2 软件体系结构核心模型

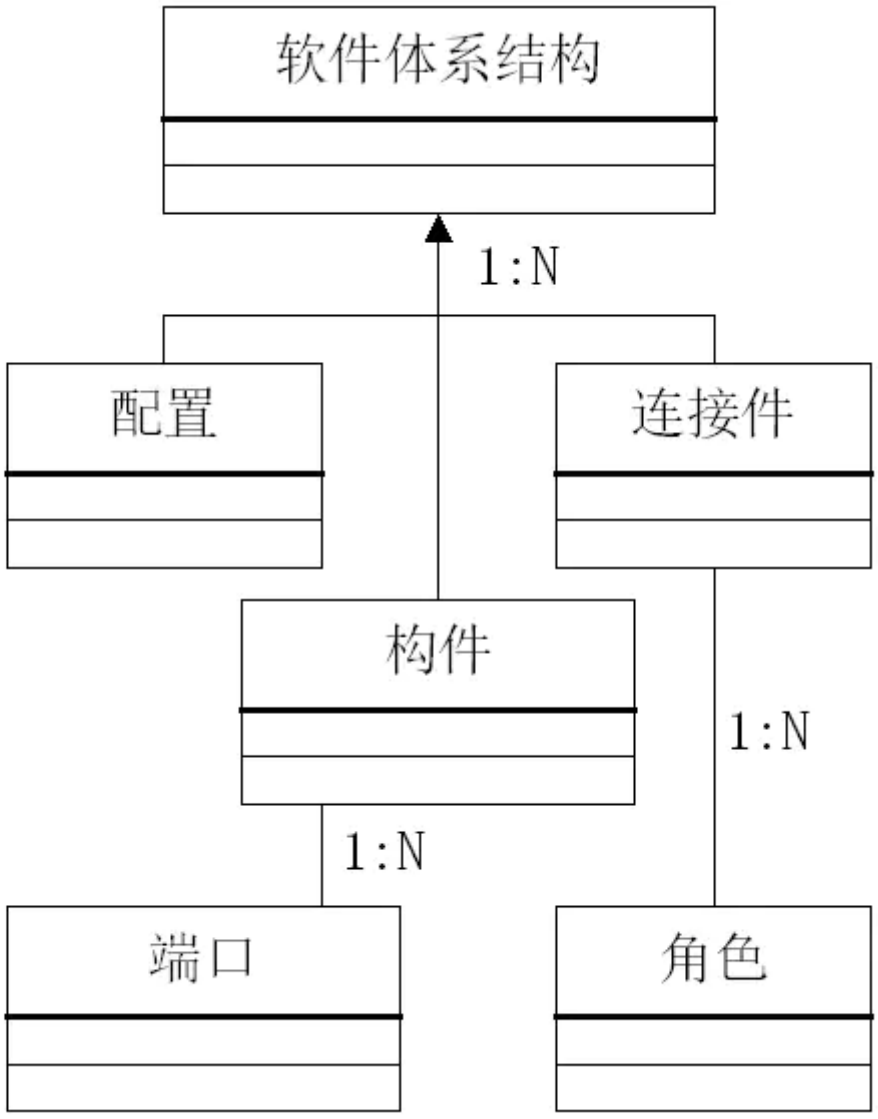
2.1 核心模型的最基本元素

> 体系结构 = 组件 + 连接件 + 约束

- 组件 - component
- 连接件 - connector
- 约束 - constrain

2.2 核心模型示意图

- 配置 - configuration
- 连接件 - connector
- 构件 - component
- 端口 - port
- 角色 - role

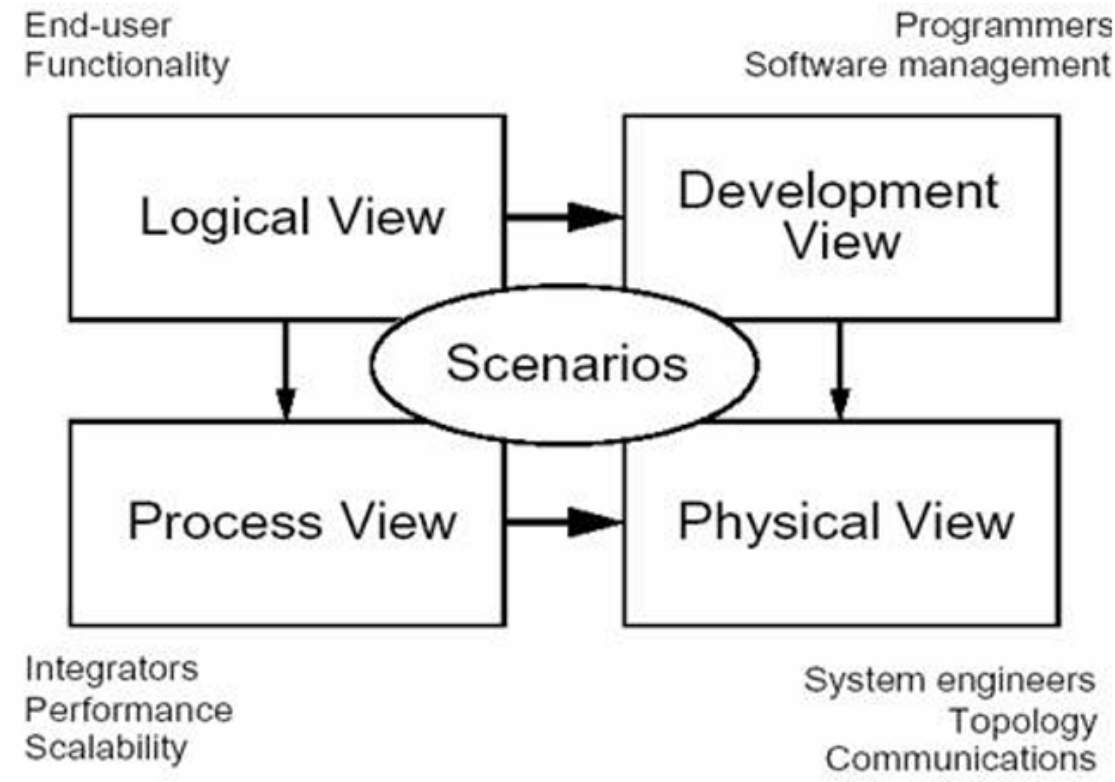


3 An Architectural Mode（4+1视图）[重点]

> 参考：另一份md文档

3.1 概述

- `logical view`（唯一，关注功能性需求）
- `process view`（性能和扩展性需求）
- `physical view`（硬件拓扑逻辑）
- `development view`（代码相关）
- `use case / scenarios`（用例/场景）



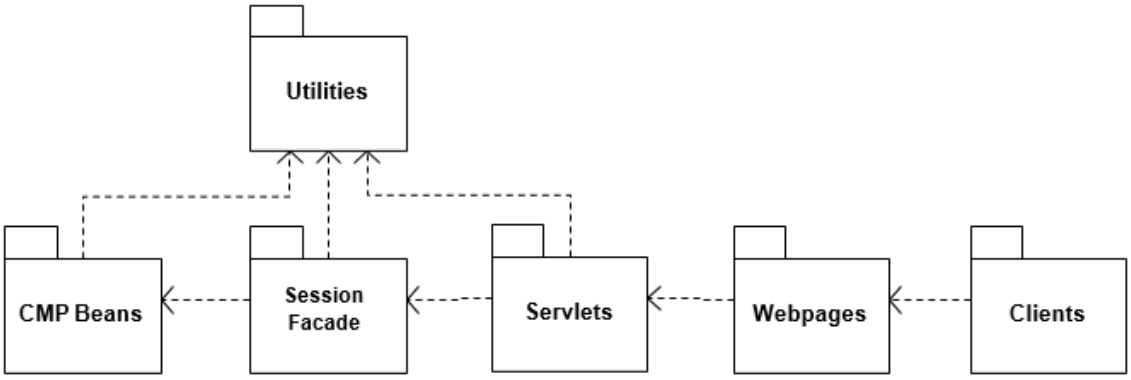
3.2 必背表格

	logic	process	development	physical	user case
视图	逻辑视图	过程视图	开发视图	物理视图	场景
组件	类	任务	模块、子系统	节点	步骤、脚本
连接工具	关联、继承、约束	会面、消息、广播、RPC 等	编译依赖性、“with”语句、“include”	通信媒体、LAN、WAN、总线等	
容器	类的种类	过程	子系统（库）	物理子系统	Web
