目录

JVM 知识点总结

这个总结顺序按照《深入理解 Java 虚拟机》这本书来做。知识点内容也是精简该书（有些是自己理解）方便复习。

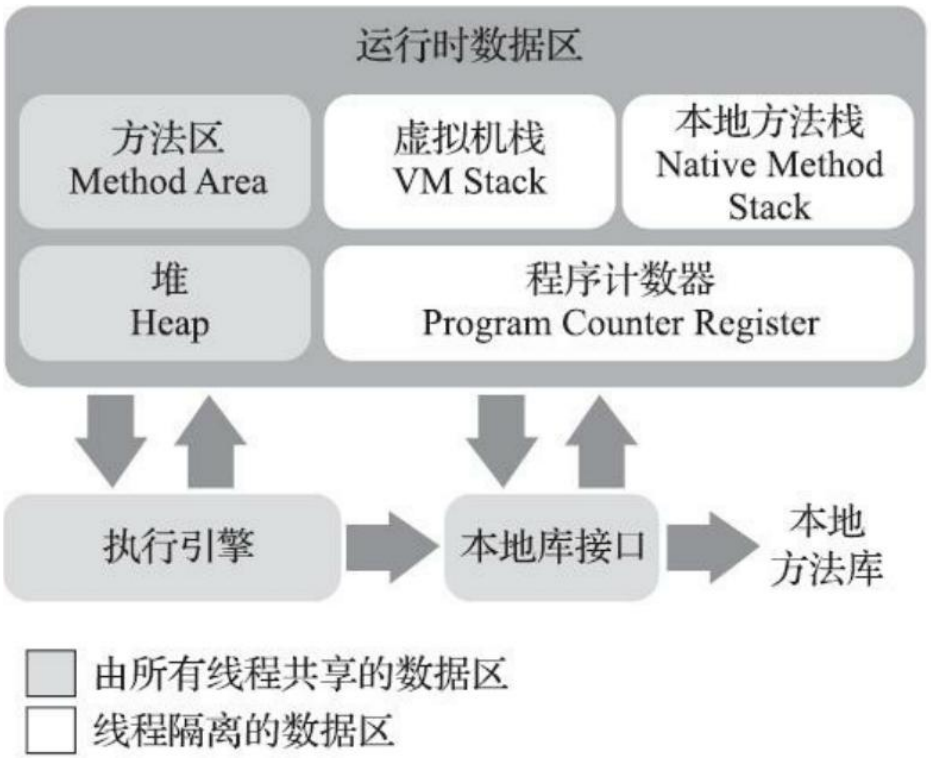
一、Java 内存区域与内存溢出异常

1.1 概述

对于 Java 来说，程序员并不需要关心内存的回收问题，因为虚拟机有自动内存管理机制，这个机制使得 Java 语言并不需要和 C/C++一样，调用 free/delete 代码，不容易出现内存泄露或者内存溢出的问题，但是，一旦出现了内存泄露或者内存回收的问题，修复工作也是极其复杂的。本章讲解各个区域出现内存溢出异常的原因，以及内存区域划分等。

1.2 运行时数据区

Java 虚拟机在执行 Java 程序的过程中，会把所管理内存划分为若干个不同的区域。根据《Java 虚拟机规范》，如图所示:



1.2.1 程序计数器

程序计数器在整个内存中占较小的空间，它可以看做是当前线程所执行字节码的行号指示器。在虚拟机的概念模型里，字节码解释器会改变程序计数器的值来获取下一条字节码指令。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要依靠这个计数器。

程序计数器是线程私有的。Java 虚拟机的多线程是通过线程轮流切换、分配处理器执行时间的方式来实现的。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一条程序计数器。

如果线程执行的是一个本地方法，那计数器记录值则为空。

该区域是唯一一个没有规定任何 OutOfMemoryError 的情况的区域。

1.2.2 虚拟机栈

虚拟机栈也是线程私有的，生命周期和线程相同。

每个方法被执行时，Java 虚拟机都会同步创建一个栈帧（Stack Frame）用来存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法被调用直至执行完毕的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。对执行引擎而言，在活动线程中，只有位于栈顶的栈帧才是生效的，这部分被称为“当前栈帧”，与这个栈帧相关联的方法称为“当前方法”。

在《Java 虚拟机规范》中，对虚拟机栈定义了两类异常情况：

如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出 StackOverflowError 异常。

如果 Java 虚拟机的栈容量可动态扩展，当栈扩展时无法申请到足够的内存，将会抛出 OutOfMemoryError 异常。

1.2.3 本地方法栈

本地方法栈和虚拟机栈所发挥作用是类似的，区别只是本地方法栈是为本地方法服务的。

与虚拟机栈一样，本地方法栈也会有 StackOverflowError 异常和 OutOfMemoryError 异常。

1.2.4 Java 堆

对于 Java 应用程序来说，堆是虚拟机所管理的内存中最大的一部分。Java 堆是所有线程共享的一块内存区域，此内存区域的唯一目的就是存放对象实例，Java 几乎所有的对象实例都在这里分配内存。《Java 虚拟机规范》中对堆的描述：所有的对象实例以及数组都应当在堆上分配。

Java 堆是垃圾收集器管理的内存区域，因此也叫 GC 堆。从回收内存角度看，由于现代垃圾收集器大多数都是基于分代收集理论设计的，因此 Java 堆中会出现像“新生代”，“老年代”等名词，但这些仅仅是一部分垃圾收集器的共同特性或设计风格，而不是固有的内存布局。

根据《Java 虚拟机规范》的规定，Java 堆可以处于物理上不连续的内存空间中，但逻辑上应当被视为连续的，就像磁盘空间去存储文件一样。但对于大对象（数组对象等），很可能要求连续存储。

Java 堆内存大小是可以扩展的（通过参数-Xms -Xmx 来设定）。当 Java 堆没有内存完成实例分配，并且堆也无法再扩展时，Java 虚拟机将抛出 OutOfMemoryError 异常。

1.2.5 方法区

方法区和堆一样，都是各个线程共享的内存区域，用来存储已被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据。

在《Java 虚拟机规范》中，方法区被描述为堆的一个逻辑部分，但它有个别名“非堆”，目的是与 Java 堆区分开。由此可见，方法区是《Java 虚拟机规范》定义的一个概念，而不是一个具体实现。JDK8 以前，方法区由永久代来实现，JDK8 以后，由元空间来实现。

对于永久代和元空间的设计。在 JDK8 以前，当时的 HotSpot 虚拟机设计团队选择把收集器的设计扩展到方法区，这样 HotSpot 垃圾收集器就可以管理这部分内存，省去专门为方法区编写内存管理代码的工作。但后来发现，这确实不是一个好主意，这种设计使得 Java 应用更容易遇到内存溢出问题。

JDK6 开始，逐步采用本地内存（这样只要不涉及到进程可用内存上限，比如 32 位系统中是 4GB，就不会出问题）来实现方法区的计划。

JDK7 开始，把原本放在永久代的字符串常量池，静态变量等移出。

JDK8 开始，完全废弃永久代，改用在本地内存实现的元空间来代替，把 JDK1.7 中永久代剩余的内容放入元空间。

《Java 虚拟机规范》中，对方法区的约束非常宽松，除了可以和堆一样不需要连续的内存，可以固定大小或者可扩展，甚至可以选择不实现垃圾回收。因此相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进了方法区就如永久代的名字一样永远存在的。对于方法区的回收，有时确实也有必要。

《Java 虚拟机规范》规定：如果方法区无法满足新的内存分配需求时，将抛出 OutOfMemoryError 异常。

根据总结：
字符串常量池和静态变量这部分在移到堆中，所以方法区在 JDK7 的时候是由永久代和堆共同实现的。

在 JDK8 的时候，运行时常量池（类信息）放在元空间，字符串常量池和静态变量这部分在还是在堆中，所以方法区是由元空间和堆共同实现的。

1.2.6 运行时常量池

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池表，用于存放在编译期时生成的各种字面量与符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。

除了保存 Class 文件中描述的符号引用外，还会把由符号引用翻译出来的直接引用也存储在运行时常量池中（符号引用和直接引用的概念见第三章）。

String 类的 intern()方法。由于运行时常量池相对于 Class 文件池的一个重要特征是具备动态性，即 Java 语言并不要求常量一定要编译期才能产生，也就是说，运行期也可以将新的常量放入常量池。

intern()方法会先检查常量池中是否有该字符串的对象，如果存在就返回，如果不存在就往池中创建该对象并返回。

由于运行时常量池是方法区的一部分，当常量池无法再申请到内存时，会抛出 OutOfMemoryError 异常。

1.2.7 直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是《Java 虚拟机规范》中定义的内存区域。

在 JDK1.4 中新加入了 NIO 类，引入了一种基于通道（channel）与缓冲区（buffer）的 I/O 方式，它可以使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

在配置虚拟机参数时，如果忽略掉直接内存，就会使各个内存区域的总和大于物理内存限制，从而动态扩展时出现 OutOfMemoryError 异常。

1.3 HotSpot 虚拟机对象探秘

1.3.1 对象的创建

在实际操作中，创建对象的过程仅仅是一个 new 关键字。

当 Java 虚拟机遇到一条字节码 new 指令时，首先将会去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析、初始化过。如果没有，必须执行相应的类加载过程。（后面会谈类加载过程）。

当类加载检查通过后，虚拟机会给新生对象分配内存，对象所需的内存大小在类加载完成后便可以完全确定。在堆中划分一块确定大小的内存块用来给对象分配。

内存分配好后，虚拟机必须将分配到的内存空间（不包括对象头）都初始化为零值。这个操作是为了保证实例对象中的字段在 Java 代码中不赋初始值就可以直接使用，使程序能访问到这些字段的数据类型所对应的零值。

接下来，Java 虚拟机还要对对象进行必要的设置，会将引用哪个类的实例。如何才能找到类的元数据信息，对象的 GC 分代年龄等信息存放在对象头中。

最后，执行<init>()方法（构造方法）初始化对象。

根据《深入理解 Java 虚拟机》总结（该总结完整，借鉴网友）：

常量池中定位类的符号引用

↓

检查符号引用所代表的类是否已被加载，解析和初始化过 →

↓ ↓

分配内存（类加载完成后，内存需求确定） ← 加载

↓

根据 java 堆是否规整（GC 方法）选择分配方法

↙ ↘

指针碰撞 空闲列表

↓

分配内存的并发保证（指针更新的原子性）

↙ ↘

CAS+失败重试 按照线程划分在不同的空间中进行 TLAB -XX:+UseTLAB -XX:-UseTLAB

↓

内存空间初始化为 0 值，保证对象的实例字段可以不赋初值就可以使用。

↓

设置对象头信息（Object Header）：引用指针，元数据，hash 值，GC 分代年龄，锁相关

↓

执行对象<init>方法

重要部分的总结，当 new Obj 时：

1. 去查找当前类的 class 对象，若未加载，加载当前对象的 class 文件
2. 虚拟机在堆中为对象分配内存
3. 分配内存的初始值设为 0 值（不包括对象头）
4. 设置对象头信息
5. 执行构造方法赋值

1.3.2 对象的内存布局

在 HotSpot 虚拟机中，对象在堆内存中的存储布局可以划分为三个部分：对象头，实例数据和对齐填充。

对象头部分包括两类信息。一部分为用于存储对象自身的运行时数据，如 hash 码、GC 分代年龄、锁状态标志、线程持有锁、偏向线程 ID、偏向时间戳等。这部分在 32 位和 64 位虚拟机中分别为 32bit 和 64bit。这部分官方称为“Mark Word”。

锁状态	32bit				
	25bit		4bit	1bit	2bit
	23bit	2bit		偏向模式	标志位
未锁定	对象哈希码		分代年龄	0	01
轻量级锁定	指向调用栈中锁记录的指针				00
重量级锁定（锁膨胀）	指向重量级锁的指针				10
GC 标记	空				11
可偏向	线程 ID	Epoch	分代年龄	1	01

另一部分为类型指针，即对象指向它的类型元数据的指针，Java 虚拟机通过这个指针来确定该对象是哪个类的实例。

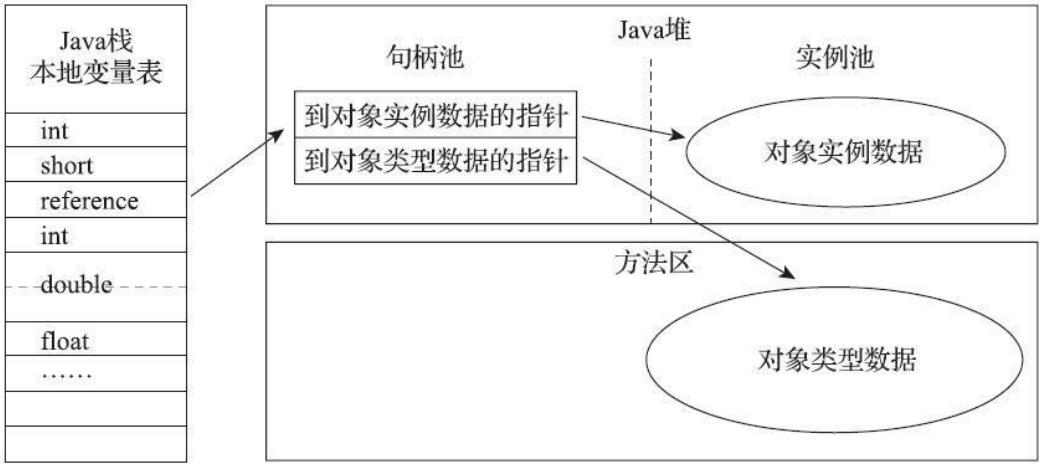
实例数据部分是对象真正存储的有效信息。即我们在程序代码里定义的各种类型的字段内容，无论是从父类继承下来的，还是子类中定义的字段都必须记录起来。

最后一部分是对其填充，这不是必然存在的，也没有特别的含义，只是让对象的大小都必须是 8 字节的整数倍。

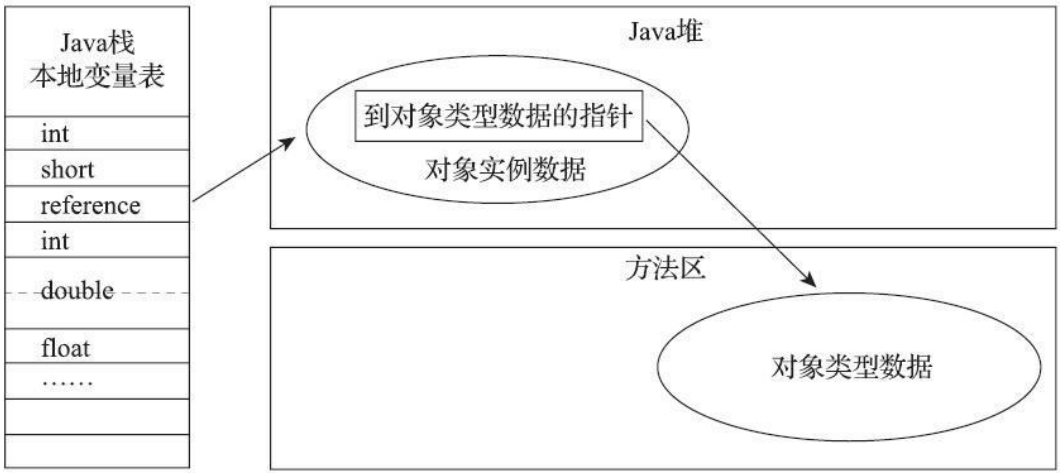
1.3.3 对象的访问定位

创建对象自然是为了后续使用该对象，我们的 Java 程序会通过栈上的 reference 数据来操作堆上的具体对象。现在主流的访问方式有使用句柄和直接指针两种。

使用句柄访问，Java 堆中将可能划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自具体的地址信息。



使用直接指针访问，Java 堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference 中存储的直接就是对象地址。



这两种访问方式各有优势，使用句柄来访问的最大好处是 reference 中存储的是稳定句柄地址，在对象被移动时，只会改变句柄中的实例数据指针，而 reference 本身不需要被修改。

使用直接指针的最大好处就是速度更快，它节省了一次指针定位的时间开销。在 HotSpot 虚拟机中，主要使用直接指针的方式来进行对象访问。

二、垃圾收起与内存分配策略

2.1 概述

而 Java 堆和方法区这两个区域则有着很显著的不确定性：一个接口的多个实现类需要的内存可能会不一样，一个方法所执行的不同条件分支所需要的内存也可能不一样，只有处于运行期间，我们才能知道程序究竟会创建哪些对象，创建多少个对象，这部分内存的分配和回收是动态的。垃圾收集器所关注的正是这部分内存该如何管理。

2.2 对象已死？

垃圾回收器在对堆进行回收之前，首先要确定哪些对象还活着，哪些已经死去。

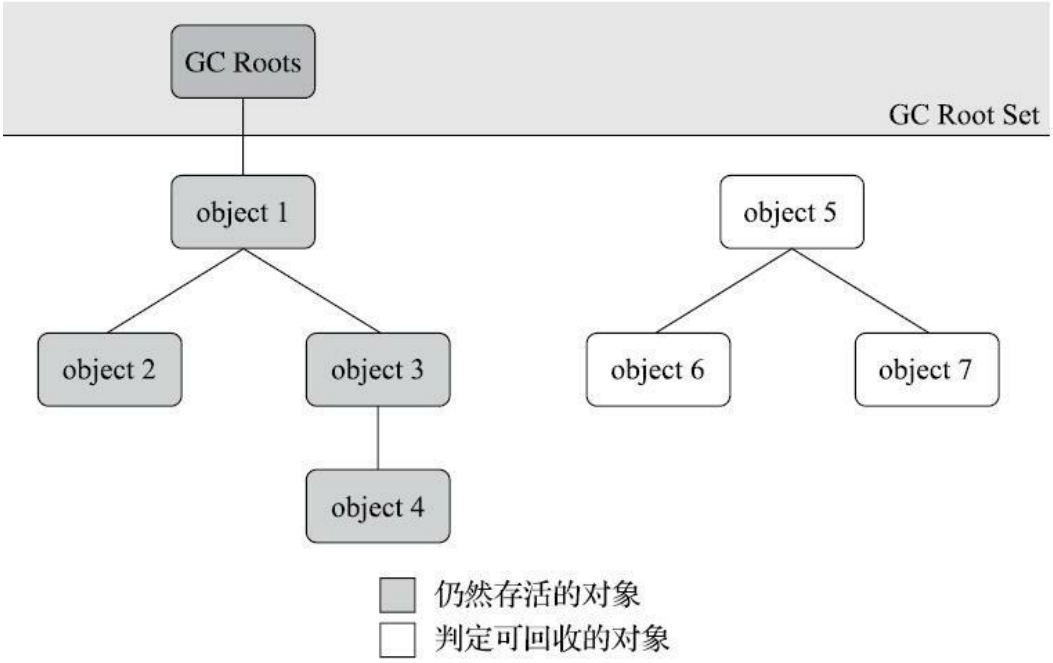
2.2.1 引用计数算法

引用计数法的基本思想：在对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加一；当引用失效时，计数器值就减一；任何时刻计数器为0的对象就是不可能再被使用的。

在Java中并没有使用引用计数算法，主要原因是，这个算法需要配合大量额外处理才能保证正确地工作，譬如单纯的引用计数就很难解决对象之间相互循环引用的问题。

2.2.2 可达性分析算法

在 Java 中，使用的可达性分析算法来判定对象是否存活。这个算法的基本思想：通过一系列称为“GC Roots”的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程所走过的路径称为“引用链”，如果某个对象到 GC Roots 间没有任何引用链相连，或者用图论的话说，从 GC Roots 到这个对象不可达时，则证明这个对象是不可能再被使用的。



在 Java 体系中，固定可作为 GC Roots 的对象：

（1）在虚拟机栈（栈帧中的本地变量表）中引用的对象，譬如各个线程被调用的方法堆栈中使用到的参数、局部变量、临时变量等（这里应该是各个线程的当前栈帧中使用到的，比如当前方法结束，方法内形参，局部变量等回收）。

（2）在方法区中类静态属性引用的对象，譬如 Java 类的引用类型静态变量。

（3）在方法区中常量引用的对象，譬如字符串常量池里的引用。

（4）在本地方法栈中 JNI（本地方法接口）引用的对象。

（5）Java 虚拟机内部的引用，如基本数据类型对应的 Class 对象，一些常驻的异常对象（如 OutOfMemoryError）等，还有系统类加载器。

（6）所有被同步锁（synchronized）持有的对象。

（7）反映 Java 虚拟机内部情况的 JMXBean、JVMT1 中注册的回调，本地代码缓存等。

除了这些固定的集合外，根据用户所选用的垃圾收集器以及当前回收的内存区域不同，还可以有其他对象临时性加入。

2.2.3 再谈引用

引用计数法判断对象的引用数量，可达性分析算法判断对象是否引用链可达，这两种方法都判断的“引用”。

在 JDK1.2 版本后，Java 对引用分为：强引用（Strongly Reference），软引用（Soft Reference），弱引用（Weak Reference），虚引用（Phantom Reference）。

（1）强引用是指程序代码中普遍存在的引用赋值。类似“Object o = new Object”这种引用关系。无论什么情况下，只要强引用关系还存在，垃圾收集器就永远不会回收掉被引用的对象。回收：“o = null”。

（2）软引用是用来描述一些还有用，但非必须的对象。只被软引用关联着的对象，在系统将要发生内存溢出异常前，会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存，才回抛出内存溢出异常。该引用常常用来实现缓存技术，内存不足就自动回收。

（3）弱引用也是用来描述那些非必须对象，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生为止。当垃圾收集器开始工作，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。

（4）虚引用也称为“幽灵引用”或者“幻影引用”，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的只是为了能在这个对象被收集器回收时收到一个系统通知。

2.2.4 生存还是死亡

在可达性分析算法判定中，被判定为不可达的对象，要经过两次标记过程才能判定死亡：如果对象在进行可达性分析后发现没有与 GC Roots 相连接的引用链，那它将会被第一次标记，随后进行一次筛选，筛选的条件是此对象是否有必要执行 finalize()方法。假如对象没有覆盖 finalize()方法，或者 finalize()方法已经被虚拟机调用过，那么虚拟机将这两种情况都视为没有必要执行。

如果该对象被判定为确有必要执行 finalize()方法，那么该对象将会被放置在一个名为 F-Queue 的队列之中，并在稍后由一条由虚拟机自动建立的、低调度优先级的 Finalizer 线程去执行它们的 finalize()方法。

这里虽然触发执行 finalize()方法，但并不承诺会等待它们运行结束。原因：如果某个对象 finalize()方法执行缓慢，或者更极端地发生了死循环，将很可能导致 F-Queue 队列中的其他对象永久出于等待，甚至导致整个内存回收子系统崩溃。

finalize()方法是对象逃脱死亡命运的最后一次机会。稍后，收集器将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，比如把自己（this）赋值给某个类变量或者对象的成员变量。这样第二次标记时它将被移出“即将回收”的集合；如果这个时候对象还没有逃脱，那基本上它就真的要被回收了。

finalize()方法是 Object 类提供的一个方法，在 GC 准备释放对象所占用的空间之前，它将先调用该方法。

finaliz()方法中一般用于释放非 Java 资源（如打开的文件资源，数据库连接等），或是调用非 Java 方法（native）时分配的内存。本书中，作者建议彻底忘掉这个方法，关闭资源等使用 try，catch 能做的更好。

避免使用的原因：对象在变得不可达开始，到 finalize()方法被执行，所花费的时间是任意长的，也就是上面说的，不会承诺等待 finalize()方法运行结束的原因。

网上找了个例子，利用 finalize()方法最多只会被调用一次，延长对象的生命周期（没有用书上的例子是因为太长了，懒得截图...网上这个例子和书上一样）：

```
class User{

    public static User user = null;

    @Override
    protected void finalize() throws Throwable {
        System.out.println("User-->finalize()");
        user = this;
    }

}

public class FinalizerTest {
    public static void main(String[] args) throws InterruptedException {
        User user = new User();
        user = null;
        System.gc();
        Thread.sleep(1000);

        user = User.user;
        System.out.println(user != null);//true

        user = null;
        System.gc();
        Thread.sleep(1000);
        System.out.println(user != null);//false
    }
}
```

可以看出，this 只调用了一次，即 finalize()方法只调用了一次。

2.2.5 回收方法区

《Java 虚拟机规范》中提到，可以不要求虚拟机在方法区中实现垃圾收集，因为方法区进行垃圾回收的性价比比较低。

方法区中垃圾收集主要回收两部分内容：废弃的常量和不再使用的类型。

回收废弃常量与回收 Java 堆中的对象非常类似。假如一个字符串“java”曾经进入常量池 中，但是当前系统又没有任何一个字符串对象的值是“java”，换句话说，已经没有任何字符串对象引用常量池中的“java”常量，且虚拟机中也没有其他地方引用这个字面量。如果在这时发生内存回收，而且垃圾收集器判断确有必要的话，这个“java”常量就将会被系统清理出常量池。常量池中其他类（接口）、方法、字段的符号引用也与此类似。

回收不再使用的类型需要先判断该类是否不再被使用。需要满足下面三个条件：

（1）该类所有的实例都已经被回收，也就是 Java 堆中不存在该类及其任何派生子类的实例。

（2）加载该类的类加载器已经被回收，这个条件除非是经过精心设计的可替换类加载器的场景，如 OSGi、JSP 的重加载等，否则通常是很难达成的。

（3）该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

而满足上面三个条件仅仅是被允许回收，而不是必须回收。

2.3 垃圾收集算法

本节重在介绍分代收集理论和几种算法思想及其发展过程。
Java 设计使用的“追踪式垃圾收集（TracingGC）”。

2.3.1 分代收集理论

现在商业虚拟机的垃圾收集器，大多数都遵循的分代收集的理论进行设计。该理论建立在两个分代假说之上：

1）弱分代假说：绝大多数对象是朝生夕灭。
2）强分代假说：熬过越多次的垃圾收集过程的对象就越难以消亡。

多款垃圾收集器根据这两个假说定义了一致的收集原则：收集器应该将 Java 划分出不同的区域，然后将回收对象依据其年龄（年龄即对象熬过垃圾收集过程的次数）分配到不同的区域之中存储。

在 Java 堆划分出不同的区域之后，垃圾收集器才可以每次只回收其中某一个或者某些部分的区域，才有了“Minor GC”“Major GC”“Full GC”这样回收类型的划分；才能针对不同区域安排与里面存储对象存亡特征相匹配的垃圾回收算法——因而发展出了标记复制，标记清除，标记整理等针对性的垃圾回收算法。

