# 第一章

## 1. 软件架构的基本定义及其理解

- 基本定义

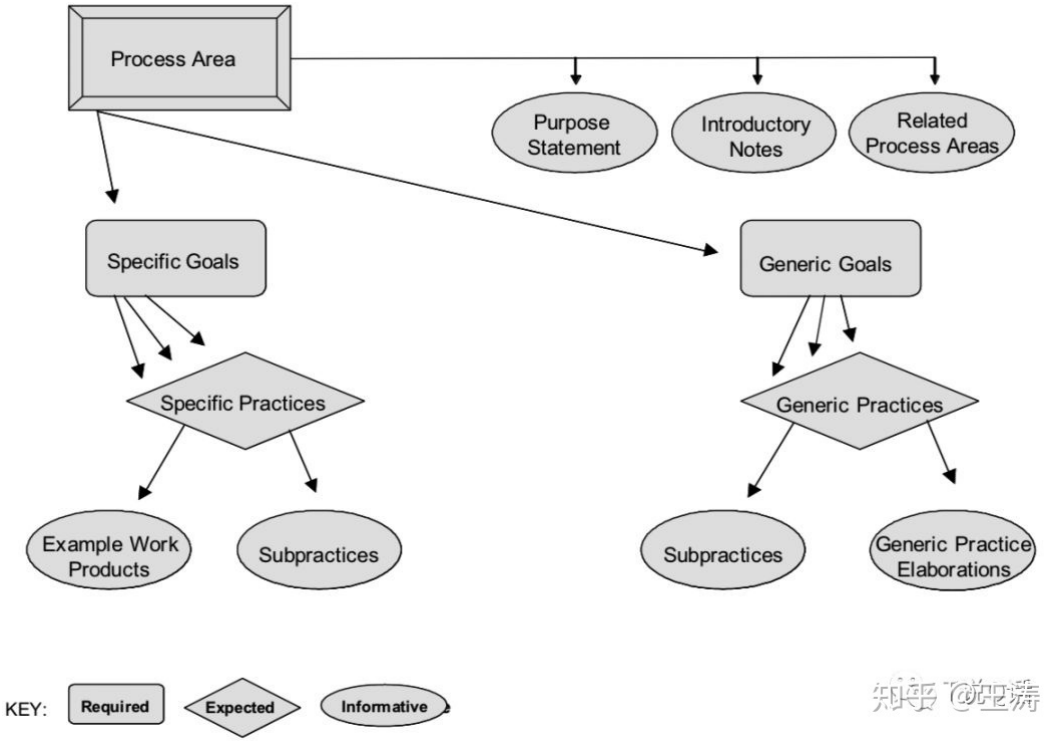
> 软件架构是由结构和功能各异、相互作用的构件集合，按照一定的结构方式构成的系统。
>
> 它包含系统的基础构成单元，它们之间的作用关系，在构成系统时它们的集成方法以及对集成约束的描述等。

- 软件的架构是关于软件的系统如何被组织起来的定义，即软件系统是由以下三要素构成 $P_8$
  - 组成系统的结构元素或统称为组成系统的组件
  - 构件与构件之间的连接以及特定的连接关系
  - 系统集成的方法和约束
- 将软件架构的定义浓缩为以下三个要素 $P_9$
  - 组成架构的元素：**构建**
  - 构件的相互联系：**连接**
  - 构件之间的相互联系关系：**连接关系**
- 进一步地理解软件构架定义，可以知道： $P_9$

> 架构是一个或多个结构（子架构、可不断细分）的抽象，是由抽象的构件来表示的，构件之间相互具有联系，相互之间的联系具有某些行为特征（连接关系）。
>
> 系统的这种抽象表示方式，屏蔽了构件内部特有的细节，使得在讨论系统构成的时候，只需关注与系统构成有关的上述三个要素的信息，而不必细看构成元素内部的内容。

## 2.CMMI模型的基本概念、分级体系、各级的含义等

- 基本概念
  - Capability Maturity Model Integration For Software——软件能力成熟度模型集成，是一套包括多个学科、可扩充的模型系列。
  - CMMI主要应用在两大方面：**能力评估**和**过程改进**




- 过程域（Process Area）

  > 用过程域的概念表示某个领域下相关的实践的集合，共有22个过程域。每个过程域都有一个固定的成熟度等级如技术解决方案（TS）的成熟度等级为3，而配置管理（CM）的成熟度等级为2.

- 成熟度等级（Maturity Level）

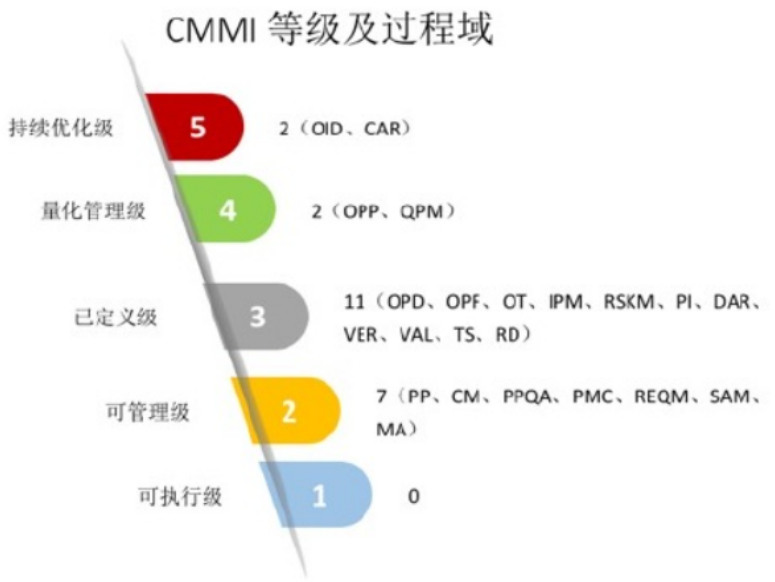
  > 包含一组预先定义的过程域以及相关的特点时间和通用实践。成熟度是一定已定义的演进方式，每一个成熟度都定义了过程域集合。

