

UML系统建模

1 概述

1.1 课程概述

- 汇集uml及其相关的一些话题
- 回顾uml相关的符号与概念
- 以电商订单相关业务为例，借助uml完成系统建模
- 将uml变成提升建模效率，表达架构思想的工具

1.2 什么是uml

Unified Modeling Language 统一建模语言，又称标准建模语言。是用来对软件密集系统进行可视化建模的一种语言。语言，也就是一个表达思想的符号约定。

1.3 uml的发展与版本

- 建模语言出现在二十世纪70年代，80年代末开始迅速发展，建模语言达到了50多种，百家争鸣
- 后来，Rumbaugh 于1994年加入Booch所在的Rational公司，他们一起研究一种统一的方法
- 一年后，Unified Method 0.8诞生
- 经过他们三年的共同努力，UML0.9和UML0.91于1996年相继面世。
- 此后UML创始人Booch等人，邀请计算机界的知名人士与企业IBM，HP，Microsoft，Oracle等对UML进行评论，听取意见。
- 1997年1月，Rational公司向OMG（对象管理组织）提交了UML1.0
- 1997年11月，OMG宣布接受UML，认定为标准的建模语言
- 1998年发布了UML 1.2
- 1999年发布了UML 1.3
- 2003年3月发布了UML 1.5
- 2004年推出UML2.0

1.4 uml可以做什么

从命名上分析：统一、建模、语言

统一：没有规矩不成方圆，它指定了一种标准，一种约束，使得大家的表达变得一致。它被OMG（Object Management Group）所认可。

OMG是一个国际化的、开放成员的、非盈利性的计算机行业标准协会，该协会成立于1989年，他是软件行业中一个标准的认可。包括客户、领域专家、分析师、设计师、程序员、测试工程师及培训人员等。uml成为他们工作中统一的沟通工具，用于充分理解和表达自己所关注的内容。

建模：复杂业务系统建模，即建立软件系统模型。uml的创始人之一Booch，曾用建一座摩天大楼来比喻uml的必要性。简单系统下，可有可无，系统复杂或大到一定程度，建模和文档成为系统周期里非常重要的一环。

语言：面向对象思想的表达。互相之间的沟通工具。一种按照特定规则和模式组成的符号系统。

1.5 学习uml的意义

- 团队或架构设计互相交流，必然需要一种沟通语言
- 是一门技能，不一定用到，但是作为架构师应该知道
- 有其他的表达办法，但是用习惯后，uml真的很方便易用

1.6 关于uml的争议

观点一：uml是鸡肋，离程序员真正需要的设计工具还差得很远。只有为数不多的程序员使用这个工具交流想法，而没有用在具体工作中。

观点二：uml设计相当的严谨与全面，在面向对象的系统架构上，可以便捷的表达你想要表达的一切想法，优美切无可替代。

个人观点：一项技能和工具，学会并不难，需要的时候能拿来用就好，艺不压身。

1.7 切忌形式化

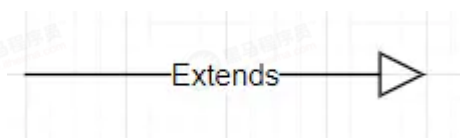
- 不要把uml过度神化
- 一个表达想法的工具而已
- 当用则用，不要刻意去套

2 理论篇

2.1 关系

关系是现实世界中事物与事物之间相互关系的符号表达，抽象到面向对象理念上，大致分为6种。

2.1.1 泛化（Generalization）



定义：

- java里的extends，可用于接口与接口之间，或父子类之间
- 单向，箭头指向父类，如Tiger指向Animal

代码：

```
//类
public class Animal {
}
public class Cat extends Animal {
}

//接口
public interface Action {
}
public interface Jump extends Action {
}
```

2.1.2 实现（Realization）



定义：

- java里的implements，箭头指向接口
- 单向，如Tiger扩展了Sleep接口，那么箭头指向Sleep

代码：

```
public interface Jump {
}
public class Tiger implements Jump {
}
```

2.1.3 依赖（Dependency）



定义：

- 某个类或对象实例，依赖于另一个而存在，在其关键方法中用到了对方
- 如果一方属性发生变动，另一方可能会受到影响
- 一般为单向，例如动物依赖于食物，eat(Food food)
- 类比在表结构上，更像是外键

代码：方法参数，局部变量

2.1.4 关联（Association）

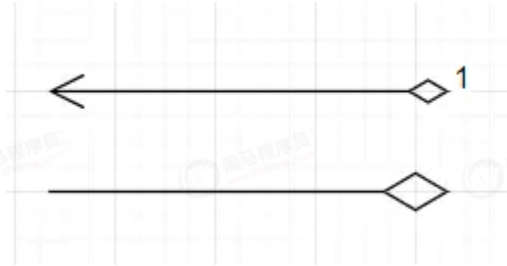


定义：

- 是一种拥有的关系，双方不一定属于同一类事物，箭头指向被拥有者
- 可以单向，也可以双向，例如Tiger与Zookeeper
- 类比在表结构上，更像是存在中间表关系

