4-1 软件设计与架构概论

2019年4月14日 10:27

1 软件设计的背景

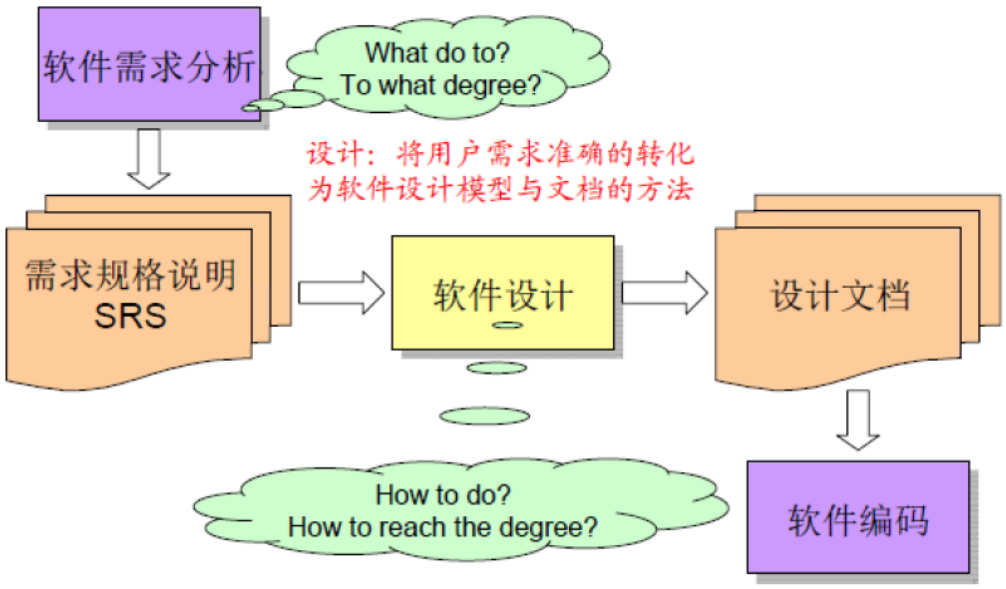
设计=天才+创造力

- 每个工程师都希望做设计工作，因为这里有“创造性”——客户需求、业务要求和技术限制都在某个产品或系统中得到集中的体现。
- “设计”是最充分展现工程师个人价值的工作。

“软件设计”的定义

- 软件设计：为问题域的外部可见行为的规约增添实际的计算机系统实现所需的细节，包括关于人机交互、任务管理和数据管理的细节。
- 关于“软件设计”的几个小例子：
  - 需求1：教学秘书需要将学生的综合成绩按高到低进行排序
    设计1：voidOrderScores(struct*scores[]){冒泡排序算法,step1;step2;...}
  - 需求2：数据字典“销售订单”
    设计2：关系数据表Order(ID,Date,Customer,...),
    OrderItem (No,PROD,QUANTITY,)
  - 需求3：“查询满足条件的图书”
    设计3：图形化web用户界面

软件设计在SE中所处的位置

- 

从建筑设计看软件设计

“设计良好的软件应该展示出坚固、适用和令人赏心悦目的特点。”

- 坚固：软件应该不含任何妨碍其功能的缺陷；
- 适用：软件要符合开发的目标，满足用户需求；
- 赏心悦目：使用软件的体验应该是愉快的。

良好的软件设计的三个特征

- 目标：设计必须是实现所有包含在分析模型中的明确需求，满足利益相关者期望的所有隐含需求；
- 形态：对开发、测试和维护人员来说，设计必须是可读的、可理解的、可操作的指南；
- 内容：设计必须提供软件的全貌，从实现的角度去说明功能、数据、行为等各个方面。

设计的目标：质量

- “设计阶段”是软件工程中形成质量的关键阶段，其后所有阶段的活动都要依赖于设计的结果。
- “编写一段能工作的灵巧的代码是一回事，而设计能支持某个长久业务的东西则完全是另一回事。”

软件质量

- 外部质量：面向最终用户

  如易用性、效率、可靠性、正确性、完整性等
- 内部质量：面向软件工程师，技术的角度

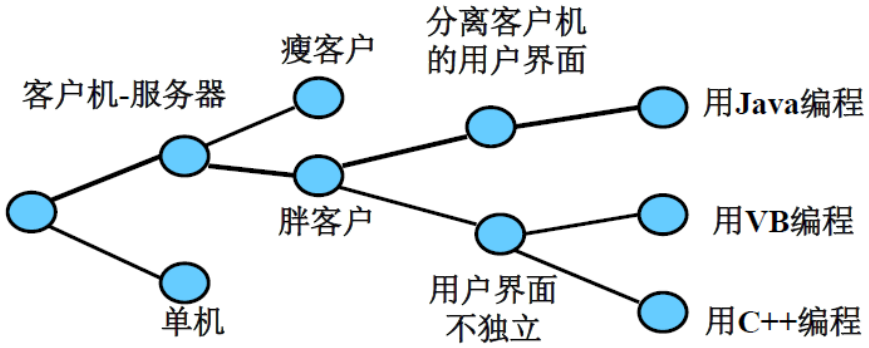
  如可维护性、灵活性、可重用性、可测试性等
- 大多数软件设计师只关注外部质量忽视内部质量，导致软件变更代价大

软件设计的方式

- 软件设计的目标是创作出坚固、适用和赏心悦目的模型或设计表示
- 方式：先实现多样化再进行聚合
  - 多样化：找到各种可能的解决方案
  - 聚合：折中选取最适合的
- 多样化和聚合需要直觉和判断力，其质量取决于
  - 构造类似实体的经验
  - 一系列指导模型演化方式的原则和（或）启发
  - 一系列质量评价的标准以及导出最终设计表示的迭代过程
- 软件工程缺少经典工程设计（如建筑）所具有的深度、灵活性和定量性，但是软件设计的方法是存在的，设计质量的标准是可以获得的，设计表示法也是能够应用的

设计=不断的作出决策

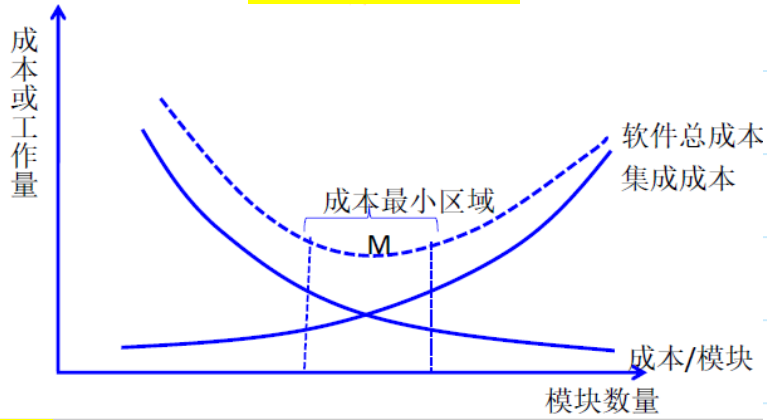
- 解决“Howtodo”，就需要不断的做出各种“设计决策”，在各类需求之间进行“折中”，使得最终设计性能达到最优。
-



