第四部分 软件设计

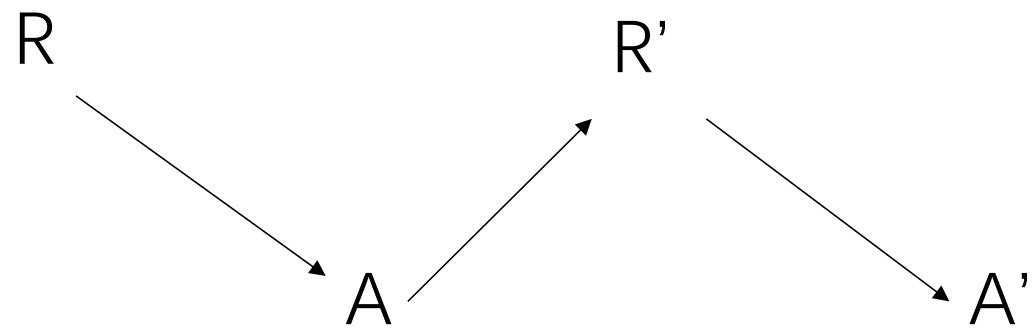
- 4.1 软件工程开发方法与软件设计
- 4.2 体系结构设计
- 4.3 类/数据建模与设计
- 4.4 行为建模与设计
- 4.5 物理建模与设计

4.2 体系结构设计

- 软件体系结构要素
- 软件体系结构风格

软件体系结构

- 需求工程师和架构师的区别。。。



软件的复杂性

- 问题：当系统的规模和复杂度不断增大的时候，构造整个系统的关键是什么？

考虑的关键问题是对整个系统的结构和行为进行抽象

软件的复杂性

- 举例：要求编写一个程序，该程序读入一个文本文件，统计在该文本文件中每个英文单词出现的频率，并输出单词频率最高的100个单词。

构造上述软件的重点是设计哪些部分

软件的复杂性

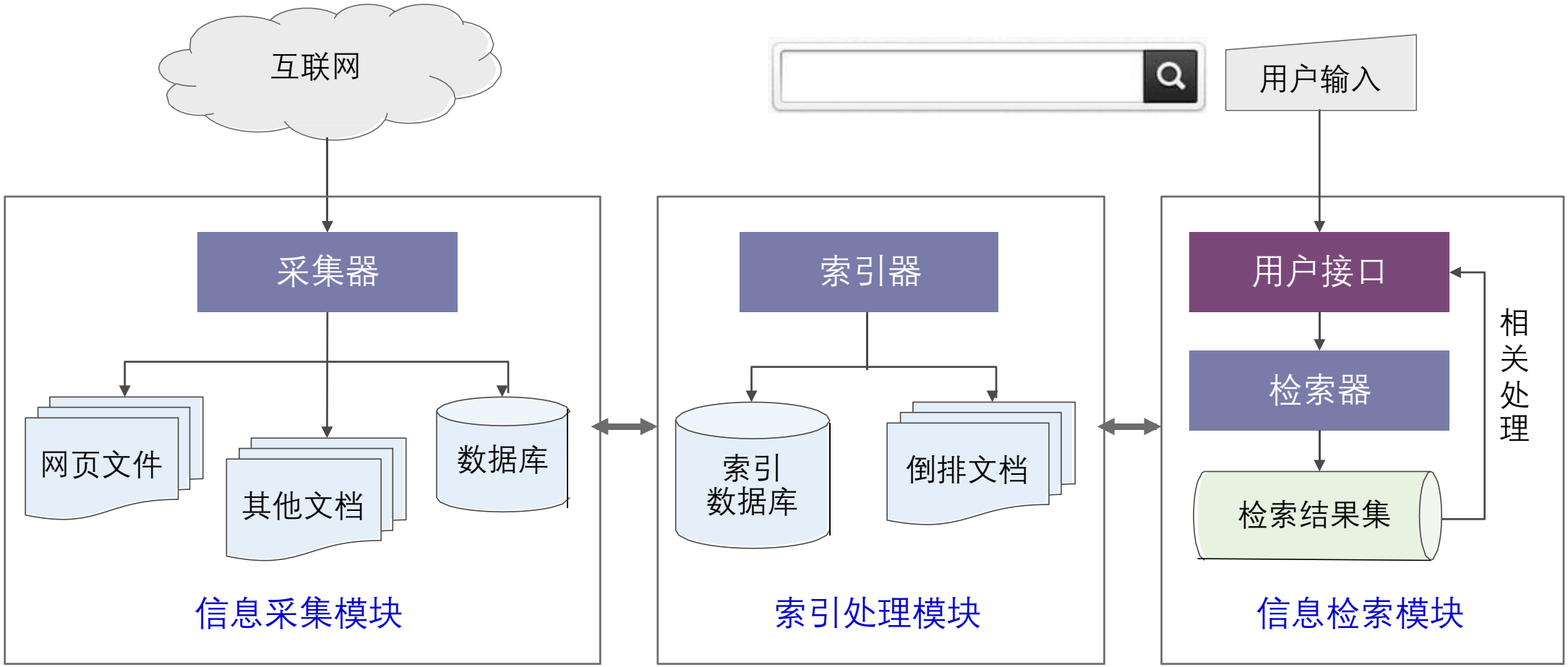
- 举例：Web信息检索是对发布在Web上的信息资源进行搜集、整理和组织，形成一个信息资源索引库，并通过检索界面将最符合用户要求的网站或网页提供给用户。

软件的复杂性

- 举例：Web信息检索是对发布在Web信息资源进行搜集、整理和组织，形成一个信息资源索引库，并通过检索界面将最符合用户要求的网站或网页提供给用户。

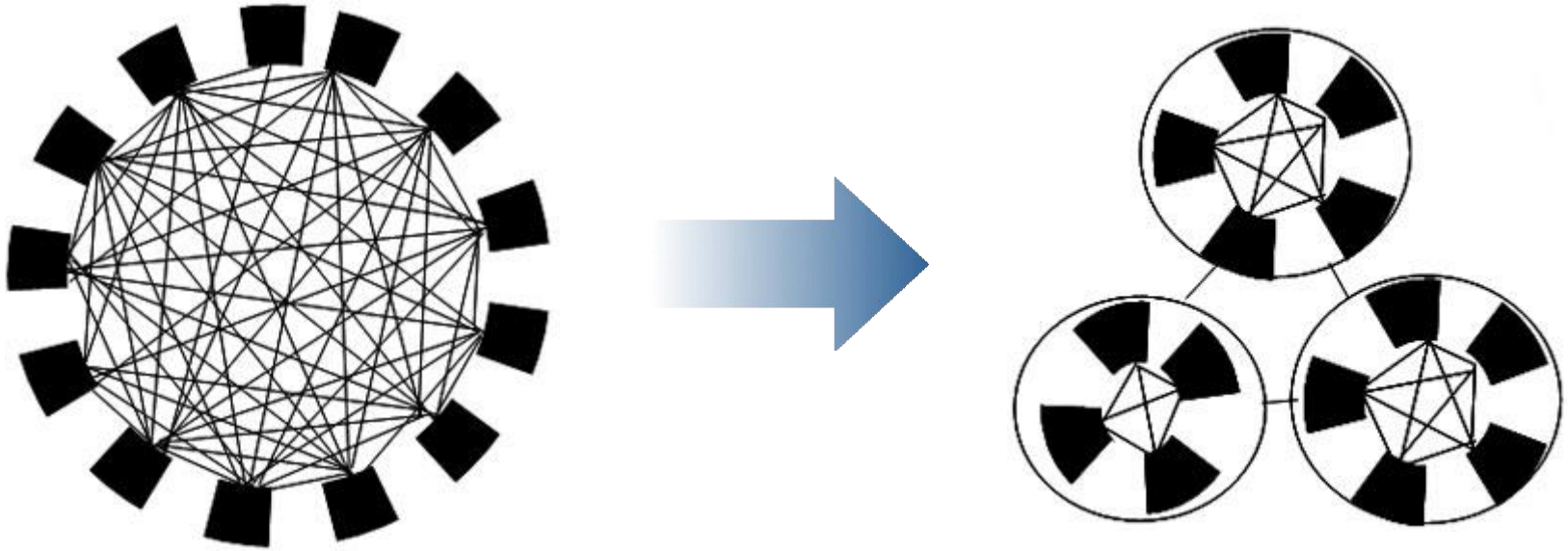
Web信息检索的处理过程：

软件的复杂性


互联网
用户输入
采集器
索引器
用户接口
检索器
相关处理
网页文件
其他文档
数据库
索引
数据库
倒排文档
检索结果集
信息采集模块
索引处理模块
信息检索模块

