西安电子科技大学
XIDIAN UNIVERSITY

计算机科学与技术学院
SCHOOL OF COMPUTER SCIENCE AND TECHNOLOGY
国家示范性软件学院
NATIONAL PILOT SCHOOL OF SOFTWARE ENGINEERING

软件体系结构

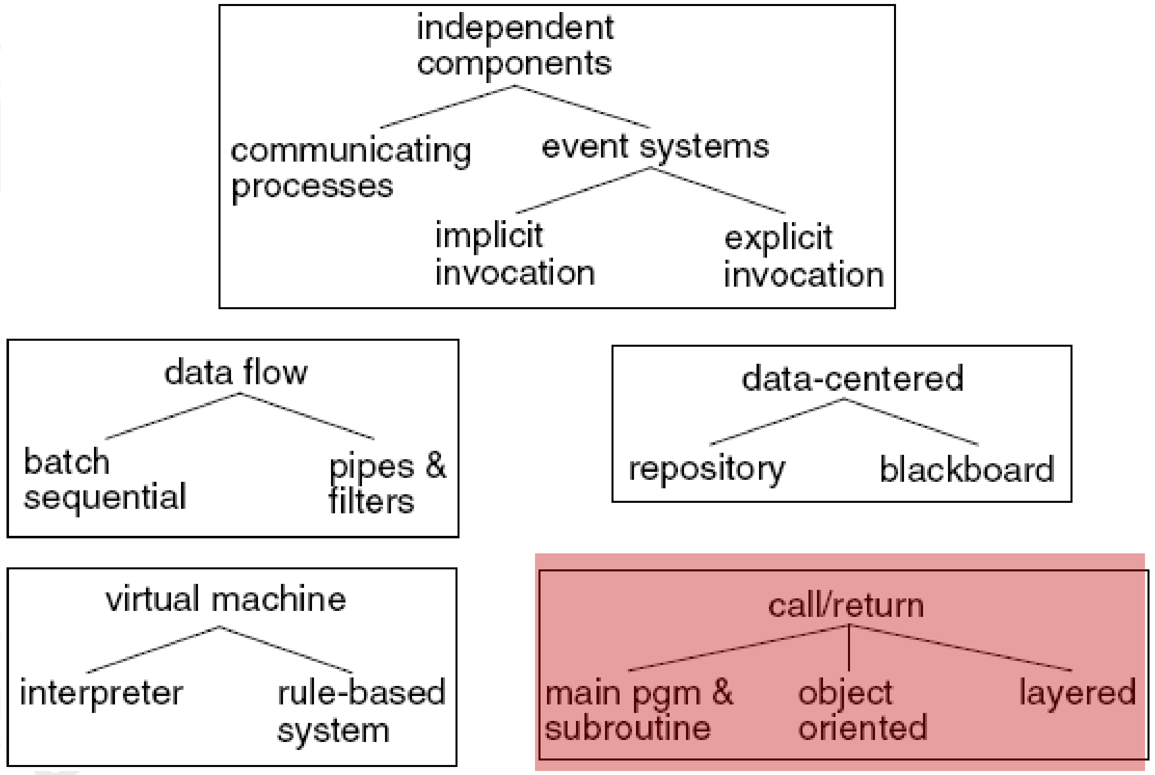
—调用/返回风格

主讲人：鲍亮 副教授

本节内容

调用/返回风格

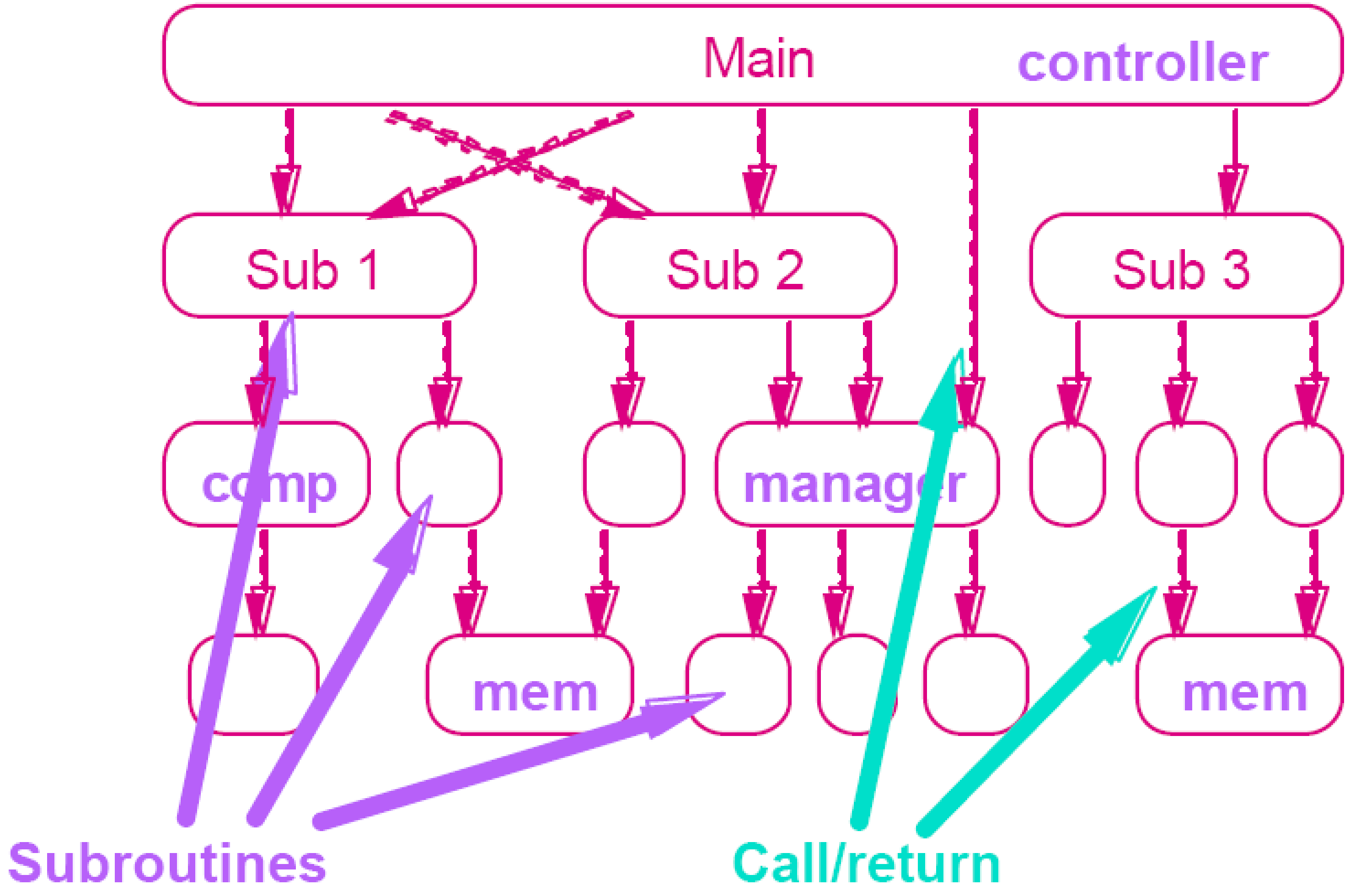
- 主程序/子程序风格
  - 经典的编程范式：功能分解
- 面向对象/抽象数据类型（ADT）
  - 信息 (表示, 方法方法) 隐藏
- 层次结构
  - 每一层只能与其直接相邻的邻居进行通信
- 其他
  - 客户端/服务器
  - ……



调用/返回风格

- 主程序/子程序风格
  - 单线程控制，划分为若干处理步骤
  - 功能模块：把步骤集成至模块中
- 抽象数据类型（ADT）
  - 操作和数据绑定在一起，隐藏实现和其他秘密
- 面向对象
  - 方法（动态绑定），多态（子类），重用（继承）
  - 对象活动于不同的进程/线程（分布式对象）
    - CS结构、分层风格
- 组件
  - 多个接口，二进制兼容，高级中间件

主程序/子程序风格


Main
controller
Sub 1
Sub 2
Sub 3
comp
manager
mem
mem
Subroutines
Call/return

主程序/子程序风格

- **Problem:** This pattern is suitable for applications in which the computation can appropriately be defined via a hierarchy of procedure definitions.
- **Context:** Many programming languages provide natural support for defining nested collections of procedures and for calling them hierarchically. These languages often allow collections of procedures to be grouped into modules, thereby introducing name-space locality. The execution environment usually provides a single thread of control in a single name space.
- **Solution:**
  - *System model:* call and definition hierarchy, subsystems often defined via modularity
  - *Components:* procedures and explicitly visible data
  - *Connectors:* procedure calls and explicit data sharing
  - *Control structure:* single thread

