UML-封神之路的开始

原创

我跟你拼啦 已于 2022-04-22 11:55:30 修改 3043 收藏 94
分类专栏： 面向对象 文章标签： uml java
于 2021-09-18 10:05:16 首次发布

第一章 面向对象技术概述

乔治 3.0 时间 : 2021-06-16

软件危机和软件工程

控制软件系统复杂性的基本方法:

(1)分解

对于复杂的软件系统, 可以逐步将它分解成越来越小的组成部分, 直到不能再分解. 如Unix中的shell和管道就采用了分解思想.

(2) 抽象

当我们用抽象这个概念时, 我们承认正在考虑的问题是复杂的, 但我们并不打算理解问题的全部, 而只是选择解决其中的主要部分, 不去注意那些细节而已.

抽象又分为过程抽象和数据抽象(数据类型)

在面向对象兴起之前, 编程以过程为中心, 例如结构化设计方法. 然而, 系统已经达到了超越其处理能力的复杂性极点. 有了对象, 我们能够提升抽象级别来构建更大的, 更复杂的系统.

(3) 模块化

高内聚, 低耦合. 高内聚指的是在一个模块中应尽量多地汇集逻辑上相关的计算资源; 低耦合指的模块之间的相互作用尽可能地少

(4)信息隐蔽

信息隐蔽也称封装, 把模块内的实现细节与外界隔离, 用户只需知道模块的功能, 而不需要了解模块内部细节.

在结构化方法中, 现实世界被映射为功能的集合. 在面向对象方法中, 现实中的实体及其相互关系被映射为对象及对象之

间的关系.

对象和实例

对象(object)中系统中用来描述客观事物的一个实体, 它是构成系统的基本单位. 一个对象由一组属性和操作这组属性的一组方法组成.

对象之间通过消息通信, 一个对象向另一个对象发送消息激活某个功能.

实例(instance)的概念和对象类似, 实例的含义更加广泛, 不仅对类, 其他建模元素也有实例. 如类的实例就是对象, 而关联的实例就是链

类

类是具有相同属性和方法的一组对象的集合,(比如:实例类) 它为属于该类的全部对象提供了统一的抽象描述. 同一个类的对象具有相同的属性和方法, 这里是指它们的定义形式相同, 而不是说每个对象的属性相同.

类是静态的, 类的语义和类之间的关系在程序执行前就已经定义完毕, 而对象是动态的, 对象是程序执行时被创建或删除的.

封装

封装(encapsulation)就是把对象的属性和方法结合成一个独立的单位, 并尽可能地隐藏对象内部细节

封装使对象成为两个部分: 接口和实现部分, 对于用户来说, 接口是可见的, 而实现的部分是不可见的.

封装提供了两种保护. 1是可以保护对象, 防止用户直接访问对象的内部细节.

2是可以保护客户端, 以防对象实现部分的变化产生副作用, 即实现部分的改变不会影响到相应客户

端的改变

继承

继承就是子类可以继承父类的属性或方法.

继承好处: 1增加了软件重用的机会, 可以降低软件开发和维护的费用. 2. 可以保证类的一致性, 父类可以为所有子类定制规则, 子类必须遵守这些规则.

如:c++的虚函数, java的接口等.

多态

在面向对象的技术中, 多态指的是一个实体拥有在不同上下文条件下具有不同的意义或用法的能力

多态往往和覆盖, 绑定等概念结合使用.多态是保证系统具有良好的适应性的重要手段之一.

消息

消息(message)就是面向对象发出的服务请求. 它包含了提供服务的对象标识, 方法标识, 输入信息和回答信息等.

消息: 包括同步消息和异步消息. 如果消息是异步的, 则一个对象发送消息后就继续自己的活动, 不等待消息接收者返回结果. 而函数往往是同步的, 函数调用者要等待接收者返回结果.

第二章 UML概述

UML的构成

1. 基本构造块:

关系(依赖dependency, 关联association, 泛化generalization, 实现realization)

图(diagram)

事物(thing):

(1). 结构事物(structural thing):类(class), 接口(interface), 协作(collaboration), 用例(use case), 主动类(action

class), 构件(component) 和节点(node)

(2). 行为事物(behavioral thing): 交互(interaction) 和 状态机(state machine)

(3). 分组事物(grouping thing): 包(package)

(4). 注释事物(annotational thing): uml中的注解

2. 规则(rule)

3. 公共机制(common mechanism)

4. 五个语义规则

命名, 范围, 可见性, 完整性, 执行

5.通用机制:

1规范说明,

2.修饰

3.通用划分,

4.扩展机制, 其中扩展机制包括版型, 标记值, 约束

UML的视图

UMl中的视图:

用例视图: 表示系统的功能性需求

逻辑视图: 表示系统的概念设计和子系统结构等

实现视图: 说明代码的结构

进程视图: 说明系统中的并发执行和同步情况

部署图: 用于定义硬件节点的物理结构

UMl的应用领域

软件开发, 金融… 范围极大

UML的应用示例

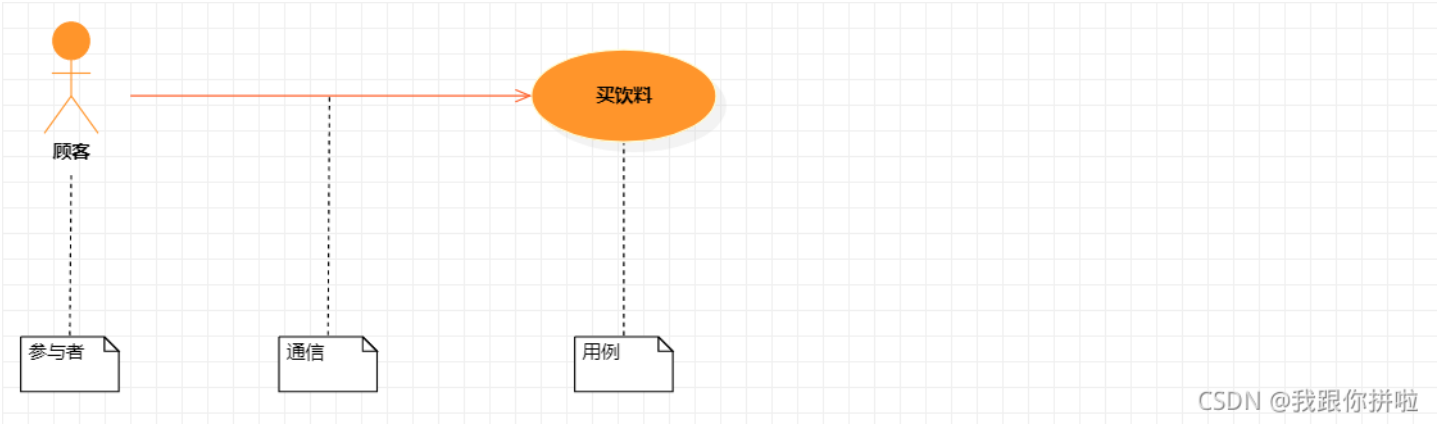
第三章 用例图: 建模的开始

什么是建模

> 选择一个角度, 对现实进行抽象. 比如, 美式咖啡和拿铁咖啡, 对这二者进行抽象, 就得到它们的父类: 咖啡类
>
> 在统一过程中, 一个用例就是一个分析单元 , 设计单元, 开发单元, 测试单元, 甚至是部署单元.

用例图基本概念

1. 用例图与协作图, 顺序图, 活动图共同组成了用例视图
2. 用例图是需求分析的第一步, 用例图用来描述系统的功能需求集合
3. 在uml中, 用例图用椭圆来表示.



参与者的概念

参与者是指系统以外的, 需要使用系统或与系统交互的事物, 包括人, 设备, 外部系统等.

例如:

1.银行业务系统中可能的参与者: 客户 管理人员 厂商 Mail系统

2.教务管理系统可能的参与者: 学生 教师 管理员

案例: 你要寄信, 把信交给了邮局工作人员. 这时, 你是参与者, 工作人员不是参与者, 而是业务工人.

寻找和确定参与者

建模是从寻找抽象角度开始的, 那么, 定义参与者就是我们寻找抽象的开始, 参与者在建模中处于核心地位

为找出参与者, 我们需要问两个问题:

1.谁对系统有着明确的目标和要求并且主动发出动作?

2.系统为谁服务?

例如: 小王去银行开户, 向大厅经理询问了办理流程, 填写了表单并交给柜台职员, 最后拿到银行存折.

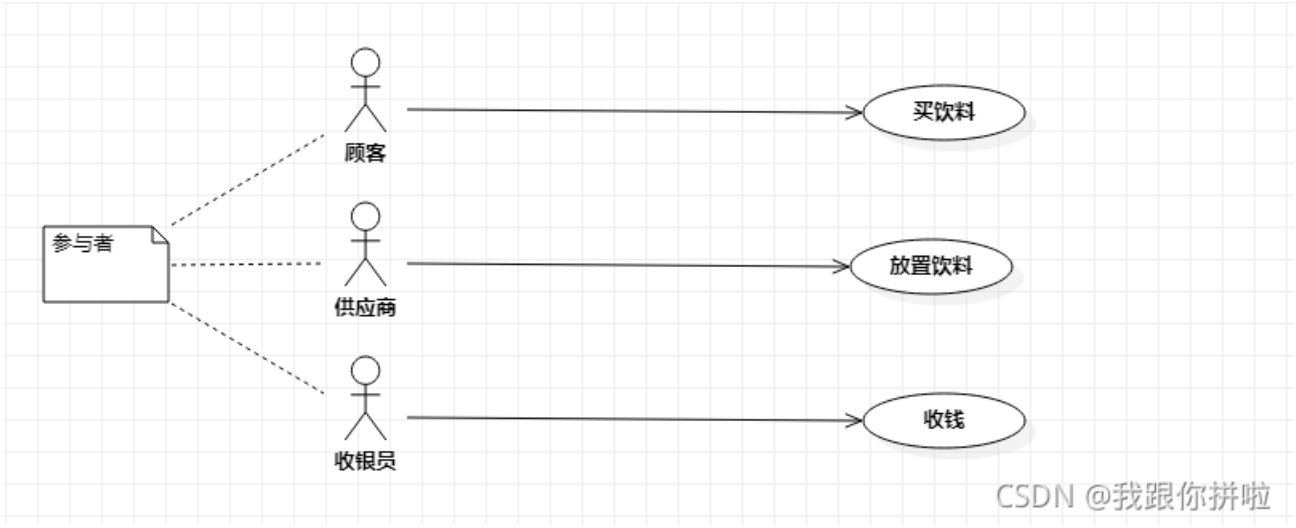
在这个案例中, 小王是参与者, 大厅经理和柜台职员是业务工人.

检查点

需要问两个问题:

1. 是否找到所有参与者?
2. 每个参与者是否至少涉及一个用例?

例如, 在自动饮料售货机中, 有三个参与者



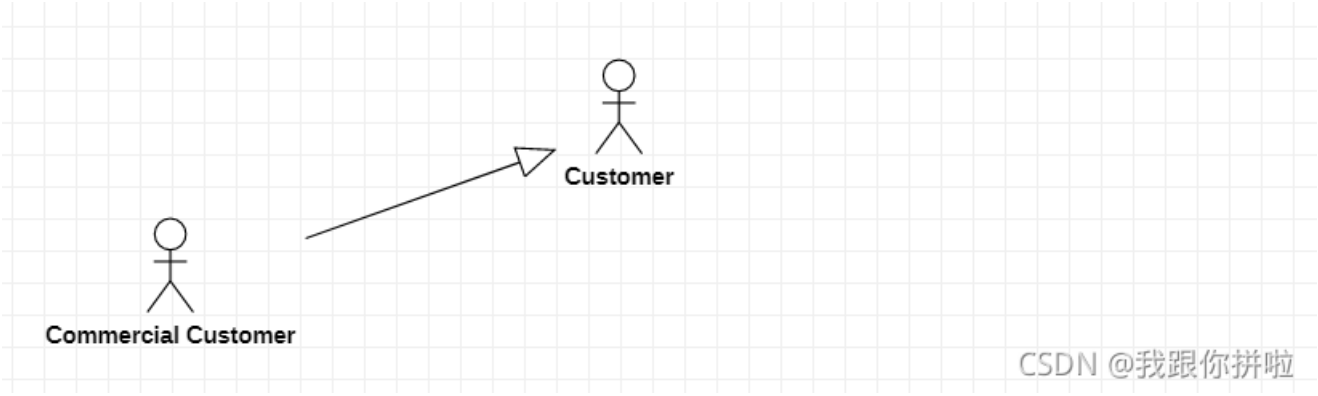
参与者之间的关系

由于参与者事实上就是类, 因此, 参与者之间也有继承关系(泛化).

参与者之间的泛化关系表示父参与者与子参与者之间的联系. 子参与者不仅可以继承父参与者的行为和含义, 还可以增加自己独有的行为和含义.

泛化关系用带三角箭头的实心表示.

例如:



用例

定义: 用例是系统, 子系统或类 和 外部的参与者交互的动作序列的说明, 包括可选的动作序列和会出现异常的动作序列

在uml中, 用例用一个椭圆表示, 用例往往用动宾结构或主谓结构命名

采用用例进行需求分析有如下特点:

1.用例从使用系统的角度描述系统中的信息

2.用例描述了用户提出的一些可见的需求

3.用例是对系统行为的动态描述

用例是有条件的, 如做饭的例子:

要做饭, 首先得有米, 这是启动用例的前提, 也称为前置条件; 用例执行完成后, 会得到一个结果: 米变成了饭, 这称为后置条件.

完整的用例定义由参与者, 前置条件, 场景, 后置条件构成.

3.6 用例分析

判断用例是否准确的依据:

1.用例是相对独立的. (这意味着不需要其他用例交互就可以独自完成参与者的目的. 如取钱是用例, 而填取款单不是)

2.用例的执行结果对参与者来说是可观测且有意义的. (如登陆系统是一个有效用例, 但输入密码不是)

3.用例必须由一个参与者发起(不存在没有参与者的用例, 用例也不应该自启动和主动启动另一个用例, 如从ATM取钱是用例,

而ATM吐钞不是)

4.用例通常是以动宾短语形式出现的.(如喝水 而"喝"和水不是)

5.用例就是一个需求单元, 分析单元, 设计单元, 开发单元, 测试单元, 甚至是部署单元, 一旦确定了用例, 软件开发工作的其他

活动都就以这个用例为基础, 围绕着它进行.

用例的粒度

在用例分析阶段,(概念建模阶段), 用例以能描述一项完整的业务流程为宜. 如取钱, 借书等, 但不要细到验证密码, 查找书上等单个步骤.

在系统分析阶段, 用例的视角是针对计算机的, 粒度以一个用例可以描述操作者与计算机的一次完整交互为宜. 如(填写申请单, 审核申请单…). 另一个要参考的粒度是一个用例的开发工作量在一周左右为宜.

在业务用例阶段, 最标准的方法是, 以该用例是否完成了参与者的某个完整的目的为依据. (如借书是用例, 而查询书目, 查询记录等只是完成这个目的的过程)

原则: 在同一个需求阶段, 所有用例的粒度应该是同一个量级.

用例的获取

需要清楚下面几个问题

1.参与者是位于系统边界之外的

2.参与者对系统有着明确的期望和回报等要求

3.参与者的期望和回报要求在系统边界之内

将每个有效的期望借助用例绘制出并命名就完成了用例获取的工作

目标和步骤的误区

在使用中, 对用例使用的另一个误区是: 混淆目标与完成目标的步骤.

