成员函数指针与高性能的 C++委托

Member Function Pointers and the Fastest Possible C++ Delegates

撰文：Don Clugston

翻译：周翔

引子

标准 C++中没有真正的面向对象的函数指针。这一点对 C++来说是不幸的，因为面向对象的指针（也叫做“闭包(closure)”或“委托(delegate)”）在一些语言中已经证明了它宝贵的价值。在 Delphi (Object Pascal)中，面向对象的函数指针是 Borland 可视化组建库（VCL，Visual Component Library）的基础。而在目前，C#使“委托”的概念日趋流行，这也正显示出 C#这种语言的成功。在很多应用程序中，“委托”简化了松耦合对象的设计模式[GoF]。这种特性无疑在标准 C++中也会产生很大的作用。

很遗憾，C++中没有“委托”，它只提供了**成员函数指针**（member function pointers）。很多程序员从没有用过函数指针，这是有特定的原因的。因为函数指针自身有很多奇怪的语法规则（比如“->*”和“.*”操作符），而且很难找到它们的准确含义，并且你会找到更好的办法以避免使用函数指针。更具有讽刺意味的是：事实上，编译器的编写者如果实现“委托”的话会比他费劲地实现成员函数指针要容易地多！

在这篇文章中，我要揭开成员函数指针那“神秘的盖子”。在扼要地重述成员函数指针的语法和特性之后，我会向读者解释成员函数指针在一些常用的编译器中是怎样实现的，然后我会向大家展示编译器怎样有效地实现“委托”。最后我会利用这些精深的知识向你展示在 C++编译器上实现优化而可靠的“委托”的技术。比如，在 Visual C++(6.0, .NET, and .NET 2003)中对单一目标委托（single-target delegate）的调用，编译器仅仅生成两行汇编代码！

函数指针

下面我们复习一下函数指针。在 C 和 C++语言中，一个命名为 my_func_ptr 的函数指针指向一个以一个 int 和一个 char*为参数的函数，这个函数返回一个浮点值，声明如下：

```
float (*my_func_ptr)(int, char *);

//为了便于理解，我强烈推荐你使用 typedef 关键字。

//如果不这样的话，当函数指针作为一个函数的参数传递的时候，

// 程序会变得晦涩难懂。

// 这样的话，声明应如下所示：

typedef float (*MyFuncPtrType)(int, char *);

MyFuncPtrType my_func_ptr;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

应注意，对每一个函数的参数组合，函数指针的类型应该是不同的。在 Microsoft Visual C++（以下称 MSVC）中，对三种不同的调用方式有不同的类型：__cdecl, __stdcall, 和__fastcall。如果你的函数指针指向一个型如 float some_func(int, char *)的函数，这样做就可以了：

```
my_func_ptr = some_func;
```

当你想调用它所指向的函数时，你可以这样写：

```
(*my_func_ptr)(7, "Arbitrary String");
```

你可以将一种类型的函数指针转换成另一种函数指针类型，但你不可以将一个函数指针指向一个 void *型的数据指针。其他的转换操作就不用详叙了。一个函数指针可以被设置为 0 来表明它是一个空指针。所有的比较运算符（==, !=, <, >, <=, >=）都可以使用，可以使用“ ==0”或通过一个显式的布尔转换来测试指针是否为空（null）。

在 C 语言中，函数指针通常用来像 qsort 一样将函数作为参数，或者作为 Windows 系统函数的回调函数等等。函数指针还有很多其他的应用。函数指针的实现很简单：它们只是“代码指针（code pointer）”，它们体现在汇编语言中是用来保存子程序代码的首地址。而这种函数指针的存在只是为了保证使用了正确的调用规范。

成员函数指针

在 C++程序中，很多函数是成员函数，即这些函数是某个类中的一部分。你不可以像一个普通的函数指针那样指向一个成员函数，正确的做法应该是，你必须使用一个成员函数指针。一个成员函数的指针指向类中的一个成员函数，并和以前有相同的参数，声明如下：

```
float (SomeClass::*my_memfunc_ptr)(int, char *);
```

//对于使用 const 关键字修饰的成员函数，声明如下：

```
float (SomeClass::*my_const_memfunc_ptr)(int, char *) const;
```

注意使用了特殊的运算符（::*），而“ SomeClass”是声明中的一部分。成员函数指针有一个可怕的限制：它们只能指向一个特定的类中的成员函数。对每一种参数的组合，需要有不同的成员函数指针类型，而且对每种使用 const 修饰的函数和不同类中的函数，也要有不同的函数指针类型。在 MSVC 中，对下面这四种调用方式都有一种不同的调用类型：__cdecl, __stdcall, __fastcall, 和 __thiscall。（__thiscall 是缺省的方式，有趣的是，在任何官方文档中从没有对__thiscall 关键字的详细描述，但是它经常在错误信息中出现。如果你显式地使用它，你会看到“它被保留作为以后使用（it is reserved for future use）”的错误提示。）如果你使用了成员函数指针，你最好使用 typedef 以防止混淆。

将函数指针指向型如 float SomeClass::some_member_func(int, char *)的函数，你可以这样写：

```
my_memfunc_ptr = &SomeClass::some_member_func;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

很多编译器（比如 MSVC）会让你去掉“ &”，而其他一些编译器（比如 GNU G++）则需要添加“ &”，所以在手写程序的时候我建议把它添上。若要调用成员函数指针，你需要先建立 SomeClass 的一个实例，并使用特殊操作符“ ->*”，这个操作符的优先级较低，你需要将其适当地放入圆括号内。

```
SomeClass *x = new SomeClass;

(x->*my_memfunc_ptr)(6, "Another Arbitrary Parameter");

//如果类在栈上，你也可以使用“ .*”运算符。

SomeClass y;

(y.*my_memfunc_ptr)(15, "Different parameters this time");
```

不要怪我使用如此奇怪的语法——看起来 C++的设计者对标点符号有着由衷的感情！C++相对于 C 增加了三种特殊运算符来支持成员指针。“ ::*”用于指针的声明，而“ ->*”和“ .*”用来调用指针指向的函数。这样看起来对一个语言模糊而又很少使用的部分的过分关注是多余的。（你当然可以重载“ ->*”这些运算符，但这不是本文所要涉及的范围。）

一个成员函数指针可以被设置成 0，并可以使用“ ==”和“ !=”比较运算符，但只能限定在同一个类中的成员函数的指针之间进行这样的比较。任何成员函数指针都可以和 0 做比较以判断它是否为空。与函数指针不同，不等运算符（<, >, <=, >=）对成员函数指针是不可用的。

