软件体系结构
SEI/BUAA
北航计算机学院
School of Computer Science & Engineering

第二部分
创建软件体系结构

张莉 教授
北京航空航天大学
软件工程研究所
lily@buaa.edu.cn

软件体系结构
SEI/BUAA

第二部分　创建软件体系结构

- 理解和实现质量属性
  - 软件系统的质量属性
  - 软件体系结构与质量属性
  - 软件体系结构设计的原子操作
  - 软件体系结构战术
  - 软件体系结构模式和风格
- 软件体系结构案例分析
  - KWIC System
  - WWW
  - CORBA

什么是软件体系结构风格

- 软件体系结构风格(Style)：是反复出现的组织模式和习惯用法，是对一系列体系结构设计的抽象。
- 本质：
  - 是一些特定的元素按照特定的方式组成一个有利于上下文环境里的特定问题的解决的结构。
- 优势：
  - 促进设计重用。
  - 能带来大量的代码重用
  - 采用例行结构，易于理解
  - 使用标准的风格，有利于系统的互操作性
  - 采用某种体系结构风格，可以重用相应的体系结构分析等技术
  - 一般可以提供特定风格的可视化

什么是软件体系结构风格

- **Patterns occur in all phases of design**
  - system pattern ---architecture style
  - design pattern ---首先简单介绍
  - code pattern

	编程模式	设计模式	软件体系结构风格
重点	开发原则、可实现性	重用、概念完整性	重用、概念完整性
抽象层次	代码	构件	体系结构
使用	直接可用	直接可用	又约束和指导作用的概念

A small catalog of architecture styles

体系结构风格	进一步细分	
数据流系统 (Data flow)	顺序批处理 (Batch Sequential)	
	管道和过滤器 (Pipes and Filters)	
调用-返回系统 (call and return)	主程序和子程序(Main program and subroutine)	
	OO 系统	
	分层结构 (Layered)	
独立组件 (Independent components)	通信过程 (Communicating Process)	
	事件系统	Implicit Invocation
		Explicit Invocation
虚拟机 (Virtual machine)	解释器 (Interpreter)	
	基于规则的系统 (Rule-Based System)	
数据为中心的系统 (Data-centered)	数据仓库 (Repository)	
	超文本系统 (Hypertext)	
	黑板 (Blackboard)	

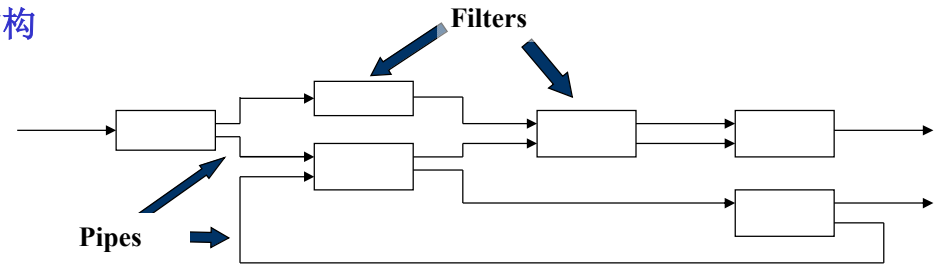
数据流风格

- 目标: 取得可重用性和可修改性
- 特点：将系统看作是对输入数据的一系列连续处理 。
- **Data Flow**
  - Batch Sequential
  - Pipes and Filters

	完整性	组合性	抽象性	重用性	构建性	异构性	可分析性
数据流图		+	+	+	+		

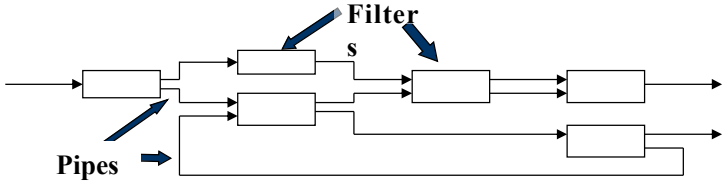
管道和过滤器（Pipes and filters）风格

- 基本结构



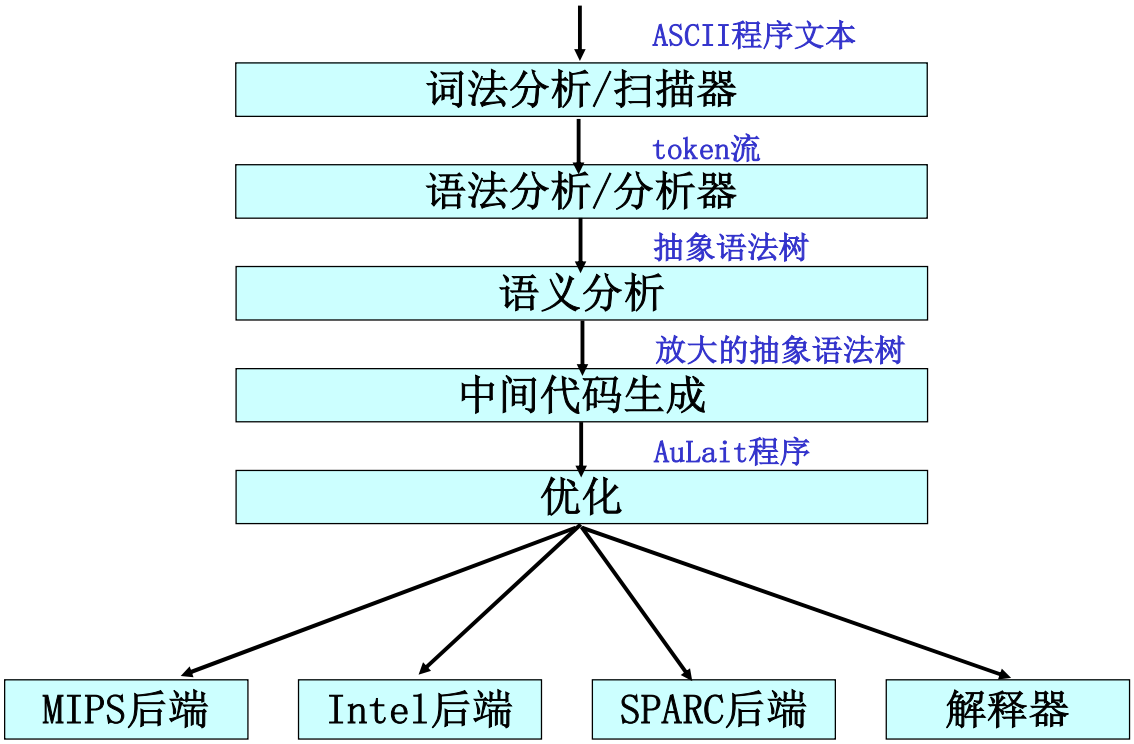
- 最早出现的Unix系统中。
- 适用于：对有序数据进行一系列已经定义的独立计算的应用程序
- 特征
  - 构件：过滤器是独立实体，相互之间不共享状态；过滤器不了解其它过滤器的信息。独立处理输入，结果从输出端流出；可以并行地使用过滤器。
  - 连接件：管道——信息流的导管作用。
  - 每个构件都有输入/输出集合，构件在输入处读取数据流，并在输出处生成数据。

管道和过滤器（续）

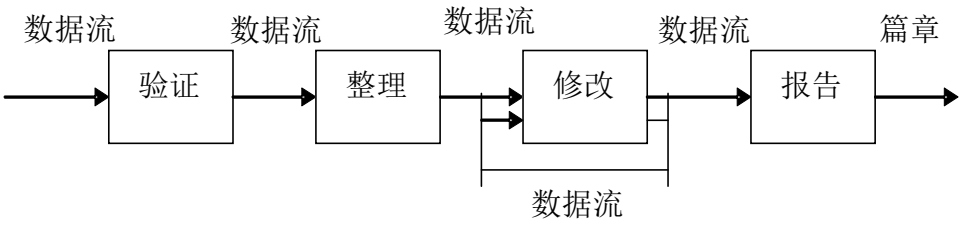


- 优点：
  - （1）每个构件的行为不受其它构件的影响，整个系统易于理解（2）系统分解；（3）软件重用；（4）系统维护和功能增强，（5）支持并行处理，（6）支持特殊分析，如吞吐量分析、死锁分析
- 缺点：
  - 不适合交互式的应用；
  - 在处理两个独立但是相关流之间的同步时可能会遇到困难
  - 系统性能低；
  - 每个过滤器的解析输入和格式化输出任务繁琐，带来系统的复杂性上升。
- 例子：
  - Unix shell:如 cat file | grep xyz | sort | uniq > out
  - 传统编译器：词法分析→语法分析→语义分析→代码生成

例子



批处理顺序系统Batch Sequential



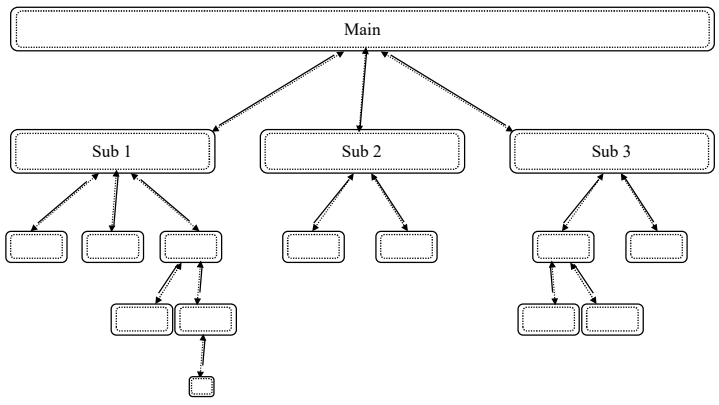
- 管道和过滤器风格的一个特例，此时过滤器作为一个独立的实体处理它的所有输入。
- 在批量顺序体系中，执行步骤或是各个组成部分都是独立的程序，并且设定每一步完成之后下一步才能开始。在各步骤之间每批数据都被作为一个整体转换。