代码：成员变量

2.1.5 聚合（Aggregation）



定义：

- 单向，空心菱形起始的箭头，箭头指向被拥有者
- 一种很弱的拥有关系，A可以拥有B，但是不是离了B就无法生存
- 群体与个体的关系，如小组包含组员

代码：成员变量，多为集合

2.1.6 组合（Composition）



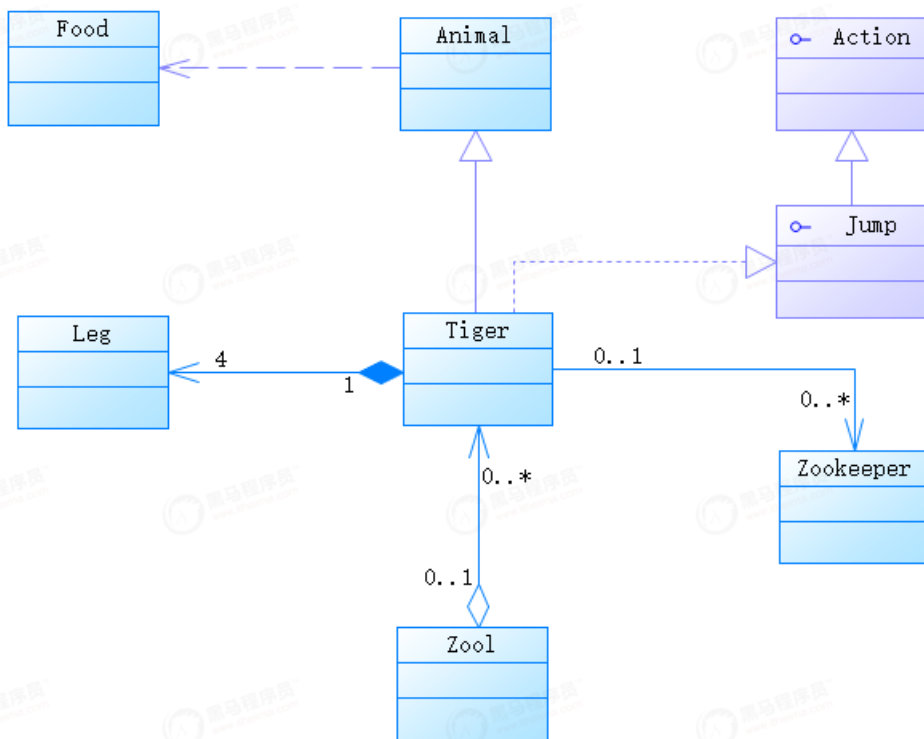
定义：

- 单向，实心菱形为起始，箭头指向子模块
- 一种整体与部分的关系，A是由B组成的，离开B则不完整。
- 单向，如人和四肢的关系

代码：成员变量，多为集合

2.1.7 实例

一张图涵盖所有的关系：

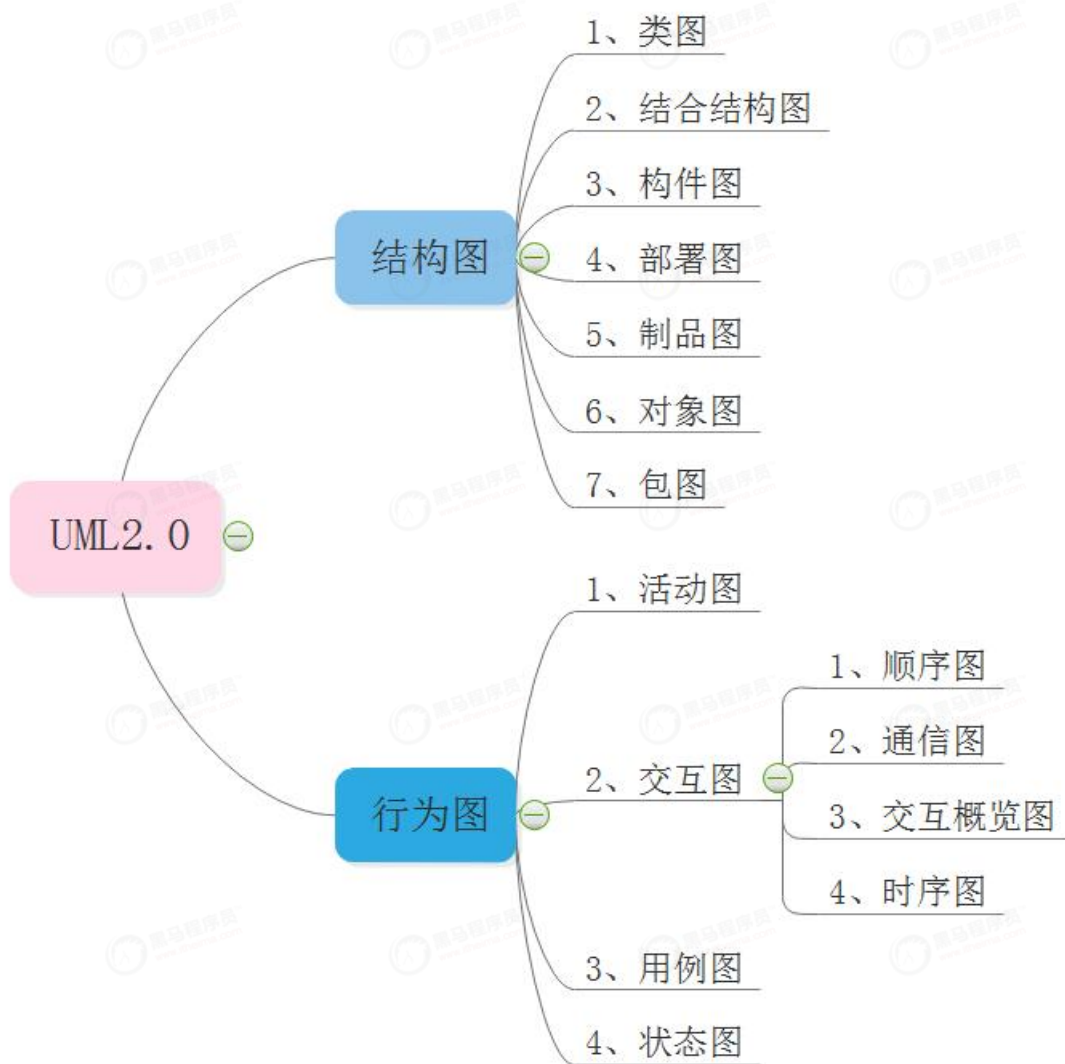


2.1.8 总结

- 继承和实现几乎不会搞混，一个上下父子关系，一个是类与接口
- 组合与聚合要注意，聚合为聚集，群体与个体。组合为组成，整体与部分
- 关联和依赖要注意，关联一般为同级别有相关性，这种相关性是长期存在的。依赖为需求关系，一方需要依赖另一方，可能会因另一方的改变而改变。
- 关系的强弱顺序：继承=实现>组合>聚合>关联>依赖

