Chapter 1. 走近 Java

1.1. 概述

1.2. Java 技术体系

1、广义上讲，Clojure、JRuby、Groovy 等运行于 Java 虚拟机上的语言及其相关的程序都属于 Java 技术体系中的一员
2、Sun 官方定义的 Java 技术体系包括以下几个组成部分

- Java 程序设计语言
- 各种硬件平台上的 Java 虚拟机
- Class 文件格式
- Java API 类库
- 来自商业机构和开源社区的第三方 Java 类库

3、我们可以把 **Java 程序语言**、**Java 虚拟机**、**Java API 类库**这三部分统称为 JDK(Java Development Kit)，JDK 是用于支持 Java 程序开发的最小环境
4、我们可以把 Java API 类库中的 **Java SE API 子集**和 **Java 虚拟机**这两部分统称为 JRE(Java Runtime Environment)，JRE 是支持 Java 程序运行的标准环境
5、Java 技术体系可以分为 4 个平台，分别为：

- **Java Card**：支持一些 Java 小程序(Applets)运行在小内存设备(如智能卡)上的平台
- **Java ME(Micro Edition)**：支持 Java 程序运行在移动终端(手机、PDA)上的平台，对 Java API 有所精简，并加入了针对移动端的支持，以前称为 J2ME
- **Java SE(Standard Edition)**：支持面向桌面级应用(如 Windows 下的应用程序)的 Java 平台，提供了完整的 Java 核心 API，以前称为 J2SE
- **Java EE(Enterprise Edition)**：支持使用多层架构的企业应用(如 ERP、CRM 应用)的 Java 平台，除了提供 Java API 外，还对其做了大量的扩充并提供了相关的部署支持，以前称为 J2EE

1.3. Java 发展史

1、<未完成>

1.4. Java 虚拟机发展史

1.4.1. Sun Classic/Exact VM

1、Sun Classic VM 的技术可能很原始，这款虚拟机的使命也早已终结
2、世界上第一款商用 Java 虚拟机

1.4.2. Sun HotSpot VM

1、HotSpot VM 是 Sun JDK 和 OpenJDK 中所带的虚拟机，也是目前使用范围最广的 Java 虚拟机

1.4.3. Sun Mobile-Embedded VM/Meta-Circular VM

1、KVM

- KVM 中的 K 是 Kilobyte 的意思，他强调简单、轻量、高度可移植性，但是运行速度较慢
- 在 Android IOS 等智能手机操作系统出现前曾经在手机平台上得到非常广泛的应用

2、CDC/CLDC HotSpot Implementation

- CDC/CLDC 全名是 Connected(Limited)Device Configuration

3、Squawk VM

4、JavaInJava

5、Maxine VM

1.4.4. BEA JRockit/IBM J9 VM

1.4.5. Azul VM/BEA Liquid VM

1.4.6. Apache Harmony/Google Android Dalvik VM

1.4.7. Microsoft JVM 及其他

1.5. 展望 Java 技术的未来

1.5.1. 模块化
1.5.2. 混合语言
1.5.3. 多核并行
1.5.4. 进一步丰富语法
1.5.5. 64 位虚拟机

实战：自己编译 JDK

<未完成>

Chapter 2. Java 内存区域与内存溢出异常

1、Java 与 C++之间有一堵内存动态分配和垃圾收集技术所围成的"高墙"，墙外面的人想进去，墙里面的人却想出来

2.1. 概述

1、对于从事 C、C++程序开发的开发人员来说，在内存管理领域，既有最高权力，又从事最基础的工作---拥有每个对象的"所有权"，并担负起每个对象生命开始到终结的维护责任

2、对 Java 程序员来说，在虚拟机自动内存管理机制的帮助下，不再需要为每一个 new 操作去写配对的 delete/free 代码，不容易出现内存泄露和内存溢出的问题

2.2. 运行时数据区域

1、Java 虚拟机在执行 Java 程序的过程中会把它所管理的内存划分为若干个不同的数据区域

2.2.1. 程序计数器

1、程序计数器(Program Counter Register)是一块较小的内存空间，它可以看做是当前线程所执行的字节码的行号指示器

2、Java 虚拟机的多线程是通过线程轮流切换并分配处理执行时间的方式来实现的，在任何一个确定的时刻，一个处理器(内核)都只会执行一条线程中的命令

3、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个独立的程序计数器**，各条线程之间计数器互不影响，独立存储，称这类内存区域为"**线程私有**"

4、如果线程正在执行一个 Java 方法，这个**计数器记录的正式在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器值则为空**

5、**此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域**

2.2.2. Java 虚拟机栈

1、与程序计数器一样，Java 虚拟机栈(Java Virtual Machine Stacks)也是**线程私有**，它的生命周期与线程相同

2、**虚拟机栈描述的是 Java 方法执行的内存模型**：每个方法在执行的同时都会创建一个栈帧(Stack Frame)用于存储局部变量表，操作数栈，动态链接，方法出口等信息

3、每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程

4、**局部变量表**

- 存放了编译器可知的各种基本类型、对象引用和 returnAddress 类型
- 其中 64 位长度的 long 和 double 类型的数据会占用两个局部变量空间(Slot)，其余的数据类型只占用一个

- 局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，**在方法运行期间不会改变局部变量表的大小**

5、在 Java 虚拟机中，对这个区域规定了两种异常状况

- 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 **StackOverflowError** 异常
- 如果虚拟机栈可以动态扩展，当扩展时无法申请到足够内存时，就会抛出 **OutOfMemoryError** 异常

2.2.3. 本地方法栈

1、本地方法栈(Native Method Stack)与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是**虚拟机栈执行 Java 方法(也就是字节码)服务**，而**本地方法栈则为虚拟机使用到的 Native 方法服务**

2、在虚拟机规范中对本地方法栈中方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它

2.2.4. Java 堆

1、Java 堆(Java Heap)是 Java 虚拟机所管理的内存中最大的一块，Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建

2、**此内存区域唯一的目的就是存放对象实例**，几乎所有的对象实例都是在这里分配内存，但不是那么绝对

3、**Java 堆是垃圾收集器管理的主要区域**，因此很多时候也被称作"GC"堆(Garbage Collected Heap)

4、从内存回收的角度看，由于现在收集器基本都采用**分代收集算法**

- 所以 Java 堆中还可以细分为：**新生代**和**老年代**
- 再细致一点还可分为 **Eden 空间**、**From Survivor 空间**、**To Survivor 空间**

5、从内存分配的角度来看，线程共享的 Java 堆中可能划分出多个线程私有的**分配缓冲区(Thread Local Allocation Buffer，TLAB)**

6、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，只要逻辑上连续即可，就像磁盘空间一样

2.2.5. 方法区

1、方法区(Method Area)与 Java 对一样，是各个线程共享的内存区域，它用于存储已**被虚拟机加载的类信息**、**常量**、**静态变量**、**即时编译器编译后的代码等数据**

2、Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 **Non-Heap(非堆)**，**目的应该是与 Java 堆区分开来**

3、Java 虚拟机规范对方法区的限制非常宽松，除了和 Java 堆一样不需要连续的内存和可以选择固定大小或者扩展外，还可以选择不实现垃圾回收

- 相对而言，垃圾收集行为在这个区域是比较少出现的，**但并非数据进入了方法区就如永久代的名字一样"永久"存在**
- 这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载

