C语言函数调用栈(二)

5 函数调用约定

创建一个栈帧的最重要步骤是主调函数如何向栈中传递函数参数。主调函数必须精确存储这些参数，以便被调函数能够访问到它们。函数通过选择特定的调用约定，来表明其希望以特定方式接收参数。此外，当被调函数完成任务后，调用约定规定先前入栈的参数由主调函数还是被调函数负责清除，以保证程序的栈顶指针完整性。

函数调用约定通常规定如下几方面内容：

1) 函数参数的传递顺序和方式

最常见的参数传递方式是通过堆栈传递。主调函数将参数压入栈中，被调函数以相对于帧基指针的正偏移量来访问栈中的参数。对于有多个参数的函数，调用约定需规定主调函数将参数压栈的顺序(从左至右还是从右至左)。某些调用约定允许使用寄存器传参以提高性能。

2) 栈的维护方式

主调函数将参数压栈后调用被调函数体，返回时需将被压栈的参数全部弹出，以便将栈恢复到调用前的状态。该清栈过程可由主调函数负责完成，也可由被调函数负责完成。

3) 名字修饰(Name-mangling)策略

又称函数名修饰(Decorated Name)规则。编译器在链接时为区分不同函数，对函数名作不同修饰。

若函数之间的调用约定不匹配，可能会产生堆栈异常或链接错误等问题。因此，为了保证程序能正确执行，所有的函数调用均应遵守一致的调用约定。

5.1 常见调用约定

下面分别介绍常见的几种函数调用约定。

1. cdecl调用约定

又称C调用约定，是C/C++编译器默认的函数调用约定。所有非C++成员函数和未使用stdcall或fastcall声明的函数都默认是cdecl方式。函数参数按照从右到左的顺序入栈，函数调用者负责清除栈中的参数，返回值在EAX中。由于每次函数调用都要产生清除(还原)堆栈的代码，故使用cdecl方式编译的程序比使用stdcall方式编译的程序大(后者仅需在被调函数内产生一份清栈代码)。但cdecl调用方式支持可变参数函数(即函数带有可变数目的参数，如printf)，且调用时即使实参和形参数目不符也不会导致堆栈错误。对于C函数，cdecl方式的名字修饰约定是在函数名前添加一个下划线；对于C++函数，除非特别使用extern "C"，C++函数使用不同的名字修饰方式。

> 【扩展阅读】**可变参数函数支持条件**
>
> 若要支持可变参数的函数，则参数应自右向左进栈，并且由主调函数负责清除栈中的参数(参数出栈)。
>
> 首先，参数按照从右向左的顺序压栈，则参数列表最左边(第一个)的参数最接近栈顶位置。所有参数距离帧基指针的偏移量都是常数，而不必关心已入栈的参数数目。只要不定的参数的数目能根据第一个已明确的参数确定，就可使用不定参数。例如printf函数，第一个参数即格式化字符串可作为后继参数指示符。通过它们就可得到后续参数的类型和个数，进而知道所有参数的尺寸。当传递的参数过多时，以帧基指针为基准，获取适当数目的参数，其他忽略即可。若函数参数自左向右进栈，则第一个参数距离栈帧指针的偏移量与已入栈的参数数目有关，需要计算所有参数占用的空间后才能精确定位。当实际传入的参数数目与函数期望接受的参数数目不同时，偏移量计算会出错！
>
> 其次，调用函数将参数压栈，只有它才知道栈中的参数数目和尺寸，因此调用函数可安全地清栈。而被调函数永远也不能事先知道将要传入函数的参数信息，难以对栈顶指针进行调整。
>
> C++为兼容C，仍然支持函数带有可变的参数。但在C++中更好的选择常常是函数多态。

2. stdcall调用约定(微软命名)

Pascal程序缺省调用方式，WinAPI也多采用该调用约定。stdcall调用约定主调函数参数从右向左入栈，除指针或引用类型参数外所有参数采用传值方式传递，由被调函数负责清除栈中的参数，返回值在EAX中。stdcall调用约定仅适用于参数个数固定的函数，因为被调函数清栈时无法精确获知栈上有多少函数参数；而且如果调用时实参和形参数目不符会导致堆栈错误。对于C函数，stdcall名称修饰方式是在函数名字前添加下划线，在函数名字后添加@和函数参数的大小，如_functionname@number。

3. fastcall调用约定

stdcall调用约定的变形，通常使用ECX和EDX寄存器传递前两个DWORD(四字节双字)类型或更少字节的函数参数，其余参数按照从右向左的顺序入栈，被调函数在返回前负责清除栈中的参数，返回值在 EAX 中。因为并不是所有的参数都有压栈操作，所以比stdcall和cdecl快些。编译器使用两个@修饰函数名字，后跟十进制数表示的

