返回总目录

目　　录

第 6 章　物理架构视图

系统架构是构成系统的各部分(结构、接口、通信机制)的框架性描述。通过定义一个合适的架构，使得更容易完成下述工作：浏览系统，找到一个特殊功能的位置或概念，标志加入一个新功能或概念的位置使其更适合总的系统架构。架构必须足够详细从而使得能够将它映射到真正的代码。架构不仅要容易浏览、足够详细，而且还要具有可伸缩性，即能够从不同的级别上了解它。例如，架构应该提供一个包括几个主要部分的顶级视图，从顶级视图中，开发人员可以选取其中的一个部分，了解其内部结构，然后更进一步了解更详细的结构。通过特定工具，详细了解系统的各个部分是可能的。

一个定义明确的架构允许加入新的功能和概念而不会给系统中的其它部分带来新的问题(例如，在一个旧的单片集成电路系统中，系统中某一部分的小的改变将导致一些好象与此无关的部分停止工作，这主要是因为系统中各部分存在着复杂的关系所致)。

架构给开发人员提供一个有关系统的视图，通过它开发人员可以知道系统是如何构造的，某一功能或概念在何处。随着时间的改变，在开发过程中因为重要的发现和经验这张视图可能不得不作些改变。架构必须随着被开发系统的变化而变化，应该实时地反应系统在各个阶段的结构。很自然地，基本架构在系统的第一个版本中定义，初始的架构的质量对于开发人员改变、扩展、更新系统功能有着至关重要的作用。

UML 中，“架构”的定义为：

架构是系统的组织结构。可以递归地将架构分解成：通过接口交互的部分，连接各个部分的关系，组装各个部分的约束。

Buschmann 等人(1996)提供了另一个软件架构的定义：

软件架构是系统的子系统和组件及其之间关系的一种描述。典型地，子系统和组件是通过不同的视图描述的，显示了软件系统与功能和非功能的性质。系统的软件架构是一个人工的产物，它是软件设计活动的结果。

可以用很多视图来描述架构，每一张视图集中描述系统的某一个方面。系统的完整结构的描述只需定义所有相关的视图。在 UML 中，这些视图通常为：用例视图，逻辑视图，并发视图，组件视图，和分布视图。一个更广的分类方法将架构分解成逻辑和物理架构。逻辑架构主要指定系统的功能特点，通过系统的功能需求来描述；物理架构则描述系统的非功能部分，如可靠性、兼容性、资源使用和系统分布。这些架构将在本章中详细讨论。

有时候，有经验的开发者有一种“神奇”的能力来定义好的架构。但是，这种能力来自于他设计的很多系统，通过大量的系统设计，他们知道什么样的解决方案能解决问题，什么样的方案不能工作。他们一般重用过去的成功的解决方案。事实上，最近，许多工作集中在研究如何描述有经验的开发人员在设计软件架构时重复使用的架构模式或框架(“解决方案”)。Buschmann 等人(1996)定义了下列架构模式：

- 层模式：将系统分解成一组组子任务，每一组子任务处于一个特殊的抽象级。
- 管道和过滤模式：系统处理数据流，将很多处理步骤封装在过滤组件中。数据通过管道在相邻的过滤器间传递，可以将过滤器重新组装建立相关的子系统或系统

行为。

- 黑板模式：系统中，几个特殊子系统组合它们的知识建立一个部分的或大约的方案来解决还没有确切解决方案的问题。
- 代理模式：将系统分解成很多组件，组件通过远程服务激活来交互。代理组件负责协调通信和传送结果和意外。
- 模型-视图-控制器模式：将一个交互式系统分解成三个组件：包括核心功能和数据的模型，一个或多个显示信息的视图，一个或多个处理用户输入的控制器。一种变化传递机制确保用户接口和模型间的一致性。
- 微内核模式：将一小部分核心功能与可扩展的功能及与客户有关的部分分开。微内核也可以作为一个接口，用来插入这些可扩展的功能，并协调它们之间的交互。

Rumbaugh 等人(1991)也定义了许多架构框架：

- 批处理变换：整个输入集的数据变换。
- 连续变换：随着输入的到达，连续进行数据变换。
- 交互式接口：系统被外部交互所驱动。
- 动态模拟：系统模拟进化现实世界中的对象。
- 事务管理器：对于关注存贮和更新数据的系统，经常需要从不同的物理位置并发访问数据。

通常情况下，没有系统仅使用这些框架或模式中的一种。在系统的不同部分、不同的级别上使用不同的模式。可以用层模式来定义一个特定子系统的架构，同时可以用另一种模式来定义子系统中的某一层。为了得到一个好的软件架构，你必须熟悉架构模式的设计，知道在什么时候使用它们，如何将它们组合在一起。

有了这些定义后，接下来的问题就是如何定义一个好的架构？下面是有关这个问题的指南：

- 正确地描述系统的各部分，包括系统的逻辑架构和物理架构。
- 在系统图中，开发人员能够容易地找到某一功能或概念的位置。功能或概念要么是面向应用(应用域中的事情的模型)的，要么是面向设计的(一些技术实现解决方案)。这也意味着系统的需求应该在处理它的代码中是可跟踪的。
- 系统改变和扩展应该很容易地在系统中找到位置，并且对系统中的其它部分不应有负面影响。
- 在不同部分之间的接口应该是：简单、有明确定义和明晰的关联性，从而在开发系统的某一部分时不需对整个系统的所有细节有完整的子解。
- 满足所有这些要求的架构是很不容易设计的，有时候不得不作出一些折中的方案。但是，定义一个好的基础架构是成功地开发一个系统中最关键的步骤。如果这一点处理得不好，将导致开发的系统很难改变、扩展、维护和理解。