涉众	最终用户	系统设计人员、集成人员	开发人员、经理	系统设计人员	最终用户、开发人员
关注点	功能	性能、可用性、S/W 容错、整体性	组织、可重用性、可移植性、产品线	可伸缩性、性能、可用性	可理解性

3.3 Logic view - 逻辑视图 - functional requirement

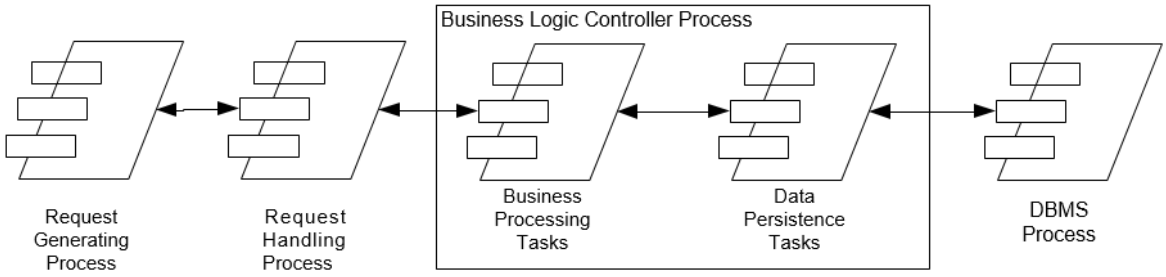
- 功能性需求

- 通常用UML中`类图`和`类模板`来展示逻辑架构
- UML的内容看《设计模式》



3.4 Process view - 处理视图 - nonfunctional requirement

- 非功能性需求
- 把软件分为一系列独立的任务，分为`主要任务`和`次要任务`，方框中为major tasks
- 对分布式系统，大部分时间花在通讯上（60%以上）



3.5 Development view - 开发视图

- 实际的软件模块组织
- 偏向于non-runtime的属性，如系统升级，系统管理
- 系统架构分为大对象和小对象

3.6 Physical view - 物理视图

- 在UML中被称为Depolyment view 部署视图

4 经典架构风格

4.1 pipe and filters - 管道与过滤器

> 该架构专门用来`处理数据流`，属于`数据流模式`

管道-过滤器模式的体系结构是**面向数据流**的软件体系结构

它最典型的应用是在编译系统。一个普通的编译系统包括词法分析器,语法分析器,语义分析与中间代码生成器,优化器,目标代码生成器等一系列对源程序进行处理的过程。人们可以`将编译系统看作一系列过滤器的连接体,按照管道-过滤器的体系结构进行设计`

