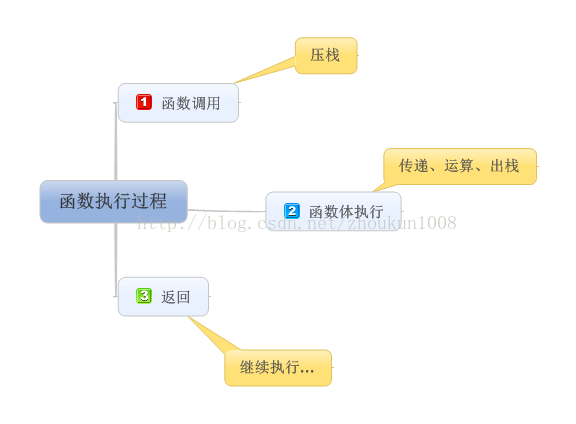
   当调用一个函数时，整个调用过程分为三步进行，第一步是函数的调用，第二步是函数体的执行，第三步是返回，即返回到函数调用表达式的位置。



第一步：函数调用

    1、将函数调用语句下一条语句的地址保存到在栈中，以便函数调用完成后返回。（将函数放到栈空间中称为压栈）。

     2、对实参表从后向前，一次计算出实参的值，并且将值压栈。

     3、跳转到函数体处。  
第二步：函数体执行

    4、如果函数体中定义了变量，将变量压栈

    5、将每一个形参以栈中对应的实参值取代，执行函数体的功能体。

6、将函数体中的变量、保存到栈中的实参值，依次从栈中取出，释放栈空间（出栈）。

第三步：返回

7、返回过程执行的是函数体中的return语句。其过程是从栈中取出刚开始调用函数时压入的地址，跳转到函数的下一条语句。当return语句不带有表达式时，按照保存的地址返回，当return语句带有表达式时，将计算出的return表达式的值保存起来，然后再返回。

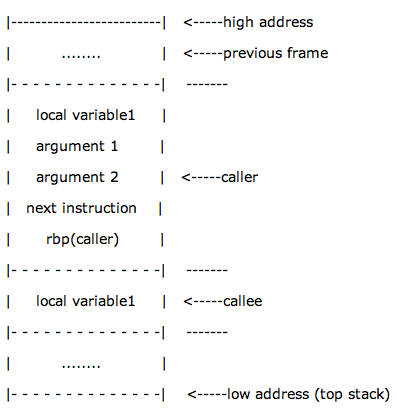
现在从代码层面去分析:

**栈(stack)**

栈，相信大家都十分熟悉，push/pop，只允许在一端进行操作，后进先出(LIFO)，凡是学过编程的人都能列出一二三点。但就是这个最简单的数据结构，构成了计算机中程序执行的基础，用于内核中程序执行的栈具有以下特点：

* 每一个进程在用户态对应一个调用栈结构(call stack)
* 程序中每一个未完成运行的函数对应一个栈帧(stack frame)，栈帧中保存函数局部变量、传递给被调函数的参数等信息
* 栈底对应高地址，栈顶对应低地址，栈由内存高地址向低地址生长

一个进程的调用栈图示如下：

* 

**寄存器(register)**

寄存器位于CPU内部，用于存放程序执行中用到的数据和指令，CPU从寄存器中取数据，相比从内存中取快得多。寄存器又分通用寄存器和特殊寄存器。

通用寄存器有ax/bx/cx/dx/di/si，尽管这些寄存器在大多数指令中可以任意选用，但也有一些规定某些指令只能用某个特定“通用”寄存器，例如函数返回时需将返回值mov到ax寄存器中；特殊寄存器有bp/sp/ip等，特殊寄存器均有特定用途，例如sp寄存器用于存放以上提到的栈帧的栈顶地址，除此之外，不用于存放局部变量，或其他用途。

对于有特定用途的几个寄存器，简要介绍如下：

* **ax(accumulator)**: 可用于存放函数返回值
* **bp(base pointer)**: 用于存放执行中的函数对应的栈帧的栈底地址
* **sp(stack poinger)**: 用于存放执行中的函数对应的栈帧的栈顶地址
* **ip(instruction pointer)**: 指向当前执行指令的下一条指令
* 不同架构的CPU，寄存器名称被添以不同前缀以指示寄存器的大小。例如对于x86架构，字母“e”用作名称前缀，指示各寄存器大小为32位；对于x86\_64寄存器，字母“r”用作名称前缀，指示各寄存器大小为64位。

**函数调用例子**

了解了栈和寄存器的概念，下面看一个函数调用实例：

//func\_call.c

int bar(int c, int d)

{

int e = c + d;

return e;

}

int foo(int a, int b)

{

return bar(a, b);

}

int main(void)

{

foo(2, 5);

return 0

}

反汇编分析:

下面我们使用gdb对func\_call进行反汇编，跟踪main->foo->bar函数调用过程。

# gdb func\_call

//此处省略gdb版本信息

Reading symbols from /tmp/lx/func\_call...done.

(gdb) start

Temporary breakpoint 1 at 0x400525: file func\_call.c, line 14.

