# 软件体系结构原理、方法与实践



新的开始，新的起点，让我们一起为梦想而努力。

## 模式 – 概述

**模式:给定上下文中普遍问题的普遍解决方案.**

高层模式:

体系结构模式:反应了开发软件系统过程中所作的基本设计决策。

低层模式:

设计模式:关注软件的系统设计,与具体的实现语言无关。

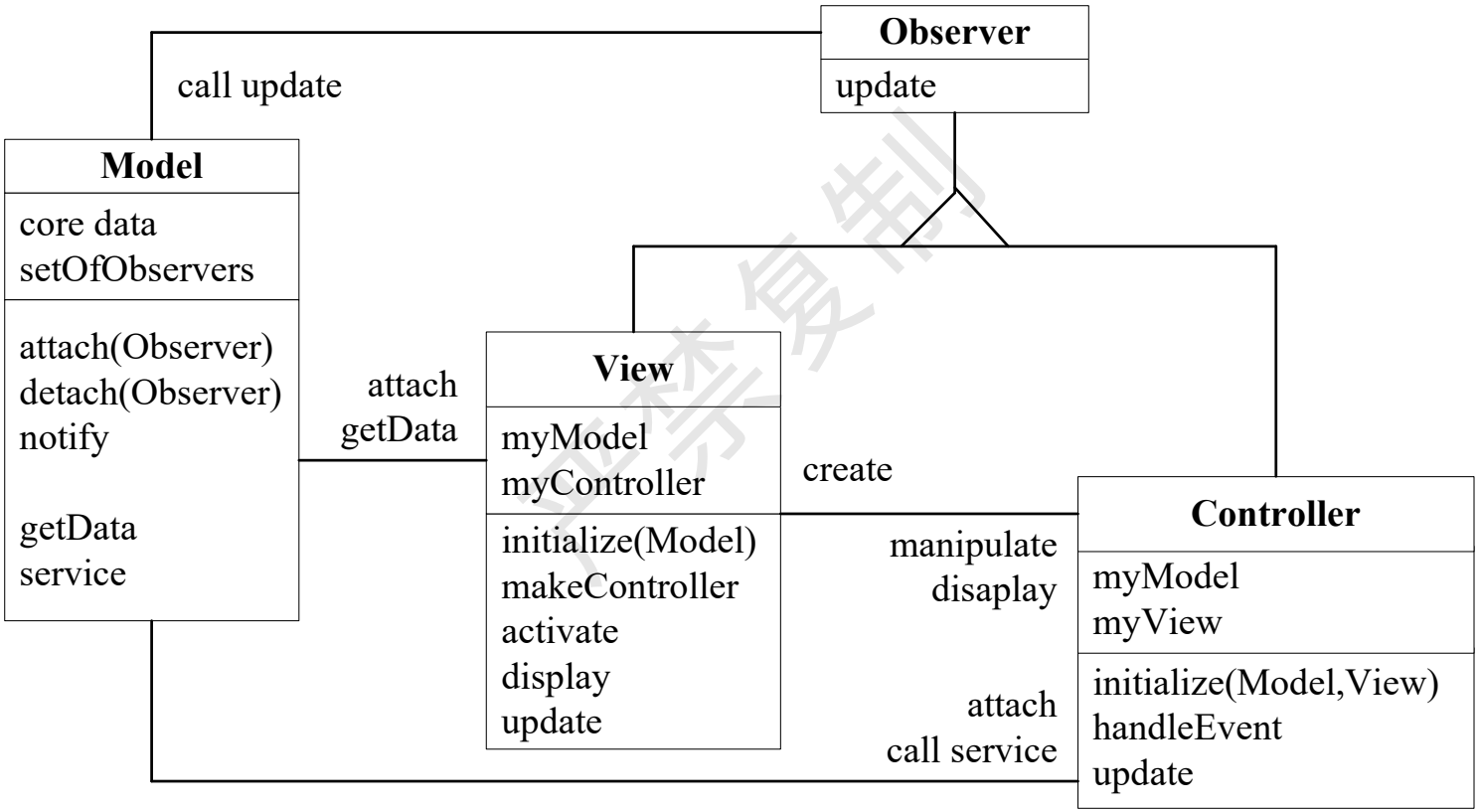
- 设计模式是一套被反复使用、多数人知晓的、经过分类编目的代码设计经验总结。使用设计模式的目的是提高代码重用性,使代码更容易被人理解,保证代码可靠性。

- 惯用法:实现时通过某种特定的程序设计语言来描述构件与构件之间的关系(如:引用,计数器就是C++中的一种惯用法)

## 设计模式 – 概述

- 模式是指从某个具体的形式中得到的一种抽象，在特殊的非任意性的环境中，该形式不断地重复出现。
- 一个软件体系结构的模式描述了一个出现在特定设计语境中的特殊的再现设计问题，并为它的解决方案提供了一个经过充分验证的通用图示。解决方案图示通过描述其组成构件及其责任和相互关系以及它们的协作方式来具体指定。

# 设计模式 – MVC模式



## 设计模式 – 好的设计模式具备的特征

- **解决一个问题**：从模式可以得到解，而不仅仅是抽象的原则或策略。
- **是一个被证明了的概念**：模式通过一个记录得到解．而不是通过理论或推测。
- **解并不是显然的**：许多解决问题的方法（例如软件设计范例或方法）是从最基本的原理得到解；而最好的方法是以非直接的方式得到解，对大多数比较困难的设计问题来说，这是必要的。
- **描述了一种关系**：模式并不仅仅描述模块，它给出更深层的系统结构和机理。
- **模式有重要的人为因素**：所有的软件服务于人类的舒适或生活质量，而最好的模式追求它的实用性和美学。

## 设计模式 – 基本成分

- 模式名称
- 问题
- 解决方案
- 后果

## 设计模式 – 描述

从以下几个方面描述

(1) 模式名称和分类
(2) 目的
(3) 别名
(4) 动机
(5) 应用
(6) 结构
(7) 成分
(8) 合作
(9) 后果
(10) 实现
(11) 例程代码
(12) 已知的应用
(13) 相关模式

# 设计模式 – 模式与体系结构

- 模式作为体系结构构造块

在开发软件时，模式是处理受限的特定设计方面的有用构造块。因此，对软件体系结构而言，模式的一个重要目标就是用已定义属性进行特定的软件体系结构的构造。例如，MVC模式提供了一个结构，用于交互应用程序的用户界面的裁剪。

- 构造异构体系结构

单个模式不能完成一个完整的软件体系结构的详细构造,它仅帮助设计师设计应用程序的某一方面。然而,即使正确设计了这个方面,整个体系结构仍然可能达不到期望的所有属性。为“整体上”达到软件体系结构的需求,需要一套丰富的涵盖许多不同设计问题的模式。可获得的模式越多,能够被适当解决的设计问题也会越多,并且可以更有力地支持构造带有已定义属性的软件体系结构。

# 设计模式 – 模式与体系结构

- 模式和方法

好的模式描述也包含它的实现指南，可将其看成是一种微方法(micro-method),用来创建解决一个特定问题的方案。通过提供方法的步骤来解决软件开发中的具体再现问题，这些微方法补充了通用的但与问题无关的分析和设计方法。

- 实现模式

目前的许多软件模式具有独特的面向对象风格。所以,人们往往认为,能够有效实现模式的唯一方式是使用面向对象编程语言,其实不然，例如：在C语言中实现策略模式可以通过采用函数指针来代替多态性和继承性。

## 设计模式 – 层次

- Coad的面向对象模式
- 代码模式
- 框架应用模式
- 形式合约

## 设计模式 – 面向对象模式

面向对象模式分为三类：

- ✓ **基本的继承和交互模式**：主要包括OOPL所提供的基本建模功能，继承模式声明了一个类能够在其子类中被修改或被补充，**交互模式**描述了在有多个类的情况下消息的传递。
- ✓ **面向对象软件系统的结构化模式**：描述了在适当情况下，一组类如何支持面向对象软件系统结构的建模。
- ✓ **与MVC框架相关的模式**：描述如何构造面向对象软件系统，描述MVC框架的各个方面，有助于构件的设计。

## 设计模式 – 代码模式

代码模式的抽象方式与面向对象程序设计语言(OOPL)中的代码规范很相似，该类模式有助于解决某种面向对象程序设计语言中的特定问题。主要目标在于：

- ✓ 指明结合基本语言概念的可用方式；
- ✓ 构成源码结构与命名规范的基础；
- ✓ 避免面向对象程序设计语言（尤其是C++语言）的缺陷。

代码模式与具体的程序设计语言或者类库有关，它们主要从语法的角度对于软件系统的结构方面提供一些基本的规范。这些模式对于类的设计不适用，同时也不支持程序员开发和应用框架，命名规范是类库中的名字标准化的基本方法，以免在使用类库时产生混淆。

## 设计模式 – 框架应用模式

- 在应用程序框架“菜谱”中有很多“菜谱条”，它们用一种不很规范的方式描述了如何应用框架来解决特定的问题。程序员将框架作为应用程序开发的基础，特定的框架适用于特定的需求。“菜谱条”通常并不讲解框架的内部设计实现，只讲如何使用。
- 不同的框架有各自的“菜谱”。

## 设计模式 – 形式合约

形式合约也是一种描述框架设计的方法，强调组成框架的对象间的交互关系。有人认为它是面向交互的设计，对其他方法的发展有启迪作用。但形式化方法由于其过于抽象，而有很大的局限性，仅仅在小规模程序中使用。

优点：

（1）符号所包含的元素很少，并且其中引入的概念能够被映射成为面向对象程序设计语言中的概念。例如，参与者映射成为对象。

（2）形式合约中考虑到了复杂行为是由简单行为组成的事实，合约的修订和扩充操作使得这种方法很灵活，易于应用。

缺点：

（1）在某些情况下很难用，过于繁琐。若引入新的符号，则又使符号系统复杂化。

（2）强制性地要求过分精密，从而在说明中可能发生隐患（例如冗余）。

（3）形式合约的抽象程度过低，接近面向对象的程序设计语言，不易分清主次。

## 设计模式 – 分类

- ✓ 创建型模式
- ✓ 结构型模式
- ✓ 行为型模式

- ✓ 工厂方法(factory method)模式
- ✓ 抽象工厂(abstract factory)模式
- ✓ 原型(prototype)模式
- ✓ 单例(singleton)模式
- ✓ 构建器(builder)模式

- ✓ 适配器(adapter)模式
- ✓ 桥接(bridge)模式
- ✓ 组合(composite)模式
- ✓ 装饰(decorator)模式
- ✓ 外观(facade)模式
- ✓ 享元(flyweight)模式
- ✓ 代理(proxy)模式

- ✓ 职责链(chain of responsibility)模式
- ✓ 命令(command)模式
- ✓ 解释器(interpreter)模式
- ✓ 迭代器(iterator)模式
- ✓ 中介者(mediator)模式
- ✓ 备忘录(memento)模式
- ✓ 观察者(observer)模式
- ✓ 状态(state)模式
- ✓ 策略(strategy)模式
- ✓ 模板方法(template method)模式
- ✓ 访问者(visitor)模式

