第 2 章 Java 执行引擎工作原理：方法调用

本章摘要

◎ JVM 如何进行方法调用
◎ JVM 如何分配方法栈
◎ JVM 如何取指
◎ JVM 如何执行逻辑运算

JVM 作为一款虚拟机，也必然要涉及计算机核心的 3 大功能。

1．方法调用

方法作为程序组成的基本单元，作为原子指令的初步封装，计算机必须能够支持方法的调用。同样，Java 语言的原子指令是字节码，Java 方法是对字节码的封装，因此 JVM 必须支持对 Java 方法的调用。

2．取指

这里的“取指”，是指取出指令。

还是那句话，方法是对原子指令的封装，计算机进入方法后，最终需要逐条取出这些指令并逐条执行。Java 方法也不例外，因此 JVM 进入 Java 方法后，也要能够模拟硬件 CPU，能够从 Java 方法中逐条取出字节码指令。

see more please visit: https://homeofpdf.com

3．运算

计算机取出指令后，就要根据指令进行相应的逻辑运算，实现指令的功能。JVM 作为虚拟机，也需要具备对 Java 字节码的运算能力。

本章主要分析 JVM 如何从内部调用 Java 方法。

2.1 方法调用

到目前为止，人类发明出了若干种编程语言，有的编程语言没有类概念，有的编程语言面向过程，但不管是哪种编程语言，至少都会包含函数的概念。通过函数将一个大的程序拆分成体积小、功能明确的一个个简短的函数，从而将一个复杂的大型问题分解成若干个简单的小问题，由繁到简。虽然函数并不总是大型软件模块化的手段，但一定是模块化得以实现的基础，否则随便开发个稍微难一点的功能，一写就是几千、几万行代码，估计没几个人能看懂，更没几个人有耐心看。

同理，Java 程序最基本的组成单位是类，而 Java 类也是由一个个的函数所组成，在这一点上，Java 也玩不出什么花样。

有的编程语言由真实的物理机器运行，有的程序运行于虚拟机上。既然所有的编程语言都由函数组成，那么运行由这些编程语言所开发出来的程序的机器就必须能够执行函数调用，不管是物理机器还是虚拟机器。JVM 作为一款虚拟机，要想具备执行一个完整的 Java 程序的能力，就必定得具备执行单个 Java 函数的能力。而要具备执行 Java 函数的能力，首先必须得能执行函数调用。

经过前面的讨论我们知道，詹爷当年为了能够让 Java 这门编程语言兼容各种平台，最终使用了一个大招——在运行时将 Java 字节码指令动态翻译成本地机器指令，从而既能获取兼容性，又能获取很高的运行效率。因此，JVM 实际上最后调用的并不是真正的 Java 函数，而是其对应的一堆机器指令。那么 JVM 究竟是怎么做到直接调用机器指令的呢？要研究清楚这个问题，必定要先弄清真实的物理机器是如何调用机器指令的。下面让我们简单了解一下真实的物理机器执行函数调用的机制。

2.1.1 真实的机器调用

本节主要通过一个汇编程序讲解一些真实的机器调用原理，若有道友看不懂也没有关系，多看几遍就懂了（哈哈）。想研究 JVM 的执行引擎原理，汇编这道坎必须得过，别无他法。

see more please visit: https://homeofpdf.com

真实的机器指令调用机制涉及的知识比较多，例如，现场保存、堆栈分配、参数传递，等等。我们不需要知道那么专业的基础知识，只需要了解大体的原理，了解了这些原理，再理解 JVM 的函数调用就会变得简单了。废话少说，翠花，上菜！

下面这段程序是使用汇编编写的，程序功能很简单，对 2 个整数进行求和（注：由于直接写机器指令，相信没人能够看得懂，因此这里以机器指令的助记符——汇编语言进行演示）。例子如下：

清单：示例程序

作用：使用汇编进行求和

```
main:
    pushl   %ebp
    movl%esp, %ebp
    subl$32, %esp
    movl$5, 20(%esp)
    movl$3, 24(%esp)
    movl24(%esp), %eax
    movl%eax, 4(%esp)
    movl20(%esp), %eax
    movl%eax, (%esp)
    calladd
    movl%eax, 28(%esp)

    movl$0, %eax
    leave
    ret

add:
    subl$16, %esp
    movl12(%ebp), %eax
    movl8(%ebp), %edx
    addl%edx, %eax
    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax
    leave
    ret
```

如果你不具备汇编基础知识，看不懂汇编程序，那么很难理解上面这段汇编程序的逻辑。不过看不懂没关系，只要能够明白其大意即可，不管哪种编程语言，其操作的无非是内存、数据，因此能够明白这段程序对内存做了什么，就可以了。

这段汇编程序想要实现的功能很简单，就是对两个整数进行求和。

分析一段程序，一般首先看该程序由哪些模块组成，分析汇编程序也是一样。这段汇编程

see more please visit: https://homeofpdf.com

序在代码段中定义了两个标号，一个是 main 标号，一个是 add 标号。汇编语言中的标号类似于 C 语言中函数的概念，这里我们就当成函数好了。那么这段汇编程序中就定义了两个函数，一个是 main()函数，一个是 add()函数。看到这里，也许聪明的你很快就猜到，这段程序是不是在 main()函数中定义了两个变量，然后传递给 add()函数，由 add()函数执行加法运算呢？恭喜你，猜对了。

既然明白了程序的大体意思，下面来一起分析下具体的算法。在这个过程中，如果你对汇编语法不怎么了解，也没关系，我们只需要关注最终内存是怎样变化的就可以了。

1. main()函数详解

首先看 main()函数。我们先整体解释下 main()函数。

清单：示例程序
作用：使用汇编编写一段求和程序

```
main:
    // 保存调用者栈基地址，并为main()函数分配新栈空间
    pushl    %ebp
    movl%esp, %ebp
    subl$32, %esp          //这里就是分配新栈，一共32个字节

    // 初始化两个数据，一个是5，一个是3
    movl$5, 20(%esp)
    movl$3, 24(%esp)

    // 压栈：将5和3压入栈中
    movl24(%esp), %eax
    movl%eax, 4(%esp)
    movl20(%esp), %eax
    movl%eax, (%esp)

    // 调用add()函数
    calladd
    movl%eax, 28(%esp) //得到add()函数的返回结果

    //返回
    movl$0, %eax
    leave
    ret
```

看过 main()函数的代码注释后，我们知道，main()函数一共包含 5 步：保存调用者栈基地址，初始化数据，压栈，函数调用和返回。下面分别分析这 5 步过程。

see more please visit: https://homeofpdf.com

1）保存栈基并分配新栈

首先看第一步，main()函数从下面两条指令开始执行：

```
pushl    %ebp
movl%esp, %ebp
```

pushl %ebp 就是保存调用者的栈基地址。调用者是谁？谁能调用一个程序的主函数？当然是操作系统啦。movl %esp,%ebp 将调用者的栈基地址指向其栈顶。这两句是所有函数调用时都必定会执行的指令。add()函数的开头也是这两条指令。待会儿讲 add()函数调用时会再讲到它们，如果你不懂，可以先跳过。

执行完上面两条指令后，main()函数接下来执行下面这条指令：

```
subl$32, %esp
```

这条指令是干嘛用的？大家整天在讲分配栈空间，分配栈空间，这条指令就是干这事的。所以对于物理机器而言，分配堆栈空间非常容易，就一句话的事。这条指令中的 subl 表示减，指令中的%esp 表示当前栈顶。整条指令的含义是：将当前栈顶减去 32 字节的长度。为什么是减，而不是加呢？这是因为在 Linux 平台上，栈是向下增长的，从内存的高地址往低地址方向增长，因此每次调用一个新的函数时，需要为新的函数分配栈空间，新函数的栈顶相对于调用者函数的栈顶，内存地址一定是低位方向，因此新函数的栈顶总是通过对调用者函数的栈顶做减法而计算出来。

执行了这条指令后，main()函数就有了自己的方法栈啦，栈空间大小是 32 字节，一个字节包含 8 个二进制位，如果一个 int 类型的整数包含 4 字节的话，那么 main()函数的方法栈就一共可以容下 8 个 int 类型的数据（如图 2.1 所示）。

图 2.1　main()函数初始化时的堆栈

see more please visit: https://homeofpdf.com

注：由于 main()的方法栈空间刚刚分配，还没有往里面加任何东西，因此此时的栈是空的，什么都没有。同时，系统为 main()函数一共分配了 32 个字节，在 64 位平台上，一个 int 型数据占 4 个字节，因此这里将 main()方法栈以 4 字节为单位进行划分，一共划分成 8 块，每一块代表 4 个字节大小的空间。

main()函数执行完上面这 3 条指令，便完成了调用者栈基地址的保存和自身栈空间的分配。

2）初始化数据

main()函数接下来执行下面 2 条指令：

```
movl$5, 20(%esp)
    movl$3, 24(%esp)
```

这两条指令的含义是：分别将 5 和 3 这两个整数保存到 main()栈中，其中 20(%esp)表示当前栈顶（即 esp 寄存器当前所指向的内存地址）往上移动 20 字节位置，数据 5 就被保存在这里。由于 main()函数的栈空间一共有 32 字节那么大，因此从 main()方法栈的栈顶往上移动 20 字节后的位置，依然在 main()的方法栈内。同理，整数 3 被保存到了 main()函数栈顶往上偏移 24 字节处的位置。由于一个整数占用 4 字节（在 64 位平台上，本书中所有示例均在 64 位平台上测试，因此本文所有的 int 型数据默认都占 4 字节，下文不再赘述），因此 5 和 3 被分别保存到 main()方法栈顶往上偏移 5 个整数和 6 个整数的位置。

语言描述让人理解起来总是很费电，还是图来得直观些。下面我们就通过图示来描述偏移位置。

在使用图示来描述内存位置之前，我们先研究一下真实的物理机器上是如何进行内存定位的。

如果我们将 main()函数的栈顶位置标记为(%esp)，整个 main()方法栈空间的 32 字节，按照每 4 字节为单元进行划分，同时按照其相对于栈顶位置的偏移量来标记 main()的方法栈，那么 main()函数的方法栈内存可以如图 2.2 所示来标记。

see more please visit: https://homeofpdf.com

图 2.2　main()函数堆栈存储单元的相对位置

这种将方法栈内存位置按照栈顶偏移量进行标记的方法，是整个计算机的理论基础，直接影响了编程语言的模块划分方法。如果没有这种方法，编译器很难在编译期就确定各个变量的位置，更不用谈各种基于编译原理的高级应用了。

其实这种标记方法在现实世界中也是很常见的。对于地球上任意一个点，人类可以使用两种方式标记其位置，一种是绝对定位，一种是相对定位。例如以某个学校为例，我们可以说这个学校在经度 30°和纬度 60°，也可以说这个学校在前方红绿灯右转 300 米。物理机器对方法内部的局部变量的定位就类似于对学校的“红绿灯右转 300 米”的相对定位法。

理解了相对定位方法后，我们再来看看 5 和 3 这两个数据在 main()方法栈中的位置，如图 2.3 所示。

图 2.3　完成局部变量初始化的 main()方法堆栈布局

see more please visit: https://homeofpdf.com

也许有些完美主义者看出点儿问题来了，即 5 和 3 为什么被分配在中间的位置，上边和下边哪都不挨，这是为什么呢？这里面是不是有什么讲究？讲究还真有，栈底那个位置即 28(%esp)是留给调用 add()函数的返回值的，这个待会儿会讲到。

现在，main()函数完成了数据准备，接着便要开始调用 add()函数进行求和了。

3）压栈

接着，main()函数开始执行下面 4 条指令：

```
movl24(%esp), %eax
movl%eax, 4(%esp)
movl20(%esp), %eax
movl%eax, (%esp)
```

这 4 条指令主要作用是“压栈”。前两条指令是把数据 3 压栈，后两条指令是把数据 5 压栈。

先看 movl 24(%esp), %eax 这条指令，该指令将 24(%esp)处的内存值传送到 eax 寄存器中。24(%esp)处的内存值是什么呢？就是刚才第 2 步中保存的整数 3。接着 movl %eax, 4(%esp)这条指令又将 eax 寄存器中的值传送到了 4(%esp)这个地方。如果将这两条指令连着看，你会发现这里进行的数据传送的路径是：x->y, y->z，使用小学数学进行推理，可以推出：x->z，即 x 处的内存值被传送到了 z 处。因此，最终的效果是：整数 3 被从 24(%esp)这个相对于栈顶的偏移位置，转移到了 4(%esp)这个偏移位置。

同样的道理，后面两条指令的最终效果是：整数 5 被从 20(%esp)这个相对于栈顶的偏移位置，转移到了(%esp)这个偏移位置。(%esp)是哪个位置？请相信你的第一直觉，就是栈顶。

也许好奇心重的同学会想，既然 CPU 可以将一个数据从 *x* 点移到 *y* 点，再从 *y* 点移到 *z* 点，那么为什么不直接从 *x* 点移到 *z* 点呢？这真是个好问题。梦想是美好的，但是现实是残酷的，因为 CPU 不支持将数据从一个内存位置直接传送到另一个内存位置，若要想实现这个效果，必须使用寄存器进行中转。这里以移动数据 3 为例，数据 3 最终被从 24(%esp)这个内存位置移到了 4(%esp)这个内存位置，CPU 先将 3 从 24(%esp)移到了 eax 寄存器，再将 3 从 eax 寄存器移到了 4(%esp)这个内存位置。CPU 无法直接执行下面的这条指令：

```
movl 24(%esp), 4(%esp)
```

只因为 24(%esp)和 4(%esp)都代表的是内存位置，因此 CPU 无法直接完成内存之间的数据传送。

这 4 条指令执行下来，main()函数的方法栈内存布局如图 2.4 所示。

see more please visit: https://homeofpdf.com

图 2.4　完成 add()参数复制后的 main()函数堆栈布局

一般而言，往栈顶传送数据的行为叫做“压栈”，这里先后将 3 和 5 都放到了栈顶处，因此这都是压栈。

main()函数为何要压栈呢？那是因为 main()函数即将要进行函数调用了。

真实的物理机器，在发起函数调用之前，必定要进行压栈。压栈的目的是为了传参。

main()函数在这里压栈的两个数据，将会被 add()函数读取到。

4）函数调用

压完了栈，main()函数终于开始进行函数调用了。对于物理机器而言，函数调用特别简单，就一条指令：

```
call    add
```

但是，看着简单，背后却有一套机制在支撑。这个下面会讲。

add()函数执行完，会将计算的结果保存到 eax 寄存器中。main()函数要取得 add()函数返回值，便直接从 eax 寄存器中拿即可。因此，执行完 call add 函数调用指令后，main()函数接着调用下面的指令：

```
movl%eax, 28(%esp)
```

通过这条指令，main()函数终于完成了求和计算，并拿到了计算结果。此时，main()函数的方法栈内存布局如图 2.5 所示。

see more please visit: https://homeofpdf.com

图 2.5　完成 add()调用后的 main()函数堆栈布局

到了这里，你会发现其实这样的内存布局还是挺美的，什么叫美？历史上无论是科学大牛还是艺术巨匠都告诉我们：对称的，才是美的。你看现在的 main()方法栈的内部布局图，就挺对称，上面的数据都挨着栈底，下面的数据都挨着栈顶。之所以会这样，全是编译器的功劳。编译器会将一个方法内的局部变量分配在靠近栈底的位置，而将传递的参数分配在靠近栈顶的位置。

5）返回

函数返回很简单，将返回值保存到 eax 寄存器中，然后执行两条例行返回指令便大功告成。main()函数的返回指令就是下面这 3 条：

```
movl$0, %eax
leave
ret
```

好了，到此为止，我们完整地分析了 main()函数的执行过程以及栈内存的布局演变。在分析的过程中，顺带着了解了 main()函数为了调用 add()函数而做的准备。在本示例中，main()函数为了调用 add()函数，主要是将入参进行了“压栈”操作，这样在 add()函数内部才能取到参数并进行求和运算。

但是，除了压栈外，其实系统还要做一部分工作，才能完成最终的方法调用。下面我们通过分析 add()函数的执行过程，来了解系统工作的机制。

2．add()函数详解

如果你对上面 main()函数的执行原理已经比较熟悉，那么理解 add()函数的运行机制就容易

see more please visit: https://homeofpdf.com

了。在分析 add()函数之前，我们先看下 add()函数的整体逻辑：

清单：示例程序

作用：使用 C 程序提供的语法糖修改 CS:IP 段寄存器的指向

```
add:
    // 保存调用者栈基地址
    pushl   %ebp
    movl%esp, %ebp
    subl$16, %esp

    // 获取入参
    movl12(%ebp), %eax
    movl8(%ebp), %edx

    // 执行运算
    addl%edx, %eax
    movl%eax, -4(%ebp)

    // 返回
    movl-4(%ebp), %eax
    leave
    ret
```

可以看到，add()函数总体上分为 4 步：保存调用者栈基地址，读取入参，执行运算，返回。下面我们逐一分析过程。

1）保存调用者栈基地址

add()函数也是以下面两条指令开始：

```
pushl   %ebp
movl    %esp, %ebp
```

这一步大家应该很熟悉，刚才在分析 main()函数时就已经讲过。这两条指令主要是保存调用者栈基地址。这里再啰嗦一次，物理机器在执行函数调用时，被调用者总是要保存调用者栈基地址。这是因为 esp 和 ebp 这两个寄存器接下来要指向被调用者的栈基地址和栈顶，这两个寄存器原本保存的是调用者的栈基地址和栈顶地址，现在即将被修改，如果不保存起来，那么当被调用者函数执行完成，程序流返回到调用者流程中时，物理机器将无法恢复调用者的栈基和栈顶，从而导致程序无法继续执行下去。

add()函数第 3 条指令是：

```
subl    $16, %esp
```

这条指令大家应该很熟悉，就是分配栈空间。为谁分配？当然是为当前函数 add()。分配多大空间？这条指令中的 16 来指定。不过要注意，这里指 16 字节，而不是 16 个二进制位。

see more please visit: https://homeofpdf.com

现在，让我们看看方法栈空间的内存布局。由于 add()函数的方法栈是在调用方 main()函数的方法栈空间基础上往下增长的，并且 add()方法栈与 main()方法栈连在一起，因此现在我们同时看 main()和 add()两个函数的方法栈。如图 2.6 所示。

图 2.6　main()函数调用 add()函数时的整体堆栈布局

现在这张栈内存布局图同时包含 main()函数和 add()函数的方法栈。main()方法栈大小为 32 字节，add()方法栈大小为 16 字节。但是奇怪的是，在 main()和 add()方法栈的中间，竟然多了 8 个字节的空间。这 8 个字节的空间是从哪里冒出来的呢？答案很简单，在 main()函数执行 call add 指令时，物理机器自动往栈顶压了一个数值——eip。前面我们扫盲汇编基础知识时讲过，CPU 所执行的指令位置由 CS:IP 这 2 个段寄存器共同决定，这里的 eip 就是 IP 寄存器。物理机器为何要将这个寄存器的值入栈呢？这主要是为了让 main()函数执行完 call 调用回来之后，能够继续处理 main()函数中接下来的指令。我们看 main()函数中 call add 指令的上下文，如果没有 call add 指令，即如果 main()函数不调用 add()函数，那么 main()函数执行完 call add 这条指令的上一条指令 movl %eax, (%esp)之后，main()函数应该接着执行 call add 这条指令的后面一条指令 movl%eax, 28(%esp)，同时，main()函数在执行 movl %eax, 28(%esp)这条指令之前，eip 寄存器需要先指向这条指令，这样 CPU 才能读取到这条指令并执行。但是现在，main()函数在执行这条指令之前，先调用了 add()函数，那么 eip 就会指向 add()函数里面的指令内存位置（这一步是

see more please visit: https://homeofpdf.com

物理机器自动执行的），那么当 add()函数执行完其最后一条指令后，物理机器怎么知道接下来要把 eip 寄存器指向哪里，或者说物理机器怎么知道接下来要执行哪里的指令呢？所谓的函数只是人类进行的模块划分，物理机器可不知道有这些东东，物理机器更不会自动记忆当前函数被哪个函数调用，更不会在执行完当前函数后，自动跳转到当前函数的调用者函数中去执行。所以，执行完一个函数，我们不去修改 eip 寄存器的值，那么物理机器根本就不知道接下来应该怎么做。基于这样的原因，在执行函数调用时，CPU 设计者在里面加了一个功能，即在物理机器执行 call 指令时，自动将当前 eip 寄存器入栈。而当被调用者执行完之后，物理机器再自动将 eip 出栈，这样，执行完被调用函数之后，物理机器会接着执行调用者的后续指令。

刚刚解释了 main()和 add()函数中间多出来的 8 字节中的 eip，eip 占 4 字节，因此还有 4 字节的空间。我们看图 2.6 可以知道，这剩下的 4 字节空间存放着 ebp 的值。这个值是在执行 add()函数时入栈的。我们看，add()函数的开头第一条指令就是 pushl %ebp，这里显式执行了 push 入栈操作。

