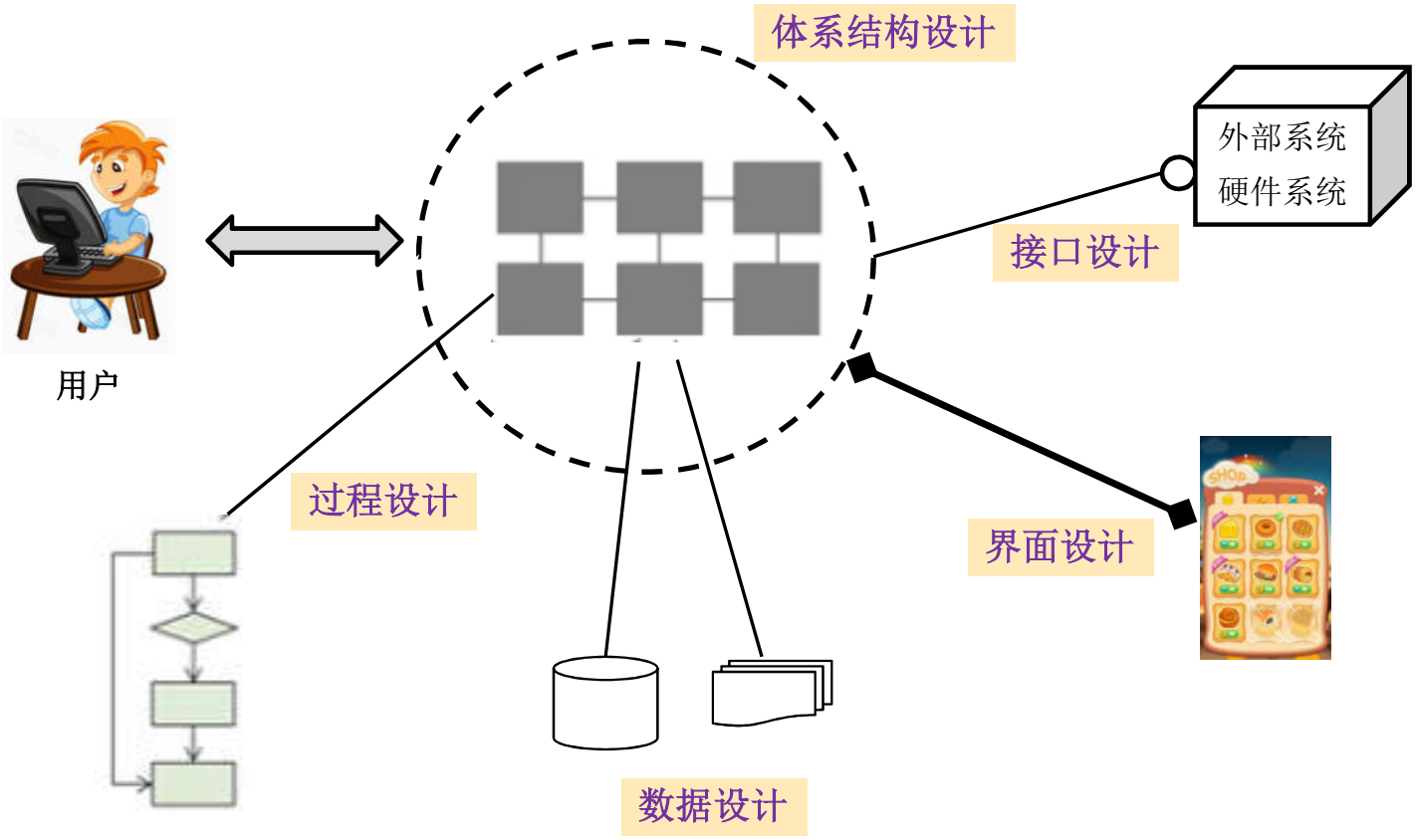
第3章 软件设计

- 软件设计概述
- 软件体系结构设计
- 模块化设计
- 数据库设计
- 界面设计
- 软件设计评审

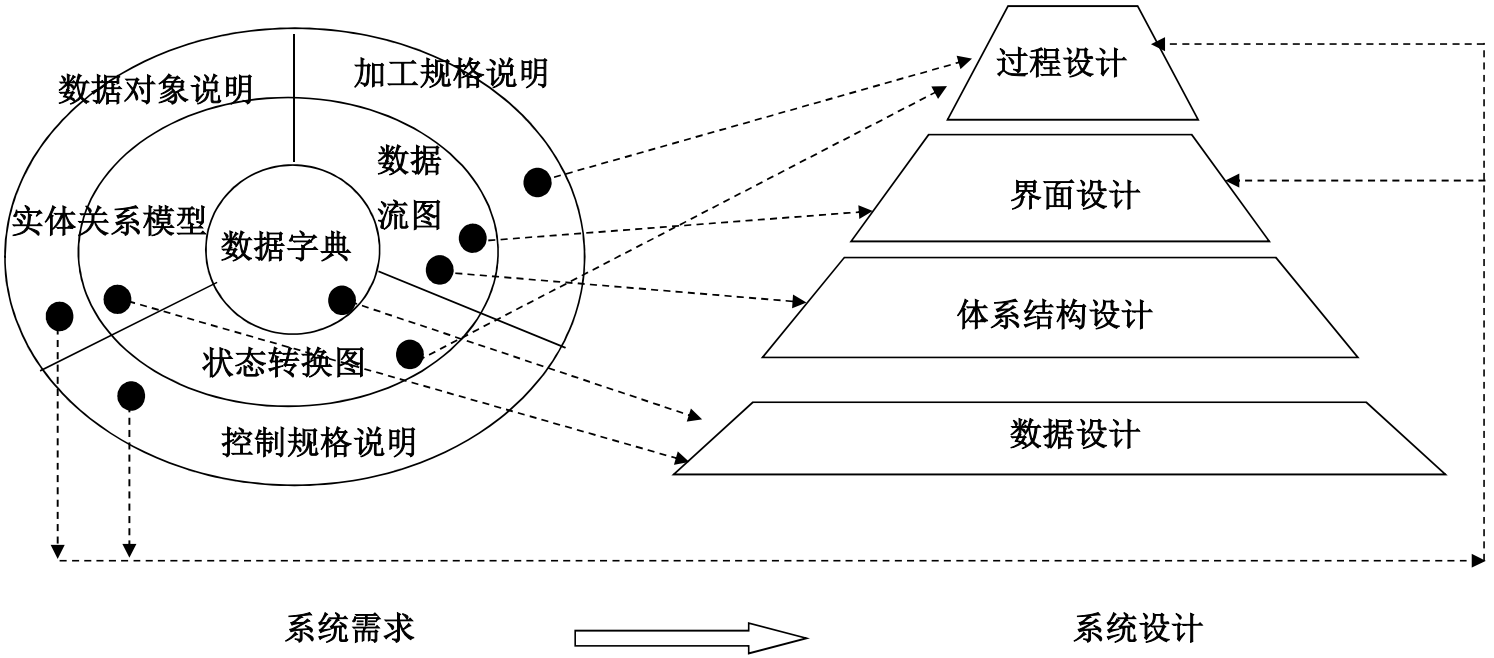
软件设计概述

软件设计阶段回答：系统“如何做”这一问题。



软件设计概述

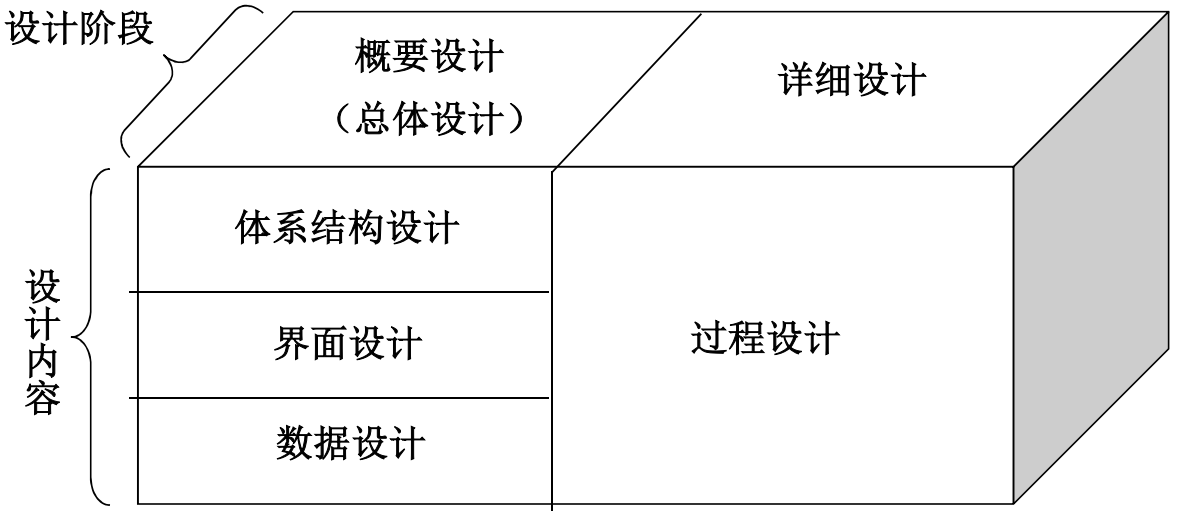
软件设计是软件开发的关键步骤，直接影响软件质量。



软件设计概述

软件设计的任务

设计任务：将需求阶段获得的需求说明（模型）转换为可实现的逻辑系统。



软件设计的目标是构造一个高内聚、低耦合的软件模型。

软件体系结构设计

软件系统的复杂性

【例1】编写一个程序，该程序统计在一个文本文件中，每个词出现的频率。

要实现此功能，基本分析如下：

1. 什么是词？如何分词？（英文 / 中文）
2. 分词后如何存储每个不同的词？链表？数组？栈？
3. 如何查找、统计？顺序查找，还是先按照某种规定排序在查找？
4. 读文本文件。如何读？读入内容以什么结构存储，以利于后续的实现？
5. 统计的词频如何保存？文件？数据库？还是……

词库设计（文件 / 数据库）

数据结构设计

算法设计

软件体系结构设计

软件系统的复杂性

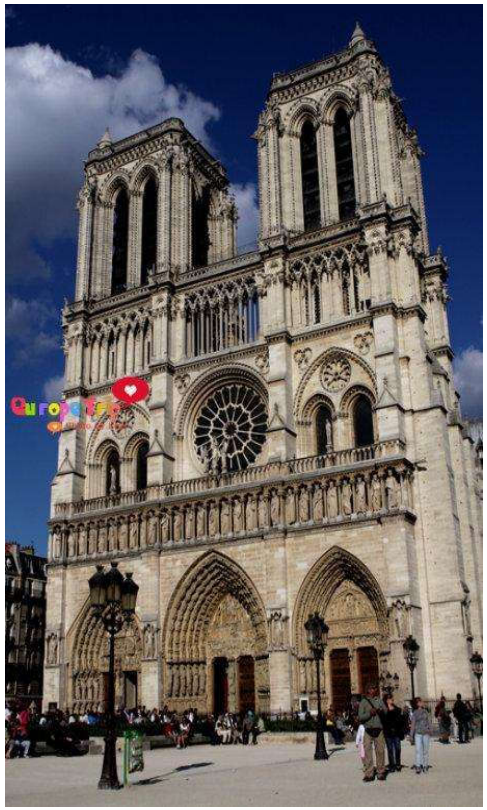
【例2】编写一个Web检索系统，该系统在收集、整理Web信息资源以后，用户能检索到相关的网站或网页信息。

基本的处理过程是：

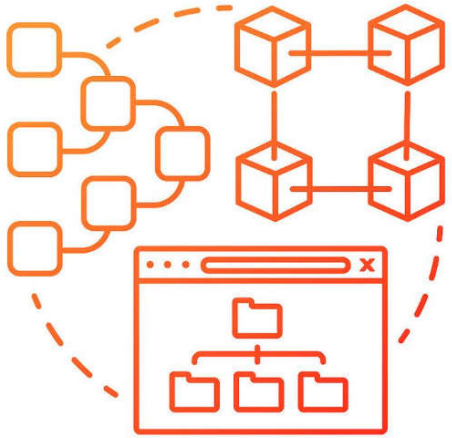
1. 收集Web网页。 → 如何收集？
2. 整理收集后的Web网页。 → 整理什么？如何整理？
3. 检索服务。 → 如何进行检索？

软件体系结构设计

从“建筑风格”谈起



软件体系结构设计



软件体系结构也有“风格”：

- 提供软件架构师能预期的体系结构描述。例如提到浏览器/服务器（B/S）模式，多层架构、数据库存储、客户端、逻辑服务器等概念和模型等的一系列描述就展现在眼前。
- 数据结构、文件组织、系统结构体现了软件设计的早期抉择。这些抉择将极大地影响着后续的软件开发，影响着软件产品的最后成功。

