

第9章

执行引擎

本章摘要

- ◎ JVM 的取指与译码机制
- ◎ 栈顶缓存原理
- ◎ 操作数栈与栈帧重叠技术
- ◎ JVM 指令集特点与实现

可以这么说，在 JVM 内部，最精华的部分便是执行引擎（当然，GC 也绝对是精华）。几乎每一款 JVM 都在执行引擎上面下足了功夫，从最原始低效的字节码解释器，到模板解释器，再到 JIT 即时编译器，许多大神提出了各种优化策略和理论来提升 Java 程序的执行速度。有些公司出品的 JVM 直接将 Java 程序翻译成本地机器码，而在安卓体系中，也使用了 AOT 技术来提升运行时效率。

JVM 的执行引擎本身是一个相当复杂深奥的模型，对于从来没有涉及过底层的广大程序员而言，想理解它十分困难。笔者力图使用比较浅显的语言来将 JVM 执行引擎实现的技术细节清楚地描述出来，使广大道友不仅仅局限于理论研究，而是能够真正一窥其具体的技术内幕，不仅能够闻其道，而且能够知其术，做到知行合一。往往越是高深精妙的理论，如果仅仅研究理论，往往越容易让人迷糊，而配合着源码看，便能真正理解这些深奥的理论。

本书在描述 Java 执行引擎时，将试图对照物理机器 CPU 的执行机制，例如指令集、取指机制、程序计数器等，从物理机器 CPU 执行的角度看 Java 的执行引擎，通过这样的对比，希望能够让各位道友不要陷入 JVM 那复杂深奥的引擎模型里面，而是能够站在一个较高的角度看待问题。

JVM 执行引擎毕竟只是个虚拟系统，本身并不具备真正的运算能力，其内部其实仍然需要依靠物理 CPU 才能完成运算功能，而物理 CPU 仅识别二进制机器指令，JVM 执行引擎既然需要依赖物理 CPU，就必然需要将字节码指令最终转换为二进制机器指令，因此下文在讲解 JVM 执行引擎的过程中，将不可避免地涉及汇编语言。而这也符合本书的宗旨——重点讲解 JVM 内部的具体技术实现，而非主要讲理论。

9.1 执行引擎概述

所谓执行引擎，就是一个运算器，能够识别所输入的指令，并根据输入的指令执行一套特定的逻辑，最终输出特定的结果。其实，相比于 JVM 这类虚拟机而言，物理实体机器也是有其特定的执行引擎的，物理机器的执行引擎便是 CPU(中央计算单元)。CPU 能够识别机器指令，并根据机器指令完成特定的运算。

物理 CPU 执行指令的流程是这样的：

(1) 取指。CPU 的控制器从内存读取一条指令并放入指令寄存器。物理机器指令一般由操作码和操作数组成，当然并不是所有的操作码都会有操作数。例如 mov ax, 1 这条机器指令，其中 mov ax 就是操作码，而 1 就是操作数，在 Intel 处理器上，这条指令所对应的十六进制数是 0xB8 01。

(2) 译码。指令寄存器中的指令经过译码，确定该指令应进行何种操作（由操作码决定），操作数在哪里（由操作数决定）。

(3) 执行。分两个阶段，“取操作数”和“进行运算”。

(4) 取下一条指令。修改指令计数器（亦称程序计数器），计算下一条指令的地址，并重新进入取指、译码和执行的循环。

只要操作系统一启动，CPU 便会一直循环往复地执行上述流程，当机器啥事也不干的时候，会进入“空转”的状态，但是并不会停止。这种机制说白了与汽车发动机一样，只要汽车一启动，发动机就会一直转下去，没挂档位的时候也会保持空转状态，如果发动机不转了汽车就熄火了。类似的机制还有很多，例如视窗系统会有一个静默线程一直保持无限的 while 循环，当收到外部消息（例如鼠标点击）时就对消息进行处理，如果一直没有收到消息就一直循环下去，以此来确保视窗程序一直运行下去。Web 服务器也是一样，会有一个线程一直保持循环，如果接收到客户端请求就启动新的线程/进程处理。

JVM 既然作为虚拟机，自然也得有这么一套虚拟的 CPU 执行机制，按照“取指→译码→执行→取下一条指令”这一流程循环往复地执行下去。不过 JVM 不像真正的操作系统那样，当

没事可干的时候让 CPU 保持空转，如果 JVM 所运行的 Java 程序执行完了，Java 程序的生命周期会终止，而 JVM 虚拟机本身也会退出。所以，如果想让 JVM 一直保持“空转”，只能在 Java 程序里的某个线程中一直保持空循环。Tomcat 这款 Web 应用服务器程序便是这种机制，否则一旦没有外部 http 请求过来，Tomcat 程序及其宿主 JVM 虚拟机都会“寿终正寝”。类似的，Hadoop、Spark 之类的分布式系统亦都有类似机制。

正因为 JVM 没有空转机制，因此 JVM 一旦启动，处理完自身的初始化逻辑，便会进入 Java 程序，执行 Java 的字节码指令。前文讲过，JVM 进入 Java 程序之前，会先确定 Java 程序的 main() 主函数及其所在的类，加载 Java 主类并执行 main() 主函数。在 JVM 调用 Java 的 main() 主函数的链路上，会经过 CallStub 例程和 zerolocals 例程，在 zerolocals 例程中，JVM 会为 Java main() 主函数创建栈帧，创建完栈帧，最终 JVM 会调用如下逻辑：

```
清单: /src/cpu/x86/vm/templateInterpreter_x86_32.cpp
功能: JVM 调用 Java 字节码指令
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
    // ...
    address entry_point = __ pc();
    // ...
    // 创建栈帧
    generate_fixed_frame(false);
    // ...
    // 跳转到目标 Java 方法的第一条字节码指令，并执行其对应的机器指令
    __ dispatch_next(vtos);
    //...
    return entry_point;
}
```

InterpreterGenerator::generate_normal_entry(bool synchronized) 函数在前文讲解 Java 方法的栈帧时，已经分析了其一部分逻辑，其中，详细地分析了这个函数中的 generate_fixed_frame() 函数，Java 方法栈帧的创建便是通过该函数实现的。当 JVM 调用 Java 主函数 main() 时，便是为 main() 主函数创建栈帧，创建完栈帧，接着又有一系列逻辑处理（例如，方法校验、调用计数等），最后会执行这个函数中的 __dispatch_next(vtos) 函数（准确地说，函数前面的 __ 是一个宏，为了简化，这些细节就不赘述了，有细节控的道友大可忽略）。从这个函数开始，JVM 将读取到 Java 主函数的第一条字节码指令，并执行第一条字节码指令所对应的机器指令，并由此进入

“轰轰烈烈”的 Java 程序的世界中去。`_dispatch_next(vtos)` 函数是一个平台相关的函数，在 32 位 x86 平台上，其对应的实现如下：

清单：/src/cpu/x86/vm/interp_masm_x86_32.cpp

功能：`dispatch_next()` 函数

```
void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
    load_unsigned_byte(rbx, Address(rsi, step));
    increment(rsi, step);
    dispatch_base(state, Interpreter::dispatch_table(state));
}
```

现在完全看不懂这个实现逻辑，下文会慢慢道来。其实，`dispatch_next()` 函数是 JVM 内部非常核心的一个函数，该函数的主要功能就是进行“取指”。前面刚刚讲过，JVM 虚拟机与真实的物理机器执行指令的流程完全一样，都是循环往复地执行“取指→译码→执行→取指”的过程，下面便围绕这个过程，详细描述 JVM 内部取指、译码和执行的实现细节。

9.2 取指

在研究 JVM 的“取指”机制之前，先了解下物理机器级别的取指方式。对于直接运行在物理机器上的软件程序，其经过编译后直接形成二进制的物理机器指令编码。当操作系统加载这个软件程序时，会在内存中为该程序创建如图 9.1 所示的数据。

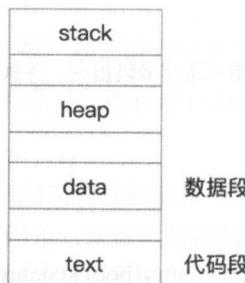


图 9.1 程序被操作系统加载后的内存映像

在一个基于段式内存管理的架构中，当操作系统将程序加载进内存之后，会将程序编译后的二进制代码指令存储到一个专门的区域——代码段（即图 9.1 中最下面的区域）。操作系统执行程序的过程，就是将代码段中的指令取出来逐个执行的过程。另外操作系统会将程序中的静态字段存储到数据段中，并为程序初始化堆栈空间，这部分内容超出了本书范围，有兴趣的道友可以另外自行研究操作系统编译原理。

操作系统会将软件程序的二进制机器码指令全部读进代码段中，当操作系统开始执行该软件程序时，CPU 便会读取代码段中的第一条机器指令，然后进入译码→执行→取下一条指令→译码→执行的循环，直到执行完该程序。

CPU 在取指时，先从程序的代码段中读取出操作码，在译码阶段，译码逻辑会判断该操作码，并从代码段中读取其所对应的操作数。例如，假设软件程序的二进制机器指令中有一条指令是 mov ax, 3，则 mov ax 是一个操作码，其操作数是 3。在取指时，CPU 首先读入 mov ax 这条操作码，译码器识别出该指令后面会跟着一个 32 位的数字（假设在 32 位 x86 平台上），于是译码器会接着从代码段内存区读取跟随在 mov ax 这条指令后面的操作数 3，如此完成整条指令的读取，接着交给运算器执行。当然，事实上 CPU 取指、译码和执行的机制是非常复杂的，这里将其简化，只是为了说明 CPU 的工作原理。

通过这个物理机器取指和译码的过程可知，物理机器要完成软件程序的运行，得具备 2 个条件：

- ◎ 内存中要存储软件程序编译后的指令。
- ◎ CPU 要能够识别出代码段中的指令和操作数。

对于第一点，操作系统运行软件之前，会将其源代码所对应的机器指令全部读进内存区。Java 程序启动后，JVM 也会将 Java 源代码所对应的字节码指令全部读进 JVM 内存中。而对于第二点，在我们的想象中，物理机器内部似乎要维护一张大而全的指令集表，每次 CPU 读入一个机器指令时，去扫描这张指令集表，从而识别出所读取到的机器码，并进一步判断该操作码后面是否有操作数。而事实上，物理机器内部这张所谓的指令集表并非仅仅是内存中的数据那么简单，物理机器“内置”的指令集其实是硬件结构，更具体地说是数字电路（呵呵，讨论得有点深，不过技多不压身啊），这些数字电路被集成进 CPU 内部，只要向 CPU 传递一个指令（0 和 1 的组合），CPU 就会依据其预先设定好的电路进行解码（高低电平），然后操作对应的寄存器或者某些电路去读取该指令操作码后面的操作数。同时，另一些电路则会被触发读取当前机器指令的下一条指令，如此一来，CPU 便能完成“取指→译码→执行→继续取指”的循环了。这便是 CPU 识别并执行机器指令的原理了。

在这一点上，JVM 基本也继承了这一思想（事实上想不继承都难，除非能够提出一种完全异于冯·诺依曼体系结构的数字计算机）。JVM 作为一款虚拟机，有其自己的一套指令集，这套指令集必定能够被 JVM 的虚拟运算器所识别，但是 JVM 并没有真正的硬件译码电路来识别 JVM 的这套指令，因此只能使用软件模拟。这种软件模拟的结果就是 JVM 需要使用软件的方式在内存中维护一套指令集，否则 JVM 无法识别 Java 方法所对应的指令操作码。这套指令集包含在下面的代码文件中：

清单：/src/share/vm/interpreter/bytecdodes.hpp

功能：JVM 指令集定义

```
enum Code {  
    _illegal          = -1,  
  
    // Java bytecodes  
    _nop              = 0, // 0x00  
    _acconst_null     = 1, // 0x01  
    _iconst_m1        = 2, // 0x02  
    _iconst_0          = 3, // 0x03  
    _iconst_1          = 4, // 0x04  
    _iconst_2          = 5, // 0x05  
    _iconst_3          = 6, // 0x06  
    _iconst_4          = 7, // 0x07  
    _iconst_5          = 8, // 0x08  
    _lconst_0          = 9, // 0x09  
    _lconst_1          = 10, // 0x0a  
    _fconst_0          = 11, // 0x0b  
    _fconst_1          = 12, // 0x0c  
    _fconst_2          = 13, // 0x0d  
    _dconst_0          = 14, // 0x0e  
    _dconst_1          = 15, // 0x0f  
    _bipush            = 16, // 0x10  
    _sipush            = 17, // 0x11  
    _ldc               = 18, // 0x12  
    _ldc_w             = 19, // 0x13  
    _ldc2_w            = 20, // 0x14  
    _iload             = 21, // 0x15  
    _lload             = 22, // 0x16  
    _fload             = 23, // 0x17  
    _dload             = 24, // 0x18  
    _aload             = 25, // 0x19  
    _iload_0           = 26, // 0x1a  
    //...  
}
```

这个枚举中定义了 JVM 的全部指令集，比如你熟悉的 iload、iconst_0 之类的，都包含在内。由于这些指令操作码定义在 C++ 的枚举类中，因此在操作系统加载 JVM 时便会将这些指令集读进内存之中，这便是 Java 虚拟机的执行引擎赖以运行的基础。在 JVM 运行期，Java 字节码的译码系统完全依赖于这套“软指令集”。

9.2.1 指令长度

既然 JVM 内部定义了这么一套指令集，是否就能完成执行引擎的功能呢？很显然，答案是不能的。仅有这么一套指令集，JVM 无法据此执行字节码指令。别说执行字节码指令，便连取指的功能都完成不了，为何？前面说过，物理机器 CPU 在读取机器指令时，会先读取指令中的操作码，CPU 会识别出操作码并据此判断操作码后面是否跟随有操作数。CPU 只有知道一个操作码后面是否跟随操作数，以及所跟随的操作数的大小，才能计算出下一条指令的位置，从而完成“取指”的功能。举个例子，例如一个 C 程序编译后包含下面 3 条机器指令：

```
mov ax, 1
mov ax, 2
mov ax, 3
```

这 3 条指令所对应的十六进制机器码如下：

```
0xB8 01
0xB8 02
0xB8 03
```

当 CPU 执行到第一条指令时，先读取 0xB8 这个操作码（这个操作码表示 mov ax），CPU 的译码电路“翻译”出这个操作码，并知道其后面会跟着一个操作数，该操作数的宽度是 32 位，CPU 据此便知道第二条指令的操作码的位置，第二条指令操作码的位置相对于第一条指令操作码的位置再往前移动 32 位，这是因为程序的机器码在内存中是连续存储的，于是 CPU 便驱动其内部的相关电路去内存中读出 mov ax 后面所跟随的操作数。同时，当 CPU 执行完第一条指令后，便能接着取下一条指令中的操作码，完成继续取指。

需要注意的是，对于计算机而言，无论操作码还是操作数，在内存中都只是一串 0 和 1 的组合而已（准确地说是一串高低电平），因此如果直接将一个数字交给 CPU，CPU 是无法知道这个数字所代表的到底是操作码还是操作数，例如 mov ax 这个操作码所对应的十六进制编码是 0xB8 (Intel CPU)，但是可能某个操作码后面所跟随的操作数也是 0xB8，因此如果直接将 0xB8 交给 CPU，CPU 是无法区分的。所以，当 CPU 在执行一段程序时，一定是先读取程序中的第一条指令的操作码并进行译码，计算该操作码后面是否跟随操作数以及操作数的数据宽度，如此 CPU 才能计算出下一条指令的起始位置并读取下一条指令中的操作码，然后继续译码，继续计算下一条指令的起始位置……，如此循环往复。这便是“取指”的关键所在。

妙也！

所以 CPU 要能够正确完成取指，不仅仅需要识别出操作码本身，还得知道操作码后面所跟随的操作数。

由于 JVM 完全继承了这一设计思想，因此也必须规定出每一条字节码指令后面所跟随的操

作用数及操作数的大小。很显然，上面在 `bytecodes.hpp` 中所定义的一套指令集并无这种规范，所以 JVM 必然在别的地方定义了这种规范，如：

清单：/src/share/vm/interpreter/bytecodes.cpp

功能：JVM 指令集定义

```
void Bytecodes::initialize() {
    // ...
    // bytecode      bytecode name      format   wide f.  result tp  stk traps
    def(_nop        , "nop"           , "b"     , NULL    , T_VOID  , 0, false);
    def(_aconst_null, "aconst_null"  , "b"     , NULL    , T_OBJECT, 1, false);
    def(_iconst_m1  , "iconst_m1"   , "b"     , NULL    , T_INT   , 1, false);
    def(_iconst_0   , "iconst_0"    , "b"     , NULL    , T_INT   , 1, false);
    def(_iconst_1   , "iconst_1"    , "b"     , NULL    , T_INT   , 1, false);
    def(_iconst_2   , "iconst_2"    , "b"     , NULL    , T_INT   , 1, false);
    def(_iload_1    , "iload_1"     , "b"     , NULL    , T_INT   , 1, false);
    //....
}
```

在该初始化函数中，JVM 定义了每个 `bytecode` 的名字（字符串）、`format`、字节码的返回结果 `result` 类型等。可是纵观这张表，并没有哪里明确指出每个字节码指令后面是否跟随操作数及操作数的宽度。其实秘密就藏在 `format` 这一列中，这一列记录了每一个字节码指令的总长度。例如，`_iconst_0` 这个字节码的 `format` 是 `b`，则表示该字节码指令，包括操作码和操作数，其总长度一共只有 1，由此可以推测，该字节码指令其实是没有操作数的。这里所谓的“长度为 1”，是指 1 字节，因为 Java 的每个字节码指令都仅占 1 字节，这也是为何 Java 字节码指令数量少于 256 个的原因。

再如 `bipush` 这条指令对应的 `format= “bc”`，则表示 `bipush` 这个操作码后面会跟一个宽度为 1 字节的操作数。同理，`sipush` 对应的 `format= “bcc”`，则表示 `sipush` 指令后面会跟一个宽度为 2 字节的操作数。这很好理解，因为 `bipush` 指令表示将一个 1 字节的数据推送至操作数栈栈顶，因此该指令后面的操作数仅占 1 字节，而 `sipush` 表示将一个 `short` 类型的操作数推送至栈顶，而一个 `short` 类型的数据占 2 字节。

既然推送一个 `short` 类型的数据至栈顶的字节码指令是 `sipush`，那么推送一个 `int` 类型的数据至栈顶的字节码指令是什么呢？直接看上面的字节码指令表是看不出来的，不过可以通过试验来验证：

清单：Test.java

功能：测试将 `int` 类型数据推送至栈顶的字节码指令

```
public static void tdd() {
    int a = 3;
}
```

该程序定义了一个 int 类型的变量 a，并赋初值为 3。由于为变量赋值的过程必定会有压栈和出栈的操作，并且为变量所赋的值直接是一个自然数，因此编译后的代码中一定会包含将整型变量推送至栈顶的字节码指令。

编译该程序，并使用 javap 命令分析编译后的字节码文件，结果输出如下：

```
public static void tdd();
Code:
Stack=1, Locals=1, Args_size=0
0:    iconst_3
1:    istore_0
2:    return
```

从分析结果可以看出，编译器使用 iconst_3 这条字节码指令将自然数 3 推送至栈顶。在刚才 Bytecodes::initialize() 函数中所定义的字节码指令格式表中找到 iconst_3，可以看到其 format = "b"。这说明 iconst_3 指令的总长度为 1，同时说明该操作码后面并没有操作数跟随。

在这里可以看到，自然数 3 虽然在源代码中被定义为整型，但是编译器直接使用一个特殊的字节码指令将其推送至栈顶，并没有使用“操作码 + 操作数”的方式来推送。JVM 如此设计的原因在于减小 Java class 的体积，字节码文件的体积减小了，其加载到内存后所占的内存空间也会降低，而 JVM 所牺牲的仅仅是在内存中多定义了一个字节码指令，以及为此多写了一段译码逻辑而已。这种牺牲是值得的，并且是非常划算的，毕竟字节码指令只定义了一次，但是 JVM 会加载成千上万个 Java 字节码文件，如果每一个字节码文件中都包含一个 int a = 3 这样的逻辑，并且假设 JVM 使用“操作码+操作数”的方式进行译码，那么每一条 int a = 3 这样的逻辑所对应的字节码，相比于 iconst_3 这套指令而言，除了字节码指令本身，还会额外多出来一个操作数，而一个操作数至少占 1 字节，虽然这一点内存空间不算啥，但是当 JVM 加载了成千上万个 Java 类时，这些多出来的内存空间累加起来就非常可观了。

事实上，JVM 为了节省空间，专门定义了 iconst_0 ~ iconst_5 这 6 条字节码指令，当将自然数 0 ~ 5 推送至栈顶时，便会分别生成这 6 条指令。其实这也是一种权衡，事实上 JVM 可以为每一个仅占 1 字节的数字分别定义一个特殊的字节码指令和译码逻辑，但是这样一来，指令和译码逻辑反而显得太臃肿，反而不美。

如果推送的整数大于 5，JVM 会如何处理呢？很简单，修改上面的 Test 类的 tdd() 方法中的 a 变量初始值，将其改成 6 看看。修改后编译 Java 类，并使用 javap 命令分析字节码文件，分析结果如下：

```
public static void tdd();
Code:
Stack=1, Locals=1, Args_size=0
0:    bipush   6
```

```

2:    istore_0
3:    return

```

可以看到，现在推送至栈顶的指令变成 bipush 6 了。现在的字节码指令格式终于变成了“操作码+操作数”这种范式。前面已经分析过，bipush 字节码指令的 format=“bc”，其长度为 2，表示 bipush 操作码后面会跟随一个只占 1 字节码宽度的操作数。那么如果要将一个宽度超过 1 字节的整数推送至栈顶呢？很简单，继续试验，将上面测试用例 Test.tdd()方法中的变量 a 的初始值改成 300，编译后的字节码指令如下：

```

public static void tdd();
Code:
Stack=1, Locals=1, Args_size=0
0:   sipush 300
3:   istore_0
4:   return

```

可以看到，现在推送至栈顶的指令变成了 sipush，该指令前面也讲过，其 format=“bcc”，表示该指令后面会跟随 1 个占 2 字节宽度的操作数。

事实上，如果将上面的 int a=3 改成 char a=3 或者 short a=3，最终所生成的字节码指令也是 iconst_3。同样地，若将 int a=300 改成 short a=300，所生成的字节码指令与 int a=300 所生成的一样，都是 sipush 300。

由此可以知道，其实 JVM 体系对内存空间的使用标准非常严格，从编译期便开始进行优化，能使用 1 个操作码完成的事，就绝不使用由 1 个操作码+1 个占 1 字节宽度的操作数所组成的指令去完成；能够使用由 1 个操作码+1 个占 1 字节宽度的操作数所组成的指令去完成的事，也绝不使用由 1 个操作码+1 个占 2 字节宽度的操作数所组成的指令去完成。

大善！

接着往下分析，如果要推送的整数比较大，超过了 2 字节的宽度，编译器会生成什么样的指令呢？这也是上面未验证完的问题。2 字节所能表示的最大无符号整数是 65535，那么如果在 Test.tdd()方法中这样定义变量 a：

```
int a = 65539;
```

所生成的字节码指令会是什么呢？

编译 Test 类并使用 javap 命令分析，输出如下：

```

public static void tdd();
Code:
Stack=1, Locals=1, Args_size=0
0:   ldc #6; //int 65539
2:   istore_0

```

```
3:     return
```

这一次的指令终于有了很大的变化，变成了 ldc #6。所谓 ldc，全称是 load constant，意思是常量池中加载（讲真，JVM 内部大部分名称都起得很直观明了，让人能够顾名思义，即使是缩写也是如此，但 ldc 是个例外，乍一看，真猜不出啥意思）。而其后面所跟随的操作数#6，其实正是 65539 这个数字在字节码文件的常量池中的索引号。由此可见，当一个 int 型整数的宽度超过 2 字节时，Java 编译器便会将其直接编译进字节码文件的常量池中，而常量池中的数据在被 JVM 加载之后，会保存进 JVM 的常量区内。如果一个整数被保存进 JVM 的常量区之中，当其他 Java class 字节码文件中也使用了同样的整数为变量赋值时，则 JVM 不会重复将该整数写入常量区。JVM 通过这种方式避免大数据（虽然只占 4 字节）的内存重复占用。

事实上，只要为整型变量赋值的自然数超过 32767，Java 编译器便会使用 ldc 字节码指令编译源代码，而非 sipush，这是因为如果存在负数，第一位将会用于表示符号。同理，只要为整型变量赋值的自然数超过 127，Java 编译器也会生成 bipush 字节码，而非 bipush，这同样是因为第一位需要用于表示符号。看下面的示例：

清单：Test.java

功能：测试 iconst、bipush、sipush 和 ldc

```
public static void tdd(){
    int a = 5;
    int a2 = 6;

    int b = 127;
    int b2 = 128;

    int c = 32767;
    int c2 = 32768;
}
```

编译后使用 javap 命令分析，结果如下：

```
public static void tdd();
Code:
Stack=1, Locals=6, Args_size=0
0:   iconst_5
1:   istore_0
2:   bipush   6
4:   istore_1
5:   bipush   127
7:   istore_2
8:   sipush   128
11:  istore_3
12:  sipush   32767
```

```

15:  istore  4
17:  ldc #2; //int 32768
19:  istore  5
21:  return

```

注意看 int a=5 和 int a2=6 所生成的字节码指令分别是 iconst_5 和 bipush 6，这是因为 JVM 规范只提供了 iconst_0 ~ iconst_5 这 6 个特殊的字节码指令，当超过 6 时，并没有提供类似于 iconst_6 这样的字节码指令。而 int b=127 和 int b2=128 所生成的字节码也不同，分别是 bipush 127 和 sipush 128，int c=32767 和 int c2=32768 所生成的字节码也不同，分别是 sipush 32767 和 ldc #2，这是因为有一个二进制位需要用作区分正负数，因此 8 个二进制位最大只能表示到 127，16 个二进制数最大只能表示到 32768。

继续回到 ldc 这个字节码指令。既然 ldc 指令后面所跟随的内容是常量池的索引，而非真正的操作数，那么来看看该指令在 JVM 内部的格式定义，如下：

```
def(_ldc, "ldc", "bk", NULL, T_ILLEGAL, 1, true );
```

其 format=“bk”，长度是 2 位，这意味着 ldc 指令后面只能跟随一个宽度为 1 字节的操作数。

不过聪明的你可能会马上想到，既然 ldc 指令后面所跟随的操作数是常量池的索引，并且这个操作数只能占 1 字节的宽度，但是 1 字节所能代表的最大数是 256，那么如果常量池很大，其中的元素超过 256 个，那么 ldc 这个指令岂不是会存在问题了吗？为了分析问题，下面再次改造 add()方法，改造后的程序如下：

清单：/Test.java

功能：测试 ldc 指令

```

public static void tdd(){
    int v1=32769; int v2=32770; int v3=32771; int v4=32772; int v5=32773;
    int v6=32774; int v7=32775; int v8=32776; int v9=32777; int v10=32778;
    int v11=32779; int v12=32780; int v13=32781; int v14=32782; int v15=32783;
    int v16=32784; int v17=32785; int v18=32786; int v19=32787; int v20=32788;

    //这里再定义若干 int 型变量，其初始值都超过 32767，并且各个变量的初始值彼此不同
    //...

    int eeeef11=46779; int eeeef12=46780; int eeeef13=46781; int eeeef14=46782;
    int eeeef15=46783;
    int eeeef16=46784; int eeeef17=46785; int eeeef18=46786; int eeeef19=46787;
    int eeeef20=46788;

    //一共定义超过 256 个局部整型变量
}