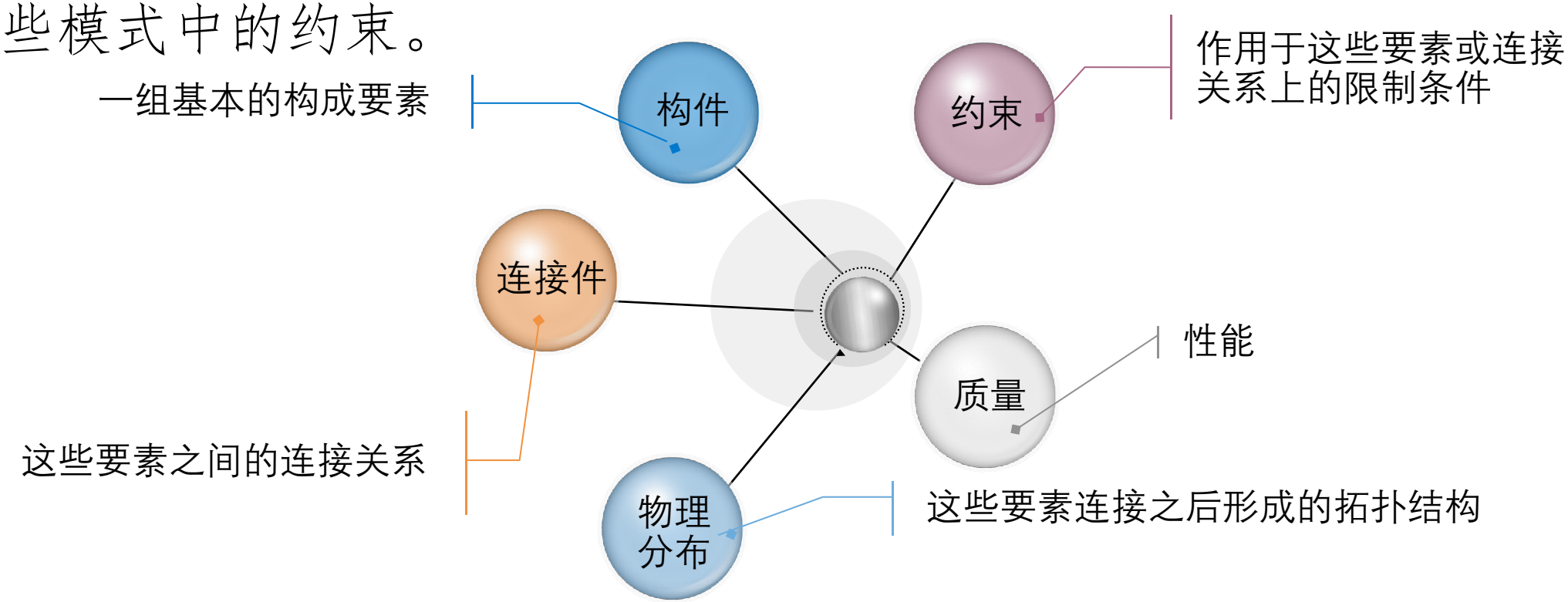
处理复杂性

- 随着软件系统的规模和复杂性不断增加，对系统的全局结构设计和规划变得比算法的选择和数据结构的设计明显重要得多。



软件体系结构概念

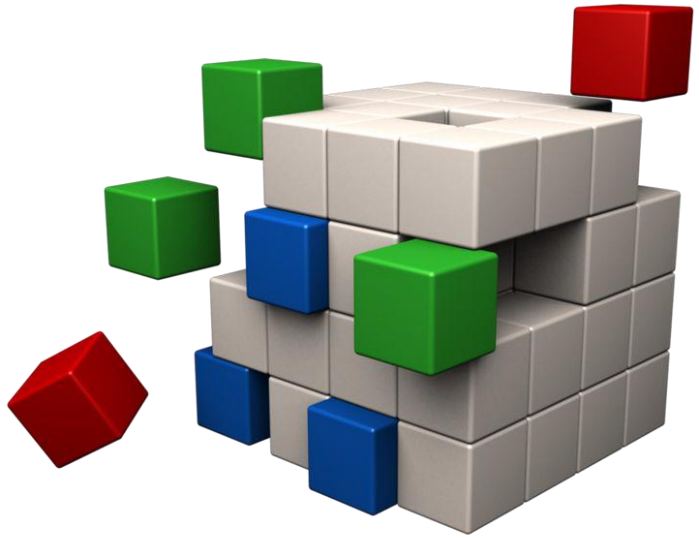
- 软件体系结构（Software Architecture）包括构成系统的设计元素的描述、设计元素之间的交互、设计元素的组合模式以及在这些模式中的约束。



软件体系结构概念

软件体系结构 = 构件 + 连接件 + 约束

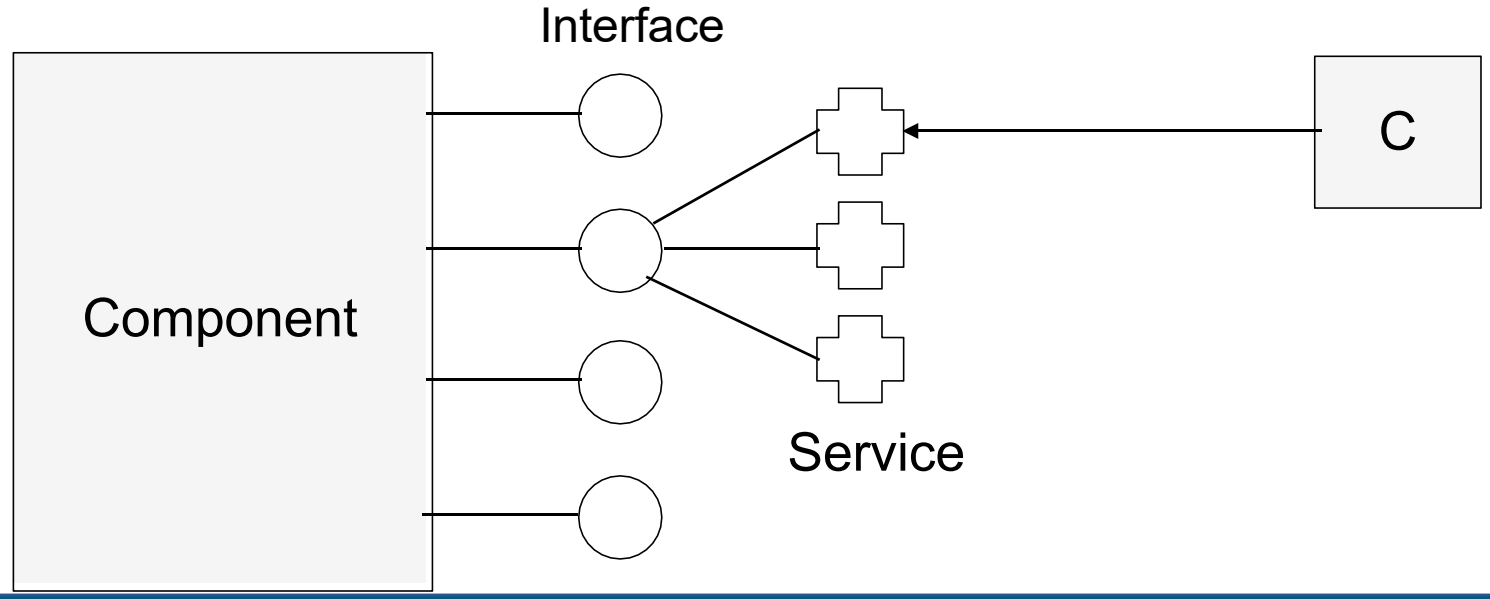
- 如何将复杂的软件系统划分成模块
- 如何规范模块的构成
- 如何将这些模块组织成为完整的系统
- 如何保证系统的质量要求



1. 构件(Component)

- 构件是具有某种功能的可复用的软件结构单元，表示系统中主要的计算元素和数据存储。
- 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
文件
相关功能的集合
……



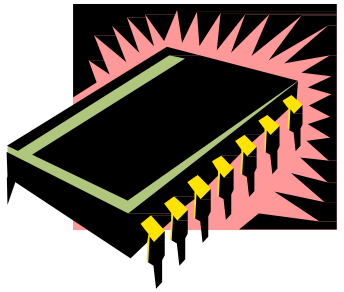
构件的特点

- 可分离：一个或数个可独立部署执行码文件
- 可替换：构件实例可被其他任何实现了相同接口的另一构件实例所替换
- 可配置：外界可通过规范化的配置机制修改构件配置数据，进而影响（或称“定制”）构件的对外服务的功能或行为
- 可复用：构件可不经源代码修改，无需重新编译，即可应用于多个软件项目或软件产品

接口(Interface)与功能(Function)

- 构件组成
  1) 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
     - 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



  2) 实现（功能）
     - 构件接口服务的实现
     - 构件核心逻辑实现
     - 构件内部所实现的功能以方法、操作(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；

机制

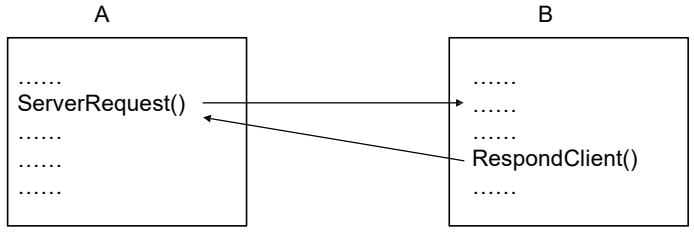
过程调用、中断、I/O、事件、进程、线程、共享、同步、并发、消息、远程调用、动态连接、API 等等

协议

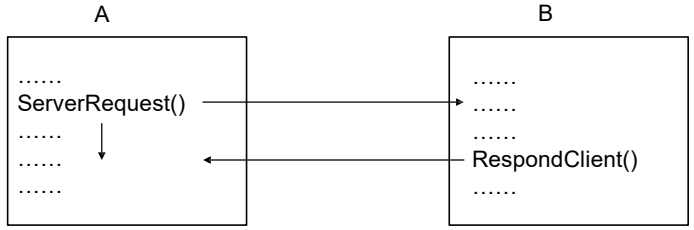
对过程调用来说：参数的个数和类型、参数排列次序
对消息传送来说：消息的格式

连接的机制(mechanism)

- 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”，分为：
  - 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。

3. 约束（Constraints）

- 高层次的软件元素可以向低层次软件元素发出请求，低层次软件元素完成计算后向高层次发送服务应答，反之不行
- 每个软件元素根据其职责位于适当的层次，不可错置，如核心层不能包含界面输入接收职责
- 每个层次都是可替换的，一个层次可以被实现了同样的对外服务接口的层次所替代

