人人都能学的会C++协程原理剖析与自我实现

导语

> 本文将会深入讨论C/C++函数调用过程，执行过程，返回过程，这些是协程实现的基础，涉及部分汇编代码设计与分析，只有从汇编层次出发，才能揭示根本的原理，让我们不再停留表面。阅读本文之前先抛出一些相关问题，函数怎么提前返回？怎么手动模拟函数调用？函数参数到底怎么传？函数如何能跨调用层次返回？怎么能获取EIP/RIP？函数怎么跳转到特定地址执行？函数怎么保存上下文?汇编中为什么要保存寄存器？汇编中怎么恢复栈？汇编怎么平衡栈？什么是__stdcall和__cdecl？...如果这些问题你都很透彻了，肯定相信也能轻松理解协程，不用看这篇文章，也可以自己写出协程了。

本文是很久以前研究的，当时研究了boost库的协程，最近又翻了翻，发现都快忘光了，想想还是记录下，作为一篇笔记，如果能帮助别人，那再好不过了。本身boost的协程库跨平台，设计的也不错，结合C++模板，封装的比较深，易用难懂，想直接研究明白，有一定的门槛。想了想如果拿boost做教程，可能不够友好， 为了揭示协程原理，手写了两个简单实现DEMO，一个是Windows下X86无独立栈协程，一个是的提取boost的fiber汇编切换代码有独立栈协程。以前我也写了个clang跨平台的，处理不同平台也加了一些宏，复杂些，还不适合教程，所以为了做教程，周末我也花了一天多写了x86协程，写汇编切换部分相对费时间。我们掌握原理后，再看boost版本，就轻松易懂了，也能体会大佬设计的巧妙。

Demo工程代码： 有栈与无栈协程DEMO

协程是什么?

协程不是线程，更不是进程，可以理解为当前线程中带上下文的子函数调用，好像概念比较抽象，我们再细化一些。对比线程来说，线程是由操作系统控制切换的，有独立上下文，协程切换是自己控制，也有独立上下文。

这些都是概念上的，不直观。从应用上说，比如释放一个技能，技能过程要有5s的动作动画播放后，再执行爆炸动画，整个过程是一个task。当然实现这个需求的方法有多种，我们做比较死的限定，要用类似C语言的顺程过程来执行，不要引入状态机或者其它技术段。那么执行这个task要等5s左右检查动作动画是否播放结束，这会阻塞住线程，效率很低，如果是带交互的程序，可能就被用户强关了。试想如果这个task放到新开工作线程中，好像也可以，但这就带来线程切换开销，如果这样的task很多，开了上千个线程，线程切换开销很大，就为检查一些特效是不是播放完，的确有点大材小用了，这时协程就派上场，可以理解为轻量级的小线程，当要等5s时，我直接退出，不阻塞，5s后我如果能再回到离开的状态，继续执行，模拟了线程，好像可以哇！协程也常用在网络与资源加载相关操作中。

协程的大致原理

上面小例子，实际有两点要求，一个函数离开时能记住离开的位置，在windows 平台下，也就是要能记住EIP/RIP寄存器值，因为EIP/RIP指示了下一条指令要执行的地址。一个是再次进入，能恢复现场，也要要能保存当时函数栈内存，寄存器的值，恢复栈内存，恢复寄存器的值，后面我们会详细说，也会手动完成这个过程，这要求我们对函数调用有深入的理解。

C#下的yield return体验

C++20之前没有原生的协程，没有关系，C#有关键字有类似的功能，我们先看看C#的一段yield return示例代码，先结合运行结果，我们好进一步分析与推理，再过渡到C++层设计。

```
using System;
using System.Collections.Generic;

class Program
{
    public static void Main(string[] args)
    {
        IEnumerable<int> Source(int max)
        {
            int first = 1, second = 1;
            Console.WriteLine($"Source Show:{first} ");
            yield return first;
            Console.WriteLine($"Source Show:{second} ");
            yield return second;
            for (int i = 0; i < max; ++i)
            {
                int third = first + second;
                first = second;
```

```
                second = third;
                Console.WriteLine($"Source Show:{third} ");
                yield return third;
            }
        }

        int iTimes = 0;
        foreach (int i in Source(8))
        {
            Console.WriteLine($"Main Show:{i},Time:{++iTimes}");
        }
        Console.WriteLine();
    }
}
```

看一下程序输出的结果：

C#下的分析与推导

从运行结果上看，明显Source的函数和foreach的执行语句交替执行，并且Source并不是一次全部执行完，每遇到yield return关键字就返回了，且下次调用时，还能接着从Source返回地方执行，有兴趣的可以打打断点，调试一下具体过程。

也就是说C#做了两件事，一件记录了函数退出时地址，下次调用可以直接到这个地址，一件是能能保存和恢复退出的现场，下次调用时可能恢复这个现场。是这个意思，只不过说的比较笼统，且这些底层都不是我们自己控制的，我们并太清楚编译器到底怎么实现的？我们下面转战到C++，来模拟这个过程，如果我们能模拟成功，说明我们真的理解了，并也提供了一种方案。

C++协程实现必用知识讲解

函数的执行环境

函数的环境我的理解分为4个部分，分为可以执行的二进制代码，运行所需寄存器，运行所需栈内存，运行可能所需堆内存。

- 二进制代码由编译器对生成，编译后就是固定的，在二进制文件的.text段，加载到内存的只读代码区，理论上不可以修改，但这也不是绝对不能修改的，还有一些动态指令，超出我们讨论范围之外。对于这部分，我们其实不用关心，反正程序编译后，代码段加载到内存后，在x86的CPU平台下由**EIP寄存器所指向下一条指令的内存地址**，在X64平台是RIP寄存器，只是64位的，本质不变。对于我们的**协程来说，就是EIP/RIP关联的内存地址我们要手动记录与切换**，具体怎么手动记录与切换EIP/RIP的值，后面会细说。

- 寄存器，这部分非常重要，也是我们原来学习汇编和编写汇编代码常打交道的地方，除了原来普通寄存器，随着CPU的升级，各种的专用的寄存器与指令也应运而生，比如SSE指令集，使用XMM相关，ARM的NEON加速指令，x86平台下我们今天不讨论这些。对于我们协程来说，我们只关心三类，**ESP，EBP操作和记录堆栈相关的，EAX，EBX，ECX等通用寄存器**，操作指令的具体运算，传参，返回等，以及上面说到**EIP的控制执行指令**，对于X64实际差不多，就是多了几个寄存器，我们协程要在离开函数与恢复函数时，要正确还原原来寄存器的值，但不一定所有，根据intel i386的ABI调用约定，有的寄存器值本来就可以改变。