调用返回风格

- **Call and return**
  - 主程序与子程序
  - 面向对象
  - 分层风格

	完整性	组合性	抽象性	重用性	构建性	异构性	可分析性
调用与返回	+	+	+				+

主程序与子程序风格



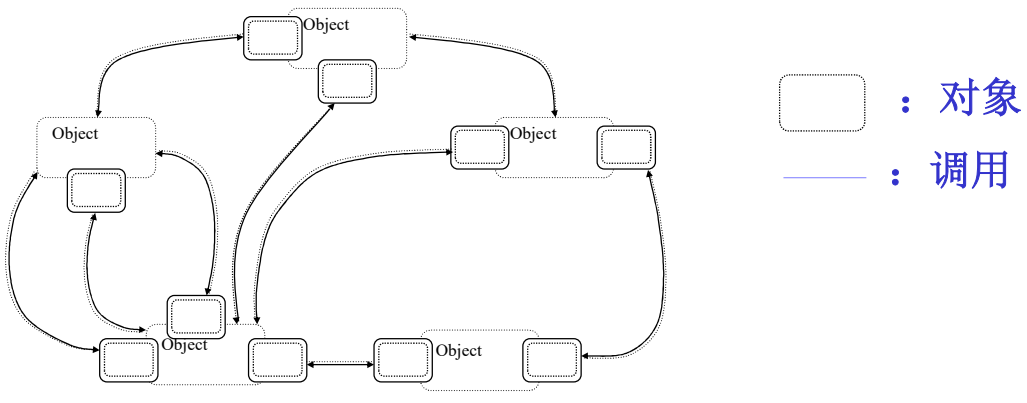
- 分层分解
- 结构化控制流
- 结构化连接方式

主程序与子程序风格（续）

- 优点
  - 是程序设计语言中最基本的表现形式。几乎所有的程序设计语言都对此提供了支持。并以函数、过程、库、包等形式出现。
  - 程序中需要多次执行的代码，可以使用函数或子过程的形式，提高代码的使用率，减少内存的使用。
  - 合理地组织子程序和恰当的过程名为软件代码的理解和维护提供方便。
- 缺点
  - 难以表达构件间复杂的连接关系。只有过程间的调用关系；
  - 软件规模增大时，造成代码可维护性和易理解性变差；
  - 可重用层次低。主要是科学计算和数据结构经典算法的重用。

面向对象和抽象数据风格

- 数据及其操作被封装成抽象数据类型(对象)。对象即是**构件**。
- **连接件**通过过程调用（方法）来实现。
- 适用于：
  - ✓ 以相互关联的数据实体的标识和保护为中心问题的应用程序



FLGURE 5.6 The object-oriented style

面向对象和抽象数据风格

- 面向对象的特点：
  - 封装：对象具有信息隐藏特性。对象负责维护本身的内部变量，其内部结构对其它对象不可见。
  - 继承：指从具有通用特征的对象开始，逐渐定义更具体的对象。新对象的创建可以在继承已有对象定义的基础上，加入其他定义。
  - 多态：指不同类型对象可以对相同的激励适当做出不同的响应。
- 数据抽象风格是特殊的OO风格，只有封装的特点，没有继承和多态的特点。

面向对象和抽象数据风格

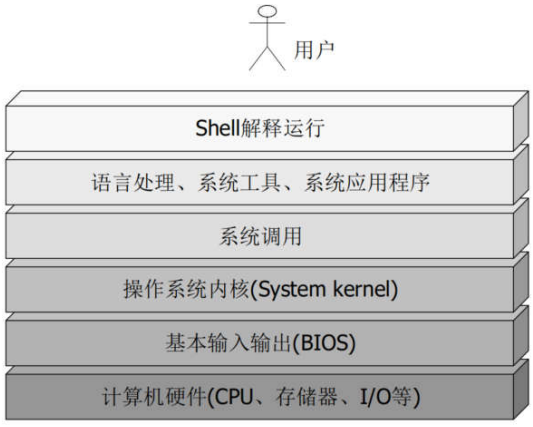
- 面向对象风格的优点
  - 封装：（1）封装提供了可重用性和易修改性；可以在不影响其客户的情况下修改对象；（2）提供功能独立性。
  - 继承：支持重用
  - 多态：系统的灵活性和动态性
  - 对象可以是单线程，也可以是多线程
  - 支持问题分解。
- 缺点
  - 过程调用依赖于对象标识的确定。和管道过滤器风格的系统不同，一个对象必须知道它要与之交互的对象的标识。
  - 不同对象的操作关联性弱。在两个对象同时访问一个对象时，可能引起副作用。
  - 对象的接口一旦改变，所有使用它的其它对象都需要修改。
- 例子：CORBA

A small catalog of architecture styles

体系结构风格	进一步细分	
数据流系统 (Data flow)	顺序批处理 (Batch Sequential)	
	管道和过滤器 (Pipes and Filters)	
调用-返回系统 (call and return)	主程序和子程序(Main program and subroutine)	
	OO 系统	
	分层结构 (Layered)	
独立组件 (Independent components)	通信过程 (Communicating Process)	
	事件系统	Implicit Invocation
		Explicit Invocation
虚拟机 (Virtual machine)	解释器 (Interpreter)	
	基于规则的系统 (Rule-Based System)	
数据为中心的系统 (Data-centered)	数据仓库 (Repository)	
	超文本系统 (Hypertext)	
	黑板 (Blackboard)	

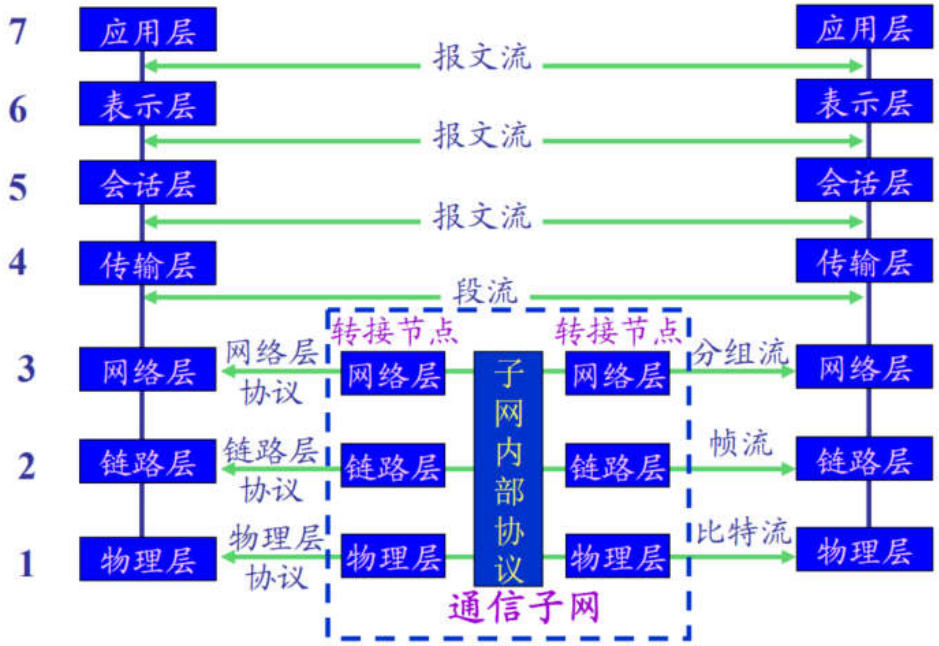
分层系统风格（Layered）

- 层次化早已经成为一种复杂系统设计的普遍性原则；
- 两个方面的原因：
  - 事物天生就是从简单的、基础的层次开始发生的；
  - 众多复杂软件设计的实践，大到操作系统，中到网络系统，小到一般应用，几乎都是以层次化结构建立起来的。

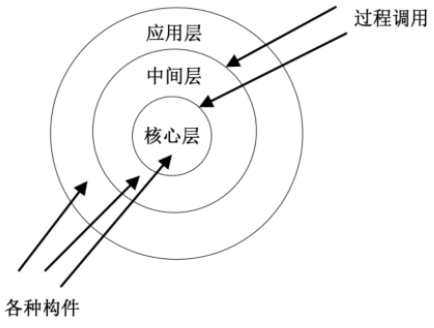


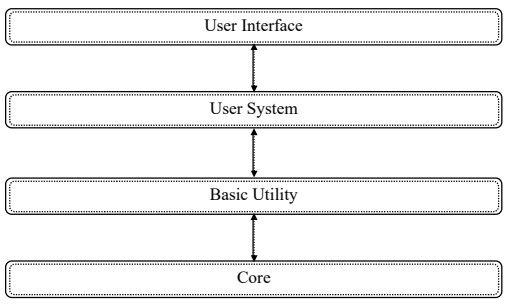
操作系统的层次结构

网络的分层结构



分层系统风格（Layered）





FLGURE 5.7 The layered style

- 适合于：按层次结构来组织不同类别的相关服务的应用程序
- 特点
  - 系统按照层次结构组织，每一层向它的上层提供服务，同时又是它的下层的客户。
  - 在某些系统中，除了邻接的层，一些内部层次对于其他外部层次是隐藏的。——这在体系结构约束中定义

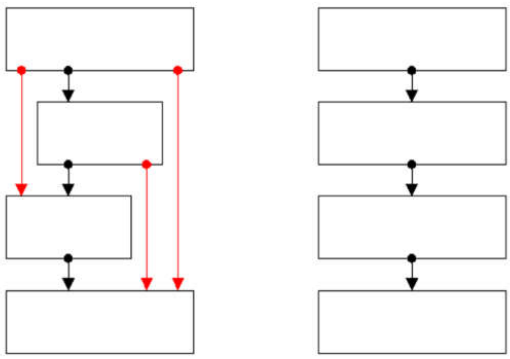
某些特殊的分层系统允许非相邻层次间直接通信，
往往是出于效率方面的考虑

分层系统（续）

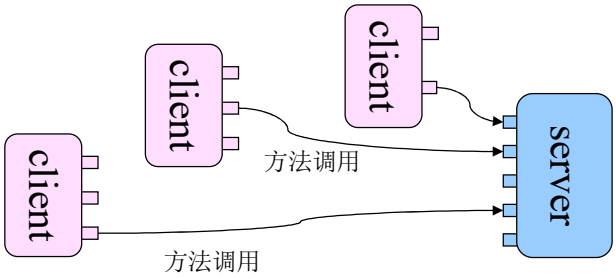
- 优点：
  - 设计体现抽象层次，支持分解复杂问题，支持增量式设计；
  - 由于对邻接关系进行了限制，所以系统易于改进和扩展；
  - 每一层的软件都支持重用，并可为每一层提供多种可替换的实现。
- 缺点：
  - 分层可能带来效率方面的问题；
  - 接口定义标准的问题
  - 如何界定层次间的划分是较复杂的问题
- 典型实例：
  - 通讯协议。如：OSI的七层参考模型
  - 操作系统，如Unix系统。
  - 数据库系统等