软件体系结构的目标

- **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。

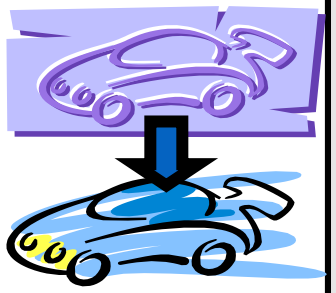
四种设计观

Decomposition & Synthesis

- Drivers:
  - Managing complexity
  - Reuse
- Example:
  - Design a car by designing separately the chassis, engine, drivetrain, etc. Use existing *components* where possible

Search

- Drivers
  - Transformation
  - Heuristic Evaluation
- Example:
  - Design a car by *transforming* an initial rough design to get closer and closer to what is desired



Negotiation

- Drivers
  - Stakeholder Conflicts
  - Dialogue Process
- Example:
  - Design a car by getting *each stakeholder* to suggest (partial) designs, and then compare and discuss them

Situated Design

- Drivers
  - Errors in existing designs
  - Evolutionary Change
- Example:
  - Design a car by observing what's wrong with existing cars *as they are used*, and identifying improvements

4.2 体系结构设计

- 软件体系结构要素
- 软件体系结构风格

从“建筑风格”开始

- 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）。


宫殿风格


园林风格


文艺复兴风格

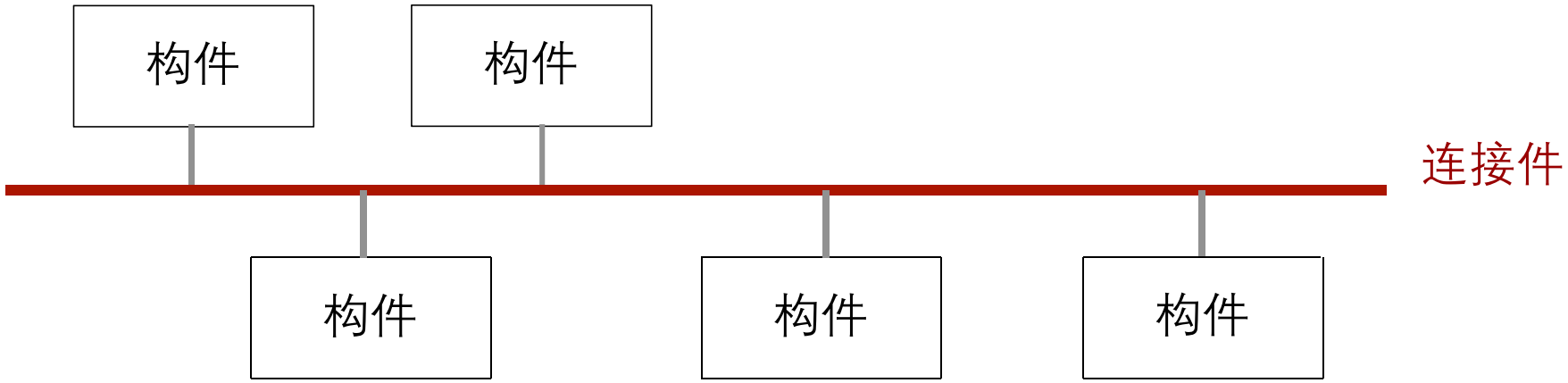

巴洛特风格

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。

软件体系结构风格

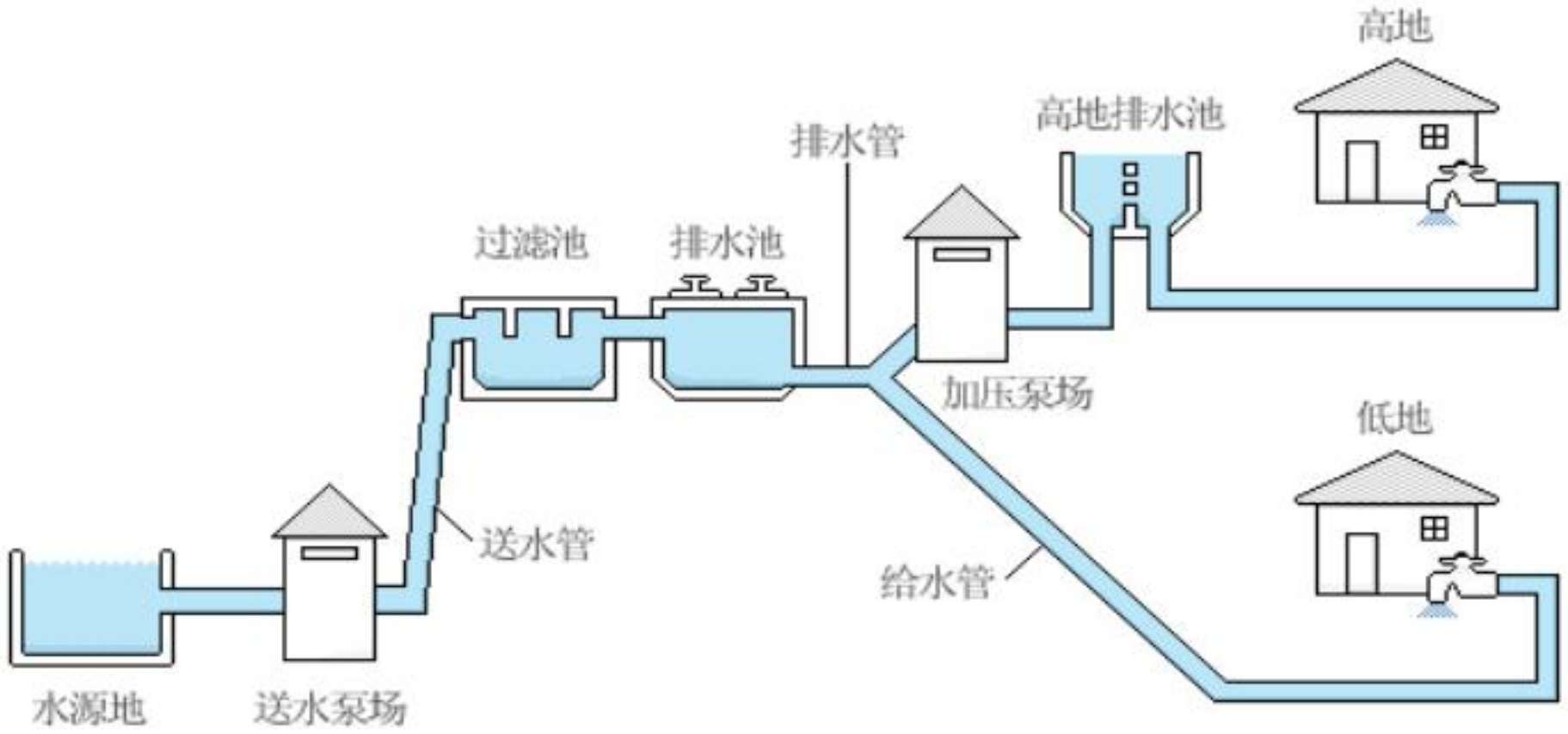
- **定义：**
  - 描述特定领域中软件系统家族的组织方式的惯用模式，反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。



经典体系结构风格

1. 数据流风格
2. 以数据为中心的风格(仓库)
3. 调用和返回体系结构风格
4. 面向对象体系结构风格
5. 层次体系结构风格
   - 客户机-服务器（C/S）
   - 浏览器-服务器（B/S)
   - C/S+B/S混合模式

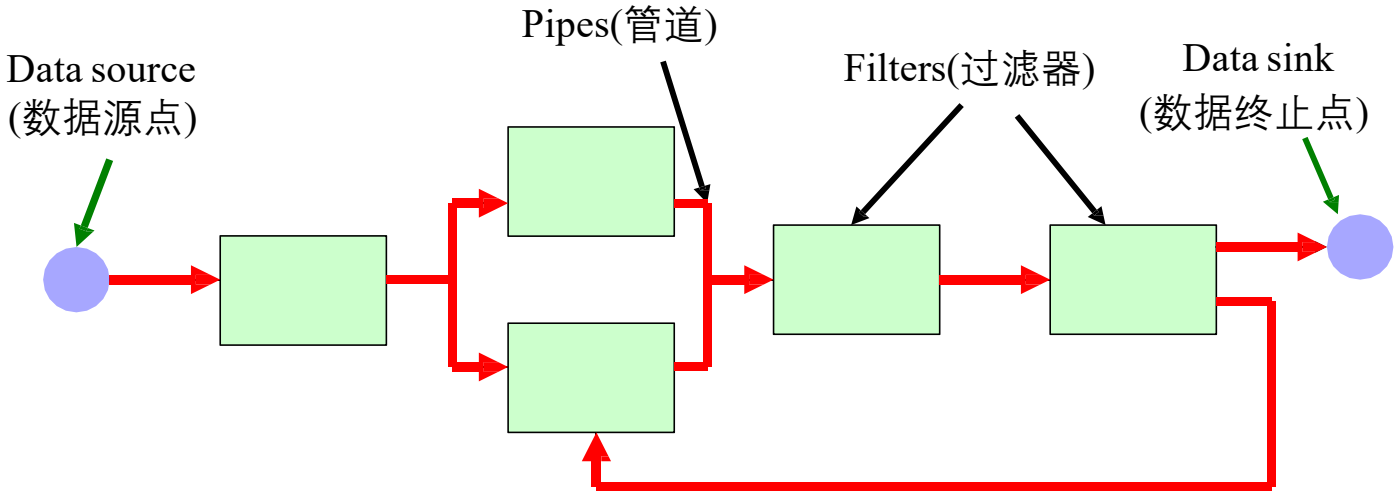
现实中的“数据流”体系结构



数据流风格

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。
- 每个过滤器独立于其上游和下游的构件而工作，过滤器的设计要针对某种形式的数据输入，并且产生某种特定形式的数据输出（到下一个过滤器）。过滤器没有必要了解与之相邻的其他过滤器的工作。

