# 概述

## UML简介

UML (Unified Modeling Language)为面向对象软件设计提供统一的、标准的、可视化的建模语言。适用于描述以用例为驱动，以体系结构为中心的软件设计的全过程。

UML的定义包括UML语义和UML表示法两个部分。

UML语义：UML对语义的描述使开发者能在语义上取得一致认识，消除了因人而异的表达方法所造成的影响。

UML表示法：UML表示法定义UML符号的表示法，为开发者或开发工具使用这图形符号和文本语法为系统建模提供了标准。

## UML的构成

事物(Things)：UML模型中最基本的构成元素，是具有代表性的成分的抽象

关系(Relationships)：关系把事物紧密联系在一起

图(Diagrams )：图是事物和关系的可视化表示

### UML事物

UML包含4种事物：构件事物、行为事物、分组事物、注释事物。

1. **构件事物： UML模型的静态部分，描述概念或物理元素**

它包括以下几种：

* 类：具有相同属性相同操作 相同关系相同语义的对象的描述
* 接口：描述元素的外部可见行为，即服务集合的定义说明
* 协作：描述了一组事物间的相互作用的集合
* 用例：代表一个系统或系统的一部分行为，是一组动作序列的集合
* 构件：系统中物理存在，可替换的部件
* 节点：运行时存在的物理元素

另外，参与者、信号应用、文档库、页表等都是上述基本事物的变体

1. **行为事物：UML模型图的动态部分，描述跨越空间和时间的行为**

* 交互：实现某功能的一组构件事物之间的消息的集合，涉及消息、动作序列、链接
* 状态机：描述事物或交互在生命周期内响应事件所经历的状态序列

1. **分组事物： UML模型图的组织部分，描述事物的组织结构**

* 包： 把元素组织成组的机制

1. **注释事物： UML模型的解释部分，用来对模型中的元素进行说明，解释**

* 注解 ：对元素进行约束或解释的简单符号

### UML关系

1. **依赖**

依赖(dependency)是两个事物之间的语义关系，其中一个事物(独立事物)发生变化，会影响到另一个事物(依赖事物)的语义

1. **关联**

关联(association)是一种结构关系，它指明一个事物的对象与另一个事物的对象间的联系。

1. **泛化**

泛化(generalization)是一种特殊/一般的关系。也可以看作是常说的继承关系

1. **实现**

实现(realization)是类元之间的语义关系，其中的一个类元指定了由另一个类元保证执行的契约

### UML图

1. **结构型（Structure Diagram）**

* 类图（Class Diagram）
* 对象图（Object Diagram）
* 构件图（Component Diagram）
* 部署图（Deployment Diagram）
* 包图（Package Diagram）

1. **行为型（Behavior Diagram）**
   * 活动图（Activity Diagram）
   * 状态机图（State Machine Diagram）
   * 顺序图（Sequence Diagram）
   * 通信图（Communication Diagram）
   * 用例图（Use Case Diagram）
   * 时序图（Timing Diagram）

## UML绘图工具

* IBM Rational Rose
* Microsoft Office Visio
* Sybase PowerDesigner
* Enterprise Architect
* Visual UML
* Star UML
* Argo UML
* UMLet
* JUDE
* BOUml

## UML参考网站

* www.umlchina.com
* www.uml.net.cn

# 类图（Class Diagram）

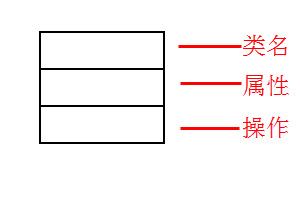
## 概念

类图以反映类的结构(属性、操作)以及类之间的关系为主要目的，描述了软件系统的结构，是一种静态建模方法

类图中的“类”与面向对象语言中的“类”的概念是对应的，是对现实世界中的事物的抽象

## 类图的组成

类图由三部分组成：类名、属性和操作（方法），其中类名是必须有的。



类如果有属性，则每一个属性都必须有一个名字，另外还可以有其它的描述信息，如可见性、数据类型、缺省值等

类如果有操作，则每一个操作也都有一个名字，其它可选的信息包括可见性、参数的名字、参数类型、参数缺省值和操作的返回值的类型等



在UML类图中，类一般由三部分组成：

(1) 第一部分是类名：每个类都必须有一个名字，类名是一个字符串。

(2) 第二部分是类的属性(Attributes)：属性是指类的性质，即类的成员变量。一个类可以有任意多个属性，也可以没有属性

UML规定属性的表示方式为：

可见性 名称:类型 [ = 缺省值 ]

