4-3 基本架构风格

2019年4月14日 10:27

1 什么是“架构风格”

从“建筑风格”开始

- 建筑风格等同于建筑体系结构的一种可分类的模式，通过诸如外形、技术和材料等形态上的特征加以区分)。
- 之所以称为“风格”，是因为经过长时间的实践，它们已经被证明具有良好的工艺可行性、性能与实用性，并可直接用来遵循与模仿(复用)。

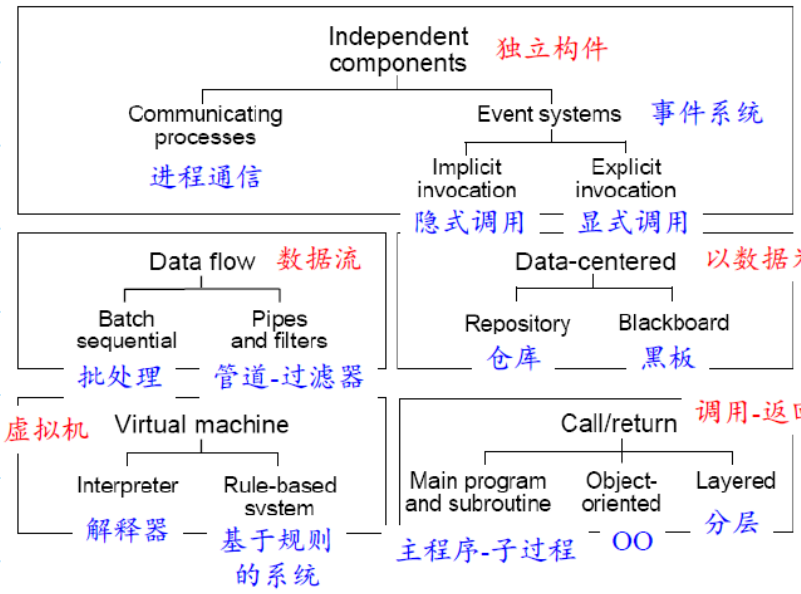
软件体系结构风格

- 软件系统同建筑一样，也具有若干特定的“风格”(softwarearchitecturalstyle)；
  - 这些风格在实践中被多次设计、应用，已被证明具有良好的性能、可行性和广泛的应用场景，可以被重复使用；
  - 实现“软件体系结构级”的复用。
- 定义：
  - 描述特定领域中软件系统家族的组织方式的惯用模式，反映了领域中众多系统所共有的结构和语义特性，并指导如何将各个模块和子系统有效地组织成一个完整的系统。

