第7章 软件体系结构与设计模式

- 软件体系结构的基本概念
- 典型的软件体系结构风格
- 特定领域的软件体系结构
- 分布式系统结构
- 体系结构框架
- 设计模式

7.1 软件体系结构的基本概念

- **什么是体系结构**

目前还没有一个公认的关于软件体系结构的定义，许多专家学者从不同角度对软件体系结构进行了描述。Bass、Clements和Kazman给出了如下定义：“一个程序或计算机系统的软件体系结构是指系统的一个或者多个结构。结构中包括软件的构件、构件的外部可见属性以及它们之间的相互关系。外部可见属性则是指软件构件提供的服务、性能、使用特性、错误处理、共享资源使用等。”

这一定义强调在任一体系结构表述中“软件构件”的角色。

7.1 软件体系结构的基本概念

Dewayne Perry和A1exander Wo1f曾这样定义：“软件
体系结构是具有一定形式的结构化元素，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。

虽然软件体系结构的定义在变化，但其意图是清晰的。体系结构设计是一系列决策和基本原理的集合，这些决策的目标在于开发高效的软件体系结构。在体系结构设计中所强调的基本原理是系统的可理解性、可维护性和可扩展性。

7.1 软件体系结构的基本概念

- **体系结构模式、风格和框架的概念**

1．模式

软件设计模式是从软件设计过程中总结出来的，是针对特定问题的解决方案。建筑师C.Alexander对模式给出的经典定义是：**每个模式都描述了一个在我们的环境中不断出现的问题及该问题解决方案的核心**。在软件系统中，可以将模式划分为以下3类。

（1）**体系结构模式**（architectural pattern）：表达了软件系统的基本结构组织形式或者结构方案，包含了一组预定义的子系统，规定了这些子系统的责任，同时还提供了用于组织和管理这些子系统的规则和向导。典型的体系结构模式如**OSI参考模型**。

7.1 软件体系结构的基本概念

（2）**设计模式**（design pattern）：为软件系统的子系统、构件或者构件之间的关系提供一个精炼之后的解决方案，描述了在特定环境下，用于解决通用软件设计问题的构件以及这些构件相互通信时的各种结构。有代表性的设计模式是Erich Gamma及其同事提出的**23种设计模式**。

（3）**惯用法**（idiom）：是与编程语言相关的低级模式，描述如何实现构件的某些功能，或者利用编程语言的特性来实现构件内部要素之间的通信功能。

7.1 软件体系结构的基本概念

2．风格

风格是带有一种倾向性的模式。同一个问题可以有不同的解决问题的方案或模式，但我们根据经验，通常会强烈倾向于采用特定的模式，这就是风格。

每种风格描述**一种系统范畴**，该范畴包括：

（1）**一组构件**（如数据库、计算模块）完成系统需要的某种功能；

（2）**一组连接件**，它们能使构件间实现“通信”、“合作”和“协调”；

（3）**约束**，定义构件如何集成为一个系统；

（4）**语义模型**，它能使设计者通过分析系统的构成成分的性质来理解系统的整体性质。

7.1 软件体系结构的基本概念

- 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。
- 对体系结构风格的研究和实践为大粒度的软件复用提供了可能。

7.1 软件体系结构的基本概念

3．框架

随着应用的发展和完善，某些带有整体性的应用模式被逐渐固定下来，形成特定的框架，包括基本构成元素和关系。**框架**是特定应用领域问题的体系结构模式，框架定义了基本构成单元和关系后，开发者就可以集中精力解决业务逻辑问题。

在组织形式上，框架是一个待实例化的完整系统，定义了软件系统的元素和关系，创建了基本的模块，定义了涉及功能更改和扩充的插件位置。典型的框架例子有MFC框架和Struts框架。

7.1 软件体系结构的基本概念

- **体系结构的重要作用**

体系结构的重要作用体现在以下三个方面 ：

（1）体系结构的表示有助于风险承担者（项目干系人）进行交流。

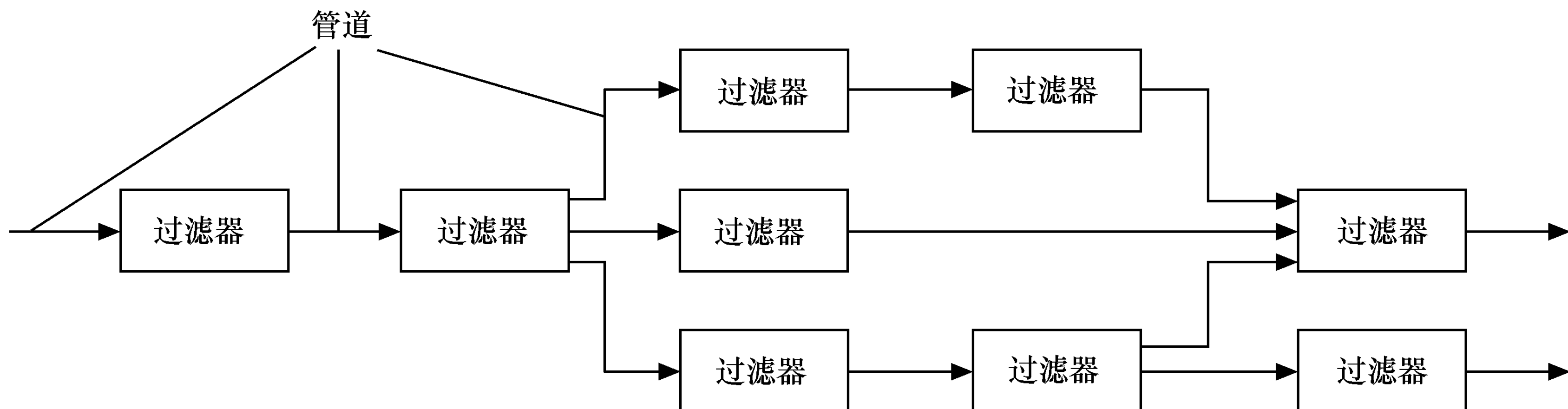
（2）体系结构突出了早期设计决策。

（3）软件体系结构是可传递和可复用的模型。

7.2 典型的体系结构风格

- **数据流风格**

  当输入数据经过一系列的计算和操作构件的变换形成输出数据时，可以应用这种体系结构。管道/过滤器、批处理序列都属于数据流风格。 管道/过滤器结构如下图所示。



管道/过滤器结构

7.2 典型的体系结构风格

从上图可看出，管道/过滤器结构拥有一组被称为过滤器（filter）的构件，这些构件通过管道（pipe）连接，管道将数据从一个构件传送到下一个构件。每个过滤器独立于其上游和下游的构件而工作，过滤器的设计要针对某种形式的数据输入，并且产生某种特定形式的数据输出。

如果数据流退化成为单线的变换，则称为批处理序列（batch sequential）。这种结构接收一批数据，然后应用一系列连续的构件（过滤器）变换它。

7.2 典型的体系结构风格

管道/过滤器风格具有以下优点：

（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）允许设计者将整个系统的输入/输出行为看成是多个过滤器的行为的简单合成。

（3）支持软件复用。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来；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。

7.2 典型的体系结构风格

管道/过滤器风格主要缺点如下：

（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，所以设计者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。

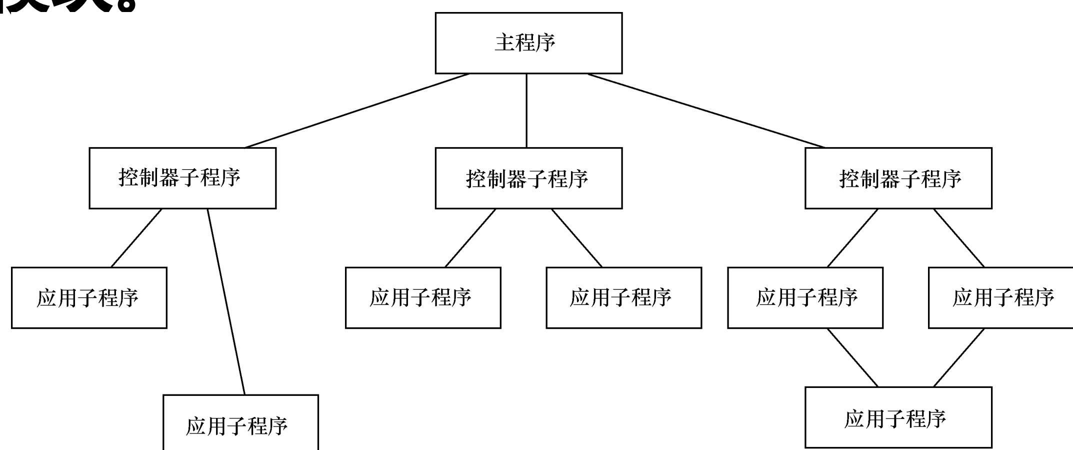
7.2 典型的体系结构风格

- **调用—返回风格**

在此类体系结构中，存在以下3种子风格。

1．主程序/子程序体系结构

这种传统的程序结构将功能分解为一个控制层次，其中“主”程序调用一组程序构件，这些程序构件又去调用别的程序构件，如下图所示。这种结构总体上为树状结构，可以在底层存在公共模块。



7.2 典型的体系结构风格

主程序/子程序体系结构的优点如下:

（1）可以使用自顶向下，逐步分解的方法得到体系结构图，典型的拓扑结构为树状结构。基于定义—使用关系对子程序进行分解，使用过程调用作为程序之间的交互机制。

（2）采用程序设计语言支持的单线程控制。

其主要缺点如下:

（1）子程序的正确性难于判断。需要运用层次推理来判断子程序的正确性，因为子程序的正确性取决于它调用的子程序的正确性。

（2）子系统的结构不清晰。通常可以将多个子程序合成为模块。

7.2 典型的体系结构风格

2．面向对象风格

系统的构件封装了数据和必须应用到该数据上的操作，构件间通过消息传递进行通信与合作。与主程序/子程序的体系结构相比，面向对象风格中的对象交互会复杂一些。面向对象风格与网络应用的需求在分布性、自治性、协作性、演化性等方面具有内在的一致性。

面向对象风格具有以下**优点**:

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

（2）设计者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。

7.2 典型的体系结构风格

其缺点如下:

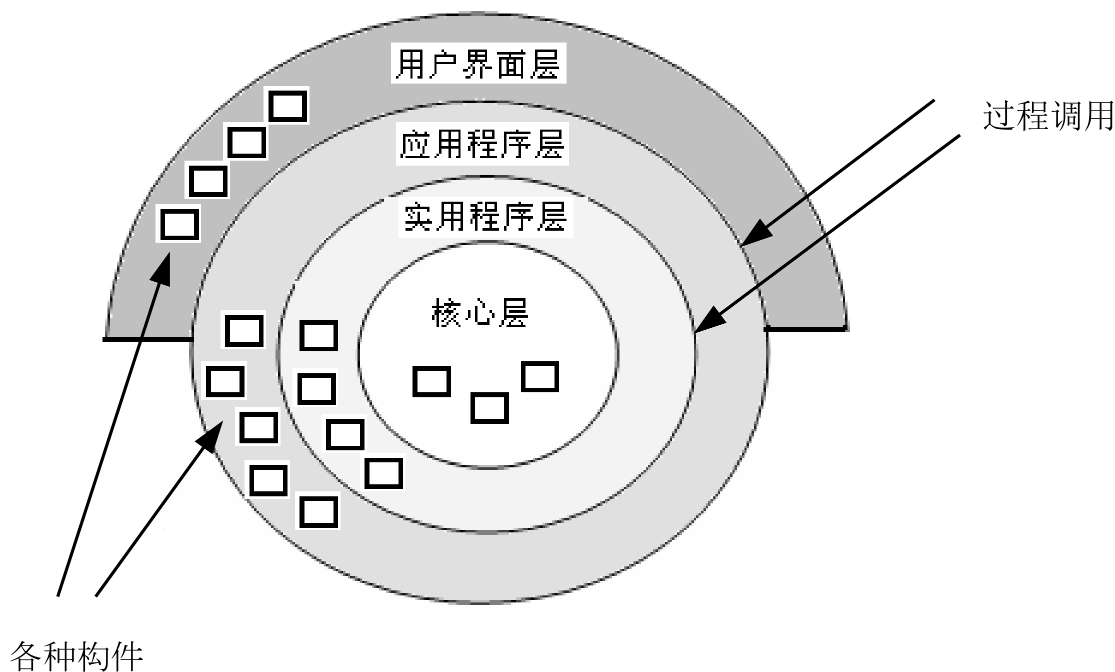
（1）为了使一个对象和另一个对象通过过程调用等进行交互，必须知道对象的标识。只要一个对象的标识改变了，就必须修改所有其他明确调用它的对象。

（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。

7.2 典型的体系结构风格

3．层次结构

层次结构的基本结构如下图所示。在这种体系结构中，整个系统被组织成一个分层结构，每一层为上层提供服务，并作为下一层的客户。



7.2 典型的体系结构风格

这种风格支持基于可增加抽象层的设计。允许将复杂问题分解成一个增量步骤序列的实现。由于每一层最多只影响两层，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，同样为软件复用提供了强大的支持。

层次结构具有以下优点:

（1）支持基于抽象程度递增的系统设计，使设计者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内外层。

7.2 典型的体系结构风格

（3）支持复用。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，从而允许各种不同的实现方法。

其缺点如下:

