第3章 软件体系结构风格

内容

- **3.1 概述**
- **3.2 数据流风格**
- **3.3 过程调用风格**
- **3.4 独立构件风格**
- 3.5 层次风格
- 3.6 虚拟机风格
- 3.7 客户/服务器风格
- 3.8 表示分离风格
- 3.9 插件风格
- 3.10 微内核风格
- 3.11 SOA风格

两类典型的独立构件风格子风格

- **3.4.1 进程通信体系结构风格**
- **3.4.2 基于事件的隐式调用风格**

3.4.1 进程通信体系结构风格

进程通信体系结构风格

- 构件是独立的进程，连接件是消息传递。消息传递通常用来实现进程之间的同步和对共享资源的互斥操作。
- 典型例子：客户-服务器架构，其中服务器通常用来为一个或多个客户端提供数据服务，客户端则用来向服务器发出请求，针对这些请求服务器通过同步或异步方式进行请求响应。

常见进程间通信方式都有哪些？
请举出至少三种说明其优缺点，并选择其中一种加以实现。

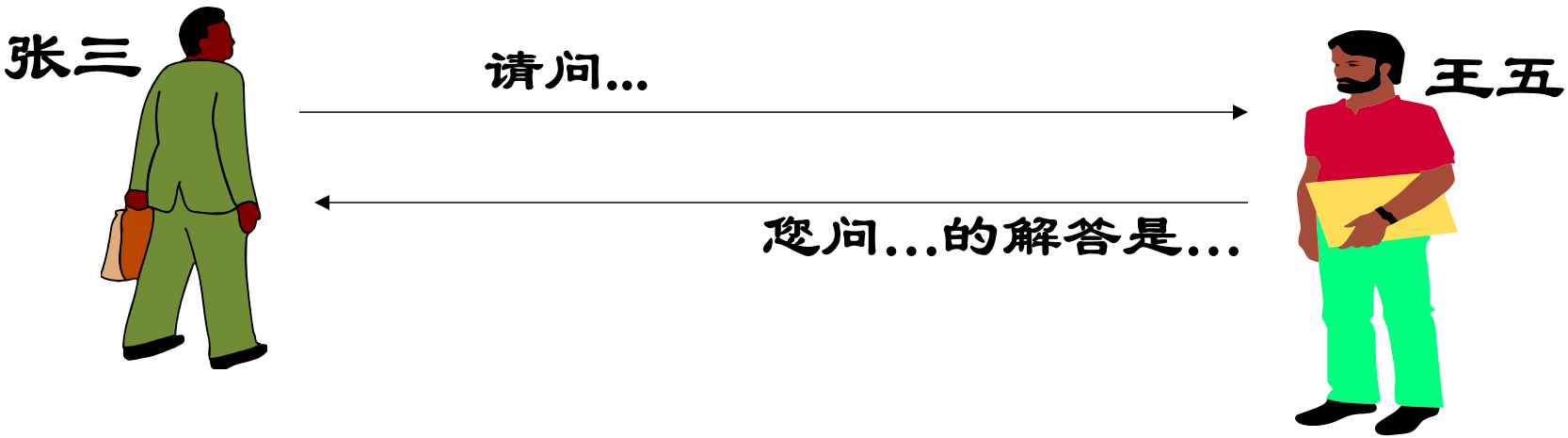
参考资料:
https://en.wikipedia.org/wiki/Inter-process_communication

Method	Short Description	Provided by (operating systemsor other environments)
File	A record stored on disk, or a record synthesized on demand by a file server, which can be accessed by multiple processes.	Most operating systems
Signal; also Asynchronous System Trap	A system message sent from one process to another, not usually used to transfer data but instead used to remotely command the partnered process.	Most operating systems
Socket	A data stream sent over a network interface, either to a different process on the same computer or to another computer on the network. Typically byte-oriented, sockets rarely preserve message boundaries. Data written through a socket requires formatting to preserve message boundaries.	Most operating systems
Unix domain socket	Similar to an internet socket but all communication occurs within the kernel. Domain sockets use the file system as their address space. Processes reference a domain socket as an inode, and multiple processes can communicate with one socket	All POSIX operating systems
Message queue	A data stream similar to a socket, but which usually preserves message boundaries. Typically implemented by the operating system, they allow multiple processes to read and write to the message queue without being directly connected to each other.	Most operating systems
Pipe	A unidirectional data channel. Data written to the write end of the pipe is buffered by the operating system until it is read from the read end of the pipe. Two-way data streams between processes can be achieved by creating two pipes utilizing standard input and output.	All POSIX systems, Windows
Named pipe	A pipe implemented through a file on the file system instead of standard input and output. Multiple processes can read and write to the file as a buffer for IPC data.	All POSIX systems, Windows, AmigaOS 2.0+
Shared memory	Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other.	All POSIX systems, Windows
Message passing	Allows multiple programs to communicate using message queues and/or non-OS managed channels, commonly used in concurrency models.	Used in RPC, RMI, and MPIparadigms, Java RMI, CORBA, DDS, MSMQ, MailSlots, QNX, others
Memory-mapped file	A file mapped to RAM and can be modified by changing memory addresses directly instead of outputting to a stream. This shares the same benefits as a standard file.	All POSIX systems, Windows