2.2 图

2.2.1 概述



uml中的图形非常多，按类型分为结构图和行为图，其中，最常用，最典型的为9种，下面分开来介绍。

- 用例图：从用户角度描述系统功能。
- 类图：描述系统中类的静态结构。
- 对象图：系统中的多个对象在某一时刻的状态。
- 状态图：是描述状态到状态控制流，用于表达系统状态的变化
- 活动图：描述了业务实现用例的工作流程，强调的是动作之间的衔接
- 序列图：对象之间的动态合作关系，强调对象发送消息的顺序
- 协作图：描述对象之间的协助关系，强调对象之间的合作关系
- 组件图：描述系统各个组件及其相关关系的静态视图
- 部署图：定义系统中软硬件的物理体系结构

2.2.2 类图

1) 说明

- 面向对象系统建模中最常用和最重要的图，是定义其它图的基础
- 主要是用来显示系统中的类、接口以及它们之间的静态结构和关系的一种静态模型
- 描述细化相关的属性和操作，是一个对业务模型面向对象化的过程，也是对系统的约束
- 可以直接构建可执行代码，但真正使用的场景相对较少

2) 可用元素



- 类:

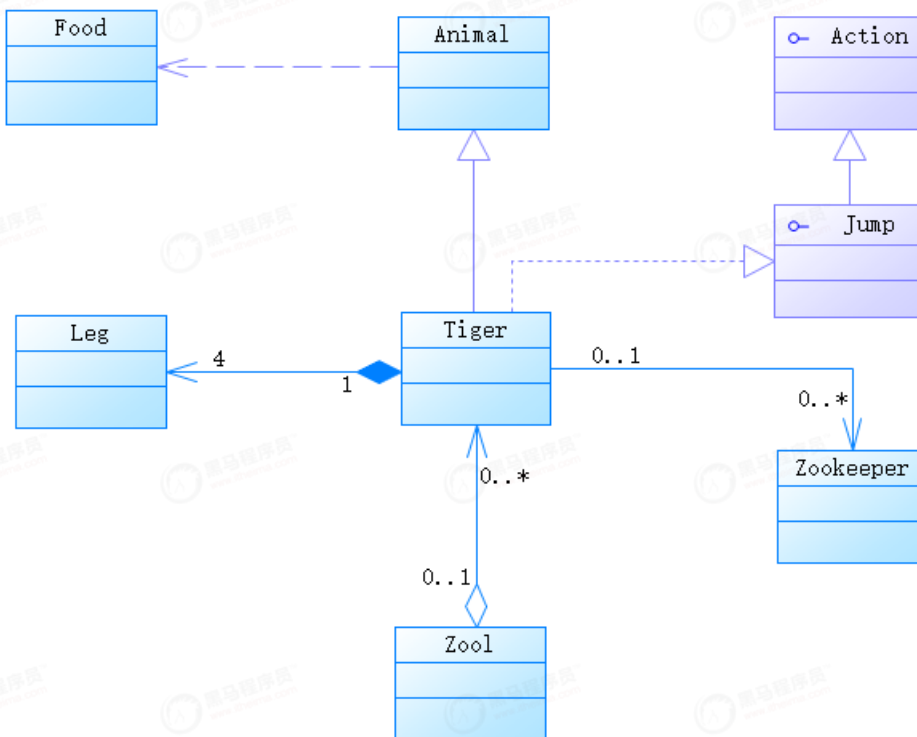
Cat	
- iAmPrivate	: int
# iAmProtected	: int
* iAmPackage	: int
+ iAmPublic	: int
+ eat ()	: int
+ sleep ()	: int

- 接口:

Actions	
+ eat	: int
+ sleep	: int
+ eat ()	: int
+ sleep ()	: int

- 关系: 可以使用上述中的6大关系。

3) 实例



2.2.3 对象图

1) 说明

- 对象图和类图一样反映系统的静态过程，但它表达的是一个实际场景。

- 对象图显示某时刻对象和对象之间的关系。可看成一个类图的快照。
- 对象图是类图的实例，所以几乎使用与类图完全相同的标识。

2) 可用元素



- 对象：

<u>LittleCat:Cat</u>	
iAmPrivate	= 1
iAmProtected	= 2
iAmPackage	= 3
iAmPublic	= 4