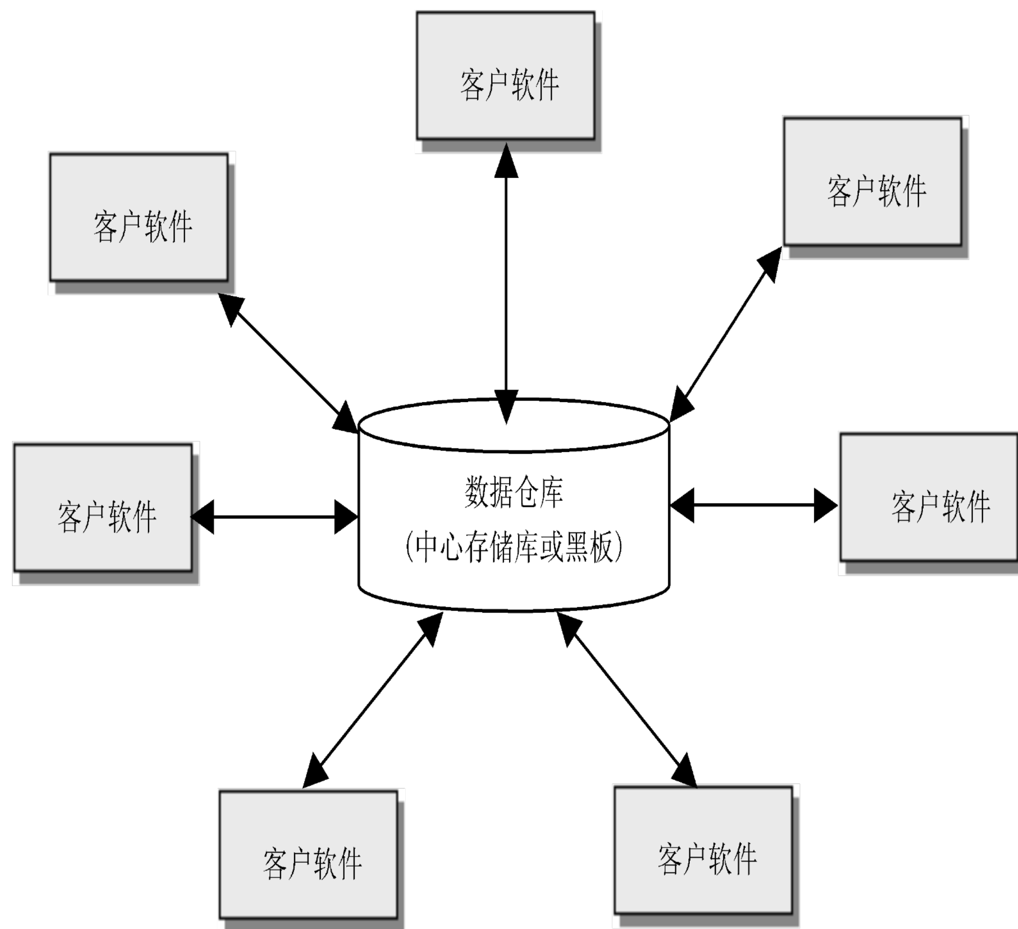
（1）并不是每个系统都可以很容易地划分为分层的模式，甚至即使一个系统的逻辑结构是层次化的，出于对系统性能的考虑，系统设计师不得不把一些低级或高级的功能综合起来。

（2）很难找到一个合适的、正确的层次抽象方法。

7.2 典型的体系结构风格

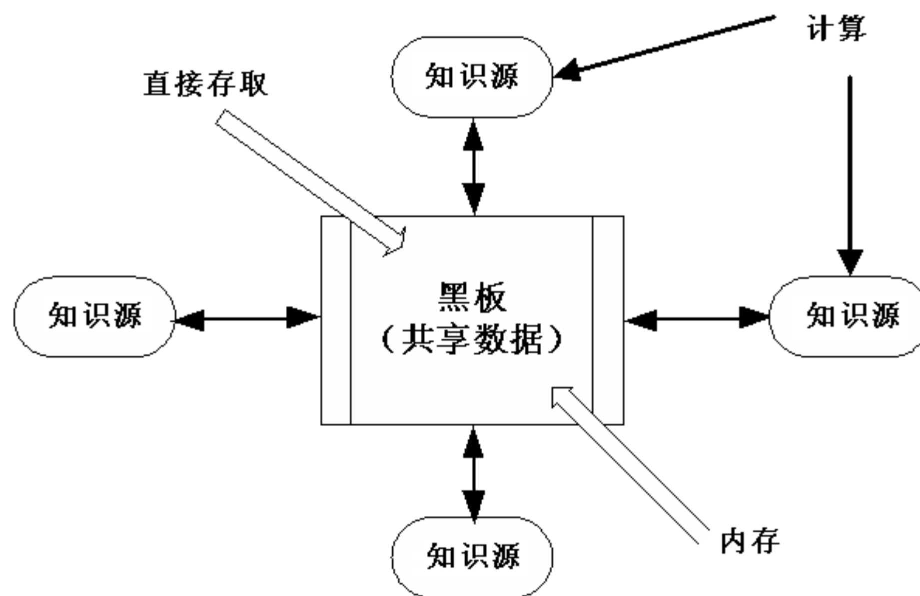
- **仓库风格**

数据库系统、超文本系统和黑板系统都属于仓库风格。在这种风格中，数据仓库（如文件或数据库）位于这种体系结构的中心，其他构件会经常访问该数据仓库，并对仓库中的数据进行增加、修改或删除操作。右图为一个典型的仓库风格的体系结构。



7.2 典型的体系结构风格

上图中,可把中心存储库变换成“黑板”，黑板构件负责协调信息在客户间的传递，当用户感兴趣的数据发生变化时，它将通知客户软件。黑板系统的组成如下图所示。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。



7.3 特定领域的软件体系结构

- **特定的应用还需要特定的体系结构模型。这些体系结构模型称为领域相关的体系结构。**
- **有两种领域相关的体系结构模型：类属模型（generic model）和参考模型（reference model）。**

7.3 特定领域的软件体系结构

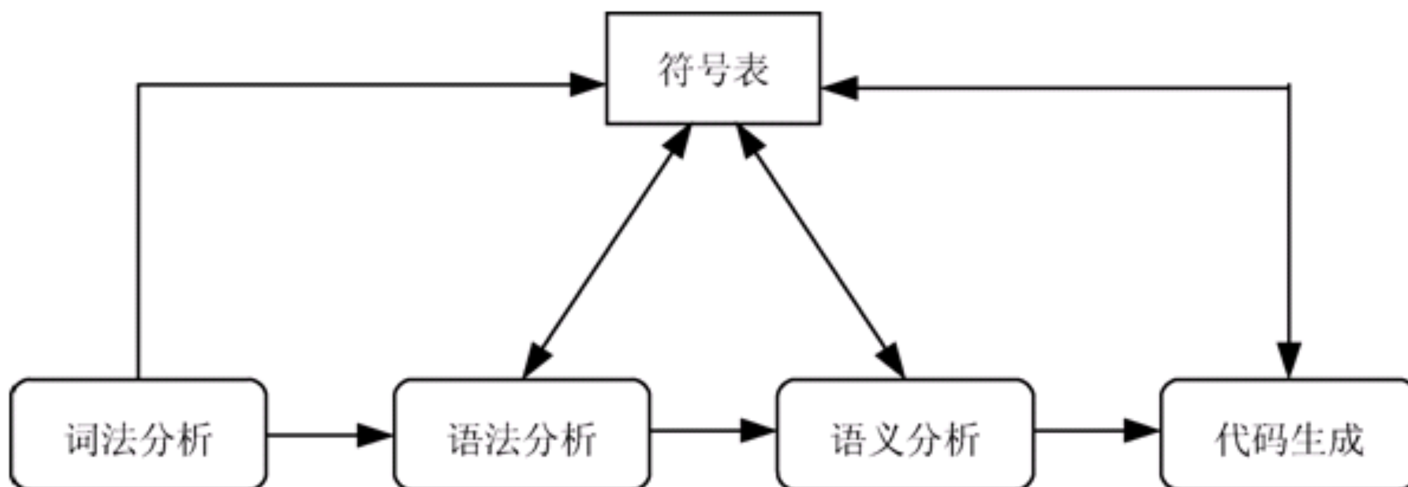
- **类属模型**

  ➢类属模型是从许多实际系统中抽象出来的一般模型，它封装了这些系统的主要特征。

  ➢例如，许多图书馆开发了自己的图书馆馆藏/流通系统，若把它们的共同功能抽取出来并创建一个让所有图书馆都认可的系统体系结构模型，这就是类属模型。

7.3 特定领域的软件体系结构

- **类属模型的一个最著名的例子是编译器模型，由这个模型已开发出了数以千计的编译器。**



7.3 特定领域的软件体系结构

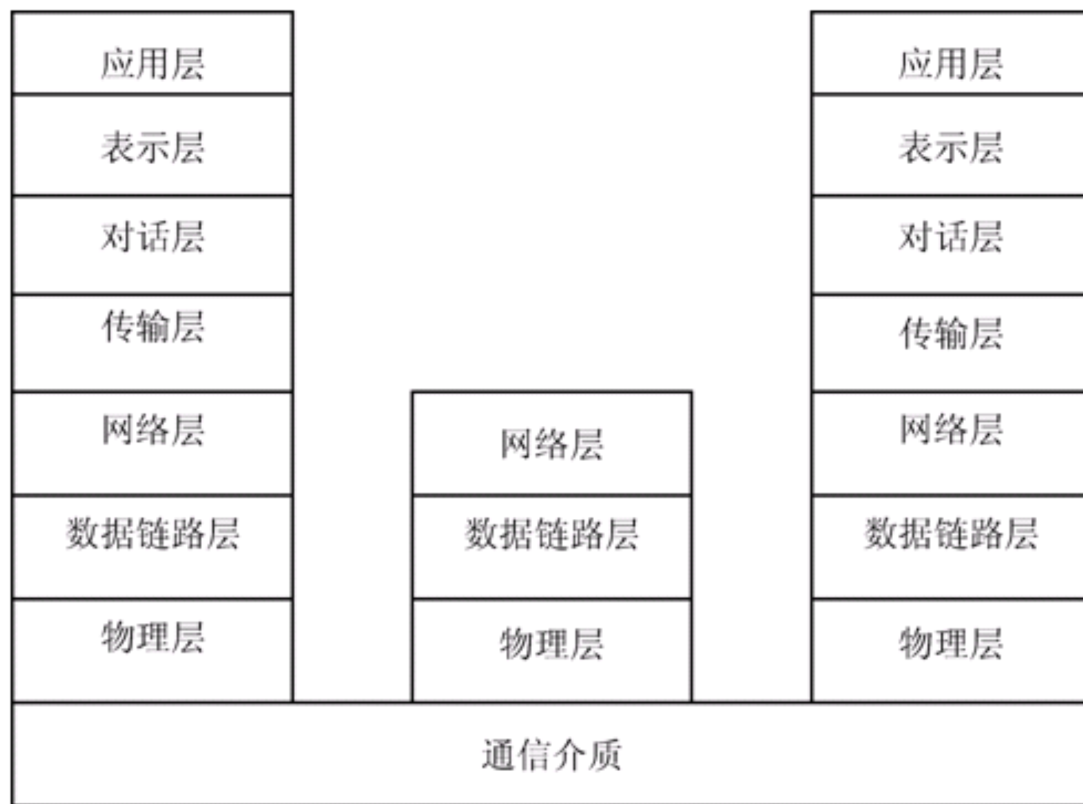
- **参考模型**

➢ 参考模型源于对应用领域的研究，它描述了一个理想化的包含了系统应具有的所有特征的软件体系结构。

➢ 它是更抽象且是描述一大类系统的模型，并且也是对设计者有关某类系统的一般结构的指导。

7.3 特定领域的软件体系结构

- **参考模型的典型例子是开放式系统互联（OSI）参考模型。**



OSI 的参考模型

7.3 特定领域的软件体系结构

以上两种不同类型的模型之间并不存在严格的区别，也可以将类属模型视为参考模型。

- 区别之一是类属模型可以直接在设计中复用，而参考模型一般是用于领域概念间的交流和对可能的体系结构做出比较。
- 另外，类属模型通常是经过“自下而上”地对已有系统的抽象，而参考模型是“由上到下”地产生的。

7.4 分布式系统结构

- **在集中式计算技术时代广泛使用的是大型机/小型机计算模型。**
- **20世纪80年代以后，集中式结构逐渐被以PC为主的微机网络所取代。个人计算机和工作站的采用，永远改变了大型机/小型机计算模型，从而产生了分布式计算模型。**

7.4 分布式系统结构

- 分布式计算模型主要具有以下优点：

(1) 资源共享。分布式系统允许硬件、软件等资源共享使用。

(2) 经济性。

(3) 性能与可扩展性。

(4) 固有分布性。

(5) 健壮性。

7.4 分布式系统结构

- **多处理器体系结构**

分布式系统的一个最简单的模型是多处理器系统，系统由许多进程组成，这些进程可以在不同的处理器上并行运行，可以极大地提高系统的性能。

由于大型实时系统对响应时间要求较高，这种模型在大型实时系统中比较常见。大型实时系统需要实时采集信息，并利用采集到的信息进行决策，然后发送信号给执行机构。虽然，信息采集、决策制定和执行控制这些进程可以在同一台处理器上统一调度执行，但使用多处理器能够提高系统性能。

7.4 分布式系统结构

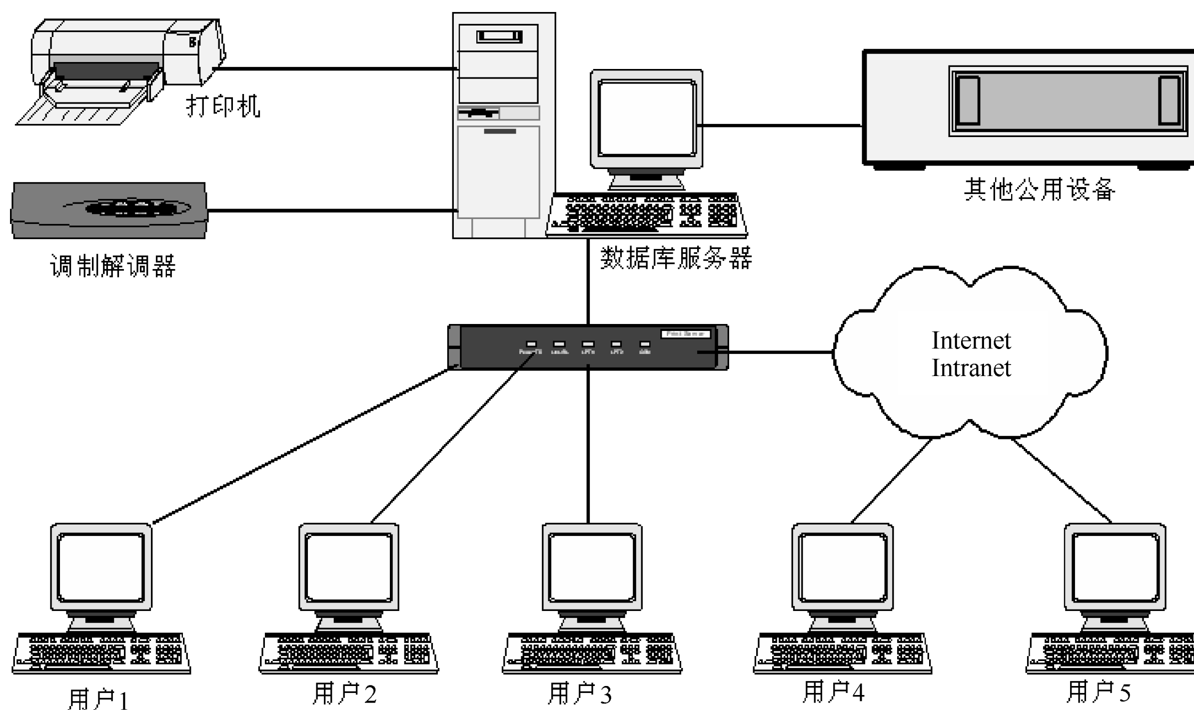
- **客户/服务器体系结构**

　　客户机/服务器（client/server，C/S）体系结构是基于资源不对等，且为实现共享而提出来的，由服务器、客户机和网络三部分组成。

　　在C/S体系结构中，客户机可以通过远程调用来获取服务器提供的服务，因此，客户机必须知道可用的服务器的名字及它们所提供的服务，而服务器不需要知道客户机的身份，也不需要知道有多少台服务器在运行。

