

第 7 章

Java 栈帧

本章摘要

- ◎ entry_point 例程
- ◎ 局部变量表创建的机制
- ◎ 堆栈与栈帧的概念
- ◎ JVM 栈帧创建的详细过程
- ◎ slot 大小到底是多大
- ◎ slot 复用
- ◎ 操作数栈复用与深度

前面的章节一直在讲 JVM 解析 Java 字节码文件的原理和机制，相信很多小伙伴看得有点厌倦了，本章就换个口味，从 Java 字节码的“泥潭”中跳出来，看看别处的美好风景。

众所周知，如果 Java 程序运行出现异常，程序会打印出相应的异常堆栈，通过异常堆栈可以知道 Java 方法的调用链路。其实，调用链路是由一个个 Java 方法栈帧所组成，每一个 Java 方法都有一个栈帧，在这一点上，Java 程序与 C/C++ 程序并无任何区别。本章与各位道友一起分析 JVM 内部实现 Java 方法栈帧的机制和技术实现。

在本章开始讲解之前，让我们将目光再次聚焦到第 2 章。在这一章中，以 JVM 调用 Java 程序的 main() 主函数为例，讲解了 CallStub 例程的实现机制。在 JVM 内部，例程就是一个功能性函数。站在宏观的角度看，它就是一种预先设定好的逻辑。至于逻辑的具体实现方式，可以有很多种，从程序员的角度看，既可以用 C 语言实现，也可以用 Delphi，或者其他编程语言实现。entry_point 例程与 CallStub 例程一样，是一段使用 C 编写，最终生成一段对应的汇编的逻辑。

JVM 调用 Java 程序的 main() 主函数，会经过 CallStub 例程，但是在 CallStub 例程里仅仅完成了 Java 主函数的参数传递，并没有开始执行 Java 程序的 main() 主函数的字节码指令，这是因为 JVM 在准备执行一个 Java 方法的字节码指令之前，必须先为该方法创建好对应的方法堆栈。对于 Java 主函数而言，在 CallStub 例程里会调用 entry_point 例程，在 entry_point 例程里完成主函数的栈帧创建，找到 Java 主函数所对应的第一个字节码指令并进入执行。在 entry_point 例程里会涉及 JVM 内部的 method 对象，即 Java 方法在 JVM 内部的表达形式。关于 method 对象将会在下一章讲解。

JVM 内部可以调用各种不同的方法类型，例如 JNI 本地函数，或者 Java 里的静态方法，或者 Java 类的成员方法。调用不同种类的方法，会触发不同的 entry_point 例程，所谓 entry_point，顾名思义，就是“进入点”，进入哪里？当然是目标方法啦。正因为 JVM 在调用目标方法之前，会先经过 entry_point，并且 JVM 在执行目标方法的指令之前，需要先为其创建好相应的方法堆栈，因此 JVM 选择在 entry_point 例程中完成方法堆栈创建。本章以 Java 主函数堆栈创建为例，讲解 JVM 创建 Java 栈帧的具体实现技术。先打个预防针，本章内容难度属于中等偏上哟！需要你具备一定的汇编基础知识哟！不过没有相关基础知识的道友也不必太过于担心，毕竟大家都很聪明的！

7.1 entry_point 例程生成

与 CallStub 例程一样，entry_point 例程也是在 JVM 启动过程中被创建。事实上，JVM 内部的所有例程都随着 JVM 的启动而创建。

entry_point 例程的总体创建链路如下（基于 x86 32 位 Linux 平台）：

```

java.c: main()
java_md.c: LoadJavaVM()
jni.c: JNI_CreateJavaVM()
Threads.c: create_vm()
    init.c: init_globals()
        interpreter.cpp: interpreter_init()
        templateInterpreter.cpp: initialize()
            templateInterpreter_x86_x32.cpp: InterpreterGenerator()
                templateInterpreter.cpp: generate_all()

```

与 CallStub 的伟大“征程”一样，本链路起步于 JVM 的 main() 函数，一路走到 init_globals() 这个全局数据初始化模块，然后便与 CallStub 分道扬镳，进入 TemplateInterpreter::initialize() 流程。initialize() 函数实现如下：

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

作用：initialize()

```
void TemplateInterpreter::initialize() {
    // ...
    { 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);
        if (PrintInterpreter) print();
    }
    //...
}
```

在 initialize() 函数中执行了 **InterpreterGenerator g(_code)** 这行代码，创建解释器生成器的实例。

在 Hotspot 内部，存在 3 种解释器，分别是字节码解释器、C++ 解释器和模板解释器。字节码解释器逐条解释翻译字节码指令，由于使用 C/C++ 这种高级语言执行字节码指令逻辑，因此执行效率比较低下。

模板解释器相比于字节码解释器的高级之处在于，模板解释器将字节码指令直接翻译成了对应的机器指令，这种直接生成的机器指令相比于字节码解释器所对应的 C/C++ 代码经编译后生成的机器指令，显然要高效很多，毕竟是人力纯手工精雕细琢。

由于模板解释器更加高效，因此 JVM 默认的解释器就是模板解释器，当然可以通过启动参数指定其他解释器。

对于 C++ 解释器和模板解释器而言，都有一个对应的“解释器生成器”，模板解释器对应的生成器是 **TemplateInterpreterGenerator**。在 **TemplateInterpreter::initialize()** 函数中执行 **InterpreterGenerator g(_code)** 时，实际上是在实例化 **TemplateInterpreterGenerator** 对象。

TemplateInterpreterGenerator 对象的实例化过程，伴随着其构造函数的调用，其构造函数定义如下（基于 x86 的 32 位 Linux 平台）：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：InterpreterGenerator() 构造函数

```
InterpreterGenerator::InterpreterGenerator(StubQueue* code)
    : TemplateInterpreterGenerator(code) {
```

```

    generate_all(); // down here so it can be "virtual"
}

```

在这里调用了 generate_all() 函数。generate_all() 顾名思义，就是“产生所有”。所有啥呢？其实就是这样一款解释器运行时所需要的各种例程及入口。对于模板解释器而言，这些例程直接就是生成好的机器指令。

generate_all() 中就包含普通 Java 函数调用所对应的 entry_point 的入口，且看 generate_all() 的定义：

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

作用：generate_all() 函数

```

void TemplateInterpreterGenerator::generate_all() {
    // ...

    { CodeletMark cm(_masm, "return entry points");
        for (int i = 0; i < Interpreter::number_of_return_entries; i++) {
            Interpreter::_return_entry[i] =
                EntryPoint(
                    generate_return_entry_for(itos, i),
                    generate_return_entry_for(itos, i),
                    //...
                    generate_return_entry_for(vtos, i)
                );
        }
    }

    { CodeletMark cm(_masm, "earlyret entry points");
        Interpreter::_earlyret_entry =
            //...
    }

    { CodeletMark cm(_masm, "deoptimization entry points");
        //...
    }

    { CodeletMark cm(_masm, "result handlers for native calls");
        // ...
    }

    //...

#define method_entry(kind)
{ CodeletMark cm(_masm, "method entry point (kind = " #kind ")");
    Interpreter::_entry_table[Interpreter::kind] =
        generate_method_entry(Interpreter::kind);
}

```

```

}

// all non-native method kinds
method_entry(zerolocals)
method_entry(zerolocals_synchronized)
method_entry(empty)
method_entry(accessor)
//...

// all native method kinds (must be one contiguous block)
Interpreter::native_entry_begin = Interpreter::code() -> code_end();
method_entry(native)
method_entry(native_synchronized)
Interpreter::native_entry_end = Interpreter::code() -> code_end();

#undef method_entry

// Bytecodes
set_entry_points_for_all_bytes();
set_safepoints_for_all_bytes();
}

```

对于模板解释器而言，本方法无疑具有里程碑式的意义，它将生成模板解释器所对应的各种模板例程的机器指令，并保存入口地址。如果一个启动的 JVM 代表一个高度进化的文明社会，那么这个方法一定代表着一个国家的重大基础设施建设工程，所有的高速公路、高铁网络、地铁交通、机场与航线、港口与轮渡，等等，都会在本阶段里完工。Java 程序中的一个个对象类型如同这个社会里的一个个鲜活的人类个体，基础设施完成以后，社会里的人可以借助于高效快速的各种交通网络去完成各自的神圣使命。

generate_all() 函数的上半段定义了一些重要的逻辑入口，例如 CodeletMark cm(_masm，“return entry points”)代码段定义了 return 指令的入口，同时会生成其对应的机器指令。而从 #define method_entry(kind) 宏定义开始，则定义了一系列“方法入口”，例如，zerolocals、abstract、java_lang_math_sin 等。

在 AbstractInterpreter 中定义了 JVM 所支持的全部方法入口：

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

作用：MethodKind 枚举声明

```

enum MethodKind {
    zerolocals,
    zerolocals_synchronized,
    native,
    native_synchronized,
    empty,
}

```

```

accessor,
abstract,
method_handle,
java_lang_math_sin,
java_lang_math_cos,
java_lang_math_tan,
java_lang_math_abs,
java_lang_math_sqrt,
java_lang_math_log,
java_lang_math_log10,
java_lang_ref_reference_get,
number_of_method_entries,
invalid = -1
};

}

```

当 JVM 调用 Java 函数时，例如 Java 类的构造函数、类成员方法、静态方法、虚方法等，或者特定的数学函数，最终就会从不同的入口进去，在 CallStub 例程中进入不同的函数入口。

对于正常的 Java 方法调用（包括 Java 程序主函数），其所对应的 entry_point 一般都是 zerolocals 或者 zerolocals_synchronized，如果方法加了同步关键字 synchronized，则其 entry_point 是 zerolocals_synchronized。因此这里关注 zerolocals 方法入口，method_entry(zerolocals)生成了该方法入口。method_entry()正是在 generate_all()中定义的宏，调用 method_entry(zerolocals)就相当于执行了下面这个逻辑：

```

Interpreter::_entry_table[Interpreter::zerolocals] =
    generate_method_entry(Interpreter::zerolocalsfd)

```

这个逻辑执行完之后，JVM 会为 zerolocals 生成本地机器指令，同时将这串机器指令的首地址保存到 Interpreter::_entry_table 数组中。这与一个国家统一注册登记各地的高速入口、机场、港口、高铁站等是一样的道理。

对于 32 位 x86 Linux 平台，为 zerolocals 方法入口生成机器指令的 generate_method_entry() 函数定义如下：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：generate_method_entry()函数

```

address AbstractInterpreterGenerator::generate_method_entry(AbstractInterpreter::MethodKind kind) {
    // determine code generation flags
    bool synchronized = false;
    address entry_point = NULL;

    switch (kind) {
        case Interpreter::zerolocals : break;
        case Interpreter::zerolocals_synchronized: synchronized = true; break;
    }
}

```

```

        case Interpreter::native : entry_point =
            ((InterpreterGenerator*)this)->generate_native
        _entry(false); break;
        case Interpreter::native_synchronized : entry_point =
            ((InterpreterGenerator*)this)->generate_native
        _entry(true); break;
        //...
        case Interpreter::java_lang_ref_reference_get : entry_point =
            ((InterpreterGenerator*)this)->generate_Reference
        _get_entry(); break;
        default : ShouldNotReachHere();
        break;
    }

    if (entry_point) return entry_point;

    return ((InterpreterGenerator*)this)->generate_normal_entry(synchronized);
}

```

在 generate_method_entry() 函数中，判断入参的枚举类型，当入参类型是 zerolocals 时啥也不干，因此跳出 switch 条件判断分支，直接到最后一句：

```
return ((InterpreterGenerator*)this)->generate_normal_entry(synchronized)
```

在 generate_normal_entry() 函数中，终于要开始为 zerolocals 生成本地机器指令了。在 32 位 x86 Linux 平台上，generate_normal_entry() 函数定义如下（为了少占篇幅，以下仅摘录主要逻辑）：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：generate_normal_entry() 函数

```

address InterpreterGenerator::generate_normal_entry(bool synchronized) {
    //...
    address entry_point = __ pc();

    // 定义寄存器变量
    const Address size_of_parameters(rbx,
methodOopDesc::size_of_parameters_offset());
    const Address size_of_locals(rbx, methodOopDesc::size_of_locals_offset());
    const Address invocation_counter(rbx,
methodOopDesc::invocation_counter_offset() +
InvocationCounter::counter_offset());
    const Address access_flags(rbx, methodOopDesc::access_flags_offset());

    // 获取 Java 方法入参数量、max_locals 及局部变量 slot 数量
    __ load_unsigned_short(rcx, size_of_parameters);
    __ load_unsigned_short(rdx, size_of_locals)
    __ subl(rdx, rcx);

```

```

//获取返回地址
__ pop(rax);

//计算 Java 方法第一个入参在堆栈中的地址
__ lea(rdi, Address(rsp, rcx, Interpreter::stackElementScale(), -wordSize));

//为局部变量 slot (不包含 Java 方法入参) 分配堆栈空间, 初始化为 0
{
    Label exit, loop;
    __ testl(rdx, rdx);
    __ jcc(Assembler::lessEqual, exit);           // do nothing if rdx <= 0
    __ bind(loop);
    __ push((int32_t)NULL_WORD);                 // initialize local variables
    __ decrement(rdx);                         // until everything initialized
    __ jcc(Assembler::greater, loop);
    __ bind(exit);
}

//创建栈帧
generate_fixed_frame(false);

//引用计数
Label invocation_counter_overflow;
Label profile_method;
Label profile_method_continue;
if (inc_counter) {
    generate_counter_incr(&invocation_counter_overflow, &profile_method,
&profile_method_continue);
    if (ProfileInterpreter) {
        __ bind(profile_method_continue);
    }
}
Label continue_after_compile;
__ bind(continue_after_compile);

//开始执行 Java 方法第一条字节码
#ifndef ASSERT
{ Label L;
    const Address monitor_block_top (rbp,
                                    frame::interpreter_frame_monitor_block_top_offset *
wordSize);
    __ movptr(rax, monitor_block_top);
    __ cmpptr(rax, rsp);
    __ jcc(Assembler::equal, L);
    __ stop("broken stack frame setup in interpreter");
    __ bind(L);
}

```

```

    }
#endif

    // jvmti support
    __ notify_method_entry();

    //进入 Java 方法第一条字节码
    __ dispatch_next(vtos);

    return entry_point;
}

```

`generate_normal_entry()`函数在 JVM 启动过程中调用，执行完成之后，会向 JVM 的代码缓存区写入对应的本地机器指令。当 JVM 调用一个特定的 Java 方法时，会根据 Java 方法所对应的 `entry_point` 类型找到对应的函数入口，并执行这段预先生成好的机器指令。

通过前面的分析可知，`CallStub` 例程所进入的 `entry_point` 例程其实就是方法入口，并且这个方法入口并不是只有一个，而是有一批。至于到底会进入哪一个人口，其实在编译期就确定了，编译器会判断 Java 方法的签名（Java 方法名、访问标识，是否使用 `synchronized` 锁定，是否是虚方法等），并根据 Java 方法的签名信息生成不同的方法调用指令。在一个 Java 类被 JVM 加载的过程中，同样会对每个 Java 方法进行签名信息分析，并最终确定一个 Java 方法的 `entry_point` 类型。

下面开始具体分析 `generate_normal_entry()` 函数执行的详细过程。由于 Java 方法栈主要由局部变量表、帧数据和操作数栈这三大部分组成，因此下面的讲解也主要从这几个方面进行讲解。

7.2 局部变量表创建

7.2.1 constMethod 的内存布局

世界是物质的，物质是运动的，运动是有规律的，规律是可以被掌握的。当 JVM 启动运行后，其内部的一切数据对象都处于不断运动的状态，而运动的规律，则由詹爷一手创建。相对于 JVM 内部的所有对象而言，詹爷就是“它们”的创世神。我等后辈如果能够掌握这种运动的规律，那么不是神也成仙了（哈哈）。

JVM 是使用 C/C++ 写成的，与其他基于 C 的程序一样，JVM 充分基于数据结构展开其独特的算法。JVM 内部变化的是对象的位置，是对象的诞生和消亡，是指向对象的指针。同样，

算法也会一直变化。但是唯一千年不变的是对象所代表的数据结构，是一串结构中各个元素的相对偏移。这种相对位移便成了 JVM 内部物质运动的一种内在规律，连 Java 函数在 JVM 内部所对应的 method 数据结构表达形式也挣脱不了这种造物主所设定的命运。

作为 JVM 世界的造物神，自然是知道这种规律的，因为规矩就是造物神定下的。在 entry_point() 例程中，这种规律将被用来定位 Java 函数所对应的字节码位置，并计算局部变量表的容量。对于 JDK 6，JVM 内部通过偏移量为 Java 函数定下了“规矩”，准确地说，至少定下了 3 条规矩：

- ◎ method 对象的 constMethod 指针紧跟在 methodOop 对象头的后面，也即 constMethod 的偏移量是固定的。
- ◎ constMethod 内部存储 Java 函数所对应的字节码指令的位置相对于 constMehtod 起始位的偏移量是固定的。
- ◎ method 对象内部存储 Java 函数的参数数量、局部变量数量的参数的偏移量是固定的。

注：constMethod 对象是 method 对象内部的一个字段。method 对象是 Java 方法在 JVM 内部所对等的数据结构，该结构会在下一章详细讲解。

这种偏移量是由 C++ 编译器保证的。JDK 6 的 method 及其相关属性的偏移量如图 7.1 所示。

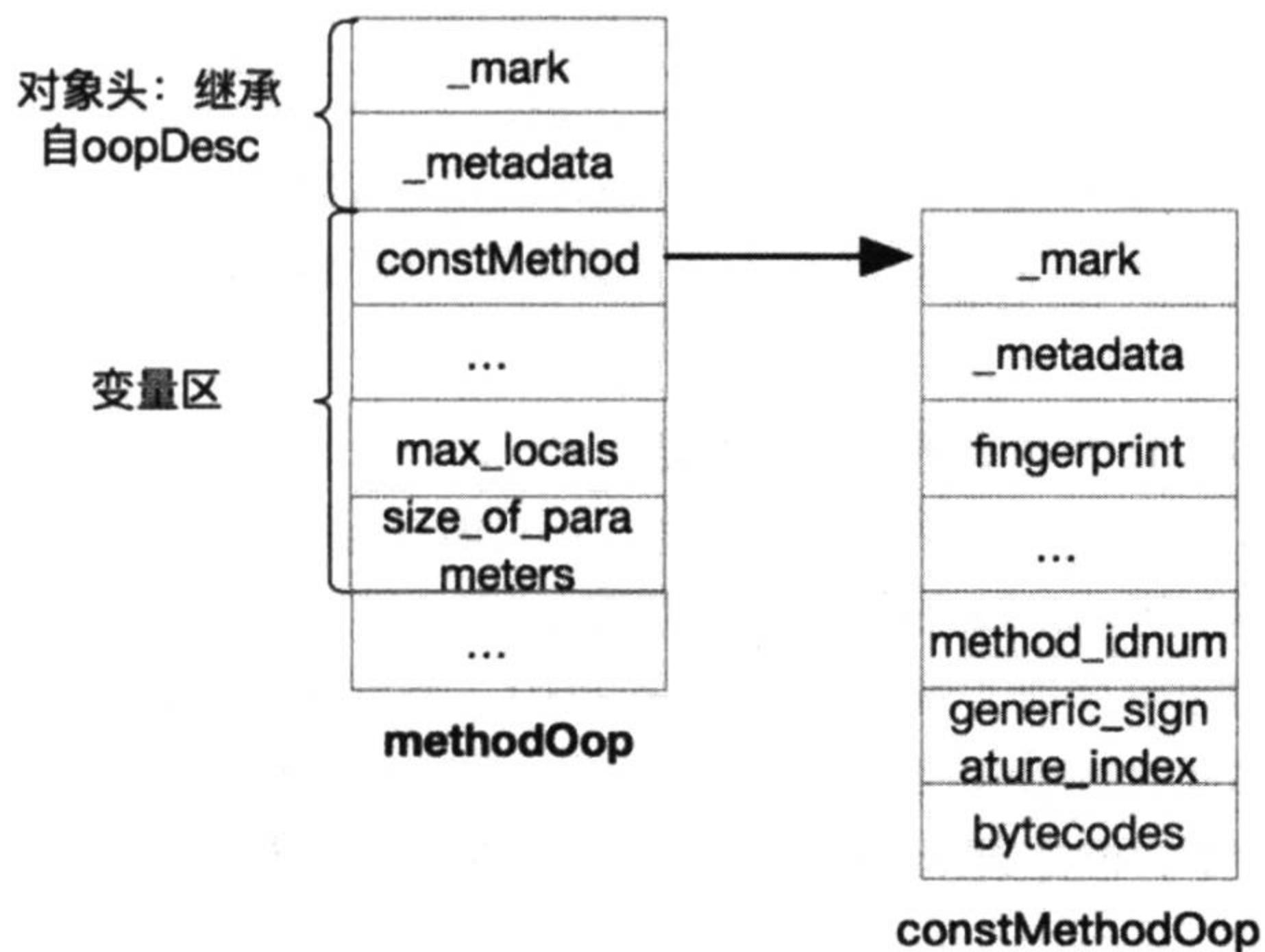


图 7.1 JDK 6 的 methodOopDesc 内部布局

首先看 constMethod，其相对于 methodOop 起始位置的偏移量是一个 oopDesc 对象头的距离。在 32 位 x86 平台上，这个距离为 8，即两个指针的宽度。

再看 Java 函数的两个十分重要的属性：`max_locals` 和 `size_of_parameters`。在 JDK 6 中，这两个参数被保存在 `methodOopDesc` 对象中，在 32 位 x86 平台上，`methodOopDesc` 的成员变量、所占用的内存大小（以字节为单位）、相对于 `methodOop` 起始位置的偏移量（以字节为单位）如表 7.1 所示。

表 7.1 `methodOopDesc` 对象内存分配

成 员 变 量	占 用 内 存 空 间	偏 移 量
header	4	0
klass	4	4
constMethodOop	4	8
constants	4	12
methodData	4	16
interp_invocation_count	4	20
access_flags	4	24
vtable_index	4	28
method_size	2	32
max_stack	2	34
max_locals	2	36
size_of_parameters	2	38

基于表 7.1，可以知道 `max_locals` 与 `size_of_parameters` 这两个参数相对于 `methodOop` 对象首地址的偏移量分别是 38 与 36。知道了这两个偏移量，JVM 将会基于此计算局部变量表的大小。

而到了 JDK 8，Oracle 公司的研发人员可能觉得，对于一个给定的 Java 函数，其 `max_locals` 与 `size_of_parameters` 这两个参数就是恒定不变的，不可能在运行过程中被修改，因此应该将其当作只读的属性。基于这种考虑的结果便是直接导致 `max_locals` 与 `size_of_parameters` 这两个参数被从 `methodOopDesc` 对象中移到了 `constMethod` 对象中，因为 `constMethod` 对象中的属性都是只读的。JDK 8 中，`methop` 与 `constMethod`（在 JDK 8 中，已经不再叫 `methodOop` 和 `constMethodOop` 了，后面的 Oop 没有了，但是数据结构并没有太大的变化）的内存布局如图 7.2 所示。

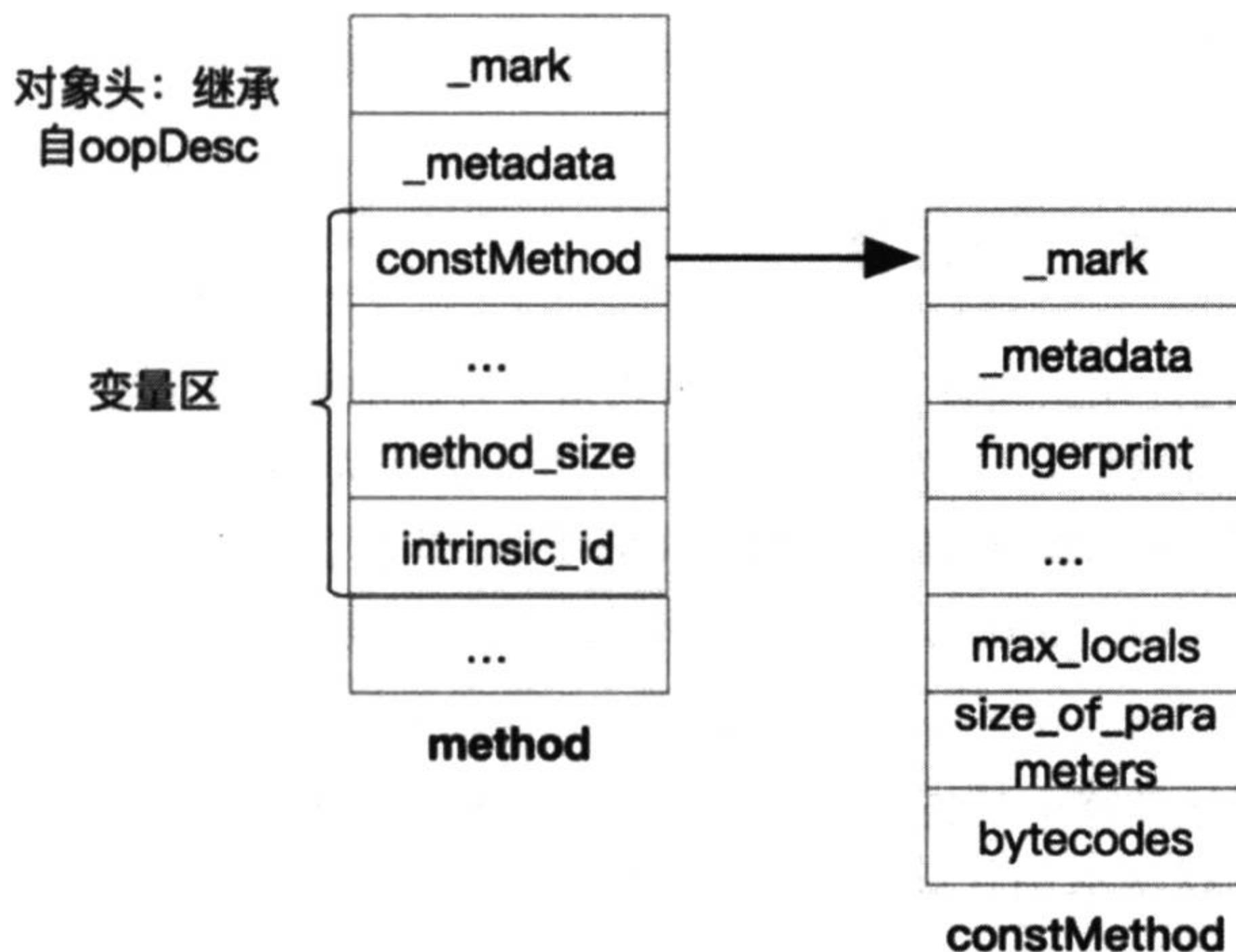


图 7.2 JDK 8 的 method 对象内存布局图

所以，在 JDK 8 中，要想从 methodOop 对象中读取到 max_locals 和 size_of_parameters 这两个参数，便不能再基于 method 的偏移量去读取了，只能基于 constMethod 的偏移量进行读取。在 32 位 x86 平台上，JDK 8 中的 constMethod 类型的成员变量、所占用的内存大小（以字节为单位）、相对于 constMethod 起始位置的偏移量（以字节为单位）如表 7.2 所示。

表 7.2 constMethod 对象内存分配

成员变量	占用内存空间	偏移量
fingerprint	8	0
constants	4	8
stackmap_data	4	12
constMethod_size	4	16
flags	2	20
code_size	2	22
name_index	2	24
signature_index	2	26
method_idnum	2	28
max_stack	2	30
max_locals	2	32
size_of_parameters	2	34

记住这里的偏移量，JVM 内部会基于偏移量来分配堆栈空间。

7.2.2 局部变量表空间计算

如果你对 JVM 内存模型或者 Java 的字节码指令有过接触，就会明白，要研究它们怎么都绕不开一个常见的内存区域——局部变量表。局部变量表作为 Java 方法堆栈(栈帧)的一部分，主要的作用就是保存 Java 方法内部所声明的局部变量，当然，也包含入参。成功为 Java 函数分配局部变量表的第一步就是正确计算出 Java 函数局部变量表所需的大小。

Java 方法的局部变量表包含 Java 方法的所有入参和方法内部所声明的全部局部变量。在编译阶段，编译器准确地计算出 Java 方法的局部变量空间。而到了运行期，仅仅知道局部变量空间大小是不够的，JVM 需要为其分配足够的堆栈空间。虽然编译期间便知道了所需要的局部变量空间大小，但是这对 JVM 进行堆栈空间分配并不能提供足够的信息，因为通常情况下，Java 方法的入参的堆栈空间是由调用方所分配，因此被调用方并不需要再分配编译期所计算出的全部局部变量空间。对于 Java 方法之间的调用（是指一个 Java 方法调用了另一个 Java 方法，而非本地函数调用 Java 方法），调用方的操作数栈与被调用方的局部变量表往往存在重叠区，这给 Java 方法局部变量表的分配带来一定的挑战(栈帧重叠的原因会在后续章节进行详细讲解)。

所以在运行期，JVM 只需要为 Java 方法的局部变量分配堆栈空间，而不需要为 Java 函数的入参额外分配空间，因为入参的堆栈空间由调用方完成分配。

很绕，有没有？

事实上，这么绕是有道理的，根本原因是编译期间所获取的信息与运行期间所能获取的信息是不对称的，并且运行期间可能会有各种优化。

对于绕的东西，举例来理解是一个不错的办法。下面是一个简单的 Java 类，包含一个很简单的方法：

清单：A.java

作用：局部变量表空间示例

```
class A{
    public void add(int x, int y) {
        int z = x + y;
    }
}
```

使用 `javap -verbose` 命令进行分析，得到的信息如下：

```
public void add(int, int);
descriptor: (II)V
```

```

flags: ACC_PUBLIC
Code:
stack=2, locals=4, args_size=3
0: iload_1
1: iload_2
2: iadd
3: istore_3
4: return

```

`add()`方法的局部变量表的最大容量是 4 (`locals=4`)，入参数量是 3 (`args_size=3`)。入参数量之所以是 3，是因为 `add()`方法是类的成员方法，因此会有隐藏的第一个人参 `this`。3 个人参，加上 `add()`方法内部所定义的一个局部变量 `z`，构成了局部变量表的容量。

注意看 `add()`方法所对应的字节码指令，当执行 $z = x + y$ 时，需要先执行两条字节码指令：

```

iload_1
iload_2

```

这两条字节码指令分别将局部变量表的第 1 个槽位和第 2 个槽位的数据推送至表达式栈栈顶（槽位起始编号从 0 开始）。第 1 和第 2 个槽位上所保存的数据，正是 `add()`方法的两个人参 `x` 和 `y`。很显然，第一个槽位上所保存的数据是 `this` 指针。更加显然的是，JVM 内部的局部变量表的确包含了入参。

当执行完 $z = x + y$ 后，对应的最后一条字节码指令是 `istore_3`，它将计算结果保存到局部变量表的第 3 个槽位。第 3 个槽位正是 `add()`方法内部所定义的局部变量 `z`。

此时的局部变量表的内部结构与索引如下：

槽位索引	对应入参/局部变量
0	<code>this</code>
1	<code>x</code>
2	<code>y</code>
3	<code>z</code>

注：在 32 位平台上，一个槽位占 32 位。

对于本例，由于 `x` 和 `y` 这两个人参在调用方调用 `add()` 方法时便已经分配完毕，因此 JVM 无须再为这两个人参分配堆栈空间，只需为 `z` 分配。而 `z` 所需的堆栈空间大小，则为编译期间所计算出的局部变量表的大小减去入参数量，对于本例而言，就是 $(4 - 3) = 1$ 。因此，JVM 只需要为 `add()` 方法内的局部变量 `z` 分配 1 个变量槽。

`entry_point` 例程里给出了这种除入参之外所需要的局部变量表的空间大小的计算方法：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：generate_normal_entry()函数

```
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
    //...
    const Address size_of_parameters(rbx,
methodOopDesc::size_of_parameters_offset());
    const Address size_of_locals      (rbx,
methodOopDesc::size_of_locals_offset());
    const Address invocation_counter(rbx,
methodOopDesc::invocation_counter_offset() + InvocationCounter::counter_offset());
    const Address access_flags      (rbx, methodOopDesc::access_flags_offset());

    // 获取 Java 函数入参数量
    __ load_unsigned_short(rcx, size_of_parameters);

    // 获取 Java 函数局部变量表最大槽数
    __ load_unsigned_short(rdx, size_of_locals);

    //最大槽数减去 Java 函数入参数量，得到除入参之外所需要分配的堆栈空间大小
    __ subl(rdx, rcx);

    //...
}
```

最终生成的机器指令如下（使用汇编展示，基于 32 位 x86 平台，JDK 6）：

```
movzwl 0x26(%ebx),%ecx
movzwl 0x24(%ebx),%edx
sub    %ecx,%edx
```

从 call_stub 例程进入 entry_point 例程之前，ebx 寄存器指向 Java 函数所对应的 method 对象首地址，根据前文所分析的 JDK 6 中 method 的结构可知，相对于 method 首地址偏移 38 个字节的位置保存的是 max_locals 参数，而偏移 36 个字节的位置保存的是 size_of_parameters 参数。38 和 36 所对应的十六进制正是 0x26 和 0x24。

到了 JDK 8，由于 max_locals 与 size_of_parameters 这两个参数保存的位置发生了变化，因此其寻址方式也跟着发生变化。JDK 8 最终所生成的机器指令如下：

```
mov    0x8(%ebx),%edx
movzwl 0x22(%edx),%ecx
movzwl 0x20(%edx),%edx
sub    %ecx,%edx
```