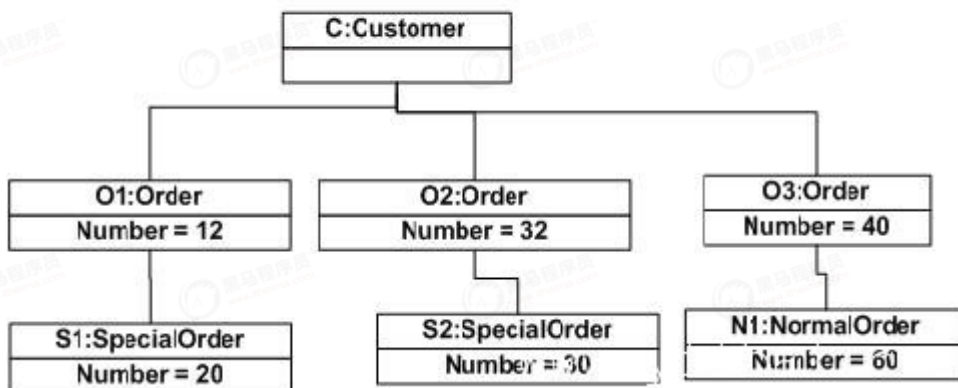
3) 关系

对象图因为是运行在某个时间节点的对象镜像，所以关系比较单一，描述的是类与类的实例之间。不涉及接口

- 关联：对象之间存在关联关系，如用户和订单
- 依赖：对象实例之间的依赖关系，如商品对象依赖店铺

4) 图例

Object diagram of an order management system

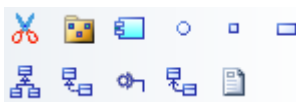


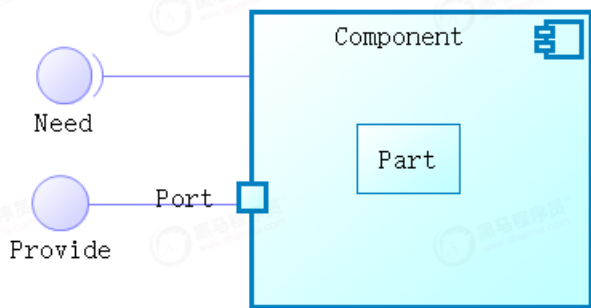
2.2.4 组件图（拓展）

1) 说明

- UML1.1中，组件图是用来描述一个系统的物理构件。包括文件，可执行文件，库等
- UML2 中，关注组件间的关联（使用什么接口，通过什么端口通讯），强调通过接口来描述组件行为
- 对于后端来说，组件图比较适用于 SOA 架构、微服务架构，描述整个系统的结构以及子系统间的通讯方式
- 对于前端来说，组件图适合在使用类似 react、vue 这样组件化的前端技术框架时，表达对组件的设计，比如一个页面会有个骨架组件，骨架组件包含了导航组件，列表组件等等

2) 可用元素



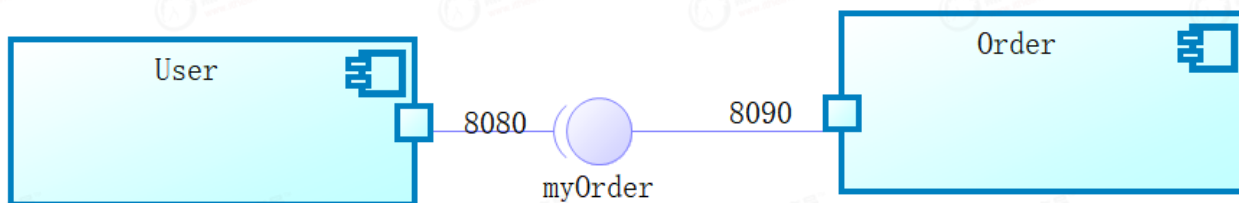


- 组件：描述的是系统的其中一个组成部分，一个完整的可独立服务的模块或单元，比如订单服务，k8s里的一个pod
- 部件：组件内可能细化为多个子模块
- 端口：组件对外提供服务就必须暴露对应的端口。如http rest服务默认的80
- 接口：部件/组件之间的一种约定，分提供者 and 需求者同时展示了某个部件提供出的功能

3) 关系

- 泛化：用于接口与接口之间存在的父子关系，组件之间也可能存在，但相对用的较少
- 实现：接口和其实现者（提供服务的组件）之间
- 关联：Require link / Connector，接口与调用者（需要接口的组件）之间

4) 图例



2.2.5 部署图（拓展）

1) 说明

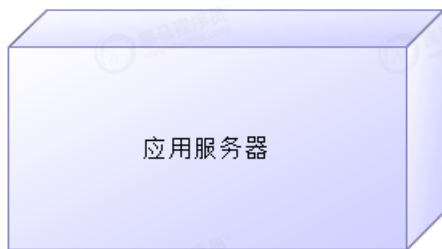
- 一种展示运行时进行处理的节点和在节点上存在的组件配置的图。
- 阐述了在实际应用中软件和它的运行环境的关系，并且描述了软件部署在硬件上的具体方式。

2) 可用元素

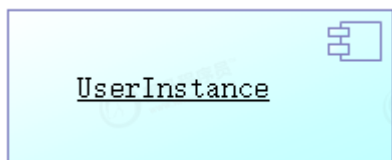


- 节点：

原先单实例MVC架构下，节点可以认为是某台物理服务器，微服务及容器化下，物理服务器大多纳入编排管理，docker实例由系统在物理节点见自由调度，组件无法锁定在某个固定物理节点上，这种环境下的node可以理解为一个容器，或k8s中的pod。



