原文出处：Event Programming http://msdn.microsoft.com/zh-cn/magazine/cc163659(en-us).aspx

在微软 .NET 框架中可以定义托管类事件并用委托和 = 操作符处理这些事件。这种机制似乎很有用，那么在本机 C 中有没有办法做同样的事情？

Several Readers

确实如此！Visual C .NET 具备所谓统一事件模型（Unified Event Model），它可以像托管类一样实现本机事件（用 __event 关键字），但是由于本机事件存在一些不明显的技术问题，而微软的老大不打算解决这些问题，所以他们要我正式奉劝你不要使用它们。那么这是不是就是说 C 程序员与事件无缘了呢？当然不是！可以通过别的方法实现。本文我将向你展示如何轻松实现自己漂亮的事件系统。

但是在动手之前，让我先大体上介绍一下事件和事件编程。它是个重要的主题，当今对事件没有坚实的理解，你是无法编写程序的——什么是事件以及什么时候使用事件。

成功的编程完全在于对复杂性的掌控。很久以前，函数被称为“子程序”（我知道，我这样说证明我已经老了！）管理复杂性的主要方式之一是自顶向下的编程模式。高层实现类似“宇宙模型”，然后将它划分为更小的任务如：“银河系模型”以及“太阳系模型”等等，直到任务被划分为可以用单个函数实现为止。目前自顶向下的编程模型仍被用于过程化的任务实现当中，但它不适用于发生顺序不确定的实时事件响应系统。经典的例子便是 GUI，程序必须响应用户的某些行为，比如按键或是鼠标移动。实际上，事件编程很到程度上源于图形用户界面的出现。

在自顶向下的模型中，在顶部的高级部分对低级的实现各种不同任务的函数——如 DoThis，DoThat 进行食物链式的调用。但不久以后，低层部分需要回调（talk back），在 Windows 中，可以调用 Rectangle 或 Ellipse 绘制一个矩形或椭圆，但最终 Windows 需要调用你的应用程序来画窗口。但应用程序都还不存在，它仍然处于被调用度状态！那么 Windows 如何知道要调用哪个函数呢？这就是事件用处之所在。

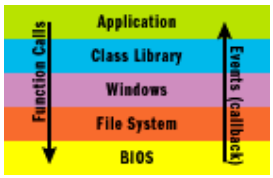

Figure 1 自顶向下和自底向上

在每个 Windows 程序的核心——不论是直接用 C 语言编写的还是使用 MFC 或 .NET 框架类编写——都是一个处理消息的窗口过程，这些消息如：WM_PAINT, WM_SETFOCUS 和 WM_ACTIVATE。你（MFC 或 .NET）实现窗口过程并将它传递给 Windows。到了该画窗口，改变输入焦点以及激活窗口的时候，Windows 用相应的消息代码调用你的过程。这个消息就是事件。窗口过程就是事件处理器。

如果过程化编程是自顶向下的，事件编程是自底向上。在典型的软件系统中，函数的调用流是从较高级部分到低级部分进行的；而事件是以相反的方向过滤的，如 Figure 1 所示。当然，在现实的开发中层次关系并不总是这么清晰。许多软件系统看起来更像 Figure 2 所示的情况：

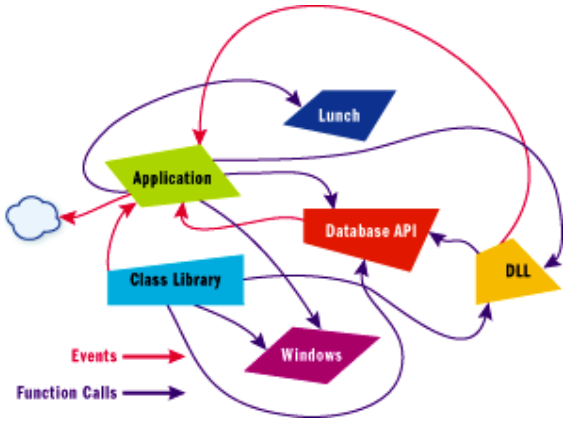

Figure 2 混合模型

那么到底什么叫事件？其实，事件就是回调。而不是在编译时就已知名字的函数调用，组件调用在运行时调用你提供的函数。在 Windows 中，它是一个窗口过程。在 .NET 框架中，它叫做委托。不管术语怎么叫，事件提供了一种软件组件调用函数的方式，这种调用方式直到运行时才知道要调用什么函数。回调被称为事件处理器。发生或触发一个事件意味调用这个事件处理器。为此，事件接收部分首先得给事件源提供一个事件处理器的指针，这个过程叫注册。