由于在 JDK 8 中，max_locals 与 size_of_parameters 这两个参数从 method 对象移到了 constMethod 对象中，因此首先执行 mov 0x8(%ebx),%edx，将 edx 寄存器指向 constMethod 对象

首地址。因为 constMethod 相对于 method 对象的首地址的偏移量为 8 字节，而程序流从 call_stub 例程进入 entry_point 例程之后，ebx 寄存器指向 method 对象首地址，因此通过 0x8(%ebx) 将 ebx 寄存器往上移动 8 字节的宽度，就得到 constMethod 首地址，并将这个首地址保存到 edx 寄存器中。接着再基于 constMethod 的偏移量分别通过 0x22(%edx) 与 0x20(%edx)，就得到 max_locals 与 size_of_parameters 这两个参数值。

7.2.3 初始化局部变量区

在 CallStub 执行 call %eax 指令之前，物理寄存器（注：不是逻辑寄存器哦）中所保存的重要信息如表 7.3 所示。

表 7.3 物理寄存器中的信息

寄存器名	指 向
edx	parameters 首地址
ecx	Java 函数入参数量
ebx	指向 Java 函数，即 Java 函数所对应的 method 对象
esi	CallStub 栈顶

计算出除 Java 方法入参之外的参数所需要分配的堆栈空间，接着就是执行空间分配。

对于绝大多数 C/C++ 语言，包括 Delphi 等可以直接编译为本地二进制机器指令的语言，分配堆栈空间基本都使用如下指令：

```
sub operand, %esp
```

esp 寄存器指向调用者函数的栈顶，要扩展堆栈的内存空间，只需将栈顶指针继续往下移动，移动多大空间，就能扩展多大空间。当然也不是随便想扩充多大就扩充多大，对于基于硬件的操作系统而言，往往有一个最大堆栈深度的限制；而对于基于软件的 JVM 虚拟机，本身也提供了这种限制，具体的限制通过参数-Xss 进行控制。

而在 entry_point 例程中，分配堆栈空间使用了另一种方式，那就是进行 push 操作。entry_point 例程所对应的源码如下：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：generate_normal_entry() 函数

```
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
    //...
    // get return address
```

```
    __ pop(rax);

    // compute beginning of parameters (rdi)
    __ lea(rdi, Address(rsp, rcx, Interpreter::stackElementScale(),
-wordSize));

    // rdx - # of additional locals
    // allocate space for locals
    // explicitly initialize locals
    {
        Label exit, loop;
        __ testl(rdx, rdx);
        __ jcc(Assembler::lessEqual, exit);      // do nothing if rdx <= 0
        __ bind(loop);
        __ push((int32_t)NULL_WORD);           // initialize local variables
        __ decrement(rdx);                   // until everything initialized
        __ jcc(Assembler::greater, loop);
        __ bind(exit);
    }
    //...
}
```

这段逻辑主要执行了 3 件事：

- (1) 将栈顶的返回地址暂存到 `rax` 寄存器中。
- (2) 获取 Java 函数第一参数在堆栈中的位置。
- (3) 为局部变量表分配堆栈空间。

在 32 位 x86 平台上，这段逻辑所生成的机器指令如下（使用汇编展示）：

```
// get return address
pop    %eax

// compute beginning of parameters (rdi)
lea    -0x4(%esp,%ecx,4),%edi //让 edi 指向第一个参数在堆栈中的位置

// rdx - # of additional locals
// allocate space for locals
// explicitly initialize locals
test   %edx,%edx    //测试 edx 是否为 0，即判断 Java 函数中是否有局部变量
jle   0x01cbbb08    //如果 Java 函数内部没有声明局部变量，则跳过堆栈空间分配
push   $0x0
dec    %edx
jg    0xb36d6559
```

下面对局部变量表的初始化逻辑进行详细讲解。

1. 暂存返回地址

JVM控制流从call_stub例程刚进入entry_point例程时，尚未对堆栈空间进行任何操作，因此到目前为止，call_stub例程与entry_point例程的堆栈空间如图7.3所示。

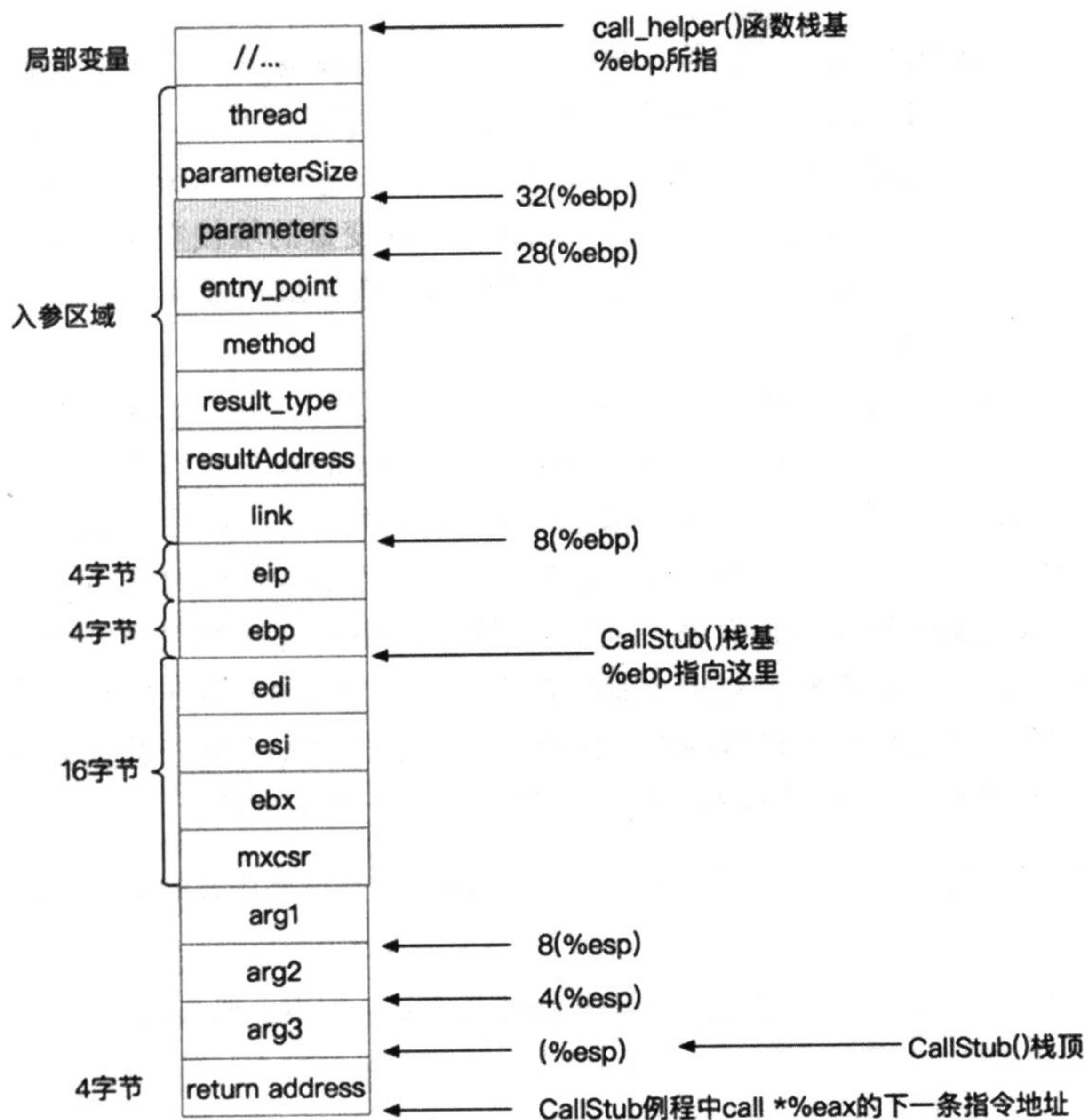


图7.3 CallStub执行完call *%eax指令后的内存布局

可以看到，在栈顶位置保存的是目标Java函数的返回地址，注意这个所谓“返回地址”的概念，在上一章讲解CallStub例程时对此进行过专门描述，它是指CallStub例程中“call *%eax”这条指令所在内存区域的下一条指令地址，这个值实际上是原本CallStub例程中eip寄存器的值。因此，这里的“返回地址”并不是指目标Java函数的返回值。不过要注意的是，JVM内部也将return address叫作“return_from_java”，因此下文有时根据需要写成return_from_java，诸君心里知道这其实也是return address即可。不管是return address也好，还是return_from_java也罢，其实都是eip。

接下来，JVM 要为被调用者函数的局部变量分配堆栈空间，在分配之前，需要先将“返回地址”临时保存到 `rax` 寄存器中。

但是为何要这样做呢？

原因主要包括以下两方面：

- ◎ 第一，Java 函数入参也是局部变量表的一部分。被调用的 Java 方法的入参信息，就保存在 `call_stub` 例程栈顶返回地址的上方，我们看图 7.3，`return_address` 存储单元的上面就是目标 Java 方法的入参，分别是 `argument word 1~argument word n`。由于接下来要为被调用的 Java 方法分配局部变量的堆栈空间，而入参也属于被调用方的局部变量，因此理所当然地要将这两块区域连成一片，中间不能有分隔，否则看着多别扭啊！
- ◎ 第二，JVM 实现了操作数栈与局部变量表的复用。当一个 Java 方法调用了另一个 Java 方法时，最终调用方的栈顶是 `return address` 与操作数栈，操作数栈位于 `return address` 的上方。但是当发生 Java 方法调用时，调用方 Java 方法会将被调用方的入参复制到调用方的操作数栈，此时操作数栈便同时充当了入参的堆栈内存。这么做的原因也很简单，就是为了提高 Java 方法调用的效率，如果不实现操作数栈与入参堆栈的复用，那么必然会专门为入参开辟出一段内存空间，这样做，除了会消耗物理机器内存，还会额外增加许多指令用于数据复制，额外耗费非常可观的 CPU 计算成本，这在时间与空间上都不划算。

既然操作数栈被同时用作了入参的堆栈，而入参同时又是被调用方的局部变量表的一部分，因此就必然要将其内存空间连成一片。

如果调用方也是 Java 方法，那么以上两点说的其实是一回事，那么调用方的操作数栈其实同时也是被调用方的入参堆栈。

基于以上两点，入参堆栈与即将分配的局部变量的堆栈之间不允许存在一个碍事的 `return address` 参数，因此 JVM 便将这个参数先移走。

JVM 对 `return address` 说：“麻烦您老兄先到别处凉快会儿，这儿暂时没您啥事儿，等这边的事儿完了，到时再麻烦您老兄移驾复位。”

`return address` 心里想：“我去！没事老拿我开涮！”，嘴上应声道：“好嘞！”

于是 JVM 念起那段古朴沧桑的强大咒语：`pop %eax`，将 `return address` 瞬间传送到 `eax` 中。

2. 获取 Java 函数第一个入参在堆栈中的位置

JVM 念完咒语, `return address` 瞬间消失, 现在 JVM 从栈顶一眼看过去, 第一个映入眼帘的参数变成了 `argument word n`, 也就是 Java 方法的最后一个人参。

`argument word n` 想跟 JVM 打声招呼, 但是 JVM 并没有理睬, 而是大声呼喊:

“第一个参数, 给我站出来!”

喊完了, 但是没人理它。

悲哀!

JVM 一言不合就念咒语, 那古老的咒语是:

```
lea    -0x4(%esp,%ecx,4),%edi
```

念完咒语, CPU 偷偷地将第一个参数的堆栈坐标写进了 `edi` 寄存器。

将这个咒语展开, 等价于下面这个表达式:

$$(\%edi) = (\%esp) + (\%ecx) * 4 - 0x4$$

程序流从 `call_stub` 例程跳转到 `entry_point` 例程时, `ecx` 寄存器中记录的是 Java 方法入参的数量, 假设入参数量是 N , 则由此可知, 第一个参数的位置在当前栈顶往上 N 字节, 再往下移动 4 字节。这里需要注意的是, 由于堆栈空间从高地址内存位置向低地址内存位置增长, 所以上面表达式的前半部分的子表达式($(\%esp) + (\%ecx) * 4$)所计算出的结果实际上是 Java 方法第一个人参的内存位置的最高位地址, 但是在大部分主流 CPU 架构平台上, 都是将一个数据的最低位内存地址标记为该数据的内存地址, 而不是最高位内存地址, 因此上面表达式最后需要减去 `0x4`, 从而得到 Java 方法的第一个人参的最低位地址, 这就是第一个人参的内存首地址。同时需要注意的是, 在 32 位平台上, 一个指针类型的数据宽度是 4 字节, 因此这里减去的是 4; 但是在 64 位平台上, 由于一个指针的数据宽度是 8 字节, 因此在 64 位平台上, 上面这段表达式就变成如下这样:

$$(\%edi) = (\%esp) + (\%ecx) * 8 - 0x8$$

在该表达式中, 自然数 `0x4` 变成 `0x8`。

这个“经”之所以这么念, 是有道理的, JVM 可是精打细算的主! 刚才在赶走 `return address` 老兄时, 所念的咒语是 `pop %eax`, 这段咒语念完, 栈顶的数据被弹出, `esp` 寄存器会自动往上移动, 移动之后的堆栈内存布局如图 7.4 所示。

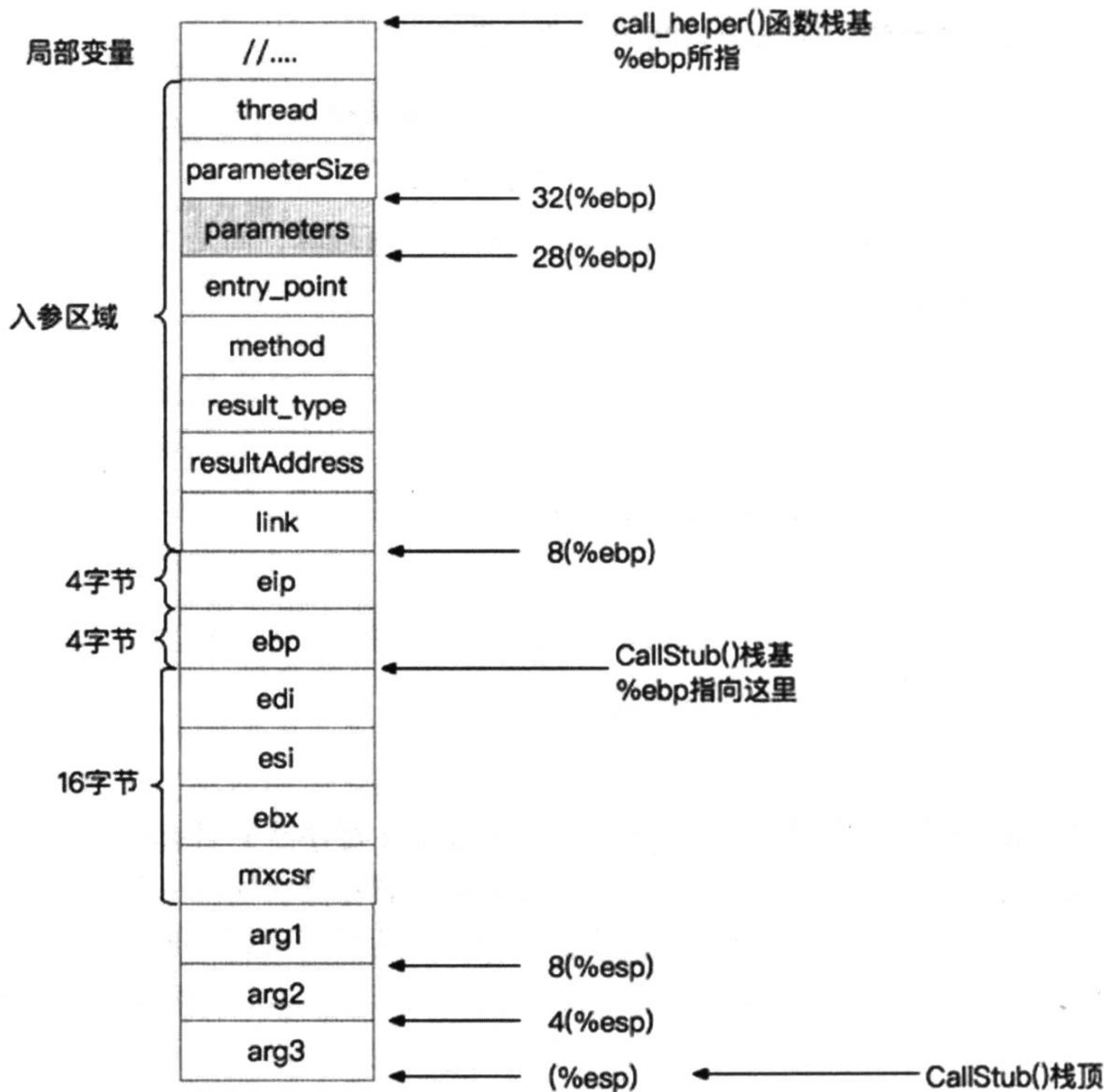


图 7.4 将 return address 弹出之后的堆栈内存布局

由于现在 `rsp` 寄存器指向了最后一个参数，因此需要往上移动 $(n-1) * 4$ 字节的位置，找到第一个参数的堆栈地址。注意，JVM 在这里使用了 `lea` 命令，而非 `mov`。`lea` 命令是专门用于获取内存地址的命令。

在这一步之所以要将 Java 函数的第一个入参的位置保存下来，是因为接下来要为 Java 函数内部所声明的局部变量分配堆栈空间，并且要执行 Java 字节码指令，字节码指令往往都会涉及在操作数栈与局部变量表之间相互传送数据，而 JVM 要读取/写入局部变量表，必然要知道局部变量表的起始位置，否则无法定位。因此在这一步将局部变量表的起始位置保存到 `edi` 寄存器中，后续相关的字节码指令例如 `iload`、`istore` 等都需要用到 `edi` 寄存器，直接从 `edi` 寄存器中读取局部变量表的起始位置。

更进一步说，这也是对局部变量表读取和写入时都基于索引的原因，因为 JVM 的确就是基于局部变量表的起始位置做偏移的，从而读取/写入局部变量表相关数据。局部变量的索引号其

实就是入参或局部变量相对于局部变量表的偏移量。

Java 方法的第一个入参的内存位置将作为 Java 方法的局部变量表的起始位置，并保存在 edi 寄存器中。这里其实偷偷地埋下了一个巨坑，假设一个 Java 方法包含 3 个入参，同时假设该方法没有局部变量，则该方法的局部变量表及 edi 寄存器所指向的位置如图 7.5 所示。

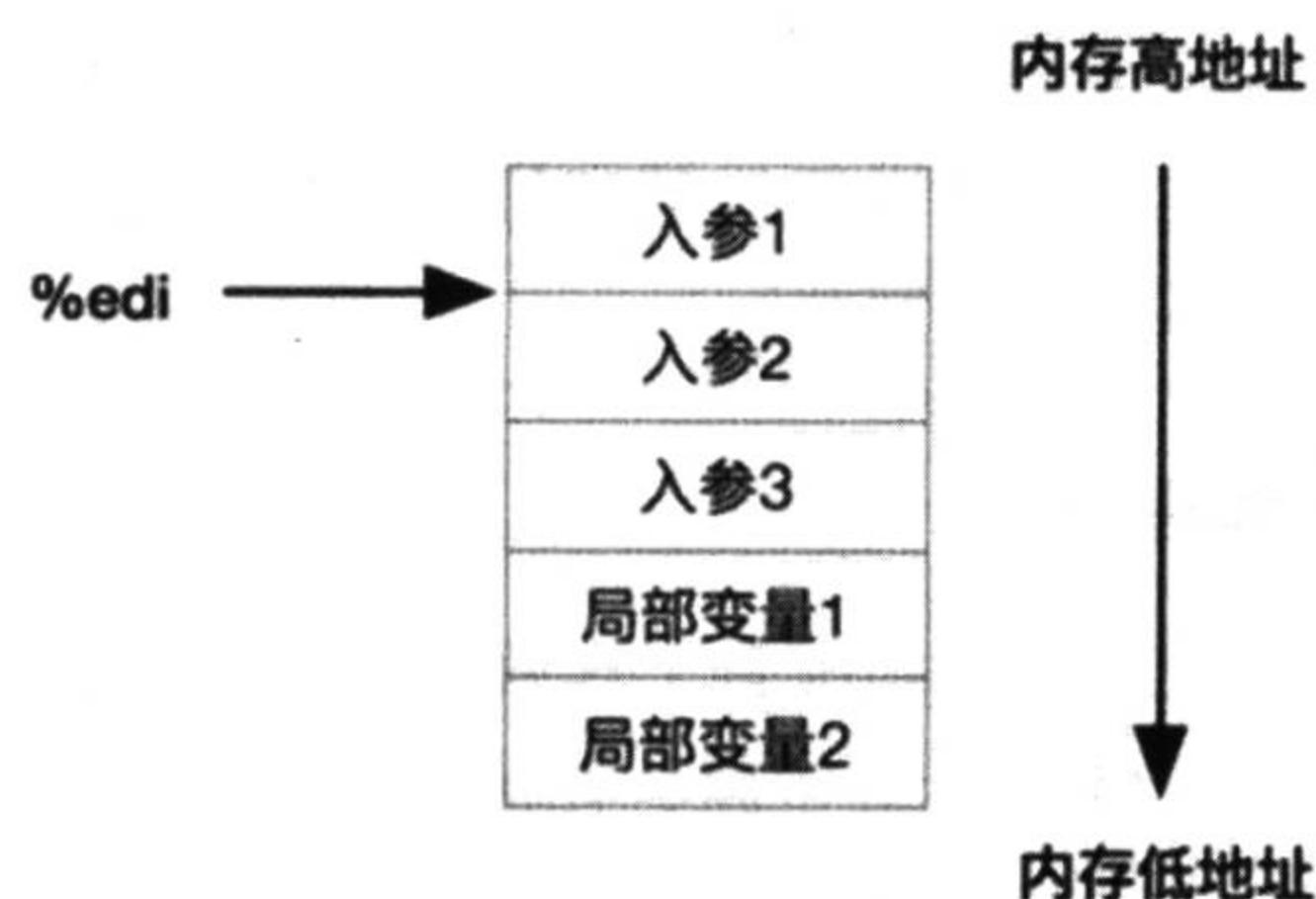


图 7.5 Java 局部变量表的 slot 槽位起始位置

edi 寄存器所保存的正是 Java 方法第一个入参的内存位置，Java 方法的局部变量表的 slot 位置也以该位置为起始点。而事实上，对于计算机内部的一段连续的数据区域，其内存首地址是整个数据区域中地址最低的那个位置，很显然，slot 的起始位置并不符合这一原则。图 7.5 中，由于 Java 方法堆栈由内存高地址向低地址方向增长，因此 Java 方法的第一个入参的内存地址反而处于最高位，因此如果按照一般逻辑，Java 方法的局部变量表的 slot 起始位置应该如图 7.6 所示才对。

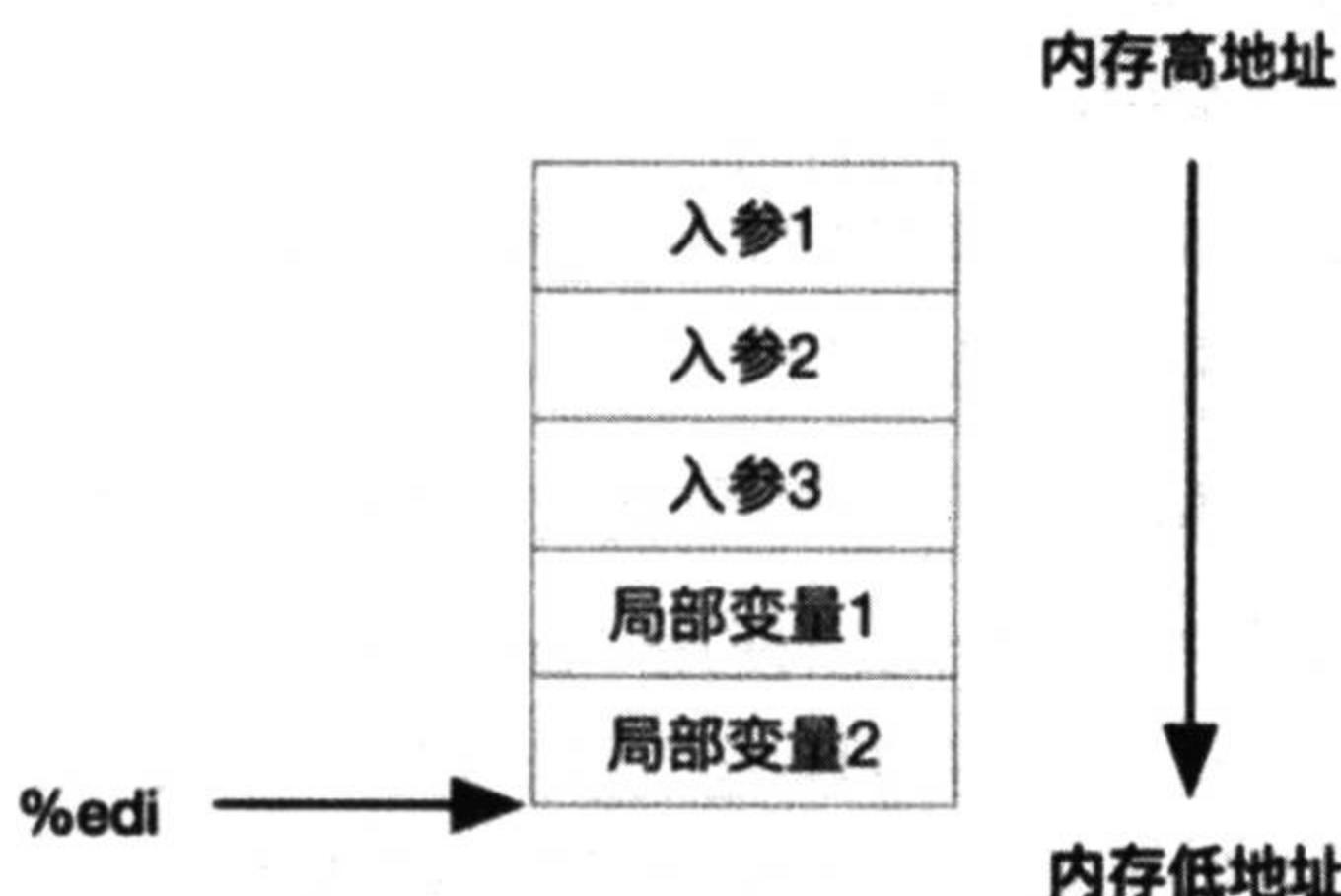


图 7.6 Java 局部变量表的 slot 槽位起始位置——按一般逻辑

由于 JVM 将整个局部变量表的高位地址作为其起始位置，因此运行期的字节码指令操作局部变量表的 slot 槽位索引时，必须充分考虑到这一点，而这也是理解读写局部变量表所对应的

load 与 store 系列的字节码指令所对应的机器码的关键所在，否则将会一头雾水。同时，在这一步计算入参位置时，默认将 Java 方法入参全部当成指针类型，因此在 32 位和 64 位平台上，Java 方法的第一个人参的内存位置要么以 32 位计算，要么以 64 位计算，但是在局部变量表中，普通数据类型（例如 int）与 long 类型所占的槽数不同，而无论在 32 位还是 64 位平台上，局部变量表中的 long 类型的数据宽度都要比指针类型的数据宽度大，因此 JVM 在处理局部变量表中的 long,double 类型的数据时，需要处理好 edi 指针的位置，计算出 long,double 类型数据在局部变量表中的真实偏移量。这一点在后文讲解局部变量表及其读写字节码指令所对应的机器码时会详细讲解。这里先提前剧透一部分“情节”。

3. 为局部变量表分配堆栈空间

JVM 念完两个古老的咒语之后，一切准备工作就绪，开始为 Java 方法内部所声明的局部变量分配堆栈空间了。这一次，JVM 不走寻常路，咒语是这样念的：

```
test    %edx,%edx      //测试 edx 是否为 0，即判断 Java 函数中是否有局部变量
jle    0x01cbbb08      //如果 Java 函数内部没有声明局部变量，则跳过堆栈空间分配
push   $0x0              //0xb36d6559
dec    %edx
jg    0xb36d6559        //这个地址就是上面第 3 行的指令地址，push $0x0
```

这段指令的逻辑是，先测试 edx 是否为 0，edx 寄存器中所保存的结果就是前面 max_locals-size_of_parameters 后得到的值。如果这个值为 0，则说明被调用的 Java 方法内部并没有声明任何局部变量，于是 JVM 不会再去分配堆栈空间。能省一件事就省一件事，整天念咒语，累啊！

如果 max_locals - size_of_parameters 不等于 0，则执行一个微循环，先通过 push \$0x0 往栈顶压入一个 0，接着通过 dec %edx 将 edx 寄存器中的值减去 1，最后再判断 edx 的值是否为 0，如果不是 0 则继续跳转回 push \$0x0，否则循环结束。

假设 Java 方法内部定义了 5 个局部变量，则这段逻辑执行完之后，最后的堆栈内存布局如图 7.7 所示（仍然假设目标 Java 方法包含 3 个人参）。

通过不断 push 的方式，JVM 完成了局部变量表的堆栈空间分配。

其实，在 entry_point 例程中，完全可以使用“sub operand, %esp”这样的方式分配堆栈空间，但是这里偏偏没有这么做。这也是有道理的，这是因为 Java 方法的堆栈空间会被反复使用。对于一片指定的堆栈区域，就像个战场，战场永远是那个战场，但是却前赴后继来了一茬一茬又一茬的军队。战争有它自己的规律，每次完成一场决战，胜利的一方都要打扫战场，清理物资。而对于同一块内存区域，这一次可能是被作为 Java 方法 a() 的堆栈，而下一次则可能变成方法 b() 的堆栈。与战场所不同的是，Java 方法执行完毕之后，并不负责清零堆栈，因此清零的工作只能由下一次使用这块堆栈空间的 Java 方法去负责。这便是在 entry_point 例程中使用循环

push 0 的方式进行分配堆栈空间的原因。如果使用“sub operand, %esp”的方式去分配堆栈空间，分配完了仍然需要进行一次循环，使用“mov 0, -operant(%esp)”这样的方式对堆栈空间进行清零，效率反而低下，不美。

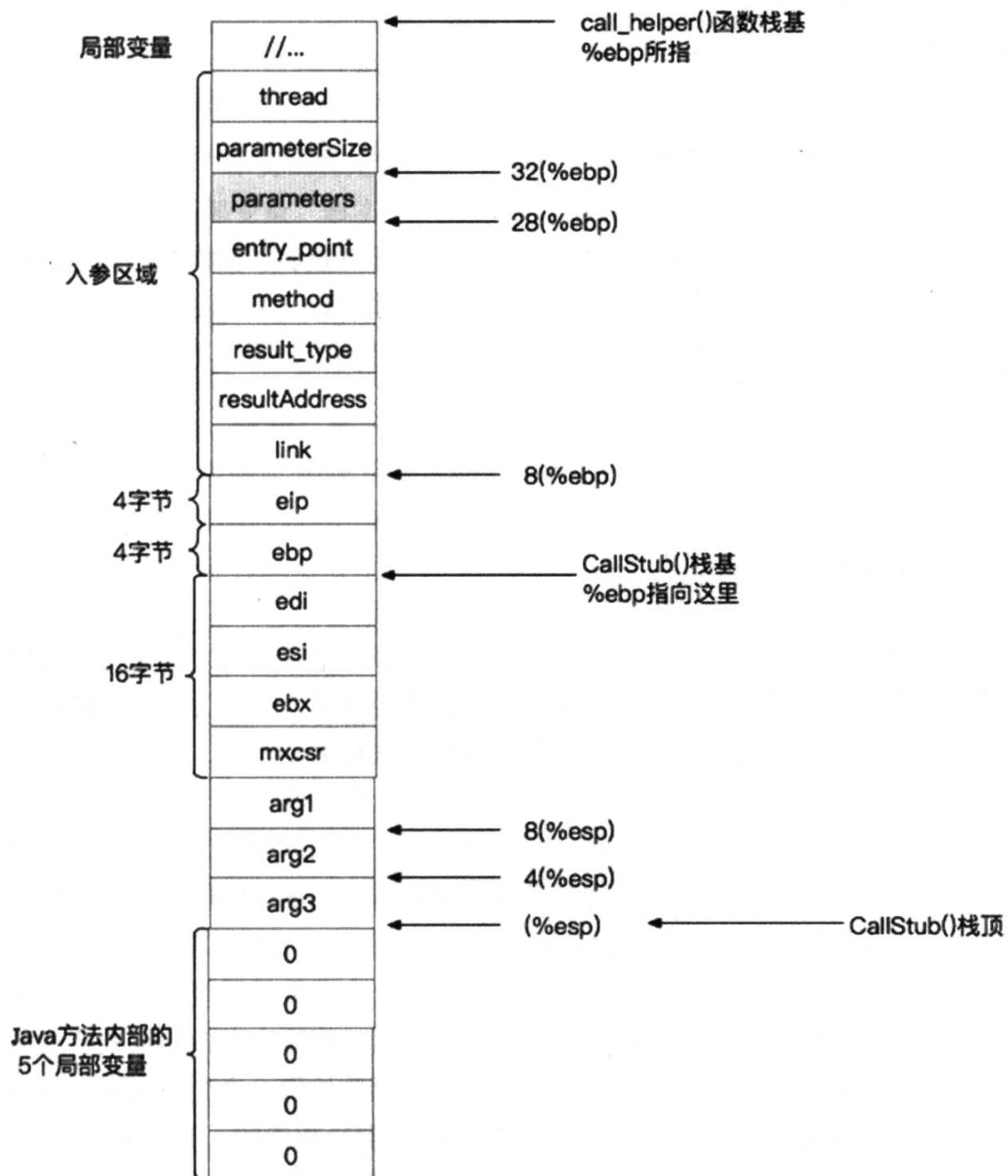


图 7.7 JVM 为局部变量分配堆栈空间

7.3 堆栈与栈帧

JVM 为局部变量分配完堆栈空间后，接着执行一项特殊的任务。这项特殊的任务就是构建栈帧，准确地说，是构建 Java 方法所对应的栈帧中的固定部分，这个固定的部分在 JVM 内部有一个专门的称谓——fixed frame。

fixed frame 不是指固定的栈帧，而是指栈帧中的固定部分。众所周知，在 JVM 内部，每个 Java 方法都对应一个栈帧，这个栈帧中会包含局部变量表、操作数栈、常量池缓存指针、返回地址等数据。

其实，对于一个 Java 方法的栈帧，其首尾分别是局部变量表和操作数栈，中间则是除此以外的其他重要信息。而恰恰是首尾的这两部分数据，不同的 Java 方法是不同的，只有除去首尾这两部分不同的部分，处于栈帧中间位置的部分，其数据结构才是固定不变的，这部分就是 fixed frame。

本书将这部分数据称为“固定帧”数据。

7.3.1 栈帧是什么

在讲解 Java 方法的栈帧之前，先了解下到底啥是栈帧。笔者敢肯定很多人早对这两个字如雷贯耳，但是却着实不知道这到底是个啥玩意儿。其实说白了，一个方法的栈帧就是这个方法所对应的堆栈。frame 有时也译作“框架”，不过这种译法也名副其实，对于一个特定的组件或者中间件，当你进入其首个方法入口后，则后续所有一切计算就都与这个入口函数息息相关了，而这个入口函数也的确形似于整个中间组件的“门户”了，所以称其为框架，的确当之无愧。而译作“帧”，汉语里一幅画称为一个“帧”，所以显示器每一次的刷屏就叫作一帧，电影胶卷里的一个胶片也叫作一帧。如果一个 gif 动画由 36 幅图组成，就说其有 36 个帧。软件程序的运行过程，说白了就是一个个函数不断调用的过程，一个个函数粉墨登场，每一个函数都有堆栈，当函数运行时，函数会向其对应的堆栈空间写入数据，将原本空白的堆栈空间渲染成由 0 和 1 所组成的有特殊业务含义的数据“画面”，一个个函数仿佛一个个绘画家，各自对着自己的堆栈“笔走龙蛇，挥毫泼墨”。这些函数的堆栈空间是前后相连的，看起来就像放电影一样，所以翻译者将 frame 译作“帧”，实在是充满了丰富的想象力，展现出十足的艺术范儿。

当然，除了栈帧的叫法，很多人也将其叫作“活动记录”，这主要来源于一个英文单词——“activation records”。当然活动记录与堆栈的概念还有一些细微的区别，活动记录更多用于特指当前位于栈顶的堆栈，因为 CPU 只关心最顶端的堆栈。所以关于堆栈的叫法实在是不少，不过现在就让我们忘了这些名字或概念，而是去思考下面两个问题：

(1) 堆栈到底是啥?

