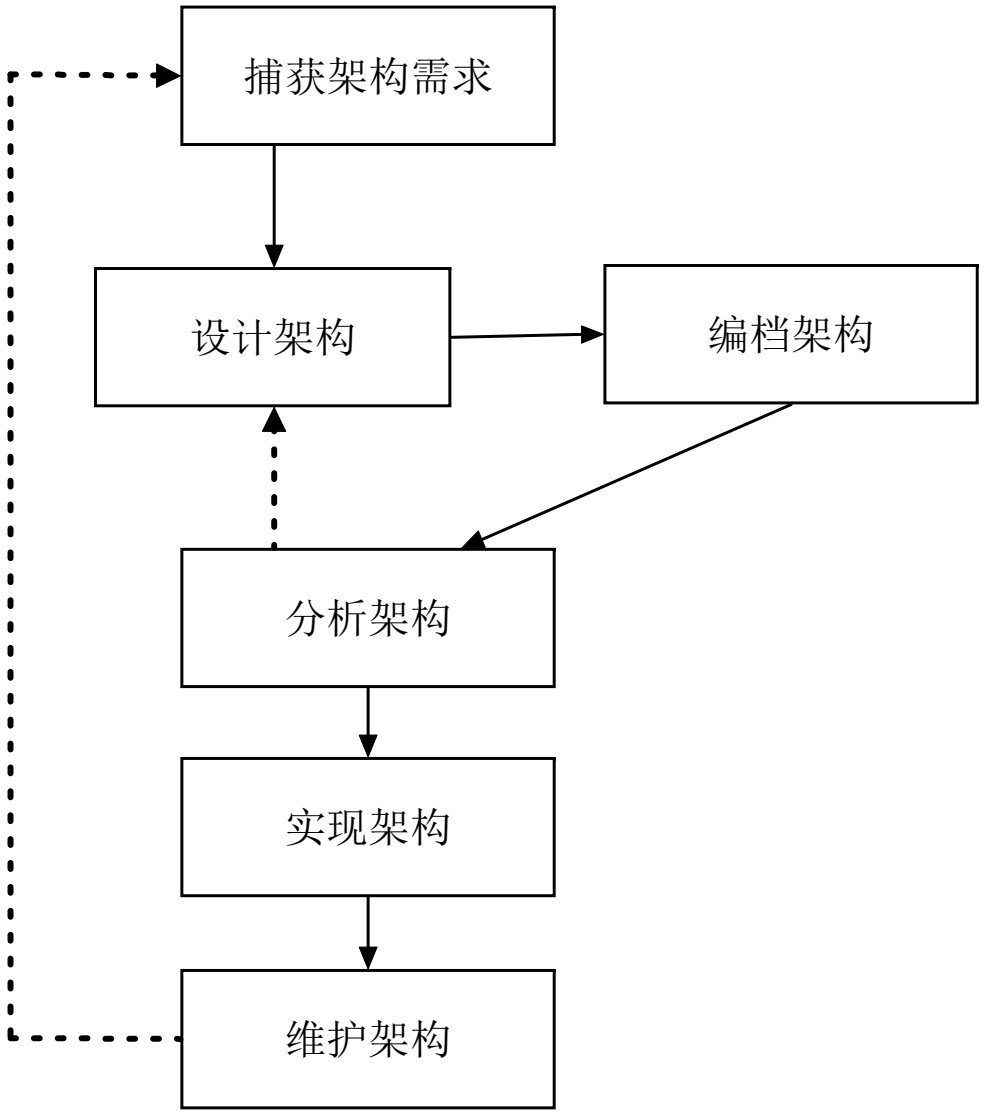
第5章 软件体系结构设计与评估

内容

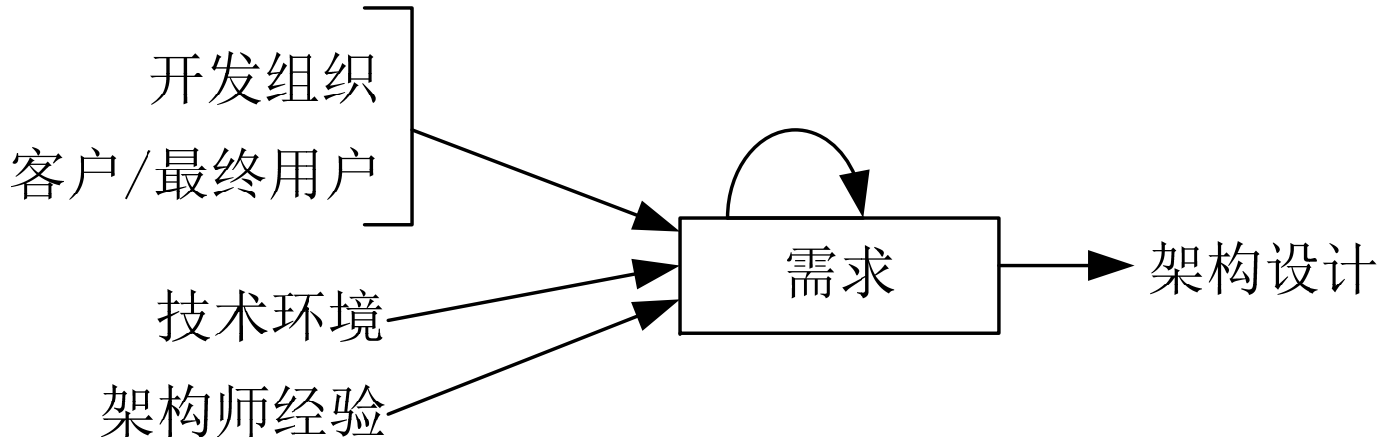
- **5.1 架构为中心的软件开发过程**
- **5.2 属性驱动的设计方法**
- **5.3 基于模式的设计方法**
- **5.4 模块设计与评估方法**
- **5.5 软件体系结构评估**

软件架构生命周期模型



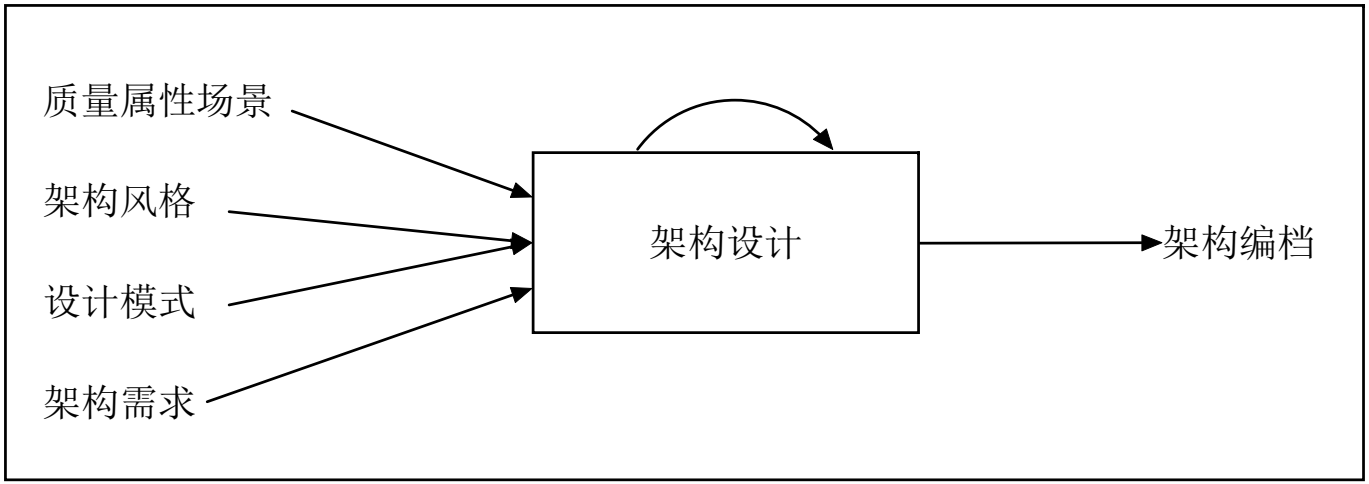
软件架构开发核心过程（1）

- **架构需求**
  - 软件体系结构需求是指用户对目标软件系统在功能、行为、性能、设计约束等方面的期望，受技术环境和架构师经验的影响。
  - 一般可根据系统的质量目标、系统的商业目标和系统开发人员的商业目标获取需求。

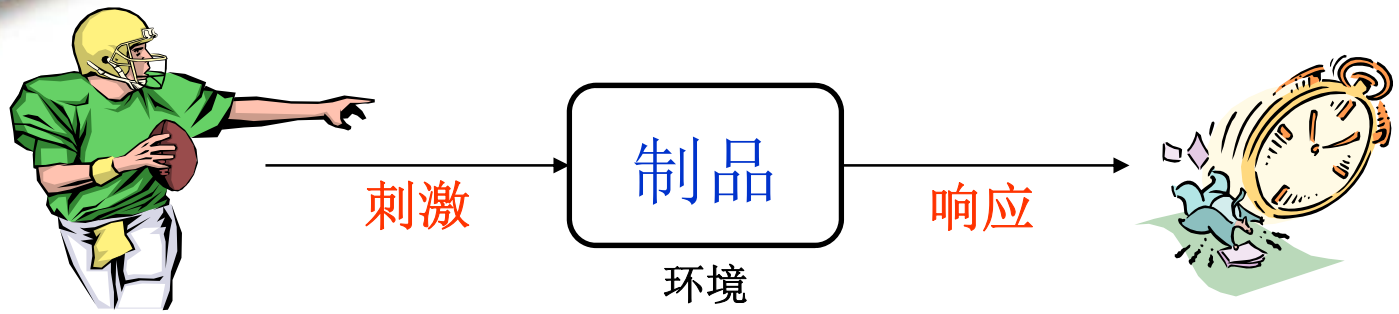


软件架构开发核心过程（2）

- 架构设计
  - 首先制定许多设计策略，再根据这些决策考虑不同的体系结构和架构视图推理，以确定最佳软件体系结构
  - 架构需求用于刺激、证明设计策略
  - 质量属性场景用于推理和验证设计策略
  - 通常可根据架构风格、设计模式等方法确定设计策略



