哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程
第四章 软件架构设计
4-3 基本架构风格

王忠杰
rainy@hit.edu.cn

2017年11月7日

主要内容

- 什么是“架构风格”
- 调用-返回风格
- 以数据为中心的风格
- 分层结构
- C/S和B/S
- 事件风格

1 软件架构的风格

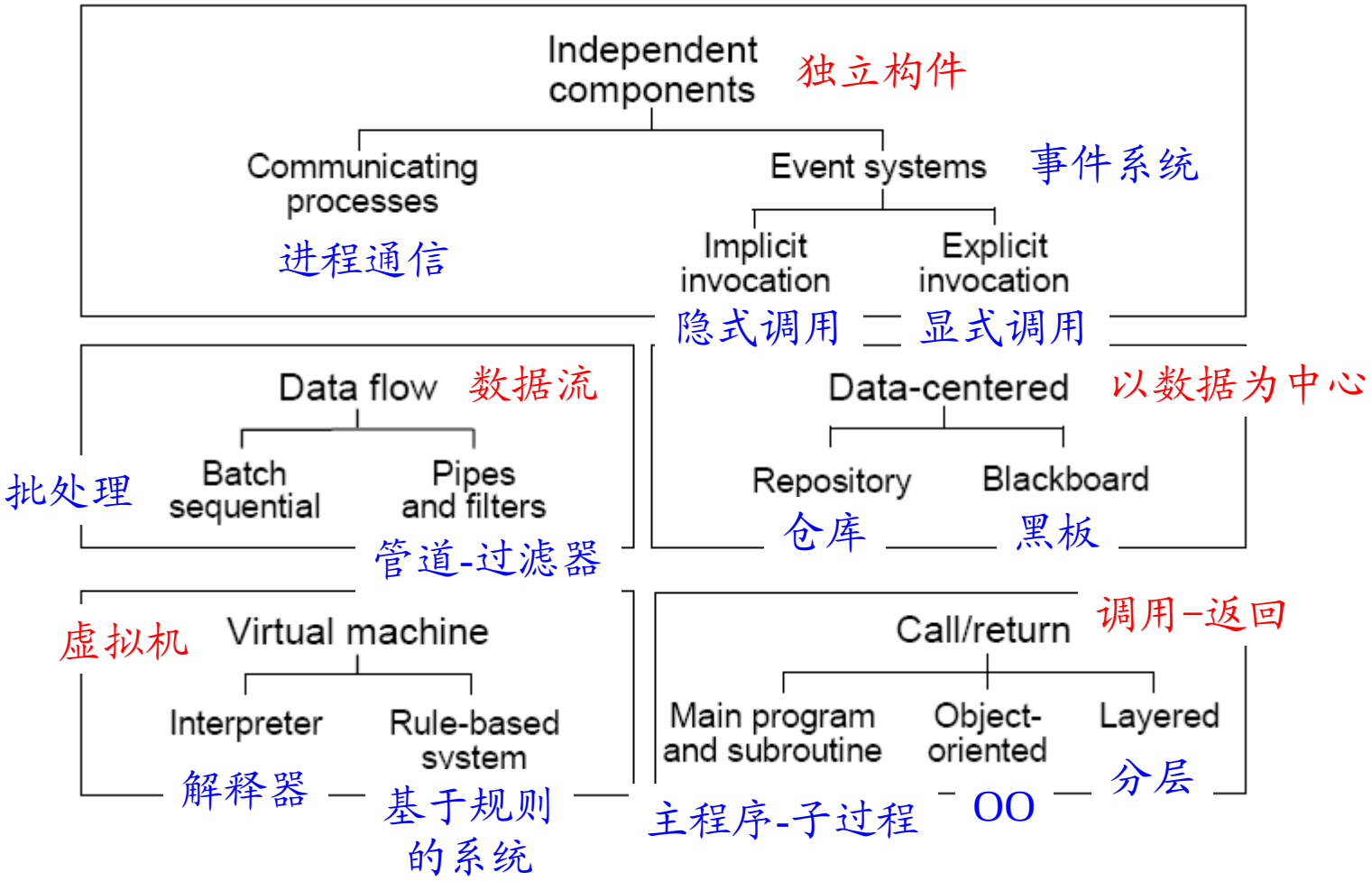
从“建筑风格”开始

- 建筑风格等同于建筑架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。
- 之所以称为“风格”，是因为经过长时间的实践，它们已经被证明具有良好的工艺可行性、性能与实用性，并可直接用来遵循与模仿 (复用)。
- 软件系统同建筑一样，也具有若干特定的“风格” (software architectural style)；
  - 这些风格在实践中被多次设计、应用，已被证明具有良好的性能、可行性和广泛的应用场景，可以被重复使用；
- 定义：
  - 描述特定领域中软件系统家族的组织方式的惯用模式，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。

“软件架构风格”的组成

- A set of component types (e.g., data repository, process, object) (一组构件类型)
- A set of connector types/interaction mechanisms (e.g., subroutine call, event, pipe) (一组连接件类型/交互机制)
- A topological layout of these components (这些构件的拓扑分布)
- A set of constraints on topology and behavior (e.g., a data repository is not allowed to change stored values, pipelines are acyclic) (一组对拓扑和行为的约束)
- An informal description of the costs and benefits of the style, e.g.: “Use the pipe and filter style when reuse is desired and performance is not a top priority” (一些对风格的成本和收益的描述)

经典架构风格


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

2 调用-返回风格

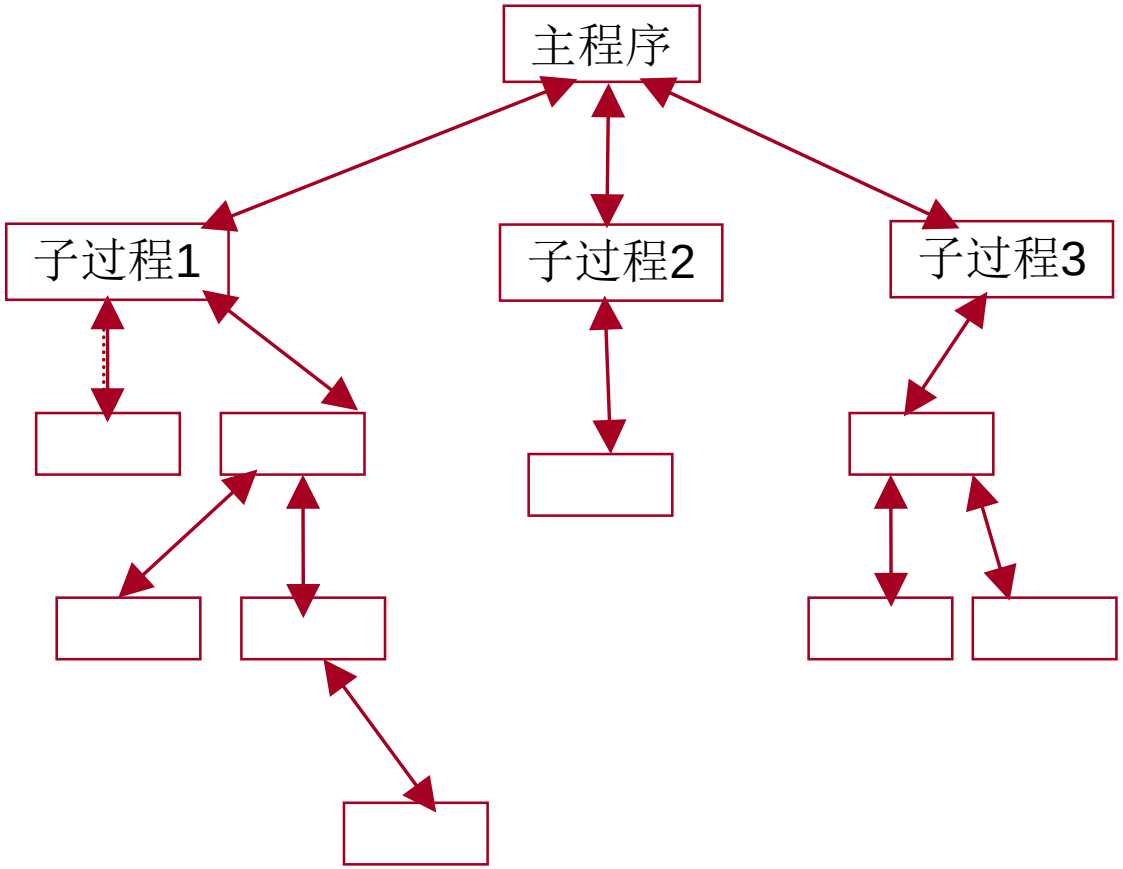
以函数/对象作为基本构造单元，彼此通过call-return相互连接

特征：不考虑分层，主要遵循同步调用方式

主程序-子过程

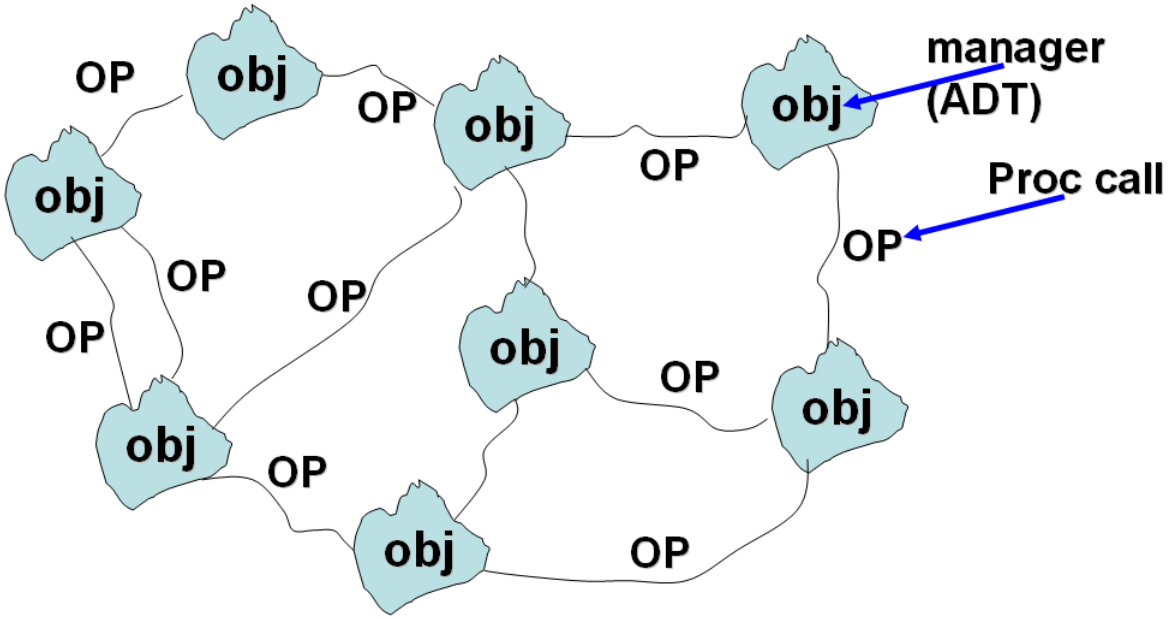
- **该风格是结构化程序设计的一种典型风格，从功能的观点设计系统，通过逐步分解和逐步细化，得到系统架构。**
  - 构件：主程序、子程序
  - 连接器：调用-返回机制
  - 拓扑结构：层次化结构

- **本质：将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能。**



面向对象风格

- 系统被看作对象的集合，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(ADT)；
- 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
  - 连接件：对象之间通过函数调用、消息传递实现交互