6.1 逻辑架构

逻辑架构处理系统的功能，将功能分配到系统的各个部分，详细说明它们是如何工作的。逻辑架构包括应用逻辑，但不是将逻辑物理上分布到不同的进程、程序或计算机。逻辑架构有助于更清楚地了解系统的结构，使得更容易管理和协调工作(尽可能有效地利用人力开发资源)。并不是逻辑架构的所有部分都需要在工程中开发，通常情况下，类库，二进制构件和模式只需购买就可。

逻辑架构解决下面的问题：

- 系统提供什么样的功能？
- 哪一个类存在，类之间是如何联系在一起？
- 类和对象是如何协作来完成系统功能？
- 什么是系统功能的时间约束？
- 许多开发人员在开发架构时都遵循的一个合适的计划是什么样？

在 UML 中，用来描述逻辑架构的图有：用例，类，状态，活动，协作和序列。这些图在前面的章节中均已作了描述。一个公共的架构是一个三层结构，系统被分成：接口层，业务对象层和数据库层。图 6-1 显示了一个这样的逻辑架构，包括每一层的可能的内部架构。从这个包开始，可以到达其它描述每一个包中的类及类间协作的图。

6.2 物理架构

物理架构详细描述系统的软件、硬件。它描述硬件结构，包括不同的节点以及节点间如何连接。物理架构还说明实现逻辑架构中定义的概念的代码模块的物理结构和相关性，软件运行时，进程、程序和其它组件的分布。物理架构试图有效地利用软、硬件资源。

物理架构解决下面的问题：

- 类和对象物理上分布在哪一个程序或进程中？
- 程序和进程在哪台计算机上运行？
- 系统中有哪些计算机和其它的硬件设备，它们是如何连接在一起？
- 不同的代码文件之间有可关联？如果某一文件被改变，其它的文件是否需要重新编译？

物理架构描述软、硬件的分解。将逻辑架构映射到物理架构，逻辑架构中的类和机制被映射到物理架构中的组件，进程和计算机。这种映射允许开发者根据逻辑架构中的类找到它的物理实现。反之亦然，跟踪程序或组件的描述找到它在逻辑架构中的设计。

如前所述，物理架构关心的是实现，因而可以用实现图建模。UML 中的实现图是组件和展开图。组件图包括软件组件：代码单元和真正的文件(源代码和二进制代码)的结构。展开图显示系统运行时的结构，包括物理设备和软件。