7.4 分布式系统结构

传统的C/S体系结构分为两层。在这种体系结构中，一个应用系统被划分为客户机和服务器两部分。典型的两层C/S体系结构如下图所示。



7.4 分布式系统结构

两层C/S体系结构可以有两种形态：

（1）瘦客户机模型。在瘦客户机模型中，数据管理部分和应用逻辑都在服务器上执行，客户机只负责表示部分。

瘦客户机模型的主要缺点：

- **它将繁重的处理负荷都放在了服务器和网络上，服务器负责所有的计算，这将增加客户机和服务器之间的网络流量。**
- **目前个人计算机所具有的处理能力在瘦客户机模型中用不上。**

7.4 分布式系统结构

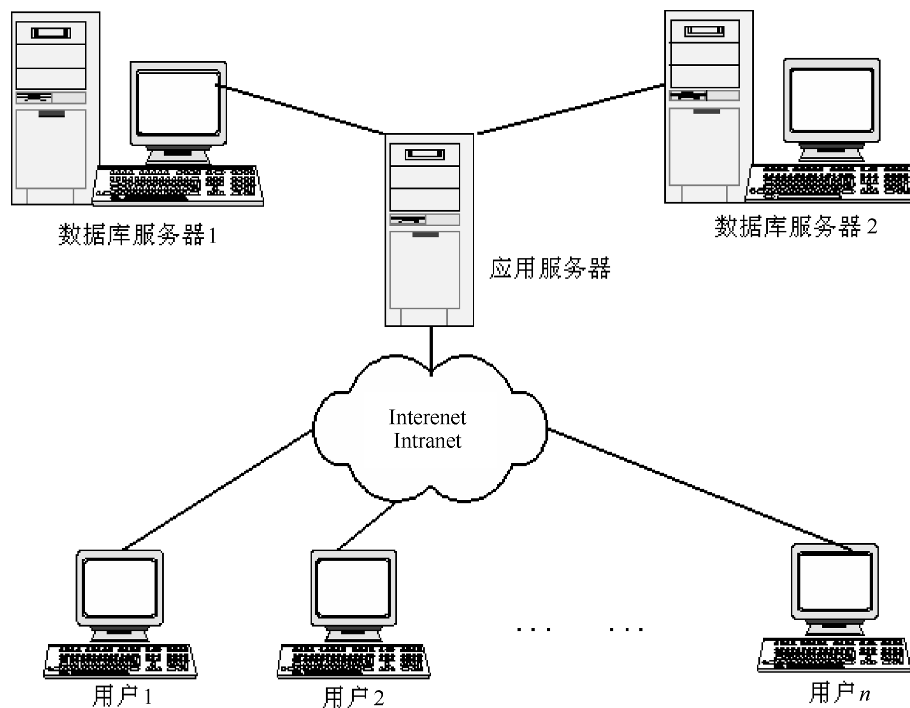
（2）**胖客户机模型**。在这种模型中，服务器只负责对数据的管理。客户机上的软件实现应用逻辑和与系统用户的交互。

胖客户机模型能够利用客户机的处理能力，比瘦客户机模型在分布处理上更有效。但另一方面，随着企业规模的日益扩大，软件的复杂程度不断提高，胖客户机模型逐渐暴露出了以下缺点：

- 开发成本较高。
- 用户界面风格不一，使用繁杂，不利于推广使用。
- 软件移植困难。
- 软件维护和升级困难。

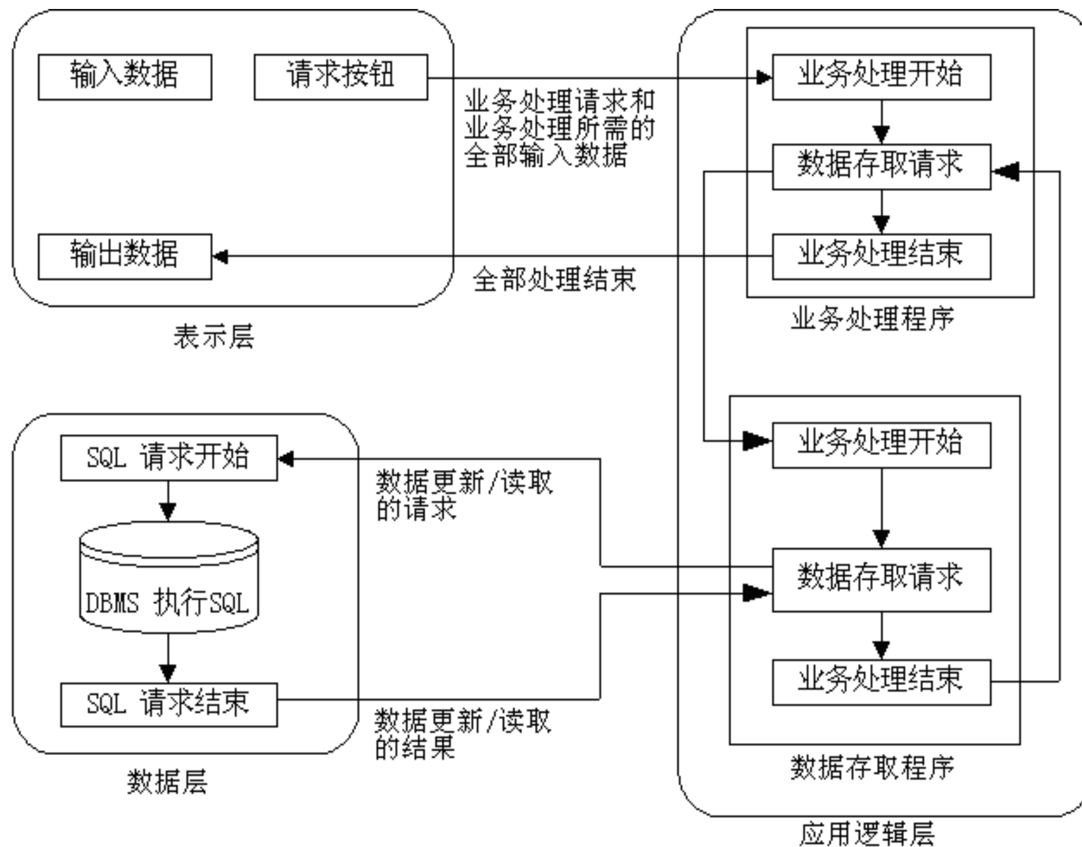
7.4 分布式系统结构

为了解决以上问题，三层C/S体系结构应运而生。三层C/S体系结构中增加了应用服务器。可以将整个应用逻辑驻留在应用服务器上，而只有表示层存在于客户机上。



7.4 分布式系统结构

三层C/S体系结构将整个系统分成表示层、应用逻辑层和数据层三个部分，其数据处理流程如下图所示。



7.4 分布式系统结构

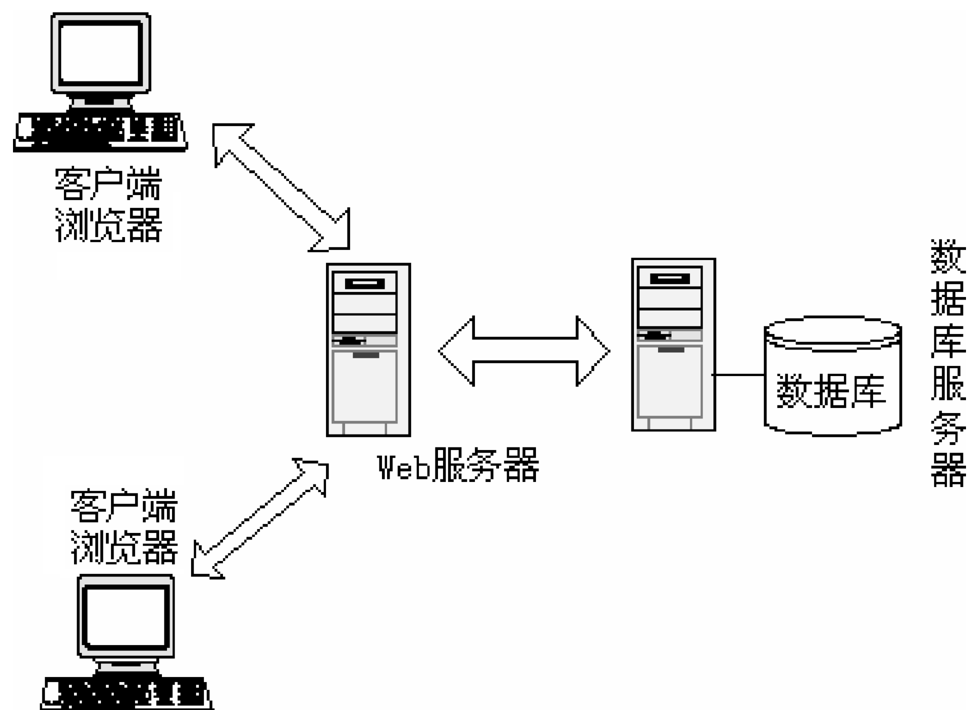
（1）**表示层**：表示层是应用系统的用户界面部分，担负着用户与应用程序之间的对话功能。它用于检查用户从键盘等输入的数据，显示应用程序输出的数据，一般采用图形用户界面（graphic user interface， GUI）。

（2）**应用逻辑层**：应用逻辑层为应用系统的主体部分，包含具体的业务处理逻辑。通常在功能层中包含有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。

（3）**数据层**：数据层主要包括数据的存储及对数据的存取操作，一般选择关系型数据库管理系统（RDBMS）。

7.4 分布式系统结构

浏览器/服务器（browser/server，B/S）风格是三层体系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。B/S体系结构如下图所示。



7.4 分布式系统结构

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结构是一种全新的软件体系结构。

B/S体系结构具有以下优点：

（1）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。

（2）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、联网和统一服务的最现实的开放性基础。

7.4 分布式系统结构

与C/S体系结构相比，B/S体系结构也有许多不足之处。

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

（3）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（OLTP）应用。

7.4 分布式系统结构

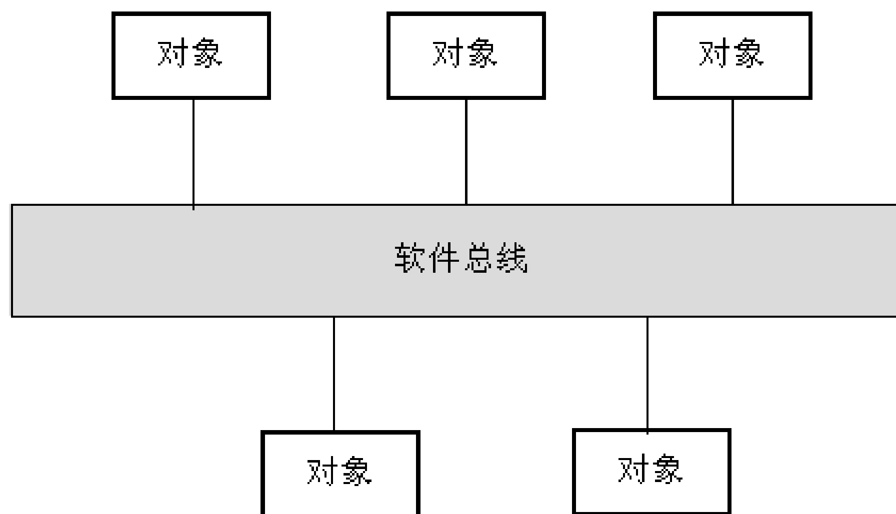
- **分布式对象体系结构**

在客户机/服务器模型中，客户机和服务器的地位是不同的。为了消除客户机与服务器之间的差别，提高系统的伸缩性以及有效地均衡负载，可采用分布式对象体系结构来设计系统。

分布式对象的实质是在分布式异构环境下建立应用系统框架和对象构件，它将应用服务分割成具有完整逻辑含义的独立子模块（称为**构件**），各个子模块可放在同一台服务器或分布在多台服务器上运行，模块之间通过中间件互相通信。

7.4 分布式系统结构

通常将这个中间件称为软件总线或对象请求代理，它的作用是在对象之间提供一个无缝接口。



分布式对象技术的应用目的是为了降低主服务器的负荷、共享网络资源、平衡网络中计算机业务处理的分配，提高计算机系统协同处理的能力，从而使应用的实现更为灵活。

7.4 分布式系统结构

- **分布式对象技术的基础是构件。构件是一些独立的代码封装体，在分布计算的环境下可以是一个简单的对象，但大多数情况下是一组相关的对象组合体，提供一定的服务。**
- **分布式环境下，构件是一些灵活的软件模块，它们可以位置透明、语言独立和平台独立地互相发送消息，实现请求服务。**
- **构件之间并不存在客户机与服务器的界限，接受服务者扮演客户机的角色，提供服务者就是服务器。**