软件体系结构的设计，要考虑：

- 对整个系统的结构和行为进行抽象，使得系统的整体设计更容易理解。
- 考虑构造大型软件系统的、有效的可重用方法，用以实现系统级的复用。

软件体系结构设计

软件体系结构设计是软件设计的早期活动，它提供了一套关于数据、行为、结构的指导性框架。

- 以数据为中心的数据仓库模型
- 客户端/服务器模式的分布式结构
- 层次模型
- 控制模型

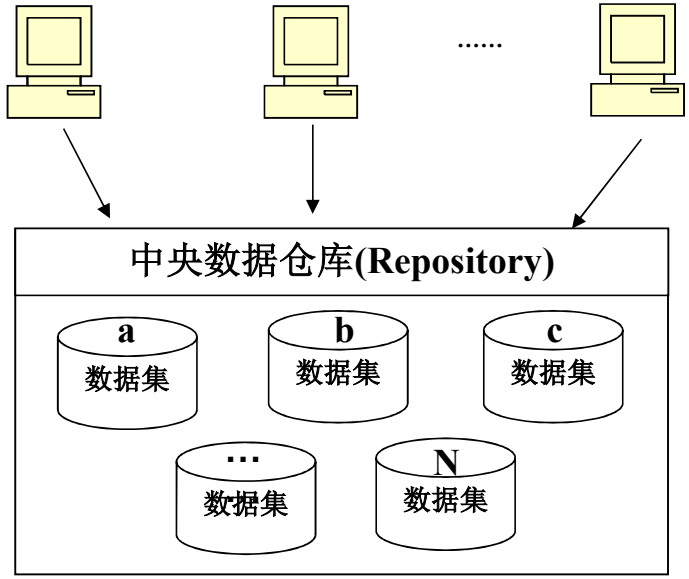
软件体系结构设计

以数据为中心的数据仓库模型，独立提供数据服务的封闭式数据环境。它不单独集成到某一应用系统中，而是为具体的应用系统提供服务。这些服务既有通用的公共服务，也有专门设计的领域服务。

构件：数据、功能模块、系统服务……

连接方式：请求——返回机制

拓扑结构：以数据为中心的层次结构



软件体系结构设计

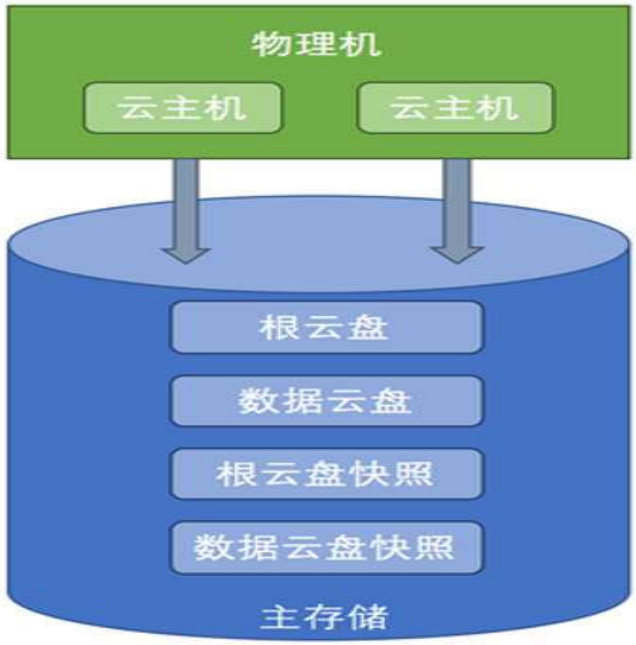
数据（仓）库集群技术

现在随着大数据、人工智能、5G移动应用等技术与需求的发展，同时自媒体、短视频等载体的火爆，互联网上的数据量呈几何级数增长。当今数据库所需要解决的问题不仅仅是记录软件系统中的信息（这仍很重要），还面临着如下挑战：

- 数据库性能：通过增加服务器的方式，使其提供更高的吞吐量，但需考虑：负载均衡和数据一致性。
- 数据库功能：数据（仓）库接口的新增与完善，但需考虑：对软件系统修改所带来的影响。
- 数据稳定：当数据（仓）库中数据的丢失、损坏、故障时，能否保证软件系统仍能正常运行。

数据（仓）库集群技术——Share Disk架构

Share Disk 架构是通过多个服务器节点，来共享存储实现数据（仓）库集群。

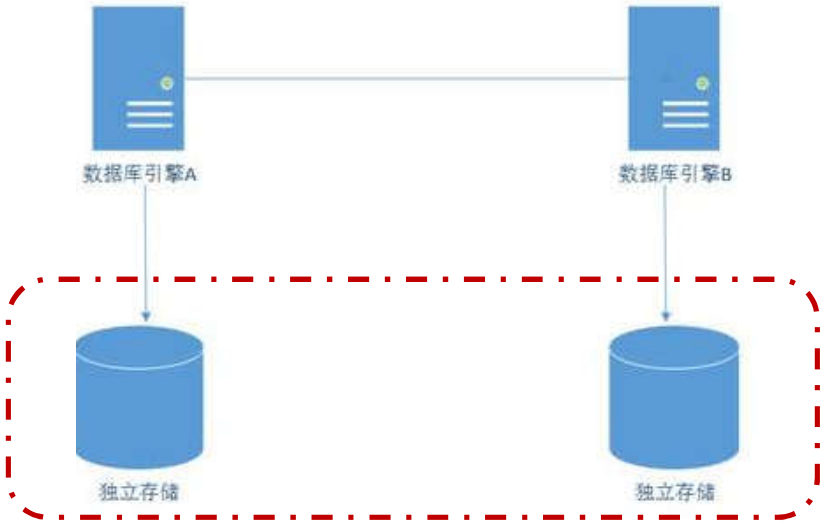


两台服务器最基本的Share Disk 架构

数据（仓）库集群技术——Share Nothing架构

- Share Nothing 架构中的每个节点都拥有自己的内存和存储，都保留数据的完整副本。
- Share Nothing架构需要考虑到“负载均衡”的问题。

最基本的Share Nothing架构

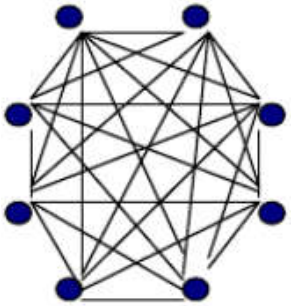


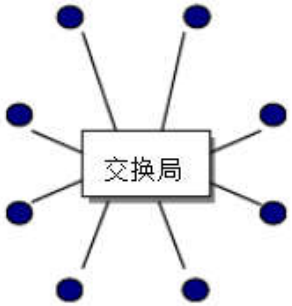
客户 / 服务 方式——起源

贝尔1877年建立贝尔电话公司（AT&T前身）。而那时的电话需要一对一对地卖，之后用户自己在两部电话之间拉一根电话线。如果一个用户与与多个用户通话，必须自己拉 *N* 根电话线。结果就成了下图。

两年之后，贝尔电话公司开办了第一家电话交互局。安装电话的用户，只需要把自家电话线接入到电话局即可。当打电话时，就手摇电话曲柄，交换局办公室电话就会响起，操作员就能根据你的要求接通对方的电话。

这样，多用户间的电话连接转换为了集中交互式模型。后来有了在不同城市间打电话的长途电话需求，建立了二级交换局。





软件体系结构设计

客户 / 服务 方式的起源

- 1904年1月2日，是清政府钦准的中国第一个官办电话局，当时安装了一台100门磁石式人工电话交换机。

软件体系结构设计

客户端 / 服务器模式（C/S）的分布式结构

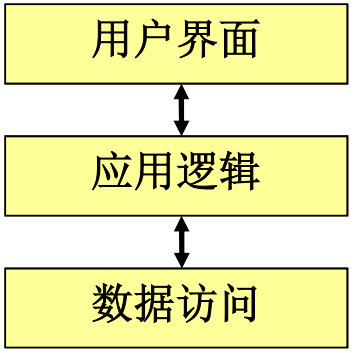
C/S结构是一种分布式模型，采用发请求、得结果的模式：

- 客户机　向服务器发出请求(数据请求、网页请求、文件传输请求等等)，
- 服务器　响应请求，进行相应的操作，将结果回传给客户机，客户机再将格式化后的结果呈现给用户。