回顾：质量属性场景（Scenario）

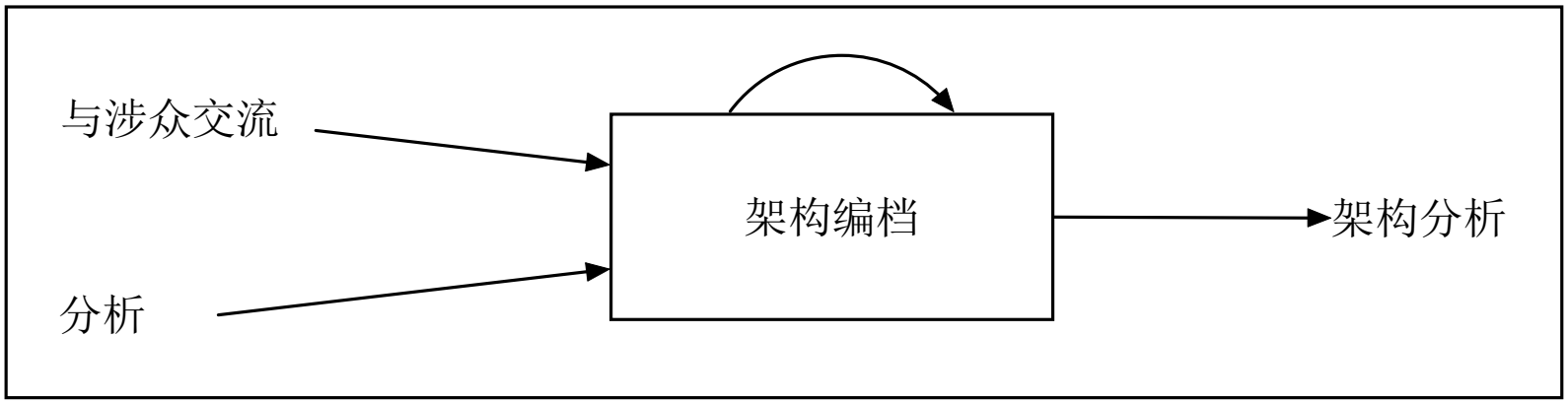


质量属性场景（Scenario）的6个组成部分

- 刺激源：生成刺激的实体(人、系统或其它)
- 刺激：当到达时引起系统进行响应的条件
- 环境：刺激到达时，系统所处的状态，如系统可能处于过载，或者正在运行或处于其他情况；或指该刺激在系统的某些条件内发生
- 制品：被刺激的事物。可能是整个系统或系统的一部分
- 响应：刺激到达时所采取的行动
- 响应度量：当响应发生时，应该能够以某种方式进行度量，以便对需求进行测试，并明确质量属性需求

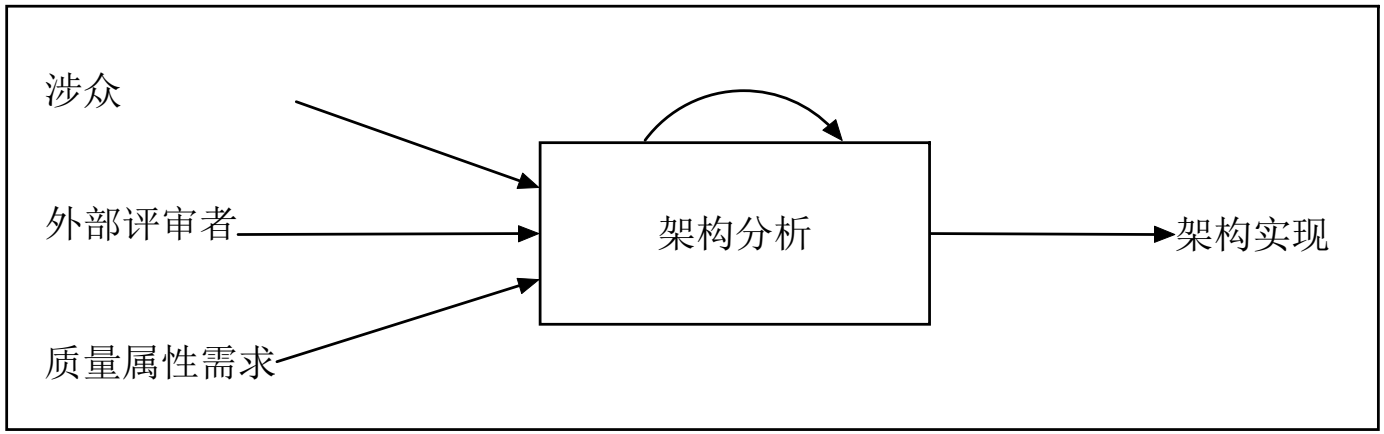
软件架构开发核心过程（3）

- 架构编档
  - 软件架构文档通常按照多视图方式加以组织，其核心是一系列满足不同涉众要求的视图
  - 视图中通常包括描述主要构件和视图关系的主要表示
  - …



软件架构开发核心过程（4）

- 架构分析
  - 在制定主要架构决策之前，架构设计、编档和分析活动是不断迭代的
  - 该阶段需要外来的审查人员参与主要架构的评估
  - 评估的主要目的是通过分析来识别架构中潜在的风险以及验证已完成的质量属性需求



软件架构开发核心过程（5）

- 架构实现
  - 当架构转换为代码时，必须考虑所有常规的软件工程和项目管理中的注意事项，如详细设计、实现、测试、配置管理等
  - 在以架构为中心的开发过程中，开发团队的组织结构必须容易映射到软件架构上，反之亦然

软件架构开发核心过程（6）

- 架构维护
  - 有了软件架构并不等同于该架构是良好编档、易于传播或维护的
  - 架构维护是架构生命周期中的重要一步，架构必须像其他被维护的系统制品一样编档和维护

架构中心方法与RUP模型融合

核心工作流

业务建模 (含SA需求建模)

需求分析 (含SA需求分析)

分析和设计(含SA分析、设计、编档、评估、维护)

