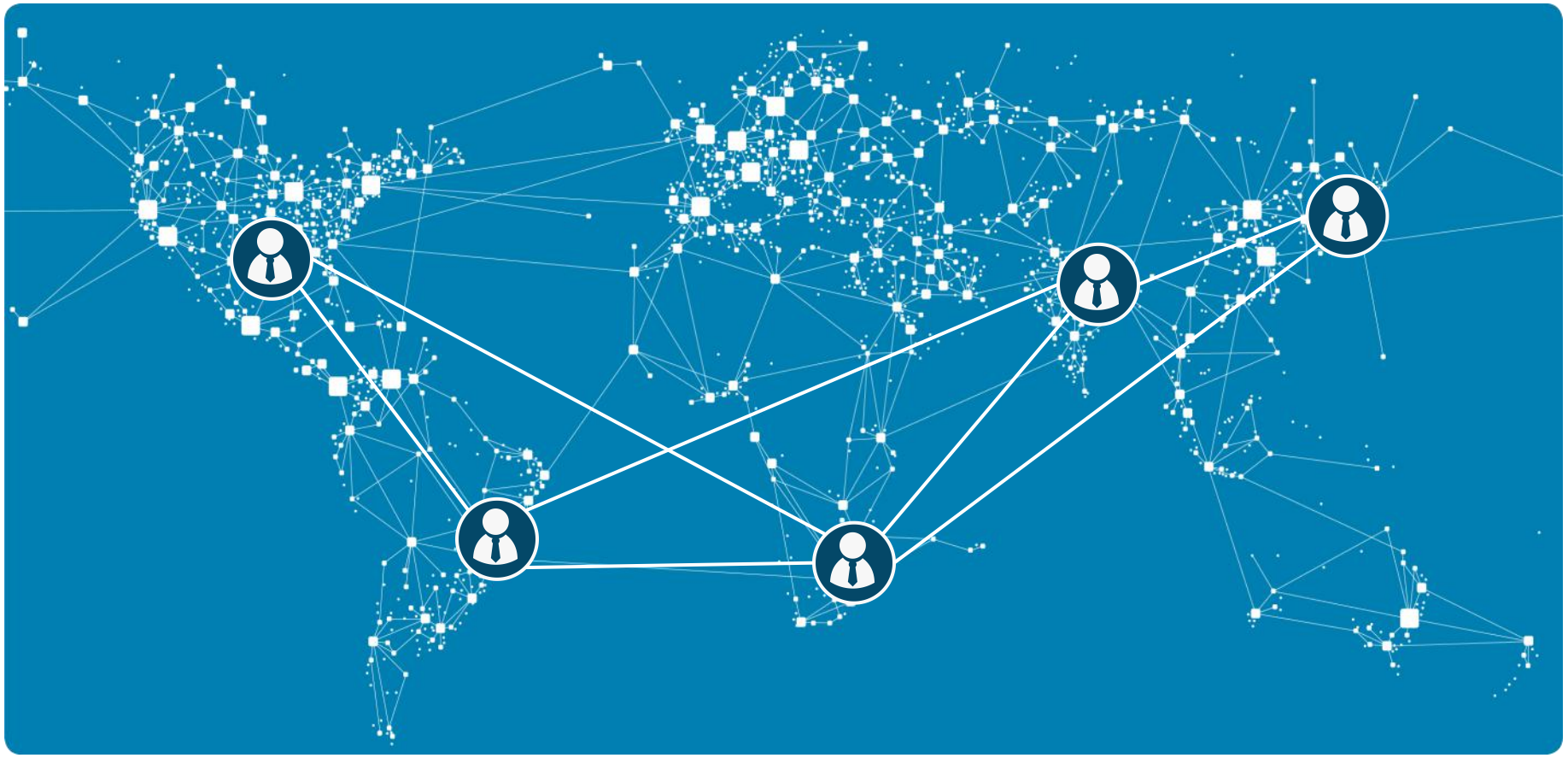


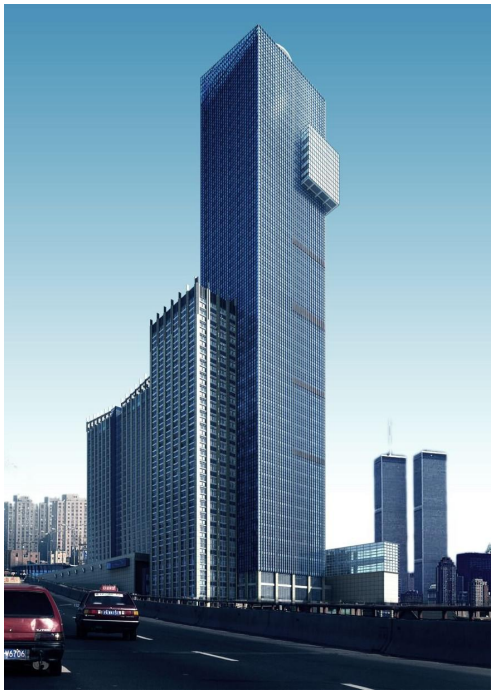
软件体系结构风格

从“建筑风格”谈起

建筑风格等同于建筑体系结构的一种可分类的模式，通过诸如外形、技术和材料等 形态上的特征加以区分。

法式建筑风格

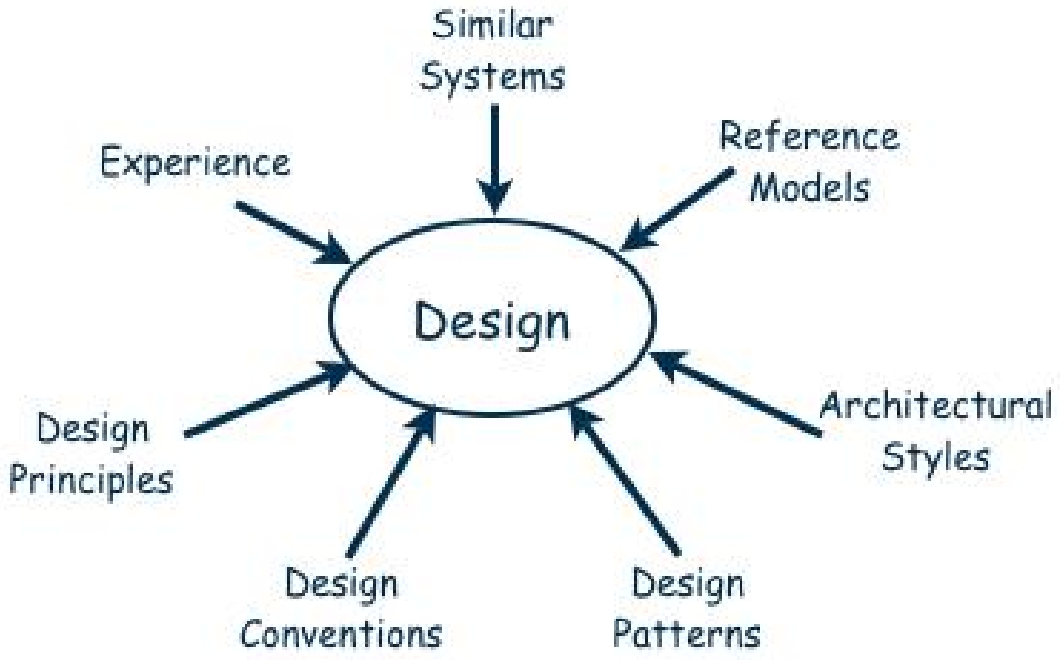
中式园林建筑风格



现代高层建筑风格

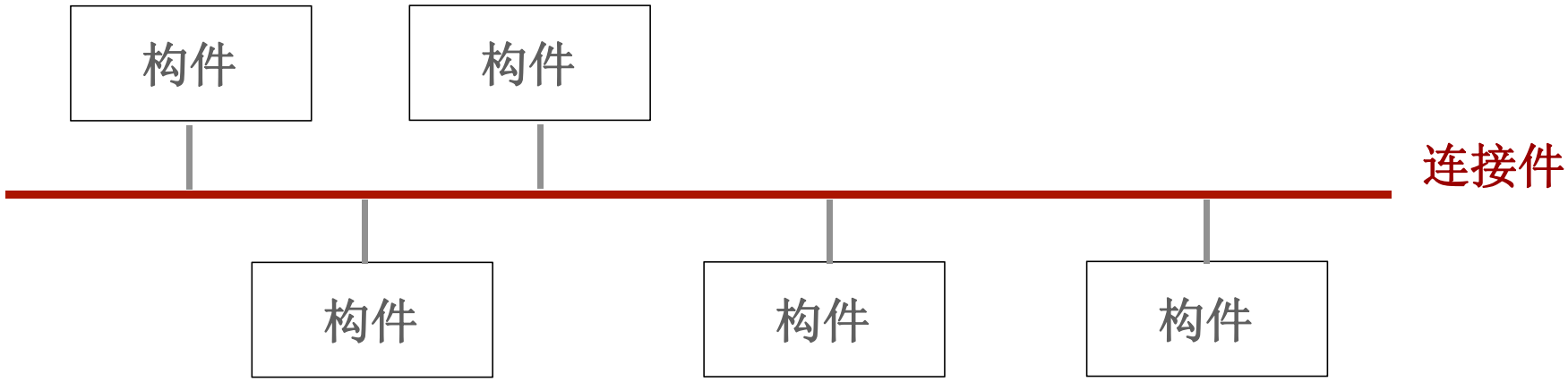
软件体系结构风格

大部分的软件设计是例行设计，即有经验的软件开发人员经常会借鉴现有的解决方案，将其改造成新的设计。

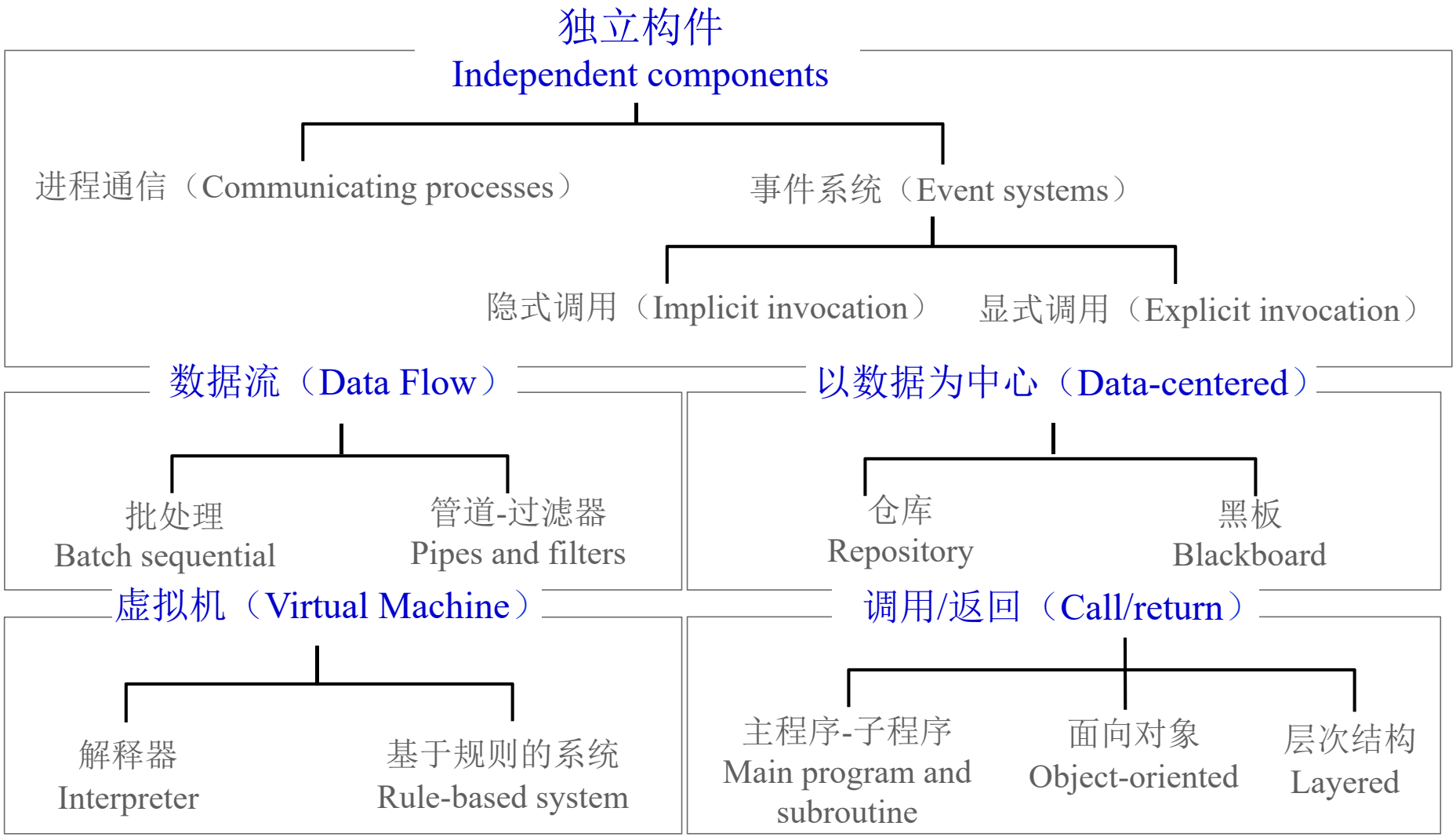


软件体系结构风格

软件体系结构风格（Architectural Styles）是描述特定系统组织方式的惯用范例，强调了软件系统中通用的组织结构。

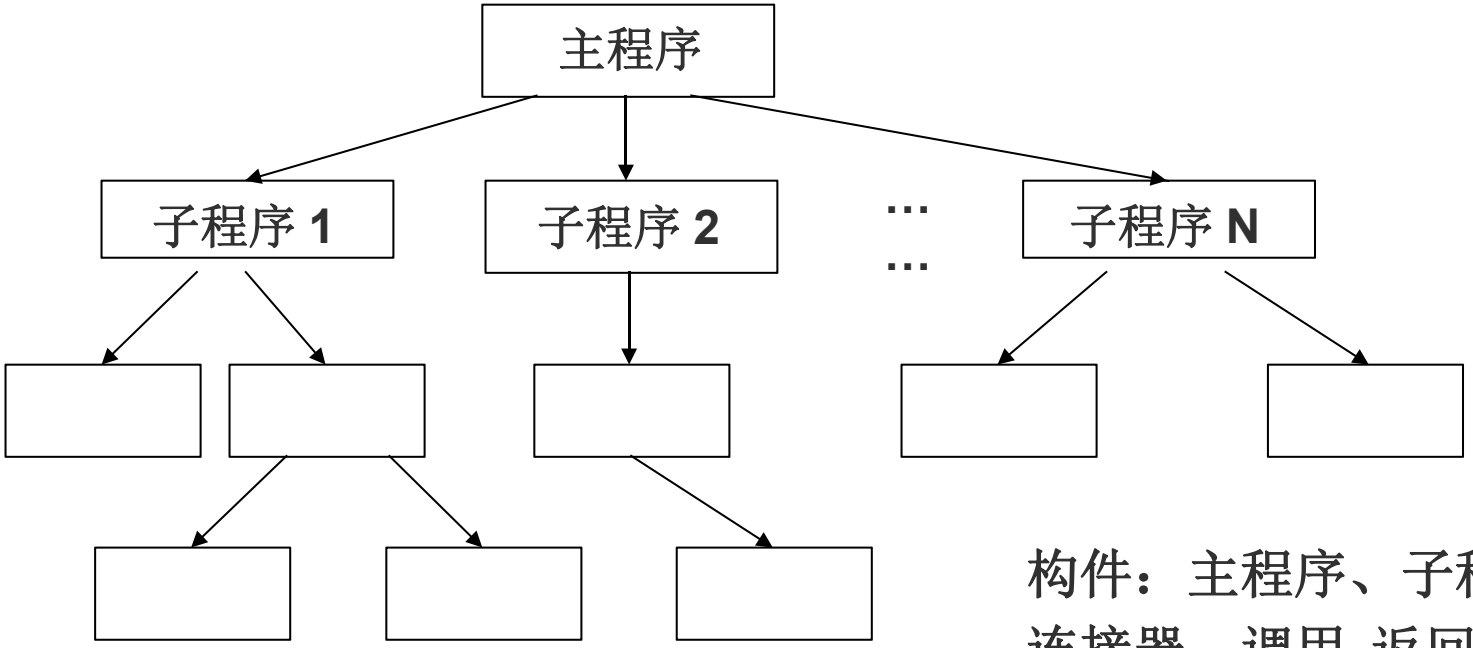


常见的体系结构风格


独立构件
Independent components
进程通信（Communicating processes）
事件系统（Event systems）
隐式调用（Implicit invocation）
显式调用（Explicit invocation）
数据流（Data Flow）
批处理
Batch sequential
管道-过滤器
Pipes and filters
以数据为中心（Data-centered）
仓库