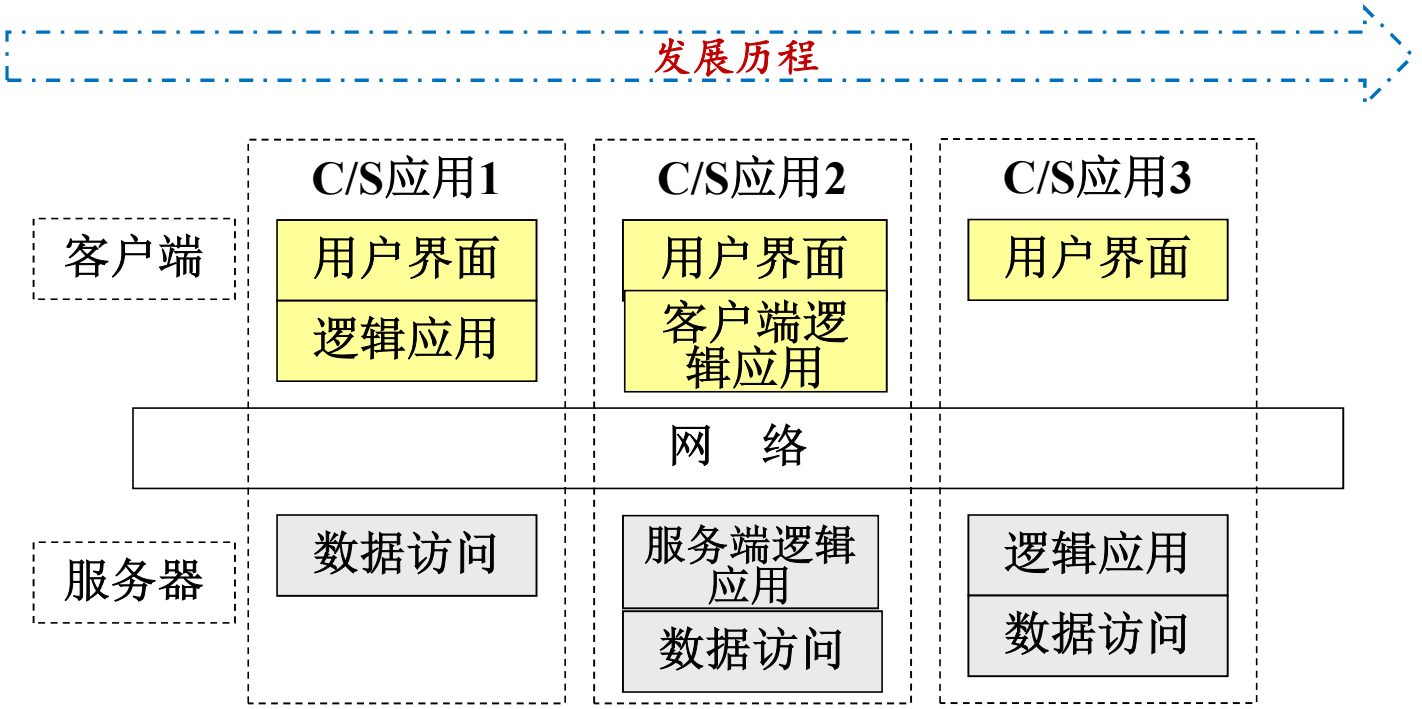
构件：浏览器、客户端、中间件、服务器端、……

连接方式：请求——响应机制

拓扑结构：网状结构



完整的应用包含三个相对独立的逻辑部分，而两层的C/S结构只有两个端应用。应用逻辑应该映射到哪一端上呢？

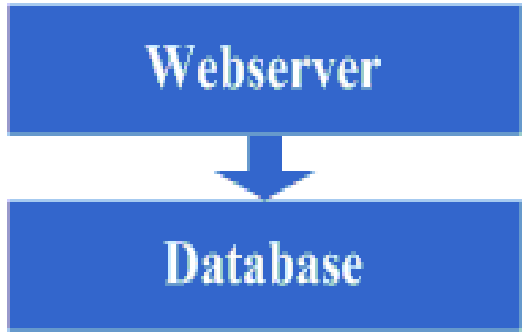


应用逻辑层的映射情况

软件体系结构设计

层次模型：一个网站架构的演化模型（1）——服务器与数据库分离

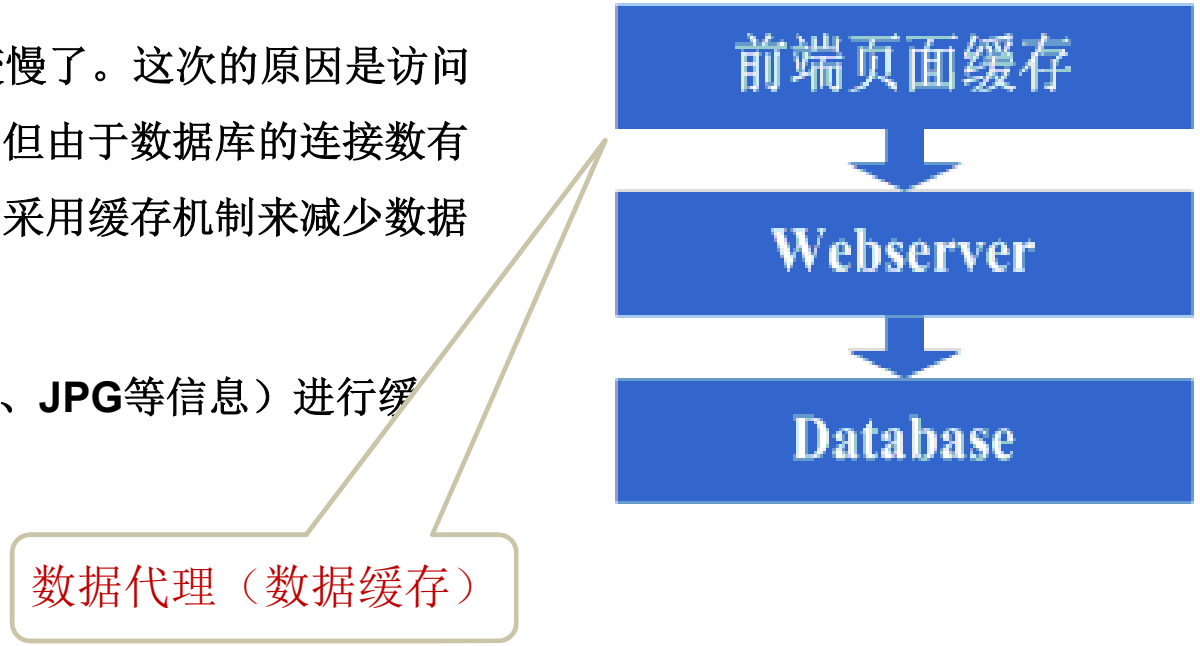
- 网站的初步建立，并随着网站的运营和访问量的不断提升，发现系统的压力越来越大，响应速度越来越慢。特别是数据库和系统应用的相互影响。
- 第一步演变阶段：将系统应用和数据库从物理上分离，变成了两台机器。虽然这一改变没有技术含量，但系统却又恢复到之前的响应速度，并且支撑了更高的访问量，且不会因为数据库和应用造成互相影响。



软件体系结构设计

层次模型：一个网站架构的演化模型（2）——增加页面缓存

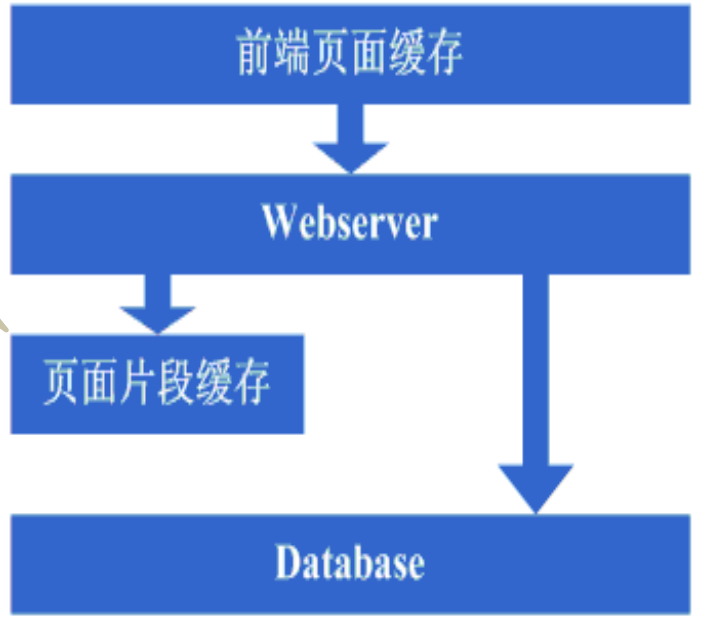
- 随着访问量的再次增加，导致响应速度又变慢了。这次的原因是访问数据库的操作太多，导致数据连接竞争激烈。但由于数据库的连接数有限，否则数据库服务器压力会很大，因而考虑采用缓存机制来减少数据库连接资源的竞争和对数据库连接的压力。
- 考虑将系统的静态页面（例如HTML、CSS、JPG等信息）进行缓（例如squid或将页面静态化等方案）。



相关的知识体系：前端页面缓存技术，squid的实现方式，缓存的失效算法等。

层次模型：一个网站架构的演化模型（3）——增加页面片段缓存

将由jsp等动态生成的页面分块后，将静态部分缓存。



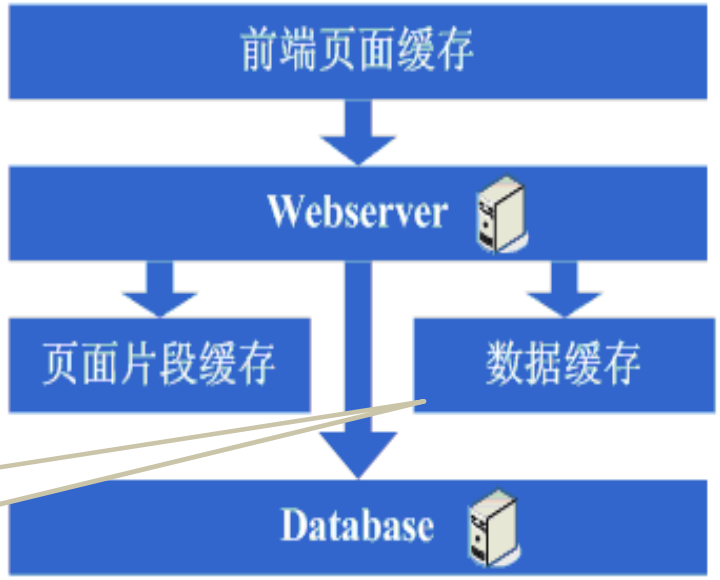
- 随着访问量的增加，系统又开始变慢了。从静态缓存中得到了启发，将动态页面里相对静态的部分也缓存起来（例如采用类似**ESI**等技术）的片段缓存策略。

相关的知识体系：页面片段缓存技术，掌握**ESI**的实现方式等。

软件体系结构设计

层次模型：一个网站架构的演化模型（4）——数据缓存

- 随着访问量的增加，系统又开始变慢。这次是由于系统中存在一些重复获取数据信息的地方，如获取用户信息等。
- 考虑将数据信息也进行缓存。将数据缓存到本地内存，改变完毕后，系统的响应速度恢复，数据库的压力也再度降低。



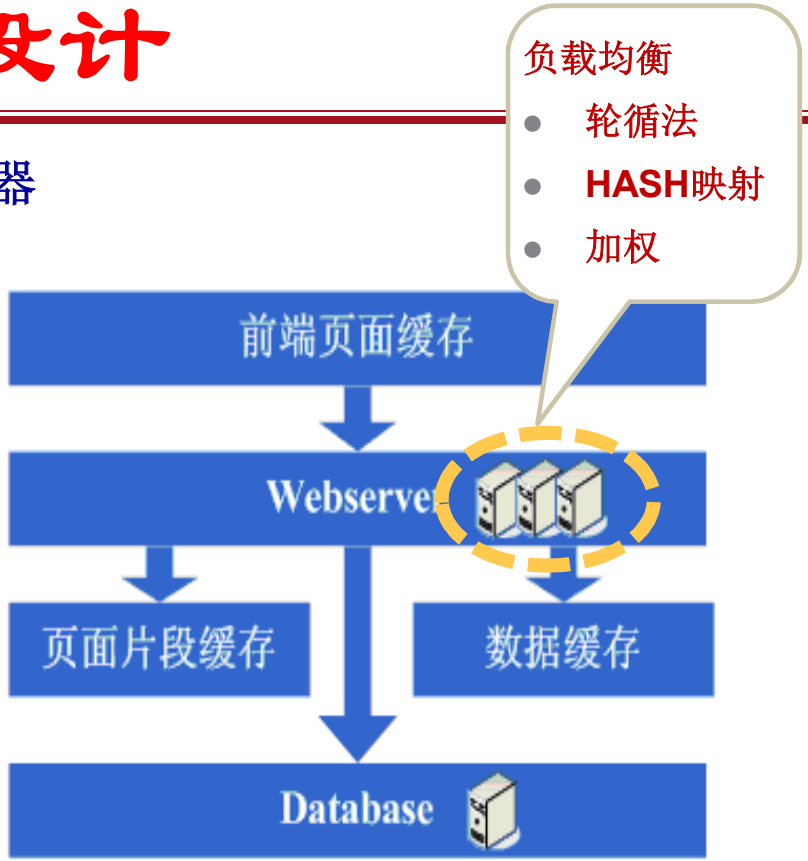
FIFO、LFV（最少访问）、MS（最大存储空间）等算法，来缓存数据。

相关的知识体系：缓存技术（如Map数据结构）、缓存算法、所选用的框架本身的实现机制等。

软件体系结构设计

层次模型：一个网站架构的演化模型（5）——增加Web服务器

- 随着访问量的增加，系统又开始变慢。考虑直接增加Web服务器，这也是为了避免单台Web服务器故障时无法相应的问题。
- 但增加一台Web服务器的同时，会遇见一些典型问：如何让访问分配到这两台机器上(负载均衡方案)、如何保持状态信息的同步，如何保持数据缓存信息的同步。

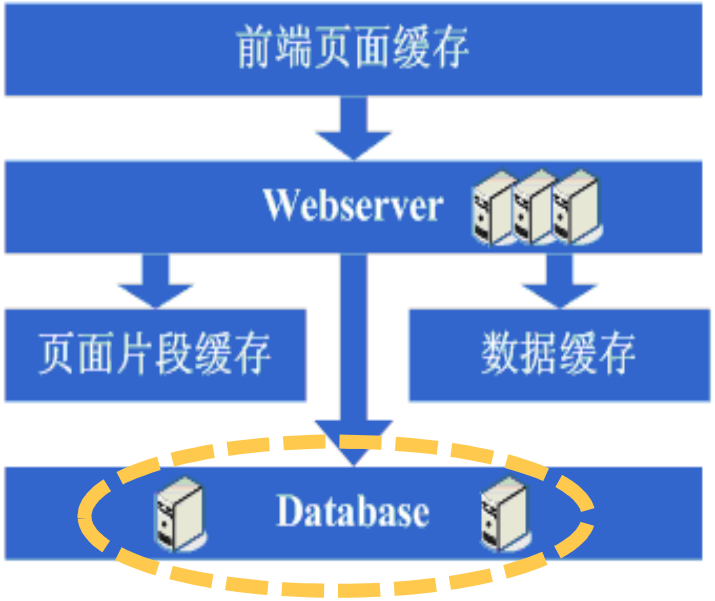


相关的知识体系：负载均衡技术（如硬件负载均衡、软件负载均衡、负载算法等）、主备技术、状态信息或缓存同步技术（如Cookie、状态信息广播等）、共享文件技术、存储等。

软件体系结构设计

层次模型：一个网站架构的演化模型（6）——数据库分库

- 随着访问量的增加，系统又开始变慢。发现数据库写入、更新操作的数据库连接的资源竞争非常激烈，导致了系统变慢，
- 考虑可选的方案有数据库集群或分库策略。集群需要数据库很好的支持，因此考虑采用分库这一较为广泛的策略。
- 分库要对原有程序进行修改。