一个用例是参与者对目标系统的一个愿望, 一个完整的事件. 为了完成这个事件需要经过很多步骤, 但这些步骤不能够完整地

反映参与者的目标, 不能作为用例

如寄信和付钱, 两个用例粒度不同, 边界不同, 它们显然不应该同时出现在一个视图里.

用例粒度的误区

产生用例粒度错误的原因首先是: 未分清目标和步骤

如系统管理管理网站和修改订单, 显然, 显然管理网站的粒度比修改订单的大.

在同一个需求阶段, 必须保持所有用例的粒度都在同一量级

业务用例

业务用例是用例版型中的一种, 专门用于需求阶段的业务建模

站在业务参与者的角度看到的将是业务边界而非系统边界

业务用例实现

业务用例实现, 也称业务用例实例, 是用例版型中的一种, 专门用于需求阶段的业务建模.

业务用例实现是业务用例的一种实现方式, 好比接口和实现类

业务用例实现的意义在于它表达了同一项业务的不同实现方式

如缴纳话费的三种实现方式: 营业厅缴费, 银行缴费, 预存话费

系统用例

系统用例是用来定义系统范围, 获取功能性需要的. 如果不是特别强调, 读者可以把用例等同于系统用例

用例实现

用例实现完整的叫法是系统用例的实现, 一个用例实现代表用例的一种实现方式

3.7 用例之间的关系

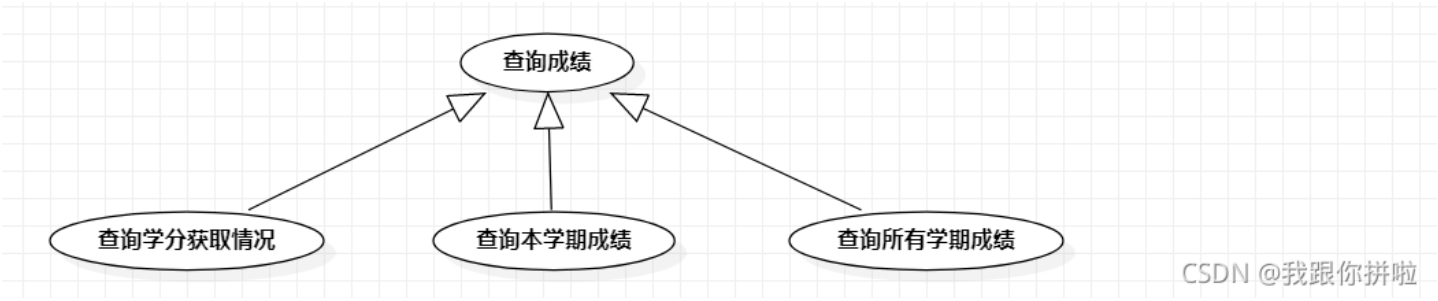
用例除了参与者之间有关联关系, 用例之间也存在着一定的关系, 如泛化关系, 包含关系, 扩展关系等.

也可以用UML的扩展机制自定义用例之间的关系

泛化关系(generalization)

在程序设计语言中, 与"继承"这个概念相似

在泛化关系中, 子用例继承了父用例的行为和含义, 子用例也可以增加新的行为和含义或者覆盖父用例的行为和含义



包含关系(include)

包含关系指的两个用例之间的关系, 其中一个用例(称作基本用例)的行为包含了另一个用例(称作包含用例)的行为

包含关系是依赖关系的版型

扩展关系(extend)

扩展关系基基本含义与包含关系类似, 也是依赖关系的版型, 也就是说扩展关系是特殊的依赖关系, 通常表示基本用例在特定情况下会用到扩展用例

与包含关系不同的是, 扩展关系的箭头指向是从扩展用例到基本用例

用例的泛化, 包含, 扩展关系的比较

包含关系: 在执行基本用例时, 一定会执行包含用例部分

扩展关系: 在执行基本用例时, 只有满足扩展点的要求时才执行扩展用例

关联(association): 参与者和用例之间的关系

泛化(generalization): 参与者之间或用例之间的关系

包含(include): 用例之间的关系

扩展(extend): 用例之间的关系

关联关系是两个或多个类元之间的关系. 这里所说的类元是一种建模元素, 常见的类元包括类, 参与者, 组件, 数据类型, 接口, 节点, 信号, 子系统, 用例等, 其中类是最常见的类元

泛化关系是两个类元之间的关系

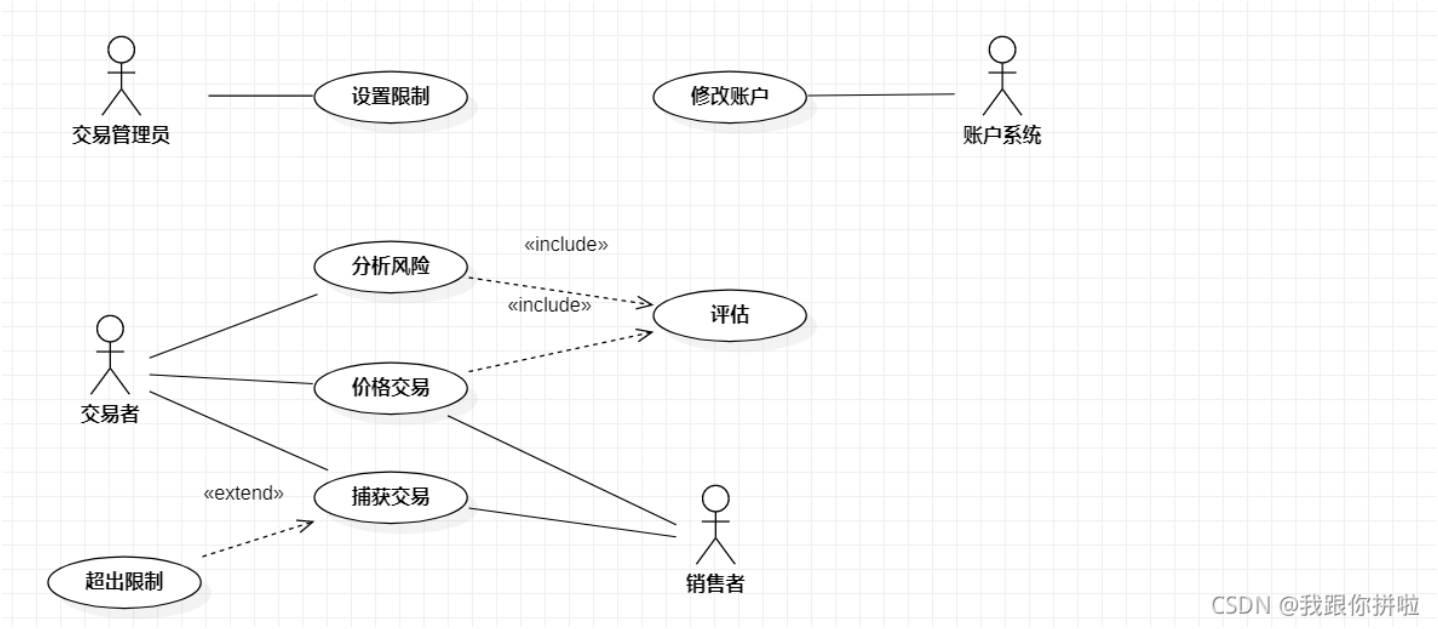
依赖关系表示的是两个元素或元素集之间的一种关系

包含关系和扩展关系都属于依赖关系

3.8 用例图

用例图是一组用例, 参与者以及它们之间的关系的图

如金融贸易系统用例图



3.9 用例的描述

事实上, 用例描述才是用例的主体部分

用例的描述一般应包括以下内容:

- 用例的目标
- 用例是怎么启动的
- 参与者和用例之间的消息是如何传送
- 用例除了主路径之外, 其它路径是什么
- 用例结束后的系统状态
- 其它需要描述的内容

总之, 描述用例的原则是尽可能写得充分

用例模板(略)

初学者在描述用例时易犯的错误:

1. 只描述系统的行为, 没有描述参与者的行为
2. 只描述参与者的行为, 没有描述系统的行为
3. 描述冗长

在描述用例时, 可以采用更为简洁的描述方式, 如合并数据项, 提供抽象的高层描述

正确的用例描述

Use Case: 购买东西

1. 顾客使用ID和密码进入系统
2. 系统验证顾客身份
3. 顾客提供个人信息(姓名, 地址, 电话), 并选择购买的商品及数量
4. 系统验证顾客是否为老顾客
5. 系统使用库存系统验证要购买的商品数量是否少于库存量
6. …(省略)

3.10 寻找用例的方法

用例分析的步骤:

1. 找出系统外部的参与者和外部系统, 确定系统的边界和范围
2. 确定每一个参与者所期望的行为
3. 把这些系统行为命名为用例
4. 使用包含, 泛化, 扩展等关系处理系统行为的公共或变更部分
5. 编制每一个用例的脚本
6. 绘制用例图
7. 区分主事件流和异常事件流, 如果有需要, 可以把异常事件流当作单独用例来处理
8. 细化用例图, 解决用例间的重复与冲突问题

3.11 建模实例

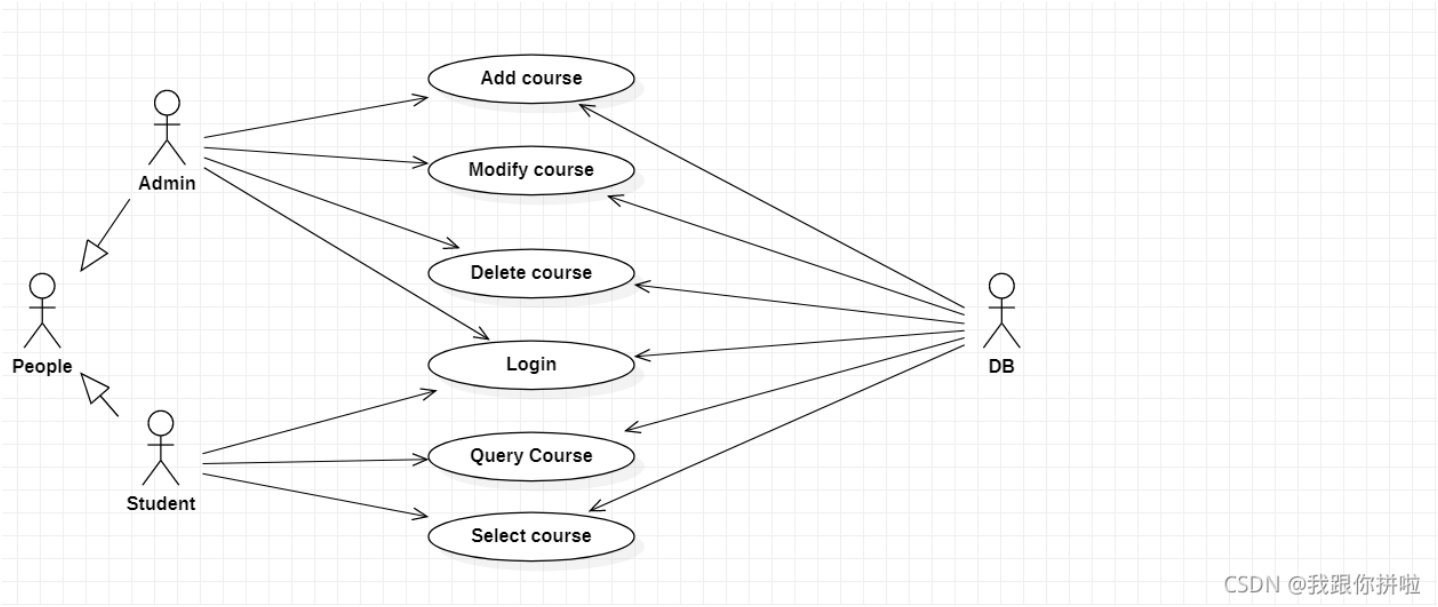
我校打算开发一个网上选课系统, 需求如下:

管理员通过系统管理界面验证身份进入系统, 建立本学期要开设的各种课程, 将课程信息保存在数据库中并可以对系统进行修改和删除. 学生通过客户机根据学号和密码进入选课界面, 在这里学生可以进行两种操作: 查询已选课程, 选课. 同样, 通过业务层, 这些操作会存入数据库中. 数据库部署在服务器上, 通过一个开放的sql接口操作数据库

确定参与者: 管理员, 学生, 数据库代理

识别用例:

1. 学生: 用户登录 查询已选课程 选课 付费
2. 管理员: 用户登录 增加课程 修改课程 删除课程
3. 数据库代理: 所有和数据库操作相关的用例



Add course的用例描述:

Documentation

1.学生进入选课系统,用例开始
2.系统提示输入学号和密码
3.学生输入学号和密码
4.系统验证
A1: 验证失败
5.进入选课主界面
6. 学生点击选课
7.系统显示所有课程信息
8. 学生选择课程
9. 系统验证课程是否可选
A2: 课程不可选

Documentation

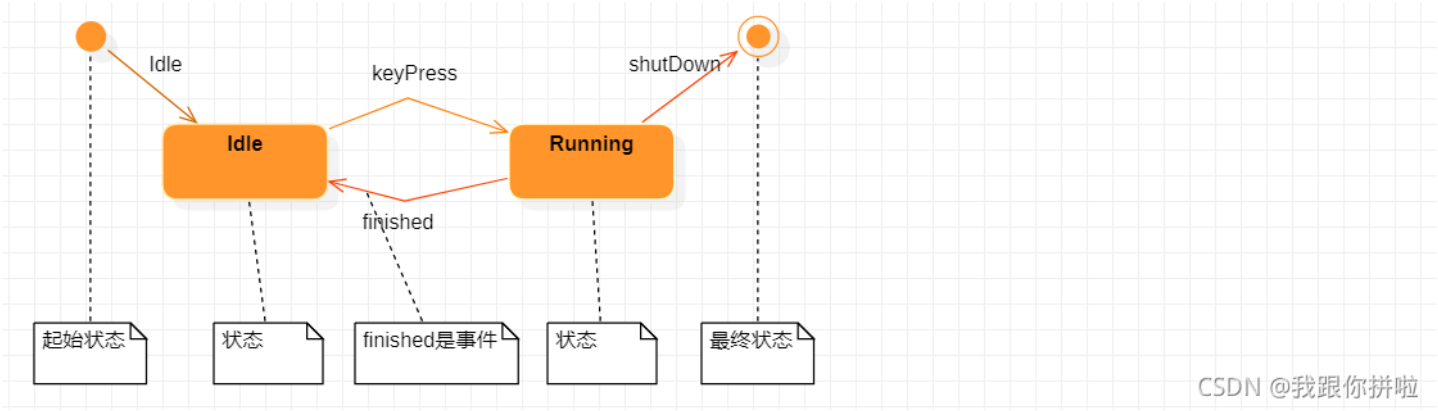
1.学生进入选课系统,用例开始
2.系统提示输入学号和密码
3.学生输入学号和密码
4.系统验证
A1: 验证失败
5.进入选课主界面
6. 学生点击选课
7.系统显示所有课程信息
8. 学生选择课程
9. 系统验证课程是否可选
A2: 课程不可选

CSDN @我跟你拼啦

第四章 状态图和活动图

4.1什么是状态图

1. 状态图(statechart diagram)主要用来描述描述一个对象在其生存期间的动态行为,表现一个对象所经历的状态序列,引起状态转移的事件(event),以及因状态转移而伴随发生的动作(action)
2. 一般可以状态机对一个对象 (这里的对象可以是类的实例,用例的实例或整个系统的实例) 的生命周期建模
3. 状态图是用于显示状态机的,重点在于描述状态之间的控制流



4. 在状态机中, 动作既可以与状态相关也可以与转移相关. 如果动作是状态相关, 则对象在进入一个状态将触发某一动作, 而不管是从哪个状态进入这个状态的. 如果动作是与转移相关的, 则对象在不同的状态转移时, 将触发相应的动作
5. Moore机: 所有的动作都与状态有关; Mealy机: 所有的动作都与转移有关.

