软件工程
第九章　软件体系结构设计

乔立民

qlm@hit.edu.cn

2011年5月11日

主要内容

- **9.1 体系结构的背景与定义**

§9.2 体系结构的基本概念

§9.3 体系结构风格

§9.4 体系结构设计

起源于建筑学的“体系结构”

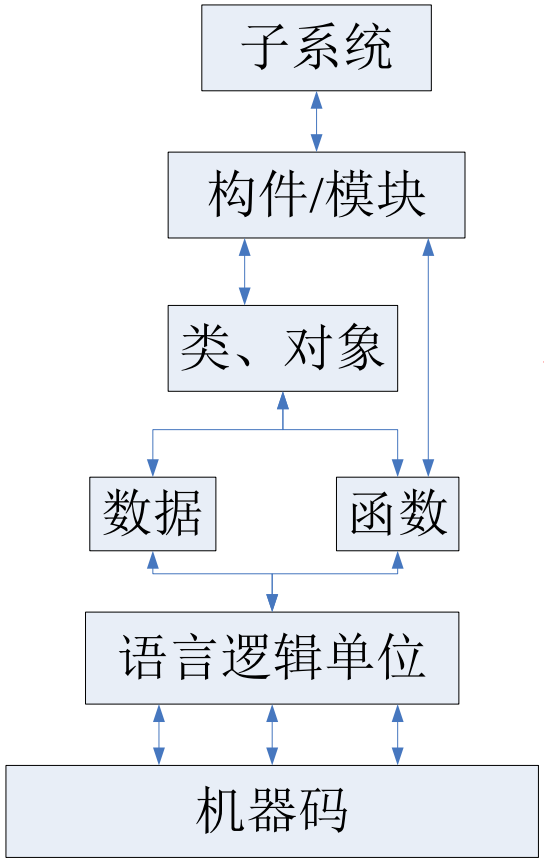
- **“体系结构(Architecture)”一词起源于建筑学**
  - 如何使用基本的建筑模块构造一座完整的建筑？
- **包含两个因素：**
  - 基本的建筑模块：砖、瓦、灰、沙、石、预制梁、柱、屋面板…
  - 建筑模块之间的粘接关系：如何把这些“砖、瓦、灰、沙、石、预制梁、柱、屋面板”有机的组合起来形成整体建筑？

计算机硬件系统的“体系结构”

- **计算机系统体系结构是程序员所看到的计算机的属性，即概念性结构与功能特性**
- **如何将设备组装起来形成完整的计算机硬件系统？**
- **包含两个因素：**
  - 基本的硬件模块：控制器、运算器、内存储器、外存储器、输入设备、输出设备…
  - 硬件模块之间的连接关系：总线
- **计算机系统体系结构的风格：**
  - SISD（单处理器）
  - SIMD（并行处理机）
  - MIMD（多处理机）

1.软件体系结构：软件构建的方式

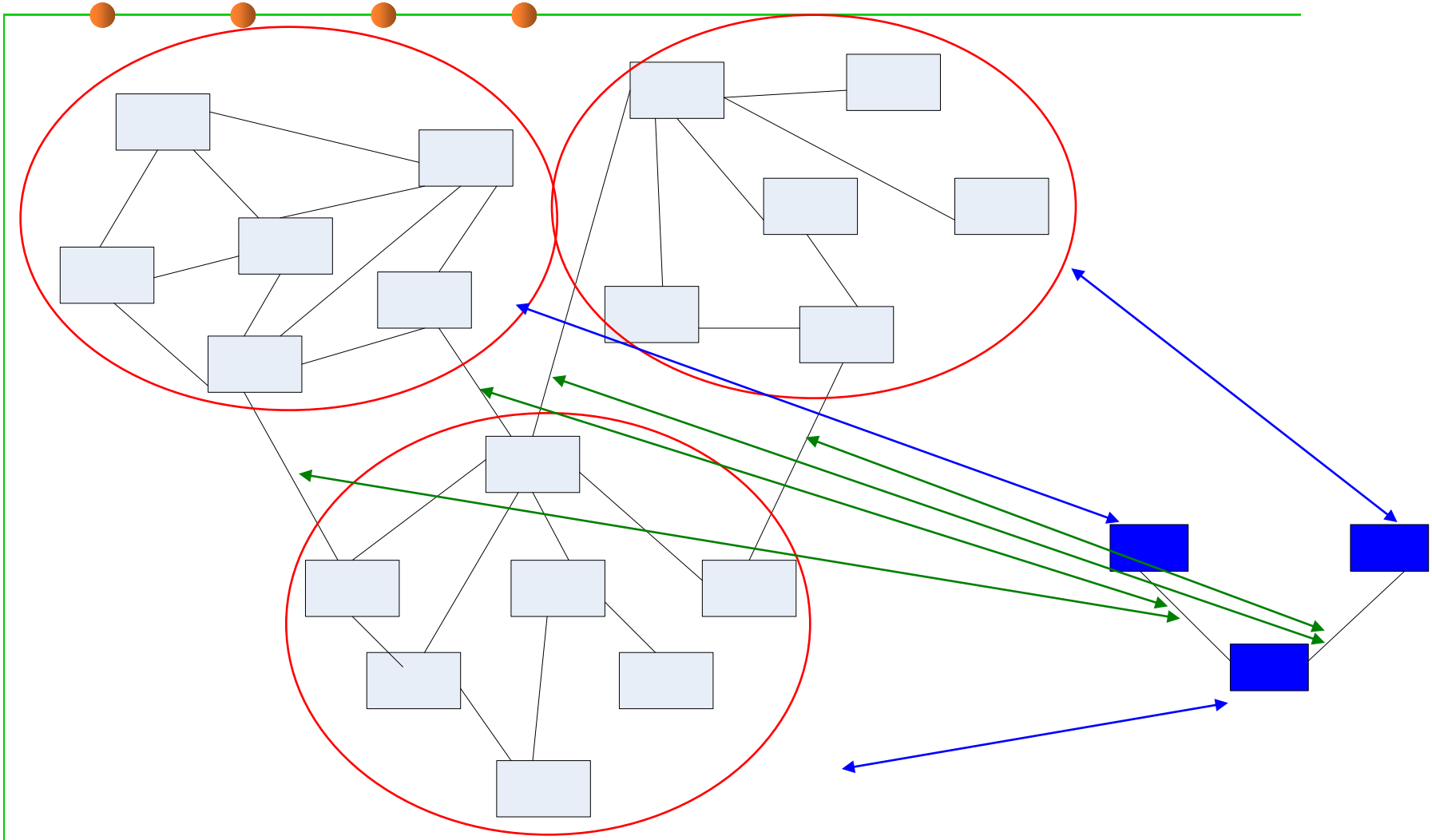
软件构建单位



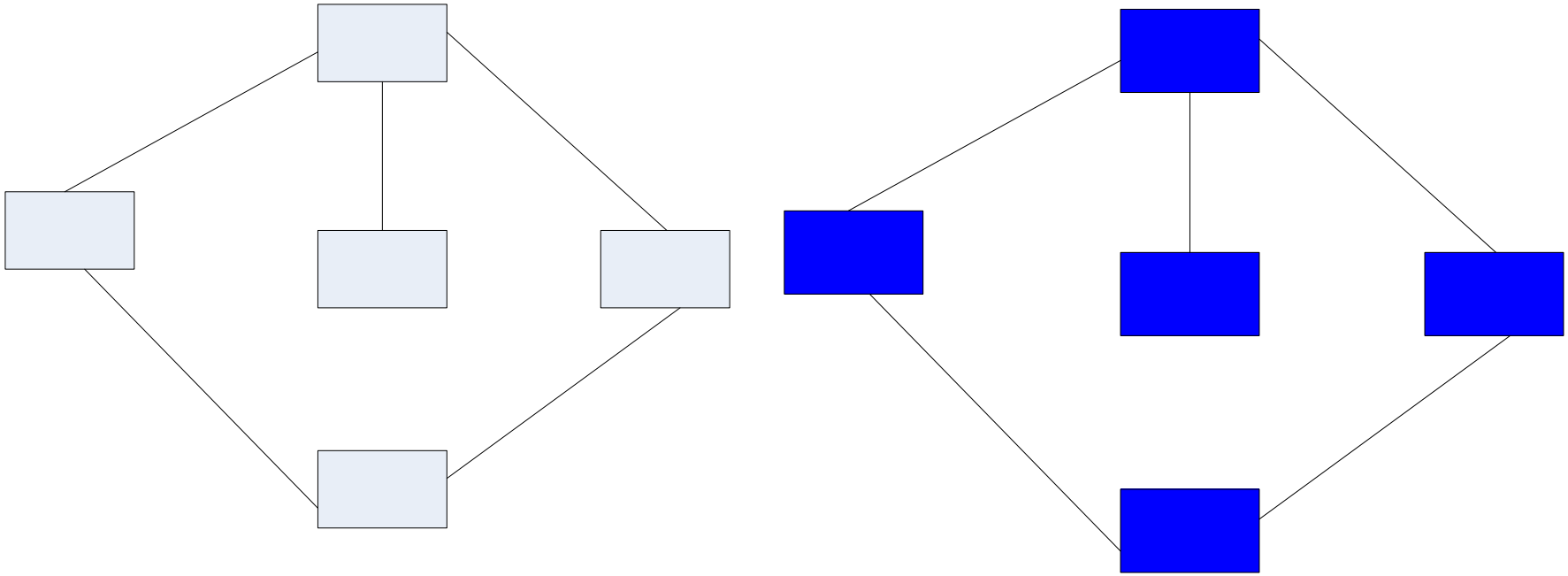
单位组织方法

- 合作
- 依赖
- 包含、继承
- 调用、使用
- 控制逻辑
- 序列

软件体系结构的例子



软件体系结构的例子



“体系结构”的共性

- **共性：**
  - 一组基本的构成要素——构件
  - 这些要素之间的连接关系——连接件
  - 这些要素连接之后形成的拓扑结构——物理分布
  - 作用于这些要素或连接关系上的限制条件——约束
  - 质量——性能

归纳：SA的定义

- **软件体系结构(SA)：**
  - 是一个关于系统形式和结构的综合框架，包括系统构件和构件的整合
  - 从一个较高的层次来考虑组成系统的构件、构件之间的连接，以及由构件与构件交互形成的拓扑结构
  - 这些要素应该满足一定的限制，遵循一定的设计规则，能够在一定的环境下进行演化
  - 反映系统开发中具有重要影响的设计决策，便于各种人员的交流，反映多种关注，据此开发的系统能完成系统既定的功能和性能需求

体系结构 = 构件 + 连接件 + 拓扑结构 + 约束 + 质量

Architecture = Components + Connectors + Topology + Constraints + Performance

2. 为什么体系结构如此重要

- 软件体系结构的表示有助于对计算机系统开发感兴趣的各方开展交流
- 体系结构突出了早期的设计决策，这些决策对随后的所有软件工程工作有深远的影响，同时对系统作为一个可运行实体的最后成功有重要作用
- 体系结构“构建了一个相对小的，易于理解的模型，该模型描述了系统如何构成以及其构件如何一起工作”
- 结论：对于大规模的复杂软件系统来说，对总体的系统结构设计和规格说明比起对计算的算法和数据结构的选择已经变得明显重要得多。

3. 软件体系结构要回答的基本问题

- **从建筑体系结构看起**
  - 基本的建筑单元都有哪些？
  - 有哪些实用、美观、强度、造价合理、可复用的大粒度建筑单元，使建造出来的建筑更能满足用户的需求？
  - 建筑模块怎样搭配才合理？
  - 有哪些典型的建筑风格？
  - 每种典型建筑(医院、工厂、旅馆)的典型结构是什么样子？需要什么样的构件？
  - 如何绘制建筑体系结构的图纸？如何根据图纸进行质量评估？
  - 如何快速节省的将图纸变为实物(即施工过程)？
  - 建筑完成之后，如何对其进行恰当程度的修改？重要模块有了更改后，如何保证整栋建筑质量不受影响？

软件体系结构要回答的基本问题

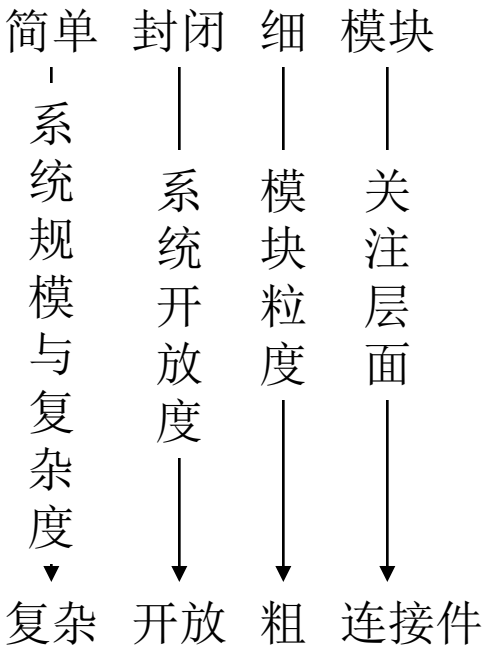
- 软件的基本构造单元是什么？
- 这些构造单元之间如何连接？
- 最终形成何种样式的拓扑结构？
- 每个典型应用领域的典型体系结构是什么样子？
- 如何进行软件体系结构的设计与实现？
- 如何对已经存在的软件体系结构进行修改？
- 使用何种工具来支持软件体系结构的设计？
- 如果对软件的体系结构进行描述，并据此进行分析和验证？

4. 软件体系结构的目标与作用

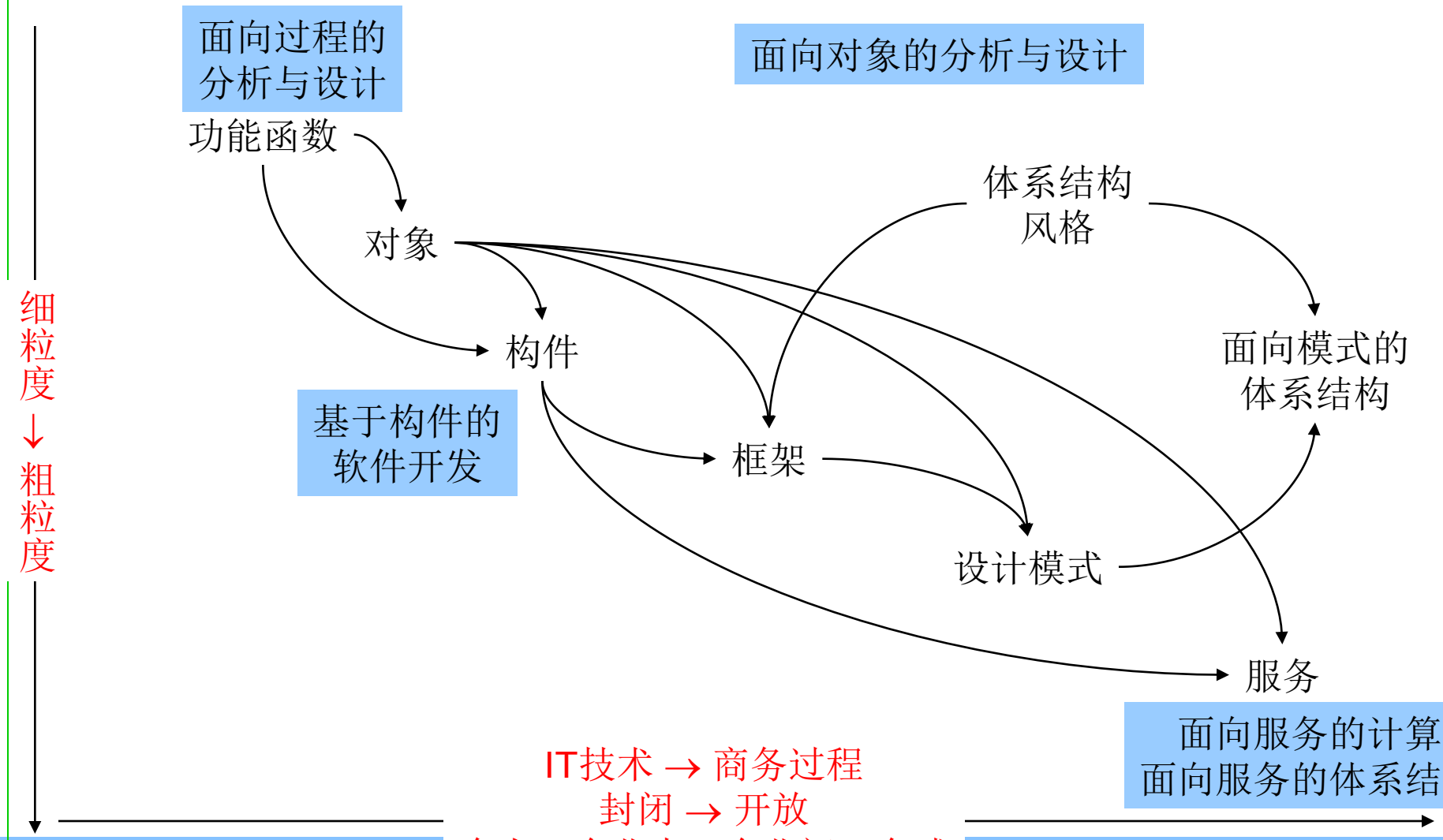
- **软件体系结构关注的是：**
  - 如何将复杂的软件系统划分为模块、如何规范模块的构成和性能、以及如何将这些模块组织为完整的系统。
- **主要目标：**
  - 建立一个一致的系统及其视图集，并表达为最终用户和软件设计者需要的结构形式，支持用户和设计者之间的交流与理解。
- **作用：**
  - 交流的手段：在软件设计者、最终用户之间方便的交流；
  - 可传递的、可复用的模型：对一些经过实现证明的体系结构进行复用，从而提高设计的效率和可靠性，降低设计的复杂度。
  - 早期决策的体现：全面表达和深刻理解系统的高层次关系，使设计者在复杂的、矛盾的需求面前作出正确的选择；正确的体系结构是系统成功的关键，错误选择会造成灾难性后果；

5. 软件体系结构的发展与演化

- 系统 = 算法 + 数据结构 (1960’s)
- 系统 = 子程序 + 函数调用 (1970’s)
- 系统 = 对象 + 消息 (1980’s)
- 系统 = 构件 + 连接件 (1990’s)
- 系统 = 服务 + 服务总线 (2000’s)



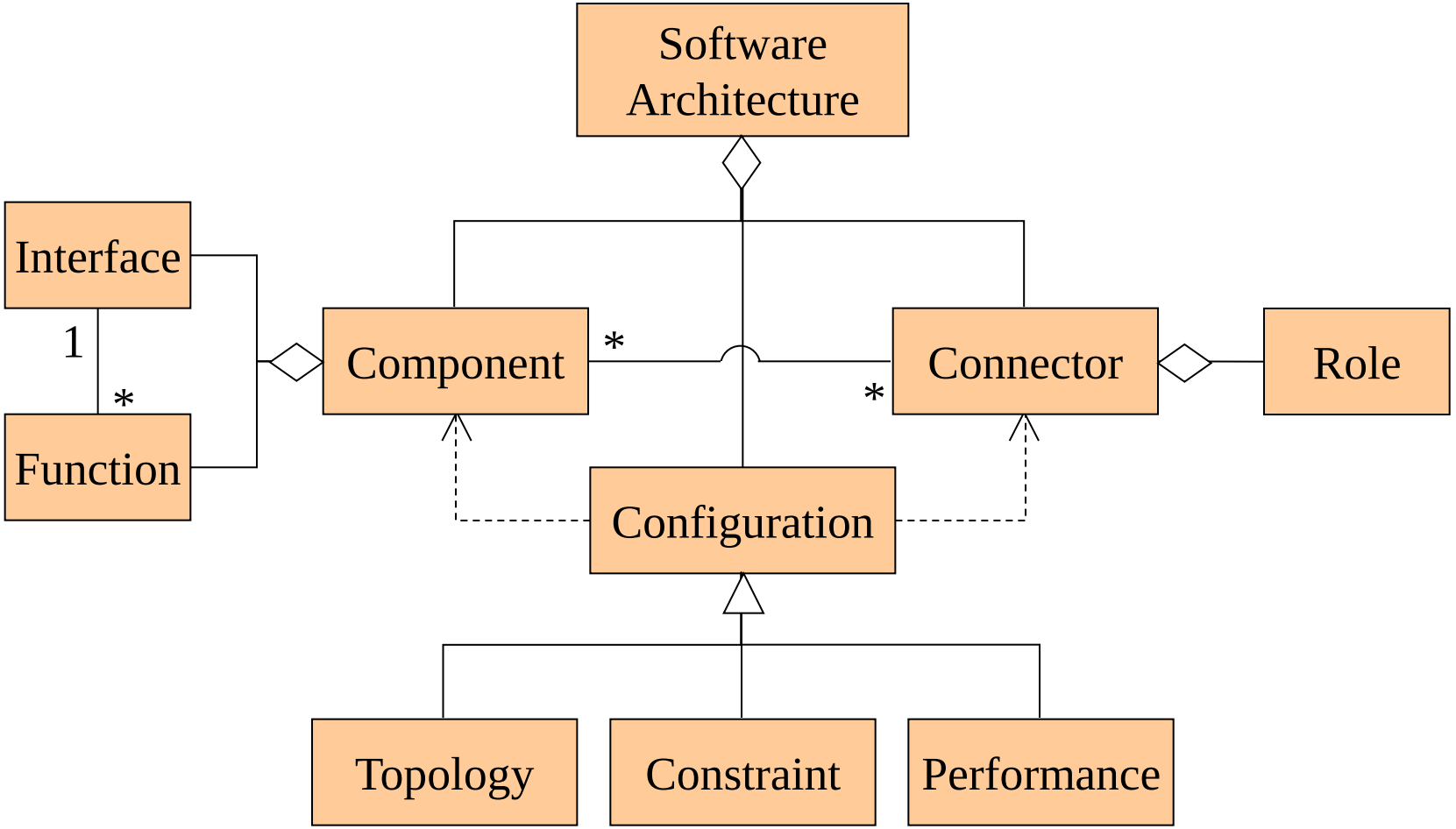
软件体系结构的演化史


面向过程的
分析与设计
面向对象的分析与设计
功能函数
对象
体系结构
风格
构件
面向模式的
体系结构
基于构件的
软件开发
框架
设计模式
服务
面向服务的计算
面向服务的体系结构
细粒度↓粗粒度
IT技术 → 商务过程
封闭 → 开放
个人→企业内→企业间→全球

主要内容

- **9.1 软件体系结构的背景与定义**
- **§9.2 软件体系结构的基本概念**
- **§9.3 软件体系结构风格**
- **§9.4 体系结构设计**

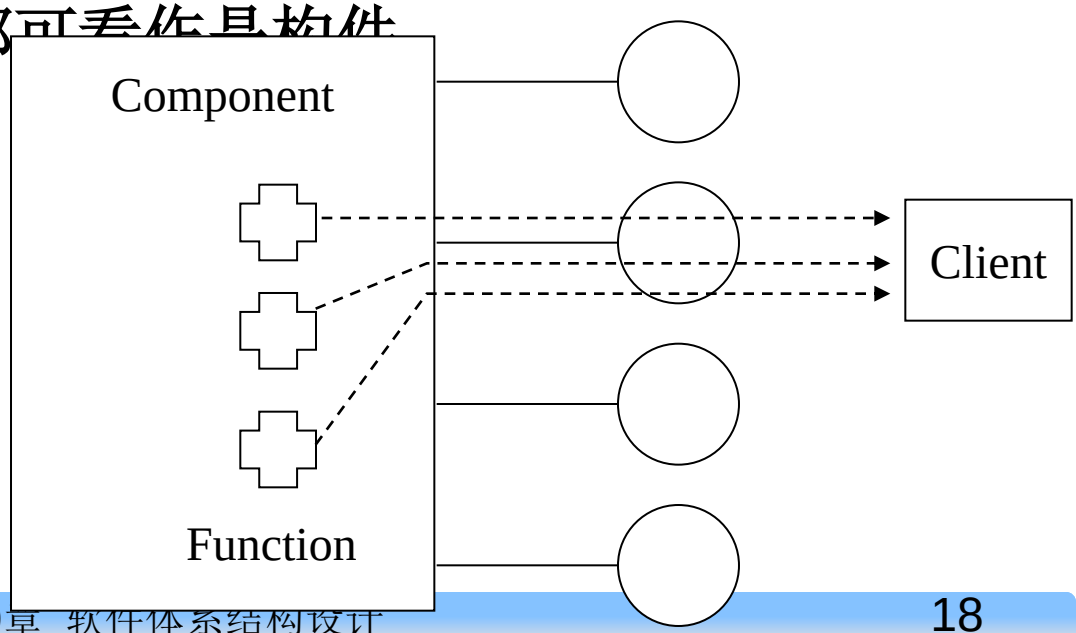
SA中的核心概念


Software Architecture
Interface
1
*
Function
Component
*
Connector
*
Role
Configuration
Topology
Constraint
Performance

1.构件(Component)

- “构件”是具有某种功能的可复用的软件结构单元，表示了系统中主要的计算元素和数据存储，具有提供的接口描述。

- 构件是一个抽象的概念，任何在系统运行中承担一定功能、发挥一定作用的软件体都可看作是构件
  - 程序函数、模块
  - 对象、类
  - 数据库
  - 文件
  - ….