Starting program: /tmp/lx/func\_call

Temporary breakpoint 1, main () at func\_call.c:14

foo(2, 5);

(gdb)

main函数：

现进程跑在main函数中，我们disassemble命令显示当前函数的汇编信息：

(gdb) disassemble /rm

Dump of assembler code for function main:

{

0x0000000000400521 <main+0>: 55 push %rbp

0x0000000000400522 <main+1>: 48 89 e5 mov %rsp,%rbp

foo(2, 5);

0x0000000000400525 <main+4>: be 05 00 00 00 mov $0x5,%esi

0x000000000040052a <main+9>: bf 02 00 00 00 mov $0x2,%edi

0x000000000040052f <main+14>: e8 d2 ff ff ff callq 0x400506 <foo>

return 0;

0x0000000000400534 <main+19>: b8 00 00 00 00 mov $0x0,%eax

}

0x0000000000400539 <main+24>: c9 leaveq

0x000000000040053a <main+25>: c3 retq

End of assembler dump.

disassemble命令的/m指示显示汇编指令的同时，显示相应的程序源码；/r指示显示十六进制的计算机指令(raw instruction)。

以上输出每行指示一条汇编指令，除程序源码外共有四列，各列含义为：

1. **0x0000000000400521**: 该指令对应的虚拟内存地址
2. **<main+0>**: 该指令的虚拟内存地址偏移量
3. **55**: 该指令对应的计算机指令
4. **push %rbp**: 汇编指令

一个函数被调用，首先默认要完成以下动作：

* 将调用函数的栈帧栈底地址入栈，即将bp寄存器的值压入调用栈中
* 建立新的栈帧，将被调函数的栈帧栈底地址放入bp寄存器中

以下两条指令即完成上面动作：

0x0000000000400521 <main+0>: 55 push %rbp

0x0000000000400522 <main+1>: 48 89 e5 mov %rsp,%rbp

也许你会问：咦？以上disassemble的输出不是main函数的汇编指令吗，怎么输出中也有上面两条指令？难道main也是一个“被调函数”？

是的，皆因main并不是程序拉起后第一个被执行的函数，它被\_start函数调用，更详细的资料参看[这里](http://learn.akae.cn/media/ch19s02.html)。

一个函数调用另一个函数，需先将参数准备好。main调用foo函数，两个参数传入通用寄存器中：

mov $0x5, %esi

mov $0x2, %edi

对于参数传递的方式，x86和x86\_64定义了不同的函数调用规约(calling convention)。相比x86\_64将参数传入通用寄存器的方式，x86将参数压入调用栈中，x86下对应foo函数传参的汇编指令，有以下形式的输出：

sub $0x8, %esp

mov $0x5, -0x4(%ebp)

mov $0x2, -0x8(%ebp)

参数的调用栈位置通过ebp保存的栈帧栈底地址索引，栈从内存高地址向低地址生长，所以索引值为负数，减少esp寄存器的值表示扩展栈帧。

万事具备，是时候将执行控制权交给foo函数了，call指令完成交接任务：

0x000000000040052f <main+14>: e8 d2 ff ff ff callq 0x400506 <foo>

参数的调用栈位置通过ebp保存的栈帧栈底地址索引，栈从内存高地址向低地址生长，所以索引值为负数，减少esp寄存器的值表示扩展栈帧。

一条call指令，完成了两个任务：

1. 将调用函数(main)中的下一条指令(这里为0x400534)入栈，被调函数返回后将取这条指令继续执行，64位rsp寄存器的值减8
2. 修改指令指针寄存器rip的值，使其指向被调函数(foo)的执行位置，这里为0x400506

执行完start命令后，现在程序停在0x400522的位置，下面我们通过gdb的si指令，让程序执行完call指令：

(gdb) **si 3**

foo (a=0, b=4195328) at func\_call.c:8

8 {

(gdb)

此时我们再来看rsp、rbp寄存器的值，它们保存了程序实际用到的物理内存地址：

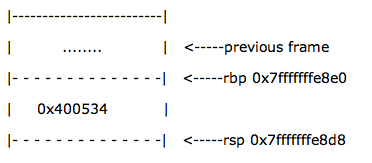
(gdb) **info registers rbp rsp**

rbp 0x7fffffffe8e0 0x7fffffffe8e0

rsp 0x7fffffffe8d8 0x7fffffffe8d8

(gdb)

main函数君的执行到此就暂时告一段落了，此时func\_call的调用栈情况如下：



相关寄存器信息如下：

esi: 0x5   edi: 0x2

**foo函数**

foo函数被执行之后，我们使用disassemble命令显示其汇编指令：

(gdb) disassemble /rm

Dump of assembler code for function foo:

11 {

0x00000000004004f0 <+0>: 55 push %rbp

0x00000000004004f1 <+1>: 48 89 e5 mov %rsp,%rbp

0x00000000004004f4 <+4>: 48 83 ec 08 sub $0x8,%rsp

0x00000000004004f8 <+8>: 89 7d fc mov %edi,-0x4(%rbp)

0x00000000004004fb <+11>: 89 75 f8 mov %esi,-0x8(%rbp)

12 return bar(a, b);

=> 0x00000000004004fe <+14>: 8b 55 f8 mov -0x8(%rbp),%edx

0x0000000000400501 <+17>: 8b 45 fc mov -0x4(%rbp),%eax

0x0000000000400504 <+20>: 89 d6 mov %edx,%esi

0x0000000000400506 <+22>: 89 c7 mov %eax,%edi

0x0000000000400508 <+24>: e8 c9 ff ff ff callq 0x4004d6 <bar>

13 }

0x000000000040050d <+29>: c9 leaveq

0x000000000040050e <+30>: c3 retq

End of assembler dump.

前面两条指令将main函数栈帧的栈底地址入栈，建立foo函数的栈帧。接着的三条指令扩展栈帧，将传入的参数存为函数内局部变量。最后三条指令与bar函数调用相对应，也是先将参数传入esi、edi寄存器，然后执行call指令。

继续执行si命令，让程序执行到call指令的位置：