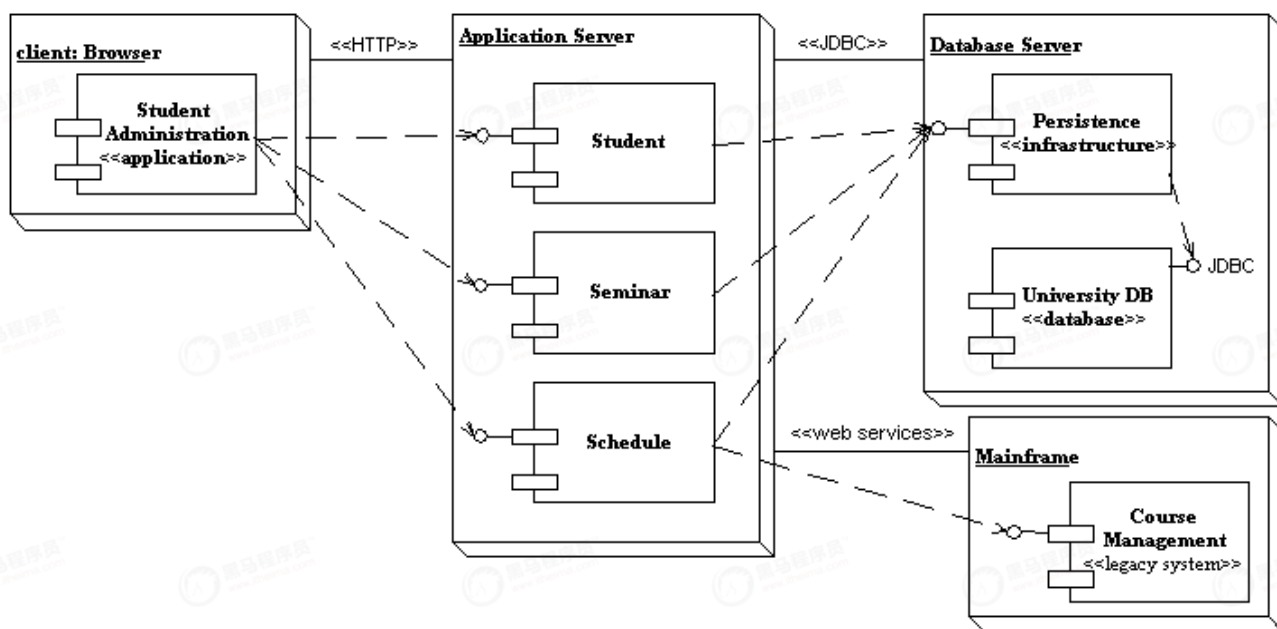
- 组件实例
相当于组件里的实例化，类比于类和对象



3) 关系

- 依赖：发生于组件之间，如用户组件依赖于订单组件
- 关联：node association，发生于节点之间，例如应用服务器需要关联mysql数据库

4) 图例



2.2.6 用例图

1) 说明

- 用例图是用来描述系统功能的技术，表示一个系统中用例与参与者及其关系的图
- 主要用于需求分析阶段，和产品文档比较贴近
- 用例图相当于从用户的视角来描述和建模整个系统，分析系统的功能与行为。

2) 可用元素



- 参与者：使用系统的人，有多少种角色就有多少参与者。



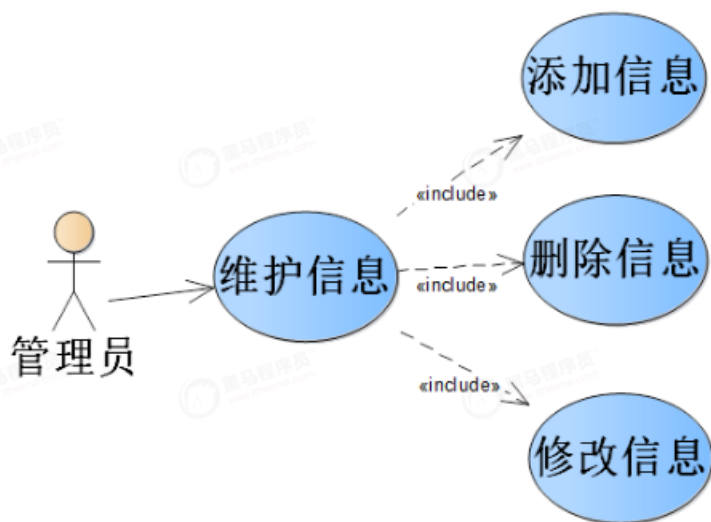
- 用例：参与者可用做的事情



3) 关系

- 泛化：参与者之间可用泛化，例如用户与普通会员；用例也可用泛化，如用户管理与修改密码
- 关联：发生于参与者和用例之间，表示该角色可用有哪些用例（行为）
- 依赖：发生于用例之间，例如登录依赖于注册

4) 图例



2.2.7 交互图

交互图分为序列图和协作图，两者既可以相互转换而不丢失信息，又存在一定差异。下面分开讲再类比

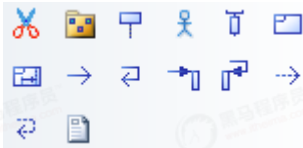
序列图

1) 说明

- 序列图主要用于按照交互发生的一系列顺序，显示对象之间的消息或行为传递。

- 序列图可以形象表达整个流程，和流程图有相似之处，但是流程图偏业务逻辑，序列图则是系统面向对象化建模后，对应到对象上的交互过程。趋向于开发者角度。

2) 可用元素

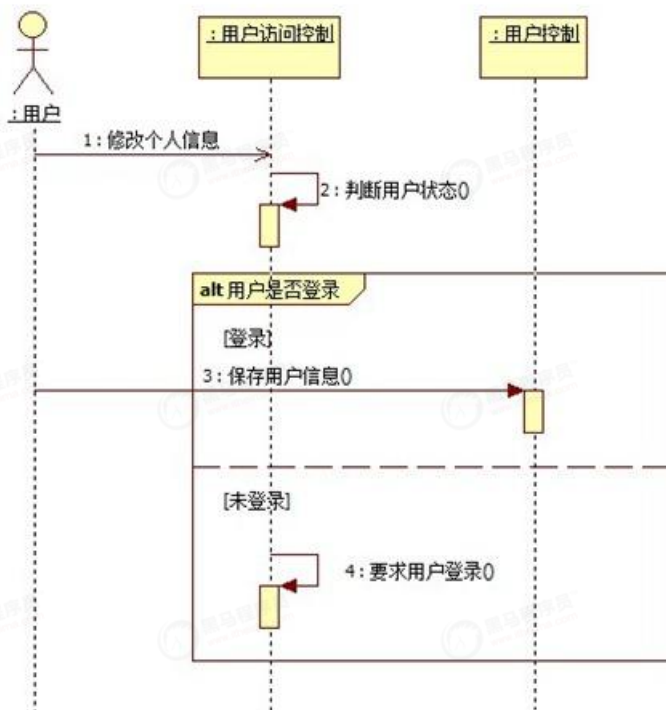


- 对象：提供功能和交互的类的实例
- 参与者：同用例种的参与人，多为一段流程的发起点
- 时间线：对象在整个交互流程中的生命周期
- 消息：对象间需要发送和返回的消息，可以自己发给自己
- 外部参考：序列图可以引入外部的一段作为参考，或参与序列中与当前图的元素交互
- 片段：将某一段序列纳入片段管理，该片段像原子一样，发生某些整体的行为，例如循环