严格分层和松散分层

- 严格分层系统要求严格遵循分层原则，限制一层中的构件只能与对等实体以及与它紧邻的下面一层进行交互
  - 优点：修改时的简单性
  - 缺点：效率低下
- 松散的分层应用程序放宽了此限制，它允许构件与位于它下面的任意层中的组件进行交互
  - 优点：效率高
  - 缺点：修改时困难



客户服务器风格（Client-Server）



- 适于应用系统的数据和处理分布在一定范围内的多个构件上，构件之间通过网络连接。
- 服务器构件：向多个用户提供服务，永远处于活跃状态，监听用户请求；
- 客户构件：向服务器构件请求服务；
- 连接件：某种进程间通讯机制，通常是基于RPC的交互协议。

客户服务器风格（续）

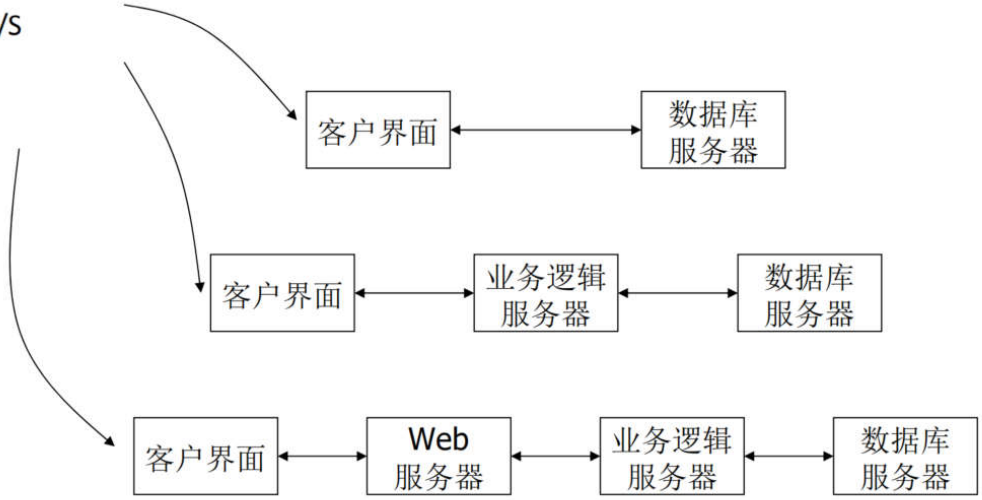
- 优点
  - 有利于分布式的数据组织
  - 构件间是位置透明的，构件间无需知道位置信息
  - 便于异质平台间的融合与匹配，构件可以运行不同的操作系统；
  - 具有良好的可扩展性，易于对服务器进行修改、扩展和增加服务器。
- 缺点
  - 客户必须知道服务器的访问标识，否则很难知道有哪些服务可用。
- 例子
  - 远程文件系统、数据库服务器、

客户服务器风格（续）

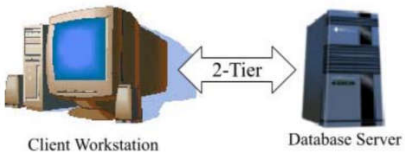
- 客户机/服务器：一个应用系统被分为两个逻辑上分离的部分，每一部分充当不同的角色、完成不同的功能，多台计算机共同完成统一的任务。
  - 客户机(前端，front-end)：业务逻辑、与服务器通讯的接口；
  - 服务器(后端：back-end)：与客户机通讯的接口、业务逻辑、数据管理。
- 一般的，
  - 客户机为完成特定的工作向服务器发出请求；
  - 服务器处理客户机的请求并返回结果。

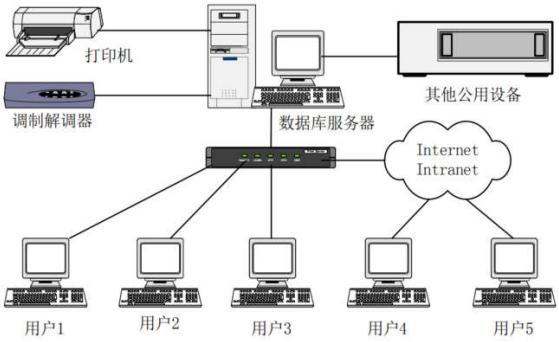
客户服务器的层次

- “客户机-服务器”结构的发展历程：
  - 两层C/S
  - 三层C/S
  - 多层C/S



两层C/S结构





◇ 任务分配

➢服务器

（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。

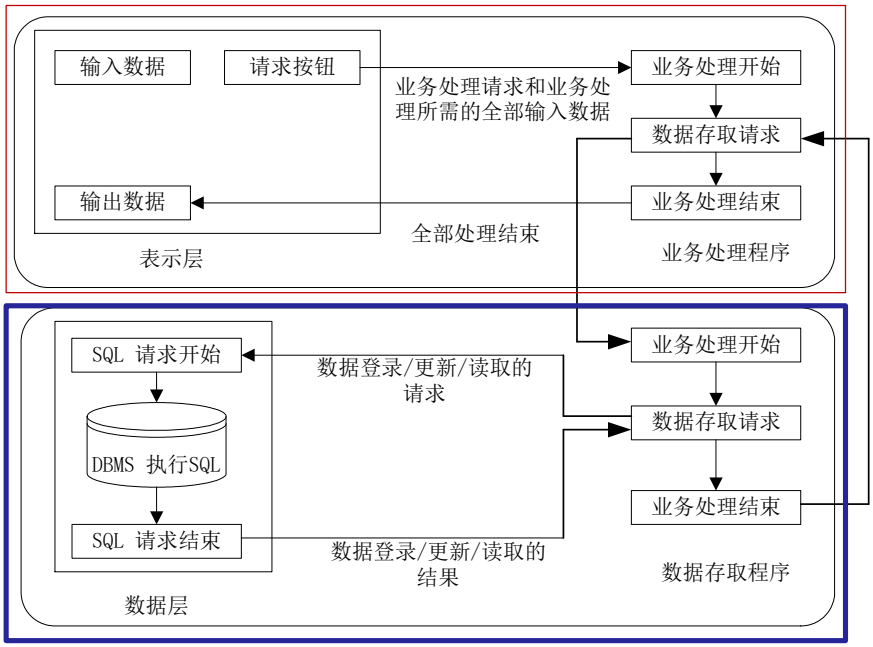
➢客户应用程序

（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。

◇ 处理流程



◇ 优点

- ✓ 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。
- ✓ 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。
- ✓ 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。

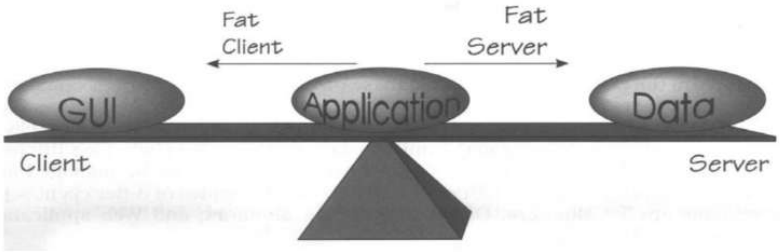
C/S 体系结构

◇ 缺点

- 开发成本较高
- 客户端程序设计复杂
- 信息内容和形式单一
- 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
- 新技术不能轻易应用

