**UML教程**

# 类图Class Diagram

|  |  |  |
| --- | --- | --- |
| **图形** | | **符号描述** |
|  | | 类是对一组具有相同属性、操作、关系和语义的对象的抽象，它是面向对象系统组织结构的核心，包括名称部分（Name）、属性部分（Attribute）和操作部分（Operation） |
| **类属性的语法为：** | | |
| [可见性] 属性名 [:类型] [=初始值] [{属性字符串}]  可见性：公有（Public）“+”、私有（Private）“-”、受保护（Protected）“#” | | |
| **类操作的语法为：** | | |
| [可见性] 操作名 [(参数表)] [:返回类型] [{属性字符串}]  可见性：公有（Public）“+”、私有（Private）“-”、受保护（Protected）“#”、包内公有（Package）“~”  **参数表：**  定义方式：“名称：类型”；若存在多个参数，将各个参数用逗号隔开；参数可以具有默认值；  **属性字符串：**  在操作的定义中加入一些除了预定义元素之外的信息。 | | |
|  |  | |

类与类之间的关系，各种关系的强弱顺序：

**泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖**

|  |  |
| --- | --- |
| **关系** | **符号描述** |
| 泛化  Generalization |  |
| 实现  Realization |  |
| 一般关联  Association |  |
| 关联-聚合  Aggregation |  |
| 关联-组合  Composition |  |
| 依赖  Dependency |  |

# 用例图Use Case Diagram

|  |  |
| --- | --- |
| **图形** | **符号描述** |
| **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。 |
| **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。 |
| **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。 |

用例中的关系图



|  |  |
| --- | --- |
| **图形** | **符号描述** |
| **关联(Association)** | http://pic001.cnblogs.com/images/2012/1/2012013015250613.gif  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** |
| **泛化(Inheritance)** | http://pic001.cnblogs.com/images/2012/1/2012013015260081.gif  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** |
| **包含(Include)** | http://pic001.cnblogs.com/images/2012/1/2012013015265841.gif  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 |
| **扩展(Extend)** | http://pic001.cnblogs.com/images/2012/1/2012013015274296.gif  扩展关系是指**用例功能的延伸**，相当于为基础用例提供一个**附加功能**。【箭头指向】：指向基础用例 |
| **依赖**  **(Dependency)** | http://pic001.cnblogs.com/images/2012/1/2012013015290255.gif  以上4种关系，是UML定义的标准关系。但VS2010的用例模型图中，添加了依赖关系，用带箭头的虚线表示，表示源用例依赖于目标用例。【箭头指向】：指向被依赖项 |
| **项目(Artifact)** | http://pic001.cnblogs.com/images/2012/1/2012013015295762.gif  用例图虽然是用来帮助人们形象地理解功能需求，但却没多少人能够通看懂它。很多时候跟用户交流甚至用Excel都比用例图强，VS2010中引入了“项目”这样一个元素，以便让开发人员能够**在用例图中链接一个普通文档**。用依赖关系把某个用例依赖到项目上：然后把项目-》属性的Hyperlink设置到你的文档上；这样当你在用例图上双击项目时，就会打开相关联的文档。 |
| **注释(Comment)** | http://pic001.cnblogs.com/images/2012/1/2012013015305290.gif |

# 时序图Sequence Diagram

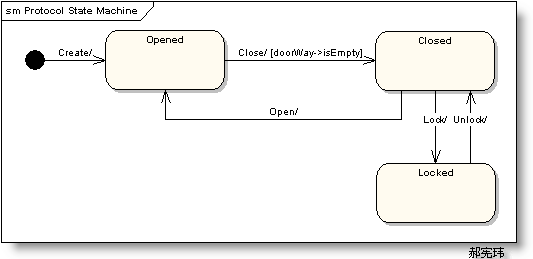
时序图（Sequence Diagram）是显示对象之间交互的图，这些对象是按时间顺序排列的。顺序图中显示的是参与交互的对象及其对象之间消息交互的顺序。时序图中包括的建模元素主要有：对象（Actor）、生命线（Lifeline）、控制焦点（Focus of control）、消息（Message）等。