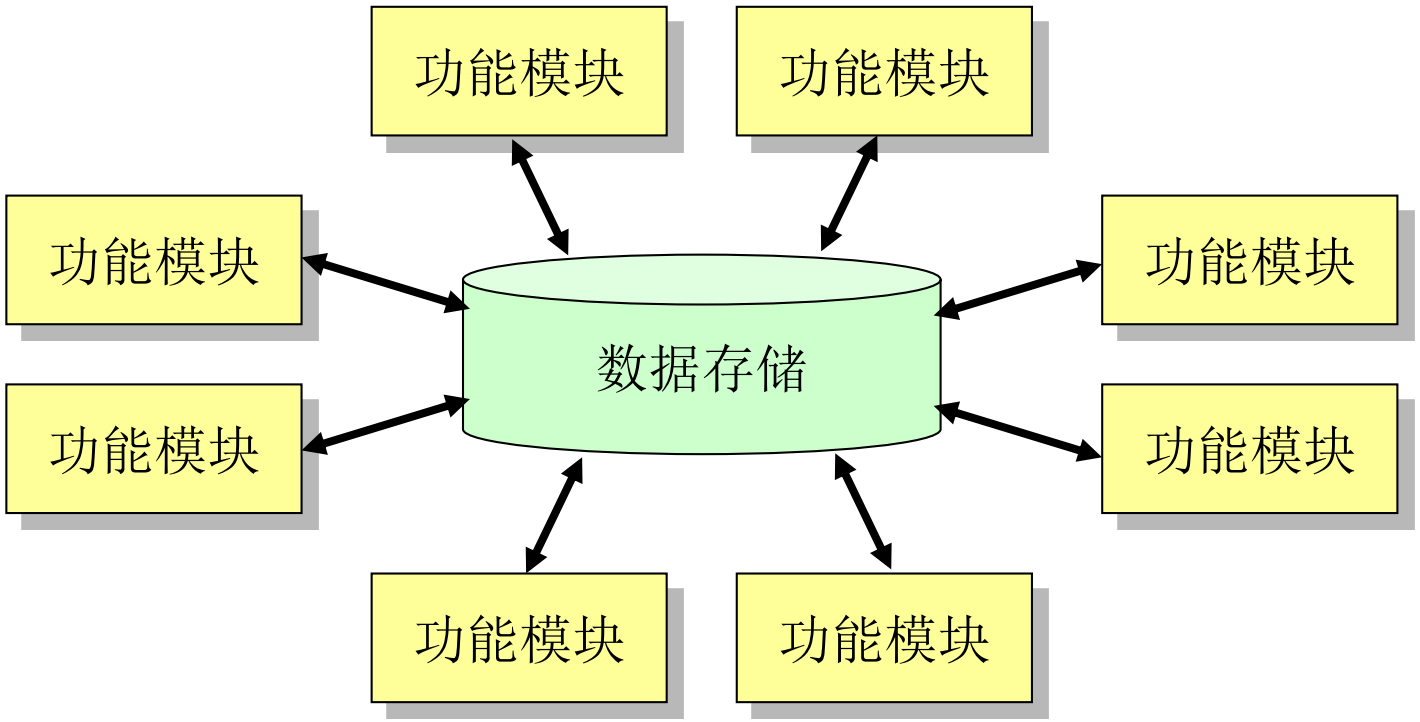


3 以数据为中心的风格

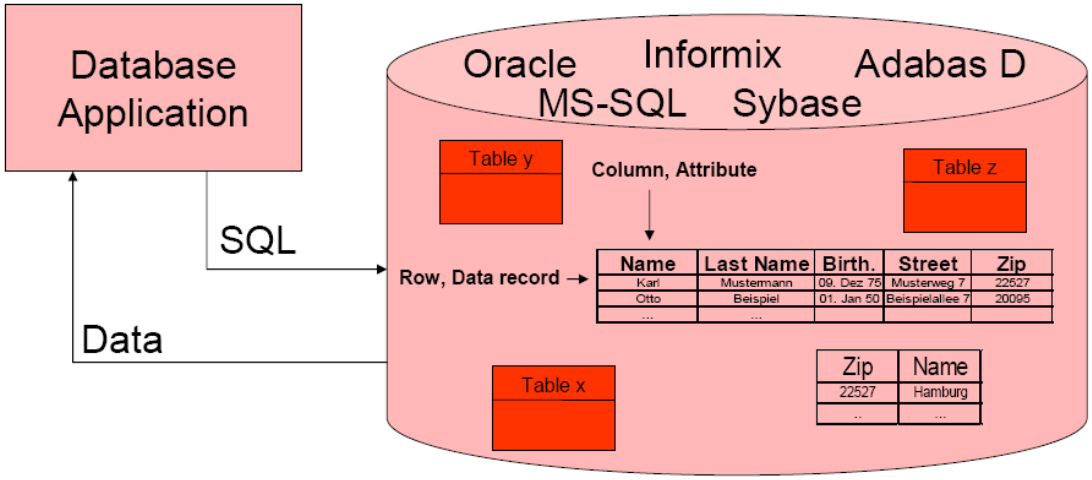
数据的持久化存储被分离出去，形成两层结构

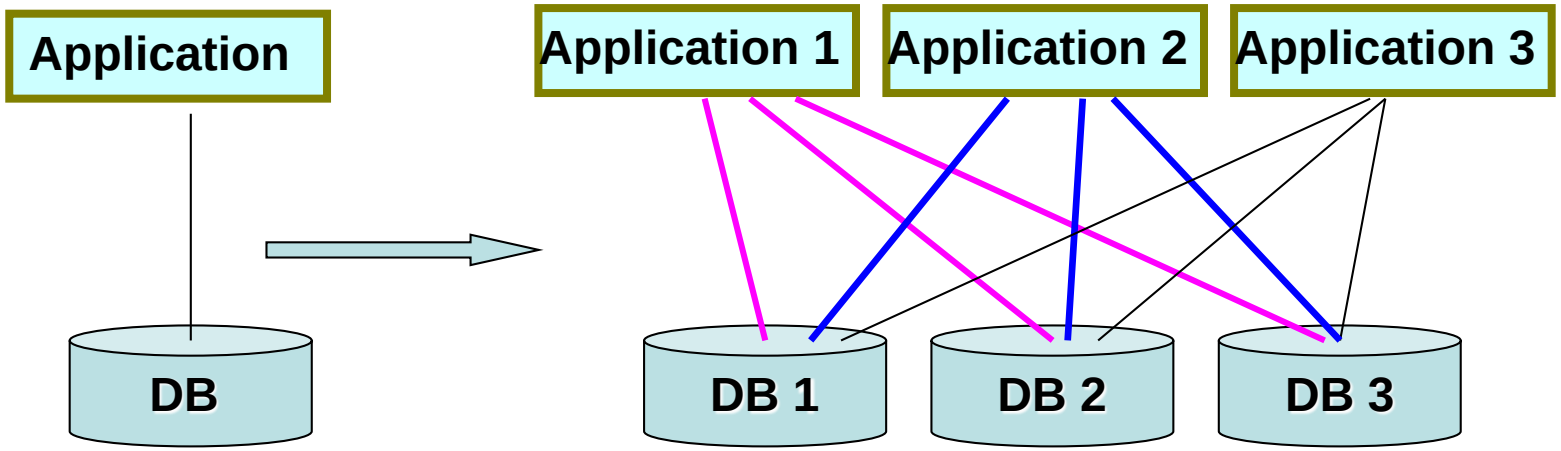
以数据为中心的架构风格

- 以数据为中心的架构风格(也称仓库风格)