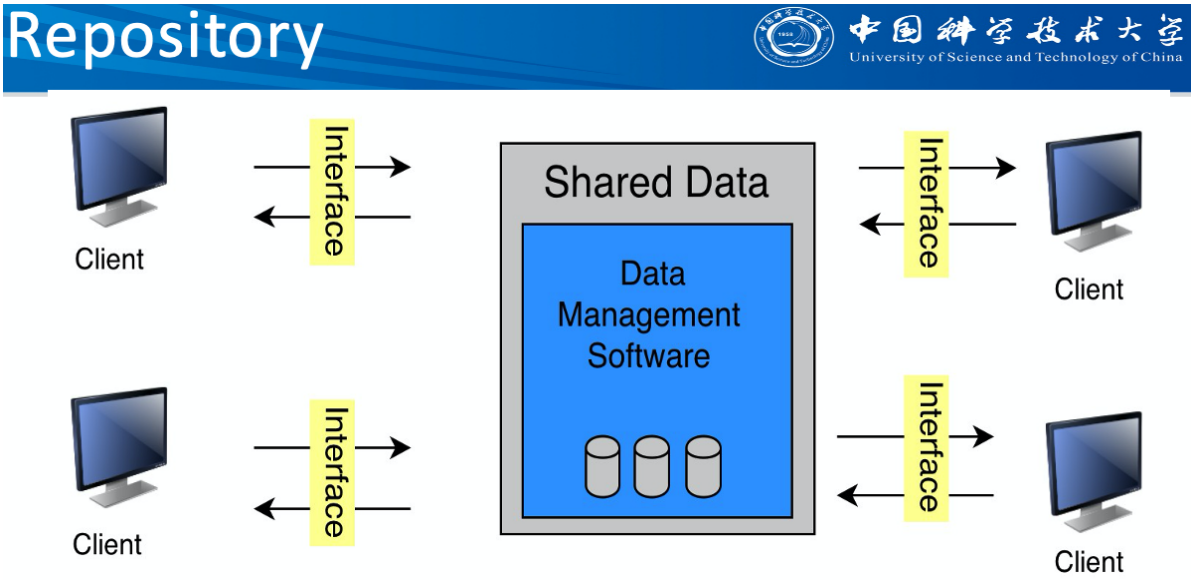
在管道-过滤器架构模式中，每个构件都有一组输入，输出，构件读取输入的数据流，经过内部处理后，产生输出数据流，该过程主要完成输入流的变换及增量计算。通常，将`这里的构件称为过滤器`，其中的`连接器就像是数据流传输的管道`，将一个过滤器的输出传送到另一过滤器的输入。管道，过滤器输出的正确性并不依赖于过滤器进行增量计算过程的顺序。