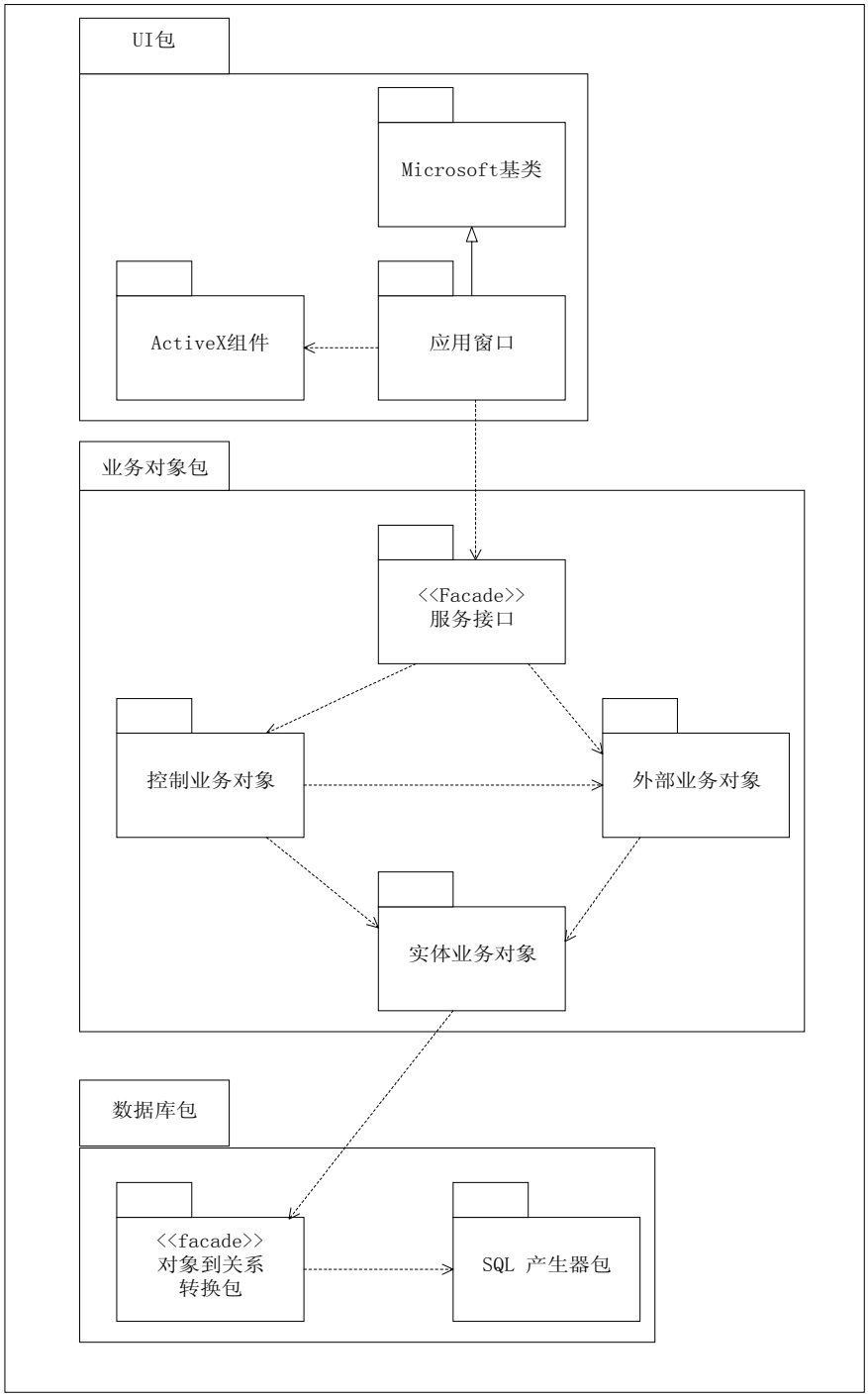

图 6-1 用 UML 包显示的公共三层架构以及包间的关系

6.2.1 硬件

物理架构中的硬件概念可以分为：

- 处理器：执行系统中的程序的计算机。处理器可以是任意大小，从嵌入式系统中

的微处理器到超级计算机。

- 设备：指的是支持设备，如打印机、路由器、读卡机等。它们一般被连接到控制它们的处理器上。在处理器和设备之间有一个好的界限(大多数设备都有自己的CPU)但是，一般来说处理器运行系统中的软件。
- 连接：处理器之间有连接。处理器与设备之间也有连接。连接表示两个节点间的通信机制，可以用物理媒体(如，光纤)和软件协议(如，TCP/IP)来描述。

6.2.2 软件

传统上，系统架构中的软件定义不是太严格，一般将软件定义为：软件由“部分”组成，“部分”可以是包，模块，组件，名字空间或子系统。在架构中处理的模块的公共名字是子系统，一个较大系统中的较小的系统。子系统有一个接口，可以将其内部分解成更详细的子系统或类和对象。可以将子系统分派给进程执行(进程可以指派给计算机执行)。

在 UML 中，将子系统抽象为类包。一个包将许多类组合成一个逻辑组，但没有定义语义。在设计中，定义一个或多个组件作为子系统的外观是很常见的。外观组件提供访问子系统(包)的接口，是系统中的其余部分唯一可见的组件。通过外观组件的使用，包成为一个非常模块化的单元，其内部设计细节被隐藏起来，只有外观组件与系统中的其它模块有关系。在查看包时，对于那些想利用包提供的服务的人来说，只对外观组件感兴趣。有的时候，在图中只显示外观组件。包既可用来进行逻辑设计，也可以用于物理架构中。在逻辑设计时，它将许多类组合成一个单元；在物理架构中，包封装许多组件(典型地，实现逻辑设计中的类)。

在 UML 中，外观有一种表示，但是后被应用到包上。一个包包含一个引用其它元素的外观包，但它却没有自己的元素。外观包表示装入它的包，它被版类《facade》解释为被装入包的外观。虽然这是外观概念的一个可能的模型，但是带外观组件(代表一个外观类)的模型方案也应该被考虑和利用。

描述软件的主要概念是组件、进程、线程、对象。

- 组件：在 UML 中，组件是指“在一组模型元素实例的物理打包时可重用的部分”。意思是说，组件是物理实现(例如，源代码文件)，它实现类图或交互图中定义的逻辑模型元素。组件可以看作是开发的不同阶段，如在编译时，链接时，运行时。在一个工程中，经常将组件的定义映射到实现环境(也就是说，编程语言和使用的工具)。
- 进程和线程：进程表示重量控制流，而线程则代表轻量控制流(有关这方面的讨论可参考第 8 章)。它们都被用来描述版类化的活动类，活动对象被分配给一个可执行的组件执行。
- 对象：对象(被动的)没有自己的执行线程。只有当其它东西发送消息给它们时(调用它们的操作)它们才运行。它们可被指派给一个进程或线程(一个可执行的对象)或直接指派给一个可执行的组件。

6.3 组 件 图

组件图描述软件组件及组件之间的关系，显示代码的结构。组件是逻辑架构中定义的概念和功能(类、对象、它们的关系、协作)在物理架构中的实现。典型情况下，组件是开发环境中的实现文件，如图 6-2 所示。

软件组件可以是下面任何一种：

- 源组件：源组件只在编译时是有意义。典型情况下，它是实现一个或多个类的源代码文件。
- 二进制组件：典型情况下，二进制组件是对象代码，它是源组件的编译结果。它应该是一个对象代码文件，一个静态库文件或一个动态库文件。二进制组件只在链接时有意义，如果二进制组件是动态库文件，则在运行时有意义(动态库只在运行时由可执行的组件装入)。
- 可执行组件：可执行组件是一个可执行的程序文件，它是链接(静态链接或动态链接)所有二进制组件所得到的结果。一个可执行组件代表处理器(计算机)上运行的可执行单元。