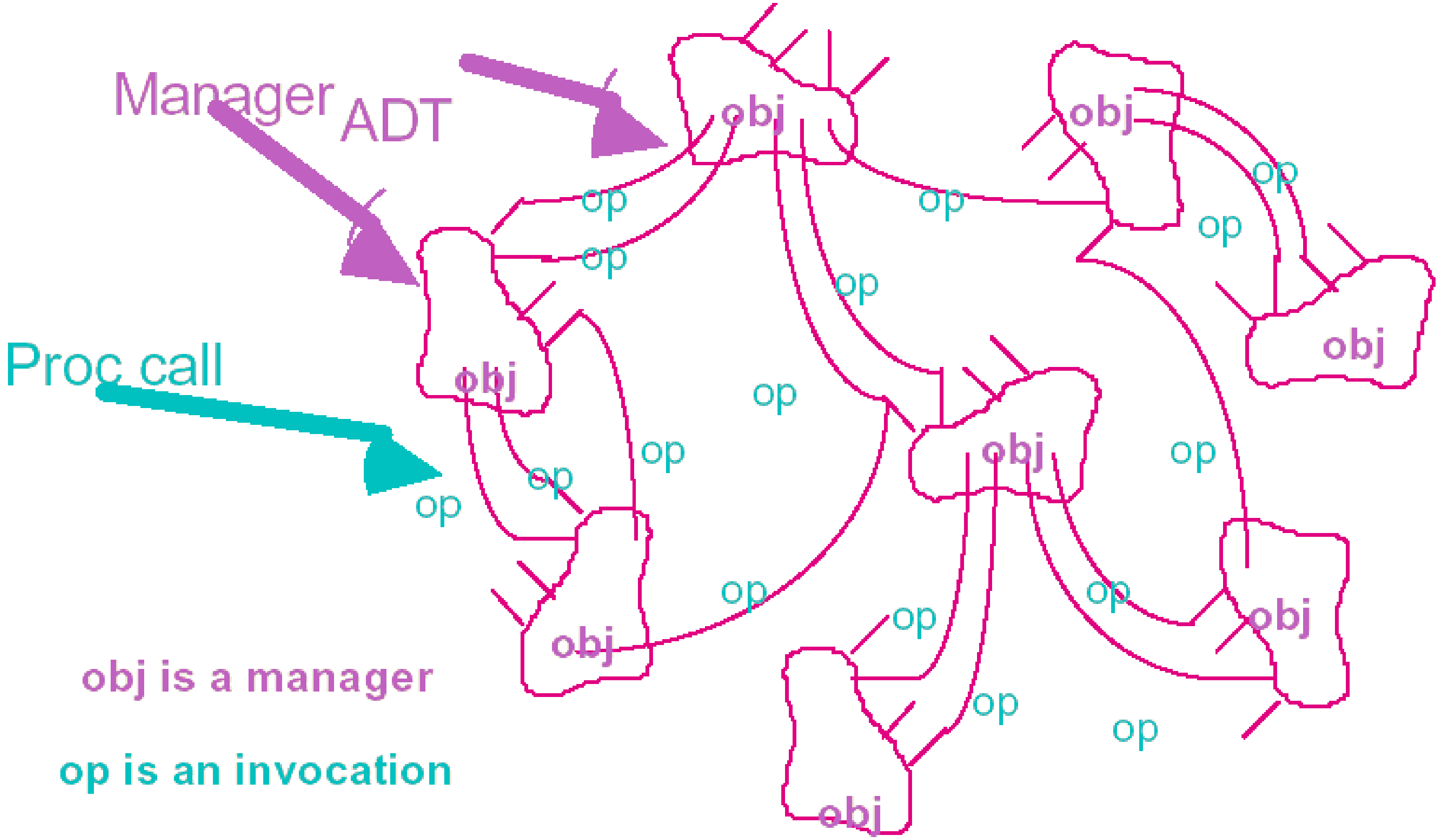
模块分解

- Parnas
  - Hide secrets. OK, what’s a “secret”?
    - Representation of data
    - Properties of a device, other than required properties
    - Mechanisms that support policies
  - Try to localize future change
    - Hide system details likely to change independently
    - Expose in interfaces assumptions unlikely to change
  - Use functions to allow for change
    - They’re easier to change than visible representation

封装/信息隐藏

- Parnas: Hide secrets (not just representations)
- Booch: Object's behavior is characterized by actions that it suffers and that it requires
- Practically speaking:
  - Object has state and operations, but also has responsibility for the integrity of its state
  - Object is known by its interface
  - Object is probably instantiated from a template
  - Object has operations to access and alter state and perhaps generator
  - There are different kinds of objects (e.g., actor, agent, server)

数据抽象/面向对象



obj is a manager

op is an invocation

数据抽象/面向对象

- **Problem:** This pattern is suitable for applications in which a central issue is identifying and protecting related bodies of information, especially representation information.
- **Context:** Numerous design methods provide strategies for identifying natural objects. Newer programming languages support various variations on the theme, so if the language choice or the methodology is fixed, that will strongly influence the flavor of the decomposition.
- **Solution:**
  - *System model:* localized state maintenance
  - *Components:* managers (e.g., servers, objects, abstract data types)
  - *Connectors:* procedure call
  - *Control structure:* decentralized, usually single thread