4、当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常

5、方法区中存放了哪些东西

1) **类型信息**：对于每个加载的类型，jvm 必须在方法区中存储以下类型信息
   - 这个类型完整有效名(即全限定名，包含包名)
   - 这个类型直接父类的完整有效名(除非这个类型是 interface 或者是 Object，这两种情况下没有父类)
   - 这个类型的修饰符
   - 这个类型直接接口的一个有序列表
2) **类型的常量池**：jvm 为每个已加载的类型都维护一个常量池，常量池就是这个类型用到的常量的一个有序集合，包括实际的常量(string,integer, 和 float point 常量)和对类型，域和方法的符号引用
   - 池中的数据像数组一样，是通过索引访问的
   - **常量池存储了一个类型所使用到的所有类型，域和方法的符号引用，所以它在 java 程序的动态链接中起了核心作用**
3) **域(Field)信息**：jvm 必须在方法区中保存类型的所有域的相关信息以及域的声明顺序，域的相关信息包括
   - 域名
   - 域类型
   - 域修饰符(public,private,protected,static,final,volatile,transient 的子集)
4) **方法(Method)信息**：jvm 必须保存所有方法的相关信息以及方法的声明顺序，方法的相关信息包括
   - 方法名
   - 方法返回类型
   - 方法参数的数量和类型(有序)
   - 方法的修饰符(public,private,protected,static,final,synchronized,native, abstract 的子集)
   - 除了 abstract 和 native 方法外，其他方法还保存方法的字节码(bytecodes)和操作数栈和方法栈帧的局部变量区的大小
5) **除了常量外的所有静态(static)变量**：类变量被类的所有实例共享，即使没有类实例时你也可以访问它，这些变量只与类相关，在方法区中，它们成为类数据在逻辑上的一部分
6) **对类加载器的引用**：jvm 必须知道一个类型是由启动加载器加载的还是由用户类加载器加载的。如果一个类型是由用户类加载器加载的，那么 jvm 会将这个类加载器的一个引用作为类型信息的一部分保存在方法区中
7) **对 Class 类的引用**：jvm 为每个加载的类型都创建一个 java.lang.Class 的实例，而 jvm 必须以某种方式把 Class 的这个实例和存储在方法去中的类型数据关联起来

2. 2. 6. 运行时常量池

1、运行时常量池(Runtime Constant Pool)是方法区的一部分

2、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(Constant Pool Table)**，用于存放**编译期生成**的各种字面量和符号引用**(包括 String、字面值等)**，这部分内容将在类加载后进入方法区的运行时常量池中存放

```
String s1=new String("Hello");
String s2=new String("Hello");
String s3="Hello";
String s4="Hello";
System.out.println(s1==s2);//false
System.out.println(s3==s4);//true
```

3、Java 虚拟机对 Class 文件的每一部分(包括常量池)的格式都有严格规定，每一个字节用于存储那种数据都必须符合规范上的要求才会被虚拟机认可、装载和执行

4、对于运行时常量池，Java 虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域

5、运行时常量池相对于 Class 文件常量池的另一个重要特征是具备动态性

- Java 语言并不要求常量一定只有编译器才能产生，也就是**并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池**
- **运行期间也能将新的常量放入池中**

6、当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常

2.2.7. 直接内存

1、直接内存(Direct Memory)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

2、本机直接内存的分配不会受到 Java 堆大小的限制，但是会受到本机总内存大小以及处理器寻址空间的限制

3、无法再申请到内存时会抛出 OutOfMemoryError 异常

2.3. HotSpot 虚拟机对象探秘

2.3.1. 对象的创建

1、虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能够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被加载、解析和初始化过，如果没有，必须先执行相应的类加载过程

2、在类加载检查通过后，虚拟机将为新生对象分配内存，对象所需内存的大小在类加载完成后便可完全确定

3、分配方式

- **指针碰撞**：仅仅把指针向空闲空间那边挪动一段与对象大小相等的距离
- **空闲列表**：维护一个列表，记录哪块内存是可用的

4、除了划分可用空间外，还需要考虑的问题是：在并发情况下是否是线程安全的

- 方案一：对分配内存空间的动作进行同步处理---虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性
- 方案二：把内存分配的动作按照线程划分在不同的空间之中进行，每个线程在 Java 堆中预先分配一小块内存，称为本地分配缓冲(Thread Local Allocation Buffer，TLAB)，只有 TLAB 用完并分配新的 TLAB 时才需要同步锁定

5、内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值(不包括对象头)
6、接下来，虚拟机要对对象进行必要的设置
- 这个对象是哪个类的实例
- 如何才能找到类的元数据信息
- 对象的哈希码
- 对象的 GC 分代年龄等信息
- 以上信息存放在对象头(Object Header)中

7、以上工作完成后，**从虚拟机的角度来看**，一个新的对象已经产生了，**但从 Java 程序的角度来看**，对象的创建才刚刚开始，初始化方法还没有执行，所有的字段还都是 0

2.3.2. 对象的内存布局

1、在 HotSpot 虚拟机中，对象在内存中存储的布局可以分为 3 块区域：对象头(Header)、实例数据(Instance Data)和对齐填充(Padding)
2、HotSpot 虚拟机的头对象包括两部分信息
- 第一部分用于存储**对象自身的运行时数据**，如
  - 哈希码(HashCode)
  - GC 分代年龄
  - 锁状态标志
  - 线程持有的锁
  - 偏向线程 ID
  - 偏向时间戳等
  - 这部分数据的长度在 32 位和 64 位虚拟机中分别为 32bit 和 64bit，官方称它为"Mark Word"
  - 对象需要存储的运行时数据很多，已经超出了 32 位、64 位 Bitmap 结构所能记录的限度，**但是头对象信息是与对象自身定义的数据无关的额外存储成本**，考虑到虚拟机的空间效率，Mark Word 被设计成一个非固定的数据结构以便在极小的空间存储尽可能多的信息
- 另外一部分是**类型指针**，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例
  - 并不是所有虚拟机的实现都必须在对象数据上保留类型指针，换句话说，**查找对象的元数据信息并不一定要经过对象本身**
  - 如果对象是一个数组，那在对象头中还必须有一块用于记录数组长度的数据

3、实例数据
- 对象真正存储的有效信息，是在程序代码中定义的各种类型的字段内容
- 这部分的存储顺序会受到虚拟机分配策略参数和字段在 Java 源码中定义顺序的影响
- HotSpot 默认的分配策略为：longs/doubles、ints、shorts/chars、bytes/booleans、oops(Ordinary Object Pointers)
  - 相同宽度的字段总是被分配到一起
  - 在满足上面一条的情况下，父类中定义的变量会出现在子类之前

4、对齐填充

- 并不是必然存在的，也没有特别的含义，仅仅起着占位符的作用
- 由于 HotSpot VM 自动内存管理系统要求对象起始地址必须是 8 字节的整倍数，即对象的大小必须是 8 字节的整倍数
- 而头对象部分正好是 8 字节的倍数(1 倍或 2 倍)，当对象实例数据部分没有对齐时，就需要对齐填充来补全

2.3.3. 对象的访问定位

1、Java 程序需要通过**栈**上的 **reference** 数据来操作**堆**上的**具体对象**

2、由于 reference 类型在 Java 虚拟机规范中只规定了一个指向对象的引用，并没有定义这个引用应该通过何种方式去定位、访问堆中的对象的具体位置，对象**访问方式**取决于虚拟机的具体实现，**主流有两种方式**