4.2 状态图的基本概念

状态

状态(state)是指对象的生命周期中的某个条件或状况, 在此期间对象将满足某些条件, 执行某些活动或等待某些事件.

所有对象都有状态,状态是对象执行一项或多项活动的结果, 当某个事件发生后, 对象的状态将发生变化

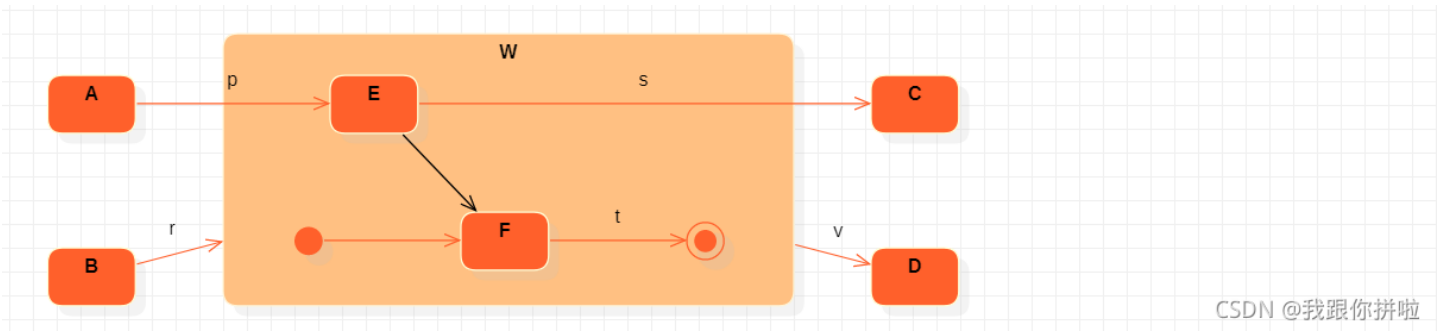
状态可以细分为不同的类型, 如初态, 终态, 中间状态, 组合状态, 历史状态等. 一个状态图只有一个初态, 但但终态可以有一个或多个, 也可以没有终态

中间状态包括两个区域: 名字域和内部转移域

组合状态和子状态

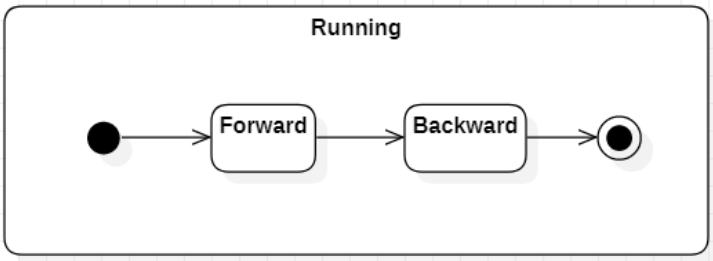
嵌套在另一个状态的状态称作子状态(substate), 含有子状态的状态称作组合状态(composite state).

如图W是组合状态, E和F是子状态

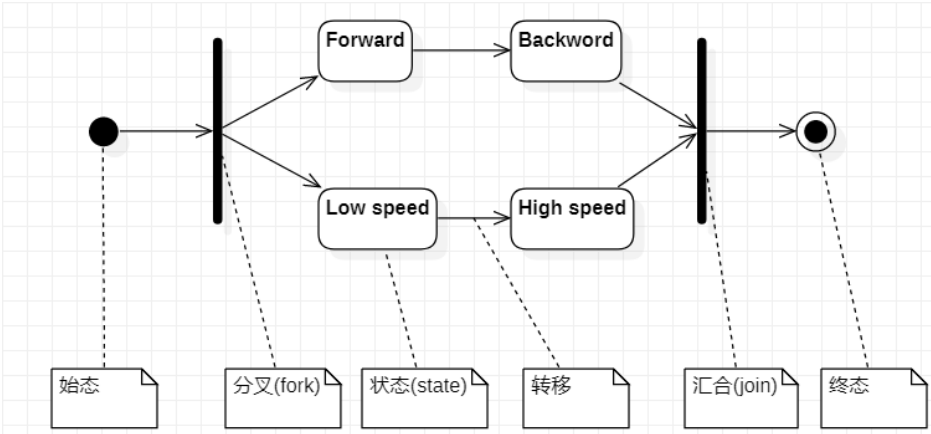


子状态可以分为or关系和and关系

1. or关系: 在某一时刻可到达一个子状态



2. and关系: 组合状态中在某一时刻可同时到达多个子状态



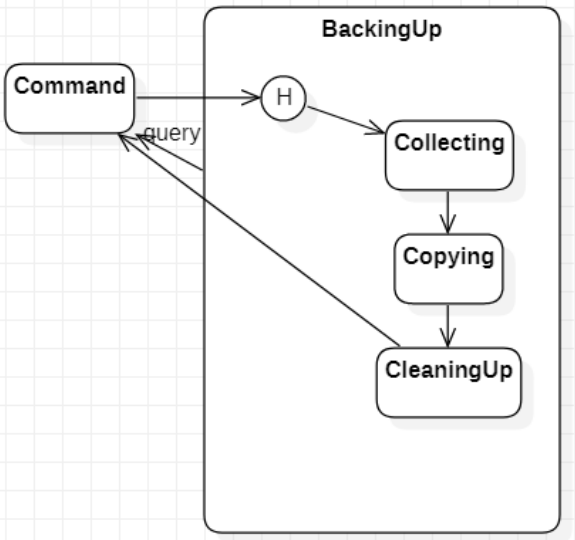
历史状态

历史状态(history state)是一个伪状态, 其目的是记住从组合状态中退出时所处的子. 当两次进入组合状态时, 可以直接进入该子状态, 而不是再次从组合的初态开始

历史状态用符号

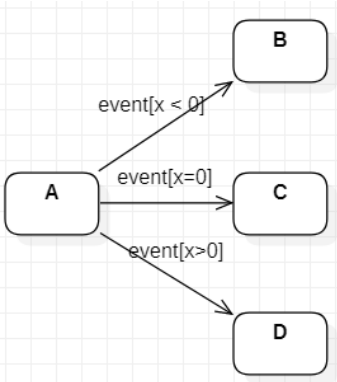
(浅历史状态shallow)和
(深历史状态deep)

浅历史状态表示只记住最外层组合状态的历史; 而深历史状态表示可以记住任何深度的组合状态的历史



转移(transition)

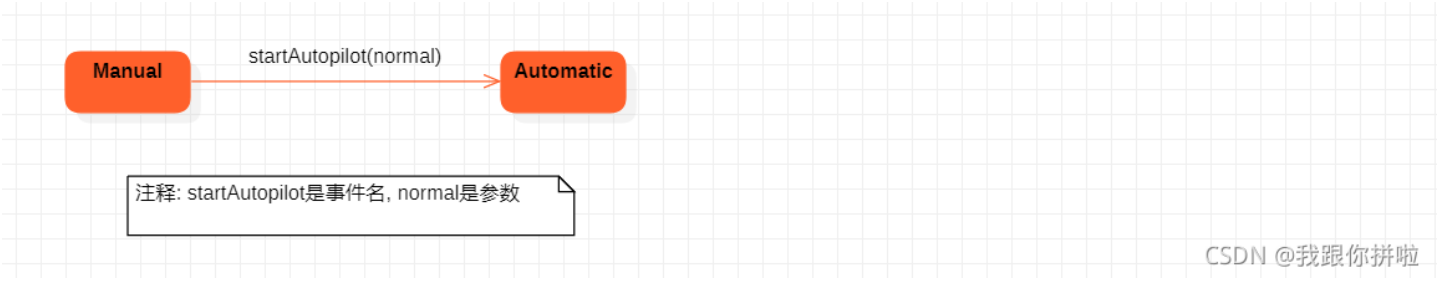
1. 转移是两个状态之间 的一种关系, 表示对象将在第一个状态中执行一定的动作, 并在某个特定事件发生而且某个特定的警戒条件满足时进入第二个状态
2. 一般来说, 状态之间的转移是由事件触发的, 因此应在转移上标出触发事件的表达式. 如果转移上未标明事件, 则表示原状态的内部活动执行完毕后自动触发转移
3. 对于一个给定状态, 最终只能产生一个转移, 所以从相同的状态出来的, 事件相同的几个转移之间的条件应该是互斥的.



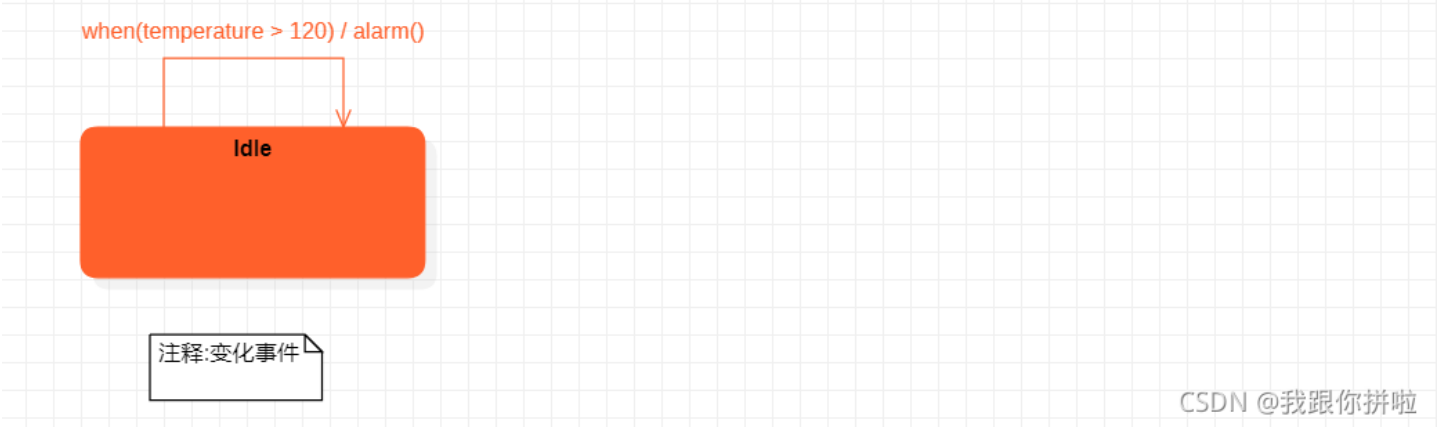
事件(event)

事件是对一个在时间和空间上占有一定位置的, 有意义的事情的详细说明.

1. 调用事件(call event): 表示对操作的高度



2. 变化事件(change event): 如果一个布尔表达式的变量发生变化, 使得布尔表达式的值发生相应的变化从而满足某些条件, 则称这种事件为变化事件. 变化事件表示的是一个要被不断测试的事件. 变化事件用关键字when表示



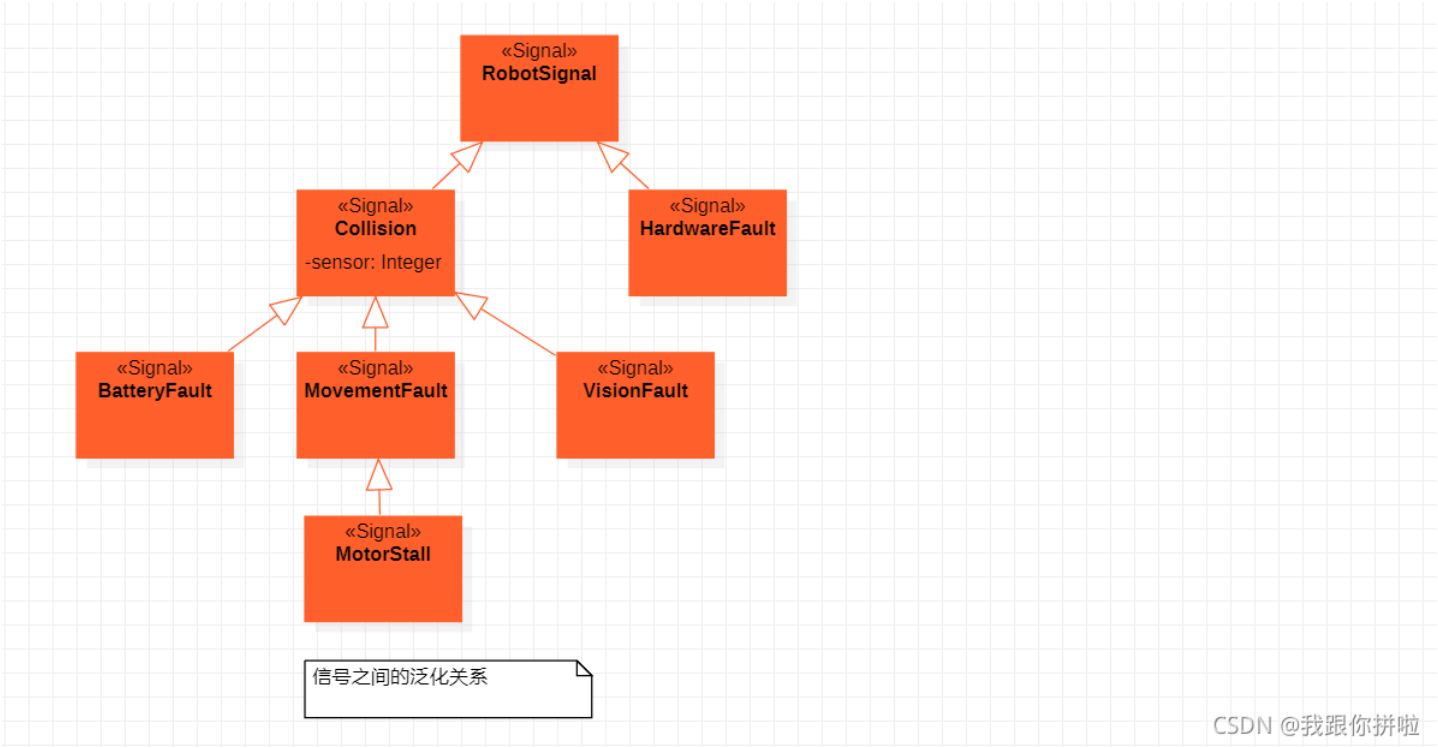
3. 时间事件(time event): 指的是满足某一时间表达式的情况出现, 如到了某一时间点或经过了某一段时间. 时间关键字用after或when表示

信号事件(signal event): 表示的是对象接收到了信号这种情况, 信号事件往往会触发状态的转移

"信号"就是由一个对象异步地发送并由另一个对象接收的, 已命名的对象

信号版型用<>表示, 信号之间可以有泛化关系, 形成层次结构

信号事件和调用事件相似, 介信号事件是异步事件, 调用事件一般是同步事件



动作

动作是一个可执行的原子计算, 不可中断, 其执行时间忽略不计.

uml规定了两种特殊的动作: 进入动作和退出动作. 进入动作表示进入状态时执行的动作, 退出动作表示退出状态时执行的动作

```
entry / setMode(onTrack)
exit / setMode(offTrack)
```

4.3 什么是活动图

活动图可以用于描述系统的工作流程和并发行为

活动其实可以看作状态图的一种特殊形式, 其中一片活动结束后立即进入下一个活动(在状态图中的状态转移可能需要事件的触发)