|  |  |
| --- | --- |
| **元素** | **符号描述** |
| **参与者**  **(Actor)** | 系统角色，可以是人、及其甚至其他的系统或者子系统。 |
| **对象**  **(Object)** | 对象包括三种命名方式：  第一种方式包括**对象名和类名**；  第二中方式只显示**类名**不显示对象名，即表示他是一个匿名对象；  第三种方式只显示**对象名**不显示类明。 |
| **生命线（Lifeline）** | http://images.cnblogs.com/cnblogs_com/ywqu/Sequence%20Diagram/lifeline.gif  生命线在顺序图中表示为从对象图标向下延伸的**一条虚线**，表示对象存在的时间，如下图 |
| **控制焦点（Focus of Control）** | http://images.cnblogs.com/cnblogs_com/ywqu/Sequence%20Diagram/ElementFOC.jpg  控制焦点是顺序图中表示时间段的符号，在这个时间段内对象将执行相应的操作。**用小矩形表示**，如下图。 |
| **消息（Message）** | 消息一般分为  同步消息（Synchronous Message），  异步消息（Asynchronous Message）和返回消息（Return Message）.如下图所示：http://images.cnblogs.com/cnblogs_com/ywqu/Sequence%20Diagram/Message.gif  **同步消息=调用消息（Synchronous Message）**  **消息的发送者把控制传递给消息的接收者，然后停止活动，等待消息的接收者放弃或者返回控制**。用来表示同步的意义。  **异步消息（Asynchronous Message）**  **消息发送者通过消息把信号传递给消息的接收者，然后继续自己的活动，不等待接受者返回消息或者控制**。异步消息的接收者和发送者是并发工作的。  **返回消息（Return Message）**  返回消息表示从过程调用返回 |
| **自关联消息（Self-Message）** | 表示方法的自身调用以及一个对象内的一个方法调用另外一个方法。http://images.cnblogs.com/cnblogs_com/ywqu/Sequence%20Diagram/SelfMessage.gif |

# 状态图State chart Diagram

状态图（Statechart Diagram）主要用于描述一个对象在其**生存期间的动态**行为，表现为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的动作（Action）。一般可以用状态机对一个对象的**生命周期建模**，状态图用于显示状态机（State Machine Diagram），重点在与描述状态图的控制流。