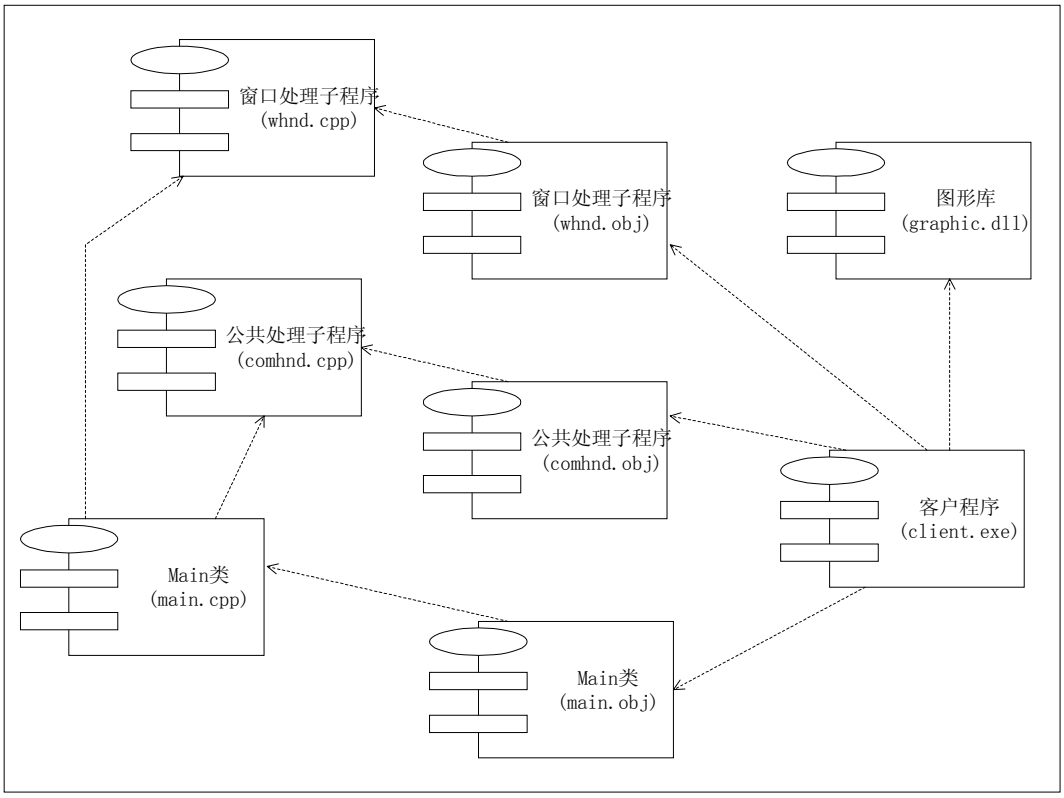

图 6-2 显示许多组件的组件图：源、二进制、可执行组件及它们的关系

在 UML 中，组件用一个左边带有一个椭圆和两个小矩形的矩形符号来表示(这个符号在 Booch 方法中表示一个模块)。组件名放在组件符号的下面或写在组件符号的大矩形内。

组件是类型，但是仅仅可执行的组件可能有实例(当它们代表的程序在处理器中执行时)。组件图只将组件显示成类型。为了显示组件的实例，必须使用展开图。在展开图中可执行组件的实例被指派给执行它们的节点实例。

组件间的相关性连接，用一条带开箭头的虚线来表示，表示一个组件只有同另一个组件在一起才有一个完整的定义。从源代码组件 A 到另一个组件 B 的相关性是指从 A 到 B 之间有一个与语言有关的相关性。在编译化语言中，可能意味着 B 的改变可能需要重要编译 A，因为编译 A 时需要用到组件 B 中的定义。如果组件是可执行的，相关性连接可以用来描述一个可执行的程序需要哪些动态链接库才能运行。

组件可以定义其它组件可见的接口。接口要么是源代码级(如在 Java 中)，要么是运行时使用的二进制级(如在 OLE 中)。接口用从组件开始画的一条线表示，线的另一端为一个小空心圆。接口名放在圆的边上。然后，组件间的相关性就能够指向用到的组件的接口，如图 6-3 所示。

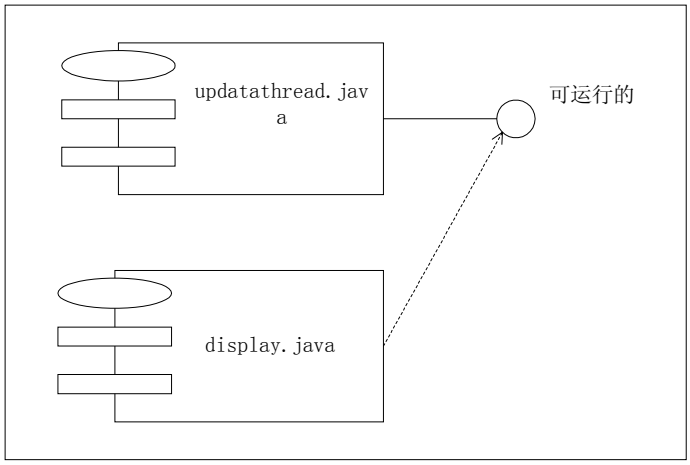

图 6-3 接口和相关性

6.3.1 编译时的组件

编译时的组件是包含工程中产生的代码的源组件，如图 6-4 所示。其它的组件，如链接时和运行时的组件是从编译时的组件中产生的。可被编译时的组件使用的某些版类为：

《file》：表示包含源代码的文件。

《page》：表示 Web 页。

《document》：表示文档(包含文档而不是可编译的文档)。

从一个编译时的组件到其它的编译时的组件的相关性显示了它需要其它的组件中的哪一个才能得到完整的定义。例如，它将哪一个编译时的组件包含在它的定义中？(如图 6-4)。

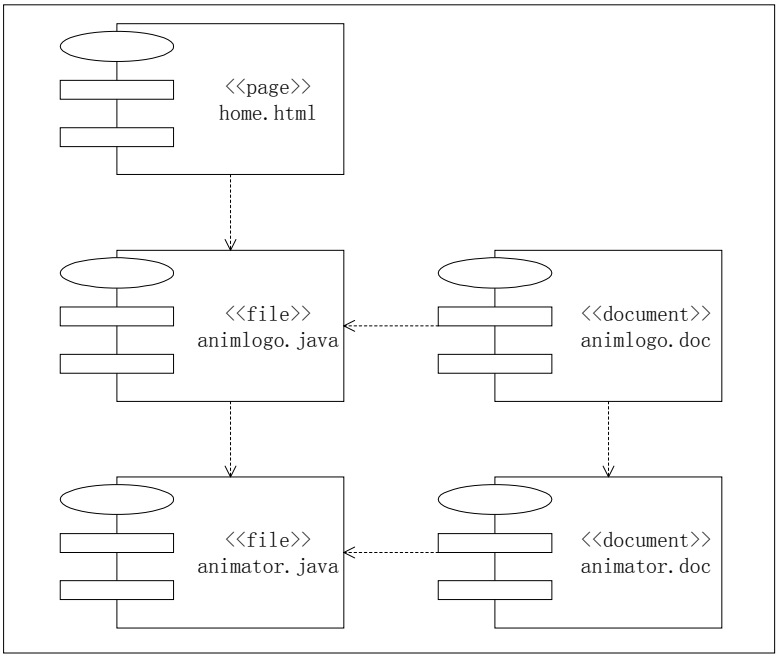

图 6-4　源代码组件间的相关性

6.3.2　链接时的组件

链接时组件用于链接系统时。典型情况下，链接时的组件代表编译时的组件所得到的结果或表示编译一个或多个编译时的组件所得到的库。动态链接库(DLL)是一种特殊情况，它是在运行时而不是在编译时被链接到一个运行时的组件。可以利用版类《library》来说明一个组件是静态库还是动态库。