活动

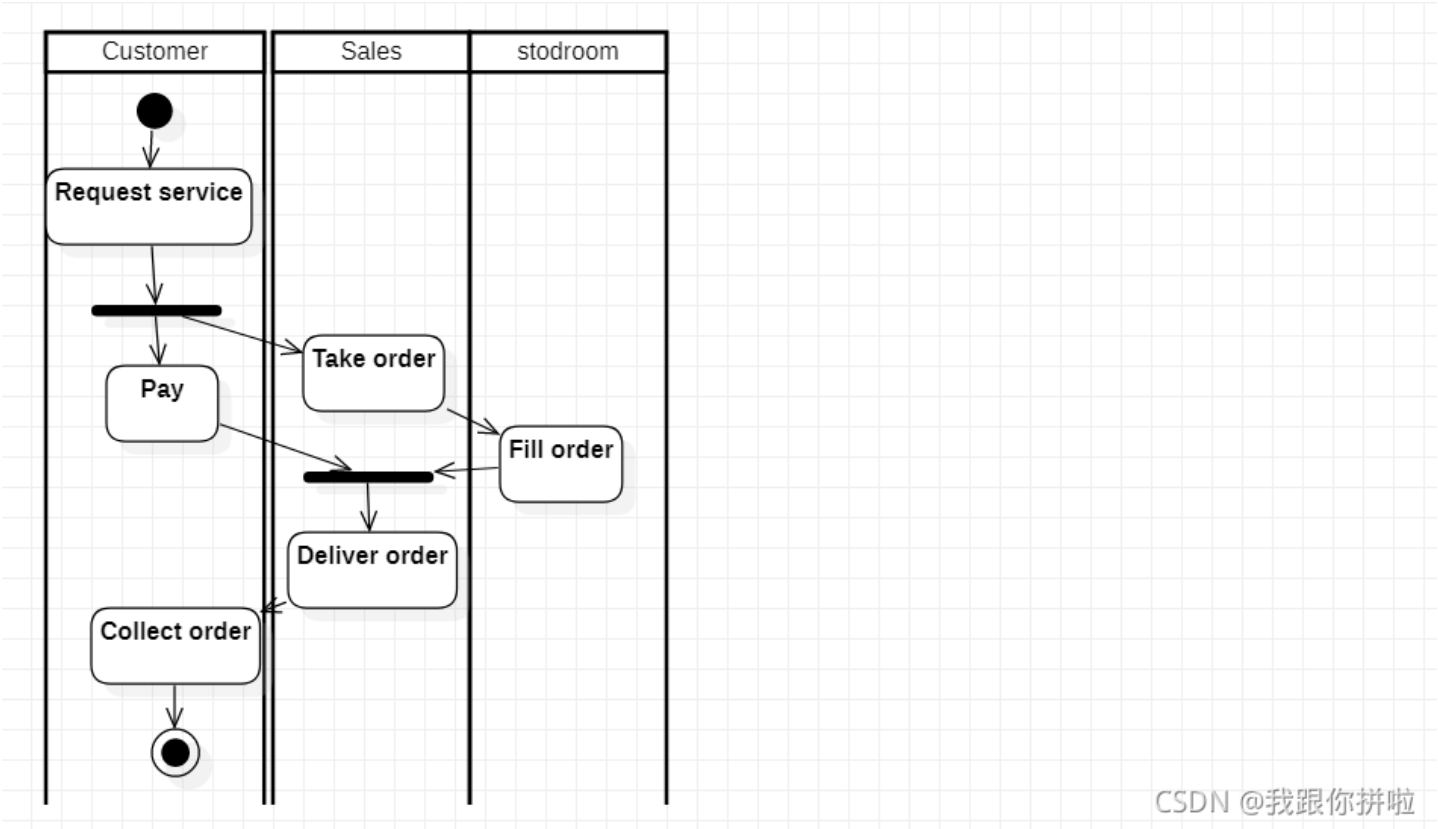
活动表示某流程中的任务执行, 它可以表示某算法过程中语句的执行

动作状态和活动状态的区分:

1. 动作状态是原子性的, 不能被分解, 其工作所占用时间可以忽略. 动作状态的目的是"进入动作", 然后转向另一个状态
2. 活动状态是可分解的, 不是原子性的, 其工作的完成需要一定的时间, 可以把动作状态看作活动状态的特例

泳道

泳道(swimlane)是活动图中的区域划分, 系统根据每个活动的职责对所有活动进行划分, 每个泳道代表一个责任区. 每个泳道并不是一一对应关系, 泳道关心的是其所代表的职责, 一个泳道可能由一个类实现, 也可能由多个类实现



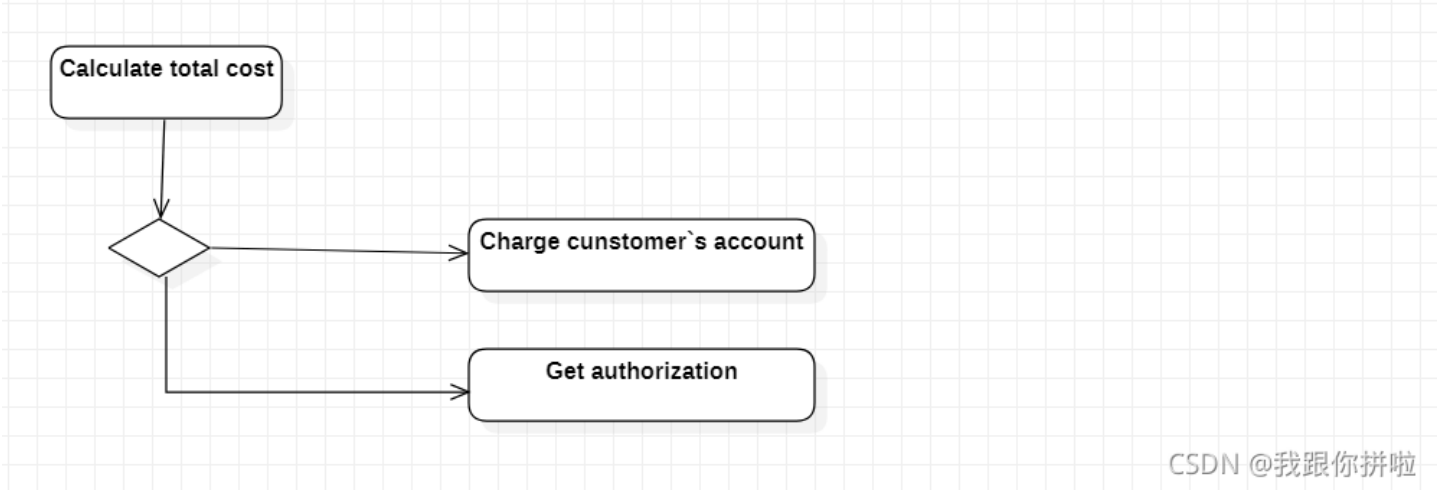
分支

在活动图中, 对于同一个触发事件, 可以根据不同的触发条件转向不同的活动, 每个可能的转移是一个分支(branch)

两种表示方法:

常规方法

菱形符号表示

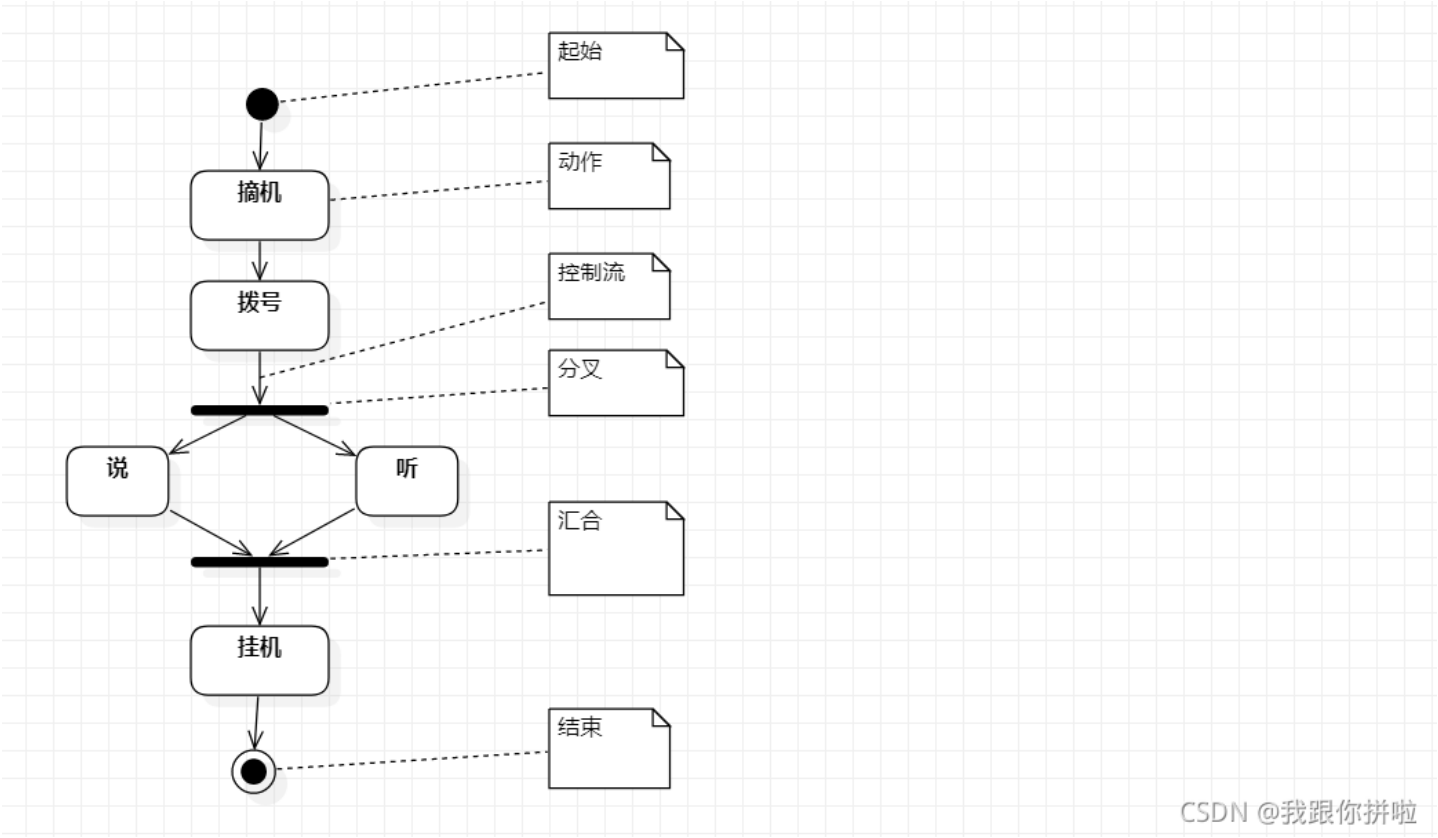


分叉和汇合

分叉表示的是一个控制流被两个或多个控制流代替, 经过分叉后, 这些控制流是并发进行的

汇合正好与分叉相反, 表示两个或多个控制流 合并为一个控制流

如打电话的例子

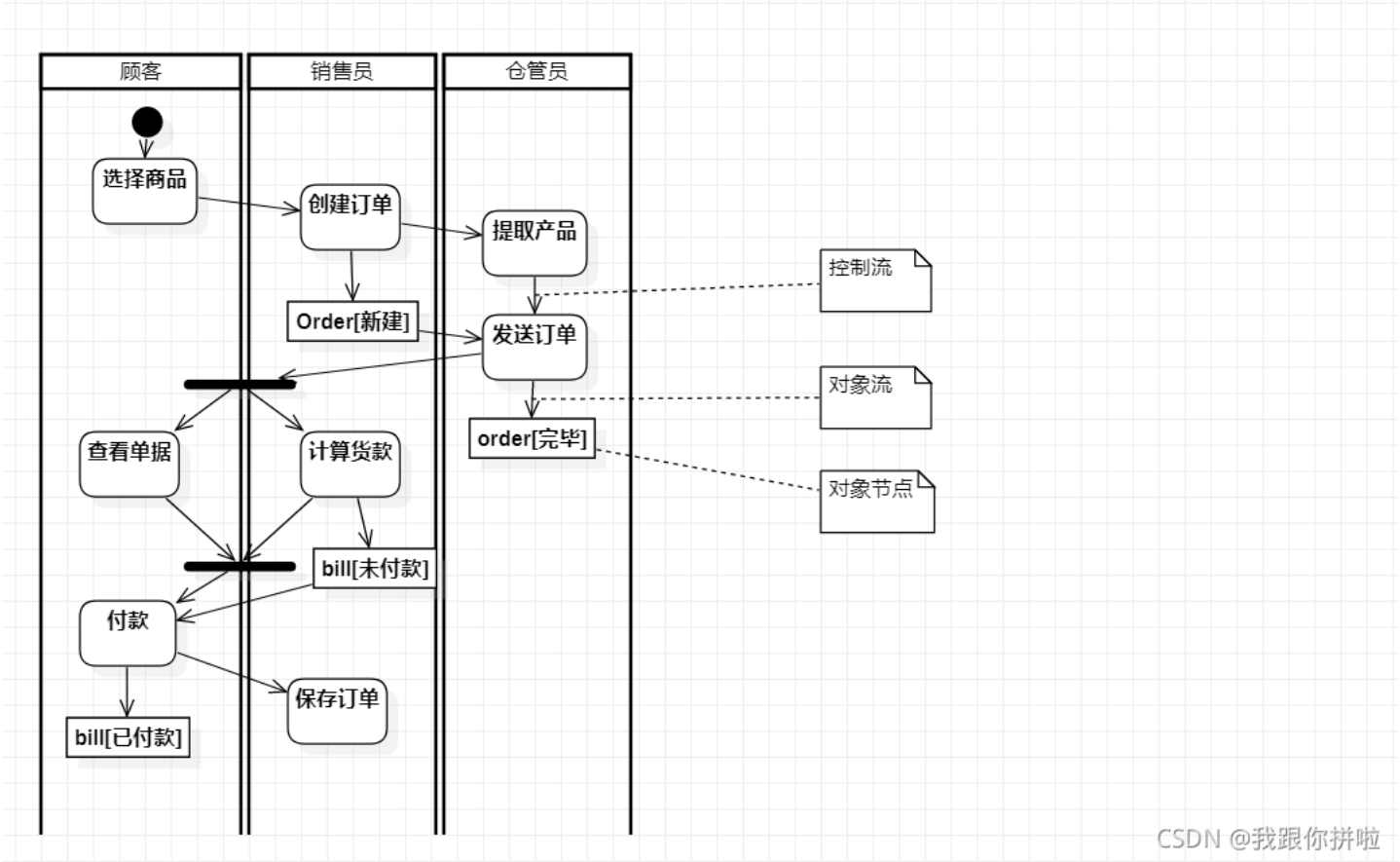


对象流

在活动图可以出现对象, 对象可以作为活动的输入或输出, 活动图中对象流表示活动和对象之间的关系, 如一个活动创建对象

(作为活动的输出)或使用对象(作为活动的输入)