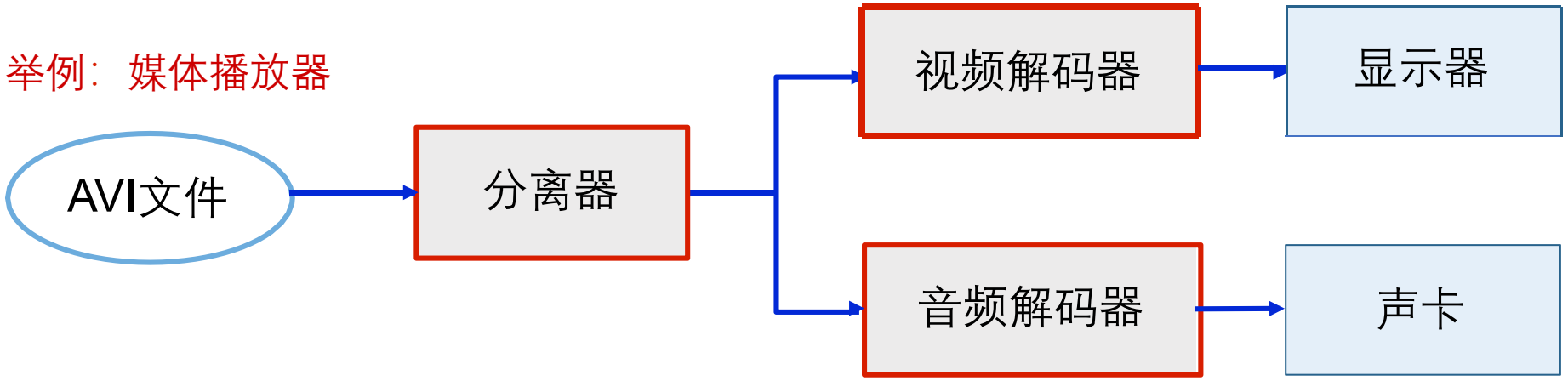


数据流风格

管道-过滤器风格：

- 把系统任务分成若干连续的处理步骤，这些步骤由通过系的数据流连接，一个步骤的输出是下一个步骤的输入。
- 每个过滤器独立于其上游和下游的构件而工作，过滤器的设计要针对某种形式的数据输入，并且产生某种特定形式的数据输出（到下一个过滤器）。过滤器没有必要了解与之相邻的其他过滤器的工作。

举例：媒体播放器



经典体系结构风格

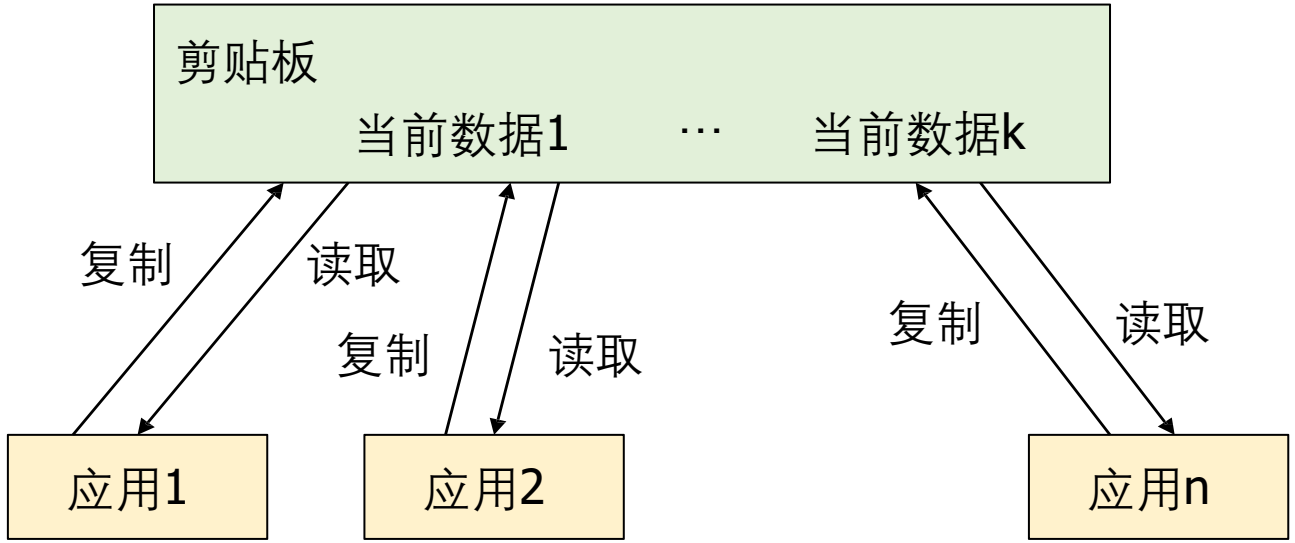
1. 数据流风格
2. 以数据为中心的风格(仓库)
3. 调用和返回体系结构风格
4. 面向对象体系结构风格
5. 层次体系结构风格
   - 客户机-服务器（C/S）
   - 浏览器-服务器（B/S)
   - C/S+B/S混合模式

以数据为中心的体系结构风格示例

- 例：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

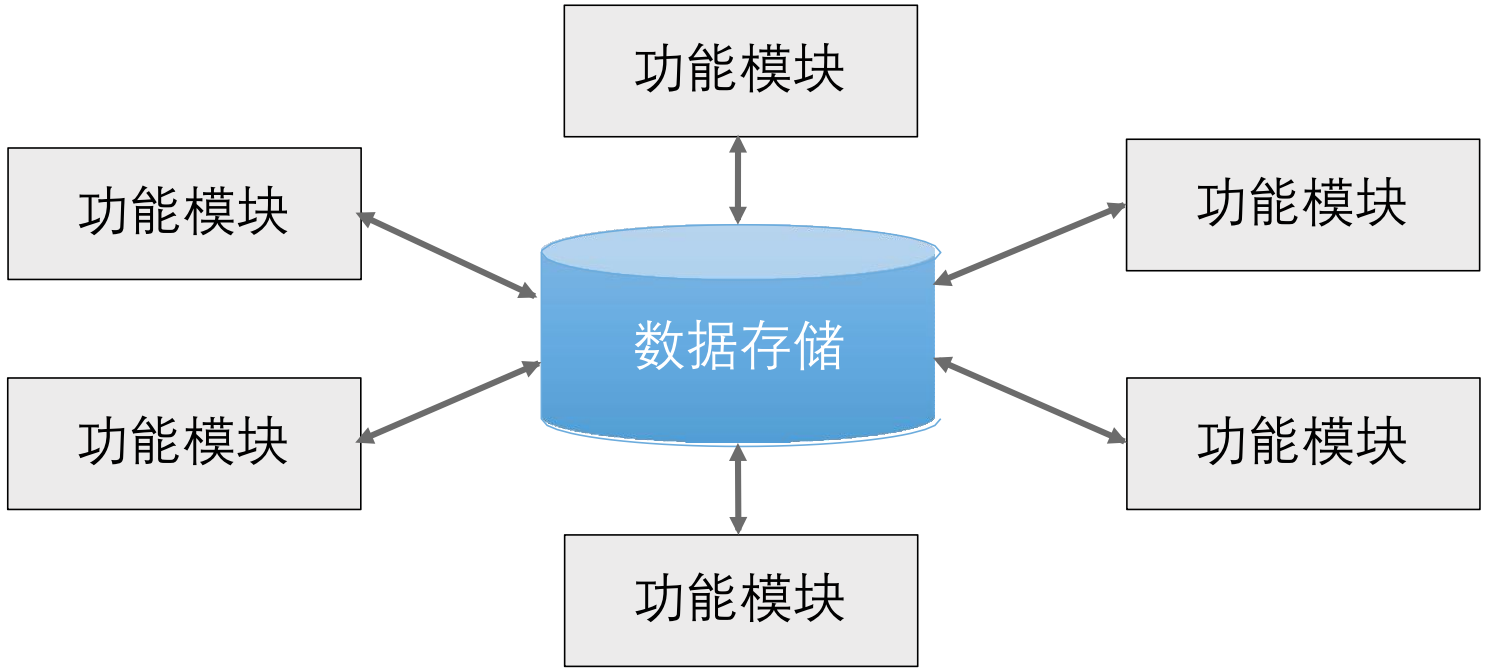
以数据为中心的体系结构风格示例

- 例：剪贴板 (Clipboard)
  - 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)；
  - 访问剪贴板的方式：copy & paste.
  - 不同的应用程序通过该区域交换格式化的信息；