经过上面对 add()函数的初步分析，我们可以得出以下结论：

◎　物理机器执行 call 函数调用时，机器会自动将 eip 入栈。

◎　物理机器执行函数调用时，被调用方需要手动将 ebp 入栈。

add()函数执行完开头的 3 条指令后，机器开始进入 add()函数域，接下来开始执行 add()函数里面真正的逻辑运算。

2）读取入参

add()函数的第 4 和第 5 这两条指令为读取入参指令，先看指令：

```
movl12(%ebp), %eax
movl8(%ebp), %edx
```

第一条指令是 movl 12(%ebp), %eax，这条指令中使用了 2 个寄存器：ebp 和 eax，其中 ebp 寄存器的用途与 esp 一样专一，只用于标识栈底位置。对于 12(%ebp)这种写法我们已经比较熟了，表示从 ebp 寄存器所指向的内存地址往高地址方向偏移 12 字节。由于在 add()函数的一开始执行了 movl %esp, %ebp 指令，因此此时 ebp 寄存器已经指向了原来 main()函数的栈顶。第一条指令合起来的意思就是，从 add()函数栈底向上偏移 12 字节的位置取出数据（占 4 字节），将该数据传送给 eax 寄存器。

同理，第二条指令的意思是，从 add()函数栈底向上偏移 8 字节的位置取出数据（占 4 字节），将该数据传送给 eax 寄存器。

通过这两条指令，add()函数成功从 main()函数中获取到了两个入参。

see more please visit: https://homeofpdf.com

为什么要从这两个位置读取入参呢？这是因为 main()函数在把两个入参压栈后，执行 call 函数调用指令，由于系统会继续将 eip 和 esp 压栈，这两个数据共占 8 字节，因此导致 add()栈顶与 main()函数中压栈的两个参数之间隔着 8 字节的距离，因此 add()函数要分别从这两个位置获取入参。

我们在堆栈内存图上将这两个入参的位置标记出来，如图 2.7 所示。

图 2.7　main()函数调用 add()函数时所压栈的入参位置

之前在 main()方法中执行参数压栈指令时，当时压栈的两个位置分别是(%esp)和 4(%esp)，而现在这两个位置的标记变成 8(%ebp)和 12(%ebp)，由此可见，随着堆栈寄存器 ebp 和 esp 所指向位置的变化，方法栈中同一个内存位置的偏移量也随之改变。同时，对于压栈的入参，既可以从通过相对于调用者函数的栈顶的偏移量来相对定位，也可以通过相对于被调用者函数的栈底的偏移量来相对定位。当然，如果你愿意，你也可以通过相对于调用者函数的栈顶偏移位置来相对定位。总之，对方法栈内存的定位手段是灵活的，可以选择不同的参考系，不同的定位基准决定了不同的偏移量和定位方法。但是对于被调用者函数的方法栈内的数据，却不能以调用者函数为基准通过偏移量获取，因为此时被调用函数尚未分配方法栈空间，根本取不到数据，甚至会取到错误的数据。下面我们在 add()指令中将会遇到不同的相对定位方式。

see more please visit: https://hometofpdf.com

3）执行运算

add()函数主要功能是求和，而求和运算是物理机器的最基本的功能之一，因此物理机器提供了一条指令专门用于求和：add。add()函数中求和的指令如下：

```
addl %edx, %eax
```

这条指令的含义是，将 edx 寄存器中的值与 eax 寄存器中的值相加，相加结果保存到 eax 寄存器中。

在 add()函数执行本指令之前，已经通过读取入参指令将 main()函数所传递过来的两个参数分别读取到了 eax 和 edx 这两个寄存器中，因此对这两个寄存器执行求和操作，就相当于对 main()函数传递过来的 2 个入参执行求和。

执行完求和运算，add()函数接着执行 movl %eax, -4(%ebp)这条指令。这条指令的作用是把 eax 寄存器中的值转移到栈基地址往下偏移 4 个字节的位置。这个位置是哪里呢？其实就是 add()函数的方法栈内的第一个位置。add()函数执行到现在，虽然分配了 16 字节的空间，但是一直还未使用过。现在终于用到了。此时 eax 寄存器中存放的是什么数据呢？就是刚刚执行 add 求和指令的结果。由此可知，add()将求和的结果保存在了其方法栈的第一个位置。此时整体堆栈内存布局如图 2.8 所示。

图 2.8　add()方法完成求和时的堆栈布局

see more please visit: https://homeofpdf.com

这里，我们可以看出，对一个具体的方法栈中的某个位置而言，我们既可以通过 ebp（栈底）进行相对定位，也可以通过 esp（栈顶）进行相对定位。例如这里，对于 add()函数方法栈的第一个位置，既可以通过-4(%ebp)相对定位，也可以通过 12(%esp)相对定位。esp 或 ebp 前面的正负符号表示相对基准位置是向上偏移还是向下偏移。

4）返回

执行完求和运算后，add()函数的使命便完成了，接下来该返回了。函数返回的一般逻辑是，如果有返回值，就把返回值放在 eax 寄存器中，然后执行 leave 和 ret 指令。如果没有返回值，则直接执行 leave 和 ret 指令。当然这里多啰嗦一点，leave 和 ret 指令其实封装了好几个命令，以后大家看到其他汇编函数返回时执行的不是这两条命令，也不要感到太意外。指令是死的，人是活的。有的人喜欢使用封装好的指令，而有的人则喜欢使用最原始的指令，就好比巫师念起那最古老的咒语一般。

至此，add()函数全部执行结束，程序流程又返回到了 main()函数中。程序流为何能够返回到 main()函数中呢？前面讲过，物理机器要想执行某个函数中的指令，必须先把 ip 段寄存器指向那个函数。ip 指向哪里，物理机器就执行到哪里。在 add()函数返回的 leave 指令中，其实顺带着执行了下面这两条指令：

```
mov %ebp, %esp
pop %eip
```

在 main()函数调用 add()函数时，物理机器执行了 push %eip 指令，将 main()函数中 add 指令的下一条指令的内存地址入栈。现在 add()函数执行完了，再把这个地址弹出来（出栈），保存到 eip 寄存器中，这样 eip 寄存器就又指向 main()函数中 add 指令的下一条指令了。当然在弹出 eip 之前，得先通过 mov %ebp, %esp 指令将栈顶指向栈基位置，这样栈顶位置就是 eip。

通过上面对一段简单的汇编程序的分析，我们知道了真实的物理机器调用函数的本质原理。总体而言，物理机器在调用函数时，需要进行下面一系列操作：

（1）参数入栈。有几个参数就把几个参数入栈。此时入的是哪家的栈？是调用者自己的栈。不过不同的物理机器，入栈顺序是不同的，有的是顺序入栈，有的是逆序。

（2）代码指针（eip）入栈，这样等被调用函数执行完之后，物理机器可以再回来继续执行原来的函数指令。

（3）调用函数的栈基地址入栈。还是为物理机器从被调用者返回调用方做准备。如果不入栈保存起来，那么等到调用方从被调用方返回来的时候，物理机器就不知道调用方的方法栈在哪里了。

（4）为被调用方分配方法栈空间。每一个函数都有自己的栈空间，如同地球上的每一个人

see more please visit: https://homeofpdf.com

都有自己的小窝。

通过对这段汇编程序的分析可知，物理机器在执行程序时，将程序划分成若干函数，每个函数都对应有一段机器码。一段程序的机器码都放在一块连续的内存中，这块内存叫做代码段。物理机器为每一个函数分配一个方法栈，方法栈与代码段在地址上没有任何关系，并且只有当物理机器执行到某个函数时，才会为其分配方法栈，否则就不会分配。函数通过自身的机器指令遥控其对应的方法栈，可以往里面放入数值，也可以将数值移动到其他地方，也可以从里面读取数据，也可以从调用者的方法栈里取值。通过一条条指令和一个个栈，物理机器得以运行完一整个程序。

知道了物理机器调用函数的这些秘密，我们再分析 JVM 的函数调用机制就不会感到高深莫测了。其实 JVM 也是将 Java 函数所对应的机器指令专门存储在内存的一块区域上，同时为每一个 Java 函数分配了方法栈。但是在真正了解 JVM 的函数调用机制之前，我们还有最关键的一步需要了解。而要了解这一步，我们需要先看看同样是高级语言的 C 语言是如何执行函数调用的。

2.1.2　C 语言函数调用

讲完物理机器函数调用的原理后，接下来我们一起看看 C 语言是如何实现函数调用的。毕竟 JVM 是混合 C 和 C++开发而成的，JVM 执行引擎最关键的一点就在于实现了由 C 语言动态执行机器指令，这正是 JVM 与机器指令的边界所在。

C 语言属于静态编译型语言，C 语言开发的程序被编译后，直接生成二进制代码，而这些二进制代码正是由一条条机器指令组成，因此可直接被物理机器执行。所以，C 程序中的函数调用，本质上还是依靠物理机器所提供的 call 指令来完成。

1. 一个简单的 C 程序

首先来看一个简单的例子：

清单：示例程序
作用：使用 C 进行求和

```
int add();
int main(){
    int c=add();
    return 0;
}

int add(){
```

see more please visit: https://homeofpdf.com

```
    int z=1+2;
    return z;
}
```

本例十分简单，add()函数主要完成两个整数的累加并返回累加结果，main()函数调用 add()函数并返回 0。

本例中的 add()函数没有参数，因此不涉及调用者与被调用者之间的参数传递。

将这段 C 程序编译成汇编程序，如下所示：

清单：示例程序

作用：C 程序对应的汇编

```
main:
    pushl   %ebp
    movl%esp, %ebp
    andl$-16, %esp
    subl$16, %esp

    calladd
    movl$0, %eax
    leave
    ret

add:
    pushl   %ebp
    movl%esp, %ebp
    subl$16, %esp
    movl$3, -4(%ebp)
    movl-4(%ebp), %eax
    leave
    ret
```

仔细观察这段汇编程序，你会发现这段汇编程序有两个标号，分别是 main 和 add。而巧合的是，源程序 C 程序中恰好包含了 main()函数和 add()函数。其实，这不是巧合，而是编译器处理的结果。编译器在编译 C 程序时，会将 C 程序中的函数名处理成汇编程序中的标号。

事实上，这段汇编程序中的 main 和 add 正是汇编程序的 2 个代码段。汇编程序由一个一个代码段组成，如同 C 程序由一个一个函数组成一样。

有了标号，汇编程序就能执行函数调用，即执行 call 指令。可以看到，在这段汇编程序中，main 代码段所对应的汇编代码中，有一条 calladd 指令，这正是汇编语言中的函数调用指令。因此，C 程序中的 main()函数调用 add()函数，在汇编程序中就转换成了 calladd。这就是 C 程序函数调用的秘密所在。

see more please visit: https://homeofpdf.com

继续仔细分析这段汇编程序，我们会发现无论是 main 代码段，还是 add 代码段，一开始都包含下面 3 条指令：

```
pushl   %ebp
    movl%esp, %ebp
    andl$-16, %esp
```

不过 add 代码段一开始的第 3 条指令与 main 代码段的第 3 条指令有所不同，add 代码段的第 3 条指令是 subl $16, %esp，该指令其实与 main 代码段中的第 3 条指令 andl %-16, %esp 所表达的含义是相同的，那就是分配堆栈空间。

这 3 条指令的含义在上面讲解汇编函数调用机制时详细说明过，其作用是，保存调用者栈基地址，并为新方法分配方法栈，这几乎成为汇编程序进行方法调用的标准定式。不过并非所有的汇编程序中的方法都会按照这样的顺序来执行，并且这 3 条指令也并不一定紧靠在一起，在本例中之所以会如此一致，那都是编译器处理后的结果。但是编译器是死的，而人是活的，在 JVM 中，很多汇编程序直接由人工编写，并非依靠编译器，因此会有所不同。不过总体而言，汇编程序在进入方法时，这 3 条指令是不会少的。

研究函数调用，堆栈分析是一定不可少的，这是物理机器实现方法调用的核心算法。在本例中，当 main()函数执行到 call add 这条指令之前，物理机器会为 main()函数分配方法栈，堆栈空间是 16 字节。main()函数的方法栈内存布局如图 2-9 所示。

图 2.9　main()函数初始堆栈布局

此时 main()方法栈是空的，啥都没有。

当物理机器执行完 add()函数的最后一条指令 movl-4(%ebp), %eax 时，堆栈内存布局如图 2.10 所示。

see more please visit: https://homeofpdf.com

图 2.10　执行完 add()函数时的整体堆栈布局

可以发现，物理机器将 eip 和 ebp 两个寄存器压入 main()函数的栈顶（上一节讲过），同时为 add()函数分配了 16 字节的堆栈空间。由于 add()函数源程序的 int z=1+2 被编译器自动算出了结果 3，因此编译器直接将立即数 3 分配到了 add()函数的方法栈中第一个位置，也即最靠近 add()函数方法栈栈底的位置，分配的指令就是汇编程序中 add 代码段中的：

```
movl $3, -4(%ebp)
```

当程序执行完 add()函数，又回到 main()函数中后，物理机器会回收 add()函数的堆栈空间，同时 eip 和 ebp 这两个寄存器出栈，于是堆栈中又只剩下了 main()函数的内存。main()函数会从 eax 寄存器中读取 add()函数的返回值（函数返回值会被暂存在 eax 寄存器中，这是约定），并将其传送到 main()函数方法栈的第一个位置，main()函数中的 movl%eax, 12(%esp)指令执行的正是这种数据传送。此时，堆栈内存布局如图 2.11 所示。

图 2.11　main()函数完成 add()调用之后的堆栈布局

see more please visit: https://homeofpdf.com

可能有人会问，main()函数其实只需要分配 4 字节的堆栈空间，用于保存计算结果 3，可为何编译器为其分配了 16 字节的空间呢？这是因为这也是一种约定，就是内存对齐。在 32 位和 64 位机器上，堆栈内存都是按照 16 字节进行对齐的，多了不退，少了一定会补齐。之所以会有这种约定，道理很简单，就是为了能够对内存进行快速定位、快速整理回收。

2．带入参的 C 程序

刚才所举例子中的 C 程序，并不涉及参数传递。下面我们将 C 程序稍微修改一下，使函数带有参数。

修改后的 C 程序如下：

清单：示例程序

作用：带有入参的 C 求和程序示例

```
int add(int a, int b);
int main(){
    int a = 5;
    int b = 3;
    int c=add(a, b);
    return 0;
}

int add(int a, int b){
    int z=1+2;
    return z;
}
```

将其编译成汇编程序，如下所示：

清单：示例程序

作用：带有入参的 C 求和程序对应的汇编

```
main:
    pushl   %ebp
    movl%esp, %ebp
    andl$-16, %esp
    subl$32, %esp

    movl$5, 20(%esp)
    movl$3, 24(%esp)
    movl24(%esp), %eax
    movl%eax, 4(%esp)
    movl20(%esp), %eax
    movl%eax, (%esp)

    calladd
```

see more please visit: https://homeofpdf.com

```
movl%eax, 28(%esp)
movl$0, %eax
leave
ret

add:
pushl   %ebp
movl%esp, %ebp
subl$16, %esp
movl$3, -4(%ebp)
movl-4(%ebp), %eax
leave
ret
```

上面这段汇编程序是不是看起来很熟悉？汇编程序中包含两个标段 main 和 add，main 可以向 add 传入两个整型参数，add 负责对两个入参进行求和并将累加结果返回给 main。其实这就是前文在分析物理机器函数调用时所举的例子。

前面那个例子其实就是这段 C 程序编译后的汇编程序。前文已经详细分析过这段汇编程序的逻辑以及执行过程中的堆栈内存布局及其变化，此处不再赘述。本节我们主要研究的是 C 程序的函数调用机制。

前面我们分析了没有参数传递场景下的 C 程序函数调用机制，这里我们再次温习一下有参数传递场景下的 C 程序函数的调用机制，主要包括以下 3 点。

1）压栈

如同前文分析，本例中的 main()函数在调用 add()函数之前，会将两个入参压栈，压入栈之后，add()就可以取得这两个入参。压入了谁的栈？当然是调用者的堆栈，即 main()函数。

2）参数传递顺序

对于 Linux 平台而言，调用者函数向被调用者函数传递参数时，采用逆向顺序压栈，即最后一个参数第一个压栈，而第一个参数最后压栈。

这种压栈顺序决定了被调用函数对入参进行寻址的方式和顺序。

3）读取入参

main()函数将入参压栈后，作为被调用者的 add()函数怎么获取入参呢？前文分析过，是通过 add()函数的栈基地址 ebp 的相对地址，从 main()函数中读取入参。最后一个入参位置在 8(%ebp)，倒数第二个入参位置在 12(%ebp)，以此类推。

之所以要从相对 ebp 寄存器往上第 8 个存储位置开始寻址，是因为调用者与被调用者函数堆栈之间隔着 eip 和 ebp 这两个寄存器值。

see more please visit: https://homeofpdf.com

以上 3 点应当牢记，这对后面分析 JVM 的执行引擎机制大有裨益。

总体而言，本节主要通过汇编程序和 C 语言程序演示了真实的物理机器执行函数调用时的原理，如果不把这些内容弄清楚，则在研究 JVM 执行引擎时会寸步难行。在真实的物理机器上，执行函数调用时主要包含以下几个步骤：

（1）保存调用者栈基地址（即当前栈基地址入栈），当前 IP 寄存器入栈（即调用者中的下一条指令地址入栈）。

（2）调用函数时，在 x86 平台上，参数从右到左依次入栈。

（3）一个方法所分配的栈空间大小，取决于该方法内部的局部变量空间、为被调用者所传递的入参大小。

（4）被调用者在接收入参时，从 8(%ebp)处开始，往上逐个获取每一个入参参数。

（5）被调用者将返回结果保存在 eax 寄存器中，调用者从该寄存器中获取返回值。

2.2　JVM 的函数调用机制

前文我们一起学习了物理机器的函数调用机制和 C 语言的函数调用机制，本质上这两者的函数调用机制是相同的，因为 C 语言是静态编译型语言，编译后就变成了能够直接被物理机器执行的二进制代码，所以 C 程序中的函数调用最终还是直接依赖物理机器的函数调用指令。

现在如果让我们来开发类似 JVM 这样的一款解释型虚拟机，首先要解决的问题就是函数调用，你会怎么做呢？

JVM 是用 C 和 C++语言编写的一款软件，当 JVM 执行 Java 函数时，实际上是执行了一段汇编代码，换言之，这中间一定存在一个边界，在边界处，边界的一边是 C 程序，边界的另一边直接是机器指令，C 语言要能够直接执行机器指令。例如，如果你编写了下面一段非常简单的 Java 代码：

清单：示例 Java 程序
作用：演示 Java 语言求和

```
public class Test {
    public static void main(String[] args) {
        add(5,8);
    }

    public static int add(int a, int b) {
        int c = a + b;
```

see more please visit: https://homeofpdf.com

```
        int d=c+9;
        return d;
    }
}
```

这段 Java 代码很简单，大家一看就懂，main()函数调用 add()函数，add()函数对两个入参进行相加求和并返回累加结果。

执行 javap –verbose 命令，分析这段 Java 代码编译成的字节码内容，得到如下输出：

清单：示例 Java 程序所对应的字节码内容

作用：Java 程序所对应的字节码内容分析

```
public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=1, Args_size=1
   0:iconst_5
   1:bipush  8
   3:invokestatic    #2; //Method add:(II)I
   6:pop
   7:return
  LineNumberTable:
   line 4: 0
   line 5: 7

public static int add(int, int);
  Code:
   Stack=2, Locals=4, Args_size=2
   0:iload_0
   1:iload_1
   2:iadd
   3:istore_2
   4:iload_2
   5:bipush  9
   7:iadd
   8:istore_3
   9:iload_3
   10:ireturn
  LineNumberTable:
   line 8: 0
   line 9: 4
   line 10: 9
}
```