3) 关系

- 不会用到6大关系，相互之间使用message交互。代表的是信息流动。

4) 图例



协作图（1.4）/通信图（2.0）

1) 说明

- 协作图与时序图类似，二者都是用对象间的交互来描述用例的。
- 两者关注角度稍有不同，时序图强调交互的时间次序，协作图强调交互的空间结构。

2) 可用元素



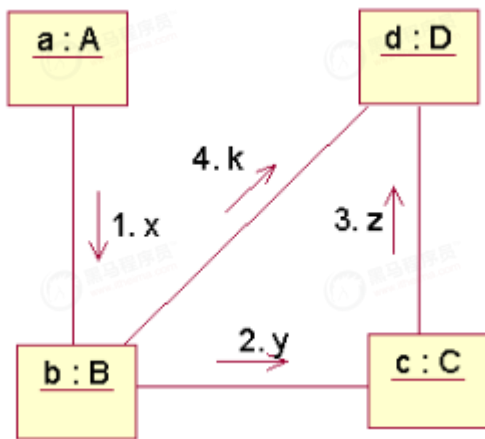
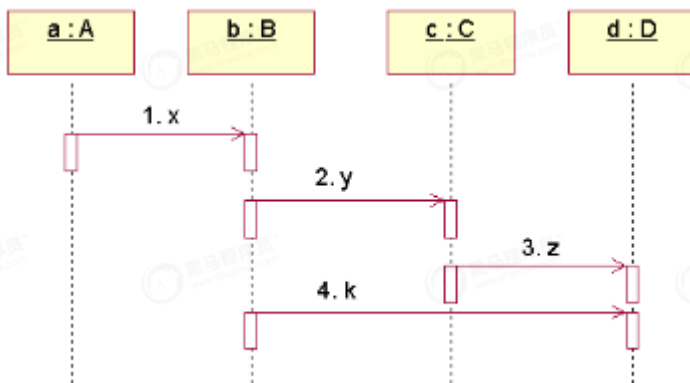
- 参与者：系统参与的角色
- 对象：同时序图，系统中实例化的对象
- 关联：对象间的关联关系
- 消息：依附于关联而存在，承载了对象间要传递的信息

3) 关系

- 不会用到6大关系，相互之间使用message交互。代表的是信息流动。

4) 图例

两种交互图可以相互转化，类比如下：



2.2.8 状态机（拓展）

状态机分为状态图和活动图，

状态图

1) 说明

- 描述一个实体基于事件反应的动态行为，它有两方面的价值，一是反映对象可能有哪些状态，二是这些状态之间是如何流转的，需要什么样的条件下进入什么样的状态。

2) 可用元素

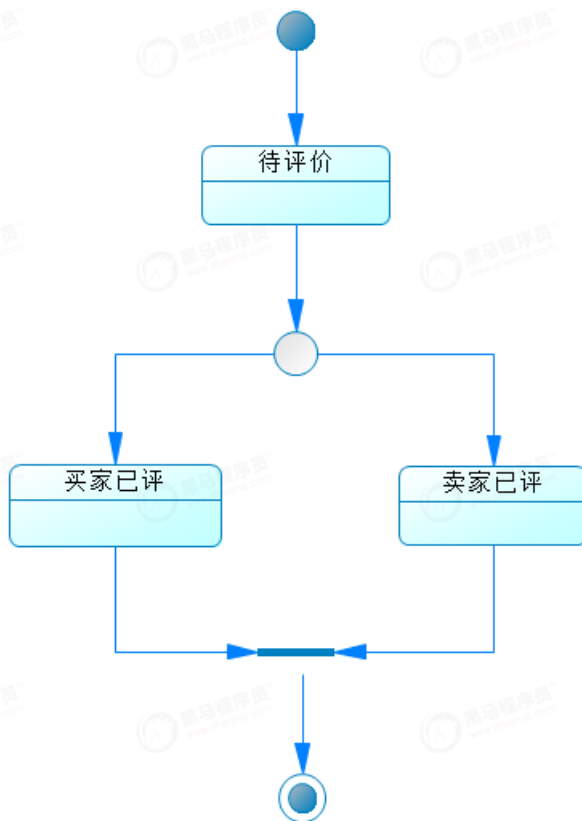


- 状态：某一个时间点，对象所在的状态
- 转移：连接状态之间，因为状态时可以互相变化转移的
- 分支/会合点：状态变化中可能产生分叉或交会，如确认收货后，双方互评产生分叉
- 开始/结束：状态的起始与终结
- 同步点：需要多个分支状态都具备时使用。多用于并行协作处理的状态流转，如互评都完成后，订单才算终结

3) 关系

- 只有转移关系，表示状态之间的变化

4) 图例



活动图

1) 说明

- 活动图用于企业的业务流程建模，是对内部活动与活动之间流转动作的表达
- 活动图类比流程图：活动图存在分支与交会，可以表达并行存在的活动，流程图多为是与否分支判断
- 活动图类比状态图：关注不同，状态图强调行为的结果（下一个状态是什么），活动图在乎行为的动作（下一步干什么）。两者可以理解为穿插配合，一动一静，活动可能会触发下一步不同的状态。