- **句柄**：
  - **Java 堆**中会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的具体地址信息
  - 使用句柄访问的最大好处就是 reference 中存储的是稳定的句柄地址，在对象被移动(**垃圾收集时移动对象是非常普遍的行为**)时只会改变句柄中的实例数据指针，而 reference 本身不需要修改
- **指针**：
  - Java 堆对象的布局就必须考虑如何放置访问类型数据的相关信息
  - 使用指针最大的好处就是速度快，它节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本
  - HotSpot 使用的是指针

2.4. 实战：OutOfMemoryError 异常

1、简称 OOM

2、本节的目的：

- 通过验证 Java 虚拟机规范中描述的各个运行时区域存储的内容
- 希望读者在工作中遇到实际的内存溢出异常时，能根据异常的信息快速判断是哪个区域的内存溢出，知道什么样的代码会导致这些区域内存溢出，以及出现这些异常后该如何处理

2.4.1. Java 堆溢出

1、Java 堆用于存储对象实例，只要不断地创建对象，并且保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，在对象数量到达最大堆的容量限制后就会产生内存溢出的异常

2、限制虚拟机堆的大小

- -Xms：堆容量最小值
- -Xmx：堆容量最大值
- 将-Xms 与-Xmx 设置为一样可避免堆自动扩展

3、让虚拟机在出现内存溢出异常时 Dump 出当前的内存堆转储快照以便事后进行分析

- -XX: +HeapDumpOnOutOfMemoryError

4、因此，总的 VM 参数设置为：

- -Xms20m -Xmx20m -XX:+HeapDumpOnOutOfMemoryError
- 右键.java 文件，Run as/Debug as -- Run/Debug Configuration

5、Java 堆内存的 OOM 异常是实际应用中常见的内存溢出异常情况

6、解决这个区域的异常

- 一般的手段是先通过内存映像分析工具(如 Eclipse Memory Analyzer)对 Dump 出来的堆转储快照进行分析
- 重点是确认内存中的对象是否是必要的

2.4.2. 虚拟机栈和本地方法栈溢出

1、由于 HotSpot 虚拟机中并不区分虚拟机栈和本地方法栈

- 因此对于 HotSpot 来说，虽然-Xoss 参数(设置本地方发栈大小)存在，但实际上是无效的
- 栈容量只由-Xss 参数设定

2、Java 虚拟机规范中描述了两种异常

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常
- 如果虚拟机栈在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常
- 这里把异常分成两种情况，看似更严谨，实际有重合的地方，当栈空间无法继续分配时，到底是内存太小还是已使用的栈空间太大，**本质上是对同一件事情的两种描述**

3、在单线程下，无论是由于栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是 StackOverflowError 异常

2.4.3. 方法区和运行时常量池溢出

1、**运行时常量是方法区的一部分，JDK 1.7 开始逐步"去永久代"**

2、String.intern()是一个 Native 方法，它的作用是：如果字符串常量池中已经包含了一个等于此 String 对象的字符串，则返回代表池中这个字符串的 String 对象，否则将此 String 对象包含的字符串添加到常量池中，并返回此 String 对象的引用

3、在 JDK 1.7 之后 String.intern()的实现有了改变

- 在 JDK 1.6 中，intern()方法会**把首次遇到的字符串实例复制到永久代中**，返回的也是永久代中这个字符串实例的引用
- 在 JDK 1.7 中，"去永久代"，intern()的实现**不再复制实例，而是在常量池中记录首次出现实例的引用**

4、方法区用于存放 Class 的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述

- 对于该区域的测试，基本思路是运行时产生大量的类去填满方法区，直到溢出
- 直接使用 Java SE API 可以产生动态类(如反射时的 GeneratedConstructorAccessor 和动态代理等)

- 可以借助 **CGLib(开源项目)**直接操作字节码运行时生成了大量动态的类
- 这样的应用场景经常会出现在实际应用中：当前很多主流框架，如 String、Hibernate，在对类进行增强时，都会使用到 CGLib 这类字节码计数，增强的类越多，就需要越大的方法区来保护动态生成的 Class 可以加载入内存

2.4.4. 本机直接内存溢出

1、DirectMemory 容量可以通过-XX:MaxDirectMemorySize 指定，如果不指定，则默认与 Java 最大值(-Xmx 指定)一样

2、<未完成>

Chapter 3. 垃圾收集器与内存分配策略

3.1. 概述

1、垃圾收集器(Garbage Collection,GC)

- GC 的历史比 Java 久远
- GC 需要完成的三件事情
  - 那些内存需要回收
  - 什么时候回收
  - 如何回收

2、目前内存的动态分配与内存回收技术已经相当成熟，为何还要了解 GC 和内存分配？

- 当需要排查各种内存溢出，内存泄露问题时，当垃圾收集成为系统达到更高并发量的瓶颈时，我们就需要对这些"自动化"的技术实施必要的监控和调节

3、**程序计数器、虚拟机栈、本地方法栈**

- 这 3 个区域随线程而生，随线程而灭
- 每个栈帧分配多少内存基本上是在类结构确定下来的时候就已知的(尽管在运行期会由 JIT 编译器进行一些优化，但大体上可认为是已知的)
- **这几个区域的内存分配和回收都具备稳定性**，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或线程结束时，内存自然就跟着回收了

4、Java 堆

- Java 堆和方法区则不一样，一个接口中多个实现类需要的内存可能也不一样
- 一个方法中的多个分支需要的内存可能也不一样，我们只有在程序运行时才能知道会创建那些对象
- 这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分内存

3.2. 对象已死吗

1、在堆里存放着 Java 世界中几乎所有的对象实例，垃圾收集器在堆进行回收前，第一件事情就是要确定这些对象之中**哪些还存活着**，**哪些已经死去**(不可能在被任何途径使用的对象)

3.2.1. 引用计数算法

1、很多教科书判断对象是否存活的算法：给对象添加一个引用计数器

- 每当有一个地方引用它，计数器就加 1
- 当引用失效时，计时器值减 1
- 任何时刻计数器为 0 的对象就是不可能再被使用的

2、客观地说，引用计数算法的实现简单，判定效率也很高，在大部分情况下都是一个不错的算法，著名案例有

- 微软的 COM(Component Object Model)技术

- 使用 ActionScript 3 的 FlashPlayer
- Python 语言
- 在游戏脚本领域被广泛应用的 Squirrel

3、**但是 Java 虚拟机没有选用引用计数来管理内存，主要原因是它难以解决对象之间相互循环引用的问题**

3.2.2. 可达性分析算法

1、在主流的商用程序语言(Java、C#，包括古老的 Lisp)的主流实现中，都是称通过**可达性分析(Reachability Analysis)**来判定对象是否存活

- 这个算法的基本思路就是通过一系列的称为"**GC Roots**"的对象作为起始点，从这些节点开始向下搜索
- 搜索所走过的路径称为**引用链(Reference Chain)**，当一个对象到 GC Roots 没有任何引用链项链时，证明对象不可用

2、在 Java 语言中，**GC Roots 的对象包括以下几种**

- 虚拟机栈(栈帧中本地变量表)中引用的对象
- 方法区中类静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中 JNI(即一般说的 Native 方法)引用的对象

3.2.3. 再谈引用

1、无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象的引用链是否可达，判定对象是否存活都与"引用"有关