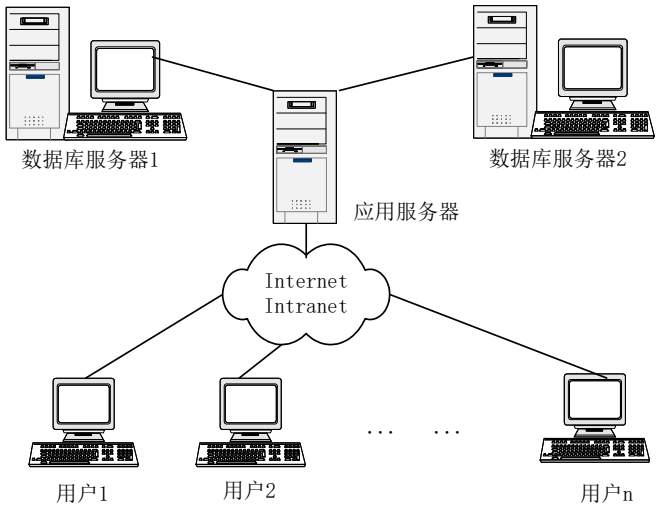
胖客户端和瘦客户端

- 业务逻辑的划分比重：在客户端多一些还是在服务器段多一些？
  - 胖客户端：客户端执行大部分的数据处理操作
  - 瘦客户端：客户端具有很少或没有业务逻辑



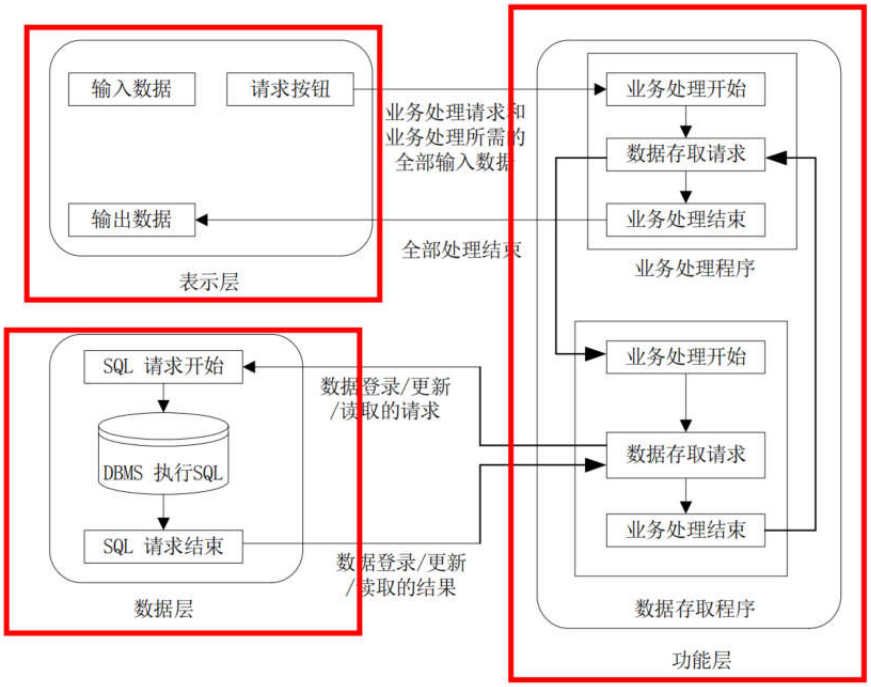
三层客户/服务器风格

- 在客户端与数据库服务器之间增加了一个中间层
  - 第一层：用户界面—表示层
  - 第二层：业务逻辑—功能层
  - 第三层：数据库—数据层



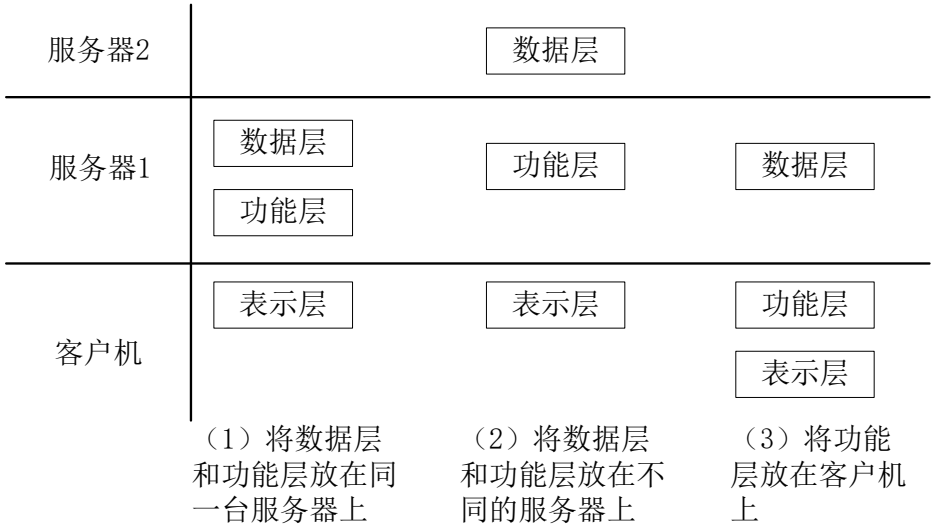
三层客户/服务器风格

◇ 处理流程



三层客户/服务器风格

◇ 物理结构



创建软件体系结构

三层客户/服务器风格

◇ 优点

- 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，从而提高系统和软件的可维护性和可扩展性。
- 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。
- 应用的各层可以并行开发，可以选择各自最适合的开发语言。
- 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，为严格的安全管理奠定了坚实的基础。

北京航空航天大学软件工程研究所

三层客户/服务器风格

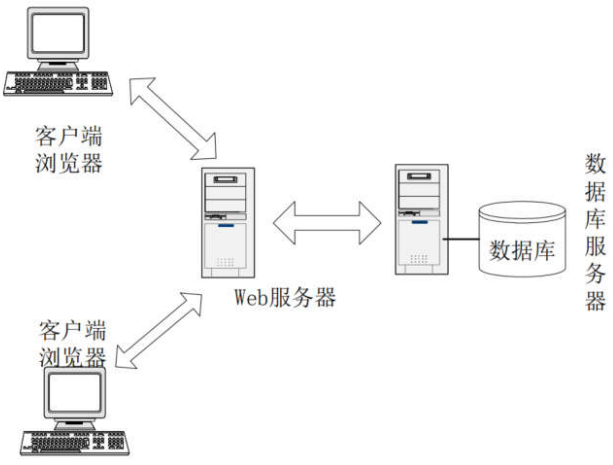
◇ 要注意的问题

➢三层C/S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。

➢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结构的关键问题。

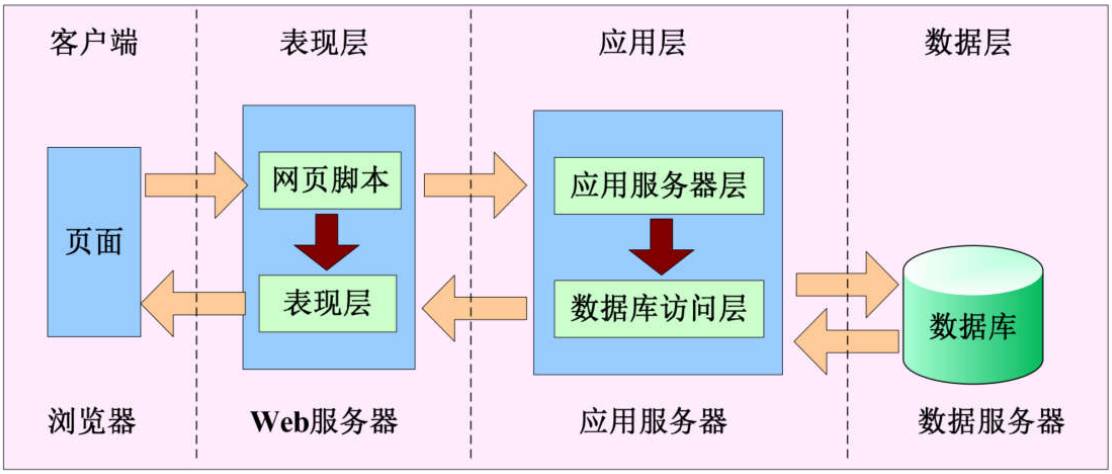
浏览器/服务器风格（B/S结构）

- 浏览器/服务器(B/S)是三层C/S风格的一种实现方式。
  - 表现层：浏览器
  - 逻辑层：
    - Web服务器
    - 应用服务器
  - 数据层：数据库服务器



浏览器/服务器风格

◇ B/S体系结构



创建软件体系结构

浏览器/服务器风格

◇ 优点

➢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。系统维护成本低。

- 客户端无任何业务逻辑，用户在使用系统时，仅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。
- 良好的灵活性和可扩展性：对于环境和应用条件经常变动的情况，只要对业务逻辑层实施相应的改变，就能够达到目的。

- B/S成为真正意义上的“瘦客户端”，从而具备了很高的稳定性、延展性和执行效率。
- B/S将服务集中在一起管理，统一服务于客户端，从而具备了良好的容错能力和负载平衡能力。

➢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。

浏览器/服务器风格

◇ 缺点

➢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。

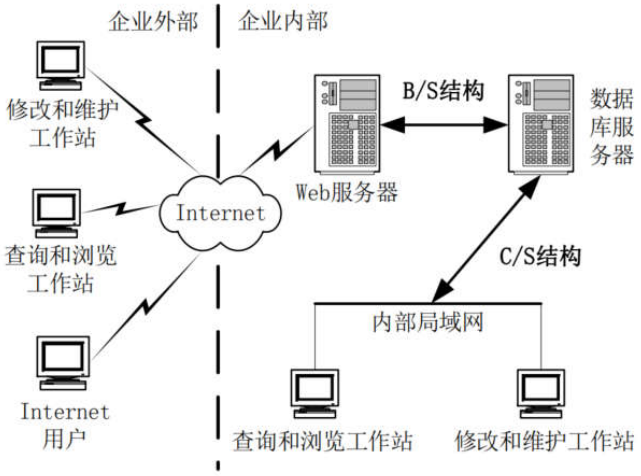
➢B/S体系结构的系统扩展能力差，安全性难以控制。

➢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。

➢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。

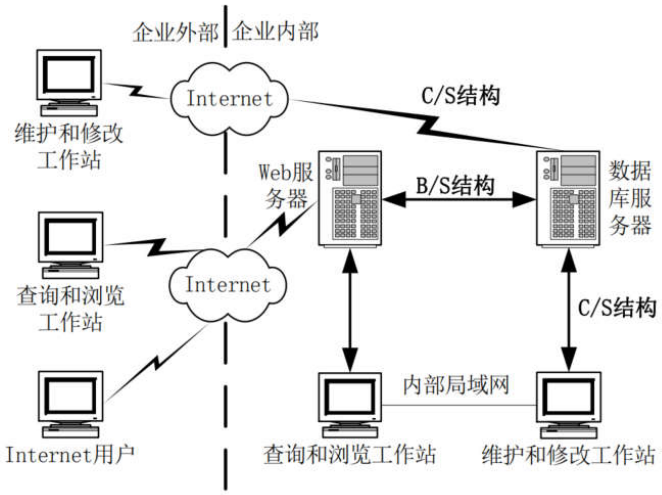
C/S+B/C内外有别模式

- 为了克服C/S与B/S各自的缺点，发挥各自的优点，在实际应用中，通常将二者结合起来；
- 遵循“内外有别”的原则：
  - 企业内部用户通过局域网直接访问数据库服务器
    - C/S结构；
    - 交互性增强；
    - 数据查询与修改的响应速度高；
  - 企业外部用户通过Internet访问Web服务器/应用服务器
    - B/S结构；
    - 用户不直接访问数据，数据安全；