- 栈内存，函数运行离不开栈内存，简单来说，栈内存了函数的参数，函数的返回地址，函数的要保护的寄存器，函数的局部变量，下图就是经典的函数栈帧结构，其它资料也有很多示意图。参数不一定都压栈，看调用约定。对于我们协程来说，我们肯定记录栈内存了，比如我们函数的局部变量做了修改，下次再调用一定要是修改后的值，这是就涉及两大主要模式，**一种有独立栈协程，一种无独立栈协程**，其它变种先不讨论。

- 堆内存，是函数执行的过程中，可能要动态分配一些堆内存，如new，malloc分配出来的，这个实际对于我们协程初步理解，还不用太关心。

综上：我们对于函数执行环境更多关心的第2条寄存器与第3条栈内存，这也是我们后续继续讨的。

函数的传参

函数的传参可以用**纯用栈来传，也可以在参数少的情况下直接用寄存器传参，在多的情况，寄存器加栈一起，至于何种方式传参，传参顺序，用那些那种寄存器，只是一种约定，也就fastcall，stdcall，cdecl等。**如windows的32平台下，eax常用来作返回什，ecx用作this指针,edx作为第二个参数，linux的32位下依次用用ebx,ecx,edx，但64位下就是RDI、RSI、RDX、RCX、R8、R9；arm64平台依次用x0,x1,x2...x7。实际上至于用什么寄存器传，也是看约定的，不是绝对的，当我们对汇编有很好的撑控能力后，的确可以不遵守，但前提要保证正确。

函数的返回值

函数的返回值，**一般来说用eax/rax/x0作返回，但也不是必然的。**一个函数也有借用其它寄存器作返回值，如果edx，xmm0。再有就是函数返回值比如是一个结构体，**一个寄存器大小无法返回，可能会把返回值地址作隐藏的参数传递**，后面说到boost的协程切换函数，就是这样的。

函数的调用与返回

从c/c++层面说函数调用与返回，实际没有太多花样。我们从汇编层面说一下，一般调用都用***call functionA***的形式，返回用ret的形式。首先这里面有一些省略，有一些等价过程，这些等价过程，对我们协程来说不但非常的重要，还要使用这些知识点，这里还涉及栈的push与pop,我们有必要细说一些。

理解Push ,Pop 指令的等效过程

```
push eax;
// 等同两条伪指令
1.sub esp,4;    // 将esp减4，因为栈向下生长的，伪指令，仅供理解
2.mov esp,eax; // 将eax 放到esp中，伪指令，仅供理解
```

```
pop eax;
// 等同两条伪指令
1.add esp,4;     // 将esp加4，因为栈向下生长的，伪指令，仅供理解
2.mov eax,esp; // 将esp 放到eax中，伪指令，仅供理解
```

深入理解call,jmp,ret指令的等效过程

```
call 内存/立即数/寄储器;
//1.原下个EIP对应指令地址入栈;(esp-4)
//2.修改EIP为新的;
//3.跳转EIP执行;

jmp 内存/立即数/寄储器;
//1.修改EIP为新的;
//2.跳转EIP执行; 注:无条件执行，不改变栈

ret;
//1.修改EIP为新的栈顶数据;
//2.esp +4;(恢复栈)
//3.跳转EIP执行;
```

有了这些知识，我们可以不用call指令来调用一个函数，完全来模拟函数调用，只需要手动的将函数要返回的下条指令所对应EIP的值压入栈，然后分配栈内存，手动的jmp到我们想跳转的函数地址。这里不太理解也没有关系，后面例子会有用到。返回时我们手动返回到父函数，也可以直接返回到父函数的父函数，只要我们掌握了原理，后面例子也会用的到。

获取EIP的值

获取EIP的值对我们来说非常的重要，因为有了EIP值,我们就可以知道向什么地址强行跳转，这样才能满足协程中手动切换。有很多种方法来获取，不用汇编，clang下系统有直接提供好的API

```
infoPtr->reRIP = (uint64_t)__builtin_return_address(0);
```

好像windows也有，但记不清，我们可以自己用汇编写，也有用C语言写的版本，根源是上面提到的call的等效过程，会将函数返回值的eip压到栈中

```
__declspec(naked) int __stdcall CoroutineGetRIP()
{
        __asm
        {
                mov eax, [esp]
                ret
        }
}
```

也可以用汇编中标签来随时获取，这个也很重要，我用过加一定偏移，需要提前知道机器码长度，强行加指令的大小。

```
__asm
{
    call NEXT
    NEXT:
    pop eax
}
```

不过到我们协程中，这个过程就比较隐蔽了，尤其是boost的那个汇编版本，但核心原理是一样的。 上面就是必了解的内容，多加练习，会对这些有更深的理解，下面我们正式开始

X86平台下的无独立栈协程

原理与C++代码的设计

有了C#的那个例子，我们先写个类似的函数RunPrintfTestCanBreak，函数遇到COROUTINE_YIELD能够返回并保存现场，再次调用时RunPrintfTestCanBreak时，能够恢复现场，接着执行，对于本例，也就是局部变量i,a的值要能正确的恢复，正确的打印。

```
void RunPrintfTestCanBreak(void* pParam)
{
    int i = 0;
    int a = 100;
    for(i = 0;i < 10;i++)
    {
        a += 10;
            std::cout << "I am run,Times:" << i << ",the a is:"<< a << std::
        COROUTINE_YIELD;
    }
    COROUTINE_END;
}
```

有了上面C++必备知识，我们可以写个结构体来描述函数协程环境，然后函数传入这个环境就可以了，嗯，是个不错主意，先给出代码。

```
//rayhutnerli
//2021/4/17


#ifndef FUNINFO_H_
#define FUNINFO_H_

#include <stdint.h>
#include <string.h>

#pragma pack(1)

typedef void (*RegCallFun)(void*);

// now we just deal x86 version
// struct FunEnvInfo
typedef struct _FunEnvInfo
{
    _FunEnvInfo()
```

```
    {
        memset(this,0,sizeof(_FunEnvInfo));
    }
    int reEIP; // 0
    int reESP; // 4
    int reEBP; // 8
    int reEAX; // 12
    int reEBX; // 16
    int reECX; // 20
    int reEDX; // 24
    int reESI; // 28
    int reEDI; // 32
    int reEFLAG; // 36
    int iStackSize; // 40
    void* pStackMem; // 44
    RegCallFun pfCallFun; // 48
}FunEnvInfo,*FunEnvInfoPtr;

#pragma pack()

#endif
```

