链滴

# JVM_02 运行时数据区 1-[程序计数器 + 虚拟机栈 + 本地方法栈]

作者：wlgzs-sjl

原文链接：https://ld246.com/article/1601907385328

来源网站：链滴

<h2 id="内存与线程">内存与线程</h2>
<h3 id="1-内存">1、内存</h3>
内存是非常重要的系统资源，是硬盘和 cpu 的中间仓库及桥梁，承载着操作系统和应用程序的时运行。JVM 内存布局规定了 JAVA 在运行过程中内存申请、分配、管理的策略，保证了 JVM 的高稳定运行。不同的 jvm 对于内存的划分方式和管理机制存在着部分差异（对于otspot 主要指方法区）

（图源阿里）JDK8 的元数据区 +JIT 编译产物 就是 JDK8 以前的方法区
<h3 id="2-分区介绍">2、分区介绍</h3>
java 虚拟机定义了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动创建，随着虚拟机退出而销毁。另外一些则是与线程一一对应的，这些与线程对应的数据区域会随着程开始和结束而创建和销毁。
如图，灰色的区域为单独线程私有的，红色的为多个线程共享的，即
<ul>
<li>每个线程：独立包括程序计数器、栈、本地栈</li>
<li>线程间共享：堆、堆外内存（方法区、永久代或元空间、代码缓存）</li>
</ul>

一般来说，jvm 优化 95% 是优化堆区，5% 优化的是方法区
<h3 id="3-线程">3、线程</h3>
<ul>
<li>线程是一个程序里的运行单元，JVM 允许一个程序有多个线程并行的执行；</li>
<li>在 HotSpot JVM，每个线程都与操作系统的本地线程直接映射。
<ul>
<li>当一个 java 线程准备好执行以后，此时一个操作系统的本地线程也同时创建。java 线程执行终后。本地线程也会回收。</li>
</ul>
</li>
<li>操作系统负责所有线程的安排调度到任何一个可用的 CPU 上。一旦本地线程初始化成功，它就调用 java 线程中的 run（）方法。</li>
</ul>
<h3 id="4-JVM系统线程">4、JVM 系统线程</h3>
<ul>
<li>如果你使用 jconsole 或者任何一个调试工具，都能看到在后台有许多线程在运行。这些后台线不包括调用 main 方法的 main 线程以及所有这个 main 线程自己创建的线程；</li>
<li>这些主要的后台系统线程在 HotSpot JVM 里主要是以下几个：
<ul>
<li>虚拟机线程：这种线程的操作是需要 JVM 达到安全点才会出现。这些操作须在不同的线程中发生的原因是他们都需要 JVM 达到安全点，这样堆才不会变化。这种线程的执行型包括“stop-the-world”的垃圾收集，线程栈收集，线程挂起以及偏向锁撤销</li>
<li>周期任务线程：这种线程是时间周期事件的体现（比如中断），他们一般于周期性操作的调度执行。</li>
<li>GC 线程：这种线程对在 JVM 里不同种类的垃圾收集行为提供了支持</li>
<li>编译线程。：这种线程在运行时会将字节码编译成本地代码</li>
<li>信号调度线程：这种线程接收信号并发送给 JVM,在它内部通过调用适当的法进行处理。</li>
</ul>
</li>
</ul>

</ul>
<h2 id="程序计数器-PC寄存器-">程序计数器（PC 寄存器）</h2>
JVM 中的程序计数寄存器（Program Counter Register）中，Register 的命名源于 CPU 的寄器，寄存器存储指令相关的现场信息。CPU 只有把数据装载到寄存器才能够运行。JVM 中 PC 寄存器是对物理 PC 寄存器的一种抽象模拟。
<h3 id="1-作用">1、作用</h3>
PC 寄存器是用来存储指向下一条指令的地址，也即将将要执行的指令代码。由执行引擎读取下条指令。
<ul>
<li>它是一块很小的内存空间，几乎可以忽略不计。也是运行速度最快的存储区域。</li>
<li>在 jvm 规范中，每个线程都有它自己的程序计数器，是线程私有的，生命期与线程的生命周期保持一致。</li>
<li>任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序数器会存储当前线程正在执行的 java 方法的 JVM 指令地址；或者，如果实在执行 native 方法，则未指定值（undefined）。</li>
<li>它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个数器来完成。</li>
<li>字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令。</li>
<li>它是唯一一个在 java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。</li>
</ul>
<h4 id="1-1-代码示例">1.1 代码示例</h4>
利用 javap -v xxx.class 反编译字节码文件，查看指令等信息