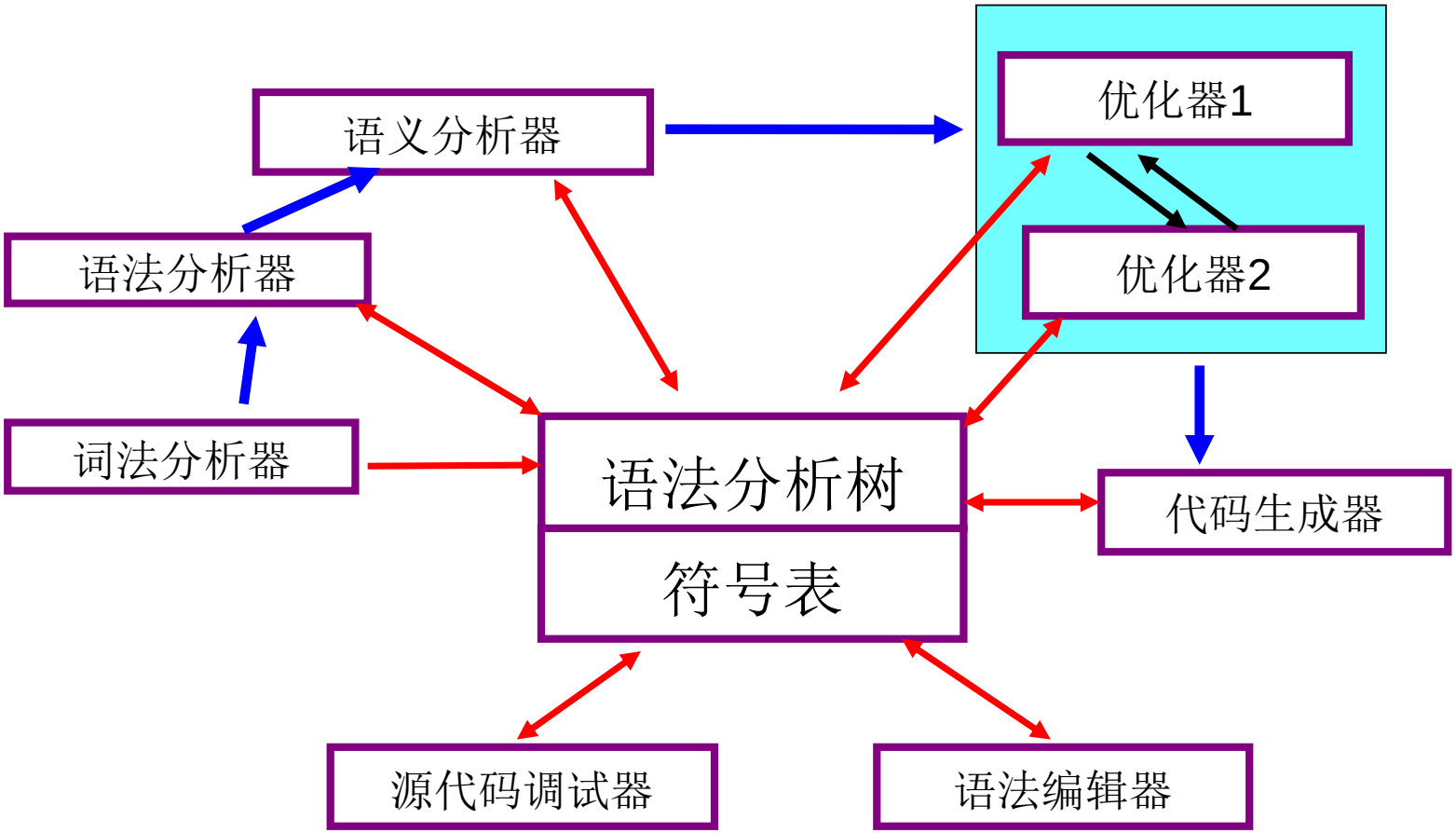


示例1：基于数据库的系统结构





示例2：仓库形式的编译器结构

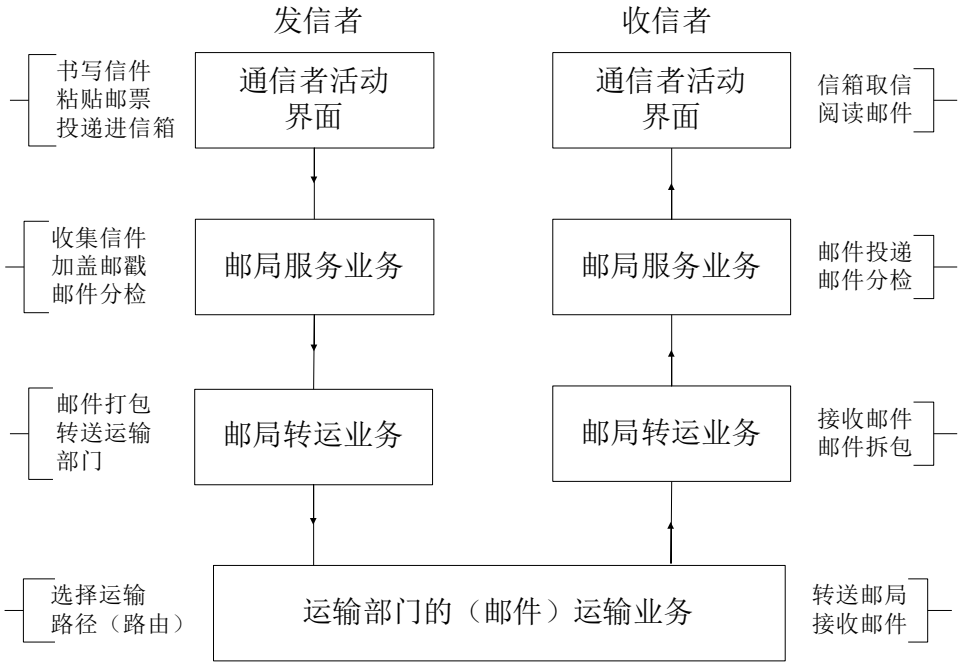


4 层次风格

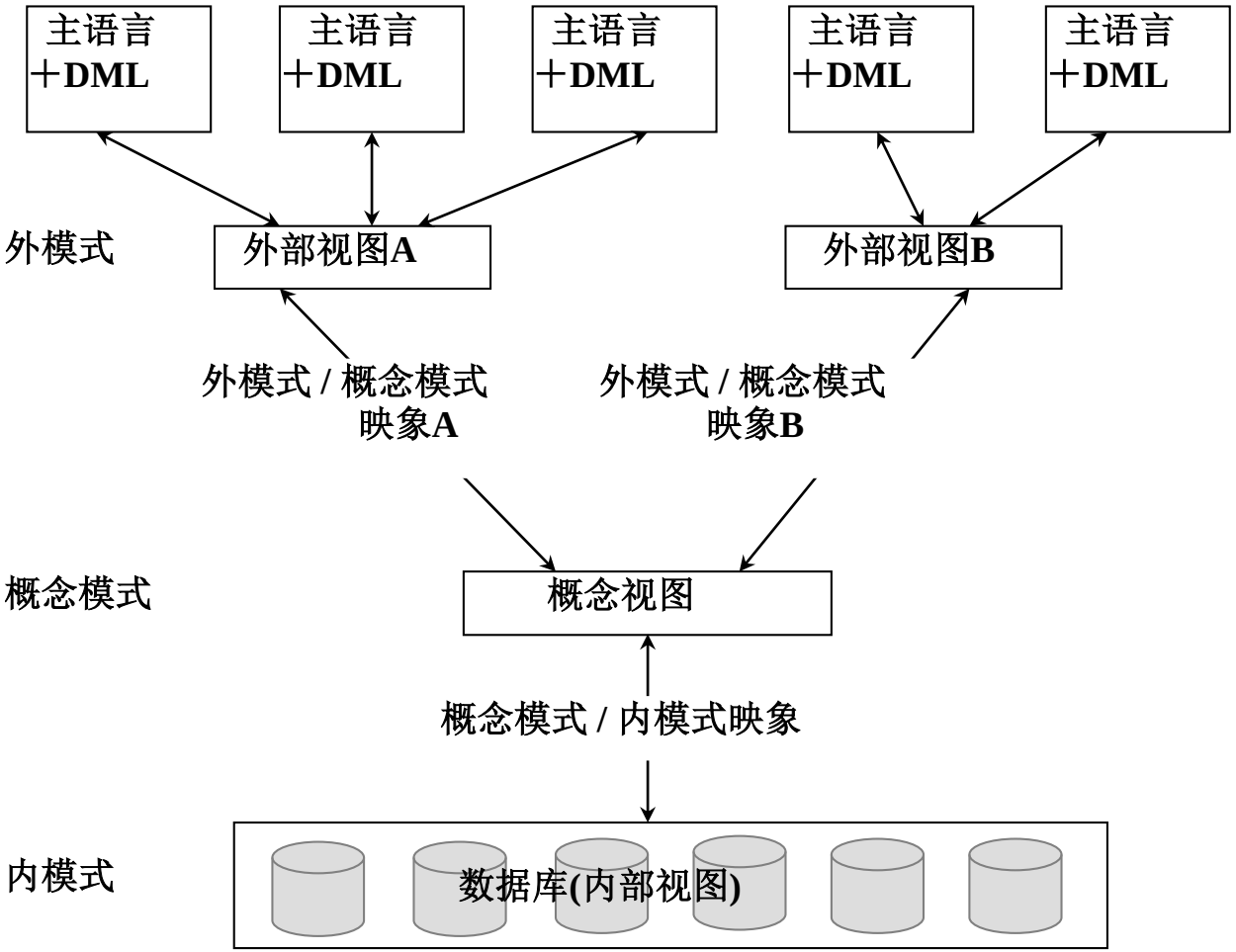
除了数据被分离出去，软件系统的其他各部分进一步分离，形成更复杂的层次结构

层次结构

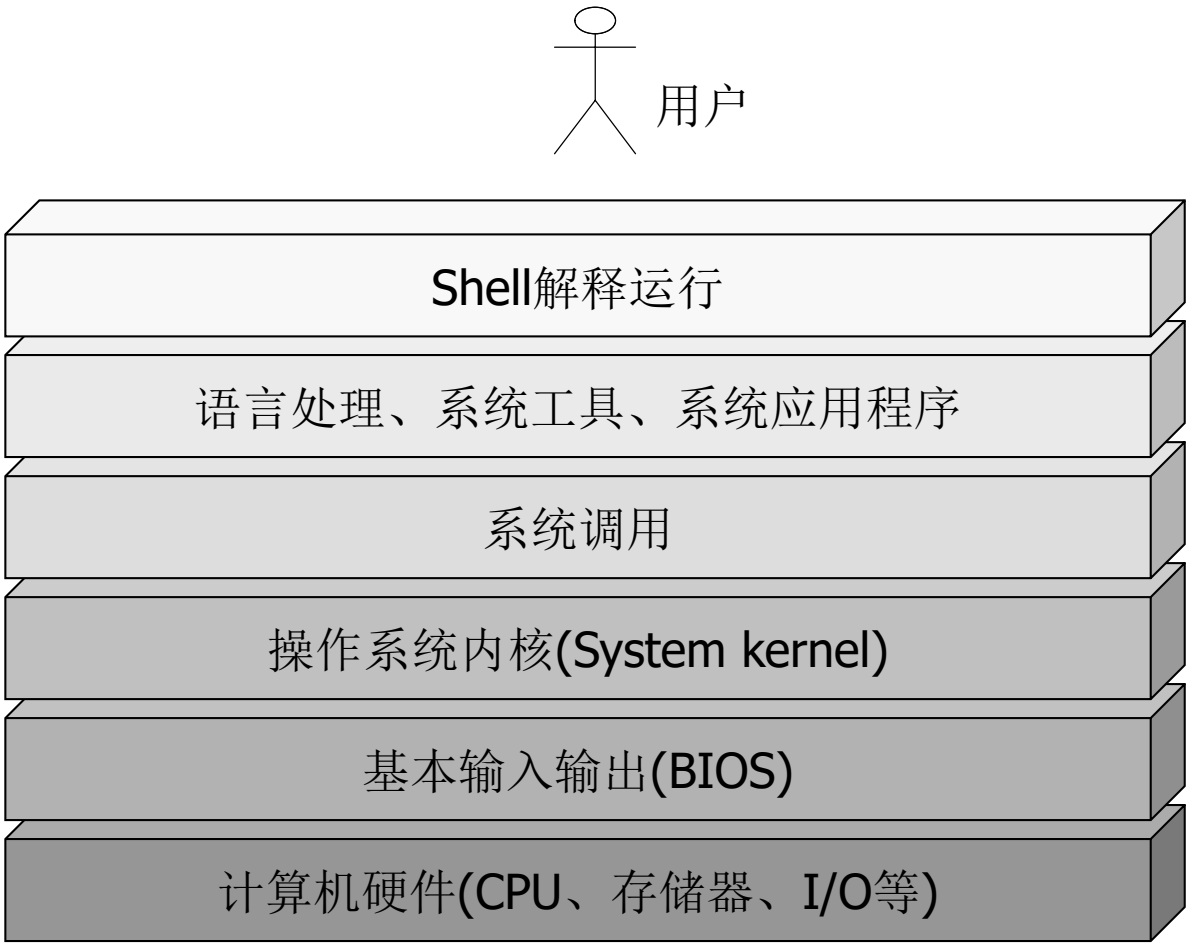
- **层次化早已经成为一种复杂系统设计的普遍性原则；**
- **两个方面的原因：**
  - 事物天生就是从简单的、基础的层次开始发生的；
  - 众多复杂软件设计的实践，大到操作系统，中到网络系统，小到一般应用，几乎都是以层次化结构建立起来的。