2 软件设计中的核心概念

设计概念

- 软件工程历史上，产生了一系列基本的软件设计概念
- 每种概念的关注程度不断变化，但都经历了时间的考验
  - 抽象：强调关键特征，忽略实现细节
    - 过程抽象：具有明确和有限功能的指令序列
    - 数据抽象：描述数据对象的冠名数据集合
  - 体系结构：程序构件（模块）的结构、构件交互的形式、构件的数据结构
    - 功能模型：表示系统的功能层次结构
    - 框架模型：解决某一类问题的处理流程
    - 结构模型：程序构件的一个有组织的集合
  - 关注点分离：关注点是一个特征或行为，软件需求模型的一部分；任何复杂问题如果被分解为可以独立解决和优化的若干块，该复杂问题能够更容易地被处理；

- 模块化：关注点分离最常见的表现，分治策略，将软件划分为独立构件（模块）
  - 合理划分模块数量 复用度？复用价值？
  - 

- 信息隐藏：每个模块对其他所有模块都隐藏自己的设计决策
  - 软件只通过定义清晰的接口通信
  - 每一个接口尽可能暴露最少的信息
  - 如果内部细节发生变化，外部的受到影响应当最小
- 功能独立：是关注点分离、模块化、抽象概念和信息隐藏的结果
  - 功能专一，避免与其他模块过多交互
  - 独立性的评价：内聚性和耦合性
- 求精：
  - 逐步求精
  - 分解细化
  - 层次结构
- 重构：是一种重新组织的技术，不改变外部行为而是改进内部结构
- 设计类：精化分析类、创建新的设计类
  - 用户接口类（边界类）：人机交互所必需的抽象
  - 领域类（实体类）：分析类的精化
  - 过程类（控制类）：底层业务抽象
  - 持久类：持续存在的数据存储
  - 系统类：软件管理和控制功能
- 模式

基于模式的软件设计

- 模式：特定问题的解决方案
- 体系结构模式：定义软件的整体结构，体现了子系统和软件构件之间的关系，并定义了说明体系结构元素（类、包、构件、子系统）之间关系的规则
- 设计模式：这些模式解决设计中特有的元素，例如解决某些设计问题中的构件聚集、构件间的联系或影响构件到构件之间通信的机制
- 框架（Framework）：已实现的特定的骨架基础设施，提供了基础功能，并规定了构件及构件的连

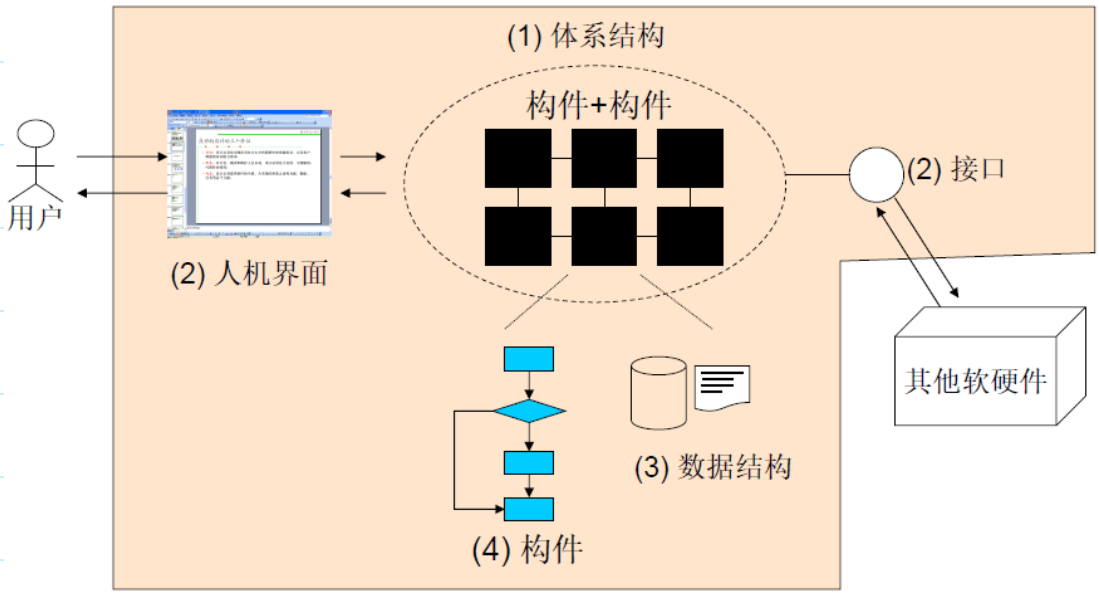
接方式。开发者只需要将注意力集中于业务逻辑的实现

设计建模原则

- 设计可追溯到分析模型
- 经常关注待建系统的架构
- 数据设计与功能设计同等重要
- 必须设计接口（包括内部接口和外部接口）
- 用户界面设计必须符合最终用户要求
- 功能独立的构件级设计
- 构件之间以及构件与外部环境之间松散耦合
- 设计表述（模型）应该做到尽可能易于理解
- 设计应该迭代式进行。每一次迭代，设计者都应该尽力简化

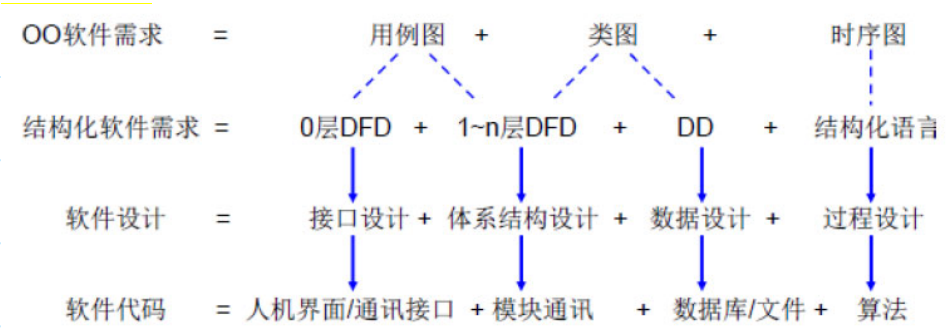
3 软件设计模型

软件设计的四方面内容

- 

软件设计的元素

- 体系结构设计：定义了软件的主要结构元素之间的联系，也用于达到系统所定义需求的体系结构风格和设计模式以及影响体系结构实现方式的约束
- 接口设计：描述了软件和协作系统之间、软件和使用人员之间是如何通信的
- 数据/类设计：将分析类模型转化为设计类的实现以及软件实现所要求的数据结构
- 构件级设计：将软件体系结构的结构元素变换为对软件构件的过程性描述

- 

软件体系结构设计

- 选择适合于需求的软件体系结构风格，软件架构设计；
- 如何以最佳的方式划分一个系统，如何标识组件，组件之间如何通信，信息如何沟通，系统的元素如何能够独立地进化
- 例如：基于功能层次结构建立系统：
  - 采用某种设计方法，将系统按功能划分成模块的层次结构
  - 确定每个模块的功能
  - 建立与已确定的软件需求的对应关系
  - 确定模块间的调用关系
  - 确定模块间的接口
  - 评估模块划分的质量