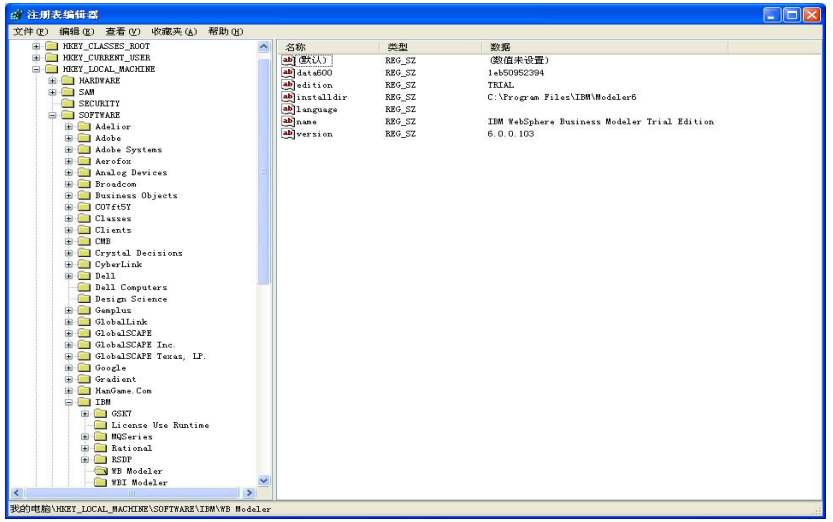
以数据为中心的体系结构风格

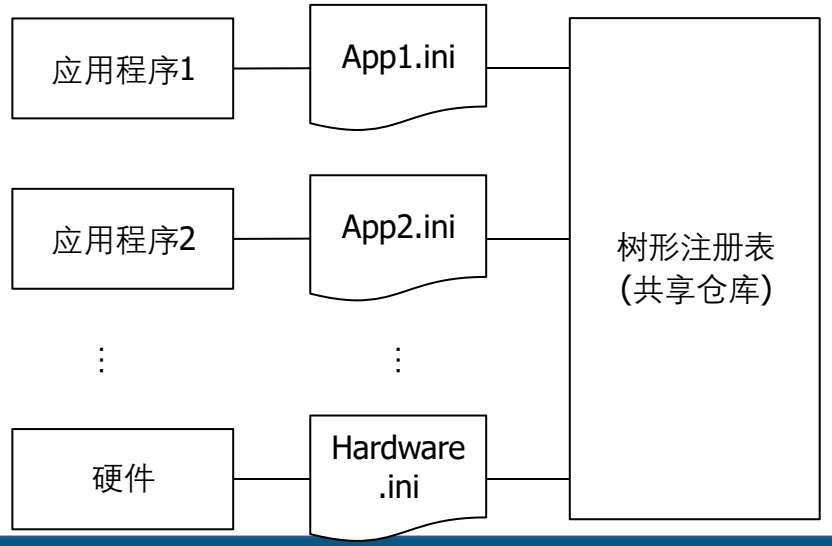
- 以数据为中心的体系结构风格(也称仓库风格)
  - 数据存储位于这种体系结构的中心，其他构件会经常访问该数据存储，并对存储中的数据进行更新、增加、删除或者修改。



以数据为中心的体系结构风格示例

- 例：注册表(Windows Registry)
  - 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。



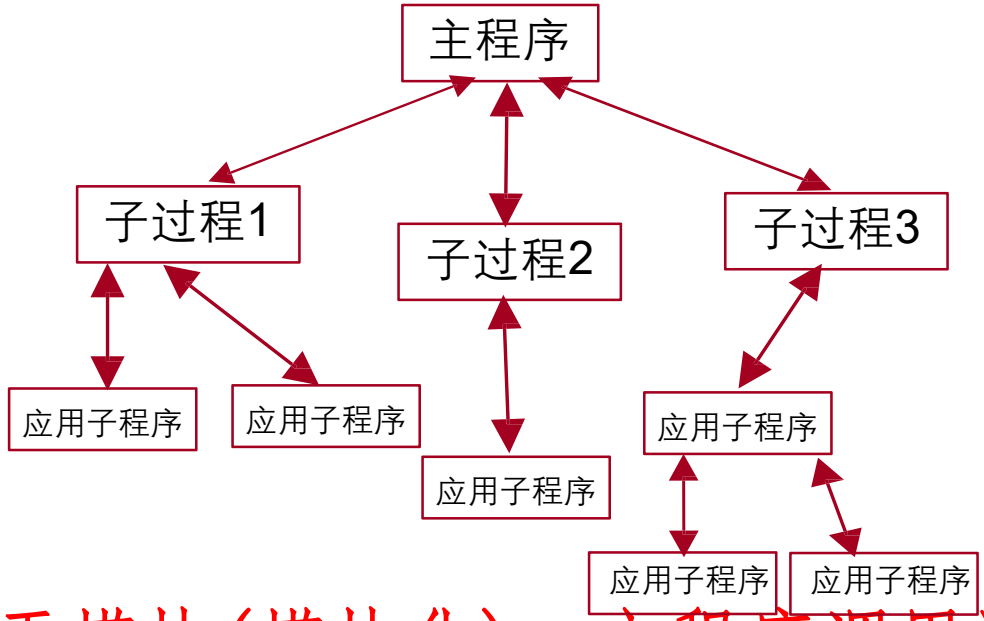


经典体系结构风格

1. 数据流风格
2. 以数据为中心的风格(仓库)
3. 调用和返回体系结构风格
4. 面向对象体系结构风格
5. 层次体系结构风格
   - 客户机-服务器（C/S）
   - 浏览器-服务器（B/S)
   - C/S+B/S混合模式

调用和返回体系结构风格

- **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



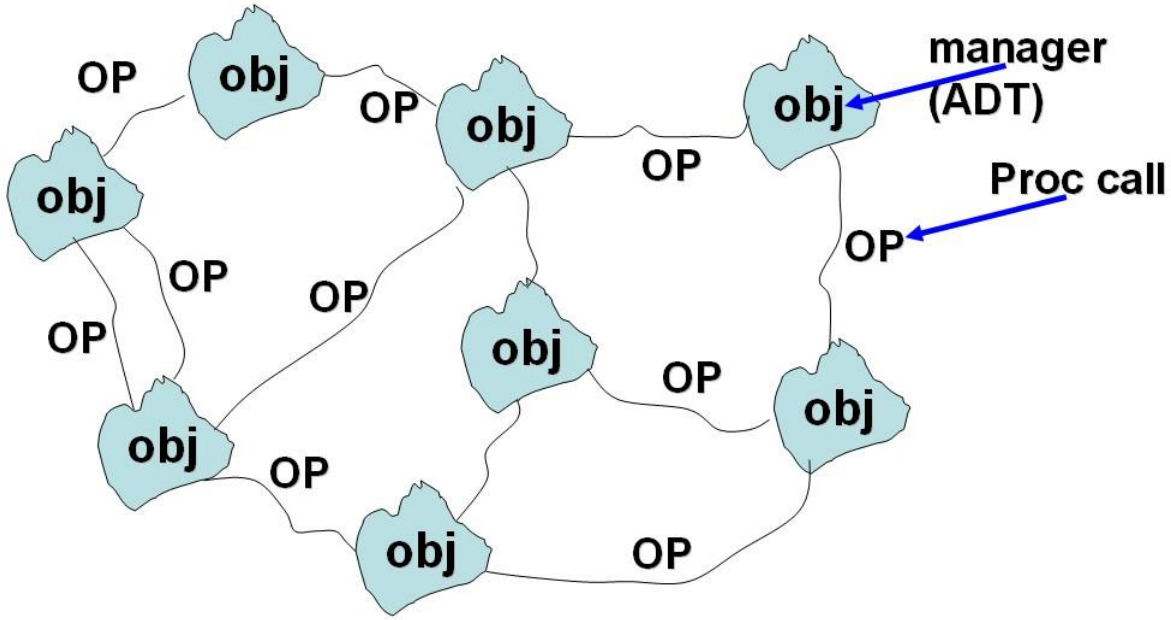
- 本质：**将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能。**

经典体系结构风格

1. 数据流风格
2. 以数据为中心的风格(仓库)
3. 调用和返回体系结构风格
4. 面向对象体系结构风格
5. 层次体系结构风格
   - 客户机-服务器（C/S）
   - 浏览器-服务器（B/S)
   - C/S+B/S混合模式

面向对象体系结构风格

- 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(Abstract Data Type)；
- 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
  - 连接件：类之间通过函数调用、消息传递实现交互

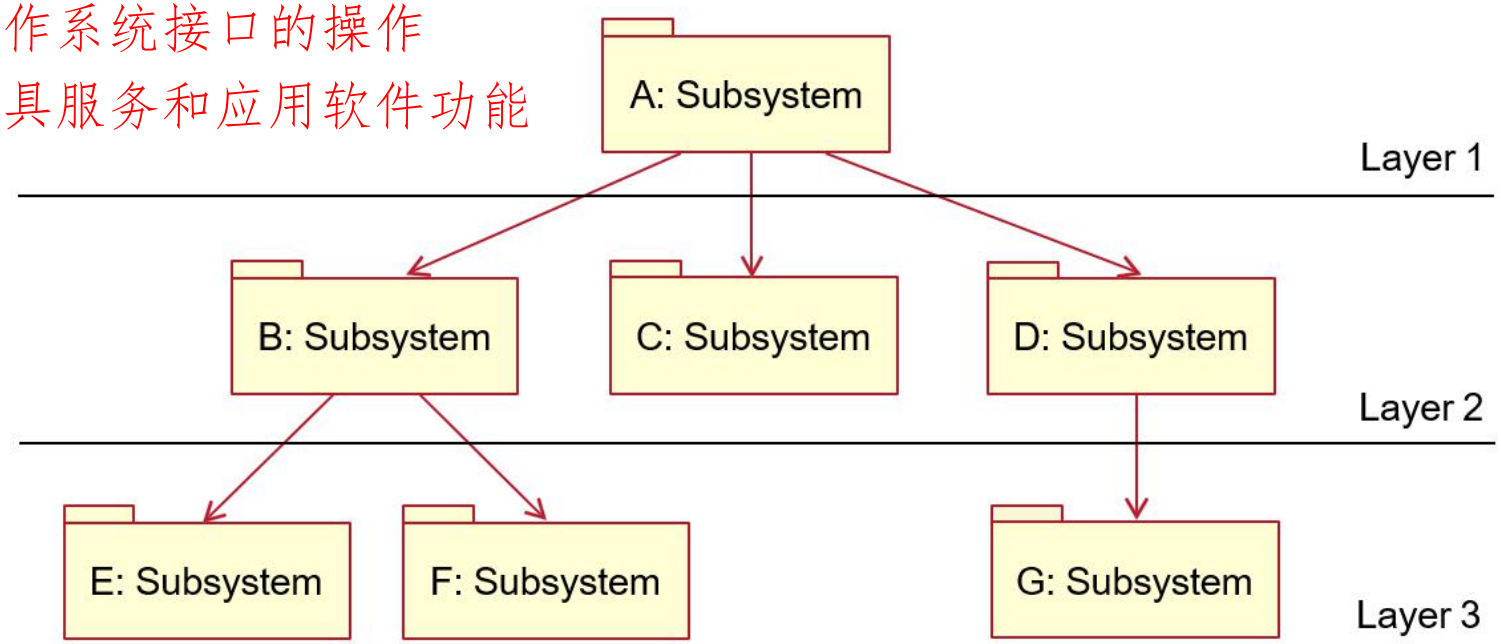


经典体系结构风格

1. 数据流风格
2. 以数据为中心的风格(仓库)
3. 调用和返回体系结构风格
4. 面向对象体系结构风格
5. 层次体系结构风格
   - 客户机-服务器（C/S）
   - 浏览器-服务器（B/S)
   - C/S+B/S混合模式