这段字节码指令，若你还不能看懂，可以暂且放下，本书后面会对其进行详细讲解。现在我们在 JVM 上运行这个 Java 字节码文件，在运行时打印 JVM 所执行的机器指令（需要为 JVM 安装 HSDIS 插件）。让 JVM 以模板解释器来解释运行这段字节码，JVM 将输出如下字节码与

see more please visit: https://homeofpdf.com

机器指令的对应内容：

清单：示例 Java 程序

作用：Java 字节码指令所对应的机器指令逻辑

```
iload_0  26 iload_0  [0xb370b660, 0xb370b6a0]  64 bytes
[Disassembling for mach='i386']
  0xb370b683: push   %eax
  0xb370b684: mov    (%edi),%eax
  0xb370b686: movzbl 0x1(%esi),%ebx
  0xb370b68a: inc    %esi
  0xb370b68b: jmp    *-0x48ecd6a0(,%ebx,4)
  // ...
--------------------------------------------------------------------------
iload_1  27 iload_1  [0xb370b6c0, 0xb370b700]  64 bytes
[Disassembling for mach='i386']
  0xb370b6e3: push   %eax
  0xb370b6e4: mov    -0x4(%edi),%eax
  0xb370b6e7: movzbl 0x1(%esi),%ebx
  0xb370b6eb: inc    %esi
  0xb370b6ec: jmp    *-0x48ecd6a0(,%ebx,4)
  // ...
--------------------------------------------------------------------------
iadd  96 iadd  [0xb370cf00, 0xb370cf20]  32 bytes
[Disassembling for mach='i386']
  0xb370cf00: pop    %eax
  0xb370cf01: pop    %edx
  0xb370cf02: add    %edx,%eax
  0xb370cf04: movzbl 0x1(%esi),%ebx
  0xb370cf08: inc    %esi
  0xb370cf09: jmp    *-0x48ecd6a0(,%ebx,4)
  0xb370cf10: add    %al,(%eax)
  0xb370cf12: add    %al,(%eax)
  // ...
```

细心的读者可以发现，这里包含 3 段机器指令，分别对应 JVM 的 3 条字节码指令：iload_0、iload_1 和 iadd。这些信息是 JVM 打印出来的，当 JVM 分别解释执行 iload_0、iload_1 和 iadd 这些字节码指令时，实际上最终执行了其对应的一大串机器码。

实际上 JVM 会将其 200 多个字节码指令所对应的机器码全部打印出来，但是本书限于篇幅，不可能将其全部显示在这里，因此仅挑选 iload_0、iload_1 和 iadd 这 3 条字节码指令。有兴趣的读者可以研究研究这 3 个字节码指令所对应的机器指令，当然本书后面也会详细分析这些指令的含义。在这里我们先不用关心这些机器指令到底代表什么逻辑，只要知道 JVM 在调用 Java

see more please visit: https://homeofpdf.com

程序时，最终其实执行的是机器指令就可以了。

经过对以上这段示例程序的分析可知，在 JVM 内部一定存在一个“边界”，边界外面是 C 程序，边界里面则直接跳转到机器码。换言之，C 程序可以直接调用汇编指令。

可是即使你翻遍有关 C 语言的教科书，也几乎不会发现有哪本教科书会介绍 C 语言这种能够直接执行机器指令的功能。

似乎是毫无头绪。想当年詹爷也一定在技术路径上有过类似的迷茫和彷徨，不知何去何从（个人瞎掰，若果猜错，请詹爷原谅）。

当然，也许有高手会说，JVM 一开始压根儿不是模板解释器，而是纯粹的字节码解释器，其会将字节码指令逐条翻译成 C 程序。这种说法是完全正确的，但是由于效率实在太低，低到为 C++程序员们所不齿，所以 JVM 没发布几个版本就将默认的字节码解释器换成模板解释器了，JVM 执行 Java 函数时直接执行机器指令。但是本书并不会纠结于这些细节，本书主要以启发式的方法来剖析 JVM 内部的运行机制，注重的是主要的思路逻辑，剖析 JVM 为什么会这样做，为什么会使用这样的技术和框架。任何事情都有其必然的道理，如果你对 JVM 能够做到“知其然”，那你已经相当牛了；但是如果你能够对 JVM 做到“知其所以然”，那么你将达到超脱的境界。你可以任意定制自己的 JVM 了，整个底层世界完全由你操控，这是一件想想就激动的事（至少我本人就是这样，当我做了很多次试验终于弄明白了 JVM 的执行引擎后，兴奋得好多天睡不着觉）。当然如果有读者是细节控，那么也请忍一忍啦。

言归正传，所幸的是，C 语言有一个秘密武器，而这种秘密武器就是使 C 语言跨越 IT 界几十年历史长河却依然屹立不倒并一直非常流行的根本和基础，那就是指针。当然，这也是 JVM 得以最终成功的关键因素和关键技术。

可以说，任何底层编程开发，只要祭出“指针”这门神功或者武器，就没有 C 语言搞不定的事（因为能够访问内存，几乎能做机器指令所做的大部分事），所以它一直很流行，各种底层软件、驱动程序、操作系统、关系数据库、办公软件、手机操作系统，等等，几乎都是用它开发而成。同时，相比于汇编，C 语言在语法上更加人性化，更易于被人类接受和理解，因此 C 语言广泛取代了汇编语言，只有在对性能要求非常苛刻的一些领域才需要“祭出”汇编这种最原始、古朴，当然威力也最强大的“神器”，例如视频软件的内存访问或者操作系统底层的原子同步。JVM 内部的原子操作也基本全部采用汇编指令来实现。

那么下面就有请 C 语言的“指针”隆重上场！大部分做 C 程序开发的同学对 C 语言的指针变量都不陌生，例如 int*、char*，等等，你可以将 char*理解成一个字符串的开始地址，也可以将其理解成一个二维字符数组的首地址。同理，既可以将 int*理解成一个整型变量的内存地址，也可以将其理解成一个二维整型数组的首地址。

see more please visit: https://homeofpdf.com

而 C 语言的指针其实还有一种更重要的用法——函数指针。通过函数指针，C 语言可以将一个变量直接指向一个函数的首地址。C 语言被编译时，C 函数将被直接编译成机器指令，而这个函数指针将直接指向这段机器指令的首地址。于是聪明的人类便打了个擦边球，在源代码编码阶段就定义好一段机器指令，然后直接将一个 C 函数指针指向这段机器指令的首地址，从而间接实现 C 语言直接调用机器指令的目的（其实，C 语言还有一种办法可以调用汇编指令，那就是内嵌汇编的方式，在 Linux 操作系统内核中就有大量的这种用法。但是这种用法与 JVM 所要实现的目标稍微有点不同，JVM 要实现直接由 C 语言调用机器指令，而内嵌汇编的方式只实现了这一目标的一半，内嵌汇编的方式只能实现由 C 语言直接调用汇编指令。（注意：汇编指令与机器指令之间还有很大差距）。下面来看一个 C 语言直接调用机器指令的示例：

清单：C 程序示例

作用：演示 C 函数直接调用机器指令

```
//下面这段字符串是一组机器码，使用十六进制表示。这段机器码表示一个函数，能够对传入的参数执行增 1 操作
//注：下面的 xe9 表示十六进制数，等同于 0xe9
//同时 xe9 前面的斜杠必须是反斜杠\，而不能是正斜杠/
const unsigned char code[] =
    "\xe9\x07\x00\x00\x00\xcc\xcc\xcc\xcc\xcc\xcc"\
    "\xcc\x55\x8b\xec\x83\xec\x40\x53\x56\x57\x8d"\
    "\x7d\xc0\xb9\x10\x00\x00\x00\xb8\xcc\xcc\xcc"\
    "\xcc\xf3\xab\x8b\x45\x08\x83\xc0\x01\x5f\x5e"\
    "\x5b\x8b\xe5\x5d\xc3";

int main(){
    int result;

    //定义函数指针 fun，其指向某个函数的入口地址
    int (*fun)(int);

    //初始化函数指针，将其指向 code 机器码的入口，这段机器码其实表示一个函数
    fun=(void*) code;

    //调用函数
    result=fun(7);

    printf("result=%d/r/n",result);
}
```

在本示例中，定义了一个全局数组 code，code 数组里保存的若干字符就是机器指令，这些机器指令能够对入参执行自增操作。

在 main()函数中通过 int (*fun)(int)定义了一个指针函数 fun，接着通过 fun = (void*)code 将该指针函数指向一个内存地址，就是 code 数组的首地址，最后通过 result = fun(7)就能调用这个

see more please visit: https://homeofpdf.com

指针函数了。当这段 C 程序被编译后，fun()实际上就指向了 code 数组的内存首地址。当物理机器加载这段编译后的程序，当执行到 result = fun(7)这条指令时，就会将 CS:IP 段寄存器指向 code 首地址，从而将 code 数组所在的这一连续内存区域当作代码段来执行。

这就是 JVM 内部能够直接由 C 程序调用和执行机器指令的奥秘所在。其实，这便是本书第 1 章 2.3.3 节所讲的内容，只不过那时并没有直接从 JVM 的角度分析问题。

在 JVM 内部也有这么一个函数指针，就是 call_stub。这个函数指针正是 JVM 内部 C 语言程序与机器指令的分水岭，JVM 在调用这个函数之前，主要执行 C 程序（其实还是 C 程序编译后的机器指令），而 JVM 通过这个函数指针调用目标函数之后，就直接进入了机器指令的领域。

call_stub 函数指针在 JVM 内部具有举足轻重的意义，在 Java 程序的执行过程中，会有很多地方涉及直接从 JVM 内部调用 Java 函数，例如执行 Java 程序主函数，或者类加载。本章通过介绍函数指针的执行原理，使你更加深入地理解 JVM 跨越 C/C++程序而调用 Java 函数的机制。

call_stub 函数指针原型定义在 stubRoutines.hpp 文件中，定义如下：

清单：/src/share/vm/runtime/stubRoutines.hpp
作用：函数原型定义

```
static CallStub call_stub()
{
    return CAST_TO_FN_PTR(CallStub, _call_stub_entry);
}
```

在这段代码里出现了一个宏：CAST_TO_FN_PTR。在 JVM 中，凡是出现函数名大写的情况，基本都是宏。事实上，这基本也是 C/C++语言的一种规范。

CAST_TO_FN_PTR 宏定义在 globalDefinitions.hpp 文件中，定义如下：

```
#define CAST_TO_FN_PTR(func_type, value) ((func_type)(castable_address(value)))
```

将 call_stub()函数体按照这个宏进行替换和展开（C 程序中的宏会在预编译阶段被替换），最终得到如下函数定义：

清单：/src/share/vm/runtime/stubRoutines.hpp
作用：函数原型定义

```
static CallStub call_stub()
{
    return (CallStub)(castable_address(_call_stub_entry));
}
```

现在这种形式又代表了什么含义呢？为了理解这个问题，我们就需要对 C 程序中的函数指针有深入的了解。

see more please visit: https://homeofpdf.com

不过在讲解之前，先对本节内容做一个简单的总结：Java 字节码指令直接对应一段特定逻辑的本地机器码，而 JVM 在解释执行 Java 字节码指令时，会直接调用字节码指令所对应的本地机器码。JVM 是使用 C/C++编写而成的，因此 JVM 要直接执行本地机器码，便意味着必须要能够从 C/C++程序中直接进入机器指令。这种技术实现的关键便是使用 C 语言所提供的一种高级功能——函数指针。通过函数指针能够直接由 C 程序触发一段机器指令。

在 JVM 内部，call_stub 便是实现 C 程序调用字节码指令的第一步——例如 Java 主函数的调用。在 JVM 执行 Java 主函数所对应的第一条字节码指令之前，必须经过 call_stub 函数指针进入对应的例程，然后在目标例程中触发对 Java 主函数第一条字节码指令的调用。

2.3 函数指针

在正式解密 call_stub()之前，有必要讲一下 C 语言中的函数指针概念，只有明白了什么是函数指针，如何调用函数指针，才能真正看懂 call_stub()。函数指针是 C/C++语言里一种高级的变量和应用，可能很多做应用开发的 C/C++程序员平时也很少接触这种用法，因此大家不妨在这里稍微花点时间看一看。首先需要区分两个概念：函数指针与指针函数。

1. 函数指针与指针函数

在 C 程序中，有两种概念容易混淆，那就是函数指针与指针函数。例如下面定义了一个指针函数：

```
int *fun(int a, int b);
```

而下面则定义了一个函数指针：

```
// 声明一个函数指针 fun
void (*fun)(int a, int b);

// 声明一个函数原型 add
void add(int x, int y);

//为函数指针赋值：将 add()函数首地址赋值给 fun 指针
fun = add;
```

仔细观察指针函数和函数指针的声明语法，可以发现这两者的一个主要区别：

如果函数名称前面的指针符号*没有被括号包含，则所定义的就是指针函数；如果被括号包含，则所定义的就是函数指针。

◎　在 int *fun(int a, int b)这行声明中，fun 前面的指针符号*没有包含在括号内，所

see more please visit: https://thomeofpdf.com

以这行代码就是在定义一个指针函数。

◎ 在 void (*fun)(int a, int b)这行声明中，fun 前面的指针符号*被包含在括号内，所以这行代码就是在声明一个函数指针。

这就是指针函数与函数指针两者之间在形式上的差别。

指针函数与函数指针在形式上只有微小的差别，但是在内容和含义上差别就大了。通俗地讲，指针函数与函数指针在内容上的含义分别如下：

◎ 指针函数声明的是一个函数，与一般的函数声明并无多大区别，唯一有区别的就是指针函数的返回类型是一个指针，而一般的函数声明所返回的则是普通变量类型。

◎ 函数指针声明的是一个指针，只不过这个指针与一般的指针不同，一般的指针指向一个变量的内存地址，而函数指针则指向一个函数的首地址。

所以，在 int *fun(int a, int b)这行声明中，fun 实际上就是一个普通的函数，这个函数包含 2 个入参，类型都是 int。同时这个函数返回一个 int*类型的指针。

在 void (*fun)(int a, int b)这行声明中，fun 实际上是一个指针变量，注意，是变量，不是函数。这行声明指出，fun 指针指向了一个函数，所指向的这个函数必须包含 2 个入参，类型都是 int。同时，fun 指针所指向的函数必须有一个 void 类型的返回值。

下面分别给出指针函数和函数指针的一个示例。

示例一：指针函数

清单：示例程序

作用：C 程序指针函数示例

```
#include<stdio.h>

// 声明指针函数
int *add(int a, int b);

int main(){
    int a = 5;
    int b = 3;

    // 调用指针函数
    int *c=add(a, b);
    printf("c = %p\n", c);

    return 0;
```


see more please visit: https://homeofpdf.com

```
}

// 定义指针函数
int *add(int a, int b){
    int c=a+b;

    // 这里返回变量 c 的内存地址
    return &c;
}
```

本程序声明了一个指针函数 add()，包含两个 int 类型的入参，同时返回 int*类型的指针。在 main()主函数调用时，使用与 add()返回值类型相同的指针类型 int *c 变量来接收 add()函数所返回的值，并最终打印出返回的指针的值。这个值实际上就是 add()函数内部变量 c 的内存地址。打印内容如下：

```
c = 0xbf9932e4
```

示例二：函数指针

清单：示例程序

作用：C 程序函数指针示例

```
#include<stdio.h>

// 声明函数指针 addPointer
int (*addPointer)(int a, int b);

// 声明一个普通函数 add
int add(int a, int b);

int main(){
    int a = 5;
    int b = 3;

    // 初始化函数指针 addPointer，将其指向 add()函数首地址
    addPointer = add;

    // 调用函数指针所指向的函数
    int c=addPointer(a, b);
    printf("c=%d\n", c);

    return 0;
}

int add(int a, int b){
    int c=a+b;
    return c;
```


see more please visit: https://homeofpdf.com

```
}
```

在本例中，定义了一个函数指针 addPointer，并将其指向了函数 add()，然后就可以通过 int c=addPointer(a, b)这样的形式，像调用普通函数一样，通过函数指针来调用其所指向的函数。由于 addPointer 指针指向了函数 add()，因此程序最终实际上调用的就是 add()函数。下面会讲到通过函数指针来调用函数的几种形式。

2. 函数指针的两种定义方式

函数指针通常可以通过两种方式进行声明。

1）方式一：直接声明

```
return_type (*func_pointer)( data_type arg1,
                             data_type arg2,
                             ...,
                             data_type argn);
```

注意：在定义函数指针时，指针运算符*一定要使用括号括起来，否则就不是定义函数指针了，而是变成定义指针函数。这一点切记。

使用这种方式定义的函数指针就相当于直接定义了一个变量，可以直接对该变量进行赋值。函数指针与普通变量一样，可以在函数外面声明，作为全局变量，也可以声明在函数内部作为局部变量。例如上面在 main()函数外面所声明的 addPointer 这一函数指针，其实也可以声明在 main()函数内部。将上面的例子改成如下：

清单：示例程序

作用：C 程序函数指针示例

```
#include<stdio.h>

// 声明一个普通函数 add
int add(int a, int b);

int main(){
    int a = 5;
    int b = 5;

    // 在 main()函数内部声明函数指针 addPointer
    int (*addPointer)(int a, int b);

    // 初始化函数指针 addPointer，将其指向 add()函数首地址
    addPointer = add;

    // 调用函数指针所指向的函数
```


see more please visit: https://homeofpdf.com

```
    int c=addPointer(a, b);
    printf("c=%d\n", c);

    return 0;
}

int add(int a, int b){
    int c=a+b;
    return c;
}
```

在本例中，addPointer 函数指针被声明在了 main()函数内部，此时它就变成了局部变量。这也是函数指针与指针函数的区别之一，指针函数毕竟本质上仍然是一个函数，而函数是不可以被声明在其他函数内部的。

2）方式二：通过类型声明

函数指针还有一种声明的方式，那就是先定义一个类型，然后通过所定义的类型去声明函数指针。

```
typedef (*func_pointer)( data_type arg1,
                         data_type arg2,
                         ...,
                         data_type argn);
```

注意：这里不是在声明函数指针，而是定义了一种函数指针的类型。这种类型的类型名就是 func_pointer。

类型声明的方式和直接声明的方式相比，主要区别在于，通过类型声明的方式定义的仅仅是一个类型，由这个类型无法直接去初始化函数指针，因为类型不是变量。例如，在这里定义了 func_pointer 这个类型后，并不能对其进行初始化并将其指向某个函数。要想使用这种类型，就必须使用类型名去声明一个函数指针变量，然后才能为所声明的函数指针变量赋值。

还是拿刚才的 addPointer 举例，说明如何通过类型定义的方式来声明函数指针：

清单：示例程序
作用：C 程序函数指针示例

```
#include<stdio.h>

// 声明一个普通函数 add
int add(int a, int b);

int main(){
    int a = 5;
    int b = 3;
```


see more please visit: https://homeofpdf.com

```
    // 定义 addPointerType 这一类型
    typedef (*addPointerType)(int a, int b);

    //声明一个 addPointerType 类型的变量
    addPointerType addPointer;

    // 初始化函数指针 addPointer，将其指向 add()函数首地址
     addPointer = add;

    // 调用函数指针所指向的函数
     int c=addPointer(a, b);
     printf("c=%d\n", c);

     return 0;
}

int add(int a, int b){
     int c=a+b;
     return c;
}
```

在本例中，先通过 typedef (*addPointerType)(int a, int b)定义了一种类型，然后再声明了一个变量 addPointer，变量 addPointer 就属于 addPointerType 这种类型。其实，addPointerType 就类似于 C 程序中的 int、char 等基本类型，只不过 int、char 等基本类型是内建的，是 C 程序本来就支持的类型，而 addPointerType 则是开发者自定义的一种类型。

如果使用 Java 程序来举例， addPointerType 好比是 Java 开发者自定义的一个 Java 类，定义好 Java 类后，就可以声明属于这种 Java 类型的变量，并对变量进行初始化。

在 C 语言中，通过类型定义的方式来声明函数指针，是一种比较普遍的做法。JVM 中的 call_stub 函数指针便是通过这种方式来声明的。下面对该函数指针进行详细的剖析。

了解了函数指针的声明方式后，还需要了解函数指针的常用方式。

3．函数指针的两种调用方式

函数指针有两种基本调用方式。

假设已经声明并定义一个函数指针 funcPointer，则可以通过如下两种格式来调用：

◎ (*funcPointer)(参数列表)

◎ funcPointer(参数列表)

第二种格式，看起来好像 funcPointer 就是一个普通的函数名，如果不看其定义方式，根本

see more please visit: https://homeofpdf.com

看不出来这到底是一个指针，还是一个函数。

第一种格式看起来比较古怪，但是有相当一部分人喜欢这么使用。之所以使用这种格式，是因为这种格式可以让人知道这是在通过指针而非函数进行函数调用。

还以上面的 addPointer 函数指针为例，下面分别给出这个指针的两种调用方式。在上面的例子中，通过第二种格式进行调用，如下：

清单：示例程序

作用：C 程序函数指针示例

```
int add(int a, int b);

int main(){
    //…

    int (*addPointer)(int a, int b);
     addPointer = add;

    // 调用函数指针所指向的函数
    int c=addPointer(a, b);
    //…
}
```

本例通过 addPointer(a, b)这种格式调用函数指针。

清单：示例程序

作用：C 程序函数指针示例

```
int add(int a, int b);

int main(){
    //…

    int (*addPointer)(int a, int b);
     addPointer = add;

    // 调用函数指针所指向的函数
    int c=(*addPointer)(a, b);
    //……
}
```

