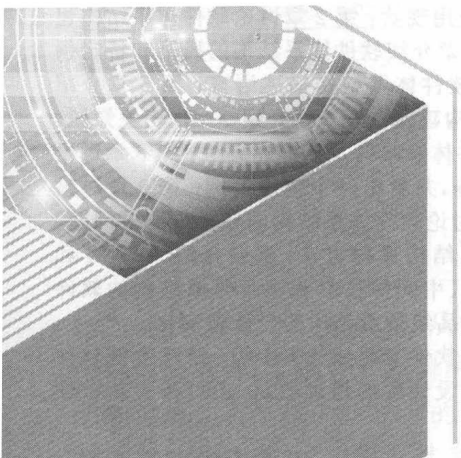
普通高等教育“十一五”国家级规划教材
高等学校软件工程专业系列教材

软件体系结构
原理、方法与实践（第3版）

◎ 张友生 编著

清華大學出版社

普通高等教育“十一五”国家级规划教材
高等学校软件工程专业系列教材



软件体系结构
原理、方法与实践（第3版）

◎ 张友生 编著

清华大学出版社
北京

内 容 简 介

本书系统地介绍软件体系结构的基本原理、方法和实践，全面反映软件体系结构研究和应用的最新进展。全书共13章，第1章简单地介绍软件体系结构的概念、发展和应用现状；第2章讨论软件体系结构建模，包括“4+1”视图模型、核心模型、生命周期模型和抽象模型；第3章介绍软件体系结构的风格和特定领域软件体系结构；第4章讨论软件体系结构的描述方法，重点介绍软件体系结构描述语言；第5章介绍UML的基础知识，以及如何使用UML及其扩展机制对软件体系结构建模；第6章介绍XML相关知识，重点讨论基于XML的软件体系结构描述语言；第7章介绍动态软件体系结构及其描述方法；第8章讨论基于服务的体系结构的关键技术和实现方法，以及如何构建一个服务，并给出一个应用实例；第9章讨论富互联网应用体系结构，重点介绍AJAX和Mashup技术；第10章讨论软件体系结构的分析与测试问题，重点介绍软件体系结构的可靠性风险分析；第11章讨论软件体系结构评估方法，重点介绍ATAM和SAAM方法；第12章讨论基于体系结构的软件开发，包括设计模式、中间件技术和基于体系结构的软件过程；第13章介绍软件产品线的原理和方法、框架技术，重点讨论产品线体系结构的设计和演化。

本书可作为计算机软件相关专业高年级本科生、硕士研究生（含软件工程硕士）和博士研究生的软件体系结构教材，也可作为软件工程高级培训、系统分析师和系统架构设计师的培训教材，还可作为高级软件开发人员的参考书。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。举报：010-62782989，beiqinquan@tup.tsinghua.edu.cn。

图书在版编目(CIP)数据

软件体系结构原理、方法与实践/张友生编著. --3版. --北京：清华大学出版社，2021.5 (2021.8重印)
高等学校软件工程专业系列教材
ISBN 978-7-302-56768-4

Ⅰ. ①软… Ⅱ. ①张… Ⅲ. ①软件－系统结构－高等学校－教材 Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2020)第211846号

策划编辑：魏江江
责任编辑：王冰飞
封面设计：刘 键
责任校对：徐俊伟
责任印制：沈 露

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦A座 **邮 编**：100084
社 总 机：010-62770175 **邮 购**：010-83470235
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-83470236
印 装 者：三河市龙大印装有限公司
经 销：全国新华书店
开 本：185mm×260mm **印 张**：23.5 **字 数**：587千字
版 次：2009年8月第1版 2021年5月第3版 **印 次**：2021年8月第3次印刷
印 数：48501～50500
定 价：59.80元

产品编号：087754-01

前言

体系结构(architecture)一词在英文里是“建筑”的意思，软件产业界通常翻译为“架构”。把软件比作一座楼房，从整体上讲，是因为它有基础、主体和装饰，即操作系统之上的基础设施软件、实现计算逻辑的主体应用程序、方便使用的用户界面程序。从细节上看每一个程序也是有结构的。早期的结构化程序就是以语句组成模块，模块的聚集和嵌套形成层层调用的程序结构，也就是体系结构。结构化程序的程序(表达)结构和(计算的)逻辑结构的一致性及自顶向下开发方法自然而然地形成了体系结构。

1. 为什么要学习软件体系结构

由于结构化程序设计时代程序规模不大，通过强调结构化程序设计方法学，自顶向下、逐步求精，并注意模块的耦合性，就可以得到相对良好的结构，所以，并未特别研究软件体系结构。

随着软件系统规模越来越大、越来越复杂，整个系统的结构和规格说明显得越来越重要。对于大规模的复杂软件系统来说，对总体的系统结构设计和规格说明比起对计算的算法和数据结构的选择已经变得明显重要得多。在此种背景下，人们认识到软件体系结构的重要性，并认为对软件体系结构的系统、深入的研究将会成为提高软件生产率和解决软件维护问题的新的最有希望的途径。

对于软件项目的开发来说，一个清晰的软件体系结构是首要的。传统的软件开发过程可以划分为从概念直到实现的若干个阶段，包括问题定义、需求分析、软件设计、软件实现及软件测试等。软件体系结构的建立应位于需求分析之后，软件设计之前。但在传统的软件工程方法中，需求和设计之间存在一条很难逾越的鸿沟，从而很难有效地将需求转换为相应的设计。而软件体系结构就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坦地过渡的问题。

2. 本书的内容

本书共分 13 章。

第 1 章简单介绍软件体系结构的概念、发展和应用现状。

第 2 章讨论软件体系结构建模，包括“4+1”视图模型、核心模型、生命周期模型和抽象模型。

第 3 章介绍软件体系结构的风格和特定领域软件体系结构。

第 4 章讨论软件体系结构的描述方法，重点介绍软件体系结构描述语言。

第 5 章介绍 UML 的基础知识，以及如何使用 UML 及其扩展机制对软件体系结构建模。

第 6 章介绍 XML 相关知识，重点讨论基于 XML 的软件体系结构描述语言。

第 7 章介绍动态软件体系结构及其描述方法。

第 8 章讨论基于服务的体系结构的关键技术和实现方法，以及如何构建一个服务，并给出一个应用实例。

第9章讨论富互联网应用体系结构,重点介绍AJAX和Mashup技术。

第10章讨论软件体系结构的分析与测试问题,重点介绍软件体系结构的可靠性风险分析。

第11章讨论软件体系结构评估方法,重点介绍ATAM和SAAM方法。

第12章讨论基于体系结构的软件开发,包括设计模式、中间件技术和基于体系结构的软件过程。

第13章介绍软件产品线的原理和方法、框架技术,重点讨论产品线体系结构的设计和演化。

3. 将本书作为教材

如果将本书作为教材,根据学生的不同层次,教师可以选讲其中一部分内容,下面列出一些建议,供参考。

(1) 本科生:建议将软件体系结构作为选修课程,安排24课时甚至更少的课时,主要讲解软件体系结构的发展和图形化建模,以及目前比较流行的一些体系结构风格和模式。可以不讲的章节包括2.2节、2.3节、2.4节,3.5节、3.6节、3.7节、3.9节、3.10节,第4章,5.10节、5.11节,6.4节,第7章,第10章,第11章,12.3节、12.5节、12.7节、12.8节,第13章。

(2) 硕士研究生:建议将软件体系结构作为选修课程(如果是计算机软件与理论专业的硕士研究生,则建议作为必修课程)。硕士研究生需要掌握软件体系结构的一些基本理论和常用的研究方法,侧重于软件体系结构学术性方面的内容。可以不讲的章节包括第5章、第6章、第8章、第9章、第12章、第13章。

(3) 软件工程硕士:建议将软件体系结构作为必修课程。软件工程硕士更侧重实践性,需要掌握软件体系结构的基本理论、软件体系结构的图形化建模,以及产业界常用的一些体系结构的原理及应用,还需要掌握软件体系结构的发展趋势。可以不讲的章节包括2.2节、2.3节、2.4节,3.6节、3.9节,第4章,5.11节,6.4节,第7章,第10章,12.7节。

(4) 博士研究生:建议将软件体系结构作为选修课程,或采用报告会议的形式,将本书的所有章节(或部分章节)作为专题报告,对计算机软件相关专业的博士研究生进行教学。

4. 诚挚致谢

在本书出版之际,要特别感谢国内外软件工程和软件体系结构专著、教材和许多高水平论文、报告的作者们(恕不一一列举,名单详见各章中的主要参考文献,扫描章末"思考题"下方的二维码可以浏览),他们的作品为本书提供了丰富的营养,使我们受益匪浅。在本书中借鉴了他们的部分材料,使本书能够尽量反映软件体系结构研究和实践领域的最新进展。

感谢阅读本书前两个版本的读者,特别要感谢使用本书作为教材的教师,他们为本书的修订和出版提出了宝贵的意见。

感谢希赛网为本书的意见反馈提供了空间和程序,感谢清华大学出版社的编辑,他们在本书的选题、编辑和定稿方面,给予了很大的帮助。

由于作者水平有限,时间紧迫,加上软件体系结构是一门新兴的学科,本身发展很快,对有些新领域作者尚不熟悉,因此,书中难免有不妥和疏漏之处,诚恳地期望各位专家和读者不吝指教和帮助。对此,我们将深为感激。

张友生

2021年1月

目　录

第 1 章　软件体系结构概论 …… 1

1.1　从软件危机谈起 …… 1

1.1.1　软件危机的表现 …… 1

1.1.2　软件危机的原因 …… 2

1.1.3　如何克服软件危机 …… 3

1.2　构件与软件重用 …… 3

1.2.1　构件模型及实现 …… 4

1.2.2　构件获取 …… 5

1.2.3　构件管理 …… 6

1.2.4　构件重用 …… 9

1.2.5　软件重用实例 …… 14

1.3　软件体系结构的兴起和发展 …… 17

1.3.1　软件体系结构的定义 …… 18

1.3.2　软件体系结构的意义 …… 19

1.3.3　软件体系结构的发展史 …… 21

1.4　软件体系结构的应用现状 …… 22

思考题 …… 27

第 2 章　软件体系结构建模 …… 28

2.1　“4＋1”视图模型 …… 28

2.1.1　逻辑视图 …… 29

2.1.2　开发视图 …… 30

2.1.3　进程视图 …… 31

2.1.4　物理视图 …… 32

2.1.5　场景 …… 34

2.2　软件体系结构的核心模型 …… 35

2.3　软件体系结构的生命周期模型 …… 36

2.3.1　各阶段之间的关系 …… 36

2.3.2　软件体系结构的生命周期 …… 37

2.4　软件体系结构的抽象模型 …… 39

2.4.1　构件及其关系的抽象描述 …… 39

2.4.2 连接件 …… 43
2.4.3 软件体系结构 …… 43
2.4.4 软件体系结构关系 …… 44
2.4.5 软件体系结构范式 …… 45
思考题 …… 47

第3章 软件体系结构风格 …… 48

3.1 经典软件体系结构风格 …… 48
3.1.1 管道与过滤器 …… 49
3.1.2 数据抽象和面向对象系统 …… 50
3.1.3 基于事件的系统 …… 50
3.1.4 分层系统 …… 51
3.1.5 仓库系统及知识库 …… 52
3.1.6 C2风格 …… 53
3.2 客户/服务器风格 …… 54
3.3 三层C/S体系结构风格 …… 56
3.3.1 各层的功能 …… 57
3.3.2 三层C/S体系结构应用实例 …… 59
3.3.3 三层C/S体系结构的优点 …… 63
3.4 浏览/服务器风格 …… 64
3.5 公共对象请求代理体系结构 …… 65
3.5.1 CORBA技术规范 …… 65
3.5.2 CORBA风格分析 …… 66
3.6 正交软件体系结构 …… 68
3.6.1 正交软件体系结构的抽象模型 …… 69
3.6.2 软件体系结构的正交化 …… 70
3.6.3 正交软件体系结构的实例 …… 71
3.6.4 正交软件体系结构的优点 …… 74
3.7 基于层次消息总线的体系结构风格 …… 75
3.7.1 构件模型 …… 76
3.7.2 构件接口 …… 77
3.7.3 消息总线 …… 77
3.7.4 构件静态结构 …… 79
3.7.5 构件动态行为 …… 79
3.7.6 运行时刻的系统演化 …… 80
3.8 异构结构风格 …… 80
3.8.1 异构结构的实例分析 …… 81
3.8.2 异构组合匹配问题 …… 83
3.9 互连系统构成的系统及其体系结构 …… 84

3.9.1 互连系统构成的系统 …… 85
3.9.2 基于 SASIS 的软件过程 …… 86
3.9.3 应用范围 …… 88
3.10 特定领域软件体系结构 …… 90
3.10.1 DSSA 的定义 …… 90
3.10.2 DSSA 的基本活动 …… 90
3.10.3 参与 DSSA 的人员 …… 92
3.10.4 DSSA 的建立过程 …… 93
3.10.5 DSSA 实例 …… 94
3.10.6 DSSA 与体系结构风格的比较 …… 97
思考题 …… 98

第 4 章 软件体系结构描述 …… 100

4.1 软件体系结构描述方法 …… 100
4.2 软件体系结构描述框架标准 …… 102
4.3 体系结构描述语言 …… 102
4.3.1 ADL 与其他语言的比较 …… 103
4.3.2 ADL 的构成要素 …… 104
4.4 典型的软件体系结构描述语言 …… 106
4.4.1 UniCon …… 107
4.4.2 Wright …… 108
4.4.3 C2 …… 109
4.4.4 Rapide …… 113
4.4.5 SADL …… 114
4.4.6 Aesop …… 114
4.4.7 ACME …… 116
思考题 …… 122

第 5 章 统一建模语言 …… 123

5.1 UML 概述 …… 123
5.1.1 UML 的发展历史 …… 123
5.1.2 UML 的应用领域 …… 124
5.2 UML 的结构 …… 125
5.2.1 结构概述 …… 125
5.2.2 事物 …… 126
5.2.3 关系 …… 127
5.2.4 图形 …… 129
5.3 用例图 …… 130
5.4 类图和对象图 …… 131

5.5 交互图 …… 132
5.5.1 顺序图 …… 133
5.5.2 通信图 …… 133
5.5.3 定时图 …… 134
5.6 状态图 …… 135
5.7 活动图 …… 136
5.7.1 基本活动图 …… 136
5.7.2 带泳道的活动图 …… 137
5.7.3 交互概览图 …… 137
5.8 构件图 …… 138
5.9 部署图 …… 139
5.10 使用 UML 建模 …… 140
5.11 使用 UML 的扩展机制 …… 144
思考题 …… 147

第 6 章 可扩展标记语言 …… 148

6.1 XML 概述 …… 148
6.1.1 XML 的特点 …… 149
6.1.2 XML 的作用 …… 150
6.1.3 XML 的应用 …… 152
6.2 解析 XML …… 152
6.2.1 XML 与 HTML 的区别 …… 153
6.2.2 XML 文档 …… 154
6.2.3 CSS 与 XSL …… 156
6.3 XML 编程接口 …… 159
6.3.1 API …… 159
6.3.2 XML 开发工具 …… 161
6.3.3 XML 建模 …… 162
6.4 基于 XML 的软件体系结构描述语言 …… 162
6.4.1 XADL 2.0 …… 163
6.4.2 XBA …… 167
思考题 …… 169

第 7 章 动态软件体系结构 …… 170

7.1 动态软件体系结构概述 …… 170
7.2 基于构件的动态系统结构模型 …… 172
7.3 πADL 动态体系结构 …… 175
7.3.1 πADL 描述体系结构的框架 …… 175
7.3.2 πADL 动态体系结构建模方法 …… 177

7.3.3 πADL 动态体系结构建模语义 …… 179
7.4 动态体系结构的描述 …… 180
7.4.1 动态体系结构描述语言 …… 180
7.4.2 动态体系结构的形式化描述 …… 182
7.5 动态体系结构的特征 …… 183
7.6 化学抽象机 …… 184
思考题 …… 187

第 8 章 基于服务的体系结构 …… 188

8.1 SOA 概述 …… 188
8.2 面向服务的分析与设计 …… 190
8.3 SOA 的关键技术 …… 192
8.4 SOA 的实现方法 …… 193
8.5 服务描述语言 …… 196
8.5.1 WSDL 概述 …… 196
8.5.2 使用 WSDL 文档 …… 196
8.5.3 WSDL 文档结构 …… 197
8.6 统一描述、发现和集成协议 …… 204
8.6.1 UDDI 数据模型 …… 205
8.6.2 注册 Web 服务 …… 209
8.6.3 调用 Web 服务 …… 209
8.7 消息封装协议 …… 209
8.7.1 消息封装和编码规则 …… 210
8.7.2 SOAP 应用 …… 211
8.7.3 REST …… 214
8.8 构造一个简单的服务 …… 215
8.8.1 编写服务器端 …… 215
8.8.2 编写客户端 …… 216
8.9 Web 服务的应用实例 …… 217
思考题 …… 220

第 9 章 富互联网应用体系结构 …… 222

9.1 RIA 的概念 …… 222
9.1.1 RIA 的提出 …… 222
9.1.2 丰富的含义 …… 223
9.1.3 RIA 的优点 …… 224
9.2 RIA 模型 …… 224
9.3 RIA 客户端开发技术 …… 225
9.4 AJAX 技术 …… 227

9.5 Mashup技术 …… 229
9.5.1 Mashup的体系结构 …… 229
9.5.2 Mashup实现技术 …… 230
思考题 …… 232

第10章 软件体系结构的分析与测试 …… 234

10.1 体系结构的可靠性建模 …… 234
10.2 软件体系结构的风险分析 …… 237
10.2.1 风险分析的方法 …… 238
10.2.2 风险分析的步骤 …… 239
10.3 基于体系结构描述的软件测试 …… 243
10.3.1 测试方法 …… 244
10.3.2 实例与实现 …… 245
思考题 …… 247

第11章 软件体系结构评估 …… 248

11.1 软件体系结构评估概述 …… 248
11.1.1 软件质量属性 …… 248
11.1.2 几个基本概念 …… 250
11.1.3 评估的主要方式 …… 252
11.2 ATAM评估方法 …… 253
11.2.1 ATAM评估的步骤 …… 254
11.2.2 ATAM评估的阶段 …… 260
11.3 SAAM评估方法 …… 263
11.3.1 SAAM评估的步骤 …… 263
11.3.2 SAAM评估实例 …… 267
思考题 …… 270

第12章 基于体系结构的软件开发 …… 272

12.1 设计模式 …… 272
12.1.1 设计模式概述 …… 272
12.1.2 设计模式的组成 …… 275
12.1.3 设计模式的描述 …… 276
12.1.4 模式和软件体系结构 …… 277
12.1.5 设计模式的层次 …… 278
12.1.6 设计模式的分类 …… 280
12.1.7 设计模式实例 …… 283
12.1.8 MVC模式的设计与实现 …… 286
12.2 中间件技术 …… 289

12.2.1 中间件概述 …… 289
12.2.2 主要的中间件 …… 291
12.2.3 中间件与构件的关系 …… 295
12.3 基于体系结构的设计方法 …… 296
12.3.1 有关术语 …… 297
12.3.2 ABSD 方法与生命周期 …… 298
12.3.3 ABSD 方法的步骤 …… 300
12.4 体系结构的设计与演化 …… 306
12.4.1 设计和演化过程 …… 307
12.4.2 实验原型阶段 …… 307
12.4.3 演化开发阶段 …… 309
12.5 基于体系结构的软件开发模型 …… 310
12.5.1 体系结构需求 …… 310
12.5.2 体系结构设计 …… 311
12.5.3 体系结构文档化 …… 312
12.5.4 体系结构复审 …… 312
12.5.5 体系结构实现 …… 312
12.5.6 体系结构演化 …… 313
12.6 应用开发实例 …… 314
12.6.1 系统简介 …… 314
12.6.2 系统设计与实现 …… 317
12.6.3 系统演化 …… 318
12.7 基于体系结构的软件过程 …… 319
12.7.1 有关概念 …… 319
12.7.2 软件过程网 …… 321
12.7.3 基本结构的表示 …… 322
12.7.4 基于体系结构的软件过程 Petri 网 …… 323
12.8 软件体系结构演化模型 …… 327
12.8.1 SA 静态演化模型 …… 328
12.8.2 SA 动态演化模型 …… 330
思考题 …… 333

第 13 章 软件产品线体系结构 …… 336

13.1 软件产品线的出现和发展 …… 336
13.1.1 软件体系结构的发展 …… 336
13.1.2 软件重用的发展 …… 337
13.2 软件产品线概述 …… 338
13.2.1 软件产品线的过程模型 …… 339
13.2.2 软件产品线的组织结构 …… 341

13.2.3 软件产品线的建立方式 …… 342
13.2.4 软件产品线的演化 …… 343
13.3 框架和应用框架技术 …… 344
13.4 软件产品线基本活动 …… 346
13.4.1 产品线分析 …… 347
13.4.2 产品开发 …… 349
13.5 软件产品线体系结构的设计 …… 350
13.5.1 产品线体系结构简介 …… 350
13.5.2 产品线体系结构的标准化和定制 …… 352
13.6 软件产品线体系结构的演化 …… 352
13.6.1 背景介绍 …… 353
13.6.2 两代产品的各种发行版本 …… 355
13.6.3 需求和演化的分类 …… 358
思考题 …… 361

第1章 软件体系结构概论

软件开发不仅要把程序写好，还要做前期的需求和体系结构设计（架构设计），而且，前期的设计比程序编码更重要。关于这一点，初入行业的软件工程师时常会感到困惑，因为他们往往初出茅庐，没有大型项目经验。在他们的心中，只有程序，没有体系结构。

为了解决软件新人的这些困惑，本章从软件危机谈起，介绍软件构件、软件重用的基本概念，阐述软件体系结构的意义、发展和应用现状。

1.1 从软件危机谈起

软件危机（software crisis）是指在计算机软件的开发（development）和维护（maintenance）过程中所遇到的一系列严重问题。20世纪60年代末至20世纪70年代初，“软件危机”一词在计算机界广为流传。事实上，从计算机诞生的那一天起，就出现了软件危机，只不过到了1968年在联邦德国加密施（Garmish）召开的国际软件工程会议上才被人们普遍认识到。

1.1.1 软件危机的表现

时至今日，“软件危机”虽然在软件行业已经家喻户晓，但很多工程师还是不能很好地理解软件危机究竟为何物，在实际的软件项目中，软件危机的“表现”在哪里，本节就这些内容做一个简单的介绍。

1. 软件成本日益增长

在计算机发展的早期，大型计算机系统主要是被设计应用于军事领域。这个时期，研制计算机的费用主要由国家财政提供，研制者很少考虑研制代价问题。随着计算机市场化和民用化的发展，代价和成本就成为投资者考虑的重要问题之一。

20世纪50年代，软件成本（cost）在整个计算机系统成本中所占的比例为10%～20%。随着软件产业的发展，软件成本日益增长。相反，计算机硬件随着技术的进步、生产规模的扩大，价格却在不断下降。这样一来，软件成本在计算机系统中所占的比例越来越大。

到20世纪60年代中期，软件成本在计算机系统中所占的比例已经增长到50%左右，而且该数字还在不断地递增。下面是一组来自美国空军计算机系统的数据：1955年，软件费用约占总费用的18%，1970年达到60%，1975年达到72%，1980年达到80%，1985年达到85%左右。如今，购买一台普通的计算机只要两三千元人民币，但如果把常用的操作系统、办公软件、安全软件等装好，却要远远超过购买计算机的费用。

2. 开发进度难以控制

软件是逻辑、智力产品,软件的开发需建立庞大的逻辑体系,这与其他产品的生产是不一样的。例如,工厂里要生产某种机器,在时间紧的情况下可以要求工人加班或者实行"三班倒",而这些方法都不能用在软件开发上。

在软件开发过程中,用户需求(requirement)变化等各种意想不到的情况层出不穷,令软件开发过程很难保证按预定的计划实现,给项目计划和论证工作带来了很大的困难。

Brooks在其名著《人月神话》中指出:"在已拖延的软件项目上,增加人力只会使其更难按期完成"。事实上,软件系统的结构很复杂,各部分附加联系极大,盲目增加软件开发人员并不能成比例地提高软件开发能力。相反,随着人员数量的增加,人员的组织、协调、通信、培训和管理等方面的问题将更为严重。

许多重要的大型软件开发项目,如IBM OS/360和世界范围的军事命令控制系统(WWMCCS),在耗费了大量的人力和财力之后,由于离预定目标相差甚远不得不宣布失败。

3. 软件质量差

软件项目即使能按预定日期完成,结果却不尽如人意。1965—1970年,美国范登堡空军基地发射火箭多次失败,绝大部分故障是由应用程序错误造成的。程序的一些微小错误可以造成灾难性的后果。

在"软件作坊"里,由于缺乏工程化思想的指导,程序员总是习惯性地以自己的想法去代替用户对软件的需求,软件设计带有随意性,很多功能只是程序员的"一厢情愿",这是造成软件不能令人满意的重要因素。

4. 软件维护困难

正式投入使用的软件,总是存在着一定数量的错误,在不同的运行条件下,软件就会出现故障,因此需要维护。但是,在软件设计和开发过程中,由于没有严格遵循软件开发标准,各种随意性很大,没有完整的真实反映系统状况的记录文档,给软件维护造成了巨大的困难。特别是在软件使用过程中,原来的开发人员可能因各种原因已经离开原来的开发组织,使得软件几乎不可维护。

另外,软件修改是一项很"危险"的工作,对一个复杂的逻辑过程,哪怕做一项微小的改动,都可能引入潜在的错误,常常会发生"纠正一个错误带来更多新错误"的问题,从而产生副作用。

有资料表明,工业界为维护软件支付的费用占全部硬件和软件费用的40%~75%。

1.1.2 软件危机的原因

从软件危机的种种表现和软件作为逻辑产品的特殊性可以发现软件危机的原因,可以归结为如下4个方面。

1. 用户需求不明确

在软件开发过程中,需求多变是造成项目失败的主要原因之一。具体来说,用户需求不明确问题主要体现在以下4个方面。

(1) 在软件开发出来之前,用户自己也不清楚软件的具体需求。

(2) 用户对软件需求的描述不精确,可能有遗漏、有二义性,甚至有错误。

(3) 在软件开发过程中，用户还提出修改软件功能(function)、界面(interface)、支撑环境(environment)等方面的要求。

(4) 软件开发人员对用户需求的理解与用户原本的愿望有差异。

2. 缺乏正确的理论指导

缺乏有力的方法学和工具方面的支持。软件开发过程是复杂的逻辑思维过程，其产品极大程度地依赖于开发人员高度的智力投入。由于过分地依靠程序设计人员在软件开发过程中的技巧和创造性，加剧软件产品的个性化，也是发生软件危机的一个重要原因。

3. 软件规模越来越大

随着软件应用范围的增大，软件规模愈来愈大。大型软件项目需要组织一定的人力共同完成，而多数管理人员缺乏开发大型软件系统的经验，多数软件开发人员又缺乏管理方面的经验。各类人员的信息交流不及时、不准确，有时还会产生误解。软件项目开发人员不能有效地、独立自主地处理大型软件的全部关系和各个分支，因此容易产生疏漏和错误。

4. 软件复杂度越来越高

软件不仅是在规模上快速地发展扩大，而且其复杂性(complexity)也急剧地增加。软件产品的特殊性和人类智力的局限性，导致人们无力处理“复杂问题”。所谓“复杂问题”的概念是相对的，一旦人们采用先进的组织形式、开发方法和工具提高了软件开发效率和能力，新的、更大的、更复杂的问题又摆在人们的面前。

1.1.3 如何克服软件危机

人们在认真地研究和分析了软件危机背后的真正原因之后，得出了“人们面临的不单是技术问题，更重要的还是管理问题。管理不善必然导致失败”的结论，便开始探索用工程的方法进行软件生产的可能性，即用现代工程的概念、原理、技术和方法进行计算机软件的开发、管理和维护。于是，计算机科学技术的一个新领域——软件工程(software engineering)，诞生了。

软件工程是用工程、科学和数学的原则与方法研制、维护计算机软件的有关技术及管理方法。软件工程包括三个要素：方法、工具和过程，其中：

(1) 软件工程方法为软件开发提供了“如何做”的技术，是完成软件工程项目的技术手段。

(2) 软件工具是人们在开发软件的活动中智力和体力的扩展和延伸，为软件工程方法提供了自动的或半自动的软件支撑环境。

(3) 软件工程过程则是将软件工程的方法和工具综合起来以达到合理、及时地进行计算机软件开发的目的。

迄今为止，软件工程的研究与应用已经取得很大成就，它在软件开发方法、工具、管理等方面的应用大大缓解了软件危机造成的被动局面。

1.2 构件与软件重用

当前社会的信息化过程对软件需求的增长非常迅速，但软件的开发与生产能力却相对不足，这不仅造成许多急需的软件迟迟不能被开发出来，而且形成了软件脱节现象。自 20 世纪 60 年代人们认识到软件危机、并提出软件工程以来，已经对软件开发问题进行了不懈的

研究。近年来人们认识到,要提高软件开发效率,提高软件产品质量,必须采用工程化的开发方法与工业化的生产技术。这包括技术与管理两方面的问题:在技术上,应该采用基于重用(英文单词为"reuse",有些文献翻译为"复用")的软件生产技术;在管理上,应该采用多维的工程管理模式。

近年来,人们认识到,要真正解决软件危机,实现软件的工业化生产是唯一可行的途径。分析传统工业及计算机硬件产业成功的模式可以发现,这些工业的发展模式均是符合标准的零部件/构件(英文单词为"component",有些文献翻译为"组件"或"部件")生产以及基于标准构件的产品生产,其中,构件是核心和基础,重用是必需的手段。实践表明,这种模式是产业工程化、工业化的成功之路,也将是软件产业发展的必经之路。

软件重用是指在两次或多次不同的软件开发过程中重复使用相同或相近软件元素的过程。软件元素包括程序代码、测试用例、设计文档、设计过程、需求分析文档甚至领域(domain)知识。通常,把这种可重用的元素称作软构件(software component),简称构件。可重用的软件元素越多,就称重用的粒度(granularity)越大。

使用软件重用技术可以减少软件开发活动中大量的重复性工作,这样就能提高软件生产率,降低开发成本,缩短开发周期。同时,由于软构件大都经过严格的质量认证,并在实际运行环境中得到检验,因此,重用软构件有助于改善软件质量。此外,大量使用软构件,软件的灵活性和标准化程度也能得到提高。

1.2.1 构件模型及实现

一般认为,构件是指语义完整、语法正确、有可重用价值的单位软件,是软件重用过程中可以明确辨识的系统;结构上,它是语义描述、通信接口和实现代码的复合体。简单地说,构件是具有一定的功能,能够独立工作或能同其他构件装配起来协调工作的程序体,构件的使用同它的开发和生产无关。从抽象程度来看,面向对象(Object Orientation,OO)技术已达到了类级重用(代码重用),它以类为封装的单位。这样的重用粒度还太小,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。整个构件隐藏了具体的实现,只用接口对外提供服务。

构件模型(model)是对构件本质特征的抽象描述。目前,国际上已经形成了许多构件模型,这些模型的目标和作用各不相同,其中部分模型属于参考模型(例如,3C模型),部分模型属于描述模型(例如,RESOLVE模型和REBOOT模型),还有一部分模型属于实现模型。近年来,已形成三个主要流派,分别是OMG(Object Management Group,对象管理组织)的CORBA(Common Object Request Broker Architecture,通用对象请求代理结构)、Sun的EJB(Enterprise Java Bean)和Microsoft的DCOM(Distributed Component Object Model,分布式构件对象模型)。这些实现模型将构件的接口与实现进行了有效的分离,提供了构件交互(interaction)的能力,从而增加了重用的机会,并适应了目前网络环境下大型软件系统的需要。

国内许多学者在构件模型的研究方面做了不少工作,取得了一定的成绩。其中,较为突出的是北京大学杨芙清院士等提出的"青鸟构件模型",下面就以这个模型为例进行讲解。

青鸟构件模型充分吸收了上述模型的优点,并与它们相容。青鸟构件模型由外部接口

(interface)与内部结构两部分组成，如图 1-1 所示。

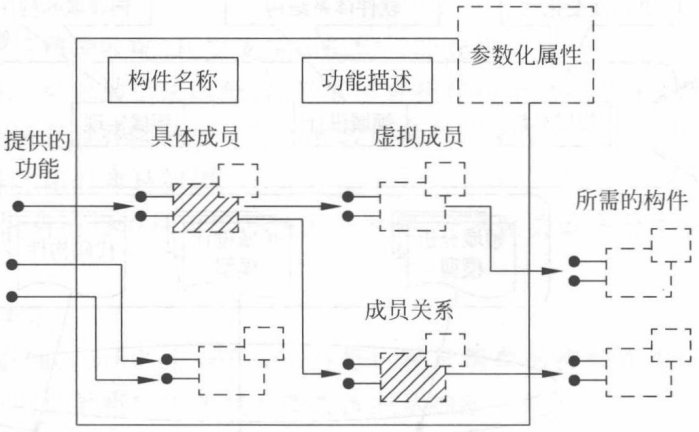


图 1-1　青鸟构件模型

1. 外部接口

构件的外部接口是指构件向其重用者提供的基本信息，包括构件名称、功能描述、对外功能接口、所需的构件、参数化属性等。外部接口是构件与外部世界的一组交互点，用来说明构件所提供的服务(消息、操作、变量)。

2. 内部结构

构件的内部结构包括两方面内容：内部成员以及内部成员之间的关系。内部成员包括具体成员与虚拟成员；成员关系包括内部成员之间的互联，以及内部成员与外部接口之间的互联。

构件实现是指具体实现构件功能的逻辑系统，通常也称代码构件。构件实现由构件生产者完成，构件重用者则不必关心构件的实现细节。重用者在重用构件时，可以对其定制，也可以对其特例化。

1.2.2　构件获取

存在大量的可重用的构件是有效地使用重用技术的前提。通过对可重用信息与领域的分析，可以得到以下结论。

(1) 可重用信息具有领域特定性，即可重用性不是信息的一种孤立的属性，它依赖于特定的问题和特定的问题解决方法。为此，在识别(identify)、获取(capture)和表示(represent)可重用信息时，应采用面向领域的策略。

(2) 领域具有内聚性(cohesion)和稳定性(stability)，即关于领域的解决方法是充分内聚和充分稳定的。一个领域的规约和实现知识的内聚性，使得可以通过一组有限的、相对较少的可重用信息来解决大量问题。领域的稳定性使得获取的信息可以在较长的时间内多次重用。

领域是一组具有相似或相近软件需求的应用系统所覆盖的功能区域，领域工程(domain engineering)是一组相似或相近系统的应用工程(application engineering)建立基本能力和必备基础的过程。领域工程过程可划分为领域分析、领域设计、领域实现等多个活动，其中的活动与结果如图 1-2 所示。

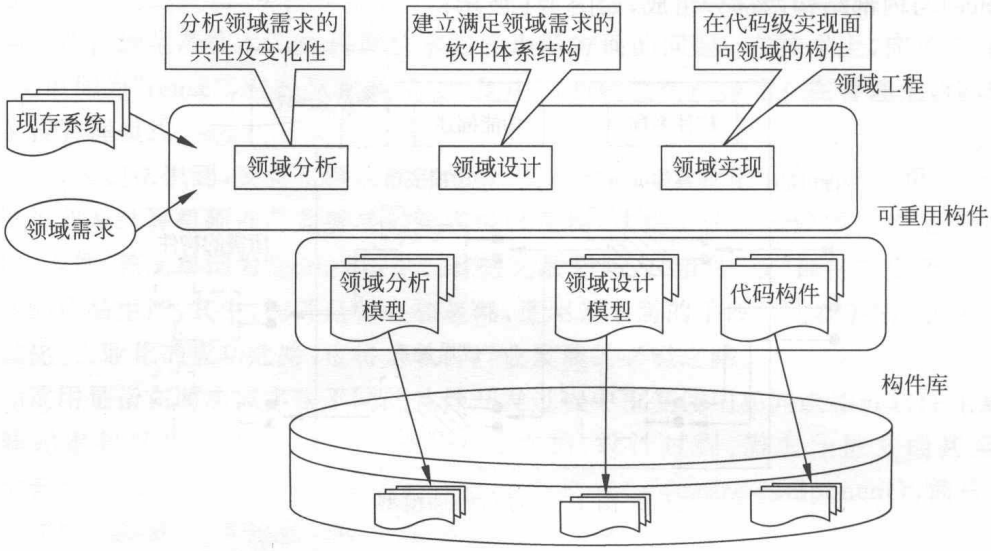


图 1-2　领域工程中的活动与结果

在建立基于构件的软件开发(Component-Based Software Development,CBSD)中,构件获取有以下多种途径。

(1) 从现有构件中获得符合要求的构件,直接使用或做适应性(flexibility)修改,得到可重用的构件。

(2) 通过遗留工程(legacy engineering),将具有潜在重用价值的构件提取出来,得到可重用的构件。

(3) 从市场上购买现成的商业构件,即 COTS(Commercial Off-The-Shell)构件。

(4) 开发新的符合要求的构件。

一个组织在进行以上决策时,必须考虑不同方式获取构件的一次性成本和后续的维护成本,然后做出最优的选择。

1.2.3　构件管理

对大量的构件进行有效的管理,以方便构件的存储、检索和提取,是成功重用构件的必要保证。构件管理的内容包括构件描述、构件分类、构件库组织、人员及权限管理、用户意见反馈等。

1. 构件描述

构件模型是对构件本质的抽象描述,主要是为构件的制作与构件的重用提供依据;从管理角度出发,也需要对构件进行描述,例如,实现方式、实现体、注释、生产者、生产日期、大小、价格、版本、关联构件等信息,它们与构件模型共同组成对构件的完整描述。

2. 构件分类与组织

为了给使用者在查询构件时提供方便,同时也为了更好地重用构件,就必须对收集和开发的构件进行分类(classify)并置于构件库的适当位置。构件的分类方法及相应的库结构对构件的检索和理解有极为深刻的影响。因此,构件库的组织应方便构件的存储和检索。

可重用技术对构件库组织方法的要求如下。

(1) 支持构件库的各种维护动作，如增加、删除以及修改构件，尽量不要影响构件库的结构。

(2) 不仅要支持精确匹配，还要支持相似构件的查找。

(3) 不仅能进行简单的语法匹配，而且能够查找在功能或行为方面等价或相似的构件。

(4) 对应用领域具有较强的描述能力和较好的描述精度。

(5) 库管理员和用户容易使用。

目前，已有的构件分类方法可以归纳为三大类，分别是关键字分类法、刻面分类法和超文本组织方法。

1) 关键字分类法

关键字分类法(keyword classification)是一种最简单的构件库组织方法，其基本思想是：根据领域分析的结果将应用领域的概念按照从抽象到具体的顺序逐次分解为树状或有向无回路图结构。每个概念用一个描述性的关键字表示。不可分解的原子级关键字包含隶属于它的某些构件。图 1-3 给出了构件库的关键字分类结构示例，它支持图形用户界面设计。

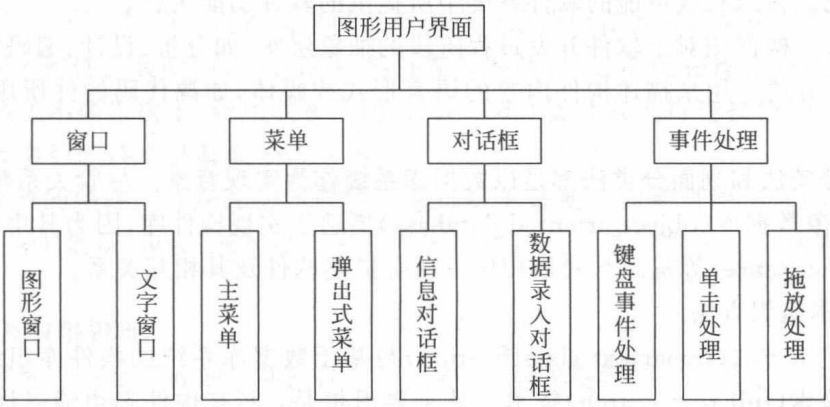


图 1-3　关键字分类结构示例

当加入构件时，库管理员必须对构件的功能或行为进行分析，在浏览上述关键字分类结构的同时将构件置于最合适的原子级关键字之下。如果无法找到构件的属主关键字，可以扩充现有的关键字分类结构，引进新的关键字。但库管理员必须保证，新关键字有相同的领域分析结果作为支持。例如，如果需要增加一个“图形文字混合窗口”构件时，则只需把该构件放到属主关键字“窗口”的下一级。

2) 刻面分类法

刻面分类法(faceted classification)的主要思想来源于图书馆学，这种分类方法是 Prieto-Diaz 和 Freeman 在 1987 年提出来的。在刻面分类机制中，定义若干用于刻画构件特征的“面”(facet)，每个面包含若干概念，这些概念表述构件在面上的特征。刻面可以描述构件执行的功能、被操作的数据、构件应用的语境或任意其他特征。描述构件的刻面的集合称为刻面描述符(facet descriptor)，通常，刻面描述被限定不超过 8 个刻面。当描述符中出现空的特征值时，表示该构件没有相应的面。

作为一个简单的在构件分类中使用刻面的例子，考虑使用下列构件描述符的模式：

```
{function,object type,system type}
```

刻面描述符中的每个刻面可含有一个或多个值,这些值一般是描述性关键词。例如,如果功能是某构件的刻面,赋给此刻面的典型值可能是:

```
function = (copy,from) or (copy,replace,all)
```

多个刻面值的使用使得原函数 copy 能够被更完全地细化。

关键词(值)被赋给重用库中的每个构件的刻面集,当软件工程师在设计中希望查询构件库以发现可能的构件时,规定一列值,然后到库中寻找匹配项。可使用自动工具完成同义词词典功能,这使得查找不仅包括软件工程师给出的关键词,还包括这些关键词的技术同义词。

青鸟构件库就是采用刻面分类法对构件进行分类的,这些刻画包括如下几项。

(1) 使用环境。使用(包括理解/组装/修改)该构件时必须提供的硬件和软件平台(platform)。

(2) 应用领域。构件原来或可能被使用到的应用领域(及其子领域)的名称。

(3) 功能。在原有或可能的软件系统中所提供的软件功能集合。

(4) 层次。构件相对于软件开发过程阶段的抽象层次,如分析、设计、编码等。

(5) 表示方法。用来描述构件内容的语言形式或媒体,如源代码构件所用的编程语言环境等。

关键字分类法和刻面分类法都是以数据库系统作为实现背景。尽管关系数据库可供选用,但面向对象数据库(object-oriented database)更适于实现构件库,因为其中的复合对象、多重继承(inheritance)等机制与表格相比更适合描述构件及其相互关系。

3) 超文本组织方法

超文本组织方法(hypertext classification)与基于数据库系统的构件库组织方法不同,它基于全文检索(full text search)技术。其主要思想是:所有构件必须辅以详尽的功能或行为(performance)说明文档;说明中出现的重要概念或构件以网状链接方式相互连接;检索者在阅读文档的过程中可按照人类的联想思维方式任意跳转到包含相关概念或构件的文档;全文检索系统将用户给出的关键字与说明文档中的文字进行匹配,实现构件的浏览式检索。

超文本是一种非线性的网状信息组织方法,它以结点为基本单位,链作为结点之间的联想式关联,如图 1-4 所示。

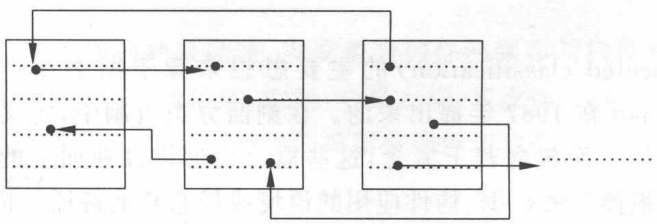


图 1-4 超文本结构示意图

结点是一个信息块,对可重用构件而言,结点可以是域概念、功能或行为名称、构件名称等。在图形用户界面上,结点可以是字符串,也可以是图像、声音、动画等。超文本组织方法

为构造构件和重用构件提供了友好、直观的多媒体方式。由于网状结构比较自由、松散，因此，超文本组织方法比前两种方法更易于修改构件库的结构。

例如，Windows 环境下的联机帮助系统就是一种典型的超文本系统。为构造构件的文档，首先要根据领域分析的结果在说明文档中标识超文本结点并在相关文档中建立链接关系，然后用类似于联机帮助系统编译器的工具对构件的说明文档进行编译，最后用相应的工具(例如，IE 浏览器)运行编译后的目标。

如果把软件系统看成是构件的集合，那么从构件的外部形态来看，构成一个系统的构件可分为以下 5 类。

(1) 独立而成熟的构件。独立而成熟的构件得到了实际运行环境的多次检验，该类构件隐藏了所有接口，用户只需要用规定好的命令进行使用。例如，数据库管理系统和操作系统等。

(2) 有限制的构件。有限制的构件提供了接口，指出了使用的条件和前提，这种构件在装配时，会产生资源冲突、覆盖等影响，在使用时需要加以测试。例如，各种面向对象程序设计语言中的基础类库等。

(3) 适应性构件。适应性构件进行了包装或使用了接口技术，对不兼容性、资源冲突等进行了处理，可以直接使用。这种构件可以不加修改地使用在各种环境中，如 ActiveX 等。

(4) 装配的构件。装配(assemble)的构件在安装时，已经装配在操作系统、数据库管理系统或信息系统不同层次上，使用胶水代码(glue code)就可以进行连接使用。目前一些软件商提供的大多数软件产品都属于这一类。

(5) 可修改的构件。可修改的构件可以进行版本替换。如果对原构件修改错误、增加新功能，可以利用重新“包装”或写接口来实现构件的替换。这种构件在应用系统开发中使用得比较多。

3. 人员及权限管理

构件库系统是一个开放的公共构件共享机制，任何使用者都可以通过网络访问构件库，这在为使用者带来便利的同时，也给系统的安全性带来了一定的风险，因此有必要对不同使用者的访问权限(privilege)做出适当的限制，以保证数据安全。

一般来讲，构件库系统包括 5 类用户，即注册用户、公共用户、构件提交者、一般系统管理员和超级系统管理员。他们对构件库分别有不同的职责和权限，这些人员相互协作，共同维护着构件库系统的正常运作。同时，系统为每一种操作定义一个权限，包括提交构件、管理构件、查询构件及下载构件。每一用户可被赋予一项或多项操作权限，这些操作权限组合形成该人员的权限，从而支持对操作的分工，为权限分配提供了灵活性。

1.2.4 构件重用

构件开发的目的是重用，为了让构件在新的软件项目中发挥作用，库的使用者必须完成以下工作。

1. 检索与提取构件

构件库的检索方法与组织方式密切相关，因此，本节针对 1.2.3 节介绍的关键字分类法、刻面分类法和超文本组织方法，分别讨论相应的检索方法。

1) 基于关键字的检索法

这种检索方法的基本思想是：系统在图形用户界面上将构件库的关键字树状结构直观

地展示给用户；用户通过对树状结构的逐级浏览寻找需要的关键字并提取相应的构件。当然，用户也可以直接给出关键字(可含通配符)，由系统自动给出合适的候选构件清单。

这种方法的优点是简单、易于实现，但在某些场合没有应用价值，因为用户往往无法用构件库中已有的关键字描述期望的构件功能或行为，对库的浏览也容易使用户迷失方向。

2) 刻面检索法

该方法基于刻面分类法，由以下三步构成。

第一步，构造查询。用户提供要查找的构件在每个刻面上的特征，生成构件描述符。此时，用户可以从构件库已有的概念中进行挑选，也可将某些特征值指定为空。系统在检索过程中将忽略特征值为空的刻面。

第二步，检索构件。实现刻面检索法的计算机辅助软件工程(Computer Aided Software Engineering，CASE)工具在构件库中寻找相同或相近的构件描述符及相应的构件。

第三步，对构件进行排序。被检索出来的构件清单除按相似程度排序外，还可以按照与重用有关的度量信息排序。例如，构件的复杂性，可重用性，已成功的重用次数等。

这种方法的优点是它易于实现相似构件的查找，但用户在构造查询时比较麻烦。

3) 超文本检索法

超文本检索法的基本步骤是：用户首先给出一个或数个关键字，系统在构件的说明文档中进行精确或模糊的语法匹配，匹配成功后，向用户列出相应的构件说明。如1.2.3节所述，构件说明是含有许多超文本结点的正文，用户阅读这些正文时可实现多个构件说明文档之间的自由跳转，最终选择合适的构件。为了避免用户在跳转过程中迷失方向，系统可以通过图形界面提供浏览历史信息图，允许将特定画面定义为“书签”并随时跳转至“书签”，并帮助用户逆跳转路径而逐步返回。

这种方法的优点是用户界面友好，但在某些情况下用户难以在超文本浏览过程中正确选取构件。

4) 其他检索方法

上述检索方法基于语法(syntax)匹配，要求使用者对构件库中出现的众多词汇有较全面的把握、较精确的理解。理论的检索方法是语义(semantic)匹配：构件库的用户以形式化(formalization)手段描述所需要的构件的功能或行为语义，系统通过定理证明及基于知识的推理过程寻找语义上等价或相近的构件。遗憾的是，这种基于语义的检索方法涉及许多人工智能(artificial intelligence)难题，目前尚难于支持大型构件库的工程实现。

2. 理解与评价构件

要使库中的构件在当前的开发项目中发挥作用，准确地理解构件至关重要。当开发人员需要对构件进行某些修改时，情况更是如此。考虑到设计信息对于理解构件的必要性以及构件的用户逆向发掘设计信息的困难性，必须要求构件的开发过程遵循公共软件工程规范，并且在构件库的文档中，全面、准确地说明以下内容。

(1) 构件的功能与行为。

(2) 相关的领域知识。

(3) 可适应性约束条件与例外情形。

(4) 可以预见的修改部分及修改方法。

但是，如果软件开发人员希望重用以前并非专为重用而设计的构件，上述假设则不能成

立。此时，开发人员必须借助 CASE 工具对候选构件进行分析。这种 CASE 工具对构件进行扫描，将各类信息存入某种浏览数据库，然后回答构件用户的各类查询，进而帮助理解。

逆向工程是理解构件的另一种重要手段。它试图通过对构件的分析，结合领域知识，半自动地生成相应的设计信息，然后借助设计信息完成对构件的理解和修改。

对构件可重用的评价，是通过收集并分析构件的用户在实际重用该构件的历史过程中的各种反馈信息来完成的。这些信息包括：重用成功的次数，对构件的修改量，构件的健壮性度量，性能度量等。

3. 修改构件

理想的情形是对库中的构件不做修改而直接用于新的软件项目。但是，在大多数情况下，必须对构件进行或多或少的修改，以适应新的需求。为了减少构件修改的工作量，要求开发人员尽量使构件的功能、行为和接口设计更为抽象化、通用化和参数化。这样，构件的用户即可通过对实参的选取来调整构件的功能或行为。如果这种调整仍不足以使构件适用于新的软件项目，用户就必须借助设计信息和文档来理解、修改构件。所以，与构件有关的文档和抽象层次更高的设计信息对于构件的修改至关重要。例如，如果需要将 C 语言书写的构件改写为 Java 语言形式，构件的算法描述就十分重要。

4. 构件组装

构件组装是指将库中的构件经适当修改后相互连接，或者将它们与当前开发项目中的软件元素相连接，最终构成新的目标软件。构件组装技术大致可分为基于功能的组装技术、基于数据的组装技术和面向对象的组装技术。

1）基于功能的组装技术

基于功能的组装技术采用子程序调用和参数传递的方式将构件组装起来。它要求库中的构件以子程序/过程/函数的形式出现，并且接口说明必须清晰。当使用这种组装技术进行软件开发时，开发人员首先应对目标软件系统进行功能分解，将系统分解为强内聚、松耦合的功能模块，然后根据各模块的功能需求提取构件，对它进行适应性修改后再挂接在上述功能分解框架(framework)中。

2）基于数据的组装技术

基于数据的组装技术首先根据当前软件问题的核心数据结构设计一个框架，然后根据框架中各结点的需求提取构件并进行适应性修改，再将构件逐个分配至框架中的适当位置。此后，构件的组装方式仍然是传统的子程序调用与参数传递。这种组装技术也要求库中构件以子程序形式出现，但它所依赖的软件设计方法不再是功能分解，而是面向数据的设计方法，例如，Jackson 系统开发方法。

3）面向对象的组装技术

由于封装和继承特征，面向对象方法比其他软件开发方法更适合支持软件重用。在面向对象的软件开发方法中，如果从类库中检索出来的基类能够完全满足新软件项目的需求，则可以直接应用。否则，必须以类库中的基类为父类采用构造法或子类法生成子类。

方法一：构造法

为了在子类中使用库中基类的属性(attribute)和方法(method)，可以考虑在子类中引进基类的对象作为子类的成员变量，然后在子类中通过成员变量重用基类的属性和方法。以下代码是一个构造法的例子。

```
//定义基类
class Person{
public:
    Person(char * name, int age);
    ~Person();
protected:
    char * name;
    int age;
};

//基类构造函数
Person::Person(char * name, int age)
{
    Person::name = new char[strlen(name) + 1];
    strcpy(Person::name,name);
    Person::age = age;
    cout << "Construct Person" << name << "," << age << ".\n";
    return;
}
//基类析构函数
Person:: ~Person()
{
    cout << "Destruct Person" << name << "," << age << ".\n";
    delete name;
    return;
}

//下面采用构造法生成 Teacher 类
class Teacher{
public:
    Teacher(char * name, int age, char * teaching);
    ~Teacher();
protected:
    Tperson * Person;
    char * course;
};

// Teacher 类的实现
Teacher::Teacher(char * name, int age, char * teaching)
{
    //重用基类的方法 Person()
    Tperson = new Person(name, age);
    strcpy(course, teaching);
    return;
}
Teacher::~Teacher()
{
    delete Tperson;
}
```

方法二：子类法

与构造法完全不同，子类法将新子类直接说明为库中基类的子类，通过继承和修改基类的属性与行为完成新子类的定义。以下代码是子类法的一个例子。

```
//定义基类
class Person{
public:
    Person(char * name, int age);
    ~Person();
protected:
    char * name;
    int age;
};

//基类构造函数
Person::Person(char * name, int age)
{
    Person::name = new char[strlen(name) + 1];
    strcpy(Person::name,name);
    Person::age = age;
    cout << "Construct Person" << name << "," << age << ".\n";
    return;
}
//基类析构函数
Person:: ~Person()
{
  cout << "Destruct Person" << name << "," << age << ".\n";
    delete name;
    return;
}

//下面采用子类法构造 Teacher 类
class Teacher : public Person{
public:
    Teacher(char * name, int age, char * teaching):Person(name,age)
    {
        course = new char[strlen(teaching) + 1];
        strcpy(course, teaching);
        return;
    }
    ~Teacher();
    {
        delete course;
        return;
    }
protected:
    char * course;
};
```

1.2.5 软件重用实例

本节针对电子政务领域产品开发过程中的重用策略与方法,从系统分析、设计到编码,介绍软件的领域重用与层次重用的方法。

1. 应用背景

随着信息化技术的发展和政府上网工程的推进,电子政府与电子政务正逐步走向成熟,信息化管理和办公自动化成为政府机关的一个战略性课题。为了进一步推动政府信息化的建设,必须进一步研究和开发适应新时代的基于 Internet 和 Intranet 的办公管理系统。通用办公管理系统(General Office Management System,GOMS)就是这样的一个系统。

GOMS 是面向政府机关的信息化管理和办公自动化系统,是电子政务的主要组成部分。因此,GOMS 和其他电子政务系统一样,存在跨平台、分布、异构以及对原有应用系统进行整合的问题。为了面对各类机关的应用需要,GOMS 采用多层 B/S 体系结构和 J2EE 技术实现了系统的分布异构及跨平台,支持流行操作系统、Web 服务器以及数据库管理系统。

2. 需求重用

在产品领域定位的指导下,经过深入的分析调研,设计师发现所有的事务型系统都有一个共同的特征:工作流程。在 ISO 9000 中也规定任何组织的事务处理必须有标准、规范的工作流程。在系统分析时,可以将这些业务工作流程抽象出来,如公文管理中的收文流程、发文流程、归档流程、稽催流程、档案管理流程等。而且,在非公文管理的其他业务中也可以抽取流程,如车辆管理业务流程、会议室管理流程、请假加班流程等。因此,可以建立一个工作流(workflow)平台,使所有的业务流程只要在工作流平台中进行定义就可以运作,从而实现"零代码编写的理想目标"。

一般的应用软件产品除了完成业务所需要的功能外,还必须有一些支持模块(module),以支持系统的正常运行。这些模块通常包括组织管理模块和系统支持模块。组织管理是机关业务得以正常运作的基础,这对于每一个电子政务领域内的应用系统来说都必不可少。通常系统支持模块是为了软件系统的正常运作所提供的必不可少的功能,如系统权限管理、日志管理、数据库备份/恢复功能等都属于此类。

对于软件企业来说,保持系列产品在风格上的一致性是非常重要的。它不但可以减少系列产品的广告费用,降低系列产品的维护、培训费用,而且还可以在软件开发时进行界面风格重用,降低软件开发成本。

3. 设计重用

在产品开发之初,被识别的所有的业务流程都可以运行于工作流平台之上。因此,在产品设计时,采用了以工作流平台为核心的领域软件产品设计框架,如图 1-5 所示。

该工作流平台向产品最终用户提供流程自定义工具,使用户无须编程就可以自定义出所需要的工作业务流程,并可对流程流转过程进行实时监控。而且,还向软件开发人员提供快速应用开发工具以及 API(Application Programming Interface,应用编程接口),使开发人员只要调用该工作流平台 API 就可以实现复杂流程业务程序。

在选好系统领域框架和统一开发方法后,系统构件的开发就应充分利用已有框架所提供的服务和工具,并力求实现大粒度构件重用。通过系统构件的分层,将频繁变动的业务逻

辑层分离出来，实现通用类构件的完全重用。在各个模块之间设计统一的接口，当某一模块业务逻辑改变时，使系统之间的影响最小。系统实现即插即用(plug and play)，容易升级。为此，设计师采用了层次式软件体系结构(见 3.1.4 节)，将产品的系统构件模型定义为 4 个层次，如图 1-6 所示。

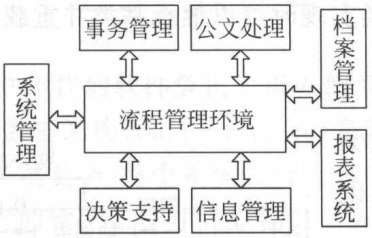


图 1-5　领域软件产品设计框架

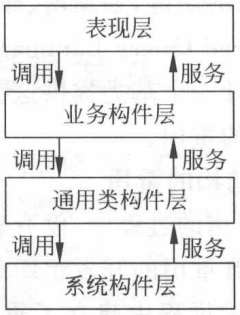


图 1-6　层次式体系结构

(1) 系统构件层。系统构件层指系统开发平台本身所提供的类库，包括 Java JDK 类库等。

(2) 通用类构件层。通用类构件层是产品重用的核心，它不但能实现产品的纵向重用，而且还可以实现系列产品的横向重用。这一层主要包含工作流平台核心模块、组织管理模块、系统管理模块、页面风格函数以及 JSP 的 CSS、JS 等字符串处理、数据库连接、通用打印和查询、权限验证和日期处理等与业务逻辑无关的类。

(3) 业务构件层。业务构件层指为了满足各个不同业务的需要而设计的软件包，并在业务软件包中设置明确的接口，方便业务之间的交互，并可以实现系列产品之间的大粒度构件重用。

(4) 表现层。表现层主要采用 JSP、Serverlet 页面来展现业务流程界面。在表现层中，JSP 只调用 JavaBean 业务逻辑接口方法实现业务逻辑的处理，作为用户与系统交互的接口，而不涉及任何业务逻辑。

GOMS 系统在层次式软件体系结构的基础上，利用面向对象的继承、封装(encapsulation)、多态(polymorphism)等特性，上层能够继承下层的所有功能，并可进行屏蔽、修改和扩充，从而实现功能的逐层扩展。通过抽象，能够将系统的大多数公共操作和通用的数据库表结构提取出来，实现一个 GOMS 系统的基本操作库(基本类库)。通过封装，能够将完成某种功能的一系列操作和数据结构封装在一个模块中，隐藏内部的具体实现过程。通过继承和重载(overloading)，后代不但能够方便地获得、扩充或者修改祖先的功能，而且还可以达到通过少量修改下层的方法来实现软件的可扩展，从而解决管理流程不断变化、软件难以适应的问题。

4. 代码重用

在采用上述可重用的分析、设计方法后，系统的实现(implementation)将变得相对容易。在各代码段的实现时，只需要调用明确的接口，就可以实现处理功能，降低算法的复杂度，大大提高编码(coding)的效率及程序的可维护性。在编码时，主要采用 JavaBean 和 JSP 技术实现业务实体逻辑和用户系统操作接口。在 JavaBean 中充分采用 Java 的优秀的面向

对象机制,实现业务实体类的处理逻辑,并尽可能地达到JavaBean方法的"一次编写,处处运行"的目标。在编写JSP时,完全引用了业务逻辑层的JavaBean,从而使JSP的页面编写变得简单,并实现了业务逻辑的封装性,提高了JSP程序代码的安全性。

另外,在编码过程中的一个重要作用是算法的重用。由于在类设计时基本上每一个类都提供了相似的功能,如新增、删除、修改、查询,而这些操作的算法基本上一致,差别只在于SQL(Structured Query Language,结构化查询语言)语句的具体细节上,所以在设计编码时,可以先设计一个基类提供这些功能,在其他类实现时可以继承基类并重载或应用这些方法,实现算法的重用。

5. 组织结构的重用

在软件重用的过程中,仅有软件重用方法是不够的,还必须有重用的开发组织结构予以支持。因此,在产品开发过程中建立了重用的组织框构,主要由三组成员组成:构件开发组、构件应用组和协调组。这三个组的关系如图1-7所示。

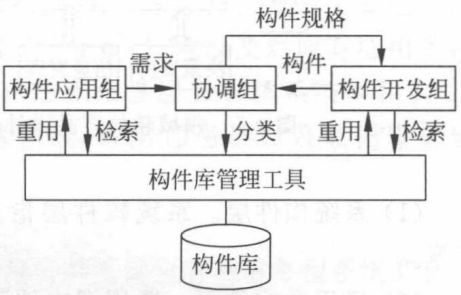


图1-7 组织结构关系

(1) 构件开发组。构件开发组负责从协调组接收构件设计规格说明,进行构件的设计和实现,把实现了的构件交给协调组。构件开发组接收到构件设计规格说明后,先在构件库中检索是否已经存在满足规格说明的构件,或者相接近/类似的构件;如果存在这样的构件,则可以直接进行重用,或者扩展已有功能以满足规格说明的需要。

(2) 构件应用组。构件应用组负责业务逻辑的设计与实现,在开发过程中查询构件库,从中检索可重用构件,使用可重用构件进行业务逻辑的快速实现。如果构件库中没有需要的构件,则把构件的需求交给协调组。

(3) 协调组。协调组在构件开发组和构件应用组之间起协调作用,负责接收并分析来自构件应用组的构件需求,对构件是自行开发还是从其他途径获取进行决策。如果决定自行开发,就形成构件设计规格说明,交给构件开发组。同时,协调组还负责构件的资格确认、质量控制、分类和存储工作。

6. 构件库

在软件重用过程当中,建立了构件库以进行可重用构件的推广和使用。在构件创建完成后,协调组将构件分类存放于产品项目工程中,提供给应用组使用,并用开发工具生成文档存放于配置管理工具的构件文档库中,以便应用时查询,同时通知构件需求提供者。

开发工程师采用超文本方法组织构件库,所有构件都辅以详细的功能和行为说明文档,说明中出现的构件以网状链接方式相互连接。应用组成员在阅读文档的过程中可按照人类的联想思维方式任意跳转到包含相关构件的文档。此外,还提供全文检索功能,将用户(构件开发组/应用组成员)给出的关键字与说明文档中的文字进行匹配,实现构件的浏览式检索。

为了避免用户在跳转过程中迷失方向,系统通过图形用户界面提供浏览历史信息图,允许将特定画面定义为命名"书签"并随时跳转至"书签",帮助用户逆跳转路径而逐步返回。

7. 小结

在通用办公管理系统及电子政务产品开发过程中，实现了产品领域横向的重用和产品开发过程中的纵向层次结构的重用；在项目规划和整个软件开发过程中系统地采用了重用的策略，并建立了一整套重用管理机制，从而大大提高了软件产品的可重用性和软件生产率(productivity)，并为后继产品的开发提供了良好的可重用基础。

1.3 软件体系结构的兴起和发展

20世纪60年代的软件危机使得人们开始重视软件工程的研究。起初，人们把软件设计的重点放在数据结构(data structure)和算法(algorithmic)的选择上，随着软件系统规模越来越大、越来越复杂，整个系统的结构和规格说明显得越来越重要。随着软件危机的程度日益加剧，现有的软件工程方法显得力不从心。对于大规模的复杂软件系统来说，对总体的系统结构设计和规格说明比起对计算的算法和数据结构的选择已经变得明显重要得多。在此种背景下，人们认识到软件体系结构(software architecture)的重要性，并认为对软件体系结构的系统地、深入地研究将会成为提高软件生产率和解决软件维护问题的新的最有希望的途径。

自从软件系统首次被分成许多模块，模块之间有相互作用，组合起来有整体的属性，就具有了体系结构。优秀的开发者常常会使用一些体系结构模式(architecture pattern)作为软件系统结构设计策略，但他们并没有规范地、明确地表达出来，这样就无法将他们的知识与别人交流。软件体系结构是设计抽象的进一步发展，满足了更好地理解软件系统，更方便地开发更大、更复杂的软件系统的需要。

事实上，软件总是有体系结构的，不存在没有体系结构的软件。“体系结构”一词在英文里是“建筑”的意思。把软件比作一座楼房，从整体上讲，是因为它有基础、主体和装饰，即操作系统之上的基础设施软件、实现计算逻辑的主体应用程序和方便使用的用户界面程序；从细节上来看，每一个程序也是有结构的。早期的结构化程序就是以语句组成模块，模块的聚集和嵌套形成层层调用的程序结构，也就是体系结构。结构化程序的程序(表达)结构和(计算的)逻辑结构的一致性及自顶向下开发方法自然而然地形成了体系结构。由于结构化程序设计时代程序规模不大，通过强调结构化程序设计方法学，自顶向下、逐步求精，并注意模块的耦合性就可以得到相对良好的结构，所以，并未特别研究软件体系结构。

可以做个简单的比喻，结构化程序设计时代是以砖、瓦、灰、沙、石、预制梁、柱、屋面板盖平房和小楼；而面向对象时代以整面墙、整间房、一层楼梯的预制件来盖高楼大厦。构件怎样搭配才合理？体系结构怎样构造容易？重要构件有了更改后，如何保证整栋高楼不倒？每种应用领域(医院、工厂、旅馆)需要什么构件？有哪些实用、美观、强度、造价合理的构件骨架使建造出来的建筑(即体系结构)更能满足用户的需求？如同土木工程进入现代建筑学一样，软件也从传统的软件工程进入现代面向对象的软件工程，研究整个软件系统的体系结构，寻求建构最快、成本最低、质量最好的构造过程。

软件体系结构虽脱胎于软件工程，但其形成同时借鉴了计算机体系结构和网络体系结构中很多宝贵的思想和方法。最近几年软件体系结构研究已完全独立于软件工程的研究，成为计算机科学的一个最新的研究方向和独立学科分支。软件体系结构研究的主要内容涉

及软件体系结构描述、软件体系结构风格、软件体系结构评价和软件体系结构的形式化方法等。解决好软件的重用、质量和维护问题,是研究软件体系结构的根本目的。

1.3.1 软件体系结构的定义

虽然软件体系结构已经在软件工程领域中有着广泛的应用,但迄今为止还没有一个公认的定义。许多专家从不同角度和不同侧面对软件体系结构进行了刻画,较为典型的定义有以下几个。

(1) Dewayne Perry 和 Alexander Wolf 曾这样定义:软件体系结构是具有一定形式的结构化元素(element),即构件的集合,包括处理构件、数据构件和连接构件。处理构件负责对数据进行加工,数据构件是被加工的信息,连接构件把体系结构的不同部分组合连接起来。这一定义注重区分处理构件、数据构件和连接构件,这一方法在其他的定义和方法中基本上得到了保持。

(2) Mary Shaw 和 David Garlan 认为,软件体系结构是软件设计过程中的一个层次,这一层次超越计算过程中的算法设计和数据结构设计。体系结构问题包括总体组织和全局控制,通信协议(protocol),同步,数据存取,给设计元素分配特定功能,设计元素的组织,规模和性能,在各设计方案间进行选择等。软件体系结构处理算法与数据结构之上关于整体系统结构设计和描述方面的一些问题,如全局组织和全局控制结构,关于通信、同步与数据存取的协议,设计构件功能定义,物理分布与合成,设计方案的选择、评估(evaluation)与实现等。

(3) Kruchten 指出,软件体系结构有 4 个角度,它们从不同方面对系统进行描述:概念(concept)角度描述系统的主要构件及它们之间的关系;模块角度包含功能分解与层次结构;运行角度描述了一个系统的动态结构;代码角度描述了各种代码和库函数在开发环境中的组织。

(4) Hayes Roth 认为,软件体系结构是一个抽象的系统规范,主要包括用其行为来描述的功能构件和构件之间的相互连接、接口和关系。

(5) David Garlan 和 Dewne Perry 于 1995 年在 IEEE(Institute of Electrical and Electronics Engineers,国际电气和电子工程师协会)软件工程学报上采用如下的定义:软件体系结构是一个程序/系统各构件的结构、它们之间的相互关系以及进行设计的原则和随时间演化(evolution)的指导方针。

(6) Barry Boehm 和他的学生提出,一个软件体系结构包括一个软件和系统构件,互联及约束的集合;一个系统需求说明的集合;一个基本原理用以说明这一构件,互联和约束能够满足系统需求。

(7) 1997 年,Bass、Ctements 和 Kazman 在《使用软件体系结构》一书中给出如下的定义:一个程序或计算机系统的软件体系结构包括一个或一组软件构件、软件构件的外部的可见特性及其相互关系。其中,"软件外部的可见特性"指软件构件提供的服务、性能、特性、错误处理、共享资源使用等。

总之,软件体系结构的研究正在发展,软件体系结构的定义也必然随之完善。在本书中,如果不特别指出,将使用软件体系结构的下列定义:

软件体系结构为软件系统提供了一个结构、行为和属性的高级抽象,由构成系统的元素

的描述、这些元素的相互作用、指导元素集成的模式以及这些模式的约束组成。软件体系结构不仅指定了系统的组织(organization)结构和拓扑(topology)结构,并且显示了系统需求和构成系统的元素之间的对应关系,提供了一些设计决策的基本原理。

1.3.2 软件体系结构的意义

以现代建筑为例,不管建多高的楼房,都是先用钢筋和水泥搭起框架(在建筑业中称之为"框剪结构"),在此结构中再用各种砖砌墙。这样的楼房,在建好后,业主在装修时可以敲掉任何一堵非承重墙,也可以在任何地方加一堵新墙,都不会影响整栋楼的质量。

如果把软件系统看作一栋楼,则软件体系结构就是这栋楼的框剪结构。换句话说,软件体系结构是整个软件系统的骨架,在软件开发中起着非常重要的作用。

1. 体系结构是风险承担者进行交流的手段

软件体系结构代表了系统的公共的高层次的抽象。这样,系统的大部分有关人员能把它作为一个互相理解的基础,形成统一认识,互相交流。

软件系统的风险承担者(客户、用户、项目管理人员、设计开发人员以及测试人员等)关心着系统的不同方面,而这些方面都受体系结构的影响,所以体系结构可能是大家都关心的一个重要因素(即使是从不同愿望出发)。例如,用户关心系统是否满足可用性和可靠性需求;客户关心的是系统能否在规定时间内完成,并且开支在预算范围内;管理人员担心在经费支出和进度条件下,按此体系结构能否使开发组成员在一定程度上独立开发,各部分的交互是否遵循统一的规范,开发进度是否可控;开发人员关心的是如何才能实现体系结构的各项目标。

总之,体系结构提供了一种共同语言来表达各种关注和协商,进而对大型复杂系统能进行理智的管理。这对项目最终的质量和使用有极大的影响。

2. 体系结构是早期设计决策的体现

软件体系结构体现了系统的最早的一组设计决策(decision),这些早期的约束比起以后的开发、设计、编码或运行服务及维护阶段的工作重要得多,对系统生命周期(life cycle)的影响也大得多。早期决策的正确性最难以保证,而且这些决策也最难以改变,影响范围也最大。

1) 软件体系结构明确了对系统实现的约束条件

所谓"实现",就是要用实体来显示出一个软件体系结构,即要符合体系结构所描述的结构性设计决策,分割成规定的构件,按规定方式互相交互。如果系统实现严格遵循体系结构设计中所做出的关于系统结构的决策,则系统实现将能够体现出体系结构的特色。也就是说,在具体实现时必须按照体系结构的设计,将系统分成若干组成部分,各部分必须按照预定的方式进行交互,而且每个部分也必须具有体系结构中所规定的外部特征。

这些约束是在系统级或项目范围内做出的,每个构件上工作的实现者是看不见的。这样一来,可以分离着重点,软件体系结构的设计者不必是算法设计者或精通编程语言,只需重点考虑系统的总体权衡(tradeoff)问题;而构件的开发人员在体系结构给定的约束下进行开发。

2) 软件体系结构决定了开发和维护组织的组织结构

体系结构不仅规定了所开发软件的系统结构,还影响着项目开发组织的结构。开发一

个大型系统时,常见的任务划分方法是系统的任务划分结构,即将系统的不同部分交由不同的小组开发完成。体系结构中包含对系统的最高层次的分解,因而一般被作为任务划分结构的基础。

任务划分结构又规定了计划、调度及预算的单位,决定了组内交流的渠道、配置控制和文件系统的组织、集成(integration)与测试计划和过程,甚至对电子公告牌的组织、开发小组出去野餐的次数这样的琐碎细节也提出了要求。各开发小组按照体系结构中对各主要构件接口的规定进行交流。一旦进入维护阶段,维护活动也会反映出软件的结构,常由不同的小组分别负责对各具体部分的维护。

3) 软件体系结构制约着系统的质量属性

一个大型系统能否具有所期望的质量属性,主要是由确定体系结构的时间决定的。小的软件系统可以通过编程或调试措施来达到质量属性的要求,而随着软件系统规模的扩大,这种技巧也将越来越无法满足要求。这是因为,在大型软件系统中,质量属性更多的是由系统结构和功能划分来实现的,而不再主要依靠所选用的算法或数据结构。

系统的质量属性可以分为两类:第一类是可以通过运行软件并观察其效果来度量的,如功能、性能、安全性及可靠性等;第二类是指那些无法通过观察系统来度量,只能通过观察开发活动或维护活动来考察的特性,包括各种可维护性问题,如可适应性、可移植性、可重用性等(例如,可重用性依赖于系统中的构件与其他构件的联系情况)。必须认识到:软件体系结构并不能单独保证系统所要求的功能与质量。低劣的下游设计及实现都会破坏一个体系结构的构架。好的软件体系结构是必要条件,但不是成功的充分条件。

4) 通过研究软件体系结构可能预测软件的质量

能否在系统开发或使用之前就确知在体系结构上做了哪些合理的决策(例如,系统是否将表现出所期望的质量属性)?答案是否定的。从理论上来说,用某种方法选择体系结构和随机选择一个体系结构没有什么两样。然而,可以使用对体系结构的评价来预测系统未来的质量属性。一些体系结构评估技术(见第11章)可使设计师全面地对按某软件体系结构开发出来的软件产品的质量及缺陷做出比较准确的预测。

5) 软件体系结构使推理和控制更改更简单

按照经典软件工程理论,软件维护阶段所花费的成本占整个软件生命周期中的60%~80%。根据这一论断,许多(即使不是全部)程序员或设计师从来没有进行过新系统的开发,而仅仅是在已有代码的基础上工作。

在整个软件生命周期内,每个体系结构都将更改划分为三类:局部的、非局部的和体系结构级的变更。局部变更是最经常发生的,也是最容易进行的,只需修改某一个构件就可以实现。非局部变更的实现则需对多个构件进行修改,但并不改动体系结构。体系结构级的变更是指会影响各部分的相互关系,甚至要改动整个系统。所以,一个优秀的体系结构应该能使更改简单易行。

重要的是要确定何时必须更改,确定哪种更改方式的风险最小,评估更改的效果,以及合理安排所提出的更改的实施顺序和优先级等。所有这些都需要深入地洞察系统的各部分的关系、相互依赖关系、性能及行为特性。

6) 软件体系结构有助于循序渐进的原型设计

确定了软件体系结构,就可以对其进行分析,并将其按可执行模型来构造原型(prototype)。

从两个方面有助于开发活动的顺利进行，一是可以在软件生命周期的早期阶段明确潜在的系统性能问题，二是在软件生命周期的早期阶段就能得到一个可执行的系统。这两个方面都将减少项目开发的潜在风险。如果所用的体系结构是若干相关体系结构中的一个，则构建原型框架的代价就可以分摊到多个系统上。

7）软件体系结构可以作为培训的基础

在对项目组新成员介绍所开发的系统时，可以首先介绍系统的体系结构，以及对构件之间如何交互从而实现系统需求的高层次的描述，让项目新成员很快进入角色。

3. 软件体系结构是可传递和可重用的模型

软件体系结构体现了一个相对来说比较小又可理解的模型。

软件体系结构级的重用意味着体系结构的决策能在具有相似需求的多个系统中发生影响，这比代码级的重用要有更大的好处。通过对体系结构的抽象可以使设计师能够对一些经过实践证明是非常有效的体系结构构件进行重用，从而提高设计的效率和可靠性。在设计过程中，设计师常常会发现，对一个体系结构中的构件稍加抽象，就可以将它应用到其他设计中去，这样会大大降低设计的复杂度。例如，在某个设计中采用了管道与过滤器（pipe-filter）体系结构风格（见 3.1.1 节），当将系统映射到 UNIX 系统中时，就会发现 UNIX 系统已经为用户提供了功能丰富的管道与过滤器功能，这样，用户就可以充分利用 UNIX 系统提供的这些构件来简化设计和开发。

鉴于体系结构的重要性，Perry 将软件体系结构视为软件开发中第一类重要的设计对象，Barry Boehm 也明确指出："在没有设计出体系结构及其规则时，那么整个项目不能继续下去，而且体系结构应该看作软件开发中可交付的中间产品"。由此可见，体系结构在软件开发中为不同的人员提供了共同交流的语言，体现并尝试了系统早期的设计决策，并作为系统设计的抽象，为实现框架和构件的共享和重用、基于体系结构的软件开发提供了有力的支持。

1.3.3 软件体系结构的发展史

软件系统的规模在迅速增大的同时，软件开发方法也经历了一系列的变革。在此过程中，软件体系结构也由最初模糊的概念发展到一个渐趋成熟的理论和技术。

20 世纪 70 年代以前，尤其是在以 ALGOL 68 为代表的高级语言出现以前，软件开发基本上都是汇编程序设计，此阶段系统规模较小，很少明确考虑系统结构，一般不存在系统建模工作。20 世纪 70 年代中后期，由于结构化开发方法的出现与广泛应用，软件开发中出现了概要设计与详细设计，而且主要任务是数据流设计与控制流设计，因此，此时软件结构已作为一个明确的概念出现在系统的开发中。

20 世纪 80 年代初到 20 世纪 90 年代中期，是面向对象开发方法兴起与成熟阶段。由于对象是数据与基于数据之上操作的封装，因而在面向对象开发方法下，数据流设计与控制流设计则统一为对象建模，同时，面向对象方法还提出了一些其他的结构视图。例如，OMT（Object Modeling Technology，对象建模技术）方法提出了功能视图、对象视图与动态视图（包括状态图和事件追踪图）；Booch 方法提出了类视图、对象视图、状态迁移图、交互作用图、模块图、进程图；统一建模语言（Unified Modeling Language，UML）则通过功能模型（用例视图）、静态模型（包括类图、对象图、构件图、包图、组合结构图）、动态模型（通信图、顺序

图、定时图、状态图、活动图、交互概览图)、配置模型(制品图、配置图)描述应用系统的结构。

20世纪90年代以后则是基于构件的软件开发阶段,该阶段以过程为中心,强调软件开发采用构件化技术和体系结构技术,要求开发出的软件具备很强的自适应性、互操作性、可扩展性和可重用性。此阶段中,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,同时,软件体系结构作为一门学科逐渐得到人们的重视,并成为软件工程领域的研究热点。

纵观软件体系结构技术的发展过程,从最初的"无结构"设计到现行的基于体系结构的软件开发,可以认为经历了以下4个阶段。

(1) 无体系结构设计阶段。以汇编语言进行小规模应用程序开发为特征。

(2) 萌芽阶段。出现了程序结构设计主题,以控制流图和数据流图构成软件结构为特征。

(3) 初期阶段。出现了从不同侧面描述系统的结构模型,以UML为典型代表。

(4) 高级阶段。以描述系统的高层抽象结构为中心,不关心具体的建模细节,划分了体系结构模型与传统软件结构的界限,该阶段以Kruchten提出的"4+1"模型为标志。

1.4 软件体系结构的应用现状

目前,软件体系结构领域研究非常活跃,例如,南加州大学专门成立了软件体系结构研究组,曼彻斯特大学专门成立了软件体系结构研究所。同时,业界许多著名企业的研究中心也将软件体系结构作为重要的研究内容。

归纳现有体系结构的研究活动,主要包括如下几个方面。

1. 软件体系结构描述语言

在提高软件工程师对软件系统的描述和理解能力中,虽然软件体系结构描述起着重要作用,但这些抽象的描述通常是非形式化的和随意的。体系结构设计经常难以理解,难以适于进行形式化分析和模拟,缺乏相应的支持工具帮助设计师完成设计工作。为了解决这个问题,用于描述和推理的形式化语言得以发展,这些语言叫作体系结构描述语言(Architecture Description Language,ADL),ADL寻求增加软件体系结构设计的可理解性和重用性。

ADL是这样一种语言,系统设计师可以利用它所提供的特性进行软件系统概念体系结构建模。ADL提供了具体的语法与刻画体系结构的概念框架。ADL使得系统开发者能够很好地描述他们设计的体系结构,以便与他人交流,能够用提供的工具对许多实例进行分析。

这种描述语言的目的就是提供一种规范化的体系结构描述,从而使得体系结构的自动化分析变得可能。研究人员已经提出了若干适用于特定领域的ADL,典型的有C2、Wright、Aesop、Unicon、Rapide、SADL、MetaH、Weaves等。Shaw和Garlan指出,一个好的ADL的框架应具备组装性、抽象性、重用性、可配置性、异构性、可分析性等几个方面的特点。在此基础上,Medividovic提出了一种ADL的分类和比较框架,详细分析了多种典型的ADL的优点与不足,对当前ADL研究做了比较全面的总结,并为将来的ADL的开发提供了很有价值的参考建议。

在 ADL 研究方面，国内的一些学者也相应提出了几种比较有特色的 ADL，如北京大学杨芙清院士等提出的 JB/SADL、中国科学院软件所唐稚松院士等提出的 XYX/ADL、吉林大学金淳兆教授等提出的 FRADL 和 Tracer、东北大学刘积仁教授等提出的 A-ADL、北京邮电大学艾波教授等提出的 XYZ/SAE，等。

有关 ADL 的详细内容，将在第 4 章中进行介绍。

2. 体系结构描述构造与表示

按照一定的描述方法，用体系结构描述语言对体系结构进行说明的结果称为体系结构的表示，而将描述体系结构的过程称为体系结构构造。在体系结构描述方面，Kruchten 提出的“4＋1”模型是当前软件体系结构描述的一个经典范例，该模型由逻辑视图、开发视图、过程视图和物理视图组成，并通过场景将这 4 个视图有机地结合起来，比较细致地描述了需求和体系结构之间的关系（见 2.1 节）。

而 Booch 从 UML 的角度给出了一种由设计视图、过程视图、实现视图和部署视图，再加上一个用例视图构成的体系结构描述模型。Medividovic 则总结了用 UML 描述体系结构的三种途径：不改变 UML 用法而直接对体系结构建模；利用 UML 支持的扩充机制扩展 UML 的元模型实现对体系结构建模概念的支持；对 UML 进行扩充，增加体系结构建模元素。我国电子科技大学的于卫等研究了其中的第二种方案，其主要思路是提炼 5 个软件体系结构的核心部件，利用 UML 扩充机制中的一种，给出了相应的 OCL（Object Constraint Language，对象约束语言）约束规则的描述，并且给出了描述这些元素之间的关系的模型。

IEEE 于 1995 年成立了体系结构工作组，综合了体系结构描述研究成果，并参考业界的体系结构描述的实践，起草了体系结构描述框架标准 IEEE P1471。

Rational 从资产重用的角度提出了体系结构描述的规格说明框架（architecture description specification），该建议草案已经提交给 OMG，可望成为体系结构描述的规范。IEEE P1471 和 Rational 的 ADS，都提出了体系结构视点（viewpoint）的概念，并从多个视点描述体系结构的框架。但问题在于：一个体系结构应该从哪几个视点进行考虑？每个视点由哪些视构成？各种视点应当使用哪种体系结构描述语言，以及采用哪些体系结构建模和分析技术等问题都未解决。

综上所述，虽然 UML 作为一个工业化标准的可视化建模语言，支持多角度、多层次、多方面的建模需求，支持扩展，并有强大的工具支持，确实是一种可选的体系结构描述语言，但是根据 Medividovic 给出的体系结构语言的框架，UML 不属于体系结构描述语言的范畴。事实上，判断一种语言是否适合用作体系结构描述语言的关键在于，它能否表达体系结构描述语言应该表达的概念与抽象，如果需要转化，其复杂性如何？

3. 体系结构分析、设计与验证

体系结构是对系统的高层抽象，并只对感兴趣的属性进行建模。由于体系结构是在软件开发过程之初产生的，因此设计好的体系结构可以减少和避免软件错误的产生和维护阶段的高昂代价。体系结构是系统集成的蓝本、系统验收的依据，体系结构本身需要分析与测试，以确定这样的体系结构是否满足需求。体系结构分析的内容可分为结构分析、功能分析和非功能分析。在进行非功能分析时，可以采用定量分析方法与推断的分析方法。在非功能分析的途径上，则可以采用单个体系结构分析与体系结构比较的分析方法。Kazman 等提出了一种非功能分析的体系结构分析方法（见第 11 章），并运用场景技术，提出了基于场

景的体系结构分析方法。Barbacci 等提出了多质量属性情况下的体系结构质量模型、分析与权衡方法(见第 11 章)。

生成一个满足软件需求的体系结构的过程即为体系结构设计。体系结构设计过程的本质在于:将系统分解成相应的组成成分(如构件、连接件),并将这些成分重新组装成一个系统。具体说来,体系结构设计有两大类方法:过程驱动方法和问题列表驱动方法。前者包括:

(1) 面向对象方法,与 OOA/OOD 相似,但更侧重接口与交互。

(2) "4+1"模型方法。

(3) 基于场景的迭代方法。

应该说,基于过程驱动的体系结构设计方法适用范围广,易于裁减,具备动态特点,通用性与实践性强。而问题列表驱动法的基本思想是枚举设计空间,并考虑设计维的相关性,以此来选择体系结构的风格(style)。显然,该方法适用于特定领域,是静态的,并可以实现量化体系结构设计空间。例如,Allen 博士的论文专门研究了用户界面类的量化设计空间,提出了 19 个功能维,26 个结构维,622 条设计规则。

体系结构设计研究的重点内容之一就是体系结构风格或模式,体系结构模式在本质上反映了一些特定的元素按照特定的方式组成一个特定的结构,该结构应有利于上下文环境(context)下的特定问题的解决。体系结构模式分为两个大类:固定术语和参考模型。已知的固定术语类的体系结构模型包括管道过滤器、客户-服务器、面向对象、黑板、分层、对等模式(基于事件调用方法,隐式调用,基于推理模式)、状态转换、一些派生的固定术语类的体系结构模式,包括 Gen Voca、C2、REST 等;而参考模型则相对较多,常常与特定领域相关,如编译器的顺序参考模型和并行参考模型、信息系统的参考模型、航空模拟环境系统的参考模型等。国内学者在这方面也做了不少工作,如北京大学杨芙清院士等提出的 JB/HBM 风格。

体系结构测试着重于仿真系统模型,解决体系结构层的主要问题。由于测试的抽象层次不同,体系结构测试策略可以分为单元、子系统、集成、验收测试等阶段的测试策略。在体系结构集成测试阶段,Debra 等提出了一组针对体系结构的测试覆盖标准,Paola Inveradi 提出了一种基于 CHAM 的体系结构语义验证技术。

应该说,体系结构分析、设计和验证已经取得了很丰富的研究成果。但是这些方法存在着一个普遍缺点:可操作性差,难于实用化,因此并没有取得很好的实践效果。

4. 体系结构发现、演化与重用

体系结构发现解决如何从已经存在的系统中提取软件的体系结构,属于逆向工程范畴。Waters 等提出了一种迭代(iteration)式体系结构发现过程,即由不同的人员对系统进行描述,然后对这些描述进行分类并融合,发现并解除冲突,将体系结构新属性加入到已有的体系结构模型中,并重复该过程直至体系结构描述充分。

由于系统需求、技术、环境、分布等因素的变化而最终导致软件体系结构的变动,称为软件体系结构演化。软件系统在运行时刻的体系结构变化称为体系结构的动态性,而将体系结构的静态修改称为体系结构扩展。体系结构扩展与体系结构动态性都是体系结构适应性和演化性的研究范畴。可以用多值代数或图重写理论来解释软件体系结构的演化。Esteban 等通过研究系统的动态可配置特性,提出了电信软件体系结构动态修改的方案。

体系结构的动态性分为有约束的和无约束的以及结构动态性和语义动态性。Darwin 和 C2 都直接支持结构动态性，而 CHAM、Wright 和 Rapide 支持语义动态性。在 C2 中定义有专门支持体系结构修改的描述语言 AML，而 Darwin 对体系结构的修改则采用相应的脚本语言，CHAM 是通过多值演算实现系统体系结构的变换，Wright 通过顺序通信进程 CSP 描述构件的交互语义。

体系结构重用属于设计重用，比代码重用更抽象。一般认为易于重用的标准包括：领域易于理解，变化相对较慢，内部有构件标准，与已存在的基础设施兼容，在大规模系统开发时体现规模效应。由于软件体系结构是系统的高层抽象，反映了系统的主要组成元素及其交互关系，因而较算法更稳定，更适合于重用。鉴于软件体系结构是应大系统开发和软件产品线(software product line)技术而出现的，在其二者之间，笔者认为：产品线中的体系结构重用将更有现实意义，并具有更大的相似性。体系结构模式就是体系结构重用研究的一个成果，而体系结构参考模型则是特定域软件体系结构的重用的成熟的象征。Katwijk 等采用扩展数据流技术 EDFG 实现了系统与构件的构造过程，得出了相应的体系结构是易于重用的结论。

总之，重用技术作为软件工程领域倡导的有效技术之一，在基于构件与体系结构的软件开发时代，软件体系结构重用将是一个重要的主题。

5. 基于体系结构的软件开发方法

本质上，软件体系结构是对软件需求的一种抽象解决方案。在引入了体系结构的软件开发之后，应用系统的构造过程变为“问题定义→软件需求→软件体系结构→软件设计→软件实现”，可以认为软件体系结构架起了软件需求与软件设计之间的一座桥梁。而在由软件体系结构到实现的过程中，借助一定的中间件(middleware)技术与软件总线(bus)技术，软件体系结构将易于映射成相应的实现。Bass 等提出了一种基于体系结构的软件开发过程。国内学者在这方面也做了不少的工作，如清华大学车敦仁教授等提出了基于体系结构的应用平台及框架仓库技术，北京邮电大学艾波教授等提出了基于体系结构的开发模型中软件体系结构的生命周期模型，北京航空航天大学陶伟博士等提出了一种以 6 个体系结构视图为中心的软件开发方式。

软件开发模型是跨越整个软件生存周期的系统开发、运行、维护所实施的全部工作和任务的结构框架，给出了软件开发活动各阶段之间的关系。目前，常见的软件开发模型大致可分为如下 3 种类型。

(1) 以软件需求完全确定为前提的瀑布模型。

(2) 在软件开发初始阶段只能提供基本需求时采用的渐进式开发模型，如螺旋模型、增量模型、敏捷方法、统一过程等。

(3) 以形式化开发方法为基础的变换模型。

所有开发方法都是要解决需求与实现之间的差距。但是，这三种类型的软件开发模型都存在这样或那样的缺陷，不能很好地支持基于软件体系结构的开发过程。因此，研究人员在发展基于体系结构的软件开发模型方面做了一定的工作。

在基于构件和基于体系结构的软件开发逐渐成为主流开发方法的情况下，已经出现了基于构件的软件工程。但是，对体系结构的描述、表示、设计和分析以及验证等内容的研究还相对不足。随着需求复杂化及其演化，切实可行的体系结构设计规则与方法将更为重要。

第12章将讨论基于体系结构的软件开发方法。

6. 特定领域的体系结构框架

特定领域的应用具有相似的特征,因而经过严格设计,并将直觉的成分减少到最小程度,可以有效地实现重用,并可借鉴领域中已经成熟的体系结构。Rick Hayers-Roth 和 Will Tracz 分别对特定领域的体系结构(Domain Specific Software Architecture,DSSA)给出了不同的定义,前者侧重于 DSSA 的特征,强调系统由构件组成,适用于特定领域,有利于开发成功应用程序的标准结构;后者侧重于 DSSA 的组成要素,指出 DSSA 应该包括领域模型、参考需求、参考体系结构、相应的支持环境或设施、实例化、细化或评估的方法与过程。两种 DSSA 定义都强调了参考体系结构的重要性。

特定领域的体系结构是将体系结构理论应用到具体领域的过程,常见的 DSSA 有:CASE 体系结构、CAD 软件的参考模型、信息系统的参考体系结构、网络体系结构 DSSA、机场信息系统的体系结构和信息处理 DSSA 等。国内学者提出的 DSSA 有:北京邮电大学周莹新博士提出的电信软件的体系结构,北京航空航天大学金茂忠教授等提出的测试环境的体系结构等。

软件体系结构充当一个理解系统构件和它们之间关系的框架,特别是那些始终跨越时间和实现的属性。这个理解对于现在系统的分析和未来系统的综合很有必要。在分析和支持下,体系结构抓住领域知识和实际的一致,促进设计的评估和构件的实施,减少仿真和构造原型。在综合的支持下,体系结构提供了建立系列产品的基础,以可预测的方式利用领域知识构造和维护模块、子系统和系统。

3.10 节将详细讨论 DSSA 相关技术。

7. 软件体系结构支持工具

几乎每种体系结构都有相应的支持工具,如 Unicon、Aesop 等体系结构支持环境,C2 的支持环境 ArchStudio,支持主动连接件的 Tracer 工具等。另外,还有支持体系结构分析的工具,如支持静态分析的工具、支持类型检查的工具、支持体系结构层次依赖分析的工具、支持体系结构动态特性仿真工具、体系结构性能仿真工具等。与其他成熟的软件工程环境相比,体系结构设计的支持工具还很不成熟,难以实用化。

8. 软件产品线体系结构

软件体系结构的开发是大型软件系统开发的关键环节。体系结构在软件产品线的开发中具有至关重要的作用,在这种开发生产中,基于同一个软件体系结构,可以创建具有不同功能的多个系统。在软件产品族之间共享体系结构和一组可重用的构件,可以降低开发和维护成本。

一个产品线代表一组具有公共的系统需求集的软件系统,它们都是根据基本的用户需求对标准的产品线构架进行定制,将可重用构件与系统独有的部分集成而得到的。采用软件生产线模式进行软件开发,将产生巨型软件企业。但目前开发的软件产品族大部分是处于同一领域的。

软件产品线是一个十分适合专业的软件开发组织的软件开发方法,能有效地提高软件生产率和质量、缩短开发时间、降低总开发成本。软件体系结构有利于形成完整的软件产品线。

第13章将详细讨论软件产品线的有关知识。

9. 建立评价软件体系结构的方法

软件体系结构的设计是整个软件开发过程中关键的一步。对于当今世界上庞大而复杂的系统来说，没有一个合适的体系结构而要设计一个成功的软件几乎是不可想象的。不同类型的系统需要不同的体系结构，甚至一个系统的不同子系统也需要不同的体系结构。体系结构的选择往往会成为一个系统设计成功的关键。

但是，怎样才能知道为软件系统所选用的体系结构是恰当的呢？如何确保按照所选用的体系结构能顺利地开发出成功的软件产品呢？要回答这些问题并不容易，因为它受到很多因素的影响，需要专门的方法对其进行评估。目前，常用的软件体系结构评估方法主要有：体系结构权衡分析方法和软件体系结构分析方法。

有关软件体系结构的评估方法，将在第 11 章进行详细讨论。

思 考 题

1. 简要概述什么是软件危机，产生软件危机的原因，以及如何克服软件危机。
2. 什么是软件重用？软件重用的层次可以分为哪几个级别？
3. 什么是可重用构件？相对于普通的软件产品，对可重用构件有什么特殊要求？
4. 基于构件的软件开发的优势是什么？基于构件的软件开发面临哪些挑战和困难？
5. 简述三种应用最为广泛的构件技术规范：COM、CORBA 和 EJB 的各自特点。
6. 基于构件的软件开发方法能够有效地提高开发效率，请简述在实际项目中，有哪些方式获取构件，如何管理以及重用构件。
7. 软件体系结构的定义众多，你是如何理解软件体系结构的？软件体系结构在软件系统中有什么作用？
8. 程序结构、软件结构和软件体系结构三者有什么区别和联系？

参考文献

第2章 软件体系结构建模

研究软件体系结构的首要问题是如何表示软件体系结构，即如何对软件体系结构建模。根据建模的侧重点不同，可以将软件体系结构的模型分为5种：结构模型、框架模型、动态模型、过程模型和功能模型。在这5个模型中，最常用的是结构模型和动态模型。

(1) 结构模型。这是一个最直观、最普遍的建模方法。这种方法以体系结构的构件、连接件(connector)和其他概念来刻画结构，并力图通过结构来反映系统的重要语义内容，包括系统的配置、约束、隐含的假设条件、风格、性质等。研究结构模型的核心是体系结构描述语言。

(2) 框架模型。框架模型与结构模型类似，但它不太侧重描述结构的细节而更侧重于整体的结构。框架模型主要以一些特殊的问题为目标，建立只针对和适应该问题的结构。

(3) 动态模型。动态模型是对结构或框架模型的补充，研究系统的“大颗粒”的行为性质。例如，描述系统的重新配置或演化。动态可以指系统总体结构的配置、建立或拆除通信通道或计算的过程。这类系统通常是激励型的。

(4) 过程模型。过程模型研究构造系统的步骤和过程，因而结构是遵循某些过程脚本的结果。

(5) 功能模型。功能模型认为体系结构是由一组功能构件按层次组成，下层向上层提供服务。它可以看作一种特殊的框架模型。

2.1 “4+1”视图模型

软件体系结构的5种模型各有所长，将5种模型有机地组合在一起，形成一个完整的模型来刻画软件体系结构更合适。例如，Kruchten在1995年提出了一个“4+1”视图模型。“4+1”视图模型从5个不同的视角(逻辑视图、进程视图、物理视图、开发视图和场景视图)来描述软件体系结构。每一个视图只反映系统的一个侧面，5个视图结合在一起才能反映系统的软件体系结构的全部内容。“4+1”视图模型如图2-1所示。

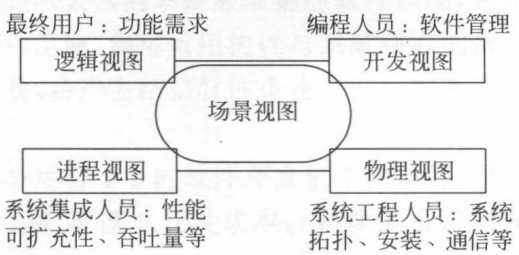


图2-1 “4+1”视图模型

2.1.1 逻辑视图

逻辑视图(logic view)主要支持系统的功能需求,即系统提供给最终用户的服务。在逻辑视图中,系统分解成一系列的功能抽象,这些抽象主要来自问题领域。这种分解不但可以用来进行功能分析,而且可用作标识在整个系统的各个不同部分的通用机制和设计元素。在面向对象技术中,通过抽象、封装和继承,可以用对象模型来代表逻辑视图,用类图(class diagram)来描述逻辑视图。可以从 Booch 标记法中导出逻辑视图的标记法,只是从体系结构级的范畴来考虑这些符号,用 Rational Rose 进行体系结构设计。图 2-2 是逻辑视图中使用的符号集合。

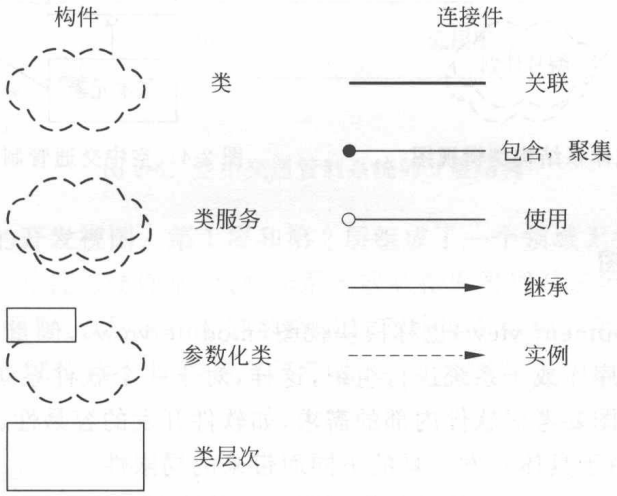


图 2-2 逻辑视图中使用的标记符号

类图用于表示类的存在以及类与类之间的相互关系,是从系统构成的角度来描述正在开发的系统。一个类的存在不是孤立的,类与类之间以不同方式互相合作,共同完成某些系统功能。关联关系表示两个类之间存在着某种语义上的联系,其真正含义要由附加在横线之上的一个短语来予以说明。在表示包含关系的图符中,带有实心圆的一端表示整体,相反的一端表示部分。在表示使用关系的图符中,带有空心圆的一端连接在请求服务的类,相反的一端连接在提供服务的类。在表示继承关系的图符中,箭头由子类指向基类。

逻辑视图中使用的风格为面向对象的风格,逻辑视图设计中要注意的主要问题是要保持一个单一的、内聚的对象模型贯穿整个系统。例如,图 2-3 是某通信系统体系结构(ACS)中的主要类。

ACS 的功能是在终端之间建立连接,这种终端可以是电话机、主干线、专用线路、特殊电话线、数据线或 ISDN 线路等,不同线路由不同的线路接口卡进行支持。线路控制器对象的作用是译码并把所有符号加入到线路接口卡中。终端对象的作用是保持终端的状态,代表本条线路的利益参与协商服务。会话对象代表一组参与会话的终端,使用转换服务(目录、逻辑地址映射到物理地址,路由等)和连接服务在终端之间建立语音路径。

对于规模更大的系统来说,体系结构级中包含数十甚至数百个类,例如,图 2-4 是一个空中交通管制系统的一级类图,该图包含 8 组类。

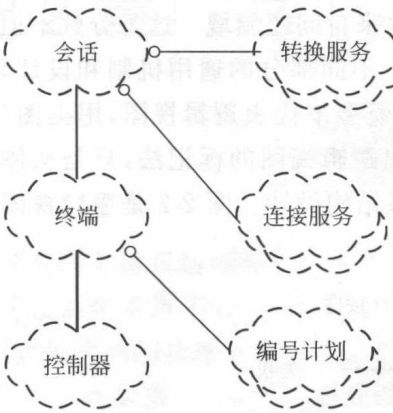


图 2-3　某通信系统体系结构逻辑视图

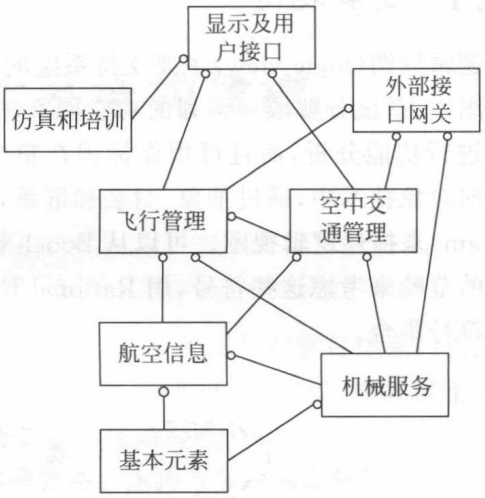


图 2-4　空中交通管制系统的一级类图

2.1.2　开发视图

开发视图(development view)也称模块视图(module view),侧重于软件模块的组织和管理。软件可通过程序库或子系统进行组织,这样,对于一个软件系统,就可以由不同的人员进行开发。开发视图要考虑软件内部的需求,如软件开发的容易性、软件的重用和软件的通用性,要充分考虑由于具体开发工具的不同而带来的局限性。

开发视图通过系统输入输出关系的模型图和子系统图来描述。可以在确定了软件包含的所有元素之后描述完整的开发角度,也可以在确定每个元素之前,列出开发视图原则。

与逻辑视图一样,可以使用 Booch 标记法中某些符号来表示开发视图,如图 2-5 所示。

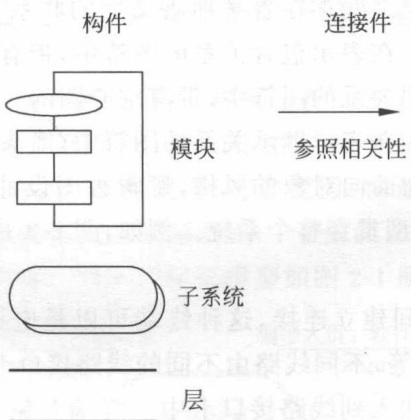


图 2-5　开发视图中使用的标记符号

在开发视图中,最好采用 4～6 层子系统,而且每个子系统仅能与同层或更低层的子系统通信,这样可以使每个层次的接口既完备又精练,避免了各个模块之间很复杂的依赖关系。而且,设计时要充分考虑,对于各个层次,层次越低,通用性越强,这样,可以保证应用程

序的需求发生改变时，所做的改动最小。开发视图所用的风格通常是层次结构风格。例如，图 2-6 所示的是空中交通管制系统的 5 层结构图。

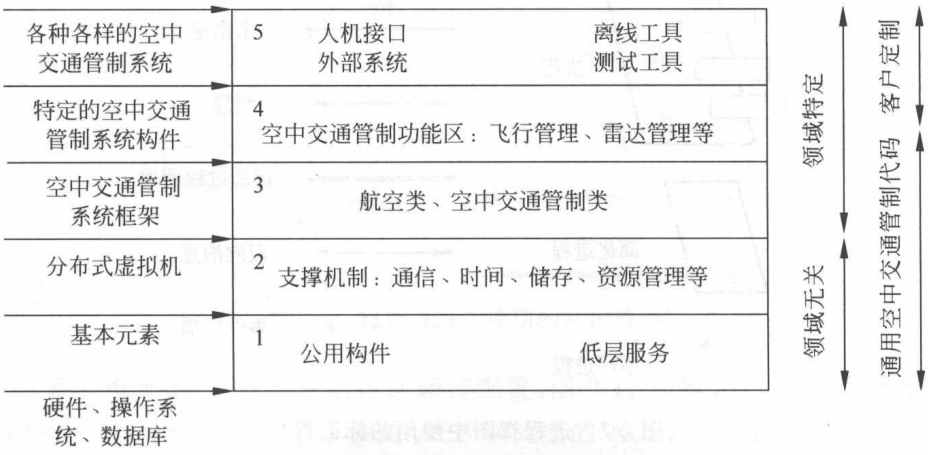


图 2-6 空中交通管制系统的 5 层结构

图 2-6 是图 2-4 的开发视图。第 1 层和第 2 层组成了一个领域无关的分布式基础设施，贯穿于整个产品线中，并且与硬件平台、操作系统或数据库管理系统等无关。第 3 层增加了空中交通管制系统的框架，以形成一个领域特定的软件体系结构。第 4 层使用该框架建立一个功能平台，第 5 层则依赖于具体客户和产品，包含大部分用户接口以及与外部系统的接口。

2.1.3 进程视图

进程视图(process view)也称并发视图，侧重于系统的运行特性，主要关注一些非功能性的需求，如系统的性能和可用性。进程视图强调并发性、分布性、系统集成性和容错能力，以及从逻辑视图中的主要抽象如何适合进程结构。它也定义逻辑视图中的各个类的操作具体是在哪一个线程(thread)中被执行的。

进程视图可以描述成多层抽象，每个级别分别关注不同的方面。在最高层抽象中，进程结构可以看作是构成一个执行单元的一组任务。它可看成一系列独立的，通过逻辑网络相互通信的程序。它们是分布的，通过总线或局域网、广域网等硬件资源连接起来。通过进程视图可以从进程测量一个目标系统的最终执行情况。例如，在以计算机网络作为运行环境的图书管理系统中，服务器需对来自各个不同的客户机的进程管理，决定某个特定进程(如查询子进程、借还书子进程)的唤醒、启动、关闭等操作，从而控制整个网络协调有序地工作。

通过扩展 Booch 对 Ada 任务的表示法，来表示进程视图，从体系结构角度来看，进程视图的标记元素如图 2-7 所示。

有很多风格适用于进程视图，如管道和过滤器风格、客户-服务器风格(多客户-单服务器，多客户-多服务器)等。图 2-8 是 2.1.1 节中的 ACS 系统的局部进程视图。

在图 2-8 中，所有终端均由同一个终端进程进行处理，由其输入队列中的消息驱动。控制器对象在组成控制器进程的三个任务之一中执行，慢循环周期(200ms)任务扫描所有挂起(suspend)终端，把任何一个活动的终端置入快循环周期(10ms)任务的扫描列表，快循环

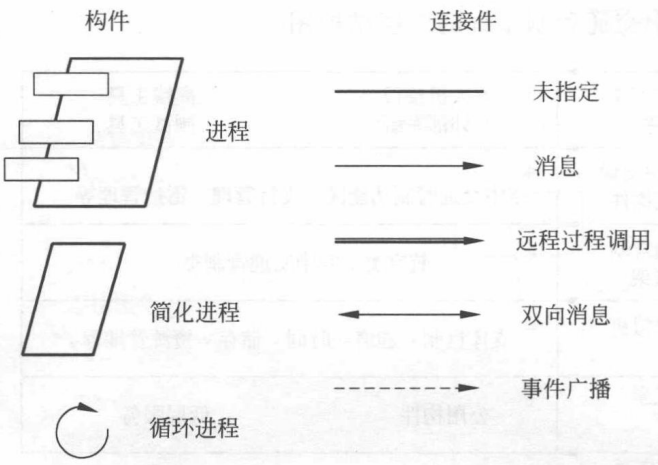


图 2-7　进程视图中使用的标记符号

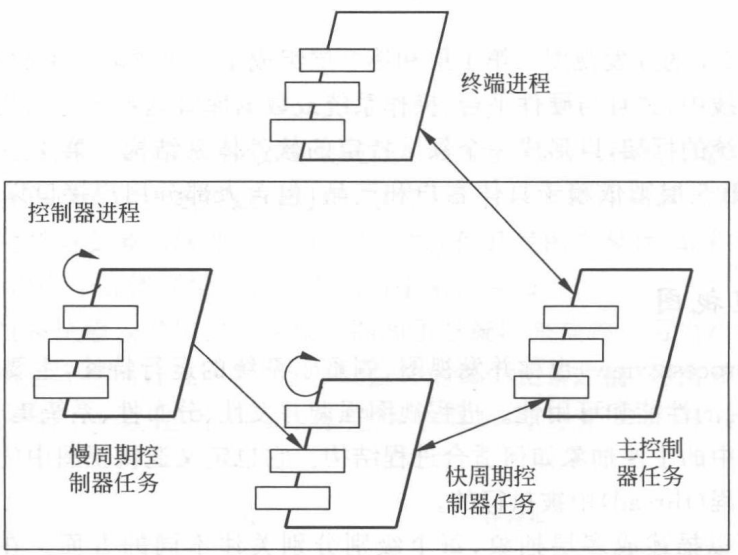


图 2-8　ACS 系统的局部进程视图

周期任务检测任何显著的状态改变,并把改变的状态传递给主控制器任务,主控制器任务解释改变,通过消息与相应的终端进行通信。在这里,通过共享内存来实现在控制器进程中传递的消息。

2.1.4　物理视图

物理视图(physical view)主要考虑如何把软件映射到硬件上,它通常要考虑到系统性能、规模、可靠性等。解决系统拓扑结构、系统安装、通信等问题。当软件运行于不同的结点上时,各视图中的构件都直接或间接地对应于系统的不同结点上。因此,从软件到结点的映射要有较高的灵活性,当环境改变时,对系统其他视图的影响最小。

大型系统的物理视图可能会变得十分混乱,因此可以与进程视图的映射一道,以多种形式出现,也可单独出现。图 2-9 是物理视图的标记元素集合。

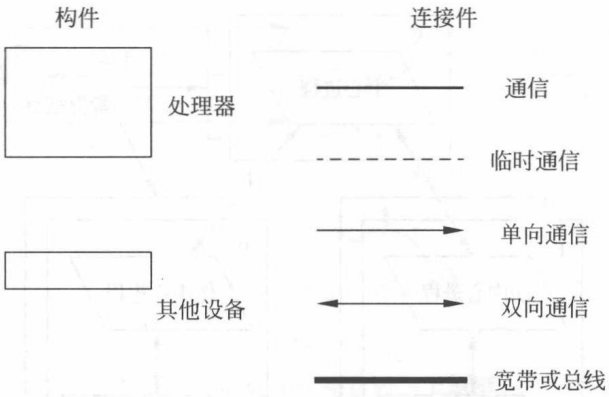


图 2-9　物理视图中使用的标记符号

图 2-10 是一个大型 ACS 系统的可能硬件配置，图 2-11 和图 2-12 是进程视图的两个不同的物理视图映射，分别对应一个小型的 ACS 和大型的 ACS，C、F 和 K 是三个不同容量的计算机类型，支持三个不同的可执行文件。

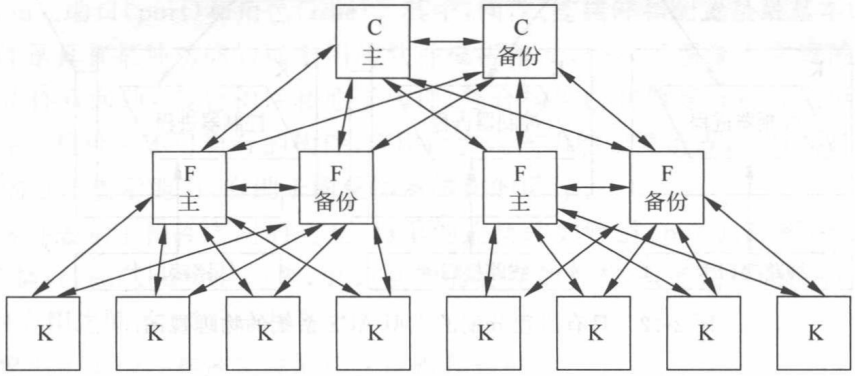


图 2-10　ACS 系统的物理视图

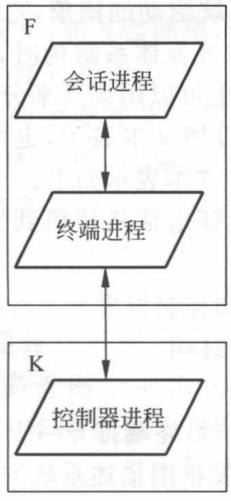


图 2-11　具有进程分配的小型 ACS 系统的物理视图

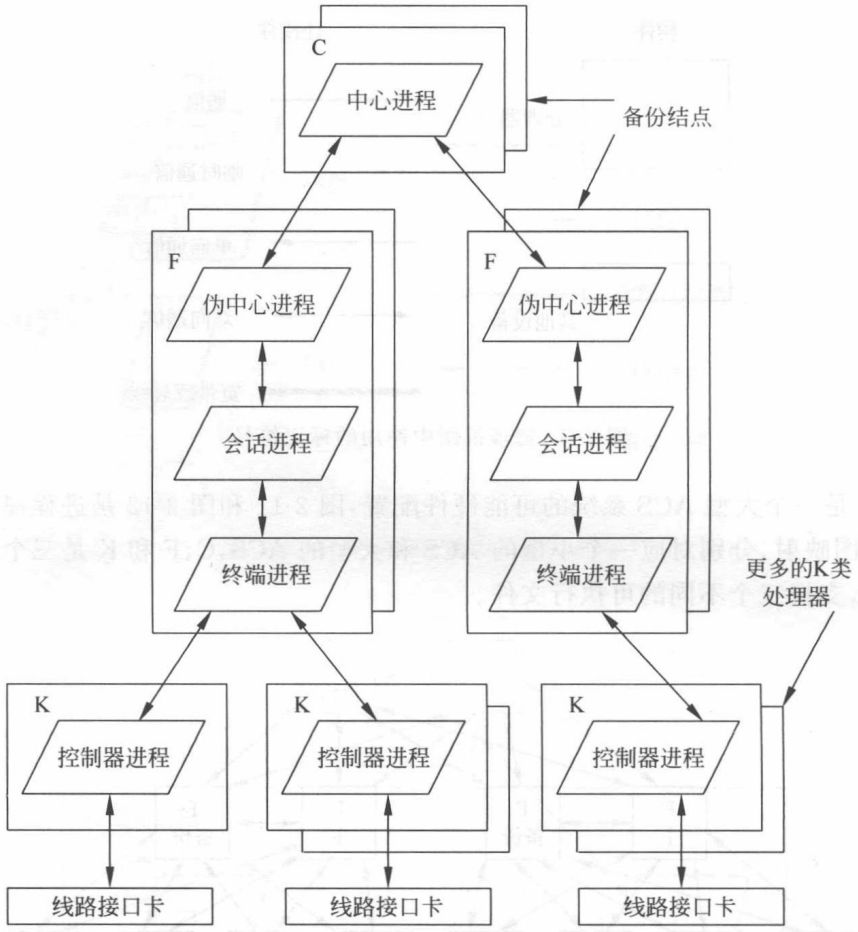


图 2-12　具有进程分配的大型 ACS 系统的物理视图

2.1.5　场景

场景(scenarios)可以看作重要系统活动的抽象,它使 4 个视图有机联系起来,从某种意义上说场景是最重要的需求抽象。在开发体系结构时,它可以帮助设计师找到体系结构的构件和它们之间的作用关系。同时,也可以用场景来分析一个特定的视图,或描述不同视图构件间是如何相互作用的。场景可以用文本表示,也可以用图形表示。例如,图 2-13是一个小型 ACS 系统的场景片段,相应的文本表示如下。

(1) 小王的电话控制器检测和验证电话从挂机到摘机状态的转变,且发送一个消息以唤醒相应的终端对象。

(2) 终端分配一定的资源,且通知控制器发出某种拨号音。

(3) 控制器接收所拨号码并传给终端。

(4) 终端使用编号计划分析号码。

(5) 当一个有效的拨号序列进入时,终端打开一个会话。

从以上分析可知,逻辑视图和开发视图描述系统的静态结构,而进程视图和物理视图描述系统的动态结构。对于不同的软件系统来说,侧重的角度也有所不同。例如,对于管理信息系统来说,比较侧重于从逻辑视图和开发视图来描述系统,而对于实时控制系统来说,则比较注重于从进程视图和物理视图来描述系统。

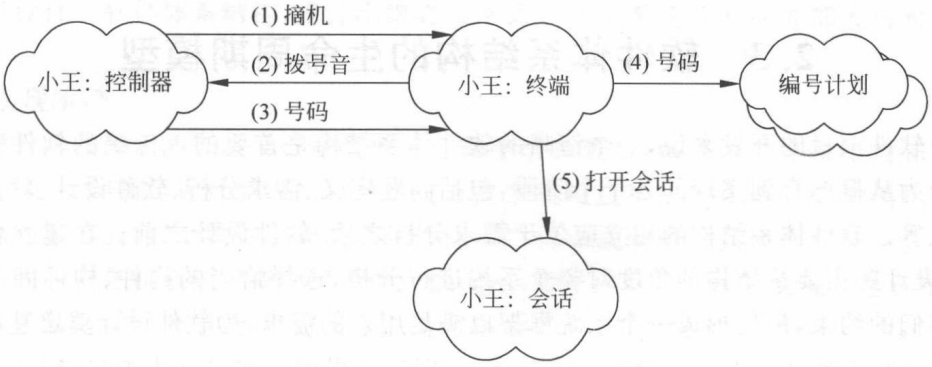


图 2-13　本地呼叫场景的一个原型

2.2　软件体系结构的核心模型

综合软件体系结构的概念，体系结构的核心模型由 5 种元素组成：构件、连接件、配置(configuration)、端口(port)和角色(role)。其中，构件、连接件和配置是最基本的元素。

(1) 构件是具有某种功能的可重用的软件模板单元，表示了系统中主要的计算元素和数据存储。构件有两种：复合构件和原子构件，复合构件由其他复合构件和原子构件通过连接而成；原子构件是不可再分的构件，底层由实现该构件的类组成，这种构件的划分提供了体系结构的分层表示能力，有助于简化体系结构的设计。

(2) 连接件表示了构件之间的交互，简单的连接件如管道(pipe)、过程调用(procedure call)、事件广播(event broadcast)等，更为复杂的交互如客户-服务器(client-server)通信协议，数据库和应用之间的 SQL 连接等。

(3) 配置表示了构件和连接件的拓扑逻辑和约束。

另外，构件作为一个封装的实体，只能通过其接口与外部环境交互，构件的接口由一组端口组成，每个端口表示了构件和外部环境的交互点。通过不同的端口类型，一个构件可以提供多重接口。一个端口可以非常简单，如过程调用，也可以表示更为复杂的界面(包含一些约束)，如必须以某种顺序调用的一组过程调用。

连接件作为建模软件体系结构的主要实体，同样也有接口，连接件的接口由一组角色组成，连接件的每一个角色定义了该连接件表示的交互的参与者，二元连接件有两个角色，例如，RPC(Remote Procedure Call，远程过程调用)的角色为 caller 和 callee，pipe 的角色是 reading 和 writing，消息传递连接件的角色是 sender 和 receiver。有的连接件有多于两个的角色，如事件广播有一个事件发布者角色和任意多个事件接收者角色。

基于以上所述，可将软件体系结构的核心模型表示为如图 2-14 所示。

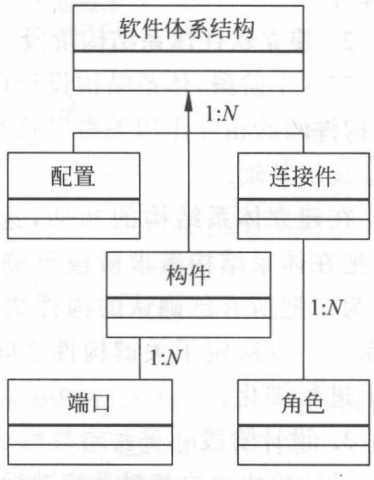


图 2-14　软件体系结构的核心模型

2.3 软件体系结构的生命周期模型

对于软件项目的开发来说,一个清晰的软件体系结构是首要的。传统的软件开发过程可以划分为从概念直到实现的若干个阶段,包括问题定义、需求分析、软件设计、软件实现及软件测试等。软件体系结构的建立应位于需求分析之后,软件设计之前。在建立软件体系结构时,设计师主要从结构的角度对整个系统进行分析,选择恰当的构件、构件间的相互作用以及它们的约束,最后形成一个系统框架以满足用户的需求,为软件设计奠定基础。

2.3.1 各阶段之间的关系

下面从各个阶段的功能出发,分析这几个层次之间的关系。

1. 需求分析阶段

需求分析阶段的任务是根据需求决定系统的功能,在此阶段,设计师应对目标对象和环境作细致深入的调查,收集目标对象的基本信息,从中找出有用的信息,这是一个抽象思维、逻辑推理的过程,其结果是软件规格说明。需求是指用户对目标软件系统在功能、行为、性能、设计约束等方面的期望,需求过程主要是获取用户需求,确定系统中所要用到的构件。

体系结构需求包括需求获取、生成类图、对类分组、将类打包成构件和需求评审等过程。其中,需求获取主要是定义开发人员必须实现的软件功能,使得用户能完成他们的任务,从而满足业务上的功能需求。与此同时,还要获得软件质量属性,满足一些非功能需求。获取了需求之后,就可以利用工具(例如,Rational Rose)自动生成类图,然后对类进行分组,简化类图结构,使之更清晰。分组之后,再要将类簇打包成构件,这些构件可以分组合并成更大的构件。

最后进行需求评审,组织一个由不同代表(例如,系统分析师、需求分析师、系统设计师、客户代表、测试人员等)组成的小组,对体系结构需求及相关构件进行仔细的审查。审查的主要内容包括所获取的需求是否真实反映了用户的要求,类的分组是否合理,构件合并是否合理等。

2. 建立软件体系结构阶段

在这个阶段,体系结构设计师主要从结构的角度对整个系统进行分析,选择恰当的构件、构件间的相互作用关系以及对它们的约束,最后形成一个系统框架以满足用户需求,为设计奠定基础。

在建立体系结构的初期,选择一个合适的体系结构风格是首要的。选择了风格之后,把在体系结构需求阶段已确认的构件映射到体系结构中,将产生一个中间结构。然后,为了把所有已确认的构件集成到体系结构中,必须认真分析这些构件的相互作用和关系。一旦决定了关键构件之间的关系和相互作用,就可以在前面得到的中间结构的基础上进行细化。

3. 设计阶段

设计阶段主要是对系统进行模块化并决定描述各个构件间的详细接口、算法和数据类型的选定,对上支持建立体系结构阶段形成的框架,对下提供实现基础。

一旦设计了软件体系结构，设计小组必须邀请独立于系统开发的外部人员对体系结构进行评审。

4. 实现阶段

将设计阶段设计的算法及数据类型进行程序语言表示，满足设计体系结构和需求分析的要求，从而得到满足设计需求的目标系统。整个实现过程是以复审后的文档化的体系结构说明书为基础的，每个构件必须满足软件体系结构中说明的对其他构件的责任。这些决定即实现的约束是在系统级或项目范围内做出的，每个构件上工作的实现者是看不见的。

在体系结构说明书中，已经定义了系统中的构件与构件之间的关系。因为在体系结构层次上，构件接口约束对外唯一地代表了构件，所以可以从构件库中查找符合接口约束的构件，必要时开发新的满足要求的构件。

然后，按照设计提供的结构，通过组装支持工具把这些构件的实现体组装起来，完成整个软件系统的连接与合成。

最后一步是测试，包括单个构件的功能性测试和被组装应用的整体功能和性能测试。

2.3.2 软件体系结构的生命周期

软件体系结构在系统开发的全过程中起着基础的作用，是设计的起点和依据，同时也是装配和维护的指南。与软件本身一样，软件体系结构也有其生命周期，图 2-15 形象地表示了体系结构的生命周期。

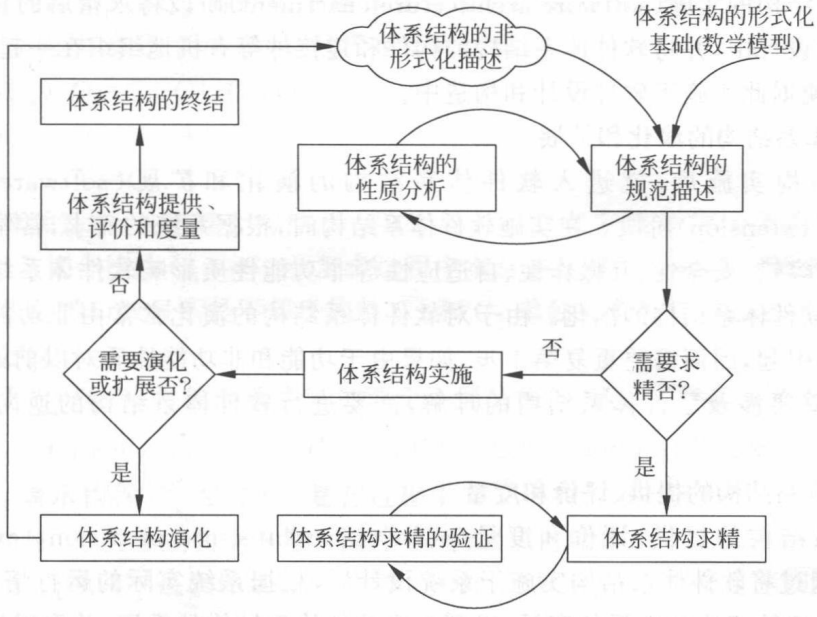


图 2-15 软件体系结构的生命周期模型

1. 软件体系结构的非形式化描述

在软件体系结构的非形式化描述(software architecture informal description)阶段，对软件体系结构的描述尽管常用自然语言，但是该阶段的工作却是创造性和开拓性的。一种软件体系结构在其产生时，其思想通常是简单的，并常常由软件设计师用非形式化的自然语言表示概念、原则。例如，客户-服务器体系结构就是为适应分布式系统的要求，从主从式演

变而来的一种软件体系结构。

2. 软件体系结构的规范描述和分析

软件体系结构的规范描述和分析(software architecture specification and analysis)阶段通过运用合适的形式化数学理论模型对第1阶段的体系结构的非形式化描述进行规范定义,从而得到软件体系结构的形式化规范描述,以使软件体系结构的描述精确、无歧义,并进而分析软件体系结构的性质,如无死锁性、安全性、活性等。分析软件体系结构的性质有利于在系统设计时选择合适的软件体系结构,从而对软件体系结构的选择起指导作用,避免盲目选择。

3. 软件体系结构的求精及其验证

软件体系结构的求精及其验证(software architecture refinement and verification)阶段完成对已设计好的软件体系结构进行验证和求精。大型系统的软件体系结构总是通过从抽象到具体,逐步求精而达到的,因为一般来说,由于系统的复杂性,抽象是人们在处理复杂问题和对象时必不可少的思维方式,软件体系结构也不例外。但是过高的抽象却使软件体系结构难以真正在系统设计中实施。因而,如果软件体系结构的抽象粒度过大,就需要对体系结构进行求精、细化,直至能够在系统设计中实施为止。在软件体系结构的每一步求精过程中,需要对不同抽象层次的软件体系结构进行验证,以判断较具体的软件体系结构是否与较抽象的软件体系结构的语义一致,并能实现抽象的软件体系结构。

4. 软件体系结构的实施

软件体系结构的实施(software architecture enactment)阶段将求精后的软件体系结构实施于系统的设计中,并将软件体系结构的构件和连接件等有机地组织在一起,形成系统设计的框架,以便据此实施于软件设计和构造中。

5. 软件体系结构的演化和扩展

当体系结构实施后,就进入软件体系结构的演化和扩展(software architecture evolution and extension)阶段。在实施软件体系结构时,根据系统的需求,常常是非功能的需求,如性能、容错、安全性、互操作性、自适应性等非功能性质影响软件体系结构的扩展和改动,这称为软件体系结构的演化。由于对软件体系结构的演化常常由非功能性质的非形式化需求描述引起,因而需要重复第1步,如果由于功能和非功能性质对以前的软件体系结构进行演化,就要涉及软件体系结构的理解,需要进行软件体系结构的逆向工程和再造工程。

6. 软件体系结构的提供、评价和度量

软件体系结构的提供、评价和度量(software architecture provision, evaluation and metric)阶段通过将软件体系结构实施于系统设计后,根据系统实际的运行情况,对软件体系结构进行定性的评价和定量的度量,以利于对软件体系结构的重用,并取得经验教训。

7. 软件体系结构的终结

如果一个软件系统的软件体系结构经过多次演化和修改,软件体系结构已变得难以理解,更重要的是不能达到系统设计的要求,不能适应系统的发展,这时,对该软件体系结构的再造工程既不必要,也不可行,说明该软件体系结构已经过时,应该摈弃,以全新的满足系统设计要求的软件体系结构取而代之。这个阶段被称为软件体系结构的终结(software architecture termination)阶段。

2.4 软件体系结构的抽象模型

本节将使用抽象代数理论，对构件、连接件和软件体系结构的定义以及它们的属性和动态行为进行讨论，建立软件体系结构的数学理论体系，讨论不同类型软件系统结构的相互关系，给出软件体系结构范式的概念和建立范式的方法。读者需要具有初步的抽象代数知识，才能阅读和理解本节内容。

2.4.1 构件及其关系的抽象描述

在软件系统中，构件不是独立存在的，构件之间具有互操作性。构件可能有不同的理解，本节把构件看成数据单元或计算单元，把构件之间的互操作抽象为代数运算，具体定义如下。

定义 1 构件是一个数据单元或一个计算单元，它由构件的对象的集合、属性的集合、动作的集合和端口的集合组成。构件可抽象为 $C=(O,A,X,P)$。这里，O 是组成构件的所有对象的集合，A 是构件属性的集合，X 是构件动作的集合，P 是构件端口的集合。

定义 2 设 C_1 和 C_2 是软件系统 S 中的两个不同构件，若先执行构件 C_1 的操作，再执行构件 C_2 的操作，就称构件 C_1 和 C_2 之间的关系为顺序结构关系，简称为顺序结构或顺序运算，记作 $C_1 \mapsto C_2$。用图形表示如图 2-16 所示。

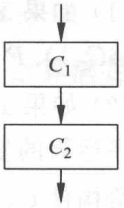


图 2-16 构件顺序结构关系图

显然，两个构件 C_1 和 C_2 通过顺序运算复合而成的构件 $C_1 \rightarrow C_2$ 仍然是 S 的一个构件，如图 2-16 所示，且满足下列性质。

1. $(C_1 \mapsto C_2).O=C_1.O \cup C_2.O$
2. $(C_1 \mapsto C_2).A=C_1.A \cup C_2.A$
3. $(C_1 \mapsto C_2).X=C_1.X \cup C_2.X$

在顺序运算关系中，虽然复合构件 $C_1 \mapsto C_2$ 仍然是软件系统 S 的一个构件，满足上述的三条性质，但是复合构件 $C_1 \mapsto C_2$ 在复合过程中本身要使用一些端口，因此构件 $C_1 \mapsto C_2$ 为外部环境提供的端口数显然是不等于构件 C_1 和 C_2 单独提供的端口数之和，所以要对复合的端口进行定义。

