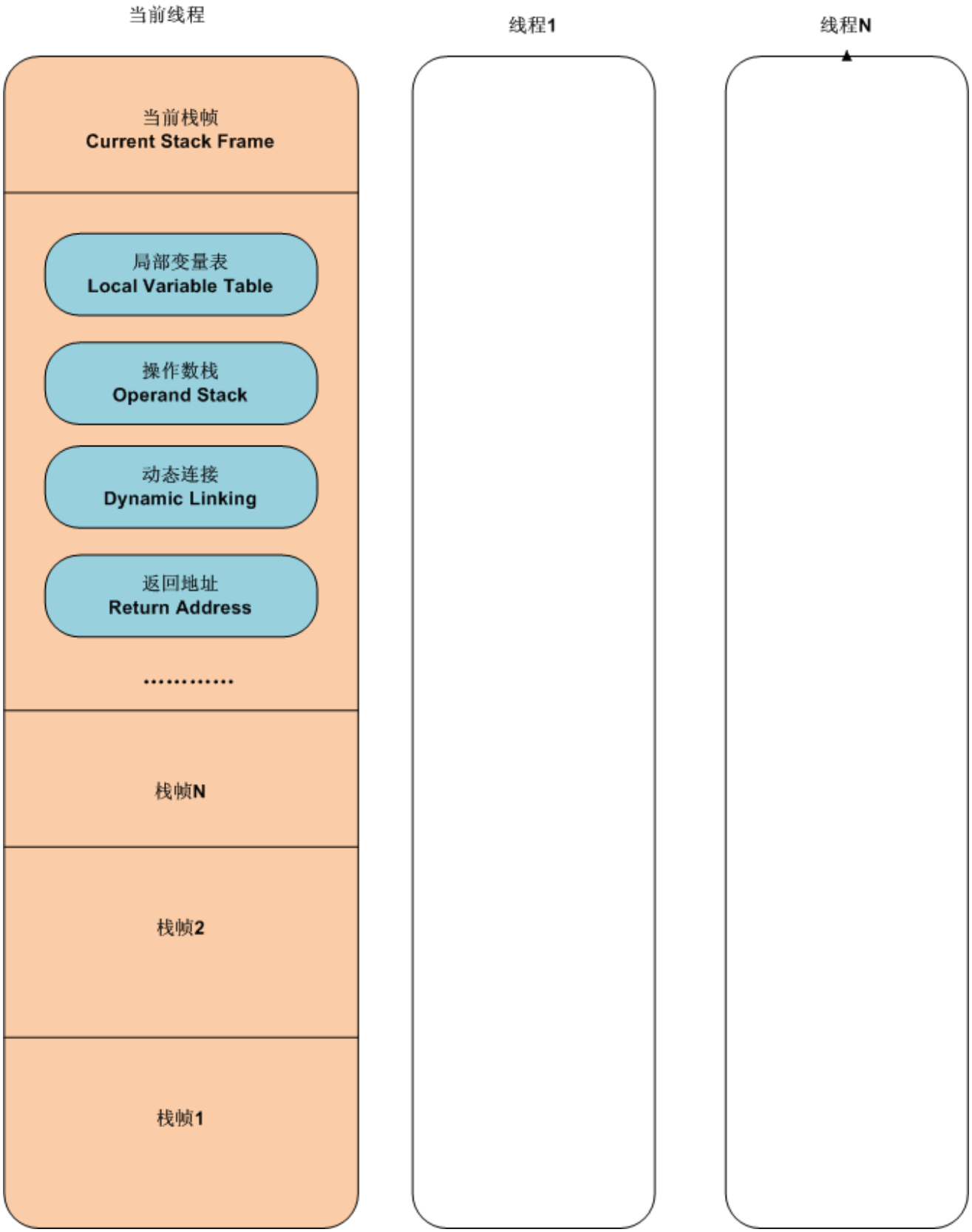
栈帧

定义

- 栈帧（stack frame）是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈的栈元素。栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。
- 每一个方法从调用开始到执行完成的过程，就对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。
- 对于执行引擎来说，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。


当前线程
线程1
线程N
当前栈帧
Current Stack Frame
局部变量表
Local Variable Table
操作数栈
Operand Stack
动态连接
Dynamic Linking
返回地址
Return Address
.....
栈帧N
栈帧2
栈帧1

组成

（1）局部变量表

- 局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在Java程序被编译成Class文件时，就在方法的Code属性的max_locals数据项中确定了该方法所需要分配的最大局部变量表的容量。
- 局部变量表的容量以变量槽（Slot）为最小单位，32位虚拟机中一个Slot可以存放一个32位以内的数据类型（boolean、byte、char、short、int、float、reference和returnAddress八种）。
  - reference类型虚拟机规范没有明确说明它的长度，但一般来说，虚拟机实现至少都应当能从此引用中直接或者间接地查找到对象在Java堆中的起始地址索引和方法区中的对象类型数据。
  - returnAddress类型是为字节码指令jsr、jsr_w和ret服务的，它指向了一条字节码指令的地址。
- 虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果是实例方法（非static），那么局部变量表的第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中通过this访问。
- Slot是可以重用的，当Slot中的变量超出了作用域，那么下一次分配Slot的时候，将会覆盖原来的数据。Slot对对象的引用会影响GC（要是被引用，将不会被回收）。
- 系统不会为局部变量赋予初始值（实例变量和类变量都会被赋予初始值）。也就是说不存在类变量那样的准备阶