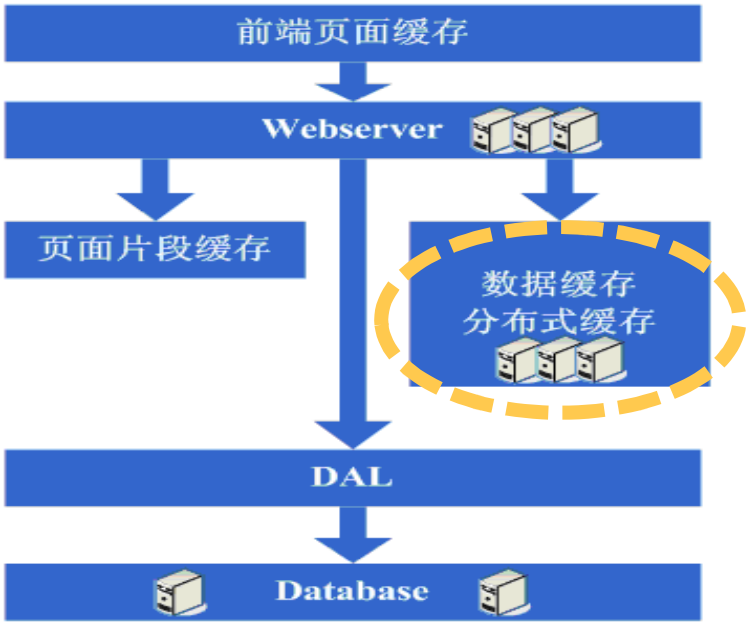


相关的知识体系：需要从业务上做合理划分，以实现分库；同时随着数据量的增大和分库的进行，在数据库的设计、调优以及维护上需要做得更好。

软件体系结构设计

层次模型：一个网站架构的演化模型（7）——分表、DAL和分布式缓存

- 随着访问量的增加，系统又开始变慢。发现是由于分库后查询仍会有些慢，于是按照分库的思想开始做分表的工作，当然，这不可避免的会需要对程序进行一些修改。
- 在这个阶段可能会发现之前的缓存同步方案出现问题，因为数据量太大，导致现在不太可能将缓存存在本地，然后同步的方式，而是需要采用分布式缓存方案。

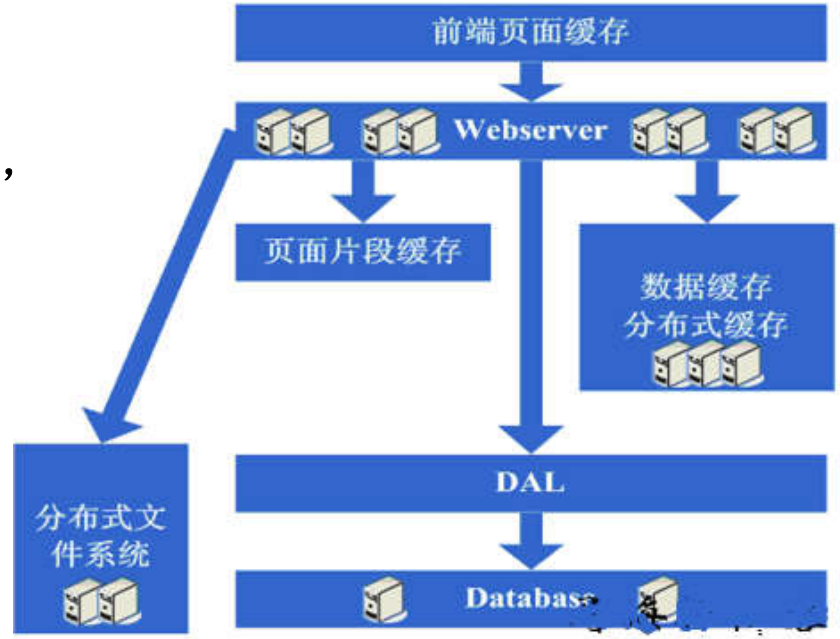


相关的知识体系：分表是在业务上进行划分，技术上主要涉及到动态hash算法等；DAL涉及到比较多的复杂技术，例如数据库连接的管理（超时、异常）、数据库操作的控制（超时、异常）、分库分表规则的封装等。

软件体系结构设计

层次模型：一个网站架构的演化模型（8）——增加更多的服务器

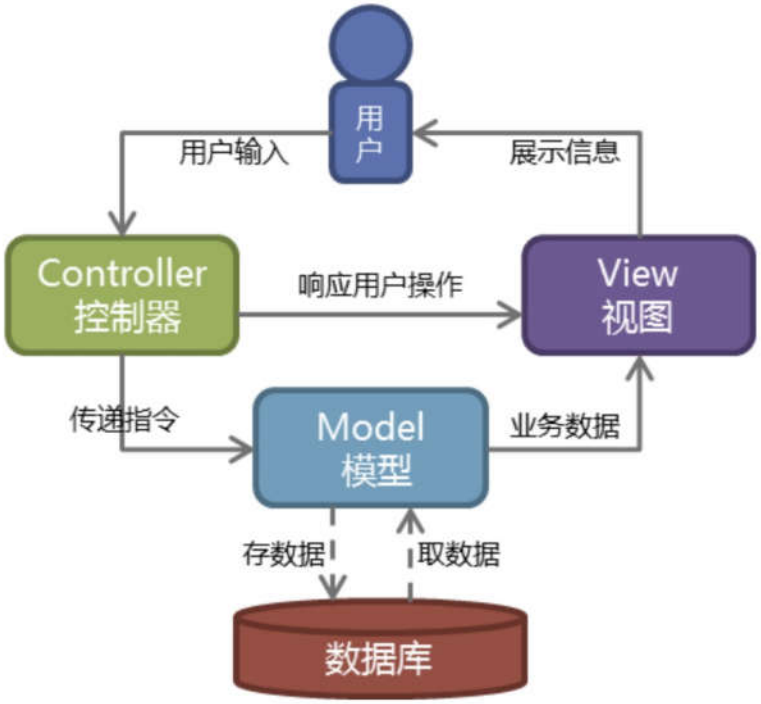
- 随着访问量的增加，系统又开始变慢。这时首先查看数据库，之后查看Web服务器，发现阻塞了很多的请求，而导致排队等待，响应速度变慢。



相关的知识体系：随着机器数的不断增长、数据量的不断增长和对系统可用性的要求越来越高，则需要根据网站的需求来定制不同性质的产品。

软件体系结构设计

MVC模型是“Model-View-Controller”的缩写，意思是“*模型—视图—控制器*”。它是一种软件设计模式，作用是把软件系统划分到模型、视图和控制器等三个框架中去，使得软件逻辑部件可以有效划分，程序设计变得容易。



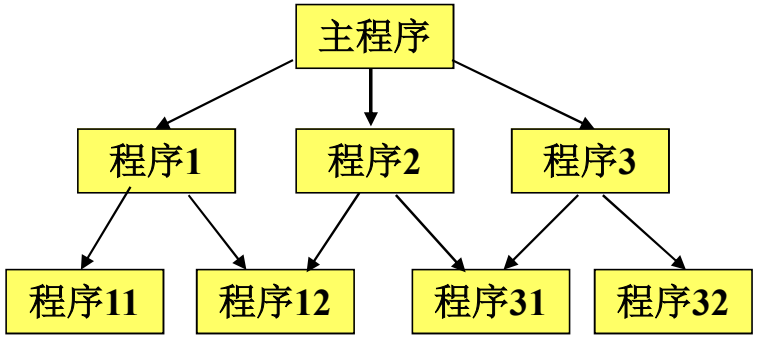
MVC得到了广泛应用，其优势主要在于：

- 一个模型对应多个不同的视图；
- 模型的自包含性；
- 控制层把一个模型和多个不同视图组合在一起，能够完成多种类型的请求；
- MVC分层模式，使得只修改其中某层，就能满足用户新的需求，使系统达到不同的效果；
- MVC利于软件工程的工程化管理。

软件体系结构设计

控制摸型——集中式控制

集中式控制是结构化程序设计的典型模型。它是从功能的角度进行设计，通过分解，形成整个软件系统的“调用——返回”机制。



构件：主程序、程序1……、程序11……

连接方式：调用——返回机制

拓扑结构：层次结构

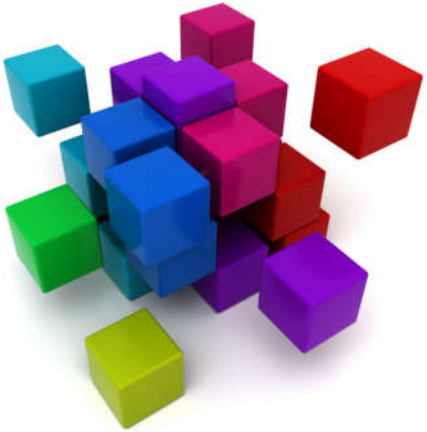
模块化设计

活字印刷术可以称得上是我国古代产品设计领域中优秀的模块化方法。

- ✓ 每个汉字代表一个文字“模块”，具有特定的功能和含义。
- ✓ 语法连接汉字模块的“接口”，通过语法将各文字“模块”连接、构建为最终产品“文章”。

模块化设计

模块化设计的基本思想，就是将一个系统或复杂问题，按照功能拆分为一系列小的系统或简单问题。这些小系统或简单问题得到解决，大系统或复杂问题就能得到解决。



- 减低程序的复杂度
- 提高模块的可靠性和复用性
- 便于管理和控制软件开发过程
- 易于维护和系统扩展

模块化设计

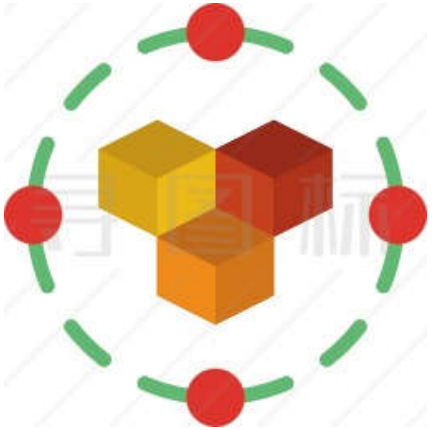
模块是程序语句的集合，它拥有独立的命名，明确的输入、输出和范围。程序设计中的函数、过程、类、库等都可作为模块。模块用矩形框表示，并用模块名称来命名。

模块设计准则：

- 软件模块化与分解
- 信息隐藏
- 模块独立性
- 启发式规则

模块化设计

4. 模块独立性是指软件系统中划分的模块完成一个相对独立的功能，而与其它模块的关联尽量只发生在接口上。因此，独立性是良好设计的关键，是衡量软件质量的重要指标之一。



SD方法提出的定性的度量标准：

- 模块之间的*耦合性*
- 模块自身的*内聚性*

模块化设计

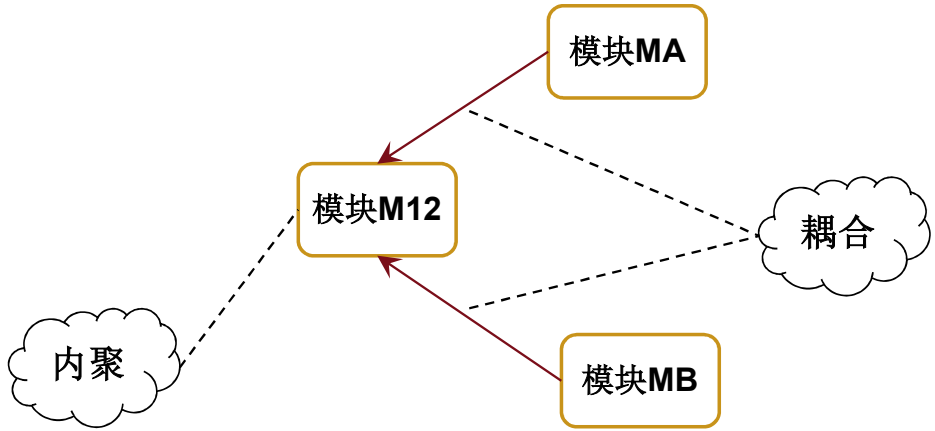
模块独立性的度量

内聚性:一个模块内部各成分之间相互关联的强度。

设计目标：高内聚

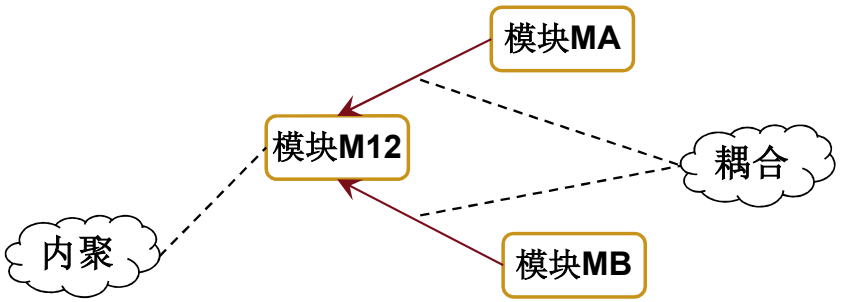
耦合度:模块间的紧密程度。

设计目标：低耦合

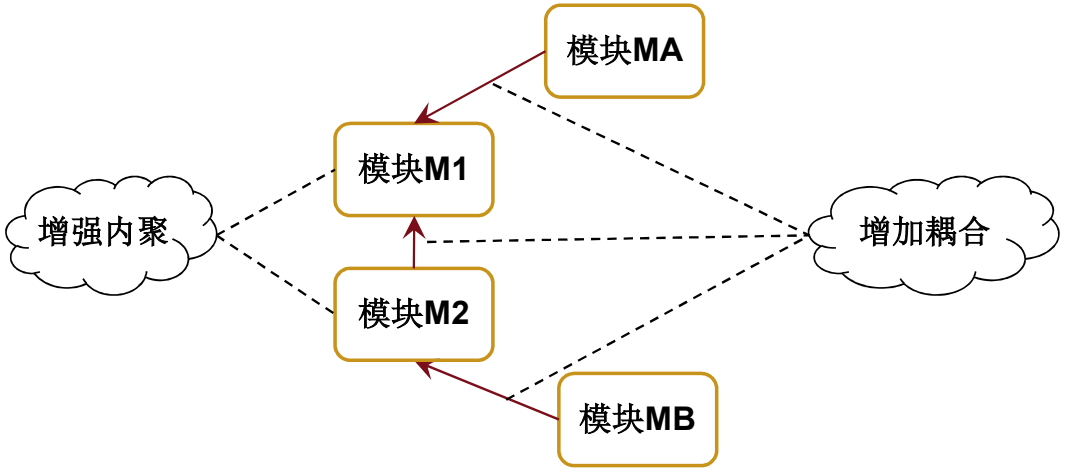


模块化设计

模块独立性的度量



- 内聚过低，相关功能分散在不同模块中，需要增加额外的耦合使这些功能聚合在一起，发生变更时影响多个模块。
- 内聚过高，不相关的功能聚集在一个模块中，耦合度高，发生变更时会产生意想不到的影响。