面向对象体系结构的元素

- Encapsulation: Restrict access to certain information
  封装：限制对某些信息的访问
- Interaction: Via procedure calls or similar protocol
  交互：通过过程调用或类似的协议
- Polymorphism: Choose the method at run-time
  多态：在运行时选择具体的操作
- Inheritance: Shared definitions of functionality
  继承：对共享的功能保持唯一的接口
- Reuse and maintenance: Exploit encapsulation and locality to increase productivity
  复用和维护：利用封装和聚合提高生产力

面向对象方法的问题

- Managing many objects
  - Vast sea of objects requires additional structuring
    对象的海洋需要额外的结构来容纳
  - Hierarchical design suggested by Booch and Parnas
- Managing many interactions
  - Single interface can be limiting & unwieldy (hence, “friends”)
    单一的接口能力有限并且笨拙（于是，”友元”）
  - Some languages/systems permit multiple interfaces (inner class, interface, multiple inheritance)
- Distributed responsibility for behavior
  - Makes system hard to understand
  - Interaction diagrams now used in design
- Capturing families of related designs
  - Types/classes are often not enough
  - Design patterns as an emerging off-shoot

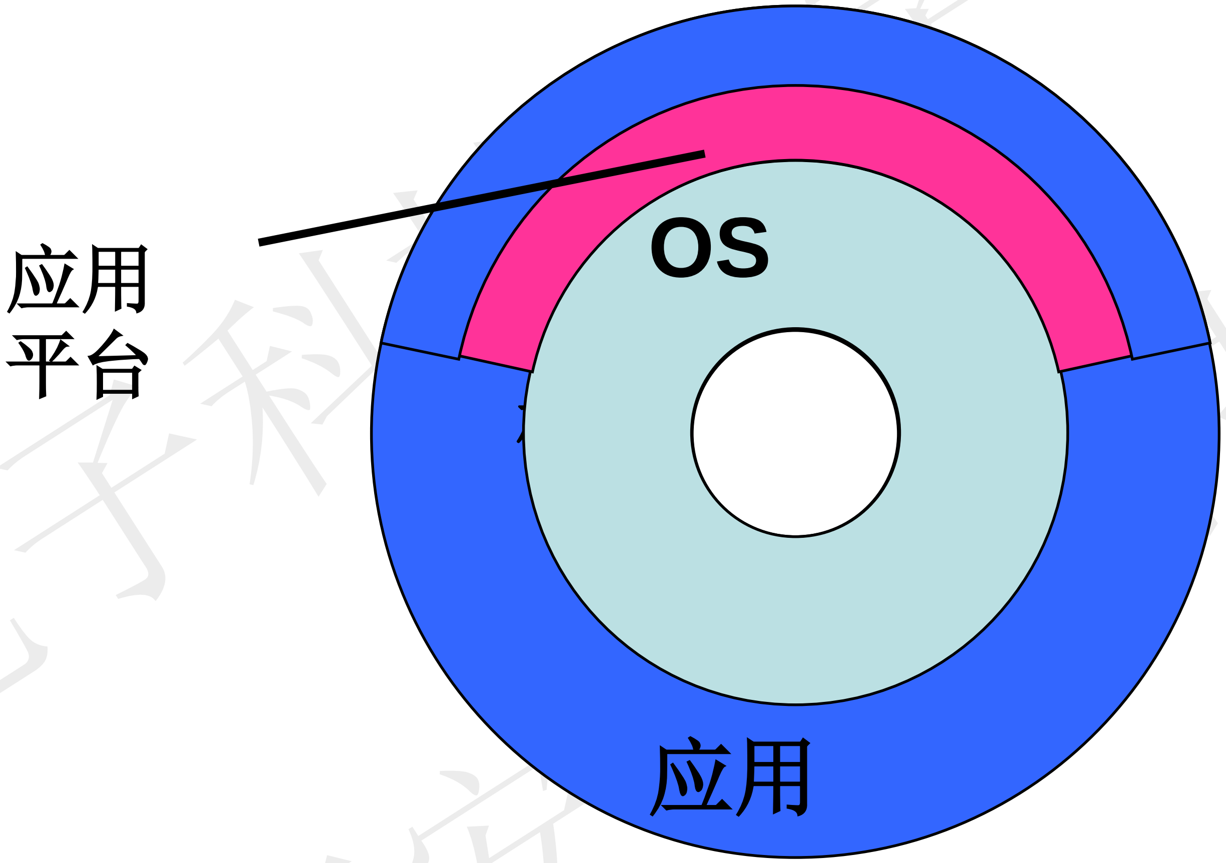
如何管理大的对象数据集

- Pure O-O design leads to large flat systems with many objects
  - Same old problems can reappear
  - Hundreds of modules => hard to find things
  - Need a way to impose structure
- Need additional structure and discipline
- Structuring options
  - Layers (which are not necessarily objects)
  - Supplemental index
  - Hierarchical decomposition: big objects and little objects