(2) 堆栈有什么作用?

要回答这两个问题，还得从机器的角度说起。

对于第一个问题，答案其实很简单，函数内部会定义局部变量，这些变量最终要在内存中占用一定的内存空间。由于这些变量都位于同一个函数中，因此一个很自然的想法就是将这些变量合起来当作一个整体，将其内存空间分配在一起，这样有利于变量空间的整体内存申请和释放。所以抛开“栈帧”本身的具体含义，完全可以将“栈帧”看作一个“容器”，这个容器中存放的是函数内部的局部变量。事实上，几乎所有编程语言的函数所对应的堆栈空间的确就是个容器，在容器内部会按顺序存放函数内的局部变量。

所以，栈帧是个容器，这个概念要记住。

在程序运行的过程中，一个函数会有一个栈帧，多个函数的栈帧连起来，就变成堆栈。在《算法与数据结构》里，“堆栈”是一种数据结构，也是一种容器。堆栈里的元素会按照 FILO (first in/last out) 的顺序增加/删除元素。对于一个运行中的程序，从主函数进入的一系列函数的栈帧按照 FILO 的顺序在内存中分配空间，形成了一个堆栈。所以，程序的“堆栈”的成员元素就是函数的栈帧，而栈帧本身也是个容器，所以程序的堆栈其实是“容器的容器”。

所以，栈帧是个容器，堆栈是多个栈帧连成一片后形成的大容器，是容器的容器。

这样就回答了上面第一个问题，“堆栈到底是啥？”接下来要解答第二个问题，“堆栈有什么作用？”要回答这个问题，必须思考为什么要使用“堆栈”这种大容器来保存函数的“栈帧”小容器？看下面这个 C 语言的例子：

清单：test.c

作用：演示栈帧

```
int add(int, int);

int main() {
    int m = 5;
    int n = 3;
    int z = add(m, n);

    return 0;
}

int add(int m, int n) {
    int z = m + n;
    return z;
}
```

这个例子很简单，add()函数用于求和。编译后，这段程序会生成对应的机器码。站在机器的角度看，要让 main()函数成功调用到 add()函数，需要解决下面 4 个主要问题：

- ◎ 当 CPU 从 main()函数跳转到 add()函数时，CPU 要能够拿到 add()函数的机器指令的位置，否则 CPU 无法执行 add()函数的机器指令。
- ◎ 当 add()函数执行完成之后，CPU 要能够重新拿到 main()函数的机器指令，以便跳转回 main()函数继续执行。
- ◎ 当 CPU 从 main()函数进入 add()函数之后，CPU 需要为 add()函数里的变量分配内存空间，以便在内存中存储 add()函数内部的局部变量的值。
- ◎ 当 CPU 从 add()函数返回 main()函数之前，由于 add()函数已经完成其使命，其内部的局部变量失去了意义，并且外部也压根儿访问不到这些局部变量，因此 CPU 需要将 add()函数的局部变量回收掉，以便让宝贵的内存空间能够重复被使用。

在考虑这 4 个问题时，让我们忘记所谓的栈帧，忘记一切，我们就是最原始的 CPU 设计师。在那个时代，还没有所谓堆栈的说法问世呢！

作为一个 CPU，面前广阔无垠的内存空间可以任意驱驰，身后跟着 ax、bx、cx 等一众小弟，它们的名字叫作寄存器。CPU 的主要任务就是读取一条机器指令，然后执行，然后再接着读取下一条指令，再执行，再读取下一条指令再执行……，如此循环往复。上面第 1 个问题和第 2 个问题比较好解决，当加载程序时，只需将 main()函数和 add()函数的机器指令分别保存到内存的两个地方，并且 CPU 能够拿到这两个地方的内存地址，就可以了。当 CPU 执行完 main()函数的最后一条机器指令后，能够得到 add()函数的第一条机器指令，这样 CPU 自然就转去执行 add()函数的机器指令。事实上在编译阶段，编译器就将 main()函数和 add()函数的内存地址以标号这种相对位移的方式（专业的说法叫代码段）保存在了内存中，所以 CPU 一定能够得到这两个地址。现在关键就是第 3 和第 4 这两个问题了。

按道理说，CPU 可以将 main()函数和 add()函数所对应的栈帧（即堆栈空间）分配到内存中的任意两个位置，而随意分配的策略也有很多种，例如可以根据函数名进行散列而得到一个内存地址，然后从这个地址开始为函数内部的局部变量分配内存。但是这种方式即使从表面看都能发现很多问题，例如内存中的很多碎片空间都是有用的，可能存放了其他程序或其他函数的机器指令，可能存放了程序的全局变量或静态变量，也可能用作其他程序的堆栈空间，因此通过随机散列的方式获取内存地址将会使事情变得十分危险，不过如果真有人愿意尝试这种危险的方案，也一定能够设计出相对应的策略，假设函数堆栈真的按照这种方式去分配，最终一个程序的若干函数的堆栈空间布局就会像图 7.8 所示的这样。

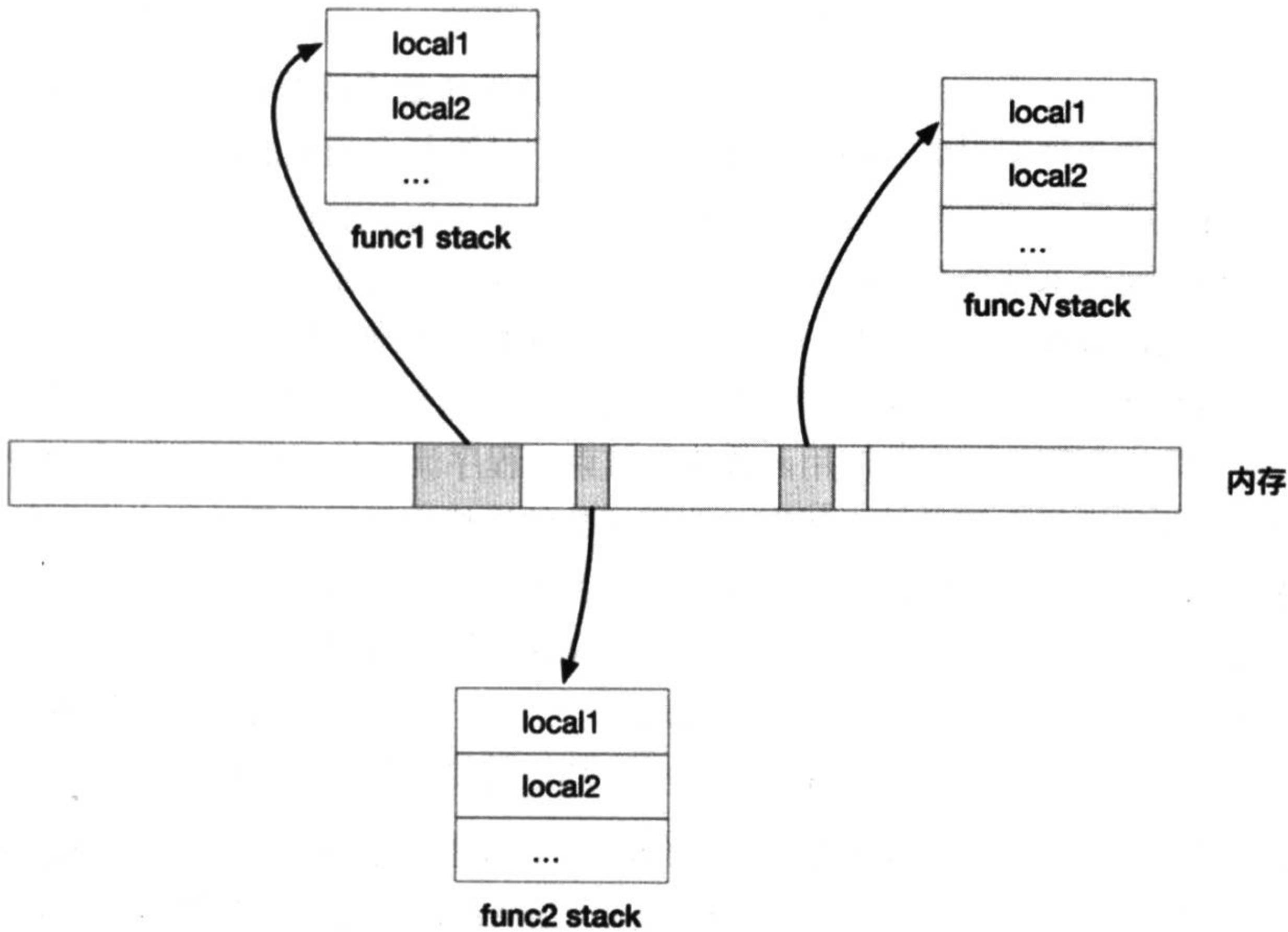


图 7.8 散列策略下的堆栈空间布局

可以看到，这种策略下的堆栈空间彼此之间是割裂的，毫无联系，因为是随机散列的嘛。这种空间分配策略最终也形成一个大容器，结构类似于 Java 语言中的 hashmap。不过空间是否割裂，这并没有任何关系，只要能用就行。但是这种碎片化的分配策略与线性的分配策略相比，除了会造成 CPU 更多的计算（要进行散列计算）外，还会造成更多的存储空间浪费。这种浪费体现在 CPU 对每一个堆栈的首地址的记忆上。图 7.9 所示是一种线性的、顺序分配的堆栈布局图。

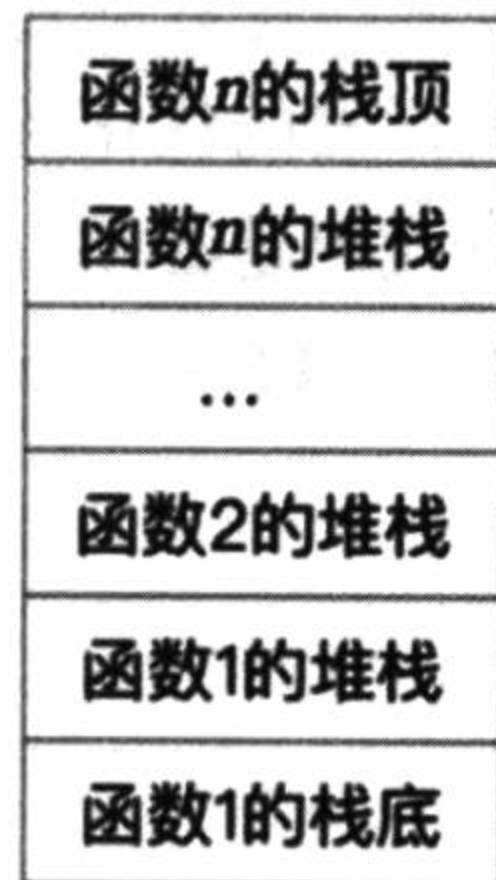


图 7.9 线性顺序分配的堆栈布局

在这种线性顺序排列的堆栈布局中，CPU 只需要知道每一个函数所需堆栈的大小，以及第一个函数的堆栈起始地址（栈底），就可以推算出后续其他函数堆栈的起始地址。事实上，对于现代成熟的计算机和各种编程语言而言，都不需要专门计算某个函数的堆栈起始地址，每一个函数的栈底地址直接等同于其调用者函数的栈顶。但是如果按照散列策略来分配堆栈空间，调用者函数的堆栈空间首末地址与被调用者函数的堆栈空间首末地址之间毫无关系，则 CPU 需要将每次散列得到的函数堆栈空间的起始地址专门保存起来，否则当程序流从调用者函数跳转到被调用者时，CPU 无法定位到被调用者函数的堆栈空间起始地址，更不用谈为被调用者函数内部的局部变量分配空间了。而当被调用函数执行结束、程序流跳转回调用函数时，CPU 需要重新定位到调用函数的堆栈空间地址。所以，调用函数的堆栈空间需要额外留出一个指针的宽度用于保存被调用函数的堆栈空间地址，而被调用函数的堆栈空间也需要额外留出一个指针的宽度用于保存调用函数的堆栈空间地址。当调用函数调用多个函数时，调用函数就可能需要留出若干空间分别保存多个被调用函数的堆栈空间地址。当然，也不一定要将这些堆栈空间的地址保存在函数的堆栈空间中，但是不管保存到哪里，都是需要消耗宝贵的内存空间的。

除了需要浪费额外的内存空间去保存函数的堆栈空间地址外，在 CPU 读取堆栈空间地址时也需要额外的计算。对于线性顺序布局的堆栈空间，当从调用函数进入被调用函数时，由于 SP 寄存器保存了调用函数的栈顶地址，因此 CPU 可以直接通过“`mov %sp, %bp`”这种纯粹寄存器数据传送的方式读取到被调用函数的栈底，在计算机内部，没有什么比纯寄存器之间的直接数据传送速度更快的了。而在随机散列策略中，由于被调用函数的堆栈空间地址被保存在内存中，因此无论如何，都避免不了 CPU 对内存的访问，而数据在内存和寄存器之间的传送速度往往要比数据直接在寄存器之间的传送效率低百倍以上。

更进一步，随机散列策略除了时间和空间上的低效，对堆栈的优化也被死死束缚。在 Java 虚拟机中，对堆栈有一项重要的优化策略是堆栈重叠（调用方法的操作数栈与被调用方法的局部变量表重叠），这种堆栈重叠的方式能够避免调用函数内部局部变量向被调用函数堆栈复制被调用函数的入参，从而能够减少相当可观的内存数据复制。这种现象不仅 Java 虚拟机中会有，其他编程语言中也会有，例如一些主流编译器的优化选项都支持这种策略。更有甚者，当在开发底层驱动器或者视频核心逻辑时，部分代码是必须直接用汇编写的，这时候就可以手动直接让堆栈重叠，例如上面的 `main()` 函数和 `add()` 函数，在 Linux 平台上编译后得到的汇编程序如下：

```

main:
    pushl    %ebp
    movl %esp, %ebp
    subl $32, %esp

    // 分配局部变量并初始化
    movl $5, 20(%esp)
    movl $3, 24(%esp)

```

```

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

call add
movl%eax, 28(%esp)

movl$0, %eax
leave
ret

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

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

movl%eax, -4(%ebp)
movl-4(%ebp), %eax
leave
ret

```

上面这段汇编脚本比较中规中矩,当 main()函数调用 add()函数并向 add()函数传递参数时,老老实实地将 add()函数的入参复制了一遍,由于 add()函数包含两个人参,因此一共使用了 4 条指令完成两个参数的压栈。

如果这段程序由人工编写,则可以进行比较激进的优化,省略复制参数的 4 条指令。在 add() 函数中读取入参时,直接基于 add()堆栈栈底往上偏移,进入 main()函数的堆栈空间读取入参。激进优化后的汇编指令如下:

```

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

//分配局部变量并初始化
movl$5, 4(%esp)
movl$3, 8(%esp)

call add

```

```

    movl %eax, 12(%esp)

    movl $0, %eax
    leave
    ret

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

    //从 main() 函数堆栈中直接读取局部变量
    movl 12(%ebp), %eax
    movl 16(%ebp), %edx

    addl %edx, %eax
    movl %eax, -4(%ebp)
    movl -4(%ebp), %eax
    leave
    ret

```

修改之后，main()函数少了 4 条参数复制的指令，同时 main()函数的堆栈空间只需要分配 16 字节，而原来是 32 字节。add()函数直接从 main()函数的堆栈中入参的原始内存位置处读取数据。这种优化方案所带来的性能提升还是相当可观的，这就是线性顺序内存分配所带来的好处，只要你想得到，就能做得到。而采用随机散列的方式分配函数堆栈，无法享受这些优化措施所带来的性能提升红利。

由于线性顺序分配栈帧（堆栈空间）有这么多好处，因此成为设计栈帧容器的上佳策略。而线性顺序存储元素成员的容器有好几种，最主要的两种容器分别是堆式容器（队列）和栈式容器，到了这里大家都知道了，堆与栈的区别就是前者是 FIFO（先进先出），后者是 FILO（先进后出）。由于函数调用过程是链式路径，越是最后调用的函数，其栈帧（堆栈空间）就越分配得晚；越是最后调用的函数，其栈帧空间也越早被释放，所以自然就使用栈式结构来作为栈帧的容器。这就是堆栈的由来。

到了这里，还有个问题没有解决，那就是操作系统一般是从高位地址开始往下扩展堆栈空间。堆栈作为一个内存容器，并不一定非得从高位地址开始往下增长，也可以从低地址开始往高地址内存方向增长，有部分操作系统就是这么干的。而更重要的一点，对于一个应用程序，操作系统明显地会将程序的内存空间区分为堆和栈（当然还有代码区等，这些对于本主题并不是十分重要，略过不表），堆区往高地址方向增长，栈区往低地址方向增长，这又是为何呢？换句话说，操作系统完全可以不区分所谓的堆区和栈区，大家都只是内存中的一部分空间而已，管它是啥结构，一股脑儿全部分配在内存中，也不是不可行。软件程序中的所谓堆和栈的容器

就全部分配在堆内存中，也没见有啥问题。

事实上，这里需要区分操作系统和应用程序的内存结构。对于操作系统，其上面会运行若干应用程序，如果操作系统不将应用程序的堆内存和栈内存区分开来，那么内存空间就会相互“打架”，并且会破坏函数栈帧在空间上的线性连续性。还是拿上面的 main() 函数和 add() 函数举例，假设 main() 函数的堆栈从内存为 0 的地址处开始分配空间（这种情况事实上是不存在的，这里仅仅为了举例方便），main() 函数堆栈需要 32 字节的空间，因此当 main() 函数的堆栈空间分配完之后，下一个数据只能从第 32 内存单元开始分配空间。假设这时 main() 函数调用了 alloc() 函数向操作系统申请内存空间，那么操作系统就会从第 32 个内存单元开始分配空间（注意，这种假设的前提是操作系统没有将应用程序的内存区分为堆内存和栈内存），等分配完了 alloc() 函数所申请的内存空间后，main() 函数开始调用 add() 函数，此时 CPU 会为 add() 函数分配栈空间，而栈空间的起始位置就是刚才 alloc() 函数所申请的内存的末端位置。这样一来，main() 函数与 add() 函数的栈帧（堆栈空间）在内存空间上的线性顺序性就被破坏了，如此一来，调用函数与被调用函数的栈帧在空间上便处于割裂状态，这与上面所举的散列方案所产生的内存空间布局也没什么两样。所以，这就要求存储程序函数栈帧的容器——堆栈，在内存空间分布上必须是完全线性的，中间不能出现任何空间分隔。既然如此，操作系统就必须将应用程序的堆内存空间与栈内存空间完全隔离开来，将此作为操作系统治理内存的基本宗旨，而这也是程序函数堆栈这个容器级别的数据结构与软件程序中的堆栈容器的数据结构之间最大的不同点。

在软件程序中设计的所谓堆栈容器，栈中的元素也可以是容器，但是栈中往往仅仅保存对应元素的容器的指针，指针所指向的容器在堆中开辟空间，这便导致各个元素所指向的容器空间之间并不是连续的线性分布，例如，下面这个程序是用 C 语言所编写的堆栈容器，堆栈的元素是栈帧，栈帧也是一个容器：

清单：stack.c

作用：使用 C 语言演示堆栈布局

```
#include <stdio.h>
#include <stdlib.h>

//栈帧结构体，假设所有函数的栈帧都一样，都仅包含 3 个布局变量
typedef struct Frame
{
    struct Frame *pre;//指向下一个栈帧
    int local1;
    int local2;
    short local3;
}frame;

//堆栈
```

```

typedef struct Stack
{
    frame *base; //栈底
    frame *top; //栈顶
    int size; //堆栈大小
} stack;

void initStack(stack *stack); //初始化
void push(stack *mystack, frame *nextFrame);
void pop(stack *mystack);
void iterate(stack *mystack);

int main(int argc, char const *argv[])
{
    //定义一个堆栈
    stack *mystack = (stack*)malloc(sizeof(struct Stack));
    printf("sizeof(struct Frame) = %lu\n", sizeof(struct Frame));

    //初始化堆栈
    initStack(mystack);

    //压入第 1 个栈帧
    frame *myframe = (frame*)malloc(sizeof(struct Frame));
    printf("当前栈帧起始地址是 %p\n", myframe);
    myframe->local1 = 1;
    myframe->local2 = 2;
    myframe->local3 = 3;
    push(mystack, myframe);

    //压入第 2 个栈帧
    myframe = (frame*)malloc(sizeof(struct Frame));
    printf("当前栈帧起始地址是 %p\n", myframe);
    myframe->local1 = 5;
    myframe->local2 = 6;
    myframe->local3 = 7;
    push(mystack, myframe);

    //压入第 3 个栈帧
    myframe = (frame*)malloc(sizeof(struct Frame));
    printf("当前栈帧起始地址是 %p\n", myframe);
    myframe->local1 = 8;
    myframe->local2 = 9;
    myframe->local3 = 10;
    push(mystack, myframe);

    //遍历堆栈
    iterate(mystack);
}

```

```
//出栈
pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);

return 0;
}

//初始化堆栈
void initStack(stack *stack)
{
    stack->base = NULL;
    stack->top = NULL;
    stack->size = 0;
}

//压栈
void push(stack *mystack, frame *nextFrame)
{
    //如果当前堆栈为空，则将栈顶和栈底同时指向第一个栈帧
    if(mystack->size == 0){
        mystack->base = nextFrame;
        mystack->top = nextFrame;
    }

    nextFrame->pre = mystack->top;//当前栈帧的上一个栈帧是栈顶
    mystack->top = nextFrame;//栈顶变成了当前栈帧

    mystack->size ++;//将堆栈数量加 1
}

//出栈
void pop(stack *mystack)
{
    int size = mystack->size;
    if(size == 0)
    {
        printf("%s\n", "当前堆栈为空，不能出栈");
        return;
    }
}
```

```

    }

    frame *currFrame = mystack->top;//获取当前堆栈顶部栈帧
    mystack->top = mystack->top->pre;//将当前栈顶指向下一个栈帧

    //释放栈顶栈帧
    free(currFrame);
    mystack->size--;
    printf("成功弹出一个栈顶栈帧\n\n");
}

//遍历堆栈
void iterate(stack *mystack)
{
    if(mystack->size == 0){
        printf("%s\n", "当前堆栈为空，遍历终止");
        return;
    }

    int size = mystack->size;
    printf("*****当前堆栈共有 %d 个栈帧*****\n", size);
    frame *currFrame = mystack->top;
    while(size > 0)
    {
        printf("当前是第 %d 个栈帧\n", size);
        printf(" local1 = %d\n", currFrame->local1);
        printf(" local2 = %d\n", currFrame->local2);
        printf(" local3 = %d\n", currFrame->local3);

        currFrame = currFrame->pre;
        size--;
    }
}

```

本示例程序中定义了两个结构体，分别是 `stack` 与 `frame`，`stack` 是堆栈容器，里面包含栈底、栈顶和堆栈大小 3 个成员项。`frame` 则用于构建栈帧。运行 `main()` 函数，会打印如下内容：

```

sizeof(struct Frame) = 24
当前栈帧起始地址是 0x7fa3d3c02000
当前栈帧起始地址是 0x7fa3d3c03290
当前栈帧起始地址是 0x7fa3d3c032b0
*****当前堆栈共有 3 个栈帧*****
当前是第 3 个栈帧
    local1 = 8
    local2 = 9
    local3 = 10
当前是第 2 个栈帧

```

```
local1 = 5
local2 = 6
local3 = 7
当前是第 1 个栈帧
local1 = 1
local2 = 2
local3 = 3
成功弹出一个栈顶栈帧

*****当前堆栈共有 2 个栈帧*****
当前是第 2 个栈帧
local1 = 5
local2 = 6
local3 = 7
当前是第 1 个栈帧
local1 = 1
local2 = 2
local3 = 3
成功弹出一个栈顶栈帧

*****当前堆栈共有 1 个栈帧*****
当前是第 1 个栈帧
local1 = 1
local2 = 2
local3 = 3
成功弹出一个栈顶栈帧

当前堆栈为空，遍历终止
当前堆栈为空，不能出栈
```

注意看程序所打印出来的 3 个栈帧的内存地址，这 3 个地址之间没有任何联系，这是因为这 3 个栈帧完全是操作系统随机分配的内存空间。所以说，软件程序里所定义的栈结构，与操作系统层面的程序函数堆栈结构之间还是有差别的。本例动态创建的 3 个栈帧，内存布局完全是无序和随机的，它们的内部空间结构如图 7.10 所示。

而程序函数堆栈中的各个栈帧之间是完全紧密顺序排列的，栈帧之间不可能出现别的数据。

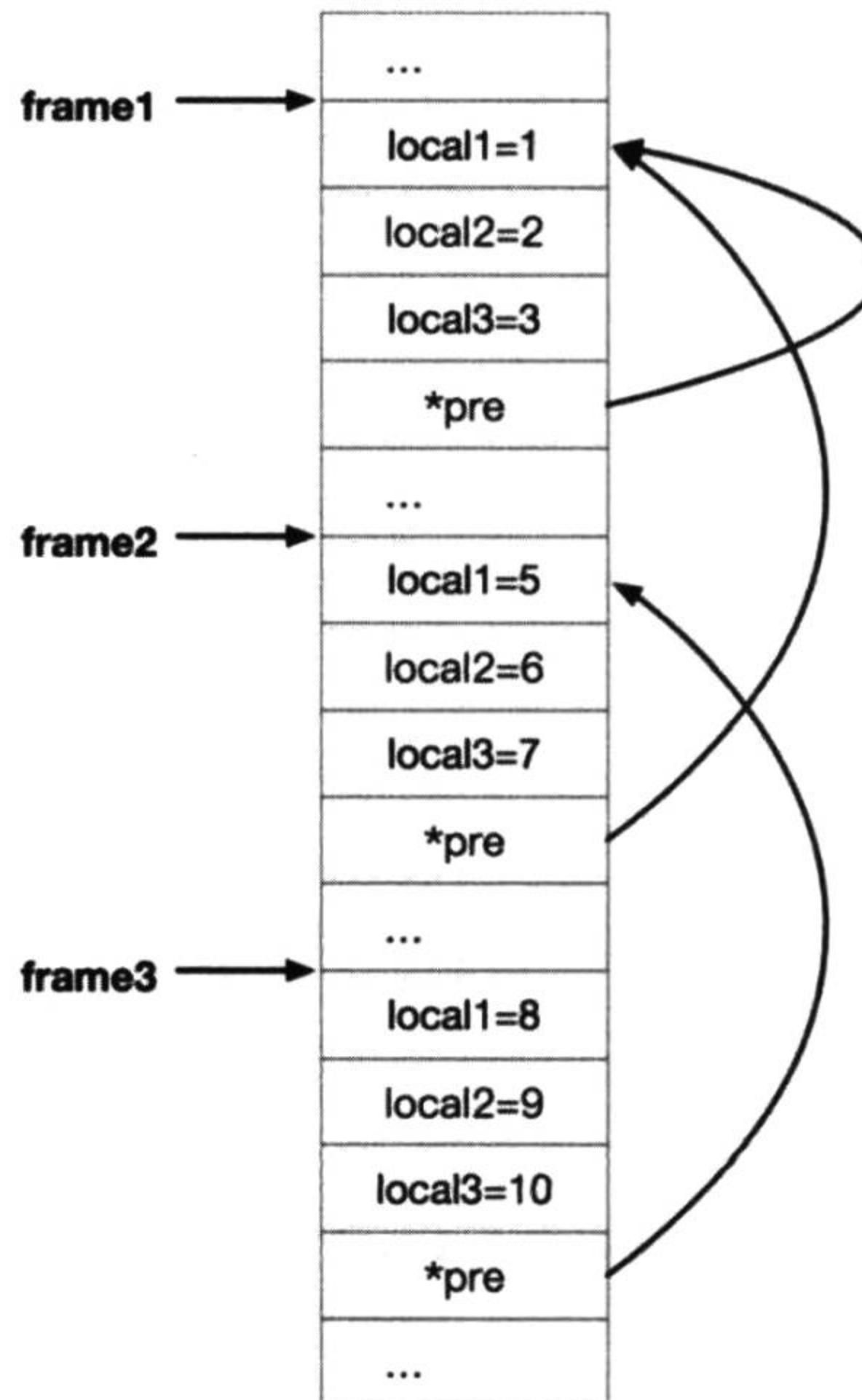


图 7.10 C 语言所模拟的随机分配的堆栈空间布局

当然，使用软件其实也能模拟出程序函数堆栈的空间分布，将上面的程序进行改造，使之分配的栈帧彼此首尾相连，改造后的程序如下：

清单：stack.c

作用：使用 C 语言演示堆栈布局

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//堆栈初始大小
#define DEFAULT_SIZE 2
//堆栈扩容因子
#define EXPAND_FACTORY 1.75

//栈帧结构体，假设所有函数的栈帧都一样，都仅包含 3 个局部变量
typedef struct Frame
{
    struct Frame *pre;//指向下一个栈帧
    int local1;
    int local2;
    short local3;
```

```
 }frame;

//堆栈
typedef struct Stack
{
    frame *base;//栈底
    frame *top;//栈顶
    int size;//堆栈真实大小
    int maxSize;//堆栈最大大小
}stack;

void initStack(stack *stack);//初始化
void push(stack *mystack, frame *nextFrame);
void pop(stack *mystack);
void expand(stack *mystack);//扩容
void iterate(stack *mystack);

int main(int argc, char const *argv[])
{
    //定义一个堆栈
    stack *mystack = (stack*)malloc(sizeof(struct Stack));

    //初始化堆栈
    initStack(mystack);

    //压入第1个栈帧
    frame myframe;
    myframe.local1 = 1;
    myframe.local2 = 2;
    myframe.local3 = 3;
    push(mystack, &myframe);

    //压入第2个栈帧
    myframe.local1 = 5;
    myframe.local2 = 6;
    myframe.local3 = 7;
    push(mystack, &myframe);

    //压入第3个栈帧
    myframe.local1 = 8;
    myframe.local2 = 9;
    myframe.local3 = 10;
    push(mystack, &myframe);

    //压入第4个栈帧
    myframe.local1 = 11;
    myframe.local2 = 12;
```

```
myframe.local3 = 13;
push(mystack, &myframe);

//遍历堆栈
iterate(mystack);

//出栈
pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);

free((void *)mystack->base);

return 0;
}

//初始化堆栈
void initStack(stack *stack)
{
    stack->size = 0;
    stack->maxSize = DEFAULT_SIZE;

    //初始化堆栈空间
    long address = (long)malloc(sizeof(struct Frame) * DEFAULT_SIZE);
    stack->base = (frame*)address;
    stack->top = (frame*)address;
}

//压栈
void push(stack *mystack, frame *nextFrame)
{
    //扩容
    expand(mystack);

    frame *newFrame;
    frame *top = mystack->top;

    //如果当前堆栈为空，则将栈顶和栈底同时指向第一个栈帧
    if(mystack->size == 0) {
        newFrame = top;
    } else{
```