我们这里FunEnvInfo保存了x86的常见寄存器，并给出了偏移地址，同时可以看出，暂时我们不处理XMM相关的寄存器。实际根据Intel i386 ABI调用约定，eax作为返回值，也基本不用处理，除非父函数有使用eax，要保护一下，我这里演示虽然保留了，但实际没有处理。iStackSize是表示函数栈内存的大小，pStackMem表示栈内存要拷贝的副本内存，是一个手动malloc分配的堆内存，pfCallFun表示协程函数，我们会注册用，每次注册会生成一个实例，函数可以相同。

我们的原理：**当协程函数退出时，FunEnvInfo里面寄存器保存协程函数当时退出时的寄存器值，用malloc分配的堆内存pStackMem来保存协程函数退出的栈内存，记录退出时EIP的值；当退出的协程函数再次调用时，我们并不切换栈，只是将手动的保存的栈内存拷贝回来，寄存器恢复，并跳转到协程函数退出时EIP的指令内存地址。**

我们说一下与协程函数交互的核心代码，先贴出函数定义，这三个函数都是我用汇编实现的，每个都会详细讲解。其中CoroutineExec用来**调用和恢复协程函数**，也可以理解为简单的调度器，CoroutineBreak用来从**协程函数退出，做保存协程函数上下文**，CoroutineEnd退出当前协程，做一些**内存清理与标记**，这三个函数重新宏定义了一下，COROUTINE_YIELD和COROUTINE_END插入到协程函数，使其看起来更像C#的yield写法。

```
//rayhutnerli
//2021/4/17
```

```
#ifndef CORE_H_
#define CORE_H_

#include "funenv.h"


void __stdcall CoroutineExec(FunEnvInfoPtr infoPtr);
void __stdcall CoroutineBreak(FunEnvInfoPtr infoPtr);
void __stdcall CoroutineEnd(FunEnvInfoPtr infoPtr);


#define COROUTINE_RUN(env) CoroutineExec(env);
#define COROUTINE_YIELD CoroutineBreak((FunEnvInfoPtr)pParam)
#define COROUTINE_END CoroutineEnd((FunEnvInfoPtr)pParam)


#endif
```

谈__declspec(naked)，__stdcall，__cdecl关键字

看一下CoroutineExec的定义，里面引入了关键字

```
__declspec(naked) void __stdcall CoroutineExec(FunEnvInfoPtr infoPtr)
{
        // don't believe MSVC compile,it always assumes you use ebp
        // I've been cheated many times,rayhunterli
        // ex: mov eax, infoPtr
    // It may be translated incorrectly
    __asm
    {
        //...
    }
}
```

__declspec(naked) 表示告诉MSVC编译器，这个函数是我完全自定义，我有能力有信心处理好汇编级函数实现，编译生成obj时不要自动加任何汇编指令代码。包括push ebp;mov ebp,esp;这样保存调用栈,ret函数返回值这样的约定，都不要给我加。如果我漏写，可能是我故意的，也可能我真的书写bug，充分信任我，出问题自己负责。

小提示：我多次遇到__declspec(naked)如果不写最原生的汇编，结合带有C++语言结合汇编，它经常会出现错误的释译机器指令，一定要反汇编看看，并去掉符号。它默认你是保存栈帧的，实际上有时我不保存，我觉得我的需求不用。

__stdcall 表示函数调用规则，用栈传递参数，从右向左传递参数，由**被调用者自己清理栈内存**， 我这里主要用到这点，因为有了这点，我们可以保证函数返回EIP不是平衡栈内存的指令，看起来干净些，不过这也无所谓，只要维护好栈，一样的。

__cdecl和__stdcall基本相同，只是栈的清理是刚好相反，__cdecl是调用者自己清理，这也是普通代码默认的。

拆解协程函数的调用与重入设计

下面正式进入关键函数的分析，一点一点，详细分析,主要完成对协程函数的调用，分为第一次和非第一次调用，第一次调用比较简单，非第一次涉及前面说的模拟函数调用过程，我拆成了6个小步。

> （关于汇编书写先做一点歉意：昨天我写这个汇编时间，对于函数参数FunEnvInfo的成员访问，我本来写成的更规范的方法，假设参数infoPtr我已经放到eax寄存器，比如访问[eax+48]，就是访问infoPtr->pfCallFun,我直接用的+48偏移，理论应该用更标准易读的形式[eax]FunEnvInfo.pfCallFun; 比如下面指令cmp [eax], 0，我应该写成cmp [eax]FunEnvInfo.reEIP, 0有更好易读性，大家下载源码后，如果研究，可以要对照FunEnvInfo结构体偏移处理了，非常抱歉。主要当时我nake函数，没有处理ebp，有时微软翻译会不符合我的想法，当时直接用数字了。现在版本已经修正）~~

```
// compare last save eip empty
mov eax, [esp + 4]
cmp [eax]FunEnvInfo.reEIP, 0

// protect some use regs
push ebp
push esi
push edi
push ecx
push ebx

// compare
je FIRSTEXCFUNCTION
```

根据前面提到内容汇编**call __stdcall**函数原理，esp保存的是函数返回的EIP值，esp+4保存的就是函数参数，本文就FunEnvInfoPtr infoPtr，是我们协程的工作环境。下面一条就是比较infoPtr->reEIP是否为0，因为reEIP偏移值为零，所以直接[eax]就可以。根据我们的设计如果为0，说明这个协程是第一次执行，跳转到FIRSTEXCFUNCTION标签地址，不为0，说明这个协程并不是第一次执行。

下面的push 几个寄存器，实际就是为了保护父函数这些寄存器的值，因为我们CoroutineExec和里面子函数调用，会修改这几个寄存器值，如果不保护，退出时，就可能直接修改掉了父函数的，这是不正确的设计。

首调协程函数

```
FIRSTEXCFUNCTION:
// prepare function param and function addresss
push eax
mov eax, [eax]FunEnvInfo.pfCallFun

// call the function
call eax
```

这里代码比较简单，先用push传参infoPtr，然后根据结构体偏移，取出协程函数infoPtr->pfCallFun，然后直接call,完成第一次对协程函数的调用，这段汇编代码相当于C++中

```
(*infoPtr->pfCallFun)(infoPtr);
```