构件 Vs. 对象

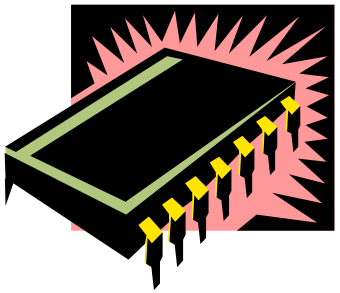
- **抽象的级别不同，构件是设计概念，而对象是实现技术**
- **规模**
  - 对象一般较小
  - 构件可以小(一个对象) 或大(一系列对象或一个完整的应用)
- **在对构件操作时不允许直接操作构件中的数据，数据真正被封装了。而对象的操作通过公共接口部分，这样数据是可能被访问操作的**
- **对象的复用是基于技术的复用（继承），构件的复用是基于功能的复用（组装）**

构件组成

- **构件是一个独立发布的功能部分，可以通过它的接口访问它的服务**
- **构件组成**
  - 接口
    - 构件接口是构件间的契约
    - 一个接口提供一种服务，完成某种逻辑行为
  - 实现（功能）
    - 构件接口服务的实现
    - 构件核心逻辑实现

接口(Interface)与功能(Function)

- 构件作为一个封装的实体，只能通过其接口(Interface)与外部环境交互，表示了构件和外部环境的交互点，内部具体实现则被隐藏起来(Black-box)；



- 构件接口与其内部实现应严格分开

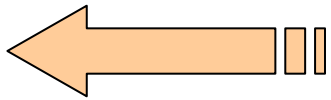
- 构件内部所实现的功能以功能(functions、behaviors)的形式体现出来，并通过接口向外发布，进而产生与其它构件之间的关联。

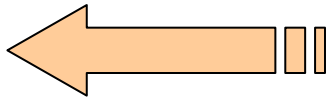
2.连接(Connection)

- **连接(Connection)：构件间建立和维护行为关联与信息传递的途径；**
- **连接需要两方面的支持：**
  - 连接发生和维持的机制——实现连接的物质基础(连接的机制)；
  - 连接能够正确、无二义、无冲突进行的保证——连接正确有效的进行信息交换的规则(连接的协议)。
  - 简称“机制”(mechanism)和“协议”(protocol)。

连接的机制(Mechanism)

- **计算机硬件提供了一切连接的物理基础：**
  - 过程调用、中断、存储、堆栈、串行I/O、并行I/O等；
- **基础控制描述层：**
  - 过程(函数)调用、中断/事件、流、文件、网络等；
- **资源及管理调度层：**
  - 进程、线程、共享、同步、并行、分时并发、事件、消息、异常、远程调用、注册表、剪贴板、动态连接、API等；
- **系统结构模式层：**
  - 管道、解释器、编译器、转换器、浏览器、中间件、ODBC等。





计算机组成原理
计算机系统结构
汇编语言

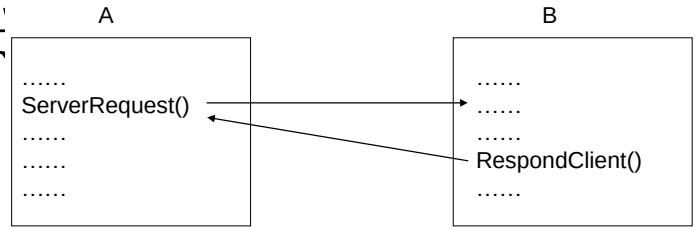
C高级程序语言
操作系统
计算机网络

操作系统

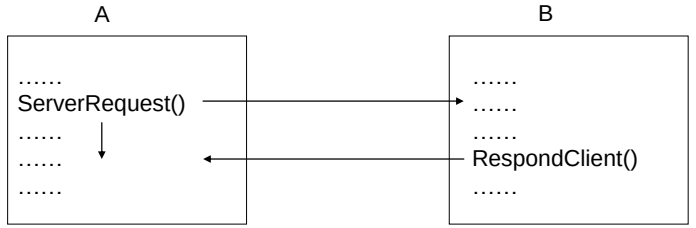
软件工程
软件体系结构

连接的种类

- **从连接目的与手段看：**
  - 函数/过程调用；　　A { call B.f();}　　B { f(); }
  - 事件/消息发送；　　A { send msg (m) to B;}　　B { receive m and do sth;}
  - 数据传输；　　A {write data to DB; }　　B {read data from DB;}
  - ..
- **除了连接机制/协议的实现难易之外，影响连接实现复杂性的因素之一是“有无连接的返回信息和返回的时间”，**
  - 同步 (Synchronous)
  - 异步 (Asynchronous)



同步连接机制



异步连接机制

连接的协议(Protocol)

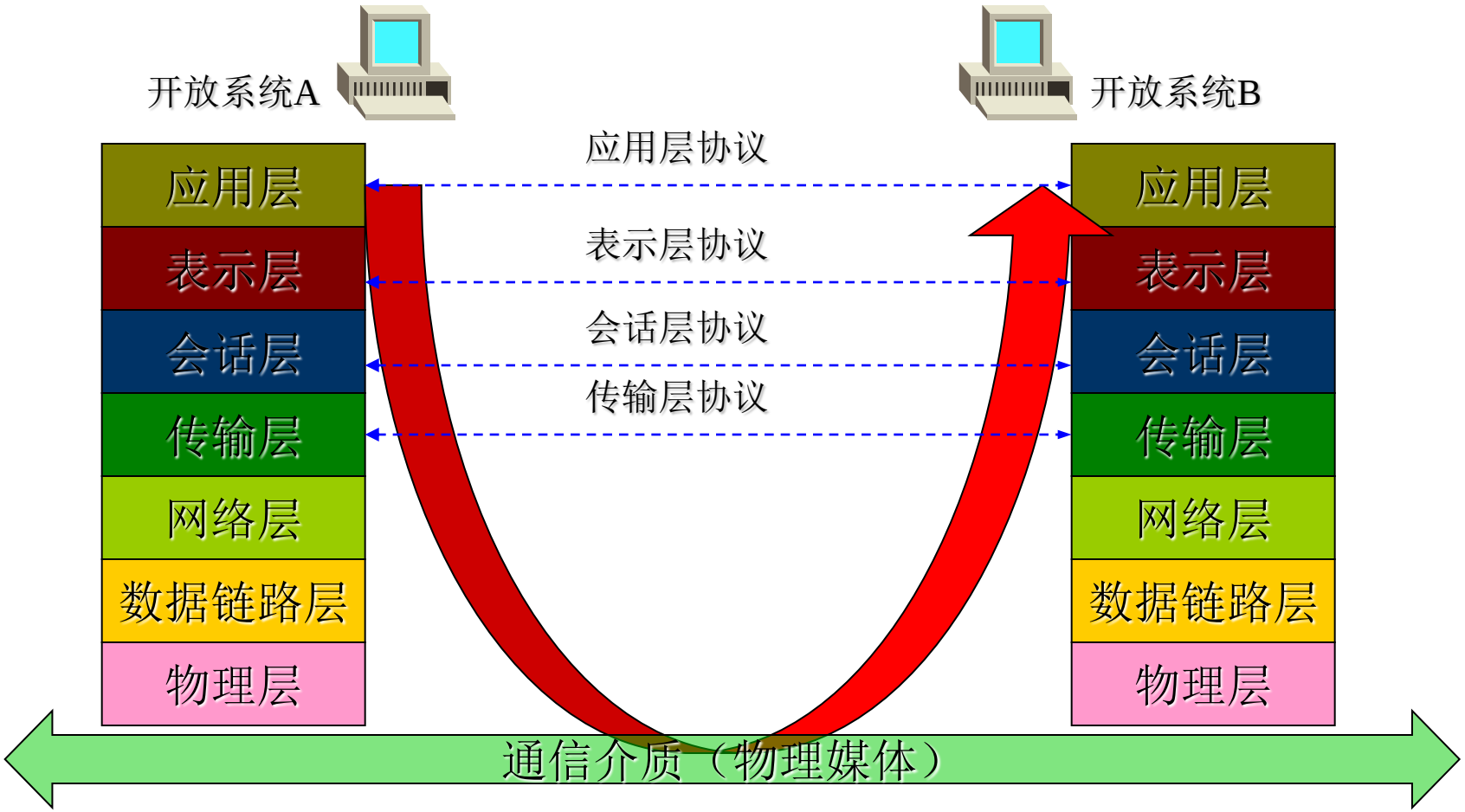
- **协议(Protocol)是连接的规约(Specification)；**
- **连接的规约是建立在物理层之上的有意义信息形式的表达规定**
  - 对过程调用来说：参数的个数和类型、参数排列次序；

    例：double GetHighestScore (int courseID, String classID) {…}
  - 对消息传送来说：消息的格式

    例：class Message {int msgNo; String bookName; String status;}

- **目的：使双方能够互相理解对方所发来的信息的语义。**

复杂的连接：协议复杂化


开放系统A
开放系统B
应用层协议
表示层协议
会话层协议
传输层协议
应用层
表示层
会话层
传输层
网络层
数据链路层
物理层
应用层
表示层
会话层
传输层
网络层
数据链路层
物理层
通信介质（物理媒体）

连接件(Connector)

- **连接件(Connector)：表示构件之间的交互并实现构件之间的连接，如：**
  - 管道(pipe)
  - 过程调用(procedure call)
  - 事件广播(event broadcast)
  - 客户机-服务器(client-server)
  - 数据库连接(SQL)

- **连接件也可看作一类特殊的构件，区别在于：**
  - 一般构件是软件功能设计和实现的承载体；
  - 连接件是负责完成构件之间信息交换和行为联系的专用构件。

主要内容

- 9.1 软件体系结构的背景与定义
- §9.2 软件体系结构的基本概念
- §9.3 软件体系结构风格
- §9.4 体系结构设计

从“建筑风格”开始

- **Architectural style constitutes a mode of classifying architecture largely by morphological characteristics in terms of form, techniques, materials, etc. (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)。**

- 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)。

中国古典建筑

宫殿风格

园林风格

江南建筑风格

欧洲古典建筑

文艺复兴风格

巴洛克风格

拜占庭风格

哥特风格

软件体系结构风格

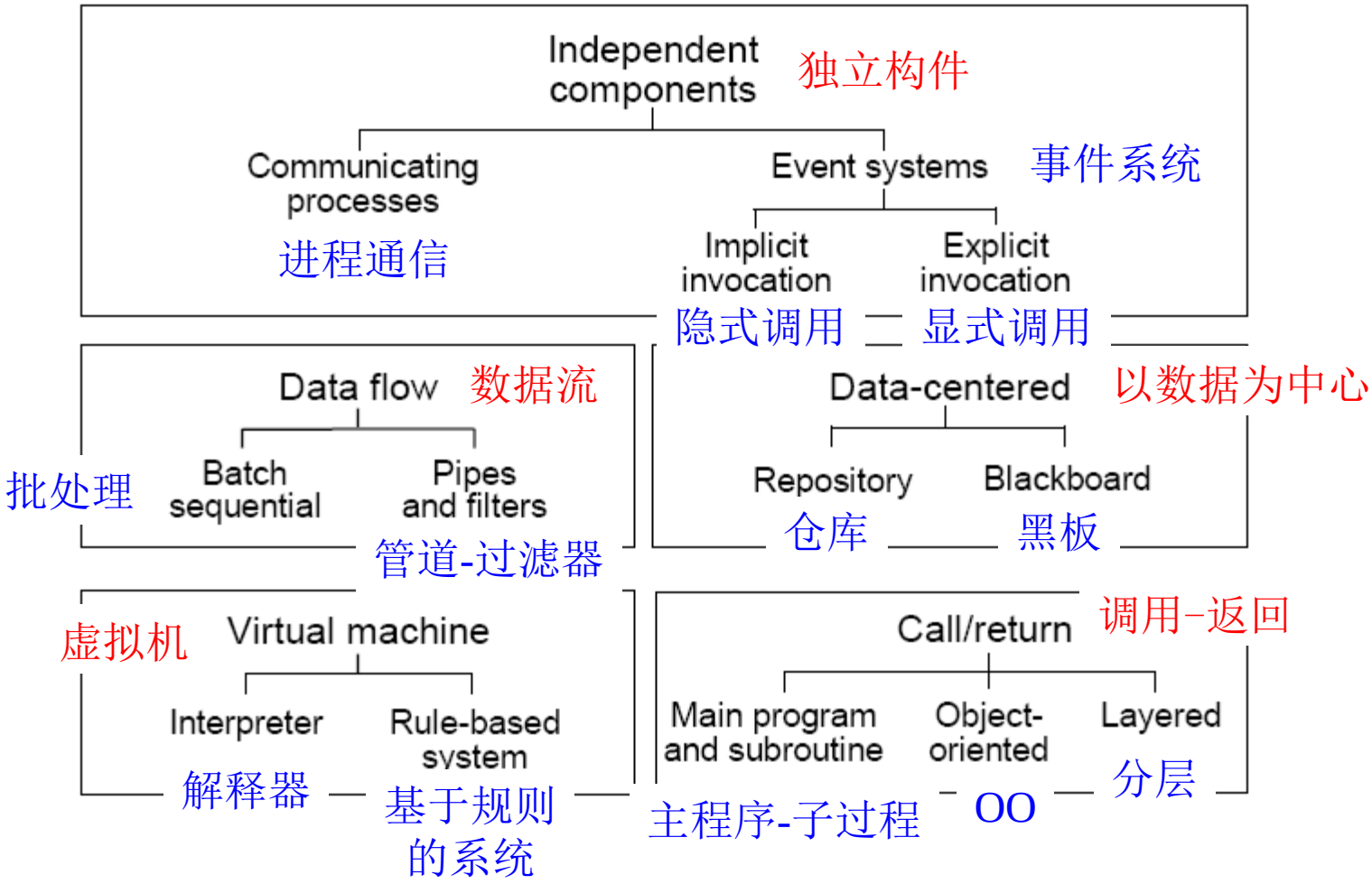
- **软件系统同建筑一样，也具有若干特定的“风格”(software architectural style)；**
  - 这些风格在实践中被多次设计、应用，已被证明具有良好的性能、可行性和广泛的应用场景，可以被重复使用；
  - 实现“软件体系结构级”的复用。

- **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。

“软件体系结构风格”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)

经典体系结构风格

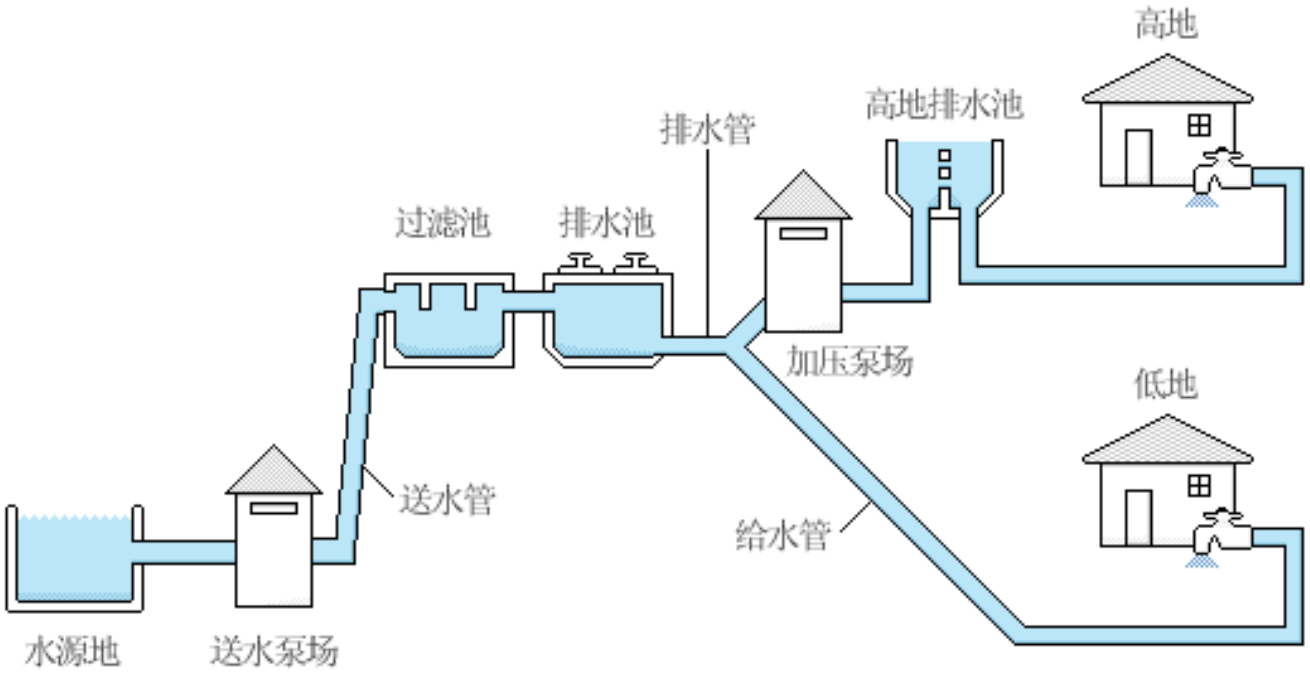

Independent components
独立构件
Communicating processes
进程通信
Event systems
事件系统
Implicit invocation
Explicit invocation
隐式调用
显式调用
Data flow
数据流
批处理
Batch sequential
Pipes and filters
管道-过滤器
Data-centered
以数据为中心
Repository
Blackboard
仓库
黑板
虚拟机
Virtual machine
Interpreter
Rule-based system
解释器
基于规则的系统
Call/return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

主要软件体系结构风格

1. **数据流风格**
2. **以数据为中心的风格(仓库)**
3. **主程序-子过程**
4. **面向对象风格**
5. **层次风格**
6. **客户机-服务器**
   - C/S
   - B/S结构
7. ***模型-视图-控制器(MVC)**