Repository
黑板
Blackboard
虚拟机（Virtual Machine）
解释器
Interpreter
基于规则的系统
Rule-based system
调用/返回（Call/return）
主程序-子程序
Main program and subroutine
面向对象
Object-oriented
层次结构
Layered

主程序-子程序

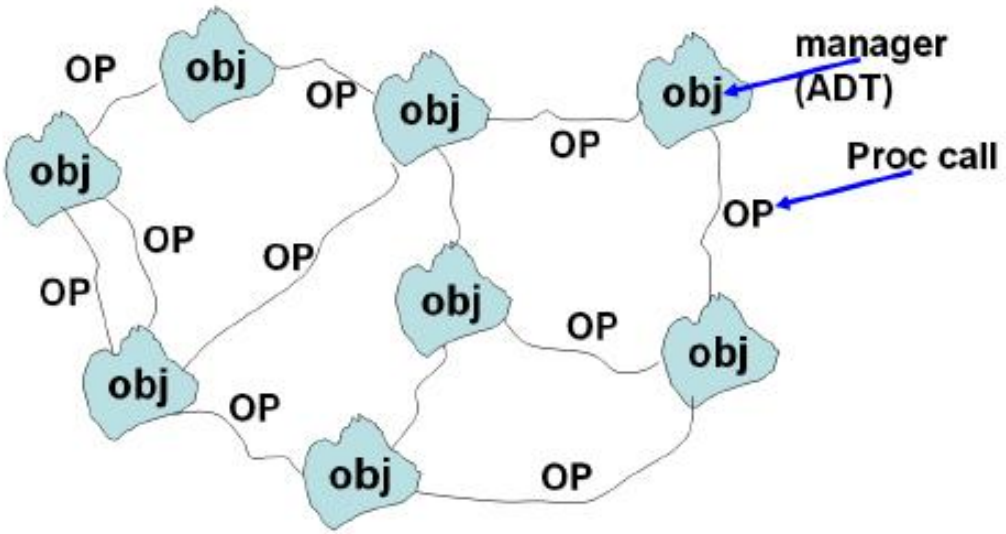
主程序--子程序风格是结构化程序设计的一种典型风格，从功能的观点设计系统，通过逐步分解和细化，形成整个系统的体系结构。



构件：主程序、子程序
连接器：调用-返回机制
拓扑结构：层次化结构

面向对象风格

- 系统被看作是对象的集合，每个对象都有一个它自己的功能集合；
- 数据及作用在数据上的操作被封装成抽象数据类型；
- 只通过接口与外界交互，内部的设计决策则被封装起来。

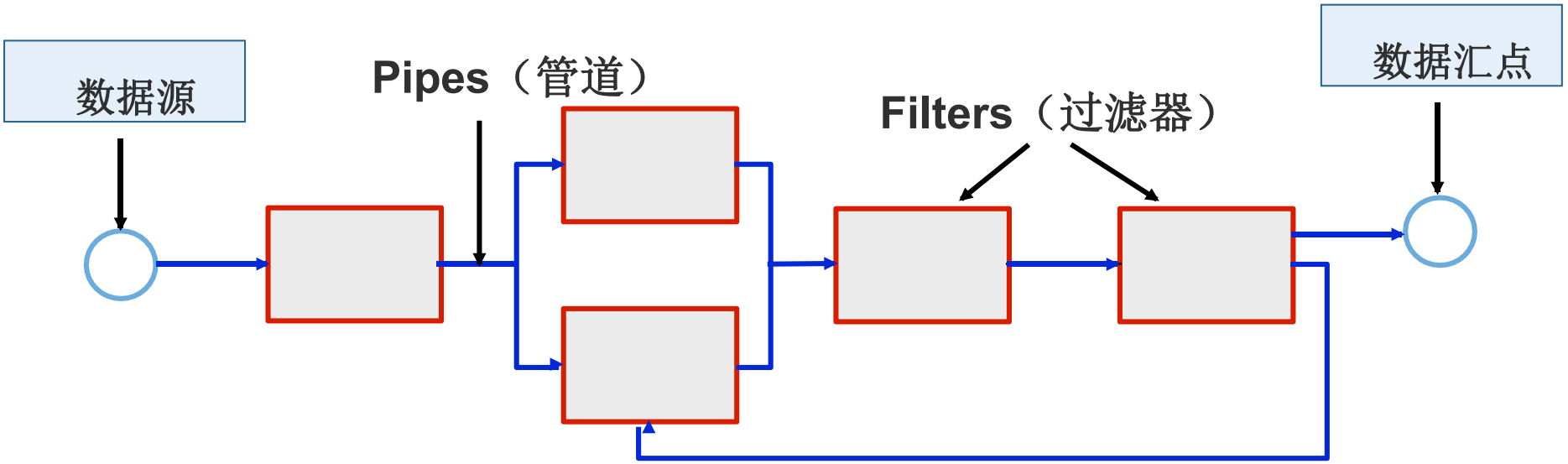


构件：类和对象

连接器：对象之间通过函数调用和消息传递实现交互

管道-过滤器风格

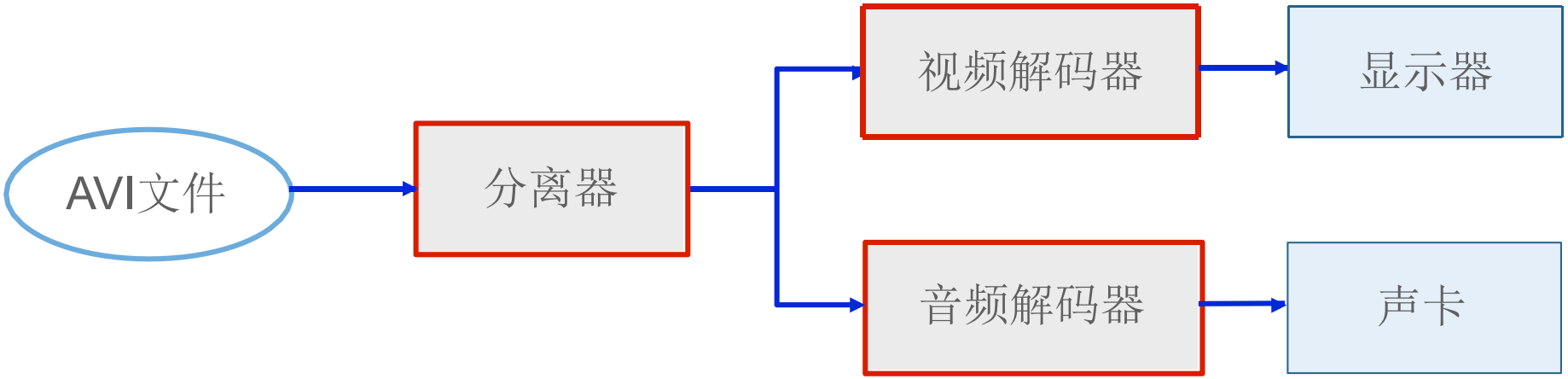
管道-过滤器风格把系统任务分成若干连续的处理步骤，这些步骤由通过系统的数据流连接，一个步骤的输出是下一个步骤的输入。



管道-过滤器风格

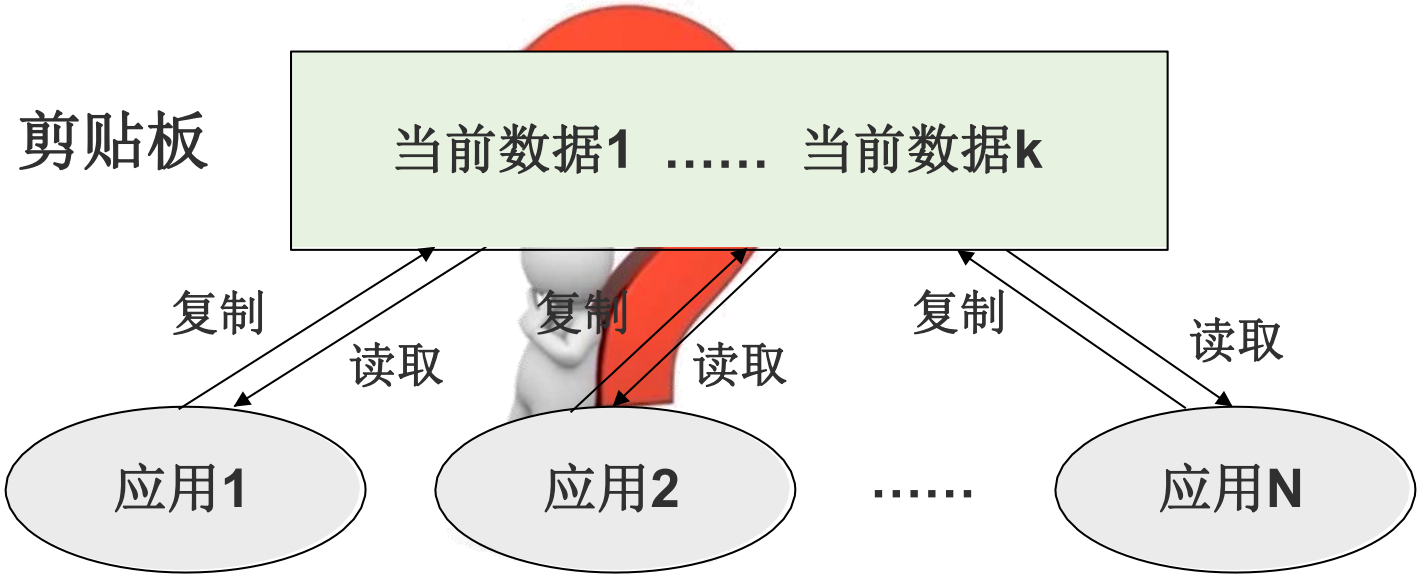
管道-过滤器风格把系统任务分成若干连续的处理步骤，这些步骤由通过系统的数据流连接，一个步骤的输出是下一个步骤的输入。