- 能力等级（Capacity Level）

  > 针对独立的过程域，通过对 Generic Goals 的能力级别满足来定义其能力等级。

- 每个过程域由模型组件组成，模型组件被分为三个种类
  - **必要的（Required）：** 用来说明满足一个过程域所要达成的成果，即目标，目标的满足用来评定最终结果。
  - **期望的（Expected）：** 用来说明要达成必要的目标所需要的方法，作为指导改进组织。
  - **辅助的（Informative）：** 用来帮助模型使用者理解必要信息。

- 分级体系：CMMI共有5个级别，代表**软件团队能力成熟度**的5个等级，数字越大，成熟度越高，高成熟度等级表示有**比较强的软件综合开发能力。**





- CMMI一级，**可执行级。**

  > 在执行级水平上，软件组织对项目的目标与要做的努力很清晰，项目的目标可以实现。但是由于任务的完成带有很大的偶然性，软件组织无法保证在实施同类项目时仍然能够完成任务。项目实施能否成功主要取决于实施人员。

- CMMI二级，**可管理级。**

  > 在管理级水平上，所有第一级的要求都已经达到，另外，软件组织在项目实施上能够遵守既定的计划与流程，有资源准备，权责到人，对项目相关的实施人员进行了相应的培训，对整个流程进行监测与控制，并联合上级单位对项目与流程进行审查。二级水平的软件组织对项目有一系列管理程序，避免了软件组织完成任务的随机性，保证了软件组织实施项目的成功率。

- CMMI三级，**已定义级。**

  > 在明确级水平上，所有第二级的要求都已经达到，另外，软件组织能够根据自身的特殊情况及自己的标准流程，将这套管理体系与流程予以制度化。这样，软件组织不仅能够在同类项目上成功，也可以在其他项目上成功。科学管理成为软件组织的一种文化，成为软件组织的财富。

- CMMI四级，**量化管理级。**

  > 在量化管理级水平上，所有第三级的要求都已经达到，另外，软件组织的项目管理实现了数字化。通过数字化技术来实现流程的稳定性，实现管理的精度，降低项目实施在质量上的波动。

- CMMI五级，**持续优化级。**

  > 在优化级水平上，所有第四级的要求都已经达到，另外，软件组织能够充分利用信息资料，对软件组织在项目实施的过程中可能出现的次品予以预防。能够主动地改善流程，运用新技术，实现流程的优化。

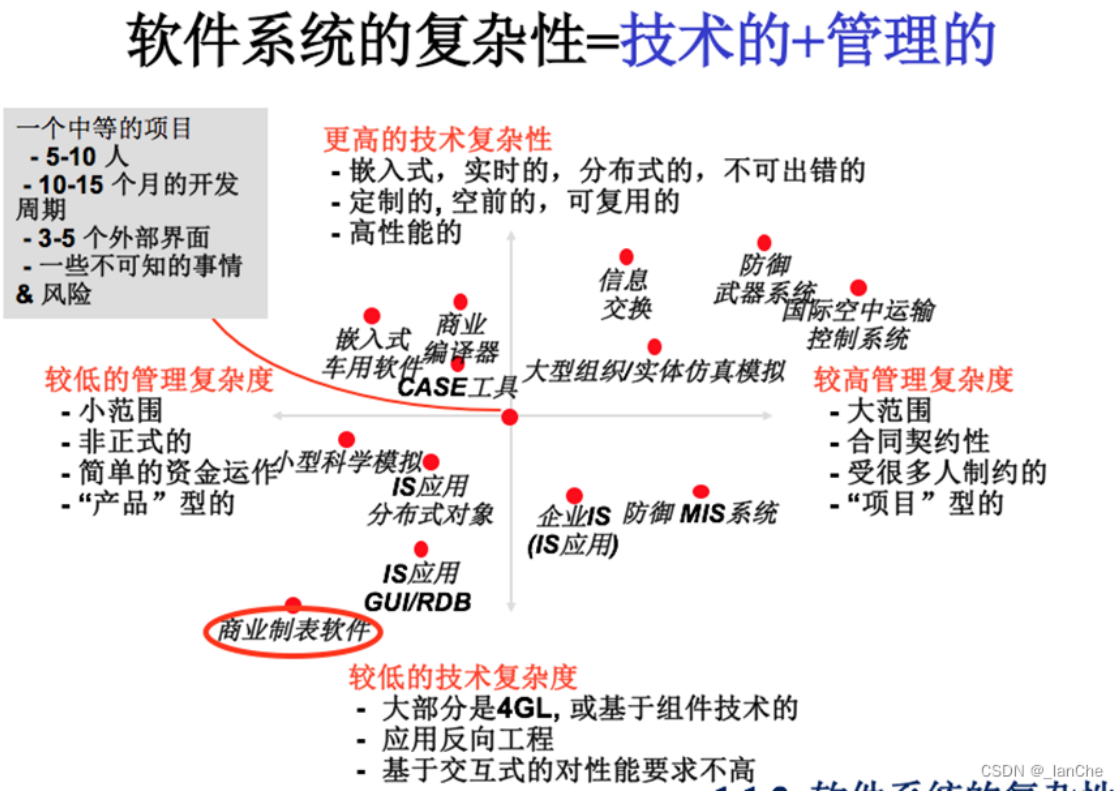
# 3.软件架构师和系统分析师的差别

- 在软件开发中的职责和角色，不难发现软件架构师与系统分析师所必需的知识体系也是不尽相同的
  - 系统分析师的主要职责是在需求分析、开发管理、运行维护等方面

        - 所以在系统分析师必须具备的知识体系中对**系统的构架与设计**等方面知识体系的要求就相对低些
    - 软件架构师的重点工作是在架构与设计这两个关键环节上
        - 软件架构师在**需求分析**、**项目管理**、**运行维护**等方面知识的要求也就相对低些