本例通过(*addPointer)(a, b)这种格式调用函数指针。

在 JVM 中，我们即将分析的 call_stub 函数指针便通过这种格式进行调用。

总体而言，函数指针作为 C 语言中的高级应用，是实现 C 语言动态扩展能力的关键技术之一，如同 Java 中的反射与类动态加载技术。

see more please visit: https://homeofpdf.com

函数指针通常有两种定义方式，一种是像定义普通变量一样定义，一种是通过 typedef 关键字进行类型声明。

函数指针本质上是一种指针，不是函数，但是由于这种指针指向的并不是某个变量的内存地址，而是某个函数的内存首地址，因此 C 语言允许像调用函数一样调用函数指针。要注意的是，普通的指针是不能被当成函数去调用的，这便是函数指针的奇特之处——与普通的指针变量在概念上完全一致但是能够被当成函数进行调用，而与函数在调用上具有相同的方式但是在概念上却有巨大的差异。

函数指针通常有两种调用方法，在 JVM 内部，使用了其中一种比较奇特的调用方式。如果不了解这种奇特的调用方式，很难理解 JVM 内部的实现机制。

2.4 CallStub 函数指针定义

上一节介绍了 C 语言中的函数指针与指针函数的概念、区别、声明方式及调用格式。现在再次将目光聚焦到 JVM 的 call_stub 指针上，这种指针其实就是函数指针。前面讲过，call_stub 函数指针在 JVM 内部具有举足轻重的作用，例如 Java 程序主函数的调用链路就必须经过 call_stub 函数指针的执行，本书后面章节在讲解类加载机制时会提到，Java 类的加载链路也会经过该函数指针。因此，对于 JVM 执行引擎而言，该函数指针无比重要，没有这个指针的存在，JVM 执行引擎便无从谈起。因此本章将会详细讲解 call_stub 的调用机制。

上面讲过，将下面这句代码：

```
return CAST_TO_FN_PTR(CallStub, _call_stub_entry);
```

进行宏替换后，得到下面这行展开式：

```
return (CallStub)(castable_address(_call_stub_entry));
```

这里的 CallStub 其实就是一种自定义的类型。先不用管 CallStub 究竟是怎样的一种类型，为了将问题简化，可以将其想象成最简单的一种类型，例如 int，使用 int 这种基本类型替换 CallStub，就得到下面的替换式：

```
return (int)(castable_address(_call_stub_entry));
```

这么一替换，这种形式立刻就变成司空见惯的形式，castable_address(_call_stub_entry)返回了一个结果类型，JVM 又将这种类型转换成了 int 类型。

现在一起看看 CallStub 究竟是怎样的一种类型。CallStub 定义在/src/share/vm/runtime/stubRoutines.hpp 文件中，声明如下：

see more please visit: https://homeofpdf.com

清单：/src/share/vm/runtime/stubRoutines.hpp

作用：类型定义

```
// Calls to Java
typedef void (*CallStub)(
        address   link,
        intptr_t* result,
        BasicType result_type,
        methodOopDesc* method,
        address   entry_point,
        intptr_t* parameters,
        int       size_of_parameters,
        TRAPS
);
```

由该定义可知，CallStub 是这样的一种函数指针类型：其指向的函数，返回值类型是 void，并且有 8 个入参。

到这里，call_stub()函数调用的方式基本理顺了。但是，前文在讲解函数指针时曾提到函数指针的两种调用方式，而在 call_stub()里似乎没有看到任何一种格式的调用。我们先看看 JVM 内部是如何调用 call_stub()的。JVM 在 javaCalls::call_helper()中执行了 call_stub()函数调用，其调用的源代码如下：

清单：/src/share/vm/runtime/javaCalls.cpp::call_helper()

作用：演示 JVM 内部调用 call_stub()

```
void JavaCalls::call_helper(JavaValue* result,
                            methodHandle* m,
                            JavaCallArguments* args,
                            TRAPS) {

    //…

    // call_stub()调用开始
    // do call
      { JavaCallWrapper link(method, receiver, result, CHECK);
        { HandleMark hm(thread);  // HandleMark used by HandleMarkCleaner

          StubRoutines::call_stub()(
                  (address)&link,
                  // (intptr_t*)&(result->_value),
                  result_val_address,
                  result_type,
                  method(),
                  entry_point,
                  args->parameters(),
                  args->size_of_parameters(),
```

see more please visit: https://homeofpdf.com

```
                    CHECK
            );

             // circumvent MS C++ 5.0 compiler bug (result is clobbered across call)
            result = link.result();
            // Preserve oop return value across possible gc points
            if (oop_result_flag) {
                    thread->set_vm_result((oop) result->get_jobject());
            }
        }
    }
    // call_stub()调用结束

    //…
}
```

JVM 内部直接调用了 call_stub()，并且传入了 8 个参数，可是 call_stub()函数原型却是没有入参的（如下所示，其实上文已经贴过一次了，不过这里为了阐述问题，再贴一次）。这又是怎么回事呢？

```
static CallStub call_stub()
{
return (CallStub)(castable_address(_call_stub_entry));
}
```

（注意：在 call_stub()的函数原型声明里，并没有入参，而 JVM 在调用 call_stub()函数时却传入了 8 个参数。）

其实，这里 JVM 隐式地调用了函数指针。call_stub()函数最终返回的是一个函数指针的实例变量（C 语言中没有实例的概念，这里借用下 Java 的实例概念，意在强调这是一个初始化了的变量）。虽然 call_stub()的原型函数里只有 return (CallStub)(castable_address(_call_stub_entry)) 这一行代码，可是这行代码所蕴含的逻辑却十分丰富，编译器编译后，这行代码会被转换为类似下面的形式：

清单：/src/share/vm/runtime/stubRoutines.hpp

作用：函数原型定义

```
static CallStub call_stub()
{
    // 使用 CallStub 类型声明一个函数指针变量
    CallStub functionPointer;

    // 调用 castable_address()，并用 address 类型的变量 returnAddress 接收返回值
    // castable_address()函数返回的变量类型就是 address，这个后面马上就会讲到
    address returnAddress = castable_address(_call_stub_entry);
```


see more please visit: https://homeofpdf.com

```
    // 将 address 类型的变量转换为 CallStub 类型的变量
    functionPointer = (CallStub)returnAddress;

    // 返回 CallStub 类型的变量
    return functionPointer;
}
```

高级语言之所以高级，就是人类往往能够只使用简短的几行代码（甚至一行）就能表达出十分丰富的含义，一切含义的表达皆由编译器在幕后默默支撑。当然，编译器并不会真的将 C 源程序转换成上面这段代码，只不过编译器会自动生成中间变量，最终所表达出来的逻辑就与上面这段代码相同。

在这段代码里，我们发现，其实在 C 程序里通过 typedef 所自定义的类型，也可以参与类型强制转换的计算。在本例中，castable_address(_call_stub_entry)返回的其实是 address 这种自定义的类型，而编译器最终将其转换成了 CallStub 这种自定义的类型。

call_stub()最终返回一个 CallStub 类型的函数指针变量，调用者其实就可以像调用普通函数那样来调用这种函数指针变量。只不过 JVM 并没有显式地调用函数指针，而是隐式地进行了调用（即偷偷摸摸、不声不响地，-_-）。但是物理机器是不会理解这种隐式调用的，最终还是要靠编译器来实现隐式调用到显式调用的转变。编译器处理后的显式调用可以用下面这段逻辑表达：

清单：/src/share/vm/runtime/javaCalls.cpp::call_helper()

作用：演示 JVM 内部 call_stub()的显式调用

```
void JavaCalls::call_helper(JavaValue* result,
                            methodHandle* m,
                            JavaCallArguments* args,
                            TRAPS) {

    //…

    // call_stub()调用开始
    // do call
      { JavaCallWrapper link(method, receiver, result, CHECK);
        { HandleMark hm(thread);  // HandleMark used by HandleMarkCleaner

          // 先声明一个 CallStub 类型的函数指针变量
          CallStub funcPointer;

          // 初始化函数指针变量，将其指向 call_stub()函数首地址
          funcPointer = StubRoutines::call_stub();

          // 调用函数指针变量，传入 8 个参数
          funcPointer(
```

see more please visit: https://homeofpdf.com

```
                    (address)&link,
                    // (intptr_t*)&(result->_value),
                    result_val_address,
                    result_type,
                    method(),
                    entry_point,
                    args->parameters(),
                    args->size_of_parameters(),
                    CHECK);

            // circumvent MS C++ 5.0 compiler bug (result is clobbered across call)
            result = link.result();
            // Preserve oop return value across possible gc points
            if (oop_result_flag) {
                    thread->set_vm_result((oop) result->get_jobject());
            }
        }
    }
  // call_stub()调用结束

  //…
}
```

这样一分解，JVM 实现 call_stub()的机制就非常清晰了（其实现在再说调用 call_stub()，大家都知道这种说法是错误的，因为实际上 JVM 调用的是 call_stub()函数所返回的函数指针变量），相信一般人都能看得很明白。由于 JVM 在使用 typedef 定义 CallStub 类型时，就规定这种函数指针类型有 8 个入参，因此 JVM 最终在调用这种类型的函数指针时，也必须传入 8 个类型完全相同的参数。

1. castalbe_address()

上文在讲解函数指针的声明时谈到，定义了一种函数指针的类型后，可以去声明属于这种类型的一个函数指针变量，并可以初始化这个变量，使其指向某个函数。

注意：上面对函数指针的初始化，都是将其指向某个函数，而实际上，由于函数指针也是一种指针（如同海马也是马），而指针的特点就是可以指向内存的任意地址，既可以指向函数的首地址，也可以指向某个变量的首地址，说白了，其实指针里无非就是存储了一个值而已。所以，函数指针也是既能指向函数首地址，也能指向某个变量首地址，或者指向任何你想指向的内存地址。

而 call_stub()函数内部其实就是让函数指针指向了某个内存地址。

see more please visit: https://homeofpdf.com

call_stub()函数宏展开后的逻辑是：

```
return (CallStub)(castable_address(_call_stub_entry));
```

这里的 castable_address 是一个函数，定义在 globalDefinitions.hpp 文件中，其定义如下：

```
inline address_word  castable_address(address x)
{
   return address_word(x) ;
}
```

这里的 address_word 也是一种自定义类型，顾名思义，它表示的是一种地址类型。该类型在 globalDefinitions.hpp 中定义，定义如下：

```
typedef uintptr_t     address_word;
```

由此可知，address_word 类型其实是 uintptr_t，而后一种类型也是 JVM 自定义的类型。但是这种类型是平台相关的，所以在 JVM 内部有 3 处定义了这种类型，分别是：

◎ globalDefinitions_gcc.hpp

◎ globalDefinitions_sparcWorks.hpp

◎ globalDefinitions_visCPP.hpp

在特定的平台上编译 JVM 时，编译器会自动根据平台类型，编译不同的 hpp 头文件。这里所列出的 3 种头文件，只看名字就能知道，分别对应的是 Linux、Macintosh 和 Windows 这 3 种主流的操作系统。这里之所以列出这 3 种文件，并不是想让大家都去研究一番，而是让大家感受一下，如果使用 C/C++语言编程，程序员必须要考虑平台架构的异构性，并且必须在代码中处理平台的兼容性。当年詹爷也是受够了这种麻烦，所以才会搞出个 Java 跨平台的编程语言出来。所以，Java 程序员是幸福的。

在 Linux 平台上，uintptr_t 类型的定义如下（globalDefinitions_gcc.hpp 文件中）：

```
typedef unsigned int            uintptr_t;
```

根据这个定义可知，uintptr_t 在 Linux 平台上的类型原型是 unsigned int，这是 C 语言的基本类型之一，即无符号整数类型。于是，绕了一小圈，终于得知 address_word 这种类型的“庐山真面目”。

让我们再把目光转回到 call_stub()函数，将 address_word 类型进行替换，得到如下代码：

```
static CallStub call_stub()
{
// 下面的代码原本是这一句：return (CallStub)(castable_address(_call_stub_entry));
   return (CallStub)(unsigned int(_call_stub_entry));
}
```


see more please visit: https://homeofpdf.com

到了这一步，call_stub()函数的逻辑基本全部还原出来了，call_stub()函数的逻辑总结起来包含下面两步：

◎ 第一步，将_call_stub_entry 变量转换为 unsigned int 这一基本类型。

◎ 第二步，将_call_stub_entry 所转换后的 unsigned int 这一基本类型再转换为 CallStub 这一自定义类型，该类型是函数指针类型。

那么_call_stub_entry 是啥？是啥类型？该类型定义在 StubRoutines.hpp 中，定义如下：

```
static address _call_stub_entry;
```

由此可知，_call_stub_entry 本身就是 address 类型，而该类型的原型是 unsigned int。

再次回顾下 JVM 调用 Java 函数的过程：

◎ JVM 先调用 call_stub()函数，该函数将_call_stub_entry 这一 unsigned int 类型的变量转换成 CallStub 自定义的类型，该类型是函数指针。

◎ JVM 将 call_stub()所返回的函数指针当成函数进行调用。

在这个过程中，似乎还有件事不明确，那就是函数指针的指向。在 call_stub()里是直接返回了_call_stub_entry 这一基本类型的变量，然后直接就被转换成了 CallStub 这一自定义的函数指针类型，接着 JVM 直接就调用该函数指针了，自始至终并没有看到函数指针被指向了哪个函数。而在前文讲解函数指针时提到，要调用函数指针，必须将其指向某个函数。

其实，JVM 在初始化的过程中，便将_call_stub_entry 这一变量指向了某个内存地址，否则 JVM 肯定无法直接调用。在 x86 32 位 Linux 平台上，JVM 在初始化过程中，存在这样一条链路：

```
java.c: main()
    java_md.c: LoadJavaVM()
     jni.c: JNI_CreateJavaVM()
         Threads.c: create_vm()
             init.c: init_globals()
                 StubRoutines.cpp: stubRoutines_init1()
                     StubRoutines.cpp: initialize1()
                         stubGenerator_x86_x32.cpp: StubGenerator_generate()
                             stubGenerator_x86_x32.cpp: StubCodeGenerator()
                                 stubGenerator_x86_x32.cpp: generate_initial()
```

这条链路从 JVM 的 main()函数开始，调用到 init_globals()这个全局数据初始化模块，最后再调用到 StubRoutines 这个例程生成模块，最终在 stubGenerator_x86_32.cpp: generate_initial()函数中会执行如下代码，对_call_stub_entry 这个变量进行初始化，如下所示：

see more please visit: https://homeofpdf.com

清单：/src/share/vm/runtime/stubGenerator_x86_x32.cpp

作用：函数原型定义

```
void generate_initial() {
        // Generates all stubs and initializes the entry points

        StubRoutines::_call_stub_entry  =
              generate_call_stub(StubRoutines::_call_stub_return_address);

    // …
}
```

最终，JVM 调用 generate_call_stub(StubRoutines::_call_stub_return_address)函数返回一个值，赋值给_call_stub_entry。generate_call_stub(StubRoutines::_call_stub_return_address)里面的逻辑十分重要，可以说是 JVM 最核心的功能。不过要分析清楚这个最核心的功能模块，得先弄清楚 call_stub()的入参。

2. CallStub()入参

准确地说，CallStub 并不是函数，因此本节标题写为“CallStub()入参”不够严谨，但是 CallStub 是一个函数指针，最终 JVM 也是通过这一函数指针调用其所指向的函数的，所以将其说成是函数也不为过。为简单起见，我们统一说成是“CallStub()入参”。

在 javaCalls.cpp::call_helper()函数中，JVM 是这样调用 CallStub()函数的：

清单：/src/share/vm/runtime/javaCalls.cpp::call_helper()

作用：演示 JVM 内部调用 call_stub()

```
StubRoutines::call_stub()(
            (address)&link,
            // (intptr_t*)&(result->_value),
            result_val_address,
            result_type,
            method(),
            entry_point,
            args->parameters(),
            args->size_of_parameters(),
            CHECK
    );
```

（注：为节省篇幅，不再贴出这段代码的上下文。）

JVM 一共传入了 8 个参数，这 8 个参数的含义如表 2.1 所示。

see more please visit: https://tomeofpdf.com

表 2.1　8 个参数说明

参 数 名	含　义
link	顾名思义，这是一个“连接器”，注意，不是“链接器”
result_val_address	函数返回值地址
result_type	函数返回类型
method()	JVM 内部所表示的 Java 方法对象
entry_point	JVM 调用 Java 方法的例程入口。JVM 内部的每一段例程都是在 JVM 启动过程中预先生成好的一段机器指令。要调用 Java 方法，都必须经过本例程，即需要先执行这段机器指令，然后才能跳转到 Java 方法字节码所对应的机器指令去执行
parameters()	Java 方法的入参集合
size_of_parameters()	Java 方法的入参数量
CHECK	当前线程对象

这些参数每一个都非常重要，下面先进行一个简单介绍。要想深入了解这些参数背后的意义，必须要等到研究完 Java 的内存模型之后。本书在讲解 Java 的内存模型时，会对这些入参进行深入分析。

这里先简单介绍 5 个参数：link、method()、entry_point、parameters()和 size_of_parameters()。

1）连接器 link

连接器 link 的作用，从其名称也可猜测一二，就是起到连接、桥梁的作用。连接谁呢？这要从 link 的类型定义来说。连接器 link 所属类型是 JavaCallWrapper，该类型定义在 javaCalls.cpp 文件中，定义如下：

清单：/src/share/vm/runtime/javaCalls.cpp::JavaCallWrapper
作用：JavaCallWrapper 类定义

```
// A JavaCallWrapper is constructed before each JavaCall and destructed after
the call.
// Its purpose is to allocate/deallocate a new handle block and to save/restore
the last
// Java fp/sp. A pointer to the JavaCallWrapper is stored on the stack.

class JavaCallWrapper: StackObj {
    friend class VMStructs;
    private:
    JavaThread*      _thread;        // the thread to which this call belongs
    JNIHandleBlock*  _handles;       // the saved handle block
    methodOop        _callee_method; // to be able to collect arguments if
```


see more please visit: https://homeofpdf.com

```
entry frame is top frame
    oop              _receiver; // the receiver of the call (if a non-static call)

    JavaFrameAnchor  _anchor; // last thread anchor state that we must restore

    JavaValue*       _result;                // result value

};
```

这是一个 C++类型，这个类里面包含这样几个变量：

◎ _thread，当前 Java 函数所在线程

◎ _handles，本地调用句柄

◎ _callee_method，调用者方法对象

◎ _receiver，被调用者（非静态 Java 方法）

◎ _anchor，Java 线程堆栈对象

◎ _result，Java 方法所返回的值

通过这些变量可知，link 其实在 Java 函数的调用者与被调用者之间搭建了一座桥梁，通过这座桥梁，我们可以实现堆栈追踪，可以得到整个方法的调用链路。

在 Java 函数调用时，link 指针将被保存到当前方法的堆栈中。

注意：JVM 内部有一个 linker，是链接器，与 link 是不同的两个对象。

2）method()

method()是当前 Java 方法在 JVM 内部的表示对象。

每一个 Java 方法在被 JVM 加载时，JVM 都会在内部为这个 Java 方法建立函数模型，说白了就是保存一份 Java 方法的全部原始描述信息。JVM 为 Java 方法所建立的模型中至少包含以下信息：

◎ Java 函数的名称、所属的 Java 类

◎ Java 函数的入参信息，包括入参类型、入参参数名、入参数量、入参顺序等

◎ Java 函数编译后的字节码信息，包括对应的字节码指令、所占用的总字节数等

◎ Java 函数的注解信息

◎ Java 函数的继承信息

◎ Java 函数的返回信息

JVM 在调用 CallStub()函数指针时，将 method()对象传递进去，最终就是为了从 method()对象中获取到 Java 函数编译后的字节码信息，JVM 拿到字节码信息之后，就能对字节码进行解

see more please visit: https://homeofpdf.com

释执行了。

注：大家通过 Java 反射对象不仅能够获取一个 Java 类的元信息，也能够获取 Java 类中函数的全部原始信息，之所以能够获取，就是因为 JVM 在内部为每一个 Java 类、每一个 Java 方法都建立了内存模型，保存 Java 类和 Java 方法的全部信息，因此在运行期通过反射只需访问这个内存模型就能得到这些信息。这是 Java 语言与 C++（两种都是面向对象的编程语言）的最大不同。C++程序由于编译后直接变成了二进制机器指令，已经擦除了所有的面向对象信息，因此 C++程序在运行期是获取不到 C++类和函数的原始信息的（当然著名的 RTTI 能够为 C++提供类型反射的能力）。

3）entry_point

entry_point 是继_call_stub_entry 这一 JVM 最核心例程之后的又一个最主要的例程入口。前面对_call_stub_entry 进行了简单的分析，JVM 每次从 JVM 内部调用 Java 函数时（相对于通过字节码指令调用目标 Java 函数），必定调用 CallStub 函数指针，而该函数指针的值就是_call_stub_entry。JVM 通过_call_stub_entry 所指向的函数地址，最终调用到 Java 函数。

在 JVM 通过_call_stub_entry 所指向的函数调用 Java 函数之前，必须要先经过 entry_point 例程。事实上，在 entry_point 例程里面会真正从 method()（刚刚讲过，这是 Java 函数在 JVM 内部的模型）对象上拿到 Java 函数编译后的字节码，JVM 通过 entry_point 可以得到 Java 函数所对应的第一个字节码指令，然后开始调用 Java 函数。