7.4 分布式系统结构

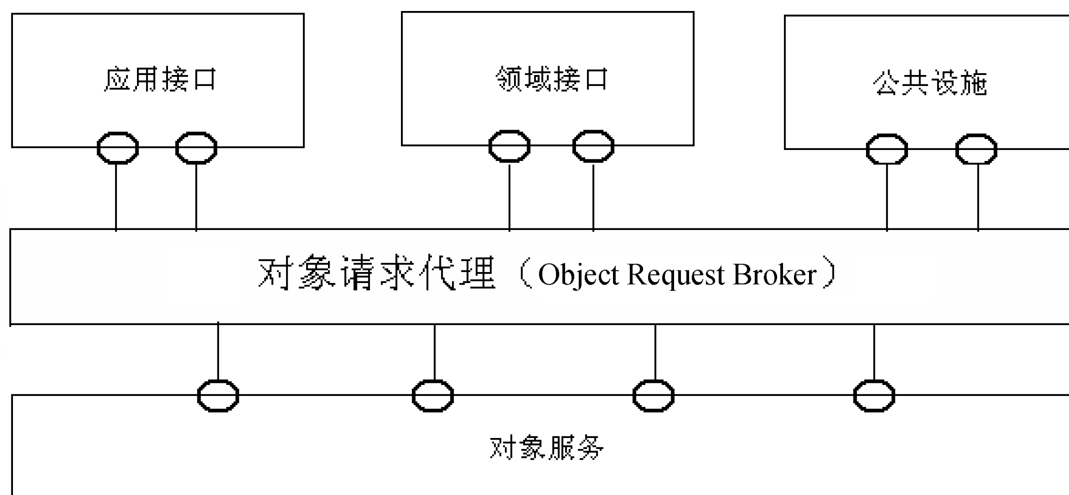
- **当前主流的分布式对象技术规范有OMG的CORBA、Microsoft公司的.NET和Sun公司的J2EE。**
- **它们都支持服务端构件的开发，都有其各自的特点。**

7.4 分布式系统结构

- **代理**

代理可以用于构建带有隔离组件的分布式软件系统，该软件通过远程服务调用进行交互。代理者负责协调通信，诸如转发请求以及传递结果和异常等。

1991年，OMG基于面向对象技术，给出了以对象请求代理（ORB）为中心的分布式应用体系结构。



7.4 分布式系统结构

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。在ORB之上有4个对象接口：

（1）对象服务：定义加入ORB的系统级服务，如安全性、命名和事务处理，它们是与应用领域无关的。

（2）公共设施：水平级的服务，定义应用程序级服务。

（3）领域接口：面向特定的领域。

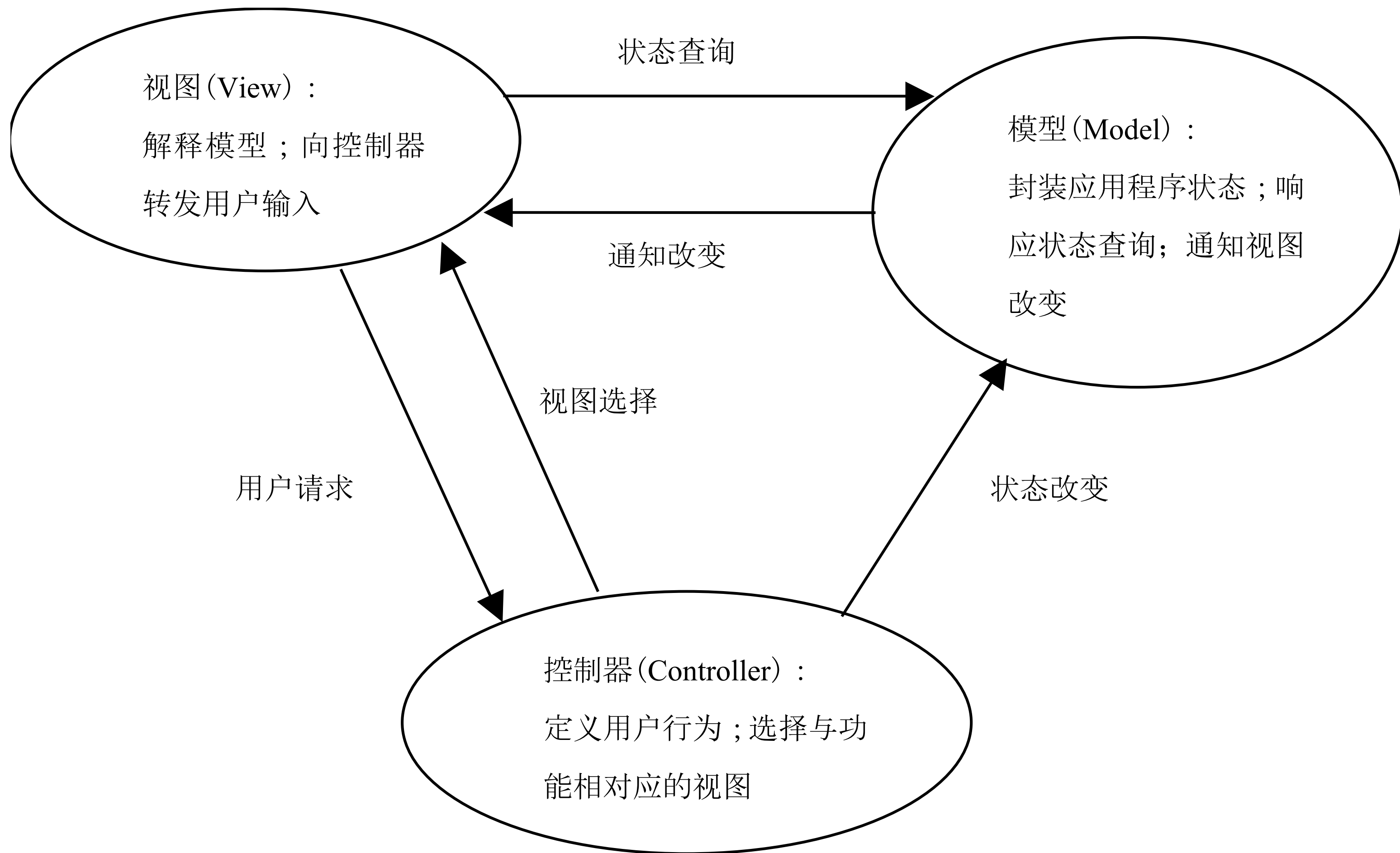
（4）应用接口：面向指定的现实世界应用。是指供应商或用户借助于ORB、公共对象服务及公共设施而开发的特定产品。

7.5 体系结构框架

- **MVC框架**

MVC框架即模型—视图—控制器（model-view-controller）框架，它强调将用户输入、数据模型和数据表示的方式分开设计，一个交互式应用系统由模型、视图和控制器3个部件组成，分别对应于内部数据、数据表示和输入/输出控制部分。

7.5 体系结构框架



MVC框架

7.5 体系结构框架

1. 模型对象

模型对象独立于外在显示内容和形式，代表应用领域中的业务实体和业务规则，是**整个模型的核心**。模型对象的变化通过事件处理通知视图和控制器对象。

2. 视图对象

视图对象代表GUI对象，并且以用户需要的格式表示模型状态，是交互系统与外界的接口。视图对象可以包含子视图，子视图用于显示模型的不同部分。通常，每个视图对象对应一个控制器对象。

7.5 体系结构框架

3. 控制器对象

控制器对象代表鼠标和键盘事件。它处理用户的输入行为并给模型发送业务事件，再将业务事件解析为模型应执行的动作；同时，模型的更新与修改也将通过控制器来通知视图，从而保持各个视图与模型的一致性。

7.5 体系结构框架

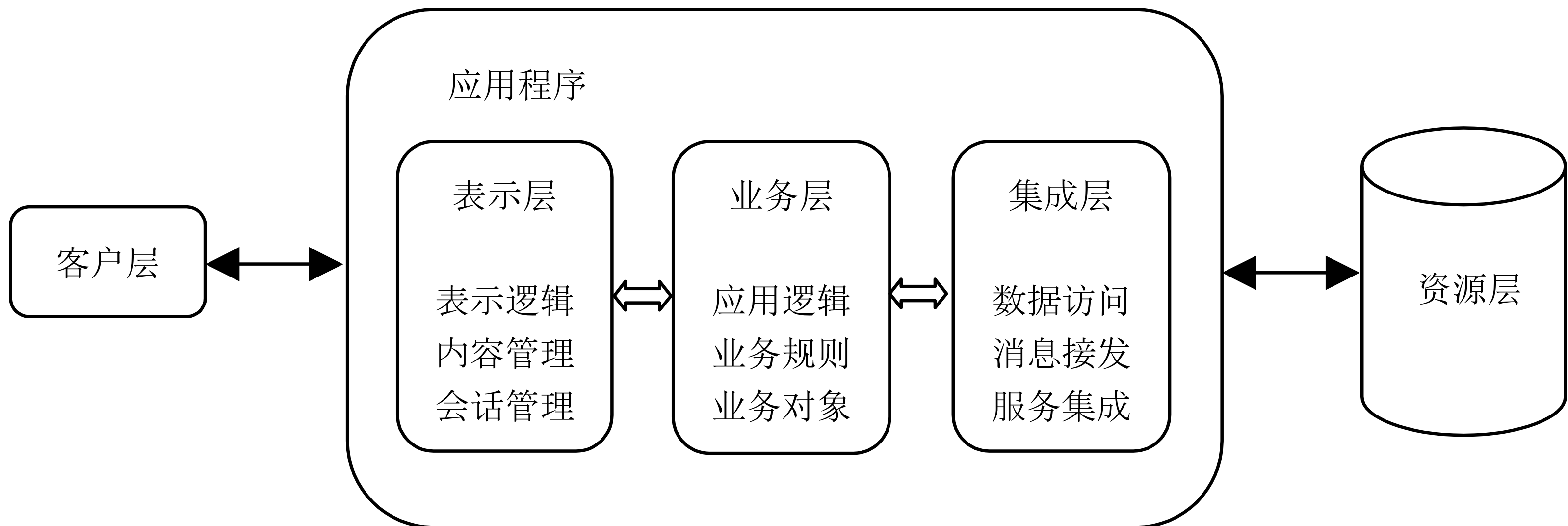
MVC的处理过程为：首先控制器接收用户的请求，并决定应该调用哪个模型来进行处理；然后模型用业务逻辑来处理用户的请求并返回数据；最后控制器用相应的视图格式化模型返回的数据，并通过表示层呈现给用户。

其中，模型是核心数据和功能，视图只关心显示数据，控制只关心用户输入，这种结构由于将数据和业务规则从表示层分开，因此可以最大化地重用代码。

7.5 体系结构框架

- **J2EE体系结构框架**

J2EE的核心体系结构就是在MVC框架的基础上进行扩展得到的，如下图所示。



J2EE的核心体系结构框架

7.5 体系结构框架

从上图可看出，J2EE模型是分层结构，中间的3层（表示层，业务层，集成层）包含应用程序构件，客户层和资源层处于应用程序的外围。

- **客户层：用户通过客户层与系统交互。该层可以是各种类型的客户端。例如，可编程客户端（如基于Java Swing的客户端或applet），纯Web浏览器客户端，WML移动客户端等。**
- **资源层：资源层可以是企业数据库，电子商务解决方案中的外部企业系统，或者是外部SOA服务。数据可以分布在多个服务器上。**

7.5 体系结构框架

- **表示层**：也称为Web层或服务器端表示层，用户通过表示层来访问应用程序。在基于Web的应用系统中，表示层由用户界面代码和运行于Web服务器或应用服务器上的过程组成。参考MVC框架，表示层包括视图构件和控制器构件。
- **业务层**：业务层包含表示层中的控制器构件没有实现的一部分应用逻辑。它负责确认和执行企业范围内的业务规则和事务，并管理从资源层加载到应用程序高速缓存中的业务对象。
- **集成层**：集成层负责建立和维护与数据源的连接。例如，通过JDBC与数据库进行通信，利用Java消息服务（JMS）与外部系统联合。

7.5 体系结构框架

• PCMEF与PCBMER框架

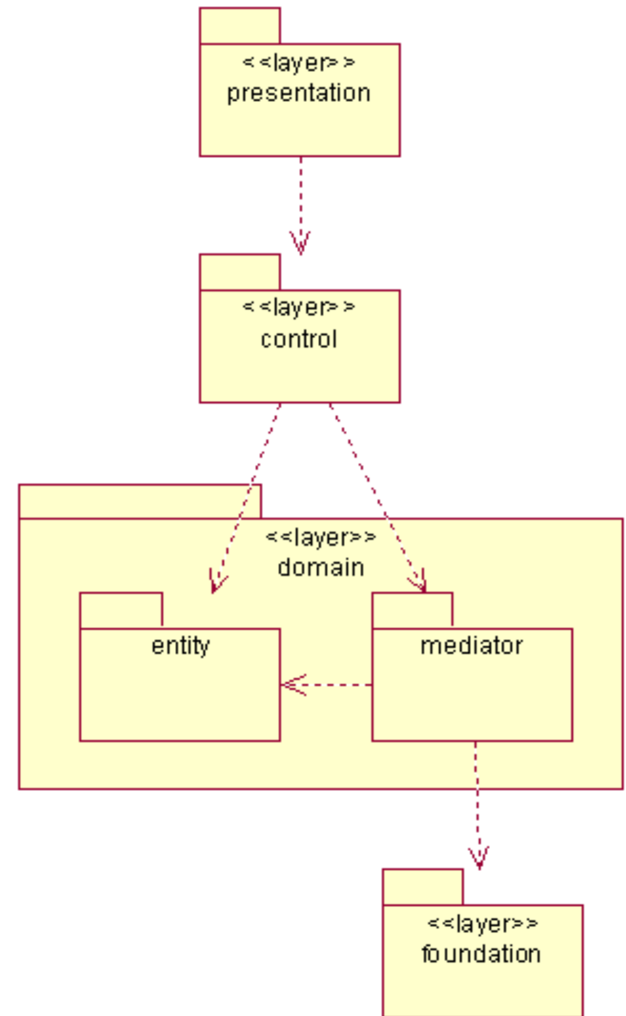
1．PCMEF框架

表示—控制—中介者—实体—基础（presentation-control-mediator-entity-foundation，PCMEF）是一个垂直层次的分层体系结构框架。每一层是可以包含其他包的包。PCMEF框架包含4层：表示层、控制层、领域层和基础层。领域层包含两个预定义包：实体（entity）包和中介者（mediator）包。