2、JDK 1.2 之前，Java 中的引用的定义很传统：

- 如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表一个引用
- 在这种定义下，只有被引用或没有被引用两种状态

3、JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为

- **强引用(String Reference)**：程序代码中普遍存在，类似 Object obj=new Object()这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象
- **软引用(Soft Reference)**：描述一些还有用但并非必须的对象，对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收返回之中进行第二次回收，如果这次回收还没有足够的内存，将抛出内存溢出的异常
- **弱引用(Weak Reference)**：描述非必须对象，被弱引用关联的对象只能生存到下一次垃圾收集发生之前，即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
- **虚引用(Phantom Reference)**：
- 这四种引用强度依次减弱：一个对象是否有虚引用的存在，完全不会对其生存事件构成影响，也无法通过虚引用来取得一个对象实例，设置虚引用关联的唯一目的：在这个对象被收集器回收时收到一个系统通知

3.2.4. 生存还是死亡

1、即使在可达性分析算法中不可达的对象，也并非是"非死不可"，这时候它们暂

时处于"缓刑"阶段

2、要真正宣告一个对象死亡，至少要经历两次标记过程：

- 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并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize()方法，当对象没有覆盖 finalize()方法或者 finalize()方法已经被虚拟机调用过，虚拟机将这两种情况视为"没有必要执行"
- 如果这个对象被判定为有必要执行 finalize()方法，那么这个对象将会放置在一个叫做 F-Queue 的队列之中，并在稍后由一个虚拟机自动建立、低优先级的 Finalizer 线程去执行它
  - 这里的执行指虚拟机会触发这个方法，但并不承诺会等待它运行结束
  - 这样做的原因是：如果一个对象在 finalize()方法中执行缓慢，或者发生了死循环，这可能导致 F-Queue 队列中其他对象永久处于等待，导致整个内存回收系统崩溃
- finalize()方法是对象逃脱死亡命运的最后一次机会
  - 稍后 GC 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(比如一个静态域赋值为该对象的 this)
  - 譬如把自己(this)赋值给某个类变量或者对象的成员变量，那么在第二次标记时将它移出"即将回收"的集合
- 并不鼓励使用 finalize()来拯救对象
  - 因为 finalize()并不等同于 C++中的析构函数
  - finalize()运行的代价高昂，不确定性大
  - 对于 finalize()能做的工作，使用 try-finally 语句会更好、更及时
  - 甚至可以忘掉有 finalize()这种语法

3.2.5. 回收方法区

1、很多人认为方法区(或者 HotSpot 虚拟机中的永久代)是没有垃圾收集的

- Java 虚拟机规范不要求虚拟机在方法区实现垃圾收集
- 而且在方法区中进行垃圾收集"性价比"一般比较低：在堆中
  - 在新生代中，常规应用进行一次垃圾收集一般可回收 70%-90%的空间
  - 永久代的垃圾收集效率远低于此

2、永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类

- 回收废弃常量与回收 Java 堆中的对象非常类似
- 判定一个常量是否废弃很简单，即判断该常量是否被引用
- 判定一个类是否无用较为苛刻，需要满足 3 个条件：
  - 该类所有的实例都已经被回收
  - 加载该类的 ClassLoader(类加载器)已经被回收
  - 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
  - 虚拟机可以对满足上述 3 个条件的无用类进行回收，仅仅是可以，并不像对象一样，不使用了就必然会回收(对象必然会回收???)
  - 是否对类进行回收，HotSpot 虚拟机提供了-Xnoclassgc 参数进行控制，还可以使用 -verbose:class 、 -XX:+TraceClassLoading 、 -XX:TraceClassUnLoading 查看类加载和卸载信息

- 在大量使用反射、动态代理、CGLib 等 ByteCode 框架、动态生成 JSP 以及 OSGi 这类频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出

3.3. 垃圾收集算法

3.3.1. 标记-清除算法

1、最基础的收集算法是"标记-清除"(Mark-Sweep)算法，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
- 在标记完后统一回收所有被标记的对象

2、不足之处：

- 效率问题：标记和清除两个过程的效率都不高
- 空间问题：标记清除之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

3.3.2. 复制算法

1、为了解决效率问题，一种称为"复制"(Copying)的收集算法出现了

- 它将可用内存按容量划分为大小相等的两块，每次只使用其中一块
- 当一块的内存用完了，将还存活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

2、这使得每次都是对整个半区进行内存回收，内存分配时也不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效

3、代价：将可用内存缩小为原来的一半

4、现在商业虚拟机都采用这种收集算法来回收新生代，IBM 公司专门研究表明，新生代中的对象 98%是"朝生夕死"，因此并不需要按 1：1 的比例来划分内存空间

- 将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间
- 每次使用 Eden 和其中一块 Survivor
- 当回收时，将 Eden 和 Survivor 中还存活着的对象一次性复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间
- HotSpot 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%(80%+10%)，只有 10%的空间会被"浪费"
- 98%是一般场景下的数据，我们没法保证每次回收都只有不多于 10%的对象存活，当 Survivor 不够用时，需要依赖其他内存(这里指老年代)进行分配担保(Handle Promotion)

3.3.3. 标记-整理算法

1、复制收集算法的不足：

- 在对象存活率较高时就要进行较多的复制操作，效率将会变低
- 要想不浪费 50%的空间，就需要有额外的空间进行分配担保

2、根据老年代的特点，提出了"标记-整理"(Mark-Compact)算法

- 标记过程仍然与"标记-清除"算法一样
- 但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

3.3.4. 分代收集算法

1、当前商业虚拟机的垃圾收集都采用"分代收集"(Generational Collection)算法

- 这种算法并没有什么新的思想，**只是根据对象存活周期的不同将内存划分为几块**
- **一般把 Java 堆分为新生代和老年代**，这样就可以根据各个年代的特点采用最适当的收集算法
- **在新生代中**：每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用**复制算法**，只需要付出少量存活对象的复制成本就可以完成收集
- **在老年代中**：对象存活率高，没有额外空间对它进行分配担保，就必须使用**"标记-清理"**或者**"标记-整理"**算法进行回收

3.4. HotSpot 的算法实现

3.4.1. 枚举根节点

1、从可达性分析中从 GC Roots 节点找到引用链这个操作为例，可作为 GC Roots 的节点主要在**全局性的引用(例如常量或静态属性)**与**执行上下文(例如栈帧中的本地变量表)中**，现在很多应用仅仅方法区就有数百兆，如果要逐个检查这里面的引用，那么必然会消耗很多时间

2、另外，可达性分析对执行时间的敏感还体现在 **GC 停顿**上

- 因为这项分析工作必须在一个能确保一致性的快照中进行，**即不可以出现分析过程中对象引用关系还在不断变化的情况**
- 若不满足的话，则分析结果将会不准确
- 这点是导致 GC 进行时必须停顿所有 Java 执行线程(Sun 将这件事称为"Stop The World")的其中一个重要原因
- 即使是在号称(几乎)不会发生停顿的 GMS 收集器中，枚举根节点时也必须要停顿

3、目前主流 Java 虚拟机使用的都是**准确式 GC(虚拟机可以知道内存中某个位置的数据具体是什么类型)**

- 当执行系统停顿下来后，并不需要一个不漏的检查完所有执行上下文和全局的引用位置
- **在 HotSpot 的实现中，是使用一组称为 OopMap 的数据结构来达到这个目的**，在类加载完成的时候，HotSpot 就把对象内什么偏移量上是什么类型的数据计算出来，在 JIT 编译过程中，也会在特定的位置记录下栈和寄存器中哪些位置是引用