有下划线的表示既可以是类模式，也可以是对象模式；无下划线的表示只是对象模式





新的开始，新的起点，让我们一起为梦想而努力。

## 设计模式 – 分类 – 创建型模式

| 设计模式名称 | 简要说明 | 可改变的方面 |
| --- | --- | --- |
| Abstract Factory<br>抽象工厂模式 | 提供一个接口，可以创建一系列相关或相互依赖的对象，而无需指定它们具体的类 | 产品对象族 |
| Builder<br>构建器模式 | 将一个复杂类的表示与其构造相分离，使得相同的构建过程能够得出不同的表示 | 如何建立一种组合对象 |
| Factory Method*<br>工厂方法模式 | 定义一个创建对象的接口，但由子类决定需要实例化哪一个类。 工厂方法使得子类实例化的过程推迟 | 实例化子类的对象 |
| Prototype<br>原型模式 | 用原型实例指定创建对象的类型，并且通过拷贝这个原型来创建新的对象 | 实例化类的对象 |
| Singleton<br>单例模式 | 保证一个类只有一个实例，并提供一个访问它的全局访问点 | 类的单个实例 |

## 设计模式 – 分类 – 结构型模式

| 设计模式名称 | 简要说明 | 可改变的方面 |
| --- | --- | --- |
| Adapter<br>适配器模式 | 将一个类的接口转换成用户希望得到的另一种接口。它使原本不相容的接口得以协同工作 | 与对象的接口 |
| Bridge<br>桥接模式 | 将类的抽象部分和它的实现部分分离开来，使它们可以独立地变化 | 对象的实现 |
| Composite<br>组合模式 | 将对象组合成树型结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性 | 对象的结构和组合 |
| Decorator<br>装饰模式 | 动态地给一个对象添加一些额外的职责。它提供了用子类扩展功能的一个灵活的替代，比派生一个子类更加灵活 | 无子类对象的责任 |
| Façade<br>外观模式 | 定义一个高层接口，为子系统中的一组接口提供一个一致的外观，从而简化了该子系统的使用 | 与子系统的接口 |
| Flyweight<br>享元模式 | 提供支持大量细粒度对象共享的有效方法 | 对象的存储代价 |
| Proxy<br>代理模式 | 为其他对象提供一种代理以控制这个对象的访问 | 如何访问对象，对象位置 |

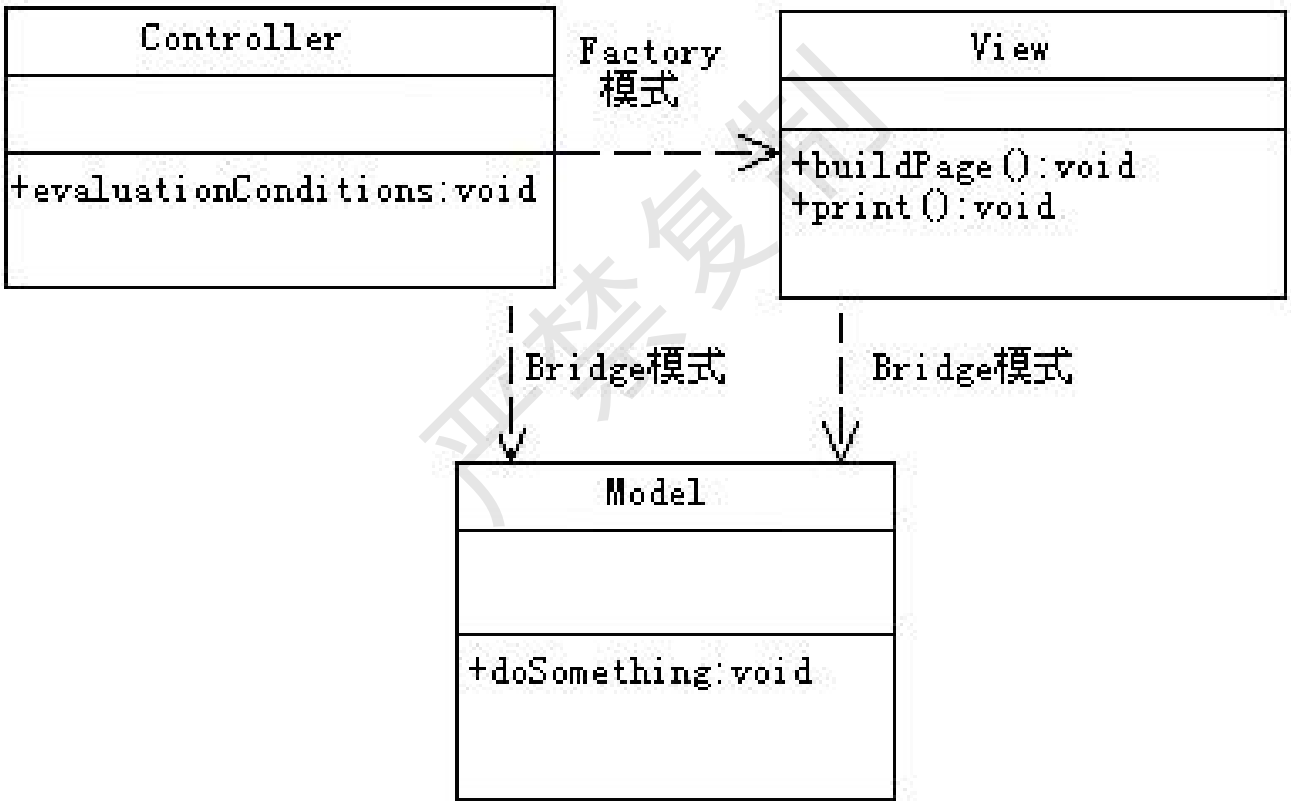
# 设计模式 – 分类 – 行为型模式(1)

| 设计模式名称 | 简要说明 | 可改变的方面 |
| --- | --- | --- |
| Chain of Responsibility<br>职责链模式 | 通过给多个对象处理请求的机会，减少请求的发送者与接收者之间的耦合。将接收对象链接起来，在链中传递请求，直到有一个对象处理这个请求 | 可满足请求的对象 |
| Command<br>命令模式 | 将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，将请求排队或记录请求日志，支持可撤销的操作 | 何时及如何满足一个请求 |
| Interpreter*<br>解释器模式 | 给定一种语言，定义它的文法表示，并定义一个解释器，该解释器用来根据文法表示来解释语言中的句子 | 语言的语法和解释 |
| Iterator<br>迭代器模式 | 提供一种方法来顺序访问一个聚合对象中的各个元素，而不需要暴露该对象的内部表示 | 如何访问、遍历聚合的元素 |
| Mediator<br>中介者模式 | 用一个中介对象来封装一系列的对象交互。它使各对象不需要显式地相互调用，从而达到低耦合，还可以独立地改变对象间的交互 | 对象之间如何交互及哪些对象交互 |

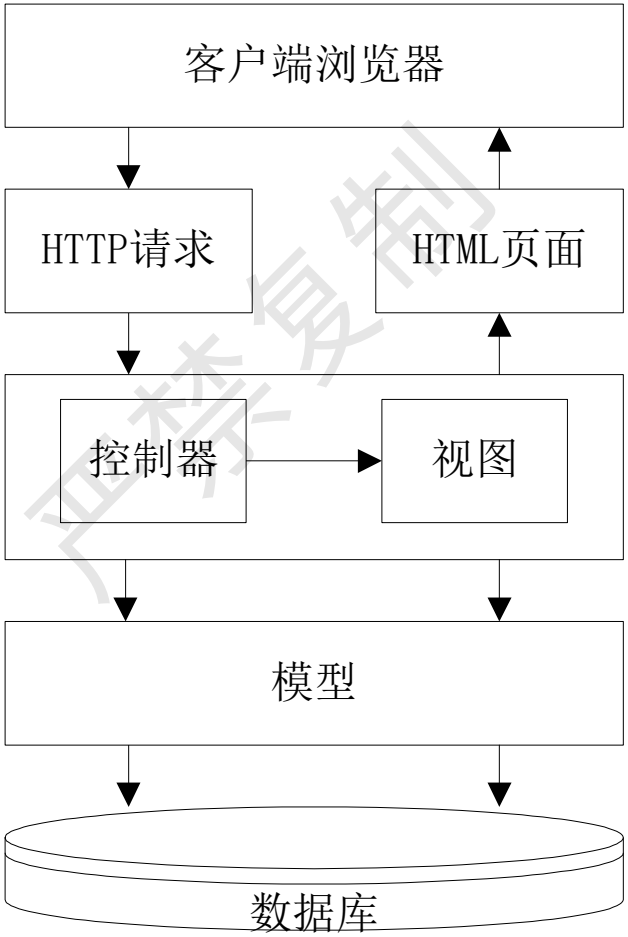
# 设计模式 – 分类 – 行为型模式(2)

| 设计模式名称 | 简要说明 | 可改变的方面 |
| --- | --- | --- |
| Memento<br>备忘录模式 | 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，从而可以在以后将该对象恢复到原先保存的状态 | 何时及哪些私有信息存储在对象之外 |
| Observer<br>观察者模式 | 定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新 | 依赖于另一对象的数量 |
| State<br>状态模式 | 允许一个对象在其内部状态改变时改变它的行为 | 对象的状态 |
| Strategy<br>策略模式 | 定义一系列算法，把它们一个个封装起来，并且使它们之间可互相替换，从而让算法可以独立于使用它的用户而变化 | 算法 |
| Template Method*<br>模板方法模式 | 定义一个操作中的算法骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重新定义算法的某些特定步骤 | 算法的步骤 |
| Visitor<br>访问者模式 | 表示一个作用于某对象结构中的各元素的操作，使得在不改变各元素的类的前提下定义作用于这些元素的新操作 | 无需改变其类而可应用于对象的操作 |