接口设计

- 接口是类、构件或其他分类（包括子系统）的外部可见的（公共的）操作说明，而没有内部结构的规格说明
  - 用户界面（UI）
  - 与其他系统、硬件的外部接口
  - 各种构件之间的内部接口

构件级设计

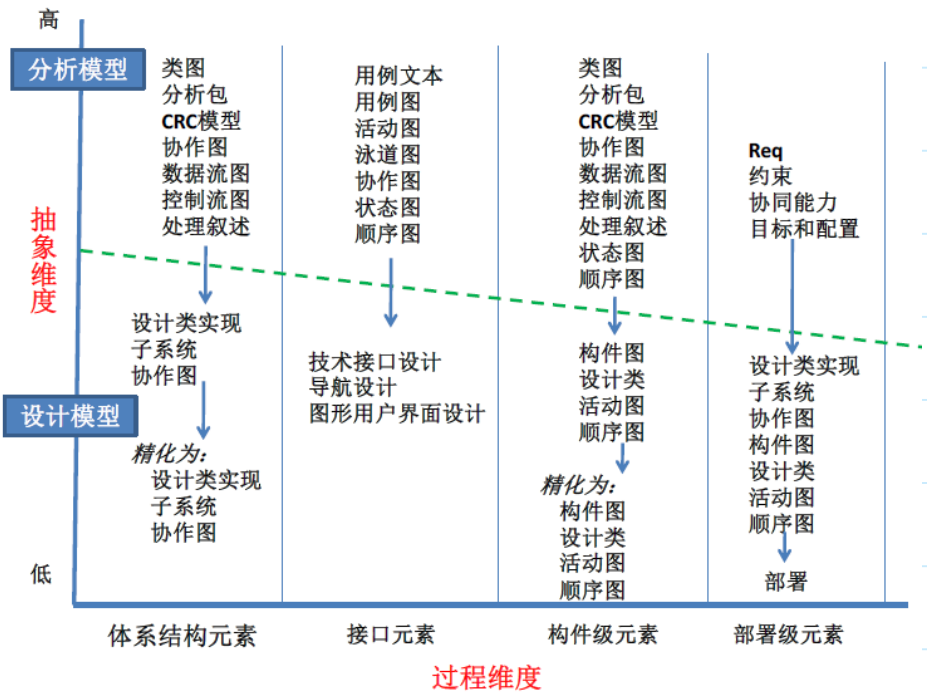
- 构件是面向软件体系架构的可复用软件模块
- 完整地描述了每个软件构件的内部细节
  - 为本地数据对象定义数据结构，为构件内的处理定义算法细节
  - 定义允许访问所有构件操作（行为）的接口

数据设计

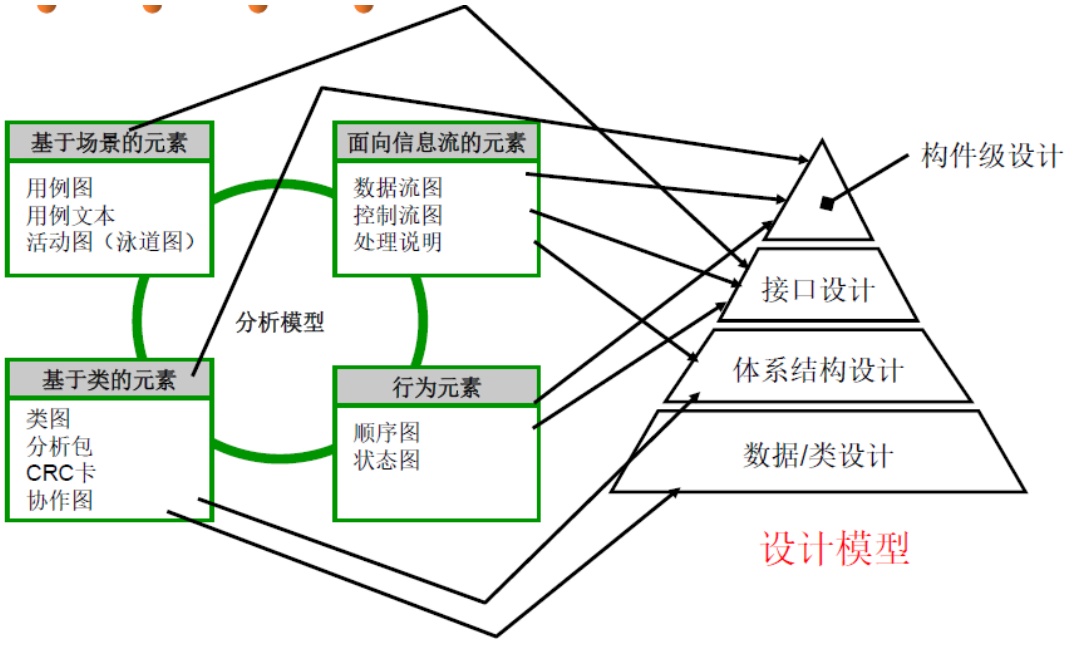
- 体系结构级数据设计
  - 确定软件涉及的文件系统的结构以及数据库的模式、子模式，进行数据完整性和安全性的设计
  - 确定输入、输出文件的详细的数据结构

- 构件级数据设计
  - 结合算法设计，确定算法所必需的逻辑数据结构及其操作
  - 确定对逻辑数据结构所必需的那些操作的程序模块(软件包)
  - 限制和确定各个数据设计决策的影响范围
  - 确定其详细的数据结构和使用规则
  - 数据的一致性、冗余性设计

设计模型的维度

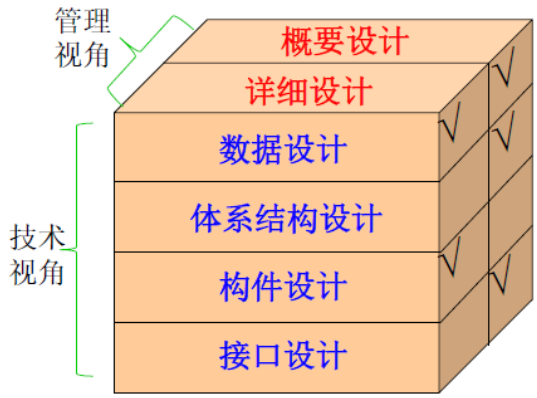
- 

从分析模型到设计模型的转化

- 

软件设计的两大阶段--从工程管理的角度看：

- 概要设计：将软件需求转化为数据结构和软件的系统结构
- 详细设计：即构件设计，通过对软件结构表示进行细化，得到软件的详细的数据结构和算法



面向对象设计

- Jacobson：“当实现的细节开始显现，那就是设计”
- 对象关注点转移到解决域
  - 对象、语义和关系被确定
  - 贯彻需求，不断迭代
  - 注重质量和原则