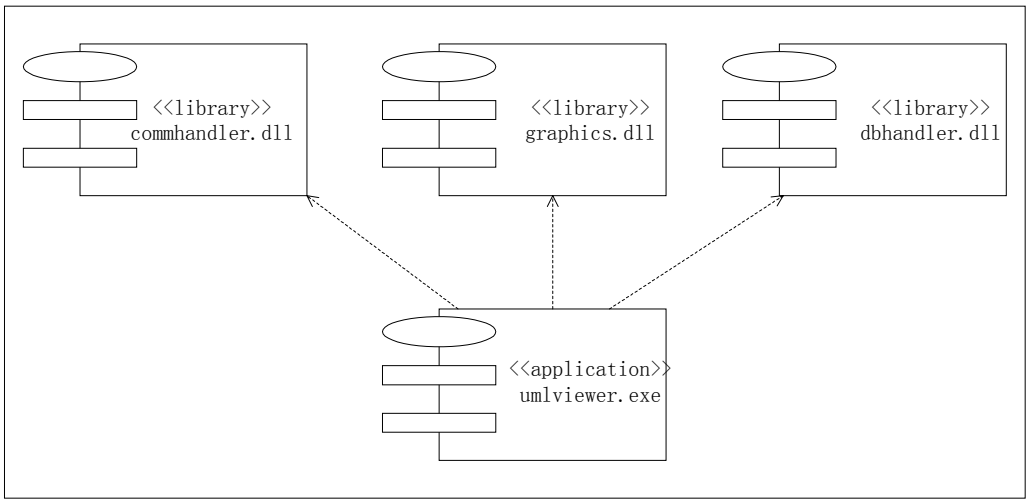

图 6-5　运行时的组件

6.3.3　运行时的组件

运行时的组件表示在执行系统时使用的组件，如图 6-5 所示。它是通过链接时的组件

(从对象代码和库)产生的，在某些情况下，可以直接从编译时的组件产生(在 Java 中，“可执行”的字节码直接从编译时的 Java 代码产生)。版类《application》表示一个可执行的程序。版类《table》表示一个数据库表，该库也可看作是运行时使用的组件。

只有运行时的组件才能有实例且放在节点上(展开图中的单元)。一个组件的运行时的实例的意思是，根据组件的类型，启动几个进程来运行组件文件中指明的应用。运行时的组件的相关性是运行它时所需的其它的组件：动态链接库、映象文件或数据库表。

6.4 展 开 图

展开图描述处理器、设备、软件组件在运行时的架构。它是系统拓朴的最终的物理描述，即描述硬件单元和运行在硬件单元上的软件的结构。在这样的架构中，在拓朴图中寻找一个指定节点是可能的，从而了解哪一个组件正在该节点上运行，哪些逻辑元素(类，对象、协作等等)是在本组件中实现的，并且最终可以跟踪到这些元素在系统的初始需求说明(在用例建模中完成的)中的位置。

6.4.1 节点

节点是拥有某些计算资源的物理对象(设备)。这些资源包括：带处理器的计算机，一些设备如打印机、读卡机、通信设备等等。在查找或确定实现系统所需的硬件资源时标识这些节点，主要描述节点两方面的内容：能力(如基本内存，计算能力，二级存储器)和位置(在所有必须的地理位置上均可得到)。

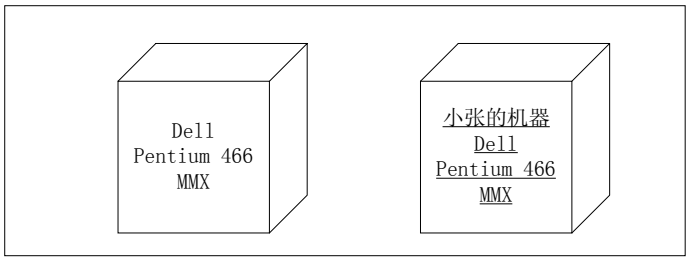

图 6-6 Dell Pentium 466 MMX 是一个节点类型，小张的机器是该类型的一个实例

可以将节点作为类型和实例(节点是一个类)来显示，类型描述处理器或设备类型的特点，实例表示类型的直正出现(机器)，如图 6-6 所示。系统能力的详细定义可以从两个方面入手：属性或特点。节点用一个三维立方体来表示，节点名放在立方体的内部，如同类和对象的做法一样，如果用该符号表示实例，则在名字下面有一条下划线。