重入协程函数6步走

下面也就是我们进入非第一次进入协程函数的设计，前面我们是直接call的infoPtr->pfCallFun，现在不同了，我们有函数的执行环境，我们有它的栈内存，它的寄存器，我们要模拟这个过程。我拆解为6步，一步一步的分解。

1. 第一步我们准备函数的返回值地址EIP，也就是说这个协程函数执行结束后我们让这个函数退到那里，当然是退到同正常call结束的地方，那么我们需要push一个eip值到栈里面，好解决，汇编中设记一个标签EXEC_RET就可以了。整个过程就模拟传参数，然给返回地址，返回地址就用标签EXEC_RET取一下放到ecx中，ok我们已经做好了call的过程。

```
// here,we jump the fuction again
// 1. prepare function param and return address
push eax
```

```
mov ecx, dword ptr[EXEC_RET]
push ecx
```

2. 正常函数调用都要保存调用栈，也就是保存ebp数据，这个简单，毕竟我们知道调用约定的流程，按照标准的流程处理即可

```
// 2.setup ebp
push ebp
mov ebp, esp
```

3. 准备完调用栈，我们就要准备调用栈了，毕竟一个函数需要多少栈内存，编译器是知道的，我们这里怎么办，没关系，我们第一次的时间，已经保存了这个大小，直接取infoPtr->iStackSize的值就可以了，然后将栈内存向后移动就这个数值就可以了。

```
// 3.calculate stack size
sub esp, [eax]FunEnvInfo.iStackSize
```

4. 准备好栈内存，我们需要将上次的栈内存拷贝回来就可了，拷贝比较简单，就是数据移动，对于汇编指令来说，设置好esi，edi，ecx就可以了，我们在前面已经备好了这三个寄存器，不用担心我们修改它了。

```
// 4.copy memory
mov edi, esp
mov esi, [eax]FunEnvInfo.pStackMem
mov ecx, [eax]FunEnvInfo.iStackSize
rep movsb
```

5. 准备好栈内存，我们需要恢复原来协程的寄存器，这个也比较简单，就是数据移动回来，轻松完成。根据前面说的，我们做教程，暂时处理ebx,ecx,edx,esi,edi就可以了

```
// 5.setup common regs
mov ebx, [eax]FunEnvInfo.reEBX
mov ecx, [eax]FunEnvInfo.reECX
mov edx, [eax]FunEnvInfo.reEDX
mov esi, [eax]FunEnvInfo.reESI
mov edi, [eax]FunEnvInfo.reEDI
```

6. 做好前面5步，我们万事具备，只欠东风，准备跳转到上次协程函数退出时EIP地址就可以了,经过这6步，基于我们对调用过程的理解，我们没有借助call指令，完全完成了模

拟函数调用，且完成函数的现场的恢复。

```
// 6.jum EIP
mov eax, [eax]FunEnvInfo.reEIP
jmp eax
```

做完第一次和非第一次的恢复，我们唯一没有处理的就是从协程函数返回的处理，这个也是前面提到EXEC_RET:标签的地方，看一下怎么处理。

```
EXEC_RET :
// We have to balance the stack

// fucntion praam
add esp, 4

// save change
pop ebx
pop ecx
pop edi
pop esi
pop ebp
// call address and param
ret 4;
```

我们解析一下，这几行汇编，第一步esp加4，是因为我们协程函数都规定有一个指针参数，是协程函数的执行环境，我们是push传入，那么我们要自然要平衡，加4就可以了。下面pop指令就平衡原来说的父函数的几个我们可能修改的寄存器，保证和开始的push是成对调用就可了。到于最后的ret 4;是因为我们是stdcall，除了返回地址外，我们还要平衡压入的参数，所以一定ret 4才能平衡。

到此我们就完成CoroutineExec的汇编解析，主要就是要能对函数调用的深入理解，理解透后，一切迎刃而解。

拆解协程函数保存上下文设计

当执行CoroutineBreak时，协程函数要中途退出，我们就要保存好函数的现场与下次要执行的EIP值，就算OK了。

```
// save some parent function regs
// we must save ebx,edx,esi,edi
mov eax, [esp + 4]
```

```
mov [eax]FunEnvInfo.reESP, esp
mov [eax]FunEnvInfo.reEBP, ebp
mov [eax]FunEnvInfo.reEBX, ebx
mov [eax]FunEnvInfo.reECX, ecx
mov [eax]FunEnvInfo.reEDX, edx
mov [eax]FunEnvInfo.reESI, esi
mov [eax]FunEnvInfo.reEDI, edi

// use esi instead of eax,eax maybe use as function return value
mov esi, eax
```

我们将父函数的寄存器值保存到协程函数的寄存器对应的环境中，这些都是根据FunEnvInfo的偏移值，一一对应即可。因为后面我们要进行函数调用，占用eax，所以我们不能一直用eax来存infoPtr的指针，交换给esi即可。

```
// get the eip and save
// it will be jump to eip when the function is called next time
mov ecx, [esp]
mov [esi]FunEnvInfo.reEIP, ecx
```

我们要获取EIP的值，根据call调用原则，这里我们也没有改写ESP的值，直接将esp所保存内容到到infoPtr->reEIP中即可，这一步完成EIP的保存。

完成了EIP和寄存器的保存，我们就要检测是不要保存栈内存，如果从来没有保存过栈内存，我们需要申请一块堆内存作为保存栈内存的副本，如果已经保存过栈内存，我们只需直接拷贝就可以了。

```
// skip the function ret address and param temp and save esp
mov ecx, [esi]FunEnvInfo.reESP
// we must add 8,4 ret,4 param
add ecx, 8
mov [esi]FunEnvInfo.reESP, ecx

// calculate stack size of parent function and iStackSize
mov edx, [esi]FunEnvInfo.reEBP
sub edx,ecx
mov [esi]FunEnvInfo.iStackSize, edx
```

这里我解释一下，我们先是对esp进行加8操作，是因为对于__stdcall调用，我们的协程函数先是push参数，然后push了函数的返回值，所以我们加8表示完全退到父函数的栈内存中，紧接着就是ebp-esp，可以准计算出父函数栈内存的大小，并保存到infoPtr->iStackSize中。

```
// if the parent function stack is greater than 0,
// allocate memory to save
test edx,edx
jle ERROR_PARENT_STACK_SIZE

// If memory has been allocated,
// it will not be allocated again
mov eax, [esi]FunEnvInfo.pStackMem
test eax, eax
jne HAS_ALLOCATED_MEMROY
```