“软件架构风格”的组成

- 一组构件类型
- 一组连接件类型/交互机制
- 这些构件的拓扑分布
- 一组对拓扑和行为的约束
- 一些对风格的成本和收益的描述

经典架构风格

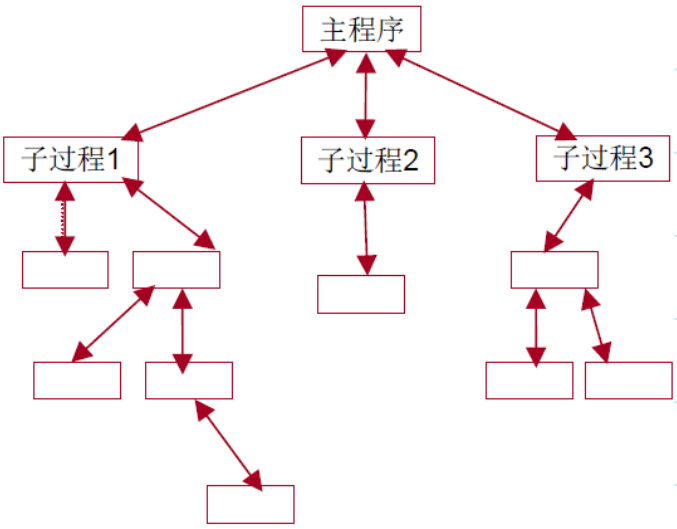
- 

新型的架构风格
点对点(P2P)
网格(grid)
Web 2.0
面向服务的架构(SOA)
事件驱动的架构(EDA)
软件即服务(SaaS)
云计算与虚拟化

2 调用返回风格

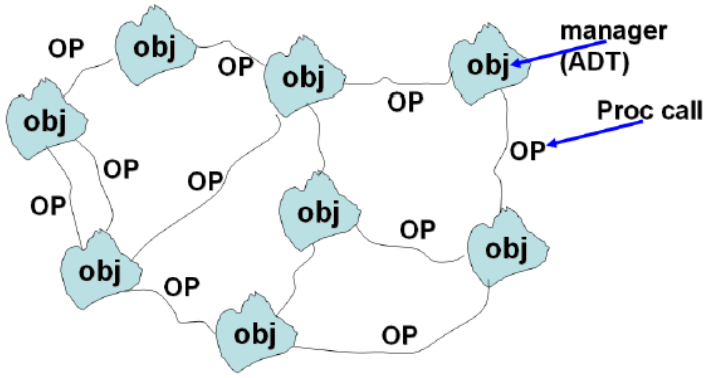
- 以函数/对象作为基本构造单元，彼此通过callreturn相互连接
- 特征：不考虑分层，主要遵循同步调用方式

主程序-子过程

- 该风格是结构化程序设计的一种典型风格，从功能的观点设计系统，通过逐步分解和逐步细化，得到系统架构。
  - 构件：主程序、子程序
  - 连接器：调用返回机制
  - 拓扑结构：层次化结构
- 本质：将大系统分解为若干模块(模块化)，主程序调用这些模块实现完整的系统功能
- 

面向对象风格

- 系统被看作对象的集合，每个对象都有一个它自己的功能集合；
- 数据及作用在数据上的操作被封装成抽象数据类型(ADT)；
- 只通过接口与外界交互，内部的设计决策则被封装起来
  - 构件：类和对象
  - 连接件：对象之间通过函数调用、消息传递实现交互
-

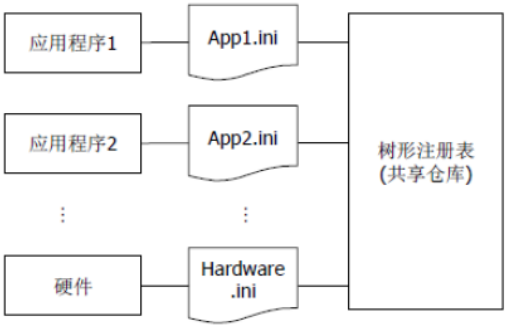


3 以数据为中心的风格

- 数据的持久化存储被分离出去，形成两层结构

例1：注册表(WindowsRegistry)

- 最初，硬件/软件系统的配置信息均被各自保存在一个配置文件中(ini)；
- 这些文件散落在系统的各个角落，很难对其进行维护；
- 为此，引入注册表的思想，将所有ini文件集中起来，形成共享仓库，为系统运行起到了集中的资源配置管理和控制调度的作用。
- 注册表中存在着系统的所有硬件和软件配置信息，如启动信息、用户、BIOS、各类硬件、网络、INI文件、驱动程序、应用程序等；
- 注册表信息影响或控制系统/应用软件的行为，应用软件安装/运行/卸载时对其进行添加/修改/删除信息，以达到改变系统功能和控制软件运行的目的
-



例2：剪贴板(Clipboard)

- 剪贴板是一个用来进行短时间的数据存储并在文档/应用之间进行数据传递和交换的软件程序
  - 用来存储带传递和交换信息的公共区域(形成共享数据仓库)；
  - 不同的应用程序通过该区域交换格式化的信息；