如下图例子，状态机描述了门对象的生存期间的状态序列，引起转移的事件，以及因状态转移而伴随的动作（Action）。



状态有Opened、Closed、Locked。

事件有Open、Close、Lock和Unlock。

注意：

1、**并不是所有的事件都会引起状态的转移**，比如当门是处于【Opened】状态，不能进行【Lock】事件。

2、**转移（Transition）有警备条件**（guard condition），比如只有doorWay->isEmpty条件满足时，才会响应事件。

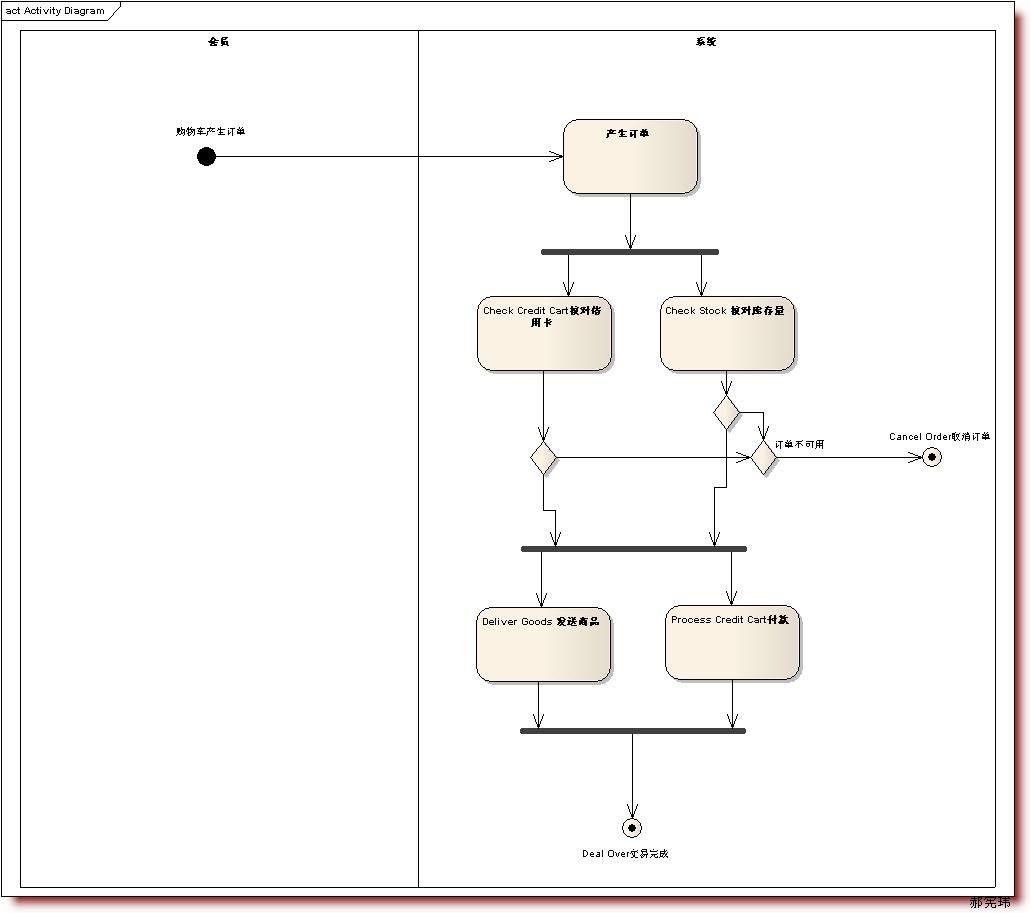
|  |  |
| --- | --- |
| **元素** | **符号描述** |
| **状态（States）** | http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm02.gif  指在对象的生命周期中的某个条件或者状况，在此期间对象将满足某些条件、执行某些活动、等待某些事件。所有对象都有状态，状态是对象执行了一系列活动的结果，**当某个事件发生后，对象的状态将发生变化**。**状态用圆角矩形表示**  初态和终态（Initial and Final States）  初态用实心圆点表示，终态用圆形内嵌圆点表示。  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm03.gif |
| **转移（Transitions）** | 转移（Transitions）是**两个状态之间的一种关系**，表示对象将在源状态（Source State）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界条件满足时进入目标状态（Target State）  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm04.gif  事件标记（Trigger）：是转移的诱因，可以是一个信号，事件、条件变化（a change in some condition）和时间表达式。  警界条件（Guard Condition）：当警界条件满足时，事件才会引发转移（Transition）。  结果（Effect）：对象状态转移后的结果。 |
| **动作**  **（State Actions）** | 动作（Actions）是一个可执行的原子操作,也就是说动作是不可中断的，其执行时间是可忽略不计的。  在上例中，对象状态转移后的结果显示在转移线上，如果目标状态有许多转移，而且每个转移有相同的结果，这时把转移后的结果（Effect）展示在目标状态中（Target State）更好一些，可以定义进入动作（Entry Action ）和退出动作（Exit Action），如下图  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm05.gif |
| **自身转移（Self-Transitions）** | 状态可以有返回自身状态的转移，称之为自身转移（Self-Transitions）  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm06.gif  2S后，Poll input事件执行，转移到自己状态【Waiting】 |
| **组合状态（Compound**  **States）** | 嵌套在另外一个状态中的状态称之为子状态（sub-state）,一个含有子状态的状态被称作组合状态（Compound States）。如下图，【Check PIN】是组合状态，【Enter PIN】是子状态。    也可用以下方式进行描述  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm08.gif  如上图，状态机【Check PIN】的细节被分割到另外一个图中了。 |
| **进入节点（Entry Point）** | 如下图所示，由于一些原因并不会执行初始化（initialization），而是直接通过一个节点进入状态【Ready】，则此节点称之为进入节点（Entry Point）  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm09.gif |
| **退出节点（Exit Point）** | http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm11.gif |
| **历史状态**  **（History States）** | 历史状态是一个伪状态（Pseudostate）,其目的是记住从组合状态中退出时所处的子状态，当再次进入组合状态，可直接进入这个子状态，而不是再次从组合状态的初态开始。  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm15.gif  在上图的状态图中，正常的状态顺序是:【Washing】- >【Rinsing】->【Spinning】。  如果是从状态【Rinsing】突然停电（Power Cut）退出，,洗衣机停止工作进入状态【Power Off】，当电力恢复时直接进入状态【Running】。 |
| **并发区域（Concurrent Regions）** | 状态图可以分为区域，而区域又包括退出或者当前执行的子状态。说明组合状态在某一时刻可以同时达到多个子状态。如下图刹车系统，同时进入前刹车【Applying Front Brakes】状态和后刹车【Applying Rear Brakes】状态。  http://images.cnblogs.com/cnblogs_com/ywqu/Statechart%20Diagram/sm16.gif |