举例：媒体播放器



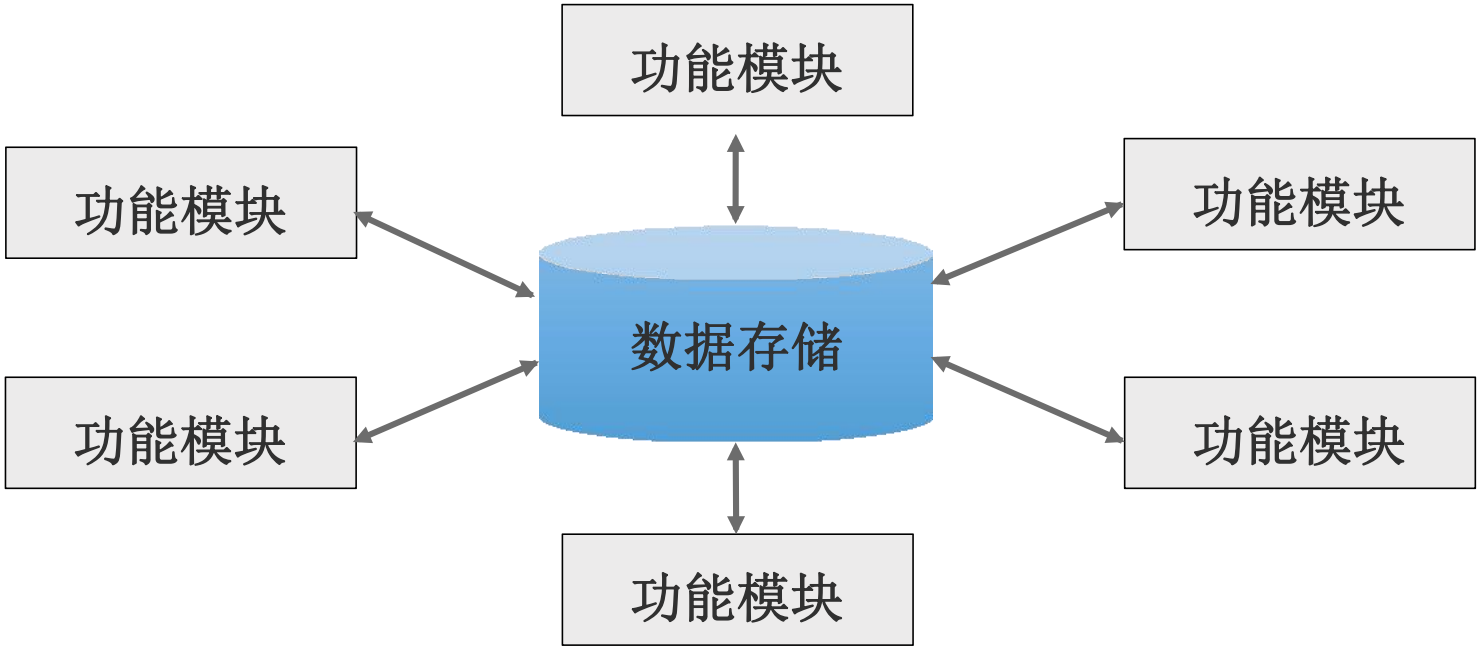
以数据为中心的风格

举例：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。

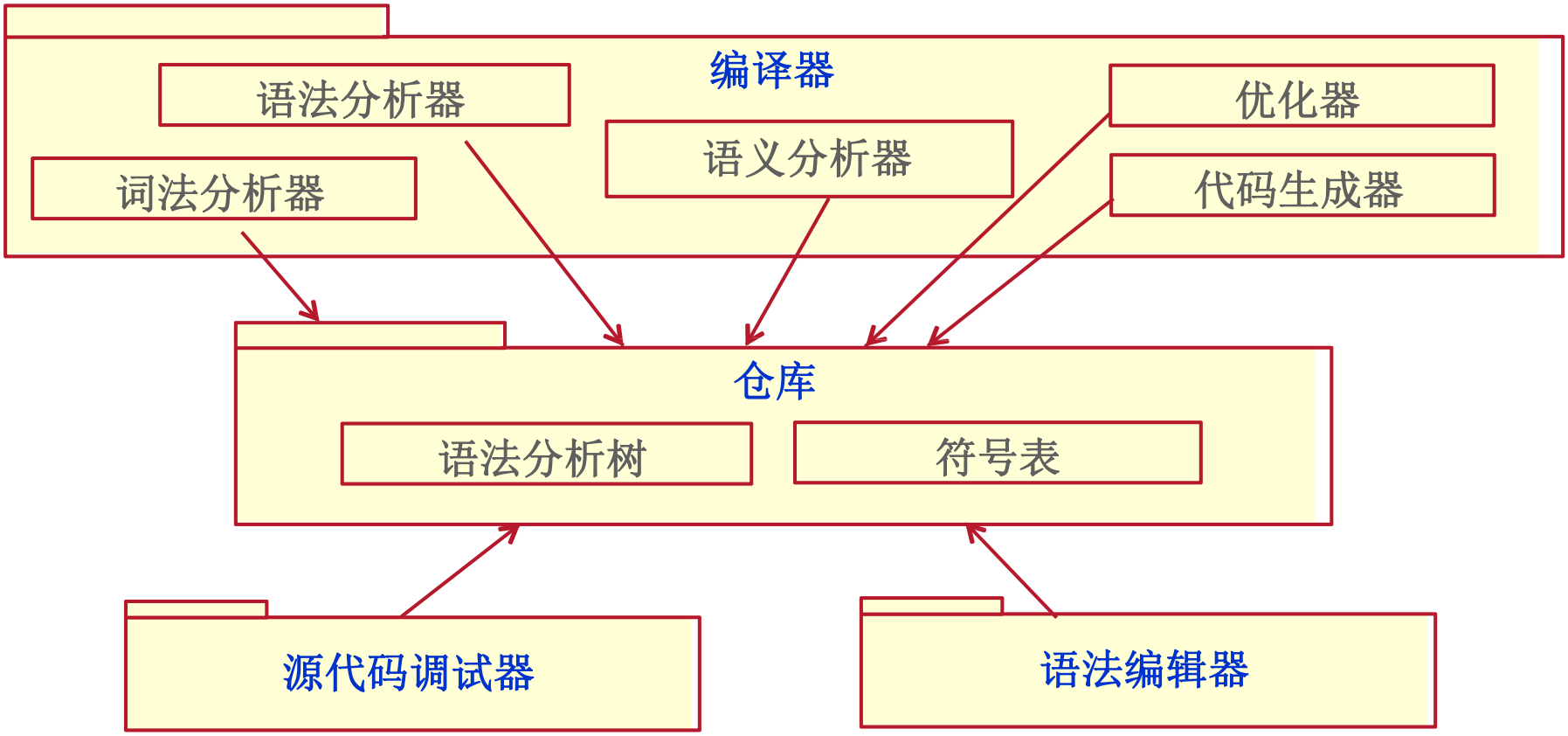


以数据为中心的风格

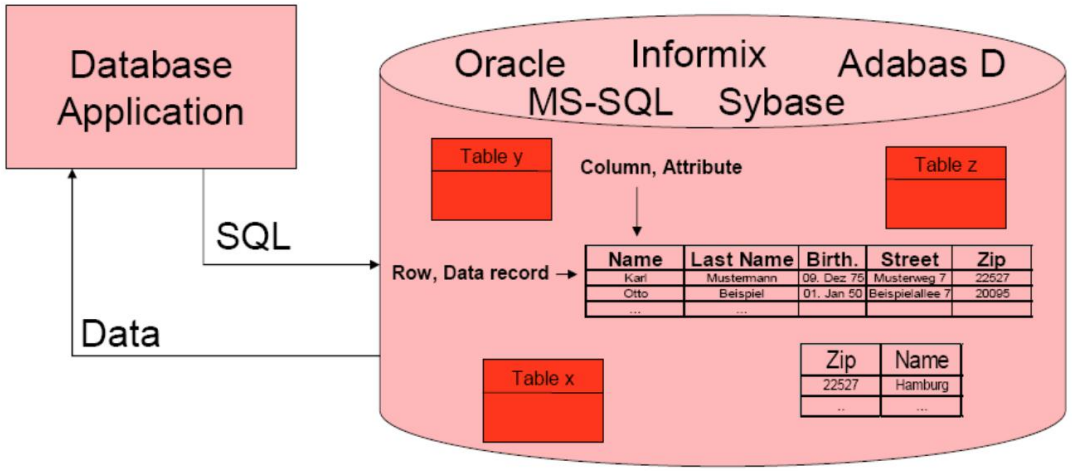
仓库体系结构（Repository Architecture）是一种以数据为中心的体系结构，适合于数据由一个模块产生而由其他模块使用的情形。