PCMEF框架中包的依赖性主要是向下依赖性。表示层依赖于控制层，控制层依赖于领域层，中介者包依赖于实体包和基础层，如下图所示。

7.5 体系结构框架

- **表示层:** 包含定义GUI对象的类。
- **控制层:** 处理表示层的请求，负责大多数程序逻辑、算法、主要计算以及为每个用户维持会话状态。
- **领域层:** 其实体包处理控制请求，中介者包用于创建一个协调实体类和基础类的通信通道。
- **基础层:** 负责与数据库和Web服务的所有通信。

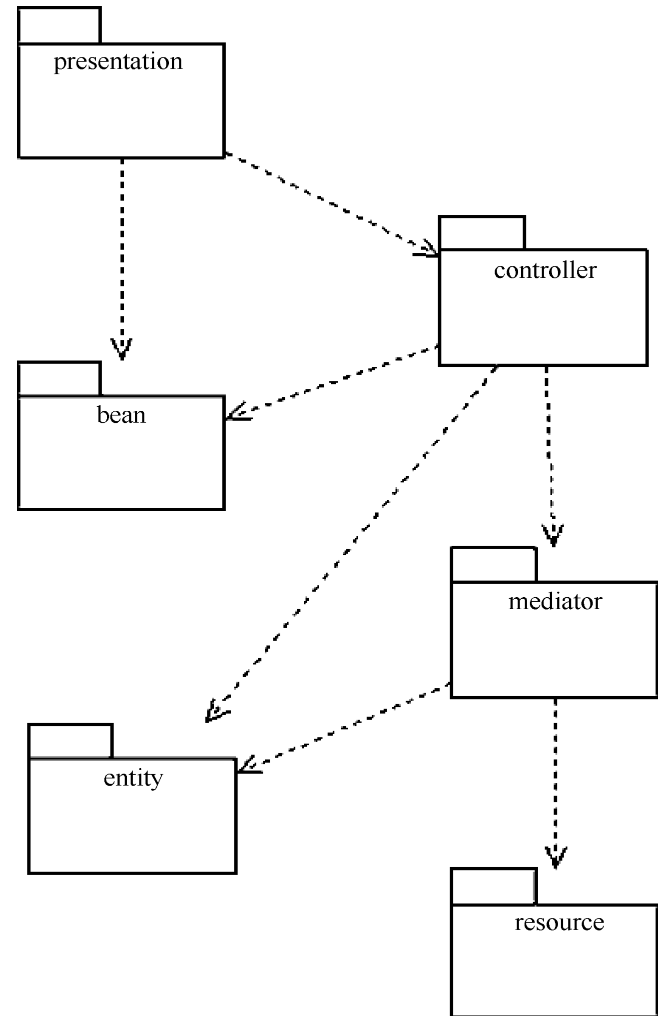


PCMEF框架

7.5 体系结构框架

2. PCBMER框架

PCBMER 框架由PCMEF框架扩展而成，代表着表示—控制器—Bean—中介者—实体—资源（presentation-control-bean-mediator-entity-resource，PCBMER）。其核心体系结构框架如右图所示。



PCBMER的核心框架

7.5 体系结构框架

在上图中，把层表示为UML包（子系统，层），带箭头的虚线表示依赖关系。例如，表示层依赖控制器层和bean层，控制器层依赖bean层。PCBMER的层次不是严格线性的，上层可以依赖多个相邻下层。

- **bean层**：表示那些预先确定要呈现在用户界面上的数据类和值对象。除了用户输入外，bean数据由实体对象（实体层）创建。
- **表示层**：表示屏幕以及呈现bean对象的UI对象。
- **控制器层**：表示应用逻辑。
- **实体层**：响应控制器和中介者。
- **中介者层**：建立了充当实体类和资源类媒介的通信管道。
- **资源层**：负责所有与外部持久数据资源（数据库、Web服务等）的通信。

7.6 设计模式

- **面向对象设计模式最初出现于70年代末80年代初。**
- **Erich Gamma等4人合著的“Design Patterns: Elements of Reusable Object-Oriented Software”被认为是设计模式方面的经典著作。**
- **目前，设计模式已经被广泛应用于多种领域的软件设计和构造中，许多当代的先进软件中已大量采用了软件设计模式的概念。**

7.6 设计模式

- 一般来说，一个模式有4个基本的要素：

(1) **模式名称**：用于描述模式的名字，说明模式的问题、解决方案和效果。

(2) **问题**：说明在何种场合使用模式。

(3) **解决方案**：描述设计的组成成分、它们之间的相互关系、各自的职责和合作方式。

(4) **效果**：描述了模式使用的效果及使用模式应当权衡的问题。

抽象工厂

(1) 目的：提供一个接口用以创建一个相联系或相依赖的对象族，而无须指定它们的具体类。

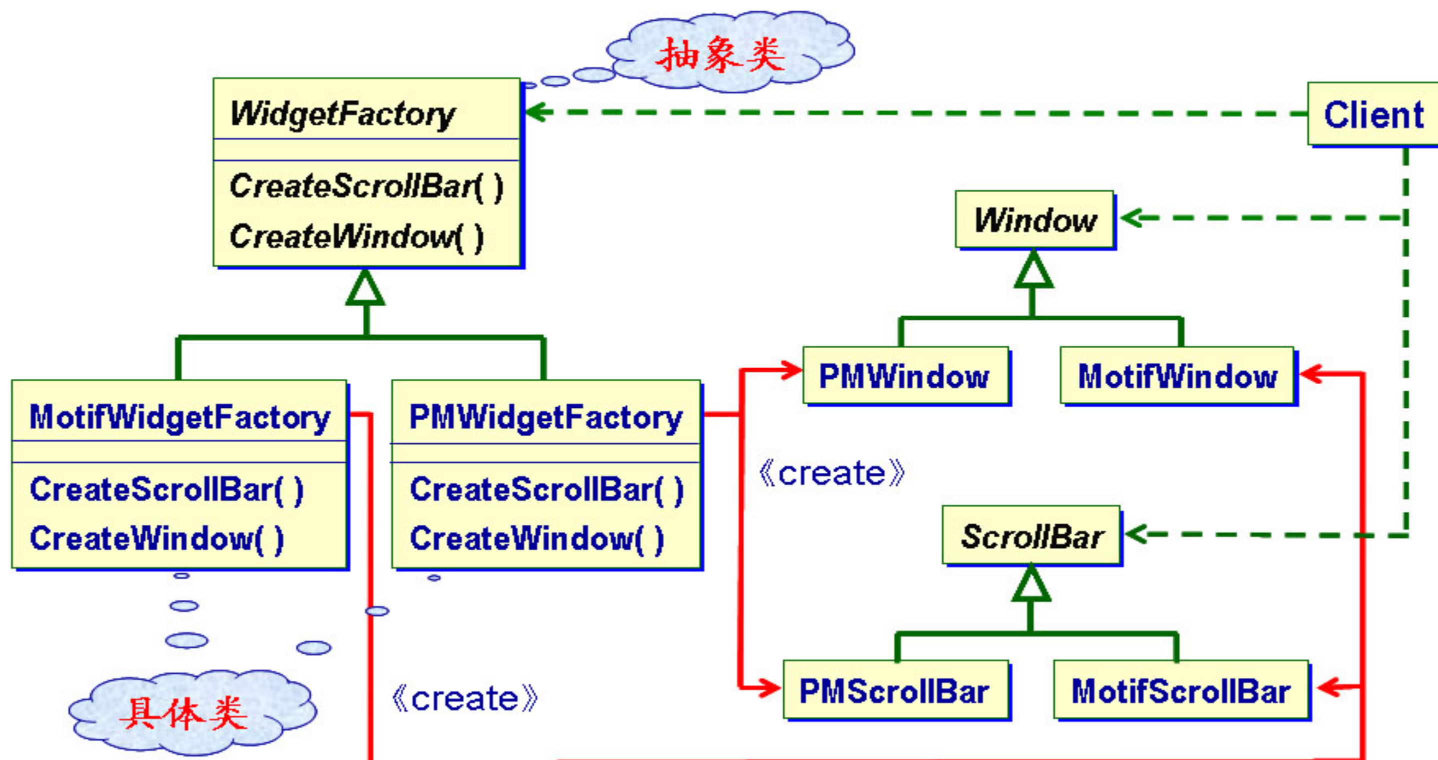
(2) 思路：例如，在创建可支持多种GUI标准（如Motif和Persentation Manager）的绘图用户界面工具包时，因为不同的GUI标准会定义出不同外观及行为的“用户界面组件”（widget），如滚动条、按钮、视窗等。为了能够囊括各种GUI标准，应用程序不能把组件写死，不能限制到特定GUI风格的组件类，否则日后很难换成其他GUI风格的组件。

抽象工厂

- **解决方法是：先定义一个抽象类WidgetFactory（用斜体字区分抽象类），这个类声明了创建各种基本组件的接口，再逐一替各种基本组件定义相对应的抽象类，如ScrollBar、Window等，让它们的具体子类来真正实现特定的GUI标准。**

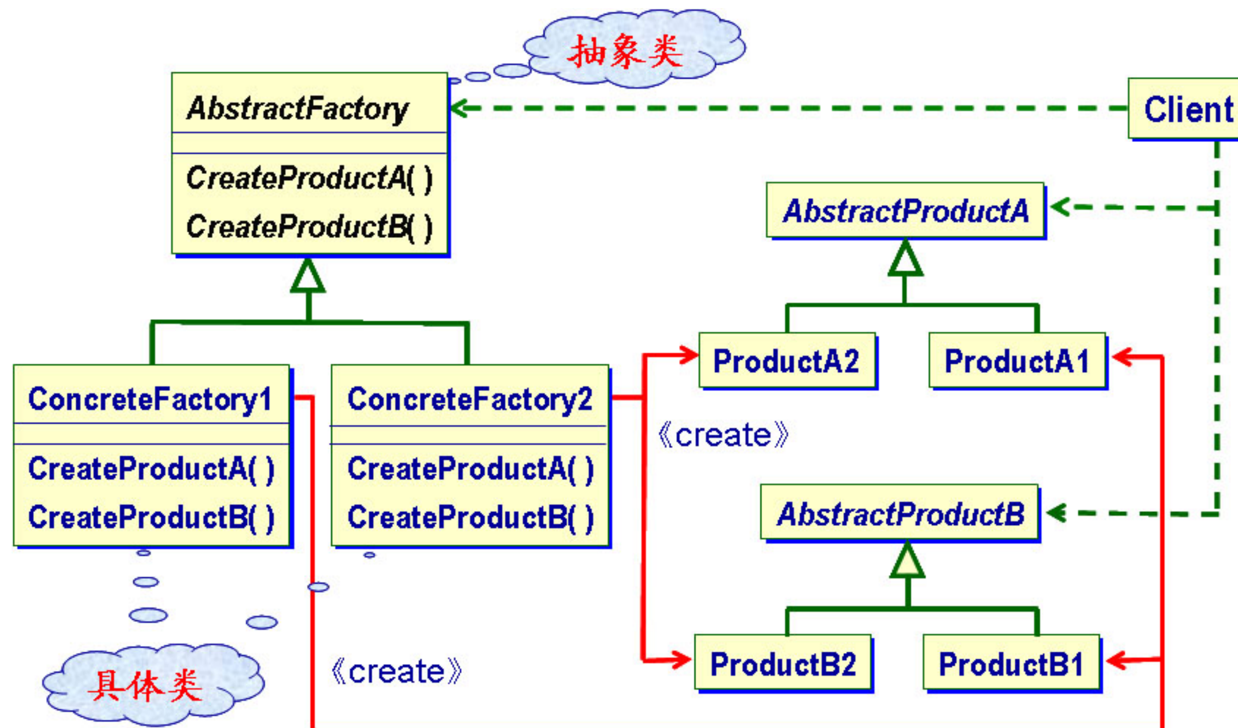
抽象工厂

- 可支持多种GUI标准的绘图用户界面工具包的结构图



抽象工厂

(3) 结构：抽象工厂模式的结构如图所示。



抽象工厂

(4) 参与者职责

a) 抽象工厂类（AbstractFactory）：声明创建抽象产品对象的操作的接口。

b) 具体工厂类（ConcreteFactory）：实现产生具体产品对象的操作。

c) 抽象产品类（AbstractProduct）：声明一种产品对象的接口。

d) 具体产品类（ConcreteProduct）：定义将被相应的具体工厂类产生的产品对象，并实现抽象产品类接口。

e) 客户（Client）：仅使用由抽象工厂类和抽象产品类声明的接口。

抽象工厂

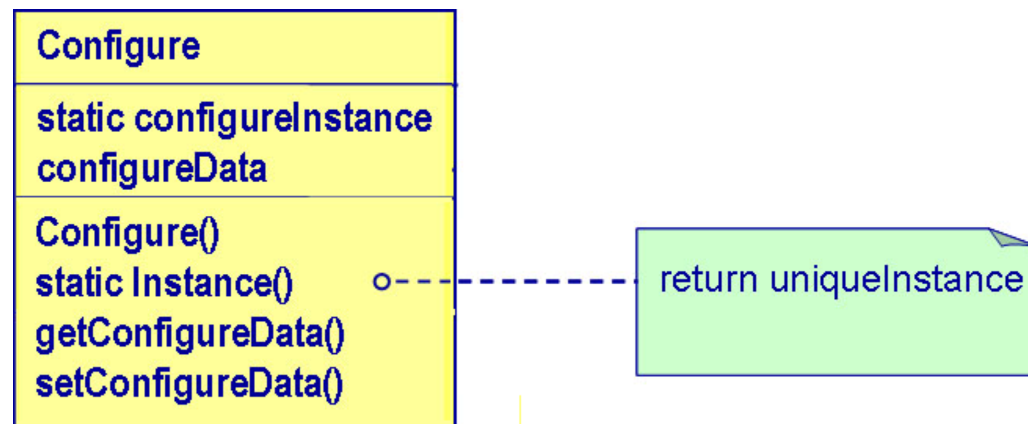
(5) 协作

- **在执行时，AbstractFactory将产品交给ConcreteFactory创建。**
- **ConcreteFactory类的实例只有一个，专门针对某种特定的实现标准，建立具体可用的产品对象。**
- **如果想要建立其他标准的产品对象，客户程序就得改用另一种ConcreteFactory。**