```

    newFrame = ++ top;
}

//frame *newFrame = (frame*) (++ top); //将栈顶指针往前移动一个栈帧的距离
printf("压入第 %d 个栈帧, 当前栈帧起始地址是 %p\n\n", mystack->size + 1,
newFrame);
newFrame->local1 = nextFrame->local1;
newFrame->local2 = nextFrame->local2;
newFrame->local3 = nextFrame->local3;

newFrame->pre = mystack->top; //当前栈帧的上一个栈帧是栈顶
mystack->top = newFrame; //栈顶变成了当前栈帧

mystack->size++; //将堆栈数量加 1
}

//出栈
void pop(stack *mystack)
{
    int size = mystack->size;
    if(size == 0)
    {
        printf("%s\n", "当前堆栈为空, 不能出栈");
        return;
    }

    frame *currFrame = mystack->top; //获取当前堆栈顶部栈帧
    mystack->top = mystack->top->pre; //将当前栈顶指向下一个栈帧

    //释放栈顶栈帧
    mystack->size--;
    printf("成功弹出一个栈顶栈帧\n\n");
}

//扩容
void expand(stack *mystack)
{
    if(mystack->size == mystack->maxSize) {
        //扩容
        int maxSize = mystack->size * EXPAND_FACTORY; //计算扩容后的大小
        long address = (long)malloc(sizeof(struct Frame) * maxSize);
        memcpy((void *)address, mystack->base, mystack->size * sizeof(struct
Frame));
    }

    //重定向栈顶和栈底
    mystack->base = (frame*)address;
    mystack->top = (frame*) (address + (mystack->size - 1) * sizeof(struct
Frame));
}

```

```

Frame));
    mystack->maxSize = maxSize;

    //重新关联当前栈帧与上一个栈帧之间的关系
    for(int i = 1; i < mystack->size; i++)
    {
        frame *currFrame = (frame*)address;
        frame *tmp = (frame*)address;
        frame *nextFrame = ++tmp;//获取下一个栈帧
        nextFrame->pre = currFrame;

        address = (long)nextFrame;
    }

    printf(">>>>扩容成功。申请的堆栈地址是 %p, 扩容后的堆栈栈顶地址是 %p\n",
mystack->base, mystack->top);
}

//遍历堆栈
void iterate(stack *mystack)
{
    if(mystack->size == 0){
        printf("%s\n", "当前堆栈为空，遍历终止");
        return;
    }

    int size = mystack->size;
    printf("*****当前堆栈共有 %d 个栈帧*****\n", size);
    frame *currFrame = mystack->top;
    while(size > 0)
    {
        printf("当前是第 %d 个栈帧，其地址是 %p\n", size, currFrame);
        printf(" local1 = %d\n", currFrame->local1);
        printf(" local2 = %d\n", currFrame->local2);
        printf(" local3 = %d\n", currFrame->local3);

        currFrame = currFrame->pre;
        size--;
    }
}

```

改造后的程序，栈帧内存空间分配算法有所改变，变成堆栈这个大容器统一分配一整块连续的内存，然后在这块内存空间里按顺序存放栈帧。当堆栈容量不够时，通过扩容重新分配一整块连续的内存空间，然后再重新按顺序进行排列。

现在运行 main() 函数，会打印出如下内容：

压入第 1 个栈帧，当前栈帧起始地址是 0x7f96e1c03290

压入第 2 个栈帧，当前栈帧起始地址是 0x7f96e1c032a8

>>>> 扩容成功。申请的堆栈地址是 0x7f96e1c032c0，扩容后的堆栈栈顶地址是 0x7f96e1c032d8

压入第 3 个栈帧，当前栈帧起始地址是 0x7f96e1c032f0

>>>> 扩容成功。申请的堆栈地址是 0x7f96e1c03310，扩容后的堆栈栈顶地址是 0x7f96e1c03340

压入第 4 个栈帧，当前栈帧起始地址是 0x7f96e1c03358

*****当前堆栈共有 4 个栈帧*****

当前是第 4 个栈帧，其地址是 0x7f96e1c03358

```
local1 = 11  
local2 = 12  
local3 = 13
```

当前是第 3 个栈帧，其地址是 0x7f96e1c03340

```
local1 = 8  
local2 = 9  
local3 = 10
```

当前是第 2 个栈帧，其地址是 0x7f96e1c03328

```
local1 = 5  
local2 = 6  
local3 = 7
```

当前是第 1 个栈帧，其地址是 0x7f96e1c03310

```
local1 = 1  
local2 = 2  
local3 = 3
```

成功弹出一个栈顶栈帧

*****当前堆栈共有 3 个栈帧*****

当前是第 3 个栈帧，其地址是 0x7f96e1c03340

```
local1 = 8  
local2 = 9  
local3 = 10
```

当前是第 2 个栈帧，其地址是 0x7f96e1c03328

```
local1 = 5  
local2 = 6  
local3 = 7
```

当前是第 1 个栈帧，其地址是 0x7f96e1c03310

```
local1 = 1  
local2 = 2  
local3 = 3
```

成功弹出一个栈顶栈帧

*****当前堆栈共有 2 个栈帧*****

当前是第 2 个栈帧，其地址是 0x7f96e1c03328

```
local1 = 5  
local2 = 6  
local3 = 7
```

当前是第 1 个栈帧，其地址是 0x7f96e1c03310

```
local1 = 1
local2 = 2
local3 = 3
```

成功弹出一个栈顶栈帧

*****当前堆栈共有 1 个栈帧*****

当前是第 1 个栈帧，其地址是 0x7f96e1c03310

```
local1 = 1
local2 = 2
local3 = 3
```

成功弹出一个栈顶栈帧

注意看程序所打印出来的 4 个栈帧的内存地址，现在这 4 个地址之间是有联系的，第 4 个栈帧的内存地址与第 3 个栈帧内存地址之差是 0x18，正好是 struct Frame 这个结构体的大小。同样，第 3 个栈帧内存地址与第 2 个栈帧的内存地址之差也是 0x18，第 2 个栈帧内存地址与第 1 个栈帧的内存地址之差也是 0x18，由此可见栈帧之间的确是线性按顺序分配空间的。这种堆栈的空间布局基本与程序函数调用的堆栈空间布局是一样的，布局结构如图 7.11 所示。

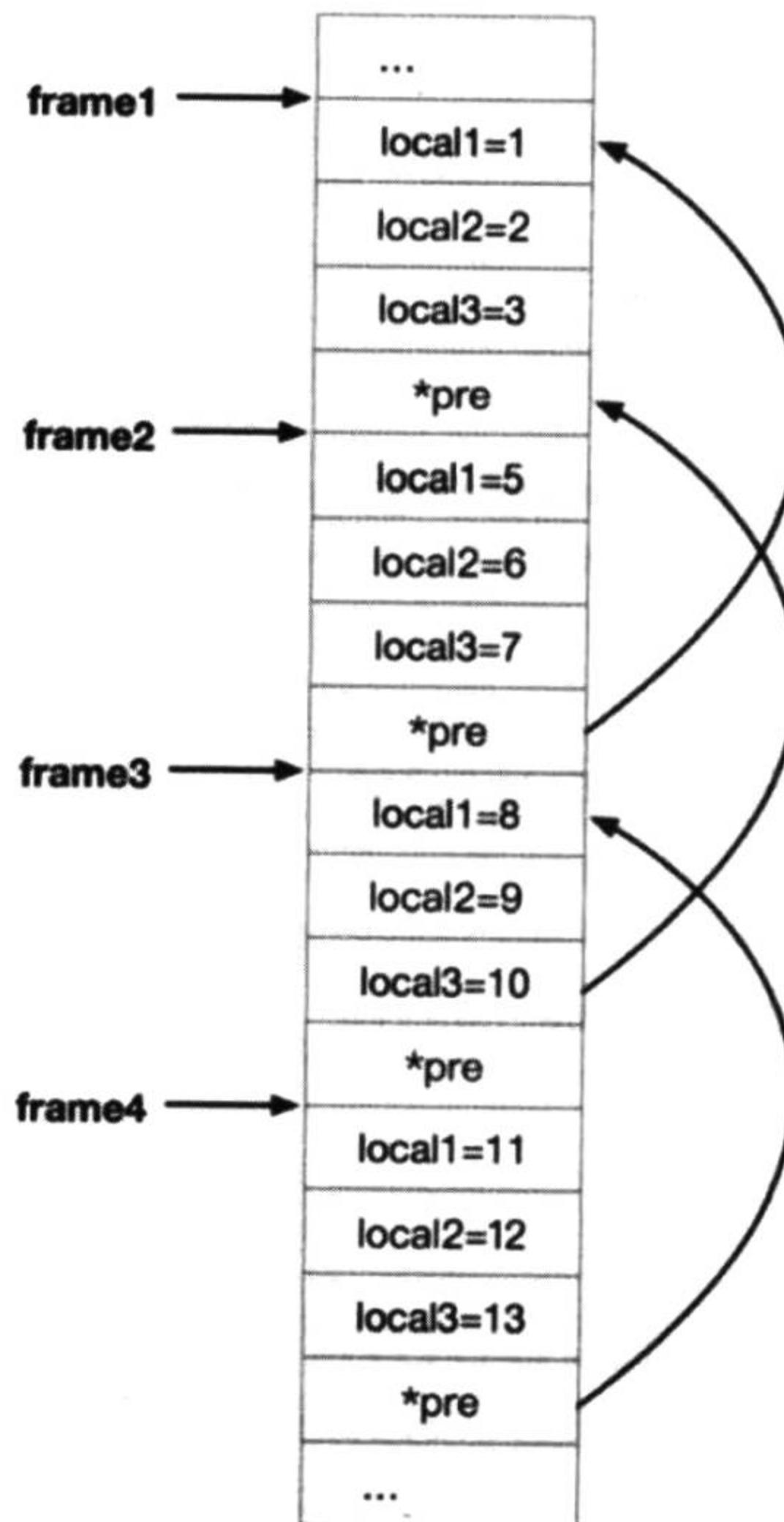


图 7.11 C 语言所模拟的线性顺序分配的堆栈空间布局

综上所述，程序函数调用的堆栈的成员元素是栈帧，“堆栈”作为一个容器，其内部存储的是“栈帧”这种元素。而栈帧又是一个小容器，存储函数内部的局部变量。函数调用堆栈与软件程序里所设计的堆栈结构的最大不同之处在于，函数调用堆栈的成员元素（栈帧）之间在空间上是连续分布的，而软件程序里所设计的堆栈结构，其成员元素往往并不相连。

7.3.2 硬件对堆栈的支持

为了支持应用程序函数栈帧所形成的堆栈结构，系统设计的老前辈们还从硬件上予以支持，专门设计了两个硬件寄存器——SP 和 BP，用于存储当前函数栈帧的栈底和栈顶。

在上面使用 C 语言模拟程序堆栈和栈帧的示例中，在结构体 stack 中专门定义了两个指针 base 和 top，分别指向堆栈的底部栈帧和顶部栈帧，注意这里不是栈顶指针和栈底指针哟！在操作系统层面，栈底和栈顶指针是针对栈帧而言的，例如，该示例中有 4 个栈帧 frame，如果是真实的函数栈帧，则每一个栈帧都会有一个栈顶和栈底指针，用于标识函数堆栈的起始地址和末端地址。而使用软件实现的栈帧，由于其数据结构是固定大小的，因此通过指针持有其起始地址，自然就知道其末端地址，何况使用软件实现堆栈和栈帧时，也不需要知道末端地址，直接通过强类型转换就能读取到栈帧各个变量的值，所以大部分使用软件实现的所谓堆栈，其栈底指针和栈顶指针更多的是用来指向整个堆栈容器的底部栈帧和顶部栈帧。如图 7.12 所示。

对于软件实现的堆栈，只需要获取指向堆栈顶部栈帧的指针，就能实现压栈和出栈的功能。而程序函数调用时为函数所开辟的堆栈空间，由于不同的函数其内部局部变量不同，因此栈帧空间结构不同，大小不同，因此必须要分别保存函数栈帧的栈顶地址与栈底地址。如果该示例中的 4 个栈帧是真实的函数调用堆栈，则 CPU 会分别使用 SP 和 BP 这两个寄存器存储当前位置位于堆栈顶部的栈帧的栈顶和栈底，结构示意图如图 7.13 所示。

其实如果没有 SP 与 BP 这两个硬件寄存器，系统也能完成对堆栈顶部函数栈帧的栈顶与栈底标记，但是那样势必要将栈顶与栈底地址保存到内存，而内存访问的性能与寄存器的访问性能完全不在一个档次。所以函数栈帧的空间分配其实用纯软件也能实现，只不过借助于两个特定的寄存器硬件，使得这一效率更高而已。

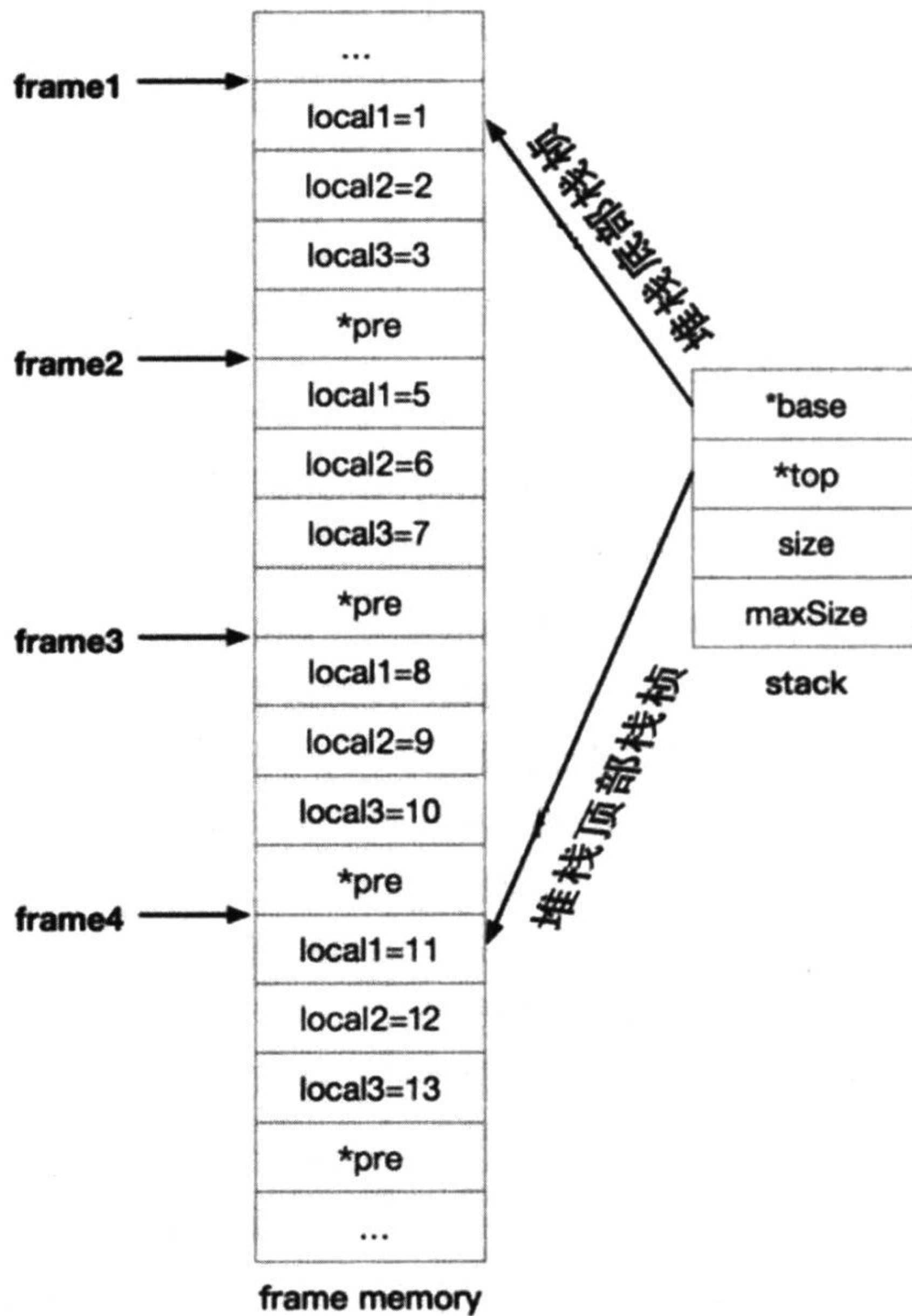


图 7.12 软件实现的堆栈，堆栈底部和顶部指针

当 CPU 完成一个函数的执行之后，程序流会跳转到当前函数的调用方，同时，CPU 需要回收掉当前函数的栈帧空间，并使 SP 和 BP 这两个寄存器重新指向调用者函数的栈顶和栈底。CPU 回收函数栈帧空间很简单，只需要将 SP 往 BP 的方向移动一定距离即可，但是 CPU 如何将 SP 与 BP 重定向至调用者函数的栈顶和栈底呢？其实也很简单，由于真实的程序函数调用堆栈里的各个栈帧都是首尾相连的，被调用函数的栈底就是调用函数的栈顶，因此当 CPU 完成函数调用后，只需将被调用函数的 SP 指向被调用函数的 BP，由于被调用函数的 BP 恰好就是调用函数的 SP，因此这就相当于 CPU 将 SP 重新指向到了调用者函数的栈顶。所以剩下的事就是 CPU 如何将 BP 也恢复到调用者函数的栈底。

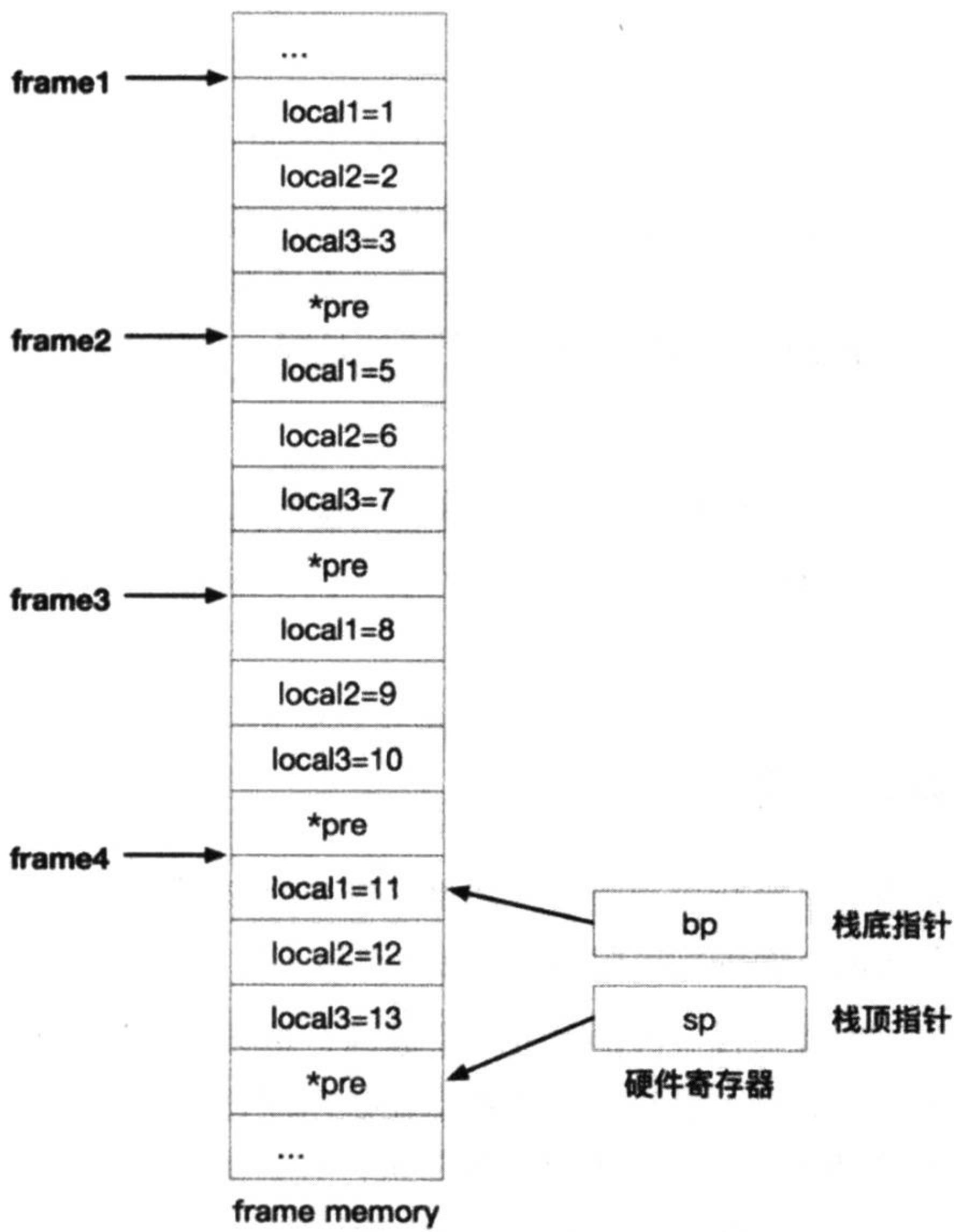


图 7.13 真实的硬件实现的栈底和栈顶指针

在上面使用 C 语言模拟函数调用堆栈和栈帧的示例中，为了实现出栈后能够定位到上一个栈帧，在每一个栈帧中都存储了一个 pre 指针，每次压入一个新的栈帧时，新的栈帧的 pre 指针就指向原本位于堆栈顶部的栈帧的起始地址，所以，当将位于堆栈顶部的栈帧出栈时，只需通过读取 pre 指针的值，就能定位到上一个栈帧，从而将堆栈的栈顶指针 top 重定向至新的堆栈顶部的栈帧。其实，在本示例中（特指上述改造后的示例程序，在改造后的示例程序中，各个栈帧是线性顺序分布的，首尾相连），由于所模拟的各个栈帧的数据结构相同，因此各个栈帧所占内存空间大小也相同，所以完全可以不依赖 pre 指针就可以直接计算出相邻的栈帧位置。

但是在机器的函数调用堆栈层面，由于不同的函数定义完全不同，因此函数的栈帧结构也完全不同，大小更不可能相同，所以当进行出栈时，想要将 BP 恢复至调用者函数的栈底，如果当前函数没有保存调用者函数的栈底地址，是根本无法恢复回去的。所以，在真实的函数调用中，物理机器必定会将调用函数的栈底地址压入到被调用函数的堆栈之中。因而，使用 C 语言及其他很多语言编写的程序被编译后，所生成的每一个函数的机器指令必定以下面这两条指

令作为“起式”：

```
push %bp
move %sp, %bp
```

第一条指令就是往被调用函数的堆栈中压入调用函数的栈底地址，这便是物理机器对程序调用堆栈的压栈与出栈的支持及算法，说起来，实在是简单至极，与使用 C 语言所模拟的堆栈和栈帧的示例程序中使用 pre 指针来恢复至上一个栈帧是同一个道理。

大道至简啊！

由此更加可以看出来，计算机程序调用的压栈与出栈机制，即使不使用 BP 和 SP 这两个硬件寄存器，也是完全可以实现的，使用纯软件算法一样可以实现。SP 与 BP 的加入，纯粹是为了提升效率。

7.3.3 栈帧开辟与回收

前面讲过，操作系统为了简化内存治理，便将程序的内存空间划分成堆和栈，并且栈必须在内存空间上是连续分布的，而堆则可以随机分配。

其实，操作系统强行将堆与栈分开的另外一个原因，想必也是由两者的空间分布的特点不同所引起的。使用 C/C++ 语言开发程序的一大难点便是内存管理（另一大难点自然就是多线程与并发了），内存管理的难点在于一不小心就容易造成内存泄漏，而造成内存泄漏的关键原因就是忘记释放已经申请的堆内存空间。对于使用 malloc() 接口所申请的内存空间，如果不使用 free() 去登记释放，除非程序运行结束，否则这部分申请的内存是不会被自动回收的。如果 malloc() 接口在一个多进程/多线程环境中被调用，那么要不了多久，操作系统内存空间便会耗尽而使系统崩溃。

下面这个程序就会造成内存泄漏：

清单：malloc_test.c

作用：演示内存泄漏

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    long k;
    while(1)
    {
        int *address = (int*)malloc(sizeof(int)*10000000);
```

```

    k++;
    printf("申请内存成功, k= %lu\n", k);
}

return 0;
}

```

对于简单的程序设计，还能看出来哪些变量只申请了内存而没有释放，而对于一个大型的程序而言，经过层层封装，一不小心，便很容易将内存释放的事情遗忘掉，所以开发 C/C++ 程序真是陷阱重重。

相比于堆内存空间管理的严酷要求，栈的开辟和释放就显得十分简单。想开辟一个栈空间，只需要调用如下指令：

```
sub $32, %sp
```

调用这条指令后，直接就将 SP 指针往前移动 32 个存储单元的距离，相当简单。

而当一个函数执行完后，需要释放其对应的栈帧空间时，也只需调用下面的指令：

```
add $32, %sp
```

将 SP 指针再移回去 32 字节的距离就行了。当然，对于高级编程语言，这条指令基本是看不到的，因为直接被封装在了 leave 指令里了。

总之，应用程序申请和释放“栈”空间，相比于申请和释放“堆”空间，要简单得多。之所以如此简单，主要得益于栈的结构——必须是连续线性分布。这也是操作系统实现内存治理时要求将堆与栈空间分开治理的原因，试想一下，如果栈与堆没有分开治理而是合在了一起，那么当应用程序申请分配栈空间时势必也要使用类似于 malloc() 这样的接口，而当函数调用完成，操作系统需要回收该函数所占用的栈帧空间时，势必也需要调用类似于 free() 这样的接口，如此一来，在函数栈帧压栈与出栈的过程中，还要涉及系统调用，那效率是相当低啊！

所以，不管是为了能够高效实现栈帧的压栈与出栈而将堆与栈空间相分离，还是纯粹为了堆与栈分开治理而发现了分离后能够享受到高效的压栈与出栈算法所带来的性能提升，总之，将应用程序的堆与栈隔离开来，并且规定栈帧在堆栈空间内必须线性连续分布，而且还借助于 SP 与 BP 这两个硬件寄存器来实现高效的压栈与出栈指令，都是历史上那些伟大的软件设计师经过千锤百炼后所得出的最优方案，是精华中的精华。

7.3.4 堆栈大小与多线程

现代操作系统都支持多进程与多线程机制，允许同一个应用程序同时运行若干进程或者线程，作为虚拟操作系统的 JVM 自然也是支持多进程与多线程应用的。

沿着上文对堆栈与栈帧认识的思路，可以继续往下推断在多进程与多线程机制下的栈空间分配机制，对于这些机制，不用看任何理论，也不用看操作系统原理相关的书，完全借助于上文所得到的结论便可分析。前面讲过，操作系统会将一个应用程序的内存空间划分成堆和栈这两大部分，函数的栈帧只能在栈空间内分配。但是在多进程/多线程环境下，就会有一个关键的问题：在多进程或多线程的环境下，操作系统会为一个进程/线程单独划分一个栈空间，还是会让一个应用程序的所有进程/线程共同使用同一个栈空间呢？说白了就是，在多线程环境下，操作系统是否会进一步将应用程序的栈空间划分成更多的子块？

我们可以先从一个假设出发去论证。假设操作系统不将应用程序的栈空间按照线程的粒度进行细分，而是让全部线程共同使用同一个堆栈空间，那么随着一个应用程序的多个线程交替执行，该应用程序的堆栈空间布局就可能会变成如图 7.14 所示的样子。

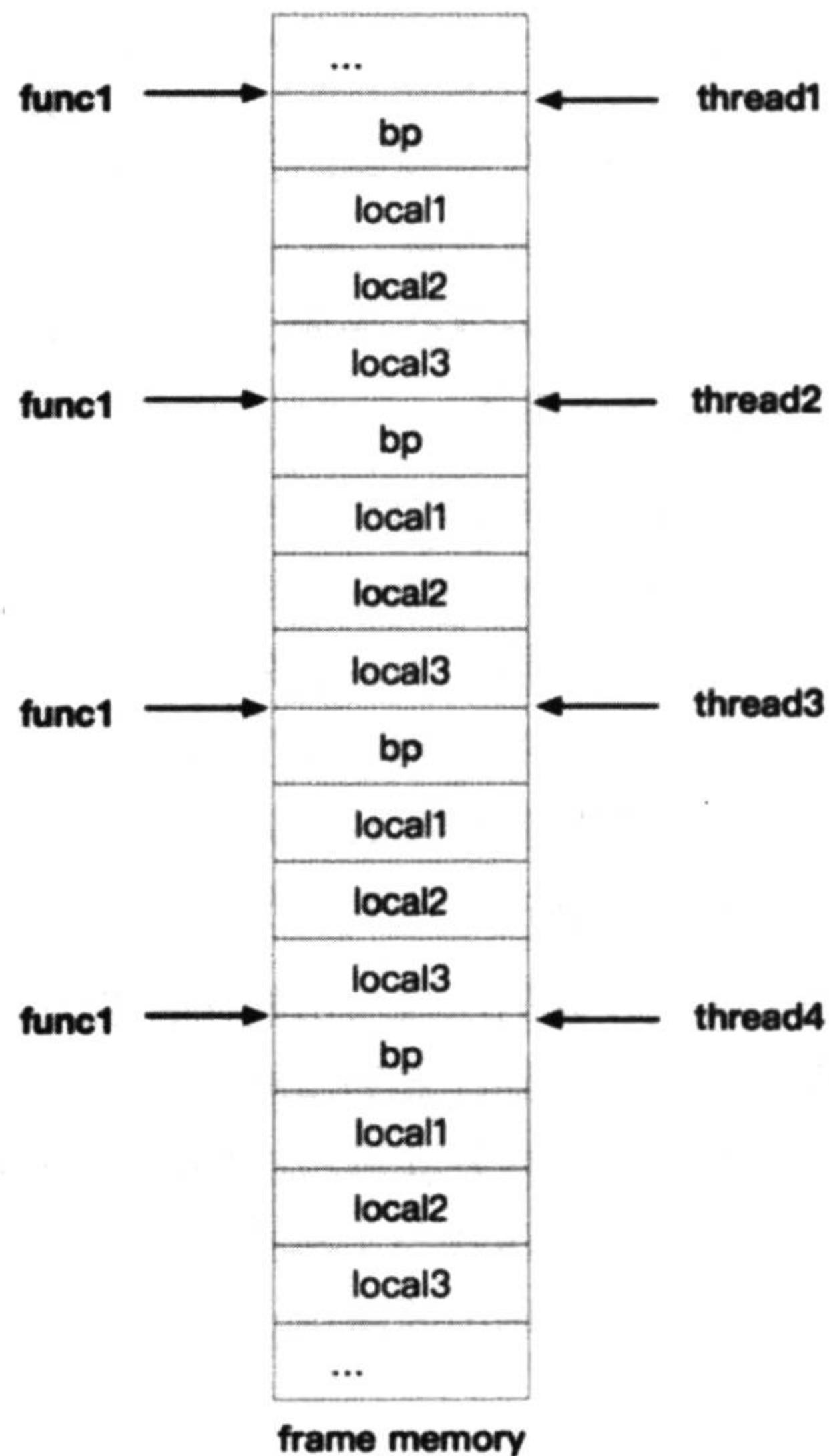


图 7.14 多线程共用同一个堆栈空间时的一种可能的栈帧布局

注：为了简化问题分析，本图在栈帧之间省略了除 bp 以外的其他堆栈所必需的数据。

假设该应用程序同时运行了4个线程，并且恰好这4个线程依次执行了func1()这个函数，那么此时的堆栈内的栈帧空间布局就会如图7.14所示。

当出现这种内存布局时，就违反了操作系统关于堆栈空间布局的最核心的原则：栈帧必须是连续分布的。例如对于线程1，其func1的下一个栈帧应该是func2（假设func1调用了func2），但是现在却变成了线程2的func1()函数的栈帧。

这样不连续的内存布局无论是对新函数的栈帧分配还是函数执行结束时的栈帧回收，都会造成相当大的问题，程序的堆栈空间变得杂乱无章，各个线程的堆栈空间相互覆盖重写，世界一片大乱。

所以，为了支持多线程应用，操作系统必须为应用程序的每一个线程专门划分一块连续的区域，各个线程的函数调用堆栈就在各自的内存区域内按照单一方向进行压栈或出栈。所以，真实的多线程应用程序的内存空间布局会是如图7.15所示的情形。