## 设计模式 – MVC模式的设计与实现 – 框架



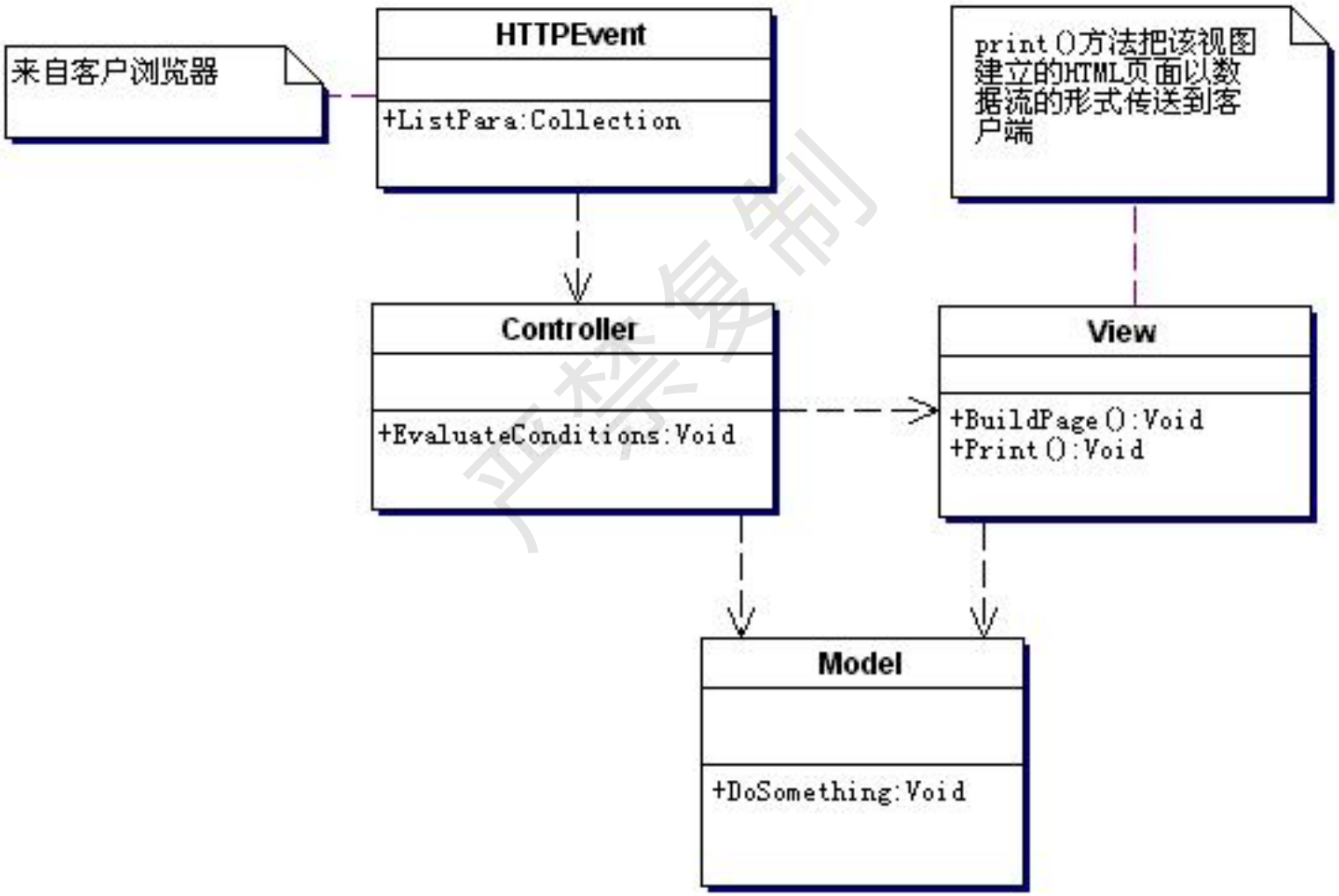
## 设计模式 – MVC模式的设计与实现 – 应用





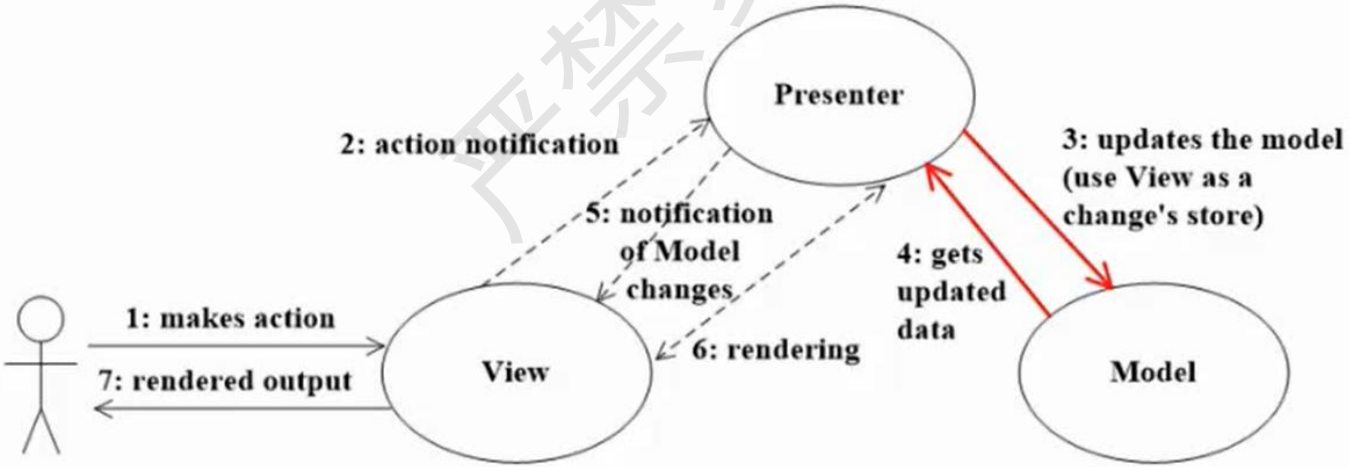

新的开始，新的起点，让我们一起为梦想而努力。

## 设计模式 – MVC模式的设计与实现 – 类图



# 设计模式 – MVP模式

- MVP是MVC的变种
- MVP实现了V与M之间的解耦（V不直接使用M，修改V不会影响M）
- MVP更好的支持单元测试（业务逻辑在P中，可以脱离V来测试这些逻辑；可以将一个P用于多个V，而不需要改变P的逻辑）
- MVP中V要处理界面事件，业务逻辑在P中，MVC中界面事件由C处理



设置

# 关于设计模式，下列说法正确是:

A 一个好的设计模式需要能够解决一个问题而不仅仅是抽象的原则和策略。

B 一个模式通常由模式名称，要解决的问题，解决方案和实施计划组成

C 典型的设计模式层次有：面向对象模式，代码模式，框架应用模式

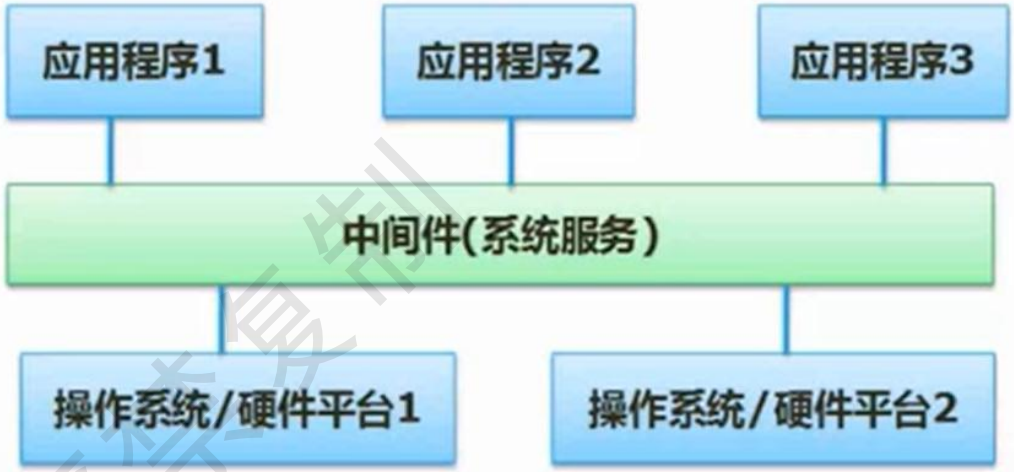
D 设计模式可分为：创建型模式，结构性模式，行为型模式，工厂型模式。



提交

# 中间件技术 – 中间件的功能

中间件是一种独立的系统软件或服务程序，可以帮助分布式应用软件在不同的技术之间共享资源



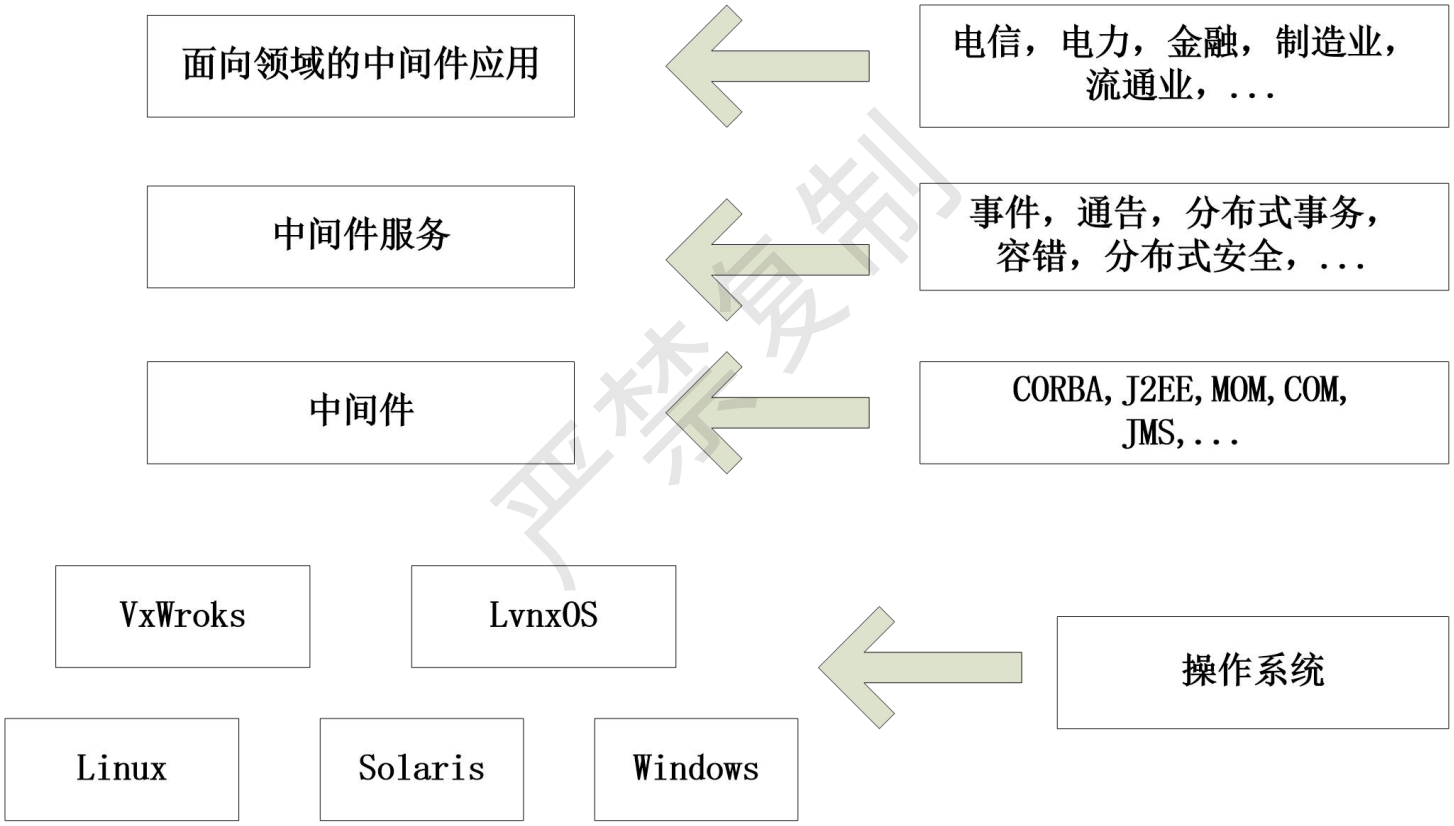

- 负责客户机与服务器之间的连接和通信，以及客户机与应用层之间的高效率通信机制
- 提供应用层不同服务之间的互操作机制，以及应用层与数据库之间的连接和控制机制
- 提供多层架构的应用开发和运行的平台，以及应用开发框架，支持模块化的应用开发
- 屏蔽硬件、操作系统、网络和数据库的差异
- 提供应用的负载均衡和高可用性、安全机制与管理功能，以及交易管理机制，保证交易的一致性
- 提供一组通用的服务去执行不同的功能，避免重复的工作和使应用之间可以协作