对象流属于控制流. 如果两个活动之间有对象流, 则控制流不必重复画出



4.4 活动图的用途

活动图对表示并发行为很有用

一般活动图可以对系统的工作流程建模, 即对系统的业务建模

可以对具体的操作建模, 用于描述计算过程的相关细节

在进行用例分析时, 可以用活动图来描述用例的内部流程

开发人员往往用一个流程图来描述一个算法. 在UML中没有流程图的概念, 从某种意义上来说, 活动图的功能已包含了流程图

4.5 状态图和活动图的比较

状态图和活动图都是对系统的动态行为建模, 两者相似, 但也有区别

首先, 两者描述的重点不同. 状态图描述的是对象的状态及状态之间的转移, 而活动图描述的是从活动到活动的控制流

其次, 两者使用的场合不同. 如果是为了显示一个对象在其生命周期内的行为, 则使用状态图较好; 如果目的是分析用例, 理解涉及多个用例的工作流程, 或者使用多线程应用等, 则使用活动图较好

如果要显赫多个对象之间的交互情况, 用状态图和活动图都不合适, 这里可以用顺序图或协作图来表示

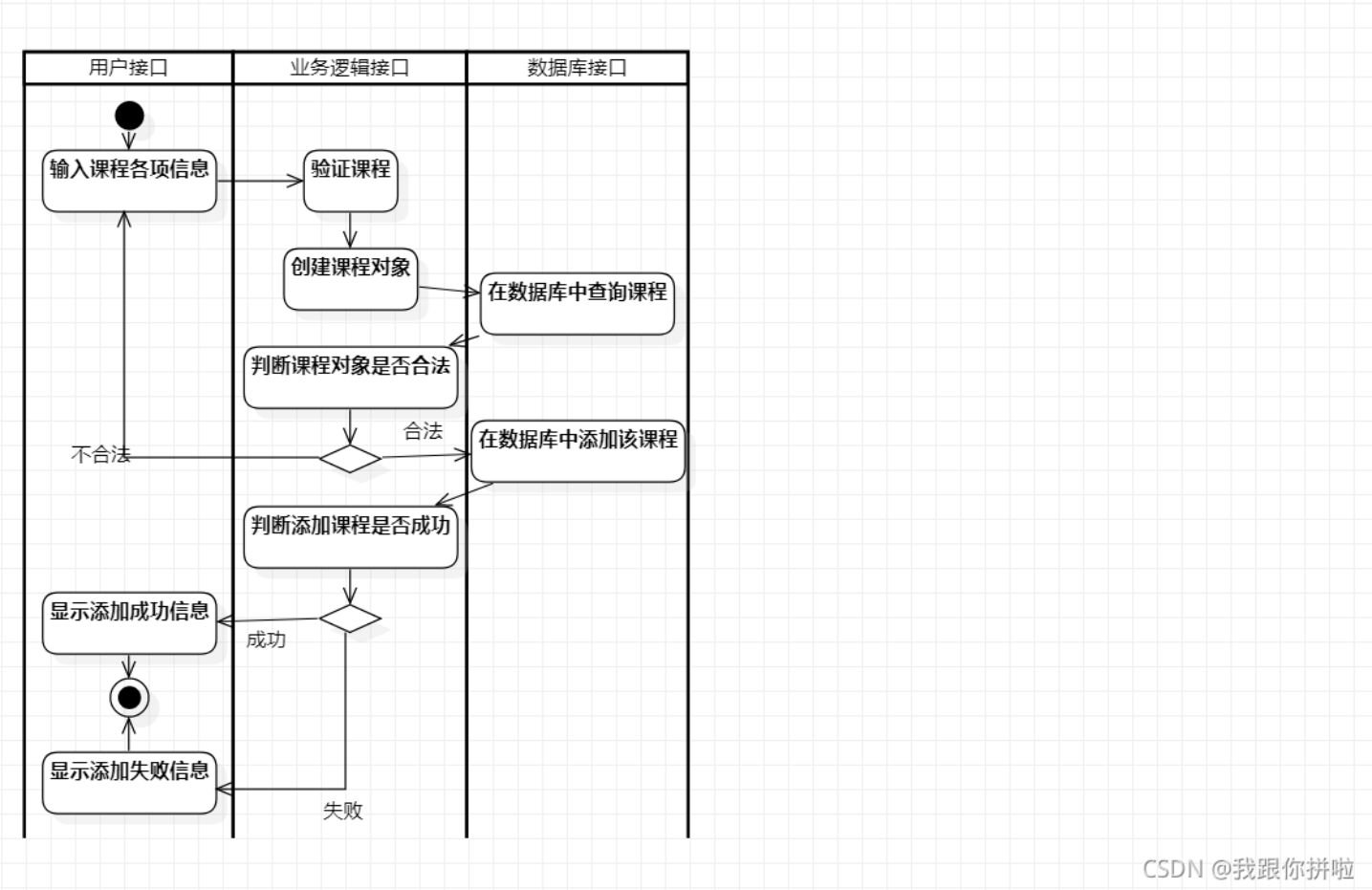
4.6 建模实例

用例图的根本意义在于明确系统的需求, 为后面的系统分析与设计奠定基础.

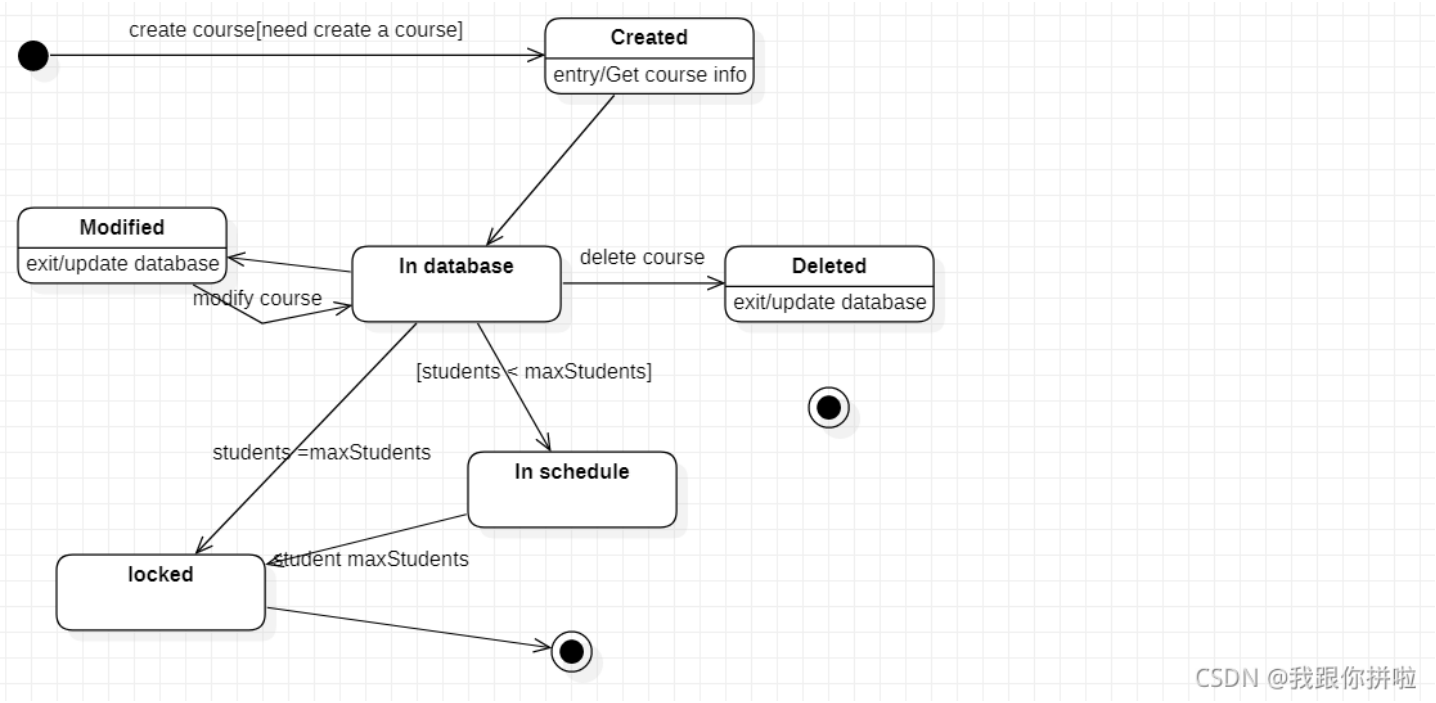
而用例文档(事件流)的作用则是对用例内部流程进行刻画, 是对用例的补充说明

活动图的作用其实和用例文档是异曲同工的, 只不过它们是两种不同的形式, 面向的受众也不同

下面采用活动图来描述课程系统中的"添加课程"用例的事件流.



course状态图



第五章 类图和包

5.1 类的定义

在UML中, 有两种图非常重要, 一种是用例图, 另一种是类图

类的定义: 类是具有相似结构, 行为和关系的一组对象的描述符

类的命名分为两种形式, simple name和path name(如Util::shape, util是包名, Shape是util中的一个类)

类的属性

属性(attribute)是已被命名的类的特性, 它描述了该特性的实例可以取值的范围

属性描述了正被建模的事件的一些特性, 这些特性是类的所有对象所共有的

属性的可见性:+ - #

```
+name: int  // +号表示public
-age: int   // -号表示private
#state: boolean  // #号表示protected
```

类的操作

操作(opeartion)用于修改, 检索类的属性或执行某些动作, 操作通常也称为功能, 但是它们被约束于类的内部, 只能作用到该

类的对象上

```
+display():Location
-hide()
#create()
```

5.2 类之间的关系

一般来说, 类之间的关系有关联, 聚集, 组合, 泛化, 依赖等

关联

关联(association)是模型元素间的一种主义联系, 它是对具有共同的结构特征, 行为特性, 关系和语义的链(link)的描述

链的概念: 链是一个实例, 就像对象是类的实例一样, 链是关联的实例, 关联表示的是类与类之间的关系, 而链表示的是对象与对象之间的关系

在类图中, 关联用一条把类连接在一起的实线来表示

一个关联可以有两个或多个关联端(association end), 每个关联端连接到一个类

关联也可以有方向

双向关联

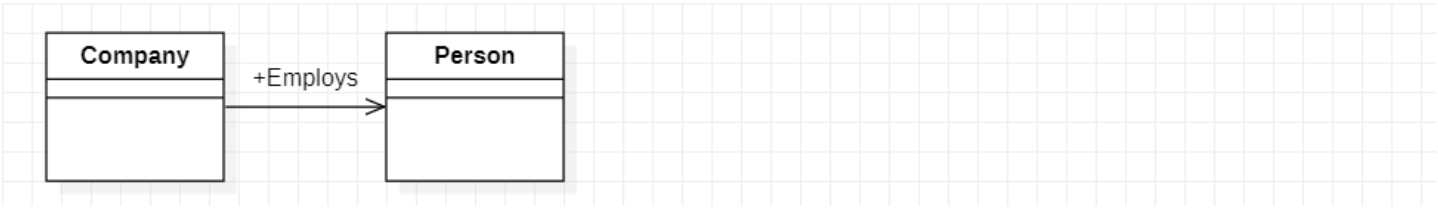
单向关联


```
// 关联关系
public class A {
 public B b;
}
```

```
public class B {
 String name;
 int age
 ...
}
```

(1).关联名

可以给关联加上关联名来描述关联的作用



如果一个关联表示的含义已经足够明确, 则无需加上关联名

(2). 关联的参与者

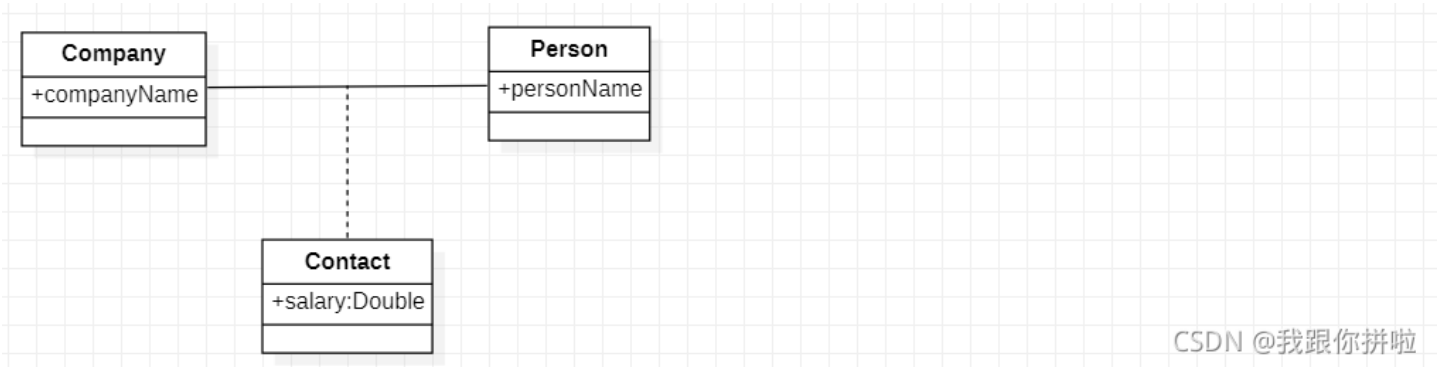
关联两端的类可以以某种参与者参与关联

参与者还具有多重性, 表示可以有多少个对象参与该关联

(3). 关联类

关联本身也有特性, 通过关联类(association class)可以进一步描述关联的属性, 操作以及其他信息

关联类通过一条虚线与关联连接




```
// Company类
public class Company {
    private String companyName;
    public Person employee[]; // 员工数组
}
```

```
//  Person类
public  class Person {
    private String personName;
    protected Company employer; // 公司老板

    // 类"contract"的代码
    pubic class Contract {
        private Double salary; // 薪资
    }
}
```

聚集和组合

聚集(aggregation)是一种特殊形式的关联. 如教室和桌子, 不是强绑定关系, 可以拆开独立存在

组合(composition)表示的也是类的整体与部分的关系, 但整体和部分具有同样的生命周期. 也就是说组合是一种特殊形式的

聚集. 例如: 脑袋和嘴巴的关系

口诀: 聚集可拆开, 组合不分割

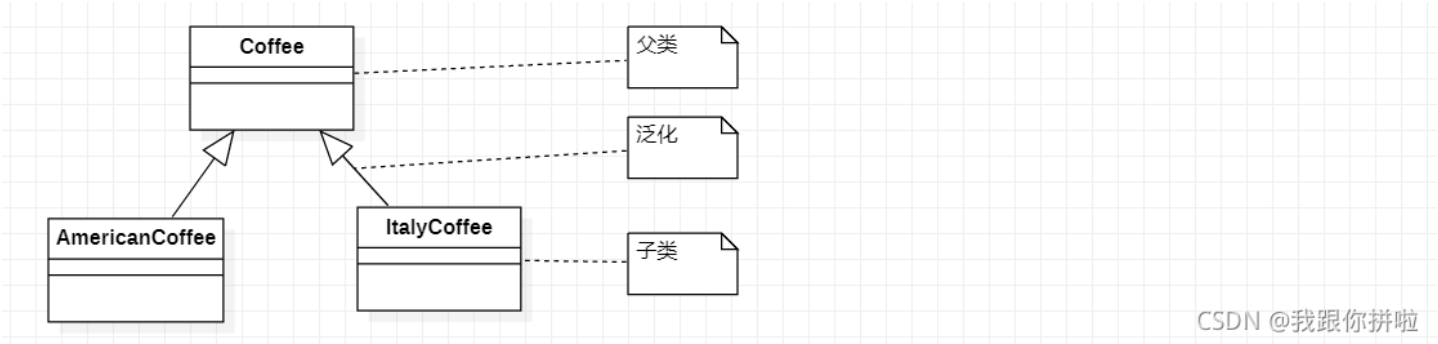
设计人员根据需求分析描述来确定是使用aggregation还是compostion

泛化

泛化(generalization)定义了一般元素和特殊元素之间的分类关系, 如果从面向对象程序设计语言的角度来说, 类与类之间的

泛化关系就是类与类的继承关系

UML中用空心三角形的连线表示泛化关系



依赖

假如有两个元素X和Y, 如果修改X的定义可能会导致另一个Y的定义的修改, 则称Y依赖于X

对于类而言, 依赖(dependency)可能由各种原因引起, 如一个类向另一个类发送消息, 或者一个类是另一个类的数据成员, 或者一个类是另一个类的操作的参数类型等

有时依赖关系和关联关系比较难以区分. 事实上, 如果类A和类B之间有关联关系, 那么类A和类B也就有依赖关系了. 但如果

两个类之间有关联关系, 只需表示出关联关系即可, 不用再表示这两个类之间有依赖关系.