```

由于 Test.tdd()方法内部定义了超过 256 个整型变量，值都大于 32767，并且彼此不同，因

此每一个给变量赋值的自然数都会在 Test 类所生成的字节码文件的常量池中的元素数组中占有
一席之地，并且部分常量池元素的索引号会大于 256，这超过 1 字节所能表示的最大范围。且
看这种情况下 ldc 字节码指令如何处理索引号大于 256 的元素加载。编译 Test 类，并使用 javap
命令分析编译后的字节码文件，输出如下：

```
$ javap -v Test
Compiled from "Test.java"
class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method    #302.#310;    //  java/lang/Object."<init>":()V
const #2 = int 32769;
const #3 = int 32770;
const #4 = int 32771;
const #5 = int 32772;
//...此处省略若干常量池元素
const #253 = int 44781;
const #254 = int 44782;
const #255 = int 44783;
const #256 = int 44784;
const #257 = int 44785;
const #258 = int 44786;
const #259 = int 44787;
//...省略若干常量池内容

{
test1();
  Code:
  Stack=1, Locals=1, Args_size=1
  0:   aload_0
  1:   invokespecial #1; //Method java/lang/Object."<init>":()V
  4:   return
LineNumberTable:
  line 18: 0

public static void tdd();
  Code:
  Stack=1, Locals=300, Args_size=0
  0:   ldc #2; //int 32769
  2:   istore_0
  3:   ldc #3; //int 32770
  5:   istore_1
  6:   ldc #4; //int 32771
```

```

8:    istore_2
9:    ldc #5; //int 32772
11:   istore_3
//...省略若干字节码指令

1004: ldc #253; //int 44781
1006: istore 252
1008: ldc #254; //int 44782
1010: istore 253
1012: ldc #255; //int 44783
1014: istore 254
1016: ldc_w #256; //int 44784
1019: istore 255
1021: ldc_w #257; //int 44785
1024: istore_w 256
1028: ldc_w #258; //int 44786
1031: istore_w 257
//...省略若干字节码指令

```

从 javap 命令的输出结果可以看到，常量池中的元素索引号已经超过 256。观察 tdd()方法的字节码指令，可以看到当使用自然数 44783 为 tdd()方法内的局部变量赋值时所使用的字节码指令是 ldc #255，而当使用自然数 44784 为 tdd()方法内的局部变量赋值时所使用的字节码指令就变成了 ldc_w，而且此后的局部变量的赋值指令都变成了 ldc_w。显然，当从常量池中索引号大于 255 的地方取值时，JVM 使用了 ldc_w 指令。该指令的含义是：将 int、float 或 String 型常量值从常量池中推送至栈顶（宽索引）。看看 bytecodes.cpp 中如何定义该指令格式：

```
def(_ldc_w, "ldc_w", "bkk", NULL, T_ILLEGAL, 1, true );
```

其 format=“bkk”，总长度变成了 3，说明 ldc_w 指令后面可以跟随一个 2 字节宽的操作数。如此一来就通了，对于将常量池中索引号大于 255 的常量池推送至栈顶，JVM 就使用 ldc_w 指令，反之就使用 ldc 指令。道理其实也很简单，这是在尽量压缩字节码文件的体积。使用十六进制编辑器打开上面更改过的 Test 类的字节码文件，可以查找到 ldc #255 和 ldc_w #256 这 2 条字节码指令的内容，如图 9.2 所示。

169	12f8 36f7 12f9 36f8 12fa 36f9 12fb 36fa
170	12fc 36fb 12fd 36fc 12fe 36fd 12ff 36fe
171	1301 0036 ff13 0101 c436 0100 1301 02c4
172	3601 0113 0103 c436 0102 1301 04c4 3601
173	0313 0105 c436 0104 1301 06c4 3601 0513
174	0107 c436 0106 1301 08c4 3601 0713 0109
175	c436 0108 1301 0ac4 3601 0913 010b c436

图 9.2 观察 ldc #255 和 ldc_w #256 指令在字节码文件中的内容

由图 9.2 可以看到，`ldc #255` 指令在字节码文件中的内容是 `0x12ff`，这是因为 `ldc` 指令的十六进制编码是 `0x12`，而 `255` 所对应的十六进制数是 `0xff`。同理，`ldc_w` 指令所对应的十六进制编码是 `0x13`，`256` 所对应的十六进制数是 `0x0100`，所以 `ldc_w #256` 指令的十六进制内容是 `0x130100`。所以在字节码文件中，`ldc` 整条指令，操作码连同操作数，一共只占 2 字节；而 `ldc_w`，与 `bytecodes.cpp` 中所定义的 `format` 一致，一共只占 3 字节。由此可见，JVM 为了尽可能地减小字节码文件的体积，真所谓无所不用其极，玄之又玄，众妙之门！而这种努力是非常值得的，随便一个 Java Web 程序的 war 包中都包含成千上万个 Java class 字节码文件，字节码文件的体积得以减少，则在网络传输（例如，远程部署）时将提升速度，并且 JVM 将它们读进内存后所占内存空间也会减少。虽然一个字节码文件的体积的减少量并不是很明显，但是宏观上的效应就很可观了。

纵观 `bytecodes.cpp::initialize()` 函数中所定义的全部字节码指令的格式，会看到绝大多数字节码指令的 `format` 所包含的字符数都是 1 位，少部分为 2 位和 3 位，而超过 3 位的则更少，寥寥无几。因此 JVM 的字节码指令集在设计上非常紧凑和简洁。有兴趣的道友可以继续研究对于 `long` 型和 `double` 型的变量赋值所对应的字节码指令，并分析指令在字节码文件中所占用空间的大小。

上面讲了这么多，除了可以知道对于 `int` 型变量的自然数赋值指令包含 `iconst`、`bipush`、`sipush`、`ldc` 和 `ldc_w` 这五种外，最主要的是明白了 JVM 内部对字节码的指令宽度是有严格定义的，而字节码指令的宽度对于 JVM 虚拟机完成取指是至关重要的，JVM 只有知道了每一个字节码指令所占的宽度，才能完成“取指→译码→执行→取指”这种循环，将程序一直执行下去。

`Bytecodes::initialize()` 函数会在 JVM 启动期间被调用，该函数执行完成之后，各个字节码指令所占的内存宽度便会被 JVM 所记录，JVM 在运行期执行 Java 程序时会不断地读取该函数所维护的表，计算每个字节码指令的长度。

9.2.2 JVM 的两级取指机制

前面分析了执行引擎取指的关键一步——计算每一个指令的总长度。无论是物理 CPU，还是 JVM 的软件模拟的执行引擎，其内在的核心机制都是类似的。在 HotSpot 内部也存在与 CPU 内部类似的译码器，HotSpot 里面通常叫作“解释器”。HotSpot 提供了好几种解释器，例如字节码解释器 `bytecodeInterpreter`、模板解释器 `templateInterpreter` 等。如果 HotSpot 以模板解释器来执行字节码指令（事实上这也是默认的方式），则所有的字节码指令都会通过 `TemplateInterpreterGenerator::generate_and_dispatch()` 这个函数来生成对应的机器指令。在该函数中实现了指令跳转（即取下一条字节码指令）的逻辑。该函数的实现如下：

清单：/src/share/vm/interpreter/templateInterpreter.cpp

功能：generate_and_dispatch()函数中的取指逻辑

```
void TemplateInterpreterGenerator::generate_and_dispatch(Template* t,
TosState tos_out) {
    int step;
    if (!t->does_dispatch()) {
        step = t->is_wide() ? Bytecodes::wide_length_for(t->bytecode()) :
Bytecodes::length_for(t->bytecode());
        if (tos_out == ilgl) tos_out = t->tos_out();

        __ dispatch_prolog(tos_out, step); //该函数暂时无实现
    }

    // generate template
    t->generate(_masm);

    // advance--取下一条指令
    if (t->does_dispatch()) {
#ifndef ASSERT
        // make sure execution doesn't go beyond this point if code is broken
        __ should_not_reach_here();
#endif // ASSERT
    } else {
        // dispatch to next bytecode
        __ dispatch_epilog(tos_out, step); //取下一条字节码指令
    }
}
```

该函数会在 JVM 启动期间被调用，用于生成固定的取指逻辑。需要注意的是，JVM 会为每一个字节码指令都生成一个特定的取指逻辑，这是因为不同的字节码其指令宽度不同，因此取指逻辑也肯定不统一。该函数其实主要干了两件事：

(1) 为 Java 字节码指令生成对应的汇编指令。

(2) 实现字节码指令跳转，即“取指”。

这两件事对于 templateInterpreter 模板解释器而言，是核心中的核心。通过该函数也可以知道，HotSpot 内部在生成“取指”逻辑的同时，也会为字节码指令生成对应的本地机器码。或者换而言之，HotSpot 在为每一个字节码指令生成其机器逻辑指令时，会同时为该字节码指令生成其取指逻辑（取其下一条指令）。该函数的第一个入参是 Template*类型的指针，Template 便是解释器为每个 Java 字节码指令所定义的汇编模板，在本函数中通过调用 t->generate(_masm)来生成字节码指令的机器码，该逻辑会在下文中详细讲解，这里重点关注 TemplateInterpreter Generator::generate_and_dispatch()函数中所调用的 __dispatch_epilog(tos_out, step)这行代码。这行

代码便是在生成跳转(取指)逻辑。该函数的第二个入参是 step, step 是 Java 字节码指令的“步长”或所占的数据宽度, 其单位是“字节”或 8 位。在 TemplateInterpreterGenerator::generate_and_dispatch() 函数中, 通过 step = t->is_wide() ? Bytecodes::wide_length_for(t->bytecode()): Bytecodes::length_for(t->bytecode()) 来计算出字节码指令的步长, 而计算的逻辑正与上文所描述的一致, 即根据 Bytecodes::initialize() 函数中为每个字节码指令所定义的 format 字段来计算, format 包含几个字符, 则字节码的步长便是几。有兴趣的道友可以跟进 Bytecodes::length_for() 函数去看下具体的实现逻辑。

在 32 位 x86 平台上, __ dispatch_epilog(tos_out, step) 函数的实现逻辑如下(由于其内部涉及汇编, 因此一定是 CPU 平台相关的):

清单: /src/cpu/x86/vm/interp_masm_x86_32.cpp

功能: 取指逻辑演示

```
void InterpreterMacroAssembler::dispatch_epilog(TosState state, int step) {
    dispatch_next(state, step);
}

void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
    // 加载下一个字节码指令
    load_unsigned_byte(rbx, Address(rsi, step));
    // advance rsi
    increment(rsi, step);
    dispatch_base(state, Interpreter::dispatch_table(state));
}
```

在 InterpreterMacroAssembler::dispatch_epilog() 函数中调用 dispatch_next() 函数, 而后者则通过“三部曲”完成两级取指逻辑。所谓两级取指逻辑, 第一级是获取字节码指令, 当前字节码指令执行完成后, JVM 必须能够自动获取到其下一条字节码指令, 这样才能循环往复地执行下去。而第二级取指逻辑则是取字节码指令所对应的本地机器指令, 当前字节码指令对应的机器码执行完成之后, JVM 必须要能够跳转到下一条字节码指令所对应的机器码。

InterpreterMacroAssembler::InterpreterMacroAssembler() 函数中的这三部曲分别是:

```
load_unsigned_byte(rbx, Address(rsi, step));
increment(rsi, step);
dispatch_base(state, Interpreter::dispatch_table(state));
```

这三条指令在不同的 CPU 平台上会生成不同的取指机器指令, 在 32 位 x86 平台上生成如下 3 条机器指令:

```
movzbl 0x1(%esi),%ebx
inc %esi
jmp *_dispatch_table(%ebx,state)
```

这 3 条机器指令中的第一条和第三条用于对本地机器码取指和跳转，其逻辑先不必理会，第二条 inc %esi 则用于取字节码指令，该指令由 InterpreterMacroAssembler::InterpreterMacroAssembler() 函数中的 increment(rsi, step) 代码生成，increment(rsi, step) 函数便是 Jvm 模板解释器用于取指的核心逻辑，该函数的功能是计算下一个即将执行的 Java 字节码指令的内存位置，而计算公式很简单：

下一个字节码指令的内存位置 = 当前字节码的位置 + 当前字节码指令所占的内存大小（以字节计）

这种公式很好理解，对于一个 Java 方法，当 JVM 将其加载进内存后，会将该 Java 方法所对应的全部字节码指令存放到一块内存区域中，这些字节码指令彼此相邻，在内存里线性按序存储，所以相邻的 2 条字节码指令所对应的内存地址的偏移量便是前面一条字节码指令所占的内存大小。事实上物理机器指令的存储方式也是这样的，因此物理 CPU 在取指时也遵循同样的逻辑。该函数就像汽车中的发动机曲轴，通过它，JVM 才能沿着人们所编写好的 Java 逻辑一直往下运行下去。而在物理机器层面，CPU 也具有取指功能，只不过 CPU 的取指功能是实实在在的硬件电路实现，而 JVM 仅仅是软件模拟。

increment(rsi, step) 函数的第一个入参是 rsi 寄存器，该寄存器“总是”指向当前字节码指令所在的内存位置，第 2 个入参是 step，step 是 Java 字节码指令的步长（即字节码指令所占的内存大小）。increment(rsi, step) 函数最终生成的机器指令类似于 $rsi = rsi + step$ ，表示将当前字节码指令所在的内存位置加上字节码指令的步长，从而得到当前字节码指令的下一条字节码指令的内存位置，这便完成了 JVM 取指的逻辑。step 的计算方式前文讲过，在模板表中通过 format 这个字段指定每个字节码指令的步长，并在 TemplateInterpreterGenerator::generate_and_dispatch() 函数中实现步长计算的逻辑，逻辑很简单，就是获取 format 字符串的字符数。例如对于 iload_0 指令，该字节码指令的 format=“b”，其步长便是 1 字节。

由于不同的 Java 字节码指令的步长是不同的，因此最终所生成的本地机器码也有所不同，如果字节码指令的步长是 1（只有操作码而没有操作数），则生成的本地机器指令便是 inc %esi，该指令表示对 esi 增加 1 字节（因为一个步长为 1 的 Java 字节码指令所占的内存空间为 1 字节）；而如果字节码指令的步长超过 1，则生成的本地机器指令便是 add \$operand, %esi，该指令表示对 esi 进行累加；假设字节码的步长为 2，则对应的指令是 add \$0x2, %esi，表示对 esi 寄存器增加 2 字节。例如，对于 iload_1 这样的字节码指令，该指令没有操作数，因此步长为 1，那么 iload_1 字节码的下一个字节码的内存位置相对于 iload_1 偏移量是 1 字节，所以生成的机器指令是 inc %esi，其将 esi 寄存器（该寄存器指向当前字节码指令的内存位置）的值加 1，便得到 iload_1 这条指令的下一条指令所在的内存位置。

可能聪明的你会想到这样一个问题：rsi 寄存器总是指向当前字节码指令所在的内存位置，

这一句恐怕不对吧，例如，当 JVM 在运行 Java 程序 main()主函数所对应的第一条字节码指令时，这个时候根本就不存在“上一条”字节码指令，那么 rsi 寄存器指向哪里呢？

事实上，在 JVM 调用 Java 的 main()主函数时，会先调用 generate_fixed_frame(false)函数来创建栈帧，而在这个过程里面，JVM 便会有一个逻辑获取 Java 的 main()主函数在 JVM 内部的第一条字节码指令，并将 esi 寄存器指向第一条字节码的位置。这个逻辑的具体实现在前面讲解 Java 函数栈帧的章节里详细讲解过，此处不再赘述。同时，关于 Java 方法在 JVM 内部的映射以及 Java 方法的字节码指令在 JVM 内部的存储，也在前面讲解 Java 方法的章节里详细讲解过，还不清楚的小伙伴可以去相关章节进行研究。

正因为 JVM 调用 Java 的 main()主函数之前会先执行 generate_fixed_frame(false)来创建栈帧，并在这个过程中将 esi 寄存器的值指向 main()函数的第一条字节码指令，所以接着调用 dispatch_next()函数时才能根据 rsi 进行偏移，顺利完成取指。

到了这里，终于可以与本章开始处所讲的 InterpreterGenerator::generate_normal_entry()函数接上头了。本章一开始便讲到，在 JVM 进入 Java 世界之前，会先找到 Java 的 main()主函数并调用，而调用 Java 主函数时，最终流程会进入 InterpreterGenerator::generate_normal_entry()这个函数的逻辑中去。而在 InterpreterGenerator::generate_normal_entry()函数中，除了会调用 generate_fixed_frame()函数为 Java 的 main()主函数创建栈帧之外（注意，在创建栈帧的过程中，JVM 会将 rsi 寄存器指向 main()主函数的第一条字节码指令的内存地址），还将调用 dispatch_next()函数执行 Java 的 main()主函数的第一条字节码指令。在 32 位 x86 平台上，在 InterpreterGenerator::generate_normal_entry()函数中所调用的 dispatch_next()函数便是定义在 /src/cpu/x86/vm/interp_masm_x86_32.cpp 中的 dispatch_next()这个函数。

当 JVM 第一次调用 Java 的 main()主函数时，rsi 指向 Java 的 main()主函数的第一条字节码指令在内存中的位置，但是从 InterpreterGenerator::generate_normal_entry()函数中调用 dispatch_next()函数时，仅传入了第一个参数 tosState，而第二个参数 step 并没有传递，因此 step 默认为 0，而当步长为 0 时，JVM 最终不会生成类似 inc %esi 这样的机器指令，即此时不会“取下一条指令”，因为 JVM 总得先执行第一条指令。只有第一条字节码指令执行完成之后，JVM 才能通过 esi 来获取下一条字节码指令所在的内存位置，从而挨个执行 Java 方法的全部字节码指令。

刚才讲过，JVM 内部其实存在两级取指逻辑，第一级取字节码指令，第二级取字节码指令对应的本地机器码。第二级的取指逻辑是通过 InterpreterMacroAssembler::InterpreterMacroAssembler()函数中的第一和第三条指令完成的，这两条指令最终所生成的本地机器指令如下（32 位 x86 平台）：

```

movzbl step(%esi),%ebx
jmp    *_dispatch_table(%ebx,state)

```

第一条指令 movzbl step(%esi),%ebx 取下一条字节码指令，并将其存储到 ebx 寄存器中，接着通过第二条指令跳转到下一条字节码指令所对应的本地机器码。注意，第二条指令中的 _dispatch_table 便是 JVM 内部所维护的跳转表，跳转表中记录了每个 JVM 字节码所对应的本地机器码实现，JVM 通过跳转表，完成第二级取指逻辑。在 JVM 取到某条字节码指令时，跳转到其对应的本地机器码并执行。

9.2.3 取指指令放在哪

前面讲过，如果 HotSpot 以模板解释器来执行字节码指令（事实上这也是默认的方式），则所有的字节码指令都会通过 TemplateInterpreterGenerator::generate_and_dispatch()这个函数生成对应的机器指令，TemplateInterpreterGenerator::generate_and_dispatch()函数主要完成两件事：

- (1) 为当前字节码指令生成其对应的本地机器码。
- (2) 为当前字节码指令生成其对应的取指逻辑（取下一条指令）。

这两件事分别通过 t->generate(_masm) 和 __ dispatch_epilog(tos_out, step) 来完成，TemplateInterpreterGenerator:: generate_and_dispatch()函数先调用 t->generate(_masm)为当前字节码指令生成其对应的本地机器码，具体生成逻辑在后文中讲解，而 __ dispatch_epilog(tos_out, step) 的逻辑刚刚讲过，通过三部曲进行两级取指。t->generate(_masm)和 __ dispatch_epilog(tos_out, step) 这两个函数会分别向 JVM 内部的代码缓冲区中写入对应的本地机器指令，由此可知，字节码的取指逻辑其实是被写入到每一个字节码指令所对应的本地机器码所在内存的后面区域。其原因很简单，因为不同的字节码指令的步长不同，因此所生成的对 rsi 寄存器进行累加的逻辑也不同，所以不同字节码指令的取指逻辑肯定不同。事实上，这还与栈顶缓存有关。

使用 HSDIS 工具可以查看 JVM 模板解释器在运行期所生成的全部字节码指令的本地机器码。例如 bipush 的本地机器指令如图 9.3 所示。

```
bipush 16 bipush [0xb36d80a0, 0xb36d80e0] 64 bytes
[Disassembling for mach='i386']
0xb36d80a0: sub    $0x4,%esp
0xb36d80a3: fstps (%esp)
0xb36d80a6: jmp    0xb36d80c4
0xb36d80ab: sub    $0x8,%esp
0xb36d80ae: fstpl (%esp)
0xb36d80b1: jmp    0xb36d80c4
0xb36d80b6: push   %edx
0xb36d80b7: push   %eax
0xb36d80b8: jmp    0xb36d80c4
0xb36d80bd: push   %eax
0xb36d80be: jmp    0xb36d80c4
0xb36d80c3: push   %eax
0xb36d80c4: movsbl 0x1(%esi),%eax
0xb36d80c8: movzbl 0x2(%esi),%ebx
0xb36d80cc: add    $0x2,%esi
0xb36d80cf: impx  *-0x48f106a0(.%ebx.4)
0xb36d80d6: xchg   %ax,%ax
0xb36d80d8: add    %al,(%eax)
0xb36d80da: add    %al,(%eax)
0xb36d80dc: add    %al,(%eax)
0xb36d80de: add    %al,(%eax)
```

图 9.3 bipush 字节码指令的取指指令

图 9.3 中所框住的部分便是 bipush 字节码的取指机器码，而所框住部分之前的逻辑，是 bipush 字节码指令本身的逻辑。再如 iadd 的本地机器指令如图 9.4 所示。

```
iadd 96 iadd [0xb36d9f40, 0xb36d9f60] 32 bytes
[Disassembling for mach='i386']
0xb36d9f40: pop    %eax
0xb36d9f41: pop    %edx
0xb36d9f42: add    %edx,%eax
0xb36d9f44: movzbl 0x1(%esi),%ebx
0xb36d9f48: inc    %esi
0xb36d9f49: impx  *-0x48f106a0(.%ebx.4)
0xb36d9f50: add    %al,(%eax)
0xb36d9f52: add    %al,(%eax)
0xb36d9f54: add    %al,(%eax)
0xb36d9f56: add    %al,(%eax)
0xb36d9f58: add    %al,(%eax)
0xb36d9f5a: add    %al,(%eax)
0xb36d9f5c: add    %al,(%eax)
0xb36d9f5e: add    %al,(%eax)
```

图 9.4 iadd 字节码指令的取指指令

同样可以看到，iadd 指令本身的逻辑在前面，而取指逻辑则在后面。

其实，JVM 虚拟机的这种取指逻辑安排与硬件 CPU 的取指逻辑是完全一致的，只不过 CPU 是通过数字电路来完成自动取指功能。当 CPU 读取到当前机器指令时，CPU 内部的数字电路便会触发取指电路去工作，取指电路会根据当前所取指令的内存地址偏移当前指令的长度，从而计算出下一条指令的内存位置。通过这种机制物理 CPU 便能够周而复始地一直执行下去，直到最后一条指令。

9.2.4 程序计数器在哪里

经常可以在各种书籍上看到 JVM 在执行字节码指令时，会有一个 pc 计数器(program counter)指向当前所执行的指令，当前指令执行完成之后，PC 会自动指向下一条字节码指令，程序计数器是保证软件程序能够连续执行下去的关键技术之一。物理 CPU 中专门有一个寄存器用于存放 PC，当计算机上的某个软件程序开始运行之前，操作系统会将该软件程序加载至内存中（数据段、代码段），加载之后，操作系统便会执行一件非常重要的事情——将该软件程序的第一条机器指令在内存中的地址送入程序计数器，操作系统会从该地址读取指令，并开始执行，由此开始操作系统将 CPU 的控制权交给软件程序。当执行指令时，处理器将自动修改 PC 的内容，每执行一条指令，PC 便会增加一个量，这个量等于指令所含的字节数，这样 PC 所指向的内存位置总是将要执行的下一条指令的地址。由于大多数指令都是按顺序来执行的，所以 PC 通常都是加 1。

JVM 内部的程序计数器的原理也与之相同，其实经过前面对 JVM 取指技术实现的分析可知，JVM 内部的所谓 PC 计数器，其实就是 esi 寄存器 (x86 平台)。当 JVM 开始执行 Java 程序的 main() 主函数时，PC (esi 寄存器) 便会指向 Java 程序的 main() 主函数的第一条字节码指令的内存位置，接着 JVM 每执行完一条字节码指令便会对 PC 执行一定的增量，从而让 PC 总是指向即将要执行的字节码指令，如此便能让 Java 程序连续执行下去。

知其然，并知其所以然，方为大善！可能聪明的你会问：JVM 为何要使用宝贵的寄存器资源作为程序计数器呢？道理很简单，就是因为 CPU 读写寄存器的速度是最快的（相对于内存和磁盘）。由于 JVM 一旦跑起来之后，所做的事情就是不断地执行“取指→译码→执行”这一循环往复的任务，因此取指在 JVM 内部可谓是最频繁的事情，如此多的数据读写，如果性能低下，必然影响到 JVM 的整体执行效率，因此 JVM 便选择一个寄存器作为程序计数器。另一方面，JVM 的指令集是面向栈的，而面向栈的指令集并不直接依赖于寄存器，因此 JVM 也有更多的资源可以直接基于寄存器。后文会通过一个 Java 程序示例来演示取指的过程，以及随着 Java 程序的执行，程序计数器的变化过程。

9.3 译码

前面详细讲解了 JVM 的取指逻辑的实现机制，对于执行引擎而言，取指只是第一步，执行才是终极目标。不过从取指到执行中间，还有一个步骤：译码。

之所以要译码，道理很简单，JVM 内部定义了两百多个字节码指令，不同字节码指令的实现机制都是不同的，因此 JVM 取出字节码指令后，需要将其翻译成不同的逻辑，然后才能执行。

9.3.1 模板表

对于物理 CPU 而言，译码逻辑直接固化在硬件数字电路中，当 CPU 读取到特定的物理机器指令时，会触发所固化的特定数字电路，这种触发机制其实便是译码逻辑。如果 CPU 对一条机器指令无动于衷，那么 CPU 便无法完成译码，更无法执行指令，严重的则直接宕机。JVM 是虚拟的机器，没有专门的硬件译码电路，因此仅能软件模拟。如果 JVM 以模板解释器来解释字节码，则这种模板定义如下所示：