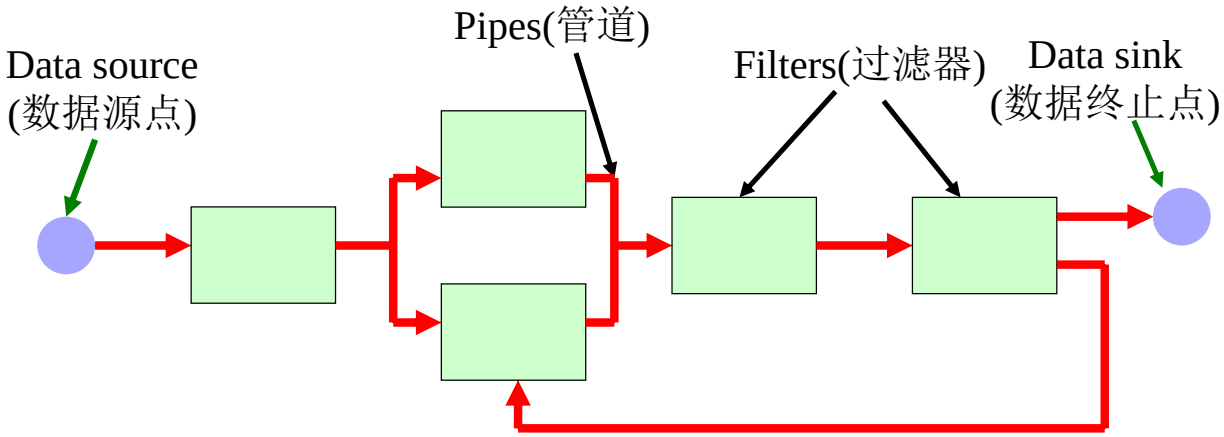
调用返回风格

1.数据流风格——现实中的“数据流”体系结构

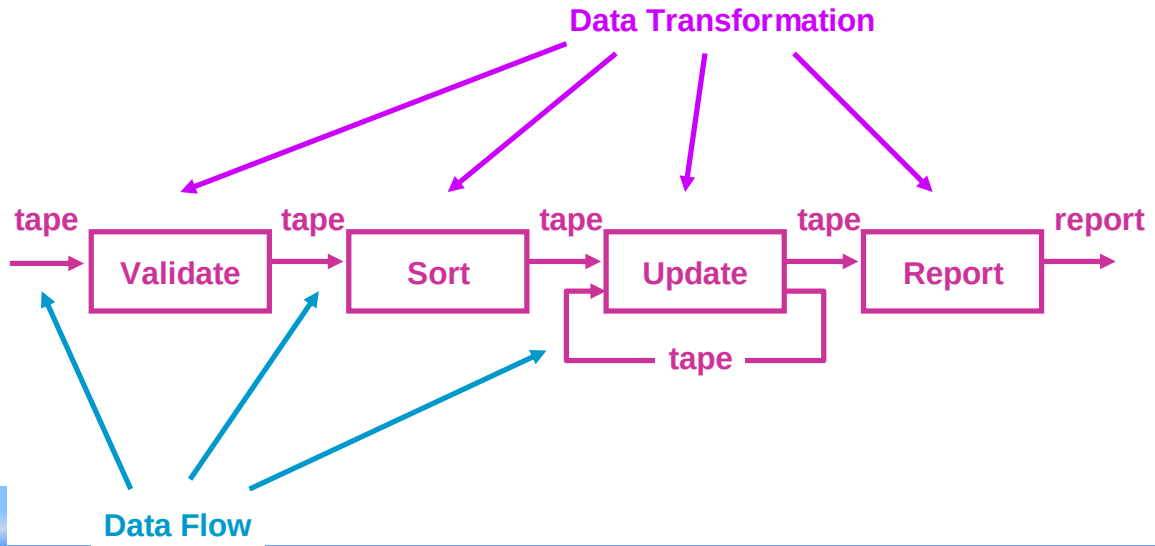


1.软件中的数据流风格

- 管道-过滤器风格

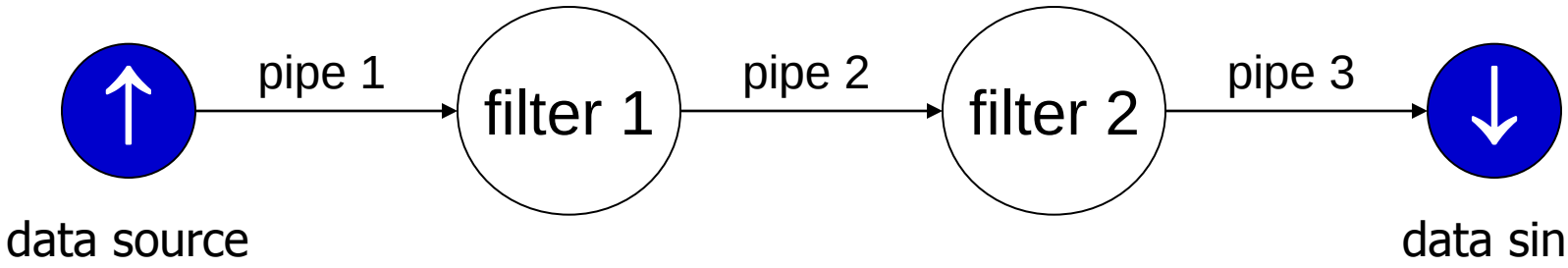


- 顺序批处理风格

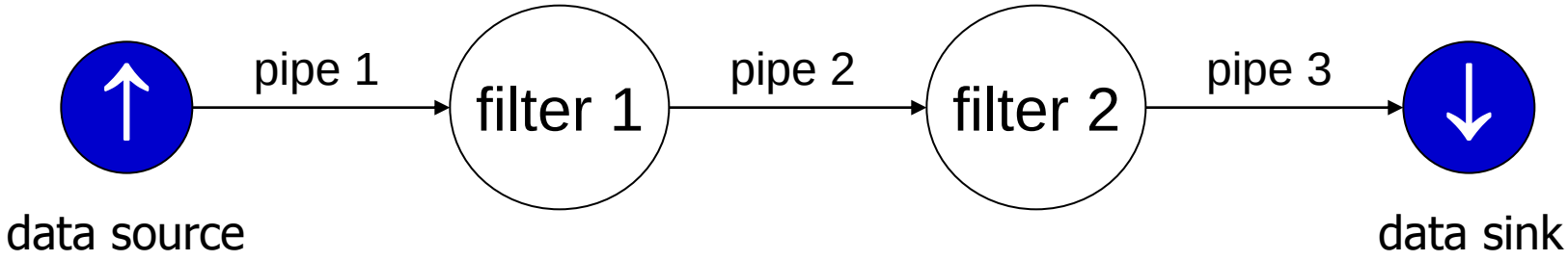


1.软件中的数据流风格

- 语境：数据源源不断的产生，系统需要对这些数据进行若干处理(分析、计算、转换等)。
- 解决方案：
  - 把系统分解为几个序贯的处理步骤，这些步骤之间通过数据流连接，一个步骤的输出是另一个步骤的输入；
  - 每个处理步骤由一个过滤器构件(Filter)实现；
  - 处理步骤之间的数据传输由管道(Pipe)负责。
- 每个处理步骤(过滤器)都有一组输入和输出，过滤器从管道中读取输入的数据流，经过内部处理，然后产生输出数据流并写入管道中。



1.软件中的数据流风格



filter1 {

Read data *d* from pipe1;

Deal with *d* and transform it to *d'*;

Write *d'* to pipe2;

}

filter2 {

Read data *d'* from pipe2;

Deal with *d'* and transform it to *d''*;

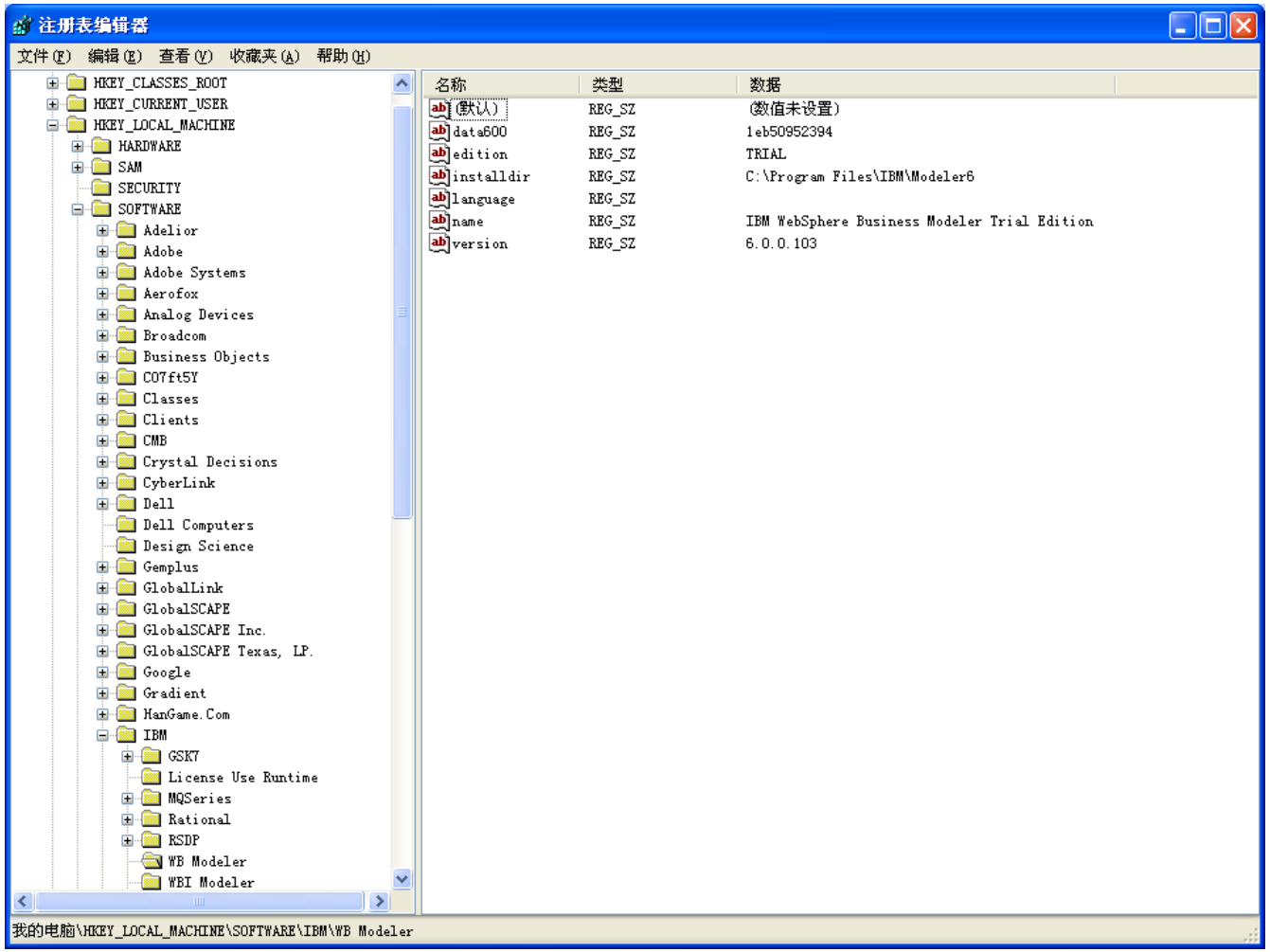
Write *d''* to pipe3;

}

现实中的典型应用领域：

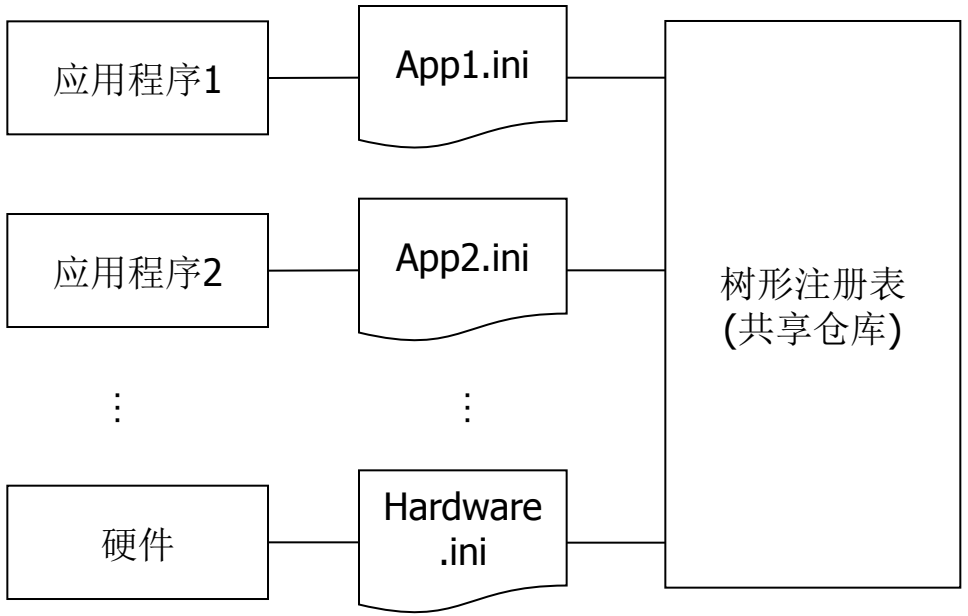
编译器、Unix管道、图像处理、信号处理、网络监控与管理等

2.以数据为中心的风格（仓库）——例1：注册表



注册表的结构

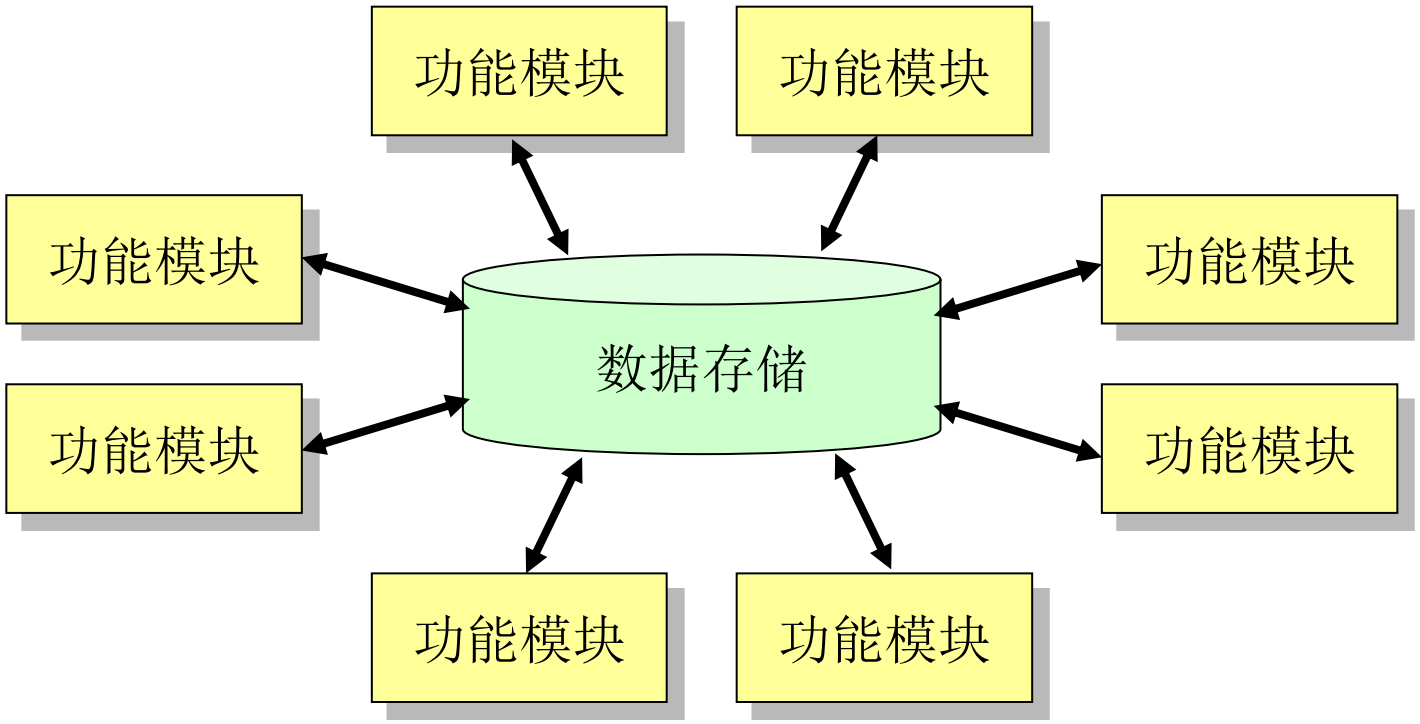
- 最初，硬件/软件系统的配置信息均被各自保存在一个配置文件中(.ini)；
- 这些文件散落在系统的各个角落，很难对其进行维护；
- 为此，引入注册表的思想，将所有.ini文件集中起来，形成共享仓库，为系统运行起到了集中的资源配置管理和控制调度的作用。



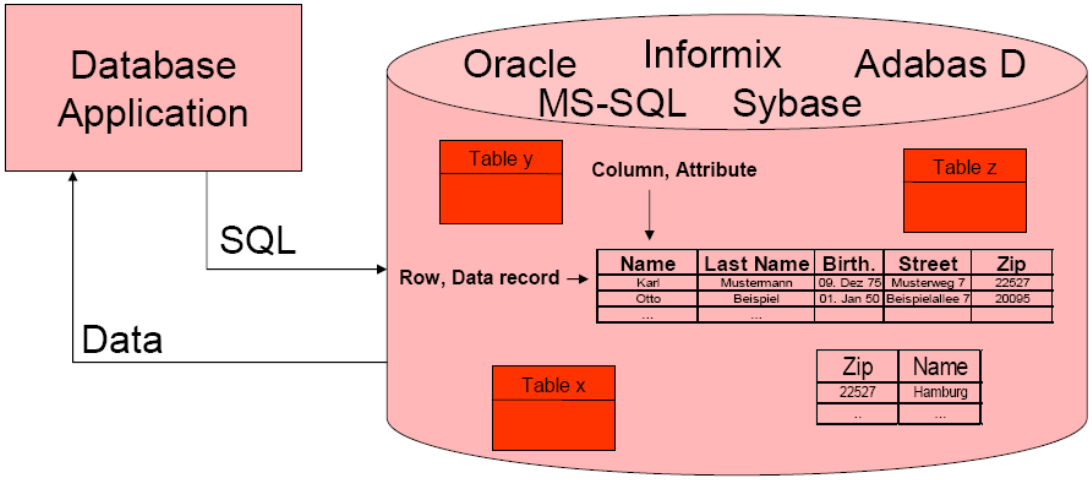
- 注册表中存在着系统的所有硬件和软件配置信息，如启动信息、用户、BIOS、各类硬件、网络、INI文件、驱动程序、应用程序等；
- 注册表信息影响或控制系统/应用软件的行为，应用软件安装/运行/卸载时对其进行添加/修改/删除信息，以达到改变系统功能和控制软件运行的目的。

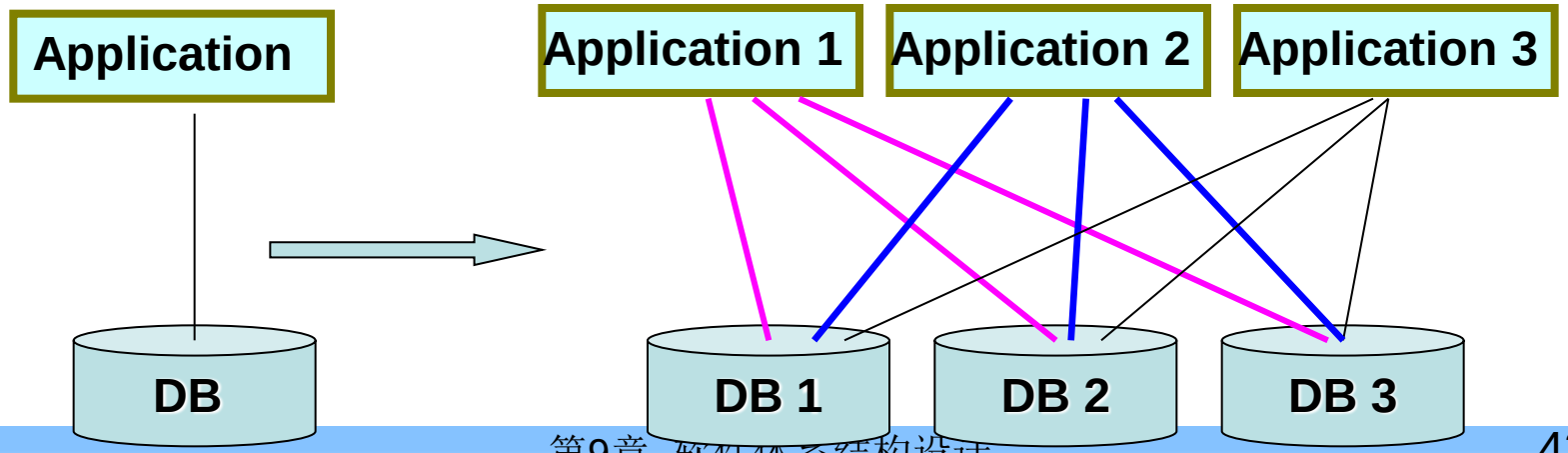
2.以数据为中心的体系结构风格

- **以数据为中心的体系结构风格(也称仓库风格)**



示例1：基于数据库的系统结构

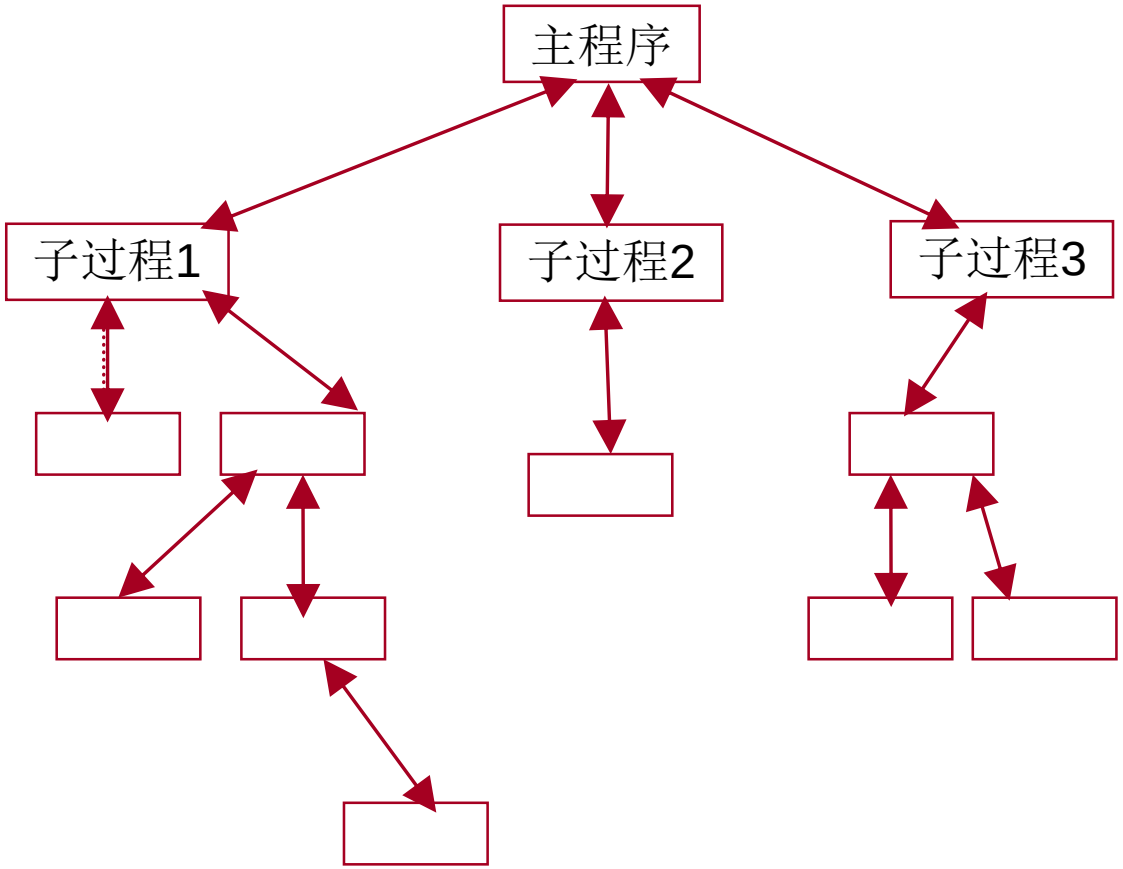




3.主程序-子过程

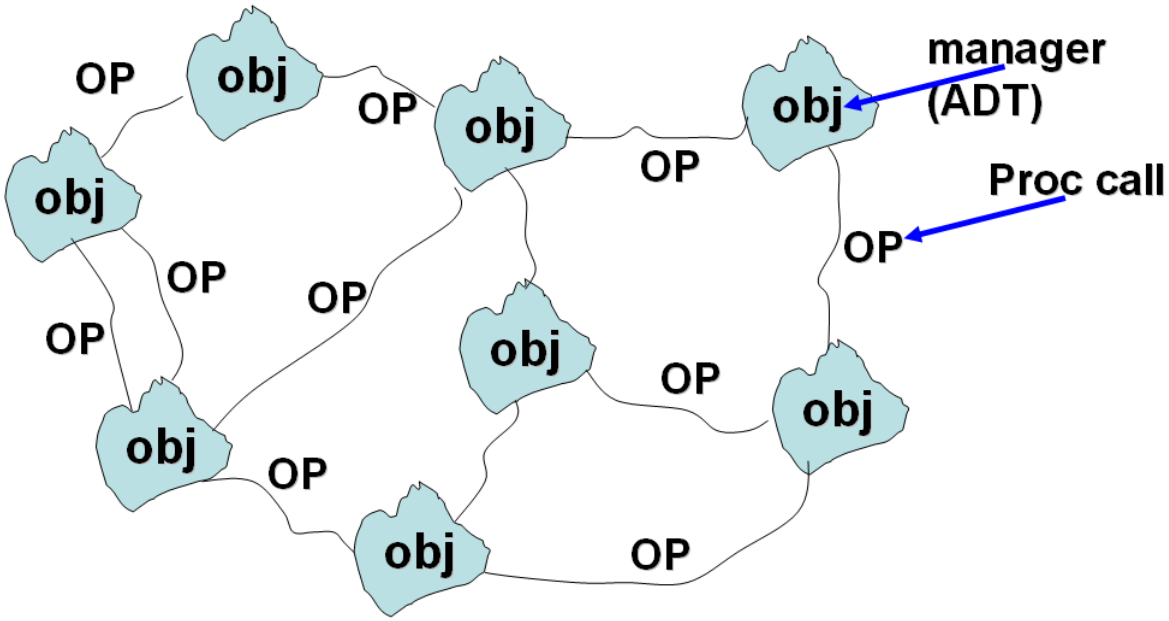
- **该风格是结构化程序设计的一种典型风格，从功能的观点设计系统，通过逐步分解和逐步细化，得到系统体系结构。**
  - 构件：主程序、子程序
  - 连接器：调用-返回机制
  - 拓扑结构：层次化结构