与关联关联关系不同的是, 依赖关系本身不生成专门的实现代码

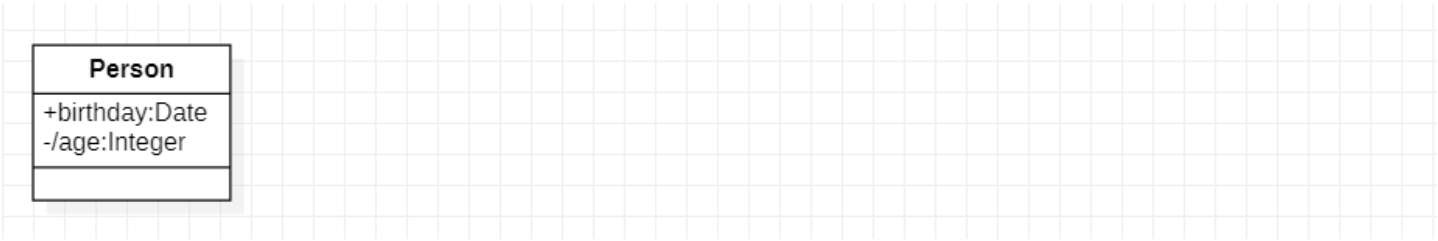
另外, 与泛化关系类似, 依赖关系也不仅仅限于类之间, 其它建模元素也有

5.3 派生属性和派生关联

派生属性(derived attribute)和派生关联(derived association)是指从其它属性和关联计算推演得到的属性和关联

如Person类的age属性即为派生属性, 因为一个人的年龄可以从当前日期和其出生日期推算出来.

在类图中, 派生关联的名字前需要加一个"/"



在生成代码时, 派生属性和派生关联不产生相应代码.

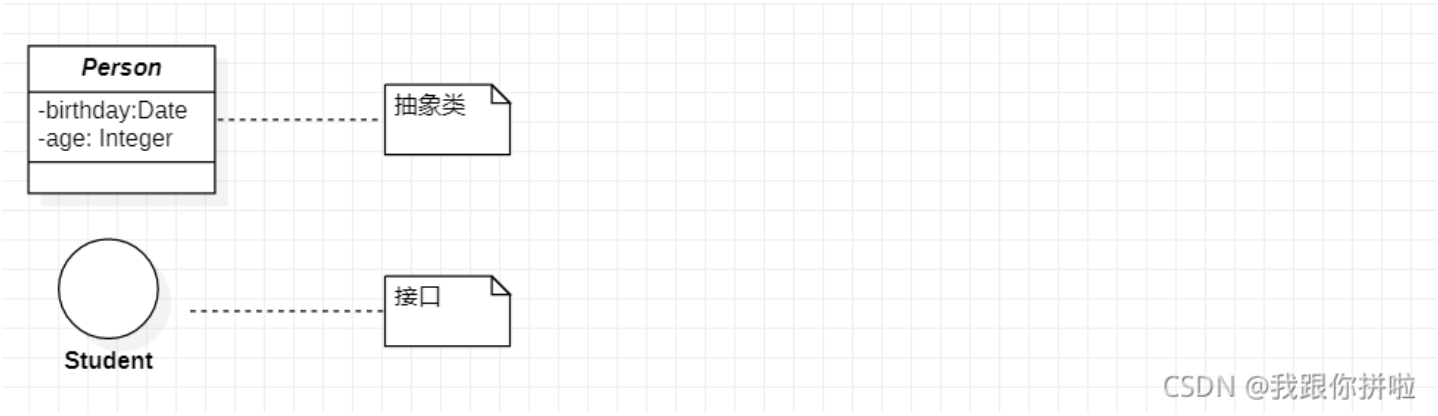
指明某些属性和关联是派生属性和派生关联有助于保障数据的一致性

5.4 抽象类和接口

抽象类(abstract class)是不能直接产生实例的类, 因为抽象类中的方法往往只是一些声明, 而没有具体的实现, 因此不能对抽象类实例化

UML中通过把类写成斜体来表示

接口是类的<>版型



面试官: 请你谈谈对java抽象类和接口的理解

接口和抽象类都是继承树的上层，他们的共同点如下：

1. 都是上层的抽象层。
2. 都不能被实例化
3. 都能包含抽象的方法，这些抽象的方法用于描述类具备的功能，但是不提供具体的实现。
   他们的区别如下：
4. 在抽象类中可以写非抽象的方法，从而避免在子类中重复书写他们，这样可以提高代码的复用性，这是抽象类的优势；接口中只能有抽象的方法。
5. 一个类只能继承一个直接父类，这个父类可以是具体的类也可是抽象类；但是一个类可以实现多个接口。
   Java语言中类的继承是单继承原因是：当子类重写父类方法的时候，或者隐藏父类的成员变量以及静态方法的时候，JVM使用不同的绑定规则。如果一个类有多个直接的父类，那么会使绑定规则变得更复杂。为了简化软件的体系结构和绑定机制，java语言禁止多继承。
   接口可以多继承，是因为接口中只有抽象方法，没有静态方法和非常量的属性，只有接口的实现类才会重写接口中方法。因此一个类有多个接口也不会增加JVM的绑定机制和复杂度。
   对于已经存在的继承树，可以方便的从类中抽象出新的接口，但是从类中抽象出新的抽象类就不那么容易了，因此接口更有利于软件系统的维护和重构。

图中如果Sparrow继承类Bird 类 ， Boyin继承Airplane类，Sparrow 和Boyin想使用同样的flay方法那么是没有办法实现的，因为类的继承是单继承。

5.5 版型

版型(stereotype)是UML的3种扩展机制之一, UML中的另外两种扩展机制是标记值(tagged value)和约束(constraint)

UML中定义了一些版型, 如包的版型有子系统等, 类的版型有接口, 参与者, 边界类, 控制类, 实体等, 当然, 用户也可以自己定义版型

5.6 类图

类加上它们之间的关系构成了类图, 类图中可以包含接口, 包, 关系等建模元素, 也可以包含对象, 链等.

可以说, 类图描述的类与类之间的静态关系.

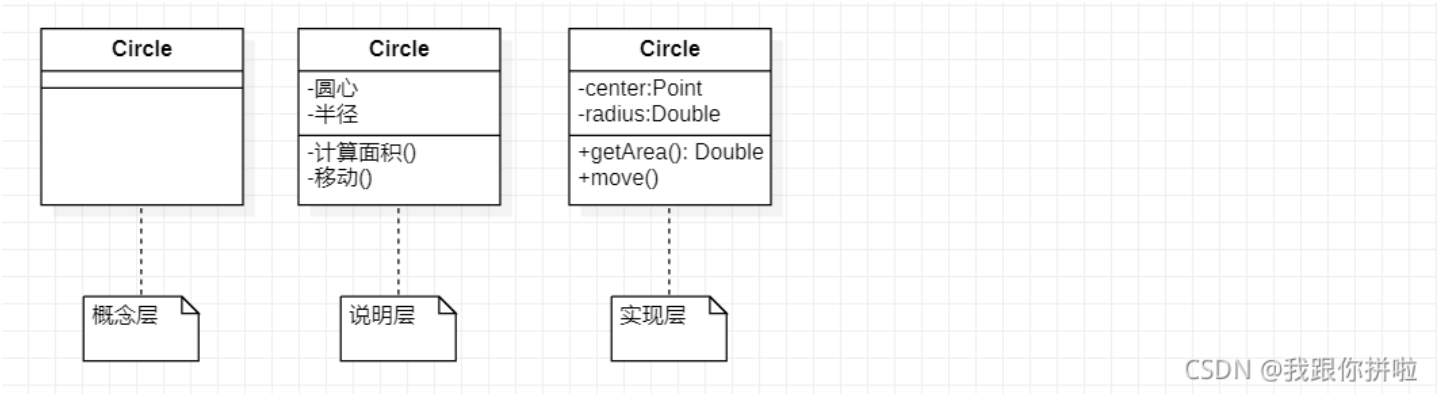
与数据模型不同的是, 类图不仅显示了信息的结构, 同时还描述了系统的行为

类图的抽象层次

在软件开发的不同阶段使用的类图具有不同的抽象层次, 一般类图可分为3个层次

1. 概念层(conceptual)类图描述应用领域中的概念. 画概念层类图时, 很少考虑或不考虑实现问题, 因此, 概念层类图独立于具体的程序设计语言
2. 说明层(specification)类图描述软件的接口部分, 而不是软件的实现部分
3. 实现层(implementation)类图都是真正考虑类的实现问题, 提供类的实现细节的部分

如圆类



概念层类图只有一个类名

说明层类图有类名, 属性名和方法名, 但对属性没有类型的说明, 在描述上常常采用接近现实世界的语言

实现层类图则对类的属性和方法都作了详细的说明, 实现层类图可能是大多数人最常用的类图.

可以用版型<>说明一个类是实现层, 用<>说明一个类是说明层或概念层, 当然, 也可

以不使用版型特别地指明

构造类图

> 根据用例描述中的名词确定类的候选者. 比如, 从事件流中寻找名词或名词词组, 将性质相同的归为一类, 或者将内容正负相反的归为一类. 在事件流中, 名词可以分为4种类型: 参与者, 类, 类属性和表达式
>
> 使用CRC分析法寻找类. CRC是类(class), 职责(responsibility)和协作(collaboration)的简称. CRC分析法根据类所要扮演
>
> 的职责来确定类
>
> 根据边界类, 控制类和实体类的划分来帮助发现系统中的类. 对领域进行分析, 或者利用已有的领域分析结果得到类
>
> 参考设计模式来确定类. 如工厂模式
>
> 根据某些软件开发过程提供的指导原由进行寻找类的工作

构造类图时不要过早陷入实现细节, 就该根据项目开发的不同阶段, 采用不同层次的类图

如果处于分析阶段, 应画概念层类图; 当开始着手软件设计时, 应画说明层类图;当考察某个特定的实现技术时, 则应画实现层

类图

5.7 包的基本概念

包就像一个容器, 可用于组织模型中的相关元素以使其更易于理解

包中可以包含其它建模元素, 如类, 接口, 构件, 节点, 用例, 包等

包中元素也可进行可见性控制

+表示public -表示private #表示protected

AWT包中有三个元素:

1. 其中window的可见性为public, 表示任何导入AWT包的包中, 都可以引用window元素
2. Form可见性为protected, 表示只有AWT包的子包才可以引用Form元素
3. EvenHandler的可见性为private, 表示只有在AWT包中才可以引用EventHandler元素

对包的命名有两种方式, 简单包名和路径包名. 如Sensors::Vision

下图的依赖关系的版型都是<>, 表示源包会存取目的包中的内容, 同时目的包中的内容是加到源包的名字空间的,

这样引用目的包中的内容就不需要加包名限定, 直接用目的包中的元素名字即可

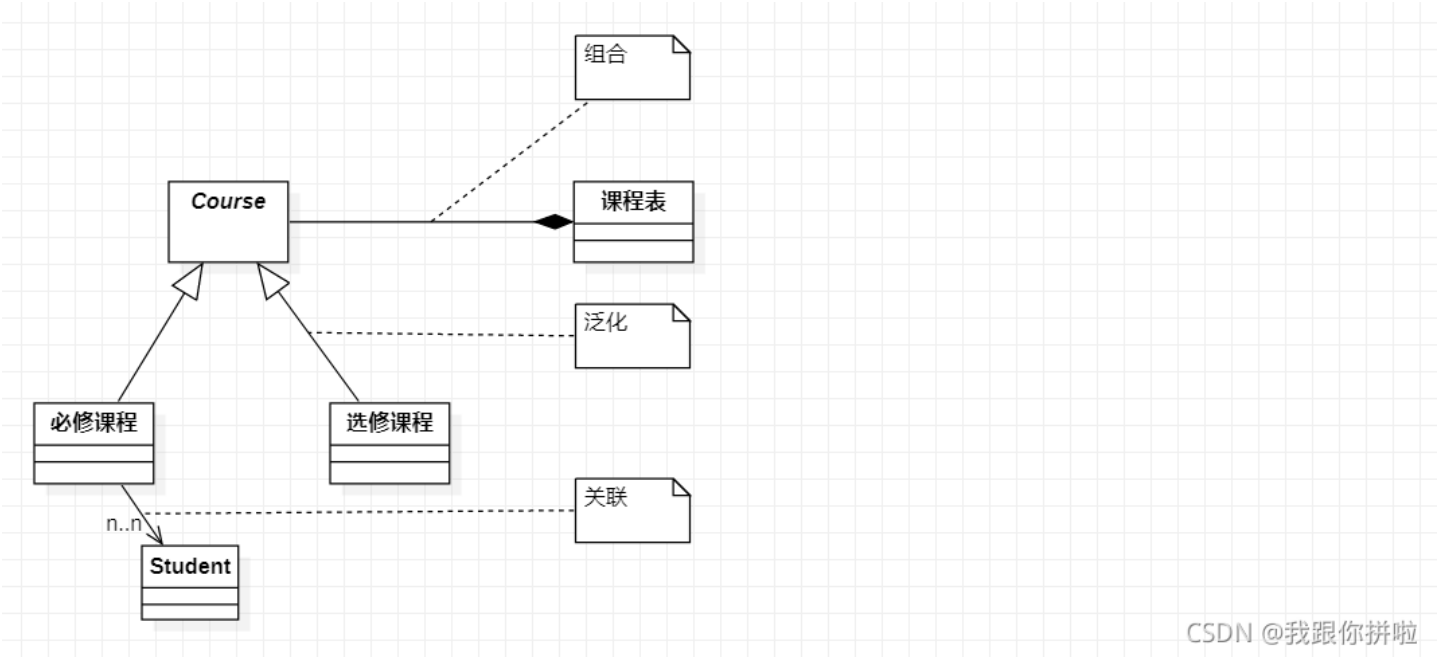
[外链图片转存失败,源站可能有防盗链机制,建议将图片保存下来直接上传(img-x855TN2y-1631930294569)(image-20210615165500840.png)]