- 访问剪贴板的方式：Copy&Paste/Cut

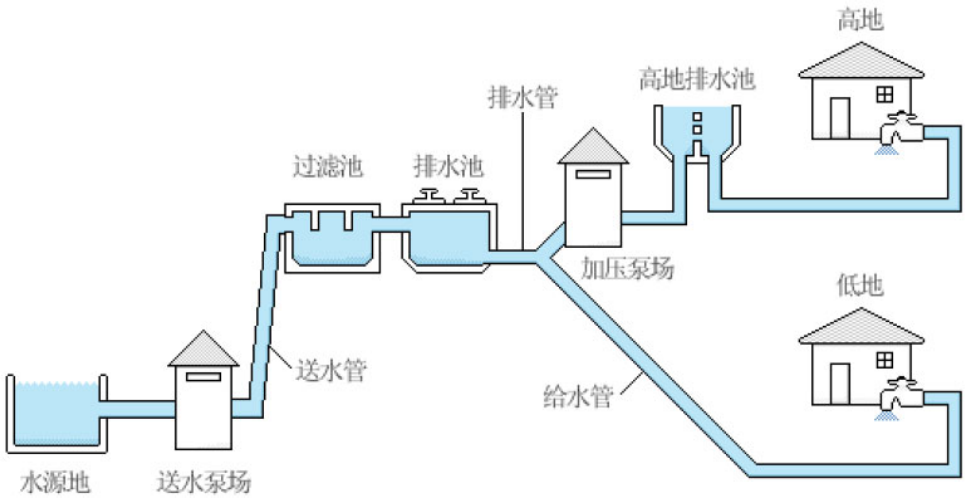
以数据为中心的体系结构风格

- 仓库风格（数据被动存储）
- 黑板风格（仓库风格的变体，数据发生变化后，主动通知相关用户）

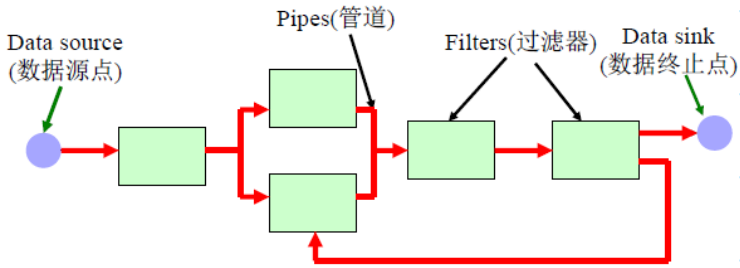
4 数据流风格

- 把系统分解为几个序贯的处理步骤，这些步骤之间通过数据流连接，一个步骤的输出是另一个步骤的输入；
- 每个处理步骤由一个过滤器构件(Filter)实现；
- 处理步骤之间的数据传输由管道(Pipe)负责

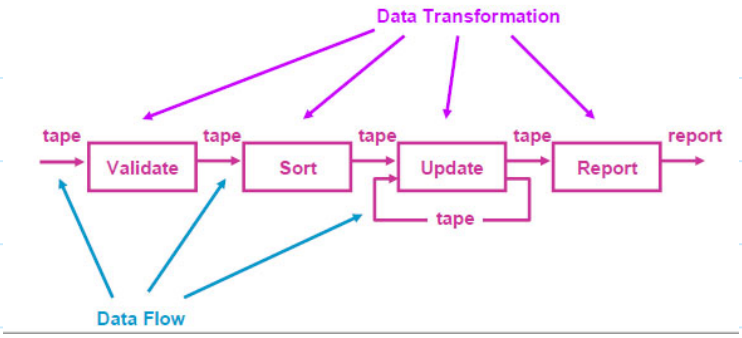
现实中的“数据流”体系结构

- 

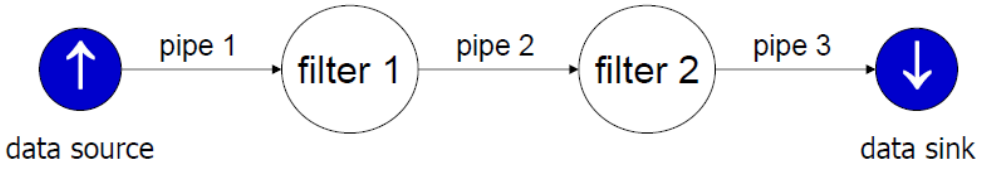
软件中的数据流风格

- 管道过滤器风格
  - 

- 顺序批处理风格

- 

软件中的数据流风格

- 语境：数据源源不断的产生，系统需要对这些数据进行若干处理(分析、计算、转换等)
- 解决方案：
  - 把系统分解为几个序贯的处理步骤，这些步骤之间通过数据流连接，一个步骤的输出是另一个步骤的输入；
  - 每个处理步骤由一个过滤器构件(Filter)实现；
  - 处理步骤之间的数据传输由管道(Pipe)负责。
- 每个处理步骤(过滤器)都有一组输入和输出，过滤器从管道中读取输入的数据流，经过内部处理，然后产生输出数据流并写入管道中。
- 