## 4.如何理解软件系统的复杂性

- 软件系统的复杂性包括：**技术+管理**




## 5.软件架构的作用

- 软件架构的作用，在系统设计层面上主要体现在4个方面：
    - 表达系统的功能需求是如何被实现的
    - 表达关键需求和设计约束的实现方案
    - 作为迭代开发的基础
    - 成为过程管理的依据

> 前三点反映的是对软件系统本身的作用，最后一点反映的是对软件开发过程的作用

# 第二章

## 1.文件传输架构设计。

- 构件 $P_{24}-P_{25}$
    - 发送和接受进程
    - 各自的文件缓冲和发送接受队列

    - 数据包
    - 上层进程
- 连接件
    - 物理连接：RS-232及串口线
    - 数据连接：信息队列、文件缓冲区
    - 控制连接：进程间通信缓冲、信号灯
- 连接关系
    - 物理连接协议：RS-232串口协议
    - 数据连接协议：信息队列和文件缓存区的使用方法（头地址、指针、长度、访问方式和限制等）、数据格式、信息含义、先进先出。
    - 控制连接协议：队列、信号灯的命令含义——暂停、继续、成功、失败。

## 最基本的->最完善的方式

- ?

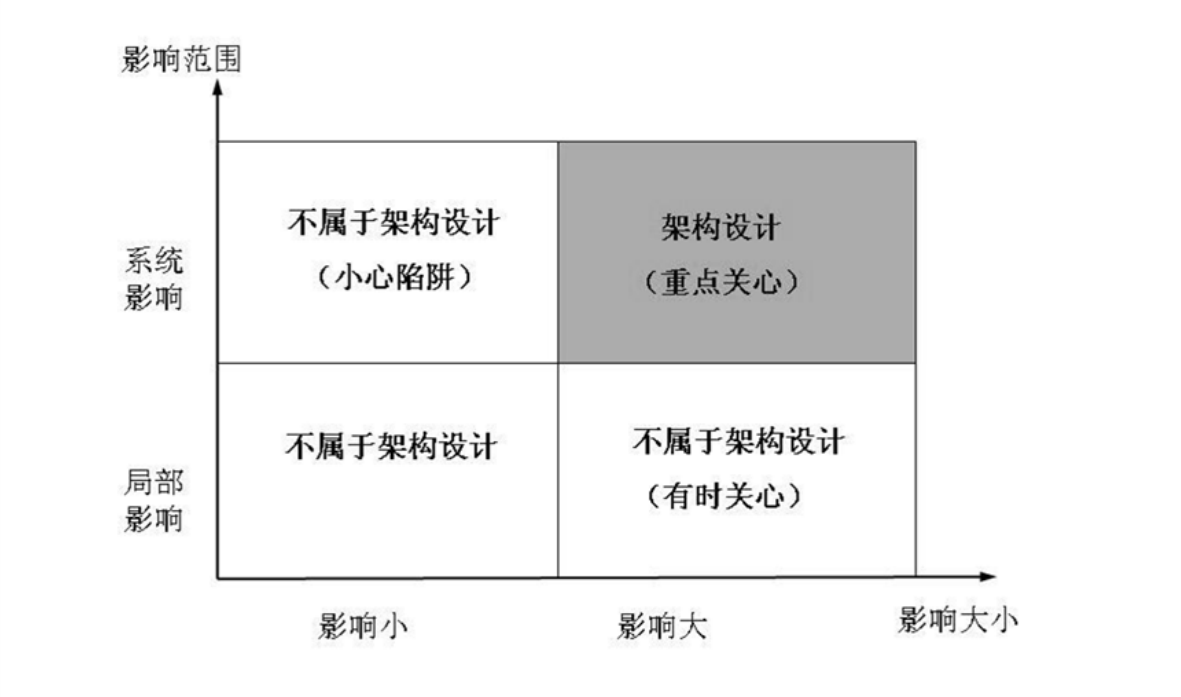
## 2. 软件系统的关键质量属性需求：有哪些属性？如何实践?

- 包括：可用性，可修改性，性能，安全性，可测试性和易用性。
- 如何实践：[软件质量属性*thinkingforever*的博客-*CSDN*博客软件质量属性]

## 3.好的软件架构和差的软件架构的典型特征有哪些

- 好的软件架构，具有如下的特征。 $P_{54}$
    - 灵活、具有可伸缩性
    - 考虑全面并可扩展
    - 思路简单明了,直接可以理解
    - 结构划分和关系定义清楚
    - 模块职责明确和分布合理
    - 效益和技术综合平衡
- 而差的软件架构的特征是：往往不是系统不能用(不是功能性问题），而是系统性能问题
    - 系统业务处理逻辑缺乏灵活性
    - 维护困难
    - 功能无法扩展
    - 运行速度太慢
    - 稳定性差，甚至经常宕机等

## 4.关键需求影响架构设计




$P_{53}$

- 区分哪些需求**根本与架构无关**——左下
- 区分哪些需求**貌似有关，而本质上是无关的**——左上
- 区分哪些需求虽然有关，但**并不需要在架构上做出特殊安排**，因为负面的代价或影响太大——右下
- 哪些是真正需要关注的需求——右上

## 5.软件产品线的基本定义及其公共性和管理性的定义。

- 软件产品线的基本定义 $P_{37}$

> 所谓产品线的一般定义是：满足特定市场或用户需求的，有一组公共的、可管理的产品特性的，符合上述两个条件的产品的组合。

- 软件产品线的公共和可管理特性是指：$P_{37}$

> 在一个“公共资源(资产)集”基础上开发的，而不是独立开发、从零开始开发的系统。理想的软件产品的产生是从公共资源库的构件开始，按定义的变化机制，对构件进行必要的裁减或添加，按产品线的公共架构组装，即在面向对象的构件与软件系统架构下的软件组装生产。