<h4 id="1-2-面试常问">1.2 面试常问</h4>
1.使用 PC 寄存器存储字节码指令地址有什么用呢？/ 为什么使用 PC 寄存器记录当前程的执行地址呢？
<blockquote>
因为 CPU 需要不停的切换各个线程，这时候切换回来以后，就得知道接着从哪开始继续执行<br>
JVM 的字节码解释器就需要通过改变 PC 寄存器的值来明确下一条应该执行什么样的字节码指令
</blockquote>
2.PC 寄存器为什么会设定为线程私有？
我们都知道所谓的多线程在一个特定的时间段内指回执行其中某一个线程的方法，CPU 会不停做任务切换，这样必然会导致经常中断或恢复，如何保证分毫无差呢？
<blockquote>
为了能够准确地记录各个线程正在执行的当前字节码指令地址，最好的办法自然是为每一个线程分配一个 PC 寄存器。每个线程在创建后，都会产生自己的程序计数器和栈帧，程序计数器在各个线之间互不影响。
</blockquote>
由于 CPU 时间片轮限制，众多线程在并发执行过程中，任何一个确定的时刻，一个处理器或者核处理器中的一个内核，只会执行某个线程中的一条指令。
<h4 id="1-3-时间片">1.3 时间片</h4>
CPU 时间片即 CPU 分配各各个程序的时间，每个线程被分配一个时间段。称作它的时间片。
在宏观上：我们可以同时打开多个应用程序，每个程序并行不悖，同时运行。<br>
但在微观上：由于只有一个 CPU，一次只能处理程序要求的一部分，如何处理公平，一种方法就是引时间片，每个程序轮流执行。
并行与并发：<br>
并行：同一时间多个线程同时执行；<br>
并发：一个核快速切换多个线程，让它们依次执行，看起来像并行，实际上是并发。
<h2 id="虚拟机栈">虚拟机栈</h2>
<h3 id="1-概述">1、概述</h3>

<h4 id="1-1-背景">1.1 背景</h4>
由于跨平台性的设计，java 的指令都是根据栈来设计的。不同平台 CPU 架构不同，所以不能设为基于寄存器的。<br>
优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更多指令。
<h4 id="1-2-内存中的堆与栈">1.2 内存中的堆与栈</h4>
<ul>
<li>
栈是运行时的单位，而堆是存储的单位
即：栈解决程序的运行问题，即程序如何执行，或者说如何处理数据。堆解决的是数据存储的问，即数据怎么放、放在哪儿。
</li>
<li>
一般来讲，对象主要都是放在堆空间的，是运行时数据区比较大的一块。
</li>
<li>
栈空间存放基本数据类型的局部变量，以及引用数据类型的对象的引用。
</li>
</ul>
<h4 id="1-3-虚拟机栈是什么">1.3 虚拟机栈是什么</h4>
<ul>
<li>java 虚拟机栈（Java Virtual Machine Stack），早期也叫 Java 栈。<br>
每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应这个一次的 java 方法调用。它是线程私有的</li>
<li>生命周期和线程是一致的</li>
<li>作用：主管 java 程序的运行，它保存方法的局部变量（8 种基本数据类型对象的引用地址）、部分结果，并参与方法的调用和返回。
<ul>
<li>局部变量：相对于成员变量（或属性）</li>
<li>基本数据变量： 相对于引用类型变量（类，数组，接口）</li>
</ul>
</li>
</ul>
<h4 id="1-4-栈的特点">1.4 栈的特点</h4>
<ul>
<li>栈是一种快速有效的分配存储方式，访问速度仅次于 PC 寄存器（程序计数器）</li>
<li>JVM 直接对 java 栈的操作只有两个
<ul>
<li>每个方法执行，伴随着进栈（入栈，压栈）</li>
<li>执行结束后的出栈工作</li>
</ul>
</li>
<li>对于栈来说不存在垃圾回收问题</li>
</ul>
<h4 id="1-5-栈中可能出现的异常">1.5 栈中可能出现的异常</h4>
java 虚拟机规范允许 Java 栈的大小是动态的或者是固定不变的
<ul>
<li>如果采用固定大小的 Java 虚拟机栈，那每一个线程的 java 虚拟机栈容量可以在线程创建的时候立选定。如果线程请求分配的栈容量超过 java 虚拟机栈允许的最大容量，java 虚拟机将会抛出一 StackOverFlowError 异常。</li>
<li>如果 java 虚拟机栈可以动态拓展，并且在尝试拓展的时候无法申请到足够的内存，或者在创建的线程时没有足够的内存去创建对应的虚拟机栈，那 java 虚拟机将会抛出一个 OutOfMemoryError 异常**。**</li>
</ul>

<h4 id="1-6-设置栈的内存大小">1.6 设置栈的内存大小</h4>
我们可以使用参数-Xss 选项来设置线程的最大栈空间，栈的大小直接决定了函数调用的最大可深度。 （IDEA 设置方法：Run-EditConfigurations-VM options 填入指定栈的大小-Xss256k）</p