实现（含SA实现）

测试

部署

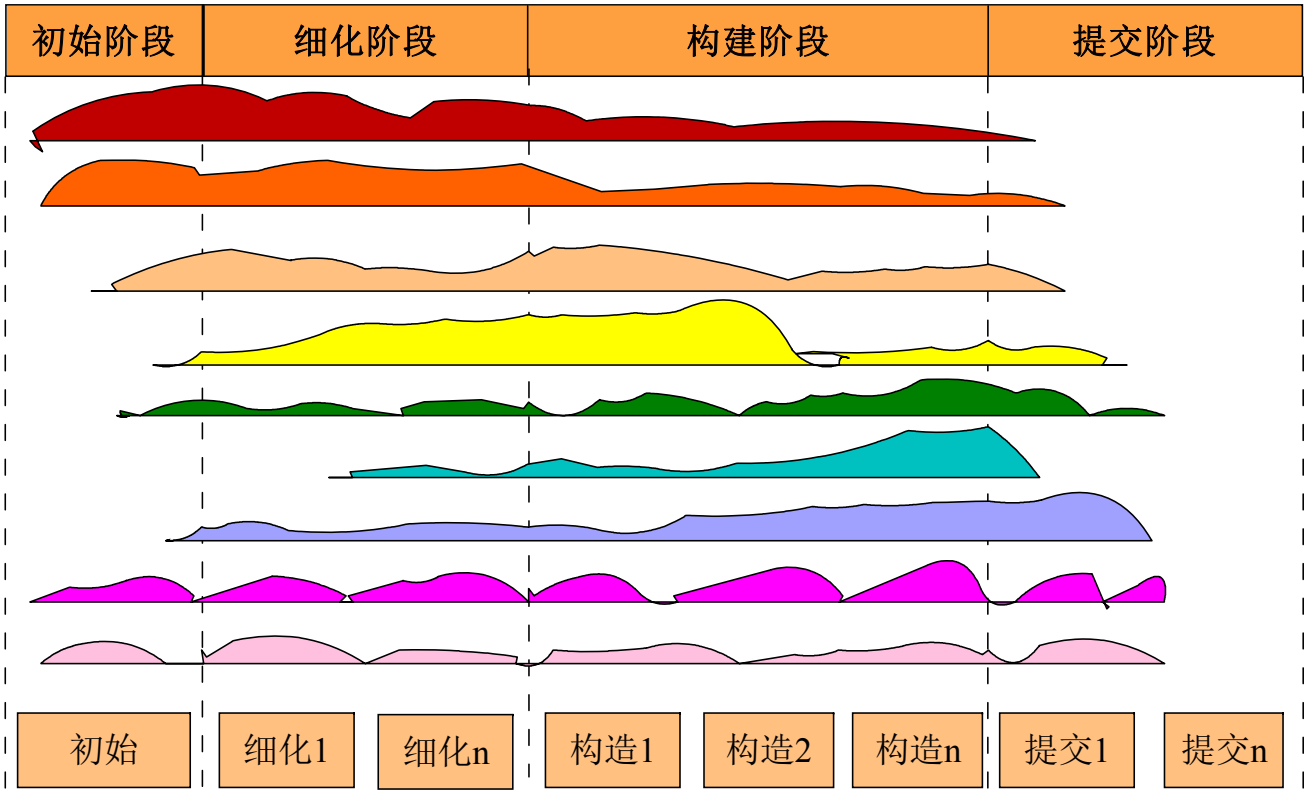
支持工作流

配置和变更管理

项目管理

环境

操作和支持

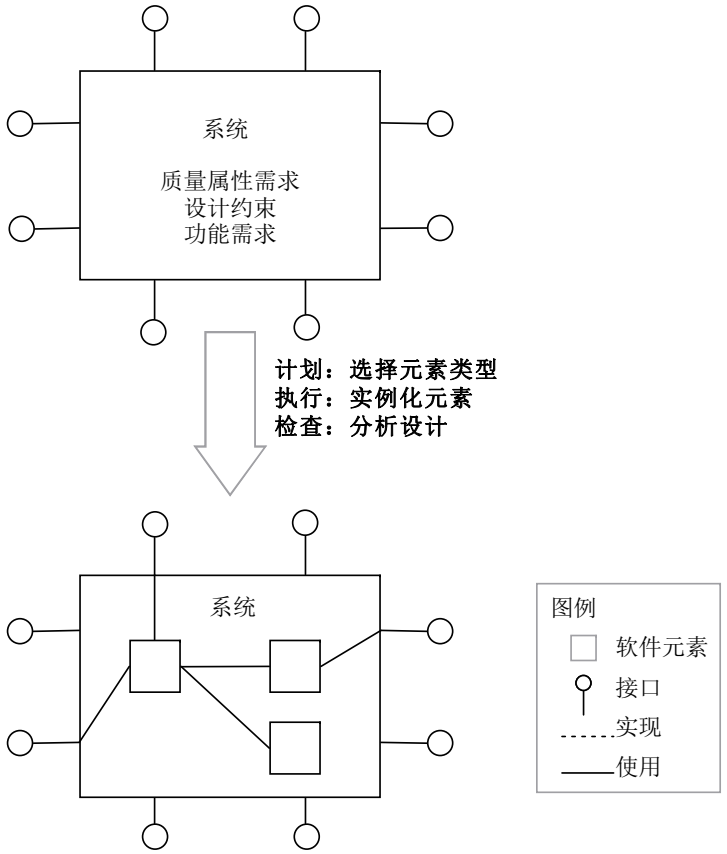


内容

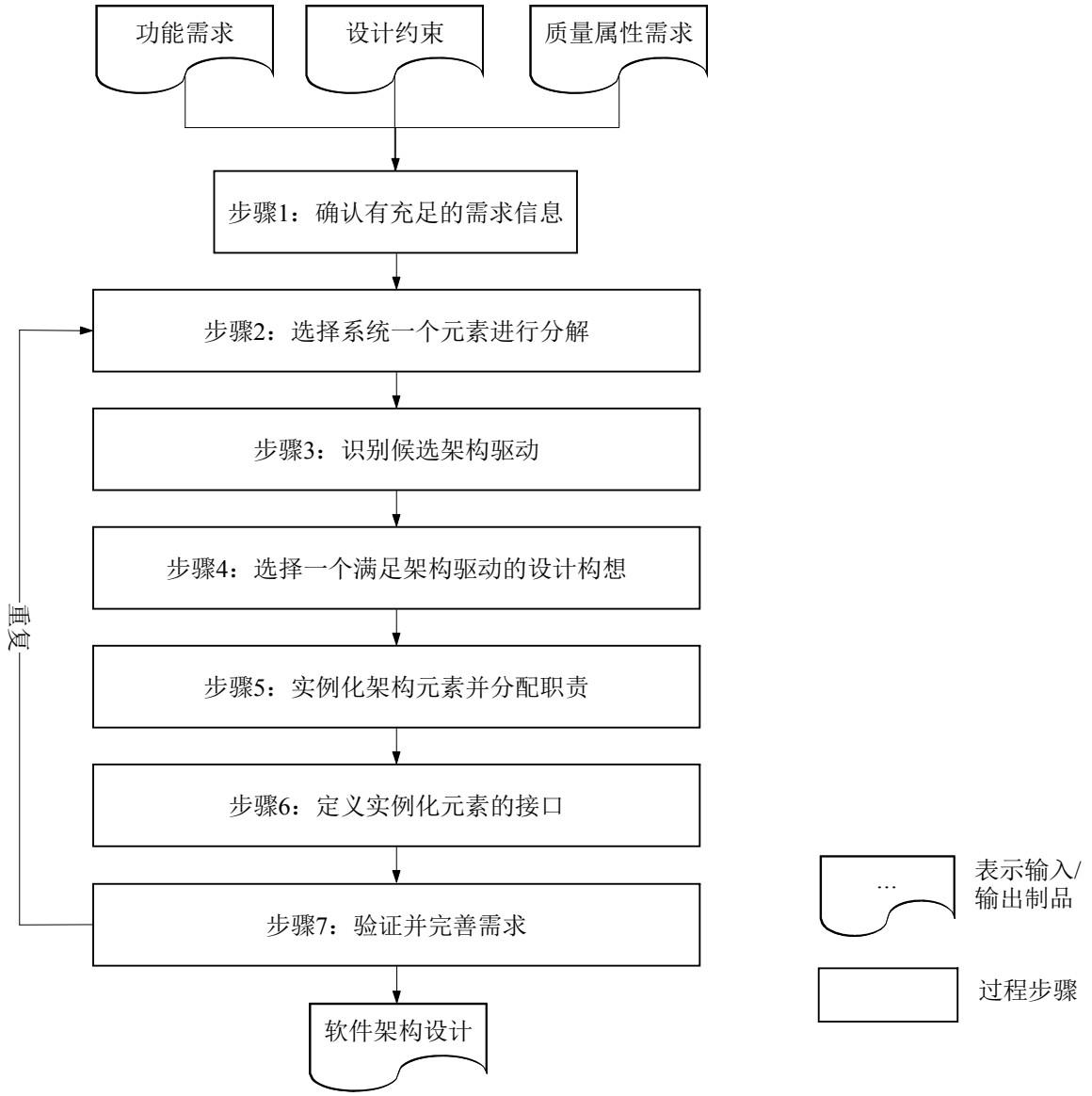
- 5.1 架构为中心的软件开发过程
- 5.2 属性驱动的设计方法
- 5.3 基于模式的设计方法
- 5.4 模块设计与评估方法
- 5.5 软件体系结构评估

ADD方法概述

- 属性驱动的设计方法(Attribute-Driven Design, ADD)是定义软件架构的一种方法，可根据软件质量属性需求实施架构设计过程
- ADD通过一个分解系统或者系统元素的循环过程，使用架构模式和策略来满足系统质量属性需求，以完成分解操作和模式



ADD的设计步骤总览


功能需求
设计约束
质量属性需求
步骤1：确认有充足的需求信息
步骤2：选择系统一个元素进行分解
步骤3：识别候选架构驱动
步骤4：选择一个满足架构驱动的设计构想
步骤5：实例化架构元素并分配职责
步骤6：定义实例化元素的接口
步骤7：验证并完善需求
重复
软件架构设计
…
表示输入/输出制品
过程步骤

ADD设计步骤1——确认有充足的需求信息

- 首先需要搜集足够多的系统需求，并将其作为ADD的输入
- 还需捕获足够多的系统质量属性需求并将其描述成具体可度量的质量属性响应（刺激源、刺激、制品、环境、响应、响应度量）
- 架构师可根据这些质量属性需求选择合适的设计模式和策略

ADD设计步骤2——选择系统一个元素进行分解

- 在步骤2中，将所选择的系统元素作为后续步骤的设计重点
- 当首次执行该步骤时，唯一可以分解的元素是系统本身（大泥球），那么就可将所有需求分配给该系统
- 已经将系统分为两个或者多个元素，也将需求分配给这些元素。那么就需要从中选择一个元素作为后续步骤的重点
- 架构师可根据这些质量属性需求选择合适的设计模式和策略

ADD设计步骤3——识别候选架构驱动

- 根据需求对架构的影响程度，再次将这些需求进行排序
- 按照（需求对于涉众的重要程度，需求对架构的潜在影响力）对需求进行排序
- 例如某个需求表示为（H,H），则该需求对于涉众非常重要，同时也会对架构的设计产生很大的影响

ADD设计步骤4——
选择一个满足架构驱动的设计构想

- 选出架构中的主要元素及其之间的关系
  - 识别候选架构驱动相关的设计问题
  - 对于每一个设计问题，创建一个解决该问题的模式列表
  - 从模式列表中选出最能满足候选架构驱动的模式
  - 考虑目前已经识别的模式，并确定他们之间的关系，将所选中的模式进行组合可以产生新的模式
  - 通过不同的架构视图来描述已经选择的模式
  - 评估并解决设计构想不一致性的问题

ADD设计步骤5

——实例化架构元素并分配职责

- 实例化在之前步骤中选择的软件元素，根据元素的类型为它们分配职责
- 实例化的元素的职责也来源于候选架构驱动关联的功能性需求以及其父元素关联的功能性需求
- 在执行完步骤5后，父元素关联的功能性需求通过一系列子元素的职责进行表达

ADD设计步骤6——
定义实例化元素的接口

- 一个接口可能包括以下任何一项：
  - 操作语法（如签名）
  - 操作语义（如描述，前置条件，后置条件，约束）
  - 信息交换（如信号事件，全局数据）
  - 个体元素或操作的质量属性需求
  - 错误处理

