其中不可缺少的有：调用约定，参数传递方式，函数调用方式等

1：**函数的调用方式**

基本上就是jmp、call、ret或者他们的变种而已

乍一看，有好多个“%”符号，还记得2.2.1节里讲的AT&T汇编语法吗？这就是那里面说——引用寄存器的时候要在前面加“%”符号。

还有一些汇编指令的后缀，如：“l”、“q”。“l”的意思是双字（long型），“q”的意思是四字（64位寄存器的后缀就是这个）。

如果您仔细观察，是不是会发现有些寄存器rbp，rsp等，感觉会跟ebp和esp有关系呢？答对了，esp寄存器是32位寄存器，而rsp寄存器是64位寄存器。这是Intel对寄存器的一种向下继承性，从最开始一字节的al，ah，到两字节的ax（16位），四字节的eax（32位），再到八字节的rax（64位），寄存器的长度在不断的扩展，对于相关指令的使用，也从“b”、“l”，“q”，也是不断的向下继承或扩展。

这里有一条指令leaveq，它等效于 movq %rbp, %rsp; popq %rbp;

callq 400474 这句的意思就是跳转到test函数里执行。其实汇编调用C函数就这么简单，如果把这条callq指令改成jmpq指令也是可以的。这要从call和jmp的区别上说起，call会把在其之后的那条指令的地址压入栈，在上面反汇编后的代码中，就是0000000000400499，然后再跳转到test函数里执行。而jmpq就不会把地址0000000000400499压入栈中。当函数执行完毕，调用retq指令返回的时候，会把栈中的返回地址弹出到rip寄存器中，这样就返回到main函数中继续执行了。

实现jmpq代替callq的伪代码如下所示：

```
pushq    $0x0000000000400499   jmpq     400474 <test>
```

对于callq 400474 这条指令也可以使用retq来实现。它的实现原理是：指令retq会将栈中的返回地址弹出，并放入到rip寄存器中，然后处理器从rip寄存器所指的地址内取指令后继续执行。根据这个原理，可以先将返回地址0000000000400499压入栈中。然后再将test函数的入口地址0000000000400474压入栈中，接着使用retq指令，以调用返回的形式，从main函数“返回”到test函数中。

实现retq代替callq的伪代码如下所示：

```
pushq $0x0000000000400499 pushq $0x0000000000400474 retq
```

2：**调用约定**

- stdcall

1.在进行函数调用的时候，函数的参数是从右向左依次放入栈中的。

如：

```
int function (int first, int second)
```

这个函数的参数入栈顺序，首先是参数second，然后是参数first。

2.函数的栈平衡操作是由被调用函数执行的，使用的指令是 retn X，X表示参数占用的字节数，CPU在ret之后自动弹出X个字节的堆栈空间。例如上面的function函数，当我们把function的函数参数压入栈中后，当function函数执行完毕后，由function函数负责将传递给它的参数first和second从栈中弹出来。

3.在函数名的前面用下划线修饰，在函数名的后面由@来修饰，并加上栈需要的字节数。如上面的function函数，会被编译器转换为_function@8。

- cdecl

1.在进行函数调用的时候，和stdcall一样，函数的参数是从右向左依次放入栈中的。
2.函数的栈平衡操作是由调用函数执行的，这点是与stdcall不同之处。stdcall使用retn X平衡栈，cdecl则使用leave、pop、增加栈指针寄存器的数据等方法平衡栈。
3.每一个调用它的函数都包含有清空栈的代码，所以编译产生的可执行文件会比调用stdcall约定产生的文件大。
cdecl是GCC的默认调用约定。但是，GCC在x64位系统环境下，使用寄存器作为函数调用的参数。按照从左向右的顺序，头六个整型参数放在寄存器RDI, RSI, RDX, RCX, R8和R9上，同时XMM0到XMM7用来放置浮点变元，返回值保存在RAX中，并且由调用者负责平衡栈。

- fastcall

1.函数调用约定规定，函数的参数在可能的情况下使用寄存器传递参数，通常是前两个DWORD类型的参数或较小的参数使用ECX和EDX寄存器传递，其余参数按照从右向左的顺序入栈。
2.函数的栈平衡操作是由被调用函数在返回之前负责清除栈中的参数。
还有很多调用规则，如：thiscall、naked call、pascal等，有兴趣的读者可以自己去研究一下

3: **参数传递方式**
函数参数的传递方式无外乎两种，一种是通过寄存器传递，另一种是通过内存传递。

- 寄存器传递

寄存器传递就是将函数的参数放到寄存器里传递，而不是放到栈里传递。这样的好处主要是执行速度快，编译后生成的代码量少。但只有少部分调用规定默认是通过寄存器传递参数，大部分编译器是需要特殊指定使用寄存器传递参数的。
在X86体系结构下，系统调用一般会使用寄存器传递，由于作者看过的内核种类有限，也不能确定所有的内核都是这么处理的，但是Linux内核肯定是这么做的。因为应用程序的执行空间和系统内核的执行空间是不一样的，如果想从应用层把参数传递到内核层的话，最方便快捷的方法是通过寄存器传递参数，否则需要使用很大的周折才能把数据传递过去，原因会在以后的章节中详细讲述。

- 内存传递

内存传递参数很好理解，在大多数情况下参数传递都是通过内存入栈的形式实现的。
在X86体系结构下的Linux内核中，中断或异常的处理会使用内存传递参数。因为，在中断产生后，到中断处理的上半部，中间的过渡代码是用汇编实现的。汇编跳转到C语言的过程中，C语言是用堆栈保存参数的，为了无缝衔接，汇编就需要把参数压入栈中，然后再跳转到C语言实现的中断处理程序中。
以上这些都是在X86体系结构下的参数传递方式，在X64体系结构下，大部分编译器都使用的是寄存器传递参数。因此，内存传递和寄存器传递的区别就不太重要了。