面向对象的设计的两个阶段

- 系统设计(SystemDesign)
  - 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(ObjectDesign)
  - 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。

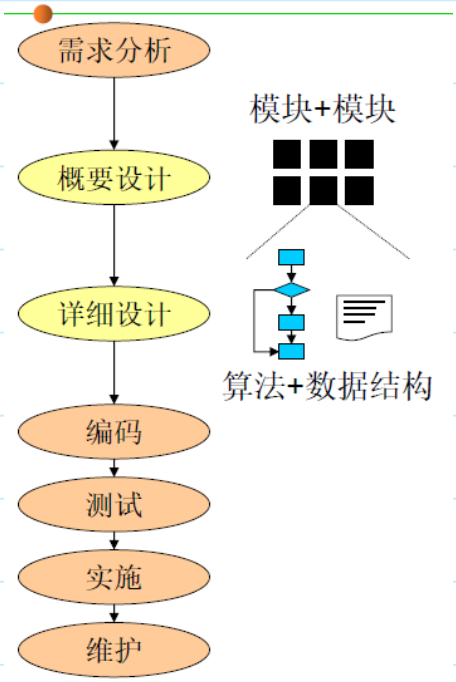
4 什么是软件架构

为什么需要软件架构?

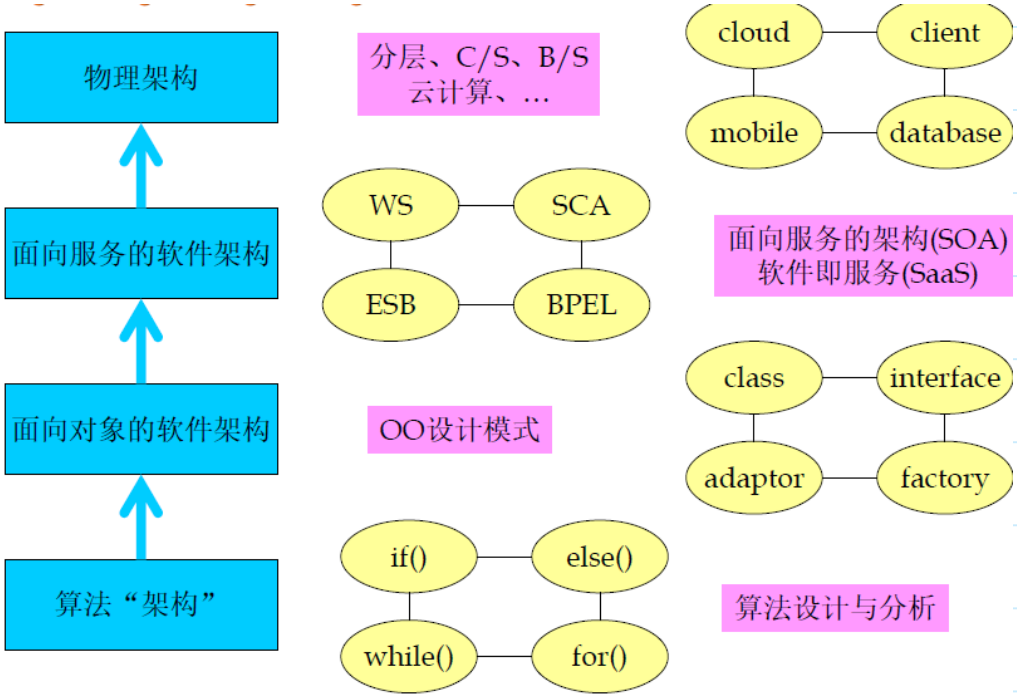
- 软件越来越复杂，组成部分越来越多

    - 多个源文件
    - 多种类型的文件：用户界面、算法、数据层程序、配置文件、etc
- 不是单纯的代码，还涉及到所依赖的硬件和网络环境
    - 物理位置：单机、服务器、手机端、可穿戴硬件、etc
    - 网络支持：有线网络、3G/4G、WiFi等
- 多个软件实体之间如何组织起来?
- 软件和硬件之间的关系如何?
- 此即“软件架构”(SoftwareArchitecture)所关注的内容。

软件设计的过程

- 

你早已接触过“架构”...



- 超市的多个收银台，顾客排长队===服务器并发处理的性能和容量
- 十字路口的车辆等待转弯===通过缓存来提高交通吞吐率
- 分层、构件化、服务化、标准化、…
- 缓存、分离/松散耦合、队列、复制、冗余、代理、…
- C/S、B/S、负载平衡、…
- 如何用它们解决具体软件问题，在博弈中寻求平衡(折中)，就是软件架构所关注的问题。
- 新的架构层出不穷，但所遵循的基本原理都是相通的。

MySQL MemCached的缓存和分布式存储架构

MemCached:

- 一个高性能的分布式内存对象缓存系统，用于动态Web应用以减轻DB负载
- 它通过在内存中缓存数据和对象来减少读取DB的次数，从而提供动态、DB驱动网站的速度;
- 用来存储各种格式的数据，包括图像、视频、文件以及数据库检索的结果等。
- 基于一个存储键/值对的HashMap;
- 通过LRU算法进行淘汰，同时可以通过删除和设置失效时间来淘汰存放在内存的数据
- 一致性hash解决增加或减少缓存服务器导致重新hash带来的大量缓存失效的弊端。

如何学习软件架构

- 从最基本的概念入手，学习传统的软件架构设计思想
  - 数据流、数据仓库、结构化、OO、分层、事件、规则、…
- 逐步过渡到各类新涌现的架构思想
  - P2P、web20、SOA、EDA、虚拟化、SaaS、云计算、…
- 通过模仿已有架构和案例来设计自己的系统
  - MVC、Struts、Hibernate、Spring、…

    - Facebook、Twitter、Android、...
- 了解不同应用领域的架构套路
    - 企业级系统、移动终端软件、在线游戏、...

什么是“体系结构”,词典的定义:

- 建筑学：设计和建造建筑物的艺术与科学
- 设计及构造的方式和方法
- 部件的有序安排；结构
- 计算机系统的总体设计或结构，包括其硬件和支持硬件运行的软件，尤其是微处理器内部的结构

“体系结构”的共性

- 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

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

为什么要重视“软件架构”

- 随着软件系统规模越来越大、越来越复杂
    - 用户需求越来越复杂，变化越来越频繁；
    - 对软件质量(功能性/非功能性)的要求越来越高；
    - 如何将成百上千个功能组合起来，满足用户质量需求，变得越来越困难。
- 此时，整个系统的结构和规格说明显得越来越重要。
    - 很多质量需求主要体现在体系结构中而非功能模块内部的实现中。
- 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。

软件架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?