单件

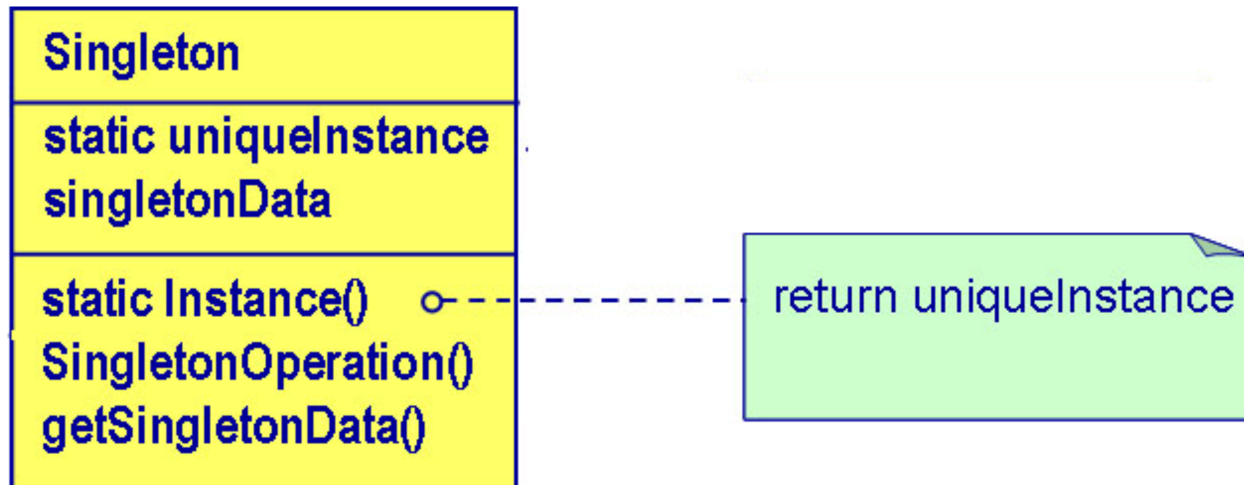
(1) 目的：一个类只有一个实例并提供一个访问它的全局访问点。该实例应在系统生存期中都存在。

(2) 思路：例如，通常情况下，用户可以对应用系统进行配置，并将配置信息保存在配置文件中，应用系统在启动时首先将配置文件加载到内存中，这些内存配置信息应该有且仅有一份。应用单件模式可以保证Configure类只能有一个实例。



单件

(3) 结构：单件模式的结构如图所示。



单件

(4) 参与者职责

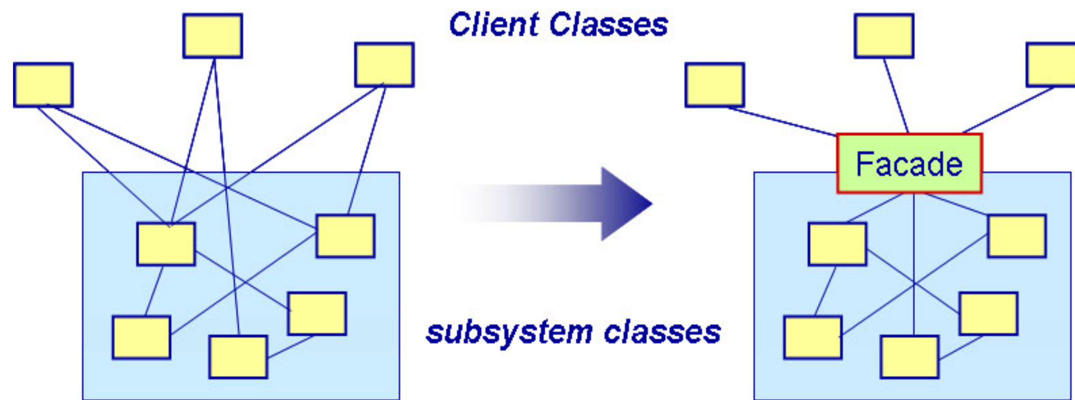
a) 单件（Singleton）：能够创建它唯一的实例；同时定义了一个Instance操作，允许外部存取它唯一的实例。Instance是一个静态成员函数

(5) 协作：客户只能通过Singleton的Instance()存取这唯一的实例。

外观

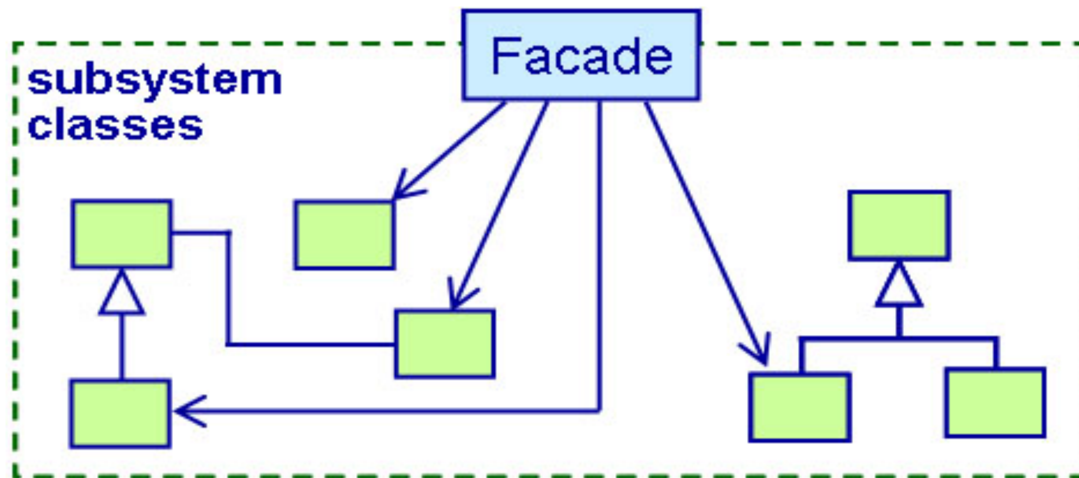
(1) 目的：给子系统中的一组接口提供一套统一的高层界面，使得子系统更容易使用。

(2) 思路：将系统划分为若干子系统，虽然可以降低整体的复杂性，但还需设法降低子系统之间的通信和相互的依赖性。一种方法就是引进一个外观（facade）对象，为子系统内各种设施提供一个简单的单一界面。



外观

(3) 结构：外观模式的结构如图所示。



外观

(4)参与者职责

- **a)外观（Façade）：知道子系统中哪个类负责处理哪种信息；并负责把外界输入的信息转交给适当的子系统对象。**
- **b)子系统中的类（subsystem classes）：实现子系统的功能；处理Facade对象分派的工作；如果不受Facade的控制，则也不会有返回Facade的引用存在。**

(5)协作：使用Facade的客户不用直接访问子系统对象。外界想与子系统交互时，把信息传送给Facade，Facade再把这些信息转交给适当的子系统对象。虽然实际处理工作是子系统对象在做，但Facade会居中做接口转换工作。

适配器

(1)目的：适配器模式将一个类的接口转换为客户期望的另一种接口，使得原本不匹配的接口而无法合作的类可以一起工作。

(2)思路：有时要将两个没有关系的类组合在一起使用，一种解决方案是修改各自类的接口，另一种办法是使用Adapter模式，在两种接口之间创建一个混合接口。

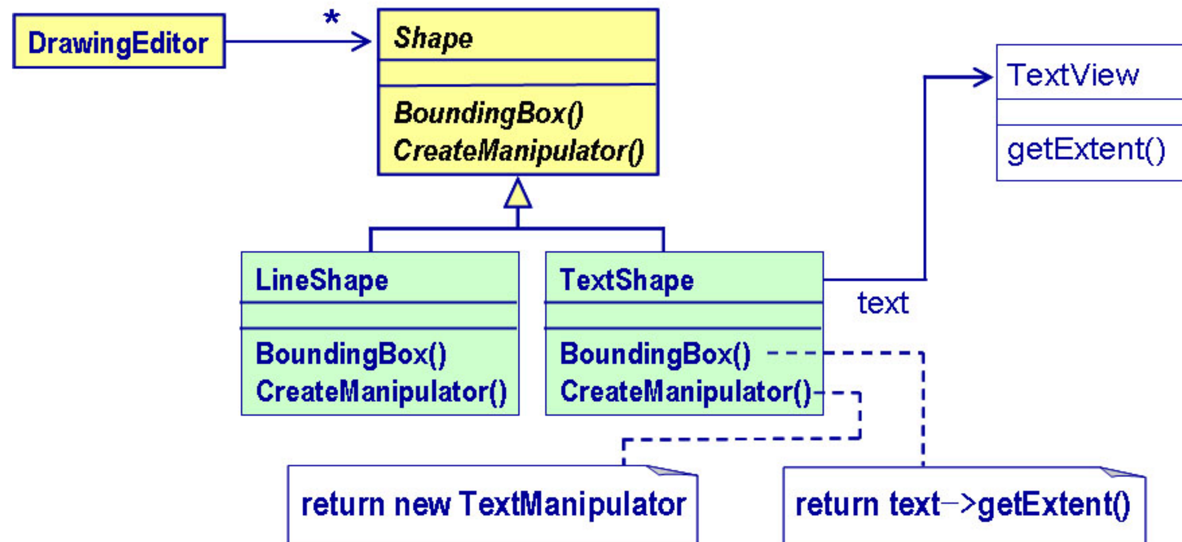
例如，设有一个图形编辑器，可画直线、多边形、文本等。它的接口定义成抽象类Shape，它的子类负责画各种图形。此外，还有一个外购的GUI软件包TextView，用于显示，但它没有Shape功能。

适配器

如何让TextView的接口转换成为Shape的接口，有两种方法：

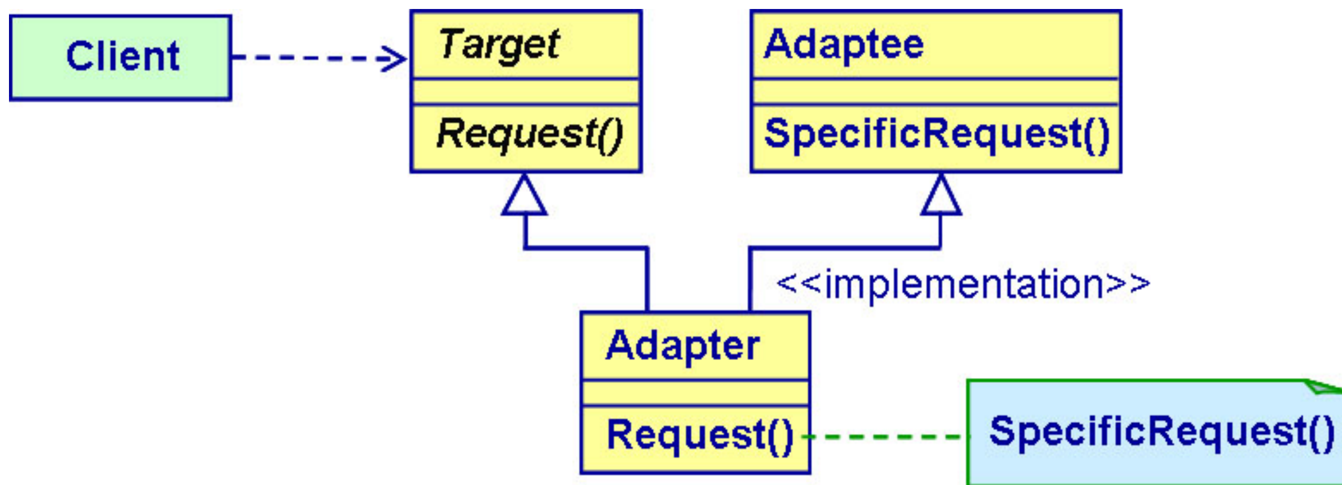
- 让TextShape同时继承Shape的接口和TextView的服务（多重继承）；
- 在TextShape中建立TextView的实例，再通过TextView给出TextShape的接口。