通常在以下几种场合下我们要使用事件：

- 通知客户机实际的事件：用户按下某个按键；午夜时钟敲响；风扇停止工作造成 CPU 烧毁；

- 当拷贝文件或搜索巨型数据库时，报告耗时操作的过程，组件可以周期性地触发某个事件以报告已拷贝了多少文件或已搜索了多少记录；
- 如果你使用 IWebBrowser2 在自己的应用程序中宿主 IE，报告所发生的重要的或引起注意的事件，浏览器会在导航到某个新页面之前或之后通知你，或者在创建一个新窗口时通知你。
- 调用应用程序提供的算法：C 运行时库函数 qsort 排序对象数组，但你必须提供比较函数。借助许多 STL 容器也能实现同样的诀窍.大多数程序员不会调用 qsort 回调某个事件，但你没有理由不考虑那种方式。它是“时间比较”事件。

一些读者问：异常和事件之间有什么差别？主要差别是：异常表示不应该发生的意外情况。例如，你的程序运行耗尽内存，或者遇到被零除。这些都是你并不希望发生的异常情况，并且一旦出现这些情况，你的程序必须要做出相应的处理。另一方面，事件则是每天常规操作的部分并且完全是预期的。用户移动鼠标或按下某个键。浏览器导航到一个新页面。从控制流的角度看，事件是一次函数调用，而异常则是堆栈的突然跳跃，用展开的语义销毁丢失的对象。

有关事件常见的概念误解是认为它们是异步的。虽然事件常常被用于处理用户输入和其它异步发生的行为 ，但事件本身是以同步方式发生的。触发一个事件与调用该事件处理器是同一件事情。用伪码表示就像如下的代码段：

```
// raise Foo event
for (/* each registered object */) {
obj->FooHandler(/* args */);
}
```

控制立即传到事件处理器，并且不会返回，除非处理完成。某些系统提供某种以异步触发事件的方式，例如，在 Windows 中，你可以用 PostMessage 代替 SendMessage。控制会从 PostMessage 立即返回，该

消息是后来才处理的。但是 .NET 框架中的事件以及我在这里讨论的事件是在触发时被立即处理的。当然，你总是可以触发来自运行在单独的线程中的消息代码事件，或者使用异步委托调用在线程池中执行每个事件处理器，在这种情况下，相对于主线程来说，事件是异步发生的。

Windows 处理事件的方式完全是通过窗口过程以及一成不变的 WPARAM/LPARAM 参数，按照现代编程标准来说，简陋而粗糙。即便是在今天，每个 Windows 程序仍然在使用这种机制。有些程序员为了传递事件，甚至创建 不可见窗口。窗口过程并不是真正意义上的事件机制，因为在 Winodows 中每个窗口只允许有一个窗口过程，虽然也可以链接多个过程，比如每个过程都调用其前面的过程，也就是众所周知的子类化过程。在真正的事件系统中，相同的事件可以不分等级地注册多个接收者。

在 .NET 框架中，事件是很成熟的机制。任何对象都可以定义事件，并且多个对象可以侦听这些事件。.NET 中的事件使用委托来实现，委托是 .NET 中的术语，它实际上就是以前说所的回调。最重要的是，委托是类型安全的。不再使用 void* 或者 WPARAM/LPARAM。

为了用托管扩展定义一个事件，你得用 __event 关键字。例如，Windows::Forms 中的 Button 类有一个 Click 事件：

```
//in Button classpublic:

__event EventHandler* Click;
```

这里 EventHandler 是某个函数的委托，该函数带有参数：Object (也就是 sender) 和 EventArgs：

```
public __delegate void EventHandler
(
Object* sender,
EventArgs* e
);
```

为了接收事件，你必须用正确的签名实现处理器成员函数并创建一个委托来包装该函数，然后调用事件的 = 操作符注册你的处理器/委托。对于上面的 Click 事件，代码应该像这样：

```
// event handler
void CMyForm::OnAbort(Object* sender, EventArgs *e)
{...}

// register my handler
m_abortButton->Click  = new EventHandler(this, OnAbort);
```

注意该处理器函数必须具备由委托定义的签名。这是托管扩展的基本原则。但是你的问题涉及的不是托管事件，你问的是本机事件——如何实现本机 C 事件？C 本身没有内建的事件机制，那么该怎么实现呢？你可以用 typedef 来定义一个回调并让客户机来提供这个回调，这种做法有些类似 qsort——但那样太老土了。更不用说处理多个事件时的繁琐。相对于静态外部函数来说，用成员函数作为事件处理器是最丑陋的做法。