- **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。**



4. 面向对象风格

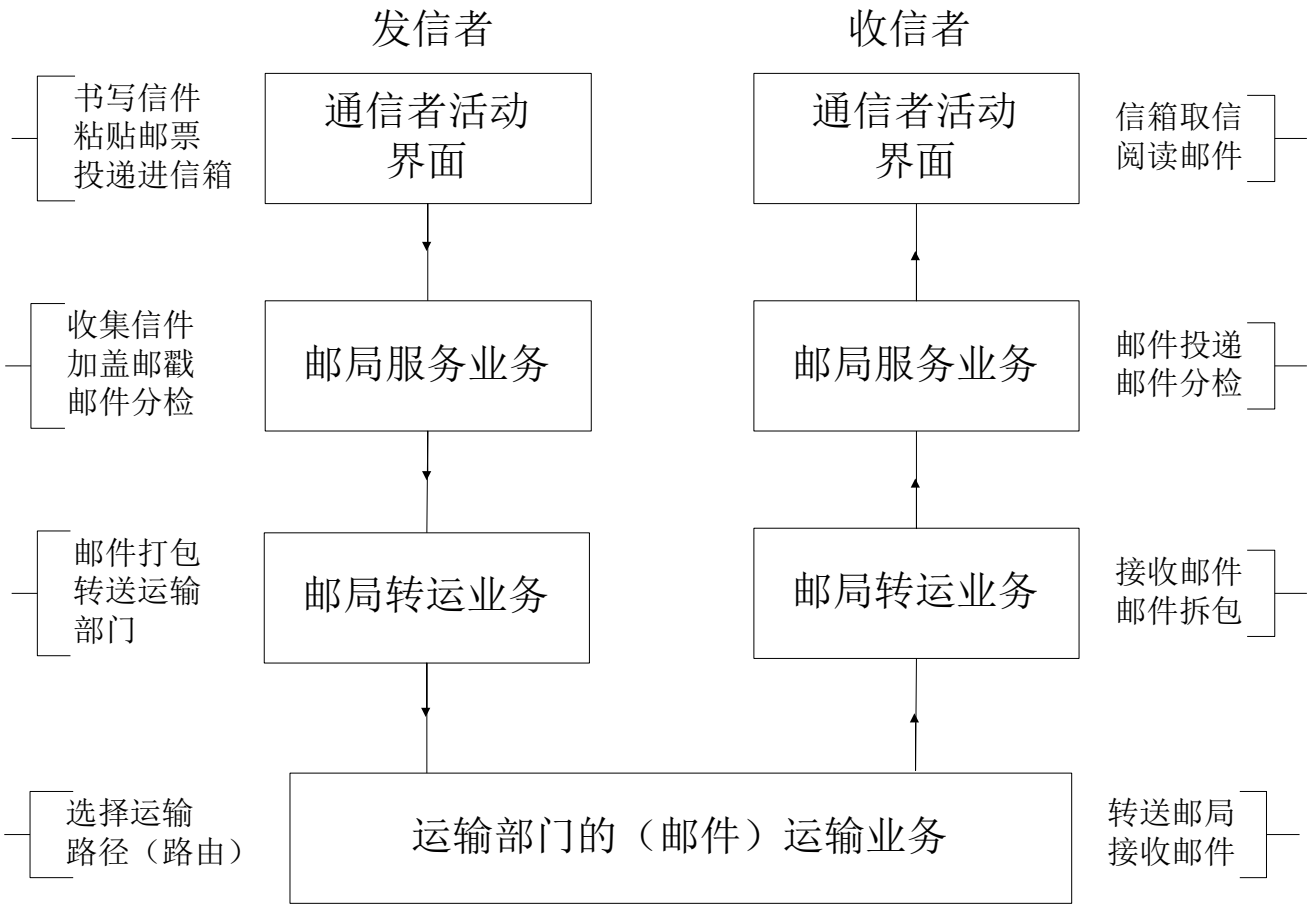
- 系统被看作对象的集合，每个对象都有一个它自己的功能集合；
- 数据及作用在数据上的操作被封装成抽象数据类型(ADT)；
- 只通过接口与外界交互，内部的设计决策则被封装起来
  - 构件：类和对象
  - 连接件：对象之间通过函数调用、消息传递实现交互



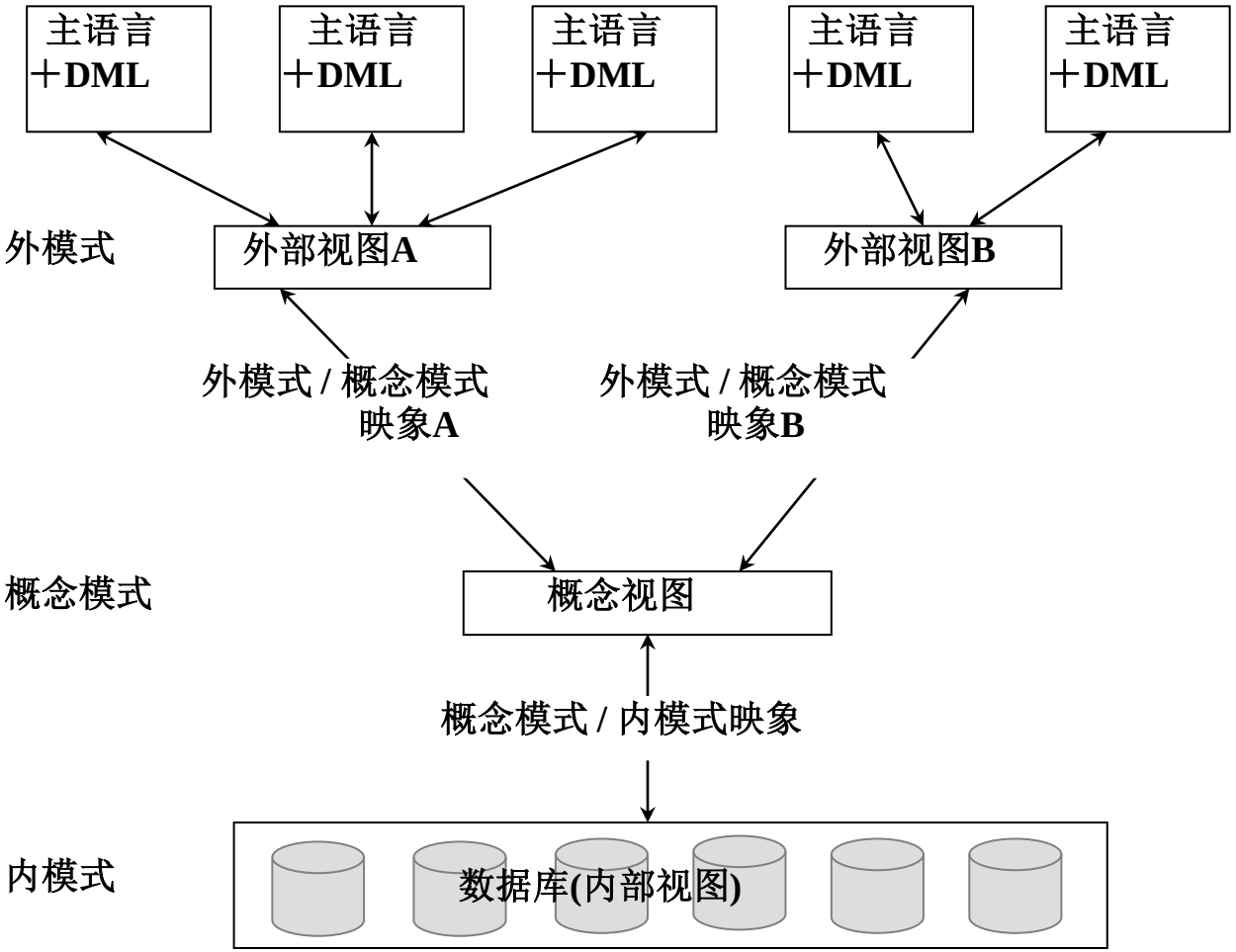
5. 层次结构

- **高楼大厦必须从底层开始建设；**
- **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。

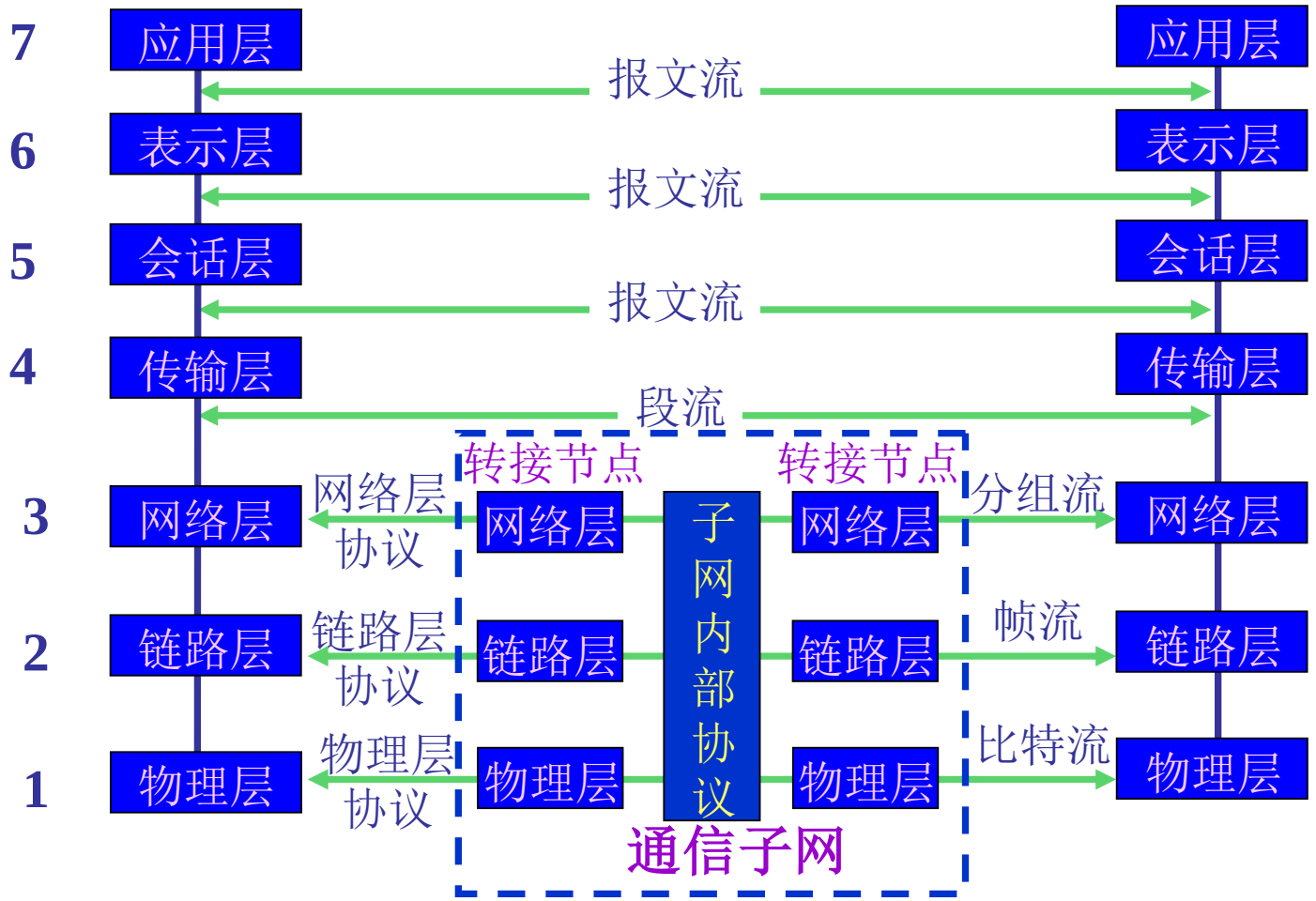
现实世界里邮政系统


发信者
收信者
书写信件
粘贴邮票
投递进信箱
通信者活动界面
通信者活动界面
信箱取信
阅读邮件
收集信件
加盖邮戳
邮件分检
邮局服务业务
邮局服务业务
邮件投递
邮件分检
邮件打包
转送运输部门
邮局转运业务
邮局转运业务
接收邮件
邮件拆包
选择运输路径（路由）
运输部门的（邮件）运输业务
转送邮局
接收邮件

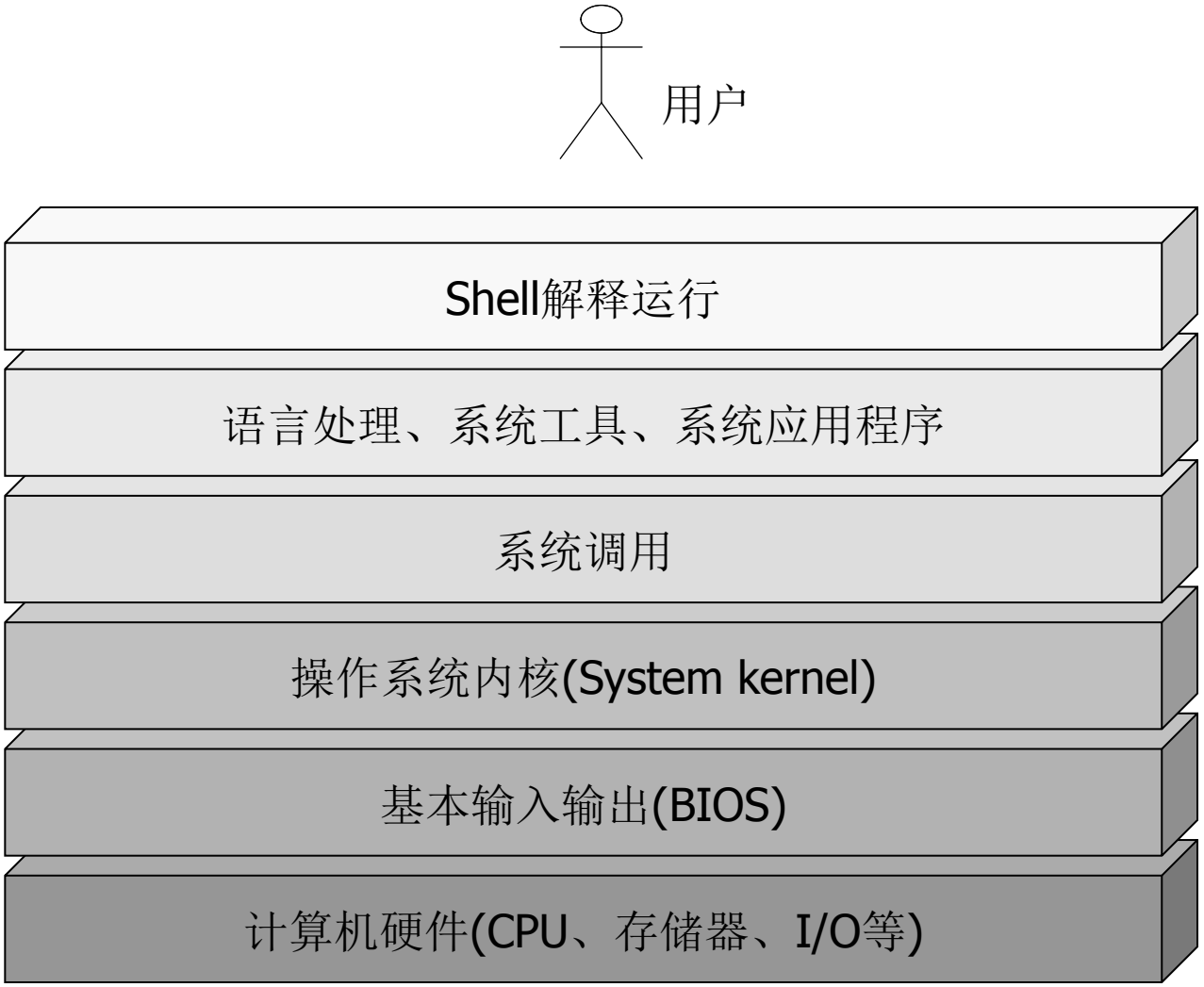
DBMS中的“三级模式-两层映像”



网络的分层模型

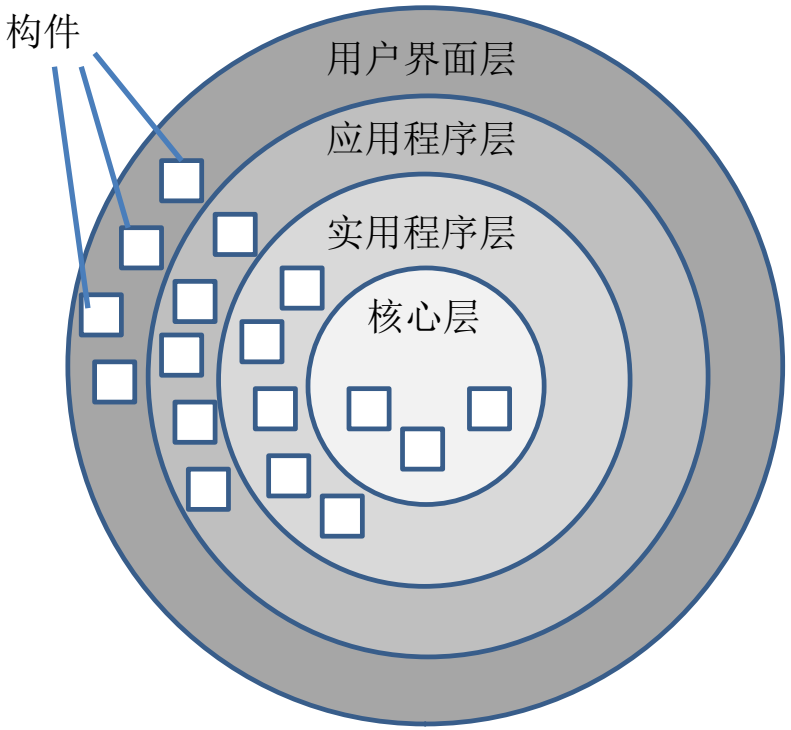

7
应用层
报文流
应用层
6
表示层
报文流
表示层
5
会话层
报文流
会话层
4
传输层
段流
传输层
转接节点
转接节点
3
网络层
网络层协议
网络层
子网内部协议
网络层
分组流
网络层
2
链路层
链路层协议
链路层
链路层
帧流
链路层
1
物理层
物理层协议
物理层
物理层
比特流
物理层
通信子网

计算机操作系统的层次结构



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

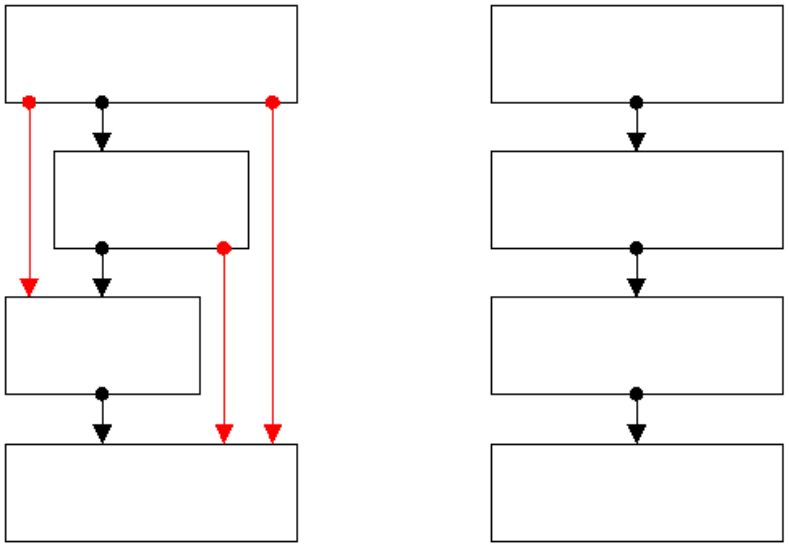


层次系统的优点

- 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。

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

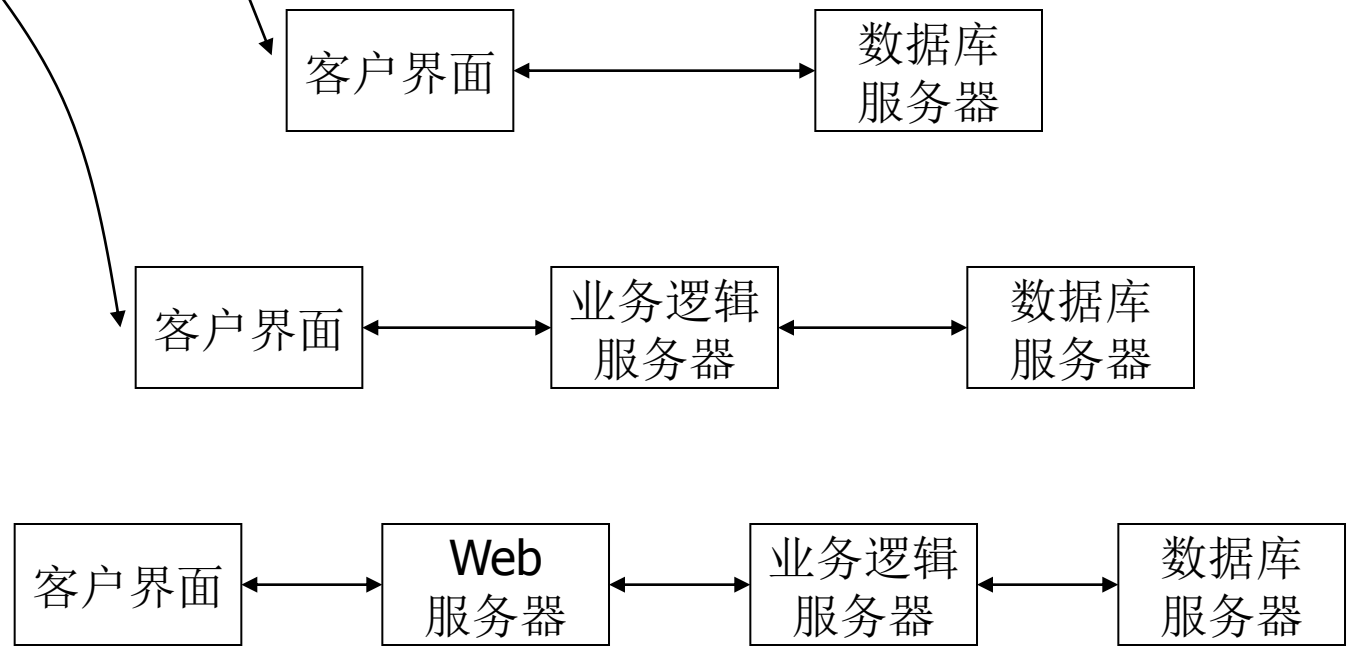


6. “客户机-服务器”体系结构

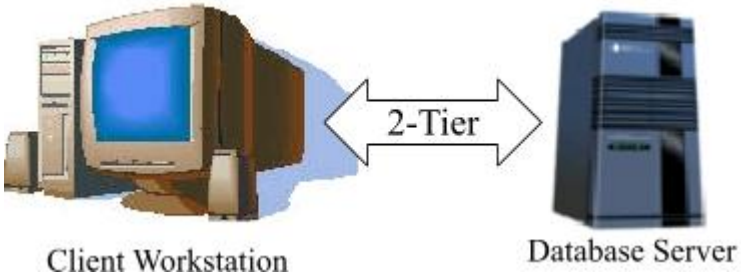
- **客户机/服务器：一个应用系统被分为两个逻辑上分离的部分，每一部分充当不同的角色、完成不同的功能，多台计算机共同完成统一的任务。**
  - 客户机(前端，front-end)：业务逻辑、与服务器通讯的接口；
  - 服务器(后端：back-end)：与客户机通讯的接口、业务逻辑、数据管理。
- **一般的，**
  - 客户机为完成特定的工作向服务器发出请求；
  - 服务器处理客户机的请求并返回结果。