接着我们就来判断infoPtr->iStackSize是否小于等于0，如果是，证明不需要栈内存，我们可以直接进行相关处理返回，如果不是，那么我们要对栈内存进行检测，查一下是否已经申请过堆内存，如果没有申请过，证明是第一次来到保存，不然就是非第一次，只需要保存栈内存就可以，而不需要申请堆内存来备份。

```
// allocate memory
push edx
push ecx
push edx
call dword ptr[malloc]
add esp, 4
mov [esi]FunEnvInfo.pStackMem, eax
pop ecx
pop edx
```

这里是malloc申请堆内存来保存栈内存，内存的大小放入edx，申请后在eax，将eax放入到infoPtr->pStackMem即可。

到这里我们保存寄存器的值，保存了EIP，对于没有申请堆内存，我们也进行了堆内存的申请，那么就要内存的拷贝了，将协程函数的栈内存保存到申请的堆内存中。

```
// memcpy the parent stack memory to new allocated memory
HAS_ALLOCATED_MEMROY:
push esi
mov edi, eax
mov esi, ecx
mov ecx, edx
rep movsb
pop esi
```

完成了这些我们也该进行协程函数的退出了

```
// we need to jump back to the coroutineexec function
// not the parent function
ERROR_PARENT_STACK_SIZE:
// calculate parent stack size,
//skip the self function return address 4 bytes
mov eax, [esi]FunEnvInfo.iStackSize
add eax, 8
// balance the stack
add esp, eax

// get the caller's parent caller
pop ebp
ret
```

我们不需只从CoroutineBreak退出，我们直接跳到父函数的父数，就是coroutineexec中。那么也就将协程函数栈都跳过，再加上CoroutineBreak函数的参数与返回值，我们还要再加上8，那么我们再进行__cdecl返回，pop ebp与ret就可以回到了coroutineexec中，深入理解了这点，我们就充分完成对函数执行过程的控制。到此CoroutineBreak的拆解完成。

拆解协程函数退出设计

这个函数比较简单，协程函数退出时，如果申请过堆内存，将其释放就可以了，看一下完整个汇编代码，判断infoPtr->pStackMem是否为空，不为空，就free掉其内存，然后将FunEnvInfo的iStackSize，pStackMem，reEIP置为0，这里一定要用dword ptr修飾，不然可能只是处理一个字节。

```
__declspec(naked) void __stdcall CoroutineEnd(FunEnvInfoPtr infoPtr)
{
        // don't believe MSVC compile,it always assumes you use ebp
        // I've been cheated many times,rayhunterli
        // ex: mov eax, infoPtr
        // It may be translated incorrectly
        __asm
        {
                // compare whether we need to delete the stack memory
                push esi
                mov esi, [esp + 8]
                mov eax, [esi]FunEnvInfo.pStackMem
                test eax, eax
                je DO_NOT_DELETE
```

```
                // delete the memory
                push edx
                push ecx
                push eax
                call dword ptr[free]
                add esp, 4
                pop ecx
                pop edx



                // clean up some function's FunEnvInfoPtr params
                // we must use dword ptr for number
                mov dword ptr[esi]FunEnvInfo.pStackMem, 0
                mov dword ptr[esi]FunEnvInfo.iStackSize, 0
                mov dword ptr[esi]FunEnvInfo.reEIP, 0

                DO_NOT_DELETE:
                pop esi
                ret 4;
        }
}
```

到此我们完成了win32下无独立栈的协程设计，相信看到这里的朋友肯定对此比较有兴趣，全部代码工程，前面已经提过，放到github中了。下面我们会介绍X64下独立栈内存的协程设计。

X64平台下基于boost库的fiber独立栈协程

原理与C++代码的设计

有了前面的知识的铺垫，我们进行X64平台下有独立栈协程就容易多了，这个协程实际是基于boost的Fiber协程的汇编切换，但我又不想用全部。本想全面介绍boost的协程，发现本文篇副已经够长了。boost封装的比较好，毕况别人是经典，代码中用了很多模板，一下理解起来有点绕，下次深入文章再介绍吧，我们只取精华，完成我们独立栈协程即可。本例是在windows平台X64下，使用VS2019编译，注意VS2019的x64不支持内嵌汇编，可以支持纯汇编，但要做一点点设定，设定过程可以参考我前面的设文章。

Visual Studio 2019 x64 C++ 编译与调用纯汇编

有了C++的那个协程，我们先写个类似的函数testfun，函数遇到COROUTINE_YIELD能够返回并保存现场，再次调用时testfun时，能够恢复现场，接着执行，对于本例，也就是局部变量actemp[512]的值要能正确的恢复，正确的打印。

```
void testfun(transfer t)
{
        int actemp[512] = { 0 };
        actemp[100] = 100;
        std::cout << "testfun:run point 1->a100:" << actemp[100] << std::end
        actemp[100] = 22;
        COROUTINE_YIELD;
        std::cout << "testfun:run point 2->a100:" << actemp[100] << std::end
        COROUTINE_YIELD;
        actemp[100] = 2111;
        std::cout << "testfun:run point 3->a100:" << actemp[100] << std::end
        actemp[100] = 27222;
        COROUTINE_YIELD;
        std::cout << "testfun:run end->a100:" << actemp[100] << std::endl;
        COROUTINE_END;
}
```

同理我们也要对协程的执行环境进行设计，但这次就有点抽像了，或者说毕竟是封装过一层了。

```
typedef void (*coroutinefunc)(transfer);

struct FuncRecord
{
        FuncRecord() :
                pkFunc(0),
                iFuncIndex(0),
                pkStackMem(nullptr),
                iStatckSize(0),
                iUseStatckSize(0),
                pkUseStack(nullptr),
                pkCorContext(nullptr) {}
        FuncRecord(coroutinefunc _pf,size_t _iFuncIndex,
                        void* _pkMem,size_t _iSize) :
                pkFunc(_pf),
                iFuncIndex(_iFuncIndex),
                pkStackMem(_pkMem),
                iStatckSize(_iSize),
```

```
                iUseStatckSize(0),
                pkUseStack(nullptr),
                pkCorContext(nullptr) {}

        void* pkStackMem; // sp memory
        size_t iStatckSize; // sp size
        coroutinefunc pkFunc;
        size_t iFuncIndex;
        void* pkUseStack;
        size_t iUseStatckSize; // use size
        corcontext pkCorContext;
};
```