一种比较好的方法是创建一个定义事件的接口。那是 COM 的做法。但你不需要用 C 编写沉重的 COM 代码；你可以用一个简单的类。我写了一个类来做示范：CPrimeCalculator；这个类的功能是查找素数。代码如 Figure 3 所示。CPrimeCalculator::FindPrimes(n) 查找开始的 n 个素数。其工作原理是这样的，

CPrimeCalculator 触发两种事件：Progress 事件和 Done 事件。这些事件都定义在 IPrimeEvents 接口中。IPrimeEvents 接口不是 .NET 和 COM 意义上的接口；它是一个纯粹的 C 抽象基类，它为每个事件处理器定义
签名（参数和返回类型）。处理 CPrimeCalculator 的客户机必须实现 IPrimeEvents，然后调用 CPrimeCalculator::Register 来注册它们的恶接口。CPrimeCalculator 将对象/接口添加到其内部列表（list）中。由于它会对每个整数进行素数检查，CPrimeCalculator 则周期性地报告到目前为止找到了多少个素数：

```
// in CPrimeCalculator::FindPrimes

for (UINT p=2; p<max; p  )
{
// figure out if p is prime
if (/* every now and then */)
NotifyProgress(GetNumberOfPrimes()); ...
}
NotifyDone();
```

CPrimeCalculator 调用内部辅助函数 NotifyProgress 和 NotifyDone 来触发事件。这些函数遍历客户机对象列表，为每个客户机调用相应的事件处理器。代码如下：

```
void CPrimeCalculator::NotifyProgress(UINT nFound)
{
    list<IPrimeEvents*>::iterator it;
    for (it=m_clients.begin(); it!=m_clients.end(); it  )
    {
        (*it)->OnProgress(nFound);
    }
}
```

如果你对 STL 不熟悉，去看看有关迭代器反引用操作符的内容，它返回当前指向的对象，上面代码段中，for 循环里的代码等同于：

```
IPrimeEvents* obj = *it;
obj->OnProgress(nFound);
```

触发 Done 事件的 NotifyDone 函数做法类似，它没有参数，如 Figure 3 所示。你也许觉得 Done 事件是多余的，因为当 FindPrimes 返回控制时，客户机已经知道 CPrimeCalculator 完成了工作。没错——但有一种情况除外，那就是多个客户机注册接收的事件，并且调用 CPrimeCalculator::FindPrimes 的对象可能不是同一个。Figure 4 是我的测试程序 PrimeCalc。该程序为素数事件实现了两个不同的事件处理器。第一个处理器是主对话框本身，CMyDlg，它利用多继承实现 IPrimeEvents。该对话框处理 OnProgress 和 OnDone，并在对话窗口显示进度，完成后发出蜂鸣声。其它的事件处理器，如 CTracePrimeEvents 也实现了 IPrimeEvents，这个实现显示诊断（TRACE）流中的信息。如 Figure 6 所示，在我的 TraceWin 程序（参见 2004 年三月的专栏）中显示的范例输出。我写的 CTracePrimeEvents 展示了多个客户机如何注册相同的事件。

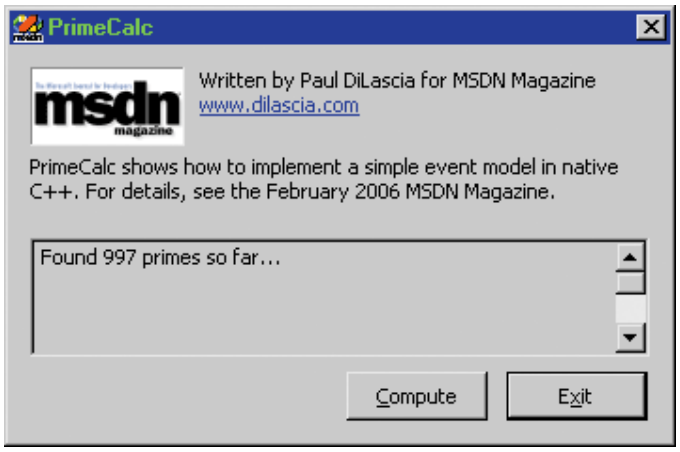

Figure 5 运行中的 PrimeCalc

从使用 CPrimeCalculator 来编写应用的程序员角度看，处理事件简单而直白。从 IPrimeEvents 派生，实现处理器函数，然后调用 Register。从编写触发事件的类的程序员看来，这个过程有些冗长乏味。首先你得定义事件接口。这并没有什么不好。但接着你得编写 Register 和 Unregister 函数，每个 Foo 事件都得有一个相应的 NotifyFoo 函数。如果有 15 个事件的话，那就十分令人不爽了，尤其是每个 NotifyFoo 函数的模式都相同：

```
void CMyClass::NotifyFoo(/* args */)

{

    list<IPrimeEvents*>::iterator it;

    for (it=m_clients.begin(); it!=m_clients.end(); it  ) {

        (*it)->OnFoo(/* args */);

    }

}
```

