软件体系结构设计

清华大学软件学院 刘强

教学提纲

1. 软件体系结构概述
2. 软件体系结构设计原则
3. 软件体系结构风格

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仅仅是数据结构设计、算法选择和数据库构造吗？

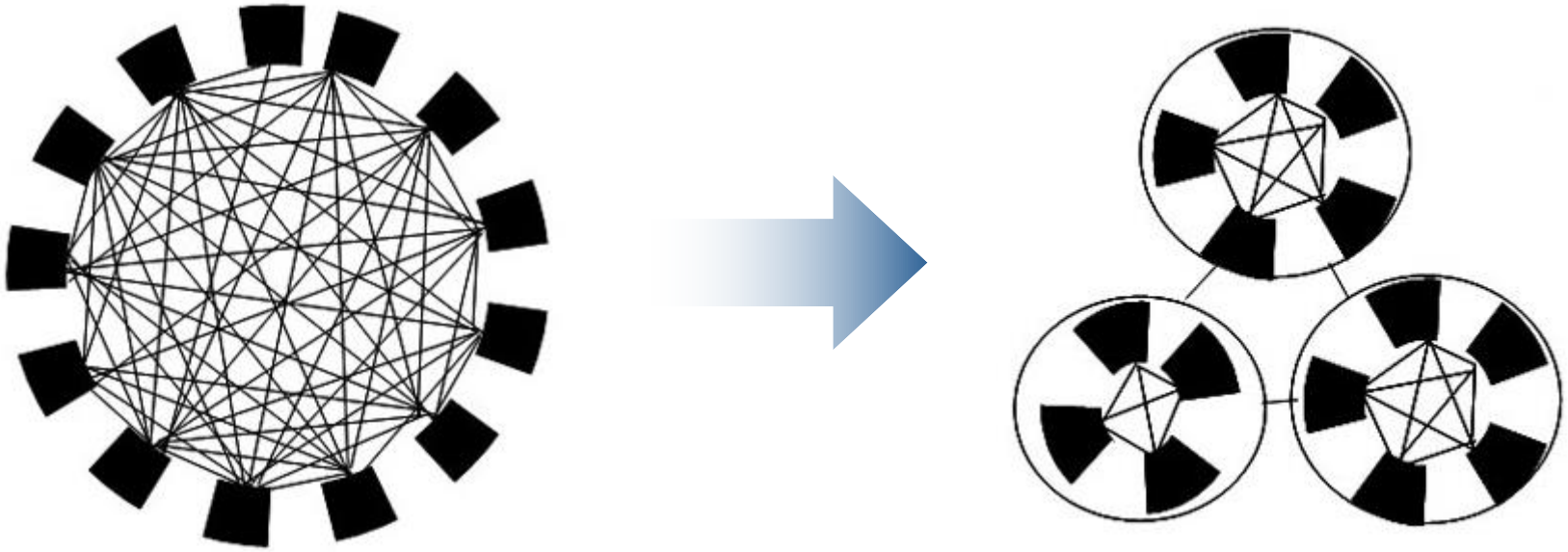
处理复杂性

分而治之

- 将一个复杂的问题分解成若干个简单的问题，然后逐个解决
- 来源于人们生活与工作的经验，完全适合于技术领域

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对算法的选择和数据结构的设计明显重要得多。



一个平面规划的例子

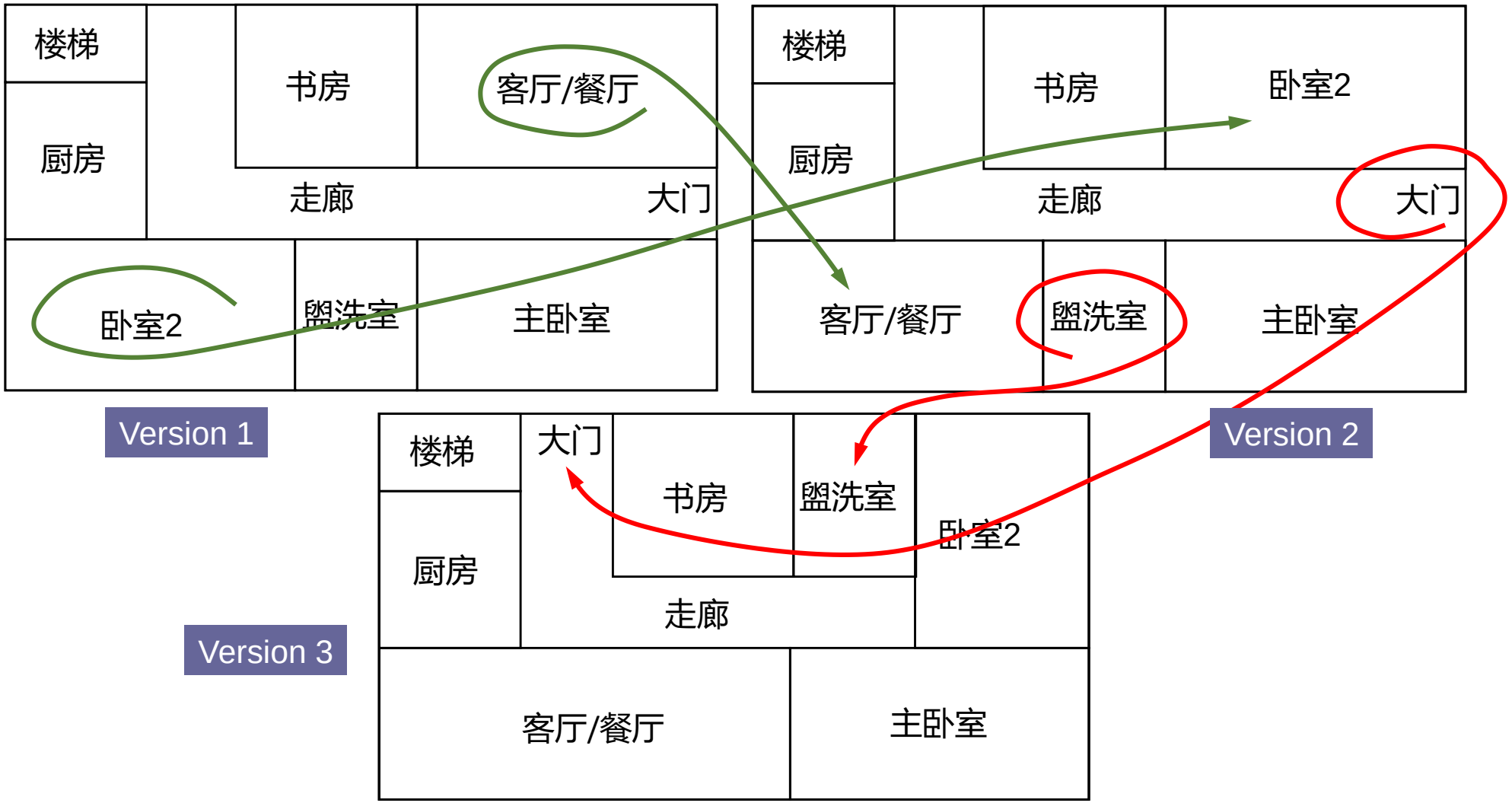
住宅设计的约束条件：

- 适合一对夫妻与一个孩子共同生活
- 有单独学习和会客的空间
- 住户每天行走的路程要求最短
- 卧室的白天采光量要求达到最大

假设：住户的大部分时间集中在客厅/餐厅和主卧室的区域内活动

如何给出满足上述约束的平面规划方案？

一个平面规划的例子


楼梯
厨房
书房
客厅/餐厅
走廊
大门
卧室2
盥洗室
主卧室
Version 1
楼梯
厨房
书房
卧室2
走廊
大门
客厅/餐厅
盥洗室
主卧室
Version 2
楼梯
厨房
大门
书房
盥洗室
卧室2
走廊
客厅/餐厅
主卧室
Version 3

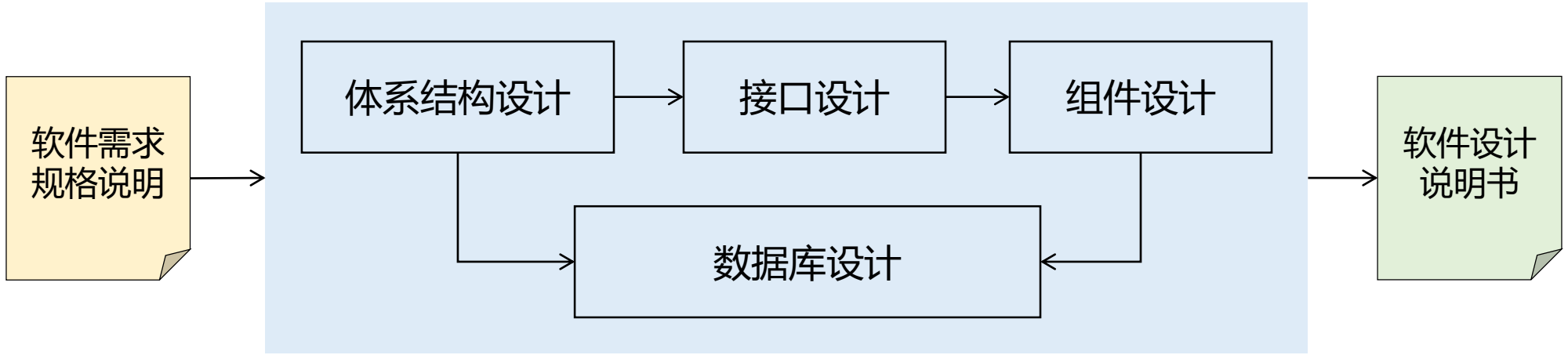
一个平面规划的例子

与软件工程概念的映射

	建筑结构概念	软件工程概念
构件	房间	子系统或模块
接口	门	服务
非功能需求	生活区	响应时间
功能需求	住房	用例（或用户故事）
返工代价	移动墙壁	子系统接口的改变