虽然按照这两个分代假说理论，设计者会把 Java 堆划分为新生代和老年代两个区域，但是也存在一个明显的困难：对象不是孤立的，对象之间会存在跨代引用。假如对新生代进行一次 Minor GC，但是新生代的对象是完全有可能被老年代引用，为了找出该区域中活着的对象，不得不在固定的 GC Roots 之外，再额外遍历整个老年代中所有对象来确保可达性分析结果的正确性。反过来也是一样。遍历整个老年代的对象虽然可行，但是增加了内存回收的性能负担。

为了解决这个问题，引入了第三条经验法则：
3）跨代引用假说：跨代引用相对于同代引用来说仅占极少数。
存在互相引用关系的两个对象，是应该倾向于同时生存或者同时消亡的。如果某个新生代的对象存在跨代引用，由于老年代对象难以消亡，该引用会使得新生代对象在收集时同样得以存活，在年龄增长后进入老年代中，这种跨代引用也随即被消除了。

依据这条假说，我们不需要因为少量的跨代引用而去扫描整个老年代，也不必浪费空间专门去记录每一个对象是否存在及存在哪些跨代引用，只需要在新生代上建立一个全局的数据结构（该结构被称为记忆集，Remembered Set），这个结构把老年代划分为若干个小块，标识出老年代的哪一块内存会存在跨代引用。此后当发生 Minor GC 时，只有包含了跨代引用的小块内存里的对象才会被加入到 GC Roots 进行扫描。虽然这种方法需要在对象改变引用关系时维护记录数据的正确性，会增加一些运行时的开销，但是对于扫描整个老年代来说是划算的。

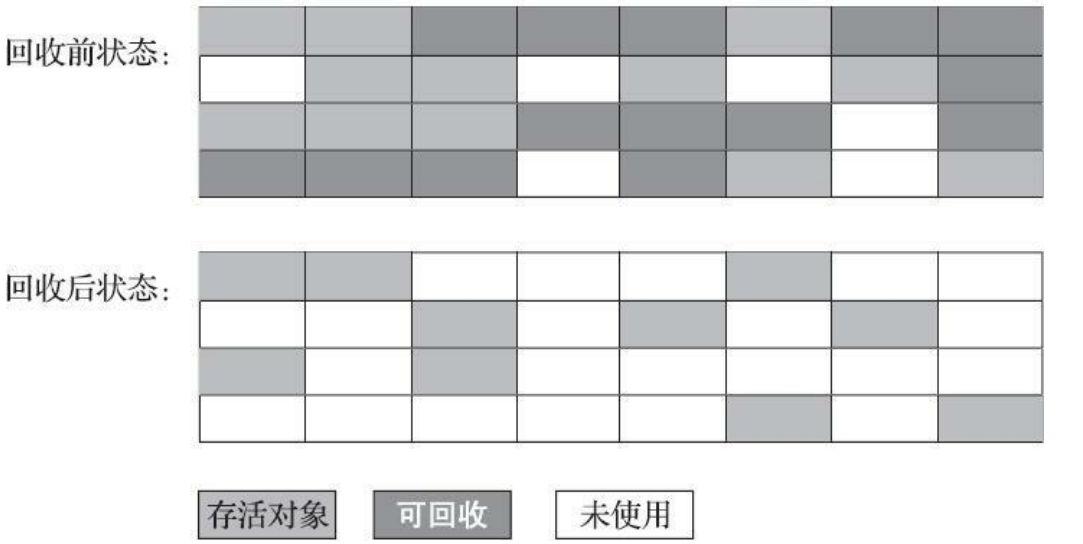
2.3.2 标记-清除算法

标记-清除算法分为标记和清除两个阶段；首先标记出所有需要回收的对象，在标记完成后，统一回收掉所有被标记的对象，也可以反过来标记存活的对象，然后回收未被标记的对象。后续的算法大多数以标记-清除算法为基础，对其缺点进行改进。

标记清除算法的缺点主要有两个：

（1）执行效率不稳定。如果 Java 堆中包含大量对象，而且其中大部分是需要被回收的，这时必须进行大量标记和清除的动作，导致标志和清除两个阶段都随着对象数量的增长而效率变低。

（2）内存空间的碎片化问题。标记清除后会产生大量不连续的内存碎片，空间碎片太多可能会导致当以后在程序运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

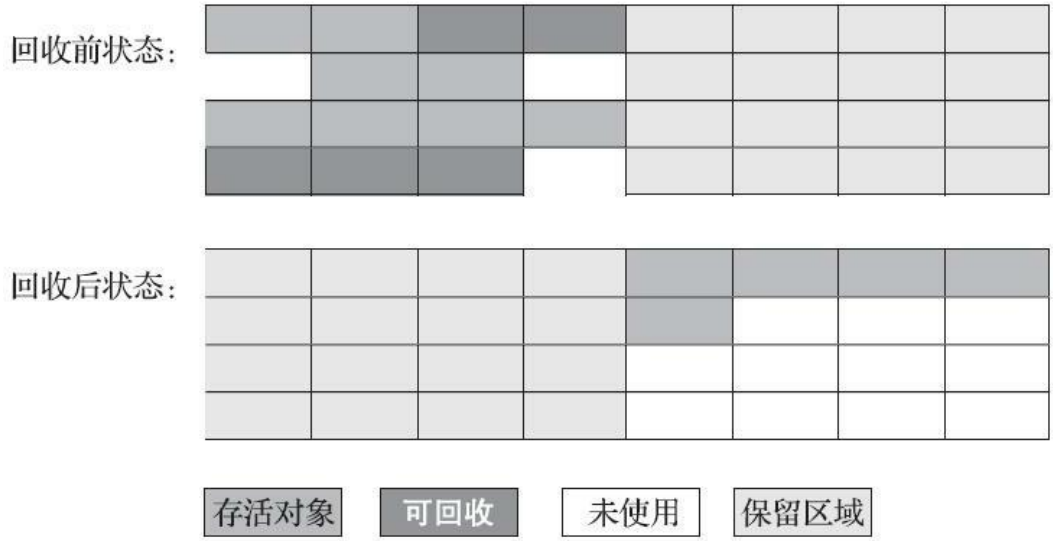


2.3.3 标记-复制算法

标记复制算法通常被简称为复制算法，该算法是为了解决标记-清除算法面对大量可回收对象时执行效率低的问题。

根据该算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。如果内存中多数对象都是存活的，这种算法将会产生大量的内存间复制开销，但如果多数对象都是可回收状态，算法需要复制的就是少量的存活对象。每次只针对整个半区进行内存回收，分配内存时也就不用考虑空间碎片的复杂情况。

半区缺点也显而易见：空间浪费的多了一点。



IBM 公司专门研究过——新生代中的对象 98%熬不过第一轮收集。因此不需要按照 1:1 的比例去划分新生代的内存空间。

现在的 HotSpot 虚拟机的 Serial、ParNew 等新生代收集器均采用了 Appel 式回收布局：把新生代划分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次分配内存只使用 Eden 和其中一块 Survivor。当发生垃圾收集时，将 Eden 和 Survivor 中仍然存活的对象一次性复制到另外一块 Survivor 空间上，然后直接清理掉 Eden 和已使用过的那块 Survivor 空间。

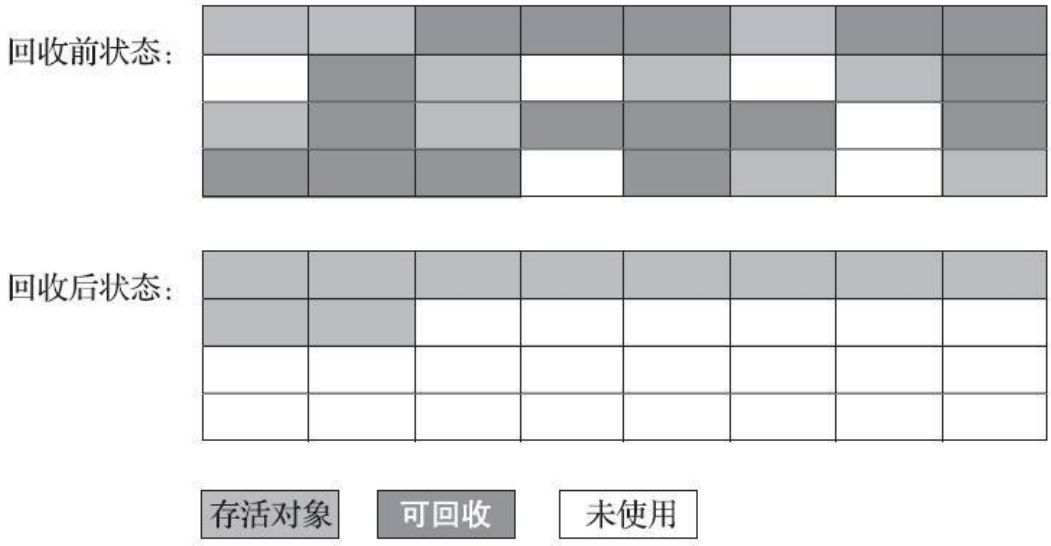
HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1，即每次新生代中可用内存空间为整个新生代容量的 90%，只有一个 Survivor 空间是被浪费的。

当 Survivor 空间不足以容纳一次 Minor GC 之后存活的对象时，就需要依赖其他区域（通常是老年代）进行分配担保。

2.3.4 标记-整理算法

标记-复制算法在对象存活率较高时就要进行较多的复制操作，效率将会降低。另外，还需要有其他空间进行担保，以应对极端情况，所以在老年代中一般不使用该算法。

针对老年代存亡特征，提出了标记-整理算法。标记过程和标记-清除算法一样，但后续过程则是将所有存活的对象都向内存空间一端移动，然后直接清理掉边界以外的内存。



标记-整理算法的缺点也显而易见：如果移动存活对象，尤其是在老年代这种每次回收都有大量对象存活区域，移动存活对象并更新所有引用这些对象的地方将会是一种极为负重的操作，而且这种对象移动操作必须全程暂停用户应用程序才能进行。

2.4 HotSpot 的算法细节实现

前面说的都是原理上的实现，而算法实现则是保证虚拟机高效运行的关键。
根节点枚举以及后面的安全点，安全区域都是对于找寻 GC Roots 做的优化；
记忆集与卡表是为了缩减 GC Roots 扫描范围；
写屏障是为了解决卡表维护问题；
并发的可达性分析则是在并发条件下，如何遍历对象图进行标记。

2.4.1 根节点枚举

我们以可达性分析算法中从 GC Roots 集合找引用链这个操作作为介绍虚拟机高效实现的第一个例子。固定可作为 GC Roots 的节点主要在全局性的引用（例如常量或类静态属性）与执行上下文（例如栈帧中的本地变量表）中，尽管目标明确，但查找过程要做到高效并非一件容易的事情。

迄今为止，所有收集器在根节点枚举这一步骤时都是必须暂停用户线程的，因此毫无疑问根节点枚举与之前提及的整理内存碎片一样会面临相似的“Stop The World”的困扰。现在可达性分析算法耗时最长的查找引用链的过程已经可以做到与用户线程一起并发（具体见 2.4.6 节），但根节点枚举始终还是必须在一个能保障一致性的快照中才得以进行——这里“一致性”的意思是整个枚举期间执行子系统看起来就像被冻结在某个时间点上，不会出现分析过程中，根节点集合的对象引用关系还在不断变化的情况，若这点不能满足的话，分析结果准确性也就无法保证。这是导致垃圾收集过程必须停顿所有用户线程的其中一个重要原因，即使是号称停顿时间可控，或者（几乎）不会发生停顿的 CMS、G1、ZGC 等收集器，枚举根节点时也是必须要停顿的。

由于目前主流 Java 虚拟机使用的都是准确式垃圾收集，所以当用户线程停顿下来之后，其实并不需要一个不漏地检查完所有执行上下文和全局的引用位置，虚拟机应当是有办法直接得到哪些地方存放着对象引用的。在 HotSpot 的解决方案里，是使用一组称为 OopMap 的数据结构来达到这个目的。一旦类加载动作完成的时候， HotSpot 就会把对象内什么偏移量上是什么类型的数据计算出来，在即时编译过程中，也会在特定的位置记录下栈里和寄存器里哪些位置是引用。这样收集器在扫描时就可以直接得知这些信息了，并不需要真正一个不漏地从方法区等 GC Roots 开始查找。

2.4.2 安全点

在 OopMap 的协助下，HotSpot 可以快速准确地完成 GC Roots 枚举，但一个很现实的问题随之而来：可能导致引用关系变化，或者说导致 OopMap 内容变化的指令非常多，如果为每一条指令都生成对应的 OopMap，那将会需要大量的额外存储空间，这样垃圾收集伴随而来的空间成本就会变得无法忍受的高昂。

实际上 HotSpot 也的确没有为每条指令都生成 OopMap，前面已经提到，只是在“特定的位置”记录了这些信息，这些位置被称为安全点（Safepoint）。有了安全点的设定，也就决定了用户程序执行时并非在代码指令流的任意位置都能够停顿下来开始垃圾收集，而是强制要求必须执行到达安全点后才能够暂停。因此，安全点的选定既不能太少以至于让收集器等待时间过长，也不能太过频繁以至于过分增大运行时的内存负荷。安全点位置的选取基本上是以“是否具有让程序长时间执行的特征”为标准进行选定的，因为每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这样的原因而长时间执行，“长时间执行”的最明显特征就是指令序列的复用，例如方法调用、循环跳转、异常跳转等都属于指令序列复用，所以只有具有这些功能的指令才会产生安全点。

对于安全点，另外一个需要考虑的问题是，如何在垃圾收集发生时让所有线程（这里其实不包括执行 JNI 调用的线程）都跑到最近的安全点，然后停顿下来。这里有两种方案可供选择：抢先式中断（Preemptive Suspension）和主动式中断（Voluntary Suspension），抢先式中断不需要线程的执行代码主动去配合，在垃圾收集发生时，系统首先把所有用户线程全部中断，如果发现有用户线程中断的地方不在安全点上，就恢复这条线程执行，让它一会再重新中断，直到跑到安全点上。现在几乎没有虚拟机实现采用抢先式中断来暂停线程响应 GC 事件。

而主动式中断的思想是当垃圾收集需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志位，各个线程执行过程时会不停地主动去轮询这个标志，一旦发现中断标志为真时就自己在最近的安全点上主动中断挂起。轮询标志的地方和安全点是重合的，另外还要加上所有创建对象和其他需要在 Java 堆上分配内存的地方，这是为了检查是否即将要发生垃圾收集，避免没有足够内存分配新对象。

由于轮询操作在代码中会频繁出现，这要求它必须足够高效。HotSpot 使用内存保护陷阱的方式， 把轮询操作精简至只有一条汇编指令的程度。

2.4.3 安全区域

使用安全点的设计似乎已经完美解决如何停顿用户线程，让虚拟机进入垃圾回收状态的问题了， 但实际情况却并不一定。安全点机制保证了程序执行时，在不太长的时间内就会遇到可进入垃圾收集过程的安全点。但是，程序“不执行”的时候呢？所谓的程序不执行就是没有分配处理器时间，典型的场景便是用户线程处于 Sleep 状态或者 Blocked 状态，这时候线程无法响应虚拟机的中断请求，不能再走到安全的地方去中断挂起自己，虚拟机也显然不可能持续等待线程重新被激活分配处理器时间。对于这种情况，就必须引入安全区域（Safe Region）来解决。

安全区域是指能够确保在某一段代码片段之中，引用关系不会发生变化，因此，在这个区域中任意地方开始垃圾收集都是安全的。我们也可以把安全区域看作被扩展拉伸了的安全点。

当用户线程执行到安全区域里面的代码时，首先会标识自己已经进入了安全区域，那样当这段时间里虚拟机要发起垃圾收集时就不必去管这些已声明自己在安全区域内的线程了。当线程要离开安全区域时，它要检查虚拟机是否已经完成了根节点枚举（或者垃圾收集过程中其他需要暂停用户线程的阶段），如果完成了，那线程就当作没事发生过，继续执行；否则它就必须一直等待，直到收到可以离开安全区域的信号为止。

2.4.4 记忆集与卡表

为解决对象跨代引用所带来的问题，垃圾收集器在新生代中建立了名为记忆集（Remembered Set）的数据结构，用以避免把整个老年代加进 GC Roots 扫描范围。

记忆集是一种用于记录从非收集区域指向收集区域的指针集合的抽象数据结构。下面列举三种方式实现记忆集：

（1）字长精度：每个记录精确到一个机器字长（就是处理器的寻址位数，如常见的 32 位或 64 位，这个精度决定了机器访问物理内存地址的指针长度），该字包含跨代指针。

（2）对象精度：每个记录精确到一个对象，该对象里有字段含有跨代指针。

（3）卡精度：每个记录精确到一块内存区域，该区域内有对象含有跨代指针。

其中，第三种“卡精度”所指的是用一种称为“卡表”（Card Table）的方式去实现记忆集，这也是目前最常用的一种记忆集实现形式。前面定义中提到记忆集其实是一种“抽象”的数据结构，抽象的意思是只定义了记忆集的行为意图，并没有定义其行为的具体实现。卡表就是记忆集的一种具体实现，它定义了记忆集的记录精度、与堆内存的映射关系等。关于卡表与记忆集的关系，读者不妨按照 Java 语言中 HashMap 与 Map 的关系来类比理解。

卡表最简单的形式可以只是一个字节数组，而 HotSpot 虚拟机确实也是这样做的。字节数组 CARD_TABLE 的每一个元素都对应着其标识的内存区域中一块特定大小的内存块，这个内存块被称作“卡页”（Card Page）。一般来说，卡页大小都是以 2 的 N 次幂的字节数。

一个卡页的内存中通常包含不止一个对象，只要卡页内有一个（或更多）对象的字段存在着跨代指针，那就将对应卡表的数组元素的值标识为 1，称为这个元素变脏（Dirty），没有则标识为 0。在垃圾收集发生时，只要筛选出卡表中变脏的元素，就能轻易得出哪些卡页内存块中包含跨代指针，把它们加入 GC Roots 中一并扫描。

2.4.5 写屏障

上面解决了如何使用记忆集来缩减 GC Roots 扫描范围的问题，但还没有解决卡表元素如何维护的问题，例如它们何时变脏、谁来把它们变脏等。

卡表元素何时变脏的答案是很明确的——有其他分代区域中对象引用了本区域对象时，其对应的卡表元素就应该变脏，变脏时间点原则上应该发生在引用类型字段赋值的那一刻。但问题是如何变脏，即如何在对象赋值的那一刻去更新维护卡表呢？

在 HotSpot 虚拟机里是通过写屏障（Write Barrier）技术维护卡表状态的。写屏障可以看作在虚拟机层面对“引用类型字段赋值”这个动作的 AOP 切面，在引用对象赋值时会产生一个环形（Around）通知，供程序执行额外的动作，也就是说赋值的前后都在写屏障的覆盖范畴内。在赋值前的部分的写屏障叫作写前屏障（Pre-Write Barrier），在赋值后的则叫作写后屏障（Post-Write Barrier）。HotSpot 虚拟机的许多收集器中都有使用到写屏障，但直至 G1 收集器出现之前，其他收集器都只用到了写后屏障。

应用写屏障后，虚拟机就会为所有赋值操作生成相应的指令，一旦收集器在写屏障中增加了更新卡表操作，无论更新的是不是老年代对新生代对象的引用，每次只要对引用进行更新，就会产生额外的开销，不过这个开销与 Minor GC 时扫描整个老年代的代价相比还是低得多的。

除了写屏障的开销外，卡表在高并发场景下还面临着“伪共享”（False Sharing）问题。伪共享是处理并发底层细节时一种经常需要考虑的问题，现代中央处理器的缓存系统中是以缓存行（Cache Line） 为单位存储的，当多线程修改互相独立的变量时，如果这些变量恰好共享同一个缓存行，就会彼此影响（写回、无效化或者同步）而导致性能降低，这就是伪共享问题。为了避免伪共享问题，一种简单的解决方案是不采用无条件的写屏障，而是先检查卡表标记，只有当该卡表元素未被标记过时才将其标记为变脏。

2.4.6 并发的可达性分析

当前主流编程语言的垃圾收集器基本上都是依靠可达性分析算法来判定对象是否存活的，可达性分析算法理论上要求全过程都基于一个能保障一致性的快照中才能够进行分析， 这意味着必须全程冻结用户线程的运行。

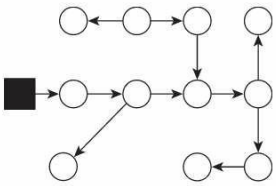
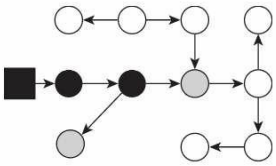
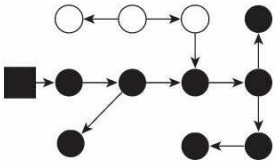
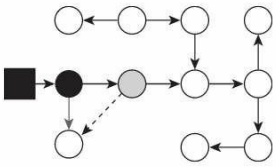
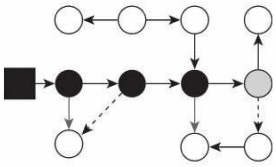
GC Roots 里的对象对于堆中来说，还是少数，另外，在各种优化下（OopMap），时间已经变得十分短暂且固定了。

但是从 GC Roots 再继续向下遍历对象图，这一步骤必然随着 Java 堆里对象的增多而停顿时间变长。现在垃圾回收算法都有标记阶段，如果标记阶段随着堆的变大而等比例增加停顿时间，那几乎会影响所有的垃圾回收器。

在并发状态下，必须要在一个能保障一致性的快照上才能进行图的遍历。现在引入三色标记来推导：

1）白色：表示对象尚未被垃圾收集器访问过。显然在可达性分析刚刚开始的阶段，所有的对象都是白色的，若在分析结束的阶段，仍然是白色的对象，即代表不可达。
2）黑色：表示对象已经被垃圾收集器访问过，且这个对象的所有引用都已经扫描过。黑色的对象代表已经扫描过，它是安全存活的，如果有其他对象引用指向了黑色对象，无须重新扫描一遍。黑色对象不可能直接（不经过灰色对象）指向某个白色对象。
3）灰色：表示对象已经被垃圾收集器访问过，但这个对象上至少存在一个引用还没有被扫描过。

三色标记在图中可看做以灰色为波峰的波纹从黑到白推进的过程。假设用户线程和收集器并发工作，收集器在对象图上标颜色，同时用户线程在改变引用关系——即修改对象图的结构，这样就会出现两种结果：
（1）把原本消亡的对象错误标记为存活。
（2）把原本存活的对象错误标记为已消亡。

	初始状态，只有 GC Roots 是黑色的。 请注意图中的箭头，引用是有向的，对象只有被黑色对象引用才能存活，否则，如果没有黑色对象引用它，它再如何引用其他对象都是会消亡的
	扫描过程中，以灰色为波峰的波纹从黑向白推进，灰色对象是黑、白对象的分界线
	扫描顺利完成，此时黑色对象就是存活的对象，白色对象就是已消亡可回收的对象
	但如果用户线程在标记进行时并发修改了引用关系，扫描就不会如此顺利了。 譬如在波纹推进过程中，正在扫描的灰色对象的一个引用被切断了，同时原来引用的对象又与已扫描过的黑对象建立了引用关系
	又譬如，这种切断后重新被黑色对象引用的对象可能是原有引用链中的一部分。 由于黑色对象不会重新扫描，这将导致扫描结束后出现两个被黑色对象引用的对象仍是白色，这个对象就会消失，这就很危险了

针对对象消失问题，当且仅当满足下面两个条件：

·1）赋值器插入了一条或多条从黑色对象到白色对象的新引用。

·2）赋值器删除了全部从灰色对象到该白色对象的直接或间接引用。

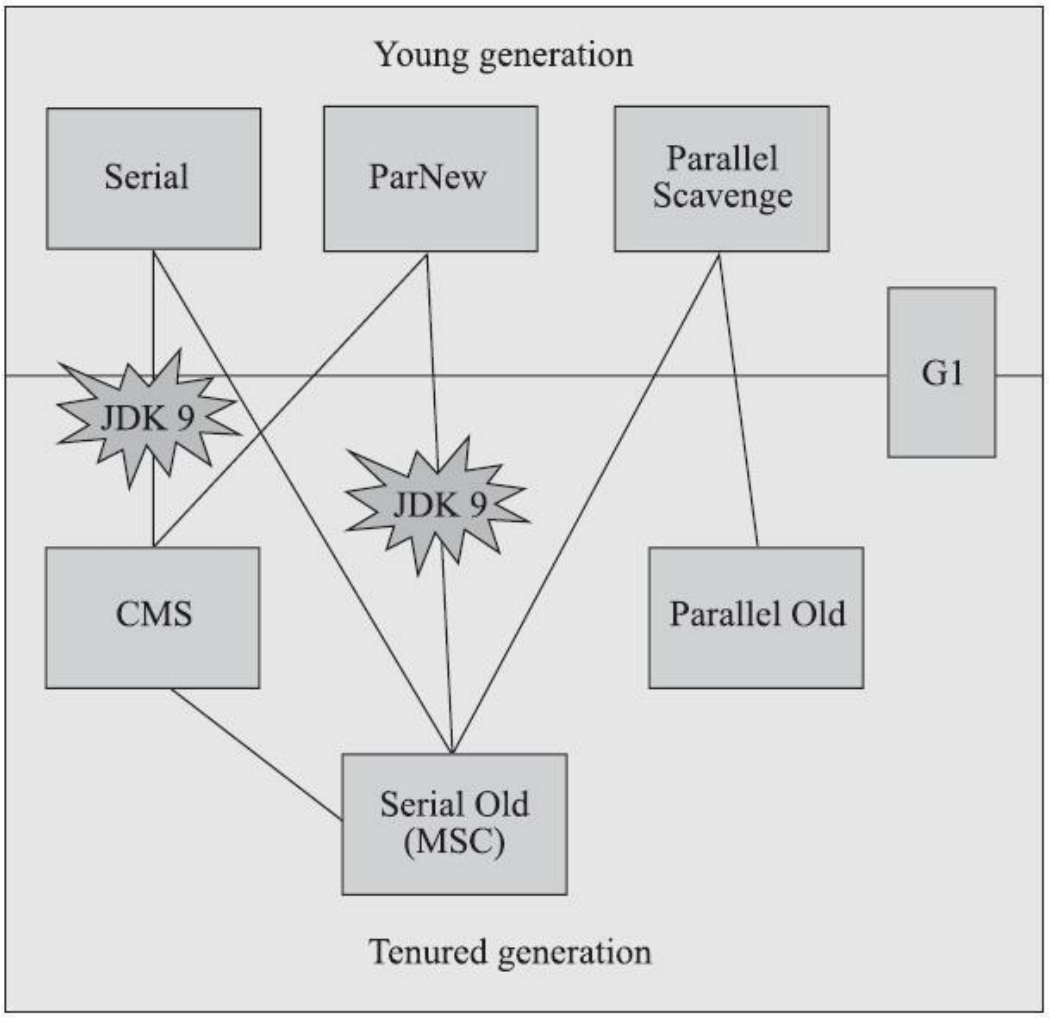
·

因此，只需要破坏这两个条件其中一个，就可以解决并发条件下对象消失问题。针对这两个问题，产生两种解决方案：增量更新和原始快照。

增量更新要破坏的是第一个条件，当黑色对象插入新的指向白色对象的引用关系时，就将这个新插入的引用记录下来，等并发扫描结束之后，再将这些记录过的引用关系中的黑色对象为根，重新扫描一次。这可以简化理解为，黑色对象一旦新插入了指向白色对象的引用之后，它就变回灰色对象了。