模块化设计

内聚类型与关系

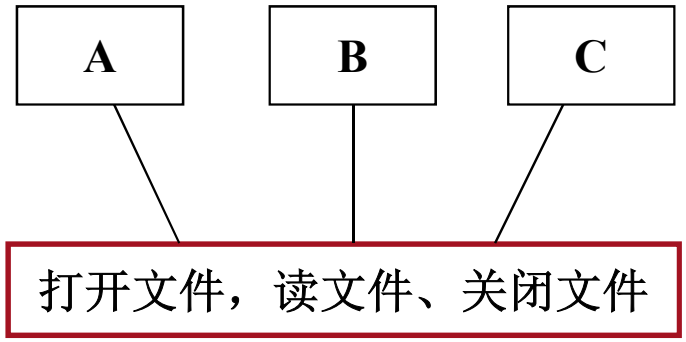
高 ← 内聚性 → 低

功能内聚	顺序内聚	通信内聚	过程内聚	时间内聚	逻辑内聚	偶然内聚

强 ← 模块独立性 → 弱

模块化设计

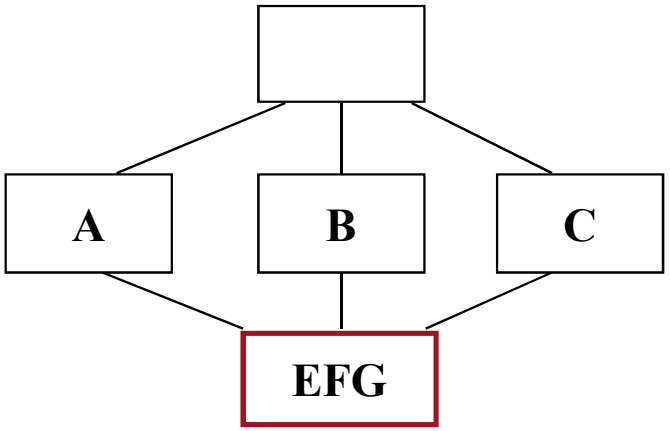
(1) 偶然内聚：模块内各部分间由于某种偶然因素（节约空间、提高效率、操作顺序等）组合在一起。



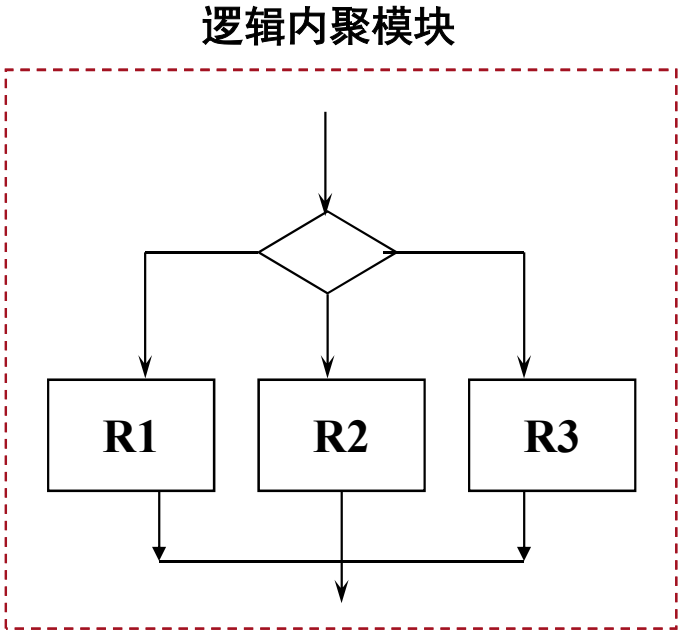
缺点：可理解性差， 可修改性差。

模块化设计

(2) 逻辑内聚：将几个逻辑上相似的功能放在一个模块中，调用时由调用模块所传递的参数确定执行的功能。



E、F、G逻辑功能相似(顺序读、随机读、逆序读)，组成模块EFG。



缺点：增强了耦合程度(控制耦合)不易修改，效率低。

模块化设计

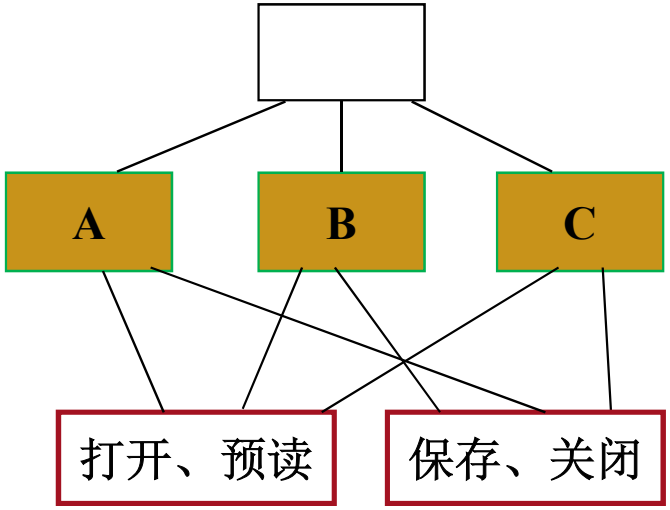
(3) 时间内聚：将需要同时执行的成分放在一个模块中，因为模块中的各功能与时间有关。

- 初始化系统模块
- 系统结束模块
- 紧急故障处理模块

```
class Stash  {
  private:
        int size;          // Size of each space
        int quantity;   // Number of storage spaces
        int next;          // Next empty space
        unsigned char* storage;
  public:
        void initialize(int size)
        {
            size = sz;
            quantity = 0;
            storage = 0;
            next = 0;
        }
        ……
};
```