成员函数指针的怪异之处

成员函数指针有时表现得很奇怪。首先，你不可以用一个成员函数指针指向一个静态成员函数，你必须使用普通的函数指针才行（在这里“成员函数指针”会产生误解，它实际上应该是“非静态成员函数指针”才对）。其次，当使用类的继承时，会出现一些比较奇怪的情况。比如，下面的代码在 MSVC 下会编译成功（注意代码注释）：

```
#include “stdio.h”

class SomeClass {

public:

virtual void some_member_func(int x, char *p) {

printf("In SomeClass"); };

};

class DerivedClass : public SomeClass {

public:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
// 如果你把下一行的注释销掉，带有 line (*)的那一行会出现错误

// virtual void some_member_func(int x, char *p) { printf("In DerivedClass"); };

};

int main() {

//声明 SomeClass 的成员函数指针

typedef void (SomeClass::*SomeClassMFP)(int, char *);

SomeClassMFP my_memfunc_ptr;

my_memfunc_ptr = &DerivedClass::some_member_func; // ---- line (*)

return 0;

}
```

奇怪的是，&DerivedClass::some_member_func 是一个 SomeClass 类的成员函数指针，而不是 DerivedClass 类的成员函数指针！（一些编译器稍微有些不同：比如，对于 Digital Mars C++，在上面的例子中，&DerivedClass::some_member_func 会被认为没有定义。）但是，如果在 DerivedClass 类中重写（override）了 some_member_func 函数，代码就无法通过编译，因为现在的&DerivedClass::some_member_func 已成为 DerivedClass 类中的成员函数指针！

成员函数指针之间的类型转换是一个讨论起来非常模糊的话题。在 C++的标准化的过程中，在涉及继承的类的成员函数指针时，对于**将成员函数指针转化为基类的成员函数指针还是转化为子类成员函数指针的问题**和**是否可以将一个类的成员函数指针转化为另一个不相关的类的成员函数指针的问题**，人们曾有过很激烈的争论。然而不幸的是，在标准委员会做出决定之前，不同的编译器生产商已经根据自己对这些问题的不同的回答实现了自己的编译器。根据标准（第 5.2.10/9 节），你可以使用 reinterpret_cast 在一个成员函数指针中保存一个与本来的类不相关的类的成员函数。有关成员函数指针转换的问题的最终结果也没有确定下来。你现在所能做的还是像以前那样——将成员函数指针转化为本类的成员函数的指针。在文章的后面我会继续讨论这个问题，因为这正是各个编译器对这样一个标准没有达成共识的一个话题。

在一些编译器中，在基类和子类的成员函数指针之间的转换时常有怪事发生。当涉及到多重继承时，使用 reinterpret_cast 将子类转换成基类时，对某一特定编译器来说有可能通过编译，而也有可能通不过编译，**这取决于在子类的基类列表中的基类的顺序！**下面就是一个例子：

```
class Derived: public Base1, public Base2 // 情况 (a)

class Derived2: public Base2, public Base1 // 情况 (b)

typedef void (Derived::* Derived_mfp)();

typedef void (Derived2::* Derived2_mfp)();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
typedef void (Base1::* Base1mfp) ();

typedef void (Base2::* Base2mfp) ();

Derived_mfp x;
```

对于**情况(a)**，static_cast<Base1mfp>(x)是合法的，而 static_cast<Base2mfp>(x)则是错误的。然而**情况(b)**却与之相反。你只可以安全地将子类的成员函数指针转化为第一个基类的成员函数指针！如果你要实验一下，MSVC 会发出 C4407 号警告，而 Digital Mars C++会出现编译错误。如果用 reinterpret_cast 代替 static_cast，这两个编译器都会发生错误，但是两种编译器对此有着不同的原因。但是一些编译器对此细节置之不理，大家可要小心了！

标准 C++中另一条有趣的规则是：**你可以在类定义之前声明它的成员函数指针。**这对一些编译器会有一些无法预料的副作用。我待会讨论这个问题，现在你只要知道要尽可能得避免这种情况就是了。

需要值得注意的是，就像成员函数指针，标准 C++中同样提供了成员数据指针（member data pointer）。它们具有相同的操作符，而且有一些实现原则也是相同的。它们用在 stl::stable_sort 的一些实现方案中，而对此很多其他的应用我就不再提及了。

成员函数指针的使用

现在你可能会觉得成员函数指针是有些奇异。但它可以用来做什么呢？对此我在网上做了非常广泛的调查。最后我总结出使用成员函数指针的两点原因：

- 用来做例子给 C++初学者看，帮助它们学习语法；或者
- 为了实现“委托（delegate）”！

成员函数指针在 STL 和 Boost 库的单行函数适配器（one-line function adaptor）中的使用是微不足道的，而且允许你将成员函数和标准算法混合使用。但是它们最重要的应用是在不同类型的应用程序框架中，比如它们形成了 MFC 消息系统的核心。

当你使用 MFC 的消息映射宏（比如 ON_COMMAND）时，你会组装一个包含消息 ID 和成员函数指针（型如：CCmdTarget::*成员函数指针）的序列。这是 MFC 类必须继承 CCmdTarget 才可以处理消息的原因之一。但是，各种不同的消息处理函数具有不同的参数列表（比如 OnDraw 处理函数的第一个参数的类型为 CDC *），所以序列中必须包含各种不同类型的成员函数指针。MFC 是怎样做到这一点的呢？MFC **利用了一个可怕的编译器漏洞（hack），它将所有可能出现的成员函数指针放到一个庞大的联合（union）中，从而避免了通常需要进行的 C++类型匹配检查。**（看一下 afximpl.h 和 cmdtarg.cpp 中名为 MessageMapFunctions 的 union，你就会发现这一恐怖的事实。）因为 MFC 有如此重要的一部分代码，所以事实是，所有的编译器都为这个漏洞开了绿灯。（但是，在后面我们会看到，**如果一些类用到了多重继承，这个漏洞在 MSVC 中就不会起作用，这正是在使用 MFC 时只能必须使用单一继承的原因。**）

在 boost::function 中有类似的漏洞（但不是太严重）。看起来如果你想做任何有关成员函数指针的比较有趣的事，你就必须做好与这个语言的漏洞进行挑战的准备。要是你想否定 C++的成员函数指针设计有缺陷的观点，看来是很难的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

在写这篇文章中，我有一点需要指明：**“允许成员函数指针之间进行转换（cast），而不允许在转换完成后调用其中的函数”**，把这个规则纳入 C++的标准中是可笑的。首先，**很多**流行的编译器对这种转换不支持（所以，转换是标准要求的，但不是可移植的）。其次，**所有**的编译器，如果转换成功，调用转换后的成员函数指针时仍然可以实现你预期的功能：那编译器就没有所谓的“ undefined behavior（未定义的行为）”这类错误出现的必要了（调用（Invocation）是可行的，但这不是标准！）。第三，允许转换而不允许调用是完全没有用处的，只有转换和调用都可行，才能方便而有效地实现委托，从而使这种语言受益。