软件架构的目标与作用

- 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;

外向目标VS内向目标

外向目标	内向目标
关心的是：为满足最终用户的需求，系统应该做些什么。	关心的是：如何使系统满足用户需求、需要建立哪些软件模块、软件模块的结构、模块之间的关系
要在系统实现之前寻求需求决策并确保系统的性能。	通过对主要构件及其关系的规划，为以后的系统设计和实施活动提供了基础和依据。
为此，需要分析当前需求、扩展或细化结构、澄清模糊性、提高一致性	为了达到这个目标，需要考虑各种可选方案，重复更新和明确设计目标，必要时作出妥协(compromise)和折中(tradeoff)。

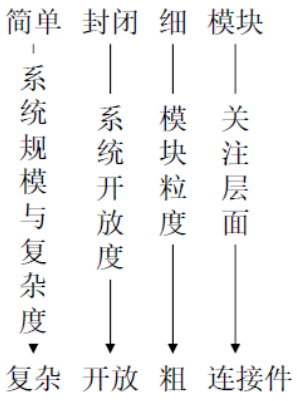
典型质量属性的架构设计

- 可用性(availability)
  - 当系统不再提供其规格说明中所描述的服务时，就出现了系统故障，即表示系统的可用性变差。
  - 关注的方面：如何检测系统故障、故障发生的频度、出现故障时的表现、允许系统有多长时间非正常运行、如何防止故障发生、发生故障后如何消除故障、等等。
  - 错误检测(FaultDetection)
    - 命令响应机制(pingecho)、心跳(heartbeat)机制、异常机制(exception)；
  - 错误恢复(Recovery)
    - 表决、主动冗余(热重启)、被动冗余(暖重启/双冗余/三冗余)、备件(spare)；
    - Shadow操作、状态再同步、检查点/回滚；
  - 错误预防(Prevention)
    - 从服务中删除、事务、进程监视器。
- 可修改性(modifiability)
  - 可以修改什么——功能、平台(HW/OS/MW)、外部环境、质量属性、容量、等；
  - 何时修改——编译期间、构建期间、配置期间、执行期间；
  - 谁来修改——开发人员、最终用户、实施人员、管理人员；
  - 修改的代价有多大?
  - 修改的效率有多高?
  - 目标：减少由某个修改所直接/间接影响的模块的数量；
  - 常用决策：
    - 高内聚/低耦合、固定部分与可变部分分离、抽象为通用模块、变“编译”为“解释”；
    - 信息隐藏、保持接口抽象化和稳定化、适配器、低扇出；
    - 推迟绑定时间——运行时注册、配置文件、多态、运行时动态替换；
- 性能(performance)
  - 与时间有关：当外部请求到达时，系统将要耗费多少时间做出相应；
  - 因素：请求的数量和到达模式、请求的频度、耗费的资源；
  - 典型战术：
    - 提高计算效率(改进算法、引入cache等)、减少计算开销(最小化通讯数据量)、控制请求频度、限制执行时间、限制队列大小；
    - 引入并发、增加可用资源、负载平衡；
    - 对请求的调度策略(FIFO、固定优先级、动态优先级)；
- 安全性(security)
  - 安全性：系统在向合法用户提供服务的同时，组织非授权使用的能力
    - 未经授权试图访问服务或数据；
    - 试图修改数据或服务；

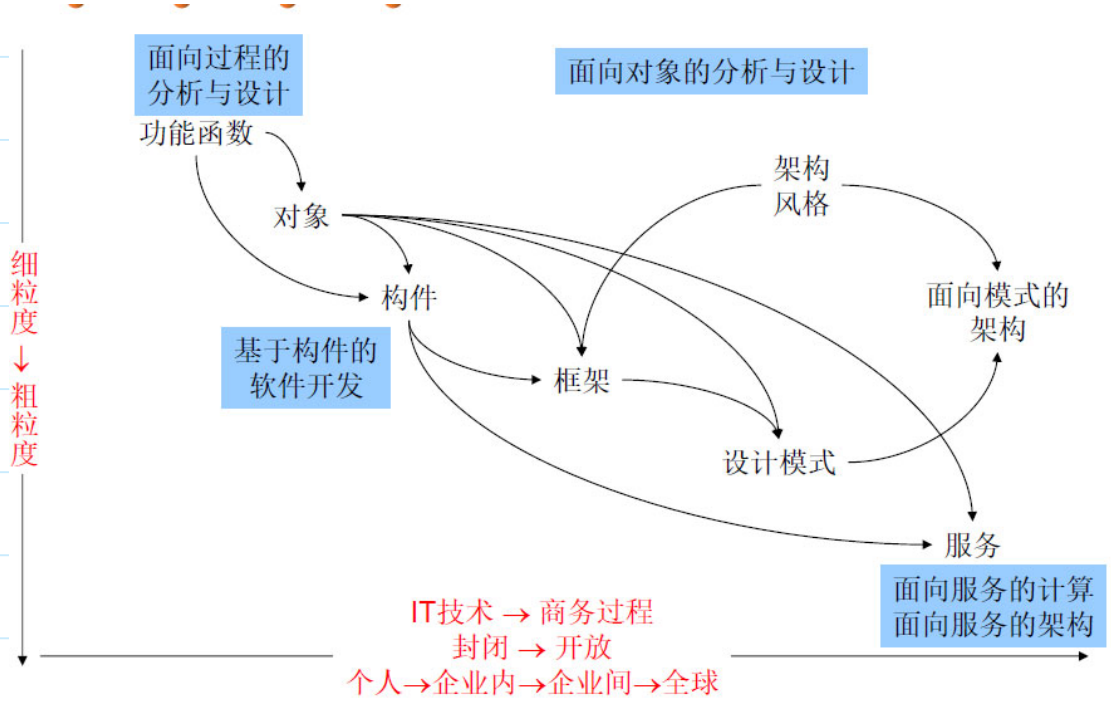
- 试图使系统拒绝向合法用户提供服务；
  - 关注点：抵抗攻击、检测攻击、从攻击中恢复。
  - 抵抗攻击——对用户进行身份认证、对用户进行授权、维护数据的机密性、限制暴露的信息、限制访问；
  - 检测攻击——模式发现、模式匹配；
  - 从攻击中恢复——将服务或数据回复到正确状态、维持审计追踪。

软件架构的发展与演化

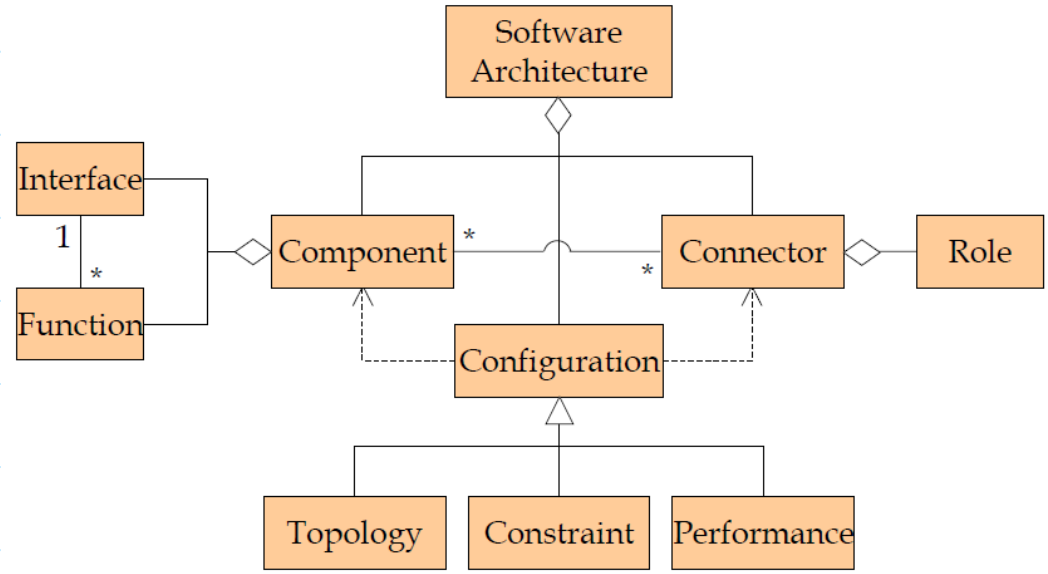
- 系统 = 算法 + 数据结构 (1960's)
- 系统 = 子程序 + 函数调用 (1970's)
- 系统 = 对象 + 消息 (1980's)
- 系统 = 构件 + 连接件 (1990's)
- 系统 = 服务 + 服务总线 (2000's)