其中：

“可见性”表示该属性对于类外的元素而言是否可见，包括公有(public)、私有(private)和受保护(protected)三种，在类图中分别用符号+、-和#表示。

“名称”表示属性名，用一个字符串表示。

“类型”表示属性的数据类型，可以是基本数据类型，也可以是用户自定义类型。

“缺省值”是一个可选项，即属性的初始值。

(3) 第三部分是类的操作(Operations)：操作是类的任意一个实例对象都可以使用的行为，是类的成员方法。

UML规定操作的表示方式为：

可见性 名称(参数列表) [ : 返回类型]

其中：

“可见性”的定义与属性的可见性定义相同。

“名称”即方法名，用一个字符串表示。

“参数列表”表示方法的参数，其语法与属性的定义相似，参数个数是任意的，多个参数之间用逗号“，”隔开。

“返回类型”是一个可选项，表示方法的返回值类型，依赖于具体的编程语言，可以是基本数据类型，也可以是用户自定义类型，还可以是空类型(void)，如果是构造方法，则无返回类型。

## 类图之间的关系表示

### 关联关系

关联(Association)关系是类与类之间最常用的一种关系，它是一种结构化关系，用于表示一类对象与另一类对象之间有联系

1. **双向关联**

默认情况下，关联是双向的。例如：顾客(Customer)购买商品(Product)并拥有商品，反之，卖出的商品总有某个顾客与之相关联。因此，Customer类和Product类之间具有双向关联关系，如图所示：



代码：

|  |
| --- |
| public class Customer {  private Product[] products;  ……  }  public class Product {  private Customer customer;  ……  } |

1. **单向关联**

类的关联关系也可以是单向的，单向关联用带箭头的实线表示。例如：顾客(Customer)拥有地址(Address)，则Customer类与Address类具有单向关联关系，如图3所示：



代码：

|  |
| --- |
| public class Customer {  private Address address;  ……  }  public class Address {  ……  } |

1. **自关联**

在系统中可能会存在一些类的属性对象类型为该类本身，这种特殊的关联关系称为自关联。例如：一个节点类(Node)的成员又是节点Node类型的对象。



|  |
| --- |
| public class Node {  private Node subNode;  ……  } |

1. **多重性关联**

多重性关联关系又称为重数性(Multiplicity)关联关系，表示两个关联对象在数量上的对应关系。在UML中，对象之间的多重性可以直接在关联直线上用一个数字或一个数字范围表示。

|  |  |
| --- | --- |
| 表示方式 | 多重性说明 |
| 1..1 | 表示另一个类的一个对象只与该类的一个对象有关系 |
| 0..\* | 表示另一个类的一个对象与该类的零个或多个对象有关系 |
| 1..\* | 表示另一个类的一个对象与该类的一个或多个对象有关系 |
| 0..1 | 表示另一个类的一个对象没有或只与该类的一个对象有关系 |
| m..n | 表示另一个类的一个对象与该类最少m，最多n个对象有关系 (m≤n) |

1. **聚合关系**

聚合(Aggregation)关系表示整体与部分的关系。在聚合关系中，成员对象是整体对象的一部分，但是成员对象可以脱离整体对象独立存在。在UML中，聚合关系用带空心菱形的直线表示。例如：汽车发动机(Engine)是汽车(Car)的组成部分，但是汽车发动机可以独立存在，因此，汽车和发动机是聚合关系，如图所示：



1. 组合关系

组合(Composition)关系也表示类之间整体和部分的关系，但是在组合关系中整体对象可以控制成员对象的生命周期，一旦整体对象不存在，成员对象也将不存在，成员对象与整体对象之间具有同生共死的关系。在UML中，组合关系用带实心菱形的直线表示。例如：人的头(Head)与嘴巴(Mouth)，嘴巴是头的组成部分之一，而且如果头没了，嘴巴也就没了，因此头和嘴巴是组合关系，如图所示：



### 依赖关系

依赖(Dependency)关系是一种使用关系，特定事物的改变有可能会影响到使用该事物的其他事物，在需要表示一个事物使用另一个事物时使用依赖关系。大多数情况下，依赖关系体现在某个类的方法使用另一个类的对象作为参数。在UML中，依赖关系用带箭头的虚线表示，由依赖的一方指向被依赖的一方。

例如：

驾驶员开车，在Driver类的drive()方法中将Car类型的对象car作为一个参数传递，以便在drive()方法中能够调用car的move()方法，且驾驶员的drive()方法依赖车的move()方法，因此类Driver依赖类Car，