为了让你确信这一具有争议的论断，考虑一下在一个文件中只有下面的一段代码，这段代码是合法的：

```
class SomeClass;

typedef void (SomeClass::* SomeClassFunction)(void);

void Invoke(SomeClass *pClass, SomeClassFunction funcptr) {

(pClass->*funcptr)(); };
```

注意到编译器必须生成汇编代码来调用成员函数指针，其实编译器对 SomeClass 类一无所知。显然，除非链接器进行了一些极端精细的优化措施，否则代码会忽视类的实际定义而能够正确地运行。而这造成的直接后果是，你可以“安全地”调用从完全不同的其他类中转换过来的成员函数指针。

为解释我的断言的另一半——转换并不能按照标准所说的方式进行，我需要在细节上讨论编译器是怎样实现成员函数指针的。我同时会解释为什么使用成员函数指针的规则具有如此严格的限制。获得详细论述成员函数指针的文档不是太容易，并且大家对错误的言论已经习以为常了，所以，我仔细检查了一系列编译器生成的汇编代码……

成员函数指针——为什么那么复杂？

类的成员函数和标准的 C 函数有一些不同。与被显式声明的参数相似，类的成员函数有一个隐藏的参数 this，它指向一个类的实例。根据不同的编译器，this 或者被看作内部的一个正常的参数，或者会被特别对待（比如，在 VC++中，this 一般通过 ECX 寄存器来传递，而普通的成员函数的参数被直接压在堆栈中）。this 作为参数和其他普通的参数有着本质的不同，即使一个成员函数受一个普通函数的支配，在标准 C++中也没有理由使这个成员函数和其他的普通函数（ordinary function）的行为相同，因为没有 thiscall 关键字来保证它使用像普通参数一样正常的调用规则。成员函数是一回事，普通函数是另外一回事（Member functions are from Mars, ordinary functions are from Venus）。

你可能会猜测，一个成员函数指针和一个普通函数指针一样，只是一个代码指针。然而这种猜测也许是错误的。在大多数编译器中，一个成员函数指针要比一个普通的函数指针要大许多。更奇怪的是，在 Visual C++中，一个成员函数指针可以是 4、8、12 甚至 16 个字节长，这取决于它所相关的类的性质，同时也取决于编译器使用了怎样的编译设置！成员函数指针比你想象中的要复杂得多，但也不总是这样。

让我们回到二十世纪 80 年代初期，那时，最古老的 C++编译器 CFront 刚刚开发完成，那时 C++语言只能实现单一继承，而且成员函数指针刚被引入，它们很简单：它们就像普通的函数指针，只是附加了额外的 this 作为它们的第一个参数，你可以将一个成员函数指针转化成一个普通的函数指针，并使你能够对这个额外添加的参数产生足够的重视。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这个田园般的世界随着 CFront 2.0 的问世被击得粉碎。它引入了模版和多重继承，多重继承所带来的破坏造成了成员函数指针的改变。问题在于，随着多重继承，调用之前你不知道使用哪一个父类的 this 指针，比如，你有 4 个类定义如下：

```
class A {

public:

virtual int Afunc() { return 2; };

};

class B {

public:

int Bfunc() { return 3; };

};

// C 是个单一继承类，它只继承于 A

class C: public A {

public:

int Cfunc() { return 4; };

};

// D 类使用了多重继承

class D: public A, public B {

public:

int Dfunc() { return 5; };

};
```

假如我们建立了 C 类的一个成员函数指针。在这个例子中，Afunc 和 Cfunc 都是 C 的成员函数，所以我们的成员函数指针可以指向 Afunc 或者 Cfunc。但是 Afunc 需要一个 this 指针指向 C::A(后面我叫它 Athis)，而 Cfunc 需要一个 this 指针指向 C（后面我叫它 Cthis）。编译器的设计者们为了处理这种情况使用了一个把戏（trick）：他们保证了 A 类在物理上保存在 C 类的头部（即 C 类的起始地址也就是一个 A 类的一个实例的起始地址），这意味着 Athis == Cthis。我们只需担心一个 this 指针就够了，并且对于目前这种情况，所有的问题处理得还可以。

现在，假如我们建立一个 D 类的成员函数指针。在这种情况下，我们的成员函数指针可以指向 Afunc、Bfunc 或 Dfunc。但是 Afunc 需要一个 this 指针指向 D::A，而 Bfunc 需要一个 this 指针指向 D::B。这时，这个把戏就不管用了，我们不可以把 A 类和 B 类都放在 D 类的头部。所以，D 类的一个成员函数指针不仅要说明要指明调用的是哪一个函数，还要指明使用哪一个 this 指针。编译器知道 A 类占用的空间有多大，所以它可以对 Athis 增加一个 delta = sizeof(A)偏移量就可以将 Athis 指针转换为 Bthis 指针。

如果你使用虚拟继承（virtual inheritance），比如虚基类，情况会变得更糟，你可以不必为搞懂这是为什么太伤脑筋。就举个例子来说吧，编译器使用**虚拟函数表**（virtual function table——“ vtable”）来保存每一个虚函数、函数的地址和 virtual_delta：将当前的 this 指针转换为实际函数需要的 this 指针时所要增加的位移量。

综上所述，为了支持一般形式的成员函数指针，你需要至少三条信息：函数的地址，需要增加到 this 指针上的 delta 位移量，和一个虚拟函数表中的索引。对于 MSVC 来说，你需要第四条信息：虚拟函数表（vtable）的地址。

成员函数指针的实现

那么，编译器是怎样实现成员函数指针的呢？这里是对不同的 32、64 和 16 位的编译器，对各种不同的数据类型（有 int、void*数据指针、代码指针（比如指向静态函数的指针）、在单一（single-）继承、多重（multiple-）继承、虚拟（virtual-）继承和未知类型（unknown）的继承下的类的成员函数指针）使用 sizeof 运算符计算所获得的数据：

编译器	选项	int	DataPtr	CodePtr	Single	Multi	Virtual	Unknown
MSVC		4	4	4	4	8	12	16
MSVC	/vmg	4	4	4	16#	16#	16#	16
MSVC	/vmg /vmm	4	4	4	8#	8#	--	8#
Intel_IA32		4	4	4	4	8	12	12
Intel_IA32	/vmg /vmm	4	4	4	4	8	--	8
Intel_Itanium		4	8	8	8	12	20	20
G++		4	4	4	8	8	8	8
Comeau		4	4	4	8	8	8	8
DMC		4	4	4	4	4	4	4
BCC32		4	4	4	12	12	12	12
BCC32	/Vmd	4	4	4	4	8	12	12
WCL386		4	4	4	12	12	12	12
CodeWarrior		4	4	4	12	12	12	12
XLC		4	8	8	20	20	20	20