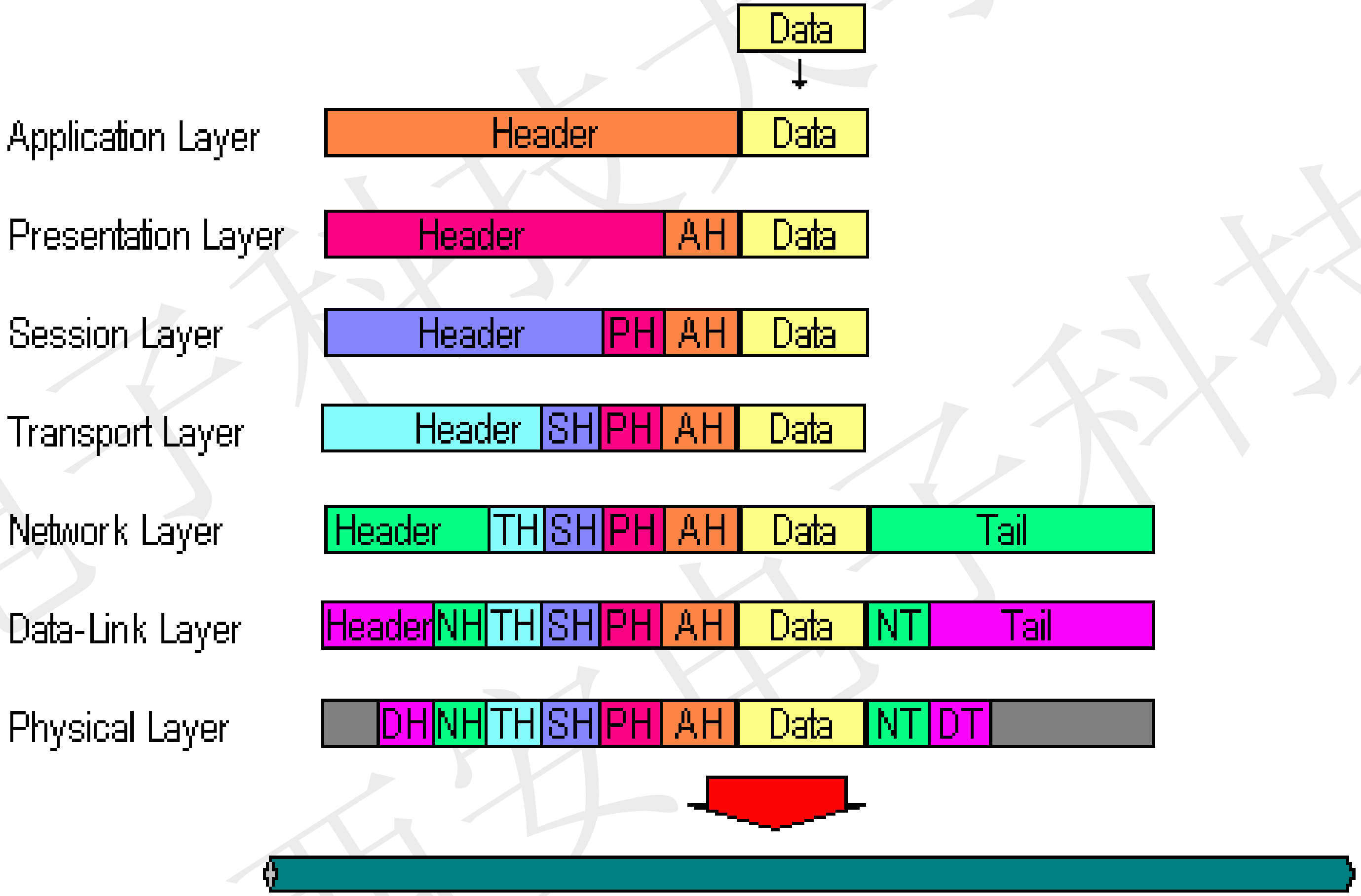
体系结构风格变种

- Client-server
  - Objects are processes
    进程就是对象
  - Asymmetric: client knows about servers, but not vice versa
    不对称：客户端知道服务器，反之则不然
- Tiered
  - Elaboration on client-server
    C/S模式的扩展
  - Aggregation into run-time strata
    运行时层的结合
  - Usually small number of tiers
    通常只有少量的层
- Components (later)
  - Multiple interfaces
  - Special protocols for dynamic reconfiguration
    支持动态配置的专门协议