- filter1 {
  Read data *d* from pipe1;
  Deal with *d* and transform it to *d'*;
  Write *d'* to pipe2;
  }

  filter2 {
  Read data *d'* from pipe2;
  Deal with *d'* and transform it to *d''*;
  Write *d''* to pipe3;
  }

现实中的典型应用领域：

编译器、Unix管道、图像处理、信号处理、网络监控与管理等

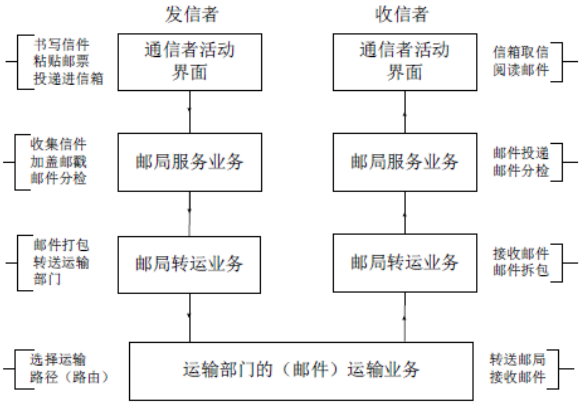
5 层次风格

- 除了数据被分离出去，软件系统的其他各部分进一步分离，形成更复杂的层次结构

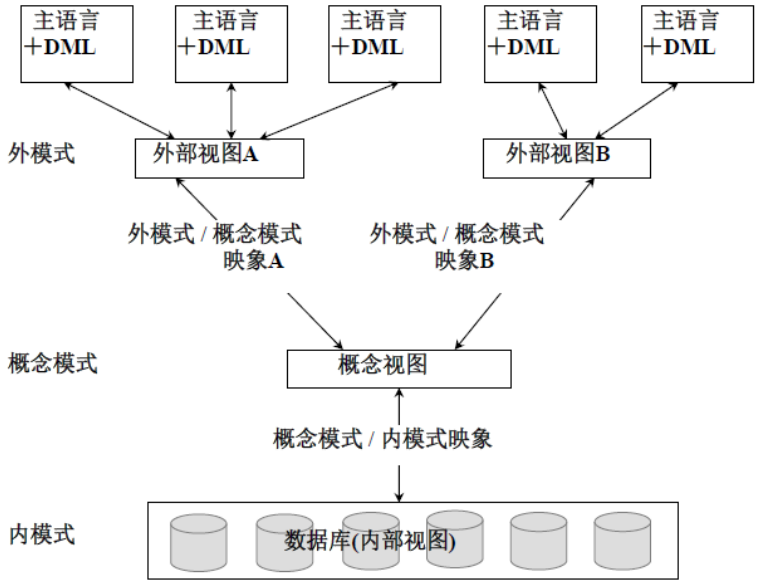
层次结构

- 层次化早已经成为一种复杂系统设计的普遍性原则；
- 两个方面的原因：
  - 事物天生就是从简单的、基础的层次开始发生的；

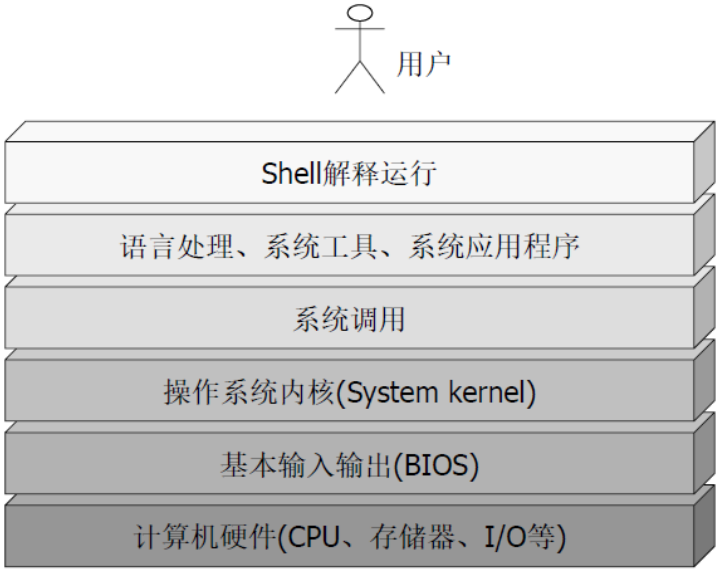
- 众多复杂软件设计的实践，大到操作系统，中到网络系统，小到一般应用，几乎都是以层次化结构建立起来的。