我来解释一下成员的意义，pkStackMem表示栈内存，但这个栈内存是独立栈，也就协程的执行环境会切到这个栈中，而不像非独立栈协程，只是保存副本，这里没有副本这个概念了。iStatckSize表示这个栈内存的大小。pkFunc表示协程函数，用来注册的协程函数。pkUseStack表示使用栈内存地址，iUseStatckSize表示使用栈内存大小，pkCorContext表示协程的上下文地址，基于boost库的，这个需要后面读汇编。

我们的原理：**每当注册一个协程函数时，创建一块堆内存，将这块堆内存做为协程的独立内存，目前而言也就是要浪费点内存换来独立，稍后字节对齐后，我们记录一下FuncRecord信信息，将在这块内存上创立协程执行的环境，这块内存保存协程运行时上下文和RIP，和线程的主环境进行动态切换，满足协程的切换与交互，这一切封装的更彻底。**

我们看一下和boost的汇编函数交互声明，我并没有取boost全部汇编,他们为了不同的平台设计太多了，实际我们只是windows x64平台，只用jump_fcontext与make_fcontext两个纯汇编函数就够了，它们分别位于JumpContext.asm和MakeContext.asm中。我们用extern "C"声明出来，只是为了方便其它地方调用，准备函数的调用方式方法，同时告编译器我们存在这个签名的函数，编译时请放心，链接时按照这个格式链接就可以了。

```
// asm.h
// just for asm code function declare
// create by rayhunterli
// 2021/4/5

#ifndef MYSIMPLECOROUTINE_ASM_H_
#define MYSIMPLECOROUTINE_ASM_H_


typedef void* corcontext;
struct transfer
{
        corcontext fcont;
```

```
        void* data;
};
typedef void (*MakeFun)(transfer);

extern "C" transfer jump_fcontext(corcontext const to ,void* vp);
extern "C" corcontext make_fcontext(void* sp,long long size,MakeFun fn);


#endif
```

我们再看一下协程调度的声明，基本等于win32的设定，也是有个容器来保存注册的函数，有了这些我们下面可以仔细讲一下汇编层面的设计，毕况C++层面的都比较简单，没有什么难度。

```
class coroutine
{
public:
        coroutine();
        ~coroutine();
        void UpdateFun();

        void RegeisterFun(coroutinefunc pf);
        void EndFuncInstance(transfer t);
        void BreakFuncInstance(transfer t);

        const bool IsEmpty() const { return m_kFunRecordMap.size() == 0; }

private:
        char* AllocMemory(size_t iSize);
private:
        const int m_iDefaultStackMemorySize;
        size_t m_iCurrentFuncIndex;
        size_t m_iCreateFuncIndex;
        std::map<size_t,FuncRecord> m_kFunRecordMap;
public:

};

extern coroutine g_kFunMgr;

#define COROUTINE_YIELD g_kFunMgr.BreakFuncInstance(t)
#define COROUTINE_END g_kFunMgr.EndFuncInstance(t)
```

不过我们先看一下怎么注册协程函数的

```
void coroutine::RegeisterFun(coroutinefunc pf)
{
	size_t iStackSize = m_iDefaultStackMemorySize;
	char* pkMem = AllocMemory(iStackSize);
	if (pkMem == nullptr)
	{
		return;
	}

	FuncRecord rec(pf, ++m_iCreateFuncIndex, (void*)pkMem, iStackSize);

	char* pkMemTop = pkMem + iStackSize;
	void* pkStackTop = reinterpret_cast<void*>(reinterpret_cast<uintptr_
	& ~static_cast<uintptr_t>(0xff));
	pkStackTop = reinterpret_cast<void*>(reinterpret_cast<uintptr_t>(pkS
	- static_cast<uintptr_t>(64));
	size_t size = (reinterpret_cast<uintptr_t>(pkStackTop) - reinterpret

	rec.pkUseStack = pkStackTop;
	rec.iUseStatckSize = size;
	rec.pkCorContext = make_fcontext(pkStackTop, size, pf);

	m_kFunRecordMap[m_iCreateFuncIndex] = rec;
}
```

从上面代码可以看出，先是直接分配内存，m_iDefaultStackMemorySize是128kb，那么本例每个协程先分配128kb，至于不够用需要动态规划，暂时不在我们第一期讨论范围之内。然后进行FuncRecord记录，再后面进行字节对齐操作，在调用make_fcontext这个汇编函数后，放入map容器中，也没有太多复杂的。

至于运行主要就靠jump_fcontext进行上下文切换，我们看一下C++代码，只不过是BreakFuncInstance保存返回值，EndFuncInstance没有保存返回值，并将协程函数的上下文设置为空，表示结束。

```
void coroutine::BreakFuncInstance(transfer t)
{
	// change from  virtual stack context to main context
	auto iter = m_kFunRecordMap.find(m_iCurrentFuncIndex);
	if (iter != m_kFunRecordMap.end())
	{
```

```
                // t.fcont is main context,
        // but the iter->second.pkCorContext is coroutine context
                iter->second.pkCorContext = jump_fcontext(t.fcont, nullptr).
        }

}

void coroutine::EndFuncInstance(transfer t)
{
        // it needs exit the coroutine function context
        auto iter = m_kFunRecordMap.find(m_iCurrentFuncIndex);
        if (iter != m_kFunRecordMap.end())
        {
                iter->second.pkCorContext = nullptr;
                jump_fcontext(t.fcont, nullptr);
        }
}
```

接下来就是到了硬核的汇编层面分析了，一定要静下心，看boost的巧妙设计。

拆解make_fcontext创建协程上下文的设计

先看一下boost对x64下寄存器的规划，可以看出看出，用了0x150h字节来保存运行的上下文，其中包括了XMM寄存器，通用寄存器，以及函数的参数，返回地址，以及windows平台规定的fiber相关操作。