这里先简单绘制一下 JVM 经过_call_stub_entry，到 entry_point，再到 Java 程序 main()主函数的路线图，如图 2-12 所示。

对于该图大家一定充满了疑惑，一方面是函数指针的调用方式实在比较特殊，如果没有这方面的开发经验，简直有点颠覆大家对函数调用的一般性认识；另一方面，_call_stub_entry 和 entry_point 这两个例程的概念也实在是云里雾里。现在我只能说，这两个例程其实就是两段机器指令，JVM 提前写好了很多例程，例如函数进入、函数调用、函数返回、异常处理、静态函数调用、本地函数调用，等等，这些例程都是直接使用机器指令写成。直接使用机器指令编写而成的这些例程，能够最大程度地提高程序运行的效率。当然，JVM 中所定义的 200 多种字节码指令，每一条字节码指令也有一个例程，也全部直接使用机器指令写成。

see more please visit: https://homeofpdf.com

图 2.12　JVM 调用 Java 程序 main()主函数的路线图

不过任何复杂的事，其实都是可以用简单的道理说明白的，如果说 JVM 中的所谓例程的概念比较玄乎，那么对于广大 Java 程序员来说，有一个名词一定是耳熟能详的，那就是——工具类。我们就将例程想象成一个个工具类即可，个人认为这两者之间完全具有可类比性。从所要实现的功能来看，无论是例程，还是工具类，最终的目的无非就是实现某一个特定的逻辑而已。两者所不同的只是，例程是直接用机器指令写的，而 Java 工具类则是使用 Java 语言写的。（严格来说，例程也不是直接用机器指令写的，而是用 C 语言在运行期动态生成的。不过本书意在讲述 JVM 主要的实现思路，以及为什么会选择各种奇妙和深奥的技术，所以如果过分注重细节，一方面比较绕，另一方面也容易“歪楼”，偏离主题，误导思路。所以本书对太细节的东西不会讲得那么清楚，望大家理解。当然，不注重细节，不代表不严谨。）

_call_stub_entry 和 entry_point 这两个例程的讲解到此先告一段落，后面会深入讲解其机器指令，对其逻辑一探究竟。我们还是继续回到主题，分析 CallStub()的入参。

4）parameters()

这个参数看名字就能猜出其含义，事实上也的确如你所猜，这个参数就是描述 Java 函数的入参信息的。

在 JVM 真正调用 Java 函数的第一个字节码指令之前，JVM 会在 CallStub()函数中解析 Java 函数的入参，解析后，JVM 会为 Java 函数分配堆栈，并将 Java 函数的入参逐个入栈。这样，JVM 就为高层次的编程语言建立了方法栈模型。

相比于 C/C++/Delphi 等编译型语言，Java 这门解释型语言最大的不同就是，不直接在物理机器上运行，而是运行在虚拟机上，所以不能直接使用物理机器的方法栈，必须在虚拟机层面

see more please visit: https://homeofpdf.com

为每一个 Java 函数都建立堆栈模型，这种堆栈模型中的局部变量都是 Java 语言的变量类型。所以，JVM 在正式调用 Java 函数之前，必须要能够获取到 Java 语言层面上的参数类型、参数对象、参数顺序等信息，唯其如此，JVM 才能正确调用并执行 Java 函数。

众所周知，Java 语言是一门面向对象的语言，在内存模型上，Java 类的实例对象分配在堆中，而堆栈中只保存了引用。所谓引用，在 JVM 内部，实际上就是一个指针。JVM 为何要这样来分配内存呢？其实并不是詹爷吃饱了撑的，而是因为必须这么处理。导致 JVM 被逼做出这种内存分配模型策略的原因是，堆栈空间通常都比较小，放不下那么大的对象。在物理机器上直接跑的程序，堆栈空间一般最大为 64MB（当然可以调大），而 Java 堆栈一般为 1MB，或者 8MB，或者再设置得大一点，但是不可能达到诸如 500MB 甚至 1GB 这样的规模。如果每一个函数都设置这么大的堆栈空间，那么函数的调用深度稍微一大，整个物理机器的内存就会溢出。正因如此，JVM 只能将类对象实例保存到堆中。否则，如果我们定义一个很大的字符串传给某个函数，如果整个字符串信息都保存在堆栈中，那么 JVM 很容易就会被各种大字符串攻击。

关于 Java 堆栈模型以及对象引用模型就先介绍到这里，后文会进行详细深入的分析。

5）size_of_parameters()

这个入参也是见其名就知其意，其含义就是参数数量，即 Java 函数的入参个数。

刚才讲到另一个入参，parameters()，这个参数保存了 Java 函数的所有入参信息，但是 parameters()里面其实是使用指针建立起来的数组模型，JVM 在后面调用 Java 函数时，直接通过 parameters()内部的指针，无法得知结束位置，因此这里需要将 Java 函数的入参数量传递进去，JVM 在为 Java 函数分配堆栈空间时，会根据这个值，计算出 Java 堆栈空间大小。

总体而言，CallStub 例程是 JVM 内部举足轻重的一个功能，而 JVM 在调用 CallStub 函数指针时，可谓是层层包装，让人一眼看不出究竟。本节通过抽丝剥茧和场景还原，剖析了 CallStub 这种函数指针的调用机制，描述了 CallStub 函数指针的定义、调用和入参。

2.5 _call_stub_entry 例程

上面花费不少精力讲完 CallStub()入参，接下来就可以分析_call_stub_entry 这个例程所指向的一段机器指令的逻辑了。由于这段机器指令牵扯的东西太多（例如汇编基础、Java 函数入参、Java 堆栈模型等），因此前面才会花那么多篇幅进行相关技术的介绍。

前面在讲 JVM 调用 CallStub()时，提到 JVM 先调用 call_stub()函数，返回_call_stub_entry，然后将_call_stub_entry 强制转换为 CallStub 这种自定义的函数指针类型，最终 JVM 调用这一函

see more please visit: https://homeofpdf.com

数指针，实现由 C 程序的世界转入机器指令的世界，完成分水岭的跨越。不过这里有一个最核心的问题，就是，既然_call_stub_entry 最终被转化成了函数指针类型，那么其必定指向了某个函数入口，这样 JVM 才能将这个函数指针当成函数一样进行调用。那么_call_stub_entry 最终究竟指向哪里了呢？这就有必要分析_call_stub_entry 例程了。

前面在分析 castable_address()函数时提到，JVM 中存在这样一条链路，对_call_stub_entry 所代表的例程进行了初始化，这条链路是（下面的链路省略了中间多个步骤，详情请参考前文）：

```
java.c: main()
    java_md.c: LoadJavaVM()
     // …
            stubGenerator_x86_x32.cpp: generate_initial()
```

在这条链路的最末端，stubGenerator_x86_32.cpp: generate_initial()函数对_call_stub_entry 变量进行了初始化，如下所示：

```
StubRoutines::_call_stub_entry =
        generate_call_stub(StubRoutines::_call_stub_return_address);
```

generate_call_stub()函数返回值赋给了_call_stub_entry 变量，generate_call_stub()函数顾名思义，就是产生_call_stub_entry 变量所指向的一个函数首地址。generate_call_stub()函数的定义如下：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：*generate_call_stub()函数*

```
address generate_call_stub(address& return_address) {
        StubCodeMark mark(this, "StubRoutines", "call_stub");
        address start = __ pc();//这里先得到当前入口的内存地址，因为本函数后续流程会继续往代码空间中写入 call_stub()的指令，而外面却只需要拿到这部分指令的首地址，而这里的start 就是首地址，因此这里先拿到，啥也不干，最后直接将其返回出去即可

        // stub code parameters / addresses
        assert(frame::entry_frame_call_wrapper_offset == 2, "adjust this code");
        bool  sse_save = false;
        const Address rsp_after_call(rbp, -4 * wordSize); // same as in generate_catch_exception()!
        const int    locals_count_in_bytes  (4*wordSize);
        const Address mxcsr_save    (rbp, -4 * wordSize);
        const Address saved_rbx     (rbp, -3 * wordSize);
        const Address saved_rsi     (rbp, -2 * wordSize);
        const Address saved_rdi     (rbp, -1 * wordSize);
        const Address result        (rbp,  3 * wordSize);
        const Address result_type   (rbp,  4 * wordSize);
        const Address method        (rbp,  5 * wordSize);
        const Address entry_point   (rbp,  6 * wordSize);
        const Address parameters    (rbp,  7 * wordSize);
```

see more please visit: https://homeofpdf.com

```
    const Address parameter_size(rbp,  8 * wordSize);
    const Address thread          (rbp,   9 * wordSize); // same as in
generate_catch_exception()!
    sse_save = UseSSE > 0;

    // stub code
    __ enter();//以 x86 为例，在 assembler_x86.cpp 中，enter()函数实现是：
push(rbp); mov(rbp, rsp);而这两句代码都调用 emit()，向代码空间中写入机器码
    __ movptr(rcx, parameter_size); //注意：这些看似汇编指令的 C 函数并不是在运
行时被真正调用的函数，它们的作用只不过是往代码空间中写入具有相同作用的机器码
    __ shlptr(rcx, Interpreter::logStackElementSize); // convert parameter
count to bytes
    __ addptr(rcx, locals_count_in_bytes);          // reserve space for
register saves
    __ subptr(rsp, rcx);
    __ andptr(rsp, -(StackAlignmentInBytes));    // Align stack

    // save rdi, rsi, & rbx, according to C calling conventions
    __ movptr(saved_rdi, rdi);
    __ movptr(saved_rsi, rsi);
    __ movptr(saved_rbx, rbx);
    // save and initialize %mxcsr

    //...

    __ BIND(is_double);
    // interpreter uses xmm0 for return values
    if (UseSSE >= 2) {
      __ movdbl(Address(rdi, 0), xmm0);
    } else {
      __ fstp_d(Address(rdi, 0));
    }
    __ jmp(exit);

    return start;
}
```

这段代码最主要的作用就是生成机器码，使用 C 语言动态生成。弄懂这段机器指令的逻辑，对理解 JVM 的字节码执行引擎至关重要。下面我们对这段代码进行详细分析。

1. pc()函数

首先看第一行代码：

```
address start = __ pc();
```

这行代码保存当前例程所对应的一段机器码的起始位置。pc()函数定义如下：

see more please visit: https://homeofpdf.com

清单：/src/share/vm/asm/assembler.hpp
作用：pc()函数定义

```
address  pc() const {
return _code_pos;
}
```

该函数特别简单，返回_code_pos 变量。该变量也定义在/src/share/vm/asm/assember.hpp 文件中，是一个 address 类型的变量。

在 JVM 启动过程中，JVM 会生成很多例程（即一段固定的机器指令，能够实现一种特定的功能逻辑），例如函数调用、字节码例程、异常处理、函数返回等。

每一个例程，一开始都有这么一行代码（即 address start = __ pc()），代码完全相同。事实上，JVM 的所有例程都在一段连续的内存中，我们可以将这段内存想象成一根直线，当 JVM 刚启动时，这根线长度为 0，没有生成任何例程。第一个例程生成时，__ pc()返回 0，因为此时是从这根直线的零点位置开始。假设第一个例程占 20 字节，则当 JVM 生成第二个例程时，第二个例程执行 start=__ pc()时，将返回 20（如果将第一个位置标记为 0，则第二个位置为 20；否则如果从 1 开始标记，则第二个位置为 21），因为第一个例程占用 20 字节。下面使用图示来演示这一过程。

当 JVM 生成第一个例程时，pc()返回 0，如图 2.13 所示。

图 2.13　JVM 第一个例程的起始位置返回 0

（注：JVM 的例程都写入 JVM 的堆内存中，在 JVM 初始化时，会初始化一个容量足够大的堆内存，例程会写入堆中靠近起始的位置。当 Java 程序开始运行后，JVM 将 Java 类对象实例陆续写入堆内存中。在讲解 JVM 初始化的章节中会有关于堆内存的详细分析。）

JVM 中每一个例程都一个对应的 generate()函数（具体的函数名不同，但是基本都以 generate_开头），假设第一个例程占 20 个字节码，则当第一个例程所对应的 generate()函数执行完成后，_code_pos 会自动累加到 20，于是当 JVM 生成第二个例程时，pc()就会返回 20，如

see more please visit: https://homeofpdf.com

图 2.14 所示。

图 2.14　JVM 例程的起始位置

通过图 2.13 和图 2.14 可知，JVM 每生成一个例程，就会将例程起始位置增加，每一个例程都会占用 JVM 堆内存的一块连续区域，相邻例程之间的内存区域相连（即内存位置是靠在一起的），所有的例程最后连成一块连续的区域。而事实上，JVM 内部的确是这样来划分内存的，后面会详细讲。

注意：在每一个例程所对应的 generate() 函数内部，例如这里的 generate_all_stub()函数，都在函数的开始处执行 address start = __ pc()，然后在函数的最后执行 return start。这是因为_code_pos 其实是偏移量，当 JVM 要生成一个新的例程时，偏移量_code_pos 指向上一个例程的最后字节的位置，而上一个例程的最后一个字节的位置，正是下一个例程的起始位置，所以在每一个例程所对应的 generate()函数的开始，先保存这个起始位置。当 generate()函数的主体逻辑执行完成，_code_pos 会不断地自增，直到到达当前例程的最后一个字节的位置。如果不在 generate()函数的开始处就保存_code_pos，那么最后返回的这个值将变成当前例程的末端位置，而不是起始位置。这样，最终 JVM 通过 CallStub 这个函数指针（或者其他例程所对应的函数指针）来执行这段动态生成的机器指令，就会因为位置不对而报错。

---

2. 定义入参

generate_all_stub()接下来执行下面一段代码：

```
assert(frame::entry_frame_call_wrapper_offset == 2, "adjust this code");
bool sse_save = false;
const Address rsp_after_call(rbp, -4 * wordSize); // same as in
generate_catch_exception()!
```


see more please visit: https://homeofpdf.com

```
const int     locals_count_in_bytes  (4*wordSize);
const Address mxcsr_save     (rbp, -4 * wordSize);
const Address saved_rbx      (rbp, -3 * wordSize);
const Address saved_rsi      (rbp, -2 * wordSize);
const Address saved_rdi      (rbp, -1 * wordSize);
//…
```

在讲解入参之前，我们先回顾下在前面分析机器调用时的调用者和被调用者堆栈模型。当时是以 main()和 add()举例说明的（可以回头再看看汇编代码和内存模型图），main()是调用者函数 caller，add()是被调用函数 callee，当物理机器由 caller 执行到 callee 时，堆栈模型的变化如图 2.15 所示。

图 2.15　main()与 add()函数堆栈整体布局

由图 2.15 可知，一个函数的堆栈空间大体上可以分为 3 部分。

1）堆栈变量区

保存方法的局部变量，或者对数据的地址引用（指针）。

如果一个方法中并没有局部变量，则编译器不会为该方法分配堆栈变量区。

2）入参区域

如果当前方法调用了其他方法，并且给其他方法传递了参数，那么这些入参会保存在调用者的堆栈中，这就是所谓的“压栈”。

see more please visit: https://homeofpdf.com

至少在 x86 平台上，入参区域相对于方法的堆栈变量区，在内存上位于低位置，即堆栈变量区在高地址方向，而入参区域则在低地址方向。x86 在分配堆栈空间时，本来就是按照由高地址向低地址的方向分配的。

3）ip 和 bp 区

ip 和 bp，一个是代码段寄存器，一个是堆栈栈基寄存器。这两个寄存器，一个用于恢复调用者方法的代码位置，一个用于恢复调用方法的堆栈位置，是完成物理机器函数调用机制的最主要的 2 个寄存器。关于这两个寄存器如何被保存，如何被恢复，在前面的章节中已经详细描述过。

函数堆栈空间的一般布局如图 2.16 所示。

图 2.16　函数堆栈空间布局通用模型

其实，物理机器在执行函数调用时，存在一定的空间浪费。入参往往同时是当前方法的局部变量，编译器会将局部变量分配在局部变量区域，而在入参时，会将局部变量再次复制一份放到压栈区域，同一份数据被分配了两次堆栈空间，其实依靠编译器的智能性，完全可以将堆栈中的入参区域去掉。但是话又说回来，正因为编译器规定了入参空间分配的原则，并使入参按照从左到右或从右到左的顺序压栈，这种规范不仅仅让被调用函数在访问入参时享受到了极大的便利，也让 JVM 设计者（詹爷）在设计 Java 函数调用机制时，能够基于这一规范，随心所欲地发挥。很难想象，如果缺少了这一规范，jvm，包括很多其他虚拟机，在设计函数调用机制时，会是怎样的一种场景，大家会想出多少招式来实现这一复杂的逻辑？

基于这一规范，在被调用者方法中，访问入参，就变得有规律可循。下面先看一个简单的例子，假设有如下一个 C 程序：

see more please visit: https://homeofpdf.com

清单：示例程序

作用：C 程序示例

```
int add(int, int);
int main(){
        int a = 6;
        int b = 8;

        // …定义若干局部变量

    int c=add(a, b);
    return 0;
}

int add(int x, int y){
    int z=x+y;
    return z;
}
```

当该程序运行时，物理机器首先为 main()函数分配局部变量，此时 main()函数堆栈内存布局如图 2.17 所示。

图 2.17　main()函数堆栈布局

（注：细心的读者会发现这个图有问题，因为当 CPU 运行 main()函数指令时，根本不会为 add()函数分配堆栈空间，因此这里将 add()函数的堆栈空间画上去是错误的。作者将其添加上去，主要是为了让大家能够总览调用者与被调用者之间的堆栈布局，从而建立这种印象。）

main()函数开始调用 add()函数之前，main()函数会将入参压栈，注意：参数被压到 main()函数的堆栈中。此时整体堆栈空间布局如图 2.18 所示。

see more please visit: https://homeofpdf.com

图 2.18　main()函数压参后的堆栈整体布局

注意：在 x86 平台上，入参以逆向顺序压栈，因此 8 先压栈，6 后压栈。关于参数压栈的机器指令，大家可以回头看前面讲解物理机器函数调用机制的章节，里面有详细的描述。

完成入参压栈后，接着物理机器将 ip 和 bp 这两个寄存器入栈，此时堆栈内存布局如图 2.19 所示（这两个寄存器的入栈原理，前文也详细描述过，若有不清楚，可以翻看前面章节）。

图 2.19　ip 和 bp 寄存器压栈

到了这一步，完成了 ip 和 bp 寄存器的入栈，物理机器就开始为被调用函数 add()分配堆栈空间了，add()函数的栈底与 ebp 寄存器相邻。在 x86 平台上，堆栈由高内存地址往低内存地址

see more please visit: https://homeofpdf.com

方向分配，因此 add()函数的栈底相对于 ebp 寄存器而言，处于低内存位置。

前面讲过，物理机器对堆栈内存寻址的方式为相对偏移，可以通过相对栈底 bp 或栈顶 sp 的位置来绝对定位堆栈内存位置。当相对 sp（栈顶）进行寻址时，如果对位于 sp 下方（低内存地址方向）的堆栈内存进行寻址，则使用-*m*(%sp)这样的方式，其含义是 sp 寄存器的值减去 *m* 字节，*m* 是个自然数，例如-3(%sp)。反之，如果对位于 sp 上方（高内存地址方向）的堆栈内存进行寻址，则使用 *m*(%sp)这样的方式，其含义是 sp 寄存器的值加上 *m* 字节，例如 3(%sp)。

相对于 bp（栈底）进行相对寻址，也是同样的道理。

接着上面的例子，当物理机器执行到 add()函数时，add()函数要执行 int z = x + y 这样的代码，就必须读取入参。此时物理机器的 bp 指向 add()函数的栈底，因此可以通过 add()函数堆栈栈底进行相对寻址，读取两个入参 x 和 y。那么 x 和 y 相对 add()函数栈底的偏移量是多少呢？

我们来进行看图说话，看图 2.19，ebp 相对于 add()栈底的偏移位置是 0，因此可以标记为(%ebp)。eip 相对于 add()栈底的偏移位置是 4 字节（32 位平台，ebp 寄存器占了 4 字节的空间），同时，eip 相对于 add()函数栈底，位于高内存地址方向，因此标记为 4(%ebp)。

同样，第一个入参 x 相对于 add()栈底的偏移位置是 8 字节（32 位平台，ebp 和 eip 这 2 个寄存器各占去 4 字节，因此一共占用了 8 字节内存空间），因此 x 的位置可以标记为 8(%ebp)。第二个入参 y 相对于 add()栈底的偏移位置是 12 字节（前面 ebp、eip、x 这 3 个数据各占 4 字节，因此一共占去 12 字节），所以 y 的位置可以标记为-12(%ebp)。用图 2.20 显示这种标记。