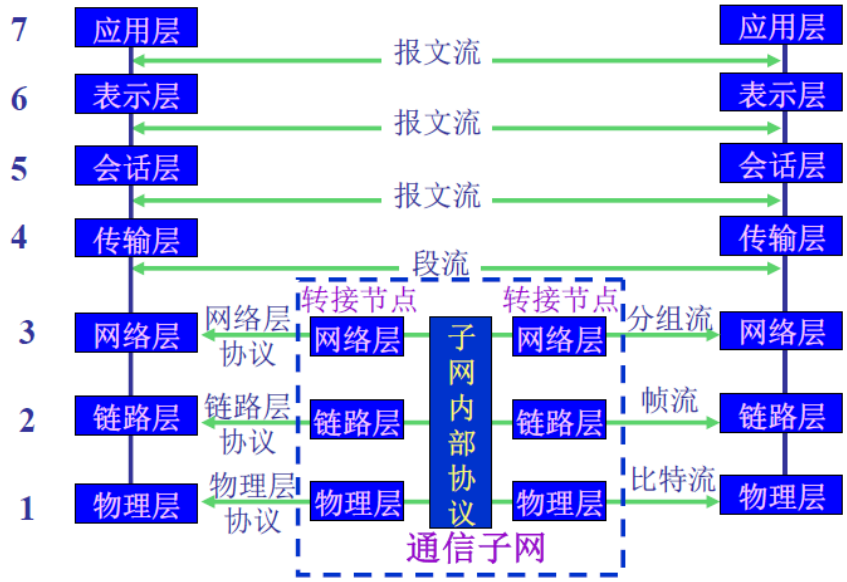
DBMS中的“三级模式两层映像”



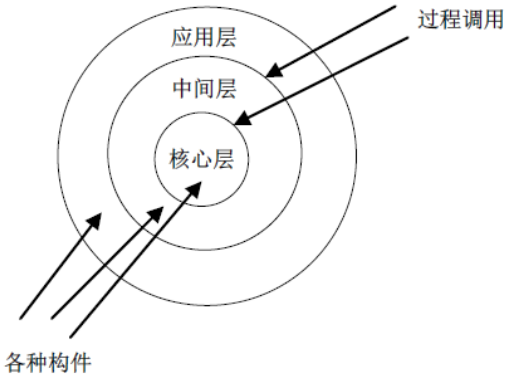
计算机操作系统的层次结构



网络的分层模型

- 

层次系统

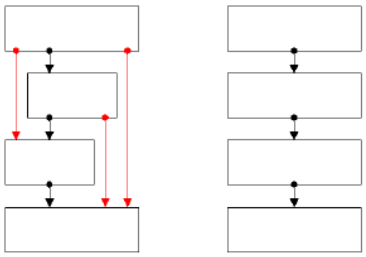
- 在层次系统中，系统被组织成若干个层次，每个层次由一系列构件组成；
- 层次之间存在接口，通过接口形成call/return的关系
  - 下层构件向上层构件提供服务
  - 上层构件被看作是下层构件的客户端
- 

层次系统的优点

- 这种风格支持基于可增加抽象层的设计，允许将一个复杂问题分解成一个增量步骤序列的实现。
- 不同的层次处于不同的抽象级别：
  - 越靠近底层，抽象级别越高；
  - 越靠近顶层，抽象级别越低；
- 由于每一层最多只影响两层，同时只要给相邻层提供相同的接口，允许每层用不同的方法实现，同样为软件复用提供了强大的支持。