PDF-XCHANGE Click to buy NOW! www.docu-track.com

DMC	small	2	2	2	2	2	2	2
DMC	medium	2	2	4	4	4	4	4
WCL	small	2	2	2	6	6	6	6
WCL	compact	2	4	2	6	6	6	6
WCL	medium	2	2	4	8	8	8	8
WCL	large	2	4	4	8	8	8	8

注：

表示使用__single/__multi/__virtual_inheritance 关键字的时候代表 4、8 或 12。

这些编译器是 Microsoft Visual C++ 4.0 to 7.1 (.NET 2003), GNU G++ 3.2 (MingW binaries, http://www.mingw.org/), Borland BCB 5.1 (http://www.borland.com/), Open Watcom (WCL) 1.2 (http://www.openwatcom.org/), Digital Mars (DMC) 8.38n (http://www.digitalmars.com/), Intel C++ 8.0 for Windows IA-32, Intel C++ 8.0 for Itanium, (http://www.intel.com/), IBM XLC for AIX (Power, PowerPC), Metrowerks Code Warrior 9.1 for Windows (http://www.metrowerks.com/), 和 Comeau C++ 4.3 (http://www.comeaucomputing.com/). Comeau 的数据是在它支持的 32 位平台（x86, Alpha, SPARC 等）上得出的。16 位的编译器的数据在四种 DOS 配置（tiny, compact, medium, 和 large）下测试得出，用来显示各种不同代码和数据指针的大小。MSVC 在/vmg 的选项下进行了测试，用来显示“成员指针的全部特性”。（如果你拥有在列表中没有出现的编译器，请告知我。非 x86 处理机下的编译器测试结果有独特的价值。）

看着表中的数据，你是不是觉得很惊奇？你可以清楚地看到编写一段在一些环境中可以运行而在另一些编译器中不能运行的代码是很容易的。不同的编译器之间，它们的内部实现显然是有很大差别的；事实上，我认为编译器在实现语言的其他特性上并没有这样明显的差别。对实现的细节进行研究你会发现一些奇怪的问题。

一般，编译器采取最差的，而且一直使用最普通的形式。比如对于下面这个结构：

```
// Borland (缺省设置) 和 Watcom C++.

struct {

FunctionPointer m_func_address;

int m_delta;

int m_vtable_index; //如果不是虚拟继承，这个值为 0。

};
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
// Metrowerks CodeWarrior 使用了稍微有些不同的方式。

//即使在不允许多重继承的 Embedded C++的模式下，它也使用这样的结构！

struct {

int m_delta;

int m_vtable_index; // 如果不是虚拟继承，这个值为-1。

FunctionPointer m_func_address;

};

// 一个早期的 SunCC 版本显然使用了另一种规则：

struct {

int m_vtable_index; //如果是一个非虚拟函数（non-virtual function），这个值为 0。

FunctionPointer m_func_address; //如果是一个虚拟函数（virtual function），这个值为 0。

int m_delta;

};

//下面是微软的编译器在未知继承类型的情况下或者使用/vmg 选项时使用的方法：

struct {

FunctionPointer m_func_address;

int m_delta;

int m_vtordisp;

int m_vtable_index; // 如果不是虚拟继承，这个值为 0

};

// AIX (PowerPC)上 IBM 的 XLC 编译器：

struct {

FunctionPointer m_func_address; // 对 PowerPC 来说是 64 位

int m_vtable_index;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
int m_delta;

int m_vtordisp;

};

// GNU g++使用了一个机灵的方法来进行空间优化

struct {

union {

FunctionPointer m_func_address; // 其值总是 4 的倍数

int m_vtable_index_2; // 其值被 2 除的结果总是奇数

};

int m_delta;

};
```

对于几乎所有的编译器，delta 和 vindex 用来调整传递给函数的 this 指针，比如 Borland 的计算方法是：

```
adjustedthis = *(this + vindex -1) + delta // 如果 vindex!=0

adjustedthis = this + delta // 如果 vindex=0
```

（其中，“ *”是提取该地址中的数值，adjustedthis 是调整后的 this 指针——译者注）

Borland 使用了一个优化方法：如果这个类是单一继承的，编译器就会知道 delta 和 vindex 的值是 0，所以它就可以跳过上面的计算方法。

GNU 编译器使用了一个奇怪的优化方法。可以清楚地看到，对于多重继承来说，你必须查看 vtable（虚拟函数表）以获得 voffset（虚拟函数偏移地址）来计算 this 指针。当你做这些事情的时候，你可能也把函数指针保存在 vtable 中。通过这些工作，编译器将 m_func_address 和 m_vtable_index 合二为一（即放在一个 union 中），编译器区别这两个变量的方法是使函数指针（m_func_address）的值除以 2 后结果为偶数，而虚拟函数表索引(m_vtable_index_2)除以 2 后结果为奇数。它们的计算方法是：

```
adjustedthis = this + delta

if (funcadr & 1) //如果是奇数

call (* ( *delta + (vindex+1)/2) + 4)

else //如果是偶数
```

```
call funcadr
```

（其中， funcadr 是函数地址除以 2 得出的结果。——译者注）

Inter 的 Itanium 编译器（但不是它们的 x86 编译器）对虚拟继承（virtual inheritance）的情况也使用了 unknown_inheritance 结构，所以，一个虚拟继承的指针有 20 字节大小，而不是想象中的 16 字节。

```
// Itanium，unknown 和 virtual inheritance 下的情况.

struct {

FunctionPointer m_func_address; //对 Itanium 来说是 64 位

int m_delta;

int m_vtable_index;

int m_vtordisp;

};
```

我不能保证 Comeau C++使用的是和 GNU 相同的技术，也不能保证它们是否使用 short 代替 int 使这种虚拟函数指针的结构的大小缩小至 8 个字节。最近发布的 Comeau C++版本为了兼容微软的编译器也使用了微软的编译器关键字（我想它也只是忽略这些关键字而不对它们进行实质的相关处理罢了）。

Digital Mars 编译器（即最初的 Zortech C++到后来的 Symantec C++）使用了一种不同的优化方法。对单一继承类来说，一个成员函数指针仅仅是这个函数的地址。但涉及到更复杂的继承时，这个成员函数指针指向一个**形式转换函数（thunk function）**，这个函数可以实现对 this 指针的必要调整并可用来调用实际的成员函数。每当涉及到多重继承的时候，每一个成员函数的指针都会有这样一个形式转换函数，这对函数调用来说是非常有效的。但是这意味着，当使用多重继承的时候，子类的成员函数指针向基类成员函数指针的转换就会不起作用了。可见，这种编译器对编译代码的要求比其他的编译器要严格得多。