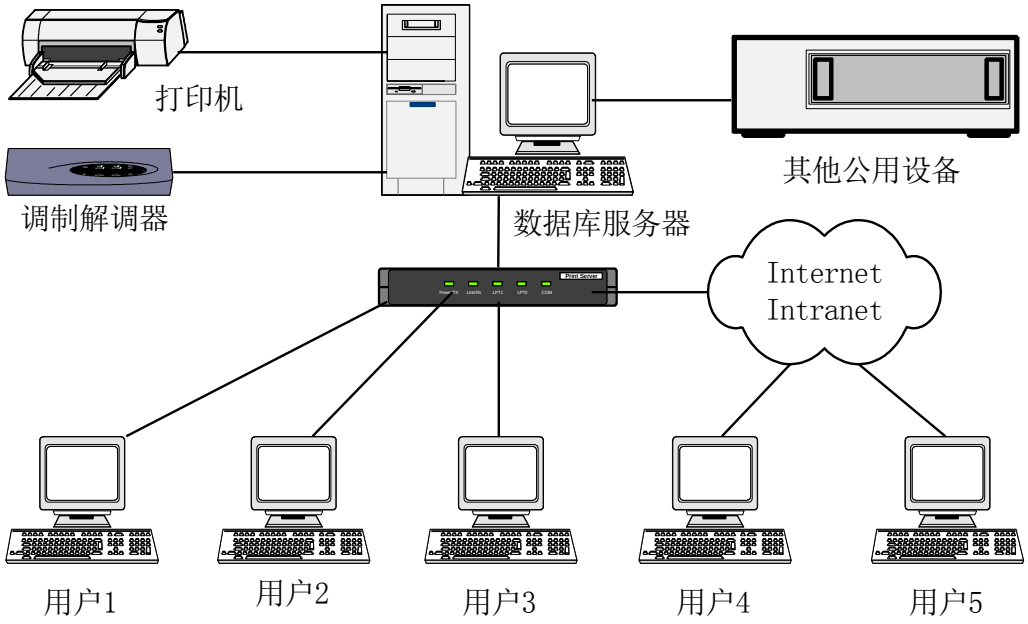
6. 客户机/服务器的层次性

- **“客户机-服务器”结构的发展历程：**
  - 两层C/S（仓库体系风格）
  - 三层C/S
  - 多层C/S

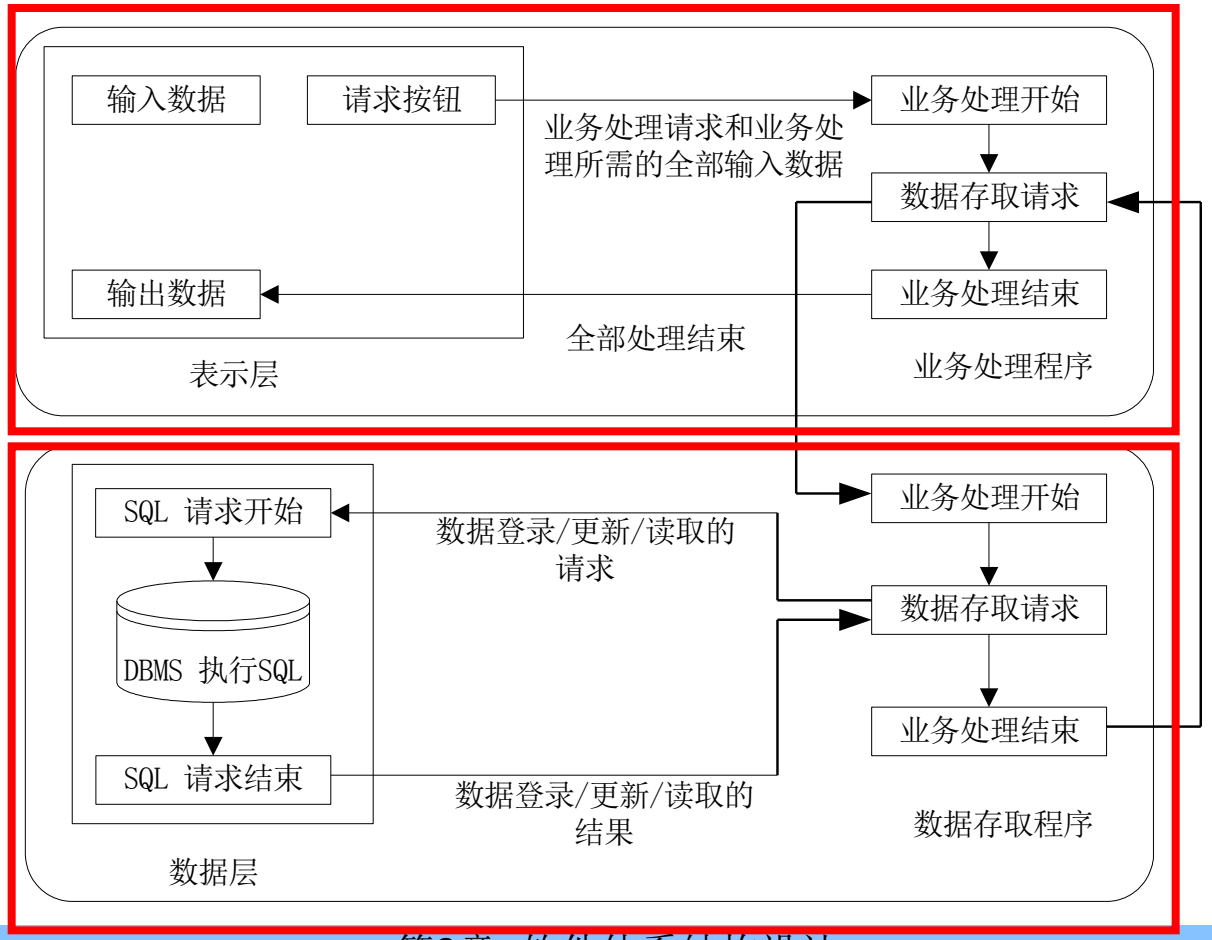


两层C/S结构


2-Tier
Client Workstation
Database Server

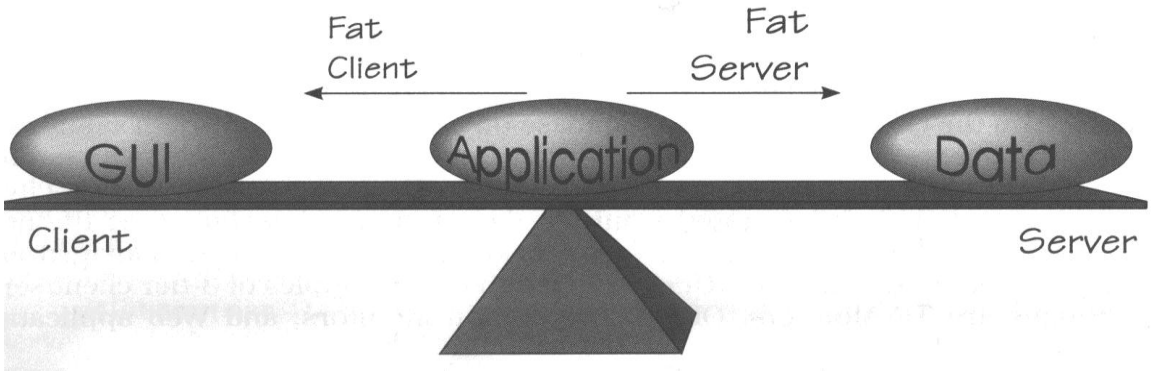

打印机
调制解调器
数据库服务器
其他公用设备
Internet
Intranet
用户1
用户2
用户3
用户4
用户5

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

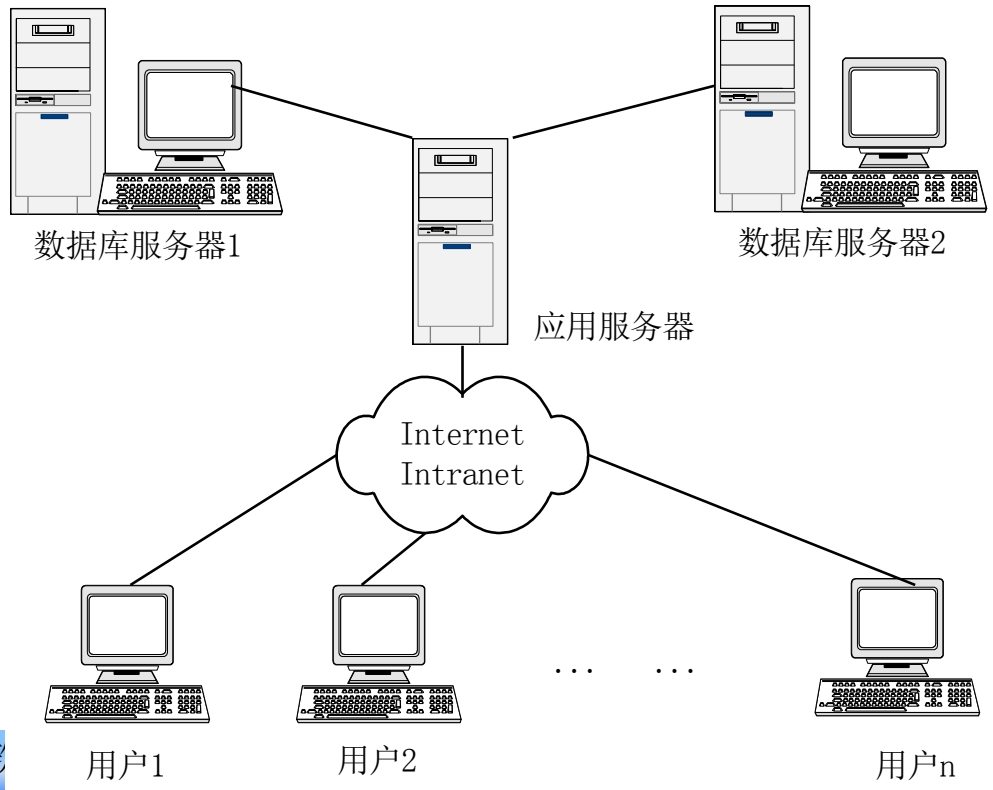
胖客户端与瘦客户端

- **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

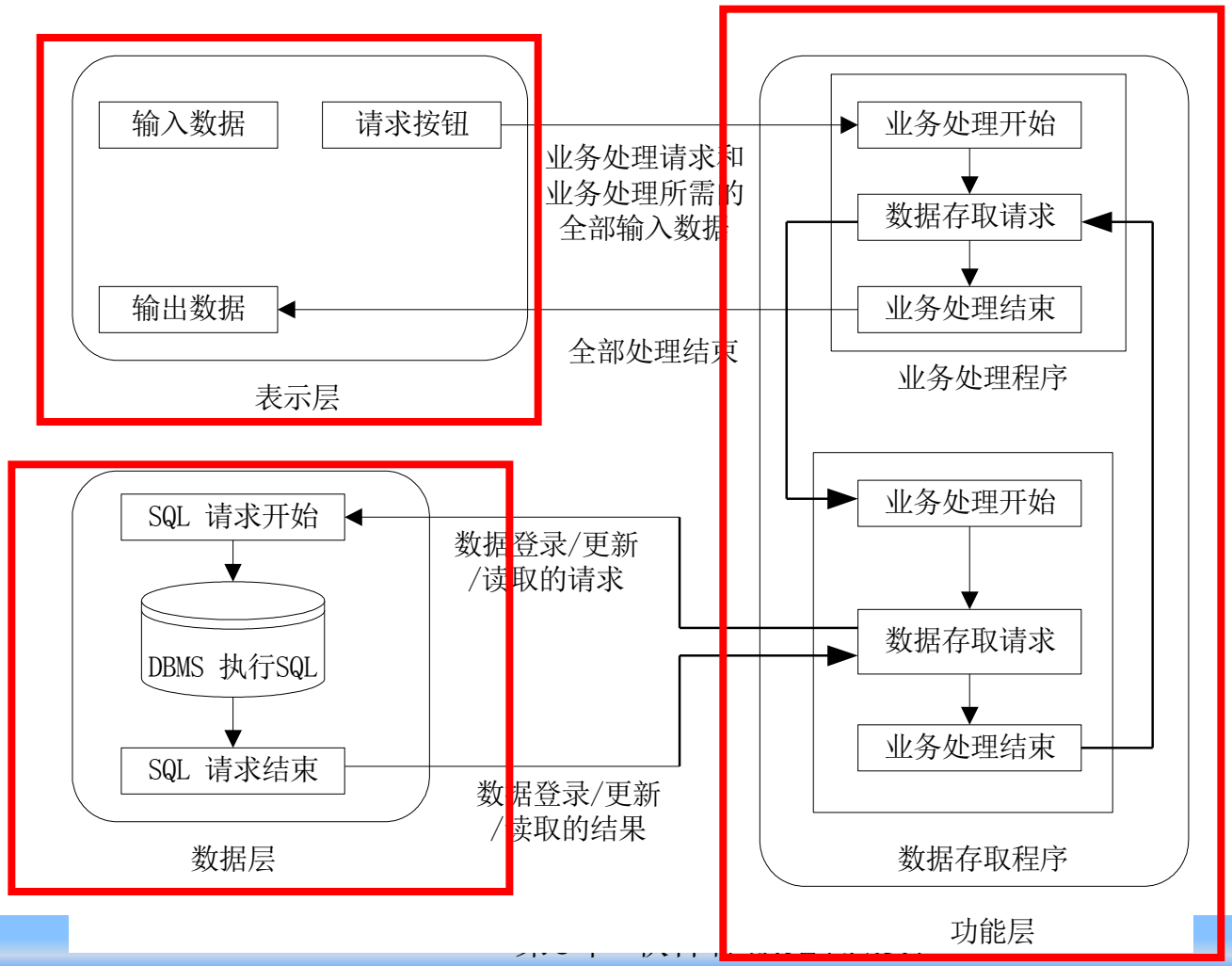


三层C/S体系结构

- **在客户端与数据库服务器之间增加了一个中间层**
  - 接口层：用户界面—表单
  - 应用逻辑层：业务逻辑—控制和连接
  - 存储层：数据库—存储层

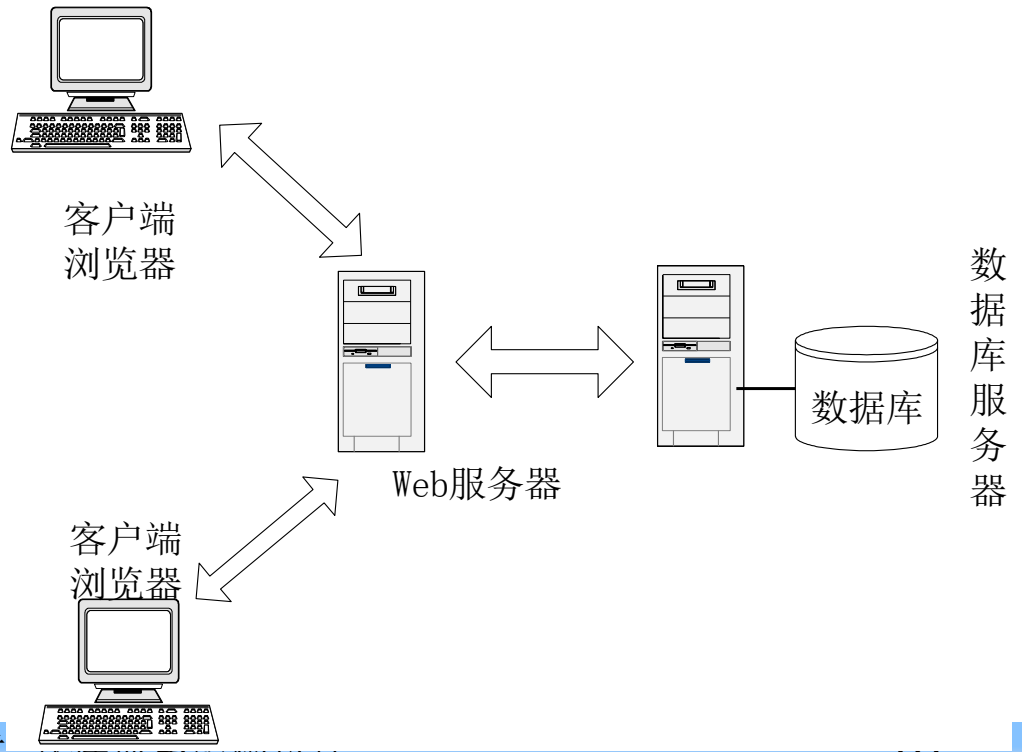


三层C/S结构

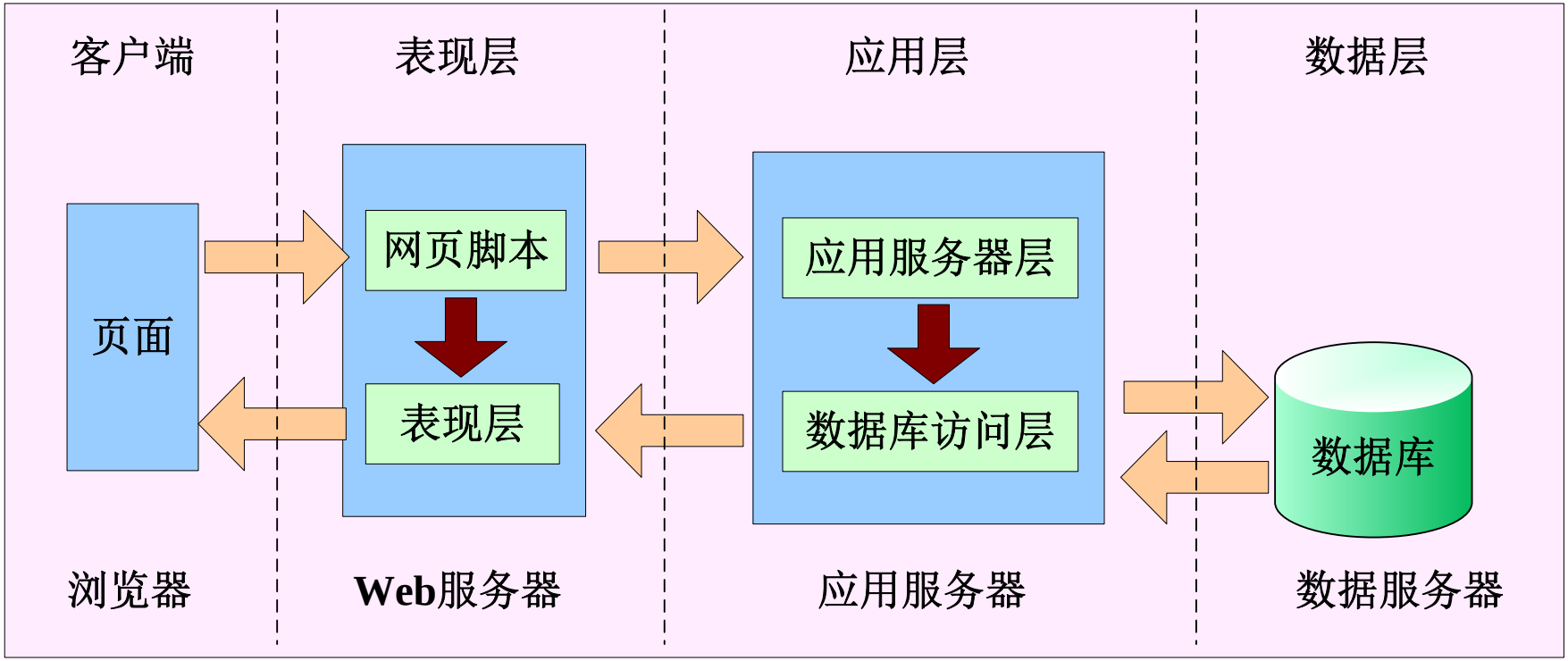

输入数据
请求按钮
输出数据
表示层
业务处理请求和
业务处理所需的
全部输入数据
全部处理结束
业务处理开始
数据存取请求
业务处理结束
业务处理程序
SQL 请求开始
DBMS 执行SQL
SQL 请求结束
数据登录/更新
/读取的请求
数据登录/更新
/读取的结果
数据层
业务处理开始
数据存取请求
业务处理结束
数据存取程序
功能层

B/S结构

- **浏览器/服务器(B/S)是一种四层体系结构**
  - 表示客户层：浏览器
  - 表示服务器层：Web服务器，表单
  - 应用逻辑层：应用服务器，控制和连接
  - 存储层：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。**

7.*模型-视图-控制器（MVC）

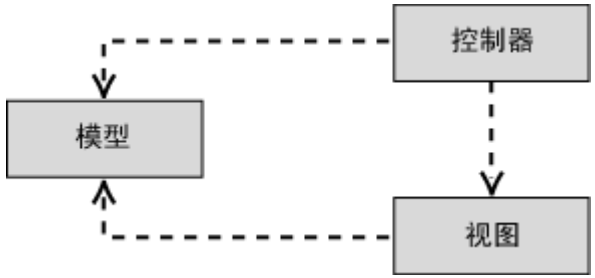
- **问题**
  - 用户界面需要频繁的修改，它是“不稳定”的。
  - 业务逻辑/数据 与 用户界面 之间应尽量少的避免直接通信。

- **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？**

解决方案：Model-View-Controller (MVC)

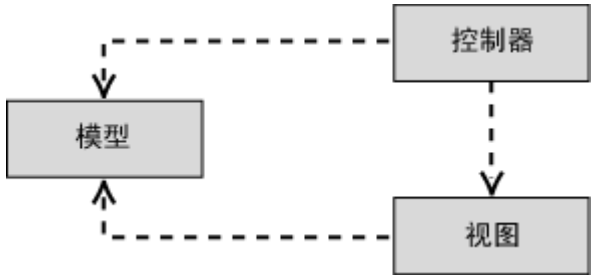
- **MVC是一种软件体系结构，它将应用程序的数据模型/业务逻辑、用户界面分别放在独立的构件中，从而对用户界面的修改不会对数据模型/业务逻辑造成很大影响。**