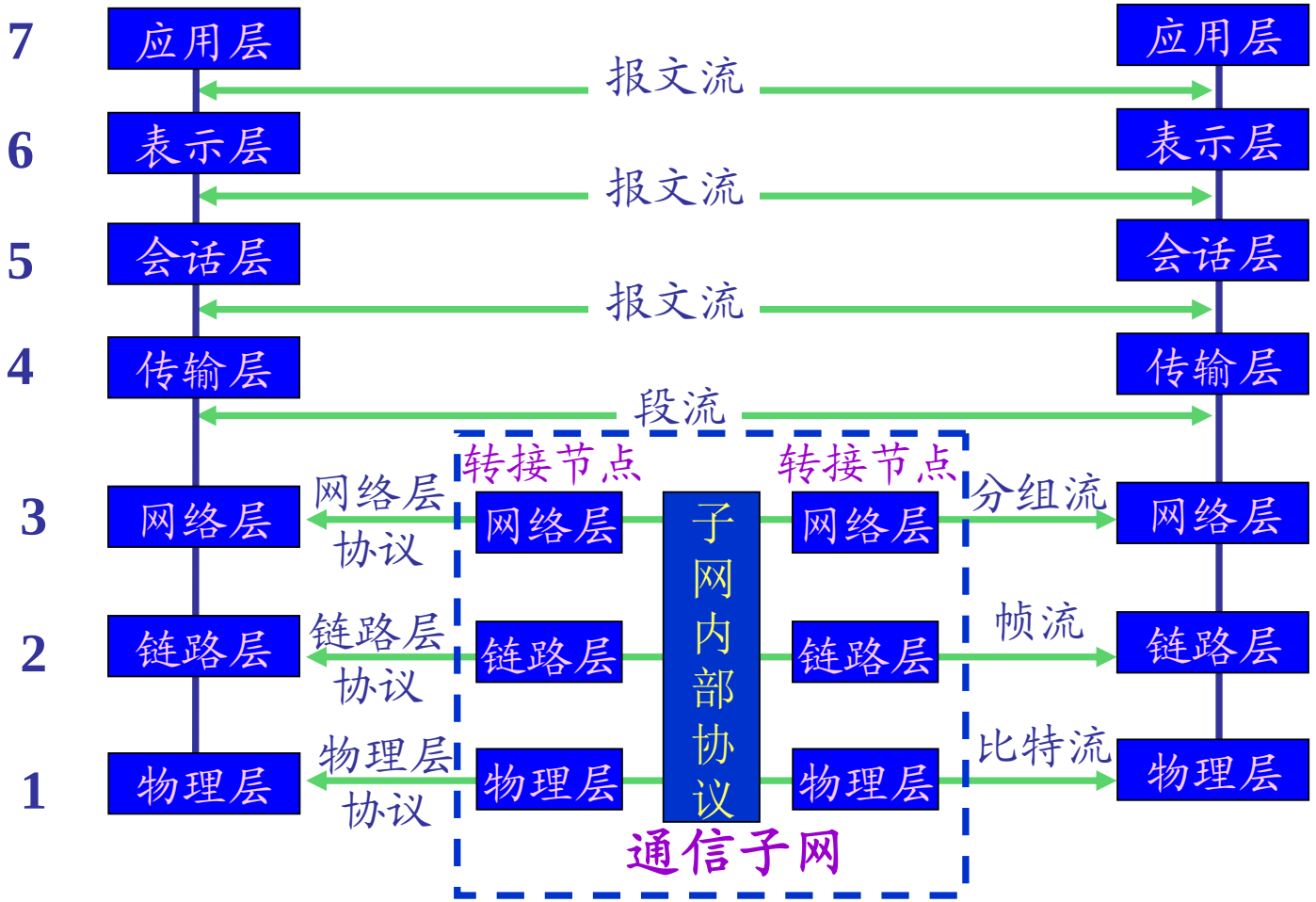
DBMS中的“三级模式-两层映像”



计算机操作系统的层次结构

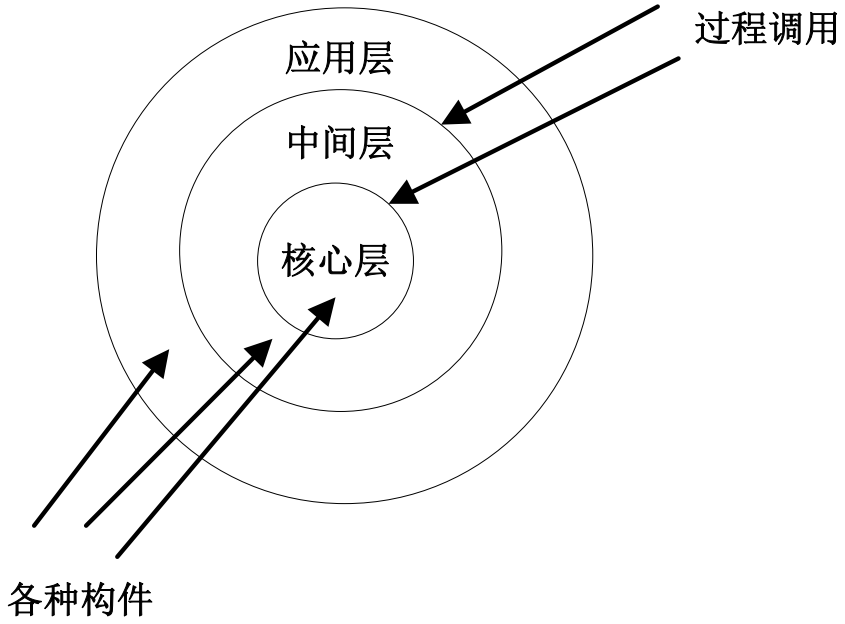


网络的分层模型


7
应用层
报文流
应用层
6
表示层
报文流
表示层
5
会话层
报文流
会话层
4
传输层
段流
传输层
转接节点
转接节点
3
网络层
网络层协议
网络层
子网内部协议
网络层
分组流
网络层
2
链路层
链路层协议
链路层
链路层
帧流
链路层
1
物理层
物理层协议
物理层
物理层
比特流
物理层
通信子网

层次系统

- 在层次系统中，系统被组织成若干个层次，每个层次由一系列构件组成；
- 层次之间存在接口，通过接口形成call/return的关系
  - 下层构件向上层构件提供服务
  - 上层构件被看作是下层构件的客户端



层次系统的优点

- 这种风格支持基于可增加抽象层的设计，允许将一个复杂问题分解成一个增量步骤序列的实现。
- 不同的层次处于不同的抽象级别：
  - 越靠近底层，抽象级别越高；
  - 越靠近顶层，抽象级别越低；
- 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。

5 C/S和B/S结构

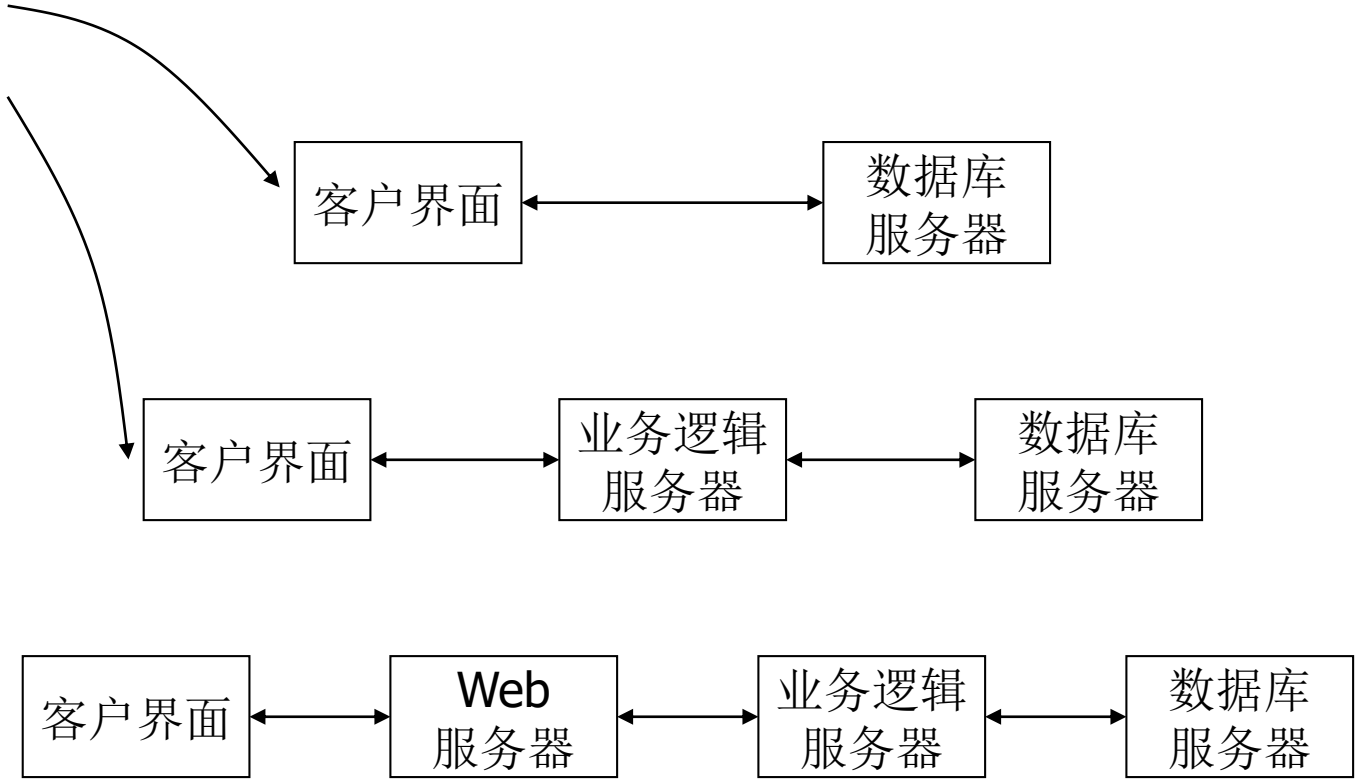
两种经典的分层结构

“客户机-服务器”架构

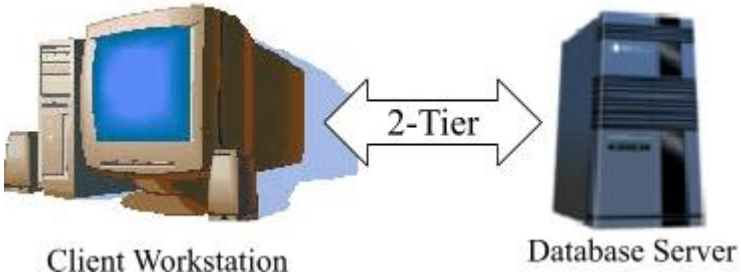
- **客户机/服务器：一个应用系统被分为两个逻辑上分离的部分，每一部分充当不同的角色、完成不同的功能，多台计算机共同完成统一的任务。**
  - 客户机(前端，front-end)：业务逻辑、与服务器通讯的接口；
  - 服务器(后端：back-end)：与客户机通讯的接口、业务逻辑、数据管理。
- **一般的，**
  - 客户机为完成特定的工作向服务器发出请求；
  - 服务器处理客户机的请求并返回结果。