函数参数列表大小(字节数)，如@function_name@number。需注意fastcall函数调用约定在不同编译器上可能有不同的实现，比如16位编译器和32位编译器。另外，在使用内嵌汇编代码时，还应注意不能和编译器使用的寄存器有冲突。

4. thiscall调用约定

C++类中的非静态函数必须接收一个指向主调对象的类指针(this指针)，并可能较频繁的使用该指针。主调函数的对象地址必须由调用者提供，并在调用对象非静态成员函数时将对象指针以参数形式传递给被调函数。编译器默认使用thiscall调用约定以高效传递和存储C++类的非静态成员函数的this指针参数。

thiscall调用约定函数参数按照从右向左的顺序入栈。若参数数目固定，则类实例的this指针通过ECX寄存器传递给被调函数，被调函数自身清理堆栈；若参数数目不定，则this指针在所有参数入栈后再入栈，主调函数清理堆栈。thiscall不是C++关键字，故不能使用thiscall声明函数，它只能由编译器使用。

注意，该调用约定特点随编译器不同而不同，g++中thiscall与cdecl基本相同，只是隐式地将this指针当作非静态成员函数的第1个参数，主调函数在调用返回后负责清理栈上参数；而在VC中，this指针存放在%ecx寄存器中，参数从右至左压栈，非静态成员函数负责清理栈上参数。

5. naked call调用约定

对于使用naked call方式声明的函数，编译器不产生保存(prologue)和恢复(epilogue)寄存器的代码，且不能用return返回返回值(只能用内嵌汇编返回结果)，故称naked call。该调用约定用于一些特殊场合，如声明处于非C/C++上下文中的函数，并由程序员自行编写初始化和清栈的内嵌汇编指令。注意，naked call并非类型修饰符，故该调用约定必须与__declspec同时使用，如VC下定义求和函数：

代码示例如下(Windows采用Intel汇编语法，注释符为;)：

```
View Code
```

注意，__declspec是微软关键字，其他系统上可能没有。

6. pascal调用约定

Pascal语言调用约定，参数按照从左至右的顺序入栈。Pascal语言只支持固定参数的函数，参数的类型和数量完全可知，故由被调函数自身清理堆栈。pascal调用约定输出的函数名称无任何修饰且全部大写。

Win3.X(16位)时支持真正的pascal调用约定；而Win9.X(32位)以后pascal约定由stdcall约定代替(以C约定压栈以Pascal约定清栈)。

上述调用约定的主要特点如下表所示：

调用方式	stdcall(Win32)	cdecl	fastcall	thiscall(C++)	
参数压栈顺序	从右至左	从右至左	从右至左，Arg1在ecx，Arg2在edx	从右至左，this指针在ecx	
参数位置	栈	栈	栈 + 寄存器	栈，寄存器ecx	
负责清栈的函数	被调函数	主调函数	被调函数	被调函数	
支持可变参数	否	是	否	否	
函数名字格式	_name@number	_name	@name@number		
参数表开始标识	"@@YG"	"@@YA"	"@@YI"		

注：C++因支撑函数重载、命名空间和成员函数等语法特征，采用更为复杂的名字修饰策略。

C++函数修饰名以"?"开始，后面紧跟函数名、参数表开始标识和按照类型代号拼出的返回值参数表。

例如，函数int Function(char *var1,unsigned long)对应的stdcall修饰名为"?Function@@YGHPADK@Z"。

Windows下可直接在函数声明前添加关键字__stdcall、__cdecl或__fastcall等标识确定函数的调用方式，如int __stdcall func()。Linux下可借用函数attribute 机制，如int __attribute__((__stdcall__)) func()。

代码示例如下：

```
View Code
```

被调函数CalleeFunc分别声明为cdecl、stdcall和fastcall约定时，其汇编代码比较如下表所示：

	cdecl	stdcall	fastcall
主调函数职责	sub $0xc,%esp	sub $0xc,%esp	sub $0x4,%esp

	movl $0x33,0x8(%esp)
	movl $0x22,0x4(%esp)
	movl $0x11,(%esp)
	call 8048354 <CalleeFunc>
		movl $0x33,0x8(%esp)
		movl $0x22,0x4(%esp)
		movl $0x11,(%esp)
		call 8048354 <CalleeFunc>
		sub $0xc,%esp
			movl $0x33,(%e
			mov $0x22,%ec
			mov $0x11,%ec
			call 8048354 <
			sub $0x4,%esp