C/S+B/C查改有别模式

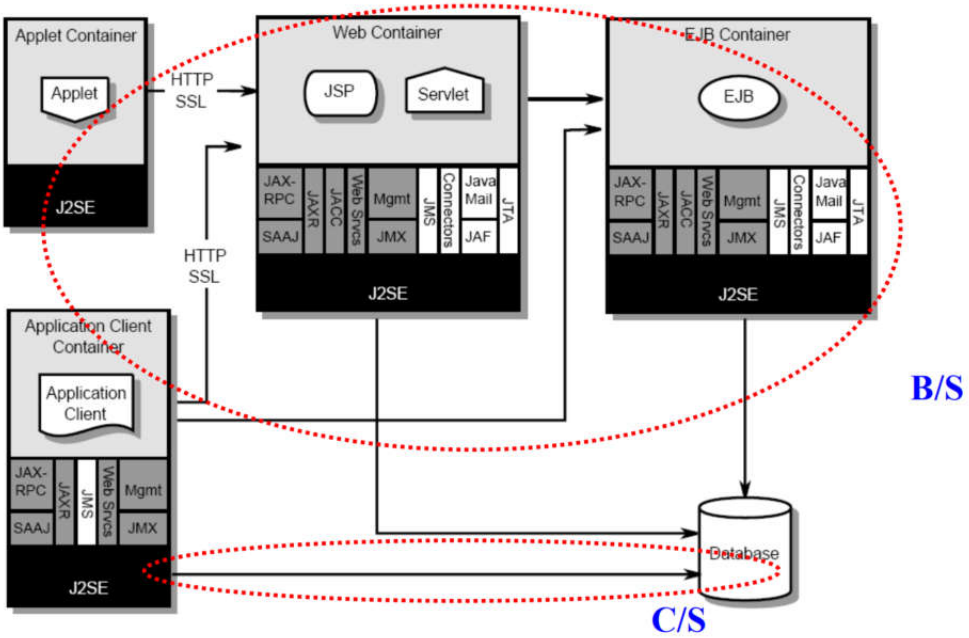
- 遵循“查改有别”的原则：
  - 不管用户处于企业内外什么位置(局域网或Internet)，凡是需要对数据进行更新操作的(Add, Delete, Update)，都需要使用C/S结构；
  - 如果只是执行一般的查询与浏览操作(Read/Query)，则使用B/S结构。



软件体系结构 SEI/BUAA
创建软件体系结构

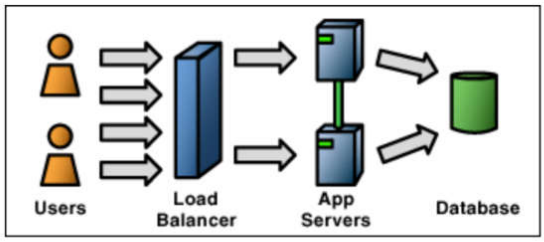
J2EE B/S+C/S Architecture

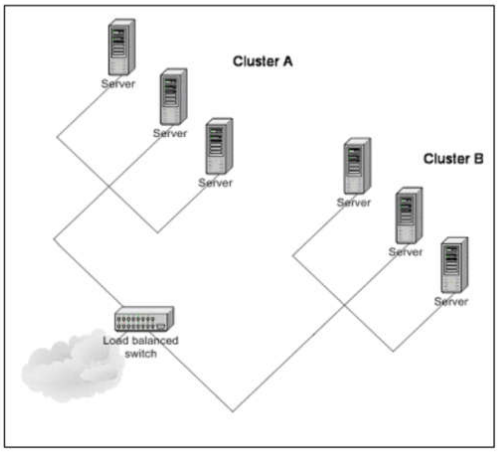
C/S与B/S的混合相当于分层风格里的“松散分层模式”



基于集群(Cluster)的C/S和B/S物理分布

- 事实上，功能层并不一定只驻留在同一台服务器上，数据层也是如此；
- 如果功能层(或数据层)分布在多台服务器上，那么就形成了基于集群(Cluster)的C/S物理分布模式。



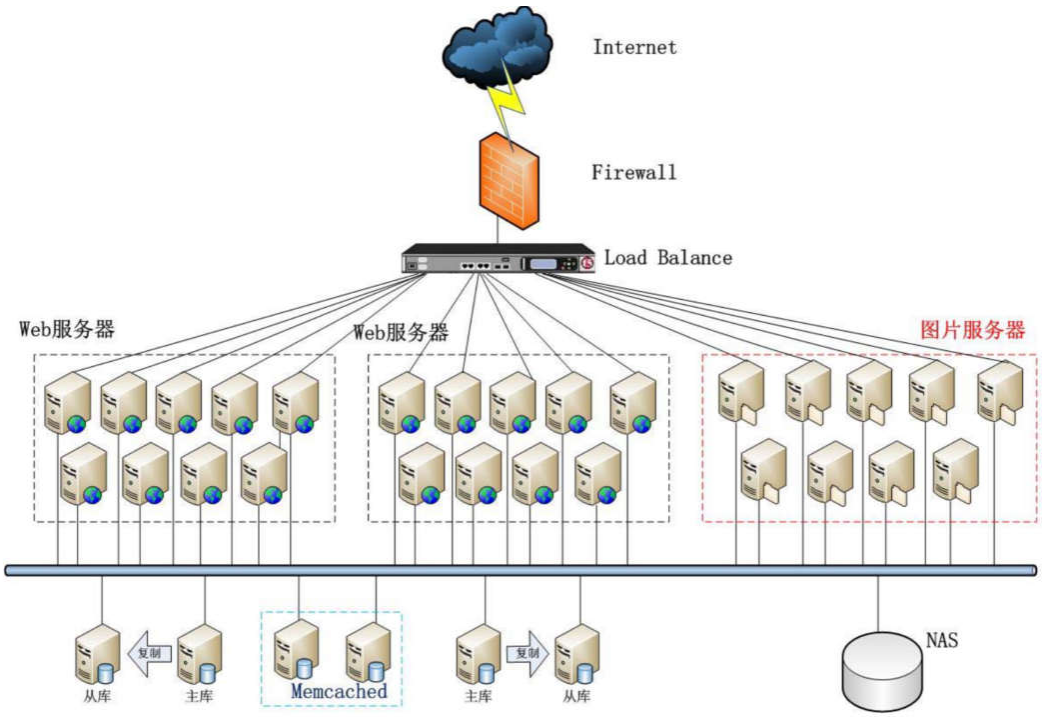


软件体系结构 SEI/BUAA
创建软件体系结构

集群(Cluster)

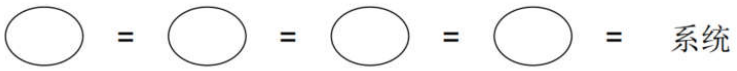
- A cluster is a group of loosely coupled servers that work together closely so that in many respects it can be viewed as though it were a single server. (一组松散耦合的服务器，共同协作，可被看作是一台服务器)
- Clusters are usually deployed to improve speed, reliability and availability over that provided by a single server, while typically being much more cost-effective than single computers of comparable speed or reliability. (用来改善速度、提高可靠性与可用性，降低成本)
- Load-balancing is a key issue in clusters. (负载平衡是集群里的一个关键要素)
- Physical cluster and logical cluster(物理集群、逻辑集群)

集群(Cluster)

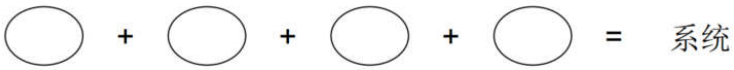


集群的方法

- 集群内各服务器上的内容保持一致(通过冗余提高可靠性与可用性)



- 集群内各服务器上的内容之和构成系统完整的功能/数据，是对系统功能集合/数据集合的一个划分(通过分布式提高速度与并发性)



A small catalog of architecture styles

体系结构风格	进一步细分	
数据流系统 (Data flow)	顺序批处理 (Batch Sequential)	
	管道和过滤器 (Pipes and Filters)	
调用-返回系统 (call and return)	主程序和子程序(Main program and subroutine)	
	OO 系统	
	分层结构 (Layered)	
独立组件 (Independent components)	通信过程 (Communicating Process)	
	事件系统	Implicit Invocation
		Explicit Invocation
虚拟机 (Virtual machine)	解释器 (Interpreter)	
	基于规则的系统 (Rule-Based System)	
数据为中心的系统 (Data-centered)	数据仓库 (Repository)	
	超文本系统 (Hypertext)	
	黑板 (Blackboard)	

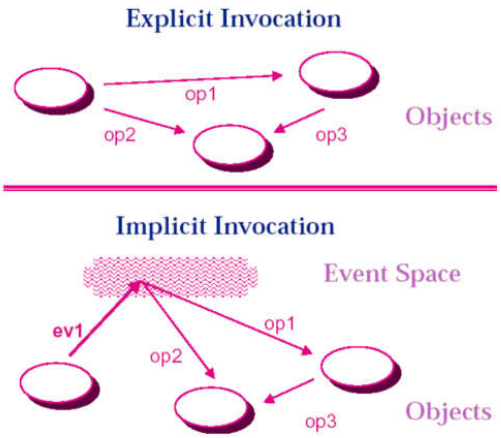
独立构件风格

- **Independent Components**
  - Communicating Processes
  - Event Systems
    - Implicit Invocation
    - Explicit Invocation

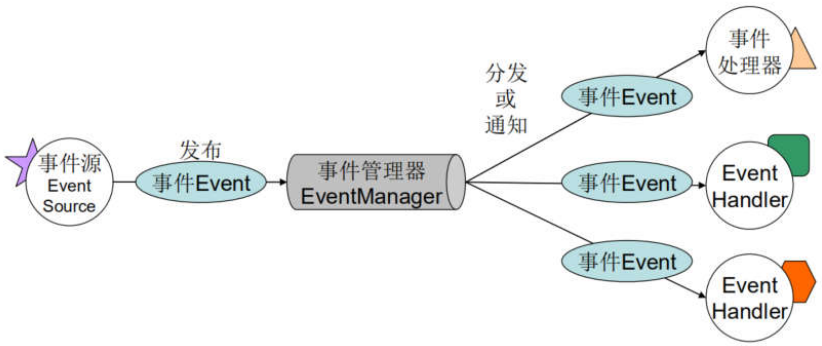
	完整性	组合性	抽象性	重用性	构建性	异构性	可分析性
独立构件				+	+	+	

显式调用 vs. 隐式调用

- 显式调用：
  - 各个构件之间的互动是由显性调用函数或程序完成的。
  - 调用过程与次序是固定的、预先设定的。
- 隐式调用：
  - 调用过程与次序不是固定的、预先未知；
  - 各构件之间通过事件的方式进行交互；