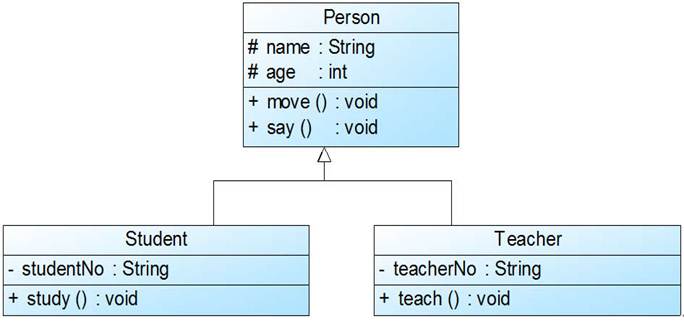


在系统实施阶段，依赖关系通常通过三种方式来实现，第一种也是最常用的一种方式是如图1所示的将一个类的对象作为另一个类中方法的参数，第二种方式是在一个类的方法中将另一个类的对象作为其局部变量，第三种方式是在一个类的方法中调用另一个类的静态方法。图1对应的Java代码片段如下：

|  |
| --- |
| public class Driver {  public void drive(Car car) {  car.move();  }  ……  }  public class Car {  public void move() {  ......  }  ……  } |

### 泛化关系

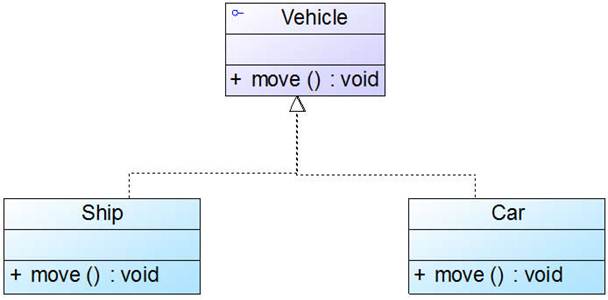
泛化(Generalization)关系也就是继承关系，用于描述父类与子类之间的关系，父类又称作基类或超类，子类又称作派生类。在UML中，泛化关系用带空心三角形的直线来表示。在代码实现时，我们使用面向对象的继承机制来实现泛化关系，如在Java语言中使用extends关键字、在C++/C#中使用冒号“：”来实现。例如：Student类和Teacher类都是Person类的子类，Student类和Teacher类继承了Person类的属性和方法，Person类的属性包含姓名(name)和年龄(age)，每一个Student和Teacher也都具有这两个属性，另外Student类增加了属性学号(studentNo)，Teacher类增加了属性教师编号(teacherNo)，Person类的方法包括行走move()和说话say()，Student类和Teacher类继承了这两个方法，而且Student类还新增方法study()，Teacher类还新增方法teach()。如图所示：



|  |
| --- |
| //父类  public class Person {  protected String name;  protected int age;  public void move() {  ……  }  public void say() {  ……  }  }  //子类  public class Student extends Person {  private String studentNo;  public void study() {  ……  }  }  //子类  public class Teacher extends Person {  private String teacherNo;  public void teach() {  ……  }  } |

### 实现关系

接口之间也可以有与类之间关系类似的继承关系和依赖关系，但是接口和类之间还存在一种实现(Realization)关系，在这种关系中，类实现了接口，类中的操作实现了接口中所声明的操作。在UML中，类与接口之间的实现关系用带空心三角形的虚线来表示。例如：定义了一个交通工具接口Vehicle，包含一个抽象操作move()，在类Ship和类Car中都实现了该move()操作，不过具体的实现细节将会不一样，如图所示：



|  |
| --- |
| public interface Vehicle {  public void move();  }  public class Ship implements Vehicle {  public void move() {  ……  }  }  public class Car implements Vehicle {  public void move() {  ……  }  } |

### 综合显示关系



# 对象图（Object Diagram）

## 概念

对象：类（Class）的实例（Instance）就是对象（Object）。

类图代表了一类事物，那么对象图就代表着某个具体的事物。

对象图应用比较少，一般使用类图表示。

## 组成

类图和对象图的表示方式区别



## 实例



“A:公司”表示对象A是公司这类的实例。

“:公司”则表示这是公司这类的实例，但是没有具体的名称。

# 构件图（Component Diagram）

## 概念

1. **什么是构件？**

•构件是系统中可替换的物理部分，它包装了实现而且遵从并提供一组接口的实现

•规格说明：对于构件，必须有一个它所提供服务的抽象描述。通俗地说，每个构件都必须提供特定的服务

•一个或多个实现：构件是一种物理概念，必须被一个或多个实现所支持，当然这些实现都必需符合规格说明

•受约束的构件标准：每一个构件，在实现时必须遵从某种构件标准

•封装方法：也就是构件遵从的封装标准

•部署方法：当件要运行时，首先要部署它

1. **构件的特点**

* 能实行一定的功能，或者提供一些服务。
* 不能单独运行，要作为系统的一部分来发挥作用。
* 是物理上的概念，不是逻辑上的概念。
* 可单独维护、可独立升级、可替换而不影响整个系统。