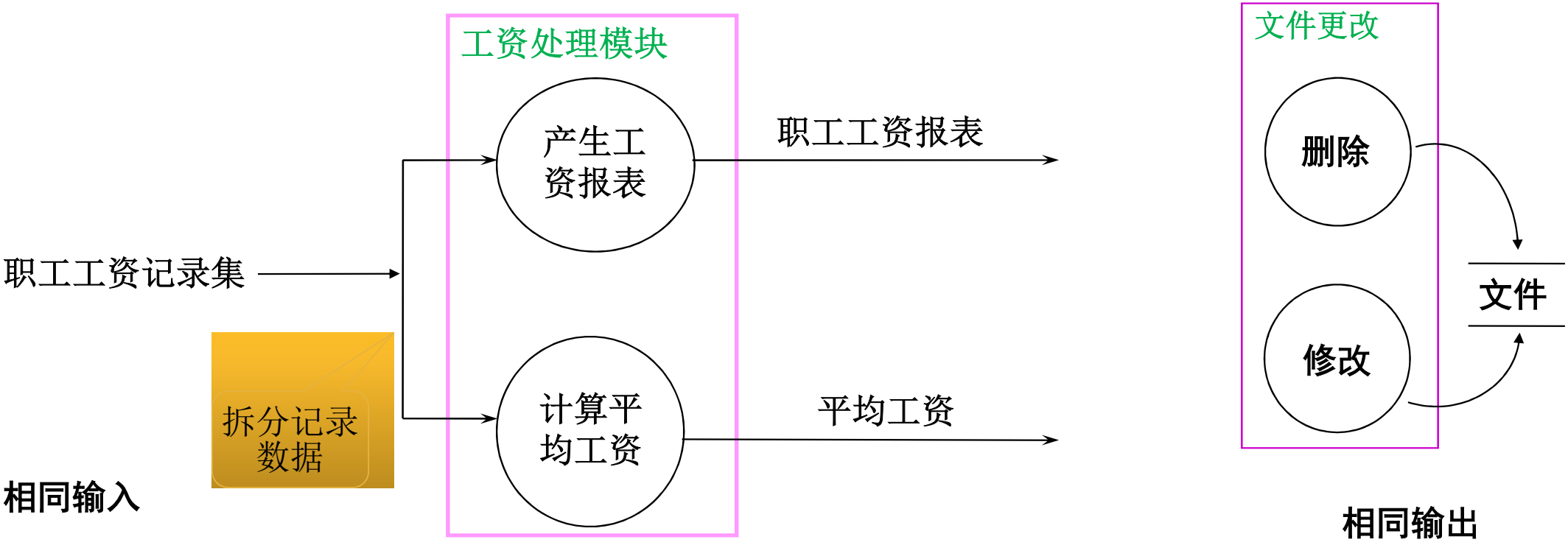
模块化设计

(4) 过程内聚：模块内部必须按照过程描述，在同一模块内自上而下地组织各任务。



模块化设计

(5) 通信内聚：模块中的成分引用共同的输入数据，或者产生相同的输出数据，则称为是通信内聚模块。



模块化设计

(6) 顺序内聚：模块中某个成分的输出是另一成分的输入。

(7) 功能内聚

模块仅包括为完成某个功能所必须的所有成分（ 模块所有成分共同完成一个功能，缺一不可 ）。

内聚性最强

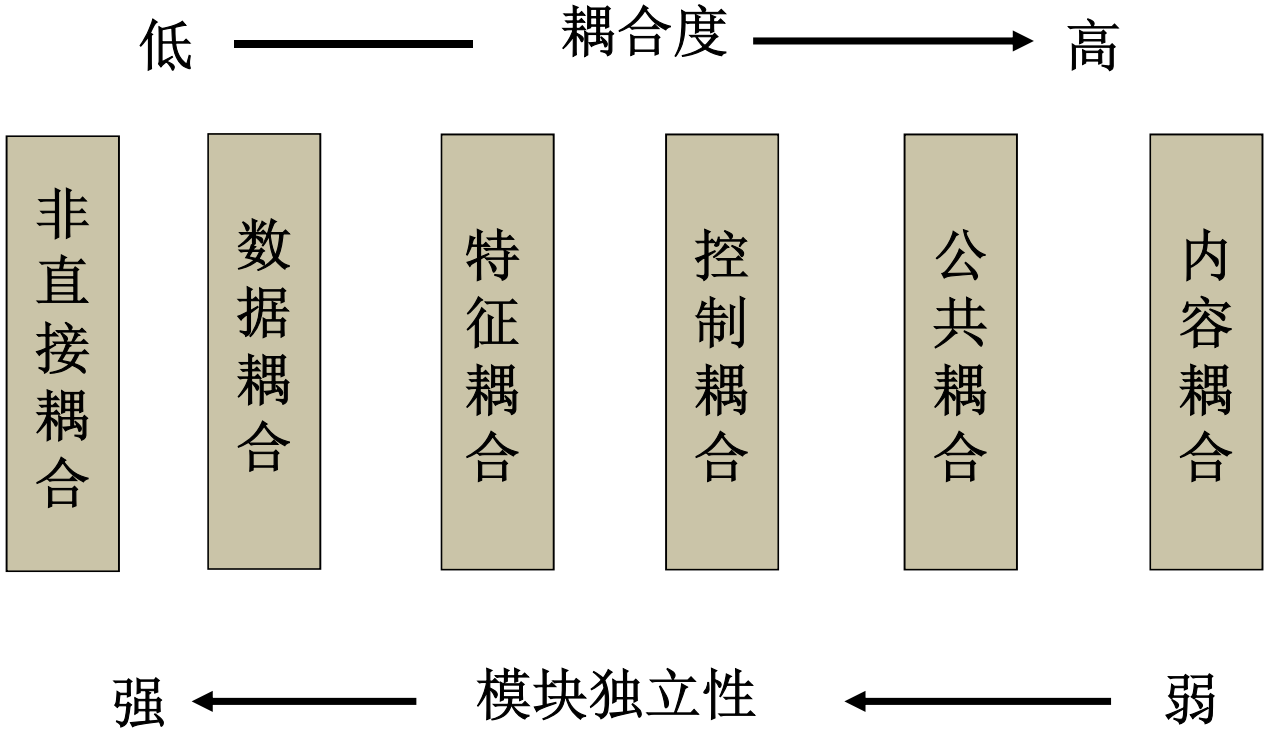
模块化设计

耦合强度依赖的因素：

- 一模块对另一模块的引用；
- 一模块向另一模块传递的数据量；
- 一模块施加到另一模块的控制的数量；
- 模块间接口的复杂程度；

模块化设计

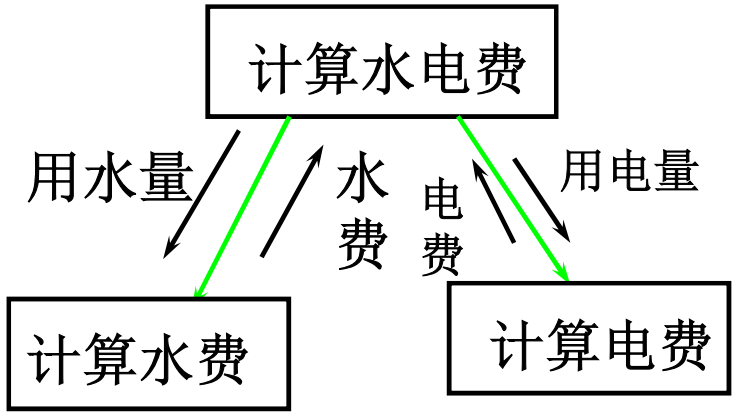
耦合类型与关系



模块化设计

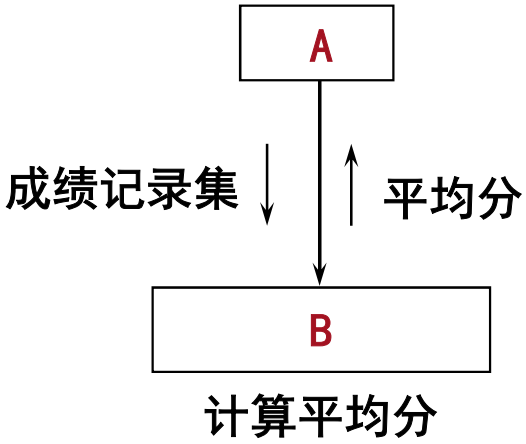
(1) 非直接耦合：模块间没有直接的相互调用关系。

(2) 数据耦合：一个模块传送给另一个模块的参数是一个单个的数据项或者单个数据项组成的数组。



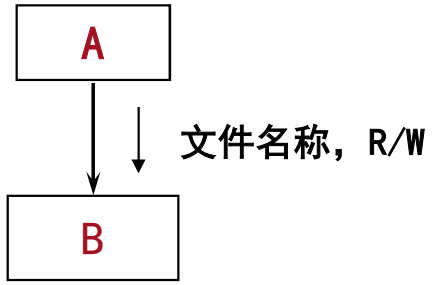
模块化设计

(3) 特征耦合：一个模块传送给另一个模块的参数是一个复合数据结构。



模块化设计

(4) 控制耦合：一个模块传递给另一模块的信息，该信息是用于控制该模块内部逻辑的控制信号。



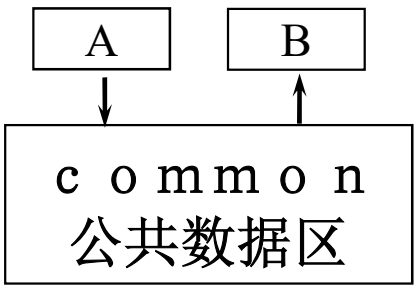
根据控制信息，以不同方式打开文件

模块化设计

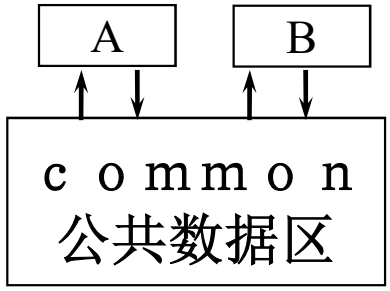
(5) 公共耦合：是指一组模块访问一个公共的数据环境。

公共数据环境包括：

- 全局数据结构
- 共享通讯区
- 内存公共覆盖区等



松散的
公共耦合



紧密的
公共耦合

公共耦合存在的问题：

(1) 软件可理解性降低(模块间存在错综复杂的连系)；

(2) 软件可维护性差(修改变量名或属性困难)；

(3) 软件可靠性差(公共数据区及全程变量无保护措施)；

模块化设计

(6) 内容耦合：一模块直接访问另一模块的内部信息(程序代码或数据)。

发生内容耦合的情形主要包括：

(1) 一模块直接访问另一模块的内部数据；

(2) 一模块不通过正常入口转到另一模块内；

(3) 两模块有一部分代码重叠；

(4) 一模块有多个入口。

模块化设计

耦合是影响软件复杂程度和设计质量的重要因素。

模块化设计目标：建立模块间耦合度尽可能松散的系统。

原则:

尽量使用数据耦合

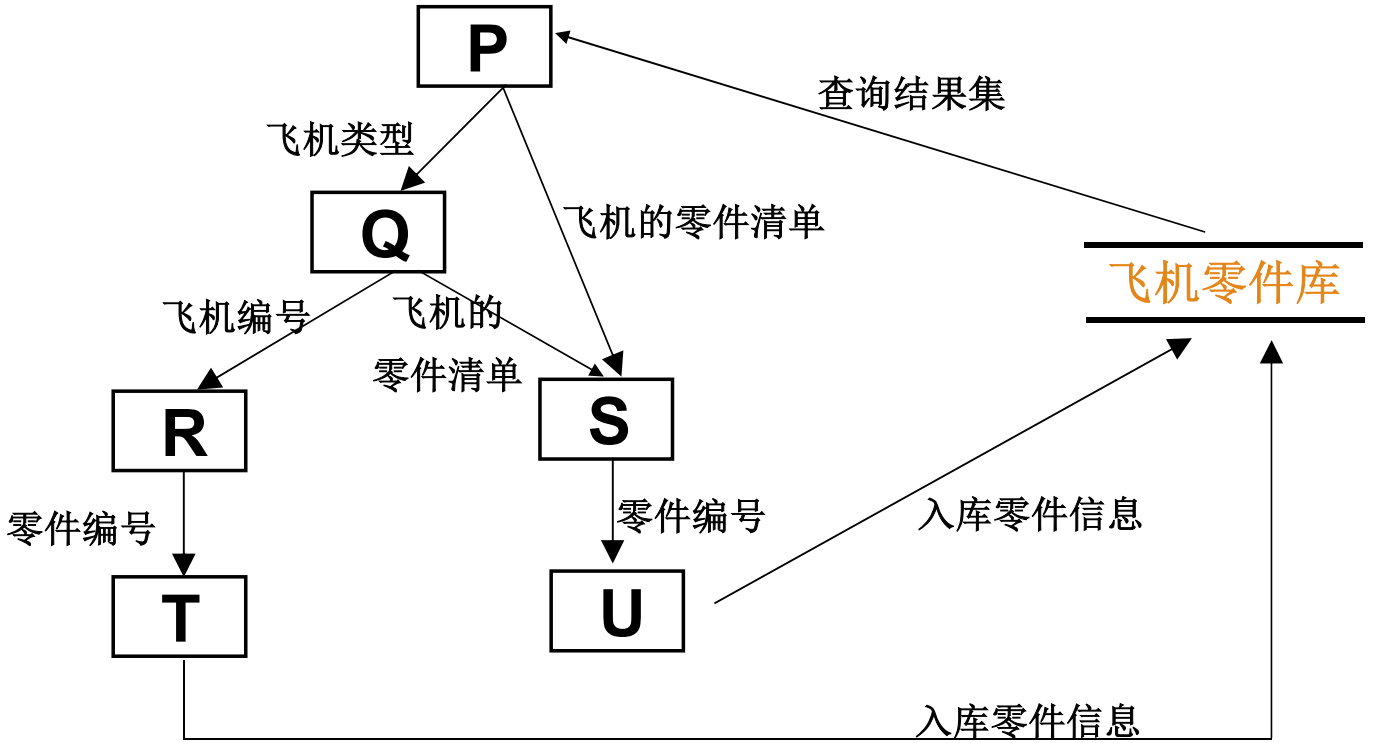
少用控制耦合

限制公共耦合的范围

坚决避免使用内容耦合

模块化设计

课堂练习：给出各模块间的耦合关系



模块化设计

内聚与耦合关系：

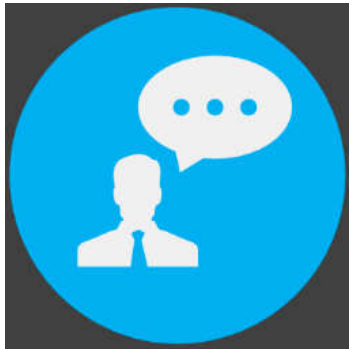
内聚与耦合密切相关，同其它模块强耦合的模块意味者弱内聚，强内聚模块意味着与其它模块间松散耦合。

设计目标：力争强内聚、弱耦合

内聚合耦合是密切相关的，模块内的高内聚往往意味着模块间的松耦合。内聚和耦合都是进行模块化设计的有力工具，但是实践表明内聚更重要，应该把更多注意力集中到提高模块的内聚程度上。

模块化设计

启发式规则



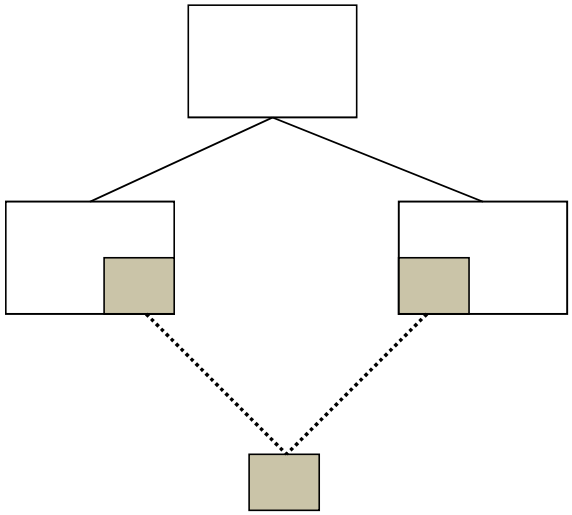
模块独立性确定原则除了设计高内聚、低耦合的模块结构之外，高质量的软件设计还需要进行优化。