# 活动图Activity Diagram

活动图是UML用于对系统的动态行为建模的另一种常用工具，它**描述活动的顺序，展现从一个活动到另一个活动的控制流**。活动图在本质上是一种**流程图**。活动图**着重表现从一个活动到另一个活动的控制流**，是内部处理驱动的流程。

|  |  |
| --- | --- |
| **元素** | **符号描述** |
| **活动状态图（Activity）** | 活动状态用于表达状态机中的非原子的运行，其特点如下：  (1)、活动状态可以分解成其他子活动或者动作状态。  (2)、活动状态的内部活动可以用另一个活动图来表示。  (3)、和动作状态不同，活动状态可以有入口动作和出口动作，也可以有内部转移。  (4)、动作状态是活动状态的一个特例，如果某个活动状态只包括一个动作，那么它就是一个动作状态。  UML中活动状态和动作状态的图标相同，但是活动状态可以在图标中给出入口动作和出口动作等信息。  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NActivities.gif |
| **动作状态**  **（Actions）** | 动作状态是指原子的，不可中断的动作，并在此动作完成后通过完成转换转向另一个状态。动作状态有如下特点：  (1)、动作状态是原子的，它是构造活动图的最小单位。  (2)、动作状态是不可中断的。  (3)、动作状态是瞬时的行为。  (4)、动作状态可以有入转换，入转换既可以是动作流，也可以是对象流。动作状态至少有一条出转换，这条转换以内部的完成为起点，与外部事件无关。  (5)、动作状态与状态图中的状态不同，它不能有入口动作和出口动作，更不能有内部转移。  (6)、在一张活动图中，动作状态允许多处出现。  UML中的动作状态图用平滑的圆角矩形表示，如下：  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NActions.gif |
| **动作状态约束（Action Constraints）** | 动作状态约束：用来约束动作状态。如下图展示了动作状态的前置条件和后置条件  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NAction-Constraints.gif |
| **动作流**  **（Control Flow）** | 动作之间的转换称之为动作流，活动图的转换用带箭头的直线表示，箭头的方向指向转入的方向。  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NControl-Flow.gif |
| **开始节点**  **（Initial Node）** | 开始节点：表示成实心黑色圆点  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NInitial-Node.gif |
| **终止节点**  **（Final Node）** | 分为活动终止节点（activity final nodes）和流程终止节点（flow final nodes）。  活动终止节点表示整个活动的结束  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NFinal-Node.gif  而流程终止节点表示是子流程的结束。  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/Nflow-final.gif |
| **对象**  **（Objects）** | http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NObjects.gif |
| **数据存储对象（DataStore）** | http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/Ndata-store.gif |
| **对象流（Object Flows）** | 对象流是动作状态或者活动状态与对象之间的依赖关系，表示动作使用对象或动作对对象的影响。用活动图描述某个对象时，可以把涉及到的对象放置在活动图中并用一个依赖将其连接到进行创建、修改和撤销的动作状态或者活动状态上，对象的这种使用方法就构成了对象流。  对象流中的对象有以下特点：  (1)、一个对象可以由多个动作操作。  (2)、一个动作输出的对象可以作为另一个动作输入的对象。  (3)、在活动图中，同一个对象可以多次出现，它的每一次出现表面该对象正处于对象生存期的不同时间点。  对象流用带有箭头的虚线表示。如果箭头是从动作状态出发指向对象，则表示动作对对象施加了一定的影响。施加的影响包括创建、修改和撤销等。如果箭头从对象指向动作状态，则表示该动作使用对象流所指向的对象。  状态图中的对象用矩形表示，矩形内是该对象的名称，名称下的方括号表明对象此时的状态。  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NObject-Flows.gif |
| **分支与合并（Decision and Merge Nodes）** | 分支与合并用菱形表示  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NDecision-and-Merge-Nodes.gif |
| **分叉与汇合（Fork and Join Nodes）** | 分为水平方向和垂直方向  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NForkandJoinNodes.gif  对象在运行时可能会存在两个或多个并发运行的控制流，为了对并发的控制流建模，UML中引入了分叉与汇合的概念。分叉用于将动作流分为两个或多个并发运行的分支，而汇合则用于同步这些并发分支，以达到共同完成一项事务的目的。 |
| **异常处理（Exception Handler）** | 当受保护的活动发生异常时，触发异常处理节点。  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NException-Handlers.gif |
| **活动中断区域（Interruptible Activity Region）** | 活动中断区域围绕一些可被中断的动作状态图。比如下图，正常情况下【Process Order】顺序流转到【Close Order】，订单处理流程完毕；但在【Process Order】过称中，会发送【Cancel Order】请求，这时会流转到【Cancel Order】，从而订单处理流程结束  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NInterruptible-Activity-Region.gif |
| **泳道（Partition）** | 泳道将活动图中的活动划分为若干组，并把每一组指定给负责这组活动的业务组织，即对象。在活动图中，泳道区分了负责活动的对象，它明确地表示了哪些活动是由哪些对象进行的。在包含泳道的活动图中，每个活动只能明确地属于一个泳道。  泳道是用垂直实线绘出，垂直线分隔的区域就是泳道。在泳道的上方可以给出泳道的名字或对象的名字，该对象负责泳道内的全部活动。泳道没有顺序，不同泳道中的活动既可以顺序进行也可以并发进行，动作流和对象流允许穿越分隔线。  http://images.cnblogs.com/cnblogs_com/ywqu/Activity%20Diagram/NPartition.gif |