## 6.软件产品线管理的三大基本活动

- 三大基本活动 $P_{37}$
  - 核心资源开发
  - 利用核心资源的产品开发
  - 核心资源和产品开发的技术和管理

# 第三章

## 1. 软件架构的五个基本结构及其含义

- 开发架构 $P_{63}$

> 反映的是开发期的质量需求。表明开发过程应遵循开发团队所在组织所规定/要求的软件过程规范，特别是有关产品线技术管理的要求，并制定/满足相应的设计决策;它的具体涉及对象是程序包、第三方构件、类库、框架。

- 物理框架

> 反映安装和部署需求。它考虑软件系统与硬件环境的对应关系，设备部署和安装方案等；这里主要考虑主机、存储、网络的具体物理部署。

- 运行架构

> 反映的是运行期的质量需求。它针对系统运行要求，例如，与分布、并发、实时等性能、安全有关的要求，制定/遵循相应的设计决策；它与进程、线程、对象，以及架构的关键运行机制有关。

- 逻辑架构

> 反映的是功能需求是如何被分解和协同实现的。逻辑架构设计是规划组成系统的所有构件，为它们分配不同的职责，使得这些构件能通过协作，完成功能需求；它的具体涉及对象是系统的逻辑层次、功能构件和类的划分 、交互等。

- 数据架构

> 反映的是数据需求。在这里应考虑数据的分布、生成与应用的关系，设计合适的数据持久化存储和传递策略等。数据存储格式、数据字典、安全备份 、复制、同步、数据传递是这里主要考虑的内容。

## 2.软件架构的五个基本视图及其含义

- 逻辑视图（Logical View）$P_{65}$

> 逻辑视图关注的是系统必领为用户提供的功能，不仅包括用户可见的功能，还包括为实现用户功能而必须提供的系统功能，即那些“辅助性的功能”和“性能”；它们可能是逻辑层、功能模块等。

- 开发视图（Development View）

> 开发视图关注的是程序包，不仅包括要编写的源程序，还包括可以直接使用的第三方 SDK 和现成框架、类库，以及开发的系统将运行于其上的系统软件或中间件等。开发视图关注软件开发环境下实际模块的组织，反映了开发难度、软件管理、重用性和通用性及由工具集、编程语言所带来的限制和约束等。