1. **构件的类型**

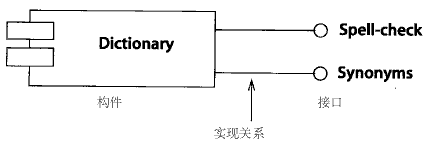
* **实施构件：**这类构件是构成一个可执行系统必要和充分的构件，例如动态链接库（dll）、可执行文件（exe），另外还包括如COM+、CORBA及企业级Java Beans、动态Web页面也属于实施构件的一部分
* **工作产品构件：**这类构件主要是开发过程的产物，包括创建实施构件的源代码文件及数据文件。这些构件并不是直接地参与可执行系统，而是用来产生可执行系统的中间工作产品
* **执行构件：**作为一个正在执行的系统的结果而被创建的，例如由DLL实例化形成的COM+对象

## 构件图的组成

|  |  |  |
| --- | --- | --- |
| **事物名称** | **含义** | **图例** |
| 构件 | 指系统中可替换的物理部分，构件名字(如图中的Dictionary)标在矩形中，提供了一组接口的实现。 |  |
| 接口 | 外部可访问到的服务 (如图中的Spell-check)。 |  |
| 构件实例 | 节点实例上的构件的一个实例，冒号后是该构件实例的名字(如图中的RoutingList)。 |  |

## 例子

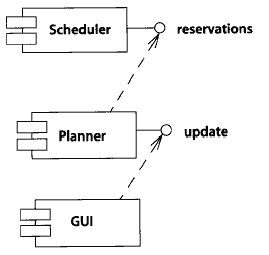
### 例子1



图中的构件名称是Dictionary字典。

该构件向外提供两个接口，即两个服务Spell-check拼写检查、Synonyms同义词。

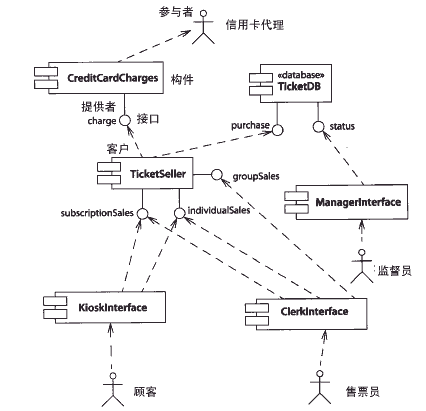
### 例子2



图中“Planner计划者”构件向外提供一个“update更新”接口服务。

同时，该构件要求外部接口提供一个“Reservations预定”服务。

### 例子3



**依赖关系：**

* 顾客需要信息亭接口提供服务
* 售票员需要职员接口提供服务
* 信用卡付款需要信用卡代理提供服务
* 职员接口需要预订销售、个人销售和团体销售提供服务
* 管理接口需要数据库状态提供服务
* 售票处需要付款和购买提供服务