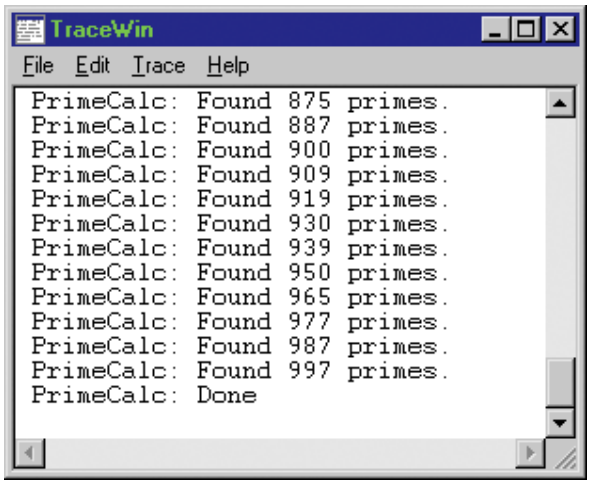

Figure 6 PrimeCalc 在 TraceWin 中的输出

NotifyFoo 迭代客户机列表，为每个注册的客户机调用相应的 OnFoo 处理器，并传递任何需要的参数。有没有什么方法实现这个一般过程，比如用宏或者模板来封装这种繁琐而固定的样板代码，将自己从重复性劳动中解放出来呢？实际上是有的。下个月的专栏文章我们将讨论这个问题。记住在同一时间，同一频道，咱们再见——顺祝编程愉快！

出处：http://msdn.microsoft.com/zh-cn/magazine/cc163645(en-us).aspx

在本文的第一部分（事件编程一），我回答了一个关于用 C 实现本机事件的问题。讨论了一般意义上的事件并示范了如何用接口为你的类定义事件处理器，事件的处理必须在客户机实现。我的实现有一些缺陷，我承诺过最终要解决掉，本文就来完成这件事情。

在开始之前，先简单回顾一下前面写的那个程序，PrimeCalc。如 Figure 1 所示：

Figure 1 计算素数

程序中使用了一个计算素数的类 CPrimeCalculator，这个类发起两个事件：Progress 和 Done。当搜

索到素数时，该类触发 Progress 事件以报告目前发现了多少素数。完成处理后触发 Done 事件。这两个事件都是由接口 IPrimeEvents 定义的：

```
class IPrimeEvents {
public:
  virtual void OnProgress(UINT nPrimes) = 0;
  virtual void OnDone() = 0;
};
```

客户机要想处理事件必须得从 IPrimeEvents 派生，实现事件处理函数，并调用 CPrimeCalculator::Register 来注册其接口。CPrimeCalculator::Register 会将客户机对象/接口添加到其内部列表。当触发了一个 Progress 事件时，CPrimeCalculator 便调用辅助函数 NotifyProgress：

```
void CPrimeCalculator::NotifyProgress(UINT nFound)
{
  list::iterator it;
  for (it=m_clients.begin(); it!=m_clients.end(); it  )
{
    (*it)->OnProgress(nFound);
  }
}
```

NotifyProgress 遍历客户机列表，调用每个客户机的 OnProgress 处理函数。当某个程序员使用 CPrimeCalculator 时，编写事件处理代码很容易——只要从 IPrimeEvents 派生并实现处理器即可。但是在实现这种触发事件的 CPrimeCalculator 类机制时冗长乏味。你必须得为每个事件（如 Foo）实现诸如 NotifyFoo 这样的函数，即使处理模式一模一样。事件触发代码被划分在两个类中，事件接口 IPrimeEvents 和 事件源 CPrimeCalculator。如果你想将同样的事件接口用于不同的事件源那该怎么办？IPrimeEvents 是很通用，我可能将它改名为 IProgressEvents 并将它用于任何以整数形式报告处理进度的类并在完成处理时用 Done。但每个触发 Progess 事件的类必须重新实现触发事件的通知函数。理想情况下，所有事件代码都应该放在单个类中。