ADD设计步骤7——验证并完善需求

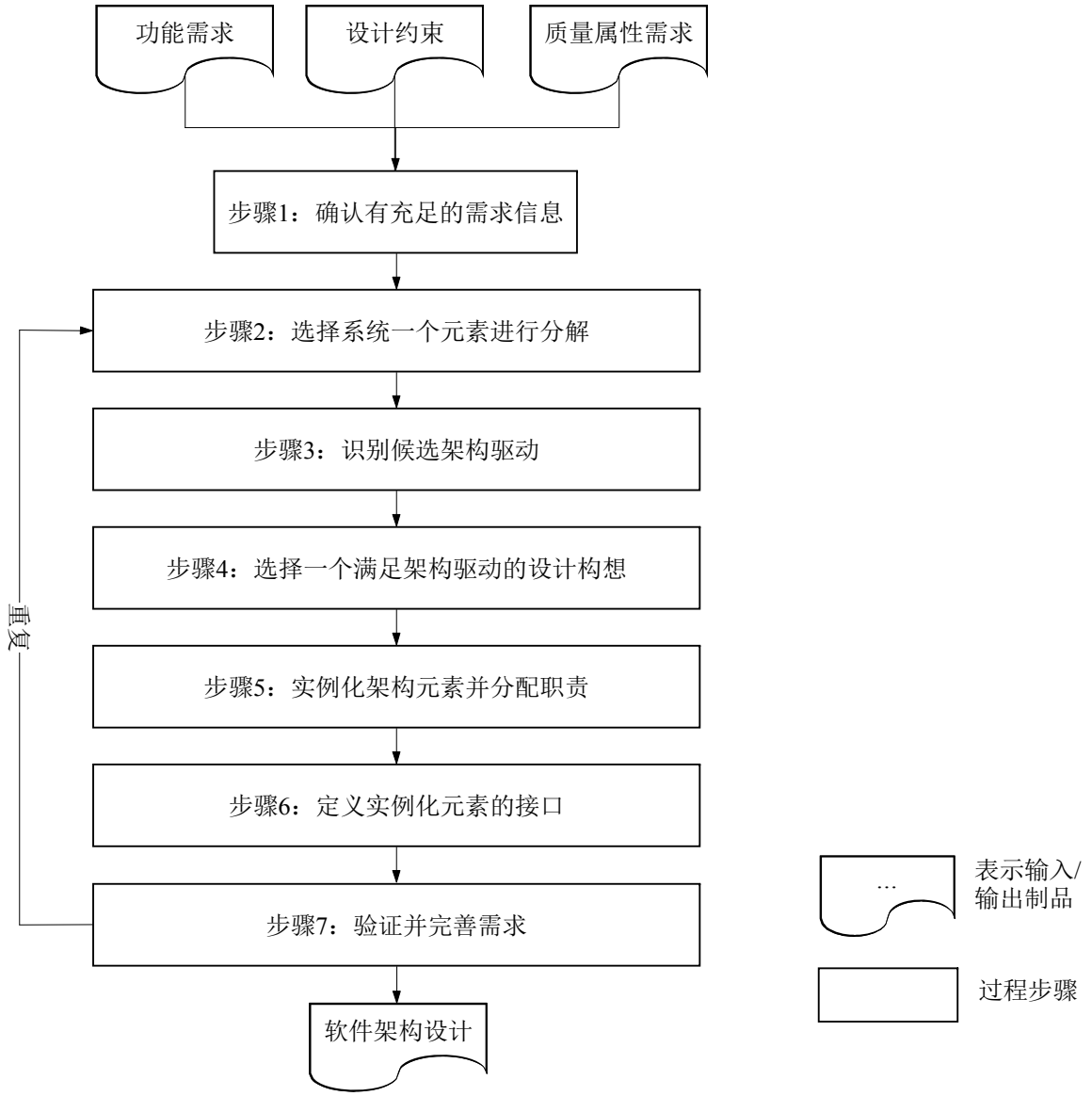
- 检验元素分解是否满足功能性需求、质量属性需求以及设计约束。同时，准备子元素为接下来的分解做准备
  - 核实与父元素关联的所有功能性需求、质量属性需求以及设计约束是否已经在分解的过程中分配给子元素。对于个体元素，将分配给子元素的所有职责转换成功能性需求。
  - 必要时，对于个体子元素完善其质量属性需求。

ADD设计步骤8——

分解系统其它元素，重复步骤2至7

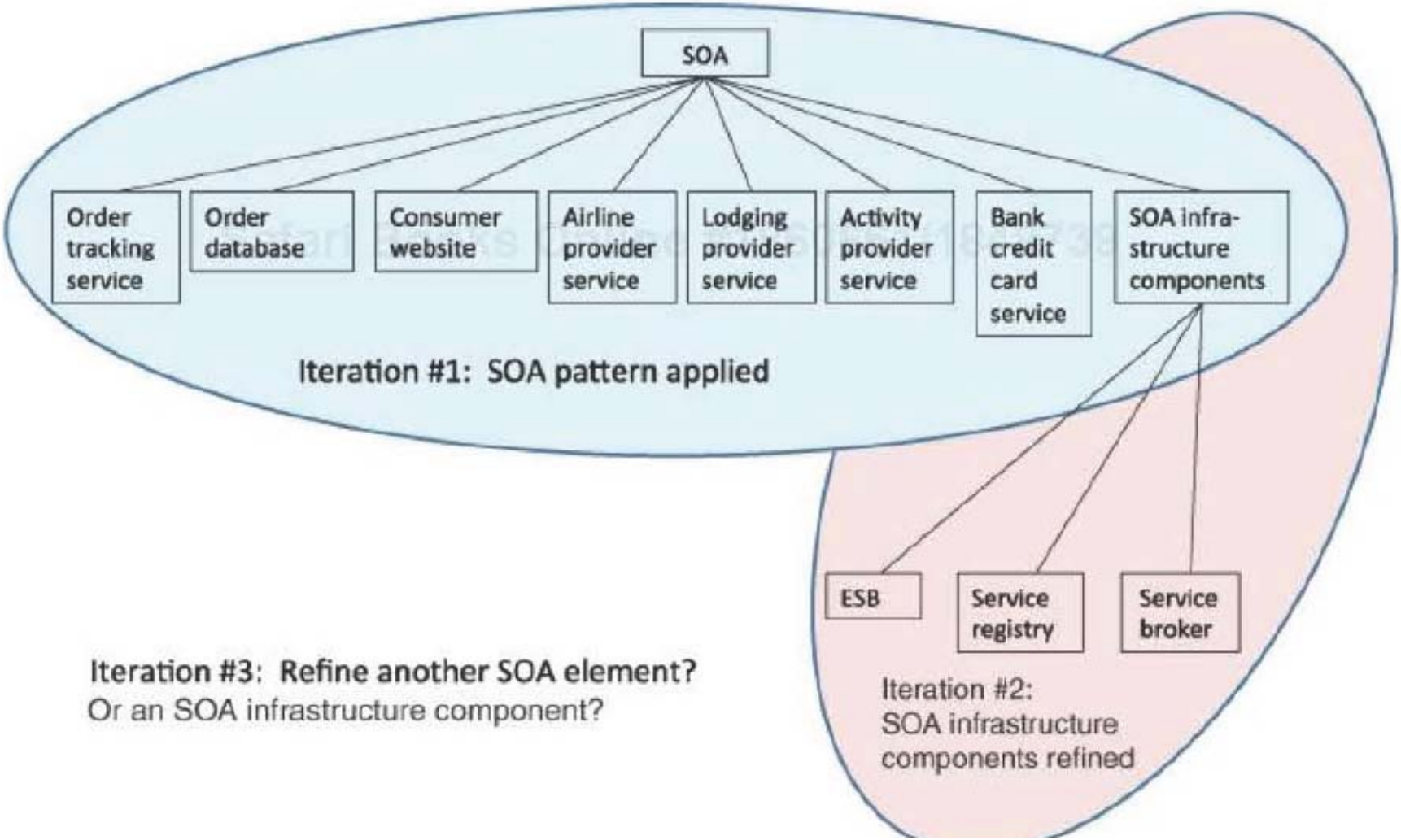
- 完成步骤1-7，就已经将1个父元素分解成多个子元素。每个子元素都是一个职责的集合，包括接口描述、功能性需求、质量属性需求以及设计约束
- 现在可以返回到步骤2的分解过程，继续选择下一个元素进行分解