**实现关系：**

* 信用卡付款提供付款服务
* 票数据库提供购买和状态查询服务
* 售票处提供预订购买、个人购买和团体购买服务

# 部署图（Deployment Diagram）

## 概念

部署图用于静态建模，是表示运行时过程节点结构、构件实例及其对象结构的图。

如果含有依赖关系的构件实例放置在不同节点上，部署视图可以展示出执行过程中的瓶颈。

部署图的两种表现形式：实例层部署图和描述层部署图(会在后面的实例中给出)。

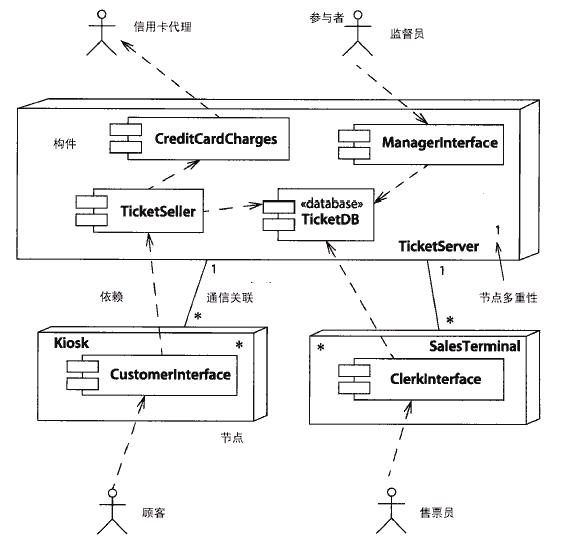
## 组成

|  |  |  |
| --- | --- | --- |
| **名称** | **解释** | **图例** |
| 节点 | 节点用一长方体表示，长方体中左上角的文字是节点的名字 (如图中的Joe’sMachine:PC) 。  节点代表一个至少有存储空间和执行能力的计算资源。  节点包括计算设备和(至少商业模型中的)人力资源或者机械处理资源，可以用描述符或实例代表。  节点定义了运行时对象和构件实例(如图中的Planner构件实例)驻留的位置。 |  |
| 构件 | 系统中可替换的物理部分。 |  |
| 接口 | 外部可访问的服务。 |  |
| 构件  实例 | 构件的一个实例。 |  |

## 关系

|  |  |  |
| --- | --- | --- |
| **关系名称** | **解释** | **图例** |
| 实现关系 | 构件向外提供服务。 | (节点内) |
| 依赖关系 | 构件依赖外部提供的服务(由构件到接口)。 |  |
| 关联关系 | 通信关联。 | (节点间) |
| 其他关系 | 对象的移动(一个位置到另一个位置)。 |  |

## 实例



节点TicketServer(售票服务)上的构件:

CreditCardCharges/ManagerInterface/ TicketSeller/TicketDB

节点Kiosk(信息亭)上的构件:

CustomerInterface

SalesTerminal(销售终端)上的构件:

ClerkInterface

通信链关系(不带箭头的直线)

TicketServe票服务器与Kiosk信息厅之间存在一对多的通信关联；与SalesTerminal售票终端也存在一对多的通信关联；

依赖关系(带箭头的虚线)

TicketSeller售票构件依赖CreditCardCharges信用卡付款构件和TicketDB票数据库构件提供的服务。

图中顾客购票的情景如下：

顾客通过位于Kiosk节点的顾客接口控件进行购票的操作，该顾客接口构件的购票操作依赖于处于TicketServer节点上的售票构件提供的服务，售票构件要完成售票操作，又要依赖统一节点上信用卡付款构件提供的付款服务和票数据库构件。

## 部署图和构件图的关系

**部署图与构件图相同的构成元素：**

构件、接口、构件实例、构件向外提供服务、构件要求外部提供的服务。

**部署图与构件图的关系：**

部署图表现构件实例；

构件图表现构件类型的定义。

部署图偏向于描述构件在节点中运行时的状态，描述了构件运行的环境；

构件图偏向于描述构件之间相互依赖支持的基本关系。

# 包图（Package Diagram）

## 概念

包可直接理解为命名空间，文件夹，是用来组织图形的封装，包图可以用来表述功能组命名空间的组织层次。

**包的作用：**

* 对语义上相关的元素进行分组；
* 定义模型中的“语义边界”；
* 提供配置管理单元；
* 在设计时，提供并行工作的单元；
* 提供封装的命名空间，其中所有名称必须惟一