原始快照要破坏的是第二个条件，当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用记录下来，在并发扫描结束之后，再将这些记录过的引用关系中的灰色对象为根，重新扫描一次。这也可以简化理解为，无论引用关系删除与否，都会按照刚刚开始扫描那一刻的对象图快照来进行搜索。

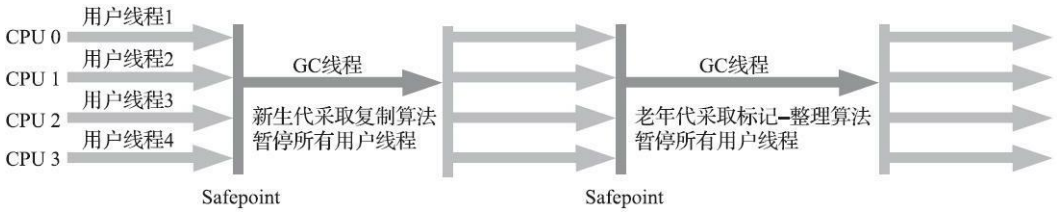
2.5 经典垃圾收集器



该图展示了七种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用[3]，图中收集器所处的区域，则表示它是属于新生代收集器抑或是老年代收集器。虽然我们会对各个收集器进行比较，但并非为了挑选一个最好的收集器出来，虽然垃圾收集器的技术在不断进步，但直到现在还没有最好的收集器出现，更加不存在“万能”的收集器，所以我们选择的只是对具体应用最合适的收集器。

2.5.1 Serial 收集器

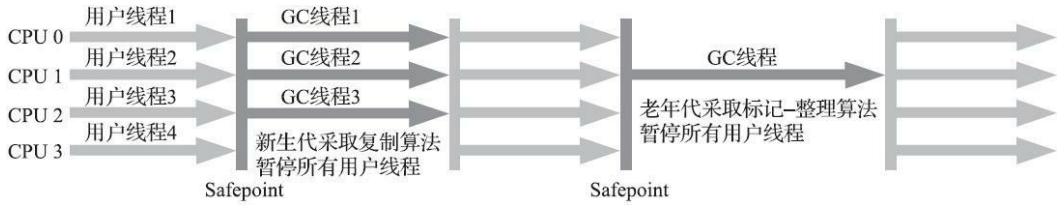
Serial 收集器是一个单线程工作的收集器，但它的“单线程”的意义并不仅仅是说明它只会使用一个处理器或一条收集线程去完成垃圾收集工作，更重要的是强调在它进行垃圾收集时，必须暂停其他所有工作线程，直到它收集结束。



虽然 Serial 收集器强行停止用户线程带来了很多恶劣体验，但是它还是有着优于其他收集器的地方，就是简单而高效（与其他收集器的单线程相比）。在客户端模式下使用该收集器是一个很好的选择。

2.5.2 ParNew 收集器

ParNew 收集器实质上是 Serial 收集器的多线程并行版本，除了同时使用多条线程进行垃圾收集之 外，其余的行为包括 Serial 收集器可用的所有控制参数（例如 ： -XX ： SurvivorRatio 、 -XX ： PretenureSizeThreshold 、 -XX ： HandlePromotionFailure 等）、收集算法、Stop The World、对象分配规则、回收策略等都与 Serial 收集器完全一致，在实现上这两种收集器也共用了相当多的代码。



ParNew 收集器除了支持多线程并行收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是不少运行在服务端模式下的 HotSpot 虚拟机。其中有一个与功能、性能无关但其实很重要的原因是：除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。

可以说直到 CMS 的出现才巩固了 ParNew 的地位，但成也萧何败也萧何，随着垃圾收集器技术的不断改进，更先进的 G1 收集器带着 CMS 继承者和替代者的光环登场。G1 是一个面向全堆的收集器，不再需要其他新生代收集器的配合工作。所以自 JDK 9 开始，ParNew 加 CMS 收集器的组合就不再是官方推荐的服务端模式下的收集器解决方案了。官方希望它能完全被 G1 所取代，甚至还取消了 ParNew 加 Serial Old 以及 Serial 加 CMS 这两组收集器组合的支持（其实原本也很少人这样使用），并直接取消了- XX：+UseParNewGC 参数，这意味着 ParNew 和 CMS 从此只能互相搭配使用，再也没有其他收集器能够和它们配合了。读者也可以理解为从此以后，ParNew 合并入 CMS，成为它专门处理新生代的组成部分。ParNew 可以说是 HotSpot 虚拟机中第一款退出历史舞台的垃圾收集器。

2.5.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器也是一款新生代收集器，它同样是基于标记-复制算法实现的收集器，也是能够并行收集的多线程收集器。

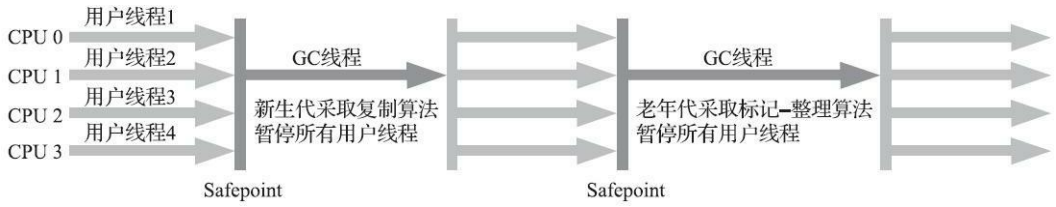
Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而 Parallel Scavenge 收集器的目标则是达到一个可控制的吞吐量（Throughput）。

$$吞吐量=\frac{运行用户代码时间}{运行用户代码时间+运行垃圾收集时间}$$

如果虚拟机完成某个任务，用户代码加上垃圾收集总共耗费了 100 分钟，其中垃圾收集花掉 1 分 钟，那吞吐量就是 99%。停顿时间越短就越适合需要与用户交互或需要保证服务响应质量的程序，良好的响应速度能提升用户体验；而高吞吐量则可以最高效率地利用处理器资源，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的分析任务。由于与吞吐量关系密切，Parallel Scavenge 收集器也经常被称作“吞吐量优先收集器”。

2.5.4 Serial Old 收集器

Serial Old 是 Serial 收集器的老年代版本，它同样是一个单线程收集器，使用标记-整理算法。这个收集器的主要意义也是供客户端模式下的 HotSpot 虚拟机使用。如果在服务端模式下，它也可能有两种用途：一种是在 JDK 5 以及之前的版本中与 Parallel Scavenge 收集器搭配使用，另外一种就是作为 CMS 收集器发生失败时的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。

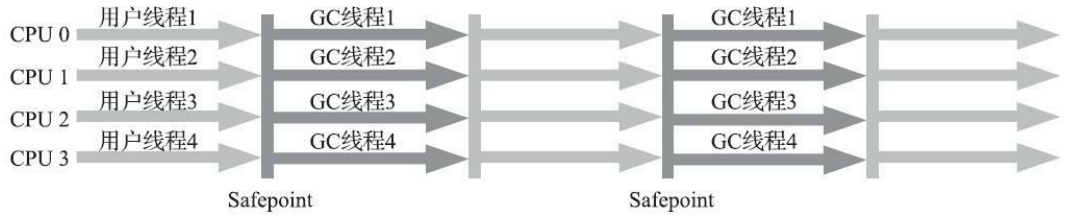


Parallel Scavenge 收集器架构中本身有 PS MarkSweep 收集器来进行老年代收集，并非直接调用 Serial Old 收集器，但是这个 PS MarkSweep 收集器与 Serial Old 的实现几乎是一样的，所以在官方的许多资料中都是直接以 Serial Old 代替 PS MarkSweep 进行讲解。

2.5.5 Parallel Old 收集器

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，支持多线程并发收集，基于标记-整理算法实现。这个收集器是直到 JDK 6 时才开始提供的，在此之前，新生代的 Parallel Scavenge 收集器一直处于相当尴尬的状态，原因是如果新生代选择了 Parallel Scavenge 收集器，老年代除了 Serial Old（PS MarkSweep）收集器以外别无选择，其他表现良好的老年代收集器，如 CMS 无法与它配合工作。由于老年代 Serial Old 收集器在服务端应用性能上的“拖累”，使用 Parallel Scavenge 收集器也未必能在整体上获得吞吐量最大化的效果。同样，由于单线程的老年代收集中无法充分利用服务器多处理器的并行处理能力，在老年代内存空间很大而且硬件规格比较高级的运行环境中，这种组合的总吞吐量甚至不一定比 ParNew 加 CMS 的组合来得优秀。

直到 Parallel Old 收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的搭配组合，在注重吞吐量或者处理器资源较为稀缺的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器这个组合。



2.5.6 CMS 收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java 应用集中在互联网网站或者基于浏览器的 B/S 系统的服务端上，这类应用通常都会较为关注服务的响应速度，希望系统停顿时间尽可能短，以给用户带来良好的交互体验。CMS 收集器就非常符合这类应用的需求。

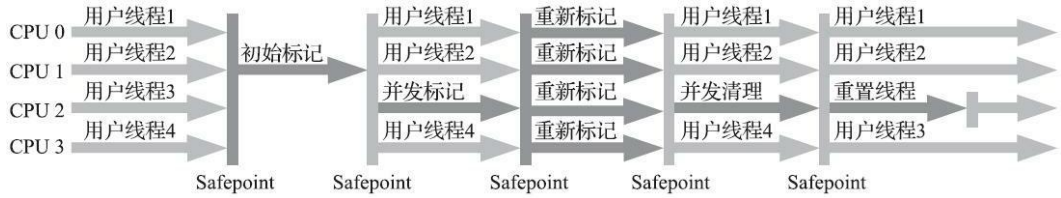
CMS 收集器是基于标记-清除算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为四个步骤，包括：

1）初始标记（CMS initial mark）：初始标记仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快。

2）并发标记（CMS concurrent mark）：并发标记阶段就是从 GC Roots 的直接关联对象开始遍历整个对象图的过程，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行。

3）重新标记（CMS remark）：重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短。

4）并发清除（CMS concurrent sweep）：并发清除阶段清理删除掉标记阶段判断的已经死亡的对象，由于不需要移动存活对象，所以这个阶段也是可以与用户线程同时并发的。



由于在整个过程中耗时最长的并发标记和并发清除阶段中，垃圾收集器线程都可以与用户线程一起工作，所以从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。

尽管 CMS 收集器成功实现并发低停顿，但是还是有缺点：

1）CMS 收集器对处理器资源十分敏感。在并发阶段，虽然不会导致用户线程停顿，但却会因为占用了一部分线程而导致应用程序变慢，降低总吞吐量。当处理器核心不足 4 个时，CMS 对用户程序的影响就可能变得很大。

2）CMS 收集器无法处理“浮动垃圾”。在 CMS 的并发标记和并发清理阶段，用户线程还是在运行的，程序在运行自然就会有新的垃圾对象不断产生，但这一部分垃圾对象是出现在标记过程结束以后，CMS 无法在档次手机中处理他们，只好留待下一次垃圾回收时处理。这部分垃圾就称为“浮动垃圾”。由于浮动垃圾的出现，有可能出现“Con-current Mode Failure”失败进而导致另一次完全“Stop The World”的 Full GC 的产生。

3）CMS 收集器基于标记-清除算法。这个算法会产生大量空间碎片，这就会给大对象的分配带来很大麻烦。

2.5.7 Garbage First 收集器

Garbage First（简称 G1）收集器是垃圾收集器技术发展历史上的里程碑式的成果，它开创了收集器面向局部收集的设计思路和基于 Region 的内存布局形式。

G1 是一款主要面向服务端应用的垃圾收集器。HotSpot 开发团队最初赋予它的期望是（在比较长期的）未来可以替换掉 JDK 5 中发布的 CMS 收集器。现在这个期望目标已经实现过半了，JDK 9 发布之日，G1 宣告取代 Parallel Scavenge 加 Parallel Old 组合，成为服务端模式下的默认垃圾收集器，而 CMS 则沦落至被声明为不推荐使用（Deprecate）的收集器。

作为 CMS 收集器的替代者和继承人，设计者们希望做出一款能够建立起“停顿时间模型”（Pause Prediction Model）的收集器，停顿时间模型的意思是能够支持指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间大概率不超过 N 毫秒这样的目标。

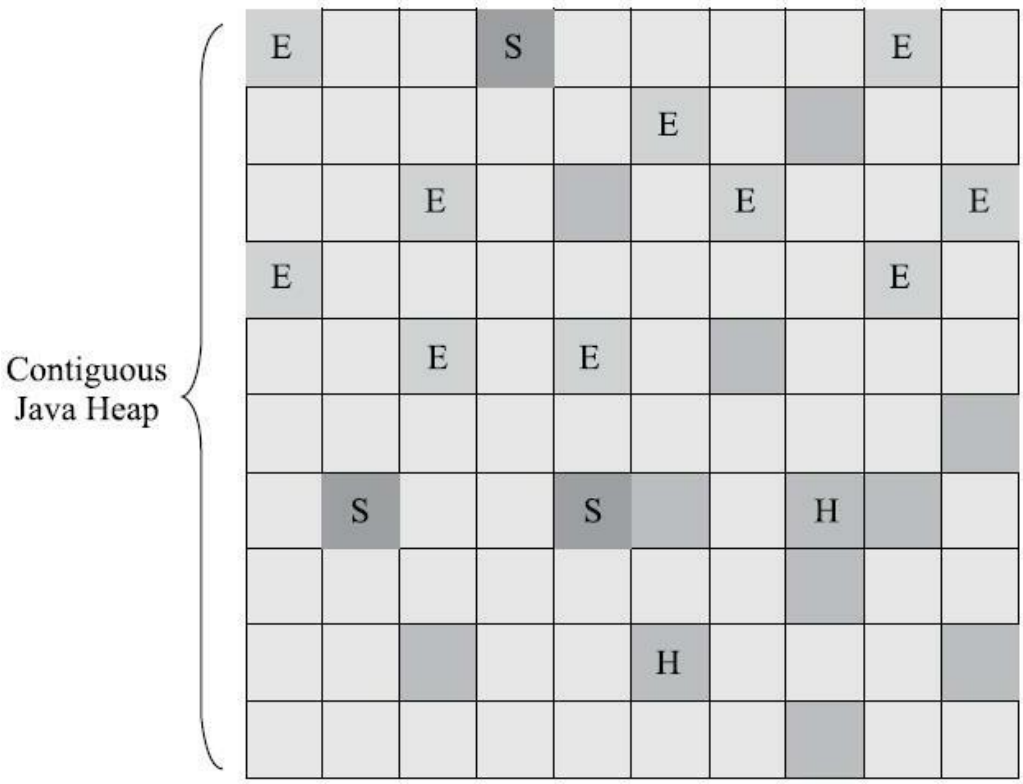
要做到这个目标，首先是思想上的转变：在 G1 收集器出现之前的所有其他收集器，包括 CMS 在内，垃圾收集的目标范围要么是整个新生代（Minor GC），要么就是整个老年代（Major GC），再要么就是整个 Java 堆（Full GC）。而 G1 跳出了这个樊笼，它可以面向堆内存任何部分来组成回收集（Collection Set，一般简称 CSet）进行回收，衡量标准不再是它属于哪个分代，而是哪块内存中存放的垃圾数量最多，回收收益最大，这就是 G1 收集器的 Mixed GC 模式。

G1 开创的基于 Region 的堆内存布局是它能够实现这个目标的关键。虽然 G1 也仍是遵循分代收集理论设计的，但其堆内存的布局与其他收集器有非常明显的差异：G1 不再坚持固定大小以及固定数量的分代区域划分，而是把连续的 Java 堆划分为多个大小相等的独立区域（Region），每一个 Region 都可以根据需要，扮演新生代的 Eden 空间、Survivor 空间，或者老年代空间。收集器能够对扮演不同角色的 Region 采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的旧对象都能获取很好的收集效果。

Region 中还有一类特殊的 Humongous 区域，专门用来存储大对象。G1 认为只要大小超过了一个 Region 容量一半的对象即可判定为大对象。对于那些超过了整个 Region 容量的超级大对象，将会被存放在 N 个连续的 Humongous Region 之中，G1 的大多数行为都把 Humongous Region 作为老年代的一部分来进行看待。

虽然 G1 仍然保留新生代和老年代的概念，但新生代和老年代不再是固定的了，它们都是一系列区域（不需要连续）的动态集合。G1 收集器之所以能建立可预测的停顿时间模型，是因为它将 Region 作为单次回收的最小单元，即每次收集到的内存空间都是 Region 大小的整数倍，这样可以有计划地避免在整个 Java 堆中进行全区域的垃圾收集。

更具体的处理思路是让 G1 收集器去跟踪各个 Region 里面的垃圾堆积的“价值”大小，价值即回收所获得的空间大小以及回收所需时间的经验值，然后在后台维护一个优先级列表，每次根据用户设定允许的收集停顿时间（使用参数-XX：MaxGCPauseMillis 指定，默认值是 200 毫秒），优先处理回收价值收益最大的那些 Region，这也就是“Garbage First”名字的由来。



G1 收集器的运作过程大致可划分为以下四个步骤：

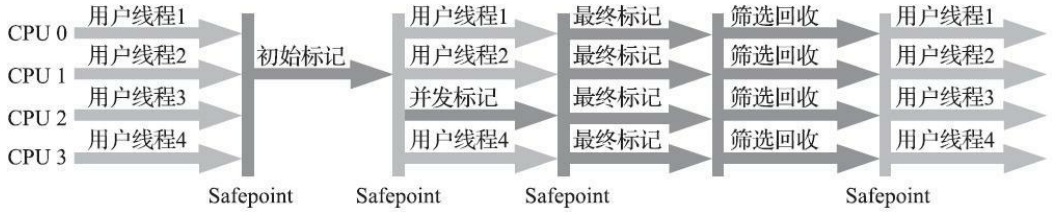
1）初始标记（Initial Marking）：仅仅只是标记一下 GC Roots 能直接关联到的对象，并且修改 TAMS 指针的值，让下一阶段用户线程并发运行时，能正确地在可用的 Region 中分配新对象。这个阶段需要停顿线程，但耗时很短，而且是借用进行 Minor GC 的时候同步完成的，所以 G1 收集器在这个阶段实际并没有额外的停顿。

a. 注：TAMS。G1 为每一个 Region 设计了两个名为 TAMS（Top at Mark Start）的指针，把 Region 中的一部分空间划分出来用于并发回收过程中的新对象分配，并发回收时新分配的对象地址都必须要在这两个指针位置以上。这实际上是解决了 CMS 无法回收浮动垃圾的特点。但，如果 G1 中，内存回收的速度赶不上内存分配的速度，G1 收集器也要被迫冻结用户线程的执行，导致 Full GC 而产生长时间“Stop The World”。

2）并发标记（Concurrent Marking）：从 GC Root 开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。当对象图扫描完成以后，还要重新处理 SATB 记录下的在并发时有引用变动的对象。

3）最终标记（Final Marking）：对用户线程做另一个短暂的暂停，用于处理并发阶段结束后仍遗留下来的最后那少量的 SATB 记录。

4）筛选回收（Live Data Counting and Evacuation）：负责更新 Region 的统计



数据，对各个 Region 的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个 Region 构成回收集，然后把决定回收的那一部分 Region 的存活对象复制到空的 Region 中，再清理掉整个旧 Region 的全部空间。这里的操作涉及存活对象的移动，是必须暂停用户线程，由多条收集器线程并行完成的。

毫无疑问，可以由用户指定期望的停顿时间是 G1 收集器很强大的一个功能，设置不同的期望停顿时间，可使得 G1 在不同应用场景中取得关注吞吐量和关注延迟之间的最佳平衡。

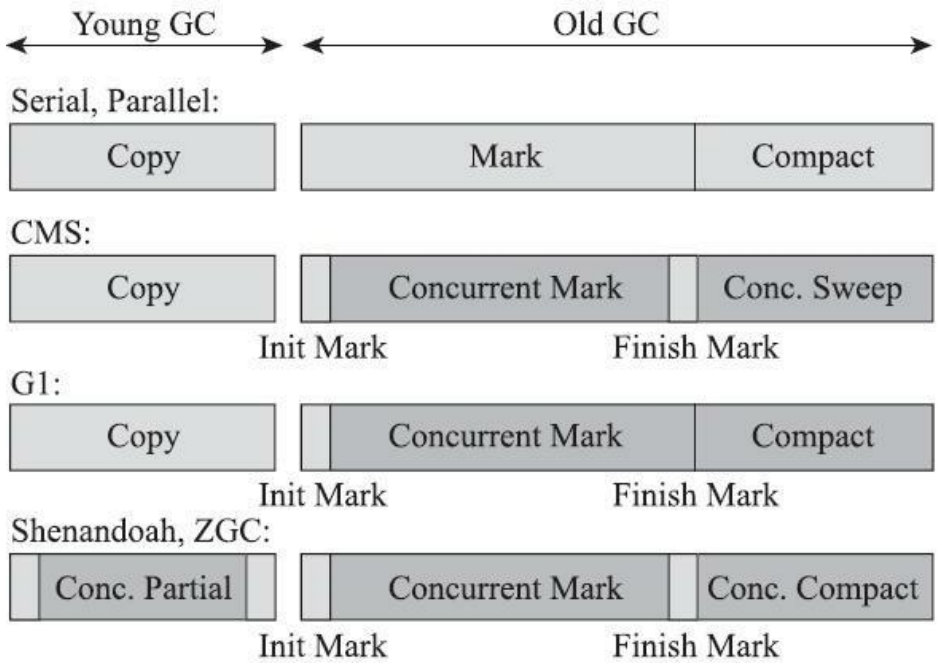
从 G1 开始，最先进的垃圾收集器的设计导向都不约而同地变为追求能够应付应用的内存分配速率（Allocation Rate），而不追求一次把整个 Java 堆全部清理干净。这样，应用在分配，同时收集器在收集，只要收集的速度能跟得上对象分配的速度，那一切就能运作得很完美。这种新的收集器设计思路从工程实现上看是从 G1 开始兴起的，所以说 G1 是收集器技术发展的一个里程碑。

G1 收集器常会被拿来与 CMS 收集器互相比较，毕竟它们都非常关注停顿时间的控制。

2.6 低延迟垃圾收集器

衡量垃圾收集器的三项最重要的指标是：内存占用（Footprint）、吞吐量（Throughput）和延迟（Latency），三者共同构成了一个“不可能三角[1]”。三者总体的表现会随技术进步而越来越好，但是要在这三个方面同时具有卓越表现的“完美”收集器是极其困难甚至是不可能的，一款优秀的收集器通常最多可以同时达成其中的两项。

在内存占用、吞吐量和延迟这三项指标里，延迟的重要性日益凸显，越发备受关注。其原因是随着计算机硬件的发展、性能的提升，我们越来越能容忍收集器多占用一点点内存；硬件性能增长，对软件系统的处理能力是有直接助益的，硬件的规格和性能越高，也有助于降低收集器运行时对应用程序的影响，换句话说，吞吐量会更高。但对延迟则不是这样，硬件规格提升，准确地说是内存的扩大，对延迟反而会带来负面的效果，这点也是很符合直观思维的：虚拟机要回收完整的 1TB 的堆内 存，毫无疑问要比回收 1GB 的堆内存耗费更多时间。



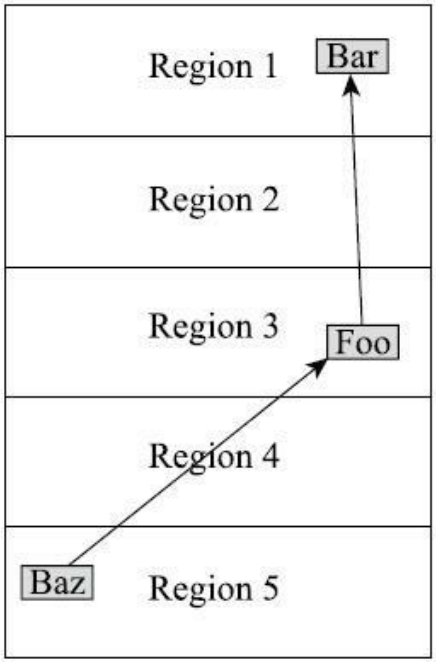
图中浅色阶段表示必须挂起用户线程，深色表示收集器线程与用户线程是并发工作的。可以看出在 CMS 和 G1 之前的全部收集器，其工作的所有步骤都会产生“Stop The World”式的停顿；CMS 和 G1 分别使用增量更新和原始快照技术，实现了标记阶段的并发，不会因管理的堆内存变大，要标记的对象变多而导致停顿时间随之增长。但是对于标记阶段之后的处理，仍未得到妥善解决。CMS 使用标记-清除算法，虽然避免了整理阶段收集器带来的停顿，但是清除算法不论如何优化改进，在设计原理上避免不了空间碎片的产生，随着空间碎片不断淤积最终依然逃不过“Stop The World”的命运。G1 虽然可以按更小的粒度进行回收，从而抑制整理阶段出现时间过长的停顿，但毕竟也还是要暂停的。

2.6.1 Shenandoah 收集器

Shenandoah 更像是 G1 的下一代继承者，它们两者有着相似的堆内存布局，在初始标记、并发标记等许多阶段的处理思路上都高度一致，甚至还直接共享了一部分实现代码，这使得部分对 G1 的打磨改进和 Bug 修改会同时反映在 Shenandoah 之上，而由于 Shenandoah 加入所带来的一些新特性，也有部分会出现在 G1 收集器中，譬如在并发失败后作为“逃生门”的 Full GC[3]，G1 就是由于合并了 Shenandoah 的代码才获得多线程 Full GC 的支持。

虽然 Shenandoah 也是使用基于 Region 的堆内存布局，同样有着用于存放大对象的 Humongous Region，默认的回收策略也同样是优先处理回收价值最大的 Region，但在管理堆内存方面，它与 G1 至少有三个明显的不同之处，最重要的当然是支持并发的整理算法，G1 的回收阶段是可以多线程并行的，但却不能与用户线程并发。其次，Shenandoah（目前）是默认不使用分代收集的，换言之，不会有专门的新生代 Region 或者老年代 Region 的存在，没有实现分代，并不是说分代对 Shenandoah 没有价值， 这更多是出于性价比的权衡，基于工作量上的考虑而将其放到优先级较低的位置上。最后， Shenandoah 摒弃了在 G1 中耗费大量内存和计算资源去维护的记忆集，改用名为“连接矩阵”（Connection Matrix）的全局数据结构来记录跨 Region 的引用关系，降低了处理跨代指针时的记忆集维护消耗，也降低了伪共享问题的发生概率。