软件架构的演化史

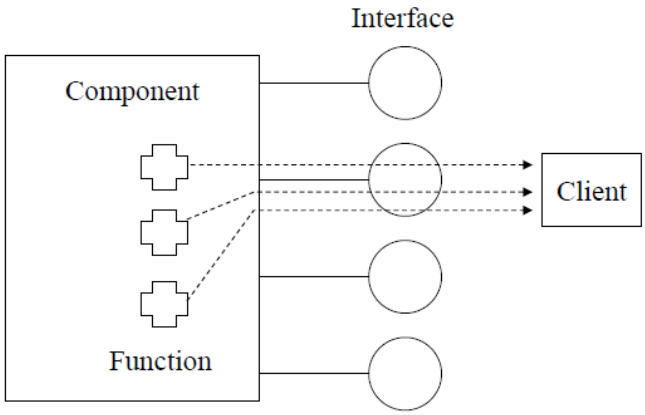


5 软件架构中的核心概念

- 

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

- 

接口(Interface)与功能(Function)

- 构件=接口+功能
- 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(Blackbox)；
- 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服务(service)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。

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）

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：
  - 过程调用、动态约束、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。

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。

连接的种类

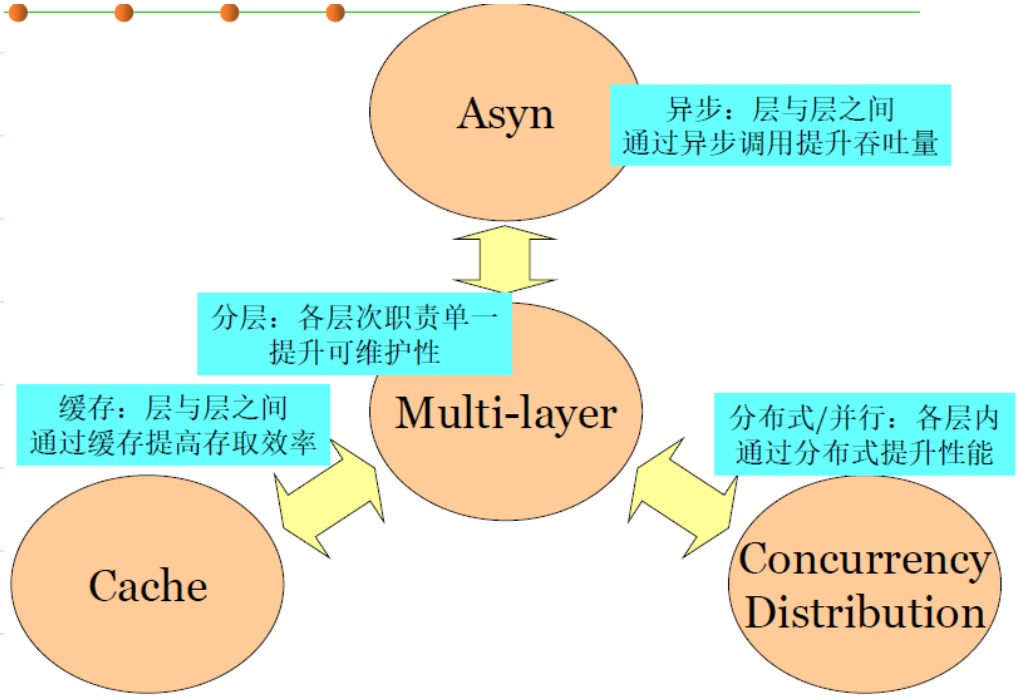
- 从连接目的与手段看
  - 函数/过程调用；A{call B.f();} B{f();}
  - 事件/消息发送；A{send msg(m) to B;} B{receive m and do sth;}
  - 数据传输；A{write data to DB;} B{read data from DB;}
- 除了连接机制/协议的实现难易之外，影响连接实现复杂性的因素之一是“有无连接的返回信息和返回的时间”，分为：
  - 同步(Synchronous)
  - 异步(Asynchronous)

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(procedurecall)
  - 事件广播(eventbroadcast)
  - 客户机服务器(clientserver)
  - 数据库连接(SQL)
- 典型连接件：CICS、MQ、JMS
- 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

6 软件架构的四大思想

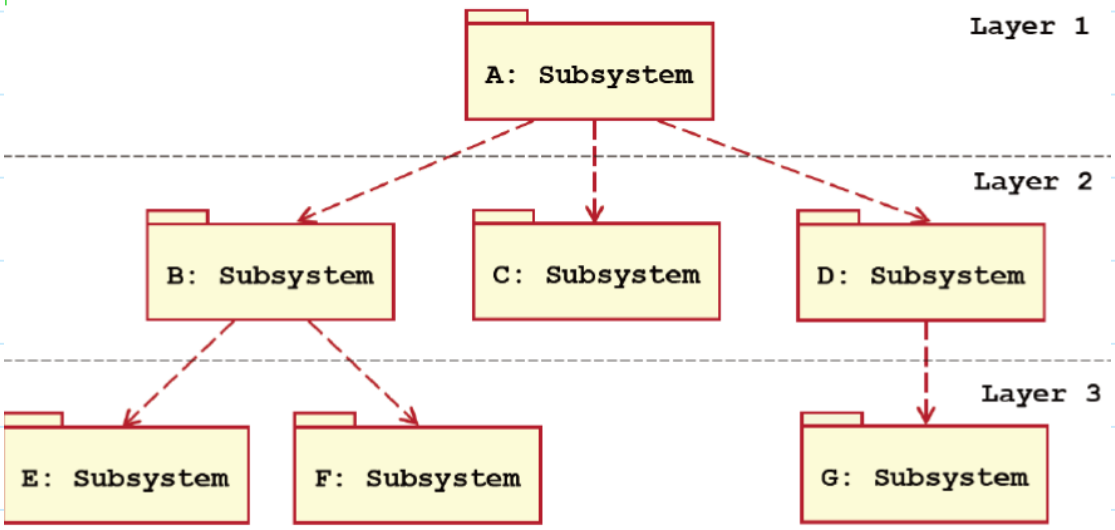
“软件架构四大器”