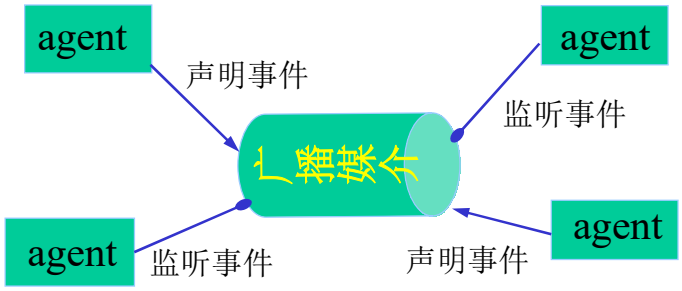
事件系统的基本构成



功能	描述
分离的交互	事件发布者并不会意识到事件订阅者的存在。
多对多通信	采用发布/订阅消息传递，一个特定事件可以影响多个订阅者。
基于事件的触发器	控制流由接收者确定（基于发布的事件）。
异步	通过事件消息传递支持异步操作。

基于事件的隐式调用风格

- 基本结构



- **特点**：此类系统中，构件不直接调用一个过程，而是声明或广播一个或多个事件。系统中的其他构件可以将某一过程注册为与它所关心的事件相关联。当某一事件发生时，系统会调用所与之关联的过程，即一个事件的激发隐含地导致了对其他模块的过程的调用。
- **构件**：是这样的模块，其接口不仅提供一个过程的集合，还提供一个事件的集合。构件可以声明或广播一个或多个事件，或者向系统注册，来表明它希望相应的事件
- **连接件**：有两类一对事件的显示或隐式调用。

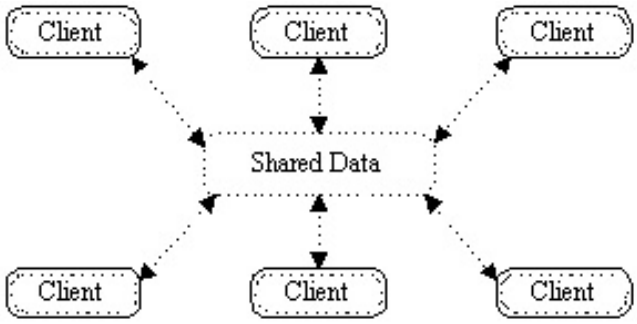
基于事件的隐式调用风格

- **适用**于涉及低耦合构件集合的应用程序，尤其是必须动态重配置的应用程序。
- **优点**：
  - 事件广播者不必知道哪些部件会被事件影响，部件之间关系弱。
  - 隐式调用有利于软件重用，因为允许任何构件注册相关事件；
  - 修改构件的接口并不影响别的构件，因而系统的演化、升级变得简单
- **缺点**：
  - 构件不再拥有计算的控制权，一旦触发一个事件，它不能确定响应的构件，也无法知道相关过程执行的顺序；在共享数据存储库的系统中，资源管理器的性能和准确度成为十分关键的因素。
  - 数据交换问题；
  - 被调用的过程的语义依赖于被触发的事件的上下文约束，很难对系统的正确性进行推理。
- **典型实例**：
  - 最早出现在守护进程、约束满足性检查和包交换网络等方面
  - 常用于：开发环境中集成各种工具、在用户界面中分离数据和表示
  - 典型实例：Windows的消息机制

A small catalog of architecture styles

体系结构风格	进一步细分	
数据流系统 (Data flow)	顺序批处理 (Batch Sequential)	
	管道和过滤器 (Pipes and Filters)	
调用-返回系统 (call and return)	主程序和子程序(Main program and subroutine)	
	OO 系统	
	分层结构 (Layered)	
独立组件 (Independent components)	通信过程 (Communicating Process)	
	事件系统	Implicit Invocation
		Explicit Invocation
虚拟机 (Virtual machine)	解释器 (Interpreter)	
	基于规则的系统 (Rule-Based System)	
数据为中心的系统 (Data-centered)	数据仓库 (Repository)	
	超文本系统 (Hypertext)	
	黑板 (Blackboard)	

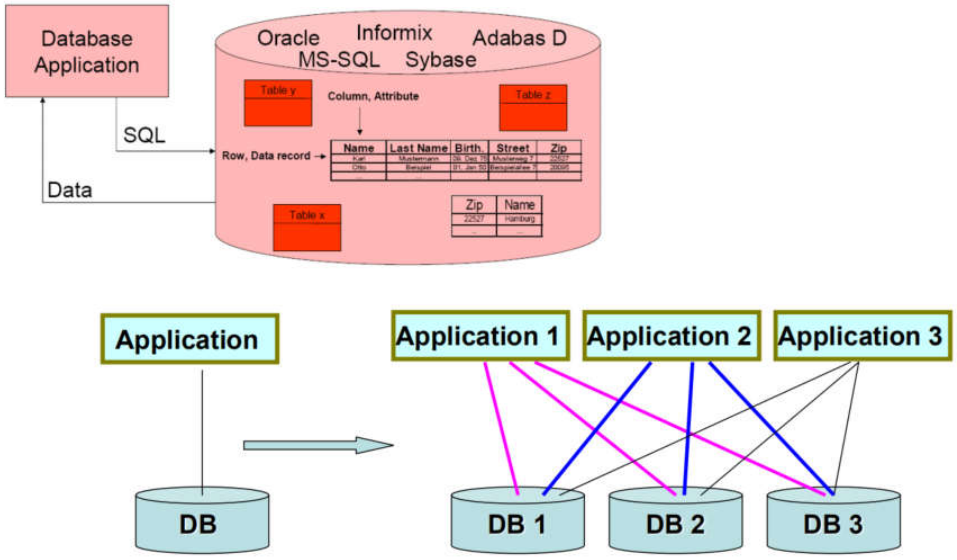
数据中心风格—仓库风格



The data-centered style.

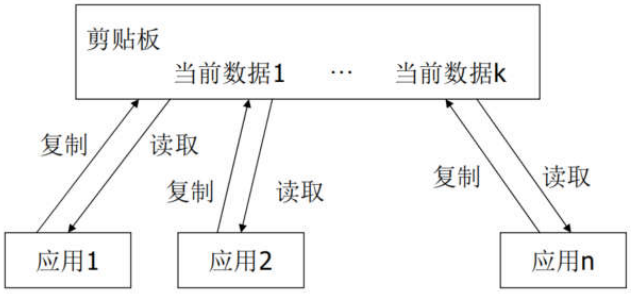
- 目标：获得数据的易集成性
- 仓库(Repositories)风格的体系结构含两类构件：
  - 一个中央数据结构，表示当前状态
  - 一个独立构件的集合，对中央数据结构进行操作

示例1：基于数据库的系统结构

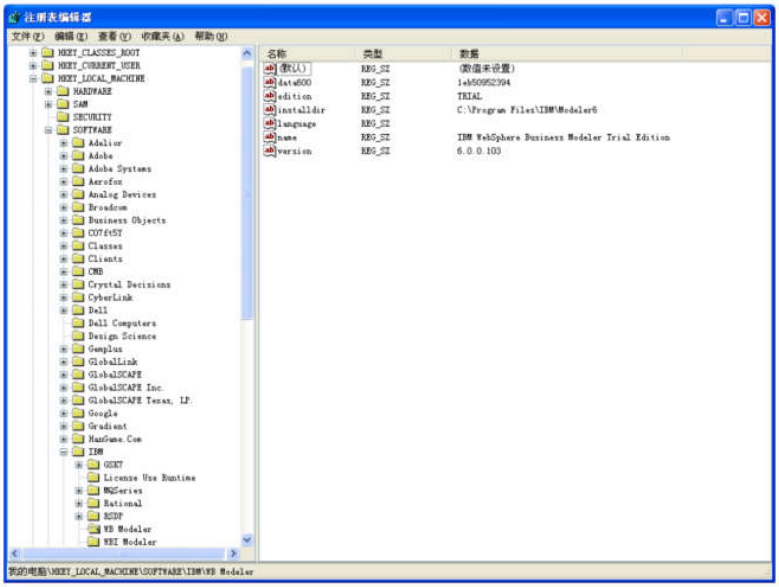


例2：剪贴板 (Clipboard)

- 剪贴板是一个用来进行短时间的数据存储并在文档/应用之间进行数据传递和交换的软件程序
  - 用来存储带传递和交换信息的公共区域(形成共享数据仓库)；
  - 不同的应用程序通过该区域交换格式化的信息；
  - 访问剪贴板的方式：copy & paste.

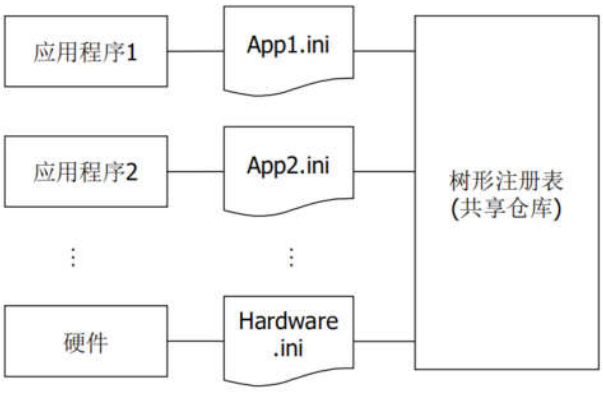


例3 注册表

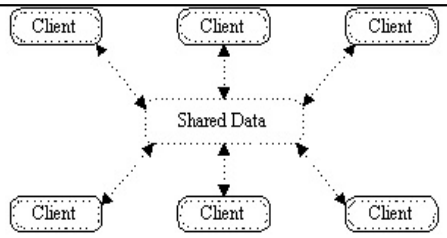


注册表的结构

- 最初，硬件/软件系统的配置信息均被各自保存在一个配置文件中(.ini)；
- 这些文件散落在系统的各个角落，很难对其进行维护；
- 为此，引入注册表的思想，将所有.ini文件集中起来，形成共享仓库，为系统运行起到了集中的资源配置管理和控制调度的作用。