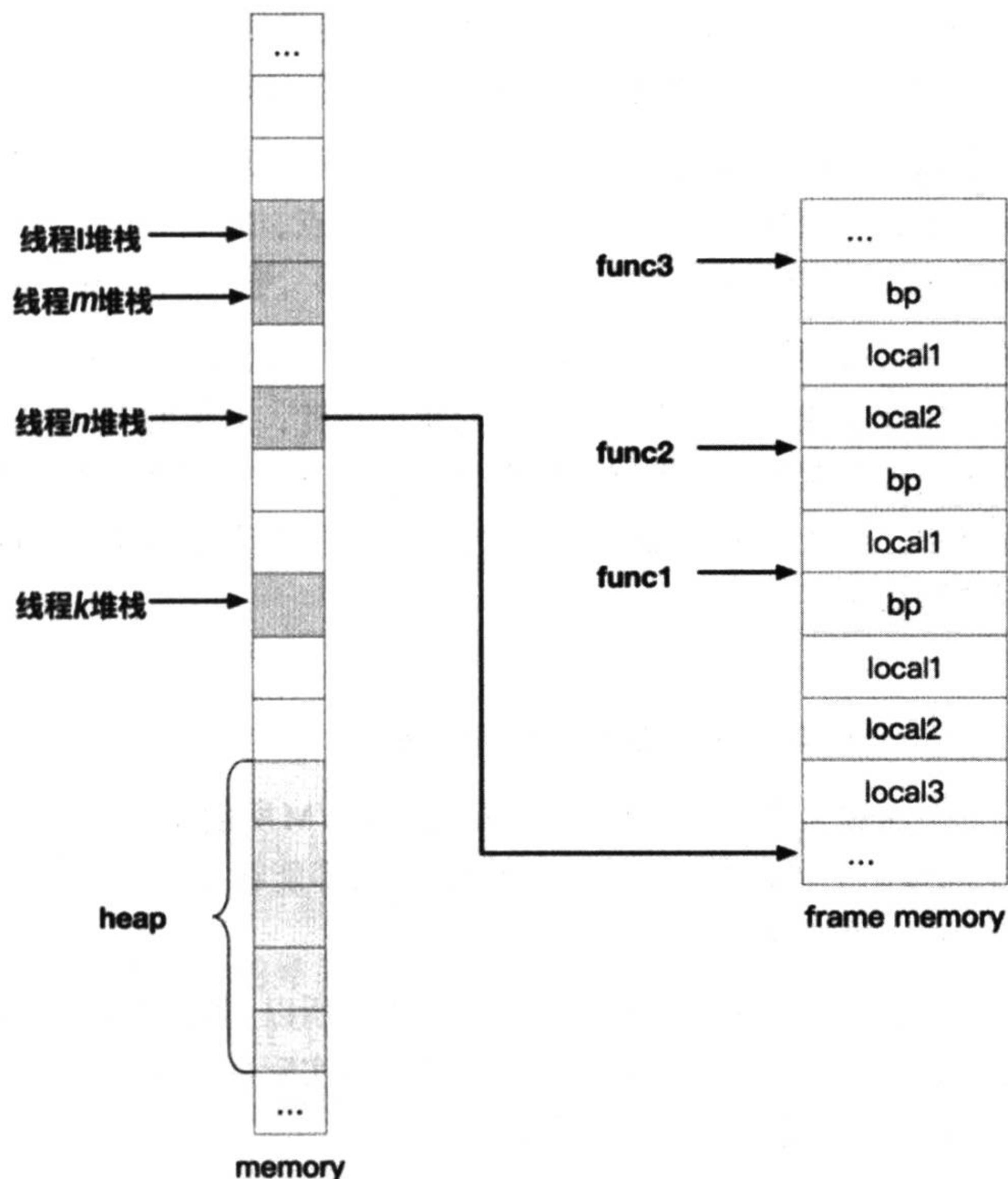


图7.15 真实的多线程应用程序的堆栈分布图

与一个应用程序能够运行多线程同理，一个操作系统上会运行若干应用程序，为了让每一个应用程序的堆栈空间能够独自完整地遵循线性顺序分布的原则（中间不允许有割裂），其实操作系统在加载一个应用程序时，就会同时划分一片内存区域给这个应用程序作为其堆栈之用，这块内存区域与其他应用程序的不会重叠。

既然操作系统为每一个应用程序划分了一块独立的、连续的内存区域，同时还为一个应用程序的每一个线程/进程也划分了一块独立的、连续的内存区域，那么问题又来了，内存空间是有限的，而应用程序和线程却可以有很多，例如一个生产环境中的 JVM 服务器就会有两三千个线程，何况这仅仅是一个 JVM 进程而已，操作系统上还要运行其他若干应用程序，而每个应用程序中又会有若干线程。所以，大量的线程与有限的内存容量之间形成了一对尖锐的矛盾，换言之，线程堆栈的空间不是任意大的，一定是有约束和限制的，否则如果随便一个线程堆栈空间都有一两百兆，那么一个 2GB 的内存空间也只能同时运行几十个线程，很显然这是肯定无法满足现代操作系统和大型应用程序的需求的。所以，几乎所有主流的操作系统都会有默认堆栈空间大小的设置。例如，在 64 位 Linux 操作系统上，默认的堆栈空间大小是 1024 KB，即 1MB。而对于 JVM 而言，开发者可以自行设置 Java 线程堆栈的空间大小。

有的小伙伴可能认为 1MB 的内存空间也太小了吧，例如 Java 程序，往往一个稍微复杂点的函数，里面就定义了若干变量，并且一个程序稍微大一点、复杂一点，其函数调用堆栈都是很深的，1MB 的堆栈空间貌似远远不够吧。

别急，1MB 的堆栈空间到底够不够，算一算就知道了。以 Java 程序为例，由于 Java 是面向对象的编程语言，因此在线程堆栈上只有指向对象的指针，对象的实例并不在堆栈上（JVM 为了优化而内部实现的栈上对象分配策略不算在内）。假设 Java 函数里全是对对象，并且假设 JVM 没有开启指针压缩选项，那么在 64 位机器上一个指针占用 64 比特的内存空间，1MB 内存空间可以容纳的指针对象数量是：

```
1 * 1024 * 1024 * 8 / 64 = 131072
```

呵呵，不多，也就只能容纳 13 万多点儿的指针宽度！

假设程序的每一个函数里都有 100 个局部变量（连同 JVM 所创建的 Java 栈帧持有的数据），那么 1MB 的堆栈空间理论上可以容纳 1 万多个函数栈帧，即能够支持 1 万多次的函数顺序调用。专业一点就是支持的最大堆栈深度达到 1 万以上。

什么样复杂的程序能够达到 1 万以上的堆栈深度呢？所以，其实 1MB 的堆栈空间对于绝大多数函数而言，已经显得十分巨大，大部分程序的线程的实际最大堆栈深度远远达不到 1 万，所以 1MB 的堆栈空间对这些线程而言显然是比较浪费的。

对于 Java 应用程序而言，可以使用下面的程序测试堆栈大小设置：

清单：ThreadStackSizeTest.java**作用：**测试 Java 程序堆栈大小

```

import java.util.concurrent.CountDownLatch;

public class ThreadStackSizeTest extends Thread{
    CountDownLatch cdl = new CountDownLatch(1);

    public void run(){
        try {
            cdl.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args){
        for(int i = 0; ; i++){
            new Thread(new ThreadStackSizeTest()).start();
            System.out.println("count=" + i);
        }
    }
}

```

以默认设置运行该 Java 程序，得到如下结果：

```

count=0
.....
count=2019
count=2020
count=2021
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:714)
    at ThreadStackSizeTest .main(Student.java:20)
    at .....

```

默认设置下，Linux 上的堆栈大小是 1MB，因此 Java 程序的默认线程堆栈大小也是 1MB。查看操作系统的剩余内存，总共只剩下 2GB 多一点，而该测试程序同时运行了 2021 个线程后，终于抛出不能再创建本地线程的异常。操作系统为每个线程都分配 1MB 的堆栈空间，2GB 内存空间正好差不多能够分配 2021 个线程。

下文继续对线程堆栈大小及栈帧大小进一步探讨，关于函数调用堆栈与栈帧的普及知识讲了很多了，先到这里为止。接下来该去“会会”Java 中所谓的栈帧了。

7.4 JVM 的栈帧

7.4.1 JVM 栈帧与大小确定

Java 虚拟机是建立在物理操作系统之上的虚拟系统，其栈帧与物理操作系统上的栈帧有所不同。物理操作系统直接基于 CPU 硬件执行指令，而 Java 虚拟机则只能基于栈式指令集运行，两者不同的运行机制决定了栈帧结构的不同。但是，Java 虚拟机本身其实并没有真正的执行能力，说到底最终还是不得不调用 CPU 硬件指令去完成程序逻辑，所以 Java 函数的栈帧与物理机器的栈帧之间又存在莫大的内在联系，这种联系体现在两方面。

1. 堆栈的设计

Java 函数的堆栈设计直接借用物理操作系统的堆栈管理思想，并无创新。Java 虚拟机也将一个 Java 应用程序的内存空间划分成堆内存与栈内存，分别治理。

另一方面，Java 函数的调用链路既然也使用“堆栈”这种容器级别的数据结构，本身就决定了 Java 应用程序的堆与栈必须要分别划分，这一点在上文进行过逻辑推导。

2. 栈帧的设计

堆栈作为栈帧的容器，其设计思路一旦确定，栈帧的设计也必然随之被确定，逃不出“压栈”与“出栈”这种“三界轮回”的铁的法则，否则 Java 函数调用链路的容器就不是“堆栈”这种数据结构了。

既然 Java 函数的内部数据使用“栈帧”这种容器进行存储，则必然导致 Java 函数的栈帧也要存储 Java 函数内的局部变量。

同时，上文也推导过，当一个操作系统决定使用“堆栈”这种容器对“栈帧”这种子容器进行线性顺序存储的策略时，由于不同函数其内部所包含的局部变量类型和数量都不同，因此栈帧大小也不同，为了支持当一个函数被执行完成后，能够让 CPU 重新定位到该函数的调用者函数的堆栈中去，一种最简单的算法就是在被调用函数的堆栈中保存调用函数的栈底地址，既然是最简单的算法，Java 虚拟机当然也得借用这种算法。所以，Java 栈帧除了需要保存 Java 方法内的局部变量外，还需要保存一些支持堆栈开辟与回收的上下文数据。Java 栈帧里保存 Java 方法内部局部变量的区域叫作“局部变量表”，而 Java 栈帧里保存上下文数据的区域叫作“JVM 帧数据”。

所以，现在知道 Java 栈帧至少由局部变量表和 JVM 帧数据所组成。

众所周知，Java的指令集是面向栈的，是栈式指令集。与栈式指令集相对的是寄存器指令集。其中寄存器指令集直接被CPU硬件所支持。对于基于寄存器指令集所设计的软件程序，其逻辑处理皆与寄存器紧密相关，一点都脱不开干系，例如下面的示例程序：

清单：test.s

作用：演示寄存器式指令集

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

    //读取第1个入参
    movl 12(%ebp), %eax

    //读取第2个入参
    movl 8(%ebp), %edx

    //对第1和第2这两个入参进行累加
    addl %edx, %eax

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

本程序使用汇编语言定义了一个add(int, int)函数，在函数里面无论是读取入参，还是对两个整数进行求和，都需要通过寄存器方能完成。

但是Java语言不是这种机制，对于下面这段Java程序：

清单：JavaStack.java

作用：演示Java的栈式指令集

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

编译后得到的字节码指令如下：

```
public static int add(int, int);
Code:
Stack=2, Locals=3, Args_size=2
0:   iload_0
1:   iload_1
```

```

2:    iadd
3:    istore_2
4:    iload_2
5:    ireturn
LineNumberTable:
line 14: 0
line 15: 4

```

诸君可以看到，Java 语言被编译后生成的指令，没有一条是与硬件寄存器相关的，反而大部分都是围绕栈的指令，例如，`iload` 将数据压栈，`istore` 将栈顶数据弹出，`iadd` 则对栈顶的两个数据进行累加，等等。

直接基于寄存器的指令之所以能够操纵寄存器，那是因为每个寄存器都能用来存储数据，只不过仅能存储一个数据。寄存器指令对寄存器的操作其实就是对相应寄存器中的数据进行的操作。同理，栈式指令集若想对栈中的数据进行操作，前提就是得有一部分内存能够被用作栈空间，如此，栈式指令集才能对栈中的数据进行逻辑运算。

同时，栈容器一定与 Java 函数相关联，或者说 JVM 必须为每一个 Java 函数划分一定的空间作为栈式指令集操作的对象，而本来每一个 Java 函数都有一个与之配套的栈帧空间，所以 JVM 干脆将函数的栈帧空间与这个操作栈合二为一，以免另外再维护一套与堆栈类似的内存空间。JVM 具体的做法是将操作栈直接嵌入到 Java 方法的栈帧之中，作为 Java 方法栈帧的一部分。

如此一来，Java 方法栈就包含了至少 3 部分数据：

- ◎ Java 方法的局部变量表。
- ◎ Java 方法堆栈调用的上下文环境数据。
- ◎ Java 方法的操作数栈。

事实上，Java 方法栈帧的结构的确与上文所推导出来的结构保持一致。一个比较详细的 Java 方法栈帧如图 7.16 所示（从下往上看）。

在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定，并且写入到了字节码文件中方法表的 `Code` 属性之中。因此，一个栈需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。在这一点上，Java 语言与 C/C++ 等语言一样，都是在编译期计算好所需栈帧大小。

在这里需要提一下“动态链接”的概念。在 C/C++ 语言中有动态链接库的概念，Java 中也有类似的概念。每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态链接。`class` 文件的常量池中有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶

段或第一次使用时转化为直接引用，这种转化称为静态解析。另一部分将在每一次的运行期间转化为直接应用，这部分称为动态链接。因此要实现动态链接的关键便是在 Java 方法栈中持有一个指针，指向常量池，如此便能得到该 Java 方法的字节码指令，并根据字节码指令映射到机器指令，从而完成方法逻辑处理。



图 7.16 Java 栈帧结构图

7.4.2 栈帧创建

HotSpot 中生成 JVM 栈帧的代码如下：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：JVM 创建栈帧

```

void TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) {
    // initialize fixed part of activation frame
    __ push(rax);                                // save return address
    __ enter();                                    // save old & set new rbp,
                                                // set sender sp
    __ push(rsi);                                // leave last_sp as null
    __ push((int32_t)NULL_WORD);
    __ movptr(rsi, Address(rbx, methodOopDesc::const_offset())); // get constMethodOop
    __ lea(rsi, Address(rsi, constMethodOopDesc::codes_offset())); // get codebase
    __ push(rbx);                                // save methodOop
}

```

```

if (ProfileInterpreter) {
    Label method_data_continue;
    __ movptr(rdx, Address(rbx, in_bytes(methodOopDesc::method_data_offset())));
    __ testptr(rdx, rdx);
    __ jcc(Assembler::zero, method_data_continue);
    __ addptr(rdx, in_bytes(methodDataOopDesc::data_offset()));
    __ bind(method_data_continue);
    __ push(rdx);                                // set the mdp (method data pointer)
} else {
    __ push(0);
}

__ movptr(rdx, Address(rbx, methodOopDesc::constants_offset()));
__ movptr(rdx, Address(rdx, constantPoolOopDesc::cache_offset_in_bytes()));
__ push(rdx);                                // set constant pool cache
__ push(rdi);                                // set locals pointer
if (native_call) {
    __ push(0);                                // no bcp
} else {
    __ push(rsi);                                // set bcp (byte code pointer)
}
__ push(0);                                // reserve word for pointer to
expression stack bottom
__ movptr(Address(rsp, 0), rsp);           // set expression stack bottom
}

```

这个过程大体上可以分为以下几步：

- (1) 恢复 return address。
- (2) 创建新的栈帧。
- (3) 将最后一个入参位置压栈。
- (4) 计算 Java 方法的第一个字节码位置。
- (5) 将 methodOop 压栈。
- (6) 将 ConstantPoolCache 压栈。
- (7) 将局部变量表压栈。
- (8) 将第一条字节码指令压栈。
- (9) 将操作数栈栈底地址压栈。

下面开始详细讲解这几个步骤。如果你将这几个步骤的技术实现都研究清楚了，那么你便会真正明白 Java 方法栈帧创建的机制。

1. 恢复 return address

JVM 创建栈帧的第一步是恢复 return address。所谓 return address 的概念，本书前面的章节多次描述过，就是 eip 寄存器，也是 JVM 调用目标 Java 方法的 call 指令的下一条指令的内存地址。

这一步在 TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数中所对应的代码是“--push(rax),”其对应的汇编是“push %eax”。

在上面的步骤——“创建局部变量表”中，使用“push \$0x0”这样的指令连续压栈，将 Java 方法内部的局部变量压入堆栈，但是在压栈之前，先执行了“pop %eax”这条指令，将 return address 从栈顶弹出至 eax 寄存器中。现在 Java 方法的局部变量全部入栈，于是又将 return address 再次还原到堆栈中。

Java 方法局部变量入栈之前和之后的堆栈结构变化如图 7.17 所示。



图 7.17 局部变量压栈之前和之后的堆栈对比图

由于 Java 方法的入参和 Java 方法内部的局部变量共同组成了局部变量表，局部变量表作为一个整体，自然不能被分隔，不能硬生生地在中间插入一个 return address，因此在对局部变量进行压栈时必然要先将 return address 拿掉，待局部变量全部压栈完成之后，再将 return address 恢复至栈顶。

2. 开辟新的栈帧

这里所谓创建新的栈帧，是指通过硬件寄存器真正开始为被调用的 Java 方法分配堆栈空间。创建栈帧的第一步就是首先要清晰地标识出自己的栈底，栈底就是一个新的栈帧的领域边界。如果没有标识出栈底，则不管分配多少新的堆栈空间，都还只是在别人的栈帧领域之内。这一点与动物世界一样，狗在路边撒泡尿就对外宣称这是自己的领域，别的动物不能再霸占，否则就要发生“世界大战”。栈帧划分自己领域的方法也很简单，执行下面这 2 条机器指令即可：

- ◎ push %ebp
- ◎ mov %esp,%ebp

从这里开始，被调用的 Java 方法终于开始有了自己栈帧的“立脚之处”，接下来新增的堆栈空间就都属于被调用的 Java 方法，而不再是“调用方法”。以后对这块区域内的数据进行访问时，都必须以新的 bp（栈底指针）为基准进行偏移寻址，而不能以调用方的 bp 为基准进行偏移寻址。

不过刚才“从这里开始，被调用的 Java 方法终于开始有了自己栈帧的立脚之处”这种说法却不够严谨，因为被调用的 Java 方法并不是从这里才开始拥有自己的栈帧空间的，在前面的步骤——创建局部变量表时，所创建的局部变量表其实正是被调用的 Java 方法自己的栈帧空间的一部分，所以说从那时候起，被调用的 Java 方法就拥有了自己的栈帧领域，而并不是执行了本步骤中的“push %ebp”和“mov %esp, %ebp”这两条指令之后才开始拥有自己的栈帧领域。这一点与 C/C++ 等语言有巨大的差异，在 C/C++ 中，大部分情况下，调用了这两条机器指令后，接下来会再配合调用“sub \$operand, %esp”为被调用的函数分配新的栈帧，因此在 C/C++ 中，这两条机器指令的出现，往往就意味着被调用的函数将会开辟自己的栈帧空间。而在 Hotspot 中，当这两条机器指令出现时，被调用的 Java 方法已经拥有自己的栈帧领域了。

不过话说回来，既然局部变量表本身就属于被调用的 Java 方法的栈帧的一部分，那么 HotSpot 应该在创建局部变量表之前就执行“push %ebp”和“mov %esp, %ebp”这两条机器指令，为被调用的 Java 方法创建栈帧。不过 HotSpot 一反常规，没有这么做，这里面的原因倒也不复杂，主要基于下面两点考虑。

1) 局部变量表的整体性

在本步骤之前有一个步骤——“创建局部变量表”，这个步骤被安排在 entry_point 例程里执行，但是严格来说，这个步骤并不能叫作“创建局部变量表”，而只能叫作将 Java 方法局部变量压栈，这是因为局部变量表不仅仅包含 Java 方法的局部变量，而且包含 Java 方法的入参，如图 7.18 所示。

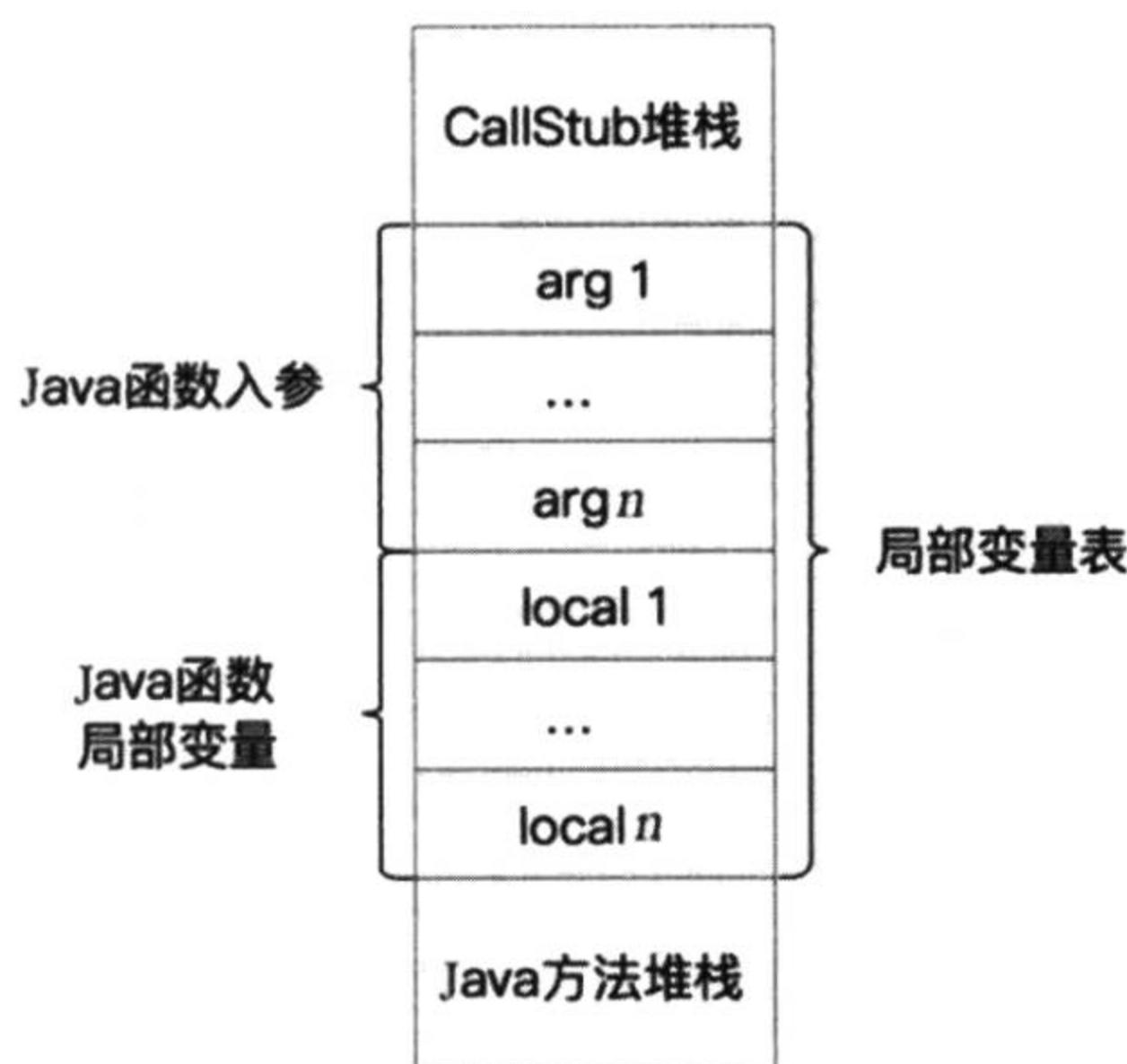


图 7.18 Java 方法的局部变量表覆盖范围

Java 函数入栈逻辑在 entry_point 例程的调用方——CallStub 例程中执行，而 Java 函数局部变量入栈逻辑则在 entry_point 例程中完成，虽然 Java 函数入参的堆栈区域与 Java 函数局部变量所在的堆栈区域分属于两个不同的栈帧（前者属于 CallStub 的栈帧，而后者按理说属于 entry_point 的栈帧），但是对于被调用的 Java 函数而言，不管是其方法入参还是方法内部的局部变量，都隶属于局部变量表，当 Java 程序执行 Java 字节码指令读写 Java 方法的局部变量表时，将 Java 函数的第一个人参所在的堆栈位置作为偏移基址对局部变量进行变址寻址。而调用函数的栈底基址 ebp 很明显并不属于被调用的 Java 方法的局部变量表的一员，自然不能被字节码读写，因此 HotSpot 并不能在对局部变量进行压栈之前执行“push %ebp”和“mov %esp, %ebp”这两条机器指令，否则局部变量表就不完整了。这与前面步骤先将 return address 出栈弹出到 rsa、等到将 Java 方法的局部变量全部入栈之后再恢复至栈顶是同一个道理。

2) Java 栈帧帧数据相对寻址

在这一步才执行“push %ebp”和“mov %esp, %ebp”这两条机器指令的另一个重要原因是，为了对 Java 方法栈帧内部的帧数据（fixed frame）进行相对寻址方便。上文讲过，Java 方法的栈帧主要由 3 部分组成：局部变量表、帧数据和操作数栈。其中只有帧数据的这部分数据的长度是固定不变的，任何 Java 方法的这部分内容所占用的堆栈内存空间大小相等，因此在 HotSpot 内部，这部分数据被称作“fixed frame”。而局部变量表与操作数栈的大小则随着 Java 方法的不

同而变化，并没有固定的大小。看图 7.19 所示的左图和右图。

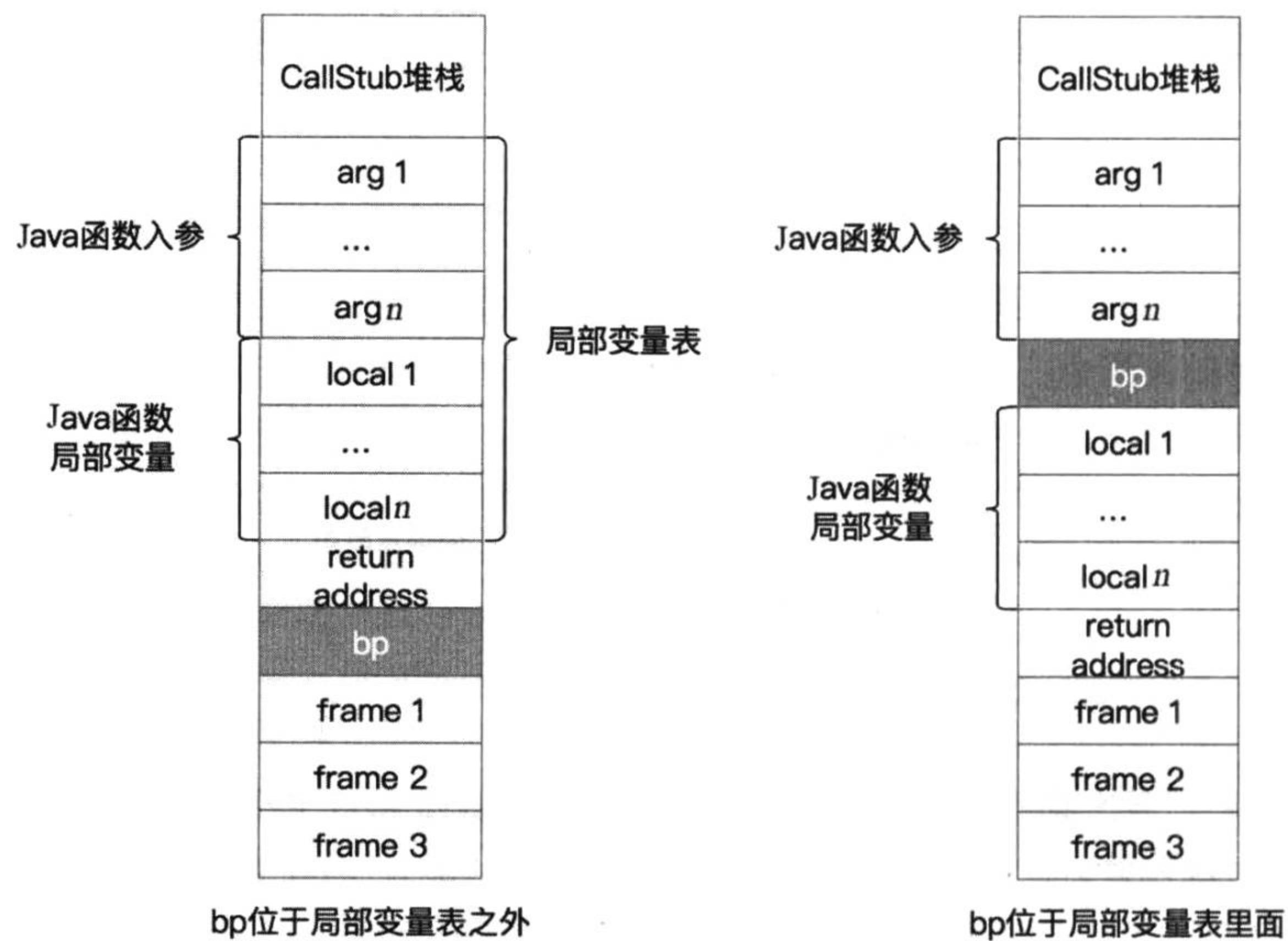


图 7.19 bp 位于局部变量表不同位置处的堆栈示意图

在图 7.19 的左图中，`bp` 是 Hotspot 实际所压栈的位置，位于局部变量表顶部；而右图中 `bp` 则位于局部变量表内部、Java 方法入参顶部、局部变量区域底部。示意图中的 `frame 1`、`frame 2`、`frame 3` 则是 Java 方法栈帧中的 `fixed frame` 部分的 3 个示例数据。在 Hotspot 执行 Java 方法的过程中，经常需要读取堆栈中的 `fixed frame` 部分的数据，而读取的方式便是基于 `bp` 进行变址寻址。以读取 `frame 1` 这个数据为例，在左图中，`frame 1` 与 `bp` 相邻，因此 Hotspot 要读取 `frame 1`，只需将 `bp` 减 4 即可（假设一个数据占用 4 字节空间）。同理，读取 `frame 2` 和 `frame 3` 只需分别将 `bp` 减去 8 和 12。但是对于右图，由于 `bp` 与 `frame 1` 之间隔了一个局部变量区域，而不同的 Java 方法，其局部变量的数量相差很大，因此要通过 `bp` 去寻址 `frame 1`，每次都得计算出 Java 方法的局部变量所占的内存总和，这将十分消耗 Java 虚拟机的性能。即使将每个 Java 方法的局部变量表所占的内存空间存储起来也于事无补，那样的话仍然要读取内存。

不过虽然 `bp` 被放在了被调用的 Java 方法的局部变量区域的后面，但是局部变量仍然应当被看作是被调用的 Java 方法的栈帧空间的一部分，毕竟 Java 方法的字节码在访问方法自己的局部变量表时，是将局部变量表当作自己堆栈空间的一部分，而不是别人的堆栈空间。

“`push %ebp`” 和 “`mov %esp, %ebp`” 这两条机器指令执行完成之后的堆栈内存空间布局如图 7.20 所示。

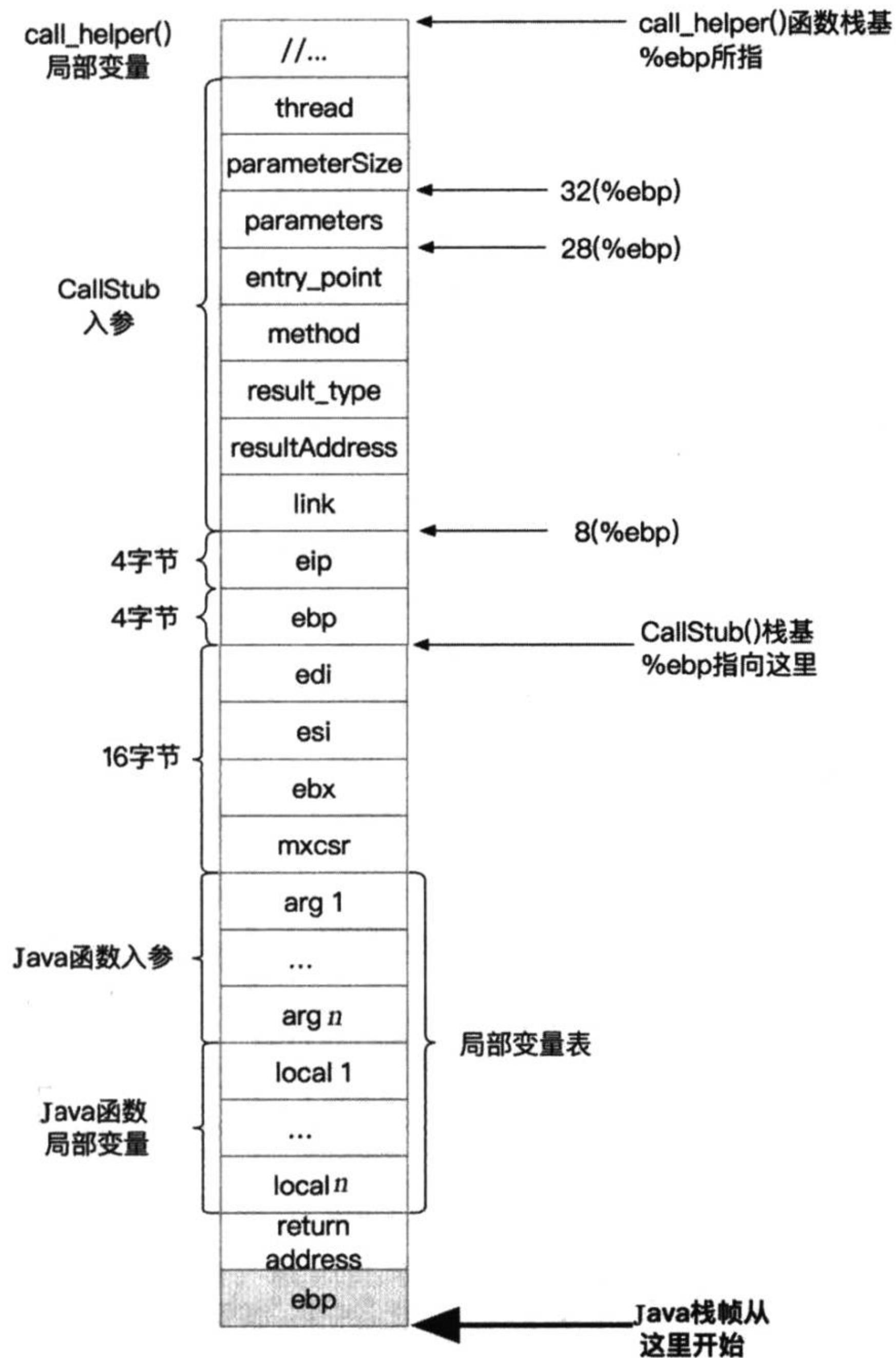


图 7.20 Java 方法栈帧刚开始的堆栈布局

3. 将最后一个人参位置压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数接着执行 __push(rsi) 指令，其对应的机器指令是“push %rsi”。在前面讲解 CallStub() 函数时，当时 rsi 寄存器里保存了 Java 方法最后一个人参在堆栈中的位置，所以执行“push %rsi”之后，Java 方法最后一个人参的位置就被压入 Java 的方法栈中。

entry_point 例程由 CallStub 例程调用，也可能会由其他例程调用，例如 Java 方法调用 Java 方法时，就会由 invoke_virtual 或者 invoke_special 等例程调用。当 entry_point 例程由 CallStub 例程调用时，CallStub 例程流程进入 entry_point 时，其栈顶元素正好就是被调用的 Java 方法的最后一个人参，所以这个位置就是 CallStub 例程所对应的函数的栈顶。同样，当 entry_point 由