- 进程视图 （Process View）

> 进程视图的关注点是运行中的进程、线程、对象等概念，以及相关的并发、同步、通信等问题。

- 物理视图（Physical View）

> 物理视图关注的是：目标程序及其依赖的运行库和系统软件最终如何安装或部署到物理机器，以及如何部署机器和网络来配合软件系统的可靠性、可伸缩性等要求。

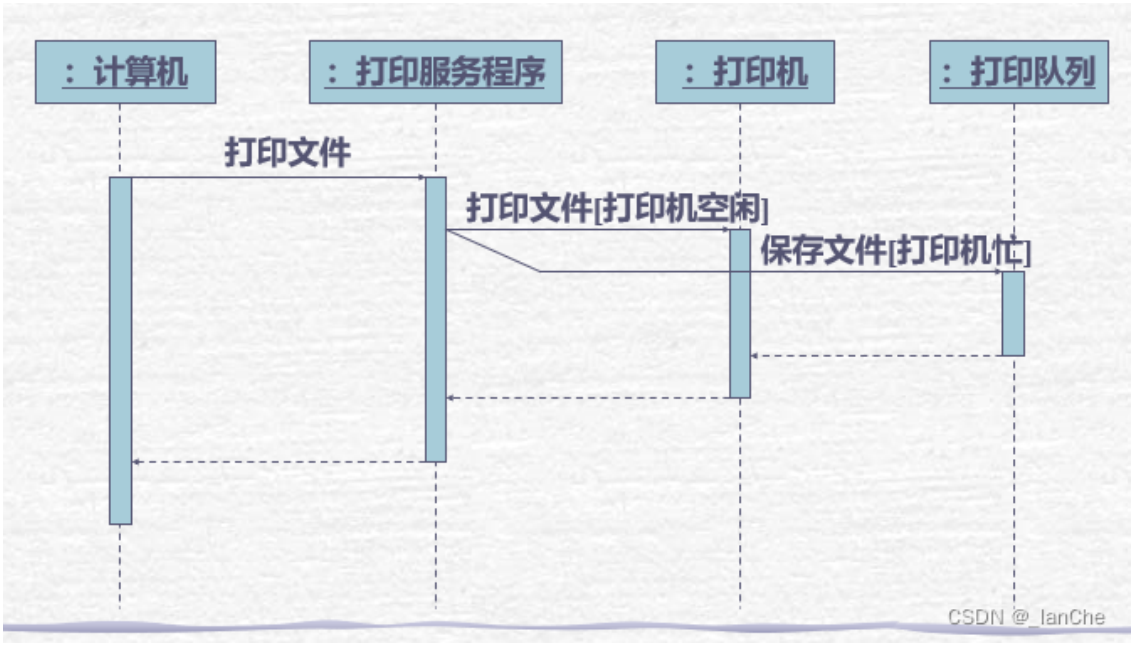
- 场景视图（Scenarios View）

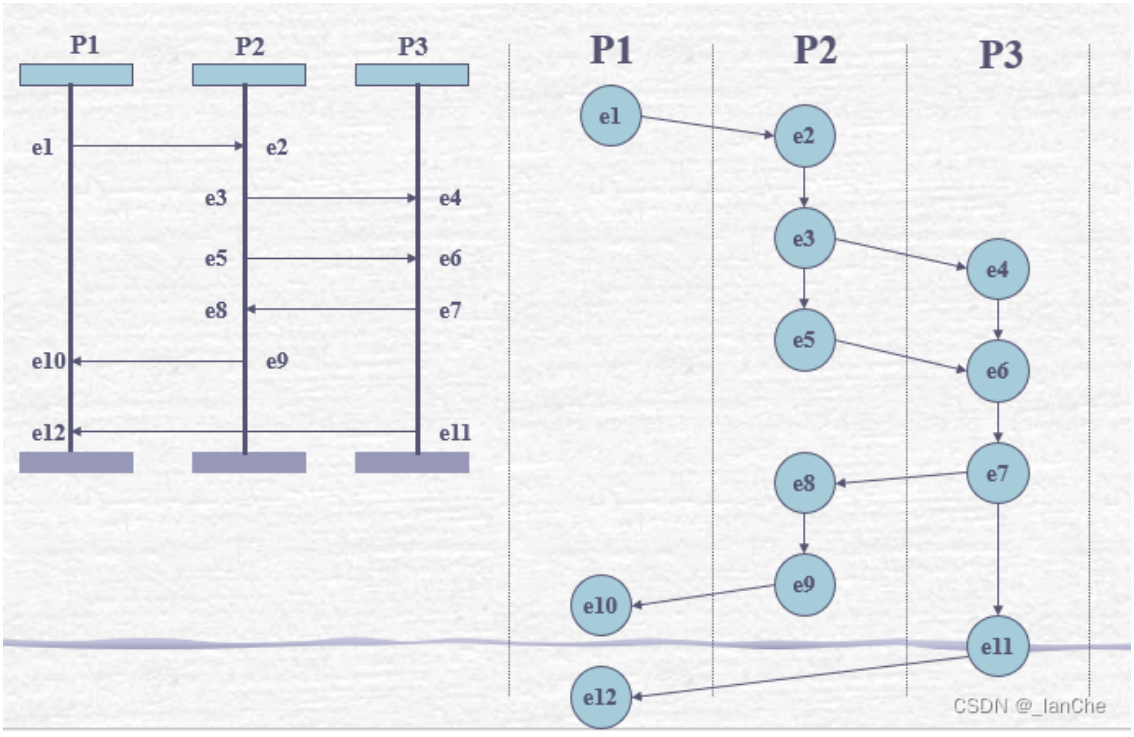
> 场景视图作为需求的描述，在某种意义上，是最重要的需求抽象。该视图是其他视图的冗余（因此称为“十1”）。

## 3. 掌握UML绘图的基本绘制方法：时序图、包图。

- 时序图（百度上更多的是第一张图）

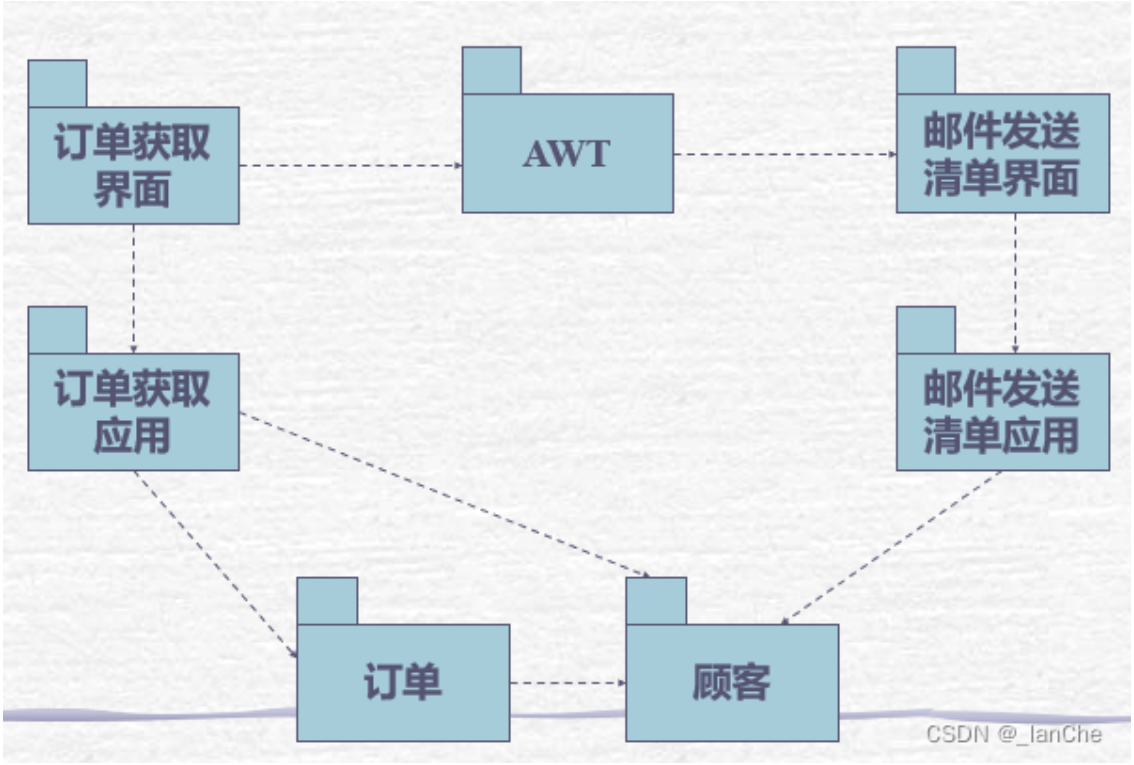
> 顺序图由一组对象构成，每个对象分别带有**一条竖线**，称作对象的生命线，它代表时间轴，时间沿竖线向下延伸。顺序图描述了这些对象随着时间的推移相互之间交换消息的过程。消息用从一条垂直的对象生命线指向另一个对象的生命线的**水平箭头**表示。图中还可以根据需要增加有关时间的说明和其他注释。