回顾：ADD的设计步骤总览



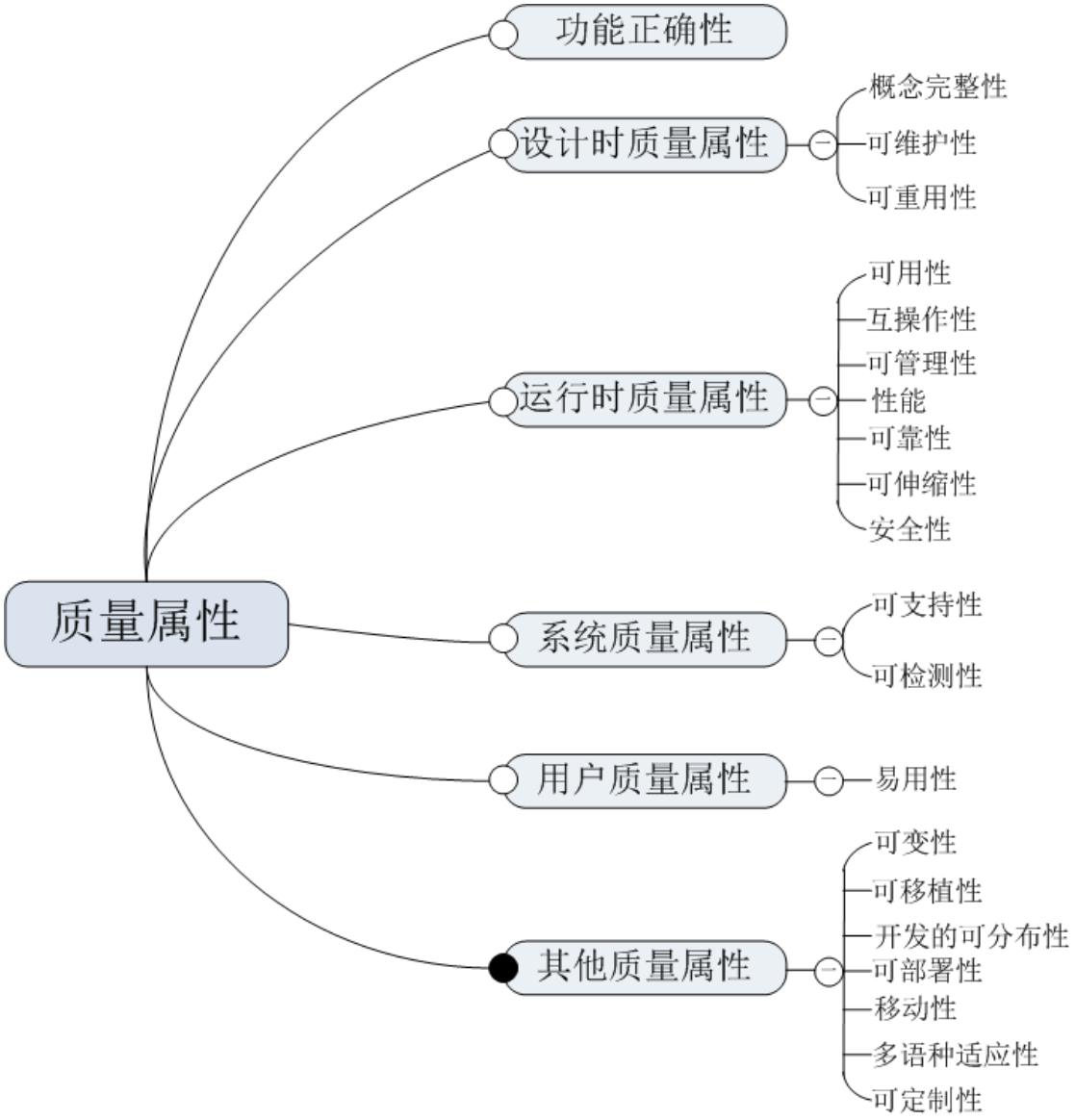
Example: SOA

- Iteration 1 applied the SOA pattern. Iteration 2 refined the infrastructure components.



请思考：SOA支持哪些质量属性？

松散耦合
位置透明
协议独立



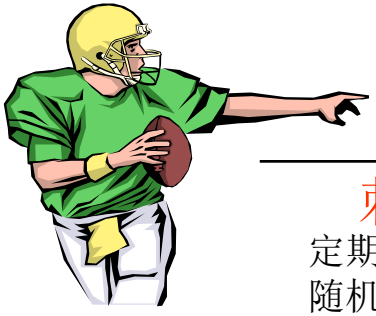
几种常见质量属性设计策略

- 性能
- 可用性
- 可修改性
- 安全性
- 易用性

质量属性设计策略

- 质量属性设计策略能够直接影响系统对一些刺激的响应，不同的设计策略往往对应着不同的质量属性。
- 质量属性设计策略就如设计模式一样，是已经广泛使用的一种设计方法

回顾：性能一般场景



刺激源

大量的独立源中的一个，可能来自系统内部

刺激

定期事件到达；
随机事件到达；
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

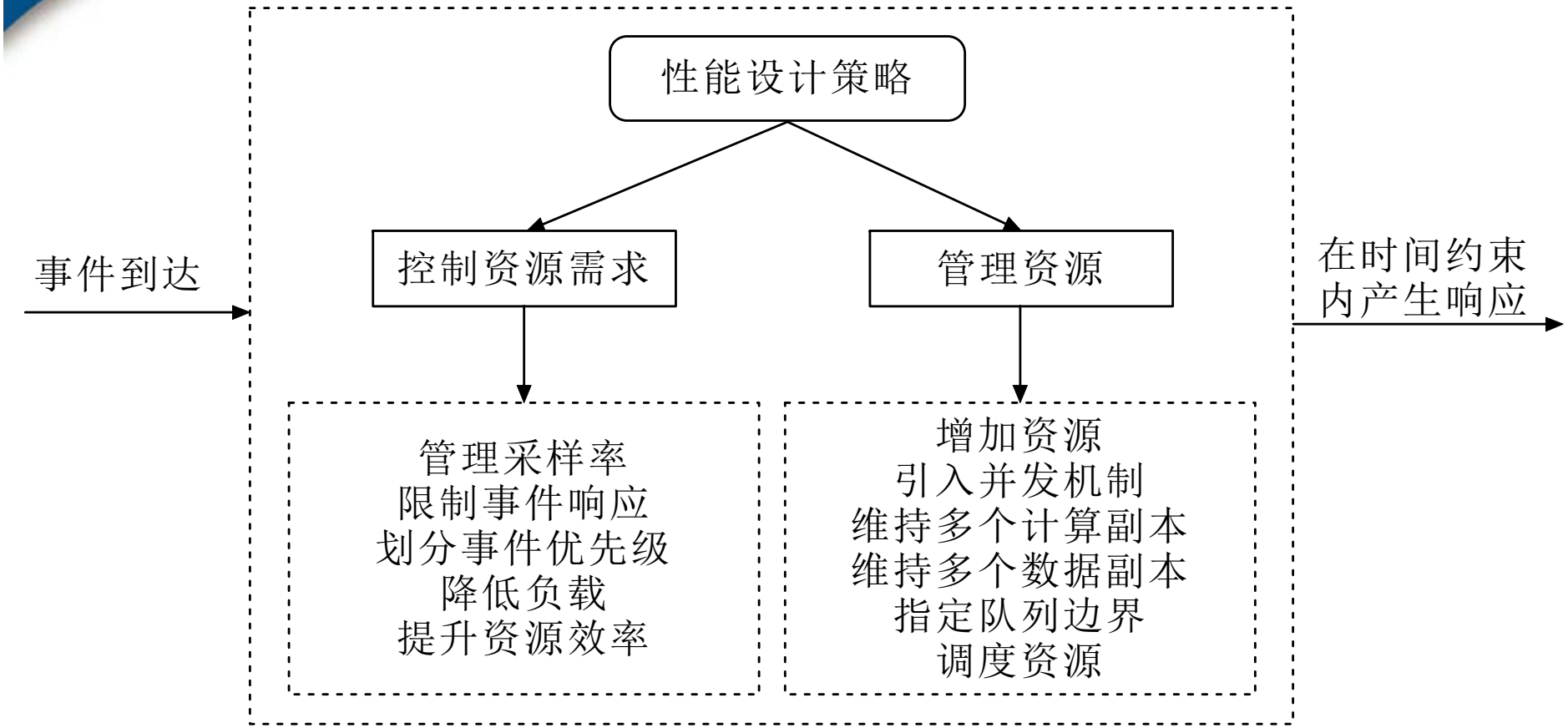
等待事件、期限、吞吐量、抖动(变化)、缺失率与数据丢失(太忙而不能处理)

性能的一般场景的可能取值

性能设计策略（1/5）

- 目标
  - 对在一定时间限制内到达系统的事件生成一个响应
- 产生响应时间的基本因素
  - 资源消耗
    - 资源包括CPU、数据存储、网络带宽、内存以及系统所定义的实体
  - 闭锁时间
    - 资源争用
      - 对一个资源争用越多，就越有可能引入等待时间
    - 资源的可用性
      - 资源离线、构件故障等原因都会导致资源不可用，以增加等待时间
    - 对其他计算的依赖
      - 需要等待自己启动的计算结果，或者等待别的构件提供的计算结果

性能设计策略（2/5）


性能设计策略
事件到达
控制资源需求
管理资源
在时间约束
内产生响应
管理采样率
限制事件响应
划分事件优先级
降低负载
提升资源效率
增加资源
引入并发机制
维持多个计算副本
维持多个数据副本
指定队列边界
调度资源

性能设计策略（3/5）

- 资源需求
  - 提高计算效率
    - 改进在关键的地方所使用的算法的效率将减少等待时间
    - 用一种资源换取另一种资源，例如可以把数据保存在硬盘上，也可以重新生成数据，取决于时间和空间资源的可用性
  - 减少计算开销
    - 如果没有计算请求，那就减少处理需求
    - 例如，删除仲裁者可以减少等待时间
  - 管理事件率
    - 降低监视环境对变量的取样频率，就可以减少需要管理的事件
  - 控制采样频率
    - 用一个较低的频率对排队的请求进行采样，也可以减少需要处理的事件
  - 限制执行时间
    - 限制用多少执行时间对事件作出响应
    - 对于迭代式、依赖于数据的算法，可以限制迭代次数
  - 限制队列大小
    - 控制了队列的到达事件的最大数量，因此控制了用来处理到达事件的资源

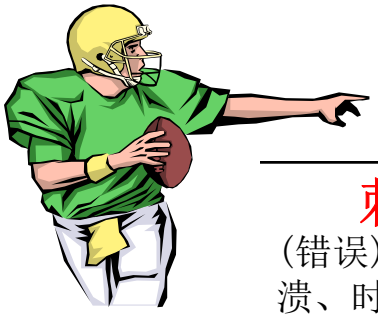
性能设计策略（4/5）

- 资源管理
  - 引入并发
    - 如果可以并行进行请求，可以减少闭锁时间
    - 通过在不同的线程上处理不同的事件流或创建额外的线程来处理不同的活动集来引入并发
  - 维持数据或计算的多个副本
    - 减少在中央服务器上进行所有的计算所引起的资源争用
    - 注意：要保证副本的数据一致和同步
  - 增加可用资源
    - 速度更快的处理器、额外的处理器、额外的内存以及速度更快的网络都可以减少等待时间