2) 可用元素



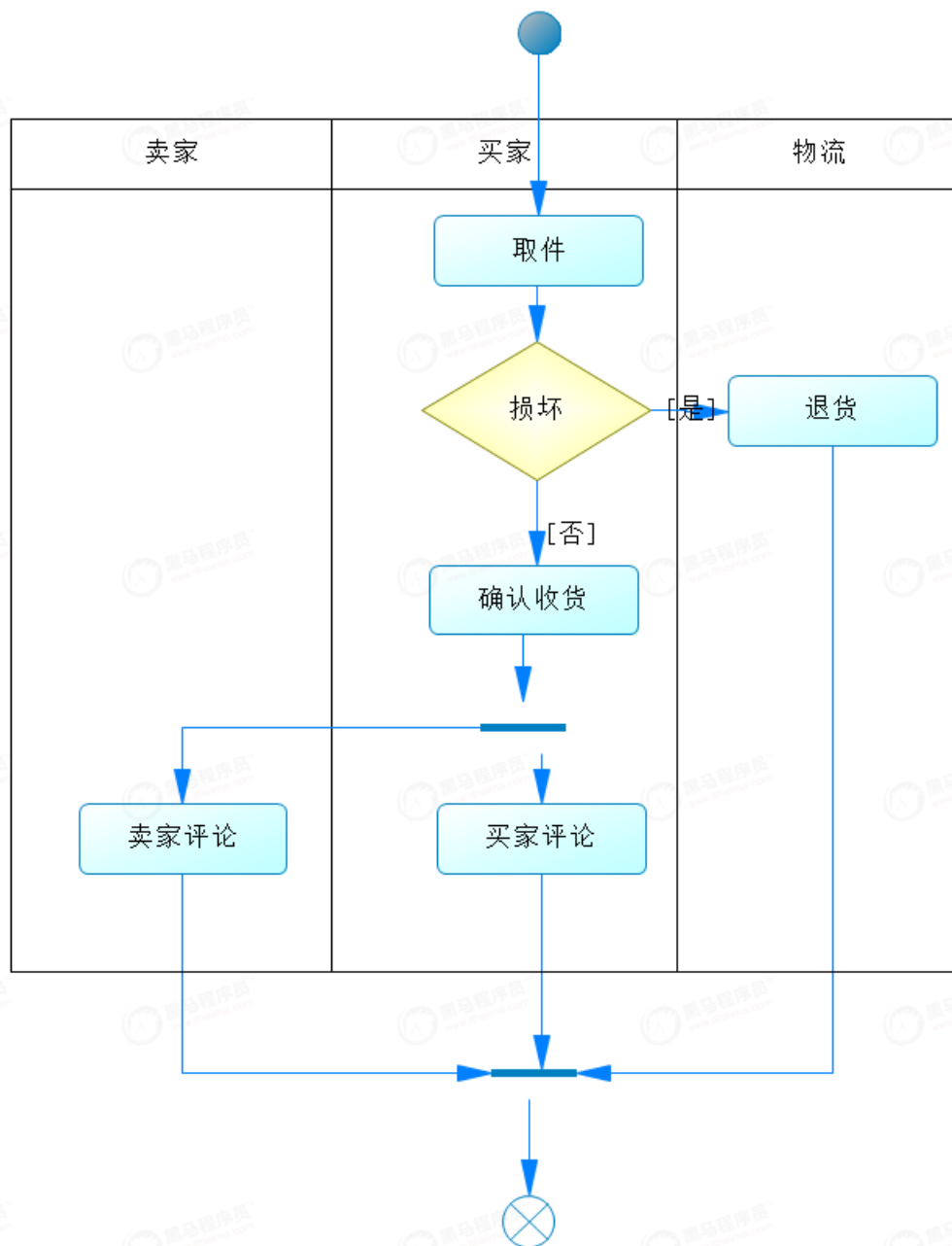
- 活动：表达系统中，或对象内的某一个可以发生的动作
- 对象：活动的发生者，或交互者
- 流转：活动的跳转，即下一步指向谁

- 判定：类似与流程图里的判决，根据条件产生不同的流转
- 同步：并行流转下的汇集，不同于流程图的地方
- 起始/结束：活动的发起与终结
- 泳道：对UML活动图中的活动进行分组，同一类活动在一个泳道上，清晰明了

3) 关系

- 只有流转，也就是活动的跳转，表示下一个活动是啥

4) 图例



3 实战篇

3.1 常用工具

3.1.1 Rational Rose

老牌，大名鼎鼎，史上最有名的UML产品，以至于大多数情况下，很多人将他等同于UML工具，需要注意的是，自从 Rational被IBM收购之后，Rational Rose已经成为历史，作为UML1.4标准的产物，现在已经不升级，但是够用。其替代品是IBM的其他产品，如IBM RSA。

3.1.2 RSA

IBM® Rational® Software Architect，IBM的旗舰产品，是一个高级而又全面的应用程序设计、建模和开发工具，用于实现端到端的软件交付。通过和IBM其他产品的协调，支持软件开发的全生命周期开发。但是也有它的缺点，笨重，繁杂（大公司产品的通病？？？）

3.1.3 Enterprise Architect

Sparx Systems 公司的旗舰产品。它覆盖了系统开发的整个周期，除了开发类模型之外，还包括事务进程分析，使用案例需求，动态模型，组件和布局，系统管理，非功能需求，用户界面设计，测试和维护等。总之你需要的基本都可以满足，价格还便宜。性价比之选。

3.1.4 StarUML

开放源码的UML开发工具，是由韩国公司主导开发出来的产品。用Delphi写的，是个大神。需要付费，网站提供购买注册码，小巧简单而易用，与rose相比更是明显。

3.1.5 VISIO

VISIO可以理解作为一种通用的画图工具，它具备常见的各种图形，从VISIO2000版本才开始涉足软件分析设计到代码生成的全部功能，单纯从画图角度，有着无可比拟的优势，UML图标仅仅是其中很少的一部分。优点是跟微软的office产品的能够很好兼容。可以把图形直接复制或者内嵌到WORD的文档中。但是到代码的生成更多是支持微软自家的产品如VB，C#，ms sql等（微软的一贯作风），如果仅是画uml图和大量的word文档表达，它可以满足你。

3.1.6 PowerDesigner

Sybase的企业建模和设计解决方案。采用模型驱动方法，将业务与IT结合起来，可帮助部署有效的企业体系架构，并为研发生命周期管理提供强大的分析与设计技术。将多种标准数据建模技术集成一体，并与IDE集成，典型的如Eclipse 插件形式。PD更多给人的印象是数据库建模技术，但是它可以完成uml的所有建模操作并映射到数据库和代码层面。并提供60多种关系数据库的连接支持。