既然通知函数在本文中是一种实验模型，那么自然会问这样的问题：它们有没有某种通用的实现方法？我能将整个事件机制封装到单个的类、模板或宏，或者任何事件源能使用的其它什么类型中吗？答案是肯定中的肯定。我将示范如何创建一个使用宏和模板的事件系统，以便将事件处理的代码量降至最低限度。我们的旅程需要借助一些高境界的 C 操作，比如嵌套模板以及仿函数类（functor class）。

我将分几个步骤实现这个系统。目的是编写一个实现通知函数 NotifyProgress 以及 NotifyDone 的模板。每个函数都具备相似而又不完全一样的模型：

```
// NotifyFoo — raise Foo event
list<IPrimeEvents*>::iterator it;
for (it=m_clients.begin(); it!=m_clients.end(); it  )
{
  (*it)->OnFoo(/*args*/);
}
```

也就是说迭代客户机列表，并针对每个客户机调用 OnFoo，传递事件参数。如何把它写成一个模板呢？可以将接口 IPrimeEvents 参数化为一个类型 T，但如何参数化事件处理函数 OnFoo，程序员可能选择的任何名字和签名。

任何时候你参数化某个函数时，都应该考虑：仿函数，也叫做 functor。仿函数是 C 语言中将函数转换为类的一种机制，它代替了给回调函数传递指针的做法，而是传递仿函数类的实例。在标准模板库 STL 中包含有丰富的 Functor，并实现了一些使用 functor 的算法，尤其是 for_each 算法，在本文中很有用：

```
for_each(m_clients.begin(), m_clients.end(),
NotifyProgress(nFound));
```

for_each 算法从头到尾迭代容器元素，并对每个元素调用函数对象 NotifyProgress。这里说的“函数对象”到底是指的什么呢？不是一个函数，它是一个对象。这个类看起来像下面这个样子：

```
class NotifyProgress {

protected:
  UINT m_nFound;
public:
  NotifyProgress(UINT n) : nFound(n) { }
  void operator()(IPrimeEvents* obj)
  {
    obj->OnProgress(nFound);
  }
};
```

NotifyProgress 实现函数 operator()(IPrimeEvents*)，它是 for_each 算法需要的东西。一般来讲，如果你具备一个类型为 T 对象集合，for_each 会需要一个实现 operator()(T) 的仿函数（functor）。它调用该集合中 T 对象的这个操作符。所以这里函数 operator 有一个 IPrimeEvents 指针参数并返回 void —— 因为客户机列表是一个 IPrimeEvents 指针列表。为了传递附加参数，构造函数将它们保存在数据成员里。NotifyProgress(nFound) 调用构造函数以创建一个用 m_nFound=nFound 初始化的堆栈实例。所以，任何触发 Foo 事件的 Foo 仿函数的一般模式是这样的：

```
class NotifyFoo {
protected:
  ARG1 m_arg1; // whatever, as many as needed
public:
  NotifyProgress(ARG1 a1, ...) : m_arg1(a1) { }
  void operator()(IMyEvents* obj)
  {
    obj->OnFoo(m_arg1, ...);
  }
};
```

构造函数将事件参数作为数据成员来保存，函数 operator 将它们传递到对象事件处理函数。对于所有仿函数来说，最终结果是——将函数 OnFoo 转换为类 NotifyFoo。这样做为什么会有用呢？因为我能编写一个模板。在我开始做之前，有一件事我必须得提一下。那就是你必须从一个叫 unary_function 的 STL 类派生你的仿函数类：

```
class NotifyProgress :
  public unary_function
{  .
  . // as before  .
```

```
};
```

也就是说，NotifyProgress 是一个一元函数，其函数 operator 带一个参数，IPrimeEvents 指针并返回 void。该一元函数使你的仿函数类“可适配”，使你能将它与 STL 适配器，如：not1、bind2nd 等等进行结合。但是即使你从来都没有打算使用适配器，就像我的事件处理例程，unary_function 仍然不失为一个好主意，因为它向这个世界宣告：“这是一个函数类。”它是一种将代码文档化的方式。有关适配器的详细讨论，参见 Effective STL：50 Specific Ways to Improve Your Use of the Standard Template Library (Addison-Wesley, 2001) by Scott Meyers

STL 的高手们也许会问：为什么我不使用 mem_fun 适配器直接将 IPrimeEvents::OnProgress 转换为函数对象。因为 OnProgress 是虚拟函数，我不能适配一个虚拟函数。如果这样做，要触及到基类。如果你使用 Boost 库，可以用其捆绑适配器直接将 OnFoo 这样的虚拟事件处理器转换为仿函数，不用编写仿函数。如果你不明白我所讲的这些内容，不用害怕，不看这些内容好了。