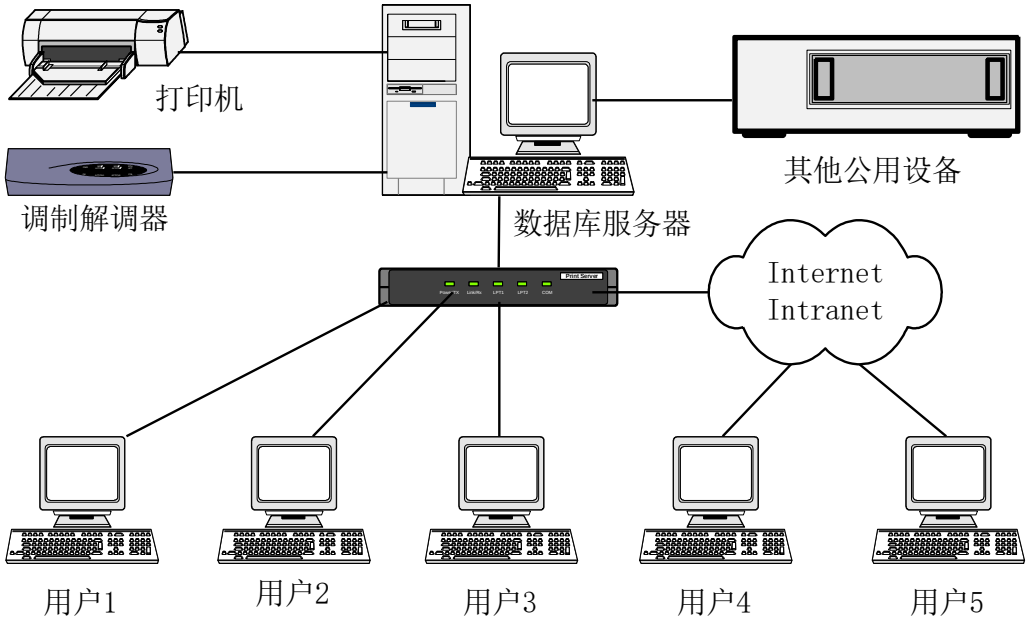
客户机/服务器的层次性

- **“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

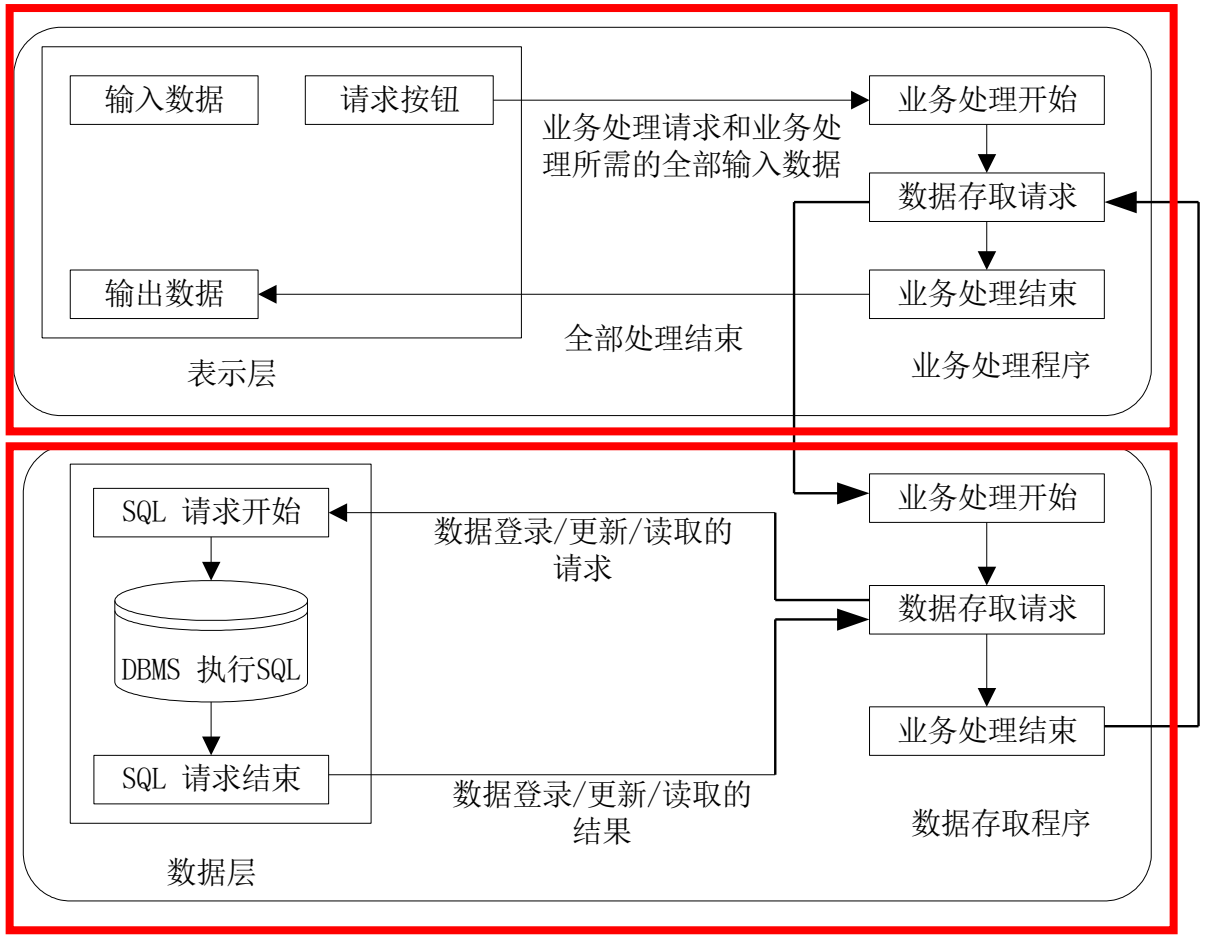


两层C/S结构


2-Tier
Client Workstation
Database Server

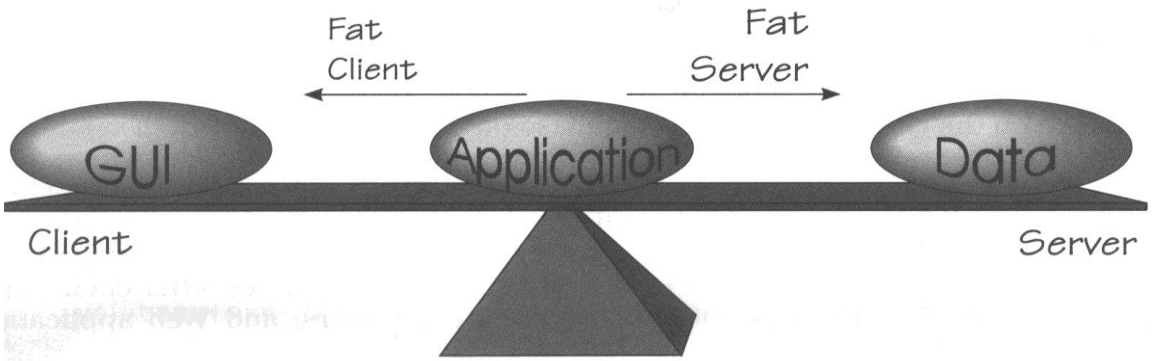

打印机
调制解调器
数据库服务器
其他公用设备
Internet
Intranet
用户1
用户2
用户3
用户4
用户5

两层C/S结构


输入数据
请求按钮
业务处理请求和业务处理所需的全部输入数据
业务处理开始
数据存取请求
输出数据
业务处理结束
全部处理结束
表示层
业务处理程序
SQL 请求开始
数据登录/更新/读取的请求
业务处理开始
数据存取请求
DBMS 执行SQL
业务处理结束
SQL 请求结束
数据登录/更新/读取的结果
数据存取程序
数据层