图 2.20　以 bp 为基址对变量进行标记

see more please visit: https://homeofpdf.com

现在，假设 main()函数在调用 add()时，不仅仅包含 2 个入参，而是包含 3 个，那么我们根据图 2.20，可以推导出这 3 个入参的内存定位：

◎ 第 1 个入参为 8(%ebp)。

◎ 第 2 个入参为 12(%ebp)。

◎ 第 3 个入参为 16(%ebp)。

根据这种推导关系，使用数学归纳法，可以得到这样的入参寻址公式：

```
Pn = (n + 1) * 4(%ebp)
```

在该公式中，*n* 表示第 *n* 个入参（按从左至右的顺序），P*n* 表示第 *n* 个入参的位置。

有了这个公式，我们在理解 JVM 调用 CallStub 函数指针时，就会理解 JVM 的入参定义。那么现在就让我们将目光再次定格到 generate_call_stub()这个函数中。由于 CallStub 函数指针最终指向 generate_call_stub()这个函数所返回的一段机器指令，因此 generate_call_stub()中生成的机器指令其实就是被调用函数对应的机器码，可以将其理解成 CallStub()的函数体。

JVM 在 javaCalls.cpp::call_helper()函数中执行 CallStub 调用，在调用时，传递了如下 8 个参数：

◎ (address)&link，连接器

◎ result_val_address，返回地址

◎ result_type，返回类型

◎ method()，Java 方法的内部对象

◎ entry_point，Java 方法调用入口例程

◎ args->parameters()，Java 方法的入参

◎ args->size_of_parameters()，Java 方法的入参数量

◎ CHECK：当前线程

按照公式 P*n* = (*n* + 1) * 4(%ebp)，计算这 8 个参数相对于被调用函数 CallStub()栈底的位置，计算后的位置如表 2.2 所示。

表 2.2　入参位置

入　参	位　置
(address)&link：连接器	8(%ebp)
result_val_address：返回地址	12(%ebp)
result_type：返回类型	16(%ebp)
method()：Java 方法的内部对象	20(%ebp)
entry_point：Java 方法调用入口例程	24(%ebp)

see more please visit: https://homeofpdf.com

续表

入　参	位　置
args->parameters()：Java 方法的入参	28(%ebp)
args->size_of_parameters()：Java 方法的入参数量	32(%ebp)
CHECK：当前线程	36(%ebp)

当 JVM 进入到 CallStub 这个函数指针所代表的函数的堆栈中后，调用者 javaCallls::call_helper 与被调用者 CallStub()之间的堆栈内存布局如图 2.21 所示。

图 2.21　javaCalls::call_helper()与 CallStub()的堆栈布局

在图 2.21 中，这 8 个入参相对于 ebp 的偏移量都是以字节为单位进行计算的，在 32 位平台上一个指针占 32 位，4 字节，如果我们将偏移量以 4 字节为一个单位进行标记，而不是以 1 字节为计量单位，那么这 8 个入参的偏移位置的值会缩小 4 倍，看起来就不会那么大，例如 28(%ebp)就会被标记为 7*N*(%ebp)，这里 *N* 为常量 4。以 4 字节为单位对这 8 个入参的偏移位置进行重新标记，如表 2.3 所示。

see more please visit: https://homeofpdf.com

表 2.3　对 CallStub()的 8 个入参的 ebp 偏移位置按 4 字节为单位进行标记

入　参	位　置
(address)&link：连接器	2*N*(%ebp)
result_val_address：返回地址	3*N*(%ebp)
result_type：返回类型	4*N*(%ebp)
method()：Java 方法的内部对象	5*N*(%ebp)
entry_point：Java 方法调用入口例程	6*N*(%ebp)
args->parameters()：Java 方法的入参	7*N*(%ebp)
args->size_of_parameters()：Java 方法的入参数量	8*N*(%ebp)
CHECK：当前线程	9*N*(%ebp)

诸如 3(%ebp)这种标记方法，属于汇编语言，而 JVM 毕竟是使用 C/C++写成的，于是 JVM 大神对汇编语法进行了抽象，使用 C++类来表示一个堆栈位置。汇编语言通过寄存器和偏移量唯一定位一个堆栈内存，那么我们就可以在 C++类中将寄存器和偏移量分别抽象成两个变量，这个 C++类可以定义成如下这种格式：

清单：test

作用：演示使用 C++类抽象汇编

```
class Address {
private:
    // 表示一个物理寄存器，寄存器的作用是标记基址，因此命名为_base
    Register        _base;

    // 表示偏移量，是个整型数
    int             _disp;

public:

    // 构造函数
    Address()
        : _base(noreg),
        _disp(0)
    {   }
}
```

该程序里有一个 Register 类，先不用关心其具体的数据结构，只需知道它能够表示一个物理寄存器就够了。使用 C++类对汇编堆栈寻址进行抽象后，便可以直接用该类进行堆栈寻址了。假设要对 8(%ebp)进行寻址，可以这样写：

```
Address position (rbp, 8);
```

这就声明了一个 C++类对象 position，最终可以通过该对象，还原出汇编指令 8(%ebp)。

see more please visit: https://homeofpdf.com

基于该C++类，我们可以这样对CallStub()的8个入参的偏移位置进行标记（如表2.4所示）。

表 2.4　对 CallStub()的 8 个入参的 ebp 偏移位置按 4 字节为单位进行标记

入　参	位　置	C++类标记
(address)&link	2*N*(%ebp)	Address link(rbp, 2*N*)
result_val_address	3*N*(%ebp)	Address returnAddress(rbp, 3*N*)
result_type	4*N*(%ebp)	Address resultType(rbp, 4*N*)
method()	5*N*(%ebp)	Address method(rbp, 5*N*)
entry_point	6*N*(%ebp)	Address entryPoint(rbp, 6*N*)
args->parameters()	7*N*(%ebp)	Address parameters(rbp, 7*N*)
args->size_of_parameters()	8*N*(%ebp)	Address parametersSize(rbp, 8*N*)
CHECK	9*N*(%ebp)	Address thread(rbp, 9*N*)

（注：表中的 *N*=4）

现在，让我们再回到 stubGenerator_x86_32.cpp 文件中的 generate_call_stub()函数开始的那10几行的类声明（）：

```
//…
const Address result          (rbp,  3 * wordSize);
const Address result_type     (rbp,  4 * wordSize);
const Address method          (rbp,  5 * wordSize);
const Address entry_point     (rbp,  6 * wordSize);
const Address parameters      (rbp,  7 * wordSize);
const Address parameter_size (rbp,  8 * wordSize);
const Address thread          (rbp,  9 * wordSize);
//…
```

对于这个定义，相信聪明的你，不难理解。例如，这里的 parameters_size 代表的堆栈位置是 8 * wordSize(%rbp)，其标记方法是：

```
const Address parameter_size(rbp,  8 * wordSize);
```

这正好与我们上面 C++类标记法是一致的，我们在上文标记为：

```
Address position (rbp, 8N);
```

事实上，JVM 里面所定义的 Address 类与我们在上文自定义的 Address 类并无本质区别，无非是额外多了几个变量而已。同时，JVM 里定义了常量 wordSize，其声明如下：

清单：/src/share/vm/utilities/globalDefinitions.hpp
作用：wordSize 常量定义

```
const int wordSize          = sizeof(char*);
```


see more please visit: https://homeofpdf.com

在 32 位平台上，sizeof(char*)将返回 4；在 64 位平台上，sizeof(char*)返回 8。由于 char* 是一个指针类型，而指针能够指向物理机器内存的任何一个地址，因此，在 *N* 位平台上，指针的宽度必须也至少是 *N* 位，这样指针才能寻址到内存任何一个位置。假如内存总大小是 64 比特，那么指针宽度只需 6 位，即可寻址到内存任何位置，2 的 6 次方正好等于 64。JVM 作为一款能够兼容大部分主流操作系统的虚拟机，兼容指针长度是其基本功。

上面这段话，对新手起到一个启发思路的作用。我们还是回到 generate_call_stub()函数一开始的变量定义，要注意，你看不到一个类似于

const Address linke (rbp, 2 * wordSize)这样的定义，这是因为这个函数中并没有用到这个入参。

同时，你还会看到如下定义：

```
//...
const Address mxcsr_save    (rbp, -4 * wordSize);
const Address saved_rbx     (rbp, -3 * wordSize);
const Address saved_rsi     (rbp, -2 * wordSize);
const Address saved_rdi     (rbp, -1 * wordSize);

const Address result        (rbp,  3 * wordSize);
//...
```

mxcsr_save、saved_rbx、saved_rsi、saved_rdi 这 4 个位置相对于 rbp 的偏移量是负数，这很容易理解，说明这 4 个参数的位置在 CallStub()函数的堆栈内部，而不是位于调用函数 javaCalls::call_helper()堆栈内，所以相对于 rbp 的偏移量才会是负数。这 4 个变量用于保存调用者的信息，在后面会详细分析这 4 个变量。

讲完了 generate_call_stub()函数开始的那 10 几行的类声明，接下来开始真正进入 CallStub 例程的逻辑分析。

3. CallStub：保存调用者堆栈

generate_call_stub()函数的逻辑部分从下面这行代码开始：

```
__ enter();
```

这行代码在不同的硬件平台上，对应不同的机器指令。在 x86 平台上，其函数定义在 assembly_x86.cpp 文件中，定义如下：

清单：/src/cpu/x86/vm/assembly_x86.cpp:enter()

作用：enter()函数定义

```
void MacroAssembler::enter() {
    push(rbp);
    mov(rbp, rsp);
}
```


see more please visit: https://homeofpdf.com

这两条指令最终会在 JVM 运行期被翻译为如下所示的对应的机器指令：

```
push %bp
mov  %sp, %bp
```

如果你认真地看过前面章节，或者熟悉汇编指令，那么你对这两条指令一定不陌生。在 x86 平台上，物理机器调用任何一个函数之前，都会执行这两条指令，push %ebp 指令的含义是保存调用者函数的栈基地址，mov %sp, %bp 指令的含义是重新指定栈基地址。由于即将开始新的函数，因此需要将栈基指向调用者函数的栈顶位置，调用者函数的栈顶位置就是被调用者函数的栈基位置（严格来说这句话是错误的，因为调用者与被调用者函数之间还隔着两个寄存器：ip 和 bp）。

执行 enter()之前，bp 和 sp 这 2 个寄存器指向位置如图 2.22 左半部分所示，此时堆栈空间属于调用者函数 javaCalls::call_helper()。执行 enter()之后，这 2 个寄存器将指向新的函数，如图 2.22 右半部分所示，此时堆栈空间属于被调用者函数 CallStub()：

图 2.22　javaCalls::call_helper()执行 enter()前后 sp 和 bp 寄存器指向

see more please visit: https://homeofpdf.com

4．CallStub：动态分配堆栈

应该说，JVM 之所以能够在物理机器上分配 Java 语言变量类型的堆栈，完全得益于机器级别对堆栈空间分配的指令支持。JVM 充分利用了这一点，通过重新分配堆栈空间，从而为 JVM 调用 Java 函数奠定基石。

大家正常写一段 C 程序，编译时，编译器会根据被调用函数中的变量声明，自动计算出被调用函数需要多大的堆栈空间。最终物理机器在执行这个 C 程序时，将按照编译器所计算出的大小为被调用函数分配堆栈空间。

而 Java 程序由于无法直接被编译为机器指令，因此 Java 编译器无法直接计算一个 Java 函数需要多大的堆栈空间（其实如果真的要做的话，也是可以做到的），但是如果仅仅解决 Java 函数的堆栈空间的自动计算，还是无法实现 Java 函数的调用，因为 Java 有自己的变量类型，这些变量类型不像 C 语言的变量，Java 的变量类型并不能直接被编译成物理机器所识别的数据类型，而 C 语言变量最终完全被编译成物理机器所识别的类型。所以，即使 Java 程序编译器能够自动计算出 Java 函数所需要的堆栈空间大小，物理机器也无法对这段堆栈内存进行读写，因为物理机器完全不识别 Java 的变量类型。当然，从技术层面上，对于强类型的 Java 语言，是完全可以做到直接将 Java 语言编译为机器指令的，JVM 的 JIT 编译器就是这种实现方式，不过由于将 Java 程序直接编译为机器指令，既要实现高级语言到原始语言的转换，还要能够确保 Java 程序的逻辑在编译后保持完全一致，这才是最大的挑战。所以 JIT 提供了多种选项，有些选项不进行任何优化就编译，有些选项则可以对 Java 程序进行激进式的编译。不经过优化的编译，与解释型运行机制基本无异，还不如不编译，因为这种编译机制所编译出来的机器指令，与直接进行解释执行时所动态生成的机器指令的数量、质量基本相差不大。但是如果进行激进式的编译，可能会破坏原本 Java 的程序逻辑。

Java 发明之初，原本是想用于智能家居领域，没想到“无心插柳柳成荫”，Java 在中间件和分布式领域大放异彩，这是由于中间件和分布式领域主要关注网络通信和数据处理，开发者不用再分散精力去关心底层实现，同时中间件需要能够兼容各种底层硬件平台，这些正好都是 Java 天生所具备的特性。也正是由于这一点，在分布式领域，JVM 无需过多关注编译质量问题，毕竟在分布式领域，大家关注的点不再是性能，而是大数据的处理能力、架构的稳定性与伸缩性等方面（这并不是说写 Java 程序就不用关注内存、不用关注程序性能了，事实上很多中间件反而十分追求卓越的性能，这里主要说明，Java 程序员并不像底层开发者那样对性能和内存十分关注，底层程序员可能对一个字节的内存消耗都很在意，而 Java 程序员很少会在意一两个字节的占用，毕竟 JVM 的堆内存都是以 GB 为单位计算的，并且 Java 里面随便声明一个简单类型的变量，都会占用超出一个字节的内存）。但是在移动操作系统领域，个人更倾向于一种能够直接将 Java 程序（或类似 Java 的编程语言所写出来的程序）编译成本地物理机器指令的机制，或

see more please visit: https://homeofpdf.com

者一种能够直接运行 JVM 字节码指令的 CPU（有公司在做这方面的研究），从而完全消除虚拟机依赖，这样既能确保商业项目对效率的极致追求，又能保证 Java 程序在本地运行时对性能的苛刻需求。

上面仅是一家之言，相信大家定有不同观点。上面扯得有点远了，回归主题。JVM 为了能够调用 Java 函数，需要在运行期知道一个 Java 函数的入参大小，然后动态计算出所需要的堆栈空间。这就是 JVM 能够调用 Java 函数的核心机制。这里的关键问题是，CallStub()作为被 javaCalls::call_helper()调用的函数，JVM 通过 javaCalls::call_helper()最终调用到 Java 函数，JVM 作为一款使用 C/C++编写而成的程序，被编译后，C/C++编译器自然会计算出 javaCalls::call_helper()传递给 CallStub()的入参的空间大小，但是 C/C++编译器并不会因此就自动计算出 Java 函数的入参数量及所需内存空间大小，因为 C/C++编译器并不识别 Java 程序，并且在 JVM 被编译期间， JVM 尚未加载任何 Java 程序，因此 JVM 对 Java 程序完全无感。等到 JVM 程序运行起来后，JVM 会加载 Java 程序，并通过 javaCalls:call_helper()调用 Java 主函数。JVM 在执行 Java 函数调用时，仍然沿用了物理机器所使用的“堆栈”这一算法数据结构，并没有发明新的轮子，因此 JVM 仍然需要为 Java 被调用函数分配堆栈内存，保存被调用函数的局部变量以及相关上下文数据。因此，JVM 必然需要在运行期动态计算 Java 被调用函数的空间大小，并动态为其分配堆栈空间。而物理机器提供了这种动态分配堆栈空间的能力，这一点在前文讲述物理机器函数调用机制时讲过。

由于物理机器不能识别 Java 程序，也不能直接执行 Java 程序，因此 JVM 必然要通过自己作为一座桥梁连接到 Java 程序，并让 Java 被调用的函数的堆栈能够“寄生”在 JVM 的某个函数的堆栈空间中，否则物理机器不会自动为 Java 方法分配堆栈。前面讲过，JVM 选择 CallStub 这一函数指针作为 JVM 内部的 C/C++程序与 Java 程序的分水岭，或者桥梁，通过这座桥梁，当 JVM 启动后，执行完 JVM 自身的一系列指令后，能够跳转到执行 Java 程序经翻译后所对应的二进制机器指令，CallStub 能够实现机器逻辑指令上的联接，同时，JVM 会调用 Java 的入口主函数 main()，并将 main()主函数的入参传递进去。因此，在分水岭之后，JVM 需要为主函数分配堆栈空间，以在主函数中读取入参数据。那么 Java 函数所需要的堆栈空间分配在哪里呢？答案是明显的，既然 CallStub()作为分水岭的函数，很自然地，JVM 将 Java 函数堆栈空间“寄生”在了 CallStub()函数堆栈中。当然，从技术实现的手段而言，JVM 并非一定要选择“寄生”这种方式，JVM 完全可以另外定义一种算法结构来支持 Java 函数的调用机制，但是 JVM 并没有这么做。

那么接下来的问题就变成了，如何才能实现“寄生”？ 这就需要依靠物理机器提供的指令，对 CallStub()堆栈进行扩展。物理机器提供了扩展堆栈空间的简单指令，如下（下述指令基于 x86 平台）：

see more please visit: https://thomeofpdf.com

```
sub operand, %sp
```

operand 是一个自然数，例如 8、16 或者其他数值。这条指令表示将堆栈向下扩展一定的空间。

如果你写了一个 C/C++程序，C/C++编译器会自动计算一个函数所需要的堆栈大小，例如下面这个例子：

清单：示例

作用：C 函数调用示例

```
#include<stdio.h>

int add(int x, int y);
int main(){
    int a=5;
    int b=3;

    int c=add(a,b);
    printf("%d\n",c);

    return 0;
}

int add(int x, int y){
    int z=x+y;
    return z;
}
```

将这段程序编译为汇编程序，得到：

清单：示例

作用：C 函数调用示例

```
main:
    pushl   %ebp
    movl%esp, %ebp

    andl$-16, %esp
    subl$32, %esp

    movl$5, 20(%esp)
    movl$3, 24(%esp)

    movl24(%esp), %eax
    movl%eax, 4(%esp)
    movl20(%esp), %eax
    movl%eax, (%esp)

    calladd
```


see more please visit: https://homeofpdf.com

```
    movl%eax, 28(%esp)

    movl28(%esp), %edx
    movl%edx, 4(%esp)
    movl%eax, (%esp)
    callprintf
    movl$0, %eax

    leave
    ret

add:
    pushl   %ebp
    movl%esp, %ebp
    subl$16, %esp

    movl12(%ebp), %eax
    movl8(%ebp), %edx
    addl%edx, %eax
    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax

    leave
    ret
```

对于被调用函数 add()，其内部只声明了一个局部变量 z。对于入参 x 和 y，在所对应的汇编程序中，并没有发现编译器为其分配堆栈空间，编译器将使用 ax 和 dx 这两个寄存器分别保存这 2 个入参。因此，add()函数其实只需要为变量 z 分配堆栈空间，大小为 32 字节。由于编译器会自动对齐，因此最终编译器为 add()函数分配了 16 字节的堆栈，分配的方式如下：

```
subl $16, %esp
```

执行这段汇编程序后，最终系统打印出正确的结果值：8。

虽然 C/C++编译器会自动计算堆栈大小，但是可以人工对计算的结果值进行修改。我们继续实验，将上面这段汇编程序中 add 代码段的堆栈空间变为 64，修改后的程序如下：

清单：示例
作用：C 函数调用示例

```
main:
    pushl   %ebp
    movl%esp, %ebp

    andl$-16, %esp
    subl$32, %esp
```


see more please visit: https://homeofpdf.com

```
    movl$5, 20(%esp)
    movl$3, 24(%esp)

    movl24(%esp), %eax
    movl%eax, 4(%esp)
    movl20(%esp), %eax
    movl%eax, (%esp)

    calladd
    movl%eax, 28(%esp)

    movl28(%esp), %edx
    movl%edx, 4(%esp)
    movl%eax, (%esp)
    callprintf
    movl$0, %eax

    leave
    ret

add:
    pushl   %ebp
    movl%esp, %ebp
    subl$64, %esp

    movl12(%ebp), %eax
    movl8(%ebp), %edx
    addl%edx, %eax
    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax

    leave
    ret
```

注意，现在 add 标段中 subl $16, %esp 变成了 subl $64, %esp，这表示为 add 分配 64 字节空间。运行修改后的汇编程序，会发现程序依然输出了正确的结果值：8。