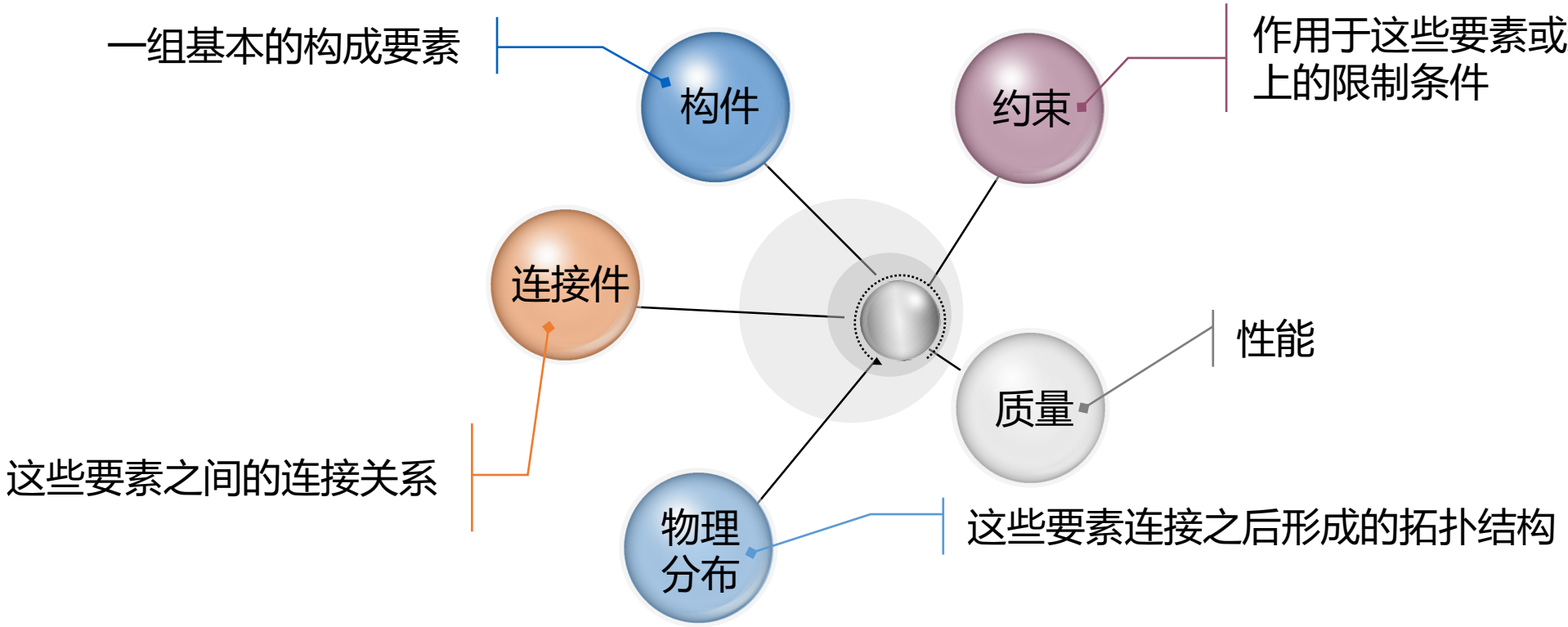
软件设计元素

软件设计是根据需求规格说明，确定软件体系结构，进一步设计每个系统部件的实现算法、数据结构及其接口等。



软件体系结构

软件体系结构是软件系统的顶层结构，它是经过系统性地思考，权衡利弊之后在现有资源约束下进行合理决策，最终明确的系统骨架。

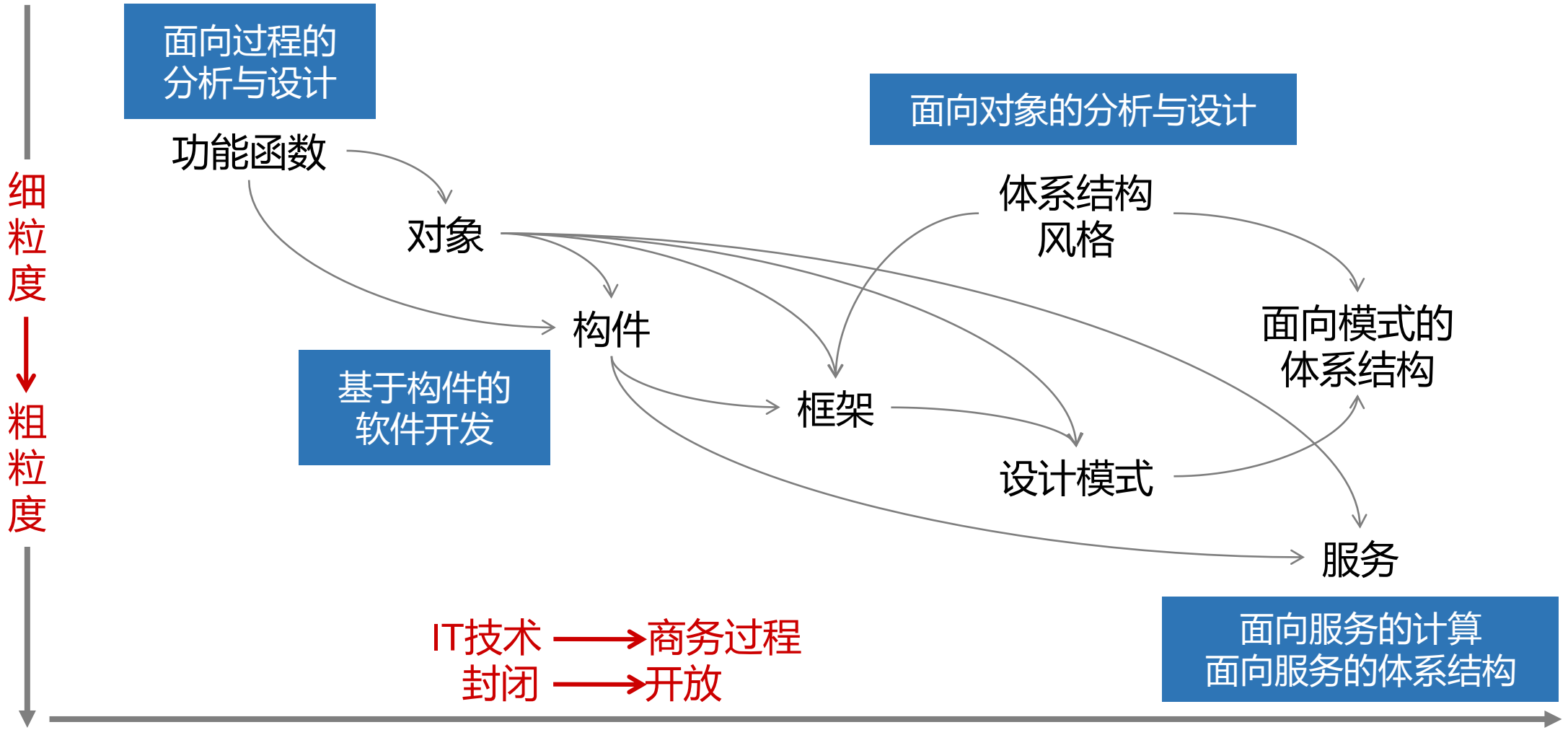


软件体系结构

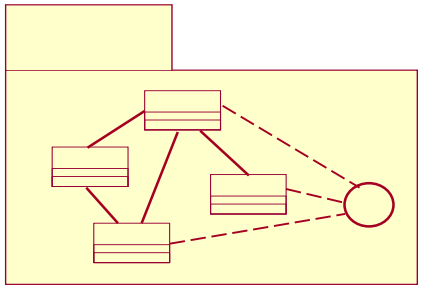
软件体系结构 = 构件 + 连接件 + 约束

- 构件是具有某种功能的可复用的软件结构单元。
- 构件作为一个封装实体，只能通过其接口与外部环境交互。
- 构件的功能以服务的形式体现出来，并通过接口向外发布，进而产生与其它构件之间的关联。
- 连接件表示构件之间的交互并实现构件之间的连接，它是负责完成构件之间信息交换和行为联系的专用构件。
- 约束是连接的规约，其目的是使双方能够互相理解对方所发来的信息语义，包括操作/过程调用、控制/事件/消息发送、数据传输等。

软件体系结构的发展


面向过程的
分析与设计
功能函数
对象
构件
基于构件的
软件开发
框架
面向对象的分析与设计
体系结构
风格
面向模式的
体系结构
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

子系统与接口

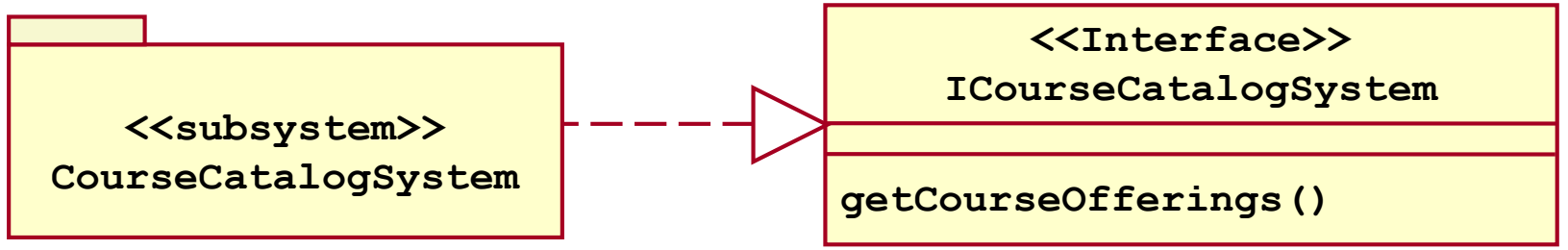


- 子系统的特征是由该子系统提供给其他子系统的服务来刻画的，一个服务是一组分担公共目的的相关操作。
- 子系统提供给其他子系统的一组操作称为子系统接口，包括操作名、操作参数、类型及其返回值。

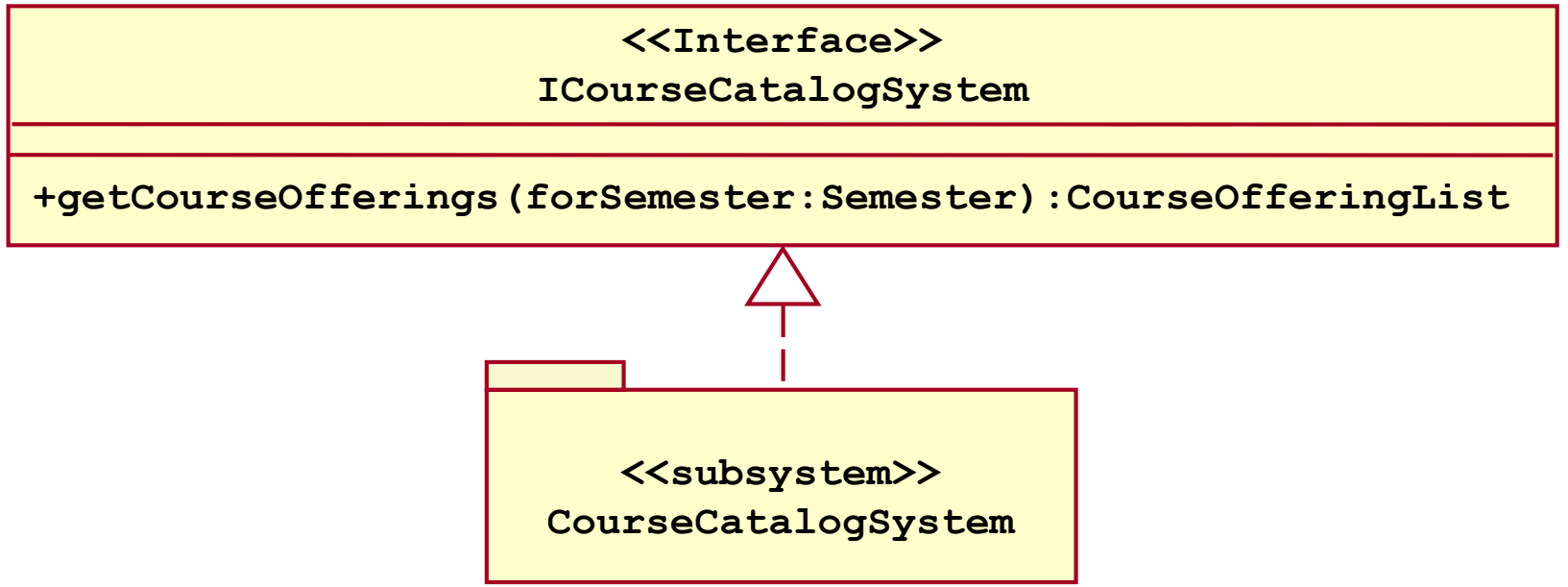
- 体系结构设计关注于定义子系统提供的服务，即列出所有操作以及这些操作的参数和操作的高层行为。
- 接口设计关注于应用程序接口（API），求精并扩展子系统接口，也包括各个操作参数的类型和返回值。

子系统与接口

体系结构设计
子系统服务



接口设计
子系统接口



模块与组件

模块：根据不同的标准，通常称作程序模块或功能模块。程序模块是一段能够实现某个目标的成员代码段，功能模块则用来说明一个功能所包含的系统行为。

组件：组件也称为构件，它是封装了一个或多个程序模块的实体，主要强调的是封装，利用接口进行交互。

	目的	特点	接口	表现形式	架构定位
模块	隔离、解耦	高内聚、低耦合	无统一接口	业务框架、业务模块	横向分块
组件	重用、封装	高重用、低耦合	统一接口	基础库、基础组件	纵向分层

框架与架构

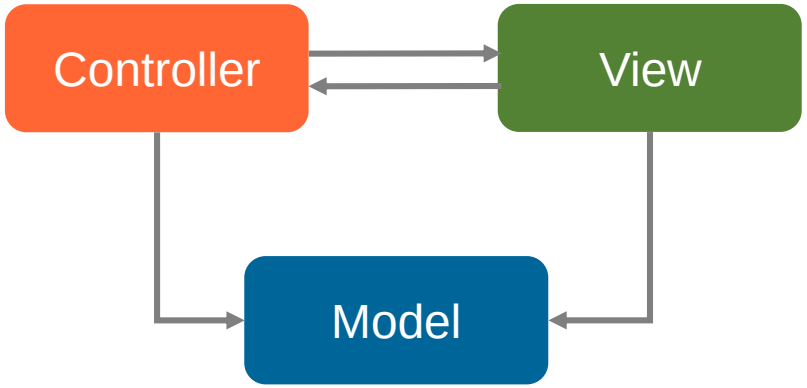
软件架构：也称软件体系结构。软件架构分析工程中的问题，针对问题设计解决方案，针对解决方案分析应具有的功能，针对功能设计软件系统的层次和模块及层次模块之间的逻辑交互关系，确定各个功能如何由这些逻辑实现。

软件框架：软件开发过程中提取软件的共性部分形成的体系结构。框架不是现成可用的应用系统，而是一个半成品。它是提供了诸多服务，供开发人员进行二次开发，实现具体功能的程序实体。

二者之间的关系：框架不是架构，框架比架构更具体，更偏重于技术，而架构更偏重于设计；架构可以通过多种框架来实现。

框架与架构

- 体系结构的呈现形式是一个设计规约，而框架则是程序代码；
- 体系结构的目的是指导软件系统的开发，而框架的目的是设计复用。



MVC是一种体系结构

ANGULARJS by Google

BACKBONE.JS

django

MVC结构可以用多种框架实现

教学提纲

1. 软件体系结构概述
2. 软件体系结构设计原则
3. 软件体系结构风格

软件体系结构设计

在需求分析的基础上，根据项目设计目标，将系统逐步分解和求精为更小的若干单元。

明确设计目标	应考虑软件设计质量，尤其是非功能需求，如性能、可扩展性等。
初始子系统分解	比较不同的设计方案，可采用标准的体系结构风格，需要考虑系统构造策略，诸如软硬件环境、数据存储、系统控制、访问控制等。
不断分解和求精	初始分解也许不能满足所有设计目标，需要进一步分解和求精。
任务组织和分配	不同的子系统或模块分配给不同的团队或者开发人员完成，由他们协商定义子系统的服务及其接口。

系统设计目标

性能

- 性能主要包括响应时间、吞吐量、存储空间等指标。
- 如果性能很关键，体系结构设计就要定位在少数几个子系统的一些关键性操作上，最大限度地减少这些子系统与其他子系统之间的通信。

安全性

- 安全性是指软件保护信息和数据的安全能力。
- 如果信息安全是一个关键性需求，那么体系结构设计采用分层结构，把重要的资源放在内层，并在每一层中采用更加严格的信息加密和安全性验证。

系统设计目标

可靠性

- 应避免内部或者外部的错误而导致失效，即使系统失效也要有重新恢复的能力。
- 如果可靠性很关键，那么体系结构设计就要有一定的冗余性，以便在无需系统停止运行的情况下更新或更换组件或设施。

可维护性

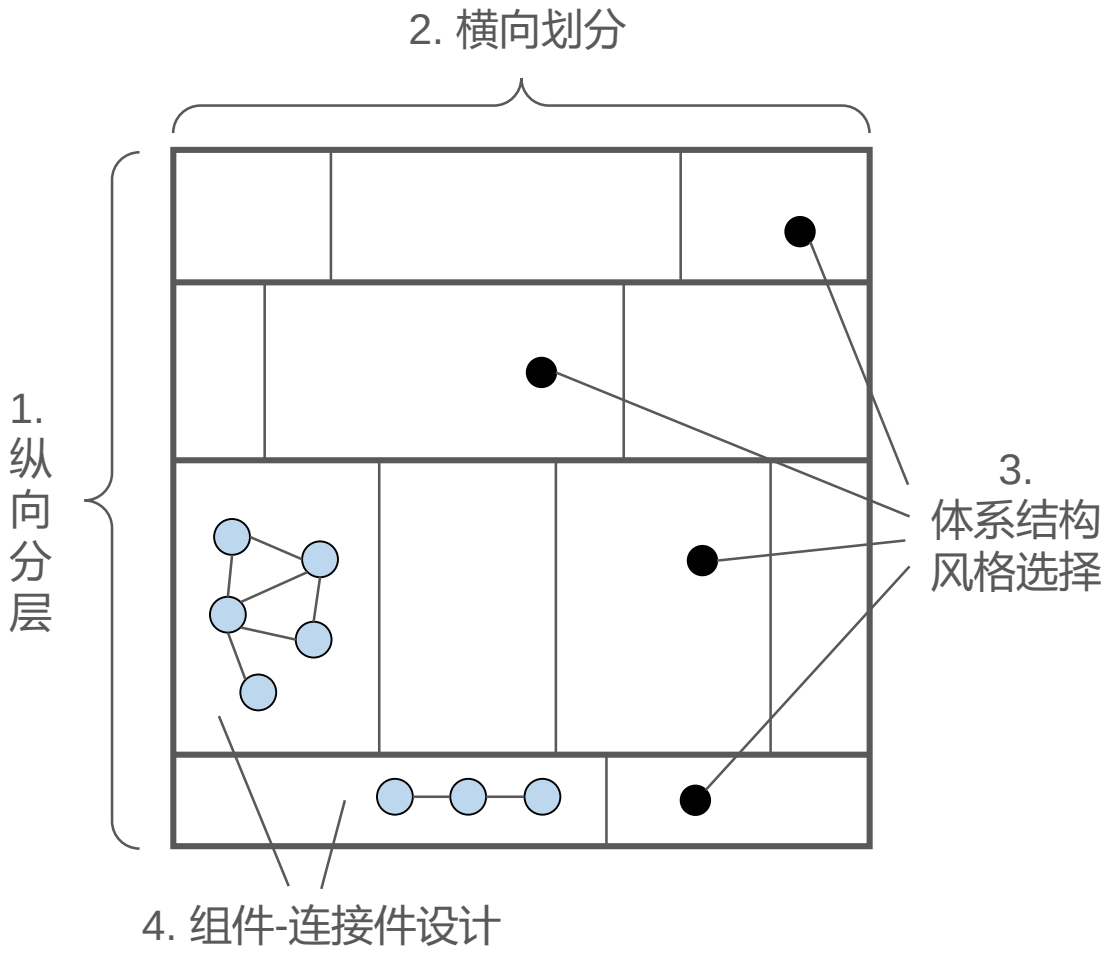
- 系统应具有可扩展性，以便于修改现有功能或增加新功能。
- 体系结构设计要使每一个组件具备最大可能的自我完备性，可独立运行、监控、部署、配置与禁用，并与其它组件之间以松散耦合的方式进行协作。