在数据建模中, 用包表示模式和域, 在数据模型和对象模型之间转换是以包为单位进行的

在Web建模中, 包可以表示某一虚拟目录(virtual directory), 在该目录下的所有Web元素都在这个包中

5.8 建模实例

以课程为例, 介绍采用StarUml创建Course类的过程



第六章 交互图: 用例的实现

6.1 交互图概述

在UML中, 用例的实现用交互图来指定和说明

交互图通过显示对象之间的关系和对象之间处理的消息来对系统的动态特性建模

交互图包括顺序图和协作图两种形式

顺序图养生描述对象按照时间顺序的消息交换, 而协作图着重描述系统成分如何协同工作

顺序图和协作图从不同的角度表达了系统中的交互和行为, 它们之间可以相互转化

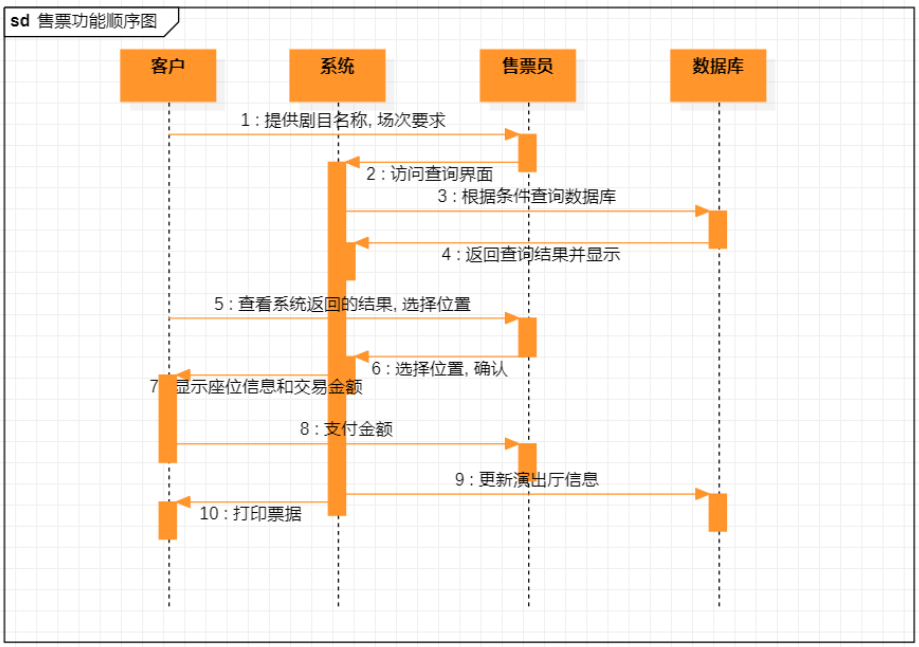
交互图和类图可以相互补充, 类图对类的描述比较充分, 但对对象之间的消息交互情况的表达不够详细, 而交互图虽不考虑系统中的所有类及对象, 但可以表示系统中某几个对象之间的交互

6.2 顺序图

顺序图也称时序图. 顺序图是显示对象之间的交互的图, 这些对象是按时间顺序排列的.

顺序图组成:

1. 对象:表示为一个矩形, 对象名称标有下划线
2. 消息: 用有标记的箭头表示
3. 生命线: 用虚线表示
4. 控制焦点: 用极窄矩形表示



顺序图在纵向是时间轴, 横向轴代表在协作中各独立对象的类元参与者. 当对象存在时, 生命线用一条纵向虚线表示, 当对象的过程处于激活状态时, 生命线是双道线

消息用从一个对象的生命线到鼍一个对象的生命线的箭头表示

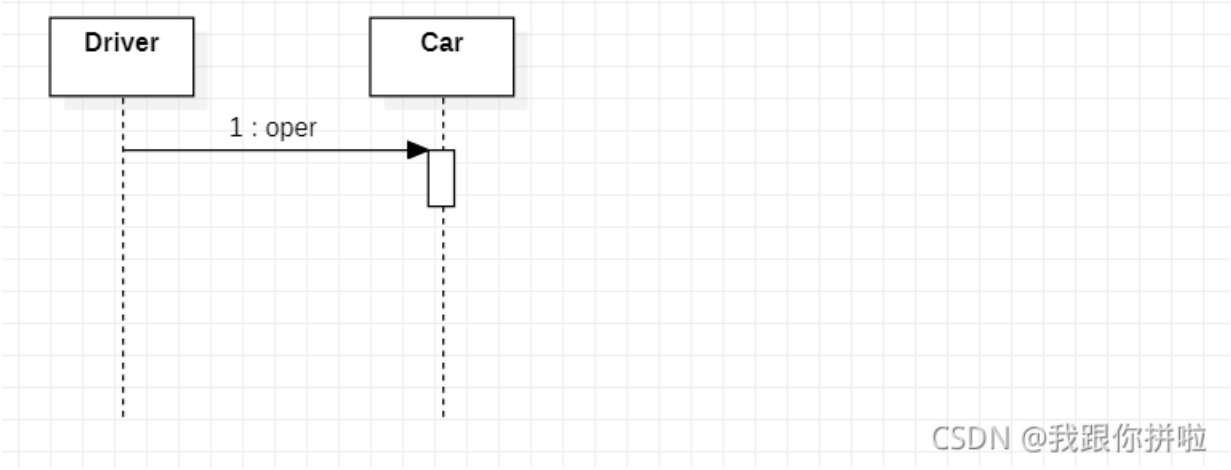
一般习惯把表示的对象放在图的两侧, 如表示人的参与者放在最左边, 表示系统的参与者放在最右边

顺序图中对象命名的3种方式

控制焦点是顺序图中表示时间段的符号.6.3 顺序图中消息

调用消息

调用消息的发送者把控制传递给消息的接收者, 然后停止活动, 等待消息接收者放弃或返回控制. 调用消息可以用来表示同步的含义



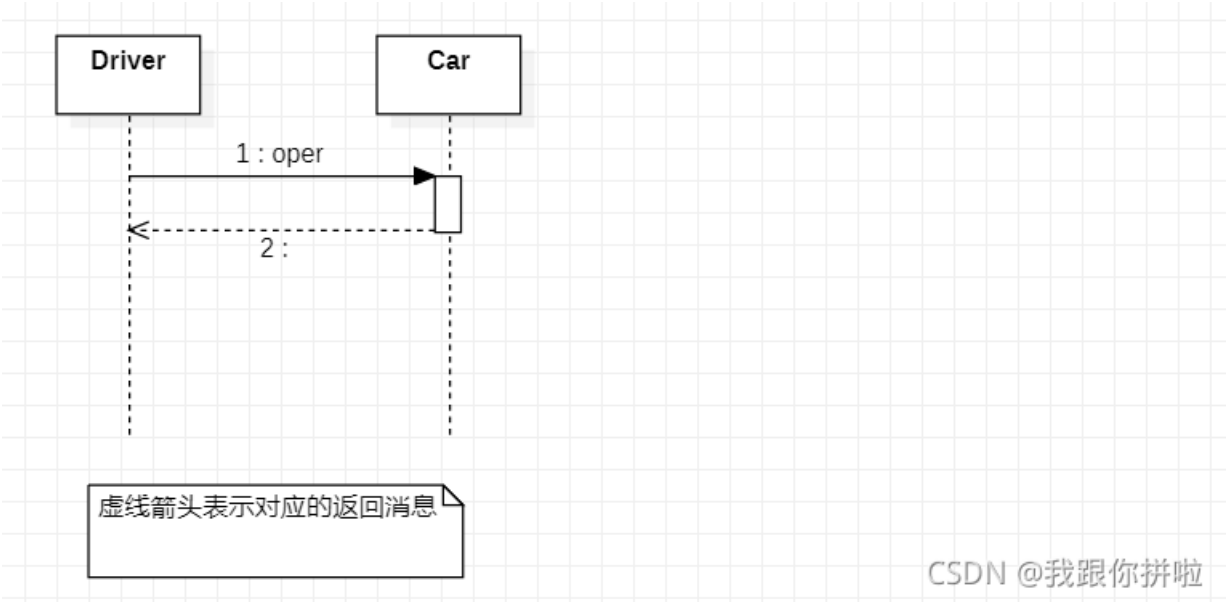
异步消息

异步消息的发送者通过消息把信号传递给消息的接收者, 然后继续自己的, 不等待接收者返回消息或控制

返回消息

返回消息表示从过程调用返回

如果是从过程调用返回, 则返回消息是隐含的, 故返回消息可以不用画出来, 对于非过程调用, 如果有返回消息, 必须明确表示出来



阻止消息和超时消息(不常用)

阻止消息是指消息发送者发出消息给消息接收者, 如果接收者无法立即接收消息, 则发送者放弃这个消息

超时消息是指消息发送者发送消息给接收者并按指定时间等待. 如果接收者无法在指定时间内接收消息, 则发送者放弃这个消息

6.4 协作图

在这里插入图片描述

协作图是用于描述系统行为是如何由各个部分协作实现的图, 协作图包括的建模元素有对象(包括参与者实例, 多对象, 主动对象等), 消息, 链等

协作图由参与者, 对象, 连接和消息等基本元素组成

对象

表示参与协作的对象

在协作图中, 多对象指的是由多个对象组成的对象集合, 一般这些对象是属于同一个类的

当需要把消息同时发送给多个对象而不是对象的时候, 就要使用多对象这概念.

协作图的多对象用多个方框的重叠表示, 而顺序图的多对象显示出来和单对象是一样的

对象关联

对象关联连接两个对象, 表示两者间的关联, 也称为链

消息

协作图的消息定义与顺序图的消息是完全一样的

消息序号

消息序号是消息的一部分, 序号表明消息传递的先后顺序

6.5 顺序图和协作图的比较

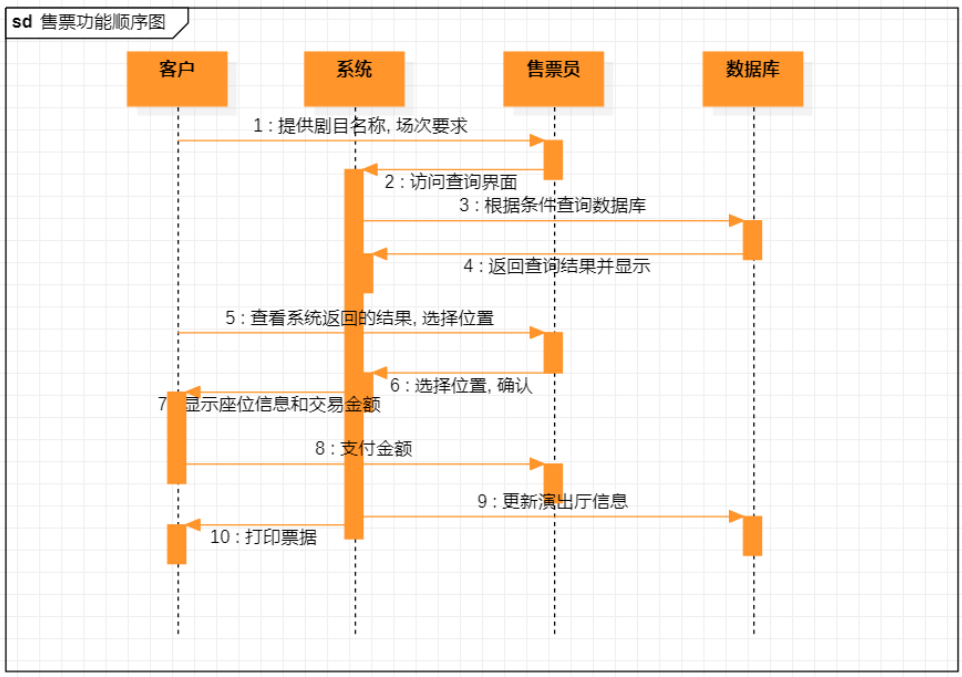
1. 顺序图强调是消息的时间顺序, 而协作图强调的是参与交互的对象的组织
2. 顺序图在表示表示算法, 对象的生命其, 具有多线程特征的对象等方面更具有优势, 协作图在表示并发控制方面更有优势
3. 顺序图中不能表示对象与对象之间的连接(即"链"), 多对象和主动对象也不能直接显示出来, 协作图则可以表示
4. 协作图不能表示生命线的分叉(fork), 而顺序图可以表示出来

6.6 问题分析

有兴趣的读者可以参考书籍page127, 这里不再赘述

6.7 建模实例

顺序图(用例: 用户买票)



第七章 数据建模

7.1 数据建模概述

1. E-R图指以实体, 关系, 属性3个基本概念分手数据的基本结构, 从而描述静态数据结构的概念模式.
2. E-R的问题是只能着眼于数据, 而不能对行为建模, 如触发器(trigger), 存储过程(stored procedure)等建模
3. UML描述能力更强, UML的类图可以看作是E-R图的扩充.
4. 可以用的类图描述数据为模式(database schema), 以及数据库表, 用类的操作来描述触发器和存储过程

7.2 数据库设计的基本过程

数据库设计主要涉及三阶段,即概念设计, 逻辑设计和物理设计

概念设计阶段是把用户的信息要求统一到一个整体逻辑结构中, 此结构能表达用户的要求, 且独立于任何数据库管理系统(DBMS)软件和硬件

逻辑结构设计阶段的任务是把概念设计阶段得到的结果转换为与先用的DBMS所支持的数据模型相符的逻辑结构

物理设计阶段的任何是对给定的数据结构模型选取一个最符合应用要求的物理结构. 数据库物理结构包括数据库的存储记录格式, 存储记录安排, 存取方法等.

数据库中的概念 版型 所应用的UML元素

数据库建模中常用的版型

数据库中的概念	版型	所应用的元素
数据库	<<database>>	构件
模式	<<Schema>>	包
表	<<Table>>	类
视图	<<View>>	类
域	<<Domain>>	类
索引	<<Index>>	操作
主键	<<PK>>	操作
外键	<<FK>>	操作
唯一性约束	<<Unique>>	操作
检查约束	<<Check>>	操作
触发器	<<Trigger>>	操作
存储过程	<<SP>>	操作
表与表之间非确定性关系	<<Non-Identifying>>	关联, 聚集
表与表之间确定性关系	<<Identifying>>	组合

7.3 数据库设计步骤

有兴趣的读者可以查看书籍Page141, 这里不再赘述

7.4 对象模型和数据模型间的转换

有兴趣的读者可以查看书籍Page141, 这里不再赘述

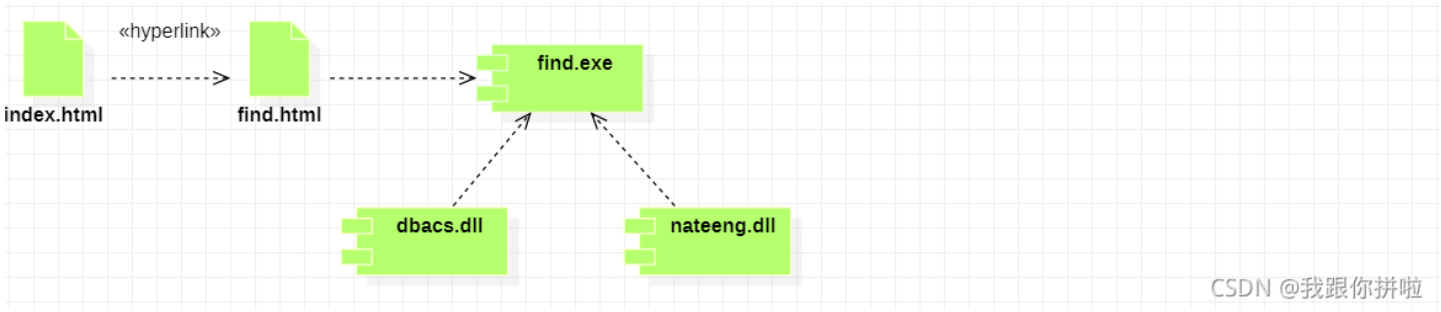
第八章 构件图和双向工程

8.1 什么是构件图

构件是系统遵从一一组接口且提供其实现的物理的, 可替换的部分.