3.4.2. 安全点

1、在 OopMap 的协助下，HotSpot 可以快速且准确地完成 GC Roots 枚举

2、随之而来的问题：可能导致引用关系变化，或者说 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，那会需要大量的额外空间，这样 GC 的空间成本就会很高

3、实际上，HotSpot 确实没有为每条指令都生成 OopMap，只是在"特定位置"记录了这些信息，这些位置称为"安全点"，**即程序执行时并非在所有地方都能停顿下来开始 GC，只有在到达安全点时才能暂停**

4、Safepoint 的选定

- 既不能太少以至于让 GC 等待时间太长
- 也不能太多以至于过分增大运行时的负荷
- 安全点的选定基本上是以程序"**是否具有让程序长时间执行的特征**"为标准进行选定
- **"长时间执行"最明显的特征就是：指令序列复用**，例如方法调用、循环跳转、异常跳转等

5、另一个要考虑的问题：如何在 GC 发生时让所有线程(不包括执行 JNI 调用的线程)都跑到最近的安全点上再停顿下来，有两种方案可供选择

- **抢先式中断(Preemptive Suspension)**：不需要线程的执行代码主动去配合，在 GC 发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就恢复线程，让它跑到安全点上，**几乎没有虚拟机采用抢先式中断来暂停线程从而响应 GC 事件**
- **主动式中断(Voluntary Suspension)**：当 GC 需要中断线程的时候，不直接对线程操作，仅仅简单的设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起，**轮询标志的地方和安全点是重合的**，

3.4.3. 安全区域

1、使用 SafePoint 似乎应完美解决了如何进入 GC 的问题，但并非如此

- Safepoint 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 GC 的 Safepoint
- 但程序不执行的时候，例如处于 Sleep 状态或者 Blocked 状态，这时候线程无法响应 JVM 的中断请求，"跑"到安全的地方去中断挂起，此时就需要**安全区域(Safe Regin)**来解决

2、安全区域：指一段代码片段之中，引用关系不会发生变化，在这个区域中任何地方开始 GC 都是安全的，我们可以把 Safe Regin 看作是扩展的 Safepoint

- 在线程执行到 Safe Regin 中的代码时，首先**标志自己已经进入了 Safe Regin**，在这段时间里 JVM 发起 GC 时，就不要管标志自己为 Safe Region 状态的线程
- 当线程要离开 Safe Regin 时，它要检查系统是否已经完成了根节点枚举(或者整个 GC 过程)，如果完成了，线程就继续执行，否则必须等待直到收到可以安全离开 Safe Regin 的信号为止

3.5. 垃圾收集器

1、收集算法是内存回收的方法论

2、垃圾收集器就是内存回收的具体实现

3、Java 虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同厂商、不同版本的虚拟机所提供的垃圾收集器可能会有很大差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器

4、我们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，因为直到现在还没有最好的收集器出现，更加没有万能的收集器

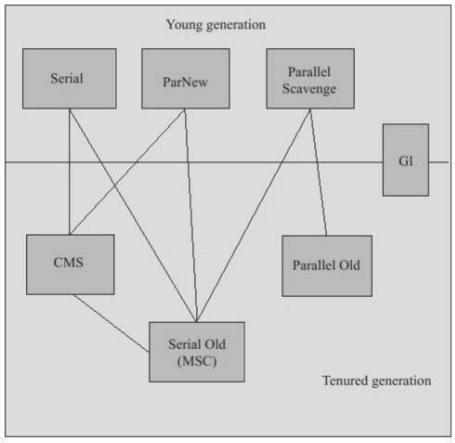

图 3-1 HotSpot 虚拟机的垃圾收集器

1) 两个收集器之间存在连线，就说明它们可以搭配使用

3.5.1. Serial 收集器

1、Serial 收集器是最基本，发展历史最悠久的收集器，在 JDK 1.3.1 之前是虚拟机新生代收集的唯一选择

2、这个收集器是一个**单线程**的收集器，这里单线程的意义并：

- **不仅仅说明它只会使用一个 CPU 或一条收集线程去完成垃圾收集工作**
- **更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束**
- 可以戏称为"Stop the World"

3、这项收集工作实际上由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说是难以接受的

4、对于" Stop the World "带给用户的不良体验，虚拟机的设计者们表示完全理解，但也表示非常委屈"**你妈妈在给你打扫房间的时候，肯定也会让你老老实实在椅子上或者房间外待着，如果你一边打扫，一边乱扔纸屑，这房间还能打扫完?**"

5、从 JDK 1.3 开始，一直到现在 JDK 1.7，HotSpot 虚拟机开发团队为消除或者减少工作线程因内存回收而导致停顿的努力一直在进行着

- 从 Serial 到 Parallel，再到 Concurrent Mark Sweep(CMS)乃至 GC 收集器的最前沿成果 Garbage First(G1)收集器
- **用户线程的停顿时间在不断缩短，但是仍然没有办法完全消除**

6、到目前为止，Serial 收集器依然是虚拟机运行在 Client 模式下的默认新生代收集器，它优于其他收集器的地方：简单而高效

7、JVM 模式

- JVM Server 模式与 JVM Client 模式启动，最主要的差别在于：-Server 模式**启动时，速度较慢，但是一旦运行起来后，性能将会有很大的提升**。JVM 如果不显式指定是-Server 模式还是-client 模式，JVM 能够进行自动判断(适用于 Java5 版本或者 Java 以上版本)
- VM 工作在 Server 模式可以大大提高性能，但应用的启动会比 client 模式慢大概 10%。当该参数不指定时，虚拟机启动检测主机是否为服务器，如果是，则以 Server 模式启动，否则以 client 模式启动，J2SE5.0 检测的根据

是至少 2 个 CPU 和最低 2GB 内存。

- 当 JVM 用于启动 GUI 界面的交互应用时适合于使用 client 模式，当 JVM 用于运行服务器后台程序时建议用 Server 模式。
- JVM 在 client 模式默认-Xms 是 1M，-Xmx 是 64M；JVM 在 Server 模式默认-Xms 是 128M，-Xmx 是 1024M。我们可以通过运行:java -version 来查看 jvm 默认工作在什么模式。

3.5.2. ParNew 收集器

1、ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括 Serial 收集器可用的所有控制参数

2、ParNew 收集器除了多线程收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关的重要原因：除了 Serial 外，只有它能与 CMS 收集器配合

3、CMS 收集器：

- 在 JDK 1.5 时期，HotSpot 推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器--CMS(Concurrent Mark Sweep)
- **这款收集器是 HotSpot 虚拟机中第一款真正意义上的并发(Concurrent)收集器，第一次实现了让垃圾收集线程与用户线程(基本上)同时工作**
- 使用 CMS 来收集老年代的时候，新生代只能选择 ParNew 或者 Serial 收集器中的一个，**因为 CMS 无法与新生代收集器 Parallel Scavenge 配合工作**

4、ParNew 收集器在单 CPU 的环境中绝对不会比 Serial 收集器有更好的效果，甚至由于线程交互的开销，在两个 CPU 的环境下也不能保证比 Serial 更好，但是随着 CPU 数量继续上升，ParNew 的优势会逐渐明显起来