软件体系结构的选择

基本分析方法：功能和复杂性的分解

- 纵向分解：分层次
- 横向分解：子系统、模块
- 针对每一个分解后得到的模块，根据该问题领域的特性，选择行为模式
- 组件、连接件设计

从大粒度风格入手，逐渐细化



软件体系结构的选择

技术因素

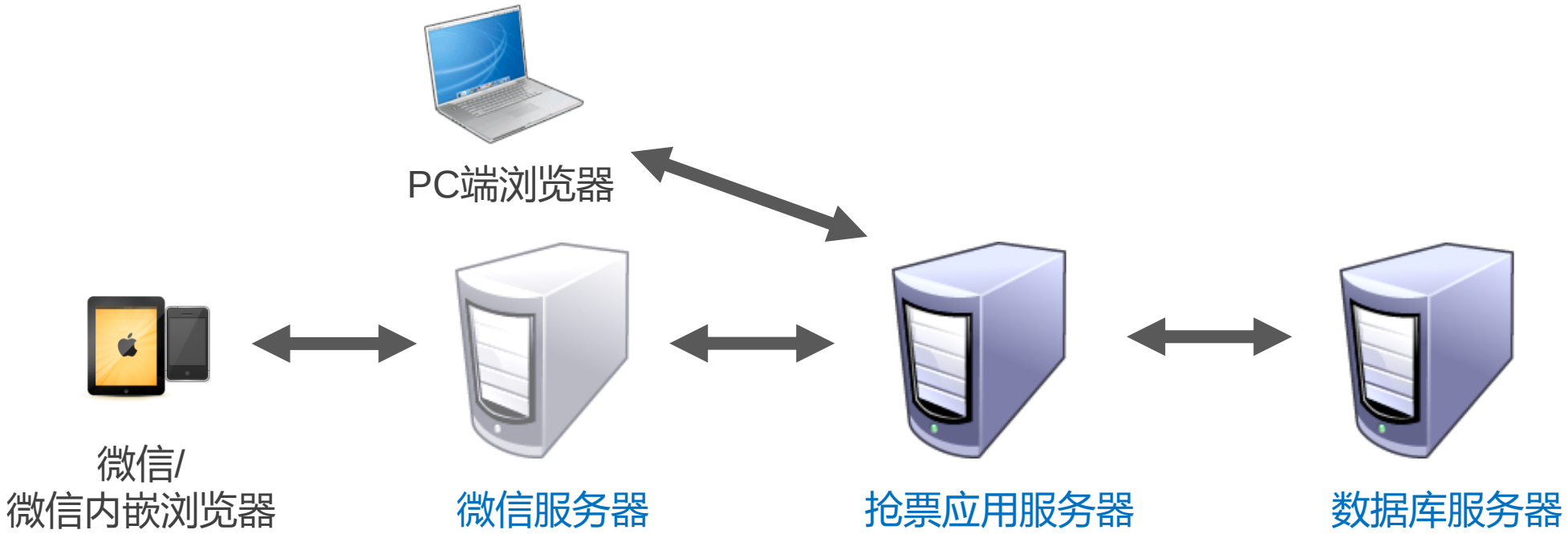
- 使用何种构件、连接件
- 在运行时，构件之间的控制机制是如何被共享、分配和转移
- 数据如何通讯
- 数据与控制如何交互

质量因素

- 性能：时空复杂性
- 安全性：保护信息和数据不泄露或被非法获取
- 可修改性：算法的变化；数据表示方式的变化；系统功能的可扩展性

系统部署方案

系统部署方案是描述系统运行期间构件和硬件节点之间的关系，在系统设计阶段处理软件/硬件的映射问题，可能会增加新的子系统或模块的定义。



数据库选择策略

- 开源的关系型数据库
- 简单易用，大量第三方插件，社区活跃，文档丰富
- 支持快速的复杂查询操作
- 支持完整的事务操作
- 具有较高的安全性

- 模式自由
- 支持海量数据的查询和插入
- 自动支持分片等分布式操作
- 支持故障恢复与备份
- 需要占用很大空间建立索引
- 不支持事务操作
- 高安全级别无法保证

- 近年兴起的内存数据库
- 访问速度非常快
- 在保证访问速度的同时，也能够进行持久化存储
- 可支持很多数据结构
- 数据在内存十分不可靠
- 不完整的事务实现

数据库选择策略

考虑一个抢票应用，功能为在某个时刻为大量用户进行抢票服务。要求能够在10s内为5000个用户正确地返回抢票结果。

(1) 用户数据需要持久化存储，考虑 mysql 和 mongo。

(2) 事务性质，不能有一张票被两个人同时抢到，考虑mysql。

(3) 要求快速响应，需要在短时间内返回结果，考虑内存数据库。

考虑到业务场景的复杂性，我们尝试能否对应用场景进行分离，即数据持久化存储用mysql，抢票的时候则使用内存数据库进行响应。然而需要注意的是在进行业务响应时，因为采用内存数据库，那么将不能保证事务性质，怎么办?

软件体系结构设计原则

抽象原则：集中表现事物的主要特征和属性，隐蔽和忽略细节部分

分解原则：大的系统分解成若干小的模块，保证“高内聚低耦合”

面向抽象编程：接口和实现分离，注重接口而非实现

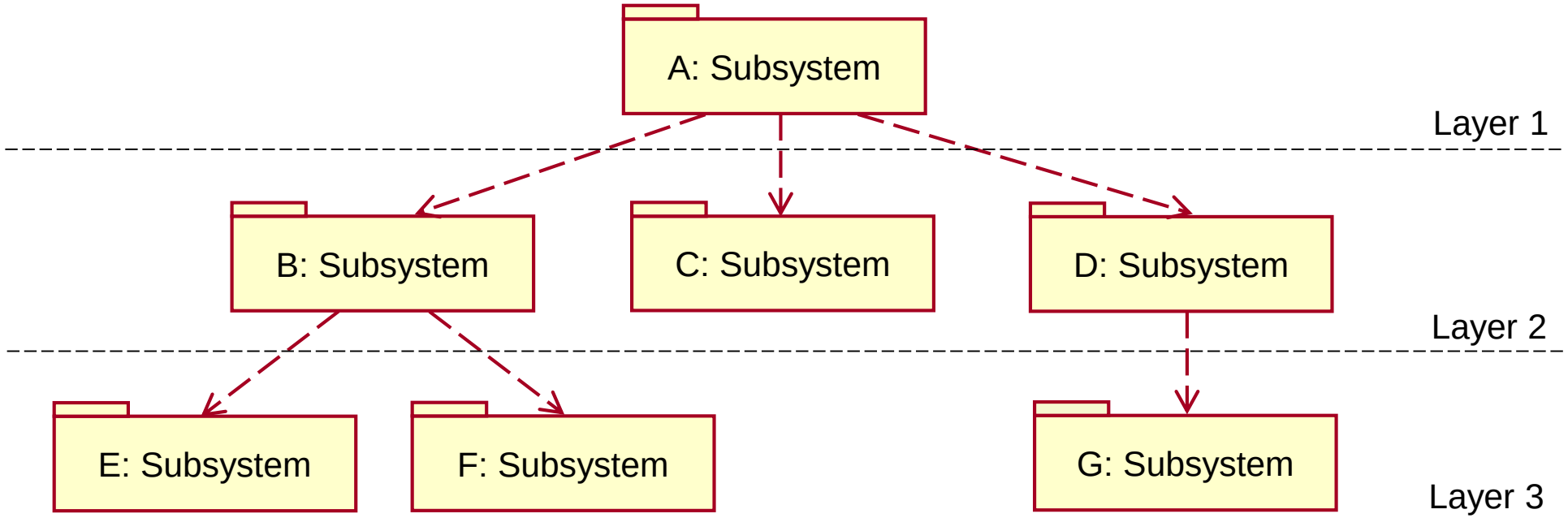
共享原则：最大化重用数据、计算资源、业务组件等资产

冗余原则：重复配置组件实例，增加系统的可靠性

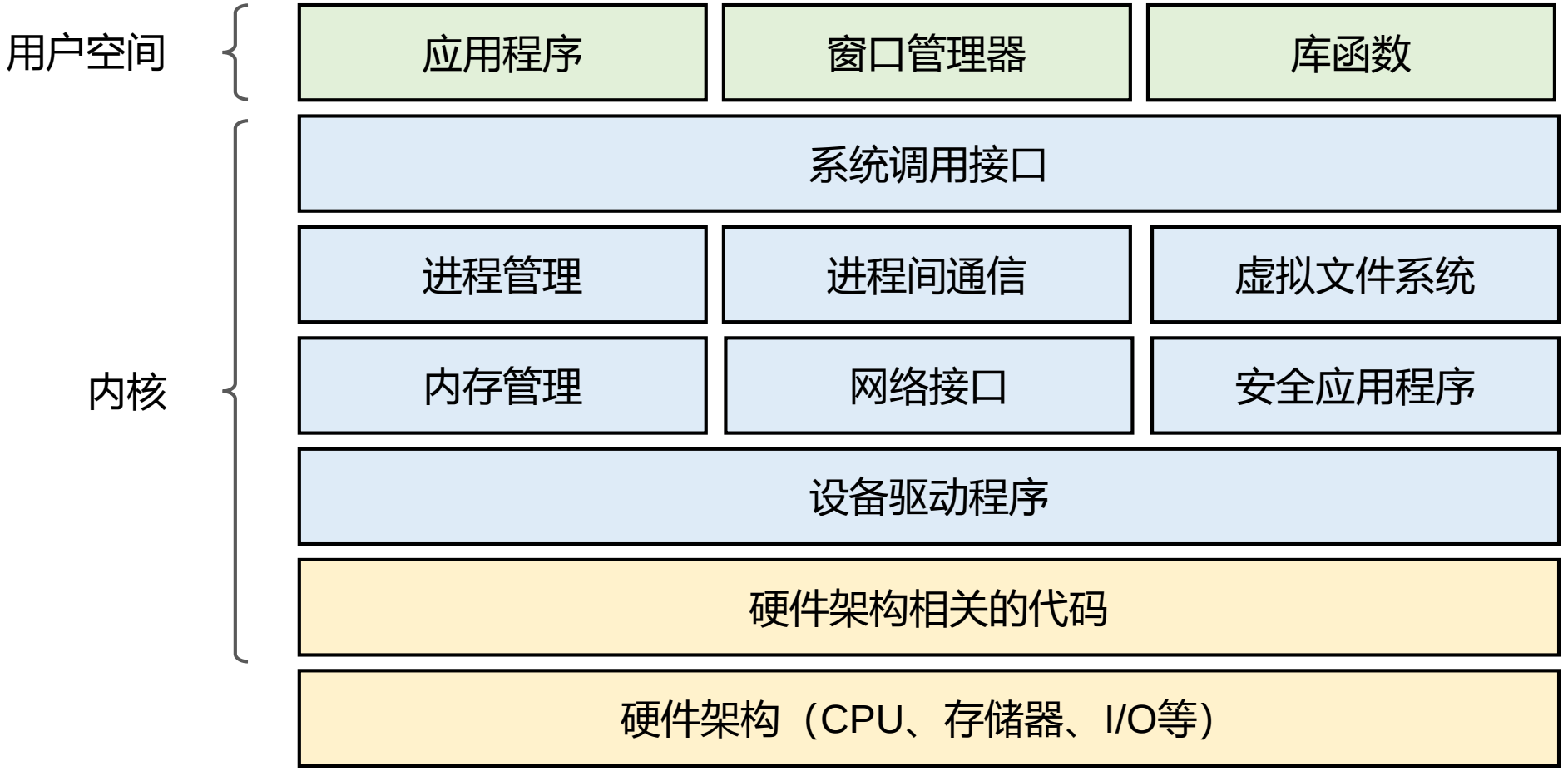
避免过度设计：软件体系结构是逐步演化形成的

软件体系结构设计原则

抽象原则：各平台（含基础设施、中间件技术服务、各层业务服务等）需要通过合理地抽象，将内部信息、处理与扩展能力聚合成标准的服务于扩展接口，并通过统一的形式提供给使用者，屏蔽内部的实现与运行细节。



举例：操作系统层次结构


用户空间
应用程序
窗口管理器
库函数
内核
系统调用接口
进程管理
进程间通信
虚拟文件系统
内存管理
网络接口
安全应用程序
设备驱动程序
硬件架构相关的代码
硬件架构（CPU、存储器、I/O等）

软件体系结构设计原则

高内聚低耦合：把模块间的耦合降到最低，努力让一个模块做到精益求精。

耦合是两个子系统之间依赖关系的强度。

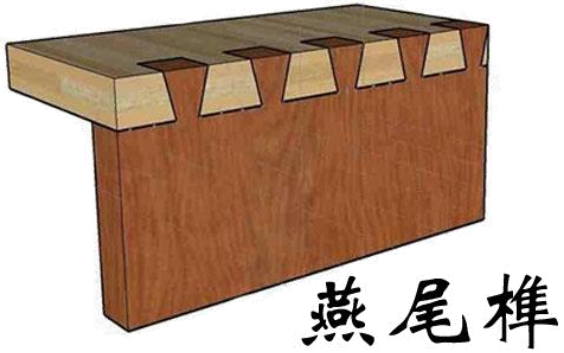
- 如果两个子系统是松耦合的，二者相互独立，那么当其中一个发生变化时对另一个产生的影响就很小；如果它们是紧耦合的，则其中一个发生变化就可能对另一个产生较大影响。

内聚是子系统内部的依赖程度。

- 如果某个子系统含有许多彼此相关的对象，并且它们执行类似的任务，那么它的内聚性比较高；如果某个子系统含有许多彼此不相关的对象，它的内聚性就比较低。

软件体系结构设计原则

面向抽象编程：把模块的行为抽象出来，形成相对固定的接口，这样不论对接口的实现如何变化，只要接口不变，就不会影响使用接口的模块。



例如：木工里常见的榫，它不关心不同部件的实现方式，只是通过连接部分将不同部件严密地结合在一起，形成一个完整的产品。

注重接口，而不是实现，依赖接口，而不是实现。接口是抽象稳定的，实现则是多种多样的。

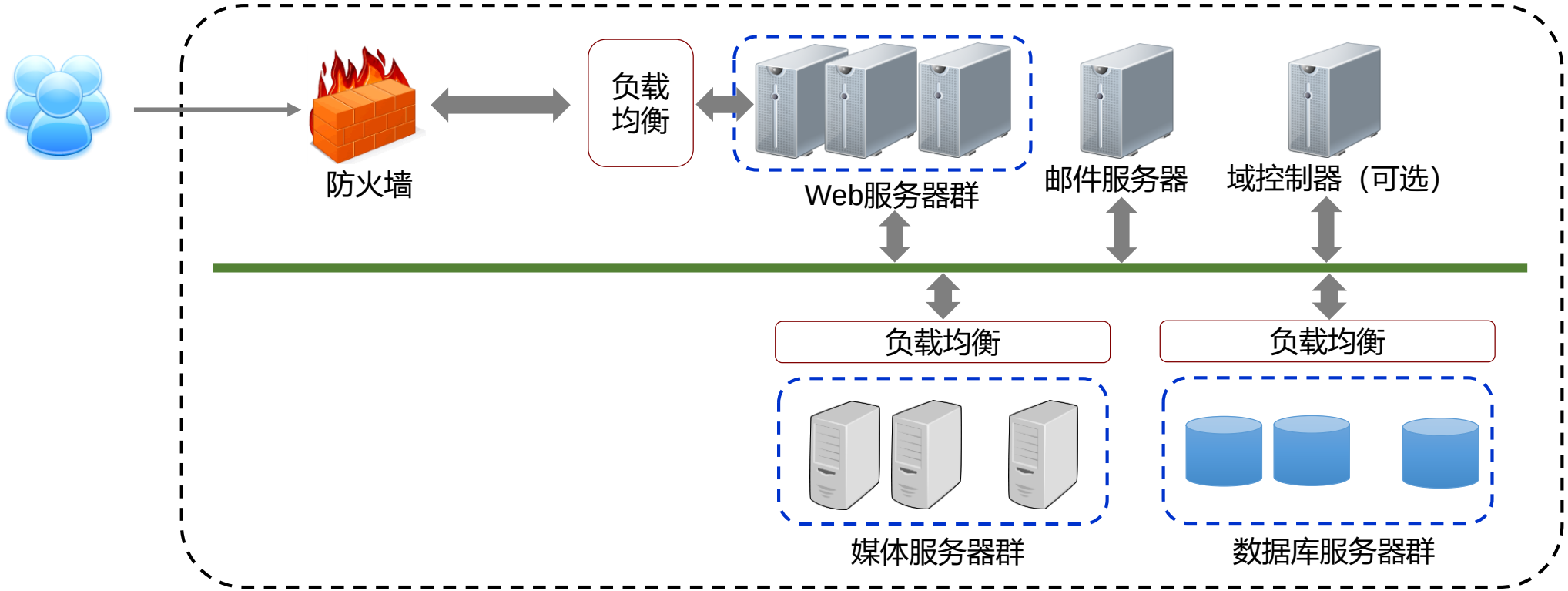
软件体系结构设计原则

共享原则：最大化重用数据、计算资源、业务组件等资产，防止数据、逻辑与技术实现的不一致性带来的管理复杂性，避免重复建设成本与管理成本，通过安全机制保证共享资产的合法使用，通过业务分级保障共享资源效益最大化。