其他例程进入时，被调用的 Java 方法的最后一个人参也是调用方例程的栈顶。所以，这最后一个人参自然就是调用方函数的栈顶，再往前，就进入了被调用方的堆栈空间，因此最后一个参数的位置自然成为调用方与被调用方的栈帧空间的分水岭。

在执行本指令之前，HotSpot 通过“push %ebp”和“mov %esp, %ebp”这两条机器指令将调用方的栈底指针 ebp 压入堆栈中，同时又将调用方的栈顶指针 esp 的值复制给 ebp，作为被调用的 Java 方法的栈底，因此被调用的 Java 方法的栈底实际上便是调用方 esp 栈顶指针，此时 esp 所指向的位置如图 7.21 所示。

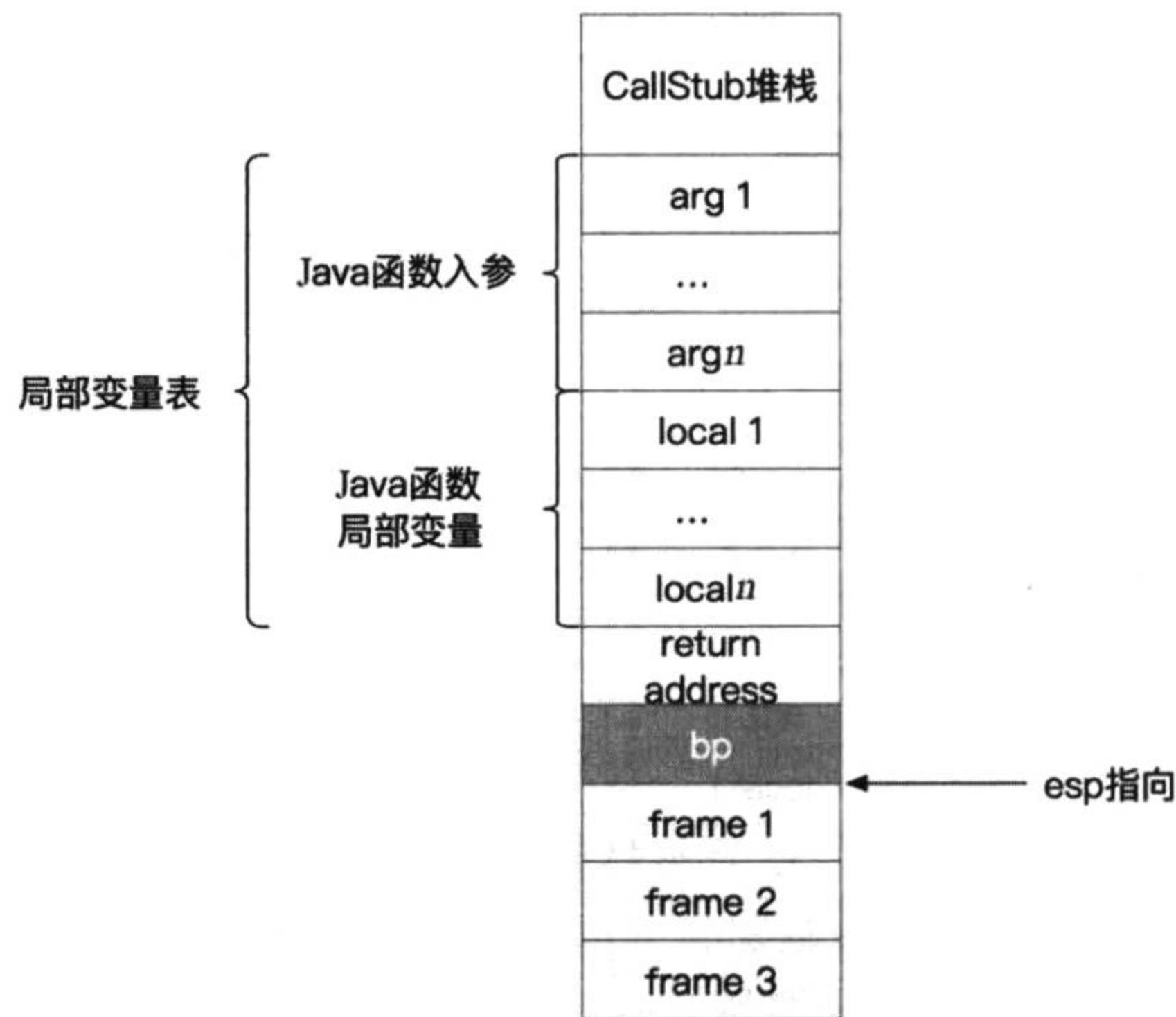


图 7.21 执行完“push %ebp”和“mov %esp, %ebp”指令后的 esp 位置

此时 esp 实际上指向了 Java 方法栈帧“fixed frame”的底部，这个位置已经超出了 Java 方法局部变量区域，这对于 Java 方法的运行倒没有任何影响，但是等 Java 运行完成，即将要退出时，就会引起很大的问题。

当 Java 程序运行完成时，HotSpot 必须回收其对应的堆栈空间。对于操作系统而言，堆栈空间的回收别无他法，全都依赖 esp 寄存器值的修改。对于 C/C++ 等本地编译型的语言，ebp 指针指向被调用函数栈帧的栈底，因此回收被调用函数的栈帧空间时，只需将 esp 恢复至被调用函数的 ebp 指针即可。Java 方法栈帧由局部变量表、帧数据和操作数栈这三部分所组成，其中局部变量表又一分为二，一部分为 Java 方法入参区域，一部分则为 Java 方法局部变量区域，前者属于调用方的栈帧空间，后者才属于被调用一方的栈帧空间，因此当被调用的 Java 方法执行完毕后，HotSpot 需要回收的堆栈空间实际上仅包含帧数据、操作数栈及局部变量表中的局部变量所在的区域，如图 7.22 所示。

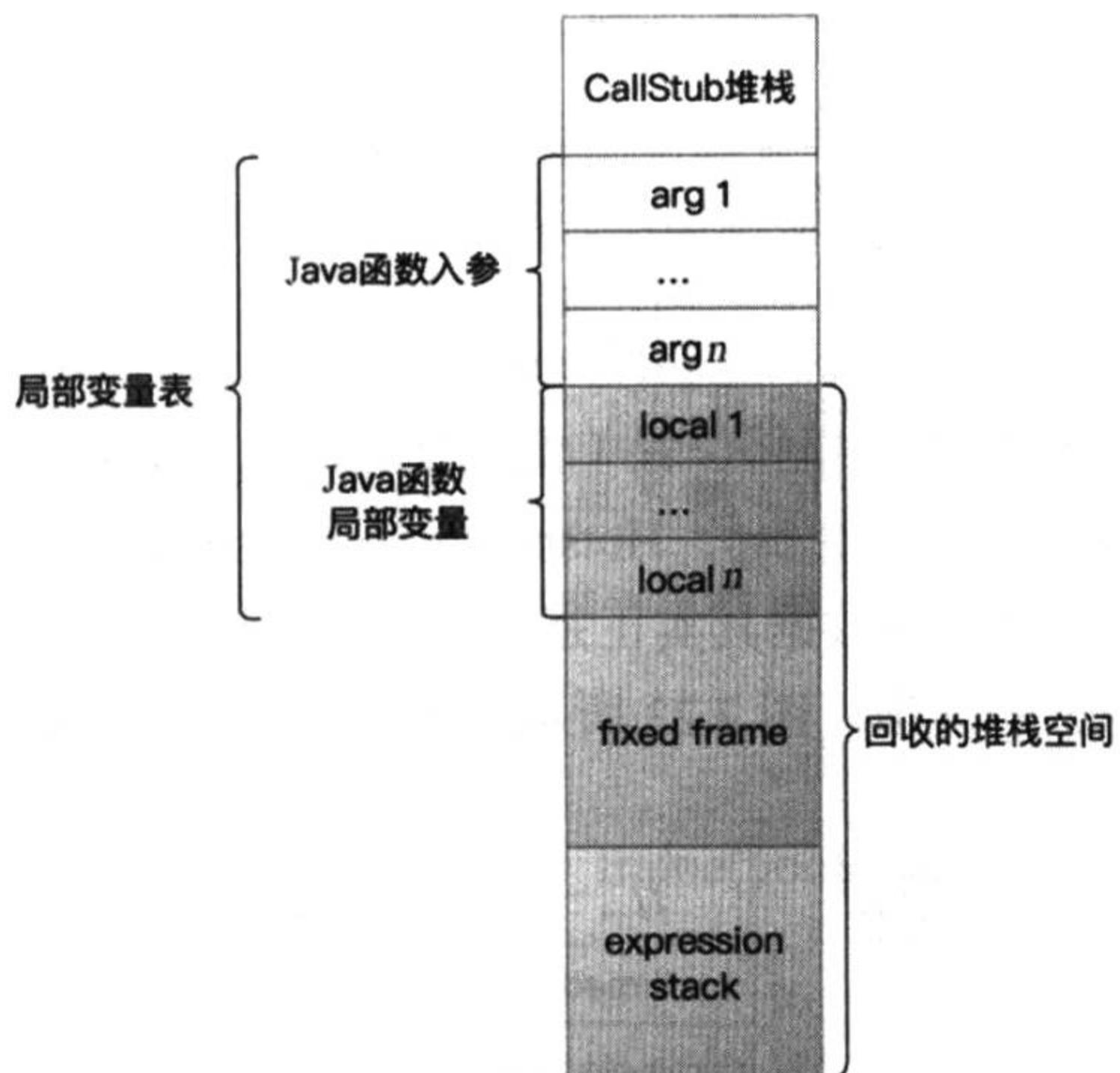


图 7.22 Java 方法执行完成之后，回收的堆栈空间涵盖范围

在上一步，执行完“push %ebp”和“mov %esp, %ebp”这两条机器指令后，调用方函数的栈顶指针 esp 被复制给了被调用的 Java 方法的栈底指针 ebp，但是 esp 指向的位置并不是 Java 局部变量表中入参区域与局部变量区域的分界线之外，因此当 Java 方法执行完成之后，如果 Hotspot 以 Java 方法的栈底指针 ebp 来确定所要回收的堆栈空间，就会出现异常，因为 Java 方法局部变量表中的局部变量区域所占的堆栈内存空间不在回收之列。

因此，为了能够正确回收被调用的 Java 方法的栈帧空间，HotSpot 必须记录 Java 方法的调用方的栈顶位置，回收堆栈空间时，就以此来确定回收范围。这便是本步骤——将 Java 方法入参的最后一个参数在堆栈中的位置进行压栈的意义所在。

事实上，如果将 Java 字节码中的 return 指令展开成机器指令，也的确会发现其中的奥妙。假设一个 Java 方法的返回值类型是 int，则该 Java 方法的最后一条字节码指令必定是 ireturn，ireturn 指令展开后的机器码如下所示（使用汇编助记符展示）：

```

ireturn 172 ireturn [0xb36dbca0, 0xb36dbe90] 544 bytes

[Disassembling for mach='i386']
0xb36dbca0: pop    %eax
0xb36dbca1: mov    %esp,%ecx
0xb36dbca3: shr    $0xc,%ecx
0xb36dbca6: mov    -0x48f07d20(%ecx,4),%ecx
//.....
0xb36dbe95: add    $0x8,%esp
0xb36dbe98: pop    %eax
0xb36dbe99: mov    -0x4(%ebp),%ebx

```

```

0xb36dbe9c: mov    %ebp,%esp
0xb36dbe9e: pop    %ebp
0xb36dbe9f: pop    %esi
0xb36dbea0: mov    %ebx,%esp
0xb36dbea2: jmp    *%esi

```

观察其中加粗的两行指令：“`mov -0x4(%ebp), %ebx`” 和 “`mov %ebx, %esp`”，第一条指令将 `ebp` 所指向的堆栈内存位置的相邻位置（低地址方向）的数据取出来，传送给 `ebp`；接着第二条指令便将 `ebp` 的值再度传送给 `esp`，Hotspot 正是通过这第 2 条指令彻底回收了当前 Java 方法的栈帧。`-0x4(%ebp)` 位置上的数据其实正是本步骤中所压栈的 Java 方法的最后一个人参在堆栈中的位置，也即调用方的栈顶。图 7.23 所示是本步骤执行完成之后的堆栈内存布局。

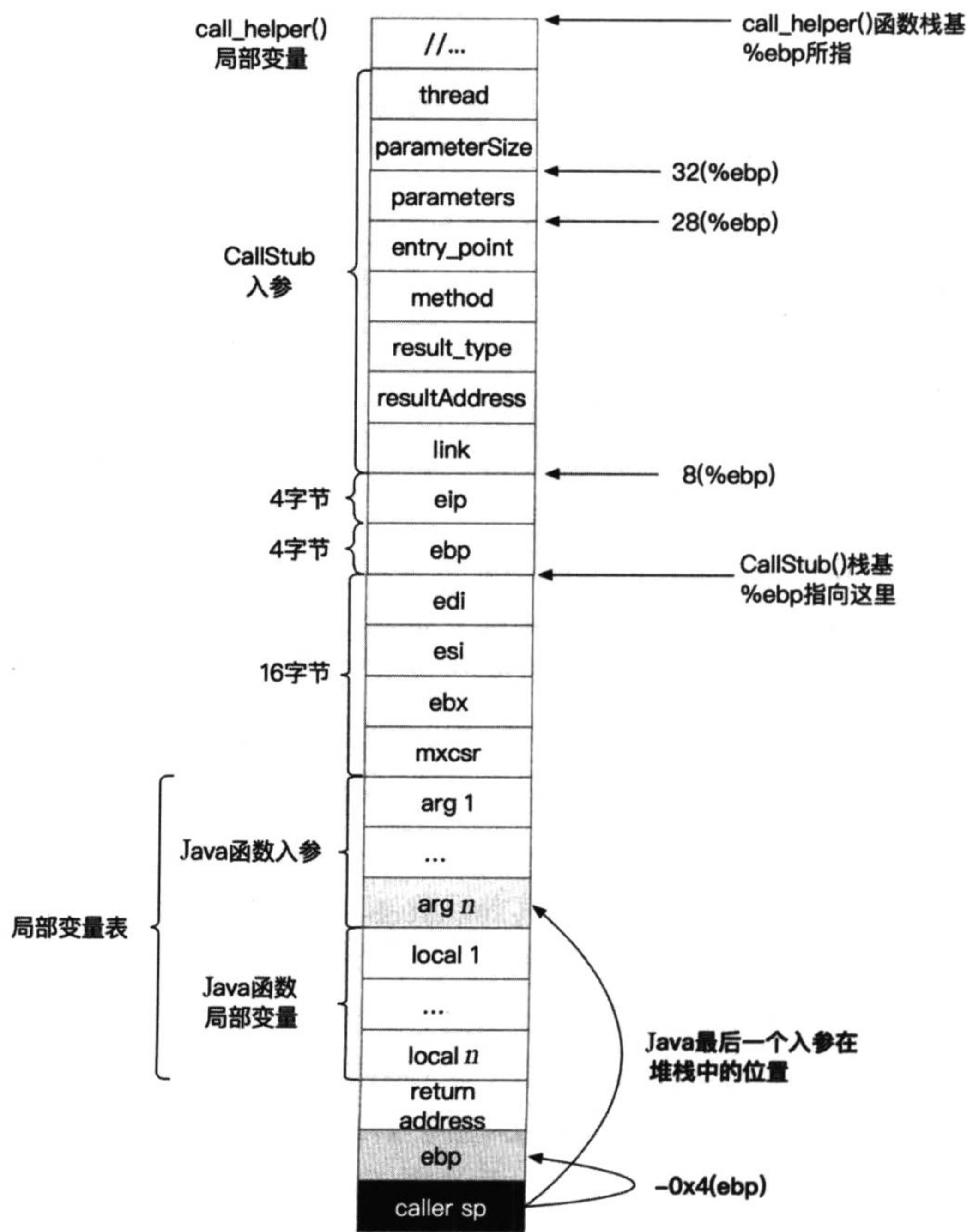


图 7.23 执行完 “push %rsi” 指令之后的堆栈空间布局图

执行完本步骤的“push %rsi”这条机器指令后，caller sp 就被压入栈中，这个位置中所存储的值其实就是 arg n 所在的堆栈内存的地址（如图 7.23 中箭头所指）。同时，紧邻 caller sp 上方的位置中所存储的值正是 ebp，在 32 位操作系统上，caller sp 相对于 ebp 的位置就是-0x4(%ebp)，因此最终 Hotspot 在执行 Java 方法的最后一条字节码指令 ireturn 时，通过-0x4(%ebp)就能获取 caller sp，而 caller sp 又指向 Java 入参的最后一个位置，此位置也恰好是 Java 方法的调用方的栈顶，注意，栈顶哟！Hotspot 正是依赖这个位置来确定需要回收的堆栈内存的界限的。

所以在 HotSpot 内部，本步骤也被叫作“set sender sp”，即调用方栈顶保存。

4. 计算 Java 方法的第一个字节码位置

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数接着执行下面两行代码：

- ◎ __ movptr(rsi, Address(rbx, methodOopDesc::const_offset()));
// get constMethodOop
- ◎ __ lea(rsi, Address(rsi, constMethodOopDesc::codes_offset()));
// get codebase

这两行代码所对应的机器码是：

- ◎ mov 0x8(%ebx),%esi。ebx 指向 method，0x8(%ebx)指向 ConstMethod。
- ◎ lea 0x30(%esi),%esi。在这一步让%esi 指向 codebase，也就是 method 的第一个字节码的位置。

这一段不难理解，在进入 entry_point 例程之前，ebx 寄存器中所保存的就是 Java 方法在 JVM 内部对应的 methodOop 对象首地址。0x8(%ebx)恰好指向 constMethodOop (method 与 constMethod 这两个对象会在下一章详细讲解)，constMethodOop 对象在常量池解析阶段生成，该对象主要保存 Java 方法中的只读信息，例如异常信息表、Java 方法注解信息、方法名、Java 方法的字节码指令等。

methodOop 的内存结构如下 (JDK 6, 32 位 x86 Linux 平台)：

```
methodOop
+0: header
+4: klass
+8: constMethodOop
+12: constants
+16: methodData
//...
```

由此可见，相对 methodOop 首地址偏移 8 字节的位置处所保存的正是指向 constMethodOop

对象的指针，因此 HotSpot 执行“`mov 0x8(%ebx), %esi`”这条指令之后，esi 寄存器中所存储的就是 constMethodOop 对象的内存地址了。

这里为了描述方便（免得众位道友辛苦翻看源码），简单贴出 constMethodOop 的布局结构与各字段偏移量（JDK 6，32 位 x86 Linux 平台）：

```
constMethodOop
+0:    header
+4:    klass
+8:    fingerprint
+16:   is_conc_safe
+20:   method
+24:   stackmap_data
+28:   exception_table
+32:   constMethod_size
+36:   interpreter_kind
+37:   flags
+38:   code_size
+40:   name_index
+42:   signature_index
+44:   method_idnum
+46:   generic_signature_index
+48:   byte codes
```

注意，在这个结构中，`is_conc_safe` 是 `bool` 类型，仅占用 1 字节。但是其后面紧跟的字段是 `methodOop` 指针类型，其宽度是 4，因此需要按 4 字节对齐，也因此 `is_conc_safe` 后面有 3 字节的补位。

前文在分析 Hotspot 解析 Java 类的方法时，会将 Java 类方法的字节码指令保存到对应的 `constMethodOop` 对象实例的末尾。`constMethodOop` 类型的最后一个字段是 `generic_signature_index`，类型是 `u2`，其相对于 `constMethodOop` 对象首地址的偏移量是 46，而 Java 方法的字节码指令就分配在该字段的后面，因此 Java 方法的字节码的第一条指令的位置相对于 `constMethodOop` 就是 48。因此，在本步骤会通过“`lea 0x30(%esi),%esi`”指令将 Java 方法的第一条字节码指令的内存位置传送给 `esi` 寄存器。在 `entry_point` 例程的最后，会通过 `call` 指令去开始真正执行 Java 方法的字节码指令。

5. 将 methodOop 压栈

`TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)` 函数接着执行下面这行代码：

```
◎ __ push(rbx) // save methodOop
```

其对应的机器指令是：

- ◎ push %rbx

rbx寄存器指向Java方法在JVM内部所对应的methodOop对象首地址，因此这一步的目的便是将methodOop对象的首地址压入栈中。在Hotspot调用Java方法的过程中，将通过这个地址读取到Java方法的全部信息，例如进行多态运行期动态方法绑定时需要定位到callee从而决定到底调用继承体系中的哪一个对象的方法。

6. 将ConstantPoolCache压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)函数接着执行下面这几行代码：

- ◎ __ movptr(rdx, Address(rbx, methodOopDesc::constants_offset()));
- ◎ __ movptr(rdx, Address(rdx, constantPoolOopDesc :: cache_offset_in_bytes()));
- ◎ __ push(rdx); // set constant pool cache

其对应的机器指令是：

- ◎ mov0xc(%ebx),%edx --ConstMethod *
- ◎ mov0xc(%edx),%edx --ConstantPoolCache *
- ◎ push%edx

在32位x86Linux平台上，相对于methodOop首地址偏移量为0xc（即十进制12）的字段是constantPoolOop指针，该指针指向Java类所对应的内存常量池首地址，Hotspot通过“mov 0xc(%ebx),%edx”指令将constantPoolOop首地址传送给edx寄存器。

而constantPoolOop的内存结构布局如下所示（前文详细描述过，此处简略展示一二）：

```
constantPoolOop
+0:    header
+4:    klass
+8:    tags
+12:   constantPoolCacheOop
+16:   _pool_holder
//...
```

相对于constantPoolOop首地址偏移量为0xc的字段是constantPoolCacheOop，HotSpot通过“mov 0xc(%edx),%edx”指令将constantPoolCacheOop首地址传送给edx寄存器，接着执

行“push %edx”指令将 constantPoolCacheOop 首地址压入 Java 方法栈中。由此可知，对于同一个 Java 类文件中的所有 Java 方法，每一个 Java 方法的栈帧中都必须持有指向该 Java 类解析后所生成的常量池缓存对象的地址。常量池缓存中的内容皆是直接引用，不必像常量池那样，存放的都是索引号。

到了这一步，堆栈内存布局结构如图 7.24 所示。

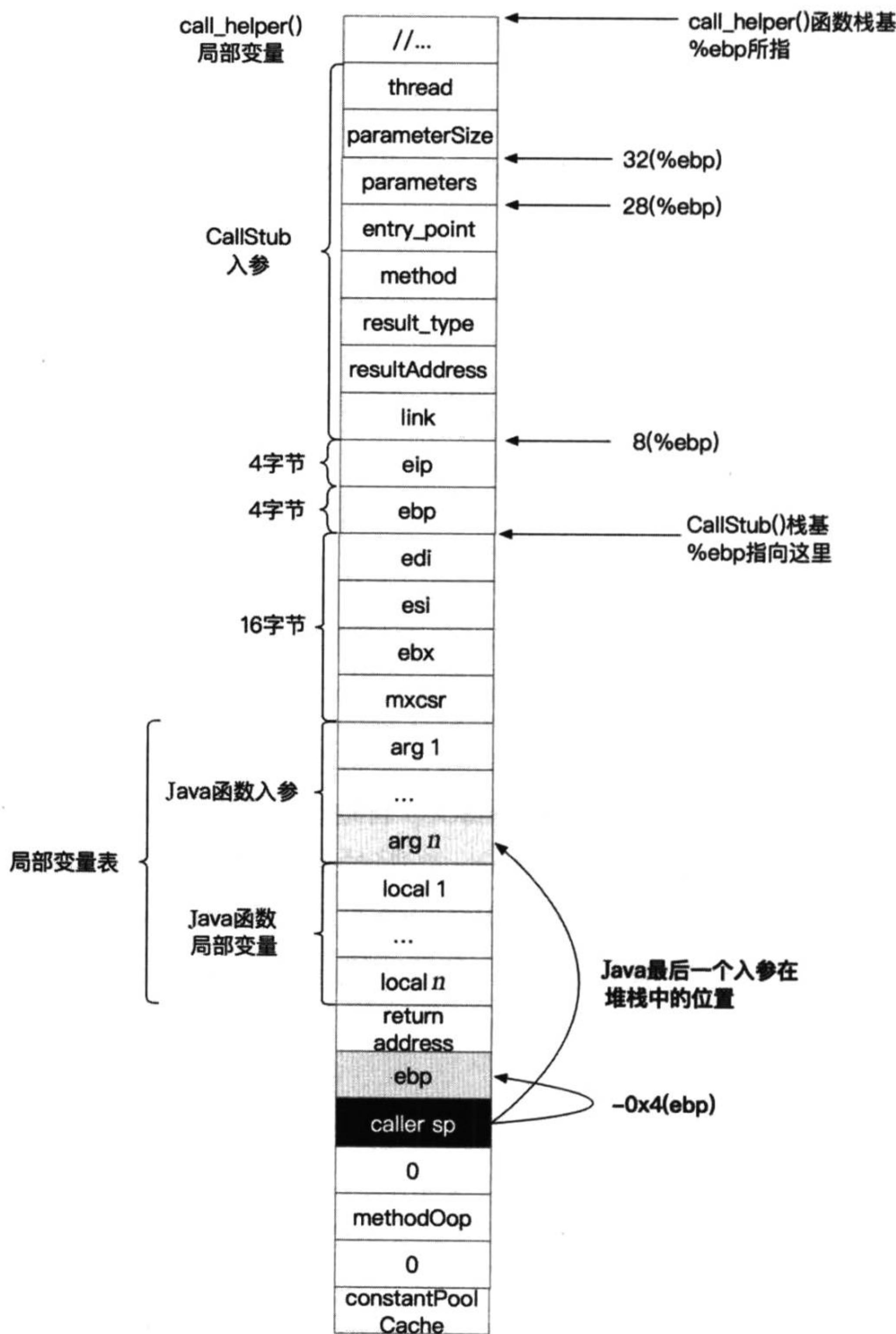


图 7.24 constantPoolCacheOop 入栈后的堆栈内存布局结构

注意，在靠近栈顶的位置处多出了 2 个 0，这两个位置为保留字段或者占位符，在 Java 方法运行过程中将会用于存储运行期的部分结果。

这里需要说明的是，这种堆栈内存布局图绘制到现在已经变得很大、很复杂了，笔者本想砍去一部分，但是最终仍然保留了全部信息。因为本书浓墨重彩地描述了 Java 方法调用的入口及栈帧创建的整个过程，从 JVM 内部的 CallStub 例程开始，一直到现在，中间横跨各种技术点，但是不管中间补充了哪些侧面的信息，主线一直未变，这张图就是主线。相信各位道友如果是从开始一路读下来，只要看到这张图，自然会感到主线清晰，并不会因为中间其他技术点的插入而扰乱了思路。所以虽然这张图变得十分复杂，但笔者丝毫不敢精简一分半点，不然可能连笔者自己写着写着都可能会写偏了。

7. 将局部变量表压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数接着执行 __push(rdi) 指令，其对应的机器指令是 push %edi。

在前面初始化 Java 方法局部变量区域时，通过不断地循环执行 “push \$0x0” 指令为 Java 方法内的局部变量开辟堆栈空间，而在循环之前，将 Java 方法的第一个入参在堆栈中的位置传给了 edi 寄存器，并且中间并没有被修改，因此此时 edi 寄存器仍然指向 Java 方法的第一个入参在堆栈中的位置。

这一步对于 Java 方法而言具有十分重要的意义，因为 Java 方法栈帧部分的局部变量表的起始位置，其实就是 Java 方法的第一个入参在堆栈中的位置，因此 Java 栈帧必须要记录下这个位置，作为 Java 方法的局部变量表的第一个槽位，不然在 Java 方法执行的过程中，Java 字节码无法读写局部变量表。

举一个十分简单的例子，有如下 Java 类：

清单：A.java

作用：Java 方法局部变量表

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

使用 javap -verbose 命令查看 doSomeThing() 方法所对应的字节码如下：

```
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

```

当 Hotspot 执行源代码中的 `int a = 3` 时，实际上执行的是 `iconst_3` 和 `istore_0` 这两条字节码指令，其中 `iconst_3` 字节码所对应的汇编指令如下：

```

iconst_3 6 iconst_3 [0xb36d7ce0, 0xb36d7d20] 64 bytes
[Disassembling for mach='i386']
0xb36d7d04: mov    $0x3,%eax //将操作数 3 推送至栈顶缓存（缓存在 eax 寄存器中）
0xb36d7d09: movzbl 0x1(%esi),%ebx
0xb36d7d0d: inc    %esi
0xb36d7d0e: jmp    *-0x48f106a0(,%ebx,4)

```

当 Hotspot 执行完 `iconst_3` 字节码指令后，操作数 3 就被推送到 `eax` 寄存器中（这里涉及栈顶缓存，下文会详细讲解栈顶缓存的概念及实现机制）。接着在 `istore_0` 字节码指令中理应读取栈顶缓存的值，并将其保存到 `doSomeThing()` 方法的局部变量表的第一个槽位上（因为变量 `a` 是 `doSomeThing()` 方法内的第一个局部变量）。来看 `istore_0` 所对应的汇编指令：

```

istore_0 59 istore_0 [0xb36d9240, 0xb36d9260] 32 bytes
[Disassembling for mach='i386']
0xb36d9241: mov    %eax,(%edi)           //这里引用了 edi 寄存器
0xb36d9243: movzbl 0x1(%esi),%ebx
0xb36d9247: inc    %esi
0xb36d9248: jmp    *-0x48f0f2a0(,%ebx,4)

```

在 `istore_0` 字节码所对应的汇编指令中，执行 “`mov %eax,(%edi)`” 指令，将栈顶缓存的值（即 `eax` 寄存器中所保存的值）传送给 `edi` 寄存器所指向的内存位置。而 `edi` 寄存器正指向 Java 局部变量表的第一个槽位，而该槽位其实也是 Java 方法的第一个人参所在的堆栈内存位置。只不过本示例中的 `doSomeThing()` 方法没有入参（静态方法，连隐藏的 `this` 指针也没有），因此 `edi` 寄存器的位置只能是 Java 方法内的第一个局部变量的堆栈位置。

在本例中，`doSomeThing()` 函数的第二句源代码是 `int b = 81`，变量 `b` 是 `doSomeThing()` 方法内的第 2 个局部变量，那么在 32 位 x86 平台上，其位置应该是在 `doSomeThing()` 方法的局部变

量表的第 2 个槽位，而 edi 寄存器指向第 1 个槽位，因此第 2 个槽位相对于第一个槽位的偏移量是 $-0x4$ ，使用 edi 表示应该是 $-0x4(%edi)$ 。看看 int b = 81 所对应的字节码指令是不是这么处理的就知道了。int b = 81 对应两条字节码指令：

- ◎ bipush 81
- ◎ istore_1

其中，bipush 81 也会将 81 这个操作数推送至栈顶缓存，保存在 eax 寄存器中。istore_1 字节码指令的功能便是将栈顶缓存的内容保存到 Java 方法局部变量表的第 2 个槽位上，一起围观其对应的机器指令：

```
istore_1 60 istore_1 [0xb36d9280, 0xb36d92a0] 32 bytes
[Disassembling for mach='i386']
0xb36d9281: mov    %eax,-0x4(%edi)
0xb36d9284: movzbl 0x1(%esi),%ebx
0xb36d9288: inc    %esi
0xb36d9289: jmp    *-0x48f0f2a0(,%ebx,4)
```

这条字节码指令果然引用了 $-0x4(%edi)$ 这个位置，通过 edi 寄存器直接定位到 Java 方法的第 2 个槽位。

总之，在 Java 方法运行期间，对 Java 方法入参和内部局部变量的读写就全靠 edi 寄存器了。这么一个重要的数据当然要保存到 Java 方法的栈帧里去了。

edi 在 Hotspot 内部也被叫作 locals pointer，即指向局部变量表的指针。

8. 将第一条字节码指令压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数接着执行的源代码是：

```
if (native_call) {
    __ push(0); // no bcp
} else {
    __ push(rsi); // set bcp(byte code pointer)
}
```

由于 JVM 既可以加载 Java 类，也可以加载 C/C++/Delphi 等程序库，因此 Hotspot 需要通过 native_call 判断即将被调用的方法是否是 Java 方法。如果是 Java 方法，就需要将 Java 方法所对应的第一条字节码指令的位置入栈，这个位置在前面的步骤中已经计算好，并被保存在 rsi 寄存器中，因此只需执行“push %rsi”指令即可。反之，如果被调用的方法不是 Java 方法，那么就是本地方法。C/C++/Delphi 等程序库都会被 JVM 当作本地方法调用。所谓本地方法，也即直接本地编译型方法，例如 C 语言程序，编译后直接生成能够被 CPU 识别的二进制机器指令，