3.4.2 显式调用 vs. 隐式调用
Explicit vs. Implicit Invocation

消 息

对象之间传递的内容，指示、打听、请求 … …

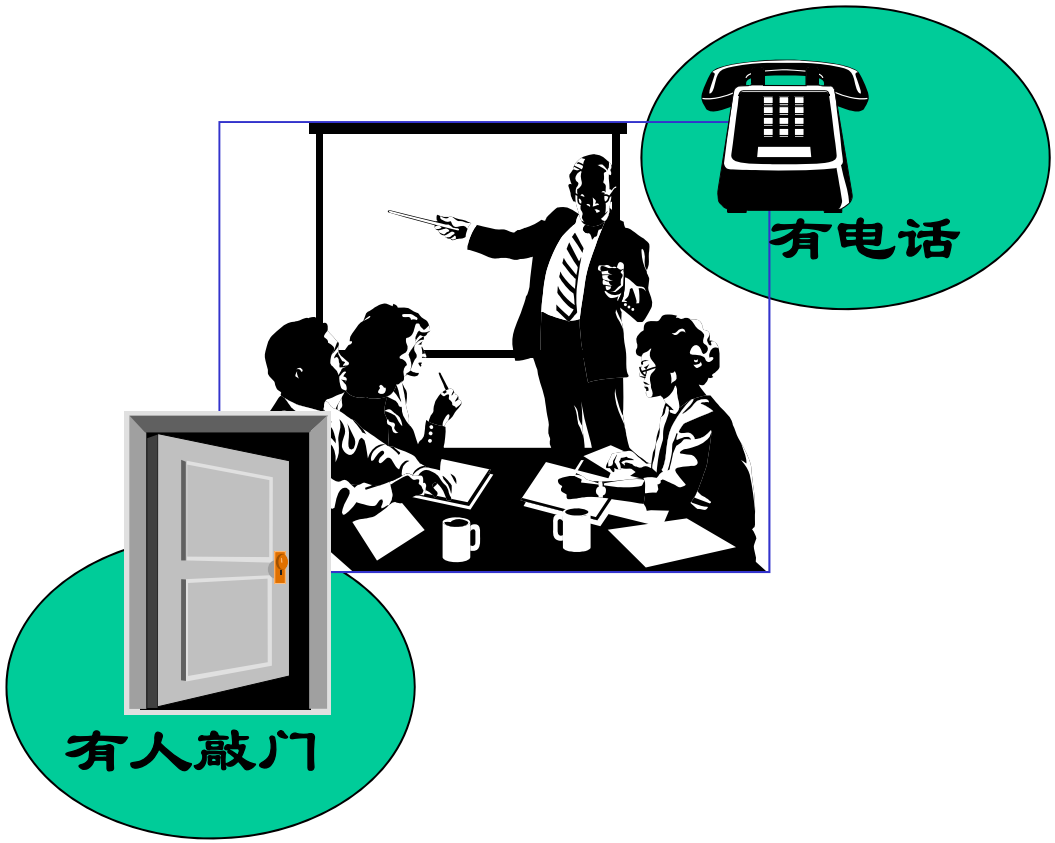


计算机中，消息是**具有特定含义的数据**

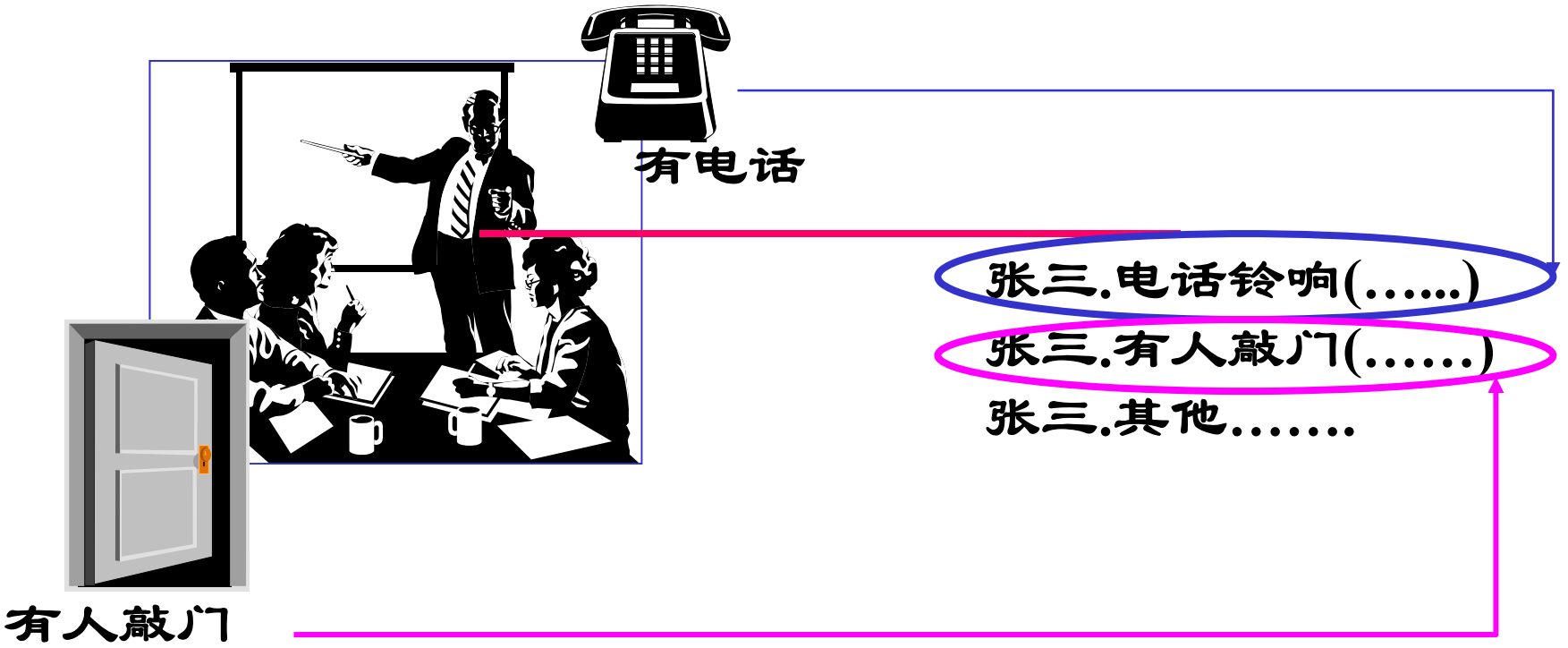
事 件

能够激活对象功能的**动作**。当发生这种动作后将给所涉及对象发送一个**消息**，对象便可执行相应的**功能**

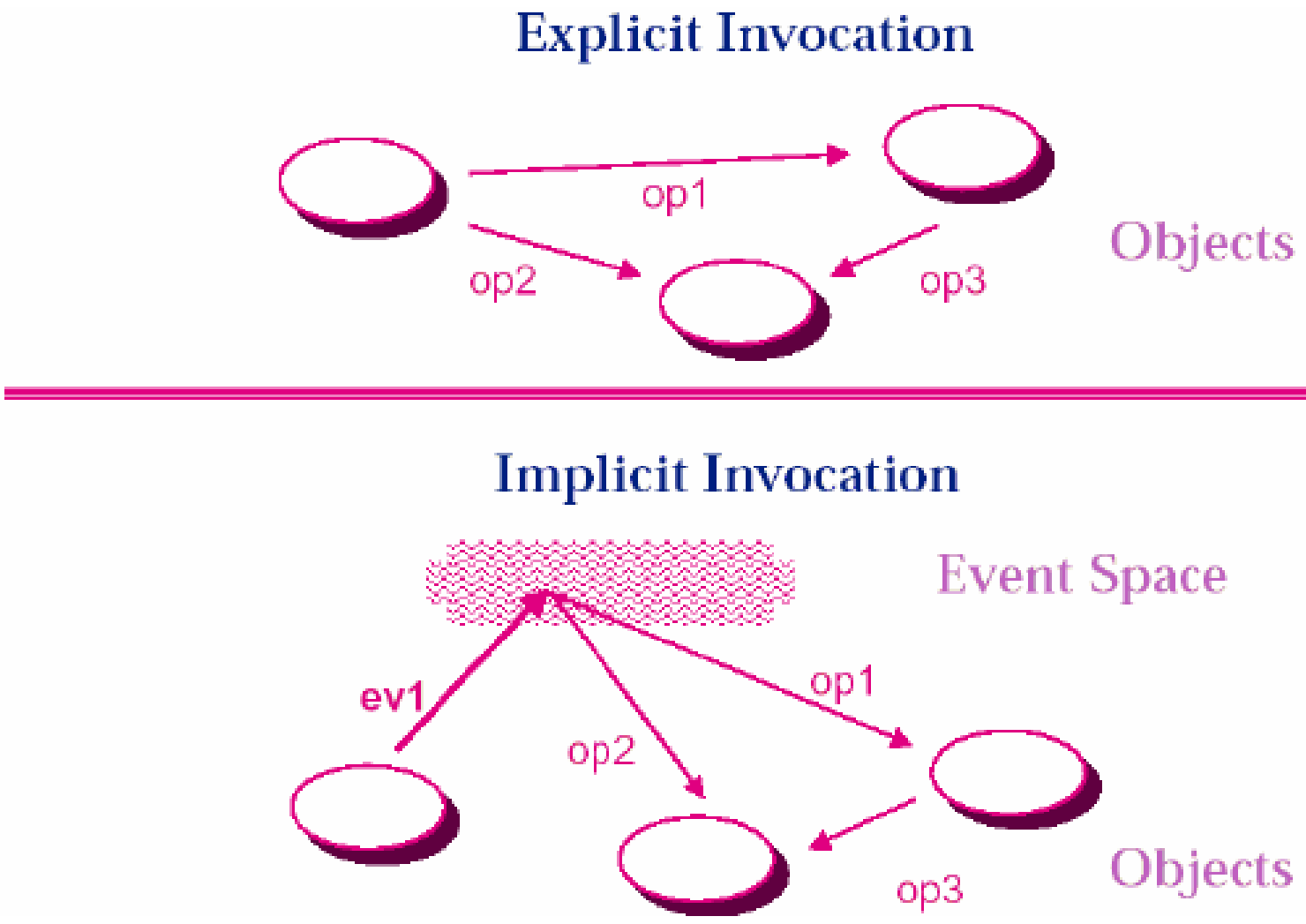
动作—>消息– >功能



对某一对象发生什么事件，该对象便找到相应的处理该事件的程序去处理这一事件



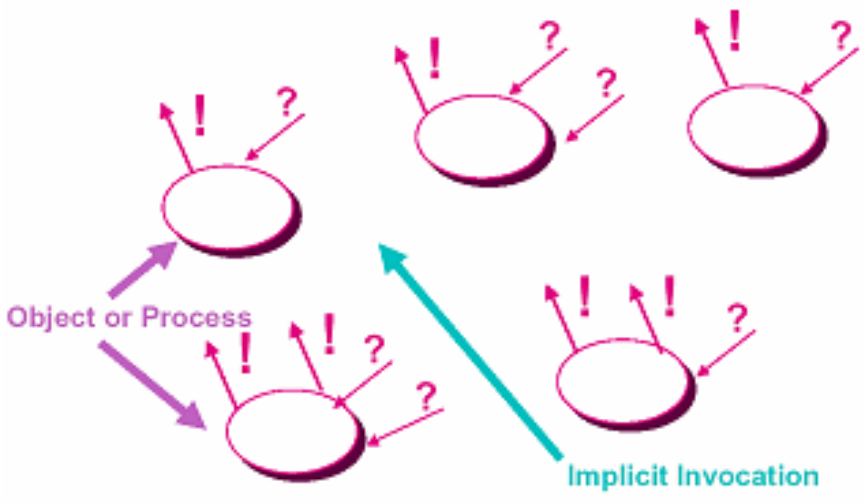
显式调用 vs. 隐式调用
Explicit vs. Implicit Invocation



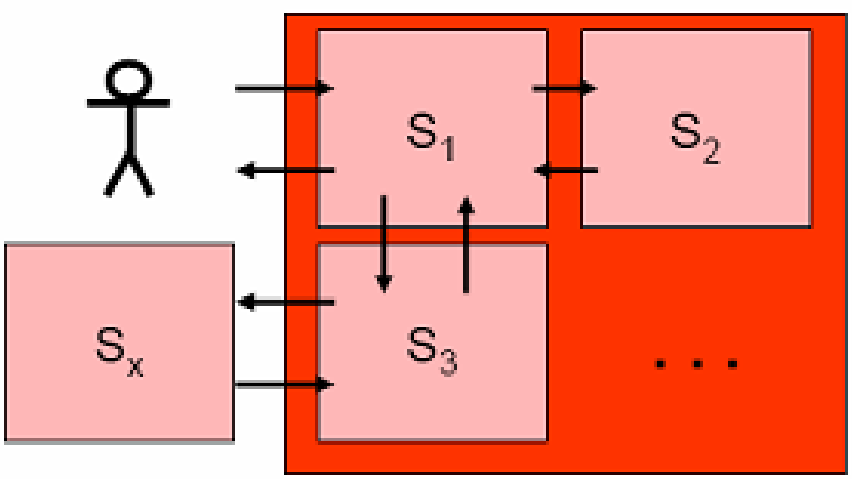
Explicit Invocation (显式调用)

- Traditionally, in a system in which the component provide a collection of routines and functions, such a an object-oriented system, components typically interact with each other by explicitly invoking those routines.
- 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。

Implicit Invocation（隐式调用）：Event Systems



Today's model of computation:
Software are interaction entities (reactive systems) that continuously interact with their environment and that evolve with their environment.



…在很多情况下，软件更多的变成被动性系统，构件持续的与其所处的环境打交道，但并不知道确切的交互次序

基于事件的隐式调用（Cont.）
Event-based, Implicit Invocation

- 构件不直接调用一个过程，而是触发或广播一个或多个事件。
- 系统中的其它构件中的过程在一个或多个事件中注册。
- 当一个事件被触发/发布，系统自动调用在这个事件中注册的所有过程。
- 这样，一个事件的触发就导致了另一模块中的过程的调用。

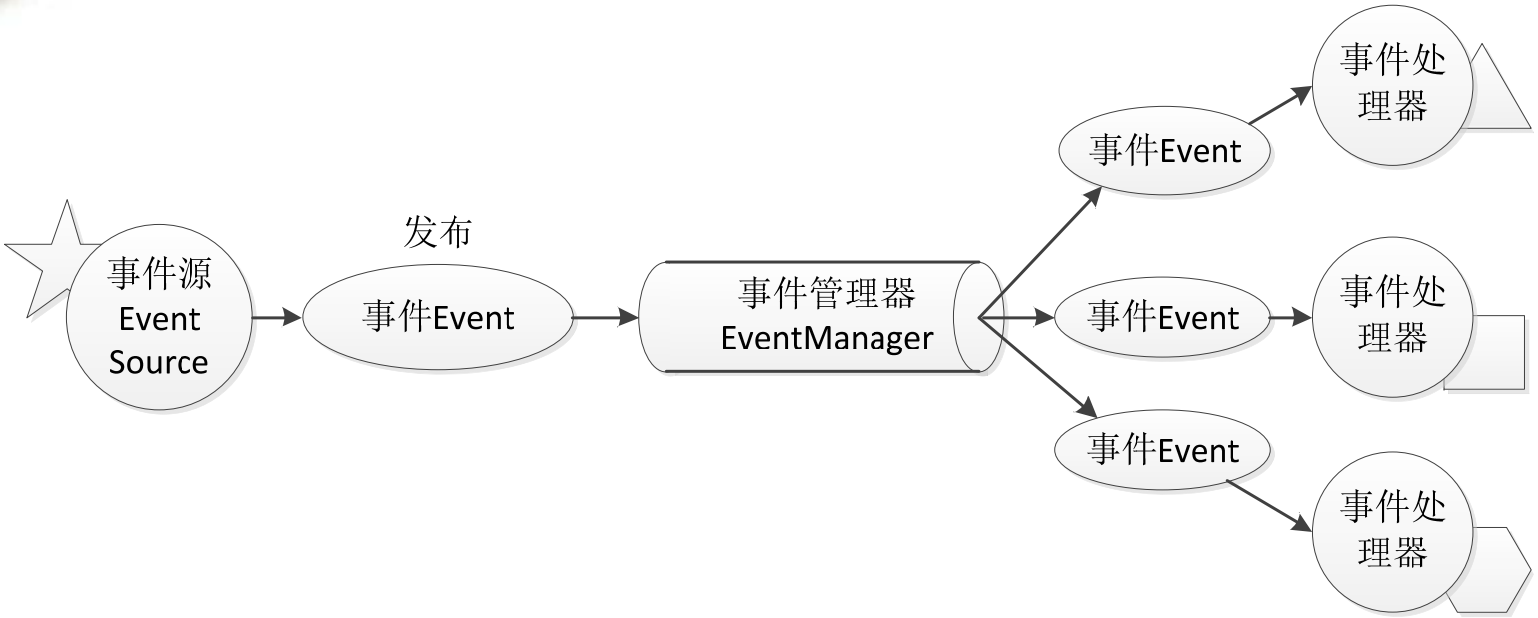
- 这种系统，称为基于事件的系统**(Event-based system)**，采用隐式调用**(Implicit invocation)**的方式。

为何称为“独立构件”风格？
Why Independent Components Style?

- 这种风格的主要特点是：事件的触发者并不知道哪些构件会被这些事件影响，相互保持独立。
- 这样不能假定构件的处理顺序，甚至不知道哪些过程会被调用；
- 各个构件之间彼此无直接的连接关系，各自独立存在，通过对事件的发布和注册实现关联。

3.4.3 事件系统的基本结构与工作原理

事件系统构成原理图



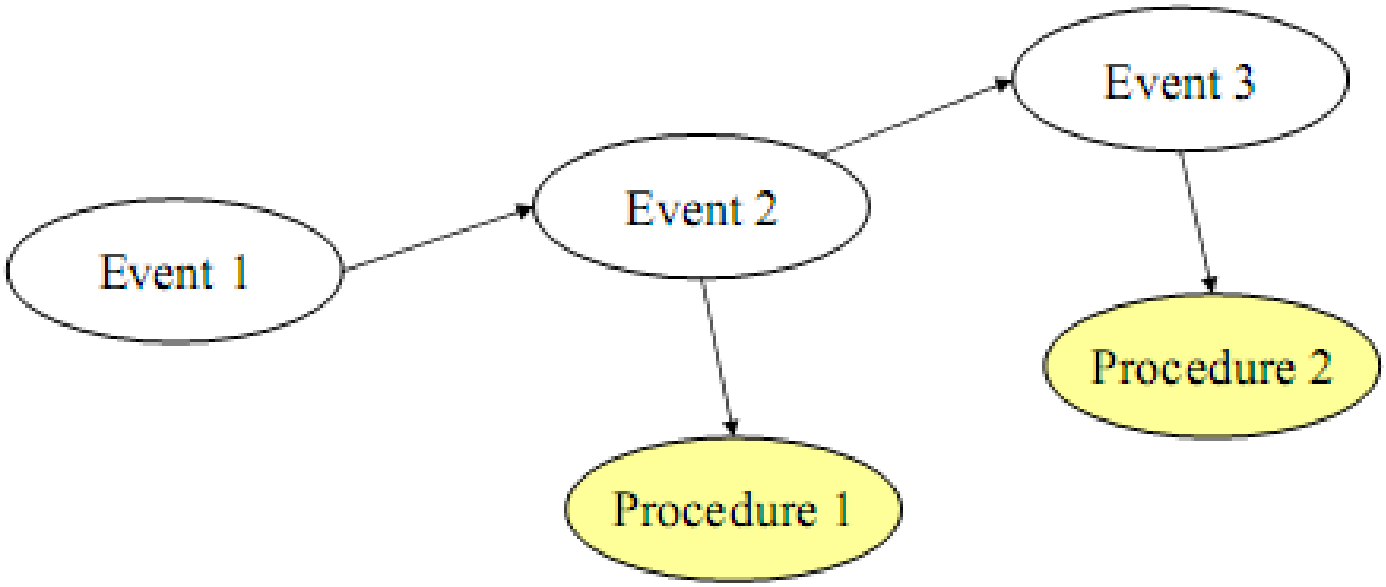
功能	描述
分离的交互	事件发布者并不会意识到订阅者的存在。
多对多通信	采用发布/订阅消息传递，一个特定事件可以影响多个订阅者。
基于事件的触发器	控制流由接收者确定（基于发布的事件）。
异步	通过事件消息传递支持异步操作。