- 统一的身份认证、访问控制与加解密机制
- 同一业务服务或技术服务有唯一提供者
- 同一数据有唯一可信源
- 控制技术多样性，但需要同时防止厂商绑定
- 为共享服务提供多租户能力
- 提供业务分级能力，对不同级别的业务提供区分服务

软件体系结构设计原则

冗余原则：各组件（计算资源、业务组件、数据等）须有充分合理的冗余实例，保证单一实例失效不影响业务正常运行，或可以通过切换备份实例快速恢复，不会丢失不可恢复的数据。



软件体系结构设计原则

避免过度设计：只考虑和设计必须的功能，避免过度设计。只实现目前需要的功能，以后需要更多功能时，可以再进行添加。

举例：假设你打算开发一个知乎的客户端，这时你开始琢磨：面向对象、扩展、技术要牛…

- 业务和界面要分开：先用库的形式搞一个知乎SDK，于是设计了一大堆接口
- 接口要通用：现在的社区网站有一定的共同点，我要把他们的SDK封装到一起！于是开始阅读各种API文档，微博、腾讯、天涯、豆瓣、甚至把Email都放进去了
- 实现也要通用：把IOS、Android和桌面系统的底层都封装起来。。。这事有点难
- 支持各大网站：采用插件机制，于是开始设计抽象UI部分。。。

软件体系结构设计原则

如果你的设计存在以下想法，就要警惕是否过度设计：

- 设计决策以是否面向扩展为首要衡量标准
- 解决问题的第一思路是设计模式
- 相信软件质量会随着使用的框架、库的增加而提升
- 追求每一个模块或层的可替换性
- 如果没有一些UML图等外在因素的辅助，很难看懂你的代码
- 大量零散接口的引入并强调是面向接口编程

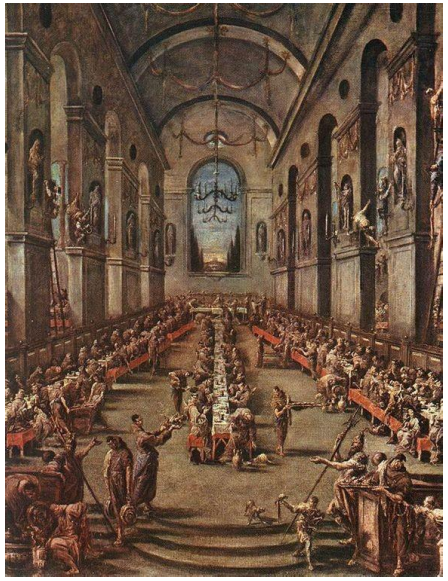
另一个极端就是设计不足，设计出来的系统复用性差，扩展性不强，不能灵活的应对变化。

教学提纲

1. 软件体系结构概述
2. 软件体系结构设计原则
3. 软件体系结构风格

从“建筑风格”谈起

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。

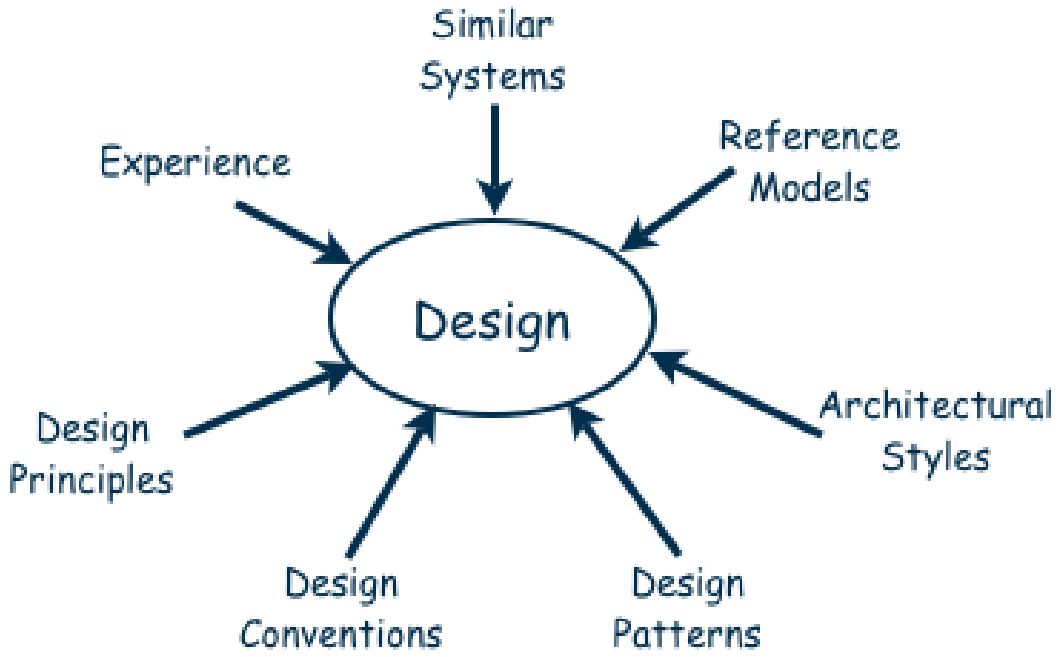


文艺复兴建筑风格

巴洛克建筑风格

软件体系结构风格

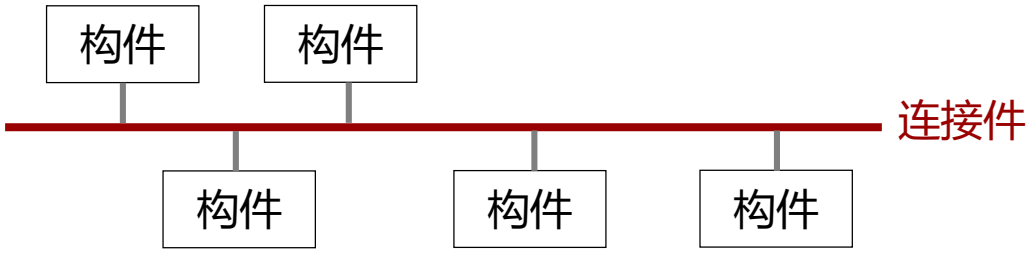
大部分的软件设计是例行设计，即有经验的软件开发人员经常会借鉴现有的解决方案，将其改造成新的设计。



软件体系结构风格

软件体系结构风格（Architectural Styles）是描述特定系统组织方式的惯用范例，强调了软件系统中通用的组织结构。

- 一组构件类型，诸如数据容器、函数、对象、类等
- 一组连接件类型，诸如管道、过程调用、事件广播、数据库连接等
- 这些构件的拓扑分布
- 一组对拓扑和行为约束
- 一些对风格的成本和收益的描述



常见的软件体系结构风格

调用-返回系统：主程序-子程序，面向对象，层次结构

数据流系统：批处理，管道-过滤器

以数据为中心系统：数据库，超文本系统，仓库（黑板）

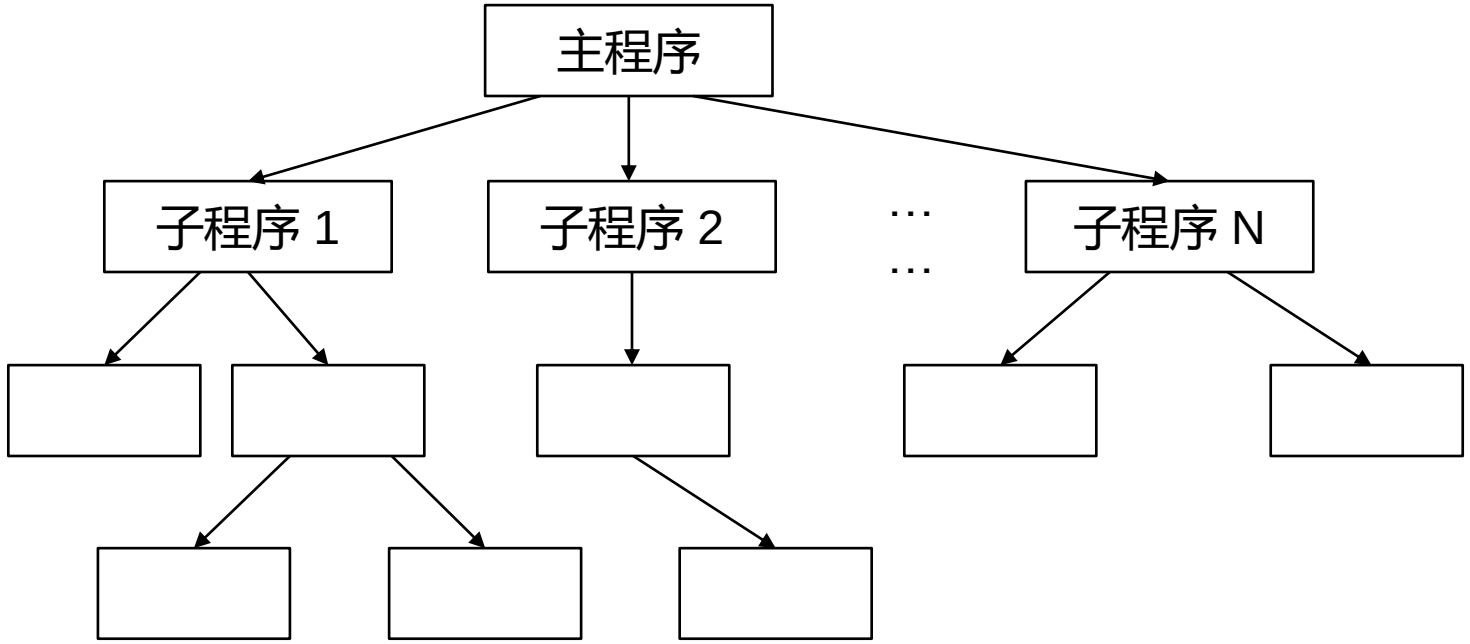
独立组件：通信过程，事件系统

虚拟机：解释器，基于规则的系统

Web应用系统：客户机-服务器，MVC，REST，GraphQL

主程序-子程序风格

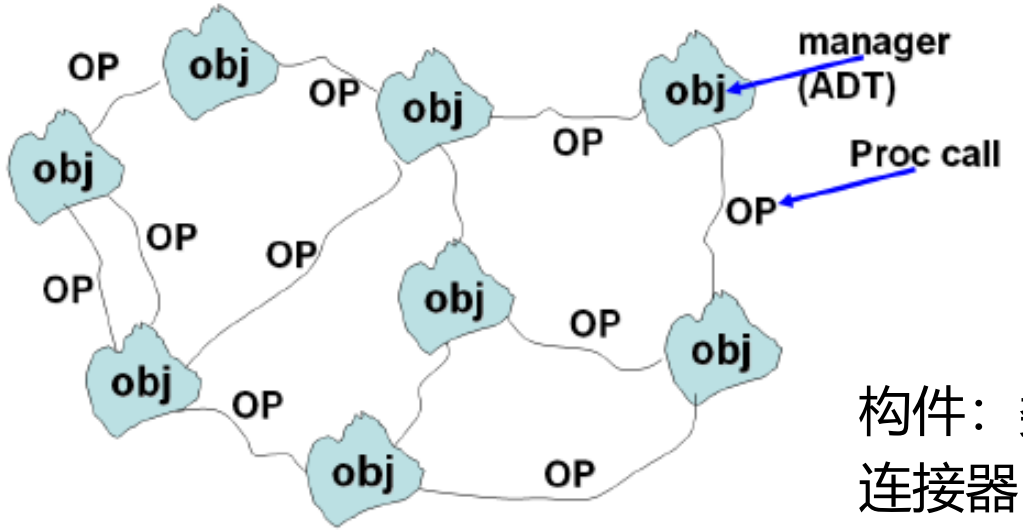
主程序-子程序风格是结构化程序设计的一种典型风格，从功能的观点设计系统，通过逐步分解和逐步细化，得到系统体系结构。



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

面向对象风格

- 系统被看作是对象的集合，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；
- 只通过接口与外界交互，内部的设计决策则被封装起来。