定义 3 设 C_1 和 C_2 是软件系统 S 的两个不同的构件，通过顺序运算复合后，则有

$$(C_1 \mapsto C_2).P=(C_1.P-C_1.P^{\downarrow}_{\text{used}}) \cup (C_2.P-C_2.P^{\uparrow}_{\text{used}})$$

其中，$C_1.P^{\downarrow}_{\text{used}}$ 表示构件 C_1 在与 C_2 复合过程中向下为构件 C_2 所提供的端口，$C_2.P^{\uparrow}_{\text{used}}$ 表示构件 C_2 在与 C_1 复合过程中向上为构件 C_1 所提供的端口。

定义 4 设 C_1 和 C_2 是软件系统 S 的两个构件，如果 C_1 和 C_2 满足下列条件。

1. $C_1.O=C_2.O$
2. $C_1.A=C_2.A$
3. $C_1.X=C_2.X$
4. $C_1.P=C_2.P$

则称 C_1 和 C_2 相等，记作 $C_1=C_2$。

定理 1 设 C_1、C_2 和 C_3 是软件系统 S 的三个任意构件，则它们对 $\mapsto$ 运算满足结合律，即：

$$(C_1 \mapsto C_2)\mapsto C_3 = C_1 \mapsto (C_2 \mapsto C_3)$$

证明：要证明上式成立，只要证明等式两边满足定义4的4个条件。为此，有代表性地证明条件1和条件4成立，其他两个条件的证明类似条件1的证明。

对 $\forall x \in ((C_1 \mapsto C_2)\mapsto C_3).O$，则由定义2的性质1可知 $x \in (C_1 \mapsto C_2).O$ 或者 $x \in C_3.O$，下面分两种情况证明。

(1) 如果 $x \in C_3.O$，则 $x \in (C_2 \mapsto C_3).O$，因此 $x \in (C_1 \mapsto (C_2 \mapsto C_3)).O$。

(2) 如果 $x \in (C_1 \mapsto C_2).O$，同样由定义2的性质1可知 $x \in C_1.O$ 或 $x \in C_2.O$。

① 若 $x \in C_1.O$，则 $x \in (C_1 \mapsto (C_2 \mapsto C_3)).O$。

② 若 $x \in C_2.O$，则 $x \in (C_2 \mapsto C_3).O$，因此也有 $x \in (C_1 \mapsto (C_2 \mapsto C_3)).O$。

同理，对 $\forall y \in (C_1 \mapsto (C_2 \mapsto C_3)).O$，可以证明有 $y \in ((C_1 \mapsto C_2)\mapsto C_3).O$ 成立。因此，$((C_1 \mapsto C_2)\mapsto C_3).O = (C_1 \mapsto (C_2 \mapsto C_3)).O$，即定义4的条件1成立。

下面证明定义4的条件4成立。

对 $\forall x \in ((C_1 \mapsto C_2)\mapsto C_3).P$，则由定义3可知 $x \in (C_1 \mapsto C_2).P - (C_1 \mapsto C_2).P^{\downarrow}_{\text{used}}$ 或者 $x \in C_3.P - C_3.P^{\uparrow}_{\text{used}}$，下面分两种情况证明。

(1) 如果 $x \in C_3.P - C_3.P^{\uparrow}_{\text{used}}$，则由定义3可知 $x \in (C_2 \mapsto C_3).P$，故 $x \in (C_1 \mapsto (C_2 \mapsto C_3)).P$。

(2) 如果 $x \in (C_1 \mapsto C_2).P - (C_1 \mapsto C_2).P^{\downarrow}_{\text{used}}$，在此，表达式 $(C_1 \mapsto C_2).P^{\downarrow}_{\text{used}}$ 表示的是顺序运算的复合构件 $C_1 \mapsto C_2$ 向构件 C_3 提供的端口，在顺序运算中实际上是由构件 C_2 向下给构件 C_3 提供端口。因此有 $(C_1 \mapsto C_2).P^{\downarrow}_{\text{used}} = C_2.P^{\downarrow}_{\text{used}}$。

由定义3可知 $(C_1 \mapsto C_2).P = (C_1.P - C_1.P^{\downarrow}_{\text{used}}) \cup (C_2.P - C_2.P^{\uparrow}_{\text{used}})$，故 $x \in C_1.P - C_1.P^{\downarrow}_{\text{used}}$ 或 $x \in C_2.P - (C_2.P^{\uparrow}_{\text{used}} + C_2.P^{\downarrow}_{\text{used}})$。

① 若 $x \in C_1.P - C_1.P^{\downarrow}_{\text{used}}$，由定义3可知 $x \in (C_1 \mapsto (C_2 \mapsto C_3)).P$。

② 若 $x \in C_2.P - (C_2.P^{\uparrow}_{\text{used}} + C_2.P^{\downarrow}_{\text{used}})$，同理可证 $x \in (C_1 \mapsto (C_2 \mapsto C_3)).P$。

同理，对 $\forall y \in (C_1 \mapsto (C_2 \mapsto C_3)).P$，可以证明 $y \in ((C_1 \mapsto C_2)\mapsto C_3).P$ 成立。因此，$((C_1 \mapsto C_2)\mapsto C_3).P = (C_1 \mapsto (C_2 \mapsto C_3)).P$，即定义4的条件4成立。证毕。

定义5 设 C_1 和 C_2 是软件系统 S 中的两个不同的构件，若根据不同的条件选择性地执行构件 C_1 或 C_2 来完成某一功能，就称 C_1 和 C_2 之间的关系为选择结构关系，简称为选择结构或选择运算，记作 $C_1 \lhd C_2$。用图形表示，如图2-17所示。

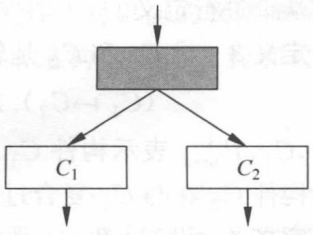


图2-17 构件选择结构关系图

软件系统 S 在执行选择运算过程中，它的执行方式是先根据条件构件进行判断，然后再有选择地沿着构件 C_1 或者构件 C_2 的路径执行。

显然，$C_1 \lhd C_2$ 也仍然是软件系统 S 的一个构件，且满足下列性质。

1. $(C_1 \lhd C_2).O = C_1.O$ 或 $(C_1 \lhd C_2).O = C_2.O$
2. $(C_1 \lhd C_2).A = C_1.A$ 或 $(C_1 \lhd C_2).A = C_2.A$
3. $(C_1 \lhd C_2).X = C_1.X$ 或 $(C_1 \lhd C_2).X = C_2.X$
4. $(C_1 \lhd C_2).P = C_1.P$ 或 $(C_1 \lhd C_2).P = C_2.P$

由定义 2 和定义 5 可以看出，↦运算与◁运算是不同的。在◁运算中，C_1 和 C_2 的功能实现是独立的，只是在功能执行上存在依赖关系。而在↦运算中，C_1 和 C_2 的功能实现是非独立的，构件 C_1 的实现代码与构件 C_2 的实现代码有关，构件之间需要提供端口供通信使用。

定理 2 设 C_1 和 C_2 是软件系统 S 中的两个任意构件，则它们对◁运算满足交换律，即

$$C_1 \lhd C_2 = C_2 \lhd C_1$$

证明：要证明上式成立，只要证明等式两边满足定义 4 的 4 个条件。为此，有代表性地证明条件 1 成立，其他三个条件的证明是类似的。

对 $\forall x \in (C_1 \lhd C_2).O$，则有 $x \in C_1.O$ 或者 $x \in C_2.O$，根据定义 5 的性质 1，显然有 $x \in (C_2 \lhd C_1).O$。

同理，$\forall x \in (C_2 \lhd C_1).O$，显然也有 $x \in (C_1 \lhd C_2).O$。

因此，$(C_1 \lhd C_2).O = (C_2 \lhd C_1).O$，即定义 4 的条件 1 成立。

同理，可证等式两边满足定义 4 的其他三个条件。因此等式定理 2 成立，证毕。

与↦运算类似，◁运算也满足结合律。

定理 3 设 C_1、C_2 和 C_3 是软件系统 S 的三个任意构件，则它们对◁运算满足结合律，即：

$$(C_1 \lhd C_2) \lhd C_3 = C_1 \lhd (C_2 \lhd C_3)$$

证明：与定理 2 的证明类似，此略。

定义 6 设 C_1 和 C_2 是软件系统 S 中的两个不同的构件，若在一定的条件下重复执行构件 C_1 和构件 C_2 的功能，就称构件 C_1 和 C_2 之间的关系是循环结构关系，简称为循环结构或循环运算，记作 $C_1 \leftrightarrow C_2$。用图形表示，如图 2-18 所示。

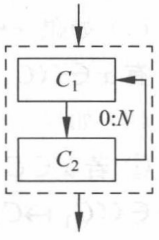


图 2-18 构件循环结构关系图

显然，$C_1 \leftrightarrow C_2$ 仍然是软件系统 S 的一个构件，且满足下列性质：

1. $(C_1 \leftrightarrow C_2).O = C_1.O \cup C_2.O$
2. $(C_1 \leftrightarrow C_2).A = C_1.A \cup C_2.A$
3. $(C_1 \leftrightarrow C_2).X = C_1.X \cup C_2.X$
4. $(C_1 \leftrightarrow C_2).P = (C_1.P - C_1.P_{used}) \cup (C_2.P - C_2.P_{used})$

对三个构件而言，如果构件 C_2 和 C_3 的输出都是构件 C_1 的输入，则根据 C_2 和 C_3 的关系，可以有下列情形：

(1) 构件 C_2 与构件 C_3 之间的关系为顺序运算关系，即构件 C_2、C_3 是顺序执行，则记为 $C_2 \mapsto (C_1 \lhd (C_3 \mapsto C_1))$，如图 2-19 所示。

(2) 构件 C_2 和构件 C_3 之间的关系为选择运算关系，即构件的执行顺序为 $C_2 \mapsto C_1$ 或者 $C_3 \mapsto C_1$，则记为 $(C_2 \lhd C_3) \mapsto C_1$，如图 2-20 所示。

(3) 构件 C_2 和 C_3 之间的关系为循环运算关系，则记为 $(C_2 \leftrightarrow C_3) \mapsto C_1$，如图 2-21 所示。

(4) 构件 C_2 和 C_3 独立，没有直接关系，则把构件 C_2 和 C_3 当作选择结果进行处理，也记为 $(C_2 \lhd C_3) \mapsto C_1$，如图 2-20 所示。

下面讨论运算之间的关系。

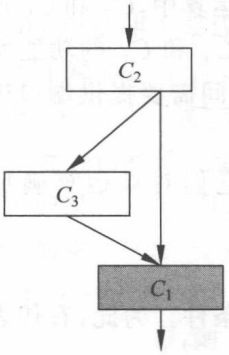


图 2-19 三个构件之间的关系 1

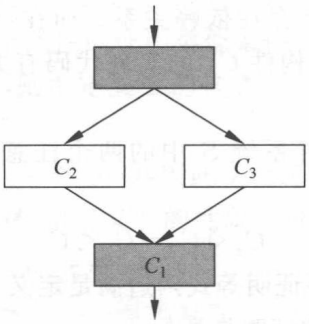


图 2-20 三个构件之间的关系 2

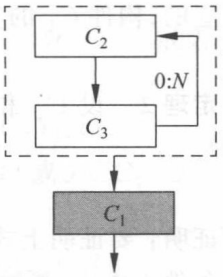


图 2-21 三个构件之间的关系 3

定理 4 设 C_1、C_2 和 C_3 是软件系统 S 的三个任意构件，则它们满足$\mapsto$运算对$\lhd$运算的分配律，即：

1. $(C_1 \lhd C_2) \mapsto C_3 = (C_1 \mapsto C_3) \lhd (C_2 \mapsto C_3)$

2. $C_3 \mapsto (C_1 \lhd C_2) = (C_3 \mapsto C_1) \lhd (C_3 \mapsto C_2)$

证明：只证明第一个等式成立，第二个等式可类似证明。

要证明第一个等式成立，只要证明等式两边满足定义 4 的 4 个条件。为此有代表性地证明条件 1 和 4 成立，其他两个条件的证明类似于条件 1 的证明。

对 $\forall x \in ((C_1 \lhd C_2) \mapsto C_3).O$，则由定义 2 的性质 1 可知 $x \in (C_1 \lhd C_2).O$ 或者 $x \in C_3.O$，下面分两种情况证明。

(1) 如果 $x \in C_3.O$，则 $x \in (C_1 \mapsto C_3).O$ 并且 $x \in (C_2 \mapsto C_3).O$，因此根据定义 5 的性质 1 有 $x \in ((C_1 \mapsto C_3) \lhd (C_2 \mapsto C_3)).O$ 成立。

(2) 如果 $x \in (C_1 \lhd C_2).O$，同样由定义 5 的性质 1 可知 $x \in C_1.O$ 或 $x \in C_2.O$。

① 若 $x \in C_1.O$，则根据定义 2 的性质 1，有 $x \in (C_1 \mapsto C_3).O$，再根据定义 5 的性质 1，有 $x \in ((C_1 \mapsto C_3) \lhd (C_2 \mapsto C_3)).O$ 成立。

② 若 $x \in C_2.O$，则根据定义 2 的性质 1，有 $x \in (C_2 \mapsto C_3).O$，再根据定义 5 的性质 1，也有 $x \in ((C_1 \mapsto C_3) \lhd (C_2 \mapsto C_3)).O$ 成立。

同理，对 $\forall y \in ((C_1 \mapsto C_3) \lhd (C_2 \mapsto C_3)).O$，可以证明有 $y \in ((C_1 \lhd C_2) \mapsto C_3).O$ 成立。因此，$(C_1 \lhd C_2) \mapsto C_3 = (C_1 \mapsto C_3) \lhd (C_2 \mapsto C_3)$。

证明满足定义 4 的条件 1 的步骤类似于定理 1 的第二部分的证明，在此略。证毕。

定理 5 软件系统 S 的所有构件组成的集合与其上的$\mapsto$运算构成一个半群，记作 $<S, \mapsto>$。

证明：$\forall C_1, C_2 \in S$，由定义 2 知 $C_1 \mapsto C_2 \in S$，即$\mapsto$运算是封闭的，所以 S 与运算$\mapsto$构成一个代数系统。又根据定理 1，$\mapsto$运算满足结合律，所以 $<S, \mapsto>$ 是一个半群。证毕。

定理 6 软件系统 S 的所有构件组成的集合与其上的运算$\lhd$构成一个交换半群，记作 $<S, \lhd>$。

证明：$\forall C_1, C_2 \in S$，由定义 5 知 $C_1 \lhd C_2 \in S$，即$\lhd$运算是封闭的，所以 S 与运算$\lhd$构成一个代数系统。又根据定理 2 和定理 3，$\lhd$运算满足结合律和交换律，所以 $<S, \lhd>$ 是一个交换半群。证毕。

2.4.2 连接件

连接件是软件体系结构的一个组成部分，它通过对构件间的交互规则的建模来实现构件间的连接。与构件不同，连接件不需编译。

定义 7 连接件是构件运算的实现，它是一个 6 元组＜ID，Role，Beha，Msgs，Cons，Non-Func＞。其中：

ID 是连接件的标识。

Role 为连接件和构件的交互点的集合，Role＝＜Id，Action，Event，LConstrains＞。其中，Id 是 Role 的标识；Action 是 Role 活动的集合，每个活动由事件的连接（谓词）组成；Event 是 Role 产生的事件集合；LConstrains 是 Role 的约束集合。把 Role 从连接件的其他属性分开来描述的目的是突出连接件的多态性，即一个连接件可同时与多个构件相连。

Beha 是连接件的行为集合。

Msgs 是连接件中各 Role 中事件产生的消息的结合。

Cons 是连接件约束的集合，它包括连接件的初始条件、前置条件和后置条件，有时为了明确表示这三个条件，可把它写成 Cons（init，pre-cond，post-cond），其中 init、pre-cond 和 post-cond 分别表示初始条件、前置条件和后置条件的集合。

Non-Func 是连接件的非功能说明，包括连接件的安全性、可靠性说明等。

2.4.3 软件体系结构

软件体系结构是一个设计，它包括所建立系统中的各元素（构件和连接件）的描述、元素之间的交互、指导装配的范例和对范例的约束。按照这一思想，并考虑到与构件和连接件定义的一致性，下面给出软件体系结构的递归定义。

定义 8 设论域为 U，

（1）构件是一个软件体系结构；

（2）连接件是一个软件体系结构；

（3）构件经有限次连接（运算）后是软件体系结构。

软件体系结构记为 $A=<C,O>$，其中 C 表示组成体系结构的构件集合，O 表示构件运算的集合。

由定义 8 可得软件体系结构的性质如下。

（1）封闭性（envelopment）：即构件与构件、构件与体系结构、体系结构与体系结构连接后仍是一个体系结构。

（2）层次性（hierarchy）：即体系结构可由构件连接而成，而体系结构又可以再经过连接组成新的更大的体系结构。

（3）可扩充性（expansibility）：即一个满足条件的新构件可以通过连接加入到结构中。

引理 1 设 $A=<C,O>$ 是一个软件体系结构，则 A 是一个代数系统。

由软件体系结构的封闭性可得引理 1 的正确性。

定理 7 设 $C_1,C_2,\cdots,C_n$ 是 A 的任意 n 个构件，则它们对 $\mapsto$ 运算仍然是 A 的一个构件。

证明：假设 $C_1 \mapsto C_2 \mapsto \cdots \mapsto C_n = T_n$，则：

$$T_n=(C_1\mapsto C_2\mapsto\cdots\mapsto C_{n-1})\mapsto C_n=T_{n-1}\mapsto C_n$$

要证 T_n 是一个构件,只要证明 T_{n-1} 是一个构件。因为如果 T_{n-1} 是一个构件,则由定义2可知 $T_{n-1}\mapsto C_n$ 是一个构件,即 T_n 是一个构件。

由上式可得以下递推关系:

$$T_{n-1}=T_{n-2}\mapsto C_{n-1},T_{n-2}=T_{n-3}\mapsto C_{n-2},\cdots,T_3=T_2\mapsto C_3,T_2=T_1\mapsto C_2$$

由定义2,显然有 T_2 是一个构件,通过上式的递推关系,则 T_{n-1} 是一个构件。

所以,$T_{n-1}\mapsto C_n$ 是一个构件,即 T_n 是 A 的一个构件,证毕。

定理8 设 $C_1,C_2,\cdots,C_n$ 是 A 的任意 n 个不同构件,则它们对◁运算仍然是 A 的一个构件。

证明:与定理7的证明类似,此略。

定理9 设 $C_1,C_2,\cdots,C_n$ 是 A 的任意 n 个不同构件,则它们对↔运算仍然是 A 的一个构件。

证明:与定理7的证明类似,此略。

构件重用技术是基于构件的软件开发方法的一种重要手段,是提高软件生产效率和软件质量的有效途径。通过对构件运算性质和推论的探讨,可以发现,有限个不同的构件通过不同的运算组合,能够生产出实现不同功能的新构件,从而为构造出更多的实现不同功能的软件构件提供依据。

定理10 A 在↦、◁、↔运算关系下构成一个代数系统。

证明:由定理7、8、9可知对 $\forall C_1,C_2\in A$,都有 $C_1\mapsto C_2,C_1\lhd C_2,C_1\leftrightarrow C_2\in A$,即 A 对↦、◁、↔运算是封闭的。

所以,A 在↦、◁、↔运算关系下是一个代数系统,证毕。

2.4.4 软件体系结构关系

在不同的体系结构中往往存在着对应关系,这些对应关系反映了结构之间的变化和更新,这种对应关系为结构的一致性和适应性检查提供了必要的支持,为此定义结构间的映射。

定义9 设 $A_1=<C_1,O_1>,A_2=<C_2,O_2>$ 是两个不同的体系结构,若对 $\forall x\in C_1$,总有 $\exists y\in C_2$,使得 y 与 x 对应,则称 A_1 与 A_2 之间存在着一个映射,记为 $f:A_1\to A_2$ 或 $y=f(x)$。若映射是满射,则称体系结构间映射为满射;若映射是一对一的,则称体系结构间为一一映射。

定义10 设 $A_1=<C_1,O_1>,A_2=<C_2,O_2>$ 是两个体系结构,f 是 A_1 到 A_2 的一个映射,若对 $\forall x\in C_1$ 和 $\forall y\in C_2$ 有 $f(x\lhd y)=f(x)\mapsto f(y)$,其中 $\lhd\in O_1,\mapsto\in O_2$,则称 f 为从 A_1 到 A_2 的同态,也称 A_1 与 A_2 同态;若 f 是单射,则称 f 是从 A_1 到 A_2 的单一同态;若 f 是一一映射,则称 f 是从 A_1 到 A_2 的同构,也称 A_1 与 A_2 同构。

定理11 设 $A_1=<C_1,O_1>,A_2=<C_2,O_2>,A_3=<C_3,O_3>$ 是三个体系结构,f 是从 A_1 到 A_2 的一个映射,g 是从 A_2 到 A_3 的一个映射。则 $g\cdot f$ 是从 A_1 到 A_3 的体系结构同态,其中"·"是映射的连接。

证明:对 $\forall a,b,c\in C_1,\exists\lhd\in O_1,\exists\mapsto\in O_2,\exists\varphi\in O_3$,根据·的定义有

$$g\cdot f(a\lhd b)=g(f(a\lhd b))=g(f(a)\mapsto f(b))=g(f(a))\phi g(f(b))=(g\cdot f)(a)\phi(g\cdot f)(b)$$

可见，$g \cdot f$ 构成了从 A_1 到 A_3 的结构同态。证毕。

定理 12 设 $A=<C,O>$是一个体系结构，如果 $\exists a,b \in C$，有 b 是 a 的一个演化，即 Evolve(b,a)成立，则体系结构 $A=<C,O>$和 $A'=<C',O>$同态，这里 $C'=(C-\{a\})\cup\{b\}$。

定理 12 的证明是显然的，此略。

定义 11 若 $A=<C,O>$是一个体系结构，如果对 $\exists a \in C$，$\exists b \in C$ 有 Evolve(b,a)成立，则称体系结构 $A'=<C',O>$是 $A=<C,O>$的一个演化，这里 $C'=(C-\{a\})\cup\{b\}$。

在软件体系结构中，一些构件之间往往存在着一定的联系。如某些构件来源于同类事物处理，某些构件有相同的连接方式等。为了更好地描述构件间的关系，下面给出相应的定义。

定义 12 给定两个体系结构 $A_1=<C_1,O_1>$，$A_2=<C_2,O_2>$，构造一个新的结构 $A_1 \lhd A_2=<C_1 \lhd C_2,\Theta>$。其中，$C_1 \lhd C_2$ 是构件集合的笛卡儿乘积，而 Θ 定义成对 $\forall x_1$，$x_2 \in C_1$ 和 $\forall y_1,y_2 \in C_2$ 有

$$<x_1,y_1>\Theta<x_2,y_2>=<x_1 \mapsto x_2,y_1 \lhd y_2>,\mapsto \in O_1,\lhd \in O_2$$

称 $A_1 \lhd A_2$ 是 A_1 到 A_2 的积体系结构，而 A_1 和 A_2 是 $A_1 \lhd A_2$ 的因子，这里的$\mapsto$和$\lhd$运算并非一定是“顺序”或“选择”运算。

在定义 12 的基础上可以定义体系结构间的关系。

定义 13 对 $\forall x_1,x_2 \in C_1$ 和 $\forall y_1,y_2 \in C_2$，在定义 11 中，所有满足$<x_1 \mapsto x_2>$的(x_1,x_2)的任意一个子集合称为关于运算$\mapsto$的一个关系：所有满足$<y_1 \lhd y_2>$的(y_1,y_2)的任意一个子集合称为关于$\lhd$运算的一个关系。

作为定义 13 的一个特例，令 $A_1=A_2$，运算$\mapsto$和$\lhd$是“顺序”运算和“选择”运算，则所有满足$<x_1 \mapsto x_2>$的(x_1,x_2)的任意一个子集合称为关于运算$\mapsto$的一个事件依赖关系，记为 $x_1 \xrightarrow{\text{事件依赖}} x_2$；所有满足$<y_1 \lhd y_2>$的$(y_1,y_2)$的任意一个子集合称为关于$\lhd$运算的一个实现依赖关系，记为 $y_1 \xrightarrow{\text{实现依赖}} y_2$。

有时为了讨论问题方便，对这两种依赖不加以区分，把它们简称为依赖，记为 $a \xrightarrow{\text{依赖}} b$。

定理 13 事件依赖和实现依赖满足传递性，即 $x \xrightarrow{\text{依赖}} y$，$y \xrightarrow{\text{依赖}} z$，则 $x \xrightarrow{\text{依赖}} z$。

证明：根据事件依赖和实现依赖的定义很容易证明，此处不再赘述，而是留给读者作为练习。

2.4.5 软件体系结构范式

软件体系结构范式应具有代表性，对软件开发有指导意义，应充分体现软件系统的高可信性和软件的可重用性。下面给出体系结构范式定义，并给出体系结构范式的有关定理。讨论之前，仿照数据库系统中的有关概念，首先给出码和范式的定义。

定义 14 设 $A=<C,O>$是一个软件体系结构，若 $\exists y \in C$，$\forall x \in C$，都有 $x \xrightarrow{\text{依赖}} y$，则称 y 为体系结构 A 的候选码(candidate key)，若候选码多于一个时，可选定其中一个作为主码(primary key)。

定义 15 设 $A=<C,O>$是一个软件体系结构，若任意一个构件都至少与另一个构件依赖，则称 A 满足第一范式，记为 1NF。

显然,1NF,是软件体系结构设计的最基本的要求。

定义 16 设 $A=<C,O>$是一个软件体系结构,若 A 中存在码,则称 A 满足第二范式,记为 2NF。

2NF 对软件体系结构的要求比 1NF 的要求要高。读者可以很容易找到满足 2NF 的软件体系结构例子。例如,C/S 就是一种典型的 2NF 体系结构。再如,为了提高系统的可靠性所采用的双机容错系统也是一种 2NF 体系结构,与一般 C/S 不同的是双机容错系统有两个码,其中一个为主码。

上述讨论并未区分依赖是何种依赖,实际上不同的依赖下软件体系结构的可靠性是不同的。基于实现依赖的体系结构可靠性明显低于基于事件依赖的体系结构。因为在基于实现依赖的体系结构中构件之间的实现不是完全独立的。一个构件失效会导致其他构件的失效,软件系统的可靠性将成几何级数下降。而基于事件依赖的体系结构的构件之间的实现是完全独立的,一个构件的失效并不影响其他构件的运行。下面从可靠性角度出发给出体系结构的可靠性范式 RNF。

定义 17 设 $A=<C,O>$是一个软件体系结构,若 A 满足第一范式,并且 C 中的构件都为事件依赖,则称 A 满足可靠性范式,记为 RNF。

显然,2NF 与 RNF 无从属关系。为了构建可靠的软件系统,一个自然的要求是设计满足 RNF 的软件体系结构。为此,给出软件体系结构范式转换定理。

引理 2 任何一个程序可以只用顺序、分支和循环结构来编码。

引理 2 来源于 Bohm 于 1996 年发表的一篇论文(见参考文献)。

定理 14 设 $A=<C,O>$是一个软件体系结构,若 A 满足第一范式,则总可以找到一种方法对 A 进行转换,使得转换后的 A 满足 RNF。

证明:设软件体系结构 $A=<C,O>$满足第一范式,分两种情况证明 A 可以被转化成满足 RNF 范式的体系结构。

(1) 构件之间只有事件依赖运算,这种情形下体系结构本身就是满足 RNF 的范式,无须证明。

(2) 构件之间存在实现依赖运算。采用 CPS(Continuation Passing Style)对实现依赖进行转换。所谓的 CPS 转换是把含有"使用"的构件分成若干程序段,在每一个程序段中不再含有对其他构件的"使用",重新构造程序段,使它们之间只有"激发"。下面分三种类型进行转换。

① 构件之间的实现依赖是顺序关系,即 $\exists a,x$,有 Use(a,x)。这时可以把构件 a 写成 $\{a_1,x,a_2\}$,如图 2-22 所示。其中,x 的输入是 a_1 的输出,a_2 的输入是 x 的输出,并且 a_1,x 和 a_2 的实现代码是相互独立的,为此总可以用消息控制程序的执行,即把所有的 Use 转换为 Invoke。该情况得证。

② 构件之间的实现依赖是分支关系,即 $\exists a,x,y$,有 Con→Use(a,x)或者 Con→Use(a,y),Con 是判断条件。这时可以把构件 a 写成$\{a_1,a_2,x,y,a_3\}$,如图 2-23 所示。a_1 是条件语句前的语句集合,a_2 为判断语句,a_3 为条件语句之后的语句集合,a_1、a_2、x、y、a_3 的实现代码是相互独立的。因此可以用判断事件是否发生来"激发" x 或 y 的执行,即把所有的 Use 转换为 Invoke。该情况得证。

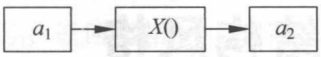


图 2-22　顺序关系

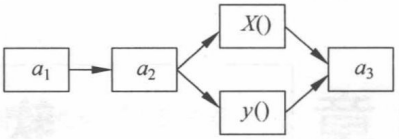


图 2-23　分支关系

③ 构件之间的实现依赖是循环关系。即 $\exists a, x$，有 Con→Use(a, x)→Con，Con 是循环条件。这时可以把构件 a 写成$\{a_1, x, a_2, a_3\}$，如图 2-24 所示。a_1 是循环语句前的语句集合，a_2 为循环语句，a_3 为循环语句之后的语句集合，a_1、x、a_2、a_3 的实现代码是相互独立的。因此可以用循环条件事件是否发生来“激发”x 的执行，即把所有的 Use 转换为 Invoke。该情况得证。

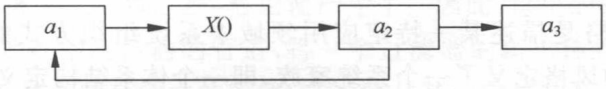


图 2-24　循环关系

由引理 2 得知定理 14 成立。证毕。

定理 14 的证明过程给出了满足 1NF 的软件体系结构如何转换成满足 RNF 的软件体系结构。

思　考　题

1. 软件体系结构模型是软件系统结构的表现形式，软件体系结构模型可以分为哪几种？具体是如何划分的？

2. 尝试用自己的语言介绍 Kruchten 的“4＋1”模型。

3. 为什么软件体系结构是必需的？软件体系结构在软件生命周期中的地位如何？

4. 在希赛公司的一个财务管理系统中，财务部要客户提供报销业务。客户先把发票交给会计建立一个凭证，然后由审核员审计该凭证，最后出纳给客户付钱。试根据体系结构建模，建立一组用例图来表示这种需求。

5. 软件体系结构是软件系统的支架，除了用能够 UML 等相关的工具进行建模外，为什么还需要对体系结构进行抽象化？

参考文献

第3章 软件体系结构风格

软件体系结构设计的一个核心问题是能否使用重复的体系结构模式，即能否达到体系结构级的软件重用。也就是说，能否在不同的软件系统中，使用同一体系结构。基于这个目的，学者们开始研究和实践软件体系结构的风格和类型问题。

软件体系结构风格是描述某一特定应用领域中系统组织方式的惯用模式(idiomatic paradigm)。体系结构风格定义了一个系统家族，即一个体系结构定义一个词汇表和一组约束。词汇表中包含一些构件和连接件类型，而这组约束指出系统是如何将这些构件和连接件组合起来的。体系结构风格反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。按这种方式理解，软件体系结构风格定义了用于描述系统的术语表和一组指导构建系统的规则。

对软件体系结构风格的研究和实践促进了对设计的重用，一些经过实践证实的解决方案也可以可靠地用于解决新的问题。体系结构风格的不变部分使不同的系统可以共享同一个实现代码。只要系统使用常用的、规范的方法来组织，就可使其他设计师很容易地理解系统的体系结构。例如，如果某人把系统描述为“客户-服务器”模式，则不必给出设计细节，相关人员立刻就会明白系统是如何组织和工作的。

3.1 经典软件体系结构风格

软件体系结构风格为大粒度的软件重用提供了可能。然而，对于应用体系结构风格来说，由于视点的不同，系统设计师有很大的选择余地。要为系统选择或设计某一个体系结构风格，必须根据特定项目的具体特点，进行分析比较后再确定，体系结构风格的使用几乎完全是特定的。

讨论体系结构风格时要回答的问题如下。

(1) 设计词汇表是什么？

(2) 构件和连接件的类型是什么？

(3) 可容许的结构模式是什么？

(4) 基本的计算模型是什么？

(5) 风格的基本不变性是什么？

(6) 此风格使用的常见例子是什么？

(7) 使用此风格的优缺点是什么？

(8) 此风格常见的特例是什么？

这些问题的回答包括体系结构风格的最关键的4要素内容，即提供一个词汇表、定义一

套配置规则、定义一套语义解释原则和定义对基于这种风格的系统所进行的分析。Garlan和Shaw根据此框架给出了通用体系结构风格的分类。

(1) 数据流风格：批处理序列、管道与过滤器。

(2) 调用/返回风格：主程序与子程序、面向对象风格、层次结构。

(3) 独立构件风格：进程通信、事件系统。

(4) 虚拟机风格：解释器、基于规则的系统。

(5) 仓库风格：数据库系统、超文本系统、黑板系统。

3.1.1 管道与过滤器

在管道与过滤器风格的软件体系结构中，每个构件都有一组输入和输出，构件读输入的数据流，经过内部处理，然后产生输出数据流。这个过程通常通过对输入流的变换及增量计算来完成，所以在输入被完全消费之前，输出便产生了。因此，这里的构件被称为过滤器，这种风格的连接件就像是数据流传输的管道，将一个过滤器的输出传到另一过滤器的输入。此风格特别重要的过滤器必须是独立的实体，它不能与其他的过滤器共享数据，而且过滤器不知道它上游和下游的标识。一个管道与过滤器网络输出的正确性并不依赖于过滤器进行增量计算过程的顺序。

图3-1是管道与过滤器风格的示意图。一个典型的管道与过滤器体系结构的例子是以UNIX Shell编写的程序。UNIX既提供一种符号，以连接各组成部分(UNIX的进程)，又提供某种进程运行时机制以实现管道。另一个著名的例子是传统的编译器。传统的编译器一直被认为是一种管道系统，在该系统中，一个阶段(包括词法分析、语法分析、语义分析和代码生成)的输出是另一个阶段的输入。

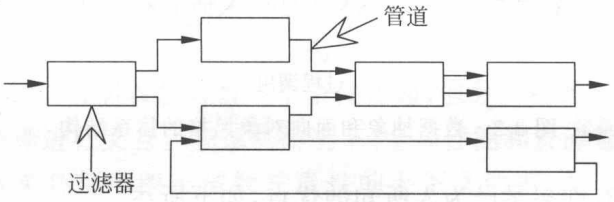


图3-1 管道与过滤器风格的体系结构

管道与过滤器风格的软件体系结构具有以下特点。

(1) 使得软构件具有良好的隐蔽性和高内聚、低耦合的特点。

(2) 允许设计师将整个系统的输入/输出行为看成是多个过滤器的行为的简单合成。

(3) 支持软件重用。只要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来。

(4) 系统维护和增强系统性能简单。新的过滤器可以添加到现有系统中；旧的可以被改进的过滤器替换掉。

(5) 允许对一些如吞吐量、死锁等属性的分析。

(6) 支持并行执行。每个过滤器是作为一个单独的任务完成，因此可与其他任务并行执行。

但是，这样的系统也存在着如下不利因素。

(1) 通常导致进程成为批处理的结构。这是因为虽然过滤器可增量式地处理数据,但它们是独立的,所以设计师必须将每个过滤器看成一个完整的从输入到输出的转换。

(2) 不适合处理交互的应用。当需要增量地显示改变时,这个问题尤为严重。

(3) 因为在数据传输上没有通用的标准,每个过滤器都增加了解析和合成数据的工作,这样就导致了系统性能下降,并增加了编写过滤器的复杂性。

3.1.2 数据抽象和面向对象系统

抽象数据类型概念对软件系统有着重要作用,目前软件界已普遍转向使用面向对象系统。这种风格建立在数据抽象和面向对象的基础上,数据的表示方法和它们的相应操作封装在一个抽象数据类型或对象中。这种风格的构件是对象,或者说是抽象数据类型的实例。对象是一种被称作管理者的构件,因为它负责保持资源的完整性。对象是通过函数和过程的调用来交互的。

图 3-2 是数据抽象和面向对象风格的示意图。

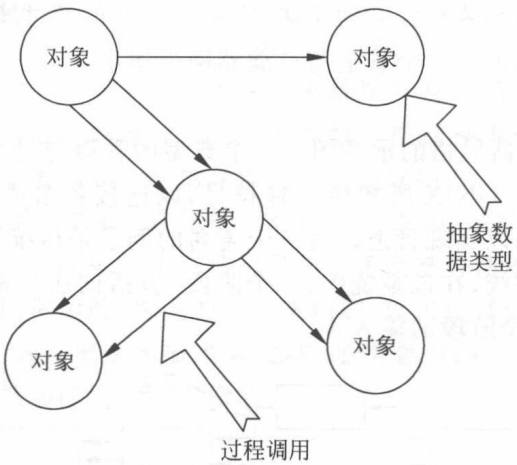


图 3-2 数据抽象和面向对象风格的体系结构

面向对象的系统有许多早已为人所知的优点,如下所述。

(1) 因为对象对其他对象隐藏它的标识,所以可以改变一个对象的标识,而不影响其他的对象。

(2) 设计师可将一些数据存取操作的问题分解成一些交互的代理程序的集合。

但是,面向对象的系统也存在着如下问题。

(1) 为了使一个对象和另一个对象通过过程调用等进行交互,必须知道对象的标识。只要一个对象的标识改变了,就必须修改所有其他明确调用它的对象。

(2) 必须修改所有显式调用它的其他对象,并消除由此带来的一些副作用。例如,如果 A 使用了对象 B,C 也使用了对象 B,那么,C 对 B 的使用所造成的对 A 的影响可能是料想不到的。

3.1.3 基于事件的系统

基于事件的系统风格的思想是:构件不直接调用一个过程,而是触发或广播一个或多个事件。系统中其他构件中的过程在一个或多个事件中注册,当一个事件被触发,系统自动

调用在这个事件中注册的所有过程，这样，一个事件的触发就导致了另一模块中的过程的调用。因此，该风格也称隐式调用。

从体系结构上说，这种风格的构件是一些模块，这些模块既可以是一些过程，又可以是一些事件的集合。过程可以用通用的方式调用，也可以在系统事件中注册一些过程，当发生这些事件时，过程被调用。

基于事件的隐式调用风格的主要特点是：事件的触发者并不知道哪些构件会被这些事件影响。这样不能假定构件的处理顺序，甚至不知道哪些过程会被调用，因此，许多隐式调用的系统也包含显式调用作为构件交互的补充形式。

支持基于事件的隐式调用的应用系统很多。例如，在编程环境中用于集成各种工具，在数据库管理系统中确保数据的一致性约束，在用户界面系统中管理数据，以及在编辑器中支持语法检查。例如，在某系统中，编辑器和变量监视器可以登记相应 Debugger 的断点事件。当 Debugger 在断点处停下时，它声明该事件，由系统自动调用处理程序，如编辑程序可以卷屏到断点，变量监视器刷新变量数值。而 Debugger 本身只声明事件，并不关心哪些过程会启动，也不关心这些过程做什么处理。

隐式调用系统的主要优点如下。

(1) 为软件重用提供了强大的支持。当需要将一个构件加入现存系统中时，只需要将它注册到系统的事件中。

(2) 为改进系统带来了方便。当用一个构件代替另一个构件时，不会影响其他构件的接口。

隐式调用系统的主要缺点如下。

(1) 构件放弃了对系统计算的控制。一个构件触发一个事件时，不能确定其他构件是否会响应它。而且即使它知道事件注册了哪些构件的过程，它也不能保证这些过程被调用的顺序。

(2) 数据交换的问题。有时数据可被一个事件传递，但另一些情况下，基于事件的系统必须依靠一个共享的仓库进行交互。在这些情况下，全局性能和资源管理便成了问题。

(3) 既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理存在问题。

3.1.4 分层系统

层次系统组织成一个层次结构，每一层为其上层服务，并作为其下层的客户。在一些层次系统中，除了一些精心挑选的输出函数外，内部的层只对相邻的层可见。这样的系统中构件在一些层实现了虚拟机(在另一些层次系统中层是部分不透明的)。连接件通过决定层间如何交互的协议来定义，拓扑约束包括对相邻层间交互的约束。

这种风格支持基于可增加抽象层的设计。这样，允许将一个复杂问题分解成一个增量步骤序列的实现。由于每一层最多影响两层，同时只要给相邻层提供相同的接口，允许每层用不同的方法实现，同样为软件重用提供了强大的支持。

图 3-3 是层次系统风格的示意图。层次系统最广泛的应用是分层通信协议。在这一应用领域，每一层提供一个抽象的功能，作为其上层通信的基础。较低的层次定义低层的交互，底层通常只定义硬件物理连接。

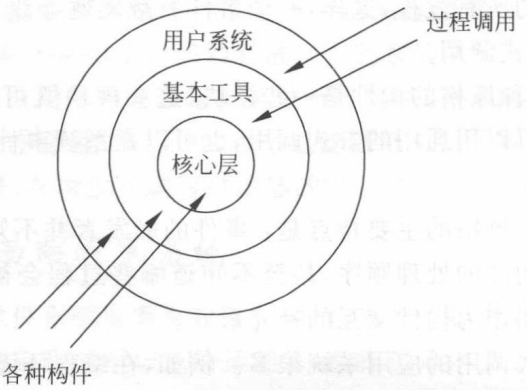


图 3-3　层次系统风格的体系结构

层次系统有许多可取的属性,如下所述。

(1) 支持基于抽象程度递增的系统设计,使设计师可以把一个复杂系统按递增的步骤进行分解。

(2) 支持功能增强。因为每一层至多和相邻的上下层交互,因此功能的改变最多影响相邻的上下层。

(3) 支持重用。只要提供的服务接口定义不变,同一层的不同实现可以交换使用。这样,就可以定义一组标准的接口,而允许各种不同的实现方法。

但是,层次系统也有其不足之处,如下所述。

(1) 并不是每个系统都可以很容易地划分为分层的模式,甚至即使一个系统的逻辑结构是层次化的,出于对系统性能的考虑,系统设计师不得不把一些低级或高级的功能综合起来。

(2) 很难找到一个合适的、正确的层次抽象方法。

3.1.5　仓库系统及知识库

在仓库(repository)风格中,有两种不同的构件:中央数据结构说明当前状态,独立构件在中央数据存储上执行,仓库与外构件间的相互作用在系统中会有大的变化。

控制原则的选取产生两个主要的子类。若输入流中某类时间触发进程执行的选择,则仓库是一传统型数据库;若中央数据结构的当前状态触发进程执行的选择,则仓库是一黑板系统。

图 3-4 是黑板系统的组成。黑板系统的传统应用是信号处理领域,如语音和模式识别,另一应用是松耦合代理数据共享存取。

从图 3-4 中可以看出,黑板系统主要由以下 3 部分组成。

(1) 知识源。知识源中包含独立的、与应用程序相关的知识。知识源之间不直接进行通信,它们之间的交互只通过黑板完成。

(2) 黑板数据结构。黑板数据是按照与应用程序相关的层次来组织的解决问题的数据,知识源通过不断地改变黑板数据来解决问题。

(3) 控制。控制完全由黑板的状态驱动,黑板状态的改变决定使用的特定知识。

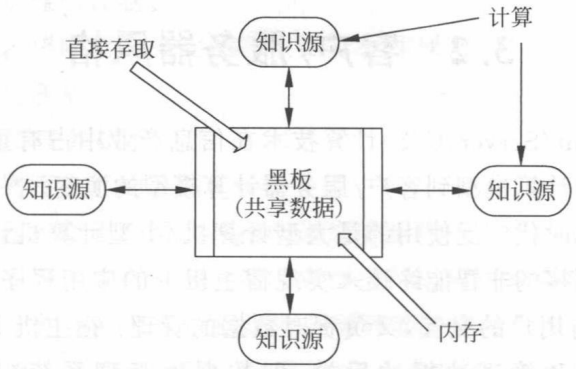


图 3-4　黑板系统的组成

3.1.6　C2 风格

C2 体系结构风格可以概括为：通过连接件绑定在一起的，按照一组规则运作的并行构件网络。C2 体系结构风格中的系统组织规则如下。

(1) 系统中的构件和连接件都有一个顶部和一个底部。

(2) 构件的顶部应连接到某连接件的底部，构件的底部则应连接到某连接件的顶部，而构件与构件之间的直接连接是不允许的。

(3) 一个连接件可以和任意数目的其他构件和连接件连接。

(4) 当两个连接件进行直接连接时，必须由其中一个的底部到另一个的顶部。

图 3-5 是 C2 体系结构风格的示意图。图中构件与连接件之间的连接体现了 C2 体系结构风格中构建系统的规则。

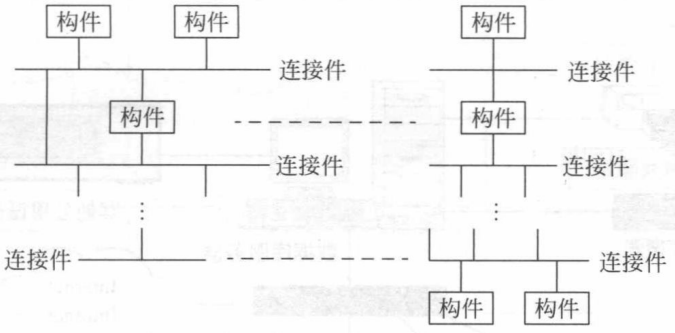


图 3-5　C2 体系结构风格

C2 体系结构风格是最常用的一种软件体系结构风格。从 C2 体系结构风格的组织规则和结构图中可以得出，C2 体系结构风格具有以下特点。

(1) 系统中的构件可实现应用需求，并能将任意复杂度的功能封装在一起。

(2) 所有构件之间的通信是通过以连接件为中介的异步消息交换机制来实现的。

(3) 构件相对独立，构件之间依赖性较少。系统中不存在某些构件将在同一地址空间内执行，或某些构件共享特定控制线程之类的相关性假设。

3.2 客户/服务器风格

客户/服务器(Client/Server,C/S)计算技术在信息产业中占有重要的地位。网络计算经历了从基于宿主机的计算模型到客户/服务器计算模型的演变。

在集中式计算技术时代广泛使用的是大型计算机/小型计算机计算模型。它是通过一台物理上与宿主机相连接的非智能终端来实现宿主机上的应用程序。在多用户环境中,宿主机应用程序既负责与用户的交互,又负责对数据的管理:宿主机上的应用程序一般也分为与用户交互的前端和管理数据的后端,即数据库管理系统(DataBase Management System,DBMS)。集中式的系统使用户能共享贵重的硬件设备,如磁盘机、打印机、调制解调器等。但随着用户的增多,对宿主机能力的要求很高,而且开发者必须为每个新的应用重新设计同样的数据管理构件。

20世纪80年代以后,集中式结构逐渐被以PC为主的微型计算机网络所取代。个人计算机和工作站的采用,彻底改变了协作计算模型,从而导致了分散的个人计算模型的产生。一方面,由于大型计算机系统固有的缺陷,如缺乏灵活性、无法适应信息量急剧增长的需求,并为整个企业提供全面的解决方案等;另一方面,由于微处理器的日新月异,其强大的处理能力和低廉的价格使微型计算机网络迅速发展,已不仅仅是简单的个人系统,这便形成了计算机界的向下规模化(downsizing)。其主要优点是用户可以选择适合自己需要的工作站、操作系统和应用程序。

C/S软件体系结构是基于资源不对等,且为实现共享而提出来的,是20世纪90年代成熟起来的技术,C/S体系结构定义了工作站如何与服务器相连,以实现数据和应用分布到多个处理机上。C/S体系结构有三个主要组成部分:数据库服务器、客户应用程序和网络,如图3-6所示。

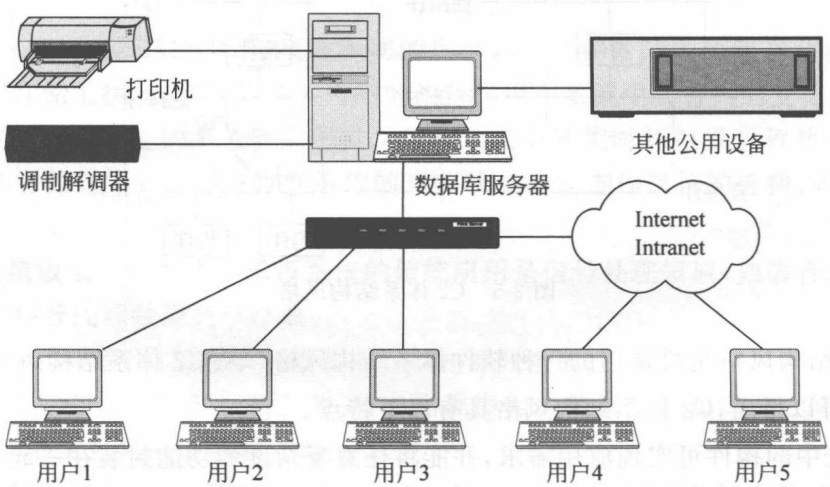


图3-6 C/S体系结构示意图

服务器负责有效地管理系统的资源,其任务集中于以下几项。

(1) 数据库安全性的要求。

(2) 数据库访问并发性的控制。

(3) 数据库前端的客户应用程序的全局数据完整性规则。

(4) 数据库的备份与恢复。

客户应用程序的主要任务如下。

(1) 提供用户与数据库交互的界面。

(2) 向数据库服务器提交用户请求并接收来自数据库服务器的信息。

(3) 利用客户应用程序对存在于客户端的数据执行应用逻辑要求。

网络通信软件的主要作用是完成数据库服务器和客户应用程序之间的数据传输。

C/S 体系结构将应用一分为二，服务器(后台)负责数据管理，客户机(前台)完成与用户的交互任务。服务器为多个客户应用程序管理数据，而客户程序发送、请求和分析从服务器接收的数据，这是一种“胖客户机(fat client)”，“瘦服务器(thin server)”的体系结构。其数据流图如图 3-7 所示。

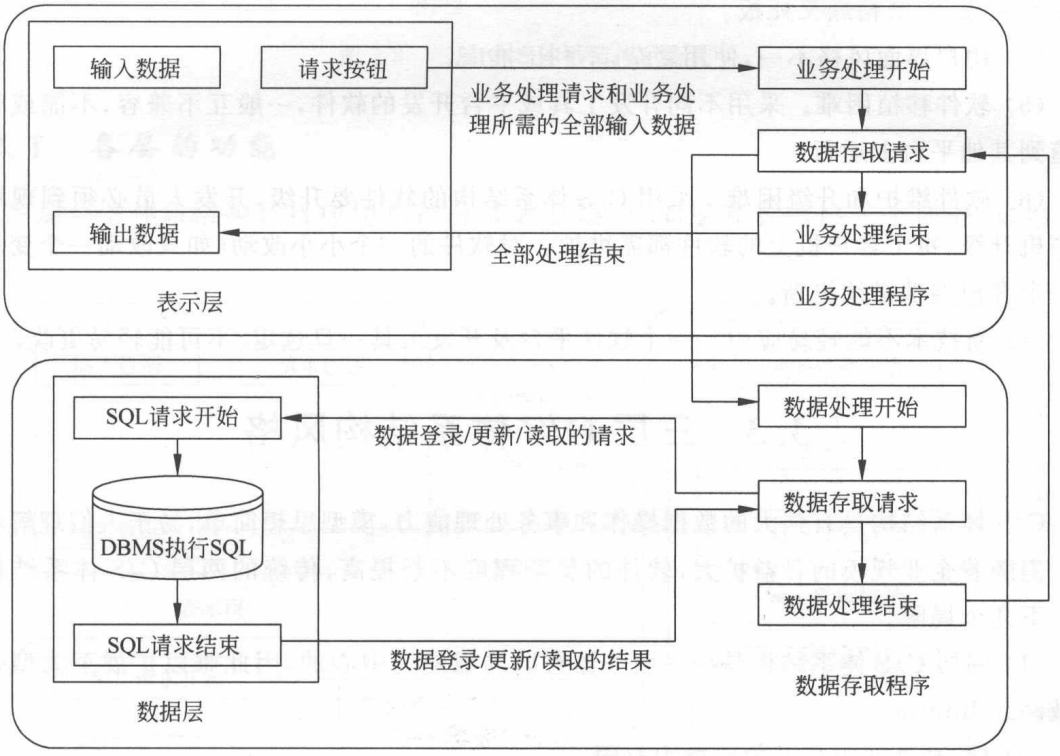


图 3-7 C/S 体系结构的一般处理流程

在一个 C/S 体系结构的软件系统中，客户应用程序是针对一个小的、特定的数据集，如一个表的行来进行操作，而不是像文件服务器那样针对整个文件进行操作，对某一条记录进行封锁，而不是对整个文件进行封锁，因此保证了系统的并发性，并使网络上传输的数据量减到最少，从而改善了系统的性能。

C/S 体系结构的优点主要在于系统的客户应用程序和服务器构件分别运行在不同的计算机上，系统中每台服务器都可以适合各构件的要求，这对于硬件和软件的变化显示出极大的适应性和灵活性，而且易于对系统进行扩充和缩小。在 C/S 体系结构中，系统中的功能

构件充分隔离,客户应用程序的开发集中于数据的显示和分析,而数据库服务器的开发则集中于数据的管理,不必在每一个新的应用程序中都要对一个 DBMS 进行编码。将大的应用处理任务分布到许多通过网络连接的低成本计算机上,以节约大量费用。

C/S 体系结构具有强大的数据操作和事务处理能力,模型思想简单,易于人们理解和接受。但随着企业规模的日益扩大,软件的复杂程度不断提高,C/S 体系结构逐渐暴露了以下缺点:

(1) 开发成本较高。C/S 体系结构对客户端软硬件配置要求较高,尤其是软件的不断升级,对硬件要求不断提高,增加了整个系统的成本,且客户端变得越来越臃肿。

(2) 客户端程序设计复杂。采用 C/S 体系结构进行软件开发,大部分工作量放在客户端的程序设计上,客户端显得十分庞大。

(3) 信息内容和形式单一。传统应用一般为事务处理,界面基本遵循数据库的字段解释,开发之初就已确定,而且不能随时截取办公信息、档案等外部信息,用户获得的只是单纯的字符和数字,既枯燥又死板。

(4) 用户界面风格不一,使用繁杂,不利于推广。

(5) 软件移植困难。采用不同开发工具或平台开发的软件,一般互不兼容,不能或很难移植到其他平台上运行。

(6) 软件维护和升级困难。采用 C/S 体系结构的软件要升级,开发人员必须到现场为客户机升级,每个客户机上的软件都需维护。对软件的一个小小改动(如只改动一个变量),每一个客户端都必须更新。

(7) 新技术不能轻易应用。一个软件平台及开发工具一旦选定,不可能轻易更改。

3.3 三层 C/S 体系结构风格

C/S 体系结构具有强大的数据操作和事务处理能力,模型思想简单,易于人们理解和接受。但随着企业规模的日益扩大,软件的复杂程度不断提高,传统的两层 C/S 体系结构存在以下几个局限。

(1) 两层 C/S 体系结构是单一服务器且以局域网为中心的,因此难以扩展至大型企业广域网或 Internet。

(2) 软、硬件的组合及集成能力有限。

(3) 客户机的负荷太重,难以管理大量的客户机,系统的性能容易变坏。

(4) 数据安全性不好。因为客户端程序可以直接访问数据库服务器,那么,在客户端计算机上的其他程序也可想办法访问数据库服务器,从而使数据库的安全性受到威胁。

由于两层 C/S 体系结构的这些局限,三层 C/S 体系结构应运而生,其结构如图 3-8 所示。

与两层 C/S 体系结构相比,三层 C/S 体系结构中,增加了一个应用服务器。将整个应用逻辑驻留在应用服务器上,只有表示层存在于客户机上,这种结构被称为瘦客户机(thin client)。

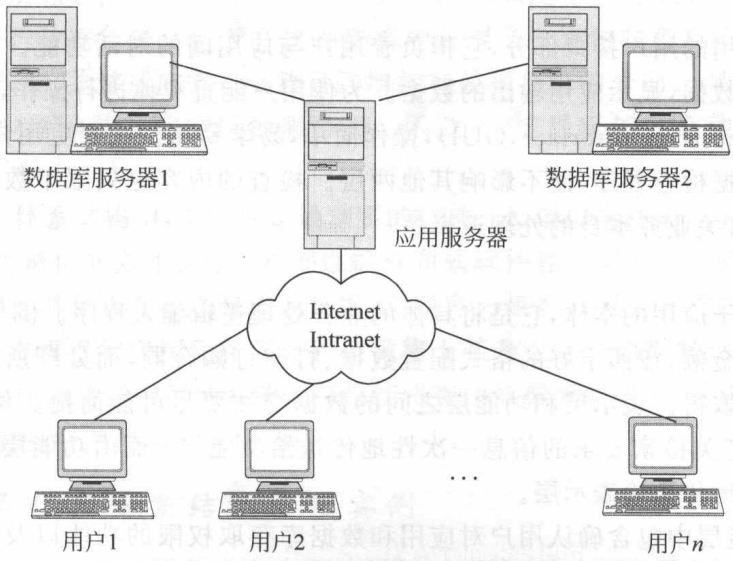


图 3-8　三层 C/S 体系结构示意图

3.3.1　各层的功能

三层 C/S 体系结构是将应用功能分成表示层、功能层和数据层三个部分，如图 3-9 所示。

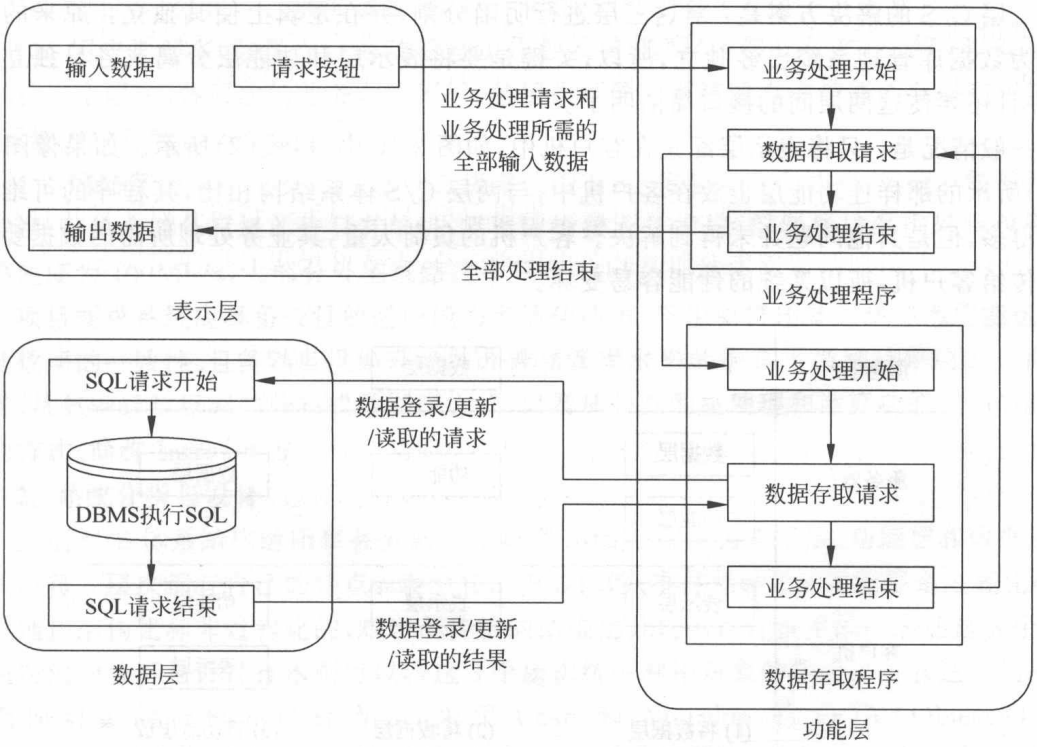


图 3-9　三层 C/S 体系结构的一般处理流程

1. 表示层

表示层是应用的用户接口部分,它担负着用户与应用间的对话功能。它用于检查用户从键盘等输入的数据,显示应用输出的数据。为使用户能直观地进行操作,一般要使用图形用户界面(Graphic User Interface,GUI),操作简单,易学易用。在变更用户界面时,只需改写显示控制和数据检查程序,而不影响其他两层。检查的内容也只限于数据的形式和取值的范围,不包括有关业务本身的处理逻辑。

2. 功能层

功能层相当于应用的本体,它是将具体的业务处理逻辑编入程序。例如,在制作订购合同时要计算合同金额,按照定好的格式配置数据、打印订购合同,而处理所需的数据则要从表示层或数据层取得。表示层和功能层之间的数据交往要尽可能简捷。例如,用户检索数据时,要设法将有关检索要求的信息一次性地传送给功能层;而由功能层处理过的检索结果数据也一次性地传送给表示层。

通常,在功能层中包含确认用户对应用和数据库存取权限的功能以及记录系统处理日志的功能。功能层的程序多半是用可视化编程工具开发的,也有使用 COBOL 和 C 语言的。

3. 数据层

数据层就是数据库管理系统,负责管理对数据库数据的读写。数据库管理系统必须能迅速执行大量数据的更新和检索。现在的主流是关系型数据库管理系统(RDBMS),因此,一般从功能层传送到数据层的要求大都使用 SQL。

三层 C/S 的解决方案是:对这三层进行明确分割,并在逻辑上使其独立。原来的数据层作为数据库管理系统已经独立,所以,关键是要将表示层和功能层分离成各自独立的程序,并且还要使这两层间的接口简洁明了。

一般情况是:只将表示层配置在客户机中,如图 3-10 中(1)或(2)所示。如果像图 3-10 中(3)所示的那样连功能层也放在客户机中,与两层 C/S 体系结构相比,其程序的可维护性要好得多,但是其他问题并未得到解决。客户机的负荷太重,其业务处理所需的数据要从服务器传给客户机,所以系统的性能容易变坏。

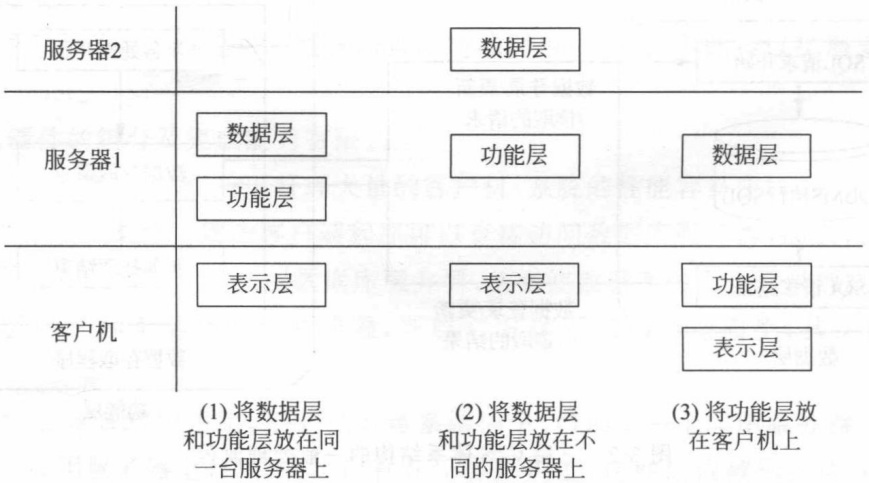


图 3-10 三层 C/S 物理结构比较

如果将功能层和数据层分别放在不同的服务器中，如图 3-10 中(2)所示，则服务器和服务器之间也要进行数据传送。由于在这种形态中三层是分别放在各自不同的硬件系统上的，所以灵活性很高，能够适应客户机数目的增加和处理负荷的变动。例如，在追加新业务处理时，可以相应增加装载功能层的服务器。因此，系统规模越大这种形态的优点就越显著。

在三层 C/S 体系结构中，中间件是最重要的构件。所谓中间件是一个用 API 定义的软件层，是具有强大通信能力和良好可扩展性的分布式软件管理框架。它的功能是在客户机和服务器或者服务器和服务器之间传送数据，实现客户机群和服务器群之间的通信。其工作流程是：在客户机里的应用程序需要驻留网络上某个服务器的数据或服务时，搜索此数据的 C/S 应用程序须访问中间件系统。该系统将查找数据源或服务，并在发送应用程序请求后重新打包响应，将其传送回应用程序。

3.3.2 三层 C/S 体系结构应用实例

本小节通过某石油管理局劳动管理系统的设计与开发，来介绍三层 C/S 体系结构的应用。

1. 系统背景介绍

该石油管理局是国有特大型企业，其劳动管理信息系统(Management Information System，MIS)具有较强的特点。

(1) 信息量大。须存储并维护全油田近 20 万名职工的基本信息以及其他各种管理信息。

(2) 单位多，分布广。系统涵盖 70 多个单位，分布范围 8 万余平方千米。

(3) 用户类型多、数量大。劳动管理工作涉及管理局(一级)、厂矿(二级)和基层大队(三级)三个层次，各层次的业务职责不同，各层次领导对系统的查询功能的要求和权限也不同，系统用户总数达 700 多个。

(4) 网络环境不断发展。70 多个二级单位中有 40 多个连入广域网，其他二级单位只有局域网，而绝大部分三级单位只有单机，需要陆续接入广域网，而已建成的广域网仅有骨干线路速度为 100Mb/s，大部分外围线路速率只有 64kb/s～2Mb/s。

项目要求系统应具备较强的适应能力和演化能力，不论单机还是网络环境均能运行，并保证数据的一致性，且能随着网络环境的改善和管理水平的提高平稳地从单机方式向网络方式，从集中式数据库向分布式数据库方式，以及从独立的应用程序方式向适应 Intranet 环境的方式(简称 Intranet 方式)演化。

2. 系统分析与设计

三层 C/S 体系结构运用事务分离的原则将 MIS 应用分为表示层、功能层和数据层 3 个层次。每一层次都有自己的特点：表示层是图形化的、事件驱动的；功能层是过程化的；数据层则是结构化和非过程化的，难以用传统的结构化分析与设计技术统一表达这 3 个层次。面向对象的分析与设计技术则可以将这 3 个层次统一利用对象的概念进行表达。当前有很多面向对象的分析和设计方法，采用 Coad 和 Yourdon 的 OOA(Object-Oriented Anaylzing，面向对象的分析)与 OOD(Object-Oriented Design，面向对象的设计)技术进行三层结构的分析与设计。

在 MIS 的三层结构中，中间的功能层是关键。运行 MIS 应用程序的最基本的任务就

是执行数千条定义业务如何运转的业务逻辑。一个业务处理过程就是一组业务处理规则的集合。中间层反映的是应用域模型,是MIS系统的核心内容。

Coad和Yourdon的OOA用于理解和掌握MIS应用域的业务运行框架,也就是应用域建模。OOA模型描述应用域中的对象,以及对象间各种各样的结构关系和通信关系。OOA模型有两个用途。首先,每个软件系统都建立在特定的现实世界中,OOA模型就是用来形式化该现实世界的"视图"。它建立起各种对象,分别表示软件系统主要的组织结构以及现实世界强加给软件系统的各种规则和约束条件。其次,给定一组对象,OOA模型规定了它们如何协同才能完成软件系统所指定的工作。这种协同在模型中以表明对象之间通信方式的一组消息连接来表示。

OOA模型划分为5个层次或视图,分别如下。

(1) 对象-类层。表示待开发系统的基本构造块。对象都是现实世界中应用域概念的抽象。这一层是整个OOA模型的基础,在劳动管理信息系统中存在一百多个类。

(2) 属性层。对象所存储(或容纳)的数据称为对象的属性。类的实例之间互相约束,它们必须遵从应用域的某些限制条件或业务规则,这些约束称为实例连接。对象的属性和实例连接共同组成了OOA模型的属性层。属性层的业务规则是MIS中最易变化的部分。

(3) 服务层。对象的服务加上对象实例之间的消息通信共同组成了OOA模型的服务层。服务层中的服务包含业务执行过程中的一部分业务处理逻辑,也是MIS中容易改变的部分。

(4) 结构层。结构层负责捕捉特定应用域的结构关系。分类结构表示类属成员的构成,反映通用性和特殊性。组装结构表示聚合,反映整体和组成部分。

(5) 主题层。主题层用于将对象归类到各个主题中,以简化OOA模型。为了简化劳动管理信息系统,将整个系统按业务职能划分为13个主题,分别为:职工基本信息管理,工资管理,劳动组织计划管理,劳动定员定额管理,劳动合同管理,劳动统计管理,职工考核鉴定管理,劳动保险管理,劳动力市场管理,劳动政策查询管理,领导查询系统,系统维护管理和系统安全控制。

在OOD方法中,OOD体系结构以OOA模型为设计模型的雏形。OOD将OOA的模型作为OOD的问题论域部分(PDC),并增加其他3个部分:人机交互部分(HIC)、任务管理部分(TMC)和数据管理部分(DMC)。各部分与PDC一样划分为5个层次,但是针对系统的不同方面。OOD的任务是将OOA所建立的应用模型计算机化,OOD所增加的3个部分是为应用模型添加计算机的特征。

(1) 问题论域部分:以OOA模型为基础,包含执行基本应用功能的对象,可逐步细化,使其最终能解决实现限制、特性要求、性能缺陷等方面的问题,PDC封装了应用服务器功能层的业务逻辑。

(2) 人机交互部分:指定了用于系统的某个特定实现的界面技术,在系统行为和用户界面的实现技术之间架起了一座桥梁。HIC封装了客户层的界面表达逻辑。

(3) 任务管理部分:把有关特定平台的处理机制底层系统的其他部分隐藏了起来。在该项目中,利用TMC实现分布式数据库的一致性管理。在三层C/S体系结构中,TMC是应用服务器的一个组成部分。

(4) 数据管理部分：定义了与所有数据库技术接口的对象。DMC 同样是三层结构中应用服务器的一部分。由于 DMC 封装了数据库访问逻辑，使应用独立于特定厂商的数据库产品，便于系统的移植和分发。

OOD 的 4 个部分与三层结构的对应关系如图 3-11 所示。

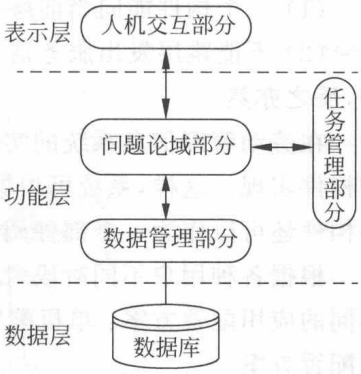


图 3-11　OOD 与三层 C/S 体系结构

3. 系统实现与配置

三层 C/S 体系结构提供了良好的结构扩展能力。三层结构在本质上是一种开发分布式应用程序的框架，在系统实现时可采用支持分布式应用的构件技术实现。

当前，有 3 种分布式构件标准：Microsoft 的 DCOM、OMG 的 CORBA 和 Sun 的 JavaBeans。这 3 种构件标准各有特点。考虑到在该项目应用环境的客户端和应用服务器均采用 Windows 98/2000 和 Windows NT/2000，采用在这些平台上具有较高效率的支持 DCOM 的 ActiveX 方式实现客户端和应用服务器的程序。

ActiveX 可将程序逻辑封装，并划分到进程内、本地或远程进程外执行。为了将应用程序划分到不同的构件里面，引入“服务模型”的概念。服务模型提供了一种逻辑性(非物理性)的方式，如图 3-12 所示。

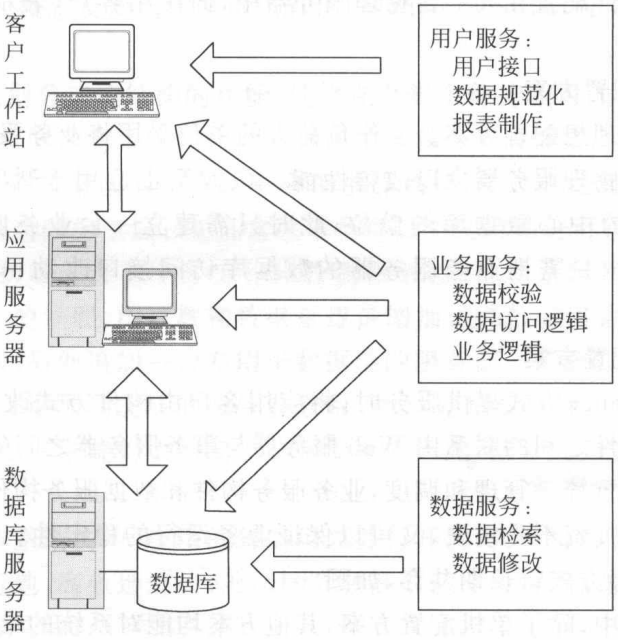


图 3-12　服务模型结构图

“服务模型”是对所创建的构件进行分组的一种逻辑方式，这种模型与语言无关。服务模型基于这样一个概念：每个构件都是一系列服务的集合，这些服务由构件提供给其他对象。

创建应用方案的时候，共有 3 种类型的服务可供选用：用户服务、业务服务以及数据服

务。每种服务类型都对应于三层C/S体系结构中的某一层。在服务模型里,为实现构件间的相互通信,必须遵守以下两条基本的规则。

(1) 一个构件能向当前层及构件层上下的任何一个层的其他构件发出服务请示。

(2) 不能跳层发出服务请求。用户服务层内的构件不能直接与数据服务层内的构件通信,反之亦然。

在劳动管理信息系统的实现中,将PDC的13个子系统以及TMC和DMC分别用单独的构件实现。这样,系统可根据各单位的实际情况进行组合,实现系统的灵活配置。而且这些构件还可以作为一个部件用于构造新的更大的MIS。

根据各种用户不同阶段对系统的不同需求以及系统未来的演化可能,拟定了如下几种不同的应用配置方案:单机配置方案、单服务器配置方案、业务服务器配置方案和事务服务器配置方案。

1) 单机配置方案

对于未能连入广域网的二级单位和三级单位中的单机用户,将三层结构的所有构件连同数据库系统均安装在同一台机器上,与中心数据库的数据交换采用拨号上网或交换磁介质的方式完成。当它连入广域网时,可根据业务量情况采用单服务器配置方案或业务服务器配置方案。

2) 单服务器配置方案

对于已建有局域网的二级单位,当建立了本地数据库且其系统负载不大时,可将业务服务构件与数据服务构件配置在同一台物理服务器中,而应用客户(表示层)构件在各用户的计算机内安装。

3) 业务服务器配置方案

这是三层结构的理想配置方案。工作负荷大的单位采用将业务服务构件和数据服务构件分别配置于独立的物理服务器内以改善性能。该方案也适用于暂时不建立自己的数据库,而使用局劳资处的中心数据库的单位,此时只需建立一台业务服务器。该单位需要建立自己的数据库时,只需把业务服务器的数据库访问接口改动一下,其他方面无须任何改变。

4) 事务服务器配置方案

当系统采用Intranet方式提供服务时,将应用客户由构件方式改为Web页面方式,应用客户与业务服务构件之间的联系由Web服务器与事务服务器之间的连接提供,事务服务器对业务服务构件进行统一管理和调度,业务服务构件和数据服务构件不必做任何修改,这样既可以保证以前的投资不受损失,又可以保证业务运行的稳定性。向Intranet方式的转移是渐进的,两种运行方式将长期共存,如图3-13所示。

在上述各种方案中,除了单机配置方案,其他方案均能对系统的维护和安全管理提供极大方便。任何应用程序的更新只需在对应的服务器上更新有关的构件即可。安全性则由在服务器上对操作应用构件的用户进行相应授权来保障。由于任何用户不直接拥有对数据库的访问权限,其操作必须通过系统提供的构件进行,这样就保证了系统的数据不被滥用,具有很高的安全性。同时,三层C/S体系结构具有很强的可扩展性,可以根据需要选择不同的配置方案,并且在应用扩展时方便地转移为另一种配置。

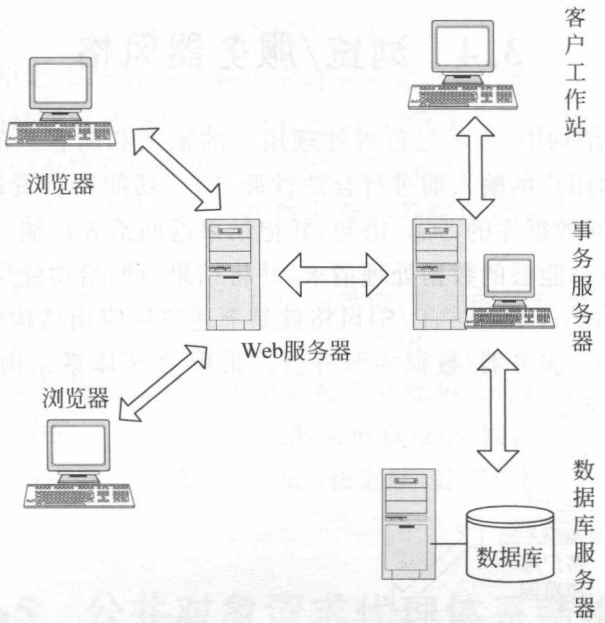


图 3-13　向 Intranet 方式的转移

3.3.3　三层 C/S 体系结构的优点

根据三层 C/S 的概念及使用实例可以看出，与两层 C/S 体系结构相比，三层 C/S 体系结构具有以下优点。

(1) 允许合理地划分三层结构的功能，使之在逻辑上保持相对独立性，从而使整个系统的逻辑结构更为清晰，能提高系统和软件的可维护性和可扩展性。

(2) 允许更灵活有效地选用相应的平台和硬件系统，使之在处理负荷能力上与处理特性上分别适应于结构清晰的三层；并且这些平台和各个组成部分可以具有良好的可升级性和开放性。例如，最初用一台 UNIX 工作站作为服务器，将数据层和功能层都配置在这台服务器上。随着业务的发展，用户数和数据量逐渐增加，这时，就可以将 UNIX 工作站作为功能层的专用服务器，另外追加一台专用于数据层的服务器。若业务进一步扩大，用户数进一步增加，则可以继续增加功能层的服务器数目，用以分割数据库。清晰、合理地分割三层结构并使其独立，可以使系统构成的变更非常简单。因此，被分成三层的应用基本上不需要修正。

(3) 三层 C/S 体系结构中，应用的各层可以并行开发，各层也可以选择各自最适合的开发语言。使之能并行地、高效地进行开发，达到较高的性能价格比；对每一层的处理逻辑的开发和维护也会更容易些。

(4) 允许充分利用功能层有效地隔离开表示层与数据层。未授权的用户难以绕过功能层而利用数据库工具或黑客手段去非法地访问数据层，这就为严格的安全管理奠定了坚实的基础；整个系统的管理层次也更加合理和可控制。

值得注意的是：三层 C/S 体系结构各层间的通信效率若不高，即使分配给各层的硬件能力很强，其作为整体来说也达不到所要求的性能。此外，设计时必须慎重考虑三层间的通信方法、通信频度及数据量。这和提高各层的独立性一样，是三层 C/S 体系结构的关键问题。

3.4 浏览/服务器风格

在三层C/S体系结构中,表示层负责处理用户的输入和向客户的输出(出于效率的考虑,它可能在向上传输用户的输入前进行合法性验证)。功能层负责建立数据库的连接,根据用户的请求生成访问数据库的SQL语句,并把结果返回给客户端。数据层负责实际的数据库存储和检索,响应功能层的数据处理请求,并将结果返回给功能层。

浏览/服务器(Browser-Server,B/S)风格就是上述三层应用结构的一种实现方式,其具体结构为:浏览器/Web服务器/数据库服务器。采用B/S体系结构的计算机应用系统的基本框架如图3-14所示。

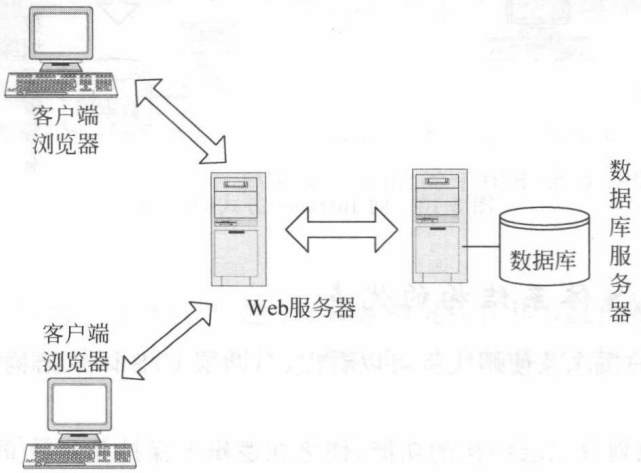


图3-14 B/S模式结构

B/S体系结构主要是利用不断成熟的WWW浏览器技术,结合浏览器的多种脚本语言,用通用浏览器就实现了原来需要复杂的专用软件才能实现的强大功能,并节约了开发成本。从某种程度上来说,B/S体系结构是一种全新的软件体系结构。

在B/S体系结构中,除了数据库服务器外,应用程序以网页形式存放于Web服务器上,用户运行某个应用程序时只需要在客户端上的浏览器中输入相应的网址,调用Web服务器上的应用程序并对数据库进行操作完成相应的数据处理工作,最后将结果通过浏览器显示给用户。可以说,在B/S模式的计算机应用系统中,应用(程序)在一定程度上具有集中特征。

基于B/S体系结构的软件,系统安装、修改和维护全在服务器端解决。用户在使用系统时,仅需要一个浏览器就可运行全部的模块,真正达到了"零客户端"的功能,很容易在运行时自动升级。B/S体系结构还提供了异种机、异种网、异种应用服务的联机、联网、统一服务的最现实的开放性基础。

B/S体系结构出现之前,管理信息系统的功能覆盖范围主要是组织内部。B/S体系结构的"零客户端"方式,使组织的供应商和客户(这些供应商和客户有可能是潜在的,也就是说可能是未知的)的计算机方便地成为管理信息系统的客户端,进而在限定的功能范围内查询组织相关信息,完成与组织的各种业务往来的数据交换和处理工作,扩大了组织计算机应用系统的功能覆盖范围,可以更加充分利用网络上的各种资源,同时应用程序维护的工作量

也大大减少。另外，B/S 体系结构的计算机应用系统与 Internet 的结合也使新的企业计算机应用(如电子商务，客户关系管理)的实现成为可能。

与 C/S 体系结构相比，B/S 体系结构也有许多不足之处，例如：

(1) B/S 体系结构缺乏对动态页面的支持能力，没有集成有效的数据库处理功能。

(2) B/S 体系结构的系统扩展能力差，安全性难以控制。

(3) 采用 B/S 体系结构的应用系统，在数据查询等响应速度上，要远远低于 C/S 体系结构。

(4) B/S 体系结构的数据提交一般以页面为单位，数据的动态交互性不强，不利于在线事务处理(OnLine Transaction Processing，OLTP)应用。

因此，虽然 B/S 体系结构的计算机应用系统有如此多的优越性，但由于 C/S 体系结构的成熟性且 C/S 体系结构的计算机应用系统网络负载较小，因此，未来一段时间内，将是 B/S 体系结构和 C/S 体系结构共存的情况。但是，很显然，计算机应用系统计算模式的发展趋势是向 B/S 体系结构转变。

3.5 公共对象请求代理体系结构

CORBA 是由 OMG 制定的一个工业标准，其主要目标是提供一种机制，使得对象可以透明地发出请求和获得应答，从而建立一个异质的分布式应用环境。

由于分布式对象计算技术具有明显优势，OMG 提出 CORBA 规范来适应该技术的进一步发展。1991 年，OMG 基于面向对象技术，给出了以对象请求代理(Object Request Broker，ORB)为中心的对象管理结构，如图 3-15 所示。

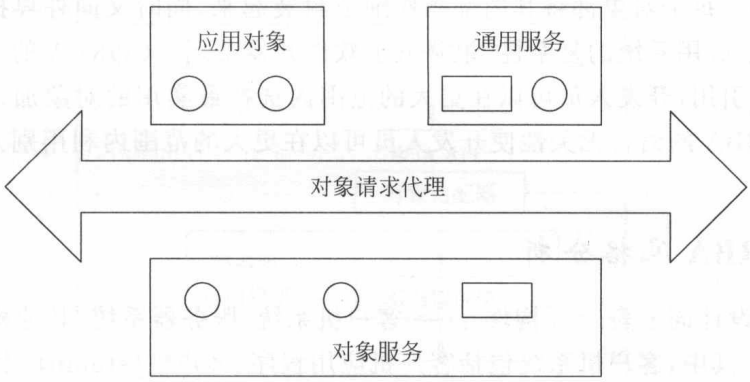


图 3-15 对象管理结构

在 OMG 的对象管理结构中，ORB 是一个关键的通信机制，它以实现互操作性为主要目标，处理对象之间的消息分布。对象服务实现基本的对象创建和管理功能，通用服务则使用对象管理结构所规定的类接口实现一些通用功能。

3.5.1 CORBA 技术规范

针对 ORB，OMG 进一步提出了 CORBA 技术规范，主要内容包括接口定义语言(Interface Definition Language，IDL)、接口池(Interface Repository，IR)、动态调用接口

(Dynamic Invocation Interface,DII)、对象适配器(Object Adapter,OA)等。

1. 接口定义语言

CORBA 利用 IDL 统一地描述服务器对象(向调用者提供服务的对象)的接口。IDL 本身也是面向对象的。它虽然不是编程语言,但它为客户对象(发出服务请求的对象)提供了语言的独立性,因为客户对象只需了解服务器对象的 IDL 接口,不必知道其编程语言。IDL 语言是 CORBA 规范中定义的一种中性语言,它用来描述对象的接口,而不涉及对象的具体实现。在 CORBA 中定义了 IDL 语言到 C、C++、SmallTalk 和 Java 语言的映射。

2. 接口池

CORBA 的接口池包括分布计算环境中所有可用的服务器对象的接口表示。它使动态搜索可用服务器的接口、动态构造请求及参数成为可能。

3. 动态调用接口

CORBA 的动态调用接口提供了一些标准函数以供客户对象动态创建请求、动态构造请求参数。客户对象将动态调用接口与接口池配合使用可实现服务器对象接口的动态搜索、请求及参数的动态构造与动态发送。当然,只要客户对象在编译之前能够确定服务器对象的 IDL 接口,CORBA 也允许客户对象使用静态调用机制。静态机制的灵活性虽不及动态机制,但其执行效率却胜过动态机制。

4. 对象适配器

在 CORBA 中,对象适配器用于屏蔽 ORB 内核的实现细节,为服务器对象的实现者提供抽象接口,以便他们使用 ORB 内部的某些功能。这些功能包括服务器对象的登录与激活、客户请求的认证等。

CORBA 定义了一种面向对象的软件构件构造方法,使不同的应用可以共享由此构造出来的软件构件。每个对象都将其内部操作细节封装起来,同时又向外界提供精确定义的接口,从而降低了应用系统的复杂性,也降低了软件开发费用。CORBA 的平台无关性实现了对象的跨平台引用,开发人员可以在更大的范围内选择最实用的对象加入到自己的应用系统之中。CORBA 的语言无关性使开发人员可以在更大的范围内利用别人的编程技能和成果。

3.5.2 CORBA 风格分析

CORBA 的设计词汇表 = [构件 ::= 客户机系统/服务器系统/其他构件; 连接件 ::= 请求/服务]。其中,客户机系统包括客户机应用程序、客户桩(stump)、上下文对象、接口仓库等构件,以及桩类型激发 API、动态激发 API 等连接件。服务器系统包括服务器应用程序方法库,服务器框架、对象请求代理等构件,以及对象适配器等连接件。CORBA 的体系结构模式如图 3-16 所示。

CORBA 体系结构中,客户机应用程序用桩类型激发 API 或者动态激发 API 向服务器发送请求。在服务器端接受方法调用请求,不进行参数引导,设置需要的上下文状态,激发服务器框架中的方法调度器,引导输出参数,并完成激发。服务器应用程序使用服务器端的服务部分,它包含某个对象的一个或者多个实现,用于满足客户机对指定对象上的某个操作的请求。

很明显,客户机系统是独立于服务器系统的,同样,服务器系统也独立于客户机系统。

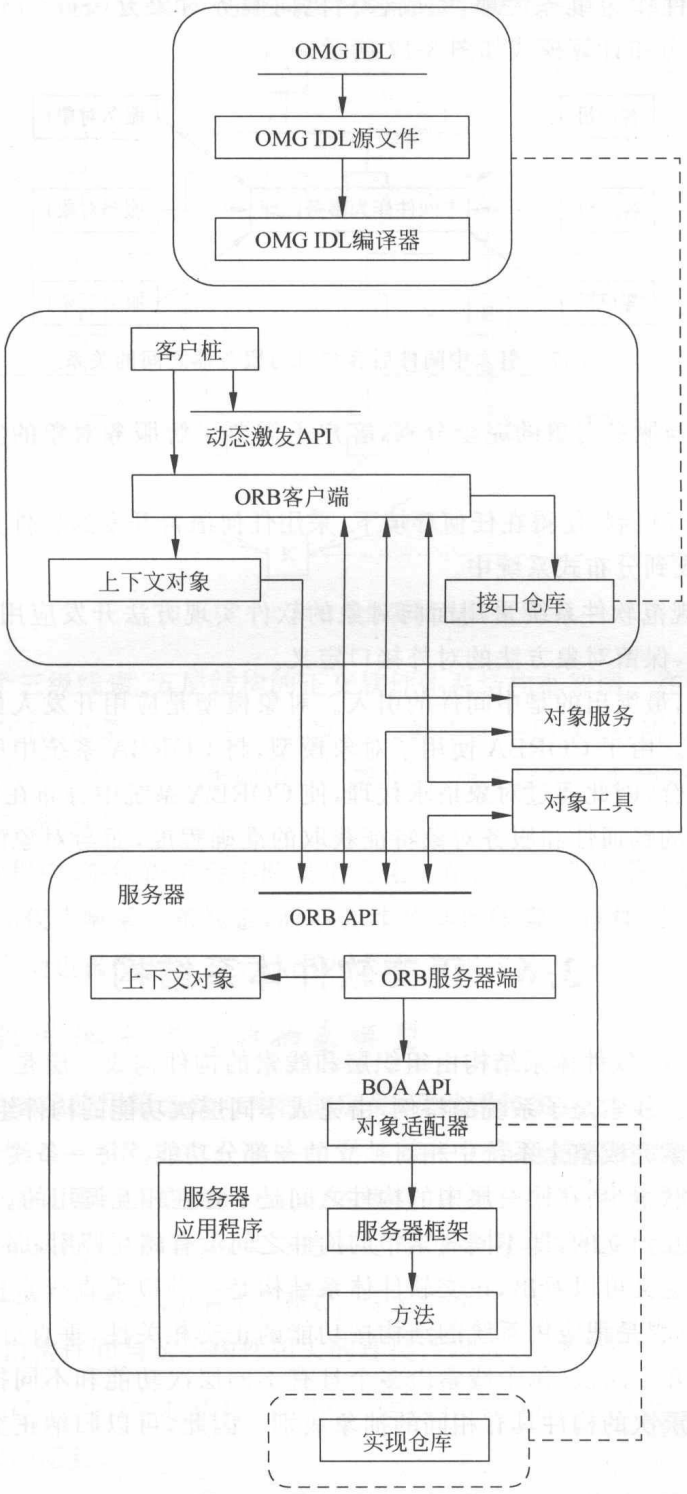


图 3-16　CORBA 的体系结构模式

CORBA 体系结构模式充分利用了现今软件技术发展的最新成果，在基于网络的分布式应用环境下实现应用软件的集成，使得面向对象的软件在分布、异构环境下实现可重用、可移植和互操作。其特点可以总结为如下几个方面。

(1) 引入中间件作为事务代理,完成客户机向服务对象方(Server)提出的业务请求。引入中间件概念后分布计算模式如图 3-17 所示。

图 3-17 引入中间件后客户机与服务器之间的关系

(2) 实现客户与服务对象的完全分离,客户不需要了解服务对象的实现过程以及具体位置。

(3) 提供软总线机制,使得在任何环境下、采用任何语言开发的软件只要符合接口规范的定义,均能够集成到分布式系统中。

(4) CORBA 规范软件系统采用面向对象的软件实现方法开发应用系统,实现对象内部细节的完整封装,保留对象方法的对外接口定义。

在以上特点中,最突出的是中间件的引入。对象模型是应用开发人员对客观事物属性和功能的具体抽象。由于 CORBA 使用了对象模型,将 CORBA 系统中所有的应用看成对象及相关操作的集合,因此通过对象请求代理,使 CORBA 系统中分布在网络中应用对象的获取只取决于网络的畅通性和服务对象特征获取的准确程度,而与对象的位置以及对象所处的设备环境无关。

3.6 正交软件体系结构

正交(orthogonal)软件体系结构由组织层和线索的构件构成。层是由一组具有相同抽象级别的构件构成。线索是子系统的特例,由完成不同层次功能的构件组成(通过相互调用来关联),每一条线索完成整个系统中相对独立的一部分功能。每一条线索的实现与其他线索的实现无关或关联很少,在同一层中的构件之间是不存在相互调用的。

如果线索是相互独立的,即不同线索中的构件之间没有相互调用,那么这个结构就是完全正交的。从以上定义可以看出,正交软件体系结构是一种以垂直线索构件族为基础的层次化结构,其基本思想是把应用系统的结构按功能的正交相关性,垂直分割为若干线索(子系统),线索又分为几个层次,每个线索由多个具有不同层次功能和不同抽象级别的构件构成。各线索的相同层次的构件具有相同的抽象级别。因此,可以归纳正交软件体系结构的主要特征如下。

(1) 正交软件体系结构由完成不同功能的 $n(n > 1)$个线索(子系统)组成。

(2) 系统具有 $m(m > 1)$个不同抽象级别的层。

(3) 线索之间是相互独立的(正交的)。

(4) 系统有一个公共驱动层(一般为最高层)和公共数据结构(一般为最低层)。

对于大型的、复杂的软件系统,其子线索(一级子线索)还可以划分为更低一级的子线索

(二级子线索),形成多级正交结构。正交软件体系结构的框架如图 3-18 所示。

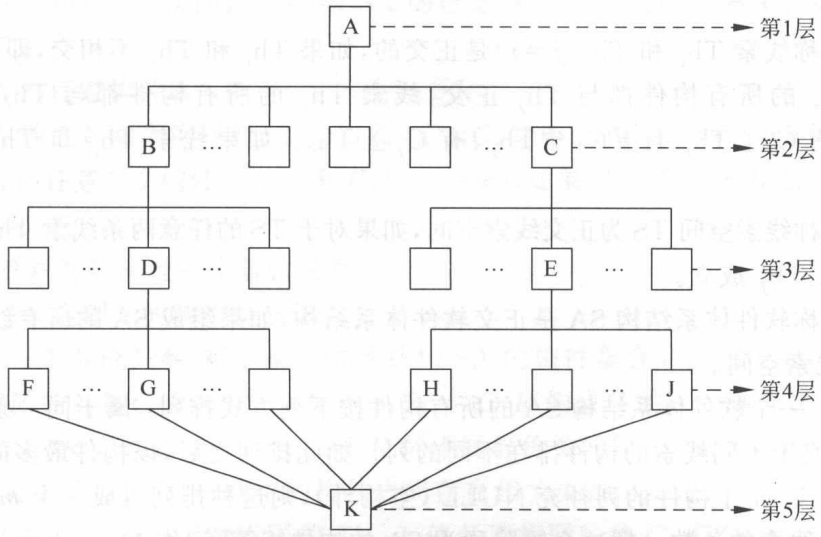


图 3-18　正交软件体系结构的框架

图 3-18 是一个三级线索、五层结构的正交软件体系结构框架图。在该图中,ABDFK 组成了一条线索,ACEJK 也是一条线索。因为 B、C 处于同一层次中,所以不允许互相调用;H、J 处于同一层次中,也不允许互相调用。一般来讲,第 5 层是一个物理数据库连接构件或设备构件,供整个系统公用。

在软件演化过程中,系统需求会不断发生变化。在正交软件体系结构中,因线索的正交性,每一个需求变动仅影响某一条线索,而不会涉及其他线索。这样,就把软件需求的变动局部化了,产生的影响也被限制在一定范围内,因此实现容易。

3.6.1　正交软件体系结构的抽象模型

这一小节中将继续使用第 2 章对构件的定义,在此不再赘述。

定义 1　连接件 $L=<R,G>$。这里 R 是连接件中角色进程组成的集合,通常集合中有等于或多于两个的元素;G 是胶水(Glue)的集合,该集合中只有一个元素。

定义 2　称两个构件 C_i 和 $C_j(j\neq i)$是相关的,如果存在连接件 L,使得 C_i 与 C_j 之间有连接,记作 $C_i \overset{L}{\longleftrightarrow} C_j$。否则,就称 C_i 和 C_j 是无关的。

定义 3　线索由构件和与这些构件相关的连接件组成,记作 Th。

$\mathrm{Th}=\{<C_i,C_{i+1}>,C_i$ 是构件,且 $\exists L_j$,使得 $C_i \overset{L_j}{\longleftrightarrow} C_{i+1}$ 成立,$0<i\leqslant n\}$。其中,n 称为软件体系结构的层数。

定义 4　称线索 Th_i 和 $\mathrm{Th}_j(j\neq i)$是相交的,如果存在构件 C,使得 $C\in \mathrm{Th}_i$ 且 $C\in \mathrm{Th}_j$ 成立。把所有这样的 C 的集合记作 S_{ij},即 $S_{ij}=\mathrm{Th}_i \cap \mathrm{Th}_j$。

定义 5　软件体系结构 SA 的所有线索组成一个集合,称该集合为线索空间。表示为 $\mathrm{TS}=(\mathrm{Th}_1,\mathrm{Th}_2,\cdots,\mathrm{Th}_m)$,其中 m 称为线索空间的维数,也就是一个软件体系结构中包含的垂直线索的条数。

定义6 称构件 C 与线索 Th 是正交的，如果对 $\forall C_i \in \mathrm{Th}$，$C$ 和 C_i 都是无关的。记作 $C \perp \mathrm{Th}$。

定义7 称线索 Th_i 和 $\mathrm{Th}_j (j \neq i)$ 是正交的，如果 Th_i 和 Th_j 不相交，即 $\mathrm{Th}_i \cap \mathrm{Th}_j = \phi$，且线索 Th_i 的所有构件都与 Th_j 正交，线索 Th_j 的所有构件都与 Th_i 正交。即对 $\forall C_i \in \mathrm{Th}_i$，有 $C_i \perp \mathrm{Th}_j$ 且 $\forall C_j \in \mathrm{Th}_j$，有 $C_j \perp \mathrm{Th}_i$。如果线索 Th_i 和 Th_j 正交，记作 $\mathrm{Th}_i \perp \mathrm{Th}_j$。

定义8 称线索空间 TS 为正交线索空间，如果对于 TS 的任意两条线索 Th_i 和 $\mathrm{Th}_j (j \neq i)$，都有 $\mathrm{Th}_i \perp \mathrm{Th}_j$ 成立。

定义9 称软件体系结构 SA 是正交软件体系结构，如果组成 SA 的所有线索的集合组成一个正交线索空间。

定义10 一个软件体系结构 SA 的所有构件按下列方式排列：属于同一条线索的构件排在同一列，属于不同线索的构件排在不同的列。如此排列之后，以构件最多的那一列为标准，设为 m，不足 m 个构件的列补充 NULL（空构件），则这种排列组成一个 $m \times n$ 矩阵，其中 n 是 SA 的线索的条数。把这个矩阵称为 SA 的构件矩阵，记作 $\boldsymbol{M}_{\mathrm{SA}}$，表示为：

$$\boldsymbol{M}_{\mathrm{SA}} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & & \vdots \\ C_{i1} & C_{i2} & \cdots & C_{in} \\ N & C_{(i+1)2} & \cdots & C_{(i+1)n} \\ N & N & \cdots & C_{(i+2)n} \\ \vdots & \vdots & & \vdots \\ N & N & \cdots & C_{mn} \end{pmatrix}$$

由 $\boldsymbol{M}_{\mathrm{SA}}$ 的表示可以看出，一个软件体系结构 SA 的构件矩阵在经过矩阵的若干次列变换后，最终会变换成一个非严格意义的上三角矩阵。

约定 NULL 构件与任意构件正交，即对任意的构件 C_i，都有 $\mathrm{NULL} \perp C_i$ 成立。

定义11 一个构件矩阵 $\boldsymbol{M}_{\mathrm{SA}}$ 被称为正交矩阵，如果对于任意两列元素组成的线索 Th_i 和 $\mathrm{Th}_j (j \neq i)$，都有 $\mathrm{Th}_i \perp \mathrm{Th}_j$ 成立。

定理1 在正交矩阵 $\boldsymbol{M}_{\mathrm{SA}}$ 中，任意两个元素 C_{ij} 和 $C_{kl} (j \neq l)$ 都是无关的。

证明：用反证法。假设在 $\boldsymbol{M}_{\mathrm{SA}}$ 中，存在两个元素 C_{ij} 和 $C_{kl} (j \neq l)$，且 C_{ij} 和 C_{kl} 相关。因为 $j \neq l$，根据定义11，在体系结构 SA 中，C_{ij} 和 C_{kl} 分别属于两条不同的线索 Th_j 和 Th_l。再根据定义8，可知 Th_j 和 Th_l 不是正交的，这与正交矩阵的定义矛盾。

3.6.2 软件体系结构的正交化

在实际的应用系统中，并不是所有的软件体系结构都是正交结构，为了使用正交结构的优点，必须对这些非正交软件体系结构进行正交化。

定义12 对任意两条线索 Th_i 和 $\mathrm{Th}_j (j \neq i)$，如果 $\exists C_i \in \mathrm{Th}_i - S_{ij}$，$C_j \in \mathrm{Th}_j - S_{ij}$，且 $\exists L$，使得 $C_i \xleftrightarrow{L} C_j$ 成立，则可把这两个构件进行改造为 $C_{i'}$ 与 $C_{j'}$，使 $C_{i'}$ 与 $C_{j'}$ 无关。这个过程称为正交化过程。

定义 13 关系 R 称为相关关系，如果 R 由且仅由以下规则生成：

(1) 如果 C_i 和 C_j 是软件体系结构 SA 的任意两个构件，且 C_i 和 C_j 是相关的，则 $<C_i,C_j>\in R$；

(2) 对 SA 的任意构件 C_i，都有 $<C_i,C_i>\in R$；

(3) 对 SA 的任意两个构件 C_i，C_j，如果 $<C_i,C_j>\in R$，则 $<C_j,C_i>\in R$；

(4) 对 SA 的任意三个构件 C_i，C_j 和 $C_k(i\neq j\neq k)$，如果 $<C_i,C_j>\in R$ 且 $<C_j,C_k>\in R$，则 $<C_i,C_k>\in R$。

定理 2 相关关系 R 是一个等价关系。

根据定义 13，定理 2 的结论是显然的。

由于 R 是一个等价关系，对于软件体系结构 SA 的构件集合 $C=\{C_1,C_2,\cdots,C_n\}$，可以构造 C 关于 R 的商集 $C/R=\{Cr_1,Cr_2,\cdots,Cr_n\}$，$C/R$ 构成 C 的一个划分，$Cr_i\subseteq C(i=1,2,\cdots,n)$。商集 C/R 是构件集合 C 上关于 R 的不同等价类的集合。因此，不同划分块中的构件之间无相关关系，同一划分块中的构件之间存在相关关系。对每一划分块 Cr_i，定义一个线索 $Th_i=C_{i1}C_{i2},\cdots,C_{ik}$（对不同的 Th_i，k 值各不相同），且 $C_{ij}\in Cr_i(j=1,2,\cdots,k)$，$i1<i2<\cdots<ik$，即 Th_i 中的构件序列不改变原构件连接的顺序。这样获得的线索 Th_1，$Th_2,\cdots,Th_n$ 之间无相关关系，它们是正交的。于是，体系结构 SA 就可正交化为线索集 $\{Th_1,Th_2,\cdots,Th_n\}$。

算法(软件体系结构正交化)

输入：非正交软件体系结构 SA

输出：正交软件体系结构 SA′

BEGIN

构造 SA 中构件的相关关系 R；

构造 SA 中构件集合 C 关于 R 的商集 C/R，得划分块的集合 $\{Cr_1,Cr_2,\cdots,Cr_n\}$；

将每一个划分块定义为一个线索，线索中不改变原构件连接的顺序，得正交

线索集合 $TS=(Th_1,Th_2,\cdots,Th_m)$；

END

软件体系结构和线索的正交化过程是需要工作量的，且正交化之后的整个线索空间的冗余度会增加。所以，在一个实际系统的设计和开发过程中，要根据正交化所需的工作量和系统冗余度与性能改进之间的关系，来确定是否进一步进行正交化。

3.6.3 正交软件体系结构的实例

以某省电力局的一个管理信息系统为例，讨论正交软件体系结构的应用。

1. 设计思想

设计初期，考虑到未来可能进行的机构改革，在系统投入运行后，单位各部门的功能有可能发生以下变化。

(1) 某些部门的功能可能改变，或取消，或转入另外的部门或其工作内容发生变更。

(2) 有的部门可能被撤销，其功能被整个并入其他部门或分解并入数个部门。

(3) 某些部门的功能可能需要扩充。

为适应将来用户需求可能发生的变化，尽量降低维护成本、提高可用性和重用性，设计

师使用了多级正交软件体系结构的设计思想。在本系统中,考虑到其系统较大和实际应用的需要,将线索分为两级:主线索和子线索。总体结构包含数个主线索(第一级),每个主线索又包含数个子线索(第二级),因此一个主线索也可以看成一个小的正交结构。这样为大型软件结构功能的划分提供了便利,使得既能对功能进行分类,又能在每一类中对功能进行细分。使功能划分既有序,又合理,能控制在一定粒度以内,合理的粒度又为线索和层次中构件的实现打下良好的基础。

在3.6.1节中提到的完全正交结构不能很好地适用于本应用,因此放宽了对结构正交的严格性,允许在线索间有适当的相互调用,因为各功能或多或少会有相互重叠的地方,因此可能发生共享某些构件的情况。进一步,还放宽了对结构分层的限制,允许某些线索(少数)的层次与其他线索(多数)不同。这些均是反复权衡理想情况时的优点和实现代价后总结出的原则,这样既易于实现,又能充分利用正交结构的优点。

2. 结构设计

按照上述思想,首先将整个系统设计为两级正交结构,第一级划分为38个主线索(子系统),系统总体结构如图3-19所示,每个主线索又可划分为数个子线索(≥2)。

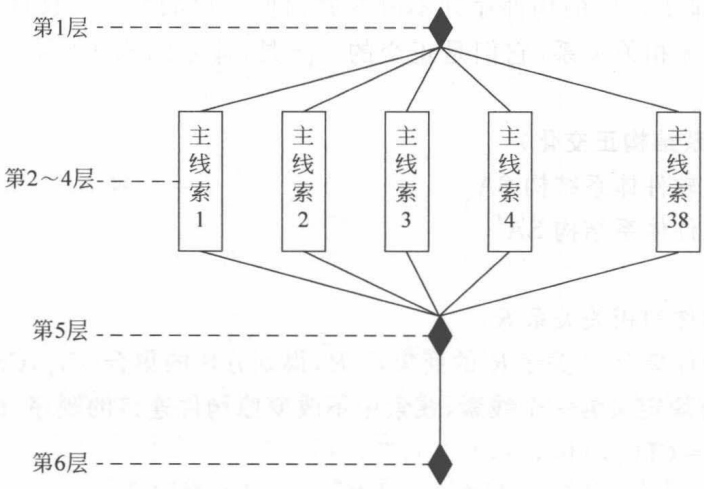


图3-19 系统总体正交体系结构设计

为了简单起见,下面就其中的一个主线索进行说明,其他主线索的子线索划分也采用大致相同的策略。该主线索所实现的功能属于多种经营管理处的范围,该处有生产经营科、安全监察科、财务科、劳务科,包括11个管理功能(如人员管理、产品质量监督、安全监察、生产经营、劳资统计等),即11个子线索,该主线索子正交体系结构如图3-20所示。

在图3-20中,主控窗口层、数据模型与数据库接口、物理数据库层分别对应图3-19中的第1层、第5层和第6层。组合图3-19和图3-20可看出,整个MIS的结构包括6个层次。

(1) 第1层实现主控窗口,由主控窗口对象控制引发所有线索运行。

(2) 第2层实现菜单接口,支持用户选择不同的处理功能。

(3) 第3层涵盖了所有的功能对话框,这也就是与功能的真正接口。

(4) 第4层是真正的功能定义,在这定义的构件有:数据录入构件(包括插入、删除、更新)、报表处理构件、快速查询构件、图形分析构件、报表打印构件等。

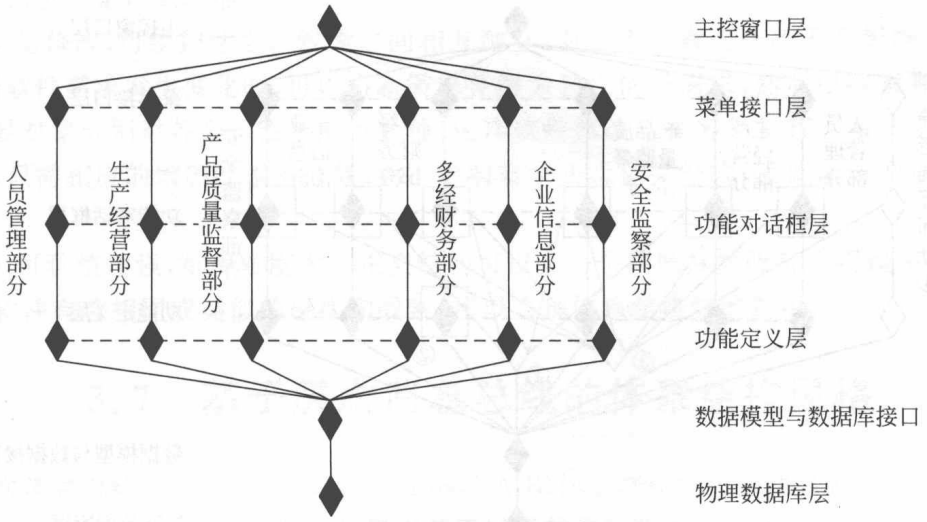


图 3-20　多种经营主线索子正交结构图

(5) 第 5 层和第 6 层是数据服务的实现，第 5 层包括特定的数据模型和数据库接口，第 6 层就是数据库本身。

3. 程序编制

在软件结构设计方案确定之后，就可以开始正式的开发工作，由于采用正交结构的思想，可以分数个小组并行开发。每个小组分配一条或数条线索，由专门一个小组设计通用共享构件。由于构件是通用的，因此不必与其他小组频繁联系，加上各条线索之间相互调用少，所以各小组不会互相牵制，再加上构件的重用，从而大大提高了编程效率，给设计带来极大的灵活性，缩短了开发周期，降低了工作量。

4. 演化控制

软件开发完成并运行一年后，用户单位提出了新的要求，要对原设计方案进行修改，按照在第 1 部分提出的设计思想和方法，首先将提出的新功能要求映射到原设计结构上，这里仍以“多种经营管理主线索”为例，总结出以下变动。

(1) 报表和报表处理功能的变动。例如，财务管理子线索有如下变动，财务报表有增删(直接在数据库中添加和删除表)，某些报表需要增加一些汇兑处理和计算功能(对它的功能定义层做上修改标记)，其他一些子功能(子线索)也需增加自动上报功能，另外，所有的子线索都需要增加浏览器功能，以便对数据进行网上浏览。

(2) 子线索的变动：增加养老统筹子线索。

对初始结构的变动如图 3-21 所示。其中，①②所指不变，③表示在功能定义层新添加的一个构件，其中包含网上浏览功能和自动上报(E-mail 发送)功能，④表示新增加养老统筹子线索。新添加的构件用空心菱形表示，要修改的构件在其上套一个矩形做标记，其他未做标记的则表示可直接重用的构件。确定演化的结构图之后，按照自顶向下，由左至右的原则，更新演化工作得以有条不紊地进行。

现对多经主线索子正交结构演化情况进行统计：原结构包含子线索 11 条，构件 36 个；新结构包含子线索 12 条，构件 41 个。其中，重用构件 24 个，修改 11 个，新增 6 个。新增子

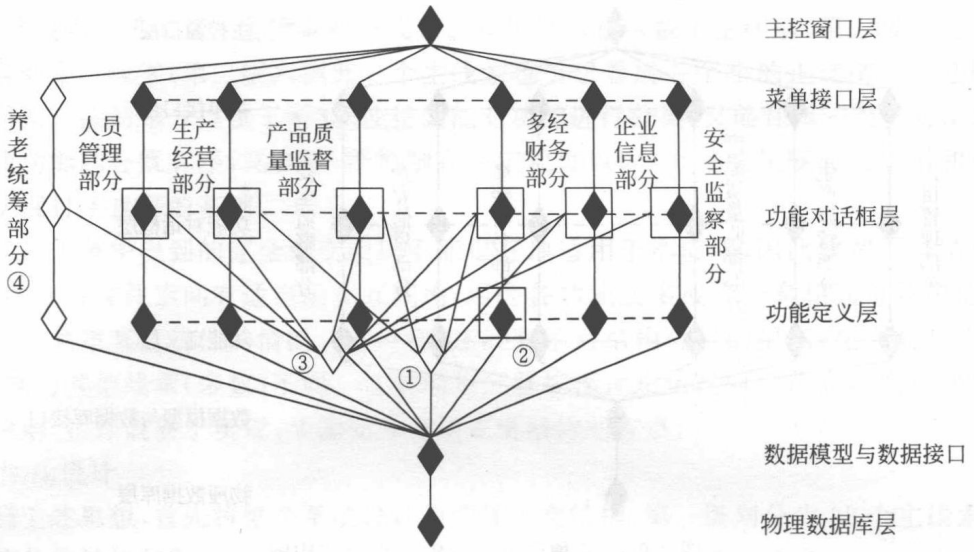


图 3-21　多经主线索结构变动情况图

线索一条。由于涉及添加公用共享构件,所以没有完全重用某条子线索的情况,但是大部分只用在功能对话框层做少量修改即可。

经分析,构件重用率为58.6%,修改率为26.8%,增加率为14.6%。表3-1是对工作量(每天8h,单位人/天)的统计分析。

38个主线索的修改与开发工作量比例均未超过30%,比传统方法工作量减少20%左右。可见多级正交结构对于降低软件演化更新的开销是行之有效的,而且非常适合大型软件开发,特别是在MIS领域,由于其结构在一定应用领域内均有许多共同点,因此有一定的通用性。

表 3-1　劳动量比较表

主　线　索	修改前开发工作量	修改工作量	修改与开发工作量之比
1	60	4	6.67%
2	120	4	3.33%
3	90	6	6.67%
4	60	4	6.67%
5	60	2	3.33%
6	60	3	5.00%
⋮	⋮	⋮	⋮
37	60	2	3.33%
38	30	3	10%

3.6.4　正交软件体系结构的优点

正交软件体系结构具有以下优点。

(1) 结构清晰,易于理解。正交软件体系结构的形式有利于理解。由于线索功能相互独立,不进行互相调用,结构简单、清晰,构件在结构图中的位置已经说明它所实现的是哪一

级抽象，担负的是什么功能。

(2) 易修改，可维护性强。线索之间相互独立，对一个线索的修改不会影响其他线索。因此，当软件需求发生变化时，可以将新需求分解为独立的子需求，然后以线索和其中的构件为主要对象分别对各个子需求进行处理，这样软件修改就很容易实现。系统功能的增加或减少，只需相应的增删线索构件族，而不影响整个正交体系结构，因此能方便地实现结构调整。

(3) 可移植性强，重用粒度大。正交结构可以为一个领域内的所有应用程序所共享，因此这些软件有着相同或类似的层次和线索，可以实现体系结构级的重用。

3.7 基于层次消息总线的体系结构风格

层次消息总线(Hierarchy Message Bus，HMB)体系结构风格是北京大学杨芙清院士等提出的一种风格，该体系结构风格的提出基于以下的实际背景。

(1) 随着计算机网络技术的发展，特别是分布式构件技术的日渐成熟和构件互操作标准的出现，如 CORBA、DCOM、EJB 等，加速了基于分布式构件的软件开发趋势。具有分布和并发特点的软件系统已成为一种广泛的应用需求。

(2) 基于事件驱动的编程模式已在图形用户界面程序设计中获得广泛应用。在此之前的程序设计中，通常使用一个大的分支语句控制程序的转移，对不同的输入情况分别进行处理，程序结构不甚清晰。基于事件驱动的编程模式在对多个不同事件响应的情况下，系统自动调用相应的处理函数，程序具有清晰的结构。

(3) 计算机硬件体系结构和总线的概念为软件体系结构的研究提供了很好的借鉴和启发，在统一的体系结构框架下(即总线和接口规范)，系统具有良好的扩展性和适应性。任何计算机厂商生产的配件，甚至是在设计体系结构时根本没有预料到的配件，只要遵循标准的接口规范，都可以方便地集成到系统中，对系统功能进行扩充，甚至是即插即用(即运行时刻的系统演化)。正是标准的总线和接口规范的指定，以及标准化配件的生产，促进了计算机硬件的产业分工和蓬勃发展。

HMB 风格基于层次消息总线、支持构件的分布和并发，构件之间通过消息总线进行通信，如图 3-22 所示。

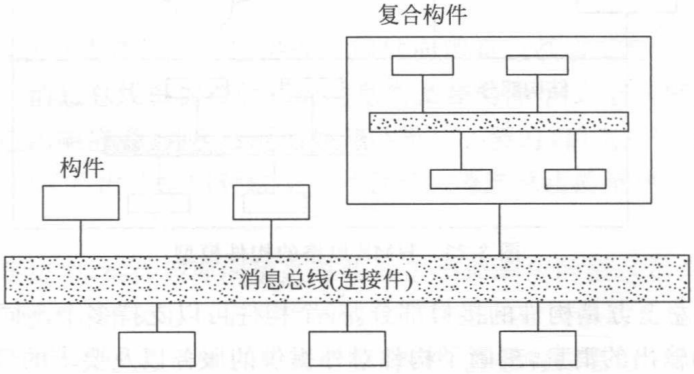


图 3-22 HMB 风格的系统示意图

消息总线是系统的连接件,负责消息的分派、传递和过滤以及处理结果的返回。各个构件挂接在消息总线上,向总线登记感兴趣的消息类型。构件根据需要发出消息,由消息总线负责把该消息分派到系统中所有对此消息感兴趣的构件,消息是构件之间通信的唯一方式。构件接收到消息后,根据自身状态对消息进行响应,并通过总线返回处理结果。由于构件通过总线进行连接,并不要求各个构件具有相同的地址空间或局限在一台机器上。该风格可以较好地刻画分布式并发系统,以及基于 CORBA、DCOM 和 EJB 规范的系统。

如图 3-22 所示,系统中的复杂构件可以分解为比较低层的子构件,这些子构件通过局部消息总线进行连接,这种复杂的构件称为复合构件。如果子构件仍然比较复杂,可以进一步分解,如此分解下去,整个系统形成了树状的拓扑结构。树结构的末端结点称为叶结点,它们是系统中的原子构件,不再包含子构件。原子构件的内部可以采用不同于 HMB 的风格,如前面提到的数据流风格、面向对象风格及管道-过滤器风格等,这些属于构件的内部实现细节。但要集成到 HMB 风格的系统中,必须满足 HMB 风格的构件模型的要求,主要是在接口规约方面的要求。另外,整个系统也可以作为一个构件,通过更高层的消息总线,集成到更大的系统中。于是,可以采用统一的方式刻画整个系统和组成系统的单个构件。

下面讨论 HMB 风格中的各个组成要素。

3.7.1 构件模型

系统和组成系统的成分通常比较复杂,难以从一个视角获得对它们的完整理解,因此一个好的软件工程方法往往从多个视角对系统进行建模,一般包括系统的静态结构、动态行为和功能等方面。例如,OMT 方法采用了对象模型、动态模型和功能模型刻画系统的以上 3 个方面。

借鉴以上思想,为满足体系结构设计的需要,HMB 风格的构件模型包括接口、静态结构和动态行为 3 个部分,如图 3-23 所示。

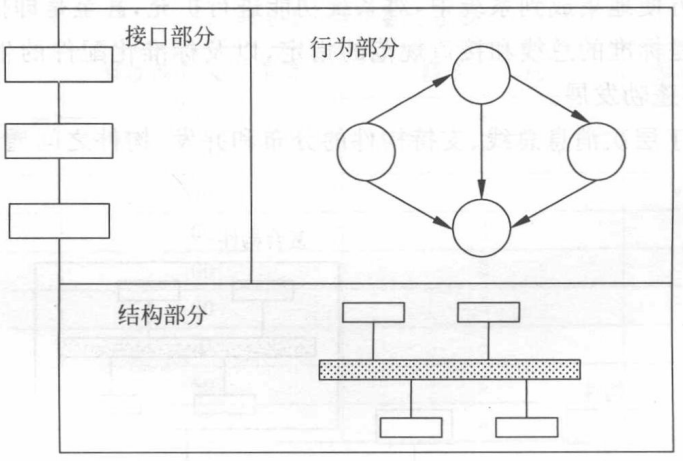


图 3-23 HMB 风格的构件模型

在图 3-23 中,左上方是构件的接口部分,一个构件可以支持多个不同的接口,每个接口定义了一组输入和输出的消息,刻画了构件对外提供的服务以及要求的环境服务,体现了该构件同环境的交互。右上方是用带输出的有限状态自动机刻画的构件行为,构件接收到外来消息后,根据当前所处的状态对消息进行响应,并可能导致状态的变迁。下方是复合构件

的内部结构定义。复合构件由更简单的子构件通过局部消息总线连接而成。消息总线为整个系统和各个层次的构件提供了统一的集成机制。

3.7.2 构件接口

在体系结构设计层次上，构件通过接口定义了同外界的信息传递和承担的系统责任，构件接口代表了构件同环境的全部交互内容，也是唯一的交互途径。除此之外，环境不应对构件做任何其他与接口无关的假设，如实现细节等。

HMB 风格的构件接口是一种基于消息的互联接口，可以较好地支持体系结构设计。构件之间通过消息进行通信，接口定义了构件发出和接收的消息集合。同一般的互联接口相比，HMB 的构件接口具有两个显著的特点。首先，构件只对消息本身感兴趣，并不关心消息是如何产生的，消息的发出者和接收者不必知道彼此的情况，这样就切断了构件之间的直接联系，降低了构件之间的耦合程度，进一步增强了构件的重用潜力，并使得构件的替换变得更为容易。另外，在一般的互联接口定义的系统中，构件之间的连接是在要求的服务和提供的服务之间进行固定的匹配，而在 HMB 的构件接口定义的系统中，构件对外来消息进行响应后，可能会引起状态的变迁。因此，一个构件在接收到同样的消息后，在不同时刻所处的不同状态下，可能会有不同的响应。

消息是关于某个事件发生的信息，上述接口定义中的消息分为以下两类。

(1) 构件发出的消息：通知系统中其他构件某个事件的发生或请求其他构件的服务。

(2) 构件接收的消息：对系统中某个事件的响应或提供其他构件所需的服务。接口中的每个消息定义了构件的一个端口，具有互补端口的构件可以通过消息总线进行通信。互补端口指的是除了消息进出构件的方向不同之外，消息名称、消息带有的参数和返回结果的类型完全相同的两个端口。

当某个事件发生后，系统或构件发出相应的消息，消息总线负责把该消息传递到此消息感兴趣的构件。按照响应方式的不同，消息可分为同步消息和异步消息。同步消息是指消息的发送者必须等待消息处理结果返回才可以继续运行的消息类型。异步消息是指消息的发送者不必等待消息处理结果的返回即可继续执行的消息类型。常见的同步消息包括过程调用，异步消息包括信号、时钟、异步过程调用等。

3.7.3 消息总线

HMB 风格的消息总线是系统的连接件。构件向消息总线登记感兴趣的消息，形成构件消息响应登记表。消息总线根据接收到的消息类型和构件-消息响应登记表的信息，定位并传递该消息给相应的响应者，并负责返回处理结果。必要时，消息总线还对特定的消息进行过滤和阻塞。图 3-24 给出了采用对象类符号表示的消息总线的结构。

1. 消息登记

在基于消息的系统中，构件需要向消息总线登记当前响应的消息集合。消息响应者只对消息类型感兴趣，通常并不关心是谁发出的消息。在 HMB 风格的系统中，对挂接在同一消息总线上的构件而言，消息是一种共享的资源，构件-消息响应登记表记录了该总线上所有构件和消息的响应关系。类似于程序设计中的“间接地址调用”，避免了将构件之间的连接“硬编码”到构件的实例中，使得构件之间保持了灵活的连接关系，便于系统的演化。

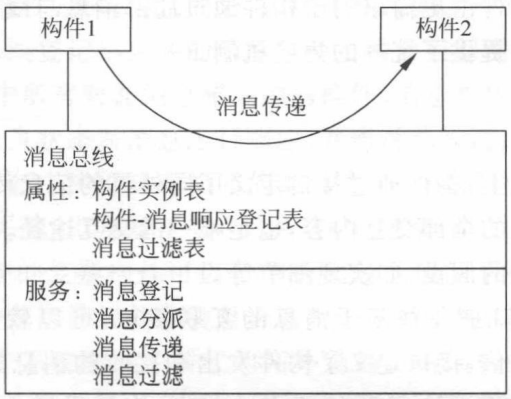


图 3-24　消息总线的结构

构件接口中的接收消息集合意味着构件具有响应这些消息类型的潜力，默认情况下，构件对其接口中定义的所有接收消息都可以进行响应。但在某些特殊的情况下，例如，当一个构件在部分功能上存在缺陷时，就难以对其接口中定义的某些消息进行正确的响应，这时应阻塞那些不希望接收到的消息。这就是需要显式进行消息登记的原因，以便消息响应者更灵活地发挥自身的潜力。

2. 消息分派和传递

消息总线负责消息在构件之间的传递，根据构件-消息响应登记表把消息分派到对此消息感兴趣的构件，并负责处理结果的返回。在消息广播的情况下，可以有多个构件同时响应一个消息，也可以没有构件对该消息进行响应。在后一种情况下，该消息就丢失了，消息总线可以对系统的这种异常情况发出警告，或通知消息的发送构件进行相应的处理。

实际上，构件-消息响应登记表定义了消息的发送构件和接收构件之间的一个二元关系，以此作为消息分派的依据。

消息总线是一个逻辑上的整体，在物理上可以跨越多个机器，因此挂接在总线上的构件也就可以分布在不同的机器上，并发运行。由于系统中的构件不是直接交互，而是通过消息总线进行通信，因此实现了构件位置的透明性。根据当前各个机器的负载情况和效率方面的考虑，构件可以在不同的物理位置上透明地迁移，而不影响系统中的其他构件。

3. 消息过滤

消息总线对消息过滤提供了转换和阻塞两种方式。消息过滤的原因主要在于不同来源的构件事先并不知道各自的接口，因此可能同一消息在不同的构件中使用了不同的名字，或不同的消息使用了相同的名字。前面提到，对挂接在同一消息总线上的构件而言，消息是一种共享的资源，这样就会造成构件集成时消息的冲突和不匹配。

消息转换是针对构件实例而言的，即所有构件实例发出和接收的消息类型都经过消息总线的过滤，这里采取简单换名的方法，其目标是保证每种类型的消息名字在其所处的局部总线范围内是唯一的。例如，假设复合构件 A 复合客户/服务器风格，由构件 C 的两个实例 c1 和 c2 以及构件 S 的一个实例 s1 构成，构件 C 发出的消息 msgC 和构件 S 接收的消息 msgS 是相同的消息。但由于某种原因，它们的命名并不一致(除此之外，消息的参数和返回值完全一样)。可以采取简单换名的方法，把构件 C 发出的消息 msgC 换名为 msgS，这样无须对构件进行修改，就解决了这两类构件的集成问题。

由简单的换名机制解决不了的构件集成的不匹配问题，如参数类型和个数不一致等，可以采取更为复杂的包装器(wrapper)技术对构件进行封装。

3.7.4 构件静态结构

HMB风格支持系统自顶向下的层次化分解，复合构件是由比较简单的子构件组装而成的，子构件通过复合构件内部的消息总线连接，各个层次的消息总线在逻辑功能上是一致的，负责相应构件或系统范围内消息的登记、分派、传递和过滤。如果子构件仍然比较复杂，可以进一步分解。图3-25是某个系统经过逐层分解所呈现出的静态结构示意图，不同的消息总线分别属于系统和各层次的复合构件，消息总线之间没有直接的连接，HMB风格中的这种总线称为层次消息总线。另外，整个系统也可以作为一个构件，集成到更大的系统中。因为各个层次的构件以及整个系统采取了统一的方式进行刻画，所以定义一个系统的同时也就定义了一组“系统”，每个构件都可看作一个独立的子系统。

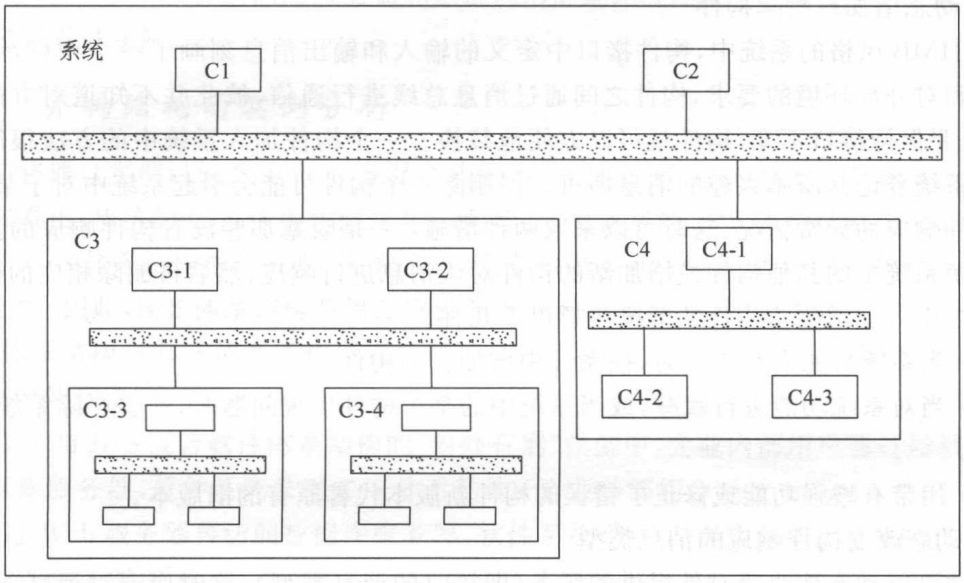


图3-25 系统/复合构件的静态结构示意图

3.7.5 构件动态行为

在一般的基于事件风格的系统中，如图形用户界面系统X-Window，对于同一类事件，构件(这里指的是回调函数)总是采取同样的动作进行响应。这样，构件的行为就由外来消息的类型唯一确定，即一个消息和构件的某个操作之间存在着固定的对应关系。对于这类构件，可以认为构件只有一个状态，或者在每次对消息响应之前，构件处于初始状态。虽然在操作的执行过程中，会发生状态的变迁，但在操作结束之前，构件又恢复到初始状态。无论以上哪种情况，都不需要构件在对两个消息响应之间，保持其状态信息。

更通常的情况是，构件的行为同时受外来消息类型和自身当前所处状态的影响。类似一些面向对象方法中用状态机刻画对象的行为，在HMB风格的系统中，采用带输出的有限状态机描述构件的行为。

带输出的有限状态机分为Moore机和Mealy机两种类型，它们具有相同的表达能力。

在一般的面向对象方法中,通常混合采用 Moore 机和 Mealy 机表达对象的行为。为了实现简单,选择采用 Mealy 机描述构件的行为。一个 Mealy 机包括一组有穷的状态集合、状态之间的变迁和在变迁发生时的动作。其中,状态表达了在构件的生命周期内,构件所满足的特定条件、实施的活动或等待某个事件的发生。

3.7.6 运行时刻的系统演化

在许多重要的应用领域中,如金融、电力、电信及空中交通管制等,系统的持续可用性是一个关键性的要求,运行时刻的系统演化可减少因关机和重新启动带来的损失和风险。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。

HMB 风格方便地支持运行时刻的系统演化,主要体现在动态增加或删除构件、动态改变构件响应的消息类型和消息过滤 3 个方面。

1. 动态增加或删除构件

在 HMB 风格的系统中,构件接口中定义的输入和输出消息刻画了一个构件承担的系统责任和对外部环境的要求,构件之间通过消息总线进行通信,彼此并不知道对方的存在。因此,只要保持接口不变,构件就可以方便地替换。一个构件加入系统中的方法很简单,只需要向系统登记其所感兴趣的消息即可。但删除一个构件可能会引起系统中对于某些消息没有构件响应的异常情况,这时可以采取两种措施:一是阻塞那些没有构件响应的消息,二是首先使系统中的其他构件或增加新的构件对该消息进行响应,然后再删除相应的构件。

系统中可能增删改构件的情况包括以下几种。

(1) 当系统功能需要扩充时,往系统中增加新的构件。

(2) 当对系统功能进行裁减,或当系统中的某个构件出现问题时,需要删除系统中的某个构件。

(3) 用带有增强功能或修正了错误的构件新版本代替原有的旧版本。

2. 动态改变构件响应的消息类型

构件可以动态地改变对外提供的服务(即接收的消息类型),这时应通过消息总线对发生的改变进行重新登记。

3. 消息过滤

利用消息过滤机制,可以解决某些构件集成的不匹配问题。消息过滤通过阻塞构件对某些消息的响应,提供了另一种动态改变构件对消息进行响应的方式。

3.8 异构结构风格

前面几节的内容介绍和讨论了一些所谓的"纯"体系结构,但随着软件系统规模的扩大,系统也越来越复杂,所有的系统不可能都在单一的标准的结构上进行设计,其原因分析如下。

(1) 从最根本上来说,不同的结构有不同的处理能力,都有各自的强项和弱点,一个系统的体系结构应该根据实际需要进行选择,以解决实际问题。

(2) 关于软件包、框架、通信以及其他一些体系结构上的问题,目前存在多种标准。即使在某段时间内某一种标准占统治地位,但变动最终是绝对的。

(3) 实际工作中，总会遇到一些遗留下来的代码，它们仍有效用，但是却与新系统有某种程度上的不协调。然而在许多场合，将技术与经济综合进行考虑时，总是决定不重写它们。

(4) 即使在某一单位中，规定了共享共同的软件包或相互关系的一些标准，仍会存在解释或表示习惯上的不同。在 UNIX 中就可以发现这类问题：即使规定用单一的标准(ASCII)来保证过滤器之间的通信，但因为不同人对关于在 ASCII 流中信息如何表示的不同的假设，不同的过滤器之间仍可能不协调。