很多嵌入式系统的编译器不允许多重继承。这样，这些编译器就避免了可能出现的问题：一个成员函数指针就是一个带有隐藏 this 指针参数的普通函数指针。

微软"smallest for class"方法的问题

微软的编译器使用了和 Borland 相似的优化方法。它们都使单一继承的情况具有最优的效率。但不像 Borland，微软在缺省条件下成员函数指针省略了值为 0 的指针入口（entry），我称这种技术为“ smallest for class”方法：对单一继承类来说，一个成员函数指针仅保存了函数的地址（m_func_address），所以它有 4 字节长。而对于多重继承类来说，由于用到了偏移地址（m_delta），所以它有 8 字节长。对虚拟继承，会用到 12 个字节。这种方法确实节省空间，但也有其它的问题。

首先，将一个成员函数指针在子类和基类之间进行转化会改变指针的大小！因此，信息是会丢失的。其次，当一个成员函数指针在它的类定义之前声明的时候，编译器必须算出要分配给这个指针多少空间，但是这样做是不安全的，因为在定义之前编译器不可能知道这个类的继承方式。对 Intel C++和早期的微软编译器来

PDF-XCHANGE Click to buy NOW! www.docu-track.com

说，编译器仅仅对指针的大小进行猜测，一旦在源文件中猜测错误，你的程序会在运行时莫名其妙地崩溃。所以，微软的编译器中增加了一些保留字：__single_inheritance, __multiple_inheritance,和__virtual_inheritance，并增设了一些编译器开关（compiler switch），如/vmg，让所有的成员函数指针有相同的大小，而对原本个头小的成员函数指针的空余部分用 0 填充。Borland 编译器也增加了一些编译器开关，但没有增加新的关键字。Intel 的编译器可以识别 Microsoft 增加的那些关键字，但它在能够找到类的定义的情况下会对这些关键字不做处理。

对于 MSVC 来说，编译器需要知道类的 vtable 在哪儿；通常就会有一个 this 指针的偏移量（vtordisp），这个值对所有这个类中的成员函数来说是不变的，但对每个类来说会是不同的。对于 MSVC，经调整过的 this 指针是在原 this 指针的基础上经过下面的计算得出的：

```
if (vindex=0) //如果不是虚拟继承（_virtual_inheritance）

adjustedthis = this + delta

else //如果是

adjustedthis = this + delta + vtordisp + *(*(this + vtordisp) + vindex)
```

在虚拟继承的情况下，vtordisp 的值并不保存在__virtual_inheritance 指针中，而是在发现函数调用的代码时，编译器才将其相应的汇编代码“嵌”进去。但是对于未知类型的继承，编译器需要尽可能地通过读代码确定它的继承类型，所以，编译器将虚拟继承指针（virtual inheritance pointer）分为两类（__virtual_inheritance 和__unknown_inheritance）。

理论上，所有的编译器设计者应该在 MFP（成员函数指针）的实现上有所变革和突破。但在实际上，这是行不通的，因为这使现在编写的大量代码都需要改变。微软曾发表了一篇非常古老的文章（http://msdn.microsoft.com/archive/en-us/dnarvc/html/jangrayhood.asp）来解释 Visual C++运作的实现细节。这篇文章是 Jan Gray 写的，他曾在 1990 年设计了 Microsoft C++的对象模型。尽管这篇文章发表于 1994 年，但这篇文章仍然很重要——这意味着 C++的对象模型在长达 15 年的时间里（1990 年到 2004 年）没有丝毫改变。

现在，我想你对成员函数指针的事情已经知道得太多了。要点是什么？我已为你建立了一个规则。虽然各种编译器的在这方面的实现方法有很大的不同，但是也有一些有用的共同点：不管对哪种形式的类，调用一个成员函数指针生成的汇编语言代码是完全相同的。有一种特例是使用了“ smallest for class”技术的非标准的编译器，即使是这种情况，差别也是很微小的。这个事实可以让我们继续探索怎样去建立高性能的委托（delegate）。

委托（delegate）

和成员函数指针不同，你不难发现委托的用处。最重要的，使用委托可以很容易地实现一个 Subject/Observer 设计模式的改进版[GoF, p. 293]。Observer（观察者）模式显然在 GUI 中有很多的应用，但我发现它对应用程序核心的设计也有很大的作用。委托也可用来实现策略（Strategy）[GoF, p. 315]和状态（State）[GoF, p. 305]模式。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

现在，我来说明一个事实，委托和成员函数指针相比并不仅仅是好用，而且比成员函数指针简单得多！既然所有的.NET 语言都实现了委托，你可能会猜想如此高层的概念在汇编代码中并不好实现。但事实并不是这样：委托的实现确实是一个底层的概念，而且就像普通的函数调用一样简单（并且很高效）。一个 C++委托只需要包含一个 this 指针和一个简单的函数指针就够了。当你建立一个委托时，你提供这个委托一个 this 指针，并向它指明需要调用哪一个函数。编译器可以在建立委托时计算出调整 this 指针需要的偏移量。这样在使用委托的时候，编译器就什么事情都不用做了。这一点更好的是，编译器可以在编译时就可以完成全部这些工作，这样的话，委托的处理对编译器来说可以说是微不足道的工作了。在 x86 系统下将委托处理成的汇编代码就应该是这么简单：

```
mov ecx, [this]

call [pfunc]
```

但是，在标准 C++中却不能生成如此高效的代码。 Borland 为了解决委托的问题在它的 C++编译器中加入了一个新的关键字（__closure）,用来通过简洁的语法生成优化的代码。GNU 编译器也对语言进行了扩展，但和 Borland 的编译器不兼容。如果你使用了这两种语言扩展中的一种，你就会限制自己只使用一个厂家的编译器。而如果你仍然遵循标准 C++的规则，你仍然可以实现委托，但实现的委托就不会是那么高效了。

有趣的是，**在 C#和其他.NET 语言中，执行一个委托的时间要比一个函数调用慢 8 倍**（参见 http://msdn.microsoft.com/library/en-us/dndotnet/html/fastmanagedcode.asp）。我猜测这可能是垃圾收集和.NET 安全检查的需要。最近，微软将“统一事件模型（unified event model）”加入到 Visual C++中，随着这个模型的加入，增加了__event、 __raise、__hook、__unhook、event_source 和 event_receiver 等一些关键字。**坦白地说，我对加入的这些特性很反感，因为这是完全不符合标准的，这些语法是丑陋的，因为它们使这种 C++不像 C++，并且会生成一堆执行效率极低的代码。**