- 包图

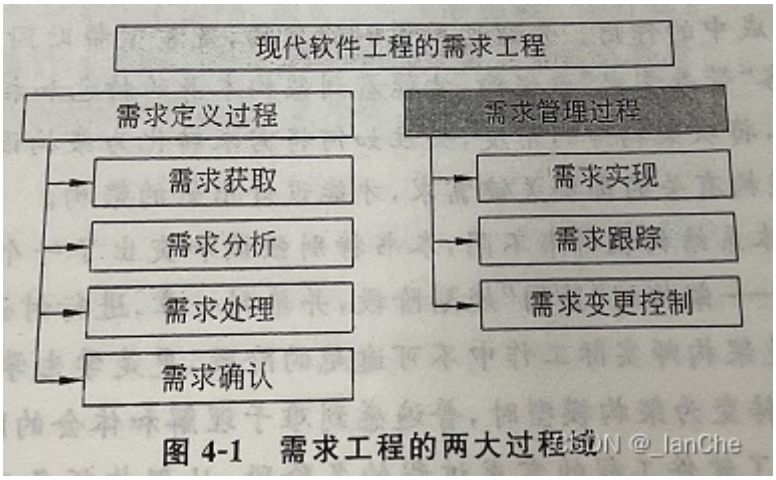
> 包是类的集合。
>
> 包图所显示的是类的包以及这些包之间的依赖关系。
>
> 如果两个包中的任意两个类之间存在依赖关系，则这两个包之间存在依赖关系。
>
> 包的依赖是不传递的。




# 第四章

## 1.理解现代软件工程的需求过程：包括需求开发过程及需求管理过程。何为需求管理过程域?

- 如下图所示 $P_{110}$




- 需求管理过程域

- 开发活动早期要确保需求被正确理解
- 开发过程中要确保各种活动产生的工作产品都能满足用户和开发方一致认可的需求。

## 2.需区分析阶段中，架构师应该关注的要点

- 软件产品的用户功能需求
- 软件产品的非功能性需求
- 组织的或产品的设计约束和限制

> 通常这三部分需求，构成了软件需求的总集。后两个部分是架构师“新增加的需求”。架构师的责任是保证这三部分的需求，能够合适的“糅合”在一起。

// 以下看似像答案

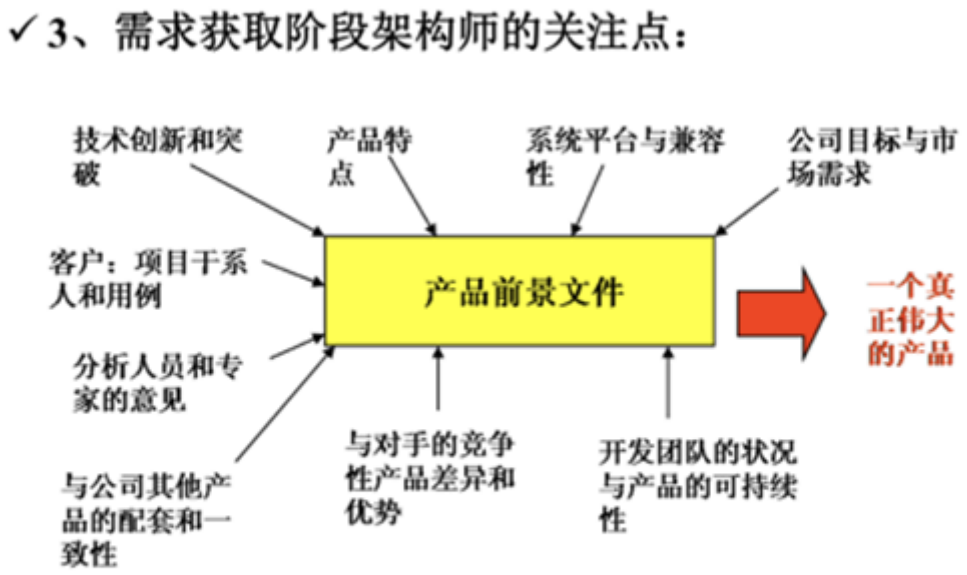
- 考虑与实现有关的一些问题

> 例如，为实现用户需求的架构方式、模块划分、交互方式等

- 用系统实现的（而不再是用户）眼光，进一步对《需求说明书》所描述的需求，进行分解、定义、取舍、平衡、折中，得到符合“系统设计需要”的需求描述

> 如面向对象的OMT(静态、动态、功能)模型中的类图、交互图等。

## 3.需求获取阶段架构师的




需求获取阶段的产品管理考虑



## 4.某个棋类游戏，从需求分析师和架构分析师的角度，分析各自不同的关注点和相同点

# 5.UC矩阵 $P_{132}$

## 5.1 UC矩阵的概念

- U/C矩阵是用来表达过程与数据两者之间的关系。

  > 矩阵中的行表示数据类，列表示过程，并以字母U（Use）和C（Create）来表示过程对数据类的使用和产生。

- U/C矩阵是一张表格，可以表示数据/功能系统化分析的结果

  > 它的左边第一列列出系统中各功能的名称，上面第一行列出系统中各数据类的名称。表中在各功能与数据类的交叉处，填写功能与数据类的关系。

## 5.2 UC矩阵的原则

## 5.3 主要功能

> 通过 U/C矩阵的正确性检验，可以及时发现前段需求分析和调查工作的疏漏和错误，分析数据的正确性和完整性。
>
> 通过对 U/C矩阵的求解过程，最终可得到子系统的划分，通过对子系统之间的联系(U),可以确定子系统之间的共享数据

## 5.4 正确性检验原则

- 完备性检验。

  > 这是指每一个数据类必须有一个产生者(即“C”) 和至少有一个使用者(即“U”)；
  >
  > 每个功能必须产生或者使用数据类。否则这个U/C矩阵是不完备的。