4.2 data-centered 数据中心架构

关注点：数据安全和完整性

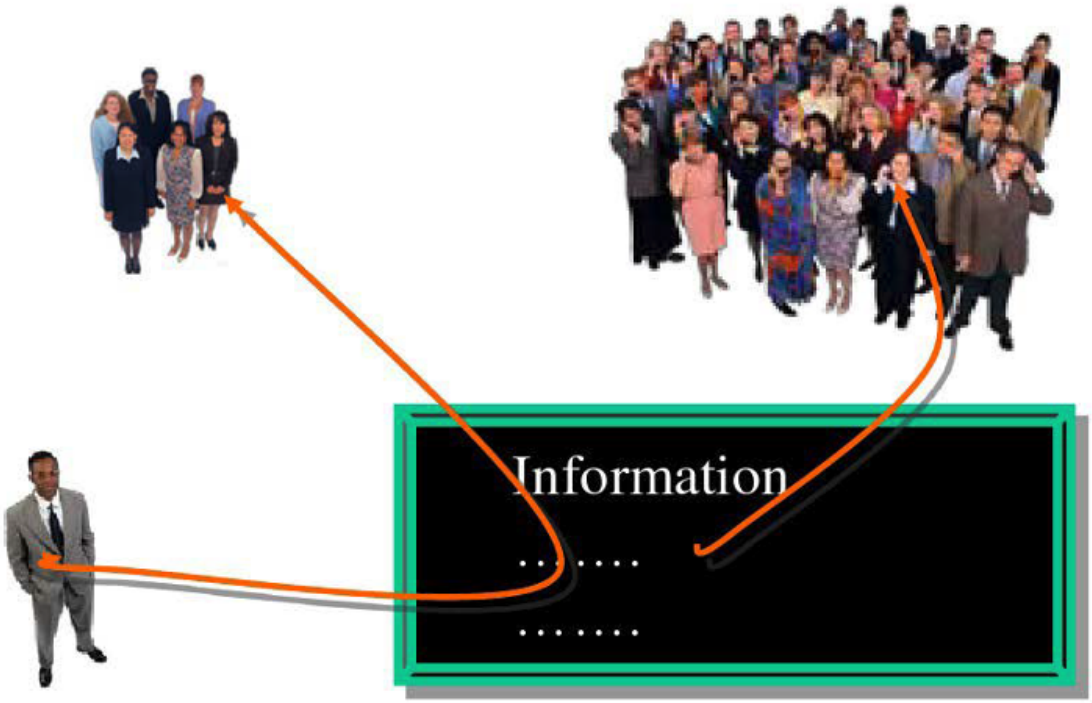
4.2.1 repository style 仓库结构风格

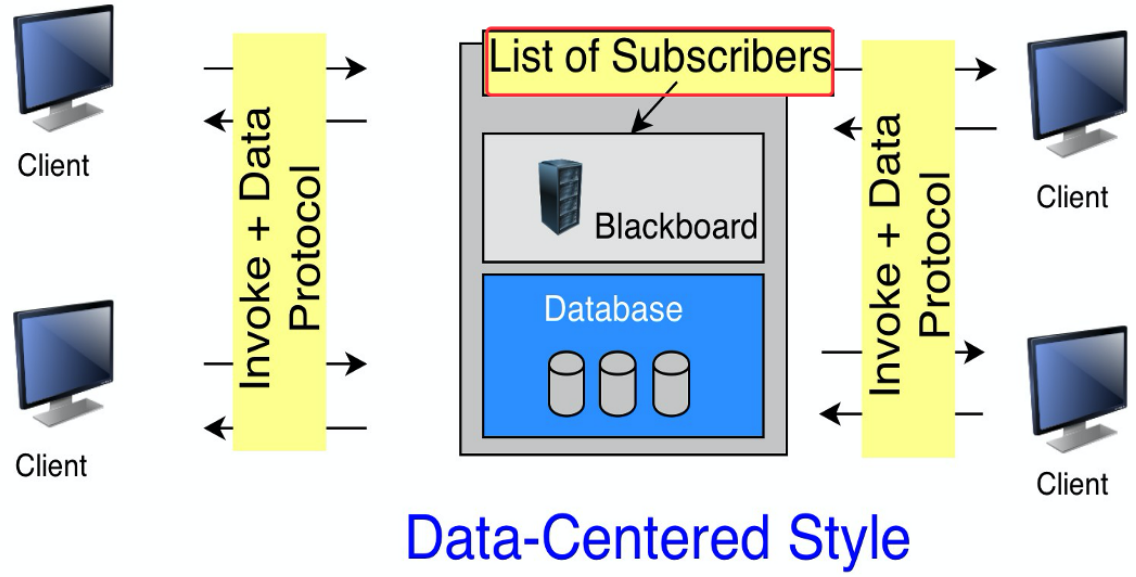
- `被动`响应请求
- `短连接`，通讯完断开



4.2.2 blackboard style 黑板结构风格

- `主动`响应请求
- `长连接`：一直保持连接，成本高，可扩展性不好，服务器不友好





4.2.3 web architecture 网络架构

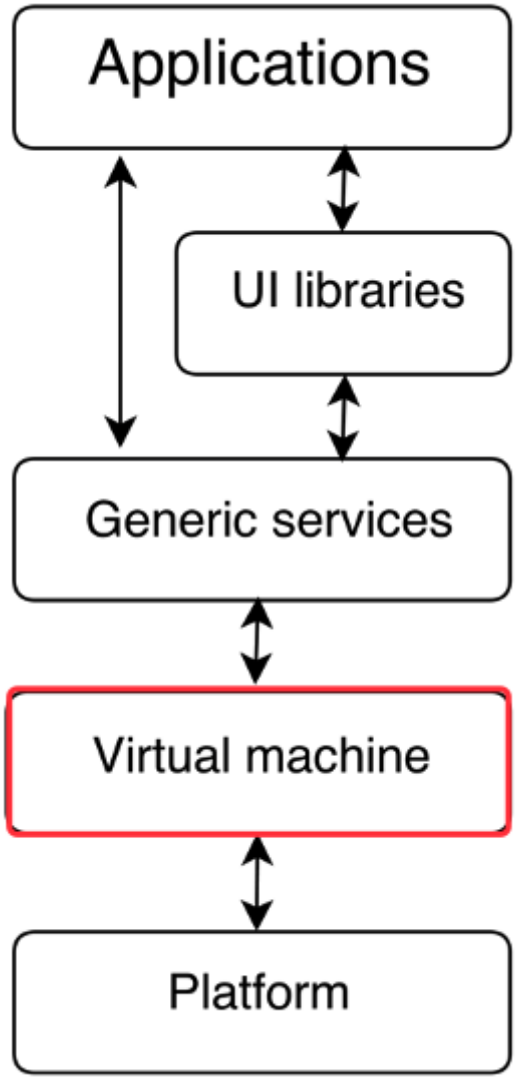
- flash：定时刷新，大多可能是无用功
- 解决web不主动响应问题，可用web socket主动通知客户端
- 微软soler 云数据库，广域网可访问（一般数据库只能局域网）

4.3 layer architectures 分层架构

- 划分为**职责清晰**的层次
- 分层：层与层之间有联系，**系统关系更简单**，**通讯只发生在层之间**not always but normally(跨层调用：无线网，I/O调用，上层通知下层结束)
- 逻辑架构一般分层，一般来说，presentation，application，data storage三层分开

4.3.1 virtual machine 虚拟机

- 一次编写，到处运行
- 虚拟机被放在了一个层次里，其它层次可以变化



4.4 notification architecture 通知架构

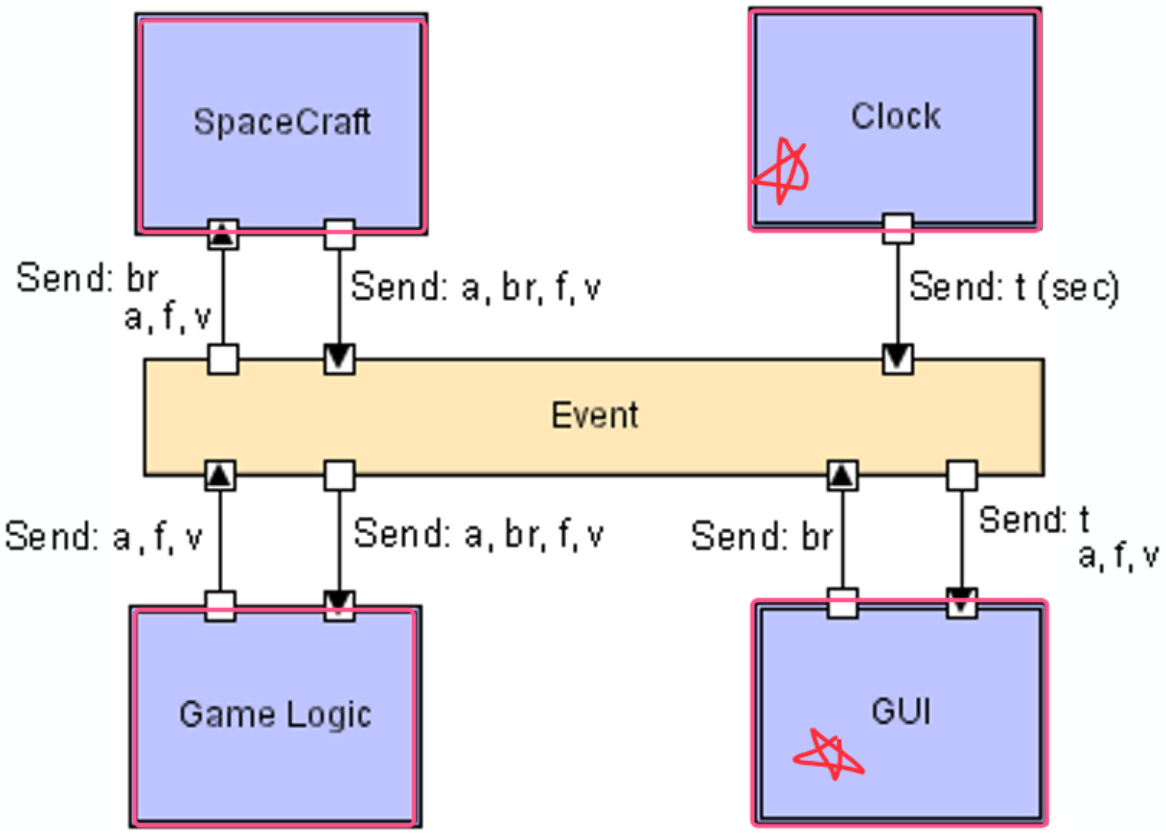
- notification 架构（严格来说是设计）
- 异步通讯，低耦合，用于`大规模系统`，分布式节点多