所以本地方法不存在所谓“Java 字节码指令”一说，因此也就无须将 rsi 寄存器压入栈中。

rsi 所指向的位置在 Hotspot 内部有一个专门的称呼——bcp，其含义是 byte code point，即指向 Java 方法字节码指令的指针。

9. 将操作数栈栈底地址压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数所执行的最后两句源代码如下：

- ◎ __ push(0);
- ◎ __ movptr(Address(rsp, 0), rsp)

其对应的机器指令如下：

- ◎ push \$0x0
- ◎ mov %esp,(%esp)

第一条机器指令往栈顶压入一个零值，接着通过第二条机器指令将当前 esp 寄存器的值覆盖为刚刚压入的零值。

在前面步骤中执行各种 push 操作，每一次 push 完之后，物理 CPU 会自动将 esp 寄存器的值更新为当前最新的栈顶位置，因此到了本步执行的时候，esp 寄存器本来就指向了当前堆栈的栈顶，而这里又通过“mov %esp,(%esp)”指令将 esp 的值覆盖为刚刚压入栈顶的零值，因此此时栈顶所保存的值就是其自己的内存位置。此时堆栈内存布局结构如图 7.25 所示。

在这一步，之所以要将 esp 的值存储起来，是因为截止到本步骤，Java 方法栈帧的“fixed frame”部分就创建完成了，Java 方法栈帧接下来的部分就是操作数栈。虽然操作数栈也属于 Java 方法栈帧的一部分，但是在 Java 方法的运行过程中，Java 字节码指令所面向的栈，实际上便是这个“操作数栈”，而不是整个 Java 方法栈帧。Java 字节码进行压栈和出栈，会基于操作数栈的栈底位置进行变址寻址。由于 Java 方法的 fixed frame 接下来相邻的部分就是操作数栈，因此 fixed frame 的栈顶位置其实就是操作数栈的栈底，HotSpot 将该位置记录下来，用作操作数栈的栈底，在 Hotspot 内部，将该位置叫作“expression stack bottom”，即表达式栈栈底。

在这一步之所以要将 esp 寄存器的值存储起来的另一个原因是，Java 字节码的压栈与出栈指令，最终都会映射成物理机器码的 push 和 pop 指令，而这两条指令都会自动修改 esp 的值。而当 Hotspot 执行到这一步时，esp 恰好指向了 fixed frame 的顶部，因此 Hotspot 将其保存起来，一方面是为了能够定位到 fixed frame 的顶部位置，另一方面却也是方便了 Java 字节码指令的压栈与出栈的执行。

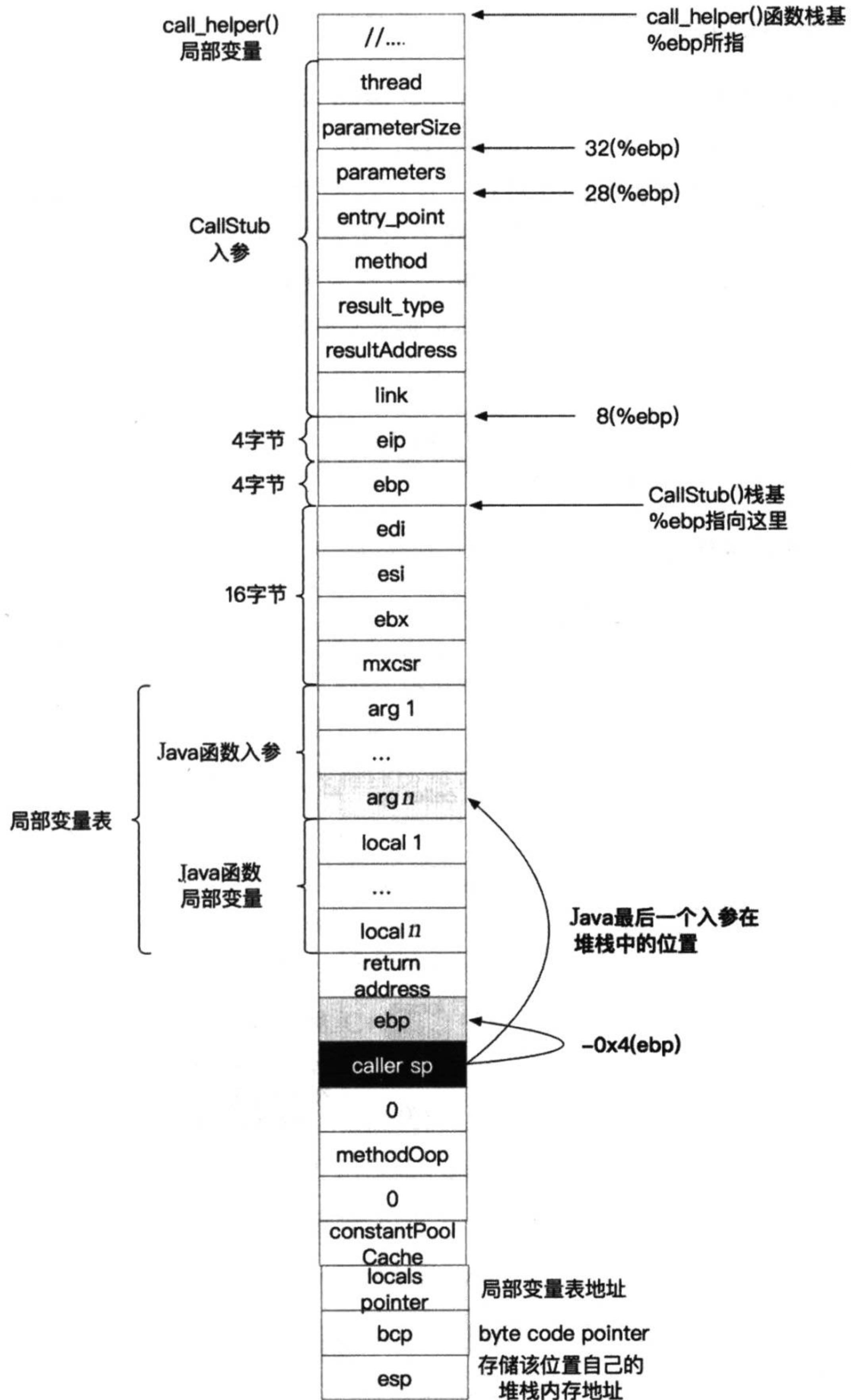


图 7.25 `TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)`
函数执行完之后的堆栈内存结构布局

10. Java 栈帧详细结构

分析完 `TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)` 这个函数，现在回头再看看 Java 方法的栈帧组成，自然是如明镜一般。前文所述，Java 方法栈帧由 3 大部分组成，分别是局部变量表、帧数据和操作数栈，现在已经知道了局部变量表与帧数据的分配机制及内存组成，仅剩下操作数栈。不过操作数栈相比于局部变量表，却是更加变化不定，并且与被调用的 Java 方法息息相关，紧密相邻。这里先总结下 Java 方法栈帧中固定的部分，如图 7.28 所示。

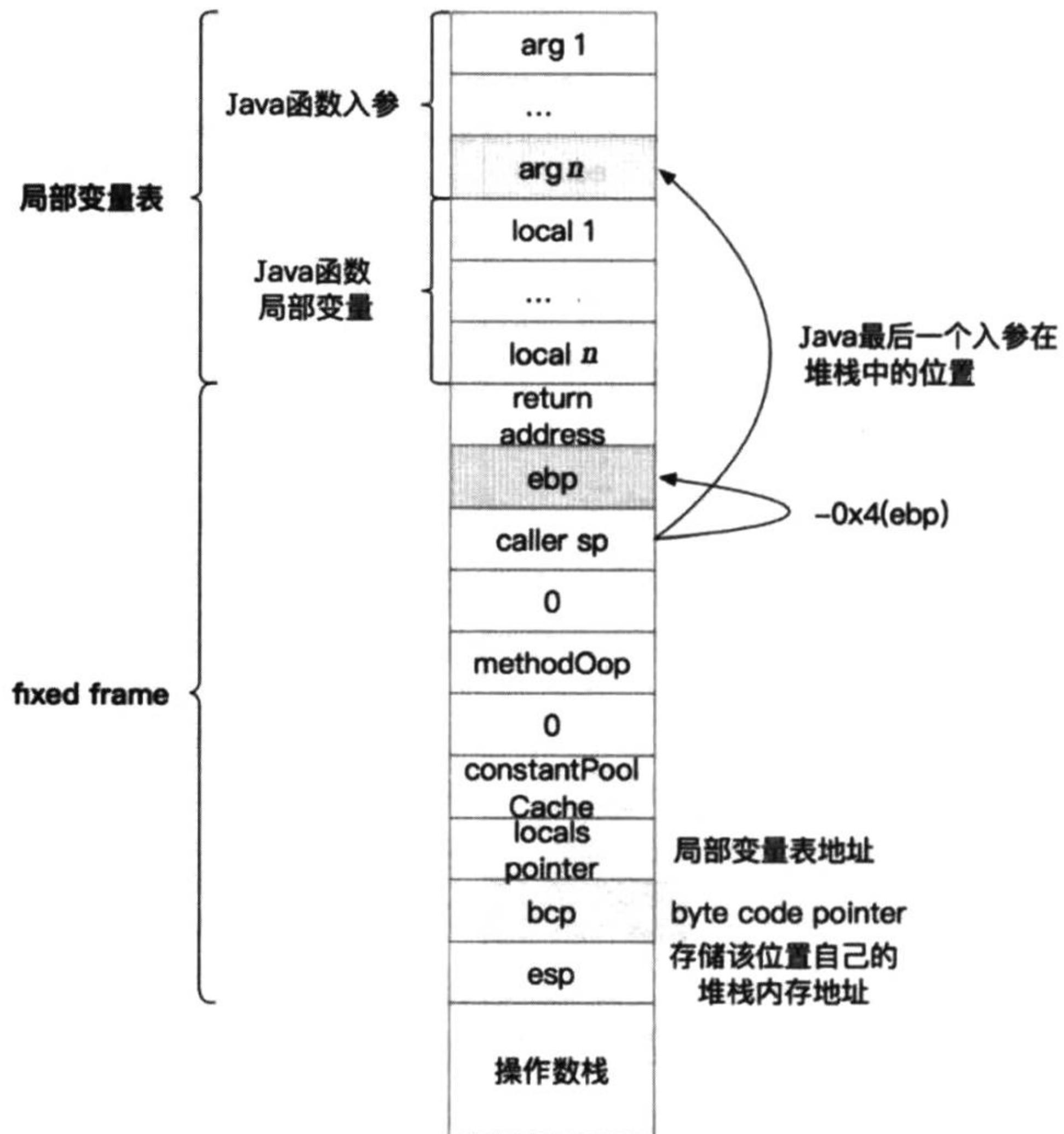


图 7.26 Java 栈帧的详细结构

Java 栈帧与 C/C++ 的栈帧自然是大不相同，Java 栈错综复杂，并且结构也是怪异到了极点，不像 C/C++ 等编程语言的栈帧那般简单明了。这些怪异点主要体现在下面几点。

1) 栈底与局部变量表底部不是同一位置

Java 方法的局部变量表包含两部分，分别是 Java 方法入参区域和 Java 内的局部变量区域。但是 Java 方法入参区域并不属于 Java 方法的栈帧，而是属于其调用方的栈帧的一部分。因此，

Java 方法栈由局部变量表、操作数栈和帧数据 (fixed frame) 组成，这一说法并不够严谨，因为 Java 方法仅包含了局部变量表的一部分区域。

由于 Java 方法的入参区域与内部局部变量区域分属于两个不同的方法，因此当被调用的函数执行完成，需要回收其堆栈空间时，必须标记其真实的栈底位置，因此 Hotspot 只能在 fixed frame 中保存 Java 方法的最后一个人参的堆栈内存位置，这个位置恰好就是调用方的栈顶，也是被调用的 Java 方法的栈底位置。HotSpot 通过这个位置来标记所需要回收的堆栈内存范围。

虽然 Java 方法的入参区域与内部局部变量区域分属于两个不同的方法，但是对于被调用的一方而言，这两个区域又必须在物理上连成一片，这样才能形成一个整体，作为一个完整的局部变量表，方便被调用的 Java 方法的字节码指令进行读写，因此 Hotspot 只能将调用方的栈底指针 bp 保存到 fixed frame 之中。这虽然未尝不可，但是毕竟与 C/C++之类的编程语言有很大差异，在程序逻辑上相差甚远。

2) Java 栈帧需要额外保存若干数据

Java 的 fixed frame 部分的数据其实与 Java 的源程序指令没有丝毫关系，但是 fixed frame 中却保存了大量的运行时数据，这一点也与其他编程语言差别较大。例如 C/C++，编译后的栈帧几乎全部用于保存函数的入参和局部变量，除了寥寥几个寄存器上下文的现场保存，几乎没有其他额外的数据。Java 栈帧的这种特性所带来的结果就是运行同样的逻辑，其堆栈空间需要占用更多的内存。

这也难怪，毕竟 JVM 仅仅只是使用软件模拟的虚拟系统而已，由于硬件资源有限，因此只能使用软件的方式来保存上下文了。例如 Java 方法的 fixed frame 中需要保存 bcp，而在 C/C++ 等编程语言中不需要保存这么一个指向指令的指针，自有那硬件寄存器 (ip 段寄存器) 自动存储。再如 Java 栈帧中的 caller sp，在 C/C++ 语言中更加不需要使用宝贵的堆栈内存去保存，也有那硬件寄存器去存储 (bp)。

当然，虽然 Java 方法栈帧的结构十分怪异，但是却自有其道理，这由 JVM 自身的执行引擎和内存模型所决定。佛曰：种善因，得善果；种恶因，得恶果。同样，JVM 为其天生所秉持的崇高理想而设计的各种巧妙机制，最终决定 Java 的方法栈只能长成这个样子，多一分不行，少一分更不行。

11. 创建 fixed frame 的全部脚本

Hotspot 调用 TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数创建 fixed frame，这个函数与其他例程一样，在 JVM 启动之初就会全部转换为机器指令，Hotspot 只需直接调用机器指令即可完成 Java 帧数据的创建。这部分所对应的机器指令 (32 位 x86 平台) 如下：

```

0xb36d6561: push    %eax          --恢复 return from java

0xb36d6562: push    %ebp          --enter
0xb36d6563: mov     %esp,%ebp

0xb36d6565: push    %esi          --最后一个参数在堆栈中的位置

0xb36d6566: push    $0x0
0xb36d656b: mov     0x8(%ebx),%esi --ebx 指向 method, 0x8(%ebx) 指向
                                         ConstMethod
0xb36d656e: lea     0x30(%esi),%esi --在这一步让%esi 指向 codebase, 也就是
                                         method 的第一个字节码的位置
0xb36d6571: push    %ebx          --将 method 入栈

0xb36d6572: mov     0x10(%ebx),%edx --method 性能数据
0xb36d6575: test   %edx,%edx
0xb36d6577: je      0xb36d6580
0xb36d657d: add    $0x58,%edx
0xb36d6580: push    %edx          --method 性能数据

0xb36d6581: mov     0xc(%ebx),%edx --ConstMethod *
0xb36d6584: mov     0xc(%edx),%edx --ConstantPoolCache *
0xb36d6587: push    %edx

0xb36d6588: push    %edi          --argument word 1 所在堆栈位置 (即局部变量
                                         表第 1 个槽位位置)
0xb36d6589: push    %esi          --method 第一个字节码位置入栈

0xb36d658a: push    $0x0          --操作数栈栈底保留字段
0xb36d658f: mov     %esp,(%esp)  --将当前栈顶地址保存到操作数栈栈底 (帧数据栈
                                         顶就是操作数栈栈底)

```

当 `TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)` 函数执行完之后（是指其所生成的机器指令执行完成，该函数本身会在 JVM 启动之初被调用执行，但是只会生成对应的机器指令，并不会直接执行），物理寄存器又有一番变化，最终各个寄存器中所存储的结果如表 7.4 所示。

表 7.4 物理寄存器当前状态

寄存器名	指 向
edx	constantPoolCache
ecx	Java 函数入参数量
ebx	指向 Java 函数，即 Java 函数所对应的 methodOop 对象
esp	Java 方法栈的 fixed frame 顶部

续表

寄存器名	指 向
ebp	Java 方法调用方的栈底
esi	bcp
edi	locals pointer
eax	return address

其中，尤其要关注 esi 与 edi 这两个寄存器，接下来 HotSpot 执行 Java 字节码指令时，字节码对 Java 方法的局部变量表的读写主要就依靠 edi 来进行相对寻址，而 Hotspot 调用字节码指令的前提是得先定位到首个字节码指令在内存中的位置，这个位置就存储在 esi 寄存器中。

7.4.3 局部变量表

前面详细讲解了局部变量表的创建过程及组成，其实说白了，局部变量表就是 JVM 为 Java 方法内的变量在堆栈上所分配的一块连续的内存空间而已，这块连续的内存空间用于存储 Java 方法的入参数和局部变量。

1. 局部变量表的基本单位

在 JVM 内部，局部变量表按照“槽位”（slot）为基本单位进行划分。那么一个槽位 slot 的单位是多大呢？JVM 规范中给出了槽位单位的长度。

JVM 规定一个 slot 槽位应该能存放一个 boolean、byte、char、short、int、float、reference 或 returnAddress 类型的数据，其中 reference 是对象的引用，可以查到对象在 Java 堆中的实例的起始地址索引和方法区中的对象数据类型。returnAddress 是为字节码指令 jsr、jsr_w 和 ret 服务的，它指向了一条字节码指令的地址，其实说白了就是一个指针类型的数据。

但是 JVM 的这一对于 slot 长度的规范让人十分迷惑，因为在 JVM 内部，一个 boolean 类型与一个 float 类型所占的内存空间大小肯定是不同的，而 JVM 规范则规定一个 slot 槽位能够存放这两种数据类型的任意一种数据，那么 slot 到底是多大？

这是一个比较复杂的问题，各位道友继续往下看。

2. 局部变量表的读写与线程安全性

Java 程序员无法直接编写 Java 代码对局部变量表进行读写，毕竟这不是 Java 这种语言级别的数据结构概念，这种结构只能通过字节码指令进行访问和写入，并由 JVM 在运行期进行动态

读写。事实上，局部变量表作为堆栈的一部分，其实也决定了其只能由机器指令或者 JVM 这种能够创建和销毁堆栈的虚拟机器访问。

字节码指令也不能由 Java 程序员去编写，它由编译器生成。当涉及对局部变量表的访问时，编译器会生成 load 和 store 这样的指令，分别对局部变量表进行读和写。

例如下面这个简单的例子：

清单：Calculator.java

作用：计算器程序

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

编译这个类，并使用 javap 命令分析编译后的字节码文件，得到该示例中的 add(int, int) 方法的字节码指令是：

```
public static int add(int, int);
Code:
Stack=2, Locals=3, Args_size=2
0:   iload_0
1:   iload_1
2:   iadd
3:   istore_2
4:   iload_2
5:   ireturn
```

对于 add()方法中 `int sum = x + y` 这样的源代码，既涉及对两个入参 `x` 和 `y` 的读取，也涉及对 `sum` 局部变量的写入，编译器将其翻译成 `iload_0`、`iload_1`、`iadd` 和 `istore_2` 这 4 条字节码指令。由于变量 `x` 和 `y` 是 `add()` 方法的入参，因此其 slot 索引号分别是 1 和 2，所以 `iload_0` 和 `iload_1` 分别表示从局部变量表中读取变量 `x` 和变量 `y`。

当执行 `iadd` 指令完成 `x` 和 `y` 的求和运算之后，求和结果被存储到变量 `sum` 中。由于 `sum` 是 `add()` 方法中的第 3 个局部变量，因此其在局部变量表的 slot 的索引号为 2，所以执行完 `iadd` 指令后，便通过 `istore_2` 指令将求和结果存储到 `sum` 变量中。

局部变量表的大小（或谓深度）由编译器直接在编译期间计算出来，因此 Java 方法的入参和局部变量的 slot 索引号在编译期便确定下来，所谓 slot 索引号，说白了其实就是变量以方法栈帧中的某个点作为基准位置进行偏移，终究是堆栈空间的一部分，运行时无法修改，在这一点上，Java 语言与其他众多语言都保持一致，虽然其他语言中未必就有局部变量表的概念，但

是在堆栈上一定为其分配了空间。不过在 C/C++ 语言中，可以通过嵌入汇编脚本或者直接调用机器指令而动态扩展/收缩栈帧空间，改变栈顶和栈底位置，“世界尽在掌握中”。所涉虽远，然而 JVM 却是个中高手，在执行引擎中到处可见这类逻辑，如前文所讲执行引擎中的 CallStub 等例程都使用这种逻辑。

当在运行期完成上述 `int sum = x + y` 这段逻辑的字节码指令之后，会执行最后一条指令 `ireturn`，结束该方法的调用。`ireturn` 指令所对应的机器指令会将 JVM 为该方法所申请的栈帧空间销毁掉，由于局部变量表就包含在栈帧之中，因此栈帧都被销毁之后，局部变量表自然也被销毁，至此，局部变量表便完成其使命，轻轻地来，轻轻地走，不带走一片云彩，亦不留下丝毫痕迹。

至于 `iload` 和 `istore` 系列的字节码指令究竟如何完成局部变量表的读和写，则会在下文继续深入探讨，这里先熟悉下即可。

总体而言，局部变量表的生命周期包括以下几个环节：

(1) 在编译期间，编译器通过文法、语义解析，计算出一个 Java 方法所需的局部变量表大小，并写入 Java class 字节码文件的方法属性的 `code` 属性表中

(2) 在 JVM 加载 Java 类的时候，会解析 Java class 字节码文件中的方法信息，并解析出局部变量表的大小，如此便将局部变量表的大小这个数据从文件系统加载到内存中。

(3) 当 JVM 准备调用 Java 方法时，会为该方法创建栈帧，而栈帧中就包含局部变量表所需的空间。局部变量表的创建过程已在上文详细讲解过。这一步局部变量表终于横空出世，Java 方法的入参和内部的局部变量们终于有了“安身立命”的场所，终于有了一个“家”，并且一人一个，大家按照先来后到的顺序按序分配，谁也别抢，谁也别争。为了防止走错了家门，每个地址都被加了门牌号，这个门牌号就叫作“slot 索引”。真是一个“完美世界”。

(4) 当 JVM 具体执行 Java 方法时，便调用 Java 的 `iload` 和 `istore` 系列指令对栈帧中的局部变量表的空间进行不断读取和写入。由于有门牌号，因此大家并不会串错了门，一切都是井然有序。

(5) 当 JVM 执行完 Java 方法后，Java 方法的栈帧空间会被销毁，由于局部变量表就包含在栈帧空间内部，因此连同局部变量表一起被销毁。

由于局部变量表建立在堆栈空间上，因此是线程私有数据，所以 JVM 对其所进行的一切操作都不用考虑并发安全问题。

3. slot 大小

虚拟机通过门牌号——slot 索引号来统一区分局部变量表，Java 方法里的人参和局部变量

一人一个门牌号。但是人有胖瘦，局部变量有大小，这个大小由数据类型所决定。

JVM 规范规定一个 slot 槽位应该能存放一个 boolean、byte、char、short、int、float、reference 或 returnAddress 类型的数据。但是这些不同类型的数据所需的内存空间是不同的，而 slot 的基本单位是确定不变的，JVM 是如何保证这一点的呢？换言之，slot 的基本单位到底是多大，或者说一个槽位所占的内存空间到底是多大？

除了这一疑问，还有另一个更大的疑问，JVM 规范规定一个 slot 槽位能够存放一个 reference，但是同时又说如果是 long 型则需要 2 个 slot 来存放，问题是，在 64 位平台上，如果不开启指针压缩功能，则一个引用类型的变量所占的内存空间与 long 类型的变量应该是一样大小，都是 64 位，但是 JVM 的这一规范着实让人十分不解，一个 slot 对两种数据宽度完全一样的数据的要求不同，那么这个 slot 的基本长度究竟是多大？到底是按照 long 的标准还是按照 reference 这个引用类型的标准？

这个答案不知道，不过可以通过编写示例程序进行测试，因为 Java 类在编译时便能确定其局部变量表的大小，并能确定各个变量的索引号，通过索引号便能知道每一种变量类型究竟占多大空间了。示例程序如下：

清单：Calculator.java

作用：计算器程序

```
class Calculator{
    public long add(long x, long y) {
        long z = x + y;
        return z;
    }
}
```

本示例程序的 add() 是类成员方法，因此在编译期实际上会生成 3 个人参，其中第一个人参是隐式的 this 指针引用。add() 方法包含两个人参，其类型都是 long，同时 add() 方法内部的变量也是 long 类型。由于 add() 的第一个隐式人参 this 指针是一个引用类型 reference，而 add() 方法内部的局部变量都是 long 类型，因此通过 add() 方法所对应的字节码指令就能知道引用类型和 long 类型的变量究竟占多大空间。

编译该类，使用 javap 命令查看字节码，如下：

```
public long add(long, long);
Code:
Stack=4, Locals=7, Args_size=3
0:   lload_1
1:   lload_3
2:   ladd
3:   lstore 5
```

```

5:    lload   5
7:    lreturn

```

javap 命令作的输出结果显示, add()方法的 locals=7, 这表示 add()方法的局部变量表包含 7 个 slot 槽位, 这很好理解, 由于 add()方法包含 1 个 this 引用类型的人参和 3 个 long 类型的局部变量, 一个引用人参占 1 个槽位, 而每个 long 类型的变量占 2 个槽位, 因此一共需要 7 个槽位。同时 javap 的输出显示 add()方法的人参 args_size=3, 表示 add()方法包含 3 个人参, 这是因为编译器会将 this 指针当作第一一个参。从 locals 和 args_size 这两个输出可以知道, JVM 内部的确让引用类型只占 1 个 slot, 而让 long 类型的数据占用 2 个 slot, 这与 JVM 的规范是一致的。

接着观察 add()方法的指令进行进一步确认。第 1 条字节码指令是 lload_1, 这表示读取 add()方法的第 1 个人参 x, 其实此时 add()方法的第 0 个人参是隐式的 this 指针, this 指针的 slot 索引是 0。而第一个参 x 的 slot 索引为 1, 这的确表示 this 指针仅占用 1 个 slot。add()方法的第 2 条字节码指令是 lload_3, 该指令读取 add()方法的第 2 个人参 y, 注意此时 lload 后面的操作数变成 3, 表明第 1 个人参 x 占用了 2 个 slot, 否则如果只占用 1 个 slot, 则人参 y 的读取指令应该为 lload_2。

剩下的字节码指令各位道友自行推敲。总之无论从 locals 与 args_size 这些参数值, 还是从字节码指令看, 验证结果都与 JVM 的规范是完全一致的。不过这更加深了疑惑, 在 64 位平台上, 引用类型 reference 的变量所占的内存空间应该与 long 类型的数据是一致的, 但是为什么它们所占的 slot 槽数不同, 这里面究竟有何秘密?

JVM 规范并没有给出答案, 不过可以换个思路进行验证。JVM 将局部变量表的每个 slot 都编上了门牌号, Java 方法的每个人参和每个局部变量都有一个唯一的门牌号, JVM 在运行期会将这个“门牌号”转换为机器指令中内存数据传送时的偏移量, 可以通过偏移量观察到每个数据类型所占据的真实的内存空间大小。JVM 在创建 Java 方法栈帧时, 将局部变量表的起始位置保存到 edi 寄存器中, 并且局部变量表的所谓起始位置其实是 Java 方法第一一个参的内存位置, 这在前文有所提及。JVM 准备调用一个 Java 方法之前, 会将 Java 方法的 N 个人参进行压栈, 复制到局部变量表中, 但是此时 JVM 并未关注这些人参的实际类型, 而是将其统一当成指针类型进行处理, 因此在 32 位平台上, 这第一一个参在堆栈的内存位置, 其实是按照其数据宽度为 4 字节进行计算的, 这里就存在一个问题: 如果 Java 方法的第一一个参类型是 long, 则最终存储到 edi 寄存器中的所谓第一一个参的内存位置, 便不能代表真实的第一一个参的内存位置。如下面这个例子:

清单: Calculator.java

作用: 计算器程序

```

class Calculator{
    public static long add(long x, long y, long z){

```

```

    long sum = x + y;
    return sum;
}
}

```

该示例中的 `add(long, long, long)` 是个静态方法，因此没有隐式的 `this` 入参，其第一个入参就是声明中的 `long x`。JVM 准备调用该方法之前，会初始化该方法的局部变量表，初始化的总体思路在前文已经详细描述过，总体上分为两步（32位平台）：

（1）先申请 N 个指针宽度的堆栈空间， N 为 Java 方法入参数量。对于本例而言，由于包含 3 个入参，同时在 32 位平台上，因此 JVM 先分配 12 字节内存空间。

（2）接着申请 $(\text{maxLocals} - \text{sizeOfParameters})$ 个指针宽度的堆栈空间。对于本例而言，使用 `javap` 命令查看编译后的字节码文件可知，其 `maxLocals=8`, `sizeOfParameters=3`，因此 JVM 会接着申请 5 个指针宽度的内存空间，在 32 位平台上，该内存空间一共包含 20 字节。

在第一步中，JVM 会顺便计算第一个人参的内存位置，并将其保存到 `edi` 寄存器中。但是这里所调的第一个人参的内存位置，并非是该入参的真正的内存位置，因为此时仅仅是先申请堆栈空间并全部初始化为 0，尚未运行 Java 字节码指令将真正的局部变量存储进来。此时所谓第一个人参的位置，其实是 JVM 将各个人参当作指针看待时的位置。这里是一个关键点。

当上面这两步都执行完之后，JVM 便为 `add(long, long)` 方法分配好局部变量表的空间，其内存布局如图 7.27 所示。

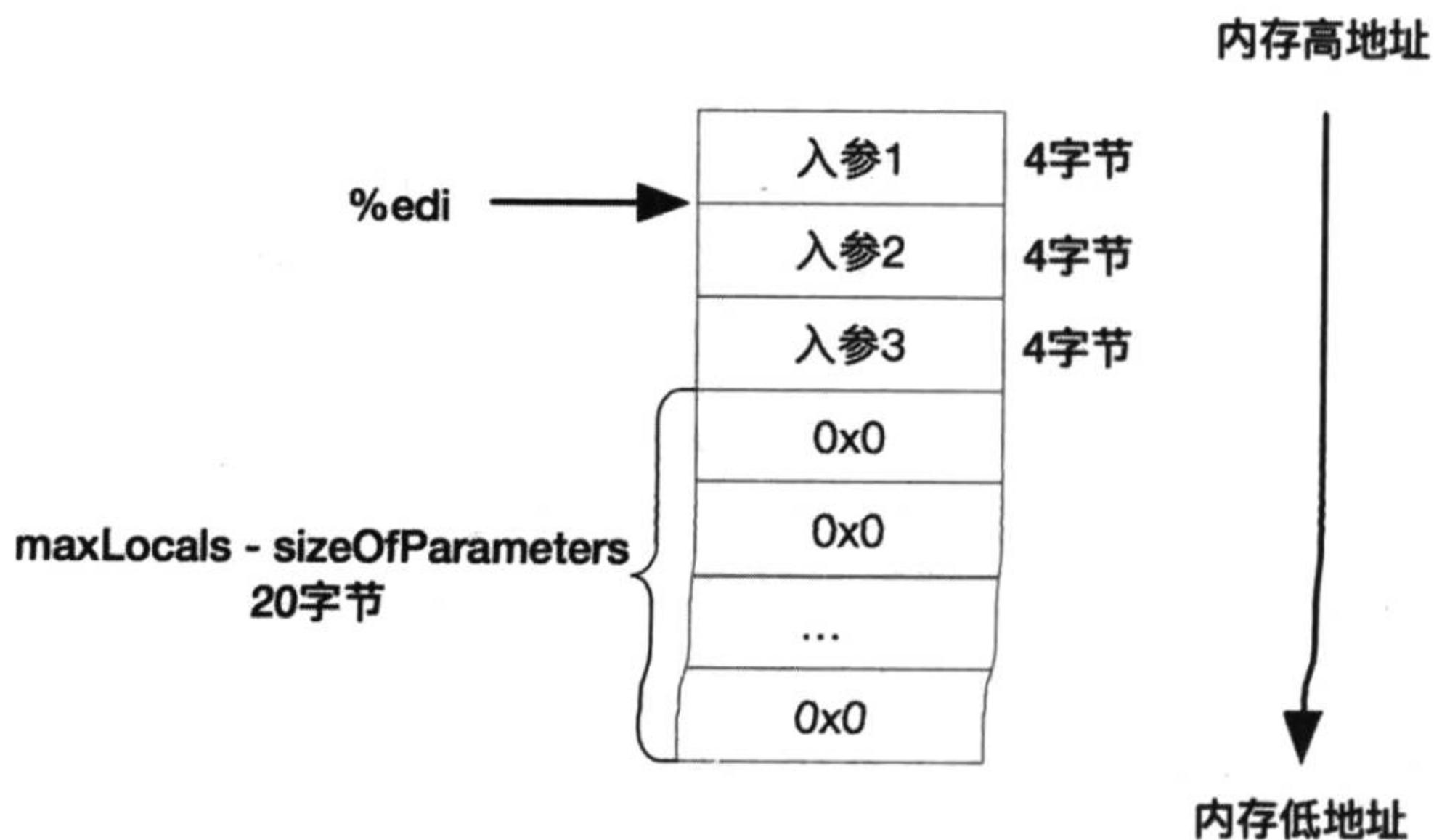


图 7.27 `add()` 方法的局部变量表初始化时的内存布局及 `edi` 寄存器指向位置

但是 `add(long, long, long)` 方法的第一个人参类型其实是 `long`, `long` 类型的数据需要占据 8 字节内存空间，因此该入参的真实内存起始位置应该是图 7.28 所示入参 2 所在的内存位置。