构件图则显示一级构件及它们之间的相互关系, 包括编译, 链接或执行时构件之间的依赖关系

如图是一个构件图的例子, 表示.html文件 .exe文件和.dll文件这些构件之间的相互依赖关系



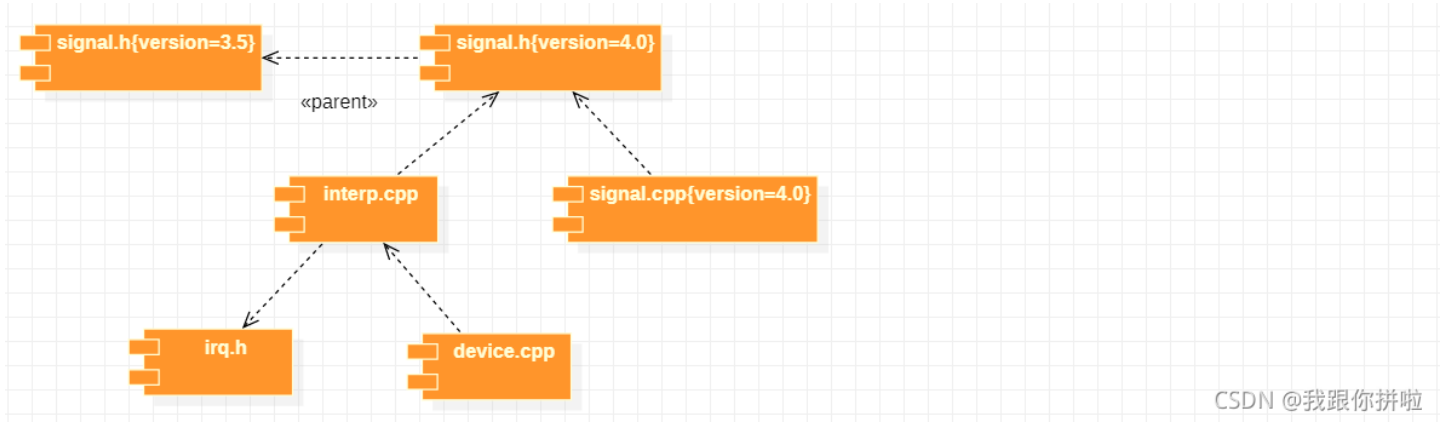
构件就是一个实际文件, 它可以有以下几种类型:

1. 构件部署(deployment component), 如.dll文件, .exe文件等
2. 工作产品构件(work product component), 如源代码文件, 数据文件, 这些构件可以用来产生部署构件
3. 执行构件(execution component), 也就是系统执行后得到的构件

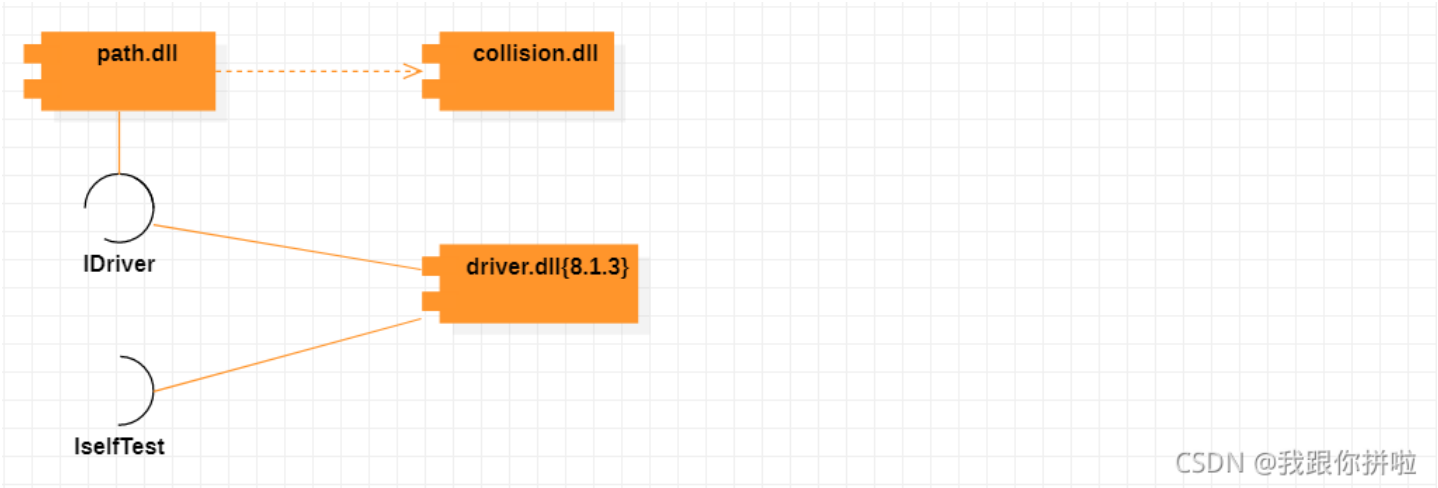
8.2 构件图的作用

构件图可以对以下几个方面建模:

1. 对源代码文件之间的相互关系建模



2. 对可执行文件的相关关系建模



8.3 构件图的工具支持

正向工程

正向工程就是根据模型来产生源代码, 当然得到源代码后再调用相应的编译器即可得到可执行代码

通过代码来得到模型

具体详情实现这里不再赘述, 有兴趣的读者可以查阅书籍P158即可.

第九章 部署图

9.1 什么是部署图

部署图也称配置图或实施图, 是对OO系统物理方面建模的两个图之一(另一个图是构件图)

部署图可以用来显示系统中计算节点的拓扑结构和通信路径与节点上运行的软构件等

一个系统模型可得一个部署图, 部署图常常用于理解分布式系统

9.2 部署图中基本概念

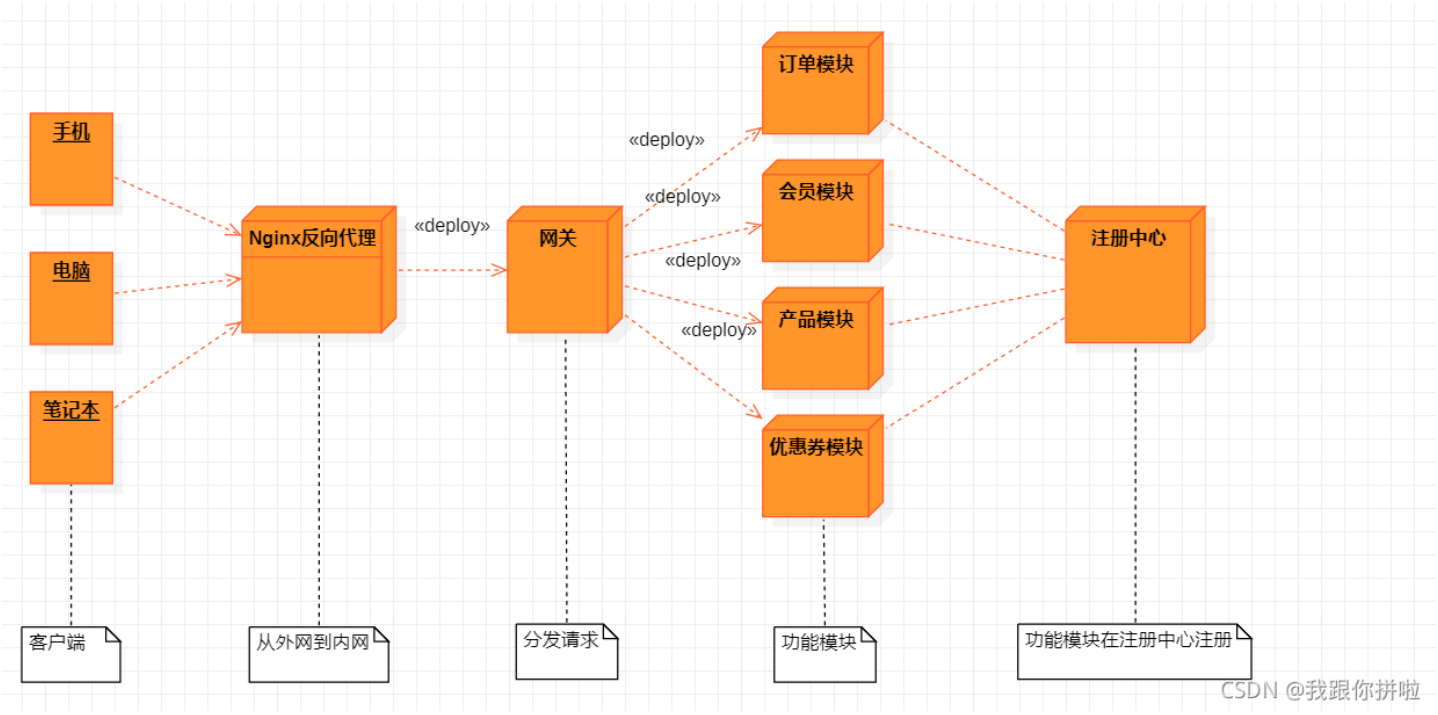
部署图有两个巧夺天工概念: 节点和连接

节点

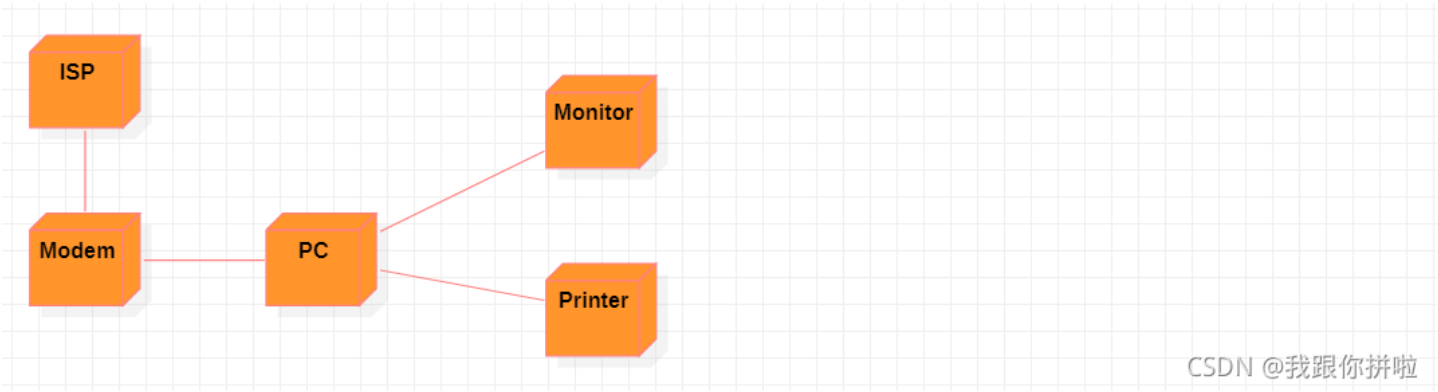
1. 节点是存在于运行时的, 代表计算资源的物理元素, 节点一般才具有一些内存, 而且学常具有处理能力
2. 节点可以代表一个物理设备及运行该设备的软件系统, 如UNIX主机, 传感器等.
3. 节点之间的连线表示系统之间进行交互的通信路径, 这个通信路径称为连接(connection)
4. 部署图中的节点分为两种类型, 即处理机(Processor)和设备(Modem)
5. 处理机是可以执行程序的硬件构件, 在部署图中, 可以说明处理机中有哪些进程, 进程的优先级与进程调度方式等
6. 设备是无计算能力的硬件结构, 如调制调解器, 终端等.

9.3 部署图的例子

微服务部署图(购物网站)



pc, 外设, 及ISP的连接部署图

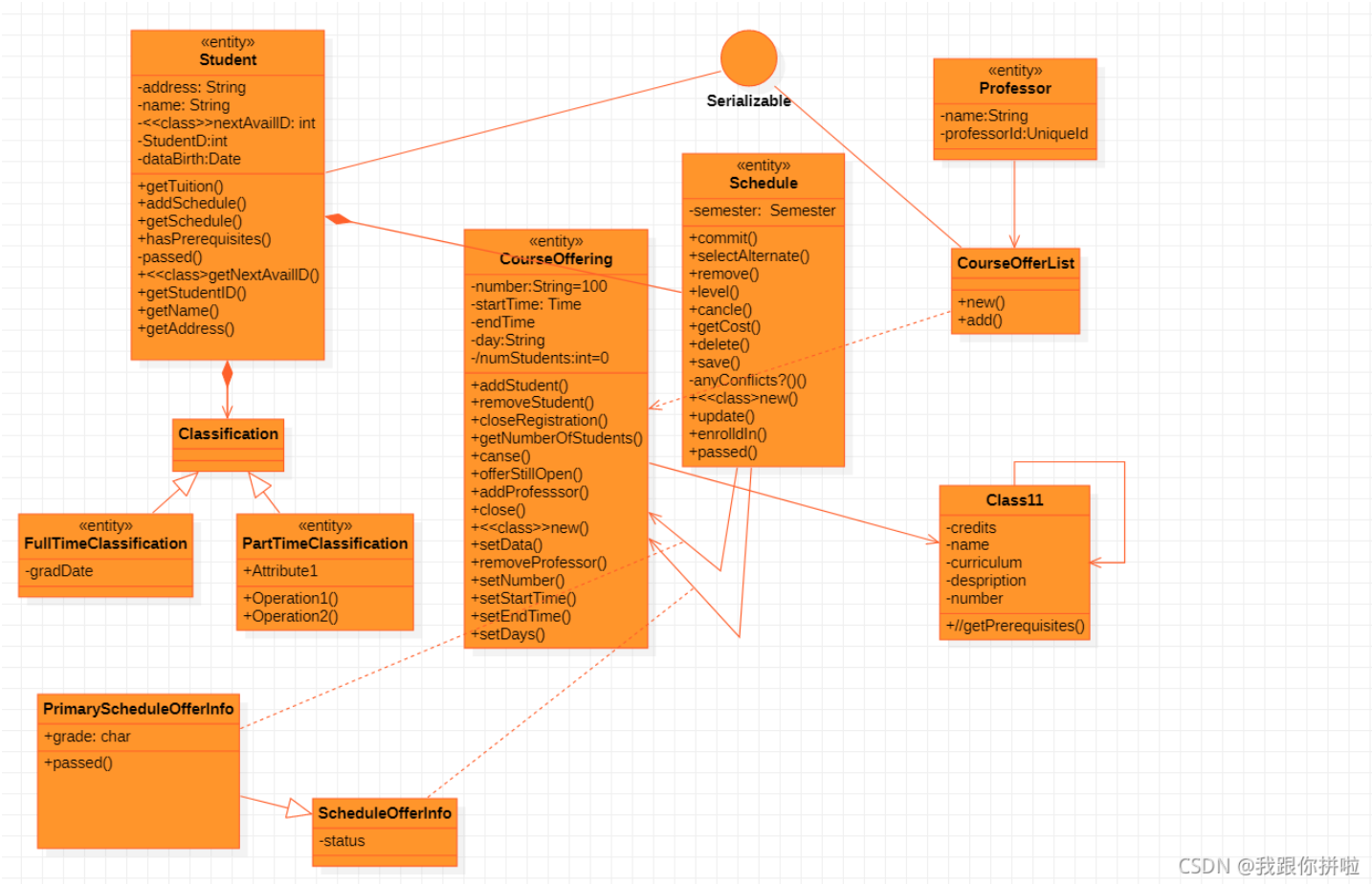


9.4 建模实例

第十章 建模实例分析

10.1 引言

有兴趣的读者可以查看书籍Page174, 此处仅提供一张类图



最后, 祝各位同学金榜题名, 事业有成！

这都不一键三连?