4.4.1 event-based style 事件驱动架构

事件驱动架构由事件发起者和事件使用者组成。事件的发起者会检测或感知事件，并以消息的形式来表示事件。它并不知道事件的使用者或事件引起的结果。

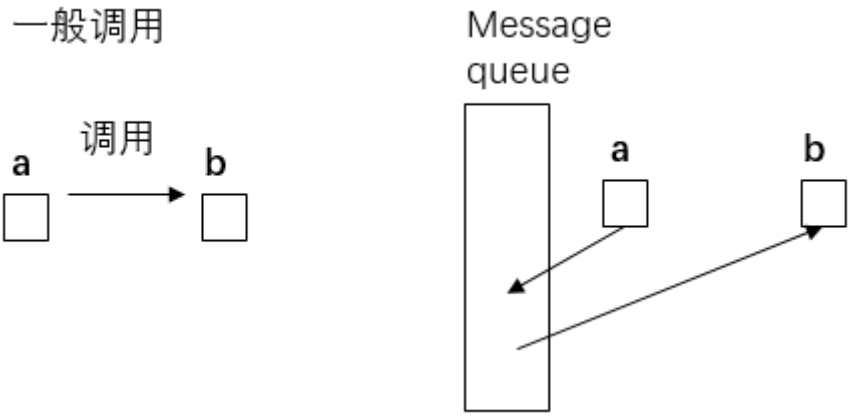
检测到事件后，系统会通过事件通道从事件发起者传输给事件使用者，而事件处理平台则会在该通道中以异步方式处理事件。事件发生时，需要通知事件使用者。他们可能会处理事件，也可能只是受事件的影响。

事件处理平台将对事件做出正确响应，并将活动下发给相应的事件使用者。通过这种下发活动，我们就可以看到事件的结果。



4.4.2 message queue 消息队列

- 异步，不存在阻塞
- 便于管理，把多—多关系 ——> 多— 1 —多（云计算，用户多，多用异步调用）
- **异步会导致可测试性下降**，不知道事件什么时候来
- service bus总线结构