解决这个问题的推动力：对高效委托（fast delegate）的迫切需求

使用标准 C++实现委托有一个过度臃肿的症状。大多数的实现方法使用的是同一种思路。这些方法的基本观点是将成员函数指针看成委托&#0;&#0;但这样的指针只能被一个单独的类使用。为了避免这种局限，你需要间接地使用另一种思路：你可以**使用模版为每一个类建立一个“成员函数调用器（member function invoker）”。委托包含了 this 指针和一个指向调用器（invoker）的指针，并且需要在堆上为成员函数调用器分配空间。**

对于这种方案已经有很多种实现，包括在 CodeProject 上的实现方案。各种实现在复杂性上、语法（比如，有的和 C#的语法很接近）上、一般性上有所不同。最具权威的一个实现是 boost::function。最近，它已经被采用作为下一个发布的 C++标准版本中的一部分[Sutter1]。希望它能够被广泛地使用。

就像传统的委托实现方法一样，我同样发觉这种方法并不十分另人满意。虽然它提供了大家所期望的功能，但是会混淆一个潜在的问题：**人们缺乏对一个语言的底层的构造。** “成员函数调用器”的代码对几乎所有的类都是一样的，在所有平台上都出现这种情况是令人沮丧的。毕竟，堆被用上了。但在一些应用场合下，这种新的方法仍然无法被接受。

我做的一个项目是离散事件模拟器，它的核心是一个事件调度程序，用来调用被模拟的对象的成员函数。大多数成员函数非常简单：它们只改变对象的内部状态，有时在事件队列（event queue）中添加将来要发生的事件，在这种情况下最适合使用委托。但是，每一个委托只被调用（invoked）一次。一开始，我使用了

boost::function，但我发现程序运行时，给委托所分配的内存空间占用了整个程序空间的三分之一还要多！“我要真正的委托！”我在内心呼喊着，“真正的委托只需要仅仅两行汇编指令啊！”

我并不能总是能够得到我想要的，但后来我很幸运。我在这儿展示的代码（代码下载链接见*译者注*）几乎在所有编译环境中都产生了优化的汇编代码。最重要的是，**调用一个含有单个目标的委托（single-target delegate）的速度几乎同调用一个普通函数一样快**。实现这样的代码并没有用到什么高深的东西，唯一的遗憾就是，为了实现目标，我的代码和标准 C++的规则有些偏离。我使用了一些有关成员函数指针的未公开知识才使它能够这样工作。如果你很细心，而且不在意在少数情况下的一些编译器相关（compiler-specific）的代码，那么高性能的委托机制在任何 C++编译器下都是可行的。

诀窍：将任何类型的成员函数指针转化为一个标准的形式

我的代码的核心是一个**能够将任何类的指针和任何成员函数指针分别转换为一个通用类的指针和一个通用成员函数的指针的**类。由于 C++没有“通用成员函数（generic member function）”的类型，所以我把所有类型的成员函数都转化为一个在代码中未定义的 CGenericClass 类的成员函数。

大多数编译器对所有的成员函数指针平等地对待，不管他们属于哪个类。所以对这些编译器来说，可以使用 reinterpret_cast 将一个特定的成员函数指针转化为一个通用成员函数指针。事实上，假如编译器不可以，那么这个编译器是不符合标准的。对于一些接近标准（almost-compliant）的编译器，比如 Digital Mars，成员函数指针的 reinterpret_cast 转换一般会涉及到一些额外的特殊代码，当进行转化的成员函数的类之间没有任何关联时，编译器会出错。对这些编译器，我们使用一个名为 horrible_cast 的内联函数（在函数中使用了一个 union 来避免 C++的类型检查）。使用这种方法看来是不可避免的&#0;&#0;boost::function 也用到了这种方法。

对于其他的一些编译器（如 Visual C++, Intel C++和 Borland C++），我们必须将多重（multiple-）继承和虚拟（virtual-）继承类的成员函数指针转化为单一（single-）继承类的函数指针。为了实现这个目的，我巧妙地使用了模板并利用了一个奇妙的戏法。注意，这个戏法的使用是因为这些编译器并不是完全符合标准的，但是使用这个戏法得到了回报：它使这些编译器产生了优化的代码。

既然我们知道编译器是怎样在内部存储成员函数指针的，并且我们知道在问题中应该怎样为成员函数指针调整 this 指针，我们的代码在设置委托时可以自己调整 this 指针。对单一继承类的函数指针，则不需要进行调整；对多重继承，则只需要一次加法就可完成调整；对虚拟继承...就有些麻烦了。但是这样做是管用的，并且在大多数情况下，所有的工作都在编译时完成！

这是最后一个诀窍。我们怎样区分不同的继承类型？并没有官方的方法来让我们区分一个类是多重继承的还是其他类型的继承。但是有一种巧妙的方法，你可以查看我在前面给出了一个列表（见中篇）——对 MSVC，每种继承方式产生的成员函数指针的大小是不同的。所以，我们可以基于成员函数指针的大小使用模版！比如对多重继承类型来说，这只是个简单的计算。而在确定 unknown_inheritance（16 字节）类型的时候，也会采用类似的计算方法。

对于微软和英特尔的编译器中采用不标准 12 字节的虚拟继承类型的指针的情况，我引发了一个编译时错误（compile-time error），因为需要一个特定的运行环境（workaround）。如果你在 MSVC 中使用虚拟继承，要在声明类之前使用 FASTDELEGATEDECLARE 宏。而这个类必须使用 unknown_inheritance（未知继承类型）指针（这相当于一个假定的__unknown_inheritance 关键字）。例如：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
FASTDELEGATEDECLARE(CDerivedClass)

class CDerivedClass : virtual public CBaseClass1, virtual public CBaseClass2 {

// : (etc)