事件系统的连接机制（1）

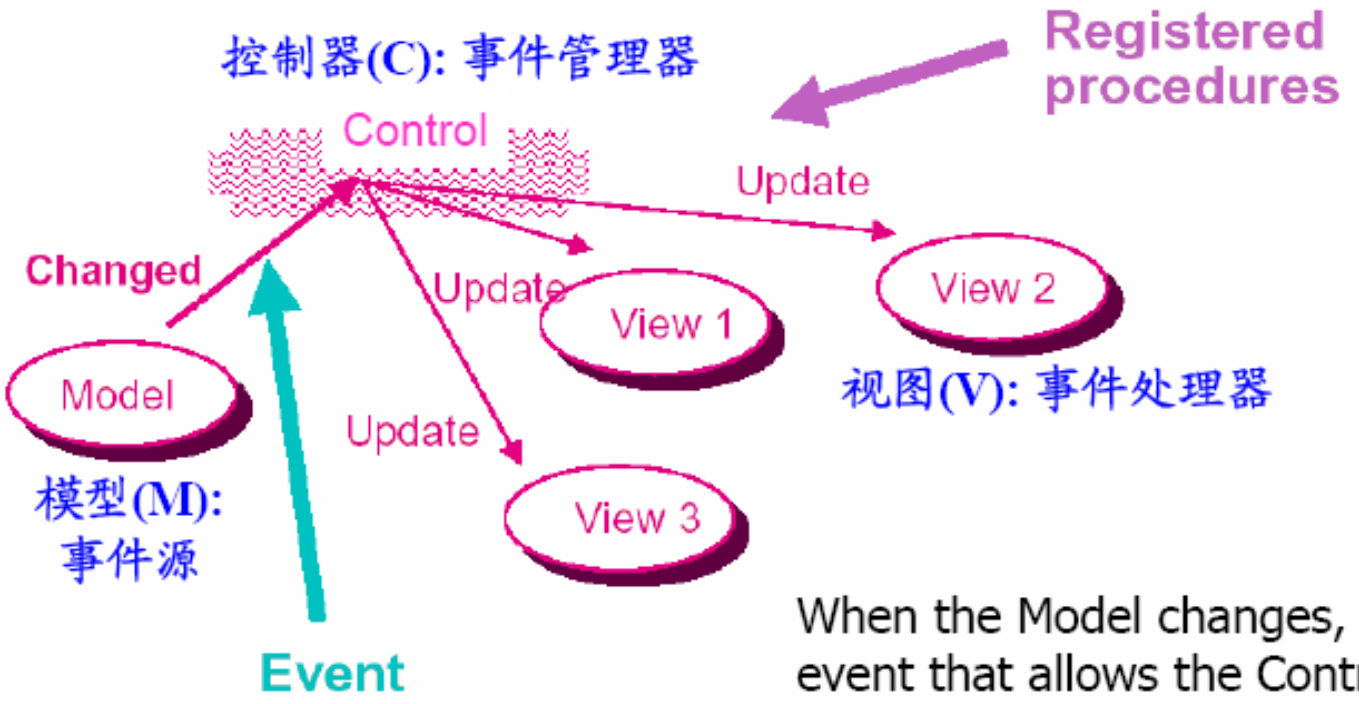
- Connectors: event-procedure bindings (连接器：事件-过程绑定)
  - Procedures are registered with events (过程<事件处理器，事件的接收和处理方>向特定的事件进行注册);
  - Components communicate by announcing events at "appropriate" times (构件<事件源>发布事件);
  - when an event is announced the associated procedures are (implicitly) invoked (当某些事件被发布时，向其注册的过程被隐式调用);
  - Order of invocation is non-deterministic (调用的次序是不确定的);

事件系统的连接机制（2）

- In some treatments, connectors are event-event bindings (在某些情况下，一个事件也可能触发其他事件，形成事件链).
- 这种连接机制称为“事件-事件绑定”(Event-event binding)。



例子：Model-View-Control (MVC)



When the Model changes, it announces an event that allows the Controller to automatically invoke the procedures to update each View related to the Model. (当模型数据发生改变时，它将发布事件，控制器自动调用相关View的“更新”过程)

事件调度策略

- When an event is announced, the system itself automatically invokes all of the procedures that have been registered for that event. (当事件发生时，已向此事件注册过的过程被激发并执行)
- How to make events dispatched to registered components in the system? (问题：事件管理器如何向这些过程分发相关的事件？)

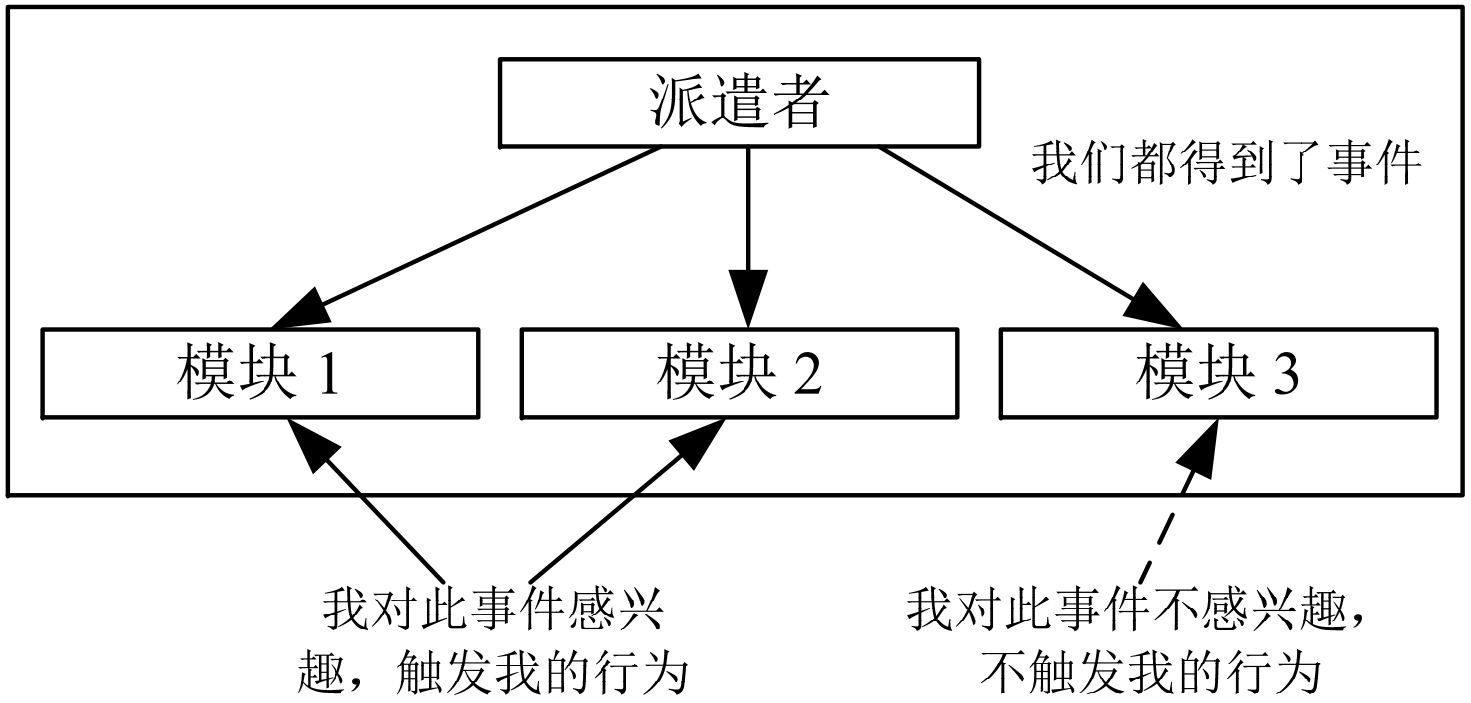
- 答案：两种策略
  - EventManager with separated dispatcher module (带有独立派遣模块的事件管理器)
  - EventManager without a central dispatcher module (没有独立派遣模块的事件管理器)

（1）事件派遣模块
Event dispatcher module

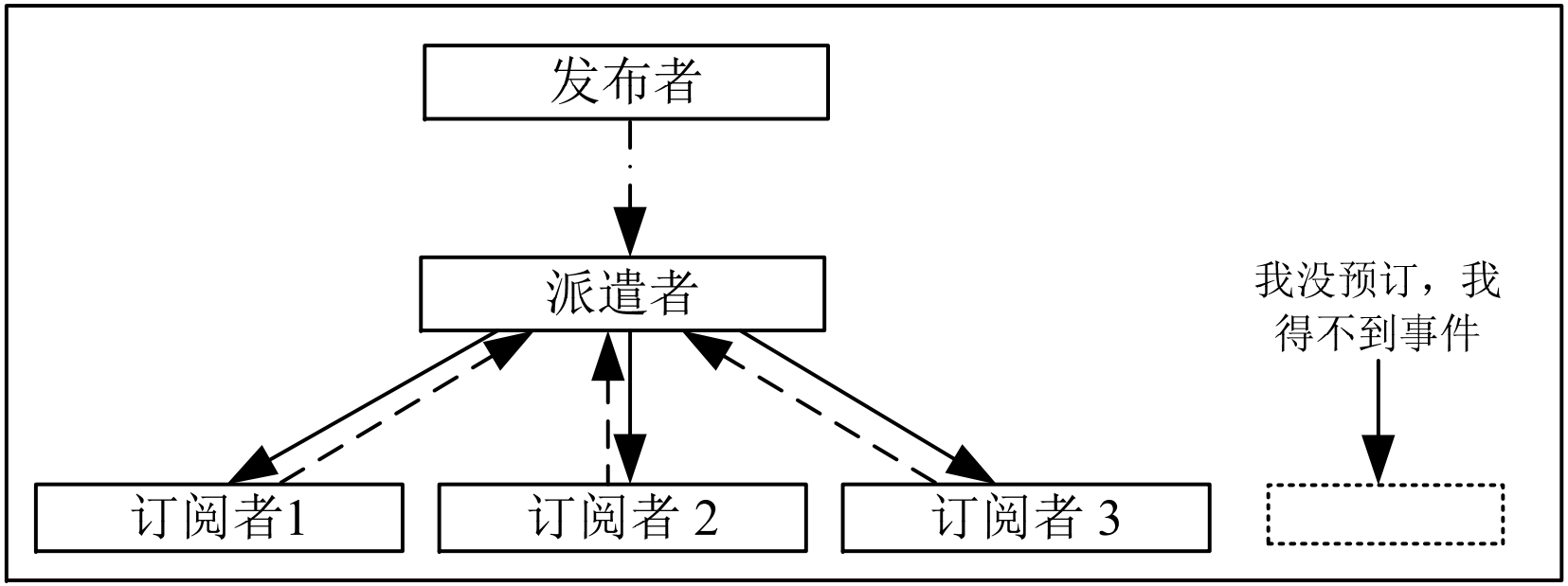
- 事件派遣模块(Event dispatcher module)的功能：
  - The dispatcher module is responsible for receiving all incoming events and dispatching them to other modules in the system. (负责接收到来的事件并派遣它们到其它模块)
  - The dispatcher should decide how events are sent to other modules with two different strategies (派遣器应该决定怎样派遣：又存在两种策略)
    - 广播式(All broadcasting)：派遣模块将事件广播到所有的模块，但只有感兴趣的模块才去取事件并触发自身的行为；
    - 选择广播式(Selected broadcasting)：派遣模块将事件送到那些已经注册了的模块中。

广播式（All broadcasting）

无目的广播，靠接受者自行决定是否加以处理或者简单抛弃



选择广播式(Selected broadcasting)