4.5 network-centered style 网络中心架构

4.5.1 Client-Server Style

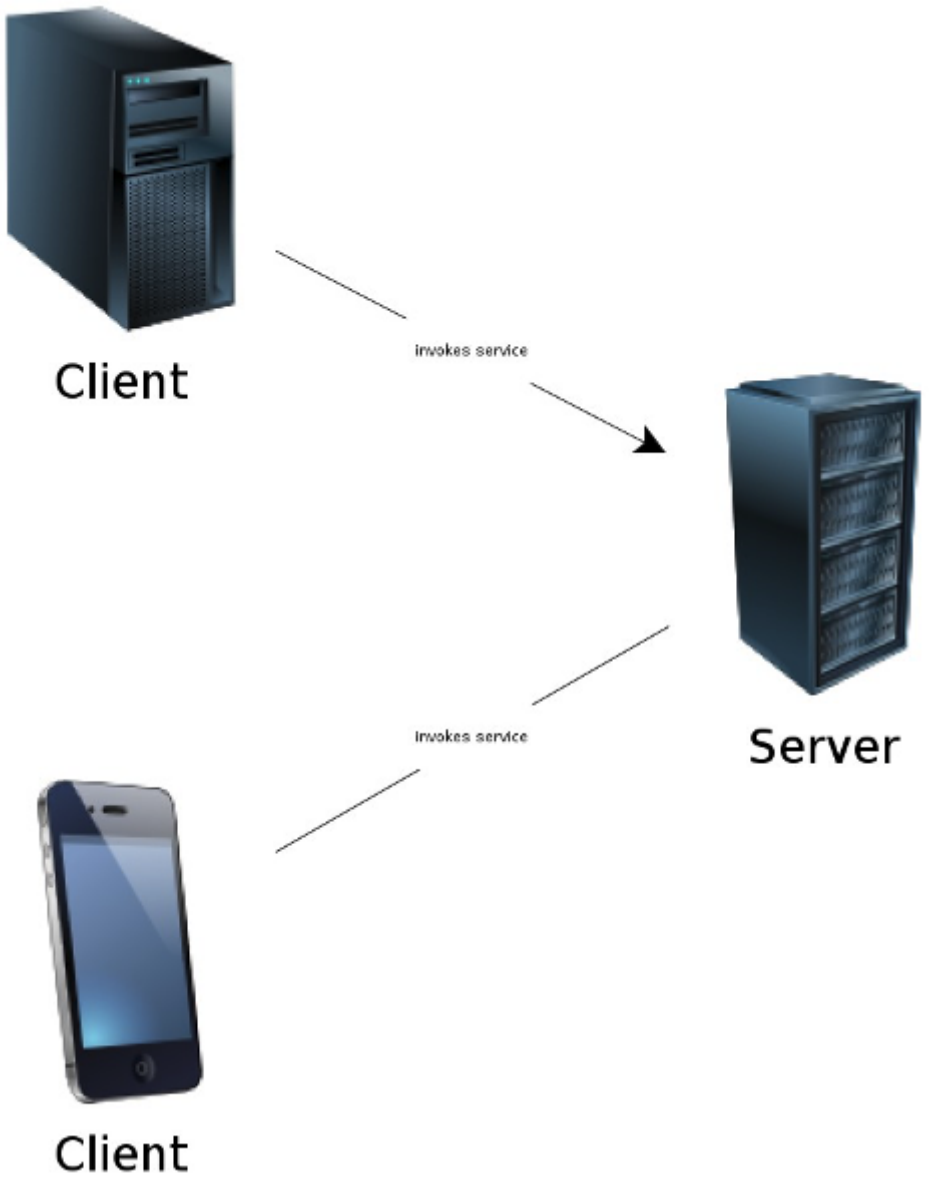
基础概念

- Client使用服务
- Server提供服务

核心概念

- 组件只有clients和servers
- Server不知道clients的数量

- client知道Server的身份
- 连接件（connetor）是一个协议——RPC-based network interaction protocols

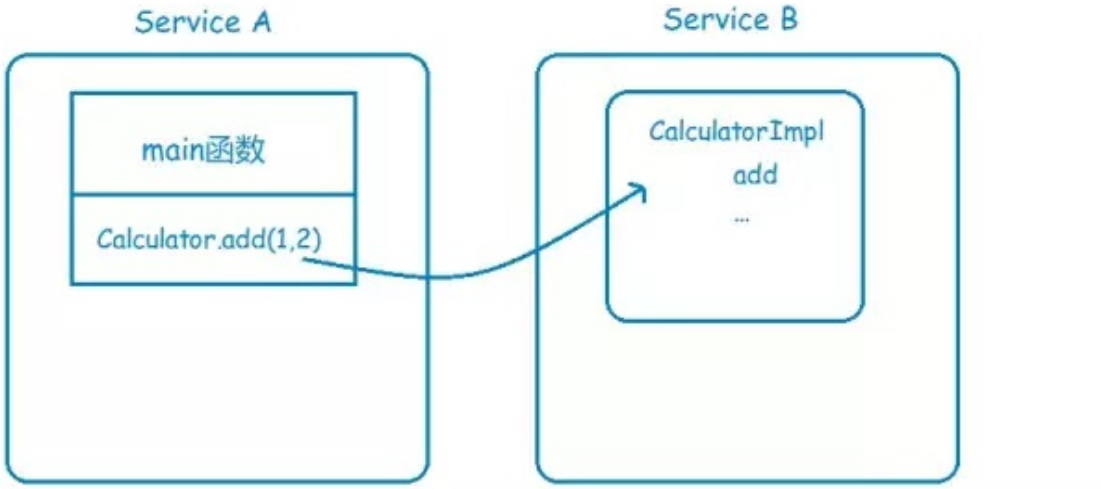


4.5.2 p2p架构 - 分布式架构

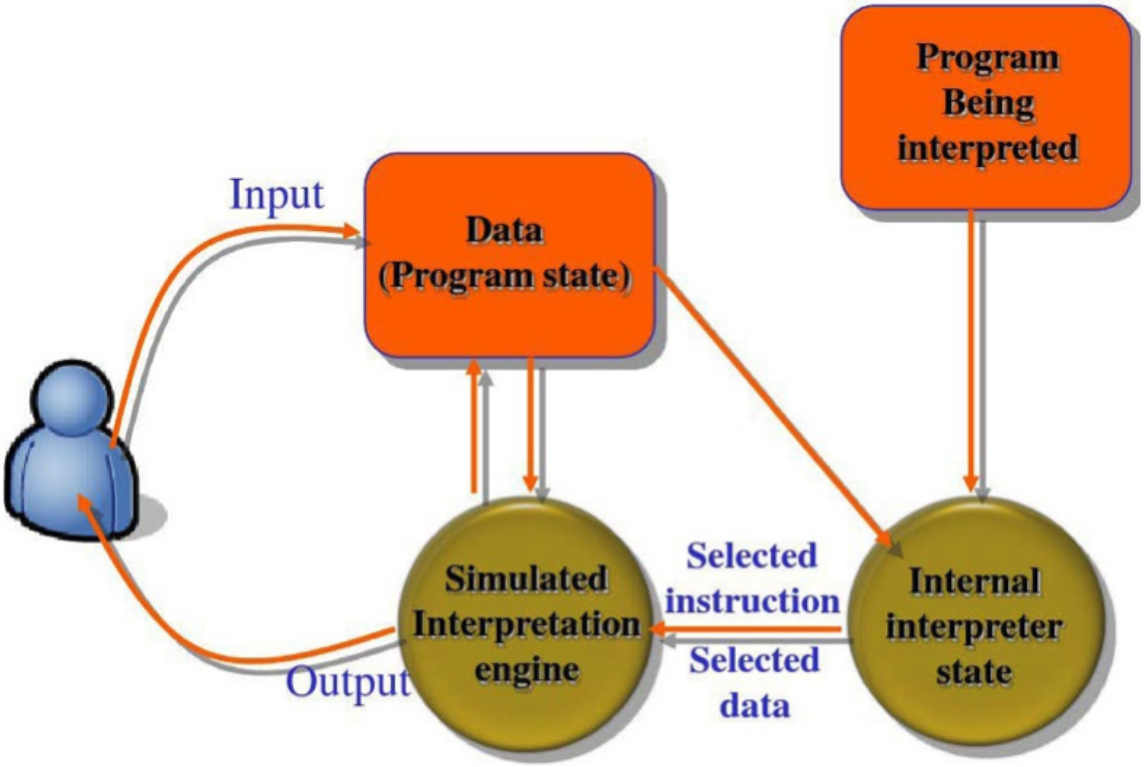
4.6 Remote Invocation Architectures 远程调用模式