系统中的设备也表示成节点，典型情况下，用版类来指定设备的类型，或用一个名字来表示类型，该名字至少能清楚地定义它是设备节点而不是处理器节点。

6.4.2 连接

节点间通过通信关联连接在一起，如图 6-8 所示。这种通信关联用一条直线表示，说明在节点间存在某类通信路径，节点通过这条通信路径交换对象或发送消息。通信类型用

版类来表示，定义通信协议或使用的网络。

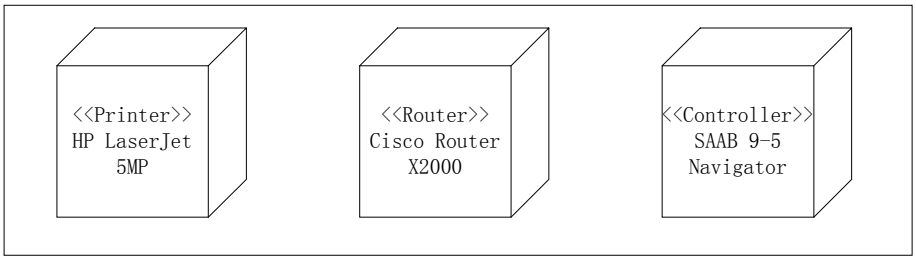

图 6-7 设备节点和可能的版类

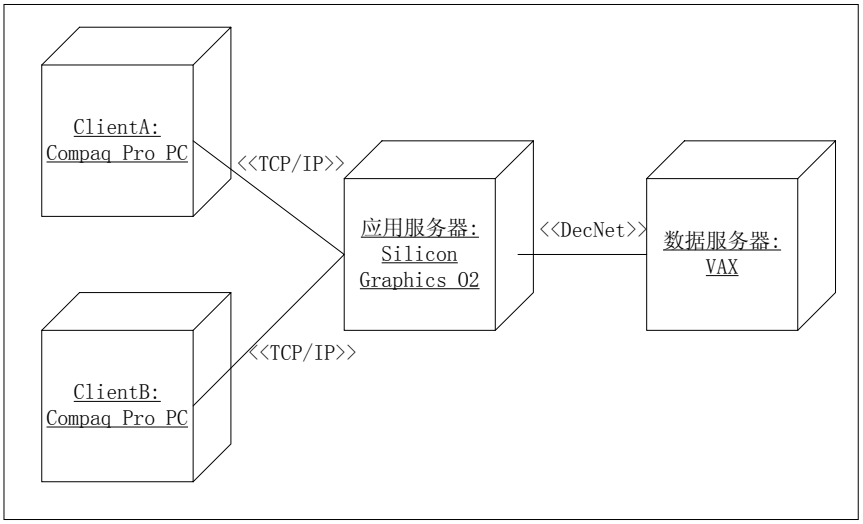

图 6-8 节点间的通信关联

6.4.3 组件

可将可执行组件的实例包含在节点实例符号中，表示它们处在同一个节点实例上，且在同一个节点实例上执行。从节点类型可以画一条带有版类《support》的相关性箭头线到运行时的组件类型，说明该节点支持指定组件。当一个节点类型支持一个组件类型时，在该节点类型实例上执行它所支持的组件的实例是允许的。例如，如果不能在 AS/400 上创建一个 Windows 程序，则节点类型 AS/400 不支持 Windows 程序组件。

可以通过虚线相关性箭头将不同组件连接在一起，如图 6-9 所示的那样(记住，在展开图中，只显示运行时的组件)。这意味着，一个组件使用另一个组件中的服务。

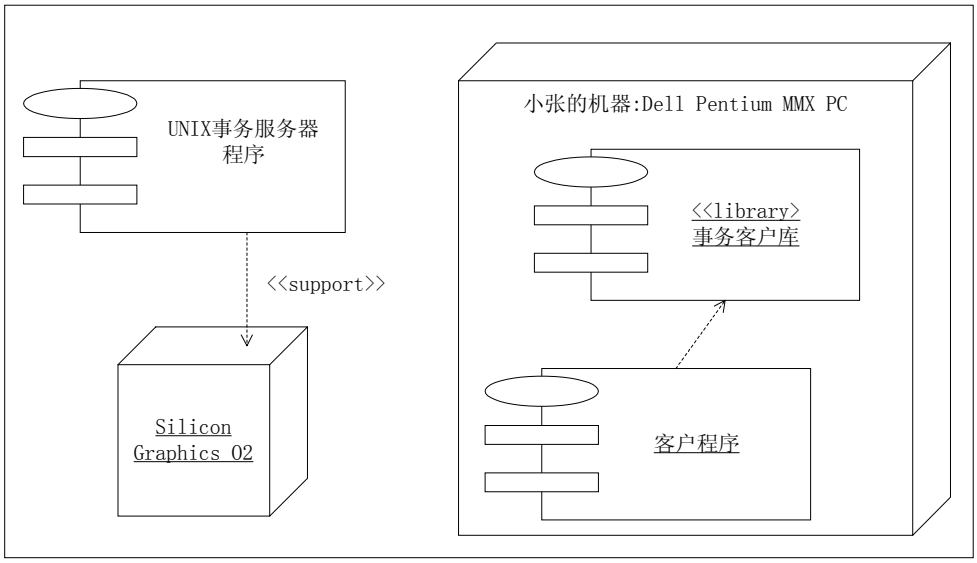

图 6-9 一个节点类型支持一个运行时的组件类型，该运行时的组件实例在节点实例上执行

6.4.4 对象

将一个对象放在一个节点实例的内部表示该对象处于该实例中。对象要么是活动的(带有版为《precess》和《thread》，用粗线表示)，在节点上执行，或是静态的(passive)(参考图 6-10)。对象被包含在另一个对象或一个组件内，对于这种情况，可以用嵌套的符号来表示，或者，如果这样表示太复杂，则可以用一个位置性质来说明对象的位置。例如，一个静态对象可以包含在一个进程(活动对象)中，进程 “活”在组件中，组件被指派给节点。

可以将对象直接画在一个节点的内部，而不必显示实现它的组件。关于实现它的组件的有关信息，要么通过位置性质来定义，要么在展开图中仍然未定义。

在分布式系统中，在系统的生命期内，对象可以在不同的节点间移动，如图 6-11 所示。在技术上，对象的移动可以通过分布式对象系统，如 OLE 或 CORBA 来实现。在这些系统中，可以通过网络传送对象，可以改变其在系统中的位置。在对象的生命期内，改变其位置的对象可以包含在它可能到达的所有节点中。为了显示对象是如何被分布在系统中，在对象出现的不同地点间画上带有版类《becomes》的相关性符号。相关性可能包含有关时间的性质或触发对象改变位置的条件。

6.5 复杂节点的建模

在 UML 中，节点被定义成类。因此，节点间的更复杂的关系在类图中描述，如图 6-12 所示。在描述一组节点时，通用化被用来定义一般的节点配置，而特殊化则被用来描述特殊情况。在关联关系中可以给节点指定一个角色，以及定义一个接口。