<h3 id="2-栈的存储结构和运行原理">2、栈的存储结构和运行原理</h3>
<ul>
<li>每个线程都有自己的栈，栈中的数据都是以**栈帧(Stack Frame)**的格式存在。</li>
<li>在这个线程上正在执行的每个方法都对应各自的一个栈帧。</li>
<li>栈帧是一个内存区块，是一个数据集，维系着方法执行过程中的各种数据信息。</li>
<li>JVM 直接对 java 栈的操作只有两个，就是对栈帧的压栈和出栈，遵循“先进后出/后进先出”的则。</li>
<li>在一条活动线程中，一个时间点上，只会有一个活动的栈帧。即只有当前正在执行的方法的栈帧栈顶栈帧）是有效的，这个栈帧被称为当前栈帧(Current Frame),与当前栈帧应的方法就是当前方法（Current Frame），定义这个方法的类就是前类（Current Class）。</li>
<li>执行引擎运行的所有字节码指令只针对当前栈帧进行操作。</li>
<li>如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，放在栈的顶端，成为新的当前帧。</li>
<li>不同线程中所包含的栈帧是不允许相互引用的，即不可能在一个栈帧中引用另外一个线程的栈帧<li>
<li>如果当前方法调用了其他方法，方法返回之际，当前栈帧会传回此方法的执行结果给前一个栈帧接着，虚拟机会丢弃当前栈帧，使得前一个栈帧重新成为当前栈帧</li>
<li>Java 方法有两种返回函数的方式，一种是正常的函数返回，使用 return 指令；另外种是抛出异常。不管使用哪种方式，都会导致栈帧被弹出。</li>
</ul>

<h4 id="2-1-栈帧的内部结构">2.1 栈帧的内部结构</h4>
每个栈帧中存储着：
<ul>
<li>局部变量表（Local Variables）</li>
<li>操作数栈（Operand Stack）(或表达式栈)</li>
<li>动态链接（Dynamic Linking）(或执行运行时常量池的方法引用)</li>
<li>方法返回地址（Return Adress）（或方法正常退出或者异常退出的定义）</li>
<li>一些附加信息。</li>
</ul>

<h3 id="3-局部变量表-Local-Variables-">3、局部变量表（Local Variables）</h3>
<h4 id="3-1-概述">3.1 概述</h4>
<ul>
<li>局部变量表也被称之为局部变量数组或本地变量表</li>
<li>定义为一个数字数组，主要用于存储方法参数和定义在方法体内的局部变量，这些数据类型包括各类基本数据类型、对象引用（reference），以及 returnAddress 类型。</li>
<li>由于局部变量表是建立在线程的栈上，是线程的私有数据，因此不存在数据安全问题</trong>。</li>
<li>局部变量表所需的容量大小是在编译期确定下来的,并保存在方法的 Code 性的 maximum local variables 数据项中。在方法运行期间是不会改变局部变量表的大小的</li>
<li>方法嵌套调用的次数由栈的大小决定。一般来说，栈越大，方法嵌套调用次数越多。对一个函数而言，他的参数和局部变量越多，使得局部变量表膨胀，它的栈帧就越大，以满方法调用所需传递的信息增大的需求。进而函数调用就会占用更多的栈空间，导致其嵌套调用次数就减少。</li>

<li>**局部变量表中的变量只在当前方法调用中有效。**在方法执行时，虚拟机通过使用局部变量表成参数值到参数变量列表的传递过程。当方法调用结束后，随着方法栈帧的销毁，局部变表也会随之销毁。</li>
</ul>
利用 javap 命令对字节码文件进行解析查看局部变量表，如图：