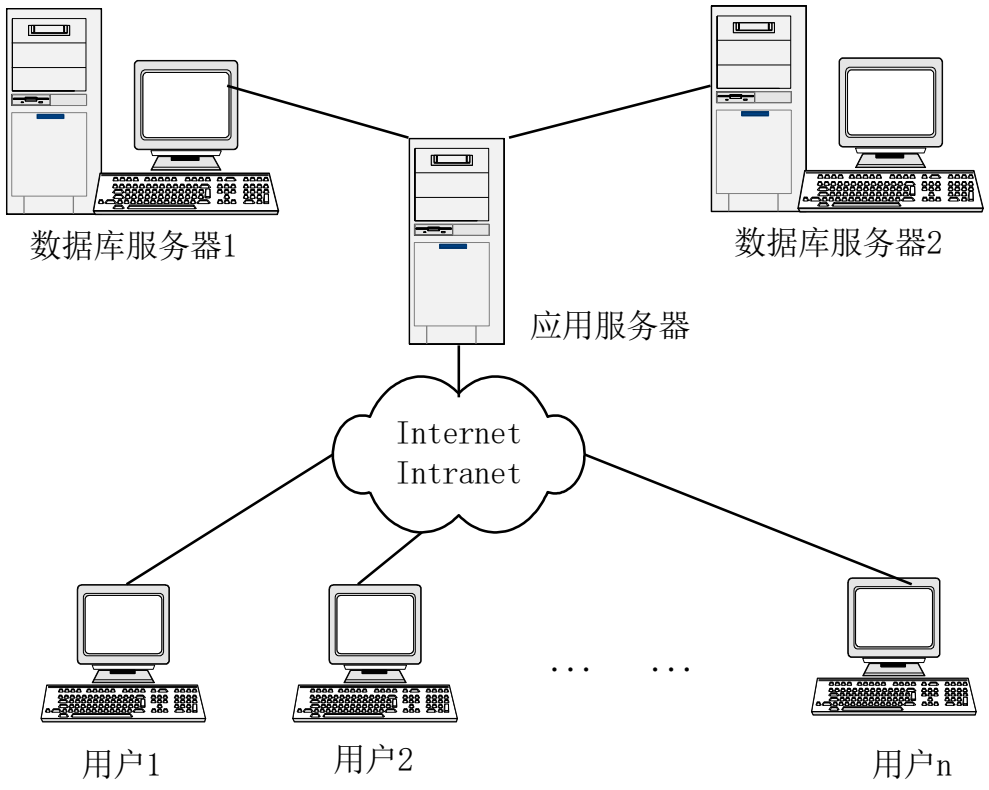
胖客户端与瘦客户端

- **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

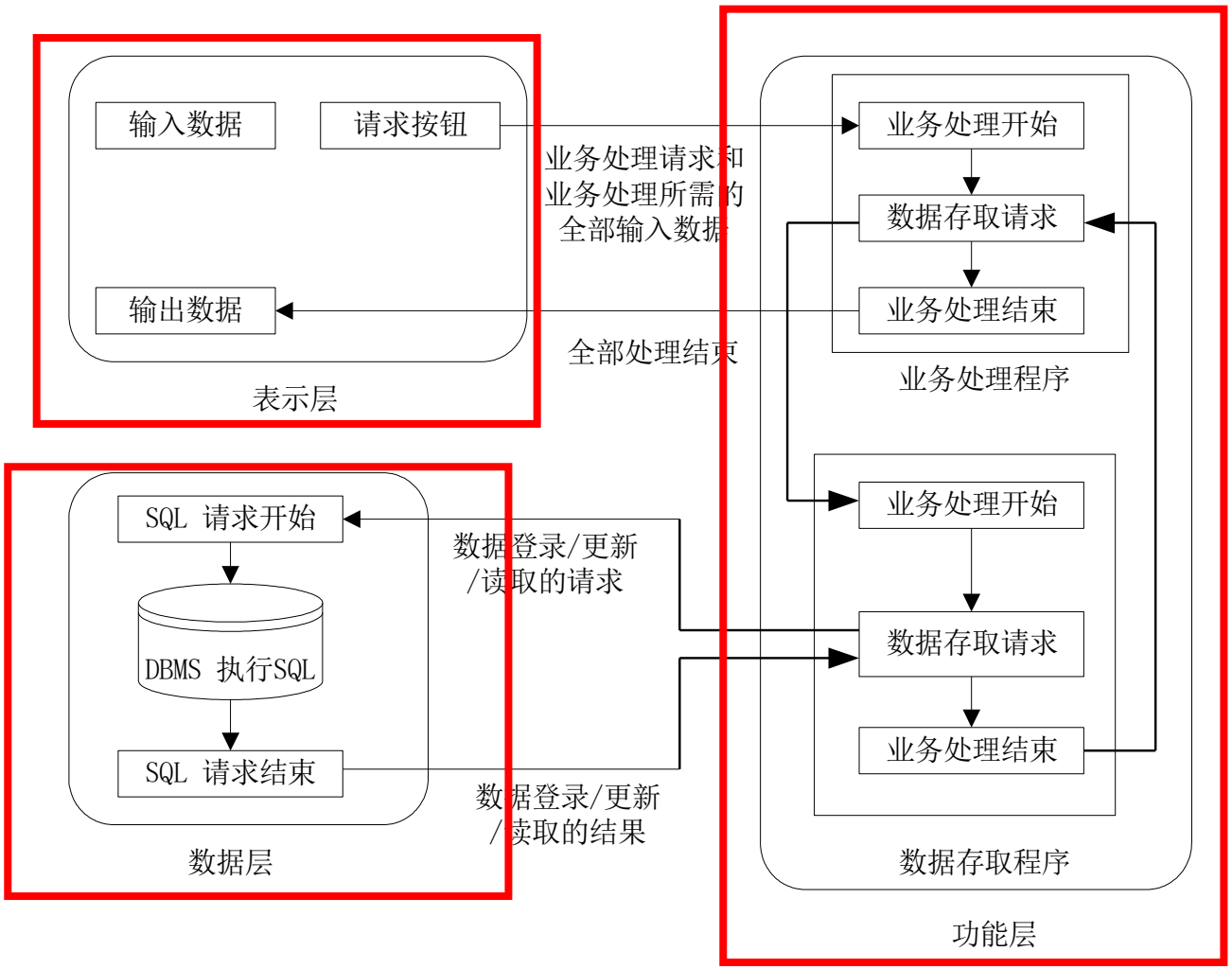


三层C/S架构

- **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

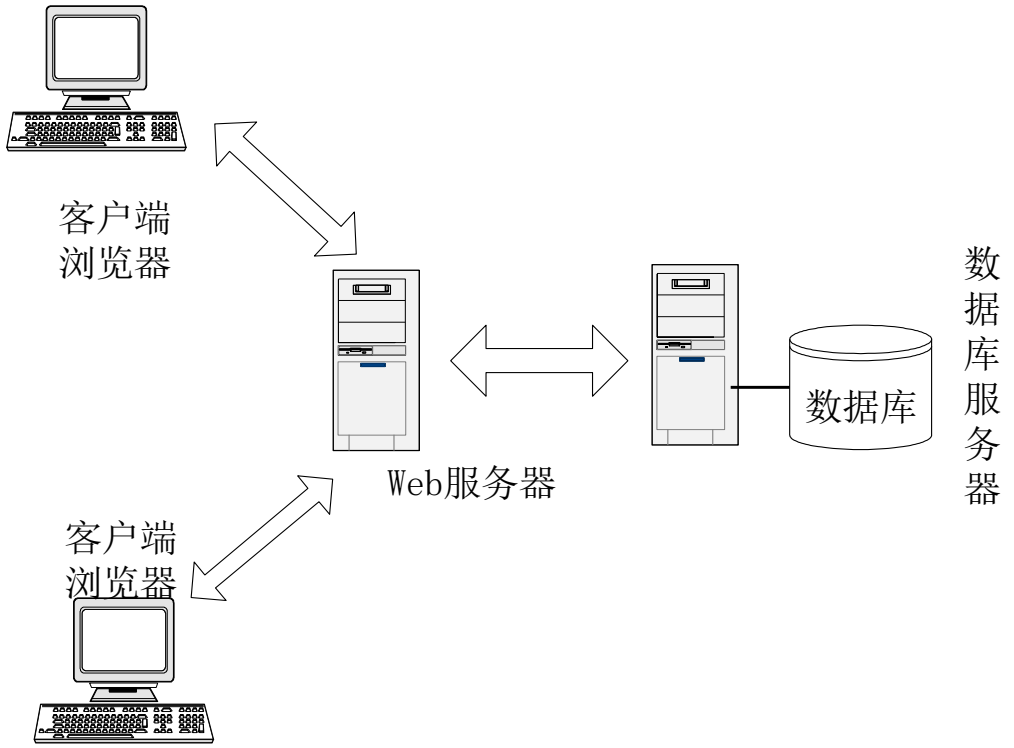


三层C/S结构

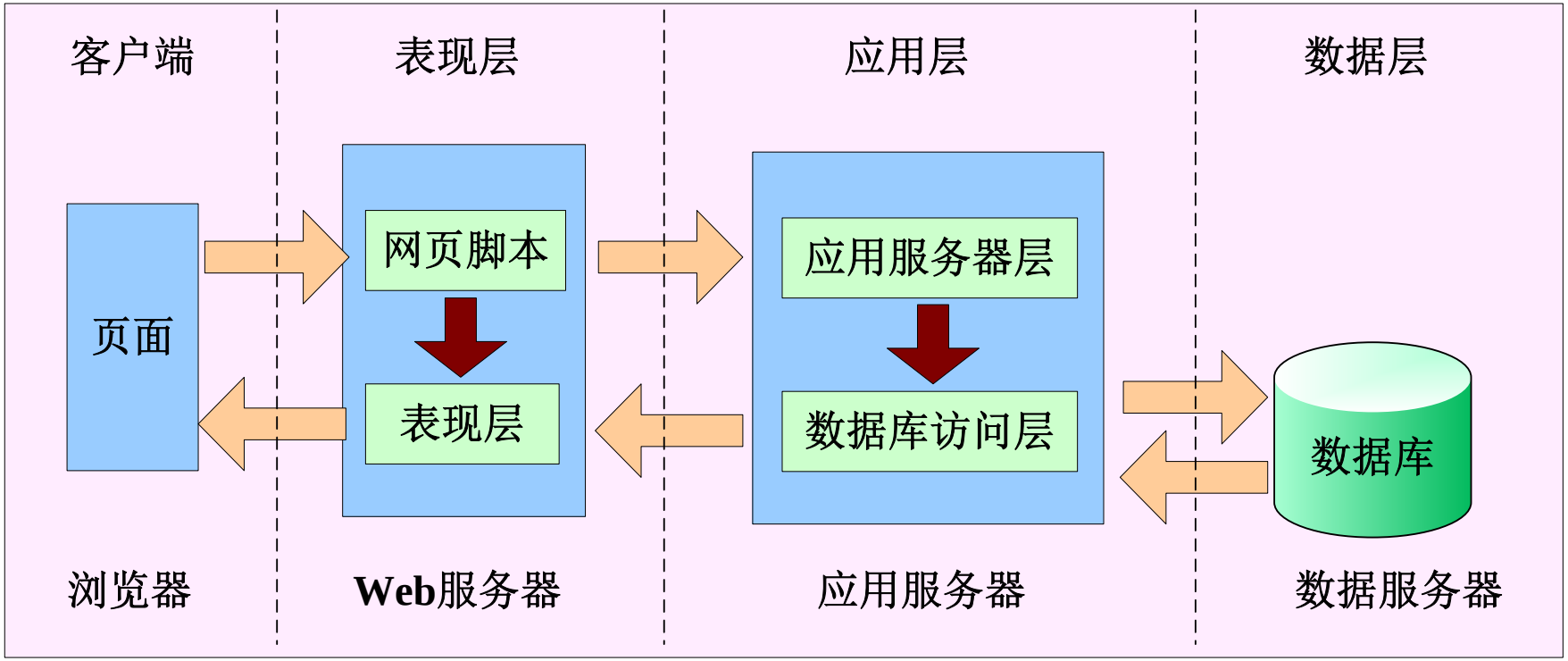

输入数据
请求按钮
输出数据
表示层
业务处理请求和
业务处理所需的
全部输入数据
全部处理结束
业务处理开始
数据存取请求
业务处理结束
业务处理程序
SQL 请求开始
DBMS 执行SQL
SQL 请求结束
数据层
数据登录/更新
/读取的请求
数据登录/更新
/读取的结果
业务处理开始
数据存取请求
业务处理结束
数据存取程序
功能层

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



B/S结构


客户端
表现层
应用层
数据层
页面
网页脚本
表现层
应用服务器层
数据库访问层
数据库
浏览器
Web服务器
应用服务器
数据服务器

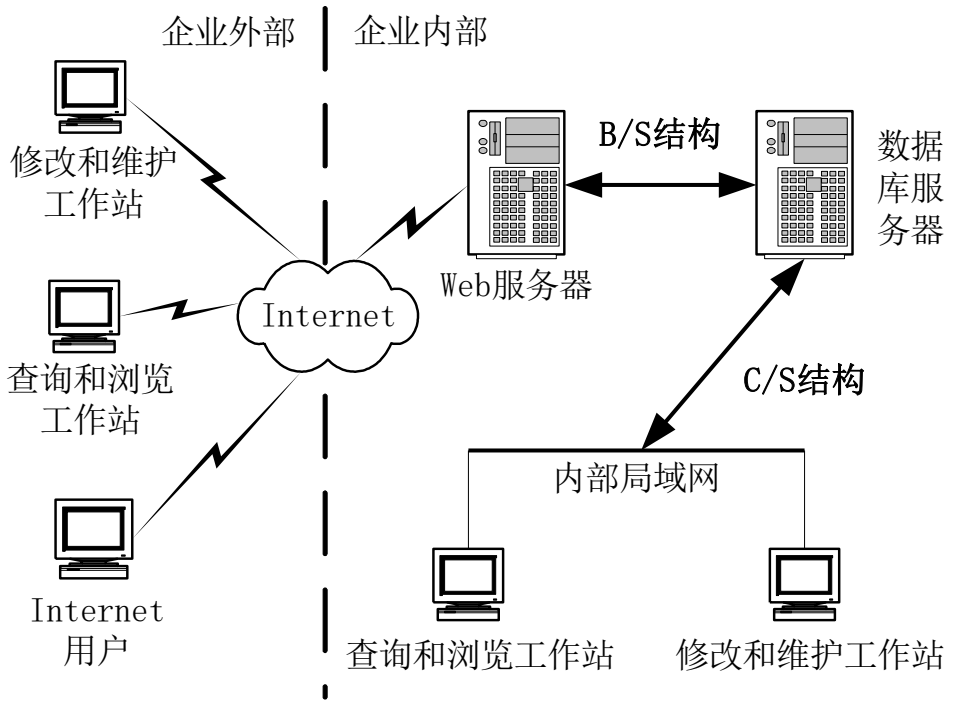
B/S结构

- **基于B/S架构的软件，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。**

C/S+B/S混合模式

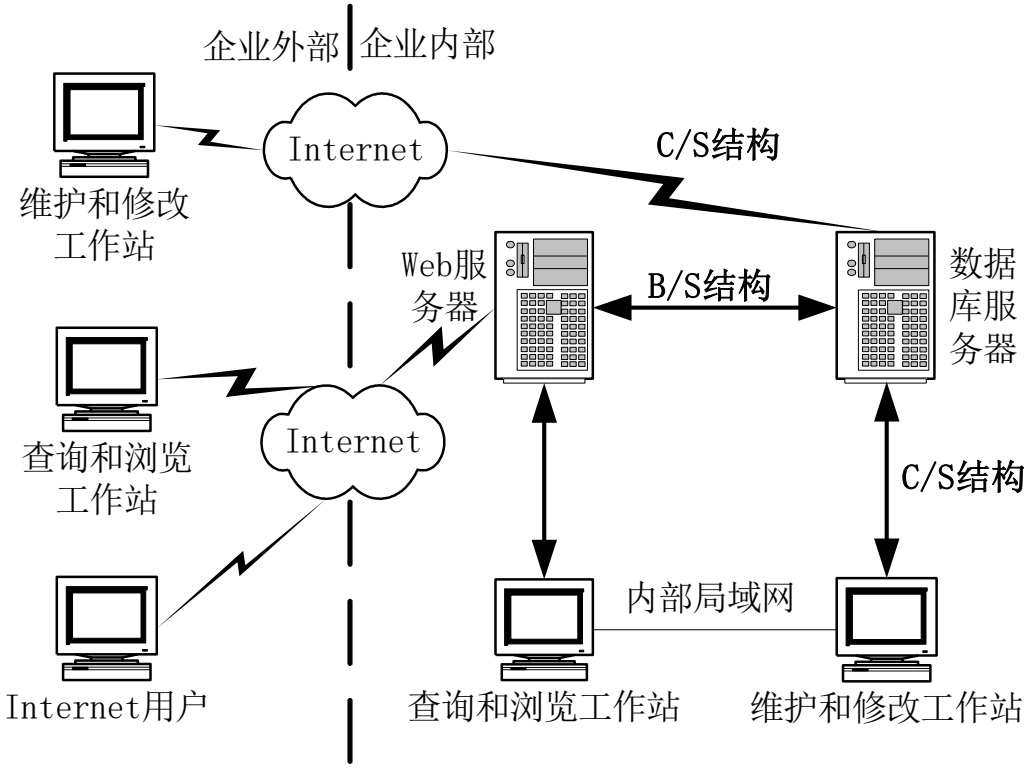
- 为了克服C/S与B/S各自的缺点，发挥各自的优点，在实际应用中，通常将二者结合起来；