继续说make_fcontext的内容，我加了详细的中文注释。

```
; standard C library function
EXTERN  _exit:PROC
.code

;函数原型是
;extern "C" BOOST_CONTEXT_DECL
;fcontext_t BOOST_CONTEXT_CALLDECL make_fcontext( \
; void * sp, std::size_t size, void (* fn)( transfer_t) );
;有三个参数，第一个栈顶指针，栈的大小，函数指针，分别对应rcx, rdx, r8
;其中fcontext_t就是void*,transfer_t是两个void*的结构体，void* context, void* da

; generate function table entry in .pdata and unwind information in
make_fcontext PROC EXPORT FRAME
```

```
; .xdata for a function's structured exception handling unwind behavior
.endprolog

; first arg of make_fcontext() == top of context-stack
    ; 将栈顶保存到rax中，栈顶这里是高内存
mov  rax, rcx

; shift address in RAX to lower 16 byte boundary
; == pointer to fcontext_t and address of context stack
    ; 将栈顶进行16对齐，位置减小一点，实际栈空间向下指一点
and  rax, -16

; reserve space for context-data on context-stack
; on context-function entry: (RSP -0x8) % 16 == 0
    ;向下偏150H的内存，实际就原来的栈空间，往下指150H，这个150H的映射关系
    ;上面表里面都有，这个地址是本次函数栈空间的结束，后面都用+操作
sub  rax, 0150h

; third arg of make_fcontext() == address of context-function
; stored in RBX
    ; 将函数指针r8传入栈内存+0x100H中，这里保留给rbx了
mov  [rax+0100h], r8

; first arg of make_fcontext() == top of context-stack
; save top address of context stack as 'base'
    ; 将第一个参数*sp rcx，表示栈顶的内存，传入栈内存+0xc8H中，这里保存base
mov  [rax+0c8h], rcx

; second arg of make_fcontext() == size of context-stack
; negate stack size for LEA instruction (== substraction)
    ; 将第二个参数size rdx，表示栈可以用的大小，进行取反
neg  rdx

; compute bottom address of context stack (limit)
    ; size进行取反后，减计算，可以得出栈底的位置，也就最低地址的大小，
; 地址是小的，不能越界
lea  rcx, [rcx+rdx]

; save bottom address of context stack as 'limit'
    ; 将最低地址的大小，放入到栈内存+0c0H中
mov  [rax+0c0h], rcx

; save address of context stack limit as 'dealloction stack'
    ; 由上面计算可以知，再次备份一下，这个地址也是将重新分配的地址大小，放到0xl
mov  [rax+0b8h], rcx
```

```
        ; set fiber-storage to zero
        ; 清空rcx，放到0xb0中
        xor  rcx, rcx
    mov  [rax+0b0h], rcx

        ; save MMX control- and status-word
        ;保存mmx的标识与控制状态
    stmxcsr  [rax+0a0h]
    ; save x87 control-word
    fnstcw  [rax+0a4h]

    ; compute address of transport_t
        ; 计算出回调函数参数transport_t的地址，也就是本函数栈底内存加上140H
    lea rcx, [rax+0140h]
    ; store address of transport_t in hidden field
        ; 将其存放到本函数栈底内存加上110H
    mov [rax+0110h], rcx

    ; compute abs address of label trampoline
        ; 计算编译后trampoline的地址，放到rcx中
    lea  rcx, trampoline
    ; save address of trampoline as return-address for context-function
    ; will be entered after calling jump_fcontext() first time
        ; 将这个地址放到本函数栈底内存加上118h中，作eip，后面会用这个来跳转
    mov  [rax+0118h], rcx

    ; compute abs address of label finish
        ; 计算编译后finish的地址，放到rcx中
    lea  rcx, finish
    ; save address of finish as return-address for context-function in RBP
    ; will be entered after context-function returns
        ; 将这个地址放到本函数栈底内存加上108h中，作eip，这里是rbp中，这里会退出
    ; 如果函数执行完，没有走清理，根据调用规则trampoline的push rbp，ret地址时
    ; 会到finish地址，也就是强制退出了
    mov  [rax+0108h], rcx

        ; 退中本函数调用
    ret ; return pointer to context-data

trampoline:
    ; store return address on stack
    ; fix stack alignment
        ; 保护rpb，并跳到rbx中，rbp由上面分析暂时完全退中标识
        ; rbx由上面分析可以知就是r8地址
```

```
    push rbp
    ; jump to context-function
    jmp rbx

finish:
    ; exit code is zero
    xor  rcx, rcx
    ; exit application
    call  _exit
    hlt
make_fcontext ENDP
END
```

本函数主要作用，rcx传新的堆内存作协程的栈内存，rdx传堆内存大小，r8协程函数，规定要的 *void (fn)(transfer_t)**格式协程函数。将传入的堆内存进行模拟压栈操作，向下分配0x150H字节进行和表中关系映射。最重要是下面两句

```
    lea  rcx, trampoline
    ; save address of trampoline as return-address for context-function
    ; will be entered after calling jump_fcontext() first time
        ; 将这个地址放到本函数栈底内存加上118h中，作eip，后面会用这个来跳转
    mov  [rax+0118h], rcx
```

取出标签trampoline的地址，并放入0x118h偏移的EIP中，下次再调用这个栈时，会巧秒的用到这个118h的EIP。接着看一下trampoline的内容。

```
trampoline:
    ; store return address on stack
    ; fix stack alignment
        ; 保护rpb，并跳到rbx中，rbp由上面分析暂时完全退中标识
        ; rbx由上面分析可以知就是r8地址
```

README.md

```
    jmp rbx

finish:
    ; exit code is zero
    xor  rcx, rcx
    ; exit application
    call  _exit
    hlt
```

可以看出就是先是pop出rbp，然后强行跳转到rbx对应的值中，所以关键是什么存放到rbx中，对于第一次来说，就是r8的值，也就函数的地址，设计的非常巧妙，如果不是第一次，我们就要看jump_fcontext函数，将什么存放rbx中。