层次风格案例-操作系统



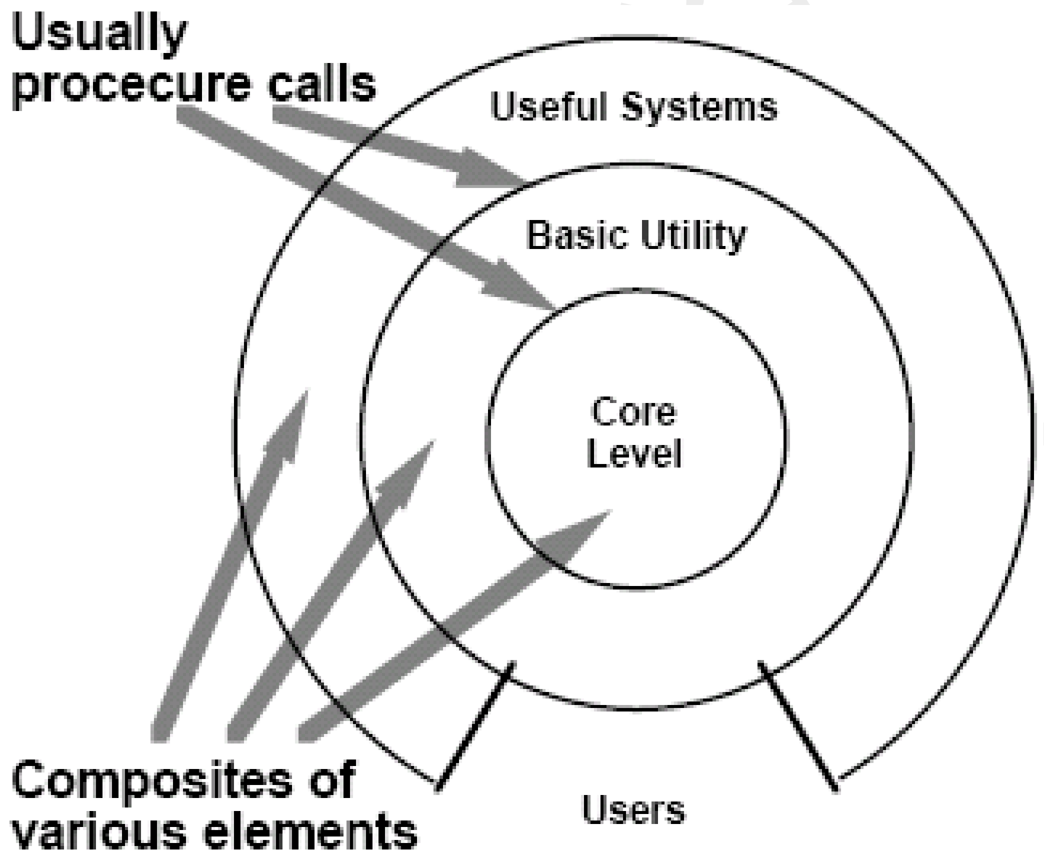
层次案例-计算机网络


Data
Application Layer
Header
Data
Presentation Layer
Header
AH
Data
Session Layer
Header
PH
AH
Data
Transport Layer
Header
SH
PH
AH
Data
Network Layer
Header
TH
SH
PH
AH
Data
Tail
Data-Link Layer
Header
NH
TH
SH
PH
AH
Data
NT
Tail
Physical Layer
DH
NH
TH
SH
PH
AH
Data
NT
DT

层次风格

- **Problem:** This pattern is suitable for applications that involve distinct classes of services that can be arranged hierarchically. Often there are layers for basic system-level services, for utilities appropriate to many applications, and for specific tasks of the application.
- **Context:** Frequently, each class of service is assigned to a layer and several different patterns are used to refine the various layers. Layers are most often used at the higher levels of design, using different patterns to refine the layers.
- **Solution:**
  – *System model:* hierarchy of opaque layers
  – *Components:* usually composites; composites are most often collections of procedures
  – *Connectors:* depends on structure of components; often procedure calls under restricted visibility, might also be client/server
  – *Control structure:* single thread