};
```

这个宏和一些常数的声明是在一个隐藏的命名空间中实现的，这样在其他编译器中使用时也是安全的。MSVC（7.0 或更新版本）的另一种方法是在工程中使用/vmg 编译器选项。而 Inter 的编译器对/vmg 编译器选项不起作用，所以你必须在虚拟继承类中使用宏。我的这个代码是因为编译器的 bug 才可以正确运行，你可以查看代码来了解更多细节。而在遵从标准的编译器中不需要注意这么多，况且在任何情况下都不会妨碍 FASTDELEGATEDECLARE 宏的使用。

一旦你将类的对象指针和成员函数指针转化为标准形式，实现单一目标的委托（single-target delegate）就比较容易了（虽然做起来感觉冗长乏味）。你只要为每一种具有不同参数的函数制作相应的模板类就行了。实现其他类型的委托的代码也大都与此相似，只是对参数稍做修改罢了。

这种用非标准方式转换实现的委托还有一个好处，就是委托对象之间可以用等式比较。目前实现的大多数委托无法做到这一点，这使这些委托不能胜任一些特定的任务，比如实现**多播委托**（multi-cast delegates）[Sutter3]。

静态函数作为委托目标（delegate target）

理论上，一个简单的非成员函数（non-member function），或者一个静态成员函数（static member function）可以被作为委托目标（delegate target）。这可以通过将静态函数转换为一个成员函数来实现。我有两种方法实现这一点，两种方法都是通过**使委托指向调用这个静态函数的“调用器（invoker）”的成员函数**的方法来实现的。

第一种方法使用了一个邪恶的方法（evil method）。你可以存储函数指针而不是 this 指针，这样当调用“调用器”的函数时，它将 this 指针转化为一个静态函数指针，并调用这个静态函数。问题是这只是一个戏法，它需要在代码指针和数据指针之间进行转换。在一个系统中代码指针的大小比数据指针大时（比如 DOS 下的编译器使用 medium 内存模式时），这个方法就不管用了。它在目前我知道的所有 32 位和 64 位处理器上是管用的。但是因为这种方法还是不太好，所以仍需要改进。

另一种是一个比较安全的方法（safe method），它是将函数指针作为委托的一个附加成员。委托指向自己的成员函数。当委托被复制的时候，这些自引用（self-reference）必须被转换，而且使“=”和“==”运算符的操作变得复杂。这使委托的大小增至 4 个字节，并增加了代码的复杂性，但这并不影响委托的调用速度。

我已经实现了上述两种方法，两者都有各自的优点：安全的方法保证了运行的可靠性，而邪恶的方法在支持委托的编译器下也可能会产生与此相同的汇编代码。此外，安全的方法可避免我以前讨论的在 MSVC 中使用多重继承和虚拟继承时所出现的问题。我在代码中给出的是“安全的方法”的代码，但是在我给出的代码中“邪恶的方法”会通过下面的代码生效：

```
#define (FASTDELEGATE_USESTATICFUNCTIONHACK)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

多目标委托（multiple-target delegate）及其扩展

使用委托的人可能会想使委托调用多个目标函数，这就是**多目标委托（multiple-target delegate）**，也称作**多播委托（multi-cast delegate）**。实现这种委托不会降低单一目标委托（single-target delegate）的调用效率，这在现实中是可行的。你只需要为一个委托的第二个目标和后来的更多目标在堆上分配空间就可以了，这意味着需要在委托类中添加一个数据指针，用来指向由该委托的目标函数组成的单链表的头部节点。如果委托只有一个目标函数，将这个目标像以前介绍的方法一样保存在委托中就行了。如果一个委托有多个目标函数，那么这些目标都保存在空间动态分配的链表中，如果要调用函数，委托使用一个指针指向一个链表中的目标（成员函数指针）。这样的话，如果委托中只有一个目标，函数调用存储单元的个数为 1；如果有 n（n>0）个目标，则函数调用存储单元的个数为 n+1（因为这时函数指针保存在链表中，会多出一个链表头，所以要再加一——译者注），我认为这样做最合理。

由多播委托引出了一些问题。怎样处理返回值？（是将所有返回值类型捆绑在一起，还是忽略一部分？）如果把同一个目标在一个委托中添加了两次那会发生什么？（是调用同一个目标两次，还是只调用一次，还是作为一个错误处理？）如果你想在委托中删除一个不在其中的目标应该怎么办？（是不管它，还是抛出一个异常？）

最重要的问题是在使用委托时会出现无限循环的情况，比如，A 委托调用一段代码，而在这段代码中调用 B 委托，而在 B 委托调用的一段代码中又会调用 A 委托。很多事件（event）和信号跟踪（signal-slot）系统会有一定的方案来处理这种问题。

为了结束我的这篇文章，我的多播委托的实现方案就需要大家等待了。这可以借鉴其他实现中的方法——允许非空返回类型，允许类型的隐式转换，并使用更简捷的语法结构。如果我有足够的兴趣我会把代码写出来。如果能把我实现的委托和目前流行的某一个事件处理系统结合起来那会是最好不过的事情了（有自愿者吗？）。

本文代码的使用

原代码包括了 FastDelegate 的实现（FastDelegate.h）和一个 demo .cpp 的文件用来展示使用 FastDelegate 的语法。对于使用 MSVC 的读者，你可以建立一个空的控制台应用程序(Console Application)的工程，再把这两个文件添加进去就好了，对于 GNU 的使用者，在命令行输入“ gcc demo.cpp”就可以了。

FastDelegate 可以在任何参数组合下运行，我建议你在尽可能多的编译器下尝试，你在声明委托的时候必须指明参数的个数。在这个程序中最多可以使用 8 个参数，若想进行扩充也是很容易的。代码使用了 fastdelegate 命名空间，在 fastdelegate 命名空间中有一个名为 detail 的内部命名空间。

Fastdelegate 使用构造函数或 bind()可以绑定一个成员函数或一个静态（全局）函数，在默认情况下，绑定的值为 0（空函数）。可以使用“！”操作符判定它是一个空值。