实例：



1、 泳道分为：会员泳道和系统泳道。会员选择商品并加入购物车，系统完成订单生成及其支付完毕。

2、 开始节点：会员添加商品到购物车，点击【订单确认】，开始交于系统处理订单流程

3、 结束节点：商品发送完毕和付款成功，订单处理流程结束

4、 活动状态：产生订单、Check Credit Cart核对信用卡、Check Stock 核对库存量、Deliver Goods 发送商品、Process Credit Cart付款

5、 分叉与汇合：【产生订单】份叉为检查库存量和会员支付金额是否足够，如果不足，取消订单，如过库存量和支付金额足够，发送商品和付款，最后汇合为订单完成。

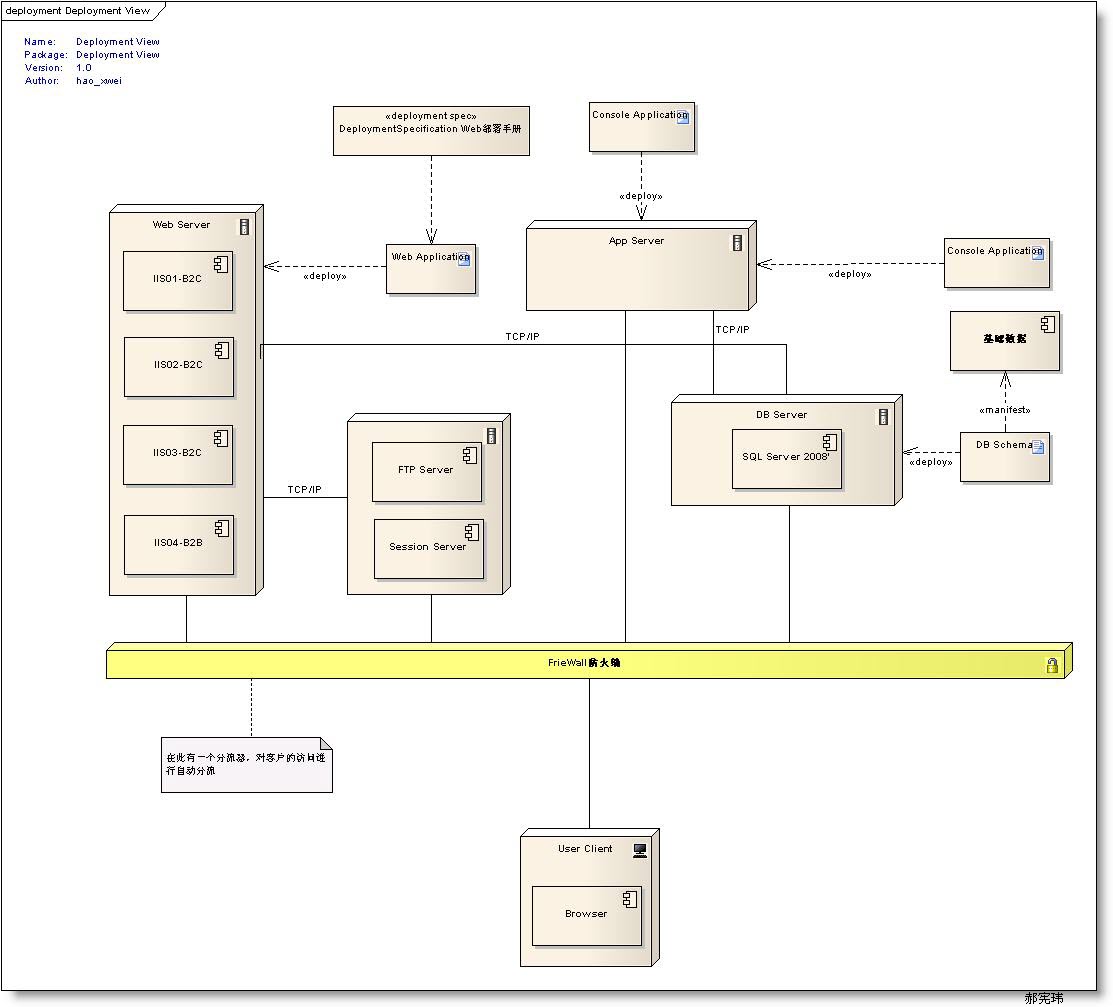
# 构件图Component Diagram

# 部署图Deployment Diagram

部署图描述的是系统运行时的结构，展示了硬件的配置及其软件如何部署到网络结构中。一个系统模型只有一个部署图，部署图通常用来帮助理解分布式系统。

|  |  |
| --- | --- |
| **图形** | **符号描述** |
| **结点**  **（Node）** | 结点是存在与运行时的**代表计算机资源的物理元素**，可以是硬件也可以是运行其上的软件系统，比如64主机、Windows server 2008操作系统、防火墙等。结点用三维盒装表示，如下图：  http://images.cnblogs.com/cnblogs_com/ywqu/Deployment%20Diagram/dd02.gif |
| **结点实例**  **（Node Instance）** | 结点实例名称格式如下Node Instance : node  与结点的区别在于名称有下划线和结点类型前面有冒号，冒号前面可以有示例名称也可以没有示例名称，如下图  http://images.cnblogs.com/cnblogs_com/ywqu/Deployment%20Diagram/dd03.gif |
| **结点类型**  **（Node Stereotypes）** | 结点类型有：«cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc»，并在结点的右上角用不同的图标表示，如下图  http://images.cnblogs.com/cnblogs_com/ywqu/Deployment%20Diagram/dd04.gif |