- **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。**

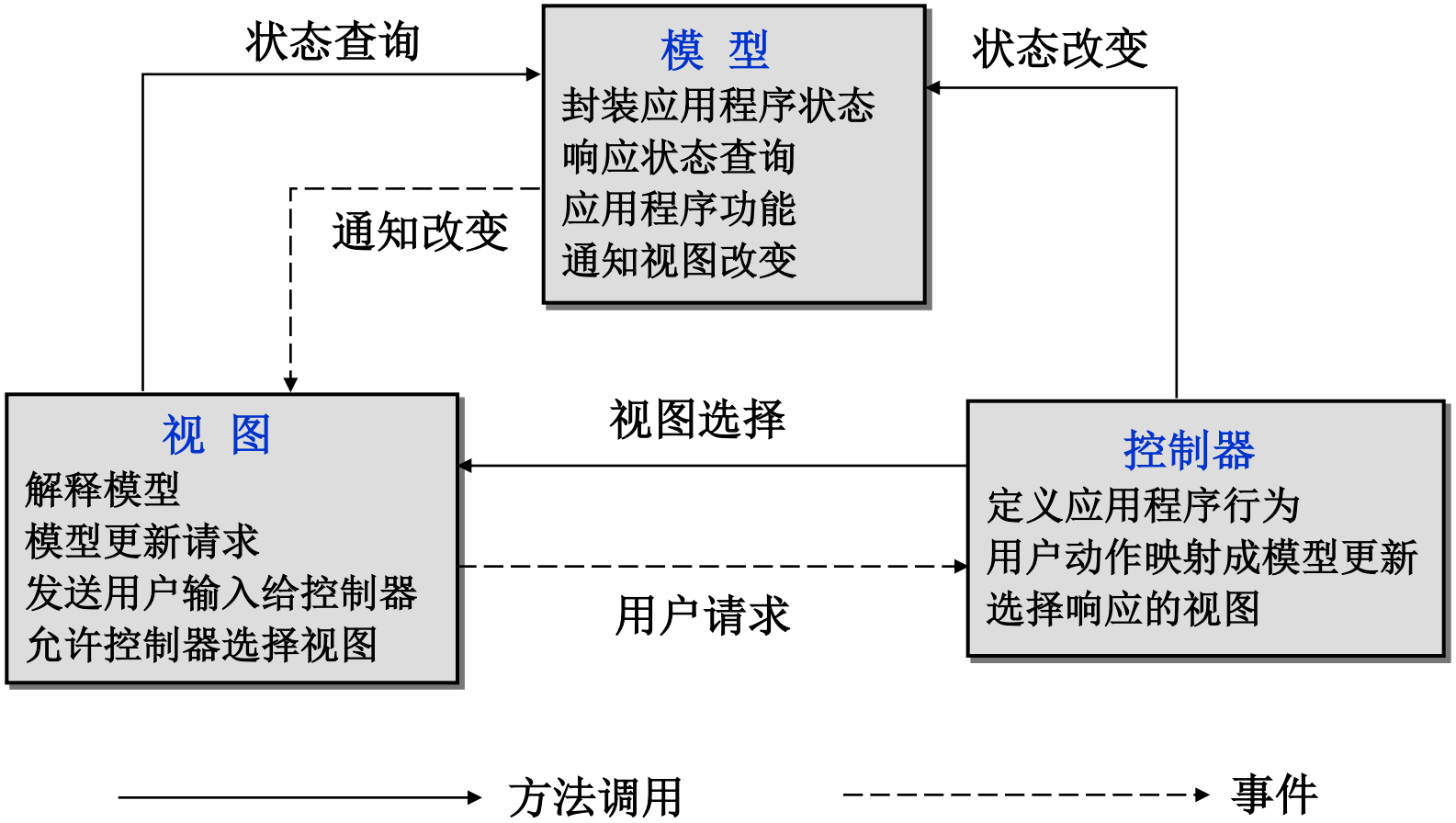


解决方案：Model-View-Controller (MVC)

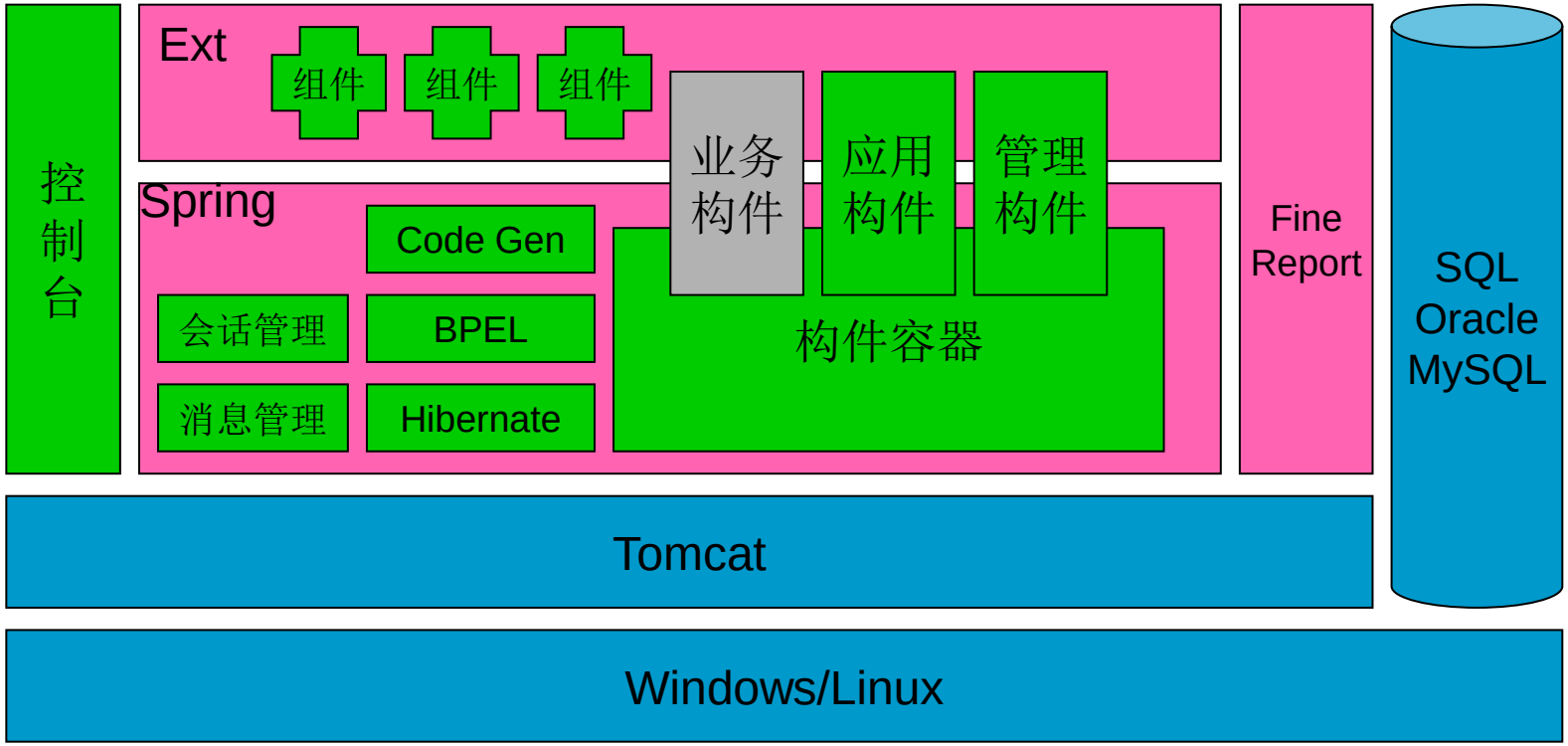
- **Model-View-Controller (MVC)：**
  - 模型(Model, M)：用于管理应用系统的行为和数据，并响应为获取其状态信息(通常来自视图)而发出的请求，还会响应更改状态的指令(通常来自控制器)；——对应于传统B/S中的业务逻辑和数据
  - 视图(View, V)：用于管理数据的显示；——对应于传统B/S中的用户界面
  - 控制器(Controller, C)：用于解释用户的鼠标和键盘输入，以通知模型和视图进行相应的更改。——在传统B/S结构中新增的元素



Model-View-Controller (MVC)

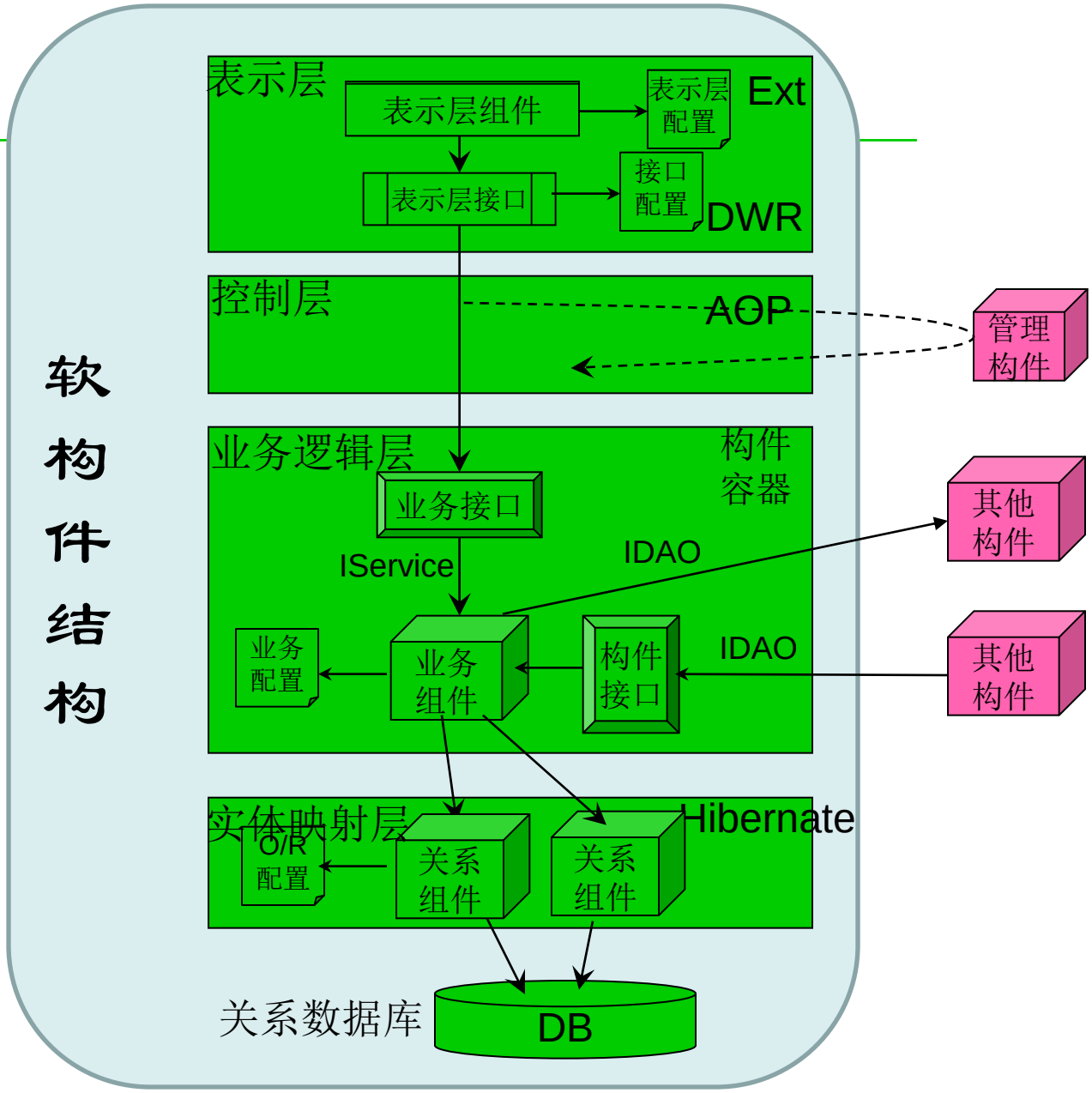


实例：慧通软件开发框架


控制台
Ext
组件
组件
组件
业务构件
应用构件
管理构件
Spring
Code Gen
会话管理
BPEL
消息管理
Hibernate
构件容器
Fine Report
SQL
Oracle
MySQL
Tomcat
Windows/Linux

实例：

慧通软件体系结构


软构件结构
表示层
表示层组件
表示层配置
Ext
表示层接口
接口配置
DWR
控制层
AOP
管理构件
业务逻辑层
构件容器
业务接口
IService
IDAO
其他构件
业务配置
业务组件
构件接口
IDAO
其他构件
实体映射层
Hibernate
O/R配置
关系组件
关系组件
关系数据库
DB

主要内容

- **9.1 软件体系结构的背景与定义**
- **§9.2 软件体系结构的基本概念**
- **§9.3 软件体系结构风格**
- **§9.4 体系结构设计**

面向对象的设计的两个阶段

- **系统设计(System Design)**
  - 相当于概要设计(即设计系统的体系结构)；
  - 选择解决问题的基本途径；
  - 决策整个系统的结构与风格；
- **对象设计(Object Design)**
  - 相当于详细设计(即设计对象内部的具体实现)；
  - 细化需求分析模型和系统体系结构设计模型；
  - 识别新的对象；
  - 在系统所需的应用对象与可复用的商业构件之间建立关联；
    - 识别系统中的应用对象；
    - 调整已有的构件；
    - 给出每个子系统/类的精确规格说明。

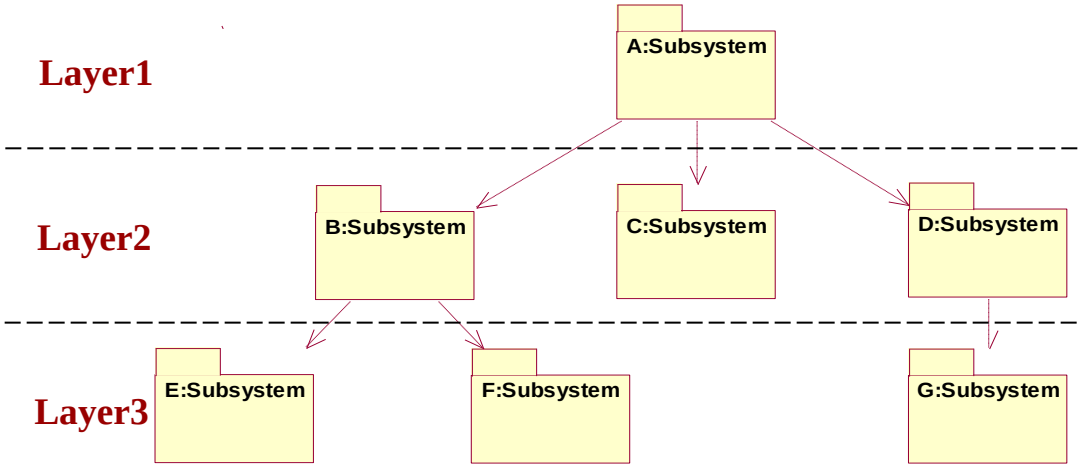
1.系统设计概述

- **设计系统的体系结构**
  - 选择合适的分层体系结构策略，建立系统的总体结构：分几层？每层的功能分别是什么？
- **识别设计元素**
  - 识别“设计类”(design class)、“包”(package)、“子系统”(sub-system)
- **部署子系统**
  - 选择硬件配置和系统平台，将子系统分配到相应的物理节点，绘制部署图(deployment diagram)
- **定义数据的存储策略**
- **检查系统设计**

设计系统的体系结构

- 以数据为中心的风格(仓库)
- 数据流风格
- 主程序-子过程
- 面向对象风格
- 层次风格
- C/S和B/S结构
- 事件风格
- “模型-视图-控制器”(MVC)

- OOD通常采用层次化体系结构风格，将系统分解为：
  - 不超过7±2个子系统；
  - 不超过5±2个层次；

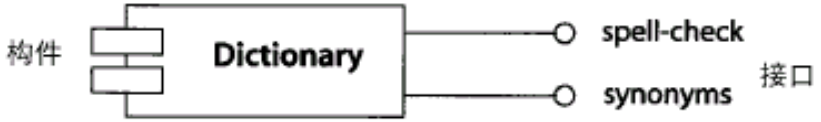


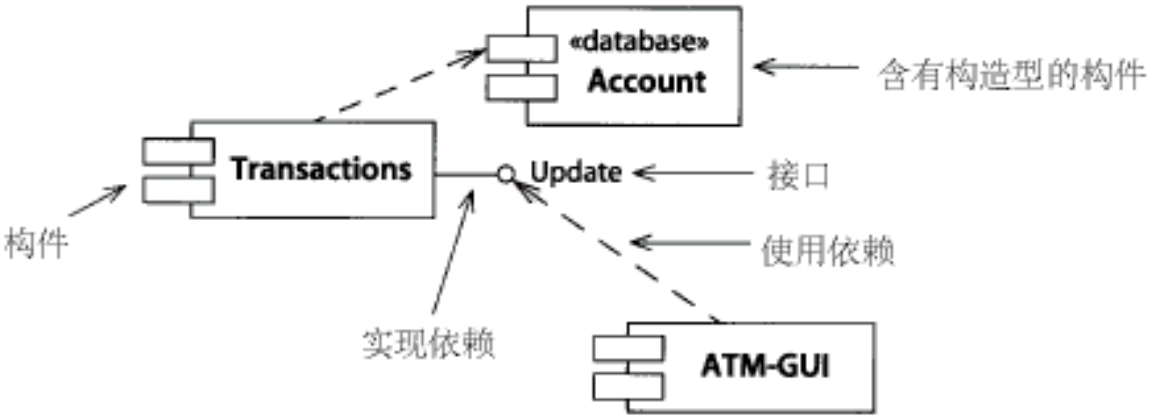
OOD——物理视图模型

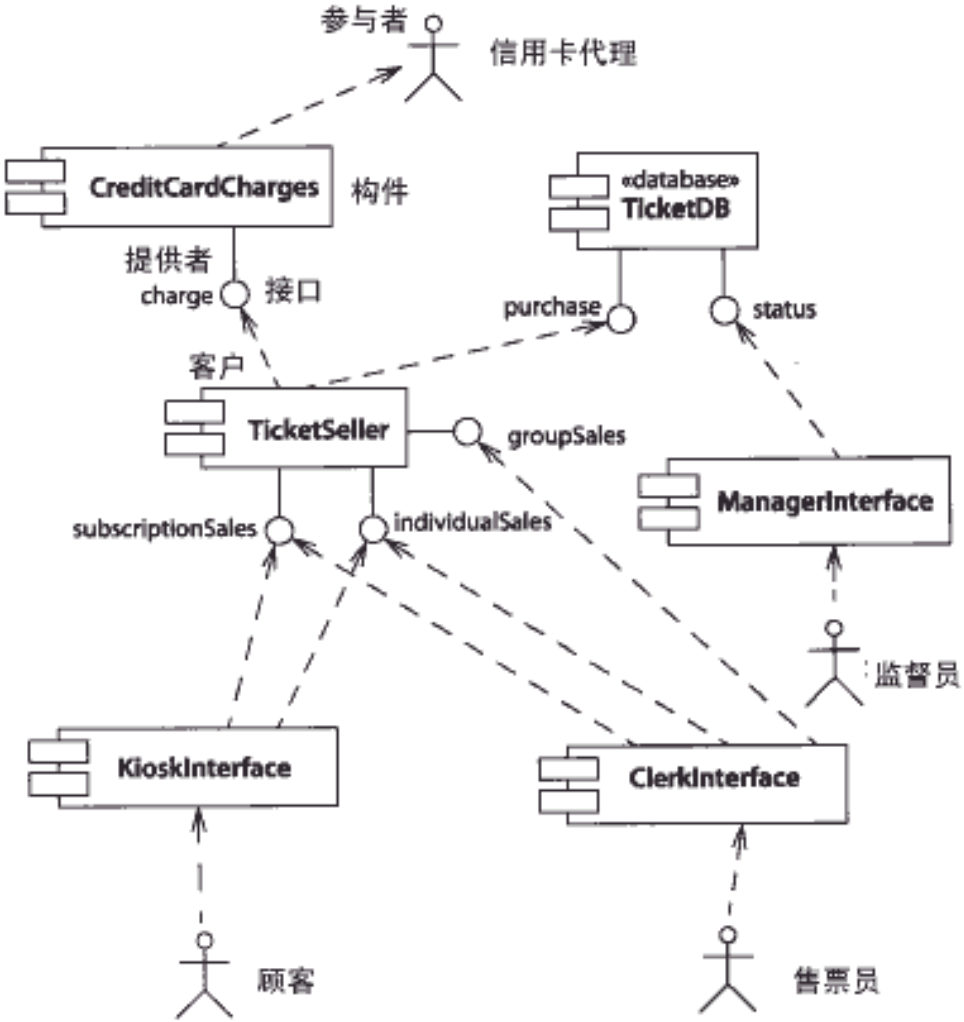
- **物理视图对应用自身的实现结构建模，例如系统的构件组织和建立在运行节点上的配置。**
- **这类视图提供了将系统中的类映射成物理构件和节点的机制。**
- **物理视图有两种：实现视图和部署视图。**
  - 实现视图为系统的构件建模型—构件即构造应用的软件单元—还包括各构件之间的依赖关系，以便通过这些依赖关系来估计对系统构件的修改给系统可能带来的影响。实现视图用构件图来表现。
  - 部署视图描述位于节点实例上的运行构件实例的安排。节点是一组运行资源，如计算机、设备或存储器。这个视图允许评估分配结果和资源分配。部署视图用部署图来表达。

OOD——构件图

- 将系统中可重用的块包装成具有可替代性的物理单元，这些单元被称为构件。
- 构件是定义了良好接口的物理实现单元，它是系统中可替换的部分
- 每个构件体现了系统设计中特定类的实现。
- 良好定义的构件不直接依赖于其他构件而依赖于构件所支持的接口。