> 关注于远程调用（通过网络）

- 优点：通过分布式计算提高性能
- 缺点：更加耦合，可寻址性的管理（收集对象的唯一标识）增加了通信开销
- 引入代理可以解耦。有助于灵活性，可维护性和可扩展性。但网络通信可能会引入新的错误类型，由于网络延迟和带宽有限，行为可能会改变



4.7 interpreter style 架构



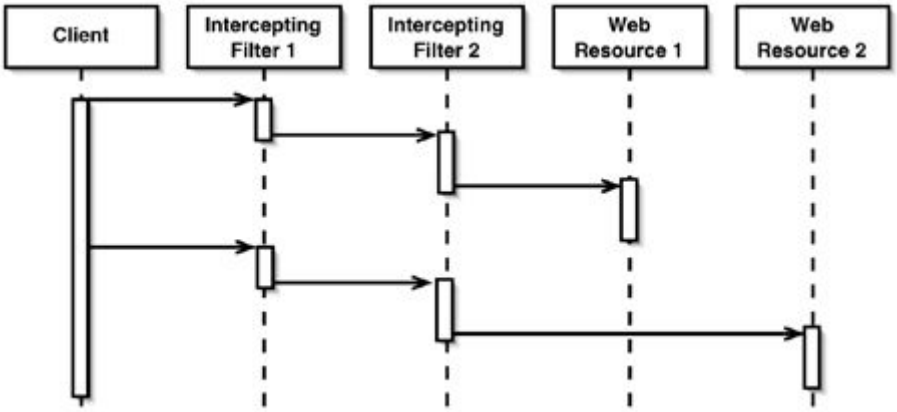
- 在动态的环境下，根据不同的输入产生不同的输出
- 一般而言解释器架构都是用来创建虚拟机
- 经典应用：`java虚拟机`

4.8 Interceptor Style 拦截器

- 开闭原则（可扩展，不可修改）
- 例子：调用登陆组件前，被拦截下来输入信息
- 优点：拦截器组件可重用，出色的可扩展性，解耦
- 缺点：会因为递归拦截变得很麻烦

`拦截器`主要用于`拦截用户请求并作相应的处理`

例如：Spring MVC通过拦截器可以进行权限验证、记录日志等

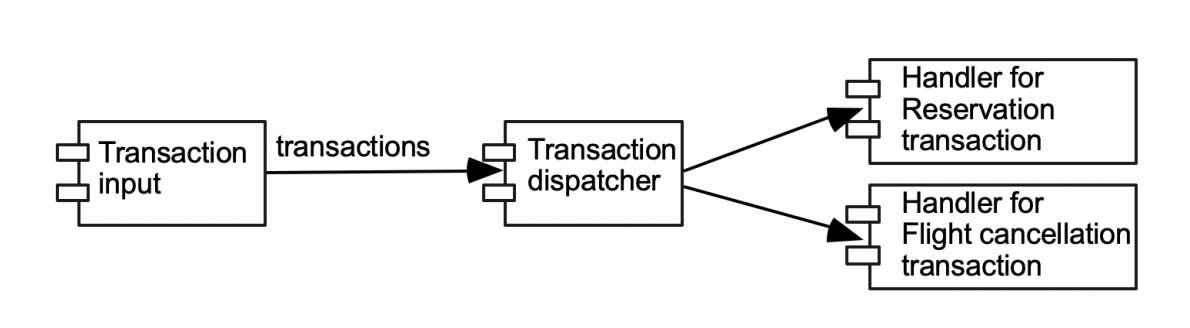


4.9 GUI Architectures

- 基于MVC

4.10 Transaction-Processing 事务处理架构

- 事务处理架构 `处理一个接一个的一系列的输入`
- 事务是应用程序中一系列严密的操作，所有操作必须成功完成，否则在每个操作中所作的所有更改都会被撤消。也就是 `事务具有原子性`，一个事务中的一系列的操作要么全部成功，要么一个都不做
- 事务的结束有两种，当事务中的所以步骤全部 `成功` 执行时，事务提交。如果其中一个步骤 `失败`，将**发生回滚操作，撤消到事务开始时的操作**



6 质量属性

6.1 availability 可用性

- 可用性关注的问题：`如何检测故障；发生故障的频度；出现故障时的现象；系统故障排除的时限；如何防止故障的发生；发生故障时的处理。`

场景部分	可能的值
刺激源	系统内部、外部
刺　激	错误：疏忽、崩溃、时间、响应
制　品	系统的处理器、通信通道、持久性存储器、进程
环　境	正常、降级模式
响　应	系统检测到事件，进行以下活动之一：记录故障，通知用户或系统；根据已定义的规则禁止故障源等
响应度量	系统修复时间，系统可以在降级模式下运行的时间间隔等