在多数场合下，启发式规则能给软件工程师有益的启示，帮助他们找到改进软件设计、提高软件质量途径。

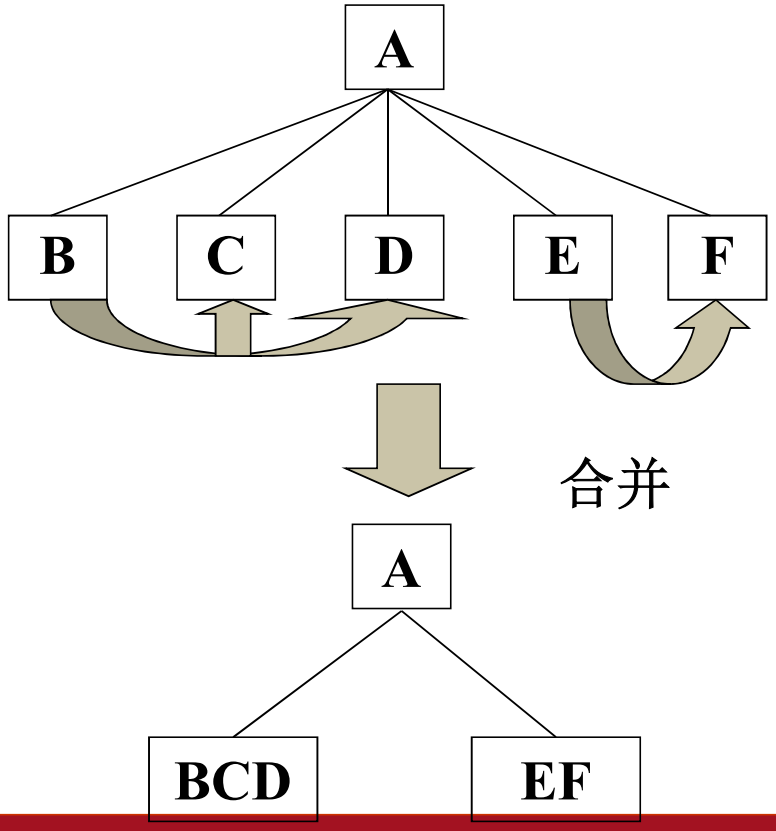
模块化设计

启发式规则

- 改进软件结构提高模块独立性



提取

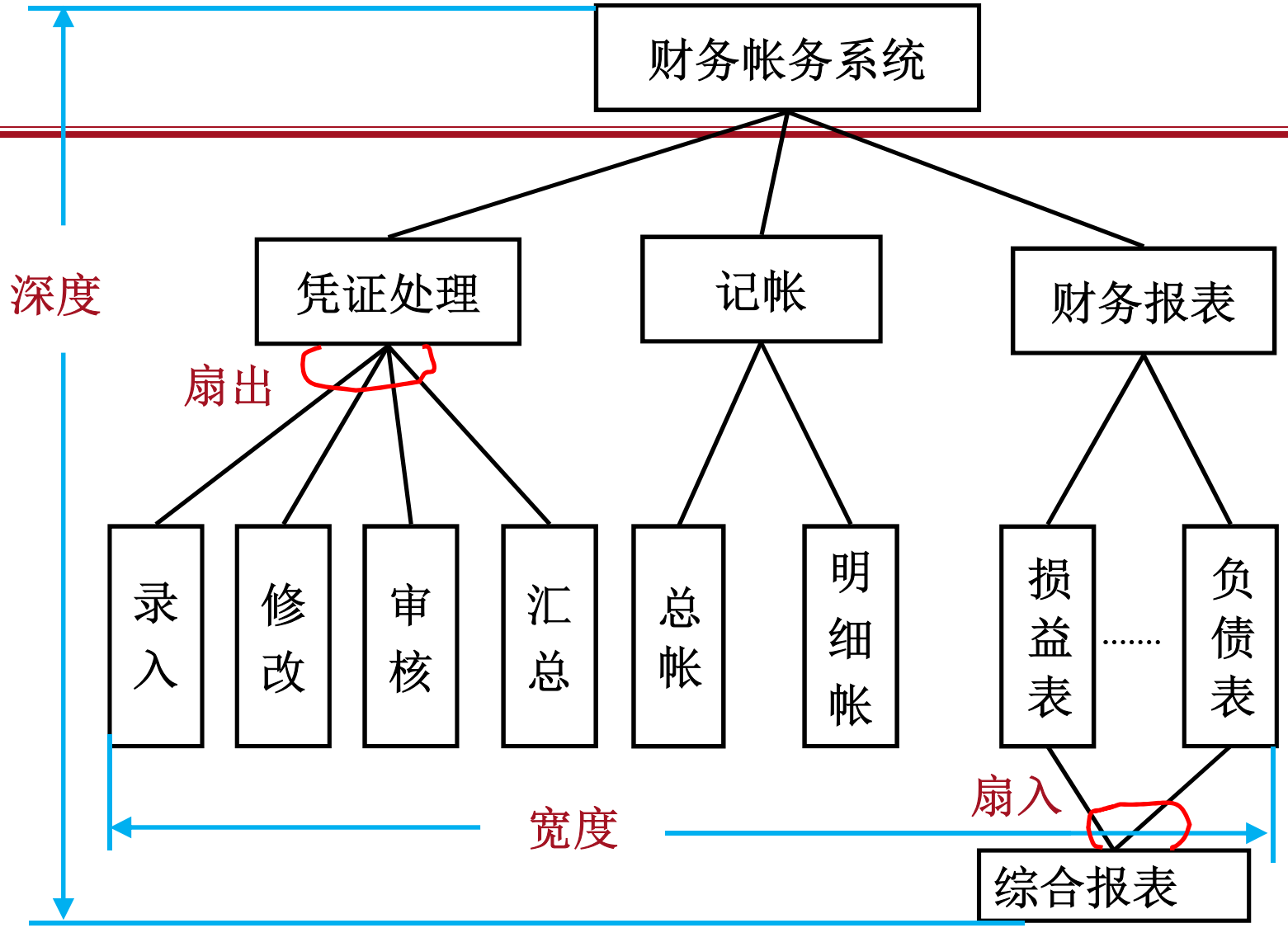


模块化设计

启发式规则

- 模块规模应该适中
- 系统结构深度、宽度、扇出和扇入都应适当

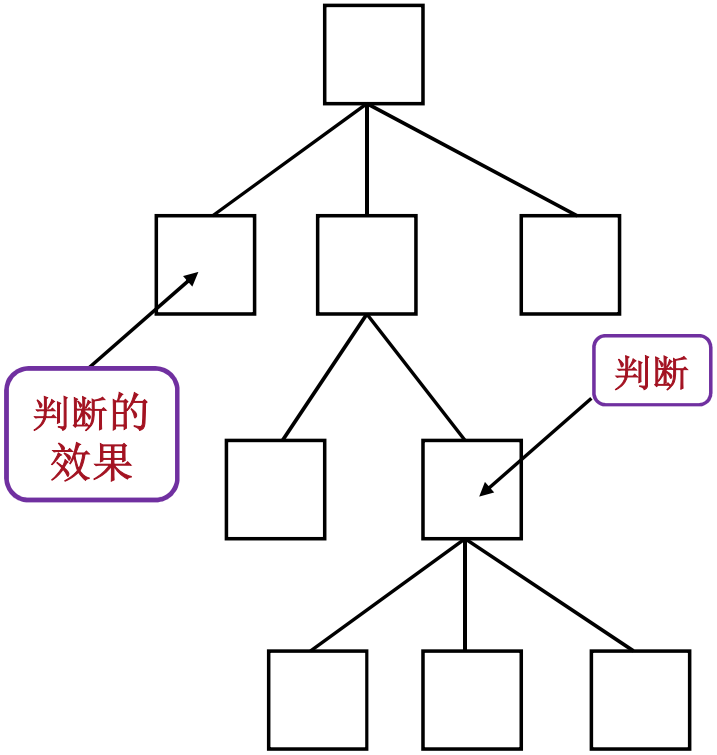
- 深度：表示软件结构中控制的层数，它往往能表示系统功能的复杂程度。
- 宽度：表示软件结构内同一个层次上的模块总数的最大值。一般来说，宽度越大，系统规模也越大。对宽度影响最大的因素是模块的扇出度。
- 扇出：表示一个模块直接控制（调用）的模块数目。扇出过大、过小都不好，一般在7±2之间。
- 扇入：表示有多少个上级模块直接调用它。


财务帐务系统
凭证处理
记帐
财务报表
深度
扇出
录入
修改
审核
汇总
总帐
明细帐
损益表
.....
负债表
扇入
宽度
综合报表

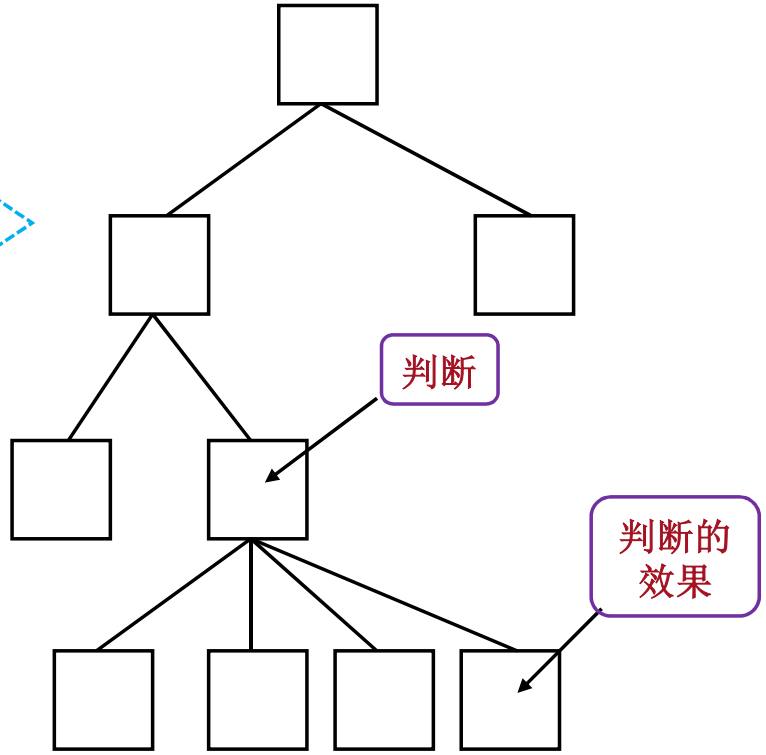
模块化设计

启发式规则

- 模块的作用域应该在控制域之内



违反准则的结构



修改后的结构（以满足准则）

模块化设计

启发式规则

- 力争降低模块接口的复杂程度

一元二次方程：$AX^2 + BX + C = 0$

定义接口如下：

QUAD-ROOT（ABC, X）;

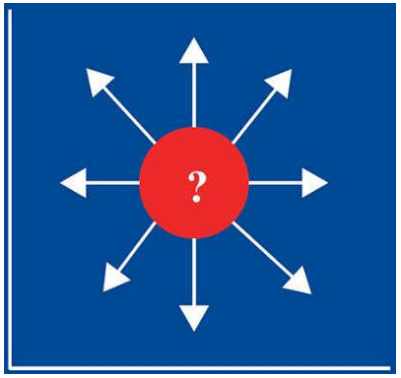
其中，**ABC** 和 **X**是数组。

修改为：

QUAD-ROOT(A, B, C, ROOT1, ROOT2);

模块化设计

启发式规则



- 模块功能可以预测

模块功能可以预测，也就是说在任何环境下和情况下，只要输入模块的数据不变，模块与其对应的结果就不会改变。这就要求模块设计尽量避免控制耦合和通信内聚，这也是软件测试和维护的要求。

数据库设计

A database **is an organized collection of data. --Wikepedia**

- 数据库就是对数据提供存储、处理、管理；
- 数据的业务处理包括增、删、改、查等操作；
- 数据管理还包括访问权限、数据持久化、分布式存储等不同形式。

在进行数据库总体设计时，基本遵循的是关系数据库 “范式”的规范。同时，在实际项目设计及应用中，需要根据具体领域背景进行综合考虑。

数据库设计

根据应用场景综合考虑——区别对待不同系统对数据库的应用性质

对数据库进行设计时，要考虑系统对数据库的应用类型。

- 事务处理型（**Transactional**）：用户关注数据的增、删、改、查的操作，称之为“**OLTP**”。**OLTP**对数据库定义要求满足不同等级的范式要求。

- 分析型（**Analytical**）：用户关注数据分析、报表、趋势预测等等功能，而对数据库的 “插入”和 “更新”等 操作相对来说是较少。这种类型称之为 “**OLAP**”。**OLAP**对可以设计一个不规范化的数据库结构。

数据库设计

根据应用场景综合考虑——数据库设计要适应需求的变更

数据库结构的变化，有时会给系统设计与实现带来灾难性的后果。因此，在概要设计阶段，设计良好的、适应需求变更的数据库结构，是一项重要而复杂的工作。

1988年Bertrand Meyer在他的著作《Object Oriented Software Construction》中提出了开闭原则，书中原文是：“Software entities should be open for extension, but closed for modification”。

数据库设计

根据应用场景综合考虑——数据库设计时的分类选择

- 数据存储形式

关系数据形式

ID	用户名	密码	昵称
001	WuKong	123456	HenXiong
002	BaJie	Abcdef	HenGui

key-value 形式

ID	Info
001	用户名:WuKong,密码:123456,昵称:HenXiong
002	用户名:BaJie,密码:abcdef,昵称:HenGui

数据库设计

根据应用场景综合考虑——数据库设计时的分类选择

- 数据存储
  - 持久化存储？内存数据库？
  - 单个数据服务器？分布式存储？
- 数据操作
  - 关系增、删、改、查？
  - 其它类型数据的增、删、改、查？
- 数据安全
  - 事务
  - 一致性
- 数据可用
  - 故障？
  - 恢复？及时、热备份或是……？

数据库设计

根据应用场景综合考虑——常用数据库

- **MySql**

  - 开放源代码且无版权制约，自主性强、使用成本低。
  - 简单易用，社区及用户非常活跃，且拥有大量第三方插件。
  - 支持多种操作系统，提供多种**API**接口，支持多种开发语言。
  - 支持关系数据复杂的查询操作。
  - 支持完整的事务操作和较高的安全性。