严格分层和松散分层

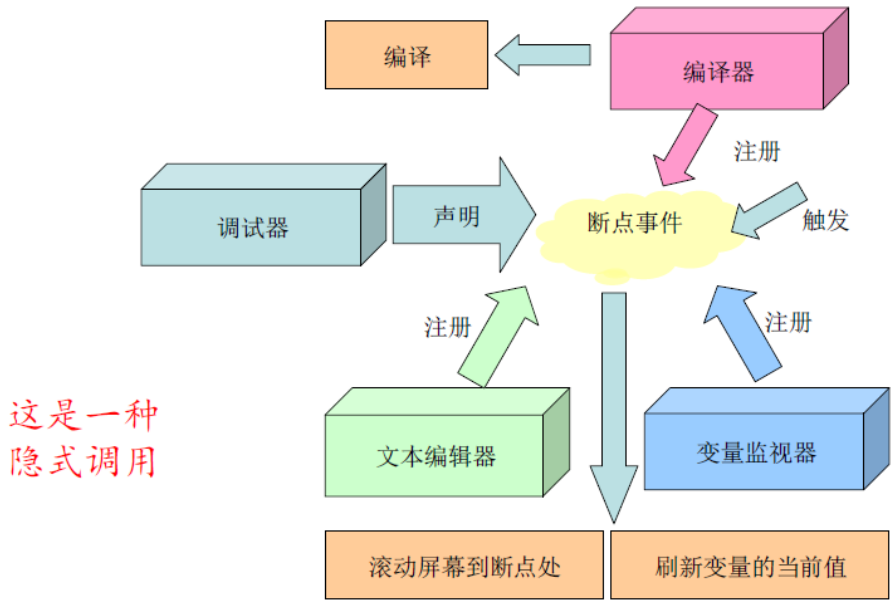
- 严格分层系统要求严格遵循分层原则，限制一层中的构件只能与对等实体以及与它紧邻的下面一层

进行交互
    - 优点：修改时的简单性
    - 缺点：效率低下
- 松散的分层应用程序放宽了此限制，它允许构件与位于它下面的任意层中的组件进行交互
    - 优点：效率高
    - 缺点：修改时困难
- 
- 任何计算机科学问题都可以通过增加一层来解决！
- 任何计算机性能问题都可以通过去掉一个间接层来解决！

7 事件风格

- 与“调用-返回”方式相对应，从同步调用变为了异步调用

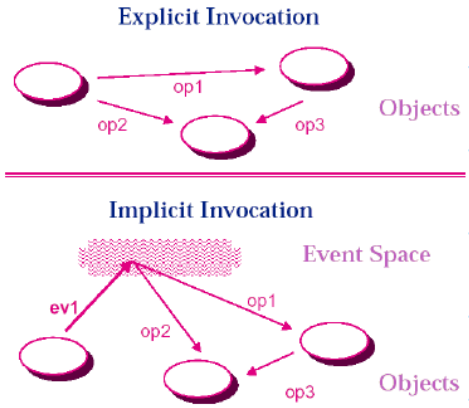
代码调试器的工作流程

- 

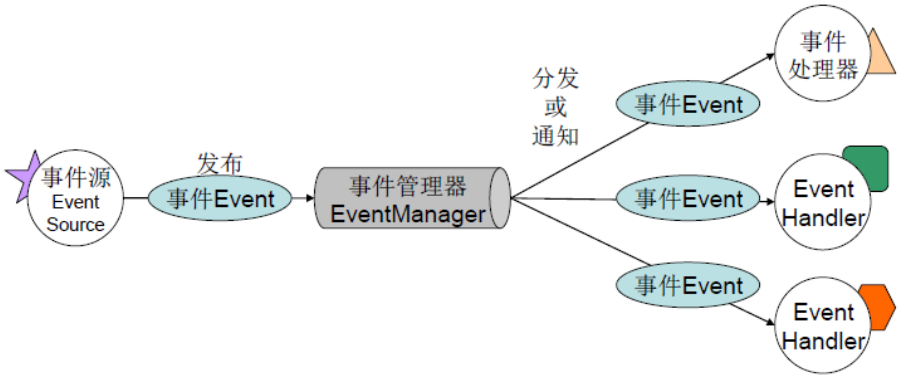
显式调用vs隐式调用

- 显式调用：
    - 各个构件之间的互动是由显性调用函数或程序完成的。
    - 调用过程与次序是固定的、预先设定的。
- 隐式调用：
    - 调用过程与次序不是固定的、预先未知；

- 各构件之间通过事件的方式进行交互;