性能设计策略（5/5）

- 资源仲裁
  - 标准：最佳的资源使用、请求重要性、最小化所使用资源的数量、使等待时间最少、使吞吐量最大等
  - 先进/先出
    - 同等对待资源的所有请求
    - 一个请求可能会被另一个需要更长等待时间来生成响应的请求阻止
  - 固定优先级调度
    - 为每一个请求分配一个特定的优先级，并按照优先级的顺序分配资源
    - 确保高优先级的请求得到更好的服务，但是优先级较低的请求可能要等待很长时间才可能得到资源
    - 语义重要性、时限时间单调、速率单调
  - 动态优先级调度
    - 轮转：对请求进行排序，把资源分配给序列中的下一个请求
    - 时限：时间最早优先：最早的挂起请求最早得到资源
  - 静态调度
    - 循环调度

回顾：可用性一般场景



刺激源

内部、外部

刺激

(错误)忽略、崩溃、时间(太早、太迟)、响应(用不正确的值)

制品

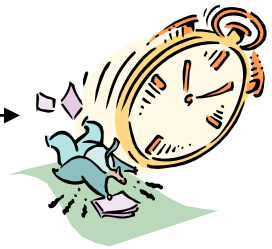
极高可用性的资源：进程、存储、处理器、通讯

环境

正常、降级操作

响应

记录(故障)、通知(用户或系统)、禁止、继续(正常/降级)、不可用(修复期间)



响应度量

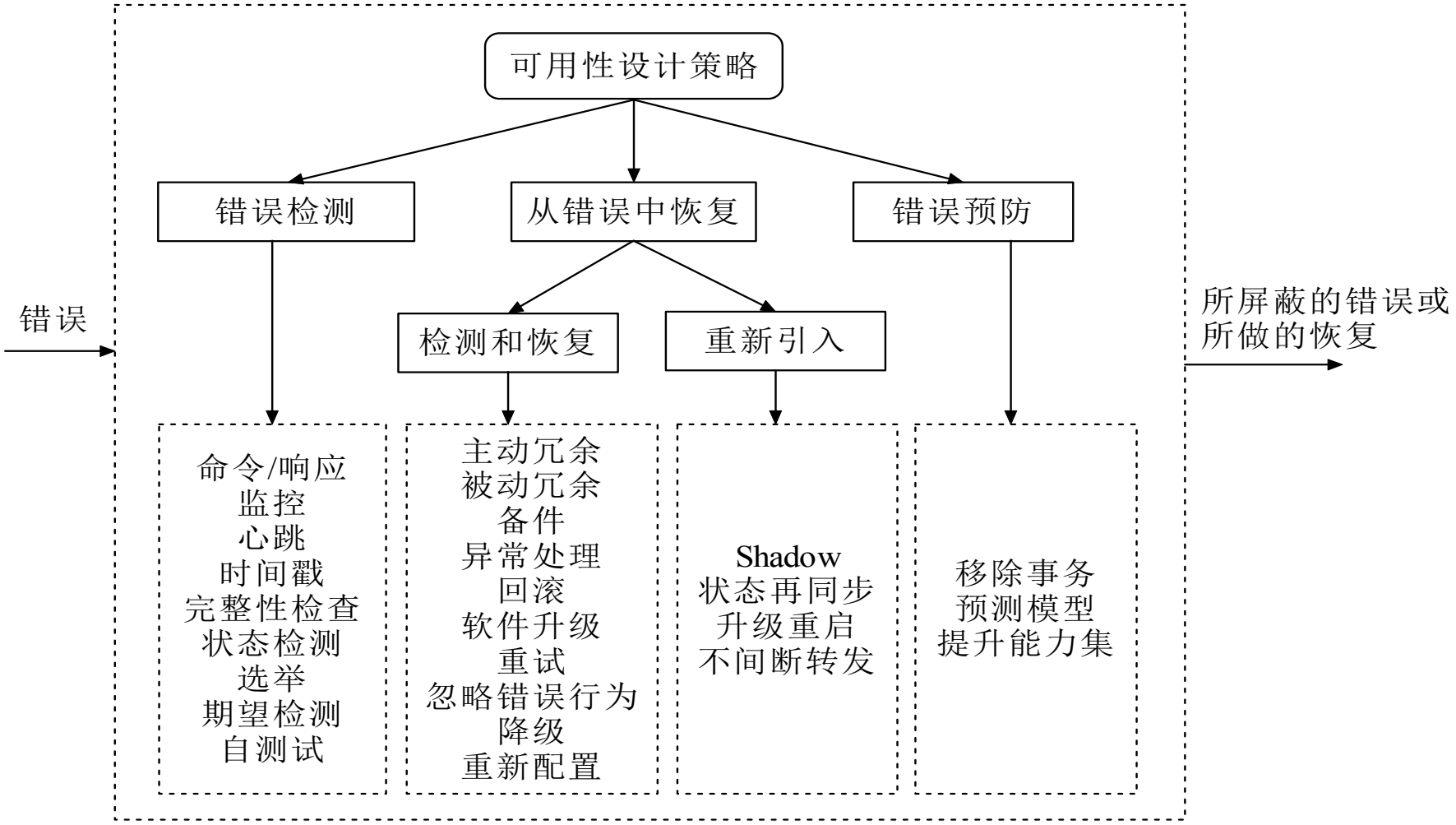
修复时间、可用性、可获得/降级的间隔时间

可用性的一般场景的可能取值

可用性设计策略（1/4）

- 目标
  - 恢复或者修复
  - 阻止错误发展成故障，或者把错误影响限制在一定范围之内，使得修复成为可能

可用性设计策略（1/4）


可用性设计策略
错误检测
从错误中恢复
错误预防
检测和恢复
重新引入
错误
所屏蔽的错误或
所做的恢复
命令/响应
监控
心跳
时间戳
完整性检查
状态检测
选举
期望检测
自测试
主动冗余
被动冗余
备件
异常处理
回滚
软件升级
重试
忽略错误行为
降级
重新配置
Shadow
状态再同步
升级重启
不间断转发
移除事务
预测模型
提升能力集

可用性设计策略（2/4）

- 错误检测
  - 命令/响应
    - 一个构件发出一个命令，测试在预定时间内收到一个来自审查构件的响应
    - 客户端可以采用该策略，以确保服务器对象和到服务器的路径在期望的性能边界内
  - 心跳（dead man计时器）
    - 一个构件定期发出一个心跳消息，另一个构件收听该消息，如果心跳失败通知错误处理构件
    - 自动柜员机可以定期向服务器发送上一次交易的日志，起到心跳和传送数据双重作用
  - 异常
    - 设置异常处理程序

可用性设计策略（3/4）

- 错误恢复：检测和修复
  - 表决（冗余）
    - 运行在冗余处理器上的每个进程都具有相等的输入，它们计算发送给表决者一个简单的输出值，如果表决者检测到某一处理器的异常行为，那么就中止这一行为
    - 用于纠正算法的错误操作或处理器的故障，通常用在控制系统中
  - 主动冗余（热重启）
    - 所有的冗余构件都以并行的方式对事件作出响应，仅使用一个构件的响应，丢弃其它响应
    - 当错误发生时，系统停机时间就是切换时间，仅为几毫秒
  - 被动冗余（暖重启/双冗余/三冗余）
    - 一个主要构件对事件作出响应，并通知其它备用构件进行状态更新
    - 当错误发生时，在继续提供服务之前，要保证备用状态是最新的

可用性设计策略（3/4）

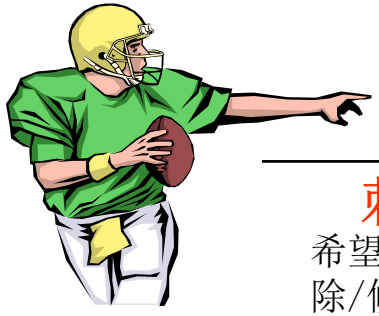
- 错误恢复：重新引入
  - Shadow操作
    - 设置一个构件短时间内以“shadow模式”运行，以确保在恢复该构件之前，模仿工作构件的行为
  - 状态再同步
    - 主动和被动冗余要求所恢复的构件在重新提供服务前要更新其状态
    - 更新方法取决于可以承受的停机时间、更新的模式以及更新所要求的消息的数量
  - 检查点/回滚
    - 记录系统的一致状态，或者是定期收集状态数据，或者是对具体事件做出响应

可用性设计策略（4/4）

- 错误预防
  - 从服务中删除
    - 从操作中删除系统的一个构件，以执行某些活动来防止预期发生的故障
  - 事务
    - 绑定几个有序的步骤，以能够立刻撤销整个绑定
    - 如果进程中的一个步骤失败的话，可以使用事务来防止数据受到影响
  - 进程监视器
    - 一旦检测到进程中出现错误，监视进程就可以删除错误执行进程，并为该进程创建一个新的实例

回顾：可修改性一般场景

可修改性是有关变更的成本问题。它关注修改什么(制品)和何时以及谁来进行更改(环境)