清单：/src/share/vm/interpreter/templateTable.cpp

功能：JVM 指令模板定义

```
void TemplateTable::initialize() {
    // ...
    // Java spec bytecodes      ubcp|disp|clvm|iswd in   out generator      argument
    def(Bytecodes::_nop      , ____|____|____|____, vtos, vtos, nop      , _      );
    def(Bytecodes::_const_null, ____|____|____|____, vtos, atos, aconst_null , _      );
    def(Bytecodes::_const_m1  , ____|____|____|____, vtos, itos, iconst   , -1     );
    def(Bytecodes::_const_0   , ____|____|____|____, vtos, itos, iconst   , 0      );
    def(Bytecodes::_const_1   , ____|____|____|____, vtos, itos, iconst   , 1      );
    def(Bytecodes::_const_2   , ____|____|____|____, vtos, itos, iconst   , 2      );
    def(Bytecodes::_fload_1   , ____|____|____|____, vtos, ftos, float    , 1      );
    def(Bytecodes::_fload_2   , ____|____|____|____, vtos, ftos, float    , 2      );
    // ...
}
```

在该函数中，通过 def() 函数对每一个字节码指令进行定义，定义了什么呢？def() 函数一共有 9 个人参，第 1 个人参是字节码指令编码，而第 8 个人参则是该指令所对应的汇编指令生成器，这种生成器在 JVM 内部被称作 generator。其实这很好理解，因为对于模板解释器而言，每一个 Java 字节码指令最终都会生成对应的一串本地机器码，JVM 在运行期将直接执行这些机器码。因此，对于模板解释器，JVM 为每一个字节码都专门配备了一个生成器。其实所谓生成器，说白了就是一个函数而已。

例如，对于将 int 型局部变量从局部变量表推送至操作数栈栈顶，JVM 中一共设计了多种 iload 字节码指令系列，例如 iload_0、iload_1 等，而这些字节码指令所对应的 generator 都是 iload() 函数。iload() 函数是 CPU 平台架构相关的，并且在 templateTable.hpp 中定义了 2 个重载函数，如下：

```
static void iload();
static void iload(int n);
```

如果字节码指令是 iload_6、iload_18 之类的，其对应的 generator 生成器便是 iload()。如果字节码指令是 iload_0、iload_1 之类的，则对应的 generator 生成器是 iload(int n)。与将自然数推送至操作数栈栈顶的机制一样，从局部变量表加载数据到操作数栈栈顶，也并非都使用同一个字节码指令。对于 int 型局部变量，当其 slot 索引号小于等于 3 时，使用 iload_0、iload_1、iload_2 或者 iload_3 这样的字节码指令。而当 slot 索引号超过 3 时，便会使用 iload slot_idx 这样的字节码指令。iload_0、iload_1、iload_2 或者 iload_3 这样的字节码指令只占 1 字节内存空间，因为没有操作数，而 iload slot_idx 这样的指令会占用 2 字节空间，操作码和操作数各占 1 字节，由此可知 JVM 中之所以设计 iload_0、iload_1、iload_2 或者 iload_3 这样的字节码指令，其目的仍然是为了给字节码文件瘦身。iload_0、iload_1、iload_2 或者 iload_3 字节码指令所对应的 generator 是 TemplateTable::iload(int n) 函数，其定义如下（在 32 位 x86 平台）：

清单：/src/cpu/x86/vm/templateTable_x86_32.cpp

功能：演示 x86 平台上的 iload 字节码指令的机器码生成函数

```
void TemplateTable::iload(int n) {
    transition(vtos, itos);
    __ movl(rax, iaddress(n));
}
```

在该函数中，通过 __ movl(rax, iaddress(n)) 这条指令将 Java 方法栈帧的局部变量表中指定索引号的变量传送至操作数栈栈顶。关于该函数的实现机制，后文会进行分析。在这里需要关心的一个问题是：对于在 TemplateTable::initialize() 函数中通过 def() 函数所定义的各种字节码指令的模板，JVM 是如何进行保存的，以便在运行期能够据此进行模板翻译？

这个秘密就藏在 def() 函数中。在 TemplateTable::initialize() 函数中通过 def() 函数定义了每一个字节码指令的机器码生成器，def 函数实现如下：

清单：/src/share/vm/interpreter/templateTable.cpp

功能：JVM 指令模板定义函数

```
void TemplateTable::def(Bytecodes::Code code, int flags, TosState in, TosState
out, void (*gen)(int arg), int arg) {
    // should factor out these constants
    const int ubcp = 1 << Template::uses_bcp_bit;
```

```

    const int disp = 1 << Template::does_dispatch_bit;
    const int clvm = 1 << Template::calls_vm_bit;
    const int iswd = 1 << Template::wide_bit;
    // determine which table to use
    bool is_wide = (flags & iswd) != 0;
    Template* t = is_wide ? template_for_wide(code) : template_for(code);
    t->initialize(flags, in, out, gen, arg);
}

```

在该函数中，先通过 `Template* t = is_wide ? template_for_wide(code) : template_for(code)` 从模板表中取出当前定义的字节码指令模板，接着通过 `t->initialize(flags, in, out, gen, arg)` 对字节码指令模板进行初始化，如此便将初始化好的字节码指令模板保存到模板表中。

模板表是什么？其实在 `TemplateTable::initialize()` 函数中定义的字节码指令及其生成器便保存在模板表中，其定义如下：

清单：/src/share/vm/interpreter/templateTable.hpp

功能：JVM 指令模板表

```

class TemplateTable: AllStatic {
public:
    // ...

private:
    static bool _is_initialized; // true if TemplateTable has been initialized
    static Template _template_table [Bytecodes::number_of_codes];
    static Template _template_table_wide[Bytecodes::number_of_codes];

    static Template* template_for (Bytecodes::Code code) { Bytecodes::check
(code); return &_template_table [code]; }
    static Template* template_for_wide(Bytecodes::Code code)
{ Bytecodes::wide_check(code); return &_template_table_wide[code]; }
    // ...
}

```

在 `TemplateTable` 类中定义了 `_template_table` 数组，该数组即是模板表，模板表记录了每个字节码指令的汇编生成器（即函数）、参数及其他相关信息。该数组的元素类型是 `Template`，而数组的大小初始化为 `Bytecodes::number_of_codes`，这就是 Java 字节码指令的数量，因此 `_template_table` 数组的初始化大小就是 Java 字节码指令的数量，道理很简单，因为每一个 Java 字节码指令对应一个模板。同时 `TemplateTable` 类中定义了模板表的访问接口 `template_for(Bytecodes::Code)` 函数，其可以根据字节码指令的编号查询对应的模板，该接口在 `TemplateTable::def()` 函数中被调用，用于读取字节码指令在模板表中所对应的模板。注意，无论是模板表 `_template_table` 还是模板表的访问接口函数，都是静态的（使用 `static` 修饰），因此在

操作系统加载 `TemplateTable` 类时便会完成模板表的初始化，只不过这个时候的模板表里的元素都是空值，模板尚未构建。因此，在 `TemplateTable::def()` 函数中才需要先从模板表中取出当前模板，并进行初始化。如此一来，等到 `TemplateTable::initialize()` 函数执行完成，则字节码指令的模板表也完成构建。

那么模板在啥时候构建呢？或者换言之，`TemplateTable::initialize()` 函数在啥时候被调用呢？其实可以很容易猜到，模板表是运行期执行 Java 字节码指令时时刻都会使用到的基础数据，因此一定在 JVM 启动期间进行构建。模板表构建的整体链路如下：

- ①. `java.c: main()`
 调用 `LoadJavaVM()`
- ②. `java_md.c: LoadJavaVM()`
 调用 `ifn->CreateJavaVM = (CreateJavaVM_t)dlsym(libjvm, "JNI_CreateJavaVM")`
- ③. `jni.cpp: _JNICALL JNI_CALL JNI_CreateJavaVM()`
 调用 `result = Threads::create_vm((JavaVMInitArgs*) args, &can_try_again)`
- ④. `thread.cpp: Thread::create_vm()`
 调用 `init_globals()`
- ⑤. `init.cpp: init_globals()`
 调用 `interpreter_init()`
- ⑥. `interpreter.cpp: interpreter_init()`
 调用 `Interpreter::initialize()`
- ⑦. `templateInterpreter.cpp: TemplateInterpreter::initialize()`
- ⑧. `templateTable.cpp: TemplateTable::initialize()`
 初始化模板表

这条链路是本地调试 HotSpot 源码时的调用链路，本地调试编译好的调试版 HotSpot 时会从 `/src/share/tools/launcher/java.c` 这个 `main()` 主函数所在的文件进入，然后一路调用下来。HotSpot 启动时进入 `java.c::main()` 主函数，`main()` 主函数由操作系统调用。接着在 `LoadJavaVM()` 中调用 `JNI_CreateJavaVM()` 接口开始创建 JVM 虚拟机，需要注意的是，在 `java_md.c:: LoadJavaVM()` 过程中有两处都会调用 `JNI_CreateJavaVM()` 接口，逻辑如下（POSIX 系统平台）：

清单: `/src/os posix/launcher/java_md.c`

功能: 创建虚拟机接口

```
jbboolean LoadJavaVM(const char *jvmpath, InvocationFunctions *ifn)
{
    #ifdef GAMMA
    /* JVM is directly linked with gamma launcher; no dlopen() */
    ifn->CreateJavaVM = JNI_CreateJavaVM;
    ifn->GetDefaultJavaVMInitArgs = JNI_GetDefaultJavaVMInitArgs;
    return JNI_TRUE;
    #else
    // ...
    #endif
}
```

```

ifn->CreateJavaVM = (CreateJavaVM_t) dlsym(libjvm, "JNI_CreateJavaVM");
// ...
#endif /* ifndef GAMMA */
}

```

在该接口中，通过判断是否定义 GAMMA 宏，分别使用两种方式调用 JNI_CreateJavaVM()，如果设置过 GAMMA 宏则直接调用该接口，否则便通过动态链接库进行调用。当在本地编译 HotSpot 源代码并生成调试版的 HotSpot 时，为了调试跟踪方便，HotSpot 提供了 gamma 启动器，通过 gamma 启动器能够直接在本地调试 HotSpot 源代码。所谓 gamma 启动器，其实就是上面链路中的入口/src/share/tools/launcher/java.c::main()函数。而对于 Java 用户而言，启动 Java 程序时所调用的命令是%JAVA_HOME%\bin\java，Java 脚本也是使用 C 语言编写的，其逻辑与 gamma 启动器的逻辑基本一样，但是这两者都共用了同一套 launcher 源代码，launcher 就是上述链路中的 java_md.c，因此需要在 java_md.c 中区分 HotSpot 是否从 gamma 启动器启动。如果不是从 gamma 启动，便是从%JAVA_HOME%\bin\java 命令中启动 HotSpot 虚拟机，如果通过 Java 命令启动虚拟机，则本地一定需要先行安装 JDK，否则 Java 程序无法启动。安装 JDK 之后，在 Linux 上会生成 libjvm.so 动态链接库，在 Windows 上会生成 jvm.dll，因此如果是从 Java 命令启动 HotSpot 虚拟机，则需要在 launcher 中加载动态链接库，从而进入虚拟机。

不管是从 gamma 启动器启动虚拟机还是从 Java 命令启动虚拟机，最终都会调用 jni.cpp::JNI_IMPORT_OR_EXPORT_jint JNICALL JNI_CreateJavaVM()接口，在该接口中会调用 Threads::create_vm()接口，该接口会执行一系列的虚拟机初始化逻辑，而 JVM 是基于解释的虚拟机，因此最终会调用 Interpreter::initialize()接口对解释器进行初始化。

在 interpreter.cpp::interpreter_init()中调用 Interpreter::initialize()时，会根据系统所设置的参数而调用对应的解释器。看 Interpreter 类的声明：

清单：/src/share/vm/interpreter/interpreter.hpp

功能：Interpreter 解释器类声明

```

class Interpreter: public CC_INTERP_ONLY(CppInterpreter) NOT_CC_INTERP
(TemplateInterpreter) {
    // ...
}

```

Interpreter 类继承时通过 CC_INTERP_ONLY(CppInterpreter) 宏和 NOT_CC_INTERP(TemplateInterpreter)宏来判断继承哪一个解释器，在 HotSpot 中，默认使用模板解释器，因此 Interpreter 类实际继承的是 TemplateInterpreter 这个模板解释器。在 interpreter.cpp::interpreter_init()函数中调用 Interpreter::initialize()函数时，实际调用的是 TemplateInterpreter::initialize()这个函数，该函数声明如下：

清单：/src/share/vm/interpreter/templateInterpreter.cpp

功能：模板解释器的初始化

```
void TemplateInterpreter::initialize() {
    // 初始化解释器
    AbstractInterpreter::initialize();

    // 构建模板表
    TemplateTable::initialize();

    // 构建解释器生成器
    { ResourceMark rm;
        TraceTime timer("Interpreter generation", TraceStartupTime);
        int code_size = InterpreterCodeSize;
        NOT_PRODUCT(code_size *= 4); // debug uses extra interpreter code space
        _code = new StubQueue(new InterpreterCodeletInterface, code_size, NULL,
                             "Interpreter");
        InterpreterGenerator g(_code);
    }

    // 初始化转发表
    _active_table = _normal_table;
}
```

在模板解释器的初始化逻辑中，主要初始化了抽象解释器 AbstractInterpreter、模板表 TemplateTable、CodeCache 的 Stub 队列 StubQueue 和解释器生成器 InterpreterGenerator。关于这几个逻辑在后文都会进行详细分析，在这里只需要知道，在模板解释器的初始化链路中，调用到了 TemplateTable::initialize()这个接口进行模板表的构建，因此在 JVM 启动过程中，解释器初始化完成，模板表也就构建完成，每一个字节码指令都与其对应的生成器函数一一进行关联。

9.3.2 汇编器

对于模板解释器，每一个字节码指令都会关联一个生成器函数，用于生成字节码指令的本地机器码。例如 iload_1 字节码指令所对应的函数是 TemplateTable::iload(int n)，在该函数中主要调用 __movl(rax, iaddress(n))函数生成对应的机器指令，所生成的机器指令是 mov reg, operand，表示将操作数（立即数）传送至指定的寄存器中。在 x86 平台上，该函数会调用如下接口：

清单：/src/cpu/x86/vm/assembler_86.cpp

功能：movl()机器指令生成接口

```
void Assembler::movl(Register dst, Address src) {
```

```

InstructionMark im(this);
prefix(src, dst);
emit_byte(0x8B);
emit_operand(dst, src);
}
}

```

该函数属于 Assembler 类, Assembler 类是 JVM 内部为模板解释器所定义的汇编器, 当 JVM 使用模板解释器来解释执行字节码指令时, 便会通过汇编器来为每一个字节码指令生成对应的本地机器码。

在 Assembler::movl() 函数中调用 emit 系列的接口生成本地机器码, emit 系列的接口则定义在 AbstractAssembler 类中, 该类顾名思义是“抽象汇编器”。emit() 接口将机器码写入特定的内存位置, JVM 在运行期解释字节码指令时, 会跳转到特定的内存位置执行机器码。该函数会在后文讲解。

每个字节码指令都会关联一个生成器函数, 而生成器函数会调用汇编器生成机器码, 但是在 HotSpot 源码中并没有在生成器函数中看到直接调用汇编器的地方。仍以 iload_1 字节码为例, 其所对应的生成器函数 TemplateTable::iload(int n) 仅仅只是调用了 __ movl(rax, iaddress(n)), 并没有调用类似 assembler.movl(rax, iaddress(n)) 的函数, 然而 movl(rax, iaddress(n)) 函数的确定性在 /src/cpu/x86/vm/assembler_86.cpp 这个汇编器类中(x86 平台), 这究竟是如何关联的呢? 秘密就隐藏在 __ movl(rax, iaddress(n)) 这句指令的前缀“__”里面, 这是两个连续的下画线。事实上这是一个宏, 在 x86 平台上, 这个宏的定义如下:

清单: /src/cpu/x86/vm/templateTable_x86_32.cpp

功能: 模板表生成器函数中调用的汇编器宏

```

#ifndef CC_INTERP
#define __ masm-
// ...
#endif /* !CC_INTERP */

```

因此, 在模板表中可以通过添加 __ 前缀直接调用汇编器中的函数, 而不用添加类名。再以 iload_1 字节码指令为例, 在 x86 平台上其对应的生成器函数是 iload(int n), 该函数调用如下函数生成本地机器码:

```
__ movl(rax, iaddress(n));
```

在 HotSpot 源码编译的预处理阶段, 这句代码会被进行宏替换, 替换之后的代码变成如下:

```
_masm->movl(rax, iaddress(n));
```

如此一来, 模板表便与汇编器关联上了, 模板表的确是通过调用汇编器接口完成本地机器码指令的生成。

不过充满求知欲的你可能接着会想，/src/cpu/x86/vm/templateTable_x86_32.cpp 中的 _masm 这个变量定义在哪里？啥时候初始化？这是一个很好的问题。对于 TemplateTable 类，其 _masm 变量定义如下：

清单：/src/share/vm/interpreter/templateTable.hpp

功能：模板表中汇编器变量定义

```
class TemplateTable: AllStatic {
public:
    // ...
    static InterpreterMacroAssembler* _masm; // 汇编器
private:
    // ...
}
```

注意：TemplateTable 模板表中的汇编器变量类型是静态的。汇编器变量在 JVM 启动期间，JVM 调用字节码指令所对应的生成器函数时会对其进行赋值。在这里，不得不再次提起 JVM 的取指逻辑，上文在讲解取指逻辑时对其进行过详细分析。在 JVM 启动期间，JVM 会为所有字节码指令生成取指逻辑，HotSpot 通过 TemplateInterpreterGenerator::generate_and_dispatch() 接口来生成取指逻辑，上文讲过，HotSpot 为每一个字节码指令生成“取指”逻辑的同时（其实是指取下一条字节码指令），会为该字节码生成其本身所对应的机器逻辑指令。在 TemplateInterpreterGenerator::generate_and_dispatch() 接口中，会先调用 t->generate(_masm) 函数，为当前字节码指令生成字节码本身的机器逻辑，接着才会调用 _dispatch_epilog(tos_out, step) 函数为该字节码指令生成其对应的取指逻辑（取下一条字节码指令）。注意观察，在调用 t->generate(_masm) 时，为其传递了一个 _masm 指针，这个指针也是汇编器，其实模板表的静态变量 _masm 函数便是在 JVM 调用 t->generate(_masm) 函数时进行了赋值，看 t->generate(_masm) 函数定义：

清单：/src/share/vm/interpreter/templateTable.cpp

功能：模板解释器的汇编器赋值

```
void Template::generate(InterpreterMacroAssembler* masm) {
    // parameter passing
    TemplateTable::_desc = this;
    TemplateTable::_masm = masm;
    // code generation
    _gen(_arg);
    masm->flush();
}
```

在调用该函数时会将汇编器作为参数传递进来，而在该函数中调用`_gen(_arg)`函数时，由于`_gen`是各个字节码指令所对应的本地机器码生成函数，这些生成函数都是`TemplateTable`类的静态函数，因此JVM在调用这些函数时，这些生成器函数会调用汇编器的接口生成本地机器码，生成器函数所调用的汇编器实际上便是`Template::generate()`函数所传入的汇编器。

有点绕不是？还是以`iload_1`字节码指令为例来理一理思路，首先，JVM启动期间调用`TemplateTable`类的`initialize()`接口将每一个字节码指令与其生成器函数进行关联，例如`iload_1`字节码指令所关联的生成器函数就是`TemplateTable::iload(int i)`，但是在这个阶段，`TemplateTable::iload(int i)`并不会被调用。接着，JVM启动期间会调用模板解释器(`templateInterpreter`)为每个字节码指令生成该字节码指令的本地机器码指令(这个后文会讲解)，同时会为该字节码生成对应的取指的本地机器码指令(取下一条字节码指令)，这两个逻辑都封装在`TemplateInterpreterGenerator::generate_and_dispatch()`接口中，在该接口中调用`t->generate(_masm)`函数来为当前字节码指令生成机器指令，注意，模板表`TemplateTable::_masm`静态变量便在这个函数内部完成初始化，在`t->generate(_masm)`函数内部执行`TemplateTable::_masm = masm`将外部所传递进来的汇编器实例传递进`TemplateTable`模板表内部。`t->generate(_masm)`内部调用`_gen(_arg)`函数为字节码指令生成本地机器码。由于`iload_1`字节码指令对应的`_gen`是`TemplateTable::iload(int i)`函数，因此JVM会调用`TemplateTable::iload(int i)`函数。在`TemplateTable::iload(int i)`函数内部调用`_movl(rax, iaddress(n))`生成本地机器码。注意，这里的前缀“`_`”是两条下画线，这是一个宏，前面刚讲过，这个宏会在预处理阶段被替换成`_masm->`，因此JVM实际上调用的是`TemplateTable::_masm->movl(rax, iaddress(n))`函数。由于刚才在执行`t->generate(_masm)`函数时，JVM根据该函数所传入的汇编器`_masm`对`TemplateTable::_masm`变量进行了初始化，因此JVM所调用的便是传递给`t->generate(_masm)`函数的汇编器的`movl()`接口。其主体链路如下：

- ①.jvm启动
 - ...
调用`TemplateTable::initialize()`
- ②.`TemplateTable::initialize()`
- ③.`TemplateTable::def()`
 - `iload_1`字节码指令的`_gen`生成器映射为`TemplateTable::iload(int i)`函数
- ④.`TemplateInterpreterGenerator::generate_and_dispatch()`
- ⑤.`t->generate(_masm)`
 - 这里用`_masm`实例对`TemplateTable::_masm`进行了初始化
- ⑥.`_gen(_arg)`
 - 这里实际调用的是`TemplateTable::iload(int i)`
- ⑦.`_movl(rax, iaddress(n))`
 - 这里实际调用的是`TemplateTable::_masm->movl()`

对照上面的文字看这条链路，思路应该会清晰很多。至此，关于 TemplateTable 模板表中的汇编器 _masm 什么时候被初始化的问题理出了头绪，相信聪明的你立马会想到：既然 TemplateTable 模板表中的汇编器是在 JVM 调用 t->genenrate(_masm) 函数时被初始化，那么这里所传入的 _masm 汇编器又是啥呢？其实也很简单，首先确定这里的 _masm 指针所声明的类。由于 t->generate(_masm) 函数在 TemplateInterpreterGenerator::generate_and_dispatch() 函数中被调用，因此可以确定 TemplateInterpreterGenerator::generate_and_dispatch() 函数里的 _masm 变量是 TemplateInterpreterGenerator 类中的变量，TemplateInterpreterGenerator 是模板解释器生成器，所谓解释器生成器，前面讲过，是专门负责为模板解释器将字节码指令翻译生成本地机器码的类型。HotSpot 内部有好几种解释器，其他解释器也有自己专门的生成器，负责将字节码指令解释为特定的逻辑。TemplateInterpreterGenerator 继承自 AbstractInterpreterGenerator 类（看 /src/share/vm/interpreter/abstractInterpreter.hpp 文件），AbstractInterpreterGenerator 中便定义了一个汇编器指针，如下：

清单：/src/share/vm/interpreter/abstractInterpreter.hpp

功能：解释器生成器中的汇编器变量声明

```
class AbstractInterpreterGenerator: public StackObj {
protected:
    InterpreterMacroAssembler* _masm;
    // ...
}
```

注意，这个类名是 AbstractInterpreterGenerator，顾名思义，该类是“抽象解释器生成器”，是解释器生成器的顶级父类。既然模板解释器生成器 TemplateInterpreterGenerator 继承自这个基类，那么 TemplateInterpreterGenerator 自然也继承了汇编器指针 _masm，所以在 TemplateInterpreterGenerator::generate_and_dispatch() 函数中 JVM 才可以将汇编器指针直接传入 Template::generate() 函数中。

至此，关于汇编器初始化的问题，便演化成 TemplateInterpreterGenerator 中的汇编器啥时候初始化的问题。其实，汇编器初始化的逻辑隐藏在 CodeletMark 类的构造函数中，该类的定义如下：

清单：/src/share/vm/interpreter/interpreter.hpp

功能：CodeletMark 类的构造函数

```
class CodeletMark: ResourceMark {
private:
    InterpreterCodelet*          _clet; // 解释器脚本代码
    InterpreterMacroAssembler** _masm; // 这里定义了汇编器
    CodeBuffer                   _cb; // 代码缓存
```

```

// ...

public:
    CodeletMark(
        InterpreterMacroAssembler*& masm,
        const char* description,
        Bytecodes::Code bytecode = Bytecodes::_illegal):
        _clet((InterpreterCodelet*)AbstractInterpreter::code())->
request(codelet_size()),
        _cb(_clet->code_begin(), _clet->code_size())
{
    // initialize Codelet attributes
    _clet->initialize(description, bytecode);

    // 实例化一个汇编器
    masm = new InterpreterMacroAssembler(&_cb);
    _masm = &masm; // 将外部传进来的汇编器指针指向刚刚创建的汇编器实例对象
}

// ...
};


```

纵观整个 HotSpot 源码，只有在 CodeletMark 类的构造函数里对汇编器进行了实例化。CodeletMark 其实是一个包装器，能够自动回收所分配的代码空间以及所实例化的汇编器。那么 CodeletMark 类与 TemplateInterpreterGenerator 类中的汇编器有啥关系呢？说到这里，就不得不先讲一讲 HotSpot 字节码指令生成本地机器码的流程了。默认情况下 HotSpot 会启用模板解释器来解释执行字节码指令，在 JVM 启动期间，JVM 会依次调用模板解释器的生成器为各个字节码指令生成对应的本地机器指令及各个字节码指令的取指逻辑，前文讲过，这个逻辑主要封装在 TemplateInterpreterGenerator::generate_and_dispatch() 这个函数中。该函数调用的整体链路如下：

- ①.java.c: main() 调用 LoadJavaVM()
 - ②. java_md.c: LoadJavaVM() 调用 ifn->CreateJavaVM = (CreateJavaVM_t)dlsym(libjvm, "JNI_CreateJavaVM")
 - ③.jni.cpp: __JNICALL JNI_CreateJavaVM() 调用 result = Threads::create_vm((JavaVMInitArgs*) args, &can_try_again)
 - ④.thread.cpp: Thread::create_vm() 调用 init_globals()
 - ⑤.init.cpp: init_globals() 调用 interpreter_init()
 - ⑥.interpreter.cpp: interpreter_init() 调用 Interpreter::initialize()
 - ⑦.templateInterpreter.cpp: TemplateInterpreter::initialize()
 - ⑧.templateTable.cpp: TemplateTable::initialize() 初始化模板表
 - ⑨.templateInterpreter_x86_32.cpp: InterpreterGenerator(_code) 解释器生成器构造函数
- templateInterpreter_x86_32.cpp:

```

TemplateInterpreterGenerator::generate_all()
⑩.templateInterpreter.cpp:
TemplateInterpreterGenerator::set_entry_points_for_all_bytes()
⑪.templateInterpreter.cpp:
TemplateInterpreterGenerator::set_entry_points()
⑫.templateInterpreter.cpp:
TemplateInterpreterGenerator::set_short_entry_points()
⑬.templateInterpreter.cpp:
TemplateInterpreterGenerator::generate_and_dispatch()