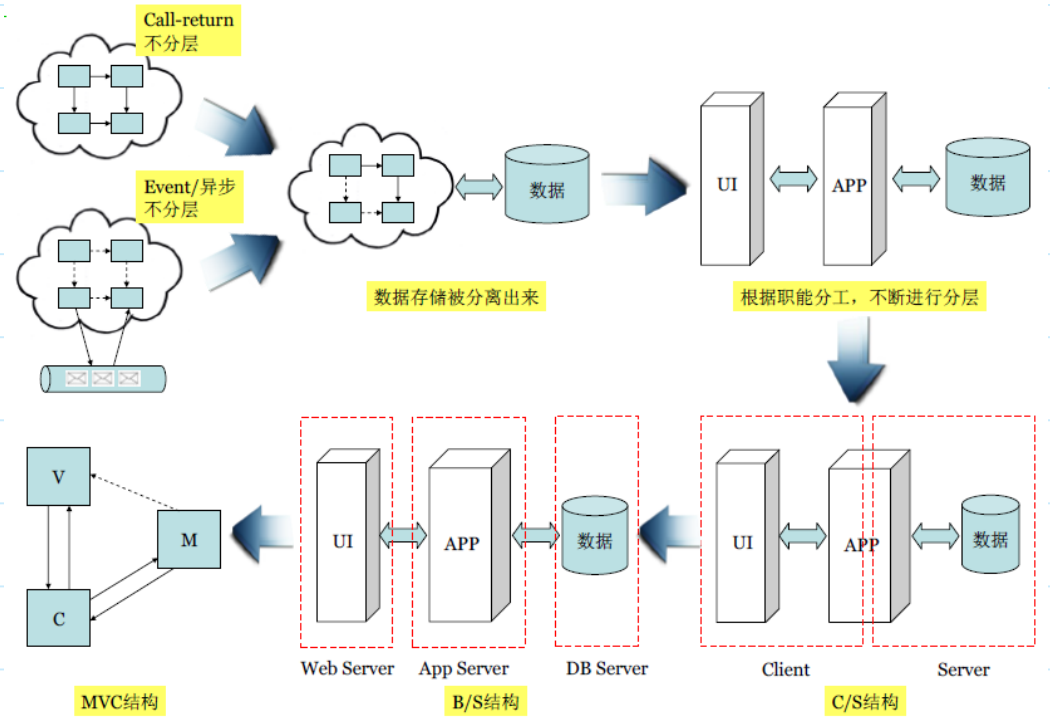
- 

软件架构的基本模式

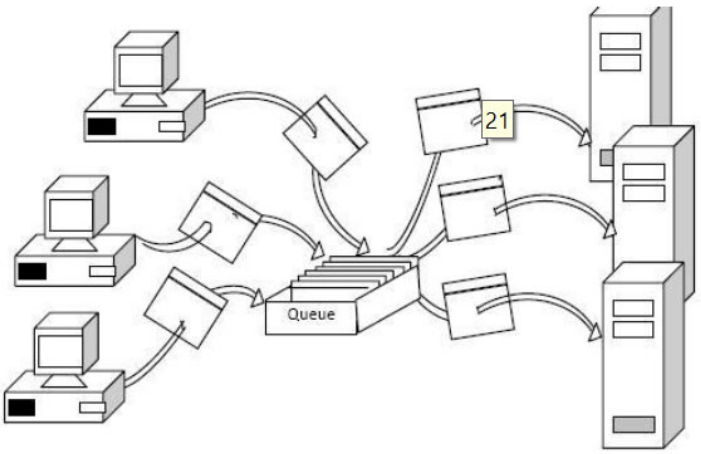
- 分层：C/S、B/S、多层，数据、计算与显示的分离(MVC)
  - 一个模块做很多事情->各负其责，分工明确
  - 牺牲了效率，提升了可维护性。
- 异步：事件、消息
  - 请求之后等待结果(同步)->请求之后继续执行，后续等待结果(异步)
  - 性能(吞吐量)提高，但实时性变差；
- 缓存：页面缓存、数据缓存、消息缓存
  - 直接到源头去取->预取
  - 提高了效率，但牺牲了准确性
- 并发(分布式)：集群、负载均衡、分布式数据库
  - 原本一个模块处理很多请求->多个模块共同处理这些请求
  - 提高了吞吐量、响应时间、可靠性，但需要考虑同步等复杂问题