5、名词

- 并行(Parallel)：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态
- 并发(Concurrent)：用户线程与垃圾收集线程同时执行(但不一定是并行的，可能交替执行)，用户程序在继续运行，而垃圾收集程序运行于另一个 CPU 上

3.5.3. Parallel Scavenge 收集器

1、Parallel Scavenge 收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器

2、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
- Parallel Scavenge 收集器的目标则是达到一个可控制的吞吐量
- 所谓**吞吐量**就是：CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即**吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)**

3、Parallel Scavenge 收集器提供两个参数用于精确控制吞吐量

- **控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis 参数**
  - MaxGCPauseMillis 参数允许设置一个大于 0 的毫秒数，收集器将尽可能地保证内存回收话费的时间不超过设定值
  - **不要认为**把这个参数设置得小一点就能使系统的垃圾收集速度变快
  - **GC 停顿时间缩短是以牺牲吞吐量和新生代空间来换取的**

- 例如收集 300MB 新生代肯定比收集 500MB 快，这也导致垃圾收集发生的更频繁一点，原来 10 秒收集一次，每次停顿 100ms，现在 5 秒收集一次，每次停顿 70ms，导致吞吐量下降

➢ **直接设置吞吐量大小的-XX:GCTimeRatio 参数**

- 该参数的值应当是一个大于 0 且小于 100 的整数
- 若设置为 19，则允许的最大 GC 时间就占总时间的%5=1/(1+19)
- 默认为 99，即 1%=1/(1+99)

4、由于与吞吐量关系密切，Parallel Scavenge 收集器也经常称为"吞吐量优先"收集器，除上述两个参数之外，Parallel Scavenge 收集器还有一个参数-XX:+UseAdaptiveSizePolicy 值得关注

➢ 这是一个开关参数，当打开这个参数后，就不需要手工指定新生代的大小、Eden(-Xmn)与 Survivor 区的比例(-XX:SurvivorRatio)，晋升老年代对象大小等细节参数了，**虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为 GC 自适应的调节策略(GC Ergonomics)**

➢ 自适应调节策略也是 Parallel Scavenge 收集器与 ParNew 收集器的一个重要区别

3.5.4. Serial Old 收集器

1、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

2、这个收集器的主要意义也是在给 Client 模式下的虚拟机使用

3、在 Server 模式下，那么它主要还有两大用途：一种用途是在 JDK 1.5 以及之前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途就是作为 CMS 收集器的后备方案，在并发收集发生 Concurrent Mode Failure 时使用

3.5.5. Parallel Old 收集器

1、Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和"标记-整理"算法

2、这个收集器在 JDK 1.6 中才开始提供，在此之前，新生的的 Parallel Scavenge 收集器一直处于比较尴尬的状态，原因是：**如果选择了 Parallel Scavenge 收集器，老年代除了 Serial Old(PS MarkSweep)收集器外别无选择，Parallel Scavenge 收集器无法与 CMS 收集器配合工作**

3、由于老年代 Serial Old 收集器在服务端应用性能上的拖累，使用了 Parallel Scavenge 收集器也未必能在整体应用上获得吞吐量最大化的效果，**由于单线程的老年代收集无法充分利用服务器多 CPU 的处理能力**，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS 的组合"给力"

4、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应用组合，在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器

3.5.6. CMS 收集器(老年代)

1、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 应用集中在互联网或者 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验，CMS 收集器就非常符合这类应用的需求

2、从名字"Mark Sweep"上就可以看出，**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集中收集器来说更复杂一点，过程分为 4 个步骤

- **初始标记(CMS initial mark)**
- **并发标记(CMS concurrent mark)**
- **重新标记(CMS remark)**
- **并发清除(CMS concurrent sweep)**
- **初始标记**，**重新标记**这两个步骤仍然需要"Stop The World"
- **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
- **并发标记**阶段就是进行 GC Roots Tracing 的过程
- **重新标记**阶段则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一般会比初始标记阶段稍长一些，但远比并发标记的时间更短

3、整个过程中**耗时最长**的**并发标记**和**并发清除**过程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一起并发执行的

4、CMS 是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿

5、CMS 的缺点

- CMS 收集器对 CPU 资源非常敏感
  - 面向并发设计的程序都对 CPU 资源比较敏感
  - 在并发阶段，虽然不会导致用户线程停顿，但是因为占用了一部分线程(或者说 CPU 资源)而导致应用程序变慢，总吞吐量会降低
  - CMS 默认启动的回收线程数是(CPU 数量+3)/4，也就是当 CPU 在 4 个以上时，并发回收垃圾收集线程不少于 25%的 CPU 资源，并且随着 CPU 数量的增加而下降，当 CPU 不足 4 个时，CMS 对用户程序的影响可能就很大
  - 虚拟机提供了一种称为"增量式并发收集器"(Incremental Concurrent Mark Sweep/i-CMS)的 CMS 收集器变种，在并发标记、并发清理的时候让 GC 线程用户线程交替运行，尽量减少 GC 线程的独占资源时间，这样整个垃圾收集过程会更长，但对用户程序的影响就会显得少一些，即速度下降没有那么明显，**实践证明，增量式的 CMS 收集器效果一般，现在不再提倡使用**
- CMS 收集器无法处理浮动垃圾(Floating Garbage)，可能出现"Concurrent Mode Failure"失败而导致另一次 Full GC 的产生
  - 由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，**这一部分垃圾出现在标记过程之后，CMS 无法在当次收集中处理掉他们，只要留待下次以 GC 时再清理掉，这一部分垃圾就被称为浮动垃圾**
  - 由于在垃圾收集阶段用户线程还需要运行，也就是需要预留足够的内存空间给用户线程使用，因此 CMS 收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集

时的程序运作使用

- 在 JDK 1.5 的默认设置下，CMS 收集器当老年代使用了 68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中，老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction 的值来提高触发百分比，以降低内存回收次数从而获取更好的性能
- 在 JDK 1.6 中，CMS 收集器的启动阈值提高到了 92%，要是 CMS 运行期间预留的内存无法满足程序需要，就会出现一次"Concurrent Mode Failure"失败，这时虚拟机将启动后备方案：临时启用 Serial Old 收集器来重新进行老年代的垃圾收集(因此参数不宜设置过高，会导致大量"Concurrent Mode Failure"失败，性能反而降低)

➢ CMS 是一款基于"标记-清除"算法实现的收集器

- 这意味着收集结束会有大量的空间碎片产生，空间碎片过多，将会给大对象分配带来很大的麻烦
- 往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次 Full GC
- 为了解决这个问题，CMS 收集器提供了一个-XX:+UseCMSCompactAtFullCollection 开关参数(默认开启)，用于在 CMS 收集器顶不住要进行 FullGC 时开启内存碎片的合并整理过程，**内存整理是无法并发的**，空间碎片问题没有了，但停顿时间不得不边长
- 此外虚拟机还提供另一个参数-XX:CMSFullGCsBeforeCompaction，这个参数用于设置执行多少次不压缩的 Full GC 后，跟着来一次带压缩的(默认值 0，表示每次进入 Full GC 时都进行碎片整理)

6、GC 方式

➢ **新生代 GC(Minor GC)**：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具，备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。

➢ **老年代 GC(Major GC/Full GC)**：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC(但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程)。MajorGC 的速度一般会比 Minor GC 慢 10 倍以上。