```

上面这条链路是在 32 位 x86 平台上执行时的路径。其中与汇编器有关的关键点便在 TemplateInterpreterGenerator::set_entry_points()这个分支流中。前面多次提到，JVM 在启动期间会调用 TemplateInterpreterGenerator::generate_and_dispatch()接口为每个字节码指令生成对应的本地机器码及对应的取指指令，通过上面这条链路可以很清晰地看出来，TemplateInterpreterGenerator:: generate_and_dispatch() 接口便是由 TemplateInterpreterGenerator::set_entry_points()函数所调用，而该函数本身则又是由 TemplateInterpreterGenerator::set_entry_points_for_all_bytes() 函数所调用。这两个函数的主要逻辑如下：

```

清单：/src/share/vm/interpreter/templateInterpreter.cpp
功能：set_entry_points()与 set_entry_points_for_all_bytes 函数
//为所有字节码指令设置入口点
//DispatchTable::length 的值就是字节码指令集的总数量
void TemplateInterpreterGenerator::set_entry_points_for_all_bytes() {
    // 遍历所有字节码指令
    for (int i = 0; i < DispatchTable::length; i++) {
        Bytecodes::Code code = (Bytecodes::Code)i;
        if (Bytecodes::is_defined(code)) {
            set_entry_points(code);
        } else {
            set_unimplemented(i);
        }
    }
}

// 为指定字节码指令设置全部入口点
void TemplateInterpreterGenerator::set_entry_points(Bytecodes::Code code) {
    CodeletMark cm(_masm, Bytecodes::name(code), code);
    // ...
    if (Bytecodes::is_defined(code)) {
        Template* t = TemplateTable::template_for(code);
        set_short_entry_points(t, bep, cep, sep, aep, iep, lep, fep, dep, vep);
    }
    if (Bytecodes::wide_is_defined(code)) {

```

```

Template* t = TemplateTable::template_for_wide(code);
set_wide_entry_point(t, wep);
}

// set entry points
EntryPoint entry(bep, cep, sep, aep, iep, lep, fep, dep, vep);
Interpreter::_normal_table.set_entry(code, entry);
Interpreter::_wentry_point[code] = wep;
}

```

TemplateInterpreterGenerator::set_entry_points_for_all_bytes()函数的逻辑很好理解，遍历每一个字节码，然后调用 TemplateInterpreterGenerator::set_entry_points()函数为各个字节码指令设置入口点。关于入口点的概念在后文会分析，这里将目光聚焦于 TemplateInterpreterGenerator::set_entry_points()函数，该函数一开始便通过 CodeletMark cm(_masm, Bytecodes::name(code), code)实例化了一个 CodeletMark 类对象，这里就是关键点。实例化 CodeletMark 时，向其构造函数传递了指针变量_masm，该指针便是 TemplateInterpreterGenerator 类从抽象解释器生成器 AbstractInterpreterGenerator 中所继承而来的私有成员变量，该指针指向汇编器的内存首地址。在通过 CodeletMark cm(_masm, Bytecodes::name(code), code)创建 CodeletMark 类型实例对象时，TemplateInterpreterGenerator 类中的私有成员变量_masm 尚未初始化，但是上文刚刚讲到，在 CodeletMark 的构造函数中会实例化一个汇编器，并将外部所传入的汇编器指针指向这个所创建的汇编器实例对象。如此一来，当 TemplateInterpreterGenerator::set_entry_points()函数执行完 CodeletMark cm(_masm, Bytecodes::name(code), code)之后，TemplateInterpreterGenerator 类的私有成员变量_masm 便完成初始化，至此，本节一直在研究的问题——汇编器何时初始化，总算有了答案。

当 TemplateInterpreterGenerator 类的私有成员变量_masm 完成初始化之后，则当调用到上面的链路图中的最后链路 TemplateInterpreterGenerator::generate_and_dispatch()时，JVM 便将_masm 汇编器指针通过 t->generate(_masm)调用传递给了 TemplateTable::_masm 这个静态字段，最终在 TemplateTable 模板表中调用相关函数为字节码指令生成本地机器码时，实际所调用的便是 TemplateTable::_masm 这个静态字段所指向的汇编器实例的接口。那么接下来，聪明的你可能又会禁不住发问，汇编器到底是啥？到底是怎么将字节码指令翻译成对应的机器码呢？

在 CodeletMark 的构造函数中，直接将_masm 实例化成 InterpreterMacroAssembler 这个汇编器类型。事实上，在 HotSpot 内部，汇编器总共包含 4 个继承层次，如图 9.5 所示（32 位 x86 平台）。

Diagram showing the inheritance hierarchy of the Interpreter Macro Assembler class in the HotSpot JVM. It shows four levels of inheritance:
1. InterpreterMacroAssembler (最底层)
2. CodeletMark (直接继承)
3. InterpreterGenerator (间接继承)
4. TemplateInterpreterGenerator (间接继承)

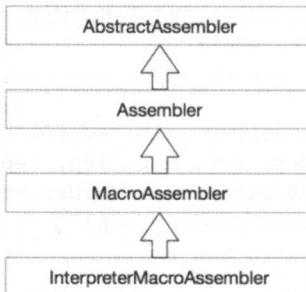


图 9.5 HotSpot 在 x86 平台上的 4 层汇编器继承关系

这 4 层汇编器所对应的文件分别如下：

- ◎ AbstractAssembler, assembler.hpp
- ◎ Assembler, assembler_x86.hpp
- ◎ MacroAssembler, assembler_x86.hpp
- ◎ InterpreterMacroAssembler, interp_masm_x86_32.hpp

顶级汇编器是 AbstractAssembler，顾名思义，就是抽象汇编器，但是该抽象汇编器其实一点都不抽象，它定义了最核心的功能和数据结构，如下：

清单：/src/share/vm/asm/assembler.hpp

功能：抽象汇编器的接口和核心字段

```
class AbstractAssembler : public ResourceObj {  
  
protected:  
    address      _code_begin;           // 指令缓存区首地址  
    address      _code_pos;            // 写入的当前位置  
  
    // ...  
  
    void emit_byte(int x); // 往指令区写入一个字节数据/指令  
    void emit_word(int x); // 往指令区写入一个 16 位的数据  
    void emit_long(jint x); // 往指令区写入一个 32 位的数据  
    void emit_address(address x); // 往指令区写入一个地址数据  
    // ...  
}
```

每一个汇编器都会往内存中写入一段指令，因此 JVM 会为每一个汇编器分配一个内存首地址，汇编器就从该首地址开始写入指令或数据。因此在这个抽象汇编器中定义了_code_begin、_code_pos 等字段，用于记录首地址及当前写入的位置。同时，抽象汇编器提供了写入本地机器指令的接口，就是 emit() 系列的函数。可以写入一个 8 位、16 位或者其他宽度的数据/指令。抽

象汇编器的子类都依赖于这些核心功能生成机器指令。而在汇编器的继承体系中，继承层次越深的汇编器，其处理的业务越抽象和复杂，也与 Java 的字节码指令越靠近。同时，除了最顶级的抽象汇编器，其子类皆与具体的硬件平台相关，这很好理解，因为机器码本身就是硬件相关的。对于 Assembler 和 MacroAssembler，在各种平台相关的源码中都有定义，例如在 assembler_sparc.hpp 中也定义了这两个类。

如果想深入理解 HotSpot 的解释执行原理，就必须对这几个汇编器子类有深入理解，下面对其进行简单的分析。

9.3.3 汇编

HotSpot 内部定义了 4 个层次的汇编器，顶级的抽象汇编器提供了往缓冲区写入本地机器码指令和数据的接口，并记录所写入的起始地址与当前地址。例如，emit_byte()接口提供往指令缓冲区写入一个字节码宽度的指令/数据的能力，其实现如下：

清单：/src/share/vm/asm/assembler.inline.hpp

功能：汇编器生成 1 字节的机器码指令功能演示

```
inline void AbstractAssembler::emit_byte(int x) {
    assert(isByte(x), "not a byte");
    *(unsigned char*)_code_pos = (unsigned char)x;
    _code_pos += sizeof(unsigned char);
    sync();
}
```

该函数在_code_pos 所指向的内存位置处写入一个输入的字节指令或数据，写入后，将_code_pos 指针再往前移动 1 字节，等待写入下一个机器指令。hotspot 默认使用模板解释器，JVM 在启动期间会初始化模板解释器，为每一个字节码指令在内存中写入其对应的机器指令片段，JVM 在运行期便能够对字节码指令进行解释，当执行某一条字节码指令时，会读取其对应的机器指令并执行，从而完成 Java 逻辑运算。

位于汇编器继承体系第二层的是 Assembler 汇编器。Assembler 其实是对物理机器指令的抽象，或者说软件封装。例如，在 x86 平台上，Assembler 类中的接口如下：

清单：/src/cpu/x86/vm/assembler_x86.hpp

功能：x86 平台上的 Assembler 定义

```
class Assembler : public AbstractAssembler {
    // ...
    void decl(Register dst);
    void decl(Address dst);
```

```

void incl(Register dst);
void incl(Address dst);

void lea(Register dst, Address src);
void mov(Register dst, Register src);

void push(int32_t imm32);
void push(Register src);
void pop(Register dst);

// ...
}

```

如果你熟悉汇编指令，应该对上面 Assembler 类中的这些 dec()、inc()、lea() 和 mov() 等接口感到非常亲切，例如，x86 平台提供的压栈指令中有一种指令格式是 push reg，于是 Assembler 类中便提供了一个接口 push(Register src)，这表示将一个硬件寄存器的值压入栈中。另一种压栈指令格式是 push operand，这表示将一个立即数压入栈中，于是 Assembler 类中便提供了一个接口 push(int32_t imm32)。Assembler 类中的这些接口基本是对物理机器指令的“纯净”模拟，即最终所生成出来的机器码与原生的机器指令是完全一致的，HotSpot 并不会往里面加入别的“杂项”。例如以压栈指令为例，在 x86 平台上，将一个立即数压栈的接口实现如下：

清单：/src/cpu/x86/vm/assembler_x86.cpp

功能：x86 平台上 Assembler::push() 的实现

```

void Assembler::push(int32_t imm32) {
    emit_byte(0x68);
    emit_long(imm32);
}

```

push() 函数里面调用 emit() 系列的接口往指令缓存区先写入 0x68 这个字节，再将 imm32 这个 32 位宽的立即数写入指令缓存区。emit() 系列的接口前文讲过，就是顶级抽象汇编器所提供的核心接口。0x68 是 Intel 处理器所提供的 push 机器指令的十六进制编码，例如 push 2，则对应的机器码是 0x68 02。因此，Assembler::push(int32_t imm32) 这个接口最终会向指令缓存区中写入下面这条机器码：

```
push imm32
```

这与实际的机器指令是一致的。因此，在 HotSpot 内部，调用 Assembler::push(int32_t imm32) 接口，可以完全等同于调用物理机器的 push operand 这个机器指令。所以 Assembler 类基本就是对物理机器指令的完全抽象，并且是“纯净版”的软抽象。

在汇编器继承体系中，到了 Assembler 汇编器的下一层，便是 MacroAssembler 汇编器，这

一层的汇编器仍然基于机器硬件指令进行抽象，但是不再是“纯净版”的抽象和模拟，而是被打上了深深的 Java 烙印，很多指令能够直接为 Java 数据所用。看看 MacroAssembler 在 x86 平台上的定义：

清单：/src/cpu/x86/vm/assembler_x86.hpp

功能：x86 平台上的 MacroAssembler 定义

```
class MacroAssembler: public Assembler {

    // bool 类型数据传送
    void movbool(Register dst, Address src);
    void movbool(Address dst, bool boolconst);
    void movbool(Address dst, Register src);

    // 加载/存储 Java 类的元数据 klass
    void load_klass(Register dst, Register src);
    void store_klass(Register dst, Register src);

    // 从 JVM 堆内存中加载 Java 对象实例
    void load_heap_oop(Register dst, Address src);
    void load_heap_oop_not_null(Register dst, Address src);
    void store_heap_oop(Address dst, Register src);

    // 对内存地址和自然数进行求和
    void addptr(Address dst, int32_t src) { LP64_ONLY(addq(dst, src))
NOT_LP64(addl(dst, src)) ; }

    // 传送 Java object 对象
    void movoop(Register dst, jobject obj);
    void movoop(Address dst, jobject obj);

    // 传送数组
    void movptr(ArrayAddress dst, Register src);

    // ...
}
```

注意观察 MacroAssembler 类所提供的接口，可以发现，也有类似物理机器级别的 mov、add 等指令，但是 MacroAssembler 类更进一步，延伸出 movbool 这种传送布尔值的接口，同时能够从 JVM 堆内存中读取 Java 类实例对象。再如，在物理机器级别，求和指令格式是 add oprd1, oprd2，但是 CPU 对 add 后面的 2 个操作数有要求，opr1 和 oprd2 均为寄存器是允许的，一个为寄存器而另一个为存储器也是允许的，但不允许两个都是存储器操作数，即不允许两个操作数都来自内存，或者一个来自内存一个来自立即数。但是在 MacroAssembler 汇编器中提供了接口 void addptr(Address dst, int32_t src)，该接口想实现的目标是将一个存储器操作数和一个立即

数累加求和，由于物理机器不支持这种指令，因此 MacroAssembler 汇编器只能对这个接口进行复杂处理，例如，在 32 位 x86 平台上，该接口最终调用如下函数才能实现累加：

清单：/src/cpu/x86/vm/assembler_x86.cpp

功能：MacroAssembler 类中实现将存储单元与立即数累加求和的逻辑

```
void Assembler::addl(Address dst, int32_t imm32) {
    InstructionMark im(this);
    prefix(dst);
    emit_arith_operand(0x81, rax, dst, imm32); // 0x81 是 Intel 平台上的 add 指令的十六进制编码
}

// immediate-to-memory forms
void Assembler::emit_arith_operand(int op1, Register rm, Address adr, int32_t imm32) {
    if (is8bit(imm32)) {
        emit_byte(op1 | 0x02); // set sign bit
        emit_operand(rm, adr, 1);
        emit_byte(imm32 & 0xFF);
    } else {
        emit_byte(op1);
        emit_operand(rm, adr, 4);
        emit_long(imm32);
    }
}
```

可以看到，对于这种物理机器不支持的指令，JVM 内部需要生成多条机器指令去处理，才能完成 MacroAssembler 汇编器中所定义的一个接口功能。因此，MacroAssembler 汇编器可以被看作对物理机器指令的组合封装（也可以认为是对其父类汇编器 Assembler 的组合封装），同时 MacroAssembler 能够支持 Java 内部数据对象级别的机器指令原语操作，从而为 JVM 内部的解释器完成特定逻辑提供必要的支撑。关于 MacroAssembler 类中的其他接口，各位有兴趣的道友可以自行深入研究。总之，汇编器模块应该代表了整个虚拟机中最精华的部分，而能否将精华全部消化吸收，完全就看个人的能力和修为。不过对于并非从事 JVM 开发的道友而言，倒没必要面面俱到，只要领悟了其思想便可。

在汇编器的继承体系中，MacroAssembler 汇编器的下一层是 InterpreterMacroAssembler 汇编器，顾名思义，这是解释器级别的汇编器，其直接为解释器提供相关汇编接口。先来看看这个类中都定义了些什么指令接口：

清单：/src/cpu/x86/vm/interp_masm_x86_32.cpp

功能：InterpreterMacroAssembler 汇编器主要接口

```
class InterpreterMacroAssembler: public MacroAssembler {
```

```

// ...

// 运行时获取参数或相关结果
void get_method(Register reg) { movptr(reg,
Address(rbp, frame::interpreter_frame_method_offset * wordSize)); }
void get_constant_pool(Register reg) { get_method(reg);
movptr(reg, Address(reg, methodOopDesc::constants_offset())); }
void get_constant_pool_cache(Register reg)
{ get_constant_pool(reg); movptr(reg, Address(reg,
constantPoolOopDesc::cache_offset_in_bytes())); }
void get_cpool_and_tags(Register cpool, Register tags)
{ get_constant_pool(cpool); movptr(tags, Address(cpool,
constantPoolOopDesc::tags_offset_in_bytes()));
}

void get_unsigned_2_byte_index_at_bcp(Register reg, int bcp_offset);
void get_cache_and_index_at_bcp(Register cache, Register index, int
bcp_offset, size_t index_size = sizeof(u2));
void get_cache_entry_pointer_at_bcp(Register cache, Register tmp, int
bcp_offset, size_t index_size = sizeof(u2));
void get_cache_index_at_bcp(Register index, int bcp_offset, size_t
index_size = sizeof(u2));

// 操作数栈相关指令
void f2ieee(); // truncate ftos to 32bits
void d2ieee(); // truncate dtos to 64bits

void pop_ptr(Register r = rax);
void pop_i(Register r = rax);
void pop_l(Register lo = rax, Register hi = rdx);
void pop_f();
void pop_d();

void push_ptr(Register r = rax);
void push_i(Register r = rax);
void push_l(Register lo = rax, Register hi = rdx);
void push_d(Register r = rax);
void push_f();

// ...

// 取指相关的指令
void dispatch_prolog(TosState state, int step = 0);
void dispatch_epilog(TosState state, int step = 0);
void dispatch_only(TosState state); // dispatch via rbx,
(assume rbx, is loaded already)
void dispatch_only_normal(TosState state); // dispatch normal
table via rbx, (assume rbx, is loaded already)

```

```

    void dispatch_only_noverify(TosState state);
    void dispatch_next(TosState state, int step = 0);           // load rbx, from
[esi + step] and dispatch via rbx,
    void dispatch_via (TosState state, address* table);        // load rbx, from
[esi] and dispatch via rbx, and table

    // jump to an invoked target
    void prepare_to_jump_from_interpreted();

    // ...
}

```

这个类中的接口分类比较明确，主要分为获取运行时参数相关的指令、操作数栈相关的指令、取指相关的指令等。事实上，还有性能监控的指令。由于 HotSpot 是一款基于栈式指令集的虚拟机，因此在 InterpreterMacroAssembler 类中可以看到各种 pop 和 push 指令接口，例如，将不同类型的数据压栈的指令包括 push_ptr()、push_i() 和 push_l() 等。同时，对于任何一个执行引擎而言，取指指令都是必须支持的（物理机器事实上没有软件取指指令，直接通过硬件数字电路触发，好高大上的感觉有没有……），因此 InterpreterMacroAssembler 类内部定义了各种取指相关的接口，这些接口统一以 dispatch 为前缀。其实，在 JVM 内部，取指也可以叫作分发，dispatch 的直接含义，这有其特殊的技术原因，下文会讲到。同时，JVM 内部在调用方法之前会创建栈帧，在动态绑定时会进行运行期链接，这些操作都需要解释器能够在运行期读取各种参数，例如，Java 方法在 JVM 内部所对应的 method 对象实例、Java class 字节码文件常量池在 Jvm 内存中的映像、字节码所对应的人口点缓存等，所以在 InterpreterMacroAssembler 类中提供了 get_method()、get_constant_pool() 等接口。以上这些核心的接口，将支撑起 JVM 内部解释器的运行期的各种调用，使得解释器能够站在 JVM 虚拟机这个层面或者这个高度来“看待”问题，或者说能够以高度抽象的思维来“思考”问题，而不仅仅局限于物理机器指令的各种原子化的琐碎的指令逻辑上。其实这本质上也是一种“面向对象”的抽象思维方式，一层一层地抽象，抽象通过软件函数的封装得以体现，函数所封装的不仅仅是函数，是能力。从机器到汇编，从汇编到 B 语言，从 B 到 C 语言，从 C 到 C++，从 C++ 到 Java，其实就是在一路抽象，一路封装，汇编是机器指令的简单符号代替，而 C 语言中的一个接口便封装了无数汇编能力，一个 Java 接口其实也封装了若干 C 和机器指令的特性。不仅编程语言如是，网络架构、分布式集群、中间件等无不是种种特性的抽象和封装。从这个角度去分析应用系统，也不难看出应用系统要分层的原因了。

大道至简。

在理解了这 4 层汇编器之后，再回头看 Java 的字节码指令。在上面所讲的这 4 层汇编器里，似乎并没有看到对 Java 字节码指令的汇编。事实上，前文讲过，字节码指令所对应的汇编生成

器都在 TemplateTable 模板表中，模板表为每个 Java 字节码指令提供了本地机器码生成器接口（或者解释入口），在生成器接口中调用汇编器生成字节码指令的本地机器逻辑。仍以 iload_1 这条字节码指令为例，其对应的生成器接口是 TemplateTable::iload(int i)，该接口仅通过调用 __movl(rax, iaddress(n)) 来生成机器码，前面分析过，这条指令会在预处理阶段将宏替换为 __masm->movl(rax, iaddress(n))，__masm 便是汇编器，movl(rax, iaddress(n)) 函数实现在前面已经给出过，这里不重复贴了，该函数最终会生成如下机器码（32位 x86 平台）：

```
mov    -0x4(%edi),%eax
```

在前面讲解 Java 方法堆栈的章节中讲过，当 JVM 调用一个 Java 函数之前会为其创建栈帧空间，被调用 Java 方法的局部变量表也会同时被创建，创建完栈帧之后，JVM 便会将 edi 寄存器指向局部变量表的第 0 个 slot 位置。因此上面这条机器指令的含义是：取相对于局部变量表第 0 个 slot 偏移量为 4 字节位置处（即第一个 int 型局部变量）的变量，将其值传送到 eax 寄存器中。

如果局部变量表的 slot 索引号大于 3，则将该变量推送至操作数栈栈顶的字节码指令便是 iload，而不是 iload_0、iload_1、iload_2 及 iload_3。例如，下面这个 Java 示例程序：

清单：/Test.java

功能：演示 iload 字节码指令

```
class Test{
    public static int add(int x, int y, int z){
        int sum = x + y + z;
        int sum2 = sum + 3;
        return sum2;
    }
}
```

在 Test 类中定义了一个静态方法 add()，其包含 3 个入参，方法内部包含两个局部变量。由于是静态方法，因此没隐式的 this 入参，所以 3 个入参的 slot 索引号分别是 0、1、2，而内部的两个局部变量的 slot 索引号则分别是 3 和 4。add()方法最终要将第 2 个局部变量 sum2 的值返回出去，因此必然会涉及一次 iload 操作。同时由于 sum2 在局部变量表中的 lslot 索引号是 4，大于 3，因此将其推送至操作数栈栈顶的字节码指令一定是 iload 4，而 add()方法的入参和其内部局部变量 sum 的入栈指令则分别是 iload_0、iload_1、iload_2 和 iload_3。可以使用 javap 命令进行验证，javap 的输出结果如下：

```
public static int add(int, int, int);
descriptor: (III)I
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=5, args_size=3
```

```

0: iload_0
1: iload_1
2: iadd
3: iload_2
4: iadd
5: istore_3
6: iload_3
7: iconst_3
8: iadd
9: istore      4
11: iload       4

```

从输出结果可以看出，编译器实际所生成的 load 系列指令与所推论的完全一致。对于 iload 4 这样的字节码指令，在模板表 TemplateTable 中为其所绑定的本地机器生成器函数是 TemplateTable::iload()，其定义如下（32 位 x86 平台）：

```

清单: /src/cpu/x86/vm/templateTable_x86_32.cpp
功能: TemplateTable::iload()函数定义
void TemplateTable::iload() {
    transition(vtos, itos);
    if (RewriteFrequentPairs) {
        Label rewrite, done;

        // ...

        __ bind(rewrite);
        patch_bytecode(Bytecodes::_iload, rcx, rbx, false);
        __ bind(done);
    }

    // Get the local value into tos
    locals_index(rbx);    //将局部变量的 slot 索引号读取到 rbx 寄存器中
    __ movl(rax, iaddress(rbx));
}

```

忽略该函数中的 if (RewriteFrequentPairs){} 分支，该函数的主要逻辑便只剩下 locals_index(rbx) 和 __ movl(rax, iaddress(rbx))，其中 locals_index(rbx) 顾名思义就是将局部变量的 slot 索引号读取到 rbx 寄存器中，而 __ movl(rax, iaddress(rbx)) 则将指定 slot 索引号的局部变量读取到 rax 寄存器中。最终，所生成的 TemplateTable::iload() 的本地机器逻辑如下：

```

--取字节码指令的操作数
--esi 寄存器指向当前字节码指令的起始地址
--假设字节码指令是 iload 6，则这条机器码将读取 iload 这个字节码指令所在内存位置的下一个
字节，下一个字节的值是操作数 6
movzbl 0x1(%esi), %ebx

```

```
--求补操作
neg    %ebx
```

--edi 指向局部变量表第 0 个 slot 位置

--上一步将 iload 6 的操作数 6 读进了 ebx 寄存器中，因此这里对第 0 个 slot 偏移 6 个位置，将该位置的局部变量值读进 eax 寄存器中

```
mov    (%edi,%ebx,4),%eax
```

如果对汇编语法很熟悉，则很容易读懂这几行汇编指令的逻辑，这便是 iload 字节码指令的机器实现逻辑。

上面分别演示了 iload_1 和 iload 这 2 条字节码指令的本地机器实现，虽然不能因此而言尽 JVM 执行引擎的全部细节，但是足以管中窥豹，把握 JVM 执行引擎的整体脉络，一通而百通。而事实上，HotSpot 为了提升性能而设计了模板解释器这种直接生成本地机器指令的机制，但是在一开始可没这么高大上，一开始的解释器真的只是解释器，并不是通过模板解释器将字节码指令直接翻译成对应的机器码去执行，而是直接使用 C 语言逻辑去解释字节码指令。在 HotSpot 中至今仍然保留着最原始的字节码解释器，其对应的文件是 bytecodeInterpreter.cpp。

下面仍以 iload 字节码为例，其在 bytecodeInterpreter 中的逻辑如下：

清单：/src/share/vm/interpreter/bytocodeInterpreter.cpp

功能：字节码解释器中的 iload 指令解析

```
BytecodeInterpreter::run(interpreterState istate) {
    // ...
    switch (opcode) {
        // ...

        CASE(_iload);
        CASE(_fload):
            SET_STACK_SLOT(LOCALS_SLOT(pc[1]), 0); // 将局部变量推送至操作数栈栈顶
            UPDATE_PC_AND_TOS_AND_CONTINUE(2, 1); // 更新 PC 计数器并取下一条字节码指令

            // ...
    }
    // ...
}
```

从这里可以看到，字节码解释器在解释 iload 指令时，调用 SET_STACK_SLOT(LOCALS_SLOT(pc[1]), 0) 函数将局部变量推送至操作数栈栈顶。该函数其实是一个宏，同时其里面的入参函数 LOCALS_SLOT 也是一个宏，这 2 个宏的定义如下（x86 平台）：

清单：/src/cpu/x86/vm/bytcodeInterpreter_x86.hpp

功能：SET_LOCALS_SLOT 和 SET_STACK_SLOT 宏定义

```
// 入参 pc[1]是 iload 指令后面的操作数，该操作数即为局部变量的 slot 索引号  
// 该宏返回指定 slot 索引号的局部变量值  
#define LOCALS_SLOT(offset) ((intptr_t*)&locals[-(offset)])  
  
// 将操作数栈顶指定位置的存储单元复制为 value  
#define SET_STACK_SLOT(value, offset) (*(intptr_t*)&topOfStack[-(offset)]  
= *(intptr_t*)(value))
```

通过这 2 个宏定义可以看出，使用 C 语言所实现的 iload 字节码指令的逻辑，与模板解释器中直接使用本地机器指令所实现的逻辑是一致的，解释 iload 字节码指令时，都是先从 iload 指令中解析出跟随在该操作码之后的操作数，该操作数便是局部变量的 slot 索引号，解释器根据该索引号取出局部变量的值，最终将该数值传送到操作数栈栈顶。