也可以在 IDEA 上安装 jclasslib byte viewcoder 插件查看字节码信息,以 main()方法为例

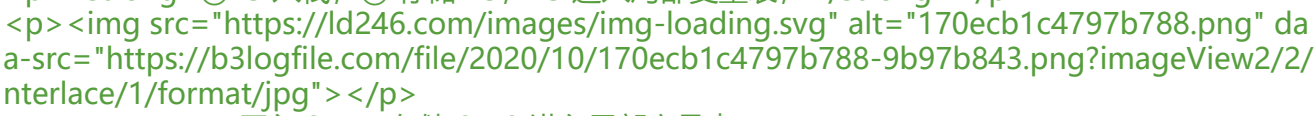
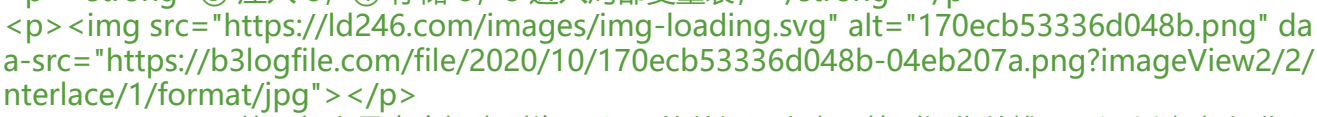
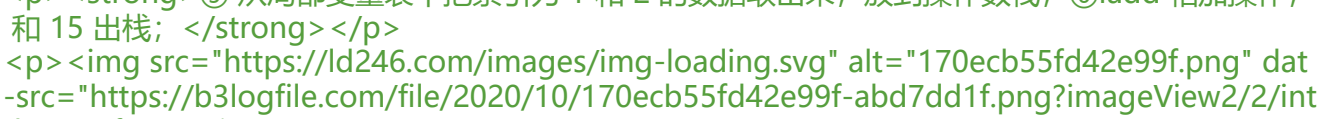
<h4 id="3-2-变量槽slot的理解与演示">3.2 变量槽 slot 的理解与演示</h4>
<ul>
<li>参数值的存放总是在局部变量数组的 index0 开始，到数组长度-1 的索引结束</li>
<li>局部变量表，最基本的存储单元是 Slot(变量槽)</li>
<li>局部变量表中存放编译期可知的各种基本数据类型（8 种），引用类型（reference），returnAddress 类型的变量。</li>
<li>在局部变量表里，32 位以内的类型只占用一个 slot（包括 returnAddress 类型），64 位的类型（long 和 double）占用两个 slot。
<ul>
<li>byte、short、char、float 在存储前被转换为 int，boolean 也被转换为 int，0 表示 false，非 0 表示 true；</li>
<li>long 和 double 则占据两个 slot。</li>
</ul>
</li>
</ul>

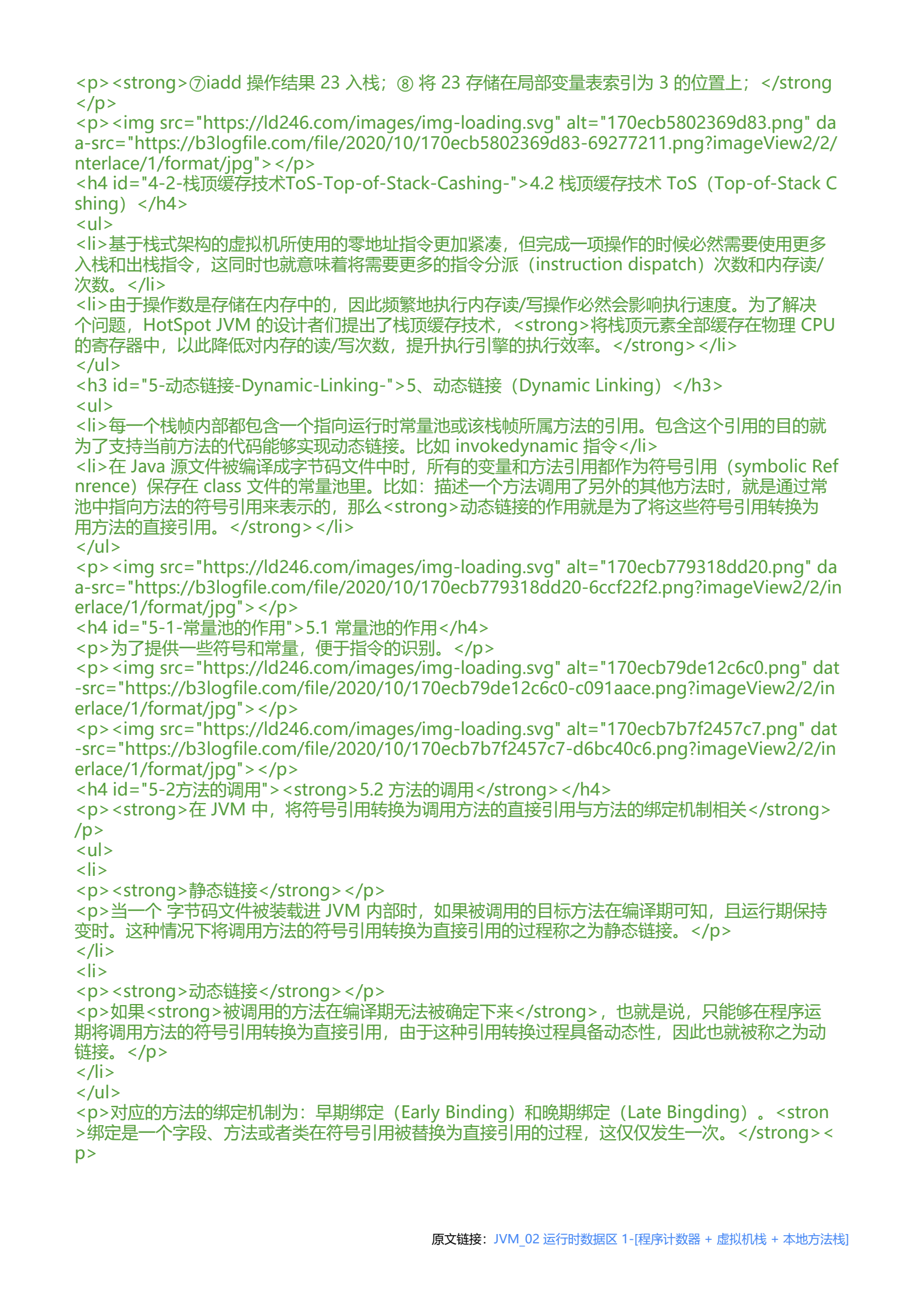
<ul>
<li>JVM 会为局部变量表中的每一个 slot 都分配一个访问索引，通过这个索引即可成功访问到局部量表中指定的局部变量值。</li>
<li>当一个实例方法被调用的时候，它的方法参数和方法体内部定义的局部变量将会按照序被复制到局部变量表中的每一个 slot 上。</li>
<li>如果需要访问局部变量表中一个 64bit 的局部变量值时，只需要使用签一个索引即可（比如：访问 long 或者 double 类型变量）</li>
<li>如果当前帧是由构造方法或者实例方法创建的，那么该对象引用 this 将会存放在 index 为 0 的 slot 处,其余的参数按照参数表顺序排列。</li>
</ul>
<pre><code class="highlight-chroma">public class LocalVariablesTest {