这就是 JVM 实现堆栈“寄生”的机制。扩展别人的堆栈，存储自己所需要的数据。

CallStub() 作为 JVM 内部 C/C++ 与 Java 程序的分水岭，CallStub() 调用者 javaCalls::call_helper()并没有直接将 Java 函数的入参传递给 CallStub()，因为这个调用者并不直接是 Java 函数自己，因此在 JVM 的编译阶段，并没有将 Java 函数压栈。在上面 C 示例程序中，main()函数调用了 add()函数，add()函数的 2 个入参 x 和 y，在 main()函数中完成了压栈，这 2

see more please visit: https://homeofpdf.com

个入参实际被保存在了 main()这个调用函数堆栈中。同理，JVM 要调用 Java 主函数（由于 JVM 第一次调用 Java 函数从 Java 程序的主函数 main()开始，因此这里以 main()为例讲述）main()，众所周知，Java 主函数 main()包含一个字符串数组入参，因此 Java 主函数的声明格式一定如下：

清单：示例

作用：Java 主函数声明示例

```
public static void main(String[] args){
    //...
}
```

Java 主函数一定包含一个 String[]类型的入参。既然 JVM 会调用这个方法，那么按照 C/C++程序的函数调用机制，JVM 中调用 Java 程序主函数 main()的函数（为了表述方便，我们假设 JVM 中调用 Java 主函数 main()的函数名为 xxx()），必然要对 Java 主函数 main()的入参 String[] args 进行压栈，JVM 会将 args 参数保存在 xxx()的堆栈中，这样在 Java 的主函数 main()内部才能访问入参数据。

但是，由于 JVM 在编译期间对 Java 程序完全“无感”，JVM 在编译时，压根儿就不知道加载的是什么样的 Java 程序，也不知道 Java 的主函数的入参数据是什么，因此 C/C++编译器在编译 JVM 时，对于调用 Java 主函数 main()的 xxx()函数，并不会将 Java 主函数 main()所需的入参 String[] args 数据压栈到 xxx()函数中，xxx()函数中根本就没有 args 数据。

那么问题来了，当 JVM 执行完自己的一系列指令后，最终开始调用 Java 的主函数 main()时，Java 主函数 main()所需的 args 入参信息保存在哪里，从哪里获取，这些信息又是什么呢？

一切奥秘都在 CallStub 这个函数指针中所指向的函数中，即上文一直在讲述的 stubGenerator_x86_32 : generate_call_stub()函数。为了讲述方便，前文一直直接使用 CallStub()来指代 generate_call_stub()函数，实际上 JVM 内部并不存在 CallStub()这个函数，CallStub 仅仅是一个函数指针。但是为了讲述方便，下文继续使用 CallStub()这一假想中的函数。

前面说了很多次，CallStub()是 JVM 内部 C/C++程序与 Java 程序之间的分水岭和桥梁，分水岭的其中一个重要作用就是能够将 Java 程序被调用的函数的入参分配到堆栈中，这样在 Java 函数中才能对 Java 类型的入参进行寻址。其实，刚才所假想的在 JVM 内部调用 Java 程序主函数 main()的 xxx()函数，就是 CallStub()函数。

刚才讲到，既然 CallStub()函数调用了 Java 程序的主函数 main()，那么在 CallStub()函数中必然要将 Java 程序主函数 main()所需的入参信息 String[] args 进行压栈，否则 Java 主函数 main()内部无法对入参进行寻址。但是在 JVM 编译期间，C/C++根本就不知道 Java 程序的任何信息，更无从谈起将 Java 主函数入参压栈。这个问题如何解决呢？那就是使用动态分配堆栈的方式，或者“寄生”。CallStub()函数所对应的机器指令是在 JVM 启动过程中动态生成的，而非编译期

see more please visit: https://homeofpdf.com

间生成，在 CallStub()内部需要知道被调用的 Java 函数的入参数量，并依此计算入参所需空间大小，最终将其压栈，这样当 JVM 在执行 Java 函数时，在被调用的 Java 函数中就能对入参进行寻址。

在 CallStub()函数（即 stubGenerator_x86_32：generate_call_stub()）中，通过如下指令计算出被调用 Java 函数的入参数量，并保存调用者数据段现场：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
        // 定义堆栈位置变量
        //...
        bool  sse_save = false;
        const Address rsp_after_call(rbp, -4 * wordSize); // same as in gener
ate_catch_exception()!
        const int     locals_count_in_bytes  (4*wordSize);
        const Address mxcsr_save    (rbp, -4 * wordSize);
        //...

        // stub code
        __ enter();
        __ movptr(rcx, parameter_size);
        __ shlptr(rcx, Interpreter::logStackElementSize);
        __ addptr(rcx, locals_count_in_bytes);
        __ subptr(rsp, rcx);
        __ andptr(rsp, -(StackAlignmentInBytes));

    //...
}
```

这段代码中的加粗部分，就是 JVM 在对被调用的 Java 函数的入参进行计算。parameter_size 是在 generate_call_stub()函数开始处定义的堆栈变量，其定义如下：

```
const Address parameter_size(rbp,  8 * wordSize);
```

这个变量指向 bp 栈基往高地址偏移 8 个字长的位置，一个字长占 4 字节（32 位平台），因此实际相对 bp 的偏移量为 32 字节。此时 bp 指向 CallStub()函数的栈底，parameter_size 指向位置如图 2.23 所示。

see more please visit: https://homeofpdf.com

图 2.23　parameter_size 变量实际所代表的堆栈位置

javaCalls::call_helper()在调用 Java 函数之前，会读取 Java 函数的入参大小，Java 函数的入参大小由 Java 编译器在编译期间计算出来，因此 JVM 在执行 Java 函数之前，可以将其直接读取出来。JVM 得到 Java 函数所需要的入参数量后，便可以直接计算出入参压栈所需要的堆栈空间。

也许有人会疑惑，不同的入参，数据类型不尽相同，其所占内存大小肯定也不同，那么仅仅根据入参数量，如何就能够确定全部入参所需要的内存空间呢？例如，在 C 程序中，int 类型的入参和 char 类型的入参，其所占的内存大小一定不同，如果要计算全部入参空间大小，必须知道每一种数据类型所占的内存大小，然后进行累加求和。更何况 Java 语言的数据类型比 C 语言更加丰富，不同数据类型所占用的空间大小更加不同。

其实，众所周知，Java 是一门面向对象的编程语言，这一点不仅表现在 Java 的语义上，同时 JVM 在内存模型上也体现了这一原则。在 Java 语义上，定义任何类型的变量（除了 Java 的基本数据类型），都需要进行实例化，从而从语法层面上实现面向对象的宗旨。而 JVM 在内存

see more please visit: https://stomeofpdf.com

中，也为每一个 Java 类型和对象创建了一个内存模型，这是 Java 能够在运行期获取 Java 类型的描述信息的根本前提。由于内存模型也遵循了面向对象的原则，因此实际上在 JVM 内部，对 Java 类型实例的访问全部通过指针来实现，访问 Java 类型实例的成员变量和方法时，亦基于指针偏移量获取到对应的内存数据。例如，下面的一段 Java 示例程序：

清单：示例

作用：Java 程序面向对象测试

```
class Animal{
    private Integer weight;//重量
    private Integer age;// 年龄

    // 测试程序：访问类的成员变量
    public static void main(String[] args){
        Animal dog = new Animal(30, 1);
        System.out.print("dog's age is: " + dog.getAge());
    }

    // 构造函数
    public Animal(Integer weight, Integer age){
        this.weight = weight;
        this.age =age;
    }

    public void setWeight(Integer weight){
        this.weight = weight;
    }
    public Integer getWeight(){
        return this.weight;
    }

    public void setAge(Integer age){
        this.age = age;
    }
    public Integer getAge(){
        return this.age;
    }
}
```

在 JVM 执行 Animal dog = new Animal(30, 1)时，会在 JVM 堆中为 dog 实例对象分配一段连续的内存空间，并根据构造函数所传入的数据对这段内存空间进行初始化，注意，这里是连续的内存空间。在 JVM 执行 System.out.print("dog's age is: " + dog.getAge())访问 dog.age 成员变量时，实际上 JVM 将 Java 程序中的 dog 处理成了一个指针，通过该指针，JVM 可以找到在堆中所分配的 dog 实例数据。Animal 类型包含 weight 和 age 这 2 个类成员变量，并且类型都是 Integer，

see more please visit: https://homeofpdf.com

因此 dog 在堆中的内存区域中持有对这两个对象的指针的引用，最终 JVM 通过 Integer 的指针获取到最终的 age 的值。

在 dog 实例所占用的连续的堆内存中，weight 和 age 这两个实例对象的指针相对于 dog 指针，具有不同的偏移量，偏移量不是在 JVM 运行期动态计算的，而是在 Java 程序的编译期，由编译器自动计算出来，JVM 最终通过指向 dog 实例的指针+偏移量，对 Java 类型的成员变量进行寻址，并对其进行赋值或取值操作。这就是 Java 语言面向对象的实现机制。后文会专门讲解具体的原理，这里旨在说明一点，JVM 在内存上建立了一套 Java 面向对象的标准模型，在 JVM 内部，一切对 Java 对象实例及其成员变量和成员方法的访问，最终皆通过指针得以寻址。同理，JVM 在传递 Java 函数参数时，所传递的也只不过是 Java 入参对象实例的指针而已。简而言之，传递的是指针。

正因为 Java 函数传参，实际所传递的只是指针，而在物理机器层面，不管何种数据类型的指针，其宽度都是相同的，指针的宽度仅与物理机器的数据总线宽度有关，而与具体某种编程语言中的具体数据类型无关。例如，在 C 语言中，不管是 char*类型的指针，还是 int*类型的指针，还是某个自定义的结构体类型的指针，这些指针的宽度完全相同。在 32 位平台上，指针宽度一定是 32 位；在 64 位平台上，指针宽度一定是 64 位。

因此，JVM 在为 Java 函数计算入参所需要的堆栈空间时，只需要入参的数量即可。

JVM 计算入参堆栈空间的指令如下：

```
__ movptr(rcx, parameter_size);
__ shlptr(rcx, Interpreter::logStackElementSize);
```

这两行代码最终会生成如下机器指令：

```
movl 0x20(%ebp), %ecx
shl $0x2, %ecx
```

movl 0x20(%ebp), %ecx 这条指令的含义是，将 ebp 栈基地址往高地址方向偏移 32 位处的数据（也即 parameterSize 变量的值）传送到 ecx 寄存器中。注意：0x20 是十六进制的写法，换算成十进制就是 32。ecx 是 CPU 中的一个普通寄存器，可以被用于保存临时变量。

接着执行 shl $0x2, %ecx 这条指令，这条指令的含义是，将 ecx 寄存器中的值左移 2 位。对于二进制数据，左移 N 位，换算成十进制，就是将所对应的十进制数乘以 2 的 N 次方，因此左移 2 位就表示将 ecx 寄存器中的数值乘以 4。为何要乘以 4？因为在 32 位平台上，每一个入参指针都占用 32 位内存，4 字节。0x20(%ebp)处的数据是 parameter_size，每一个 parameter 指针占用 4 字节，因此最终要将其乘以 4。

see more please visit: https://homeofpdf.com

shl $0x2, %ecx 所对应的 C 代码是__ shlptr(rcx, Interpreter::logStackElementSize)，logStackElementSize 定义在 globalDefinitions.hpp 文件中，定义如下：

```
#ifdef _LP64
const int LogBytesPerWord    = 3;
#else
const int LogBytesPerWord    = 2;
#endif
```

logStackElementSize 兼容了 32 位和 64 位平台，如果是 64 位平台，值是 3，否则是 2。最终的效果是，如果在 64 位平台上，就将 Java 函数入参数量左移 3 位，相当于乘以 8，这表示每一个入参都占用 8 字节堆栈空间。同理，如果在 32 位平台上，就将 Java 函数入参数量左移 2 位，相当于乘以 4。

CallStub()（即 stubGenerator_x86_32 : generate_call_stub()函数）执行完__ movptr(rcx, parameter_size)和__ shlptr(rcx, Interpreter::logStackElementSize)之后，就计算出即将被调用的 Java 函数入参所需要的堆栈空间。但是，CallStub()还要保存调用者的数据段现场，这些用于保存调用者所执行到的 Java 程序所对应的机器指令的基址和变址，因此 CallStub()接着会执行下面这行代码：

```
__ addptr(rcx, locals_count_in_bytes);
```

这主要用于保存 rdi、rsi、rbx、mxcsr 这 4 个寄存器的值。这行代码最终会被翻译成下面的机器指令：

```
add $0x10, %ecx
```

在 32 位平台上，由于这 4 个寄存器各占 4 字节的内存，因此需要再将 ecx 寄存器加上 16，最终 JVM 为将被调用的 Java 函数所分配的堆栈空间会再增加 16 字节大小。

基址和变址用于 Java 字节码取指，一个 Java 函数对应若干条字节码指令，而一条 JVM 字节码指令由若干机器指令组成（准确地说，是转换为若干机器指令），物理机器能够自动取指，但是无法对 JVM 字节码进行自动取指，因此对 JVM 字节码的取指机制需要由 JVM 自己去实现。后面会对此进行详细分析。

上面的指令旨在计算出将被调用的 Java 函数入参所占用的堆栈空间，可以看到，最终所占用的空间大小为：

$$\text{Java 函数入参数量} \times 4 + 4 \times 4$$

4×4 就是最后 rdi、rsi、rbx、mxcsr 这 4 个寄存器所占用的堆栈空间大小。

完成了 Java 函数入参空间计算后，接下来就需要执行最主要的一步：动态分配堆栈内存。这一步很简单，直接执行 sub operand, %esp 即可实现，operand 就表示刚才计算出来的堆栈大小。

see more please visit: https://homeofpdf.com

在 CallStub()（即 stubGenerator_x86_32 : generate_call_stub()函数）中使用下面的 C 代码实现：

```
__ subptr(rsp, rcx);
```

这行代码最终会生成下面这条机器指令：

```
sub %ecx, %esp
```

在这条指令之前，JVM 所计算出的堆栈空间大小保存在 ecx 寄存器中，因此这里直接将 esp 减去 ecx 寄存器的值，就完成堆栈空间的分配。

为了加速内存寻址和回收，物理机器在分配堆栈空间时都会进行内存对齐，JVM 也保留了这一原则，因此完成堆栈内存分配后，接着 CallStub()执行下面这行代码：

```
__ andptr(rsp, -(StackAlignmentInBytes));
```

其最终对应的机器指令如下：

```
and $0xfffffff0, %esp
```

堆栈按 16 位对齐，相当于减去后 4 位的值。如果前面为堆栈空间分配的字节数不够 16 的整数倍，这里就会减小 esp 寄存器的值，使其按 16 位对齐。

至此，JVM 完成了动态堆栈内存分配，这是 JVM 最具里程碑意义的事件！

这一关迈过去之后，JVM 终于跨越 C/C++程序与 Java 程序之间的桥梁，翻越中间的分水岭，纵身一跃，开始要进入 Java 程序的领域“地界”中了。进入 Java 的世界，向着詹爷当年的伟大目标大步前进！作者每每阅读至此，总会有种神圣的使命感，更有一种踏上伟大征程的激情！这一刻，完全可以比肩当年法拉第向戴维呈现会议演讲稿，那个足以改变人类社会和生活的方方面面，使人类迈向现代化的历史时刻；完全可以比肩当年爱因斯坦发表相对论，那个足以推翻经典物理学，使人类全新认识物理、宇宙、时间并因此而得以进入太空探索的历史时刻。因为这一刻，IT 界即将发生翻天覆地的变革，若干程序员不用再面对内存、指针、寄存器，商业编程的门槛极大地降低，信息化得以飞速发展，移动互联网得以普及，并因此带动大数据、人工智能、物联网、云计算等热门领域的深度发展，因此再次改变人类的消费、交易等社会活动方式，深刻变革经济政治的内在结构和布局，使人类向着更加高级的文明进化。虽然肯定不少人会觉得我言过其实，IT 界各种伟大的发明实在太多，但 Java 的确是站在了巨人的肩上。如同不能因为法拉第和爱因斯坦太过伟大就否认为此做出各种铺垫研究的前人的惊人成就，同样不能否认 Java 所依赖的全部重要变革的成就，但是的确要承认，Java 的出现推动了很多领域的高速发展。

十万个感叹号，向詹爷致敬！！！

see more please visit: https://homeofpdf.com

5. CallStub：调用者保存

JVM 为即将被调用的 Java 方法分配了堆栈空间，调用者是 CallStub()所指向的函数，其实就是 stubGenerator_x86_32 : generate_call_stub()。接下来 JVM 就要将 CPU 的控制权转交给被调用的 Java 方法，但是在转交之前，调用者需要保存自己的寄存器数据，这些寄存器主要包括：edi、esi、edx。

这里稍微交代下关于汇编的部分背景知识，没兴趣的小伙伴大可略过不看，不影响后续理解。懂汇编的朋友都知道，内存中一切数据都是二进制，不管是“真的”数据，还是机器指令，都是数据。如果不加以区分，你可以将一个机器指令当做一串普通的数字，也可以将一个数据看成是一个特定的机器指令。计算机区分内存中的一块数据到底是机器指令，还是普通的数据，主要取决于 CS:IP 寄存器，被 CS:IP 寄存器所指的内存数据就是机器指令，会被 CPU 执行，否则就是数据，CPU 可以对其进行数据传送。

每次在执行函数调用时，CS:IP 寄存器会从当前调用者函数的机器指令处跳转到被调用函数，这样 CPU 才能执行被调用的函数。但是当被调用函数执行完了后，CPU 需要跳转到调用者函数中继续执行调用者函数的机器指令，换言之，需要恢复 CS:IP 的值，使之重新指向调用者函数中执行被调用函数的下一条指令。如何恢复呢？前文讲过，每次发生函数调用时，机器会将调用者函数的 CS:IP 压入栈中，而在函数调用结束后，再次将调用者函数的 CS:IP 从栈中弹出来进行恢复。

物理机器通过 CS:IP 来区分一个内存中的数据到底是真实的数据还是机器指令，而对于数据，物理机器一般会用 edi 和 esi 分别保存目的偏移地址和源偏移地址。例如在字符串复制时，一段优化的汇编代码中会同时使用 edi 和 esi，分别指向目标字符串索引位置和源字符串索引位置。

而在 JVM 中，edi 和 esi 却被赋予了更多神圣的职责，例如在 Java 函数调用过程中，esi 会用于 Java 字节码寻址。每当 JVM 开始执行 Java 函数的某个字节码指令时，JVM 会首先将 esi 寄存器指向目标字节码指令的偏移地址，然后 JVM 跳转到该字节码所对应的第一个机器指令开始执行。

所以 edi 和 esi 在 JVM 中是与调用者函数紧密关联的寄存器，是调用者函数的私有数据。

ebx 是一个通用的寄存器，但是也经常被用来作为一段数据的基地址，例如使用汇编对一个一维数组的成员元素进行寻址，可以将 ebx 定位到这个一维数组的起始地址，然后使用一个变址定位到数组中的某个元素。在 JVM 中，ebx 便被赋予了这种非常实际的作用，在执行 Java 函数调用时，ebx 会用来存放 Java 函数中即将被执行的字节码指令的基地址，然后通过 jmp 指令跳转到该字节码位置进行字节码解释执行。因此 ebx 也会与 edi、esi 一样，与调用者函数息

see more please visit: https://homeofpdf.com

息相关，也是调用者函数的私有数据。

既然 esi、edi、ebx 都属于调用者函数的私有数据，因此在发生函数调用之前，调用者函数必须将这些数据保存起来，因为在被调用者函数中，这些数据也会被被调用函数所使用，其中的数据会被被调用函数修改，这样，当被调用函数执行完毕，程序流重新回到调用者函数中时，如果调用函数之前没有保存这些数据，这些数据就无法恢复，从而使程序发生异常。

esi、edi 和 ebx 的保存并不是必须的，例如随便写一段 C 程序然后编译，编译器往往并不会保存这些数据，因为很多编译器只会使用有限的寄存器保存函数特有的数据，而不会使用 esi、edi 和 ebx 这 3 类寄存器。

保存的方式有很多种，可以将其保存到应用程序的堆中，也可以保存到栈中。由于 esi、edi 和 ebx 可以被看做是调用者函数的私有数据，因此 JVM 直接将其保存到了被调用者函数的堆栈中。注意，是保存到了被调用函数的堆栈中，而不是调用函数的堆栈中。关于这一点，只是一种约定俗成的做法，其实保存到调用者函数的堆栈中也是可以的。无论保存到谁的堆栈中，只要在被调用函数执行完之后系统能够恢复调用者函数的这些私有数据即可。

保存 esi、edi、ebx 很简单，如下：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
        //…

        // save rdi, rsi, & rbx, according to C calling conventions
        //保存 rdi
        __ movptr(saved_rdi, rdi);
        //保存 rsi
        __ movptr(saved_rsi, rsi);
        //保存 rbx
        __ movptr(saved_rbx, rbx);

        //…
}
```

这几条模板最终会生成如下机器指令（使用汇编助记符表示）：

```
mov    %edi,-0x4(%ebp)
mov    %esi,-0x8(%ebp)
mov    %ebx,-0xc(%ebp)
```