大多数应用程序只使用 10%的代码实现系统的公开的功能，剩下 90%的代码完成系统管理功能：输入和输出、用户界面、文本编辑、基本图表、标准对话框、通信、数据确认和旁听跟踪、特定领域(如数学或统计库)的基本定义等。

如果能从标准的构件构造系统 90%的代码是很理想的，但不幸的是，即使能找到一组符合要求的构件，很有可能发现它们不会很容易组织在一起。通常问题出在：各构件做了数据表示、系统组织、通信协议细节或某些明确的决定(如谁将拥有主控制线程)等不同的假设。例如，UNIX 中的排序操作，过滤器和系统调用都是标准配置中的部分，虽然都是排序，但它们之间不可互换。

3.8.1 异构结构的实例分析

本节将通过实例讨论 C/S 与 B/S 混合软件体系结构。从 3.2 节、3.3 节和 3.4 节的讨论中可以看出，传统的 C/S 体系结构并非一无是处，而新兴的 B/S 体系结构也并非十全十美。由于 C/S 体系结构根深蒂固，技术成熟，原来的很多软件系统都是建立在 C/S 体系结构基础上的，因此，B/S 体系结构要想在软件开发中起主导作用，要走的路还很长。笔者认为，C/S 体系结构与 B/S 体系结构还将长期共存，其结合方式主要有以下两种模型。

1. “内外有别”模型

在 C/S 与 B/S 混合软件体系结构的“内外有别”模型中，企业内部用户通过局域网直接访问数据库服务器，软件系统采用 C/S 体系结构；企业外部用户通过 Internet 访问 Web 服务器，通过 Web 服务器再访问数据库服务器，软件系统采用 B/S 体系结构。“内外有别”模型的结构如图 3-26 所示。

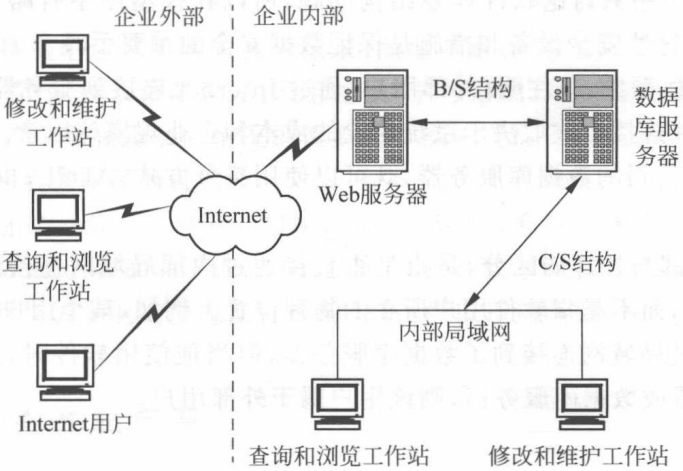


图 3-26 “内外有别”模型

"内外有别"模型的优点是：外部用户不直接访问数据库服务器，能保证企业数据库的相对安全；企业内部用户的交互性较强，数据查询和修改的响应速度较快。

"内外有别"模型的缺点是：企业外部用户修改和维护数据时，速度较慢，较烦琐，数据的动态交互性不强。

2. "查改有别"模型

在C/S与B/S混合软件体系结构的"查改有别"模型中，不管用户是通过什么方式(局域网或Internet)连接到系统，凡是须执行维护和修改数据操作的，均使用C/S体系结构；如果只是执行一般的查询和浏览操作，则使用B/S体系结构。"查改有别"模型的结构如图3-27所示。

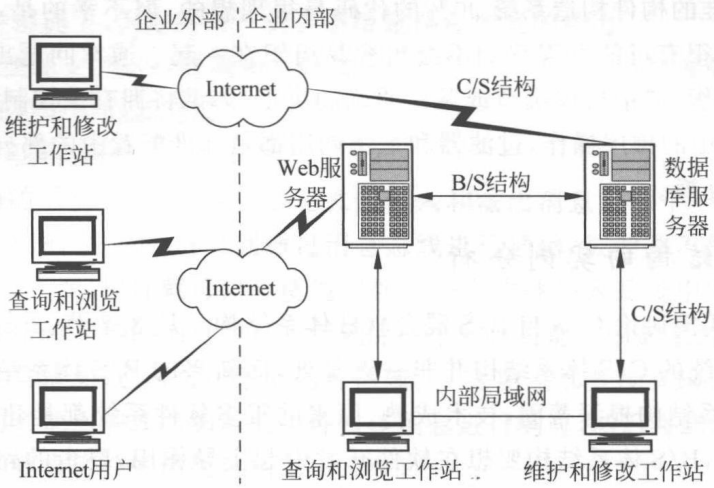


图3-27 "查改有别"模型

"查改有别"模型体现了B/S体系结构和C/S体系结构的共同优点。但由于外部用户能直接通过Internet连接到数据库服务器，企业数据容易暴露给外部用户，给数据安全造成了一定的威胁。

3. 几点说明

(1) 因为在本节中只讨论软件体系结构问题，所以在模型图中省略了有关网络安全的设备，如防火墙等，这些安全设备和措施是保证数据安全的重要手段。

(2) 在这两个模型中，只注明(外部用户)通过Internet连接到服务器，但并没有解释具体的连接方式，这种连接方式取决于系统建设的成本和企业规模等因素。例如，某集团公司的子公司要访问总公司的数据库服务器，既可以使用拨号方式、ADSL，也可以使用DDN方式等。

(3) 本节对内部与外部的区分，是指是否直接通过内部局域网连接到数据库服务器进行软件规定的操作，而不是指软件用户所在的物理位置。例如，某个用户在企业内部办公室里，其计算机也通过局域网连接到了数据库服务器，但当他使用软件时，是通过拨号的方式连接到Web服务器或数据库服务器，则该用户属于外部用户。

4. 应用实例

当前，我国电力系统正在进行精简机构的改革，变电站也在朝着无人、少人和一点带面的方向发展(如，一个有人值班220kV变电站带若干无人值班220kV和110kV变电站)，

“减人增效”是必然的趋势，而要很好地达到这个目的，使用一套完善的变电综合信息管理系统（简称 TSMIS）显得很有必要。为此，作者曾组织有关力量，针对电力系统变电运行管理工作的需要，结合变电站运行工作经验，开发了一套完整的变电综合信息管理系统。

1）体系结构设计

在设计 TSMIS 系统时，充分考虑到变电站分布管理的需要，采用 C/S 与 B/S 混合软件体系结构的“内外有别”模型，如图 3-28 所示。

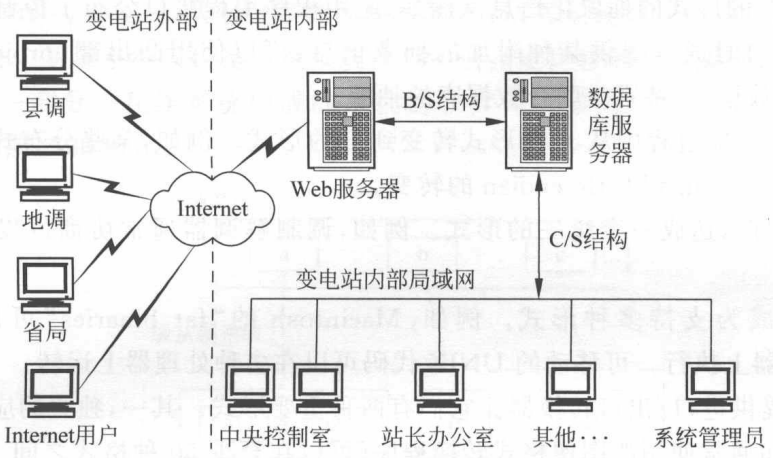


图 3-28　TSMIS 系统软件体系结构

在 TSMIS 系统中，变电站内部用户通过局域网直接访问数据库服务器，外部用户（包括县调、地调和省局的用户及普通 Internet 用户）通过 Internet 访问 Web 服务器，再通过 Web 服务器访问数据库服务器。外部用户只需一台接入 Internet 的计算机，就可以通过 Internet 查询运行生产管理情况，无须做太大的投入和复杂的设置。这样也方便所属电业局及时了解各变电站所的运行生产情况，对各变电站的运行生产进行宏观调控。此设计能很好地满足用户的需求，符合可持续发展的原则，使系统有较好的开放性和易扩展性。

2）系统实现

TSMIS 系统包括变电运行所需的运行记录、图形开票、安全生产管理、生产技术管理、行政管理、总体信息管理、技术台账管理、班组建设、学习培训、系统维护等各个业务层次模块。实际使用时，用户可以根据实际情况的需要选择模块进行自由组合，以达到充分利用变电站资源和充分发挥系统作用的目的。

系统的实现采用 Visual C++、Visual Basic、Java 等语言和开发平台进行混合编程。服务器操作系统使用 Windows 2003 Advanced Server，后台数据库采用 SQL Server 2005。系统的实现充分考虑到我国变电站所电压等级的分布，可以适用于大、中、小电压等级的变电站所。

3.8.2　异构组合匹配问题

软件工程师有许多技术来处理结构上的不匹配。最简单的描述这些技术的例子是只有两个构件的情况。这两个构件可以是对等构件、一对相互独立的应用程序、一个库和一个调

用者、客户机和服务器等,基本形式如图3-29所示。

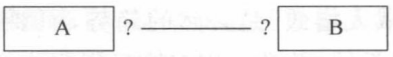


图3-29 构件协调问题

A和B不能协调工作的原因可能是它们事先做了对数据表示、通信、包装、同步、语法、控制等方面的假设,这些方面统称为形式(form)。下面给出若干解决A与B之间不匹配问题的方法。这里假设A和B是对称的,它们可以互换。

(1) 形式A改变成B的形式。为了与另一构件协调,彻底重写其中之一的代码是可能的,但又是很昂贵的。

(2) 公布A的形式的抽象化信息。例如,应用程序编程接口公布了控制一个构件的过程调用,开放接口时通常提供某种附加的抽象信息,可以使用凸出部(projections)或视图(views)来提供数据库,特别是联合数据库的抽象。

(3) 在数据传输过程中从A的形式转变到B的形式。例如,某些分布式系统在数据传输中进行从big-endian到little-endian的转变。

(4) 通过协商,达成一个统一的形式。例如,调制解调器通常协商以发现最快的通信协议。

(5) 使B成为支持多种形式。例如,Macintosh的"fat binaries"可以在680X0或PowerPC处理器上执行。可移动的UNIX代码可以在多种处理器上运转。

(6) 为B提供进口/出口转换器。它们有两种重要形式。其一,独立的应用程序提供表示转换服务。如通常使用的图像格式转换程序(可以在至少50种格式之间、10种平台上进行转换)。其二,某些系统协调扩充或外部add-ons,它们可以完成内外部数据格式之间的相互转换。

(7) 引入中间形式。

第一,外部相互交换表示,有时通过界面描述语言(Interface Description Languages, IDLs)支持,能够提供一个中介层。这在存在多个形式互不相同的构件结合在一起的时候特别有用。

第二,标准的发布形式,如RTF、MIF、Postscript;或Adobe Acrobat提供了另一种可供选择的方式,一种广泛流通的安全的表示法。

第三,活跃的调解者(active media)可以被插入系统,作为中介。

(8) 在A上添加一个适配器(adapter)或包装器。最终的包装器可能是一种处理器模拟另一种处理器的代码。软件包装器可以在形式上掩盖不同,如Mosaic和其他Web浏览器隐藏了所显示的文件的表示一样。

(9) 保持对A和B的版本并行一致。虽然较微妙,但保持A和B自己的形式,并完成两种形式的所有改变是有可能的。

以上这些技术有各自的优点和缺点,它们在初始化、时间和空间效率、灵活性和绝对的处理能力上差别很大。

3.9 互连系统构成的系统及其体系结构

开发大规模系统时,其复杂程度将大大增加。它不但要求开发人员能够理解一套更为复杂的流程,并且由于需要管理更多的资源,为此要负担额外的开销。互连系统构成的系统

(System of Interconnected Systems, SIS)是 Herbert H. Simon 在 1981 年提出的一个概念。1995 年,Jacobson 等人对这种系统进行了专门的讨论。

3.9.1 互连系统构成的系统

SIS 是指系统可以分成若干不同的部分,每个部分作为单独的系统独立开发。整个系统通过一组互连系统实现,而互连系统之间相互通信,履行系统的职责。其中一个系统体现整体性能,称为上级系统(superordinate system);其余系统代表整体的一个部分,称为从属系统(subordinate system)。从上级系统的角度来看,从属系统是子系统,上级系统独立于其从属系统,每个从属系统仅仅是上级系统模型中所指定内容的一个实现,并不属于上级系统功能约束的一部分。互连系统构成的系统的软件体系结构(Software Architecture for SIS,SASIS)如图 3-30 所示。

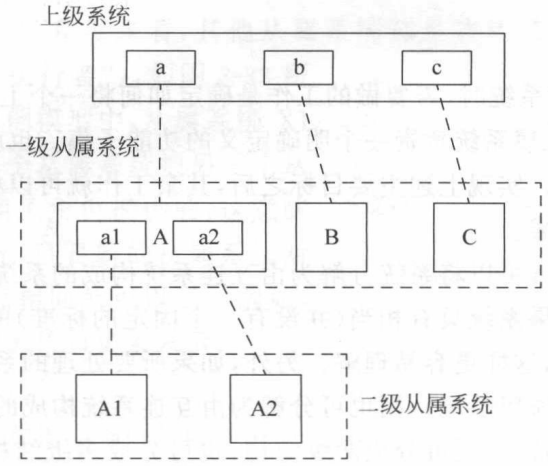


图 3-30 SIS 的体系结构

图 3-30 是一个三级结构的 SIS 体系结构示意图,上级系统的功能约束由一个互连系统构成的系统来实现,其中一级从属系统 A、B 和 C 分别是上级系统的子系统 a、b 和 c 的具体实现。二级从属系统中的 A1 和 A2 分别是一级从属系统 A 的子系统 a1 和 a2 的具体实现。

在 SASIS 中,上级系统与从属系统是相对而言的。例如,在图 3-30 中,一级从属系统既是上级系统的从属系统,同时又是二级从属系统的上级系统。

需要说明的是,并不是每一个 SIS 系统都只能分解为三级结构,需要根据系统的规模进行分解,可以只分解为两级结构。如果要开发的系统规模很大,可能需要进一步划分从属系统,形成多级结构的 SIS 系统。

从属系统可以自成一个软件系统,脱离上级系统而运行,有其自己的软件生命周期,在生命周期内的所有活动中都可以单独管理,可以使用不同的开发流程来开发各个从属系统。

常用的系统开发过程也可应用于 SIS 系统,可以将上级系统与从属系统的实现分离。通过把从属系统插入由互连系统构成的其他系统,就很容易使用从属系统来实现其上级系统。

在 SIS 系统中,要注意各从属系统之间的独立性,在上级系统的设计模型中,每个从

属系统实现一个子系统。子系统依赖于彼此的接口,但相互之间是相对独立的。因此,当某从属系统的需求发生变化时,不必开发新版本的上级系统,只需要对该从属系统内部构件进行变更,不会影响其他从属系统。只有在主要功能变更时才要求开发上级系统的新版本。

3.9.2 基于SASIS的软件过程

在SIS系统中,首先开始的是上级系统的软件过程(software process,有关软件过程的详细介绍见第12章)。一旦上级系统经历过至少一次迭代而且从属系统的接口相对稳定,从属系统的软件过程即可开始。上级系统和从属系统可以使用相同的软件过程进行开发。其软件过程如图3-31所示。

图3-31 基于SASIS的软件过程

1. 系统分解

开发基于SASIS的系统时,需要做的工作是确定如何将一个上级系统的功能分布在几个从属系统之间,每个从属系统负责一个明确定义的功能子集。也就是说,主要目标是定义这些从属系统间的接口。实现上述主要目标之后,其余工作就可以根据"分而治之"的原则,由每个从属系统独立完成。

那么,在什么情况下,可以将系统分解为由互连系统构成的系统呢?一般根据系统的规模和复杂性来决定。如果系统具有相当(并没有一个固定的标准)的规模和复杂性,则可以把问题分成许多小问题,这样更容易理解。另外,如果所要处理的系统可由若干物理上独立的系统组成(例如,处理遗留系统),则也可分解为由互连系统构成的系统。

对一个系统进行分解时,既可分成两级结构,也可分成多级结构。那么,分解到何种程度才能停止呢?要做到适度分解,体系结构设计师需要有丰富的实践经验。过度的分解会导致资源管理困难,项目难以协调。而且,不适度地使用物理上分离的系统或者物理上分离的团队,会在很大程度上扼杀任何形式的软件重用。

为了降低风险,做到适度分解,需要成立一个体系结构设计师小组,负责监督整个开发工作。体系结构设计师小组应该重点关注以下问题。

(1) 适当关注从属系统之间的重用和经验共享。

(2) 对开发什么工件(构件或文档)、从属系统工件与上级系统工件之间有什么关系有清晰的理解。

(3) 定义一个有效的变更管理策略,并得到所有团队的遵守。

2. 用例建模

开发人员应该为每个系统(包括上级系统和从属系统)在SIS系统中建立一个用例(use case,关于用例的细节问题将在第5章介绍)模型,上级系统的高级用例通常可以分解到(所有或部分)子系统上,每个"分块"将成为从属系统模型中的一个用例,如图3-32所示。

在图3-32中,上级系统X有3个用例,分别为A、B和C。其中,用例A分解到从属系统X1的分块为A1,分解到从属系统X2的分块为A2,分解到从属系统X3的分块为A3;用例B只分解到从属系统X1和X2(分别为B1和B2);用例C只分解到从属系统X2和X3(分别为C2和C3)。

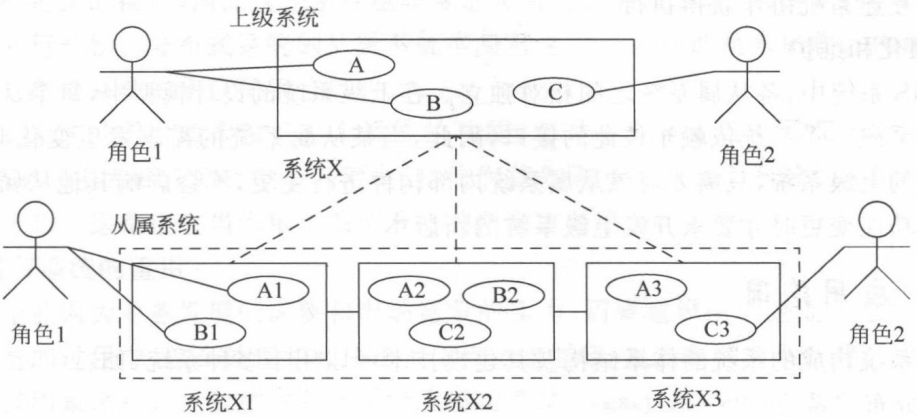


图 3-32　上级系统的高级用例与从属系统的详细用例之间的关系

从任何一个从属系统的角度来看，其他从属系统都是该从属系统用例模型的角色(actor，有的文献翻译为“执行者”)，如图 3-33 所示。在从属系统 X2 的用例模型中，从属系统 X1 和 X3 都被视为角色。一个从属系统把另一个从属系统的接口看作是由相关角色提供的，这意味着可以更换某个从属系统，只要新的从属系统对其他从属系统扮演的角色相同就可以了。例如，新的从属系统可以用相同的接口集来表示。

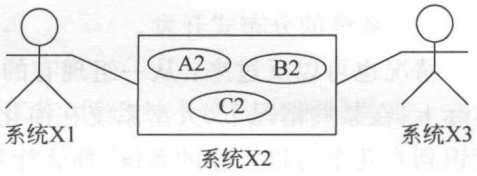


图 3-33　一个从属系统是另一个从属系统的角色

3. 分析和设计

分析和设计的目的是获得一个健壮的系统体系结构，这对 SIS 系统来说极其重要。SIS 中的每个系统，包括上级系统和从属系统，都应该定义自己的体系结构，选择相应的体系结构风格。分析设计的过程又可分为标识构件、选择体系结构风格、映射构件、分析构件相互作用和产生体系结构等 5 个子过程，将在第 12 章详细讨论这些过程。

对于上级系统，选择体系结构风格时应考虑上级系统的关键用例或场景(scenario)、SIS 的分级结构以及如何处理从属系统之间的重用和重用内容等问题。对于从属系统，选择体系结构风格时应考虑从属系统在 SIS 系统内的角色、从属系统的关键用例或场景、从属系统如何履行在 SIS 系统的分级结构中为它定义的职责、如何重用等问题。

在决定 SIS 系统的体系结构时，还要综合考虑以下两个问题。

(1) 软件重用。要明确哪些构件是两个以上从属系统公有的，需要建立哪些机制来允许从属系统互相通信。

(2) 所有从属系统都能使用的构件及其实现。所有从属系统都应该使用通用的通信、错误报告、容错管理机制，提供统一的人机界面。

4. 实现

在从属系统的软件过程中，实现过程主要完成构件的开发和测试工作。在上级系统的软件过程中，不必经过实现过程，而只执行一些原型设计工作，研究系统特定方面的技术。

5. 测试

这里的测试指的是，组装不同从属系统时的集成测试，并测试每个上级用例是否根据其

规约通过互连系统协作获得执行。

6. 演化和维护

在SIS系统中,各从属系统之间相对独立。在上级系统的设计模型中,每个从属系统实现一个子系统。子系统依赖于彼此的接口,因此,当某从属系统的需求发生变化时,不必开发新版本的上级系统,只需要对该从属系统内部构件进行变更,不会影响其他从属系统。只有在主要功能变更时才要求开发上级系统的新版本。

3.9.3 应用范围

互连系统构成的系统的体系结构及其建模技术可以用于多种系统,如:

(1) 分布式系统。

(2) 很大或者很复杂的系统。

(3) 综合几个业务领域的系统。

(4) 重用其他系统的系统。

(5) 系统的分布式开发。

情况也可以反过来:从一组现有的系统,通过组装系统来定义由互连系统构成的系统。实际上,在某些情况下,大型系统在演化的早期阶段就是以这种方式发展的。如果开发人员意识到有几个可以互连的系统,那么让系统互连可以创建一个"大型"系统,它比两个独立的系统能创造更多的价值。

事实上,如果一个系统的不同部分本身可以看作系统,笔者建议把它定义为由互连系统构成的系统。即使现在它还是单个的系统,由于分布式开发、重用或者客户只需要购买它的几个部分等原因,把系统分割成几个独立的产品是有必要的。

1. 大规模系统

电话网络可能是世界上最大的由互连系统构成的系统。这是一个很好的示例,在电话网络中需要管理两个以上系统级别的复杂性。它还作为这类案例的示例:顶层上级系统由一个标准化实体掌握,不同的竞争公司开发一个或几个必须符合该标准的从属系统。这里将讨论移动电话网络GSM(全球移动电话系统),说明将大规模系统作为互连系统构成的系统来实施的优势。

大规模系统的功能通常包含几个业务领域。例如,GSM标准包括从呼叫用户到被呼叫用户的整个系统。换句话说,它既包括移动电话的行为,也包括网络结点的行为。由于系统的不同部分是单独购买的,甚至是由不同客户购买的,因而它们本身可当作系统。例如,开发完整GSM系统的公司向用户销售移动电话,向话务员销售网络结点。这是把GSM系统的不同部分当作不同从属系统的一个原因。另一个原因是,把GSM这样大型复杂的系统作为单个系统进行开发的时间太长;不同的部分必须由不同开发团队并行开发。

另外,由于GSM标准包括整个系统,因此有理由将系统作为一个整体即上级系统来考虑。这有助于开发人员理解问题领域,以及不同部分是怎样彼此相关的。

2. 分布式系统

在分布在几个计算机系统的系统中,使用由互连系统构成的系统体系结构是非常合适的。顾名思义,分布式系统(distributed system)至少由两个部分组成。由于分布式系统中

必须有明确定义的接口，因此这些系统也非常适合进行分布式开发，也就是说，几个独立的开发团队并行开发。分布式系统的从属系统本身甚至可以当作产品来销售。因而，将分布式系统视为独立系统的集合是很自然的。

分布式系统的需求通常包括整个系统的功能，有时不预先定义不同部分之间的接口。而且，如果对开发者而言问题领域是陌生的，他们首先需要考虑整个系统的功能，而不用管它将如何分布。这是把它当作单个系统看待的两个重要原因。

3. 遗留系统的重用

许多企业因为业务发展的需要和市场竞争的压力，需要建设新的企业信息系统。在这种升级改造的过程中，怎么处理和利用遗留下来的老系统，成为影响新系统建设成功和开发效率的关键因素之一。一般称这些老系统为遗留系统(legacy system)。

目前，学术和工业界对遗留系统的定义没有统一的意见。Bennett 在 1995 年对遗留系统作了如下的定义：遗留系统是用户不知道如何处理但对用户的组织又是至关重要的系统。Brodie 和 Stonebraker 对遗留系统的定义如下：遗留系统是指任何基本上不能进行修改和演化以满足新的变化了的业务需求的信息系统。

笔者认为，遗留系统应该具有以下特点。

(1) 系统虽然完成企业中许多重要的业务管理工作，但已经不能完全满足要求。一般实现业务处理电子化及部分企业管理功能，很少涉及经营决策。

(2) 系统在性能上已经落后，采用的技术已经过时。如多采用主机/终端形式或小型计算机系统，软件使用汇编语言或第三代程序设计语言的早期版本开发，使用文件系统而不是数据库。

(3) 通常是大型的软件系统，已经融入企业的业务运行和决策管理机制之中，维护工作十分困难。

(4) 系统没有使用现代软件工程方法进行管理和开发，现在基本上已经没有文档，很难理解。

在企业信息系统升级改造过程中，如何处理和利用遗留系统，成为新系统建设的重要组成部分。处理恰当与否，直接关系到新系统的成败和开发效率高低。

大型系统常常重用遗留系统，遗留系统可作为从属系统来描述。为了理解遗留系统如何在上级系统的大环境中工作，需要为它“重建”一个用例模型，有时还有一个分析模型。这些重建的模型不必完整，但至少要包含对互连系统构成的系统的其余部分有直接影响的遗留功能，否则需要进行修改。

总之，在大规模系统的软件开发中，采用 SASIS 可以处理相当复杂的问题，且具有以下优点。

(1) 使用“分而治之”的原则来理解系统，能有效地降低系统的复杂性。

(2) 各从属系统相对独立，自成系统，提高了系统开发的并行性。

(3) 当某从属系统的需求发生变化时，不必开发新版本的上级系统，只需要对该从属系统内部构件进行变更，提高了系统的可维护性。

然而，SASIS 也有固有的缺点，如资源管理开销增大，各从属系统的开发进度无法同步等。

3.10 特定领域软件体系结构

早在20世纪70年代就有人提出程序族、应用族的概念,并开拓了对特定领域软件体系结构的早期研究,这与软件体系结构研究的主要目的"在一组相关的应用中共享软件体系结构"是一致的。为了解脱因为缺乏可用的软件构件以及现有软件构件难以集成而导致软件开发过程中难以进行重用的困境,1990年,Mettala提出了特定领域软件体系结构(Domain Specific Software Architecture,DSSA),尝试解决这类问题。

3.10.1 DSSA的定义

简单地说,DSSA就是在一个特定应用领域中为一组应用提供组织结构参考的标准软件体系结构。对DSSA研究的角度、关心的问题不同导致了对DSSA的不同定义。

Hayes-Roth对DSSA的定义如下:"DSSA就是专用于一类特定类型的任务(领域)的、在整个领域中能有效地使用的、为成功构造应用系统限定了标准的组合结构的软件构件的集合"。

Tracz的定义为:"DSSA就是一个特定的问题领域中支持一组应用的领域模型、参考需求、参考体系结构等组成的开发基础,其目标就是支持在一个特定领域中多个应用的生成"。

通过对众多的DSSA的定义和描述的分析,可知DSSA的必备特征如下。

(1) 一个严格定义的问题域和/或解决域。

(2) 具有普遍性,使其可以用于领域中某个特定应用的开发。

(3) 对整个领域的合适程度的抽象。

(4) 具备该领域固定的、典型的在开发过程中可重用元素。

一般的DSSA的定义并没有对领域的确定和划分给出明确说明。从功能覆盖的范围角度有两种理解DSSA中领域的含义的方式。

(1) 垂直域:定义了一个特定的系统族,包含整个系统族内的多个系统,结果是在该领域中可作为系统的可行解决方案的一个通用软件体系结构。

(2) 水平域:定义了在多个系统和多个系统族中功能区域的共有部分,在子系统级上涵盖多个系统族的特定部分功能,无法为系统提供完整的通用体系结构。

在垂直域上定义的DSSA只能应用于一个成熟的、稳定的领域,但这个条件比较难以满足;若将领域分割成较小的范围,则相对容易,也容易得到一个一致的解决方案。

3.10.2 DSSA的基本活动

实施DSSA的过程中包含一些基本活动。虽然具体的DSSA方法可能定义不同的概念、步骤、产品等,但这些基本活动大体上一致。以下分三个阶段介绍这些活动。

1. 领域分析

这个阶段的主要目标是获得领域模型(domain model)。领域模型描述领域中系统之间的共同需求。称领域模型所描述的需求为领域需求(domain requirement)。在这个阶段中首先要进行一些准备性的活动,包括定义领域的边界,从而明确分析的对象;识别信息源,

即领域分析和整个领域工程过程中信息的来源，可能的信息源包括现存系统、技术文献、问题域和系统开发的专家、用户调查和市场分析、领域演化的历史记录等。在此基础上，就可以分析领域中系统的需求，确定哪些需求是被领域中的系统广泛共享的，从而建立领域模型。当领域中存在大量系统时，需要选择它们的一个子集作为样本系统。对样本系统需求的考察将显示领域需求的一个变化范围。一些需求对所有被考察的系统是共同的，一些需求是单个系统所独有的。很多需求位于这两个极端之间，即被部分系统共享。

领域分析的机制如图 3-34 所示。

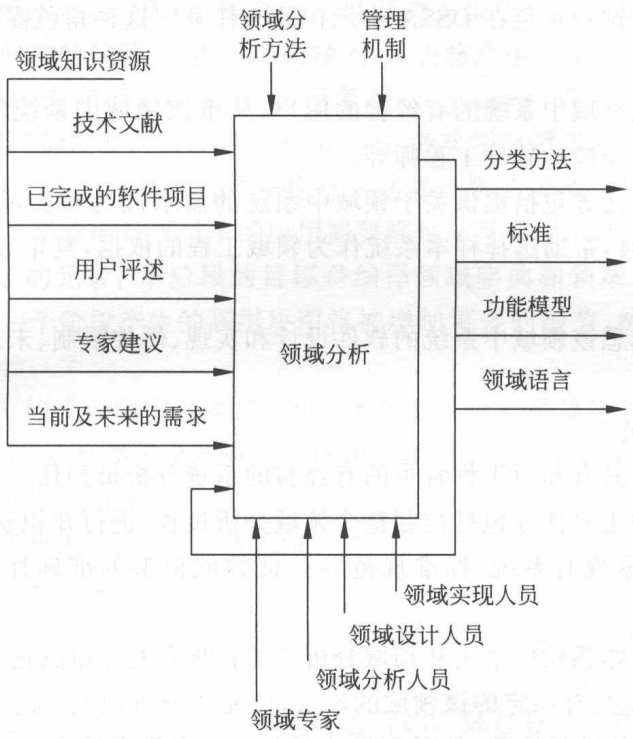


图 3-34　领域分析的机制

2. 领域设计

这个阶段的目标是获得 DSSA。DSSA 描述在领域模型中表示的需求的解决方案。它不是单个系统的表示，而是能够适应领域中多个系统的需求的一个高层次的设计。建立了领域模型之后，就可以派生出满足这些被建模的领域需求的 DSSA。由于领域模型中的领域需求具有一定的变化性，DSSA 也要相应地具有变化性。它可以通过表示多选一的(alternative)、可选的(optional)解决方案等来做到这一点。由于重用基础设施是依据领域模型和 DSSA 来组织的，因此在这个阶段通过获得 DSSA，也就同时形成了重用基础设施的规约。

3. 领域实现

这个阶段的主要目标是依据领域模型和 DSSA 开发和组织可重用信息。这些可重用信息可能是从现有系统中提取得到，也可能需要通过新的开发得到。它们依据领域模型和 DSSA 进行组织，也就是说领域模型和 DSSA 定义了这些可重用信息的重用时机，从而支持了系统化的软件重用。这个阶段也可以看作重用基础设施的实现阶段。

值得注意的是,以上过程是一个反复的、逐渐求精的过程。在实施领域工程的每个阶段中,都可能返回到以前的步骤,对以前的步骤得到的结果进行修改和完善,再回到当前步骤,在新的基础上进行本阶段的活动。

3.10.3 参与DSSA的人员

如图3-34所示,参与DSSA的人员可以划分为4种角色:领域专家、领域分析人员、领域设计人员和领域实现人员。下面对这4种角色分别通过回答3个问题进行介绍:这种角色由什么人员担任?这种角色在DSSA中承担什么任务?这种角色需要哪些技能?

1. 领域专家

领域专家包括该领域中系统的有经验的用户、从事该领域中系统的需求分析、设计、实现以及项目管理的有经验的软件工程师等。

领域专家的主要任务包括提供关于领域中系统的需求规约和实现的知识,帮助组织规范的、一致的领域字典,帮助选择样本系统作为领域工程的依据,复审领域模型、DSSA等领域工程产品,等等。

领域专家应该熟悉该领域中系统的软件设计和实现、硬件限制、未来的用户需求及技术走向等。

2. 领域分析人员

领域分析人员由具有知识工程背景的有经验的系统分析员担任。

领域分析人员的主要任务包括控制整个领域分析过程,进行知识获取,将获取的知识组织到领域模型中,根据现有系统、标准规范等验证领域模型的准确性和一致性,维护领域模型。

领域分析人员应熟悉软件重用和领域分析方法;熟悉进行知识获取和知识表示所需的技术、语言和工具;应具有一定的该领域的经验,以便于分析领域中的问题及与领域专家进行交互;应具有较高的进行抽象、关联和类比的能力;应具有较高的与他人交互和合作的能力。

3. 领域设计人员

领域设计人员由有经验的软件设计人员担任。

领域设计人员的主要任务包括控制整个软件设计过程,根据领域模型和现有的系统开发出DSSA,对DSSA的准确性和一致性进行验证,建立领域模型和DSSA之间的联系。

领域设计人员应熟悉软件重用和领域设计方法;熟悉软件设计方法;有一定的该领域的经验,以便于分析领域中的问题及与领域专家进行交互。

4. 领域实现人员

领域实现人员由有经验的程序设计人员担任。

领域实现人员的主要任务包括根据领域模型和DSSA,或者从头开发可重用构件,或者利用再工程的技术从现有系统中提取可重用构件,对可重用构件进行验证,建立DSSA与可重用构件间的联系。

领域实现人员应熟悉软件重用、领域实现及软件再工程技术;熟悉程序设计;具有一定的该领域的经验。

3.10.4 DSSA 的建立过程

因所在的领域不同，DSSA 的创建和使用过程也各有差异。Tracz 曾提出了一个通用的 DSSA 应用过程，这些过程也需要根据所应用到的领域来进行调整。一般情况下，需要用所应用领域的应用开发者习惯使用的工具和方法来建立 DSSA 模型。同时，Tracz 强调了 DSSA 参考体系结构文档工作的重要性，因为新应用的开发和对现有应用的维护都要以此为基础。

DSSA 的建立过程分为 5 个阶段，每个阶段可以进一步划分为一些步骤或子阶段。每个阶段包括一组需要回答的问题，一组需要的输入，一组将产生的输出和验证标准。该过程是并发的(concurrent)、递归的(recursive)、反复的(iterative)，或者可以说，它是螺旋型(spiral)。完成该过程可能需要对每个阶段经历几遍，每次增加更多的细节。

(1) 定义领域范围：本阶段的重点是确定什么在感兴趣的领域中以及本过程到何时结束。这个阶段的一个主要输出是领域中的应用需要满足一系列用户的需求。

(2) 定义领域特定的元素：本阶段的目标是编译领域字典和领域术语的同义词词典。在领域工程过程的前一个阶段产生的高层块图将被增加更多的细节，特别是识别领域中应用间的共同性和差异性。

(3) 定义领域特定的设计和实现需求约束：本阶段的目标是描述解空间中有差别的特性。不仅要识别出约束，并且要记录约束对设计和实现决定造成的后果，还要记录对处理这些问题时产生的所有问题的讨论。

(4) 定义领域模型和体系结构：本阶段的目标是产生一般的体系结构，并说明构成它们的模块或构件的语法和语义。

(5) 产生、搜集可重用的产品单元：本阶段的目标是为 DSSA 增加构件使得它可以被用来产生问题域中的新应用。

DSSA 的建立过程是并发的、递归的和反复进行的。该过程的目的是将用户的需要映射为基于实现约束集合的软件需求，这些需求定义了 DSSA。在此之前的领域工程和领域分析过程并没有对系统的功能性需求和实现约束进行区分，而是统称为“需求”。图 3-35 是 DSSA 的一个三层次系统模型。

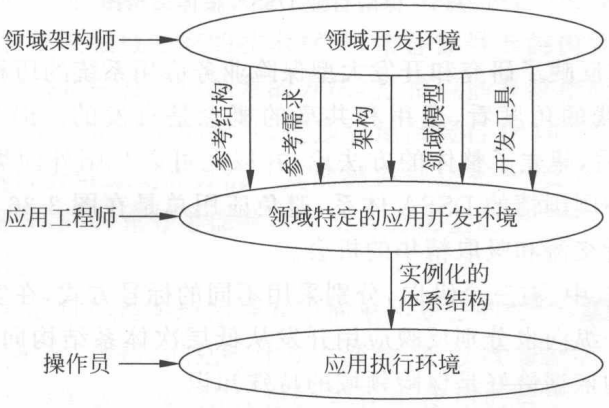


图 3-35 DSSA 的三层次的系统模型

DSSA的建立需要设计人员对所在特定应用领域(包括问题域和解决域)必须精通,他们要找到合适的抽象方式来实现DSSA的通用性和可重用性。通常DSSA以一种逐渐演化的方式发展。

3.10.5 DSSA实例

本节将介绍一个保险行业特定领域软件体系结构。本节所指的保险行业应用系统,特指财产险的险种业务管理系统,它同样适用于人寿险的业务管理系统。图3-36是一个简化了的保险行业DSSA整体结构图。

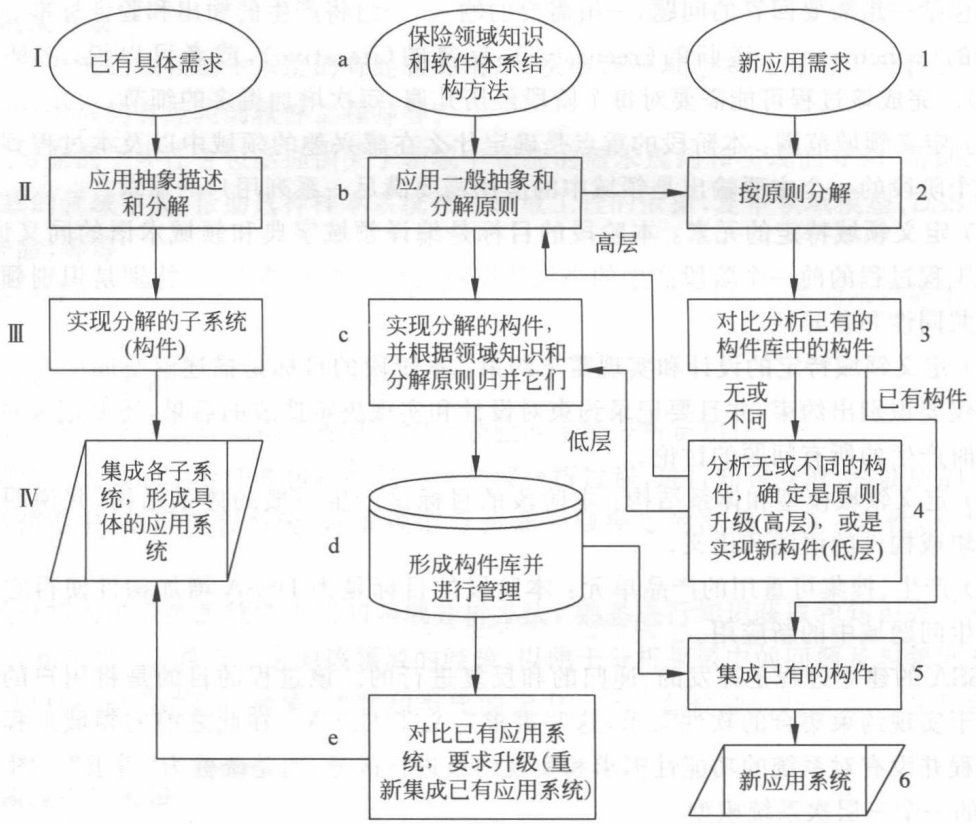


图3-36 保险行业DSSA整体结构图

图3-36从左到右反映了研究和开发大型保险业务应用系统的历程,中间的环节引入了DSSA的概念。从实践的角度看,通用和共享的概念是自发的。但是当研究了国外有关DSSA的最新发展之后,建立了整体的方法论,并以此可以与国外同类技术发展进行沟通,取长补短,形成适合中国国情的DSSA体系,避免应用总是在图3-36中最左一列反复地开发,失去了与国外同行交流和吸取精华的机会。

图3-36中分为左、中、右三个纵向,分别采用不同的标号方式,在实例说明时,将对照标号进行详细讨论,三个纵向也分别反映应用开发从低层次体系结构向高层次体系结构转化的过程,而这种转化的依据恰好是保险领域的特殊知识。

1. 传统构造保险应用方法

对应于保险行业DSSA结构图3-36的左列,回顾传统的应用构造方法。标号Ⅰ反映传

统应用的具体要求，所谓险种业务管理系统是保险产品在整个销售和服务过程中的计算机管理系统，险种相当于产品的品种。国内的保险从大类上可以分为机动车保险、企业财产保险、家庭财产保险、货物运输保险、船舶保险、建筑工程和安装工程保险、责任保险、信用保险、飞机/卫星保险等；从保险条款上区分又可分成为国内保险和涉外保险，总之具体的险种产品多达上百种。从前台销售和服务过程上看，可分成以下主要环节，展业、承保、交/退费、批改、理赔、汇总统计。这样一个涉及如此多专业内容的应用系统，开发量是巨大的，更由于险种条款的多变性，产品服务设计中更多地注意条款的专业性，而给信息系统建设带来困难，所以多年来系统开发技术水平发展缓慢，存在大量的低水平反复，从整体需求上看，只能保证在某一时刻上的稳定性。

传统上，人们在标号Ⅱ的工作主要是划分险种的分类，及分类后的过程管理，分类的依据也是险种产品的条款特征。把一种分类的管理，称为险种纵向管理，主要仍是包括展业、承保、交/退费、批改、理赔、汇兑统计等，它们对应一套独立的子系统，子系统的组成可以粗粒度分成以上 6 个部分，这样对一个财产险业务管理构成近 20 个子系统，且按管理过程都分成非常类似的 6 个部分，如果全面实现应用将面对上百个粗粒度独立部分的设计和开发。

在标号Ⅲ中，实际上是组织开发这上百个粗粒度的部分，从条款上看，只是相似但绝不相同，由于开发所用的平台是关系数据库，所以库表都十分类似，但是业务含义不同，必须分别开发。从工程组织上看当然可以使用一些低层次的共享手段，突出表现在函数/模块化的管理、编程约定、机构移植等方面。

传统的标号Ⅳ的内容主要是集成各子系统，由于各部分组合较紧密，所以对子系统内部集成工作等同于更大范围的开发，但结果是构成了近二十个子系统，它们分别管理着险种纵向的业务操作，这就是最初级的保险应用系统。

从以上的过程不难看出其中的不足，其关键是大量基本单元相似和共享问题，其隐含的需求还包括基本单元的需求稳定性，这种需求实际上是一种更高层面的软件体系结构上的需求。

2. 采用 DSSA 后的变化

为了解决保险行业 DSSA 整体结构图左列方法引起的不足，引入了 DSSA 方法，它形成了图 3-36 中的中间一列。标号 a 实际上反映了对领域工程一般原理的学习，以及保险领域知识的深入理解，它反映领域专家的技术背景，结合标号 b 的内容，对保险行业内共同的特征进行提取，所用方法是一种高层抽象的方法，把相似抽象成相同，就像把算术问题抽象成代数问题一样，形成概念定义上的提升，这是实现保险行业 DSSA 的关键步骤之一，由此形成了相对抽象的构件分解原则和方法。其前提条件是充分理解行业的发展，对细节概念和定义加以屏蔽和提升，同时充分考虑数据库实现方法上的技巧性，保证实现上的可操作性和新技术方法的应用。

在标号 c 中，主要的目标是实现标号 b 中抽象和划分的构件，实际上是针对标号Ⅲ中的展业、承保、交/退费、批改、理赔、汇兑统计 6 大部分，把领域知识加入，重新实现它们，很显然这是从思想到实现的关键一步，结合在标号Ⅳ中使用的模块化设计，归纳为如下实现方法。

首先，构件要实体分类，这是因为实现机制是关系数据库。在抽象时，要保证数据结构

分类的一致性,但是在用户操作方面又要考虑外层的语义,所以提供相对专用的界面。因此,主要的应用操作逻辑,应适应数据结构的通用性和操作界面的专用性,并按对象模式进行分类实现。例如,在承保构件中,由于主、从表的不同有两种方式:一种是主表方式,它没有更细的表达;另一种是既有主表,又有从表,且更新数据时先从后主,查看数据时,先主后从。主、从表设计,在数据结构中,充分利用"权利、义务"对等原则设计属性项,并提供若干概念相对通用的编码属性,以解决分类标识问题和需求扩充问题,以此保证结构的稳定性。同样,对交/退费、批改构件、理赔构件、汇兑统计构件等使用类似的方法分别实现。

其次,构成实体构件的操作部分,还要进一步加强模块化,提供通用的工具,大至菜单、报表、查询打印工具,小至通用的计算函数、约定编程技巧等。这部分的内容,从构件实现上保证了大粒度,因为即使使用第四代语言开发应用系统,在模式一级也需要提供更大粒度的实现体,它对构件内部的更新和升级提供了基础条件。

总之,在整个实现过程中,充分使用应用构件内、外部的重用技术,充分利用领域知识抽象通用特性和提高应用操作的有效性,最终完成各构件的编程实现。

标号d只反映构件管理的辅助内容,因为当系统开发越来越庞大之后,管理的工作越来越多,它也包括传统软件开发版本管理的全部内容。

标号e实际上集成应用子系统,因为构件只有在装配和配置之后,才能成为最后的应用系统。当作为产品实施时,它充分反映客户化的内容,目前国外大型的应用系统有时要客户花一年或更多的时间。

3. 采用DSSA后的应用构成和系统升级

建立了一套构件库之后,新的业务应用系统生成就采用新的方式,参考图3-36的右列,做如下讨论。

标号1反映新的应用需求,它可能是新险种应用,也可能是对老险种的管理新要求,总之要适应保险行业竞争发展的管理需要。标号2表示使用标号b的抽象和分解原则明确化需求,使其具有统一的划分构件的方法。标号3的内容是与构件库的构件进行对比分析,简单的情况表示已有相应的构件,这样可以通过标号5集成和客户化新的应用系统;复杂的情况,没有对应的构件或已有构件只是其中的一部分或构件的粒度太大,都将进入标号4。标号4是决定构件库做什么层次升级的决定机构,高层反映要对过去抽象和分解原则进行变化,低层可能直接完善构件库,以下可以对返b(高层)和返c(低层)进行实例说明。

- 返b的实例。例如,在承保的构件中,近期要加入独立的客户管理内容,保险公司内部的人员管理、系统的操作权限管理、核保管理等,这实际上要对原来的构件划分原则作新的调整,并由此产生新的构件,同时引起标号c中的新的实现,同样的内容也会发生在批改、理赔的构件中,由此引起了构件库的全面升级。
- 返c的实例。例如,在承保的构件中,在从表中加入新的一对多子表,从而使原来的主、从结构模式变成了新的三级结构,并从实现机制上增加了更详细信息的表达方式。从发展的眼光看,应用不是一次开发到位的,系统实现之初,构件的粒度可能相对较大,但随着应用的深入和对业务的本质把握,将逐步细化实现粒度,以适应整个系统的灵活性和效率。

总之，标号 b 和标号 c 的变动都将引起构件库标号 d 的变动，标号 e 反映这种变化后，应用系统是否升级。在实际应用中，标号Ⅳ的系统也是定期升级的，当所有的标号Ⅳ的应用系统都不使用标号 d 中的某些构件时，就可以做构件的清理，使构件库更加高效。

由标号 5 集成的新应用系统，构成了标号 6。从应用角度看，它们也是标号Ⅳ的同类，所不同的是当构件库发生变化时，老应用的整体升级是有选择的，即使它们不升级，作为遗留系统，它们仍然运行良好，而且这种升级受整体客户花费限制，不一定马上进行。从前面给出的实例看，很多升级意味着整体系统功能的增加，对新的应用实施单位往往要进行一定选择。

采用了 DSSA 结构之后，获得了如下好处。

(1) 相对于过去的开发方法，系统开发、维护的工作量大幅度减少，整个应用系统的构件重用程序相当大。

(2) 便于系统开发的组织管理。在大型系统开发过程中，最突出的问题是人员的组织问题。采用了 DSSA 之后，开发中涉及核心技术的人员从 15 人左右下降到 5 人左右，其他的人力进入外围产品化的工作，如产品包装、市场销售、工具的开发、客户化服务等。而过去这方面内容在技术部门是被忽视的，而在应用软件工程中，它们也占重要的位置。

(3) 系统有较好的环境适应性。构件的升级引发应用系统的升级，并在构件库中合理地控制粒度，使系统的总体结构设计与算法和模块化设计同等重要，并灵活地保证新、老应用系统的共存。

3.10.6 DSSA 与体系结构风格的比较

在软件体系结构的发展过程中，由于研究者的出发点不同，出现了两个互相直交的方法和学科分支：以问题域为出发点的 DSSA 和以解决域为出发点的软件体系结构风格。两者侧重点不同，它们在软件开发中具有不同的应用特点。

DSSA 只对某一个领域进行设计专家知识的提取、存储和组织，但可以同时使用多种体系结构风格；而在某个体系结构风格中进行体系结构设计专家知识的组织时，可以将提取的公共结构和设计方法扩展到多个应用领域。

DSSA 的特定领域参考体系结构通常选用一个或多个适合所研究领域的体系结构风格，并设计一个该领域专用的体系结构分析设计工具。但该方法提取的专家知识只能用于一个较小的范围——所在领域中。不同参考体系结构之间的基础和概念有较少的共同点，所以为一个领域开发 DSSA 及其工具在另一个领域中是不适应的或不可重用的，而工具的开发成本是相当高的。

体系结构风格的定义和该风格应用的领域是直交的，提取的设计知识比用 DSSA 提取的设计专家知识的应用范围要广。一般的、可调整的系统基础可以避免涉及特定的领域背景，所以建立一个特定风格的体系结构设计环境的成本比建立一个 DSSA 参考体系结构和工具库的成本要低得很多。因为对特定领域内的专家知识和经验的忽略，使其在一个具体的应用开发中所起的作用并不比 DSSA 大。

DSSA 和体系结构风格是互为补充的两种技术。在大型软件开发项目中基于领域的设计专家知识和以风格为中心的体系结构设计专家知识都扮演着重要的角色。

思 考 题

1. 选择一个熟悉的大型软件系统,分析其体系结构中用到的风格,以及表现出的特点。(为什么要采用这种风格?采用这种风格带来哪些优势?具有哪些不足?)

2. 选择两种风格,设计简单的体系结构,并实现简单的原型系统。

3. 查阅相关资料,尝试从面向服务的体系结构(SOA)的实现技术的角度出发,深入分析 CORBA、DCOM 以及基于 UDDI 的 Web Service 等分布式计算技术,在此基础上,对这些技术做比较和归纳,涉及 CORBA、DCOM 和基于 UDDI 的 Web Service 之间的互操作的方案。

4. 本章独立介绍了每种体系结构,但事实上,所有的体系结构不仅有紧密的联系,而且在大多数情况下是被一起使用的。对于一个实际的系统甚至不能判断它是哪种风格,因此就存在异构风格的共存。图 3-37 展示了一个虚拟系统,它整合了哪种体系结构风格。试说明该系统中包含哪些体系结构风格?

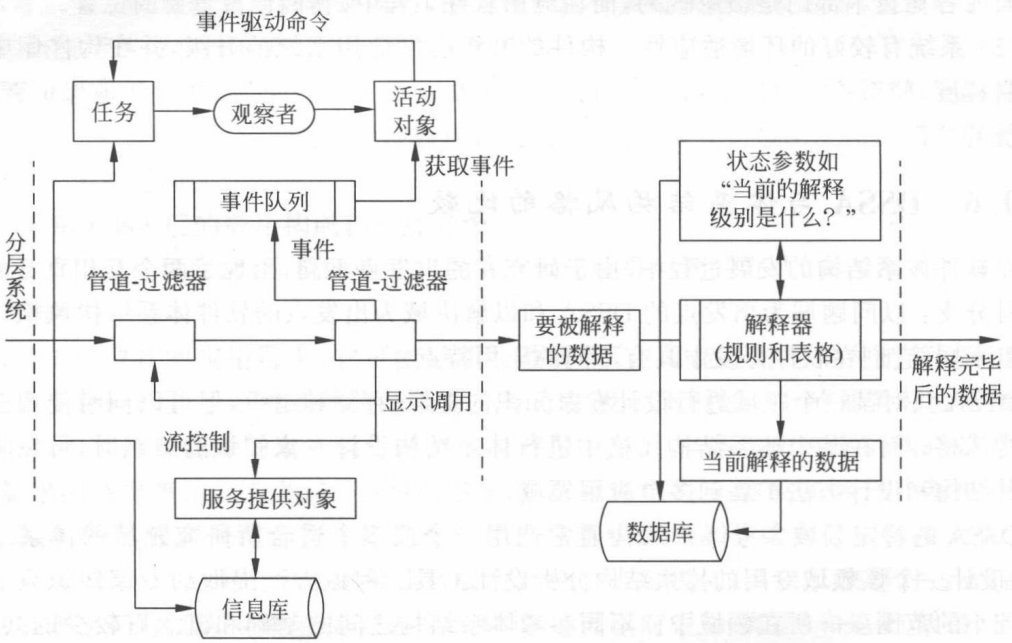


图 3-37 一个虚拟系统

5. 不同的体系结构风格具有各自的特点、优劣和用途。试对管道-过滤器风格、事件驱动风格、分层系统、C2 风格和基于消息总线的风格进行分析比较。

6. 试说明特定领域软件体系结构包括哪些基本活动,应该如何选择参与人员及其创建过程。

7. 希赛公司计划开发一个车辆定速巡航控制系统,以确保车辆在不断变化的地形中以固定的速度行驶。图 3-38 给出了该系统的简化示意图,表 3-2 描述了各种系统输入的含义。

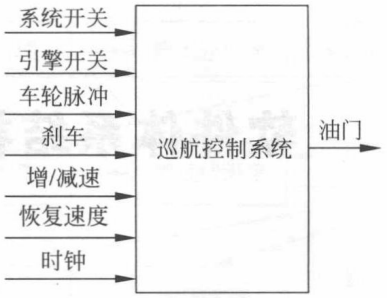


图 3-38　定速巡航控制系统简化示意图

表 3-2　定速巡航控制系统输入说明

输入名称	作　　用
系统开关	开启/关闭巡航控制系统
引擎开关	开启/关闭汽车引擎(引擎开启时,巡航控制系统处于就绪状态)
车轮脉冲	车轮每转一次,相应地发出一次脉冲
刹车	当刹车被踩下时,定速巡航控制系统会临时恢复到人工控制
增/减速	增加或减慢当前车速(仅在定速巡航控制系统处于开启的状态下可用)
恢复速度	恢复原来保持的车速(仅在定速巡航控制系统处于开启的状态下可用)
时钟	每毫秒定时脉冲

公司的领域专家对需求进行深入分析后,将系统需求认定为:任何时刻,只要定速巡航控制系统处于工作状态,就要有确定的期望速度,并通过调整引擎油门的设定值来维持期望速度。

在对车辆定速巡航控制系统的体系结构进行设计时,公司的设计师王工提出采用面向对象的体系结构风格,而李工则主张采用控制环路的体系结构风格。在体系结构评估会议上,专家对这两种方案进行综合评价,最终采用了面向对象和控制环路相结合的混合体系结构风格。

【问题 1】

在实际的软件项目开发中,采用成熟的体系结构风格是项目成功的保证。请用 200 字以内的文字说明:什么是软件体系结构风格;面向对象和控制环路两种体系结构风格各自的特点。

【问题 2】

用户需求没有明确给出该系统如何根据输入集合计算输出。请用 300 字以内的文字针对该系统的增减速功能,分别给出两种体系结构风格中的主要构件,并详细描述计算过程。

【问题 3】

实际的软件系统体系结构通常是多种体系结构风格的混合,不同的体系结构风格都有其适合的应用场景。以该系统为例,针对面向对象体系结构风格和控制环路体系结构风格,各给出两个适合的应用场景,并简要说明理由。

参考文献

第4章 软件体系结构描述

从软件体系结构研究和应用的现状来看，当前对软件体系结构的描述，在很大程度上还停留在非形式化的基础上，很大程度上依赖于体系结构设计师个人的经验和技巧。目前，通用的软件开发方法中，对软件体系结构的描述通常是采用非形式化的图和文本，不能描述系统期望的存在于构件之间的接口，更不能描述不同的组成系统的组合关系的意义。这种描述方法难以被开发人员理解，难以进行形式化分析和模拟，缺乏相应的支持工具帮助设计师完成设计工作，更不能用来分析其一致性和完整性等特性。

因此，形式化的、规范化的体系结构描述对于体系结构的设计和理解非常重要。然而，要实现体系结构设计、描述等的形式化并不能一蹴而就，必须先经历一个非形式化的过程，在非形式化的发展过程中逐步提取一些形式化的标记和符号，然后将它们标准化，从而完成体系结构设计、描述等的形式化。

4.1 软件体系结构描述方法

从本质上来说，软件体系结构的描述方法可以分为文字表达工具、数学表达工具和图形表达工具。其中，在产业界用得最多的是图形表达方式，学术界更偏向使用数学表达工具。

1. 图形表达工具

对于软件体系结构的描述和表达，一种简洁易懂且使用广泛的方法是采用由矩形框和有向线段组合而成的图形表达工具。在这种方法中，矩形框代表抽象构件，框内标注的文字为抽象构件的名称，有向线段代表辅助各构件进行通信、控制或关联的连接件。例如，图4-1表示某软件辅助理解和测试工具的部分体系结构描述。

目前，这种图形表达工具在软件设计中占据着主导地位。尽管由于在术语和表达语义上存在着一些不规范和不精确，而使得以矩形框与线段为基础的传统图形表达方法在不同系统和不同文档之间有着许多不一致甚至矛盾，但该方法仍然以其简洁易用的特点在实际的设计和开发工作中被广泛使用，并为工作人员传递了大量重要的体系结构思想。

为了克服传统图形表达方法中所缺乏的语义特征，有关研究人员试图通过增加含有语义的图元素的方式来开发图文法理论。

2. 模块内连接语言

软件体系结构的第二种描述和表达方法是采用将一种或几种传统程序设计语言的模块连接起来的模块内连接语言(Module Interconnection Language，MIL)。由于程序设计语言和模块内连接语言具有严格的语义基础，因此它们能支持对较大的软件单元进行描述，诸如

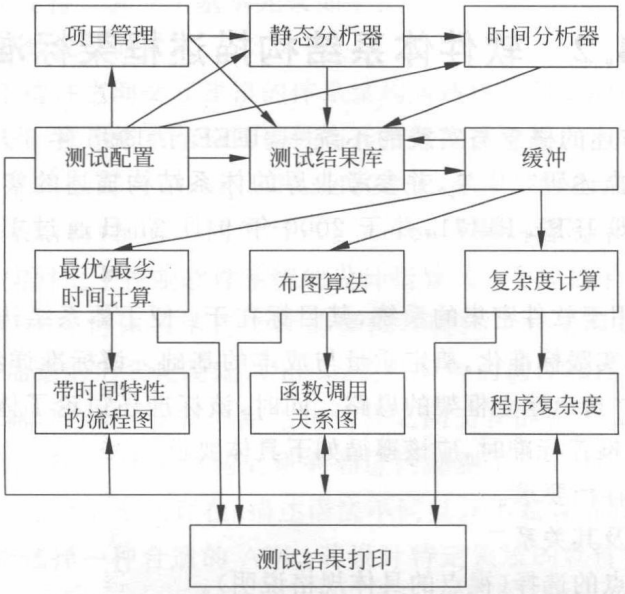


图 4-1　某软件辅助理解和测试工具部分体系结构描述

定义/使用、扇入/扇出等操作。例如，Ada 语言采用 use 实现包的重用，Pascal 语言采用过程(函数)模块的交互，等。

MIL 方式对模块化的程序设计和分段编译等程序设计与开发技术确实发挥了很大的作用。但是这些语言处理和描述的软件设计开发层次过于依赖程序设计语言，因此限制了它们处理和描述比程序设计语言元素更为抽象的高层次软件体系结构元素的能力。

3. 基于软构件的系统描述语言

软件体系结构的第三种描述和表达方法是采用基于软构件的系统描述语言。基于软构件的系统描述语言将软件系统描述成一种是由许多以特定形式相互作用的特殊软件实体构造组成的组织或系统。

例如，一种多变配置语言(Proteus Configuration Language，PCL)就可以用来在一个较高的抽象层次上对系统的体系结构建模，Darwin 最初用作设计和构造复杂分布式系统的配置说明语言，因具有动态特性，也可用来描述动态体系结构。

这种表达和描述方式虽然也是较好的一种以构件为单位的软件系统描述方法，但是它们所面向和针对的系统元素仍然是一些层次较低的以程序设计为基础的通信协作软件实体单元，而且这些语言所描述和表达的系统一般而言都是面向特定应用的特殊系统，这些特性使得基于软构件的系统描述仍然不是十分适合软件体系结构的描述和表达。

4. 软件体系结构描述语言

软件体系结构的第四种描述和表达方法是参照传统程序设计语言的设计和开发经验，重新设计、开发和使用针对软件体系结构特点的专门的软件体系结构描述语言(Architecture Description Language，ADL)。由于 ADL 是在吸收了传统程序设计中的语义严格精确的特点基础上，针对软件体系结构的整体性和抽象性特点，定义和确定适合于软件体系结构表达与描述的有关抽象元素，因此，ADL 是当前软件开发和设计方法学中一种发展很快的软件体系结构描述方法，目前，已经有几十种常见的 ADL。

4.2 软件体系结构描述框架标准

鉴于体系结构描述的概念与实践的不统一,IEEE于1995年8月成立了体系结构工作组,综合体系结构描述研究成果,并参考业界的体系结构描述的实践,负责起草了体系结构描述框架标准即IEEE P1471,并于2000年9月21日通过IEEE-SA标准委员会评审。

IEEE P1471适用于软件密集的系统,其目标在于:便于体系结构的表达与交流,并通过体系结构要素及其实践标准化,奠定质量与成本的基础。该标准详细介绍了一套体系结构描述的概念框架,并给出建立框架的思路。同时,该标准还讨论了体系结构描述实践,在应用体系结构描述的推荐标准时,应该遵循如下具体要求。

(1) 体系结构的存档要求。

(2) 能识别人员及其关系。

(3) 体系结构视点的选择(视点的具体规格说明)。

(4) 体系结构视点。

(5) 体系结构视点之间的一致性。

(6) 体系结构原理。

IEEE P1471仅提供了体系结构描述的概念框架、其体系结构描述实践应该遵循的规范,但在如何描述以及具体的描述技术等方面缺乏更进一步的指导。

在IEEE P1471推荐的体系结构描述的概念框架基础上,Rational起草了可重用的软件资产规格说明,专门讨论了体系结构描述的规格说明,提出了一套易于重用的体系结构描述规范。

可重用的体系结构描述框架建议,基于RUP(Rational United Process)、采用UML模型描述软件的体系结构,认为体系结构描述的关键是定义视点、视图以及建模元素之间的映射关系,可以从4个视点出发描述体系结构,即需求视点、设计视点、实现视点和测试视点。并在此基础上提出了7个体系结构视图,即用例视图、域视图、非功能需求视图、逻辑视图、实现视图、过程视图和部署视图。然后,从系统建模的角度考虑多个视图之间的映射关系,并建议了这些视图的表示和视图之间的映射关系的表示。

Rational的建议标准采纳了IEEE P1471中提出的体系结构描述概念模型,覆盖了"4+1"体系结构模型中的4个视图,并在原则上讨论了视图以及视图之间的映射的表示问题。与IEEE P1471相比,该建议标准的体系结构描述方案涉及面比较窄,所注重的层次比较低,因而更具体。由于将体系结构的描述限于UML和RUP,具有一定的局限性,但该建议标准结合了业界已经广泛采用的建模语言和开发过程,因而易于推广,可以有效实现在跨组织之间重用体系结构描述结果。

4.3 体系结构描述语言

ADL是这样一种形式化语言,它在底层语义模型的支持下,为软件系统的概念体系结构建模提供了具体语法和概念框架。基于底层语义的工具为体系结构的表示、分析、演化、

细化、设计过程等提供支持。其三个基本元素如下。

(1) 构件：计算或数据存储单元。

(2) 连接件：用于构件之间交互建模的体系结构构造块及其支配这些交互的规则。

(3) 体系结构配置：描述体系结构的构件与连接件的连接图。

主要的体系结构描述语言有 Aesop、MetaH、C2、Rapide、SADL、Unicon、Wright 等，尽管它们都描述软件体系结构，却有不同的特点。Aesop 支持体系结构风格的应用，MetaH 为设计师提供了关于实时电子控制软件系统的设计指导，C2 支持基于消息传递风格的用户界面系统的描述，Rapide 支持体系结构设计的模拟并提供了分析模拟结果的工具，SADL 提供了关于体系结构加细的形式化基础，Unicon 支持异构的构件和连接类型并提供了关于体系结构的高层编译器，Wright 支持体系结构构件之间交互的说明和分析。这些 ADL 强调了体系结构不同的侧面，对体系结构的研究和应用起到了重要的作用，但也有负面的影响。每一种 ADL 都以独立的形式存在，描述语法不同且互不兼容，同时又有许多共同的特征，这使设计人员很难选择一种合适的 ADL，若设计特定领域的软件体系结构又需要从头开始描述。

4.3.1 ADL 与其他语言的比较

按照 Mary Shaw 和 David Garlan 的观点，典型的 ADL 在充分继承和吸收传统程序设计语言的精确性和严格性特点的同时，还应该具有构造、抽象、重用、组合、异构和分析推理等各种能力和特性。其中：

(1) 构造能力指的是 ADL 能够使用较小的独立体系结构元素来建造大型软件系统。

(2) 抽象能力指的是 ADL 使得软件体系结构中的构件和连接件描述可以只关注它们的抽象特性，而不管其具体的实现细节。

(3) 重用能力指的是 ADL 使得组成软件系统的构件、连接件甚至是软件体系结构都成为软件系统开发和设计的可重用部件。

(4) 组合能力指的是 ADL 使得其描述的每一系统元素都有其自己的局部结构，这种描述局部结构的特点使得 ADL 支持软件系统的动态变化组合。

(5) 异构能力指的是 ADL 允许多个不同的体系结构描述关联存在。

(6) 分析和推理能力指的是 ADL 允许对其描述的体系结构进行多种不同的性能和功能上的多种推理分析。

根据这些特点，可以将下面这样的语言排除在 ADL 之外：高层设计符号语言、MIL、编程语言、面向对象的建模符号、形式化说明语言。ADL 与需求语言的区别在于后者描述的是问题空间，而前者则扎根于解空间。ADL 与建模语言的区别在于后者对整体行为的关注要大于对部分的关注，而 ADL 集中在构件的表示上。ADL 与传统的程序设计语言的构成元素既有许多相同和相似之处，又各自有着很大的不同。

下面给出程序设计语言和 ADL 的典型元素的属性和含义比较，以及软件体系结构中经常出现的构件和连接件元素，分别如表 4-1 和表 4-2 所示。

表 4-1 典型元素含义比较

程序设计语言		软件体系结构	
程序构件	组成程序的基本元素及其取值或值域范围	系统构件	模块化级别的系统组成成分实体,这些实体可以被施以抽象的特性化处理,并以多种方式得到使用
操作符	连接构件的各种功能符号	连接件	对组成系统的有关抽象实体进行各种连接的连接机制
抽象规则	有关构件和操作符的命名表达规则	组合模式	系统中的构件和连接件进行连接组合的特殊方式,也就是软件体系结构的风格
限制规则	一组选择并决定具体使用何种抽象规则来作用于有关的基本构件及其操作符的规则和原理	限制规则	决定有关模式能够作为子系统进行大型软件系统构造和开发的合法子系统的有关条件
规范说明	有关句法的语义关联说明	规范说明	有关系统组织结构方面的语义关联说明

表 4-2 常见的软件体系结构元素

系统构件元素		连接件元素	
纯计算单元	这类构件只有简单的输入/输出处理关联,对它们的处理一般也不保留处理状态,如数学函数、过滤器、转换器等	过程调用	在构件实体之间实现单线程控制的连接机制,如普通过程调用、远程过程调用等
数据存储单元	具有永久存储特性的结构化数据,如数据库、文件系统、符号表、超文本等	数据流	系统中通过数据流进行交互的独立处理流程连接机制。其最显著的特点是根据得到的数据来进行构件实体的交互控制,如 UNIX 操作系统中的管道机制等
管理器	对系统中的有关状态和紧密相关操作进行规定与限制的实体,如抽象数据类型、系统服务器等	隐含触发器	由并发出现的事件来实现构件实体之间交互的连接机制。在这种连接机制中,构件实体之间不存在明显确定的交互规定,如时间调度协议、自动垃圾回收处理等
控制器	控制和管理系统中有关事件发生的时间序列,如调度程序、同步处理协调程序等	消息传递	独立构件实体之间通过离散和非在线的数据(可以是同步或非同步的)进行交互的连接机制,如 TCP/IP 等
连接器	充当有关实体间信息转换角色的实体,如通信连接器、用户界面等	数据共享协议	构件之间通过相同的数据空间进行并发协调操作的机制,如黑板系统中的黑板、多用户数据库系统中的共享数据区等

4.3.2 ADL 的构成要素

体系结构描述的基本构成要素有构件、连接件和体系结构配置。软件体系结构的核心模型参见图 2-14。

1. 构件

构件是一个计算单元或数据存储。也就是说，构件是计算与状态存在的场所。在体系结构中，一个构件可能小到只有一个过程或大到整个应用程序。它可以要求自己的数据与/或执行空间，也可以与其他构件共享这些空间。作为软件体系结构构造块的构件，其自身也包含多种属性，如接口、类型、语义、约束、演化和非功能属性等。

接口是构件与外部世界的一组交互点。与面向对象方法中的类说明相同，ADL 中的构件接口说明了构件提供的服务(消息、操作、变量)。为了能够充分地推断构件及包含它的体系结构，ADL 提供了能够说明构件所需的工具。这样，接口就定义了构件能够提出的计算委托及其用途上的约束。

构件作为一个封装的实体，只能通过其接口与外部环境交互。构件的接口由一组端口组成，每个端口表示了构件和外部环境的交互点。通过不同的端口类型，一个构件可以提供多重接口。一个端口可以非常简单，如过程调用；也可以表示更为复杂的界面，如必须以某种顺序调用的一组过程调用。

构件类型是实现构件重用的手段。构件类型保证了构件能够在体系结构描述中多次实例化，并且每个实例可以对应于构件的不同实现。抽象构件类型也可以参数化，进一步促进重用。现有的 ADL 都将构件类型与实例区分开来。

由于基于体系结构开发的系统大都是大型、长时间运行的系统，因而系统的演化能力显得格外重要。构件的演化能力是系统演化的基础。ADL 通过构件的子类型及其特性的细化来支持演化过程。目前，只有少数几种 ADL 部分地支持演化，对演化的支持程度通常依赖于所选择的程序设计语言。其他 ADL 将构件模型看作静态的。

2. 连接件

连接件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。与构件不同，连接件可以不与实现系统中的编译单元对应。它们可能以兼容消息路由设备实现(如 C2)，也可以以共享变量、表入口、缓冲区、对连接器的指令、动态数据结构、内嵌在代码中的过程调用序列、初始化参数、客户服务协议、管道、数据库、应用程序之间的 SQL 语句等形式出现。大多数 ADL 将连接件作为第一类实体，也有 ADL 不将连接件作为第一类实体。

连接件作为建模软件体系结构的主要实体，同样也有接口。连接件的接口由一组角色组成，连接件的每一个角色定义了该连接件表示的交互参与者；二元连接有两个角色，如消息传递连接件的角色是发送者和接收者；有的连接件有多于两个的角色，如事件广播有一个事件发布者角色和任意多个事件接收者角色。

显然，连接件的接口是一组它与所连接构件之间的交互点。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。它能够推导出软件体系结构的形成情况。体系结构配置中要求构件端口与连接件角色的显式连接。

体系结构级的通信需要用复杂协议来表达。为了抽象这些协议并使之能够重用，ADL 应该将连接件构造为类型。构造连接件特性可以将作为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。

为完成对构件接口的有用分析，保证跨体系结构抽象层的细化一致性，强调互联与通信约束等，体系结构描述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。ADL 可以通过强制风格

不变性来实现约束,或通过接受属性限制给定角色中的服务。

3. 体系结构配置

体系结构配置或拓扑是描述体系结构的构件与连接件的连接图。体系结构配置提供信息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。

体系结构适合于描述大的、生命周期长的系统,利用配置支持系统的变化,使不同技术人员都能理解并熟悉系统。为帮助开发人员在一个较高的抽象层上理解系统,就需要对软件体系结构进行说明。为了使开发者与有关人员之间的交流容易些,ADL必须以简单的、可理解的语法来配置结构化信息。理想的情况是从配置说明中澄清系统结构,即不需研究构件与连接件就能使构建系统的各种参与者理解系统。体系结构配置说明除文本形式外,有些ADL还提供了图形说明形式。文本描述与图形描述可以互换。多视图、多场景的体系结构说明方法在最新的研究中得到了明显的加强。

为了在不同细节层次上描述软件系统,ADL将整个体系结构作为另一个较大系统的单个构件。也就是说,体系结构具有复合或等级复合的特性。另外,体系结构配置支持采用异构构件与连接件。这是因为软件体系结构的目的之一是促进大规模系统的开发,即倾向于使已有的构件与不同粒度的连接件进行连接和组装,这些构件与连接件的设计者、形式模型、开发者、编程语言、操作系统、通信协议可能都不相同。另外一个事实是,大型的、长期运行的系统是在不断增长的。因而,ADL必须支持可能增长的系统的说明与开发。大多数ADL提供了复合特性,所以,任意尺度的配置都可以相对简捷地在足够的抽象高度表示出来。

体系结构设计是整个软件生命周期中关键的一环,一般在需求分析之后,软件设计之前进行。而形式化的、规范化的体系结构描述对于体系结构的设计和理解都是非常重要的。因此,ADL如何能够承上启下是十分重要的问题,一方面是体系结构描述如何向其他文档转移,另一方面是如何利用需求分析成果直接生成系统的体系结构说明。

现有的ADL大多与领域相关,这不利于对不同领域体系结构的说明。针对不同领域的ADL在某些方面大同小异,造成了资源的冗余。有些ADL可以实现构件与连接件的演化,这样的演化大多是通过子类型实现的,但这样的演化能力是有限的,系统级的演化能力才是最终目的。尽管现有的ADL都提供了支持工具集,但将这些ADL与工具应用于实际系统开发中的成功范例有限;支持工具的可用性与有效性较差,严重地阻碍了这些ADL的广泛应用。

4.4 典型的软件体系结构描述语言

4.1节介绍了常见的软件体系结构描述方法,这些方法包括图形表达工具、模块内连接语言、基于软构件的系统描述语言、软件体系结构描述语言等。其中,ADL是在吸收了传统程序设计中的语义严格精确的特点基础上,针对软件体系结构的整体性和抽象性特点,定义和确定适合于软件体系结构表达与描述的有关抽象元素,因此,ADL是当前软件开发和设计方法学中一种发展很快的软件体系结构描述方法。

4.4.1 UniCon

作为一种体系结构描述语言，UniCon 的主要目的在于支持对体系结构的描述，对构件交互模式进行定位和编码，并且对需要不同交互模式的构件的打包加以区分。具体地说，UniCon 及其支持工具的主要目的如下。

(1) 提供对大量构件和连接件的统一的访问。

(2) 区分不同类型的构件和连接件，以便对体系结构配置进行检查。

(3) 支持不同的表示方式和不同开发人员的分析工具。

(4) 支持对现有构件的使用。

在 UniCon 中，通过定义类型、特性列表与用于和连接件相连的交互点来描述构件。连接件也是通过类型、特性列表和交互点来描述。其中，构件的交互点称为端口，连接件的交互点称为角色。系统组合构造通过定义构件的端口和连接件的角色之间的连接来完成。

为了达到目标(1)，UniCon 提供了一组预先定义的构件和连接件类型，体系结构的开发者可以从中选择合适的构件或连接件。对于(2)，UniCon 区分所有类型的构件和连接件的交互点，并对它们的组合方式进行限制。根据这些限制，UniCon 工具可以提供对组合失配进行检查，但是这种检查带有局部性，即无法对系统的全局约束进行检查。目标(3)的重要性已经获得公认，特性列表的方法已经被 ACME 和 USC 开发的 Architecture Capture Tool 采纳。对于已有构件，通过利用 UniCon 的术语对其接口重新定义，使得它们可以被 UniCon 使用。

以下程序是一个连接件约束的例子，使连接件角色接受指定的构件的端口。

```
ROLE output IS Source
    MAXCONNS (1)
    ACCEPT (Filter.StreamIn)
END onput
```

以下程序是 UniCon 描述管道的一个例子，在这个例子中，两个连接由构件和连接件实例分开。

```
USES p1 PROTOCOL Unix - pipe
USES sorter INTERFACE Sort - filter
CONNECT sorter.output TO p1.source
USES p2 PROTOCOL Unix - pipe
USES printer INTERFACE Print - filter
CONNECT sorter.input TO p2.sink
```

再来看一个完整的体系结构描述的例子。假设一个实时系统采用 C/S 体系结构，在该系统中，有两个任务共享同一个计算机资源，这种共享通过 RPC 实现。UniCon 对该系统结构的描述如下。

```
component Real_Time_System
   interface is
      type General
```

```
    implementation is
      uses client interface rtclient
        PRIORITY(10)
        ...
      end client
      uses server interface rtserver
        PRIORITY(10)
        ...
      end server
      establish RTM - realtime - sched with
        client.application1 as load
        server.application2 as load
        server.services as load
        ALGORITHM(rate_monotonic)
        ...
      end RTM - realtime - sched
      establish RTM - remote - proc - call with
        client.timeget as caller
        server.timeger as definer
        IDLTYPE(Mach)
      end RTM - remote - proc - call
      ...
    end implementation
end Real - Time - System
connector RTM - realtime - sched
    protocol is
      type RTScheduler
      role load is load
    end protocol
    implementation is builtin
    end implementation
end RTM - realtime - sched
...
```

4.4.2 Wright

Wright支持对构件之间交互的形式化和分析。连接件通过协议来定义,而协议刻画了与连接件相连的构件的行为。对连接件角色的描述表明了对参与交互的构件的“期望”以及实际的交互进行过程。构件通过其端口和行为来定义,表明了端口之间是如何通过构件的行为而具有相关性的。一旦构件和连接件的实例被声明,系统组合便可以通过构件的端口和连接件的角色之间的连接来完成。

Wright的主要特点为:对体系结构和抽象行为的精确描述、定义体系结构风格的能力和一组对体系结构描述进行一致性和完整性的检查。体系结构通过构件、连接件以及它们之间的组合来描述,抽象行为通过构件的行为和连接件的胶水来描述。

在Wright中,对体系结构风格的定义是通过描述能在该风格中使用的构件和连接件,以及刻画如何将它们组合成一个系统的一组约束来完成的。因此,Wright能够支持针对某一特定体系结构风格所进行的检查。但是,它不支持针对异构风格组成的系统的检查。

Wright 提供的一致性和完整性检查有：端口-行为一致性、连接件死锁、角色死锁、端口-角色相容性、风格约束满足、胶水完整性等。

以下程序是用 Wright 对管道连接件进行描述的例子。在这个例子中，定义了 Pipe 连接件，该连接件具有两个角色，分别为 Writer 和 Reader。其中，→表示事件变迁，√表示过程成功地终止，□表示确定性的选择。

```
connector Pipe =
    role Writer = write → Writer □ close → √
    role Reader =
        let ExitOnly = close → √
        in let DoRead = (read → Reader □ read - eof → ExitOnly)
        in DoRead □ ExitOnly
    glue = let ReadOnly = Reader.read → ReadOnly
                        □ Reader.read - eof → Reader.close → √
                        □ Reader.close → √
        in let WriteOnly = Writer.write → WriteOnly □ Writer.close → √
        in Writer.write → glue
            □ Reader.read → glue
            □ Writer.close → ReadOnly
            □ Reader.close → WriteOnly
```

4.4.3 C2

C2 和其提供的设计环境(Argo)支持采用基于时间的风格来描述用户界面系统，并支持使用可替换、可重用的构件开发 GUI 的体系结构。其工作的重点在于对构件的重用，以及对运行时体系结构的动态改变以使系统满足某些 GUI 体系结构方面的特性。

在 C2 中，连接件负责构件之间消息的传递，而构件维持状态、执行操作并通过两个名字分别为 top 和 bottom 的端口和其他的构件交换信息。每个接口包含一种可发送的消息和一组可接收的消息。构件之间的消息要么是请求其他构件执行某个操作的请求消息，要么是通知其他构件自身执行了某个操作或状态发生改变的通知消息。构件之间的消息交换不能直接进行，只能通过连接件来完成。每个构件接口最多只能和一个连接件相连，而连接件可以和任意数目的构件或连接件相连。请求消息只能向上层传送，通知消息只能向下层传送。

C2 要求通知消息的传递只对应于构件内部的操作，和接收消息的构件的需求无关。这种对通知消息的约束保证了底层独立性，即可以在包含不同的底层构件(如不同的窗口系统)的体系结构中重用 C2 构件。C2 对构件和连接件的实现语言、实现构件的线程控制、构件的部署、连接件使用的通信协议等都不加限制。

以下程序是 C2 对构件的描述例子。

```
Component : : =
    component component_name is
        interface component_message_interface
        parameters component_parameters
        methods component_methods
```

```
        [behavior component_behavior]
        [context component_context]
    end component_name;
```

以下程序是C2对构件接口的描述例子。

```
component_message_interface :: =
    top_domain_interface
    bottom_domain_interface

top_domain_interface :: =
    top_domain is
        out interface_requests
        in interface_notifications

bottom_domain_interface :: =
    bottom_domain is
        out interface_notifications
        in interface_requests

interface_requests :: =
    {request;} | null;

interface_notifications :: =
    {notification;} | null;

request :: =
    message_name(request_parameters)

request_parameters :: =
    [to component_name][parameter_list]

notification :: =
    message_name[parameter_list]
```

下面以会议安排系统为例,详细讨论C2风格的描述语言。会议安排系统的C2风格体系结构如图4-2所示。

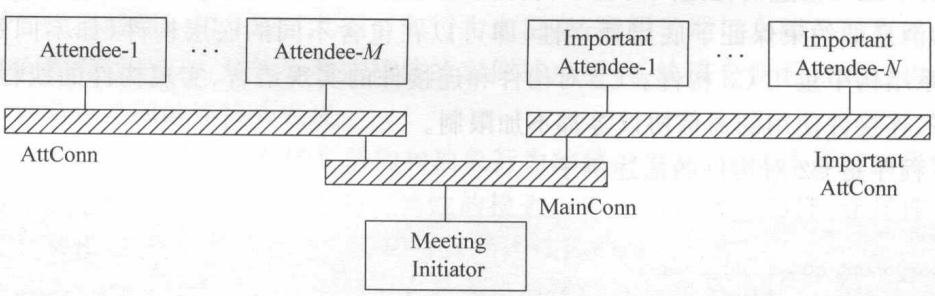


图4-2 会议安排系统的C2风格体系结构

系统包含3种功能构件，分别是一个MeetingInitiator，若干Attendee和ImportantAttendee，3个连接件(MainConn、AttConn和Impportant AttConn)用来在构件之间传递消息，某些消息可由MeetingInitiator同时发送给Attendee和ImportantAttendee，但某些消息只能传递给ImportantAttendee。因为一个C2构件在top、bottom端分别只有一个通信端口，且所有消息路由功能都与连接件相关，所以MainConn必须保证其top端的AttConn和ImportantAttConn分别只接收那些与它们相连的构件有关的消息。

MeetingInitiator构件通过发送会议请求信息给Attendee和ImportantAttendee来进行系统初始化。Attendee和ImportantAttendee构件可以发送消息给MeetingInitiator，告诉MeetingInitiator自己喜欢的会议日期、地点等信息；但不能向MeetingInitiator递交请求，因为在C2体系结构中，它们处在MeetingInitiator的top端。

MeetingInitiator构件的描述如下。

```
component MeetingInitiator is
  interface
    top_domain is
     out
        GetPrefSet();
        GetExclSet();
        GetEquipReqts();
        GetLocPrefs();
        RemoveExclSet();
        RequestWithdrawal(to Attendee);
        RequestWithdrawal(to ImportantAttendee);
        AddPrefDates();
        MarkMtg(d:date;l:lov_type);
     in
        PrefSet(p:date_mg);
        ExclSet(e:data_mg);
        EquipReqts(eq:equip_type);
        LocPref(l:loc_type);
   behavior
     startup always_generate GetPrefSet, GetExclSet, GetEquipReqts,
          GetLocPrefs;
     received_messages PrefSet may_generate RemoveExclSet xor
          RequestWithdrawal xor MarkMtg;
     received_messages  ExclSet  may_generate  AddPrefDates  xor RemoveExclSet
          xor RequestWithdrawal xor MarkMtg;
     received_messages EquipReqts may_generate AddPrefDates xor
          RemoveExclSet xor RequestWithdrawal xor MarkMtg;
     received_messages LocPref always_generate null;
end MeetingInitiator;
```

Attendee和ImportantAttendee构件接收来自MeetingInitiator构件的会议安排请求，把自己的有关信息发送给MeetingInitiator。这两种构件只能通过其bottom端与体系结构中的其他元素进行通信。Attendee构件的描述如下。

```
component Attendee is
  interface
    bottom_domain is
     out
        PrefSet(p:date_mg);
        ExclSet(e:date_mg);
        EquipReqts(eq:equip_type);
      in
        GetPrefSet();
        GetExclSet();
        GetEquipReqts();
        RemoveExclSet();
        RequestWithdrawal();
        AddPrefDates();
        MarkMtg(d:date;l:loc_type);
   behavior
      received_messages GetPrefSet always_generate PrefSet;
      received_messages AddPrefDates always_generate PrefSet;
      received_messages GetExclSet always_generate ExclSet;
      received_messages GetEqipReqts always_generate EqipReqts;
      received_messages RemoveExclSet always_generate ExclSet;
      received_messages ReuestWithdrawal always_generate null;
      received_messages MarkMtg always_generate null;
end Attendee;
```

ImportantAttendee构件是Attendee构件的一个特例,它具有Attendee构件的一切功能,并且还增加了自己特定的功能(可以指定会议地点),因此ImportantAttendee可以作为Attendee的一个子类型,保留了Attendee的接口和行为。ImportantAttendee构件的描述如下。

```
component ImportantAttendee is subtype Attendee(int and beh)
  interface
    bottom_domain is
     out
        LocPrefs(l:loc_type);
        ExclSet(e:date_mg);
        EquipReqts(eq:equip_type);
      in
        GetLocPrefs();
   behavior
      received_messages GetLocPrefs always_generate LocPrefs;
end ImportantAttendee;
```

有了上述3个构件的描述之后,可以得到体系结构的描述如下。

```
architecture MeetingScheduler is
   conceptual_components
       Attendee;ImportantAttendee;MeetingInitiator;
   connectors
```

```
        connector MainConn is message_filter no_filtering;
        connector AttConn is message_filter no_filtering;
        connector ImportantAttConn is message_filter no_filtering;
    architectural_topology
        connector AttConn connections
            top_ports Attendee;
            bottom_ports MainConn;
         connector ImportantAttConn connections
              top_ports ImportantAttendee;
              bottom_ports MainConn;
            connector MainConn connections
              top_ports AttConn;ImportantAttConn;
              bottom_ports MeetingInitiator;
end MeetingScheduler;
```

当实例化构件时，可以指定体系结构的一个实例。例如，当有 3 个普通与会人员、两个重要与会人员时，会议安排应用对应的一个实例的描述如下。

```
system MeetingScheduler_1 is
  architecture MeetingScheduler with
    Attendee instance Att_1,Att_2,Att_3;
    ImportantAttendee instance ImpAtt_1,ImpAtt_2;
    MeetingInitiator instance MtgInit_1;
end MeetingScheduler_1;
```

4.4.4 Rapide

Rapide 是一种可执行的 ADL，其目的在于通过定义并模拟基于事件的行为对分布式并发系统建模。它通过事件的偏序(partial order)集合来刻画系统的行为。构件计算由构件接收到的事件触发，并进一步产生事件传送到其他构件，由此触发其他计算。Rapide 模型的执行结果为一个事件的集合，其中的事件满足一定的因果或时序关系。Rapide 由 5 种子语言构成。

(1) 类型语言：定义接口类型和函数类型，支持通过继承已有的接口来构造新的接口类型。

(2) 模式语言：定义具有因果、独立、时序等关系的事件所构成的事件模式。

(3) 可执行语言：包含描述构件行为的控制结构。

(4) 体系结构语言：通过定义同步和通信连接来描述构件之间的事件流。

(5) 约束语言：定义构件行为和体系结构所满足的形式化约束，其中约束为需要的或禁止的偏序集模式。

Rapide 的优点在于能够提供多种分析工具。它所支持的分析都基于检测在某个模拟过程中的事件是否违反了某种次序关系。Rapide 允许仅基于接口而定义体系结构，把利用具有特定行为的构件对接口的替换留到开发的下一个阶段。因此，开发者可以在某个体系结构中使用尚未存在的构件，只要该构件符合特定的接口即可。

以下程序是 Rapide 描述构件的一个例子。

```
type Application is interface
   in action Request(p: params);
   out action Results(p: params);
behavior
   (?M in String) Receive(?M) => Results(?M);
end Application;
```

4.4.5 SADL

SADL的工作基于如下假设：大型软件系统的体系结构是通过一组相关的具有层次化的体系结构来描述的，其中高层是低层的抽象，它们具有不同数量和种类的构件和连接件。如果这种层次化和结构缺少正规的描述方法，则求精过程就容易出错。

在SADL中，提出了能够保证正确性和具有可组合性的体系结构求精模式的概念，使用求精模式的实例能够保证每一步求精过程的正确性。采用该方法能够有效地减少体系结构设计的错误，并且能够广泛地、系统地实现对设计领域的知识和正确性证明的重用。

为了证明两个体系结构在求精意义上的正确性，必须建立它们之间的解释映射。解释映射包含一个名字映射和一个风格映射。名字映射建立抽象体系结构中的对象的名字和具体体系结构中的对象的名字之间的关系。风格映射描述抽象层次中风格的构造是如何使用具体层次风格的构造中的术语来实现的。风格映射将会比较复杂，但是它只需要定义并证明一次，便能够在任何需要的时候被安全地重用。

以下程序是SADL描述连接件接口的例子。其中，前半部分定义和约束一个连接件，后半部分对数据流风格的连接件类型进行指定，所有Dataflow_Channel类型的连接件都支持两个构件之间的接口。

```
CONNECTORS
   channel : Dataflow_Channel < SEQ(token)>
CONFIGURATION
   Token_flow : CONNECTION = Connects(channel, oport, iport)

Dataflow_Channel : TYPE <= CONNECTOR
Connects : Predicate(3)
Connects_argtype_1 : CONSTRAINT =
   (/\ x)(/\ y)(/\ z)[Connects(x, y, x) => Dataflow_Channel(x)]
Connects_argtype_2 : CONSTRAINT =
   (/\ x)(/\ y)(/\ z)[Connects(x, y, x) => Outport(y)]
Connects_argtype_1 : CONSTRAINT =
   (/\ x)(/\ y)(/\ z)[Connects(x, y, x) => Inport(z)]
```

4.4.6 Aesop

Aesop是由美国卡耐基-梅隆大学的Garlan等人创建的一门体系结构描述语言，其目的是建立一个工具包，为领域特定的体系结构快速构建软件体系结构设计环境。每个环境都支持以下5个方面。

(1) 与风格词汇表相对应的一系列设计元素类型，即特定风格的构件和连接件。

(2) 检查设计元素的成分，满足风格的配置约束。

(3) 优化设计元素的语义描述。

(4) 一个允许外部工具进行分析和操作体系结构描述的接口。

(5) 多个风格特定的体系结构的可视化,以及操作它们的图形编辑工具。

Aesop 采用了产生式方法,该方法构建在已有的软件开发环境技术上。Aesop 把一个/组风格描述与一个普遍使用的、共享的工具包联系在一起,该工具包称作 Fable。

以下程序是 Aesop 描述一个管道-过滤器体系结构的例子。

```
//Generates code for a pipe - filter system
int main(int argc,char ** argv)
{
   fable_init_event_system(&argc,argv,BUILD_PF);  //init local event system
   fam_initialize(argc,argv);                     //init the database

   arch_db = fam_arch_db::fetch();                //get the top - level DB pointer
   t = arch_db.open(READ_TRANSACTION);            //start read transaction on it

   fam_object o = get_object_parameter(argc,argv);//get root object

   if (!o.valid() || !o.type().is_type(pf_filter_type))
   {                                              //not valid filter
      cerr << argv[0] << ":invalid parameter\n";  //stop now
      t.close();
      fam_terminate();
      exit(1);
   }

   pf_filter root = pf_filter::typed(o);
   pf_aggregate ag = find_pf_aggregate(root);     //find root's aggregate

   start_main();                                  //write standard start of generated main()
   outer_io(root);                                //bind outer ports to stdin/out

   if (ag.valid())
   {
      pipe_name(ag);                              //write code to connect up pipes
      bindings(root);                             //and to alias the bindings
      spawn_filters(ag);                          //and to fork off the filters
   }

   finish_main();                                 //write standard end of generated main()

   make_filter_header(num_pipes);                 // write header file for pipe names

   t.close();                                     //close transaction
   fam_terminate();                               //terminate fam
   fable_main_event_loop();                       //wait for termination event
   fable_finish();                                //and finish
   return 0;
}
```

4.4.7 ACME

ACME也是由美国卡耐基梅隆大学的Garlan等人创建的一门体系结构描述语言,其最初目的是创建一门简单的、具有一般性的ADL,该ADL能用来为体系结构设计工具转换形式和/或为开发新的设计和分析工具提供基础。

因此,严格说来,ACME并不是一种真正意义上的ADL,而是一种体系结构变换语言,它提供了一种在不同ADL的体系结构规范描述之间实现变换的机制。因此,虽然ACME没有支持工具,但却能够使用各种ADL提供的支持工具。

ACME提供了描述体系结构的结构特性的方法,此外还提供了一种开放式的语义框架,使得可以在结构特性上标注一些ADL相关的属性。这种方法使得ACME既能表示大多数ADL都能描述的公共的结构信息,又能使用注解来表示与特定的ADL相关的信息。有了这种公共又灵活的表示方法,再加上ADL之间关于属性的语义转换工具,就能顺利地实现ADL之间的变换,从而使ADL之间能够实现分析方法和工具的共享。

ACME支持从4个不同的方面对软件体系结构进行描述,分别是结构、属性、设计约束、类型和风格。

1. 结构

在ACME中定义了7种体系结构实体,分别是构件、连接件、系统、端口、角色、表述(representations)和表述映射(map)。其中,前5种如图4-3所示。

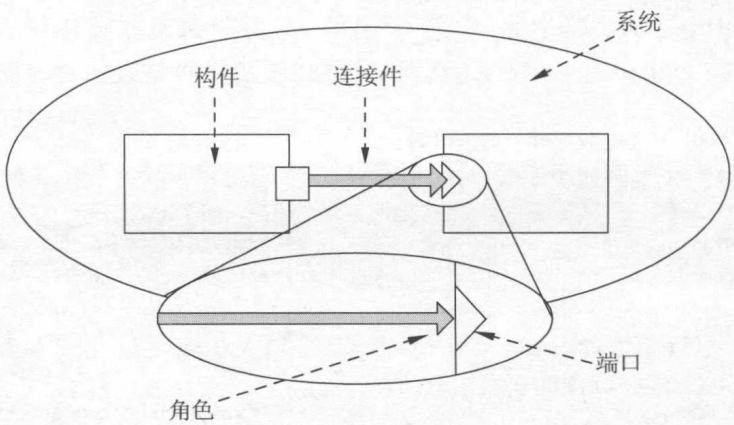


图4-3 ACME描述的元素

以下程序是一个以ACME描述的一个简单的C/S体系结构的例子。其中,client构件只有一个sendRequest端口,server也只有一个receive_Request端口,连接件rpc有两个角色,分别为caller和callee。该系统的布局(topology)由与构件端口和连接件角色绑定的attachments定义。其中,client的请求端口绑定到rpc的caller角色,server的请求处理端口绑定到rpc的callee端口。

```
System simple_CS = {
    Component client = { Port sendRequest }
    Component Server = { Port receiveRequest }
    Connector rpc = { Roles {caller, callee}}
```

```
    Attachments : {
      client.sendRequest to rpc.caller;
      server.receiveRequest to rpc.callee }
  }
```

为了支持体系结构的分级描述，ACME 允许任何一个构件或连接件由一个或多个更详细的、低级的描述代表，称每个这样的代表为一个表述(representation)。图 4-4 是一个构件表述的例子。

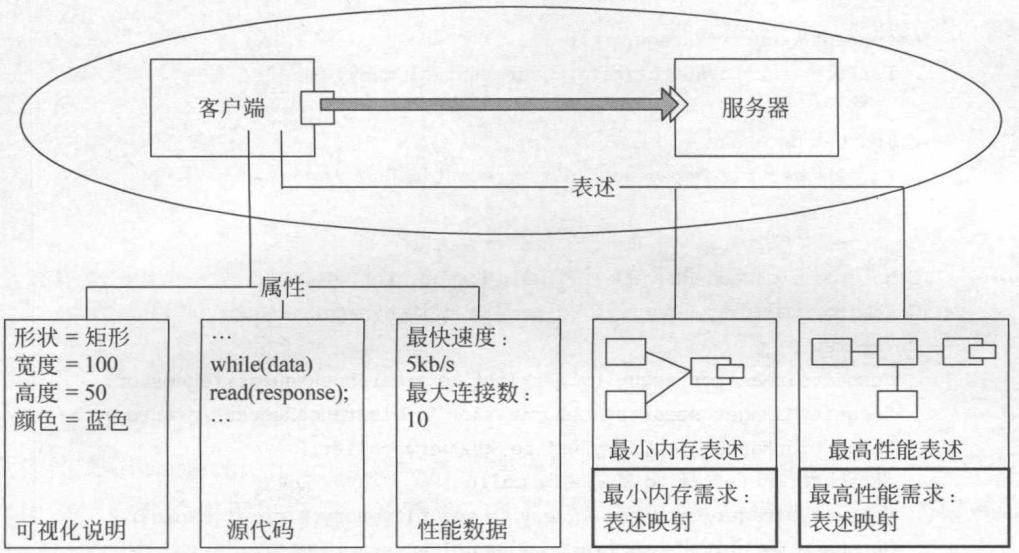


图 4-4　构件的表述和属性

当构件或连接件有一个体系结构表述时，必须由某些方式表明在内部系统表述和所表述的构件或连接件的外部接口之间的通信。表述映射就定义了这种通信。在最简单的情况下，表述映射提供了在内部端口和外部端口(对连接件而言，是内部角色和外部角色)之间的关联。

图 4-5 说明了细化后的 C/S 结构的表述的使用。在图 4-5 中，服务器构件由一个更详细的体系结构表述所细化，其 ACME 描述如以下程序所示。

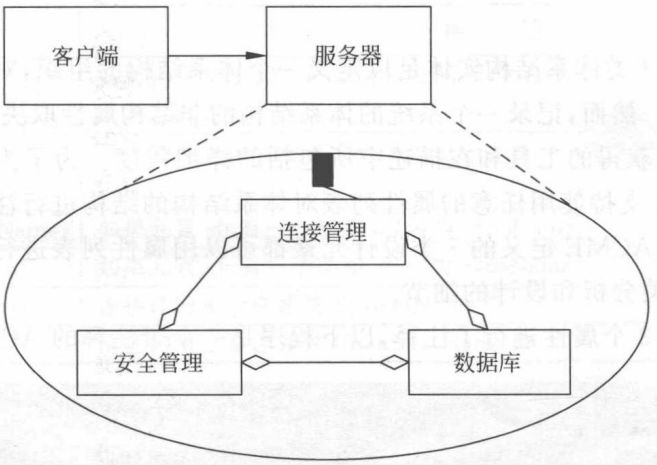


图 4-5　带有表述的 C/S 结构

```
System simple_CS = {
  Component client = { … }
  Component Server = {
    Port receiveRequest;
    Representation serverDetails = {
      System serverDetailsSys = {

        Component connectionManager = {
          Ports { exernalSocket; securityCheckIntf;dbQueryIntf; }}

        Component securityManager = {
          Ports { securityAuthorization; credentialQuery; }}

        Component database = {
          Ports { securityManagementIntf; credentialQuery; }}

        Connector SQLQuery = { Roles { caller; callee }}
        Connector clearanceRequest = { Role { requestor; grantor }}
        Connector securityQuery = { Roles { securityManager; requestor }}
        Attachments {
          ConnectoinManager.securityCheckIntf to clearanceRequest.requestor;
          SecutityManager.securityAuthorization to clearanceRequest.grantor;
          ConnectionManager.dbQueryIntf to SQLQuery.caller;
          database.queryIntf to SQLQuery.callee;
          securityManager.credentialQuery to securityQuery.securityManager;
          database.securityManagerIntf to securityQuery.requestor; }

    }
    Bindings { connectionManager.externalSocket to server.receiveRequest }
  }
}
Connector rpc = { … }
Attachments { client.sendRequest to rpc.caller;
              server.receiveRequest to rpc.callee }
```

2. 属性

ACME定义的7类体系结构实体足以定义一个体系结构的组织,对结构的清晰描述有利于系统的文档化。然而,记录一个系统的体系结构的非结构属性取决于系统的特性、所需要的分析种类、可以获得的工具和在描述中所包括的详细程度。为了为说明辅助信息提供开放性需求,ACME支持使用任意的属性列表对体系结构的结构进行注释。每个属性有名称、可选类型和值。ACME定义的7类设计元素都可以用属性列表进行注释。属性可用来记录与体系结构相关分析和设计的细节。

在图4-4中,对3个属性进行了注释,以下程序是一个带注释的ACME描述例子。

```
System simple_CS = {
  Component client = {
    Port sendRequest;
    Properties { requestRate : float = 17.0;
```

```
                sourceCode : externalFile = "CODE-LIB/client.c"}}

Component Server = {
  Port receiveRequest;
  Properties { idempotent : boolean = true;
               maxConcurrentClients : integer = 1;
               multithreaded : Boolean = false;
               sourceCode : externalFile = "ODE-LIB/server.c" }}

Connector rpc = {
  Roles caller;
  Roles callee;
  Properties { synchronous : boolean = true;
               maxRoles : integer = 2;
               protocol : WrightSpec = "…" }}

Attachments : {
  client.sendRequest to rpc.caller;
  server.receiveRequest to rpc.callee }
}
```

3. 设计约束

设计约束(design constraints)是体系结构描述的关键成分,它们决定体系结构设计是如何演化的。设计约束可以当作一种特殊的属性,但因为在体系结构中,设计约束起着核心作用,所以,ACME 提供了特定的语法来描述设计约束。

ACME 使用基于一阶谓词逻辑(First Order Predicate Logic,FOPL)的约束语言来描述设计约束,在体系结构规格说明中,约束被当作谓词。约束语言包括标准的 FOPL 集合,同时还包含一些体系结构方面的特殊函数。例如,有决定两个构件是否有连接(包括直接连接和间接连接)的谓词,有决定一个构件是否有特殊的属性的谓词。表 4-3 是设计约束描述的例子。

表 4-3 ACME 对带有属性的 C/S 结构的描述

结　构	描　述
Connected(comp1,comp2)	如果构件 comp1 与 comp2 之间至少有一个连接件,则取 True,否则取 False
Reachable(comp1,comp2)	如果构件 comp2 处在 Connected(comp1,*)的路径上,则取 True,否则取 False
HasProperty(elt,propName)	如果元素 elt 有一个属性,取名为 propName
HasType(elt,typeName)	如果元素 elt 有一个类型,取名为 typeName
SystemName.Connectors	连接件的集合在系统 SystemName 中
ConnectorName.Roles	角色的集合在连接件 ConnectorName 中

约束可以与 ACME 描述中的任何设计元素相关联,约束的范围取决于关联。例如,如果一个约束与一个系统相关联,则它可以引用包含在该系统中的任何设计元素(构件、连接件的全部或部分)。另外,与一个构件相关联的约束则只能引用该构件及其部分(端口、属性

和表述)。例如,下面语句的约束与一个系统相关联:

```
connected(client,server)
```


如果名字为 client 的构件与名字为 server 的构件直接由一个连接件连接,则该约束取 True。

又如,约束

```
forall conn : connector in systemInstance.Connectors @ size(conn.roles) = 2
```

当系统中的所有连接件都是二元连接件时取 True。

约束可以定义合法属性值的范围,例如:

```
self.throughputRate >= 3095
```

约束也可以表明两个属性之间的关系,例如:

```
comp.totalLatency = (comp.readLatency + comp.processingLatency + comp.writeLatency)
```

约束可以通过两种方式附加到设计元素,分别是 invariant 和 heuristic,其中 invariant 约束当作不可违反的规则看待,heuristic 约束当作应该遵守的规则看待。对 invariant 约束的违反会使体系结构规格说明无效,而对 heuristic 约束的违反会当作一个警告处理。以下程序说明了如何在 ACME 中使用约束。其中,invariant 约束描述合法缓冲的大小范围,heuristic 约束描述了期望的速度的最大值。

```
System messagePathSystem = {
    ...
    Connector MessagePath = {
        Roles {source; sink; }
        Property expectedThroughput : float = 512;
        Invariant ( queueBufferSize >= 512) and (queueBufferSize <= 4096);
        Heuristic expectedThroughput <= (queueBufferSize / 2);
    }
}
```

4. 类型和风格

体系结构描述的一个重要能力是能够定义系统的风格或族,风格允许设计师定义领域特定的或应用特定(application-specific)的设计词汇(vocabulary),以及如何使用这些词汇的约束。支持对领域特定的设计经验的打包(package)、特定目的的分析和代码生成工具的使用、设计过程的简化、与体系结构标准一致性的检查等。

在 ACME 中,定义风格的基本构造块是一个类型系统,类型系统可用来封装循环结构和关系。设计师可定义 3 种类型,分别是属性类型、结构类型和风格。

1) 属性类型

在第 3 点,已经讨论过属性类型,在此不再赘述。

2) 结构类型

结构类型使定义构件、连接件、端口和角色的类型变得可能,每一个这样的类型提供了一个类型名称和一个所需要的子结构、属性和约束的列表。以下程序是一个 client 构件类

型的描述例子。类型定义指明了任何构件如果是类型 client 的任何一个实例，则必须至少有一个端口称作 Request 和一个浮点型的属性称作 request-rate。与类型关联的 invariant 约束要求 client 构件的所有端口都有一个取值为 rpc-client 的协议属性，每个 client 都不得超过 5 个端口，构件的请求个数必须大于 0。最后，还有一个 heuristic 约束，要求请求的个数小于 100。

```
Component Type Client = {
    Port Request = { Property protocol : CSPprotocolT };
    Property request - rate : Float;
    Invariant size(self.Ports) <= 5;
    Invariant request - rate >= 0;
    Heuristic request - rate < 100;
```

3) 风格

在 ACME 中，风格被称为族(family)，就好像结构类型代表一组结构元素一样，族代表一组系统。在 ACME 中，可通过指定 3 件事情来定义一个族，分别是一组属性和结构元素，一组约束，以及默认结构。属性类型和结构类型为族提供了设计词汇，约束决定了如何使用这些类型的实例。默认结构描述了必须出现在族中任何系统中的实例的最小集合。以下程序是一个定义管道-过滤器族和使用该族的一个系统的实例。在这个族中，定义了两个构件类型，一个连接件类型和一个属性类型。该族唯一的 invariant 约束指定了所有连接件必须使用管道，没有默认的结构。系统 simplePF 作为族的一个实例来定义，这种定义允许系统使用族的任何类型，且必须满足族的所有 invariant 约束。

```
Family PipeFilterFam = {

    Component Tyrp FilterT = {
        Ports { stdin; stdout; };
        Property throughput : int;
    };
    component Type UnixFilterT extends FilterT with {
        Port stderr;
        Property implementatationFile : String;
    };
    connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
    };
    Property Type StringMsgFormatT = Record [ size : int; msg : String; ];
    Invariant Forall c in self.Connectors @ HasType(c, PipeT);
}

System simplePF :PipeFilterFam = {

        Component smooth : FilterT = new FilterT;
        Component detectErrors : FilterT;
        Component showTracks : UnixFilterT = new UnixFilterT extended with {
```

```
            Property implementationFile : String = "IMPL_HOME/showTracks.c";
};

//Declare the system's connectors
connector firstPipe : PipeT;
Connector secondPipe : PipeT;

//Define the system's topology
attachments { smooth.stdout to firstPipe.source;
              detectErrors.stdin to firstPipe.sink;
              detectErrors.stdout to secondPipe.source;
              showTracks.stdin to secondPipe.sink; }
}
```

从以上介绍中可以看出,这些 ADL 的共同目的都是以构件和连接件的方式描述软件体系结构,不同的只是底层的语法和语义。目前的 ADL 基本上都满足必需的语言标准,尤其是重用的重要性。为了更好地使用 ADL,通常需要一个配套的开发环境。这类环境通常提供以下工具:创建和浏览设计的图形化编辑器、体系结构一致性检查、代码生成器、模式仓储等。

思 考 题

1. 软件体系结构描述方法和体系结构描述标准是形式化的基础,简要说明有哪些软件体系结构描述方法和描述标准。

2. 典型的软件体系结构描述语言包括哪些?它们分别需要描述体系结构中的哪些基本元素?

3. 任选本章介绍的 3 种 ADL,对同一个简单系统的体系结构或其中一个部分进行描述,并对你所做出的 ADL 描述结果加以对比。

4. 对一个你曾经开发过的软件系统进行考虑,如果要使用 ADL 对其体系结构进行描述,你会选择哪一种?为什么?这样选择的优点和缺点各有哪些?

参考文献

第5章 统一建模语言

在20世纪80年代到20世纪90年代，面向对象的分析与设计方法获得了长足的发展，而且相关的研究也十分活跃，涌现出了一大批新的方法学。其中，最著名的是Booch的Booch 1993、Jacobson的OOSE(Object-Oriented Software Engineering，面向对象软件工程)和Rumbaugh的OMT(Object Modeling Technique，对象建模技术)方法。UML正是在融合了Booch、Rumbaugh和Jacobson方法论的基础上形成的标准建模语言。

UML并不是本书的直接研究对象，而只是一种建模的工具。为了保持知识的连续性，本章首先介绍UML的一些基础知识，然后重点研究如何直接使用UML对软件体系结构建模，以及如何使用UML的扩展机制对软件体系结构建模。

5.1 UML概述

UML是用于系统的可视化建模语言，尽管它常与建模OO软件系统相关联，但由于其内建了大量扩展机制，还可以应用于更多的领域中，如工作流程、业务领域等。

(1) UML是一种语言：UML在软件领域中的地位与价值就像“1、2、3、+、−、…”等符号在数学领域中的地位一样。它为软件开发人员之间提供了一种用于交流的词汇表，一种用于软件蓝图的标准语言。

(2) UML是一种可视化语言：UML只是一组图形符号，它的每个符号都有明确语义，是一种直观、可视化的语言。

(3) UML是一种可用于详细描述的语言：UML所建的模型是精确的、无歧义和完整的，因此，适合于对所有重要的分析、设计和实现决策进行详细描述。

(4) UML是一种构造语言：UML虽然不是一种可视化的编程语言，但其与各种编程语言直接相连，而且有较好的映射关系，这种映射允许进行正向工程、逆向工程。

(5) UML是一种文档化语言：它适合于建立系统体系结构及其所有的细节文档。

5.1.1 UML的发展历史

面向对象的建模语言最早出现在20世纪70年代中期，在20世纪80年代末进入快速发展阶段，截至1994年，从不到10种发展到50多种。1994年之后，各种方法论逐渐拉开了差距，以Booch提出的Booch方法和Rumbaugh提出的OMT成为可视化建模语言的主导，而Jacobson的Objectory方法则成为最强有力的方法。

Booch是面向对象方法最早的倡导者之一。他在1984年出版的*Software Engineering with Ada*一书中就描述了面向对象软件开发的基本问题。1991年，他在*Object-Oriented*

Design 一书中,将以前针对 Ada 的工作扩展到整个面向对象设计领域,他对继承和类的阐述特别值得借鉴。Booch1993 比较适合于系统的设计和构造。

Runbaugh 等提出了 OMT,采用了面向对象的概念并引入各种独立于程序设计语言的表示符号。这种方法用对象模型、动态模型、功能模型和用例模型共同完成对整个系统的建模,所定义的概念和符号可用于软件开发的分析、设计和实现的全过程,软件开发人员不必在开发过程的不同阶段进行概念和符号的转换。OMT-2 特别适用于分析和描述以数据为中心的信息系统。

Jacobson 于 1994 年提出了 OOSE 方法,其最大特点是面向用例,在用例的描述中引入了外部角色的概念。用例的概念贯穿于整个开发过程(包括对系统的测试和验证),是精确描述需求的重要武器。目前,在学术界和工业界已普遍接受用例的概念,并认为是面向对象技术走向第二代的标志。OOSE 比较合适支持业务工程和需求分析。

各种建模语言各有长短,面对众多的建模语言,用户无力区分不同建模语言之间的差别和使用范围。由于不同的用户使用不同的建模语言和不同建模语言表达方式上的差异,使得用户之间的沟通方面出现了困难。要解决以上问题,就必须在综合分析各种不同建模语言的优缺点、适用情况的基础上统一各种不同的建模语言。

1994 年 10 月,Booch 和 Rumbaugh 开始了这项工作。首先,他们将 Booch93 和 OMT-2 统一起来,并于 1995 年 10 月发布了第一个公开版本,称之为 UM 0.8(Unified Method,标准方法)。同年,Jacobson 加盟到这项工作中,经过 Booch、Rumbaugh 和 Jacobson 的共同努力,于 1996 年 6 月和 10 月分别发布了两个新的版本(UML 0.9 和 UML 0.91),并将 UM 重新命名为 UML。

1997 年,OMG 采纳了 UML,一个开放的 OO 可视化建模语言工业标准诞生了。现在 UML 已经经历了 1.1、1.2 和 1.4 共 3 个版权的演变,2003 年发布了 2.0 版标准。

5.1.2 UML 的应用领域

UML 统一了 Booch、OMT、OOSE 和其他面向对象方法的基本概念和符号,同时汇集了面向对象领域中很多人的思想,是优秀的面向对象方法和丰富的计算机科学实践中总结而成的。

目前,UML 是最先进、最实用的标准建模语言,而且还在不断发展演化之中。UML 是一种建模语言而不是一种方法,其中并不包括过程的概念,其本身是独立于过程的,可以在任何过程中使用它。不过与 UML 结合最好的是用例驱动的、以体系结构为中心的、迭代的、增量的开发过程。

作为一种标准建模语言,UML 的核心是以面向对象的思想描述客观世界,具有广阔的应用领域。目前,主要应用领域是在软件系统建模,但是它同样可以应用于其他领域,如机械系统、企业机构或业务过程,以及处理复杂数据的信息系统、具有实时要求的工业系统或工业过程等。总之,UML 是一个通用的标准建模语言,可以对任何系统的动态行为和静态行为进行建模。

UML 可以为信息系统从需求分析到系统维护的整个生命周期提供有效的支持。在需求分析阶段,可以通过用例模型来捕获和组织用户的需求,分析系统对于用户的价值。通过用例建模,描述对系统感兴趣的外部角色及其对系统(用例)的功能要求。分析阶段主要关

心问题域中的主要概念(如抽象、类和对象等)和机制,以及这些概念之间的相互协作,并用UML类图来描述。至于类之间的协作关系则可以用交互图和顺序图来描述。在分析阶段,只对问题域的对象(现实世界的概念)建模,而不考虑定义软件系统中技术细节的类(如处理用户接口、数据库、通信、并行性等问题的类)。这些技术细节将在设计阶段引入,因此设计阶段为构造阶段提供更详细的规格说明。

编码阶段的主要任务是将设计阶段的设计结果转换成为实际的代码。在设计阶段需要注意的是,不要过早地考虑设计结果要用哪种编程语言实现。如果过早地考虑这个问题,会使设计工作陷入细节的泥潭,不利于对模型进行全面理解。

UML还可以对测试阶段提供有效的支持。不同的测试阶段可以使用不同的UML图作为测试的依据。例如,在单元测试阶段,可以使用类图和类的规格说明来指导测试;在集成测试阶段,可以使用通信图、活动图和部署图;系统测试和验收测试阶段则可以使用顺序图和用例图来验证系统的外部行为。

总之,UML能够用面向对象的方法描述任何类型的系统,并对系统开发从需求调研到测试和维护的各个阶段进行有效的支持。

5.2 UML的结构

UML的结构包括UML的基本构造块、支配这些构造块如何放在一起的规则(体系结构)和一些运用于整个UML的机制。

5.2.1 结构概述

本节将详细介绍UML结构的各组成元素。

1. 构造块

UML有3种基本的构造块,分别是事物(thing)、关系(relationship)和图(diagram)。事物是UML中重要的组成部分,关系把事物紧密联系在一起,图是很多有相互关系的事物的组。

2. 公共机制

公共机制是指达到特定目标的公共UML方法,主要包括规格说明(详细说明)、修饰、公共分类(通用划分)和扩展机制4种。

(1) 规格说明:规格说明是元素语义的文本描述,它是模型真正的核心。

(2) 修饰:UML为每一个事物设置了一个简单的记号,还可以通过修饰表达更多的信息。

(3) 公共分类:包括类元与实体(类元表示概念,实体表示具体的实体)、接口和实现(接口用来定义契约,而实现就是具体的内容)两组公共分类。

(4) 扩展机制:包括约束(添加新规则来扩展事物的语义)、构造型(用于定义新的事物)、标记值(添加新的特殊信息来扩展事物的规格说明)。

3. 规则

UML用于描述事物的语义规则分别是为事物、关系和图命名。给一个名字以特定含义的语境,即范围;怎样使用或看见名字,即可见性;事物如何正确、一致地相互联系,即完

整性；运行或模拟动态模型的含义是什么，即执行。

UML对系统体系结构的定义是系统的组织结构，包括系统分解的组成部分、它们的关联性、交互、机制和指导原则，这些提供系统设计的信息。而具体来说，就是指5个系统视图，分别是逻辑视图、进程视图、实现视图、部署视图和用例视图。

(1) 逻辑视图：以问题域的词汇组成的类和对象集合。

(2) 进程视图：可执行线程和进程作为活动类的建模，它是逻辑视图的一次执行实例，描述了所设计的并发与同步结构。

(3) 实现视图：对组成基于系统的物理代码的文件和构件进行建模。

(4) 部署视图：把构件部署到一组物理的、可计算的结点上，表示软件到硬件的映射及分布结构。

(5) 用例视图：最基本的需求分析模型。

提示：UML还允许在一定的阶段隐藏模型的某些元素、遗漏某些元素以及不保证模型的完整性，但模型逐步地要达到完整和一致。

5.2.2 事物

UML中的事物也称建模元素，包括结构事物(structural things)、行为事物(behavioral things，动作事物)、分组事物(grouping things)和注释事物(annotational things，注解事物)。这些事物是UML模型中最基本的面向对象的构造块。

1. 结构事物

结构事物在模型中属于最静态的部分，代表概念上或物理上的元素。总共有7种结构事物，具体如下。

(1) 类。类是描述具有相同属性、方法、关系和语义的对象的集合。一个类实现一个或多个接口。

(2) 接口。接口是指类或构件提供特定服务的一组操作的集合。因此，一个接口描述了类或构件的对外的可见的动作。一个接口可以实现类或构件的全部动作，也可以只实现一部分。

(3) 协作。协作定义了交互的操作，是一些角色和其他元素一起工作，提供一些合作的动作，这些动作比元素的总和要大。因此，协作具有结构化、动作化、维的特性。一个给定的类可能是几个协作的组成部分。这些协作代表构成系统的模式的实现。

(4) 用例。用例是描述一系列的动作，这些动作是系统对一个特定角色执行，产生值得注意的结果的值。在模型中用例通常用来组织行为事物。用例是通过协作实现的。

(5) 活动类。活动类的对象有一个或多个进程或线程。活动类和类很相似，只是它的对象代表的元素的行为和其他的元素是同时存在的。

(6) 构件。构件是物理上或可替换的系统部分，它实现了一个接口集合。在一个系统中，可能会遇到不同种类的构件。

(7) 结点。结点是一个物理元素，它在运行时存在，代表一个可计算的资源，通常占用一些内存和具有处理能力。一个构件集合一般来说位于一个结点，但有可能从一个结点转到另一个结点。

2. 行为事物

行为事物是 UML 模型中的动态部分。它们是模型的动词，代表时间和空间上的动作。总共有两种主要的行为事物，具体如下所述。

(1) 交互(内部活动)。交互是由一组对象之间在特定上下文中，为达到特定的目的而进行的一系列消息交换而组成的动作。交互中组成动作的对象的每个操作都要详细列出，包括消息、动作次序(消息产生的动作)、连接(对象之间的连接)等。

(2) 状态机。状态机由一系列对象的状态组成。

交互和状态机是 UML 模型中最基本的两个动态事物，它们通常和其他的结构事物、主要的类、对象连接在一起。

3. 分组事物

分组事物是 UML 模型中组织的部分，可以把它们看成一个盒子，模型可以在其中被分解。UML 只有一种分组事物，称为包。包是一种将有组织的元素分组的机制。结构事物、行为事物甚至其他的分组事物都有可能放在一个包中。与构件(存在于运行时)不同的是包纯粹是一种概念上的东西，只存在于开发阶段。

4. 注释事物

注释事物是 UML 模型的解释部分。

5.2.3 关系

UML 用关系把事物结合在一起，UML 中的关系主要有以下 4 种。

(1) 依赖(dependencies)：两个事物之间的语义关系，其中一个事物发生变化会影响另一个事物的语义。

(2) 关联(association)：描述一组对象之间连接的结构关系，例如，聚合关系描述了整体和部分间的结构关系。

(3) 泛化(generalization)：一般化和特殊化的关系，描述特殊元素的对象可替换一般元素的对象。

(4) 实现(realization)：类之间的语义关系，其中的一个类指定了由另一个类保证执行的契约。

1. 用例之间的关系

两个用例之间的关系可以概括为两种情况。一种是用于重用的包含关系，用构造型<< include >>表示；另一种是用于分离出不同行为的扩展关系，用构造型<< extend >>表示。

(1) 包含关系：当可以从两个或两个以上的原始用例中提取公共行为，或者发现能够使用一个构件来实现某一个用例很重要的部分功能时，应该使用包含关系来表示它们。提取出来的公共用例称为抽象用例。

例如，希赛图书订单处理系统中，“创建新订单”和“更新订单”两个用例都需要检查客户的账号是否正确，为此定义一个抽象用例“核查客户账户”。用例“创建新订单”和“更新订单”与用例“核查客户账户”之间的关系就是包含关系。

(2) 扩展关系：如果一个用例明显地混合了两种或两种以上的不同场景，即根据情况可能发生多种事情，则可以断定将这个用例分为一个主用例和一个或多个辅用例进行描述可能更加清晰。

例如,希赛图书管理系统中,读者归还图书时,需要判断当前日期是否已经超过了图书借阅的周期。如果超过了借阅周期,则必须罚款。“归还图书”和“罚款”用例之间的关系就是扩展关系。

用例之间还存在一种泛化关系。用例可以被特别列举为一个或多个子用例,这被称作用例泛化。当父用例能够被使用时,任何子用例也可以被使用。例如,购买飞机票时,既可以通过电话订票,也可以通过网上订票,则订票用例就是电话订票和网上订票的泛化。

2. 类之间的关系

在建立抽象模型时,很少有类会单独存在,大多数都将会以某种方式彼此通信。因此,还需要描述这些类之间的关系。

(1) 关联关系。描述了给定类的单独对象之间语义上的连接。关联提供了不同类之间的对象可以相互作用的连接。其余的关系涉及类元自身的描述,而不是它们的实例。关联关系用———表示。

(2) 依赖关系。有两个元素X、Y,如果修改元素X的定义可能会引起对另一个元素Y的定义的修改,则称元素Y依赖于元素X。在UML中,使用带箭头的虚线------▸表示依赖关系。

在类中,依赖由各种原因引起。例如,一个类向另一个类发送消息;一个类是另一个类的数据成员;一个类是另一个类的某个操作参数。如果一个类的接口改变,则它发出的任何消息都可能不再合法。

(3) 泛化关系。泛化关系描述了一般事物与该事物中的特殊种类之间的关系,即父类与子类之间的关系。继承关系是泛化关系的反关系,即子类是从父类继承的,而父类则是子类的泛化。在UML中,使用带空心箭头的实线———▷表示泛化关系,箭头指向父类。

(4) 聚合关系。聚合是一种特殊形式的关联,它是传递和反对称的。聚合表示类之间的关系是整体与部分的关系。例如,一辆轿车包含4个车轮、一个方向盘、一个发动机和一个底盘,就是聚合的一个例子。在UML中,使用一个带空心菱形的实线———◇表示聚合关系,空心菱形指向的是代表“整体”的类。

(5) 组合关系。如果聚合关系中的表示“部分”的类的存在与否,与表示“整体”的类有着紧密的关系,如“公司”与“部门”之间的关系,那么就应该使用“组合”关系来表示这种关系。在UML中,使用带有实心菱形的实线———◆表示组合关系。

(6) 实现关系将说明和实现联系起来。接口是对行为而非实现的说明,而类则包含实现的结构。一个或多个类可以实现一个接口,而每个类分别实现接口中的操作。在UML中,实现关系用-----▷表示。

(7) 流关系将一个对象的两个版本以连续的方式连接起来。它表示一个对象的值、状态和位置的转换。流关系可以将类元角色在一次相互作用中连接起来。流的种类包括变成(同一个对象的不同版本)和复制(从现有对象创造出一个新的对象)两种。在UML中,用----->表示流关系。

说明:对于聚合关系和组合关系,各种文献的说法有些区别。在这些文献中,首先定义聚集关系(整体与部分的关系),然后再把聚集关系分为两种,分别是组合聚集(相当于上述的“组合关系”)和共享聚集(相当于上述的“聚合关系”)。

5.2.4 图形

UML 2.0 包括 14 种图，分别列举如下。

(1) 类图(class diagram)：描述一组类、接口、协作和它们之间的关系。在面向对象系统的建模中，最常见的图就是类图。类图给出了系统的静态设计视图，活动类的类图给出了系统的静态进程视图。

(2) 对象图(object diagram)：描述一组对象及它们之间的关系。对象图描述了在类图中所建立的事物实例的静态快照。和类图一样，这些图给出系统的静态设计视图或静态进程视图，但它们是从真实案例或原型案例的角度建立的。

(3) 构件图(component diagram)：描述一个封装的类和它的接口、端口，以及由内嵌的构件和连接件构成的内部结构。构件图用于表示系统的静态设计实现视图。对于由小的部件构建大的系统来说，构件图是很重要的。构件图是类图的变体。

(4) 组合结构图(composite structure diagram)：描述结构化类(如构件或类)的内部结构，包括结构化类与系统其余部分的交互点。它显示联合执行包含结构化类的行为的构件配置。组合结构图用于画出结构化类的内部内容。

(5) 用例图(use case diagram)：描述一组用例、参与者(一种特殊的类)及它们之间的关系。用例图给出系统的静态用例视图。这些图在对系统的行为进行组织和建模时非常重要。

(6) 顺序图(sequence diagram，序列图)：是一种交互图(interaction diagram)。交互图展现了一种交互，它由一组对象或角色以及它们之间可能发送的消息构成。交互图专注于系统的动态视图。顺序图是强调消息的时间次序的交互图。

(7) 通信图(communication diagram)：也是一种交互图，它强调收发消息的对象或角色的结构组织。顺序图和通信图表达了类似的基本概念，但每种图所强调的概念不同，顺序图强调的是时序，通信图则强调消息流经的数据结构。在 UML 1.X 版本中，通信图被称为协作图(cooperation diagram)。

(8) 定时图(timing diagram，计时图)：也是一种交互图，它强调消息跨越不同对象或角色的实际时间，而不仅仅只是关心消息的相对顺序。

(9) 状态图(state diagram)：描述一个状态机，它由状态、转移、事件和活动组成。状态图给出了对象的动态视图。它对于接口、类或协作的行为建模尤为重要，而且它强调事件导致的对象行为，这非常有助于对反应式系统建模。

(10) 活动图(activity diagram)：将进程或其他计算的结构展示为计算内部一步步的控制流和数据流。活动图专注于系统的动态视图。它对系统的功能建模特别重要，并强调对象间的控制流程。

(11) 部署图(deployment diagram)：描述对运行时的处理结点及在其中生存的构件的配置。部署图给出了体系结构的静态部署视图，通常一个结点包含一个或多个部署图。

(12) 制品图(artifact diagram)：描述计算机中一个系统的物理结构。制品包括文件、数据库和类似的物理比特集合。制品图通常与部署图一起使用。制品也给出了它们实现的类和构件。

(13) 包图(package diagram)：描述由模型本身分解而成的组织单元，以及它们的依赖关系。

(14) 交互概览图(interaction overview diagram)：是活动图和顺序图的混合物。

5.3 用 例 图

用例实例是在系统中执行的一系列动作,这些动作将生成特定参与者可见的价值结果。一个用例定义一组用例实例。它确定了一个和系统参与者进行交互、并可由系统执行的动作序列。用例模型描述的是外部参与者(actor)所理解的系统功能。用例模型用于需求分析阶段,它的建立是系统开发者和用户反复讨论的结果,表明了开发者和用户对需求规格达成的共识。

在UML中,用例表示为一个椭圆。图5-1显示了希赛教育图书管理系统的用例图。其中,"新增书籍信息""查询书籍信息""修改书籍信息""登记外借信息""查询外借信息""统计金额与册数"等都是用例的实例。

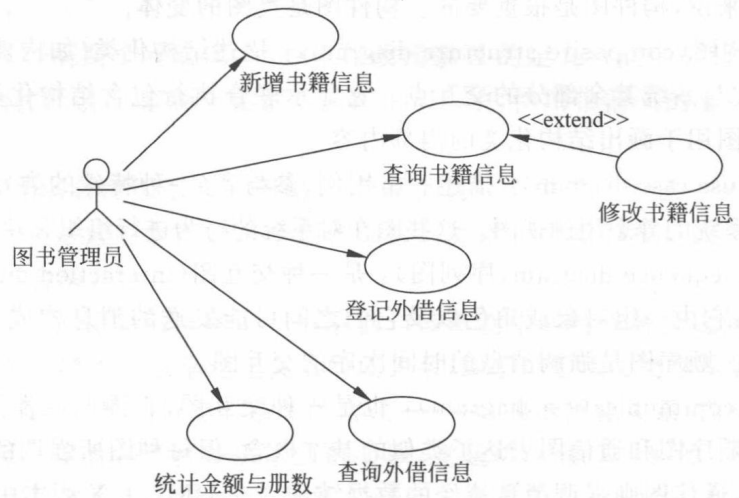


图5-1 用例图示例

1. 参与者

参与者代表与系统接口的任何事物(包括其他系统),它是指代表某一种特定功能的角色,因此参与者是虚拟的概念。在UML中,用一个小人表示参与者。

图5-1中的"图书管理员"就是参与者。对于该系统来说,可能可以充当图书管理员角色的有多个人,因为他们对于系统而言均起着相同的作用,扮演相同的角色,因此,只用一个参与者表示。切记不要为每一个可能与系统交互的人画出一个参与者。

可以通过以下问题找到系统的相关参与者。

- 谁是系统的主要用户?
- 谁从系统获得信息?
- 谁向系统提供信息?
- 谁从系统删除信息?
- 谁支付、维护系统?
- 谁管理系统?
- 系统需要与其他哪些系统交互?