## 包之间的关系

**1、《use》关系**

是一种默认的依赖关系，如果在依赖关系中没有指名类型，就默认为《use》关系**。**

《use》关系说明（客户包）发出者中的元素以某种方式使用（提供者包）箭头指向的包的公共元素，也就是说发出者包依赖于箭头指向的包

**2、《import》关系**

说明提供者包的命名空间添加到客户包的命名空间中，客户包中的元素也能访问提供者包中的所有公共元素

**3、《access》关系**

说明客户包中的元素能访问提供者包中的所有公共元素，但是命名空间不合并，在客户包中必须使用路径名。

《use》和《access》的区别：《use》表示包中元素间的依赖，而《access》依赖却不存在包中元素的依赖，只简单表达包间的一般依赖。

**4、《trace》关系**

表示一个包到另一个包的发展历史

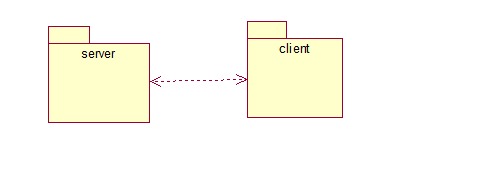
## 绘图

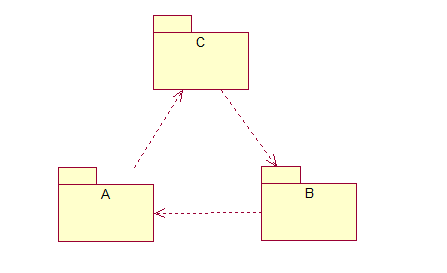
1、遵循“最小化系统间的耦合关系”原则：

最小化包之间的依赖，最小化每个包中的public、protected元素的个数，最大化每个包中private元素的个数

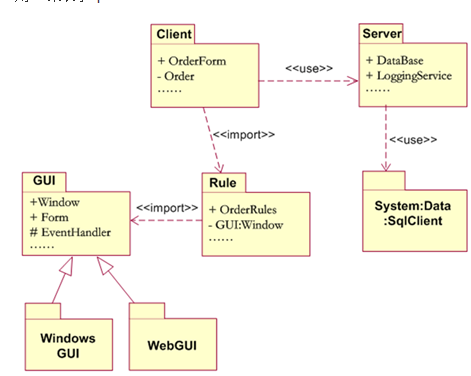
2、建模时避免包之间的循环依赖，也就是不能包含相互依赖的情况。

应避免出现的模型：





## 实例



# 活动图（Activity Diagram）

## 概念

※ 描述系统的动态行为。

※ 包含活动状态(ActionState)，活动状态是指业务用例的一个执行步骤或一个操作，不是普通对象的状态。

※ 活动图适合描述在没有外部事件触发的情况下的系统内部的逻辑执行过程；否则，状态图更容易描述。

※ 类似于传统意义上的流程图。

※ 活动图主要用于：

业务建模时，用于详述业务用例，描述一项业务的执行过程；

设计时，描述操作的流程。

## 活动图事物

开始状态(Initial State):实心的原点，表示流程的开始。  
 结束状态（Final State）：空心的原点，表示流程的结束。  
活动（Activity）：圆角矩形，流程中的一个步骤。  
判断（Decision）：菱形，从菱形开始的分支，叫做判断。

合并（Merge）：菱形，分支流程汇聚到一个菱形上。  
监护（Guard）：  
泳道（Swimlane）：垂直泳道和水平泳道

|  |  |  |
| --- | --- | --- |
| 活动 (ActionState) | 动作的执行 |  |
| 起点 (InitialState) | 活动图的开始 |  |
| 终点(FinalState) | 活动图的终点 |  |
| 对象流(ObjectFlowState) | 活动之间的交换的信息 |  |
| 发送信号(signalSending) | 活动过程中发送事件，触发另一活动流程 |  |
| 接收信号(SignalReceipt) | 活动过程中接收事件，接收到信号的活动流程开始执行 |  |
| 泳道(SwimLane) | 活动的负责者 |  |

## 活动图关系

|  |  |  |
| --- | --- | --- |
| 迁移(transition) | 活动的完成与新活动的开始 |  |
| 分支(junction point) | 根据条件，控制执行方向 |  |
| 分叉(fork) | 以下的活动可并发执行 |  |
| 结合(join) | 以上的并发活动再此结合 |  |