# 中间件技术 – 中间件的分类

- 底层中间件：JVM（Java Virtual Machine，Java虚拟机）、CLR（Common Language Runtime，公共语言运行库）、ACE（Adaptive Communication Environment，自适配通信环境）等
- 通用型中间件：也称为平台，其主流技术主要有RPC、ORB、MOM（Message-Oriented Middleware，面向消息的中间件）等
- 集成型中间件：WorkFlow、EAI等

## 中间件技术 – 中间件的应用

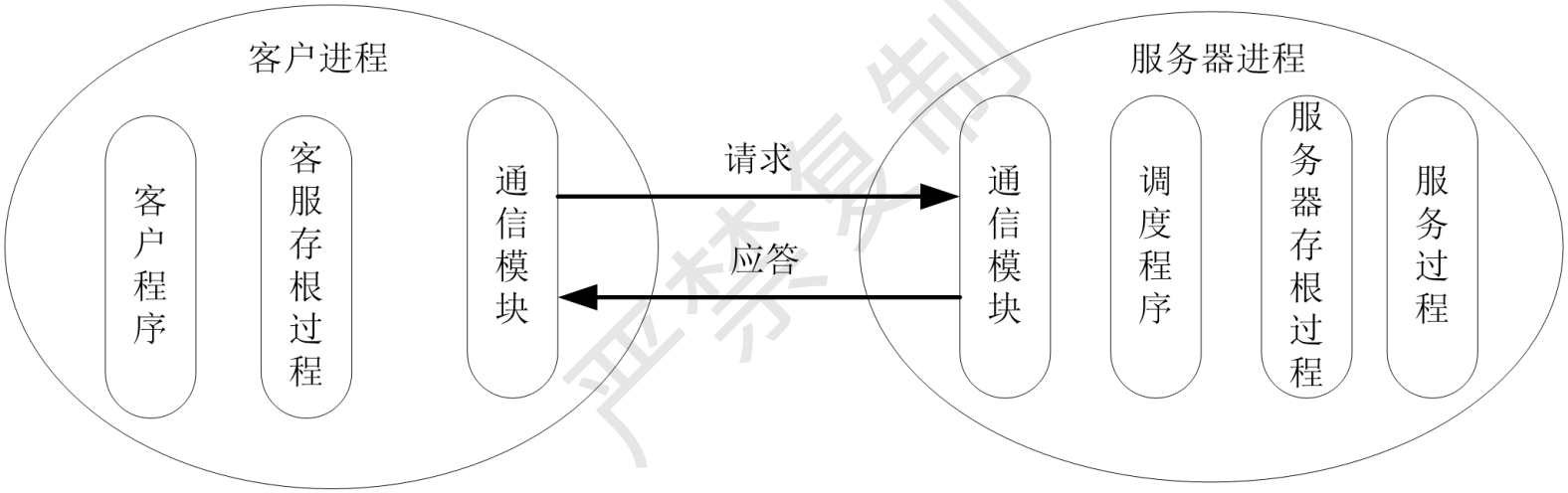
# 中间件技术 – 中间件的应用



## 中间件技术 – 中间件的发展趋势

- **规范化。**规范的建立极大地促进了中间件技术的发展，同时保证了系统的扩展性、开放性和互操作性。
- **构件化和松耦合。**基于XML和Web Service的中间件技术，使得不同系统之间、不同应用之间的交互建立在非常灵活的基础上。
- **平台化。**一些大的中间件厂商在已有的中间件产品基础上，提出了完整的面向互联网的软件平台战略计划和应用解决方案。

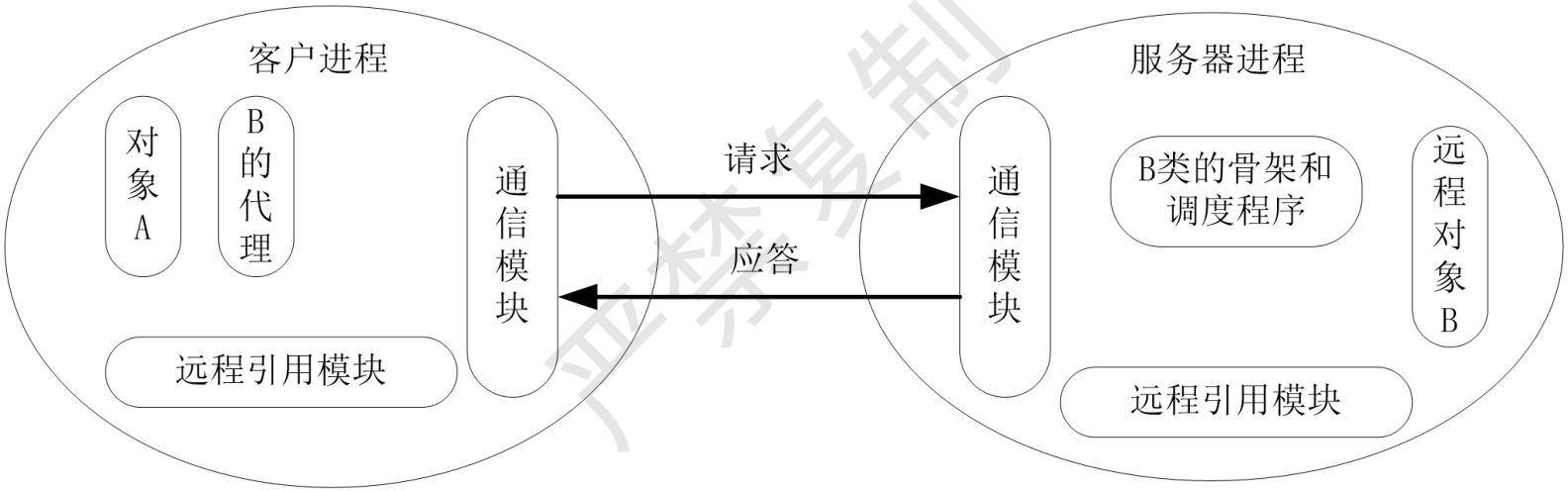
## 中间件技术 – 主要的中间件 – 远程过程调用



## 中间件技术 – 主要的中间件 – 对象请求代理（ORB）

- ORB在CORBA规范中处于核心地位，定义异构环境下对象透明地发送请求和接收响应的基本机制，是建立对象之间C/S关系的中间件。
- ORB使得对象可以透明地向其他对象发出请求或接受其他对象的响应，这些对象既可以位于本地，也可以位于远程机器。
- ORB拦截请求调用，并负责找到可以实现请求的对象、传送参数、调用相应的方法、返回结果等。客户对象并不知道与服务器对象通讯、激活或存储服务器对象的机制，也不必知道服务器对象位于何处、它是用何种语言实现的、使用什么操作系统或其他不属于对象接口的系统成分。

## 中间件技术 – 主要的中间件 – 远程方法调用



## 中间件技术 – 主要的中间件 – 面向消息的中间件

- 通信程序可在不同的时间运行。程序不在网络上直接相互通信，而是间接地将消息放入消息队列。
- 对应用程序的结构没有约束。在复杂的应用场合中，通信程序之间不仅可以是一对一的关系，还可以进行一对多和多对一方式。
- 程序与网络复杂性相隔离。程序将消息放入消息队列或从消息队列中取出消息来进行通信，与此关联的全部活动，例如，维护消息队列、维护程序和队列之间的关系、处理网络的重新启动和在网络中移动消息等是MOM的任务，程序不直接与其他程序通信，并且它们不涉及网络通信的复杂性。

## 中间件技术 – 主要的中间件 – 事务处理监控器

- 进程管理，包括启动服务器进程、为其分配任务、监控其执行并对负载进行平衡。
- 事务管理，即保证在其监控下的事务处理的原子性、一致性、独立性和持久性。
- 通信管理，为客户和服务器之间提供多种通信机制，包括请求/响应、会话、排队、订阅/发布和广播等。

# 中间件技术 – 中间件与构件的关系

## 面向需求

基于体系结构的构件化软件开发应当是面向需求的，即设计师集中精力于业务逻辑本身，而不必为分布式应用中的非功能质量属性耗费大量的精力，理想的体系结构在这些方面应当为软件提供良好的运行环境。事实上，这些正是中间件所要解决的问题，因此，基于中间件开发的应用真正是面向需求的，从本质上符合构件化设计的思想。

## 业务的分隔和包容性

构件要求有很好的业务自包容性，应用开发人员可以按照不同的业务进行功能的划分，体现为不同的接口或交互模式。针对每种业务，设计和开发是可以独立进行的。在提供业务的分隔和包容性方面，体系结构和中间件有同样的目标。

# 中间件技术 – 中间件与构件的关系

- **设计与实现隔离**

构件对外发生作用或构件间的交互，都是通过接口进行的，构件使用者只需要知道构件的接口，而不必关心其内部实现，这是设计与实现分离的关键。中间件在分布交互模式上也规定了接口(或类似)机制

- **隔离复杂的系统资源**

软件体系结构很重要的一个功能就是将系统资源与应用构件隔离，这是保证构件可复用甚至“即插即用”的基础，与中间件的意图也是一致的。中间件最大的优势之一就是屏蔽多样的系统资源，保证良好的互操作性。应用构件开发人员只需要按照中间件规定的模式进行设计开发，而不必考虑下层的系统平台。因此，中间件真正提供了与环境隔离的构件开发模式。

# 中间件技术 – 中间件与构件的关系

- **符合标准的交互模型**

软件体系结构是一种抽象的模型，但模型中应当定义一些可操作的成分。例如，标准的协议等。中间件则实现了体系结构的模型，实现了标准的协议。基于中间件的构件是符合标准模型的。