- 系统需要操纵哪些硬件？
- 在预设的时间内，有事情自动发生吗？
- 系统从哪里获得信息？
- 谁对系统的特定需求感兴趣？
- 几个人在扮演同样的角色吗？
- 一个人扮演几个不同的角色吗？
- 系统使用外部资源吗？
- 系统用在什么地方？

2. 用例

用例是对系统行为的动态描述，它可以促进设计人员、开发人员与用户的沟通，理解正确的需求，还可以划分系统与外部实体的界限，是系统设计的起点。在识别出参与者之后，可以使用下列问题帮助识别用例。

- 每个参与者的任务是什么？
- 有参与者将要创建、存储、修改、删除或读取系统中的信息吗？
- 什么用例会创建、存储、修改、删除或读取这个信息？
- 参与者需要通知系统外部的突然变化吗？
- 需要把系统中正在发生的事情通知参与者吗？
- 什么用例将支持和维护系统？
- 所有的功能需求都对应到用例中了吗？
- 系统需要何种输入输出？输入从何处来？输出到何处？
- 当前运行系统的主要问题是什么？

5.4 类图和对象图

在面向对象建模技术中，对象指现实世界中有意义的事物，具有封装性和自治性的特点；而类指具有相同属性和行为的一组对象。类、对象和它们之间的关联是面向对象技术中最基本的元素。对于一个想要描述的系统，其类模型和对象模型揭示了系统的结构。在UML中，类和对象模型分别由类图和对象图表示。类图技术是OO方法的核心。图5-2显示了希赛教育图书管理系统的类图。

对象与人们对客观世界的理解相关，它通常用来描述客观世界中某个具体的事物。而类是对一组具有相同属性，表现相同行为的对象的抽象。因此，对象是类的实例。在UML中，类的可视化表示为一个划分成3个格子的长方形(下面两个格子可省略)。在图5-2中，“书籍”“借阅记录”等都是一个类。

(1) 类的命名：最顶部的格子包含类的名字。类的命名应尽量用应用领域中的术语，应明确、无歧义，以利于开发人员与用户之间的相互理解和交流。

(2) 类的属性：中间的格子包含类的属性，用以描述该类对象的共同特点。该项可省略。在图5-2中，“书籍”类有“书名”“书号”等属性。UML规定类的属性的语法为：“可见性 属性名：类型 = 默认值{约束特性}”。

① 可见性：包括Public、Private和Protected，分别用+、-、#表示。

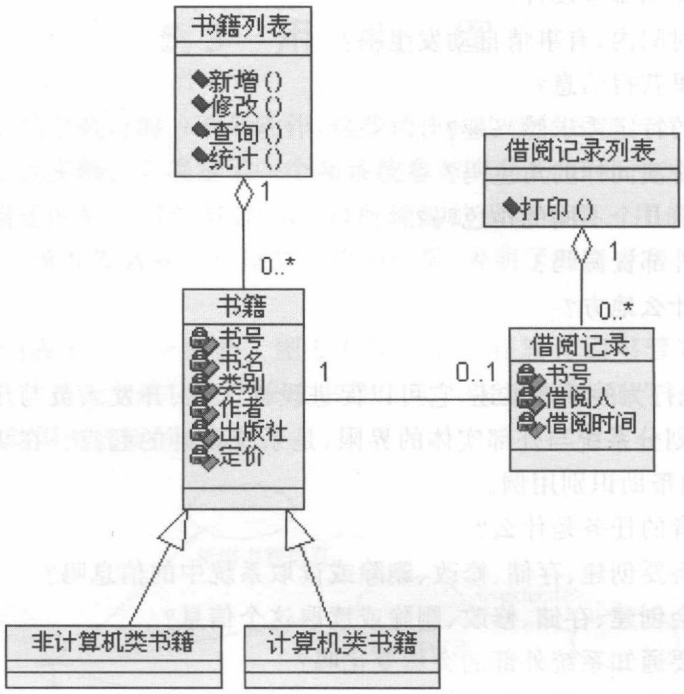


图 5-2　希赛教育图书管理系统的类图

② 类型：表示该属性的种类，它可以是基本数据类型(如整数、实数、布尔型等)，也可以是用户自定义的类型。一般它由所涉及的程序设计语言确定。

③ 约束特性：用户对该属性性质一个约束的说明。例如，“{只读}”说明该属性是只读属性。

(3) 类的操作(Operation)：该项可省略。操作用于修改、检索类的属性或执行某些动作。操作通常也被称为功能，但是它们被约束在类的内部，只能作用到该类的对象上。操作名、返回类型和参数表组成操作界面。UML 规定操作的语法为：“可见性：操作名(参数表)：返回类型 {约束特性}”。

类图描述了类和类之间的静态关系。定义了类之后，就可以定义类之间的各种关系了，请参考 5.2.3 节的介绍。与数据模型不同，类图不仅显示了信息的结构，同时还描述了系统的行为。类图是面向对象建模中最重要的模型。

UML 中对象图与类图具有相同的表示形式，对象图可以看作是类图的一个实例。对象之间的链(Link)是类之间的关联的实例。

5.5　交　互　图

交互图是表示各组对象如何依某种行为进行协作的模型。通常可以使用一个交互图表示和说明一个用例的行为。在 UML 中，包括 4 种不同形式的交互图，分别是顺序图、通信图、定时图和交互概览图。其中，顺序图强调的是时序，通信图则强调消息流经的数据结构，定时图强调消息跨越不同对象或角色的实际时间，交互概览图是活动图和顺序图的混合物。

顺序图和通信图是两种基本的交互图，它们之间没有什么本质不同，只是排版不尽相同而已；定时图和交互概览图是两种特殊的变体。

本节简单介绍顺序图、通信图和定时图，有关交互概览图的知识，请阅读 5.7.3 节。

5.5.1 顺序图

顺序图用来描述对象之间动态的交互关系，着重体现对象间消息传递的时间顺序。顺序图允许直观地表示出对象的生存期，在生存期内，对象可以对输入消息做出响应，并且可以发送信息。

如图 5-3 所示，顺序图存在两个轴：水平轴表示不同的对象；垂直轴表示时间，表示对象及类的生命周期。

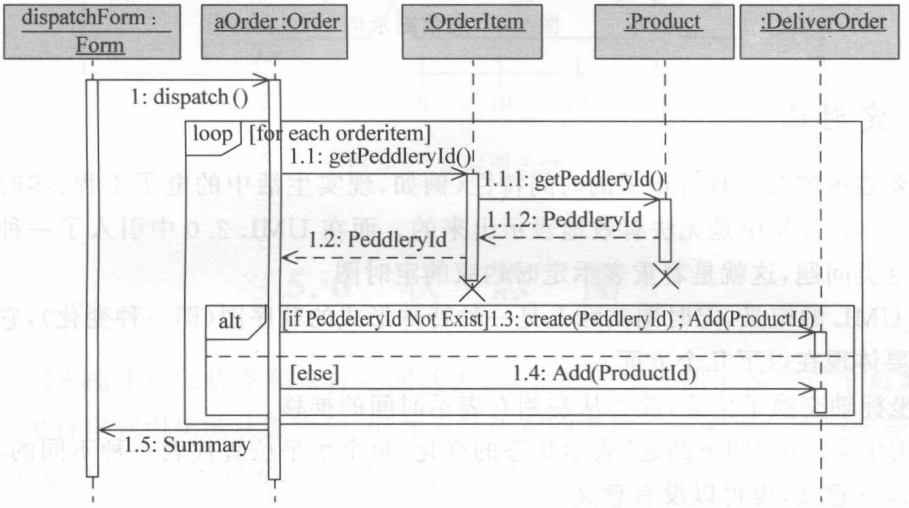


图 5-3 顺序图示例

对象间的通信通过在对象的生命线间画消息表示。消息的箭头指明消息的类型。顺序图中的消息可以是信号、操作调用或类似于 C++中的 RPC 和 Java 中的 RMI。收到消息时，接收对象立即开始执行活动，即对象被激活。通过在对象生命线上显示一个细长矩形框表示激活。

消息可以用消息名及参数标识，消息也可带有顺序号。消息还可带有条件表达式，表示分支或决定是否发送消息。如果用于表示分支，则每个分支相互排斥，即在某一时刻仅可发送分支中的一个消息。

5.5.2 通信图

通信图强调收发消息的对象或角色的结构组织。顺序图和通信图表达了类似的基本概念，但每种图强调概念的不同视图，顺序图强调时序，通信图强调消息流经的数据结构。图 5-4就是与图 5-3 相对应的通信图，从图 5-4 中可以很明显地发现通信图与顺序图之间的异同点。

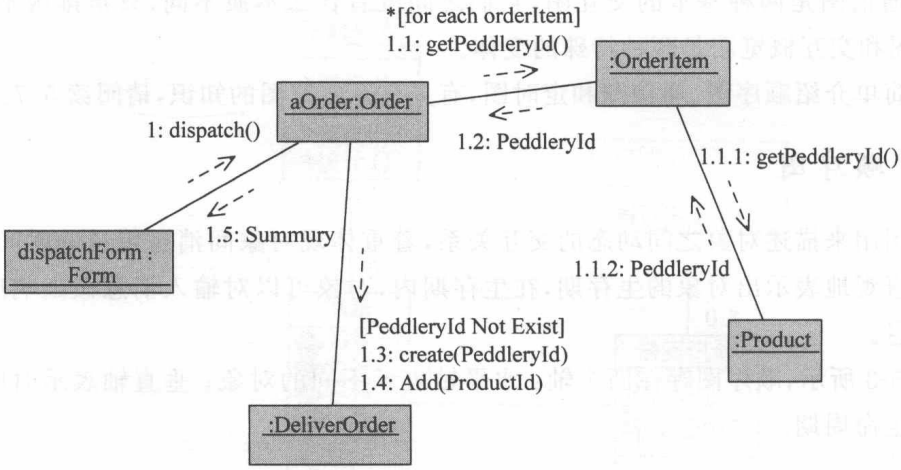


图 5-4　通信图示例

5.5.3　定时图

如果要表示的交互具有很强的时间特性(例如,现实生活中的电子工程、实时控制等系统中),在 UML 1.X 中是无法有效地表示出来的。而在 UML 2.0 中引入了一种新的交互图来解决这类问题,这就是着重表示定时约束的定时图。

根据 UML 的定义,定时图实际上是一种特殊形式的顺序图(即一种变化),它与顺序图的区别主要体现在以下几个方面。

(1) 坐标轴交换了位置,改为从左到右表示时间的推移。

(2) 用生命线的"凹下凸起"表示状态的变化,每个水平位置代表一种不同的状态,状态的顺序可以有意义,也可以没有意义。

(3) 生命线可以跟在一根线后面,在这根线上显示一些不同的状态值。

(4) 可以显示一个度量时间值的标尺,用刻度表示时间间隔。

图 5-5 是一个定时图的实例,其中小黑点加曲线表示的是注释。它用来表示一个电子门禁系统的控制逻辑,该门禁系统包括门(物理的门)、智能读卡器(读取用户的智能卡信息)、处理器(用来处理是否开门的判断)。

在这个例子中,它所表示的意思是一开始读卡器是启用的(等用户刷卡)、处理器是空闲的(没有验证的请求)、门是关的;接着,当用户刷卡时,读卡器就进入了"等待校验"的状态,并发一个消息给处理器,处理器就进入了校验状态。如果校验通过,就发送一个"禁用"消息给读卡器(门开的时候,读卡器就可以不工作了),使读卡器进入禁用状态;并且处理器转入启用状态,这时门的状态变成了"开"。而门"开"了 30s(根据时间刻度得知)之后,处理器将会把它再次"关"上,并且发送一个"启用"消息给读卡器(门关了,读卡器又要重新工作了),这时读卡器再次进入启用状态,而处理器已经又回到空闲状态。

从上面的例子中,不难看出定时图所包含的图元并不多,主要包括生命线、状态、状态变迁、消息、时间刻度,可以根据需要来使用它。

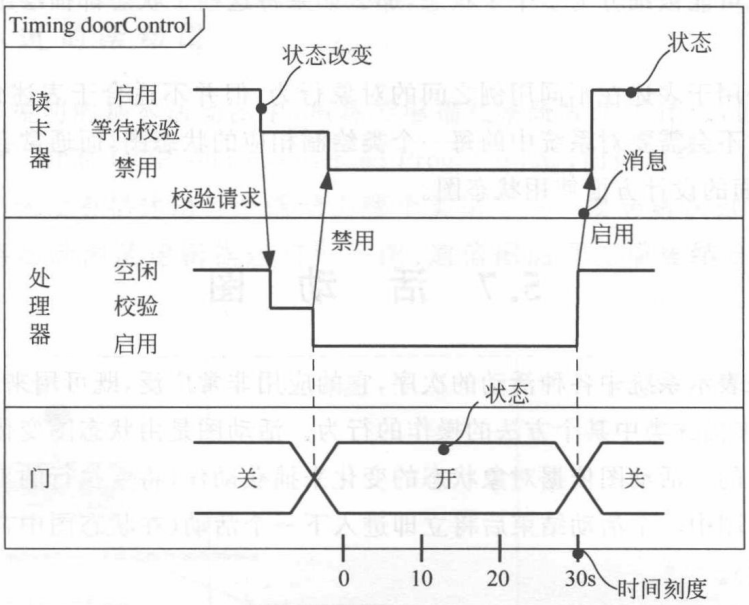


图 5-5　定时图示例

5.6　状　态　图

状态图用来描述对象状态和事件之间的关系，通常用状态图来描述单个对象的行为。它确定了由事件序列引出的状态序列，但并不是所有的类都需要使用状态图来描述它的行为，只有具有重要交互行为的类，才会使用状态图来描述。

图 5-6 是一个数码冲印店的订单状态图实例，如图 5-6 所示，状态图包括以下部分。

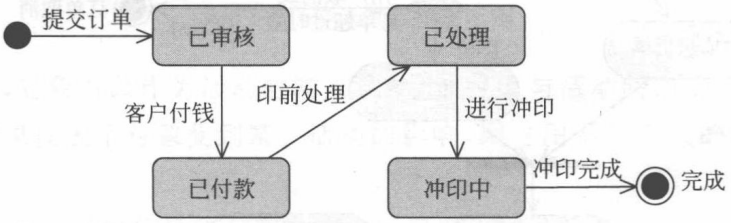


图 5-6　状态图示例

(1) 状态：又称中间状态，用圆角矩形框表示。

(2) 初始状态：又称初态，用一个黑色的实心圆圈表示。一张状态图中只能有一个初始状态。

(3) 结束状态：又称终态，在黑色的实心圆圈外面套上一个空心圆。在一张状态图中可能有多个结束状态。

(4) 状态转移：用箭头说明状态的转移情况，并用文字说明引发这个状态变化的相应事件是什么。

一个状态也可能被细分为多个子状态,那么如果将这些子状态都描绘出来,则这个状态就是复合状态。

状态图适合用于表述在不同用例之间的对象行为,但并不适合于表述包括若干协作的对象行为。通常不会需要对系统中的每一个类绘制相应的状态图,而通常会在业务流程、控制对象、用户界面的设计方面使用状态图。

5.7 活 动 图

活动图用来表示系统中各种活动的次序,它的应用非常广泛,既可用来描述用例的工作流程,也可以用来描述类中某个方法的操作的行为。活动图是由状态图变化而来的,它们各自用于不同的目的。活动图依据对象状态的变化来捕获动作(将要执行的工作或活动)与动作的结果。活动图中一个活动结束后将立即进入下一个活动(在状态图中,状态的变迁可能需要事件的触发)。

5.7.1 基本活动图

图 5-7 给出了一个基本活动图的例子。

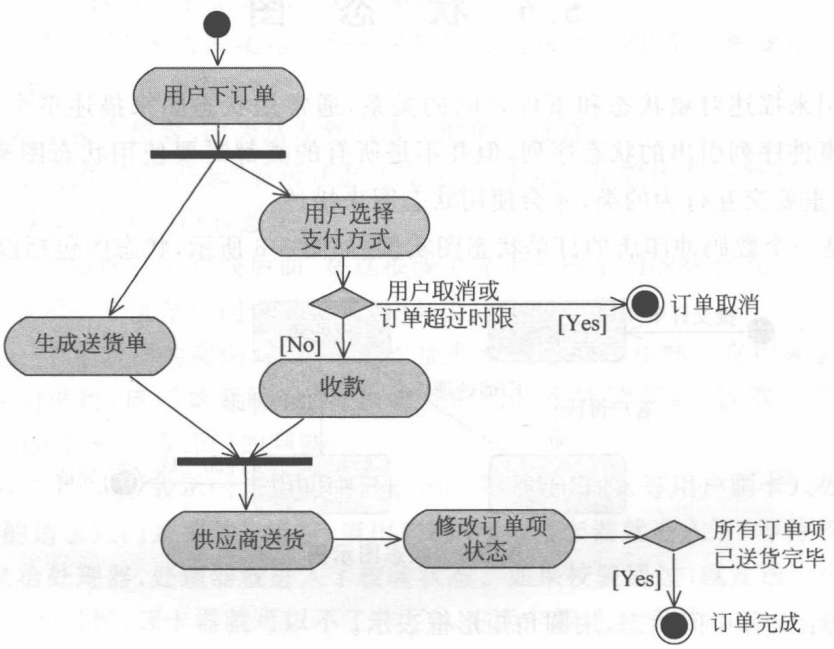


图 5-7 基本活动图示例

如图 5-7 所示,活动图与状态图类似,包括初始状态、终止状态,以及中间的活动状态,每个活动之间,也就是一种状态的变迁。在活动图中,还引入了以下几个概念。

(1) 判定:说明基于某些表达式的选择性路径,在 UML 中使用菱形表示。

(2) 分叉与结合:由于活动图建模时经常会遇到并发流,因此,在 UML 中引入了如图 5-7所示的粗实线表示分叉和结合。

5.7.2 带泳道的活动图

在 5.7.1 节说明的基本活动图中，虽然能够描述系统发生了什么，但无法说明完成这个活动的对象。针对 OOP(Object-Oriented Programming，面向对象的程序设计)而言，这就意味着活动图没有描述出各个活动由哪个类来完成。要想解决这个问题，可以通过泳道。泳道将活动图的逻辑描述与顺序图、通信图的责任描述结合起来，如图 5-8 所示。

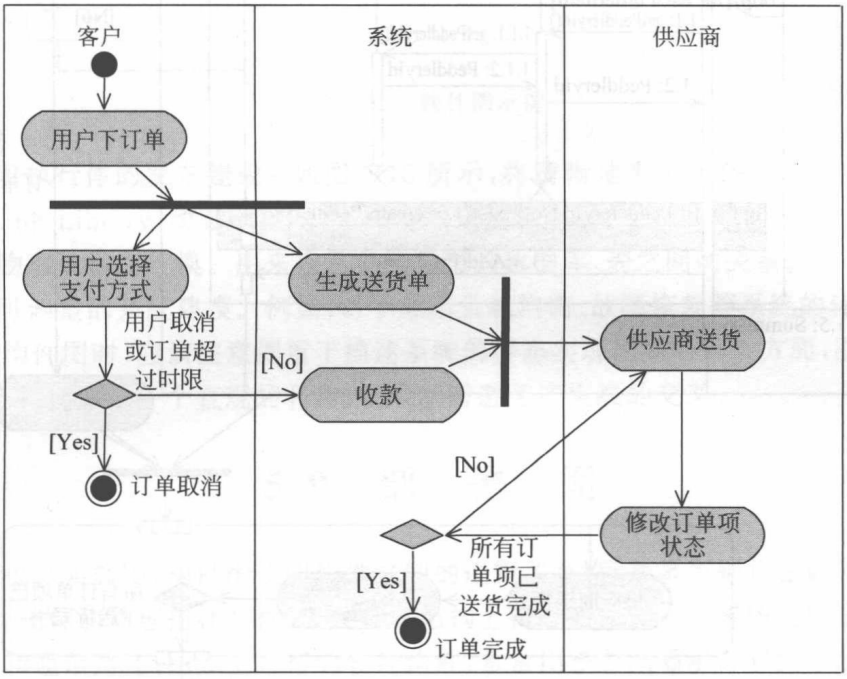


图 5-8 带泳道活动图示例

在活动图中，对象可以作为活动的输入或输出，对象与活动间的输入/输出关系由虚线箭头表示。如果仅表示对象受到某一活动的影响，则可用不带箭头的虚线连接对象与活动。

在活动图中，可以通过信号的发送和接收标记表示信号的发送和接收；发送和接收标记也可与对象相连，用于表示消息的发送者和接收者。

5.7.3 交互概览图

交互概览图并没有引入新的事物，其所有的图示法都已经在顺序图和活动图中阐述过。例如，在图 5-7 中关于“用户订单处理”的活动图中，如果想表达出“用户下订单”和“生成送货单”两个活动结点内的对象控制流，就可以结合 5.5.1 节中的顺序图的知识，绘制出一张交互概览图，其结果如图 5-9 所示。

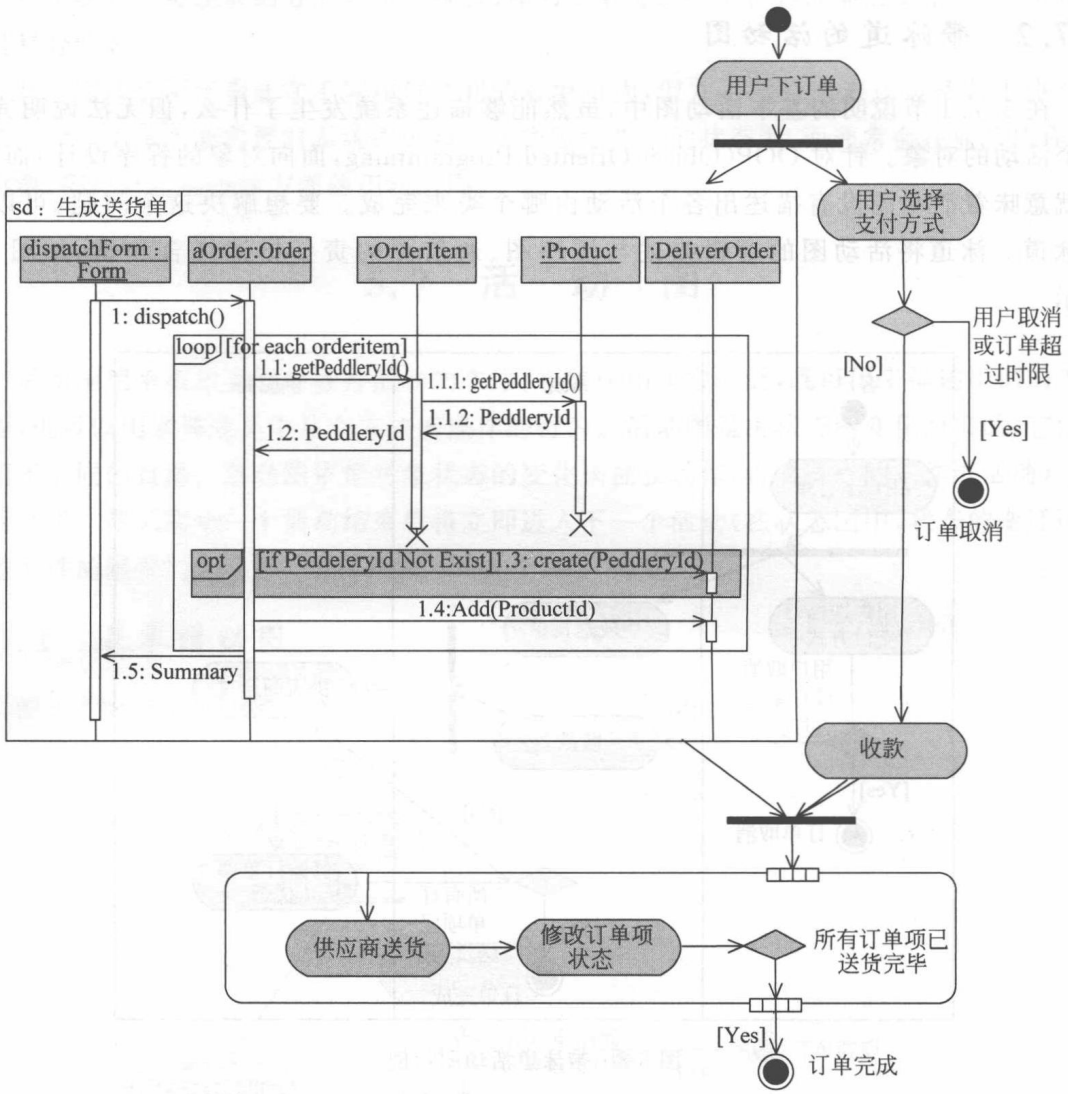


图 5-9　交互概览图示例

5.8　构　件　图

对面向对象系统物理方面进行建模，需要使用两种图，分别是构件图和部署图。构件图可以有效地显示一组构件，以及它们之间的关系。构件图中通常包括构件、接口以及各种关系。图 5-10 就是一个构件图的例子。

构件指源代码文件、二进制代码文件、可执行文件等，而构件图用来显示编译、链接或执行时构件之间的依赖关系。例如，在图 5-10 中，就是说明 QueryClient. exe 将通过调用 QueryServer. exe 来完成相应的功能，而 QueryServer. exe 则需要 Find. exe 的支持，Find. exe 在实现时调用了 Query. dll。

通常来说，可以使用构件图完成以下工作。

(1) 对源代码进行建模。构件图可以清晰地表示出各个不同源程序文件之间的关系。

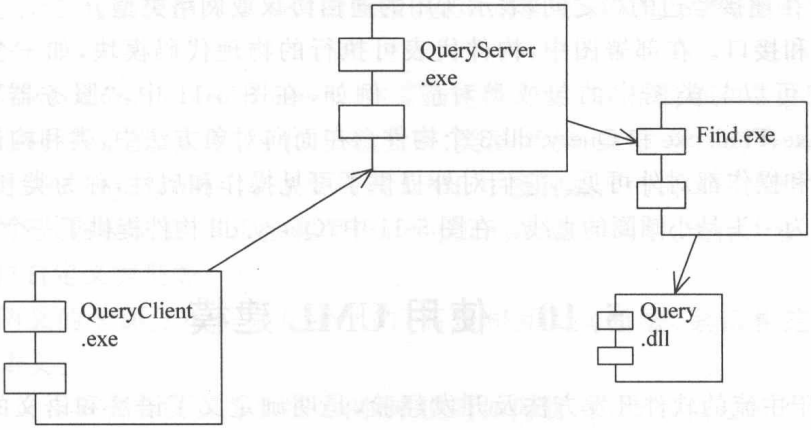


图 5-10　构件图示例

(2) 对可执行体的发布建模：如图 5-10 所示，将清晰地表示出各个可执行文件、DLL (Dynamic Link Library，动态链接库)文件之间的关系。

(3) 对物理数据库建模。用来表示各种类型的数据库、表之间的关系。

(4) 对可调整的系统建模。例如，对于应用负载均衡、故障恢复等系统的建模。

在绘制构件图时，应该注意侧重于描述系统的静态实现视图的一个方面，图形不要过于简化，应该为构件图取一个直观的名称，在绘制时避免产生线的交叉。

5.9　部　署　图

部署图也称实施图，构件图说明构件之间的逻辑关系，而部署图则是在此基础上更进一步，描述系统硬件的物理拓扑结构，以及在此结构上执行的软件。部署图可以显示计算结点的拓扑结构和通信路径、结点上运行的软件构件，常常用于帮助理解分布式系统。

图 5-11 就是与图 5-10 对应的部署图，这样的图示可以使系统的安装、部署更为简单。在部署图中，通常包括以下一些关键的组成部分。

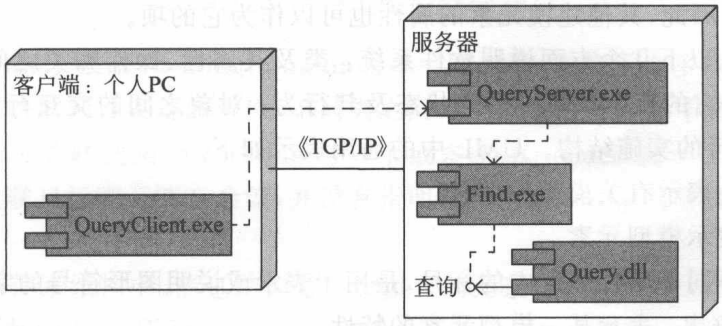


图 5-11　部署图示例

(1) 结点(Node)和连接。结点代表一个物理设备以及其上运行的软件系统，如一台 Windows Server 主机、一个 PC 终端、一台打印机、一个传感器等。如图 5-11 所示，“客户端：PC”和“服务器”就是两个结点。在 UML 中，使用一个立方体表示一个结点，结点名放在左上角。结点之间的连线表示系统之间进行交互的通信路径，在 UML 中被称为连接。

通信类型则放在连接旁边的《》之间,表示所用的通信协议或网络类型。

(2) 构件和接口。在部署图中,构件代表可执行的物理代码模块,如一个可执行程序等。逻辑上它可以与类图中的包或类对应。例如,在图 5-11 中,"服务器"结点中包含 QueryServer. exe、Find. exe 和 Query. dll 3 个构件。在面向对象方法中,类和构件等元素并不是所有的属性和操作都对外可见。它们对外提供了可见操作和属性,称为类和构件的接口。界面可以表示为一头是小圆圈的直线。在图 5-11 中,Query. dll 构件提供了一个"查询"接口。

5.10 使用 UML 建模

UML 基于主流的软件开发方法及开发经验,是明确定义了语法和语义的可视化建模语言。其中,图形表示的语法由实例和图形元素到语义模型中元素的映射表示,语义模型的语法和语义由元模型、自然语言和约束来说明。

UML 的 4 层元模型的体系结构如图 5-12 所示。其中元-元模型(meta-meta model)层定义了元模型(meta model)层的规格说明语言,元模型层为给定的建模语言定义规格说明,模型层用来定义特定软件系统的模型,用户对象(user object)用来构建给定模型的特定实例。在 4 层元模型中,UML 的结构主要体现在元模型中,分为 3 个逻辑包,分别是基础包(foundation)、行为元素包(behavioral elements)和一般机制包(general mechanisms),这些包依次又分为若干子包。例如,基础包分为核心元素、辅助元素、扩充机制、数据类型几个子包。

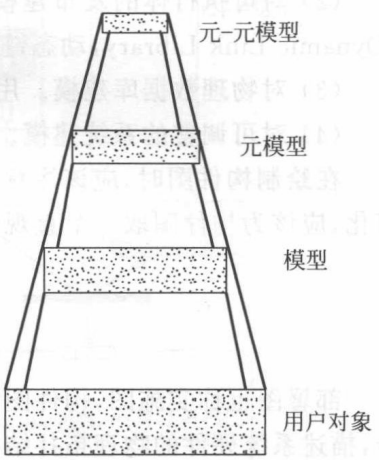


图 5-12 UML 的 4 层元模型体系结构

语义约束由对象约束语言 OCL 表示,OCL 基于一阶谓词逻辑,每一个 OCL 表达式都处于一些 UML 模型元素的背景下(由 self 引用),可使用该元素的属性和关系作为其项(term),同时 OCL 定义了在集合(sets)、袋(bags)等上的公共操作集和遍历建模元素间关系的构造,因此,其他建模元素的属性也可以作为它的项。

UML 模型从以下几个方面说明软件系统:类及其属性、操作和类之间的静态关系,类的包(package)和它的依赖关系;类的状态及其行为;对象之间的交互行为,使用事例;源码的结构以及执行的实施结构。UML 中的通用表示如下。

(1) 字符串:表示有关模型的信息。

(2) 名字:表示模型元素。

(3) 标号:不同于编程语言中的标号,是用于表示或说明图形符号的字符串。

(4) 特殊字符串:表示某一模型元素的特性。

(5) 类型表达式:声明属性、变量及参数,含义同编程语言中的类型表达式。

(6) 实体类型(stereotype):它是 UML 的扩充机制,运用实体类型可定义新类型的模型元素。

(7) UML 语义部分是对 UML 的准确表示,主要由以下 3 部分组成。

① 通用元素(common element):主要描述 UML 中各元素的语义。通用元素是 UML

中的基本构造单位，包括模型元素和视图元素，模型元素用来构造系统，视图元素用来构成系统的表示成分。

② 通用机制(common mechanism)：主要描述使 UML 保持简单和概念上一致的机制的语义，包括定制、标记值、注记、约束、依赖关系、类型-实例、类型-类的对应关系等机制。

③ 通用类型(common type)：主要描述 UML 中各种类型的语义。这些类型包括布尔类型、表达式类型、列表类型、多重性类型(multiplicity)、名字类型、坐标类型、字符串类型、时间类型、用户自定义类型等。

UML 中语义的 3 部分不是相互独立的，而是相互交叉重叠、紧密相连，共同构成了 UML 的完整语义。

下面以 4.4.3 节中会议安排系统为例，来说明如何使用 UML 描述软件体系结构及其元素。

图 5-13 是用 UML 的类图表示会议安排系统，该图描述了会议安排系统的领域模型，包括领域中的类及其继承和关联的关系。

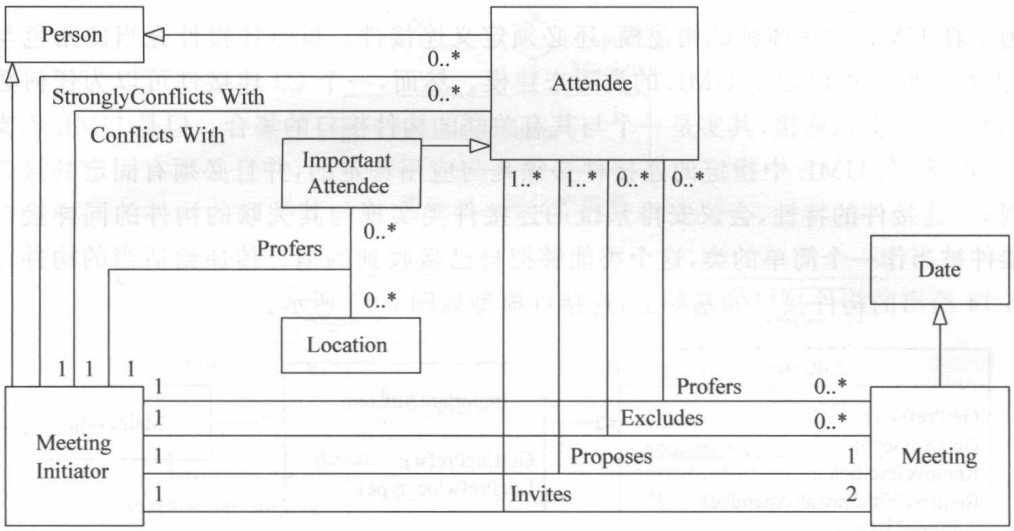


图 5-13　会议安排系统的类图

在图 5-13 中，除了限制 MeetingInitiator 为单一实例外，省略了许多体系结构细节。例如，把领域中的类映射到可实现的构件，不同类之间的交互顺序等。图中还省略了类交互的语义。例如，Invites 关联把两个 Meeting 与一个或多个 Attendee 或一个 MeetingInitiator 关联，然而这种关联只标明了两个会议，并没有指明这两个 Meeting 打算提交会议可能举行的日期范围。

在 C2 风格的体系结构中，消息结构是一个重要的元素。因此，在 UML 设计中，要对类的接口进行建模，如图 5-14 所示。

在图 5-14 中，接口 ImportantMtgInit 和 ImportantMtgAttend 分别继承了接口 MtgInit 和 MtgAttend，唯一的区别是增加了会议地点的请求和通知。要注意的是，图 5-14 中每一个方法签名(Method Signature，即 UML 的操作)都对应了一个 C2 消息(见 4.4.3 节)。UML 模型中的所有操作将作为异步消息传递来实现。正因如此，图 5-14 中的方法签名没有写返回类型。

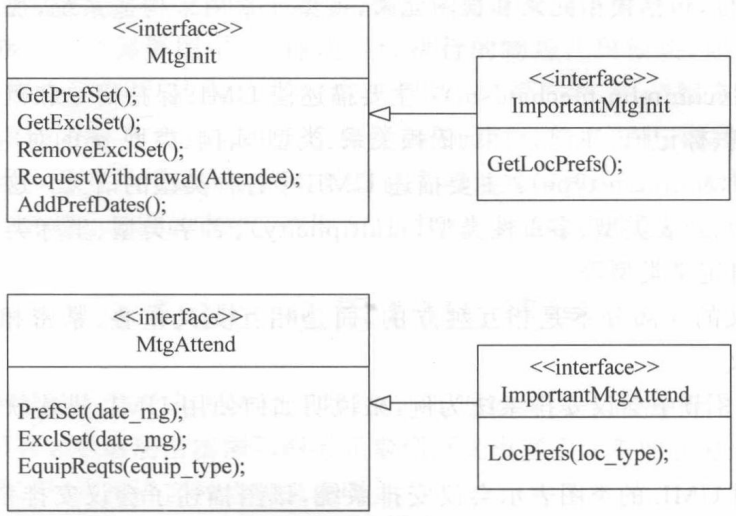


图 5-14　会议安排系统类接口

为了在 UML 中为体系结构建模，还必须定义连接件。虽然连接件充当的角色与构件不同，但是它们也可以使用 UML 的类图来建模。然而，一个 C2 连接件可以为任何数量和类型的 C2 构件提供连接，其实是一个与其有关联的构件接口的集合。但是 UML 不支持这种属性，相反，在 UML 中指定的连接件必须是与应用特定的，并且必须有固定的接口。为了体现 C2 连接件的特性，会议安排系统的连接件类实现与其关联的构件的同样接口。每个连接件被当作一个简单的类，这个类能够把自己接收到的消息传递给适当的构件，因此，在图 5-14 给出的构件接口的基础上，连接件模型如图 5-15 所示。

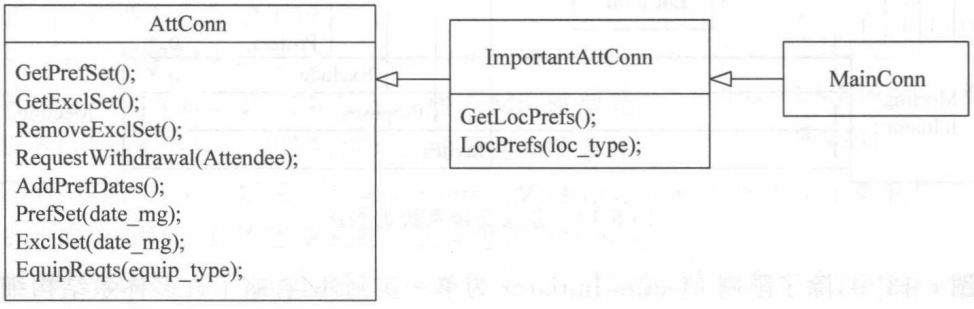


图 5-15　C2 连接件模型

在领域模型层次上，连接件不能增加任何功能，因此在图 5-13 中没有被标记出来。图 5-16 是一个细化了的会议安排系统的类图，该图描述了类之间的接口关系。

在图 5-16 中，每个实线圆弧旁边的文字是接口的名称，描述了接口的实现。每个虚线箭头表示类对接口的依赖性。要特别说明的是，在图 5-14 中，类 Attendee 和 ImportantAttendee 是由接口继承进行关联的，但在图 5-16 中没有明显体现出来。图 5-16 省略了图 5-13 中的 Location、Meeting 和 Date 类，因为这些类只是代表系统中构件之间交换的数据；同时，也省略了构件超类 Person 和连接件超类 Conn。

图 5-17 描述了 MeetingInitiator 类实例和 Attendee 类以及 ImportantAttendee 类实例之间的协作。

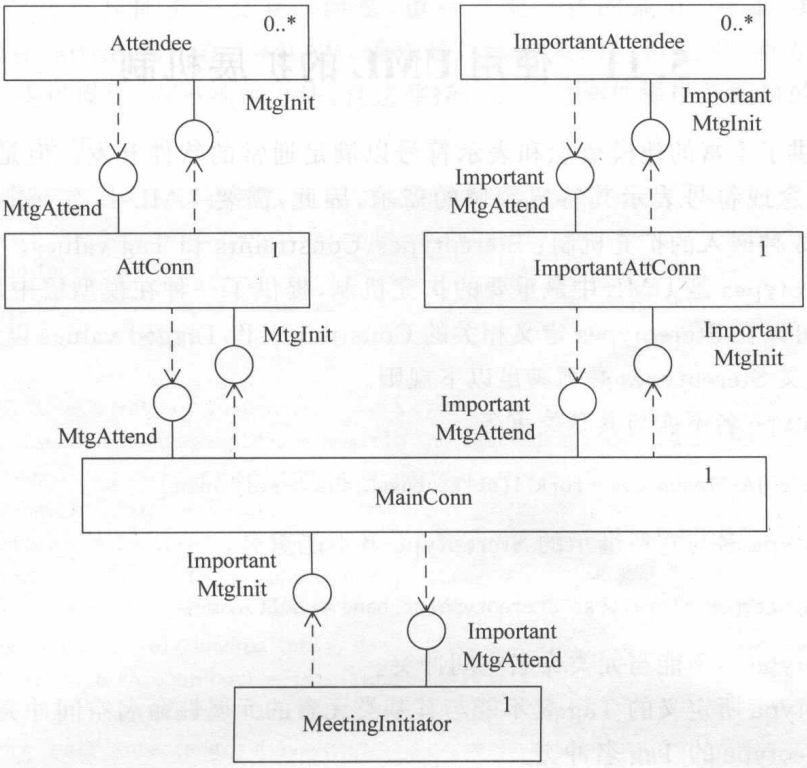


图 5-16　细化的类图

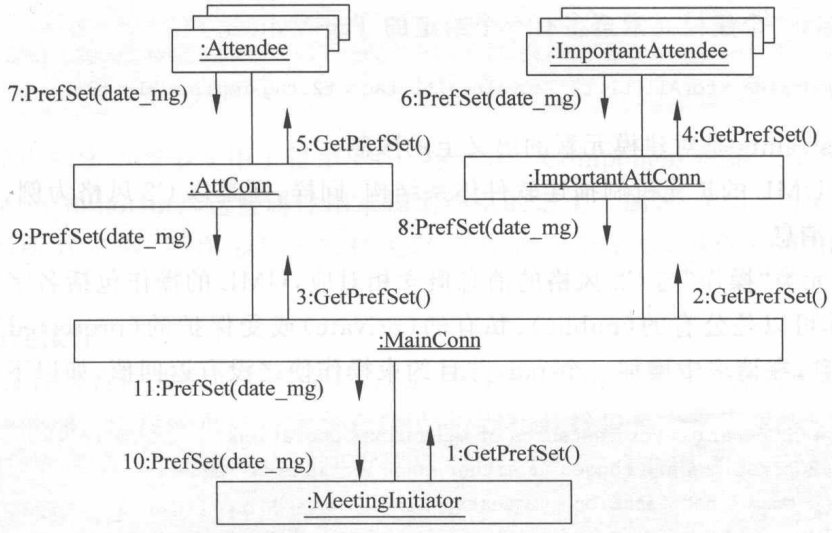


图 5-17　会议安排系统的协作图

在图 5-17 中，MeetingInitiator 发出会议安排日期请求，连接件 MainConn 的实例把这个请求转交给其上方的连接件 AttConn 和 ImportantAttConn 的实例，这两个实例再把这个请求转交给其上方的构件，每个与会人员构件选择自己喜欢的日期，并把该选择通知其下方的构件，这个通知信息最终由连接件转交给 MeetingInitiator。但是，如果 MeetingInitiator 发出会议地点安排请求，则 MainConn 只把该请求转交给 ImportantAttConn，则任何 Attendee 类的实例都无法接收该请求。

5.11 使用 UML 的扩展机制

UML 提供了丰富的建模概念和表示符号以满足通常的软件开发。但是,用户有时候需要另外的概念或符号表示其特定领域的需求,因此,需要 UML 具有一定的扩充能力。UML 提供了 3 种嵌入的扩充机制: Stereotypes、Constraints 和 Tag values。

(1) Stereotypes 是 UML 中最重要的扩充机制,提供了一种在模型层中加入新的建模元素的方式,可以在 Stereotypes 定义相关的 Constraints 和 Tagged values 以说明特定的语义和特征。定义 Stereotypes 必须满足以下规则。

① Stereotype 名不能与其基类重名,

```
Stereotype.oclAllInstances→forAll(st|st.baseClass <> self.name)
```

② Stereotype 名与它所继承的 Stereotype 名不能重名,

```
Self.allSupertypes→forAll(st:Stereotype|st.name <> self.name)
```

③ Stereotype 名不能与元类命名空间冲突。

④ Stereotype 所定义的 Tag 名不能与其基类元素的元属性命名空间冲突,也不能与它所继承的 Stereotype 的 Tag 名冲突。

(2) 对于模型中的建模元素,Tag Values 允许加入新的属性。Tag 表明了建模元素可扩展的特性的名称,value 可以是任意的值,值的范围取决于用户或工具对 Tag 的解释。对每一个 Tag 名,一个建模元素至多有一个给定的 Tag Values。

```
Self.Taggedvalue→forAll(t1,t2:Tagvalues|t1.tag = t2.tag implies t1 = t2)
```

(3) Constraints 是对建模元素的语义上的限制。

下面,用 UML 的扩充机制描述软件体系结构,同样,还是以 C2 风格为例。

1. C2 的消息

UML 的元类"操作"与 C2 风格的消息概念相对应,UML 的操作包括名字、参数列表和返回值。操作可以是公有的(public)、私有的(private)或受保护的(protected)。为了描述 C2 风格的消息,在请求中增加一个 tag,并且约束操作使之没有返回值,如以下程序所示。

```
Stereotype C2Operation for instances of meta class Operation
 -- 1 -- C2Operations are tagged as either notifications or requests.
c2MsgType : enum { notification, requests }
 -- 2 -- C2Operations are tagged as either incoming or outgoing.
c2MsgDir : enum { in, out }
 -- 3 -- C2messages do not have return values.
Self.parameter -> forAll(p | p.kind <> return)
```

以上程序中的实体类型既包含 tagged values(c2MsgType 和 c2MsgDir),又包含一个通用的对实体类型化的操作的参数约束。

2. C2 的构件

UML 的元类"类"和 C2 风格的构件概念接近。类可以提供多重带有操作(operation)

的接口，可以参与与其他类的交互。但是，也有不太一致的地方，例如，类可以有方法(method)和属性(attribute)，在 UML 中，操作是过程抽象的规格说明，而方法是过程体。C2 中的构件只提供操作，而不提供方法，且这些操作必须是构件提供的接口的一部分，而不是构件本身的一部分。

```
Stereotype C2Interface for instances of meta class Interface
 --1-- A C2Interface has a tagged value identifying its position.
c2pos : enum { top, bottom }
 --2-- All C2Interface operations must have stereotype C2Operation
self.operation -> forAll(o.stereotype = C2Operation)

Stereotype C2Component for instances of meta class Class
 --1-- C2Components must implement exactly two interfaces, which must be
   -- C2Interfaces, one top, and the other bottom.
self.interface -> size = 2 and
self.interface -> forAll(i | i.stereotype = C2Interface) and
self.interface -> exists(i | i.c2pos = top) and
self.interface -> exists(i | i.c2pos = bottom)
 --2-- Requests travel "upward" only, i.e., they are sent through top interface
   -- and received through bottom interfaces.
let topInt = self.interface -> select(i | i.c2pos = top) in
let botInt = self.interface -> select(i | i.c2pos = bottom) in
topInt.operation -> forAll(o | o.c2MsgType = request) implies (o.c2MsgDir =
out)) and
botInt.operation -> forAll(o | o.c2MsgType = request) implies (o.c2MsgDir = in))
 --3-- Notifications travel "downward" only. Similar to the constraint above.
 --4-- Each C2Component has at least one instance in the running system.
self.allInstances -> size >= 1
```

以上程序中的实体类型使用了很多 OCL 特征，C2 Component 的第二个约束定义了附件的属性(topInt 和 botInt)，这些属性用来辅助约束的定义。属性 allInstances 返回所关联的模型元素的所有实例，操作 select 选择关联集合的一个子集，条件是其指定的表达式为真。

3. C2 的连接件

C2 的连接件与构件共享了很多约束，但是，在下面，把构件和连接件当作不同的体系结构组成成分来处理。连接件可以不定义它自己的接口，其接口是由它连接的构件决定的。

与描述 C2 的构件一样，也可以使用实体类型来描述 C2 的连接件，下面重用一些约束，再增加两个新约束。为了把构件的约束引入连接件中，首先定义 3 个构件的附加属性，这些附加属性是决定构件接口所必需的。

```
Stereotype C2AttachOverComp for instances of meta class Association
 --1-- C2attachments are binary associations.
self.associationEnd -> size = 2
 --2-- One end of the attachment must be a single C2Component.
let ends = self.associationEnd in
ends[1].multiplicity.min = 1 and ends[1].multiplicity.max = 1 and
ends[1].class.sterrotype = C2Component
```

```
-- 3 -- The other end of the attachment must be a single C2Connector.
let ends = self.associationEnd in
ends[2].multiplicity.min = 1 and ends[2].multiplicity.max = 1 and
ends[2].class.sterrotype = C2Connector

Stereotype C2AttachUnderComp for instance of meta class Association. Same as
C2AttachOverComp, but with the order reversed.

Stereotype C2AttachConnConn for intstance of meta class Association
-- 1 -- C2 attachments are binaty associations.
self.associationEnd -> size = 2
-- 2 -- Each end of the association must be on a C2 connector.
self.associationEnd -> forAll(ae | ae.multiplicity.min = 1 and
    ae. multiplicity.max = 1 and ae.class.stereotype = C2Connector)
-- 3 -- The two ends are not the same C2Connector.
self.associationEnd[1].class <> self.associationEnd[2].class

Stereotype C2Connector for instance of meta class Class
-- 1 through 3 -- Same as constraints 1 - 3 on C2Component.
-- 4 -- Each C2 connector has exactly one instance in the running system.
self.allInstances -> size = 1
-- 5 -- The top interface of a connector is determined by the components and
  connectors attached to its bottom.
let topInt = self.interface -> select(i | i.c2pos = top) in
let downAttach = self.associationEnd.associatin - > select(a |
    a.associationEnd[2] = self) in
let topsIntsBelow = downAttach.associationEnd[1].interface -> select(i | i.c2pos = top) in
topsInsBelow.operation -> asset = topInt.operation -> asset
-- 6 -- The bottom interface of a connector is determined by the components and
connectors attached to its top. This is similar to the constraint above.
```

以上程序中的实体类型使用了关联端的 multiplicity 属性，因为在 Association 和 AssociationEnd 之间的元级关联是有序的(ordered)，所以 AssociationEnd 是一个序列而不是一个集合。但对 UML 来说，这种有序没有语义表示。

4. C2 体系结构

C2 体系结构由构件和连接件组成，构件在 top 端可以与一个连接件关联，在 bottom 端与一个连接件关联，而一个连接件的 top 端或 bottom 端可以与任意数量的其他连接件的 top 端或 bottom 端关联。

```
Stereotype C2Architecture for instances of meta class Model
-- 1 -- The classes in a C2Architecture must all be C2 model elements.
Self.modelElement -> select(me | me.oclIsKingdof(Class)) -> forAll(c |
    c.stereotype = C2Compnent or c.stereotype = c2Connector)
-- 2 -- The associations in a C2Architecture must all be C2 model elements.
Self.modelElement -> select(me | me.oclIskindof(Association)) -> forAll(a |
    a.stereotype = C2AttachOverComp or
    a.stereotype = C2AttachUnderComp or
    a.stereotype = C2AttachConnConn)
```

```
-- 3 -- Each C2Component has at most one C2AttachOverComp.
let comps = self.modelElement -> select(me | me.stereotype = C2Component) in
comps -> forAll(c | c.associationEnd.association -> select(a | a.stereotype =
    C2AttachOverComp) -> size <= 1)
-- 4 -- Each C2Component has at most one C2AttachUnderComp. Similar to the
  constraint above.
-- 5—C2Connectors do not participater in any non - C2 associations.
let cons = self.modelElement -> select (me | me.stereotype = C2Connector) in
cons.associationEnd.association -> forAll(a | a.stereotype = C2AttachOverComp
or
a.stereotype = C2AttachUnderComp or
a.stereotype = C2AttachConnConn)
-- 6 -- C2Components do not participate in any non - C2 associations. Similar to
    the constraint above, but without the third disjunct.
-- 7 -- Each C2Connector must be attached to some connector or component.
let cons = self.modelElement -> select(e | e.stereotype = C2Connector) in
cons -> forAll(c | c.associationEnd -> size > 0)
-- 8 -- Each C2Component must be attached to some connector. Similar to the
constraint above.
```

以上程序的实体类型中 oclIsKingof 操作是所关联的模型实例中的元模型类的断言，当这个实例属于所指定的类或其子类时，断言为真。

思 考 题

1. 在某银行业务的用例模型中，“取款”用例需要等到“存款”用例执行之后才能执行，两个用例之间的关系什么？“取款”和“存款”两个用例中都需要执行查询余额的功能，将查询余额提取成独立的用例，那么“取款”和“存款”用例与“查询余额”用例之间的关系是什么？

2. 希赛图书订单处理系统中，“创建新订单”和“更新订单”两个用例都需要检查客户的账号是否正确，为此定义一个通用的用例“核查客户账户”。用例“创建新订单”和“更新订单”与用例“核查客户账户”之间是什么关系？

3. 如果一个并发的状态由 n 个并发的子状态图组成，那么，该并发状态在某时刻的状态由多少个子状态图中各取一个状态组合而成？

4. 用例从用户角度描述系统的行为。用例之间可以存在一定的关系。在希赛教育图书馆管理系统用例模型中，所有用户使用系统之前必须通过“身份验证”，“身份验证”可以有“密码验证”和“智能卡验证”两种方式，则“身份验证”与“密码验证”和“智能卡验证”之间是什么关系？

5. 软件体系结构可以通过 UML 直接进行描述，请说明 UML 包括哪些图，以及各自的作用是什么。

6. 在 UML 的众多图形中，最常用的图形是哪一个？在为同一个软件系统建模的过程中，是否需要将这些图形都要画出来？为什么？

7. 如何使用 4 层元模型描述体系结构？

8. 请选择一个小型的软件系统，并用 UML 为该系统建模。

参考文献

第6章 可扩展标记语言

软件体系结构描述语言 ADL 是一种形式化语言，它在底层语义模型的支持下，为软件的概念体系结构建模提供了具体语法和框架。ADL 强调的是概念体系结构，它通常要包含形式化的语义理论，并以此来刻画体系结构的底层框架。虽然这些特点使得用 ADL 描述软件体系结构时，具有精确、完全的特点，但也导致专业术语过多，语义理论过于复杂等问题，使其不利于向产业界推广。而 XML(eXtensible Markup Language，可扩展标记语言)被设计成简单和易于实现，并且由于 XML 在工业界广泛使用，因此，若能够用 XML 表示软件体系结构，必将能极大推广软件体系结构领域的研究成果在软件产业界的应用。

XML 是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。它也是元标记语言，用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。XML 与 HTML 一样，都来自 SGML(Standard Generalized Markup Language，标准通用标记语言)。

6.1 XML 概述

SGML 是一种用标记描述文档资料的通用语言，它包含一系列的 DTD(Document Type Definition，文档类型定义)，DTD 中定义了标记的含义，因而 SGML 的语法是可以扩展的。SGML 十分庞大，既不容易学，又不容易使用，在计算机上实现也十分困难。鉴于这些因素，Web 的发明者(欧洲核子物理研究中心的研究人员)根据当时(1989 年)计算机技术的能力，提出了 HTML。

HTML 只使用 SGML 中很少一部分标记，例如，HTML 3.2 定义了 70 种标记。为了便于在计算机上实现，HTML 规定的标记是固定的，即 HTML 语法是不可扩展的，它不需要包含 DTD。HTML 这种固定的语法使它易学易用，在计算机上开发 HTML 的浏览器也十分容易。正是由于 HTML 的简单性，使 Web 技术从计算机界走向全社会，走向千家万户，Web 的发展如日中天。

随着 Web 的应用越来越广泛和深入，人们渐渐觉得 HTML 不够用了，HTML 过于简单的语法严重地阻碍了用它来表现复杂的形式。尽管 HTML 推出了一个又一个新版本，已经有了脚本、表格、帧等表达功能，但始终满足不了不断增长的需求。另外，计算机技术的发展也十分迅速，已经可以实现比当初发明 HTML 时复杂得多的 Web 浏览器，所以开发一种新的 Web 页面语言既是必要的，也是可能的。

有人建议直接使用 SGML 作为 Web 语言，这固然能解决 HTML 遇到的困难。但是 SGML 太庞大了，用户学习和使用不方便尚且不说，要全面实现 SGML 的浏览器就非常困难。

于是，Web 标准化组织 W3C 就提出了简化 SGML、提供超链接能力、样式表、使用简单、文件解析等几个改进方案，并实现以下 10 个目标。

(1) 能够直接应用在 Internet 上：克服 SGML 没有支持 Web 的通信协议，因此，需要对其改进，使其支持 HTPP、URL、Script、Java 等，而且要尽量瘦身。

(2) 能被各式应用软件使用：可通过 XML 解析器来使用 XML 文件。

(3) 能与 SGML 兼容：SGML 已经使用多年，因此，保持兼容性能够最有效地利用原有资料。

(4) 能轻易发展 XML 相关软件：也就是需要有效降低开发 XML 应用软件的成本。

(5) 能简化 SGML：SGML 具有大量不常使用的可选项，XML 应尽可能减少类似情况。

(6) XML 文件可读性高：也就是 XML 文件应该具有良好的结构性。

(7) XML 规范能尽快完成：HTML 的瓶颈效应越来越明显，因此，应该尽可能地快速提出 XML。

(8) XML 规范必须简洁：也就是降低学习时间和成本。

(9) XML 文件易于建立：允许 XML 文件中不一定带有 DTD 文档。

(10) 语法不可模糊不清：克服 HTML 中的相关问题。

6.1.1 XML 的特点

根据 W3C 提出的方案，XML 应运而生，与传统的 SGML 或 HTML 相比，XML 具有以下特点。

1. 简洁有效

XML 是一个精简的 SGML，它将 SGML 的丰富功能与 HTML 的易用性结合到 Web 应用中。XML 保留了 SGML 的可扩展功能，这使 XML 从根本上有别于 HTML。XML 要比 HTML 强大得多，它不再是固定的标记，而是允许定义数量不限的标记来描述文档中的资料，允许嵌套的信息结构。HTML 只是 Web 显示数据的通用方法，而 XML 提供了一个直接处理 Web 数据的通用方法。HTML 着重描述 Web 页面的显示格式，而 XML 着重描述的是 Web 页面的内容。

XML 中包括 XSL(eXtensible Style Language，可扩展样式语言)和 XLL(eXtensible Linking Language，可扩展链接语言)。XSL 用于将 XML 数据翻译为 HTML 或其他格式的语言。XSL 提供了一种叠式页面 CSS(Cascading Style Sheets)的功能，使开发者可以构造出具有表达层结构的 Web 页面，以有别于 XML 的数据结构。XSL 也能和 HTML 一起构造叠式页面。XSL 可以解释数量不限的标记，它使 Web 的版面更丰富多彩，如动态的文本、跑马式的文字等。此外，XSL 还处理多国文字、双字节的汉字显示、网格的各种各样的处理等；XLL 是 XML 的链接语言，它与 HTML 的链接相似，但功能更强大。XLL 支持可扩展的链接和多方向的链接，它打破了 HTML 只支持超级文本概念下最简单的链接限制，能支持独立于地址的域名、双向链路、环路、多个源的集合链接等。XLL 链接可不受文档制约，完全按用户要求来指定和管理。

2. 易学易用

为了使 XML 易学易用，XML 精简了许多 SGML 功能。SGML 中常用的部分只占

20%,XML 抛弃了 SGML 中不常用的部分,使它精简了 80%。这样,XML 的语法说明书只有 30 页,而 SGML 的语法说明书却有 500 页。

XML 设计中也考虑了它的易用性,易用性来自两个方面:一方面用户编写 Web 页面方便,另一方面设计人员实现 XML 浏览器也不太困难。

3. 开放的国际化标准

XML 使 SGML 在市场上有许多成熟的软件可用来帮助编写、管理等,XML 通过验证的标准技术,使得其具有高度的开放性。众多业界顶尖公司,与 W3C 的工作组并肩合作,协助确保交互作业性,支持各种系统和浏览器上的开发人员、作者和使用者,以改进 XML 标准。其中,包括如下标准。

(1) XML 标准。这是 W3C 正式批准的,这意味着这个标准是稳定的,完全可用于 Web 和工具的开发。

(2) XML 名域标准。用来描述名域的句法,支持能识别名域的 XML 解析器。

(3) DOM(Document Object Model,文档对象模型)标准。为给结构化的数据编写脚本提供标准,这样,开发人员就能够与计算机在基于 XML 的数据上进行交互。

(4) XSL 标准。XSL 有两个模块:XSL 转换语言和 XSL 格式化对象。其中,转换语言可用来转换 XML 以满足显示要求。由于 XSL 的两部分是模块化的,因此,转换语言能够独立地用来进行多用途的转换,包括把 XML 转换成结构完整的 HTML。

(5) XLL 标准和 XML 指针语言(XPointer)标准。XLL 提供类似于 HTML 的链接,但功能更强大。例如,链接可以是多方向的,可以存在于对象上而不仅是页面上。

提示:XML 通过采用一个新的编码标准,可以支持世界上大多数文字。因此,XML 不仅能在不同的计算机系统之间交换信息,而且能跨国界和超越不同文化疆界交换信息。

4. 高效且可扩充

XML 支持复用文档片断,使用者可以发明和使用自己的标签,也可与他人共享,可延伸性大。在 XML 中,可以定义无限量的一组标注。XML 提供了一个表示结构化资料的体系结构。一个 XML 构件可以宣告与其相关的资料为零售价、营业税、书名、数量或其他任何数据元素。随着世界范围内的许多机构逐渐采用 XML 标准,将会有更多的相关功能出现。XML 提供了一个独立的运用程序的方法共享数据。使用 DTD,不同组中的人就能够使用共同的 DTD 交换数据。应用程序可以使用 DTD 验证用户接收到的数据是否有效,也可以使用一个 DTD 验证用户自己的数据。

总之,XML 使用一个简单而又灵活的标准格式,为基于 Web 的应用提供了一个描述数据和交换数据的有效手段。HTML 描述了显示全球数据的通用方法,而 XML 提供了直接处理全球数据的通用方法。

6.1.2 XML 的作用

XML 可以让信息提供者根据需要,自行定义标记及属性名,结构化地描述信息内容,因此赋予了应用软件强大的灵活性,为开发者和用户带来许多好处。

1. 使得搜索更加有意义

在非 XML 构建的 Web 系统中,搜索软件必须了解每个数据库是如何构建的。这实际上是不可能的,因为每个数据库描述数据都是不同的。而 XML 是一个自描述的语言,它可

以使得数据“知道”自身的信息，从而可以进行更有使用价值的搜索功能。

2. 开发灵活的 Web 应用软件

XML 的应用将使得三层 Web 应用软件的开发更加简单，由于其能够有效地进行消息与数据标准的统一，从而设计、开发出更加灵活的 Web 应用软件。

3. 实现不同数据的集成

不同的数据库系统，其存储结构、应用程序接口都存在着许多不同点，因此基本上无法开发出一套能够针对这些相互不兼容的数据库的查询程序。而 XML 的出现，则改变了这个现象，由于数据是结构化的，因此，即使它们的来源不同，也能够很容易地结合在一起。在开发时，可以在中间层的服务器上对从后端数据库和其他应用系统来的数据进行集成。然后，数据就能被发送到客户或其他服务器做进一步的集合、处理和分发。

4. 使用于多种应用环境

XML 的高扩展性、高灵活性特性，使得其可以描述各种不同种类的应用软件中的各种不同类型的数据。另外，XML 具有自描述性，可以很容易进行交换、处理，而且还不需要多余的内部描述。

5. 客户端数据处理与计算

由于 XML 格式的标准化，许多浏览器软件都能够提供很好的支持。因此，只需要简单地将 XML 格式的数据发送给客户端，客户端就可以自行对其进行编辑和处理，而不仅是显示。而且，XML 的 DOM 还允许客户端利用脚本或其他编程语言处理数据，而无须回到服务器端。这种将数据视图与内容分离的机制，可以更容易地创建出基于 Web 的、功能强大的应用，而无须基于高端数据库。

6. 数据显示多样化

XML 将显示和数据内容分离，提供了一种简单、开放、扩展的方式来描述结构化数据。与 HTML 不同的是，HTML 描述了数据的外观，而 XML 则描述的是数据本身。因此，XML 定义的数据可以指定不同的显示方式，利用 CSS 或 XSL 等工具来提供显示机制。

7. 局部数据更新

通过 XML，数据可以实现局部的更新。也就是说，当有其中的一部分数据变化时，并不需要重发整个结构化的数据，服务器只需将变化的元素发送给客户。而不是只要一条数据变化了，整个页面都必须重建。而且，还可以将新增加的信息加入已存在的页面，这样，就可以使得应用具有更高的性能。

8. 与现有 Web 发布机制相兼容

由于 XML 也是一个开放的基于文本的格式，可以像 HTML 一样使用 HTTP 进行传送，不需要对现存的网络进行变化，可以很方便地应用于 Web 上的数据发布。

9. 可升级性

由于 XML 彻底把标识的概念与显示分开，处理者能够在结构化的数据中嵌套程序化的描述以表明如何显示数据。这是一个强大的机制，使得客户计算机与使用者间的交互作用尽可能的减少了。同时，也减少了服务器的数据交换量和浏览器的响应时间。另外，XML 减少了服务器的工作量，大大增强了服务器的升级性能。

10. 压缩性能高

XML压缩性能很好,因为用于描述数据结构的标签可以重复使用。XML数据是否压缩要根据应用来定,还取决于服务器与客户间数据的传递量。XML能够使用HTTP 1.1中的压缩标准。

6.1.3 XML的应用

XML的种种优点可以在以下几种应用情况下发挥巨大作用。

1. 应用于客户需要与不同的数据源进行交互时

数据可能来自不同的数据库,它们都有各自不同的复杂格式。但客户与这些数据库间只通过一种标准语言进行交互,那就是XML。由于XML的自定义性及可扩展性,它足以表达各种类型的数据。客户收到数据后可以进行处理,也可以在不同数据库间进行传递。也就是说,在这种情况下,XML解决了数据的统一接口问题。

2. 应用于将大量运算负荷分布在客户端

客户可根据自己的需求选择和制作不同的应用程序以处理数据,而服务器只需要发出同一个XML文件。如按传统的客户/服务器工作方式,客户向服务器发出不同的请求,服务器分别予以响应,这不仅加重了服务器本身的负荷,而且网络管理者还须事先调查各种不同的用户需求以做出相应不同的程序,但假如用户的需求繁杂而多变,则仍然将所有业务逻辑集中在服务器端是不合适的,因为服务器端的编程人员可能来不及满足众多的应用需求,也来不及跟上需求的变化,双方都很被动。应用XML则将处理数据的主动权交给了客户,服务器只是尽可能完善、准确地将数据封装进XML文件中,正是各取所需、各司其职。XML的自解释性使客户端在收到数据的同时也能理解数据的逻辑结构与含义,从而使广泛、通用的分布式计算成为可能。

3. 应用于将同一数据以不同的面貌展现给不同的用户

XML类似于同一个剧本,却可以用电视剧、电影、话剧、动画片等不同形式表现出来。这一应用将会为网络用户界面个性化、风格化的发展铺平道路。

4. 应用于网络代理对所取得的信息进行编辑、增减以适应个人用户的需要

有些客户取得数据并不是为了直接使用而是为了根据需要组织自己的数据库。例如,希赛教育网站上建立了一个庞大的题库,实际考试时,随机将题库中的若干题目取出组成试卷,再将试卷封装进XML文件。接下来便是最精彩的部分,在各个考点让其通过一个过滤器,滤掉所有的答案,再发送到考生面前,未经过滤的内容则可直接送到老师手中。此外,XML文件中还可以包含诸如难度系数、往年错误率等其他相关信息,这样,只需几个小程序和一个XML文件,便可变成多个文件传送到不同的用户手中。

6.2 解析XML

XML使用的是非常简单的数据格式,可以用100%的纯ASCII(American Standard Code for Information Interchange,美国信息交换标准代码)文本来书写,也可以用几种其他定义好的格式来书写。ASCII文本几乎不会被“磨损”。丢失一些字节甚至是相当多的字节,剩下的数据还是可以读取的。这就与许多格式形成了鲜明的对比。例如,压缩数据或是

串行的 Java 对象，这些数据即使丢失一个字节，剩余的数据也变得不可读取了。

更重要的是，XML 是自描述的。假设在 25 世纪有一个信息考古学者，他在某个地方发现了如下一段经过时间的“冲刷”保存下来的 XML 代码。

```
<PERSON ID = "p1100" SEX = "M">
    <NAME>
        <GIVEN> Judson </GIVEN>
        <SURNAME> McDaniel </SURNAME>
    </NAME>
    <BIRTH>
        <DATE> 2 Feb 1934 </DATE>
     </BIRTH>
    <DEATH>
        <DATE> 9 Dec 2005 </DATE>
    </DEATH>
</PERSON>
```

即使这个考古学家不熟悉 XML，但假设他可以讲 21 世纪时的英语，那么就可以很好地了解名为 Judson McDaniel 的人，此人出生在 1934 年 2 月 2 日，死于 2005 年 12 月 9 日。事实上，数据中有一些空白或损坏，还是可以得到这些信息。但对于专有格式的电子表格或是字处理程序的格式，就不是这么回事了。

更进一步说，XML 有很好的规格文档。W3C 的 XML 1.0 规范和大量的论文书籍，如本书，都向人们准确地说明了如何阅读 XML 数据，没有什么秘密使得人们发生失误。

6.2.1 XML 与 HTML 的区别

虽然 XML 与 HTML 都是标记语言，但它们在结构和应用上有很大的区别。

HTML 是一种格式化的语言，一个 HTML 文本可以看作一个格式化的程序。HTML（及类似的语言）定义了一套固定的标记，用来描述一定数目的元素。如果标记语言中没有所需的标记，用户也就没有办法了。这时只好等待标记语言的下一个版本，希望在新版本中能够包括所需的标记。另外，用 HTML 描述的程序或文本具有“内容＋格式”的双重属性。一个 HTML 程序在不同平台、不同浏览器上的表现一模一样。而一段符合 XML 语法规范的文本则是一段“纯”数据，它的结构由其他的称为 DTD 的文本来描述，而它的处理则可能是任何其他支持 XML 的容器或程序。例如，IE 浏览器依据相关的 CSS 或 XSL 文件来显示 XML 数据。开发人员可以用任何支持 XML 的开发工具开发自己的 XML 处理程序。

与 HTML 相比的另一个不同点是，XML 是一种元标记语言。它可以被用于定义其他的标记语言，甚至 DTD 和 XSL 文档也是用 XML 语法描述的。例如，在 Peter Murray-Rust 的化学标记语言（Chemical Markup Language，CML）中的 MOL. DTD 文件中描述了词汇表和分子科学的句法，其中包括 chemistry（化学）、crystallography（结晶学）、solid state physics（固体物理）等词汇。它包括用于 atoms（原子）、molecules（分子）、bonds（化学键）、spectra（光谱）等的标记。这个 DTD 可与分子科学领域中的许多不同的人共享。对于其他领域也有类似的 DTD，用户还可以创建自己的 DTD。

XML 定义了一套元句法，与特定领域有关的标记语言（如 MusicML、MathML、CML

等)都必须遵守。如果一个应用程序可以理解这一元句法,那么它也就自动地能够理解所有的由此元语言建立起来的语言。浏览器不必事先了解多种不同的标记语言使用的每个标记。事实是,浏览器在读入文档或是它的DTD时才了解给定文档使用的标记。

有了XML,就意味着不必等待浏览器的开发商来满足用户的需要了。用户可以创建自己需要的标记,当需要时,告诉浏览器如何显示这些标记即可。

6.2.2 XML文档

一个实用的XML文档必须满足两点,分别是组织良好(Well-formed)和有效(Valid)。下面分别举例说明。

下例试图描述一个欢迎词。

```
<?xml version = "1.0"?>
<visit>
        <to>alluser</to>
        <from>educity.cn</from>
        <heading>welcome</heading>
        <body>Welcome to the best online education website!</body>
</visit>
```

这就是一份可以接受的XML文档。由于使用了一些人们易于理解的标记如<from>、<body>等,令人觉得这些数据十分易读和有意义。

这也是一个组织良好的XML文档,它满足以下三项基本规则。

(1) 文档以XML定义<? xml version="1.0"? >开始。

(2) 有一个包含所有其他内容的根元素,如上面例子中的<visit>和</visit>标记符。

(3) 所有元素必须合理地嵌套,不允许交叉嵌套。

组织良好的XML可以对应为一棵逻辑上的树。没有组织好的文档在HTML中可能不算什么,因为浏览器已经被设计成可以处理这种问题。但是,在XML中却是致命的,因为应用程序将拒绝处理没有组织好的文件。

XML文档单是组织良好是不够的。例如,如下的"欢迎词"数据。

```
<?xml version = "1.0"?>
<visit>
        <to>alluser</to>
        <from>educity.cn</from>
        <heading>Welcome</heading>
</visit>
```

这段数据是组织良好的,但却不会产生什么实用价值,因为它没有关键的内容(<body>)。因此,还必须要有一个文档对XML数据文档的内容做出规定,这种文档就是DTD。例如,关于visit的DTD可以这么写:

```
<?xml version = "1.0"?>
<!DOCTYPE visit [
        <!ELEMENT visit (to,from,heading,body)>
        <!ELEMENT to (#PCDATA)>
```

```
        <!ELEMENT from (#PCDATA)>
        <!ELEMENT heading (#PCDATA)>
        <!ELEMENT body (#PCDATA)>
]>
```

! DOCTYPE visit 第二行的结点是 XML 文档中的 visit 类型；第三行说明元素 visit 有 4 个元素 to、from、heading 和 body；第四行定义了 to 元素的类型为 #PCDATA；第五行定义了 from 元素的类型为 #PCDATA，等等。

DTD 可以看作对 XML 数据的语法结构的描述。使用 DTD，不同的人就能够使用共同的 DTD 来交换数据。有专门的小程序对 XML 文档是否组织良好，以及是否符合 DTD 定义的有效性进行扫描、判断和报告。

除 DTD 外，还有一种方式来定义有效的 XML 文档，那就是 W3C 与 XML 标准一起定义的 XML 模式(XML Schema)。与 DTD 相比，XML 模式有如下几个优势。

(1) XML 模式使用 XML 语法。换句话说，XML 模式是一个 XML 文档。这意味着可以像处理任何其他文档一样处理模式。例如，可以编写一个 XSLT(eXtensible Stylesheet Language for Transformation，用于转换的可扩展样式表语言)样式表，该样式表将 XML 模式转换成具有自动生成的 JavaScript 代码的 Web 表单，其中的 JavaScript 代码可以验证输入的数据。

(2) XML 模式支持数据类型。尽管 DTD 也支持数据类型，但很明显，这些数据类型是从发布的角度开发的。XML 模式支持 DTD 中的所有原始数据类型(诸如标识和标识引用之类的类型)。它们还支持整数、浮点数、日期、时间、字符串、URL 和其他对数据处理和验证有用的数据类型。

(3) XML 模式是可扩展的。除了 XML 模式规范中定义的数据类型以外，还可以创建自己的数据类型，并且可以基于其他数据类型派生出新的数据类型。

(4) XML 模式有更强的表达能力。例如，下面的例子是关于邮政地址的 DTD 描述相匹配的 XML 模式。在 XML 模式定义中，它增加了两个约束：<state>元素的值必须刚好是两个字符长，<postal-code> 元素的值必须与正则表达式 [0-9]{5}(-[0-9]{4})? 相匹配。用 DTD 无法做这些事，尽管这个模式比 DTD 长很多，但它更清楚地表达了有效的文档看起来是什么样子。

```
<!-- address.dtd -->
<!ELEMENT address (name, street, city, state, postal-code)>
<!ELEMENT name (title? first-name, last-name)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postal-code (#PCDATA)>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<xsd:element name = "address">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref = "name"/>
      <xsd:element ref = "street"/>
      <xsd:element ref = "city"/>
      <xsd:element ref = "state"/>
      <xsd:element ref = "postal - code"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "name">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref = "title" minOccurs = "0"/>
      <xsd:element ref = "first - Name"/>
      <xsd:element ref = "last - Name"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "title"          type = "xsd:string"/>
<xsd:element name = "first - Name"   type = "xsd:string"/>
<xsd:element name = "last - Name"    type = "xsd:string"/>
<xsd:element name = "street"         type = "xsd:string"/>
<xsd:element name = "city"           type = "xsd:string"/>

<xsd:element name = "state">
  <xsd:simpleType>
    <xsd:restriction base = "xsd:string">
      <xsd:length value = "2"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>

<xsd:element name = "postal - code">
  <xsd:simpleType>
    <xsd:restriction base = "xsd:string">
      <xsd:pattern value = "[0 - 9]{5}( - [0 - 9]{4})?"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
</xsd:schema>
```

6.2.3 CSS 与 XSL

XML 文档最终是要通过一定的应用程序来表现的，如浏览器。用浏览器来表现 XML 就要经过 CSS 或 XSL 这一媒介，它们都是样式语言，描述了 XML 数据与 HTML 标记的映

射关系。或者可以说，XSL 是对 XML 文档进行排版的语言。

1. CSS

CSS 之于 HTML 文档的作用类似于 MS Word 中的“样式”的作用，可以在某种程度上把 HTML 文档中的排版格式信息与其他数据信息分离开。

一个 CSS 样式单就是一组规则(rule)，样式再根据特定的一套规则级联起来。每个规则给出此规则所适用的元素的名称，以及此规则要应用于哪些元素的样式。例如，下列清单是一首诗的 CSS 样式单。

```
POEM { display: block }
TITLE { display: block; font - size: 16pt; font - weight: bold }
POET { display: block; margin - bottom: 10px }
STANZA { display: block; margin - bottom: 10px }
VERSE { display: block }
```

此样式单有 5 个规则。每个规则有一个选择符(规则所应用的元素的名称)和一组适用于此元素实例的属性。第一个规则说明 POEM 元素应以块的形式(display：block)显示其内容。第二个规则说明 TITLE 元素应以 16 磅(font-size：16pt)、粗体(font-weight：bold)将其内容显示在块中(display：block)。第三个规则说明 POET 元素应通过自身显示在块中(display：block)，并且与紧随其后的下一块相距 10 个像素(margin-bottom：10px)。第四个规则与第三个相同，所不同的只是前者应用于 STANZA 元素。最后，第五个规则只简单地说明 VERSE 元素也是显示在自己的块中。

定义了如上的 CSS 以后，HTML 文档就可以用 POEM、TITLE 来说明真正的数据了。

CSS 是专为 HTML 设计的，HTML 的标志必须和 CSS 标志兼容。例如，要正确地支持 CSS 的 nowrap 属性就要求废除 HTML 中非标准的但又是经常使用的 NOWRAP 元素。而 XML 元素没有任何预定义的格式规定，它甚至不限于 HTML，所以，利用 XML 可以开发更强大的方式来解决 HTML 的样式与内容分离的问题，那就是 XSL。虽然也可以用 XML 来写 CSS，但是已经很少有人那样做了。

2. XSL

XSL 是专门用于 XML 文档的样式单语言，可以把 XSL 当成一种能够把 XML 转变成 HTML 的语言，一种能够筛选和排序 XML 文档中数据的语言，一种能够根据 XML 的数据数值格式化 XML 数据的语言(例如，把负数显示成红色)。XSL 文档本身就是结构完整的 XML 文档。

以下是一个应用 XSL 的例子。假设有一个 XML 数据文档：

```
portfolio.xml
<?xml version = "1.0"?>
<portfolio>
    <stock exchange = "nyse">
      <name> zacx corp </name>
      <symbol> ZCXM </symbol>
      <price> 28.875 </price>
```

```
  </stock>
  <stock exchange = "nasdaq">
    <name>zaffymat inc</name>
    <symbol>ZFFX</symbol>
    <price>92.250</price>
  </stock>
</portfolio>
```

为了在浏览器中显示这个 XML 文档,使用如下 XSL 文件。

```
portfolio.xsl
<?xml version = '1.0'?>
<xsl:stylesheet xmlns:xsl = "http://www.w3.org/TR/WD - xsl">
<xsl:template match = "/">
  <html>
  <body>
  <table border = "2" bgcolor = "yellow">
      <tr>
        <th>Symbol</th>
        <th>Name</th>
        <th>Price</th>
      </tr>
      <xsl:for - each select = "portfolio/stock">
        <tr>
          <td><xsl:value - of select = "symbol"/></td>
          <td><xsl:value - of select = "name"/></td>
          <td><xsl:value - of select = "price"/></td>
        </tr>
      </xsl:for - each>
  </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

最后,在原来的 XML 文档第二行前加入对 XSL 的引用:

```
<?xml - stylesheet type = "text/xsl" href = "portfolio.xsl"?>
```

就可以实现在一个 HTML 表格中显示股票交易数据了。

3. CSS 与 XSL 的比较

CSS 只能改变特定元素的格式,也只能以元素为基础。但 XSL 样式单可以重新排列元素并对元素进行重排序。这种样式单可以隐藏一些元素而显示另外一些元素。更进一步说,还可以选择应用样式的标记,而不仅是基于标记的,而且,XSL 还基于标记的内容和特性,基于标记在文档中相对于其他元素的位置,以及基于各种其他的准则。

CSS 的优越性在于具有广泛的浏览器支持。但是 XSL 更为灵活和强大,可更好地适用于 XML 文档。而且,带 XSL 样式单的 XML 文档可以很容易地转换为带 CSS 样式单的 HTML 文档。

如果只是要对一些固定数据进行排版，可以使用 HTML + CSS 方式；如果这些数据是与某些应用程序相关，并且独立于程序存在的，并且要独立于程序来使用，则应该充分使用 XML 技术，采用 HTML + XML + XSL。

6.3 XML 编程接口

本节将介绍 XML 的多种编程接口，这些接口为开发人员使用 XML 文档提供了一致的接口。

6.3.1 API

有许多 API 可以使用，本节介绍最流行和广泛使用的 API 中的 4 种，分别是 DOM、SAX(Simple API for XML，用于 XML 的简单 API)、JDOM 和 JAXP(Java API for XML Parsing，用于 XML 解析的 Java API)。

1. DOM

文档对象模型为 XML 文档的已解析版本定义了一组接口。解析器读入整个文档，然后构建一个驻留内存的树结构，然后代码就可以使用 DOM 接口来操作这个树结构。用户可以遍历树以了解原始文档包含什么，可以删除树的几个部分，还可以重新排列树和添加新的分支等。

DOM 提供了一组丰富的功能，用户可以用这些功能解释和操作 XML 文档，但使用它们是有代价的。在开发用于 XML 文档的原始 DOM 时，很多人也提出了 DOM 的几个问题。

(1) DOM 构建整个文档驻留内存的树。如果文档很大，就会要求有极大的内存。

(2) DOM 创建表示原始文档中每个东西的对象，包括元素、文本、属性和空格。如果用户只关注原始文档的一小部分，那么创建那些永远不被使用的对象是极其浪费的。

(3) DOM 解析器必须在代码取得控制权之前读取整个文档。对于非常大的文档，这会引起显著的延迟。

这些仅仅是由 DOM 的设计引起的问题，撇开这些问题，DOM API 是解析 XML 文档非常有用的方法。

2. SAX

为了解决 DOM 问题，就产生了 SAX 接口。SAX 的几个特征解决了 DOM 的问题。

(1) SAX 解析器向代码发送事件。当解析器发现元素开始、元素结束、文本、文档的开始或结束时，它会告诉用户。用户可以决定什么事件对自己重要，而且可以决定要创建什么类型的数据结构以保存来自这些事件的数据。如果没有显式地保存来自某个事件的数据，它就被丢弃。

(2) SAX 解析器根本不创建任何对象，它只是将事件传递给应用程序。如果希望基于哪些事件创建对象，这将由编程者自己来完成。

(3) SAX 解析器在解析开始的时候就开始发送事件。当解析器发现文档开始、元素开始和文本时，代码会收到一个事件。应用程序可以立即开始生成结果；不必一直等到整个文档被解析完毕。更妙的是，如果只查找文档中某些内容，代码一旦找到所要找的东西就可以

抛出一个异常。该异常会停止SAX解析器,然后代码用它找到的数据做它需要做的任何事。

SAX解析器也有些问题引人关注,如下所述。

(1) SAX事件是无状态的。当SAX解析器在XML文档中发现文本时,它就向代码发送一个事件。该事件仅给用户发现的文本,它不告诉用户什么元素包含那个文本。如果想知道这一点,则用户必须自己编写状态管理代码。

(2) SAX事件不是持久的。如果应用程序需要一个数据结构来对XML文档建模,则必须自己编写那样的代码。如果需要从SAX事件访问数据,并且没有把那个数据存储在代码中,那么用户不得不再次解析该文档。

3. JDOM

用DOM和SAX模型完成某些任务时的困难使Jason Hunter和Brett McLaughlin感到失望,于是他们创建了JDOM包。JDOM是基于Java技术的开放源码项目,它试图遵循80/20规则:用DOM和SAX的20%的功能来满足80%的用户需求。JDOM使用SAX和DOM解析器,因此,它是作为一组相对较小的Java类被实现的。

JDOM的主要特性是它极大地减少了用户必须编写的代码数量,JDOM应用程序的长度通常是DOM应用程序的1/3,大约是SAX应用程序的一半。JDOM并不做所有的事,但对于大多数用户要做的解析,它可能正好适合用户的需求。

4. JAXP

尽管DOM、SAX和JDOM为大多数常见任务提供了标准接口,但仍有些事情是它们不能解决的。例如,在Java程序中创建DOMParser对象的过程因DOM解析器的不同而不同。为了修正这个问题,Sun发布了JAXP,该API为使用DOM、SAX和XSLT处理XML文档提供了公共接口。

JAXP提供的诸如DocumentBuilderFactory和DocumentBuilder之类的接口,为不同的解析器提供了一个标准接口。还有一些方法可以允许用户控制底层的解析器是否可以识别名称空间,以及是否使用DTD或模式验证XML文档。

5. 接口的选择

在实际应用中,为了选择合适的接口类型,需要理解所有接口的设计要点,而且需要理解应用程序用XML档做什么。考虑下面的问题将有助于找到正确的方法。

(1) 要用Java编写应用程序吗?JAXP使用DOM、SAX和JDOM;如果用Java编写代码,那么应使用JAXP将代码与各种解析器实现的细节隔离。

(2) 应用程序将如何部署?如果应用程序将要作为Java applet部署,那么会希望使要下载的代码数量最小,SAX解析器比DOM解析器小,而使用JDOM时,除了SAX或DOM解析器之外还要求编写少量的代码。

(3) 一旦解析了XML文档,还需要多次访问那些数据吗?如果需要回过头来访问XML文件的已解析版本,DOM可能是正确的选择。而SAX事件被触发时,如果以后需要它,则由(开发人员)自己决定以某种方式保存它。如果需要访问不曾保存的事件,则必须再次解析该文件;而DOM自动保存所有的数据。

(4) 只需要XML源文件的少量内容吗?如果只需要XML源文件的少量内容,那么SAX可能是正确的选择。SAX不会为源文件中的每个东西创建对象。使用SAX,要检查每个事件以了解它是否与需要有关,然后相应地处理它。

(5) 正在一台内存很少的机器上工作吗？若是，不管可能考虑到的其他因素是什么，SAX 都是最佳选择。

6.3.2 XML 开发工具

支持 XML 的开发工具比较多，开发语言有脚本语言（如 JavaScript、VBScript、Perl 等）、编程语言（如 Java、C++、Object Pascal 等），开发环境有命令行工具（如 ANT 等）、集成式开发平台（如 Visual Studio .NET、VisualAge、Inprise JBuilder 等），建模工具有 IBM Rose 等。

下面是一个用脚本语言 JavaScript 处理 DOM 的例子，用于在 HTML 中简单地显示前面提到过的 visit 数据。

```
<?xml version = "1.0"?>
<visit>
        <to>alluser</to>
        <from>educity.cn</from>
        <heading>Welcome</heading>
        <body>Welcome to the best online education website!</body>
</visit>
```

visit.htm 的内容如下。

```
<html>
<head>
<script language = "JavaScript" for = "window" event = "onload">
        var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
        xmlDoc.async = "false";
        xmlDoc.load("visit.xml");
        nodes = xmlDoc.documentElement.childNodes;
        to.innerText = nodes.item(0).text;
        from.innerText = nodes.item(1).text;
        header.innerText = nodes.item(2).text;
        body.innerText = nodes.item(3).text;
</script>
<title>HTML using XML data</title>
</head>
<body bgcolor = "yellow">
        <h1>Refsnes Data Internal Note</h1>
        <b>To:</b>
        <span id = "to"></span>
        <br>
        <b>From:</b>
        <span id = "from"></span>
        <hr>
        <b><span id = "header"></span></b>
        <hr>
        <span id = "body"></span>
</body>
</html>
```

6.3.3 XML建模

与XML开发相关,而且可能比开发更重要的工作,是XML建模。XML技术的出现,使用户对信息的描述能力增强了。它使得用户得以设计不会随着关键数据变化而要修改代码的程序,提高系统的可维护性和适应性,构造与平台无关的数据,构造可以在不同系统中使用和交换的数据,定义在不同的部门共同遵守的信息标准。但要享受XML带来的惊喜,就必须很好地掌握怎样用XML描述信息,也就是XML建模。

XML并不是程序设计层面的技术。用户使用XML建模的能力越强,它给用户带来的回报就越多。XML建模包括以下方面。

(1) 描述具体数据:这是XML的经典用途,可以用任何文本编辑工具编写XML文档。

(2) 描述数据结构和模式:这需要用到DTD。

(3) 描述数据的表现:XSL定义了一组元素(称为格式化对象),它们描述应该如何格式化数据;XSLT是一个描述如何将XML文档转换成别的东西的XML词汇表。

(4) 描述数据中的位置:XPath(XML Path Language,XML路径语言)是描述XML文档中位置的语法。使用XSLT样式表中的XPath描述用户希望转换XML文档的哪个部分。XPath也用在其他XML标准中,这就是为什么它是独立于XSLT的标准的原因。

(5) 描述数据中的链接关系:XLink(XML Linking Language,XML链接语言)定义将不同资源链接在一起的各种方法。用户可以进行正常的点对点链接(就像用HTML <a> 元素)或扩展的链接,后者可包括多点链接、通过第三方的链接以及定义转向给定链接的意义的规则。XPointer使用XPath作为引用其他资源的方法,它还包括对XPath的一些扩展。

(6) 描述数据的应用关系:通过SOAP、WSDL、UDDI来描述,有关这些概念的详细内容,将在第8章中进行介绍。

XML技术为信息规划和信息系统分析人员扩展了视野,但独立的、简单易用的XML建模工具并不多见,这些工具大多嵌入在其他强大的建模工具或是开发平台中。

6.4 基于XML的软件体系结构描述语言

在软件体系结构的研究领域中,在体系结构描述上已经取得了很大的成就,出现了多种软件体系结构描述语言。然而,很难从这些语言中挑选出一个比较通用的"超级"ADL,适用于软件工程的所有项目和各个领域。

虽然人们自然地将XML文件与传统的文档联系在一起,但是XML的弹性使其有能力去描述非文档信息模型,其中一个重要的应用就是作为ADL。另外,因为XML标准已经迅速且广泛地在全球展开,许多大公司纷纷表示要对XML进行支持,所以XML更加有能力和潜力消除各种ADL无法统一的局面,使现在及将来的应用可以操作、查找、表现、存储这些XML模型,并且在软件环境和软件工具经常变换的情况下仍然保持可用性和重用性。

由于XML在体系结构描述上的许多优点,学者们已经开发出了不同的基于XML的体系结构描述语言,如XADL 2.0、XBA、ABC/ADL等。本节主要介绍XADL 2.0和XBA。

6.4.1 XADL 2.0

XADL 2.0 是一个具有高可扩展性的软件体系结构描述语言，它通常用于描述体系结构的不同方面。XADL 2.0 和其他的 ADLs(如 Rapid、Wright 等)一样，在 XADL 2.0 中，对体系结构的描述主要由 4 个方面组成，分别是构件、连接件、接口和配置。

XADL 2.0 在所有的 ADLs 中具有很多独特的性质。首先，XADL 2.0 在结构上具有很好的扩展性；其次，XADL 2.0 作为一个模型化的语言而建立，它并不是为了描述某一模型而建立的单一语言，而是一个对模型描述的集合；XADL 2.0 能够随着模型的增加或者模型的扩展而发展成模型集。

XADL 2.0 的所有模型是通过 XML Schema 来实现的，使得 XADL 2.0 作为一个基于 XML 的语言。所有 XADL 2.0 的文档(如体系结构描述)是有效的、遵循 XADL 2.0 模式的 XML 文档。另外，XADL 2.0 也能够被最终用户扩展或优化为部分领域的最优化 ADLs。下面是一个用 XADL 2.0 描述构件的例子。

```
<types:component xsi:type = "types:Component" types:id = "xArchADT">
  <types:description xsi:type = "instance:Description"> xArchADT </types:description>
  <types:interface xsi:type = "types:Interface" types:id = "xArchADT.IFACE_TOP">
    <types:description xsi:type = "instance:Description"> xArchADT Top Interface </types:
description>
    <types:direction xsi:type = "instance:Direction"> inout </types:direction>
    <types:type xsi:type = "instance:XMLLink" xlink:type = "simple" xlink:href = "#
C2TopType" />
  </types:interface>
  <types:interface xsi:type = "types:Interface" types:id = "xArchADT.IFACE_BOTTOM">
    <types:description xsi:type = "instance:Description"> xArchADT Bottom Interface </types:
description>
    <types:direction xsi:type = "instance:Direction"> inout </types:direction>
    <types:type xsi:type = "instance:XMLLink" xlink:type = "simple" xlink:href = "#
C2BottomType" />
  </types:interface>
  <types:type xsi:type = "instan0063e:XMLLink" xlink:type = "simple" xlink:href = "#xArchADT_type" />
</types:component>
```

1. XADL 2.0 的核心

XADL 2.0 模式的核心是实例模式，在 instances.xsd 文件中定义了实例模式，该模式是由 UC Irvine 和卡耐基梅隆大学合作创建的。它定义了体系结构的构成的“最小公分母”和语义的中立者，因此，它不用逐步定义这些构成，以及行为的约束和规则。

在实例模式结构中定义了 5 个方面的内容，分别是构件实例(包括子体系结构)、连接件实例(包括子体系结构)、接口实例、链接实例和通用组。所有这些内容都归类在一个称为 ArchInstance 的顶层元素下。每一个 ArchInstance 元素对应一个概念上的体系结构。这些元素的 XML 关系如图 6-1 所示。

在 XADL 2.0 中，实例模式的元素通常用于定义上述实例模式内容的运行时实例，它为运行时软件系统的部件提供了一个描述方法，相当于一个系统设计。

在 XADL 2.0 中，很多元素都具有 ID 和/或描述。IDs 和描述是首先被定义，并且始终

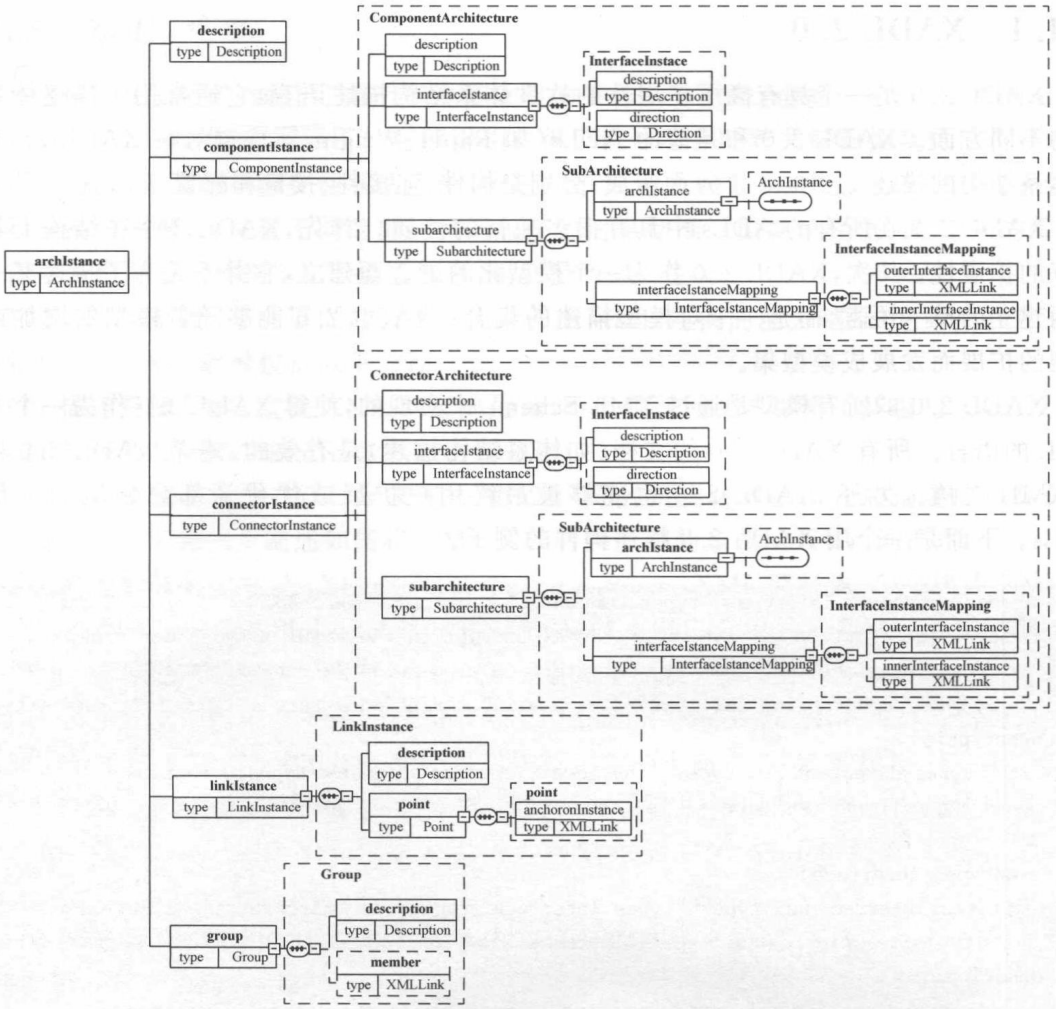


图 6-1　构件、连接件、连接实例及群组图

贯穿于整个 XADL 2.0。标识符在详细的文档中是唯一的,它必须具有易读性,必要时可以作为帮助。创建描述的好处是当描述元素的标识符变动时容易阅读。

此外,实例模式定义了一个用于结束其他 XML 2.0 模式的 XML Link 的元素类型,它是连接其他的 XML 元素。由于 XLink 标准缺少对应工具的支持,XADL 2.0 从 XML 标准引入链接策略,XADL 2.0 文档的编写者应该遵循 XML 子 Links 协议。

在 XADL 2.0 中,任何事物都具有一个用于识别 XMLLink(如事件存在的指向)的 ID。每一个 XMLLink 必须由 type 和 href 两个部分组成,而且都是通过 XLink 标准来定义的。href 域应该是由一个 URL 链接组成,http://server/directory/document.xml#id。这是一个清楚的标记,完全指定了 URL 链接,它链接到一个文档,而且用于 URL 部分的锚(如上面的#符号)指出了具体标记元素的标识符。

2. 实例模式的语义

一个体系结构实例是由一系列的构件实例、连接件实例和链接实例,以及它们之间的组合构成。每一个构件和连接件实例都有一个接口实例集。另外,在此对这些元素安排的等级关系做了一个假设,假设如下:

(1) 每一个构件和连接件实例都有一系列的接口实例。这些接口属于对应的构件和连接件实例并为它们服务。也就是说,从一个构件或连接件实例到其他构件、连接件的数据传输必须通过这些接口。接口实例相当于构件、连接件实例的内部“网关”,这和面向对象观点的接口是不一致的。

(2) 链接实例端点存在于普通的 XML 链接内,是用于链接两个接口实例的。因此,一个链接可能链接一个构件实例的接口实例 A 的“输入”细节,也可能是构件实例的接口实例 B 的“输出”细节。

(3) 链接实例是不定向的,这和指向顺序是不相关的。定向的数据流通过一个链接是由接口实例的方向(如“输入”“输出”等)决定。

(4) 链接实例不能通过扩展把语义增加到链接上,如果链接已经具有语义,那这个链接就是一个链接件。

构件、连接件、接口和链接的关系如图 6-2 所示。

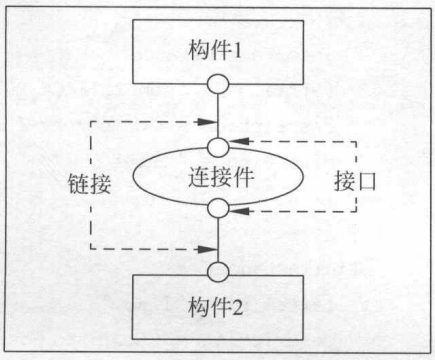


图 6-2 构件、连接件、接口和链接的关系

图 6-2 显示,链接的端点是接口,接口是构件和连接件与外部联系的“网关”。上述例子用 XADL 2.0 术语表示如下。

```
archInstance{
  componentInstance{
    (attr) id = "comp1"
    description = "Component 1"
    interfaceInstance{
      (attr) id = "comp1.IFACE_TOP"
      description = "Component 1 Top Interface"
      direction = "inout"
    }
    interfaceInstance{
      (attr) id = "comp1.IFACE_BOTTOM"
      description = "Component 1 Bottom Interface"
      direction = "inout"
    }
  }
  connectorInstance{
    (attr) id = "conn1"
    description = "Connector 1"
    interfaceInstance{
      (attr) id = "conn1.IFACE_TOP"
      description = "Connector 1 Top Interface"
      direction = "inout"
  }
  interfaceInstance{
    (attr) id = "conn1.IFACE_BOTTOM"
    description = "Connector 1 Bottom Interface"
    direction = "inout"
```

```
    }
    }
    componentInstance{
      (attr) id = "comp2"
      description = "Component 2"
      interfaceInstance{
         (attr) id = "comp2.IFACE_TOP"
         description = "Component 2 Top Interface"
         direction = "inout"
      }
      interfaceInstance{
      (attr) id = "comp2.IFACE_BOTTOM"
      description = "Component 2 Bottom Interface"
      direction = "inout"
      }
    }
    linkInstance{
      (attr) id = "link1"
      description = "Comp1 to Conn1 Link"
      point{
         (link) anchorOnInterface = "#comp1.IFACE_BOTTOM"
      }
      point{
         (link) anchorOnInterface = "#conn1.IFACE_TOP"
      }
    }
    linkInstance{
      (attr) id = "link2"
      description = "Conn1 to Comp2 Link"
      point{
         (link) anchorOnInterface = "#conn1.IFACE_BOTTOM"
      }
      point{
        (link) anchorOnInterface = "#comp2.IFACE_TOP"
      }
    }
}
```

3. XADL 2.0 的类系统

在软件体系结构研究范围内,类的精确含义和用法一直是一个激烈讨论的主题。有些方法采用的是相当传统的程序语言中的类和实例模型;其他的采用类的约束模型,即一个类是一个对整个元素的简单约束,任何元素所有的约束都是属于类的。

XADL 2.0 采用的是基于程序语言的类和实例模型。在这个模型中,构件、连接件和接口都有类(分别是构件类、连接件类和接口类)。由于链接不包含任何的体系结构语义,因此链接是不存在类的。类、结构和实例之间的关系如表 6-1 所示。

表 6-1　类、结构和实例之间的关系表

Instance(Run-time)	Structure(Design-time)	Type(Design-time)
Component Instance	Component	Component Type
Connector Instance	Connector	Connector Type
Interface Instance	Interface	Interface Type
Link Instance	Link	(None)
Group	Group	(None)

由于在实际的软件体系结构中，构件、连接件和接口可以存在非常相似的多重元素，它们可能共享行为或一个执行，所以构件、连接件和接口可以被类化。

所有相关的类结构是通过一个叫作 ArchTypes 的顶层 XML 元素来组织的。XADL 2.0 结构中可用的模型化体系结构类有构件类和连接件类。

模型化体系结构类语义时应该包括署名和设计时的子体系结构，有关这方面的具体内容，读者可以参考 UCI 的 XADL 2.0 的相关文献。

6.4.2　XBA

作为一种体系结构描述语言，XBA 主要是把 XML 应用于软件体系结构的描述，通过对组成体系结构的基本元素进行描述，同时利用 XML 的可扩展性，对现有的各种 ADL 进行描述及定义。

XBA 主要围绕体系结构的 3 个基本抽象元素(构件、连接件和配置)来展开，它实现了一种切实可行的描述软件体系结构的方法。下面通过一个实例对 XBA 进行说明。

图 6-3 给出了简单的客户-服务器结构，在图 6-3 中，Client 和 Server 是两个构件，RPC 为连接件，构件和连接件的实例构成了这个 C/S 系统的配置。

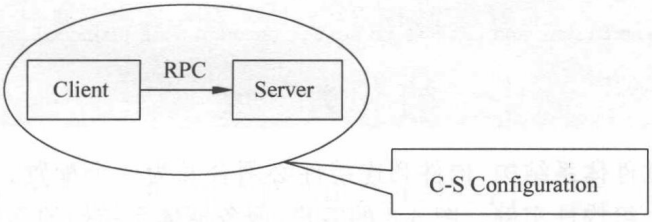


图 6-3　Client-Server 结构

在 XBA 中，一个构件描述了一个局部的、独立的计算，对一个构件的描述有两个重要的部分，即接口(interface)和计算(computation)。一个接口由一组端口(port)组成，每一个端口代表这个构件可能参与的交互。计算部分描述了这个构件实际所做的动作。

针对图 6-3 的体系结构风格，XBA 对构件类型的 XML Schema 形式的定义如下。

```
<complexType name = "componentType">
    <sequence>
        <element name = "Port" type = "portType" minOccurs = "0" maxOccurs = "unbounded">
        <element name = "Computation" type = "computationType" minOccurs = "0">
        <element name = "Port" type = "portType" minOccurs = "0" maxOccurs = "unbounded">
```

```
        </sequence>
        <attribute name = "Name" type = "string"/>
    </complexType>
    <complexType name = "portType">
        <element name = "Description" type = "string" minOccurs = "0"/>
        <attribute name = "Name" type = "string"/>
    </complexType>
    <complexType name = "computationType">
        <element name = "Description" type = "string"/>
        <attribute name = "Name" type = "string"/>
    </complexType>
```

一个连接件代表了一组构件间的交互,使用连接件的一个主要好处就是它通过结构化一个构件与系统的其他部分交互的方式,增加了构件的独立性。一个连接件实际上提供了构件必须满足的一系列要求和一个信息隐藏的边界,这个边界阐明了构件对外部环境的要求。下面给出了连接件 RPC 的 XBA 描述。

```
<Connector name = "Rpc">
  <Role name = "Source">
      <Description>
      Get request from client
      </Description>
   </Role>
  <Role name = "Sink">
      <Description>
      Give request from server
      </Description>
   </Role>
   <Glue>
   Get request from Client and pass it to server through some protocol
   </Glue>
</Connector>
```

为了描述整个软件体系结构,构件和连接件必须合并为一个配置。一个配置就是通过连接件连接起来的一组构件实例。图 6-3 的客户/服务器体系结构的 XBA 描述如下。

```
<Configuration name = "Client - Server">
   <Component name = "Client">
   <Component name = "Server">
   <Component name = "Rpc">
   <Instance>
      <ComponentInstance>
         <ComponentName> MyClient </ComponentName>
         <ComponentType> Client </ComponentType>
      </ComponentInstance>
      <! -- ect -->
      <ConnectorInstance>
         <ComponentName> MyRpc </ComponentName>
         <ComponentType> Rpc </ComponentType>
```

```
        </ConnectorInstance>
    </Instance>
    <Attachments>
        <Attachment>
          <From>MyClient.request</From>
          <To>MyRpc.Source</To>
        <Attachment>
        <!-- ect -->
      </Attachments>
</Configuration>
```

为了区分一个配置中出现的每一个构件和连接件的不同实例，XBA 要求每一个实例都有明确的命名，并且这个命名应唯一。Attachments 通过描述哪些构件参与哪些交互定义了一个配置的布局或称为拓扑(topology)，这是通过构件的端口和连接件的角色联系在一起来完成的。

通过利用 XML Schema 的扩展性机制，还可以方便地通过 XBA 核心 XML Schema 定义来获得新的可以描述其他 ADL 的 XML Schema。

采用 XBA 描述软件体系结构的主要优点如下。

(1) XBA 具有开放的语义结构，继承了 XML 的基于 Schema 的可扩展机制，在使用了适当的扩张机制之后，XBA 可以表示多种体系结构风格，而且可以利用 XML Schema 的 include 和 import 等机制重用已经定义好的 XML Schema，实现 ADL 的模块化定义。

(2) 利用 XML 的链接机制，可以实现体系结构的协作开发。可以先把体系结构的开发分解，由不同的开发者分别开发，然后利用 XML 的链接机制把它们集成起来。

(3) 易于实现不同 ADL 开发环境之间的模型共享。可能会存在多种基于 XML 的 ADL 开发工具，尽管它们所使用的对 ADL 的 XML 描述会有不同，但通过 XSLT 技术，可以很方便地在对同一体系结构的不同 XML 描述之间进行转换。

思 考 题

1. 请简述 XML、HTML、SGML、XHTML 之间的关系。
2. 请简述 CSS、XSL、DOM 的作用与功能。
3. 请简述 DOM 解析 XML 文档的过程。
4. 试对 XML 的各种编程接口进行比较分析，指出各自的优点、缺点，以及应用场合。
5. 就目前来看，XML 具有多种应用，但技术是发展的，请根据你的知识背景，阐述 XML 未来可能会应用在什么场合？
6. 试用 XADL 2.0 和 XBA 描述一个基于 B/S 软件体系结构的软件系统。

参考文献

第7章 动态软件体系结构

当前，软件体系结构研究主要集中在静态体系结构上，这种体系结构在运行时不能发生改变。但是一些需要长期运行并且具有特殊使命的系统（如金融系统、航空航天系统、交通系统、通信系统等），如果系统需求或环境发生变化，此时停止运行进行更新或维护，将会引起高额的费用和巨大的风险，对系统的安全性也会产生很大的影响。静态体系结构缺乏表示动态更新的机制，很难用它来分析描述这样的系统。因此，有关学者开始研究动态软件体系结构（dynamic software architecture），主要研究软件系统由于特殊需要必须在连续运营情况下的体系结构变化与支撑平台。

7.1 动态软件体系结构概述

历经传统的结构化开发方法和面向对象开发方法，基于软件体系结构、构件的开发方法已经逐渐成为当前软件开发的主流，软件开发的基本单位已从传统的代码行、对象类转变为各种粒度的构件、构件之间的拓扑结构所形成的软件体系结构。这种转变给软件开发带来更多的灵活性，可以通过构件重用和替换来实现，即实现构件的“即插即用”。而灵活性的一方面就是动态性。在软件体系结构层次上实现动态性会给大型软件系统的开发提供可扩展性，用户自定义和可演化性。而且，软件体系结构的动态改变和演化对于需要长期运行或具有特殊任务的系统尤其重要。

由于系统需求、技术、环境、分布等因素的变化而最终导致软件体系结构的变动，称为软件体系结构演化。软件系统在运行时刻的体系结构变动称为体系结构的动态性，而将体系结构的静态修改称为体系结构扩展。体系结构的扩展和体系结构的动态性都是体系结构适应性和演化性的研究范畴。

体系结构的动态性主要分为以下三类。

（1）交互式动态性。例如，允许在复合构件的固定连接中改变数据。

（2）结构化动态性。例如，允许对系统添加或删除构件或连接件。

（3）体系结构动态性。例如，允许构件的整个配置发生改变。

允许在系统运行时发生更新的软件体系结构称为动态软件体系结构，动态体系结构在系统创建后可以动态地更新。系统结构的动态改变将会影响正在运行的系统的内部计算，这使得运行系统的动态更新变得很复杂且难以很好地解决。目前，一些主流操作系统和部分构件对象模型中，更新机制已经得到一些应用，例如，UNIX 内核动态链接库、CORBA 和 DCOM 中的构件组装机制等。这些机制允许系统在运行时添加新的库并执行，使得系统在不需要重编译的情况下进行更新。

目前，动态体系结构的研究主要分为两个方面，一个方面是研究模拟和描述体系结构动态更新的语言，另一个方面是研究支持体系结构动态更新的执行工具。

1. 模拟和描述体系结构动态更新

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 ADL 只描述系统的静态结构，不支持对体系结构动态性的描述。

近年来，对这一方面的研究主要集中在对现有的一些 ADL 扩展以支持体系结构的动态性，现已研究出一些支持动态体系结构的 ADL。在动态体系结构建模和描述方面，C2 支持结构动态性，它定义了专门支持体系结构修改的描述语言 AML；Darwin 对软件体系结构的修改采用相应的脚本语言，它具有惰性和动态实例化两种动态机制，但它只涉及计算结构单元，对于事物逻辑和体系结构配置的互动没有考虑；Unicon 只提供了有限的机制定义新的连接件类型，不能描述运行状态下的体系结构变化；Wright 通过事件和控制视图决定重新配置的条件和如何触发重新配置，具有一定的动态体系结构描述能力，但它侧重于动态的体系结构形式，缺乏很强的结构单元动态特征，同时，引入外界不透明的控制事件也影响了它的可复用性。

2. 体系结构动态更新的执行

对于动态体系结构应用方面的研究，还很不成熟。目前，支持动态体系结构机制的主要有 ArchStudio 工具集和 SAA(Software Architecture Assistant，软件体系结构助理)。

ArchStudio 工具集是由加州大学 Irwine 分校提出的，它支持交互式图形化描述和 C2 风格描述的体系结构的动态修改，概念模型如图 7-1 所示。

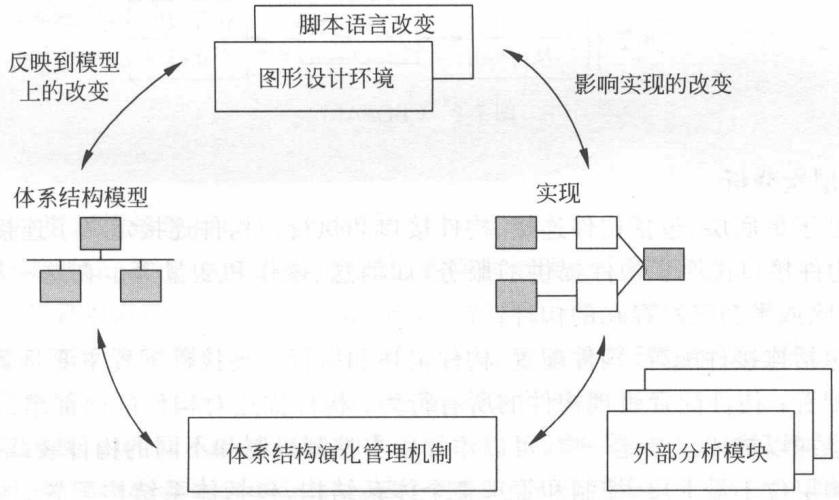


图 7-1　ArchStudio 概念模型

运行系统的改变通过一系列工具反映到体系结构模型上，例如，脚本语言的改变和交互式的图形设计环境。体系结构的改变包括增加、删除或更新构件、连接件，以及系统拓扑结构发生改变。AEM(Architecture Evolution Manager，体系结构演化管理机制)通报这些改变，并有权力撤销破坏系统整体性的改变。如果改变没有破坏系统的整体性，AEM 就对系统的执行作相应的改变。

SAA是由伦敦皇家学院提出的,它也是一种交互式图形工具,可以用来描述、分析和建立动态体系结构。体系结构设计人员可以用SAA图形化地描述Darwin系统结构模型,用一些外部工具分析结构,并生成框架代码。尽管SAA提供了智能化图形设计,但它不支持运行系统的监控和操作。

当前主流的体系结构模型CORBA、COM/DCOM、EJB等,都不支持体系结构的动态更新。同时,动态体系结构由于其本身的复杂性,比静态体系结构需要更多的形式化描述机制和分析工具,形式化描述机制用来描述运行时的更新,分析工具用来帮助验证这些更新的属性。由于缺乏通用的结构模型、有效的形式化描述机制和分析工具,使得目前学术界对于动态体系结构的研究还不成熟,处于摸索阶段。

7.2 基于构件的动态系统结构模型

CBDSAM(Component Based Dynamic System Architecture Model,基于构件的动态系统结构模型)支持运行系统的动态更新,该模型分为三层,分别是应用层、中间层和体系结构层。其结构如图7-2所示。

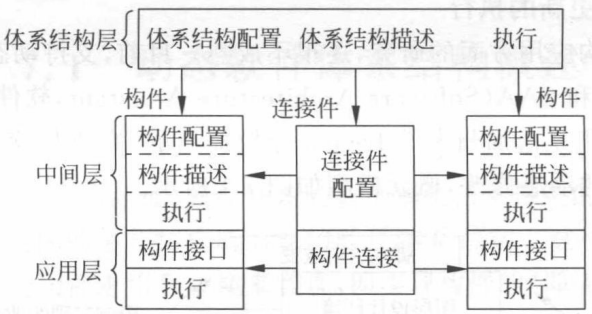


图7-2 CBDSAM

1. 各个层次分析

应用层处于最底层,包括构件连接、构件接口和执行。构件连接定义了连接件如何与构件相连接;构件接口说明了构件提供的服务,如消息、操作和变量等。在这一层,可以添加新的构件、删除或更新已经存在的构件。

中间层包括连接件配置、构件配置、构件描述和执行。连接件配置主要是管理连接件及接口的通信配置;构件配置管理构件的所有行为;构件描述对构件的内部结构、行为、功能和版本等信息加以描述。在这一层,可以添加版本控制机制和不同的构件装载方法。

体系结构层位于最上层,控制和管理整个体系结构,包括体系结构配置、体系结构描述和执行。其中,体系结构描述主要是描述构件以及它们相联系的连接件的数据;体系结构配置控制整个分布式系统的执行,并且管理配置层;体系结构描述主要对体系结构层的行为进行描述。在这一层,可以更改和扩展更新机制,更改系统的拓扑结构,以及更改构件到处理元素之间的映射。

CBDSAM中,每一层都有一个执行部分,主要是对相应层的操作进行执行。在更新时,必要情况下将会临时孤立所设计的构件。在更新执行之前,要确保所涉及的构件停止发送新的请求;在更新开始之前,连接件的请求队列中的请求全部已被执行。而且,模型封装了

连接件的所有通信，这样可以很好地解决动态更新时产生的不一致性问题。

2. 更新请求描述

更新可以由用户提出，也可以由系统自身发出请求。以下程序描述了一个包含多种更新操作的更新请求实例。

```
<updata_descriptor>
   <add_obj to = "server01//comp1">
      <object name = "C">
         <implemetation>…</implementation>
      </object>
   </add>
<remove_obj from = "server01//comp1">
   <object name = "D">
   </object>
</remove>
<updata_obj in = "server01//comp1">
   <object name = "A" method = "replace">
      <old_version>1.0</old_version>
      <new_version>1.1</new_version>
      <implemetation>…</implementation>
   </object>
<updata>
<updata_obj in = "server01//comp1">
   <object name = "B" method = "dynamic">
      <old_version>1.0</old_version>
      <new_version>1.1</new_version>
      <updata_function>…</updata_function>
      <implemetation>…</implementation>
   </object>
<updata>
<updata_description>
```

一般来说，更新描述包括以下几个部分。

(1) 更新类型(updata type)：包括添加、删除和修改一个新的构件。

(2) 更新对象列表(list of updata objects)：需要更新对象类的 ID 号。

(3) 对象的新版本说明(new version of objects)：对象的新版本执行情况。

(4) 对象更新方法(updata method)：更替、动态和静态。

(5) 更新函数(updata function)：用来更新一个执行对象进程的状态转换函数。

(6) 更新限制(updata constraints)：描述更新(包括子更新)和它们之间的关系的序列，例如只有对象 A 的版本≥2.0 时，对象 A 才能被更新。

3. 更新执行步骤

按照 CBDSAM 的结构，对系统进行更新，一般来说，有以下几个步骤。

(1) 检测更新的范围：在执行更新之前，首先要判断是局部更新还是全局更新，局部更新作用于需更新构件的内部而不影响系统的其他部分。全局更新影响系统的其他部分，全局更新需要发送请求到更高的抽象层。

(2) 更新准备工作：如果更新发生在应用层，构件配置器等待参与的进程(或线程)发

出信号,以表明它们已处于可安全执行更新的状态;如果更新发生在配置层,就需要等待连接件中断通信,其他构件配置器已完成它们的更新。

(3) 执行更新:执行更新,并告知更新发起者更新的结果。

(4) 存储更新:将构件或体系结构所执行的更新存储到构件或者体系结构描述中。

4. 实例分析

下面通过局部更新和全局更新来分析 CBDSAM 是如何支持体系结构动态更新的。

1) 局部更新

局部更新由于只作用于需要更新的构件内部,不影响系统的其他部分,因此比全局更新要简单,步骤如图 7-3 所示。

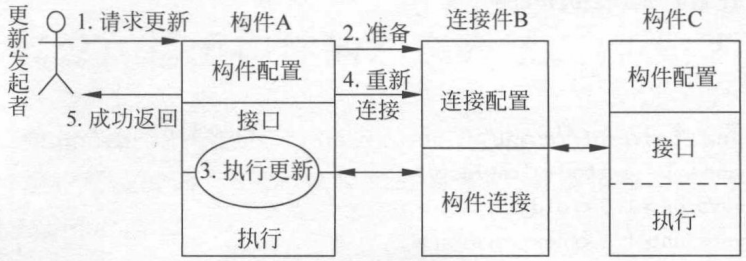


图 7-3 局部更新

图中线条人表示更新发起者,更新发起者可以是系统的某一状态,也可以是系统用户。构件 A 和构件 C 通过连接件 B 连接,构件 A 内部需要更新。局部更新请求可按下列步骤执行。

(1) 更新发起者发出一个更新请求,这个请求被送到构件 A 的配置器中,构件配置器将分析更新的类型,从而判断它是对象的局部更新。

(2) 由于更新为局部更新,构件 A 的配置器发出一个信号给连接件以隔离构件 A 的通信,准备执行更新。

(3) 构件 A 的配置器开始执行更新。

(4) 更新执行完毕后,构件 A 的构件描述被更新,并且构件 A 发送一个消息给连接件 B,两者间的连接被重新存储起来。

(5) 将更新结果返回给更新发起者。

由上述分析可知,在整个更新过程中,构件 C 都没有受到影响,这说明按照 CBDSAM 的方法,不会影响系统的其他部分运行。

2) 全局更新

下面以一个 Client-Server 系统动态更新实例来说明 CBDSAM 在全局更新中的应用。在这个例子中,要求更新某一 Server 构件。按照 CBDSAM,在此采用 UML 的顺序图来描述动态更新过程,如图 7-4 所示。

根据图 7-4 的描述,更新过程如下。

(1) Server 构件配置器接收到更新发起者提出的更新请求后,向体系结构配置器提出更新请求。

(2) 体系结构配置器对更新请求的类型进行分析,判断是否在更新请求限制(属于全局更新还是局部更新)范围内,不在更新范围内的更新不予执行;如果在更新限制范围内,体

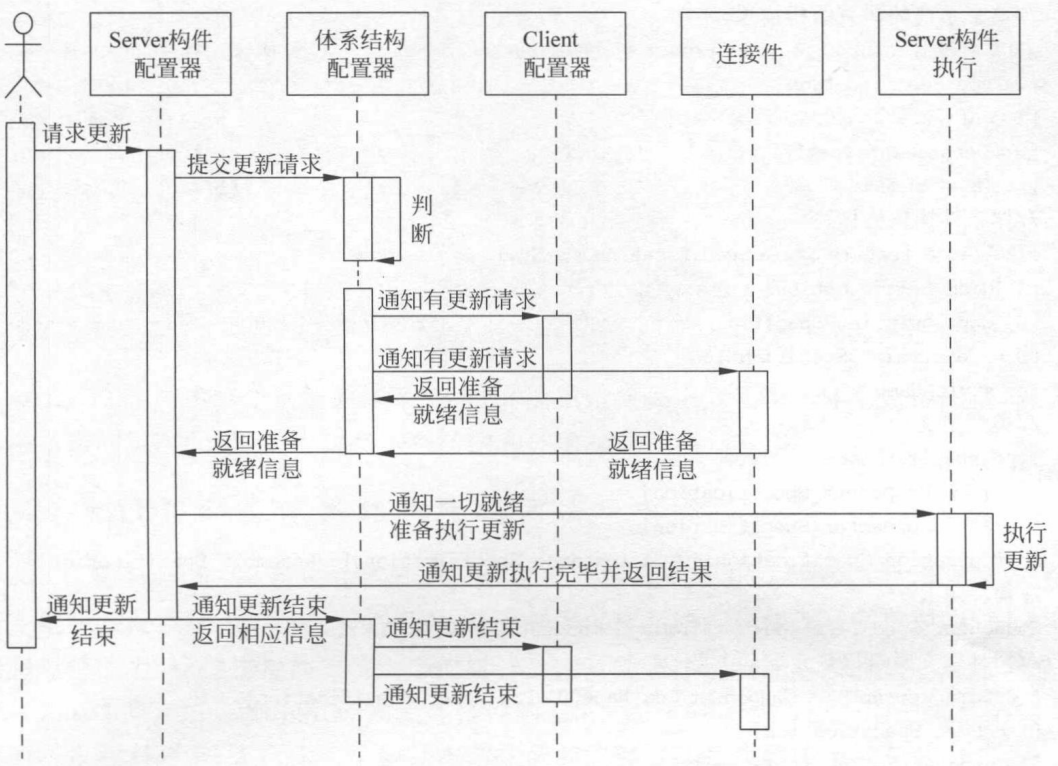


图 7-4　Client-Server 系统更新顺序图

系结构配置器对更新所涉及的连接件和构件(本例中为 Client 构件和连接件)发出消息,要求它们做好更新准备工作。

(3) 准备工作完成后,Client 构件配置器和连接件向体系结构配置器返回就绪信息。

(4) 一切准备就绪后,体系结构配置器通知 Server 构件进行更新。

(5) 更新执行完毕后,向 Server 构件配置器、体系结构配置器和更新发起者通知更新执行完毕并返回更新结果;同时,体系结构配置器通知 Client 构件和连接件更新结束,可继续正常运行。

这样,在没有影响系统运行的情况下,按照更新发起者的要求对系统进行了更新,并维护了系统的一致性。

7.3　πADL 动态体系结构

πADL 借鉴与遵循 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,围绕体系结构抽象级别的实体(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。

7.3.1　πADL 描述体系结构的框架

下面运用扩充的 BNF 范式给出 πADL 描述体系结构的框架。其中,运用//标记注释,[1...]表示其中的项出现一次。[01...]表示其中的项出现 0 次或 1 次,[1+...]表示其中的项出现 1 次或多次,[0+...]表示其中的项出现 0 次或多次。

```
//单个系统的体系结构描述
πADL Architecture Specification::= System Systme_Name
[1 Type_Specification]
[1 Configuration_Specification]
[0 + Constraint_specification]
[End System_Name]
//体系结构风格描述
π ADL Architecture Style Specification::= Style Style_Name
[1 Idiom_Specification]
[01 Configuration_Specification]
[0 + Constraint_Specification]
End style_Name
//类型定义
Type_specification::= Type
    [1 + Component_Specification]
    [1 + Connector_Specification]
Configuration_Specification::= [1 Instance_Specification][1 Assemble_Specification]
//构件定义
Component_Specification::= Atomic_Component|COMPOITE_COMPONENT
//构件定义时可以规定它的参数
Atomic_component::= Component Com_Name[01 Parameter_Specification]
[1 + Port_Specification]
[1 Computation_Specification]
//端口的行为用扩充的π进程、接口类型或参数定义
[Port_Specification::= PortpName = EXTENDED_π_PROCESS
                          |Interface_Name|Parameter_Name]
[Computation_Specification::= Computation = EXTENDED_π_PROCESS]
//连接件描述
Connector_Specification::= Atomic_Connector|COMPOITE_CONNECTOR
//连接件定义时可以规定它的参数
Connector_Specification::= connector Con_Name[01 Parameter_Specification]
[1 + Role_Specification]
[1 Glue_Specification]
//角色的行为用扩充的π进程、接口类型或参数定义
[Role_Specification::= Role RName = EXTENDED_π_PROCESS
                              |Interface_Name|Parameter_Name]
[Glue_Specification::= Glue = EXTENDED_π_PROCESS]
//实例规约
Instance_Specification::= Instances:[1 + Instance_Definition]
Instance_Definition::= [1 Component_Definition|Connector_Definition]
//实例定义时,可能需要设定相关参数
Component_Definition::= [1 + Com_Name[01(Actual_parameter)]]:Com_type
Connector_Definition::= [1 + Con_Name[01(Actual_parameter)]]:Con_Type
//组装规约
Assemble_Specification::= Assemble:
[1 Connection_Assembly|SIMPLE_ASSEMBLY|REPLICATE_ASSEMBLY]
Connection_Assembly::= [1 + Com_Name.Pname AttachTo Con_Name.Rname]
//风格规约
Idiom_Specification::= [1 + Interface_Specification]Component_Specification
                      |Connector_Specification|
```

```
Interface_Specification::= Interface_Name = EXTENDED_π_PROCESS
//参数可以是进程、整数、行为名字或其他,πADL 规定的任意部分可为参数
Parameter_Value::= EXTENDED_π_PROCESS|INT|Action_Name|OTHER
//名字规约
Com_Name::= IDENTIFIER          Com_Type::= IDENTIFIER
Con_Name::= IDENTIFIER          Con_Type::= IDENTIFIER
PName::= IDENTIFIER             RName::= IDENTIFIER
Config_Name::= IDENTIFIER       Style_Name::= IDENTIFIER
Interface_Name::= IDENTIFIER    Parameter_Name::= IDENTIFIER
Parameter_Type::= IDENTIFIER    Action_Name::= IDENTIFIER
```

πADL 既能描述单个系统的体系结构,又能描述代表一类系统的体系结构风格,并且支持体系结构的层次配置,支持动态体系结构和体系结构演化。

7.3.2 πADL 动态体系结构建模方法

π 演算是 20 世纪 90 年代计算机并行理论领域最重要的并发计算模型,能够描述结构不断变化的并发系统。πADL 借鉴了 Wright 提供的体系结构建模框架和思想,利用 π 演算的动态建模能力,通过连接件实现动态配置。πADL 动态体系结构建模的基本思路如下。

(1) 构件的 Computation(计算)、连接件的 Glue(胶水)进程中,插入用于体系结构动态演化的特定控制名字,表达体系结构动态演化的起因和安全地进行动态演化的时间。

(2) 运用 π 演算作为统一的形式语义基础,建立单独的动态配置进程,形式化描述体系结构动态演化方案。它与构件、连接件的 Computation 进程、Glue 进程进行相互交互和作用,总体控制体系结构动态演化。

(3) 运用 π 演算固有的动态建模能力,通过连接件动态输入不同构件的交互行为名字,实现与构件的动态连接和交互,并据此给出体系结构动态配置的形式语义。

πADL 进行动态体系结构建模的具体方法如下。

(1) 对体系结构动态配置的起因建模。πADL 在构件的 Computation 进程、连接件的 Glue 进程中向配置进程输出特定的控制名字,表达引发动态配置的内部原因,如输出 Error、Exception 表达构件或连接件内部发生错误或异常,要求配置进程进行动态配置。而在配置进程中从外部环境输入特定的控制名字,表达体系结构动态配置的外部原因。如 RequestUpgrade 表示用户要求服务升级。

(2) 对体系结构动态配置的时间建模。πADL 在构件的 Computation 进程、连接件的 Glue 进程中向配置进程中插入特定的控制名字 BeginOK,表达它们运行到该处的时候,处于一个安全的状态,能够开始接收动态配置指令,进行系统演化。

(3) 对体系结构动态配置的非瞬时性建模。πADL 根据体系结构配置方案,在构件的 Computation、连接件的 Glue 进程中向配置进程引入特定的控制名字 EndOK,与配置进程的相关行为同步,控制构件或连接件,使它们只有在动态配置结束后,才能重新启动执行。

(4) 对构件从断点开始继续执行的能力进行建模。根据 πADL 动态体系结构建模语义,构件从连接件上撤换下来时,系统中不再存在能够与之交互的进程,它的交互要求不能得到满足,故不能继续运行。当它再次与连接件连接时,连接件输入它的交互行为名字,从而能够与它交互,构件得以在断点开始继续执行。

(5)对体系结构动态变化的基本操作建模。πADL运用特定的动作名字表示动态配置的基本操作。τ_New_Iname_Tname表示生成类型Tname的实例Iname,τ_Delete_Iname表示删除构件或连接件实例Iname。τ_Delete_Attach_M_N_TO_I_J、τ_Detach_M_N_From_I_J分别表示在端口M.N和角色I.J之间建立连接和撤销连接。它们都是配置进程的内部行为。

(6)对体系结构动态演化的方案建模。πADL运用π演算作为统一的语义基础,建立专门的动态配置进程Dynamic_Configuror,描述体系结构动态演化的方案。Dynamic_Configuror模拟一个监控进程,根据系统运行情况和环境变化因素,实施相应动态变化方案。Dynamic_Configuror通过专门的通道X_Config、En_Config与实例X、En外部环境进行通信,协调体系结构动态变化的诸多因素,控制体系结构动态变化。运用BNF范式,定义Dynamic_Configuror的语法和结构如下。其中,单字母的终结符运用双引号标记,如"+"表示π演算的"和"运算符。