3.1.7 总结

- 以上工具各有利弊，根据自己实际情况和爱好选择即可
- 基本都涵盖软件建模的完整周期，uml只是他们的一部分功能
- 任何一种都可以满足你学习和工作中uml建模的使用需求

3.2 订单建模实战

3.2.1 B2C交易用例图

用例图从订单系统使用人角色出发，反馈订单体系里面有哪些人，可以做哪些事情

1) 业务需求：

- 买家：浏览商品，下单、支付、签收
- 卖家：开店，确认订单，发货，商品维护
- 双方：退货，换货，评价，收藏

3.2.2 订单模块类图

订单类图从业务模型出发，用面向对象思想，订单业务中的模型抽象为一个个对象

1) 元素：

- 人物类：Seller, Buyer, User
- 商品类：Shop, Product
- 交易环节类：Cart, Order, Invoice, AliPay, WeichatPay, ICBCPay...
- 交易环节接口：Pay
- 促销相关类：DiscountPromotion, ReductionPromotion...
- 促销接口：Promotion

2) 关系：

- 关联：Order→Seller, Buyer, Pay; Shop→Seller
- 依赖：Order→Cart→Promotion, Invoice;
- 组合：Shop→Product
- 聚合：Cart→Product
- 泛化：Buyer, Seller→User
- 实现：DiscountPromotion, ReductionPromotion→Promotion; AliPay, WeichatPay, ICBCPay→Pay

3.2.3 订单下单时的对象图

对象图表达的是下订单的时刻，系统存在的对象及对象之间的关联关系。对象具备了实际的属性值

1) 元素：

- 与类图一致，但是接口将不复存在，而变为实际实现类
- Cart生命周期终结，Invoice还没诞生，Product, Promotion依附在了订单上
- 对象上的属性具备了实际值，不再是泛化的类属性的概念

2) 关系：

- 对象之间变为实例关联（Instance link），泛化和实现不再被使用。
- 弱类型可以使用依赖，比如Order与打折的Promotion

3.2.4 B2C下单支付序列图

序列图反应下单到支付完成这段时间里，各个对象怎么协作和交互，互相之间需要传递什么消息。

1) 元素

- 人物：Buyer
- 对象：Product, Cart, Order, Promotion, Pay, AliPay（外部）

2) 时间序列

- 顺向：Buyer→筛选→Product→添加→Cart→促销结算→Promotion→下单→Order→支付→Pay→跳转→AliPay
- 回路：Buyer←通知←Order←开单←Pay←回调←AliPay

- 环路: Cart $\longleftrightarrow$ 增删商品

3.2.5 下单支付协作图

协作图同序列图类似,可以由序列图转化而来。但是协作图反映的是交互关系,像是铺开的时序图

1) 元素,同时序图

- 人物: Buyer
- 对象: Product, Cart, Order, Promotion, Pay, AliPay (外部)

2) 交互,同序列图

- 关联
- 消息

3.2.6 B2B先款后货状态图 (拓展)

以B2B先款后货业务模式下,订单对象的流转状态为例,实现业务状态展示

1) 元素

- 起始
- 订单的各个状态值
- 交会
- 同步
- 终结

2) 流转

- 开始 $\rightarrow$ 合同已生成 $\rightarrow$ 已付款 $\rightarrow$ 交收单已生成 $\rightarrow$ 已发货 $\rightarrow$ 验货验票 $\rightarrow$ 待评价 $\rightarrow$ 分支 $\rightarrow$ 买方已评, 卖方已评 $\rightarrow$ 同步 $\rightarrow$ 终结

3.2.7 B2B先款后货活动图 (拓展)

在先款后货的交易中,双方都要做哪些活动或者说操作,通过活动图建模体现

1) 元素

- 泳道: Buyer, Seller, Manager
- 活动 (Buyer): 生成合同, 支付货款, 验货验票, 买方评价
- 活动 (Seller): 生成交收单, 发货, 卖方评价
- 活动 (Manager): 仲裁
- 判定: 是否有异议

2) 流程

- 开始 $\rightarrow$ 生成合同 $\rightarrow$ 支付货款 $\rightarrow$ 生成交收单 $\rightarrow$ 发货 $\rightarrow$ 验票验货 $\rightarrow$ 判断 (是否有异议?) $\rightarrow$ 否 $\rightarrow$ 分支 $\rightarrow$ 买方, 卖方评价 $\rightarrow$ 同步 $\rightarrow$ 终结
- 判断异议 $\rightarrow$ 是 $\rightarrow$ 仲裁 $\rightarrow$ 终结

3.2.8 订单服务组件图 (拓展)

界定构成订单服务的各个对象以及他们之间的可用接口,相当于定义了一组约定。

1) 元素

- Order: create (Cart) , open (Pay)
- Cart: addProduct (Buyer) , removeProduct (Buyer) , settle (Buyer)
- Promotion: discount (Cart)
- Pay: pay (Buyer)

3.2.9 订单中心部署图（拓展）

将订单中心如何部署到服务器？部署图的职责就是完成这部分的内容。在docker容器化编排和云环境下，部署图变得不那么的确切。node可以类比理解为docker容器或者是k8s等编排工具中的pod，而组件也不再固定在某个node中，而是由调度器动态调度，分散部署。

1) 元素

- node: App1, App2, App3, 假设有三台；mysql数据库，假设1主2从
- component: Order, Cart, Pay, Promotion

2) 关系

- 组件离散分布于3台App
- App关联mysql
- mysql主从为依赖

3.3 总结

- 一切皆工具，适合你的就是最好的
- 灵活变通，不要拘泥规矩，规矩是死的人是活的
- 表达思想才是目的，一切都是为了能说清楚你的想法，这也是语言的本质
- 不要为了画图而画图，uml不是必须的，但是有了它你的架构工作会变得更顺畅

希望uml能成为你架构工作的利器，提升效率，解决问题。Thanks!