    private int coun
 = 1;
    //静态方法不能用this
    public static vo

d testStatic(){
        //编译错误，为this变量不存在与当前方法的局部变量表中！！！
        System.out.println(this.count);
    }
}
</code></pre>
<h4 id="3-3-slot的重复利用">3.3 slot 的重复利用</h4>
栈帧中的局部变量表中的槽位是可以重复利用的，如果一个局部变量过了作用域，那么在其作用域之后申明的新的局部变量就很有可能会复用过期局部变量的槽位，从而达到节省资源的目的。
<pre><code class="highlight-chroma">private void test2() {
        int a = 0;
        {
            int b = 0;
            b = a+1;
        }
        //变量c使用前以及经销毁的变量b占据的slot位置
        int c = a+1;
    }
</code></pre>
<h4 id="3-4-静态变量与局部变量的对比及小结">3.4 静态变量与局部变量的对比及小结</h4>
变量的分类：
<ul>
<li>按照数据类型分：
<ul>
<li>① 基本数据类型;</li>
<li>② 引用数据类型；</li>
</ul>
</li>
<li>按照在类中声明的位置分：
<ul>
<li>① 成员变量：在使用前，都经历过默认初始化赋值。
<ul>
<li>static 修饰：类变量：类加载 linking 的准备阶段给类变量默认赋值——> 初始化阶段给类变显式赋值即静态代码块赋值；</li>
<li>不被 static 修饰：实例变量：随着对象的创建，会在堆空间分配实例变量空间，并进行默认赋值<li>
</ul>
</li>
<li>② 局部变量：在使用前，必须要进行显式赋值的！否则，编译不通过。</li>
</ul>
</li>
</ul>
补充：
<ul>
<li>在栈帧中，与性能调优关系最为密切的部分就是局部变量表。在方法执行时，虚拟机使用局部变表完成方法的传递。</li>
<li>局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或间接引的对象都不会被回收。</li>
</ul>

<h3 id="4-操作数栈-Operand-Stack-">4、操作数栈（Operand Stack）</h3>
栈：可以使用数组或者链表来实现
<ul>
<li>每一个独立的栈帧中除了包含局部变量表以外，还包含一个后进先出的操作数栈，也可以称为表式栈。</li>
<li>操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或提取数据，即入（push）或出栈（pop）
<ul>
<li>某些字节码指令将值压入操作数栈，其余的字节码指令将操作数取出栈，使用他们后再把结果压栈。（如字节码指令 bipush 操作）</li>
<li>比如：执行复制、交换、求和等操作</li>
</ul>
</li>
</ul>