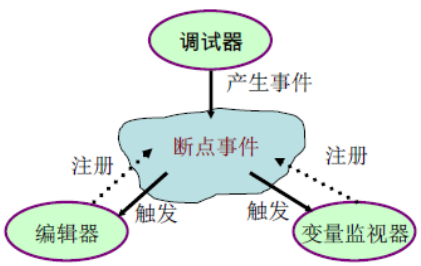
事件系统的基本构成



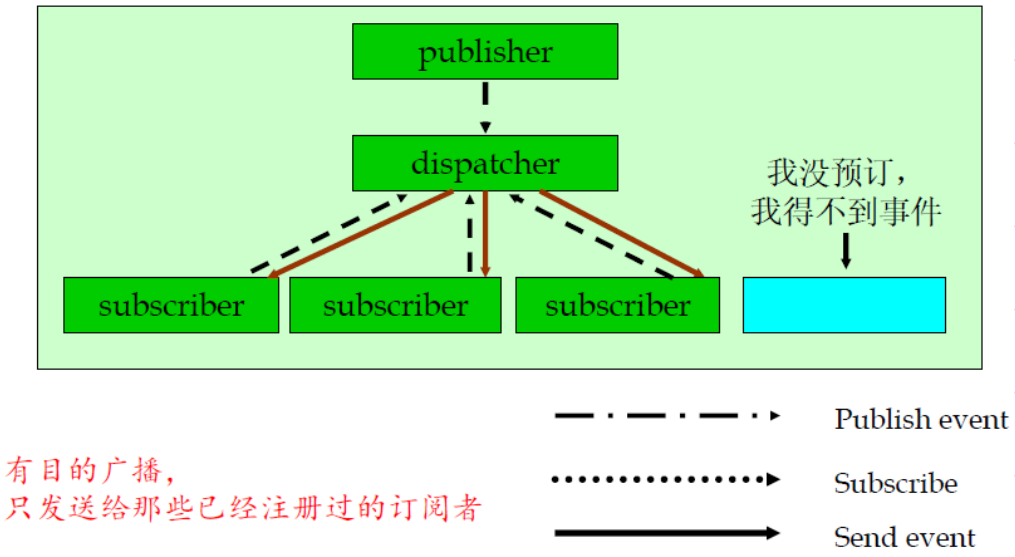
功能	描述
分离的交互	事件发布者并不会意识到事件订阅者的存在。
多对多通信	采用发布/订阅消息传递，一个特定事件可以影响多个订阅者。
基于事件的触发器	控制流由接收者确定（基于发布的事件）。
异步	通过事件消息传递支持异步操作。

回到调试器的例子

- EventSource:debugger(调试器)
- EventHandler:editorandvariablemonitor(编辑器与变量监视器)
- EventManager:IDE(集成开发环境)
- 编辑器与变量监视器向调试器注册，接收“断点事件”；
- 一旦遇到断点，调试器发布事件，从而触发“编辑器”与“变量监测器”
- 编辑器将源代码滚动到断点处，变量监测器则更新当前变量值并显示出来。