- 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/S混合模式

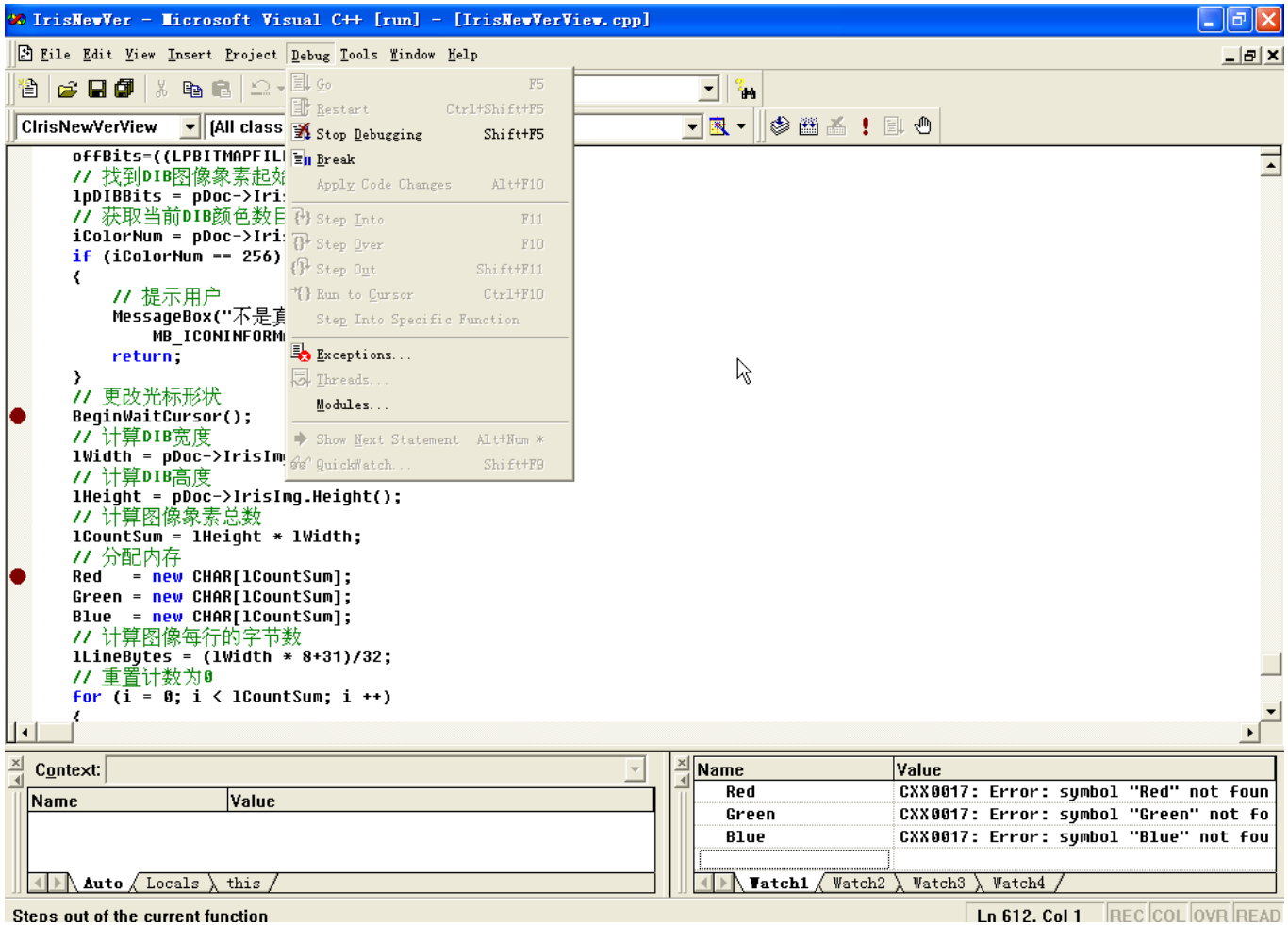
- **遵循“查改有别”的原则：**
  - 不管用户处于企业内外什么位置(局域网或Internet)，凡是需要对数据进行更新操作的(Add, Delete, Update)，都需要使用C/S结构；
  - 如果只是执行一般的查询与浏览操作(Read/Query)，则使用B/S结构。



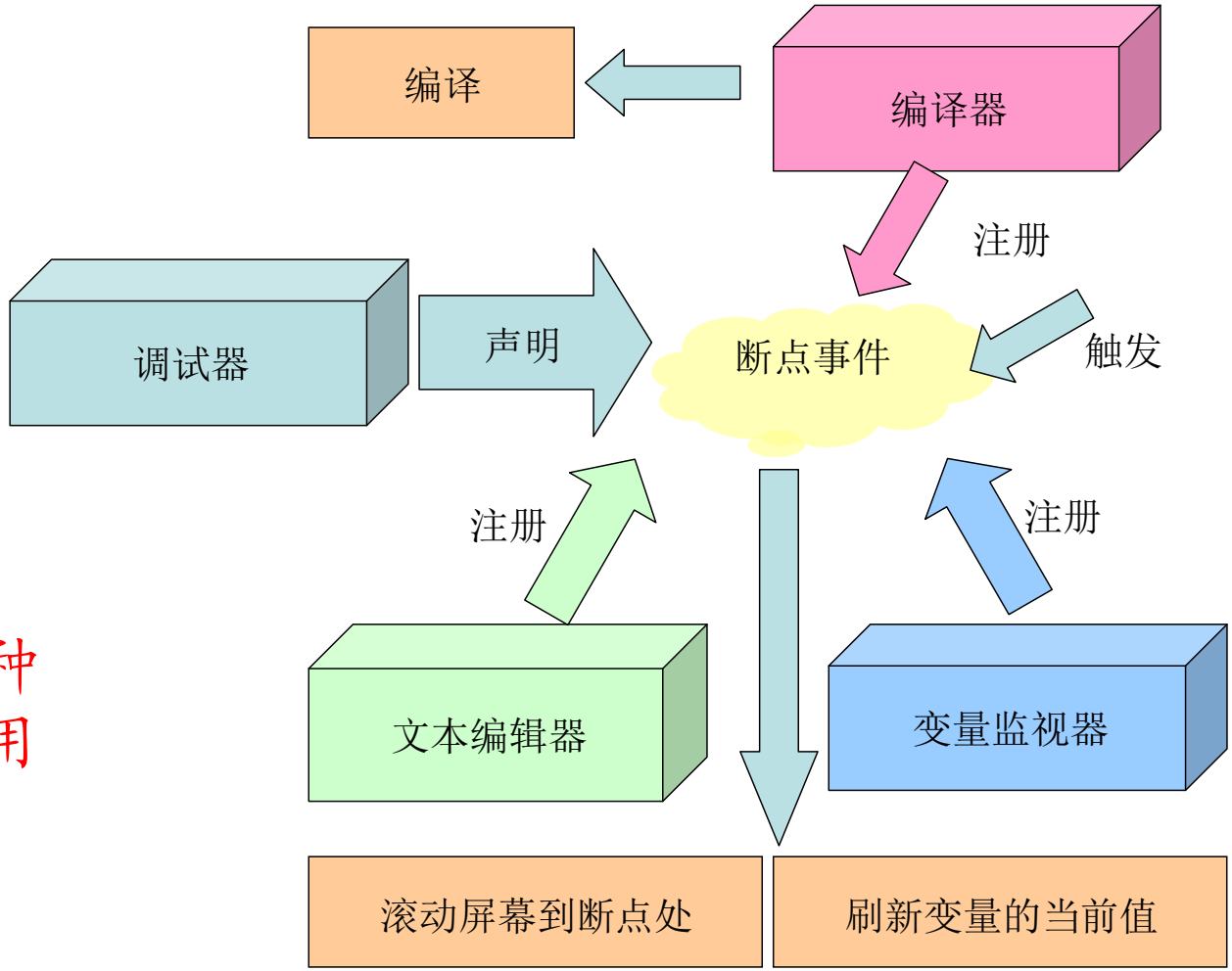
6 事件风格

与“调用-返回”方式相对应，从同步调用变为了异步调用

例子：调试器


IrisNewVer - Microsoft Visual C++ [run] - [IrisNewVerView.cpp]
File Edit View Insert Project Debug Tools Window Help
Go F5
Restart Ctrl+Shift+F5
Stop Debugging Shift+F5
Break
Apply Code Changes Alt+F10
Step Into F11
Step Over F10
Step Out Shift+F11
Run to Cursor Ctrl+F10
Step Into Specific Function
Exceptions...
Threads...
Modules...
Show Next Statement Alt+Num *
QuickWatch... Shift+F9
ClrisNewVerView
offBits=((LPBITMAPFIL
// 找到DIB图像象素起始
lpDIBBits = pDoc->Iri
// 获取当前DIB颜色数目
iColorNum = pDoc->Iri
if (iColorNum == 256)
{
// 提示用户
MessageBox("不是真
MB_ICONINFORM
return;
}
// 更改光标形状
BeginWaitCursor();
// 计算DIB宽度
lWidth = pDoc->IrisIm
// 计算DIB高度
lHeight = pDoc->IrisImg.Height();
// 计算图像象素总数
lCountSum = lHeight * lWidth;
// 分配内存
Red = new CHAR[lCountSum];
Green = new CHAR[lCountSum];
Blue = new CHAR[lCountSum];
// 计算图像每行的字节数
lLineBytes = (lWidth * 8+31)/32;
// 重置计数为0
for (i = 0; i < lCountSum; i ++)
{
Context:
Name Value
Auto Locals this
Name Value
Red CXX0017: Error: symbol "Red" not foun
Green CXX0017: Error: symbol "Green" not fo
Blue CXX0017: Error: symbol "Blue" not fou
Watch1 Watch2 Watch3 Watch4
Steps out of the current function
Ln 612, Col 1
REC COL OVR READ