<h4 id="4-1-概述">4.1 概述</h4>
<ul>
<li>操作数栈，主要用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储间。</li>
<li>操作数栈就是 jvm 执行引擎的一个工作区，当一个方法开始执行的时候，一个新的栈帧也会随之创建出来，这个方法的操作数栈是空的</li>
<li>每一个操作数栈都会拥有一个明确的栈深度用于存储数值，其所需的最大深度在编译期就定义好，保存在方法的 code 属性中，为 max_stack 的值。</li>
<li>栈中的任何一个元素都是可以任意的 java 数据类型
<ul>
<li>32bit 的类型占用一个栈单位深度</li>
<li>64bit 的类型占用两个栈深度单位</li>
</ul>
</li>
<li>操作数栈并非采用访问索引的方式来进行数据访问的，而是只能通过标砖入栈(push)和出栈(pop)操作来完成一次数据访问</li>
<li>如果被调用的方法带有返回值的话，其返回值将会被压入当前栈帧的操作数栈中，并更新 PC 寄存器中下一条需要执行的字节码指令。</li>
<li>操作数栈中的元素的数据类型必须与字节码指令的序列严格匹配，这由编译器在编译期间进行验，同时在类加载过程中的类验证阶段的数据流分析阶段要再次验证。</li>
<li>另外，我们说 Java 虚拟机的解释引擎是基于栈的执行引擎,其中的栈指的是操作数栈。</li>
</ul>
结合下面的图来看一下一个方法（栈帧）的执行过程：
①15 入栈；② 存储 15，15 进入局部变量表；

③ 压入 8；④ 存储 8，8 进入局部变量表；

⑤ 从局部变量表中把索引为 1 和 2 的数据取出来，放到操作数栈；⑥iadd 相加操作，和 15 出栈；


⑦iadd 操作结果 23 入栈；⑧ 将 23 存储在局部变量表索引为 3 的位置上；

<h4 id="4-2-栈顶缓存技术ToS-Top-of-Stack-Cashing-">4.2 栈顶缓存技术 ToS（Top-of-Stack Cashing）</h4>
<ul>
<li>基于栈式架构的虚拟机所使用的零地址指令更加紧凑，但完成一项操作的时候必然需要使用更多入栈和出栈指令，这同时也就意味着将需要更多的指令分派（instruction dispatch）次数和内存读/次数。</li>
<li>由于操作数是存储在内存中的，因此频繁地执行内存读/写操作必然会影响执行速度。为了解决个问题，HotSpot JVM 的设计者们提出了栈顶缓存技术，将栈顶元素全部缓存在物理 CPU 的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率。</li>
</ul>
<h3 id="5-动态链接-Dynamic-Linking-">5、动态链接（Dynamic Linking）</h3>
<ul>
<li>每一个栈帧内部都包含一个指向运行时常量池或该栈帧所属方法的引用。包含这个引用的目的就为了支持当前方法的代码能够实现动态链接。比如 invokedynamic 指令</li>
<li>在 Java 源文件被编译成字节码文件中时，所有的变量和方法引用都作为符号引用（symbolic Refnrence）保存在 class 文件的常量池里。比如：描述一个方法调用了另外的其他方法时，就是通过常池中指向方法的符号引用来表示的，那么动态链接的作用就是为了将这些符号引用转换为用方法的直接引用。</li>
</ul>

<h4 id="5-1-常量池的作用">5.1 常量池的作用</h4>
为了提供一些符号和常量，便于指令的识别。

<h4 id="5-2方法的调用">5.2 方法的调用</h4>
在 JVM 中，将符号引用转换为调用方法的直接引用与方法的绑定机制相关
<ul>
<li>
静态链接
当一个 字节码文件被装载进 JVM 内部时，如果被调用的目标方法在编译期可知，且运行期保持变时。这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接。
</li>
<li>
动态链接
如果被调用的方法在编译期无法被确定下来，也就是说，只能够在程序运期将调用方法的符号引用转换为直接引用，由于这种引用转换过程具备动态性，因此也就被称之为动链接。
</li>
</ul>
对应的方法的绑定机制为：早期绑定（Early Binding）和晚期绑定（Late Bingding）。绑定是一个字段、方法或者类在符号引用被替换为直接引用的过程，这仅仅发生一次。