- **软件复用**

软件复用是构件化软件开发的根本目标之一，中间件提供了构件封装、交互规则、与环境的隔离等机制，这些都为软件复用提供了方便的解决方案。

# 中间件技术 – 中间件与构件的关系

- 提供对应构件的管理

  基于中间件的软件可以方便地进行管理,因为构件总可以通过标识机制进行划分。

基于中间件开发的应用是构件化的，中间件提供了构件的软件体系结构，大大提高了应用构件开发的效率和质量。

中间件作为应用软件系统集成的关键技术，保证了构件化思想的实施，并为构件提供了真正的运行空间。

构件对新一代中间件产品也起到了促进作用，构件化的中间件在市场上具有强大的生命力。

设置

# 下列关于中间件的描述正确的是：

A 中间件是 一种独立的系统软件或服务程序

B 中间件采用自低向上可分为底层中间件，通用型中间件，集成型中间件

C 从某种程度上说中间件是构件存在的基础，中间件促进了构件化的实现

D RMI应用程序通常包括服务器和客户端两个独立的部分

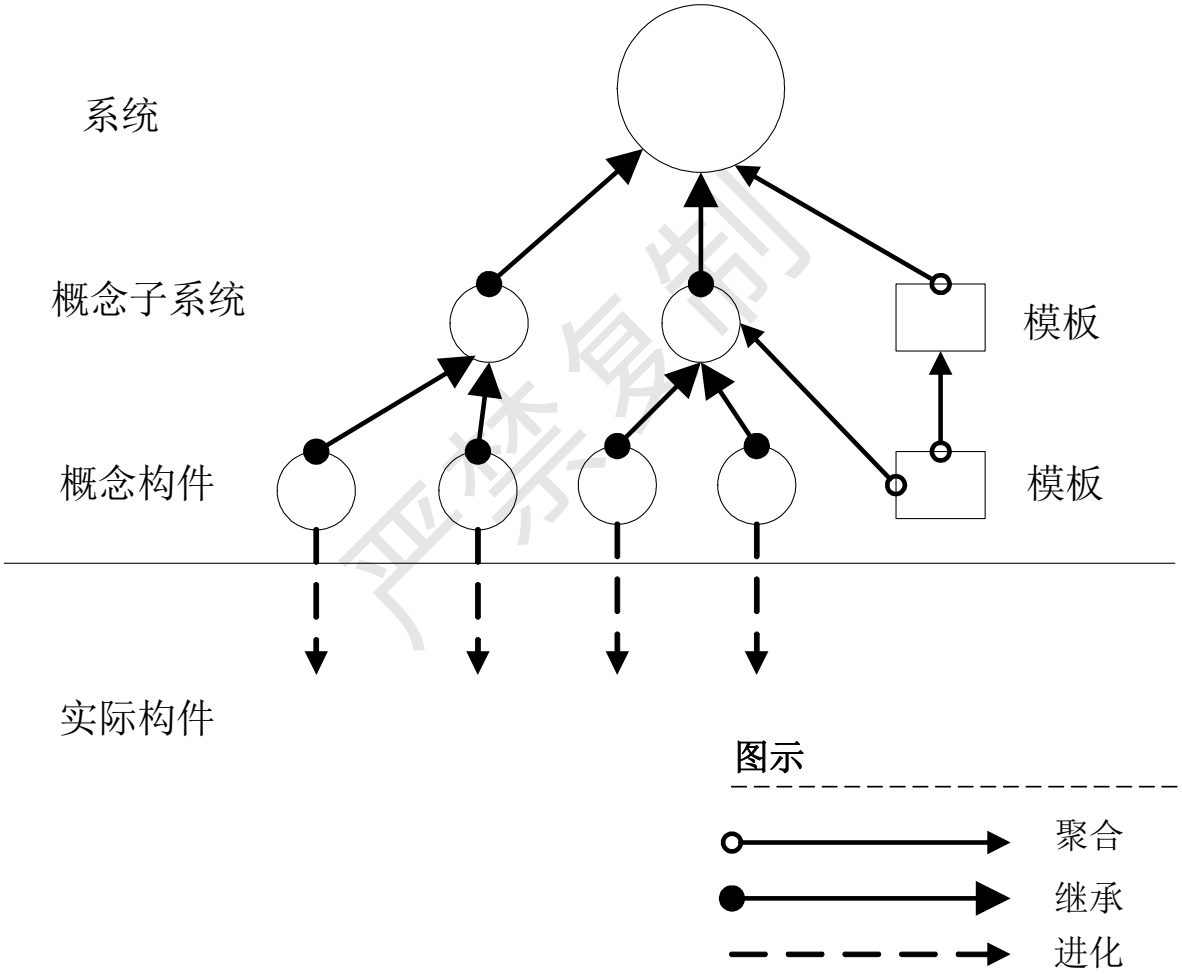


提交

# ABSD方法 – 基本概念

- ABSD方法为产生软件系统的概念体系结构提供构造，概念体系结构描述了系统的主要设计元素及其关系。概念体系结构代表了在开发过程中作出的第一个选择，相应地，它是达到系统质量和商业目标的关键，为达到预定功能提供了一个基础。
- 体系结构驱动，是指构成体系结构的**商业、质量和功能需求**的组合。
- 使用ABSD方法，设计活动可以在体系结构驱动一决定就开始，这意味着需求抽取和分析还没有完成，就开始了软件设计。设计活动的开始并不意味着需求抽取和分析活动就可以终止，而是应该与设计活动并行。特别是在不可能预先决定所有需求时，例如产品线系统或长期运行的系统，快速开始设计是至关重要的。
- 软件模板是一个特殊类型的软件元素，包括描述所有这种类型的元素在共享服务和底层构造的基础上如何进行交互。软件模板还包括属于这种类型的所有元素的功能，这些功能的例子有：每个元素必须记录某些重大事件，每个元素必须为运行期间的外部诊断提供测试点等。

# ABSD方法 – 设计元素



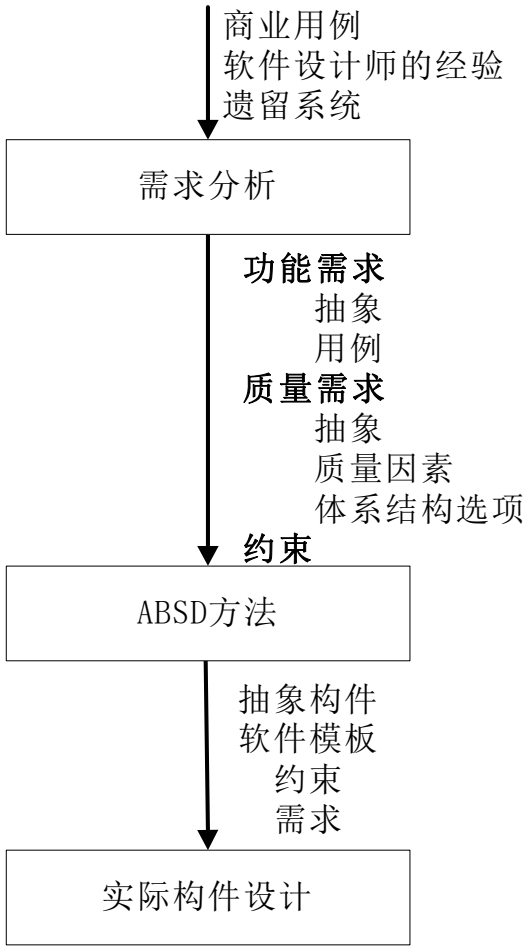



新的开始，新的起点，让我们一起为梦想而努力。

## ABSD方法 – 质量场景

- 在使用用例捕获功能需求的同时，我们通过定义特定场景来捕获质量需求，并称这些场景为质量场景。这样一来，在一般的软件开发过程中，我们使用质量场景捕获变更、性能、可靠性和交互性，分别称之为**变更场景**、**性能场景**、**可靠性场景**和**交互性场景**。质量场景必须包括预期的和非预期的刺激。
- 例如，一个预期的性能场景是估计每年用户数量增加10%的影响，一个非预期的场景是估计每年用户数量增加100%的影响。非预期场景可能不能真正实现，但它们在决定设计的边界条件时很有用。

# ABSD方法 – ABSD方法与生命周期




ABSD方法与生命周期

ABSD方法的组成：

1、抽象功能需求，包括变化的需求和通用的需求。

2、用例(实际功能需求)