刺激源

开发人员、系统管理人员、最终用户

刺激

希望增加/删除/修改/改变功能、质量属性、容量

制品

系统用户界面、平台、环境、与目标系统交互的系统

环境

在设计、构建、编译、运行时

响应

查找SA中修改的位置，进行修改且不会影响其它功能，测试所做修改，部署所做修改

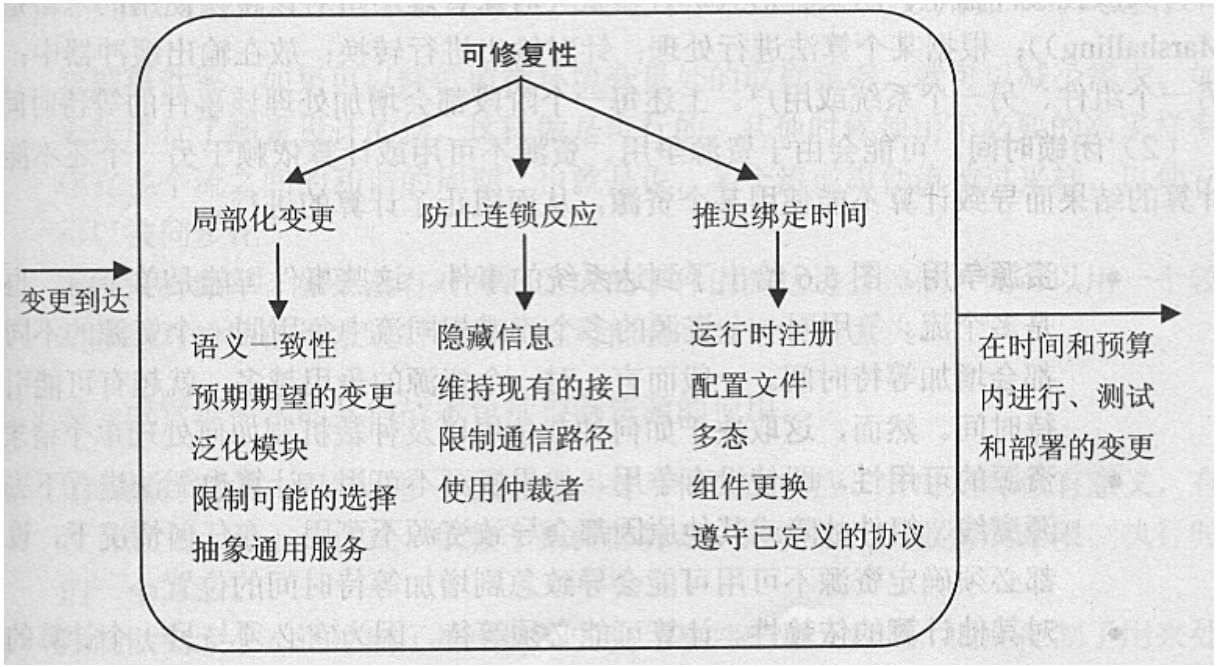
响应度量

根据所影响的元素的数量度量的成本、努力、资金；该修改对其它功能或质量属性所造成影响的程度

可修改性一般场景的可能取值

可修改性设计策略（1/7）

- 目标
  - 减少由某个变更直接影响的模块数量
  - 限制对局部化的模块的修改，防止连锁反应
  - 控制部署时间和成本
- 可修改性设计策略



可修改性设计策略（2/7）

- 局部化修改
  - 维持语义的一致性
    - 抽象通用服务，由专门的模块封装通用的服务，以提供复用，注重模块责任的一致性
    - 对通用服务的修改只需要进行一次，而不需要在使用这些服务的每个模块中进行修改，不仅支持局部化修改，还防止连锁反应
    - 应用框架和中间件的使用
  - 预期期望的变更
    - 不关心模块责任的一致性，只关心如何使变更的影响达到最小
    - 不可能预期所有的变更，通常结合语义一致性使用
  - 泛化模块
    - 使一个模块更通用，根据不同的输入提供不同的功能
    - 通过调整参数而不是修改模块来进行变更
  - 限制可能的选择
    - 修改的范围非常大，因此可能修改很多模块
    - 限制可能的选择会降低修改所造成的影响

可修改性设计策略（3/7）

- 防止连锁反应
  - 修改所产生的连锁反应就是需要改变该修改并没有直接影响到的模块
  - 依赖性：如果改变了模块A以完成某个特定的修改，那么必须改变模块B，成为模块B依赖于模块A
- 8种类型的依赖性
  - （1）语法
    - 数据：要使B正确编译或者执行，由A产生并由B使用的数据的类型必须与B所假定的数据类型一致
    - 服务：要使B正确编译或者执行，由A提供并且由B调用的服务的签名（signature）必须与B假定的一致
  - （2）语义
    - 数据：要使B正确执行，由A产生并由B使用的数据的语义必须与B所假定的数据类型一致
    - 服务：要使B正确执行，由A提供并且由B调用的服务的语义必须与B假定的一致
  - （3）顺序
    - 数据：要使B正确执行，必须以一个固定的顺序接受由A产生的数据
    - 控制：要使B正确执行，A必须在一定时间限制内执行

续：可修改性设计策略（4/7）

- （4）A的一个接口的身份
  - A可以有多个接口，要使B正确编译和执行，该接口的身份必须与B假定的一样
- （5）A的位置
  - 要使B正确执行，A的运行时位置必须与B假定的一样，例如A位于某个特定服务器的特定进程中
- （6）A提供的服务/数据的质量
  - 要使B正确执行，涉及A所提供的数据或服务的一些质量属性必须与B假定的一样
- （7）A的存在
  - 要使B正确执行，A必须存在
- （8）A的资源行为
  - 要使B正确执行，A的资源行为必须与B的假定一致

续：可修改性设计策略（5/7）

- 信息隐蔽
  - 把某个实体的责任分解为更小的部分，并选择哪些信息是公有的，哪些信息是私有的
  - 目的是将变更隔离在一个模块内，防止扩散
- 维持现有的接口
  - 添加接口
    - 通过添加新的接口提供新增的可见服务和数据，从而使得现有的接口不变并提供相同的signature
  - 添加适配器
    - 添加一个封装A的适配器，并提供原始signature
  - 提供一个占位程序A
    - 如果需要删除A，但是B仅依赖A的signature，那么为A提供一个占位程序能够使B保持不变
- 限制通信路径
  - 减少使用由该模块所产生的数据的模块的数量，以及产生由该模块所使用的数据的模块的数量，以减少连锁反应

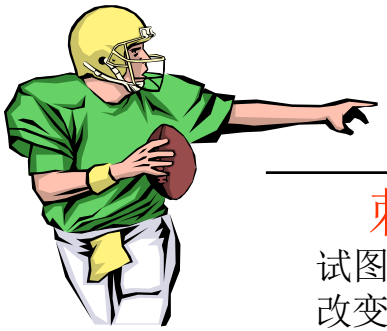
可修改性设计策略（6/7）

- 仲裁者的使用：如果B对A有非语义的任何形式的依赖，那么可以引入一个仲裁者（Arbiter）来管理与该依赖相关的活动
  - 数据（语法）
    - 数据存储构件可以充当数据生产者和使用之间的仲裁者，将A产生的语法转换为B要求的语法
  - 服务（语法）
    - Façade、Bridge、Proxy、Mediator、Strategy、Factory Method等模式都提供了把服务的语法从一种形式转换为另一种形式的仲裁者，可以防止A的变化扩散到B
  - A的接口的身份
    - 可以使用经纪人（Broker）模式屏蔽一个接口的身分的变化
  - A的位置（运行时）
    - Name服务器能够使A的位置发生变化时，不影响到B
  - A的资源行为或由A控制的资源（运行时）
    - 资源管理器是一个负责进行资源分配的仲裁者
  - A的存在
    - 工厂模式能够根据需要来创建实例，解决B对A的存在依赖

可修改性设计策略（7/7）

- 推迟绑定时间
  - 运行时注册
    - 支持即插即用操作（插件），但需要管理注册的额外开销
  - 配置文件
    - 在启动时设置参数
  - 多态
    - 允许方法调用的后期邦定
  - 构件更换
    - 允许载入时间绑定
  - 遵守已定义的协议
    - 允许独立进程的运行时绑定