3.5.7. G1 收集器

1、G1(Gargabe-First)收集器是当今收集器计数发展的最前沿成果之一

2、G1 是一款面向服务端应用的垃圾收集器，HotSpot 开发团队赋予它的使命是(在比较长期的)未来可以替换掉 JDK 1.5 中发布的 CMS 收集器，与其他 GC 收集器相比，G1 具备如下特点：

➢ **并行与并发**：**G1 能充分利用多 CPU**，多核环境下的硬件优势，**使用多个 CPU(或 CPU 核心)来缩短 Stop-The-World 停顿的时间(并行)**，部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过**并发**的方式让 Java 程序继续执行

➢ **分代收集**：与其他收集器一样，分代的概念在 G1 中仍然保留，虽然 **G1 可以不需要其他收集器配合就能独立管理整个 GC 堆**，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间的、熬过多次 GC 的旧对象以获取更好的收集效果

- **空间整合**：与 CMS 的"标记-清理"算法不同，G1 从整体来看是基于"标记-清理"算法实现的收集器，从局部(两个 Region 之间)上来看是基于"复制"算法实现的，**这两种算法都意味着 G1 运作期间不会产生内存空间碎片，收集后能提供规整的可用内存**
- **可预测的停顿**：这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS 共同的关注点，**但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 M 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒**

3、在 G1 之前的其他收集器进行收集的范围都是整个新生代或者老年代，而 G1 不再是这样，使用 G1 收集器时，Java 堆的内存布局就与其他收集器有很大差别，它将整个 Java 堆规划为多个大小相等的独立区域(Region)，虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region(不需要连续)的集合

4、G1 收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个 Java 堆中进行全区域的垃圾收集

- **G1 跟踪各个 Region 里面的垃圾堆积的价值大小，在后台维护一个优先列表，每次根据允许的收集时间，有限回收价值最大的 Region(这也是 Garbage-First 名称的由来)**
- 这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限时间内可以获取尽可能高的收集效率

5、G1 把内存"化整为零"的思路，理解起来似乎很容易，但是实现的细节却远远没有那么简单，从第一篇 G1 论文发表到 G1 商用，用了整整 10 年

- 把 Java 堆分为多个 Region 后，垃圾收集是否就真的能以 Region 为单位进行了?
- Region 不可能是孤立的，一个对象分配在某个 Region 中，它并非只能被本 Region 中的其他对象引用，而是可以与整个 Java 堆任意的对象发生引用关系
- 这个问题并非 G1 中才有，只是 G1 中更加突出，在以前的分代收集中，新生代的规模一般都比老年代要小许多，新生代的收集也比老年代要频繁许多，回收新生代对象也面临同样的问题，如果回收新生代时也不得不扫描老年代的话，那么 Minor GC 的效率可能下降不少

6、在 G1 收集器中，Region 之间的对象引用以及其他收集器中的新生代与老年代之间的对象引用，虚拟机都是使用 Remembered Set 来避免全堆扫描的

- G1 中每个 Region 都有一个与之对应的 Remembered Set，虚拟机发现程序在对 Reference 类型的数据进行写操作时，会产生一个 Write Barrier 暂时中断操作，检查 Reference 引用的对象是否处于不同的 Region 之中(在分代的例子中就是检查是否老年代中的对象引用了新生代中的对象)
- 如果是，便通过 CardTable 把相关引用信息记录到被引用对象(指红色部分的新生代还是老年代对象???)所属的 Region 的 Remembered Set 之中
- 当进行内存回收时在 GC 根节点的枚举范围中加入 Remembered Set 即可保证不对全堆扫描也不会有遗漏

7、如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤

- **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，并修改 TAMS(Next Top at Mark Start)的值，让下一阶段用户程序并发运行时，能在正确可用的 Region 中创建新对象，这阶段需要停顿线程，但耗时很短

- **并发标记(Concurrent Marking)**：从 GC Root 开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时长，但可与用户程序并发执行
- **最终标记(Final Marking)**：为了修正在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标记记录，虚拟机将这段时间对象变化记录在线程 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中，这阶段需要停顿线程，但可并行(多个 GC 线程)执行
- **筛选回收(Live Data Counting and Evacuation)**：首先对各个 Region 的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率，因此该阶段设置为停顿的

3.5.8. 理解 GC 日志

1、阅读 GC 日志是处理 Java 虚拟机内存问题的基础技能，它只是一些人为确定的规则，没有太多技术含量

2、每一种收集器的日志形式都由它们自身的实现所决定，换而言之，每个收集器的日志格式可以不一样，但虚拟机设计者为了方便用户阅读，将各个收集器的日志都维持一定的共性

3、通用格式解析：

- 最前面的数组：代表了 GC 发生的时间，即从虚拟机启动以来经过的秒数
- GC 日志开头的"[GC"和"[Full GC"说明这次垃圾收集的停顿类型
  - 有 Full 说明这次 GC 发生了 Stop-The-World
  - 如果调用 System.gc()方法所触发的收集，那么将显示"[Full GC(System)"
  -
- 接下来的"[DefNew"、"[Tenured"、"[Perm"表示 GC 发生的区域，这里显示的区域名称与使用的 GC 收集器密切相关
- 后面方括号内部的"3324K->152K(3712K)"含义是"GC 前该内存区域已使用容量->GC 后该内存区域已使用容量(该内存区域总容量)
- 在方括号之外的"3324K->152K(11904K)"表示"GC 前 Java 堆已使用容量->GC 后 Java 堆已使用容量(Java 堆总容量)
- "0.0025925 secs"表示该内存区域 GC 所占用的时间，单位是秒

3.5.9. 垃圾收集器参数总结

1、垃圾收集相关的常用参数

- **UseSerialGC**：虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial+Serial Old 的收集器组合进行内存回收
- **UseParNewGC**：打开此开关后，使用 ParNew+Serial Old 的收集器组合进行内存回收
- **UseConcMarkSweepGC**：打开此开关后，使用 ParNew+CMS+Serial Old 的收集器组合进行内存回收，Serial Old 收集器将作为 CMS 收集器出现在

Concurrent Mode Failure 失败后的后备收集器使用

- **UseParallelGC**：虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge+Serial Old(PS MarkSweep)的收集器组合进行内存回收
- **UseParallelOldGC**：打开此开关后，使用 Parallel Scavenge+Parallel Old 的收集器组合进行内存回收
- **SurvivorRatio**：新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden：Survivor=8：1
- **PretenureSizeThreshold**：直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
- **MaxTenuringThreshold**：晋升到老年代的对象年龄，每个对象在坚持过一次 Minor GC 之后，年龄就增加 1，当超过这个参数值时就进入老年代
- **UseAdaptiveSizePolicy**：动态调整 Java 堆中各个区域的大小以及进入老年代的年龄
- **HandlePromotionFailure**：是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
- **ParallelGCThreads**：设置并行 GC 时进行内存回收的线程总数
- **GCTimeRatio**：GC 时间占总时间的比率，默认值为 99，即允许 1%的 GC 时间，仅在使用 Parallel Scavenge 收集器时生效
- **MaxGCPauseMillis**：设置 GC 的最大停顿时间，仅在使用 Parallel Scavenge 收集器时生效
- **CMSInitiatingOccupancyFraction**：设置 CMS 收集器在老年代空间被使用多少后出发垃圾收集，默认值为 68%，仅在使用 CMS 收集器时生效
- **UseCMSCompactAtFullCollection**：设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理，仅在使用 CMS 收集时生效
- **CMSFullGCsBeforeCompaction**：设置 CMS 收集器在进行若干垃圾收集后再启动一次内存碎片整理，仅在使用 CMS 收集器时生效