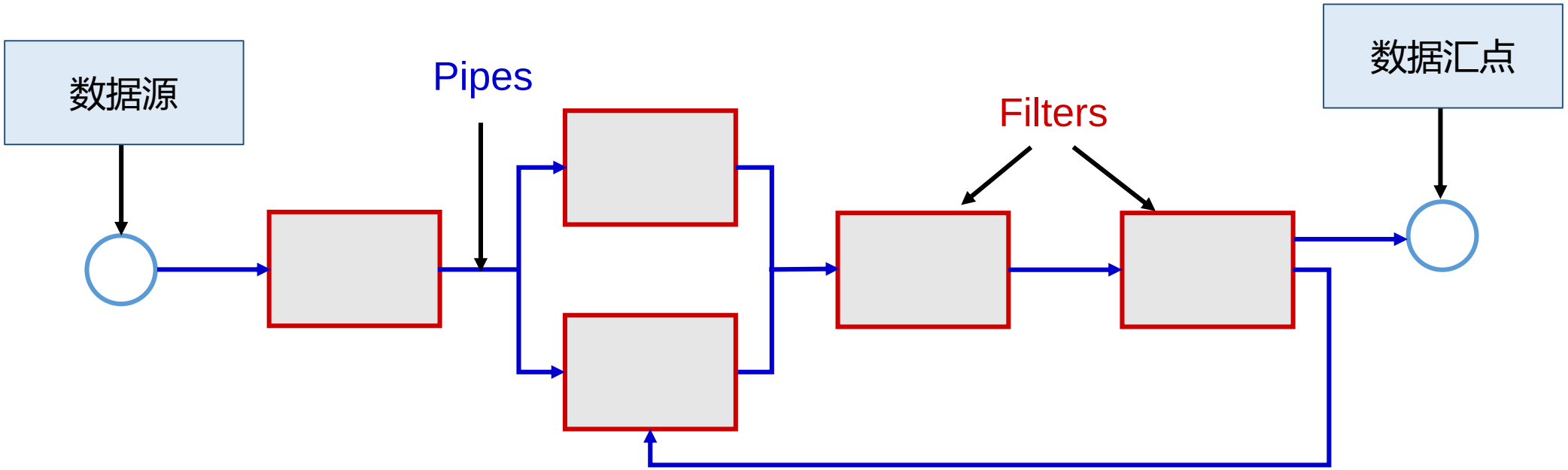


构件：类和对象

连接器：对象之间通过函数调用和消息传递实现交互

管道-过滤器风格

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。

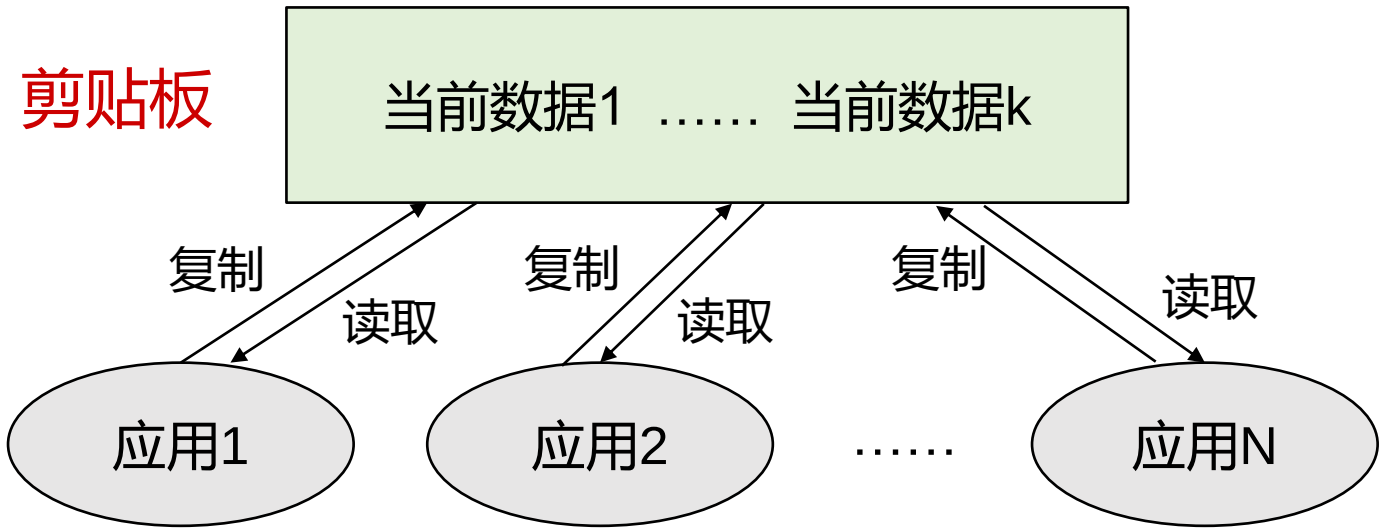


以数据为中心的风格

举例：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。

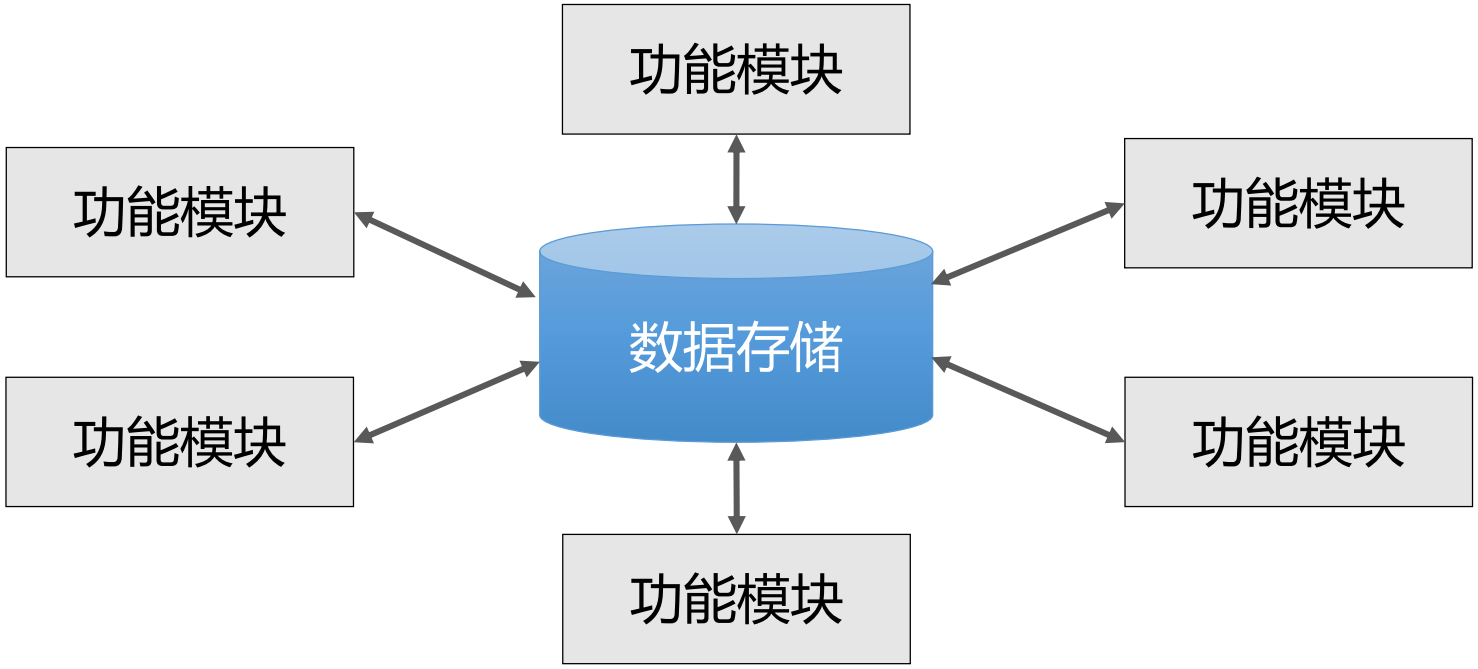
以数据为中心的风格

举例：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。

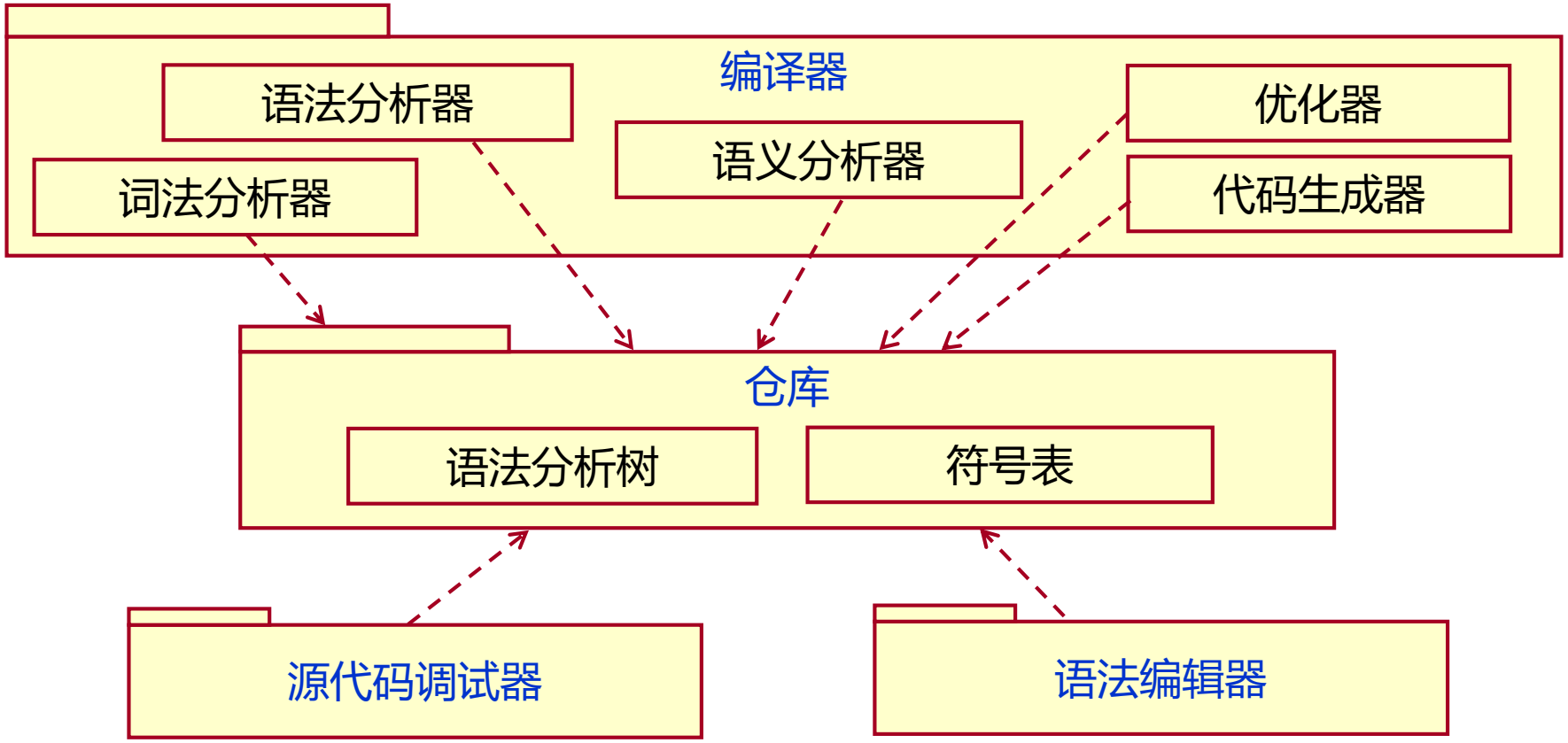


以数据为中心的风格

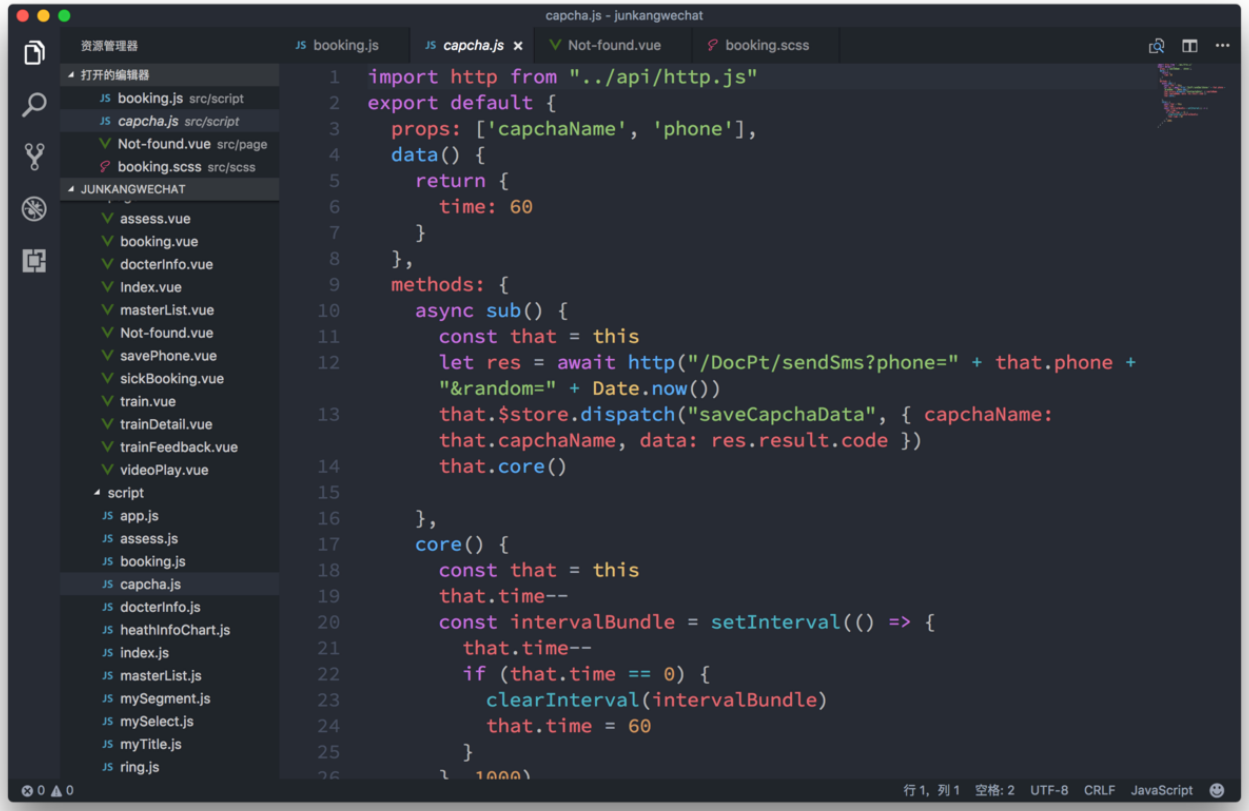
仓库体系结构（Repository Architecture）是一种以数据为中心的体系结构，适合于数据由一个子系统产生而由其他子系统使用的情形。