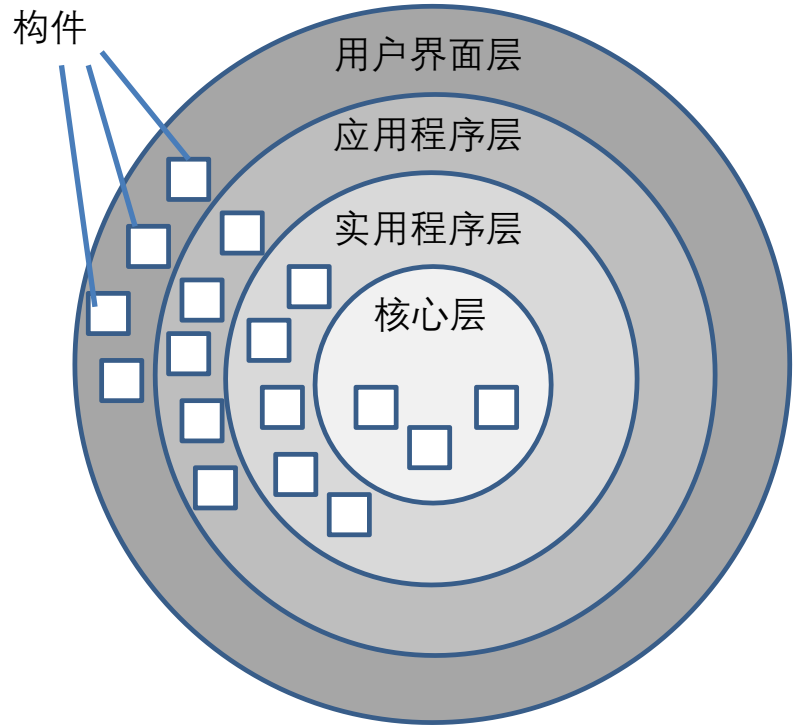
层次体系结构风格

- 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
  - 在外层，构件完成建立用户界面的操作
  - 在内层，构件完成建立操作系统接口的操作
  - 在中层，提供各种实用工具服务和应用软件功能



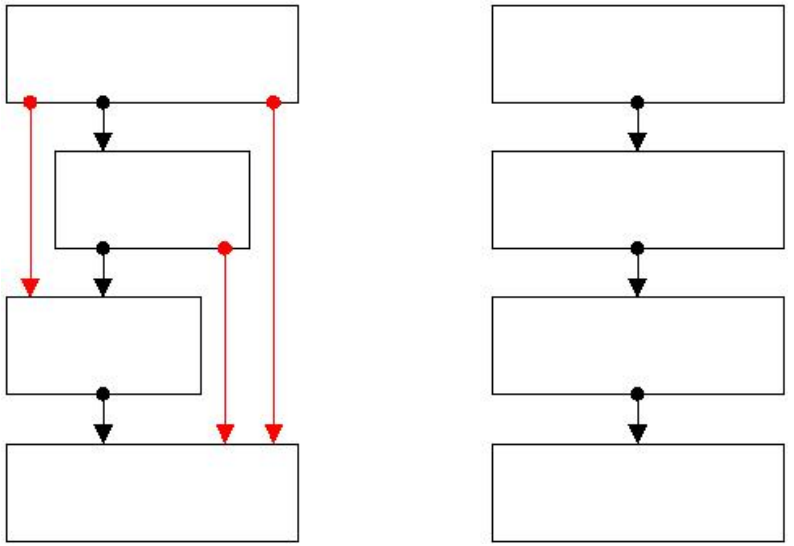
层次体系结构风格

- **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。
  - 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。
- **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；



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



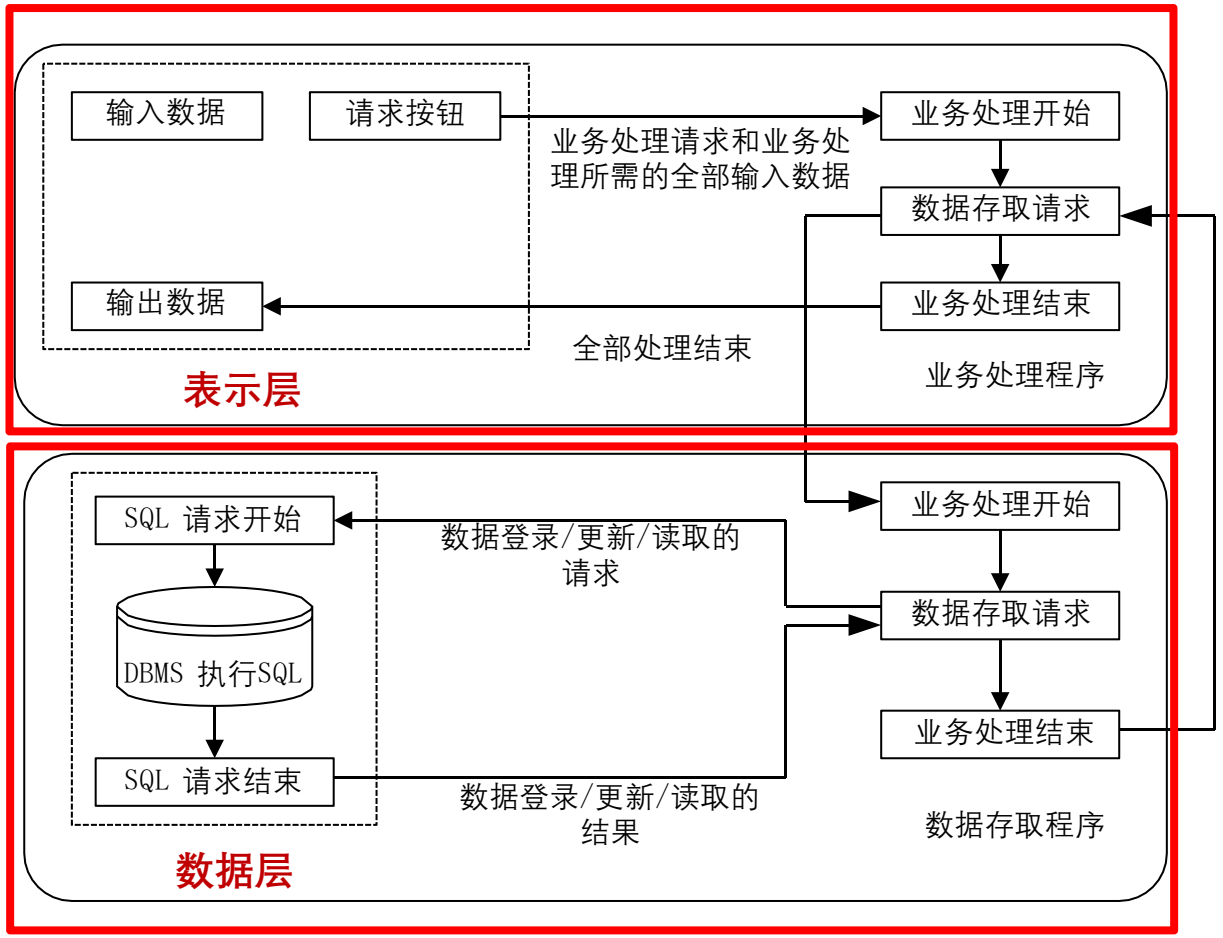
层次体系结构风格

1) 客户机/服务器(Client/Server，C/S)：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)：用户交互、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。

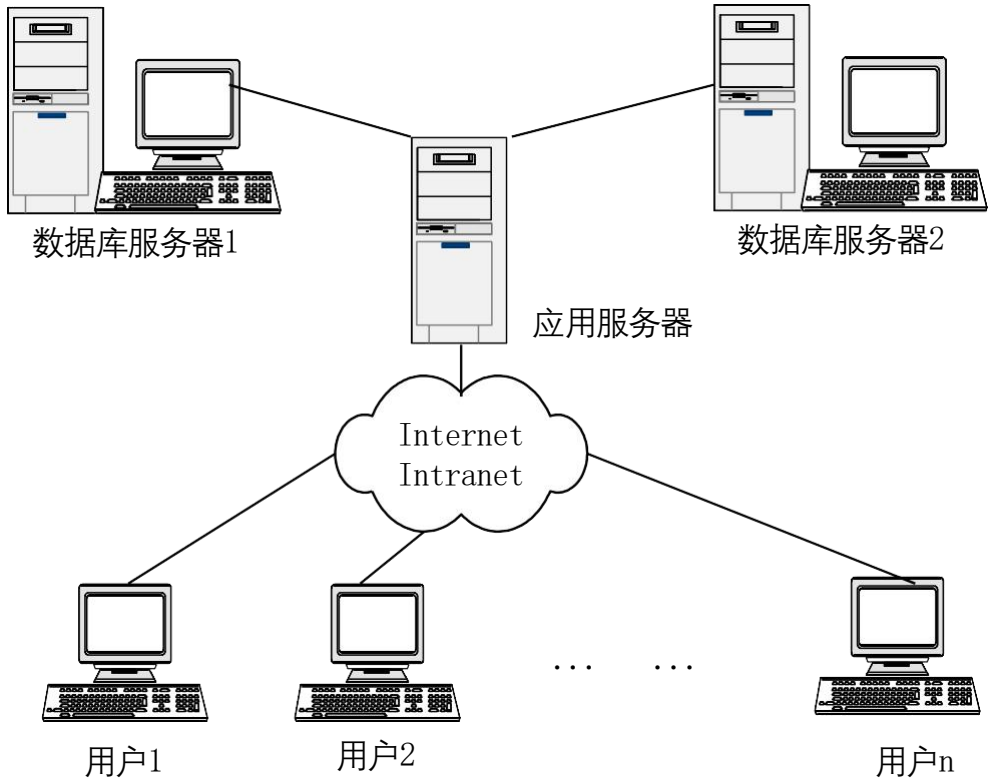
层次体系结构风格

客户机/服务器
(Client/Server)