```
//动态配置进程包括两个部分: 系统初始配置和动态配置
DYNAMIC_CONFIGUROR:: = Dynamic_Configuror
    [1 Initial_Configuration_Instruction]        //系统初始配置指令
    [1 Dynamic_Configuration_Program]            //系统动态配置方案
//执行系列配置动作,创建初始系统,并执行动态监控进程 Dconfiguror
Initial_Configuration_Instruction:: = [1 + Action"."] Dconfiguror
//动态配置方案给出 Dconfiguror 的定义,它监控系统和环境,并执行动态配置
Dynamic_Configuration_Program:: = Dconfiguror =
    [01 System_Monitor]                          //系统监控部分
    [01 "+"Environment_Monitor]                  //环境监控部分
//从多个构件或连接件实例接受动态配置的控制信息并执行动态配置规则
System_Monitor:: = [1 X_EnConfig(y).Rule] [0 + "+"X_Config(y).Rule]
//环境监控部分接受环境的动态配置信息并执行动态配置规则
Environment_Monitor:: = En_Config(y).Rule
//根据收到的各种控制信息,执行具体的动态配置方案
Rule:: = [1 ([y = controlname].Subrule[0 + "+"[y = controlname].Subrule])]
//向多个相关实例发布动态配置开始指令
Begin_Config:: = [1 + X_Config<BeginOk>"."]
//执行多个动态配置行为
Dynamic_Cofig:: = [1 + Action"."]
//向多个相关实例发布动态配置结束指令,并循环执行动态配置方案 Dconfiguror
End_Config:: = [1 + X_Config<EndOk>"."]Dconfiguror
//配置基本命令包括实例的生成和删除,连接关系的建立和撤销
Action:: = τ_New_Iname_Tname_|τ_Delete_Iname|
          τ_Attach_M_N_TO_I_J|τ_Detach_M_N_From_I_J
//动态配置控制名字,动态体系结构建模人员能够定义自己的控制名字
Controlname:: = Error|Exception|FixOk|RequestUpdate| …
X:: = IDENTIFIER                y:: = IDENTIFIER
Iname:: = IDENTIFIER            Tname:: = IDENTIFIER
M:: = IDENTIFIER                N:: = IDENTIFIER
```

具体过程解释如下。

第一步,Dynamic_Configuror执行系列配置动作,创建一个运行系统。

第二步,通过X_Configuror、En_Configuror通道监听各个实例X和外部环境。当收到

表示动态配置起因的控制名字时，根据不同的控制名字，执行不同的配置规则 SubRule。在 SubRule 中，首先通过系列 X_Config 通道向实例 X 输出控制名字 BeginOK，发出动态配置开始指令，等待和控制相关构件、连接件处于安全状态。

第三步，执行系列配置行为，最后输出系列控制名字 EndOK，告诉相关构件、连接件配置结束，重新开始执行。

第四步，SubRule 执行完动态配置后，执行 Dconfiguror，继续处于监控状态。

7.3.3 πADL 动态体系结构建模语义

πADL 动态体系结构建模运用 π 进程作为体系结构动态配置的形式语义，形式化描述动态配置系统的整体行为。其基本思想是：动态配置软件系统的整体行为是所有构件实例的进程、连接件实例的进程和动态配置进程的并发运行。但关键要点是表达构件、连接件的动态创建和删除，以及构件、连接件之间的动态连接关系。πADL 的方法如下所述。

(1) 构件、连接件的动态创建和删除。分析 Configuror 程序，在创建初始系统和每次动态配置结束的地方，插入代表新创建的构件、连接件的实例进程并与余下的 Configuror 进程并发执行，实现构件的动态创建。π 演算并不直接提供进程删除的手段，故只能运用等价的方法对构件、连接件的删除建模，即使它们不能再执行。连接件删除时，通过输入行为，替换连接件的所有通道名字为 Void，从而它也不能找到对偶的通道名字，不能再执行，等价于连接件删除。

(2) 运用 π 演算的动态建模能力，借助连接件实现构件、连接件的动态连接关系。πADL 在连接件的 Glue 进程和配置进程 Configuror 中自动插入连接件交互通道的输入、输出行为。动态配置中，Configuror 根据配置变化情况，向连接件输出当前的连接通道，连接件接受该输出，得到新的连接通道，从而动态改变与构件的连接关系，实现体系结构的动态变化。借助连接件实现体系结构的动态变化，构件的 Computation 进程不受动态变化的影响，便于动态变化的分析和实现。

下面给出动态体系结构行为推导算法。算法的骨架是分析动态配置进程 Dynamic_Configuror 和构造描述动态体系结构行为的 π 进程。

算法(动态体系结构行为推导算法)

输入：构件、连接件定义，动态体系结构配置进程 Dynamic_Configuror

输出：描述动态体系结构行为的 π 进程

步骤：

第一步，改造连接件，在连接件中插入交互通道输入行为。在所有连接件 Glue 代码中，在动态配置开始和结束之间，即在 Config(y).[y=BeginOK]和 Config(z).[y=EndOk]之间，插入用于输入连接交互通道的行为 Config($\bar{x}$)，其中 $\bar{x}$ 为该连接件所有交互行为名字构成的矢量序列，于是形成 Config(y).[y=BeginOk].Config($\bar{x}$).Config(z).[z=EndOk]。它的作用是动态配置结束之前，连接件重新得到交互通道，从而动态改变构件、连接件之间的交互关系，实现动态配置。

第二步，分析进程 Dynamic_Configuror 的初始行为，动态创建系统的初始配置。即分析 Initial_Configuration_Instruction::=[1+ Action"."]Dconfiguror，根据其中系列构件、连接件创建动作和它们之间的连接配置信息，调用静态配置的组装推导算法，得到表达初始

配置的进程 Pinitial。并运行 Pinitial|Dconfiguror 替换 Dconfiguror,得到[1+ Action". "](Pinitial|Dconfiguror)。该步骤的作用是动态创建初始系统,该初始系统与动态配置进程并发执行。

第三步,分析 Configuror 的动态配置行为,动态改变体系结构。即分析语法元素 SubRule::=[1 Begin_Config][1 Dynamic_Config][1 End_Config]中的动态配置部分 Dynamic_Config,具体执行步骤如下。

(1)动态创建新的构件和连接件。根据动态配置行为中构件、连接件的创建信息,新建连接件的端口角色连接情况,调用静态配置的组装推导算法,得到表达它们行为进程的 PCreated。

(2)动态改变端口、角色的连接关系。对已存在,但所连接构件发生变化的连接件 X_1, X_2,…,X_n,根据新的端口、角色连接关系,向它们输出新的交互通道,得到 X_1_Config$<\bar{y}_1>$……X_n_Config$<\bar{y}_n>$,其中 $\bar{y}_i$ 是交互通道的名字序列,它的元素形如 M_N_A 表示构件 M 的端口 N 连接到该连接件上,连接件接受 M_N_A,从而与构件 M 建立新的连接交互关系。

(3)连接件删除的处理。设在动态配置行为中,删除连接件 C_1,C_2,…,C_m,向它们输出代表空的 Void 交互通道,得到 C_1_Config$<\bar{v}_1>$……C_n_Config$<v_n>$,其中 v_i 是元素 C_i_Void 构成的元组,它们得到该交互通道后,因为系统中的其他进程皆不具有该交互通道,因此它们不能执行,从而等价于连接件删除。

(4)综合。根据(1)、(2)、(3)的结果,修改 Subrule 如下:在动态行为后插入 X_1_Config$<\bar{y}_1>$……X_n_Config$<\bar{y}_n>$. C_1_Config$<v_1>$……C_n_Config$<\bar{v}_n>$,并用(Pcreated|DConfiguror)代替 End_Config::=[1+ X_Config<EndOk>". "]Dconfiguror 中的 Dconfiguror。

第一步对连接件处理之后,经过第二步、第三步,对动态配置进程 Dynamic_Configuror 进行分析、处理,结果得到的进程就是描述该动态体系结构行为的 π 进程。

7.4 动态体系结构的描述

与静态体系结构的描述一样,动态体系结构的描述既可以使用描述语言,也可以使用形式化的描述;既可以使用 UML 建模机制,也可以使用 XML 建模机制。

7.4.1 动态体系结构描述语言

现在已经存在一些用于帮助设计师对软件体系结构进行可视化、形式化描述的工具,Rational Rose 就是其中的代表。UML 作为体系结构建模工具,它不是一种体系结构的描述语言,而是一种设计语言。因此,开发动态软件 ADL 是很有必要的。近年来,已经开发出了很多动态体系结构描述语言,如 Darwin、Dynamic ACMED、Dynamic Wright、Rapide 等。下面对 Dynamic Wright 和 Darwin 的特点进行简单的介绍。

1. Dynamic Wright

Dynamic Wright 是体系结构描述语言 Wright 的一个扩展,Dynamic Wright 的主要目的是试图模拟或标记,以解决软件系统的动态性。采用 Dynamic Wright 具有较多的良好特

性，其中之一就是能够很容易地描述软件的动态环境。

下面从一个实例出发，说明采用 Dynamic Wright 来描述客户-服务器体系结构的方法。在客户-服务器的体系结构中，如果设计师想要对 Client（客户）的每一步工作所依赖的 Server（服务器）进行说明，比如 Client 最初是依赖于 Server1 的，当且仅当 Server1 出现问题时，Client 将与 Server2 进行交互。因此，设计师必须用一个符号来表示每一个接触点 L。

图 7-5 是上述问题的一个静态描述，它并不能够清楚地反映出设计师对体系结构的动态依赖关系，这样有可能导致设计师把一些关键的方面遗漏，它还需要一些额外的文本对体系结构的行为进行说明。因此，Dynamic Wright 在其他方面提出了一些用于描述体系结构动态变化的新符号，即配置 configuror(C)。通过增加一个 configuror 到 Wright 的标记中，设计师就能够很好地阐述控制行为，使得设计师能更好地描述系统体系结构。

configuror 主要涉及以下一些问题。

(1) 什么时候软件体系结构应该重新配置？

(2) 什么原因使得软件体系结构需要进行重新配置？

(3) 重新配置应该怎样进行？

图 7-6 是对上述问题的动态描述，通过引入虚线和配置 C 动态地描述该系统的动态特征。这样设计师就能够很容易地把系统的动态环境表述清楚。因此，Dynamic Wright 是非常适合动态软件体系结构的环境描述，有关 Dynamic Wright 的详细内容，读者可以参考相关文献。

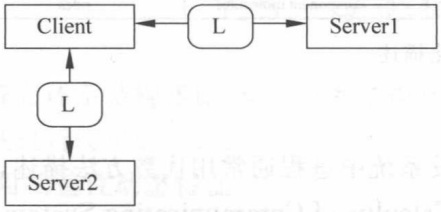


图 7-5　静态描述

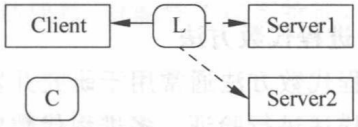


图 7-6　动态描述

2. Darwin

Darwin 是一个用于描述系统配置规划的 ADL，它把一个程序看成是由不同级别的构件进行相应的配置。相对于其他的 ADL，老程序员在使用 Darwin 上显得更容易些。

Darwin 具有很多其他 ADL 的图形表示和文本表示的特点，Darwin 与其他的 ADL 的主要不同之处在于：Darwin 具有一个用于对构件所需要的和提供的服务进行指定的规则。图 7-7 是一个采用 Darwin 对 filter 构件的图形化和文本化的描述。

通常，构件能够提供一些服务也可能需要一些服务。服务的命名是局部命名（如 next 和 output），在 Darwin 中，每一个服务需要被局部地指定，也就是说每一个构件能够从系统中分离出来并且进行独立测试。

Darwin 对于表示体系结构构件的开发和设计是一个相当成熟的工具，但是 Darwin 在其他方面的描述上并不是完美的。

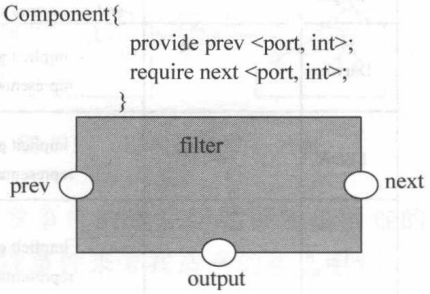


图 7-7　构件 filter 的 Darwin 描述

7.4.2 动态体系结构的形式化描述

动态软件系统的形式化描述包括软件体系结构的描述、体系结构的重新配置和系统行为的描述。对动态软件体系结构的形式化描述,通常可以采用图形化方法、进程代数方法、逻辑方法等。

1. 图形化方法

由于图能够直观地描述系统体系结构及其风格,因此,用图形化方法描述软件体系结构和体系结构风格是很自然的事情。因此,用图的重写规则描述动态软件体系结构的配置也是一种很自然的方法。采用图形化方法描述动态软件体系结构的主要有 Le Metayer 的描述方法、Hirsh 等的描述方法、Taentzer 等的描述方法和 CHAM 方法。图 7-8 是不同的图形化描述方法对软件体系结构的各个方面的描述。

	Architectural Structure		Architectural Element Behavior		Architectural Reconfiguration
	Architectural Style	System Architecture	components	connectors	
Le Metayer approach	Context-free graph grammar	Graph(formally defined as a multiset)	nodes of a graph and a csp like behavior specification	edges of a graph	graph rewriting rules with side conditions to refer to the status of public variables
Hirsch et al approach	Context-free graph grammar	hypergraph	edges of a graph with CCS labels	nodes of a graph[point-point communication and broadcast communication]	graph rewriting rules
Taentzer et al approach	—	distributed graph (network graph)	local graph of each network graph node and local transformations between local graphs	edges of a graph	graph rewriting rules
CHAM	creation CHAM	—	molecule	links between two component molecules	evolution CHAM reaction rules

图 7-8 图形化描述

2. 进程代数方法

进程代数方法通常用于研究并发系统,在并发系统中进程通常用代数方法描述,用微分方法对描述进行验证。多进程代数应用于 CCS(Calculus of Communicating Systems),CSP(Communicating Sequential Processes)和 π 演算。常用的进程代数方法有 Dynamic Wright、Darwin、LEDA、PiLar 等。图 7-9 是不同的进程代数描述方法对软件体系结构的各个方面的描述。

	Architectural Structure		Architectural Element Behavior		Architectural Reconfiguration
	Architectural Style	System Architecture	components	connectors	
Dynamic Wright	—	implicit graph representation(components and connectors are nodes)	port(interface)+compution (behavior)	roles(interface) + glue(behavior)	CSP
Darwin	—	implicit graph representation	programming language + component specification of objects	support for simple bindings	CSP
LEDA	—	implicit graph representation	interface specification, composition and attachment spcification	attachments at top level components	π-calculus
PiLar	—	implicit graph representation	components with ports, instances of other components and constrains	support for simple bindings	CCS

图 7-9 进程代数描述方法

3. 逻辑化描述方法

逻辑化描述方法是动态软件体系结构描述的形式化基础，常用的逻辑描述方法有Gerel (Generic Reconfiguration Language)、Aguirre-Maibaum 方法、ZCL 方法等。图 7-10 是不同的逻辑描述方法对软件体系结构的各个方面的描述。

	Architectural Structure		Architectural Element Behavior		Architectural Reconfiguration
	Architectural Style	System Architecture	components	connectors	
Gerel	—	implicit graph representation	interface in Gerel language and behavior in a programming language	defined by bind operation in configuration component	first order logic
Aguirre-Maibaum	—	implicit graph representation	class with attributes, actions and read variables	association consisting of participants and synchronization connections	first order logic, temporal logic
ZCL	—	implicit graph representation (defined by set of state schema in Z)	state schema in Z	connection between ports of component	operation schema in Z(predicate logic and set theory)

图 7-10　逻辑化描述方法

上述的几类动态软件体系结构形式化描述方法，各有其自身的优点和缺点。如图形化描述方法能够很直观地表示出软件系统的动态结构和风格，但在具体的动态行为描述上就受到一定的限制；进程代数方法能够详细描述系统的动态行为，逻辑方法在对体系结构风格的描述上比较困难。

7.5　动态体系结构的特征

与静态体系结构相比，动态体系结构具有可构造性、适应性、智能性等动态特征，本节对这些特性进行简单的介绍。

1. 可构造性动态特征

可构造性动态特征通常可以通过结合动态描述语言、动态修改语言和一个动态更新系统来实现。动态描述语言用于描述应用系统软件体系结构的初始配置；当体系结构发生改变的时候，这种改变可以通过动态修改语言进行描述，该语言支持增加或删除、激活或取消体系结构元素和系统遗留元素；而动态更新可以通过体系结构框架或者中间件实现。可构造性动态特征如图 7-11 所示。

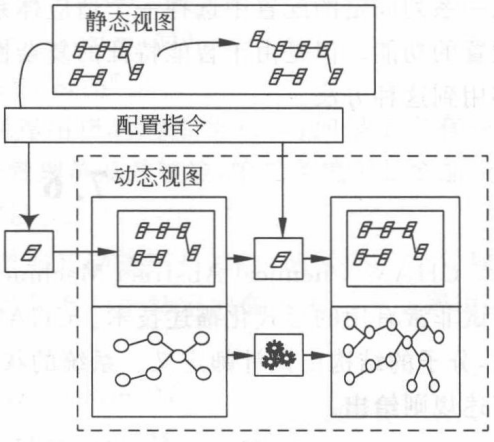


图 7-11　可构造性动态特征

软件体系结构的动态改变是经过一些事件触发开始的，通过追踪工具发出一个配置平衡信号追踪触发事件，系统维护人员开发出一个可选配置解决触发事件的追踪问题。在某些情况下要解决触发事件的追踪问题可能是困难的，此时，维护人员不得不经过反复的试验，通过增加优先候选事件来寻找适当的候选事件。这样，维护人员就可以通过发送配置指示给应用程序执行体系结构的改变。在应用程序中，这些指示充当体系结构的第一类请求。有些系统可以采用第一类请求的方法，但是另外一些

系统就不能选择第一类请求方法；此时，系统的动态改变是通过维护人员的预先的方法进行动态改变的评估和离线评估验证。

2. 适应性动态特征

某些应用程序必须有立即反映当前事件的能力，此时程序不能进行等待，必须把初始化、选择和执行整合到体系结构框架或中间件里面。

适应性动态特征是基于一系列预定义配置而且所有事件在开发期间已经进行了评估。执行期间，动态改变是通过一些预定义的事件集触发，体系结构选择一个可选的配置来执行系统的重配置。如图7-12所示描述了由事件触发改变的适应性动态特征。

3. 智能性动态特征

智能性动态特征是用一个有限的预配置集来移除约束。如图7-13所示，它描述的是一个具有智能性动态特征的应用程序体系结构。

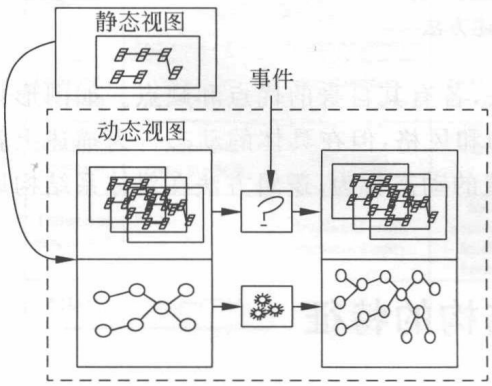


图7-12　适应性动态特征

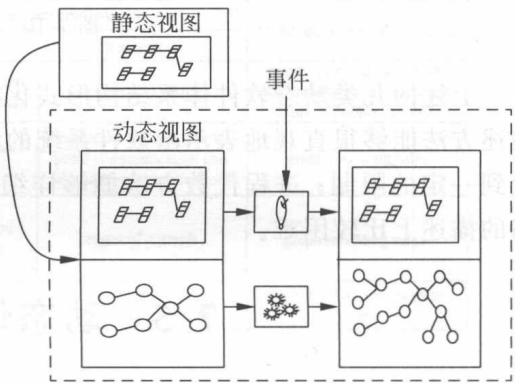


图7-13　智能性动态特征

对比适应性体系结构特征，智能性体系结构特征改善了选择转变的功能，适应性特征是从一系列固定的配置中选择一个适应体系结构的配置，而智能性特征是包括动态构造候选配置的功能。但是由于智能特征的复杂性，在实际的软件体系结构中并不是太多的系统能够用到这种方法。

7.6　化学抽象机

CHAM(Chemical Abstract Machine，化学抽象机)是一种对动态软件体系结构的分析、测试非常有用的形式化描述技术。CHAM模型把计算看作分子的两个数据元素之间的反应，分子的结构由设计师定义。系统的状态通过多个分子集来描述，因此，可能的反应通过下述规则给出。

$$m_1, m_2, \cdots, m_i \rightarrow m'_1, m'_2, \cdots, m'_j$$

其中，$m_{1\cdots i}$ 和 $m_{1\cdots j}$ 是分子。在应用此规则时，式子左边的分子可能用右边的分子取代。

1. 化学抽象机模型

通常，一个CHAM通过采用规则模式表示，而实际的规则存在于规则模式的实例中。在CHAM模型中并没有一个在每一时刻用于专门控制规则的机制，在同一时刻可能有多个规则被应用，此时，CHAM从这些非确定性的规则当中选取一个规则。

在CHAM模型中，任何一个方法通过膜操作后都可以看作是一个单一的分子。在膜范围内的一个方法可能是其他方法或一个分子操作依据的子规则。例如，如果有分子构造器的值恒为0和一元函数s，此时

$$s(\{|\ 0, s(0)\ |\}), 0$$

就是一个包含两个分子的方法。由于膜把分子封装起来了，这就迫使分子之间的反应只能是局部发生的。换而言之，在一个规则的内部，一个膜的演化与该方法的其他的膜演化是相互独立的。例如，一个CHAM包含以下规则：$0\rightarrow$，则该方法通过两个步骤的重写就可以转换为$s(\{|s(0)|\})$。

空锁操作$\lhd$能够从规则$s=m \uplus s$，构造出分子$m \lhd \{|s'|\}$，也就是说，空锁操作能够从一个规则中选出一个分子，并把剩余的分子用一个膜封装起来；该操作是可逆的，也就是说s可以通过$m \lhd \{|s'|\}$获得。例如，采用两次空锁操作可以把初始方案转换为：

$$o \lhd \{|\ s(\{|\ s(0) \lhd \{|\ 0\ |\}\ |\})\ |\}$$

2. 描述软件体系结构

在CHAM模型中可以把体系结构中的每一个构件(阶段)作为一个分子来表示其状态。根据它们的状态，特定的分子可以进行相互交互，交换数据(特征、符号等)。下面通过客户/服务器的例子来描述怎样把体系结构转换为CHAM模型。

分子结构通过下述文法给出，并给出了构件标识的精确语法。

$$\begin{aligned}
&\text{Molecule}:=\text{Component}\,|\,\text{Link}\,|\,\text{Command}\\
&\text{Component}:=\text{Id}:\text{Type}\\
&\text{Type}:=C\,|\,M\,|\,S\\
&\text{Link}:=\text{Id}-\text{Id}\\
&\text{Command}:=\text{cc}(\text{Component})\,|\,\text{rc}(\text{Id})
\end{aligned}$$

CHAM形式化描述客户-服务器体系结构风格如下。

$$\begin{aligned}
&\text{cc}(m:M)\rightarrow c:C,\ c-m,\ \text{cc}(m:M)\\
&\text{cc}(m:M)\rightarrow s:S,\ m-s,\ \text{cc}(m:M)\\
&s:S,\ \text{cc}(m:M)\rightarrow s:S,\ m:M
\end{aligned}$$

假设初始方法是由用户给出，方案包含用于创建用户名的命令cc()，如果至少有一台服务器已经运行，则上式的第一条规则是增加一个管理员及其链接，第二条规则是增加一个服务器及其链接，最后一条规则实际上是创建管理员。

现在来看客户-服务器体系结构的演化，除了增加和删除用户之外，同时也考虑管理员的创建和删除以及服务器的处理。每一个改变都必须有一个特定的命令来明确地调用，其过程是通过CHAM重配置的反应规则处理。

$$\begin{aligned}
&\text{cc}(c:C), m:M \rightarrow c:C, c-m, m:M\\
&\text{cc}(s:S), m:M \rightarrow s:S, m-s, m:M\\
&\text{rc}(c), c:C, c-m \rightarrow\\
&s':S, \text{rc}(s), s:S, m-s \rightarrow s':S\\
&m:M, \text{rc}(m), \text{cc}(m':M) \rightarrow m':M\\
&m-s, m':M \rightarrow m'-s, m':M\\
&c-m, m':M \rightarrow c-m', m':M
\end{aligned}$$

该方法分为4部分,第一部分的规则是对客户端和服务器的创建和移除的处理,第二部分的规则通过创建和删除命令指出处理管理员的取代,最后两条规则是把客户端存在的链接和给管理员。注意,不同的变量是通过不同标识符来标记的,而且右边的方法可以通过左边的方法实例化,也就是说这两个方法可以相互取代,但是是不可逆的。

这个CHAM例子演示了重配置,重配置命令仅出现在规则的左边,即命令仅通过CHAM识别,因此它必须被用户置入方法当中。作为一个重配置的例子,假设已经有一个具有单一服务器(具有管理员)的体系结构,想要增加一个客户端和替换管理员,用于CHAM演化的初始方法是:

$$m1:M, m1-s1, s1:S\mathrm{cc}(c1:C), \mathrm{rc}(m1), \mathrm{cc}(m2:M)$$

使得该方法的状态变为稳定步骤如下。

$$m1:M, m1-s1, s1:S, c1, :C, c1-m1, \mathrm{rc}(m1), \mathrm{cc}(m2:M)$$
$$m2:M, m1-s1, s1:S, c1:C, c1-m1$$
$$m2:M, m2-s1, s1:S, c1:C, c1-m1$$
$$m2:M, m2-s1, s1:S, c1:C, c1-m2$$

通常,为了确定形式化描述的正确性,提供一个CHAM模板是必要的,以使得一个稳定的方法能够实现。通常,CHAM模板对初始方法进行了一定的假设,正如前面的例子,假设初始方法包含一个分子cc(m:M)。此时,CHAM风格的选取的终止就变得很容易了(假设在规则选取成功):第三条规则通过命令cc()实现,也就是说任何一个规则都有一个触发命令是必要的,这样才能使得计算结束。

上述例子是对客户/服务器体系结构的描述,下面是CHAM对管道-过滤器体系结构风格的描述。

管道是一个具有in和out两个通信端口的复合构件,一个管道的内部又可以由一个管道和一个过滤器组成;过滤器具有3个端口:一个输入prev端口和两个输出端口。图7-14是对管道的描述。

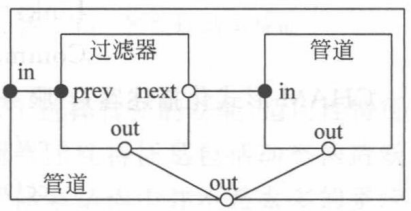


图7-14 管道结构描述

下面是采用CHAM对管道进行的描述。首先是定义分子语法,采用正整数作为构件的唯一标识符。

Molecule:=Component|Connection|Comman
Component:=Port|Number:Melecule
Port:=PortName/Molecule|PortName=|PortName=Message
PortName:=Number.Id|Id
Connection:=PortName−PortName
Command:=cp(Number)
Number:=1|1Number

接下来是用于描述两个构件之间的通信规则,第一条规则描述了一条消息通过端口p传输到端口q的过程和q准备接收此消息。

$$\mathrm{p}=msg, \mathrm{p}-\mathrm{q}, \mathrm{q} \Longrightarrow \mathrm{p}=, \mathrm{p}-\mathrm{q}, \mathrm{q}=msg$$

下面两条规则描述了信息允许通过膜在两个复合构件之间进行通信。第一条规则是使得端口在运行环境中是可视的,另一条规则是一旦接收端口收到消息后就把端口封闭。然

后，如果构件把消息处理完毕，则此规则就可以被再次使用。

$$n:\{|\ p=\lhd S\ |\}\rightarrow n.p=,n:S$$

$$n.p=\text{msg},n:S\rightarrow n:\{|\ p=\text{msg}\lhd S\ |\}$$

在这个例子中，最后一条通信规则是很重要的，一旦第一条消息被分送或接收，它就可以增加一个特定的方法到端口或系统。这个方法可能包含特定的重配置命令、新构件和连接。因此，它也可以推广到 Darwin 的动态构造中。

$p=\text{msg},p/\{|S|\}\rightarrow \text{p}=\text{msg},S$ 是一个创建管道的重配置命令，它也是一个必要的命令。

假设初始化方法包括分子 $cp(1)$，则创建最外层管道的规则如下。

```
cp(n)→n:{|in=,out=,
1n:{|prev=,next=,out=|},
in-1n.prev,1n.out-out,
1n.next/{|cp(11n),
1n.next-11n.in,11n.out-out|}|}
```

当过滤器通过端口 next 发送第一条消息，此时一个新的管道和连接就会被创建。假如有一个重配置计划，由于有 CP()出现在两边，因此体系结构的递归结构就可以立即从规则中体现出来。

CHAM 是一个简单的和易操作的设计模型，它有一个单一的数据结构和设计结构，而且都是使用人们所熟悉和直观的概念，也非常适合依赖内部状态的动态软件系统的演化描述。CHAM 是一种非常适合于动态软件体系结构描述、分析和测试的形式化方法。

思 考 题

1. 什么是动态软件体系结构？动态软件体系结构的动态性主要体现在哪 3 个方面？

2. 请简述基于构件的动态体系结构模型是如何支持运行系统的动态更新的？

3. πADL 采用运行时的视点对系统进行建模，它更侧重由构件、连接件和风格组成的动态软件体系结构，请用 πADL 对一个简单系统的体系结构或者体系结构的某一部分进行描述。

4. 对动态软件体系结构的描述方法包括图形化方法、进程代数方法和逻辑化描述方法，试用上述方法对一个简单系统的动态软件体系结构进行描述。

参考文献

第8章 基于服务的体系结构

迄今为止，对于面向服务的体系结构(Service-Oriented Architecture，SOA)还没有一个公认的定义。许多组织从不同的角度和不同的侧面对 SOA 进行了描述，较为典型的有以下 3 个。

(1) W3C 的定义：SOA 是一种应用程序体系结构，在这种体系结构中，所有功能都定义为独立的服务，这些服务带有定义明确的可调用接口，能够以定义好的顺序调用这些服务来形成业务流程。

(2) Service-architecture. com 的定义：服务是精确定义、封装完整、独立于其他服务所处环境和状态的函数。SOA 本质上是服务的集合，服务之间彼此通信，这种通信可能是简单的数据传送，也可能是两个或更多的服务协调进行某些活动。服务之间需要某些方法进行连接。

(3) Gartner 的定义：SOA 是一种 C/S 体系结构的软件设计方法，应用由服务和服务使用者组成，SOA 与大多数通用的 C/S 体系结构模型的不同之处，在于它着重强调构件的松散耦合，并使用独立的标准接口。

8.1 SOA 概述

SOA 是一种在计算环境中设计、开发、部署和管理离散逻辑单元(服务)模型的方法。SOA 并不是一个新鲜事物，而只是面向对象模型的一种替代。虽然基于 SOA 的系统并不排除使用 OOD 构建单个服务，但是其整体设计却是面向服务的。由于 SOA 考虑到了系统内的对象，所以虽然 SOA 是基于对象的，但是作为一个整体，它却不是面向对象的。

SOA 系统原型的一个典型例子是 CORBA，它已经出现很长时间，其定义的概念与 SOA 相似。SOA 建立在 XML 等新技术的基础上，通过使用基于 XML 的语言描述接口，服务已经转到更动态且更灵活的接口系统中，CORBA 中的 IDL 无法与之相比。图 8-1 描述了一个完整的 SOA 模型。

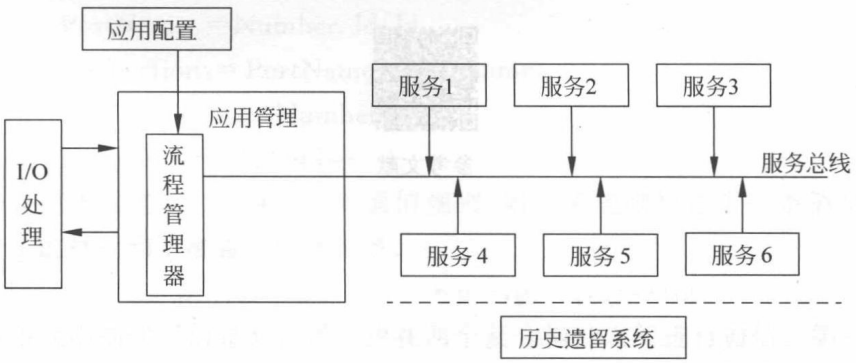


图 8-1 SOA 模型示例

在 SOA 模型中，所有的功能都定义成了独立的服务。服务之间通过交互和协调完成业务的整体逻辑。所有的服务通过服务总线或流程管理器连接。这种松散耦合的体系结构使得各服务在交互过程中无须考虑双方的内部实现细节，以及部署在什么平台上。

1. 服务的基本结构

一个独立的服务基本结构如图 8-2 所示。

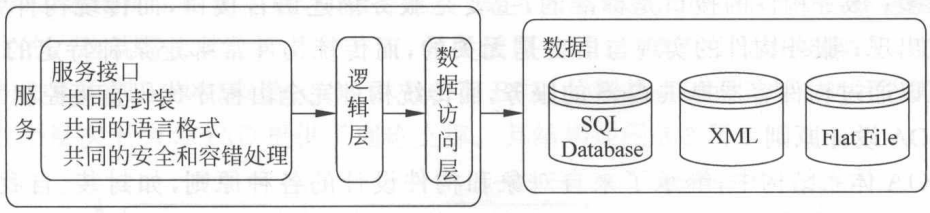


图 8-2　单个服务内部结构

由图 8-2 可以看出，服务模型的表示层从逻辑层分离出来，中间增加了服务对外的接口层。通过服务接口的标准化描述，使得服务可以提供给在任何异构平台和任何用户接口使用。这允许并支持基于服务的系统成为松散耦合、面向构件和跨技术实现，服务请求者很可能根本不知道服务在哪里运行、由哪种语言编写，以及消息的传输路径，而是只要提出服务请求，然后就会得到答案。

2. SOA 的特征

SOA 是一种粗粒度、松耦合的服务体系结构，其服务之间通过简单、精确定义接口进行通信，不涉及底层编程接口和通信模型。这种模型具有下面几个特征。

(1) 松散耦合。SOA 是松散耦合构件服务，这一点区别于大多数其他的构件体系结构。松散耦合旨在将服务使用者和服务提供者在服务实现和客户如何使用服务方面隔离。服务提供者和服务使用者之间松散耦合的关键点是服务接口作为与服务实现分离的实体而存在。这是服务实现能够在完全不影响服务使用者的情况下进行修改。大多数松散耦合方法都依靠基于服务接口的消息，基于消息的接口能够兼容多种传输方式(如 HTTP、TCP/IP、MOM 等)，基于消息的接口可以采用同步或异步协议实现。

(2) 粗粒度服务。服务粒度(Service Granularity)指的是服务所公开功能的范围，一般分为细粒度和粗粒度，其中，细粒度服务是能够提供少量业务流程可用性的服务。粗粒度服务是能够提供高层业务逻辑的可用性服务。选择正确的抽象级别是 SOA 建模的一个关键问题。设计应该在不损失或损坏相关性、一致性和完整性的情况下，尽可能地进行粗粒度建模。通过一组有效设计和组合的粗粒度服务，业务专家能够有效地组合出新的业务流程和应用程序。

(3) 标准化接口。SOA 通过服务接口的标准化描述，使得该服务可以在任何异构平台和任何用户接口中使用。这一描述囊括了与服务交互需要的全部细节，包括消息格式、传输协议和位置。该接口隐藏了实现服务的细节，允许独立于实现服务基于的硬件或软件平台和编写服务所用的编程语言使用服务。

3. 服务构件与传统构件

服务构件体系结构(Service Component Architecture，SCA)是基于 SOA 的思想描述服务之间组合和协作的规范，它描述用于使用 SOA 构建应用程序和系统的模型。它可简化

使用 SOA 进行的应用程序开发和实现工作。SCA 提供了构建粗粒度构件的机制,这些粗粒度构件由细粒度构件组装而成。SCA 将传统中间件编程从业务逻辑分离,从而使程序员免受其复杂性的困扰。它允许开发人员集中精力编写业务逻辑,而不必将大量的时间花费在更为底层的技术实现上。

SCA 服务构件与传统构件的主要区别在于,服务构件往往是粗粒度的,而传统构件以细粒度居多;服务构件的接口是标准的,主要是服务描述语言接口,而传统构件常以具体 API 形式出现;服务构件的实现与语言是无关的,而传统构件常绑定某种特定的语言;服务构件可以通过构件容器提供 QoS 的服务,而传统构件完全由程序代码直接控制。

4. SOA 设计原则

在 SOA 体系结构中,继承了来自对象和构件设计的各种原则,如封装、自我包含等。那些保证服务的灵活性、松散耦合和复用能力的设计原则,对 SOA 体系结构同样非常重要。关于服务,常见的设计原则如下。

(1) 明确定义的接口。服务请求者依赖于服务规约调用服务,因此,服务定义必须长时间稳定,一旦公布,不能随意更改;服务的定义应尽可能明确,减少请求者的不适当使用;不要让请求者看到服务内部的私有数据。

(2) 自包含和模块化。服务封装了在业务上稳定、重复出现的活动和构件,实现服务的功能实体完全独立自主,独立进行部署、版本控制、自我管理和恢复。

(3) 粗粒度。服务数量不应该太多,依靠消息交互而不是远程过程调用,通常消息量比较大,但是服务之间的交互频度较低。

(4) 松耦合。服务请求者可见的是服务的接口,其位置、实现技术、当前状态、私有数据等,对服务请求者而言是不可见的。

(5) 互操作性、兼容和策略声明。为了确保服务规约的全面和明确,策略成为一个越来越重要的方面。这可以是技术相关的内容,例如,一个服务对安全性方面的要求;也可以是与业务有关的语义方面的内容,例如,需要满足的费用或者服务级别方面的要求,这些策略对于服务在交互时是非常重要的。

8.2 面向服务的分析与设计

从概念上讲,SOA 有 3 个主要的抽象级别,分别是操作、服务和业务流程。其中,位于底层的操作代表了单个逻辑单元的事物。执行操作通常会导致读、写或修改一个或多个持久性数据。SOA 操作可以直接与面向对象的方法相比,它们都有特定的结构化接口,并且返回结构化的响应,完全同方法一样。位于第二层的服务代表了操作的逻辑分组。例如,如果将客户管理视为服务,则按照电话号码查找客户、按照名称和邮政编码列出客户和保存新客户的数据就代表相关操作。最高层的业务流程则是为了实现特定业务目标而执行的一组长期运行的动作或活动。业务流程通常包括多个业务调用。在 SOA 术语中,业务流程包括依据一组业务规则按照有序序列执行的一系列操作。其中,操作的排序、选择和执行成为服务或流程的编排,典型的情况是调用已编排服务响应业务事件。

从建模的观点来看,SOA 带来的主要挑战是,如何描述设计良好的操作、服务和流程抽象的特征,以及如何系统地构造它们。针对这个问题,Olaf Zimmermann 和 Pal Krogdahl

综合了面向对象的分析与设计(OOAD)、企业体系结构(EA)框架和业务流程建模(BPM)中的适当原理,将这些规则中的原理与许多独特的新原理组合起来,提出了面向服务的分析与设计(Service-Oriented Analysis and Design,SOAD)的概念。

SOA 实现项目经验表明,诸如 OOAD、EA 和 BPM 这样的现有开发流程和表示法仅涵盖支持 SOA 范式所需要的部分要求。SOA 方法在加强已经制定的良好通用软件体系结构原则的同时,还增加了附加主题,如服务编排、服务库服务总线中间件模式,在建模时是需要特别关注的。这就需要整合这 3 种方法,保留适用的理论,摒弃不适用的地方,并且融入一些新的方法和原则。总的来说,OOAD、EA 和 BPM 分别从基础设计层、应用结构层和业务组织层 3 个层次上为 SOAD 提供了理论支撑。其结构如图 8-3 所示。

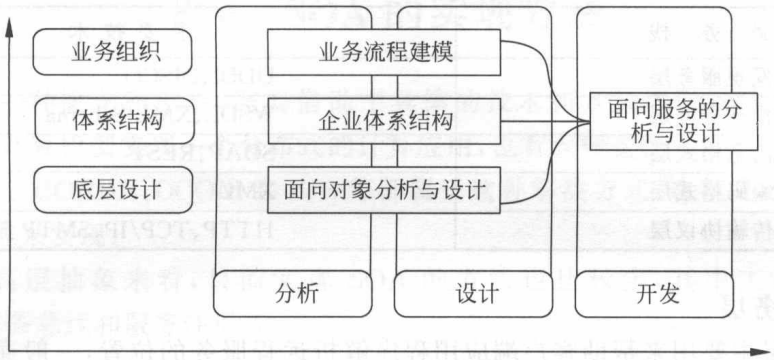


图 8-3 SOAD 结构图

1. 基础设计层

SOAD 的第一层是基础设计层,它采用了 OOAD 的思想,其主要目标是能够进行快速而有效的设计、开发以及执行灵活且可扩展的底层服务构件。对于设计已定义的服务中的底层类和构件结构,OO 是一种很有价值的方法。目前与 SOAD 有关的 OO 设计在实践中也存在着一些问题:OO 的粒度级别集中在类级,对于业务服务建模来说,这样的抽象级别太低。类似继承这样的强关联产生了相关方之间一定程度的紧耦合。与此相反,SOAD 试图通过松耦合促进灵活性和敏捷性。这使得 OO 难以与 SOAD 体系结构保持一致。诸如这些问题有待进一步解决,尽管如此,OO 还是为 SOAD 提供了丰富的理论源泉。

2. 体系结构层

SOAD 第二层是体系结构层,它采用了 EA 的理论框架。企业应用程序和 IT 基础体系结构发展成 SOA 是一个庞大的工程,其中可能涉及众多的业务流水线和组织单元。因此,需要应用 EA 框架和参考体系结构,以努力实现单独的解决方案之间体系结构的一致性。在 SOA 中,体系结构层必须以表示业务服务的逻辑构件为中心,并且集中于定义服务之间的接口和服务级协定。

3. 业务层

SOAD 第三层是业务层,它采用了 BPM 规则。BPM 是一个不完整的规则,包含许多不同的形式、表示法和资源,其中应用较为广泛的是 UML。SOA 必须利用所有现有的 BPM 方法作为 SOAD 的起点,同时需要服务流程编排模型中用于驱动候选服务和它们的操作的附加技术对其加以补充。此外,SOAD 中的流程建模必须与设计层用例保持同步。

SOAD 以 OOAD、EA 和 BPM 为基础,为 SOA 体系结构的业务和 IT 实现之间搭建了

一座桥梁,并且为SOA项目的分析和设计提供了一套理论方法。随着实践的深入,SOAD有待于在理论和实践上加以完善。

8.3 SOA的关键技术

SOA是一种全新的体系结构,为了支持其各种特性,相关的技术规范不断被推出。服务要以一种可互操作的方式执行发布、发现和绑定这3个操作,必须有一个包含每一层标准的服务栈。因此,整个SOA的技术系列被称为服务栈,如表8-1所示。

表8-1 服务栈

服 务 栈	主 要 技 术
发现服务层	UDDI、DISCO
描述服务层	WSDL、XML Schema
消息格式层	SOAP、REST
编码格式层	XML
传输协议层	HTTP、TCP/IP、SMTP等

1. 发现服务层

发现服务层主要用来帮助客户端应用程序解析远程服务的位置,一般通过UDDI来实现。UDDI规范由Microsoft、IBM和Ariba三家公司在2000年7月提出,它是服务的信息注册规范,以便被需要该服务的用户发现和使用它。UDDI规范描述了服务的概念,同时也定义了一种编程接口。通过UDDI提供的标准接口,企业可以发布自己的服务供其他企业查询和调用,也可以查询特定服务的描述信息,并动态绑定到该服务上。通过UDDI,Web服务可以真正实现信息的"一次注册到处访问"。

2. 描述服务层

描述层为客户端应用程序提供正确地与远程服务交互的描述信息,主要通过WSDL来实现。与UDDI一样,WSDL也是由Microsoft、IBM和Ariba三家公司在2000年7月提出的。WSDL为服务提供者提供以XML格式描述服务请求的标准格式,将网络服务描述为能够进行消息交换的通信端点集合,以表达一个服务能做什么,它的位置在哪,如何调用它等信息。

3. 消息格式层

消息格式层主要用来保证客户端应用程序和服务器端在格式设置上保持一致,一般通过SOAP来实现。SOAP定义了服务请求者和服务提供者之间的消息传输规范。SOAP用XML来格式化消息,用HTTP来承载消息。SOAP包括3个部分:定义了描述消息和如何处理消息的框架的封装(SOAP封装)、表达应用程序定义的数据类型实例的编码规则(SOAP编码规则),以及描述远程过程调用和应答的协议(SOAP RPC表示)。

4. 编码格式层

编码格式层主要为客户端和服务器之间提供一个标准的、独立于平台的数据交换编码格式,一般通过XML来实现。XML是一种元语言,可以用来定义和描述结构化数据。XML使用基于文本的、利用标准字符集的编码方案,从而避开了二进制编码的平台不兼容

问题。XML 有很多优点，包括跨平台支持，公用类型系统和对行业标准字符集的支持，它是服务得以实现的语言基础。服务的其他协议规范都是以 XML 形式来描述和表达的。

5. 传输协议层

传输协议层主要为客户端和服务器之间提供两者交互的网络通信协议，一般通过 HTTP（Hypertext Transfer Protocol，超文本协议）和 SMTP（Simple Mail Transport Protocol，简单邮件传输协议）来实现。HTTP 是一个在 Internet 广泛使用的协议，为服务部件通过 Internet 交互奠定了协议基础，并具有穿透防火墙的良好特性。SMTP 则适合于异步通信，如果服务中断，SMTP 可以自动进行重试。

8.4 SOA 的实现方法

SOA 只是一种概念和思想，需要借助于具体的技术和方法来实现它。从本质上看，SOA 是用本地计算模型实现一个分布式的计算应用，也有人称这种方法为“本地化设计，分布式工作”模型。CORBA、DCOM 和 EJB 等都属于这种解决方式，也就是说，SOA 最终可以基于这些标准来实现。

从逻辑和高层抽象来看，目前实现 SOA 的方法也比较多，其中主流方式有 Web Service、企业服务总线和服务注册表。

1. Web Service

在 Web Service（Web 服务）的解决方案中，一共有 3 种工作角色，其中服务提供者和服务请求者是必需的，服务注册中心是可选的。它们之间的交互和操作构成了 SOA 的一种实现体系结构，如图 8-4 所示。

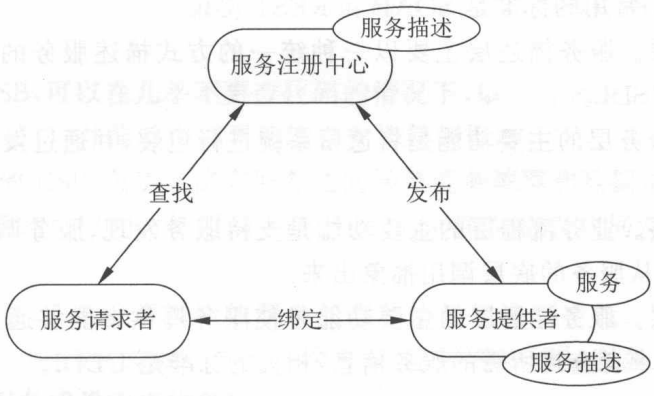


图 8-4　Web Service 模型

（1）服务提供者。服务提供者是服务的所有者，该角色负责定义并实现服务，使用 WSDL 对服务进行详细、准确、规范的描述，并将该描述发布到服务注册中心，供服务请求者查找并绑定使用。

（2）服务请求者。服务请求者是服务的使用者，虽然服务面向的是程序，但程序的最终使用者仍然是用户。从体系结构的角度看，服务请求者是查找、绑定并调用服务，或与服务进行交互的应用程序。服务请求者角色可以由浏览器担当，由人或程序（如另外一个服务）控制。

(3) 服务注册中心。服务注册中心是连接服务提供者和服务请求者的纽带,服务提供者在此发布他们的服务描述,而服务请求者在服务注册中心查找他们需要的服务。在某些情况下,服务注册中心是整个模型中的可选角色。例如,如果使用静态绑定的服务,服务提供者则可以把描述直接发送给服务请求者。

Web Service 模型中的操作包括发布、查找和绑定,这些操作可以单次或反复出现。

(1) 发布。为了使用户能够访问服务,服务提供者需要发布服务描述,以便服务请求者可以查找它。

(2) 查找。在查找操作中,服务请求者直接检索服务描述或在服务注册中心查询所要求的服务类型。对服务请求者而言,可能会在生命周期的两个不同阶段中涉及查找操作,首先是在设计阶段,为了程序开发而查找服务的接口描述;其次是在运行阶段,为了调用而查找服务的位置描述。

(3) 绑定。在绑定操作中,服务请求者使用服务描述中的绑定细节来定位、联系并调用服务,从而在运行时与服务进行交互。绑定可以分为动态绑定和静态绑定。在动态绑定中,服务请求者通过服务注册中心查找服务描述,并动态地与服务交互;在静态绑定中,服务请求者已经与服务提供者达成默契,通过本地文件或其他方式直接与服务进行绑定。

在采用 Web Service 作为 SOA 的实现技术时,应用系统大致可以分为 6 个层次,分别是底层传输层、服务通信协议层、服务描述层、服务层、业务流程层和服务注册层。

(1) 底层传输层。底层传输层主要负责消息的传输机制,HTTP、JMS(Java Messaging Service,Java 消息服务)和 SMTP 都可以作为服务的消息传输协议,其中 HTTP 使用最广泛。

(2) 服务通信协议层。服务通信协议层的主要功能是描述并定义服务之间进行消息传递所需的技术标准。常用的标准是 SOAP 和 REST 协议。

(3) 服务描述层。服务描述层主要以一种统一的方式描述服务的接口与消息交换方式,相关的标准是 WSDL。

(4) 服务层。服务层的主要功能是将遗留系统进行包装,并通过发布的 WSDL 接口描述被定位和调用。

(5) 业务流程层。业务流程层的主要功能是支持服务发现,服务调用和点到点的服务调用,并将业务流程从服务的底层调用抽象出来。

(6) 服务注册层。服务注册层的主要功能是使服务提供者能够通过 WSDL 发布服务定义,并支持服务请求者查找所需的服务信息,相关的标准是 UDDI。

2. 服务注册表

服务注册表(service registry)虽然也具有运行时的功能,但主要在 SOA 设计时使用。它提供一个策略执行点(Policy Enforcement Point,PEP),在这个点上,服务可以在 SOA 中注册,从而可以被发现和使用。服务注册表可以包括有关服务和相关构件的配置、依从性和约束文件。从理论上来说,任何帮助服务注册、发现和查找服务合约、元数据和策略的信息库、数据库、目录或其他结点都可以被认为是一个注册表。大多数商用服务注册产品支持服务注册、服务位置和服务绑定功能。

(1) 服务注册。服务注册是指服务提供者向服务注册表发布服务的功能(服务合约),包括服务身份、位置、方法、绑定、配置、方案、策略等描述性属性。使用服务注册表实现

SOA 时，要限制哪些新服务可以向注册表发布，由谁发布，以及谁批准和根据什么条件批准等，以便使服务能够有序地注册。

(2) 服务位置。服务位置是指服务使用者，帮助它们查询已注册的服务，寻找符合自身要求的服务。这种查找主要通过检索服务合约来实现，在使用服务注册表实现 SOA 时，需要规定哪些用户可以访问服务注册表，以及哪些服务属性可以通过服务注册表进行暴露等，以便服务能得到有效的、经过授权的使用。

(3) 服务绑定。服务使用者利用查找到的服务合约来开发代码，开发的代码将与注册的服务进行绑定，调用注册的服务，以及与它们实现互动。可以利用集成的开发环境自动将新开发的服务与不同的新协议、方案和程序间通信所需的其他接口绑定。

3. 企业服务总线

ESB 的概念是从 SOA 发展而来，它是一种为进行连接服务提供的标准化的通信基础结构，基于开放的标准，为应用提供了一个可靠的、可度量的和高度安全的环境，并可帮助企业对业务流程进行设计和模拟，对每个业务流程实施控制和跟踪、分析并改进流程和性能。

在一个复杂的企业计算环境中，如果服务提供者和服务请求者之间采用直接的端到端的交互，那么随着企业信息系统的增加和复杂度的提高，系统之间的关联会逐渐变得非常复杂，形成一个网状结构，这将带来昂贵的系统维护费用，同时也使得 IT 基础设施的复用变得困难重重。ESB 提供了一种基础设施，消除了服务请求者与服务提供者之间的直接连接，使得服务请求者与服务提供者之间进一步解耦。

ESB 是由中间件技术实现并支持 SOA 的一组基础体系结构，是传统中间件技术与 XML、Web Service 等技术结合的产物，是在整个企业集成体系结构下的面向服务的企业应用集成机制。具体来说，ESB 具有以下功能。

(1) 支持异构环境中的服务、消息和基于事件的交互，并且具有适当的服务级别和可管理性。

(2) 通过使用 ESB，可以在几乎不更改代码的情况下，以一种无缝的非侵入方式使现有系统具有全新的服务接口，并能够在部署环境中支持任何标准。

(3) 充当缓冲器的 ESB(负责在诸多服务之间转换业务逻辑和数据格式)与服务逻辑相分离，从而使不同的系统可以同时使用同一个服务，无须在系统或数据发生变化时，改动服务代码。

(4) 在更高的层次，ESB 还提供诸如服务代理和协议转换等功能。允许在多种形式下通过像 HTTP、SOAP 和 JMS 总线的多种传输方式，主要是以网络服务的形式，为发表、注册、发现和使用企业服务或界面提供基础设施。

(5) 提供可配置的消息转换翻译机制和基于消息内容的消息路由服务，传输消息到不同的目的地。

(6) 提供安全和拥有者机制，以保证消息和服务使用的认证、授权和完整性。

在企业应用集成方面，与现存的、专有的集成解决方案相比，ESB 具有以下优势。

(1) 扩展的、基于标准的连接。ESB 形成一个基于标准的信息骨架，使得在系统内部和整个价值链中可以容易地进行异步或同步数据交换。ESB 通过使用 XML、SOAP 和其他标准，提供了更强大的系统连接性。

(2) 灵活的、服务导向的应用组合。基于 SOA，ESB 使复杂的分布式系统(包括跨多个

应用、系统和防火墙的集成方案)能够由以前开发测试过的服务组合而成,使系统具有高度可扩展性。

(3) 提高复用率,降低成本。按照SOA方法构建应用,提高了复用率,简化了维护工作,进而减少了系统总体成本。

(4) 减少市场反应时间,提高生产率。ESB通过构件和服务复用,按照SOA的思想简化应用组合,基于标准的通信、转换和连接实现这些优点。

8.5 服务描述语言

8.3节已经指出,服务是一个协议栈,其中包含很多相关的协议,这些协议包含对服务的调用、描述、发布、寻找、管理、安全等。其中最重要的也是最成熟的3个协议是SOAP、WSDL和UDDI。掌握了这3个协议,就可以构造完整的Web服务。本节将详细介绍WSDL协议,8.6节将详细介绍UDDI协议,8.7节将详细介绍SOAP协议。

在讨论协议之前,先给出一个具体的服务的例子,后面对于WSDL、UDDI和构造服务的方法都会用到这个例子。许多经典的语言教程都是以"Hello World"开始的,因此,本章将第一个服务设计为"Hello your name"。这个服务需要服务使用者传入string类型的参数,并返回一个形式为"Hello <string>"的字符串。

8.5.1 WSDL概述

WSDL是对服务进行描述的语言,它有一套基于XML的语法定义。WSDL描述的重点是服务,它包含Service Implementation Definition(服务实现定义)和Service Interface Definition(服务接口定义),如图8-5所示。

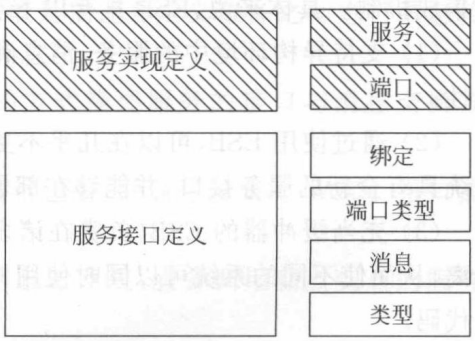


图8-5 基本服务描述

采用抽象接口定义对于提高系统的扩展性很有帮助。服务接口定义就是一种抽象的、可重用的定义,行业标准组织可以使用这种抽象的定义规定一些标准的服务类型,服务实现者可以根据这些标准定义来实现具体的服务。这就好比在Java中定义的一个抽象接口可以有多个实现一样。

服务实现定义描述了给定服务提供者如何实现特定的服务接口。服务实现定义中包含服务和端口描述。服务描述了一个特定的Web服务所提供的所有访问入口的部署细节,一个服务往往会包含多个服务访问入口,而每个访问入口都会使用一个端口元素来描述;端口描述的是一个服务访问入口的部署细节,包括通过哪个URL访问,应当使用怎样的消息调用模式访问等。

服务接口定义和服务实现定义结合在一起,组成了完整的WSDL定义。

8.5.2 使用WSDL文档

WSDL文档是按照WSDL语法规范描述某个特定服务的文档。WSDL文档是一个简单的XML文档,有些工具可以自动生成服务的WSDL文档,开发者也可以根据自己的需

要书写 WSDL 文档。

虽然说可以把 WSDL 文档看作服务的 SDK，但开发者并不直接使用 WSDL 文档，WSDL 文档是由程序使用的，这也是服务的重要优点。程序级的利用可以让开发者书写出更灵活的代码，更便于不同程序之间的集成。

具体来说，有两个不同的时期来使用 WSDL 文档，一个是在编写调用服务的客户端程序的设计时期，一个是在程序运行时期。

在设计时期，开发者根据 WSDL 文档生成调用 Web 服务的客户端代码，可以将这样的代码称为服务代理类，实际的服务使用都发生在代理类与服务之间。在编写客户应用程序的过程中，开发者就像使用本地类一样直接使用服务代理类，开发客户端程序。图 8-6 表明了这种使用方式的流程。

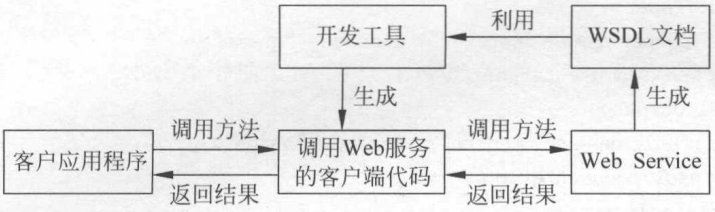


图 8-6　设计期 WSDL 文档的使用方法

在运行期，客户应用程序通过它所处的运行环境发出对服务的调用请求，运行环境根据 WSDL 文档生成正确的服务调用请求，并接受 Web Service 返回的结果，然后把结果传递给客户应用程序。图 8-7 表明了这种使用方法的流程。

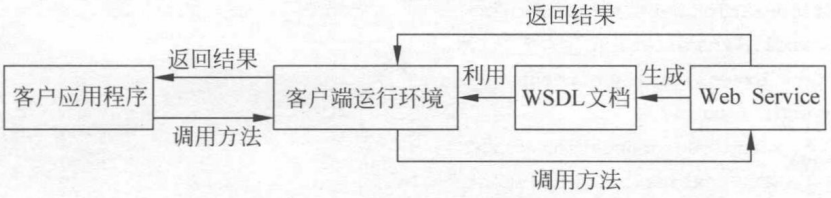


图 8-7　运行期 WSDL 文档的使用方法

8.5.3　WSDL 文档结构

WSDL 文档是一个按照严格规范完成的 XML 文档，WSDL 规范就是这个 XML 文档的规范。一个标准的 WSDL 文档形式如下。

```
<wsdl:definitions name = "nmtoken"? targetNamespace = "uri">
<import namespace = "uri" location = "url"/>*

<wsdl:documentation .../>?

<wsdl:types>?
<wsdl:documentation ... />?
<xsd:schema .../>*
<-- extensibility element -->*
```

```
</wsdl:types>

<wsdl:message name = "nmtoken">*
<wsdl:documentation ... />?
<part name = "nmtoken" element = "qname"? type = "qname"?/>*
</wsdl:message>

    <wsdl:portType name = "nmtoken">*
    <wsdl:documentation ... />?
    <wsdl:operation name = "nmtoken">*
        <wsdl:documentation ... />?
        <wsdl:input name = "nmtoken"? message = "qname">?
            <wsdl:documentation .../>?
        </wsdl:input>
        <wsdl:output name = "nmtoken"? message = "qname">?
            <wsdl:documentation .../>?
        </wsdl:output>
        <wsdl:fault name = "nmtoken"? message = "qname">?
            <wsdl:documentation .../>?
        </wsdl:fault>
    </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name = "nmtoken" type = "qname">*
    <wsdl:documentation .../>?
    <-- extensibility element -->*
    <wsdl:operation name = "nmtoken">*
        <wsdl:documentation .../>?
        <-- extensibility element -->*
        <wsdl:input>?
            <wsdl:documentation ... />?
            <-- extensibility element -->
        </wsdl:input>
        <wsdl:output>?
            <wsdl:documentation .../>?
            <-- extensibility element -->*
        </wsdl:output>
        <wsdl:fault name = "nmtoken">?
            <wsdl:documentation .../>?
            <-- extensibility element -->*
        </wsdl:fault>
    </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name = "nmtoken">*
    <wsdl:documentation .../>?
    <wsdl:port name = "nmtoken" binding = "qname">*
        <wsdl:documentation .../>?
        <-- extensibility element -->
    </wsdl:port>
```

```
    <-- extensibility element -->
    </wsdl:service>

    <-- extensibility element -->*

    </wsdl:definitions>
```

WSDL 文档的根元素是<definitions>，在根元素之下包含若干子元素，这些子元素包括如下几个。

(1) types(类型)：数据类型定义的容器，提供了用于描述正在交换的消息的数据类型定义，一般使用 XML Schema 中的类型系统。

(2) message(消息)：通信消息数据结构的抽象定义。message 使用 types 所定义的类型来定义整个消息的数据结构。

(3) operation(操作)：对服务中所支持的操作的抽象描述，一般单个 operation 描述了一对访问入口的请求/响应消息。

(4) portType(端口类型)：描述了一组操作，每个操作指向一个输入消息和多个输出消息规范。

(5) binding(绑定)：为特定端口类型定义的操作和消息指定具体的协议和数据格式。

(6) port(端口)：指定用于绑定的地址，定义服务访问点。

(7) service：相关服务访问点的集合。

各个元素之间的逻辑关系如图 8-8 所示。

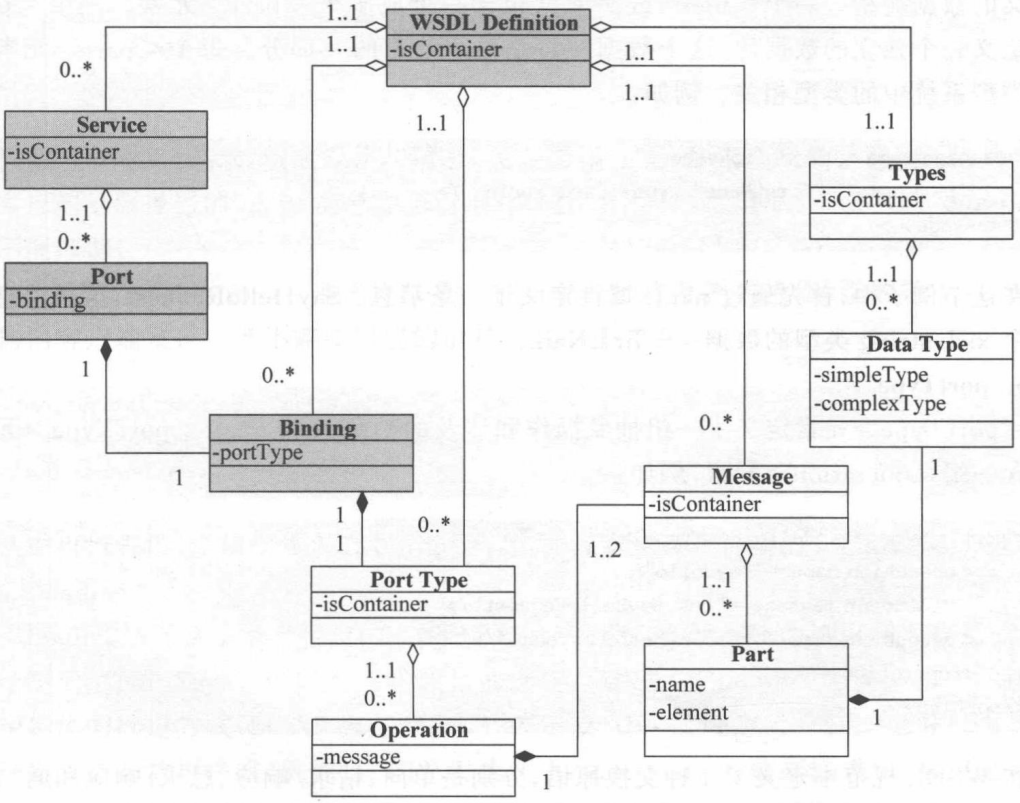


图 8-8　逻辑关系图

下面简单介绍各元素的作用与书写规范。

1. definitions

<definitions>元素是WSDL文档的根元素,任何WSDL文档有且仅有一个<definitions>元素。在<definitions>元素中可以定义全局的命名空间(namespace)。

2. import

<import>元素紧随<definitions>之后出现,<import>元素与C++中的#include的功能类似,可以引用其他的文档。通过<import>元素,用户可以把整个WSDL文档分成独立的若干部分,然后通过<import>连接起来。这对于一个很大的WSDL文档来说,既便于编写也便于维护。

3. types

<types>元素是一个容器类型的元素,在<types>元素中可以定义一系列的数据类型供<message>使用。目前,XSD(XML Schema Definitions,XML样式定义)类型系统是最常用的数据类型系统,在<types>元素中既可以沿用XSD类型系统,也可以根据需要定义新的类型。例如,下面就定义了一个简单类型——Reference。

```
<xsd:simpleType name = "Reference">
    <xsd:restriction base = "xsd:string" />
</xsd:simpleType>
```

4. message

<message>元素用来为数据交换建立模型,在<message>元素中将引用在<types>中定义的数据类型。一个<message>元素包含一个或多个<part>元素。一个<part>元素定义一个独立的数据片,这个数据片将是整个消息的一部分。每个<part>元素都与某种类型系统中的类型相关。例如:

```
<message name = "SayHelloRequest">
    <part name = "firstName" type = "xsd:string"/>
</message>
```

在这个例子中,首先通过name属性定义了一条消息:SayHelloRequest,这条消息中包含一个xsd:string类型的数据——firstName。

5. portType

<portType>元素定义了一组抽象操作和涉及的抽象消息,一个<portType>元素可以包含一组<operation>元素,例如:

```
<portType name = "Hello_PortType">
    <operation name = "sayHello">
        <input message = "tns:SayHelloRequest"/>
    <output message = "tns:SayHelloResponse"/>
    </operation>
</portType>
```

在WSDL规范中定义了4种交换原语,分别是单向、请求/响应、恳求/响应和通知。

(1) 单向(one-way)。单向操作指的是Web Service客户端向服务器发送一条不要求

服务器回应的消息，即只有 input 消息，而没有 output 消息。其语法如下。

```
<wsdl:operation name = "nmtoken">
    <wsdl:input name = "nmtoken"? message = "qname"/>
</wsdl:operation>
```

(2) 请求/响应(request-response)。请求/响应操作指的是客户端向服务器发送一条请求，期望服务器同步地返回该请求的响应，这种操作模式非常类似于过程调用，在服务器没有返回响应前，客户端会一直阻塞。其语法如下。

```
<wsdl:operation name = "nmtoken">
    <wsdl:input name = "nmtoken"? message = "qname"/>
    <wsdl:output name = "nmtoken"? message = "qname"/>
    <wsdl:fault name = "nmtoken"? message = "qname"/>*
</wsdl:operation>
```

注意：在请求/响应操作中，input 消息必须在 output 消息之前出现。

(3) 恳求/响应(solicit-response)。恳求/响应操作虽然在名字上与请求/响应操作类似，但其实完全不一样。恳求/响应操作指的是服务器向客户端恳求得到客户端的响应。换句话说，服务器首先向客户端发送一条请求，客户端在接收到请求之后给予响应。其语法如下。

```
<wsdl:operation name = "nmtoken">
    <wsdl:output name = "nmtoken"? message = "qname"/>
    <wsdl:input name = "nmtoken"? message = "qname"/>
    <wsdl:fault name = "nmtoken"? message = "qname"/>*
</wsdl:operation>
```

从中可以看出，恳求/响应操作与请求/响应操作互逆，首先出现的消息是 output，然后才是客户端给服务器的 input 消息。与请求/响应操作一样，这里面的 input 和 output 的顺序也不能颠倒。

(4) 通知(notification)。通知操作是一种与单向操作互逆的操作，指的是 Web Service 服务器向客户端发送一条不需要回应的消息。其语法如下。

```
<wsdl:operation name = "nmtoken">
    <wsdl:output name = "nmtoken"? message = "qname"/>
</wsdl:operation>
```

从中可以看出，通知操作仅有 output 消息，而没有来自客户端的 input 消息。

6. binding

<binding>元素是对于特定的<portType>元素定义的具体的协议和数据格式，对于一个给定的<portType>可以有任意数目的绑定。通过<binding>元素中指定的协议可以确定具体的绑定方式(SOAP 绑定、HTTP 绑定或 MIME 绑定)。同时，也可以由指定的协议确定具体的数据格式(例如，HTTP 对应了 header/body 的数据格式)。绑定是调用服务的一个重要操作，不同的绑定方式有各自的特点，其中以 SOAP 绑定最为常用。

7. service

虽然在<binding>元素中非常详细地指明了服务的访问方法(协议和数据格式),但并没有给出访问服务的实际URL。在<service>元素中将给出访问服务的URL并完成整个服务的描述。在<service>元素中可以包含一个或多个<port>元素,每一个<port>元素都表明了一个服务访问点,如下所示。

```
<service name = "Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
<port binding = "tns:Hello_Binding" name = "Hello_Port">
        <soap:address location = "http://localhost:8080/soap/servlet/rpcrouter"/>
    </port>
</service>
```

注意:一些WSDL文档中根本就没有<service>元素。这是因为,WSDL文档的目标是描述Web Service的抽象接口,而<service>元素描述的是Web服务访问点。

下面给出Hello CSAI的WSDL文档的完整内容供读者参考,自动生成该文档的方法将在8.8.1节中介绍。

```
<?xml version = "1.0" encoding = "utf-8"?>
<definitions xmlns:http = "http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap = http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:s = http://www.w3.org/2001/XMLSchema
    xmlns:s0 = "http://tempuri.org/"
    xmlns:soapenc = "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tm = "http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:mime = "http://schemas.xmlsoap.org/wsdl/mime/"
    targetNamespace = "http://tempuri.org/"
xmlns = "http://schemas.xmlsoap.org/wsdl/">
<types>
    <s:schemaelementFormDefault = "qualified"
        targetNamespace = "http://tempuri.org/">
    <s:element name = "SayHello">
    <s:complexType>
        <s:sequence>
        <s:element minOccurs = "0" maxOccurs = "1" name = "name" type = "s:string" />
        </s:sequence>
    </s:complexType>
    </s:element>
    <s:element name = "SayHelloResponse">
        <s:complexType>
            <s:sequence>
            <s:element minOccurs = "0" maxOccurs = "1" name = "SayHelloResult"
              type = "s:string" />
            </s:sequence>
        </s:complexType>
    </s:element>
    <s:element name = "string" nillable = "true" type = "s:string" />
    </s:schema>
```

```
</types>
<message name="SayHelloSoapIn">
    <part name="parameters" element="s0:SayHello" />
</message>
<message name="SayHelloSoapOut">
  <part name="parameters" element="s0:SayHelloResponse" />
</message>
<message name="SayHelloHttpGetIn">
    <part name="name" type="s:string" />
</message>
<message name="SayHelloHttpGetOut">
    <part name="Body" element="s0:string" />
</message>
<message name="SayHelloHttpPostIn">
  <part name="name" type="s:string" />
</message>
<message name="SayHelloHttpPostOut">
    <part name="Body" element="s0:string" />
</message>
<portType name="HelloMessageSoap">
    <operation name="SayHello">
        <input message="s0:SayHelloSoapIn" />
        <output message="s0:SayHelloSoapOut" />
  </operation>
</portType>
<portType name="HelloMessageHttpGet">
    <operation name="SayHello">
        <input message="s0:SayHelloHttpGetIn" />
        <output message="s0:SayHelloHttpGetOut" />
    </operation>
</portType>
<portType name="HelloMessageHttpPost">
    <operation name="SayHello">
        <input message="s0:SayHelloHttpPostIn" />
        <output message="s0:SayHelloHttpPostOut" />
   </operation>
</portType>
<binding name="HelloMessageSoap" type="s0:HelloMessageSoap">
    <soap:binding transport=
        "http://schemas.xmlsoap.org/soap/http" style="document" />
<operation name="SayHello">
    <soap:operation soapAction="http://tempuri.org/SayHello"
        style="document" />
    <input>
            <soap:body use="literal" />
    </input>
    <output>
            <soap:body use="literal" />
    </output>
</operation>
```

```
    </binding>
    <binding name = "HelloMessageHttpGet" type = "s0:HelloMessageHttpGet">
        <http:binding verb = "GET" />
        <operation name = "SayHello">
            <http:operation location = "/SayHello" />
            <input>
          <http:urlEncoded />
            </input>
            <output>
             <mime:mimeXml part = "Body" />
        </output>
    </operation>
    </binding>
    <binding name = "HelloMessageHttpPost" type = "s0:HelloMessageHttpPost">
      <http:binding verb = "POST" />
      <operation name = "SayHello">
        <http:operation location = "/SayHello" />
        <input>
          <mime:content type = "application/x-www-form-urlencoded" />
        </input>
        <output>
          <mime:mimeXml part = "Body" />
        </output>
      </operation>
    </binding>
    <service name = "HelloMessage">
      <port name = "HelloMessageSoap" binding = "s0:HelloMessageSoap">
        <soap:address location = "http://localhost/HelloService.asmx" />
      </port>
      <port name = "HelloMessageHttpGet" binding = "s0:HelloMessageHttpGet">
        <http:address location = "http://localhost/HelloService.asmx" />
      </port>
      <port name = "HelloMessageHttpPost" binding = "s0:HelloMessageHttpPost">
        <http:address location = "http://localhost/HelloService.asmx" />
      </port>
    </service>
</definitions>
```

8.6 统一描述、发现和集成协议

UDDI是一种用于描述、发现、集成Web服务的技术,它是Web服务协议栈的一个重要部分。通过UDDI,企业可以根据自己的需要动态查找并使用Web服务,也可以将自己的Web服务动态地发布到UDDI注册中心,供其他用户使用。

UDDI对于B2B模式的电子商务有着巨大的作用。例如,A公司是一个半导体配件的制造商,通过将自己的Web服务发布到UDDI注册中心,A公司的产品可以很容易地被分布于世界各地的采购方查找到,从而加入全球的供应链。同时,B公司是一个半导体配件的采购方,既从A公司采购部分半导体配件,也从其他公司采购原料。使用UDDI和Web

Service,B公司可以更容易地将自己和原材料供应厂商的系统集成在一起。那么B公司可以使用统一的平台以统一的方式管理自己的供应链,可以通过自己的信息系统和互联网,直接在其他公司的Web Service平台上下电子订单。当然,B公司也可以使用UDDI查找合适的供货商,并迅速地把这个新的合作伙伴加入自己的系统中。

在UDDI技术规范中,主要包含以下3个部分的内容。

(1) UDDI数据模型。UDDI数据模型是一个用于描述业务组织和Web服务的XML Schema。

(2) UDDI API。UDDI API是一组用于查找或发布UDDI数据的方法,UDDI API基于SOAP。

(3) UDDI注册服务。UDDI注册服务数据是Web服务中的一种基础设施,UDDI注册服务对应着服务注册中心的角色。

8.6.1 UDDI数据模型

在UDDI中定义了4种基本的数据结构,分别是businessEntity、businessService、bindingTemplate和tModel,这4种基本数据结构的关系如图8-9所示。

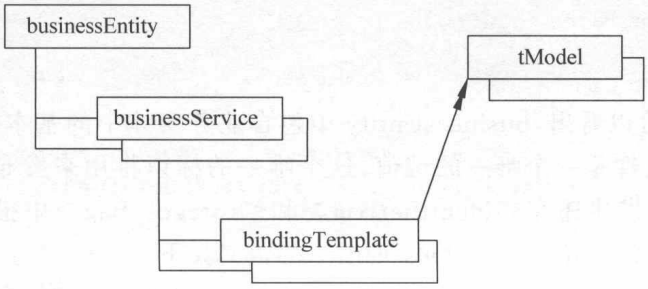


图8-9 基本数据结构关系

1. businessEntity

businessEntity元素对企业信息进行描述,包括企业的基本信息(企业名称、联系方式等)、分类信息(企业的类型等)以及标识信息。例如,下面是Microsoft的businessEntity中描述企业基本信息的一部分。

```
<businessEntity businessKey = "0076b468-eb27-42e5-ac09-9955cff462a3"
    operator = "Microsoft Corporation" authorizedName = "Martin Kohlleppel">
    <name>Microsoft Corporation</name>
    <description xml:lang = "en">Empowering people through great software any time, any place
and on any device is Microsoft's vision. As the worldwide leader in software for personal and
business computing, we strive to produce innovative products and services that meet
our customer's...
    </description>
    <contacts>
      <contact useType = "Corporate Addresses and telephone">
        <description xml:lang = "en">Corporate Mailing Addresses</description>
        <personName />
        <phone useType = "Corporate Headquarters">(425) 882-8080</phone>
```

```
        <address sortCode = "~" useType = "Corporate Headquarters">
          <addressLine > Microsoft Corporation </addressLine >
          <addressLine > One Microsoft Way </addressLine >
          <addressLine > Redmond, WA 98052-6399 </addressLine >
          <addressLine > USA </addressLine >
        </address >
      </contact >
      <contact useType = "Technical Contact - Corporate UD">
          <description xml:lang = "en"> World Wide Operations </description >
          <personName > Martin Kohlleppel </personName >
          <email > martink@microsoft.com </email >
      </contact >
    </contacts >
        <identifierBag >
          <keyedReference tModelKey = "uuid:8609c81e-ee1f-4d5a-b202-3eb13ad01823"
                keyName = "D-U-N-S" keyValue = "08-146-6849" />
        </identifierBag >
        <categoryBag >
          <keyedReference tModelKey = "uuid:c0b9fe13-179f-413d-8a5b-5004db8e5bb2"
                keyName = "NAICS: Software Publisher" keyValue = "51121" />
        </categoryBag >
</businessEntity >
```

从这个例子中可以看出,businessEntity 中包含业务实体中的基本信息。其中,第一行中的 businessKey 属性是一个唯一的键值,这个唯一的标识将用来发布服务。除了基本的联系信息外,这一段描述还在<identifierBag>和<categoryBag>中描述了业务标识和业务分类。使用 XML Schema 对 businessEntity 的定义如下。

```
<element name = "businessEntity">
  <complexType >
    <sequence >
      <element ref = "discoveryURLs" minOccurs = "0"/>
      <element ref = "name" maxOccurs = "unbounded"/>
      <element ref = "description" minOccurs = "0" maxOccurs = "unbounded"/>
      <element ref = "contacts" minOccurs = "0"/>
      <element ref = "bussinessService" minOccurs = "0"/>
      <element ref = "identifierBag" minOccurs = "0"/?
      <element ref = "categoryBag" minOccurs = "0"/>
    </sequence >
    <attribute ref = "businessKey" use = "required"/>
    <attribute ref = "operator"/>
    <attribute ref = "authorizedName">
  </complexType >
</element >
```

2. businessService

如上面的 businessEntity 定义所示,在 businessEntity 中可以包含一组 businessService 元素。businessService 描述了一个单一的或一组相关的 Web Service,其中包括名称、描述和一组 bindingTemplate。businessService 结构的定义如下。

```
<element name = "businessService">
  <complexType>
    <sequence>
      <element ref = "name" maxOccurs = "unbounded"/>
      <element ref = "description" minOccurs = "0" maxOccurs = "unbounded"/>
      <element ref = "bindingTemplates"/>
      <element ref = "categoryBag" minOccurs = "0"/>
    </sequence>
    <attribute ref = "serviceKey" use = "required"/>
    <attribute ref = "businessKey"/>
  </complexType>
</element>
```

3. bindingTemplate

一组 bindingTemplate 元素包含在 businessService 中。在 bindingTemplate 中描述了如何访问一个特定的 Web Service 并给出了访问点的地址(accessPoint)。bindingTemplate 的结构定义如下。

```
<element name = "bindingTemplate">
  <complexType>
    <sequence>
      <element ref = "description" minOccurs = "0" maxOccurs = "unbounded"/>
      <choice>
        <element ref = "accessPoint" minOccurs = "0"/>
        <element ref = "hostingRedirector" minOccurs = "0"/>
      </choice>
      <element ref = "tModelInstanceDetails"/>
    </sequence>
    <attribute ref = "bindingKey" use = "required"/>
    <attribute ref = "serviceKey"/>
  </complexType>
</element>
```

下面给出 XMethod. net 的 Web 服务中对于 businessService 和 bindingTemplate 描述的片断。

```
<businessService
  serviceKey = "d5921160 - 3e16 - 11d5 - 98bf - 002035229c64"
  businessKey = "ba744ed0 - 3aaf - 11d5 - 80dc - 002035229c64">
  <name> XMethods Delayed Stock Quotes </name>
  <description xml:lang = "en"> 20 - minute delayed stock quotes </description>
  <bindingTemplates>
    <bindingTemplate
      serviceKey = "d5921160 - 3e16 - 11d5 - 98bf - 002035229c64"
      bindingKey = "d594a970 - 3e16 - 11d5 - 98bf - 002035229c64">
      <description xml:lang = "en">
        SOAP binding for delayed stock quotes service
      </description>
      <accessPoint URLType = "http">
```

```
        http://services.xmethods.net:80/soap
      </accessPoint>
      <tModelInstanceDetails>
        <tModelInstanceInfo
          tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
      </tModelInstanceDetails>
    </bindingTemplate>
  </bindingTemplates>
</businessService>
```

4. tModel

在bindingTemplate元素中包含一个服务的URL和一个tModel。tModel结构是UDDI数据模型中最复杂也是最难理解的结构。tModel是描述技术规范的模型,它可以为外部的技术规范提供引用地址。例如,在上面XMethods.net的例子中,虽然bindingTemplate提供了访问Web服务的相关信息(如SOAP绑定的地址等),但是在bindingTemplate中并没有指定这个Web服务的接口,也就是说客户仍然无法直接访问该Web服务。而tModel通过对外部规范的引用,可以解决这个问题。例如:

```
<tModel tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64"
  operator="www.ibm.com/services/uddi" authorizedName="0100001QS1">
  <name>XMethods Simple Stock Quote</name>
  <description xml:lang="en">Simple stock quote interface</description>
  <overviewDoc>
    <description xml:lang="en">wsdl link</description>
    <overviewURL>
      http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference
      tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
        keyName="uddi-org:types" keyValue="wsdlSpec" />
  </categoryBag>
</tModel>
```

这个例子在<overviewDoc>中给出了这个Web服务接口的技术规范(一份WSDL文档)和获得该技术规范的地址(http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl)。tModel结构的定义如下。

```
<element name="tModel">
  <complexType>
    <sequence>
      <element ref="name"/>
      <element ref="description" minOccurs="0" maxOccurs="unbounded"/>
      <element ref="overviewDoc" minOccurs="0"/>
      <element ref="identifierBag" minOccurs="0"/>
      <element ref="categoryBag" minOccurs="0"/>
```

```
    </sequence>
    <attribute ref = "tModelKey" use = "required"/>
    <attribute ref = "operator"/>
    <attribute ref = "authorizedName"/>
  </complexType>
</element>
```

8.6.2 注册 Web 服务

使用 UDDI 注册 Web 服务可分为两个步骤。首先，为 UDDI 条目建立模型；然后，将 UDDI 条目注册到 UDDI 注册中心。建立 UDDI 条目时，需要确定以下事项。

(1) 确定 Web 服务的 tModel(WSDL 文档)。

(2) 确定组织(企业、事业单位、各类机构等，下同)名称、简介以及所提供的 Web 服务的主要联系方法。

(3) 确定组织正确的标识和分类。

(4) 确定组织通过 UDDI 提供的 Web 服务。

(5) 确定所提供的 Web 服务的正确分类。

事实上，这一系列的步骤也正是为了根据 UDDI 指定的数据结构建立正确的 UDDI 条目。在完成 UDDI 条目建模之后，就可以将其发布到 UDDI 注册中心。通过 UDDI 注册中心的 Web 页面和使用 UDDI API 编写程序都可以完成 Web 服务的发布。使用 UDDI API 可以直接编写程序，把 Web 服务发布到 UDDI 注册中心的服务器上。由于篇幅限制，本书不再介绍 UDDI API 的详细内容，感兴趣的读者可以查阅参考文献中的相关资料。

8.6.3 调用 Web 服务

使用 UDDI 调用 Web 服务的步骤如下。

(1) 编写调用 Web 服务的程序时，程序员使用 UDDI 注册中心来定位并获得 businessEntity。

(2) 程序员可以进一步获得更详细的 businessService 信息，或是得到一个完整的 businessEntity 结构。因为 businessEntity 结构中包含已发布的 Web 服务的全部信息，所以，程序员可以从中选择一个 bindingTemplate 待以后使用。

(3) Web 服务在 tModel 中提供了相关规范的引用地址，程序员可以根据引用地址获得规范并编写程序。

(4) 运行时，程序员可以按需要使用已经保存下来的 bindingTemplate 信息，调用 Web 服务。

8.7 消息封装协议

SOAP 以 XML 形式提供一个简单、轻量的用于在分散或分布环境中交换结构化和类型信息的机制。SOAP 本身并没有定义任何应用程序语义(如编程模型或特定语义的实现)，实际上，它通过提供一个有标准构件的包模型和在模块中编码数据的机制，定义了一个

简单的表示应用程序语义的机制。这使 SOAP 能够被用于从消息传递到 RPC 的各种系统。

SOAP 主要包括以下 4 个部分。

(1) SOAP 封装结构。定义一个整体框架用来表示消息中包含什么内容,谁来处理这些内容,以及这些内容是可选的或是必需的。

(2) SOAP 编码规则。用以交换应用程序定义的数据类型的实例的一系列机制。

(3) SOAP RPC 表示。定义一个用来表示远程过程调用和应答的协议。

(4) SOAP 绑定。定义一个使用底层传输协议来完成在结点间交换 SOAP 信封的约定。

SOAP 客户端是一种能够创建 XML 文档的本地应用程序,它不仅可以是普通的桌面应用程序,还可以是一种基于 Web 的应用程序,如 Web 表单等。它所创建的 XML 文档应该包含分布式 RPC 所需的信息,这些消息和请求一般都是通过 HTTP 进行传递的,而通常的安全机制都不会禁止 HTTP,因此,也就意味着 SOAP 可以顺利地通过防火墙,实现跨越不同平台的信息交换。

SOAP 服务器通常只是用来监听 SOAP 消息的特殊代码,它可以对 SOAP 文档进行分配和解释。它通常可以与基于 J2EE 技术或者是.NET 技术的应用程序服务器交互,这种应用程序服务器可以处理多种客户端的 SOAP 请求。SOAP 服务器也是通过 HTTP 连接接收文档,然后将其转换成为可以被另一端对象理解的语言。由于 SOAP 在所有的通信中都采用 XML 格式,因此,可以用来实现两种不同语言中的对象之间的通信。

从某种意义上,可以将 SOAP 简单地理解为:SOAP= HTTP+RPC+XML,也就是采用 HTTP 作为底层通信协议,RPC 作为通用的调用途径,XML 作为数据打包的格式,提供了一个能够穿越防火墙的通信交互能力。

8.7.1 消息封装和编码规则

SOAP 消息是一个 XML 文档,在实际应用中,提到 SOAP 消息时,往往是指这个 XML 文档。那么,SOAP 是如何进行封装,如何进行编码的呢? 这就是本小节要介绍的内容。

1. SOAP 的消息封装

SOAP 消息包括以下 3 个部分。

(1) 封装。封装的元素名是 Envelope,在表示消息的 XML 文档中,封装是顶层元素,在 SOAP 消息中必须出现。

(2) SOAP 头。SOAP 头的元素名是 Header,提供了向 SOAP 消息中添加关于这条 SOAP 消息的某些要素(feature)的机制。SOAP 定义了少量的属性用来表明这项要素是否可选以及由谁来处理。SOAP 头在 SOAP 消息中可能出现,也可能不出现。如果出现,必须是 SOAP 封装元素的第一个直接子元素。SOAP 头可以包含多个条目,每个条目都是 SOAP 头元素的直接子元素。

(3) SOAP 体。SOAP 体的元素名是 Body,是包含消息的最终接收者想要的信息的容器。SOAP 体在 SOAP 消息中必须出现且必须是 SOAP 封装元素的直接子元素。如果有头元素,则 SOAP 体必须直接跟在 SOAP 头元素之后;如果没有头元素,则 SOAP 体必须是 SOAP 封装元素的第一个直接子元素。SOAP 体可以包括多个条目,每个条目必须是

SOAP 体元素的直接子元素。

2. SOAP 的编码规则

XML 允许非常灵活的数据编码方式，SOAP 定义了一个较小的规则集合，下面的术语用来描述编码规则。

(1) 值(value)。值是一个字符串、类型(数字、日期、枚举等)的名或是几个简单值的组合。所有的值都有特定的类型。

(2) 简单值(simple value)。简单值没有名部分，如特定的字符串、整数、枚举值等。

(3) 复合值(compound value)。复合值是相关的值的组合，如订单、股票报表、街道地址等。在复合值中，每个相关的值都以名、序号或这两者来区分，这称为访问者(accessor)。在复合值中，多个访问者有相同的名是允许的。

(4) 数组(array)。数组是一个复合值，成员值按照在数组中的位置相互区分。

(5) 结构(struct)。结构也是一个复合值，成员值之间的唯一区别是访问者的名字，访问者名互不相同。

(6) 简单类型(simple type)。简单类型是简单值的类，如字符串类、整数类、枚举类等。

(7) 复合类型(compound type)。复合类型是复合值的类，它们有相同的访问者名，但可能会有不同的值。

在复合类型中，如果类型内的访问者名互不相同，但是可能与其他类型中的访问者名相同，即访问者名加上类型名形成一个唯一的标志符，这个名叫作“局部范围名”。如果名是直接或间接地基于 URI(Universal Resource Identifier，通用资源标志符)的一部分，那么不管它出现在什么类型中，这个名本身就可以唯一标志这个访问者，这样的名叫作“全局范围名”。

所有的值以元素内容的形式表示，对于每个具有值的元素，值的类型必须用下述 3 种方式之一描述。

(1) 所属元素实例有 xsi:type 属性。

(2) 所属元素是一个有 SOAP-ENC:arrayType 属性(该属性可能是默认的)的元素的子元素。

(3) 所属元素的名具有特定的类型，类型可以由 schema 确定。

8.7.2 SOAP 应用

SOAP 的应用很广泛，下面简单介绍 SOAP 在 HTTP 和 RPC 中的应用。

1. 在 HTTP 中使用 SOAP

把 SOAP 绑定到 HTTP，无论使用或不用 HTTP 扩展框架，都有很大的好处：在利用 SOAP 的形式化和灵活性的同时，使用 HTTP 种种丰富的特性。在 HTTP 中携带 SOAP 消息，并不意味着 SOAP 改写了 HTTP 已有的语义，而是将构建在 HTTP 之上的 SOAP 语义自然地对应到 HTTP 语义。SOAP 自然地遵循 HTTP 的请求/应答消息模型，使得 SOAP 的请求和应答参数可以包含在 HTTP 请求和应答中。

提示：SOAP 的中间结点与 HTTP 的中间结点并不等同，也就是说，不要期望一个根据 HTTP 连接头中的域寻址到的 HTTP 中间结点能够检查或处理 HTTP 请求中的 SOAP 消息。在 HTTP 消息中包含 SOAP 消息时，应用程序必须使用媒体类型 text/xml。

1) SOAP HTTP 请求

虽然 SOAP 可能与各种 HTTP 请求方式相结合,但是绑定仅定义了在 HTTP POST 请求中包含 SOAP 消息。

2) HTTP 头中 SOAPAction 域

一个 HTTP 请求头中的 SOAPAction 域用来指出这是一个 SOAP HTTP 请求,它的值是所要的 URI。在格式、URI 的特性和可解析性上没有任何限制。当 HTTP 客户发出 SOAP HTTP 请求时必须在 HTTP 头中使用这个域。

```
soapaction = "SOAPAction" ":" [ <"> URI - reference <"> ]
URI - reference = < as defined in RFC 2396 [4]>
```

HTTP 头中的 SOAPAction 域使服务器能正确地过滤 HTTP 中的 SOAP 请求消息。如果这个域的值是空字符串(""),表示 SOAP 消息的目标就是 HTTP 请求的 URI。这个域没有值,则表示没有 SOAP 消息的目标信息。例如:

```
SOAPAction: "http://electrocommerce.org/abc#MyMessage"
SOAPAction: "myapp.sdl"
SOAPAction: ""
SOAPAction:
```

3) SOAP HTTP 应答

SOAP HTTP 遵循 HTTP 中表示通信状态信息的 HTTP 状态码的语义。例如,2xx 状态码表示这个包含 SOAP 构件的客户请求已经被成功地收到、理解和接收。在处理请求时如果发生错误,SOAP HTTP 服务器必须发出应答 HTTP 500 "Internal Server Error",并在这个应答中包含一个 SOAP Fault 元素,表示这个 SOAP 处理错误。

4) HTTP 扩展框架

一个 SOAP 消息可以与 HTTP 扩展框架一起使用以区分是否有 SOAP HTTP 请求和它的目标。是使用扩展框架或是普通的 HTTP,关系到通信各方的策略和能力。通过使用一个必需的扩展声明和"M-"HTTP 方法名前缀,客户可以强制使用 HTTP 扩展框架。服务器可以使用 HTTP 状态码 510 "Not Extended"强制使用 HTTP 扩展框架。也就是说,使用一个额外的消息,任何一方都可以发现另一方的策略并依照执行。

5) SOAP HTTP 举例

使用 POST 的 SOAP HTTP:

```
POST /StockQuote HTTP/1.1
Content - Type: text/xml; charset = "utf - 8"
Content - Length: nnnn
SOAPAction: "http://electrocommerce.org/abc#MyMessage"
< SOAP - ENV:Envelope... HTTP/1.1 200 OK
Content - Type: text/xml; charset = "utf - 8"
Content - Length: nnnn
< SOAP - ENV:Envelope...
```

使用扩展框架的 SOAP HTTP:

```
M - POST /StockQuote HTTP/1.1 Man:
"http://schemas.xmlsoap.org/soap/envelope/"; ns = NNNN
Content - Type: text/xml;
charset = "utf - 8"
Content - Length: nnnn
NNNN - SOAPAction:
"http://electrocommerce.org/abc # MyMessage"
< SOAP - ENV:Envelope...
HTTP/1.1 200 OK
Ext:
Content - Type: text/xml; charset = "utf - 8"
Content - Length: nnnn
< SOAP - ENV:Envelope...
```

2. 在 RPC 中使用 SOAP

设计 SOAP 的目的之一就是利用 XML 的扩展性和灵活性来封装和交换 RPC 调用，在 RPC 中使用 SOAP 和 SOAP 绑定紧密相关。在使用 HTTP 作为绑定协议时，一个 RPC 调用自然地映射到一个 HTTP 请求，RPC 应答同样映射到 HTTP 应答。但是，在 RPC 中使用 SOAP 并不限于绑定 HTTP。

要进行方法调用，一般需要以下信息：目标对象的 URI、方法名、方法签名(signature)、方法的参数、头数据，其中方法签名和头数据是可选的。SOAP 依靠协议绑定提供传送 URI 的机制。例如，对 HTTP 来说，请求的 URI 指出了调用的来源。除了必须是一个合法的 URI 之外，SOAP 对一个地址的格式没有任何限制。

1) RPC 和 SOAP 体

RPC 方法调用和应答都包含在 SOAP Body 元素中，它们使用如下的表示形式。

(1) 方法调用结构表示。一个方法调用被看作单个的结构，每个[in]和[in/out]参数有一个访问者。结构的名和类型与方法相同，它们的出现顺序和方法中定义的参数顺序相同。

(2) 方法应答用结构表示。一个方法应答被看作单个的结构，返回值和每个[in]和[in/out]参数有一个访问者。第一个访问者是返回值，之后是参数访问者，参数访问者的出现顺序和方法中定义的参数顺序相同。每个参数访问者的名称和类型与参数的名称和类型相对应。

(3) 方法错误用 SOAP Fault 元素表示。如果绑定的协议有额外的规则表示错误，则这些规则也必须要遵从。因为返回结果表示调用成功，错误表示调用失败，所以，如果在方法应答中同时包含“返回结果”和“错误表示”，则是错误的。

2) RPC 和 SOAP 头

在 RPC 编码中，可能会有与方法请求有关但不是正规的方法签名的附加信息。如果这样，它必须作为 SOAP 头元素的子元素。使用这种头元素的一个例子是在消息中传递事务 ID。由于事务 ID 不是方法签名的一部分，通常由底层的构件而不是应用程序代码控制，所以没有一种直接的方法在调用中传递这个必要的信息。通过在 SOAP 头中添加一个给定名字的条目，接收方的事务管理器就可以析取这个事务 ID，而且不影响 RPC 的代码。

3) 示例

一个消息处理器从传入消息中检索并除去 SOAP 头的示例代码如下。

```
public class SOAPHeaderHandler extends javax.xml.rpc.handler.GenericHandler {
   ...
   public boolean handleRequest(MessageContext arg) {
    if (arg instanceof SOAPMessageContext) {
        SOAPMessageContext context = (SOAPMessageContext)arg;
        try {
            SOAPHeader header =
           context.getMessage().getSOAPPart().getEnvelope().getHeader();
            Iterator headers =
                    header.extractHeaderElements
                    ("http://schemas.xmlsoap.org/soap/actor/next");
            while (headers.hasNext()) {
                SOAPHeaderElement he =
                                    (SOAPHeaderElement)headers.next();
                // process the retrieved header element here
            }
        } catch (SOAPException x) {
            // insert error handling here
        }
    }
    return true;
   }
}
```

8.7.3 REST

REST(Representational State Transfer,表述性状态转移)是一种只使用 HTTP 和 XML 进行基于 Web 通信的技术,可以降低开发的复杂性,提高系统的可伸缩性。它的简单性和缺少严格配置文件的特性,使它与 SOAP 很好地隔离,REST 从根本上来说只支持几个操作(POST、GET、PUT 和 DELETE),这些操作适用于所有的消息。REST 提出了如下设计概念和准则。

(1) 网络上的所有事物都被抽象为资源。

(2) 每个资源对应一个唯一的资源标识符。

(3) 通过通用的连接件接口对资源进行操作。

(4) 对资源的各种操作不会改变资源标识。

(5) 所有的操作都是无状态的。

REST 中的资源所指的不是数据,而是数据和表现形式的组合。例如,“最新访问的 10 位会员”和“最活跃的 10 位会员”在数据上可能有重叠或者完全相同,而由于他们的表现形式不同,所以被归为不同的资源,这也是 REST 取名的原因。不管是图片、Word 文件还是视频文件,甚至只是一种虚拟的服务,也不管是 XML 格式、txt 文件格式还是其他文件格式,全部通过 URI 对资源进行唯一的标识。

对资源使用一致的命名规则(naming scheme)最主要的好处就是用户不需要提出自己

的规则，而是依靠某个已被定义、在全球范围中几乎完美运行，并且能被绝大多数人所理解的规则。例如，如果某个应用中包含一个对顾客的抽象，用户会希望将一个指向某个顾客的链接，能通过电子邮件发送到同事那里，或者加入到浏览器的书签中，甚至写到纸上。更透彻地讲，如果在一个类似于 Amazon 的在线商城中，没有用唯一的 ID(一个 URI)标识它的每一件商品，可想而知这将是多么可怕的业务决策。

8.8 构造一个简单的服务

作为一种采用松散耦合结构集成各式各样应用程序的标准，服务的技术特色在于动态发布、描述和调用机制。换句话说，服务的技术闪光点在于 SOAP、UDDI 和 WSDL 这 3 种协议巧妙的结合。相比之下，书写一段服务程序则相对轻松，并不比编写普通的应用程序更复杂。本节给出使用.NET 平台开发服务的例程供读者参考。

8.8.1 编写服务器端

使用.NET 平台开发服务的系统要求是，Windows 2000 Server 或以上版本，安装有.NET Frameworks SDK 和 IIS(Internet Information Server)5.0 以上版本。使用任何一个文本编辑器(如记事本等)编写如下代码。

```
<@ WebService Language = "C#" Class = "HelloMessage" %>
using System;
using System.Web.Services;
public class HelloMessage:WebService
{
    [WebMethod] public String SayHello(String name)
    {
        return "Hello, " + name;
    }
}
```

将这段代码保存为 HelloService.asmx，然后把这个文件复制到 IIS 服务的根目录下。打开浏览器，访问地址 http://localhost/HelloService.asmx 就可以看到该服务的测试页面。在这个测试页面中列出了服务提供的访问接口，对于编写的服务来说，将会列出 SayHello 方法。单击该方法的链接将会进入该方法的调试页面。在 SayHello 方法中包含一个参数 String name，在调试页面中有一个文本框用于输入该参数。在参数输入框中输入 CSAI，并单击“调用”按钮，IIS 将会返回一个 XML 文件，内容如下。

```
<?xml version = "1.0" encoding = "utf - 8" ?>
<String xmlns = "http://tempuri.org/">Hello, CSAI</string>
```

其中，包含该方法的调用结果和结果类型。下面对这段程序进行简单的说明。

```
<@ WebService Language = "C#" Class = "HelloMessage" %>
```

这一行代码声明了该程序是一个使用 C# 语言编写的 Web 服务程序。

```
using System;
using System.Web.Services;
```

这两行代码引用了命名空间,第一个是所有C#语言都会使用到的Sytem,另一个是Web服务程序中使用到的System.Web.Services。

完成声明之后,程序定义了类HelloMessage,类HelloMessage继承自Web.Service。类中定义了一个Web.Service方法:public String SayHello(String name),为了标识该方法可以通过Internet调用,在方法前使用了[WebMethod]作为修饰。所有的Web.Service方法都必须有这个修饰,否则不能被服务器处理。该方法只有一行代码,返回了字符串Hello和参数字符串的合成。

紧接着,可以生成该服务的WSDL文档,调用http://localhost/HelloService.asmx?WSDL,服务器将自动生成描述文档并返回。在8.5.3节的最后所给出的文档就是由服务器自动生成的WSDL文档。

8.8.2 编写客户端

前面提到过,调用服务可以通过静态绑定和动态绑定两种方式,在本节中,使用静态绑定的方式调用8.8.1节编写的服务。

首先,需要根据服务的描述文档产生相应的服务代理类,在执行过程中,客户使用代理类中的信息访问服务,并进行实际的方法调用。

假设.NET Framework SDK安装在C盘的默认路径下。首先,将C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin增加到系统的查找路径中(这里的路径仅是一个例子,因安装方式不同会有不同的路径)。在Framework SDK中提供了一个自动生成服务代理类的工具wsdl.exe,这个工具就位于刚才添加的路径中。进入Windows的命令行方式,输入命令:

```
wsdl.exe /language:c# http://localhost/HelloService.asmx?wsdl
```

稍微等待一小段时间就会看到命令完成的提示,同时生成客户代理类的代码文件HelloService.cs。然后打开文本编辑器,输入一段代码:

```
using System;
using System.IO;
public class HelloClient
{
    public static void Main()
    {
        HelloMessage sampClient = new HelloMessage();
        Console.WriteLine(sampClient.SayHello("CSAI"));
    }
}
```

将其保存为HelloClient.cs文件,并将这个文件放在和HelloService.cs相同的目录。

这时,需要使用到另外一个编译工具csc.exe。在默认情况下,csc.exe在C:\WINNT\

Microsoft .NET\Framework\V **** 目录下，其中 **** 代表了 Framework SDK 的实际版本号，依不同的安装情况而不同。同样，也可以将这个目录放到默认的查找路径中，或者直接输入“C:\WINNT\Microsoft .NET\Framework\V1. 0. 3705\csc. exe HelloClient. cs HelloMessage. cs”。这个命令将编译出一个可执行文件 HelloClient. exe，这就是调用 Web 服务的客户端。执行这个文件就可以看到，在系统控制台输出“Hello, CSAI”的结果。

8.9 Web 服务的应用实例

ERP(Enterprise Resource Plan，企业资源计划)是一个庞大、复杂的信息系统，传统的设计与开发模式已经不能满足其发展要求。因此，相关的开发人员提出了基于 Web 服务技术，按照面向服务的设计思想和开发模式，建立起包括系统入口、服务集成器、原子服务库和后台数据等 4 层体系结构(如图 8-10 所示)，并将 ERP 的业务逻辑划分为可复用、可扩充、面向整个 Internet 的 Web 服务单元，各个服务之间基于 SOAP 规范进行数据交换。

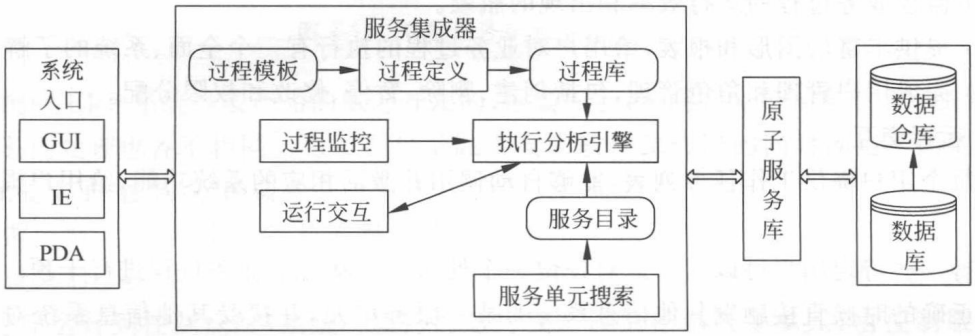


图 8-10 基于服务的 ERP 体系结构

1. 系统入口

ERP 系统作为企业信息化管理系统的集成平台，向用户提供单一的系统入口，该层可以是程序，也可以是网络浏览器，以及其他任何能够访问服务的程序单元。用户甚至也可以通过 PAD、掌上电脑、手机等通信工具访问系统。

2. 服务集成器

服务集成器是整个系统的核心部分，它支持企业内部以及跨越整个价值链的业务过程模型的建立、执行和监控，并能够实时地与其他相关信息系统进行集成。在实际运行过程中，由系统自动调用相关的服务单元，这些对用户都是透明的。它是用业务过程定义来驱动(或调用)功能单元的执行，而不是像传统的系统那样由程序代码决定业务过程。这种基于业务过程的系统集成方法，使得企业能够迅速完成对业务过程的建立、改变、分析和管理。

服务集成器主要包括如下 5 部分功能。

1) 过程定义

(1) 提供图形化界面，使用拖放式操作进行业务过程模型的实时建立，同时由系统自动在后台完成服务单元的“装配”，不需要再编写任何代码。

(2) 提供业务模板，方便地完成新业务过程的建立，达到重用的目的。

(3) 把业务逻辑与业务功能的代码实现相分离,从而在业务逻辑改变时不需要对信息系统进行重构。

(4) 能够对业务过程进行模拟运行,及早发现可能会出现的问题。

2) 执行分析引擎

(1) 负责业务过程执行的部分或全部运行控制环境,在业务的执行过程中对各种信息化系统、信息技术、信息平台和人员进行协调。

(2) 对过程定义进行解释,根据外部事件(例如,某一活动的完成等)和执行分析引擎的特定控制(例如,过程内部对下一步活动的引导)来自动改变过程状态。

(3) 控制业务过程句柄的创建、激活、暂停、终止等。

(4) 对过程活动进行引导,包括顺序或并行操作、限期安排、业务过程相关数据的安排等。

(5) 维护业务过程控制数据和相关数据。

3) 业务过程监控

(1) 实时对每一个业务过程的状态和发展全过程进行记录,控制过程定义版本。

(2) 监控业务过程的运行效率和出现的瓶颈。

(3) 提供丰富的图形和报表,给用户对业务过程的执行有一个全面、系统的了解。

(4) 完成用户管理和角色管理,包括创建、删除、暂停、修改和权限分配。

4) 运行交互

给每个用户提供工作任务列表,能够自动调用并激活相应的系统功能,给用户提供操作环境。

使得某些特定用户可以实时地对任何一个处于活动状态的业务过程进行干预。

在正确的时候直接触发其他信息系统的某一服务单元,并接受其他信息系统对系统的支配,实现不同信息系统的实时动态集成。

对延误的业务环节进行报警并提示管理人员。

5) 服务单元搜索

对企业中目前所使用的信息系统(可能是 COM 构件、CORBA 对象、SQL 语句、XML 样式表、其他形态的程序代码)自动进行搜索,然后将它们进行注册登记,存放到服务单元目录中,供业务过程执行时进行调用。

3. 原子库服务

在该体系结构中,所有的业务执行功能均以"网络服务单元"的形式出现。原子服务库是所有服务单元的集合,它不仅包含 ERP 系统自身的服务单元,而且可以将其他信息系统的功能单元进行 Web 包装后所得的服务单元进行整合,从而迅速完成系统集成。

原子服务库中的服务单元分为两种,分别为功能服务单元和控制服务单元。功能服务单元用来完成具体的业务操作和数据处理,包括 ERP 系统的所有主要功能实现,而且服务之间可以进行继承。功能服务单元的顶层分类如图 8-11 所示。

控制服务单元用来决定业务流程的流向并实现权限管理,它本身并不是一个独立可执行的部件,依附于功能服务单元,可为功能服务单元提供约束。控制服务单元的调用触发机制也适用于对功能服务单元操作,不同的是对控制服务单元的调用是获取某种服务最终仍回至当前功能服务单元,而对功能服务单元的调用则可能启动该功能即进入当前功能之外的另一个功能,即功能与功能之间的快速连接。

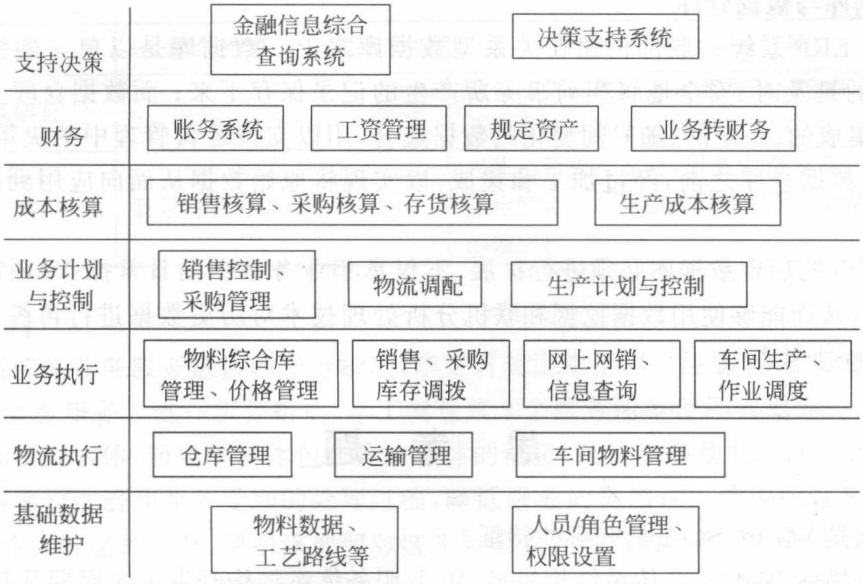


图 8-11　功能服务单元顶层分类图

传统的 ERP 系统是按产品的概念开发的，但实际上，不同的客户需求千差万别，即使对同一事务的处理也各不相同。为了在尽量减少改动软件源代码的同时满足不同的客户需求，因此提出了“控制点”的概念。

所谓控制点，是指企业业务流程中的某些相邻基本处理过程之间的转折点，业务流程在该点之后会出现多条分支，选择不同的分支就会产生不同的处理模式。在 ERP 系统中，当程序运行到控制点时，系统将软件的运行权开放给控制服务单元，由控制服务单元对功能服务单元进行特异化的处理，从而将程序的特异性要求与共同的部分隔离。当程序运行到控制点时，主程序出让所有的控制权，由控制部件负责对业务的处理，处理完之后再将控制权交回主程序。这样只要为不同的客户开发不同的控制服务单元，就能够使系统具备良好的适应性。例如，在采购流程中，在“签订采购合同”之后设定一个控制点，用户可以通过该控制点的控制服务单元进行后续业务的设定，如“先收货后结算”或者“先付款后结算”，选择了一种方式之后，系统就沿着所设定的处理模式进行后续业务的执行。

传统方式与控制点方式的比较如图 8-12 所示。

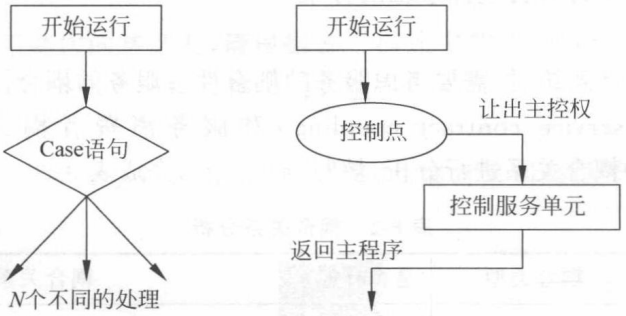


图 8-12　传统方式与控制点方式的比较

4. 数据库与数据仓库

传统的ERP系统一般都构建在关系型数据库之上。数据库是以单一的数据资源为中心,其目的是及时、安全地将当前事务所产生的记录保存下来;而数据仓库是指一个面向主题的、集成的、稳定的、随时间变化的数据集合,用以支持经营管理中的决策制定过程,数据在进入数据仓库之前,经过加工和集成,以实现将原始数据从面向应用到面向主题的转变。

ERP系统的后台数据库必须进行扩展,不仅承担业务数据的日常操作,还需要建立起分析型环境,从而能够使用数据挖掘和联机分析处理技术对历史数据进行再综合、再处理,更好地支持企业决策。

思 考 题

1. 什么是SOA? SOA具有哪些特征?

2. 简单描述Web服务体系结构模型,Web服务体系结构开发生命周期及其优势。

3. Web服务作为Web服务体系结构的核心,简要说明Web服务的核心技术及其作用。

4. 请用WSDL描述一个简单的Web服务体系结构系统。

5. 简述面向服务的体系结构的概念及其设计原则。

6. 结合你实际开发的一个项目,谈谈该如何进行基于Web服务体系结构的系统设计、开发和部署?

7. 希赛公司欲对现有的核心咨询业务系统进行升级改造,以适应大量用户的个性化咨询服务要求,提高系统的灵活性。公司主管将核心业务系统的升级改造工作交给了公司的系统分析师王工和李工。一个月后,王工和李工分别向公司提交了自己的方案。王工主张以公司现有的、采用面向对象技术和Java语言实现的业务系统为基础,针对新的业务需求对系统进行重构、改造与升级;李工则认为现有系统的业务逻辑过于复杂,对系统进行重构的成本太高,可以采用面向服务的思想,提炼可复用的业务功能形成服务,实现系统的灵活性。经过公司相关人员共同开会讨论,最终采用了李工的改造方案。

【问题1】

请从系统业务功能实现和功能集成两个方面,对王工和李工的方案进行分析和对比,并结合项目需求说明公司为何会选择李工的方案。

【问题2】

采用服务思想设计系统时,需要考虑服务的耦合性。服务的耦合性可以分为两类,分别是服务契约耦合(service contract coupling)和服务消费者耦合(service consumer coupling)。请对各种耦合关系进行分析,填写(a)~(g),完成表8-2。

表8-2 耦合关系分析

耦合关系	耦合类型	是否有害	耦合关系解释
业务逻辑-契约耦合	(a)	(d)	(f)
契约-技术耦合	(b)	(e)	(g)

续表

耦合关系	耦合类型	是否有害	耦合关系解释
消费者-实现耦合	(c)	是	服务消费者需要通过服务契约使用服务功能，而不能直接与服务实现相关
消费者-契约耦合	服务消费者耦合	否	在使用服务功能时，服务消费者需要与服务契约进行绑定。这样能够提高服务消费者和服务之间的独立性

【问题 3】

在对系统的业务服务进行初步分析后，李工首先提取了客户注册、业务受理和发票开具3个典型的业务服务。进一步分析后，李工发现这3个服务需要使用数据库中的客户实体、业务实体和发票实体，而发票实体包含客户实体的信息，这样会导致发票开具服务内部包含并重复实现客户注册和业务受理的处理过程，降低服务的复用性。请说明这种情况产生的主要原因，并针对这种情况，说明该如何对这3个服务进行重构，使它们具有更好的重用性。

8. 希赛是一个大型的电信软件开发公司，公司内部采用多种商业/开源的工具进行软件系统设计与开发工作。为了提高系统开发效率，公司管理层决定开发一个分布式的系统设计与开发工具集成框架，将现有的系统设计与开发工具有效集成在一起。集成框架开发小组经过广泛调研，得到了如下核心需求。

(1) 目前使用的系统设计与开发工具的运行平台和开发语言差异较大，集成框架应无缝集成各个工具的功能。

(2) 目前使用的系统设计与开发工具所支持的通信协议和数据格式各不相同，集成框架应实现工具之间的灵活通信和数据格式转换。

(3) 集成框架需要根据实际的开发流程灵活、动态地定义系统工具之间的协作关系。

(4) 集成框架应能集成一些常用的第三方实用工具，如即时通信，邮件系统等。

集成框架开发小组经过分析与讨论，最终决定采用企业服务总线(ESB)作为集成框架的基础体系结构。

【问题 1】

ESB是目前企业级应用集成常用的基础体系结构。请列举出ESB的4个主要功能，并从集成系统的部署方式、待集成系统之间的耦合程度、集成系统的可扩展性3个方面说明采用ESB作为集成框架的基础体系结构的原因。

【问题 2】

在ESB基础体系结构的基础上，请根据题干描述中的4个需求，说明每个需求应该采用哪种具体的集成方式或体系结构风格最为合适。

【问题 3】

请指出在实现工具之间数据格式的灵活转换时，通常采用的设计模式是什么，并对实现过程进行简要描述。

参考文献

第9章 富互联网应用体系结构

互联网日益成为应用程序开发的默认平台，传统的Web应用程序是基于HTML页面、服务器端数据传递的模式，而HTML是适合于文本的。随着Web应用程序复杂性越来越高，传统的Web应用程序已经不能满足Web浏览者更高的、全方位的体验要求了。为此，富互联网应用(Rich Internet Application，RIA)应运而生。

9.1 RIA的概念

基于HTML的应用程序之所以变得流行是由于应用系统的部署成本低、结构简单，且HTML易于学习和使用。很多用户和开发人员都乐于放弃由桌面计算机带来的用户界面改进，来实现对新数据和应用系统的快速访问。与丧失一些重要的用户界面功能相比，基于Web的方式所带来的好处要大得多。

然而，某些应用系统并不完全适合采用HTML技术。复杂的应用系统可能要求多次提取网页来完成一项事务处理，在某些领域中，如医药和财务领域，这往往导致交互速度低得无法接受。此外，虽然HTML开始走向简单，但是即使简单的交互活动也仍然需要用很多的脚本来完成。即使一个输入窗体经过仔细的布置和全面的脚本设计，它从浏览器所能发送的也仅仅是简单的“名字/值”对。如果一个HTML窗体能够以XML文档形式发送和接收更复杂的数据结构，那就好多了。

9.1.1 RIA的提出

总的来说，传统的Web应用程序存在以下几个缺点。

(1) 操作复杂性。由于传统Web应用程序的局限性，当进行一个多步骤或多选项的事务时，用户要么会看到一份很长的、笨拙的页面，要么就得通过反复翻转若干网页、令人沮丧地执行操作步骤。

(2) 数据复杂性。高效率地表达复杂的数据，是现有Web应用程序所面临的巨大挑战。理想的图形工具应该能够既操作简便，又能生动明了地展示各种错综复杂的数据信息。

(3) 交互复杂性。互动性需求的应用程序使得交互的问题日益突出，用户的耐心越来越少，他们的要求是向桌面应用程序的速度靠齐。

传统Web应用程序之所以不能表达高度的复杂性，其主要原因是HTML网页技术的先天不足。当应用进行到一定深度时，这种缺陷便逐渐显露出来。一直以来，非智能的客户端提交请求并得到服务器的响应，这种以网页为载体的网络逐步形成了如今的互联网。在这种模式下，作为默认用户界面的网页，它的上下文自然地同时又是人为地进行流程分割，

以便映射将业务处理分解为步骤的机制。尽管网页逐渐地加入越来越多的动态特点进行改良，然而，在展示能力及与用户互动方面仍然后劲乏力。

一味地提升服务器和网络的速度既不现实又不经济，一种可行的技术方案就是采用高度互动性和局部智能型的客户端应用程序。这样，就可以在无须刷新全页或增加带宽需求的情况下，迅速响应用户的输入并做出相应的处理。

企业级应用程序经历了几次系统结构方面的重要转变，在此过程中，客户端的表现能力有起有落。图 9-1 显示了 RIA 的发展过程。

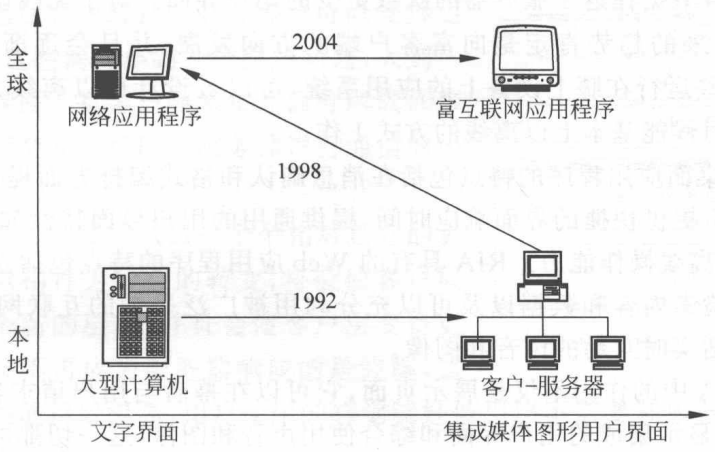


图 9-1　RIA 的发展过程

9.1.2　丰富的含义

RIA 是 Web 开发和部署模式的一种演变。“富”的含义有两种，分别是丰富的数据模型和丰富的用户界面。

丰富的数据意味着客户端的用户界面能表现和应对更多更复杂的数据模式，这样才能处理客户端的运算以及异步发送、接收数据。当页面在服务器上创建完成并交付给 HTML 后，客户端的程序为用户提供比与服务器交互更良好的感受。为了达到高度复杂的数据模式，客户端允许用户构建一个高响应、交互式的应用程序。

丰富也指 RIA 能提供更多的改良界面。HTML 只能为用户的界面控制提供有限的功能；反之，RIA 允许一些富有创造性的界面控制，巧妙地与数据模式相合。传统的互联网模式是线性设计方式，用户唯一的选择就是用批处理方式提交页面到服务器。在这种技术限制下用户体验很糟糕，这种程序不是用户所需要的。连续处理服务器请求和页面更新存在许多障碍，包括页面响应时间、不良的网络带宽，以及满足 session 或 state 交叉连接而不断增长的日常开销。伴随着丰富的用户界面，用户可以从早期的服务器响应影响整个界面的运作模式，迁移到只对发出请求的特定区域进行改变的模式上来。本质上，意味着界面将会被分解为由单独个体组成，来适应局部改变、服务器交互，以及客户端内部构件的通信。

丰富的结果是用户可以创建一个客户端界面，这样更容易反映丰富性与复杂性共存的数据和逻辑。

9.1.3 RIA的优点

RIA利用相对健壮的客户端描述引擎,这个引擎能够提供内容密集、响应速度快和图形丰富的用户界面。除了提供一个具有各种控件(滑标、日期选择器、窗口、选项卡、微调控制器、标尺等)的界面之外,RIA还允许使用SVG(Scalable Vector Graphics,可伸缩向量图)或其他技术随时构建图形。一些RIA技术甚至能够提供全活动的动画对数据变化做出响应。

RIA的另一个好处在于,数据能够被缓存在客户端,从而可以实现一个比基于HTML的响应速度更快且数据往返于服务器的次数更少的用户界面。对于无线设备和需要偶尔连接的设备来说,将来的趋势肯定是向富客户端的方向发展,并且会逐渐远离基于文本的Web客户端。那些运行在膝上设备上的应用系统,可以被设计成以离线方式工作,或者至少当连接丢失的时候能基本上以离线的方式工作。

RIA具有的桌面应用程序的特点包括在消息确认和格式编排方面提供互动用户界面,在无刷新页面之下提供快捷的界面响应时间,提供通用的用户界面特性如拖放式(drag and drop)以及在线和离线操作能力。RIA具有的Web应用程序的特点包括立即部署、跨平台、采用逐步下载来检索内容和数据以及可以充分利用被广泛采纳的互联网标准。RIA具有通信的特点则包括实时互动的声音和图像。

客户机在RIA中的作用不仅是展示页面,它可以在幕后与用户请求异步地进行计算、传递和检索数据、显示集成的用户界面和综合使用声音和图像,这一切都可以在不依靠客户机连接的服务器或后端的情况下进行。

对于企业来说,部署RIA的好处如下。

(1) RIA可以继续使用现有的应用程序模型(包括J2EE和.NET),因而无须大规模替换现有的Web应用程序。通过RIA技术,可以轻松构建更为直观、易于使用、反应更迅速并且可以脱机使用的应用程序。

(2) RIA可以帮助企业提供多元化的重要业务效益,包括提高产品销量、提高品牌忠诚度、延长网站逗留时间、较频繁地重复访问、减少带宽成本、减少支持求助、增强客户关系等。

9.2 RIA模型

在J2EE中,典型的网络部署系统的结构如图9-2所示。

其中,客户层包含用户界面和程序接口的容器;表示层包含表述内容的逻辑、处理用户会话、状态管理;业务层包含系统的业务逻辑;集成层包含访问远端程序和数据源的连接器和适配器;资源层包含数据库资料以及像ERP这样的企业信息资源和XML文件。

图9-2 典型的Web部署系统

在网络部署系统中,决定程序流程、内容创建和内容传递的逻辑是存在于中间层的,因为传统的Web应用程序的主要特点是一个基于客户端浏览器的请求/响应模型。如果要传递界面,系统需要在它通过中间层向客户层传递之前对内容进行以HTML

的形式的格式化和编译。对任何表示层的修改都需要向服务器发出请求，做出响应是整个页面的而不是仅做出改动的部分。而在传统的网络模型中采用 Applet，JavaScript/DHTML(Dynamic HTML，动态 HTML)等方法来代替客户层的解决方案，始终没有成为主流，这主要是因为各种各样的技术问题，特别是不同浏览器的兼容问题。

RIA 模型可以更好地反映应用的结构，如图 9-3 所示。

图 9-3　典型的富客户端模型

不像 J2EE 应用那样，客户端的请求会导致系统生成一个页面再返回客户端；一个富客户可以支持更小的单元或构件，这些构件小到一个投票问题，大到一个完整的视图或界面。富客户模型将界面分解成许多既可以和用户直接交互又可以和服务器进行通信的小单元模块。

这种将应用系统的设计从以一个个相对独立的页面为中心转移到以构件为中心的转变，将会使客户层的设计提升到一个新的层次，并且会使客户层变得更加灵活。富客户层不再成为服务器响应的最终端，这同时也使程序的性能得以提高，用户使用的感觉就好像程序不需要和服务器进行通信或者只是偶尔才需要进行通信。

RIA 模型是一种事件模型，不像传统的模型那样，服务器收到请求后由上至下地创建客户端界面，用户不用预测事件的顺序。既然每个构件都是独立的，就没有必要因为一个请求而做出影响整个视图的反应。要使每个构件都具有向服务器传送信息的能力需要每个构件知道如何处理服务器传递回来的信息。在 RIA 中，客户端和服务器端交互数据是不同步的，这样用户就可以控制构件创建信息发送给服务器和处理服务器的响应，可以为更零散地控制去耦合和分离程序功能并且建立面向服务的程序结构。

9.3　RIA 客户端开发技术

本节简单介绍几种常用的 RIA 客户端开发技术。

1. Macromedia Flash/Flex

Flash 是一个成熟的商业产品，它可以在 Web 网页中引入交互式的图形界面。最新版本包含建立窗体风格的应用程序的功能。尽管 Flash 作为一个在 Web 上最广泛部署的前端技术还有争议，但据称已经有 98%以上的桌面系统都支持 Flash，这使得以 Macromedia Flash Player 为客户端的 RIA 可以支持种类广泛的平台和设备。用来创建动画式图形的 Flash 工具功能十分强大，而且是可视化的(与之相反，其他技术要求进行低级的图形编码)，所以图形设计人员使用起来十分得心应手。

Flex 是为满足希望开发 RIA 的企业级程序员的需求而推出的表示服务器和应用程序的框架，它可以运行于 J2EE 和.NET 平台。Flex 表示服务器提供基于标准的、声明性的编程方法和流程，并提供运行时服务，用于开发和部署 RIA 的表示层。Flex 开发者使用直观的基于 XML 的 MXML 来定义丰富的用户界面。该语言由 Flex 服务器翻译成 SWF 格式

的客户端应用程序,在 Flash Player 中运行。

Flex 和 Flash 的最大缺点在于对 XML 和 Web 服务等标准的支持很有限,而且作为应用开发工具的环境还不成熟。Flex 和 Flash 的优点在于可以很容易地用来创建复杂的动画式显示,以及可以使用第三方插件。

2. AJAX

AJAX(Asynchronous JavaScript And XML,异步 JavaScript 和 XML)用来描述一组技术,它使浏览器可以为用户提供更为自然的浏览体验。借助于 AJAX,可以在用户单击按钮时,使用 JavaScript 和 DHTML 立即更新用户界面,并向服务器发出异步请求,以执行更新或查询数据库;当请求返回时,就可以使用 JavaScript 和 CSS 相应地更新用户界面,而不是刷新整个页面。最重要的是,用户甚至不知道浏览器正在与服务器通信,Web 站点看起来是即时响应的。

AJAX 是当前实现 RIA 的主流技术,将在 9.4 节进行详细介绍。

3. Laszlo

Laszlo 是一个开源的富客户端开发环境。使用 Laszlo 平台时,开发者只需编写名为 LZX 的描述语言(其中整合了 XML 和 JavaScript),运行在 J2EE 应用服务器上的 Laszlo 平台会将其编译成 SWF 格式的文件并传输给客户端展示。从这点上来说,Laszlo 的本质和 Flex 一样。Flash 是任何浏览器都支持的展示形式,从而一举解决了浏览器之间的移植问题。而且,Laszlo 还可以将 LZX 编译成 Java 或.NET 本地代码,从而大大提高运行效率。

4. Avalon

Microsoft 的 Avalon 是 Windows 的一部分,是一个图形和展示引擎,主要由新加到.NET 框架中的一组类集合而成。Avalon 定义了一个新标记语言,其代号为 XAML(可扩展应用程序标记语言)。可以使用 XAML 定义文本、图像和控件的布局,程序代码可以直接嵌入 XAML 中,也可以将它保留在一个单独的文件内。这与 Flex 中的 MXML 或者 Laszlo 中的 LZX 非常相似。不同的是,基于 Avalon 的应用程序必须运行在 Windows 环境中,而 Flex 和 Laszlo 是不依赖于平台的,仅需要装有 Flash 播放器的浏览器即可。

5. Java SWT

Java 完全支持创建基于窗体的用户界面。除了 Java 基础类(JFC/Swing)中的用户界面构件之外,开发人员还可以使用来自 Eclipse Project 的 SWT 工具箱和许多第三方工具箱进行开发。对于图形来说,可以采用 Java 2D API,这是一个非常完整且非常复杂的图形 API。用户可以通过一个 Web 浏览器使用 Java 插件软件,或使用 Java 运行时环境中较新的 Java Web Start 技术部署应用程序。使用 Java 建立富客户端的主要缺陷是它的复杂性(即使是简单的窗体和图形,也要求编写非常烦琐的代码)和 Java 浏览器插件的低市场占有率。

6. XUL

XUL(XML User Interface Language,XML 用户界面语言)来自 Mozilla 的开放源码项目。XUL 可用于建立窗体应用程序,这些应用程序不但可以在 Mozilla 浏览器上运行,而且也可以运行在其他描述引擎上,如 Zulu(一个 Flash MX 构件)和 Thinleys(一个 Java 实现)。XUL 描述引擎都非常小(通常都在 100KB 以下),它既可以使用 XML 数据,也可以生成 XML 数据。

XUL 的一个主要缺点在于它目前还没有获得一个主要商业实体的支持。XUL 最大的优点在于它与 Gecko 引擎的集成(打开了通向大量 Web 标准的大门),以及与大多数其他 XML 用户界面描述语言相比,它是一种非常具有表达力的、简洁的语言。

7. Bindows

Bindows 是用 JavaScript 和 DHTML 开发的 Web 窗体框架。JavaScript 用于客户端界面的显示和处理,XML HTTP 用于客户端与服务器的信息传输。JavaScript 在客户端的表现力不容置疑,利用 JavaScript 几乎可以实现 Windows 应用程序所能干的大部分事情,XML HTTP 一直以来常被用于实现无刷新的 Web 页面,它和 JavaScript 配合,可以完成数据从服务器和客户端的传输。

Bindows 的一个主要缺点是它采用一次全部载入的方式实现脚本库,在窗口的加载期,需要一个漫长的等待过程,甚至浏览器的进程会产生无响应的情况。在这一点上,Bindows 根本没有遵循"用多少取多少"的准则。另外,Bindows 内部大量利用了 IE6 的技术,没有考虑到非 IE 的浏览器,这限制了 Bindows 的流行。

8. Oracle Forms

Oracle Forms 是用来构建以数据库为中心的互联网应用系统的一个成熟的商品化产品。通过 Oracle Forms,用户可以使用一个输出窗体模块文件的可视化设计器创建窗体。为了便于在该设计工具外部进一步进行处理,模块文件要么采用私有的 FMT 格式,要么采用 XML 格式。这些模块文件驱动一个描述窗体的 Java 运行时环境。除了所有窗体的标准窗口小部件之外,还可以通过集成附加的可插入的 Java 构件和一些定制的 JavaBean 来实现更多的功能。

Oracle Forms 采用的脚本语言为 PL/SQL,Oracle 数据库也采用同样的脚本语言。Oracle Forms 的一个非常有趣的特点是,用来建立、编辑和编译窗体模块文件的 Java API——开发人员可以通过创建脚本生成众多的窗体应用程序,或者进行全局性的改动。Oracle Forms 的主要缺点是,进行 Web 部署需要获得 Oracle 应用服务器的使用许可。Oracle Forms 的优点是,它可以与 Oracle 数据库和 Oracle 平台的其他部分紧密集成,对国际化的广泛支持,以及高效率地创建以数据为中心的应用程序。

9.4 AJAX 技术

AJAX 是由几种蓬勃发展的技术以新的方式组合而成的,包含基于 XHTML (eXtensible HyperText Markup Language,可扩展超文本标识语言)和 CSS 标准的表示;使用 DOM 进行动态显示和交互;使用 XMLHttpRequest 与服务器进行异步通信;使用 JavaScript 绑定一切。

(1) XML。XML 是一套从 SGML 中派生的定义语义标记的规则,这些标记将文档分成许多部件并对这些部件加以标识。它也是元标记语言,用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。XML 的高扩展性、高灵活性特性,使得其可以描述各种不同种类的应用软件中的各种不同类型的数据,可以实现不同数据的集成。由于 XML 格式的标准化,许多浏览器软件都能够提供很好的支持,因此,只需简单地将 XML 格式的数据发送给客户端,客户端就可以自行对其进行编辑和处理,而不仅是显示。

(2) XHTML。XHTML是一个基于XML的标记语言,是一个扮演着类似HTML角色的XML,结合了部分XML的强大功能和大多数HTML的简单特性,建立XHTML的目的是实现HTML向XML的过渡。

(3) JavaScript。JavaScript是一种黏合剂,使AJAX应用的各部分集成在一起。在AJAX中,JavaScript主要用来传递用户界面上的数据到服务器端并返回结果。

(4) XMLHttpRequest。XMLHttpRequest对象用来响应通过HTTP传递的数据,一旦数据返回客户端,就可以立刻使用DOM将数据显示在网页上。XMLHttpRequest对象在大部分浏览器中已经实现,而且拥有一个简单的接口,允许数据从客户端传递到服务器端,但并不会打断用户当前的操作。使用XMLHttpRequest传递的数据可以是任何格式的。

(5) DOM。DOM为XML文档的已解析版本定义了一组接口。解析器读入整个文档,构建一个驻留内存的树结构,然后代码就可以使用DOM接口来操作这个树结构。

(6) XSLT。XSLT是一种将XML文档转换为XHTML文档或其他XML文档的语言,可以用在客户端和服务器端,它能够减少大量的用JavaScript编写的应用逻辑。

(7) CSS。一个CSS样式单就是一组规则,样式再根据特定的一套规则级联起来。每个规则给出其所适用的元素名称,以及要应用于哪些元素的样式。CSS提供了从内容中分离应用样式和设计的机制。虽然CSS在AJAX应用中扮演至关重要的角色,但它也是构建跨浏览器应用的一大阻碍,因为不同的浏览器厂商支持各种不同的CSS级别。

借助于AJAX,可以在用户单击按钮时,使用JavaScript和XHTML立即更新用户界面,并向服务器发出异步请求,以执行更新或查询数据库。当请求返回时,就可以使用JavaScript和CSS来相应地更新用户界面,而不是刷新整个页面。最重要的是,用户甚至不知道浏览器正在与服务器通信,Web站点看起来是即时响应的。使用AJAX的最大优点,是能在不更新整个页面的前提下维护数据,这使得Web系统更为迅捷地回应用户动作,并避免了在网络上发送那些没有改变过的信息。

使用AJAX的主要缺点是,它可能破坏浏览器"后退"按钮的正常行为。在动态更新页面的情况下,用户无法回到前一个页面状态,这是因为浏览器只能记下历史记录中的静态页面。用户通常都希望单击"后退"按钮,就能够取消他们的前一次操作,但在AJAX系统中,却无法做到这一点。解决这个问题的主要方法是,在用户单击"后退"按钮访问历史记录时,通过建立或使用一个隐藏的IFRAME重现页面上的变更。例如,当用户在Google Maps中单击"后退"按钮时,它在一个隐藏的IFRAME中进行搜索,然后将搜索结果反映到AJAX元素上,以便将系统恢复到当时的状态。另外,使用动态页面更新时,用户难以将某个特定的状态保存到收藏夹中。解决这个问题的主要方法是,使用URL(Uniform Resource Locator,统一资源定位符)片断标识符(通常称为锚点,即URL中#后面的部分)保持跟踪,允许用户回到指定的某个系统状态。

进行AJAX开发时,需要慎重考虑网络延迟。不给予用户明确的回应,没有恰当的预读数据,或者对XMLHttpRequest的不恰当处理,都会使用户感到延迟,这是用户不希望看到的,也是他们无法理解的。通常的解决方案是,使用一个可视化的组件告知用户,系统正在进行后台操作并且正在读取数据和内容。

9.5 Mashup 技术

Mashup 就是糅合的意思，是当今网络上的一种技术，该技术将两种以上使用公共或者私有数据库的 Web 应用加在一起，形成一个整合应用。Mashup 源于语义 Web 领域的数据建模技术和松耦合、面向服务、与平台无关的通信协议相结合，最终将提供一种开发可充分利用并整合大量 Web 信息的应用程序所必需的基础设施。

Mashup 通过开放 API、RSS(Really Simple Syndication，内容聚合)等方式把不同内容聚合起来，它和 AJAX 的结合增加了与用户的交互。简单的聚合原理和图形化界面的开发工具，产生一种新的应用软件开发模式，这种模式简化了开发的难度，吸引不具备精深专业知识的人员参与到他们感兴趣的服务创建中，普通用户可以轻松地按照自己的需求完成一个应用程序。

9.5.1 Mashup 的体系结构

Mashup 从体系结构上由 3 个不同的部分组成：API/内容提供者、Mashup 站点和客户机的 Web 浏览器，如图 9-4 所示。这 3 个部分在逻辑上和物理上都是相互脱离的(可能由网络和组织边界分隔)。

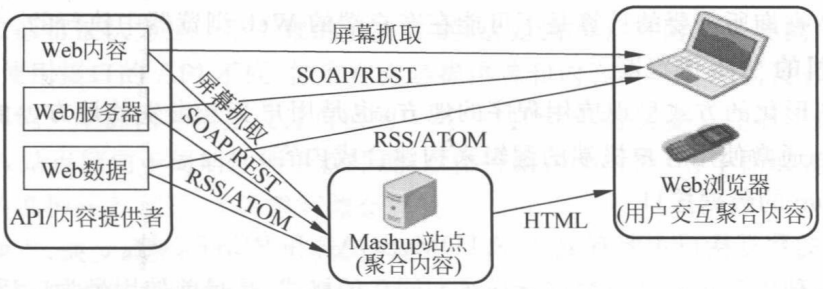


图 9-4 Mashup 的体系结构

1. API/内容提供者

内容提供者是正在进行融合的内容的提供者。内容提供者提供的内容通常有 3 种类型，分别是数据型、应用逻辑型和用户界面型，分别对应提供数据、应用逻辑功能和 GUI；内容提供者和 Mashup 应用程序的接口又可分为 CRUD(Creater-Read-Update-Delete，创建-读取-更新-删除)接口、具体程序语言接口、XML/HTML 等标记语言、GUI 元素等；一个内容提供者可以同时提供几种接口和不同类型的内容。

为了方便数据的检索，内容提供者通常会将自己的内容通过 Web 协议对外提供(如 REST、Web 服务、RSS/ATOM 等)。然而，很多有趣的潜在数据源可能并没有方便地对外提供 API。从诸如维基百科，以及所有政府和公共领域的 Web 站点上提取内容的 Mashup 都是通过一种称为屏幕抓取(screen scraping)的技术实现的。在这种情况中，屏幕抓取意味着使用一种工具从内容提供者那里提取信息的过程，这个工具会尝试对提供者的专为阅读而设计的页面进行分析。

2. Mashup 站点

Mashup 站点就是 Mashup 所在的地方,但是,这里指的是 Mashup 逻辑所在的地方,而不是执行这些逻辑的地方。Mashup 站点聚合内容的方式有两种,分别是服务器端的聚合和客户端的聚合。

一方面,Mashup 可以直接使用服务器端的动态内容生成技术(如 JSP、ASP、PHP 或 Ruby)实现类似传统 Web 的应用程序。

另一方面,合并内容可以直接在客户机的浏览器中通过客户端脚本(如 JavaScript 等)生成。这种客户端的逻辑通常是直接在 Mashup 的 Web 页面嵌入代码与这些 Web 页面引用的脚本 API 库或 Applet(由内容提供者提供)的组合。Mashup 使用的这种方法就是 RIA,它们是以交互式用户体验为导向的。Google Maps API 的设计就是为了通过浏览器端的 JavaScript 进行访问,这是客户机端技术的一个例子。客户端进行数据聚合的优点如下。

(1) 对 Mashup 服务器所产生的负载较轻,因为数据可以直接从内容提供者那里传送过来。

(2) 具有更好的无缝用户体验,页面可以请求对内容的一部分进行更新,而不用刷新整个页面。

通常,Mashup 使用服务器和客户端逻辑的组合实现自己的数据集成。很多 Mashup 应用程序使用了直接由用户提供的数据,至少使一个数据集是本地的。另外,对多数据源的数据执行复杂查询所需要的计算是不可能在客户端的 Web 浏览器中执行的。

3. 客户机的 Web 浏览器

这是以图形化的方式呈现应用程序的地方,也是用户交互发生的地方。如上面介绍的一样,Mashup 通常使用客户机端的逻辑来构建合成内容。

4. Mashup 的数据接口

Mashup 的数据接口主要有 RSS、ATOM、SOAP 和 REST 4 种。

RSS 是一种用于对网站内容进行描述和同步的格式,是目前使用最为广泛的 Web 资源发布方式。RSS 是一种基于 XML 标准,在互联网上被广泛采用的内容包装和投递协议,任何内容源都可以采用这种方式来发布,包括专业新闻、网络营销、企业甚至个人等站点。若在用户端安装了 RSS 阅读器软件,用户就可以按照喜好、有选择性地将感兴趣的内容来源聚合到该软件的界面中,为用户提供多来源信息的"一站式"服务。

ATOM 与 RSS 相类似,但比 RSS 有更大的弹性。ATOM 是一种基于 XML 的文档格式以及基于 HTTP 的协议,它被站点、客户工具等用来聚合网络内容。ATOM 借鉴了各种版本 RSS 的使用经验,开发一个新的网站内容摘要格式以解决 RSS 存在的问题,如混乱的版本号、不是一个真正的开放标准、表示方法的不一致等。ATOM 希望提供一个清晰的版本以解决每个人的需要,其设计完全不依赖于内容提供者,任何人都可以对之进行自由扩展。

9.5.2 Mashup 实现技术

Mashup 不是一项单独的技术,与 AJAX 一样,它是多种技术的综合使用。Mashup 聚合的内容可概括为服务和数据。如果聚合的是服务,则通过调用 API 获取各个源的功能。

Mashup 最常用的 API 类型一般有两种，分别是 REST 和 SOAP。如果聚合的是数据，则使用 RSS 或 Atom 获取数据。

在 Mashup 聚合时，语义 Web 和 RDF(Resource Description Framework，资源描述框架)可以帮助实现高质量的 XML 数据聚合和 RSS 摘要内容聚合。使用语义技术和 RDF 可以让 Mashup 用户更好地控制服务、信息和表示，高效高质地创建 Mashup 应用程序。

Mashup 常以 Widget 形式封装功能和数据源，这种可视化的小部件可供用户使用拖动来实现功能和数据的聚合，使得用户可以真正参与到终端编程中。

总之，Mashup 和 AJAX 结合起来创建 RIA，可丰富用户界面，加速用户与网页交互响应和改善用户体验。

1. 屏幕抓取

正如前面介绍的一样，如果内容提供者没有提供 API(此时，其实不是真正意义上的“内容提供者”)，通常会强制要求 Mashup 开发人员采取屏幕抓取的方式来提取自己希望集成的信息。抓取是使用软件工具处理并分析最初为人们阅读而编写的内容，从中提取可以通过编程进行使用和操作的信息的语义数据结构表示。

有些 Mashup 使用屏幕抓取技术获取数据，特别是从公用领域提取数据。例如，房地产地图 Mashup 就可以在制图供应商提供的地图上显示售价和租价，这些数据可能是从当地的记录办公室抓取的数据。

屏幕抓取通常被认为是一个不雅的解决方案，这是因为它有两个主要的固有缺点。第一个缺点，与使用接口的 API 不同，抓取在内容提供者和内容消费者之间没有明确的联系。抓取者必须围绕一个源内容模型设计自己的工具，并且希望提供者一直采用这种模型来呈现内容。Web 站点倾向于周期性地更新外观，以保持新颖和时尚，对于抓取者来说，这是一项非常头痛的维护任务，因为工具很可能会失效。

第二个缺点，缺少成熟的可重用屏幕抓取工具包软件，此类 API 和工具包的消亡很大程度上是由于每种抓取工具都有极为特定于应用程序的需求。这为开发人员带来了过多的开发工作，他们必须对内容进行反向工程处理、开发数据模型、分析并从提供者站点上汇集原始数据。

2. 语义 Web 和 RDF

屏幕抓取不好的一面直接源自一个事实：为阅读而创建的内容并不太适合机器自动处理。这促进了语义 Web 的诞生，它是传统 Web 的增强版本，在为人们设计的内容中增加了足够多的可供机器阅读的信息。在语义 Web 环境中，信息这个术语与数据有所差异。数据只有在传达了自己的含义(即数据可被理解)之后才会变成信息。语义 Web 的目标是创建 Web 基础设施，使用元数据对数据进行增强，从而使数据变得有意义，最终使数据变得适合进行自动化、集成、推理和重用。

RDF 就是服务于这个目的的技术，它用来建立描述数据的语义结构。XML 本身并不足以实现这种功能。RDF-Schema 补充了 RDF 的能力，提供了以机器可读的方式编码概念的功能。一旦可通过一种数据模型描述数据对象，RDF 就提供了通过“主语-谓语-对象”三元组(主语 S 与对象 O 具有关系 R)在数据对象之间构建关系的能力。

思 考 题

1. 请从体系结构发展的角度回答,为什么要使用 RIA 技术?

2. 根据你的技术背景,以及本书前 9 章的学习,请思考一个问题:技术发展的驱动力是什么?

3. AJAX 技术的核心是什么? AJAX 是如何将多种已有的技术绑定在一起的? 这些技术各自起到什么作用?

4. traveler.com 是一家在线旅游信息服务公司,其主要业务是为自助旅游者提供关于旅游线路及周边信息的服务。随着公司业务的不断发展,公司用户要求提供基于位置的增值旅游信息服务,即希望能够在给定位置(利用 GPS 全球定位系统获取)的情况下得到周边的地理位置、住宿、餐饮和交通等旅游相关信息。针对该需求,公司技术人员对现有系统的体系结构和运行模式进行了认真分析,决定采用 Mashup 技术集成来自其合作网站(设为 A、B、C、D)的信息,满足用户的需求。具体实现方式如下。

(1) 利用 A 网站提供的地图信息,得到用户位置相关的周边地理信息。

(2) B 网站根据用户的位置信息向其提供周边的住宿信息。

(3) C 网站根据用户的位置信息向其提供周边的餐饮信息。

(4) D 网站根据用户的位置信息向其提供周边的公交线路等信息。

【问题 1】

图 9-5 是公司进行 Mashup 的流程示意图,请阅读并补充图中数字标出部分的内容。

(1) 用户向 Traveler 网站请求服务,请求页面提供用户的位置信息。

(2) _____。

(3) _____。

(4) A 网站向 Traveler 网站返回用户所处位置周边的地图信息。

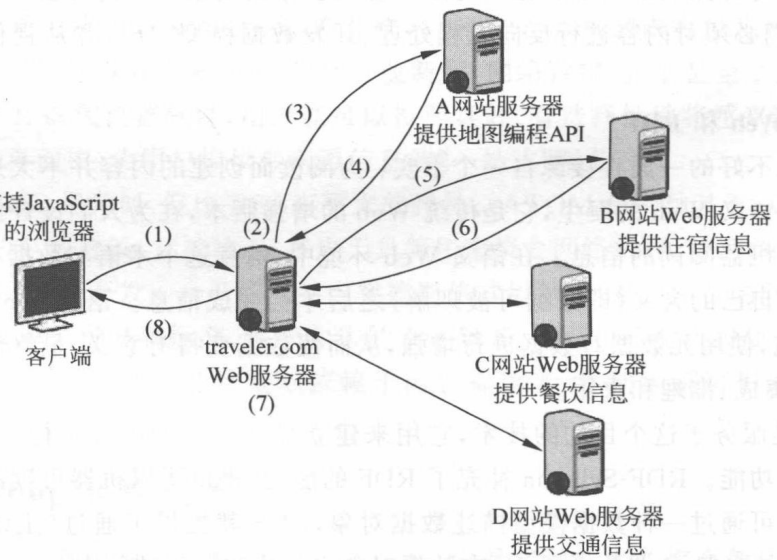


图 9-5 Mashup 流程示意图

(5) _____。

(6) B 网站向 Traveler 网站返回用户所处位置周边的住宿信息。

(7) _____。

(8) Traveler 网站向用户返回用户所处位置周边整合的旅游信息。

【问题 2】

目前，互联网上已经存在很多 Mashup 应用，各大网站也纷纷提供了各种格式的 Mashup 数据接口。请给出 3 种目前经常使用的 Mashup 数据接口，并加以简单说明。

【问题 3】

在实现 Mashup 应用时，进行内容聚合的物理位置是一个十分重要的因素。目前很多 Mashup 站点都选择在客户端机器上进行内容聚合，构成所谓的富互联网应用程序。请说明在客户端进行内容聚合的优点。

参考文献

第 10 章 软件体系结构的分析与测试

软件体系结构作为一个高层次的抽象，它描述了构件及其之间的关系，但是当前的课程体系中对软件体系结构的介绍是从高层抽象结构进行说明的。这让很多读者感觉体系结构的内容太抽象，导致其不能够被及时地理解。因此，本章介绍对软件体系结构的定量化描述，有利于读者对软件体系结构有一个量化的认识。

10.1 体系结构的可靠性建模

在基于构件的可靠性模型中，通过状态图来描述系统的行为。一个状态表示一个构件的执行，从一个状态到另一个状态的迁移概率通过系统的操作剖面获得。软件系统的可靠性依赖于状态的执行顺序和每一个状态的可靠性。在此模型中，每一个迁移可以看作一个马尔可夫过程，即构件执行的下一个状态只依赖于当前构件的状态，与下一个状态和前一个状态是无关的。

状态图是一个有向图，在状态图中每一个结点 S_i 表示一个状态，从状态 S_i 到 S_j 的迁移通过连接边(S_i,S_j)表示。假设 R_i 表示状态 S_i 的可靠性，P_{ij} 表示从状态 S_i 到 S_j 成功迁移的概率，基于状态图可以定义一个迁移矩阵 $\boldsymbol{M}$ 和可达值 $\boldsymbol{M}(i,j)=R_i\times P_{ij}$，$\boldsymbol{M}(i,j)$表示能够成功地从状态 S_i 到状态 S_j 的概率。

$$
\boldsymbol{M}=\begin{array}{c}\\ S_1\\ S_2\\ \vdots\\ S_i\\ \vdots\\ S_{n-1}\\ S_n\end{array}\begin{array}{c}\begin{array}{ccccccc} S_1 & S_2 & \cdots & S_i & \cdots & S_{n-1} & S_n \end{array}\\ \begin{bmatrix} 0 & R_1P_{12} & \cdots & R_1P_{1i} & \cdots & R_1P_{1n-1} & R_1P_{1n}\\ R_2P_{21} & 0 & \cdots & R_2P_{2i} & \cdots & R_2P_{2n-1} & R_2P_{2n}\\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots\\ R_iP_{i1} & R_iP_{i2} & \cdots & 0 & \cdots & R_iP_{in-1} & R_iP_{in}\\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots\\ R_iP_{(n-1)1} & R_iP_{(n-1)2} & \cdots & & P_iP_{(n-1)i} & 0 & R_iP_{(n-1)n}\\ R_nP_{n1} & R_nP_{n2} & \cdots & R_nP_{ni} & \cdots & R_nP_{n(n-1)} & 0 \end{bmatrix}\end{array}
$$

下面介绍如何通过迁移矩阵计算系统的可靠性。假设 $S=\{S_1,S_2,\cdots,S_n\}$是一个状态的集合，S_1 是开始状态，S_n 是最终状态。$\boldsymbol{M}^k(i,j)$表示通过 k 个迁移从状态 S_i 到状态 S_j 的成功概率，此时该系统通过 k 个迁移，由 S_i 到状态 S_j 的可靠性为 $R=\boldsymbol{M}^k(i,j)\times R_j$，$k$ 的取值范围是 0 到无穷大，取 0 时表示最初状态也是最终状态，取无穷大时表示此时发生一个循环。因此，考虑每一个可能的迁移结果是非常必要的。

假设矩阵 $\boldsymbol{T}=\boldsymbol{I}+\boldsymbol{M}+\boldsymbol{M}^2+\boldsymbol{M}^3+\cdots=\sum_{k=0}^{\infty}\boldsymbol{M}^k=\frac{\boldsymbol{I}}{\boldsymbol{I}-\boldsymbol{M}}=(\boldsymbol{I}-\boldsymbol{M})^{-1}$，其中 $\boldsymbol{I}$ 是 $n\times n$ 阶单位矩阵，此时，整个系统的可靠性可以通过公式 $R=T(1,n)\times R_n$ 计算，其中 $T(1,n)=(-1)\frac{|\boldsymbol{E}|}{|\boldsymbol{I}-\boldsymbol{M}|}$，$\boldsymbol{E}$ 是矩阵$(\boldsymbol{I}-\boldsymbol{M})$除去第 n 行和第一列的矩阵。

基于构件模型的可靠性分析可以通过上述的方法计算，然而一个复杂的系统通常包括顺序、并行计算、容错和客户-服务器 4 种常用的结构风格模型。容错体系结构风格的目的是通过一系列的容错构件修正初始系统的错误来改善系统的可用性。下面分别对这 4 种结构进行讨论。

1. 顺序结构风格

在顺序结构风格中，系统的运行按构件的顺序依次执行，顺序结构风格广泛应用于银行系统的日常数据更新。该风格的模型图如图 10-1 所示。

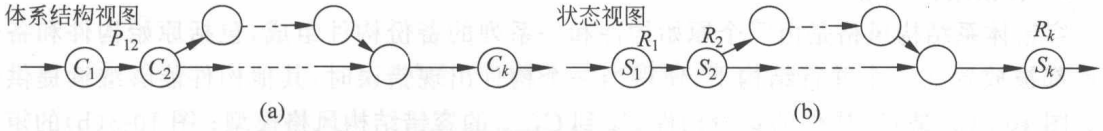


图 10-1　顺序风格模型

假设体系结构由 k 个构件顺序组成，此风格可以认为在马尔可夫链中有 k 个状态，其迁移矩阵通过下式构造：

$$\begin{cases}\boldsymbol{M}(i,j)=0, & S_i \text{ 不能够到达 } S_j\\ \boldsymbol{M}(i,j)=R_iP_{ij}, & S_i \text{ 能够到达 } S_j\end{cases}, \quad 1\leqslant i,\ j\leqslant k$$

其中，$\boldsymbol{M}(i,j)$是状态 S_i 能够成功到达状态 S_j 的概率。

2. 并行/管道-过滤器结构风格

在顺序执行环境中，某一时刻只有一个构件执行，然而在当前执行环境中构件通常是同时运行的，这种行为称为并行/管道-过滤器结构风格。它通常可以通过用一个马尔可夫链来模型化，应用于当前环境中有多个构件同时运行的情况，主要用来提高系统性能。

在并行/管道-过滤器结构风格中，多个构件可以同时执行，其结构如图 10-2 所示。并行结构风格和管道-过滤器结构风格的不同之处在于，并行结构通常用于多处理器环境中，如图 10-2(a) 所示；而管道-过滤器结构风格通常发生在单处理器多进程环境中，如图 10-2(b)所示。执行构件 C_2 到 C_{k-1} 对应并行结构的状态集 S_p 的状态 S_{p1}。

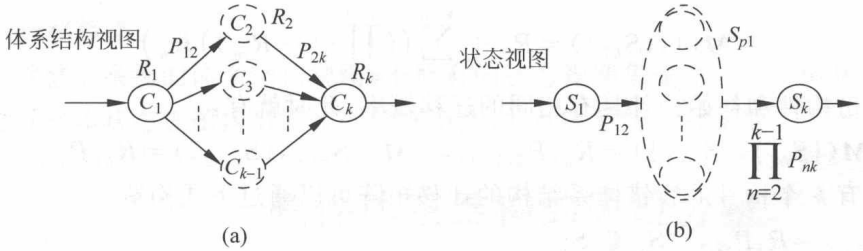


图 10-2　并行/管道-过滤器结构风格模型

在图 10-2(a)中，是一个具有 k 个构件的体系结构，其中有 $l=k-2$ 个构件是在同一状态下并行运行的，因此该系统的总状态的数量是 $k-l+1$。由于并行风格的特点，从构件

C_1 到构件 $C_2, C_3, \cdots, C_{k-1}$ 的迁移概率都为 P_{12}，P_{12} 也是从状态 S_1 到 S_{p1} 的迁移概率。为了方便计算，再引入 $\{S_i\}$，在矩阵中它返回状态变量 S_i 的行数或列数。

在 $\boldsymbol{M}(\{S_{p1}\},\{S_k\})$ 中，状态 S_{p1} 从 C_2 到 C_{k-1} 所有的构件成功执行并到达最终状态 S_k，由于构件的可靠性和迁移概率是相互独立的，因此 $\boldsymbol{M}(\{S_{p1}\},\{S_k\})$ 为：

$$\boldsymbol{M}(\{S_{p1}\},\{S_k\}) = \prod_{n=2}^{k-1} R_n P_{nk}$$

因此，具有 k 个构件的并行/管道-过滤器体系结构风格的迁移矩阵可以通过下式构造。

$$\begin{cases} \boldsymbol{M}(i,j) = R_i P_{ij}, & S_i \notin S_p \\ \boldsymbol{M}(i,j) = \prod_{CninSi} R_n P_{nj}, & S_i \in S_p, \quad 1 \leqslant i, \ j \leqslant |S| \text{ 且 } 1 \leqslant n \leqslant k \\ \boldsymbol{M}(i,j) = 0, & S_i \text{ 不能到达 } S_j \end{cases}$$

3. 容错结构风格

容错体系结构风格是由一个原始构件和一系列的备份构件组成，包括原始构件和备份构件都被放置在一个并行结构下，使得当一个构件出现错误时，其他构件能够继续提供服务。图 10-3(a)是 C_2 构件的备份构件 C_3 到 C_{k-3} 的容错结构风格模型；图 10-3(b)的矩形框内的容错构件聚集在状态 S_{b1} 内。

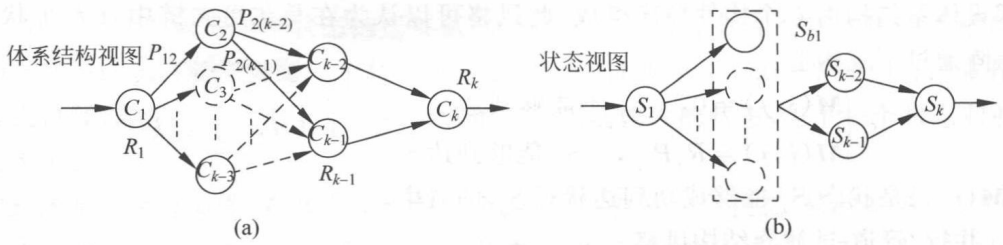


图 10-3　容错结构风格

假设所有的备份构件的迁移概率与原始构件的迁移概率是相同的。在图 10-3(a)中，假设一个有 k 个构件的容错结构风格，其中有 $l=k-4$ 个构件是在同一状态，因此状态的总数为 $k-l+1$。当前的容错结构风格类似于并行结构风格，假设从构件 C_1 到 $C_2, C_3, \cdots, C_{k-3}$ 的迁移概率为 P_{12}，P_{12} 也是从状态 S_1 到状态 S_{b1} 的迁移概率。

在容错结构风格中，从状态 S_1 到状态 S_{k-2} 和状态 S_{k-1} 时，如果状态 S_2 出现错误，则使用 S_3 的状态，如果 S_2 和 S_3 都出现错误，则使用 S_4 的状态，以此类推。则此时有：

$$\boldsymbol{M}(1,\{S_{b1}\}) = R_2 + \sum_{n=3}^{k-3}\left(\left(\prod_{m=2}^{n-1}(1-R_m)\right)R_n\right)$$

假设备份构件和初始构件具有相同的迁移概率，此时就有：

$$\boldsymbol{M}(\{S_{b1}\},\{S_{k-1}\}) = R_{b1}P_{2(k-1)}, \quad \boldsymbol{M}(\{S_{b1}\},\{S_{k-2}\}) = R_{b1}P_{2(k-2)}$$

因此具有 k 个构件的容错体系结构的迁移矩阵可以通过下式构造。

$$\begin{cases} \boldsymbol{M}(i,j) = R_i P_{ij}, & S_i \notin S_b \\ \boldsymbol{M}(i,j) = R_{a1} + \sum_{q=a2}^{ar}\left(\left(\prod_{m=a1}^{q-1}(1_R_m)\right)R_n\right), & S_i \in S_b \text{ 并且 } S_i \text{ 包括 } C_{a1}\cdots C_{ar} \\ \boldsymbol{M}(i,j) = 0, & S_i \text{ 不能到达 } S_j, \quad 1 \leqslant i, \ j \leqslant |S|, \quad 1 \leqslant a_r \leqslant k \end{cases}$$

4. 调用-返回结构风格

在调用-返回结构风格中，一个构件在完成一次迁移之前，在执行过程中可能需要调用其他构件提供的服务，因此必须在该调用完成之后并返回调用构件后，再执行当前构件剩下的下一个状态。

在调用-返回结构风格中，被调用构件可能被多次调用，而调用构件只执行一次。如图 10-4 所示，图 10-4(b)的状态图可以通过图 10-4(a)的体系结构视图的一一对应映射获得，S_1 是调用状态，S_2 是给 S_1 提供服务的被调用状态。在执行过程中，状态 S_2 在状态 S_1 迁移到状态 S_3 之前被多次调用。

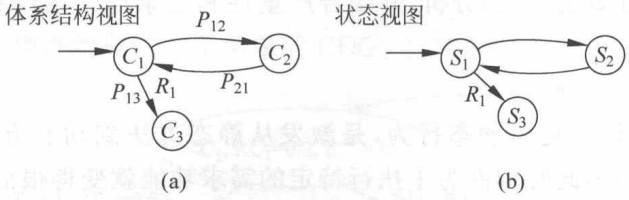


图 10-4　调用-返回结构风格

此时，$M(1,3)$等于 R_1P_{13}，它是状态 S_1 的可靠性乘以从状态 S_1 到 S_3 的迁移概率。同理，$M(2,1)$也可以通过状态 S_2 的可靠性和从状态 S_2 到状态 S_1 的迁移概率相乘获得。值得注意的是：由于状态 S_1 只有在迁移到状态 S_3 时才被访问，不管状态 S_1 访问状态 S_2 多少次，因此在 $M(1,2)$中仅考虑从状态 S_1 到状态 S_2 的迁移概率，并不考虑状态 S_1 的可靠性。

假设存在由 k 个构件组成的调用-返回结构风格，其状态数量为 k，其迁移矩阵可以通过下式构造。

$$\begin{cases} \boldsymbol{M}(i,j)=R_iP_{ij}, & S_i \text{ 可以到达 } S_j \\ \boldsymbol{M}(i,j)=P_{ij}, & S_i \text{ 可以到达 } S_j, S_j \text{ 是被调用构件}, \quad 1\leqslant i, \quad j\leqslant k \\ \boldsymbol{M}(i,j)=0, & S_i \text{ 不能到达 } S_j \end{cases}$$

通过上述方法，能够计算出单一结构风格的体系结构的可靠性，但是，实际上一个软件系统可能有多种类型的结构风格，那么前面的方法就无法计算出其体系结构的可靠性。此时，系统的可靠性可以通过以下步骤来模型化。

(1) 通过系统的详细说明书，确定系统所采用的体系结构风格。

(2) 把每一种体系结构风格转换成状态视图，并计算状态视图中每一个状态的可靠性及其相应的迁移概率。

(3) 通过整个系统的体系结构视图，把所有的状态视图集成为一个整体状态视图。

(4) 通过整体状态视图构造系统的迁移矩阵，并计算系统的可靠性。

10.2　软件体系结构的风险分析

风险评估过程通常用于验证需要详细检测的复杂模型，估计潜在的模型问题和测试效果，在不同的开发阶段都可以执行分析评估。在体系结构级进行风险评估更有利于开发阶段的后期评估。

风险评估对于任何软件风险管理计划都是一个重要的过程。有些风险评估技术是基于个人的主观的领域经验判断,主观的风险评估技术是个人行为,带有一定的错误。风险评估应该是基于能够通过定量的方法对软件产品属性进行的度量。

10.2.1 风险分析的方法

本节介绍一个体系结构级的风险评估方法,它是基于动态方法的。该方法用动态复杂性和动态耦合性来定义用于描述体系结构元素(构件、连接件)的复杂性因子。通过采用FMEA(Failure Mode and Effect Analysis,失效模式和有效性分析)对软件体系结构模型进行严重性(失效后所导致的影响)分析,并结合严重性和复杂性为构件和连接件设计出启发式风险因子。

1. 动态方法

许多实时应用软件的复杂动态行为,是激发从静态方法到动态方法的一个重要手段。当构件被调用或执行时,此时构件为了执行特定的需求功能就变得很活跃,然而由于活跃构件的频繁执行和状态的频繁改变,这就使得越是活跃的构件它们就越有可能出现错误。因此,如果在一个活跃的构件中存在一个错误,那么它可能发生错误的概率就很高。为了在体系结构级进行风险分析,只对错误风险感兴趣,因此有目的把在系统运行时用动态方法评估构件和连接件的复杂度作为目标。

动态方法用来评估执行中的软件体系结构的动态耦合度和动态复杂度。动态耦合度用来度量在特定的执行场景中,两个相互连接的构件或连接件之间的活跃程度。该度量作为连接件的复杂度,可以通过场景侧面的比例获取。动态复杂度方法用来测量特定构件在给定场景下的动态行为,也可以通过场景侧面的比例获得构件复杂度。

在此,用动态方法为每一个体系结构元素定义了复杂性因子,构件复杂性因子是用于构件的行为描述,可以通过动态复杂度方法获取;连接件复杂性因子是用于连接件的消息传输协议,可以通过动态耦合方法获取。

2. 构件依赖图

构件依赖图(Component Dependency Graph,CDG)是用于在体系结构级进行可靠性分析的概率模型。一个构件依赖图是一个对基于构件的软件系统的可靠性分析模型,它是控制流图的一个扩展。它把系统的构件、连接件及其之间的关系模型转化为一个CDG。CDG是一个有向图,它描述了构件、构件的可靠性,连接件、连接件的可靠性和迁移概率。CDG是从一个场景中开发出来的,一个场景就是一系列的通过特定输入激发的构件交互,通常用UML的顺序图对场景进行建模。通过顺序图,能够收集统计所需要的用于建立CDG相关信息,如某个构件在该场景中的评价执行时间、场景的评价执行时间、构件之间的可能交互等。

定义 CDG=$<N,E,s,t>$,N是结点集,E是边集,s和t分别是初始点和终结点。

例如,$N=\{n\}$,$E=\{e\}$,$n=<C_i,\mathrm{RC}_i,\mathrm{EC}_i>$,其中,$C_i$表示$i$个构件,$\mathrm{RC}_i$表示第$i$个构件的可靠性,$\mathrm{EC}_i$是第$i$个构件的平均执行时间,$\mathrm{EC}_i$可以通过下式获得。

$$\mathrm{EC}_i=\sum_{k=1}^{|S|}\mathrm{PS}_k\times\mathrm{Time}(C_i)_{C\,i\,in\,S_k}$$

其中,PS_k是场景S_k的执行概率,$|S|$是场景的总数,$\mathrm{Time}(C_i)$是构件C_i的执行时间,

Time(C_i)可以通过在序列图中构件 C_i 在其生命线上的活跃时间总数进行估计。C_i 是场景 S_k 中的构件。

$e=<T_{ij},\mathrm{RT}_{ij},\mathrm{PT}_{ij}>$，其中，$T_{ij}$ 是从结点 n_i 到 n_j 的迁移，RT_{ij} 是该迁移的可靠性，PT_{ij} 是迁移概率，PT_{ij} 可以通过下式获得。

$$\mathrm{PT}_{ij}=\sum_{k=1}^{|S|}\mathrm{PS}_k\times\left(\frac{|\ \mathrm{Interact}(C_i,C_j)\ |}{|\ \mathrm{Interact}(C_i,C_l)\ |_{l=1,\cdots,n}}\right)$$

$C_i,C_j,C_l,(j,l\neq i)$都是场景 S_k 中的构件，同样，PS_k 是场景 S_k 的执行概率，$|S|$是场景的总数，n 是应用系统中的构件数量，$|\mathrm{Interact}(C_i,C_j)|$是构件 C_i 和构件 C_j 在场景 S_k 中的交互次数。

图 10-5 是由 4 个构件组成的一个简单的 CDG。

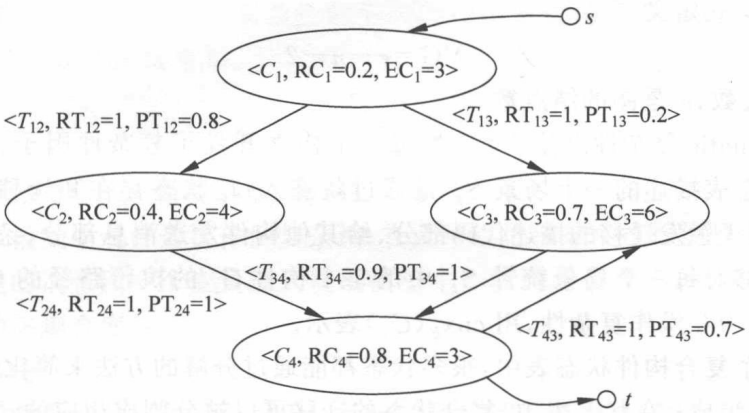


图 10-5　一个简单的 CDG

10.2.2　风险分析的步骤

本节的体系结构分析方法的主要步骤如下。

(1) 采用体系结构描述语言对体系结构进行建模。

(2) 通过模拟方法进行复杂性分析。

(3) 通过 FMEA 和模拟运行进行严重性分析。

(4) 为构件和连接件开发其启发式风险因子。

(5) 建立用于风险评估的 CDG。

(6) 通过图论中的算法进行风险评估和分析。

下面对上述步骤进行详细介绍。

1. 体系结构风险建模

ADL 能够形式化描述软件体系结构，并在详细设计之前，执行初步的分析以尽早发现一些问题，现在已经有很多用于软件体系结构描述的 ADL。

软件系统的动态行为是用于描述系统运行期间的行为。软件系统的风险可以和系统运行时的失效性相关联，因此，软件系统的风险可以通过对单个构件的行为和系统的动态行为进行分析和评估。本节介绍的风险分析方法主要关注软件体系结构的动态描述，它可以在两个构件进行交互时获取，也可以在单个构件的行为上获取。在此，选择的 ADL 必须对构件之间的交互和单个构件的行为提供支持。

UML作为对软件系统建模的标准语言,它定义了能够描述体系结构动态行为的模型规范。本节使用UML的状态图来描述单个构件的行为,使用UML的顺序图来描述构件之间的交互。同样,对于体系结构风险的建模也可以使用Rose Real-Time等工具。

2. 复杂性分析

传统上,系统可靠性的度量是通过计算某一特定时期的失效或者错误数量来衡量的。为了提高软件的质量和降低软件系统的风险,应该在软件生命周期的开发阶段就对可能存在高错误率的构件进行预测分析,在此把复杂性度量应用于风险评估技术。

1)构件的复杂性

软件的复杂性度量已经使用了很长一段时间,1976年,McCabe提出了作为系统易测试性、可维护性质量指标的Cyclomatic复杂性测量方法。Cyclomatic复杂性是基于规划图的,它可以通过下式定义。

$$\mathrm{VG}=e-n-2$$

其中,e是图的边数,n是图的结点数。

基于Cyclomatic复杂性度量方法,为每一个构件开发了复杂性因子。每一个执行场景,比如构件状态表描述的一个场景S_k是通过状态入口、状态存在和迁移触发来执行的。这个场景包括用于触发迁移的描述代码部分、给其他构件发送消息部分、条件检测、触发部分等。因此,能够对每一个场景统计S_k中的每个构件C_i的执行路径的Cyclomatic复杂性。这类度量称为互操作复杂性,用$\mathrm{cpx}_k(C_i)$表示。

有时,在一个复合构件状态表中,很多状态都能通过分解的方法来简化。一个复合状态由许多简单状态组成;在状态表中,复合状态的迁移可以被分割成相应的迁移片断,迁移点为相同迁移的不同迁移片断提供了一个基础。为了建立在两个复合状态间的Cyclomatic迁移复杂性,可以通过统计所有迁移片断的Cyclomatic复杂性,包括起始点的出口代码片断和终结点的入口代码片断。如图10-6所示,从状态s_{11}到状态s_{22}的一个迁移的VG是:

$$\mathrm{VG}_x(s_{11})+\mathrm{VG}_a(t_{11})+\mathrm{VG}_a(t_{12})+\mathrm{VG}_e(s_1)+\mathrm{VG}_a(t_{13})+\mathrm{VG}_e(s_{22})$$

其中,VG_x是出口编码片断的复杂性,VG_a是活动代码片断的复杂性,VG_e是入口代码片断的复杂性。

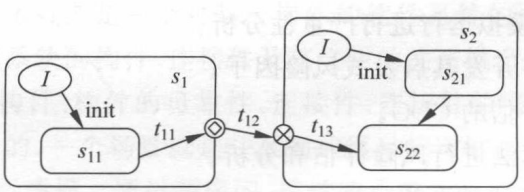


图10-6 两个复合状态之间的迁移图

为了模拟模型中获得的动态复杂性,为每一个构件加上一个复杂性变量,在每一个执行场景中,这些变量是根据每一个特定的触发执行场景进行更新的。在模拟结束时,模拟工具会报告每一个构件的动态复杂性的值。使用场景概率PS_k能够通过下式统计每一个构件的操作复杂性度量。

$$\mathrm{cpx}(C_i)=\sum_{k=1}^{|S|}\mathrm{PS}_k\times\mathrm{cpx}_k(C_i)$$

其中,$|S|$表示场景数。此时,通过该方法可以为每一个构件计算其对应的复杂度。

2）连接件的复杂性

为了计算体系结构中连接件的复杂性因子，可以使用动态耦合法来分析。动态耦合中包括输出动态耦合和输入动态耦合。假设 $EC_k(C_i,C_j)$ 是与构件 C_j 相关的构件 C_i 的输出耦合，它等于从构件 C_i 到构件 C_j 所发的消息数在场景 S_k 的执行期内所交换的消息总数的百分比，即：

$$EC_k(C_i,C_j)=\frac{|\{M_k(C_i,C_j)\mid C_i,C_j\in A\wedge C_i\neq C_j\}|}{MT_k}\times 100$$

$M_k(C_i,C_j)$ 是在场景 S_k 执行期间，从构件 C_i 到构件 C_j 所发送的消息集；MT_k 是在场景 S_k 执行期间，所有构件进行的消息交换总数。

$EC_k(C_i,C_j)$ 通过一个特定的执行场景来定义，并可以把该方法扩展到整个执行场景。因此，某一构件在给定的场景下的复杂性的平均值可以通过场景 PS_k 的执行概率来衡量，因此 $EC(C_i,C_j)$ 度量公式就变为：

$$EC(C_i,C_j)=\sum_{k=1}^{|S|}PS_k\times EC_k(C_i,C_j)$$

其中，$|S|$ 是场景总数，PS_k 是场景 S_k 的执行概率。

为了从模拟模型中统计连接件的复杂性，在此可以把在某个执行场景中，从一个构件到另一个构件发送的消息用一个变量加入模拟模型中。使用场景概率，就能够计算出每一个连接件的平均动态耦合度。

3. 严重性分析

由于构件的复杂性并不是一个用于估计失效风险的完整方法，因此必须考虑系统中的构件由于严重性或失效的危险程度所需要的特别开发资源。有些构件的复杂度很低，但它们却充当一个主要的安全角色，如果它们失效，可能导致一个灾难性的后果。此时，就必须考虑每一个构件的失效后果所产生的严重性。

通过每一个潜在的失效方式的级别和失效方式的后果来进行严重性分析。FMEA 技术是一个用于描述系统可能的失效方式和识别失效后果的系统方法。在分析失效方式时，分析者必须把每一个体系结构元素作为失效方式的焦点。首先是分析者识别体系结构元素的失效方式，并研究其影响，每一次失效的严重性级别和识别对系统的最坏影响。

1）失效方式的识别

为了简化起见，只考虑下面一些失效分析技术。

（1）单个构件的失效方式。在体系结构模拟模型中，每一个构件用一个状态图来描述，它是一个基于状态的构件行为描述。在识别单个构件的失效方式中，仅考虑功能性故障分析和基于状态的故障分析。

（2）单个连接件的失效方式。在体系结构模拟模型中，为了识别单个连接件的失效方式，仅考虑在消息参数与消息参数之间的接口错误失效误差。

顺序图可以获得构件之间的交互，通过识别构件和连接件的失效方式，可以得到执行场景的失效方式。在关注某一时刻特定场景中一个构件或连接件的角色之前，就使识别过程变得更容易。

2）严重性分级

在一个特定的失效严重性分级上，特定领域专家充当着重要角色。特定领域专家可以

通过他们的经验成果为特定领域建立相应的严重性分级制度。本节中通过以下方法对严重性进行分级。

(1) 灾难性的。一个错误可能导致整个系统的失败或毁灭。

(2) 危急的。一个错误可能导致严重的损坏、主要性能的破坏、主要系统的损坏,或者主要产品的失败。

(3) 边际性的。一个错误对性能、系统产生一个较小的损坏,或推迟产品的完成日期。

(4) 较小的。一个错误并不产生任何的损坏,但需要不定时地进行维护和修理。

4. 开发体系结构元素的可靠性风险因子

这部分内容主要是通过复杂性和严重性因素,为体系结构中的每一个构件和连接件计算其启发式风险因子。

体系结构的每一个构件的启发式风险因子可以通过下式进行计算。

$$\mathrm{hrf}_i = \mathrm{cpx}_i \times \mathrm{svrty}_i$$

其中,$\mathrm{cpx}_i \in [0,1]$是第 i 个构件的动态复杂性,$\mathrm{svrty}_i \in [0,1]$是第 i 个构件的严重性级别。

体系结构中每一个连接件的启发式风险因子可以通过下式进行计算。

$$\mathrm{hrf}_{ij} = \mathrm{cpx}_{ij} \times \mathrm{svrty}_{ij}$$

其中,$\mathrm{cpx}_{ij} \in [0,1]$是第 i 个构件到第 j 个构件之间的连接件的动态耦合度,可以通过 $\mathrm{EC}(C_i, C_j)$计算; $\mathrm{svrty}_{ij} \in [0,1]$是第 i 个构件到第 j 个构件之间的连接件的严重性级别。

5. CDG 的开发

上面对体系结构元素的可靠性风险因子进行了定量的描述,为了评估一个系统的风险值,需要定义一个风险聚合算法,在此算法中要利用 CDG 模型。CDG 模型可以通过下述步骤构造。

第一步,通过估计相对于整个场景而言的每一个场景执行的频率,来估计每个场景执行的概率。

第二步,对每一个场景中的构件,通过模拟报告来记录每一个构件的执行时间。这样就可以通过每一个构件在每个场景中的使用概率和所用时间,来估计每个构件的平均执行时间。

第三步,通过场景的使用概率和场景之间的迁移概率,来计算构件之间的迁移概率;给定场景下构件之间的迁移概率,是通过发送消息的构件所发送的消息在目标构件的消息中所占的比例来计算。

第四步,通过模拟,为每个构件、连接件估计复杂性因子和严重性指标,通过综合复杂性因子和严重性指标获取每个构件、连接件的风险因子。

6. 可靠性风险分析算法

体系结构的风险因子可以通过聚合单个构件和连接件的风险因子来获得。例如,假设存在一个有 L 个构件的执行序列,则该执行序列 L 的风险因子为:

$$\mathrm{HRF} = 1 - \prod_{i=1}^{L} (1 - \mathrm{hrf}_i)$$

CDG 模型构造完以后,就可以通过图 10-7 的算法使用构件、连接件的风险因子函数分析应用程序的风险。

Algorithm
Procedure Assess Risk
Parameters
consumes CDG, AE_{Appl} //AE_{Appl} 是平均执行时间
produces $Risk_{appl}$
Initialization:
$R_{appl}=R_{temp}=1$ //R_{temp} 是(1−RiskFactor)的临时变量
Time=0
Algorithm
push tuple$<C_1, hrf_1, EC_1>$, Time, R_{temp}
while Stack not EMPTY do
pop$<C_i, hrf_i, EC_i>$, Time, R_{temp}
if Time$>AE_{appl}$ or $C_i=t$;//t 为终结点
$R_{appl}+=R_{temp}$;
else
$\forall <Cj, hrfj, ECj> \in$ children(Ci)
push($<C_j, hrf_j, EC_j>$, Time$+=EC_i$, $R_{temp}=R_{temp}*(1-hrf_i)*(1-hrf_{ij})*PT_{ij}$)
end
endwhile
$Risk_{appl}=1-R_{appl}$
end Procdure Assess Risk

图 10-7　应用程序风险因子算法

该算法扩展了从开始结点开始的 CDG 的所有分支，树的宽度扩展表示逻辑“或”路径，因此转换为聚合风险因子的总数通过沿着每条路径的迁移概率来衡量。每条路径的深度表示顺序执行的构件序列，即逻辑“与”，因此就转换为风险因子的乘积。“与”路径考虑了连接件风险因子(hrf_{ij})，一个终结点路径的深度扩展是一个场景的平均执行时间的总和。由于依赖图的概率特性，在图中可能存在循环结构，但是通过使用场景的平均执行时间来终止图的迁移深度，这样循环就不会导致死锁的发生。因此，在执行该算法时死锁问题是不可能发生的，算法的终止也就很明显了。

该算法的复杂性高度依赖于在 CDG 中结点被访问的次数，一个结点的访问次数是一个与场景的平均执行时间、场景的剖面、构件的平均执行时间、在场景中构件交互方式和图的结点数相关的函数。风险因子很大程度上依赖于对体系结构的认识，因此构造出来的 CDG 并不是唯一的。

本节对使用定量的方法对软件体系结构的可靠性风险进行了分析，并说明了其所使用的相关方法、手段和工具。其主要思想是通过用 UML 建立软件体系结构场景模型，针对这些场景模型建立 CDG，然后进行构件、连接件的复杂性、严重性分析，最后通过可靠性风险分析算法计算整个软件体系结构的可靠性风险。

10.3　基于体系结构描述的软件测试

测试大型复杂的软件系统是一项困难且花费巨大的工作，但在软件开发和维护过程中却是一项非常重要的工作。如何尽早地开展软件测试工作，怎样将形式化方法与软件测试

技术结合起来,已成为软件测试研究的重点。由于软件体系结构描述语言具有形式化理论的基础,如Petri网、状态图、Z语言、CSP等,这为在体系结构级对系统进行分析和测试提供了理论基础和数学方法,从而在系统开发初期就能发现体系结构级的错误。

本节在软件体系结构描述语言的基础上,对构件之间交互所引起的错误进行测试,建立静态分析和动态测试模型。根据不同的体系结构抽象层次,基于不同的路径覆盖级别,生成测试用例,由此平衡开发成本、进度与质量需求之间的关系。

10.3.1 测试方法

软件体系结构测试与程序测试有所不同,它是检查软件设计的适用性,这种测试不考虑软件的实现代码,所以基于实现和说明的程序测试方法对软件体系结构测试并不适用。

1. 测试内容

与传统的软件测试一样,基于体系结构的软件测试也需要研究测试内容、测试准则、测试用例、测试充分性及测试方法等问题。

体系结构的描述必须满足一个基本的要求,即体系结构描述的各个部分必须相互一致,不能彼此冲突,因为体系结构主要关注系统的结构和组装,如果参与组装的各个部分之间彼此冲突,那么由此组装、细化和实现的系统一定不能工作。因此,体系结构的分析和测试主要考虑:构件端口行为与连接件约束是否一致、兼容,单元间的消息是否一致、可达,相关端口是否可连接,体系结构风格是否可满足。

为了保证测试的充分性,必须对关联构件、连接件的所有端口行为和约束进行测试,即所有接口应该是连接的,数据流、控制流和命令应该是可达的,且在并发时应该保证无死锁发生。

2. 测试准则

在传统测试方法中,测试准则是基于实现和规约得到的,基于实现的测试准则是结构化的,它是利用软件的内部结构来定义测试数据以覆盖系统。例如,通过结构技术从程序代码中或通过功能化技术从规约说明中得到单元测试计划,基于数据流和控制流定义的语句覆盖和分支覆盖来测试准则。

近年来,在规约的基础上结构化准则得到了进一步的发展,并根据规约的语法、语义和结构定义了测试准则。软件体系结构描述语言在高层抽象层上的形式化的系统静态、动态特征及系统交互模型,为定义体系结构测试准则奠定了基础。基于体系结构的抽象模型,通过分析体系结构组成单元之间的关联性,可以定义如下的测试准则。

测试准则:测试应覆盖所有的构件及各个构件的接口、各个连接件的接口、构件之间的直接连接、构件之间的间接连接。

对于不同的ADL,这个测试准则有不同的演绎。对于特定的ADL,可以从该准则中定义测试需求并据此生成测试用例。

3. 测试需求和测试用例的生成

实现完整测试的典型方法是利用测试准则定义测试需求,进而生成测试用例。参照相关研究工作,可定义以下几种测试路径。

(1) 构件或连接件内部消息的传递路径。

(2) 构件或连接件内部端口的执行顺序路径。

(3) 构件之间到连接件或连接件到构件的消息传递路径。

(4) 构件之间的直接连接路径。

(5) 构件之间的间接连接路径。

(6) 所有构件的连接路径。

在大多数的软件开发过程中，软件开发的成本和周期制约着软件的质量，三者之间的关系需要平衡。根据不同的体系结构抽象层次，测试准则和覆盖准则的力度也有所不同。

软件体系结构测试过程可以分为单元测试、集成测试和系统测试，单元测试是最底层的测试活动，指构件开发者对构件本身的测试，涉及的消息流是构件内部的消息，一般由构件开发者完成；集成测试的主要任务是测试构件之间的接口以保证构件能够交互，它将构件本身抽象为单元，并关注于构件间的消息传递，构件的交互行为可以通过形式化规约得到，因此这种测试可提前进行；系统测试的主要任务是测试整个系统能否正常运行，以保证系统符合其设计模型。

如图 10-8 所示，在不同阶段，测试关注的信息和特征也不相同，因此测试准则的级别也就不同，根据由低到高的逐步抽象过程，可定义测试准则的级别。

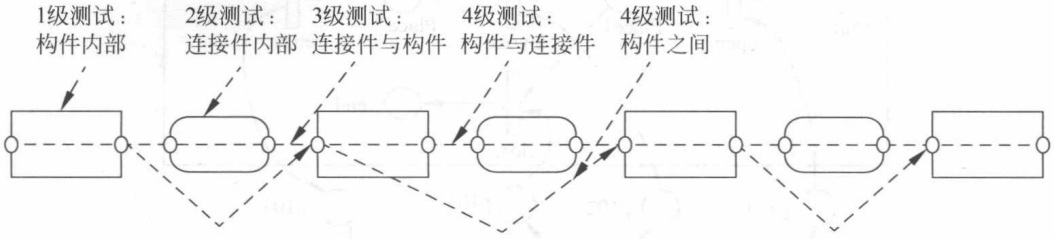


图 10-8　测试准则及测试级别

在测试过程中，根据测试准则级别，可选择不同的测试路径，生成测试用例。例如，如果要进行构件级测试，通过对构件行为的细化，即进一步展开构件，再利用第 1 级测试准则覆盖所有路径，从而得到该构件的测试用例。对于集成测试，可利用第 3 级测试准则覆盖图 10-8 中的构件库所、变迁和弧的路径，从而得到集成测试的测试用例。对于系统测试，需覆盖图中的路径获得测试用例。

10.3.2　实例与实现

以客户-服务器结构为例，Client 和 Server 这两个构件通过客户服务器协议连接，连接件有两个角色 Role(C)，Role(S)，它们分别负责与 Client 端和 Server 端的连接，Role(C)的事件集为{开始，请求，接收结果，成功}，Role(S)的事件集为{开始，激发，返回结果，成功}。

客户-服务器遵循以下规则。

(1) Client 首先初始化进程并与 Server 连接，一旦连接，Client 端通过 Role(C)向 Server 发出请求。

(2) Server 在收到 Client 端连接请求后，与 Client 端进行连接，当请求被激发后，将请求结果返回给 Client 端，然后等待 Client 端的其他请求。

(3) Client 端接收结果后，可以发出更多的请求或终止连接，如果 Client 端终止了连接，Server 也就中断了与 Client 的连接。

Server 端的应用构件描述如下。

Type Appl is interface

External Action Request(Msg:String)

Public Action Results(Msg:String)

Behavior

(∃M∈Msgs(Appl)) ∧ Receive(Appl,M)⇒Results(Appl,M)

Cons(client. open|sever. open) ∧ (pre_cond=connected ∧

(invoke⇒return) ∧ (client. close|sever. close))

End Appl

其中,External 是构件所需要的环境集合,Public 是构件能提供给环境或其他构件的功能集合,Behavior 是构件行为语义描述,Msgs 是消息集合,Cons 是构件行为约束,Pre_cond 表示前置条件。它们所描述的动态行为如图 10-9 所示。

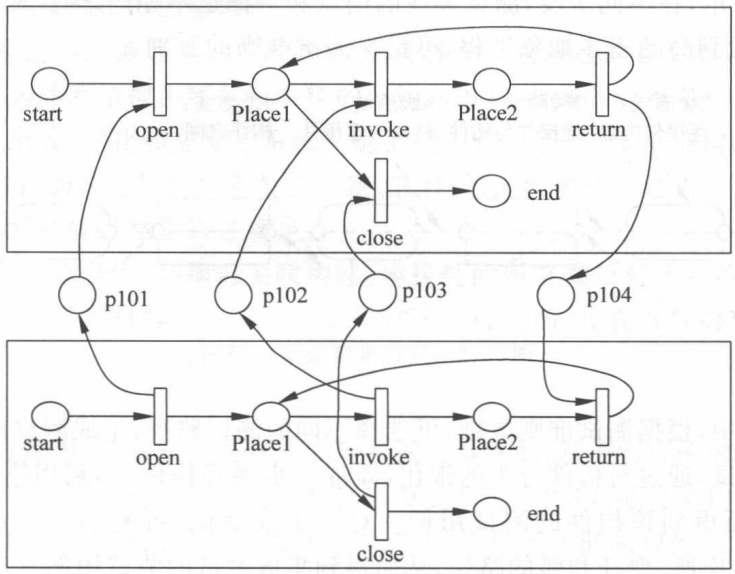


图 10-9　客户端、服务器行为图

以 Client 为例,若在构件内测试,测试路径应为:

Client. start→Client. open→Client. place1→Client. invoke→Client. place2→Client. return→Client. place1→Client. close→Client,结束。Client. start→Client. open→Client. place1→Client,结束。

当进行 Client-Server 集成测试时,根据图 10-9 测试路径应该为:

Client. start→Client. open→p101→ Server. open;

Client. invoke→p102→Sever. invoke;

Client. close→p103→Server. close;

Server. return →p104→Client. return。

根据路径覆盖准则,覆盖以上路径即可生成其测试用例。若构件之间有并发请求,同时有多个信息交互,同样可以建立系统的动态行为模型,再根据不同的测试准则,可生成满足要求的测试用例。

思 考 题

1. 什么是软件体系结构的可靠性？软件体系结构的可靠性可以从哪些方面进行评估？

2. 为什么要对软件体系结构进行风险分析？请结合一个实际项目，采取风险分析方法对软件体系结构进行评估。

3. 你是如何理解基于体系结构的软件测试的？结合实际项目设计一个基于体系结构的测试用例。

参考文献

第11章 软件体系结构评估

众所周知，软件体系结构的设计是整个软件开发过程中关键的一步。对于当今世界上庞大而复杂的系统来说，没有一个合适的体系结构而要有一个成功的软件设计几乎是不可想象的。不同类型的系统需要不同的体系结构，甚至一个系统的不同子系统也需要不同的体系结构。体系结构的选择往往会成为一个系统设计的关键。

但是，怎样才能知道为软件系统所选用的体系结构是恰当的呢？如何确保按照所选用的体系结构能顺利地开发出成功的软件产品呢？要回答这些问题并不容易，因为它受很多因素的影响，需要用专门的方法对其进行评估。

11.1 软件体系结构评估概述

本节首先介绍软件质量属性以及软件体系结构评估的几个基本概念，然后对软件体系结构的几种评估方式进行简单介绍和对比。

11.1.1 软件质量属性

体系结构评估可以只针对一个体系结构，也可以针对一组体系结构。在体系结构评估过程中，评估人员所关注的是系统的质量属性，所有评估方法所普遍关注的质量属性有以下几个。

1. 性能

性能(performance)是指系统的响应能力，即要经过多长时间才能对某个事件做出响应，或者在某段时间内系统所能处理的事件的个数。经常用单位时间内所处理事务的数量或系统完成某个事务处理所需的时间对性能进行定量的表示。性能测试经常要使用基准测试程序(用以测量性能指标的特定事务集或工作量环境)。

2. 可靠性

可靠性(reliability)是软件系统在应用或系统错误面前，在意外或错误使用的情况下维持软件系统的功能特性的基本能力。可靠性是最重要的软件特性，通常用它衡量在规定的条件和时间内，软件完成规定功能的能力。可靠性通常用平均失效等待时间(Mean Time To Failure，MTTF)和平均失效间隔时间(Mean Time Between Failure，MTBF)来衡量。在失效率为常数和修复时间很短的情况下，MTTF 和 MTBF 几乎相等。可靠性可以分为以下两个方面。

(1) 容错。其目的是在错误发生时确保系统正确的行为，并进行内部“修复”。例如，在一个分布式软件系统中失去了一个与远程构件的连接，接下来恢复了连接。在修复这样的

错误之后，软件系统可以重新或重复执行进程间的操作直到错误再次发生。

(2) 健壮性。这里说的是保护应用程序不受错误使用和错误输入的影响，在遇到意外错误事件时确保应用系统处于已经定义好的状态。值得注意的是，和容错相比，健壮性并不是说在错误发生时软件可以继续运行，它只能保证软件按照某种已经定义好的方式终止执行。软件体系结构对软件系统的可靠性有巨大的影响。例如，软件体系结构通过在应用程序内部包含冗余或集成监控构件和异常处理，来支持可靠性。

3. 可用性

可用性(availability)是指系统能够正常运行的时间比例。经常用两次故障之间的时间长度或在出现故障时系统能够恢复正常的速度来表示。

4. 安全性

安全性(security)是指系统在向合法用户提供服务的同时能够阻止非授权用户使用的企图或拒绝服务的能力。安全性是根据系统可能受到的安全威胁的类型来分类的。安全性又可划分为机密性、完整性、不可否认性、可控性等特性。其中，机密性保证信息不泄露给未授权的用户、实体或过程；完整性保证信息的完整和准确，防止信息被非法修改；可控性保证对信息的传播及内容具有控制的能力，防止为非法者所用。

5. 可修改性

可修改性(modifiability)是指能够快速地以较高的性能价格比对系统进行变更的能力。通常以某些具体的变更为基准，通过考察这些变更的代价衡量可修改性。可修改性包含4个方面。

(1) 可维护性(maintainability)。这主要体现在问题的修复上：在错误发生后“修复”软件系统。为可维护性做好准备的软件体系结构往往能做局部性的修改并能使对其他构件的负面影响最小化。

(2) 可扩展性(extendibility)。这一点关注的是使用新特性来扩展软件系统，以及使用改进版本来替换构件并删除不需要或不必要的特性和构件。为了实现可扩展性，软件系统需要松散耦合的构件。其目标是实现一种体系结构，它能使开发人员在不影响构件客户的情况下替换构件。支持把新构件集成到现有的体系结构中也是必要的。

(3) 结构重组(reassemble)。这一点处理的是重新组织软件系统的构件及构件间的关系。例如，通过将构件移动到一个不同的子系统而改变它的位置。为了支持结构重组，软件系统需要精心设计构件之间的关系。在理想情况下，它们允许开发人员在不影响实现的主体部分的情况下灵活地配置构件。

(4) 可移植性(portability)。可移植性使软件系统适用于多种硬件平台、用户界面、操作系统、编程语言或编译器。为了实现可移植，需要按照硬件无关的方式组织软件系统，其他软件系统和环境被提取出。可移植性是系统能够在不同计算环境下运行的能力。这些环境可能是硬件、软件，也可能是两者的结合。在关于某个特定计算环境的所有假设都集中在一个构件中时，系统是可移植的。如果移植到新的系统需要做些更改，则可移植性就是一种特殊的可修改性。

6. 功能性

功能性(functionality)是系统所能完成所期望的工作的能力。一项任务的完成需要系统中许多或大多数构件的相互协作。

7. 可变性

可变性(changeability)是指体系结构经扩充或变更而成为新体系结构的能力。这种新体系结构应该符合预先定义的规则,在某些具体方面不同于原有的体系结构。当要将某个体系结构作为一系列相关产品(如软件产品线)的基础时,可变性是很重要的。

8. 可集成性

可集成性(integrability)是指系统能与其他系统协作的程度。

9. 互操作性

作为系统组成部分的软件不是独立存在的,经常与其他系统或自身环境相互作用。为了支持互操作性(interoperation),软件体系结构必须为外部可视的功能特性和数据结构提供精心设计的软件入口。程序和用其他编程语言编写的软件系统的交互作用就是互操作性的问题,这种互操作性也影响应用的软件体系结构。

11.1.2 几个基本概念

为了后面讨论的需要,先介绍几个概念。

1. 敏感点和权衡点

敏感点(sensitivity point)和权衡点(tradeoff point)是关键的体系结构决策。

敏感点是一个或多个构件(和/或构件之间的关系)的特性。研究敏感点可使设计人员或分析员明确在搞清楚如何实现质量目标时应注意什么。

权衡点是影响多个质量属性的特性,是多个质量属性的敏感点。例如,改变加密级别可能会对安全性和性能产生非常重要的影响。提高加密级别可以提高安全性,但可能要耗费更多的处理时间,影响系统性能。如果某个机密消息的处理有严格的时间延迟要求,则加密级别可能就会成为一个权衡点。

2. 风险承担者

风险承担者(stakeholders)在项目管理领域中通常翻译为"项目干系人"或"涉众"。系统的体系结构涉及很多人的利益,这些人都对体系结构施加各种影响,以保证自己的目标能够实现。表11-1列出了在体系结构评估中可能涉及的一些风险承担者及其所关心的问题。

表11-1 体系结构评估中的风险承担者

风险承担者	定　义	所关心的问题
系统的生产者		
体系结构设计师	负责软件体系结构以及在相互竞争的质量需求间进行权衡的人	对其他风险承担者提出的质量需求的缓解和调停
开发人员	设计人员或程序员	体系结构描述的清晰与完整、各部分的内聚性与受限耦合、清楚的交互机制
维护人员	系统初次部署完成后对系统进行更改的人	可维护性,确定出某个更改发生后必须对系统中哪些地方进行改动的能力
集成人员	负责构件集成和组装的开发人员	与开发人员相同
测试人员	负责系统测试的开发人员	集成、一致的错误处理协议,受限的构件耦合、构件的高内聚性、概念完整性

续表

风险承担者	定　义	所关心的问题
系统的生产者		
标准专家	负责所开发软件必须满足的标准细节的开发人员	对所关心问题的分离、可修改性和互操作性
性能工程师	分析系统的工作产品以确定系统是否满足其性能及吞吐量需求的人员	易理解性、概念完整性、性能、可靠性
安全专家	负责保证系统满足其安全性需求的人员	安全性
项目经理	负责为各小组配置资源、保证开发进度、保证不超出预算的人员，负责与客户沟通	体系结构层次清晰，便于组建小组；任务划分结构、进度标志、最后期限等
产品线经理	设想该体系结构和相关资产怎样在该组织的其他开发中得以重用的人员	可重用性，灵活性
系统的消费者		
客户	系统的购买者	开发的进度、总体预算、系统的有用性、满足需求的情况
最终用户	所实现系统的使用者	功能性、可用性
应用开发者（对产品体系结构而言）	利用该体系结构及其他已有可重用构件，通过将其实例化而构建产品的人员	体系结构的清晰性、完整性、简单交互机制、简单裁减机制
任务专家、任务规划者	知道系统将会怎样使用以实现战略目标的客户代表	功能性、可用性、灵活性
系统服务人员		
系统管理员	负责系统运行的人员	容易找到可能出现问题的地方
网络管理员	管理网络的人员	网络性能、可预测性
技术支持人员	为系统在该领域中的使用和维护提供支持的人员	使用性、可服务性、可裁减性
接触系统或与系统交互的人		
领域代表	类似系统或所考察系统将要在其中运行的系统的构建者或拥有者	可互操作性
体系结构设计师	整个系统的体系结构设计师，负责在软件和硬件之间进行权衡并选择硬件环境的人	可移植性、灵活性、性能和效率
设备专家	熟悉该软件必须与之交互的硬件的人员，能够预测硬件技术的未来发展趋势的人员	可维护性、性能

3. 场景

在进行体系结构评估时，首先要精确地得出具体的质量目标，并以之作为判定该体系结构优劣的标准。为得出这些目标而采用的机制叫作场景(scenarios)。场景是从风险承担者的角度对与系统的交互的简短描述。在体系结构评估中，一般通过刺激(stimulus)、环境(environment)和响应(response)三方面对场景进行描述。

刺激是场景中解释或描述风险承担者怎样引发与系统的交互部分。例如，用户可能会激发某个功能，维护人员可能会做某个更改，测试人员可能会执行某种测试等，这些都属于

对场景的刺激。

环境描述的是刺激发生时的情况。例如,当前系统处于什么状态,有什么特殊的约束条件,系统的负载是否很大,某个网络通道是否出现了阻塞,等等。

响应是指系统如何通过体系结构对刺激做出反应的。例如,用户所要求的功能是否得到满足,维护人员的修改是否成功,测试人员的测试是否成功,等等。

11.1.3 评估的主要方式

从目前已有的软件体系结构评估技术来看,某些技术通过与经验丰富的设计人员交流获取他们对待评估软件体系结构的意见;某些技术对针对代码的质量度量进行扩展以自底向上地推测软件体系结构的质量;某些技术把对系统的质量的需求转换为一系列与系统的交互活动,分析软件体系结构对这一系列活动的支持程度等。尽管看起来它们采用的评估方式各不相同,但基本可以归纳为三类主要的评估方式:基于调查问卷或检查表的评估方式、基于场景的评估方式和基于度量的评估方式。

1. 基于调查问卷或检查表的评估方式

卡耐基梅隆大学的软件工程研究所(CMU/SEI)的软件风险评估过程采用了这一方式。调查问卷是一系列可以应用到各种体系结构评估的相关问题,其中有些问题可能涉及体系结构的设计决策;有些问题涉及体系结构的文档,如体系结构的表示用的是何种 ADL;有的问题针对体系结构描述本身的细节问题,如系统的核心功能是否与界面分开。检查表中也包含一系列比调查问卷更细节和具体的问题,它们更趋向于考察某些关心的质量属性。例如,对实时信息系统的性能进行考察时,很可能问到系统是否反复多次地将同样的数据写入磁盘等。

这一评估方式比较自由灵活,可评估多种质量属性,也可以在软件体系结构设计的多个阶段进行。但是由于评估的结果很大程度上来自评估人员的主观推断,因此不同的评估人员可能会产生不同甚至截然相反的结果,而且评估人员对领域的熟悉程度、是否具有丰富的相关经验也成为评估结果是否正确的重要因素。尽管基于调查问卷与检查表的评估方式相对比较主观,但由于系统相关的人员的经验和知识是评估软件体系结构的重要信息来源,因而它仍然是进行软件体系结构评估的重要途径之一。

2. 基于场景的评估方式

场景是一系列有序的使用或修改系统的步骤。基于场景的方式由 SEI 首先提出并应用在体系结构权衡分析方法(Architecture Tradeoff Analysis Method,ATAM)和软件体系结构分析方法(Software Architecture Analysis Method,SAAM)中。这种软件体系结构评估方式分析软件体系结构对场景也就是对系统的使用或修改活动的支持程度,从而判断该体系结构对这一场景所代表的质量需求的满足程度。例如,用一系列对软件的修改反映易修改性方面的需求,用一系列攻击性操作代表安全性方面的需求,等等。

这一评估方式考虑到了包括系统的开发人员、维护人员、最终用户、管理人员、测试人员等在内的所有与系统相关的人员对质量的要求。基于场景的评估方式涉及的基本活动包括确定应用领域的功能和软件体系结构之间的映射,设计用于体现待评估质量属性的场景以及分析软件体系结构对场景的支持程度。

不同的应用系统对同一质量属性的理解可能不同。例如,对操作系统来说,可移植性被

理解为系统可在不同的硬件平台上运行；而对于普通的应用系统而言，可移植性往往是指该系统可在不同的操作系统上运行。由于存在这种不一致性，对一个领域适合的场景设计在另一个领域未必合适，因此基于场景的评估方式是特定于领域的。这一评估方式的实施者一方面需要有丰富的领域知识以对某一质量需求设计出合理的场景；另一方面，必须对待评估的软件体系结构有一定的了解以准确判断它是否支持场景描述的一系列活动。

3. 基于度量的评估方式

度量是指为软件产品的某一属性所赋予的数值，如代码行数、方法调用层数、构件个数等。传统的度量研究主要针对代码，但近年来也出现了一些针对高层设计的度量，软件体系结构度量即是其中之一。代码度量和代码质量之间存在着重要的联系，类似地，软件体系结构度量应该也能够作为评判质量的重要依据。赫尔辛基大学提出的基于模式挖掘的面向对象软件体系结构度量技术、Karlskrona 和 Ronneby 提出的基于面向对象度量的软件体系结构可维护性评估、西弗吉尼亚大学提出的软件体系结构度量方法等都在这方面进行了探索，提出了一些可操作的具体方案。这类评估方式称作基于度量的评估方式。

上述基于度量的评估方式都涉及 3 个基本活动：首先需要建立质量属性和度量之间的映射原则，即确定怎样从度量结果推导出系统具有什么样的质量属性；然后从软件体系结构文档中获取度量信息；最后根据映射原则分析推导出系统的某些质量属性。因此，这些评估技术被认为都采用了基于度量的评估方式。

基于度量的评估方式提供更为客观和量化的质量评估。这一评估方式需要在软件体系结构的设计基本完成以后才能进行，而且需要评估人员对评估的体系结构十分了解，否则不能获取准确的度量。自动的软件体系结构度量获取工具能在一定程度上简化评估的难度。例如，MAISA 可从文本格式的 UML 图中抽取面向对象体系结构的度量。

4. 评估方式的比较

通过对 3 类主要的软件体系结构质量评估方式的分析，用表 11-2 从通用性、评估者对体系结构的了解程度、评估实施阶段、评估方式的客观程度等方面对这 3 类方式进行简单的比较。

表 11-2　3 类评估方式比较表

评估方式	调查问卷或检查表		场景	度量
	调查问卷	检查表		
通用性	通用	特定领域	特定系统	通用或特定领域
评估者对体系结构的了解程度	粗略了解	无限制	中等了解	精确了解
实施阶段	早	中	中	中
客观性	主观	主观	较主观	较客观

11.2　ATAM 评估方法

使用 ATAM 方法对软件体系结构进行评估的目标是理解体系结构关于软件系统的质量属性需求决策的结果。ATAM 方法不但揭示了体系结构如何满足特定的质量目标（如性能和可修改性），而且还提供了这些质量目标是如何交互的，即它们之间是如何权

衡的。这些设计决策很重要,一直会影响整个软件生命周期,并且在软件实现后很难修改这些决策。

11.2.1 ATAM评估的步骤

整个ATAM评估过程包括9个步骤,按其编号顺序分别是描述ATAM方法、描述业务动机、描述体系结构、确定体系结构方法、生成质量属性效用树、分析体系结构方法、讨论和分级场景、分析体系结构方法(是第6步的重复)、描述评估结果。有时,可以修改这9个步骤的顺序以满足体系结构信息的特殊需求。也就是说,虽然这9个步骤按编号排列,但并不总是一个瀑布过程,评估人员可在这9个步骤中跳转或进行迭代。

下面分别介绍这9个步骤。

1. 描述ATAM方法

ATAM评估的第一步要求评估小组负责人向参加会议的风险承担者介绍ATAM评估方法。其中,风险承担者包括开发人员、维护人员、操作人员、终端用户、中间商、测试人员、系统管理员等所有与系统有关的人员。在这一步,要解释每个人将要参与的过程,并预留出解答疑问的时间,设置好其他活动的环境和预期结果。关键是要使每个人都知道要收集哪些信息,如何描述这些信息,将要向谁报告等。特别是要描述以下事项。

(1) ATAM方法步骤简介。

(2) 获取和分析技术:效用树的生成,基于体系结构方法的获取/分析,场景的映射等。

(3) 评估结果:所得出的场景及其优先级,用户理解/评估体系结构的问题,描述驱动体系结构的需求并对这些需求进行分类,所确定的一组体系结构方法和风格,一组所发现的风险点和无风险点、敏感点和权衡点。

2. 描述业务动机

参加评估的所有人员必须理解待评估的系统,在这一步,项目经理要从业务角度介绍系统的概况,下面给出了这种描述的一个框架。

> 业务环境/驱动描述(约12张幻灯片,45min)
>
> (1) 描述业务环境、历史、市场划分、驱动需求、风险承担者、当前需要以及系统如何满足这些需要(3或4张幻灯片)。
>
> (2) 描述业务方面的约束条件(推向市场的时间、客户需求、标准和成本等)(1~3张幻灯片)。
>
> (3) 描述技术方面的约束条件(COTS、与其他系统的互操作、所需要的软硬件平台、遗留代码的重用等)(1~3张幻灯片)。
>
> (4) 质量属性需求(系统平台、可用性、安全性、可修改性、互操作性、集成性和这些需求来自的业务需要)(2或3张幻灯片)。
>
> (5) 术语表(1张幻灯片)。

除了初步从高级抽象层介绍系统本身外,一般来说,还要描述如下几项。

(1) 系统最重要的功能需求。

(2) 技术、管理、经济或政治方面的约束条件。

(3) 业务目标和环境。

(4) 主要的风险承担者。

(5) 体系结构驱动因素(形成体系结构的主要质量属性目标)。

3. 描述体系结构

在这一步中,首席设计师或设计小组要对体系结构进行详略适当的介绍,这里的"详略适当"取决于多个因素,例如,有多少信息已经决定了下来,并形成了文档;可用时间是多少;系统面临的风险有哪些;等等。这一步很重要,将直接影响可能要做的分析及分析的质量。在进行更详细的分析之前,评估小组通常需要收集和记录一些额外的体系结构信息。在体系结构描述中,至少应该包括以下内容。

(1) 技术约束(例如,操作系统、硬件、中间件等)。

(2) 要与本系统交互的其他系统。

(3) 用以满足质量属性要求的体系结构方法。

这时,评估小组开始初步考察体系结构方法,如下所示的框架将有助于保证介绍恰当的内容,也有助于保证评估工作按计划进行。

体系结构描述(约 20 张幻灯片,60min)

(1) 驱动体系结构的需求(性能、可用性、安全性、可修改性、互操作性、集成性等),以及与这些需求相关的可度量的量和满足这些需求的任何存在的标准、模型或方法(2 或 3 张幻灯片)。

(2) 高层体系结构视图(4～8 张幻灯片)。

① 功能:函数、关键的系统抽象、领域元素及其依赖关系、数据流;

② 模块/层/子系统:描述系统功能组成的子系统、层、模块,以及对象、过程、函数及它们之间的关系(过程调用、方法使用、回调和包含等);

③ 进程/线程:进程、线程及其同步,数据流和与之相连的事件;

④ 硬件:CPU、存储器、外设/传感器,以及连接这些硬件的网络和通信设备。

(3) 所采用的体系结构方法或风格,包括它们所强调的质量属性和如何实现的描述(3～6 张幻灯片)。

(4) COTS 的使用,以及如何选择和集成(1 或 2 张幻灯片)。

(5) 介绍 1～3 个最重要的用例场景,如果可能,应包括对每个场景的运行资源的介绍(1～3 张幻灯片)。

(6) 介绍 1～3 个最重要的变更场景,如果可能,应描述通过变更构件、连接件或接口所带来的影响(1～3 张幻灯片)。

(7) 与满足驱动体系结构需求相关的体系结构问题或风险(2 或 3 张幻灯片)。

(8) 术语表(1 张幻灯片)。

4. 确定体系结构方法

ATAM 评估方法主要通过理解体系结构方法来分析体系结构,在这一步,由设计师确定体系结构方法,由分析小组捕获,但不进行分析。

ATAM 评估方法之所以强调体系结构方法和体系结构风格的确定,是因为这些内容代

表了实现最高优先级的质量属性的体系结构手段。也就是说,它们是保证关键需求按计划得以实现的手段。这些体系结构方法定义了系统的重要结构,描述了系统得以扩展的途径,对变更的响应,对攻击的防范以及与其他系统的集成等。

5. 生成质量属性效用树

在这一步中,评估小组、设计小组、管理人员和客户代表共同确定系统最重要的质量属性目标,并对这些质量目标设置优先级和细化。这一步很关键,它对以后的分析工作起指导作用。即使是体系结构级的分析,也并不一定是全局的,所以,评估人员需要集中所有相关人员的精力,注意体系结构的各个方面,这对系统的成败起关键作用。这通常是通过构建效用树的方式来实现的。

效用树的输出结果是对具体质量属性需求(以场景形式出现)的优先级的确定,这种优先级列表为 ATAM 评估方法的后面几步提供了指导,它告诉了评估小组该把有限的时间花在哪里,特别是该在哪里去考察体系结构方法与相应的风险、敏感点和权衡点。另外,效用树可使质量属性需求具体化,使评估小组和客户能精确地定义自己的需求。

图 11-1 是一棵效用树的样例。在图 11-1 中,"效用"是树的根结点,代表了系统的整体质量。质量属性构成了效用树的二级结点,典型的质量属性(如性能、可修改性、可用性和安全性)构成了效用的子结点。以此类推,可对每个质量属性依次展开。

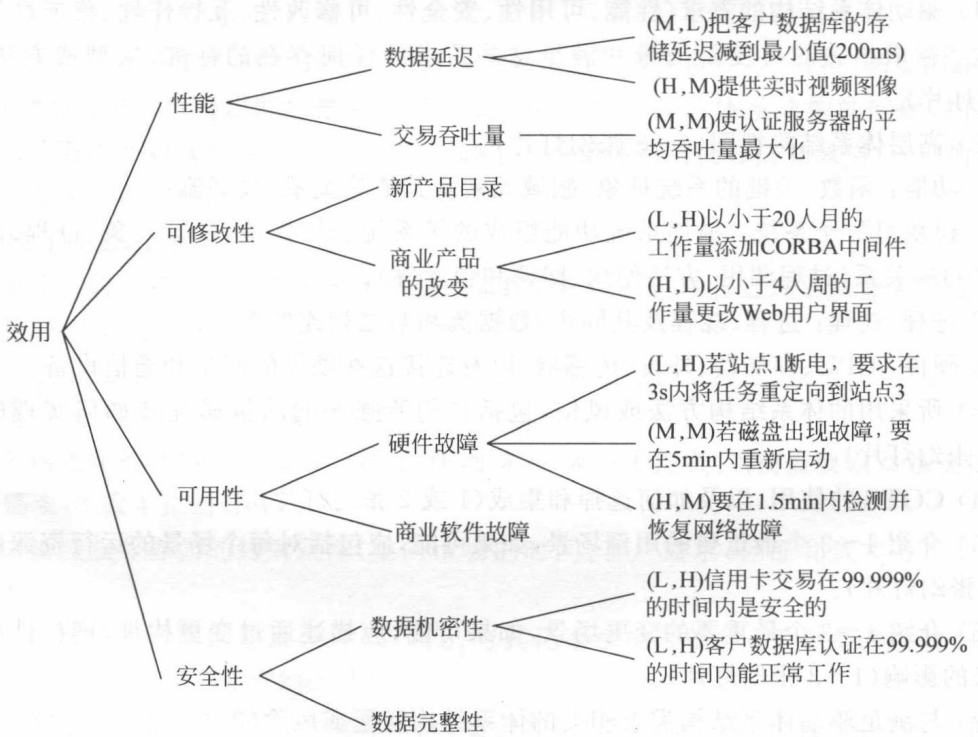


图 11-1 效用树样例

6. 分析体系结构方法

一旦有了效用树的结果,评估小组可以对实现重要质量属性的体系结构方法进行考察。这是通过注意文档化这些体系结构决策和确定它们的风险、敏感点、权衡点等来实现的。

在这一步中,评估小组要对每一种体系结构方法都考察足够的信息,完成与该方法有关

的质量属性的初步分析。这一步的主要结果是一个体系结构方法或风格的列表，与之相关的一些问题，以及设计师对这些问题的回答。通常产生一个风险列表、敏感点和权衡点列表。

事实上，效用树告诉评估人员考察体系结构的哪些方面（因为这是系统成败的关键因素），并希望设计师所做的响应中回答了这些需要。评估小组可以用效用树质量属性的问题更深入地考察相关的体系结构方法。这些问题能帮助评估小组：

（1）理解体系结构方法；

（2）找出该方法的缺陷，这些缺陷应该是大家都明了的；

（3）找出该方法的敏感点；

（4）发现与其他方法的交互和权衡点。

最后，上述各个方面都可能为风险描述提供基本素材，且被记录为不断扩展的风险决策列表中。

第 6 步的第一件事就是要把最高优先级的质量属性需求与实现它们的体系结构方法关联起来，设计人员应该为每一个效用树生成的高级场景确定构件、连接件、配置和约束。

评估小组和设计小组通过问一系列方法特定和质量属性特定的问题来实现每个体系结构方法，这些问题可能来自与风格有关的文档化的经验，也可能来自有关软件体系结构的书籍，或来自前人的经验。这些问题并不意味着工作的终结，恰恰相反，每个这样的问题都是一场讨论的起点，也是确定潜在的有无风险决策、敏感点和权衡点的开始。根据设计师的不同回答，这些问题还可能促进更深入的分析。例如，在一个客户-服务器体系结构的系统中，如果设计师不能刻画出客户端的负载情况，不能说明如何分配进程的优先级，进程如何分配给硬件等，那么进一步的分析就很有必要。

表 11-3 是一个捕获体系结构方法的框架。

表 11-3　捕获体系结构方法的框架

场景：＜来自效用树的一个场景＞			
属性：＜性能、安全性、可用性＞			
环境：＜对系统所依赖的环境的相关假设＞			
刺激：＜对该场景体现的质量属性刺激（故障、安全威胁、修改等）的精确描述＞			
响应：＜对质量属性响应的精确叙述（响应时间、修改难度等）＞			
体系结构决策	**风险**	**敏感度**	**权衡**
影响质量属性响应的体系结构决策列表	风险列表	敏感点编号	权衡点编号
……	……	……	……
……	……	……	……
推理： ＜关于为什么这组体系结构决策能够满足质量属性响应需求的定性或定量的推理＞			
体系结构图： ＜一个或多个体系结构图，在图中标注出上述推理的体系结构信息，可带解释性文字描述＞			

例如，针对需求具有高可用性的系统的效用树中的某个场景，第 6 步可能从设计师那里获得如表 11-4 所示信息。

表 11-4 体系结构方法分析

场景：S12(检测主 CPU 故障并恢复系统)			
属性：可用性			
环境：正常操作			
刺激：CPU 失效			
响应：可用切换概率为 0.999 999			
体系结构决策	**风险**	**敏感度**	**权衡**
备用 CPU	R8	S2	
无备用数据通道	R9	S3	T3
看门狗(watchdog)		S4	
心跳(heartbeat)		S5	
故障切换路由		S6	

推理：

(1) 通过使用不同的硬件和操作系统，保证没有通用模式故障。

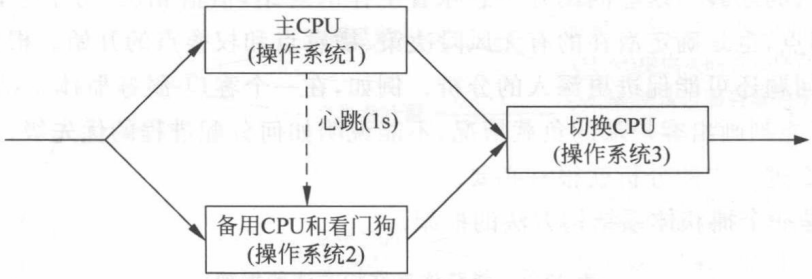
(2) 完成恢复时间最多不超过 4s(一个运算状态时间)。

(3) 基于心跳和看门狗的速度，保证在 2s 内能检测到故障。

(4) 看门狗是简单的和可靠的(已被证实过)。

(5) 由于没有备用数据通道，可用性需求可能存在风险。

体系结构图：



如表 11-4 所示，根据这一步的结果，评估人员可以确认和记录一组敏感点、权衡点、风险和非风险。所有敏感点和权衡点都是潜在的风险，在 ATAM 方法结束时，要按照是否有风险，对每个敏感点和权衡点进行分类，要分别将敏感点、权衡点、有风险决策和无风险决策列成一个单独的表。

在这一步结束时，评估小组应该对整个体系结构的绝大多数重要方面，所做出的关键设计决策、风险列表、敏感点、权衡点有一个清楚的认识。

7. 讨论和分级场景

场景在驱动 ATAM 测试阶段起主导作用。实践证明，当有很多风险承担者参与 ATAM 评估时，生成一组场景可为讨论提供极大的方便。场景是体系结构刺激示例，可用于以下两项活动。

(1) 描述风险承担者感兴趣的问题。

(2) 理解质量属性需求。

这时，风险承担者需进行两项相关的活动：集体讨论用例场景(描述风险承担者期望使用系统的方式)和改变场景(描述风险承担者所期望的系统在将来变更的方式)。用例场景是场景的一种。在用例场景中，风险承担者是一个终端用户，使用系统执行一些功能。改变

场景代表系统的变更，可分为成长场景和考察场景两类。

成长场景描述的是体系结构在中短期的改变，包括期望的修改、性能或可用性的变更、移植性、与其他软件系统的集成等。考察场景描述的是系统成长的一个极端情形，即体系结构由下列情况所引起的改变：根本性的性能或可用性需求（如数量级的改变）、系统基础结构或任务的重大变更等。成长场景能够使评估人员看清在预期因素影响系统时，体系结构所表现出来的优缺点；而考察场景则试图找出敏感点和权衡点，这些点的确定有助于评估者评估系统质量属性的限制。

一旦收集了若干场景，就必须设置优先级。评估人员可通过投票表决的方式来完成，每个风险承担者分配相当于总场景数的30%的选票，且此数值只入不舍。例如，如果共有17个场景，则每个风险承担者将拿到6张选票，这6张选票的具体使用则取决于风险承担者，他可以把这6张票全部投给某一个场景，或者每个场景投两张或三张票，还可以一个场景一张票等。

一旦投票结果确定，所有场景就可设置优先级。设置优先级和投票的过程既可公开也可保密。

表11-5是对某车辆调度系统进行评估时所得到的几个场景及其得票情况。

表11-5　场景得票表

场景编号	场景描述	得票数量
4	在10min内动态地对某次任务重新安排	28
27	把对一组车辆的管理分配给多个控制站点	26
10	在不重新启动系统的情况下，改变已开始任务的分析工具	23
12	在发出指令后10s内，完成对不同车辆的重新分配，以处理紧急情况	13
14	在6人月内将数据分配机制从CORBA改变为新兴的标准	12

这时，评估人员可以暂停，把对设置场景优先级的结果和第5步中效用树的结果进行比较，找出其中的相同之处和不同之处。在高优先级的场景和效用树中的高优先级的结点之间的任何差异都必须重新调整，至少要得到合理的解释。重新调整可能需要改变一个场景含义的清晰性或场景的优先级。解释可能需要理解在效用树中设置优先级的标准和在场景中设置优先级的标准的异同。不管采用哪种方法，这都是一个绝好的机会，可以用来保证每个风险承担者的需求都得到很好的理解，且所有需求之间不存在矛盾。

例如，表11-6不仅给出了表11-5中列出的高优先级的场景，而且还给出了每个场景影响最大的一个或多个质量属性。

表11-6　场景与质量属性

场景编号	得票数量	质量属性
4	28	性能
27	26	性能、可修改性、可用性
10	23	可修改性
12	13	性能
14	12	可修改性

生成效用树和讨论场景的活动反映了质量属性目标，但通过不同的推导途径，通常会着重于不同的风险承担者的需求。系统设计师和关键的开发人员常常只创建初步的效用树，

而在产生场景和对场景设置优先级时,有很多风险承担者参与。把这两步得出的高优先级场景进行对比,经常可以揭示出体系结构设计师认为重要的质量属性与大多数风险承担者认为重要的质量属性之间的差异。这种情况可以通过加强设计师所没有注意到的方面,从而发现体系结构中的重大风险。

8. 分析体系结构方法

在收集并分析了场景之后,设计师就可把最高级别的场景映射到所描述的体系结构中,并对相关的体系结构如何有助于该场景的实现做出解释。

在这一步中,评估小组要重复第6步中的工作,把新得到的最高优先级场景与尚未得到的体系结构工作产品对应起来。在第7步中,如果未产生任何在以前的分析步骤中都没有发现的高优先级场景,则第8步就是测试步骤。

9. 描述评估结果

最后,要把ATAM分析中所得到的各种信息进行归纳,并反馈给风险承担者。这种描述一般要采用辅以幻灯片的形式,也可以在ATAM评估结束之后,提交更完整的书面报告。

在描述过程中,评估负责人要介绍ATAM评估的各个步骤,以及各步骤中得到的各种信息,包括业务环境、驱动需求、约束条件、体系结构等。最重要的是要介绍ATAM评估的结果:

(1) 已文档化了的体系结构方法/风格;

(2) 场景及优先级;

(3) 基于属性的问题;

(4) 效用树;

(5) 所发现的风险决策;

(6) 已文档化了的无风险决策;

(7) 所发现的敏感点和权衡点。

11.2.2 ATAM评估的阶段

目前为止,介绍了ATAM评估方法的各个步骤,下面将介绍ATAM方法的各个步骤是如何随着时间的推移而展开的,可大致分为两个阶段。第一个阶段以体系结构为中心,重点是获取体系结构信息并进行分析。第二个阶段以风险承担者为中心,重点是获取风险承担者的观点,验证第一个阶段的结果。

之所以要分为两个阶段,是因为评估人员要在第一个阶段收集信息。在整个ATAM评估过程中,评估小组中的部分人(通常是1~3人)要与体系结构设计师和一个或两个其他关键的风险承担者(如项目经理,客户经理,市场代表)一起工作,收集信息。对支持分析而言,在大多数情况下,这种信息是不完整的或不适当的,所以,评估小组必须与体系结构设计师协作引导出必需的信息,这种协作通常要花几周的时间。当评估人员觉得已经收集了足够的信息,并已把这些信息记录成文档,就可进入第二个阶段了。

1. 第一个阶段的工作

ATAM评估小组要与提交待评估的体系结构的小组见面(或许这是双方第一次会见),这一会议有两方面的目的,一是组织和安排以后的工作,二是收集相关信息。从组织角度来看,体系结构小组负责人要确保让合适的人选参加后续会议,还要确保这些人为参加相关会

议做了充分的准备，抱着正确的态度。

第一天通常作为整个 ATAM 过程的一个缩影，主要关注 1～6 步的工作。第一次会议所收集的信息意味着要确保体系结构能得到正确的评估。同时，在第一次会议也会收集和分析一些初步的场景，作为理解体系结构、需要收集和提交的信息、所产生的场景的含义的一种途径。

例如，第一天，体系结构设计师可能提交部分体系结构，确定部分体系结构风格或方法，创建初步的效用树，就选定的一组场景进行工作，展示每个场景是如何影响体系结构的（如可修改性），体系结构又是如何做出响应的（例如，对质量属性而言，可以是性能、安全性和可用性）。其他的风险承担者（关键开发人员、客户、项目经理等）可以描述业务环境、效用树的构建，以及产生场景的过程。

第一个阶段是一个小型会议，评估小组需要尽可能多地收集有关信息，这些信息用来决定：

（1）后续评估工作是否可行，能否顺利进行；

（2）是否需要更多的体系结构文档。如果需要，则应明确需要哪些类型的文档，如何提交这些文档；

（3）哪些风险承担者应参与第二个阶段的工作。

这一天的最后，评估人员将对项目的状态和环境、驱动体系结构需求，以及体系结构文档有较清晰的认识。

在第一次会议和第二次会议之间有一段中断时间，其长短取决于第一个阶段完成的情况。在这段时间内，体系结构设计小组要和评估小组协作，做一些探索和分析工作。前面已经提到过，在第一个阶段评估小组并不构建详细的分析模型，而是构建一些初步模型，以使评估人员和设计人员能对体系结构有更充分的认识，从而保证第二个阶段的工作更有效率。另外，在这段时间内，还要根据评估工作的需要、可用人员的状况和计划来决定评估小组的最终人选。例如，如果待评估的系统对安全性的要求很高，则需要让安全专家参与评估工作；如果待评估的系统是以数据为中心的，则需要让数据库设计方面的专家参与评估。

2. 第二个阶段的工作

这时，体系结构已经被文档化，且有足够的信息来支持验证已经进行的分析和将要进行的分析。已经确定了参与评估工作的合适的风险承担者，并且给他们提供了一些书面阅读材料，如对 ATAM 方法的介绍，某些初步的场景，包括体系结构、业务案例和关键需求的系统文档等。这些阅读材料有助于保证风险承担者建立对 ATAM 评估方法的正确期望。

因为将有更多的风险承担者参与第二次会议，且因为在第一次会议和第二次会议之间，可能还要间隔几天或几个星期，所以第二个阶段首先有必要重新简单介绍 ATAM 方法，以使所有与会者达成共同的理解。另外，在每一步进行之前，简单扼要地介绍该步的工作，也是很有好处的。

3. ATAM 各步骤中相关的风险承担者

涉及 ATAM 评估的风险承担者少则只有 3～5 个，多则 40～50 个，并不是每一步都需要所有风险承担者的参与。根据待评估系统的规模大小、重要性和复杂性，评估小组可大可小。如果系统具有复杂的质量属性需求，或者用于一个复杂的、特殊的领域，则需要邀请领域专家参与评估，以壮大核心评估小组。另外，系统的特征还决定了邀请哪些客户代表参加评估。表 11-7 列出了 ATAM 方法中每个步骤的参与者的一个典型实例。

表 11-7 与风险承担者相关的 ATAM 步骤

步骤编号	所做的工作	风险承担者群体
1	描述 ATAM 方法	评估小组/客户代表/体系结构设计小组
2	描述业务动机	评估小组/客户代表/体系结构设计小组
3	描述体系结构	评估小组/客户代表/体系结构设计小组
4	确定体系结构方法	评估小组/客户代表/体系结构设计小组
5	生成质量属性效用树	评估小组/客户代表/体系结构设计小组
6	分析体系结构方法	评估小组/客户代表/体系结构设计小组
7	讨论和对场景进行分级	所有风险承担者
8	分析体系结构方法	评估小组/客户代表/体系结构设计小组
9	描述评估结果	所有风险承担者

4. ATAM 评估日程安排

虽然各个系统的 ATAM 评估有细微的区别,但其主要的活动是固定不变的。表 11-8 给出了一个典型的 ATAM 评估的日程安排。表中的每项工作后面的括号内都标出了对应的评估步骤(如果有)。虽然表中所列的时间不必原样照搬,但其时间安排代表了对可用时间的合理划分。

表 11-8 ATAM 评估方法日程安排表

开始时间	所做的工作	
第一天		第一个阶段
8:30	介绍/描述 ATAM 方法(第 1 步)	
10:00	客户描述业务动机(第 2 步)	
10:45	休息	
11:00	客户描述体系结构(第 3 步)	
12:00	确定体系结构方法(第 4 步)	
12:30	午餐	
13:45	生成质量属性效用树(第 5 步)	
14:45	分析体系结构方法(第 6 步)	
15:45	休息	
16:00	分析体系结构方法(第 6 步)	
17:00	休会	
中断几个星期		
第二天		第二个阶段
8:30	介绍/描述 ATAM 方法(第 1 步)	
9:15	客户描述业务环境/动机(第 2 步)	
10:00	休息	
10:15	客户描述体系结构(第 3 步)	
11:15	确定体系结构方法(第 4 步)	
12:00	午餐	
13:00	生成质量属性效用树(第 5 步)	
14:00	分析体系结构方法(第 6 步)	
15:30	休息	

续表

开始时间	所做的工作	
中断几个星期		
第二天		第二个阶段
15:45	分析体系结构方法(第 6 步)	
17:00	休会	
第三天		
8:30	介绍/扼要重述 ATAM 方法	
8:45	分析体系结构方法(第 6 步)	
9:30	讨论场景(第 7 步)	
10:30	休息	
10:45	设置场景的优先级(第 7 步)	
11:15	分析体系结构方法(第 8 步)	
12:30	午餐	
13:30	分析体系结构方法(第 8 步)	
14:45	准备汇报结果/休息	
15:30	描述结果(第 9 步)	
16:00	进一步的分析/角色的分配	
17:00	休会	

11.3 SAAM 评估方法

SAAM 方法是最早形成文档并得到广泛使用的软件体系结构分析方法,最初是用来分析体系结构的可修改性的,但实践证明,SAAM 方法也可用于对许多质量属性(可移植性、可扩充性、可集成性等)及系统功能进行快速评估。

与 ATAM 方法相比,SAAM 比较简单,这种方法易学易用,进行培训和准备的工作量都比较少。

11.3.1 SAAM 评估的步骤

图 11-2 给出了 SAAM 评估的步骤,在这些步骤进行之前,通常有必要对系统做简要的介绍,包括对体系结构的业务目标的说明等。

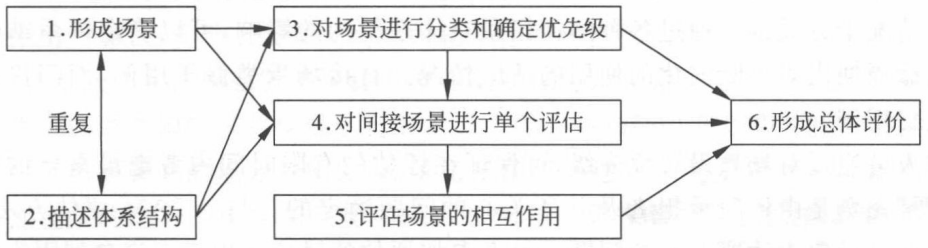


图 11-2 SAAM 评估步骤

1. 形成场景

在形成场景的过程中,要注意全面捕捉系统的主要用途、系统用户类型、系统将来可能的变更、系统在当前及可预见的未来必须满足的质量属性等信息。只有这样,形成的场景才

能代表与各种风险承担者相关的任务。

形成场景的过程也是集中讨论的过程。集体讨论能够使风险承担者在一个友好的氛围中提出一个个场景,这些场景反映了他们的需求,也体现了他们对体系结构将如何实现他们的需求的认识。某一个场景可能只反映一个风险承担者的需求,也可能反映多个(或多类)风险承担者的需求。例如,对于某个变更,开发人员关心的是实现该变更的难度和对性能的影响,而系统管理员则关心此变更对体系结构的可集成性的影响。在评估过程中,随着场景的不断提出,记录人员要把它们都记录在册,形成文档,供所有参加评估的人员查阅。

提出和收集场景的过程经常要重复两次或更多次。形成场景和描述体系结构的工作是相关联的,这两个步骤可重复进行,是一个迭代的过程。

2. 描述体系结构

在这一步,体系结构设计师应该采用参加评估的所有人员都能够充分理解的形式,对待评估的体系结构进行适当的描述。这种描述必须要说明系统中的运算和数据构件,也要讲清它们之间的联系。除了要描述这些静态特性外,还要对系统在某段时间内的动态特征做出说明。描述既可采用自然语言,也可采用形式化的手段。

场景的形成和对体系结构的描述通常是相互促进的。一方面,对体系结构的描述使风险承担者考虑针对所评估的体系结构的某些具体特征的场景;另一方面,场景也反映了对体系结构的需求,因此必须体现在体系结构的描述中。

3. 对场景进行分类和确定优先级

在SAAM评估中,场景就是对所期望的系统中某个使用情况的简短描述。体系结构可能直接支持该场景,即这一预计的使用情况不需要对体系结构做任何修改即可实现。这一般可以通过演示现有的体系结构在执行此场景时的表示来确定。在SAAM评估方法中称这样的场景为直接(direct)场景。也就是说,直接场景就是按照现有体系结构开发的系统能够直接实现的场景。与在设计时已经考虑过的需求相对应的直接场景并不会让风险承担者感到意外,但将增进对体系结构的理解,促进对性能、可靠性等其他质量属性的研究。

如果所评估的体系结构不能直接支持某一场景,就必须对所描述的体系结构做些更改。可能要对执行某一功能的一个或多个构件进行更改、为实现某一功能而增加一个构件、为已有构件建立某种新的联系、删除某个构件或某种联系、更改某一接口,或者是以上多种情况的综合。这样的场景叫作间接(indirect)场景。换句话说,间接场景就是需要对现有体系结构做些修改才能支持的场景。间接场景对于衡量体系结构对系统在演化过程中将出现的变更的适应情况十分关键。通过各种间接场景对体系结构的影响,可以确定体系结构在相关系统的生命周期内对不断演化的使用的适应情况。直接场景类似于用例,而间接场景有时也叫作变更用例(change case)。

评估人员通过对场景设置优先级,可保证在评估的有限时间内考虑最重要的场景。这里的"重要"完全是由风险承担者及其所关心的问题确定的。与ATAM评估方法一样,风险承担者通过投票表达所关心的问题。每个参加评估的风险承担者都将拿到固定数量的选票。向每个风险承担者发放的选票数一般是待评估场景数量的30%,他们可以用自己认为合适的方式投票,可把这些票全部投给某一个场景,或者每个场景投两张或三张票,还可以一个场景投一张票等。

一般而言,基于SAAM的评估关心的是诸如可修改性的质量属性,所以在划分优先级

之前要对场景进行分类。风险承担者最关心的通常是搞清间接场景对体系结构相应部分的影响。这是第4步要做的工作。

4. 对间接场景进行单个评估

一旦确定了要考虑的一组场景，就要把这些场景与体系结构的描述对应起来。对于直接场景而言，体系结构设计师需要讲清所评估的体系结构将如何执行这些场景；对于间接场景而言，体系结构设计师应说明需要对体系结构做哪些修改才能适应间接场景的要求。

SAAM评估也使评估人员和风险承担者更清楚地认识体系结构的组成及各构件的动态交互情况。风险承担者的讨论对于搞清场景描述的实际意义、搞清参评人员认为场景与质量属性的对应是否合适等都具有重要意义。这种对应的过程也能暴露体系结构及其文档的不足之处。

对每一个间接场景，必须列出为支持该场景而需要对体系结构所做的改动，并估计这些变更的代价。对体系结构的更改意味着引入某个新构件或新联系，或者需要对已有构件或联系的描述进行修改。在这一步将要结束时，应该给出全部场景的总结性列表。对每个间接场景，都应描述出要求做的更改，并由记录人员记录下来，形成文档。在这种描述中，应包括对完全实现每个更改的代价的估计(包括测试和调试的时间)。

表11-9给出了对某一实验工具进行SAAM评估的例子。

表11-9 SAAM场景评估示例

场景编号	场景描述	直接/间接	需要做的更改	更改/新增构件的数量	更改的工作量(估计)
7	更改底层的局域网，实现与广域网的通信	间接	抽象，到数据仓库的接口	1	1人月
8	改变数据项之间的关系(如添加将场景与风险承担者对应起来的能力)	间接	场景和风险承担者之间的代理	2	2人天
13	更改实体的数据结构(如为每个场景存储一个数据项)	间接	受影响实体的代理	1	1人天

5. 评估场景的相互作用

当两个或多个间接场景要求更改体系结构的同一个构件时，就称这些场景在这一组构件上相互作用。那么，为什么要强调场景的相互作用呢？

首先，场景的相互作用暴露了设计方案中的功能分配。语义上无关的场景的相互作用清楚地表明了体系结构中哪些构件运行着语义上无关的功能。场景交互比较多的地方很可能就是功能分离不够好的地方。所以，场景相互作用的地方也就是设计人员在以后的工作中应该多加注意的地方。场景相互作用的多少与结构复杂性、耦合度、内聚性等有关。例如，如果场景1和场景2是属于不同类别的，并且都影响构件X，那么构件X在结构划分方面可能存在着耦合问题。这时场景1和场景2的交互体现出系统结构没有很好地划分构件。另一方面，如果场景1和场景2是同一类的(如改变菜单字体的大小和颜色)，那么它们在构件X内部的交互反映出该模块具有良好的内聚性。

其次，场景的相互作用能够暴露体系结构设计文档未能充分说明的结构分解。如果场

景在某一构件内相互作用,但该构件实际上又分解成未表现出场景相互作用的子构件,就会出现这种文档描述不当的情况。如果真的出现了这种情况,则必须重新审核第2步(描述体系结构)的工作。

6. 形成总体评估

最后,评估人员要对场景和场景之间的交互做一个总体的权衡和评价,这一权衡反映该组织对表现在不同场景中的目标的考虑优先级。根据对系统成功的相对重要性为每个场景设置一个权值,权值的确定通常要与每个场景所支持的业务目标联系起来。

如果是要比较多个体系结构,或者针对同一体系结构提出了多个不同的方案,则可通过权值的确定得出总体评价。权值的设置具有很强的主观性,所以,应该让所有风险承担者共同参与,但也应合理组织,要允许对权值及其基本思想进行公开讨论。

同一个软件体系结构,对于有不同目的的组织来说,会得到不同的评价结果。例如,有些组织最关心系统的安全性,而有些则可能更关心系统的容错能力。不同的组织通过提出不同的场景来表明他们对系统的哪些方面特别关心,使用这些场景进行评价得出的结论也就比较适合他们的标准。

7. SAAM评估日程安排

上述6个步骤是关于SAAM评估中技术方面的问题,与ATAM评估方法类似,在进行SAAM评估时,也要考虑合作关系、准备工作等问题。需要对评估会议的时间做出安排、确定评估小组的人员组成、确定会议室、邀请各类风险承担者、编制会议日程等。这些工作都是必需的。表11-10给出了SAAM评估的一般日程安排。

表11-10 SAAM评估日程安排示例

时　间	事　项
第一天	
8:15—8:45	介绍,说明评估的目的,对评估方法的概要介绍,声明会议纪律
8:45—9:00	概要介绍要评估的系统,包括构架目标
9:00—10:00	形成场景(第1步)
10:00—10:30	休息
10:30—12:00	形成场景(第1步)
12:00—13:00	午餐
13:00—14:30	描述体系结构(第2步)
14:30—15:00	休息
15:00—15:30	形成场景(第1步)
15:30—17:00	对场景进行分类和确定优先级(第3步)
第二天	
8:15—10:00	对间接场景的单个评估(第4步)
10:00—10:30	休息
10:30—12:00	对间接场景的单个评估(第4步)
12:00—13:00	午餐
13:00—14:30	评估场景的相互作用(第5步)
14:30—15:30	形成总体评估(第6步)
15:30—16:00	休息
16:00—17:00	总结、报告

11.3.2 SAAM 评估实例

本节介绍一个使用 SAAM 方法的实例。所要介绍的系统是一个简单的在文章中查找和重组关键词(Key Word In Context,KWIC)的系统,该系统很小,很容易理解,且读者都很熟悉。KWIC 系统的基本功能是,输入一些句子,KWIC 系统把这些句子中的词语重新组合成新的句子,然后按字母顺序进行输出。例如:

```
输入:
predicting software quality
architecture level evaluation

则 KWIC 的输出为:
architecture level evaluation
evaluation architecture level
level evaluation architecture
predicting software quality
quality predicting software
software quality predicting
```

1. 定义角色和场景

KWIC 系统感兴趣的角色有两个,分别是最终用户和开发人员。共使用 4 个场景,其中两个场景经过了不同的最终用户的讨论:

(1) 修改 KWIC 程序,使之成为一个增量方式而不是批处理的方式。这个程序版本将能一次接收一个句子,产生一个所有置换的字母列表。

(2) 修改 KWIC 程序,使之能删除在句子前端的噪声单词(例如,前置词、代名词、连词等)。

使用的另外两个场景是经过开发人员讨论,但最终用户不知道的:

(1) 改变句子的内部表示(如压缩和解压缩);

(2) 改变中间数据结构的内部表示(例如,既可直接存储置换后的句子,也可存储转换后的词语的地址)。

2. 描述体系结构

第 2 步就是使用通用的表示对待评估的体系结构进行描述,这种描述是为了使评估过程更为容易。使评估人员知道体系结构图中的框或箭头的准确含义。

1) 共享内存的解决方案

在第一个待评估的体系结构中,有一个全局存储区域,被称作 Sentences,用来存储所有输入的句子。其执行的顺序是:输入例程读入句子→存储句子→循环转换例程转换句子→字母例程按字母顺序排列句子→输出。当需要时,主控程序传递控制信息给不同的例程。图 11-3 描述了这个过程,不同计算构件(computational component)上的数字代表场景编号,在这一步中可忽略(将在后面用到)。

2) 抽象数据类型解决方案

待评估的第二个体系结构使用抽象数据类型(Abstract Data Type,ADT),如图 11-4 所示。其中每个功能都隐藏和保护了其内部数据表示,提供专门的存取函数作为唯一的存储、

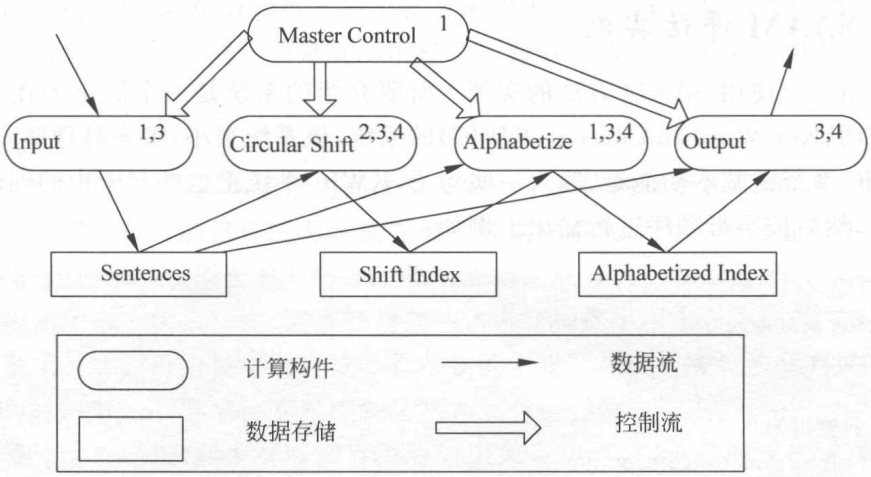


图 11-3　共享内存的解决方案示例

检索和查询数据的方式。ADT Sentences 有两个函数,分别是 set 和 getNext,用来增加和检索句子;ADT Shifted Sentences 提供了存取函数 setup 和 getNext,分别用来建立句子的循环置换和检索置换后的句子。

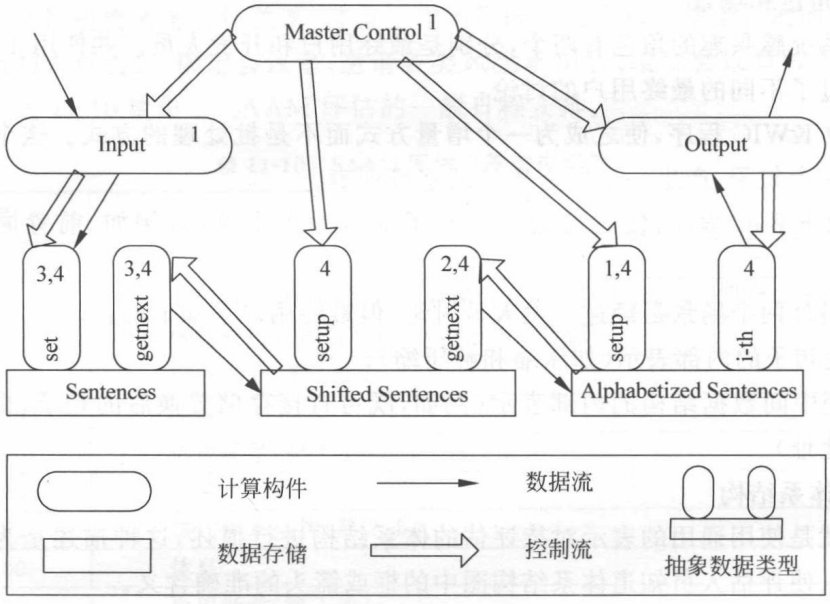


图 11-4　抽象数据类型解决方案示例

ADT Shifted Sentences 使用 ADT Sentences 的 getNext 函数来重新存储输入的句子。ADT Alphabetized Sentences 提供了一个 setup 函数和一个 i-th 函数,setup 函数重复调用 Shifted Sentences 的 getNext 函数,以检索已经存储的所有行和进行排序,i-th 函数根据参数 i,从存储队列中返回第 i 个句子。

3. 评估体系结构

既然已经把待评估的体系结构用通用的符号标记了出来,接下来就要评估体系结构满足场景的程度。通过依次考虑每个场景进行评估,所选择的用来评估的所有场景都是间接

场景，也就是说，这些场景不能被待评估的体系结构直接执行，因此评估依赖于体系结构的某些修改。

1）场景 1

第一个场景是从批处理模式转移到增量模式，也就是说，不是把所有句子都输入完后，再一次性进行处理，而是一次只处理一个句子。

对共享内存解决方案而言，这需要修改 Input 例程，使之在读入一个句子后让出控制权，同时，也要修改 Master Control 主控程序，因为子例程不再是按顺序一个只调用一次，而是一个迭代调用的过程。还要修改 Alphabetizer 例程，因为使用增量模式后，牵涉到插入排序的问题。假设 Circular Shift 例程一次只处理一个句子，且输出函数只要被调用，就可以输出。

注意，所做的假设只是针对共享内存解决方案而言的，一般来说，判断的准确性取决于不同的计算构件的内部工作知识。这也是为什么要期望评估人员中，既有计算构件一般知识的人，也有特定构件知识的人。

对 ADT 解决方案而言，Input 函数需要修改，使之在被调用时，一次只输入一行。假设 Sentences 当存储了输入之后放弃控制权，这无须改变。也假设当 Shifted Sentences 被调用时，能请求和转换所有可获得的句子，这样，该例程也无须改变。与在共享内存解决方案中一样，Alphabetized Sentences 也必须修改。

综上所述，对第一个场景而言，两个待评估的体系结构受到的影响是均等的，因此，判定其为中性。

2）场景 2

第二个场景要求删除句子中的“噪声”单词。无论在共享内存解决方案还是在 ADT 解决方案中，这种需求均可通过修改转换函数很容易地实现（在共享内存体系结构中，修改 Circular Shift 函数，在 ADT 体系结构中，修改 Shifted Sentences 函数）。因为在两种体系结构中，转换函数都是局部的，且噪声单词的删除不会影响句子的内部表示，所以，对两种体系结构而言，这种修改是等价的。

3）场景 3

第三个场景要求改变句子的内部表示，例如，从一个未压缩的表示转换到压缩的表示。在共享内存体系结构中，所有函数共享一个公用的表示，因此，除了主控函数 Master Control 外，所有函数都受该场景的影响。在 ADT 体系结构中，输入句子的内部表示由 Sentences 提供缓冲。因此，就第三个场景而言，ADT 体系结构比共享内存的体系结构要好。

4）场景 4

第四个场景要求改变中间数据结构的内部表示（例如，既可直接存储置换后的句子，也可存储转换后的词语的地址）。对于共享内存体系结构，需要修改 Circular Shift，Alphabetizer 和 Output 3 个例程。对于 ADT 体系结构，需要修改 ADT Shifted Sentences 和 Alphabetized Sentences。因此，ADT 体系结构解决方案所受影响的构件数量要比共享内存体系结构解决方案的少。

5）比较分析

图 11-3 和图 11-4 都标记了反映每个场景的影响。例如，在图 11-3 中，Master Control

构件中的"1"反映了该构件必须修改以支持场景1。检查待评估体系结构,看其有多少个构件受场景的影响,每个构件最多受多少个场景的影响。从这方面来看,ADT体系结构要比共享内存体系结构好。在共享内存体系结构和ADT体系结构中,两者都有4个构件受场景的影响,但是,在共享内存体系结构中,有两个构件(Circular Shift和Alphabetize)受3个场景的影响,而在ADT体系结构中,所有构件最多只受两个场景的影响。

6) 评估结果

表11-11概括了评估的结果,其中,0表示对该场景而言,两个体系结构是不分好坏的,在实际的评估中,还需要根据组织的偏好设置场景的优先级。例如,如果功能的增加是风险承担者最关心的问题(就像第二个场景一样),那么这两个体系结构是不相上下的,因为在这一点上,它们之间没有什么区别。

表11-11 评估结果概要

	场景1	场景2	场景3	场景4	比较
共享内存体系结构	0	0	−	−	−
ADT体系结构	0	0	+	+	+

但是,如果句子内部表示的修改是风险承担者最关心的问题(就像第三个场景一样),那么ADT体系结构显然是要首选的体系结构。

使用SAAM方法评估系统的结果通常容易理解,容易解释,而且和不同组织的需求目标联系在一起。开发人员、维护人员、用户和管理人员会找到各自所关心的问题的直接回答,只要这些问题是以场景的方式提出的。

SAAM最初提出的目的是对针对同一问题的不同的体系结构设计做比较。在许多的实例研究中,SAAM已经被证明是成功的。在这些实例研究中,不同领域的专家根据领域自身的特点和需要,提出若干场景来指导评估小组评估软件系统特定方面的性能。对于同一系统,如果提出的场景不同,评估的结果也将不同。

思 考 题

1. 什么是软件体系结构评估?软件体系结构评估对软件系统有什么作用?

2. 软件体系结构的评估方法有哪些?简要说明其各自的优势和缺点。

3. ATAM和SAAM作为软件体系结构评估的两种主要方法,请比较分析这两种方法的异同及各自的优缺点。

4. 某公司欲为某电子商务企业开发一个在线交易平台,支持客户完成网上购物活动中的在线交易。在系统开发之初,企业对该平台提出了如下要求。

(1) 在线交易平台必须在1s内完成客户的交易请求。

(2) 该平台必须保证客户个人信息和交易信息的安全。

(3) 当发生故障时,该平台的平均故障恢复时间必须小于10s。

(4) 由于企业业务发展较快,需要经常为该平台添加新功能或进行硬件升级。添加新功能或进行硬件升级必须在6h内完成。

针对这些要求，该软件开发公司决定采用基于体系结构的软件开发方法，以体系结构为核心进行在线交易平台的设计与实现。

(1) 软件质量属性是影响软件体系结构设计的重要因素。请列举 6 种不同的软件质量属性名称，并解释其含义。

(2) 请对该在线交易平台的 4 个要求进行分析，指出每个要求对应何种软件质量属性；并针对每种软件质量属性，各给出两种实现该质量属性的体系结构设计策略。

参考文献

第 12 章 基于体系结构的软件开发

模式是给定上下文中普遍问题的普遍解决方案，在软件开发方面，分为高层的模式与低层的模式，具体包括体系结构模式、设计模式(design pattern)、惯用法等。

体系结构模式一般简称为架构模式，是指软件体系结构的风格(见第 3 章)，这是软件设计中的高层决策。例如，C/S 结构就属于体系结构模式，体系结构模式反映了开发软件系统过程中所做的基本设计决策。

设计模式主要关注软件系统的设计，与具体的实现语言无关。设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结，使用设计模式的目的是提高代码的可重用性，让代码更容易被他人理解，并保证代码可靠性。毫无疑问，这些设计模式已经在前人的系统中得以证实并广泛使用，它使代码编写真正实现工程化，将已证实的技术表述成设计模式，也会使新系统开发者更加容易理解其设计思路。

惯用法是实现时通过某种特定的程序设计语言描述构件与构件之间的关系，例如，引用-计数就是 C++语言中的一种惯用法。

12.1 设 计 模 式

设计模式是前人经验的总结，它使人们可以方便地复用成功的设计和体系结构。当人们在特定的环境下遇到特定类型的问题，采用他人已使用过的一些成功的解决方案，一方面可以降低分析、设计和实现的难度，另一方面可以使系统具有更好的可复用性和灵活性。

目前，设计模式已广泛应用于面向对象系统的设计和开发，成为面向对象领域的一个重要组成部分。随着面向对象技术的发展和广泛应用，设计模式不再是一个新兴名词，它已逐步成为软件工程师所需掌握的基本技能之一。

12.1.1 设计模式概述

随着面向对象技术的出现和广泛使用，一方面软件的可重用性在一定程度上已经有所解决，另一方面对软件可重用性的要求也越来越高。设计面向对象的软件很难，而设计可重复使用的面向对象的软件难度更大。开发人员必须找到适当的对象，将它们分解到粒度合适的类、定义类接口和继承体系，并建立它们之间的关键联系。

在某个时候，设计师的设计可能是针对当前的具体问题而进行的，但它应该可能通用到足以适应未来的问题和需求。因为他们总是希望避免重复设计，至少将之减少到最低水平。在一个设计完成之前，有经验的面向对象的设计师往往要重复使用若干次，而且每次都要进行改进。他们知道，不能只用最初的方法解决每个问题，要常常重复使用那些过去用过的解

决方案。当他们找到一个好的解决方案时，总是一次又一次地使用它。这些经验也正是他们成为专家的法宝，这就是设计经验的价值。

因此，可将设计面向对象软件的经验记录成设计模式。每个设计模式都有系统的命名、解释和评价了面向对象系统中一个重要的设计。设计师的目标是将设计经验收集成人们可以有效利用的模型。为此，可以记录一些最重要的设计模式，并以目录形式表现出来。

利用设计模式可方便地重用成功的设计和结构。把已经证实的技术表示为设计模式，使它们更加容易被新系统的开发者所接受。设计模式帮助设计师选择可使系统重用的设计方案，避免选择危害到可重用性的方案。设计模式还提供了类和对象接口的明确的说明书和这些接口的潜在意义，来改进现有系统的记录和维护。

设计模式的概念最早是由一位美国的建筑理论家 Christopher Alexander 提出的，他试图找到一种结构化、可重用的方法，以在图纸上捕捉到建筑物的基本要素。他把注意力放在建筑物和城镇的设计和结构上，逐渐地他的思想影响了软件研究，并在最近流行起来。Alexander 提出的模式是指经过时间考验的解决方案，使用模式可以降低解决问题的复杂度。在编程时，很多情况下代码都不是从头编写，而是经过模仿得来，即从别处搬过来，再经过一定改造使之适应当前情况。设计模式可以视为这种模仿的一种抽象，包含一组规则，描述了如何在软件开发领域中完成一定的任务。从这个意义上讲，所有的算法都属于编程领域的设计模式。面向对象的设计模式解决如何在面向对象软件开发中完成一定的任务。

在介绍设计模式的具体定义之前，先看一个例子。在开发人机界面软件时考虑“模型-视图-控制器”(Model-View-Controler，MVC)这种模式。用户界面承担着向用户显示问题模型、与用户进行操作、输入/输出交互的作用。用户希望保持交互操作界面的相对稳定，但更希望根据需要改变和调整显示的内容和形式。例如，要求支持不同的界面标准或得到不同的显示效果，适应不同的操作需求。这就要求界面结构能够在不改变软件功能的情况下，支持用户对界面结构的调整。要做到这一点，从界面构成的角度看，困难在于：在满足对界面要求的同时，如何使软件的计算模型独立于界面的构成。MVC 就是这样的一种交互界面的结构组织模型。

对于界面设计可变性的需求，MVC 把交互系统的组成分解成模型、视图和控制 3 种构件。其中，模型构件独立于外在显示内容和形式，是软件所处理的问题逻辑的内在抽象，它封装了问题的核心数据、逻辑和功能的计算关系，独立于具体的界面表达和输入/输出操作；视图构件把表示模型数据及逻辑关系和状态的信息以特定形式展示给用户，它从模型获得显示信息，对于相同的信息可以有多个不同的显示形式或视图；控制构件处理用户与软件的交互操作，其职责是决定软件的控制流程，确保用户界面与模型间的对应联系，它接收用户的输入，将输入反馈给模型，进而实现对模型的计算控制，它是使模型和视图协调工作的部件。

模型、视图与控制器的分离，使得一个模型可以具有多个显示视图。如果用户通过某个视图的控制器改变了模型的数据，所有其他依赖于这些数据的视图都应反映出这些变化。因此，无论何时发生了何种数据变化，控制器都会将变化通知所有的视图，导致显示的更新。

如图 12-1 所示的对象模型技术类图描述了 MVC 解决方案。

从上面的例子中，可以导出软件体系结构模式的下列属性。

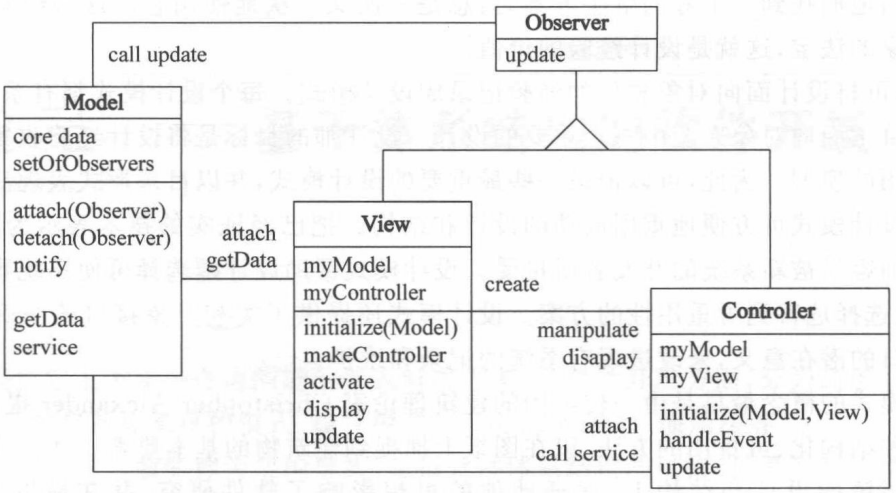


图 12-1 MVC 解决方案

一个模式关注一个在特定设计环境中出现的重现设计问题,并为它提供一个解决方案。在前面的例子中,问题是支持用户界面的可变性。开发人机交互软件系统时,这个问题就会出现。

所谓设计模式,简单地理解,是一些设计面向对象的软件开发的经验总结。一个设计模式事实上是系统地命名、解释和评价某一个重要的、可重现的面向对象的设计方案。正如 Alexander 所说的"每一个模式描述了一个在我们身边一再发生的问题,它告诉我们这个问题的解的关键,以使你可以成千上万次地利用这个解,而不需要再一次去解它。"尽管 Alexander 所说的是有关建筑和城镇的模式,它同样适用于面向对象的设计模式,只不过要解决的问题是软件开发中一再出现的问题。

受到普遍认可的设计模式的定义是由 Dirk Riehle 和 Heinz Zullighoven 于 1996 年在其 *Understanding and Using Patterns in Software Development* 一文中给出的:"模式是指从某个具体的形式中得到的一种抽象,在特殊的非任意性的环境中,该形式不断地重复出现"。

模式的概念是"随设计中要解决的问题的变化而变化的"。更明确地说,重复发生的具体形式就是这一重复出现的问题的解。但是一个模式又并不仅仅是它的解。问题是在一个特殊的环境中发生的,因此有很多复杂的考虑因素。给定一个环境,所提出的问题包含一些平衡各方面考虑的结构,或称为"权衡"。使用模式的形式,解决方案的描述可以把握住方案所体现的本质,故而别人可以从中学到一些东西,进而在相似的情况下可以进行应用。每个模式都有一个名字,帮助人们讨论模式和它所给出的信息。

关于设计模式,目前的研究方向主要有:设计模式与其他面向对象设计方法(如特定领域的框架)的关系,它们各自的优劣和适应范围。除此以外,人们还在各个方面总结设计模式(如通信等领域),以及研究如何让设计模式的使用更加自动化等。关于设计模式还有很多工作要做,而且它并不是一个只要看一遍就能掌握的方法。在工作中要不断总结、不断回顾以前使用过的设计模式。不同的模式之间有联系,同时也有各自的优缺点,在应用中要注意仔细考虑、权衡利弊、加以取舍。只有这样才可能真正用好设计模式。

12.1.2 设计模式的组成

设计模式描述了一个出现在特定设计语境中的特殊的再现设计问题，并为它的解决方案提供了一个经过充分验证的通用图示。解决方案图示通过描述其组成构件及其责任和相互关系以及它们的协作方式来具体指定。

一个好的设计模式必须做到以下几点。

(1) 解决一个问题：从模式可以得到解，而不仅仅是抽象的原则或策略。

(2) 是一个被证明了的概念：模式通过一个记录得到解，而不是通过理论或推测。

(3) 解并不是显然的：许多解决问题的方法(如软件设计范例或方法)是从最基本的原理得到解；而最好的方法是以非直接的方式得到解，对大多数比较困难的设计问题来说，这是必要的。

(4) 描述了一种关系：模式并不仅描述模块，它给出更深层的系统结构和机理。

(5) 模式有重要的人为因素：所有的软件服务于人类的需求，而最好的模式追求它的实用性和美学。

一般地说，一个模式有模式名称、问题、解决方案和效果4个基本成分，下面对这4个部分分别进行介绍。

1. 模式名称

模式名称通常用来描述一个设计问题、它的解法和效果，由一到两个词组成。模式名称的产生使人们可以在更高的抽象层次上进行设计并交流设计思想。因此寻找好的模式名称是一个很重要也是很困难的工作。

2. 问题

问题告诉人们什么时候要使用设计模式、解释问题及其背景。例如，MVC模式关心用户界面经常变化的问题。它可能描述诸如如何将一个算法表示成一个对象这样的特殊设计问题。在应用这个模式之前，也许还要给出一些该模式的适用条件。

模式的问题陈述用一个强制条件(force)集表示。该词最初是从建筑学和Christopher Alexander借用来的，设计模式使用术语“强制条件”来说明问题要解决时应该考虑的各个方面。

(1) 解决方案必须满足的需求。例如，对等进程间通信必须是高效的。

(2) 必须考虑的约束。例如，进程间通信必须遵循特定的协议。

(3) 解决方案必须具有期望的特性。例如，软件更改应该是容易的。

MVC模式指出了两个强制条件：它必须易于修改用户界面，但软件的功能核心不能被修改所影响。一般地，强制条件从多个角度讨论问题并有助于设计师了解它的细节。强制条件可以相互补充或相互矛盾。例如，系统的可扩展性与代码的最小化构成了两个相互矛盾的强制条件。如果希望系统可扩展，那么就应倾向于使用抽象超类。如果想使代码最小化(用于嵌入式系统)，就不能承受抽象超类的奢侈。但更重要的是，强制条件是解决问题的关键。它们平衡得越好，对问题的解决方案就越好。所以，强制条件的详细讨论是问题陈述的重要部分。

3. 解决方案

解决方案描述设计的基本要素是：它们的关系，各自的任务以及相互之间的合作。解

决方案并不是针对某一个特殊问题而给出的。设计模式提供有关设计问题的一个抽象描述以及如何安排这些基本要素以解决问题。一个模式就像一个可以在许多不同环境下使用的模板,抽象的描述使人们可以把该模式应用于解决许多不同的问题。

模式的解决方案部分给出了如何解决再现问题,或者更恰当地说是如何平衡与之相关的强制条件。在软件体系结构中,这样的解决方案包括两个方面。

(1) 每个模式规定了一个特定的结构,即元素的一个空间配置。例如,MVC 模式的描述包括以下语句:"把一个交互应用程序划分成 3 部分:处理、输入和输出"。

(2) 每个模式规定了运行期间的行为。例如,MVC 模式的解决方案部分包括以下陈述:"控制器接收输入,而输入往往是鼠标移动、单击鼠标按键或键盘输入等事件。事件转换成服务请求,这些请求再发送给模型或视图"。

注意:解决方案不必解决与问题相关的所有强制条件,可以集中于特殊的强制条件,对于其余的强制条件进行部分解决或完全不解决,特别是在强制条件相互矛盾时。

4. 效果

效果描述应用设计模式后的结果和权衡。比较与其他设计方法的异同,得到应用设计模式的代价和优点。对于软件设计来说,通常要考虑的是空间和时间的权衡。也会涉及语言问题和实现问题。对于一个面向对象的设计而言,可重用性很重要,效果还包括对系统灵活性、可扩充性及可移植性的影响。可以明确看出这些效果有助于理解和评价设计模式。

另外,不同的观点会影响人们对设计模式的解释。某一个人的模式对另一个人来说可能只是一个基本的构造块。这里把设计模式处理到一定的抽象程度,它不用于直接编码或类重用,也不是复杂到可作为一个完整的应用或子系统的领域专用的设计,而是对一定的对象与类的关系进行描述,进而可对其进行一定程度的修改使之可解决在一定条件下的通用设计问题。

设计模式命名、抽象并确定了一个普遍的设计结构的关键方面。这些方面有助于得到可重用的面向对象的设计。设计模式确定了参与的类和实例、它们的地位和协作,以及责任的分配。每一个设计模式都集中于特定的面向对象设计问题,描述了何时使用、是否能在其他设计约束条件下使用及使用后的结果和折中。

12.1.3 设计模式的描述

如果要理解和讨论模式,就必须以适当形式描述模式。好的描述有助于人们立即抓住模式的本质,即模式关心的问题是什么,以及提出的解决方案是什么?

模式也应该以统一方式来描述。这有助于人们对模式进行比较,尤其在设计师为一个问题寻求可选择的解决方案时。那么,如何描述一个设计模式呢?仅依靠图示的方法是不够的。尽管图示的方法很重要也很有用,但它们只能把设计的最终结果表示成一些类和对象的关系。事实上,为了重用该设计,还应该记录下产生这个设计的决策和权衡过程。具体的实例也很重要,从中可以看到设计模式的运转过程。模式概念的创始者 Alexander 采用如下所示的格式来描述设计模式。