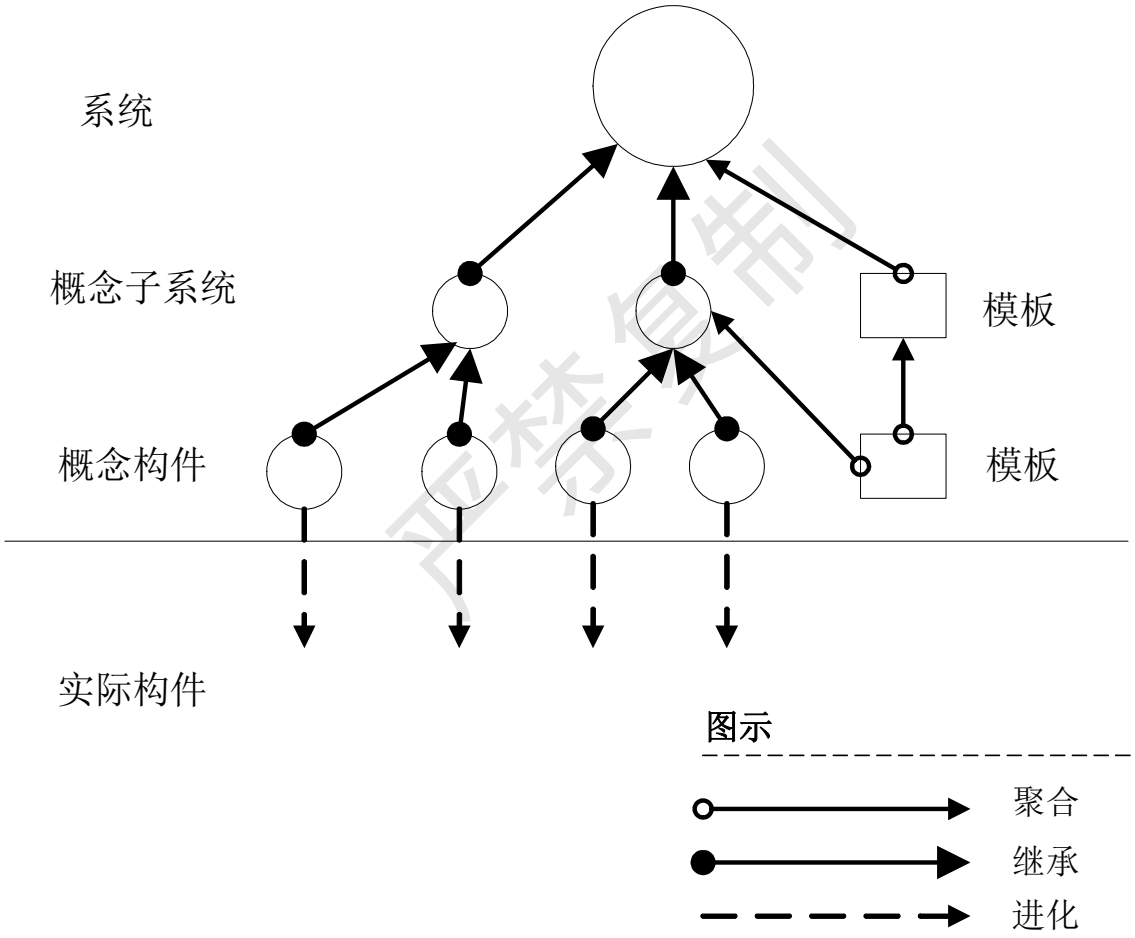
3、抽象的质量和业务需求

4、质量因素（实际质量和业务需求）

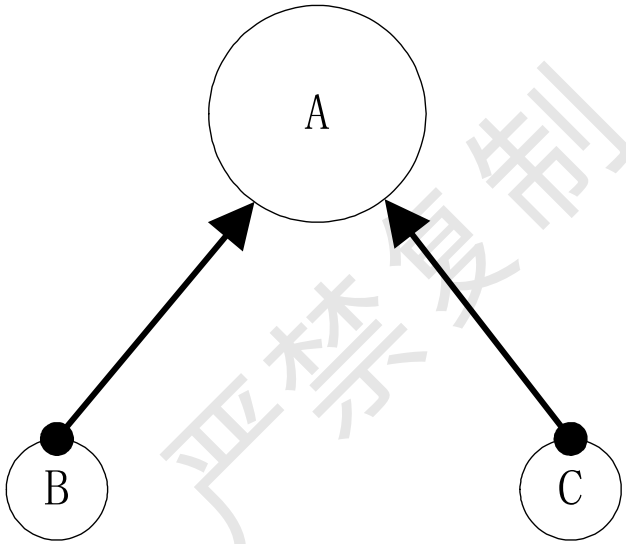
5、体系结构选项

6、约束

## ABSD设计方法 – 步骤 – ABSD方法定义的设计元素



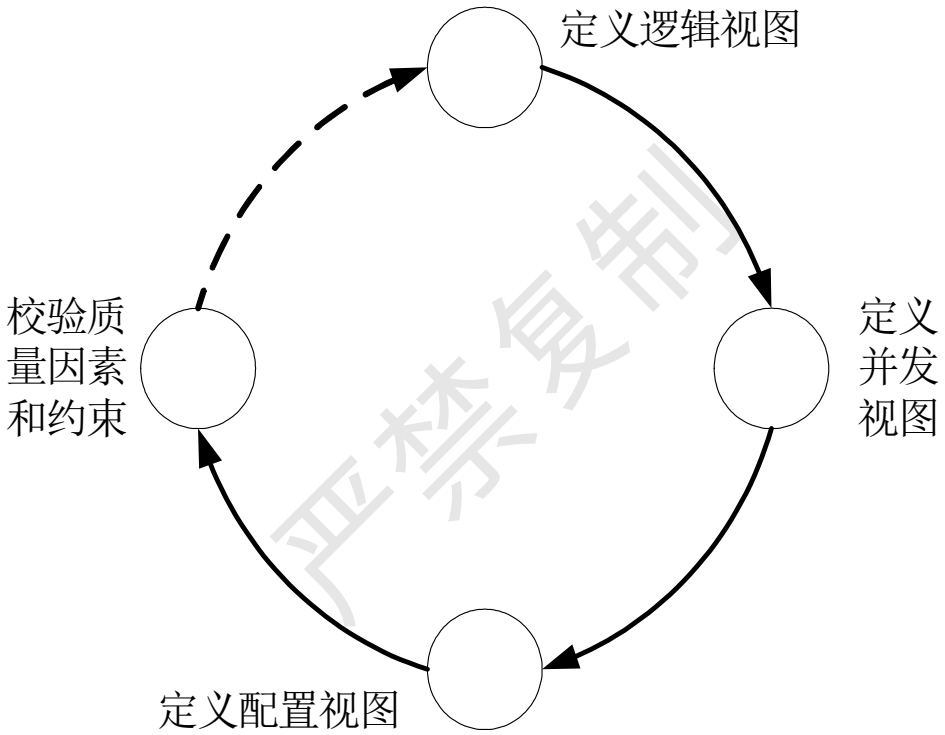
## ABSD设计方法 – 步骤 – 设计元素的产生顺序






设计元素A分解为两个小的设计元素B和C，元素A必须满足某些需求(功能、业务和性能)，在把A分解成B和C的过程中，A的功能也被分解成B和C的需求。

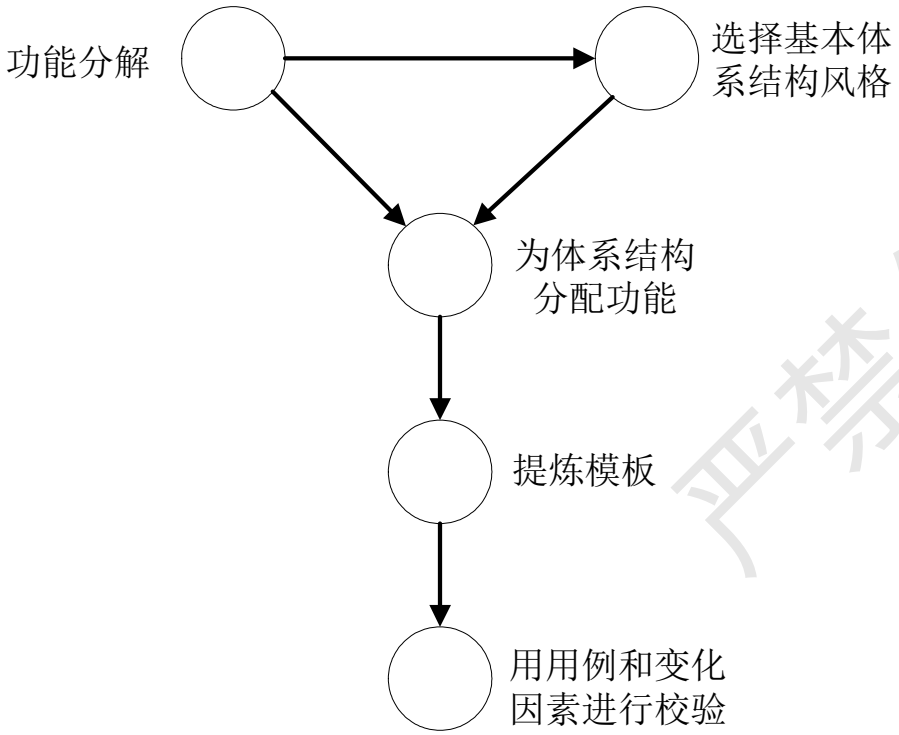
# ABSD设计方法 – 步骤 – 设计元素的活动

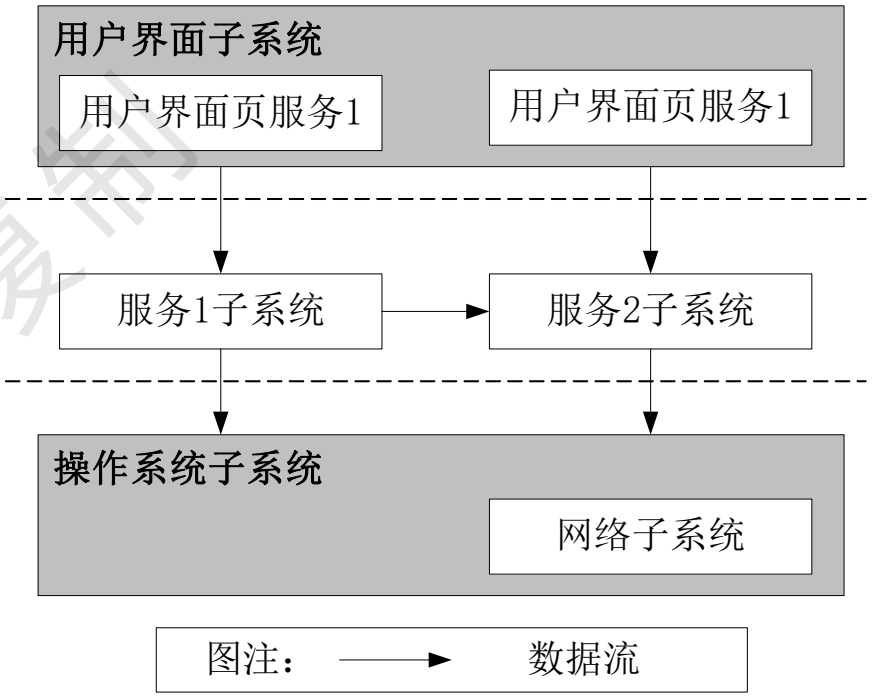





**分解一个设计元素的步骤**

# ABSD设计方法 – 步骤 – 设计元素的活动 – 定义逻辑视图



## ABSD设计方法 – 步骤 – 设计元素的活动 – 功能分解

- 一个设计元素有一组功能，这些功能必须分组。分解的目的是使每个组在体系结构内代表独立的元素。分解可以进一步细化。这种分解的标准取决于对一个特定的设计元素来说是很重要的性能。在不同的性能基础上，可以进行多重分解。
- 如果象通常的产品一样，在分解中起关键作用的性能要求是可修改的，则功能的分组可选择几个标准：
  （1）功能聚合。
  （2）数据或计算行为的类似模式。
  （3）类似的抽象级别。
  （4）功能的局部性。

## ABSD设计方法 – 步骤 – 设计元素的活动 – 功能分解

- 一个设计元素有一组功能，这些功能必须分组。分解的目的是使每个组在体系结构内代表独立的元素。分解可以进一步细化。这种分解的标准取决于对一个特定的设计元素来说是很重要的性能。在不同的性能基础上，可以进行多重分解。
- 如果象通常的产品一样，在分解中起关键作用的性能要求是可修改的，则功能的分组可选择几个标准：
  （1）功能聚合。
  （2）数据或计算行为的类似模式。
  （3）类似的抽象级别。
  （4）功能的局部性。

## ABSD设计方法 – 步骤 – 设计元素的活动 – 功能分解标准

(1)功能聚合。需求分组必须遵守“高内聚低耦合”的原则，这是对功能分解的一个标准技术。用例能够用来检查内聚和耦合，用来处理变化的性能因素也可用来检查内聚和耦合。

(2)数据或计算行为的类似模式。在数据或计算行为上，如果有类似模式的功能，则应该分在同一组。这意味着展示类似行为取决于使用系统的特定领域。如果数据获取是一个功能，则一个类似模式可能是样本周期。如果一个特定功能是需要很大计算量的，则它应该与其他计算量的功能分在一组。用同一模式存取数据库的功能也应该分在一组。

(3)类似的抽象级别。与硬件相近的功能不应该与那些抽象级别较高的功能分在一组。另一方面，处在同一抽象级别的功能应该分在一组。

(4)功能的局部性。那些为其他服务提供服务的功能不应该与纯局部功能分在一组。

## ABSD设计方法 – 步骤 – 设计元素的活动 – 选择体系结构风格

- 每个设计元素有一个主要的体系结构风格或模式，这是设计元素如何完成它的功能的基础。主要风格并不是唯一风格，为了达到特定目的，可以进行修改。
- 体系结构风格的选择建立在设计元素的体系结构驱动基础上。
- 在软件设计过程中，并不总是有现成的体系结构风格可供选择为主要的体系结构风格。
- 一旦选定了一个主要的体系结构风格，该风格必须适应基于属于这个设计元素的质量需求，体系结构选择必须满足质量需求。
- 为设计元素选择体系结构风格是一个重要的选择，这种选择在很大程度上依赖于软件设计师的个人设计经验。

## ABSD设计方法 – 步骤 – 设计元素的活动 – 为风格分配功能

- 选择体系结构风格时产生了一组构件类型，我们必须决定这些类型的数量和每个类型的功能，这就是分配的目的。在功能分解时产生的功能组，应该分配给选择体系结构风格时产生的构件类型，这包括决定将存在多少个每个构件类型的实例，每个实例将完成什么功能。这样分配后产生的构件将作为设计元素分解的子设计元素。
- 每个设计元素的概念接口也必须得到标识，这个接口包含了设计元素所需的信息和在已经定义了的体系结构风格内的每个构件类型所需要的数据和控制流。