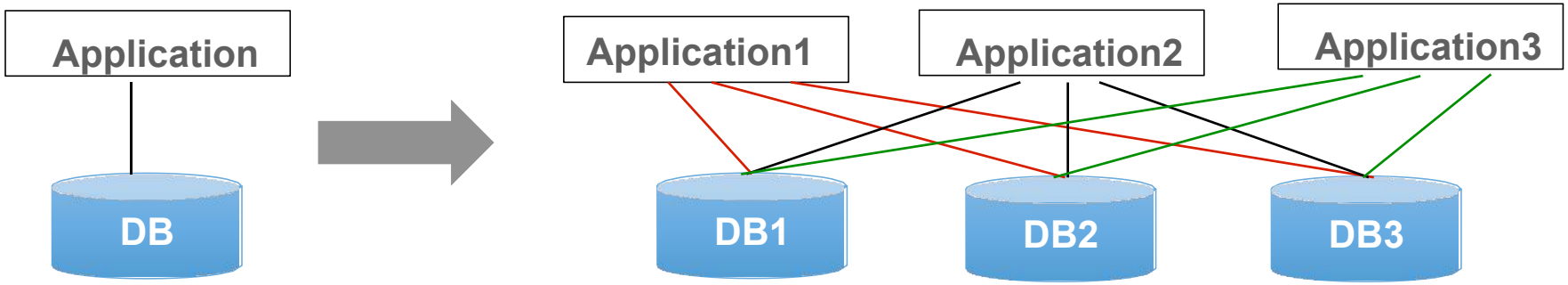


示例1：程序设计语言编译器

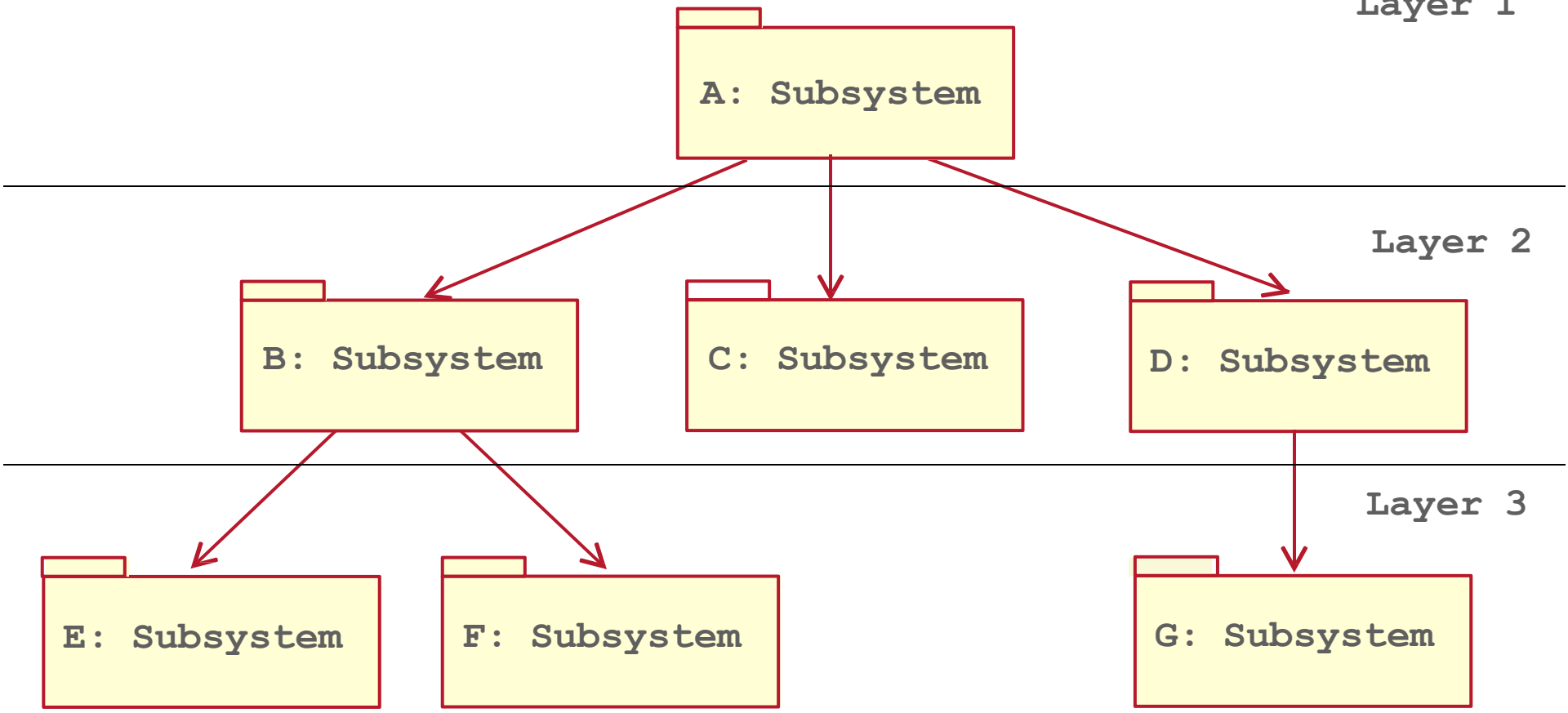


示例2：基于数据库的系统结构

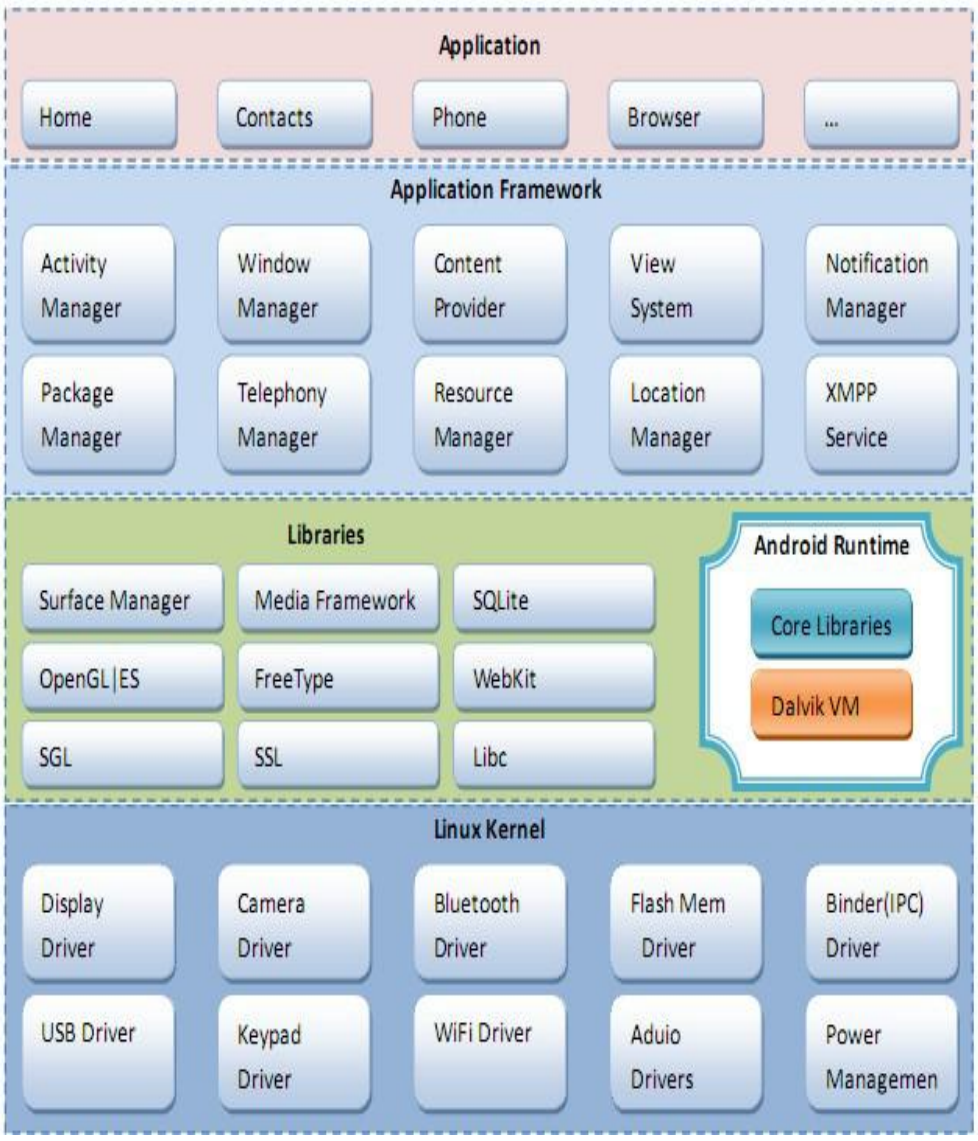




层次结构


Layer 1
A: Subsystem
Layer 2
B: Subsystem
C: Subsystem
D: Subsystem
Layer 3
E: Subsystem
F: Subsystem
G: Subsystem

示例1：安卓操作系统层次结构



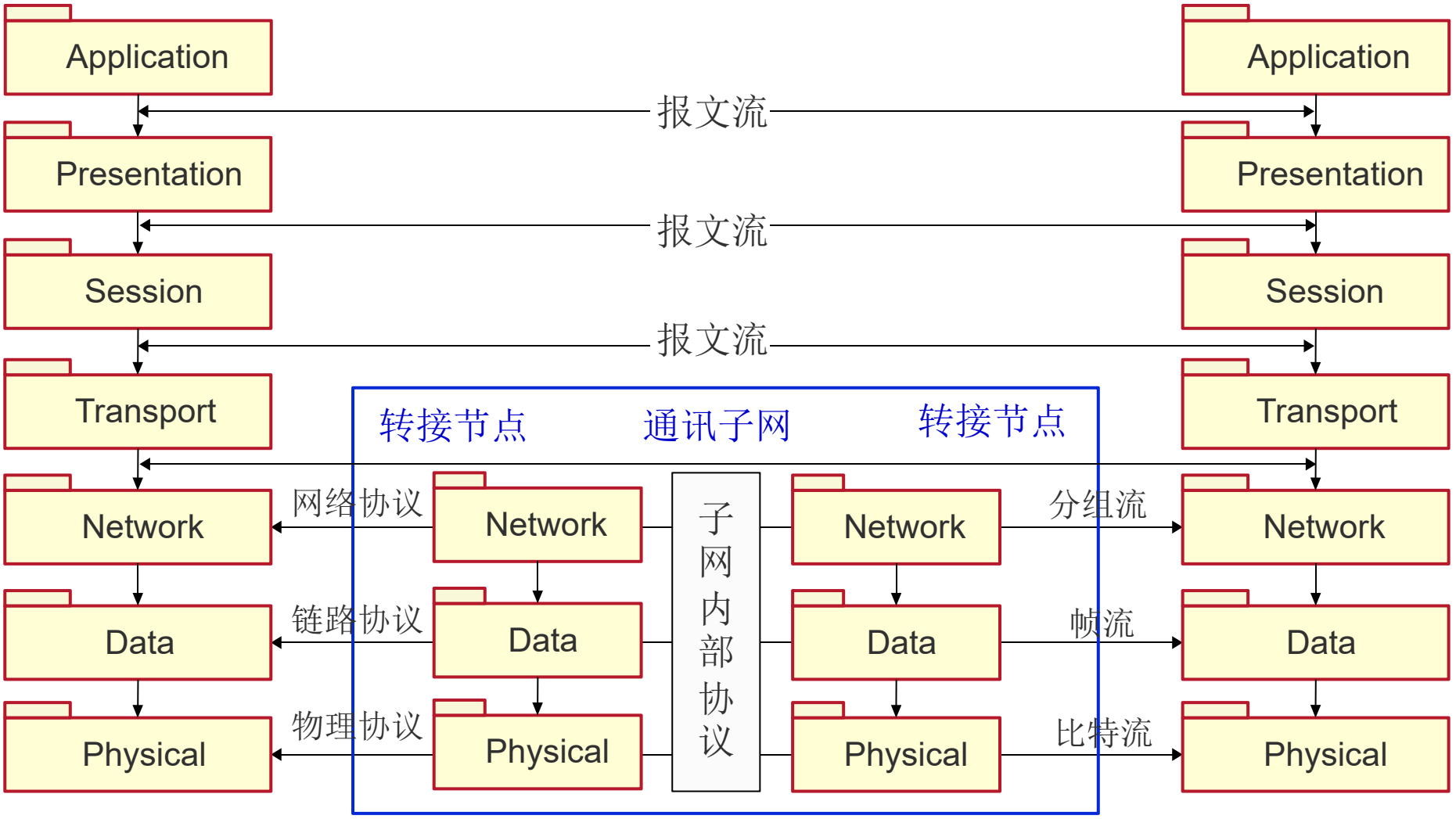
应用层：运行在虚拟机上的Java应用程序。

应用框架层：支持第三方开发者之间的交互，使其能够通过抽象方式访问所开发的应用程序需要的关键资源。

系统运行库层：为开发者和类似终端设备拥有者提供需要的核心功能。

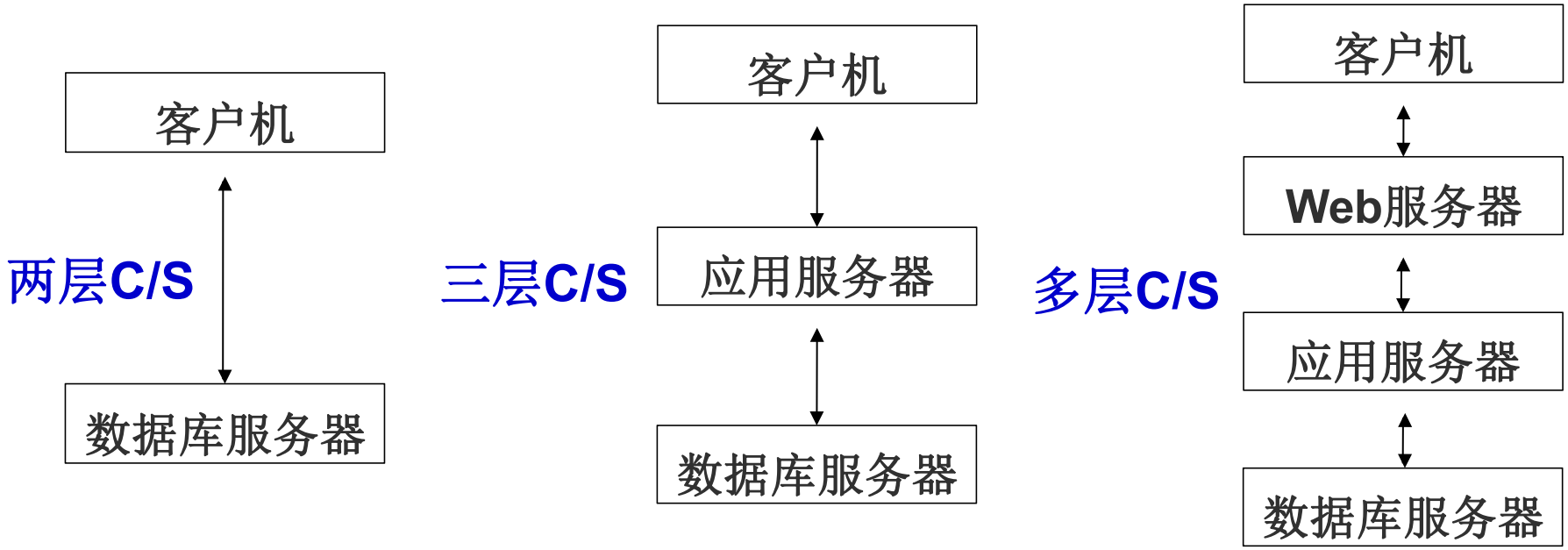
Linux内核层：提供启动和管理硬件以及Android应用程序的最基本的软件。

示例2：网络分层模型


Application
Application
报文流
Presentation
Presentation
报文流
Session
Session
报文流
Transport
Transport
转接节点
通讯子网
转接节点
Network
网络协议
Network
子网内部协议
Network
分组流
Network
Data
链路协议
Data
Data
帧流
Data
Physical
物理协议
Physical
Physical
比特流
Physical

客户机/服务器结构

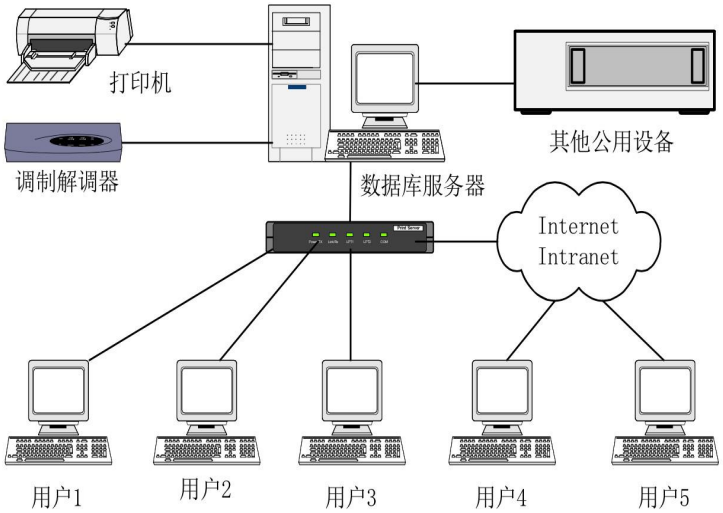
客户机 / 服务器体系结构（Client/Server）是一种分布式系统模型，作为服务器的子系统为其他客户机的子系统提供服务，作为客户机的子系统负责与用户的交互。