Region Number	1	2	3	4	5	6	7
1		×					
2							
3					×		
4							
5							
6							
7							



如图所示：Region5 引用 Region3 就在矩阵的[5,3]位置打个标记。

Shenandoah 收集器的工作过程大致可以划分为以下九个阶段：

1）初始标记（Initial Marking）：与 G1 一样，首先标记与 GC Roots 直接关联的对象，这个阶段仍是“Stop The World”的，但停顿时间与堆大小无关，只与 GC Roots 的数量相关。

2）并发标记（Concurrent Marking）：与 G1 一样，遍历对象图，标记出全部

可达的对象，这个阶段是与用户线程一起并发的，时间长短取决于堆中存活对象的数量以及对象图的结构复杂程度。

3）最终标记（Final Marking）：与 G1 一样，处理剩余的 SATB 扫描，并在这个阶段统计出回收价值最高的 Region，将这些 Region 构成一组回收集（Collection Set）。最终标记阶段也会有一小段短暂的停顿。

4）并发清理（Concurrent Cleanup）：这个阶段用于清理那些整个区域内连一个存活对象都没有找到的 Region（这类 Region 被称为 Immediate Garbage Region）。

5）并发回收（Concurrent Evacuation）：并发回收阶段是 Shenandoah 与之前 HotSpot 中其他收集器的核心差异。在这个阶段，Shenandoah 要把回收集里面的存活对象先复制一份到其他未被使用的 Region 之中。复制对象这件事情如果将用户线程冻结起来再做那是相当简单的，但如果两者必须要同时并发进行的话，就变得复杂起来了。其困难点是在移动对象的同时，用户线程仍然可能不停对被移动的对象进行读写访问，移动对象是一次性的行为，但移动之后整个内存中所有指向该对象的引用都还是旧对象的地址，这是很难一瞬间全部改变过来的。对于并发回收阶段遇到的这些困难，Shenandoah 将会通过读屏障和被称为“Brooks Pointers”的转发指针来解决。并发回收阶段运行的时间长短取决于回收集的大小。

6）初始引用更新（Initial Update Reference）：并发回收阶段复制对象结束后，还需要把堆中所有指向旧对象的引用修正到复制后的新地址，这个操作称为引用更新。引用更新的初始化阶段实际上并未做什么具体的处理，设立这个阶段只是为了建立一个线程集合点，确保所有并发回收阶段中进行的收集器线程都已完成分配给它们的对象移动任务而已。初始引用更新时间很短，会产生一个非常短暂的停顿。

7）并发引用更新（Concurrent Update Reference）：真正开始进行引用更新操作，这个阶段是与用户线程一起并发的，时间长短取决于内存中涉及的引用数量的多少。并发引用更新与并发标记不同，它不再需要沿着对象图来搜索，只需要按照内存物理地址的顺序，线性地搜索出引用类型，把旧值改为新值即可。

8）最终引用更新（Final Update Reference）：解决了堆中的引用更新后，还要修正存在于 GC Roots 中的引用。这个阶段是 Shenandoah 的最后一次停顿，停顿时间只与 GC Roots 的数量相关。

9）并发清理（Concurrent Cleanup）：经过并发回收和引用更新之后，整个回收集中所有的 Region 已再无存活对象，这些 Region 都变成 Immediate Garbage Regions 了，最后再调用一次并发清理过程来回收这些 Region 的内存空间，供以后新对象分配使用。

其中最重要的三个阶段就是并发标记，并发回收，并发引用更新。

在并发回收阶段，使用了叫“Brooks Pointers”的转发指针。Brooks 提出，阿在对象头最前面统一增加一个新的引用字段，在正常情况下（不并发移动），该引用指向对象自己。当并发回收阶段复制对象到新的 Region 时，只需要将旧对象转发指针的引用位置指向新对象就行了。这样，对原来旧对象的访问就会转发到新对象。

2.6.2 ZGC 收集器

ZGC（“Z”并非什么专业名词的缩写，这款收集器的名字就叫作 Z Garbage Collector）是一款在 JDK 11 中新加入的具有实验性质[1]的低延迟垃圾收集器，是由 Oracle 公司研发的。2018 年 Oracle 创建了 JEP 333 将 ZGC 提交给 OpenJDK，推动其进入 OpenJDK 11 的发布清单之中。

ZGC 和 Shenandoah 的目标是高度相似的，都希望在尽可能对吞吐量影响不太大的前提下，实现在任意堆内存大小下都可以把垃圾收集的停顿时间限制在十毫秒以内的低延迟。但是 ZGC 和 Shenandoah 的实现思路又是差异显著的，如果说 RedHat 公司开发的 Shen-andoah 像是 Oracle 的 G1 收集器的实际继承者的话，那 Oracle 公司开发的 ZGC 就更像是 Azul System 公司独步天下的 PGC（Pauseless GC）和 C4（Concurrent Continuously Compacting Collector）收集器的同胞兄弟。

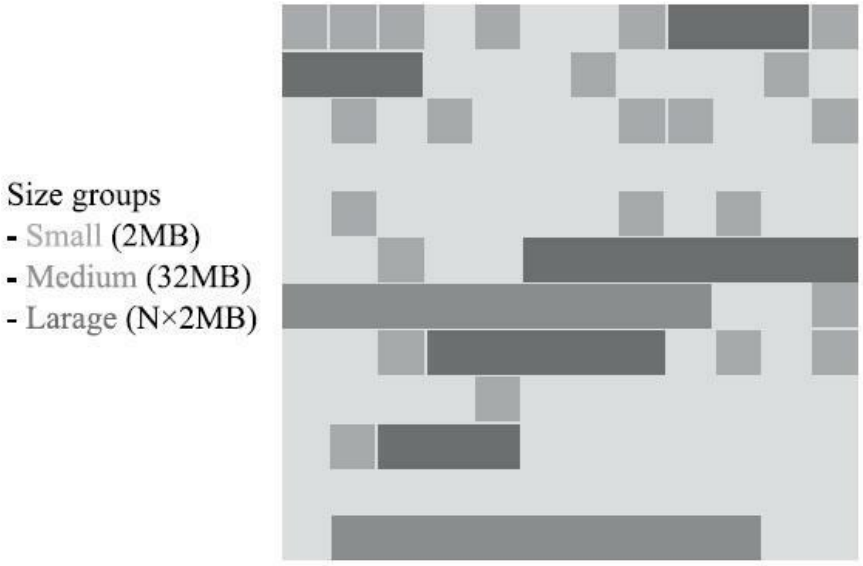
早在 2005 年，运行在 Azul VM 上的 PGC 就已经实现了标记和整理阶段都全程与用户线程并发运行的垃圾收集，而运行在 Zing VM 上的 C4 收集器是 PGC 继续演进的产物，主要增加了分代收集支持，大幅提升了收集器能够承受的对象分配速度。无论从算法还是实现原理上来讲，PGC 和 C4 肯定算是一脉相承的，而 ZGC 虽然并非 Azul 公司的产品，但也应视为这条脉络上的另一个节点，因为 ZGC 几乎所有的关键技术上，与 PGC 和 C4 都只存在术语称谓上的差别，实质内容几乎是一模一样的。相信到这里读者应该已经对 Java 虚拟机收集器常见的专业术语都有所了解了，如果不避讳专业术语的话，我们可以给 ZGC 下一个这样的定义来概括它的主要特征：ZGC 收集器是一款基于 Region 内存布局的，（暂时） 不设分代的，使用了读屏障、染色指针和内存多重映射等技术来实现可并发的标记-整理算法的，以低延迟为首要目标的一款垃圾收集器。

ZGC 与 Shenandoah 和 G1 一样，采用基于 Region 的堆内存布局，但与它们不同的是，ZGC 的 Region 具有动态性——动态创建和销毁，以及动态的区域容量大小。在 x64 硬件平台下，ZGC 的 Region 可以具有大、中、小三类：

1）小型 Region（Small Region）：容量固定为 2MB，用于放置小于 256KB 的小对象。

2）中型 Region（Medium Region）：容量固定为 32MB，用于放置大于等于 256KB 但小于 4MB 的对象。

3）大型 Region（Large Region）：容量不固定，可以动态变化，但必须为 2MB 的整数倍，用于放置 4MB 或以上的大对象。每个大型 Region 中只会存放一个大对象，这也预示着虽然名字叫作“大型 Region”，但它的实际容量完全有可能小于中型 Region，最小容量可低至 4MB。大型 Region 在 ZGC 的实现中是不会被重分配（重分配是 ZGC 的一种处理动作，用于复制对象的收集器阶段） 的，因为复制一个大对象的代价非常高昂。



在并发整理阶段，Shenandoah 使用转发指针和读屏障来实现并发整理，ZGC 虽然同样用到了读屏障，但用的却是一条与 Shenandoah 完全不同，更加复杂精巧的解题思路。

ZGC 收集器有一个标志性的设计是它采用的染色指针技术。（关于染色指针这里就不总结了，太长有点难理解，建议看书）

ZGC 的运作过程大致可划分为以下四个大的阶段。全部四个阶段都是可以并发执行的，仅是两个阶段中间会存在短暂的停顿小阶段，这些小阶段， 譬如初始化 GC Root 直接关联对象的 Mark Start，与之前 G1 和 Shenandoah 的 Initial Mark 阶段并没有什么差异。

1）并发标记（Concurrent Mark）：与 G1、Shenandoah 一样，并发标记是遍历对象图做可达性分析的阶段，前后也要经过类似于 G1、Shenandoah 的初始标记、最终标记（尽管 ZGC 中的名字不叫这些）的短暂停顿，而且这些停顿阶段所做的事情在目标上也是相类似的。与 G1、Shenandoah 不同的是，ZGC 的标记是在指针上而不是在对象上进行的，标记阶段会更新染色指针中的 Marked 0、Marked 1 标志位。

2）并发预备重分配（Concurrent Prepare for Relocate）：这个阶段需要根据特定的查询条件统计得出本次收集过程要清理哪些 Region，将这些 Region 组成重分配集（Relocation Set）。重分配集与 G1 收集器的回收集（Collection Set）还是有区别的，ZGC 划分 Region 的目的并非为了像 G1 那样做收益优先的增量回收。相反，ZGC 每次回收都会扫描所有的 Region，用范围更大的扫描成本换取省去 G1 中记忆集的维护成本。因此，ZGC 的重分配集只是决定了里面的存活对象会被重新复制到其他的 Region 中，里面的 Region 会被释放，而并不能说回收行为就只是针对这个集合里面的 Region 进行，因为标记过程是针对全堆的。此外，在 JDK 12 的 ZGC 中开始支持的类卸载以及弱引用的处理，也是在这个阶段中完成的。

3）并发重分配（Concurrent Relocate）：重分配是 ZGC 执行过程中的核心阶段，这个过程要把重分配集中的存活对象复制到新的 Region 上，并为重分配集中的每个 Region 维护一个转发表（Forward Table），记录从旧对象到新对象的转

向关系。得益于染色指针的支持，ZGC 收集器能仅从引用上就明确得知一个对象是否处于重分配集之中，如果用户线程此时并发访问了位于重分配集中的对象，这次访问将会被预置的内存屏障所截获，然后立即根据 Region 上的转发表记录将访问转发到新复制的对象上，并同时修正更新该引用的值，使其直接指向新对象，ZGC 将这种行为称为指针的“自愈”（Self- Healing）能力。这样做的好处是只有第一次访问旧对象会陷入转发，也就是只慢一次，对比 Shenandoah 的 Brooks 转发指针，那是每次对象访问都必须付出的固定开销，简单地说就是每次都慢，因此 ZGC 对用户程序的运行时负载要比 Shenandoah 来得更低一些。还有另外一个直接的好处是由于染色指针的存在，一旦重分配集中某个 Region 的存活对象都复制完毕后，这个 Region 就可以立即释放用于新对象的分配（但是转发表还得留着不能释放掉），哪怕堆中还有很多指向这个对象的未更新指针也没有关系，这些旧指针一旦被使用，它们都是可以自愈的。

4）并发重映射（Concurrent Remap）：重映射所做的就是修正整个堆中指向重分配集中旧对象的所有引用，这一点从目标角度看是与 Shenandoah 并发引用更新阶段一样的，但是 ZGC 的并发重映射并不是一个必须要“迫切”去完成的任务，因为前面说过，即使是旧引用，它也是可以自愈的，最多只是第一次使用时多一次转发和修正操作。重映射清理这些旧引用的主要目的是为了不变慢（还有清理结束后可以释放转发表这样的附带收益），所以说这并不是很“迫切”。因此，ZGC 很巧妙地把并发重映射阶段要做的工作，合并到了下一次垃圾收集循环中的并发标记阶段里去完成，反正它们都是要遍历所有对象的，这样合并就节省了一次遍历对象图的开销。一旦所有指针都被修正之后，原来记录新旧对象关系的转发表就可以释放掉了。

相比 G1、Shenandoah 等先进的垃圾收集器，ZGC 在实现细节上做了一些不同的权衡选择，譬如 G1 需要通过写屏障来维护记忆集，才能处理跨代指针，得以实现 Region 的增量回收。记忆集要占用大量的内存空间，写屏障也对正常程序运行造成额外负担，这些都是权衡选择的代价。ZGC 就完全没有使用记忆集，它甚至连分代都没有，连像 CMS 中那样只记录新生代和老年代间引用的卡表也不需要，因而完全没有用到写屏障，所以给用户线程带来的运行负担也要小得多。

但是，ZGC 的这种选择也限制了它能承受的对象分配速率不会太高，可以想象以下场景来理解 ZGC 的这个劣势：ZGC 准备要对一个很大的堆做一次完整的并发收集，假设其全过程要持续十分钟以上（请读者切勿混淆并发时间与停顿时间，ZGC 立的 Flag 是停顿时间不超过十毫秒），在这段时间里面，由于应用的对象分配速率很高，将创造大量的新对象，这些新对象很难进入当次收集的标记范 围，通常就只能全部当作存活对象来看待——尽管其中绝大部分对象都是朝生夕灭的，这就产生了大量的浮动垃圾。如果这种高速分配持续维持的话，每一次完整的并发收集周期都会很长，回收到的内存空间持续小于期间并发产生的浮动垃圾所占的空间，堆中剩余可腾挪的空间就越来越小了。目前唯一的办法就是尽可能地增加堆容量大小，获得更多喘息的时间。但是若要从根本上提升 ZGC 能够应对的对象分配速率，还是需要引入分代收集，让新生对象都在一个专门的区域中创建，然后专门针对这个区域进行更频繁、更快的收集。Azul 的 C4 收集器实现了分代收集后，能够应对的对象分配速率就比不分代的 PGC 收集器提升了十倍之多。

三、类文件结构

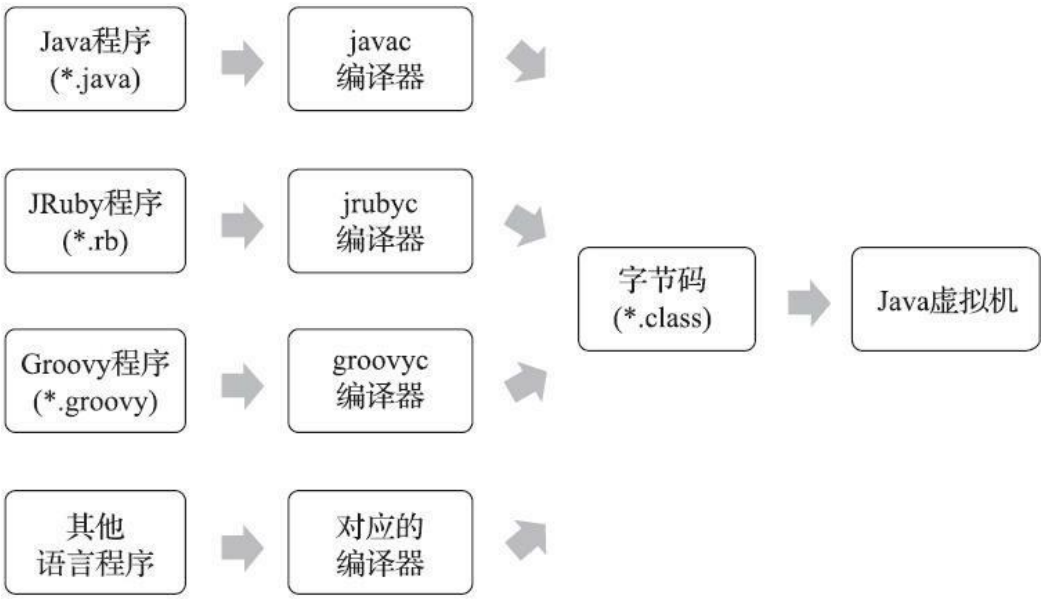
3.1 概述

在现在虚拟机以及建立在虚拟机上各种语言发展迅速的时代，直接翻译成机器码已不再是唯一，更多的程序语言选择了与操作系统和机器指令无关的、平台中立的格式作为程序编译后的存储格式。

本章主要是学习 Class 文件存储格式的具体细节。

3.2 无关性的基石

实现语言无关性的基础仍然是虚拟机和字节码存储格式。Java 虚拟机不与包括 Java 语言在内的任何程序语言绑定，它只与“Class 文件”这种特定的二进制文件格式所关联，Class 文件中包含了 Java 虚拟机指令集、符号表以及若干其他辅助信息。基于安全方面的考虑，《Java 虚拟机规范》中要求在 Class 文件必须应用许多强制性的语法和结构化约束，但图灵完备的字节码格式，保证了任意一门功能性语言都可以表示为一个能被 Java 虚拟机所接受的有效的 Class 文件。作为一个通用的、与机器无关的执行平台，任何其他语言的实现者都可以将 Java 虚拟机作为他们语言的运行基础，以 Class 文件作为他们产品的交付媒介。例如，使用 Java 编译器可以把 Java 代码编译为存储字节码的 Class 文件，使用 JRuby 等其他语言的编译器一样可以把它们的源程序代码编译成 Class 文件。虚拟机丝毫不关心 Class 的来源是什么语言，它与程序语言之间的关系如图所示。



3.3 Class 类文件的结构

Java 技术能够一直保持着非常良好的向后兼容性，Class 文件结构的稳定功不可没，任何一门程序语言能够获得商业上的成功，都不可能去做升级版本后，旧版本编译的产品就不再能够运行这种事 情。本章所讲述的关于 Class 文件结构的内容，绝大部分都是在第一版的《Java 虚拟机规范》（1997 年发布，对应于 JDK 1.2 时代的 Java 虚拟机）中就已经定义好的，内容虽然古老，但时至今日，Java 发展经历了十余个大版本、无数小更新，那时定义的 Class 文件格式的各项细节几乎没有出现任何改变。尽管不同版本的《Java 虚拟机规范》对 Class 文件格式进行了几次更新，但基本上只是在原有结构基础上新增内容、扩充功能，并未对已定义的内容做出修改。

注意：任何一个 Class 文件都对应着唯一的一个类或接口的定义信息（有反例，除了 package-info.class 等描述性的），但是反过来说，类或接口并不一定都得定义在文件里（譬如类或接口也可以动态生成，直接送入类加载器中）。

Class 文件是一组以 8 个字节为基础单位的二进制流，各个数据项目严格按照顺序紧凑地排列在文件之中，中间没有添加任何分隔符，这使得整个 Class 文件中存储的内容几乎全部是程序运行的必要数据，没有空隙存在。当遇到需要占用 8 个字节以上空间的数据项时，则会按照高位在前的方式分割成若干个 8 个字节进行存储。

根据《Java 虚拟机规范》的规定，Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：“无符号数”和“表”。后面的解析都要以这两种数据类型为基础。

无符号数属于基本的数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节和 8 个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照 UTF-8 编码构成字符串值。

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，为了便于区分，所有表的命名都习惯性地以“_info”结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上也可以视作是一张表，这张表由图所示的数据项按严格顺序排列构成。

类　型	名　称	数　量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

3.3.1 魔数与 Class 文件版本

每个 Class 文件的头 4 个字节被称为魔数（Magic Number），它的唯一作用是确定这个文件是否为一个能被虚拟机接受的 Class 文件。不仅是 Class 文件，很多文件格式标准中都有使用魔数来进行身份识别的习惯，譬如图片格式，如 GIF 或者 JPEG 等在文件头中都存有魔数。使用魔数而不是扩展名来进行识别主要是基于安全考虑，因为文件扩展名可以随意改动。文件格式的制定者可以自由地选择魔数 值，只要这个魔数值还没有被广泛采用过而且不会引起混淆。Class 文件的魔数取得很有“浪漫气息”， 值为 0xCAFEBABE（咖啡宝贝？）。

紧接着魔数的 4 个字节存储的是 Class 文件的版本号：第 5 和第 6 个字节是次版本号（Minor Version），第 7 和第 8 个字节是主版本号（M ajor Version）。Java 的版本号是从 45 开始的，JDK 1.1 之后的每个 JDK 大版本发布主版本号向上加 1（JDK 1.0～1.1 使用了 45.0～45.3 的版本号），高版本的 JDK 能向下兼容以前版本的 Class 文件，但不能运行以后版本的 Class 文件，因为《Java 虚拟机规范》在 Class 文件校验部分明确要求了即使文件格式并未发生任何变化，虚拟机也必须拒绝执行超过其版本号的 Class 文件。

3.3.2 常量池

紧接着主、次版本号之后的是常量池入口，常量池可以比喻为 Class 文件里的资源仓库，它是 Class 文件结构中与其他项目关联最多的数据，通常也是占用 Class 文件空间最大的数据项目之一，另外，它还是在 Class 文件中第一个出现的表类型数据项目。

常量池中主要存放两大类常量：字面量（Literal）和符号引用（Symbolic References）。字面量比较接近于 Java 语言层面的常量概念，如文本字符串、被声明为 final 的常量值等。而符号引用则属于编译原理方面的概念，主要包括下面几类常量：

1）被模块导出或者开放的包（Package）
2）类和接口的全限定名（Fully Qualified Name）
3）字段的名称和描述符（Descriptor）
4）方法的名称和描述符
5）方法句柄和方法类型（Method Handle、Method Type、Invoke Dynamic）
6）动态调用点和动态常量（Dynamically-Computed Call Site、Dynamically-Computed Constant）。

Java 代码在进行 Javac 编译的时候，并不像 C 和 C++那样有“连接”这一步骤，而是在虚拟机加载 Class 文件的时候进行动态连接。也就是说，在 Class 文件中不会保存各个方法、字段最终在内存中的布局信息，这些字段、方法的符号引用不经过虚拟机在运行期转换的话是无法得到真正的内存入口地址，也就无法直接被虚拟机使用的。当虚拟机做类加载时，将会从常量池获得对应的符号引用，再在类创建时或运行时解析、翻译到具体的内存地址之中。

关于常量池表的具体结构描述，这里先不做总结，内容比较繁琐，后面如果出现有必要总结就在此补上。

3.3.3 访问标志

在常量池结束之后，紧接着的 2 个字节代表访问标志（access_flags），这个标志用于识别一些类或者接口层次的访问信息，包括：这个 Class 是类还是接口；是否定义为 public 类型；是否定义为 abstract 类型；如果是类的话，是否被声明为 final；等等。

标 志 名 称	标志值	含　义
ACC_PUBLIC	0x0001	是否为 public 类型
ACC_FINAL	0x0010	是否被声明为 final，只有类可设置
ACC_SUPER	0x0020	是否允许使用 invokespecial 字节码指令的新语义，invokespecial 指令的语义在 JDK 1.0.2 发生过改变，为了区别这条指令使用哪种语义，JDK 1.0.2 之后编译出来的类的这个标志都必须为真
ACC_INTERFACE	0x0200	标识这是一个接口
ACC_ABSTRACT	0x0400	是否为 abstract 类型，对于接口或者抽象类来说，此标志值为真，其他类型值为假
ACC_SYNTHETIC	0x1000	标识这个类并非由用户代码产生的
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举
ACC_MODULE	0x8000	标识这是一个模块

access_flags 中一共有 16 个标志位可以使用，当前只定义了其中 9 个，没有使用到的标志位要求一律为零。

3.3.4 类索引、父类索引与接口索引集合

类索引（this_class）和父类索引（super_class）都是一个 u2 类型的数据，而接口索引集合（interfaces）是一组 u2 类型的数据的集合，Class 文件中由这三项数据来确定该类型的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名。由于 Java 语言不允许多重继承，所以父类索引只有一个，除了 java.lang.Object 之外，所有的 Java 类都有父类，因此除了 java.lang.Object 外，所有 Java 类的父类索引都不为 0。接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按 implements 关键字（如果这个 Class 文件表示的是一个接口，则应当是 extends 关键字）后的接口顺序从左到右排列在接口索引集合中。

3.3.5 字段表集合

字段表（field_info）用于描述接口或者类中声明的变量。Java 语言中的“字段”（Field）包括类级变量以及实例级变量，但不包括在方法内部声明的局部变量。字段可以包括的修饰符有字段的作用域（public、private、protected 修饰符）、是实例变量还是类变量（static 修饰符）、可变性（final）、并发可见性（volatile 修饰符，是否强制从主内存读写）、可否被序列化（transient 修饰符）、字段数据类型（基本类型、对象、数组）、字段名称。上述这些信息中，各个修饰符都是布尔值，要么有某个修饰符，要么没有，很适合使用标志位来表示。而字段叫做什么名字、字段被定义为什么数据类型，这些都是无法固定的，只能引用常量池中的常量来描述。

字段修饰符放在 access_flags 项目中，跟随 access_flags 标志的是两项索引值：name_index 和 descriptor_index。它们都是对常量池项的引用，分别代表着字段的简单名称以及字段和方法的描述符。

全限定名和简单名称：“org/fenixsoft/clazz/TestClass”是这个类的全限定名，仅仅是把类全名中的“.”替换成了“/”而已，为了使连续的多个全限定名之间不产生混淆，在使用时最后一般会加入一个“；”号表示全限定名结束。简单名称则就是指没有类型和参数修饰的方法或者字段名称，这个类中的 inc()方法和 m 字段的简单名称分别就是“inc”和“m”。

方法和字段的描述符：描述符的作用是用来描述字段的数据类型、方法的参数列表（包括数量、类型以及顺序）和返回值。根据描述符规则，基本数据类型（byte、char、double、float、int、long、short、boolean）以及代表无返回值的 void 类型都用一个大写字符来表示，而对象类型则用字符 L 加对象的全限定名来表示。例如“java.lang.String[][]”类型的二维数组将被记录成“[[Ljava/lang/String；”，一个整型数组“int[]”将被记录成“[I”。