<ul>
<li>
早期绑定
早期绑定就是指被调用的目标方法如果在编译期可知，且运行期保持不变时，即可将这个方法与所属的类型进行绑定，这样一来，由于明确了被调用的目标方法究竟是哪一个，此也就可以使用静态链接的方式将符号引用转换为直接引用。
</li>
<li>
晚期绑定
如果被调用的方法在编译期无法被确定下来，只能够在程序运行期根据实际的类型绑相关的方法，这种绑定方式也就被称之为晚期绑定。
</li>
</ul>
随着高级语言的横空出世，类似于 java 一样的基于面向对象的编程语言如今越来越多，尽管这编程语言在语法风格上存在一定的差别，但是它们彼此之间始终保持着一个共性，那就是都支持封装集成和多态等面向对象特性，既然这一类的编程语言具备多态特性，那么自然也就具备早期绑定和晚绑定两种绑定方式。
Java 中任何一个普通的方法其实都具备虚函数的特征，它们相当于 C++ 语言中的虚函数（C++ 中则需要使用关键字 virtual 来显式定义）。如果在 Java 程序中不希望某个方法拥有虚函数的特征时则可以使用关键字 final 来标记这个方法。
<h4 id="5-3-虚方法和非虚方法">5.3 虚方法和非虚方法</h4>
非虚方法
<ul>
<li>如果方法在编译器就确定了具体的调用版本，这个版本在运行时是不可变的。这样的方法称为非方法</li>
<li>静态方法、私有方法、final 方法、实例构造器、父类方法都是非虚方法</l>
<li>其他方法称为虚方法</li>
</ul>
<code>子类对象的多态性使用前提：①类的继承关系②方法的重写</code>
<h5 id="虚拟机中提供了以下几条方法调用指令-">虚拟机中提供了以下几条方法调用指令：</h5>
普通调用指令：
1.invokestatic：调用静态方法，解析阶段确定唯一方法版本；
2.invokespecial:调用方法、私有及父类方法，解析阶段确定唯一方法版本；
3.invokevirtual 调用所有虚方法；
4.invokeinterface：调用接口方法；
动态调用指令：
5.invokedynamic：动态解析出需要调用的方法，然后执行 .
前四条指令固化在虚拟机内部，方法的调用执行不可人为干预，而 invokedynamic 指令则支持用户确定方法版本。其中 invokestatic 指令和 invokespecial 指令调用的方法称为非虚方，其余的（final 修饰的除外）称为虚方法。
<h5 id="关于invokedynamic指令">关于 invokedynamic 指令</h5>
<ul>
<li>JVM 字节码指令集一直比较稳定，一直到 java7 才增加了一个 invokedynamic 指令，这是 <strng>Java 为了实现【动态类型语言】支持而做的一种改进。</li>
<li>但是在 java7 中并没有提供直接生成 invokedynamic 指令的方法，需要借助 ASM 这种底层字码工具来产生 invokedynamic 指令.直到 Java8 的 Lambda 表达式的出现，invokedynamc 指令的生成，在 java 中才有了直接生成方式。</li>
<li>Java7 中增加的动态语言类型支持的本质是对 java 虚拟机规范的修改，而不是对 java 语言规则修改，这一块相对来讲比较复杂，增加了虚拟机中的方法调用，最直接的受益者就是运行在 java 平的动态语言的编译器。</li>
</ul>
<h5 id="动态类型语言和静态类型语言">动态类型语言和静态类型语言</h5>
<ul>

<li>动态类型语言和静态类型语言两者的区别就在于对类型的检查是在编译期还是在运行期，满足前就是静态类型语言，反之则是动态类型语言。</li>
<li>直白来说静态语言是判断变量自身的类型信息；动态类型语言是判断变量值的类型信，变量没有类型信息，变量值才有类型信息,这是动态语言的一个重要特征</li>
<li>Java 是静态类型语言（尽管 lambda 表达式为其增加了动态特性），js，python 是动态类型语言</li>
</ul>
<h4 id="5-4-方法重写的本质">5.4 方法重写的本质</h4>
<ul>
<li>找到操作数栈的第一个元素所执行的对象的实际类型，记作 C。</li>
<li>如果在类型 C 中找到与常量中的描述符合简单名称都相符的方法，则进行访问权限校验，如果通则返回这个方法的直接引用，查找过程结束；如果不通过，则返回 java.lang.IllegalAccessError 异。</li>
<li>否则，按照继承关系从下往上依次对 c 的各个父类进行第二步的搜索和验证过程。</li>
<li>如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError 异常。</li>
</ul>
IllegalAccessError 介绍<br>
程序视图访问或修改一个属性或调用一个方法，这个属性或方法，你没有权限访问。一般的，这个会起编译器异常。这个错误如果发生在运行时，就说明一个类发生了不兼容的改变。
<h4 id="5-5-虚方法表">5.5 虚方法表</h4>
<ul>
<li>在面向对象编程中，会很频繁期使用到动态分派，如果在每次动态分派的过程中都要重新在类的法元数据中搜索合适的目标的话就可能影响到执行效率。因此，为了提高性能，vm 采用在类的方法区建立一个虚方法表（virtual method table）（非虚方法不会出现在中）来实现。使用索引表来代替查找。</li>
<li>每个类中都有一个虚方法表，表中存放着各个方法的实际入口。</li>
<li>虚方法表会在类加载的链接阶段被创建并开始初始化，类的变量初始值准备完成之后，jvm 会把类的方法表也初始化完毕。</li>
</ul>