- 一致性检验。

  > 这是指每一个数据类仅有一个产生者，即在矩阵中每个数据类只有一个“C”。
  >
  > 如果有多个产生者的情况出现，则会产生数据不一致的现象。

- 无冗余性检验。

  > 这是指每一行或每一列必须有“U” 或“C”，即不允许有空行空列。
  >
  > 若存在空行空列，则说明该功能或数据的划分是没有必要的、冗余的。

# 第五章

# 1.OSI模型的启发意义：服务、接口、协议等的启发作用。

$P_{203}$

> 从架构的观点看，OSI参考模型的最大贡献，是对以下三个概念做出了明确的区分和定义

- 服务

  > 每一层都为它的上一层提供服务(原语操作)，服务定义了这一层应该做什么，而不管上面的层如何访问它，以及本层如何具体实现预定的服务工作。

  - 对架构设计的参考意义

  > 以架构格局来看，每一层提供了一个相对上一层而言，更“基础“的服务功能。架构师对于服务层次的设计，应该看到的是”逐级抽象“。

- 接口

  > 接口（原语）告诉自己的上层，应该如何访问自己，定义了访问的参数和预期的结果，这也和该层如何实现无关。

  - 对架构设计的参考意义

  > 接口不单是模块之间访问的通道，而且成为基于架构的系统开发标准。

- 协议

  > 协议是端到端的对等协议（帧格式、信息定义），是该层的内部事物。因此，采用什么协议或协议的改变，是端与端之间的事，甚至与上层(服务)无关。

  - 对架构设计的参考意义

  > 在架构师看来，协议是模块内部的功能，所以不关心。

- 从架构师的眼光看，开放式系统互连参考模型的优点

  > 它完全符合结构化模式的设计要求。

- 缺点

  > 由于模型产生在应用(协议发明)之前，有些功能不知道应该归在哪层更合适。

# 2.典型软件架构模型

## 流程处理系统 $P_{209}$

- 基本方式

  > 流程处理系统以程序算法和数据结构为中心，如输送或过滤器般地处理数据。手机、调制解调器等流式信息处理系统是典型的流程处理系统。其每一个处理过程中，先接受输人数据，对它们进行处理(过滤)，最后产生输出数据，交给下一个处理流程。

- 特点

  - 优点

  > 流程处理系统的优点是：

> 1. 系统的总体行为，是各个处理模块的简单、线性的组合。
> 2. 只要输人和输出数据合适，处理模块可以在不同系统中重复使用。
> 3. 只要增加新的处理模块，系统可以很容易地进行功能扩展。
> 4. 如果存在合适的并行算法，系统可以采用并行计算的方式进行，因而可以在处理速度和处理能力上获得提升，从而解决更复杂的工程技术和科研难题。

- 缺点

> 1. 它主要以一个批量处理的方式进行，不太适合作交互式应用；
> 2. 当有大量和不同形式的输人和输出数据时，数据处理和管理控制将变得非常复杂；而总体的处理能力受某一个具体处理"瓶颈"的影响。

## 客户 / 服务器系统 $P_{210}$

- 基本方式

> 简单的客户/服务器系统结构分成以下两部分。
> (1）容户机负责用户输人和展示，服务器则处理底层的功能，例如数据库的运作等。服务器通常含有一组服务器对象,能同时为多个客户机服务;
>
> (2）在客户/服务器系统中，商业逻辑可由客户机,也可由服务器处理。客户机和服务器之间，通过约定的协议来交互，常见的协议有：HTTP（超文本传输协议），CORBA 等。

- 优点

> - 客户机与服务器分离，允许作长距离的连线运作。
> - 两者的开发也可以分开同时进行
> - 一个服务器可以服务于多个客户机

- 缺点

> - 客户机与服务器的通信依赖于网络，可能成为整体系统运作的瓶颈
> - 倘若服务器及其界面定义有改变,则客户机也要做相应改变。
> - 当客户机数量很大时，要维护所有的容户端都买用最新软件版术，可能要花很高的成本。

## 层状系统 $P_{212}$

- 基本方式

> 层状系统则是带有这些分组或分层结构的软件系统，它常见于服务器中，下层给上层提供服务。已经介绍过的互连参考模型和TCP/IP 网络协议层次结构和OS 层次结构，就是这样的例子。

- 优点

> (1）在模块或节点中分设层，有助于把复杂的问题按功能分解，使 整体设计更为清晰；
> (2）由于内层与外界隔绝，内层函数和服务受到有效的控制，只有界面层的对象作为界面类向外界公开；
> (3）新的运算可以在界面层引入，它们把内层（核心或持久）的一些运算组合起来;
> (4）如果界面合适，某一个层反复用在不同地方，一个自成一体的层，也可以作为模块或节点使用。

- 缺点

> (1）层数多时，系统性能就会下降。因为界面函数可能需要通过好几层，才能到达某一内层；
> (2）标准化的层界面可能变得臃肿，降低函数调用的性能。

### 三级和多级系统 $P_{212}$

- 基本方式

> 层是模块或节点以内的对象分组，而“级”则是系统中相互联系而各自分离的模块或节点实体。
>
> 三级系统的概念可以推广到多级系统，它是由一连串的客户/服务器对组成。所以在 $N$ 级系统中，可能有 $N-1$ 对客户/服务器。

- 优点

> (1）因为系统的功能分布在几级或服务器上，系统维修和扩展都比较容易；
>
> (2）从底层到高层，可以分级控制,对不同级的客户机提供不同水平的服务；
>
> (3）方便企业水平的整合：通常的作法是把中间级与企业的其他系统连接起来；
>
> (4）多级系统可以扩充，以服务大量同时使用系统的客户机。

- 缺点

> (1）不同的客户/服务器对之间可能存在多种不同的通信协议；
> (2）由于数据要通过多级节点,而各个节点可能在不同的计算机和操作系统中，协调、提升系统的整体性能就很不容易。

# 第七章

## 接口的相关概念及应用

$P_{254}$

- 接口的概念

> 接口（硬件类接口）是指同一计算机不同功能层之间的通信规则称为接口
>
> 接口（软件类接口）是指对协定进行定义的引用类型。其他类型实现接口，以保证它们支持某些操作。接口指定必须由类提供的成员或实现它的其他接口。与类相似，接口可以包含方法、属性、索引器和事件作为成员。

- 应用
  - `interface` 定义接口
  - `implements` 实现接口

```
public interface USB {
    void read();
    void write();
}
```

```
class YouPan implements USB {
    @Override
    public void read() {
        System.out.println("U盘正在通过USB功能读取数据");
    }

    @Override
    public void write() {
        System.out.println("U盘正在通过USB功能写入数据");
    }
}
```

