, Austria, August 2002.

［89］许文，方海，林惠民. π演算互模拟判定算法的优化和实现［J］. 软件学报，2001，12(2):159-166.

［90］R.De Nicola, F.W.Vaandrager. Three Logics for Branching Bisimulation［J］. Journal of the ACM, 1995, 42(2):458-487.

［91］T.Bolusset, F.Oquendo. Formal Refinement of Software Architectures Based on Rewriting Logic［J］. ZB2002 International Workshop on Refinement of Critical Systems: Methods,Tools and Experience, Grenoble, Janvier 2002,29(5):1-20.

［92］李长云，邬惠峰，应晶，李赣生．支持领域复用的过程元模型［J］．小型微型计算机系统，2006，27(6):1083-1087．

［93］ D.Garlan. Style-based refinement for software architecture［J］. Second International Software Architecture Workshop (ISAW-2). San Francisco, CA, October 1996:72-75.

［94］A.Egyed, R.N.Mehta, N.Medvidovic. Software Connectors and Refinement in Family Architectures［J］. IW-SAPF 2000, Las Palmas de Gran Canaria, Spain, March 2000:96-106.

［95］D.Giannakopoulou. Model Checking for Concurrent Software Architectures［J］. PhD Thesis, University of London, March 1999.

［96］ T.Batista, A.Joolia, G.Coulson. Managing Dynamic Reconfiguration in Component-Based Systems［J］. EWSA 2005,2005:1-17.

［97］ B.Warboys, M.Greenwood, I.Robertson, R.Morrison, et al. The ArchWare Tower: The Implementation of an Active Software Engineering Environment Using a π-Calculus Based Architecture Description Language［J］. EWSA 2005, 2005，LNCS 3527:30-40.

［98］李长云，李赣生，李莹．基于扩展的粒度计算的软件体系结构模型：EGSA［J］．电子学报，2005，33(2):271-275．

［99］B.Warboys, B.Snowdon, R.M.Greenwood, et al. An Active-Architecture Approach to COTS

Integration［J］. IEEE SOFTWARE, 2005,July/August:20-27.

［100］ I. Georgiadis, J. Magee, J. Kramer. Self-Organising Software Architectures for Distributed Systems ［J］. Proceedings of the ACM SIGSOFT Workshop on Self-healing Systems, Charleston, South Carolina, ACM Press, 2002:33-38.