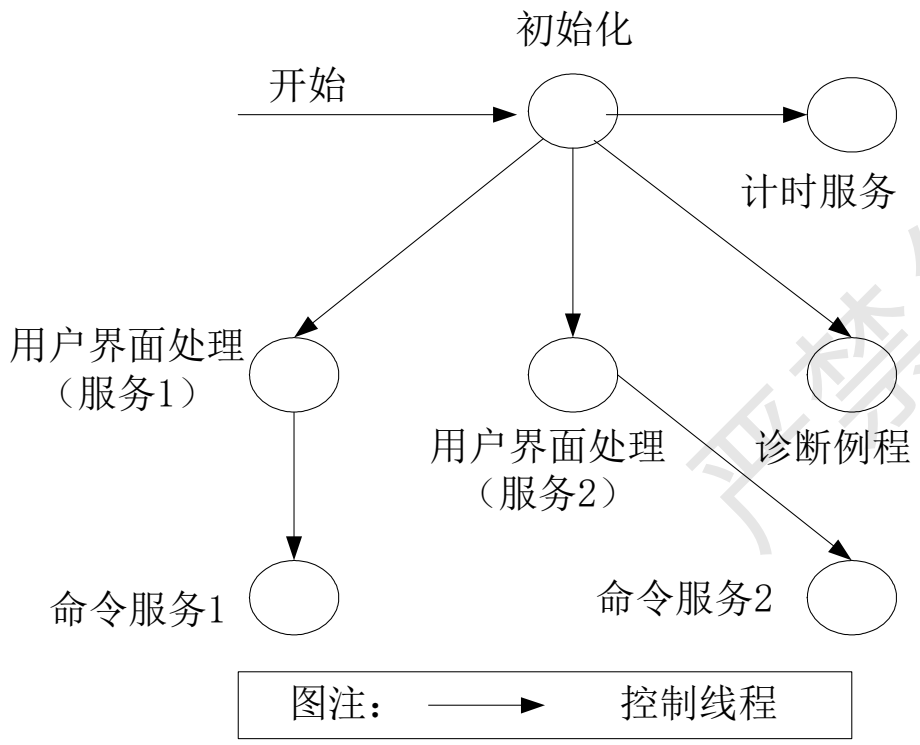
## ABSD设计方法 – 步骤 – 设计元素的活动 – 细化模板

- 被分解的设计元素有一组属于它的模板。在ABSD方法的初期，系统没有模板。当模板细化了以后，就要把功能增加上去。这些功能必须由实际构件在设计过程中加以实现。
- 最后，需要检查模板的功能，以判断是否需要增加附加功能到系统任何地方的设计元素中。也就是说，要识别在该级别上已经存在的任何横向服务。模板包括了什么是一个好的设计元素和那些应该共享的功能。每种类型的功能可以需要附加支持功能，这种附加功能一旦得到识别，就要进行分配。

## ABSD设计方法 – 步骤 – 设计元素的活动 – 功能校验

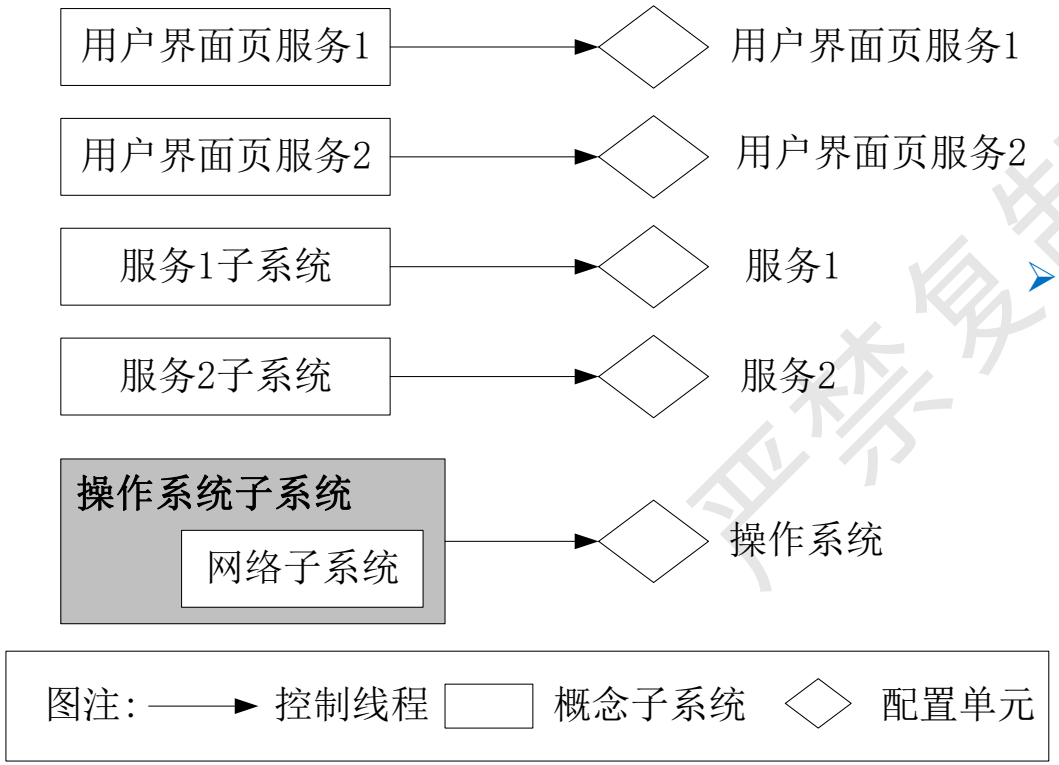
- 用例用来验证他们通过有目的的结构能够达到。子设计元素的附加功能将可能通过用例的使用得到判断。然而，如果用例被广泛地使用于功能分解的过程中，将几乎不能发现附加功能。
- 也可以使用变化因素，因为执行一个变化的难点取决于功能的分解。
- 从这种类型的校验出发，设计就是显示需求（通过用例）和支持修改（通过变化因素）。

# ABSD设计方法 – 步骤 – 设计元素的活动 – 创建并发视图




- 检查并发视图的目的是判断哪些活动是可以并发执行的。这些活动必须得到识别，产生进程同步和资源竞争。
- 对并发视图的检查是通过虚拟进程来实现的。虚拟进程是通过程序、动态模块或一些其他的控制流执行的一条单独路径。虚拟进程与操作系统的进程概念不一样，操作系统的进程包括了额外的地址空间的分配和调度策略。一个操作系统进程是几个虚拟进程的连接点，但每个虚拟进程不一定都是操作系统进程。虚拟进程用来描述活动序列，使同步或资源竞争可以在多个虚拟进程之间进行。

# ABSD设计方法 – 步骤 – 设计元素的活动 – 创建配置视图




- 如果在一个系统中，使用了多个处理器，则需要对不同的处理器配置设计元素，这种配置通过配置视图来进行检查。
- 例如，我们检查网络对虚拟线程的影响，一个虚拟线程可以通过网络从一个处理器传递到另一个处理器。我们使用物理线程来描述在某个特定处理器中的线程。也就是说，一个虚拟线程是由若干个物理线程串联而成的。通过这种视图，我们可以发现一个单一的处理器上的同步的物理线程和把一个虚拟线程从一个处理器传递到其他处理器上的需求。

## ABSD设计方法 – 步骤 – 设计元素的活动 – 验证质量场景

- 一旦创建了三个视图，就要把质量场景应用到所创建的子设计元素上。对每个质量场景，都要考虑是否仍然满足需求，每个质量场景包括了一个质量属性刺激和所期望的响应。考虑到目前为止所作出的设计决策，看其是否能够达到质量属性的要求。
- 如果不能达到，则需重新考虑设计决策，或者设计师必须接受创建质量场景失败的现实。

## ABSD设计方法 – 步骤 – 设计元素的活动 – 验证约束

最后一步就是要验证所有的约束没有互相矛盾的地方，对每一个约束，都需提问“该约束是否有可能实现？”。一个否定的回答就意味着对应的质量场景也不能满足。这时，需要把问题记录进文档，对导致约束的决策进行重新验证。

## 体系结构的设计与演化 – 设计与演化过程

- 实验原型阶段。这一阶段考虑的首要问题是**要获得对系统支持的问题域的理解**。为了达到这个目的，软件开发组织需要构建一系列原型，与实际的最终用户一起进行讨论和评审，这些原型应该演示和支持全局改进的实现。但是，来自用户的最终需求是很模糊的，因此，整个第一个阶段的作用是使最终系统更加精确化，有助于决定实际开发的可行性。
- 演化开发阶段。实验原型阶段的结果可以决定是否开始实现最终系统，如果可以，开发将进入第二个阶段。与实验原型阶段相比，**演化开发阶段的重点放在最终产品的开发上**。这时，原型即被当作系统的规格说明，又可当作系统的演示版本。**这意味着演化开发阶段的重点将转移到构件的精确化**。

## 体系结构的设计与演化 – 实验原型阶段 – 第一个开发周期

- 第一个开发周期没有具体的、明确的目标。此时，为了提高开发效率，缩短开发周期，所有开发人员可以分成了两个小组，一个小组创建图形用户界面，另一个小组创建一个问题域模型。两个小组要并行地工作，尽量不要发生相互牵制的现象。
- 在第一个周期结束时，形成了两个版本，一个是图形用户界面的初始设计，另一个是问题域模型，该模型覆盖了问题域的子集。用户界面设计由水平原型表示，也就是说，运行的程序只是实现一些用户界面控制，没有实现真正的系统功能。

## 体系结构的设计与演化 – 实验原型阶段 – 第二个开发周期

- 标识构件
- 提出软件体系结构模型
- 把已标识的构件映射到软件体系结构中
- 分析构件之间的相互作用
- 产生软件体系结构
- 软件体系结构正交化

## 体系结构的设计与演化 – 演化开发阶段(步骤)

- 需求变动归类
- 制订体系结构演化计划
- 修改、增加或删除构件
- 更新构件的相互作用
- 产生演化后的体系结构
- 迭代
- 对以上步骤进行确认，进行阶段性技术评审
- 对所做的标记进行处理