## 活动图实例

### 一般活动图



### 泳道活动图



# 状态机图（State Machine Diagram）

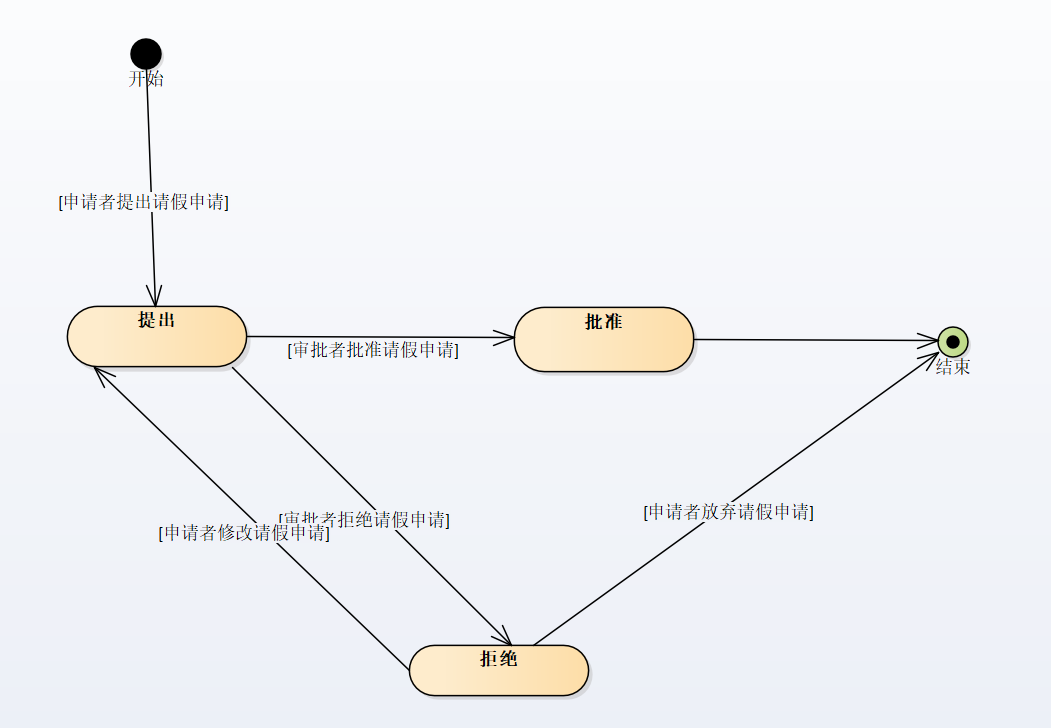
## 概念

说明对象在它的生命期中响应事件所经历的状态序列，以及它们对那些事件的响应。

## 组成

|  |  |  |
| --- | --- | --- |
| **状态** | 上格放置名称，下格说明处于该状态时，系统或对象要做的工作(见可选活动表) |  |
| **转移** | 转移上标出触发转移的事件表达式。如果转移上未标明事件，则表示在源状态的内部活动执行完毕后自动触发转移 | **消息(属性)[条件]/*动作*** |
| **开始** | 初始状态(一个) |  |
| **结束** | 终态(可以多个) |  |

## 实例



## 状态机图与活动图的区别

|  |  |  |
| --- | --- | --- |
|  | **活动图** | **状态机图** |
| 文字表达 | 采用主动宾或者动宾的表达方式，表示某某做什么事情 | 一般使用形容词或名词，表示某状态 |
| 框框的形状 | 左右两边都是弧形 | 四个角是弧线 |

# 顺序图（Sequence Diagram）

## 概念

顺序图用来表示用例中的行为顺序。当执行一个用例行为时，顺序图中的每条消息对应了一个类操作或状态机中引起转换的事件。

顺序图展示对象之间的交互，这些交互是指在场景或用例的事件流中发生的。 顺序图属于动态建模。

顺序图的重点在消息序列上，也就是说，描述消息是如何在对象间发送和接收的。表示了对象之间传送消息的时间顺序。

浏览顺序图的方法是：从上到下查看对象间交换的消息。

## 组成

|  |  |  |
| --- | --- | --- |
| **事物名称** | **解释** | **图** |
| **参与者** | 与系统、子系统或类发生交互作用的外部用户(参见用例图定义)。 |  |
| **对象** | 顺序图的横轴上是与序列有关的对象。对象的表示方法是：矩形框中写有对象或类名，且名字下面有下划线。 |  |
| **生命线** | 坐标轴纵向的虚线表示对象在序列中的执行情况(即发送和接收的消息，对象的活动)这条虚线称为对象的“生命线”。 |  |
| **消息符号** | 消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在图中从上到下排列。 |  |

## 实例

1. **一般时序图**



1. **顺序图循环、分支结构**



# 通信图（Communication Diagram）

## 概念

通信图描述的是对象和对象之间的关系，即一个类操作的实现。描述对象和对象之间的调用关系，体现的是一种组织关系。

**共同点：**时序图与协作图均显示了对象间的交互。

**不同点：**时序图强调交互的时间次序。

协作图强调交互的空间结构。

## 组成

对象：图中矩形元素即对象，其中冒号前面部分为对象名，后面部分为类名。即表示一个类的实例！当消息是给对个对象发的时，接收方用重叠的矩形表示。参考后面多态和广播的图例。

链接：链接用来在通信图中关联对象，目的是让消息在不同系统对象之间传递。可以理解为链接是公路，消息是车。用两个对象之间的单一线条表示。

消息：是通信图中对象与对象之间通信的方式。消息具有如下类型与符号表示。

|  |  |  |
| --- | --- | --- |
| **事物名称** | **解释** | **图** |
| 参与者 | 发出主动操作的对象，负责发送初始消息，启动一个操作。 |  |
| 对象 | 对象是类的实例，负责发送和接收消息，与顺序图中的符号相同，冒号前为对象名，冒号后为类名。 |  |
| 消息流  (由箭头和标签组成) | 箭头指示消息的流向，从消息的发出者指向接收者。标签对消息作说明，其中，顺序号指出消息的发生顺序，并且指明了消息的嵌套关系；冒号后面是消息的名字。 | 标签 |