（2）操作数栈

- Java虚拟机的解释执行引擎被称为"基于栈的执行引擎"，其中所指的栈就是指 - 操作数栈。操作数栈也常被称为操作栈。和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。但是和前者不同的是，它不是通过索引来访问，而是通过标准的栈操作—压栈和出栈—来访问的。比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。
- 虚拟机在操作数栈中存储数据的方式和在局部变量区中是一样的：如int、long、float、double、reference和returnType的存储。对于byte、short以及char类型的值在压入到操作数栈之前，也会被转换为int。
- 虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。比如，iadd指令就要从操作数栈中弹出两个整数，执行加法运算，其结果又压回到操作数栈中，看看下面的示例，它演示了虚拟机是如何把两个int类型的局部变量相加，再把结果保存到第三个局部变量的：

```
begin
iload_0    // push the int in local variable 0 onto the stack
iload_1    // push the int in local variable 1 onto the stack
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
end
```

- 在这个字节码序列里，前两个指令iload_0和iload_1将存储在局部变量中索引为0和1的整数压入操作数栈中，其后iadd指令从操作数栈中弹出那两个整数相加，再将结果压入操作数栈。第四条指令istore_2则从操作数栈中弹出结果，并把它存储到局部变量区索引为2的位置。下图详细表述了这个过程中局部变量和操作数栈的状态变化，图中没有使用的局部变量区和操作数栈区域以空白表示。

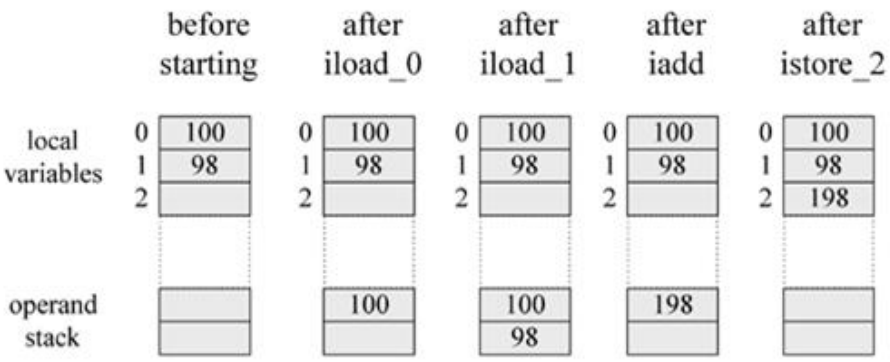


Figure 5-10. Adding two local variables.

（3）例子

Slot 对GC的影响

- 为了节省栈帧空间，局部变量表中的 Slot 是可以重用的。当离开了某些变量的作用域之后，这些变量对应的 Slot 就可以交给其他变量使用。这种机制有时候会影响垃圾回收行为。

```
//代码1
public static void main(String[] args){
    {
    byte[] placeholder = new byte[64*1024*1024];
    }
    System.gc();
}
//需加上 -verbose:gc参数
/*
运行结果：
    [GC 602K->378K(15872K), 0.0603803 secs]
    [Full GC 378K->378K(15872K), 0.0323107 secs]
    [Full GC 66093K->65914K(81476K), 0.0074124 secs]
*/
```

```
//代码2
public static void main(String[] args){
    {
    byte[] placeholder = new byte[64*1024*1024];
```

```
    }

    int a = 0;
    System.gc();
}
/*
    运行结果:
    [GC 602K->378K(15872K), 0.0018270 secs]
    [Full GC 378K->378K(15872K), 0.0057871 secs]
    [Full GC 66093K->378K(81476K), 0.0054067 secs]
*/
```

- 分析：通过结果可以知道，代码一和代码二内的 placeholder 变量在 System.gc() 执行后理应被回收了，可是结果却是只有代码二被回收了，这是为什么呢?
    - 这是因为代码一中 placeholder 虽然离开了作用域，但之后没有任何局部变量对其进行读写，也就是说其占用的 Slot 没有被复用，也就是说 placeholder 占用的内存仍然有引用指向它，因而它没有被回收。而代码二中的变量a由于复用了 placeholder 的 Slot ，导致 placeholder 引用被删除，因此占用的内存空间被回收。
- 《Practical Java》一书中把"不使用的对象应手动赋值为 null "作为一条推荐的**编码规则**，这并不是一个完全没有意义的操作。但是不应该对 赋 null 值有过多的依赖，主要有两点原因:
- 从编码的角度来讲，用恰当的变量作用域来控制变量的回收才是最优雅的解决方法。
- 从执行角度将，使用赋值 null 的操作优化内存回收是建立在对字节码执行引擎概念模型基础上的，但是概念模型与实际执行模型可能完全不同。在使用解释器执行时，通常离概念模型还比较接近，但是一旦经过JIT 编译为本地代码才是虚拟机执行代码的主要方式，赋 null 值在JIT编译优化之后会被完全消除，这时候赋 null 值是完全没有意义的。（其实，上面代码1在 JIT 编译为本地代码之后，gc() 之后内存也会被自动回收）

代码体会

```
//结果是什么呢?
public class Test
{
    public static void main(String[] args)
    {
        int  i = 1 ;
        i = i++ ;
        int j = i++ ;
        int k = i + ++i * i++ ;
        System.out.println("i = " + i ) ;
        System.out.println("j = " + j) ;
        System.out.println("k = " + k) ;
    }
}
```