| **物件**  **（Artifact）** | 物件是**软件开发过程中的产物**，包括过程模型（比如用例图、设计图等等）、源代码、可执行程序、设计文档、测试报告、需求原型、用户手册等等。物件表示如下，带有关键字«artifact»和文档图标  http://images.cnblogs.com/cnblogs_com/ywqu/Deployment%20Diagram/dd05.gif |
| **连接（Association）** | 结点之间的连线**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（Association），如下图所示，连接中有网络协议。  http://images.cnblogs.com/cnblogs_com/ywqu/Deployment%20Diagram/dd06.gif |
| **结点容器（Node as Container）** | 一个结点可以包括其他的结点，比如组件或者物件，则称此结点为结点容器（Node as Container）。如下图所示，结点（Node）包容了物件（Artifact）。  http://images.cnblogs.com/cnblogs_com/ywqu/Deployment%20Diagram/dd01.gif |

部署图示例



分析：

1、Web Server服务器

一台Web服务器**预装4个操作系统**及其之上的**4个IIS**，由于客户访问量大，3个用来部署B2C Web, 1个用来部署B2BB2E Web.

2、Web访问量分流设备

根据网站流量，自动定位客户访问流量小的服务器。

3、FTP Server And Session Server服务器

网站所有的图片都统一上传到FTP服务器上，同时B2CWeb下的Session统一转移到此服务器上。

4、DB Server服务器

数据库服务器

5、App Server服务器

定时执行排程（控制台程序）的服务器，用于與倉庫系統交互。

6、FrieWall防火墙

所有对服务器的操作通过防火墙过滤。

7、User Client

用户个人PC，预装有浏览器。

# 组合结构图Composite Structure Diagram

# 协作图Collaboration Diagram