前者是适配器的类模式，后者是对象模式。下图就是适配器的对象模式。



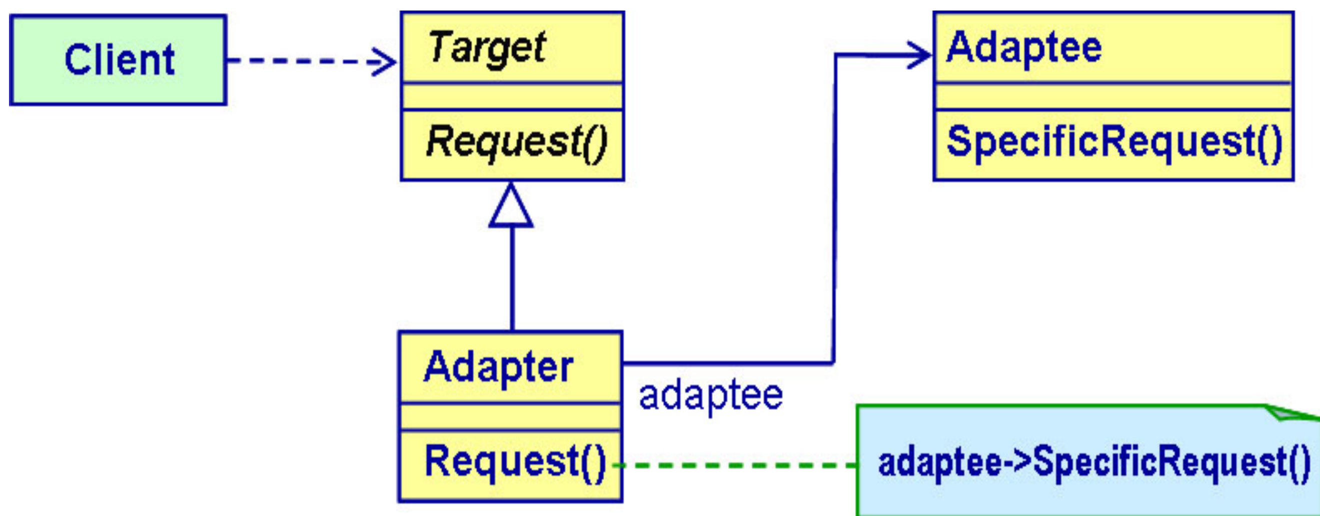
适配器

(3) 结构：适配器模式有类适配器模式和对象适配器模式。类适配器可以通过多继承方式实现不同接口之间的相容和转换，如图所示。



适配器

- **而一个对象适配器则依赖对象组合的技术实现接口的相容和转换，如图所示。**



适配器

(4) 参与者职责

a) 目标（Target）：定义客户使用的与应用领域相关的接口。

b) 客户（Client）：与具有Target接口的对象合作。

c) 被匹配者（Adaptee）：需要被转换匹配的一个已存在接口。

d) 适配器（Adapter）：将Adaptee的接口与Target接口匹配。

适配器

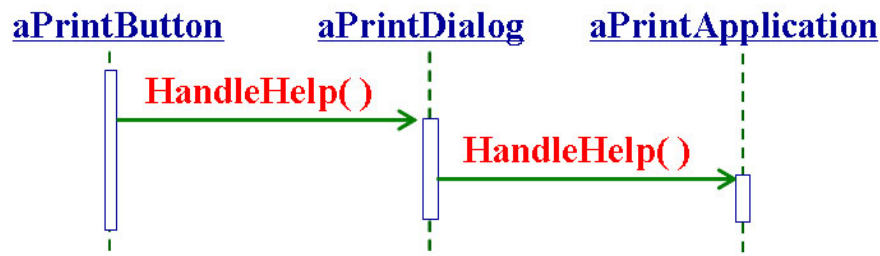
(5) 协作：客户调用Adapter对象的操作，然后Adapter的操作又调用Adaptee对象中负责处理相应请求的操作。

责任链

(1) 目的：通过一条隐式的对象消息链传递处理请求。该请求沿着这条链传递，直到有一个对象处理它为止。其核心是避免将请求的发送者直接耦合到它的接受者。

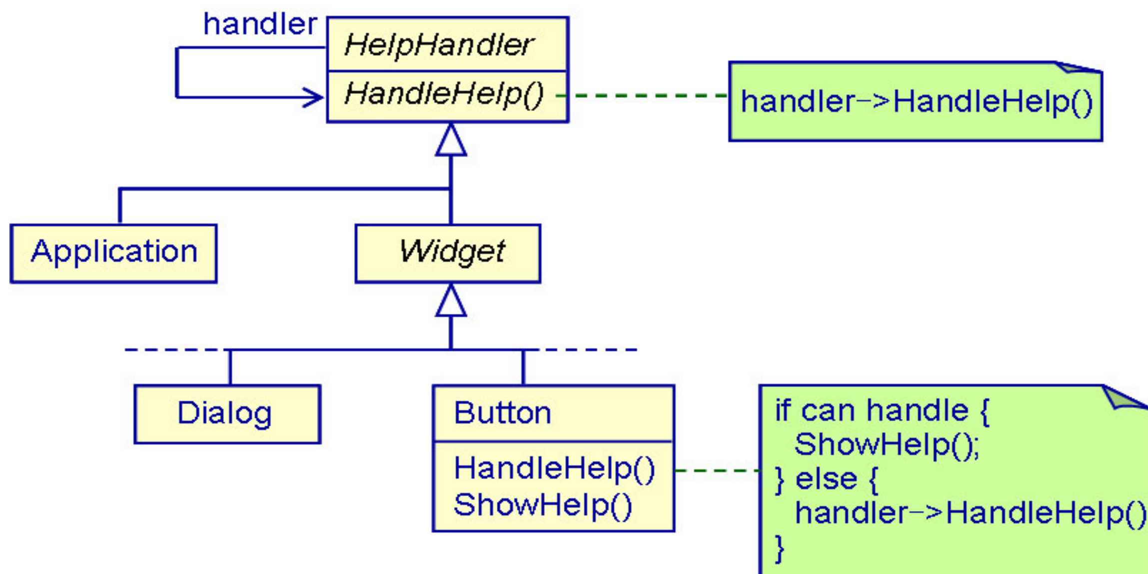
(2) 思路：以GUI系统的联机帮助系统为例。用户可以在软件中任一位置按下help键，软件就可以根据该信息和当前上下文环境弹出适当的说明。

如果用户在PrintDialog对话框里“打印”按钮上按了帮助键，帮助信息的顺序图如图所示。



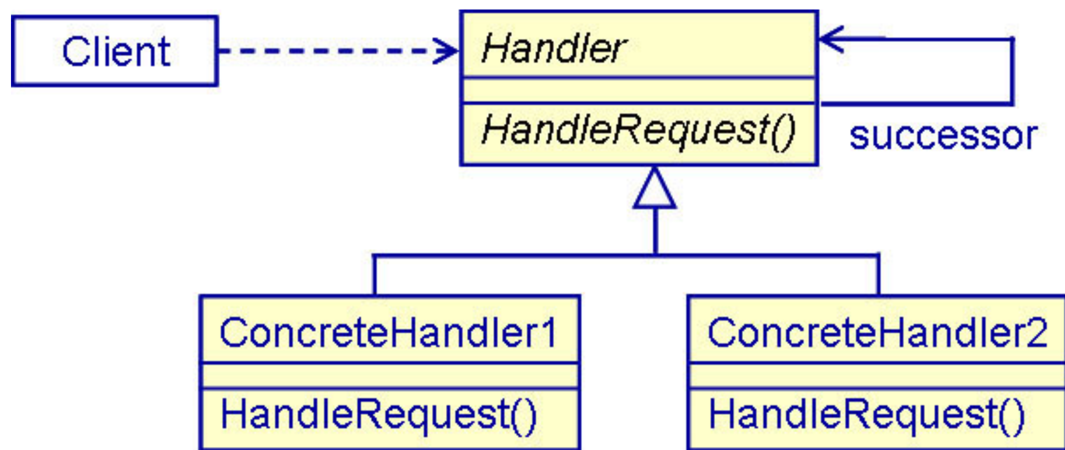
责任链

- **联机帮助系统定义了一个抽象类HelpHandler和抽象操作HandleHelp()，所有想处理信息的类可以继承该类。HelpHandler的HandleHelp() 操作的内定做法是把信息传递给后继者去处理，由各个子类分别来实现具体的打印功能。如图所示。**



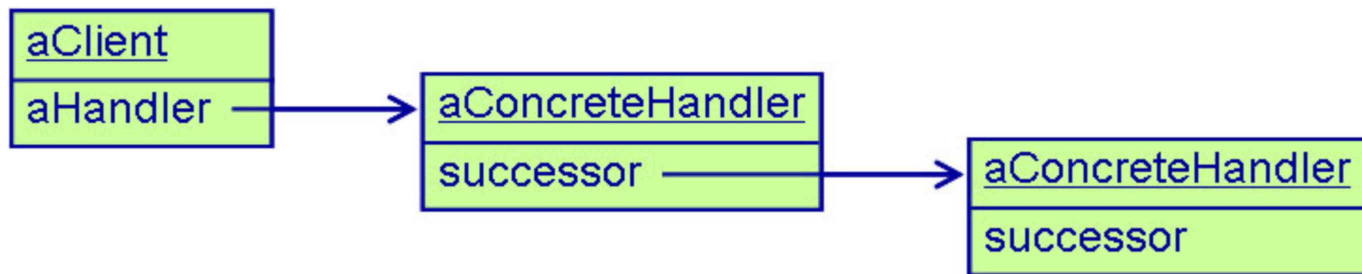
责任链

(3) 结构：责任链模式的结构如图所示。



责任链

- 典型的对象间的结构如图所示。



责任链

(4) 参与者职责

- a) **处理者（Handler）：定义处理请求的接口；实现对后继者的链接（可选）。**
- b) **具体处理者（ConcreteHandler）：处理它所负责的请求；可访问它的后继；如果它能够处理请求，就处理该请求，否则将请求传送给后继者。**
- c) **客户（Client）：将处理请求提交给职责链中的ConcreteHandler对象。**

(5) 协作：当**Client**发出请求之后，请求会在责任链中传递，直到有一个**ConcreteHandler**对象能处理为止。

中介者

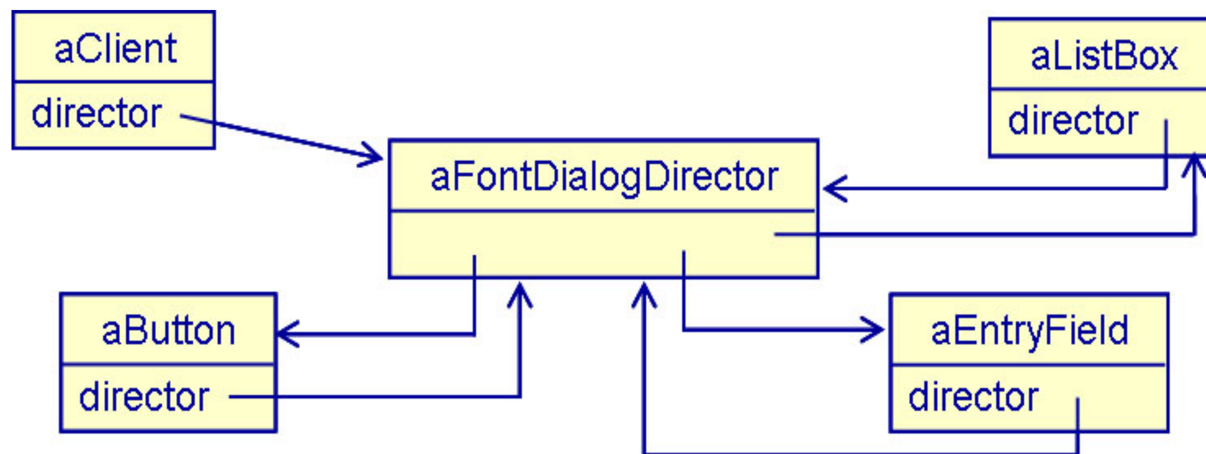
(1) 目的：用一个中介对象来封装一系列复杂对象的交互情景。中介者通过阻止各个对象显式地相互引用来降低它们之间的耦合，使得人们可以独立地改变它们之间的交互。

(2) 思路：以GUI系统的对话框为例，对话框中会布置许多窗口组件，如按钮、菜单、文字输入栏等。对话框中各窗口组件之间往往相互牵连。

为此，可以将这些窗口组件的集体行为封装成一个中介者（mediator）对象。中介者负责居中指挥协调一组对象之间的交互行为，避免互相直接引用。这些对象只认得中介者，因而可降低交互行为的数目。

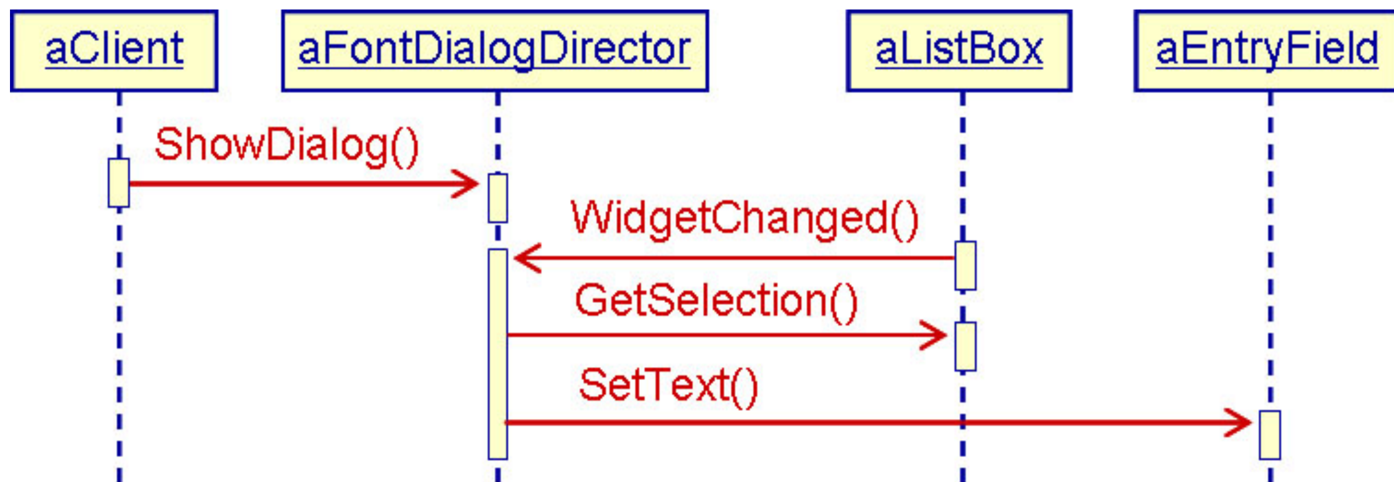
中介者

- **例如，可用FontDialogDirector当作对话框内各窗口组件之间的中介者。FontDialogDirector对象认得所有组件，协调彼此之间的交互，如同一个通信枢纽，如图所示。**