<h3 id="6-方法返回地址-Return-Address-">6、方法返回地址（Return Address）</h3>
<ul>
<li>存放调用该方法的 PC 寄存器的值。</li>
<li>一个方法的结束，有两种方式：
<ul>
<li>正常执行完成</li>
<li>出现未处理的异常，非正常退出</li>
</ul>
</li>
<li>无论通过哪种方式退出，在方法退出后都返回到该方法被调用的位置。方法正常退出时，**调用的 pc 计数器的值作为返回地址，即调用该方法的指令的下一条指令的地址。**而通过异常退出时，回地址是要通过异常表来确定，栈帧中一般不会保存这部分信息。</li>
<li>本质上，方法的退出就是当前栈帧出栈的过程。此时，需要恢复上层方法的局部变量表、操作数、将返回值也如调用者栈帧的操作数栈、设置 PC 寄存器值等，让调用者方法继续执行下去。</li>
<li>正常完成出口和异常完成出口的区别在于：通过异常完成出口退出的不会给他的上层用者产生任何的返回值。</li>
</ul>
当一个方法开始执行后，只有两种方式可以退出这个方法：
<ol>
<li>执行引擎遇到任意一个方法返回的字节码指令（return），会有返回值传递给上层的方法调用者简称正常完成出口；</li>
</ol>

<ul>
<li>一个方法在正常调用完成之后究竟需要使用哪一个返回指令还需要根据方法返回值的实际数据类而定</li>
<li>在字节码指令中，返回指令包含 ireturn（当返回值是 boolena、byte、char、short 和 int 类时使用）、lreturn、freturn、dreturn 以及 areturn，另外还有一个 return 指令供声明为 void 的方、实例初始化方法、类和接口的初始化方法使用</li>
</ul>
<ol start="2">
<li>在方法执行的过程中遇到了异常（Exception），并且这个异常没有在方法内进行处理，也就是要在本方法的异常表中没有搜素到匹配的异常处理器，就会导致方法退出，简称异常完成口</li>
</ol>
方法执行过程中抛出异常时的异常处理，存储在一个异常处理表，方便在发生异常的时候找到处异常的代码。

<h3 id="7-一些附加信息">7、一些附加信息</h3>
栈帧中还允许携带与 java 虚拟机实现相关的一些附加信息。例如，对程序调试提供支持的信息（很多资料都忽略了附加信息）
<h3 id="8-虚拟机栈的5道面试题">8 虚拟机栈的 5 道面试题</h3>
1.举例栈溢出的情况？（StackOverflowError）
<ul>
<li>递归调用等，通过-Xss 设置栈的大小。</li>
</ul>
2.调整栈的大小，就能保证不出现溢出么？
<ul>
<li>不能 如递归无限次数肯定会溢出，调整栈大小只能保证溢出的时间晚一些</li>
</ul>
3.分配的栈内存越大越好么？
<ul>
<li>不是 会挤占其他线程的空间。</li>
</ul>
4.垃圾回收是否会涉及到虚拟机栈？
<ul>
<li>不会</li>
</ul>

5.方法中定义的局部变量是否线程安全？
<ul>
<li>要具体情况具体分析。</li>
<li>如果只有一个线程可以操作此数据，则必是线程安全的。如果有多个线程操作此数据，则此数据共享数据。如果不考虑同步机制的话，会存在线程安全问题。</li>
</ul>
<h2 id="本地方法栈">本地方法栈</h2>
<ul>
<li>Java 虚拟机栈用于管理 Java 方法的调用，而本地方法栈用于管理本地方法的调用</li>
<li>本地方法栈，也是线程私有的。</li>
<li>允许被实现成固定或者是可动态拓展的内存大小。（在内存溢出方面是相同的）
<ul>

<li>如果线程请求分配的栈容量超过本地方法栈允许的最大容量，Java 虚拟机将会抛出一个 StackOverFlowError 异常。</li>
<li>如果本地方法栈可以动态扩展，并且在尝试扩展的时候无法申请到足够的内存，或者在创建新的程时没有足够的内存去创建对应的本地方法栈，那么 java 虚拟机将会抛出一个 OutOfMemoryError 异常。</li>
</ul>
</li>
<li>本地方法是使用 C 语言实现的</li>
<li>它的具体做法是 Native Method Stack 中登记 native 方法，在 Execution Engine 执行时加载地方法库。</li>
<li>当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世。它和虚拟机拥有同样的权限
<ul>
<li>本地方法可以通过本地方法接口来访问虚拟机内部的运行时数据区</li>
<li>它甚至可以直接使用本地处理器中的寄存器</li>
<li>直接从本地内存的堆中分配任意数量的内存</li>
</ul>
</li>
<li>并不是所有的 JVM 都支持本地方法。因为 Java 虚拟机规范并没有明确要求本地方法的使用语言、具体实现方式、数据结构等。如果 JVM 产品不打算支持 native 方法，也可无需实现本地方法栈。</li>
<li>在 hotSpot JVM 中，直接将本地方法栈和虚拟机栈合二为一。</li>
</ul>