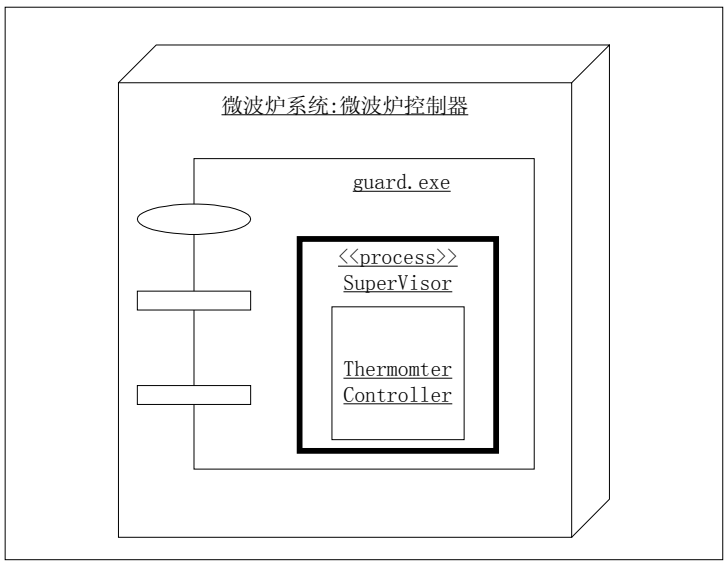

图 6-10 “活”在一个组件实例(类型为 guard.exe)中的位于一个活动进程对象中的静态对象(对象类为 Thermometer Controller)，组件被指派给节点微波炉系统(类型为微波炉控制器)

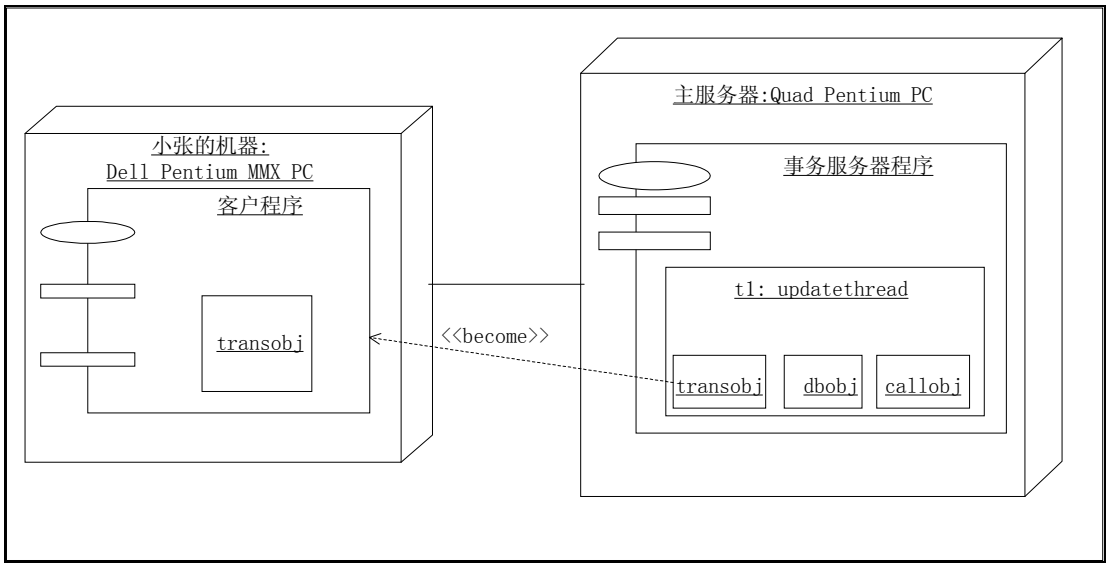

图 6-11 对象被分配给节点。开始在主服务器上的 transobj 对象可分布到 Dell PC 节点上(用版类《becomes》表示)

6.6 节点的组件配置

在逻辑设计中定义的类和协作被分配给实现它们的组件。分配与使用的程序语言有关。例如，C++实现一个类用两个文件：一个包含说明的头文件(.h)和一个包含操作实现的实现文件(.cpp)。Java 实现一个类只用一个文件，该文件包含说明和实现。某些语言，

如 Java，有模块概念，模块比类的规模要大一些，可以用它来组合类。这个可被用来实现 UML 中的包(在 Java 中也称为包)。编程语言定义将类分配给组件的规则。

进程被分配给执行它们的组件。这种分配是根据活动对象的需要或地理上分布系统的需要来实施的。标识并行执行的需求(即活动对象)和将系统划分为许多活动将在第 8 章中讨论。