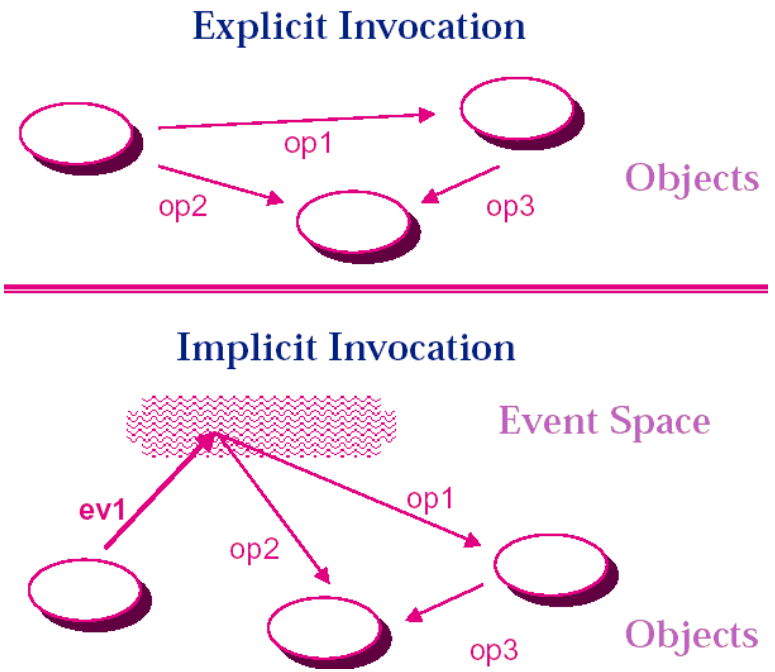
调试器的工作流程



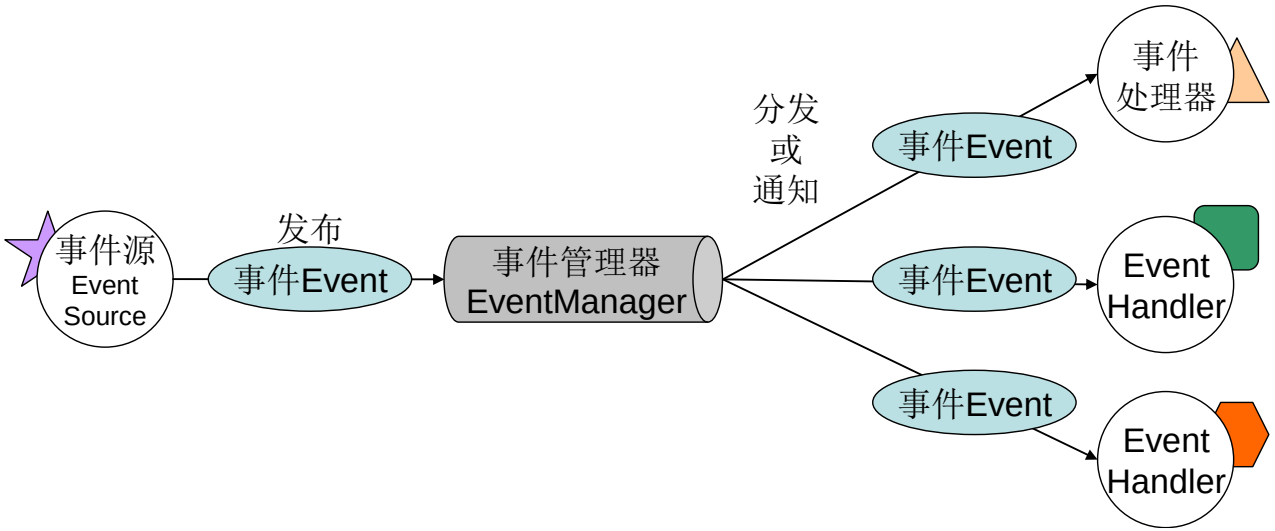
这是一种
隐式调用

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

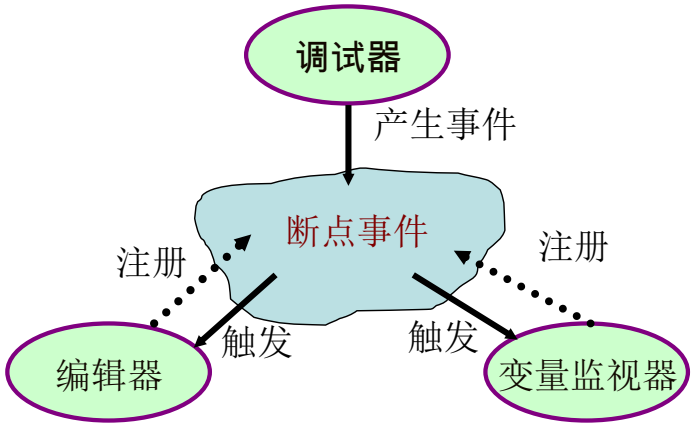


功能	描述
分离的交互	事件发布者并不会意识到事件订阅者的存在。
多对多通信	采用发布/订阅消息传递，一个特定事件可以影响多个订阅者。
基于事件的触发器	控制流由接收者确定（基于发布的事件）。
异步	通过事件消息传递支持异步操作。

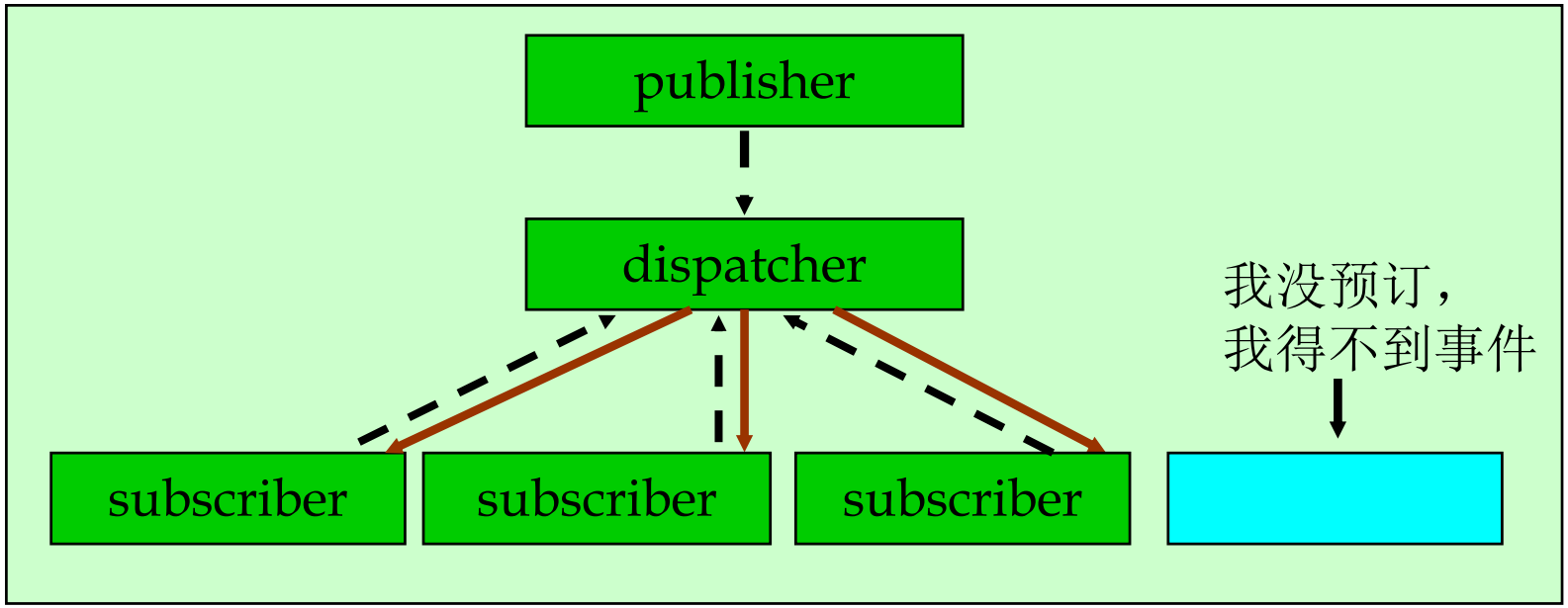
回到调试器的例子

- EventSource: debugger (调试器)
- EventHandler: editor and variable monitor (编辑器与变量监视器)
- EventManager: IDE (集成开发环境)

- 编辑器与变量监视器向调试器注册，接收“断点事件”；
- 一旦遇到断点，调试器发布事件，从而触发”编辑器”与”变量监测器”；
- 编辑器将源代码滚动到断点处，变量监测器则更新当前变量值并显示出来。

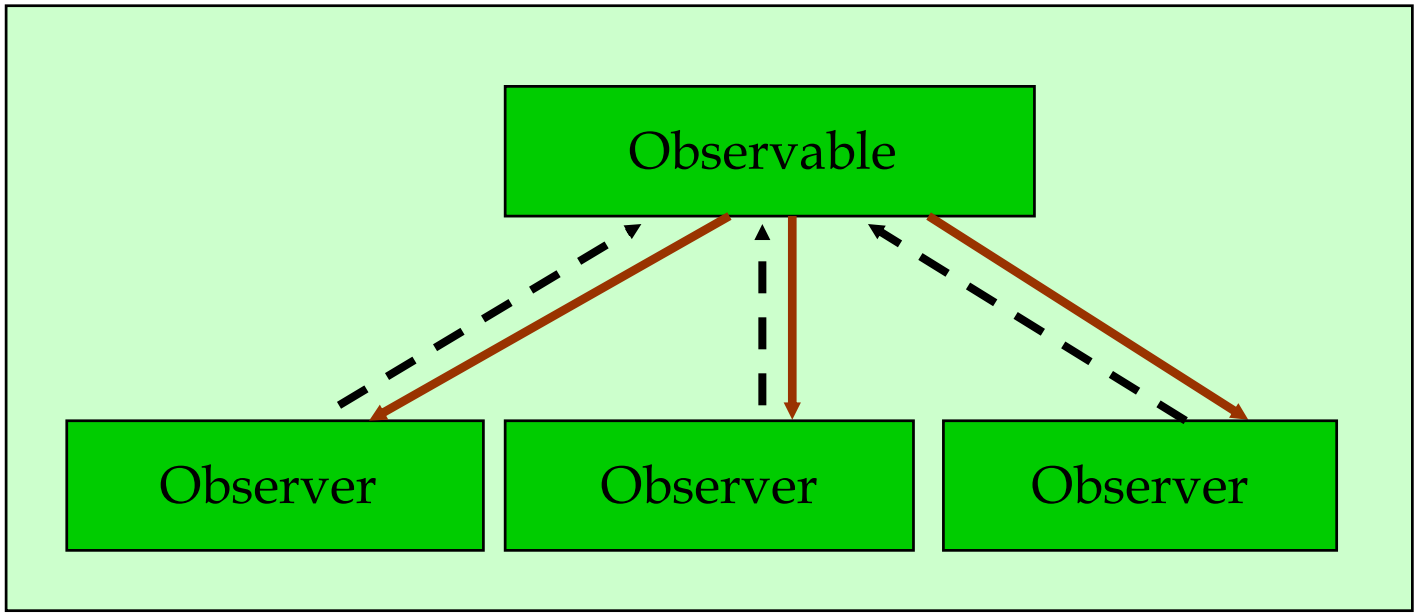


事件风格的实现策略之一：选择广播式



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



Observable/Observer module

Legend: ••••••▸ Register event ——▸ Send event

课堂讨论

- **针对你所擅长的一种编程语言（如JAVA），阐述它对C/S、B/S、事件风格（包括队列、发布-订阅、观察者模式）的实现机制，思考其各自所采用的具体编程实现技术。**

- **Amazon Simple Queue Service (SQS) 是一项快速可靠、可扩展且实惠的消息队列服务。通过查阅相关资料了解以下内容：**
  - SQS的体系结构；
  - SQS如何为你所开发的应用提供消息队列支持？
  - SQS如何保证其高可靠性和可扩展性？
  - SQS和传统的消息中间件（MOM）相比，其优势体现在哪里？
  - SQS所采用的消息格式和传输协议是什么？

结束

2017年11月7日