- 注册表中存在着系统的所有硬件和软件配置信息，如启动信息、用户、BIOS、各类硬件、网络、INI文件、驱动程序、应用程序等；
- 注册表信息影响或控制系统/应用软件的行为，应用软件安装/运行/卸载时对其进行添加/修改/删除信息，以达到改变系统功能和控制软件运行的目的。

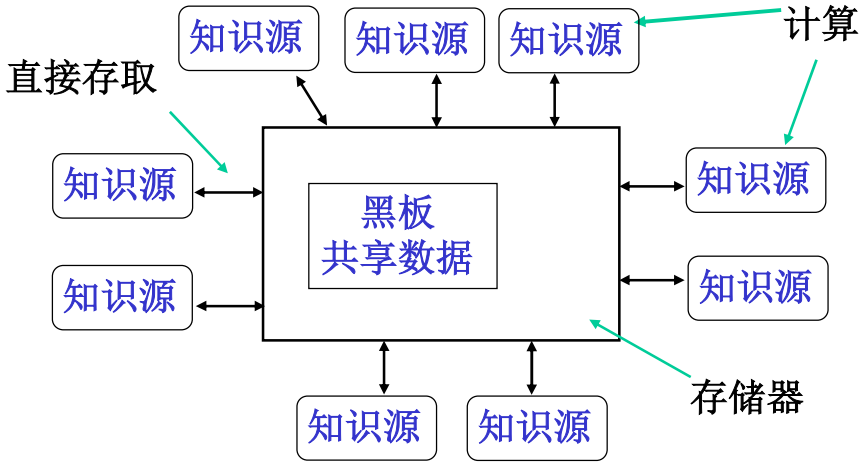


The data-centered style.

- 分类（对数据和状态的控制方法）
  - Passive Repository，如传统的数据库体系结构：由输入事务选择进行何种处理，并将执行结果作为当前状态存储到中央数据结构中。此时，仓库是传统的数据库体系结构。
  - Active Repository，如 blackboard—黑板体系结构：由中央数据结构的当前状态决定进行何种处理。
- 黑板体系结构是仓库体系结构的特殊化。
  - 传统上用于在信号处理方面进行复杂解释的应用程序，以及松散耦合的构件访问共享数据的应用程序；
  - 适用于：需要解决冲突并处理可能存在的不确定性。

黑板系统的组成

- **知识源：** 指特定应用程序知识的独立片断。知识源之间的交互只在黑板内部发生。知识源代理独立进行自己的处理，可能向知识源添加知识，供其它知识源开展工作。
- **黑板数据结构：** 是反映应用程序求解状态的数据。知识源不断地对黑板数据进行修改，直到得出问题的解。——是知识源间的通讯机制。
- **控制器：** 控制（即对知识源的调用）是由黑板的状态决定的。一旦黑板数据的改变使得某个知识源成为可用，该知识源就被激活。



黑板风格的体系结构

- 特点：
  - 追求的是可能随时间变化的目标，各个代理需要不同的资源、关心不同的问题，但用一种相互协作的方式使用和维护共享数据结构。
  - 知识源之间的交互只通过黑板完成
  - 问题的解决是通过知识源不断地改变黑板完成的
- 优点：
  - 便于多客户共享大量数据，而不关心数据何时有、何时提供、如何提供；
  - 既便于添加新的知识源代理（应用程序），也便于扩展共享的黑板数据结构
-

黑板风格的体系结构

- 缺点
  - 不同的知识源代理对于共享数据结构要达成一致；同时也导致对黑板数据结构的修改困难（要考虑各个代理）
  - 需要一定的同步/加锁机制保证数据结构的完整性和一致性，增大了系统的复杂性
- 例子
  - 主要用于人工智能应用系统，如：语音识别、模式识别、三维分子结构建模。

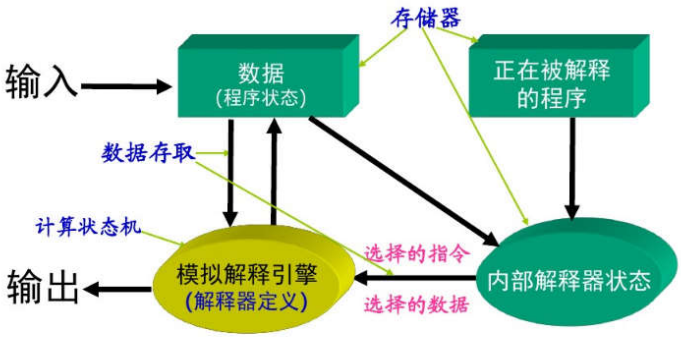
北京航空航天大学软件工程研究所

A small catalog of architecture styles

体系结构风格	进一步细分	
数据流系统 (Data flow)	顺序批处理 (Batch Sequential)	
	管道和过滤器 (Pipes and Filters)	
调用-返回系统 (call and return)	主程序和子程序(Main program and subroutine)	
	OO 系统	
	分层结构 (Layered)	
独立组件 (Independent components)	通信过程 (Communicating Process)	
	事件系统	Implicit Invocation
		Explicit Invocation
虚拟机 (Virtual machine)	解释器 (Interpreter)	
	基于规则的系统 (Rule-Based System)	
数据为中心的系统 (Data-centered)	数据仓库 (Repository)	
	超文本系统 (Hypertext)	
	黑板 (Blackboard)	

虚拟机结构/解释器风格

基本结构

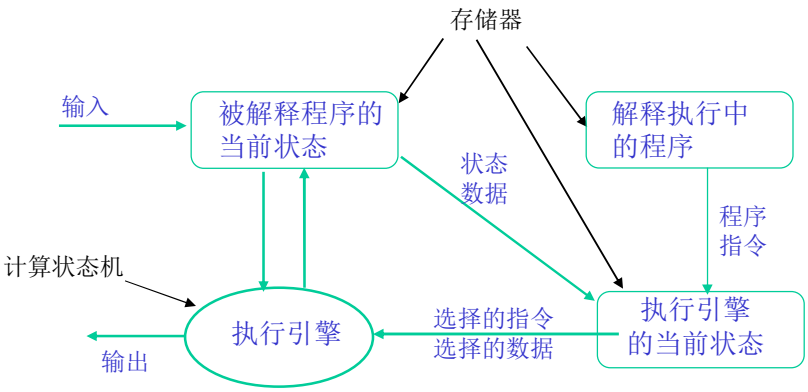


Interpreters/Virtual Machine

- 作用：常被用于建立一种虚拟机以弥合程序的语义合作为计算引擎的硬件之间的间隙。
- 构件：四个构件——作为执行引擎的状态机和三个存储器
- 连接件：过程调用和直接存储访问。

虚拟机结构/解释器风格

基本结构



- 作用：常被用于建立一种虚拟机以弥合程序的语义合作为计算引擎的硬件之间的间隙。
- 构件：四个构件——作为执行引擎的状态机和三个存储器
- 连接件：过程调用和直接存储访问。

虚拟机结构/解释器风格

- 优点：
  - 有助于应用程序的可移植性和程序设计语言的跨平台能力
  - 可以对未实现的硬件进行仿真（因为实际测试可能是复杂、昂贵或危险的）
- 缺点
  - 额外的间接层次带来了系统性能的下降。
- 实例
  - 适用于应用程序不能直接运行在最合适的机器上，或不能直接以最适合的语言执行。
  - 典型的例子是专家系统、基于规则的系统
  - 程序设计语言的解释器，如Java、Smalltalk等
  - 脚本语言，如Awk、Perl等

北京航空航天大学软件工程研究所

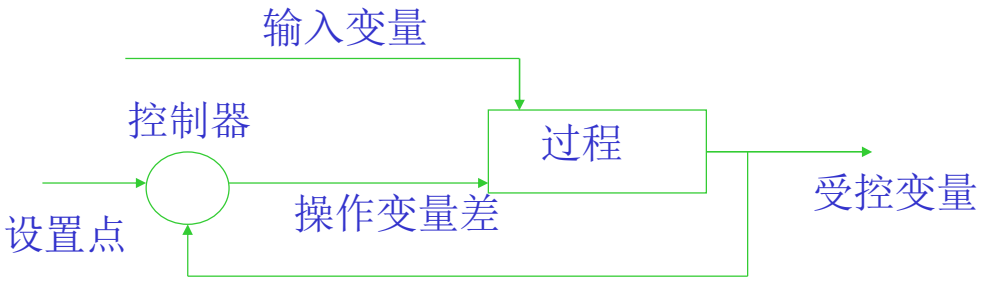
Process Control（Loop）

（1）背景

当软件被用来操作一个物理系统时，软件与硬件之间可以粗略地表示为一个反馈循环，这个反馈循环通过接受一定的输入，确定一系列的输出，最终使环境达到一个新的状态。

适合于嵌入式系统，涉及连续的动作与状态。

（2）控制系统的定义



Process Control（Loop）

（3）控制系统模型的构成

计算型模型

过程定义：包括操作某些过程变量的机制

控制算法：用来决定如何操纵过程变量

数据元素

过程变量：指定的输入、操纵变量等

设置点：受控变量的参考值

传感器：用于获得控制所需的过程变量值

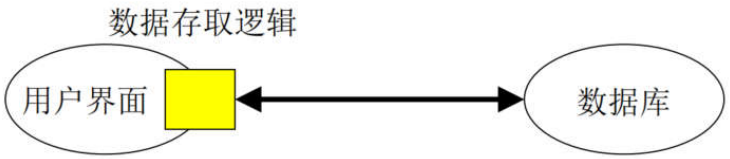
控制环模型

建立控制算法之间的关系，它收集关于过程实际的和欲达到的状态，并调节过程变量，以使实际状态向目标状态发展

北京航空航天大学软件工程研究所

“模型-视图-控制器”(MVC)

C/S体系结构存在的问题



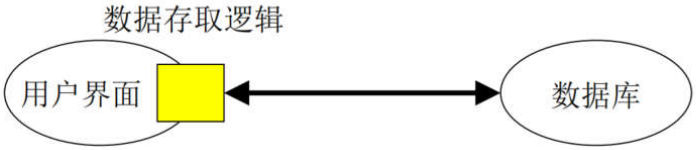
- 许多应用系统的用途都是从数据库存储检索数据并将其显示给用户。
- 在用户更改数据之后，系统再将更新内容存储到数据存储中。
- 因为关键的信息流发生在数据存储和用户界面之间，所以一般倾向于将这两部分捆绑在一起，以减少编码量并提高应用程序性能。