数据库设计

根据应用场景综合考虑——常用数据库

- **MongoDB**

ID	Info
001	用户名:WuKong,密码:123456,昵称:HenXiong
002	用户名:BaJie,密码:abcdef,昵称:HenGui

- 数据模式自由，较为容易的修改数据格式；
- 支持索引，支持海量数据的查询和插入；
- 支持故障恢复与备份；
- 第三方支持也较为丰富。

- 不支持事务，不支持复杂查询；
- 安全性较差；
- 建立索引占据空间较大。

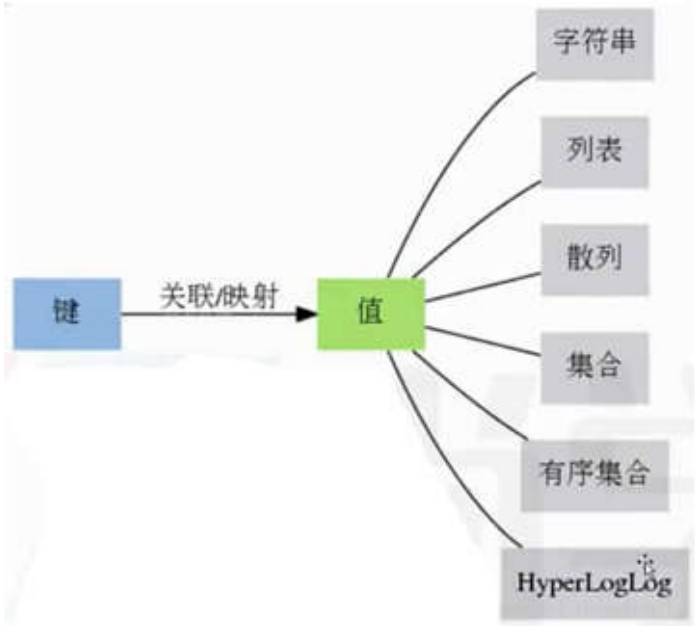
数据库设计

根据应用场景综合考虑——常用数据库

- **Redis**

  - 内存数据库，数据保存在内存中，确保高效访问；
  - 同时也支持持久化存储；
  - 对**key-value**中多种**value**形式的支持；
  - 访问速度就是快！

  - 数据保存在内存中，可靠性难以保证；
  - 事务机制较差；
  - 安全性较差。



数据库设计

根据应用场景综合考虑——数据库的选择（一）

二手交易网站的需求分析（性能）：要求在用户登录时，每秒同时在线人数（并发请求）大约为**400**人。

- 显然交易过程中的各类信息，需要同时持久化存储，不考虑**Redis**；
- 从交易安全性上分析，由于二手物品交易每笔金额不大，数量不多，可以考虑**MySql**或者**MongoDB**；
- 二手物品要有较为详细的描述，特征较多（几成新？用了多久？外观？有无发票等），建议采用**MongoDB**。

数据库设计

根据应用场景综合考虑——数据库的选择（二）

班级微信公众号的需求分析：发布班级信息的公众号，无需注册与登录即可浏览。预估高峰时段同时在线人数（并发请求）大约为**16000**人/每秒（全校本科生）。

- 由于无需用户注册与登录，因而没有用户数据；
- 仅提供信息浏览，没有复杂的业务逻辑；
- 由于班级信息信息大多由文字、图片、动画、短视频等组成，页面结构较为简单、固定，建议考虑**Redis**；
- 展示的信息通常对安全性、可靠性等没有特别要求，因此**Redis**是合理的。

数据库设计

根据应用场景综合考虑——数据库的选择（三）

校园内二手交易网站的需求分析：要求在用户登录时，每秒同时在线人数（并发请求）大约为**400**人。此外，同学可以通过微信公众号，浏览二手交易的物品。预估高峰时段同时在线人数（并发请求）大约为**16000**人/每秒。

- 显然交易过程中的物品信息需要持久化存储，因此不考虑**Redis**；
- 从交易安全性上分析，由于校园二手物品交易每笔金额不大，数量不多，可以考虑**MySql**及**MongoDB**；
- 二手交易网站主要提供物品浏览，结构较为简单，交易过程也无复杂的业务逻辑，考虑**MySql**及**MongoDB**；
- 由于校院内的二手物品类型较少，以书籍及常用生活物品为主，物品信息结构变化不大，建议用**MySql**。

界面设计

- 界面是用户和系统间进行接收数据、数据变换、展示数据的平台，它实现的是系统内部信息表示和用户数据显示之间的转换。
- 按照以用户为中心的观点，是软件设计的核心。
- 软件是否成功，不是由技术专家使用专业的标准来评判，而是由用户来评判，由用户是否认可、是否喜欢来评判。

界面设计

界面设计的任务——用户特性分析

以用户为中心，对于软件设计人员来说，需要先有几个基本观念：

易出错


健忘


笨的


注意力涣散


懒惰

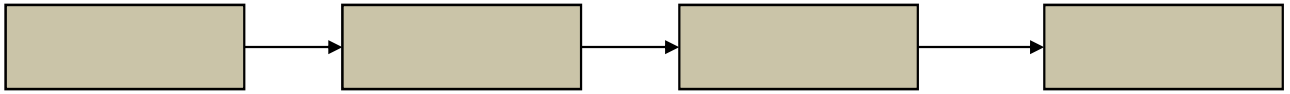

脾气差

界面设计

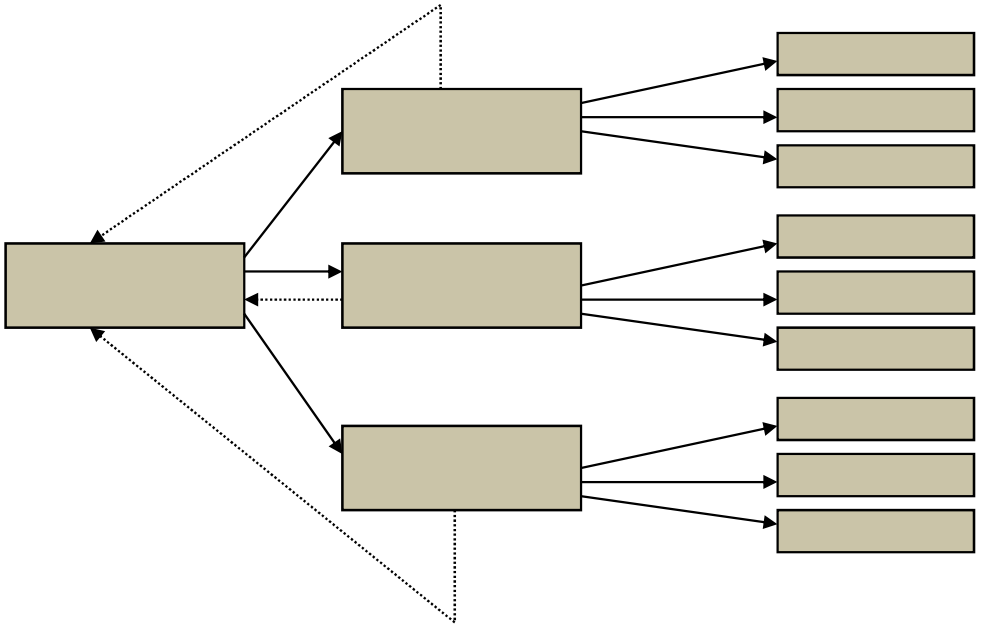
界面设计原则——提供网页向导

导航方式:线性、层次、网络式、混合式

线性

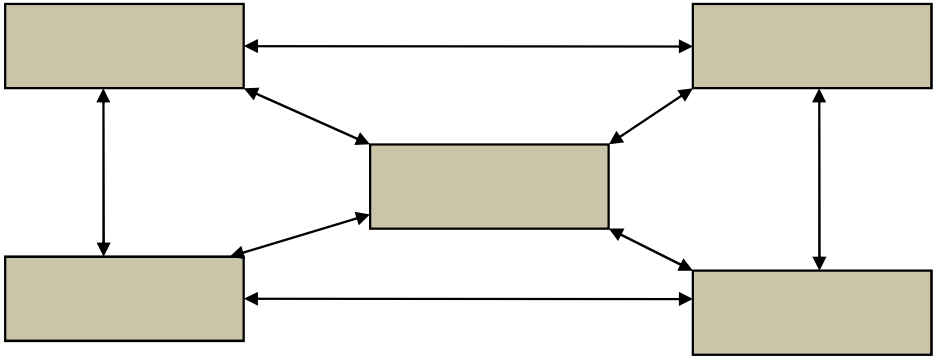


层次

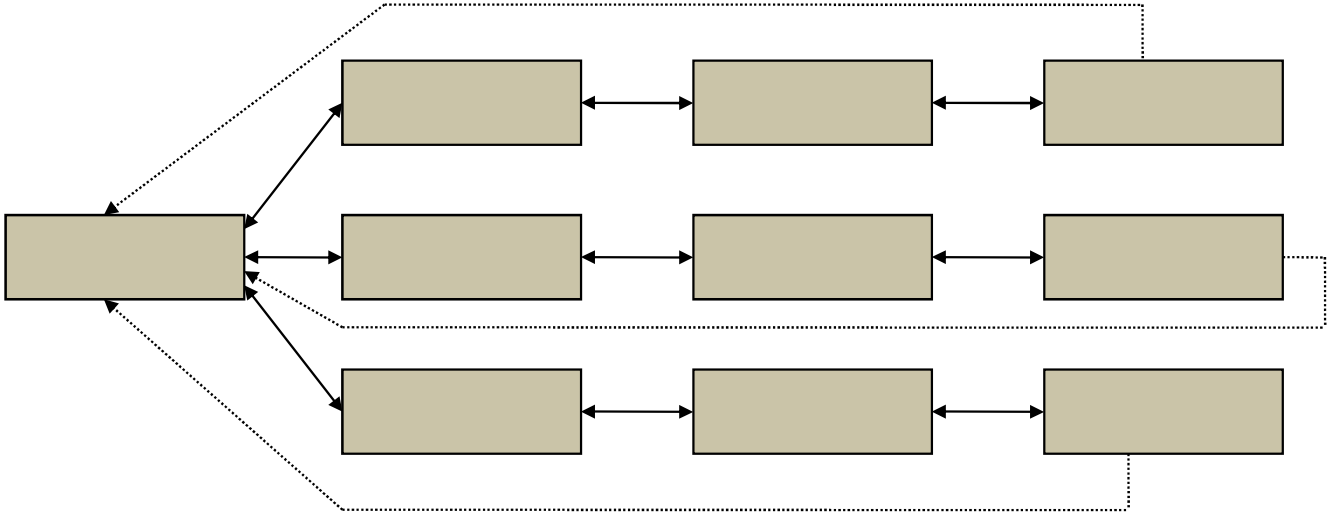


界面设计

网络式



混合式



界面设计

Theo Mandel 提出著名的界面设计三条“黄金规则”

置用户于控制之下

- 用户控制系统，而不是系统控制用户；
- 提供灵活的交互；
- 允许用户交互被中断或撤销；
- 允许用户与界面上的对象进行交互。

减少用户记忆负担

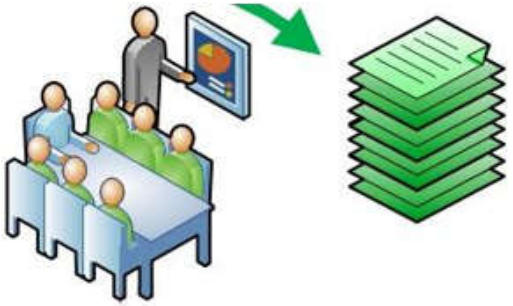
- 减少对短期记忆的要求；
- 设置有意义的缺省值；
- 定义简单易用的快捷键；
- 界面应反映用户真实应用环境；
- 以不断进展的方式（不同抽象层次）给出信息。

保持界面一致

- 提供各界面一致的操作形式和颜色编码；
- 与用户已有的交互保持一致，例如快捷方式等。

KISS准则：Keep it Simple and Stupid

软件设计评审



在软件设计完成之前，必须编写软件设计规格说明、数据设计说明和接口设计说明，并确定软件界面设计方案，并按照评审标准对软件设计过程和说明书进行评审，目的是发现并消除其中存在的遗漏、错误和不足，使得说明书符合标注及规范的要求。

通过了评审的软件设计规格说明、数据设计说明、接口设计和界面设计方案，就成为基线配置项，纳入项目管理的过程。

软件设计评审

软件设计评审标准

- 与软件需求规格说明书的契合度
- 对软件设计规格说明文档评审的标准
- 对整个软件设计合理性的评审标准

第3章 软件设计 小结

- 软件设计概述
- 软件体系结构设计
- 模块化设计
- 数据库设计
- 界面设计
- 软件设计评审