由于字节码解释器使用 C/C++ 逻辑来解释字节码指令，使用 C/C++ 的解释逻辑在静态编译阶段所生成的机器指令，肯定要比模板解释器中人工生成的机器指令烦琐和冗长，因此解释效率相当低，所以如今的 JVM 不再使用它了，但是源码仍然保留。

本书将字节码解释器特地摘录出来说明一二，只是希望有兴趣的道友能够更加明白模板解释器的运作机制，说不定原来对模板解释器的概念不甚清楚，看了字节码解释器之后能够恍然大悟，则也是值得的。

9.4 栈顶缓存

前面讲过，当 JVM 使用模板解释器运行字节码指令时，最后生成的 iload_1 这条指令的本地机器码如下（32 位 x86 平台）：

```
mov -0x4(%edi), %eax
```

HotSpot 提供了工具（HSDIS），可以使用该工具打印模板解释器为各个字节码指令所生成的本地机器码，可以据此验证实际生成的机器码是否与源码中所写的一致。HotSpot 在运行期所打印的 iload_1 的本地机器逻辑如下（32 位 x86 平台）：

```
iload_1 27 iload_1 [0xb36d8700, 0xb36d8740] 64 bytes  
  
[Disassembling for mach='i386']  
0xb36d8700: sub $0x4,%esp  
0xb36d8703: fstps (%esp)  
0xb36d8706: jmp 0xb36d8724  
0xb36d870b: sub $0x8,%esp
```

```

0xb36d870e: fstpl  (%esp)
0xb36d8711: jmp    0xb36d8724
0xb36d8716: push   %edx
0xb36d8717: push   %eax
0xb36d8718: jmp    0xb36d8724
0xb36d871d: push   %eax
0xb36d871e: jmp    0xb36d8724
0xb36d8723: push   %eax
0xb36d8724: mov    -0x4(%edi),%eax
0xb36d8727: movzbl 0x1(%esi),%ebx
0xb36d872b: inc    %esi
0xb36d872c: jmp    *-0x48f106a0(,%ebx,4)
0xb36d8733: nop
0xb36d8734: add    %al,(%eax)
0xb36d8736: add    %al,(%eax)
0xb36d8738: add    %al,(%eax)
0xb36d873a: add    %al,(%eax)
0xb36d873c: add    %al,(%eax)
0xb36d873e: add    %al,(%eax)

```

根据 `TemplateTable::iload(int n)` 这个 `iload_1` 字节码指令的生成器函数接口的代码逻辑，分析出只会生成 `mov -0x4(%edi),%eax` 这一条机器指令，可是 JVM 工具所打印出来的机器逻辑竟然有这么多，其中包含了 `mov -0x4(%edi),%eax` 这条指令，很奇怪吧！难道之前的分析存在问题？

为了进一步确认，可以再看一看 `iload` 这条字节码指令。前面讲过，模板解释器为该字节码指令所生成的本地机器逻辑如下（32位x86平台）：

```

movzbl 0x1(%esi),%ebx
neg    %ebx
mov    (%edi,%ebx,4),%eax

```

再看看JVM工具在运行时所生成的指令，如下（32位x86平台）：

```

iload 21 iload [0xb36d8420, 0xb36d8480] 96 bytes
[Disassembling for mach='i386']
0xb36d8420: sub    $0x4,%esp
0xb36d8423: fstps  (%esp)
0xb36d8426: jmp    0xb36d8444
0xb36d842b: sub    $0x8,%esp
0xb36d842e: fstpl  (%esp)
0xb36d8431: jmp    0xb36d8444
0xb36d8436: push   %edx
0xb36d8437: push   %eax
0xb36d8438: jmp    0xb36d8444
0xb36d843d: push   %eax
0xb36d843e: jmp    0xb36d8444
0xb36d8443: push   %eax

```

```
0xb36d8444: movzbl 0x1(%esi),%ebx  
0xb36d8448: neg    %ebx  
0xb36d844a: mov    (%edi,%ebx,4),%eax  
0xb36d844d: movzbl 0x2(%esi),%ebx  
0xb36d8451: add    $0x2,%esi  
0xb36d8454: jmp    *-0x48f106a0(%ebx,4)  
0xb36d845b: mov    0x2(%esi),%ebx  
0xb36d845e: bswap  %ebx  
0xb36d8460: shr    $0x10,%ebx  
0xb36d8463: neg    %ebx  
0xb36d8465: mov    (%edi,%ebx,4),%eax  
0xb36d8468: movzbl 0x4(%esi),%ebx  
0xb36d846c: add    $0x4,%esi  
0xb36d846f: jmp    *-0x48f106a0(%ebx,4)  
0xb36d8476: xchg  %ax,%ax  
0xb36d8478: add    %al,(%eax)  
0xb36d847a: add    %al,(%eax)  
0xb36d847c: add    %al,(%eax)  
0xb36d847e: add    %al,(%eax)
```

可以看到，`iload` 指令也存在同样的问题，实际所生成的本地机器逻辑与模板表中的生成器接口所设计的逻辑不一致。难道这里面真的存在什么问题不成？

事实上，模板表中所定义的生成器接口中所设计的逻辑没有错，JVM 工具打印出来的本地机器逻辑也没有错，所有的问题都与一个关键的优化措施有关，这个优化措施便是“栈顶缓存”。

可能绝大多数人第一眼看到栈顶缓存，脑子里都会打上一个大大的问号，这是个什么东西？栈顶就栈顶，为何还要加个缓存？缓存加在哪里？有什么好处？

在讲述 JVM 的栈顶缓存概念之前，先讲一讲 Java 开发中的缓存。有过 Java 开发及优化经验的道友基本都用过缓存技术，例如查询 DB 之前，通常会先查询缓存，如果缓存中没有，再去查询 DB。下面给出一段伪代码用于表述这种逻辑：

清单：/UserDao.java

功能：演示 Java 中查询 DB 时的缓存使用

```
class UserDao{  
    public List<User> queryUserId(Long userId){  
        List<User> userList = null;  
  
        // 先查询缓存  
        userList = cache.getForKey(userId);  
        if(userList != null && userList.size() > 0){  
            return userList;  
        }  
    }  
}
```

```
// 如果在缓存中查询不到，再查询 DB
userList = queryFromDB(userId);
if(userList != null && userList.size() > 0){

    // 如果从 DB 中查到数据，则将数据保存到缓存
    cache.put(userId, userList);

    return userList;
}

return userList;
}
```

相信这一段示例程序对于大部分 Java 道友而言，都不会陌生。在 Java 程序中使用缓存，可以减轻 DB 负担，从而提升应用程序的响应速度。这种缓存可以是本地缓存，也可以是远程的缓存集群。但是不管是本地缓存还是远程的缓存中心，数据应该都是存放在内存单元中，肯定不可能存放在磁盘中，否则还不如直接查询 DB。JVM 中的所谓栈顶缓存，与之完全不同。栈顶缓存的数据通过寄存器来暂存，并非内存。要明白这一点，还得懂一点计算机的硬件知识。对于 CPU 而言，一方面，就读取速度而言，CPU 从寄存器中读取速度最快，其次是内存，再次是磁盘。CPU 从寄存器中读取数据的速度往往比从内存中读取要快上好几个数量级（例如百倍），这种速度方面的差异非常大，毕竟相差百倍以上，这是任何一个优秀的系统设计师都不能避免的问题。另一方面，CPU 在执行运算时，例如做加法运算，并不能直接对两个内存中的数据直接进行求和，要么将两个数据全部从内存读取到寄存器中然后对两个寄存器进行求和，要么将一个数据读取进寄存器而另一个保留在内存中，这种规则本身没有问题，问题的关键在于，寄存器的数量是相当稀少的，一个 CPU 能够集成几十个寄存器便已经是奢侈至极，比起现代计算机的内存动辄就是 8GB、16GB 甚至 32GB 的巨大空间，这真的是沧海一粟。由于寄存器的数量很稀少，因此 CPU 往往在空间和效率上不能两全。例如要对两个数据进行求和，最快的方式当然是 CPU 直接对内存中的两个数据累加，但是 CPU 并不支持这种运算，因此在对两个数据进行求和之前，只能将其中一个数据先读取进寄存器，或者将两个数据全部读进寄存器然后才能进行累加。这中间必然存在效率上的牺牲。既然对内存数据进行运算这么慢，那将数据全部放进寄存器不就行了吗？这种方式并不可行，因为刚刚讲过，寄存器的数量是极其稀少的，因此编译器在将高级语言编译为机器语言时，会将数据加载进内存，而非全都加载进寄存器。

JVM 的栈顶缓存正是针对 CPU 这种在时间与空间上不能两全的遗憾而进行的改进措施，当然，这种改进措施蕴含了计算机运行机制的精华，也是 JVM 执行引擎的精华。

由于 CPU 无法同时兼顾时间与空间，而 JVM 追求的是性能，因此只能舍弃空间，这种取舍的结果便是，模板解释器在执行操作数栈操作时，如果按照常规的思路，肯定会将数据直

接压入栈顶，栈顶其实也是内存存储单元，但是 JVM 并没有走寻常路，为了追求性能，JVM 在执行操作数栈相关操作时，会优先将数据传送到寄存器，而非真正的栈顶。在后续流程中，CPU 执行运算时，便无须将数据再从栈顶传送到寄存器，因为数据本来就缓存在寄存器中，这便节省了一次内存读写，从而提升了 JVM 虚拟机运算指令的执行效率。这便是栈顶缓存。这种机制与 java 应用开发中使用缓存中间件的思路类似，只不过对于栈顶缓存而言，缓存介质是硬件寄存器，而非内存单元。

由于寄存器的数量十分有限，多的也就几十个，因此 JVM 并不是每次都能将数据存进寄存器，所以在将数据存进寄存器（即栈顶缓存）之前，必须先判断寄存器中是否有数据，如果有数据，得先想办法将数据移走，然后才能将当前操作的数据存进去。另外，Java 内建的数据类型很丰富，因此栈顶缓存的数据类型也是五花八门，什么都有，JVM 在将这些数据移走时必须考虑真实的数据类型，对待不同的数据类型处理逻辑也是不同的，所以 JVM 在解释一个字节码指令时，需要包含处理栈顶不同类型数据的逻辑，这便是前文使用 JVM 工具打印运行时所输出的 `iload_1` 和 `iload` 这 2 个字节码指令的本地机器码时代码比想象中要多得多的原因。这一点其实与 Java 应用开发的缓存策略类似，在 Java 应用开发中，如果要查询满足某种条件的数据，则需要考虑缓存中是否有数据，如果有，就从读取缓存的入口进入获取结果集，否则就从读取 DB 的入口进入获取结果集。很多应用为了进一步提升性能，会考虑两级缓存——本地缓存和远程缓存，本地缓存通常缓存热点数据，将热点数据加载到本地内存，从而避免访问远程缓存中心时的网络开销，从而在命中率超过 80% 的情况下，应用的整体性能将会提升。因此，在 Java 应用开发中要访问数据，通常也会有很多“入口点”，如果本地一级缓存中有数据，就从中取数；否则就判断远程缓存中是否有，若有，则从远程缓存中取数；否则就只能查询 DB。这些查询一级缓存和二级缓存的前置逻辑，便类似于 Java 字节码指令处理的前置逻辑，都是处理缓存。

以 `iload_1` 这条字节码指令为例，当栈顶缓存为空时，则 JVM 会直接将 slot 索引号为 1 的局部变量传送到寄存器 `ax` 中（x86 平台），并不需要先将栈顶数据移出去，因此 JVM 在运行期会直接从如下入口进入：

```
iload_1 27 iload_1 [0xb36d8700, 0xb36d8740] 64 bytes

[Disassembling for mach='i386']
0xb36d8700: sub    $0x4,%esp
0xb36d8703: fstps (%esp)
0xb36d8706: jmp    0xb36d8724
0xb36d870b: sub    $0x8,%esp
0xb36d870e: fstpl (%esp)
0xb36d8711: jmp    0xb36d8724
0xb36d8716: push   %edx
0xb36d8717: push   %eax
0xb36d8718: jmp    0xb36d8724
```

```

0xb36d871d: push    %eax
0xb36d871e: jmp     0xb36d8724
0xb36d8723: push    %eax
0xb36d8724: mov     -0x4(%edi),%eax      =====>直接从这一句开始解释
0xb36d8727: movzbl  0x1(%esi),%ebx
0xb36d872b: inc    %esi
0xb36d872c: jmp     *-0x48f106a0(,%ebx,4)
0xb36d8733: nop
0xb36d8734: add    %al,(%eax)
0xb36d8736: add    %al,(%eax)
0xb36d8738: add    %al,(%eax)
0xb36d873a: add    %al,(%eax)
0xb36d873c: add    %al,(%eax)
0xb36d873e: add    %al,(%eax)

```

而当栈顶缓存不为空时，例如栈顶缓存此时已经存储了一个 int 型的数据，则 JVM 在执行 `iload_1` 字节码指令时，需要先将栈顶缓存中的数据移到其本来应该存储的内存位置，然后再将 slot 索引号为 1 的局部变量传送到栈顶缓存（即寄存器）中，此时，`iload_1` 的进入点如下：

```
iload_1 27 iload_1 [0xb36d8700, 0xb36d8740] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d8700: sub    $0x4,%esp
0xb36d8703: fstps  (%esp)
0xb36d8706: jmp    0xb36d8724
0xb36d870b: sub    $0x8,%esp
0xb36d870e: fstpl   (%esp)
0xb36d8711: jmp    0xb36d8724
0xb36d8716: push   %edx
0xb36d8717: push   %eax
0xb36d8718: jmp    0xb36d8724
0xb36d871d: push   %eax
0xb36d871e: jmp    0xb36d8724      =====>直接从这一句开始解释
0xb36d8723: push   %eax
0xb36d8724: mov     -0x4(%edi),%eax
0xb36d8727: movzbl  0x1(%esi),%ebx
0xb36d872b: inc    %esi
0xb36d872c: jmp     *-0x48f106a0(,%ebx,4)
0xb36d8733: nop
0xb36d8734: add    %al,(%eax)
0xb36d8736: add    %al,(%eax)
0xb36d8738: add    %al,(%eax)
0xb36d873a: add    %al,(%eax)
0xb36d873c: add    %al,(%eax)
0xb36d873e: add    %al,(%eax)
```

这一次的入口点变成了 `push %eax`，其作用是将 `eax` 寄存器（即栈顶缓存）中的数据先推送

至操作数栈栈顶，然后再接着执行 iload_1 本身的逻辑。

那么 iload_1 在什么情况下，在执行之前，栈顶缓存是空的呢？可以看下面这个例子：

清单：/Test.java

功能：示例 iload_1 执行之前栈顶缓存为空的情况

```
class Test{
    public static int add(int x, int y){
        int z = 8;
        int sum = y + z;
        return sum;
    }
}
```

编译该 Java 类并使用 javap 命令分析编译后的字节码文件，输出如下：

```
public static int add(int, int);
descriptor: (II)I
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=4, args_size=2
  0: bipush     8
  2: istore_2
  3: iload_1
  4: iload_2
  5: iadd
  6: istore_3
  7: iload_3
  8: ireturn
```

观察这一段字节码指令，在执行 iload_1 之前，先执行了 bipush 8 和 istore_2，这 2 条字节码指令将完成为变量 z 赋值的逻辑。istore_2 字节码指令执行完成之后，由于栈顶并没有数据等待操作，用不着将数据缓存在寄存器中，因此此时寄存器中没有数据。在这种情况下，当执行 iload_1 指令时，JVM 便会直接从 iload_1 的本地机器码 mov -0x4(%edi),%eax 开始执行。

修改上面的 Test 类，改成如下：

清单：/Test.java

功能：示例 iload_1 执行之前栈顶缓存不为空的情况

```
class Test{
    public static int add(int x, int y){
        int z = 8;
        int sum = x + y;
        return sum;
    }
}
```

编译该 Java 类并使用 javap 命令分析编译后的字节码文件，输出如下：

```
public static int add(int, int);
descriptor: (II)I
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=4, args_size=2
  0: bipush      8
  2: istore_2
  3: iload_0
  4: iload_1
  5: iadd
  6: istore_3
  7: iload_3
  8: ireturn
```

注意观察，此时在执行 `iload_1` 指令之前，JVM 必须先执行 `iload_0` 这条指令。`iload_0` 指令执行完之后，由于使用了栈顶缓存策略，因此 JVM 会将 slot 索引号为 0 的局部变量直接传送到寄存器指针（即栈顶缓存）中，而不是直接将该局部变量推送至栈顶，如此一来，栈顶缓存（即寄存器）中便有数据了，并且是 int 类型的。接着 JVM 开始执行 `iload_1` 这条指令，由于栈顶缓存中已经有数据，因此 JVM 必须先将 `iload_0` 所加载的数据从栈顶缓存中移走，所以 JVM 执行 `iload_1` 指令时就必须从 `push %eax` 这条机器码开始执行，该指令会将 `iload_0` 指令加载到栈顶缓存的数据推送至操作数栈栈顶。

以上便是栈顶缓存的基本原理。其实栈顶缓存远不止这么简单，但是要想深入研究该技术，必须对机器指令及计算机底层非常熟悉，笔者不知道各位道友对此是否足够感兴趣，因此关于栈顶缓存的话题先讲这么多。如果大家果真求知欲特别强烈，笔者会探本求原，深入揭示栈顶缓存的来龙去脉。

9.5 栈式指令集

Java 的字节码指令都是面向栈的，面向栈的指令集往往有一个特点：不需要指定操作数，专业术语就是“零地址”指令。例如，在 Java 中对两个 int 类型的数据求和，其对应的字节码指令是 `iadd`。乍一看，这种指令很是让人感觉“无厘头”，不是说好的对两个数求和的吗，数呢？这种指令的写法与通常意义上我们所见到的加法表达式格式相去甚远，因此有点让人莫名其妙。在数学中，对两个数求和，司空见惯的一种表达式肯定下面这种：

$x = y + z$

这种表达式非常简易明了，随便一个人一看都知道是在对两个数求和。不仅计算器讲究封

装的艺术，数学也讲究封装，而事实上，数学里的封装要算得上计算机封装的老祖宗了，现代计算机的诞生其实是数学概念抽象的结果，抽象便是封装。将上面这个数学表达式抽象成一个数学函数，变成：

```
add(x, y, z)
```

计算机本来就是用于处理数字信号的，对于这种函数，某些 CPU 天生能支持，例如，ARM 处理器。CPU 在处理这种数学函数时，会通过约定的指令格式来完成。例如，上面这个函数翻译成计算器指令，可以如下：

```
add x, y, z
```

这便是物理计算机支持的硬件指令，这种指令能够对操作码后面的两个数求和，并将累加结果保存到操作码后面的第一个操作数中。将该指令的表达形式进行抽象，可以得到下面这种一般意义上的指令格式：

```
op dest, src1, src2
```

这种指令格式的含义是：对 op 操作码后面的两个操作数 src1 和 src2 进行某种操作，并将操作结果保存到 op 操作码后面的第一个操作数 dest 中。有了这种通用的表达式，计算机便能够支持各种数学运算，例如，对两个数执行减法运算，指令如下：

```
sub dest, src1, src2
```

其实，类似 “op dest, src1, src2” 这种格式的指令，在计算机中叫作“三地址”指令，所谓三地址指令，其实就是指操作码 op 的后面跟了 3 个操作数。由于在计算机内部，数据都存储在内存或寄存器中，因此 op 操作码后面的操作数通常都是指某个内存单元编号或者某个特定的寄存器，所以叫作“三地址”。

不过有些 CPU 设计者认为三地址指令太长，需要简化。例如， $x = y + z$ 这种表达式可以简化成 $y += z$ ，这种表达式仍然能够对两个数进行求和，并将求和结果保存到其中一个数字之中。对 $y += z$ 表达式重新编排，可以写成下面这种格式：

```
+= y, z
```

乍一看，这种格式很奇怪，但是如果将其使用数学函数来表达，则可以抽象成下面这种函数：

```
add(y, z)
```

这样的数学函数表达式大家都懂。如果让计算机指令来支持这种数学函数，则可以写成下面这种格式：

```
add y, z
```

x86 系列的处理器便能够直接执行这种指令对两个数进行求和。若使用 C 语言编写程序计算 $x = y + z$, 则在 x86 平台上编译后便会生成类似上面这条指令的机器码。将该指令进行抽象, 可以得到下面这种通用的指令格式:

```
op dest, src
```

这种指令的含义是: 对 op 操作码后面的两个操作数进行某种运算, 并将运算结果保存在 dest 第一个操作数中。这种格式的指令使用专业术语描述叫作“二地址”指令, 意思很明确, 操作码后面跟了 2 个操作数。相比于三地址指令, 二地址指令所需要的内存空间变小了, 并且整个指令也更加精简。

上面这两种指令格式, 无论是三地址格式还是二地址格式, 就指令格式本身而言, 普罗大众都比较容易根据指令格式而明白其功能, 并且这种格式也符合数学领域中的函数抽象, 相比于 Java 中的 iadd 这种“零地址”的指令格式, 更加容易被人接受。其实, “零地址”这种格式的指令会让人产生慌乱, 慌乱的原因是: 不知道这种指令对谁执行累加。而无论是二元地址还是三元地址的指令, 至少所操作的目标数据都包含在指令中。

事实上, 大凡设计成“零地址”格式的指令集, 通常都是面向栈的指令集, 既然是面向栈的, 则指令所操作的源数据和目标数据, 便默认存放于栈上。而上面二元地址和三元地址的指令集, 更多的是基于寄存器的架构, 所操作的数据直接位于寄存器中(当然也有部分位于内存单元中)。之所以二元和三元地址指令集大多数直接面向寄存器, 是因为这种多元地址指令后面所跟随的操作数可以直接被指定为目标寄存器, 而 CPU 读取寄存器的性能要远远高于内存, 因此能够直接基于寄存器运算的指令, 没必要设计成面向栈式操作, 否则反而不美。而零地址指令往往不能设计成面向寄存器操作, 或者说设计难度很大, 因为不同的硬件平台, 其所集成的寄存器数量、内部标识、指令格式并不相同, 而零地址指令由于并不直接包含操作数, 因此无法明确指定所操作的数据到底位于哪个寄存器中, 所以不能很容易地设计成面向寄存器操作, 只能设计成基于栈操作, 因为不管何种计算器, 栈的概念都是被支持的, 并且语义也是一致的。同时由于这种特性, 因此寄存器式指令集一般是硬件平台相关的, 而栈式指令集则能跨平台。例如, 在 x86 平台上执行两个数据相加, 可以直接在指令中指定所操作的元数据位于哪个寄存器, 以及结果数据位于哪个寄存器, 例如下面这条指令:

```
add %ax, %bx
```

这条求和指令直接指定源数据分别位于 ax 和 bx 这两个通用寄存器中, 并且操作结果存储在 bx 寄存器中(AT&T 语法)。而如果使用栈式指令进行求和, 例如 Java 中的 iadd 指令, 假设该指令是基于寄存器的, 那么问题来了, iadd 本身就是一条完整的指令, 并不包含任何操作数, 那么其所操作的源数据到底在哪两个寄存器中呢? ax? bx? 不知道, 32 位的 x86 CPU 有 8 个 32 位通用寄存器, 而 SUN 的 SPARC 处理器则有 24 个通用寄存器。寄存器是如此之多, 而 iadd

指令又无法指定所操作的源数据位于哪里，因此如果硬生生将零地址指令设计成面向寄存器的，难度可想而知。

进一步说，即使规定默认的寄存器地址而将基于栈的指令集设计成面向寄存器的架构，其灵活性也会大打折扣。例如，在 x86 平台上能够支持直接对一个内存存储单元和一个寄存器求和，例如：

```
add (%ax), %bx
```

该指令将 ax 寄存器所指的内存地址的数据与 bx 寄存器进行累加，并将累加结果保存进 bx 寄存器中（AT&T 语法）。对于这种灵活的求和指令，很显然，零地址的指令集根本支持不了。JVM 由于刚“出道”时便以“跨平台”作为最大的特性大力宣传，因此其指令集便选择了零地址的格式，而零地址的指令集又只能选择基于栈的架构，实在是有其深刻的技术制约因素。由于指令集是面向栈的，因此 JVM 的执行引擎内部也只能紧紧围绕栈来实现，上文所讲的栈顶缓存便是针对堆栈而进行的一项优化措施，其优化的思路还是优先使用寄存器。需要注意的是，这里所言的栈是指“求值栈”，而不是指系统调用栈（system call stack）。有些虚拟机把求值栈实现在系统调用栈上，但两者概念上不是一个东西。

栈式指令集既然是零地址格式，那就意味着其操作的源数据与目的数据皆位于求值栈栈顶。既然所操作的数据位于栈顶，那么在操作之前，必然会有指令将数据先传送到栈顶，所以使用一条寄存器式指令便能实现的逻辑，往往需要多条栈式指令方能实现。例如，在 JVM 执行 iadd 指令进行求和之前，必定会有 iload 系列或者 iconst 或者 bipush 等将数据推送至栈顶的前置指令。例如下面这个示例：

清单：/Test.java

功能：演示 iadd 指令

```
class Test{
    public static void main(String[] args) {
        int a = 66;
        int b = 87;
        int sum = a + b;
    }
}
```

编译该类并使用 javap 命令分析编译后的字节码文件，分析结果输出如下：

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=4, args_size=1
0: bipush      66
```

```

2:  istore_1
3:  bipush      87
4:  istore_2
5:  iload_1
6:  iload_2
7:  iadd
8:  istore_3
9:  return
10: 
```

观察这段字节码指令，在 iadd 指令之前，存在两条前置指令 iload_1 和 iload_2，这两条指令分别将 slot 索引号为 1 和 2 的 int 类型的数据推送至操作数栈栈顶，这两个推送的数据其实便是接下来 iadd 指令的源操作数。那么 iadd 指令执行完成之后，所求取的结果数据存储在哪里呢？这得先从字节码指令执行的原理说起。

前文讲过，JVM 在启动阶段会为所有字节码指令生成本地机器码指令。而本地机器码指令大多数都是基于寄存器的。从这个角度来看，JVM 的栈式指令集其实只能算作是“伪指令”，因为一方面，JVM 并不具备真正具有运算能力的硬件数字电路，另一方面，JVM 的字节码指令最终仍然需要基于寄存器架构的本地机器指令才能执行。对于 iadd 指令而言，使用 HSDIS 工具在运行期打印该指令的本地指令如下（32 位 x86 平台）：