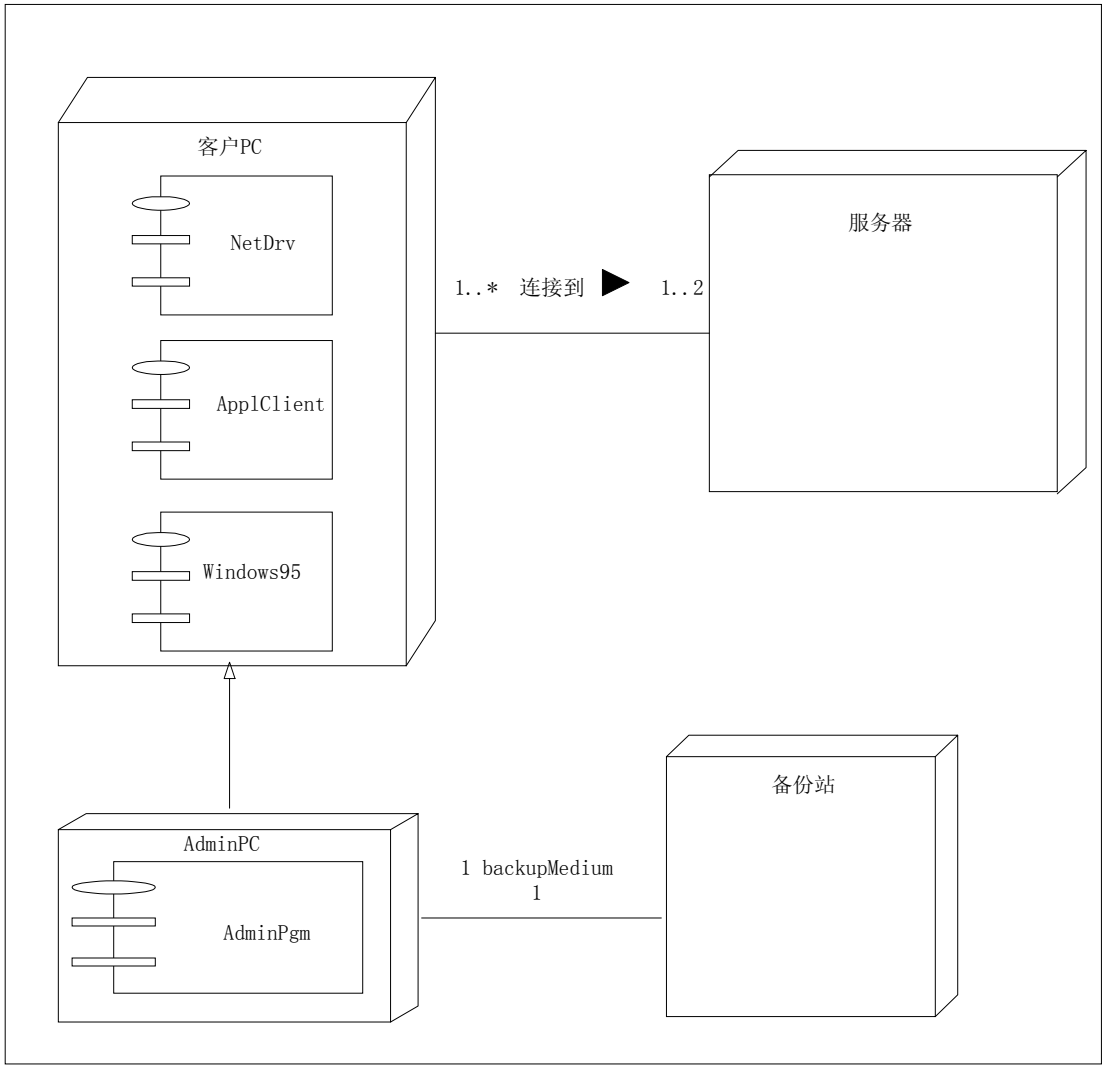

图 6-12 类图。在该类图中，不同节点类型间的关系被模型化。客户 PC 节点同服务器节点有一个关联。AdminPC 节点是客户 PC 的一种特殊情况，它上面有一些额外的程序。AdminPC 同备份站(BackupStation)之间也有一个关联

最后，组件被分配给节点。一个组件实例至少在一个节点实例上运行。将组件分配给节点也可能影响真正的拓朴，所以不得不对节点的建议配置作些改变。在分配组件给节点时，有许多因素是需要考虑的：

- 资源利用：决定物理架构和组件分配的一个主要目的是硬件资源的利用。当然应该用一种有效的方式来利用硬件，以充分利用每一个节点的能力(当然，不能用过

度，否则将导致可怜的性能)。

- 地理位置：何处(哪一个节点)需要某一功能，本地需要什么样的功能(因为性能，或即使其它的节点不工作本地也必须有这一功能)也是一个重要的因素。
- 设备访问：节点需要什么设备？打印机可以连接到服务器，或每一个客户都需要一台本地打印机吗？
- 安全：哪一种架构能够以优化的和有效的方式来处理访问权限和信息保护问题？访问可能比较关心地理位置(将服务器放在一个安全的地方)和通信解决方案(使用安全的软件和硬件通信)。
- 性能：高性能通信有时会影响组件的位置。通过在本地节点上产生一个代理而不是访问另一个节点上的真正的组件，可能会改善性能。
- 可扩展性和可移植性：当不同的节点有不同的操作系统和机器体系结构时，你必须考虑哪些组件依赖于某一操作系统，哪些组件必须是可移植到许多操作系统上。这同样会影响这些组件的位置，也许也影响实现用的编程语言。

重复的设计应该产生展开图。需要测试不同的解决方案，首先在建模阶段进行讨论，然后实现一个原型。理想情况下，系统是很灵活的，即一个组件可以在不同的节点上移动。分布式对象系统，如 OLE 或 CORBA，有助于实现这一目标。

6.7 小　　结

系统架构可以分成逻辑架构和物理架构。逻辑架构显示类和对象，以及它们之间的关系和协作，从而完整显示系统的功能。可以用用例、类、状态、序列、协作图，活动图来写逻辑架构的文档。

物理架构处理代码构件的结构和组成系统的硬件结构。它详细描述逻辑架构中定义的概念的实现。类被分配给实现它的构件。构件可能处于不同的阶段：编译时，链接时，运行时。运行时的构件典型地依赖于链接时的构件(静态链接或动态链接)，而链接时的构件依赖于一个或多个编译时的构件(编译它们产生链接构件)。构件和构件间的相关性在 UML 的构件图中显示。

展开图描述硬件节点和节点间的连接。节点指的是物理对象，如一台计算机或一台设备，可用类型和实例来描述。可以用版类来描述节点属于哪一类功能的节点，如版类《printer》。节点间的连接用关联来表示，表明节点间可以相互通信。关联上的版类可以说明使用的协议或物理媒体，如《TCP/IP》。可执行的构件可分配给执行它们的节点，对象(静态或活动的)能被分配给组件。展开图完成所有这些关系的映射。