示例：程序设计语言编译器



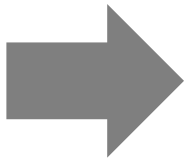
事件风格

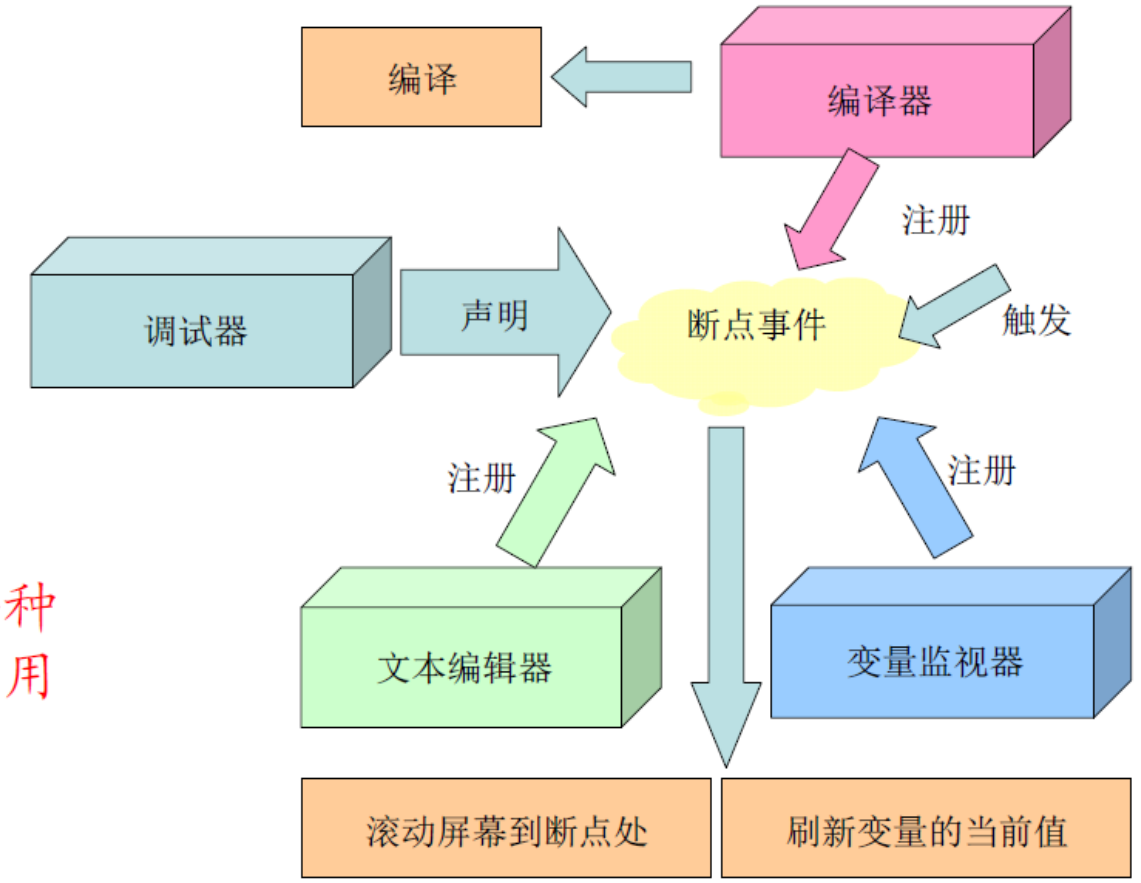


程序调试器的体系结构是什么类型的

事件风格

调试器的工作流程





这是一种
隐式调用

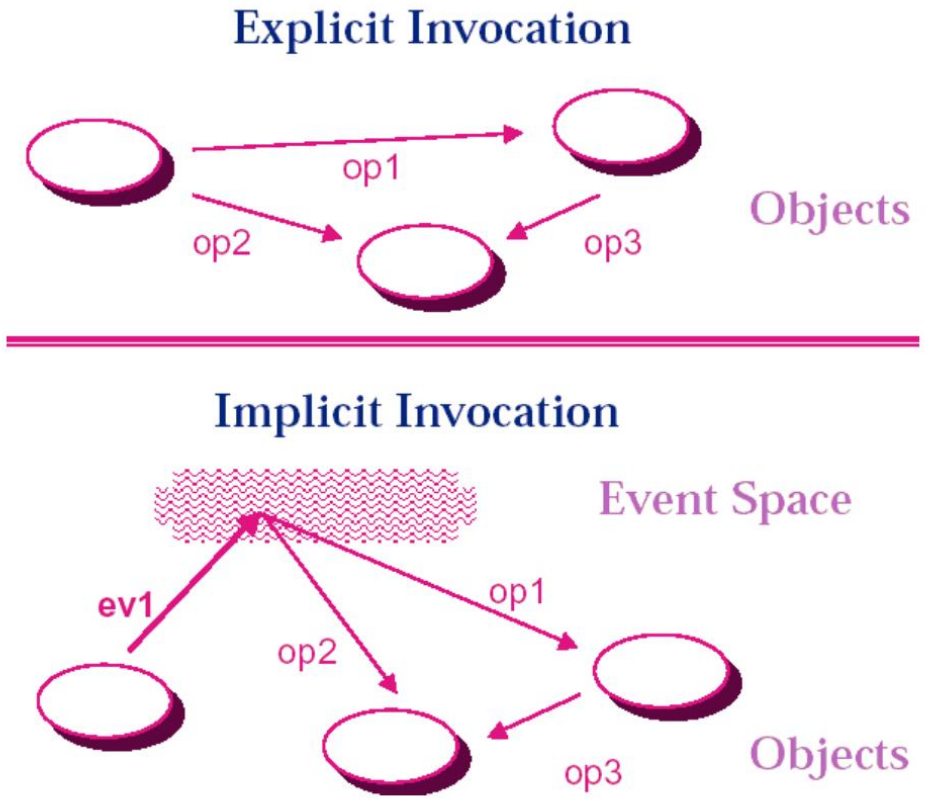
事件风格

显式调用：

- 各个构件之间的互动是由显性调用函数或程序完成的
- 调用过程与次序是固定的、预先设定的

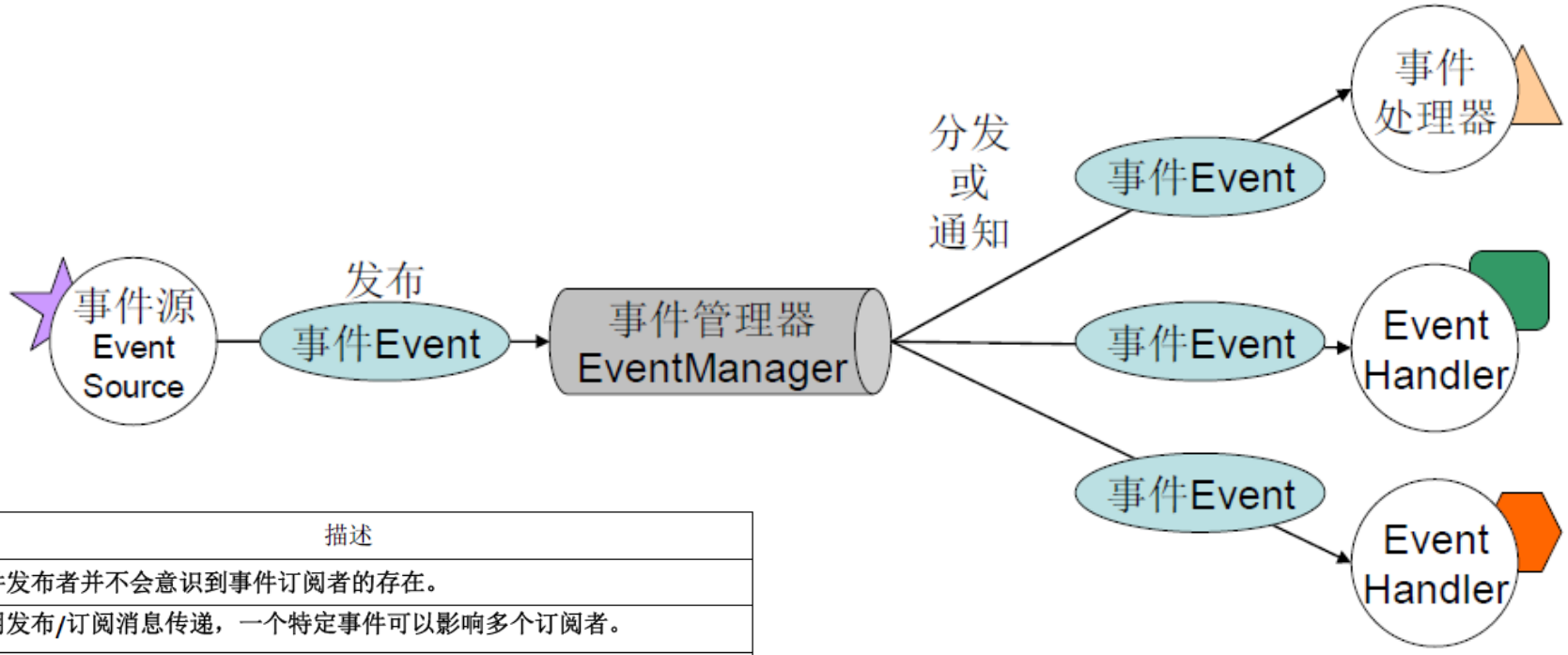
隐式调用：

- 调用过程与次序不是固定的、预先未知
- 各构件之间通过事件的方式进行交互



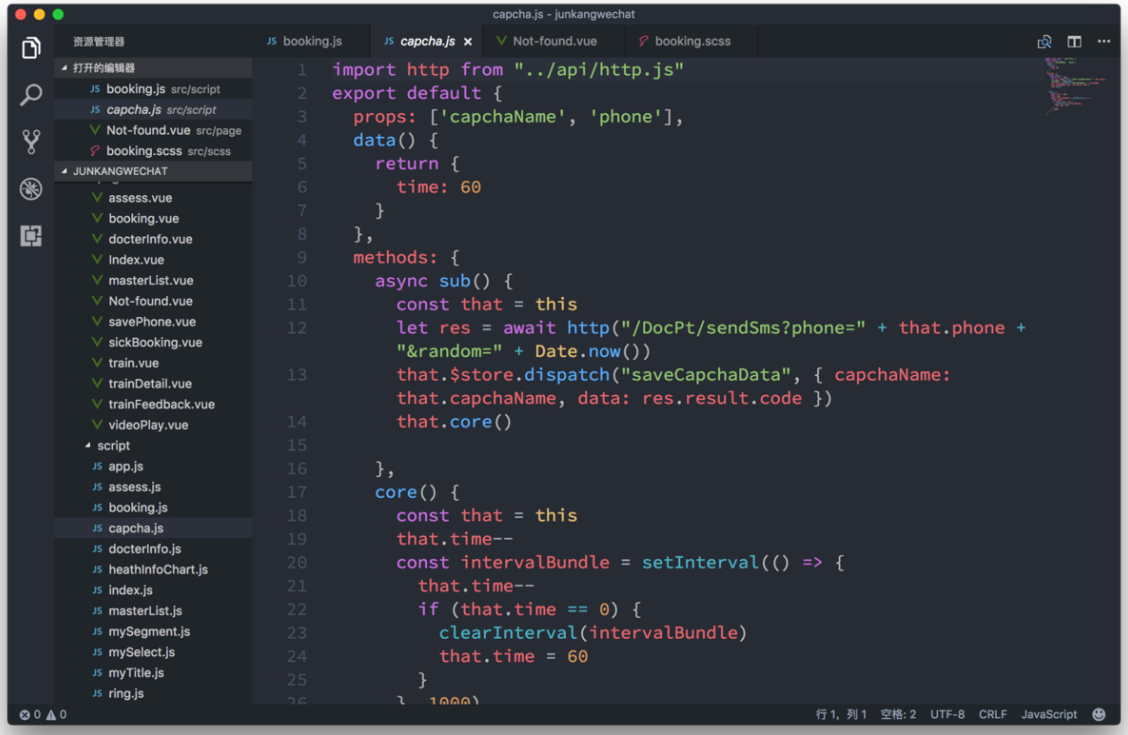
事件风格

事件系统是将应用看成是一个构件集合，每个构件直至发生对它有影响的事件时才有所动作。



功能	描述
分离的交互	事件发布者并不会意识到事件订阅者的存在。
多对多通信	采用发布/订阅消息传递，一个特定事件可以影响多个订阅者。
基于事件的触发器	控制流由接收者确定（基于发布的事件）。
异步	通过事件消息传递支持异步操作。