有目的广播，
只发送给那些已经注册过的订阅者

发布事件
订阅
发送事件

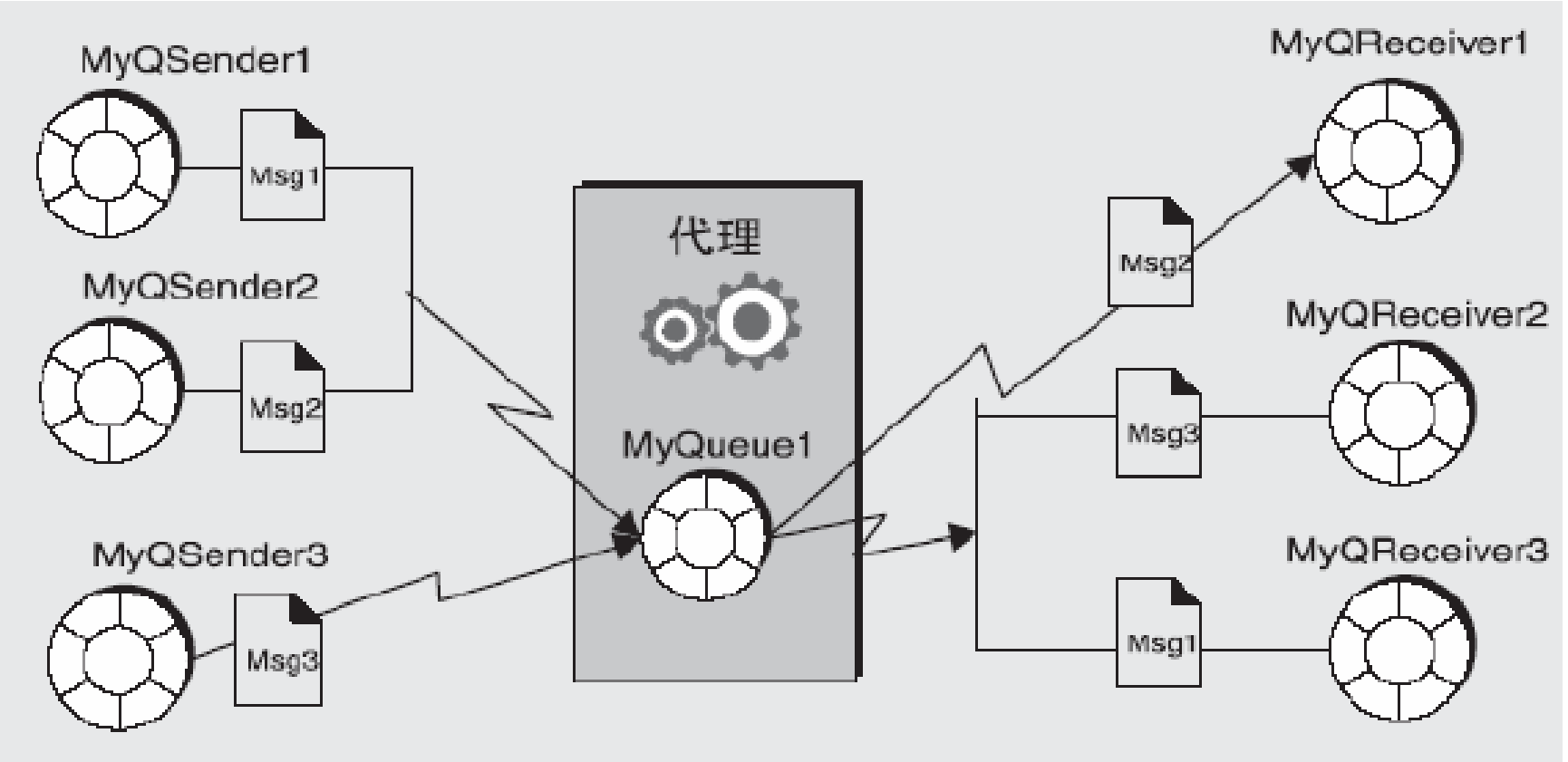
选择广播式的两种策略

- Point-to-Point (message queue) (点对点模式：基于消息队列)
- Publish-Subscribe (发布-订阅模式)

点对点的选择广播式

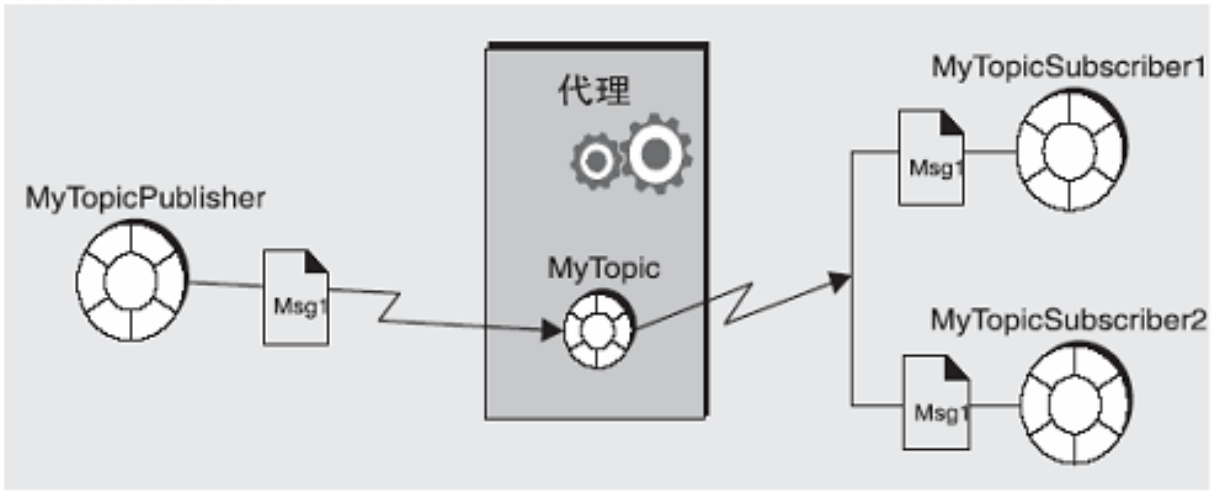
- System installs and configures a queue manager and defines a named message queue. (系统安装并配置一个队列管理器，并定义一个命名的消息队列)
- An application then registers a software routine that “listens” for messages placed onto the queue. (某个应用向消息队列注册，以监听并处理被放置在队列里的事件)
- Second and subsequent applications may connect to the queue and transfer a message onto it. (其他的应用连接到该队列并向其中发布事件)
- The queue manager stores the messages until a receiving application connects and then calls the registered software routine. (队列管理器存储这些消息，直到接收端的应用连接到队列，取回这些消息并加以处理)
- Message can be consumed by only one client. (消息只能够被唯一的消费者所消费，消费之后即从队列中删除)

（更复杂的）点对点的选择广播式



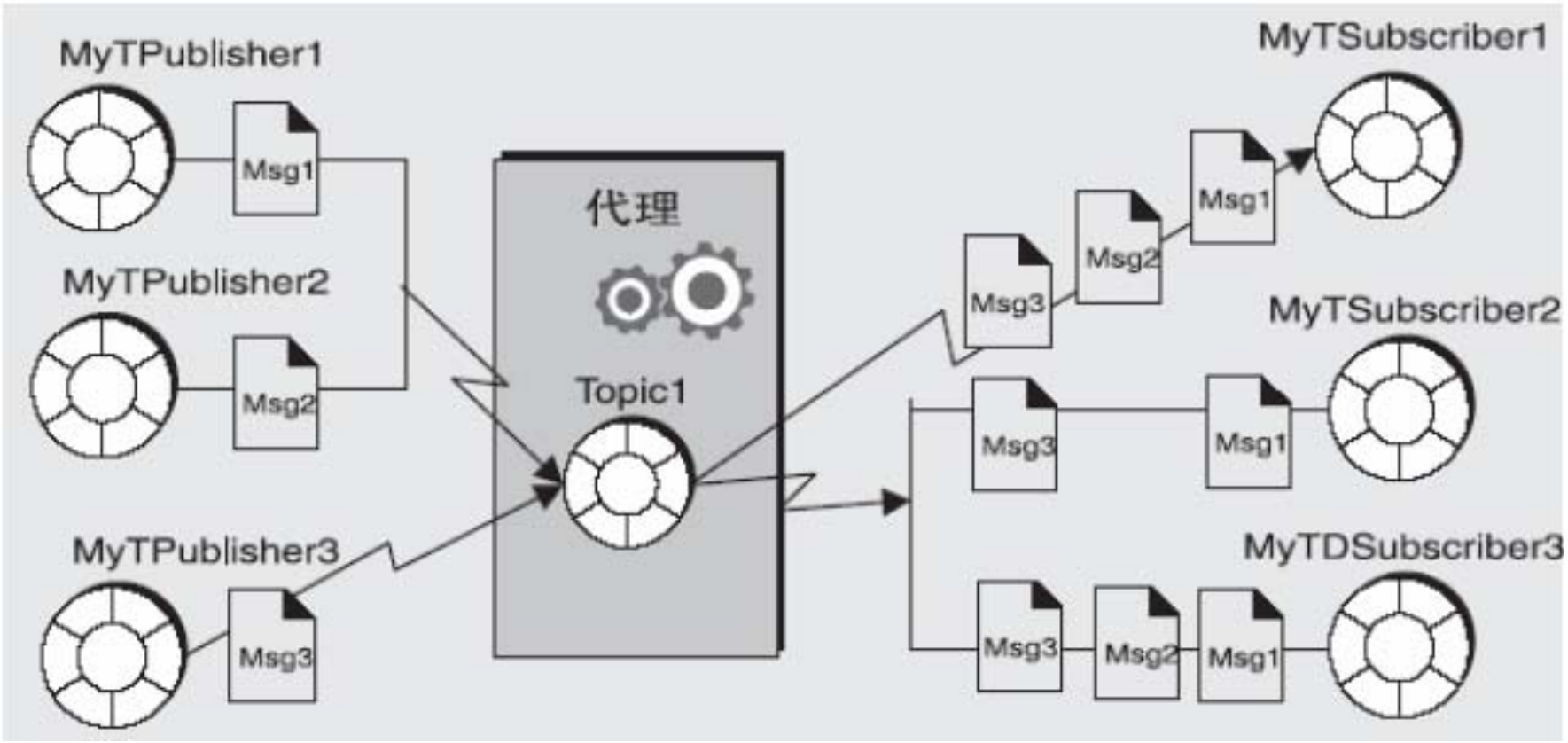
发布-订阅的选择广播式

- Publishers post messages to an intermediary Broker and subscribers register subscriptions with that broker. (事件发布者向“主题”发布事件，订阅者向“主题”订阅事件)
  - 一个事件可以被多个订阅者消费；
  - 事件在发送给订阅者之后，并不会马上从topic中删除，topic会在事件过期之后自动将其删除。