分层

- 

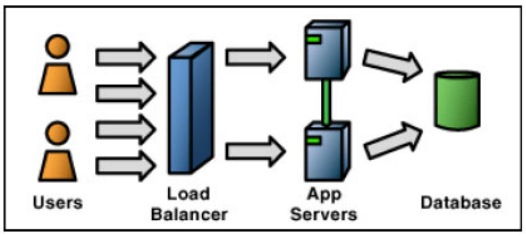
- 

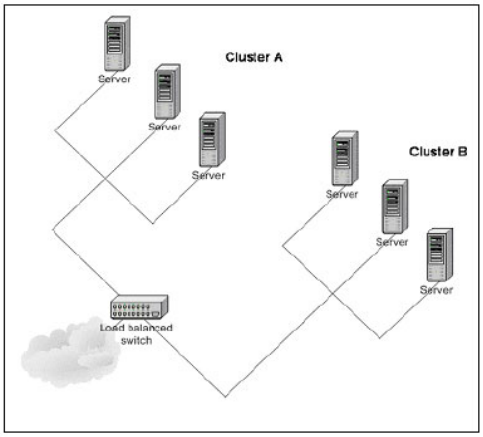
异步

- 通过“第三者”——消息——完成模块之间的功能调用。
- 

分布式+并发

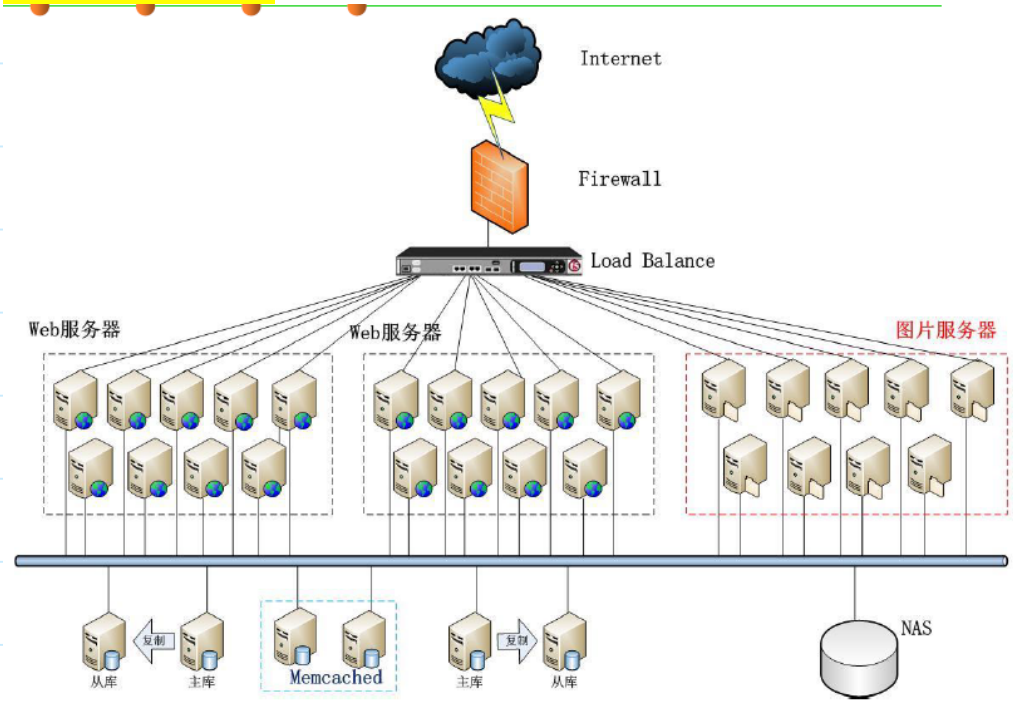
- 事实上，功能层并不一定只驻留在同一台服务器上，数据层也是如此；
- 如果功能层(或数据层)分布在多台服务器上，那么就形成了基于集群(Cluster)的C/S物理分布模式。





集群(Cluster)

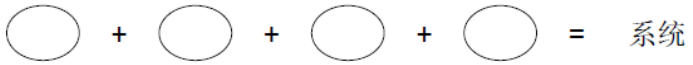
- 一组松散耦合的服务器，共同协作，可被看作是一台服务器
- 用来改善速度、提高可靠性与可用性，降低成本
- 负载平衡是集群里的一个关键要素
- 物理集群、逻辑集群



集群的方法

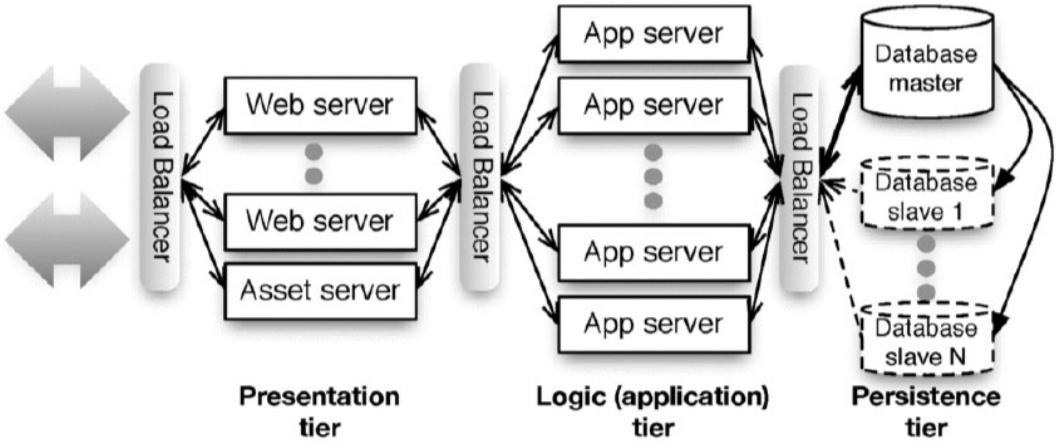
- 集群内各服务器上的内容保持一致(通过并发/冗余提高可靠性与可用性)
  - ○ = ○ = ○ = ○ = 系统
- 集群内各服务器上的内容之和构成系统完整的功能/数据，是对系统功能集合/数据集合的一个划分

(通过分布式提高速度与并发性)

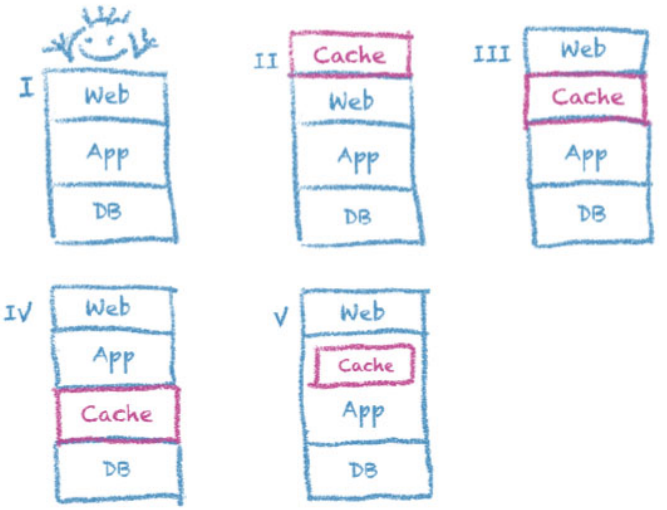
- 

负载均衡(LoadBalance)

三个层次的负载均衡

- 

缓存(cache)

- 缓存：解决web应用性能瓶颈。
  - “缓存就像清凉油，哪里不舒服，抹一下就好了！”
- 性能瓶颈的体现：高延时、拥塞和服务器负载
  - “下载HTML只需要总用户响应时间的10-20%，剩下的80-90%全部用于下载页面中的其它组成内容(如各种图像等)”
- 缓存：
  - 一种临时存储，将数据复制到不同于原始数据源的位置，访问缓存数据的速度比访问原始数据的速度要快得多，从而可以减小服务器负载和带宽消耗，提高用户性能。
- Cache可以处在多个不同的位置
  - 

一个典型的分布式系统基础架构(Hadoop)

- Hadoop是一个由Apache基金会开发的分布式系统基础架构，是一个能够对大量数据进行分布式处理的软件框架，以一种可靠、高效、可伸缩的方式进行数据处理。用户可以在不了解分布式底层细节的情况下，开发分布式程序，充分利用集群的威力进行高速运算和存储。Hadoop框架最核心的设计是HDFS和MapReduce。HDFS为海量的数据提供存储，MapReduce为海量的数据提供计算。
- Hadoop的优点：
  - 高可靠性：Hadoop按位存储和处理数据的能力值得人们信赖。
  - 高扩展性：Hadoop是在可用的计算机集簇间分配数据并完成计算任务的，这些集簇可以方便地扩展到数以千计的节点中。
  - 高效性：Hadoop能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。
  - 高容错性：Hadoop能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。
  - 低成本：与一体机、商用数据仓库以及QlikView、YonghongZSuite等数据集市相比，hadoop是开源的，项目的软件成本因此会大大降低。
- hadoop大数据处理的意义
- Hadoop在大数据处理应用中广泛应用得益于其自身在数据提取、变形和加载(ETL)方面上的天然优势。Hadoop的分布式架构，将大数据处理引擎尽可能的靠近存储，对例如像ETL这样的批处理操作相对合适，因为类似这样操作的批处理结果可以直接走向存储。Hadoop的MapReduce功能实现了将单个任务打碎，并将碎片任务(Map)发送到多个节点上，之后再以单个数据集的形式加载(Reduce)到数据仓库里。