这 3 条指令执行之后，堆栈布局如图 2.24 所示。

see more please visit: https://homeofpdf.com

图 2.24　堆栈布局

注意，JVM 还保存了 mxcsr 寄存器，这属于 Intel 的 SSE 技术，该议题比较高级，如果要详细讲的话估计会占去不少篇幅，有兴趣的小伙伴可以自行研究，这里就不“歪楼”了。

以上过程有一个专门的术语，叫作“现场保存”。当调用者函数的现场全部保存完之后，CPU 的控制权马上就要移交给被调用者函数了。

6．CallStub：参数压栈

前面在分析“动态分配堆栈”时，分析出 CallStub 函数指针为即将调用的函数分配的堆栈空间大小为：

$$\text{Java 函数入参数量} \times 4 + 4 \times 4$$

4×4 就是最后 rdi、rsi、rbx、mxcsr 这 4 个寄存器所占用的堆栈空间大小。

CallStub 为被调用者函数所分配的堆栈空间大小完全取决于 Java 函数的入参数量，为了分

see more please visit: https://homeofpdf.com

析内存空间分布情况，这里假设即将被调用的 Java 函数包含 3 个参数，则执行完前面两步（分别是动态分配堆栈和保存调用者现场）之后，堆栈实际布局如图 2.25 所示。

图 2.25　CallStub 分配堆栈和保存调用者现场后的内存布局

CallStub 为调用者分配的堆栈空间还剩余 3 个数据需要填充，接下来 JVM 要做的就是将即将被调用的 Java 函数的入参复制到这剩余的堆栈空间里去。

既然要进行数据复制，CallStub 至少要知道两点：

◎　即将被调用的 Java 函数的入参数量有多少。
◎　即将被复制的 Java 函数的参数集合在哪。

这两个要素在进入本流程之前，全都已经计算得到，并且作为 CallStub 的参数传递给了 CallStub 所指向的函数。在图 2.25 中，28(%ebp)代表的堆栈位置保存的便是 Java 函数的入参数量，而 32(%ebp)所代表的堆栈位置保存的便是 Java 函数的第一个入参。

see more please visit: https://homeofpdf.com

由于不同的 Java 函数的入参数量是不同的，因此 CallStub 使用了循环进行处理，而这种循环直接是基于机器指令的。在机器层面进行循环，一个约定俗成的做法是将循环次数暂存到 ecx 寄存器。因此 CallStub 必然要先将 Java 函数的入参数量传送到 ecx 寄存器中。同时，由于 Java 函数的多个入参在内存中实际上是一个队列，是一个“串”，而对串的寻址通常使用基址+变址的偏移寻址指令，因此在 CallStub 中将以 edx 寄存器存储基址，以 ecx 存储变址。

对于 Java 函数的入参队列而言，所谓基址，实际上就是第一个入参的内存地址。由于 Java 语言是面向对象的，因此其入参队列在 JVM 里实际上是指针的队列，每一个指针指向不同的入参实例。指针的宽度都是相同的，因此只要知道了第一个入参的位置，便可以知道其后续所有入参的位置，只需要基于第一个入参位置进行偏移即可。

假设第一个入参位置记为 $P1$，则其后面第 N 个入参的位置是：

$$P1 + (N - 1) * 4$$

根据这个很简单的原理，CallStub 只要将 $P1$ 作为基址，将 N 作为变址就能对 Java 函数的全部入参进行寻址。所以 CallStub 将 Java 函数的参数进行入栈的第一步就是分别获取基址和变址：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
        //…

        // pass parameters if any
        Label parameters_done;
        __ movl(rcx, parameter_size);  // 参数数量
        __ testl(rcx, rcx);
        __ jcc(Assembler::zero, parameters_done);

        // parameter passing loop
        __ movptr(rdx, parameters);    // 第一个入参地址
        __ xorptr(rbx, rbx);

        //…
}
```

这段模板在 JVM 启动过程中会生成如下机器指令（使用汇编助记符给出）：

```
//将 32(%ebp)处的数据传送给 ecx 寄存器
//32(%ebp)保存的是 Java 函数入参数量，即 parameter_size
mov    0x20(%ebp),%ecx

//校验 parameter_size 是否为 0，若是 0 则直接跳过参数处理
```


see more please visit: https://homeofpdf.com

```
test    %ecx,%ecx
je      0xb370b68b;;该地址每次启动 JVM 时都不一样

//将 28(%ebp)处的数据传送给 edx 寄存器
//28(%ebp)保存的是 Java 函数的第一个入参指针
mov     0x1c(%ebp),%edx

//把%ebx 设为 0
xor     %ebx,%ebx
```

此时物理寄存器（注意，不是逻辑寄存器哦）中所保存的重要信息如下所示：

寄存器名	指　向
edx	parameters 首地址
ecx	Java 函数入参数量

一切准备就绪，开始循环将 Java 函数参数压栈：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
        //…

        // pass parameters if any
        Label loop;
        __ BIND(loop);

        // get parameter
        __ movptr(rax, Address(rdx, rcx, Interpreter::stackElementScale(), -
wordSize));
        __ movptr(Address(rsp, rbx, Interpreter::stackElementScale(),
                    Interpreter::expr_offset_in_bytes(0)), rax);          //
 store parameter
        __ increment(rbx);
        __ decrement(rcx);
        __ jcc(Assembler::notZero, loop);

        //…
}
```

这段模板最终会生成下面的机器指令（使用汇编助记符表示）：

```
mov     -0x4(%edx,%ecx,4),%eax
mov     %eax,(%esp,%ebx,4)
inc     %ebx
dec     %ecx
jne     0xb370b696
```


see more please visit: https://homeofpdf.com

在机器层面进行循环，一般有两种方式：一种是使用 loop 指令；另一种则使用跳转。很显然这里使用了跳转。

熟悉汇编的小伙伴可能看出来这段汇编中对循环因子 ecx 做的是减法（dec %ecx，表示减去 1），而常见的循环中一般是做加法，一般使用 inc %ecx。这主要是因为 CallStub 对 Java 函数的入参采取的是逆向遍历，也就是从后往前遍历参数，并将读取到的入参传送到堆栈中。

上面这段太“底层”了，很显然这会让没有汇编基础的小伙伴们难以理解，所以还是通过举例子的方式来说明。

假设被调用的 Java 函数包含 3 个入参，分别使用 arg1、arg2 和 arg3 表示，则 CallStub 指针所指向的函数的 parameters 入参指向 Java 函数实际存储 Java 函数 3 个入参的内存区的首地址。此时堆栈空间布局如图 2.26 所示。

图 2.26　Java 函数参数入栈之前的堆栈布局

see more please visit: https://homeofpdf.com

当第一轮循环完成之后，Java 函数的第三个入参被压栈，如图 2.27 所示。

图 2.27　Java 函数参数第一轮压栈

see more please visit: https://homeofpdf.com

第二轮压栈后，堆栈空间布局如图 2.28 所示。

图 2.28　Java 函数参数第二轮压栈

see more please visit: https://homeofpdf.com

第三轮压栈后，堆栈空间布局如图 2.29 所示。

图 2.29　Java 函数参数第三轮压栈

至此，Java 函数的 3 个入参全部被压入栈中。离 Java 函数的调用越来越近了。

7．CallStub：调用 entry_point 例程

前面经过调用者框架栈帧保存（栈基）、堆栈动态扩展、现场保存、Java 函数参数压栈这一系列的逻辑处理，JVM 终于为 Java 函数的调用演完前奏，一切就绪，就等着吹响进攻的冲锋号。而负责吹响冲锋号的，就是 entry_point 例程。

关于到底啥是 entry_point 例程，为何要取这么一个古里古怪的名字，放到后文再解释。这里只需要知道这是一个与 CallStub 一样的例程即可。

see more please visit: https://homeofpdf.com

在 JVM 调用 CallStub 所指向的函数时，已经将 entry_point 例程的首地址传递给 CallStub 函数了，作为其第 5 个入参。entry_point 其实也是一个指向函数的指针，对于 CPU 而言，只要能够拿到函数的入口地址，就能执行函数调用。调用的指令很简单，就是 call。我们来看 CallStub 是如何执行 entry_point 调用的：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
        //...

        // call Java function
        __ BIND(parameters_done);
        __ movptr(rbx, method);          // get methodOop
        __ movptr(rax, entry_point);      // get entry_point
        __ mov(rsi, rsp);                // set sender sp
        BLOCK_COMMENT("call Java function");
        __ call(rax);

        //...
}
```

这段模板代码最终会被翻译成下面这段机器指令（使用汇编助记符表示）：

```
//将 method 首地址传送给 ebx 寄存器
mov    0x14(%ebp),%ebx

//将 entry_point 传送给 eax 寄存器
mov    0x18(%ebp),%eax

//将当前栈顶保存到 esi 寄存器中
mov    %esp,%esi

//调用 entry_point
call   *%eax
```

这段机器指令很简单，主要将 Java 函数所对应的 method 对象的首地址保存到 ebx 寄存器中，同时将 CallStub 栈顶地址保存到 esi 寄存器中。

眼尖的小伙伴们会发现，到这里为止，貌似 CallStub 函数入参的一半都已经被传送到相关寄存器中保存起来了，那么，其他入参的去向如何呢？下面给出了到目前为止 CallStub 函数所有入参的去向一览表（见表 2.5），表中清楚地显示了各入参的现状。

see more please visit: https://homeofpdf.com

表 2.5　CallStub 函数入参

参　数	保存位置
link	仍在堆栈中 8(%ebp)
result_val_address	仍在堆栈中 12(%ebp)
result_type	仍在堆栈中 16(%ebp)
method	被传送到 ebx 寄存器中，原本在堆栈中 20(%ebp)
entry_point	被传送到 eax 寄存器中，原本在堆栈中 24(%ebp)
parameters	被传送到 edx 寄存器中，原本在堆栈中 28(%ebp)
size_of_parameters	被传送到 ecx 寄存器中，原本存在于堆栈中 32(%ebp)
CHECK	仍在堆栈中 36(%ebp)

CallStub 中的 method、entry_point、parameters 和 size_of_parameters 这 4 个入参被从堆栈中传送到寄存器中。你不禁要问这样一个问题，为何这几个入参需要从堆栈转移到寄存器中呢？如果一直基于 ebp 寄存器偏移量去寻址不是很好么？

这主要是因为这个参数在即将被调用的 entry_point 例程中会被使用，而从 CallStub 例程“跳转”到 entry_point 例程时使用的是 call 指令，而 call 指令一般都会使用“套路”，会出“组合拳”，配合 call 指令一起使用的是 push %ebp 指令，该指令会将调用函数的栈帧保存起来。配合 call 指令的另一个“组合拳”自然就是 move %esp, %ebp，其将被调用函数的栈帧指向调用函数的栈顶。这套组合拳打出去之后，调用函数的栈帧指针 ebp 就会被改变，因此从 CallStub 例程进入 entry_point 例程之后，要想再基于 ebp 进行变址寻址，就无法获取到 method、parameters 等参数，因此 CallStub 只能将这 4 个参数先复制到寄存器中暂存起来，在 entry_point 例程中通过读取这几个寄存器便可恢复这几个参数数据。

不过又有一个问题，为何 CallStub 的另外 4 个参数不用通过寄存器临时存储起来呢？这是因为另外 4 个参数在 entry_point 例程中不会被使用到，entry_point 例程不关心这几个数据，只有 CallStub 例程关心。CallStub 调用完 entry_point 例程再回到 CallStub 例程后，ebp 又重新指向 CallStub 例程的栈帧，因此通过 ebp 栈帧进行变址选址，依然能够获取这 4 个参数。

在 CallStub 执行 call %eax 指令之前，物理寄存器（注，不是逻辑寄存器哦）中所保存的重要信息如表 2.6 所示。

表 2.6　物理寄存器信息

寄存器名	指　向
edx	parameters 首地址
ecx	Java 函数入参数量

see more please visit: https://homeofpdf.com

续表

寄存器名	指　向
ebx	指向 Java 函数，即 Java 函数所对应的 method 对象
esi	CallStub 栈顶
eax	entry_point 例程入口

此时 eax 寄存器已经指向了 entry_point 例程入口，因此 CallStub 只需执行 call %eax 指令便可以直接跳转到 entry_point 例程，去执行 entry_point 例程。

在 entry_point，还会经过一段“铺垫”性的逻辑处理，并最终寻找到 Java 函数的第一个字节码并从第一个字节码开始执行。

在前面的叙述中，多次提到 CallStub 的一个入参——method 对象，该对象代表即将被调用的 Java 函数，通过该对象可以寻址到 Java 函数所对应的第一个字节码指令。那么这个对象到底是啥，在 entry_point 中究竟如何使用呢？

要理清楚这些问题，不得不先研究清楚 JVM 的另一个重要的主题——内存模型。将 JVM 的内存模型理清楚之后，method 对象的结构、Java 类的内存结构等概念都会被逐一破解，这也是理解 entry_point 例程所必须掌握的基础。没有这个基础，研究 entry_point 无异于盲人摸象。

8．CallStub：获取返回值

CallStub 执行 entry_point 例程调用时，使用的是 call 指令，而非 jmp，因此最终 entry_point 例程执行完毕之后，CPU 的控制权还是会回到 CallStub，继续执行 entry_point 例程调用之后的指令。

调用完 entry_point 例程之后，会有返回值，CallStub 会获取返回值并继续处理。

本章前面详细描述了物理机器执行 call 调用的原理，简而言之就是，物理 CPU 执行 call 调用时，会将 ip 压入栈中。ip 是一种专门的段寄存器，通常与 cs 段寄存器一起用于指向下一条即将被 CPU 执行的指令地址，这样当被调用函数执行完成之后，CPU 只需要从栈顶取出 ip，便能重新定位到调用函数的下一条指令地址，继续执行调用函数中的逻辑。

当 CallStub 执行完 call *%eax 这条指令后，堆栈内存的布局如图 2.30 所示（假设目标 Java 函数包含 3 个入参）。

在 JVM 内部，调用函数被压栈的 ip 寄存器值有一个专门的称谓——return address，即返回地址。注意，返回地址与“返回值”是两个不同的概念，返回值是指被调用函数所返回的结果值，而返回地址则是指调用函数执行被调用函数所对应指令的下一条指令的内存地址。在下一章讲解 entry_point 时还会对此进行细述。

see more please visit: https://homeofpdf.com

理解清楚这一点，在下一章就好理解 entry_point 例程中的“return address”这个参数了。事实上，在图 2.30 所示这种堆栈内存布局中，最下面的 eip 在 entry_point 中的描述统一变成了“return address”，因此在下一章所绘制的堆栈内存布局图中，这个存储单元的名称便也统一改成“return address”，诸君谨记！在后续流程中，JVM 为了让被调用函数的入参、局部变量、固定帧以及操作数栈在内存上相连，会不断移动 eip 在堆栈中的位置，各位道友只需要知道，在后续章节中所描述的 return address 这个东东，就是紧跟在 call *%eax 后面的那条指令的地址。

图 2.30　CallStub 执行完 call *%eax 指令后的内存布局图

CallStub 中接下来的两条指令如下：

```
◎ mov    0xc(%ebp),%edi  //result
◎ mov    0x10(%ebp),%esi //result_type
```


see more please visit: https://homeofpdf.com

这 2 条指令分别读取被调用函数所返回的值 result 和数据类型 result_type。看看图 2.30 所示的内存布局图就知道了，0xc(%ebp)和 0c10(%ebp)这 2 个堆栈位置所保存的正是 resultAddress 与 result_type。JVM 将这 2 个值分别存储进 edi 和 esi 这 2 个寄存器中，调用方在获取被调用函数的返回值与返回类型时，也会从这 2 个寄存器中读取，这种完全是基于约定的技术实现方式。

9. CallStub：汇编指令总览

所谓例程，就是一段预先写好的函数，JVM 通过例程函数在启动过程中生成机器指令，当执行 Java 函数调用时，JVM 直接跳转到例程所生成的这段机器指令去执行。CallStub 例程最终生成的机器指令如下（使用汇编助记符表示，笔者运行平台是 Linux x86）：

```
//保存调用者栈帧
push    %ebp
mov     %esp,%ebp

//动态分配堆栈
mov     0x20(%ebp),%ecx
shl     $0x2,%ecx
add     $0x10,%ecx
sub     %ecx,%esp
and     $0xfffffff0,%esp

//保存 edi、esi、ebx 这 3 个寄存器
mov     %edi,-0x4(%ebp)
mov     %esi,-0x8(%ebp)
mov     %ebx,-0xc(%ebp)

//保存 mxcsr 寄存器的值，属于 SSE，在 VS 中的寄存器窗口右击，然后选择 SSE 就可以看到了
stmxcsr -0x10(%ebp)

//循环遍历 Java 函数入参，并传送到 CallStub 堆栈中
mov     0x20(%ebp),%ecx
test    %ecx,%ecx //parameter_size 是 0，直接跳过参数处理
je      0xb370b68b
mov     0x1c(%ebp),%edx //对应 parameters
xor     %ebx,%ebx //把%ebx 设为 0

//开始循环
mov     -0x4(%edx,%ecx,4),%eax
mov     %eax,(%esp,%ebx,4)
inc     %ebx
dec     %ecx
jne     0xb370b696

//开始 entry_point 例程调用
mov     0x14(%ebp),%ebx //对应 method
```


see more please visit: https://homeofpdf.com

```
mov    0x18(%ebp),%eax //对应 entry_point
mov    %esp,%esi
call   *%eax

// call_stub_return_address:
mov    0xc(%ebp),%edi  //result
mov    0x10(%ebp),%esi //result_type
```

call_stub 例程的讲解至此先告一段落。到目前为止，各位道友仍然没有看到 JVM 内部如何从 C/C++程序完成 Java 函数的调用，因为这部分逻辑需要在后文讲解完 entry_point 例程后才能完全揭开它的真相，但是大家至少可以知道 JVM 是如何将 Java 函数入参压入到机器层面意义上的方法堆栈之中的。其实，这部分压入的参数，便形成了 Java 方法堆栈中的“局部变量表”的一部分，关于局部变量表的概念会在后面讲解 Java 方法堆栈时进行详解。

总之，万丈高楼平地起，想要研究清楚 JVM 执行引擎的内部实现机制，call_stub 这个例程是无论如何也绕不过去的，该例程对 JVM 执行引擎的实现也起到至关重要的桥梁作用，让程序流从 JVM 的世界直接“穿越”进入 Java 的世界。

事实上，在 JVM 规范中，也提供了函数调用的好几种字节码指令，例如在调用 Java 静态方法和类成员方法时，或者 native 方法时，所生成的字节码函数调用指令是不同的，其中部分函数调用的指令在执行时，最终也会经过 call_stub 这个例程，因此弄懂 call_stub 例程的实现机制，对于研究 JVM 规范中的其他函数调用字节码指令的执行原理大有好处，能够启迪思维，然后触类旁通。

2.6　本章总结

总体而言，本章的内容放在全书中都是属于难度非常高的（后面还有两章的难度非常高，即 JVM 的字节码执行原理和 Java 方法栈帧这两章），涉及到大量汇编和 C 语言的基础知识，尤其是涉及到函数指针这种非常具有挑战性的技术，相信很多即使做了多年 C/C++开发的底层开发者可能都很少接触到函数指针的应用，更别提其实现原理了。而函数指针正是 JVM 内部实现 C/C++程序直接执行物理机器指令的关键技术，没有之一！所以对于广大非常想研究 JVM 执行引擎机制的道友而言，汇编和 C 语言的这一关是必须要闯过去的，否则你的梦想可能永远都实现不了。

本章主要讲解了 JVM 内部的 call_stub 例程的定义和调用机制，但是并没有一开始便直入主题，主要是考虑到技术的难度太大，很多人理解不了。因此作者只能用心良苦地设计了很多 C 和汇编程序，从物理机器执行函数调用的机制开始讲起，为大家揭开物理机器在调用函数时

see more please visit: https://homeofpdf.com

涉及的若干细节。

接着抛出问题：若想要在 C/C++程序中直接执行本地机器指令，技术上该如何实现呢？答案是函数指针。只有通过函数指针才能实现这一目标。而 JVM 内部正是通过 call_stub 这个函数指针实现从 JVM 的世界穿越进 Java 的世界。

理解了 call_stub 函数指针的设计背景之后，本章接着详细分析了 call_stub 函数指针的声明、定义和对应例程的实现机制，花了大量篇幅讲解 call_stub 例程的 pc()函数、入参定义、现场保存、堆栈空间动态分配、参数压栈以及对 entry_point 例程的调用。不仅详细分析了这些关键步骤的技术实现，还阐述了为何需要这样实现。可以这么说，Java 语言本身的设计宗旨（跨平台和面向对象）决定了 JVM 内部必须要能够由 C/C++程序直接执行本地机器指令，而这种技术的实现又决定了 call_stub 内部必须这么实现，别无他法。

其实，JVM 与其他任何一个成功的人或物一样，都是满满的套路，套路很深哟！

see more please visit: https://homeofpdf.com