```

iadd 96 iadd [0xb36d9f40, 0xb36d9f60] 32 bytes
[Disassembling for mach='i386']
0xb36d9f40: pop    %eax
0xb36d9f41: pop    %edx
0xb36d9f42: add    %edx,%eax
0xb36d9f44: movzb1 0x1(%esi),%ebx
0xb36d9f48: inc    %esi
0xb36d9f49: jmp    *-0x48f106a0(,%ebx,4)
```

如果你熟悉汇编，上面这段代码一看便能明白其逻辑：先执行 pop %eax 和 pop %edx 指令将栈顶的两个数据分别弹出至 eax 和 edx 这两个寄存器中，接着执行 add %edx,%eax 指令对这两个数据进行求和，并将求和结果再存储进 eax 寄存器中（AT&T 语法）。这样便完成了累加的运算。不过运算完成之后，结果被存储在了 eax 寄存器中，其实这里仍然使用了栈顶缓存策略——优先将数据存储进寄存器。但是事实上，按照 JVM 内部对 iadd 指令的定义，该指令执行完成之后，数据应该会被存储进栈顶——因为整个 JVM 的执行引擎和指令集都是面向栈操作的。

前面讲了 JVM 的指令集为何是面向栈的，以及为何被设计成零地址格式和栈式指令集的实现方式。总体而言，虚拟机选择栈式指令集有利有弊，好处大抵有如下几个：

- 编译器更加容易设计和实现。因为不用考虑一堆寄存器了，只需要关注栈顶即可。
- 指令集非常精简小巧。因为大部分指令都是零地址格式，一个字节就能完成某种

操作，所占内存空间也小。

- ◎ 跨平台。虽然不同的硬件平台，其寄存器和机器级别的指令格式不同，但是 JVM 不管这些，只关注栈，栈总是所有硬件平台都要支持的一种内存结构，或者说即使没有栈，只要有内存即可。

但是得到这些好处的同时，也带来了如下硬伤：

- ◎ 实现同样的功能，需要更多的指令。以求和为例，基于寄存器的指令集只需执行 add dest, src 这样的指令便能完成求和，而栈式指令集则需要先 load 再 add，需要多条指令。所以虽然单个指令变小了，但是完成一个功能需要更多的指令，总体上其实并没有精简太多，甚至局部反而更加烦琐。
- ◎ 性能下降。这是由于栈式指令集需要更多的指令才能完成某个功能，并且完成同一个功能所对应的本地机器码比原本使用纯粹的基于寄存器的机器指令更多，CPU 需要更多的时钟周期，需要制造更多电子脉冲。

前面说过，JVM 之所以使用栈式指令集，是为了跨平台，但是同样使用 Java 语法规范的 Android 技术体系，却直接使用了寄存器式指令集。安卓 4 之前，安卓使用 Dalvik 虚拟机来解释执行安卓字节码文件，虽然 HotSpot 与 Dalvik 同样都是虚拟机，但是为何 Dalvik 没有使用栈式指令集呢？问题就出在跨平台上。安卓刚“出道”时，所针对的就是 ARM 系列处理器，该处理器有 16 个 32 位通用寄存器，而 Dalvik 里面则设计了 16 个虚拟寄存器，在运行期，Dalvik 会将虚拟寄存器全部映射到物理寄存器，从而让 Dalvik 能够在 ARM 处理器上高效执行，从而充分发挥寄存器架构的优势。所以安卓系统一开始仅仅是借用了 Java 跨平台的语法，但是并没有为其打造一颗跨平台的“心脏”，所以安卓并不能直接移植到其他异构的手机平台上。

由于 JVM 的性能比较低，因此大神们想尽了各种办法来优化性能，各种奇思妙想层出不穷，各路大招层见迭出，所用技能让人叹为观止！HotSpot 内部解释器从最原始的字节码解释器（以 C 语言函数逻辑解释执行字节码）升级到模板解释器（直接生成本地机器码），解决了原始解释器效率低下的突出问题（很严重，严重到被 C/C++ 程序员嘲笑）。接着加入了 JIT 编译器，其能够在运行期针对热点代码进行即时编译，JIT 使用了多种优化策略使得编译出来的代码指令更高效，例如将代码内联减少字节码跳转次数，能够加快热点代码的运行效率。HotSpot 还针对客户端和服务端分别开发了 C1 和 C2 两层编译优化功能，C1 和 C2 属于动态自适应编译器，其中 C1 编译器仅做了简单优化，这主要是因为客户端的 JVM 虚拟机追求快速启动、快速响应，如果编译时间过长，反而影响客户端体验，而 C2 则会做深层次的优化，这种编译器所编译出来的代码质量较高，但是因此需要较长的编译时间成本，所以仅适合用于服务端的 JVM。JIT、C1、C2 这些优化技术太过于底层，并且需要很深厚的编译原理内功，有兴趣的道友可以一起

研究。

但是对于 JIT 等优化技术也不能过于迷信，使用 JIT 编译出来的代码指令的执行效率并不一定就比直接解释执行的效率高，甚至如果一个方法并不是经常被调用，那么可能执行一次 JIT 编译的时间成本都要高于直接解释执行一次该方法的时间。但是很多时候使用 JIT 编译所获得的代码质量，却比使用 C/C++ 代码完成同样的 Java 逻辑的代码质量还要高，这才是 JIT 迷人的地方。

这里举一个例子来比较 JVM 使用模板解释器和使用 JIT、C1 和 C2 编译器所生成的本地代码，Java 代码示例如下：

清单：/Test.java

功能：比较 JVM 使用模板解释器和使用 JIT、C1 和 C2 编译器所生成的本地代码

```
public class Test {
    public static int calc(int i, int j) {
        int b = 0x211;
        int sum = i + b;
        sum = sum - j;
        b = sum * i;
        sum = j * (b - 5);
        return sum;
    }

    public static void main(String[] args) throws Exception {
        for(int k = 0; k < 100000; k++) {
            calc(k, k+1);
        }
    }
}
```

该示例的 main() 主函数中使用 for 循环重复调用 calc(int, int) 方法，并且循环了 10 万次。之所以要循环这么多次，是为了让 JVM 能够“侦查”到 calc(int, int) 方法是一个热点代码。对于热点代码，HotSpot 会调用 JIT 即时编译器将其编译成高质量的本地代码。同时，在 calc(int, int) 方法里面进行了比较复杂的运算，之所以设计得比较复杂，是为了接下来要比较 C1 和 C2 这两款编译器所生成的本地机器指令的质量，如果算法简单，JVM 可能压根就不开启 C2 分层优化编译器，这样便无法比较。

使用 HSDIS 插件可以查看运行后 JIT 所生成的本地汇编指令，所生成的本地汇编代码分为 C1 版和 C2 版（即 C1 编译器和 C2 编译器分别生成的本地指令），其中 C1 版如下：

```
CompilerOracle: print *Test.calc
Java HotSpot(TM) 64-Bit Server VM warning: printing of assembly code is
enabled; turning on DebugNonSafepoints to gain additional output
```

```

Compiled method (c1) 103 35 3 Test::calc (24 bytes)
total in heap [0x000000010eaa3b90,0x000000010eaa3ea0] = 784
relocation [0x000000010eaa3cb8,0x000000010eaa3ce0] = 40
//.....
Loaded disassembler from HSDIS-amd64.dylib
Decoding compiled method 0x000000010eaa3b90:
Code:
[Disassembling for mach='i386:x86-64']
[Entry Point]
[Verified Entry Point]
[Constants]
# {method} {0x00000001276f0330} 'calc' '(II)I' in 'Test'
# parm0: rsi      = int
# parm1: rdx      = int
#           [sp+0x40] (sp of caller)
0x000000010eaa3ce0: mov    %eax,-0x14000(%rsp)
0x000000010eaa3ce7: push   %rbp
0x000000010eaa3ce8: sub    $0x30,%rsp
0x000000010eaa3cec: movabs $0x1276f04e0,%rax ; {metadata(method data
for {method} {0x00000001276f0330} 'calc' '(II)I' in 'Test')}
0x000000010eaa3cf6: mov    0xdc(%rax),%edi
0x000000010eaa3cff: add    $0x8,%edi
0x000000010eaa3cfc: mov    %edi,0xdc(%rax)
0x000000010eaa3d05: movabs $0x1276f0330,%rax ; {metadata({method}
{0x00000001276f0330} 'calc' '(II)I' in 'Test')}
0x000000010eaa3d0f: and    $0x1ff8,%edi
0x000000010eaa3d15: cmp    $0x0,%edi
0x000000010eaa3d18: je     0x000000010eaa3d3e ;*sipush
; - Test::calc@0 (line 35)

0x000000010eaa3d1e: mov    %rsi,%rax
0x000000010eaa3d21: add    $0x211,%eax
0x000000010eaa3d27: sub    %edx,%eax
0x000000010eaa3d29: imul   %esi,%eax
0x000000010eaa3d2c: sub    $0x5,%eax
0x000000010eaa3d2f: imul   %edx,%eax
0x000000010eaa3d32: add    $0x30,%rsp
0x000000010eaa3d36: pop    %rbp
0x000000010eaa3d37: test   %eax,-0x20c0c3d(%rip) # 0x000000010c9e3100
; {poll_return}
0x000000010eaa3d3d: retq
0x000000010eaa3d3e: mov    %rax,0x8(%rsp)
0x000000010eaa3d43: movq   $0xfffffffffffffff,%rsp
0x000000010eaa3d4b: callq  0x000000010ea875e0 ; OopMap{off=112}
; *synchronization entry
; - Test::calc@-1 (line 35)
; {runtime_call}

```

```

0x000000010eaa3d50: jmp    0x000000010eaa3d1e
0x000000010eaa3d52: nop
0x000000010eaa3d53: nop
0x000000010eaa3d54: mov    0x2a8(%r15),%rax
0x000000010eaa3d5b: movabs $0x0,%r10
0x000000010eaa3d65: mov    %r10,0x2a8(%r15)
0x000000010eaa3d6c: movabs $0x0,%r10
0x000000010eaa3d76: mov    %r10,0x2b0(%r15)
0x000000010eaa3d7d: add    $0x30,%rsp
0x000000010eaa3d81: pop    %rbp
0x000000010eaa3d82: jmpq   0x000000010e9f5b20 ; {runtime_call}
0x000000010eaa3d87: hlt
// ...
0x000000010eaa3d9f: hlt
[Exception Handler]
// ...

```

观察上面 C1 编译器所生成的机器指令，HSDIS 工具给出 Test.calc()方法入参的载体，这两个人参的载体分别是：

```

# parm0:    rsi      = int
# parm1:    rdx      = int

```

这表示 Test.calc(int i, int j) 的入参 i 的值将被从 main() 主函数传递给 rsi 寄存器，而入参 j 的值将被传递给 rdx 寄存器。C1 编译器所生成的核心算法指令从地址 0x000000010eaa3d1e 开始，到地址 0x000000010eaa3d2f 结束，一共包含 6 条机器指令，如下：

```

0x000000010eaa3d1e: mov    %rsi,%rax
0x000000010eaa3d21: add    $0x211,%eax
0x000000010eaa3d27: sub    %edx,%eax
0x000000010eaa3d29: imul   %esi,%eax
0x000000010eaa3d2c: sub    $0x5,%eax
0x000000010eaa3d2f: imul   %edx,%eax

```

这 6 条机器指令完成 Test.calc(int, int) 方法内部的算法逻辑，懂汇编的道友一看就会明白这 6 条指令的含义，的确与 Test.calc(int, int) 方法的逻辑完全保持一致。这 6 条指令之前有 11 条机器指令完成相关的准备工作，这 6 条指令执行完成之后，再执行 3 条指令便会执行 retq 指令而结束。

再看看 C2 版的汇编指令：

```

Compiled method (c2) 105 36 4 Test::calc (24 bytes)
total in heap [0x000000010eaa6b10,0x000000010eaa6d00] = 496
relocation [0x000000010eaa6c38,0x000000010eaa6c40] = 8
// ...
dependencies [0x000000010eaa6cf8,0x000000010eaa6d00] = 8
Decoding compiled method 0x000000010eaa6b10:

```

```

Code:
[Entry Point]
[Verified Entry Point]
[Constants]
# {method} {0x00000001276f0330} 'calc' '(II)I' in 'Test'
# parm0:    rsi      = int
# parm1:    rdx      = int
# [sp+0x20]  (sp of caller)
0x000000010eaa6c40: sub    $0x18,%rsp
0x000000010eaa6c47: mov    %rbp,0x10(%rsp) ;*synchronization entry
; - Test::calc@-1 (line 35)

0x000000010eaa6c4c: mov    %esi,%eax
0x000000010eaa6c4e: sub    %edx,%eax
0x000000010eaa6c50: add    $0x211,%eax
0x000000010eaa6c56: imul   %esi,%eax
0x000000010eaa6c59: add    $0xfffffffffffffff,%eax
0x000000010eaa6c5c: imul   %edx,%eax ;*imul
; - Test::calc@20 (line 39)

0x000000010eaa6c5f: add    $0x10,%rsp
0x000000010eaa6c63: pop    %rbp
0x000000010eaa6c64: test   %eax,-0x20c3c6a(%rip) # 0x000000010c9e3000
; {poll_return}
0x000000010eaa6c6a: retq
0x000000010eaa6c6b: hlt
// ...
0x000000010eaa6c7f: hlt
[Exception Handler]
[Stub Code]
// ...

```

C2 版的指令数量总体上比 C1 版的要少很多，但是主要的算法逻辑并没有精简，依然由 6 条机器码指令完成，如下：

```

0x000000010eaa6c4c: mov    %esi,%eax
0x000000010eaa6c4e: sub    %edx,%eax
0x000000010eaa6c50: add    $0x211,%eax
0x000000010eaa6c56: imul   %esi,%eax
0x000000010eaa6c59: add    $0xfffffffffffffff,%eax
0x000000010eaa6c5c: imul   %edx,%eax

```

这 6 条指令也用于完成 Test.calc(int, int)方法的算法逻辑，但是这 6 条指令与上面 C1 版的 6 条指令有所不同，调整了部分顺序，并且使用了补码操作代替减法运算，可以算作部分优化。但是由于 C2 版的机器指令数量远远少于 C1 版的机器指令数量，因此 C2 编译器的编译质量更高。但是无论是 C1 编译器还是 C2 编译器，其编译后的本地机器码整体质量大多（不能绝对）

比直接使用 JVM 内置的模板解释器所生成的本地机器码的质量要高很多，而影响 JVM 内置的模板解释器效率的一个核心因素便是字节码指令的分发，字节码指令的分发需要对应的 jmp 机器指令才能完成，有几条字节码指令就会有几次 jmp，这种跳转一方面使得模板解释器生成的本地机器码数量增多，另一方面也降低了执行效率。

使用 javap 命令输出 Test.calc(int, int)方法所对应的字节码指令如下：

```
public static int tt(int, int);
  descriptor: (II)I
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=4, args_size=2
      0: sipush      529
      3: istore_2
      4: iload_0
      5: iload_2
      6: iadd
      7: istore_3
      8: iload_3
      9: iload_1
     10: isub
     11: istore_3
     12: iload_3
     13: iload_0
     14: imul
     15: istore_2
     16: iload_1
     17: iload_2
     18: iconst_5
     19: isub
     20: imul
     21: istore_3
     22: iload_3
     23: ireturn
```

通过输出结果可知，Test.calc(int, int)方法一共对应 24 条字节码指令，假设每条字节码指令仅对应 4 条本地机器码（4 条机器码指令是最少的了，仅仅 dispather_next 分发就需要占用 3 条机器码），那么 24 条字节码指令至少也会生成 96 条本地机器码，这种级别的数量相比于上面 C1 和 C2 编译器所生成的本地机器码数量而言，无疑是巨大的。由此可见，C1 和 C2 这种分层自适应编译器所带来的性能提升当真不是说着玩的，而是真刀真枪的，能够将性能提升几个数量级。

除了这些优化技术，HotSpot 还使用一些内存分配、并发控制方面的优化技术，其中比较突出的是“逃逸分析”。所谓“逃逸分析”是指当一个 Java 对象被定义后，可能会被外部方法引用，例如被当作参数传递到其他方法中，这称为“方法逃逸”；也可能被其他线程访问，这个称

为“线程逃逸”。若能证明一个 Java 不会逃逸到其他方法中，则在为该对象分配内存空间时，可以直接进行“栈上分配”，即直接将 Java 对象实例分配在栈中，而非堆内存，这种分配策略所带来的一个直接好处便是不需要通过 GC 来回收对象实例，当 Java 方法调用完成，方法栈被回收时，Java 对象实例也跟着被销毁。其实这种分配方式在 C/C++ 中都有对应的实现，在 C 语言中，如果想在栈上直接分配一个结构体（姑且将 C 语言中的结构体看作一个对象，其实无论结构体还是类型，本质上都可以认为是一种复合的数据结构），可以直接通过 struct A a（假设 A 是一种自定义的结构体）这样的方式来声明；而如果想在堆中分配结构体实例，则需要通过 malloc() 函数来实现。在 C++ 中要实现栈上分配和堆上分配就更简单了，就看你创建对象实例时是否使用 new 关键字了。在 HotSpot 中实现了这种“栈上分配”技术，当确定一个 Java 类不会逃逸到其他方法中，并且该 Java 类结构比较简单（可以直接拆分成标量，也即最原始的基本类型）时，HotSpot 会将 Java 对象实例直接分配在当前线程所关联的高速缓存中，这种优化策略有一个专门的术语——标量替换。

与方法逃逸类似，如果能够证明一个 Java 对象不会逃逸到其他线程，则该对象便不会存在多线程锁的竞争，那么方法上的同步措施便会消除。很显然，消除了同步锁之后，代码的执行效率会更高。

JVM 中所使用的这些优化技术，每一个都像盛开在阳光下的鲜花，赏之不尽！

而在与 JAVA 关联紧密的另一个世界——安卓虚拟机，也实现了 JIT 技术，但是谷歌的技术大神们觉得这样仍然不够，研究出了一个 AOT (ahead of time) 技术。其实所谓 AOT，就是提前编译，或者叫作静态编译，这种技术是相对于 JIT 而言的。AOT 是在 Java 程序运行之前就提前编译好，直接编译成本地相关的机器指令。对于 JVM 而言，如果纯粹使用解释器解释执行，则每次执行一个方法都需要将字节码指令翻译成对应的机器指令，Java 方法调用几次，则这种工作便会被重复做几次；而如果开启 JIT，则每次 Java 程序重新启动运行后，都要进行一次这种编译，这种工作还是重复的。而 AOT 的思想则是，在编译阶段就把这些工作做完，直接将 Java 程序翻译成对应的本地机器指令，这样就不用在运行期重复去做这些事情了。正是由于 AOT 的出现，安卓原生的 Dalvik 虚拟机便在安卓 5 时代被抛弃了，谷歌转而使用 ART 虚拟机。所谓 ART，其实就是 AOP 的运行时环境，专门负责运行 AOT 后的指令。

上面介绍了 Java/Android 虚拟机的各种优化技术，这些都属于非常底层同时也非常高深的技术范畴，需要非常深厚的技术积累方能玩转。希望国内有兴趣的道友们共同研究，相互分享。

9.6 操作数栈在哪里

JVM 的指令集架构是面向栈的，所设计的大部分字节码指令也都是紧紧围绕栈进行操作，

例如上文提到的 iadd 指令，该指令是零地址指令，指令中并没有显式标记其所操作的源数据和目标数据究竟位于哪里，之所以没有显式标记，是因为无论是源数据还是目标数据，其实都位于栈中。讲了半天，这个栈到底在哪里呢，长啥样？大部分书籍中并没有直接的答案。其实这里所谓的栈，是 JVM 内部所实现的一个“求值栈”，这个求值栈也叫作“操作数栈”或者“表达式栈”，JVM 内部将其称为“expression stack”。其实前文讲解 JVM 的堆栈实现机制时，已经提到过表达式栈了。当 JVM 准备执行一个 Java 方法时，会先为其创建一个栈帧，前文讲过，栈帧主要包含三大部分，分别是局部变量表、固定帧和操作数栈，其详细结构如图 9.6 所示。

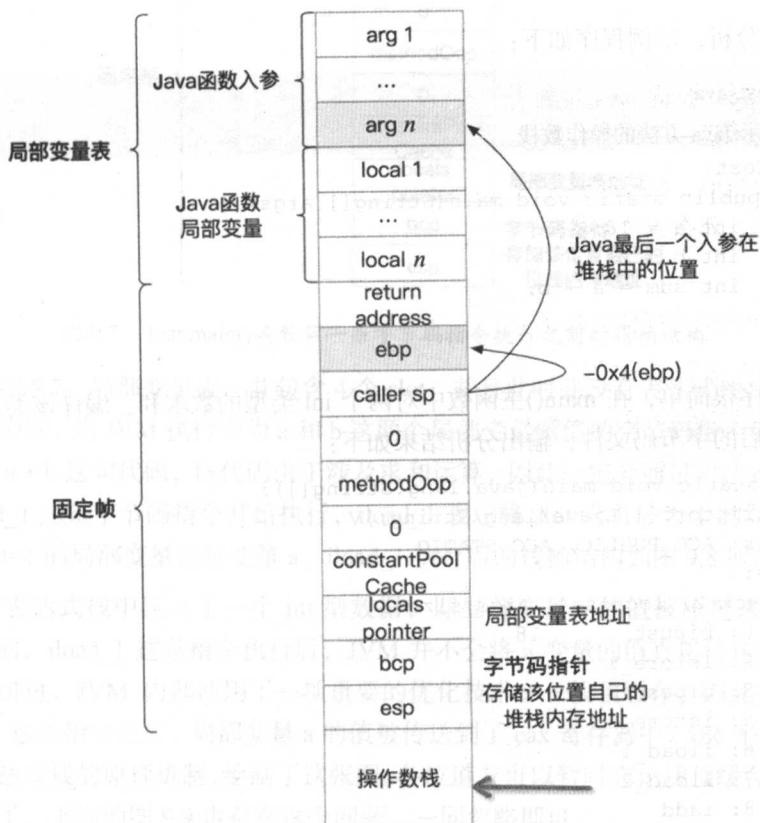


图 9.6 Java 方法栈的三大组成部分

Java 方法栈帧的局部变量表、固定帧和操作数栈按照内存从高位向低位顺序增长（x86 平台），不过在 JVM 开始执行 Java 方法的第一条字节码指令之前，操作数栈其实并没有被创建，JVM 仅仅执行到创建 Java 方法栈帧的最后一步——将当前线程栈顶位置压入当前栈顶位置，也即图 9.6 中的最底部操作数栈的上一个存储单元。由于 JVM 在创建 Java 方法栈帧时，将

methodOop、constantPoolCache、bcp（字节码指针）等压入固定帧（fixed frame）中所使用的指令都是 push，因此当执行完之后，物理机器的 esp 指针——栈顶指针，其实就指向当前线程栈的栈顶，这个位置便是图 9.6 中的最底部操作数栈的上一个存储单元。Java 方法所对应的操作数栈便从这个位置开始，因此在 JVM 内部将该位置叫作“expression stack bottom”，即表达式栈栈底。至此，JVM 便为 Java 字节码的执行准备好一切，就等着执行指令。假设 Java 方法的字节码指令中有 iload_1 这条指令，则该指令最终会被翻译成本地机器指令 push %eax，该指令会将 Java 方法栈帧的局部变量表中 slot 索引号为 1 的局部变量压入栈顶——从操作数栈底部开始压入。

下面举例分析，示例程序如下：

清单：/Test.java

功能：演示 Java 方法的操作数栈

```
class Test{  
    public static void main(String[] args){  
        int a = 18;  
        int b = 21;  
        int sum = a + b;  
    }  
}
```

该示例程序很简单，在 main() 主函数中对两个 int 类型的数求和。编译该类，并使用 javap 命令分析编译后的字节码文件，输出分析结果如下：