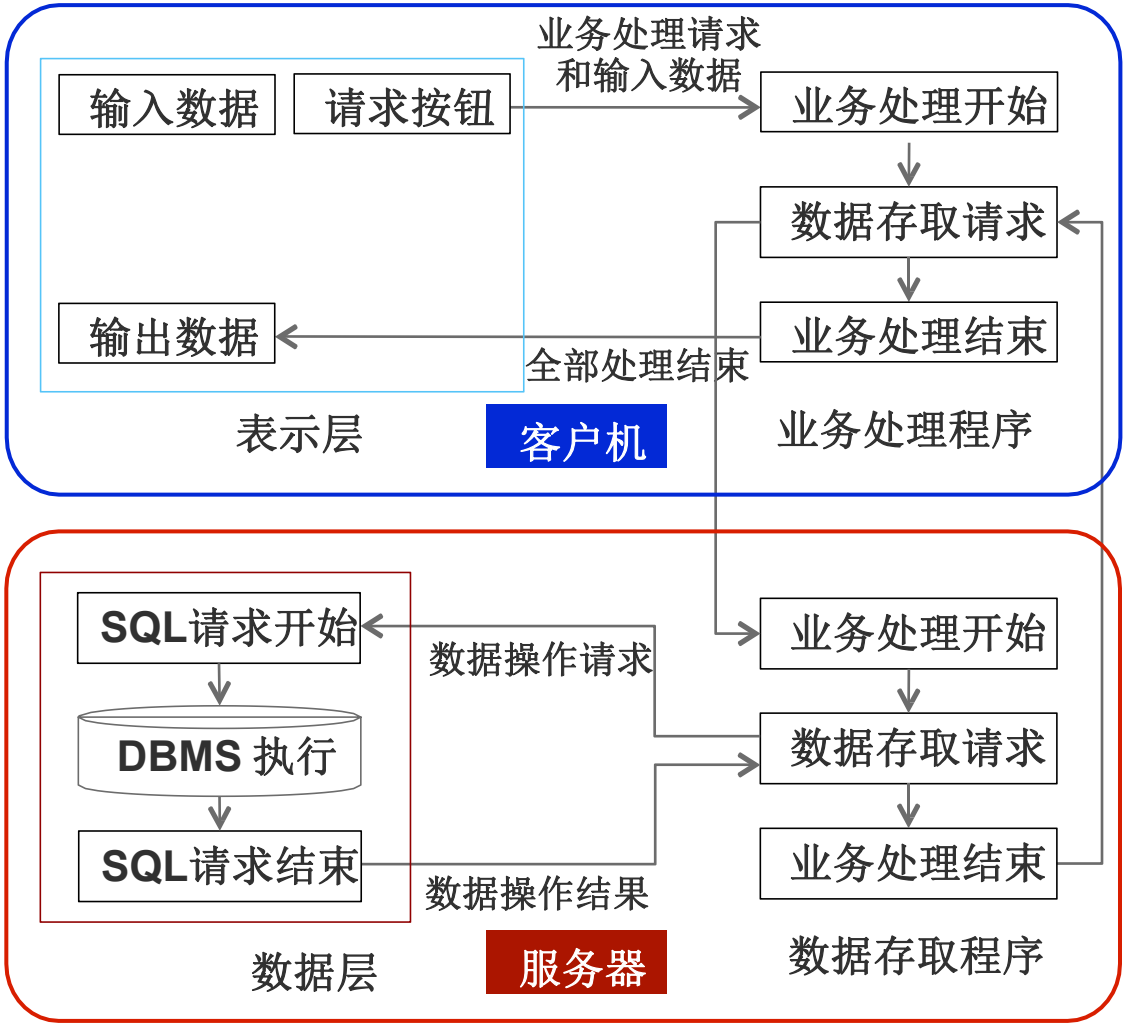


两层C/S结构

胖客户端模型：

- 服务器只负责数据的管理
- 客户机实现应用逻辑和用户的交互

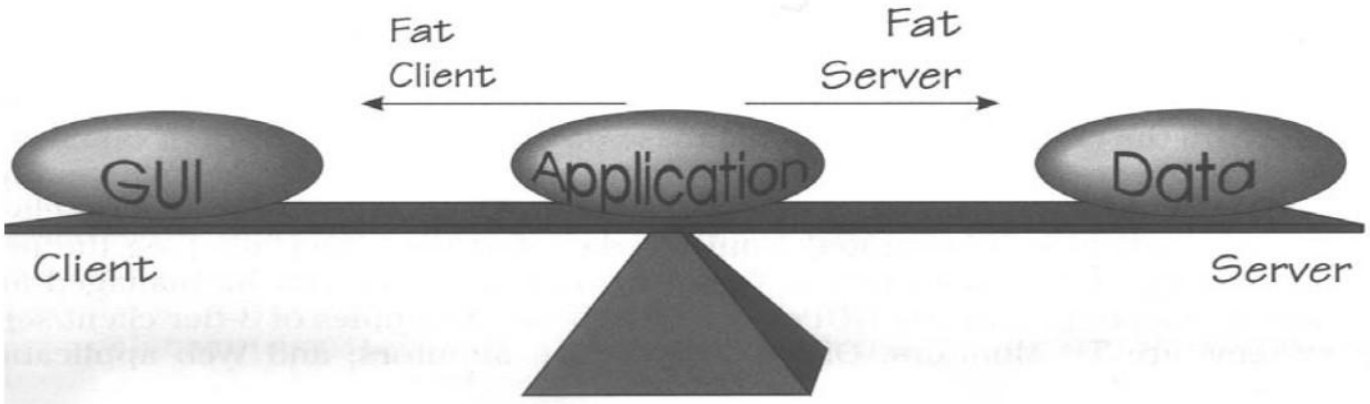




胖客户端与瘦客户端

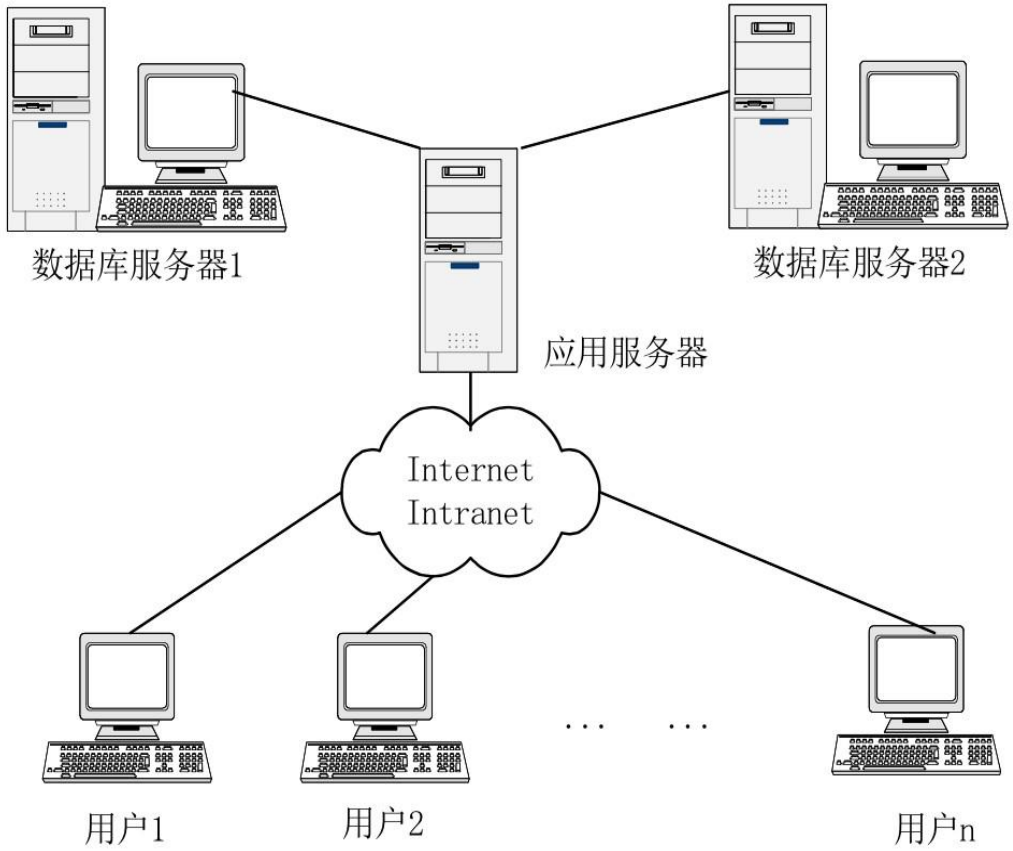
业务逻辑的划分比重：在客户端多一些还是在服务器端多一些?

- 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



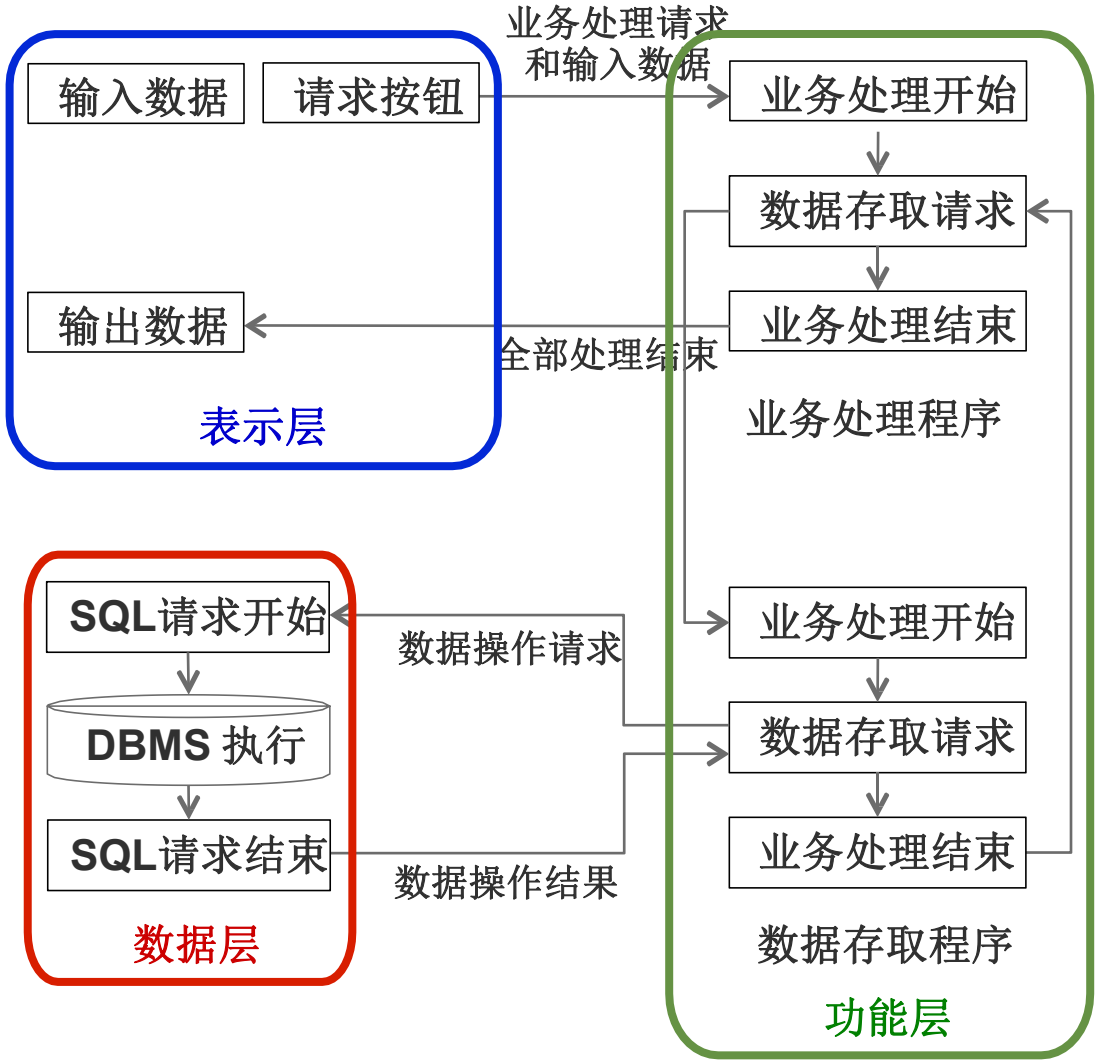
三层C/S结构

- **表示层：**包括所有与客户机交互的边界对象，如窗口、表单、 网页等。
- **功能层（业务逻辑层）：**包括所有的控制和实体对象，实现应用程序的处理逻辑和规则。
- **数据层：**实现对数据库的存储、查询和更新。



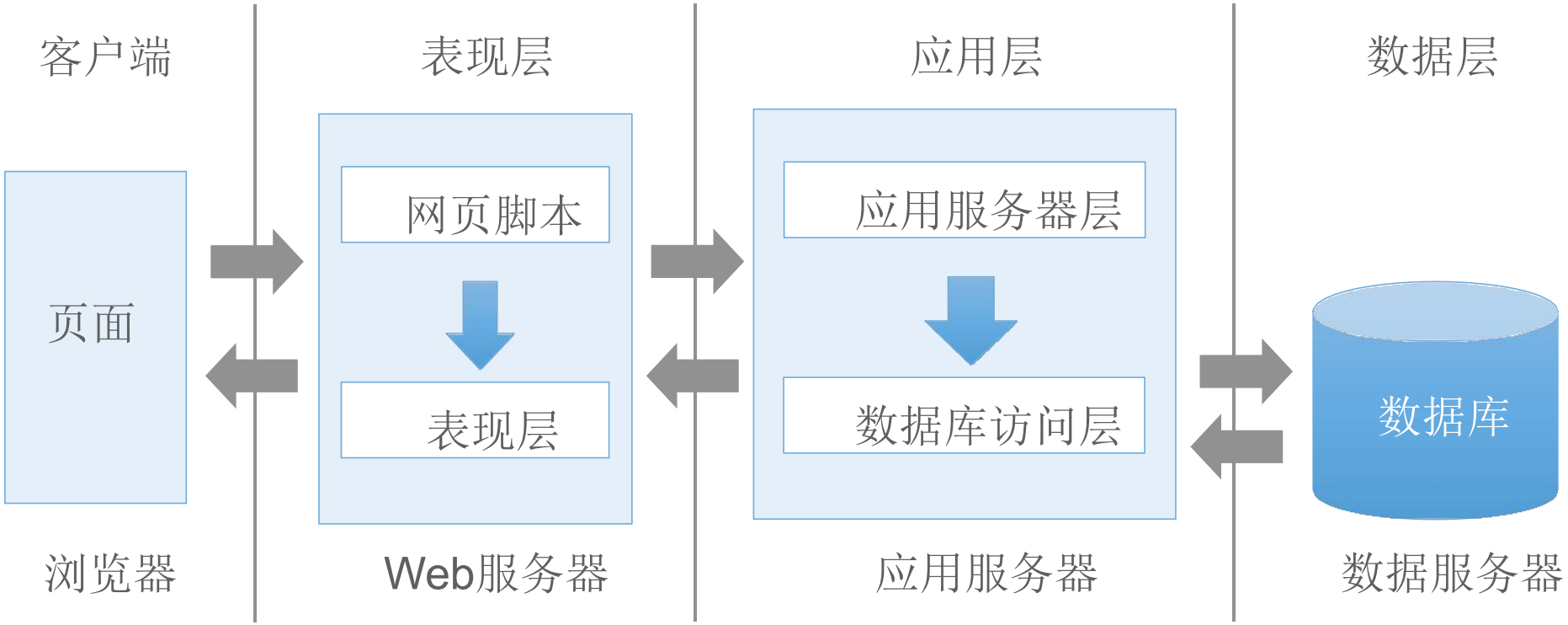
三层C/S结构

- **表示层**：包括所有与客户机交互的边界对象，如窗口、表单、 网页等。
- **功能层（业务逻辑层）**：包括所有的控制和实体对象，实现应用程序的处理逻辑和规则。
- **数据层**：实现对数据库的存储、查询和更新。