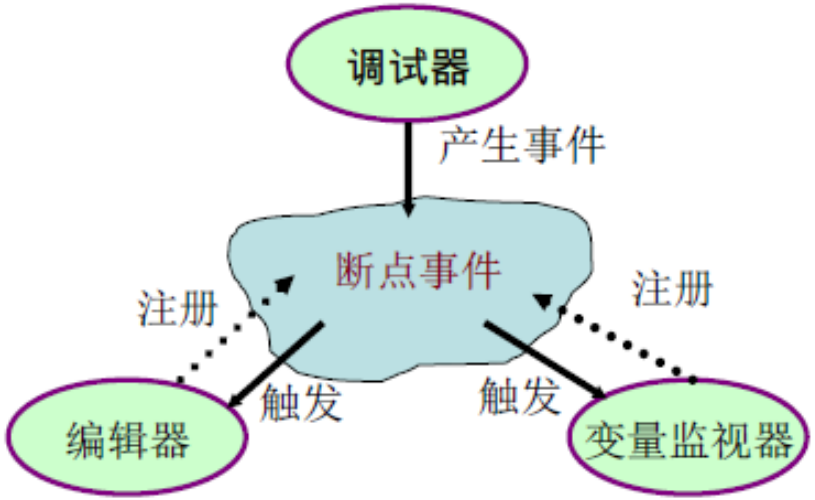
事件风格



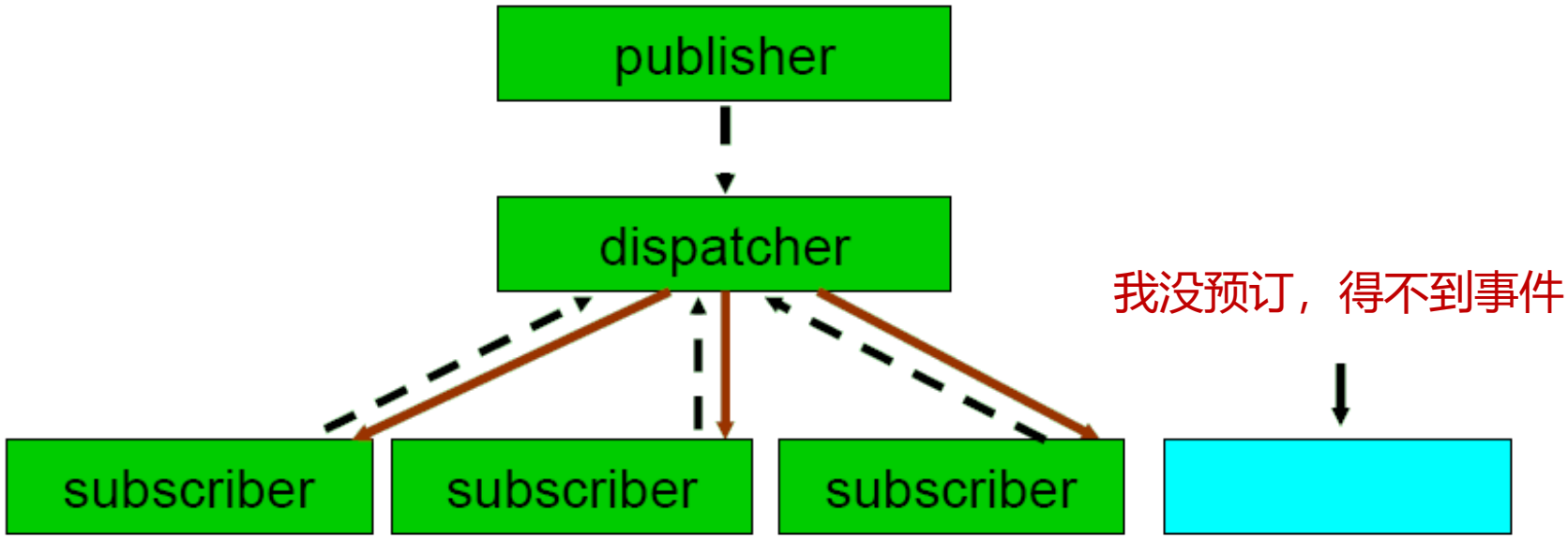
事件源：调试器

事件处理器：编辑器与变量监视器

事件管理器：IDE（集成开发环境）

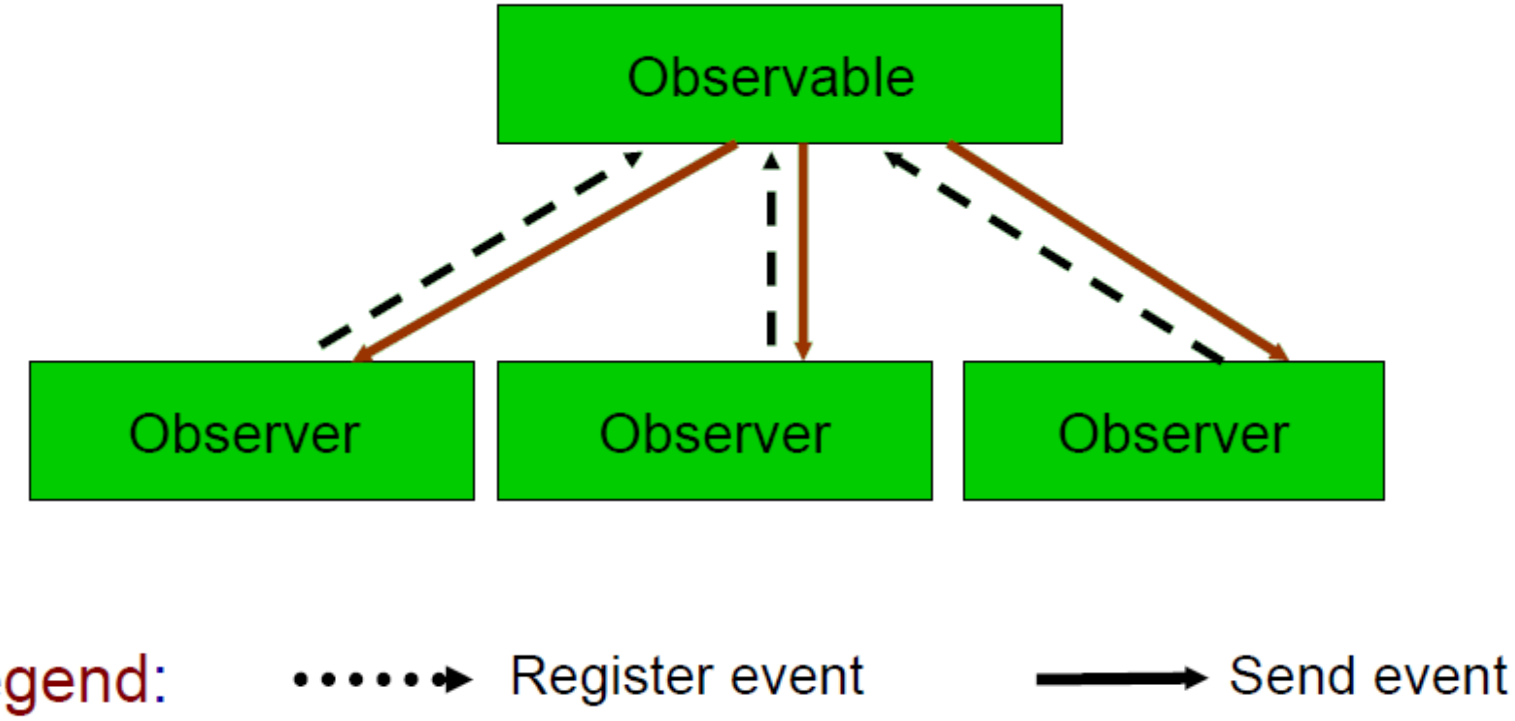


事件风格的实现策略之一：选择广播式



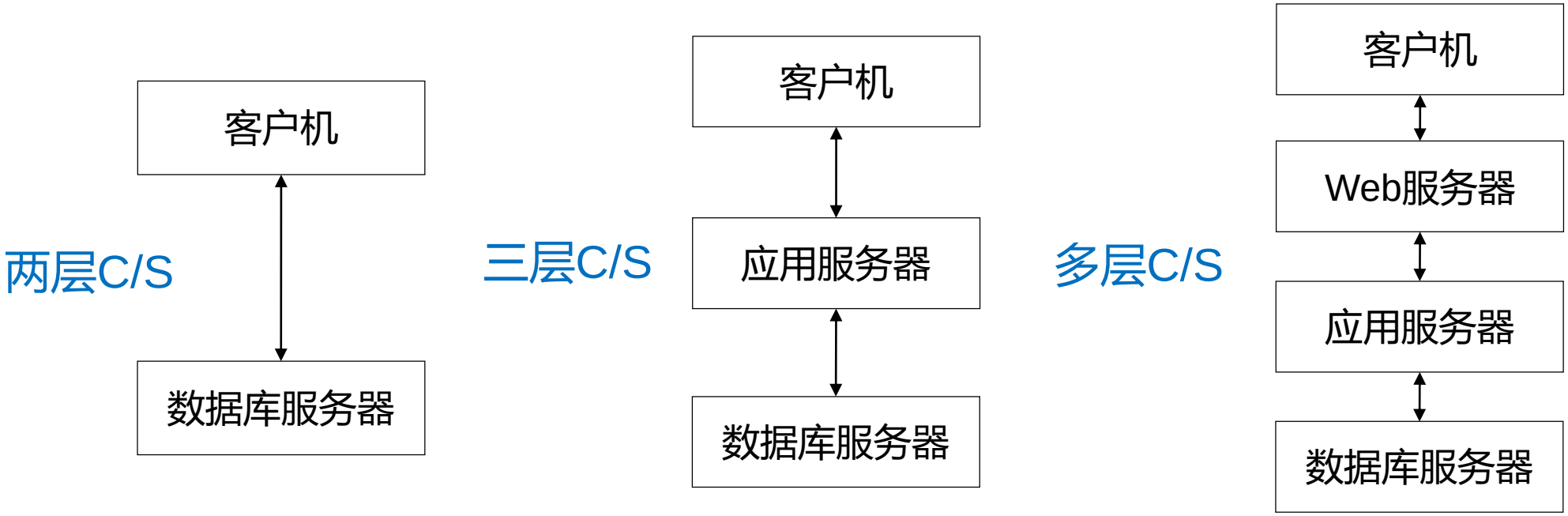
说明：这种方式是有目的广播，只发送给那些已经注册过的订阅者。

事件风格的实现策略之二：观察者模式



客户机 / 服务器结构

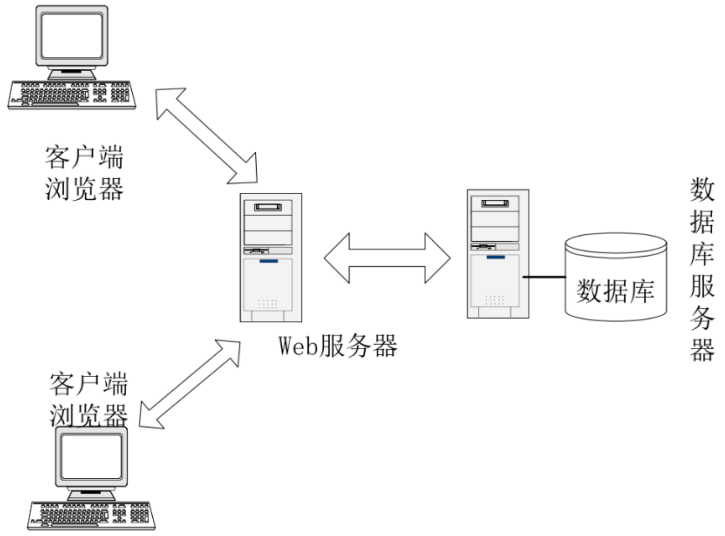
客户机 / 服务器体系结构（Client/Server）是一种分布式系统模型，作为服务器的子系统为其他客户机的子系统提供服务，作为客户机的子系统负责与用户的交互。

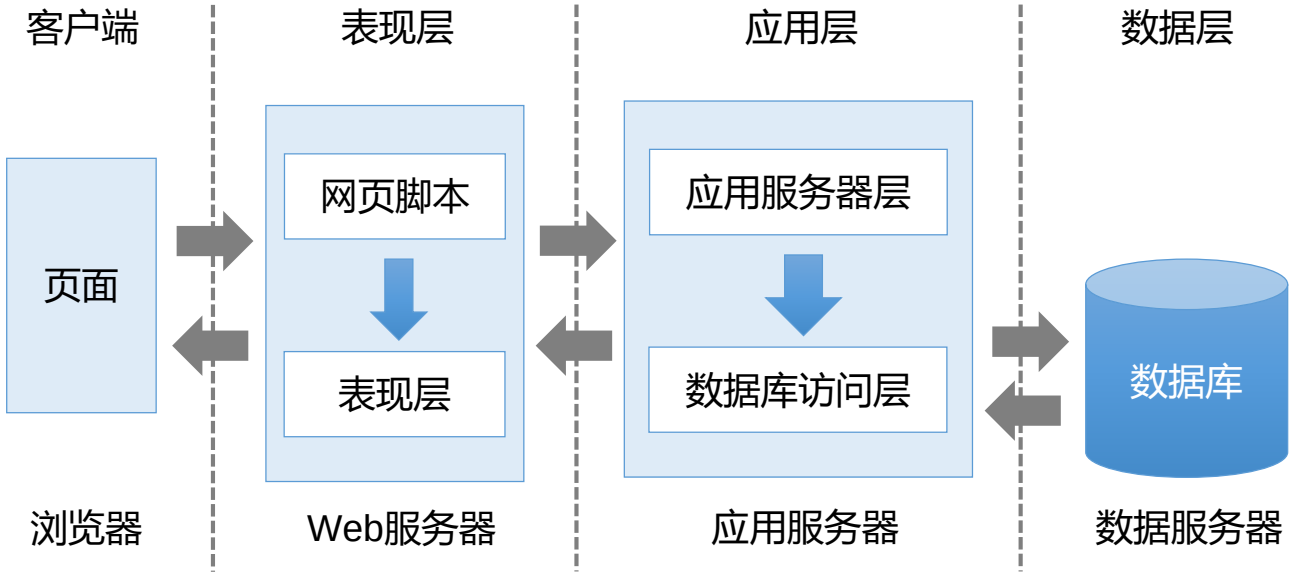


客户机 / 服务器结构

浏览器/服务器（Browser/Server）结构是三层C/S风格的一种实现方式

- 表现层：浏览器
- 业务逻辑层：Web服务器、应用服务器
- 数据层：数据库服务器

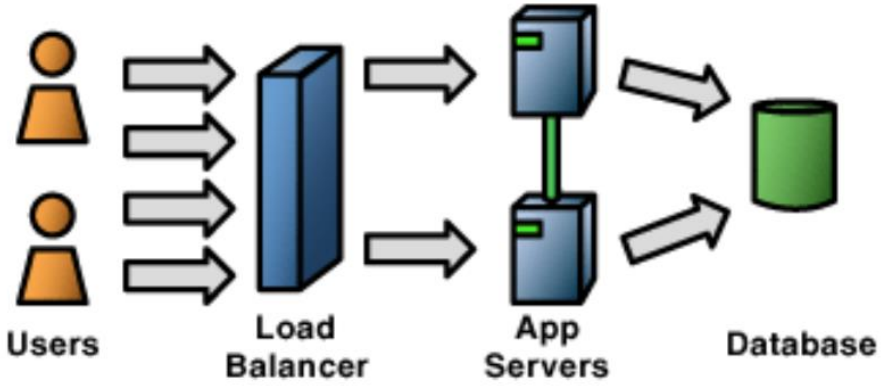




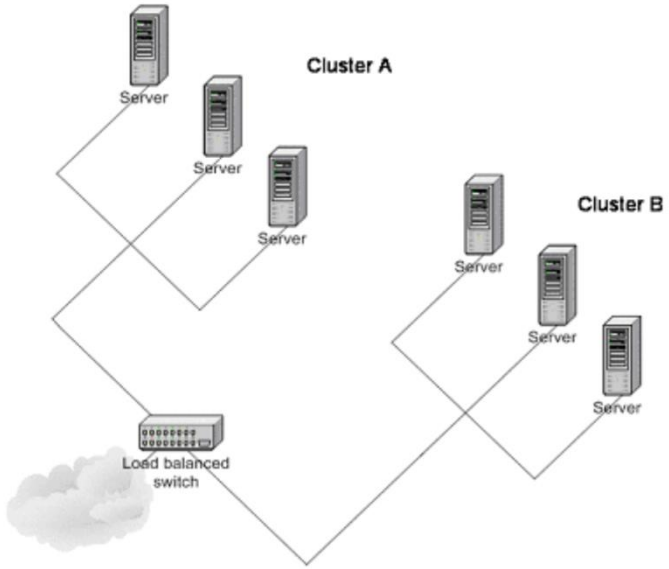
集群结构

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）是一组松散耦合的服务器，它们共同协作，可以被看作是一台服务器。