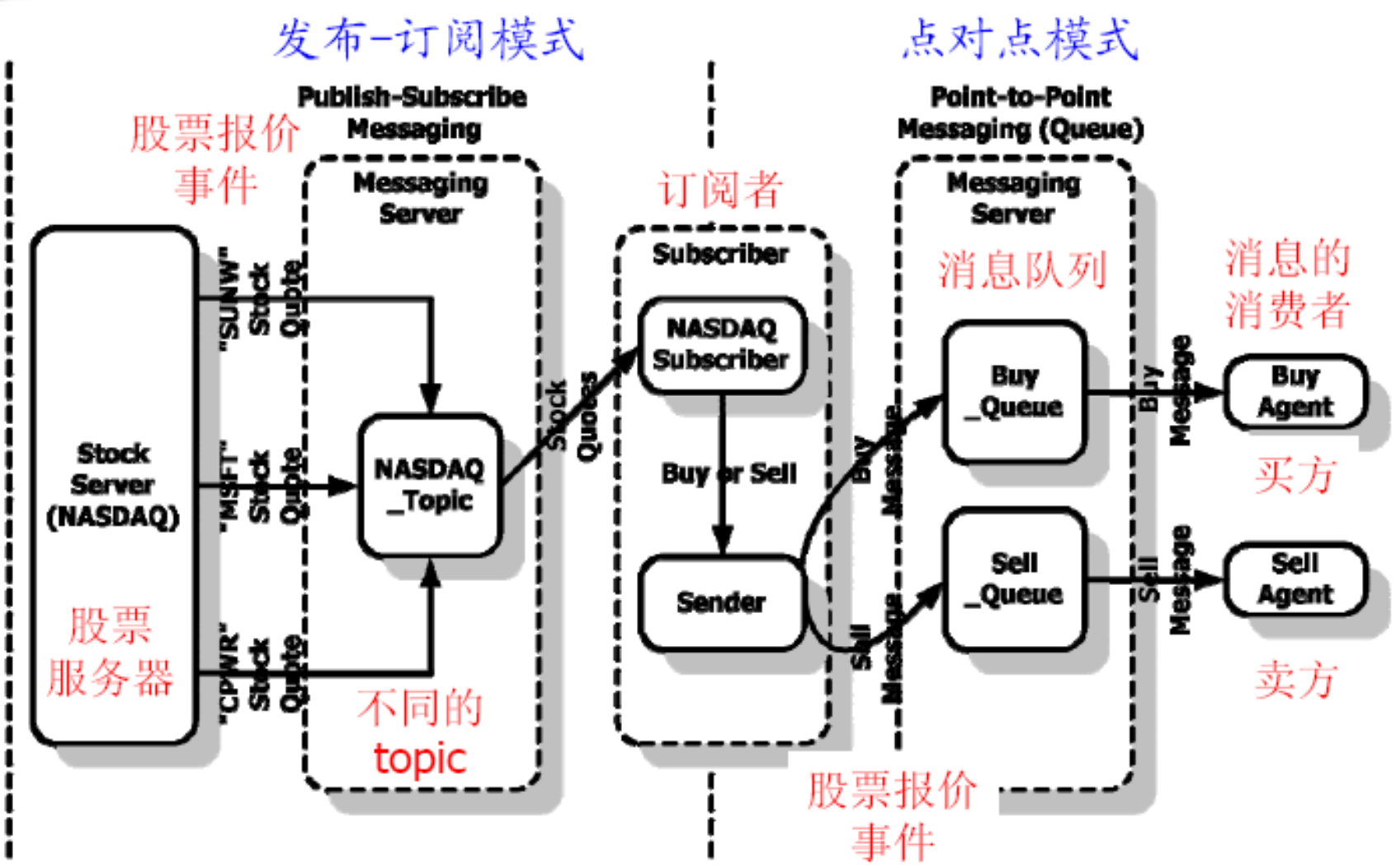


生活中的例子：新闻订阅/RSS　想想微信订阅号

(更复杂的)发布-订阅的选择广播式



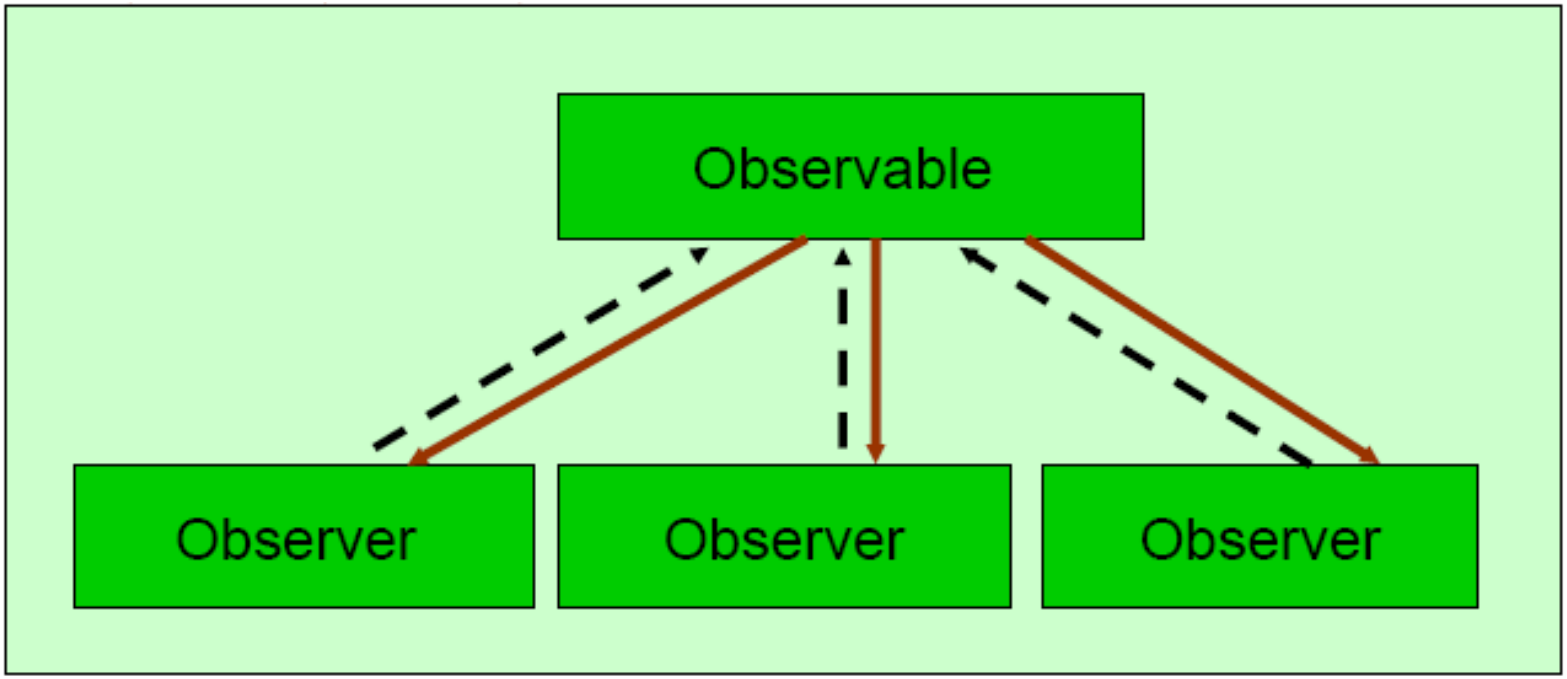
一个集成的例子：股票交易市场



无独立调度模块的事件系统

- This module is usually called Observable/Observer (称为“被观察者/观察者”).
- Each module allows other modules to declare interest in events that they are sending. (每一个模块都允许其他模块向自己所能发送的某些消息表明兴趣)
- Whenever a module sends an event it sends that event to exactly those modules that registered interest in that event. (当某一模块发出某一事件时，它自动将这些事件发布给那些曾经向自己注册过此事件的模块)

无独立调度模块的事件系统

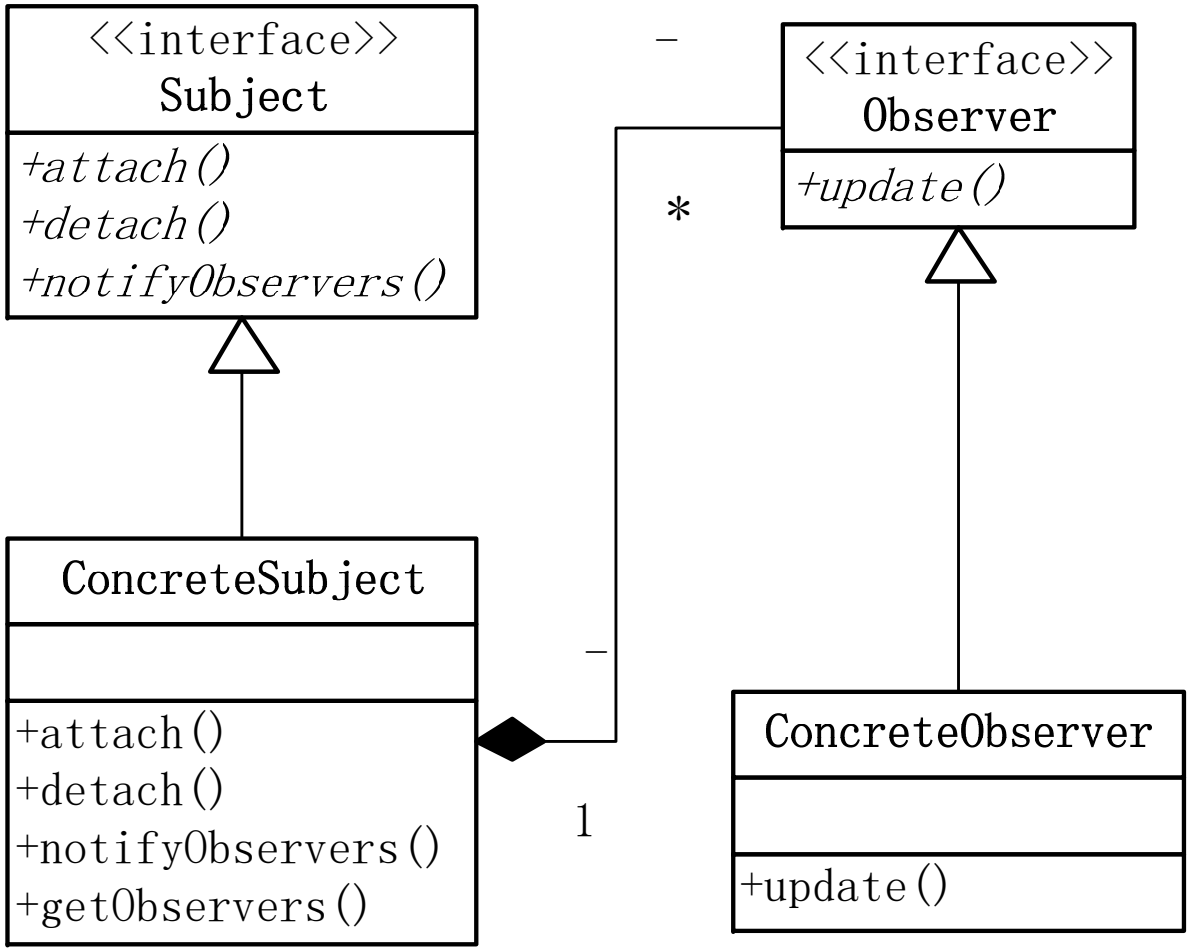


Observable/Observer module

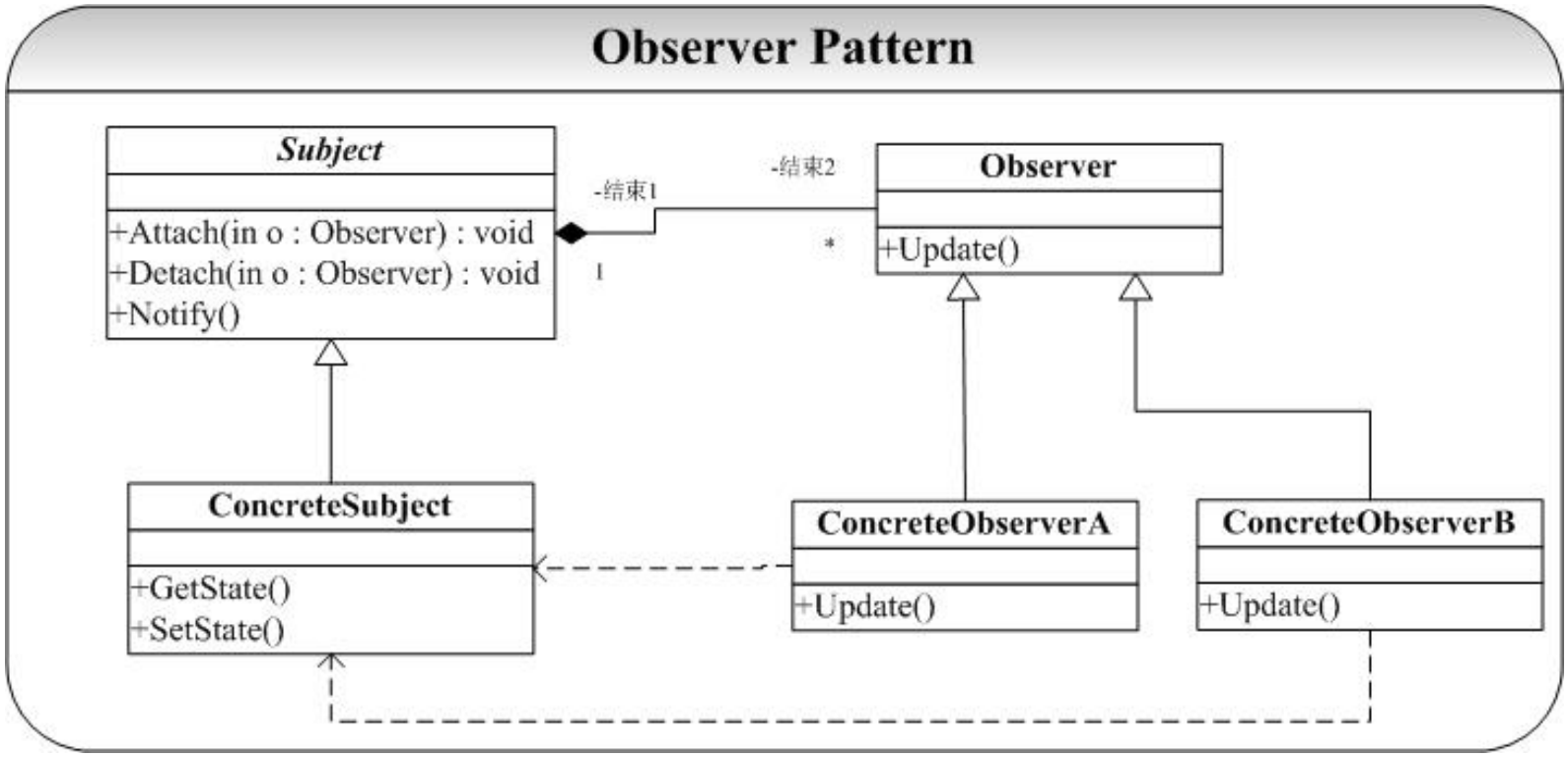
Legend: ••••→ Register event ——→ Send event

无独立调度模块的事件系统

通过“观察者”模式（Observer Pattern）来进行实现



C++实现观察者模式



- GoF 23种设计模式解析附C++实现源码(2nd Edition)

Java实现观察者模式 (1)

```
public class TestObserver
{
        public static void main(String[] args)
        {
                ConcreteSubject sbj= new ConcreteSubject();
                Observer  co1=new ConcreteObserver1();
                Observer  co2=new ConcreteObserver2();
                sbj.attach(co1);
                sbj.attach(co2);
                sbj.notifyObservers();
        }
}
```