6.2 modifiability 可修改性

- 可修改性关注的问题：`可以修改什么？何时以及谁进行修改？`
- 刺激源：`开发人员、系统管理员、用户`（三者通过不同的方面进行修改）
- 刺激：修改淘宝网上的商品信息
- 制品：卖家管理页面（系统用户界面）
- 环境：运行时
- 响应：对一件商品的信息进行修改时，不会影响其他的商品以及功能
- 响应度量：不影响其他商品的正常售卖等功能

场景部分	可能的值
刺激源	开发人员、系统管理员、最终用户
刺　激	希望修改功能，质量属性或系统容量
制　品	系统用户界面、系统运行平台、环境或与目标系统交互的系统
环　境	设计时、构建时、编译时、运行时
响　应	查找构架中需要修改的位置，进行修改且不会影响其他功能，对所做的修改进行测试；部署所做的修改
响应度量	根据所影响的元素的数量的成本、资金；该修改对其他功能的影响

6.3 security 安全性

- 安全性的关注点：`阻止非授权使用的能力`
- 场景：有黑客对淘宝网进行sql注入，试图非法入侵网站后台，获取用户信息
- 刺激源：黑客（非授权用户）
- 刺激：试图采用非法手段来入侵淘宝后台以获取信息
- 制品：淘宝中的数据
- 环境：在线环境
- 响应：对访问用户进行验证，阻拦不正当的用户访问数据

- 响应度量：查到非法入侵时在1秒以内做出反应，进行阻拦处理，保护数据安全性

场景部分	可能的值
刺激源	授权或非授权用户；访问了有限的资源/大量资源
刺　激	试图修改数据，访问系统服务
制　品	系统服务、系统中的数据
环　境	在线或离线、直接或通过防火墙入网
响　应	对用户验证，阻止或允许访问数据或服务
响应度量	避开安全措施所需要的时间或资源；恢复数据/服务

6.4 usability 易用性

- 易用性的关注点：对用户来说完成某个期望任务的难易程度
- 场景：在一个商品店铺中浏览商品时，会显示"类似商品"方便用户选择
- 刺激源：淘宝用户
- 刺激：是用户使用更加便捷
- 制品：淘宝系统
- 环境：运行时
- 响应：显示出相关商品或者类似商品
- 响应度量：80%可能推荐出用户满意的商品

场景部分	可能的值
刺激源	最终用户
刺　激	想要学习系统特性、有效使用系统、使错误的影响最低，适配系统
制　品	系统
环　境	在运行时或配置时
响　应	上下文相关的帮助系统；数据和/或命令的集合，导航；撤销、取消操作，从系统故障中恢复；定制能力，国际化；显示系统状态
响应度量	任务时间，错误数量，用户满意度、用户知识的获得，成功操作的比例等

6.5 testability可测试性

- 可测试性的关注点：揭示软件缺陷的难易程度
- 场景：内测用户使用不正确的用户名密码来登录系统
- 刺激源：淘宝内测用户
- 刺激：内测阶段，测试登录系统，输入错误密码

- 制品：完整应用
- 环境：完成时
- 响应：密码错误的情况下不能登录
- 响应度量：错误密码账户100%不能完成登录

场景部分	可能的值
刺激源	单元开发人员、系统集成人员、系统验证人员、测试人员、用户
刺　激	已完成的一个阶段，如分析、构架、　　类和子系统的集成，所交付的系统
制　品	设计、代码段、完整的应用
环　境	设计时、开发时、编译时、部署时
响　应	可以控制系统执行所期望的测试
响应度量	已执行的可执行语句的百分比；最长测试链的长度，执行测试的时间，准备测试环境的时间

6.6 Scalability 伸缩性

- 伸缩性要求`软件系统能够跟着所需处理的工作量相应的伸缩`。例如如果计算机是多CPU多核心的，软件是否能够相应的利用到这些计算资源。另一个方面就是软件是不是能够部署到分布式的网络，有效的利用网络中的每一个节点的资源。

有两个方向的伸缩，垂直和水平：

- 在`垂直方向`的伸缩（scale up）是指`提高单节点的处理能力`，比如提高CPU主频和内核数，增大内存，增大磁盘容量等等。SAP的HANA就是一个典型的垂直方向的伸缩。
- 在`水平方向`的伸缩（scale out）通常是指通过并发和分布的方式来`增加节点`以提高处理能力。Hadoop就是一个很好地水平伸缩的例子。

6.7 performance 性能

- 性能的关注点：`事件源的数量和到达模式。`
- 场景：淘宝用户要购买一件商品，点击购买，启动付款进程
- 刺激源：淘宝用户
- 刺激：用户点击购买商品，启动付款进程
- 制品：淘宝购物系统
- 环境：正常环境
- 响应：用户请求被处理
- 响应度量：`响应时间平均在3秒以内`

场景部分	可能的值
刺激源	大量独立源中的一个，可能来自系统内部
刺　激	定期、随机或偶然事件到达
制　品	系统
环　境	正常模式；超载模式
响　应	处理刺激；改变服务级别
响应度量	等待时间、时间期限、吞吐量、抖动、缺失率、数据丢失

6.8 reliability 可靠性

- 可靠性：`软件系统在一定的时间内无故障运行的能力`
- 高容错：时间冗余，计算冗余，数据冗余
- 主动冗余（成本高，需解决一致性问题）：关键性领域用（供电，银行）
- 时间冗余：回滚

6.9 maintainability可维护性

可维护性有两个不同的角度，一个是指从软件用户和运维人员的角度，另一个是从软件开发人员的角度：

- 从`用户和运维人员`的角度，软件的可维护性是**指软件是不是容易安装，升级，打补丁，有了问题是不是容易修复，能不能很容易的获得支持。**
- 从`开发人员`的角度，软件的可维护性是指软件的**架构是不是清楚简单，代码是不是容易阅读，有了问题是不是容易定位错误的原因，有没有可以提供帮助的文档，等等。**