被调函数职责	push %ebp
	mov %esp,%ebp
	mov 0xc(%ebp),%eax
	add 0x8(%ebp),%eax
	add 0x10(%ebp),%eax
	pop %ebp
	ret
		push %ebp
		mov %esp,%ebp
		mov 0xc(%ebp),%eax
		add 0x8(%ebp),%eax
		add 0x10(%ebp),%eax
		pop %ebp
		ret $0xc
		//执行ret指令并清理参数占用的堆栈(栈顶指针上移参数个数*4=12个字节，以释放压栈的参数)
			push %ebp
			mov %esp,%eb
			sub $0x8,%esp
			mov %ecx,0xfff
			mov %edx,0xfff
			mov 0xfffffff8(%
			add 0xfffffffc(%
			add 0x8(%ebp)
			leave
			ret $0x4
			//ret <压栈参数字两个，则ret指令不被压栈

5.2 调用约定影响

当函数导出被其他程序员所使用(如库函数)时，该函数应遵循主要的调用约定，以便于程序员使用。若函数仅供内部使用，则其调用约定可只被使用该函数的程序所了解。

在多语言混合编程(包括A语言中使用B语言开发的第三方库)时，若函数的原型声明和函数体定义不一致或调用函数时声明了不同的函数约定，将可能导致严重问题(如堆栈被破坏)。

以Delphi调用C函数为例。Delphi函数缺省采用stdcall调用约定，而C函数缺省采用cdecl调用约定。一般将C函数声明为stdcall约定，如：int __stdcall add(int a, int b);

在Delphi中调用该函数时也应声明为stdcall约定：

```
View Code
```

不同编译器产生栈帧的方式不尽相同，主调函数不一定能正常完成清栈工作；而被调函数必然能自己完成正常清栈，因此，在跨(开发)平台调用中，通常使用stdcall调用约定(不少WinApi均采用该约定)。

此外，主调函数和被调函数所在模块采用相同的调用约定，但分别使用C++和C语法编译时，会出现链接错误(报告被调函数未定义)。这是因为两种语言的函数名字修饰规则不同，解决方式是使用extern "C"告知主调函数所在模块：被调函数是C语言编译的。采用C语言编译的库应考虑到使用该库的程序可能是C++程序(使用C++编译器)，通常应这样声明头文件：

```
View Code
```

这样C++编译器就会按照C语言修饰策略链接Func函数名，而不会出现找不到函数的链接错误。

5.3 x86函数参数传递方法

x86处理器ABI规范中规定，所有传递给被调函数的参数都通过堆栈来完成，其压栈顺序是以函数参数从右到左的顺序。当向被调函数传递参数时，所有参数最后形成一个数组。由于采用从右到左的压栈顺序，数组中参数的顺序(下标0~N-1)与函数参数声明顺序(Para1~N)一致。因此，在函数中若知道第一个参数地址和各参数占用字节数，就可通过访问数组的方式去访问每个参数。

5.3.1 整型和指针参数的传递

整型参数与指针参数的传递方式相同，因为在32位x86处理器上整型与指针大小相同(均为四字节)。下表给出这两种类型的参数在栈帧中的位置关系。注意，该表基于tail函数的栈帧。

调用语句	参数	栈帧地址
tail(1, 2, 3, (void *)0);	1	8(%ebp)
	2	12(%ebp)

	3	16(%ebp)
	(void *)0	20(%ebp)

5.3.2 浮点参数的传递

浮点参数的传递与整型类似，区别在于参数大小。x86处理器中浮点类型占8个字节，因此在栈中也需要占用8个字节。下表给出浮点参数在栈帧中的位置关系。图中，调用tail函数的第一个和第三个参数均为浮点类型，因此需各占用8个字节，三个参数共占用20个字节。表中word类型的大小是4字节。

调用语句	参数	栈帧地址
tail(1.414, 2, 3.998e10);	word 0: 1.414	8(%ebp)
	word 1: 1.414	12(%ebp)
	2	16(%ebp)
	word 0: 3.998e10	20(%ebp)
	word 1: 3.998e10	24(%ebp)

5.3.3 结构体和联合体参数的传递

结构体和联合体参数的传递与整型、浮点参数类似，只是其占用字节大小视数据结构的定义不同而异。x86处理器上栈宽是4字节，故结构体在栈上所占用的字节数为4的倍数。编译器会对结构体进行适当的填充以使得结构体大小满足4字节对齐的要求。

对于一些RISC处理器(如PowerPC)，其参数传递并不是全部通过栈来实现。PowerPC处理器寄存器中，R3～R10共8个寄存器用于传递整型或指针参数，F1～F8共8个寄存器用于传递浮点参数。当所需传递的参数少于8个时，不需要用到栈。结构体和long double参数的传递通过指针来完成，这与x86处理器完全不同。PowerPC的ABI规范中规定，结构体的传递采用指针方式，而不是像x86处理器那样将结构从一个函数栈帧中拷贝到另一个函数栈帧中，显然x86处理器的方式更低效。可见，PowerPC程序中，函数参数采用指向结构体的指针(而非结构体)并不能提高效率，不过通常这是良好的编程习惯。