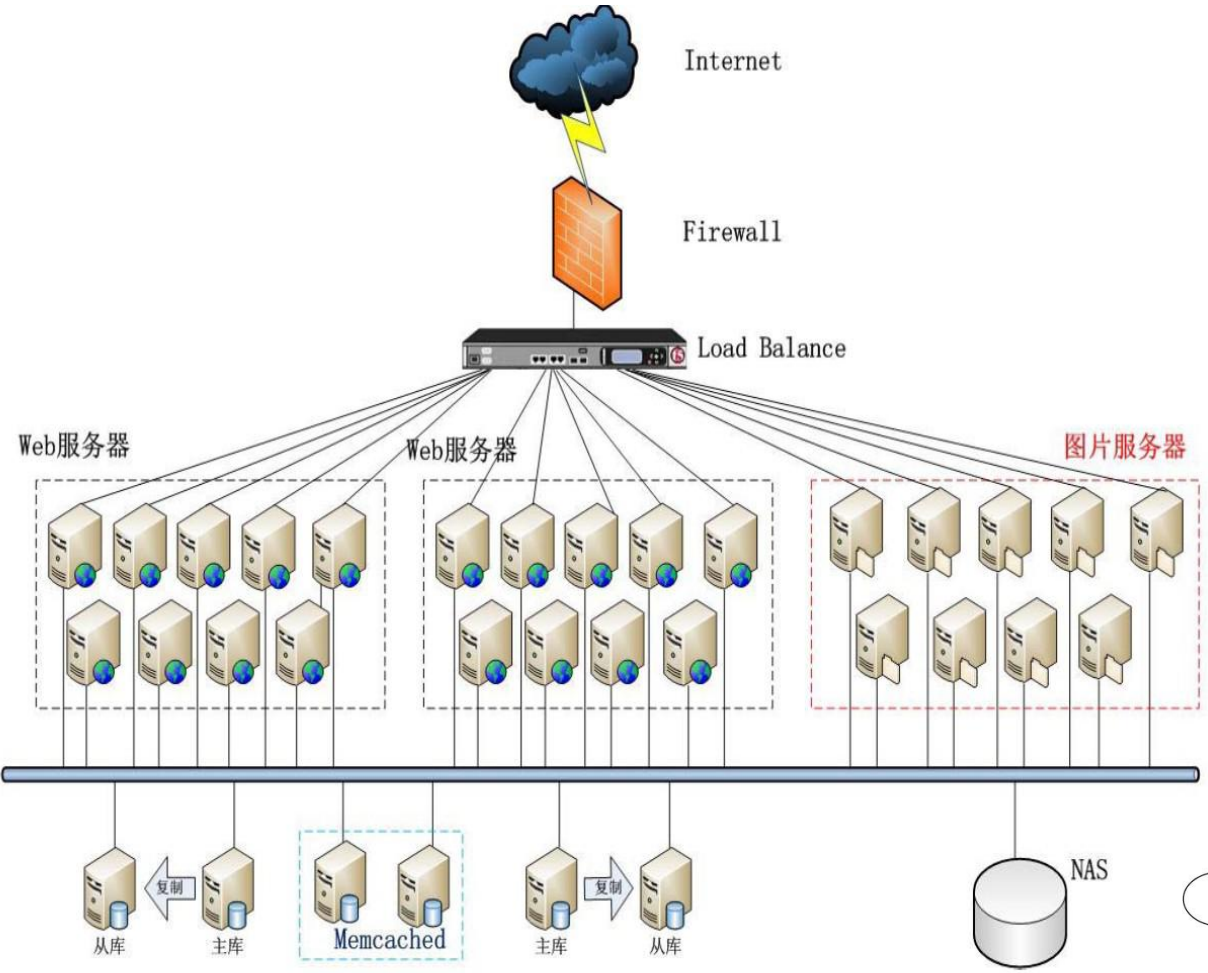


B/S结构

浏览器/服务器（Browser/Server）结构是三层C/S风格的一种实现方式。



集群结构



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）

= = = = 系统

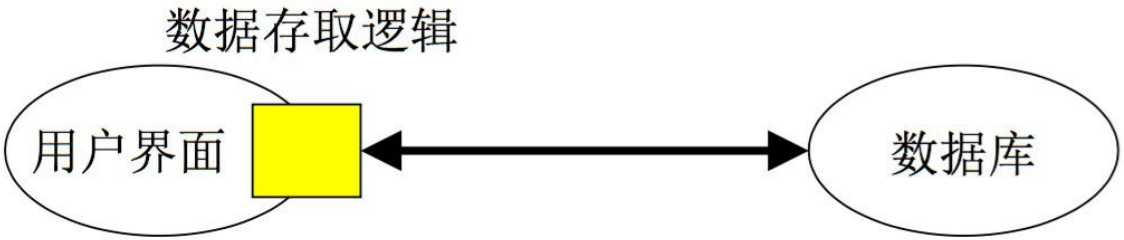
集群内各服务器上的内容之和构成
系统完整的功能/数据
（通过分布式提高速度与并发性）

+ + + = 系统

MVC结构

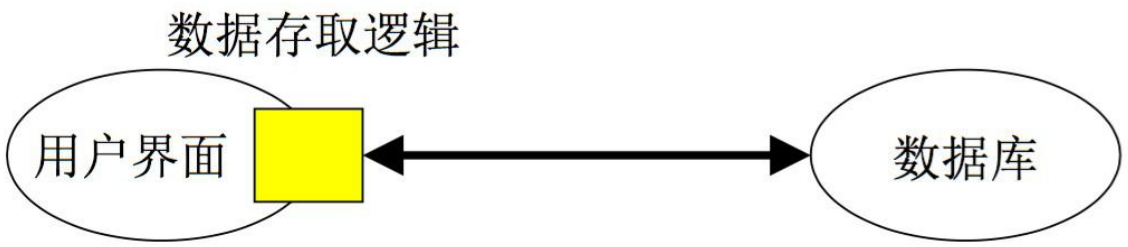
客户机—服务器结构：

- 许多应用系统的用途都是从数据库中检索数据，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。



MVC结构

思考：纯粹的B/S结构会不会解决这个问题？



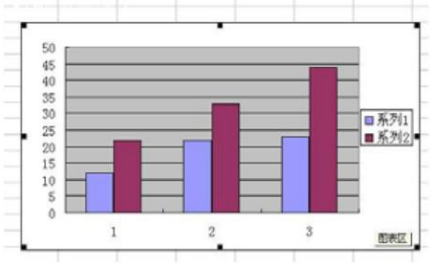
如何实现模块化，以便可以轻松地单独修改各个部分而不影响其他部分?

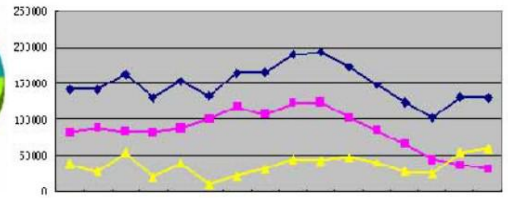
- 用户界面仍然需要显式地调用功能层的业务逻辑
- 仍然难以避免“用户界面的修改→业务逻辑的修改”的问题

MVC结构

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。
- 如果将UI代码和业务逻辑组合一起并放在UI中，则每次更改界面都可能引起对业务逻辑的修改。
- 在某些情况下，应用程序以不同的方式显示同一数据。

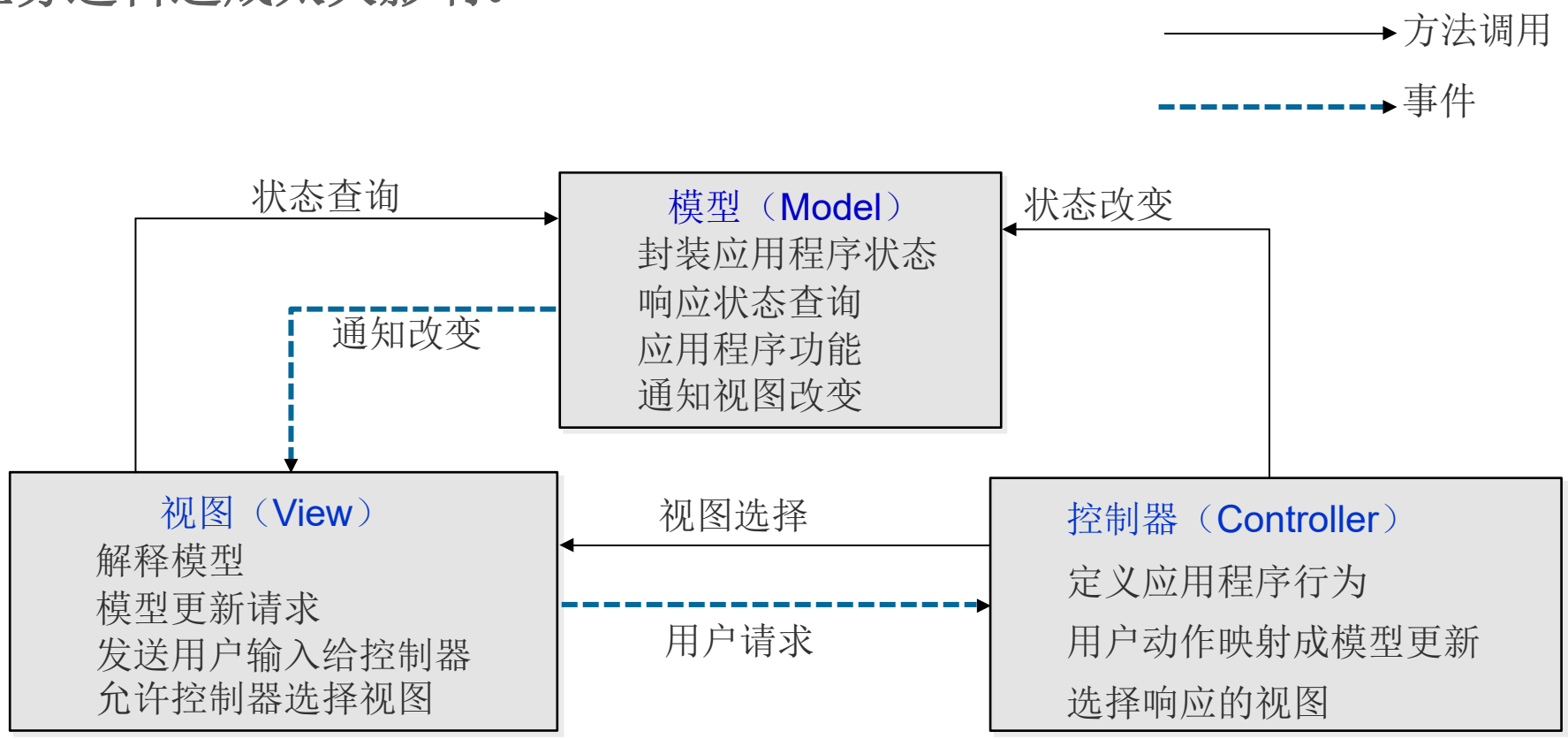




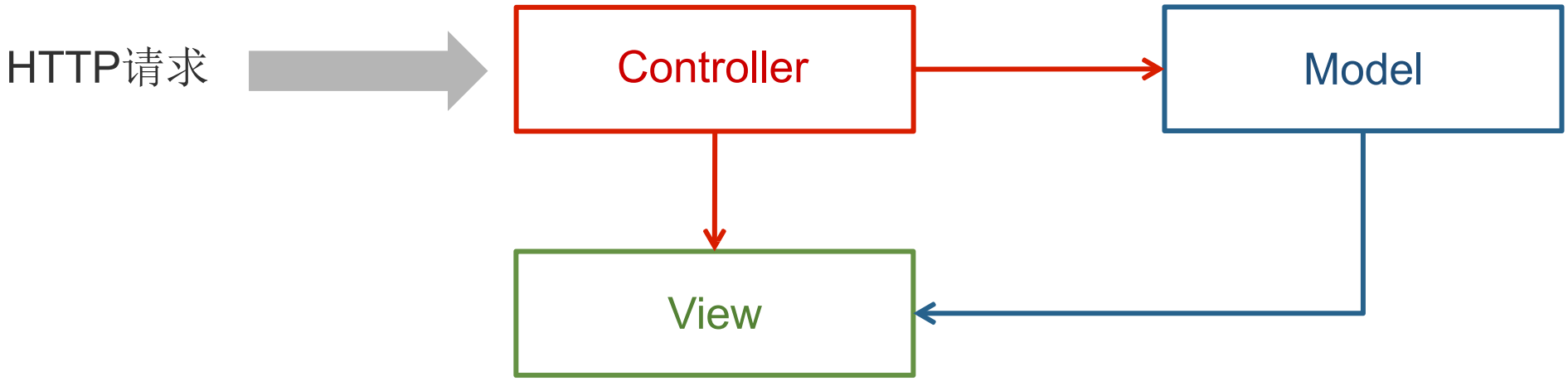
- 与业务逻辑相比，用户界面代码对设备的依赖性往往更大。
- 设计美观而有效的用户界面与开发复杂业务逻辑需要不同的编程技能。
- 通常，为用户界面创建自动测试比为业务逻辑更难、更耗时。