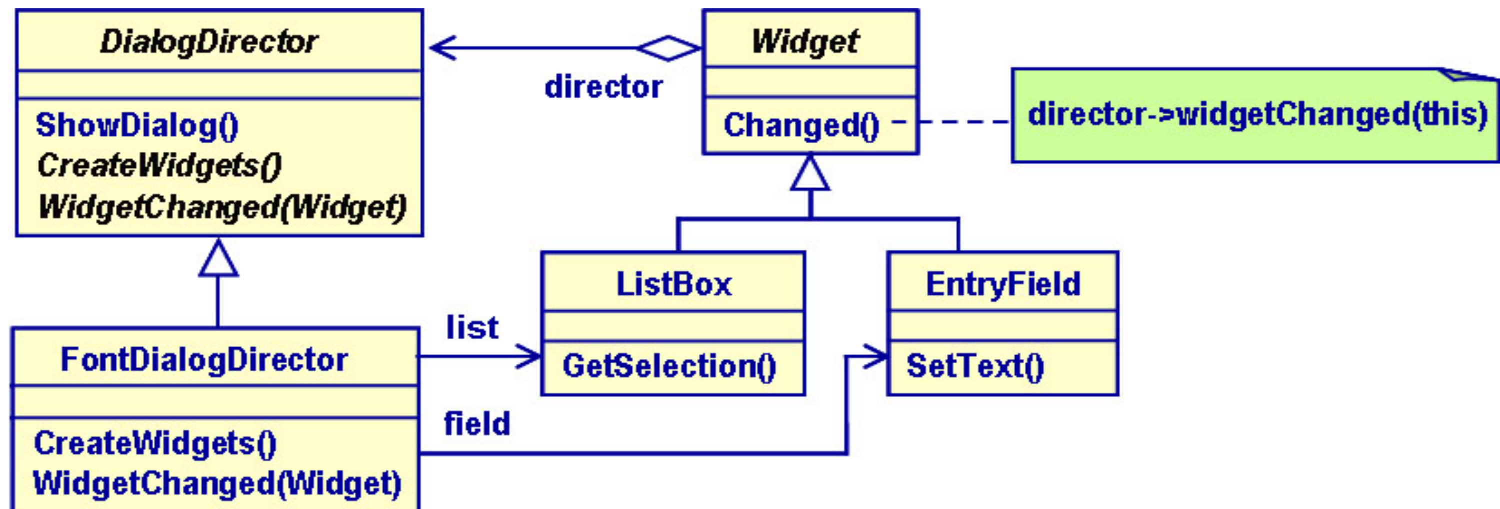
中介者

- 描述**mediator**作用的顺序图 。



中介者

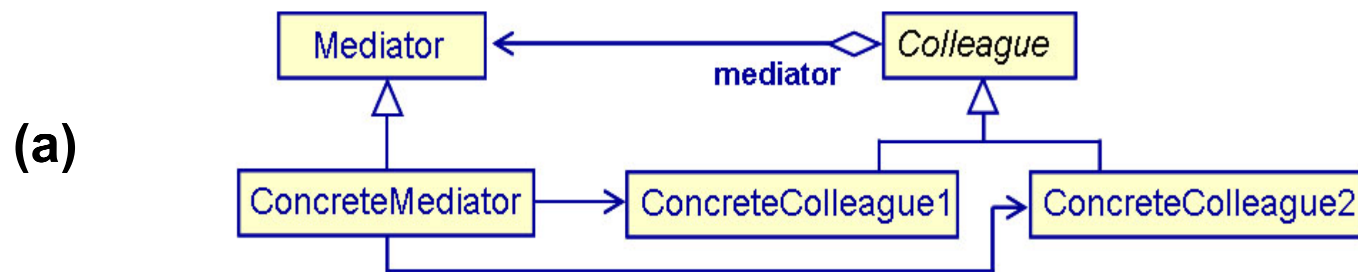
- **下图显示了加入FontDialogDirector后的类结构。**



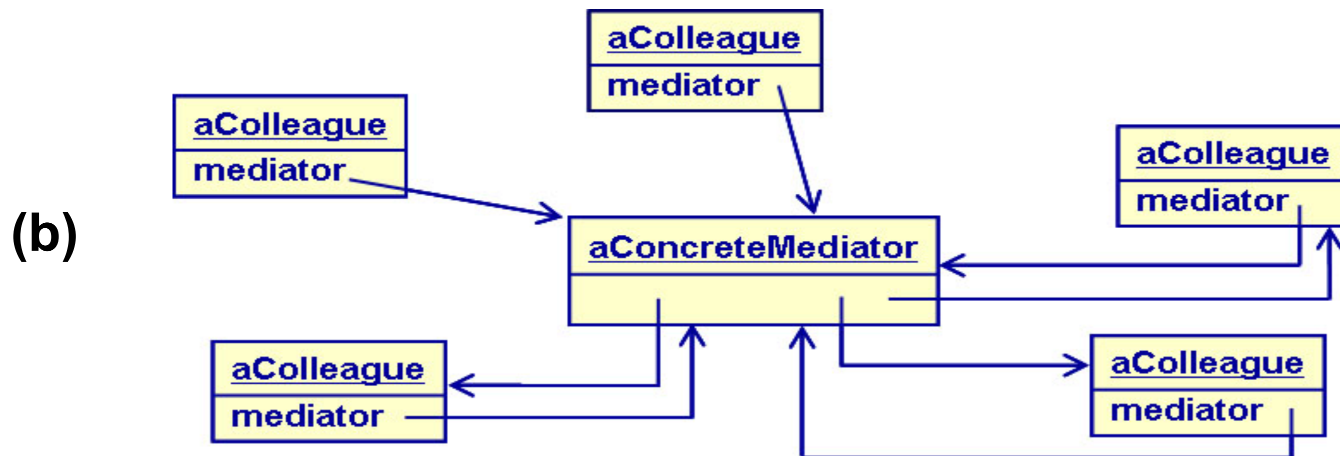
中介者

(3) 结构：图(a)给出了中介者的类结构，图(b)给出了典型的对象结构。

(a)



(b)



中介者

(4) 参与者职责

a) **中介者（Mediator）：定义与各个同事（Colleague）对象通信的接口。**

b) **具体中介者（ConcreteMediator）：协调各个同事对象，实现协作行为；了解并维护各个同事对象。**

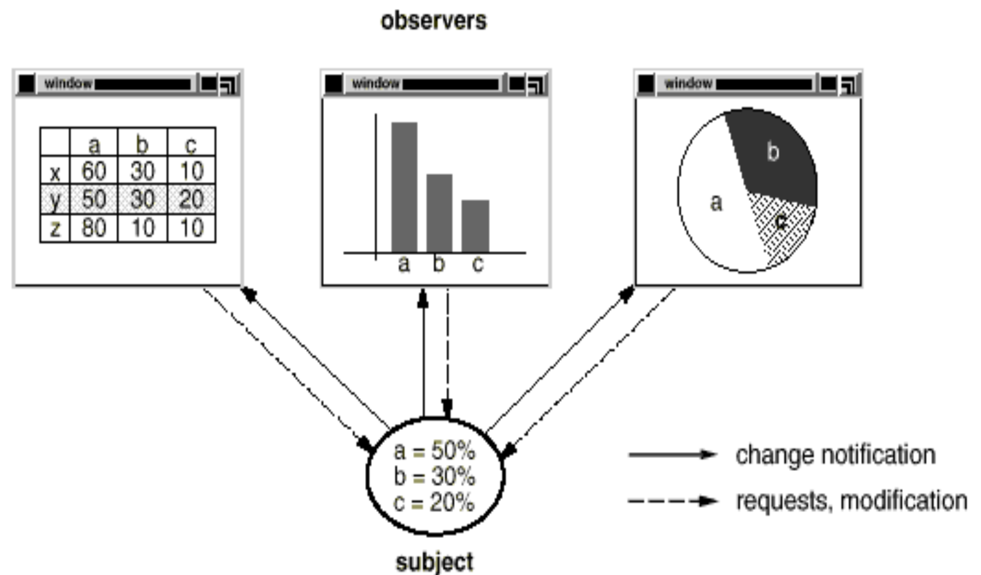
c) **同事类（Colleague classes）：这些同事类的对象都了解中介者；一个同事对象与另一个同事对象之间的通信都需要通过中介者来间接实现。**

(5) 协作：同事向中介者对象发送或接收请求，中介者则将请求传送给适当的同事对象（一个或多个），协调整体行为。

观察者

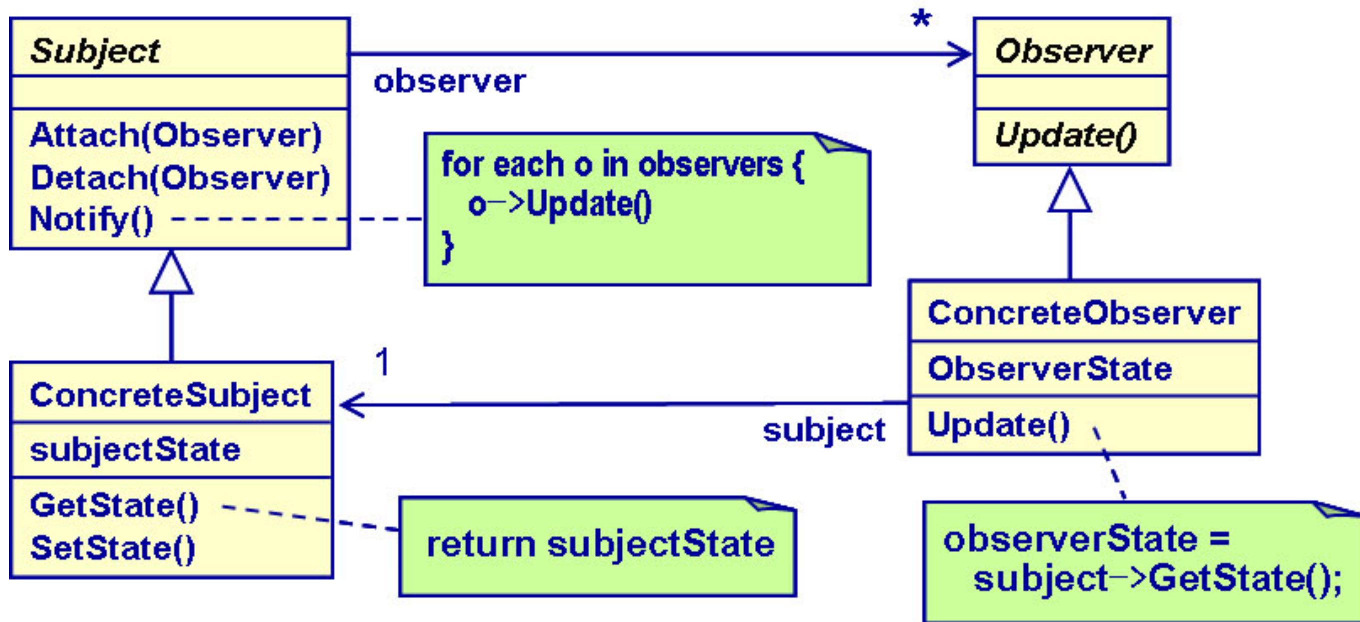
(1)目的：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知，并被自动更新。

(2)思路：例如，许多GUI软件包都将数据显示部分与应用程序底层的数据表示分开，以利于分别复用。但这些类也能合作，如图所示的计算表和直方图都是针对同一数据对象的两种不同表示方式。



观察者

(3) 结构：Observer模式的结构如图所示。



观察者

(4) 参与者职责

a) 主题（Subject）：认得它的观察者。任意数目的观察者对象均可订阅一个主题。另外，提供一个连接观察者对象和解除连接的接口。

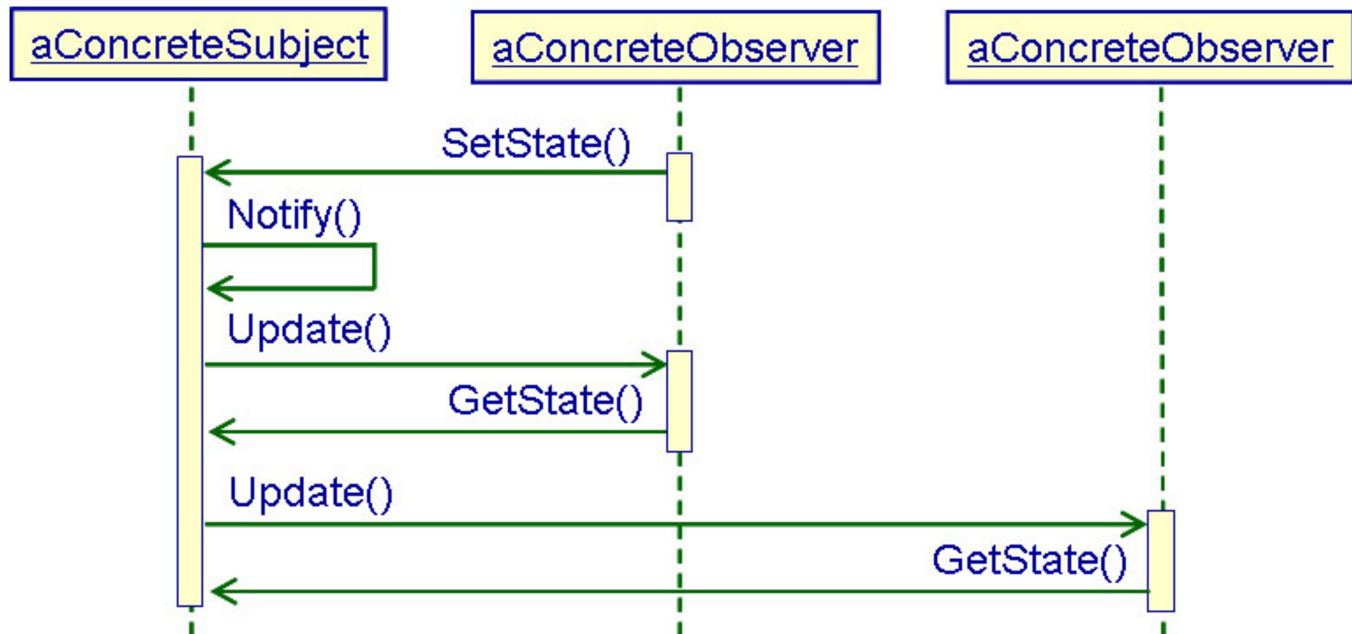
b) 观察者（Observer）：定义了一个自我更新的接口。一旦发现主题有变时借助接口通知自己随之改变。

c) 具体主题（ConcreteSubject）：存储具体观察者对象关心的状态；当状态改变时向它的观察者发送通知。

d) 具体观察者（ConcreteObserver）：维持一个对具体主题对象的引用；存储要与主题一致的状态；实现观察者的自我更新接口，确保自己的状态与主题的状态一致。

观察者

(5) 协作：当具体主题发生会导致观察者的状态不一致的情况时，就会主动通知所有该通知的观察者。当具体观察者收到通知后，向主题询问，根据所得信息使自己的状态与主题的状态保持一致。下图给出了一个主题和两个观察者对象之间的交互情况。



That's All!