```
public static void main(java.lang.String[]);  
descriptor: ([Ljava/lang/String;)V  
flags: ACC_PUBLIC, ACC_STATIC  
Code:  
stack=2, locals=4, args_size=1  
0: bipush      18  
2: istore_1  
3: bipush      21  
5: istore_2  
6: iload_1  
7: iload_2  
8: iadd  
9: istore_3  
10: return
```

分析结果显示 locals=4，表示 main() 主函数的局部变量表的 slot 编号最大为 4，同时 args_size=1，表明一共只有 1 个入参。当 JVM 准备调用 main() 主函数的第一个字节码指令时，main() 方法的栈帧结果如图 9.7 所示。

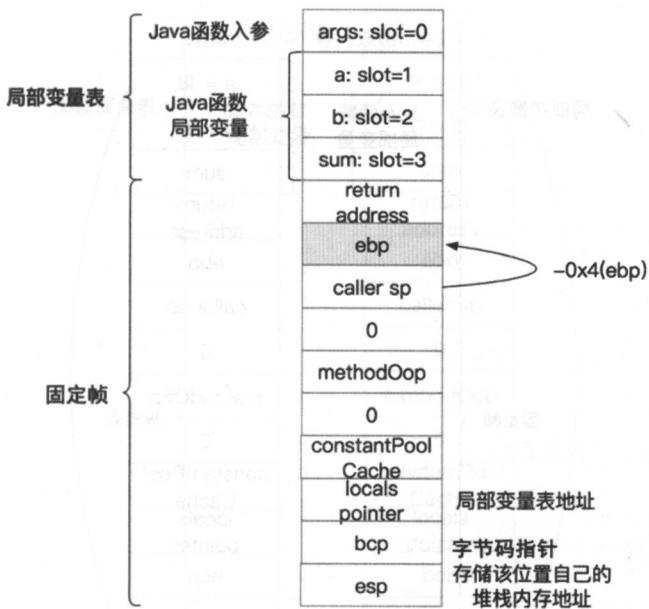


图 9.7 Test.main() 函数第一条字节码指令执行之前的栈帧结构

注意观察图 9.7，局部变量表一共包含 4 个 slot，并且此时并没有表达式栈，或者说此时的表达式栈空间为零。当 JVM 执行完为 a 和 b 这两个局部变量赋值的字节码指令后，接着便开始执行 `int sum = a + b` 这句代码，该代码由于涉及求和运算，因此一定会通过表达式栈来完成。这句代码从 `iload_1` 这条字节码指令开始执行，`iload_1` 表示将 slot 索引号为 1 的局部变量传送到表达式栈，`slot=1` 的局部变量正是变量 a，`iload_1` 执行后的栈帧结构如图 9.8 所示。

注意此时表达式栈中压入了一个 int 型数据，即局部变量 a 的值被压进来。而事实上，图 9.8 是错误的，`iload_1` 这条指令执行后，JVM 并不会将 a 变量的值直接传送到表达式栈中，别忘了前文刚讲过，JVM 内部使用了一项重要的优化技术——栈顶缓存。因此实际的情况是，执行完 `iload_1` 这条指令之后，局部变量 a 的值被传到了 `eax` 寄存器中（x86 平台）。这里仅仅是为了演示表达式栈的原理机制，绘制了这张图。各位道友可以暂时忘记栈顶缓存技术的存在，这样就不矛盾了。下面的图 9.9 也存在这个问题，一同忽略即可。

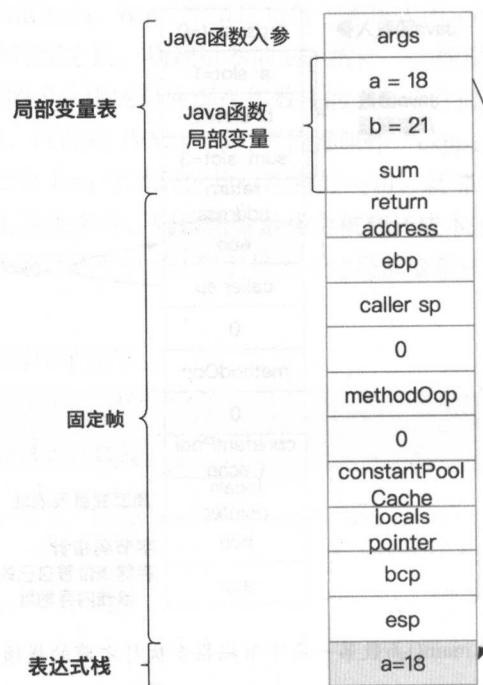
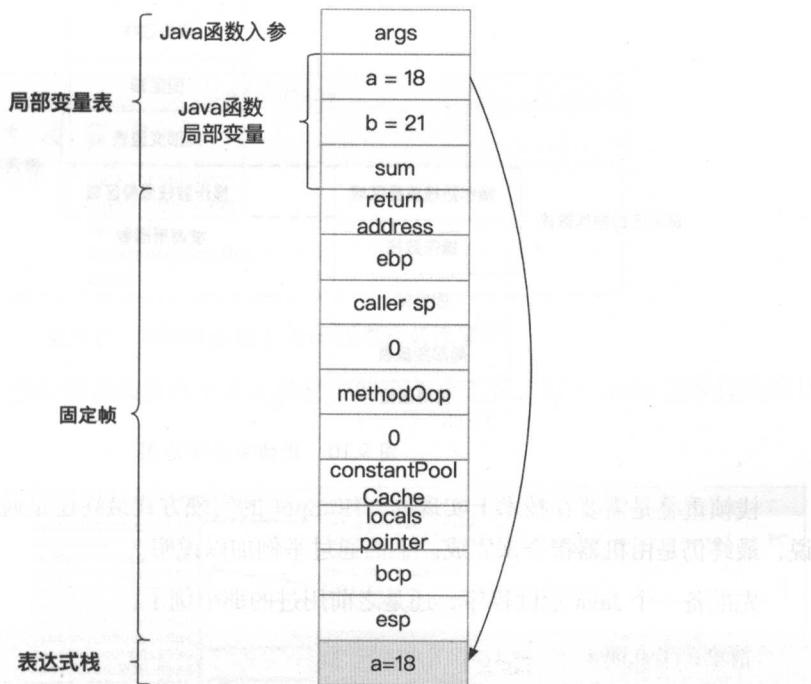


图 9.8 Test.main()函数执行完 iload_1 字节码指令之后的栈帧结构

接着 JVM 执行 `iload_2`, 该字节码指令将 slot 编号为 2 的局部变量的值加载到表达式栈中, 而对于 `Test.main()` 方法而言, `slot=2` 的局部变量是变量 `b`。该指令执行完之后, `main()` 的方法栈结构如图 9.9 所示。

至此, 表达式栈中已经压入了两个 `int` 型数据, 这就为接下来要执行的 `iadd` 指令准备好了源数据——源数据的确是位于栈顶的。

上面通过一个简单的示例演示了 Java 方法的表达式栈的创建机制, 由此可见, Java 方法的表达式栈其实就位于 Java 方法的栈帧之中。知其然, 更要知其所以然, 方为大善。在这里不禁要问: JVM 为何要将表达式栈放在 Java 方法的栈帧中? 其实答案很简单, 这种方式在技术实现上很简单。假设不这么实现, 而是将表达式栈存放在内存中其他位置, 那么 JVM 得另外维护一套求值栈管理机制了, 比较麻烦。使用这种实现方式, 当创建 Java 方法栈时, 表达式栈的底部位置便已确定; 而当 Java 方法执行完成之后方法栈被销毁时, 表达式栈也会跟着一起被销毁, JVM 无须额外设计一套复杂的机制来管理。

图 9.9 `Test.main()` 函数执行完 `iload_2` 字节码指令之后的栈帧结构

9.7 栈帧重叠

无论 JVM 的指令集是基于栈还是基于寄存器，其方法调用所基于的数据结构完全相同，都是基于堆栈。前面花了整整一个章节详细地描述了 JVM 内部方法栈帧的创建过程，JVM 在准备调用一个 Java 方法之前，会先为其创建栈帧，随着执行引擎对字节码的执行，JVM 会动态地读写 Java 方法的操作数栈。在概念模型中，存在直接调用关系的两个 Java 方法的栈帧在堆栈空间上是彼此线性串联的，并且彼此都拥有完整的栈帧结构。但是大多数虚拟机的实现都会进行一些优化，其中一项很成熟的优化技术便是栈帧重叠。所谓栈帧重叠，就是使两个相邻的栈帧出现一部分重叠，让前一个栈帧的操作数栈与后一个栈帧的局部变量表区域部分重叠在一起，这样在进行方法调用时就能共用这部分堆栈空间，并且无须进行额外的参数复制。

图 9.10 所示是栈帧重叠的示意图。



图 9.10 栈帧重叠示意图

栈帧重叠是需要在技术上实现的。HotSpot 的实现方式最终还是通过 java 字节码，往深了说，最终仍是由机器指令来完成。下面通过举例加以说明。

先准备一个 Java 示例程序，还是之前用过的那个例子：

清单：/Test.java

功能：示例程序

```
public class Test{

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

该示例很简单，在主函数 main()里面调用 Test 对象实例的 add()方法。

仅仅讲述理论未免略显枯燥，还是实际练练手来得更加生动。对于堆栈重叠技术，HSDB 依然带着神器的光环，它能够带领各位道友一起领略传说中的堆栈重叠的风姿。

使用 JDB 启动 Test 程序，并在 add()方法的第 2 行上打上断点，然后就让程序一直处于暂停状态。接着使用 JPS 查看 Test 进程的进程号，使用 HSDB 连接上这个进程，此时 HSDB 的主

窗口如图 9.11 所示。

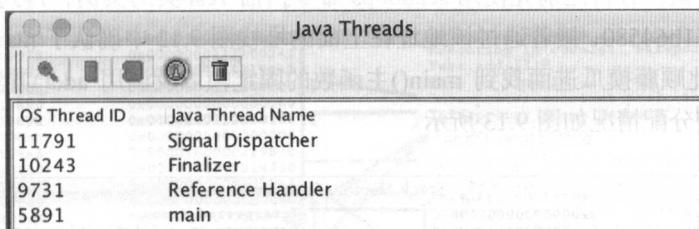


图 9.11 HSDB 连接上 Java 进程后的主窗口

选中 main 主线程，然后单击该窗口上方工具栏中的第 2 个工具，打开 main 主线程的堆栈窗口，如图 9.12 所示。

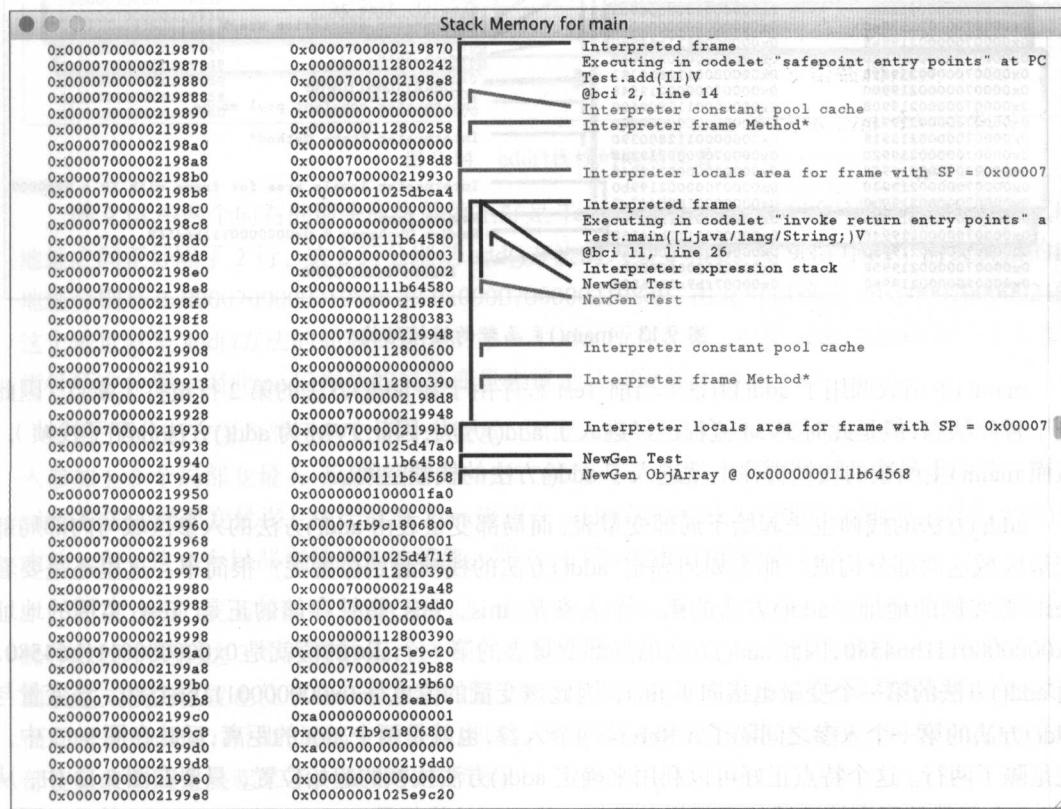


图 9.12 HSDB 显示堆栈详细内容

前面在使用 HSDB 演示 Java 方法栈的一节中其实使用的也是这个例子，并且当时分析了 main()主函数的栈帧。分析之前先使用 scanoops 命令扫描 Test 类的实例，得到 Test 类实例的地址为 0x0000000111b64580，接着通过该地址在上面的堆栈图 9.12 中确认了 main()主函数的栈帧起始位置，并据此顺藤摸瓜进而找到 main()主函数的固定帧以及调用 add()方法时的操作数栈，main()方法的栈帧分配情况如图 9.13 所示。

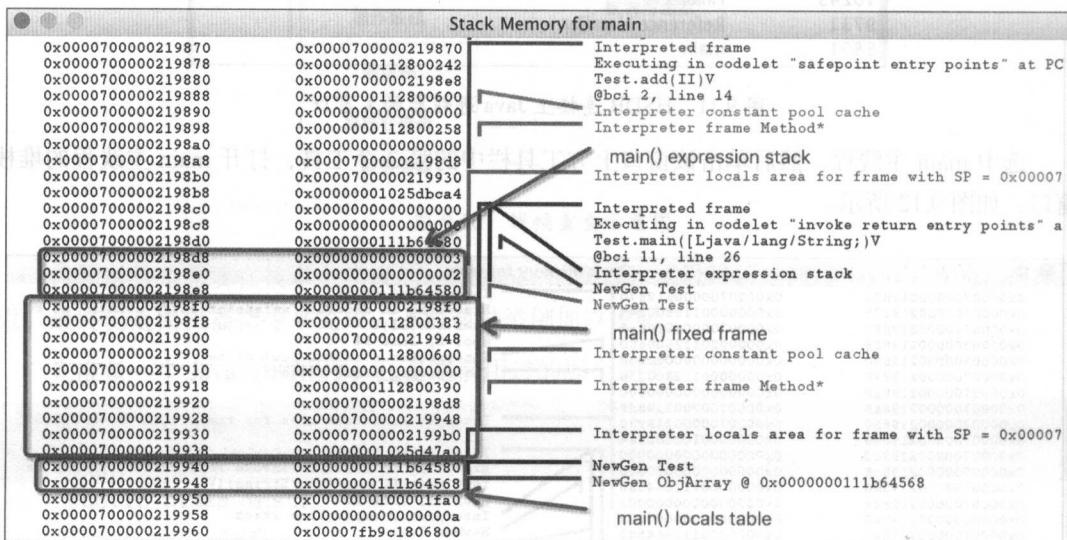


图 9.13 main()主函数的栈帧结构

main()主函数调用了 add()方法（当前 Test 程序由于在 add()方法的第 2 行被打上断点，因此处于暂停状态，但是此时 JVM 流程已经进入了 add()方法，因此 JVM 为 add()方法分配了栈帧），从而 main()主函数的栈帧再往上就进入了 add()方法的栈帧空间。

add()方法的栈帧也是起始于局部变量表，而局部变量表由 add()方法的入参区域与内部局部变量区域这两部分构成。那么如何确定 add()方法的栈帧起始位置呢？很简单，这里还是要看 Test 类实例的地址。add()方法的第一个入参是 this，this 指针存储的正是 Test 实例的地址 0x0000000111b64580，因此 add()方法的局部变量表的第一个 slot 的值就是 0x0000000111b64580。而 add()方法的第一个变量也指向了 this，因此该变量的值也是 0x0000000111b64580。该变量与 add()方法的第一个入参之间隔了 a 和 b 这两个入参，也就是两个 slot 的距离，反映在图 9.13 中，就是隔了两行。这个特点正好可以利用来确定 add()方法栈帧的起始位置，只要在图 9.13 中，从 main()主函数的固定帧区域的顶部往上寻找，找到两处值都是 0x0000000111b64580 并且又隔了两行的地方，就是 add()方法的栈帧的起始位置。从图 9.13 中能够很容易就找到这个位置，如

图 9.14 所示。

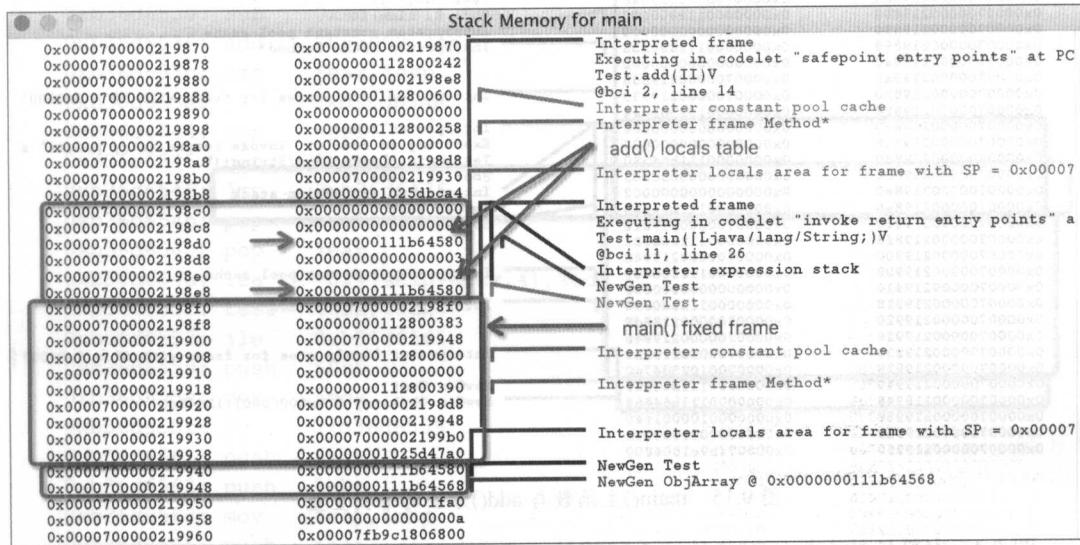


图 9.14 add()栈帧的起始位置

图 9.14 中两个向右的箭头所指的值正好是 Test 实例地址 0x00000000111b64580，并且这两个地址之间正好隔了 2 行，这 2 行分别是 add()方法的入参 a 和 b。图 9.14 中两个箭头所指的内存地址分别是 0x00007000002198e8 和 0x00007000002198d0。由此可以确定，0x00007000002198e8 这个地址就是 add()方法的局部变量表的第一个 slot 所在位置，也是 add()方法的第一个入参 this 指针所在位置，因此 add()方法的局部变量表就是从该位置开始。

由于 add()方法内部还定义了 3 个变量，因此 add()方法的局部变量表一共有 6 个成员（3 个入参数上 3 个局部变量），反映在图 9.14 中，就是占了 6 行，因此图 9.14 中最顶部的方框就是 add()方法的局部变量表。由于此时 Test 进程在 add()方法的第 2 行被打上断点，因此 add()方法内的 z 和 x 这 2 个局部变量尚未被赋值，图 9.14 显示它们的值皆为 0。

而在 main()主函数调用 add()方法时，由于 add()方法有 3 个入参，因此 main()主函数需要向操作数栈中复制这 3 个入参的值，这 3 个入参共同组成了运行时的操作数栈（亦称表达式栈），而表达式栈的位置也位于 main()方法的固定帧的上面。而现在，main()主函数栈帧的固定帧的上面同时也是 add()方法的局部变量表的区域，因此 main()方法的表达式栈空间与 add()方法的局部变量表空间重叠起来了，如图 9.15 所示。

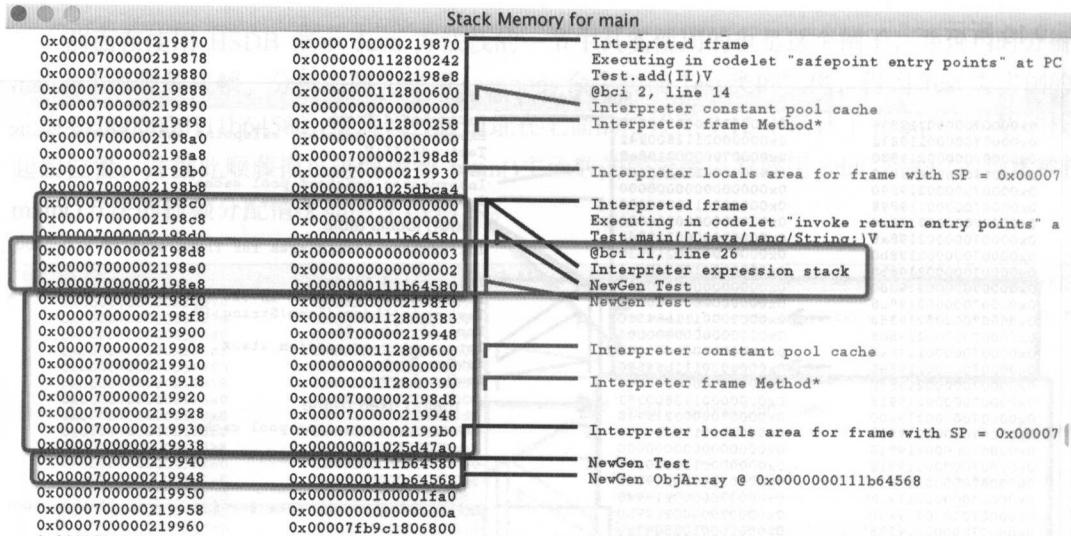


图 9.15 main()主函数与 add()方法的堆栈重叠

图 9.15 中最长的方框所框住的这 3 行就是 main()主函数与 add()方法的栈帧重叠的部分。

由此可以知道，栈帧重叠，所重叠的部分也只是操作数栈部分，不涉及被调用的方法内部的局部变量区域。并且重叠的空间大小，随着当前方法所调用的方法的入参区域大小的不同而不同，并没有固定的大小。各位道友可以自行在自己的电脑上使用 HSDB 来不断实验，进一步加深对栈帧重叠的机制的理解。

9.8 entry_point 例程机器指令

如下：

```
method entry point (kind = zerolocals) [0xb36d6500, 0xb36d6600] 256 bytes
[Disassembling for mach='i386']
0xb36d6500: movzwl 0x26(%ebx),%ecx
0xb36d6504: movzwl 0x24(%ebx),%edx
0xb36d6508: sub    %ecx,%edx
0xb36d650a: cmp    $0x3f6,%edx
0xb36d6510: jbe    0xb36d654c
0xb36d6516: push   %esi
0xb36d6517: mov    %esp,%esi
0xb36d6519: shr    $0xc,%esi
0xb36d651c: mov    -0x48f07d20(%esi,4),%esi
```

```

0xb36d6523: lea    0x28(%edx,4),%eax
0xb36d652a: add    0xa8(%esi),%eax
0xb36d6530: sub    0xac(%esi),%eax
0xb36d6536: add    $0x3000,%eax
0xb36d653c: cmp    %eax,%esp
0xb36d653e: ja     0xb36d654b
0xb36d6544: pop    %esi
0xb36d6545: pop    %eax
0xb36d6546: jmp    0xb36d6476
0xb36d654b: pop    %esi
0xb36d654c: pop    %eax
0xb36d654d: lea    -0x4(%esp,%ecx,4),%edi
0xb36d6551: test   %edx,%edx
0xb36d6553: jle    0xb36d6561
0xb36d6559: push   $0x0
0xb36d655e: dec    %edx
0xb36d655f: jg     0xb36d6559
0xb36d6561: push   %eax
0xb36d6562: push   %ebp
0xb36d6563: mov    %esp,%ebp
0xb36d6565: push   %esi
0xb36d6566: push   $0x0
0xb36d656b: mov    0x8(%ebx),%esi
0xb36d656e: lea    0x30(%esi),%esi
0xb36d6571: push   %ebx
0xb36d6572: mov    0x10(%ebx),%edx
0xb36d6575: test   %edx,%edx
0xb36d6577: je     0xb36d6580
0xb36d657d: add    $0x58,%edx
0xb36d6580: push   %edx
0xb36d6581: mov    0xc(%ebx),%edx
0xb36d6584: mov    0xc(%edx),%edx
0xb36d6587: push   %edx
0xb36d6588: push   %edi
0xb36d6589: push   %esi
0xb36d658a: push   $0x0
0xb36d658f: mov    %esp,(%esp)
0xb36d6592: mov    %esp,%eax
0xb36d6594: shr    $0xc,%eax
0xb36d6597: mov    -0x48f07d20(%eax,4),%eax
0xb36d659e: movb   $0x1,0x175(%eax)
0xb36d65a5: mov    %eax,-0x1000(%esp)
0xb36d65ac: mov    %eax,-0x2000(%esp)
0xb36d65b3: mov    %eax,-0x3000(%esp)
0xb36d65ba: mov    %esp,%eax
0xb36d65bc: shr    $0xc,%eax
0xb36d65bf: mov    -0x48f07d20(%eax,4),%eax

```

```
0xb36d65c6: movb    $0x0,0x175(%eax)
0xb36d65cd: cmpb    $0x0,0xb70cc4dd
0xb36d65d4: je      0xb36d65f3
0xb36d65da: mov     %esp,%ecx
0xb36d65dc: shr     $0xc,%ecx
0xb36d65df: mov     -0x48f07d20(%ecx,4),%ecx
0xb36d65e6: mov     -0xc(%ebp),%ebx
0xb36d65e9: push    %ebx
0xb36d65ea: push    %ecx
0xb36d65eb: call    0xb6f330d0
0xb36d65f0: add     $0x8,%esp
0xb36d65f3: movzbl  (%esi),%ebx
0xb36d65f6: jmp    *-0x48f0f2a0(%ebx,4)
0xb36d65fd: xchg    %ax,%ax
```

9.9 执行引擎实战

JVM 的执行引擎的原理基本分析完了，机智的小伙伴们早就看透了一切！

不过机智是需要表现出来的，如果真的有的话。

而实战是霸气外露的最好机会。阅读 HotSpot 源代码绝不亚于进行一场战争，这场战争的敌军是密密麻麻的源代码。可是，有一块源码在 HotSpot 的源文件中是无法直接看到的，那就是一个 Java 程序最终所生成的本地机器指令，这些指令是 JVM 在运行时动态组装拼接出来的。

这就带来一个很严重的问题，都快要上战场了，结果竟然连敌军在哪里，是谁都不知道，这可是要命的。

本来想在战场上好好秀一秀自己的肌肉，期望在战场上能够打出我军的气势，找出敌军的战略意图，结果找不着敌军，别说霸气侧漏了，根本就露不出来，这仗没法打，这日子没法过啦！

为了让小伙伴们见识一下真正的敌军，笔者也是操碎了心，精心为大家准备了一个示例，为我军引路，找到真正的敌军。

由于在 Java 程序的运行期所动态组装出来的都是本地机器指令，因此如果对汇编实在没兴趣，可以跳过本章，跳过本章并不影响你对 JVM 执行引擎的理解。

9.9.1 一个简单的例子

下面的这个例子十分简单，简单到大家伙儿都瞧不上眼：

清单: /test/A.java

作用: 一个简单的 Java 程序

```
class A{
    public static void doSomeThing(){
        int a = 3;
        int b = 81;
        int c = a + b - 9;
    }
}
```

使用 javap 命令打印这段程序的常量池和方法字节码，得到如下信息（仅摘录主要信息）：

清单: ./A.java

作用: 打印常量池和字节码

```
Classfile /test/A.class
  class A
    minor version: 0
    major version: 52
    flags: ACC_SUPER
  Constant pool:
    #1 = Methodref      #3.#11    // java/lang/Object."<init>":()V
    #2 = Class          #12      // A
    // ...

  {
    public static void doSomeThing();
    descriptor: ()V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=3, args_size=0
      0: iconst_3
      1: istore_0
      2: bipush     81
      4: istore_1
      5: iload_0
      6: iload_1
      7: iadd
      8: bipush     9
      10: isub
      11: istore_2
      12: return
  }
```

根据 javap 命令所打印出的 doSomeThing()方法的信息可知，doSomeThing()方法的操作数栈最大深度为 2 (stack=2)，局部变量表大小为 3 (locals=3)。同时，该方法经过编译后，一共有

12 条字节码指令。

下面先分析这 12 条字节码指令的运行过程。

9.9.2 字节码运行过程分析

`doSomething()`方法开始运行之前，JVM 便已经分配好局部变量表和操作数栈（也叫表达式栈，expression stack），具体创建的过程在前文讲解栈帧的章节详细描述过，此处略过不表。对于本例，`doSomething()`方法的栈帧准备之后的局部变量表与操作数栈内存布局如图 9.16 所示。

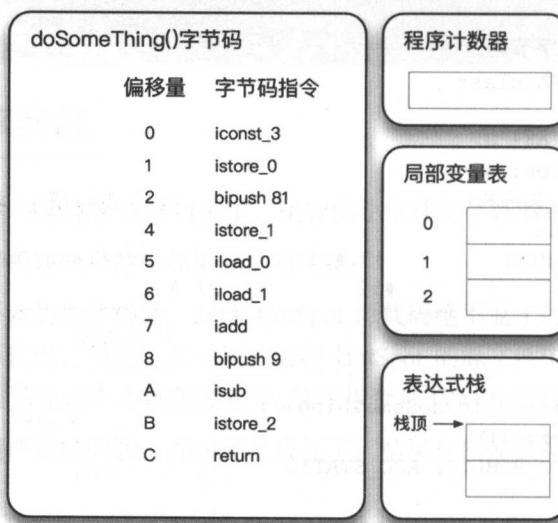


图 9.16 初始堆栈布局

图 9.16 左侧显示字节码，随着程序运行，会标示当前运行到的字节码指令。右侧则显示程序运行过程中的局部变量表和表达式栈的内存数据，同时标注当前程序计数器。

值得注意的是，图 9.16 的左侧分成两列，第一列是当前字节码相对于基址的偏移量，第二列则是具体的字节码。其中第 4 个字节码指令 `istore_1` 的偏移量是 4，而不是 3，这是因为第 3 条字节码指令 `bipush 81` 占用了 2 字节的宽度，`bipush` 占用 1 个，立即数 81 也占用 1 个。同理，第 9 条字节码指令 `isub` 的偏移量是 10，而不是 9，也是因为其上一条指令 `bipush 9` 占用了 2 字节宽度。

所谓的程序计数器，其实上文讲过，就是指示当前字节码指令相对于 Java 方法的字节码区域的起始位置的偏移量所在的堆栈内存位置。对于 x86 的 32 位平台而言，使用 esi 寄存器保存