## 实例



# 用例图（Use Case Diagram）

## 概念

用例图是被称为参与者的外部用户所能观察到的系统功能的模型图。

用例图列出系统中的用例和系统外的参与者，并显示哪个参与者参与了哪个用例的执行(或称为发起了哪个用例)。

用例图多用于静态建模阶段(主要是业务建模和需求建模)。

## 组成

|  |  |  |
| --- | --- | --- |
| **事物名称** | **解释** | **图** |
| 参与者  (Actor) | 表示与您的应用程序或系统进行交互的用户、组织或外部系统。用一个小人表示。 | http://pic001.cnblogs.com/images/2012/1/2012013015202157.gif |
| 用例  (Use Case) | 用例就是外部可见的系统功能，对系统提供的服务进行描述。用椭圆表示。 | http://pic001.cnblogs.com/images/2012/1/2012013015210973.gif |
| 子系统  (Subsystem) | 用来展示系统的一部分功能，这部分功能联系紧密。 | http://pic001.cnblogs.com/images/2012/1/2012013015215321.gif |

## 关系

用例图中涉及的关系有：关联、泛化、包含、扩展：

|  |  |  |  |
| --- | --- | --- | --- |
| **关系** | | **解释** | **图** |
| 参与者与用例之间的关系 | 关联 | 表示参与者与用例之间的交互，通信途径。  (关联有时候也用带箭头的实线来表示，这样的表示能够显示地表明发起用例的是参与者。) |  |
| 用例之间的关系 | 包含 | 箭头指向的用例为被包含的用例，称为包含用例；箭头出发的用例为基用例。包含用例是必选的，如果缺少包含用例，基用例就不完整；包含用例必须被执行，不需要满足某种条件；其执行并不会改变基用例的行为。 | 《include》 |
| 扩展 | 箭头指向的用例为被扩展的用例，称为扩展用例；箭头出发的用例为基用例。扩展用例是可选的，如果缺少扩展用例，不会影响到基用例的完整性；扩展用例在一定条件下才会执行，并且其执行会改变基用例的行为。 | 《extend》 |
| 参与者之间的关系 | 泛化 | 发出箭头的事物“is a”箭头指向的事物。泛化关系是一般和特殊关系，发出箭头的一方代表特殊的一方，箭头指向的一方代表一般一方。特殊一方继承了一般方的特性并增加了新的特性。 |  |

**（一）关联(Association)**

　　表示参与者与用例之间的通信，任何一方都可发送或接受消息。

　　【箭头指向】：指向消息接收方



**（二）泛化(Inheritance)**

　　就是通常理解的继承关系，子用例和父用例相似，但表现出更特别的行为；子用例将继承父用例的所有结构、行为和关系。子用例可以使用父用例的一段行为，也可以重载它。父用例通常是抽象的。

　　【箭头指向】：指向父用例



**（三）包含(Include)**

包含关系用来把一个较复杂用例所表示的功能分解成较小的步骤。

　　【箭头指向】：指向分解出来的功能用例



## 用例表

用例模板表：

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | [用例编号，如：UC-01] | 名称 | [用例名称，即用例图汇总的用例的描述] |
| 执行者 | [用户、角色等] | 优先级 | □高 □中 □低 |
| 描述 | [简单地描述本用例，重点说明执行者的目标] | | |
| 前置条件 | [列出执行本用例前必须存在的系统状态，如：必须录入什么数据，须先实现是其他什么用例等。注意排除特殊情况，不要写类似“登录系统”等每个用例几乎都需要具备的前置条件] | | |
| 基本流程 | [说明在“正常”情况下，最常用的流程。通常是执行者和系统之间交互的文字描述] | | |
| 结束状况 | [列出在“正常”结束情况下的用例的结果] | | |
| 可选流程1 | [说明和基本流程不同的其他可能的流程] | | |
| 可选流程n | [说明和基本流程不同的其他可能的流程] | | |
| 异常流程 | [说明出现错误或者其他异常情况和基本流程的不同之处] | | |
| 说明 | [对本用例的补充说明，如：业务概念、业务规则等] | | |

## 实例



# 时序图（Timing Diagram）

## 概念

时序图是具体描述单个或多个对象状态变化的时间点以及维持特定状态的时间段。

时序图在实时应用、嵌入式软件设计等领域会使用到。

## 组成

**（一）组成元素**

组成元素：时间约束、持续时间约束、生命线、状态、条件、事件。

时间进行线：在生命线内用高低起伏代表事件的进行与状态的转移。

事件：这里的事件要注意的是以纯文字来表达的。

**（二）表示方法**

线条表示法：

区间表示法：

## 实例