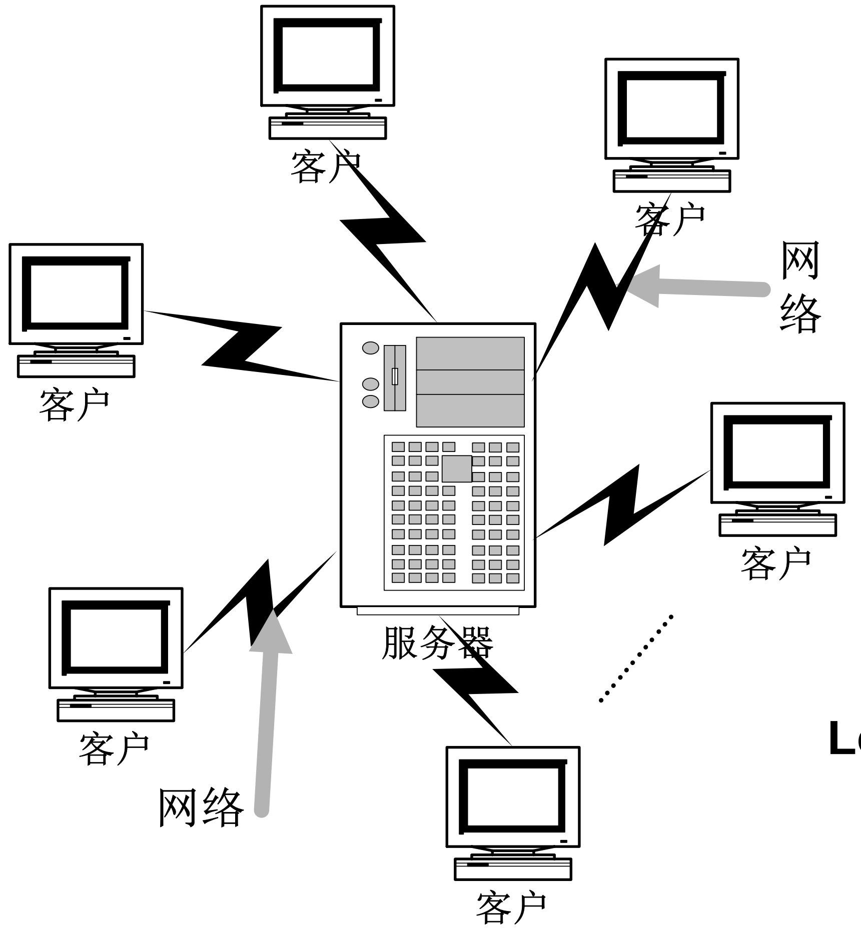
层次风格参考结构

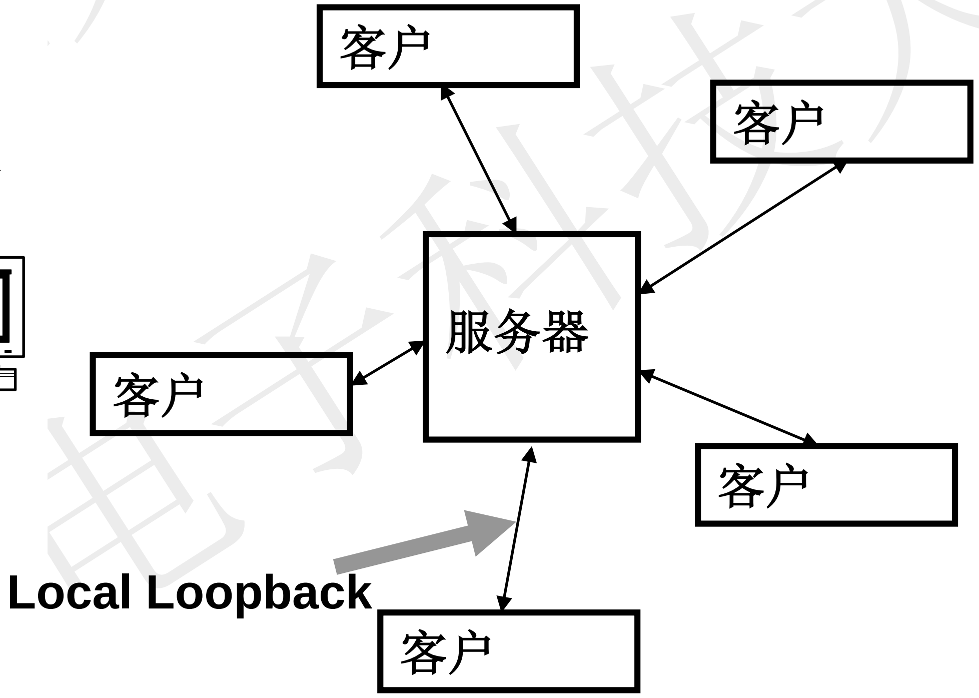

Usually
procecure calls
Useful Systems
Basic Utility
Core
Level
Composites of
various elements
Users

层次风格特点

- 每层为上一层提供服务，使用下一层的服务，只能见到与自己邻接的层
  - 适当时候(必不得已的时候)，可以允许一定的越层操作
- 大的问题分解为若干个渐进的小问题，逐步解决，隐藏了很多复杂度
- 修改一层，最多影响两层，而通常只能影响上层。接口稳固，则谁都不影响
- 上层必须知道下层的身份，不能调整层次之间的顺序
- 层层相调，影响性能