回顾：安全性一般场景



刺激源

正确识别、非正确识别或身份未知的个人或系统(来自内或外部)；经过了授权/未经授权而访问了有限/大量的资源

刺激

试图显示数据、改变/删除数据、访问系统服务、降低系统服务的可用性

制品

系统服务，系统中断的数据

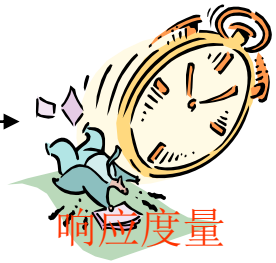
环境

在线/离线、联网/断网、连接有防火墙/直接连接到网络

对用户进行身份验证；隐藏用户身份；阻止/允许对数据和/或服务的访问；授权或收回对数据和/或服务的许可；

响应

根据身份记录访问/修改或试图访问/修改数据或服务；以一种不可读的格式存储数据；识别无法解释的对服务的高需求；通知用户或另外一个系统并限制服务的可用性

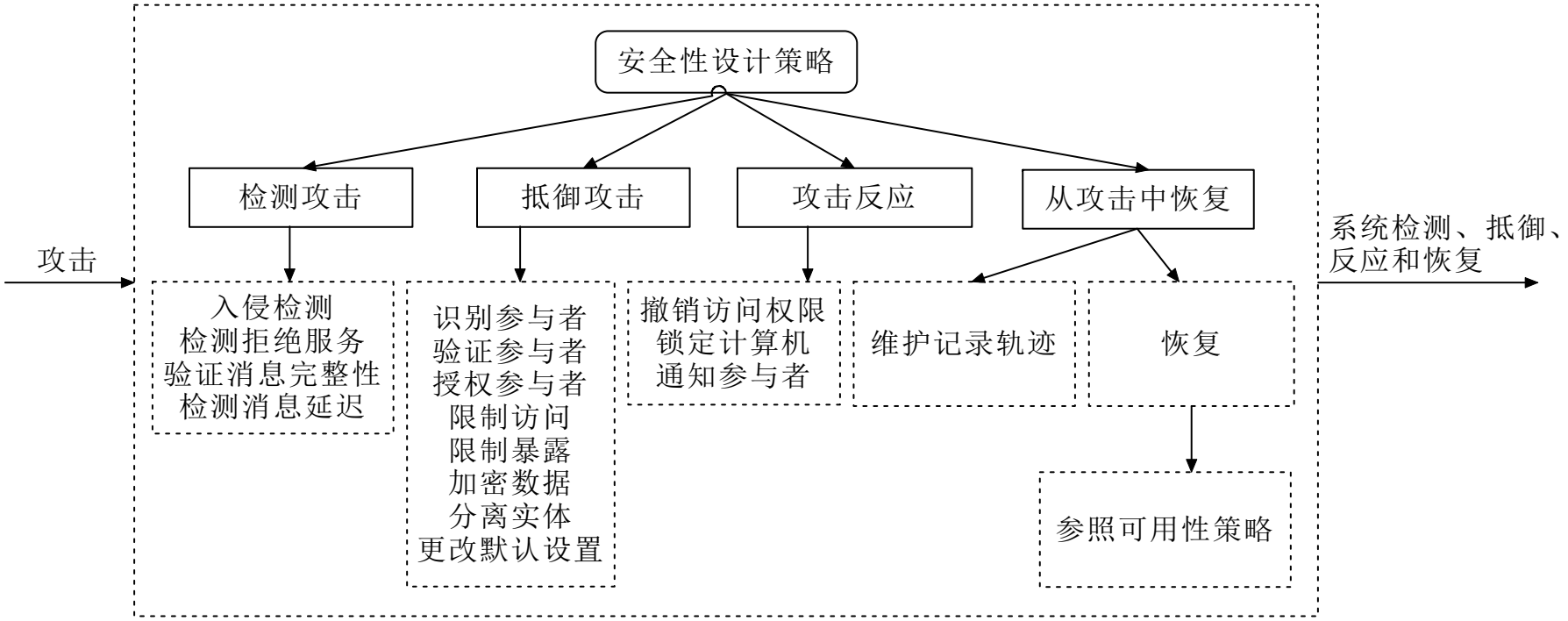


响应度量

用成功的概率表示、避免安全防范措施所需要的时间/努力/资源；检测到攻击的可能性；确定攻击或访问/修改数据和/或服务的个人的可能性；在拒绝服务攻击的情况下仍然可以获得服务的百分比；恢复数据服务；被破坏的数据/服务和/或被拒绝的合法访问的范围

安全性一般场景的可能取值

安全性设计策略（1/3）


安全性设计策略
检测攻击
抵御攻击
攻击反应
从攻击中恢复
攻击
系统检测、抵御、
反应和恢复
入侵检测
检测拒绝服务
验证消息完整性
检测消息延迟
识别参与者
验证参与者
授权参与者
限制访问
限制暴露
加密数据
分离实体
更改默认设置
撤销访问权限
锁定计算机
通知参与者
维护记录轨迹
恢复
参照可用性策略

安全性设计策略（1/3）

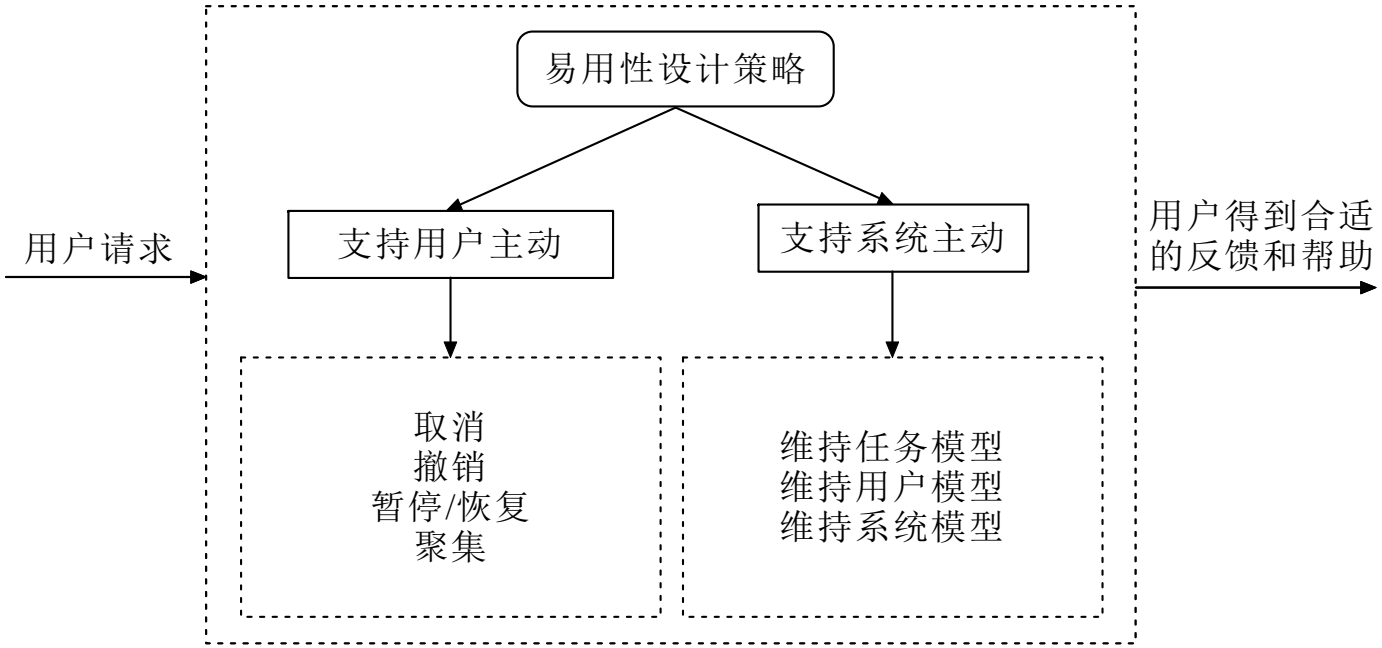
- **抵抗攻击**
  - 身份认证
    - 保证进行访问的用户或远程计算机确实是它所声称的用户或计算机
    - 密码（含验证码）、数字证书、生物识别（高铁站刷脸）
  - 用户授权
    - 保证经过了身份认证的用户有权访问和修改数据或者服务
  - 维护数据的机密性
    - 对数据进行保护，以防止未经授权的访问
    - 对数据和通信线路进行加密保护
  - 维护完整性
    - 对数据中的冗余信息，例如校验和哈希值，进行加密
  - 限制访问
    - 来自未知源的消息可能是某种形式的攻击
    - 防火墙根据消息源和目的地端口来限制访问（校图书馆电子资源）

安全性设计策略（1/3）

- **检测攻击**
  - 通过“入侵检测”系统进行，将通信模式和已知攻击的历史模式进行比较
  - 可以根据数据包中的协议、TCP标记、有效负荷大小、源或目的地地址以及端口号等进行监测
- **从攻击中恢复**
  - 恢复状态
    - 可以采用可用性设计策略，都是从不一致的状态恢复到一致状态，特别要注意维护系统管理数据的冗余副本，如密码、访问控制列表、域名服务和用户资料数据等
  - 识别攻击者
    - 使用审计信息来追踪攻击者的操作
    - 审计信息包括系统中数据的变更、事务的执行和身份识别等信息

易用性设计策略

- 目标
  - 易用性关注的是用户完成期望任务的难易程度以及系统能够提供给用户的支持。
  - 易用性的场景包括用户主动和系统主动两个方面。
- 易用性设计策略



支持用户主动

- 支持用户主动：一旦系统开始执行，向用户反馈当前系统状态并让用户做出合适的响应能够增加易用性。例如，以下策略（取消、撤销、暂停/恢复和聚集）能够支持用户修正错误或是提高效率。
  - 取消：中断所取消的命令、释放所使用的资源、通知与其进行…
  - 撤销：为了使系统支持撤销操作，系统必须能够记录足够多的系统状态。记录可以是状态“快照”的形式（如检查点），或者是一组可逆的操作，然而并非所有的操作都是可逆的？
  - 暂停/恢复：当一个用户启动一个长时间的操作，如下载大文件
  - 聚集：当一个用户正在执行重复的操作，或者该操作对许多对象产生了相同的影响时，将低级别的对象聚集到一个小组中，然后将操作应用到整个小组，就可以使得用户不必不断重复地完成一项操作

支持系统主动

- 当系统采取主动时，它必须依赖于用户模型、任务模型或者系统本身的状态模型。
  - 维持任务模型：任务模型用来判断上下文，使得系统能够知道用户正在做什么并对用户提供帮助。
  - 维持用户模型：该模型显式地表达了用户的一些基本信息，用户行为的期望响应时间，以及用户的一些其他方面的因素。例如，用户模型能使系统对自动提供给用户的提示信息的数量进行控制。
  - 维持系统模型：系统自身维持一个模型，该模型用来决定期望的系统行为，使其能向用户传递合适的反馈信息。一个系统模型的常见示例是进度条，它能够预测完成当前活动所需的剩余时间。

支持易用性的其他策略

- 将用户接口与应用的其余部分分离开来
  - 在开发和部署中，预计用户接口会频繁发生变化，因此单独维护用户接口代码会把变更局部化
  - 使用MVC、MVP模式
  - 使用插件模式（最大程度上允许用户改变界面配置）
  - …