字段表集合中不会列出从父类或者父接口中继承而来的字段，但有可能出现原本 Java 代码之中不存在的字段，譬如在内部类中为了保持对外部类的访问性，编译器就会自动添加指向外部类实例的字段。另外，在 Java 语言中字段是无法重载的，两个字段的数据类型、修饰符不管是否相同，都必须使用不一样的名称，但是对于 Class 文件格式来讲，只要两个字段的描述符不是完全相同，那字段重名就是合法的。

3.3.6 方法表集合

Class 文件存储格式中对方法的描述与对字段的描述采用了几乎完全一致的方式，方法表的结构如同字段表一样，依次包括访问标志（access_flags）、名称索引（name_index）、描述符索引（descriptor_index）、属性表集合（attributes）几项。

方法里的 Java 代码，经过 Javac 编译器编译成字节码指令之 后，存放在方法属性表集合中一个名为“Code”的属性里面。

与字段表集合相对应地，如果父类方法在子类中没有被重写（Override），方法表集合中就不会出现来自父类的方法信息。但同样地，有可能会出现由编译器自动添加的方法，最常见的便是类构造器“<clinit>()”方法和实例构造器“<init>()”方法。

在 Java 语言中，要重载（Overload）一个方法，除了要与原方法具有相同的简单名称之外，还要求必须拥有一个与原方法不同的特征签名。特征签名是指一个方法中各个参数在常量池中的字段符号引用的集合，也正是因为返回值不会包含在特征签名之中，所以 Java 语言里面是无法仅仅依靠返回值的不同来对一个已有方法进行重载的。但是在 Class 文件格式之中，特征签名的范围明显要更大一些， 只要描述符不是完全一致的两个方法就可以共存。也就是说，如果两个方法有相同的名称和特征签 名，但返回值不同，那么也是可以合法共存于同一个 Class 文件中的。

3.3.7 属性表集合

与 Class 文件中其他的数据项目要求严格的顺序、长度和内容不同，属性表集合的限制稍微宽松一些，不再要求各个属性表具有严格顺序，并且《Java 虚拟机规范》允许只要不与已有属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息，Java 虚拟机运行时会忽略掉它不认识的属性。为了能正确解析 Class 文件，《Java 虚拟机规范》最初只预定义了 9 项所有 Java 虚拟机实现都应当能识别的属性，而在最新的《Java 虚拟机规范》的 Java SE 12 版本中，预定义属性已经增加到 29 项（书上有图，感觉不是重点，就不记录了）。

1）code 属性：Java 程序方法体里面的代码经过 Javac 编译器处理之后，最终变为字节码指令存储在 Code 属性内。Code 属性出现在方法表的属性集合之中，但并非所有的方法表都必须存在这个属性，譬如接口或者抽象类中的方法就不存在 Code 属性。

Code 属性是 Class 文件中最重要的一个属性，如果把一个 Java 程序中的信息分为代码（Code，方法体里面的 Java 代码）和元数据（Metadata，包括类、字段、方法定义及其他信息）两部分，那么在整个 Class 文件里，Code 属性用于描述代码，所有的其他数据项目都用于描述元数据。

```
// 原始Java代码
public class TestClass
    { private int m;

    public int inc()
        { return m + 1;
    }
}

C:\>javap -verbose TestClass
// 常量表部分的输出见代码清单6-1，因版面原因这里省略掉
{
public org.fenixsoft.clazz.TestClass();
    Code:
        Stack=1, Locals=1, Args_size=1
        0:   aload_0
        1:   invokespecial   #10; //Method java/lang/Object."<init>":()V
        4:   return
    LineNumberTable:
        line 3: 0

    LocalVariableTable:
        Start  Length  Slot  Name    Signature
        0      5       0     this    Lorg/fenixsoft/clazz/TestClass;

public int inc();
    Code:
        Stack=2, Locals=1, Args_size=1
        0:   aload_0
```

2）运行时注解属性：早在 JDK 5 时期，Java 语言的语法进行了多项增强，其中之一是提供了对注解（Annotation）的支持。为了存储源码中注解信息，Class 文件同步增加了 RuntimeVisibleAnnotations、RuntimeInvisibleAnnotations 、

RuntimeVisibleParameterAnnotations 和 RuntimeInvisibleParameter- Annotations 四个属性。到了 JDK 8 时期，进一步加强了 Java 语言的注解使用范围，又新增类型注解（JSR 308），所以 Class 文件中也同步增加了 RuntimeVisibleTypeAnnotations 和 RuntimeInvisibleTypeAnnotations 两个属性。由于这六个属性不论结构还是功能都比较雷同，因此我们把它们合并到一起，以 RuntimeVisibleAnnotations 为代表进行介绍。

RuntimeVisibleAnnotations 是一个变长属性，它记录了类、字段或方法的声明上记录运行时可见注解，当我们使用反射 API 来获取类、字段或方法上的注解时，返回值就是通过这个属性来取到的。

类　　型	名　　称	数　　量
u2	attribute_name_index	1
u4	attribute_length	1
u2	num_annotations	1
annotation	annotations	num_annotations

num_annotations 是 annotations 数组的计数器，annotations 中每个元素都代表了一个运行时可见的注解，注解在 Class 文件中以 annotation 结构来存储。

类　　型	名　　称	数　　量
u2	type_index	1
u2	num_element_value_pairs	1
element_value_pair	element_value_pairs	num_element_value_pairs

type_index 是一个指向常量池 CONSTANT_Utf8_info 常量的索引值，该常量应以字段描述符的形式表示一个注解。num_element_value_pairs 是 element_value_pairs 数组的计数器，element_value_pairs 中每个元素都是一个键值对，代表该注解的参数和值。

（其实还有很多属性没有总结，这里列出两种，其他的想了解请参考书）

3.4 字节码指令简介

这一节基本上是字节码的指令做字典做说明总结，就不再过多总结一遍。

四、虚拟机类加载机制

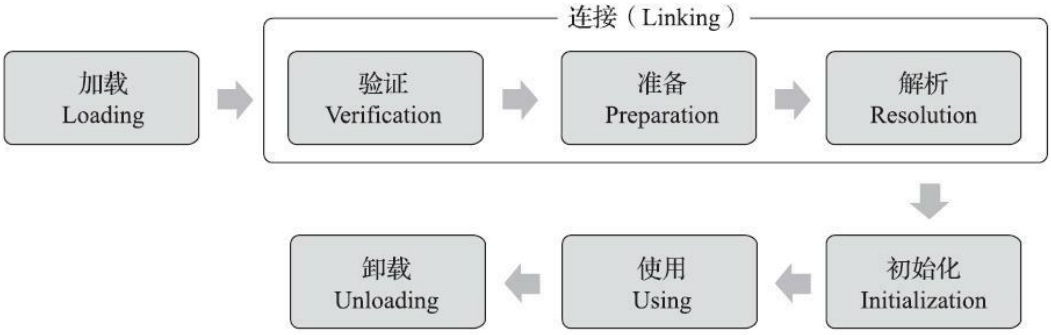
4.1 概述

上一章学习 Class 文件格式，本章主要是学习虚拟机如何加载 Class 文件，Class 文件中的信息进入虚拟机后会发生什么变化。

Java 虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这个过程被称作虚拟机的类加载机制。与那些在编译时需要进行连接的语言不同，在 Java 语言里面，类型的加载、连接和初始化过程都是在程序运行期间完成的，这种策略让 Java 语言进行提前编译会面临额外的困难，也会让类加载时稍微增加一些性能开销， 但是却为 Java 应用提供了极高的扩展性和灵活性，Java 天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。例如，编写一个面向接口的应用程序，可以等到运行时再指定其实际的实现类，用户可以通过 Java 预置的或自定义类加载器，让某个本地的应用程序在运行时从网络或其他地方上加载一个二进制流作为其程序代码的一部分。这种动态组装应用的方式目前已广泛应用于 Java 程序之中，从最基础的 Applet、JSP 到相对复杂的 OSGi 技术，都依赖着 Java 语言运行期类加载才得以诞生。

4.2 类加载的时机

一个类型从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期将会经历加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）七个阶段，其中验证、准备、解析三个部分统称为连接（Linking）。这七个阶段的发生顺序如图所示：



加载、验证、准备、初始化和卸载这五个阶段的顺序是确定的，类型的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始， 这是为了支持 Java 语言的运行时绑定特性（也称为动态绑定或晚期绑定）。这些阶段通常都是互相交叉地混合进行的，会在一个阶段执行的过程中调用、激活另一个阶段。

对于加载阶段，《Java 虚拟机规范》并没有进行强行约束。但是对于初始化阶段，《Java 虚拟机规范》严格规定了有且只有六种情况必须立即对类进行“初始化”：

1）遇到 new、getstatic、putstatic 或 invokestatic 这四条字节码指令时，如果类型没有进行过初始化，则需要先触发其初始化阶段。

a. 使用 new 关键字实例化对象时。Obj o = new Obj();
b. 读取或者设置一个类型的静态字段的时候（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）。Obj.num=1;
c. 调用一个类的静态方法时。Obj.func();

2）使用 java.lang.reflect 包的方法对类型进行反射调用的时候，如果类型没有进行过初始化，则需要先触发其初始化。

3）当初始化类的时候，如果发现父类没有初始化，则需要先触发父类的初始化。

4）当虚拟机启动的时候，用户需要指定一个要执行的主类（包含 main()方法的那个类），虚拟机会先初始化这个类。

5）当使用 JDK 7 新加入的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果为 REF_getStatic、REF_putStatic、REF_invokeStatic、REF_newInvokeSpecial 四种类型的方法句柄，并且这个方法句柄对应的类没有进行过初始化，则需要先触发其初始化。

6）当一个接口中定义了 JDK 8 新加入的默认方法（被 default 关键字修饰的接口方法）时，如果有这个接口的实现类发生了初始化，那该接口要在其之前被

初始化。（关于接口的初始化，在实际测试中存在问题，先不记为重点）

上面六种场景的行为称为对一个类型进行主动引用。

除了上面六种行为，其他行为都称为被动引用。例如：

```
package org.fenixsoft.classloading;

/**
 * 被动使用类字段演示一：
 * 通过子类引用父类的静态字段，不会导致子类初始化
 **/
public class SuperClass {

    static {
        System.out.println("SuperClass init!");
    }

    public static int value = 123;
}

public class SubClass extends SuperClass {

    static {
        System.out.println("SubClass init!");
    }
}

/**
 * 非主动使用类字段演示
 **/
public class NotInitialization {

    public static void main(String[] args) {
        System.out.println(SubClass.value);
    }

}
```

上述代码运行之后，只会输出“SuperClass init！”，而不会输出“SubClass init！”。对于静态字段， 只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。至于是否要触发子类的加载和验证阶段，在《Java 虚拟机规范》中并未明确规定，所以这点取决于虚拟机的具体实现。对于 HotSpot 虚拟机来说，可通过-XX：+TraceClassLoading 参数观察到此操作是会导致子类加载的。

```
package org.fenixsoft.classloading;

/**
 * 被动使用类字段演示二：
 * 通过数组定义来引用类，不会触发此类的初始化
 **/
public class NotInitialization {

    public static void main(String[] args) {
        SuperClass[] sca = new SuperClass[10];
    }

}
```

上面代码运行后，运行之后发现没有输出“SuperClass init！”，说明并没有触发类 org.fenixsoft.classloading.SuperClass 的初始化阶段。但是这段代码里面触发了另一个名为“[Lorg.fenixsoft.classloading.SuperClass”的类的初始化阶段，对于用户代码来说，这并不是一个合法的类型名称，它是一个由虚拟机自动生成的、直接继承于 java.lang.Object 的子类，创建动作由字节码指令 newarray 触发。

```
package org.fenixsoft.classloading;

/**
 * 被动使用类字段演示三:
 * 常量在编译阶段会存入调用类的常量池中，本质上没有直接引用到定义常量的类，因此不会触发定义常量的
   类的初始化
 **/
public class ConstClass {

    static {
        System.out.println("ConstClass init!");
    }

    public static final String HELLOWORLD = "hello world";
}

/**
 * 非主动使用类字段演示
 **/
public class NotInitialization {

    public static void main(String[] args) {
        System.out.println(ConstClass.HELLOWORLD);
    }
}
```

上面代码运行后，也没有输出“ConstClass init！”，这是因为虽然确实引用了 ConstClass 类中的 HELLOWORLD 字段，但是该字段由于 final 的关系，在编译阶段就将常量值直接存入 NotInitialization 类的常量池中，以后 NotInitialization 对常量 ConstClass.HELLOWORLD 的引用，实际都被转化为 NotInitialization 类对自身常量池的引用了。也就是说，实际上 NotInitialization 的 Class 文件之中并没有 ConstClass 类的符号引用入口，这两个类在编译成 Class 文件后就已不存在任何联系了。

接口的加载过程与类加载过程稍有不同，针对接口需要做一些特殊说明：接口也有初始化过程，这点与类是一致的，上面的代码都是用静态语句块“static{}”来输出初始化信息的，而接口中不能使 用“static{}”语句块，但编译器仍然会为接口生成“<clinit>()”类构造器，用于初始化接口中所定义的成员变量。（这里我没有测试出有生成<clinit>()，后面查询资料后再补充关于接口的初始化）

类与接口初始化的区别在于，一个类在初始化时，要求其父类全部初始化过了，然而对于接口来说，一个接口在初始化时，并不要求其父接口全部初始化，只用用到父接口（如引用接口中定义的常量）才会初始化。

4.3 类加载的过程

4.3.1 加载

“加载”（Loading）阶段是整个“类加载”（Class Loading）过程中的一个阶段。

在加载阶段，Java 虚拟机需要完成以下三件事情：
1）通过一个类的全限定名来获取定义此类的二进制字节流。
2）将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
3）在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。（这个 Class 对象用来实例化其他对象）

《Java 虚拟机规范》对这三点要求其实并不是特别具体，留给虚拟机实现与 Java 应用的灵活度都是相当大的。例如“通过一个类的全限定名来获取定义此类的二进制字节流”这条规则，它并没有指明二进制字节流必须得从某个 Class 文件中获取，确切地说是根本没有指明要从哪里获取、如何获取。仅仅这一点空隙，Java 虚拟机的使用者们就可以在加载阶段搭构建出一个相当开放广阔的舞台，Java 发展历程中，充满创造力的开发人员则在这个舞台上玩出了各种花样，许多举足轻重的 Java 技术都建立在这一基础之上，例如：

1）从 ZIP 压缩包中读取，这很常见，最终成为日后 JAR、EAR、WAR 格式的基础。
2）从网络中获取，这种场景最典型的应用就是 Web Apple。
3）由其他文件生成，典型场景是 JSP 应用，由 JSP 文件生成对应的 Class 文件。
4）从数据库中读取，这种场景相对少见些，例如有些中间件服务器（如 SAP Netweaver）可以选择把程序安装到数据库中来完成程序代码在集群间的分发。
5）可以从加密文件中获取，这是典型的防 Class 文件被反编译的保护措施，通过加载时解密 Class 文件来保障程序运行逻辑不被窥探。

相较于其他类加载阶段，非数组类型的加载阶段（准确的说，是加载阶段中获取类的二进制流的动作）是开发人员可控性最强的阶段。加载阶段既可以使用 Java 虚拟机里内置的引导类加载器来完成，也可以由用户自定义的类加载器去完成，开发人员通过定义自己的类加载器去控制字节流的获取方式（重写一个类加载器的 findClass()或 loadClass()方法），实现根据自己的想法来赋予应用程序获取运行代码的动态性。

对于数组类而言，情况就有所不同，数组类本身不通过类加载器创建，它是由 Java 虚拟机直接在内存中动态构造出来的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型（Element Type，指的是数组去掉所有维度的类型）最终还是要靠类加载器来完成加载，一个数组类（下面简称为 C）创建过程遵循以下规则：

1）如果数组的组件类型（Component Type，指的是数组去掉一个维度的类型，注意和前面的元素类型区分开来，Obj[]的组件类型是 Obj 引用类型）是引用类型，那就递归采用双亲委派过程去加载这个组件类型，数组 C 将被标识在加载该组件类型的类加载器的类名称空间上。

2）如果数组的组件类型不是引用类型（例如 int[]数组的组件类型为 int），Java 虚拟机将会把数组 C 标记为与引导类加载器关联。

3）数组类的可访问性与它的组件类型的可访问性一致，如果组件类型不是引用类型，它的数组类的可访问性将默认为 public，可被所有的类和接口访问到。

加载阶段结束后，Java 虚拟机外部的二进制字节流就按照虚拟机所设定的格式存储在方法区之中。方法区中的数据存储格式完全由虚拟机实现自定义，《Java 虚拟机规范》并未规定此区域的具体数据结构。类型数据妥善放到方法区后，会在 Java 堆内存中实例化一个 java.lang.Class 类的对象，这个对象将作为程序访问方法区中的类型数据的外部接口。

加载阶段与连接阶段的部分动作（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在加载阶段之中进行的动作，仍然属于连接阶段的一部分，这两个阶段的开始时间仍然保持着固定的先后顺序（这个顺序仅仅是开始顺序）。

4.3.2 验证

验证是连接阶段的第一步，这一阶段的目的是确保 Class 文件的字节流中包含的信息符合《Java 虚拟机规范》的全部约束要求，保证这些信息被当作代码运行后不会危害虚拟机自身的安全。

Java 语言本身是相对安全的编程语言（起码对于 C/C++来说是相对安全的），使用纯粹的 Java 代码无法做到诸如访问数组边界以外的数据、将一个对象转型为它并未实现的类型、跳转到不存在的代码行之类的事情，如果尝试这样去做了，编译器会毫不留情地抛出异常、拒绝编译。但前面也曾说过， Class 文件并不一定只能由 Java 源码编译而来，它可以使用包括靠键盘 0 和 1 直接在二进制编辑器中敲出 Class 文件在内的任何途径产生。上述 Java 代码无法做到的事情在字节码层面上都是可以实现的，至少语义上是可以表达出来的。Java 虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有错误或有恶意企图的字节码流而导致整个系统受攻击甚至崩溃，所以验证字节码是 Java 虚拟机保护自身的一项必要措施。

验证阶段是非常重要的，这个阶段是否严谨，直接决定了 Java 虚拟机是否能承受恶意代码的攻 击，从代码量和耗费的执行性能的角度上讲，验证阶段的工作量在虚拟机的类加载过程中占了相当大的比重。但是《Java 虚拟机规范》的早期版本（第 1、2 版）对这个阶段的检验指导是相当模糊和笼统的，规范中仅列举了一些对 Class 文件格式的静态和结构化的约束，要求虚拟机验证到输入的字节流如不符合 Class 文件格式的约束，就应当抛出一个 java.lang.VerifyError 异常或其子类异常，但具体应当检查哪些内容、如何检查、何时进行检查等，都没有足够具体的要求和明确的说明。直到 2011 年《Java 虚拟机规范（Java SE 7 版）》出版，规范中大幅增加了验证过程的描述（篇幅从不到 10 页增加到 130 页），这时验证阶段的约束和验证规则才变得具体起来。受篇幅所限，本书中无法逐条规则去讲解， 但从整体上看，验证阶段大致上会完成下面四个阶段的检验动作：文件格式验证、元数据验证、字节码验证和符号引用验证。

（1）文件格式验证：验证字节流是否符合 Class 文件规范，并能被虚拟机受理。包括是否以魔数开头，主次版本号是否在虚拟机接受范围内等等。注意，文件格式验证是读取操作字节流来进行验证。过了这个阶段，就进入 Java 虚拟机内存的方法区中进行存储，因此后面三个阶段都是基于方法区的存储结构上进行的。

（2）元数据验证：对字节码描述的信息进行语义分析，保证其描述的信息符合《Java 语言规范》的要求。包括这个类是否有父类，这个类的父类是否继承了不允许继承的类（final）等等。

（3）字节码验证：该阶段是验证过程中最复杂的阶段，主要目的是通过数据流分析和控制流分析，确定程序语义是合法的、符合逻辑的。这阶段要对类的方法体（Class 文件中的 code 属性）进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。注意：由于数据流分析和控制流分析的高度复杂性，Java 虚拟机的设计团队为了避免过多的执行时间消耗在字节码验证阶段中，在 JDK 6 之后的 Javac 编译器和 Java 虚拟机里进行了一项联合优化，把尽可

能多的校验辅助措施挪到 Javac 编译器里进行。

（4）符号引用验证：该阶段的校验行为发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。符号引用验证可以看作是对类自身以外（常量池中的各种符号引用）的各类信息进行匹配性校验。包括符号引用中通过字符串描述的全限定名是否能找到对应的类，在指定类中是否存在符合方法的字段描述符及简单名称所描述的方法和字段等等。

验证阶段对于虚拟机的类加载机制来说，是一个非常重要的、但却不是必须要执行的阶段，因为验证阶段只有通过或者不通过的差别，只要通过了验证，其后就对程序运行期没有任何影响了。如果程序运行的全部代码（包括自己编写的、第三方包中的、从外部加载的、动态生成的等所有代码）都已经被反复使用和验证过，在生产环境的实施阶段就可以考虑使用-Xverify：none 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

4.3.3 准备

准备阶段是正式为类中定义的变量（即静态变量，被 static 修饰的变量）分配内存并设置类变量初始值（零值）的阶段，从概念上讲，这些变量所使用的内存都应当在方法区中进行分配，但必须注意到方法区本身是一个逻辑上的区域，在 JDK 7 及之前，HotSpot 使用永久代来实现方法区时，实现是完全符合这种逻辑概念的；而在 JDK 8 及之后，类变量则会随着 Class 对象一起存放在 Java 堆中（其中还有字符串常量池，其他的都在元空间），这时候“类变量在方法区”就完全是一种对逻辑概念的表述了。

关于准备阶段，首先是这时候进行内存分配的仅包括类变量，而不包括实例变量，实例变量将会在对象实例化（这个阶段在类加载阶段的初试化后了）时随着对象一起分配在 Java 堆中。

4.3.4 解析

解析阶段是 Java 虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定是已经加载到虚拟机内存当中的内容。各种虚拟机实现的内存布局可以各不相同， 但是它们能接受的符号引用必须都是一致的，因为符号引用的字面量形式明确定义在《Java 虚拟机规范》的 Class 文件格式中。

直接引用（Direct References）：直接引用是可以直接指向目标的指针、相对偏移量或者是一个能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局直接相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经在虚拟机的内存中存在。

对同一个符号引用进行多次解析请求是很常见的事情，除 invokedynamic 指令以外，虚拟机实现可以对第一次解析的结果进行缓存，譬如在运行时直接引用常量池中的记录，并把常量标识为已解析状态，从而避免解析动作重复进行。无论是否真正执行了多次解析动作，Java 虚拟机都需要保证的是在同一个实体中，如果一个符号引用之前已经被成功解析过，那么后续的引用解析请求就应当一直能够成功；同样地，如果第一次解析失败了，其他指令对这个符号的解析请求也应该收到相同的异常，哪怕这个请求的符号在后来已成功加载进 Java 虚拟机内存之中。

invokedynamic 指令：用于在运行时动态解析出调用点限定符所引用的方法。并执行该方法。前面四条调用指令的分派逻辑都固化在 Java 虚拟机内部，用户无法改变，而 invokedynamic 指令的分派逻辑是由用户所设定的引导方法决定的。

当碰到某个前面已经由 invokedynamic 指令触发过解析的符号引用时，并不意味着这个解析结果对于其他 invokedynamic 指令也同样生效（即不会有解析结果的缓存）。因为 invokedynamic 指令的目的本来就是用于动态语言支持，它对应的引用称为“动态调用点限定符（Dynamically-Computed Call Site Specifier）”，这里“动态”的含义是指必须等到程序实际运行到这条指令时，解析动作才能进行。相对地，其余可触发解析的指令都是“静态”的，可以在刚刚完成加载阶段，还没有开始执行代码时就提前进行解析。

解析动作主要针对于类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符这 7 类符号引用进行。

（解析过程就不总结了，比较繁琐）

4.3.5 初始化

类的初始化阶段是类加载过程的最后一个步骤，之前介绍的几个类加载的动作里，除了在加载阶段用户应用程序可以通过自定义类加载器的方式局部参与外，其余动作都完全由 Java 虚拟机来主导控制。直到初始化阶段，Java 虚拟机才真正开始执行类中编写的 Java 程序代码，将主导权移交给应用程序。

进行准备阶段时，变量已经赋过一次系统要求的初始零值，而在初始化阶段，则会根据程序员通过程序编码制定的主观计划去初始化类变量和其他资源。我们也可以从另外一种更直接的形式来表达：初始化阶段就是执行类构造器<clinit>()方法的过程。

<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。如图：

```
public class Test {
    static {
        i = 0;  //  给变量复制可以正常编译通过
        System.out.print(i);  // 这句编译器会提示"非法向前引用"
    }
    static int i = 1;
}
```

<clinit>()方法与类的构造函数（即在虚拟机视角中的实例构造器<init>()方法）不同，它不需要显式地调用父类构造器，Java 虚拟机会保证在子类的<clinit>()方法执行前，父类的<clinit>()方法已经执行完毕。因此在 Java 虚拟机中第一个被执行的<clinit>()方法的类型肯定是 java.lang.Object。如图打印结果是 2：

```
static class Parent {
    public static int A = 1;
    static {
        A = 2;
    }
}

static class Sub extends Parent {
    public static int B = A;
}


public static void main(String[] args) {
    System.out.println(Sub.B);
}
```

<clinit>()方法对于类或接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>()方法。

接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成<clinit>()方法。但接口与类不同的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法， 因为只有当父接口中定义的变量被使用

时，父接口才会被初始化。此外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。（这里没测试出有 clinit 方法出现，所以先不加重点）

Java 虚拟机必须保证一个类的<clinit>()方法在多线程环境中被正确地加锁同步，如果多个线程同时去初始化一个类，那么只会有其中一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行完毕<clinit>()方法。如果在一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞（虽然其他进程阻塞了，但是如果执行<clinit>()方法的线程退出<clinit>()方法后，其他线程唤醒后不会再次进入<clinit>()方法。同一个类加载器下，一个类型只会被初始化一次），在实际应用中这种阻塞往往是很隐蔽的。

4.4 类加载器

Java 虚拟机设计团队有意把类加载阶段中的“通过一个类的全限定名来获取描述该类的二进制字节流”这个动作放到 Java 虚拟机外部去实现，以便让应用程序自己决定如何去获取所需的类。实现这个动作的代码被称为“类加载器”（Class Loader）。

类加载器可以说是 Java 语言的一项创新，它是早期 Java 语言能够快速流行的重要原因之一。类加载器最初是为了满足 Java Applet 的需求而设计出来的，在今天用在浏览器上的 Java Applet 技术基本上已经被淘汰[1]，但类加载器却在类层次划分、OSGi、程序热部署、代码加密等领域大放异彩，成为 Java 技术体系中一块重要的基石，可谓是失之桑榆，收之东隅。