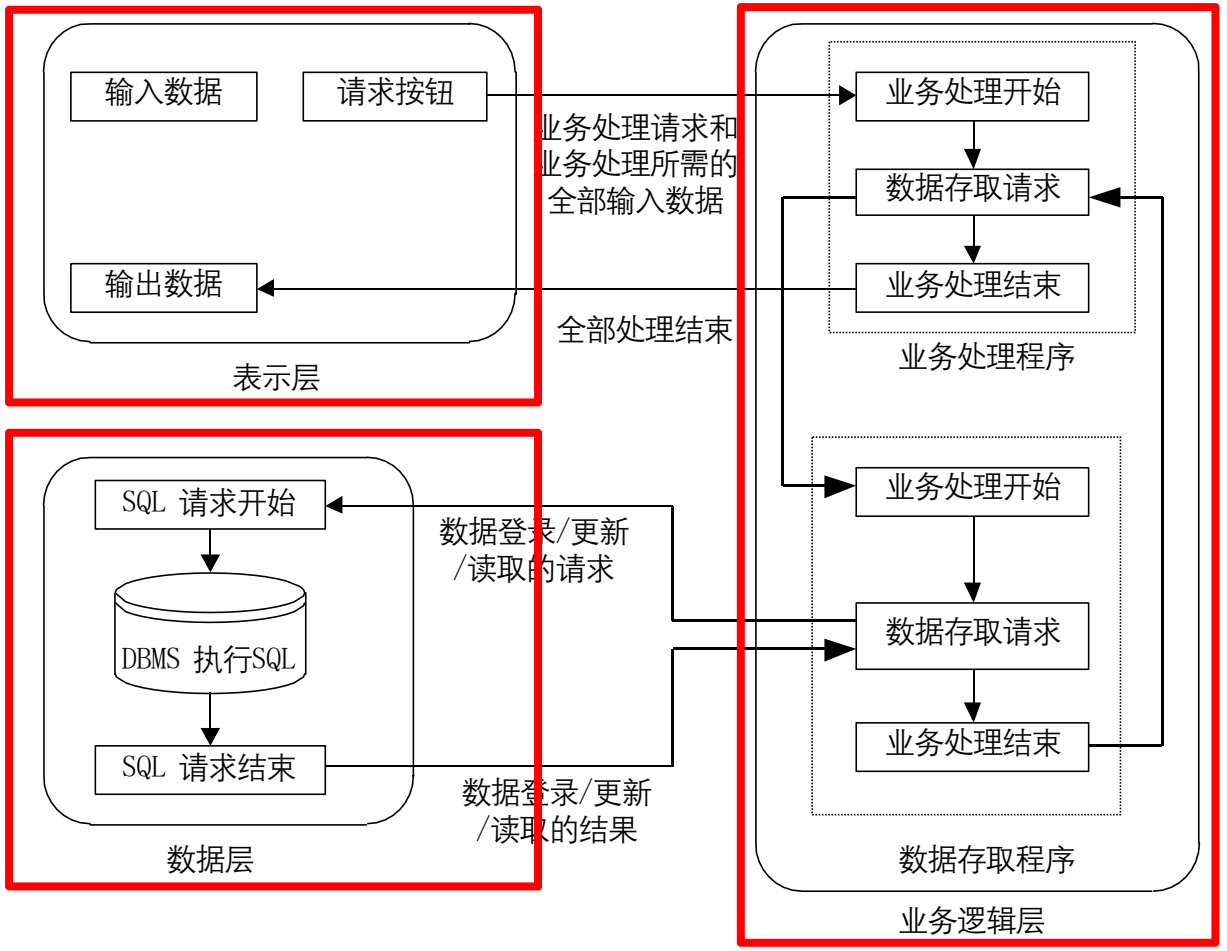
层次体系结构风格

- 三层客户机/服务器(Client/Server，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
  - 表示层：用户界面—界面设计
  - 业务逻辑层：业务处理—程序设计
  - 数据层：数据存储—数据库设计