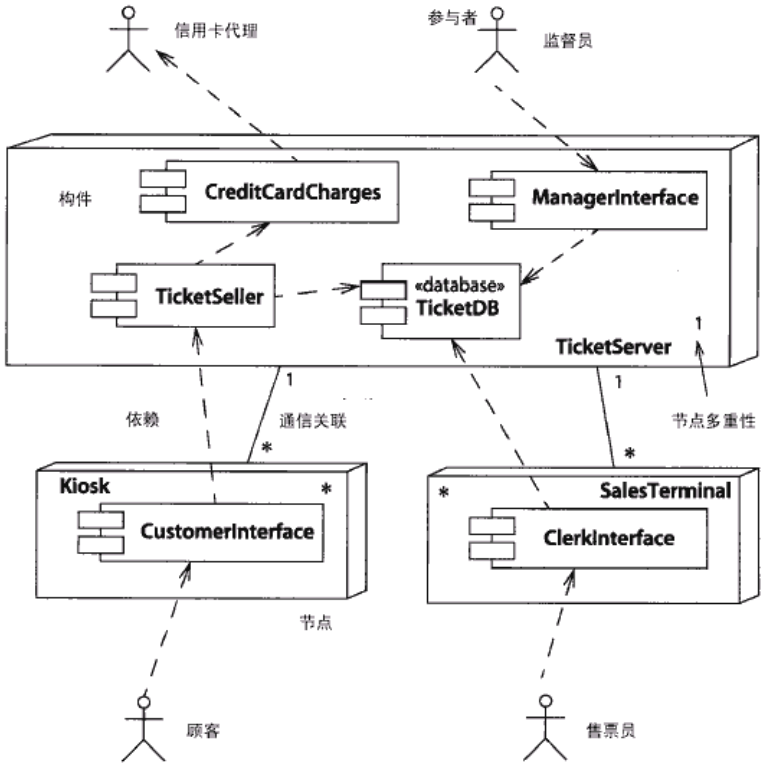


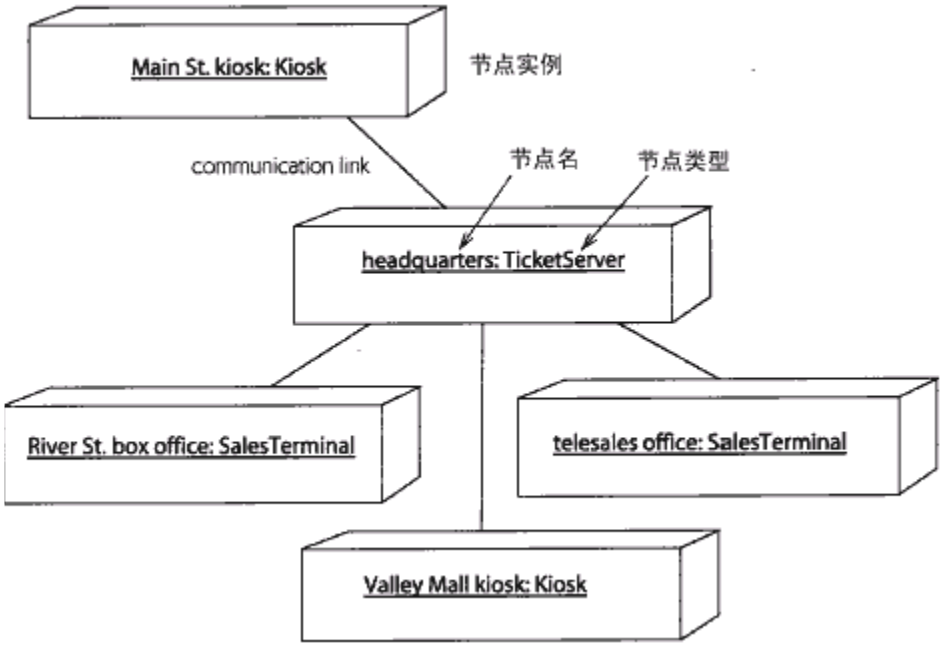



参与者
信用卡代理
CreditCardCharges
构件
«database»
TicketDB
提供者
charge
接口
purchase
status
客户
TicketSeller
groupSales
ManagerInterface
subscriptionSales
individualSales
监督员
KioskInterface
ClerkInterface
顾客
售票员

OOD——部署图

- 反映系统硬件的拓扑结构
- 节点是表示计算资源的运行时的物理对象，通常具有内存和处理能力
- 目的：
  - 明确构件的分布
  - 找出性能的瓶颈
- 由架构师、网络工程师和系统工程师开发


信用卡代理
参与者
监督员
构件
CreditCardCharges
ManagerInterface
TicketSeller
«database»
TicketDB
TicketServer
1
依赖
通信关联
*
节点多重性
Kiosk
CustomerInterface
SalesTerminal
ClerkInterface
节点
顾客
售票员

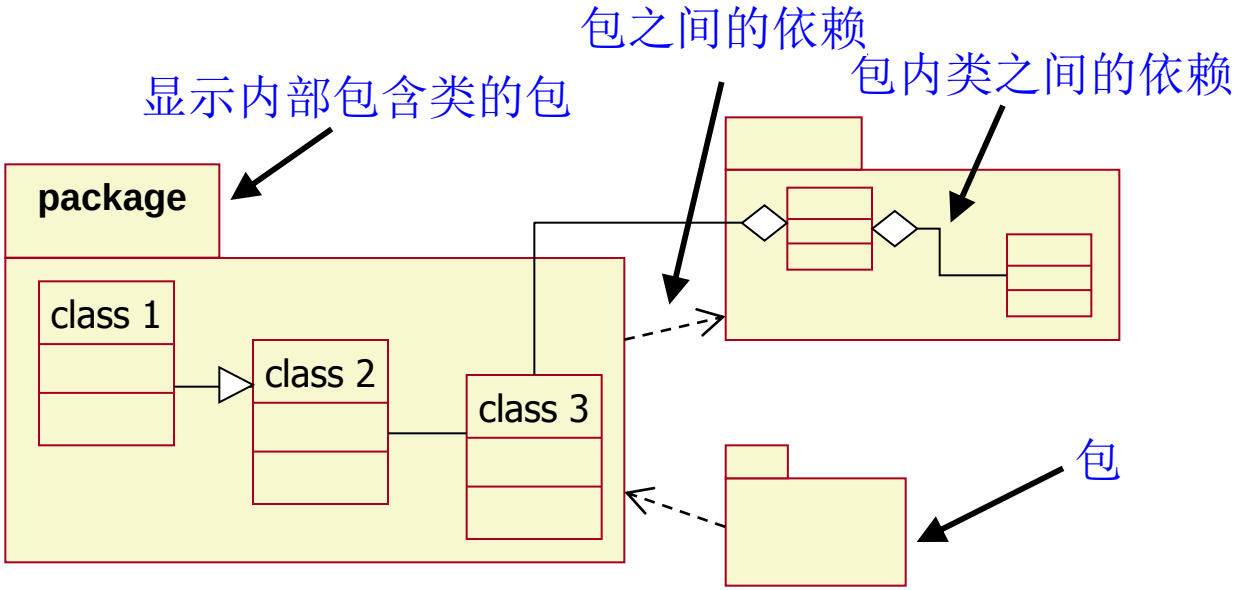

Main St. kiosk: Kiosk
节点实例
communication link
节点名
节点类型
headquarters: TicketServer
River St. box office: SalesTerminal
telesales office: SalesTerminal
Valley Mall kiosk: Kiosk

模型管理视图——包图

- 包图是在 UML 中用类似于文件夹的符号表示的模型元素的组合。系统中的每个元素都只能为一个包所有，一个包可嵌套在另一个包中。使用包图可以将相关元素归入一个系统。一个包中可包含附属包、图表或单个元素。
- 一个"包图"可以是任何一种的UML图组成，通常是UML用例图或UML类图

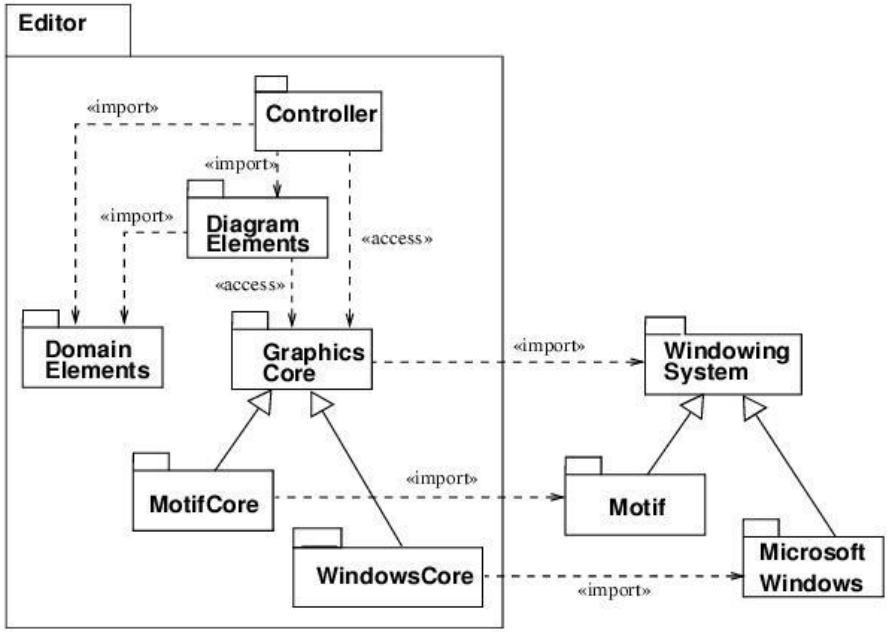
面向对象设计中的基本元素

- **基本单元：设计类(design class)——对应于OOA中的“分析类”**
- **为了系统实现与维护过程中的方便性，将多个设计类按照彼此关联的紧密程度聚合到一起，形成大粒度的“包”(package)；**



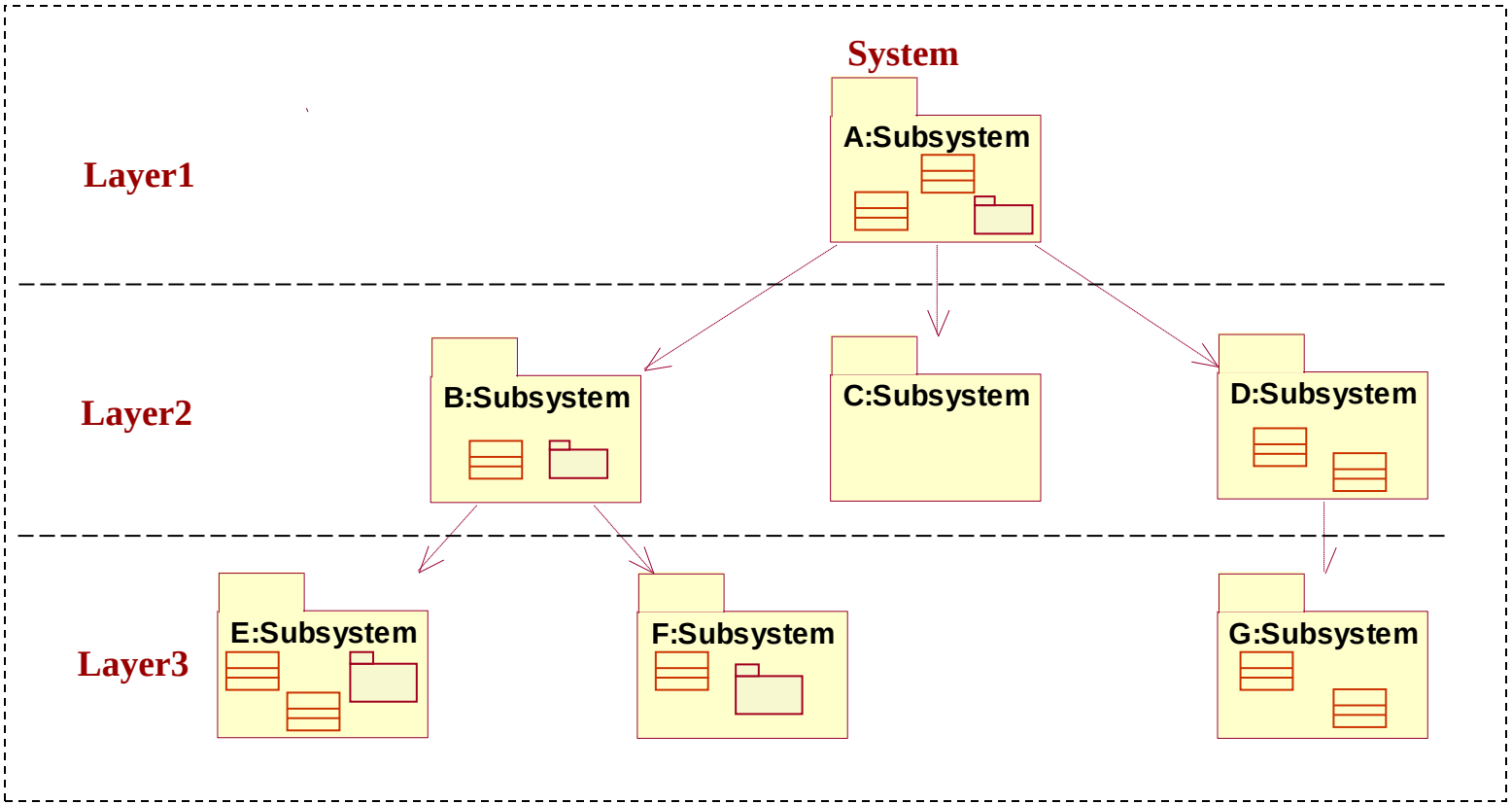
绘制包图(package diagram)

- 对一个复杂的软件系统，要使用大量的设计类，这时就必要把这些类分组进行组织；
- 把在语义上接近且倾向于一起变化的类组织在一起形成“包”，既可控制模型的复杂度，有助于理解，而且也有助于按组来控制类的可见性；
- 结构良好的包是松耦合、高内聚的，而且对其内容的访问具有严密的控制。

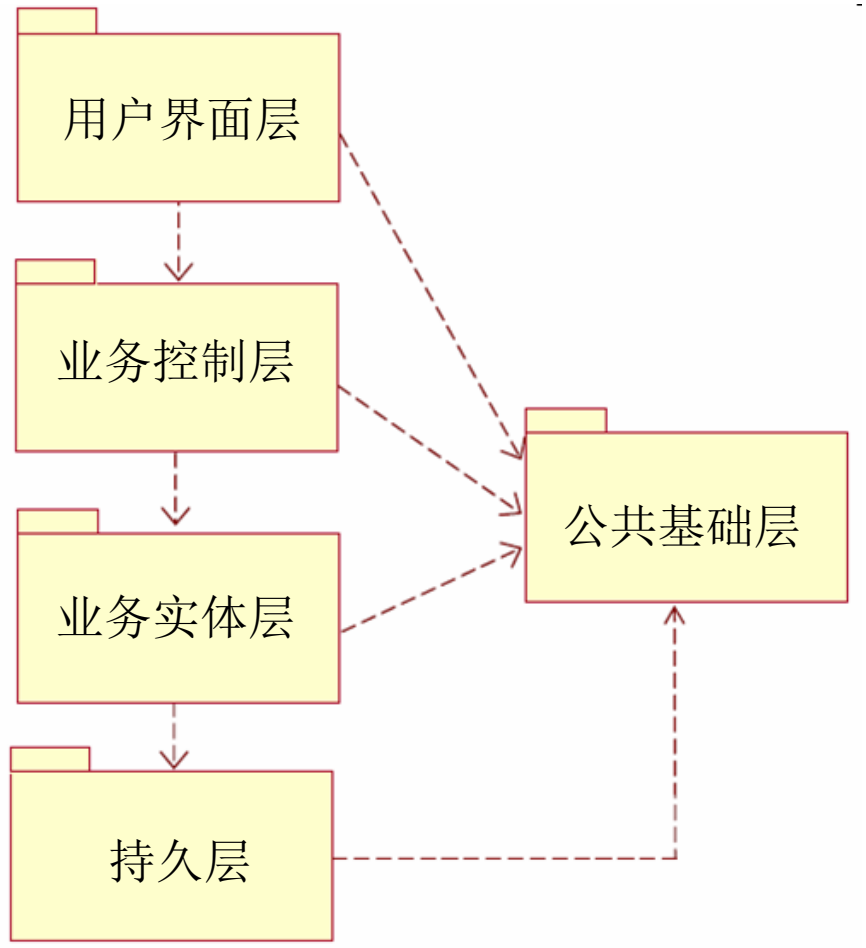


按系统划分包的分层结构

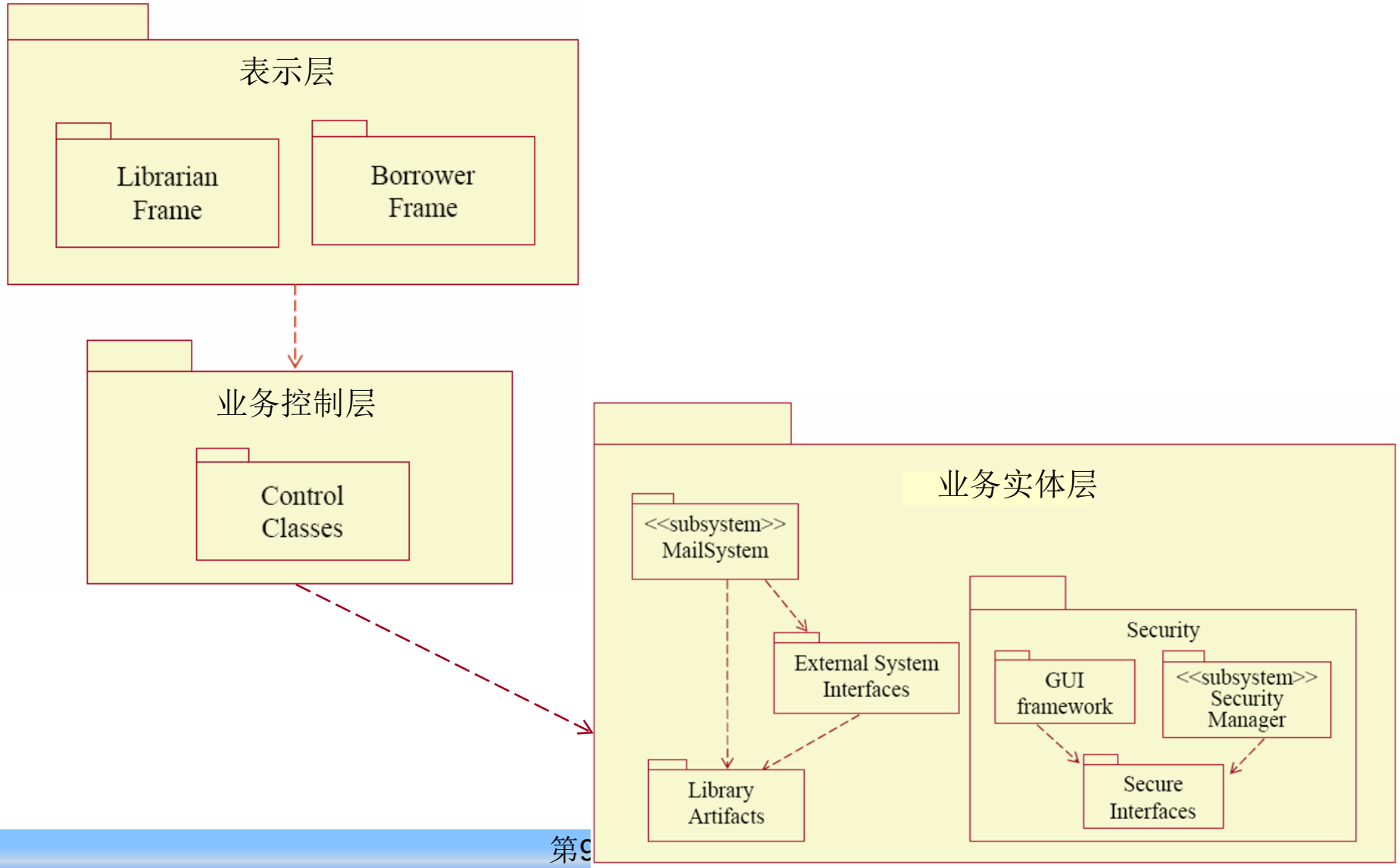
- 一个或多个包聚集在一起，形成“子系统”(sub-system)
- 多个子系统，构成完整的“系统”(system)



按实现技术划分包的分层结构

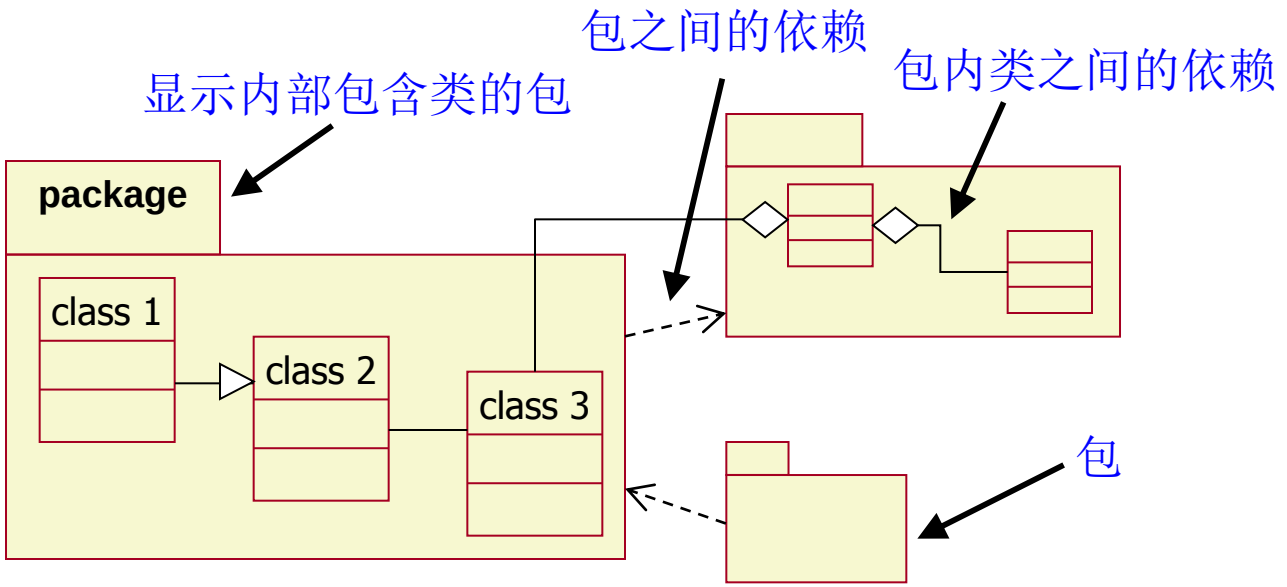


图书管理系统：软件体系结构



包之间的关系

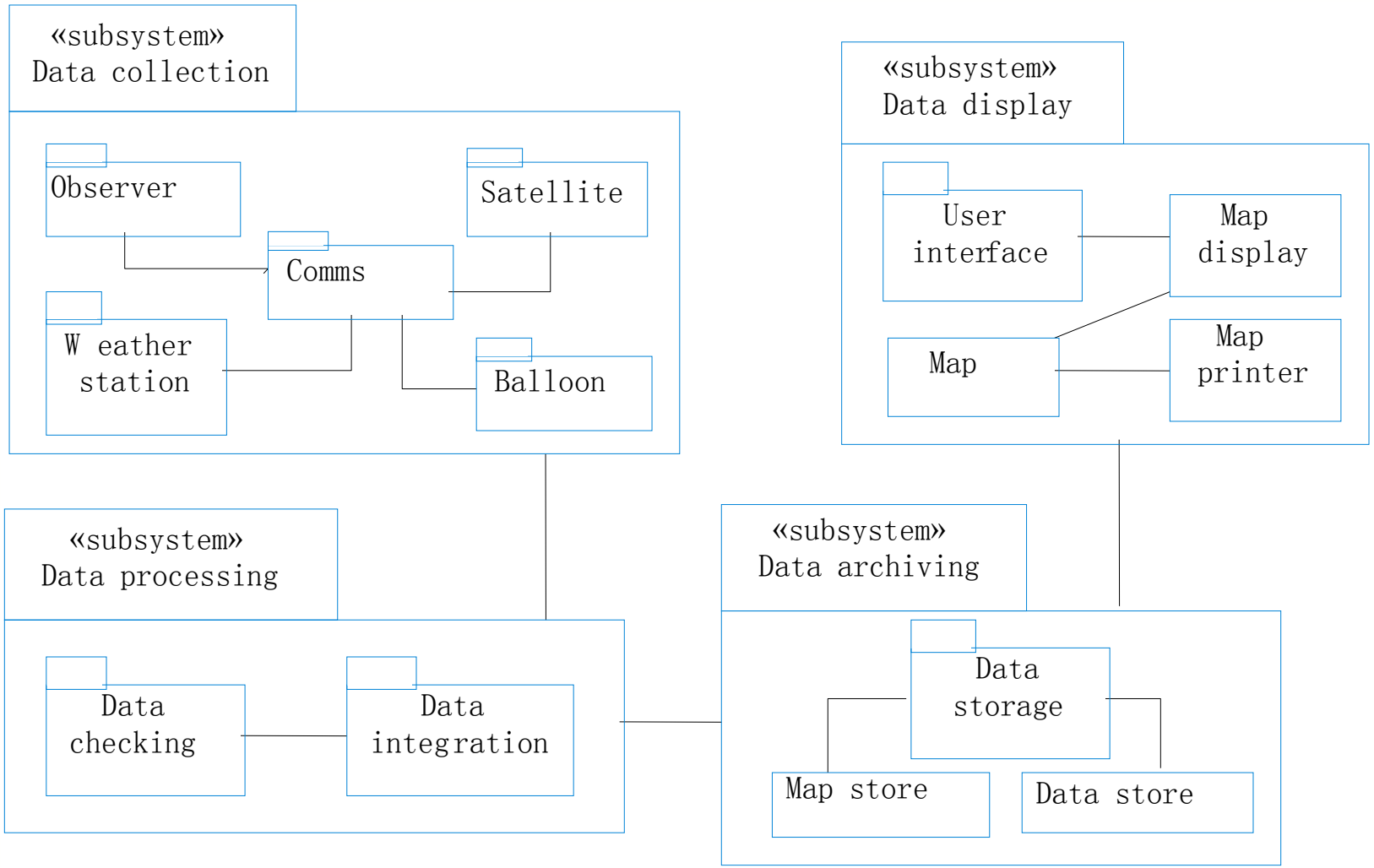
- 类与类之间的存在的“聚合、组合、关联、依赖”关系导致包与包之间存在依赖关系，即“包的依赖”(dependency)；
- 类与类之间的存在的“继承”关系导致包与包之间存在继承关系，即“包的泛化”(generalization)；