不像用其他方法实现的委托，这个委托支持等式运算符（==, !=）。

下面是 FastDelegateDemo.cpp 的节选，它展示了大多数允许的操作。CBaseClass 是 CDerivedClass 的虚基类。你可以根据这个代码写出更精彩的代码，下面的代码只是说明使用 FastDelegate 的语法：

```
using namespace fastdelegate;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
int main(void)

{

printf("-- FastDelegate demo --\nA no-parameter

delegate is declared using FastDelegate0\n\n");

FastDelegate0 noparameterdelegate(&SimpleVoidFunction);

noparameterdelegate();

//调用委托，这一句调用 SimpleVoidFunction()

printf("\n-- Examples using two-parameter delegates (int, char *) --\n\n");

typedef FastDelegate2 MyDelegate;

MyDelegate funclist[12]; // 委托初始化，其目标为空

CBaseClass a("Base A");

CBaseClass b("Base B");

CDerivedClass d;

CDerivedClass c;

// 绑定一个成员函数

funclist[0].bind(&a, &CBaseClass::SimpleMemberFunction);

//你也可以绑定一个静态（全局）函数

funclist[1].bind(&SimpleStaticFunction);

//绑定静态成员函数

funclist[2].bind(&CBaseClass::StaticMemberFunction);

// 绑定 const 型的成员函数

funclist[3].bind(&a, &CBaseClass::ConstMemberFunction);

// 绑定虚拟成员函数

funclist[4].bind(&b, &CBaseClass::SimpleVirtualFunction);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
// 你可以使用”=”来赋值

funclist[5] = MyDelegate(&CBaseClass::StaticMemberFunction);

funclist[6].bind(&d, &CBaseClass::SimpleVirtualFunction);

//最麻烦的情况是绑定一个抽象虚拟函数（abstract virtual function）

funclist[7].bind(&c, &CDerivedClass::SimpleDerivedFunction);

funclist[8].bind(&c, &COtherClass::TrickyVirtualFunction);

funclist[9] = MakeDelegate(&c, &CDerivedClass::SimpleDerivedFunction);

// 你也可以使用构造函数来绑定

MyDelegate dg(&b, &CBaseClass::SimpleVirtualFunction);

char *msg = "Looking for equal delegate";

for (int i=0; i<12; i++) {

printf("%d :", i);

// 可以使用”==”

if (funclist[i]==dg) { msg = "Found equal delegate"; };

//可以使用”!”来判应一个空委托

if (!funclist[i]) {

printf("Delegate is empty\n");

} else {

// 调用生成的经过优化的汇编代码

funclist[i](i, msg);

};

}

};
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

因为我的代码利用了 C++标准中没有定义的行为，所以我很小心地在很多编译器中做了测试。具有讽刺意味的是，它比许多所谓标准的代码更具有可移植性，因为几乎所有的编译器都不是完全符合标准的。目前，核心代码已成功通过了下列编译器的测试：

- Microsoft Visual C++ 6.0, 7.0 (.NET) and 7.1 (.NET 2003) (including /clr 'managed C++'),
- GNU G++ 3.2 (MingW binaries),
- Borland C++ Builder 5.5.1,
- Digital Mars C++ 8.38 (x86, both 32-bit and 16-bit),
- Intel C++ for Windows 8.0,
- Metroworks CodeWarrior for Windows 9.1 (in both C++ and EC++ modes)

对于 Comeau C++ 4.3 (x86, SPARC, Alpha, Macintosh)，能够成功通过编译，但不能链接和运行。对于 Intel C++ 8.0 for Itanium 能够成功通过编译和链接，但不能运行。

此外，我已对代码在 MSVC 1.5 和 4.0，Open Watcom WCL 1.2 上的运行情况进行了测试，由于这些编译器不支持成员函数模版，所以对这些编译器，代码不能编译成功。对于嵌入式系统不支持模版的限制，需要对代码进行大范围的修改。（这一段是在刚刚更新的原文中添加的——译者注）

而最终的 FastDelegate 并没有进行全面地测试，一个原因是，我有一些使用的编译器的评估版过期了，另一个原因是——我的女儿出生了！如果有足够的兴趣，我会让代码在更多编译器中通过测试。(这一段在刚刚更新的原文中被删去了，因为作者目前几乎完成了全部测试。——译者注)

总结

为了解释一小段代码，我就得为这个语言中具有争议的一部分写这么一篇长长的指南。为了两行汇编代码，就要做如此麻烦的工作。唉~！

我希望我已经澄清了有关成员函数指针和委托的误解。我们可以看到为了实现成员函数指针，各种编译器有着千差万别的方法。我们还可以看到，与流行的观点不同，委托并不复杂，并不是高层结构，事实上它很简单。我希望它能够成为这个语言（标准 C++）中的一部分，而且我们有理由相信目前已被一些编译器支持的委托，在不久的将来会加入到标准 C++的新的版本中（去游说标准委员会！）。

据我所知，以前实现的委托都没有像我在这里为大家展示的 FastDelegate 一样有如此高的性能。我希望我的代码能对你有帮助。如果我有足够的兴趣，我会对代码进行扩展，从而支持多播委托（multi-cast delegate）以及更多类型的委托。我在 CodeProject 上学到了很多，并且这是我第一次为之做出的贡献。

参考文献

[GoF] "Design Patterns: Elements of Reusable Object-Oriented Software", E. Gamma, R. Helm, R. Johnson, and J. Vlissides.

I've looked at dozens of websites while researching this article. Here are a few of the most interesting ones:

我在写这篇文章时查看了很多站点，下面只是最有趣的一些站点：

[Boost] Delegates can be implemented with a combination of boost::function and boost::bind. Boost::signals is one of the most sophisticated event/messaging system available. Most of the boost libraries require a highly standards-conforming compiler. （http://www.boost.org/）

[Loki] Loki provides 'functors' which are delegates with bindable parameters. They are very similar to boost::function. It's likely that Loki will eventually merge with boost.
（http://sourceforge.net/projects/loki-lib）

[Qt] The Qt library includes a Signal/Slot mechanism (i.e., delegates). For this to work, you have to run a special preprocessor on your code before compiling. Performance is very poor, but it works on compilers with very poor template support. （http://doc.trolltech.com/3.0/signalsandslots.html）

[Libsigc++] An event system based on Qt's. It avoids the Qt's special preprocessor, but requires that every target be derived from a base object class (using virtual inheritance - yuck!).
（http://libsigc.sourceforge.net/）

[Hickey]. An old (1994) delegate implementation that avoids memory allocations. Assumes that all pointer-to-member functions are the same size, so it doesn't work on MSVC. There's a helpful discussion of the code here. （http://www.tutok.sk/fastgl/callback.html）

[Haendal]. A website dedicated to function pointers?! Not much detail about member function pointers though. （http://www.function-pointer.org/）

[Sutter1] Generalized function pointers: a discussion of how boost::function has been accepted into the new C++ standard. （http://www.cuj.com/documents/s=8464/cujcexp0308sutter/）

[Sutter2] Generalizing the Observer pattern (essentially, multicast delegates) using std::tr1::function. Discusses the limitations of the failure of boost::function to provide operator ==.

（http://www.cuj.com/documents/s=8840/cujexp0309sutter）

[Sutter3] Herb Sutter's Guru of the Week article on generic callbacks.
（http://www.gotw.ca/gotw/083.htm）

关于作者 Don Clugston

我在澳大利亚的 high-tech startup 工作，是一个物理学家兼软件工程师。目前从事将太阳航空舱的硅质晶体玻璃（CSG）薄膜向市场推广的工作。我从事有关太阳的（solar）研究，平时喜欢做一些软件（用作数学模型、设备控制、离散事件触发器和图象处理等），我最近喜欢使用 STL 和 WTL 写代码。我非常怀念过去的光荣岁月：）而最重要的，我有一个非常可爱的儿子（2002 年 5 月出生）和一个非常年轻的小姐（2004 年 5 月出生）。

“黑暗不会战胜阳光，阳光终究会照亮黑暗。”

译者注

由于本文刚发表不久，作者随时都有可能对文章或代码进行更新，若要浏览作者对本文的最新内容，请访问：

http://www.codeproject.com/cpp/FastDelegate.asp

点击以下链接下载 FastDelegate 的源代码：

http://www.codeproject.com/cpp/FastDelegate/FastDelegate_src.zip