Java实现观察者模式 (2)

```
public interface Subject
{
        public void attach(Observer observer);
        public void detach(Observer observer);
        void notifyObservers();
}
```

Java实现观察者模式 (3)

```
public class ConcreteSubject implements Subject
{
        private Vector observersVector = new
                java.util.Vector();
        public void attach(Observer observer)
        {    observersVector.addElement(observer);    }
        public void detach(Observer observer)
        {    observersVector.removeElement(observer);      }
```

Java实现观察者模式 (4)

```
    public void notifyObservers()
    {
        Enumeration enumeration = observers();
        while (enumeration.hasMoreElements())
        { ((Observer)enumeration.nextElement()).update(); }
    }
    public Enumeration observers()
    {
            return ((Vector)
            observersVector.clone()).elements();
    }
}
```

Java实现观察者模式 (5)

```
public interface Observer
{
        void update();
}
```

Java实现观察者模式(6)

```
public class ConcreteObserver1 implements Observer
{
        public void update()
        {
                System.out.println("I am observer 1. I got you”+
                        “message and I am going to”+
                        “do something ");
        }
}
public class ConcreteObserver2 implements Observer
{
        public void update()
        {
                System.out.println("I am observer 2. I got you”+
                        “message and I am going to”+
                        “do something ");
        }
}
```

3.4.4 事件系统的实现机制

事件系统的实现机制（案例）分析
Implementation Mechanisms in Event Systems

- Windows事件机制分析
- VC++ 事件处理案例分析
- Java事件处理案例分析
- JMS (Java Message Service)中的事件处理

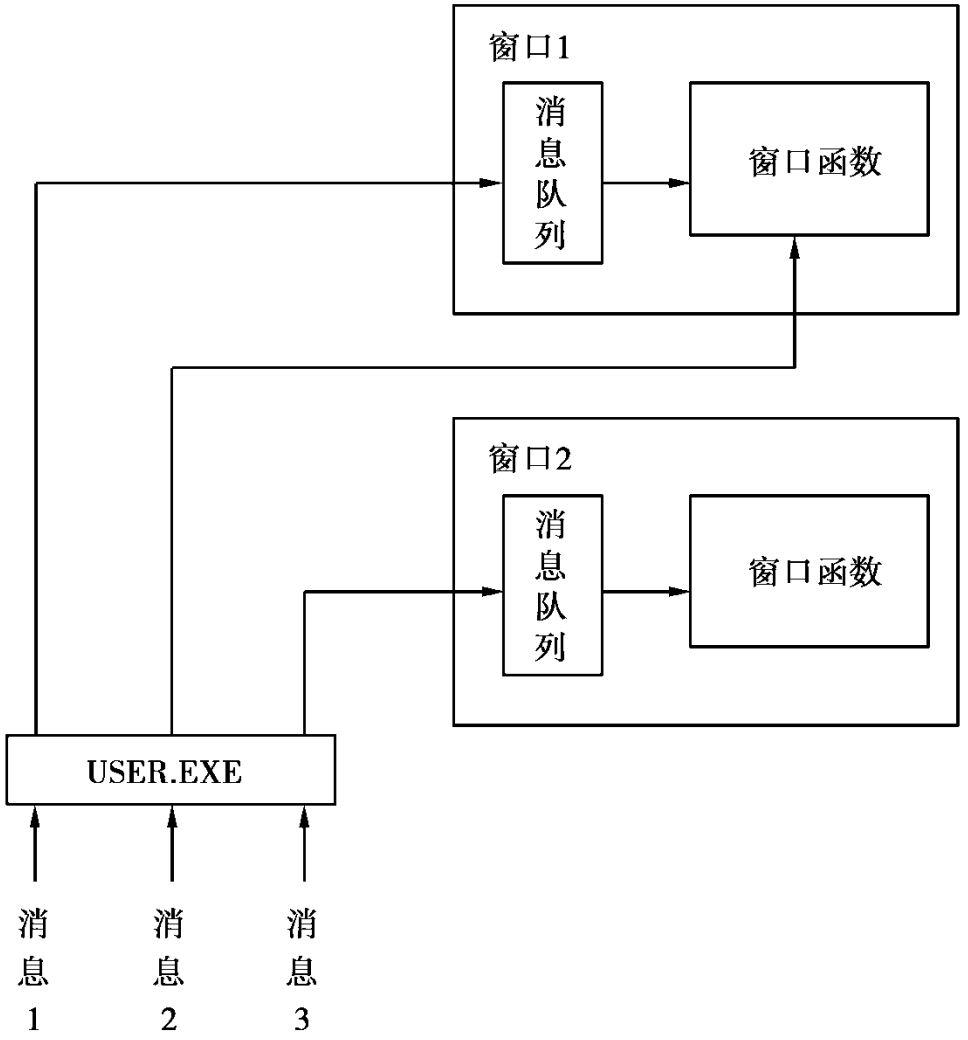
Windows GUI程序—事件及消息机制

- Windows是事件驱动（消息驱动）的OS，也是基于消息的OS。
- Windows 应用程序中消息有两种送出途径；直接和排队。
  - Windows或某些运行的应用程序可直接发布消息给窗口过程。
  - 消息也可送到消息队列，在应用程序执行期间应用程序对象连续不断轮询消息队列的消息。凡是以排队方式送出的消息都被送到一个由操作系统提供的消息队列的保留区。在OS中当前执行的每个进程都有各自的消息队列。
- 事件驱动程序不是由事件的顺序来控制，而是由事件的发生来控制，而事件的发生是随机的、不确定的，这就允许程序的用户用各种合理的顺序来安排程序的流程。
- 事件驱动围绕消息的产生与处理展开，它是靠消息循环机制来实现的，消息是一种报告有关事件发生的通知。

Windows消息分类

- 消息循环
  - 事件驱动是靠消息循环机制来实现的。
  - 消息是一种报告有关事件发生的通知。

    Windows消息来源有以下4种：
    - ✓输入消息
    - ✓控制消息
    - ✓系统消息
    - ✓用户消息



Windows 消息来源

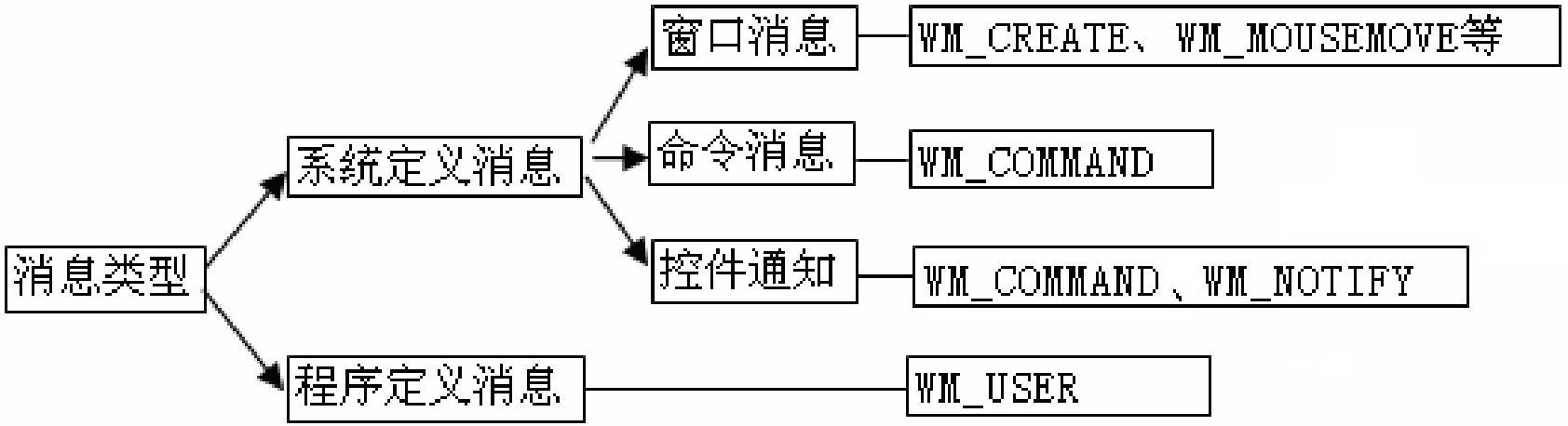
1）输入消息：包括键盘和鼠标的输入。这类消息首先放在OS消息队列中，然后由Windows将它们送到应用程序的消息队列中，由应用程序来处理消息。

2）控制信息：用来与Windows的控制对象（列表框、按钮等）进行双向通信。这类消息一般不经过应用程序消息队列，而是直接发送到控制对象上去。

3）系统消息：对程序化的事件或系统时钟中断作出反映。有些消息（动态数据交换消息DDE）要通过OS消息队列，而有的则不通过OS消息队列而直接送入应用程序的消息队列（如创建窗口消息）。

4）用户消息：这是程序员自己定义并在应用程序中主动发出的消息，一般由应用程序的某一部分内部处理。

Windows消息类型



来源：Windows消息机制（MFC）
http://www.cnblogs.com/findumars/p/3948427.html

Windows系统定义消息(1)

- 窗口消息(Windows Message)
  - 与窗口的内部运作有关，如创建窗口，绘制窗口，销毁窗口等。可以是一般的窗口，也可以是Dialog，控件等。
  - 如：
    - ✓**WM_CREATE**
    - ✓**WM_PAINT,**
    - ✓**WM_MOUSEMOVE,**
    - ✓**WM_CTLCOLOR,**
    - ✓**WM_HSCROLL**
    - ✓**...**

Windows系统定义消息(2)

- 命令消息(Command Message)
  - 与处理用户请求有关， 如单击菜单项、工具栏或控件时，就会产生命令消息。
  - **WM_COMMAND**：
    - ✓**LOWORD(wParam)**表示菜单项、工具栏按钮或控件的**ID**。
    - ✓如果是控件, **HIWORD(wParam)**表示控件消息类型（**notification code**）, 比如**BN_CLICKED, BN_DBLCLK**等，标志用户对控件的操作，双击，单击之类。

Windows系统定义消息（3）

- 控件通知(Notify Message)
  - 控件通知消息，是指这样一种消息，一个窗口内的子控件发生了一些事情，需要通知父窗口。
  - 当用户与控件窗口交互（如鼠标从工具栏按钮上掠过）时，那么控件通知消息就会从控件窗口发送到它的主窗口
  - 这是最灵活的消息格式， 其Message, wParam, lParam分别为：

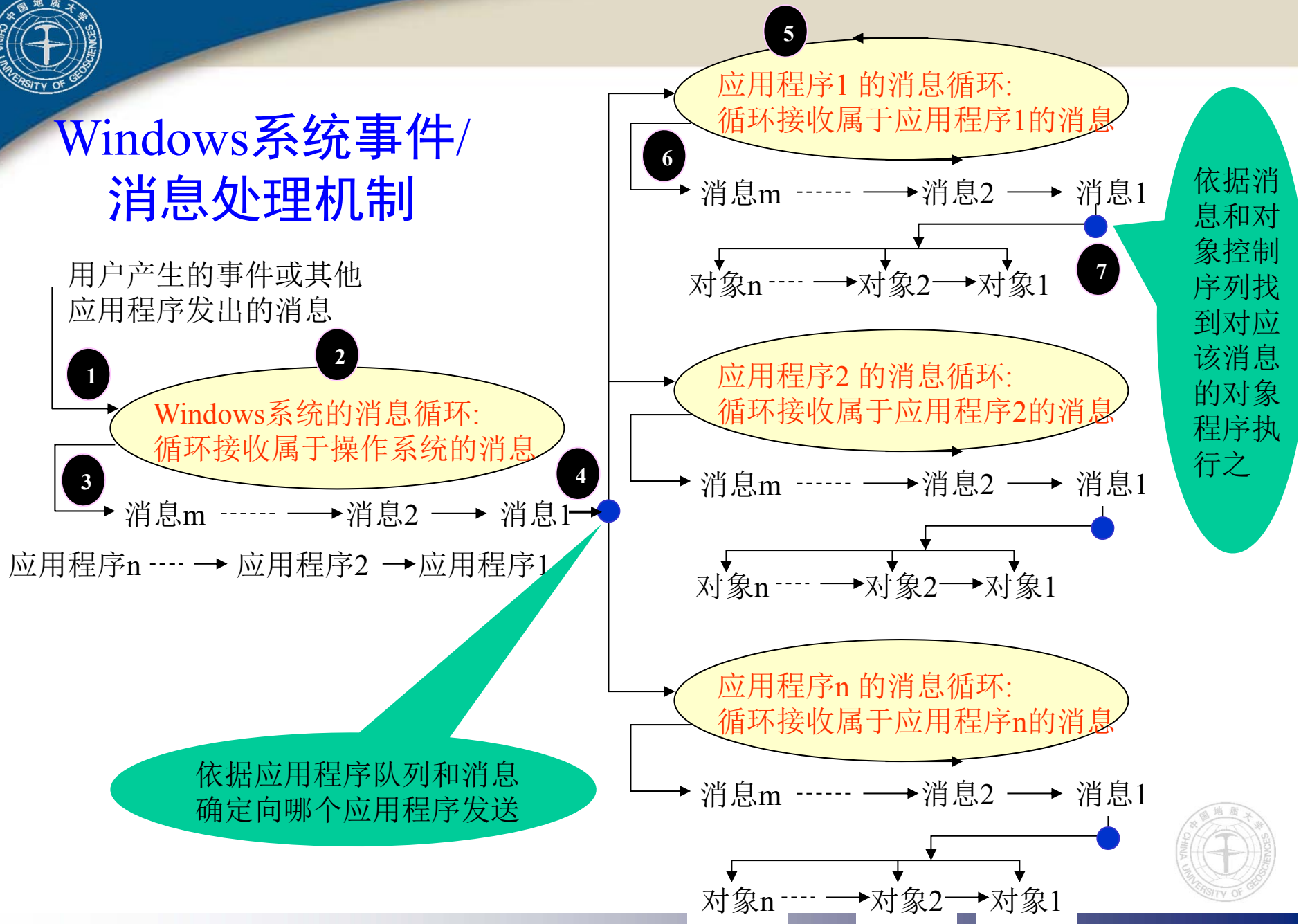
    (WM_NOTIFY, 控件ID，指向NMHDR的指针)。其中，NMHDR包含控件通知的内容（为一个结构类型），可以任意扩展。