随便说一下，这里rbp存放提finish的地址，如果不合法，程序ret返回时，会直接到finish标签下，直接调用系统调用函数exit，退出程序。

我作了一副图

拆解jump_fcontext切换协程上下文的设计

在分析之前，我们先看一下，这个汇编函数的具体内容，我已经加了详细的中文注释。

```
;保存当前的栈和寄存器与调用时rip（rsp+118h）（隐藏的）
;将jump的返回值transform地址写入到栈+110h地址中
;将当前栈转到transform。context，传入的r8转到transform。data

;将rdx的对应的栈切入，rsp
;将rdx栈的对应rip换入rsp，并到r10中，那么栈内存更大一级
;将上述的transform当参数，转给新函数地址
;无条件跳转过去执行
;将rdx栈对应110h地址，作为rax返回地址，返回的结果是传参用的transform

;所以jump函数，切到新context并跳转过去执行，保存老的栈并将老的栈地址-118h返回


.code

jump_fcontext PROC EXPORT FRAME
    .endprolog

    ; prepare stack
    ; 分配118H字节栈内存
    ;太重要了，就是118h完成rip交换
    lea rsp, [rsp-0118h]

    ;保存XMM寄存器相关操作
IFNDEF BOOST_USE_TSX
    ; save XMM storage
    movaps  [rsp], xmm6
    movaps  [rsp+010h], xmm7
    movaps  [rsp+020h], xmm8
```

```
    movaps  [rsp+030h], xmm9
    movaps  [rsp+040h], xmm10
    movaps  [rsp+050h], xmm11
    movaps  [rsp+060h], xmm12
    movaps  [rsp+070h], xmm13
    movaps  [rsp+080h], xmm14
    movaps  [rsp+090h], xmm15
    ; save MMX control- and status-word
    stmxcsr  [rsp+0a0h]
    ; save x87 control-word
    fnstcw  [rsp+0a4h]
ENDIF

    ; load NT_TIB
    ; 获取线程的TEB数据，MS x64约定的
    mov  r10,  gs:[030h]
    ; save fiber local storage
    ; 保存纤程大小数据，并放到rsp+b0H中，MS x64约定
    mov  rax, [r10+020h]
    mov  [rsp+0b0h], rax
    ; save current deallocation stack
    ; 保存当前线程分配栈的地址，并放到rsp+b8H中，MS x64约定
    mov  rax, [r10+01478h]
    mov  [rsp+0b8h], rax
    ; save current stack limit
    ; 保存当前线程分配栈的低地址，并放到rsp+c0H中，MS x64约定
    mov  rax, [r10+010h]
    mov  [rsp+0c0h], rax
    ; 保存当前线程分配栈的高地址，并放到rsp+c8H中，MS x64约定
    ; save current stack base
    mov  rax,  [r10+08h]
    mov  [rsp+0c8h], rax

    ; 保存常规寄存器
    mov [rsp+0d0h], r12  ; save R12
    mov [rsp+0d8h], r13  ; save R13
    mov [rsp+0e0h], r14  ; save R14
    mov [rsp+0e8h], r15  ; save R15
    mov [rsp+0f0h], rdi  ; save RDI
    mov [rsp+0f8h], rsi  ; save RSI
    mov [rsp+0100h], rbx  ; save RBX
    mov [rsp+0108h], rbp  ; save RBP

    ; rcx是特殊的，参传进来的，表示返回值
    mov [rsp+0110h], rcx  ; save hidden address of transport_t
```

```
    ; preserve RSP (pointing to context-data) in R9
    ; 昨时保存rsp到r9中
    mov  r9, rsp

    ; restore RSP (pointing to context-data) from RDX
    ; rdx是特殊的，参传进来的，这里是上次用到栈内存
    mov  rsp, rdx

    ;从上次的MM寄存器相关操作恢复
IFNDEF BOOST_USE_TSX
    ; restore XMM storage
    movaps  xmm6, [rsp]
    movaps  xmm7, [rsp+010h]
    movaps  xmm8, [rsp+020h]
    movaps  xmm9, [rsp+030h]
    movaps  xmm10, [rsp+040h]
    movaps  xmm11, [rsp+050h]
    movaps  xmm12, [rsp+060h]
    movaps  xmm13, [rsp+070h]
    movaps  xmm14, [rsp+080h]
    movaps  xmm15, [rsp+090h]
    ; restore MMX control- and status-word
    ldmxcsr  [rsp+0a0h]
    ; save x87 control-word
    fldcw   [rsp+0a4h]
ENDIF

    ; 同样从上次栈中恢复前程数据
    ; load NT_TIB
    mov  r10,  gs:[030h]
    ; restore fiber local storage
    mov  rax, [rsp+0b0h]
    mov  [r10+020h], rax
    ; restore current deallocation stack
    mov  rax, [rsp+0b8h]
    mov  [r10+01478h], rax
    ; restore current stack limit
    mov  rax, [rsp+0c0h]
    mov  [r10+010h], rax
    ; restore current stack base
    mov  rax, [rsp+0c8h]
    mov  [r10+08h], rax

    ; 恢复常规寄存器
```

```
    mov r12, [rsp+0d0h]  ; restore R12
    mov r13, [rsp+0d8h]  ; restore R13
    mov r14, [rsp+0e0h]  ; restore R14
    mov r15, [rsp+0e8h]  ; restore R15
    mov rdi, [rsp+0f0h]  ; restore RDI
    mov rsi, [rsp+0f8h]  ; restore RSI
    mov rbx, [rsp+0100h]  ; restore RBX --------------very important,will ju
    mov rbp, [rsp+0108h]  ; restore RBP

    ; 恢复110的数据
    mov rax, [rsp+0110h] ; restore hidden address of transport_t

    ; prepare stack
    ; 将上次的EIP内存地址放到rsp寄存器中，very nice
    lea rsp, [rsp+0118h]

    ; load return-address
    ;这里将rip间接的传到到r10中，并rsp+8
    pop  r10

    ; transport_t returned in RAX
    ; return parent fcontext_t
    ;将r9，r8保存到【rax】对应的内存中
    ;r9是切换前的栈内存，r8是传入第二个参数所指内存
    mov  [rax], r9
    ; return data
    mov  [rax+08h], r8

    ; transport_t as 1.arg of context-function
    ; 将rax转为参数rcx
    mov  rcx,  rax

    ; indirect jump to context
    ;跳转到r10执行代码，这时间r10实际是指向make_fcontext汇编的trampoline指令地址
    ;也就是push rbp，jmp rbx
    jmp  r10
jump_fcontext ENDP
END
```

这段汇编非常巧秒的完成上下文的切换，其实主要作用功能如果下。

可以看出RIP的交换主要通过前面说的0x118h,并不像我们自己设计的那么直白，带有一定的隐藏性。

```
; prepare stack
; 将上次的EIP内存地址放到rsp寄存器中，very nice
lea rsp, [rsp+0118h]

; load return-address
;这里将rip间接的传到到r10中，并rsp+8
pop  r10
```

实际汇编的第一句，根据call的调用原则，rsp实际就是指向函数返回的EIP，但他并不是直接保存，来了减118h，然后下次调用是加上118h，然后pop r10，多么巧的设计。

```
; prepare stack
; 分配118H字节栈内存
;太重要了，就是118h完成rip交换
lea rsp, [rsp-0118h]
```

总结： 相信认真读到这里的，也理解了，根据GitHub的示例代码，再调试调试，一定有能彻底理解透。因为代码是demo代码，可能有一些我暂未发现的问题，可以提出一起讨论。本文就到这结束吧，下次可以讲一下高级点的用法。

Releases

No releases published

Packages

No packages published

Languages

- **Assembly** 61.0%
- **C++** 31.8%
- **C** 4.5%
- **C#** 2.7%