客户端/服务器风格

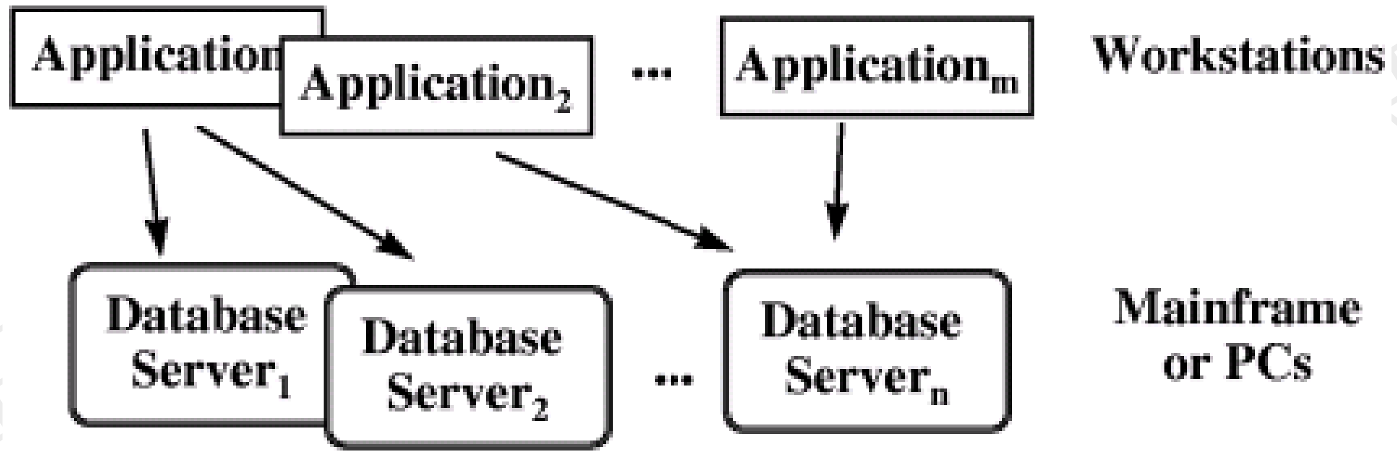




客户端/服务器风格

- 两层C/S结构
- 三层C/S结构
- B/S结构（浏览器/服务器风格）

两层客户端/服务器风格


Application$_1$
Application$_2$
...
Application$_m$
Workstations
Database Server$_1$
Database Server$_2$
...
Database Server$_n$
Mainframe or PCs

两层客户端/服务器风格

- C/S软件体系结构是基于资源不对等，且为实现共享而提出的，20世纪90年代成熟起来
- C/S体系结构有三个主要组成部分：数据库服务器、客户应用程序和网络
- 服务器（后台）负责数据管理，客户机（前台）完成与用户的交互任务。 “胖客户机，瘦服务器”
- 缺点：
  - 对客户端软硬件配置要求较高，客户端臃肿
  - 客户端程序设计复杂
  - 数据安全性不好。客户端程序可以直接访问数据库服务器。
  - 信息内容和形式单一
  - 用户界面风格不一，使用繁杂，不利用推广使用
  - 软件维护与升级困难。每个客户机上的软件都需要维护

三层客户端/服务器风格

- 与二层C/S结构相比，增加了一个应用服务器。
- 整个应用逻辑驻留在应用服务器上，只有表示层存在于客户机上：“瘦客户机”
- 应用功能分为表示层、功能层、数据层三层
  - 表示层是应用的用户接口部分。通常使用图形用户界面
  - 功能层是应用的主体，实现具体的业务处理逻辑
  - 数据层是数据库管理系统。
  - 以上三层逻辑上独立。
  - 通常只有表示层配置在客户机中

软件体系结构

- B/S体系结构是三层C/S体系结构的特例
- 客户端有http浏览器即可
  - 为增强功能，往往还需要安装flash、jvm及一些专用插件
- 使用标准http/https协议，省却很多麻烦
- 只能“拉”，不能“推”
- 客户之间的通信只能通过服务器中转
- 对客户机资源和其他网络资源的利用受限
- B/S结构的安全性较难控制(SQL注入攻击…)
- B/S结构的应用系统在数据查询等相应速度上，要远远低于C/S体系结构
- 服务器的负荷大，客户机的资源浪费
  - 用jvm、flash、ActiveX等客户端计算技术解决

调用/返回风格总结

- 主程序/子程序风格
  - 单线程控制，划分为若干处理步骤
  - 功能模块：把步骤集成至模块中
- 抽象数据类型（ADT）
  - 操作和数据绑定在一起，隐藏实现和其他秘密
- 面向对象
  - 方法（动态绑定），多态（子类），重用（继承）
  - 对象活动于不同的进程/线程（分布式对象）
    - CS结构、分层风格
- 组件
  - 多个接口，二进制兼容，高级中间件

西安电子科技大学
XIDIAN UNIVERSITY

谢谢大家！

计算机科学与技术学院微信

课程讨论群

鲍亮 副教授 博导

邮箱：baoliang@xidian.edu.cn

主页：https://web.xidian.edu.cn/yslin/