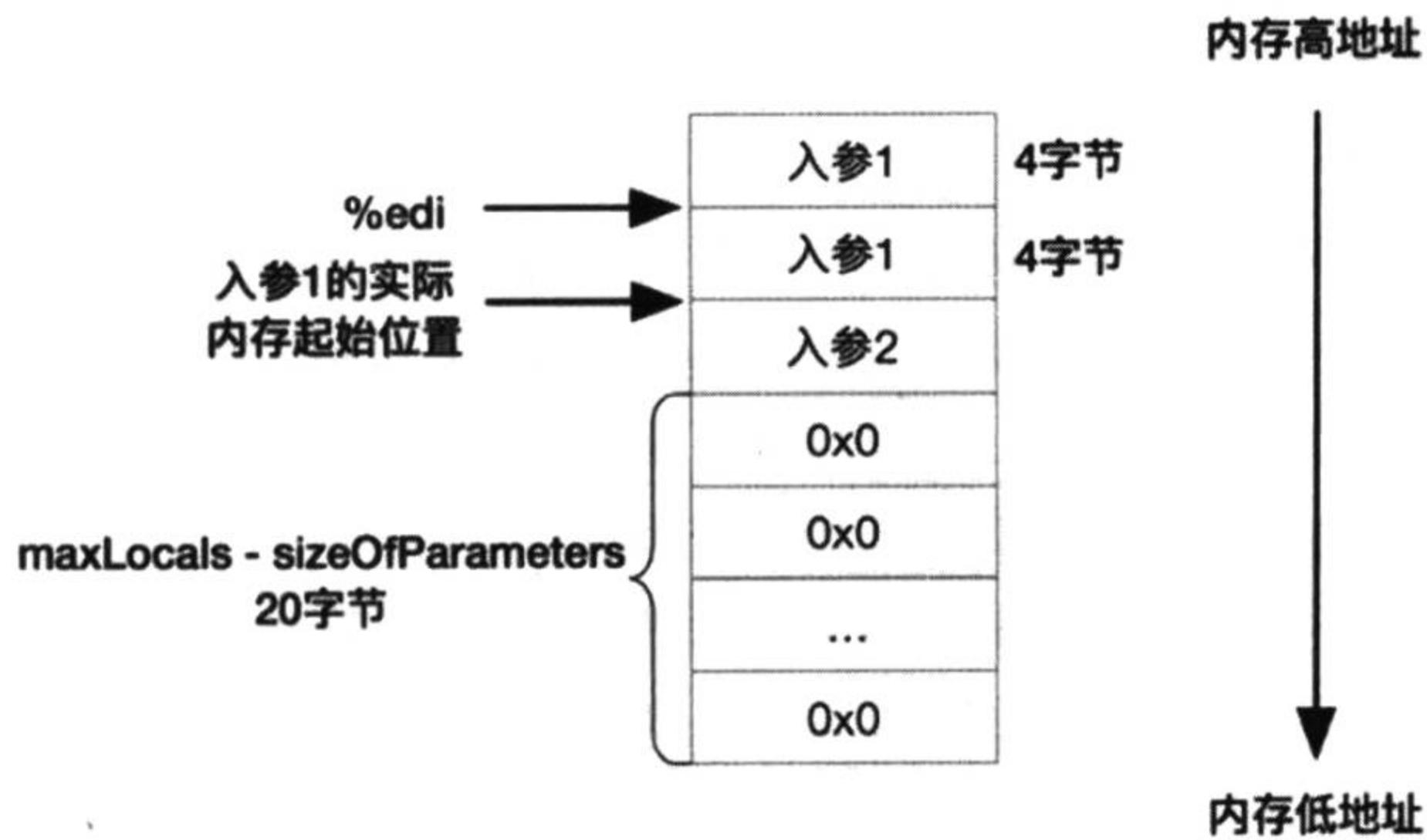
图 7.28 `add()`方法第一个入参的实际内存位置

图 7.28 给出一个很明显的信息：`edi` 寄存器所存储的位置，并非是局部变量表中第一个变量的实际内存位置。这种特性直接影响到 Java 字节码指令中读写局部变量表的 `load` 和 `store` 系列指令所对应的机器码层面的处理，因为 `load` 和 `store` 系列指令所对应的机器码在读写局部变量表时，实际上将 `edi` 寄存器所指的位置当作基准偏移位置，而现在 `add(long, long, long)` 方法的第一个 `long` 类型的入参的实际内存位置与 `edi` 寄存器这个基准位置不同，最终反映到机器码层面，必定要基于 `edi` 进行偏移。读取局部变量表中第一个 `long` 类型的数据的字节码指令是 `lload_0`，且看该指令在 32 位平台上所生成的机器指令：

```
lload_0 30 lload_0 [0xb36d8820, 0xb36d8860] 64 bytes

[Disassembling for mach='i386']
0xb36d8844: mov    -0x4(%edi),%eax
0xb36d8847: mov    (%edi),%edx
0xb36d8849: movzbl 0x1(%esi),%ebx
0xb36d884d: inc    %esi
0xb36d884e: jmp    *-0x48f102a0(,%ebx,4)
```

注意，第一条机器码指令是“`mov -0x4(%edi), %eax`”，机器码果然从 `edi` 寄存器所指的下一个 4 字节的位置开始读取局部变量表中的 `long` 类型的数据，并将其保存到 `eax` 寄存器中。由于在 32 位平台上，一次 `mov` 指令最大只能传送 4 字节数据，因此这里对于 `long` 类型的数据连续使用了 2 条 `mov` 指令，连续读取 2 个 4 字节数据并分别保存到 `eax` 与 `ebx` 寄存器中。这里其实是使用了栈顶缓存技术，这里暂且不表，总之由此可以验证，`edi` 寄存器中所存储的的确不是第一个人参的真实内存起始位置。对于这一点，还有一个验证点，如果 Java 方法的第一个人参类型是 `int`，在 32 位平台上，Java 中的一个 `int` 类型数据占 4 字节内存空间，这与指针类型的数据宽度是一致的。那么如果 Java 方法的第一个人参若果真是 `int` 类型，则 `edi` 寄存器所指向的位置便与该入参的真实内存位置相同。若 Java 中读取局部变量表第一个数据且类型是 `int` 的字节

码指令是 `iload_0`，则该字节码指令所对应的机器指令应该如下：

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

查看 JVM 在 32 位平台上字节码指令所对应的汇编指令，`iload_0` 指令如下：

```
iload_0 26 iload_0 [0xb36d86a0, 0xb36d86e0] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d86c4: mov    (%edi),%eax
0xb36d86c6: movzbl 0x1(%esi),%ebx
0xb36d86ca: inc    %esi
0xb36d86cb: jmp    *-0x48f106a0(,%ebx,4)
```

结果果然与预测中的完全一致！所以如果 Java 方法的第一个入参类型是 `int` 类型，则 `edi` 寄存器所指的内存位置便与第一个人参的真实内存位置保持一致。

搞清楚了这一层的原理（前文说这是巨坑，道理就在这里，大脑得学会急转弯，否则就要撞树了），接着便可以验证 JVM 规范中所说的一个 slot 究竟占多大内存空间的问题了。

在 32 位平台上，一个 slot 能够存放一个 `int` 类型，也能存放一个 `reference` 类型，因此如果一个 Java 方法的第 1 和第 22 个人参的类型都是 `int` 类型，则读取局部变量表中这 2 个人参的字节码指令分别是 `iload_0` 和 `iload_1`，如果要验证 32 位平台上一个 slot 槽位究竟占多大内存空间，只需要分析 `iload_0` 与 `iload_1` 这 2 条字节码指令所对应的机器指令中，分别相对于 `edi` 寄存器做多大的偏移量即可。刚刚展示了 `iload_0` 字节码指令所对应的机器指令，那么在 32 位平台上，`iload_1` 字节码指令所对应的机器指令如下：

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

[Disassembling for mach='i386']
0xb36d8724: mov    -0x4(%edi),%eax
0xb36d8727: movzbl 0x1(%esi),%ebx
0xb36d872b: inc    %esi
0xb36d872c: jmp    *-0x48f106a0(,%ebx,4)
```

可以看到，最终机器指令从局部变量表中读取第二个且数据类型是 `int` 类型的变量时，使用了“`mov -0x4(%edi), %eax`”这条机器指令，这表示第二个 `int` 类型的变量在局部变量表中的起始位置相对于 `edi` 所指的位置偏移了 `-0x4` 字节，而这个偏移距离正好是 32 位平台上一个 slot 槽位的大小，由此可以验证 32 位平台上一个 slot 槽位占 4 字节内存空间。由于在 32 位平台上，一个指针类型的数据宽度也是 4 字节，而 Java 内部的 `reference` 引用类型的数据其实就是一个指针，因此 32 位平台上的 1 个 slot 槽位也能容纳下一个 `reference` 类型的数据。

按照这种思路，可以顺便验证下 32 位平台上一个 long 类型的数据是否的确需要两个 slot 槽位。

按照同样的思路，验证最让人疑惑的 64 位平台上的 slot 大小。由于 JVM 规范规定一个 slot 能够容纳一个 int 类型的数据，并无 32 位与 64 位平台之分，因此即使在 64 位平台上，一个 int 类型的数据仍然只需要 1 个 slot 即可。要验证 64 位平台上 1 个 slot 大小，只需要看 iload_0 与 iload_1 这 2 条字节码指令所对应的机器码相对于 edi 寄存器的偏移量，这个偏移量便是 slot 的大小。在 64 位平台上，iload_0 这条字节码指令所对应的机器码如下：

```
iload_0 26 iload_0 [0x000000010439f7e0, 0x000000010439f840] 96 bytes

0x000000010439f810: mov    (%edi),%eax
0x000000010439f813: movzbl 0x1(%esi),%ebx
0x000000010439f818: inc    %esi
0x000000010439f81b: jmp    *-0x103ceb100(,%ebx,8)
```

由第一条机器码可知，iload_0 指令读取局部变量表第一个且类型是 int 型的变量时，相对于 edi 寄存器的偏移量是 0。

64 位平台上，iload_1 字节码指令所对应的机器码如下：

```
iload_1 27 iload_1 [0x000000010439f860, 0x000000010439f8c0] 96 bytes

0x000000010439f890: mov    -0x8(%edi),%eax
0x000000010439f894: movzbl 0x1(%esi),%ebx
0x000000010439f899: inc    %esi
0x000000010439f89c: jmp    *-0x103ceb100(,%ebx,8)
```

由第一条机器码可知，iload_0 指令读取局部变量表第一个且类型是 int 型的变量时，相对于 edi 寄存器的偏移量是-0x8。而上面 iload_0 指令相对于 edi 的偏移量是 0，由此可知 iload_0 与 iload_1 这 2 条字节码指令所读取的数据的内存位置的相对偏移量为 8 字节，而这正是 1 个 slot 的大小。

在进一步验证 64 位平台上的 slot 大小之前，有必要弄清楚在 64 位平台上 edi 寄存器所指的位置。当 JVM 准备执行一个 Java 方法时，先为其创建局部变量表并初始化为 0。还是以上的 Calculator.add(long x, long y, long z)方法为例，该方法的局部变量表创建完成之后，其内存布局及 edi 寄存器所指位置如图 7.29 所示。

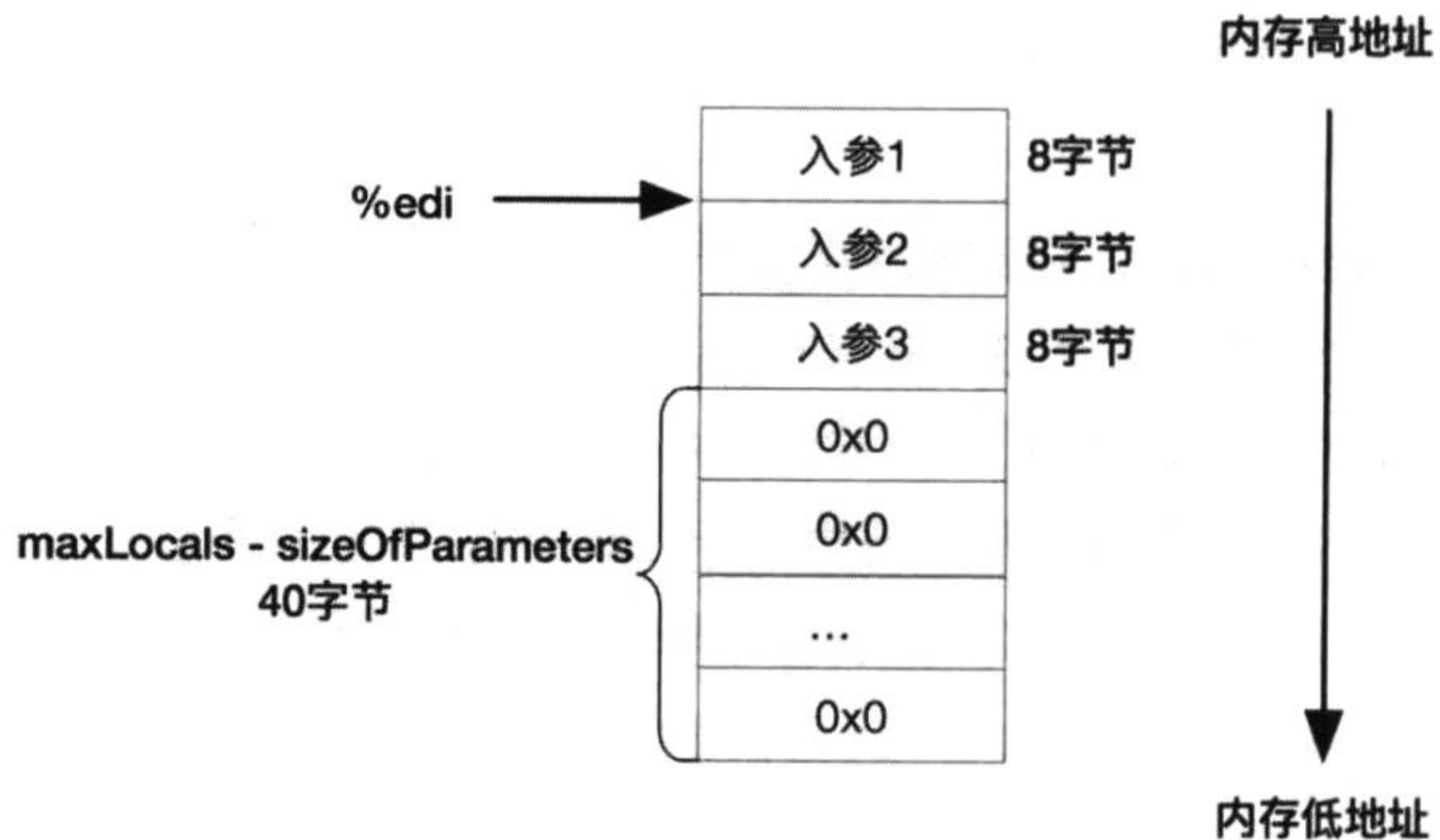


图 7.29 64 位平台上，add()方法的局部变量表初始化时的内存布局及 edi 寄存器指向位置

64 位平台与 32 位平台最大之不同便在于，在 JVM 创建局部变量表时，JVM 仍然将每一个入参当作一个指针类型的数据处理，但是一个指针在 64 位平台上占 8 字节，因此 edi 寄存器所指的位置，其后面其实能够容纳一个 long 类型的数据（即入参 1 的大小）。那么如果 Java 方法的第一个入参类型是 long，则 lload_0 字节码指令所对应的机器码应该直接从 edi 寄存器所指的位置处开始读取数据，那么来看下 64 位平台上，lload_0 这条字节码指令所对应的机器码，如下所示：

```

lload_0 30 lload_0 [0x000000010439f9e0, 0x000000010439fa40] 96 bytes

0x000000010439fa10: mov    -0x8(%edi),%rax
0x000000010439fa14: movzbl 0x1(%esi),%ebx
0x000000010439fa19: inc    %esi
0x000000010439fa1c: jmp    *-0x103ceb100(,%ebx,8)

```

关注第一条机器指令“`mov -0x8(%edi), %rax`”，这个结果令人吃惊，与预测的完全不同，`lload_0` 竟然不是从 edi 寄存器所指的位置开始读取 long 类型的数据，而是以 edi 所指位置为基准又向低地址方向偏移了 8 字节。由此可见，在 64 位平台上，一个 long 类型的数据在局部变量表中似乎占据了 16 字节的大小。不过这与 JVM 规范倒是完全相符，因为刚才验证过在 64 位平台上，一个 slot 的宽度为 8 字节，而 JVM 规范又规定存储一个 long 类型的数据需要 2 个 slot，而 2 个 slot 的宽度就是 16 字节。由此，谜底终于揭开了，JVM 的规范一点没错，无论是在 32 位平台还是在 64 位平台上都会一样生效。同时，这也解释了 64 位平台上的 reference 引用类型数据所占的 slot 数量为 1 的原因，因为虽然在 64 位平台上，一个引用类型本质上是一个指针类型，其所需内存大小按理应该与一个 long 类型的数据所需的内存大小相同，但是在 JVM 的局部变量表中，这两种类型的数据所需的 slot 数是不同的，这是由早期的 JVM 规范所规定的，所以到了 64 位平台上的 JVM，只能将一个 slot 实现为占据 8 字节大小的内存区域，如此才能让一个 slot 能容纳一个引用类型的数据。

不过在 JVM 内部，long 类型的变量也只有在局部变量表中才会占据 16 字节，而在堆内存中，该类型的数据该占据多大的内存空间，还是占据多大的内存空间。例如，如果一个 Java 类的成员变量类型是 long 类型，则该变量会被分配在堆内存中，在堆内存中，该变量就只占 8 字节。从这个角度来看，64 位平台上的 JVM 的局部变量表对内存存在一定的空间浪费。

这里有个问题，可能有些道友认为既然可以使用 iload_0 与 iload_1 这两条字节码指令所对应的机器码相对于 edi 寄存器的偏移量之差来确定 slot 的大小，那么也应该能使用 lload_0 与 lload_1 这两条字节码指令来确定。例如，上面给出了 lload_0 字节码指令相对于 edi 的偏移量为 -0x8，而 lload_1 字节码指令在 64 位平台上所对应的机器码如下：

```
lload_1 31 lload_1 [0x000000010439fa60, 0x000000010439fac0] 96 bytes

0x000000010439fa90: mov    -0x10(%edi),%rax
0x000000010439fa94: movzbl 0x1(%esi),%ebx
0x000000010439fa99: inc    %esi
0x000000010439fa9c: jmp    *-0x103ceb100(,%ebx,8)
```

这条字节码指令从相对于 edi 偏移量为 -0x10 的位置开始读取局部变量。由于 lload_0 从相对于 edi 偏移量为 -0x8 的位置开始读取，这两个偏移量之差为 0x8，即 8 字节，由此得出 long 类型的数据在局部变量表中只需占据 8 字节大小的结论。很显然，这个结论是错误的，其根本原因在于，如果一个 Java 方法编译后能够产生 lload_1 这样的字节码指令，则说明该方法的局部变量表的第二个变量类型是 long，并且其索引号是 1，那么便说明 Java 方法的局部变量表中的第一个变量只占 1 个 slot 槽位，因此第一个变量类型一定是 int、reference 等类型，而绝不可能是 long 类型，否则第二个变量的读取指令就应该是 lload_2。如果第一个变量类型是 int，则其读取指令是 iload_0，因此需要比较 iload_0 与 lload_1 这两条字节码指令所读取的相对于 edi 寄存器所指内存的偏移量之差，由此才能确定局部变量表中一个 long 类型的数据所占据的 slot 槽数。很显然，iload_0 字节码指令从相对于 edi 偏移量为 0 的位置开始读取局部变量，而 lload_1 从相对于 edi 偏移量为 -0x10 的位置开始读取局部变量，这两者的偏移量之差为 0x10，即 16 字节，因此也能验证出一个 long 类型的数据在局部变量表中占据 16 字节的内存大小。这个结果与上述的验证结果完全一致。

至此，关于 JVM 内部局部变量表中的 slot 到底是多大的问题便验证完毕。总结如下：

- ◎ 32 位平台上，一个 slot 大小为 4 字节。
- ◎ 64 位平台上，一个 slot 大小为 8 字节。
- ◎ 64 位平台上，long 类型数据在局部变量表中占据 16 字节的内存空间，但是在堆内存中该占据多大内存空间还是占据多大。

不过根据 JVM 规范，还有一个疑问，那就是一个 slot 能够容纳一个 int、reference 等类型

的数据，还能容纳一个 short、char 等类型的数据。由于 char、short 所需内存空间，小于 int 与 reference 类型所需的内存空间，那么对于这类窄数据，JVM 如何在局部变量表中进行存取呢？看下面示例：

清单：Calculator.java

作用：计算器程序

```
class Calculator{
    public static int mixed(short s, char c){
        s = 32;
        c = 'D';
        short ss = (short)(s + (short)3);
        char cc = (char)(c + (char)2);
        int i = s + c;
        return i;
    }
}
```

编译程序，并使用 javap 命令分析编译后的字节码文件，javap 命令的分析结果输出如下：

```
public static int mixed(short, char);
Code:
Stack=2, Locals=5, Args_size=2
0:   bipush  32
2:   istore_0
3:   bipush  68
5:   istore_1
6:   iload_0
7:   iconst_3
8:   iadd
9:   i2s
10:  istore_2
11:  iload_1
12:  iconst_2
13:  iadd
14:  i2c
15:  istore_3
16:  iload_0
17:  iload_1
18:  iadd
19:  istore  4
21:  iload   4
23:  ireturn
```

从 javap 命令的分析结果可以看出，对于 char 和 short 类型的数据，无论是读还是写，所使用的指令全都是 iload 与 istore 系列的指令，因此在 JVM 内部，对于数据宽度小于 int 类型的数据类型，仍然将其处理成 int 类型的数据，并使用基于 int 类型数据的读写指令对 char、short 等类型的数据进行读写。有兴趣的道友可以自行设计程序验证 boolean 与 byte 类型的数据所对应的读写字节码指令是什么。

7.5 栈帧深度与 slot 复用

通常一个 Java 程序会包含多个线程，每个线程都会包含若干方法栈帧，这些若干方法栈帧就组成了线程的堆栈空间。线程堆栈空间（stack space）不会无限制地增长，而是会受到约束，直接由操作系统加载的软件程序的线程堆栈空间大小会受到操作系统层面所设置的栈空间大小限制，而对于 Java 线程，同时还会受到 JVM 虚拟机所设置的堆栈空间大小限制。

正是因为堆栈空间大小会受到限制，所以当一个线程中所调用的方法太深时，导致 JVM 所分配的栈帧太多，就有可能耗尽 stack space，从而抛出 stackOverflow 异常。所以合理地设置默认堆栈空间大小是一门学问。在 JVM 中，可以通过 XSS 来设置默认的堆栈空间大小。

前面讲过，Java 方法栈帧由三大部分所组成：局部变量表、固定帧和操作数栈。固定帧的大小是固定不变的，无论所调用的是何种 Java 方法。因此 Java 方法栈帧的大小取决于局部变量表和操作数栈的大小，而由于调用者方法的操作数栈会作为被调用者方法的局部变量表的一部分（栈帧重叠），因此可以认为操作数栈属于被调用者方法的栈帧的一部分，由此看来，一个 Java 方法的栈帧大小主要取决于局部变量表的大小，而 Java 方法的局部变量表主要由两部分组成：一是 Java 方法入参；二是 Java 方法局部变量。

因此，Java 方法的栈帧大小最终取决于 Java 方法的入参和局部变量的大小。为何要分析这个问题呢？实在是因为这与线程堆栈的合理运用存在着莫大的关系。当 JVM 设定默认的堆栈空间大小后，一个 Java 线程所能调用的最大方法深度便直接取决于 Java 方法局部变量表的大小。如果 Java 方法的局部变量表所占的空间大，则 Java 线程所能调用的最大方法深度便会变小。

例如下面这个示例：

清单：TestXSS.java

作用：演示 Java 堆栈空间大小与局部变量表的关系

```
public class TestXSS implements Runnable{
    public static void main(String[] args)
        Thread t = new Thread(new TestXSS());
        t.start();
```

```

    }

    public static void test(long a) {
        long b = a + 3;

        System.out.println(a++);
        test(a);
    }

    @Override
    public void run() {
        test(1);
    }
}

```

该程序主要通过递归调用 `test(long)` 函数，来演示递归多少次之后会将线程堆栈空间耗尽。

设置 JVM 的 XSS 大小为 200KB，运行该程序，输出如下：

```

//...
950
951
952
953
Exception in thread "Thread-0" java.lang.StackOverflowError
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeArrayLoop(DoubleByte.java:578)
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeLoop(DoubleByte.java:617)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
    at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
    at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
    at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
    // ...

```

可见，当递归调用 953 次之后，线程的堆栈空间终于被耗尽。当堆栈空间被耗尽后，JVM 便会抛出 `java.lang.StackOverflowError` 异常。

接着对上面的示例程序稍做修改，主要修改 `test(long)` 方法，修改后的方法如下：

清单：TestXSS.java

作用：演示 Java 堆栈空间大小与局部变量表的关系

```

public static void test(long a) {
    long b = a + 3;
    long c = b - a + b *5;
    long d = a & 6;

    System.out.println(a++);
    test(a);
}

```

修改后的 test(long)方法内部另外声明了两个局部变量，并且都是 long 类型的。仍然将 JVM 的 XSS 参数设置为 200KB，运行修改后的程序，输出结果如下：

```
//...
739
740
741
742
743
Exception in thread "Thread-0" java.lang.StackOverflowError
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeArrayLoop(DoubleByte.java:578)
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeLoop(DoubleByte.java:617)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
    at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
    at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
    at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
// ...
```

现在可以看到，test(long)函数仅递归调用了 743 次便耗尽了线程的堆栈空间。由此可见，当 Java 方法的局部变量表增大后，的确会减少方法调用的深度。各位道友可以继续在 test(long)方法内部定义更多的局部变量并测试所能递归调用的最大次数。

由于局部变量表的大小直接影响到一个线程所能调用的方法深度，因此在声明方法局部变量时，应该尽量使 slot 能够复用。所谓 slot 复用，便是让方法内部的不同变量能够占用局部变量表中的同一个槽位，这样便能减小局部变量表的大小，从而提高一个线程能调用的最大方法深度。

仍然以上面修改后的 test(long)方法作为示例，该方法内部包含 3 个局部变量，分别是 b、c 和 d，但是仔细观察可以发现，从源程序的 d 变量声明开始，一直到方法结束，a 和 b 变量都没有再被使用到，因此可以使用花括号 {} 将 a 和 b 变量的声明语句括起来，如下：

清单：TestXSS.java

作用：演示 Java 堆栈空间大小与局部变量表的关系

```
public static void test(long a) {
{
    long b = a + 3;
    long c = b - a + b *5;
}

long d = a & 6;

System.out.println(a++);
test(a);
}
```

现在仍然将 JVM 的 XSS 设置为 200KB，运行该示例程序，输出结果如下：

```
// ...
831
832
833
834
835
836
Exception in thread "Thread-0" java.lang.StackOverflowError
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeArrayLoop(DoubleByte.java:578)
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeLoop(DoubleByte.java:617)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
    at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
    at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
    at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
    at java.io.BufferedWriter.flushBuffer(BufferedWriter.java:129)
    at java.io.PrintStream.write(PrintStream.java:526)
// ...
```

从输出结果可以看出，`test(long)`方法被递归调用的最大次数相比于刚才的 743 次增大到 836 次，由此可见增加花括号的方式的确有效。这是因为使用花括号将前面两个变量括起来之后，Java 编译器便会认为其作用域不会超出花括号的范围，因此在出了花括号的范围之后，JVM 便会清空这两个变量所占用的 slot 槽位，空出来给 `test(long)`方法内部后续的变量复用。各位道友可以通过 `javap` 命令分别分析使用花括号前后，`test(long)`方法的 locals 值。

7.6 最大操作数栈与操作数栈复用

与 Java 方法堆栈息息相关的一个重要参数是 `max stack`，即最大操作数栈。Java 虚拟机的指令集基于栈，所有的计算逻辑都需要通过栈来完成，这个栈便是“操作数栈”，JVM 内部也叫“表达式栈”。操作数栈的大小由 Java 编译器在编译期计算，但是编译器只能计算出一个“最大”的栈深度，这是因为操作数栈与 slot 槽一样，也可以实现复用。而之所以要实现复用，是为了节省宝贵的内存空间。最大操作数栈被作为 Java class 字节码文件内部 `Code` 属性区的一部分，Java 源文件被编译后，使用 `javap -v` 命令可以查看每一个 Java 方法的最大操作数栈大小，例如下面这个例子：

清单：Test.java

作用：演示 Java 最大操作数栈

```
public class Test {
    public static void main(String[] args) {
```

```

    int a = 0;
}
}
}

```

使用 javap -v 命令分析编译后的 class 字节码文件，输出如下信息：

```

public static void main(java.lang.String[]);
Code:
Stack=1, Locals=2, Args_size=1
0:  iconst_1
1:  istore_1
2:  return

```

打印结果中的“stack=1”，表示本程序的 main() 主函数的最大操作数栈空间只需要 1 个即可。stack 的数据宽度与 slot 槽位宽度保持一致，因此这里也可以说，main() 主函数的最大操作数栈空间为 1 个槽位。在本示例程序中，main() 主函数的字节码指令中包含“iconst_1”，该指令会将自然数 1 推送至栈顶，因此 main() 主函数需要 1 个槽位大小的操作数栈空间。

修改上面的示例程序，变成如下：

清单：Test.java

作用：演示 Java 最大操作数栈

```

public class Test {
    public static void main(String[] args) {
        int a = 1;
        int b = 1;
    }
}

```

现在在 main() 主函数中定义了 2 个变量 a 和 b，使用 javap 命令分析后得知 main() 主函数的最大操作数栈仍然是 1，这是因为当 JVM 执行 main() 函数中的“int a=1”这条指令时需要将自然数 1 推送至栈顶，但是当执行完之后，栈顶的自然数 1 会从栈顶被传送至局部变量表中，因此当执行“int b=1”时，JVM 便可以复用栈顶的 1 个空间。

接着看下面这个示例：

清单：Test.java

作用：演示 Java 最大操作数栈

```

public class Test {
    public static void main(String[] args) {
        int a = 1;
        int b = 1;

        add(a, b);
    }
}

```

```

public static void add(int m, int n) {
    int a = m + n;
}
}

```

本示例与上面的示例相比，在 main() 函数里面多了一步——调用 add() 方法。使用 javap 命令分析，输出结果如下：

```

public static void main(java.lang.String[]);
Code:
Stack=2, Locals=3, Args_size=1
0:  iconst_1
1:  istore_1
2:  iconst_1
3:  istore_2
4:  iload_1
5:  iload_2
6:  invokestatic #2; //Method add:(II)V
9:  return

```

可以看到，现在 Stack 的值为 2。为何会是 2 呢？这主要是因为 main() 主函数调用了 add() 方法。由于 add() 方法包含 2 个入参，因此当 main() 函数调用 add() 方法时，需要将 add() 方法所需的两个实参推送至 main() 方法的操作数栈栈顶。JVM 在执行 Java 方法调用时，实现了“堆栈重叠”技术，因此这两个栈顶实参将被当作 add() 方法局部变量表的一部分。在本例中，如果没有操作数栈复用技术，则 main() 函数的最大操作数栈一定为 4，但是由于“int b=1” 指令复用了“int a=1” 指令的操作数栈空间，而“add(a,b)” 指令又复用了“int b=1” 指令的操作数栈空间，因此最终 main() 函数只需要分配 2 个槽位大小的操作数栈空间，便能满足全部指令的逻辑计算。由此可以明白 Java 方法的操作数栈之“最大”的含义——这个“最大”实则是在内存复用的基础上，从一个 Java 方法所有指令中选出一个需要占用最多操作数栈空间的指令，以该指令所需的操作数栈空间作为一个 Java 方法的“最大”操作数栈空间。

明白了这个道理，各位道友可以猜一猜下面这个示例中 main() 主函数的最大栈是多少：

清单：Test.java

作用：演示 Java 最大操作数栈

```

public class Test {
    public static void main(String[] args) {
        int a = 1;
        int b = 1;

        add(a, b);
        add(a, a, a);
    }
}

```

```
public static void add(int m, int n){  
    int a = m + n;  
}  
  
public static void add(int m, int n, int x){  
    int a = m + n + x;  
}  
}
```

7.7 本章总结

如果你能阅读到这里，说明你的功力非常深厚，或者你是一个十分有耐心的道友。耐得住寂寞，才能守得住繁华，有耐心的人必有所成就！

本章的难度在全书而言，属于难度系数最高的部分，一方面，涉及函数指针，另一方面，Java方法栈帧的创建全依赖机器指令，这对于一般人而言，几乎等同于天书。在前面讲解 CallStub 例程的章节中，这两方面大家都见识过，然而，本章另一个难点在于栈帧的结构。

本章全面分析了 Java 方法栈帧创建的过程，机器指令几乎是逐个讲解的。然而，笔者在阅读 JVM 的这部分机制指令的过程中，不仅仅停留于分析机器指令本身的含义，还进行了更深入的思考，仔细推敲了每一条机器指令的背景，为何要这么实现，如果不这么实现有没有问题。相信真正有耐心读下来并且能够读懂的道友能够体会笔者的这种深入思考！

这也是本书区别于其他书籍的最大不同点，不仅追求“知其然”，更加追求“知其所以然”。