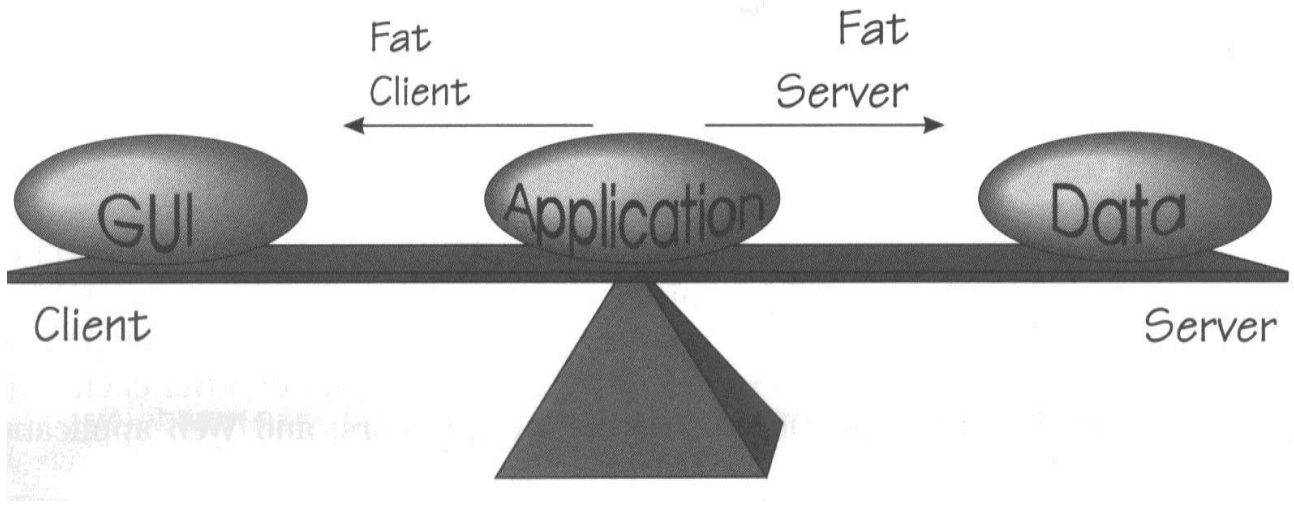
层次体系结构风格

三层客户机/服务器(Client/Server, C/S)示例



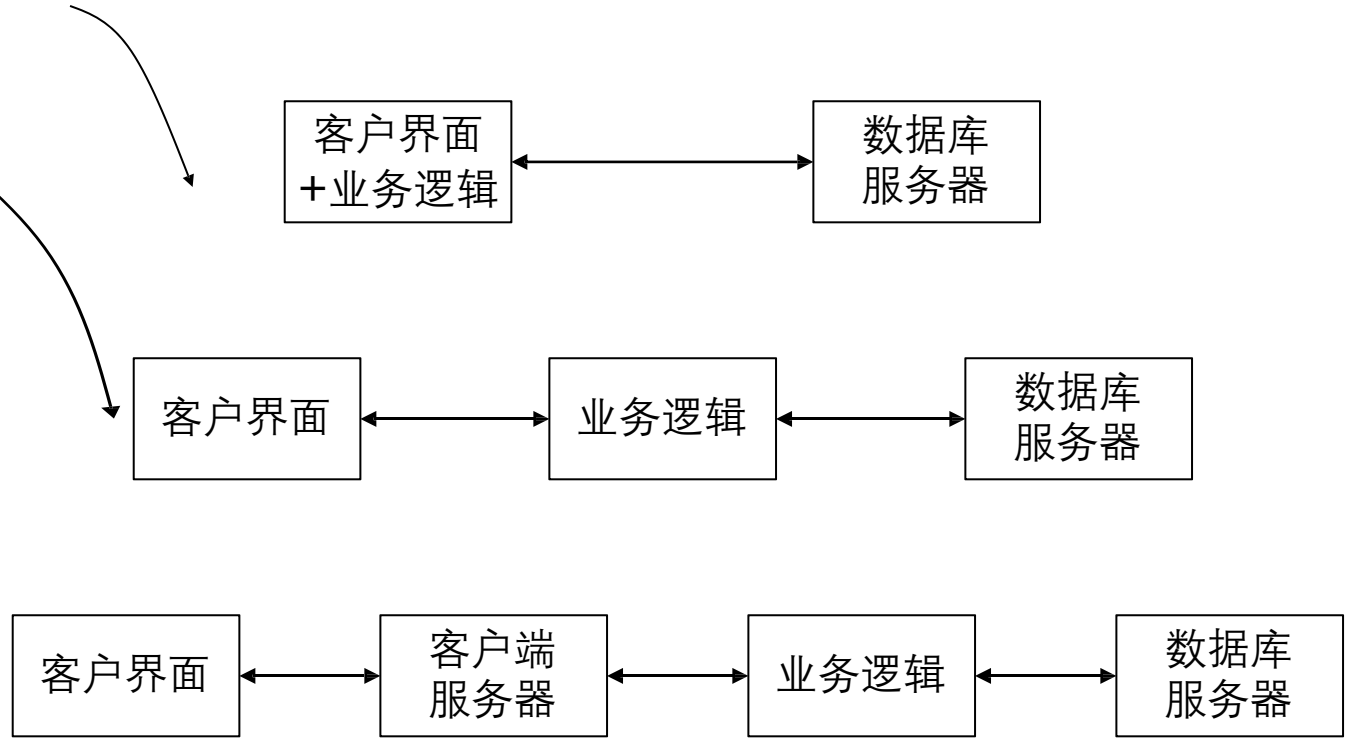
层次体系结构风格

- **胖客户端与瘦客户端：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



层次体系结构风格

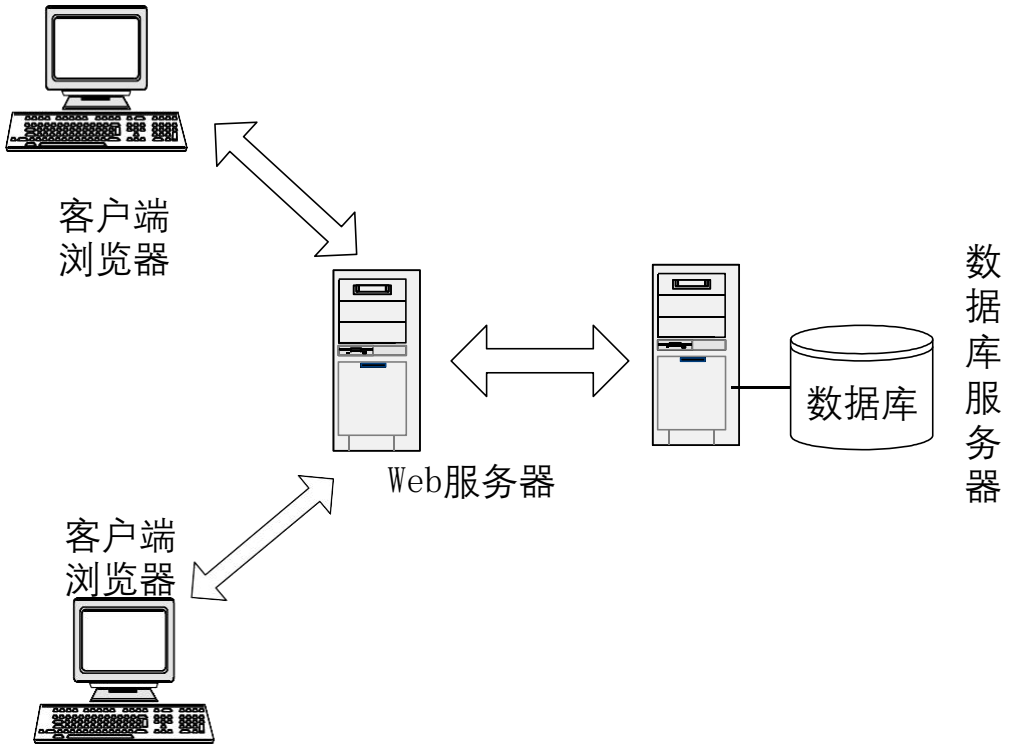
- **“客户机-服务器”结构的发展历程：**
  - 两层C/S（仓库体系风格）
  - 三层C/S
  - 多层C/S



层次体系结构风格

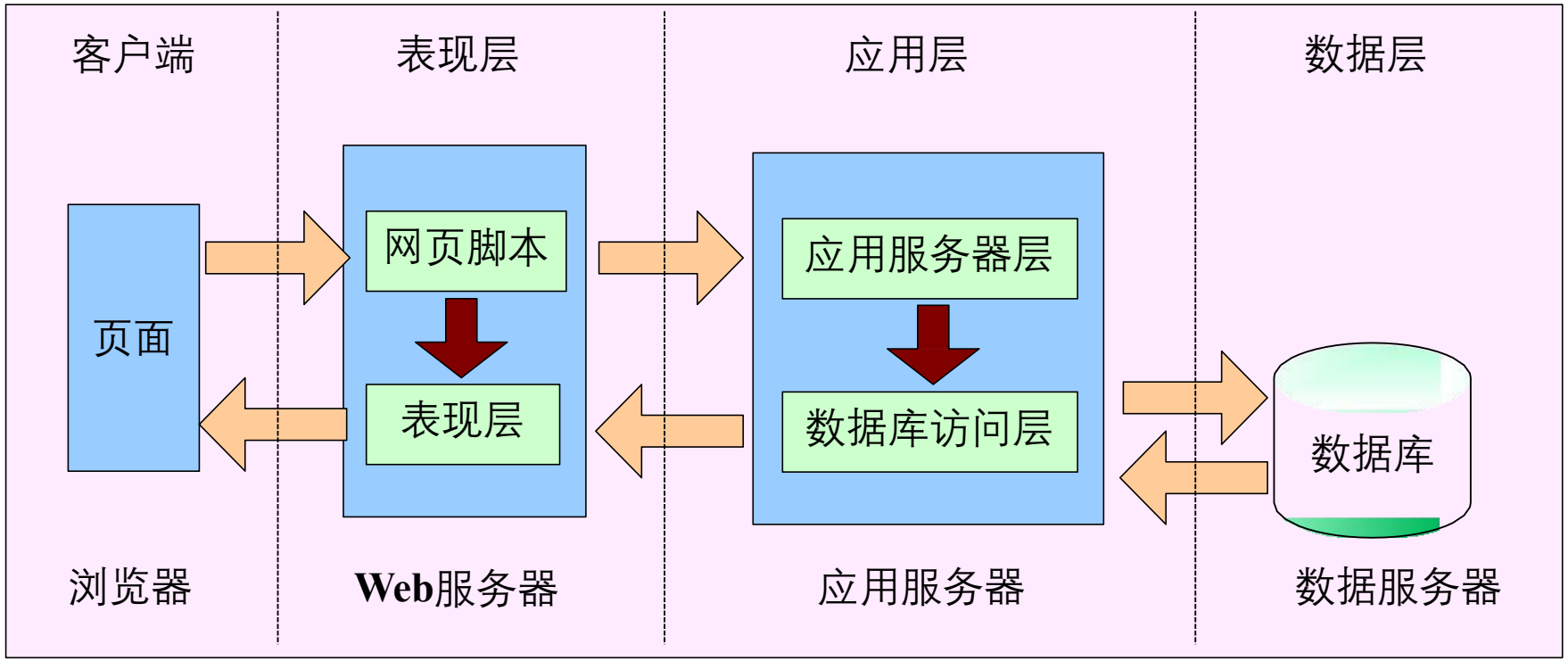
2) 浏览器/服务器(Browser/Server)是四层C/S风格的一种实现方式 。

- 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



层次体系结构风格

- 浏览器/服务器(Browser/Server)示例



层次体系结构风格

- 浏览器/服务器(Browser/Server)：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。

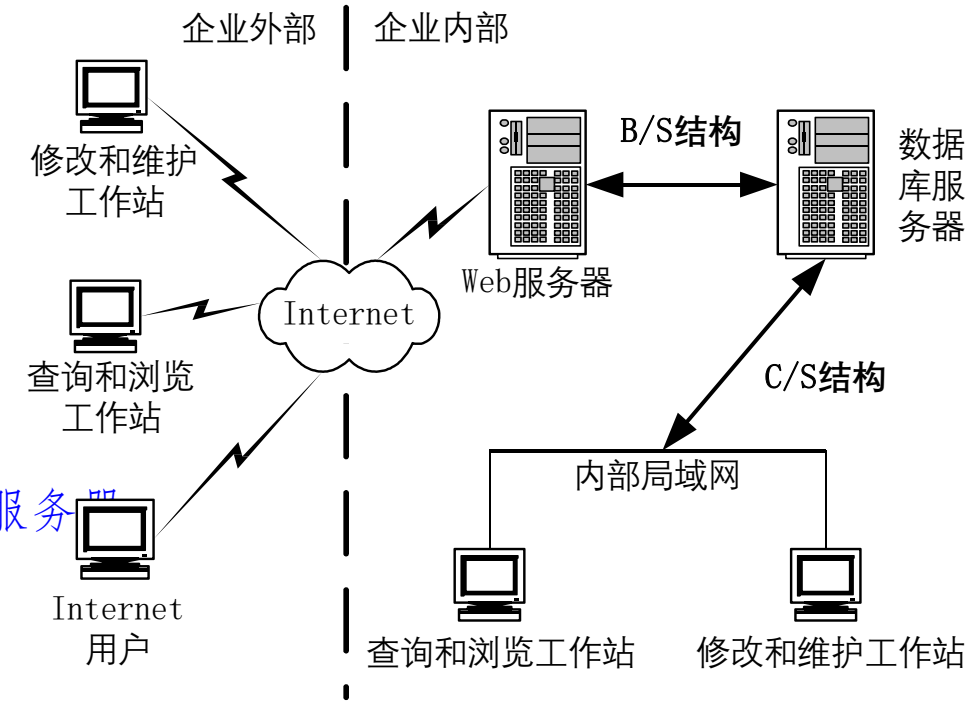
- B/S成为真正意义上的“瘦客户端”，从而具备了很高的稳定性、延展性和执行效率。
- B/S将服务集中在一起管理，统一服务于客户端，从而具备了良好的容错能力和负载平衡能力。

层次体系结构风格

3) 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；



小结

- 软件体系结构要素
- 软件体系结构风格

- 下一节：类/数据建模与设计