当然，我还需要一个 Done 事件的 NotifyDone。由于 Done 没有参数，构造函数也没有：

```
class NotifyDone : public unary_function
{
public:
  NotifyDone() { }
  void operator()(IPrimeEvents* obj)
  {
    obj->OnDone();
  }
};
```

现在我有了自己的仿函数类，我可以用 for_each 代替手工迭代客户机列表。可我把它们放在哪呢？仿函数属于与事件说明有关的范畴，所以我把它们放在 IPrimeEvents 接口中，用嵌套类的形式。代码如 Figure 2 所示。细心的读者会注意到我在两个地方还做了细小的恶修改。仿函数的命名没有用 NotifyProgress，而是叫做 Progress。稍后你会明白这样做使代码更易读；还有就是我没有把事件处理器都声明为纯虚拟函数，而是将它们定义为空实现。IPrimeEvents 只有两个事件，但对于一般的事件机制来说，如果程序员感兴趣的的处理并不多，但要让他们实现每一个事件处理器似乎不是很友好。所以这里每个处理器默认实现什么也不做。为了使基类抽象化，我声明了一个纯虚拟析构函数。当你想抽象化一个没有任何纯虚函数的基类时，这是一个标准的 C 技巧。唯一的要做的是你必须定义一个析构函数。纯虚拟函数没有定义——除非它是析构函数。既然每一个派生类的析构都调用其基类的析构函数，那么基类需要一个实现，即便它是纯虚拟的：

```
inline IPrimeEvents::~IPrimeEvents() { }
```

有了我的仿函数定义，CPrimeCalculator 是这样触发 Progress 事件的：

```
// in CPrimeCalculator:

void NotifyProgress(UINT nFound)
{
  for_each(m_clients.begin(), m_clients.end(),
    IPrimeEvents::Progress(nFound));
}
```

到这里，我已经介绍了仿函数类 Progress 和 Done，同时，NotifyProgress 和 NotifyDone 都能用 STL 的 for_each 算法。下一步该做什么？记住，我的目的是完全摆脱 NotifyFoo 函数——或者说得更具

体一点，就是把它们实现为模板，以便程序员在创建事件时不必为他们定义的每个事件编写千篇一律的函数。将 for 循环转化为 for_each 算法只是万里长征的第一步。

通过将虚拟成员函数 OnFoo 转换为 Foo 仿函数类型，从而为模板化创造条件。（仿函数在这里有点像 .NET 中的委托。）现在我的通知函数根据类型的不同而不同，替代了函数名，我可以将它们参数化。这样一来，我便可以将整个事件实现移出 CPrimeCalculator ，把它们放入新的模板类 CEventMgr 中，这是一个完全通用的类。如 Figure 3 所示。CEventMgr<I> 保存 I* 指针列表。它具备 Register 和 Unregister 方法以便添加元素和从其列表中删除元素，此外它还有一个模板成员函数 Raise 用于触发事件：

```
template
class CEventMgr
{  ...
  template
  void Raise(F fn)
  {
    for_each(m_clients.begin(), m_clients.end(), fn);
  }
};
```

很难相信，平时几乎碰不到的模板套模板的情况？在此处派上用场了。现在触发事件我们可以这样做：

```
void NotifyProgress(UINT nFound)
{
  m_eventmgr.Raise(IPrimeEvents::Progress(nFound));
}
```

没有 for 循环，甚至都没有 for_each，所有细节都被封装在 CEventMgr 之中，事件的触发使用一行代码。我甚至可以完全省略掉 NotifyProgress，每当想要触发事件时仅仅调用 CEventMgr::Raise 即可——然而，好的编码规范促使我宁愿将 Raise 封装在某个函数中，以防万一我要修改 CEventMgr 或将事件触发函数暴露给客户机。既然 NotifyProgress 是内联函数，就不会有幸能丢失。

如果模板使你伤脑筋，我就再讲清楚一些吧。CEventMgr 是一个参数化的模板类，其参数是事件接口 I。因此 CEventMgr<IPrimeEvents> 根据 IPrimeEvents 实例化一个事件管理器。它保存数据成员 m_clients，该成员是一个 IPrimeEvents 指针列表：list<IPrimeEvents*>。CEventMgr 中是一个模板成员函数：Raise<F>，它将仿函数参数 F 传递给 for_each。所以当你写下面这条语句时：

```
m_eventmgr.Raise(IPrimeEvents::Progress(nFound));
```