4.4.1 类与类加载器

类加载器虽然只用于实现类的加载动作，但它在 Java 程序中起到的作用却远超类加载阶段。对于任意一个类，都必须由加载它的类加载器和这个类本身一起共同确立其在 Java 虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个 Class 文件，被同一个 Java 虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

这里所指的“相等”，包括代表类的 Class 对象的 equals()方法、isAssignableFrom()方法、isInstance() 方法的返回结果，也包括了使用 instanceof 关键字做对象所属关系判定等各种情况。如果没有注意到类加载器的影响，在某些情况下可能会产生具有迷惑性的结果：

```
public class ClassLoaderTest {

    public static void main(String[] args) throws Exception {

        ClassLoader myLoader = new ClassLoader() {
            @Override
            public Class<?> loadClass(String name) throws ClassNotFoundException {
                try {
                    String fileName = name.substring(name.lastIndexOf(".") + 1)+".class";
                    InputStream is = getClass().getResourceAsStream(fileName);
                    if (is == null) {
                        return super.loadClass(name);
                    }
                    byte[] b = new byte[is.available()];
                    is.read(b);
                    return defineClass(name, b, 0, b.length);
                } catch (IOException e) {
                    throw new ClassNotFoundException(name);
                }
            }
        };

        Object obj = myLoader.loadClass("org.fenixsoft.classloading.ClassLoaderTest").newInstance();

        System.out.println(obj.getClass());
        System.out.println(obj instanceof org.fenixsoft.classloading.ClassLoaderTest);
    }
}
```

运行结果：

```
class org.fenixsoft.classloading.ClassLoaderTest
false
```

根据书上的例子：false 的原因是，在同一个 Class 文件下用了两个不同的类加载器去加载，一个是应用程序类加载器，一个是自定义的 myLoader。

4.4.2 双亲委派模型

站在 Java 虚拟机的角度来看，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用 C++语言实现，是虚拟机自身的一部分；另外一种就是其他所有的类加载器，这些类加载器都由 Java 语言实现，独立存在于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。

站在 Java 开发人员的角度来看，类加载器就应当划分得更细致一些。

1）启动类加载器：

a. 这个类加载器负责加载存放在<JAVA_HOM E>\lib 目录，或者被-Xbootclasspath 参数所指定的路径中存放的，而且是 Java 虚拟机能够识别的（按照文件名识别，如 rt.jar、tools.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机的内存中。

b. 该加载器无法被 Java 程序直接使用（使用 C++实现）

c. 返回值为 null。

2）扩展类加载器：

a. 这个类加载器负责加载<JAVA_HOM E>\lib\ext 目录中，或者被 java.ext.dirs 系统变量所指定的路径中所有的类库。

b. JDK 的开发团队允许用户将具有通用性的类库放置在 ext 目录里以扩展 Java SE 的功能，在 JDK 9 之后，这种扩展机制被模块化带来的天然的扩展能力所取代。由于扩展类加载器是由 Java 代码实现的，开发者可以直接在程序中使用扩展类加载器来加载 Class 文件。

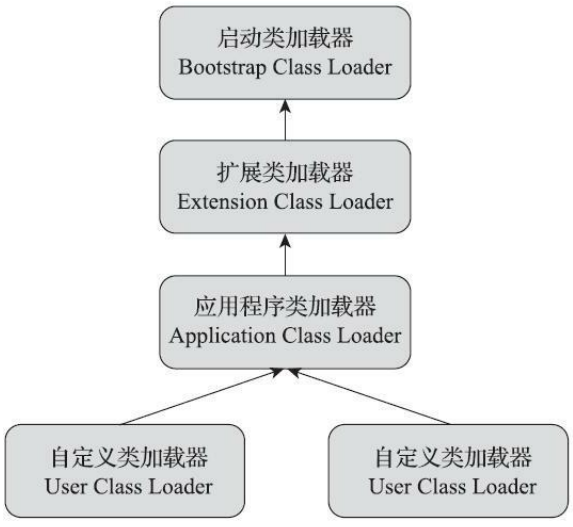
c. 返回值为 sun.misc.Launcher$ExtClassLoader

3）应用程序类加载器：

a. 这个类加载器负责加载用户类路径（ClassPath）上所有的类库。

b. 开发者同样可以直接在代码中使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

c. 返回值为 sun.misc.Launcher$AppClassLoader



如图所示的各个类加载器之间的层次关系被称为类加载器的“双亲委派

模型”。双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应有自己的父类加载器。不过这里类加载器之间的父子关系一般不是以继承（Inheritance）的关系来实现的，而是通常使用组合（Composition）关系来复用父加载器的代码。

双亲委派模型的工作过程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到最顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去完成加载。

使用双亲委派模型来组织类加载器之间的关系，一个显而易见的好处就是 Java 中的类随着它的类加载器一起具备了一种带有优先级的层次关系（保护基础核心类）。例如类 java.lang.Object，它存放在 rt.jar 之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都能够保证是同一个类。反之，如果没有使用双亲委派模型，都由各个类加载器自行去加载的话，如果用户自己也编写了一个名为 java.lang.Object 的类，并放在程序的 ClassPath 中，那系统中就会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无从保证，应用程序将会变得一片混乱。

（后面的破坏双亲委派模型不做总结，了解就行）

4.5 Java 模块化系统

在 JDK9 的时候引入了 Java 模块化系统，这个系统是对 Java 技术的一次重要升级，为了能够实现模块化的关键目标——可配置的封装隔离机制，Java 虚拟机对类加载架构也做出了相应的变动调整，才使模块化系统得以顺利地运作。

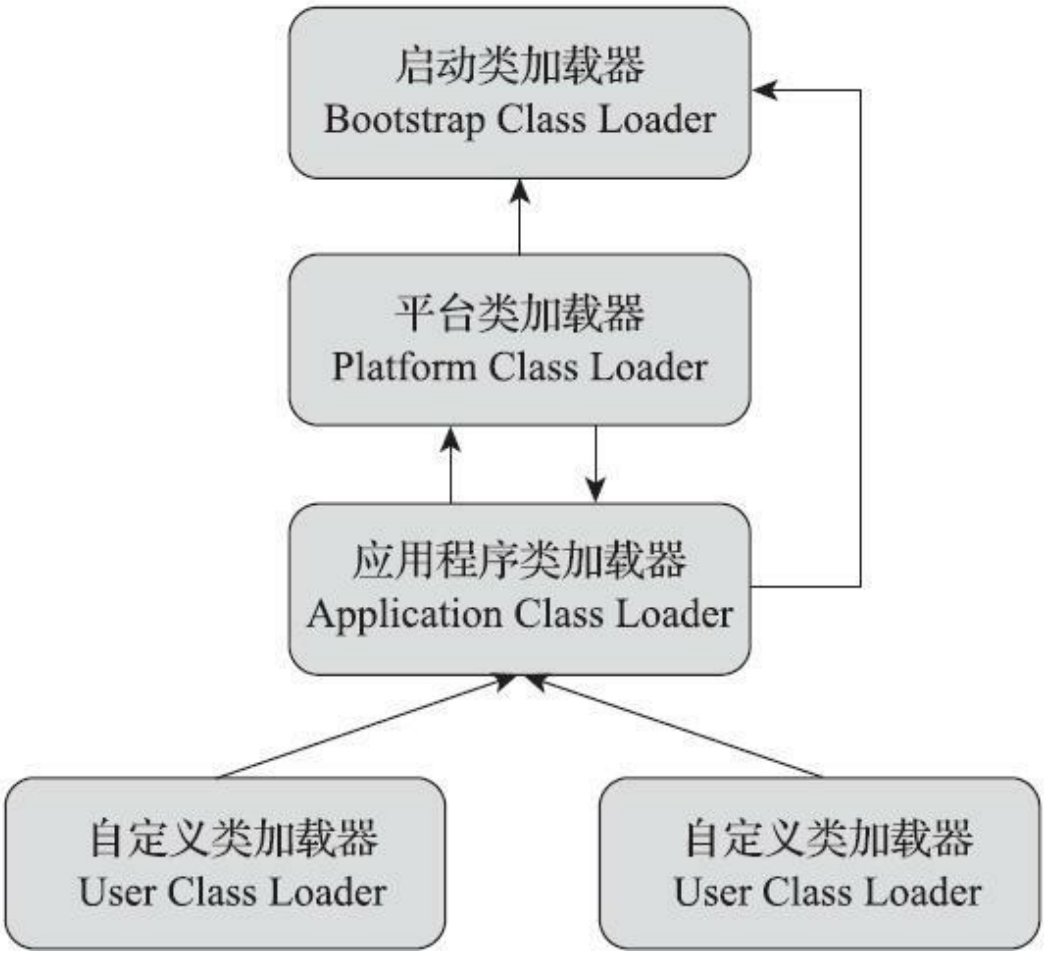
为了保证兼容性，JDK 9 并没有从根本上动摇从 JDK 1.2 以来运行了二十年之久的三层类加载器架构以及双亲委派模型。但是为了模块化系统的顺利施行，模块化下的类加载器仍然发生了一些应该被注意到变动，主要包括以下几个方面。

首先是扩展类加载器（Extension Class Loader）被平台类加载器（Platform Class Loader）取代。既然整个 JDK 都基于模块化进行构建（原来的 rt.jar 和 tools.jar 被拆分成数十个 JMOD 文件），其中的 Java 类库就已天然地满足了可扩展的需求，那自然无须再保留<JAVA_HOME>\lib\ext 目录，类似地，在新版的 JDK 中也取消了<JAVA_HOM E>\jre 目录，因为随时可以组合构建出程序运行所需的 JRE 来。

其次，平台类加载器和应用程序类加载器都不再派生自 java.net.URLClassLoader，如果有程序直接依赖了这种继承关系，或者依赖了 URLClassLoader 类的特定方法，那代码很可能会在 JDK 9 及更高版本的 JDK 中崩溃。现在启动类加载器、平台类加载器、应用程序类加载器全都继承于 jdk.internal.loader.BuiltinClassLoader，在 BuiltinClassLoader 中实现了新的模块化架构下类如何从模块中加载的逻辑，以及模块中资源可访问性的处理。

JDK9 之前

JDK9 之后



最后，JDK 9 中虽然仍然维持着三层类加载器和双亲委派的架构，但类加载的委派关系也发生了变动。当平台及应用程序类加载器收到类加载请求，在委派给父加载器加载前，要先判断该类是否能够归属到某一个系统模块中，如果可以找到这样的归属关系，就要优先委派给负责那个模块的加载器完成加载，也许这可以算是对双亲委派的第四次破坏。

五、虚拟机字节码执行引擎

5.1 概述

执行引擎是 Java 虚拟机核心的组成部分之一。“虚拟机”是一个相对于“物理机”的概念，这两种机器都有代码执行能力，其区别是物理机的执行引擎是直接建立在处理器、缓存、指令集和操作系统层面上的，而虚拟机的执行引擎则是由软件自行实现的，因此可以不受物理条件制约地定制指令集与执行引擎的结构体系，能够执行那些不被硬件直接支持的指令集格式。

在《Java 虚拟机规范》中制定了 Java 虚拟机字节码执行引擎的概念模型，这个概念模型成为各大发行商的 Java 虚拟机执行引擎的统一外观（Facade）。在不同的虚拟机实现中，执行引擎在执行字节码的时候，通常会有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执行）两种选择，也可能两者兼备，还可能会有同时包含几个不同级别的即时编译器一起工作的执行引擎。但从外观上来看，所有的 Java 虚拟机的执行引擎输入、输出都是一致的：输入的是字节码二进制流，处理过程是字节码解析执行的等效过程，输出的是执行结果，本章将主要从概念模型的角度来讲解虚拟机的方法调用和字节码执行。

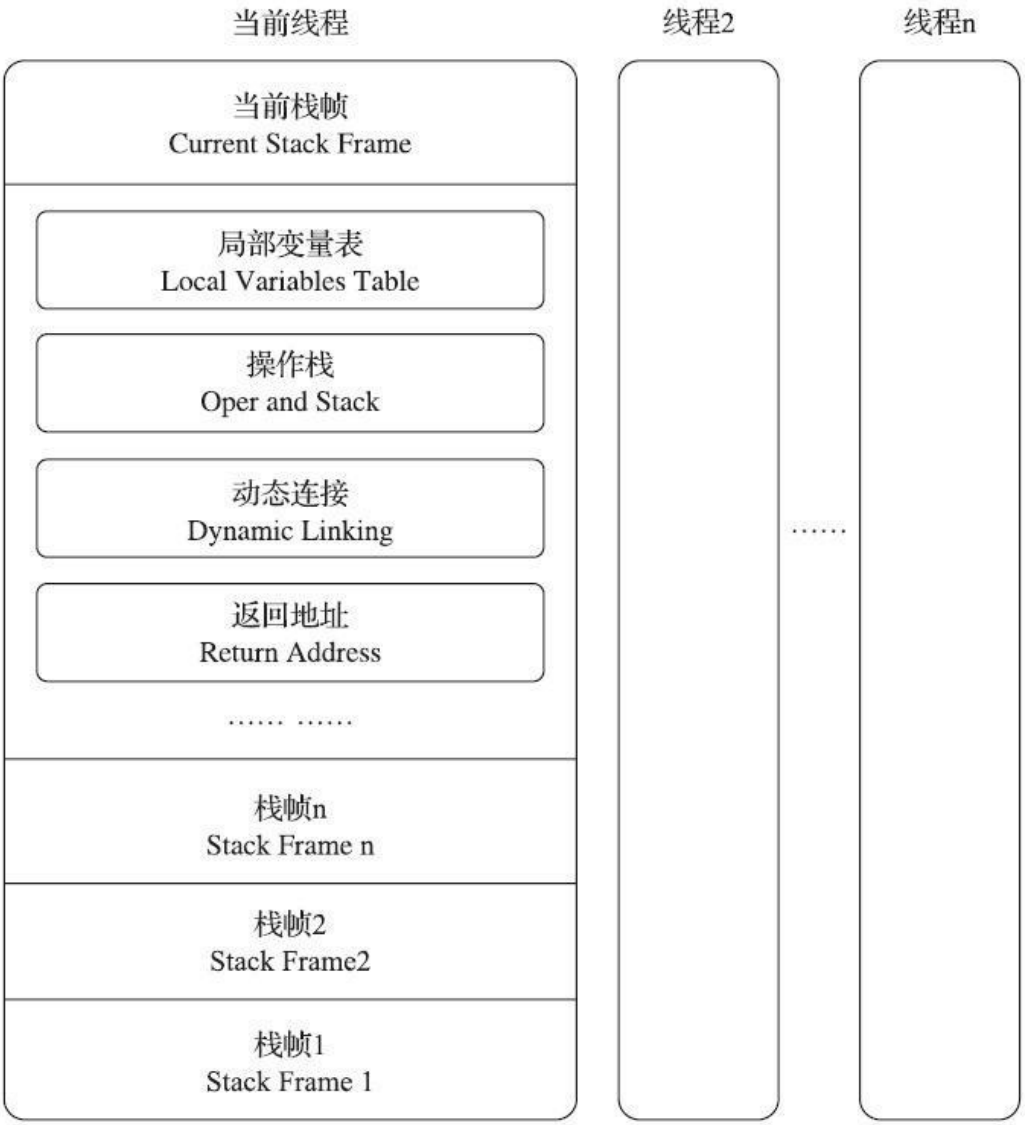
5.2 运行时栈帧结构

Java 虚拟机以方法作为最基本的执行单元，“栈帧”（Stack Frame）则是用于支持虚拟机进行方法调用和方法执行背后的数据结构，它也是虚拟机运行时数据区中的虚拟机栈（Virtual Machine Stack）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至执行结束的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

每一个栈帧都包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。在编译 Java 程序源码的时候，栈帧中需要多大的局部变量表，需要多深的操作数栈就已经被分析计算出来，并且写入到方法表的 Code 属性之中。换言之，一个栈帧需要分配多少内存，并不会受到程序运行期变量数据的影响，而仅仅取决于程序源码和具体的虚拟机实现的栈内存布局形式。

一个线程中的方法调用链可能会很长，以 Java 程序的角度来看，同一时刻、同一条线程里面，在调用堆栈的所有方法都同时处于执行状态。而对于执行引擎来讲，在活动线程中，只有位于栈顶的方法才是在运行的，只有位于栈顶的栈帧才是生效的，其被称为“当前栈帧”（Current Stack Frame），与这个栈帧所关联的方法被称为“当前方法”（Current Method）。

一个典型的栈帧结构如图所示：


当前线程
线程2
线程n
当前栈帧
Current Stack Frame
局部变量表
Local Variables Table
操作栈
Oper and Stack
动态连接
Dynamic Linking
返回地址
Return Address
…… ……
栈帧n
Stack Frame n
栈帧2
Stack Frame2
栈帧1
Stack Frame 1
……

5.2.1 局部变量表

局部变量表（Local Variables Table）是一组变量值的存储空间，用于存放方法参数和方法内部定义的局部变量。在 Java 程序被编译为 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分配的局部变量表的最大容量。局部变量表的容量以变量槽（Variable Slot）为最小单位。

一个变量槽可以存放一个 32 位以内的数据类型，Java 中占用不超过 32 位存储空间的数据类型有 boolean、byte、char、short、int、float、reference[1]和 returnAddress 这 8 种类型。

reference 类型表示对一个对象实例的引用，《Java 虚拟机规范》既没有说明它的长度，也没有明确指出这种引用应有怎样的结构。但是一般来说，虚拟机实现至少都应当能通过这个引用做到两件事情，一是从根据引用直接或间接地查找到对象在 Java 堆中的数据存放的起始地址或索引，二是根据引用直接或间接地查找到对象所属数据类型在方法区中的存储的类型信息，否则将无法实现《Java 语言规范》中定义的语法约定。

当一个方法被调用时，Java 虚拟机会使用局部变量表来完成参数值到参数变量列表的传递过程，即实参到形参的传递。如果执行的是实例方法（没有被 static 修饰的方法），那局部变量表中第 0 位索引的变量槽默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问到这个隐含的参数。其余参数则按照参数表顺序排列，占用从 1 开始的局部变量槽，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的变量槽

为了尽可能节省栈帧耗用的内存空间，局部变量表中的变量槽是可以重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法体，如果当前字节码 PC 计数器的值已经超出了某个变量的作用域，那这个变量对应的变量槽就可以交给其他变量来重用。不过，这样的设计除了节省栈帧空间以外，还会伴随有少量额外的副作用，例如在某些情况下变量槽的复用会直接影响到系统的垃圾收集行为：

```
public static void main(String[] args)() {
    byte[] placeholder = new byte[64 * 1024 * 1024];
    System.gc();
}
```

如图所示：现在定义一个 byte[]数组，里面有 64M 的数据。现在通知垃圾回收器回收，发现运行后还是未被回收。没有被回收还解释的过去，因为在执行 gc()时，placeholder 还在作用域内，虚拟机自然不敢回收掉 placeholder 的内存。（2.2.3 中说过，这是个强引用，只有 placeholder = null 了才能回收掉那部分内存）现在修改下代码：

```
public static void main(String[] args)() {
    {
        byte[] placeholder = new byte[64 * 1024 * 1024];
    }
    System.gc();
}
```

加了代码块后，placeholder 的作用域被局限于代码块中，现在再 gc()，发现仍然没有回收掉。解释原因之前，再修改下代码看下结果：

```
public static void main(String[] args)() {
    {
        byte[] placeholder = new byte[64 * 1024 * 1024];
    }
    int a = 0;
    System.gc();
}
```

这个代码仅仅在原来的基础上加了 int a=0，但是结果却莫名其妙的发现，内存被回收了。

在上面的三个代码中，placeholder 能否被回收的根本原因就是：局部变量表中的变量槽是否还存有关于 placeholder 数组对象的引用。第一次修改中，代码虽然已经离开了 placeholder 的作用域，但在此之后，再没有发生过任何对局部变量表的读写操作，placeholder 原本所占用的变量槽还没有被其他变量所复用，所以作为 GC Roots 一部分的局部变量表仍然保持着对它的关联。

类的字段变量有两次赋初始值的过程，一次在准备阶段，赋予系统初始值；另外一次在初始化阶段，赋予程序员定义的初始值。因此即使在初始化阶段程序员没有为类变量赋值也没有关系，类变量仍然具有一个确定的初始值，不会产生歧义。但局部变量就不一样了，如果一个局部变量定义了但没有赋初始值，那它是完全不能使用的。（即局部变量必须赋初始值，类成员变量，其实包括普通成员变量会自动赋初始值）

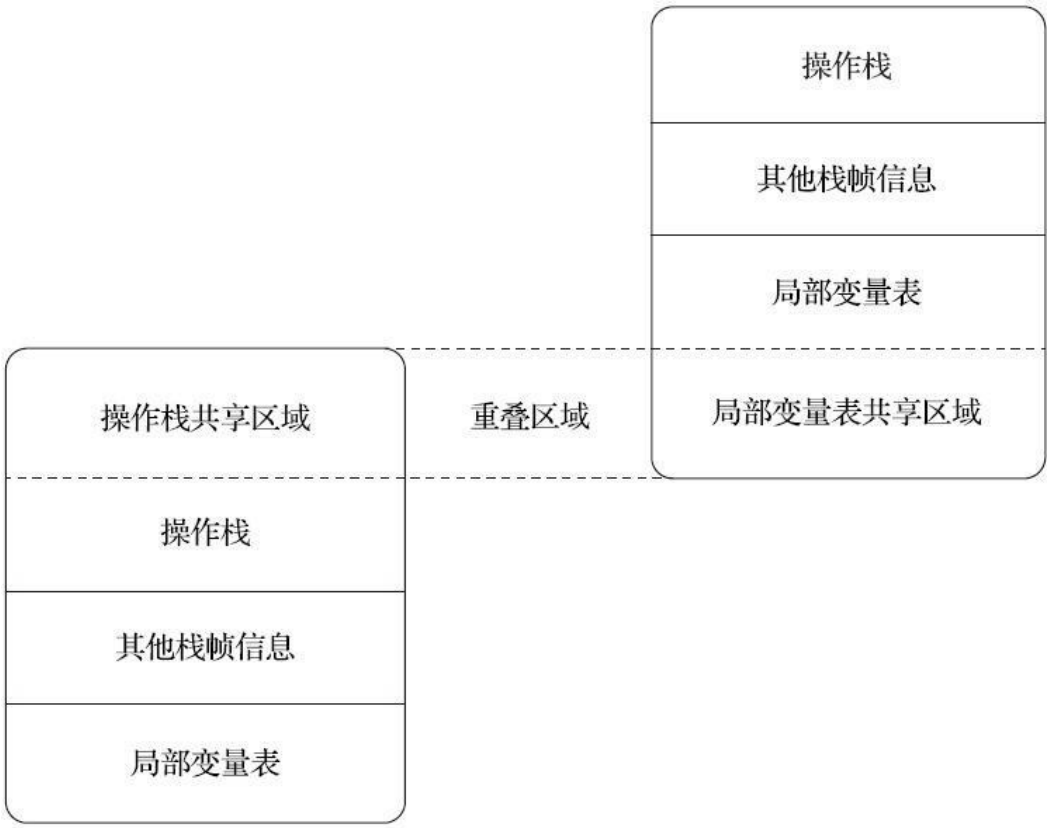
5.2.2 操作数栈

操作数栈（Operand Stack）也常被称为操作栈，它是一个后入先出（Last In First Out，LIFO）栈。同局部变量表一样，操作数栈的最大深度也在编译的时候被写入到 Code 属性的 max_stacks 数据项之中。操作数栈的每一个元素都可以是包括 long 和 double 在内的任意 Java 数据类型。32 位数据类型所占的栈容量为 1，64 位数据类型所占的栈容量为 2。

当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是出栈和入栈操作。举个例子，例如整数加法的字节码指令 iadd，这条指令在运行的时候要求操作数栈中最接近栈顶的两个元素已经存入了两个 int 型的数值，当执行这个指令时，会把这两个 int 值出栈并相加，然后将相加的结果重新入栈。

两个不同栈帧作为不同方法的虚拟机栈的元素，是完全相互独立的。但是在大多虚拟机的实现里都会进行一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样做不仅节约了一些空间，更重要的是在进行方法调用时就可以直接共用一部分数据，无须进行额外的参数复制传递了。

Java 虚拟机的解释执行引擎被称为“基于栈的执行引擎”，里面的“栈”就是操作数栈。



5.2.3 动态连接

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连接（Dynamic Linking）。通过前面的讲解，我们知道 Class 文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池里指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或者第一次使用的时候就被转化为直接引用，这种转化被称为静态解析。另外一部分将在每一次运行期间都转化为直接引用，这部分就称为动态连接。关于这两个转化过程的具体过程，将在后面小节中再详细讲解。

5.2.4 方法返回地址

当一个方法开始执行后，只有两种方式退出这个方法。第一种方式是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者（调用当前方法的方法称为调用者或者主调方法），方法是否有返回值以及返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式称为“正常调用完成”（Normal Method Invocation Completion）。

另外一种退出方式是在方法执行的过程中遇到了异常，并且这个异常没有在方法体内得到妥善处理。无论是 Java 虚拟机内部产生的异常，还是代码中使用 athrow 字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为“异常调用完成（Abrupt Method Invocation Completion）”。一个方法使用异常完成出口的方式退出，是不会给它的上层调用者提供任何返回值的。

无论采用何种退出方式，在方法退出之后，都必须返回到最初方法被调用时的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层主调方法的执行状态。一般来说，方法正常退出时，主调方法的 PC 计数器的值就可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中就一般不会保存这部分信息。

方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，把返回值（如果有的话）压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方法调用指令后面的一条指令等。笔者这里写的“可能”是由于这是基于概念模型的讨论，只有具体到某一款 Java 虚拟机实现，会执行哪些操作才能确定下来。

5.2.5 附加信息

《Java 虚拟机规范》允许虚拟机实现增加一些规范里没有描述的信息到栈帧之中，例如与调试、性能收集相关的信息，这部分信息完全取决于具体的虚拟机实现，这里不再详述。在讨论概念时，一般会把动态连接、方法返回地址与其他附加信息全部归为一类，称为栈帧信息。

5.3 方法调用

方法调用并不等同于方法中的代码被执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方法），暂时还未涉及方法内部的具体运行过程。在程序运行时，进行方法调用是最普遍、最频繁的操作之一。

Class 文件的编译过程中不包含传统程序语言编译的连接步骤，一切方法调用在 Class 文件里面存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址（也就是之前说的直接引用）。这个特性给 Java 带来了更强大的动态扩展能力，但也使得 Java 方法调用过程变得相对复杂，某些调用需要在类加载期间，甚至到运行期间才能确定目标方法的直接引用。