```
IF you find yourself in CONTEXT
        For example EXAMPLES,
        With PROBLEM,
```

```
        Entailing FORCESS
THEN for some REASONS,
        Apply DESIGN FORM AND/OR RULE
        To construct SOLUTION
    Leading to NEW CONTEXT and OTHER PATTERNS
```

Erich Gamma 博士等采用下面的固定模式来描述，这也是目前最常用的格式。

(1) 模式名称和分类：模式名称和一个简短的摘要。

(2) 目的：回答下面的问题，即本设计模式的用处、它的基本原理和目的、它针对的是什么特殊的设计问题。

(3) 别名：由于设计模式的提取是由许多专家得到的，同一个模式可能会被不同的专家冠以不同的名称。

(4) 动机：描述一个设计问题的方案，以及模式中类和对象的结构是如何解决这个问题的。

(5) 应用：在什么情况下可以应用本设计模式，如何辨认这些情况。

(6) 结构：用对象模型技术对本模式的图像表示。另外，也给出了对象间相互的要求和合作的内在交互图。

(7) 成分：组成本设计模式的类和对象及它们的职责。

(8) 合作：成分间如何合作实现它们的任务。

(9) 效果：该模式如何支持它的对象；如何在使用本模式时进行权衡，即其结果如何；可以独立地改变系统结构的哪些方面。

(10) 实现：在实现本模式的过程中，要注意哪些缺陷、线索或者技术；是否与编程语言有关。

(11) 例程代码：说明如何用 C++或其他语言来实现该模式的代码段。

(12) 已知的应用：现实系统中使用该模式的实例。

(13) 相关模式：与本模式相关的其他模式，它们之间的区别，以及本模式是否要和其他模式共同使用。

在特定的软件开发领域中，可以采用不同的描述方法。在描述过程中，可以忽略这 13 个要素中的某些要素(例如，可以忽略“别名”)，或者可以将几个要素合并为一个要素(例如，可以把“应用”和“已知的应用”合并)。

12.1.4 模式和软件体系结构

判断模式取得成功的一个重要准则是它们在多大程度上达到了软件工程的目标。模式必须支持复杂的、大规模系统的开发、维护以及演化。它们也必须支持有效的产业化的软件生产，否则它们就只是停留在有趣的智能概念上，而对于构造软件没有什么用途。

1. 模式作为体系结构构造块

在开发软件时，模式是处理受限的特定设计方面的有用构造块。因此，对软件体系结构而言，模式的一个重要目标就是用已定义属性进行特定的软件体系结构的构造。例如，MVC 模式提供了一个结构，用于交互应用程序的用户界面的裁剪。

软件体系结构的一般技术，例如，使用面向对象特征(如继承和多态性)的指南，并没有

针对特定问题的解决方案。绝大多数现有的分析和设计方法在这一层次也是失败的。它们仅提供构建软件的一般技术,特定体系结构的创建仍然基于直觉和经验。

模式使用特定的面向问题的技术来有效补充这些通用的与问题无关的体系结构技术。注意,模式不会舍弃软件体系结构的现有解决方案,相反,它们填补了一个没有被现有技术覆盖的缺口。

2. 构造异构体系结构

单个模式不能完成一个完整的软件体系结构的详细构造,它仅帮助设计师设计应用程序的某一方面。然而,即使正确设计了这个方面,整个体系结构仍然可能达不到期望的所有属性。为"整体上"达到软件体系结构的需求,需要一套丰富的涵盖许多不同设计问题的模式。可获得的模式越多,能够适当解决的设计问题也会越多,并且可以更有力地支持构造带有已定义属性的软件体系结构。

为了有效使用模式,需要将它们组织成模式系统(pattern system)。模式系统统一描述模式,对它们进行分类,更重要的是,说明它们之间如何交互。模式系统也有助于设计师找到正确的模式来解决一个问题或确认一个可选解决方案。这和模式目录(pattern catalog)相反,在模式目录中每个模式描述的多少与别的模式无关。

3. 模式和方法

好的模式描述也包含它的实现指南,可将其看成是一种微方法(micro-method),可用来创建解决一个特定问题的方案。通过提供方法的步骤解决软件开发中的具体再现问题,这些微方法补充了通用的但与问题无关的分析和设计方法。

4. 实现模式

从模式与软件体系结构的集成中产生的另一个方面是用来实现这些模式的一个范例。目前的许多软件模式具有独特的面向对象风格。所以,人们往往认为,能够有效实现模式的唯一方式是使用面向对象编程语言,其实不然。

一方面,许多模式确实使用了诸如多态性和继承性等面向对象技术。策略(strategy)模式和代理(proxy)模式是这种模式的例子。另一方面,面向对象特征对实现这些模式并不是最重要的。例如,代理模式通过放弃继承性而失去了一小部分简洁性。在C语言中实现策略模式可以通过采用函数指针代替多态性和继承性。

在设计层次,大多数模式只需要适当的编程语言的抽象机制,如模块或数据抽象。因此,可以用几乎所有的编程范例并在几乎所有的编程语言中来实现模式。另外,每种编程语言都有它自己特定的模式,即语言的惯用法。这些惯用法捕获了现有的有关该语言的编程经验并为它定义了一个编程风格。

总之,可以说,没有单个的范例或语言可以用来实现模式。模式可以与构造软件体系结构用到的每一个范例进行集成。

12.1.5 设计模式的层次

从总体上来看,设计模式也只是一个概念,在不同层次上有不同的表现和特征。典型的有面向对象模式、代码模式、框架应用模式、形式合约等。

1. Coad的面向对象模式

1992年,美国的面向对象技术大师Peter Coad从MVC的角度对面向对象系统进行了

讨论，设计模式由最底层的构成部分(类和对象)及其关系来区分。他使用了一种通用的方式来描述一种设计模式。

(1) 模式所能解决问题的简要介绍与讨论。

(2) 模式的非形式文本描述以及图形表示。

(3) 模式的使用方针：在何时使用以及能够与哪些模式结合使用。

可以将 Coad 的模式划分为以下 3 类。

(1) 基本的继承和交互模式：主要包括面向对象程序设计语言所提供的基本建模功能，继承模式声明了一个类能够在其子类中被修改或被补充，交互模式描述了在有多个类的情况下消息的传递。

(2) 面向对象软件系统的结构化模式：描述了在适当情况下，一组类如何支持面向对象软件系统结构的建模。主要包括条目(item)描述模式、为角色变动服务的设计模式和处理对象集合的模式。

(3) 与 MVC 框架相关的模式。

几乎所有 Coad 提出的模式都指明如何构造面向对象软件系统，有助于设计单个的或者一小组构件，描述了 MVC 框架的各个方面。但是，他没有重视抽象类和框架，没有说明如何改造框架。

2. 代码模式

代码(coding)模式的抽象方式与面向对象程序设计语言中的代码规范很相似，该类模式有助于解决某种面向对象程序设计语言中的特定问题。代码模式的主要目标如下。

(1) 指明结合基本语言概念的可用方式。

(2) 构成源码结构与命名规范的基础。

(3) 避免面向对象程序设计语言(尤其是 C++语言)的缺陷。

代码模式与具体的程序设计语言或者类库有关，它们主要从语法的角度对于软件系统的结构方面提供一些基本的规范。这些模式对于类的设计不适用，同时也不支持程序员开发和应用框架，命名规范是类库中的名字标准化的基本方法，以免在使用类库时产生混淆。

3. 框架应用模式

在应用程序框架“菜谱”(application framework cookbook recipes)中有很多“菜谱条”，它们用一种不很规范的方式描述了如何应用框架来解决特定的问题。程序员将框架作为应用程序开发的基础，特定的框架适用于特定的需求。“菜谱条”通常并不讲解框架的内部设计实现，只讲如何使用。

不同的框架有各自的“菜谱”，例如 Glenn E. Krasner 和 Stephen T. Pope 提出了如何使用 MVC 框架的“菜谱”，苹果公司在 1989 年提出的“菜谱”说明如何利用 MacApp 的 GUI 应用程序框架在 Macintosh 机器上开发应用系统，还有其他一些学者提出了建立图形编辑器框架的“菜谱”等。

实践证明，“菜谱”的概念非常适合于框架的应用，它覆盖了大部分典型的框架应用，但是这些“菜谱”基本上都是不完全的。在“菜谱”中说明的应用情况越多，就越不容易找到相应的“菜谱条”，并且有的应用可以用数种方案来解决，或者要用数种方案的结合来解决，这种交错结构的不清晰性使程序员很容易糊涂。为了避免这样的问题，“菜谱”应该由那些对框架本身有相当深入的理解的人来撰写，最理想的情况是由框架的开发者来撰写。

超文本系统能够很好地支持这种“菜谱”方法,更高级的超文本系统(如ET++)已经超出了简单的应用“菜谱”的范畴,它们还可以基于设计模式方法(如设计模式目录、元模式等)对框架的设计做文档。

4. 形式合约

形式合约(formal contracts)也是一种描述框架设计的方法,强调组成框架的对象间的交互关系。有人认为它是面向交互的设计,对其他方法的发展有启迪作用。但形式化方法由于过于抽象,而有很大的局限性,仅在小规模程序中使用。

Richard Helm等是形式合约模式的倡导者,他们最先在面向对象系统领域内探索用抽象的方法来描述被他们称为行为合成(behavioral composition)的内容。他们所使用的规范符号有如下优点。

(1) 符号所包含的元素很少,并且其中引入的概念能够被映射成为面向对象程序设计语言中的概念。例如,参与者映射成为对象。

(2) 形式合约中考虑到了复杂行为是由简单行为组成的事实,合约的修订和扩充操作使得这种方法很灵活,易于应用。

形式合约模式的缺点有以下3点。

(1) 在某些情况下很难用,过于烦琐。若引入新的符号,则又使符号系统复杂化。

(2) 强制性地要求过分精密,从而在说明中可能发生隐患(如冗余)。

(3) 形式合约的抽象程度过低,接近面向对象的程序设计语言,不易分清主次。

12.1.6 设计模式的分类

根据目的和用途不同,设计模式可分为创建型(creational)模式、结构型(structural)模式和行为型(behavioral)模式3种。创建型模式主要用于创建对象,结构型模式主要用于处理类或对象的组合,行为型模式主要用于描述类或对象的交互以及职责的分配。

根据处理范围不同,设计模式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,在编译时刻就被确定下来,属于静态关系;对象模式处理对象之间的关系,这些关系在运行时刻变化,更具动态性。

1. 创建型模式

创建型模式对类的实例化过程(即对象的创建过程)进行了抽象,能够使软件模块做到与对象的创建和组织无关。创建型模式隐藏了对象是如何被创建的和组合在一起的,以达到使整个系统独立的目的。创建型模式包括工厂方法模式、抽象工厂模式、原型模式、单例模式、建造者模式等。

(1) 工厂方法(factory method)模式。工厂方法模式又称虚拟构造器(virtual constructor)模式或多态模式,属于类的创建型模式。在工厂方法模式中,父类负责定义创建对象的公共接口,子类则负责生成具体的对象,这样做的目的是将类的实例化操作延迟到子类中完成,即由子类决定究竟应该实例化(创建)哪一个类。

(2) 抽象工厂(abstract factory)模式。抽象工厂模式又称Kit模式,属于对象创建型模式。抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态,它提供了一个创建一系列相关或相互依赖对象的接口,而无须指定它们具体的类。在抽象工厂模式中,引入了产品等级结构和产品族的概念,产品等级结构是指抽象产品与具体产品所构成的

继承层次关系，产品族是同一个工厂所生产的一系列产品，即位于不同产品等级结构且功能相关联的产品组成的家族。当抽象工厂模式退化到只有一个产品等级结构时，即变成了工厂方法模式。

(3) 原型(prototype)模式。在系统开发过程中，有时候有些对象需要被频繁创建，原型模式通过给出一个原型对象来指明所要创建的对象的类型，然后通过复制这个原型对象的办法，创建出更多同类型的对象。原型模式是一种对象创建型模式，用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。原型模式可分为两种：浅克隆和深克隆。浅克隆仅复制所考虑的对象，而不复制它所引用的对象，也就是其中的成员对象并不复制；深克隆除了对象本身被复制外，对象包含的引用也被复制，即成员对象也被复制。

(4) 单例(singleton)模式。单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

(5) 建造者(builder)模式。建造者模式强调将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式是一步一步地创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。

2. 结构型模式

结构型模式描述如何将类或对象结合在一起形成更大的结构。结构型模式描述两种不同的事物，即类与类的实例(对象)，根据这一点，可以分为类结构型模式和对象结构型模式。结构型模式包括适配器模式、桥接模式、组合模式、装饰模式、外观模式、享元模式、代理模式等。

(1) 适配器(adapter)模式。适配器模式将一个接口转换成客户希望的另一个接口，从而使接口不兼容的那些类可以一起工作。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。在类适配器模式中，通过使用一个具体类将适配者适配到目标接口中；在对象适配器模式中，一个适配器可以将多个不同的适配者适配到同一个目标。

(2) 桥接(bridge)模式。桥接模式将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(handle and body)模式或接口(interface)模式。桥接模式类似于多重继承方案，但是多重继承方案往往违背了类的单一职责原则，其复用性比较差，桥接模式是比多重继承方案更好的解决方法。

(3) 组合(composite)模式。组合模式又称为整体-部分(part-whole)模式，属于对象的结构模式。在组合模式中，通过组合多个对象形成树状结构以表示整体-部分的结构层次。组合模式对单个对象(即叶子对象)和组合对象(即容器对象)的使用具有一致性。

(4) 装饰(decorator)模式。装饰模式是一种对象结构型模式，可动态地给一个对象增加一些额外的职责，就增加对象功能来说，装饰模式比生成子类实现更为灵活。通过装饰模式，可以在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责；当需要动态地给一个对象增加功能，这些功能可以再动态地被撤销时可使用装饰模式；当不能采用生成子类的方法进行扩充时也可使用装饰模式。

(5) 外观(facade)模式。外观模式是对象的结构模式，要求外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

(6) 享元(flyweight)模式。享元模式是一种对象结构型模式,通过运用共享技术,有效地支持大量细粒度的对象。系统只使用少量的对象,而这些对象都很相似,状态变化很小,对象使用次数增多。享元对象能做到共享的关键是区分内部状态和外部状态。内部状态是存储在享元对象内部并且不会随环境改变而改变,因此内部状态可以共享;外部状态是随环境改变而改变的、不可以共享的状态,享元对象的外部状态必须由客户端保存,并在享元对象被创建之后,在需要使用的时候再传入享元对象内部,外部状态之间是相互独立的。

(7) 代理(proxy)模式。代理模式是一种对象结构型模式,可为某个对象提供一个代理,并由代理对象控制对原对象的引用。代理模式能够协调调用者和被调用者,能够在一定程度上降低系统的耦合度。其缺点是请求的处理速度会变慢,并且实现代理模式需要额外的工作。

3. 行为型模式

行为型模式是对在不同的对象之间划分责任和算法的抽象化,它不仅是关于类和对象的,而且是关于它们之间的相互作用的。行为型模式分为类行为模式和对象行为模式两种,其中,类行为模式使用继承关系在几个类之间分配行为,而对象行为模式则使用对象的聚合来分配行为。行为型模式包括职责链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式、访问者模式等。

(1) 职责链(chain of responsibility)模式。职责链模式是一种对象的行为型模式,避免请求发送者与接收者耦合在一起,让多个对象都有可能接收请求,将这些对象连接成一条链,并且沿着这条链传递请求,直到有对象处理它为止。职责链模式不保证每个请求都被接收,由于一个请求没有明确的接收者,那么就不能保证它一定会被处理。

(2) 命令(command)模式。命令模式是一种对象的行为型模式,类似于传统程序设计方法中的回调机制,它将一个请求封装为一个对象,从而使得可用不同的请求对客户进行参数化;对请求排队或者记录请求日志,以及支持可撤销的操作。命令模式是对命令的封装,将发出命令的责任和执行命令的责任分割开,委派给不同的对象,以实现发送者和接收者完全解耦,提供更大的灵活性和可扩展性。

(3) 解释器(interpreter)模式。解释器模式属于类的行为型模式,描述了如何为语言定义一个文法,如何在该语言中表示一个句子,以及如何解释这些句子,这里的"语言"是使用规定格式和语法的代码。解释器模式主要用在编译器中,在应用系统开发中很少用到。

(4) 迭代器(iterator)模式。迭代器模式是一种对象的行为型模式,提供了一种方法来访问聚合对象,而不用暴露这个对象的内部表示。迭代器模式支持以不同的方式遍历一个聚合对象,复杂的聚合可用多种方法进行遍历;允许在同一个聚合上可以有多个遍历,每个迭代器保持它自己的遍历状态,因此,可以同时进行多个遍历操作。

(5) 中介者(mediator)模式。中介者模式是一种对象的行为型模式,通过一个中介对象来封装一系列的对象交互。中介者使得各对象不需要显式地相互引用,从而使其耦合松散,而且可以独立地改变它们之间的交互。中介者对象的存在保证了对象结构上的稳定,也就是说,系统的结构不会因为新对象的引入带来大量的修改工作。

(6) 备忘录(memento)模式。备忘录模式确保在不破坏封装的前提下,捕获一个对象的内部状态,并在该对象之外保存这个状态,这样可以在以后将对象恢复到原先保存的状态。备忘录模式提供了一种状态恢复的实现机制,使得用户可以方便地回到一个特定的历

史步骤。

(7) 观察者(observer)模式。观察者模式又称为发布-订阅模式、模型-视图模式、源-监听器模式或从属者(dependents)模式,是一种对象的行为型模式。它定义了对象之间的一种一对多的依赖关系,使得每当一个对象状态发生改变时,其相关依赖对象都得到通知并被自动更新。观察者模式的优点在于实现了表示层和数据层的分离,并定义了稳定的更新消息传递机制,类别清晰,抽象了更新接口,使得相同的数据层可以有各种不同的表示层。

(8) 状态(state)模式。状态模式是一种对象的行为型模式,允许一个对象在其内部状态改变时改变它的行为,对象看起来似乎修改了它的类。状态模式封装了状态的转换过程,但是它需要枚举可能的状态,因此,需要事先确定状态种类,这也导致在状态模式中增加新的状态类时将违反开闭原则,新的状态类的引入将需要修改与之能够进行转换的其他状态类的代码。状态模式的使用必然会增加系统类和对象的个数。

(9) 策略(strategy)模式。策略模式是一种对象的行为型模式,定义一系列算法,并将每一个算法封装起来,并让它们可以相互替换。策略模式让算法独立于使用它的客户而变化,其目的是将行为和环境分隔,当出现新的行为时,只需要实现新的策略类。

(10) 模板方法(template method)模式。模板方法模式是一种类的行为型模式,用于定义一个操作中算法的骨架,而将一些步骤延迟到子类中。模板方法模式使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤,其缺点是对于不同的实现,都需要定义一个子类,这会导致类的个数增加,但是更加符合类职责的分配原则,使得类的内聚性得以提高。

(11) 访问者(visitor)模式。访问者模式是一种对象的行为型模式,用于表示一个作用于某对象结构中的各元素的操作,它使得用户可以在不改变各元素的类的前提下定义作用于这些元素的新操作。访问者模式使得增加新的操作变得很容易,但在一定程度上破坏了封装性。

12.1.7 设计模式实例

本节以 Abstract Factory 模式为例,介绍设计模式的具体实现。其他设计模式的实现,请参考有关设计模式方面的专门书籍。

1. 模式名称

Abstract Factory,也经常称为抽象工厂模式。

2. 意图解决的问题

在程序中创建一个对象似乎是不能再简单的事情,其实不然。在大型系统开发中:

(1) "object new ClassName;"是最常见的创建对象的方法,但这种方法会造成类名的硬编码,需要根据不同的运行环境动态加载接口相同但实现不同的类实例,这样的创建方法就需要配合复杂的判断,实例化为不同的对象。

(2) 为了适用于不同的运行环境,经常使用抽象类定义接口,并在不同的运行环境中实现这个抽象类的子类。普通的创建方式必然造成代码与运行环境的强绑定,软件产品无法移植到其他的运行环境。

抽象工厂模式就可以解决这样的问题,根据不同的配置或上下文环境加载具有相同接口的不同类实例。

3. 模式描述

Abstract Factory 模式的结构如图 12-2 所示。

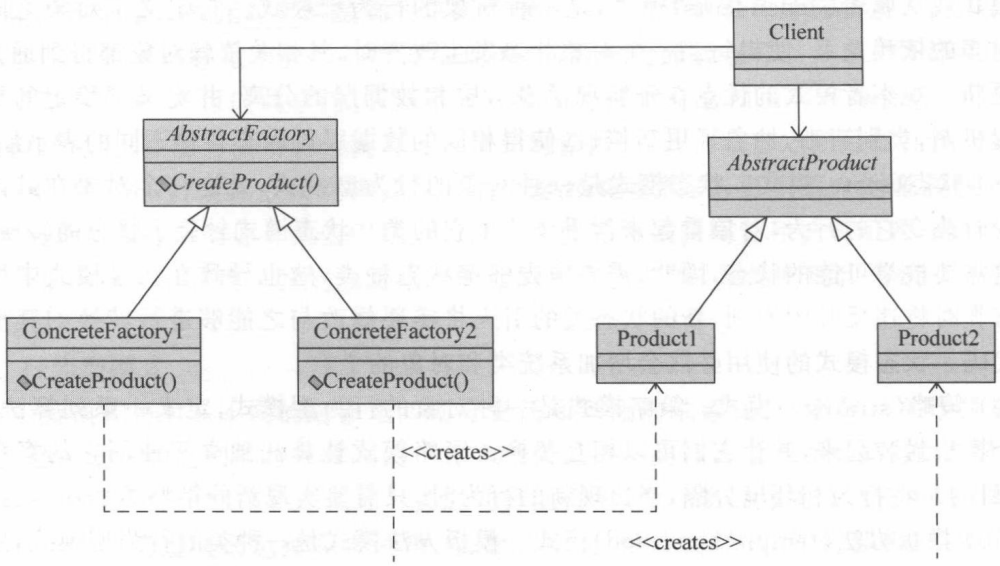


图 12-2　Abstract Factory 结构图

就如同抽象工厂的名字一样，AbstractFactory 类将接受 Client 的“订单”(Client 发送的消息)，使用不同的“车间”(不同的 ConcreteFactory)，根据已有的“产品模型”(AbstractProduct)，生产出特定的“产品”(Product)。不同的车间生产出不同的产品供客户使用，车间与产品的关系是一一对应的。由于所有的产品都遵循产品模型，具有相同的接口，所以，这些产品都可以直接交付客户使用。

在抽象工厂模式中，AbstractFactory 可以有多个类似于 CreateProduct()的虚方法，就如同一个工厂中有多条产品线一样。CreateProduct1()创建产品线 1，CreateProduct2()创建产品线 2。

注意：一般在 AbstractFactory 中的 CreateProduct()方法与 AbstractProduct 是一一对应的。而 ConcreteFactory 的数量与实际的 Product 的数量是一致的。

4. 效果

应用 Abstract Factory 模式可以实现对象可配置的、动态的创建。灵活运用 Abstract Factory 模式可以提高软件产品的移植性，尤其是当软件产品运行于多个平台，或有不同的功能配置版本时，Abstract Factory 模式可以减轻移植和发布时的压力，提高软件的复用性。

5. 实现

```
/* 抽象工厂的抽象类,由于使用 Java 语言,故定义为接口 */
public interface AbstractFactory {
        //创建产品的方法
        public AbstractProduct createProduct();
}
/* 抽象的产品接口 */
public interface AbstractProduct {
```

```
        ...
}
public class Product1 implements AbstractProduct {
        //实际的产品 1
}
public class Product2 implements AbstractProduct {
        //实际的产品 2
}

/* 在这个车间中,将创建实际产品 1 */
public class CFactory1 implements AbstractFactory {
        public AbstractProduct createProduct() {
            return new Product1();
        }
}

/* 在这个车间中,将创建实际产品 2 */
public class CFactory2 implements AbstractFactory {
        public AbstractProduct createProduct() {
            return new Product2();
        }
}
```

6. 相关讨论

在实际应用中,Abstract Factory 可以有更灵活的变化。事实上,如果仔细观察 Abstract Factory 模式就可以发现,对于 Client 来说,最关注的就是在不同条件下获得接口一致但实现不同的对象,只要避免类名的硬编码,采用其他方式也可以实现。所以,也可以采用其他的方式实现。例如,在 Java 中就可以采用接口的方式实现,如图 12-3 所示。

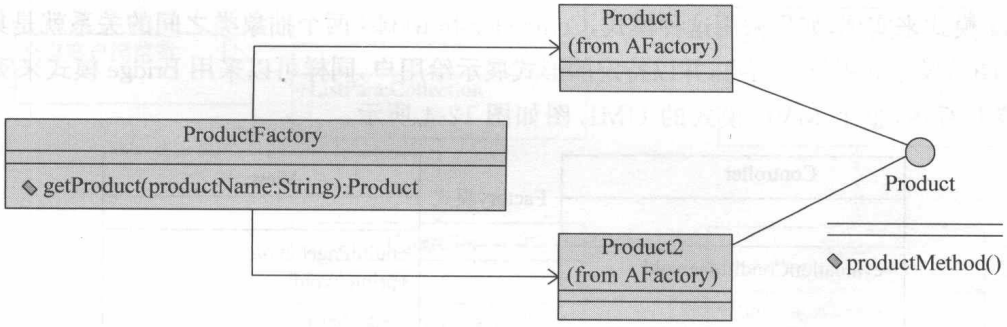


图 12-3　面向接口的简单工厂模式

图 12-3 中所描绘的类就应用了 Abstract Factory 的思想,采用面向接口的方式,简单实现了工厂模式。ProductFactory 既可以看作抽象工厂 AbstractFactory,也可以看作具体的工厂(车间)ConcreteFactory。其中提供了一个获得产品的方法:getProduct(productName:String),该方法将根据产品的名称创建特定的产品,并返回。所有的产品都实现了 Product 接口,可以在客户程序中加以使用。与抽象工厂模式一样,工厂中获得对象的方法(getProduct())与实际的产品线数量是一致的,如果要增加新的产品线,例如,定义新的产品接口 ProductX,需要增加相应的 getProduct()方法。由于这种方式把 AbstractFactory

和 ConcreteFactory 合并为一个 ProductFactory，所以增加新的实现 Product 接口的 Product*n* 不需要修改任何代码。

除了这种面向接口的方法外，工厂模式还可以有更多的变化。学习设计模式最重要的是学习设计思想，学习了工厂模式后，应该知道：

(1) 可配置的对象创建方法可以提高系统的移植性和复用性。

(2) 充分利用面向对象多态的特性可以避免对象创建过程的硬编码。

12.1.8 MVC 模式的设计与实现

有关 MVC 的基本概念，参见 12.1.1 节，在本节中，具体讨论 MVC 模式的设计与实现。由于运用 MVC 的应用程序的 3 个部件是相互独立的，改变其中任何一个都不会影响其他两个，所以依据这种设计思想能构造良好的松耦合的构件。

1. MVC 模式

MVC 模式可以包括 3 个抽象类，一个是 View 抽象类，它从模型获得显示信息，并以特定的形式展示给用户，对于相同的信息可以有多个不同的显示形式或视图；一个是 Controller 抽象类，处理用户与软件的交互操作，其职责是决定软件的控制流程，确保用户界面与模型间的对应联系，它接收用户的输入，将输入反馈给模型，进而实现对模型的计算控制，再根据用户的需求，创建合适的视图返回用户界面，它是使模型和视图协调工作的部件；另外一个是 Model 抽象类，它向视图(View 抽象类)和控制器(Controller 抽象类)提供业务逻辑服务。这 3 个抽象类之间的关系可以描述如下：控制器把接收到的请求或者数据传送到模型去处理，再根据用户的要求，创建一个合适的视图，该视图从模型中读取处理后的结果，以特定的形式显示出来。

由此可见，控制器用于创建视图，所以，可以采用 Factory 模式，而由于控制器将输入数据反馈给模型，进而实现对模型的计算控制，所以可以采用 Bridge 模式来实现(也可以采用 Adapter 模式来实现，如果采用这种模式，Controller 和 Model 两个抽象类之间的关系就是集合关系)，视图从模型获得显示信息，并以特定的形式展示给用户，同样可以采用 Bridge 模式来实现。

综上所述，整个 MVC 模式的 UML 图如图 12-4 所示。

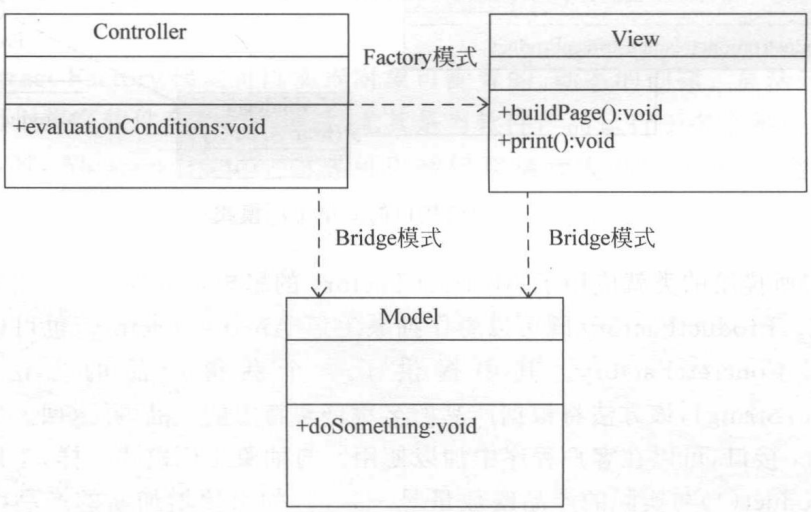


图 12-4　MVC 模式框架图

2. MVC 的设计与实现

在这一节中，以网络应用为例，讨论 MVC 模式的具体设计与实现。图 12-5 是一个简单的应用 MVC 模式的瘦客户网络应用程序的结构图。从图 12-5 中可以看出，整个系统的结构分为两个部分，一部分是客户端，另一部分是服务器。客户端不运行任何和应用程序相关的代码，它所需要的工具就是网络浏览器，通过网络浏览器，用户可以向服务器提出各种请求，包括显示网页、查询信息等请求，然后将服务器返回的请求结果再通过网络浏览器显示出来以满足自己的需求；而服务器和客户端不同，整个系统的代码都运行在服务器端，而 MVC 模式也是在服务器端实现的。

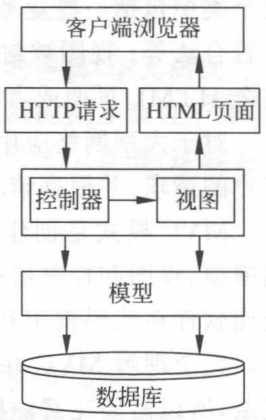


图 12-5　MVC 在网络中的应用

当客户端发送一个请求到服务器时，MVC 中的控制器负责接收这个请求，并将请求中包含的参数传送给模型，同时，调用模型中的服务来处理这个请求，并根据模型的处理结果选择一个合适的视图，由这个视图创建一个 HTML 页面返回给用户。在这里，如果有必要，视图可以调用模型服务来获取处理请求的结果数据，以便组建 HTML 页面供用户查看。例如，当控制器接收到一个用户登录请求时，它首先将用户名和密码传送给模型去处理，而模型通过查询数据库，检查传送来的用户名和密码是否合法，并返回成功与否的标志。控制器根据模型返回的成功与否的标志来选取不同的视图，如果失败，则选取登录失败视图，由该视图组建一个 HTML 页面，提示用户输入的用户名或者密码无效；如果成功，则选取登录成功视图，由该视图组建一个 HTML 页面，由于该页面需要显示该用户的某些个人信息，所以，该视图调用模型的服务来得到该用户的相关信息以完成页面的组建，并将其发送给用户。

为了实现图 12-5 所描述的 MVC 网络应用程序，用下面的 UML 类图来说明其具体的设计和实现过程，如图 12-6 所示。

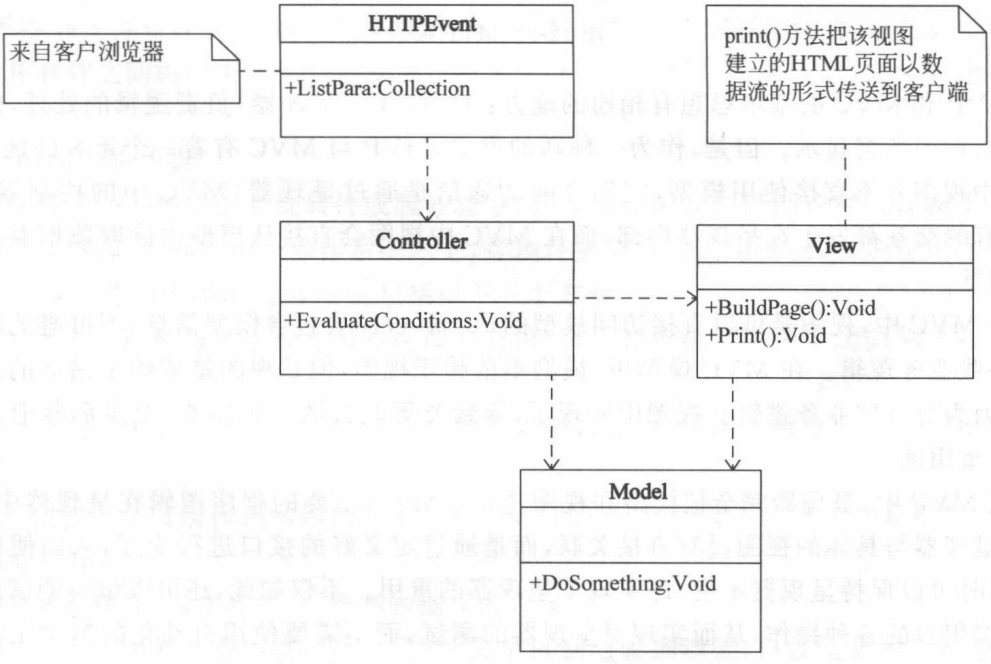


图 12-6　实现 MVC 的类图

把用户请求封装到一个 HttpEvent 类中,这个类中的数据成员包含请求参数,例如,用户名和密码等;控制器被封装在 Controller 类中,在这个类中,包括处理用户请求的成员函数,例如,判断用户的请求是否合法或者选取合适的视图等;模型被封装在 Model 类中,在这个类中包括一些业务处理逻辑的成员函数,例如,根据控制器传送的用户信息判断该用户是否合法等;视图被封装在 View 类中,这个类包含一些和显示相关的成员函数,例如,建立一个 HTML 页面或者将该页面传送到客户端等。

对于大型网络应用程序,还可以扩展 MVC 模式,引进一个引擎,由这个引擎接收来自客户的请求,并根据请求的类型选取一个合适的控制器,由该控制器去处理这个请求。

MVC 模式是创建软件的很好途径,它所提倡的一些原则,如内容和显示互相分离,隔离模型、视图和控制器的构件等,会使应用程序的体系结构更健壮,更具扩展性;同时,它也会使软件在代码重用和体系结构方面上一个新的台阶。然而,MVC 由于没有明确的定义,所以完全理解 MVC 并不是很容易。使用 MVC 需要精心的计划,由于它的内部原理比较复杂,所以需要花费一些时间去思考。

3. MVP 模式

MVP 模式是从经典的 MVC 演变而来,其逻辑结果如图 12-7 所示。

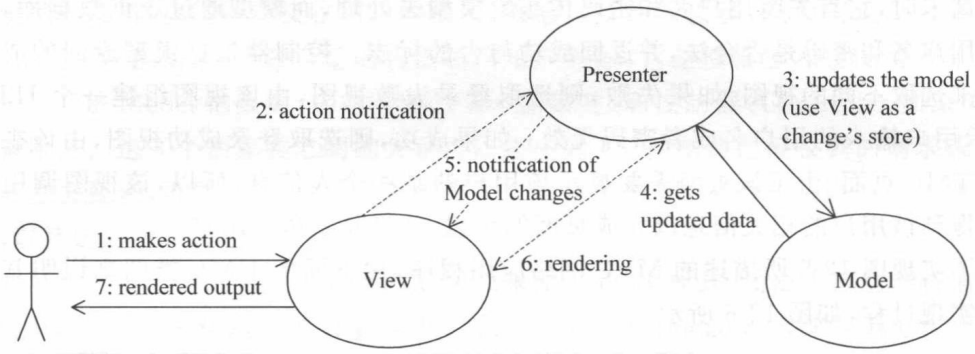


图 12-7 MVP 模式

MVP 和 MVC 的基本思想有相通的地方:Presenter(呈现器)负责逻辑的处理,模型提供数据,视图负责显示。但是,作为一种新的模式,MVP 与 MVC 有着一个重大的区别:在 MVP 中视图并不直接使用模型,它们之间的通信是通过呈现器(MVC 中的控制器)进行的,所有的交互都发生在呈现器内部,而在 MVC 中视图会直接从模型中读取数据而不是通过控制器。

在 MVC 中,视图是可以直接访问模型的,从而,视图会包含模型信息,不可避免地还要包括一些业务逻辑。在 MVC 模型里,模型不依赖于视图,但是视图是依赖于模型的。不仅如此,因为有一些业务逻辑在视图中实现了,导致要更改视图比较困难,至少那些业务逻辑是无法重用的。

在 MVP 中,呈现器完全把模型和视图进行了分离,主要的程序逻辑在呈现器中实现。而且,呈现器与具体的视图没有直接关联,而是通过定义好的接口进行交互,从而使得在变更视图时可以保持呈现器不变,即实现了呈现器的重用。不仅如此,还可以编写测试用的视图,模拟用户的各种操作,从而实现对呈现器的测试,而不需要使用自动化的测试工具。甚至可以在模型和视图都没有完成时,就可以通过编写接口来测试呈现器的逻辑。

综上所述，MVP 具有以下优点。

(1) 模型与视图完全分离，可以修改视图而不影响模型。

(2) 可以更高效地使用模型，因为所有的交互都发生在一个地方：呈现器内部。

(3) 可以将一个呈现器用于多个视图，而不需要改变呈现器的逻辑。这个特性非常有用，因为视图的变化总是比模型的变化频繁。

(4) 如果把逻辑放在呈现器中，那么就可以脱离用户接口测试这些逻辑(单元测试)。

然而，在 MVP 中，由于对视图的渲染放在呈现器中，视图和呈现器的交互会过于频繁。另外，如果呈现器过多地渲染了视图，往往会使得它与特定视图的联系过于紧密，这就会导致呈现器和视图的紧耦合。

12.2 中间件技术

我国企业从 20 世纪 80 年代开始就逐渐进行信息化建设，由于方法和体系的不成熟，以及企业业务和市场需求的不断变化，一个企业可能同时运行着多个不同的业务系统，这些信息系统可能基于不同的操作系统、不同的数据库、异构的网络环境。现在的问题是，如何把这些信息系统结合成一个有机的协同工作整体，真正实现企业跨平台、分布式应用。中间件(Middleware)便是解决之道，它用自己的复杂换取了企业应用的简单。

12.2.1 中间件概述

20 世纪 90 年代初，企业的应用系统开始采用分层体系结构。从最早的单机应用，发展到 C/S 体系结构，然后是三层 C/S 体系结构和基于浏览器的三层 B/S 体系结构，到现在的多层分布式系统结构。在这种多层分布式的系统结构中，服务器和客户机之间都是通过网络连接起来的，并有大量信息和数据进行传递。对每个应用系统而言，在设计和开发时不仅要关心业务逻辑，还必须处理分布环境中复杂的通信和异构系统问题，而目前的系统软件(操作系统和支撑软件)还不能满足多样的应用要求。为此，出现了中间件，它是处于系统软件和应用软件之间的一类软件。它使设计师集中设计与应用有关的部分，大大简化了设计和维护工作。

1. 中间件的功能

中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于操作系统之上，管理计算资源和网络通信，实现应用之间的互操作。具体来说，中间件的基本功能包括以下 6 个方面。

(1) 负责客户机与服务器之间的连接和通信，以及客户机与应用层之间的高效率通信机制。

(2) 提供应用层不同服务器之间的互操作机制，以及应用层与数据库之间的连接和控制机制。

(3) 提供一个多层体系结构的应用开发和运行的平台，以及一个应用开发框架，支持模块化的应用开发。

(4) 屏蔽硬件、操作系统、网络和数据库的差异。

(5) 提供应用的负载均衡和高可用性、安全机制与管理功能，以及交易管理机制，保证

交易的一致性。

(6) 提供一组通用的服务去执行不同的功能,避免重复的工作和使应用之间可以协作。

2. 中间件的分类

中间件的范围十分广泛,针对不同的应用需求涌现出了多种各具特色的中间件产品。因此,在不同的角度或不同的层次上,对中间件的分类也会有所不同。

采用自底向上的方式划分,可分为底层中间件、通用型中间件和集成型中间件3个大的层次。底层中间件的主流技术主要有JVM(Java Virtual Machine,Java虚拟机)、CLR(Common Language Runtime,公共语言运行库)、ACE(Adaptive Communication Environment,自适配通信环境)等;通用型中间件也称为平台,其主流技术主要有RPC、ORB、MOM(Message-Oriented Middleware,面向消息的中间件)等;集成型中间件的主流技术主要有WorkFlow、EAI等。

IDC(International Data Corporation,国际数据公司)在1998年对中间件进行了分类,把中间件分为终端仿真/屏幕转换中间件、数据访问中间件、远程过程调用中间件、消息中间件、交易中间件和对象中间件6大类。但是,如今所保留下来的只有消息中间件和交易中间件,其他的类型已经被逐步融合到其他产品中,在市场上已经没有单独的产品形态出现。

3. 中间件的应用

中间件提供了应用系统基本的运行环境,也为应用系统提供了更多的高级服务功能,如名字服务、事件服务、通告服务、日志服务等。在这些服务之上,还需要考虑不同行业的需求、不同的应用领域。中间件技术应用层次如图12-8所示。

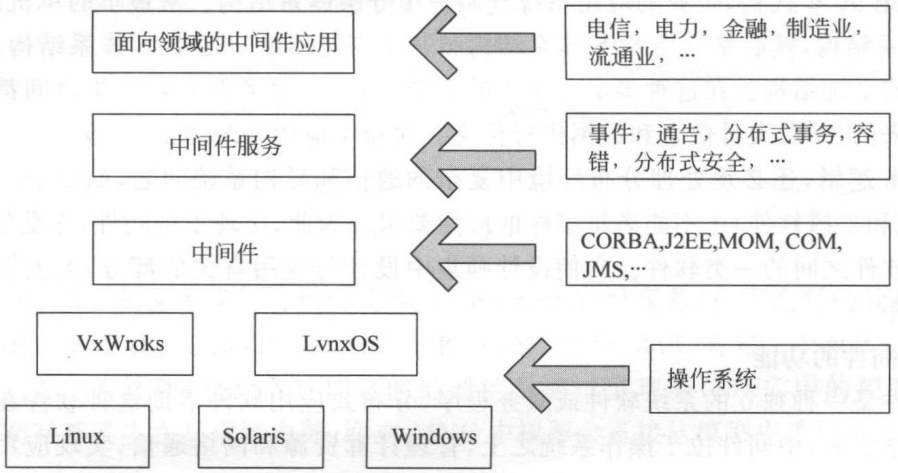


图12-8 中间件技术应用层次图

中间件技术在企业应用集成中扮演着重要的角色,可以从不同层次采用不同种类,不同技术的中间件产品进行应用集成。如图12-9所示,可以从传输、消息、构件、过程等各个层面分别加以集成。

从图12-9中还可以看出,为了完成不同层次的集成,可以采用不同的技术和产品。例如,为了完成系统底层传输层的集成,可以采用CORBA技术;为了完成不同系统的信息传递,可以采用消息中间件产品;为了完成不同硬件和操作系统的集成,可以采用J2EE(Java 2 Platform Enterprise Edition,Java2平台企业版)中间件产品。

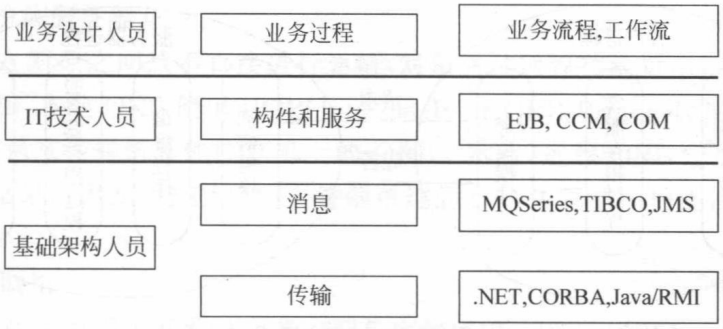


图 12-9　不同层次的集成示意图

4. 中间件的发展趋势

中间件作为构筑企业信息系统和电子商务系统的基石和核心技术，正在向着标准化和构件化方向发展。具体来看，有以下 3 种发展趋势。

(1) 规范化。对于不同类型的中间件，目前都有一些规范可以遵循，如消息类的 JMS，对象类的 CORBA 和 COM/DCOM，应用服务器类的 J2EE，数据访问类的 ODBC 和 JDBC 等。这些规范的建立极大地促进了中间件技术的发展，同时保证了系统的扩展性、开放性和互操作性。

(2) 构件化和松耦合。除了已经得到较为广泛应用的 CORBA、DCOM 等适应 Intranet 的构件技术外，随着企业业务流程整合和电子商务应用的发展，中间件技术朝着面向 Web 和松散耦合的方式发展。基于 XML 和 Web Service 的中间件技术，使得不同系统之间、不同应用之间的交互建立在非常灵活的基础上。

(3) 平台化。一些大的中间件厂商在已有的中间件产品基础上，提出了完整的面向互联网的软件平台战略计划和应用解决方案。

12.2.2　主要的中间件

中间件可为上层应用提供不同形式的通信服务，包括同步、排队、订阅/发布、广播等，在这些基本的通信平台上，可构建各种框架，为应用程序提供不同领域内的服务，如事务处理监控器、分布式数据访问、对象事务管理器等。中间件为上层应用屏蔽了异构平台的差异，而其上的框架又定义了相应领域内的应用体系结构、标准的服务构件等，用户只需告诉框架所关心的事件，然后提供处理这些事件的代码。当事件发生时，框架则会调用这些代码。用户程序不必关心框架结构、执行流程等，所有这些由框架负责完成。因此，基于中间件开发的应用具有良好的可扩充性、易管理性、高可用性和可移植性。下面，针对几类主要的中间件分别进行简要介绍。

1. 远程过程调用

RPC 是一种广泛使用的分布式应用程序处理方法。应用程序使用 RPC 远程执行一个位于不同地址空间的过程，并且从效果上看和执行本地调用相同。这里的“远程”既可以指不同的计算机，也可以指同一台计算机上的不同进程。一个 RPC 应用可分为服务器和客户两个部分。这里的“服务器”和“客户”指逻辑上的进程，而不是指物理计算机。支持 RPC 的软件如图 12-10 所示。

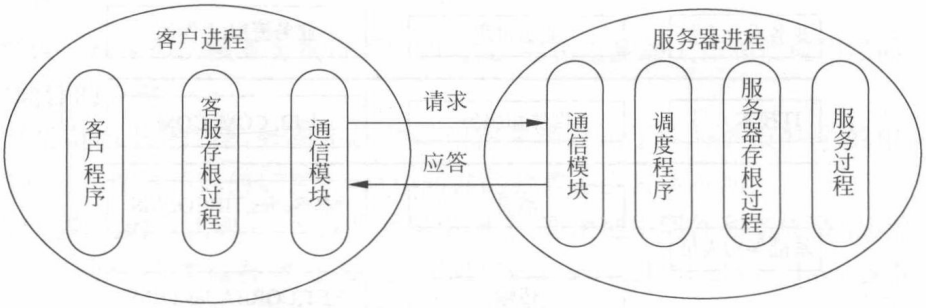


图 12-10　RPC 模型

在图 12-10 中,通信模块实现请求-应答协议,在客户和服务器之间传递请求和应答消息。服务器提供一个或多个远程过程,客户向服务器发出远程调用。服务器和客户可以位于同一台计算机,也可以位于不同的计算机,甚至运行在不同的操作系统之上。它们通过网络进行通信,相应的存根(stub)过程和运行支持提供数据转换和通信服务,从而屏蔽不同的操作系统和网络协议。存根过程用来解码请求消息中的参数、调用相应的服务过程和编码应答消息中的返回值。服务过程实现服务接口中的过程。调度程序根据请求消息中的过程标识选择一个服务器存根过程。客户存根过程、服务器存根过程和调度程序都可由接口编译器根据服务的接口定义而生成。

在这里,RPC 通信是同步的。如果采用线程的方式,则可以进行异步调用。在 RPC 模型中,客户和服务器只要具备了相应的 RPC 接口,并且具有 RPC 运行支持,就可以完成相应的互操作,而不必限制于特定的服务器。因此,RPC 为分布式计算提供了强有力的支持。同时,RPC 所提供的是基于过程的服务访问,客户与服务器进行直接连接,没有中间机构来处理请求,因此,也具有一定的局限性。例如,RPC 通常需要一些网络细节以定位服务器;在客户发出请求的同时,要求服务器必须是活动的。

2. 对象请求代理

随着对象技术与分布式计算技术的发展,两者相互结合形成了分布对象计算,并发展为当今软件技术的主流方向。1990 年年底,OMG 首次推出对象管理体系结构(Object Management Architecture,OMA)模型,ORB 是这个模型的核心组件。它的作用在于提供一个通信框架,透明地在异构的分布计算环境中传递对象请求。CORBA 规范包括 ORB 的所有标准接口。

ORB 是对象总线,它在 CORBA 规范中处于核心地位,定义异构环境下对象透明地发送请求和接收响应的基本机制,是建立对象之间 C/S 关系的中间件。ORB 使得对象可以透明地向其他对象发出请求或接受其他对象的响应,这些对象既可以位于本地,也可以位于远程机器。ORB 拦截请求调用,并负责找到可以实现请求的对象、传送参数、调用相应的方法、返回结果等。客户对象并不知道与服务器对象通信、激活或存储服务器对象的机制,也不必知道服务器对象位于何处、它是用何种语言实现的、使用什么操作系统或其他不属于对象接口的系统成分。

值得指出的是,客户和服务器角色只是用来协调对象之间的相互作用,根据相应的场合,ORB 上的对象可以是客户,也可以是服务器,甚至兼有两者的功能。当对象发出一个请求时,它是处于客户角色;当它在接收请求时,它就处于服务器角色。大部分的对象都是既

扮演客户角色又扮演服务器角色。

另外，CORBA 对象之间从不直接进行通信，对象通过远程存根对运行在本地计算机上的 ORB 发出请求。本地 ORB 使用 IIOP(Internet Inter-ORB Protocol，互联网内部对象请求代理协议)将该请求传递给其他计算机上的 ORB。然后，远程 ORB 定位相应的对象、处理该请求并返回结果。因此，与 RPC 所支持的单纯的 C/S 体系结构相比，ORB 可以支持更加复杂的结构。

3. 远程方法调用

RMI 是 Java 的一组用于开发分布式应用程序的 API。RMI 使用 Java 语言接口定义远程对象，它集合了 Java 序列化和 Java 远程方法协议。RMI 大大增强了 Java 开发分布式应用的能力，它可以被看作是 RPC 的 Java 版本。

与 RPC 一样，RMI 应用程序通常也包括两个独立的部分，分别是服务器和客户。服务器将创建多个远程对象，使这些远程对象能够被引用，然后等待客户调用这些远程对象的方法。而客户则从服务器中得到一个或多个远程对象的引用，然后调用远程对象的方法。RMI 系统如图 12-11 所示，对象 A 拥有远程对象 B 的远程对象引用并调用 B 的一个方法。

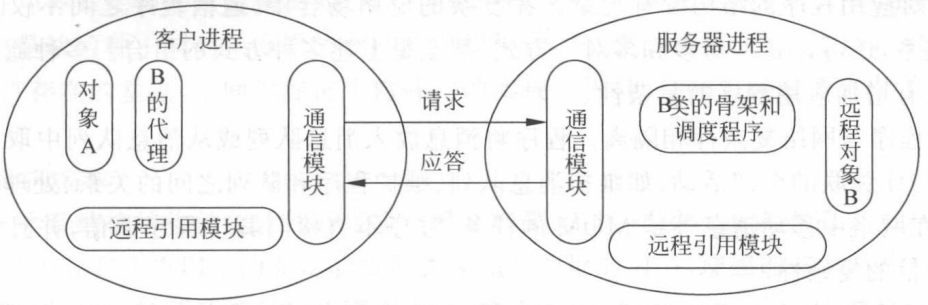


图 12-11　RMI 模型

在图 12-11 中，通信模块在客户和服务器之间传递请求和应答消息，远程引用模块负责翻译本地和远程对象引用，以及创建远程对象引用。代理类、调度程序类和骨架类都可由接口编译器自动创建。

代理的作用是通过在调用者面前表现得像本地对象一样，使远程方法调用对客户透明，它不执行调用，而是将调用放在消息中传递给远程对象。它隐藏了远程对象引用的细节、参数的编码、结果的解码，以及客户消息的发送和接收。对于具有远程对象引用的进程，其每个远程对象都有一个代理。

调度程序接收来自通信模块的请求消息，并传递请求消息，使用方法 ID 选择骨架中恰当的方法。调度程序和代理对远程接口中的方法使用相同的方法 ID。

骨架用于实现远程接口中的方法，骨架方法解码请求消息中的参数，并调用远程对象中的相应方法，然后等待调用的完成，将结果和任何异常信息编码进应答消息，发送给代理的方法。

RMI 目前使用 Java 远程消息交换协议(Java Remote Messaging Protocol，JRMP)进行通信。JRMP 是专为 Java 的远程对象制定的协议。因此，RMI 具有 Java 语言本身的一些优点，用 RMI 开发的应用系统可以部署在任何支持 Java 运行环境的平台上。但由于 JRMP 是专为 Java 对象制定的，RMI 对于用非 Java 语言开发的应用系统的支持不足，不能

与用非Java语言书写的对象进行通信。

RMI和CORBA常被视为相互竞争的技术,因为两者都提供对远程分布式对象的透明访问,但实际上这两种技术是相互补充的。RMI和CORBA的结合产生了RMI-IIOP,RMI-IIOP是企业服务器端Java开发的基础。

4. 面向消息的中间件

MOM指的是利用高效可靠的消息传递机制进行平台无关的数据交换,并基于数据通信进行分布式系统的集成。通过提供消息传递和消息排队模型,它可在分布式环境下扩展进程间的通信,并支持多种通信协议、语言、应用程序、硬件和软件平台。例如,IBM的MQSeries、BEA的MessageQ等都属于MOM产品。

在MOM中,消息传递和排队技术具有以下3个主要特点。

(1) 通信程序可在不同的时间运行。程序不在网络上直接相互通信,而是间接地将消息放入消息队列。因为程序间没有直接的联系,所以它们不必同时运行。消息放入适当的队列时,目标程序甚至根本不需要正在运行;即使目标程序在运行,也不意味着要立即处理该消息。

(2) 对应用程序的结构没有约束。在复杂的应用场合中,通信程序之间不仅可以是一对一的关系,还可以是一对多和多对一方式,甚至是上述多种方式的组合。多种通信方式的构造并没有增加应用程序的复杂性。

(3) 程序与网络复杂性相隔离。程序将消息放入消息队列或从消息队列中取出消息进行通信,与此关联的全部活动,如维护消息队列、维护程序和队列之间的关系、处理网络的重新启动、在网络中移动消息等是MOM的任务,程序不直接与其他程序通信,并且它们不涉及网络通信的复杂性。

MOM系统的基本元素是客户、消息和MOM提供者,后者包括API和管理工具。MOM提供者既可以使用集中式消息服务器,也可以将路由和传送功能分布在每个客户上,某些MOM产品结合了这两个方法。客户可以进行API调用,以便将消息发送到由提供者管理的目的地。该调用会调用提供者服务以路由和传送消息。在发送消息之后,客户会继续执行其他工作,并确信在接收方客户端检索该消息之前,提供者一直保留该消息。基于消息的模型与提供者的协调耦合在一起,使得创建松散耦合的构件系统成为可能。这样的系统可以继续可靠地工作,即使在有个别构件或连接失败时也不会停机。

5. 事务处理监控器

事务处理监控器(Transaction Processing Monitor,TPM)又称为交易中间件,是当前应用最广泛的中间件之一。它能支持数以万计的客户进程对服务器的并发访问,使系统具有极强的扩展性,因此,适于电信、金融、证券等拥有大量客户的领域。在对效率、可靠性要求严格的关键任务系统中具有明显优势。TPM一般支持负载均衡,支持分布式两阶段提交,保证事务完整性和数据完整性,并具有安全认证和故障恢复等功能,能很好地满足应用开发的要求。

TPM介于客户和服务器之间,进行事务管理与协调、负载平衡、失败恢复等,以提高系统的整体性能,它可以被看作是事务处理应用程序的"操作系统"。总体上来说,TPM具有以下功能。

(1) 进程管理,包括启动服务器进程、为其分配任务、监控其执行并对负载进行平衡。

(2) 事务管理,即保证在其监控下的事务处理的原子性、一致性、独立性和持久性。

(3) 通信管理,为客户和服务器之间提供多种通信机制,包括请求/响应、会话、排队、订阅/发布和广播等。

典型的事务处理中间件有 BEA 的 Tuxedo、IBM 的 CICS 等。TPM 能够为大量的客户提供服务。如果服务器为每一个客户都分配其所需要的资源,那服务器将不堪重负。实际上,在同一时刻并不是所有的客户都需要请求服务,而一旦某个客户请求了服务,它就希望得到快速的响应。TPM 在操作系统之上提供一组服务,对客户请求进行管理并为其分配相应的服务进程,使服务器在有限的系统资源下能够高效地为大规模的客户提供服务。

12.2.3 中间件与构件的关系

从本质上来说,中间件是对分布式应用的抽象,它抛开了与应用相关的业务逻辑细节,保留了典型的分布交互模式的关键特征。经过抽象,将纷繁复杂的分布式系统经过提炼和必要的隔离后,以统一的形式呈现给应用。应用在中间件提供的环境中可以更好地集中于业务逻辑,并以构件的形式存在,最终自然而然地在异构环境中实现良好的协同工作。

构件与体系结构是紧密相关的,抛开体系结构谈构件,其意义并不大。中间件与体系结构实际上是从两种不同的角度看待软件的中间层次。从某种程度上说,中间件就是体系结构,是构件存在的基础,中间件促进了构件化的实现。因此,中间件与体系结构在本质上是一致的。

1. 面向需求

基于体系结构的构件化软件开发应当是面向需求的,即设计师集中精力于业务逻辑本身,而不必为分布式应用中的非功能质量属性耗费大量的精力,理想的体系结构在这些方面应当为软件提供良好的运行环境。事实上,这些正是中间件所要解决的问题,因此,基于中间件开发的应用真正是面向需求的,从本质上符合构件化设计的思想。

2. 业务的分隔和包容性

服务器构件要求有很好的业务自包容性,应用开发人员可以按照不同的业务进行功能的划分,体现为不同的接口或交互模式。针对每种业务,设计和开发是可以独立进行的。在提供业务的分隔和包容性方面,体系结构和中间件有同样的目标。例如,消息中间件规定了消息是有属性的,其中部分属性与业务的划分有关。构件只进行相应类型的消息交互,至于如何保证业务的分类运行与管理,则是中间件的事情。

3. 设计与实现隔离

构件对外发生作用或构件间的交互,是通过接口进行的,构件使用者只需知道构件的接口,而不必关心其内部实现,这是设计与实现分离的关键。中间件在分布交互模式上也规定了接口(或类似)机制。例如,IDL 就是描述接口的语言规范,从早期的 DCE(Distributed Computing Environment,分布式计算环境)到现在的 CORBA、DCOM、RMI 等都使用 IDL 描述接口,所不同的只是语言规范。

4. 隔离复杂的系统资源

软件体系结构很重要的一个功能就是将系统资源与应用构件隔离,这是保证构件可复用甚至"即插即用"的基础,与中间件的意图一致。中间件最大的优势之一是屏蔽多样的系统资源,保证良好的互操作性。应用构件开发人员只需按照中间件规定的模式进行设计开

发,而不必考虑下层的系统平台。因此,中间件真正提供了与环境隔离的构件开发模式。

5. 符合标准的交互模型

软件体系结构是一种抽象的模型,但模型中应当定义一些可操作的成分,如标准的协议等。中间件则实现了体系结构的模型,实现了标准的协议。例如,基于CORBA的中间件使用的是CORBA规范作为体系结构模型,定义了数据表示语法、数据包格式、消息语义等内容,以实现互操作性协议。因此,基于中间件的构件是符合标准模型的。

6. 软件复用

软件复用是构件化软件开发的根本目标之一。中间件提供了构件封装、交互规则、与环境的隔离等机制,这些都为软件复用提供了方便的解决方案。另外,中间件可以建立访问过去的应用的通道,或者在新的中间件体系中建立特殊的运行容器,封装以往的应用,从而最终做到对遗留系统的继承性复用。

7. 提供对应用构件的管理

基于中间件的软件可以方便地进行管理,因为构件总可以通过标识机制进行划分。例如,COM就是利用Windows系统注册表配合几种唯一标识构件的方式,实现构件的登记、注销和定位。CORBA规范中有接口池、实现池等规范定义,配合应用登记管理的机制,也能对应用构件实施管理。

总之,基于中间件开发的应用是构件化的,中间件提供了构件的软件体系结构,大大提高了应用构件开发的效率和质量。中间件作为应用软件系统集成的关键技术,保证了构件化思想的实施,并为构件提供了真正的运行空间。反过来,构件对新一代中间件产品也起到了促进作用,构件化的中间件在市场上具有强大的生命力。

12.3 基于体系结构的设计方法

为软件系统设计一个体系结构不是一件容易的事。软件必须长期运行和具有自适应性,它必须支持广义的软件需求,详细的需求要等到最终产品开发完成后才能知道。而且,在做出最基本的设计决定时,就要进行初步的体系结构设计,这些决策一旦出现错误,将难以改正。为了有效地设计一个软件体系结构,软件设计师需要一个严格的设计方法。这些方法关注创造性的过程,为处理非确定软件需求提供策略;在设计过程中,为组织做出设计决策提供指导。

本节将介绍基于体系结构的软件设计(Architecture-Based Software Design,ABSD)方法。ABSD方法为产生软件系统的概念体系结构提供基础,概念体系结构是由Hofimeister、Nord和Soni提出的4种不同的体系结构中的一种,它描述了系统的主要设计元素及其关系。概念体系结构代表了在开发过程中做出的第一个选择,相应地,它是达到系统质量和业务目标的关键,为达到预定功能提供了一个基础。

ABSD方法取决于决定系统的体系结构驱动。所谓体系结构驱动,是指构成体系结构的业务、质量和功能需求的组合。使用ABSD方法,设计活动可以在体系结构驱动一决定就开始,这意味着需求获取和分析还没有完成(甚至远远没有完成),就开始了软件设计。设计活动的开始并不意味着需求获取和分析活动就可以终止,而是应该与设计活动并行。特别是在不可能预先决定所有需求时,如产品线系统(将在第13章讨论)或长期运行的系统,

快速开始设计是至关重要的。

ABSD 方法有 3 个基础。第一个基础是功能的分解。在功能分解中，ABSD 方法使用已有的基于模块的内聚和耦合技术。第二个基础是通过选择体系结构风格来实现质量和业务需求。第三个基础是软件模板的使用。软件模板利用了一些软件系统的结构。然而，对于设计方法来说，软件模板的使用是一个新概念，下面进行简单的介绍。

软件模板是一个特殊类型的软件元素，包括描述所有这种类型的元素在共享服务和底层构造的基础上如何进行交互。软件模板还包括属于这种类型的所有元素的功能。例如，每个元素必须记录某些重大事件，每个元素必须为运行期间的外部诊断提供测试点等。在软件产品线系统中，软件模板显得格外重要，因为新元素的引入是一个通用的技术，这种技术用来使产品线体系结构适应一个特定的产品。

ABSD 方法是递归的，且迭代的每一个步骤都是清晰地定义的。因此，不管设计是否完成，体系结构总是清晰的，这有助于降低体系结构设计的随意性。

12.3.1 有关术语

为了帮助读者理解后续内容，下面介绍一些基本概念。

1. 设计元素

ABSD 方法的目的是组织最早的设计决策，不包括形成实际的软件构件和类，也不包括把构件组织成进程和操作系统线程。用“实际构件”这个名字来指已经成为类、进程或线程的构件。另一方面，尽管在 ABSD 方法中不产生实际构件，但必须做出有关功能划分和达到不同质量属性的机制的决策。ABSD 方法是一个递归细化的方法，软件体系结构通过该方法得到细化，直到能产生软件构件和类。

ABSD 方法中使用的设计元素如图 12-12 所示。在最顶层，系统被分解为若干概念子系统和一个或若干软件模板。在第二层，概念子系统又被分解成概念构件和一个或若干附加软件模板。

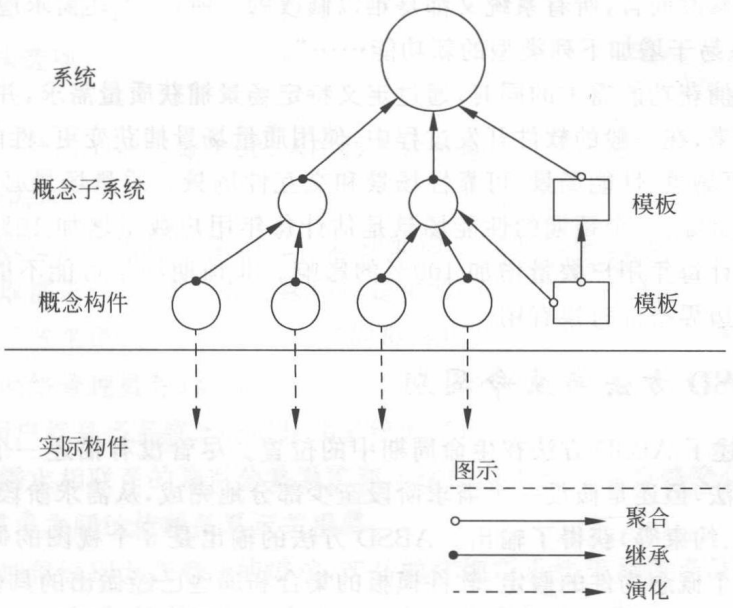


图 12-12 设计元素的分解

因为ABSD方法是递归的,对系统所使用的步骤与对概念子系统所使用的步骤是一样的,与对概念构件所使用的步骤也是一样的,使用"设计元素"这个概念来泛指软件系统、概念子系统或概念构件。设计元素有一个概念接口,该接口封装了输入和输出的数据信息,可能的一种情形是一个阶段的特定设计元素的功能在后续阶段中将发散为若干设计元素。一旦决定开始构造类、方法、进程或线程,ABSD方法就终止了。然而,经验表明,一旦概念构件已经定义好,总是可以产生实际构件。

2. 视角和视图

考虑体系结构时,重要的是从不同的视角(perspective)来检查,这促使设计师考虑体系结构的不同属性。例如,展示功能组织的静态视角能判断质量特性,展示并发行为的动态视角能判断系统行为特性。在ABSD方法中,使用不同的视角观察设计元素,一个子系统并不总是一个静态的体系结构元素,而是可以从动态和静态视角观察的体系结构元素。

将选择的特定视角或视图与Kruchten提出的类似,也就是逻辑视图、进程视图、实现视图和配置视图。使用逻辑视图记录设计元素的功能和概念接口,设计元素的功能定义了它本身在系统中的角色,这些角色包括功能、性能等。本节称第二种视图为并发视图,使用并发视图检查系统多用户的并发行为。使用"并发"代替"进程",是为了强调没有对进程或线程进行任何操作,一旦执行这些操作,则并发视图就演化为进程视图。使用的最后一个视图是配置视图,配置视图代表计算机网络中的结点,也就是系统的物理结构。配置视图只能用在多处理器的系统中。

3. 用例和质量场景

用例已经成为推测系统在一个具体设置中的行为的重要技术,它被用在很多不同的场合。本节中,用例是系统给予用户一个结果值的功能点,用来捕获功能需求。

正如用例使功能需求具体化一样,用例还必须使质量需求具体化。所谓的"系统必须易于修改"之类的需求没有多大意义,因为相对于某些修改而言,所有的系统都是易于修改的。而相对于另一些修改而言,所有系统又都是难以修改的。所以,上述需求应该按如下格式具体化:"系统应该易于增加下列类型的新功能……"。

在使用用例捕获功能需求的同时,通过定义特定场景捕获质量需求,并称这些场景为质量场景。这样一来,在一般的软件开发过程中,使用质量场景捕获变更、性能、可靠性和交互性,分别称为变更场景、性能场景、可靠性场景和交互性场景。质量场景必须包括预期的和非预期的刺激。例如,一个预期的性能场景是估计每年用户数量增加10%的影响,一个非预期的场景是估计每年用户数量增加100%的影响。非预期场景可能不能真正实现,但它们在决定设计的边界条件时很有用。

12.3.2 ABSD方法与生命周期

图12-13描述了ABSD方法在生命周期中的位置。尽管没有描述一个需求获取、组织或跟踪的特定方法,但还是假设一个需求阶段至少部分地完成,从需求阶段(包括功能需求、质量和业务需求、约束等)获得了输出。ABSD方法的输出是3个视图的概念构件的集合,包括能够产生每个概念构件的假定、软件模板的集合和那些已经做出的具体实现的决策,把具体实现决策当作附加约束来维护。

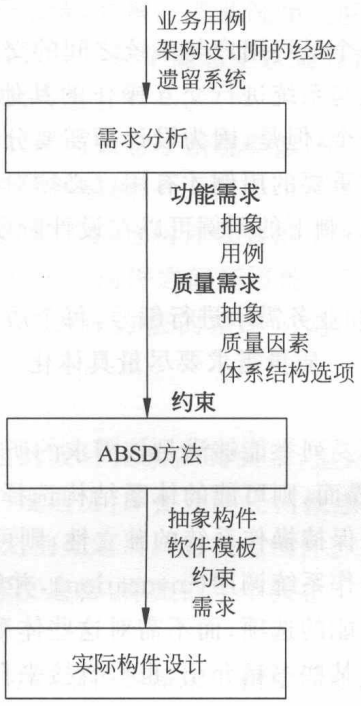


图 12-13　ABSD 方法与生命周期

在 ABSD 方法中，必须记录所有做出的决策以及这些决策的原理，这有利于决策的可跟踪性和决策评审。

ABSD 方法的输入由下列部分组成。

(1) 抽象功能需求，包括变化的需求和通用的需求。

(2) 用例(实际功能需求)。

(3) 抽象的质量和业务需求。

(4) 质量因素(实际质量和业务需求)。

(5) 体系结构选项。

(6) 约束。

下面描述需求阶段的假定输出，即 ABSD 方法的输入。

1. 抽象功能需求

ABSD 方法假定需求阶段的输出之一是功能需求的抽象描述，包括这些需求的粗略变化的描述。当获取需求时，考虑所有最终用户是重要的。

对一个特定系统来说，通常有不同类型的最终用户。不同的系统管理员(数据库管理员、系统管理员、网络管理员等)都可以是最终用户。维护工程师也可以是系统的最终用户。总之，一个最终用户就是当系统运行时使用系统的任何人员。

与抽象功能需求相联系的是对公共需求和与这些需求相关的粗略变化的描述，在设计阶段，理解这些需求之间的依赖关系至关重要。

必须在某种抽象级别上获取功能需求，产品的详细需求要等具体产品开发完成后才能知道。当详细需求明确时，抽象功能的获取为详细需求提供了分类。

2. 用例

如前所述,用例是一个或多个最终用户与系统之间的交互的具体表述。在这里,最终用户既可以是操作人员,也可以是与系统进行交互操作的其他软件系统。虽然用例很容易找到和创建,甚至可能有成百上千个,但是,因为设计师需要分析用例,所以必须限制用例的数量。在体系结构设计阶段,只有重要的用例才有用。必须对所创建的用例进行分组、设置优先级,以便筛选出最重要的用例,剩下的用例可以在设计阶段的任何时候创建。

3. 抽象的质量和业务需求

必须对待构建系统的质量和业务需求进行编号,每个质量属性都包含一个特定的刺激,以及希望得到的响应(response)。质量需求要尽量具体化。

4. 体系结构选项

对每个质量和业务需求,都要列举能够满足该需求的所有可能的体系结构。例如,如果需求是支持一系列不同的用户界面,则可能的体系结构选择就是把不同的用户界面分解成不同的构件。又如,如果需求是保持操作系统的独立性,则可能的体系结构选择就是构建虚拟的操作系统层,接受所有的操作系统调用(invocation),并解释为当前操作系统所能支持。

这个时候,只需列举所有可能的选项,而不需对这些体系结构选项进行决策。这种列举取决于设计师的经验,既可来自某些书籍介绍,也可直接来自设计师本身的实践。

5. 质量场景

正如用例使功能需求具体化一样,质量场景使质量需求具体化。质量场景是质量需求的特定扩充。

与用例一样,质量场景也很容易找到和创建,可以创建很多个。必须对质量场景进行分组、设置优先级,只需验证最重要的质量场景。

6. 约束

约束是一个前置的设计决策,设计过程本身包含决策。某些决策可以直接由业务目标导出而无须考虑对设计的影响。例如,如果一个公司在某个中间件产品上投入了大量资金,那么在产品的选择上就可以不必考虑其他决策。在需求获取阶段,约束主要来自系统的业务目标。

在某些特殊情况下,约束由遗留系统决定。几乎没有软件系统不参考已有系统。常见的情况是,新老系统同时并存,或者新系统替代老系统,但是必须尽可能重用老系统的功能。在设计阶段,虽然这些遗留系统处于被设计系统的外部,但设计师必须考虑遗留系统的特征。也就是说,在某种程度上,遗留系统影响着当前的设计。因此,理解遗留系统的结构和解决问题的技术都很重要。出于商业目的,可能要求重用遗留系统的构件,这种需求就变成了约束。

12.3.3 ABSD 方法的步骤

1. ABSD 方法定义的设计元素

ABSD 方法是基于整个系统的分解,设计元素的分解如图 12-12 所示。

每个系统都由应用部分和基础部分组成,尽管这两个部分之间的边界通常并不明显,设计时还是要考虑应用和基础需执行的功能。ABSD 方法通过把系统当作应用部分和基础部分的结合体来获取这两类基础需求,这两个部分都归结于系统的分解和后续定义。系统的

顶层分解是把系统分解为概念子系统，与概念子系统相关联的是子系统模板。由概念子系统和子系统模板聚合成整个软件系统。每个概念子系统有自己的相对独立的功能，可通过逻辑视图、并发视图和配置视图来考察。

概念子系统被分解成概念构件，与概念子系统类似，每一个概念构件也可通过3种视图来考察。与概念构件相关联的是构件模板。

在图12-12中，把实际构件放到了概念构件分解的下一步。一个实际构件反映了一个软件元素，例如类。在概念构件和实际构件之间还可能有附加的设计元素，这取决于所设计系统的大小。但是，对于使用ABSD方法的系统来说，这些附加级的设计元素不是必需的。

2. 设计元素的产生顺序

图12-12展示了ABSD方法的设计元素及其关系，但并没有指出元素树的遍历过程。一种可能的方法是对该树进行广度遍历，即在分解任何概念子系统之前先定义所有概念子系统，接着在构建任何实际构件之前先定义所有概念构件。

另一种方法是对该树进行深度遍历，即先把某个概念子系统分解成若干概念构件，然后把该概念构件再分解成实际构件。

在产生概念子系统时，洞察可能获得的需求可以增加新的需求或修改已有需求，在创建概念构件期间，可根据新的信息重新修改在定义概念子系统期间做出的决策。

为了理解设计选择和选择某个选项进行具体实现，有关设计的某些观点的详细调查可以在过程中的任何时候进行。

对一个特定开发来说，决定遍历设计元素树的考虑如下。

(1) 领域知识。如果设计师有广泛的领域知识，则探索就不一定是必需的了。

(2) 新技术的融合。如果新技术被当作中间件或操作系统，则必须构造原型。这不但可以帮助理解新技术的性能和限制，还可以为体系结构设计团队提供使用新技术的经验。

(3) 体系结构设计团队的个人经验。具有不同经验的设计人员可以研究设计元素树的不同部分。

图12-14描述了设计元素A分解为两个小的设计元素B和C。下面用图12-14讨论需求和功能之间的内部协作关系。首先，单独考察A，A必须满足某些需求(功能、业务和性能)，这些需求由A的功能来满足。把A分解为B和C的过程中，A的功能也被分解成对B和C的需求。

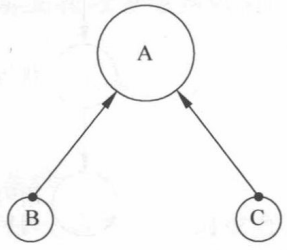


图12-14　设计元素A分解为两个小的设计元素B和C

3. 设计元素的活动

为了分解设计元素，讨论了对设计元素树进行遍历的算法，但没有讨论究竟如何分解一个设计元素。根据ABSD方法，可以利用一组需求(包括功能需求和性能需求)、适合设计元素的一个模板和一组约束来开始分解每个设计元素。这个过程的输出是一个子设计元素列表，每个子设计元素都有一组需求、一组适合于自身的模板和一组约束。图12-15是设计元素分解的一般过程序列示意图，在校验和逻辑视图的定义之间有一个反馈环。当然也可能从其他视图开始，反馈环用来保证其他视图也在考虑之列。例如，如果系统被设计成具有非通常的连通性，则设计师可能希望从配置视图开始。

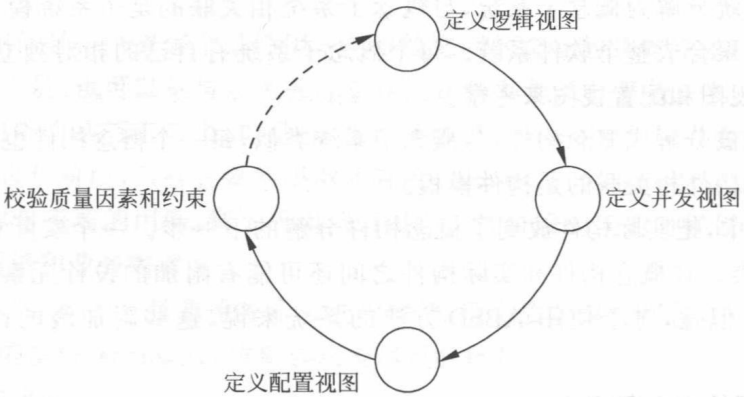


图 12-15　分解一个设计元素的步骤

要注意的是,虽然在图 12-15 中没有注明,其实也可能从并发视图或配置视图直接反馈回逻辑视图。

图 12-16 是逻辑视图定义的步骤示意图。在这个情况下,功能得以分解,一个基本的体系结构风格被选择,功能被分配给所选的风格。

和 ABSD 的所有步骤一样,完成了一个步骤后将对前一个步骤进行重新考虑。例如,创建并发视图时,可能要识别在需求中没有考虑到的附加功能。这种功能可以引起对为设计元素重新考虑基本体系结构风格。而且,一个步骤的讨论可以揭示属于后一个步骤的信息。例如,在功能分解的讨论中,可以发现模板项,这些项一旦被发现,就需要记录下来。

作为一个例子,假设某系统的逻辑视图如图 12-17 所示。

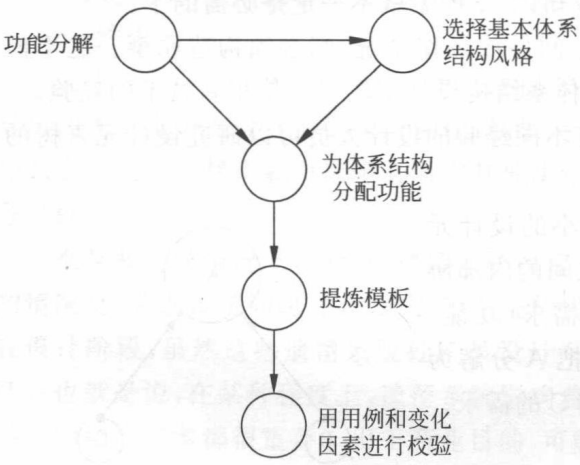


图 12-16　定义逻辑视图

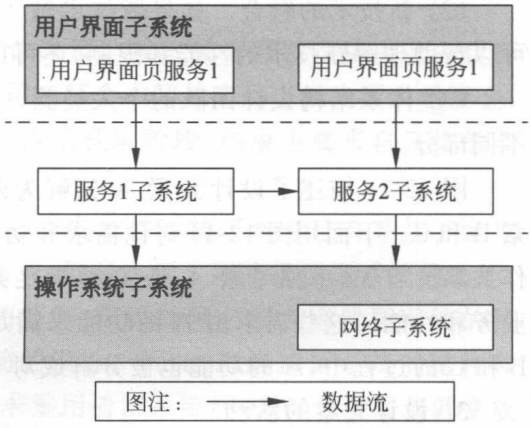


图 12-17　子系统结构逻辑视图

在图 12-17 中,采用分层体系结构,用矩形表示概念子系统或概念构件的设计元素。系统共有 4 个概念子系统,分别为用户界面子系统、服务 1 子系统、服务 2 子系统和操作系统子系统。在用户界面子系统和操作系统子系统中,又包含更小的构件。每个概念子系统和概念构件有一个功能列表,该列表最初是基于功能需求和用例需求的。

1) 功能分解

一个设计元素有一组功能,这些功能必须分组。分解的目的是使每个组在体系结构内

代表独立的元素。分解可以进一步细化。这种分解的标准取决于对一个特定的设计元素来说是很重要的性能。在不同的性能基础上，可以进行多重分解。

如果像通常的产品一样，在分解中起关键作用的性能要求是可修改的，则功能的分组可选择以下几个标准。

(1) 功能聚合。需求分组必须遵守“高内聚低耦合”的原则，这是对功能分解的一个标准技术。用例能够用来检查内聚和耦合，用来处理变化的性能因素也可用来检查内聚和耦合。

(2) 数据或计算行为的类似模式。在数据或计算行为上，如果有类似模式的功能，则应该分在同一组。这意味着展示类似行为取决于使用系统的特定领域。如果数据获取是一个功能，则一个类似模式可能是样本周期。如果一个特定功能需要很大计算量，则它应该与其他计算量的功能分在一组。用同一模式存取数据库的功能也应该分在一组。

(3) 类似的抽象级别。与硬件相近的功能不应该与那些抽象级别较高的功能分在一组。另一方面，处在同一抽象级别的功能应该分在一组。

(4) 功能的局部性。为其他服务提供服务的功能不应该与纯局部功能分在一组。

2) 选择体系结构风格

每个设计元素有一个主要的体系结构风格或模式，这是设计元素完成它的功能的基础。主要风格并不是唯一风格，为了达到特定目的，可以进行修改。体系结构风格的选择建立在设计元素的体系结构驱动基础上。因此，这个过程就是为设计元素决定体系结构驱动和从体系结构驱动及功能分解角度考虑问题，决定主要的体系结构风格。

在体系结构设计过程中，并不总是有现成的体系结构风格可供选择。在这种情况下，设计师应该设计一种新的风格适合实际情况。有时，一个设计元素的功能包含一组相对独立的子功能，一个主要的风格不足以辨识。在这种情况下，选择一个具有与设计元素外界连接的独立的过滤器风格，是一种好的选择。

一旦选定了一个主要的体系结构风格，该风格必须适应基于属于这个设计元素的质量需求，体系结构选择必须满足质量需求。也就是说，要检查每一个质量需求，判断它是否与所分解的设计元素有关；如果有关，则选择一个选项与质量需求关联和应用到风格的选择之中。

当一个特定的质量需求与多个设计元素有关时，希望使用同样的选择，以使该选择在设计记录中与该设计元素相关联。

选择和细化一个体系结构风格的结果是一组体系结构构件类型。例如，可以是一个“客户”类型，该“客户”既与客户机所要计算的功能没有关系，也与存在多少个实例没有关系。这些问题都将在下一步决定。一些构件类型，特别是由质量需求导出的构件，可能与功能有关联，如“虚拟设备”。

为设计元素选择体系结构风格是一个重要的选择，这种选择在很大程度上依赖于设计师的个人设计经验。可供选择的风格有运行时风格(如客户-服务器结构)、开发时风格(如层次式结构)，或者二者兼有(如三层结构)。在任何情况下，风格的选择都可能产生附加功能，这些附加功能必须加到功能组中。

3) 为风格分配功能

选择体系结构风格时产生了一组构件类型，必须决定这些类型的数量和每个类型的功

能,这就是分配的目的。在功能分解时产生的功能组,应该分配给选择体系结构风格时产生的构件类型,包括决定将存在多少个每个构件类型的实例,每个实例将完成什么功能。这样分配后产生的构件将作为设计元素分解的子设计元素。

每个设计元素的概念接口必须得到标识。这个接口包含设计元素所需的信息和在已经定义了的体系结构风格内的每个构件类型所需要的数据和控制流。

重复以上3个步骤,需在各种质量属性之间进行折中处理,设计师必须判断在哪里折中合适。

4) 细化模板

被分解的设计元素有一组属于它的模板。在ABSD方法的初期,系统没有模板。当模板细化了以后,就要把功能增加上去。这些功能必须由实际构件在设计过程中加以实现。

对于存在的模板的每个功能来说,要考虑如下问题:这些功能的方方面面是否都由子设计元素处理了,或者这些功能仍然留在当前区域?该问题有以下两种可能答案。

(1) 这些功能仍然留在当前区域。此时,无须做任何事情。

(2) 功能将在各方面进行分解,满足当前区域。

子设计元素的功能也要进行检查,以决定是否应该加到模板中。这些是:

(1) 这些功能可以被某些子设计元素所共享,而不是某个子设计元素专有;

(2) 每个子设计元素必须管理的职责,如错误处理、活动日志或为外部诊断提供检查点。

最后,需要检查模板的功能,以判断是否需要增加附加功能到系统任何地方的设计元素中。也就是说,要识别在该级别上已经存在的任何横向服务。模板包括什么是一个好的设计元素和那些应该共享的功能。每种类型的功能可以需要附加支持功能,这种附加功能一旦得到识别,就需要进行分配。

5) 功能校验

用例用来验证设计元素,验证它们能否通过一定的结构达到目的。子设计元素的附加功能将可能通过用例的使用得到判断。然而,在功能分解的过程中,如果过度地使用用例,将很难发现附加功能。

也可以使用变化因素,因为执行一个变化的难点取决于功能的分解。

从这种类型的校验出发,设计就是显示需求(通过用例)和支持修改(通过变化因素)。

6) 创建并发视图

检查并发视图的目的是判断哪些活动是可以并发执行的。必须识别这些活动,以产生进程同步和资源竞争。

对并发视图的检查是通过虚拟进程实现的。虚拟进程是通过程序、动态模块或一些其他的控制流执行的一条单独路径。虚拟进程与操作系统的进程概念不一样,操作系统的进程包括额外的地址空间的分配和调度策略。一个操作系统进程是几个虚拟进程的连接点,但每个虚拟进程不一定都是操作系统进程。虚拟进程用来描述活动序列,使同步或资源竞争可以在多个虚拟进程之间进行。

用例用来检查两个用户的影响,并行性影响一个用户的活动。

发现同步和资源竞争可能增加新的功能。例如,资源管理可以增加,用来管理竞争。在这种情况下,新功能必须分配为设计元素的功能。

例如，图 12-18 描述了一个用例确定在初始化阶段必须发生的数个功能的初始化过程，其中箭头表示线程。

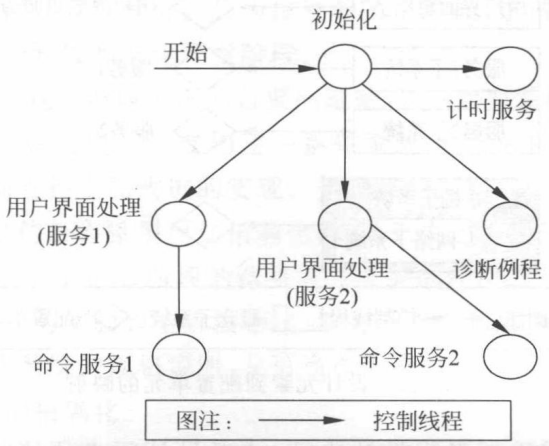


图 12-18　初始化阶段的进程视图

假设系统在启动后，直接进入初始化阶段。在这个阶段，一些后台事务（如计时服务和诊断）开始执行，而且假设用户界面必须准备就绪，能够接收用户的输入，还假设启动了两个不同的相互独立的和并行运行的用户界面。其中最后一个假设的目的是使用户界面一定总是准备就绪，以接收外部输入，这就意味着用户界面必须与先前执行的命令并行运行。

图 12-18 中，椭圆代表在分析该用例阶段确认的特定功能，其中有 3 个功能（计时、诊断和初始化）不是在先前就确定为任何设计元素的功能的。

注意：*并发视图描述了贯穿设计元素的不同控制线程，也就是说，设计元素及其功能是每个视图的基本之所在。对视图进行推理，将导致为设计元素增加功能。*

7）创建配置视图

如果在一个系统中，使用了多个处理器，则需要对不同的处理器配置设计元素，这种配置通过配置视图进行检查。例如，检查网络对虚拟线程的影响，一个虚拟线程可以通过网络从一个处理器传递到另一个处理器。使用物理线程描述在某个特定处理器中的线程。也就是说，一个虚拟线程是由若干物理线程串联而成的。通过这种视图，可以发现一个单一的处理器上的同步的物理线程和把一个虚拟线程从一个处理器传递到其他处理器上的需求。

在配置视图中，使用配置单元的概念，配置单元是能分配给处理器的最小设计元素，其准确大小取决于设计元素的粒度。设计师必须做出哪个级别的设计元素组成一个配置单元的决策。如果一个设计元素的粒度大于配置单元，则需要对其进行分解，分配给数个处理器。如果一个设计元素的粒度小于或等于配置单元，则创建配置视图就意味着必须做出配置决策。

例如，在图 12-17 的基础上，可以确认下列配置单元：用户界面页服务 1、用户界面页服务 2、服务 1、服务 2 和操作系统。图 12-19 描述了不同的设计元素是如何映射为配置单元的。

8）验证质量场景

一旦创建了 3 个视图，就要把质量场景应用到所创建的子设计元素上。对每个质量场景，都要考虑是否仍然满足需求，每个质量场景包括一个质量属性刺激和所期望的响应。考虑到目前为止所做出的设计决策，看其是否能够达到质量属性的要求。

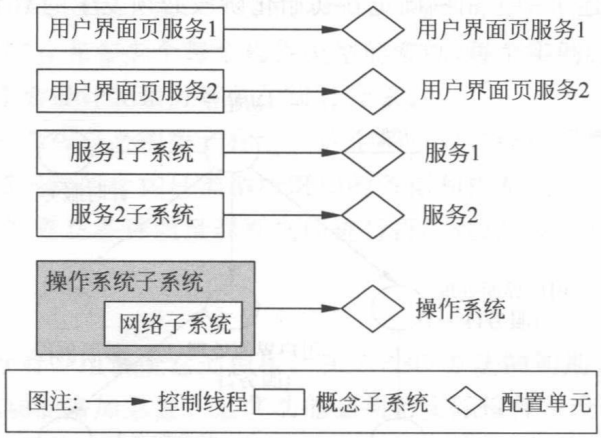


图 12-19　设计元素到配置单元的映射

如果不能达到,则需重新考虑设计决策,或者设计师必须接受创建质量场景失败的现实。

9) 验证约束

最后一步就是要验证所有的约束没有互相矛盾的地方,对每一个约束,都需提问"该约束是否有可能实现?"一个否定的回答就意味着对应的质量场景不能满足。这时,需要把问题记录进文档,对导致约束的决策进行重新验证。

12.4　体系结构的设计与演化

面向对象已经成为软件开发方法的主流思想,而以演化和增量(increment)方法为基础的迭代开发过程已经成为面向对象开发过程的标准。然而,应用这些迭代方法的软件开发也导致了很多新的问题。使用传统的瀑布过程进行软件开发,可以对将要开发的软件进行明确的描述,也就是说,在软件产品开发出来之前,人们就已经知道该软件将具有什么功能,满足哪些性能。当使用迭代过程进行开发时,随着软件需求不断地发生变化,最终软件产品可能与初始原型(initial prototype)相差很大。如果软件体系结构设计不当,在每一个演化过程中,可能需要修改很多模块和代码,甚至修改整个软件体系结构。

对于软件项目的开发来说,一个清晰的软件体系结构是首要的,即使在初始原型阶段也不例外。然而,在系统开发的初始阶段就设计好系统的最终结构是不可能的,也是不现实的,因为需求还在不断地发生变化。所以,一个好的软件体系结构应该可以创建或再创建功能、用户界面和问题域(problem domain)模型,演化原型以满足新的软件需求。也就是说,软件体系结构本身也是可演化的,这种演化可基于需求的变化,增进对问题域的理解,对实现系统的技术方式的进一步理解。从这种意义上来说,不但软件系统以原型方式演化,体系结构本身也以原型方式演化。

一个软件系统开发完毕正式投入使用之后,如果要将该系统移植到另一个环境运行,且新环境的需求也有相应的变化时,软件也要进行修改。通常,这种修改所需的工作量与软件需求变化的多少和变化的范围有直接关系。但是,一个好的软件体系结构能大大减少修改工作量。本节以正交软件体系结构的设计和演化为例,讨论软件体系结构的设计和演化过程。

12.4.1 设计和演化过程

基于体系结构的软件开发过程可以分为独立的两个阶段，这两个阶段分别是实验原型(experimental prototype)阶段和演化开发阶段。

(1) 实验原型阶段。这一阶段考虑的首要问题是，要获得对系统支持的问题域的理解。为了达到这个目的，软件开发组织需要构建一系列原型，与实际的最终用户进行讨论和评审，这些原型应该演示和支持全局改进的实现。但是，来自用户的最终需求是很模糊的，因此，第一个阶段的作用是使最终系统更加精确化，有助于决定实际开发的可行性。

(2) 演化开发阶段。实验原型阶段的结果可以决定是否开始实现最终系统，如果可以，开发将进入第二个阶段。与实验原型阶段相比，演化开发阶段的重点放在最终产品的开发上。这时，原型既被当作系统的规格说明，又可当作系统的演示版本。这意味着演化开发阶段的重点将转移到构件的精确化。

虽然实验原型阶段的结果可以决定是否开始实现最终系统，但在实验原型阶段之后，并不是所有的功能需求都已经足够准确。然而，系统有哪些组成部分和这些部分该如何相互作用应该是明确的了。

在每个阶段中，都必须以一系列的开发周期为单位安排和组织工作，一个开发周期的时间长短可根据软件项目的性质、功能复杂性、开发阶段等因素决定。每一个开发周期都要有不同的着重点，要有一个分析、设计和实现的过程，这个过程取决于当前对系统的理解和前一个开发周期的结果。为了控制开发进度，在每一开发周期结束时，都必须对当前产品安排一次技术评审，评审组成员由最终用户代表和开发组织的管理人员组成。技术评审的目的是，指出当前产品中可能存在的问题，制订下一开发周期的工作计划。

12.4.2 实验原型阶段

通常实验原型阶段的第一个开发周期没有具体的、明确的目标。此时，为了提高开发效率，缩短开发周期，所有开发人员可以分成两个小组，一个小组创建图形用户界面，另一个小组创建一个问题域模型。两个小组要并行地工作，尽量不要发生相互牵制的现象。

在第一个周期结束时，形成了两个版本，一个是图形用户界面的初始设计，主要包括一些屏幕元素，如窗口、菜单等；另一个是问题域模型，该模型覆盖了问题域的子集。用户界面设计由水平原型表示，也就是说，运行的程序只是实现一些用户界面控制，没有实现真正的系统功能。问题域模型可由一个 UML 类图表示，该类图并不是运行的原型的一部分。然而，它并不只是一个简单的类图，可由一个 CASE 工具(如 Rational Rose)自动产生代码，而且，当一个新的元素增加到模型中时，这些代码会自动进行增量更新。

第二个开发周期的任务是设计和建立一个正交软件体系结构，该结构不但应具有 3.6 节中描述的特征，还应该具有以下特征。

(1) 必须足够灵活，不但能包含现有的元素，而且能包含新增的功能。

(2) 必须提供一个相当稳定的结构，在这个结构中，原型能在实验原型阶段进行演化。

(3) 必须支持一个高效的开发组织，允许所有开发人员并行地在原型的基础上进行开发。

整个第二个开发周期又可细分为以下 6 个小阶段。

(1) 标识构件。为系统生成初始逻辑结构,包含大致的构件。这一阶段可分为3个小步骤。

第一步,生成类图。生成类图的CASE工具有很多,例如,图12-20是由Rational Rose自动生成的类图(图中的箭头表示继承关系,菱形表示聚合关系。没有箭头的实线表示双向关联,其中的数字1..n或0..n分别表示"一到多"和"零到多"的关系。虚线表示具有"或关系"的关联)。

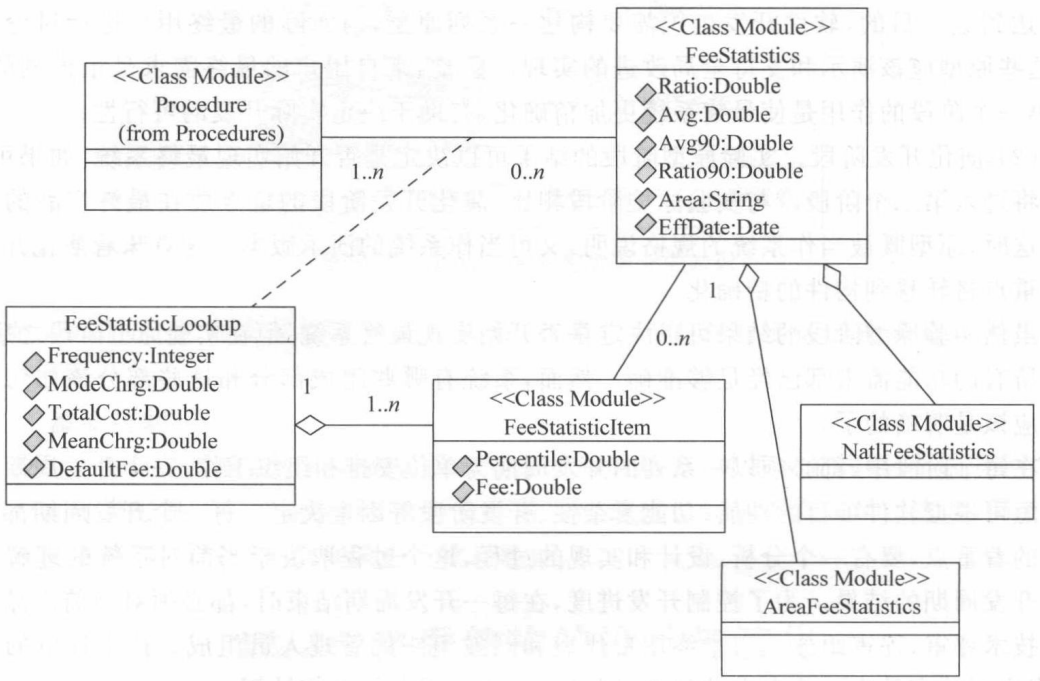


图12-20 Rational Rose自动生成的类图

第二步,对类进行分组。使用一些标准对类进行分组可以大大简化类图。一般地,与其他类隔离的类形成一个组,由泛化(generalization)关联的类组成一个附加组,由聚合(aggregation)或组合(composition)关联的类也形成一个附加组。

第三步,把类打包成构件。把在第二步得到的类簇打包成构件,这些构件可以分组合并成更大的构件。

(2) 提出软件体系结构模型。在建立体系结构的初期,选择一个合适的体系结构风格是首要的。在这个风格基础上,开发人员通过体系结构模型,可以获得关于体系结构属性(如程序逻辑结构、开发平台等)的理解。此时,虽然这个模型是理想化的(其中的某些部分可能错误地表示了应用的特征),但是,该模型为将来的调整和演化过程建立了目标。

(3) 把已标识的构件映射到软件体系结构中。把在第(1)阶段已标识的构件映射到体系结构中,将产生一个中间结构,这个中间结构只包含那些能明确适合体系结构模型的构件。

(4) 析构件之间的相互作用。为了把所有已标识的构件集成到体系结构中,必须认真分析这些构件的相互作用和关系。可以使用UML的顺序图完成这个任务,图12-21是由Rational Rose生成的一个顺序图(其中的线条人表示参与者,箭头表示对象间的通信)。

(5) 产生软件体系结构。一旦决定了关键的构件之间的关系和相互作用,就可以在第(3)阶段得到的中间结构的基础上进行细化。可以利用顺序图标识中间结构中的构件和剩下的构件之间的依赖关系,分析第(2)阶段模型的不一致性,如丢失连接等。

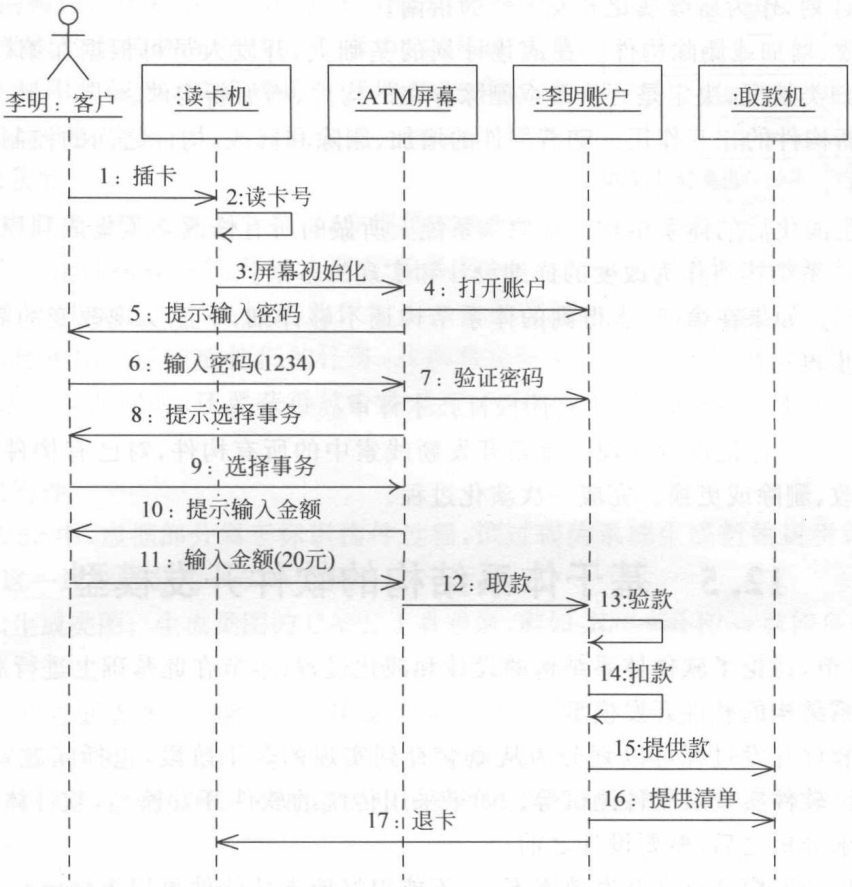


图 12-21　用 Rational Rose 生成的顺序图

(6) 软件体系结构正交化。在(1)～(5)阶段产生的软件体系结构不一定满足正交性，例如，同一层次的构件之间可能存在相互调用。整个正交化过程以原体系结构的线索和构件为单位，自顶向下、由左到右进行。通过对构件的新增、修改或删除，调整构件之间的相互作用，把那些不满足正交性的线索进行正交化。

12.4.3　演化开发阶段

一旦软件的正交体系结构得以确定，就可以开始正式的构件开发工作。由于体系结构的正交性，可以把开发人员分成若干小组进行并行开发，视开发难度情况，每个小组负责一条或数条线索。由于各条线索之间没有相互调用，所以各小组工作不会相互牵制。这样，可大大提高编程的效率，缩短开发周期。

在构件开发过程中，最终用户的需求可能还有变动。在软件开发完毕，正常运行后，由一个单位移植到另一个单位，需求也会发生变化。这两种情况下，必须使用系统演化步骤去修改应用，以满足新的需求，主要包括以下 8 个步骤。

(1) 需求变动归类。首先必须对用户需求的变化进行归类，使变化的需求与已有构件和线索对应。对找不到对应构件和线索的变动，也要做好标记，在后续工作中，将创建新的构件或线索，以对应这部分变化的需求。

(2) 制订体系结构演化计划。在改变原有结构之前，开发组织必须制订一个周密的体

系结构演化计划,作为后续演化开发工作的指南。

(3) 修改、增加或删除构件。在演化计划的基础上,开发人员可根据在第(1)步得到的需求变动的归类情况,决定是否修改或删除存在的构件、增加新构件。

(4) 更新构件的相互作用。随着构件的增加、删除和修改,构件之间的控制流必须得到更新。

(5) 产生演化后的体系结构。在原来系统上所做的所有修改必须集成到原来的体系结构中。这个体系结构将作为改变的详细设计和实现的基础。

(6) 迭代。如果在第(5)步得到的体系结构还不够详细,不能实现改变的需求,可以把第(3)~(5)步再迭代一次。

(7) 对以上步骤进行确认,进行阶段性技术评审。

(8) 对所做的标记进行处理。重新开发新线索中的所有构件,对已有构件按照标记的要求进行修改、删除或更换。完成一次演化过程。

12.5 基于体系结构的软件开发模型

在12.4节,讨论了软件体系结构的设计和演化过程,本节在此基础上进行总结和推广,讨论基于体系结构的软件开发模型。

传统的软件开发过程可以划分为从概念直到实现的若干阶段,包括问题定义、需求分析、软件设计、软件实现及软件测试等。如果采用传统的软件开发模型,软件体系结构的建立应位于需求分析之后,概要设计之前。

传统软件开发模型存在开发效率不高,不能很好地支持软件重用等缺点。本节在12.4节的基础上介绍一个基于体系结构的软件开发模型(ABSDM)。ABSDM模型把整个基于体系结构的软件过程划分为体系结构需求、体系结构设计、体系结构文档化、体系结构复审、体系结构实现和体系结构演化6个子过程,如图12-22所示。

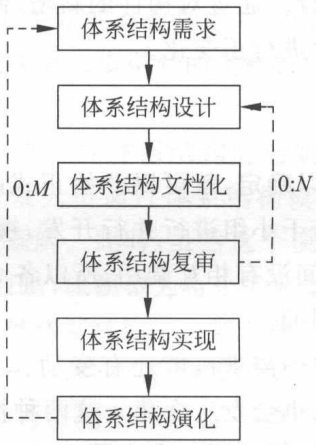


图12-22 基于体系结构的软件开发模型

12.5.1 体系结构需求

体系结构需求是指用户对目标软件系统在功能、行为、性能、设计约束等方面的期望。体系结构需求受技术环境和设计师的经验影响。需求过程主要是获取用户需求,标识系统

中所要用到的构件。体系结构需求过程如图 12-23 所示。如果以前有类似的系统体系结构的需求，可以从需求库中取出，加以利用和修改，以节省需求获取的时间，减少重复劳动，提高开发效率。

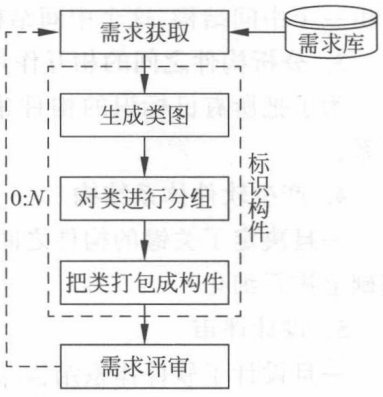


图 12-23　体系结构需求过程

1. 需求获取

体系结构需求一般来自 3 个方面，分别是系统的质量目标、系统的业务目标和系统开发人员的业务目标。软件体系结构需求获取过程主要是定义开发人员必须实现的软件功能，使得用户能完成他们的任务，从而满足业务上的功能需求。与此同时，还要获得软件质量属性，满足一些非功能需求。

2. 标识构件

在图 12-23 中，虚框部分属于标识构件过程，该过程为系统生成初始逻辑结构，包含大致的构件。这一过程可分为 3 步来实现。

第一步，生成类图。生成类图的 CASE 工具很多，例如，Rational Rose 就能自动生成类图。

第二步，对类进行分组。在生成的类图基础上，使用一些标准对类进行分组可以大大简化类图结构，使之更清晰。一般地，与其他类隔离的类形成一个组，由泛化关联的类组成一个附加组，由聚合或组合关联的类也形成一个附加组。

第三步，把类打包成构件。把在第二步得到的类簇打包成构件，这些构件可以分组合并成更大的构件。

3. 需求评审

组织一个由不同代表（如分析人员、客户、设计人员和测试人员）组成的小组，对体系结构需求及相关构件进行仔细的审查。审查的主要内容包括所获取的需求是否真实反映了用户的要求，类的分组是否合理，构件合并是否合理等。

必要时，可以在“需求获取—标识构件—需求评审”之间进行迭代。

12.5.2　体系结构设计

体系结构需求用来激发和调整设计决策，不同的视图被用来表达与质量目标有关的信息。体系结构设计是一个迭代过程，如果要开发的系统能够从已有的系统中导出大部分，则可以使用已有系统的设计过程。软件体系结构设计过程如图 12-24 所示。

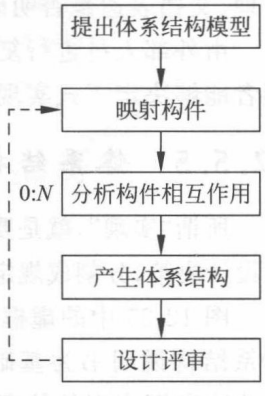


图 12-24　体系结构设计过程

1. 提出软件体系结构模型

在建立体系结构的初期，选择一个合适的体系结构风格是首要的。在这个风格基础上，开发人员通过体系结构模型，可以获得关于体系结构属性的理解。此时，虽然这个模型是理想化的（其中的某些部分可能错误地表示了应用的特征），但是，该模型为将来的实现和演化过程建立了目标。

2. 把已标识的构件映射到软件体系结构中

把在体系结构需求阶段已标识的构件映射到体系结构中，将

产生一个中间结构,这个中间结构只包含能明确适合体系结构模型的构件。

3. 分析构件之间的相互作用

为了把所有已标识的构件集成到体系结构中,必须认真分析这些构件的相互作用和关系。

4. 产生软件体系结构

一旦决定了关键的构件之间的关系和相互作用,就可以在第2阶段得到的中间结构的基础上进行细化。

5. 设计评审

一旦设计了软件体系结构,必须邀请独立于系统开发的外部人员对体系结构进行评审。

12.5.3 体系结构文档化

绝大多数的体系结构都是抽象的,由一些概念上的构件组成。例如,层的概念在任何程序设计语言中都不存在。因此,要让开发人员去实现体系结构,还必须对体系结构进行文档化。文档是在系统演化的每一个阶段,系统设计与开发人员的通信媒介,是为验证体系结构设计和提炼或修改这些设计(必要时)所执行预先分析的基础。

体系结构文档化过程的主要输出结果是体系结构需求规格说明和测试体系结构需求的质量设计说明书这两个文档。生成需求模型构件的精确的形式化的描述,作为用户和开发者之间的一个协约。

软件体系结构的文档要求与软件开发项目中的其他文档是类似的。文档的完整性和质量是软件体系结构成功的关键因素。文档要从使用者的角度进行编写,必须分发给所有与系统有关的开发人员,且必须保证开发者手上的文档是最新的。

12.5.4 体系结构复审

从图12-22中可以看出,体系结构设计、体系结构文档化和体系结构复审是一个迭代过程。从这个方面来说,在一个主版本的软件体系结构分析之后,要安排一次由外部人员(用户代表和领域专家)参加的复审。

体系结构复审的目的是标识潜在的风险,及早发现体系结构设计中的缺陷和错误,包括体系结构能否满足需求、质量需求是否在设计中得到体现、层次是否清晰、构件的划分是否合理、文档表达是否明确、构件的设计是否满足功能与性能的要求等。

由外部人员进行复审的目的是保证体系结构的设计能够公正地进行检验,使组织的管理者能够决定正式实现体系结构。

12.5.5 体系结构实现

所谓"实现",就是要用实体来显示出一个软件体系结构,即要符合体系结构所描述的结构性设计决策,分割成规定的构件,按规定方式互相交互。体系结构的实现过程如图12-25所示。

图12-25中的虚框部分是体系结构的实现过程。整个实现过程是以复审后的文档化的体系结构说明书为基础的,每个构件必须满足软件体系结构中说明的对其他构件的责任。这些决定即实现的约束是在系统级或项目范围内做出的,每个构件上工作的实现者是看不见的。

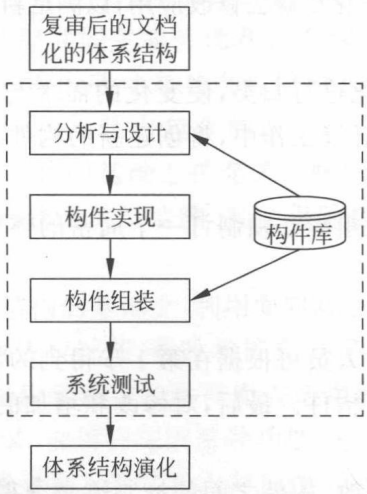


图 12-25　体系结构实现过程

在体系结构说明书中，已经定义了系统中的构件与构件之间的关系。因为在体系结构层次上，构件接口约束对外唯一地代表了构件，所以可以从构件库中查找符合接口约束的构件，必要时开发新的满足要求的构件。

然后，按照设计提供的结构，通过组装支持工具把这些构件的实现体组装起来，完成整个软件系统的连接与合成。

最后一步是测试，包括单个构件的功能性测试和被组装应用的整体功能和性能测试。

12.5.6　体系结构演化

在构件开发过程中，最终用户的需求可能还有变动。在软件开发完毕，正常运行后，由一个单位移植到另一个单位，需求也会发生变化。在这两种情况下，就必须相应地修改软件体系结构，以适应新的变化了的软件需求。体系结构演化过程如图 12-26 所示。

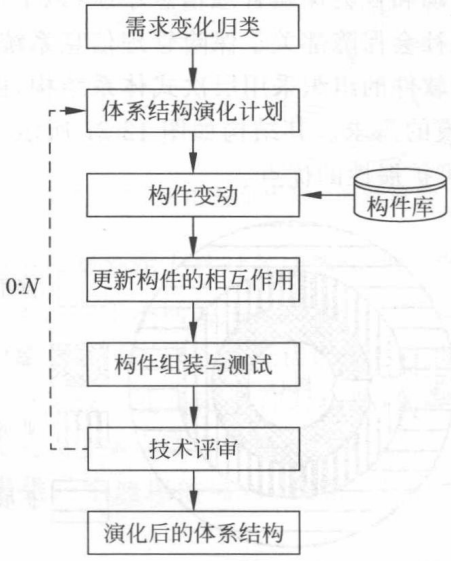


图 12-26　体系结构演化过程

体系结构演化是使用系统演化步骤去修改应用,以满足新的需求,主要包括以下7个步骤。

1. 需求变化归类

首先必须对用户需求的变化进行归类,使变化的需求与已有构件对应。对找不到对应构件的变动,也要做好标记,在后续工作中,将创建新的构件,以对应这部分变化的需求。

2. 制订体系结构演化计划

在改变原有结构之前,开发组织必须制订一个周密的体系结构演化计划,作为后续演化开发工作的指南。

3. 修改、增加或删除构件

在演化计划的基础上,开发人员可根据在第1步得到的需求变动的归类情况,决定是否修改或删除存在的构件、增加新构件。最后,对修改和增加的构件进行功能性测试。

4. 更新构件的相互作用

随着构件的增加、删除和修改,构件之间的控制流必须得到更新。

5. 构件组装与测试

通过组装支持工具把这些构件的实现体组装起来,完成整个软件系统的连接与合成,形成新的体系结构。然后对组装后的系统整体功能和性能进行测试。

6. 技术评审

对以上步骤进行确认,进行技术评审。评审组装后的体系结构是否反映需求变动,符合用户需求。如果不符合,则需要在第2~6步进行迭代。

7. 产生演化后的体系结构

在原来系统上所做的所有修改必须集成到原来的体系结构中,完成一次演化过程。

12.6 应用开发实例

12.6.1 系统简介

作者曾经负责了某省劳动和社会保险管理信息系统(以下简称SIMIS)的设计与开发,SIMIS服从于国家原劳动和社会保障部关于保险管理信息系统的总体规划,系统建设坚持一体化的设计思想。SIMIS软件的组织采用层次式体系结构,由内向外各层逐渐进行功能扩展,满足用户不同系统规模的需求。其结构如图12-27所示。这种结构组织方式具有便于增加新功能,使系统具有可扩展性的优点。

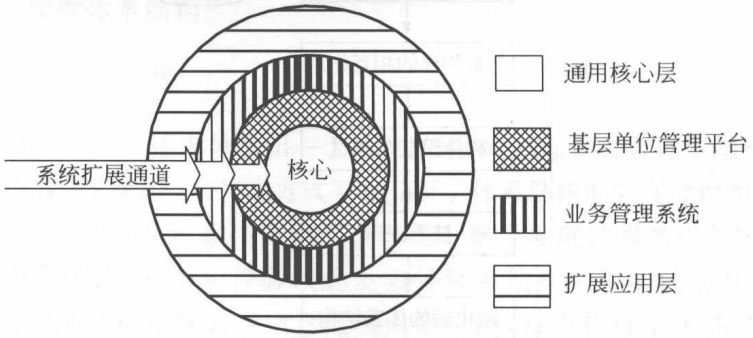


图12-27 系统软件的层次式结构

通用核心层完成与具体业务无关的基本操作；基本应用层完成与劳动和社会保险业务相关的基本操作，它与通用核心层共同完成劳动和社会保险业务的基本操作，例如，基层参保单位及个人资料的管理等，构成劳动和社会保险管理信息系统的基本系统；业务管理层能够实现数据的初步汇总，它与其内包含的两层一起构成了 SIMIS 的典型应用系统；扩展应用层是在典型应用系统的基础上扩充了一些更为复杂的功能，例如，对政策决策提供依据和支持，对政策执行状况进行监测、社会保险信息发布、个人账户电话语音查询系统等。

SIMIS 在层次式软件体系结构的基础上，利用面向对象的继承、封装、多态等特性，外层能够继承内层的所有功能，并可进行屏蔽、修改和扩充，从而实现功能的逐层扩展。通过抽象，能够将系统的大多数公共操作和通用的数据库表结构提取出来，实现一个 SIMIS 的基本操作库(基本类库)。通过封装，能够将完成某种功能的一系列操作和数据结构封装在一个模块中，隐藏内部的具体实现过程。通过继承和重载，后代不但能够方便地获得、扩充或者修改祖先的功能，而且还可以达到通过少量修改内层的方法来实现软件的可扩展，从而解决劳动和社会保险政策和措施不断变化、软件难以适应的问题。

1. 通用核心层

通用核心层完成的是软件的一些通用的公共操作，这些操作能够尽量做到不与具体的数据库和表结构相关。通用核心层操作不但可以应用于 SIMIS 中，还可以方便地移植到其他的应用软件上。从面向对象的角度看，通用核心层构成了一个基类。以后各层的操作都是在继承基类的基础上，逐渐增强功能而得到的。通用核心层中主要含有以下一些基类。

(1) 通用打印基类。获取用户定义的数据，按用户定义的格式输出到打印机。可以设置不同的打印机类型、纸张的大小、走纸方向、打印份数等。

(2) 通用查询基类。可以按任意条件组合查询数据库中的数据，并提供排序、对数值求和、打印查询结果等功能。其功能示意如图 12-28 所示。

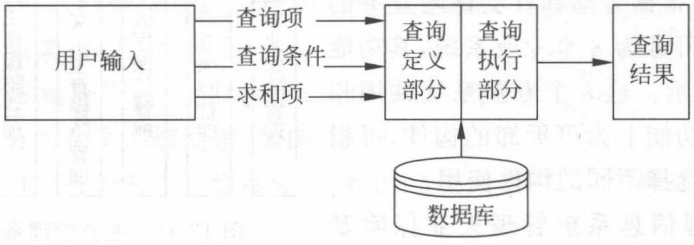


图 12-28　通用查询基类功能示意图

(3) 权限验证基类。用于控制操作人员对各功能模块的使用权限。

(4) 通用数据库连接基类。用于不同的用户登录时连接到数据库并赋予不同的权限。

(5) 系统错误信息捕获基类。可以捕获系统错误信息并将之转换成相应的中文信息。

(6) 字符处理基类。主要完成一些特定字符的转换，如日期转换函数完成日期由数字转换成汉字；金额转换函数完成金额由数字转换成汉字。

(7) 码表维护基类。提供一个通用的插入、删除、更新、保存的模板用于所有编码的维护。

(8) 数据转换基类。以自动或手工方式将数据从一个数据库中转换到另一个数据库

中。为了采集原系统的数据,要定义许多数据转换基类。数据转换基类的逻辑结构如图 12-29 所示。

2. 基层单位管理平台

基层单位管理平台是 SIMIS 数据采集的重要来源,是参保单位的管理平台,包括劳资人事管理、工资管理、岗位管理和社会保险管理系统。基本应用层的功能分解如图 12-30 所示。

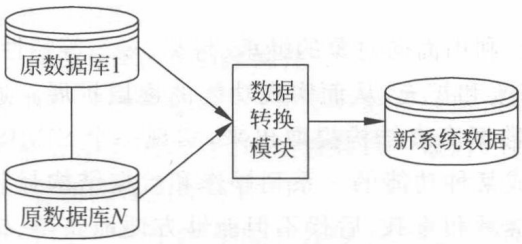


图 12-29　数据转换基类逻辑结构图

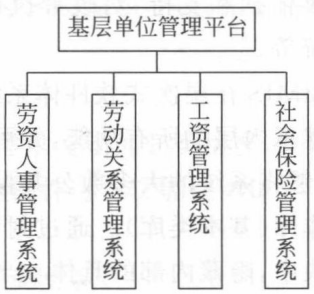


图 12-30　基层单位管理平台功能分解图

劳资人事管理系统主要管理本单位的人事关系;工资管理系统主要管理本单位职工工资的发放和统计;劳动关系管理系统主要管理本单位劳动合同等问题;社会保险管理信息系统是社会保险管理系统中的一个子集,是 SIMIS 数据采集的主要来源。

3. 业务管理系统

业务管理系统是对基本应用系统的进一步扩展,连同其内部的两层一起构成了 SIMIS 的典型应用层。

业务管理系统主要完成 SIMIS 的业务管理,管理内容涉及劳动者个人、企业和其他劳动组织的微观信息。根据劳动和社会保险业务的内容,典型应用层可分为 8 个专业系统,其功能分解如图 12-31 所示。这 8 个专业系统共用相同的基本信息,在功能上为可拆卸的构件,可根据具体应用需求,选择不同的构件使用。

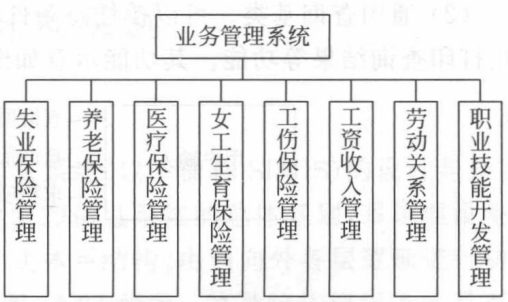


图 12-31　业务管理系统功能分解图

失业保险管理信息系统管理失业保险基金、失业职工信息等。同时,在一定的范围内,为领导者提供有力的决策参考。养老保险管理信息系统是用于城镇基本养老保险业务管理和服务的系统,是劳动和社会保险管理信息系统的重要组成部分。医疗保险是国家和社会在社会成员遇到疾病或非因工伤残风险时,为其提供医疗费用和社会服务,以保障恢复其健康的一种社会保障制度。女工生育保险是女性劳动者在从事人类自身在生产孕期、产褥期、哺乳期中参加社会劳动而丧失劳动收入时,从社会获得收入补偿以保障其顺利度过生育风险的一项社会保障制度。工伤保险是指职工因工作原因遭遇意外事故致伤、致残、致死或患职业性疾病,在暂时或永久丧失劳动能力的情况下,由社会给予其本人或遗属必要的物质帮助的社会保险制度。工资收入管理系统是宏观调控的一个重要手段,通过与工商、银行、税务等部门的联网,可有效地监控企业的工资管理,帮助企业建立自我约束机制。劳动关系管理系

统包括劳动合同管理、劳动争议处理、突发性事件处理、劳动监察、劳动关系宏观预测等方面的内容。职业技能开发管理系统包括职业技能鉴定、职业技能培训、择业指导等方面的内容。

4. 扩展应用层

扩展应用层包括统计信息管理系统、基金监测系统、决策支持系统和政策法规系统。扩展应用层的功能分解如图 12-32 所示。

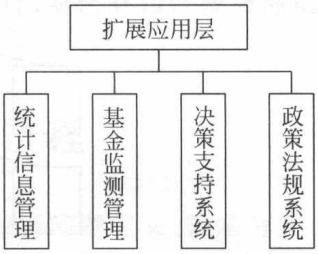


图 12-32　扩展应用层功能分解图

统计信息管理系统包括统计性数据的采集、整理、分析和发布，其信息来源于前台处理系统。基金监测系统对社会保险基金管理状况进行监控、调剂。决策支持系统利用已有的统计性数据、监测数据和政策参数，对基金调剂进行指导、对政策进行敏感性分析、对基金支撑能力进行中长期预测。政策法规系统输入和查询与劳动和社会保险有关的现有政策、法规。

在下面的讨论中，将以业务管理系统为例进行描述。

12.6.2　系统设计与实现

SIMIS 的业务管理系统的功能是完成劳动和社会保险的主要业务管理，即“五保合一”管理，包括养老保险、医疗保险、劳动就业和失业保险、工伤保险、女工生育保险。整个业务流程十分复杂，牵涉面相当广泛。例如，整个系统要与银行、企业、事业机关、医院、财政部门、税务部门、邮局等多种单位建立连接关系。虽然国家原劳动和社会保障部对整个业务有一套规定的指导性的流程，但是，在调研的过程中，发现各省、市都或多或少地存在使用“土政策”的情况，正是这种“土政策”致使软件在设计阶段具有很多的不确定性需求。另外，还考虑到将来用户需求可能会发生变化，为了尽量降低维护成本，提高可重用性，引入了正交软件体系结构的设计思想。

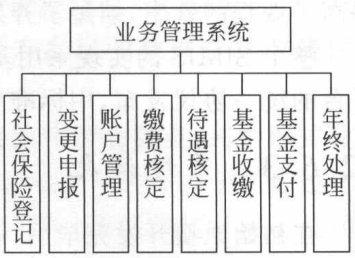


图 12-33　系统一级线索结构

在本系统的设计中，将整个系统设计为三级正交结构，第一级划分为 8 个线索，如图 12-33 所示。

每个一级线索可划分为若干二级线索。例如，一级线索“年终处理”可划分为 3 个二级线索，如图 12-34 所示。

每个二级线索可划分为若干三级线索。例如，二级线索“下年工作准备”可划分为 4 个三级线索，如图 12-35 所示。

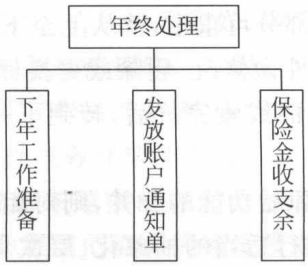


图 12-34　“年终处理”二级线索结构

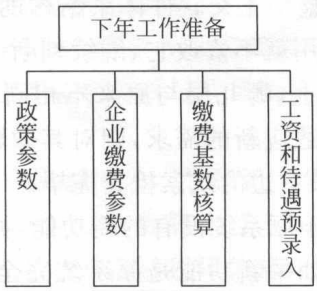


图 12-35　“下年工作准备”三级线索结构

所有线索是相互独立的,即不同线索中的构件之间没有相互调用。例如,“年终处理”线索不调用“变更申报”线索中的构件。这样一来,整个系统为三级正交线索,5个层次。其中的一条完整的线索结构如图12-36所示。

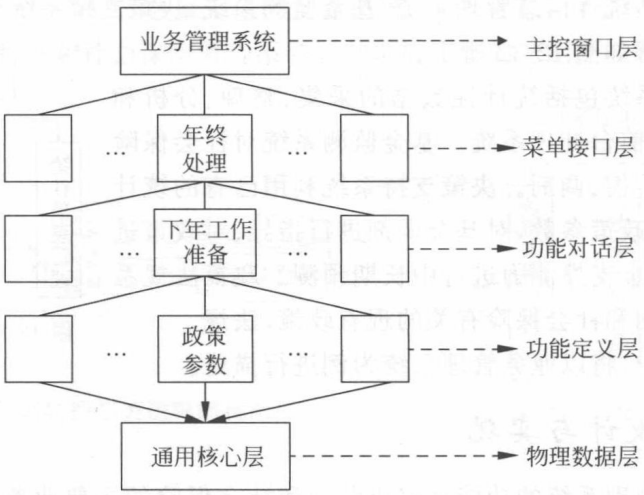


图12-36 完整的一条线索结构

在软件结构设计方案确定之后,开始正式的开发工作。由于采用了正交结构的思想,分成若干小组并行开发,视开发难度情况,每个小组负责一条或数条线索,由一个小组设计通用共享的数据存取构件。由于各条线索之间没有相互调用,所以各小组不会相互牵制,大大提高了编程的效率,缩短了开发周期,降低了工作量。

整个SIMIS的实现采用基于Internet的运行方式,应用由构件集成,通过标准语言、跨平台的统一协议发布,用标准用户界面显示,支持多种系统平台和数据库管理系统。

12.6.3 系统演化

在初始原型开发完毕后,就把SIMIS送给社保有关部门试运行,并征求意见,进行迭代过程开发。当SIMIS在一个单位正常运行后,又把该系统推广到其他单位,因为单位的级别不一样,即使是同一级别的单位,因具体情况不同,其管理流程和运转方式也不一样。因此,对不同的社保单位,就总有不同的需求,必须对软件进行修改。

在整个迭代开发和修改过程中,对于软件需求的变化,遵循以下步骤进行演化。

(1) 从整个正交软件体系结构的最左边一条线索开始,判断该线索是否可重用。如果可以完全重用就不修改它,继续判断下一条线索;如果部分可重用,就从上至下判断重用发生在哪几层上,哪几层与原来不相同,对需要变动的构件做修改、删除或更换标记;如果线索已完全不适应新的需求,则对其做删除标记。最左边的线索完成后,转到下一线索,以此类推,直到最右边的线索检查完毕。

(2) 对于原系统没有的新功能,如果新功能是多个原始功能的合并,则对相应线索做上合并标记;如果新功能是原系统完全没有的,则对其进行适当的抽象化、层次化和分级,建立一条新的线索。

(3) 更新构件的相互作用。随着构件的增加、删除和修改,必须更新构件之间的控制流。

(4) 形成新的体系结构。如果在以上步骤得到的体系结构还不够详细,不能实现改变的需求,就将第(1)～(3)步再迭代一次。最终形成新的正交软件体系结构,将这个体系结构作为改变的详细设计和实现的基础。

(5) 对以上步骤进行确认,进行阶段性技术评审。

(6) 对所做的标记进行处理。重新开发新线索中的所有构件,对已有构件按照标记的要求进行修改、删除或更换。完成一次演化过程。

12.7 基于体系结构的软件过程

传统的软件过程可以划分为从概念直到实现的若干阶段,包括问题定义、需求分析、软件设计、软件实现、软件测试等。传统的软件过程模型存在开发效率不高、可移植性差、重用粒度小等缺点。

本节综合利用软件过程理论、软件体系结构理论以及 Petri 网理论,讨论一个新的软件过程模型——基于体系结构的软件过程 Petri 网模型(ABSPN)。下面先学习有关概念。

12.7.1 有关概念

1. 软件过程

软件过程(software process)是人们建立、维护和演化软件产品整个过程中所有技术活动和管理活动的集合。目前,软件过程技术是一个非常活跃的研究领域,吸引了大批来自学术界和工业界的专家和学者。从 1984 年起,每年有软件过程国际研讨会(ISPW);从 1991 年起,开始召开软件过程国际会议(ICSP),几乎每个国家都有自己的软件过程改进网络(SPN)。软件过程技术的研究主要有 3 个方向。

(1) 软件过程分析和建模。软件过程建模方法是软件过程技术的起点,其中形式化半形式化建模方法有基于规则的、基于过程程序的等。过程分析和过程建模对于保证过程定义的质量、建立全面和灵活的过程体系具有重要的作用。对软件过程的建模主要是使用过程建模语言(Process Modeling Language,PML)。PML 最基本的功能是用于描述和定义过程,建立过程模型。PML 的能力和表达方式直接影响着过程模型的质量和建模效率。所以,选择合适的 PML,成为过程分析、过程建模和选择建模工具的关键。

(2) 软件过程支持。软件过程支持主要是指研究和开发支持软件过程活动的 CASE 工具,过程支撑工具作为一种技术基础设施能够很好地支持、管理并规范化软件过程。它的使用将使得软件过程的透明度好,为项目的软件过程提供指导,使得开发者和管理者都有据可依,便于更有效地管理软件过程。软件过程支持工具主要包括软件过程流程工具、过程文档工具、评审工具和人员管理工具。

(3) 软件过程评估和改进。软件过程改进对生产高质量软件产品和提高软件生产率的重要性已被越来越多的软件开发组织所认同。由美国卡耐基梅隆大学软件工程研究所(CMU/SEI)提出的能力成熟度模型集成(CMMI)除了用于软件过程评估外,还向软件组织提供了指导其进行软件过程管理和软件过程改进的框架。软件过程改进的基本原则是采用过去项目中成功的实践经验。因此,理解、记录和重用部分软件过程是软件过程改进研究的一个重要方向。

2. Petri 网

Petri 网的概念最早由德国的 C. A. Petri 博士于 1962 年提出,是一种用于系统描述和分析的数学工具。在此后的几十年中,Petri 网理论得到了极大的丰富,并被广泛地应用于许多研究领域,如协议工程、柔性制造系统、业务处理等。使用 Petri 网描述业务过程主要有以下几个原因。

(1) 形式化的语义。Petri 网具有严密的数学基础,为形式化描述和语法建立奠定了基础。每个 Petri 网都有形式化的语义定义,一个 Petri 网模型加上相应的语义就能够描述一个业务过程。

(2) 直观的图形表示。Petri 网是一种图形化语言。经典的 Petri 网有两种元素:变迁(用方框表示)和位置(用圆圈表示),而有向边表示这两种元素之间的关系。Petri 网的图形表示特点,使 Petri 网尽管具有严密抽象的数学表示,对用户来说却较容易理解,结构清晰。

(3) 丰富的分析技术。Petri 网模型一个很重要的特点在于它提供了丰富的系统分析技术,如对系统活性、有界性、安全性等分析计算。

(4) 基于状态的表示方式。一般软件工程领域的图形表示方法往往是基于事件的表示。Petri 网基于状态的描述能清晰地区分一个任务是处于授权状态还是处于执行状态。因此,Petri 网可以实现竞争任务。

下面来看一个简单的 Petri 网的例子。图 12-37 用 Petri 网描述了在一个多任务系统中的两个进程使用一个公共资源时,利用通信原语 LOCK(对资源加锁)和 UNLOCK(对资源解锁)控制资源的使用,保证进程间的同步的例子。

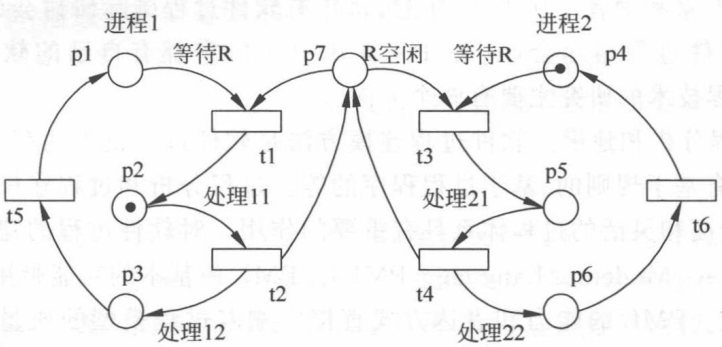


图 12-37 进程同步机制的 Petri 网描述

图 12-37 中每个进程是一个数据对象,它有 3 个状态:等待资源(p1 或 p4),占用资源执行的处理(p2 或 p5),不占用资源执行的处理(p3 或 p6)。另外,系统有一个状态:资源空闲(p7)。以进程 1 为例,如果公共资源 R 被进程 2 所用时,就进入等待状态 p1,如果资源可用,则进入状态 p2。资源利用完毕后,释放资源,使资源返回空闲状态 p7,而进程本身进入不占用资源执行的处理状态 p3。

有的状态中有一个黑点⊙,称为标记或令牌,表明系统或对象当前正处于此状态。在图 12-37 中,标记在 p2 和 p4 状态中,说明进程 1 正处于 p2 状态,而进程 2 正处于 p4 状态。

关于 Petri 网的基础知识,有兴趣的读者可阅读有关参考文献。应用到软件过程的描述中,用变迁(transition)表示定义的过程,用位置(place)表示过程进行的条件,位置中的标记(token)表示条件的真假。

12.7.2 软件过程网

本节对经典 Petri 网进行必要的扩展，使之在对软件过程进行建模时比经典 Petri 网更加清晰，同时避免复杂的扩展方式，使系统可以利用更多以往的 Petri 网分析工具。本节定义软件过程网及其相关特性。首先，给出 Petri 网系统的定义。

定义 1 设 $N=(P,T;F)$是一个 Petri 网，$PN=(N,M_0)$是一个 Petri 网系统。其中，

(1) P：有限位置集合。

(2) T：有限变迁集合，且 $P\cap T=\varnothing$，$P\cup T\neq\varnothing$。

(3) F：$F\subseteq(P\times T)\cup(T\times P)$，为连接位置和变迁的有向弧线，其中×为笛卡儿积。

(4) $\mathrm{dom}(F)\cup\mathrm{cod}(F)=P\cup T$，其中 $\mathrm{dom}(F)=\{x\mid\exists y:(x,y)\in F\}$，$\mathrm{cod}(F)=\{y\mid\exists x:(x,y)\in F\}$分别为 F 的定义域和值域。

(5) M_0 表示 N 的初始标识(Initial Marking)。

与经典 Petri 网类似，${}^*t$，t^*，${}^*p$，p^* 分别代表变迁和位置的前集(Pre-set)和后集(Post-set)。

定义 2 Petri 网系统 $PN=(N,M_0)$，其中 $N=(P,T;F)$，M 是 N 上的任一标识，R 是 M 的前向可达集。

(1) 变迁 $t\in T$ 是活的，当且仅当 $\forall M\in R(M_0)$，$\exists M'\in R(M)$，$M'[t>M$。

(2) PN 是活的，当且仅当 $\forall t\in T$ 是活的。

(3) PN 是有界的，当且仅当存在整数 k，$\forall p\in P$，$\forall M\in R(M_0)$，$M(p)\leqslant k$。

(4) 标识 M 是活的，当且仅当(N,M)是活的。

(5) N 是结构活的，当且仅当 $\exists M_0$，(N,M_0)是活的。

(6) N 是结构有界的，当且仅当 $\forall M_0$，(N,M_0)是有界的。

下面给出 Petri 网系统的路径和强连通的概念。

定义 3 Petri 网系统 $PN=(N,M_0)$，其中 $N=(P,T;F)$。在 N 中，称 C 为 n_1 到 n_k 的路径，是指存在一系列$(n_1,n_2,\cdots,n_k)$满足$(n_i,n_{i+1})\in F$，其中 $1\leqslant i\leqslant k-1$。

集合 $P_C=\{n_i,1\leqslant i\leqslant k\}\cap P$ 称为路径上的位置结点集。集合 $T_C=\{n_i,1\leqslant i\leqslant k\}\cap T$ 称为路径上的变迁结点集。

若 C 中的所有 $n_1,n_2,\cdots,n_k$ 都互不相同，则称 C 为简单路径。

在 N 中，如果从 n_i 到 n_j 存在一条路径，从 n_j 到 n_i 也存在一条路径，则称 n_i 和 n_j 是迭代的。

定义 4 Petri 网系统 $PN=(N,M_0)$，其中 $N=(P,T;F)$，$H\subseteq P$。H 在 N 中是强连通的，当且仅当对 $\forall n_i,n_j\in H$，在 H 中存在一条从 n_i 到 n_j 的路径。如果 $H=P$，则称 PN 是强连通的。

有了上面的准备工作之后，下面给出软件过程网的定义。

定义 5 一个 Petri 网系统 $PN=(N,M_0)$，其中 $N=(P,T;F)$，称为软件过程网，当且仅当：

(1) N 有两个特殊的位置 i 和 o，${}^*i=\phi;o^*=\phi$。N 有两个特殊的变迁 t_i 和 t_o，其中 ${}^*t_i=\{i\}\wedge i^*=\{t_i\}\wedge t_o{}^*=\{o\}\wedge{}^*o=\{t_o\}$。

(2) 如果在 N 上增加一个变迁 t_r 连接 i 和 o，即 ${}^*t_r=\{o\}$，$t_r^*=\{i\}$，用 N^e 表示，称为

N 的扩展网,则 N^e 为强连通的。

为了区别一般的 Petri 网系统,把软件过程网记作 SPN。在实际的软件过程网 SPN 中,某些变迁过程可能又包含许多子过程,其子过程的集合本身也组成一个 Petri 网,则把该 Petri 网称为 SPN 的下层 Petri 网。

定义 6 在软件过程网 SPN 中,所有层次变迁都被其代表的下层 Petri 网置换后,得到的 Petri 网系统称为完全展开软件过程网,记作 C_SPN。

定理 1 在软件过程网 SPN 中,对 $\forall n_i \in \{{}^*t\}$,在 n_i 和 t 之间存在一条路径;对 $\forall n_j \in \{t^*\}$,在 t 和 n_j 之间存在一条路径。

证明:只证明定理的前半部分,后半部分可类似地证明。

根据前集 $\{{}^*t\}$ 的定义,因为 $n_i \in \{{}^*t\}$,则由 n_i 开始,经过某些变迁和位置,必可到达 t。也就是说,存在一个由变迁和位置组成的序列 $(n_i, n_{i+1}, \cdots, n_{k-1}, n_k, t)$,满足 $(n_i, n_{i+1}) \in F \wedge (n_k, t) \in F$,其中 $i \leqslant k-1$。根据定义 3,可得在 n_i 和 t 之间存在一条路径。

定理 2 在软件过程网 SPN 中,对 $\forall n_i \in \{{}^*o\}$,在 n_i 和 o 之间都存在一条路径;对 $\forall n_j \in \{o^*\}$,在 o 和 n_j 之间也存在一条路径。

证明:与定理 1 的证明类似。

在对软件过程模型进行分析时,首先需要判定一个用 Petri 网描述的软件过程在过程上是否正确。下面给出软件过程网 SPN 过程正确性的形式化定义。

定义 7 软件过程网 SPN$=(N, M_0)$,其中 $N=(P, T; F)$,称为是过程基本正确的,当且仅当:

(1) 任何由初始状态可达的状态 M 均可到达终结状态,即:

$$(N, M_0)[*>(N, M') \Rightarrow (N, M')[*>(N, \{o\})$$

(2) 过程结束时,系统中只有位置 o 包含一个标记。

(3) 无死变迁。即

$$\forall t \in T, \exists M, M'(M_0[*>M[t>M')$$

定理 3 软件过程网 SPN 是过程基本正确的,当且仅当它的扩展网 SPN^e 是活的和有界的。

定理 3 的证明比较复杂,在此省略,有兴趣的读者可以阅读本章参考文献[6]。定理 3 的贡献是使得对 SPN 的过程正确性证明转换为对一般 Petri 网的特性分析。

定义 8 软件过程网 SPN$=(N, M_0)$,其中 $N=(P, T; F)$,称为是过程正确的,当且仅当其完全展开网 C_SPN 是过程基本正确的。

12.7.3 基本结构的表示

通常在 Petri 网的图形表示中,用圆圈(○)表示位置,矩形(▭)表示变迁,小黑点(•)表示标记。在软件过程的定义中,主要有顺序、与汇合、或汇合、与分支、或分支和迭代 6 种基本结构。下面分别用 Petri 网表示这 6 种基本结构。

(1) 顺序:表示过程 1、过程 2 和过程 3 顺序执行,其相应的 Petri 网表示如图 12-38 所示。

(2) 与汇合:表示该连接件左边的所有事件都完成后,右边的事件才可以发生。其相应的 Petri 网表示如图 12-39 所示。

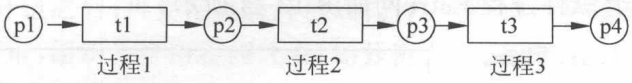


图 12-38　顺序过程

(3) 或汇合：表示该连接件左边的任一事件完成后，右边的事件就可以发生。其相应的 Petri 网表示如图 12-40 所示。

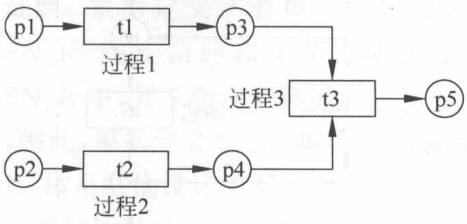


图 12-39　与汇合过程

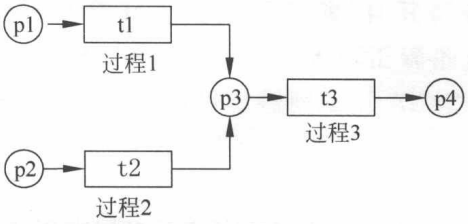


图 12-40　或汇合过程

(4) 与分支：表示该连接件左边的过程完成后，右边的过程均满足启动条件，即表示并行过程。其相应的 Petri 网表示如图 12-41 所示。

(5) 或分支：表示该连接件左边的过程完成后，右边的过程只有一个过程可以启动，在图 12-42 中，过程 2 或过程 3 在过程 1 之后发生，即表示在过程 2 和过程 3 之中选择一个进行。位置 p2 表示 t2 和 t3 的前条件，但是，t1 完成后，t2 和 t3 只有一个可以进行。即过程 2 和过程 3 竞争运行。此时，要求在 t1 的定义中给出选择的标准。

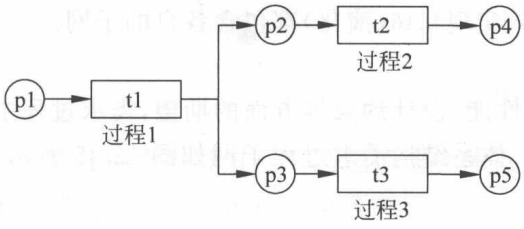


图 12-41　与分支过程

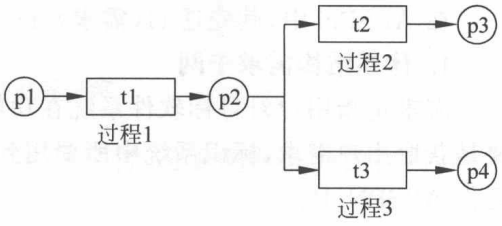


图 12-42　或分支过程

(6) 迭代：表示该连接件右边的过程完成后，又回到其左边的过程。其相应的 Petri 网表示如图 12-43 所示。在图 12-43 中，过程 3 在过程 2 之后发生，当过程 3 完成后，又回到过程 2，从而形成一个迭代过程。

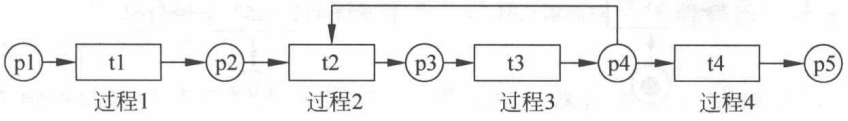


图 12-43　迭代过程

12.7.4　基于体系结构的软件过程 Petri 网

在本节中，将在 12.5 节的基础上讨论一个基于体系结构的软件过程 Petri 网模型。该模型把整个基于体系结构的软件过程划分为体系结构需求、设计、文档化、复审、实现、演化

和退役7个子过程。其软件过程Petri网如图12-44所示。

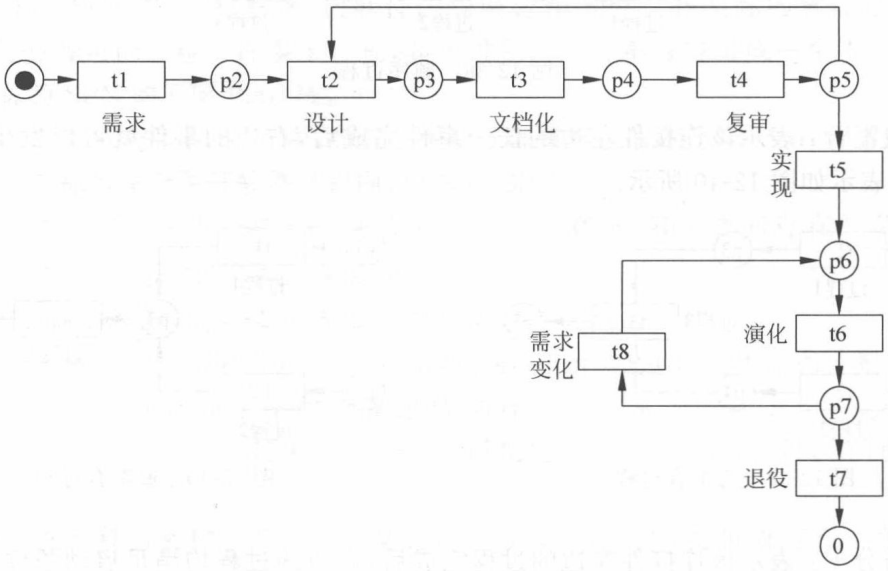


图12-44　基于体系结构的软件过程网

显然,图12-44满足定义5的要求,即图12-44是一个SPN,记作ABSPN。

定理4　ABSPN是活的和有界的;ABSPN是过程基本正确的。

证明:根据定义2和图12-44,ABSPN的活性和有界性是显然的,且其扩展网ABSPN^{e}也是活的和有界的。因此,根据定理3,ABSPN是过程基本正确的。

在ABSPN中,其变迁t1(需求)、t2(设计)、t5(实现)、t6(演化)都包含各自的子网。

1. 体系结构需求子网

需求是指用户对目标软件系统在功能、行为、性能、设计约束等方面的期望,需求过程主要是获取用户需求,标识系统中所要用到的构件。体系结构需求过程子网如图12-45所示,记作ABSPN_R。

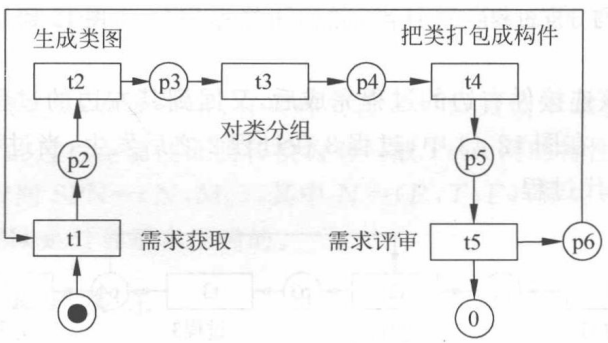


图12-45　体系结构需求过程子网

体系结构需求过程子网的变迁有需求获取、生成类图、对类分组、把类打包成构件和需求评审等。其中,需求获取变迁主要是定义开发人员必须实现的软件功能,使得用户能完成他们的任务,从而满足业务上的功能需求。与此同时,还要获得软件质量属性,满足一些非功能需求。获取了需求之后,就可以利用工具自动生成类图,然后对类进行分组,简化类图

结构，使之更清晰。分组之后，再要把类簇打包成构件，这些构件可以分组合并成更大的构件。最后进行需求评审，组织一个由不同代表（如分析人员、客户、设计人员、测试人员）组成的小组，对体系结构需求及相关构件进行仔细的审查。审查的主要内容包括所获取的需求是否真实反映了用户的要求，类的分组是否合理，构件合并是否合理等。必要时，可以在这些变迁之间进行迭代。

定理 5 ABSPN_R 是过程正确的，活的和有界的。

证明：根据定义 2 和图 12-45，ABSPN_R 的活性和有界性是显然的，且其扩展网 $ABSPN_R^e$ 也是活的和有界的，因此，根据定理 3，ABSPN_R 是过程基本正确的。在 ABSPN_R 中，其 5 个变迁都不再含有子过程，即它的完全展开软件过程网就是 ABSPN_R 本身，因此，根据定义 8，ABSPN_R 是过程正确的。

2. 体系结构设计子网

体系结构设计是一个迭代过程，如果要开发的系统能够从已有的系统中导出大部分，则可以使用已有系统的设计过程。软件体系设计过程子网如图 12-46 所示，记作 ABSPN_D。

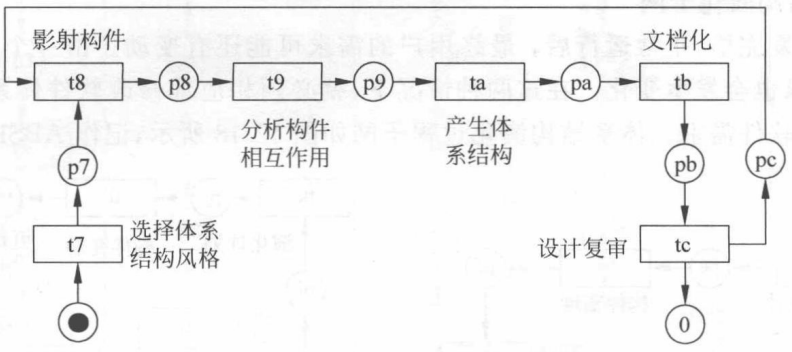


图 12-46 体系结构设计过程子网

软件体系设计过程子网包括选择体系结构风格、影射构件、分析构件之间的作用、产生体系结构、文档化、设计复审等变迁。

在建立体系结构的初期，选择一个合适的体系结构风格是首要的。选择了风格之后，把在体系结构需求阶段已标识的构件映射到体系结构中，将产生一个中间结构。然后，为了把所有已标识的构件集成到体系结构中，必须认真分析这些构件的相互作用和关系。一旦决定了关键的构件之间的关系和相互作用，就可以在前面得到的中间结构的基础上进行细化。

体系结构文档化过程的主要输出结果是体系结构需求规格说明和测试体系结构需求的质量设计说明书这两个文档。

在一个主版本的软件体系结构分析之后，要安排一次由外部人员（用户代表和领域专家）参加的复审。复审的目的是标识潜在的风险，及早发现体系结构设计中的缺陷和错误，包括体系结构能否满足需求、质量需求是否在设计中得到体现、层次是否清晰、构件的划分是否合理、文档表达是否明确、构件的设计是否满足功能与性能的要求等。如果复审通过，则进入下一过程，否则，在本过程的几个变迁中进行迭代。

定理 6 ABSPN_D 是过程正确的，活的和有界的。

定理 6 的证明类似于定理 5，此略。

3. 体系结构实现子网

实现就是要用实体来显示出一个软件体系结构,即要符合体系结构所描述的结构性设计决策,分割成规定的构件,按规定方式互相交互。体系结构的实现过程如图 12-47 所示,记作 ABSPN_I。

体系结构实现子网包含分析与设计、构件实现、构件组装和系统测试 4 个变迁。

整个实现过程以复审后的文档化的体系结构说明书为基础,每个构件必须满足软件体系结构中说明的对其他构件的约束。因为在体系结构层次上,构件接口约束对外唯一地代表了构件,所以可以从构件库中查找符合接口约束的构件,必要时开发新的满足要求的构件。然后,按照设计提供的结构,通过组装支持工具把这些构件的实现体组装起来,完成整个软件系统的连接与合成。最后一步是测试,包括单个构件的功能性测试和被组装应用的整体功能和性能测试。

定理 7 ABSPN_I 是过程正确的,活的和有界的。

定理 7 的证明类似于定理 5,此略。

4. 体系结构演化子网

在软件开发完毕,正常运行后,最终用户的需求可能还有变动。由一个单位移植到另一个单位,需求也会发生变化。在这两种情况下,就必须相应地修改软件体系结构,以适应新的变化了的软件需求。体系结构演化过程子网如图 12-48 所示,记作 ABSPN_E。

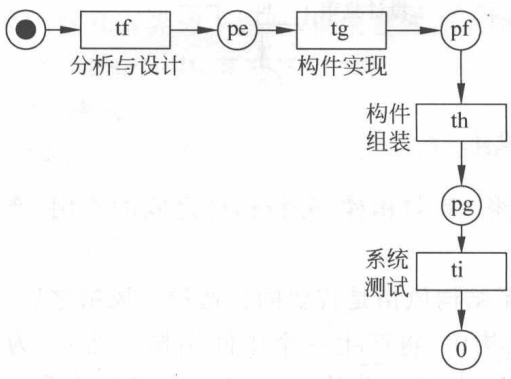


图 12-47 体系结构实现过程子网

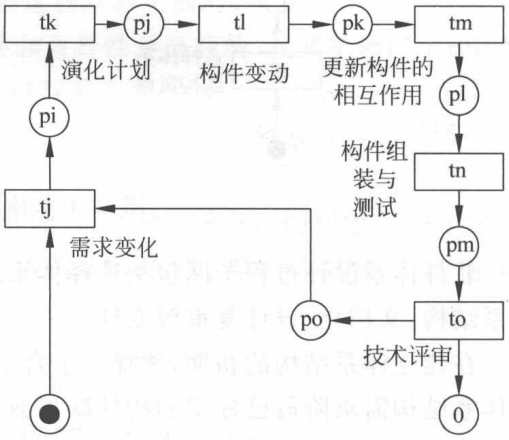


图 12-48 体系结构演化过程子网

体系结构演化子网包括需求变化、演化计划、构件变动、更新构件的相互作用、构件组装与测试和技术评审 6 个变迁。

首先,对用户需求的变化进行归类,使变化的需求与已有构件对应。对找不到对应构件的变动,也要做好标记。在后续工作中,将创建新的构件,以对应这部分变化的需求。在改变原有结构之前,开发组织必须制订一个周密的体系结构演化计划,作为后续演化开发工作的指南。在演化计划的基础上,开发人员可根据需求变动的归类情况,决定是否修改或删除存在的构件、增加新构件。随着构件的增加、删除和修改,构件之间的控制流必须得到更新。

然后,通过组装支持工具把这些构件的实现体组装起来,完成整个软件系统的连接与合成,形成新的体系结构。对组装后的系统整体功能和性能进行测试。

最后，对以上步骤进行确认，进行技术评审。评审组装后的体系结构是否反映需求变动，符合用户需求。如果不符合，则需要在以上各变迁之间进行迭代。

定理 8 ABSPN_E 是过程正确的，活的和有界的。

定理 8 的证明类似于定理 5，此略。

5. 完全展开网

根据以上对基于体系结构的软件过程网的变迁 t1、t2、t5 和 t6 的子网分析，可以得到 ABSPN 的完全展开网，如图 12-49 所示，记作 C_ABSPN。

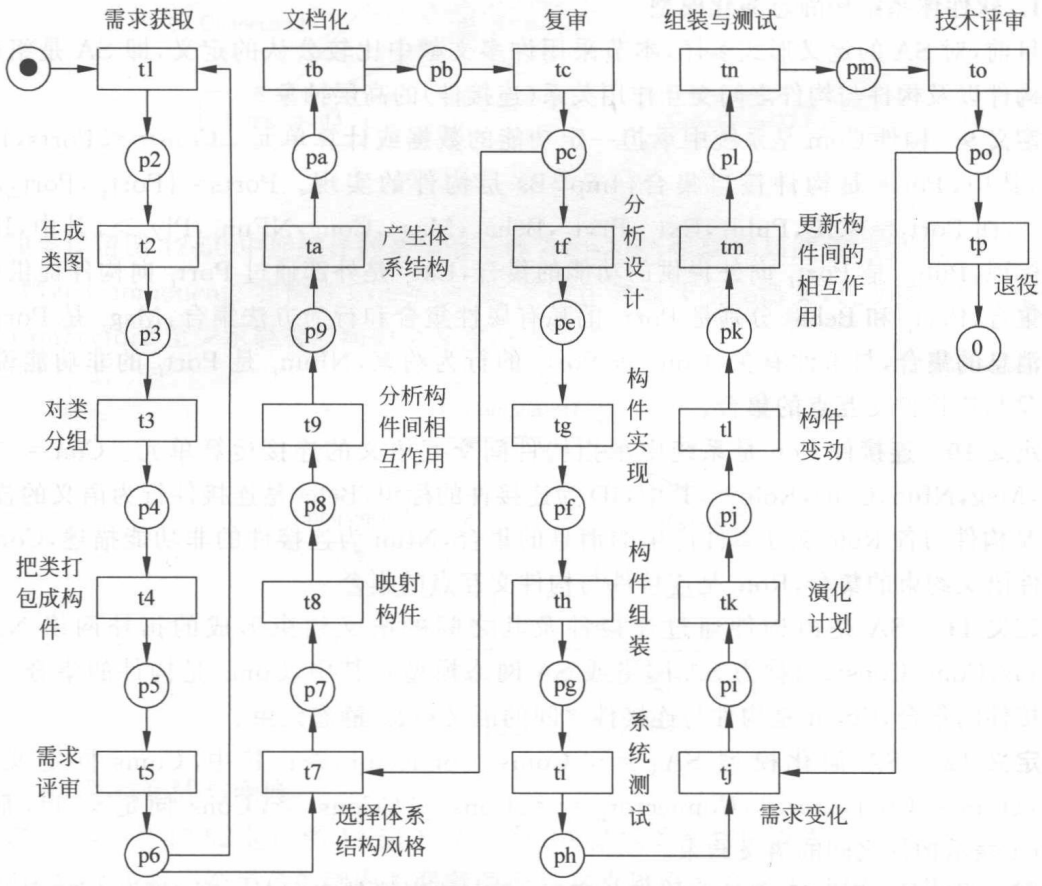


图 12-49 基于体系结构的软件过程网

定理 9 ABSPN 是过程正确的。

证明：根据定义 8，需要证明 C_ABSPN 是基本正确的；再根据定理 3，需要证明 C_ABSPN^e 是活的和有界的；而根据图 12-49，这是显然的。因此，ABSPN 是过程正确的。

12.8 软件体系结构演化模型

构造性和演化性是软件的两个基本特性。软件进行渐变并达到所希望的形态就是软件演化，软件演化由一系列复杂的变化活动组成。对软件变化的控制是软件开发者历来追求的目标。引起软件变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。所以，对软件变化甚至演化进行理解和控制显得比较复

杂和困难。

软件体系结构(SA)是软件生命周期的早期产品,着重解决软件系统结构和需求向实现平坦过渡的问题,是软件生命周期中开发、集成、测试和维护更改的基础;此外,在生命周期早期基于SA进行软件检测和修改,代价会相对低些。显而易见,要刻画复杂的软件演化,并对演化中的影响效应进行观察和控制,自然应从SA演化研究开始。

12.8.1 SA静态演化模型

1. 软件体系结构静态演化模型

目前,对SA的定义形式多样,本节采用许多文献中比较公认的定义,即SA是组成系统的构件以及构件与构件之间交互作用关系(连接件)的高层抽象。

定义9 构件Com是系统中承担一定功能的数据或计算单元。Com=<Ports,Imp_Bs>,其中,Ports是构件接口集合,Imp_Bs是构件的实现。Ports={$Port_1$,$Port_2$,…,$Port_n$},而$Port_i$=<ID,$Publ_i$,Ext_i,$Prvt_i$,$Beha_i$,Msg_i,Con_i,$NFun_i$,Ply_i>,其中,ID为构件标识,$Publ_i$是$Port_i$向外提供的功能的集合,Ext_i是外部通过$Port_i$向构件提供的功能的集合,$Prvt_i$和$Beha_i$分别是$Port_i$的私有属性集合和行为方法集合,Msg_i是$Port_i$产生的消息的集合,与事件有关,Con_i是$Port_i$的行为约束,$NFun_i$是$Port_i$的非功能说明,Ply_i是与连接件交互点的集合。

定义10 连接件Con是系统中承担构件间交互语义的连接运算单元。Con=<ID,Beha,Msg,Nfun,Cons,Role>,其中,ID为连接件的标识,Beha是连接件行为语义的描述,Msg是构件与各Role交互事件产生的消息的集合,Nfun为连接件的非功能描述,Cons是连接件语义约束的集合,Role是连接件与构件交互点的集合。

定义11 SA是由构件通过连接件及其之间的语义约束形成的拓扑网络N_{SA}=<Coms,Cons,Const>(称为SA网络或SA网络模型),其中,Coms是构件的集合,Cons是连接件的集合,Const是构件与连接件之间的语义动态、静态约束。

定义12 SA简化模型SA_S=<Coms,Connectors>;其中,Coms同定义11;Connectors={$Connector_{lr}$},$Connector_{lr}$=<Cons,C-CConst>,Cons同定义11,而C-CConst表示构件之间的语义约束。

要说明的是,定义11中的连接件和定义12中的连接件之间是一种"展开"和"合并"的关系,分别适应于本节中对SA动态演化和静态演化的研究。下面不再做特别说明。

定义13 由于系统需求、技术、环境和分布等因素的变化而最终导致的SA按照一定的目标形态的变动,称为SA演化。

就单一软件来说,SA演化被划分为静态演化和动态演化两个方面。对SA在非运行时刻的修改和变更称为SA的静态演化(如软件版本的升级等),而软件在运行时刻的SA变换称为SA的动态演化。本节从静态演化和动态演化两个方面建立其刻画模型,并对SA的演化进行统一描述。

为了描述的方便,根据定义12先给出一个实例。假设一个系统的SA由5个构件和6个连接件构成,其交互关系模型如图12-50所示。其中,构件Component1通过连接件Connector1、Connector3、Connector5、Connector6分别与构件Component2、Component3、Component4和Component5发生交互关系;其他构件之间的关系依然。

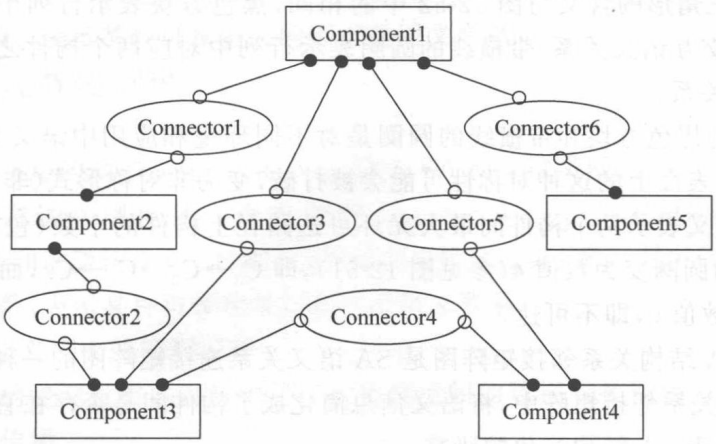


图 12-50　SA 模型示例

如果保留图 12-50 中连接件的基本语义即方向语义，则得到如图 12-51 所示的有向图。其中，构件 Component2 和构件 Component3 之间存在双向连接语义，说明图 12-50 中的连接件 Connector2 至少承载着双向的交互方向语义关系。

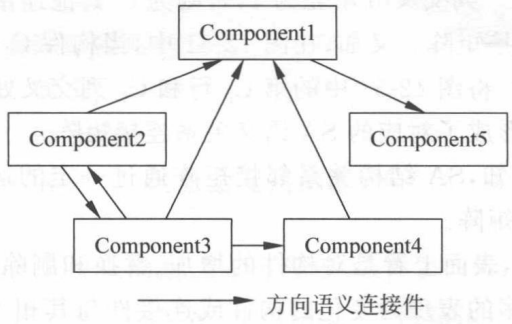


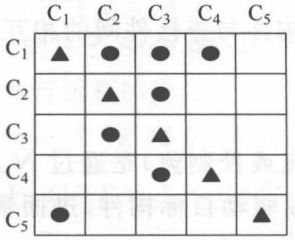
图 12-51　图 12-50 的简化模型

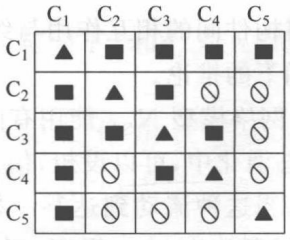
2. SA 邻接矩阵与变换

定义 14　图 12-52 是由图 12-51 构造出的 SA 结构关系邻接矩阵图。

图 12-52 中每一个黑色的圆圈关联着两个具有直接交互方向关系构件，其方向性由行和列分别表示，行为被关联方向的末尾构件，列为被关联方向的起始构件。另外，图 12-52 中的三角形表示构件的自交互关系，这种关系隐藏在构件自身内部。本节并不关心构件自身内部的交互关系。

定义 15　图 12-53 是由图 12-51 构造出的 SA 语义关系连接矩阵图。





C_1:Component1;C_2:Component2;C_3:Component3;C_4:Component4;C_5:Component

图 12-52　图 12-51 的 SA 关系邻接矩阵　　图 12-53　图 12-51 的 SA 语义关系连接矩阵图

图 12-53 中三角形的含义与图 12-52 中的相同,黑色方块表示行列中对应两个构件之间存在着直接的交互语义关系,带横线的圆圈表示行列中对应两个构件之间不存在或存在间接的交互语义关系。

图 12-53 中的黑色方块和带横线的圆圈是对不同环境和应用中语义形式的抽象,在具体化时,矩阵图在表面上的这种对称性可能会被打破,变为非对称形式(非对称矩阵图)。例如,当连接件的语义表示两个构件间最大无环可达路径上构件的个数(包括两端)时,列 C_3 和行 C_5 交叉处的圆圈变为数值 4(参见图 12-51),即 $C_3 \rightarrow C_2 \rightarrow C_1 \rightarrow C_5$,而列 C_5 与行 C_3 交叉处的圆圈变为数值 0,即不可达。

由此可见,SA 结构关系邻接矩阵图是 SA 语义关系连接矩阵图的一种简单形式。也就是说,在 SA 结构关系邻接矩阵中,将语义信息简化成了构件间是否存在直接交互以及交互的方向语义,有利于 SA 结构演化的研究。

定义 16 给 SA 结构关系邻接矩阵图和 SA 语义关系连接矩阵图中的十字交叉处填充上具有一定语义的数值后,形成的对应矩阵分别称为 SA 结构关系邻接矩阵和 SA 语义关系连接矩阵。SA 结构关系邻接矩阵简称 SA 邻接矩阵或邻接矩阵。

例如,在图 12-51 中,当构件 C_i 到构件 C_j ($i,j=1,2,3,4,5$)存在一条直接的连接时,图 12-52 中的第 C_i 行和 C_j 列交叉处填充为 1,否则为 0,其他连接以此类推。这样,就形成了对应的 SA 结构关系邻接矩阵。又如,在图 12-51 中,当构件 C_i 到构件 C_j ($i,j=1,2,3,4,5$)存在一条可达路径时,将图 12-53 中的第 C_i 行和 C_j 列交叉处填充为 1,否则为 0,其他连接以此类推。这样,就形成了对应的 SA 语义关系连接矩阵。

由图论的相关理论可知,SA 结构关系邻接矩阵通过一定的运算(如闭包运算)后可以转换为 SA 语义关系连接矩阵。

在 SA 的静态演化中,表面上看是对构件的增加、替换和删除,但这种变化蕴含着一系列的连带和波及效应,更多的表现为变化的构件或连接件与其相关联的构件或连接件的重新组合和归整。

12.8.2 SA 动态演化模型

SA 的动态演化比静态演化更为复杂。为了能把握其宏观动态性,仅依靠邻接矩阵和可达矩阵来刻画是不够的。下面结合基于动态语义网的润湿理论,对 SA 动态演化做进一步的描述。

1. SA 动态语义网

在 SA 的动态演化研究中,要求 SA 模型不仅具有刻画静态结构特性的能力,还应具有描述构件状态变化、构件间通过连接件的相互作用和局部网络的集群行为等动态特性的能力。因此,需要将构件间的相互作用与约束细化为构件与连接件间的相互作用与约束。由定义 11 可得出如下的推论。

推论 1 SA 网络模型 N_{SA} 能用有向图刻画。

在 SA 的动态演化中,可以设想,信息流(数据流或控制流)是通过 N_{SA} 中的连接件在构件之间流动的,当这种流动到达某一构件时,继续驱动目标构件,进而辐射出去或终止。这种反复出现的过程称为 N_{SA} 浸润。

定义 17 对被考察的系统 S 形式化后的动态语义网是一个有向图 $G_s = <V(G_s),$

$E(G_s),F_s>$,其中,$V(G_s)$是策动源;$E(G_s)$是浸润传播途径集;F_s是$V(G_s)$到$E(G_s)$的有序集合上的函数,表示某种可预定义的形式化语义联系。

定理 10 N_{SA}与G_s同构。

证明略。

定义 18 在 SA 的动态演化中,将N_{SA}称为 SA 动态语义网,记为G_{SA}。

可见,SA 动态语义网也是一个有向图$G_{SA}=<V(G_{SA}),E(G_{SA}),F_{SA}>$,其中,$V(G_{SA})$是策动源和被激构件集;$E(G_{SA})$是浸润传播途径集;$F_{SA}$是$V(G_{SA})$到$E(G_{SA})$的有序集合上的函数,表示某种可预定义的形式化语义联系。

2. SA 动态语义网中的浸润过程

动态语义网的浸润过程可以抽象为有向图结点间的数据或控制信息的驱动,即是一个由源结点策动的传播。

定义 19 浸润域是浸润发生过程的载体,它是一个有向图$G=<V(G),E(G),F>$,其中$V(G),E(G),F$含义类似于定义 17。

推论 2 在G_{SA}上同样可以定义与定义 19 含义完全相同的浸润域。

以下概念的描述是在浸润域G_{SA}上进行的,不再做特别说明。

定义 20 策动源$\text{Act_Nodes}^{(t)}=\{V_i|V_i\in V(G)\wedge S_i=1\wedge 0\leqslant i<|V(G)|\}$,即$t$时刻$V(G)$中的状态为 1 的所有结点构成的集合。$S_i$表示$V_i$的状态,定义为:

$$S_i=\begin{cases}1, F_i(x_1,x_2,\cdots,x_n)>\theta_i(x_1,x_2,\cdots,x_n)\\0,\text{其他}\end{cases}$$

其中,$F_i(x_1,x_2,\cdots,x_n)\in R$是$V_i$的激励变换函数;$\theta_i(x_1,x_2,\cdots,x_n)\in R$是$V_i$的门限函数。

可见$S_i\in\{0,1\}$,其中,0 表示抑制,1 表示活跃;$x_i(1\leqslant i\leqslant n)$表示结点$V_i$的输入。另外,如果定义$Y_i=F_i(x_1,x_2,\cdots,X_n)/S_i$作为结点$V_i$的输出;可见,当$S_i$抑制时$Y_i$是无意义的。

定义 21 图G的邻接矩阵$\boldsymbol{X}=(x_{ij})$,$0\leqslant i,j\leqslant|V(G)|$,其中$x_{ij}=\begin{cases}1, & S_i=1\wedge<i,j>\in F_G(E(G))\\0, & \text{其他}\end{cases}$;其中,$F_G(E(G))$表示图$G$对应的边集$E(G)$中对应的所有边的两端结点之间的某种可预定义的语义联系的集合。

为了有效地描述 SA 的动态演化,定义 21 对定义 16 做了进一步的细化。

t时刻图G的全部结点状态的集合表示为$S^{(t)}=\{S_i|V_i\in V(G),0\leqslant i\leqslant|V(G)|\}$。

定义 22 对整个矩阵$\boldsymbol{X}$的操作,等价于对矩阵$\boldsymbol{X}$中每个元素的操作。同样θ_i对整个矩阵$\boldsymbol{X}$的操作,等价于对矩阵$\boldsymbol{X}$中每个元素的操作。

定义 23 浸润算子 Soak 是在时刻t起作用的、限定F_i和θ_i对$X^{(t)}$进行一个离散时间步长的、逻辑上并行运算的操作。

定义 24 一个浸润步是指浸润算子 Soak 作用在三元组$<\bigcup_{vi\in\text{Act_Nodes}(t)}\{V_j|<i,j>\in F_G(V(E))\wedge F_i(X^{(t)},S^{(t)})>\theta(X^{(t)},S^{(t)})\}^{(t+1)},X^{(t+1)},t+1>$记为$\text{Soak_Step}^{(t+1)}$。

性质 1 $\text{Soak_Step}^{(t+1)}=\text{Soak}(\text{Soak_Step}^{(t)})$, $t=0,1,\cdots,N$。

定义 25 浸润过程是有限个时刻相继的浸润步的逻辑衔接。

定义 26 在浸润过程中,当 t 到达某一时刻 T 时,则称此时刻的点为不动点。

定义 27 存在不动点的浸润过程称为收敛的,否则就称为不收敛的。

定理 11 一个浸润过程收敛的充要条件是浸润中的每个结点 V_i 的 F_i 当 $t=T$ 足够大时收敛于 θ_i。

证明:略,有兴趣的读者请参阅本章参考文献[22]。

定义 28 在 G_{SA} 的浸润过程中,当 t 足够大到 T 时,如果所有结点的状态和对应的邻接矩阵的元素全不为 0,则称 SA 是最优的。

可见,在最优的 SA 中,各构成元素之间"紧密"合作,表现为内聚性较高。

3. SA 动态演化分析

SA 的动态演化分析比静态演化分析复杂,建立 SA 动态演化过程模型是动态分析的关键。

定义 29 SA 特定条件域 U_i 是指与 SA 相关的软件系统在一次运行中所对应的环境和约束条件集。

在 SA 特定条件域 U_i 下,SA 对应的软件系统在运行中数据和控制信息的流动总是在若干确定的构件和连接件之间传播,这若干确定的构件和连接件组成了新的 SA,称为 SA 特定条件域 U_i 下的 SA,记为 SA_i。

定义 30 SA 条件域 U 是指软件系统所有可能的 SA 特定条件域 U_i 的并集,即 $U=\bigcup_{i=1}^{N} U_i$,N 为 U 的非空子集的个数。

定义 31 $\forall U_i \in U, \exists SA_i \subseteq SA (i=1,2,\cdots,N, N$ 为 SA 条件域 U 的非空子集的个数),称为不动点软件体系结构 SA_i,简称不动点 SA_i。其中,$SA_i \subseteq SA$,$N_{SA}=<Coms, Cons, Const>$,$N_{SAi}=<Coms_i, Cons_i, Const_i>$,且 $Coms_i \subseteq Coms$,$Cons_i \subseteq Cons$,$Const_i \subseteq Const$。

定义 31 说明,当 SA 特定条件域确定之后,对应的 SA_i 也是确定的。从 G_{SA} 浸润的全过程来看,$SA_i \subseteq SA$ 相当于 SA 在此特定条件域下的一个稳态。

定义 32 在一个 SA 中,自一个不动点 SA_i 到另一个不动点 $SA_j (i,j=1,2,\cdots,N, N$ 为 SA 条件域 U 的非空子集的个数)之间的变换称为 SA 的不动点转移。

可见,软件运行环境条件组合的变化决定了 SA 的不动点转移,即 SA 条件域 U_i 的变化决定了 SA 的不动点转移。另外,SA 的动态演化过程可用在 G_{SA} 的浸润过程来刻画,而浸润步是 SA 动态演化的基本活动。

通过不动点的转移,SA 由一种形态变为另外一种形态,实际上是:软件在特定的环境条件下(包括运行的硬件环境输入和使用人员等),软件只有部分成分(构件和连接件)起作用。也就是说,数据或信息只在部分成分中流动,且决定了在这种特定环境下的局部的 SA,即不动点 $SA_i (i=1,2,\cdots,N, N$ 为 SA 条件域 U 的非空子集的个数)。

从不动点转移的角度来看一个软件,SA 静态演化实质上是 SA 动态演化的一个子过程。在 SA 动态演化研究中,为了宏观地把握演化的动态性,假定被研究系统的 SA 静态结构是确定的,且构成了稳定的网状通道,在系统运行时,信息经过这个网状通道传播,即信息的流动范围随着运行状态的变化而扩张或收缩,这种扩张或收缩反映了系统运行在 SA 特定条件域 U_i 中时 SA 的形态。

思 考 题

1. 什么是设计模式？在什么场合下需要使用设计模式？请结合实际项目举例说明应该如何使用设计模式？

2. 常用的设计模式包括哪些？根据你的了解列举你所知的设计模式。

3. 设计模式可以分为哪几类？请简要说明软件体系结构和设计模式的关系。

4. 什么是基于体系结构的设计方法？简要说明基于体系结构的设计方法的生命周期模型及设计步骤。

5. 结合一个实际项目，说明如何利用基于体系结构的软件开发模型进行系统设计和开发。

6. 简述软件体系结构演化模型概念，它是如何实现体系结构的动态演化的？

7. 根据已知信息，完成下面工程模式的设计。

纯虚类 FoodRestaurant 的定义(FoodRestaurant.cs)