当前字节码指令的内存地址，当前字节码指令运行结束之后，由当前字节码指令自己增加 `esi` 的值，从而将 `esi` 指向下一条字节码指令的内存地址。这一点与 CPU 硬件层面的程序计数器的工作原理类似，只不过 CPU 是纯数字电路驱动，不需要软件程序自己实现计数的逻辑，而 JVM 则由软件逻辑进行控制，但是基于 JVM 这一虚拟机器的上层的 Java 应用程序与基于物理 CPU 之上的软件程序一样，也不需要感知代码指令的偏移。

虽然程序计数器所代表的硬件寄存器中实际所保存的是目标字节码指令的内存地址，但是为了理解简单，大家都不约而同地将其简单理解为指向下一条指令相对于第一条指令的偏移位置，本书也采用这种简化的理解方法，即程序计数器指向的是指令的相对偏移位置。

另外，在 32 位机器上，图 9.16 中局部变量表和表达式栈的每一个小方框代表 4 字节、32 位比特的内存存储单元，前面也分析过，在 32 位平台上，JVM 的一个 slot 槽位正好占据 32 位存储空间。

1. 执行 `iconst_3` 指令

`iconst_3` 指令的作用是将操作数 3 推送至栈顶（表达式栈的栈顶）。执行之后的内存布局如图 9.17 所示。

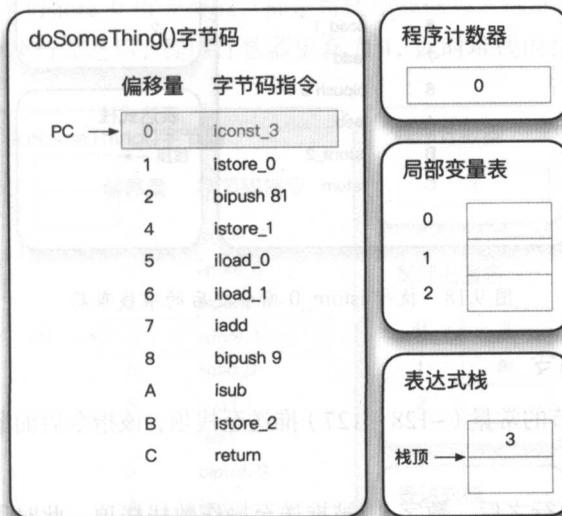


图 9.17 执行 `iconst_3` 指令之后的堆栈布局

此时操作数栈栈顶的值变成 3。根据 JVM 解释器的运行机制，JVM 会先执行字节码指令所对应的逻辑，执行完之后才会将程序计数器更新为下一条字节码指令所在的位置，但是在更新

之前，程序计数器仍然指向当前刚刚执行完的字节码指令。下面为了描述方便，会统一表述为：当执行完当前字节码指令时，程序计数器指向当前字节码指令所在的位置，请细节控不要纠结语言表述上的逻辑问题。

2. 执行 istore_0 指令

istore 指令将表达式栈栈顶的数据弹出来，并传送至局部变量表中指定的位置，该位置由紧跟在 istore 指令后面的数字指定。

当 istore_0 指令执行之后，数字 3 从表达式栈栈顶被弹出，并保存到局部变量表的第 0 个槽位（slot），此时程序计数器的值更新为 1，堆栈内存布局如图 9.18 所示。

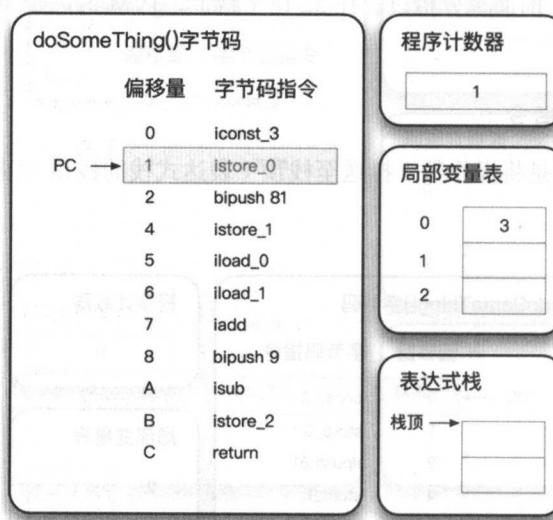


图 9.18 执行 istore_0 指令之后的堆栈布局

3. 执行 bipush 指令

bipush 指令将单字节的常量（-128 ~ 127）推送至栈顶，该指令后面紧跟一个单字节的操作数。

当 bipush 81 指令执行之后，数字 81 被推送至操作数栈栈顶，此时程序计数器的值更新为 2，堆栈内存布局如图 9.19 所示。

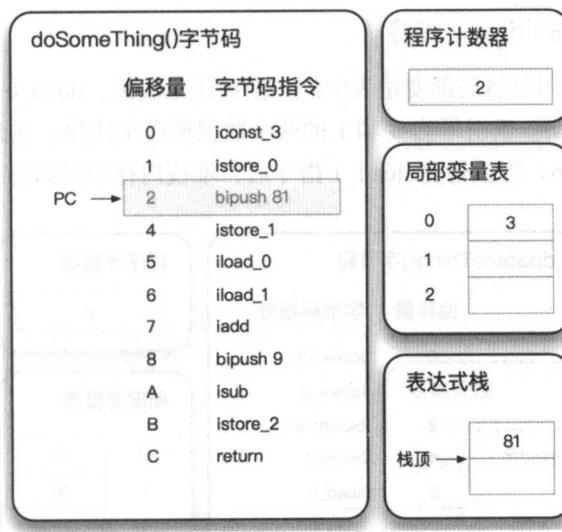


图 9.19 执行 bipush 81 指令之后的堆栈布局

4. 执行 istore_1 指令

istore_1 指令与前面的 istore_0 指令类似，都是将栈顶数据写入局部变量表，不过写入的位置是第 1 个 slot。本指令执行完之后，程序计数器更新为 4，此时堆栈内存布局如图 9.20 所示。

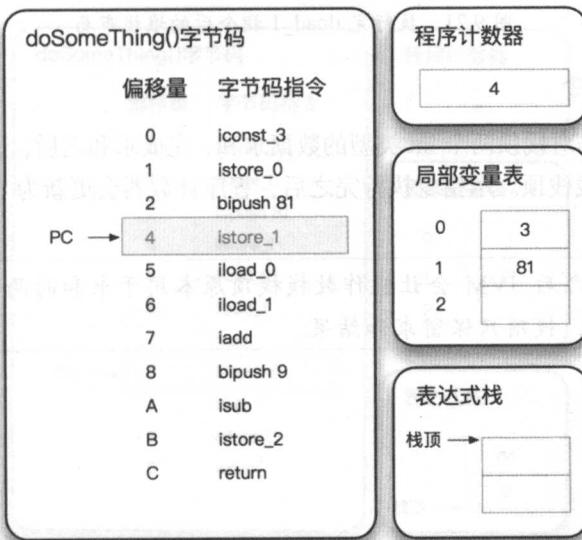


图 9.20 执行 istore_1 指令之后的堆栈布局

5. 执行 iload_0 与 iload_1 指令

iload 系列指令的作用是将局部变量表中的数据推送至栈顶，iload_0 与 iload_1 指令的作用分别是将局部变量表中 slot 索引号为 0 和 1 的两个数据推送至栈顶。在此过程中，程序计数器的值会分别更新为 5 和 6，当执行完 iload_1 指令后，堆栈内存布局如图 9.21 所示。

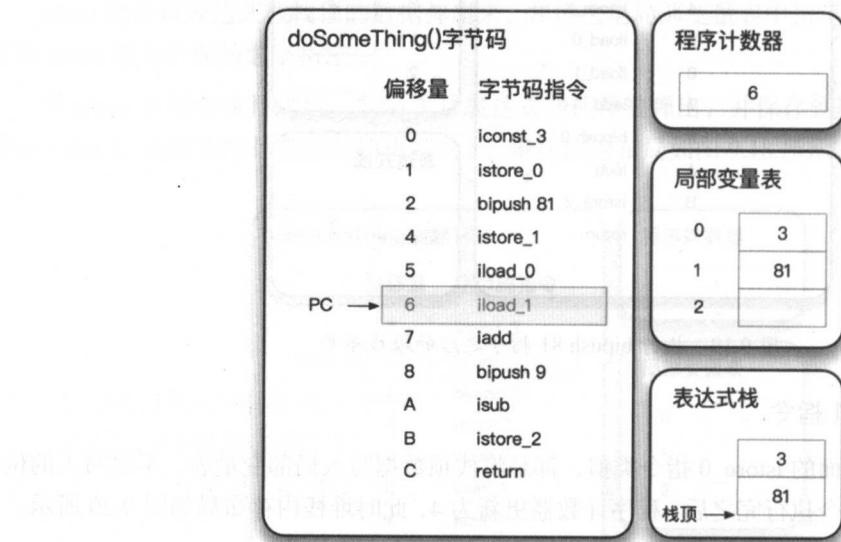
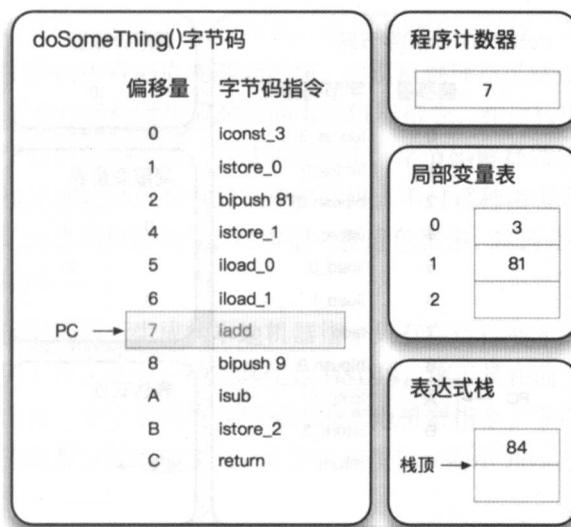


图 9.21 执行完 iload_1 指令后的堆栈布局

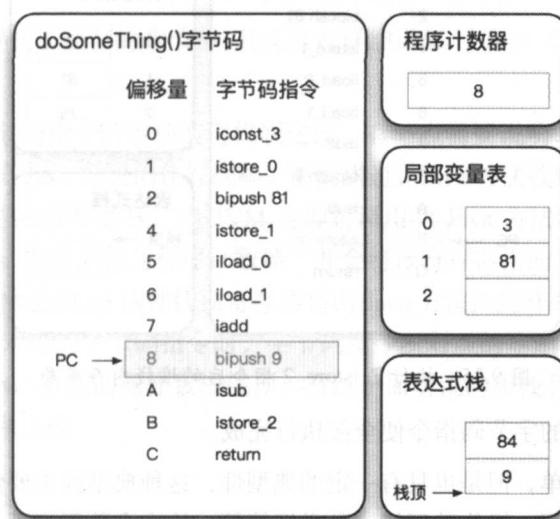
6. 执行 iadd 指令

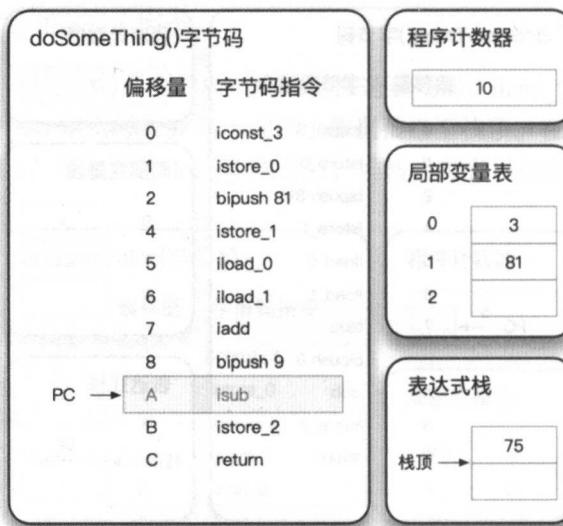
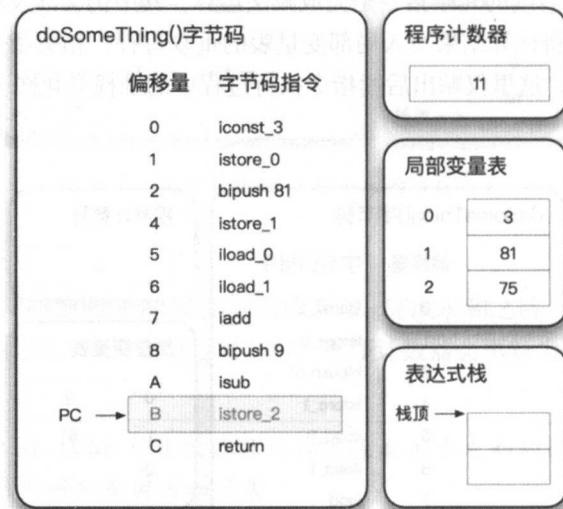
iadd 指令的作用是对栈顶两个 int 类型的数据求和，完成求和之后，让栈顶元素出栈，并将累加结果压入操作数栈栈顶。本指令执行完之后，程序计数器会更新为 7，同时堆栈内存布局如图 9.22 所示。

注意，求和之后 JVM 会让操作数栈栈顶原本用于求和的两个数据出栈，所以完成求和之后，栈顶只保留求和结果。

图 9.22 执行完 `iadd` 指令后的堆栈内存布局

完成 `iadd` 指令后，后续的几条指令是完成减法运算，其中仍会涉及将数据推送至栈顶，执行减法运算，再从栈顶将计算结果写入局部变量表的重复过程，相关指令的流程机制与前面相同，具体过程不再赘述，这里仅贴出后续指令执行过程中的堆栈变化图（如图 9.23、图 9.24 及图 9.25 所示）。

图 9.23 执行完 `bipush 9` 指令后的堆栈内存布局

图 9.24 执行完 `isub` 指令后的堆栈内存布局图 9.25 执行完 `istore_2` 指令后的堆栈内存布局

至此，本示例程序的字节码指令便全部执行完成。

本示例程序虽然简单，但是也具有一定的典型性，这种典型性主要体现在字节码指令上，例如，局部变量表的读写、操作数压栈、数学运算等，绝大多数程序逻辑都离不开这几种最常用的字节码指令。在 JVM 内部，每一种解释器（例如字节码解释器、模板解释器）都给出了

Java 字节码指令对应的实现，或用 C 语言写成，或用机器指令完成。其中字节码解释器的实现最为直观，基本能够看懂 C 语言的道友都能看懂；而模板解释器虽然也使用 C 语言解释字节码逻辑，但是 C 语言仅负责在运行时生成对应的本地机器指令，在运行期实际上是通过动态生成的机器指令去完成字节码指令的逻辑。不过话说回来，字节码解释器使用 C 语言直接解释字节码，虽然编译时会被编译器解释成对应的本地机器码，不过这种由编译器生成的机器指令相比于模板解释器中由人工生成的机器指令，在质量上要逊色很多，这便是字节码解释器从 JVM 很早的版本便被弃用的原因。

不过正是由于模板解释器所生成的本地机器指令只有在运行期才能看到，在编译期无法直接看到，因此必须使用工具，其中一种工具便是 HSDIS，该工具在前文多次提到过。使用该工具能够在运行期打印出每一个字节码指令所对应的本地机器指令，下面便基于本工具，打印本示例程序中常见的字节码指令的本地实现，并分析其逻辑，揭示 Java 字节码指令内部实现的真正原理。

9.10 字节码指令实现

在前文讲解栈顶缓存机制时，提到模板解释器所生成的本地机器指令与栈顶缓存有很大的关系，几乎大部分字节码指令的本地实现都使用了栈顶缓存这种优化技术。这里将要讲解的 `iconst_3`、`istore_0` 和 `iadd` 等指令也都使用了栈顶缓存技术。

同时，在前面讲解栈顶缓存的章节中，其实已经详细讲解了 `iload` 系列指令的本地实现机制，因此这里便不再赘述。

另外，在描述字节码指令的本地实现机制之前，有一点需要说明，当 JVM 开始调用一个 Java 方法之前，会为该 Java 方法创建好栈帧，前面讲过，Java 方法栈帧主要包含 3 大块，分别是局部变量表、固定帧和操作数栈。当 JVM 为即将调用的 Java 方法准备好栈帧之后，在 x86 平台上，JVM 会以 `esi` 寄存器作为程序计数器，并会将该寄存器指向局部变量表的第一个 slot 的内存位置，同时 JVM 会将 `sp` 这种栈顶寄存器指向 Java 方法的操作数栈栈顶，因此，在 JVM 执行目标 Java 方法所对应的字节码指令时，字节码指令所对应的本地机器码指令 `push` 与 `pop`，实际上便是在操作 Java 方法的操作数栈栈顶，所以下面所说的压栈和出栈实际上是指对 Java 方法操作数栈的压栈和出栈。

9.10.1 `iconst_3`

在 32 位 x86 平台上，使用 HSDIS 工具打印该指令对应的本地机器逻辑如下：

```
iconst_3 6 iconst_3 [0xb36d7ce0, 0xb36d7d20] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d7ce0: sub    $0x4,%esp
0xb36d7ce3: fstps  (%esp)
0xb36d7ce6: jmp    0xb36d7d04
0xb36d7ceb: sub    $0x8,%esp
0xb36d7cee: fstpl  (%esp)
0xb36d7cff1: jmp    0xb36d7d04
0xb36d7cff6: push   %edx
0xb36d7cff7: push   %eax
0xb36d7cff8: jmp    0xb36d7d04
0xb36d7cffd: push   %eax
0xb36d7cffe: jmp    0xb36d7d04
0xb36d7d03: push   %eax ----->栈顶缓存入口点
0xb36d7d04: mov    $0x3,%eax ----->无缓存入口点
0xb36d7d09: movzbl 0x1(%esi),%ebx ----->取指逻辑开始
0xb36d7d0d: inc    %esi
0xb36d7d0e: jmp    *-0x48f106a0(%ebx,4)
0xb36d7d15: xchg   %ax,%ax
0xb36d7d18: add    %al,(%eax)
0xb36d7d1a: add    %al,(%eax)
0xb36d7d1c: add    %al,(%eax)
0xb36d7d1e: add    %al,(%eax)
```

`iconst_3` 指令对应的本地机器码貌似很多，但是对于本示例程序（即 9.9 节“执行引擎实战”中的简单 Java 测试程序，下同）而言，由于 `iconst_3` 指令是方法的第一条字节码指令，在其之前并没有其他字节码指令会向栈顶缓存数据，因此此时栈顶状态为空，所以 `iconst_3` 指令便从上面这段本地机器指令的“无缓存入口点”进入，即下面这条机器指令：

```
mov $0x3, %eax
```

该指令驱动物理 CPU 将操作数 3 传送到 `eax` 寄存器中。JVM 完成这条传送指令之后，便直接开始取指，准备执行下一条字节码指令。这中间似乎存在一个问题，不是说好的，`iconst_3` 指令会将数字 3 压入操作数栈栈顶的吗，为何这里仅仅看到机器指令将其传送到寄存器中，反倒没栈顶啥事儿了？答案很简单，这里仍然在履行栈顶缓存策略，`iconst_3` 指令并没有真的将数据入栈，而是先临时存放在 `eax` 这个缓冲器中。

那么 `iconst_3` 指令到底啥时候才会将数字 3 入栈呢？这需要看具体的场景，具体来说需要看其后面的那条字节码指令是啥，如果其后面的那条指令仍然是进行压栈操作，例如又来一条

iconst 系列的指令，或者来一条 iload 系列的指令，或者 sipush、bipush、ldc 等指令，由于这些指令也会使用栈顶缓存策略，而对于一个给定的 CPU 硬件平台，栈顶缓存只能有一个寄存器，所以 JVM 为了给这些后续的指令腾出栈顶缓存空间，只能对 iconst_3 指令中所隐含的操作数 3 执行真正压栈操作。但是如果 iconst_3 指令后面的那条字节码指令不是压栈操作，而是运算指令，例如 iadd、isub 等，或者是写局部变量表操作，例如 istore 系列的指令等，则 JVM 永远不会对 iconst_3 执行真正的压栈操作，数字 3 最多只能到达用于栈顶缓存的寄存器之中，而不会被传送到栈顶，这是为了减少几次内存读写操作，从而提升运算速度。

对于本示例程序，由于 iconst_3 指令后面的那条指令是 istore_0，该指令不会继续进行压栈操作，因此实际上 JVM 在运行本测试程序时，并不会将数字 3 压入栈顶，从这个角度而言，其实上一节所绘制的内存堆栈布局图是错误的，不过上一节的重点是分析字节码的字面含义，并不考虑栈顶缓存这种优化技术，因此基于字节码字面含义而绘制的堆栈布局图并没有问题。事实上，JVM 规范也只是定义出了一套字节码指令集，至于各种 JVM 虚拟机内部究竟如何实现，则没有相应的规范，所以通常对字节码指令的理解也只能是基于其字面含义。

9.10.2 istore_0

在 32 位 x86 平台上，使用 HSDIS 工具打印该指令对应的本地机器逻辑如下：

```
istore_0 59 istore_0 [0xb36d9240, 0xb36d9260] 32 bytes

[Disassembling for mach='i386']
0xb36d9240: pop    %eax ----->无缓存入口点
0xb36d9241: mov    %eax,(%edi) ----->栈顶缓存入口点
0xb36d9243: movzbl 0x1(%esi),%ebx ----->取指逻辑开始
0xb36d9247: inc    %esi
0xb36d9248: jmp    *-0x48f0f2a0(%ebx,4)
0xb36d924f: nop
0xb36d9250: add    %al,%eax
0xb36d9252: add    %al,%eax
0xb36d9254: add    %al,%eax
0xb36d9256: add    %al,%eax
0xb36d9258: add    %al,%eax
0xb36d925a: add    %al,%eax
0xb36d925c: add    %al,%eax
0xb36d925e: add    %al,%eax
```

本示例程序所对应的第二条字节码指令是 istore_0，该指令的字面含义是将栈顶数据写入局部变量表中索引号为 0 的 slot 槽位中。

同样，本字节码指令依然使用了栈顶缓存技术，其对应的第一条机器指令是 pop %eax，这

条机器指令的含义是将栈顶数据弹出至 eax 寄存器中。

在 JVM 执行字节码跳转时，会判断栈顶缓存状态，当栈顶缓存为空时，则会执行 pop 系列的机器指令以将栈顶数据弹出至用于缓存的寄存器之中。这是一个通用逻辑。在 x86 平台上，用作栈顶缓存的寄存器是 eax，在 JVM 执行 istore 系列的字节码指令之前，如果 eax 寄存器中已经有数据，则 JVM 不再执行 pop %eax 这条机器指令，而是直接从该指令的下一条指令——mov %eax,(%edi)开始执行。这条指令将 eax 寄存器中的数据传送到(%edi)所指向的内存位置，前面讲过，edi 寄存器指向 Java 方法栈帧的局部变量表第 0 个 slot 位置，因此这条指令实际上在解释执行 Java 字节码指令 istore 的字面含义——将栈顶数据写入局部变量表。

9.10.3 iadd

在 32 位 x86 平台上，使用 HSDIS 工具打印该指令对应的本地机器逻辑如下：

```
iadd 96 iadd [0xb36d9f40, 0xb36d9f60] 32 bytes
```

```
[Disassembling for mach='i386']
0xb36d9f40: pop    %eax      ----->无缓存入口点
0xb36d9f41: pop    %edx      ----->缓存入口点
0xb36d9f42: add    %edx,%eax
0xb36d9f44: movzbl 0x1(%esi),%ebx ----->取指逻辑开始
0xb36d9f48: inc    %esi
0xb36d9f49: jmp    *-0x48f106a0(%ebx,4)
0xb36d9f50: add    %al,(%eax)
0xb36d9f52: add    %al,(%eax)
0xb36d9f54: add    %al,(%eax)
0xb36d9f56: add    %al,(%eax)
0xb36d9f58: add    %al,(%eax)
0xb36d9f5a: add    %al,(%eax)
0xb36d9f5c: add    %al,(%eax)
0xb36d9f5e: add    %al,(%eax)
```

iadd 指令的作用是对 Java 方法栈栈顶的两个 int 型数据执行求和运算。由于 JVM 本身不具备数学运算的能力，最终仍然要依靠物理 CPU 才能完成，而在 x86 平台上，物理机器执行求和运算的一种方式便是直接对两个寄存器中的数据进行累加，例如：

```
add %edx, %eax
```

JVM 执行求和逻辑时，也会将 Java 方法栈栈顶的两个数据传送到 edx 和 eax 这两个寄存器中，这样才能触发 CPU 硬件的求和指令，从而完成真正的求和运算。

iadd 指令同样履行了栈顶缓存策略，如果缓存寄存器 eax 中没有数据，则会从上面第一条

机器指令 pop %eax 开始执行，将 Java 方法栈栈顶的第一个 int 型数据弹出至 eax 寄存器中，接着执行上面第二条机器指令 pop %edx，将 Java 方法栈栈顶的第二个 int 型数据弹出至 edx 寄存器中。而如果缓存寄存器 eax 中已经有数据，例如 iadd 指令的上一条指令是 iload 系列或者 iconst 系列的指令等，这些指令会将操作数缓存至 eax 寄存器中，因此在执行 iadd 指令时，会直接从上面第二条机器指令开始执行，第一条机器指令不需要执行。如此一来，原本需要连续执行两次 pop 指令才能将栈顶的两个数据弹出至寄存器中，现在只需要执行一次 pop 指令。CPU 在执行 pop 指令时需要将数据从内存传送至寄存器中，而读写内存的效率相比于读写寄存器是非常低的，因此 JVM 每节省一次内存读写，性能便能提高不少。

总体而言，JVM 虽然有一个专门的执行引擎模块，能够执行常规的若干指令，但是毕竟计算机的运算能力只能依靠硬件赋予，因此 JVM 字节码指令最终都需要转换为对应的硬件 CPU 指令。上面展示了在 x86 平台上的几种常见的字节码指令的解释原理，其他字节码指令的解释原理也都大同小异，都是基于 Java 方法操作数栈和局部变量表而翻译成对应的机器逻辑。

9.11 本章总结

JVM 最核心的技术便是执行引擎，最难的也是执行引擎，而这也是本书写作的初衷！

要想透彻理解 JVM 的执行引擎，就必须先理解物理计算机 CPU 执行运算的机制。本书详细描述了物理 CPU 进行取指、译码、运算的原理，并从这个点出发，逐步深入讲解 JVM 的执行引擎的运行机制。

相比于物理 CPU 的取指机制，JVM 的取指机制显得更加复杂。从整体效果来看，JVM 的取指机制其实糅合了物理 CPU 的取指机制和 JVM 本身的字节码取指机制。由于一个 Java 方法对应若干字节码指令，因此每当 JVM 执行完一条字节码指令后，便需要执行一次“取指”。而每一条字节码指令又对应多条机器指令，因此在 JVM 执行一条目标字节码指令时，需要同时处理本地机器指令的跳转。

JVM 在执行字节码指令时，综合使用了若干技巧，这些技巧可以节省 CPU 资源，提高程序性能。这些技巧包括栈顶缓存、堆栈重叠及 JIT 等。其中 JIT 属于非常高级的技术主题，同时也是一个永恒的话题，毕竟既想拥有 Java 简单易学的语法特性(最重要的是不需要管理内存)，又想尽可能地提升程序执行效率，不是一件容易的事。