设置

# 以下关于体系结构的设计与演化说法正确的是

A 基于体系结构的开发过程可以分为两个独立的阶段即实验原型阶段和演化开发阶段

B 实验原型阶段的作用是使最终系统更加精确化，有助于决定实际开发的可行性

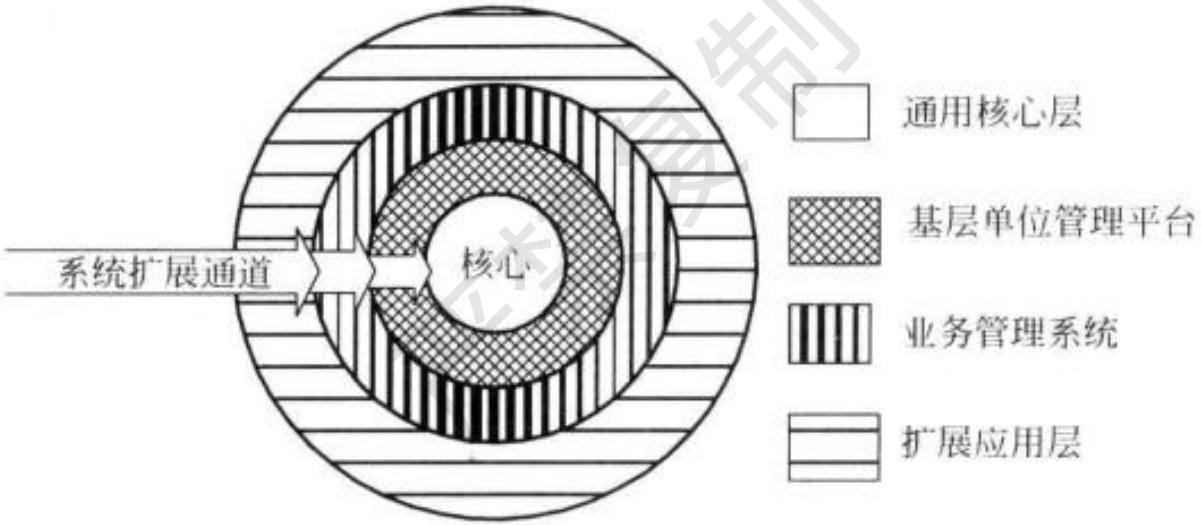
C 系统移植引起需求变化首先要对新需求进行归类

D 演化开发阶段的重点放在最终产品的开发上,以实现构件的精确化。



提交

## 应用开发实例-社会保险管理信息系统(SIMIS)



## 应用开发实例-SIMIS-通用核心层

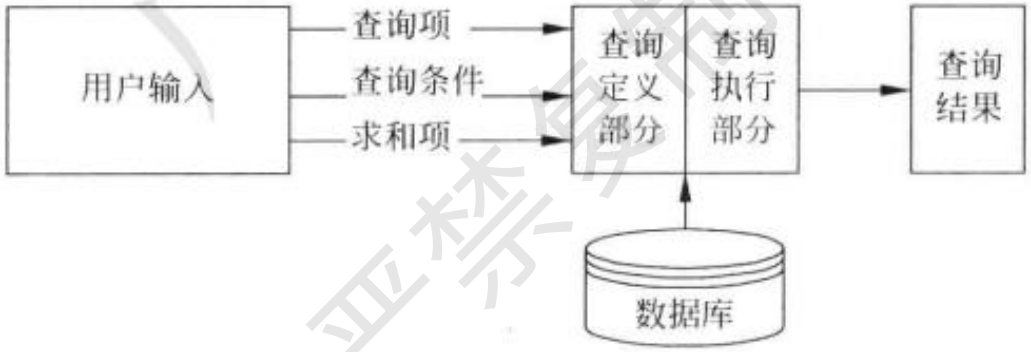



图 12-28 通用查询基类功能示意图

## 应用开发实例-SIMIS-通用核心层

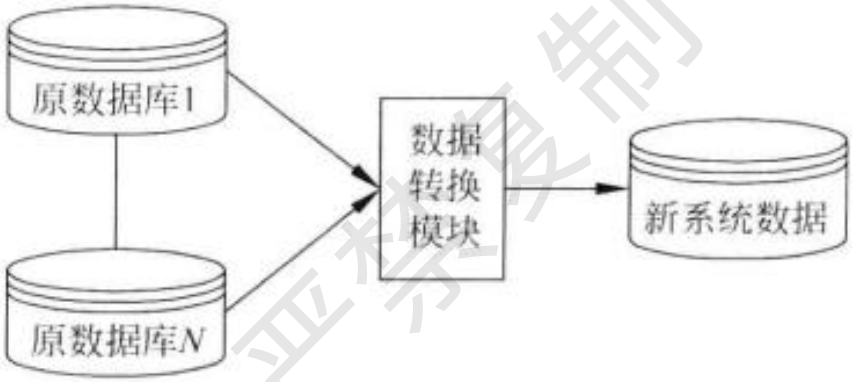
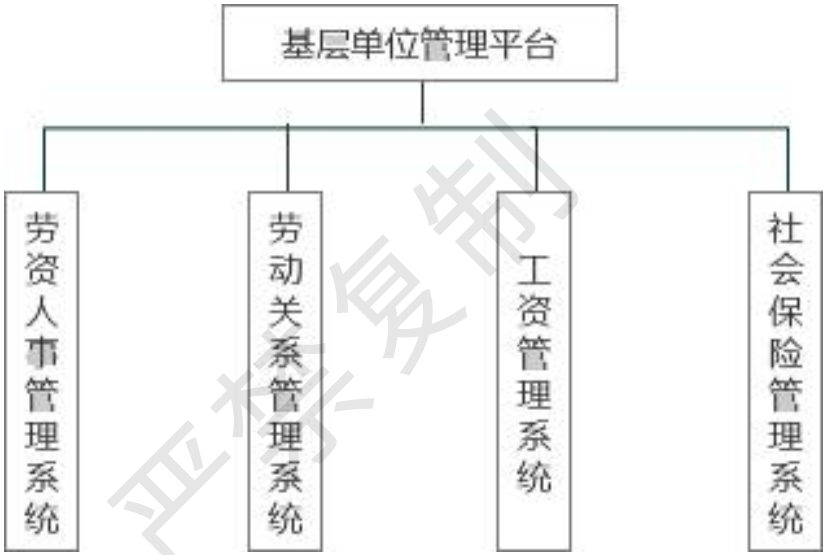



图 12-29 数据转换基类逻辑结构图

## 应用开发实例-SIMIS-基层单位管理平台



## 应用开发实例-SIMIS-业务管理系统

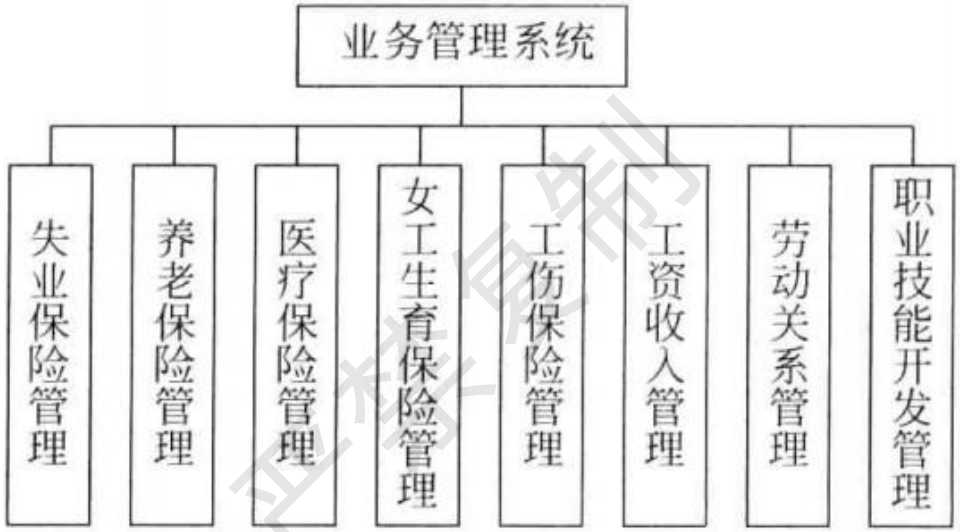
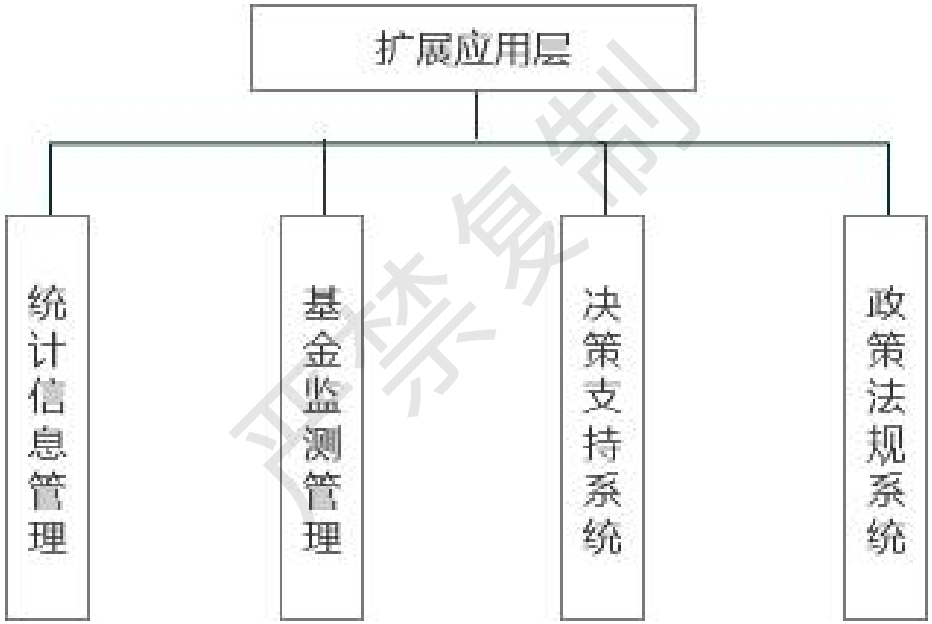
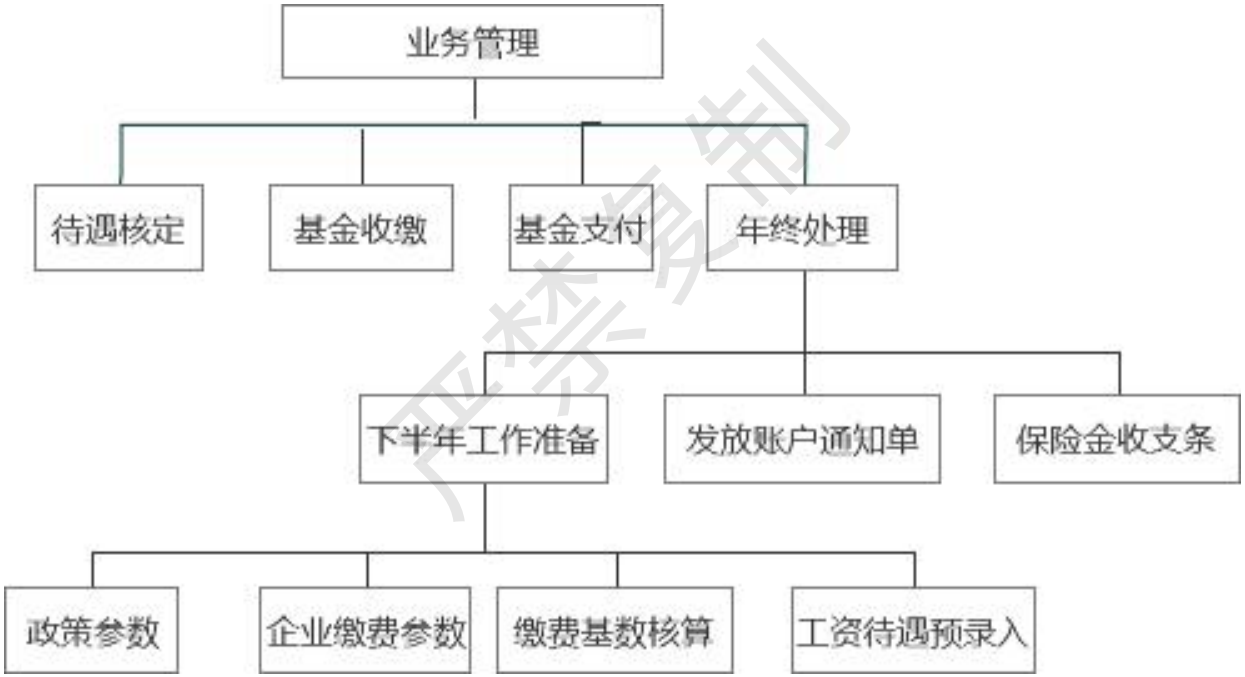



图 12-31 业务管理系统功能分解图

## 应用开发实例-SIMIS-扩展应用层



## 应用开发实例-SIMIS-系统设计与实现



## 应用开发实例-SIMIS-系统演化

- 判断完全可重用和部分可重用线索，并对变动的构件做修改，删除或更换标记
- 对原系统没有的新功能建立一条新线索
- 更新构件的相互作用
- 形成新的体系结构
- 确认以上步骤，进行阶段性技术评审
- 对所作的标记进行处理，完成一次演化过程