```
#include "Food.h"
class FoodRestaurant
    {
        Public: FoodRestaurant();
        virtual ~ FoodRestaurant();
        virtual YuF * MakeYuF() = 0;
        virtual YaL * MakeYaL() = 0;
        virtual JiH * MakeYaH() = 0;
    };
```

各种食品类的定义(Food.h)如下。

```
class Food
    {
        Protected: Food()
        {
            m_strName = "Food";
        }
        string m_strName;
        public : string GetName();
    };

class YuF: public Food
    {
        public: YuF
        {
            m_strName = "YuF";
        }
    };
class YaL: public Food
```

```
    {
        public: YaL
        {
            m_strName = "YaL";
        }
    };

class JiH: public Food
    {
        public: JiH
        {
            m_strName = "JiH";
        }
    };

class KaoYa: public Food
    {
        public: KaoYa
        {
            m_strName = "KaoYa";
        }
    };

class JianYa: public Food
    {
        public: JianYa
        {
            m_strName = "JianYa";
        }
    };

class Malayu: public Food
    {
        public: Malayu
        {
            m_strName = "Malayu";
        }
    };
```

(1) 定义和实现餐厅类 A：餐厅 A 提供 Malayu、JianYa 和 KaoYa，通过从 FoodRestaurant 继承来的 Make 函数来选择需要的菜。

(2) 定义和实现餐厅类 B：餐厅 B 提供 KaoYa、JiH 和 YaL，同样通过从 FoodRestaurant 继承来的 Make 函数来选择需要的菜。

(3) 在 Main 函数中，定义一个 FoodRestaurant 类的指针(C++中抽象类可以拥有指针)，设为 pRest，用它来指向需要的餐厅 A 或者餐厅 B。通过 pRest 调用 Make 函数来选择需要的菜，例如，pRest 指向餐厅 A 时，pRest→Make Yu()表示需要的是 MalaYu。

8. 希赛公司承担了某企业应用系统的开发任务，用户要求系统最终应发布到 Web 上供企业员工和企业客户使用。项目组在进行方案论证时，首先肯定了该系统需使用 B/S 结

构，但在系统应采用的底层平台上产生了分歧，一方认为应采用微软.NET 平台，一方认为应采用 Java 企业版平台。经过认真讨论，结合两种平台的特点及项目的实际需求，项目组最终决定采用 Java 企业版平台作为系统开发运行的基础平台。

(1) 请给出.NET 平台与 Java 企业版平台各自具备的优势，以及两个平台共有的特点。

(2) 请分别针对基于 EJB 的重量级框架和基于 Struts 等的轻量级框架，说明 MVC 模式中的各组件应采用何种构件实现。

(3) 请从组件耦合度、组件分工及对开发工程化支持等 3 方面说明 MVP 模式与 MVC 模式的主要区别。

(4) 因为系统中大量业务逻辑涉及企业的核心商业数据，为保证系统数据一致性，完善的事务(Transaction)控制是系统实现时必须考虑的重要因素之一。请说明事务的基本特征，并简单描述 EJB 规范中提供的两种事务控制的基本方法。

参考文献

第13章 软件产品线体系结构

软件产品线(software product line)是一个十分适合专业的软件开发组织的软件开发方法,能有效地提高软件生产率和质量、缩短开发时间、降低总开发成本;它也是一个新兴的、多学科交叉的研究领域,研究内容和范围都相当广泛。

软件体系结构的开发是大型软件系统开发的关键环节。体系结构在软件产品线的开发中具有至关重要的作用,在这种开发生产中,基于同一个软件体系结构,可以创建具有不同功能的多个系统。在软件产品族之间共享体系结构和一组可重用的构件,可以降低开发和维护成本。

13.1 软件产品线的出现和发展

产品线的起源可以追溯到1976年Parnas对程序族的研究。软件产品线的实践早在20世纪80年代中期就出现了。最著名的例子是瑞士CelsiusTech公司的舰艇防御系统的开发,该公司从1986年开始使用软件产品线开发方法,使得整个系统中软件和硬件在总成本中所占比例之比从使用软件产品线方法之前的65∶35下降到使用后的20∶80,系统开发时间从近9年下降到不到3年。

虽然软件产业界已经在大量使用软件产品线开发方法,但是正式的对软件产品线的理论研究到20世纪90年代中期才出现,并且早期的研究主要以实例分析为主。到了20世纪90年代后期,软件产品线的研究已经成为软件工程领域最热门的研究领域。得益于丰富的实践和软件工程、软件体系结构、软件重用技术等坚实的理论基础,对软件产品线的研究发展十分迅速,目前软件产品线的发展已经趋向成熟。很多大学已经锁定了软件产品线作为一个研究领域,并有大学已经开设软件产品线相关的课程。

与软件体系结构的发展类似,软件产品线的发展也得益于军方的支持。如美国国防部支持的两个典型项目:关于基于特定领域软件体系结构的软件开发方法的研究项目(DSSA)和关于过程驱动、特定领域和基于重用的软件开发方法的研究项目(STARS)。这两个项目在软件体系结构和软件重用两个方面极大地推动了软件产品线的研究和发展。

可以说,软件产品线方法是软件工程领域中软件体系结构和软件重用技术发展的结果。下面从软件体系结构和软件重用两个侧面介绍软件产品线的发展。

13.1.1 软件体系结构的发展

当软件体系结构研究者和使用者发现面向独立系统的专有体系结构虽然可以满足应用的特殊需求却无法有效地支持重用,而通用的标准体系结构虽然可以用于各类应用但无法

满足应用的特殊需求时，DSSA 被发明出来了。DSSA 用一个特定的应用领域中可重用的参考体系结构有效地改善复杂软件系统的设计、分析、开发和维护。DSSA 的创建需要以该领域的需求分析和建模为基础，就相应地出现了对领域建模(domain modeling)的研究。DSSA 的出现给软件工程的另外一个研究分支——领域工程带来了重大的变化。

领域工程出现在 20 世纪 80 年代，可以把它看作软件工程在某个特定领域中的应用。领域工程将软件重用作为一个主要目标，不过在它刚出现时，软件重用还是“偶然性”的。20 世纪 90 年代初期 DSSA 出现后，领域工程便迅速与 DSSA 结合。领域工程中的软件重用成为系统化重用，它面向多个客户的多个应用系统，通过挖掘相似的或相关系统之间的共性来提供对跨应用的变化的支持。20 世纪 90 年代中后期，市场驱动的思想开始带入领域工程，领域专家采用经济技术通过对市场的分析，选择能满足最大比例客户需要的最小的需求和概念集合，并以该集合作为领域工程后续阶段的基础。这样，就出现了以软件产品线为核心的领域工程。现在，领域工程更多地被认为是软件产品线开发方法的一个重要组成部分。

图 13-1 给出了软件产品线在软件工程中的定位，以及软件产品线与相关研究领域如软件体系结构、特定领域软件体系结构、领域工程等的关系。因软件重用技术与软件工程重叠、与其他领域交叉，所以未在图中标出。

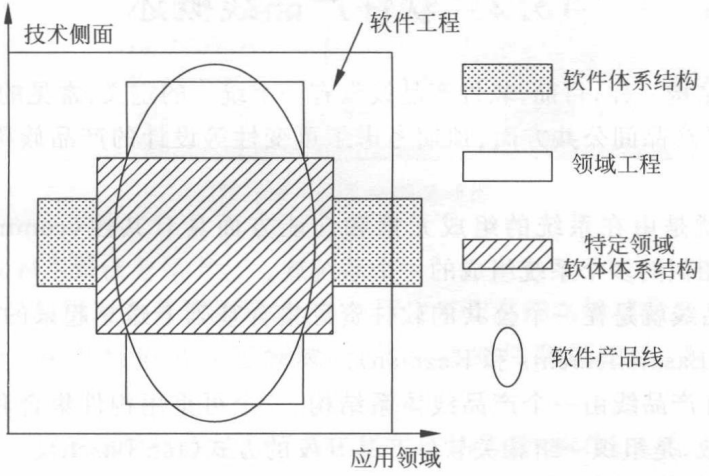


图 13-1 软件产品线在软件工程中的地位

13.1.2 软件重用的发展

随着软件规模和复杂度的增大，软件的开发和维护成本急剧上升。软件已经代替硬件成为影响系统成败的主要因素。为了解决面临的“软件危机”，软件开发者试图寻找一个将投资均摊到多个系统以降低成本的方法。软件重用是一个降低软件系统的平均成本的主要策略和技术。它的基本思想是尽最大可能重用已有的软件资源。

软件重用长期以来一直是软件工程界不断追求的目标。自 1968 年 Mcllroy 提出了软件重用概念的原型后，人们一直在尝试用不同的方法实现通过软件模块的组合来构造软件系统。软件重用也从代码重用到函数和模块的重用，再发展到对象和类的重用。当构件技术兴起时，曾经有人预测，基于构件的软件开发将分为构件开发者、应用开发者(构件用户)。但跨组织边界的构件重用是很困难的。但是对于一个软件开发组织来说，它总是在开发一

系列功能和结构相似的软件系统,有足够的经济动力驱使它对已开发的和将要开发的软件系统进行规划、重组,并尽量在这些系统中共用相同的软件资源。于是"世界范围内的重用"开始向"组织范围内的重用"转移。随着对软件体系结构的重要性的认识和软件体系结构的发展,基于构件技术的重用在软件重用中的主要地位就逐渐被基于软件产品线的重用代替。

基于产品线的软件重用也符合软件重用的发展趋势:从小粒度的重用(代码、对象重用)到构件重用,再发展到软件产品线的策略重用以及大粒度的部件(软件体系结构、体系结构框架、过程、测试用例、构件和产品规划)的重用,能使软件重用发挥更大的效益。到目前为止,软件产品线是最大程度的软件重用,可以有效地降低成本、缩短产品面世时间、提高软件质量。

虽然新的产品线技术和方法在不断涌现,但是软件体系结构和软件重用在引导产品线设计上的绝对重用性是不变的。软件产品线代表着跨产品的软件资源的大规模重用,并且是"有规划的"和"自顶向下"的重用,而不是在该领域已被证明了不成功的"偶然的"和"自底向上"的重用。作为指导软件产品线设计最重要的软件体系结构,产品线体系结构是重用规划的载体,是最有价值的可重用核心资源(core asset)。

13.2 软件产品线概述

与软件体系结构一样,目前,软件产品线没有一个统一的定义,常见的定义如下。

(1) 将利用了产品间公共方面、预期考虑了可变性等设计的产品族称为产品线(Weiss和Lai)。

(2) 产品线就是由在系统的组成元素和功能方面具有共性(commonalities)和个性(variabilities)的相似的多个系统组成的一个系统族。

(3) 软件产品线就是在一个公共的软件资源集合基础上建立起来的,共享同一个特性集合的系统集合(Bass,Clements和Kazman)。

(4) 一个软件产品线由一个产品线体系结构、一个可重用构件集合和一个源自共享资源的产品集合组成,是组织一组相关软件产品开发的方式(Jan Bosch)。

相对而言,卡耐基梅隆大学软件工程研究所(CMU/SEI)对产品线和软件产品线的定义,更能体现软件产品线的特征:

"产品线是一个产品集合,这些产品共享一个公共的、可管理的特征集,这个特征集能满足选定的市场或任务领域的特定需求。这些系统遵循一个预描述的方式,是在公共的核心资源(core assets)基础上开发的"。

根据SEI的定义,软件产品线主要由两部分组成:核心资源和产品集合。核心资源是领域工程的所有结果的集合,是产品线中产品构造的基础。也有组织将核心资源库称为"平台(Platform)"。核心资源必定包含产品线中所有产品共享的产品线体系结构,新设计开发的或者通过对现有系统的再工程得到的、需要在整个产品线中系统化重用的软件构件。与软件构件相关的测试计划、测试用例以及所有设计文档、需求说明书和领域模型还有领域范围的定义也是核心资源,采用COTS的构件也属于核心资源。产品线体系结构和构件是用于软件产品线中的产品构建的核心资源中最重要的部分。

软件产品线开发有4个基本技术特点:过程驱动、特定领域、技术支持和体系结构为中

心。与其他软件开发方法相比，软件开发组织选择软件产品线的宏观上的原因有：对产品线及其实现所需的专家知识领域的清楚界定，对产品线的长期远景进行了战略性规划。

13.2.1 软件产品线的过程模型

主要的软件产品线的过程模型有双生命周期模型、SEI 模型和三生命周期模型。

1. 双生命周期模型

最初的和最简单的软件产品线开发过程的双生命周期模型来自 STARS，分成两个重叠的生命周期：领域工程和应用工程。两个周期内部都分为分析、设计和实现 3 个阶段，如图 13-2 所示。

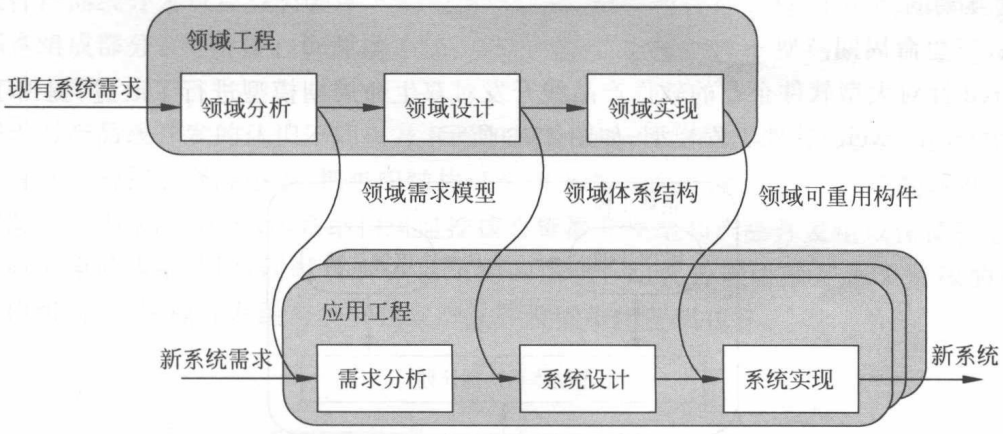


图 13-2 双生命周期模型

领域工程阶段的主要任务如下。

(1) 领域分析：利用现有系统的设计、体系结构和需求建立领域模型。

(2) 领域设计：用领域模型确定领域/产品线的共性和可变性，为产品线设计体系结构。

(3) 领域实现：基于领域体系结构开发领域可重用资源(构件、文档、代码生成器)。

应用工程在领域工程结果的基础上构造新产品。应用工程需要根据每个应用独特的需求，经过以下阶段，生成新产品。

(1) 需求分析：将系统需求与领域需求比较，划分成领域公共需求和独特需求两部分，得出系统说明书。

(2) 系统设计：在领域体系结构基础上，结合系统独特需求设计应用的软件体系结构。

(3) 系统实现：遵照应用体系结构，用领域可重用资源实现领域公共需求，用定制开发的构件满足系统独特需求，构建新的系统。

应用工程将产品线资源不能满足的需求返回给领域工程以检验是否将之合并入产品线的需求中。领域工程从应用工程中获得反馈或结合新产品的需求进入又一次周期性发展，称此为产品线的演化。

STARS 的双生命周期模型定义了典型的产品线开发过程的基本活动、各活动内容和结果以及产品线的演化方式。这种产品线方法综合了软件体系结构和软件重用的概念，在模型中定义了一个软件工程化的开发过程，目的是提高软件生产率、可靠性和质量，降低开

发成本,缩短开发时间。

2. SEI模型

SEI将产品线的基本活动分为3部分,分别是核心资源开发(即领域工程)、产品开发(即应用工程)和管理(详见13.4节)。其主要特点如下。

(1) 循环重复是产品线开发过程的特征,也是核心资源开发、产品线开发以及核心资源和产品之间协作的特征。

(2) 核心资源开发和产品开发没有先后之分。

(3) 管理活动协调整个产品线开发过程的各个活动,对产品线的成败负责。

(4) 核心资源开发和产品开发是两个互动的过程,3个活动和整个产品线开发之间也是双向互动的。

3. 三生命周期模型

Fred针对大型软件企业的软件产品线开发对双生命周期模型进行了改进,提出了三生命周期(tri-lifecycle)软件工程模型,如图13-3所示。

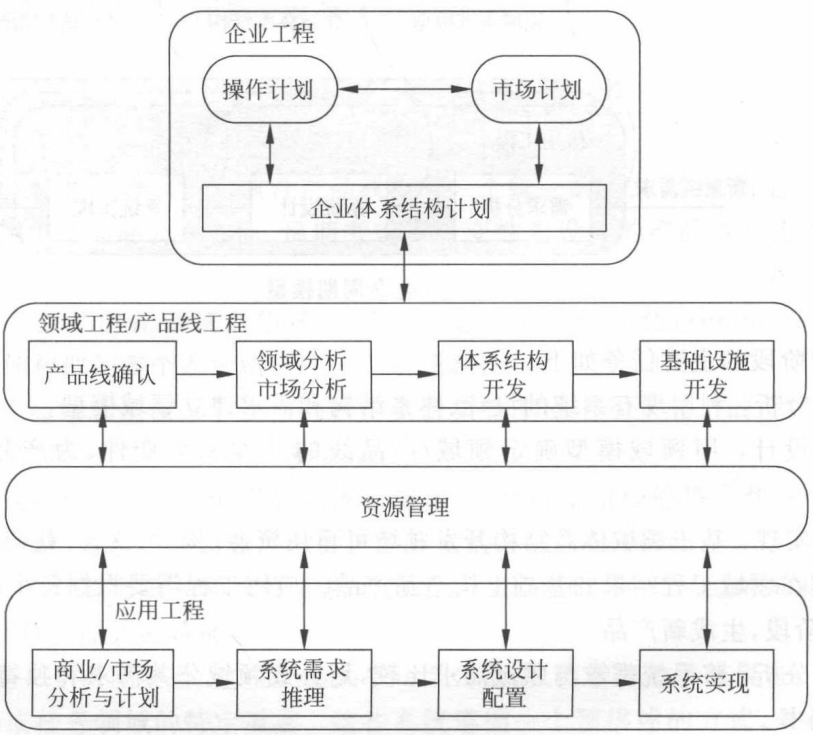


图13-3 产品线的三生命周期模型

为有多个产品线的大型企业增加企业工程(enterprise engineering)流程,以便在企业范围内对所有资源的创建、设计和重用提供合理规划。为了强调产品线工程在满足市场需求上与一般的系统化重用的区别,在领域工程中增加了产品线确定作为起始阶段,和领域分析阶段、体系结构开发阶段、基础资源开发阶段组成整个领域工程,还为领域分析阶段增加市场分析的任务;同样为应用领域增加了商业/市场分析与计划。在领域工程和应用工程之间的双向交互中添加核心资源管理作为桥梁,核心资源管理和领域工程、应用工程之间的支持和交互是双向的,以便于产品线核心资源的管理和演化。

以上描述的软件产品线开发过程并没有明确描述如何重用软件组织内遗留资源(legacy assets)。实际上大多数将要建立软件产品线的软件组织都积累有产品线所在领域的大量应用代码和相关文档,这些代码和文档中包含的知识对领域工程来说至关重要。Boeing 公司的 Margaret J. Davis 将软件再工程(reengineering)和产品线方法结合,该方法将软件再工程应用于领域工程中,用一种系统化的方法挖掘遗留系统中的知识。根据产品线和遗留系统采用技术的差异大小,能恢复(recovery)出的资源可能包括人员组织的交互和过程信息、软件体系结构和高层设计、算法代码和过程等。

13.2.2 软件产品线的组织结构

软件产品线开发过程分为领域工程和应用工程,相应的软件开发的组织结构也应该有两个基本组成部分:负责核心资源的小组、负责产品的小组。这也是产品线开发与独立系统开发的主要区别。

基于对产品线开发的认识不同以及开发组织背景不同,有很多组织结构方式。可以根据是否有独立的核心资源小组,把组织结构分为两大类。其中,设立独立小组的典型的组织结构如图 13-4 所示。其中,体系结构组监控核心资源开发组和产品开发组以保证核心资源和产品能够遵循体系结构,同时负责体系结构的演化;配置管理组维护每个资源的版本。体系结构组、核心资源开发组与负责独立产品开发的小组互相独立。

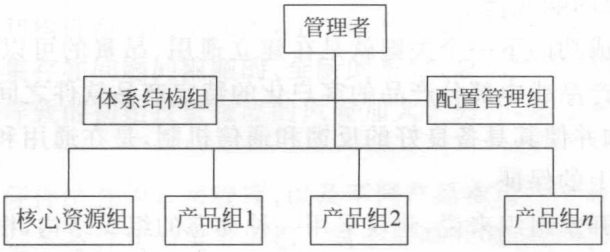


图 13-4 典型产品线开发组织结构

SEI 在其推荐的组织结构中强调市场人员在获取需求和推介产品中的作用。将产品线组织分为 4 个工作小组。

(1) 市场人员是产品线和产品能力、客户需求之间的沟通桥梁。

(2) 核心资源组负责体系结构和其他核心资源的开发。

(3) 应用组负责交付给客户的系统的开发。

(4) 管理者负责开发过程的协调、商务计划等。

SEI 还将客户提出的需求和对系统的反馈作为产品线组织的重要外部组织接口。

设有独立核心资源小组的组织结构通常适合于至少由 50～100 人组成的较大型的软件开发组织,设立独立的核心资源小组可以使小组成员将精力和时间集中在核心资源的认真的设计和开发上,得到更通用的资源。但独立的核心资源小组很容易迷失于建立极好的高度抽象、高度可重用的核心资源上,而忽视了这些资源对应用工程中需求的满足程度,因为这样的结构容易抑制应用工程中的反馈,使得所开发的核心资源无法在整个产品线中获得良好的应用。

另外一种典型的组织结构不设立独立的核心资源小组,核心资源的开发融入各系统开

发小组中,只是设立专人负责核心资源开发的管理。这种组织结构的重点不在核心资源的开发上,所以比较适合于组成产品线的产品共性相对较少、开发独立产品所需的工作量相对较大的情况。也是小型软件组织向软件产品线开发过渡时采用的一种方法。

Jan Bosch在研究了众多采用软件产品线开发方法的公司后,将软件产品线的组织结构归纳为4种组织模型。

(1) 开发部门(development unit):所有的软件开发集中在一个部门,每个人都可以承担领域工程和应用工程中适合的任务,简单、利于沟通,适用于不超过30人的组织。

(2) 业务部门(business unit):每个部门负责产品线中一个和多个相似的系统,共性资源由需要使用它的一个和几个部门协作开发,整个团队都可享用。资源更容易共享,适用于30～100人的组织,主要缺点是业务部门更注重自己的产品而将产品线的整体利益放在第二位。

(3) 领域工程部门(domain engineering unit):有一个专门的单位——领域工程部门负责核心资源库的开发和维护,其他业务部门使用这些核心资源来构建产品。这种结构可有效地降低通信的复杂度、保持资源的通用性,适于超过100人的组织。缺点是难以管理领域工程部门和不同产品工程部门之间的需求冲突和因此导致的开发周期延长。

(4) 层次领域工程部门(hierarchical domain engineering unit):对于非常巨大和复杂的产品线可以设立多层(一般为两层)领域工程部门,不同层部门服务的范围不同。这种模型趋向臃肿,对新需求的响应慢。

软件产品线开发成功的下一个关键就是在建立通用、昂贵的可以服务于所有产品的通用资源和开发服务于产品线中部分产品的客户化的特定产品软件之间的权衡。选择一个合理的、弹性的组织结构并使其具备良好的反馈和通信机制,是在通用和特定之间保持均衡的一个组织和基础机制上的保证。

对于中小型软件开发组织来说,建议采用一种动态的组织结构,根据产品线的建立方式和发展阶段、成熟程度的变化,由一种组织结构向另一种组织结构演变。这种方法的主要依据是在产品线不同发展阶段,领域工程和应用工程在总工作量中所占的比例是不同的。例如,对于从零开始建立的产品线,在其建立初期,核心资源的开发工作量要大大多于产品的开发。此时集中力量组织成专门的小组进行核心资源的开发,当核心资源基本完成时,可以将该小组部分成员逐步转移到产品开发中。而对于已有多个产品的情况下建立产品线的演变过程使用相反的方向更为合适。

这种动态的组织结构可以使中小型组织采用产品线开发方式造成的在人力资源上的压力得到缓解,使人力资源的需求在产品线的整个开发工程中趋于平稳。人员在两种小组之间的流动可以使流动人员作为小组之间信息交流的一种补充方式,虽然这不是一种最好的、合乎规范的信息交流方式,但毕竟也是一种快速有效的方式。组织结构的变化对产品线来说是一个很重要的问题,需要制订相应的变化规划并要有良好的管理技术的支持来保证整个产品线的成功。

13.2.3 软件产品线的建立方式

软件产品线的建立需要希望使用软件产品线方法的软件组织有意识的、明显的努力才有可能成功。软件产品线的建立通常有4种方式,其划分依据有两个。

(1) 该组织是用演化方式(evolutionary)还是革命方式(revolutionary)引入产品线开发过程。

(2) 是基于现有产品还是开发全新的产品线。

4 种方式的基本特征见表 13-1。下面对这几种方式进行简要分析。

表 13-1 软件产品线建立方式基本特征

基 础	演化方式	革命方式
基于现有产品集	基于现有产品体系结构开发产品线的体系结构。经演化现有构件,一次开发一个产品线构件	产品线核心资源的开发基于现有产品集的需求和可预测的、将来需求的超集
全新产品线	产品线核心资源随产品新成员的需求而演化	开发满足所有预期产品线成员的需求的产品线核心资源

1. 将现有产品演化为产品线

在基于现有产品体系结构设计的产品线体系结构的基础上,将特定产品的构件逐步地、越来越多地转化为产品线的共用构件,从基于产品的方法“慢慢地”转化为基于产品线的软件开发。主要优点是通过对投资回收期的分解、对现有系统演化的维持使产品线方法的实施风险降到了最低,但完成产品线核心资源的总周期和总投资都比使用革命方式要大。

2. 用软件产品线替代现有产品集

基本停止现有产品的开发,所有努力直接针对软件产品线的核心资源开发。遗留系统只有在符合体系结构和构件需求的情况下,才可以和新的构件协作。这种方法的目标是开发一个不受现有产品集存在问题的限制的、全新的平台、总周期和总投资较演化方法要少,但因重要需求的变化导致的初始投资报废的风险加大。另外,基于核心资源的第一个产品面世的时间将会推后。

现有产品集中软硬件结合的紧密程度,以及不同产品在硬件方面的需求的差异,也是产品线开发采用演化还是革命方式的决策依据。对于软硬件结合密切且硬件需求差异大的现有产品集因无法满足产品线方法对软硬件同步的需求,只能采用革命方式替代现有产品集。

3. 全新软件产品线的演化

当一个软件组织进入一个全新的领域,要开发该领域的一系列产品时,同样也有演化和革命两种方式。演化方式将每一个新产品的需求与产品线核心资源进行协调。好处是先期投资少,风险较小,第一个产品面世时间早。另外,因为是进入一个全新的领域,演化方法可以减少和简化因经验不足造成的初始阶段错误的修正代价。缺点是已有的产品线核心资源会影响新产品的需求协调,使成本加大。

4. 全新软件产品线的开发

设计师和工程师首先要得到产品线所有可能的需求,基于这个需求超集来设计和开发产品线核心资源。第一个产品将在产品线核心资源全部完成之后才开始构造。优点是一旦产品线核心资源完成后,新产品的开发速度将非常快,总成本也将减少。缺点是对新领域的需求很难做到全面和正确,使得核心资源不能像预期的那样支持新产品的开发。

13.2.4 软件产品线的演化

从整体来看,软件产品线的发展过程有 3 个阶段:开发阶段、配置分发阶段和演化阶段。

引起产品线体系结构演化的原因与引起任何其他系统演化的原因一样：产品线与技术变化的协调、现有问题的改正、新功能的增加、对现有功能的重组以允许更多的变化等。产品线的演化包括产品线核心资源的演化、产品的演化和产品的版本升级。这样在整个产品线就出现了：核心资源的新旧版本、产品的新旧版本、新产品等。它们之间的协调是产品线演化研究的主要问题。

在小型软件开发组织中，一般设立专门的人员和小组监控技术的变化和新产品的创建，并对产品线的体系结构和核心资源进行维护，因产品线中产品数量相对较少，产品线演化也就比较小；技术人员在产品线和产品族之间的流动也使同步的维护较容易。在大、中型软件开发组织中，则需要有更谨慎、更规范的做法。

如果不对核心资源进行更新以反映最新产品的需求变化，就只能在产品中创建相应资源的变体，而核心资源自此就不再适应这类需求，新产品和产品线体系结构、核心资源之间就产生“漂移”。这样发展下去，核心资源就逐渐失去了可重用性，维护成本加大，产品线的好处也失去了。

产品线演化同样造成问题。例如，需要开发产品的新版本时，作为基础的核心资源已经有了新版本。此时若仍然使用原来的老版本核心资源显然对产品的改动要少，成本要低，但以后会产生核心资源多版本维护问题以及该产品与核心资源之间的“漂移”问题。为了维护一致性，应该采用新版本核心资源。同样，对使用 COTS 产品进行软件产品线开发的组织来说，COTS 厂商总是不停地推出新版本的构件，令这些应用开发组织烦恼的是，是否、怎样和什么时候将这些新版本构件融入系统。

为此，笔者推荐使用以下方法：在开发新产品或产品的新版本时，使用核心资源的最新版本，已有的产品并不追随核心资源的演化。核心资源则要不断演化，以反映创建新产品和开发产品的新版本时反馈的需要核心资源调整配合才能满足的新需求。当然，也要防止对产品线体系结构和设计的过大、过早的演化，以免发生太多构件不做修改就无法使用的情况。保持产品的将来版本和产品线核心资源之间的同步是防止产品线退化的最基本要求。

13.3 框架和应用框架技术

随着软件技术的发展，软件重用已经从模块、对象的重用发展到了基于构件的重用和基于框架的重用，这也是当前最主要的两个软件重用的方式。从重用粒度上看，框架要比构件大。框架重用是一种面向领域的软件重用方式，更适合于软件产品线。框架一般建立在同一个或相似领域中，即所要开发的软件系统要具有较强的相似性，通过框架把领域中不变或易变的部分在一定时间间隔内固定下来，把易变的部分以用户接口的形式保留下来，从而达到设计和代码的重用。框架技术与构件技术的结合产生了基于构件的应用框架技术，这是框架技术的一个发展趋势。除此之外，本节讨论的框架主要指面向对象领域中的框架(object-oriented framework)。这是因为框架技术与面向对象技术关系十分密切，如框架的基础实现技术“动态绑定”(binding)就是由面向对象语言的多态机制支持的，并且很多具体的框架技术都是在面向对象环境中描述的。到 20 世纪 90 年代中期，框架的研究已经成为面向对象领域中的热点。在体系结构的设计实现和软件产品线开发中，框架作为一种基础

设计实现技术，也受到越来越多的重视。

1. 框架的定义

最早的对框架的描述由 Deutsch 在 1983 年给出："多个抽象类和它们相关算法的集合可组成一个框架，该框架在特定应用中可以通过专用代码的添加来将具体子类组织在一起运作。框架由抽象类及其实现的操作和对具体子类的期望组成"。

其他对框架的比较重要的定义和描述有以下几个。

(1) Johnson 和 Foot 在 1988 年给出的定义：框架是封装了特定应用族抽象设计的抽象类的集合，框架又是一个模板，关键的方法和其他细节在框架实例中实现。

(2) Buschmann：框架是一个可实例化的、部分完成的软件系统或子系统，定义了一组系统或子系统的体系结构并提供了构造系统的基本构造模块，还定义了对特殊功能实现所需要的调整方式。在一个面向对象的环境中，框架由抽象类和具体类组成；框架的实例化包括现有类的实例化和衍生。

(3) Johnson：框架＝模式＋构件。框架是由开发人员定制的应用系统的骨架(skeleton)，是整个系统或子系统的可重用设计，由一组抽象构件和构件实例间的交互方式组成。

以上是对一般框架概念的描述，软件产品线中的框架主要指的是应用框架。对应用框架的描述和定义主要有以下几个。

(1) Gamma：应用框架又称通用应用，是为一个特定应用领域的软件系统提供可重用结构的一组相互协作的类的集合。

(2) Buschmann：特定领域应用的框架被称为应用框架。

(3) Froehlich：应用框架就是某个领域公共(generic)问题的骨架式解决方案。框架为该领域所有应用提供公共的体系结构和功能基础。

(4) Batory：应用框架技术是用于应用产品线的、通用的、面向对象的代码结构化技术。一个框架就是表达抽象设计的抽象类的集合；框架实例就是为可执行子系统提供的抽象类的子集的具体类的集合。框架是为了重用而设计的；抽象类封装了公共代码，具体类封装特定实例的代码。

经过分析，可以得出以上众多对应用框架的描述的共同点，具体如下所述。

(1) 应用框架解决的是一个领域或产品族的问题，规定了问题应该如何分解。

(2) 包含应用或子系统的设计，由一个互相协作的类或构件集合组成。

(3) 可以通过继承或类的组合来创建应用。

2. 框架的特征

对框架技术的基本特征总结如下。

(1) 反向控制：类库是客户代码调用库中已存在类的方法。框架内嵌了控制流，框架调用客户代码——加入框架的新构件和抽象类的方法实例。

(2) 可重用性：框架提供了设计和代码的重用能力。

(3) 扩展性：为规划的变化提供了"热点"(hotspot)或"钩子"(hook)等显式说明方式。

(4) 模块化或构件化：框架有固定的、稳定的接口和封装的热点。

一般框架有 3 种建立方式：自顶向下，自底向上和混合方式。因为应用框架和软件产品线之间的密切关系，前两种框架建立方式与建立全新的软件产品线时的革命方式和演化

方式十分类似,也具有相同的过程和优缺点。混合方式指在大型应用框架的建立过程中,先将应用领域划分为不同的子区域,再分别解决,最终集成为一个完整框架的做法。

3. 框架的分类

根据框架的使用和扩展方式,可以将框架分为两大类:黑盒框架和白盒框架。

黑盒框架通过构件/类的组合来支持重用和扩展。应用中的类由框架的不同构件组合而成。在框架所在领域中,每个构件都有一个预定义的标准接口,一组共享相同接口但能满足不同应用需求的构件组成一个"插接兼容"的构件集合。

白盒框架一般使用类的继承机制实现,由未完成的类(抽象类)组成,类有一个或多个抽象接口或虚方法。应用需要在抽象类的继承子类中,提供特定意义的方法实例重用框架。开发者通过虚方法的实例化,将特定应用的代码联入框架来生成应用,所以虚方法又称"钩子"或"热点"。

白盒重用需要对框架有很好的理解,生成紧耦合系统。黑盒重用不需要对框架的内部结构有太多的了解,产生松耦合系统。具体的框架实际上都是"灰色"的,是可继承和可组合方式的结合。

灰色框架可以分成3部分:固定的、可选择的和开放的。框架的固定部分包含该领域最基本的功能、内建了应用的控制流,由框架主干实现,对应着领域共用部分。框架的可选择部分为该领域中相对固定的、应用特定的功能特征即领域个性部分,用可组合的类或构件实现,在应用构造时在这些构件或类中进行选择、组合。对一些无法准确估计和预测的功能特征即框架的开放部分,只能为其规定统一的接口和与框架的挂接点,用可继承的抽象类的方式来实现,这些部分可以根据应用的具体需求变化进行单独的调整。

与体系结构的层次结构类似,框架也可设计为层次结构,可称之为层次框架。例如,把一个完整的框架划分为:应用框架、领域框架和支撑框架多个层次,框架层次间是标准的或统一的接口。层次框架与层次体系结构具有相同的优点。

13.4 软件产品线基本活动

本节以SEI的软件产品线的过程模型为线索,讨论软件产品线开发的基本活动。

从本质上看,产品线开发包括核心资源库的开发和使用核心资源的产品开发,这两者都需要技术和组织的管理。核心资源的开发和产品开发可同时进行,也可交叉进行。例如,新产品的构建以核心资源库为基础,或者核心资源库可从已存在的系统中抽取。有时,把核心资源库的开发也称为领域工程,把产品开发称为应用工程。图13-5说明了产品线各基本活动之间的关系。

每个旋转环代表一个基本活动,3个环连接在一起,不停地运动着。3个基本活动交错连接,可以任何次序发生,且是高度重叠的。旋转的箭头表示不但核心资源库被用来开发产品,而且已存在的核心资源的修改甚至新的核心资源常常可以来自产品开发。

在核心资源和产品开发之间有一个强的反馈环,当新产品开发时,核心资源库就得到刷新。对核心资源的使用反过来又会促进核心资源的开发活动。另外,核心资源的价值通过使用它们的产品开发得到体现。

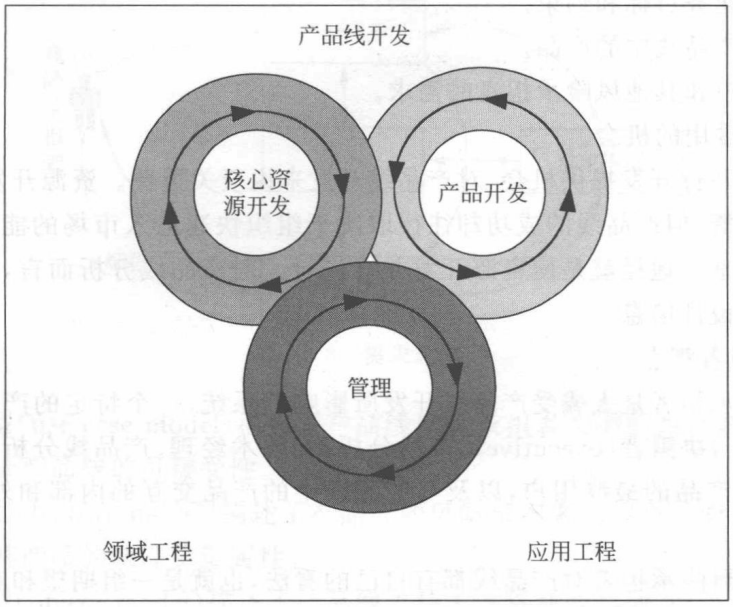


图 13-5　产品线基本活动

13.4.1　产品线分析

产品线分析是产品线的需求工程，是业务机遇的确认和产品线体系结构的设计之间的桥梁。产品线分析强调以下 3 个方面的问题。

(1) 通过捕获风险承担者的观点来揭示产品线需求。

(2) 通过系统的推理和分析、集成功能需求和非功能需求来完成产品线需求。

(3) 产品线设计师对产品线需求的可用性。

1. 上下文

产品线的开发包括资源开发、产品计划和产品开发几个步骤，产品线分析是资源开发的一部分，如图 13-6 所示。

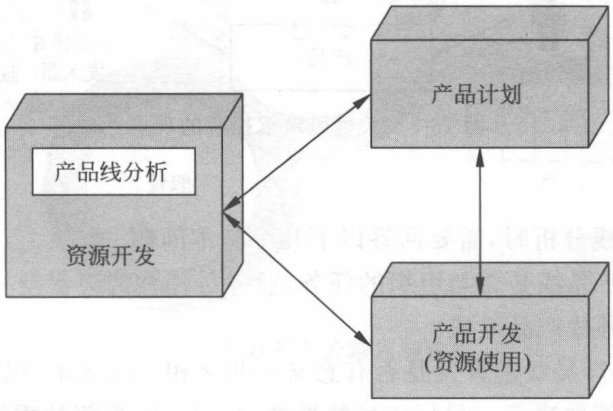


图 13-6　产品线分析

产品线分析是把对业务机遇的初步确认细化为需求模型，对正在开发的产品线而言，需要获取以下信息。

(1) 组织的业务目标和约束。

(2) 包含在产品线中的产品。

(3) 最终用户和其他风险承担者的需求。

(4) 大粒度重用的机会。

分析能否为并行开发提供机会,对产品线开发来说至关重要。资源开发需要固定投资,特别是及时的投资,但产品线的成功却往往取决于组织快速进入市场的能力。减少产品线进入市场时间的唯一途径就是使资源开发并行进行。对产品线分析而言,这意味着要尽可能快地发现重大设计信息。

2. 风险承担者观点

产品线风险承担者是人或受产品线开发所影响的系统,一个特定的产品线的风险承担者包括(但不限于)决策者(executive)、市场分析员、技术经理、产品线分析员、产品分析员、设计师、程序员、产品的最终用户,以及与产品线中的产品交互的内部和外部系统、政府机构、保险公司等。

每个产品线风险承担者对产品线都有自己的看法,也就是一组期望和对产品线的需求。因为许多风险承担者对产品线有同样的期望和需求,因此,只关注起关键作用的风险承担者。

对产品线开发来说,关键的风险承担者包括决策者、最终用户和产品线开发人员,如图 13-7 所示。决策者把产品线看作达到组织目标的机制,最终用户注重产品线中的特定产品所能提供的服务,产品线开发人员注重体系结构、产品计划和开发产品线中的产品所需的构件。

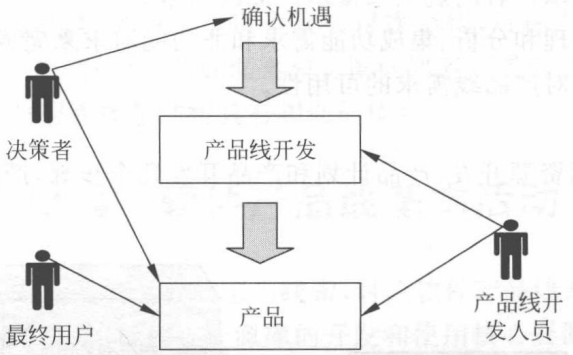


图 13-7 关键风险承担者的观点

3. 需求建模

在开始启动产品线分析时,需要回答以下几个基本问题。

(1) 将要开发的产品线是否与组织的任务、业务目标和约束保持一致?

(2) 产品线将由哪些产品组成?

(3) 对组织来说,产品线的开发是否有意义?与之相关的成本、风险和利润是什么?

对这些问题的回答取决于对目标市场特性的初步估计,期望的重用利益和诸如时间、经验、工具等资源的可用性。

产品线分析基于面向对象的分析、用例建模等。产品线需求模型是 4 个相互联系的工作产品的集合,如图 13-8 所示。

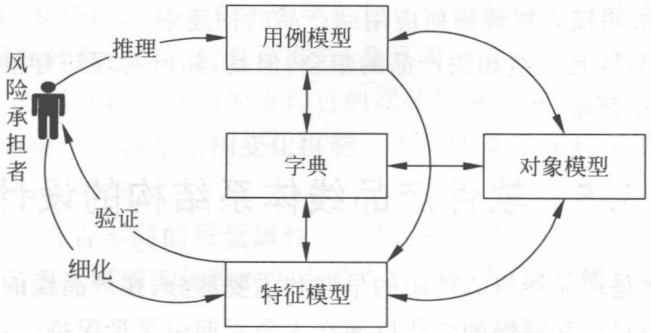


图 13-8　需求建模

(1) 用例模型(use case model)描述了产品线风险承担者和他们与产品线的关键交互，风险承担者将验证产品线的可接受性。

(2) 特征模型(feature model)描述了产品线的风险承担者的观点。它捕获产品的功能特征和产品线及其产品的软件质量属性。

(3) 对象模型(object model)描述了产品线支持上述特征的功能，以及这些功能的通用性和可变性。

(4) 字典(dictionary)定义了用在工作产品中的、支持产品线需求的一致观点的术语。

需求模型支持发现和文档化最终用户和其他风险承担者的期望和需求，提供影响产品线范围的早期和详细的信息，它是把风险承担者的需求映射为系列开发工作产品的基础，这种映射有利于决定和估计潜在的用户驱动(user-driven)的变更的影响。

13.4.2　产品开发

产品开发活动取决于产品线范围、核心资源库、产品计划和需求的输出，图 13-9 描述了它们之间的关系。

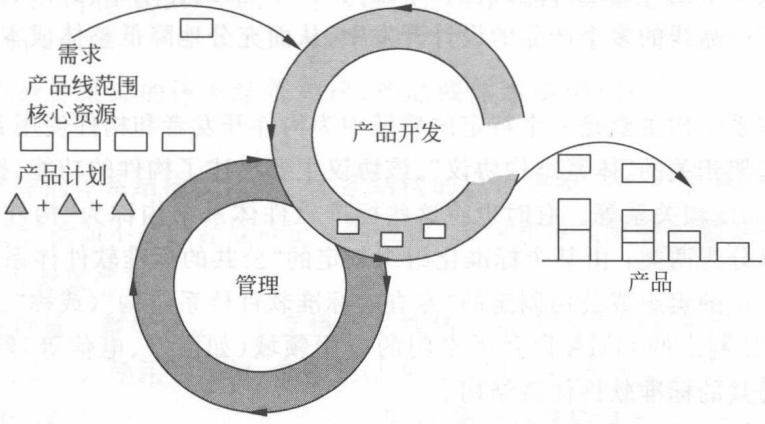


图 13-9　产品开发的输入与输出

产品开发的输入如下。

(1) 特定产品的需求，通常由包含在产品线范围内的一些产品描述来表达。

(2) 产品线范围，指明正在考虑的产品是否适合包含在产品线中。

(3) 构建产品所需的核心资源库。

(4) 产品计划,指明核心资源如何应用到产品的构建中。

从本质上说,产品线是一组相关产品的集合,但是,如何实现却有很大的不同,这取决于资源、产品计划和组织环境。

13.5 软件产品线体系结构的设计

产品线体系结构是产品线核心资源的早期和主要部分,在产品线的生命周期中,产品线体系结构应该保持相对小和缓慢的变化以便在生命周期中尽量保持一致。产品线体系结构要明确定义核心资源库中软件构件集合及其相关文档。

在各个产品的应用工程中,产品线体系结构被用来导出产品的体系结构。此时,如果发现有新变化点或者产品线体系结构不能满足的需求模式,需要将这些信息反馈给产品线体系结构设计师,由他们决定是否对产品线体系结构进行修改并实施。

13.5.1 产品线体系结构简介

软件体系结构设计的主要目的是满足对软件的质量需求。软件体系结构的应用方式有3种:用于独立软件系统、软件产品线体系结构和用于公共构件市场的标准软件体系结构。

独立软件系统的体系结构是常规软件开发周期的一部分,建立在独立软件系统的需求抽取和说明上,随后是详细设计、实现、测试等。

软件产品线体系结构指一个软件开发组织为一组相关应用或产品建立的公共体系结构。鉴于产品线软件开发在提高软件生产率和质量、缩短开发时间、降低总开发成本方面的重要作用,产品线体系结构又是软件产品线核心资源中的最主要部分之一。面向独立软件系统的软件体系结构设计方法并不完全适用于软件产品线体系结构的设计,因为它们一般没有考虑产品线中不同产品之间的共性和个性问题。产品线体系结构可以使软件开发组织将总成本均摊到产品线的多个产品的设计开发中,从而充分地降低整体成本、有效地提高软件生产率。

标准软件体系结构主要是一个特定的领域中为构件开发者和构件使用者之间提供一个与构件的基础框架相关的"体系结构协议",该协议主要描述了构件的功能、提供的和需要的接口、构件之间的依赖关系等。有时也将这些标准软件体系结构称为"构件框架"。标准软件体系结构可以分为两类:由某个标准化组织制定的"公共的标准软件体系结构"和由某个领域中占主导地位的组织或公司制定的"专有的标准软件体系结构"(或称"工业标准软件体系结构")。OMG制定的OMA以及为专门的应用领域(如医疗、电信等)制定的领域接口规范就是一种公共的标准软件体系结构。

所有的软件体系结构都是抽象的,它们都允许有多个实例。独立软件系统的软件体系结构对体系结构的变化没有说明和限制。在体系结构实例化过程中,为了满足目标系统的行为和质量目标需求,允许对体系结构进行任意的变化。软件产品线体系结构作为产品线中所有产品共享的体系结构和各个产品的体系结构的导出基础,必须在软件产品线体系结构中对允许进行的变化进行显式的说明和限定,最终的实例化结果才能既保持领域共性、又能满足特定产品的需求。

同领域模型一样，软件产品线体系结构中也可以分为共性部分和个性部分。共性部分是产品线中所有产品在体系结构上的共享部分，是不可改变的。个性部分指产品线体系结构可以变化的部分。产品线体系结构的设计目的就是尽量扩展产品线中所有产品的共性部分，同时提供一个尽量灵活的体系结构变化机制。产品线体系结构主要需考虑以下因产品线的特殊性而出现的变化需求。

(1) 产品线的产品有着不同的质量属性。例如，一个产品需要高度安全但运算速度要求低，另一产品可能需要运算速度快但对安全没有特别要求，产品线体系结构需要足够灵活以支持两个产品。

(2) 产品之间的差异可能体现在各个方面：行为、质量属性、平台和中间件技术、网络、物理配置、规模等。产品线体系结构需要对这些差异进行处理。

有多种技术支持体系结构的变化。例如，采用构造时(build-time)对构件、子系统的参数组合进行设置来适应产品线变化，但该方法假设所有的变化都是可预测的，并且所有变化在构件的代码中都要实现，每一组参数组合对应一个产品的实现。

在面向对象系统中可以用继承和动态绑定等面向对象技术将类设计为在不同的产品中能对变化点进行不同的实现。面向对象框架是这类技术的集中体现。也可以在变化点用构件替换来实现所希望的变化，这实际上是一种构件组合方式的变形。

Mikael Svahnberg 和 Jan Bosch 对软件产品线体系结构个性的实现机制总结如下。

(1) 继承：用于对象方法在产品中的不同实现和扩展。

(2) 扩展和扩展点：通过增加行为和功能扩展构件的某些部分。

(3) 参数化：用于构件的行为特征可以抽象并在构件构造时可确定的情况。例如，宏定义和模板都是参数化方法的一种。

(4) 配置和模块互连语言：用于定义系统构造时结构和构件的选择方式和结果。

(5) 自动生成：用更高级的语言来定义构件的特征，并自动生成相应的构件。

(6) 编译时(compile-time)不同实现的选择：用于构件的变化可以通过选择不同代码段实现的情况，如 #ifdef。

与用于独立软件系统的体系结构相比，产品线体系结构设计面临的主要困难和问题如下。

(1) 没有熟练的体系结构设计师：体系结构的设计还是一个不成熟的领域，设计更多地依靠设计师的经验而不是已经规范定义的规则、惯例和模式集合。尤其在某个特定的应用领域，该问题可能更严重。

(2) 参数化问题：参数化是一个支持产品线体系结构变化的有效的方法，但要注意过于参数化易使系统难以使用和理解，参数化过少又会限制系统的变化能力。过早的参数绑定易使变化困难，绑定过晚(运行时刻的动态绑定)易导致性能降低。

(3) 必须有良好的领域分析和产品规则基础作保证，对技术发展趋势要做出准确预测，还要注意吸取相关领域的教训。

(4) 软件开发、管理和市场人员的管理和文化对基于软件体系结构开发的适应程度。

(5) 目前，支持软件体系结构设计的 CASE 工具较少。

(6) 产品线体系结构设计师和产品开发者之间的沟通。

13.5.2 产品线体系结构的标准化和定制

软件体系结构是产品线所有可重用的核心资源中最重要的部分,是软件产品线成功的关键技术性资源。产品线体系结构将用于产品线中所有产品,需要使每个产品的体系结构都能符合它的行为特性、性能和其他质量属性需求。

产品线体系结构的设计有两种方式:使用标准体系结构和体系结构定制。作为标准,会有众多的软件开发组织遵循它,开发各自的应用或者为该体系结构提供基础构件和应用开发的辅助工具等。采用标准体系结构作为产品线的软件体系结构,可以获得第三方软件开发组织的支持、有效地缩短开发时间、提高产品的可靠性和与同类系统的可集成性等。所以如果产品线所在领域有相应的体系结构标准,应该尽量遵循它。

在宏观体系结构上,对标准的遵循比较容易,下面以层次体系结构为例进行说明

(1) 为适应应用的规模增大、复杂度提高,软件技术不断发展,相继出现了中间件技术、软件产品线等。应用的宏观体系结构自然地形成了"硬件-网络与操作系统-中间件平台-领域核心资源-应用"这样一个层次软件体系结构。

OMG 制定的软件体系结构标准 OMA 就是一个层次体系结构在面向对象环境中的演变,其层次为"ORB-公共服务对象-公共设施对象-领域对象-应用对象"。这个面向对象的层次结构和 OMA 的对象框架为产品线的宏观体系结构提供了很好的参考标准。另外,选用了 OMA 也就意味着选择了 CORBA 中间件平台,同时也获得了 OMG 相关标准和规范支持(如 UML、MOF、XMI、MDA 等)。对于这个层次结构的低层部分(如公共服务、公共设施等)的标准遵循比较容易:中间件技术已经成为当前软件开发的主流技术,几个主流的中间件平台在相关的公共服务标准上也出现融合的趋势。但该结构越是高层的部分,与特定领域和应用的相关性越大,在产品线体系结构设计中要遵循这些高层的标准就比较困难。因为 OMG 是以通用目的建立体系结构标准和领域接口规定的,面对的是整个软件应用领域,所以 OMG 划分的领域范围一般比较大。另外,标准的制定只能针对应用领域当前的普遍情况,对快速变化的需求有一个逐步调整的过程。而某个软件开发组织的软件产品线的范围的确定要考虑市场需求,该组织的技术、文化、管理背景,所在领域的现状和发展趋势等多方面,很难做到和某个标准组织定义的应用领域范围一致。

(2) 体系结构风格是一个使产品和产品线具有良好的可移植性的结构,产品和产品线通过最小的修改就可移植到一个新的平台上。这里的平台包括硬件平台、操作系统、网络系统、中间件环境等。如果产品线中的产品需要运行在不同的平台上,或者整个产品线也有可能移植到一个新的平台上,层次体系结构则是产品线体系结构最好的选择。

综上所述,在产品线的宏观软件体系结构中,建议使用层次体系结构。在该层次体系结构中,公共服务和设施及其以下层次遵循标准体系结构;在领域层以标准体系结构为参考,在应用层根据应用的特定需求进行定制。

13.6 软件产品线体系结构的演化

产品线体系结构就是一个软件体系结构和一组在一族产品中可重用的构件,为增加软件重用、为企业降低软件开发和维护的成本提供了一个重要的途径。

产品线体系结构中的软件一旦开发出来，就要经历演化，因为新的需求总是在不断地出现。因为需要处理如此多的需求(这些需求甚至可能是自相矛盾的)，通常的处理方法是创建两个独立的演化周期。也就是说，对每个产品而言，需要合并产品特定的需求；对整个产品线体系结构而言，需要合并影响整个产品线中所有产品或大多数产品的需求。

产品线的演化是由需求变更驱动的，这些需求的变更可以来自多个方面，如市场、企业将来的需要、在产品线中引入新的产品等。产品线的演化可分为两个部分：一是企业如何组织其产品线结构的变化；另一个是实际的演化，该演化作为一个需求通过静态组织进行传播。

图 13-10 所示的组织结构图，其中特殊的业务部门由数个需求驱动。这些需求在该业务部门负责的产品和整个产品线的一般需求之间进行划分。产品线体系结构既可能影响一个特殊的体系结构框架，从而在其接口上创建一个变更，也可能引起该框架的一个或多个具体实现的改变。在某些情况下，需求甚至可以使一个构件分解为两个构件，或者在产品线体系结构中引入一个全新的构件，产品线体系结构和数个具体的框架实现实例化为一组产品。

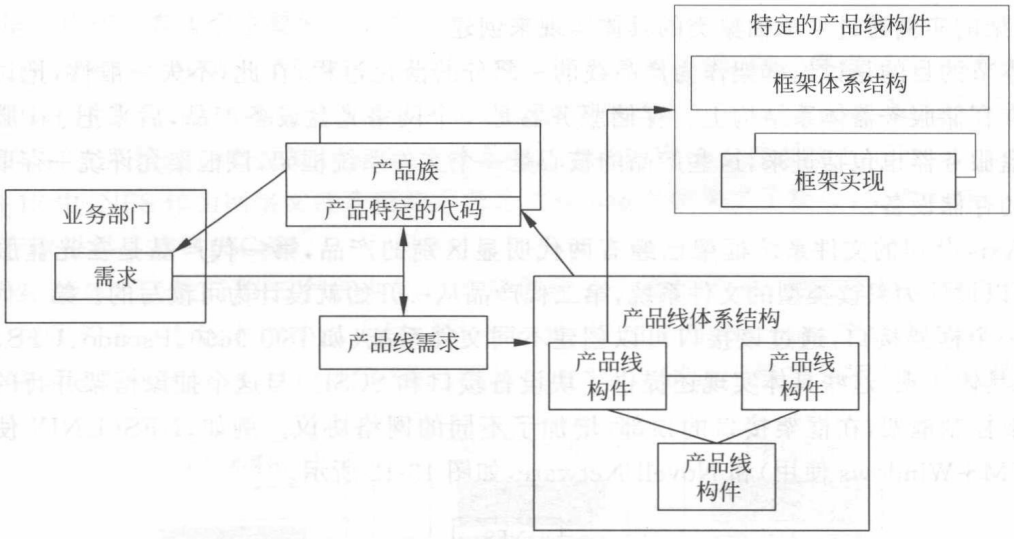


图 13-10　产品线的演化

下面通过一个案例讨论产品线体系结构的演化过程。

13.6.1　背景介绍

本节选择瑞典的 Axis 通信公司的产品线体系结构作为讨论案例。之所以选择该公司，是因为 Axis 是瑞典的一家大型企业，他们的产品研制和开发受到政府部门的资助，可以说是软件开发企业的一个典型代表。

Axis 公司是一家相对较大的软件和硬件企业，专业从事网络设备的开发。从单一的产品(IBM 打印服务器)开始，现在已经延伸到包括摄像服务器、扫描服务器、光盘服务器以及其他的存储服务器在内的产品线。Axis 公司在 20 世纪 90 年代就开始使用产品线方法，他们的软件产品线由一组大小不一的、可重用的、面向对象的框架组成。他们的产品线体系结构是作为不同的产品、产品类型及产品族的等级结构形式出现的，如图 13-11 所示。每个产品族都由一个业务部门进行维护，维护和演化产品线体系结构，使之满足特定部门的需要。

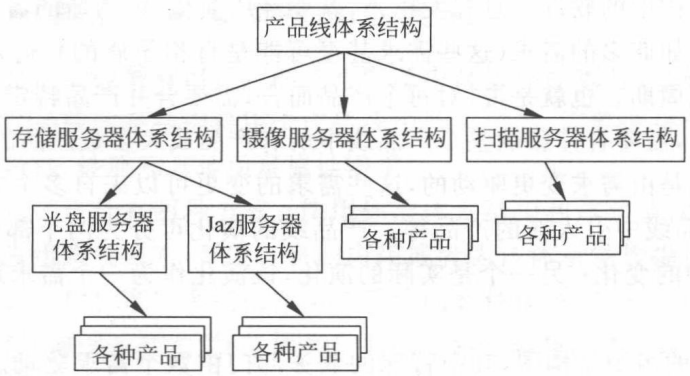


图 13-11　产品线体系结构的等级

按照 Axis 公司的观点,产品线体系结构是由构件及其关系组成的。构件在自己的面向对象的框架中,框架管理一些特殊的函数。这与传统的观点有点不同,在传统观点中,框架往往被当作整个产品。如上所述,Axis 的框架由一个抽象的体系结构和数个具体实现组成,框架的实例通过继承抽象类的具体实现来创建。

本节的目的是讨论框架作为产品线的一部分的演化过程,在此,不失一般性,把讨论焦点放在存储服务器体系结构上。存储服务器是一个网络光盘设备产品,后来把 Jaz 服务器和磁盘服务器也包括进来,这些产品的核心是一个文件系统框架,该框架允许统一存取所有类型的存储设备。

Axis 公司的文件系统框架已经有两代明显区别的产品,第一代产品是在光盘服务器中,所以设计为只读类型的文件系统,第二代产品从一开始就设计为可读写的。第一代产品包含一个框架接口,通过该接口可以创建不同文件系统(如 ISO 9660、Pseudo、UFS、FAT 等)的具体实现,这些具体实现还提供了块设备接口和 SCSI。与这个抽象框架并行的是一个存取控制框架,在框架接口的顶部,增加了不同的网络协议。例如,NFS(UNIX 使用)、SMB(MS-Windows 使用)和 Novell Netware,如图 13-12 所示。

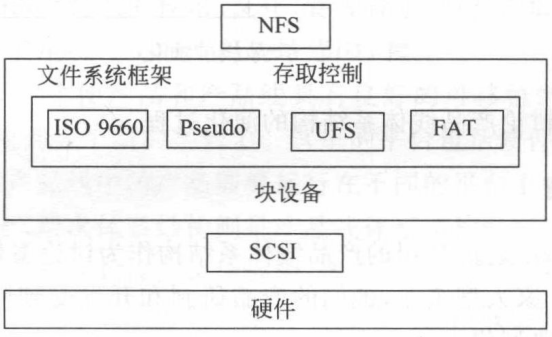


图 13-12　第一代文件系统框架

第二代产品在很大程度上与第一代产品类似,但做得更加模块化,从一开始就预见了系统将来可能的功能增强,因为这已经在第一代产品中发生过。第二代文件系统框架如图 13-13 所示。可以看出,该框架可以划分为更小的和更专业化的框架。值得注意的是,存取控制部分也分离成一个单独的框架。

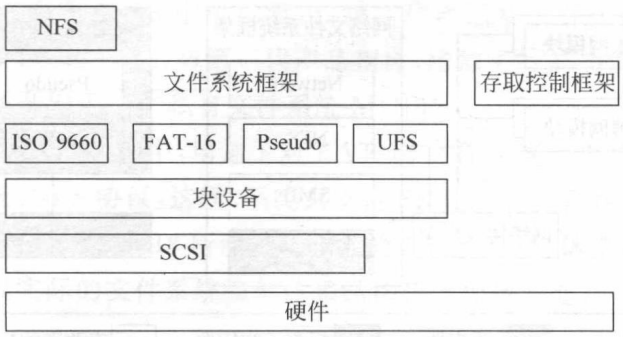


图 13-13　第二代文件系统框架

13.6.2　两代产品的各种发行版本

本节将介绍两代产品的各种主要发行版本。

1. 第一代产品

第一代产品有 4 个主要发行版本。

(1) 版本一。在第一个版本中，主要用来支持光盘服务器，支持网络通信、网络文件系统、光盘文件系统，能够访问光盘硬件。文件系统支持 ISO9660 文件系统，同时为了控制和配置的目的，系统还支持虚拟 Pseudo 文件系统。支持的网络文件系统有 NFS 和 SMB。在图 13-12 中，NFS 作为网络文件系统的示例为文件系统框架提供了接口，文件系统框架又为硬件存取提供了接口(SCSI)。

(2) 版本二。发行第二个版本的目的是创建一个新的产品，使之支持令牌网(Token Ring)以代替第一个版本中的以太网(Ethernet)。增加了对 Netware 文件系统的支持，对 SMB 协议进行了扩展，而且设计了 SCSI 模块。图 13-14 描述了第二个版本的改变情况。

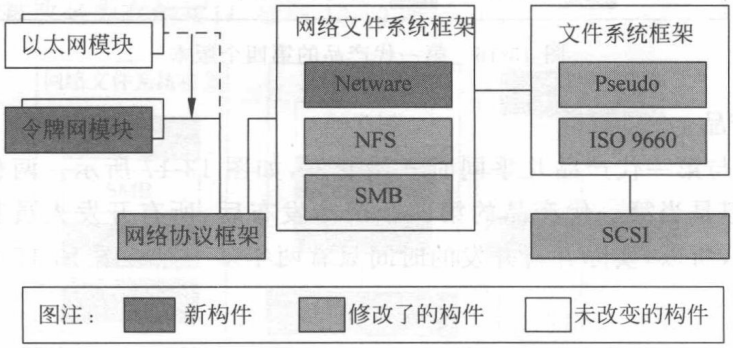


图 13-14　第一代产品的第二个版本

(3) 版本三。发行产品线体系结构的第三个版本的目的是整理和修改以前版本中存在的问题。修改了 SCSI 驱动以支持新版本的硬件，同时，增加了一个 Web 接口用来浏览 Pseudo 文件系统，在 ISO 9660 模块中增加了对长文件名的支持。图 13-15 描述了第三个版本的改变情况。

(4) 版本四。第四个版本是第一代产品的最后一个版本。在这个版本中，增加了对 NDS(一个 Netware 协议)的支持，同时改进了对 Netware 协议的支持。NDS 需要新的算法

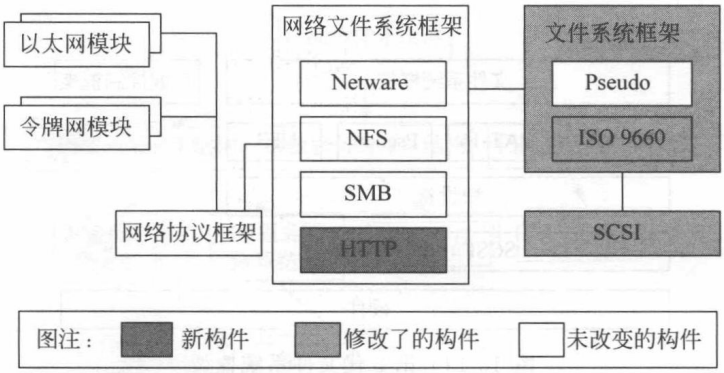


图 13-15　第一代产品的第三个版本

来获取存取文件的权限，因此，必须修改其他所有的网络文件系统的接口，使之为这种新算法提供支持。同时，去掉了在第二个版本中引入的名字空间缓冲(namespace cache)。图 13-16 描述了第四个版本的改变情况。

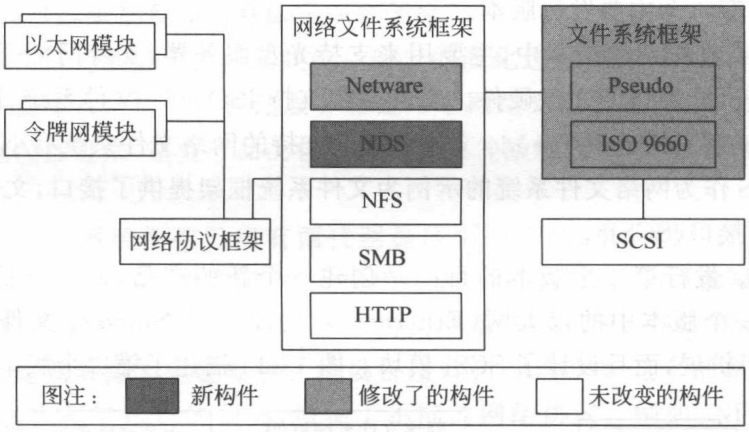


图 13-16　第一代产品的第四个版本

2. 第二代产品

第二代产品与第一代产品几乎同时开始开发，如图 13-17 所示。两代产品并存了将近 4 年的时间，但是当第一代产品的第四个版本发布后，所有开发人员和其他资源都转向了第二代产品，所以，实际并行开发的时间只有两年多一点。图 13-17 中的箭头表示资源的转移。

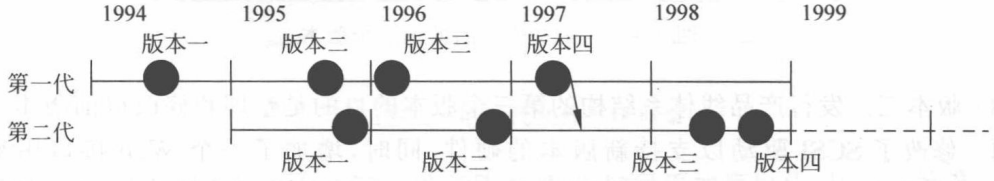


图 13-17　文件系统框架的时间线

(1) 版本一。第二代产品的第一个版本的需求与第一代产品的第一个版本十分相似，所不同的是第二代产品从一开始就是要开发可读写的系统。这时，利用了第一代产品开发

中的经验。从图 13-12 和图 13-13 中可以看出，第二代产品从一开始就注重模块化。在第一个版本中，只支持 NFS 和 SMB，与第一代产品相比，增加了写的功能。另外，为了理解基于结点的文件系统，还开发了一个私有文件系统 MUPP。

(2) 版本二。在第二个版本中，增加了对 FAT-16 文件系统的支持，删除了第一个版本中的 MUPP 文件系统、NFS 协议，这样，系统只支持 SMB 协议。另外，对 SCSI 模块和块设备模块也做了一些修改，如图 13-18 所示。因为在第二代产品中，体系结构中的可变部分被分离成几个框架，所以实际的文件系统框架体系结构仍然保持不变。

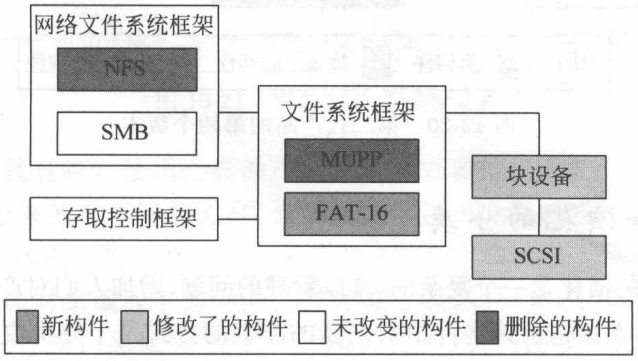


图 13-18　第二代产品的第二个版本

(3) 版本三。第三个版本的需求来自开发一个支持备份和 RAID 的硬盘服务器。为了支持磁带备份，增加了一个新的文件系统 MTF，并且决定增加对基于结点的文件系统 UDF 的支持，SMB 和 Netware 用来支持网络文件系统。另外，新版本还支持网络管理协议 SNMP。MTF 是作为一个新的文件系统来开发的，而 Netware 是从第一代产品中复制过来的。为了与新的网络文件系统、Netware、HTTP、SNMP 等协同工作，也修改了存取控制框架。最后，块设备被改名为存储接口，如图 13-19 所示。

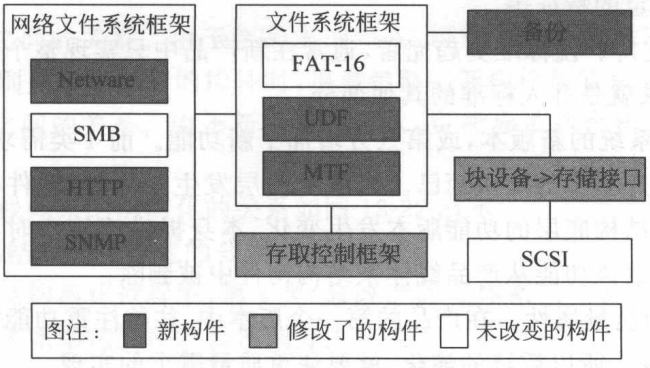


图 13-19　第二代产品的第三个版本

(4) 版本四。第四个版本主要是为了开发一个与光盘服务器协同工作的光盘转换器，为了满足这个需求，做了一些小的改动。在文件系统构件中，实现了使 ISO 9660 支持两种方式(Rockridge 和 Joliet)的长文件名。在网络文件系统中，又重新引入了 NFS 协议。抛弃了原来的存取控制框架，重新编写了新的存取控制框架，如图 13-20 所示。

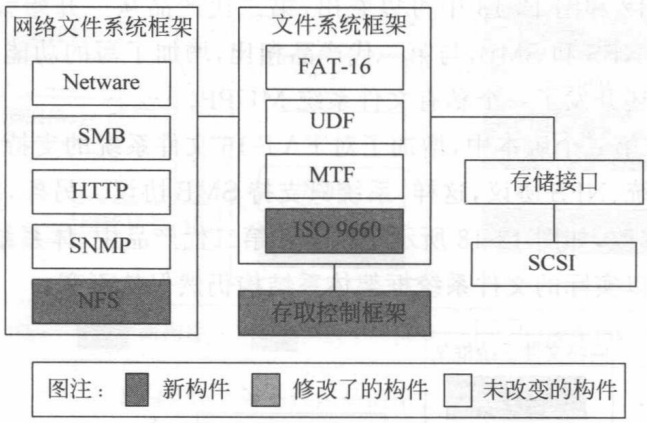


图 13-20　第二代产品的第四个版本

13.6.3　需求和演化的分类

产品线体系结构的演化是一个复杂的、难以管理的问题，增加人们对产品线体系结构演化的理解，改进在产品线体系结构及其构件中引入新的需求的方式是十分重要的。本节对需求的演化、产品线体系结构的演化、产品线体系结构构件的演化进行分类，讨论这些分类之间的关系。

1. 需求分类

根据 Axis 产品线体系结构的演化过程，可以把需求大致分为 6 类。

(1) 构建新的产品族。这类需求来自市场对新类型产品的需要，往往导致新业务部门的成立和新产品线的建立等。

(2) 在产品线中增加新的产品。一旦有了产品线，这种需求的主要工作就是改进功能，对软件进行个性化以满足最终用户的需要。这通常通过在产品线中增加新的产品来完成。

(3) 改进已有功能。这种需求不足以创建新的产品，只需要改进软件产品，使之支持新的标准、增加用户特定的特征等。

(4) 扩展标准支持。使标准更趋完善，通常在新产品中只实现整个标准的一部分，后续版本的一个典型需求就是并入标准的其他部分。

(5) 硬件、操作系统的新版本，或第三方增加了新功能。前 4 类需求都只是在软件本身范围内，为了支持用户更多的期望而已。当系统底层发生变化时，软件构件也要发生变更。例如，当产品线体系结构底层的功能版本发生变化，本身提供连接库时，开发人员会自然地使用底层连接库，导致该功能从产品线体系结构构件中被删除。

(6) 改进框架的质量属性。在产品的第一个版本中，往往注重功能需求的实现，从而对质量需求关注得较少。所以后续的演化，就要注重质量需求的实现。

以上 6 类需求分类的关系如图 13-21 所示。

2. 产品线体系结构演化的分类

把新的需求引入所有产品，将对产品线体系结构产生影响。下面给出产品线体系结构演化的 8 种情况。

(1) 产品线体系结构的分解。当决定开发一组新的产品时，就要在究竟把这组产品纳入现有产品线体系结构之中，还是有必要把现有产品线体系结构一分为二之间做个抉择。

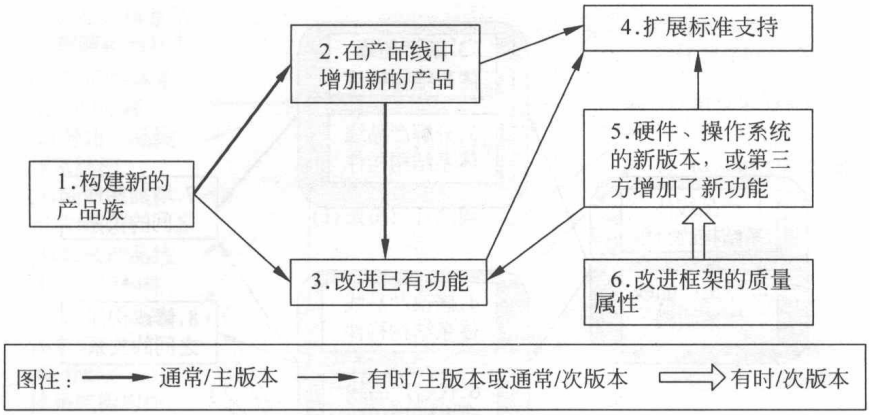


图 13-21　需求分类之间的关系

如果选择一分为二，就意味着使用已有的产品线模板创建另一个产品线，新的产品族或产品线体系结构可以与原来的有不同的方向，当然也可以保持类似的特征。

(2) 导出产品线体系结构。当引入一组新的产品时，第二种方法就是定义一个导出产品线，这类似于在面向对象编程中的子类。与复制产品线，然后两个产品线独立演化不同，导出的产品线是作为原来产品线的一个子产品线，这样，父产品线中的一些通用功能在子产品线中照样使用。

(3) 新产品线体系结构构件。有些需求无法由原有的产品线体系结构中的构件实现，在这种情况下，需要在产品线体系结构中增加新的构件，相应地调整构件之间的关系。

(4) 修改产品线体系结构构件。当软件的功能需求发生变化时，需要修改现有的产品线体系结构构件，以满足变化了的需求。

(5) 分解产品线体系结构构件。在产品线体系结构构件中创建新构件的另一种方法就是从框架中把功能分解出来。当一个构件包含的功能太多的时候，往往就会出现这种情况。

(6) 代替产品线体系结构构件。为了引入新的需求，仅修改框架接口是不够的，需要重新编写构件。

(7) 增加构件之间的关系。由于新需求的引入，可能导致原来没有关系的两个构件之间发生了关系。特别是在增加新的构件时，很显然要把新构件与原来的构件关联起来。

(8) 修改构件之间的关系。由于新需求的引入，可能导致两个构件之间的已有关系发生变化。

上述 8 类产品线体系结构演化的关系如图 13-22 所示。

3. 产品线体系结构构件演化的分类

在产品线体系结构演化过程中，第 4 类(修改产品线体系结构构件)是最常见的演化类型。下面把产品线体系结构构件的修改细分为 5 类。

(1) 新框架的实现。给定一个框架体系结构，扩展产品族支持功能的传统方法就是增加一个框架的实现。

(2) 修改框架的实现。当增加新的功能或重写已有功能时，可能会发生这种类型的演化。例如，为了更好地支持某些质量属性。

(3) 在框架实现中缩减功能。作为需求分类的第 5 点(硬件、操作系统的新版本，或第三方增加了新功能)的一个直接结果，一个框架中的功能可以缩减，因为它的部分功能在其他地方得以实现。

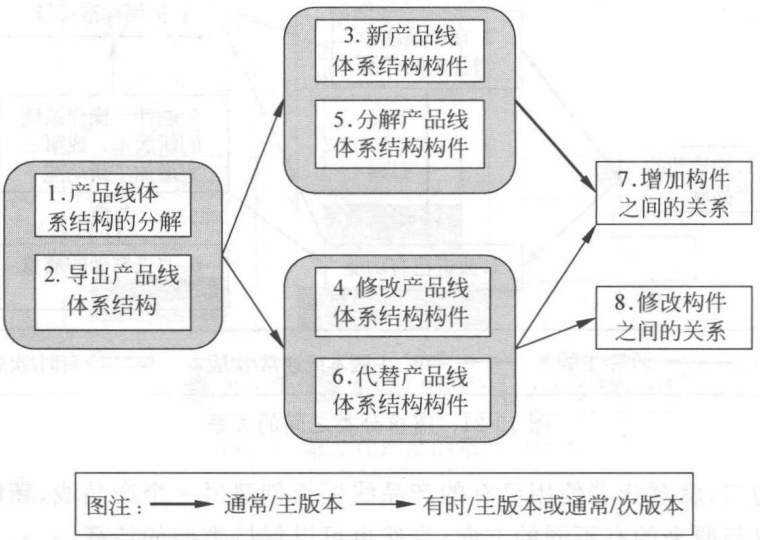


图 13-22　产品线体系结构分类之间的关系

(4) 增加框架功能。这与第 1 个分类有点类似，与第 1 个分类不同的是，第 1 个分类增加一个新的实现，该实现支持与其同族产品同样的功能。而第 4 个分类在框架的所有实现中增加新的功能。当现有框架不能满足需求功能时，就会发生这种演化。

(5) 使用其他的框架体系结构，增加外部构件。当一个项目临近结束时，开发人员都不愿意改动现有的框架，因为这样会影响系统的很大部分，会引入错误，需要进行更多的测试，且增加了风险。如果按照正常的实现方式，增加一个新的需求(或者重新理解和解释已有需求)，就会导致体系结构级影响的改变。为了不使前面的努力白费，同时又能满足新的需求，开发人员往往会在框架外部实现这个需求。

上述 5 类产品线体系结构构件演化的关系如图 13-23 所示。

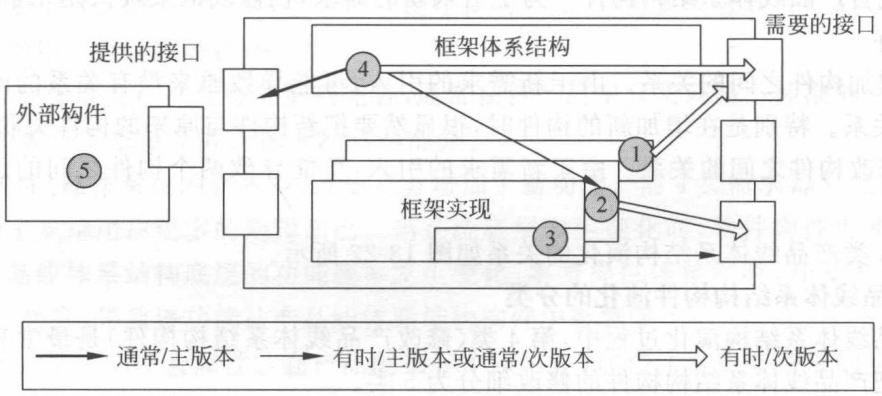


图 13-23　产品线体系结构构件演化分类

上面分别介绍了需求演化、产品线体系结构的演化和产品线体系结构构件的演化的分类。这些分类不是独立的，它们之间都有着各种联系。图 13-24 是描述了需求分类是如何导致产品线体系结构的演化和产品线体系结构构件的演化的。

在很多情况下，演化类型并不是由需求变化直接引起的，而是需求变化的间接结果。一种类型的变化引起另一种类型的变化，后者与前者往往处于同一个级别，即在产品线体系结

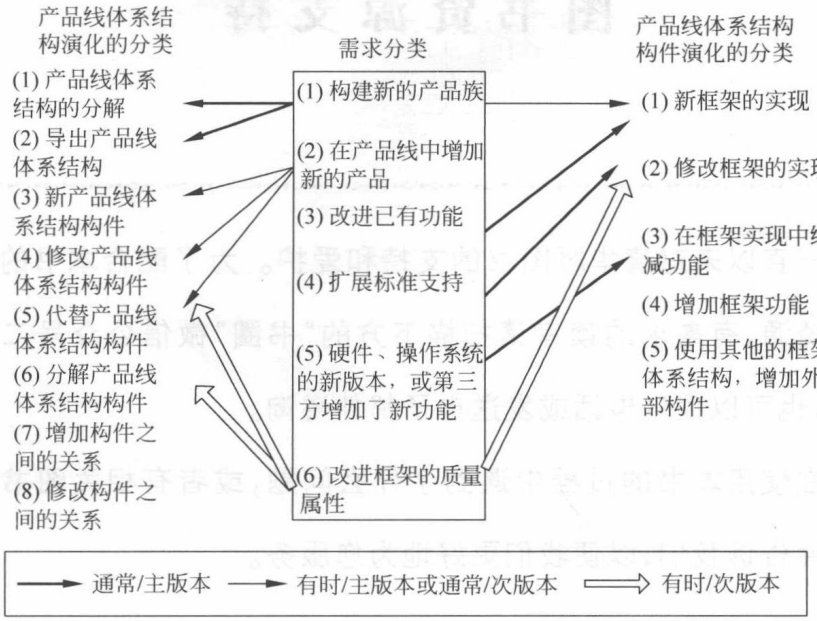


图 13-24　各种演化之间的关系

构中的变化会引起另一个产品线体系结构类型的变化。但也有例外，例如，一个新的框架实现可以导致另一个框架实现的改变，从而改变两个相关构件之间的关系。

思　考　题

1. 什么是软件产品线？软件产品线在软件开发过程中有什么作用？

2. 软件产品线包括哪些过程？如何实现软件产品线的创建与演化？

3. 结合一个你所熟悉的软件产品，说明你该如何建立该产品的软件产品线体系结构？

4. 软件产品线的演化是指什么？根据你所熟悉的软件产品线来考虑该产品线是否该演化，如何实现其演化？

5. 希赛公司是一家中等规模的计算机企业，专业从事网络安全防护软件系统的开发。最初仅开发基于 Windows 的个人防火墙产品开始，现在已经延伸到基于 Linux、Windows 系列、MAC 操作系统的个人防火墙、企业防火墙、入侵检测系统、病毒扫描系统、安全扫描系统等多种产品。公司原来的产品是一个一个地开发，为每个软件对应地组织一个项目组。为了适应快速变化的市场，降低开发成本，公司想引入产品线方法。然而，由于软件产品线方法涉及一个软件开发企业的多个产品，所以，公司的王总决定在弄清楚以下 3 个问题之后再做决定。

(1) 请说明希赛公司是否适合采用产品线方法，并说明理由。

(2) 请说明在原有产品的基础上建立软件产品线的方式，并做简要评价。

(3) 请说明成功实施产品线的主要因素。

参考文献

图书资源支持

感谢您一直以来对清华版图书的支持和爱护。为了配合本书的使用，本书提供配套的资源，有需求的读者请扫描下方的“书圈”微信公众号二维码，在图书专区下载，也可以拨打电话或发送电子邮件咨询。

如果您在使用本书的过程中遇到了什么问题，或者有相关图书出版计划，也请您发邮件告诉我们，以便我们更好地为您服务。

我们的联系方式：

地　　址：北京市海淀区双清路学研大厦 A 座 714

邮　　编：100084

电　　话：010-83470236　010-83470237

客服邮箱：2301891038@qq.com

QQ：2301891038（请写明您的单位和姓名）

资源下载：关注公众号“书圈”下载配套资源。

资源下载、样书申请

书圈

获取最新书目

观看课程直播

高等学校软件工程专业系列教材

软件体系结构
原理、方法与实践（第3版）

课件下载·样书申请

书 圈

清华大学出版社

官 方 微 信 号

ISBN 978-7-302-56768-4

定价：59.80元

[General Information]
书名=软件体系结构原理、方法与实践（第3版）
页数=362
SS号=15106009