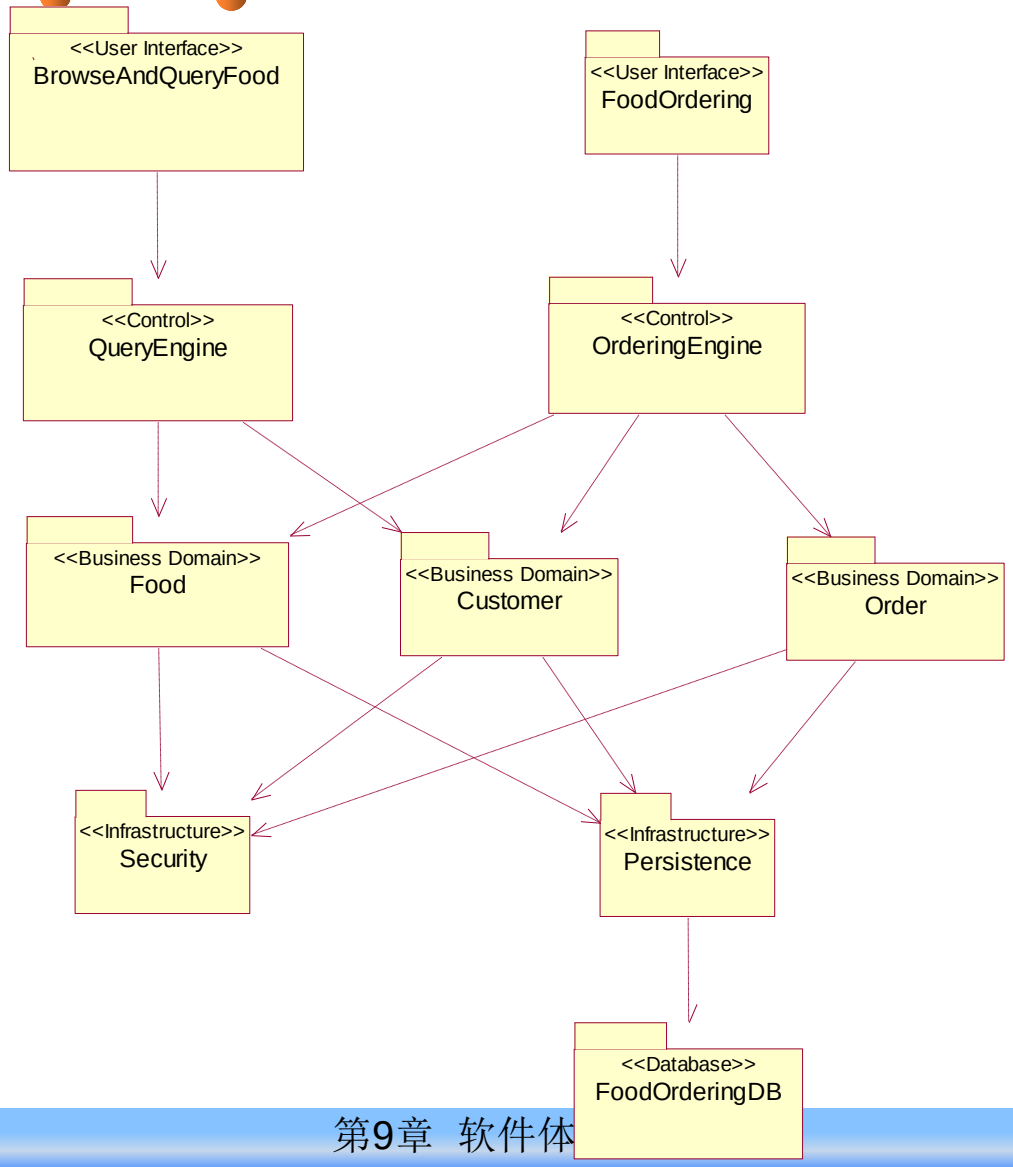
绘制包图(package diagram)的方法

- **分析设计类，把概念上或语义上相近的模型元素纳入一个包。**
- **可以从类的功能相关性来确定纳入包中的类：**
  - 如果一个类的行为和/或结构的变更要求另一个相应的变更，则这两个类是功能相关的。
  - 如果删除一个类后，另一个类便变成是多余的，则这两个类是功能相关的，这说明该剩余的类只为那个被删除的类所使用，它们之间有依赖关系。
  - 如果两个类之间大量的频繁交互或通信，则这两个类是功能相关的。
  - 如果两个类之间有一般/特殊关系，则这两个类是功能相关的。
  - 如果一个类激发创建另一个类的对象，则这两个类是功能相关的。
- **确定包与包之间的依赖关系(<<import>>、<<access>>等)；**
- **确定包与包之间的泛化关系；**
- **绘制包图。**

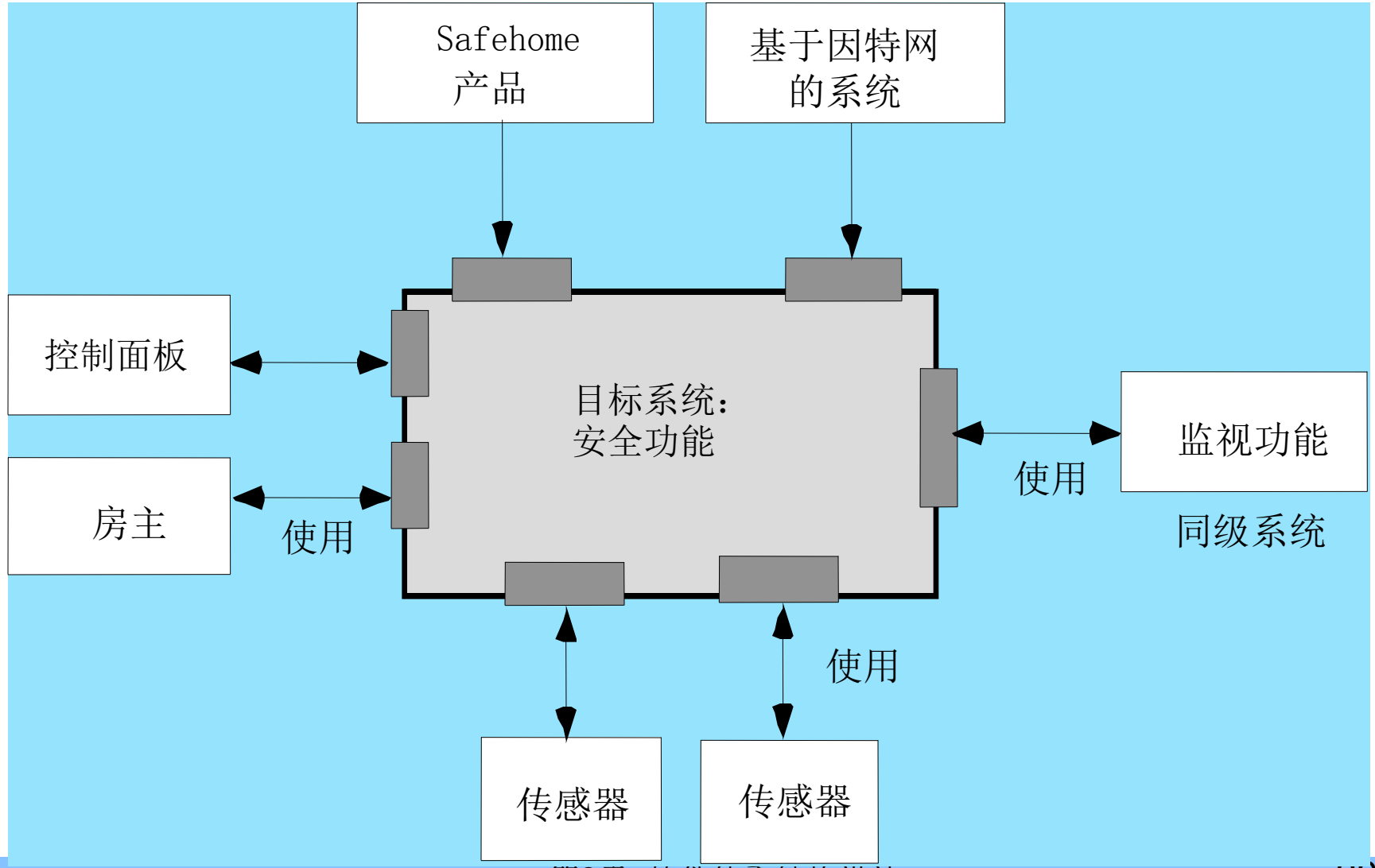
包图(package diagram)：示例



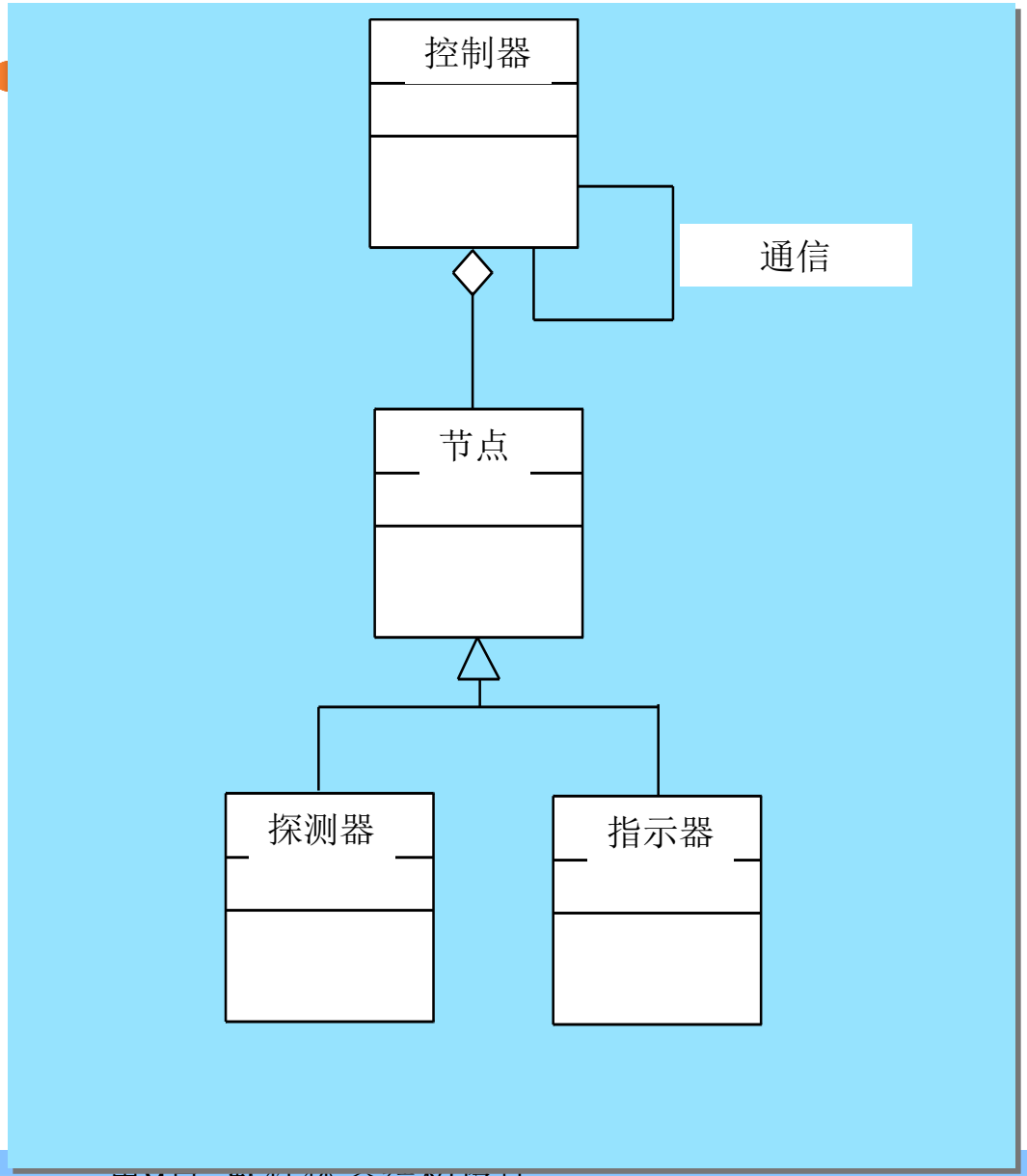
包图(package diagram)：示例



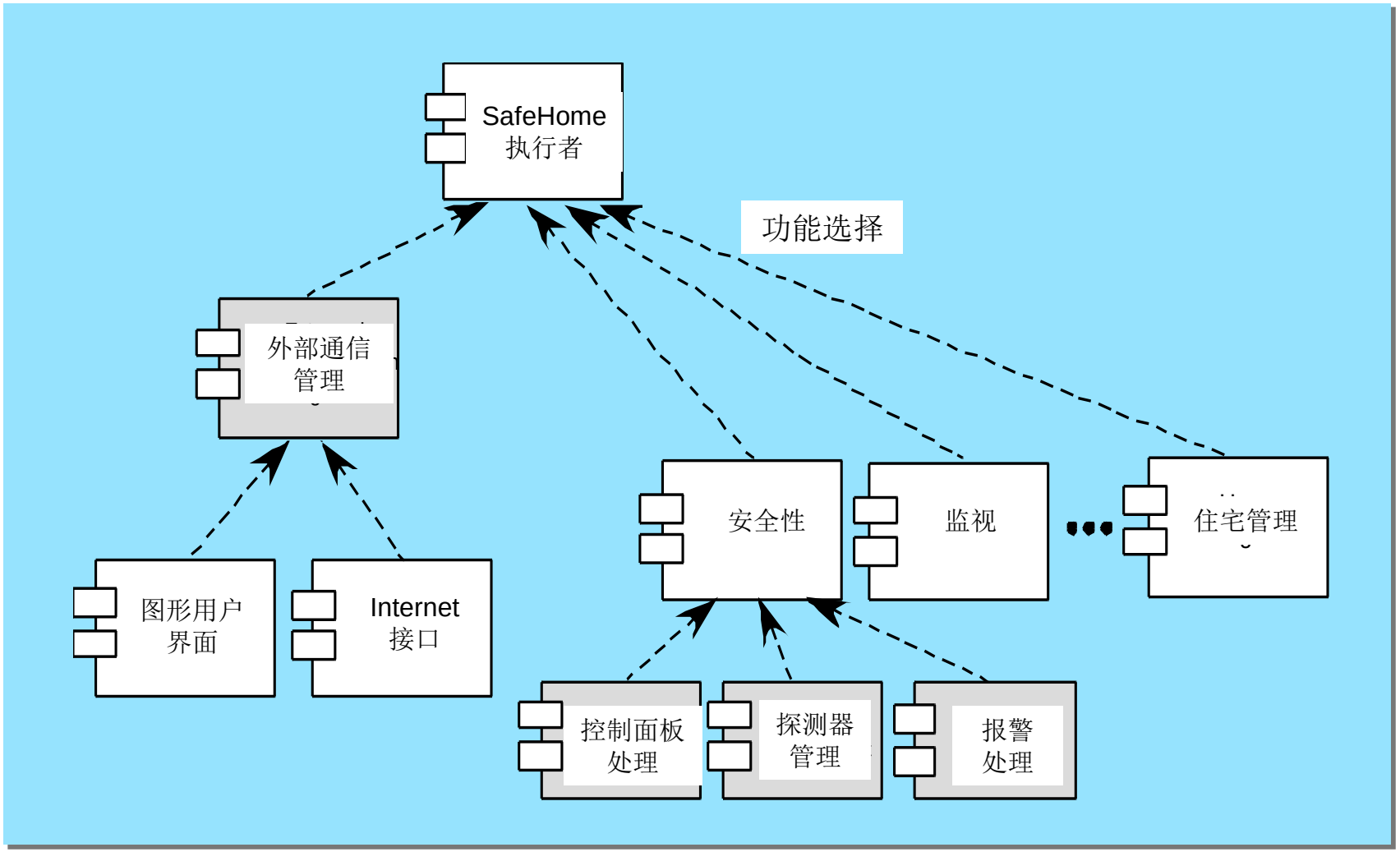
系统的环境表示


Safehome
产品
基于因特网
的系统
控制面板
房主
使用
目标系统:
安全功能
使用
监视功能
同级系统
使用
传感器
传感器

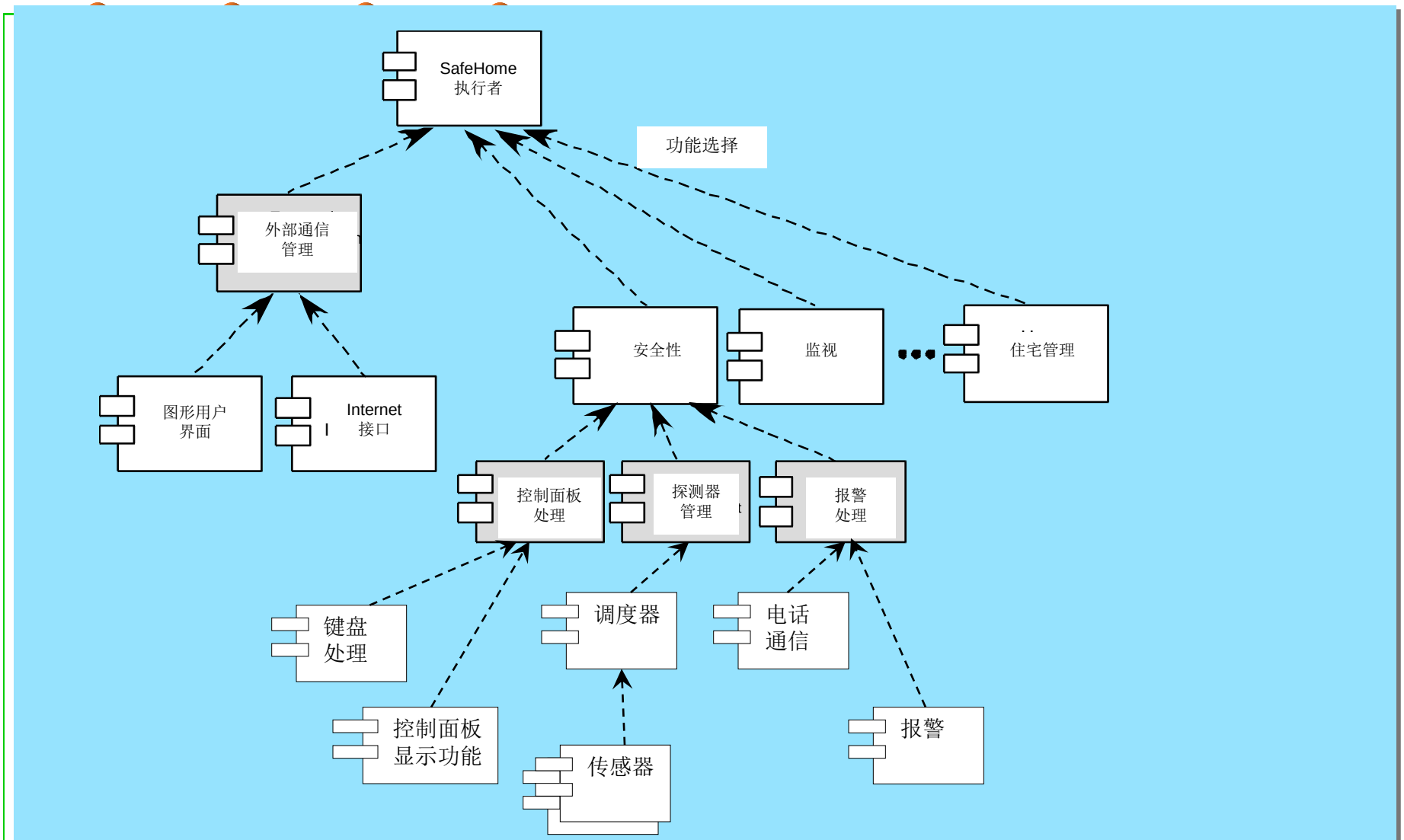
定义原始模型



将体系结构精化为构件


SafeHome
执行者
功能选择
外部通信
管理
安全性
监视
住宅管理
图形用户
界面
Internet
接口
控制面板
处理
探测器
管理
报警
处理

描述系统实例


SafeHome
执行者
功能选择
外部通信
管理
安全性
监视
住宅管理
图形用户
界面
Internet
接口
控制面板
处理
探测器
管理
报警
处理
键盘
处理
调度器
电话
通信
控制面板
显示功能
传感器
报警

结束

2011年5月11日