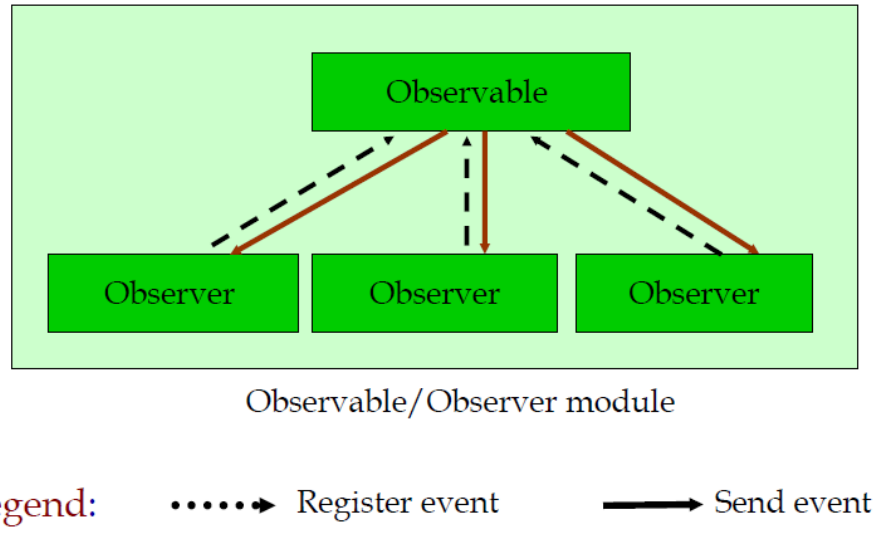


事件风格的实现策略之一：选择广播式

- 

有目的广播，
只发送给那些已经注册过的订阅者

事件风格的实现策略之二：观察者模式

- 

9 (*仅了解)面向服务的体系结构

SOA中可用的基本构件是“服务”；

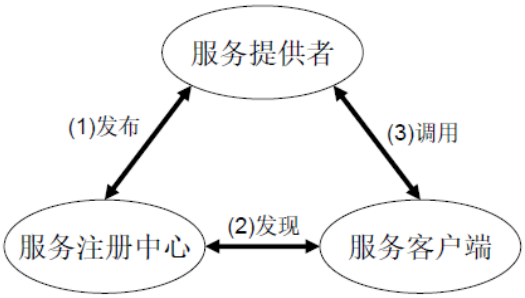
- 一个服务是一个服务提供者为一个服务消费者获得其想要的最终结果所提供的一个软件单元。
- 从外特性上看，一个服务被定义为显式的、独立于服务具体实现技术细节的接口。
- 从内特性上看，服务封装了可复用的业务功能，这些功能通常是大粒度业务，如业务过程、业务活动等。服务的实现可采用任何技术平台，如J2EE、Net等。

服务之间的“连接件”

- 通过接口，采用位置透明的、可互操作的协议进行调用，与客户端以“松散耦合”(looselycoupling)的方式绑定在一起。
- SOA中所有协议均是基于XML的文本文件。

SOA的基本体系结构样式之一：发布访问

- 发布(Publish)：为了使服务可访问，需要发布服务描述以使服务使用者可以发现它
- 发现(Find)：服务请求者定位服务，方法是查询服务注册中心来找到满足其标准的服务
- 调用(invoke)：在检索到服务描述之后，服务使用者继续根据服务描述中的信息来调用服务。
-



WebService中该模式的实现机制

- WSDL：Web服务描述语言
  - 用于服务接口的描述——Whatcantheservicedo?
- UDDI：统一描述、发现和集成协议
  - 服务使用者通过UDDI发现相应的服务并据此将服务集成在自身的系统中——Whatkindofservicesareneeded?
- SOAP：简单对象访问协议
  - 用户在服务客户端与服务提供者之间传递信息，通过HTTP或JMS等各类基于文本的消息传递协议来运输

软件体系结构风格的选择

- 简单地判断某一个具体的应用应该采取何种体系结构是非常困难的，需要借助于丰富的经验。
- 绝大多数实际运行的系统都是几种体系结构的复合：
  - 在系统的某些部分采用一种体系结构而在其他部分采用另外的体系，故而需要将复合几种基本的体系结构组合起来形成复合体系结构。
  - 在实际的系统分析和设计中，首先将整个系统作为一个功能体进行分析和权衡，得到适宜的和最上层的体系结构；如果该体系结构中的元素较为复杂，可以继续进行分解，得到某一部分的局部体系结构。
- 将焦点集中在系统总体结构的考虑上，避免较多地考虑使用的语言、具体的技术等实现细节上。
- 技术因素
  - 使用何种构件、连接件
  - 在运行时，构件之间的控制机制是如何被共享、分配和转移
  - 数据如何通讯
  - 数据与控制如何交互
- 质量因素
  - 可修改性：算法的变化；数据表示方式的变化；系统功能的可扩展性

    - 性能：时空复杂性
    - 可复用性

基于经验的选择原则

- 层次化的思想在任何系统中都可能得到应用
- 如果问题可分解为连续的几个阶段，那么考虑使用顺序批处理风格或管道过滤器风格
- 如果核心问题是应用程序中数据的理解、管理与表示，那么考虑使用仓库或者抽象数据类型（ADT）/OO风格
- 如果数据格式的表示可能发生变化，ADT/OO可限制这种变化所影响的范围
- 如果数据是持久存在的，则使用仓库结构
- 如果任务之间的控制流可预先设定、无须配置，那么考虑使用主程序子过程风格、OO风格
- 如果任务需要高度的灵活性与可配置性、松散耦合性或者任务是被动性的，那么考虑使用事件系统或C/S风格
- 如果设计了某种计算，但没有机器可以支持它运行，那么考虑使用虚拟机/解释器体系结构
- 如果要实现一些经常发生变化的业务逻辑，考虑使用基于规则的系统