编译器明白你试图以 IPrimeEvents::Progress 类型参数调用 CEventMgr::Raise，于是它用模板产生成员函数 CEventMgr::Raise(IPrimeEvents::Progress)。实现代码将仿函数实例传递给 for_each，它为客户机列表中的每个 I* 对象调用仿函数的 operator()。仿函数调用对象的 OnProgress 处理例程——这就是我想要的！模板不是很酷吗？

我们已经快到终点了。仿函数让我参数化事件方法并使用 for_each，但它们还是太长，我讨厌敲入太多的东西。所以最后一步是引入一些宏来解决这个问题。下面就是 IPrimeEvents 最终的浓缩定义。

```
class IPrimeEvents
```

```
{
  DECLARE_EVENTS(IPrimeEvents);
public:
  DEFINE_EVENT1(IPrimeEvents, Progress, UINT /*nFound*/);
  DEFINE_EVENT0(IPrimeEvents, Done);
};
IMPLEMENT_EVENTS(IPrimeEvents);
```

完整的源代码参见 Figure 4 —— 从代码中你可以体会到我竭尽全力进行精简和浓缩。只留下最基本的信息：每个事件处理器的名字和签名。宏假设 Foo 的事件处理器是 OnFoo。一些编程的唯美主义者不喜欢宏，但我不那样。有工具为什么不使用呢？DECLARE_EVENTS 声明构造函数和析构函数；IMPLEMENT_EVENTS 实现内联析构。宏 DEFINE_EVENT0，DEFINE_EVENT1 以及 DEFINE_EVENT2 分别声明和定义了 OnFoo 事件处理器以及不带参数，带一个参数和带两个参数的 Foo 事件仿函数。如果你需要更多的参数，可以定义一个结构，用一个事件参数来传递此结构的指针：

```
MumbleArgs args;
args.a = 1;
args.b = 2;
// etc.
m_eventMgr.Raise(IMyEvents::Mumble(&args));
```

还有一种选择，你可以实现 DEFINE_EVENT3。但是记住：仿函数对象通过值传递的，所以它们应该很小。当可以传递指针时，为什么要在堆区和栈区来回拷贝一大堆参数呢？如果事件处理器需要返回值，也可以借助结构。为了简单起见，我让事件处理器返回 void。

经常有程序员会问仿函数会不会带来太大的额外开销。事实上，仿函数通常比函数更有效率。理由是它是内联的。当你在 C 中传递指向函数的指针时，即使你将函数定义为内联，它就是一个指针。你不能通过传值的方式来传递一个函数。但是当你传递一个对象实例到某个模板函数时，如果你象那样定义函数，编译器产生的所有东西都是内联的。对于事件来说，通过指针仍然只有一个函数调用，它发生在函数 operator 调用虚拟 OnFoo 处理器的时候。
编程愉快!

一. 先看看事件接口和实现

```
#ifndef IEVENT_H
#define IEVENT_H

/*
 以下各基础设施是在 C++中事件机制的完整实现，事件是面向组件开发的必要特性之一。
 最开始打算用函数指针模拟事件，但由于 C++中成员函数指针不能和 void*相互强转，而且
 typedef 中不能含有模板，所以才不得已以接口继承实现。这样效果也不错 :)

 创 作 者：sky
```

```
 时       间：2005.06.22
 修订时间：2005.06.22
*/

#include "../Collection/SafeArrayList.h"
template<class SenderType ,class ParaType> class EventPublisher ;

class NullType
{
};

// IEventHandler   是事件处理句柄，预定事件的类从此接口继承以实现事件处理函数
template<class SenderType ,class ParaType>
interface IEventHandler
{

public:
 virtual ~IEventHandler(){}
private:
 virtual void HandleEvent(SenderType sender ,ParaType para)    = 0 ;
 friend class EventPublisher<SenderType ,ParaType> ;
};

// IEvent  事件预定方通过此接口预定事件
template<class SenderType ,class ParaType>
interface IEvent
{
public:

 virtual ~IEvent(){}
 virtual void Register    (IEventHandler<SenderType ,ParaType>* handler) = 0 ;
 virtual void UnRegister(IEventHandler<SenderType ,ParaType>* handler) = 0 ;
};

// IEventActivator  事件发布方通过此接口触发事件
template<class SenderType ,class ParaType>
interface IEventActivator
{
public:

 virtual ~IEventActivator(){}
 virtual void Invoke(SenderType sender ,ParaType para) = 0;
 virtual int    HandlerCount() = 0;
 virtual IEventHandler<SenderType ,ParaType>* GetHandler(int index) = 0;
```

```
};

// IEventPublisher 事件发布方发布事件相当于就是发布一个 IEventPublisher 派生类的对象
// 不过仅仅将该对象的 IEvent 接口发布即可。
template<class SenderType ,class ParaType>
interface IEventPublisher : public IEvent<SenderType ,ParaType> ,public IEventActivator<SenderType ,ParaType>
{
};

// EventPublisher 是 IEventPublisher 的默认实现
template<class SenderType ,class ParaType>
class EventPublisher :public IEventPublisher<SenderType ,ParaType>
{
private:
 SafeArrayList< IEventHandler<SenderType ,ParaType> > handerList ;
 IEventHandler<SenderType ,ParaType>* innerHandler ;

public:
 void Register(IEventHandler<SenderType ,ParaType>* handler)
 {
  this->handerList.Add(handler) ;
 }

 void UnRegister(IEventHandler<SenderType ,ParaType>* handler)
 {
  this->handerList.Remove(handler) ;
 }

 void Invoke(SenderType sender ,ParaType para)
 {
  int count = this->handerList.Count() ;
  for(int i=0 ;i<count ;i++)
  {
   IEventHandler<SenderType ,ParaType>* handler = this->handerList.GetElement(i) ;
   handler->HandleEvent(sender ,para) ;
  }
 }

 int  HandlerCount()
 {
  return this->handerList.Count() ;
 }
```

```
  IEventHandler<SenderType ,ParaType>* GetHandler(int index)
  {
    return this->handerList.GetElement(index) ;
  }
};

#endif
```