Windows用户定义消息

- 用户自定义的消息，对于其范围有如下规定：
  - **WM_USER: 0x0400-0x7FFF (e.x. WM_USER+10)**
  - **WM_APP(winver>4.0): 0x8000-0xBFFF (ex.WM_APP+4)**
  - **RegisterWindowMessage: 0xC000-0xFFFF**

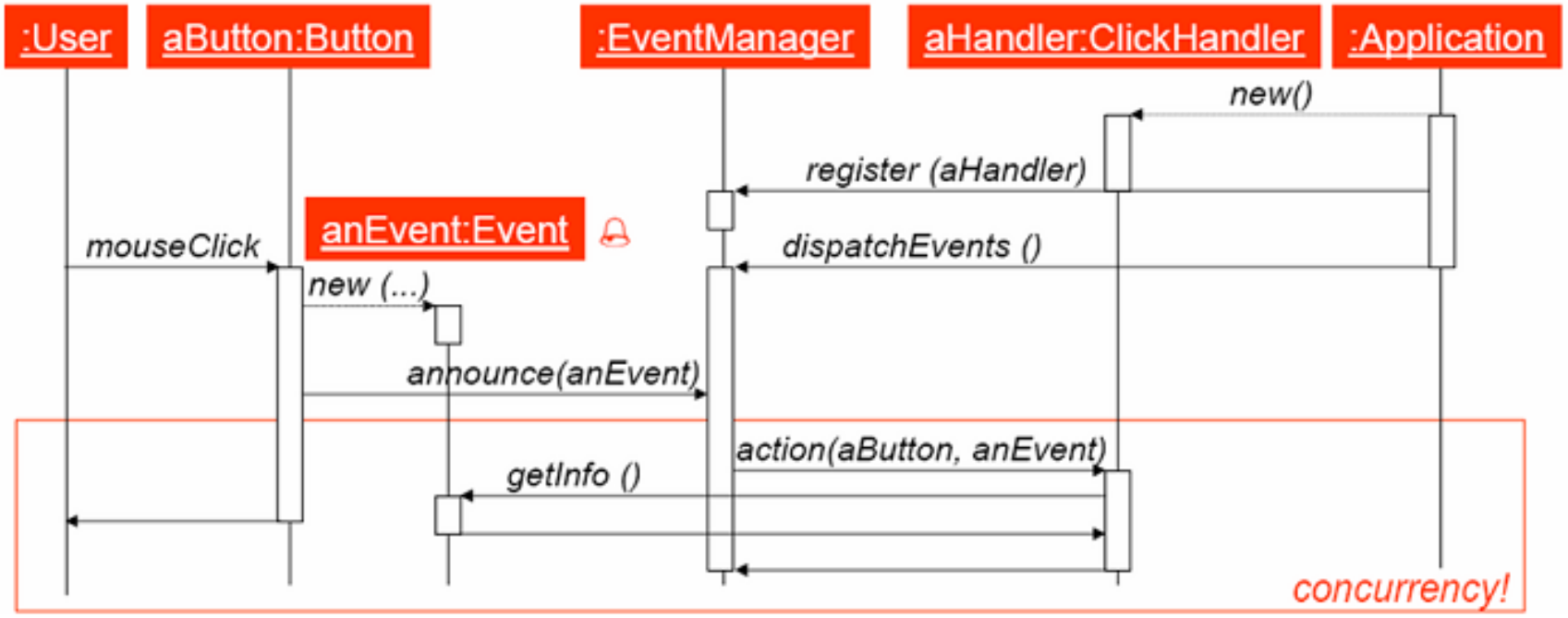
范围	表示
0 ~ WM_USER–1	操作系统保留的消息。
WM_USER ~ 0x7FFF	私有窗口类用的整数型消息。
WM_APP ~ 0xBFFF	应用程序用的消息。
0xC000 ~ 0xFFFF	应用程序用的字符串消息。
0xFFFF ~	操作系统保留的消息。

Windows系统事件/消息处理机制



Win32 GUI Event Processing

Event: "Button" "double-clicked" "17:31:22"
EventSource : Button managed by the GUI of Win32
EventHandler : Handling method in the application code
EventManager: OS or GUI library code



Visual C++ Message Handling分析（1）

In <MainFrame.h>

```
……
//{{AFX_MSG(CMainFrame) //注释宏
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
        afx_msg void OnSizing(UINT fwSide, LPRECT pRect);
        afx_msg void OnStructTabSelChange(NMHDR* pNMHDR, LRESULT* pResult) ;
        afx_msg void OnViewInfoBar();
        afx_msg void OnUpdateViewInfoBar(CCmdUI* pCmdUI);
        afx_msg void OnActivate(UINT nState, CWnd* pWndOther, BOOL bMinimized);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
……
```

afx_msg宏表示声明的是一个消息响应函数

针对每一个事件，EventHandler都分别是什么？

Visual C++ Message Handling分析（2）

In <MainFrame.cpp>

开始消息映射的定义

```
……
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
//{{AFX_MSG_MAP(CMainFrame)
      ON_WM_CREATE()
      ON_WM_DESTROY()
      ON_WM_SIZING()
      ON_NOTIFY(TCN_SELCHANGE, IDI_STRUCTTABCTRL,
      OnStructTabSelChange)
      ON_COMMAND(ID_VIEW_INFO_BAR, OnViewInfoBar)
      ON_UPDATE_COMMAND_UI(ID_VIEW_INFO_BAR,
      OnUpdateViewInfoBar)
      ON_WM_ACTIVATE()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
……
```

说明：要处理的这些事件已经在系统中注册，可能为系统GUI所发出，也可能为其他应用所发出

Java程序中的事件处理分析

- 在Java程序中，处理事件的一般步骤是（类似Win32事件机制）：

  （1）注册监听器以监听事件源产生的事件(如通过**ActionListener**来响应用户点击按钮)；

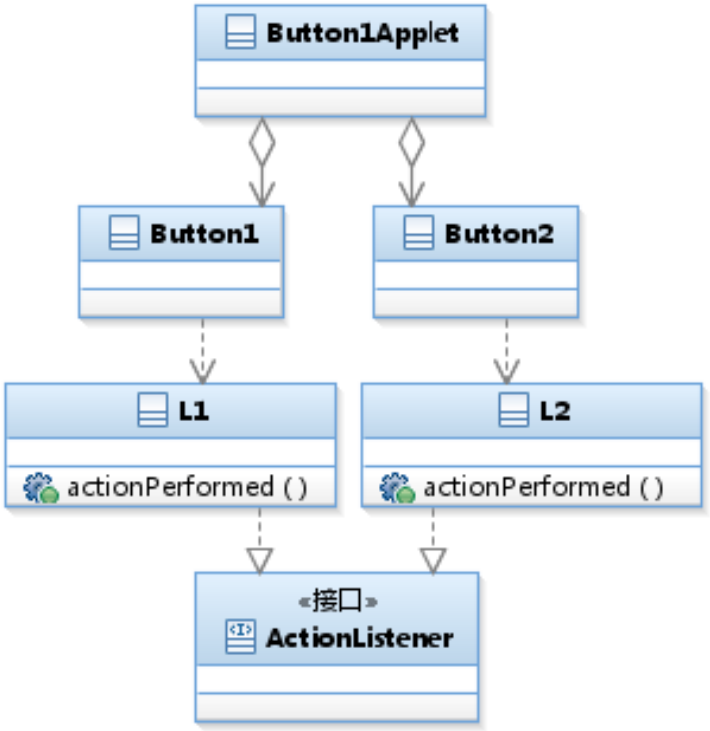
  （2）定义处理事件的方法(如在ActionListener中的actionPerformed中定义相应方法)

- 两种事件派遣实现方式：

  （1）选择广播式：事件（如点击按钮）发生时，该事件仅会传递到注册过的监听器（ActionListener）中，并进行处理；

  （2）广播式：利用继承自父类的监听器以及多个if语句来决定是哪个组件产生的事件并加以处理

Java选择广播式实现案例（1）



使用多个监听器分别响应各种事件

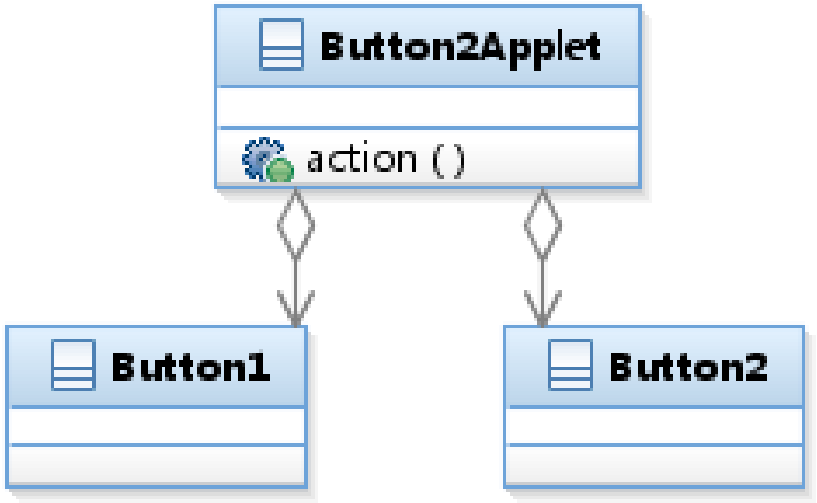
```
public class Button1Applet extends Applet{
        Button b1 = new Button("Button 1");
        Button b2 = new Button("Button 2");
        public void init(){
                b1.addActionListener(new L1(this));
                b2.addActionListener(new L2(this));
                add(b1);
                add(b2);
        }
}
```

Java选择广播式实现案例（2）

```
class L1 implements ActionListener{
        private Applet applet;
        L1 (Applet applet){this.applet = applet;}
        public void actionPerformed(ActionEvent e) {
                applet.getAppletContext().showStatus("Button 1 pressed");
        }
}
class L2 implements ActionListener{
        private Applet applet;
        L2 (Applet applet){this.applet = applet;}
        public void actionPerformed(ActionEvent e) {
                applet.getAppletContext().showStatus("Button 2 pressed");
        }

}
```

Java广播式实现案例（1）

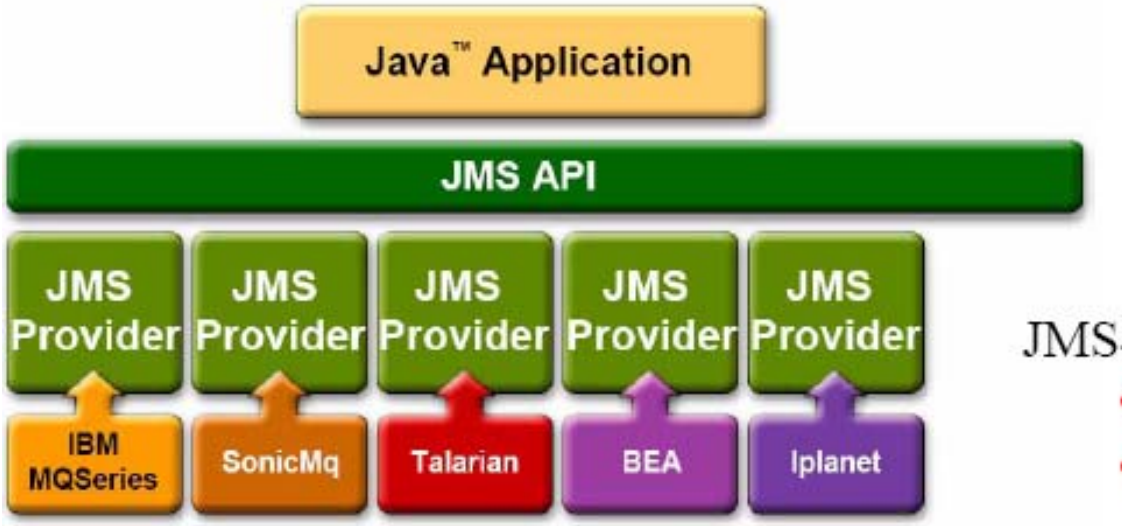


Java广播式实现案例（2）

利用继承自父类的监听器决定响应的事件

```
public class Button2Applet extends Applet{
        Button b1 = new Button("Button 1");
        Button b2 = new Button("Button 2");
        public void init(){
                add(b1);
                add(b2);
        }
        public boolean action (Event evt, Object obj){
                if(evt.target.equals(b1))
                        getAppletContext().showStatus("Button 1 pressed");
                else if(evt.target.equals(b2))
                        getAppletContext().showStatus("Button 2 pressed");
                else
                        return super.action(evt, obj);
                return true;
        }
}
```