# 设计模式的基本概念及要素

- 基本概念 $P_{275}$
  - 软件设计模式：一种从软件设计过程中总结出来的、广泛应用和成熟的结构和结构关系
  - 每个模式都描述了一个在特定的环境中不断出现的问题，然后描述了该问题的解决方案的核心。所以设计模式是针对特定问题的成功解决方案。
- 要素 $P_{279}$
  - 模式名 (Pattern name)

    > 用一个词来简单表示模式的问题、解决方案和效果。恰当和贴切的模式名，可以让使用者对模式的作用“一目了然”

  - 问题 (Problem)

    > 描述应该在何时使用模式，它针对哪些特定的设计问题、问题的前因和后果，以及使用模式的先决条件

  - 解决方案 (Solution)

    > 解决方案是模式本体，描述模式是如何构成的，它们之间的相互关系和责任。模式是一个问题类的解决方案的抽象模板，因此，它不描达具体的设计或实现，只提供设计问题的抽象思路。

  - 效果 (Consequences)

    > 描述模式的应用效果和使用模式应权衡的问题。这是在设计阶段进行设计选择时，不可避免要考虑的问题。包括：好处和代价、时间和空间、复用和灵活性、可扩展和可移植性等的平衡。

# 单例模式 $P_{287}$

[单例模式 | 菜鸟教程 (runoob.com)]

- 意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- 动机

> 有很多环境要求只能有一个实例。例如，系统只能定义一个默认打印机，如果有两个就不叫默认了。类负责保存它的唯一实例，保证没有其他实例可以被创建(通过截获创建新实例的请求），并且它可以提供一个访向该实例的方法，这就是单例模式。

- 适用性

  > 当类只能有一个实例并且客户从一个众所周知的访同点访问它时(无需选择)。当这个唯一的实例应该是通过子类化可扩展的，并且客户应该无须重改代码就能使用一个扩展的实例时(只有一条通道走到底)!

- 注意

  > - 单例类只能有一个实例。
  > - 单例类必须自己创建自己的唯一实例。
  > - 单例类必须给所有其他对象提供这一实例。

- 实现

```
public class SingleObject {

   //创建 SingleObject 的一个对象
   private static SingleObject instance = new SingleObject();

   //让构造函数为 private，这样该类就不会被实例化
   private SingleObject(){}

   //获取唯一可用的对象
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //不合法的构造函数
      //编译时错误：构造函数 SingleObject() 是不可见的
      //SingleObject object = new SingleObject();

      //获取唯一可用的对象
      SingleObject object = SingleObject.getInstance();

      //显示消息
      object.showMessage();
   }
}
```

## Adapter模式

适配器模式 | 菜鸟教程 (runoob.com)

- 定义

  > 作为两个不兼容的接口之间的桥梁，这种类型的设计模式属于**结构型模式**，它结合了两个独立接口的功能。

> 适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高，且要求程序员了解现有组件库中的相关组件的内部结构，所以应用相对较少些。

- **意图：**将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
- **主要解决：**主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。
- **使用场景：**有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式。
- **注意事项：**适配器不是在详细设计时添加的，而是解决正在服役的项目的问题。
- 适配器继承或依赖已有的对象，实现想要的目标接口

# 第八章

## MVC模式的核心思想

- 核心思想 $P_{352}$

> MVC的核心思想是将一个应用分成三个基本部分：Model（模型）、View（视图）和Controller（控制器），这三个部分以最少的耦合协同工作，从而提高应用的可扩展性及可维护性。

- MVC的观察者（Observer）机制：MVC的关键是实现了所谓的观察者机制。

## ORM

- ORM(Object-Relational Mapping）$P_{391}$

> 即对象关系映射，是指以 O/R 原理设计的持久化框架(Framework），包括 0/R机制、SQL自生成、事务处理和 Cacbe 管理等。

- ORM 的实现思想

> 将关系数据库表中的数据记录，映射成为对象，以对象的形式展现，这样开发人员就可以把对数据库的操作转化为对这些对象的操作。因此它的目的是为了方便开发人员以面向对象的思想来实现对数据库的操作。

## 容器

- 容器是指为组件提供特定服务和技术支持的一个标准化的运行时的环境
- 容器是一种沙盒技术，主要目的是为了将应用运行在其中，与外界隔离；及方便这个沙盒可以被转移到其它宿主机器。
- 容器底层主要运用了名称空间（Namespaces）、控制组（Control groups）和切根（chroot）。
  - 名称空间

    > 每个运行的容器都有自己的名称空间。

  - 控制组

- 切根

> 切根的意思就是改变一个程序运行时参考的根目录位置，让不同容器在不同的虚拟根目录下工作，从而相互不直接影响。

- Spring中的应用 $P_{385}$

> Spring 容器是 Spring 框架的核心。容器将创建对象，把它们连接在一起，配置它们，并管理他们的整个生命周期从创建到销毁。Spring 容器使用依赖注入（DI）来管理组成一个应用程序的组件。这些对象被称为 Spring Beans
>
> Spring 包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，用户可以配置自己的每个 Bean 如何被创建：基于一个可配置原型 (prototype),Bean 可以创建一个单独的实例或者每次需要时都生成一个新的实例—以及它们是如何相互关联的。然而,Spring 不应该被混同于传统的重量级的 $EJB$ 容器，它们经常是庞大与笨重的，难以使用。

# IoC 控制反转

- Spring中的应用

> Spring 通过一种称作控制反转(Inversion of Conttol,IoC)的技术促进了松耦合。当应用了 IoC，一个对象依赖的其他对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。可以认为 IoC 与JNDI 相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求,就主动将依赖传递给它——依赖注入(Dependence Injection, DI).

- 与MVC 的核心是变更—传播机制一样，Spring 框架的核心，就是控制翻转 IoC/依赖注入DI机制。
- IoC 是指由容器中控制组件之间的关系，而非传统实现中由程序代码直接操控，这种将控制权由程序代码到外部容器的转移，称为“翻转”
-