MVC结构

模型-视图-控制器（MVC）结构将应用程序的数据模型、业务逻辑和用户界面分别放在独立构件中，这样对用户界面的修改不会对数据模型/业务逻辑造成太大影响。

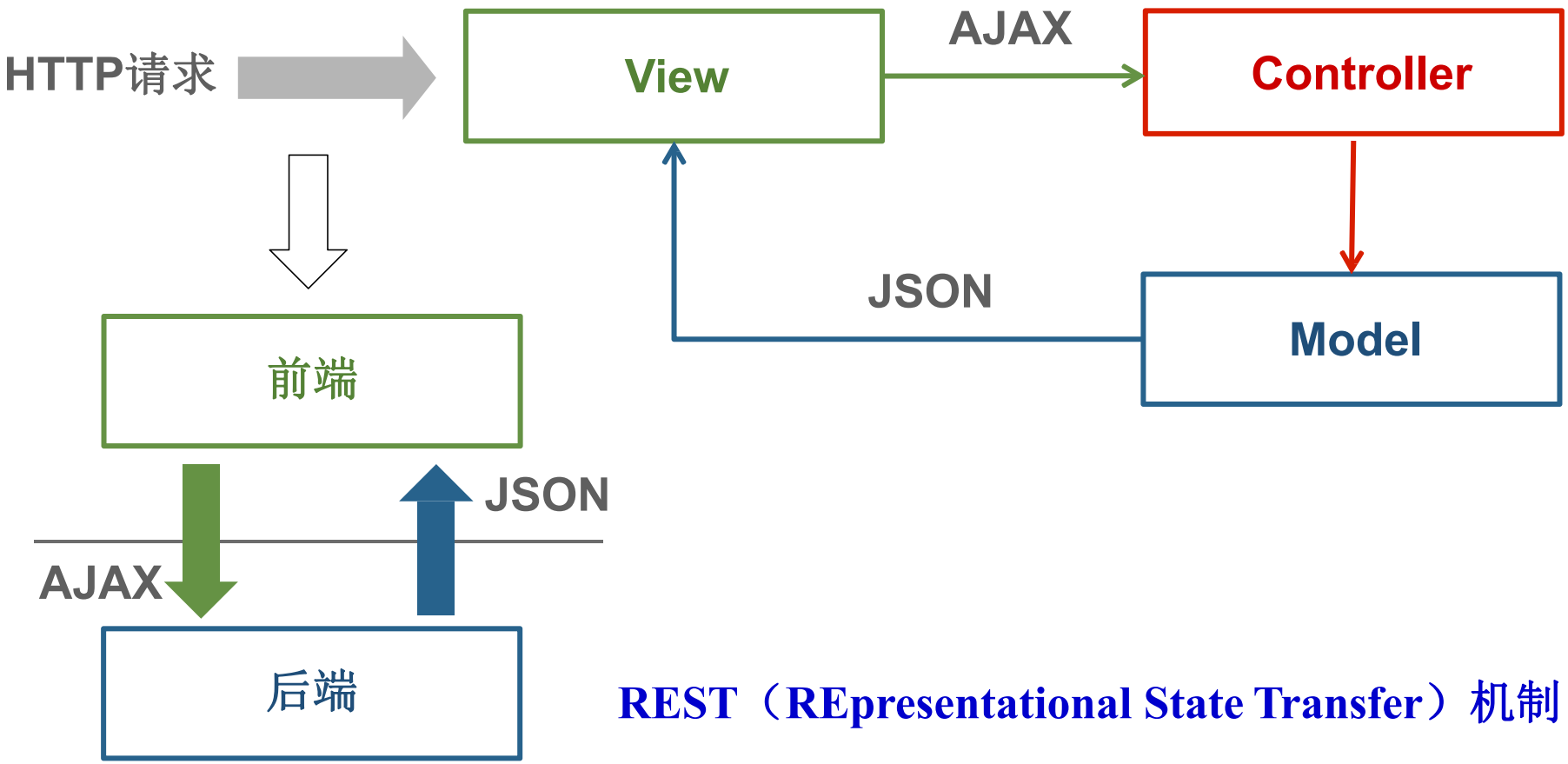


MVC结构



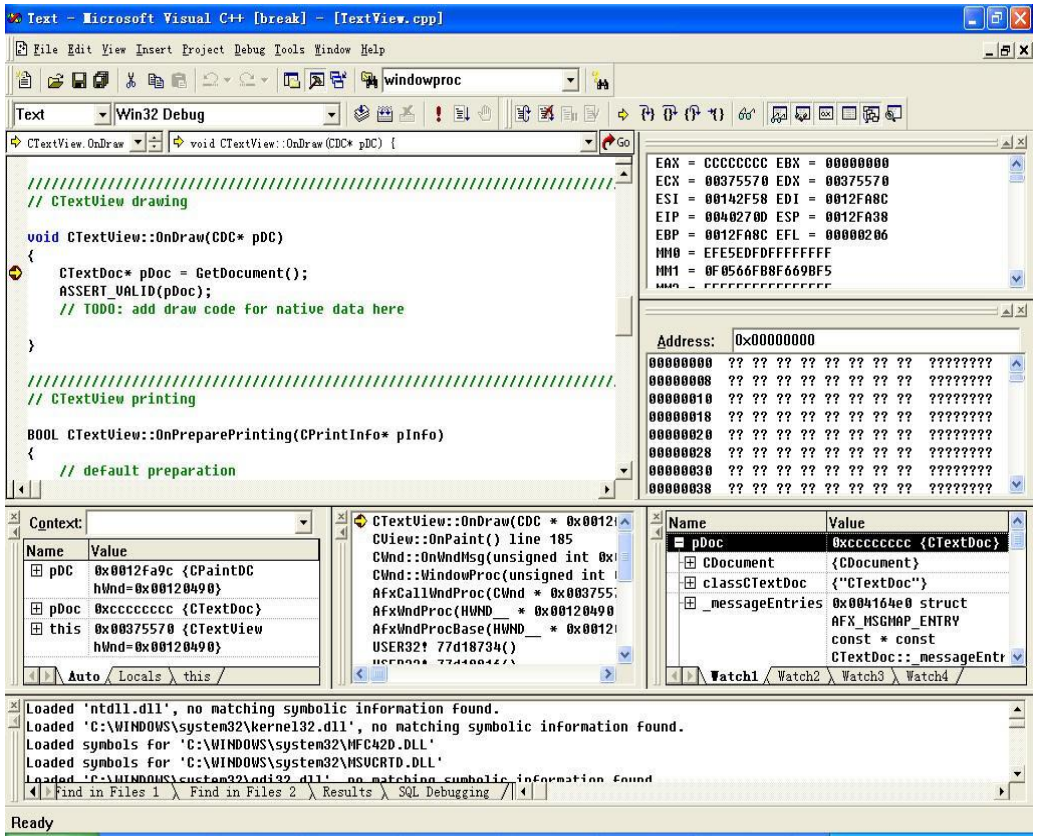
- 每次请求必须经过“控制器->模型->视图”过程，才能看到最终展现的界面
- 视图是依赖于模型的
- 渲染视图在服务端完成，呈现给浏览器的是带有模型的视图页面，性能难优化

MVC结构



REST（REpresentational State Transfer）机制

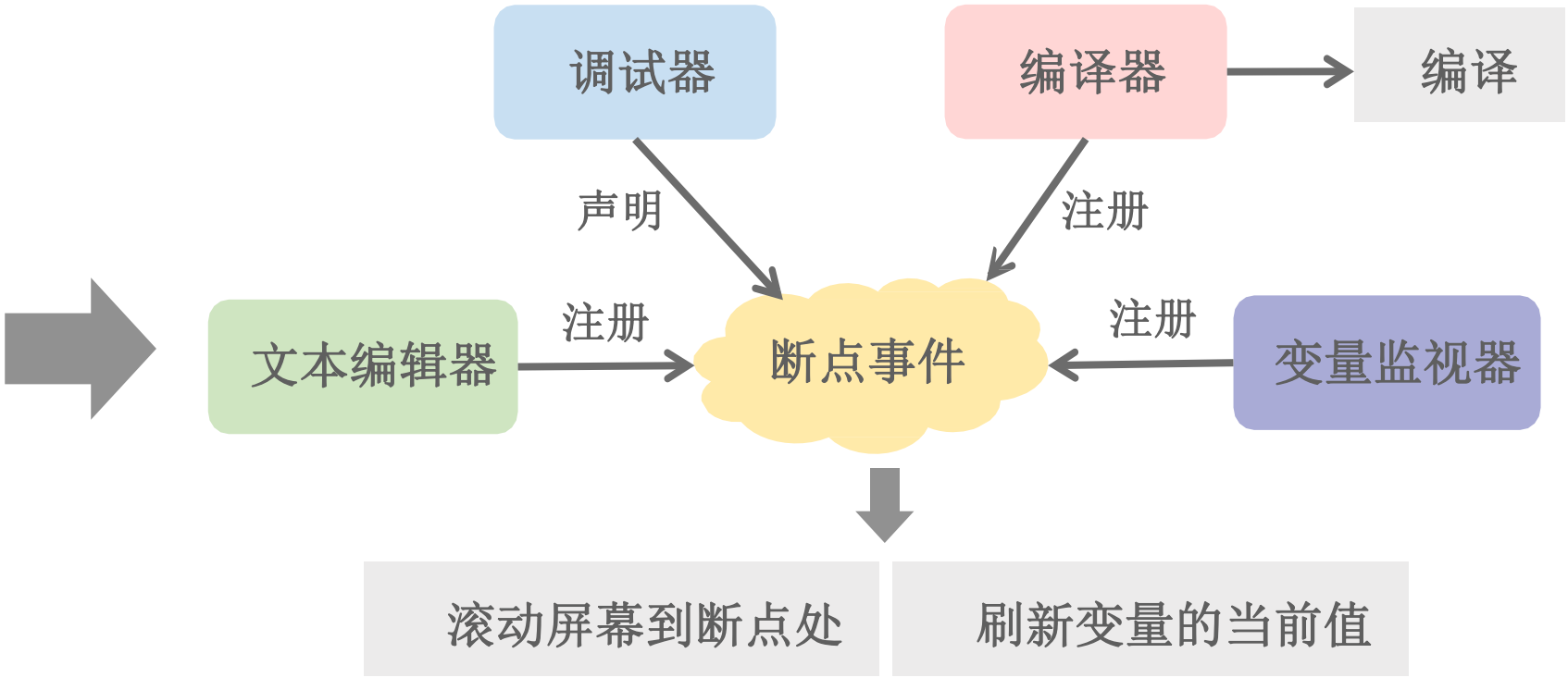
事件风格



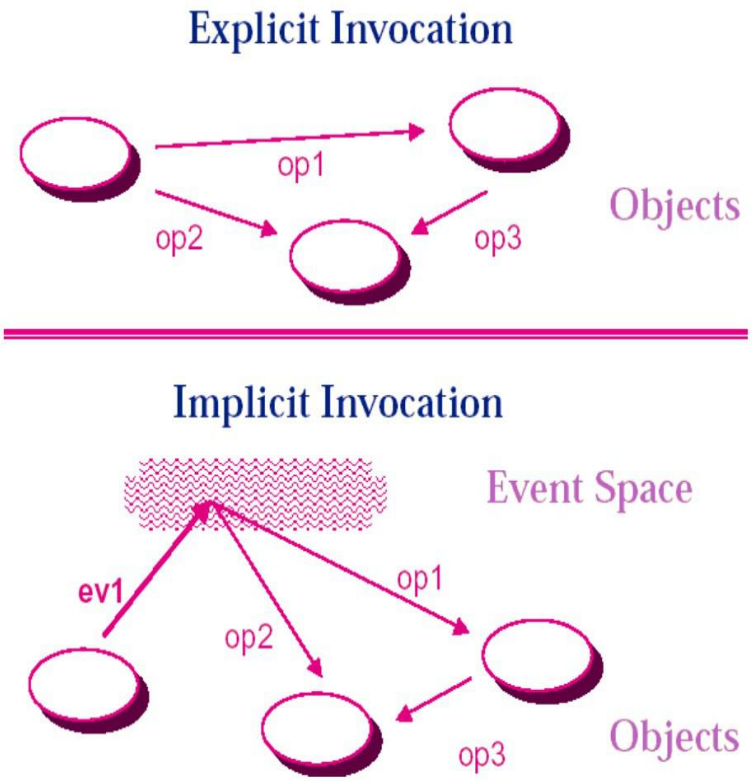
程序调试器的体系结构

事件风格

调试器的工作流程



事件风格



显式调用：

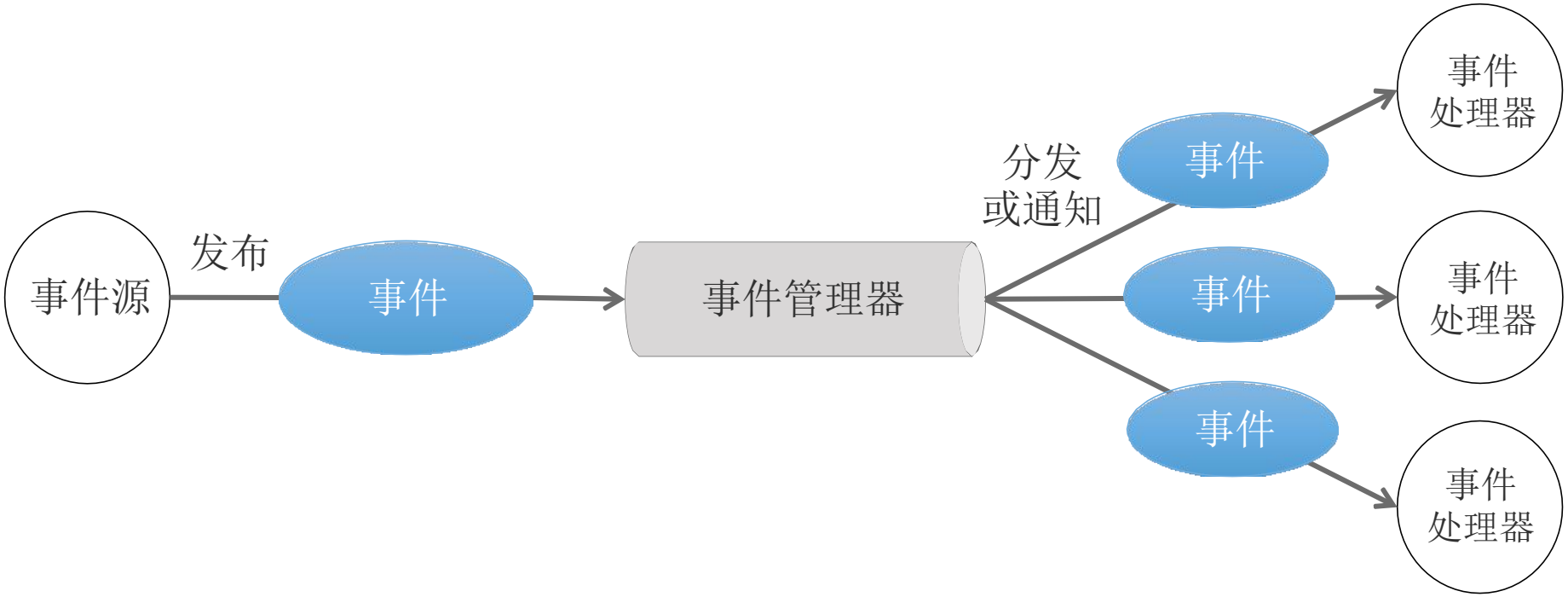
- 各个构件之间的互动是由显性调用函数或程序完成的
- 调用过程与次序是固定的、预先设定的