5.3.1 解析

所有方法调用的目标方法在 Class 文件里面都是一个常量池中的符号引用，在类加载的解析阶段，会将其中的一部分符号引用转化为直接引用，这种解析能够成立的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译那一刻就已经确定下来。这类方法的调用被称为解析（Resolution）。

在 Java 语言中符合“编译期可知，运行期不可变”这个要求的方法，主要有静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法各自的特点决定了它们都不可能通过继承或别的方式重写出其他版本，因此它们都适合在类加载阶段进行解析。

调用不同类型的方法，字节码指令集里设计了不同的指令。在 Java 虚拟机支持以下 5 条方法调用字节码指令，分别是：

invokestatic。用于调用静态方法。

invokespecial。用于调用实例构造器<init>()方法、私有方法和父类中的方法。

invokevirtual。用于调用所有的虚方法。

invokeinterface。用于调用接口方法，会在运行时再确定一个实现该接口的对象。

invokedynamic。先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法。前面 4 条调用指令，分派逻辑都固化在 Java 虚拟机内部，而 invokedynamic 指令的分派逻辑是由用户设定的引导方法来决定的。

只要能被 invokestatic 和 invokespecial 指令调用的方法，都可以在解析阶段中确定唯一的调用版本，Java 语言里符合这个条件的方法共有静态方法、私有方法、实例构造器、父类方法 4 种，再加上被 final 修饰的方法（尽管它使用 invokevirtual 指令调用），这 5 种方法调用会在类加载的时候就可以把符号引用解析为该方法的直接引用。这些方法统称为“非虚方法”（Non-Virtual Method），与之相反，其他方法就被称为“虚方法”（Virtual Method）。（即在运行时常量池中能找到具体方法符号引用）

Java 中的非虚方法除了使用 invokestatic、invokespecial 调用的方法之外还有一种，就是被 final 修饰的实例方法。虽然由于历史设计的原因，final 方法是使用 invokevirtual 指令来调用的，但是因为它也无法被覆盖，没有其他版本的可能，所以也无须对方法接收者进行多态选择，又或者说多态选择的结果肯定是唯一的。在《Java 语言规范》中明确定义了被 final 修饰的方法是一种非虚方法。

解析调用一定是个静态的过程，在编译期间就完全确定，在类加载的解析阶段就会把涉及的符号引用全部转变为明确的直接引用，不必延迟到运行期再去完成。而另一种主要的方法调用形式：分派（Dispatch）调用则要复杂许多，它可能是静态的也可能是动态的，按照分派依据的宗量数可分为单分派和多分派。这两类分派方式两两组合就构成了静态单分派、静态多分派、动态单分派、动态多分派 4 种分派组合情况。

5.3.2 分派

在本节，将会揭示多态性特征的一些最基本的体现，如“重载”和“重写”在 Java 虚拟机中是如何实现的。实际上是虚拟机是如何确定正确的方法。

1. 静态分派

在英文原版的《Java 虚拟机规范》和《Java 语言规范》里的说法都是“Method Overload Resolution”，即应该归入“解析”里去讲解。但在英文教材中或者国内翻译的中文资料里，都将这种行为称为“静态分派”。

解释重载（Overload）和静态分派前，先看一个代码：

```
public class StaticDispatch {

    static abstract class Human {
    }

    static class Man extends Human {
    }

    static class Woman extends Human {
    }

    public void sayHello(Human guy)
        { System.out.println("hello,guy!");
    }

    public void sayHello(Man guy)
        { System.out.println("hello,gentleman!");
    }

    public void sayHello(Woman guy)
        { System.out.println("hello,lady!");
    }

    public static void main(String[] args)
        { Human man = new Man();
        Human woman = new Woman();
        StaticDispatch sr = new StaticDispatch();
        sr.sayHello(man);
        sr.sayHello(woman);
    }

}
```

结果：都执行的参数类型为 Human 的重载方法。打印“hello,guy!”。

在解释这个结果前，先解释两个关键概念，对于：

Human man = new Man();

我们把“Human”称为变量的“静态类型”，或者叫“外观类型”，后面的“Man”则称为“实际类型”，或者叫“运行时类型”。静态类型和实际类型在程序中都可能会发生变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型在编译期可知；而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。

main()里面的两次 sayHello()方法调用，在方法接收者已经确定是对象“sr”的前提下，使用哪个重载版本，就完全取决于传入参数的数量和数据类型。代码中故意定义了两个静态类型相同，而实际类型不同的变量，但虚拟机（或者准确地说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。由于静态类型在编译期可知，所以在编译阶段，Javac 编译器就根据参数的静态类型决定了会使用哪个重载版本，因此选择了 sayHello(Human)作为调用目标，并把这个方法的符号引用写到 main()方法里的两条 invokevirtual 指令的参数中。

```
26: invokevirtual #8                  // Method sayHello:(LStaticDispatch$Human;)V
29: aload_3
30: aload_2
31: invokevirtual #8                  // Method sayHello:(LStaticDispatch$Human;)V
```

所有依赖静态类型来决定方法执行版本的分派动作，都称为静态分派。静态分派的最典型应用表现就是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的，这点也是为何一些资料选择把它归入“解析”而不是“分派”的原因。

需要注意 Javac 编译器虽然能确定出方法的重载版本，但在很多情况下这个重载版本并不是“唯一”的，往往只能确定一个“相对更合适的”版本。例如：

```
package org.fenixsoft.polymorphic;

public class Overload {

    public static void sayHello(Object arg)
        { System.out.println("hello Object");
    }

    public static void sayHello(int arg)
        { System.out.println("hello int");
    }

    public static void sayHello(long arg)
        { System.out.println("hello long");
    }

    public static void sayHello(Character arg)
        { System.out.println("hello Character");
    }

    public static void sayHello(char arg)
        { System.out.println("hello char");
    }

    public static void sayHello(char... arg)
        { System.out.println("hello char ...");
    }

    public static void sayHello(Serializable arg)
        { System.out.println("hello Serializable");
    }

    public static void main(String[] args)
        { sayHello('a');
    }
}
```

输出：hello char

如果注释掉 sayHello(char arg)方法，输出：hello int。
这是因为这个发生了自动转换，将'a'代表了 97。

继续注释掉 sayHello(int arg)方法，输出：hello long。
这时发生了两次自动类型转换，'a'转型为整数 97 后，进一步转型为长整型 97L，匹配了参数类型为 long 的重载。实际上自动转型还能继续发生多次，按照 char>int>long>float>double 的顺序转型进行匹配，但不会匹配到 byte 和 short 类型的重载，因为 char 到 byte 或 short 的转型是不安全的。

继续注释掉 sayHello(long arg)方法，输出：hello Character。
这时发生了一次自动装箱，'a'被包装为它的封装类型 java.lang.Character，所以匹配到了参数类型为 Character 的重载。

继续注释掉 sayHello(Character arg)方法，输出：hello Serializable。
当自动装箱后找不到装箱类，但是找到了装箱类所实现的接口类型，所以又发生了一次自动转型。char 可以转型成 int， 但是 Character 是绝对不会转型为 Integer 的，它只能安全地转型为它实现的接口或父类。Character 还实现了另外一个接口 java.lang.Comparable<Character>，如果同时出现两个参数分别为 Serializable 和 Comparable<Character>的重载方法，那它们在此时的优先级是一样的。编译器无法确定要自动转型为哪种类型，会提示"类型模糊"（Type Ambiguous），并拒绝编译。

继续注释掉 sayHello(Serializable arg)方法，输出：hello Object。
这时是 char 装箱后转型为父类了，如果有多个父类，那将在继承关系中从下往上开始搜索，越接上层的优先级越低。即使方法调用传入的参数值为 null 时，这个规则仍然适用。

继续注释掉 sayHello(Object arg)方法，输出：hello char...
可见变长参数的重载优先级是最低的。

上面这个例子，除了用作面试题为难求职者之外，在实际工作中几乎不可能存在任何有价值的用途。

解析与分派这两者之间的关系并不是二选一的排他关系，它们是在不同层次上去筛选、确定目标方法的过程。例如前面说过静态方法会在编译期确定、在类加载期就进行解析，而静态方法显然也是可以拥有重载版本的，选择重载版本的过程也是通过静态分派完成的。

2. 动态分派

动态分派的实现过程，与 Java 语言的——重写（Override）相关。先看下列代码：

```
package org.fenixsoft.polymorphic;

/**
 * 方法动态分派演示
 * @author zzm
 */
public class DynamicDispatch {

     static abstract class Human {
         protected abstract void sayHello();
     }

     static class Man extends Human
         { @Override
         protected void sayHello()
             { System.out.println("man say hello");
         }
     }

     static class Woman extends Human
         { @Override
         protected void sayHello()
             { System.out.println("woman say hello");
         }
     }

     public static void main(String[] args)
         { Human man = new Man();
         Human woman = new Woman();
         man.sayHello();
         woman.sayHello();
         man = new Woman();
         man.sayHello();
     }

}
```

运行结果：

```
man say hello
woman say hello
woman say hello
```

显然这里选择调用方法版本不可能是根据静态类型来决定，因为静态类型同样都是 Human 的两个变量在调用 sayHello 时产生了不同的行为，甚至执行了两个不同的方法。导致这个原因是因为两个变量的实际类型不同。

查看该代码的字节码：

```
public static void main(java.lang.String[]);
    Code:
        Stack=2, Locals=3, Args_size=1
         0:   new     #16; //class org/fenixsoft/polymorphic/DynamicDispatch$Man
         3:   dup
         4:   invokespecial   #18; //Method org/fenixsoft/polymorphic/Dynamic Dispatch$Man."<init>":()V
         7:   astore_1
         8:   new     #19; //class org/fenixsoft/polymorphic/DynamicDispatch$Woman
        11: dup
        12:  invokespecial   #21; //Method org/fenixsoft/polymorphic/DynamicDispatch$Woman."<init>":()V
        15: astore_2
        16: aload_1
        17:  invokevirtual   #22; //Method org/fenixsoft/polymorphic/Dynamic Dispatch$Human.sayHello:()V
        20: aload_2
        21:  invokevirtual   #22; //Method org/fenixsoft/polymorphic/Dynamic Dispatch$Human.sayHello:()V
        24:  new     #19; //class org/fenixsoft/polymorphic/DynamicDispatch$Woman
        27: dup
        28:  invokespecial   #21; //Method org/fenixsoft/polymorphic/DynamicDispatch$Woman."<init>":()V
        31: astore_1
        32: aload_1
        33:  invokevirtual   #22; //Method org/fenixsoft/polymorphic/Dynamic Dispatch$Human.sayHello:()V
        36: return
```

0-15 是分配对象空间，实例化对象的过程。

16-21 行是关键，在 16 行和 20 行，aload 指令将创建的两个对象压入栈顶，在 17 行和 21 行，都是调用的 Human.sayHello()方法。根据《Java 虚拟机规范》，invokevirtual 指令的运行时解析过程大致分为以下几步：

1）找到操作数栈顶的第一个元素所指向的对象的实际类型，记作 C。

2）如果在类型 C 中找到与常量中的描述符和简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；不通过则返回 java.lang.IllegalAccessError 异常。

3）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。

4）如果始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError 异常。

正是因为 invokevirtual 指令执行的第一步就是在运行期确定接收者的实际类型，所以两次调用中的 invokevirtual 指令并不是把常量池中方法的符号引用解析到直接引用上就结束了，还会根据方法接收者的实际类型来选择方法版本，（上面那个例子，第一个对象先压入栈顶，然后 invokevirtual 指令找到栈顶元素所指向的对象的实际类型 Man）这个过程就是 Java 语言中方法重写的本质。我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

既然这种多态性的根源在于虚方法调用指令 invokevirtual 的执行逻辑，那自然我们得出的结论就只会对方法有效，对字段是无效的，因为字段不使用这条指令。事实上，在 Java 里面只有虚方法存在， 字段永远不可能是虚的，换句话说，字段永远不参与多态，哪个类的方法访问某个名字的字段时，该名字指的就是这个类能看到的那个字段。当子类声明了与父类同名的字段时，虽然在子类的内存中两个字段都会存在，但是子类的字段会遮蔽父类的同名字段。

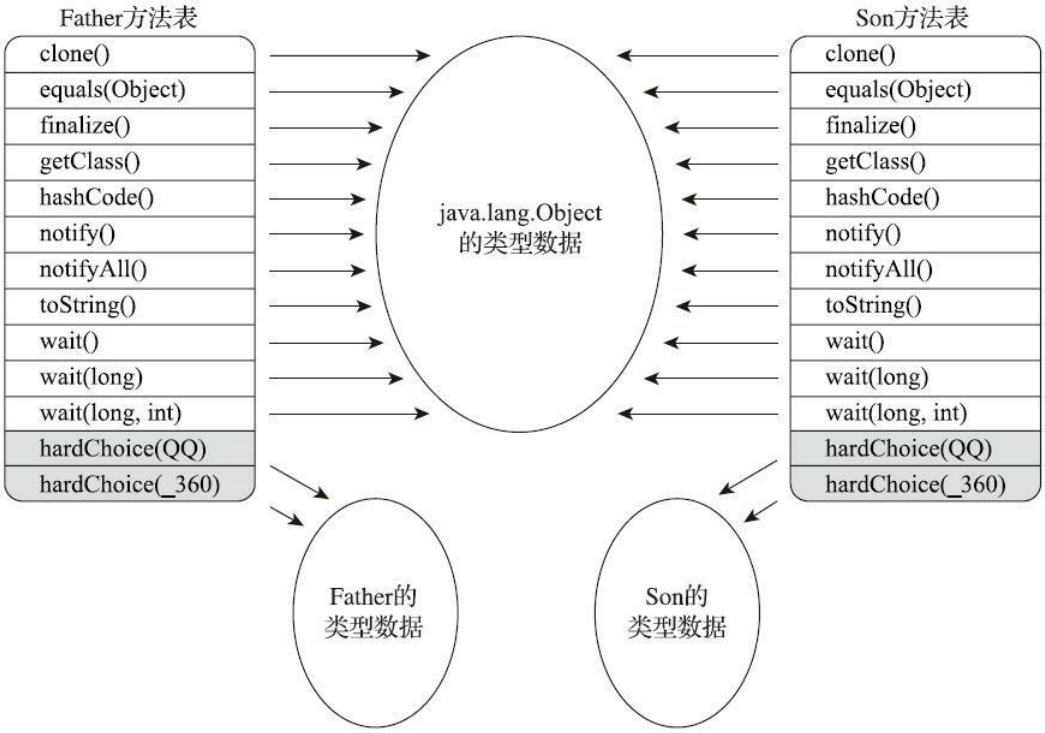
3. 单分派与多分派

方法的接收者与方法的参数统称为方法的宗量，这个定义最早应该来源于著名的《Java 与模式》一书。根据分派基于多少种宗量，可以将分派划分为单分派和多分派两种。单分派是根据一个宗量对目标方法进行选择，多分派则是根据多于一个宗量对目标方法进行选择。

按照目前 Java 语言的发展趋势，它并没有直接变为动态语言的迹象，而是通过内置动态语言（如 JavaScript）执行引擎、加强与其他 Java 虚拟机上动态语言交互能力的方式来间接地满足动态性的需求。但是作为多种语言共同执行平台的 Java 虚拟机层面上则不是如此，早在 JDK 7 中实现的 JSR-292[7] 里面就已经开始提供对动态语言的方法调用支持了，JDK 7 中新增的 invokedynamic 指令也成为最复杂的一条方法调用的字节码指令.

4. 虚拟机动态分派的实现。

动态分派是执行非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在接收者类型的方法元数据中搜索合适的目标方法，因此，Java 虚拟机实现基于执行性能的考虑，真正运行时一般不会如此频繁地去反复搜索类型元数据。面对这种情况，一种基础而且常见的优化手段是为类型在方法区中建立一个虚方法表（Virtual Method Table，也称为 vtable，与此对应的，在 invokeinterface 执行时也会用到接口方法表——Interface Method Table，简称 itable），使用虚方法表索引来代替元数据查找以提高性能。



虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那子类的虚方法表中的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类虚方法表中的地址也会被替换为指向子类实现版本的入口地址。

为了程序实现方便，具有相同签名的方法，在父类、子类的虚方法表中都应当具有一样的索引序号，这样当类型变换时，仅需要变更查找的虚方法表，就可以从不同的虚方法表中按索引转换出所需的入口地址。虚方法表一般在类加载的连接阶段进行初始化，准备了类的变量初始值后，虚拟机会把该类的虚方法表也一同初始化完毕。

5.4 动态类型语言支持

5.4.1 动态类型语言

动态类型语言的关键特征是它的类型检查的主体过程是在运行期而不是编译期进行的，满足这个特征的语言有很多，常用的包括：APL、Clojure、Erlang、Groovy、JavaScript、Lisp、Lua、PHP、Prolog、Python、Ruby、Smalltalk、Tcl，等等。那相对地，在编译期就进行类型检查过程的语言，譬如 C++和 Java 等就是最常用的静态类型语言。如图：

```
public static void main(String[] args)
    { int[][][] array = new  int[1][0][-1];
}
```

上面这段 Java 代码能够正常编译，但运行的时候会出现 NegativeArraySizeException 异常。在《Java 虚拟机规范》中明确规定了 NegativeArraySizeException 是一个运行时异常（Runtime Exception），通俗一点说，运行时异常就是指只要代码不执行到这一行就不会出现问题。与运行时异常相对应的概念是连接时异常，例如很常见的 NoClassDefFoundError 便属于连接时异常，即使导致连接时异常的代码放在一条根本无法被执行到的路径分支上，类加载时也照样会抛出异常。

5.4.2 Java 与动态类型

Java 虚拟机层面对动态类型语言的支持一直都还有所欠缺，主要表现在方法调用方 面：JDK 7 以前的字节码指令集中，4 条方法调用指令（invokevirtual、invokespecial、invokestatic、invokeinterface）的第一个参数都是被调用的方法的符号引用（CONSTANT_Methodref_info 或者 CONSTANT_InterfaceMethodref_info 常量），前面已经提到过，方法的符号引用在编译时产生，而动态类型语言只有在运行期才能确定方法的接收者。

5.4.3 java.lang.invoke 包

JDK 7 时新加入的 java.lang.invoke 包[1]是 JSR 292 的一个重要组成部分，这个包的主要目的是在之前单纯依靠符号引用来确定调用的目标方法这条路之外，提供一种新的动态确定目标方法的机制，称为“方法句柄”（Method Handle）。

```
import static java.lang.invoke.MethodHandles.lookup;
import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodType;

/**
 * JSR 292 MethodHandle 基础用法演示
 * @author zzm
 */
public class MethodHandleTest {

    static class ClassA {
        public void println(String s)
            { System.out.println(s);
        }
    }

    public static void main(String[] args) throws Throwable {
        Object obj = System.currentTimeMillis() % 2 == 0 ? System.out : new ClassA();
        // 无论 obj 最终是哪个实现类，下面这句都能正确调用到 println 方法。
        getPrintlnMH(obj).invokeExact("icyfenix");
    }

    private static MethodHandle getPrintlnMH(Object reveiver) throws Throwable {
        // MethodType：代表“方法类型”，包含了方法的返回值（methodType()的第一个参数）
           和具体参数（methodType()第二个及以后的参数）。
        MethodType mt = MethodType.methodType(void.class, String.class);
        // lookup()方法来自于 MethodHandles.lookup，这句的作用是在指定类中查找符合给定的
           方法名称、方法类型，并且符合调用权限的方法句柄。

        // 因为这里调用的是一个虚方法，按照 Java 语言的规则，方法第一个参数是隐式的，代表该方法
           的接收者，也即 this 指向的对象，这个参数以前是放在参数列表中进行传递，现在提供了
           bindTo() 方法来完成这件事情。
        return lookup().findVirtual(reveiver.getClass(), "println", mt).bindTo(reveiver);
    }
}
```

方法 getPrintlnMH()中实际上是模拟了 invokevirtual 指令的执行过程，只不过它的分派逻辑并非固化在 Class 文件的字节码上，而是通过一个由用户设计的 Java 方法来实现。而这个方法本身的返回值（MethodHandle 对象），可以视为对最终调用方法的一个“引用”。

Reflection 和 MethodHandle 的区别：

Reflection 和 MethodHandle 机制本质上都是在模拟方法调用，但是 Reflection 是在模拟 Java 代码层次的方法调用，而 MethodHandle 是在模拟字节码层次的方法调用。在 MethodHandles.Lookup 上的 3 个方法 findStatic()、findVirtual()、findSpecial()正是为了对应于 invokestatic、invokevirtual（以及 invokeinterface）和 invokespecial 这几条字节码指令的执行权限校验行为，而这些底层细节在使用 Reflection API 时是不需要关心的。

Reflection 中的 java.lang.reflect.Method 对象远比 MethodHandle 机制中的 java.lang.invoke.MethodHandle 对象所包含的信息来得多。前者是方法在 Java 端的全面映像，包含了方法的签名、描述符以及方法属性表中各种属性的 Java 端表示方式，还包含执行权限等的运行期信息。而后者仅包含执行该方法的相关信息。用开发人员通俗的话来讲，Reflection 是重量级，而 MethodHandle 是轻量级。

由于 MethodHandle 是对字节码的方法指令调用的模拟，那理论上虚拟机在这方面做的各种优化（如方法内联），在 MethodHandle 上也应当可以采用类似

思路去支持（但目前实现还在继续完善中），而通过反射去调用方法则几乎不可能直接去实施各类调用点优化措施。

最关键的一点还在于去掉前面讨论施加的前 提“仅站在 Java 语言的角度看”之后：Reflection API 的设计目标是只为 Java 语言服务的，而 MethodHandle 则设计为可服务于所有 Java 虚拟机之上的语言，其中也包括了 Java 语言而已，而且 Java 在这里并不是主角。

5.4.4 invokedynamic 指令

JDK7 为了更好地支持动态语言，引入第五条方法调用的字节码指令 invokedynamic。某种意义上可以说 invokedynamic 指令与 MethodHandle 机制的作用是一样的，都是为了解决原有 4 条“invoke*”指令方法分派规则完全固化在虚拟机之中的问题，把如何查找目标方法的决定权从虚拟机转嫁到具体用户代码之中，让用户（广义的用户，包含其他程序语言的设计者）有更高的自由度。

每一处含有 invokedynamic 指令的位置都被称作“动态调用点（Dynamically-Computed Call Site）”，这条指令的第一个参数不再是代表方法符号引用的 CONSTANT_Methodref_info 常量，而是变为 JDK 7 时新加入的 CONSTANT_InvokeDynamic_info 常量，从这个新常量中可以得到 3 项信息：引导方法（Bootstrap Method，该方法存放在新增的 BootstrapMethods 属性中）、方法类型（MethodType）和名称。引导方法是有固定的参数，并且返回值规定是 java.lang.invoke.CallSite 对象，这个对象代表了真正要执行的目标方法调用。根据 CONSTANT_InvokeDynamic_info 常量中提供的信息，虚拟机可以找到并且执行引导方法，从而获得一个 CallSite 对象，最终调用到要执行的目标方法上。

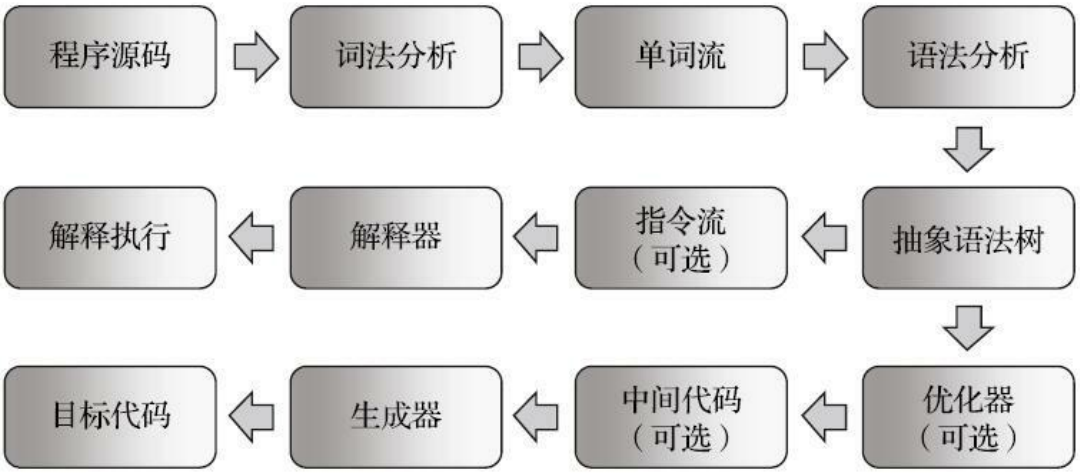
5.5 基于栈的字节码解释执行引擎

许多 Java 虚拟机的执行引擎在执行 Java 代码的时候都有解释执行（通过解释器执行）和编译执行（通过即时编译器产生本地代码执 行）两种选择，在本节中，我们将会分析在概念模型下的 Java 虚拟机解释执行字节码时，其执行引擎是如何工作的。

5.5.1 解释执行

Java 语言经常被人们定位为“解释执行”的语言，在 Java 初生的 JDK 1.0 时代，这种定义还算是比较准确的，但当主流的虚拟机中都包含了即时编译器后，Class 文件中的代码到底会被解释执行还是编译执行，就成了只有虚拟机自己才能准确判断的事。

无论是解释还是编译，也无论是物理机还是虚拟机，对于应用程序，机器都不可能如人那样阅 读、理解，然后获得执行能力。大部分的程序代码转换成物理机的目标代码或虚拟机能执行的指令集之前，都需要经过如图所示的各个步骤。如果读者对大学编译原理的相关课程还有印象的话，很容易就会发现如图所示中下面的那条分支，就是传统编译原理中程序代码到目标机器代码的生成过程；而中间的那条分支，自然就是解释执行的过程。



如今，基于物理机、Java 虚拟机，或者是非 Java 的其他高级语言虚拟机（HLLVM）的代码执行过程，大体上都会遵循这种符合现代经典编译原理的思路，在执行前先对程序源码进行词法分析和语法分析处理，把源码转化为抽象语法树（Abstract Syntax Tree，AST）。对于一门具体语言的实现来说， 词法、语法分析以至后面的优化器和目标代码生成器都可以选择独立于执行引擎，形成一个完整意义的编译器去实现，这类代表是 C/C++语言。也可以选择把其中一部分步骤（如生成抽象语法树之前的步骤）实现为一个半独立的编译器，这类代表是 Java 语言。又或者把这些步骤和执行引擎全部集中封装在一个封闭的黑匣子之中，如大多数的 JavaScript 执行引擎。

在 Java 语言中，Javac 编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生成线性的字节码指令流的过程。因为这一部分动作是在 Java 虚拟机之外进行的，而解释器在虚拟机的内部，所以 Java 程序的编译就是半独立的实现。

5.5.2 基于栈的指令集与基于寄存器的指令集

Javac 编译器输出的字节码指令流，基本上是一种基于栈的指令集架构（Instruction Set Architecture，ISA），字节码指令流里面的指令大部分都是零地址指令，它们依赖操作数栈进行工作。与之相对的另外一套常用的指令集架构是基于寄存器的指令集，最典型的就是 x86 的二地址指令集，如果说得更通俗一些就是现在我们主流 PC 机中物理硬件直接支持的指令集架构，这些指令依赖寄存器进行工作。

例如，分别使用这两种指令集去计算“1+1”的结果，基于栈的指令集会是这样子的：

a. iconst_1
b. iconst_1
c. iadd
d. istore_0

两条 iconst_1 指令连续把两个常量 1 压入栈后，iadd 指令把栈顶的两个值出栈、相加，然后把结果放回栈顶，最后 istore_0 把栈顶的值放到局部变量表的第 0 个变量槽中。这种指令流中的指令通常都是不带参数的，使用操作数栈中的数据作为指令的运算输入，指令的运算结果也存储在操作数栈之中。而如果用基于寄存器的指令集，那程序可能会是这个样子：

a. mov eax, 1
b. add eax, 1

mov 指令把 EAX 寄存器的值设为 1，然后 add 指令再把这个值加 1，结果就保存在 EAX 寄存器里面。这种二地址指令是 x86 指令集中的主流，每个指令都包含两个单独的输入参数，依赖于寄存器来访问和存储数据。

基于栈的指令集主要优点是可移植，因为寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。栈架构的指令集还有一些其他的优点， 如代码相对更加紧凑（字节码中每个字节就对应一条指令，而多地址指令集中还需要存放参数）、编译器实现更加简单（不需要考虑空间分配的问题，所需空间都在栈上操作）等.