二. 一个定时器类 Timer，演示如何发布事件。想必大家都知道定时器的用途了哦，这个 Timer 就像 C#中的 Timer 类一样。

```
#ifndef TIMER_H
#define TIMER_H
/*
  Timer  定时器,每经过一段指定时间就触发事件

  创 作 者：sky
  时       间：2005.06.22
  修订时间：2005.06.22
*/
#include "IEvent.h"
#include "Thread.h"

void TimerThreadStart(void* para) ;

class Timer
{
private:
  int spanInMillSecs ;
  volatile bool isStop ;
  volatile bool timerThreadDone ;

public:
  friend void TimerThreadStart(void* para) ;
  IEvent<Timer* ,NullType>* TimerTicked ;

  Timer(int span_InMillSecs)
  {
    this->isStop = true ;
    this->timerThreadDone = true ;
    this->spanInMillSecs = span_InMillSecs ;
    this->TimerTicked = new EventPublisher<Timer* ,NullType> ;
```

```
        }

        ~Timer()
        {
          this->Stop() ;
          delete this->TimerTicked ;
        }

        void Start()
        {
          if(! this->isStop)
          {
            return ;
          }

          this->isStop = false ;
          Thread thread ;
          thread.Start(TimerThreadStart ,this) ;
          //unsigned int    dwThreadId ;
          //HANDLE    hThread    =    (HANDLE)_beginthreadex(NULL,    0    ,    (unsigned    int
(_stdcall*)(void*))&TimerThreadStart , this, 0, &dwThreadId);

        }

        void Stop()
        {
          if( this->isStop)
          {
            return ;
          }

          this->isStop = true ;

          //等待工作线程退出
          while(! this->timerThreadDone)
          {
            Sleep(200) ;
          }
        }

      private:
        void WorkerThread()
        {
          this->timerThreadDone = false ;
```

```
   while(! this->isStop)
   {
    Sleep(this->spanInMillSecs) ;

    if(this->isStop)
    {
     break ;
    }

    NullType nullObj ;
    ((EventPublisher<Timer* ,NullType>*)this->TimerTicked)->Invoke(this ,nullObj) ;
   }

   this->timerThreadDone = true ;
  }
};

void TimerThreadStart(void* para)
{
      Timer* timer = (Timer*)para ;
      timer->WorkerThread() ;
}
#endif
```

三. 预定事件例子

```
class TimerEventExample    :public IEventHandler<Timer* ,NullType> ,CriticalSection
{
private:
  Timer* timer ;

public:
  TimerEventExample(int checkSpan)
  {
   this->timer = new Timer(checkSpan) ;
   this->timer->TimerTicked->Register(this) ;
  }

  ~TimerEventExample()
  {
   delete this->timer ;
  }
```

```
private:
  //处理定时事件
  void HandleEvent(Timer* sender ,NullType para)
  {
    cout<<"Time ticked !"<<endl ;
  }
};
```

//注意：**IEvent** 必须提供虚拟析构函数，否则在 **Timer** 中 **IEvent<Timer* ,NullType>* TimerTicked ;** 在调用 **delete this->TimerTicked ;** 的时候会造成内存泄漏**.**