5.4 x86函数返回值传递方法

函数返回值可通过寄存器传递。当被调用函数需要返回结果给调用函数时：

1) 若返回值不超过4字节(如int、short、char、指针等类型)，通常将其保存在EAX寄存器中，调用方通过读取EAX获取返回值。

2) 若返回值大于4字节而小于8字节(如long long或_int64类型)，则通过EAX+EDX寄存器联合返回，其中EDX保存返回值高4字节，EAX保存返回值低4字节。

3) 若返回值为浮点类型(如float和double)，则通过专用的协处理器浮点数寄存器栈的栈顶返回。

4) 若返回值为结构体或联合体，则主调函数向被调函数传递一个额外参数，该参数指向将要保存返回值的地址。即函数调用foo(p1, p2)被转化为foo(&p0, p1, p2)，以引用型参数形式传回返回值。具体步骤可能为：a.主调函数将显式的实参逆序入栈；b.将接收返回值的结构体变量地址作为隐藏参数入栈(若未定义该接收变量，则在栈上额外开辟空间作为接收返回值的临时变量)；c. 被调函数将待返回数据拷贝到隐藏参数所指向的内存地址，并将该地址存入%eax寄存器。因此，在被调函数中完成返回值的赋值工作。

注意，函数如何传递结构体或联合体返回值依赖于具体实现。不同编译器、平台、调用约定甚至编译参数下可能采用不同的实现方法。如VC6编译器对于不超过8字节的小结构体，会通过EAX+EDX寄存器返回。而对于超过8字节的大结构体，主调函数在栈上分配用于接收返回值的临时结构体，并将地址通过栈传递给被调函数；被调函数根据返回值地址设置返回值(拷贝操作)；调用返回后主调函数根据需要，再将返回值赋值给需要的临时变量(二次拷贝)。实际使用中为提高效率，通常将结构体指针作为实参传递给被调函数以接收返回值。

5) 不要返回指向栈内存的指针，如返回被调函数内局部变量地址(包括局部数组名)。因为函数返回后，其栈帧空间被"释放"，原栈帧内分配的局部变量空间的内容是不稳定和不被保证的。

函数返回值通过寄存器传递，无需空间分配等操作，故返回值的代价很低。基于此原因，C89规范中约定，不写明返回值类型的函数，返回值类型默认为int。但这会带来类型安全隐患，如函数定义时返回值为浮点数，而函数未声明或声明时未指明返回值类型，则调用时默认从寄存器EAX(而不是浮点数寄存器)中获取返回值，导致错误！因此在C++中，不写明返回值类型的函数返回值类型为void，表示不返回值。

> 【扩展阅读】**GCC返回结构体和联合体**
>
> 通常GCC被配置为使用与目标系统一致的函数调用约定。这通过机器描述宏来实现。但是，在一些目标机上采用不同方式返回结构体和联合体的值。因此，使用PCC编译的返回这些类型的函数不能被使用GCC编译的代码调

用，反之亦然。但这并未造成麻烦，因为很少有Unix库函数返回结构体或联合体。

GCC代码使用存放int或double类型返回值的寄存器来返回1、2、4或8个字节的结构体和联合体(GCC通常还将此类变量分配在寄存器中)。其它大小的结构体和联合体在返回时，将其存放在一个由调用者传递的地址中(通常在寄存器中)。

相比之下，PCC在大多目标机上返回任何大小的结构体和联合体时，都将数据复制到一个静态存储区域，再将该地址当作指针值返回。调用者必须将数据从那个内存区域复制到需要的地方。这比GCC使用的方法要慢，而且不可重入。

在一些目标机上(如RISC机器和80386)，标准的系统约定是将返回值的地址传给子程序。在这些机器上，当使用这种约定方法时，GCC被配置为与标准编译器兼容。这可能会对于1，2，4或8字节的结构体不兼容。

GCC使用系统的标准约定来传递参数。在一些机器上，前几个参数通过寄存器传递；在另一些机器上，所有的参数都通过栈传递。原本可在所有机器上都使用寄存器来传递参数，而且此法还可能显著提高性能。但这样就与使用标准约定的代码完全不兼容。所以这种改变只在将GCC作为系统唯一的C编译器时才实用。当拥有一套完整的GNU 系统，能够用GCC来编译库时，可在特定机器上实现寄存器参数传递。

在一些机器上(特别是SPARC)，一些类型的参数通过“隐匿引用”(invisible reference)来传递。这意味着值存储在内存中，将值的内存地址传给子程序。