栈架构指令集的主要缺点是理论上执行速度相对来说会稍慢一些，所有主流物理机的指令集都是寄存器架构也从侧面印证了这点。在解释执行时，栈架构指令集的代码虽然紧凑，但是完成相同功能所需的指令数量一般会比寄存器架构来得更多，因为出栈、入栈操作本身就产生了相当大量的指令。更重要的是栈实现在内存中， 频繁的栈访问也就意味着频繁的内存访问，相对于处理器来说，内存始终是执行速度的瓶颈。尽管虚拟机可以采取栈顶缓存的优化方法，把最常用的操作映射到寄存器中避免直接内存访问，但这也只是优化措施而不是解决本质问题的方法。因此由于指令数量和内存访问的原因，导致了栈架构指令集的执行速度会相对慢上一点。

5.5.3 基于栈的解释器执行过程

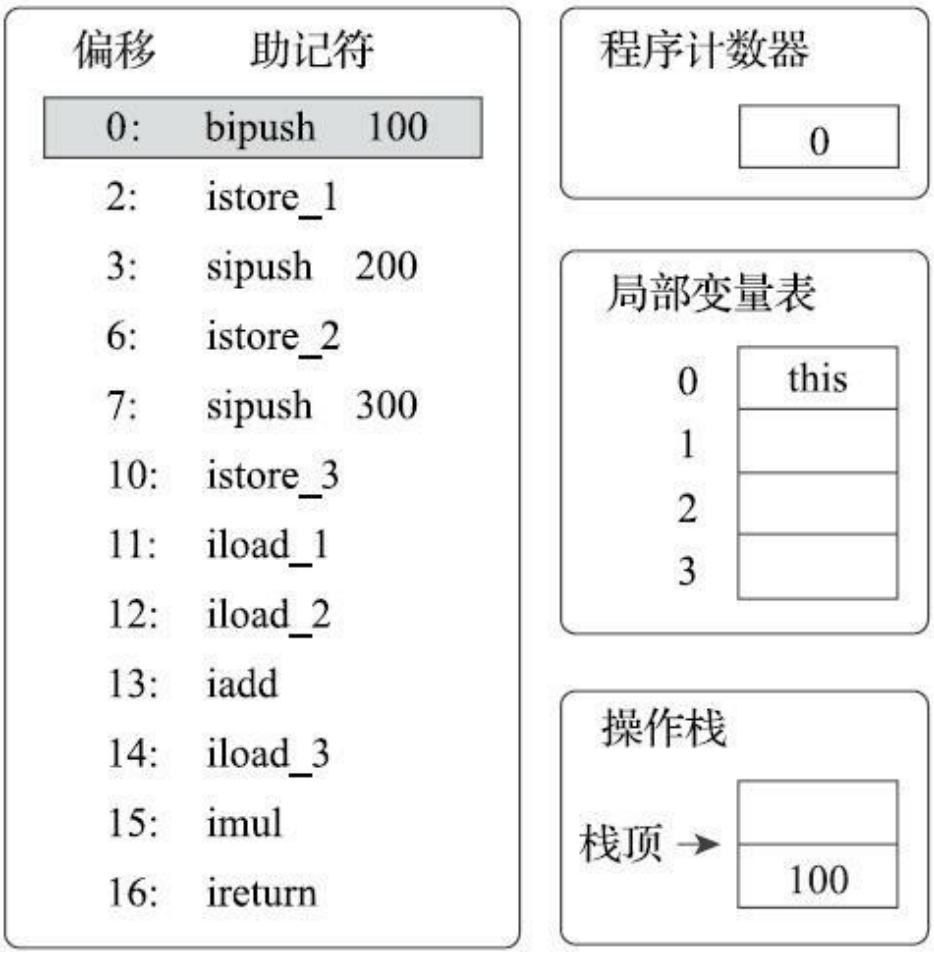
现在从一段 Java 代码来看栈架构的解释器执行过程：

```
public int calc(){
    int a = 100;
    int b = 200;
    int c = 300;
    return ( a + b ) * c;
}
```

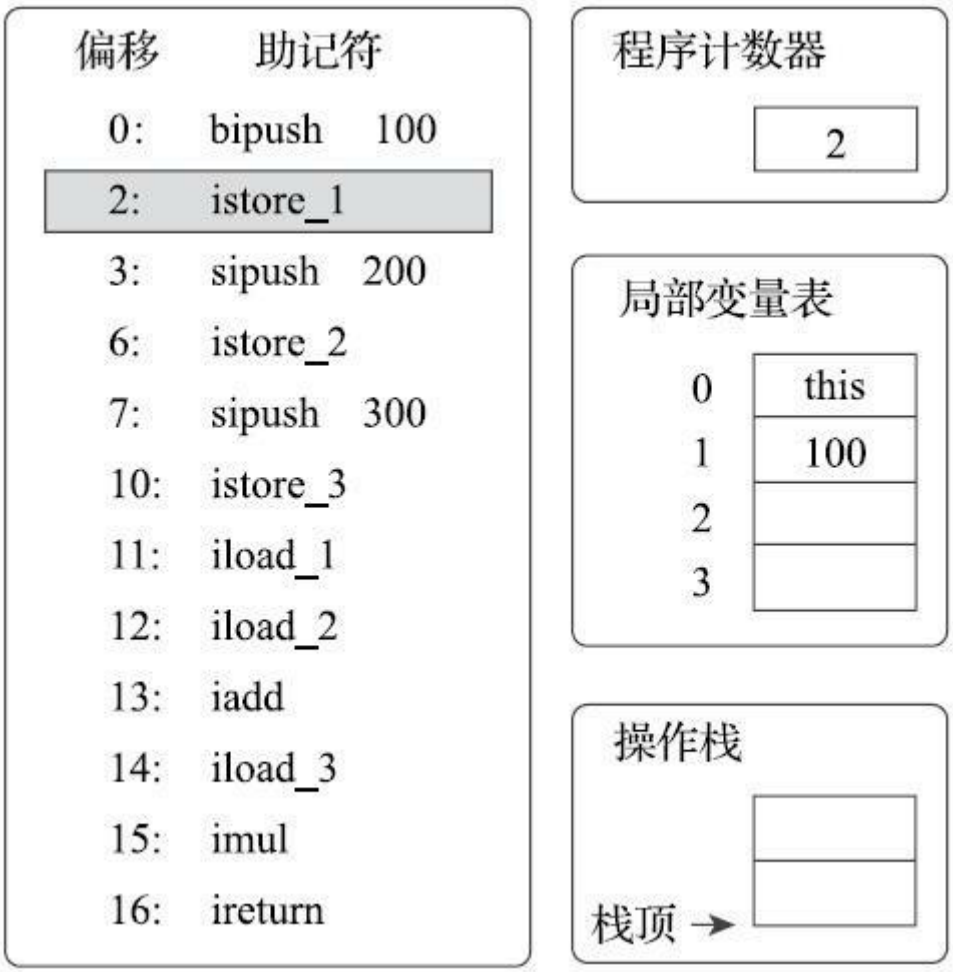
通过 javap 命令查看字节码指令：

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush 100
         2:   istore 1
         3:   sipush 200
         6:   istore_2
         7:   sipush 300
        10: istore_3
        11:  iload_1
        12:  iload_2
        13: iadd
        14: iload_3
        15: imul
        16: ireturn
}
```

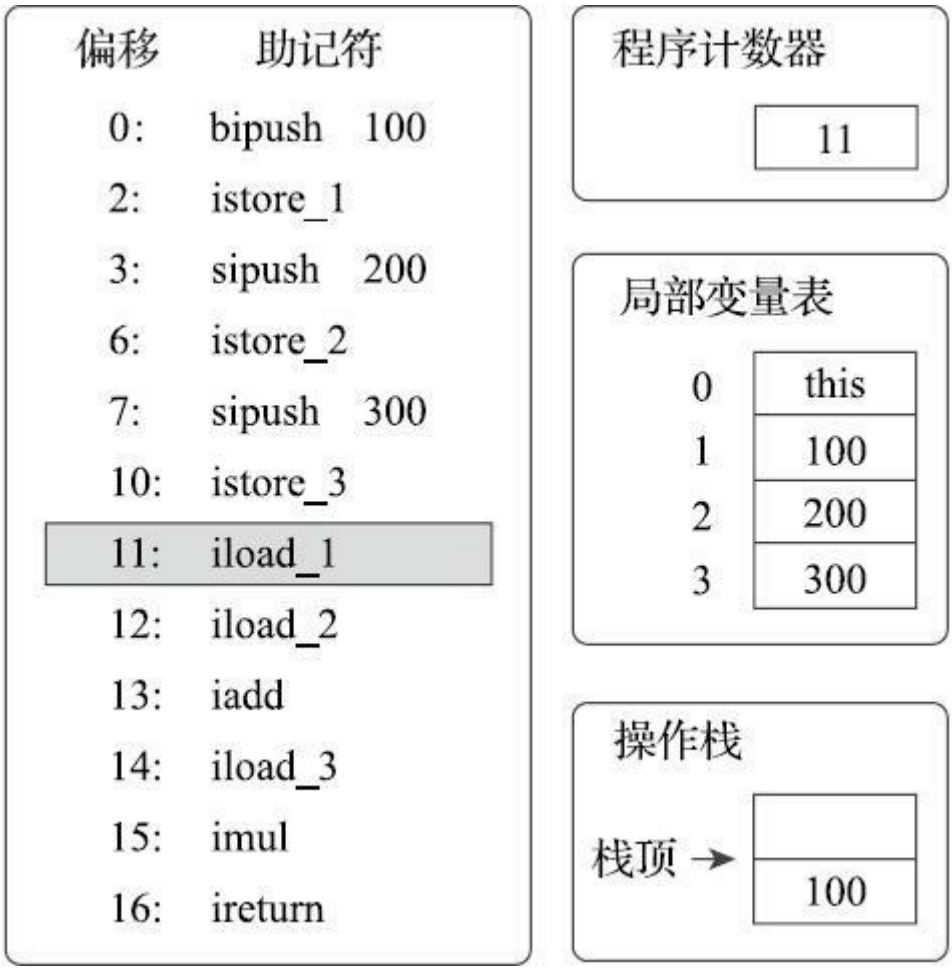
stack = 2 表示需要深度为 2 的操作数栈，Locals = 4 表示要 4 个变量槽的局部变量空间（这里再次证明了在编译期，操作数栈的大小以及局部变量表 slot 的空间就已经确定了）。后面的操作全跟着书上流程走：



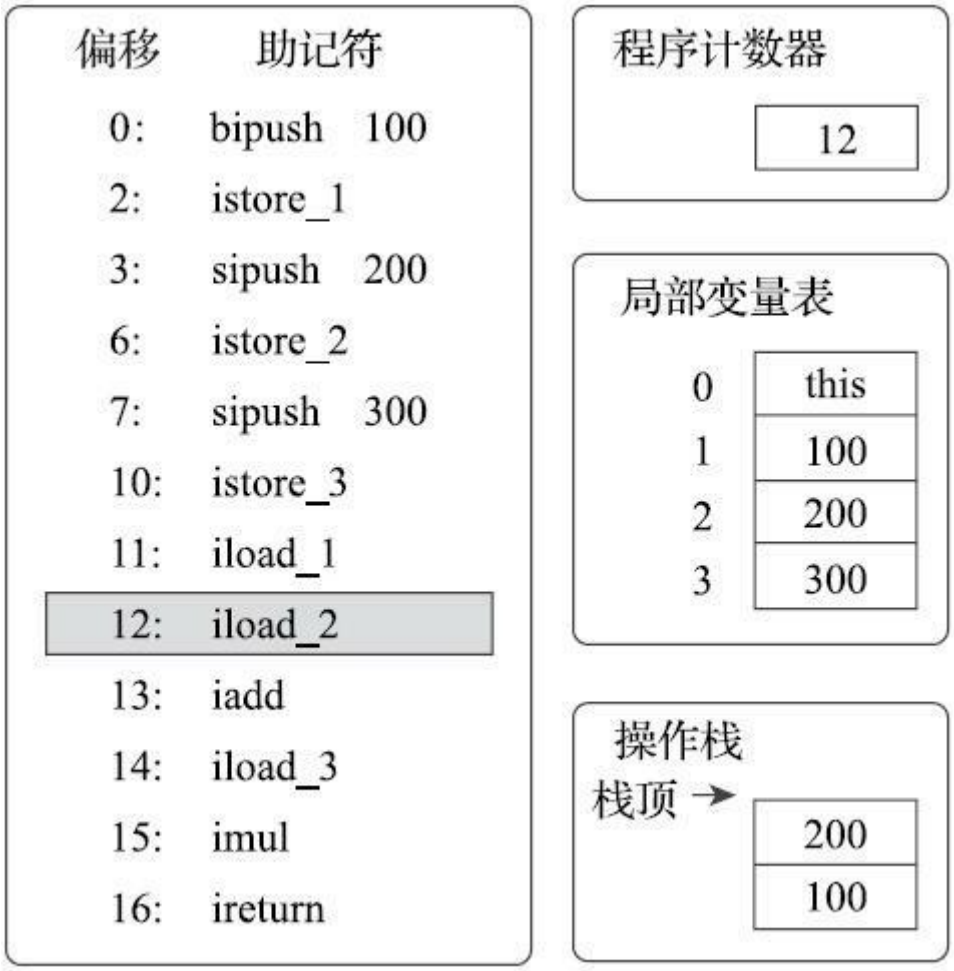
首先，执行偏移地址为 0 的指令（程序计数器的指向的 0 的位置，字节码解释器去改变值），Bipush 指令的作用是将单字节的整型常量值（-128～127）推入操作数栈顶，跟随有一个参数，指明推送的常量值，这里是 100。



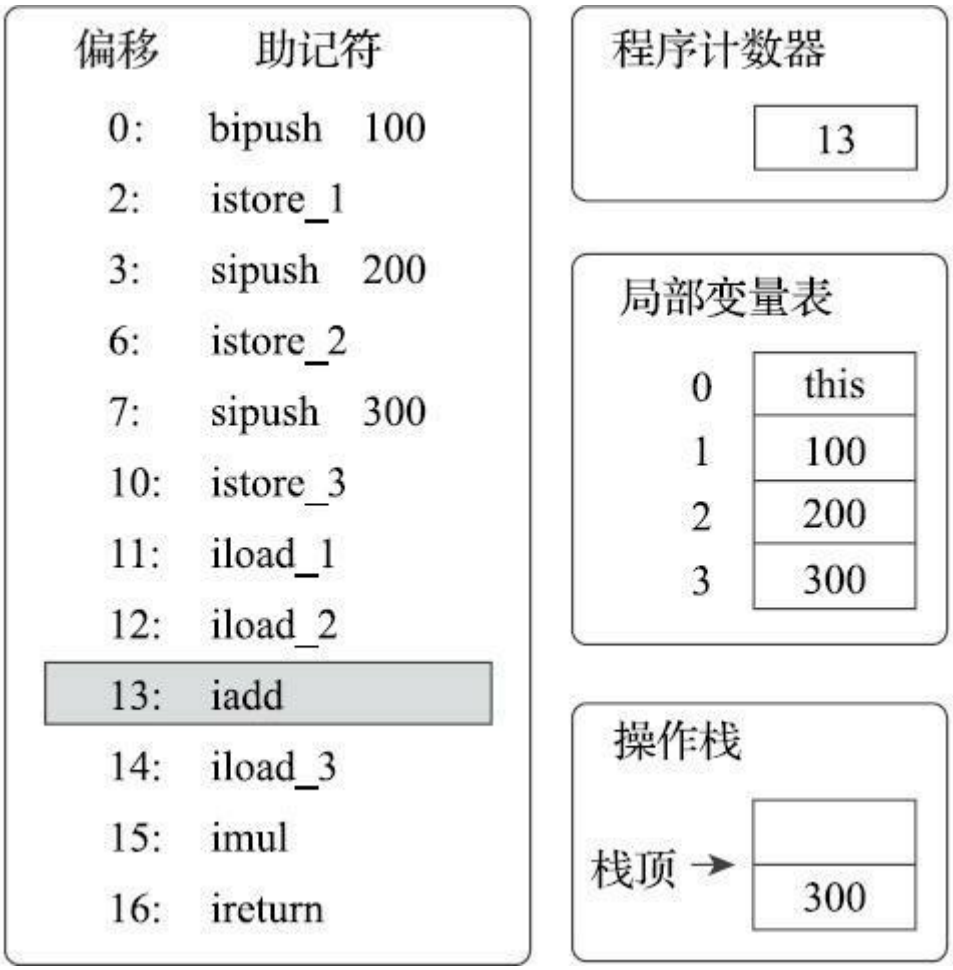
执行偏移地址为 2 的指令，istore_1 指令的作用是将操作数栈顶的整型值出栈并存放到第 1 个局部变量槽中。后续 4 条指令（直到偏移为 11 的指令为止）都是做一样的事情，也就是在对应代码中把变量 a、b、c 赋值为 100、200、300。这 4 条指令的图示略过。



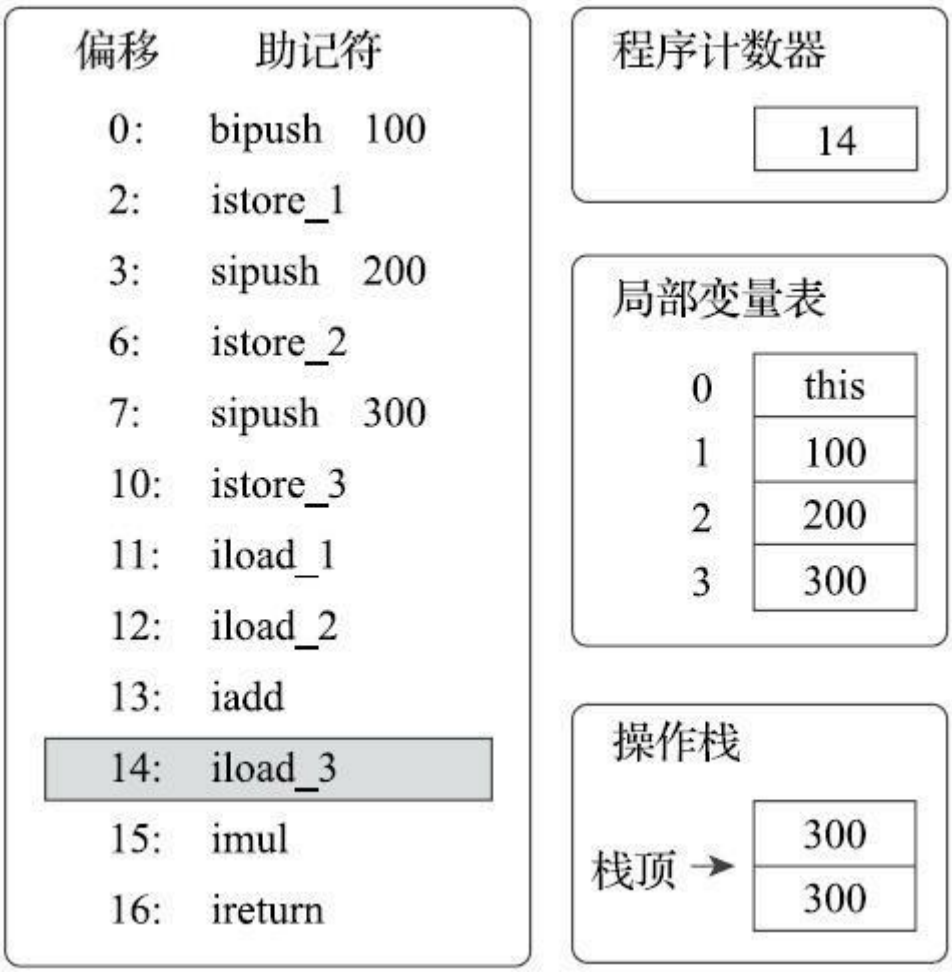
执行偏移地址为 11 的指令，iload_1 指令的作用是将局部变量表第 1 个变量槽中的整型值复制到操作数栈顶。



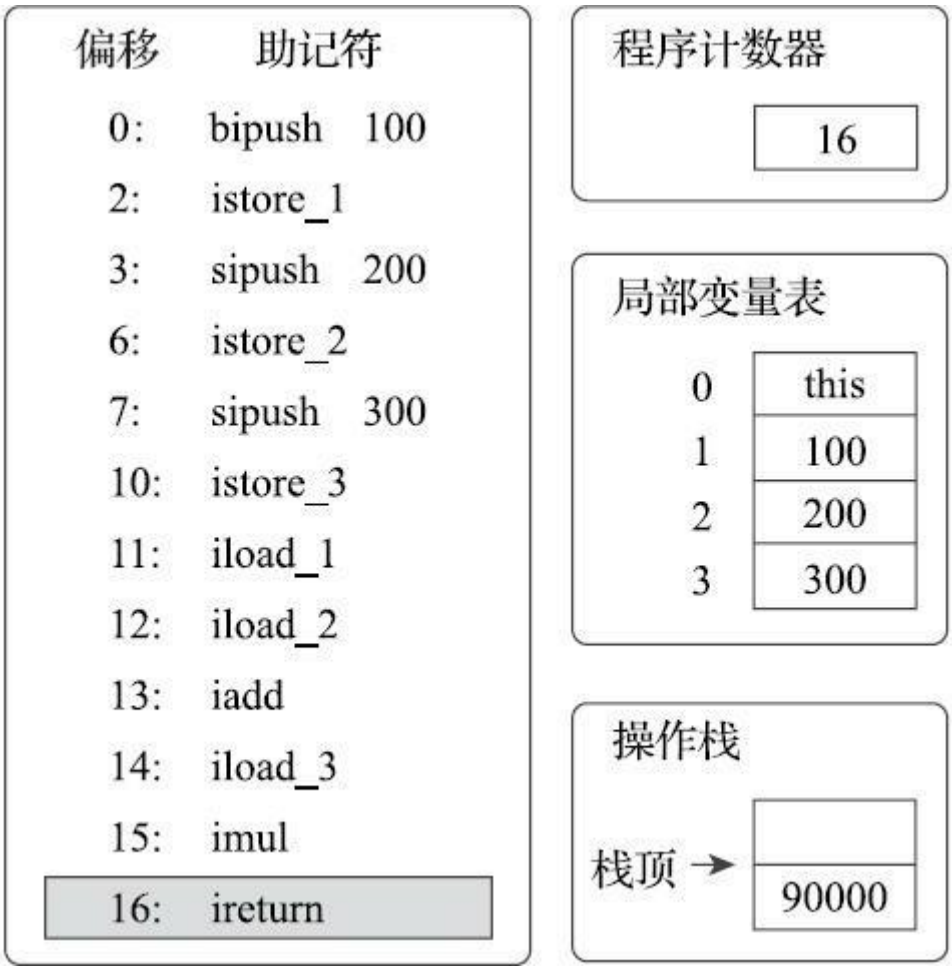
执行偏移地址为 12 的指令，iload_2 指令的执行过程与 iload_1 类似，把第 2 个变量槽的整型值入栈。画出这个指令的图示主要是为了显示下一条 iadd 指令执行前的堆栈状况。



执行偏移地址为 13 的指令，iadd 指令的作用是将操作数栈中头两个栈顶元素出栈，做整型加法， 然后把结果重新入栈。在 iadd 指令执行完毕后，栈中原有的 100 和 200 被出栈，它们的和 300 被重新入栈。



执行偏移地址为 14 的指令，iload_3 指令把存放在第 3 个局部变量槽中的 300 入栈到操作数栈中。这时操作数栈为两个整数 300。下一条指令 imul 是将操作数栈中头两个栈顶元素出栈，做整型乘法，然后把结果重新入栈，与 iadd 完全类似。后面省略。



执行偏移地址为 16 的指令，ireturn 指令是方法返回指令之一，它将结束方法执行并将操作数栈顶的整型值返回给该方法的调用者。到此为止，这段方法执行结束。

再次强调上面的执行过程仅仅是一种概念模型，虚拟机最终会对执行过程做出一系列优化来提高性能，实际的运作过程并不会完全符合概念模型的描述。更确切地说，实际情况会和上面描述的概念模型差距非常大，差距产生的根本原因是虚拟机中解析器和即时编译器都会对输入的字节码进行优 化，即使解释器中也不是按照字节码指令去逐条执行的。

总结

关于《深入理解 Java 虚拟机》这本书后面的 10 和 11 章，前端编译和后端编译由于内容比较繁琐，就不做总结了。至于后面的并发知识，我会再看一遍《Java 并发编程的艺术》这本书，然后以这本书为主再总结一遍。

全部总结仍然是以书为主，但是书写的很多太仔细了，很多概念我现在也没有理解到，所以想着用理解到的知识加上书上的语言去做一个总结。

做这个总结花了我一周，不过对于我来说也相当满意，书上的细节我基本上又过了一遍，相当于又复习了一遍。

电子书 732 页最后总结成 114 页的篇幅，有些细节我仍然加上了，这个知识点蓝色部分都是我认为比较重要的。有些虽然没有用蓝色表示，但是建议也看一看，对理解那些概念的东西会更加好。

最后，我可能会去再看一遍书，把书上的细节再抠出来仔细看看，再纠正下这个知识点文档，对于复习来说，这个知识点已经够了。