隐式调用：

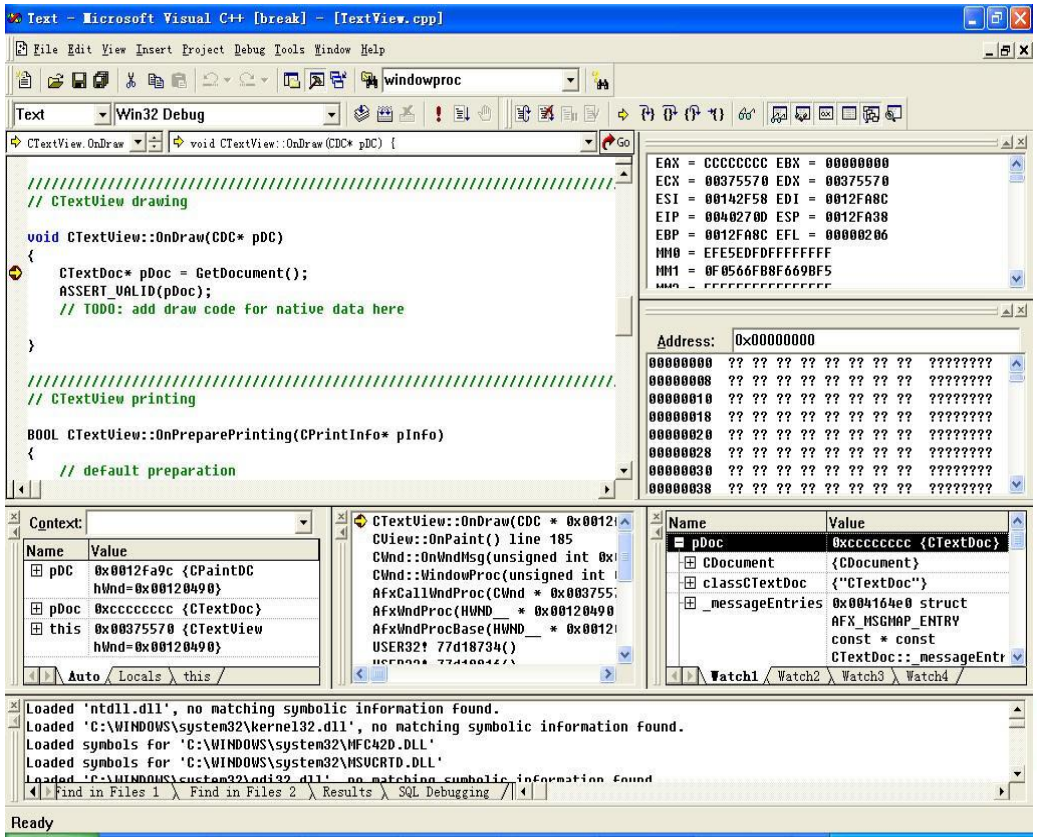
- 调用过程与次序不是固定的、预先未知
- 各构件之间通过事件的方式进行交互

事件风格

事件系统是将应用看成是一个构件集合，每个构件直至发生对它有影响的事件时才有所动作。



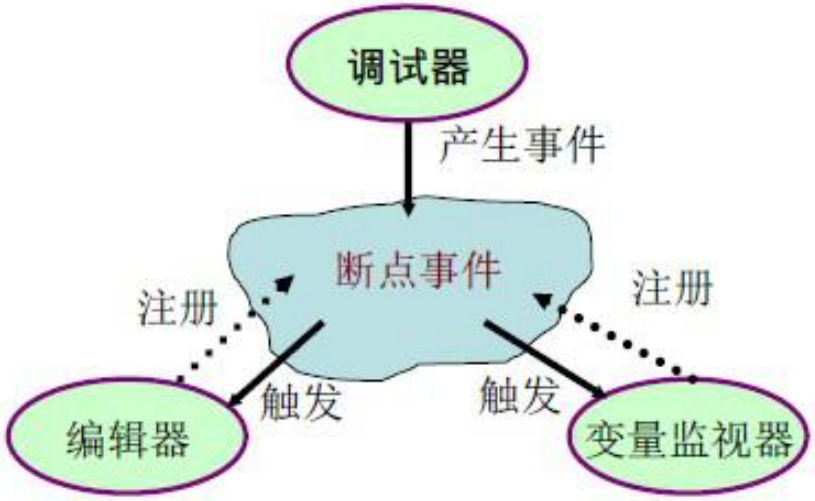
事件风格



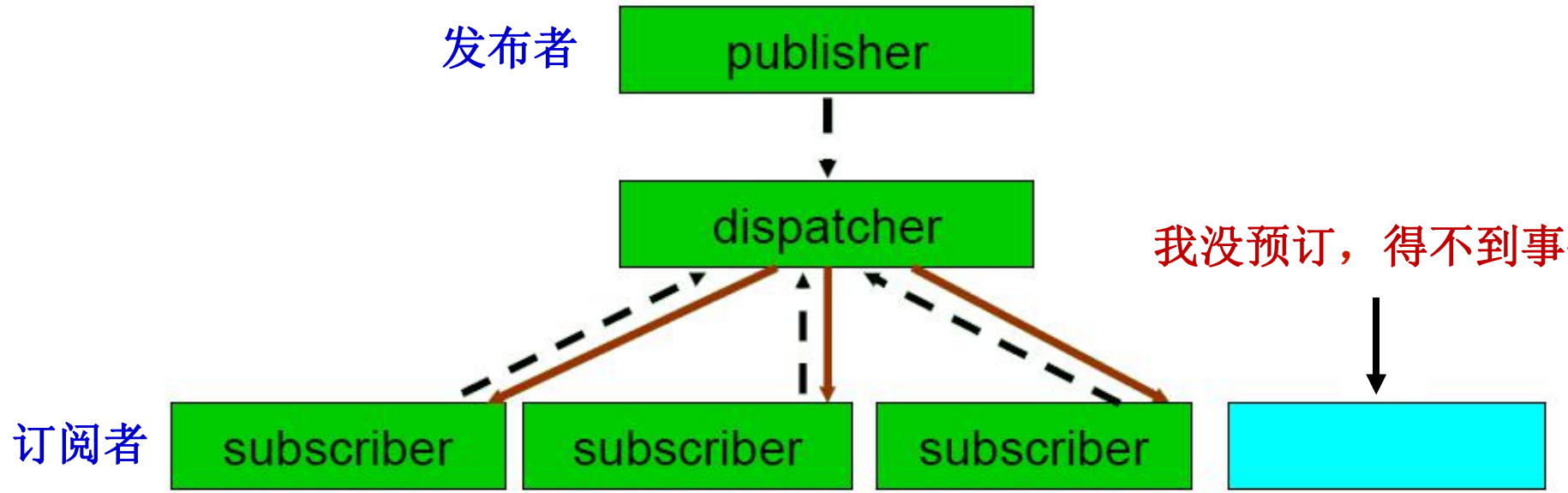
事件源：调试器

事件处理器：编辑器与变量监视器

事件管理器：IDE（集成开发环境）

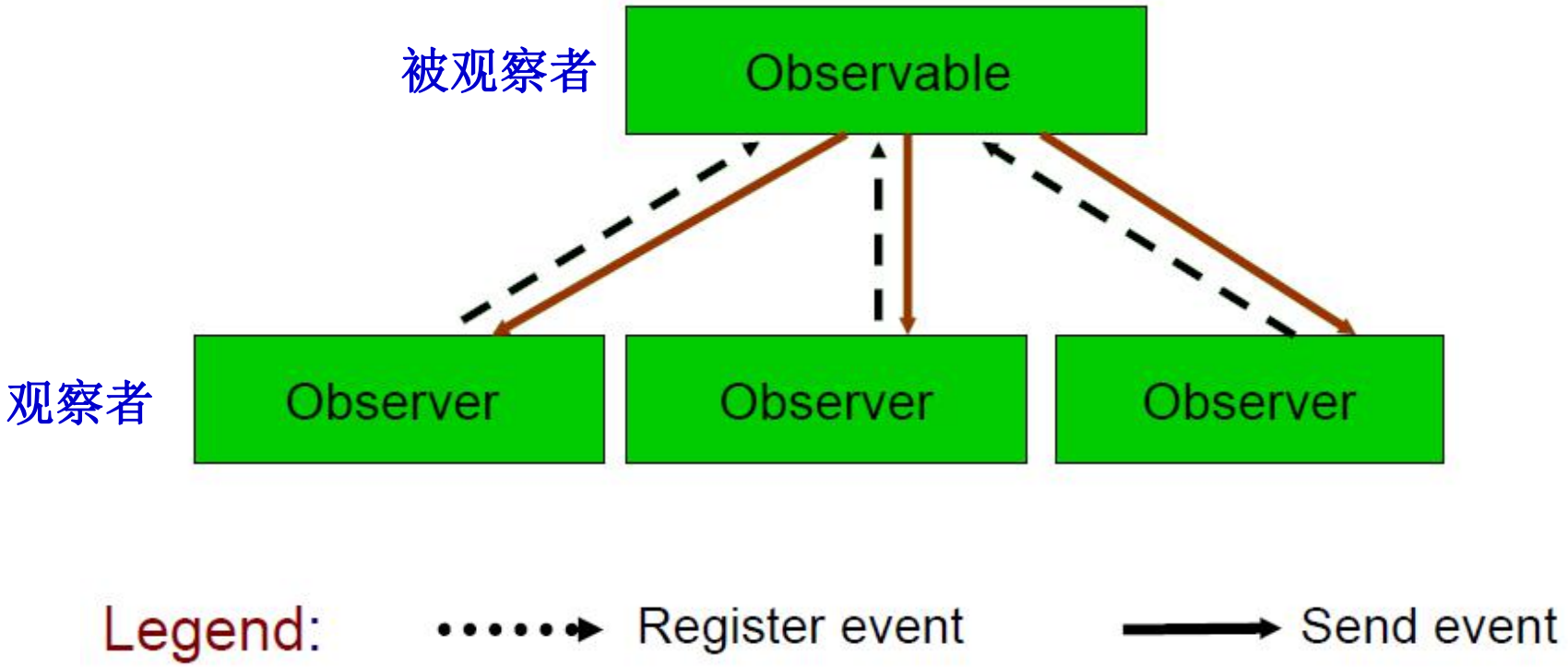


事件风格的实现策略之一：选择广播式



说明：这种方式是有目的广播，只发送给那些已经注册过的订阅者。

事件风格的实现策略之二：观察者模式



软件体系结构风格的选择

- 简单地判断某一个具体的应用应该采取何种体系结构是非常困难的，需要借助于丰富的经验。

- 绝大多数实际运行的系统都是几种体系结构的复合：

  - 在系统的某些部分采用一种体系结构而在其他部分采用另外的体系，故而需要将复合几种基本的体系结构组合起来形成复合体系结构。

  - 在实际的系统分析和设计中，首先将整个系统作为一个功能体进行分析和权衡，得到适宜 的和最上层的体系结构；如果该体系结构中的元素较为复杂，可以继续进行分解，得到某一部分的局部体系结构。

- 将焦点集中在系统总体结构的考虑上，避免较多地考虑使用的语言、具体的技术等实现细节上。

软件体系结构风格的选择

技术因素

- 使用何种构件、连接件
- 在运行时，构件之间的控制机制是如何被共享、分配和转移
- 数据如何通讯
- 数据与控制如何交互

质量因素

- 可修改性：算法的变化；数据表示方式的变化；系统功能的可扩展性
- 性能：时空复杂性
- 可复用性

基于经验的选择原则

- 层次化的思想在任何系统中都可能得到应用
- 如果问题可分解为连续的几个阶段，那么考虑使用顺序批处理风格或管道--- 过滤器风格
- 如果核心问题是应用程序中数据的理解、管理与表示，那么考虑使用仓库或者抽象数据类型（ADT）/OO风格
- 如果数据格式的表示可能发生变化，ADT/OO可限制这种变化所影响的范围
- 如果数据是持久存在的，则使用仓库结构

基于经验的选择原则

- 如果任务之间的控制流可预先设定、无须配置，那么考虑使用主程序---子过程风格、OO风格
- 如果任务需要高度的灵活性与可配置性、松散耦合性或者任务是被动性的，那么考虑使用事件系统或C/S风格
- 如果设计了某种计算，但没有机器可以支持它运行，那么考虑使用虚拟机/ 解释器体系结构
- 如果要实现一些经常发生变化的业务逻辑，考虑使用基于规则的系统

基于经验的选择原则

软件体系结构的选择和应用

需要经验，也需要创新