JMS (Java Message Service)

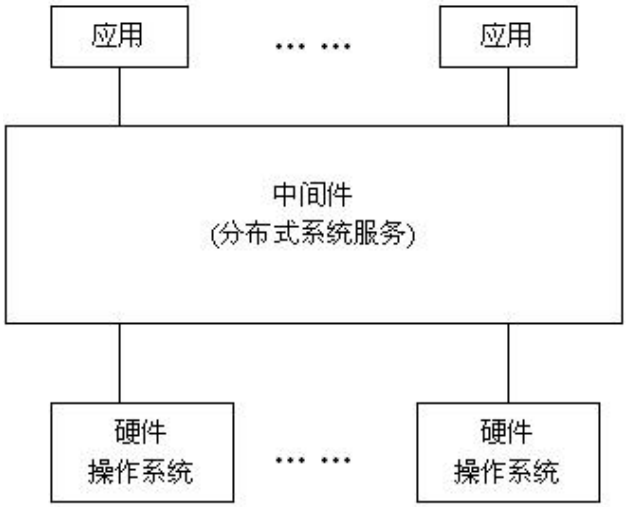


JMS提供了两种事件处理策略：

- Point-to-Point (点对点)
- Publish/Subscribe (发布-订阅)

面向消息的中间件

- **MOM（Message-Oriented Middleware）**面向消息的中间件



中间件

面向消息的中间件

面向消息的中间件

此类中间件是指利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信（回顾：进程通信体系结构风格
）。

消息中间件可以既支持同步方式，又支持异步方式。异步中间件比同步中间件具有更强的容错性，在系统故障时可以保证消息的正常传输。异步中间件技术又分为两类：广播方式和发布/订阅方式。由于发布/订阅方式可以指定哪种类型的用户可以接受哪种类型的消息，更加有针对性，事实上已成为异步中间件的非正式标准。目前主流的消息中间件产品有Sun的JMS、IBM的MQ Series，BEA的MessageQ和等。

JMS简介

Java Message Service(JMS, Java消息服务)是SUN及其伙伴公司提出的旨在统一各种消息中间件系统接口的规范。它定义了一套通用的接口和相关语义，提供了诸如持久、验证和事务的消息服务，它最主要的目的是允许Java应用程序访问现有的消息中间件。JMS规范没有指定在消息节点间所使用的通讯底层协议，来保证应用开发人员不用与其细节打交道，一个特定的JMS实现可能提供基于TCP/IP、HTTP、UDP或者其它的协议。

目前许多厂商采用并实现了JMS API，现在，JMS产品能够为企业提供一套完整的消息传递功能，下面将要介绍一些比较流行的JMS商业软件和开源产品。

JMS规范（1）

JMS1.0版本于1998年推出，最新的版本是**2002发布的JMS 1.1规范**。 Java Message Service 规范 1.1 声称：JMS 是一组**接口和相关语义**，它定义了 JMS 客户如何访问企业消息产品的功能。

在 JMS 之前，每一家 MOM 厂商都用专有 API 为应用程序提供对其产品的访问，通常可用于许多种语言，其中包括 Java 语言。JMS 通过 MOM 产品为 Java 程序提供了一个发送和接收消息的标准的、便利的方法。用 JMS 编写的程序可以在任何实现 JMS 标准的 MOM 上运行。

JMS 可移植性的关键在于：JMS API 是由 Sun 作为一组接口而提供的。提供了 JMS 功能的产品是通过提供一个实现这些接口的提供者来做到这一点的。开发人员可以通过定义一组消息和一组交换这些消息的客户机应用程序建立 JMS 应用程序。

JMS规范（2）

JMS支持消息中间件的两种传递模式：**点到点模式（PTP）**和**发布-订阅模式（Pub/Sub）**。在JMS 1.1以前的版本中，每一种都有自己的特定于该模式的一组客户机接口。JMS1.1版本提供了单一的一组接口，它允许客户机可以在两个模式中发送和接收消息。这些“模式无关的接口”保留了每一个模式的语义和行为，是实现 JMS 客户机的最好选择。

统一模式的好处是：

1) 使得用于客户机的编程更简单。

2) 队列和主题的操作可以是同一事务的一部分。

3) 为JMS提供者提供了优化其实现的机会。

IBM MQSeries

IBM MQ系列产品提供的服务使得应用程序可以使用消息队列进行相互交流，通过一系列基于Java的API，提供了MQSeries在Java中应用开发的方法。它支持点到点和发布/订阅两种消息模式，在基本消息服务的基础上增加了结构化消息类，通过工作单元提供数据整合等内容。

Web Logic

Web Logic是BEA公司实现的基于工业标准的J2EE应用服务器，支持大多数企业级Java API，它完全兼容JMS规范，支持点到点和发布/订阅消息模式，它具有以下一些特点：

1) 通过使用管理控制台设置JMS配置信息；

2) 支持消息的多点广播；

3) 支持持久消息存储的文件和数据库；

4) 支持XML消息，动态创建持久队列和主题。

SonicMQ

SonicMQ是Progress公司实现的JMS产品。除了提供基本的消息驱动服务之外，SonicMQ也提供了很多额外的企业级应用开发工具包，它具有以下一些基本特征：

1) 提供JMS规范的完全实现，支持点到点消息模式和发布/订阅消息模式；
2) 支持层次安全管理；
3) 确保消息在Internet上的持久发送；
4) 动态路由构架（DRA）使企业能够通过单个消息服务器动态的交换消息；
5) 支持消息服务器的集群。

Active MQ

Active MQ是一个基于Apache 2.0 License发布，**开放源码的JMS产品**。其特点为：

1) 提供点到点消息模式和发布/订阅消息模式；
2) 支持JBoss、Geronimo等开源应用服务器，支持Spring框架的消息驱动；
3) 新增了一个P2P传输层，可以用于创建可靠的P2P JMS网络连接；
4) 拥有消息持久化、事务、集群支持等JMS基础设施服务。

Apollo

Apollo以Active MQ原型为基础，是一个更快、更可靠、更易于维护的消息代理工具，Apache称Apollo为最快、最强健的**STOMP**（Streaming Text Orientated Message Protocol，流文本定向消息协议）服务器。其特点为：

1)支持STOMP, AMQP, MQTT, WebSockets等多种协议；

2)主题持久订阅；

3)支持SSL/TLS，证书验证

4)消息过期和交换；

5) REST Management API。

OpenJMS

OpenJMS是一个开源的JMS规范的实现，它包含以下几个特征：

1) 它支持点到点模型和发布/订阅模型；

2) 支持同步与异步消息发送；

3) 可视化管理界面，支持Applet；

4) 能够与Jakarta Tomcat这样的Servlet容器结合；

5) 支持RMI、TCP、HTTP与SSL协议。

A	B	C	D	E
特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
PRODUCER-COMSUMER	支持	支持	支持	支持
PUBLISH-SUBSCRIBE	支持	支持	支持	支持
REQUEST-REPLY	支持	支持		
API完备性	高	高	高	高
多语言支持	支持，JAVA优先	语言无关	只支持JAVA	支持，java优先
单机吞吐量	万级	万级	万级	十万级
消息延迟		微秒级	毫秒级	毫秒级
可用性	高（主从）	高（主从）	非常高（分布式）	非常高（分布式）
消息丢失	低	低	理论上不会丢失	理论上不会丢失
消息重复		可控制		理论上会有重复
文档的完备性	高	高	高	高
提供快速入门	有	有	有	有
首次部署难度		低		中
社区活跃度	高	高	中	高
商业支持	无	无	阿里云	无
成熟度	成熟	成熟	比较成熟	成熟日志领域
特点	功能齐全，被大 量开源项目使用	由于Erlang语言的并发能 力，性能很好	各个环节分 布式扩展设计，主从 HA；支持上万个队列；多 种消费模式；性能很好	
支持协议	OpenWire、 STOMP、 REST、XMPP、AMQP	AMQP	自己定义的一 套(社区提供 JMS--不成熟)	
持久化	内存、文件、数据库	内存、文件	磁盘文件	
事务	支持	支持	支持	
负载均衡	支持	支持	支持	
管理界面	一般	好	有web console实现	
部署方式	独立、嵌入	独立	独立	
评价	优点：成熟的产品，已经在很多公 司得到应用（非大规模场景）。有 较多的文档。各种协议支持较好， 有多重语言的成熟的客户端； 缺点：根据其他用户反馈，会出莫 名其妙的问题，切会丢失消息。其 重心放到 activemq6.0 产品—apollo 上去了，目前社区不活跃，且对5.x 维护较少； Activemq 不适合用于上 千个队列的应用场景。	优点：由于erlang语言的特 性，mq性能较好；管理界面 较丰富，在互联网公司也有 较大规模的应用；支持amqp 系诶，有多中语言且支持 amqp的客户端可用； 缺点：erlang语言难度较 大。集群不支持动态扩展。	优点：模型简单，接口易用（JMS的接口很多 场合并不太实用）。在阿里大规模应用。目前 支付宝中的余额宝等新兴产品均使用 rocketmq 。集群规模大概在50台左右，单日处理消息上 百亿；性能非常好，可以大量堆积消息在 broker中；支持多种消费，包括集群消费、广 播消费等。开发度较活跃，版本更新很快。 缺点：产品较新文档比较缺乏。没有在mq核心 中去实现JMS等接口，对已有系统而言不能兼 容。阿里内部还有一套未开源的MQAPI，这一 层API可以将上层应用和下层MQ的实现解耦 （阿里内部有多个mq的实现，如notify、 metaq1.x，metaq2.x，rocketmq等），使得下 面mq可以很方便的进行切换和升级而对应用无 任何影响，目前这一套东西未开源。	

3.4.5 事件系统的优缺点分析

Advantages and Disadvantages of Event Systems

事件系统：优点

- 能够很好的支持交互式系统（如用户输入/网络通信等），系统中的操作可异步执行，不必同步等待执行结果。
- 为软件复用提供了强大的支持。当需要将一个构件加入现存系统中时，只需将它注册到系统的事件中；
- 为系统动态演化带来了方便。构件独立存在，当用一个构件代替另一个构件时，不会影响到其它构件的接口；
- 对事件的并发处理将提高系统性能；
- 健壮性：一个构件出错将不会影响其他构件。

事件系统：缺点

- 分布式的控制方式使得系统的同步、验证和调试变得异常困难。
- 构件放弃了对系统计算的控制。一个构件触发一个事件时，不能确定其他构件是否会影响它。而且即使它知道事件注册了哪些构件的构成，它也不能保证这些过程被调用的顺序。
- 既然过程的语义必须依赖于被触发事件的上下文约束，关于正确性的推理存在问题。
- 传统的基于先验和后验条件的验证变得不可能。
- 数据交换的问题。有时数据可被一个事件传递，但另一些情况下，基于事件的系统必须依靠一个共享的仓库进行交互。在这种情况下，全局性能和资源管理便成了问题。
- 相比直接方式的连接，隐式调用增加了中间层必要的消耗，会降低事件的响应速度。

作 业

- 1.常见进程间通信方式有哪些？举出至少三种说明其优缺点，并选择其中一种试着加以实现。
- 2.自己上机调试“观察者”模式C++/Java代码，构建一个“点对点”或“发布订阅”模式的C++/Java小应用程序。
- 3.使用Spy++工具，完成如下操作实验：
  - 激活灰色按钮
  - 监视IE上网记录

参考：
http://baike.baidu.com/link?url=C3qR3dxBCY2Ul8VKhFX34FSpDOdZghgFM-1mv-xhojRAK7_7xJcJQ4vVag_BxiKFyCpJxmoWIEtmRkGxW2uag_