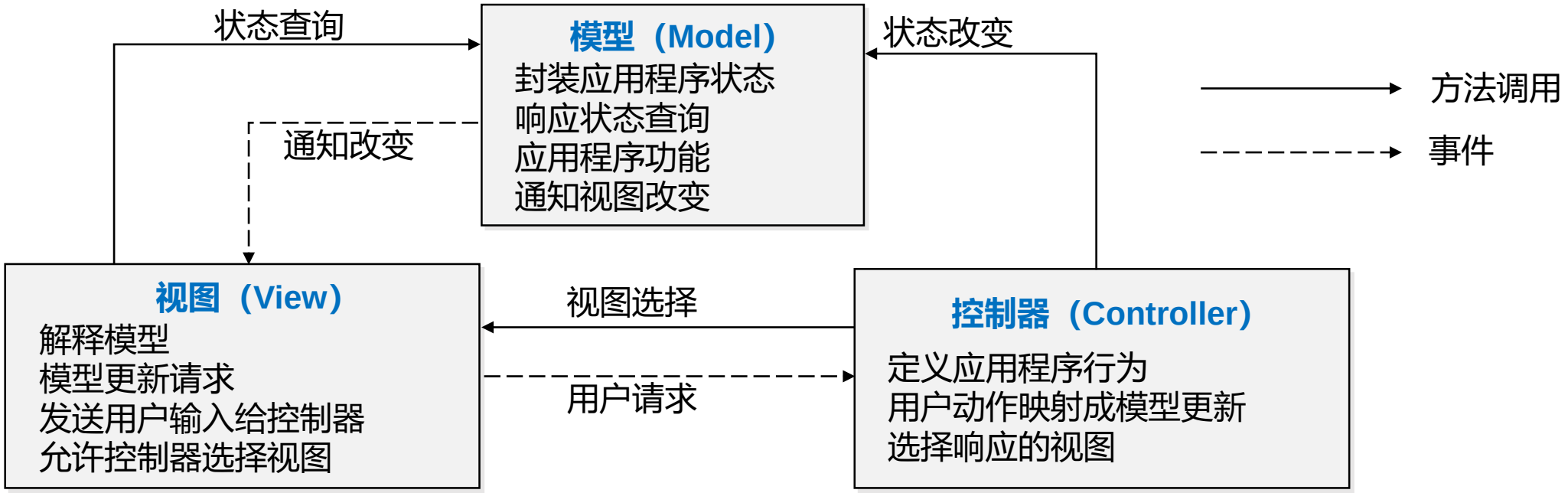


负载平衡是集群里的一个关键要素

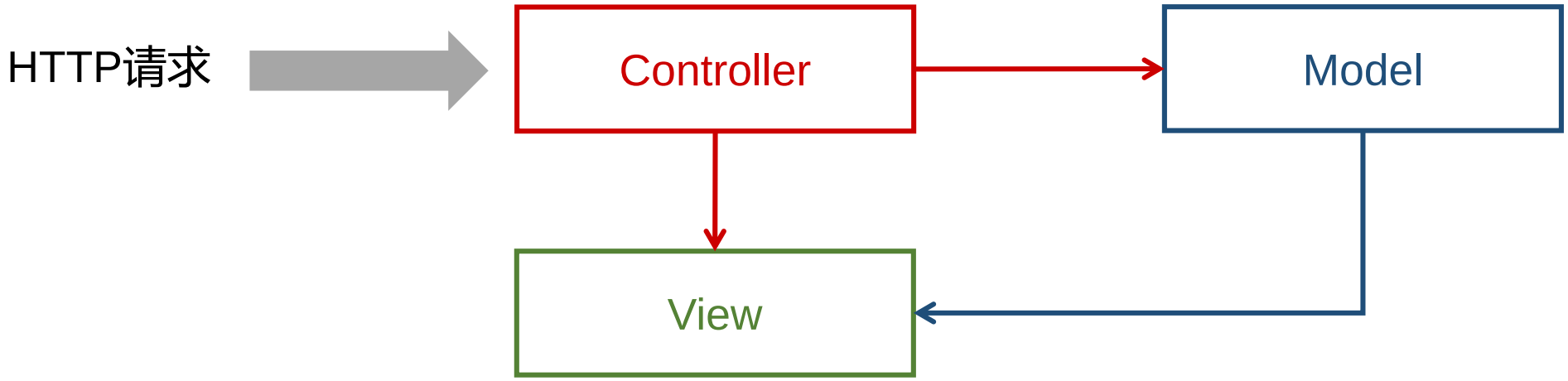


MVC结构

模型-视图-控制器（MVC）结构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。

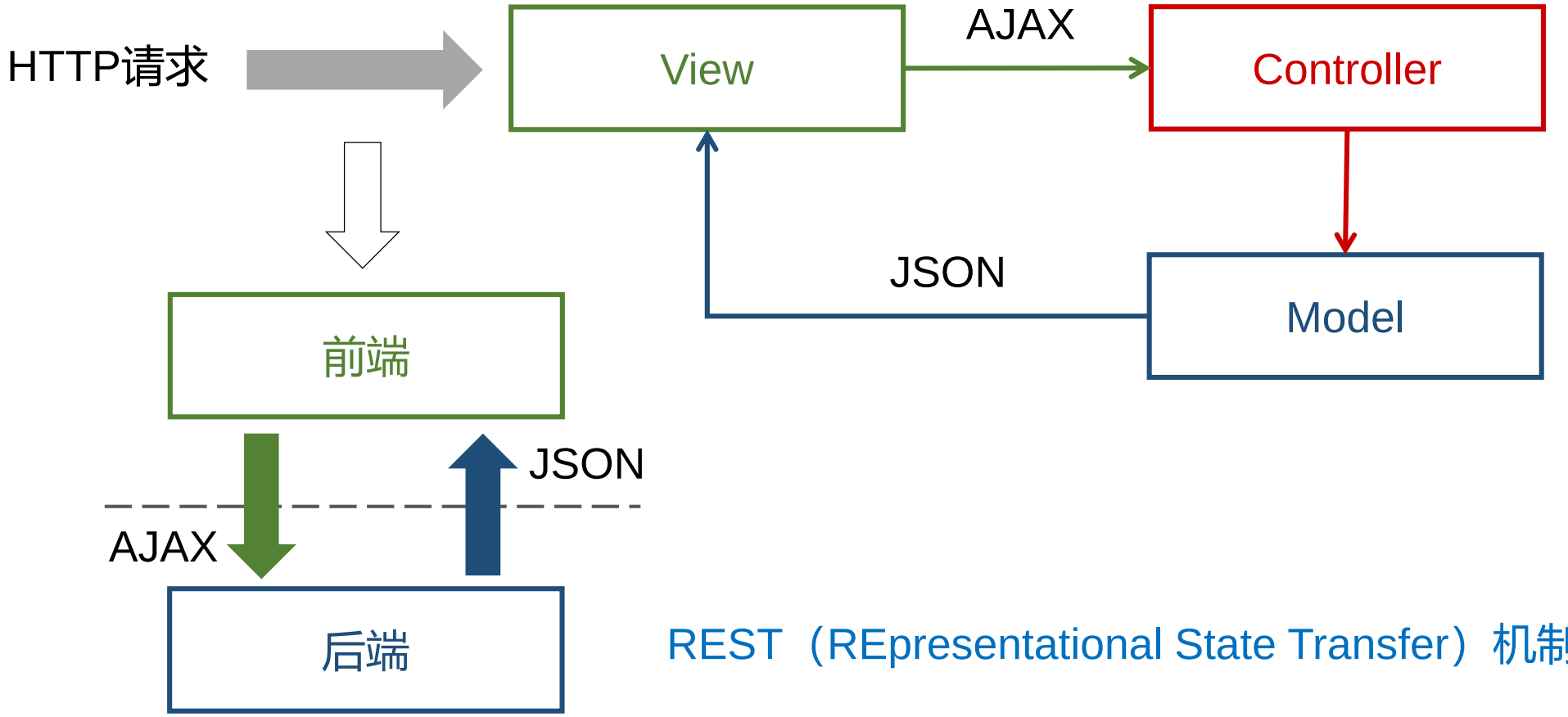


MVC结构



- 每次请求必须经过“控制器->模型->视图”过程，才能看到最终展现的界面
- 视图是依赖于模型的
- 渲染视图在服务端完成，呈现给浏览器的是带有模型的视图页面，性能难优化

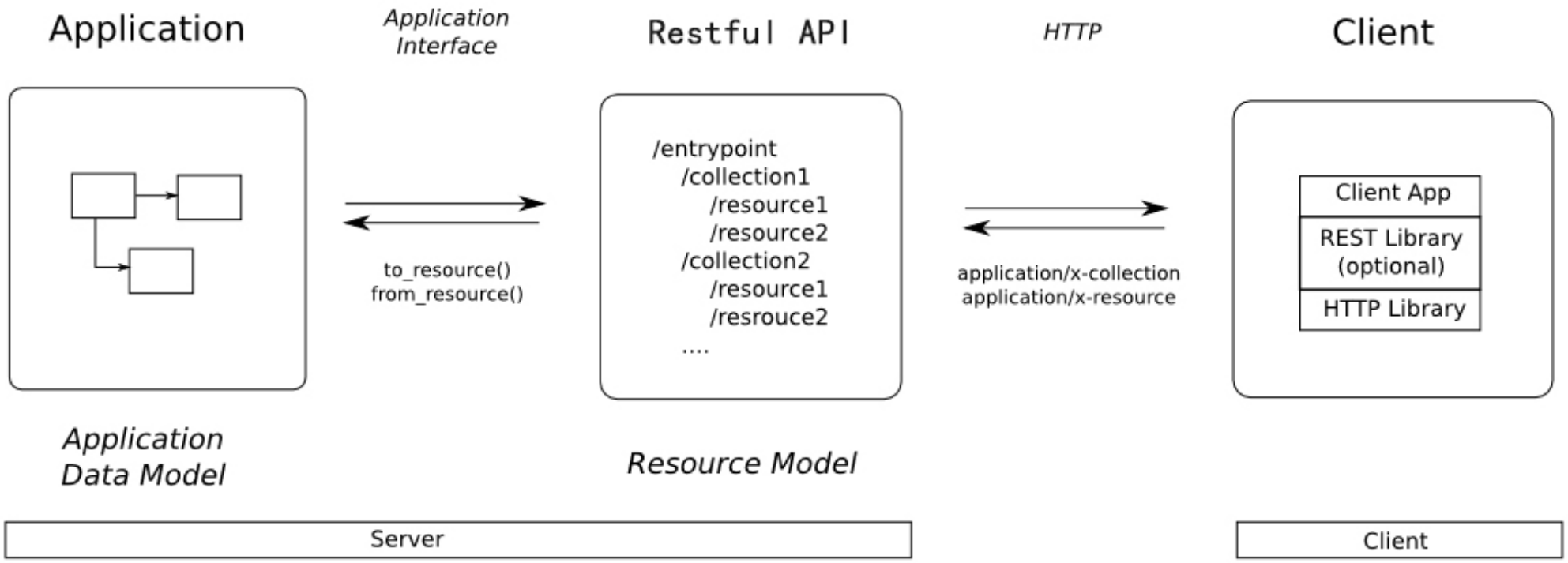
MVC结构



REST（REpresentational State Transfer）机制

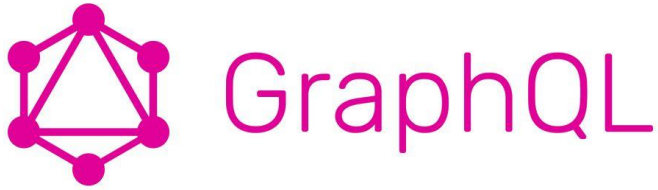
RESTful 架构

RESTful 是一种以资源为核心的分布式应用架构风格



GraphQL 架构

GraphQL用于替换RESTful API，由Facebook开发，服务端可以用任何语言实现。它是一种API查询语言，即开发者定义数据的类型系统在服务器端的运行时。



用户信息对应的数据模型是固定的，每次请求其实是对这些数据做了过滤和筛选。对应到数据库操作，就是数据的查询操作。

如果客户端也能够像“查询”一样发送请求，就可以从后端接口这个“大数据库”去过滤筛选业务需要的数据。

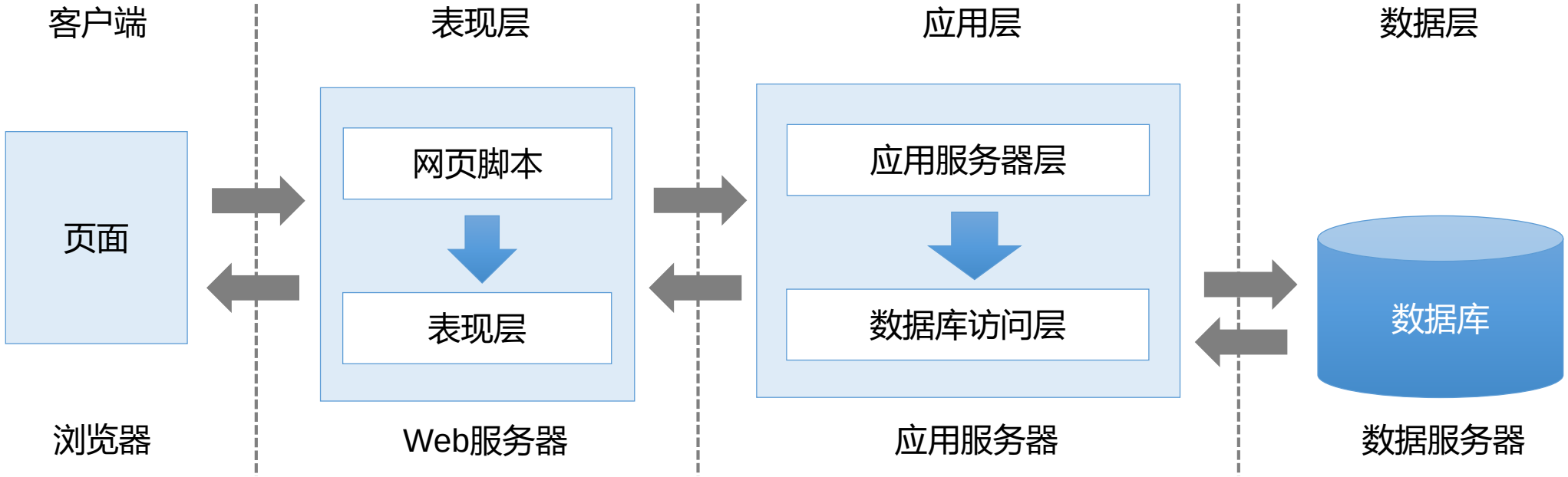
无服务器架构

无服务器架构（Serverless）是一种依赖第三方服务（Backend as a Service，BaaS）的应用程序设计方式，或者把代码交由托管临时容器中执行（Function as a Service，FaaS）。

- BaaS是由服务商为开发者提供整合云后端的服务，如提供文件存储、数据存储、推送服务、身份验证服务等功能，以帮助开发者快速开发应用。
- FaaS提供了一个平台，允许开发人员能够响应事件执行代码，而不需要构建和维护复杂的基础架构，由第三方应用程序或服务来管理服务器端的逻辑和状态。
- 2014年亚马逊AWS推出了Lambda成为最主要的无服务架构的代表，它允许用户仅仅上传代码而无需提供和管理服务器，由它负责代码的执行、高可用扩展，支持从别的AWS服务或其他Web应用直接调用等。

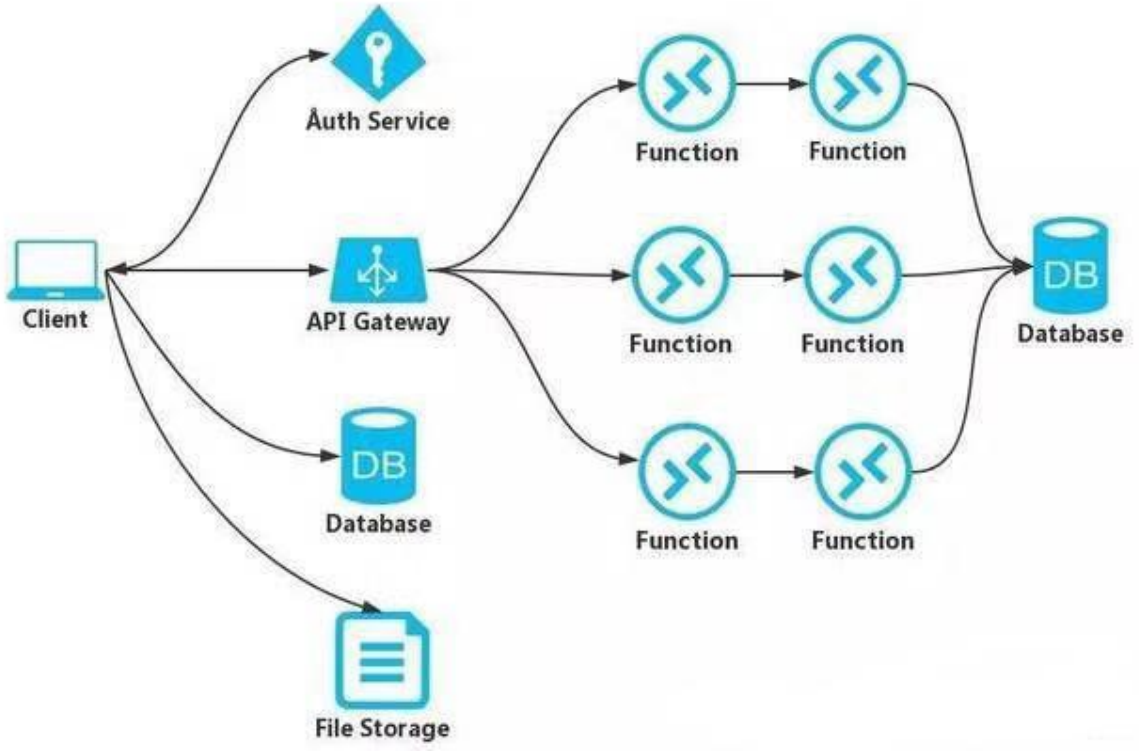
无服务器架构

在经典的Web应用中，服务器后端系统实现了大部分业务逻辑：认证、搜索、事务等。



无服务器架构

在FaaS架构中，认证、数据库等将采用第三方的服务，从原来的单体后端里分拆出来（可能需要在原来的客户端里加入一些业务逻辑）。对于大部分的任务，通过函数的形式进行执行，而不再使用一直在线的服务器进行支持。



无服务器架构

- FaaS里的应用逻辑单元都可以看作是一个函数，开发人员只关注如何实现这些逻辑，而不用提前考虑性能优化，让工作聚焦在这个函数里，而非应用整体。
- FaaS是无状态的，无状态意味着本地内存、磁盘里的数据无法被后续的操作所使用。大部分的状态需要依赖于外部存储，比如数据库、网络存储等。
- FaaS的函数应当可以快速启动执行，并拥有短暂的生命周期。函数在有限的时间里启动并处理任务，并在返回执行结果后终止。如果它执行时间超过了某个阈值，也应该被终止。
- FaaS需要借助于API Gateway将请求的路由和对应的处理函数进行映射，并将响应结果代理返回给调用方。
- FaaS架构消除了对传统的始终在线服务器的大部分需求，可以显著降低运维成本、复杂性以及减少项目的上线准备时间，代价是增加了对供应商和相对不成熟支持服务的依赖。

谢谢大家！

THANKS