C/S体系结构存在的问题

- 但是，这种看起来自然而然的方法有一些问题。
  - 用户界面的更改往往比数据存储的更改频繁得多。
  - 除了数据存取功能之外，应用程序的业务逻辑中往往还会包含很多的其他业务逻辑。

...考虑(CCP) The Common Closure Principle 共同封闭原则

...考虑(SIP) The Single Responsibility Principle 单一责任原则



B/S体系结构存在的问题

- 不采用C/S结构，而采用纯粹的B/S结构，会不会解决这个问题？
  - ...用户界面仍然需要显式的调用功能层的业务逻辑...
  - ...仍然难以避免“用户界面的修改→业务逻辑的修改”的问题

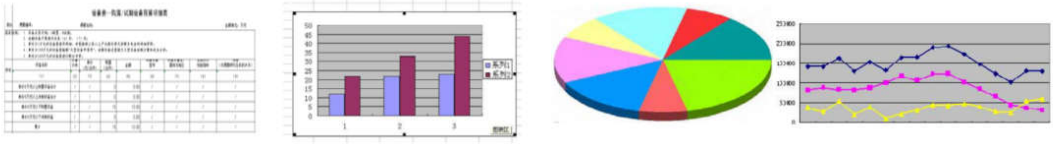
问题

- ——用户界面需要频繁的修改，它是“不稳定”的。
- ——业务逻辑/数据 与 用户界面 之间应尽量少的避免直接通信。

- 问题：如何让 Web 应用程序的用户界面与业务逻辑功能实现模块化，以便使程序开发人员可以轻松地单独修改各个部分而不影响其他部分？

影响因素

- 在基于 Web 的应用程序中，用户界面逻辑的更改往往比业务逻辑频繁。
  - 例如，可能添加新的用户界面页，或者可能完全打乱现有的页面布局。
- 如果将UI代码和业务逻辑组合一起并放在UI中，则每次更改用户界面时，都有可能引起对业务逻辑的修改。
  - 这很可能引入错误，并需要在对用户界面进行每个极小更改之后都要重新测试所有业务逻辑。
- 在某些情况下，应用程序以不同的方式显示同一数据。



  - 如果用户在一个视图中更改了数据，则系统必须自动更新该数据的其他所有视图。

软件体系结构 SEI/BUAA
创建软件体系结构

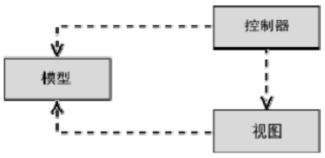
- 与业务逻辑相比，用户界面代码对设备的依赖性往往更大。
  - 如果要将应用程序从基于浏览器的应用程序迁移到智能手机上，则必须替换很多用户界面代码，而业务逻辑可能不受影响。
  - 这两部分的完全分离可以使迁移更快完成，并最大限度地降低将错误引入业务逻辑的风险。
- 设计美观而有效的 用户界面(如HTML、JSP等)通常要求采用一套与开发复杂业务逻辑不同的编程技能。
  - 很少有人同时具有这两种技能。
  - 将这两部分的开发工作分隔开来是比较理想的。
- 通常，为用户界面创建自动测试比为业务逻辑更难、更耗时。
  - 因此，减少直接绑定到用户界面中的代码量可提高应用程序的可测试性。

解决方案：Model-View-Controller (MVC)

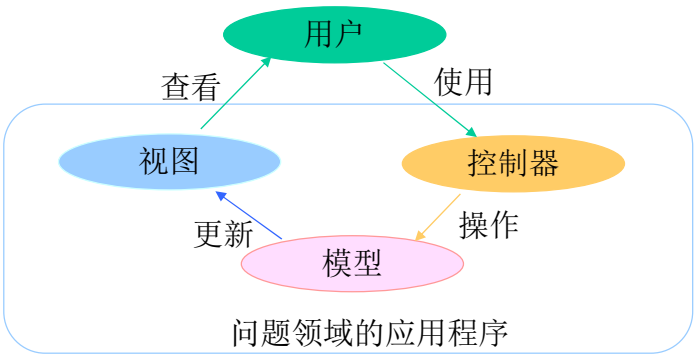
•MVC风格主要处理软件用户界面开发所面临的问题。
—界面变动大，不同系统平台不一样。

- **MVC**是一种软件体系结构，它将应用程序的数据模型/业务逻辑、用户界面分别放在独立的构件中，从而对用户界面的修改不会对数据模型/业务逻辑造成很大影响。

- **MVC**在传统的**B/S**体系结构的基础上加入了一个新的元素：控制器，由控制器来决定视图与模型之间的依赖关系。



模型—视图—控制器风格



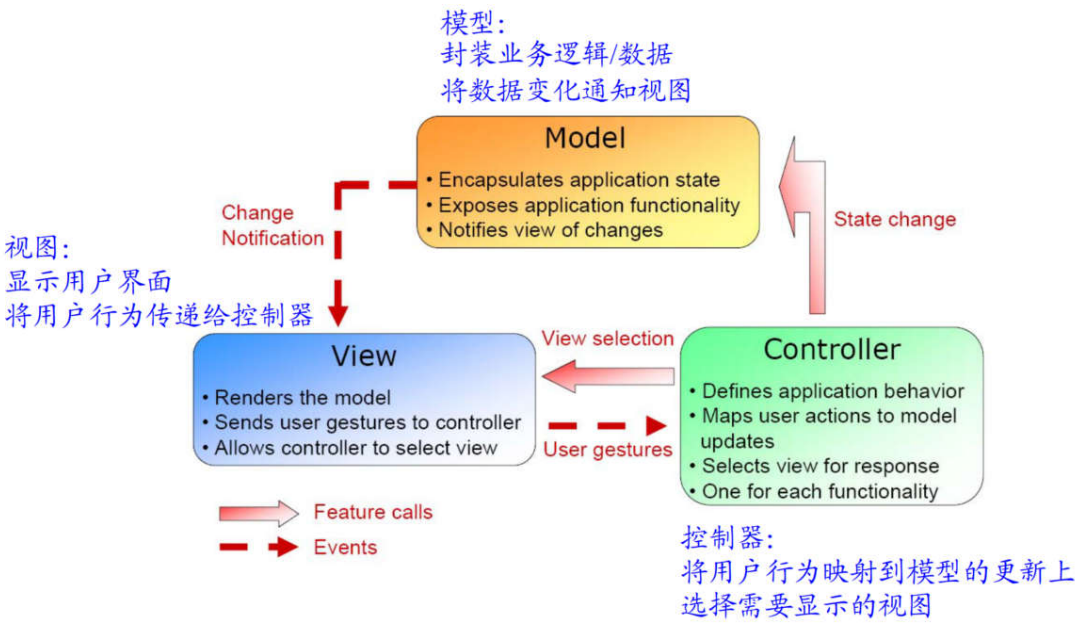
- 构件

– 模型(Model, M)：用于管理应用系统的行为和数据，并响应为获取其状态信息(通常来自视图)而发出的请求，还会响应更改状态的指令(通常来自控制器)；——对应于传统B/S中的业务逻辑和数据

– 视图(View, V)：用于管理数据的显示；——对应于传统B/S中的用户界面

– 控制器(Controller, C)：用于解释用户的鼠标和键盘输入，以通知模型和视图进行相应的更改。——在传统B/S结构中新增的元素

MVC运行机制

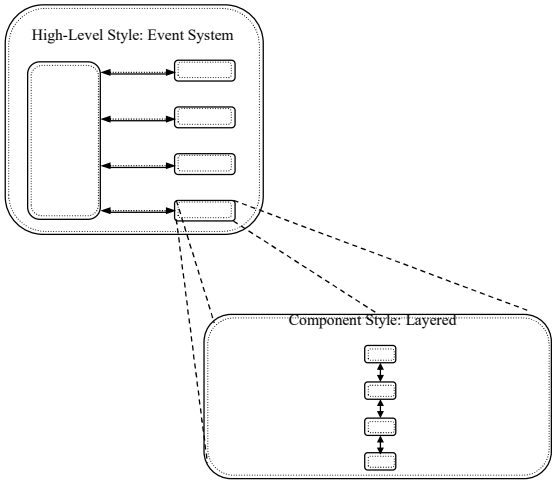


模型－视图－控制器风格

- 优点
  - 问题分解考虑，简化系统设计，保证系统的可扩展性；
  - 改变系统界面不影响应用程序的功能内核，易于演化，可维护性好；
  - 易于改变，甚至可能在运行时改变，提供了很好的动态机制。
- 缺点
  - 主要局限在应用软件的用户界面开发领域
- 典型实例
  - Windows应用程序的文档视图结构
  - 在SmallTalk和Java应用程序中常用到。

异质体系结构

- 异构是不可避免的
  - 一个系统的体系结构往往是不同风格的组合。
  - 不同风格的结构适合于不同的应用场合
  - 新系统需要和老系统协调工作



Hierarchical heterogeneity

- 组合的方法
  - 空间异质（Spatial/Locational heterogeneous）
    - 允许构件使用不同连接件的混合。系统以一种体系结构实现一个子系统，以另外一种体系结构实现另一个子系统。如：一个构件既可以通过其部分接口访问仓库，也可以用管道和系统中的其他构件交互，还可以通过其它接口接受中央信息。
  - 时间异质（Temporal/Simultaneously heterogeneous）
    - 系统在不同的时间采用不同的风格。如，一个管道过滤器系统的构件被实现为一个独立进程。整个系统的运行时体系结构就会随有无此进程的参与而发生变化。
  - 层次异质（Hierarchically heterogeneous）
    - 按照层次结构组好子。采用了某种体系结构风格的系统，其构件和连接件内部可以是另一种体系结构。例如，分层系统中的某一层可以用不同的风格实现。

参考书

- 很多体系结构的教材都有“体系结构风格”
- “面向模式的软件体系结构”，共3卷。机械出版社。