3.6. 内存分配与回收策略

1、Java 技术体系中所提倡的自动内存管理最终可以规划为自动地解决了两个问题：给对象分配内存以及回收分配给对象的内存

2、对象的内存分配

- 往大方向讲，就是在堆上分配(但也可能经过 JIT 编译后被拆散为标量类型并间接地栈上分配)
- 对象主要分配在新生代的 Eden 区上，如果启动了本地线程分配缓冲，将按线程优先在 TLAB**(Thread Local Allocation Buffer)**上分配
- 少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置

3、接下来几节：讲解几条最普遍的内存分配规则，并通过代码去验证这些规则

3.6.1. 对象优先在 Eden 分配

1、大多数情况下，对象在新生代 Eden 区中分配，当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 MinorGC

2、虚拟机提供了-XX:+PrintGCDetails 这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前的内存各区域分配情况

- 在实际应用中，内存回收日志一般是打印到文件后通过日志工具进行分析，不过本实验的日志并不多，直接阅读就能看的很清楚

3.6.2. 大对象直接进入老年代

1、所谓大对象是指，需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串以及数组(如 byte[])

2、大对象对虚拟机的内存分配来说就是一个坏消息，**比遇到一个大对象更加坏的消息就是：遇到一群"朝生夕灭"的"短命"大对象**，写程序时应当避免，经常出现大对象容易导致内存还有不少空间就提前触发垃圾收集以获取足够的连续空间来"安置"它们

3、虚拟机提供了一个-XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代分配，这样做的目的是避免在 Eden 区一级两个 Survivor 区之间发生大量的内存赋值

- 若设置为 3M，那么不能写成 3M，而要写成 3145728

3.6.3. 长期存活的对象将进入老年代

1、既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中

- 为做到这一点，虚拟机给每个对象定义了一个对象年龄(Age)计数器
- 如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并能被 Survivor 容纳的话，将被移动到 Survivor 空间中，且对象年龄设为 1
- 对象在 Survivor 区中每"熬过"一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度(默认为 15 岁)，就将会被晋升到老年代中
- 对象晋升老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold 设置

3.6.4. 动态对象年龄判断

1、为了能更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 空间中相同年龄所有对象大小的总和等于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无需等到 MaxTenuringThreshold 中要求的年龄

3.6.5. 空间分配担保

1、在发生 Minor GC 之前，虚拟机会先检查老年代最大可用连续空间是否大于新生代所有对象总空间

- 如果这个条件成立，那么 Minor GC 可以确保是安全的
- 如果这个条件不成立，则虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败
  - 如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小。如果大于，将尝试着进行一次 Minor GC，尽管这次 Minor GC 是有风险的；如果小于，或者 HandlePromotionFailure

设置不允许冒险，那这时也要改为进行一次 Full GC

2、冒险

- 新生代使用复制收集算法，但为了内存利用率，只使用其中一个 Survivor 空间来作为轮换备份，因此当出现大量对象在 Minor GC 后仍然存活的情况(最极端的情况是内存回收后新生代中所有对象都存活)，就需要老年代进行分配担保，把 Survivor 无法容纳的对象直接进入老年代
- 而老年代进行这些担保，**前提是老年代本身还有容纳这些对象的剩余空间**，一共有多少对象会存活下来在实际完成内存回收之前是无法明确知道的，**所以只好在每一次回收晋升到老年代对象容量的平均大小值作为经验值**，与老年代剩余空间进行比较，决定是否进行 Full GC 来让老年代腾出更多空间
- 去平均值进行比较仍然是一种动态概率的手段，也就是说，如果某次 Minor GC 存活后的对象突增，远远高于平均值，依然会导致担保失败(Handle Promotion Failure)。如果出现了 HandlePromotionFailure 失败，那就只好在失败后重新发起一次 Full GC
- 虽然担保失败时绕过的圈子是最大的，但大部分情况下还是会将 HandlePromotionFailure 开关打开，避免 Full GC 过于频繁

Chapter 4. 虚拟机性能监控与故障处理工具

4.1. 概述

1、给一个系统定位问题的时候，知识、经验是关键基础，数据是依据，工具是运用知识处理数据的手段

- 数据包括：运行日志、异常堆栈、GC 日志、线程快照(threaddump/javacore 文件)、堆转储快照(heapdump/hprof 文件)等

4.2. JDK 的命令行工具

1、Java 开发人员肯定都知道 JDK 的 bin 目录中有 java.exe、javac.exe 这两个命令行工具，但并非所有程序员都了解过 JDK 的 bin 目录之中其他命令行程序的作用
2、这些故障处理工具被 Sun 公司作为"礼物"附赠给 JDK 的使用者，并在软件的使用说明中把它们声明为"没有技术支持且是实验性质的"产物，但事实上，这些工具都非常稳定且功能强大，能在处理应用程序性能问题、定位故障时发挥很大作用
3、这些命令行工具大多是 jdk/lib/tools.jar 类库的一层薄包装而已
4、Sun JDK 监控和故障处理工具

- jps：JVM Process Status Tool，显示指定系统内所有的 HotSpot 虚拟机进程
- jstat：JVM Statistics Monitoring Tool，用于收集 HotSpot 虚拟机各方面的运行数据
- jinfo：Configuration Info for Java，显示虚拟机配置信息
- jmap：Memory Map for Java，生成虚拟机的内存转储快照(heapdump 文件)
- jhat：JVM Heap Dump Browser，用于分析 heapdump 文件，它会建立一个 HTTP/THML 服务器，让用户可以在浏览器上查看分析结果
- jstack：Stack Trace for Java，显示虚拟机的线程快照

4.2.1. jps：虚拟机进程状况工具

1、JDK 的很多小工具的名字都参考了 UNIX 命令的命名方式，jps(JVM Process Status Tool)是其中的典型
2、<jps>

- jps [options] [hostid]
- 列出正在运行的虚拟机进程，并显示虚拟机执行主类(Main Class, main()函数所在的类)名称以及这些进程的本地虚拟机唯一 ID(Local Virtual Machine Identifier, LVMID)
- 虽然功能比较单一，但它是使用频率最高的 JDK 命令行工具，因为其他 JDK 工具大多需要输入它查询到的 LVMID 来确定要监控的是哪一个虚拟机进程
- 对本地虚拟机来说，LVMID 与操作系统的进程 ID(Process Identifier，PID)是一致的，使用 Windows 的任务管理器或者 UNIX 的 ps 命令也可以查询到虚拟机进程的 LVMID，如果同时启动了多个虚拟机进程，无法根据进程名

称定位时，就只能依赖 jps 命令显示主类的功能才能区分了

- -q：只输出 LVMID，省略主类的名称
- -m：输出虚拟机进程启动时传递给主类 main()函数的参数
- -l：输出主类的全名，如果进程执行的是 Jar 包，输出 Jar 路径
- -v：输出虚拟机进程启动时的 JVM 参数

4.2.2. jstat:虚拟机统计信息监视工具

1、jstat(JVM Statistics Monitoring Tool)是用于监视虚拟机各种运行状态信息的命令行工具

2、jstat 可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT 编译等运行数据，在没有 GUI 图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具

3、<jstat>

- jstat [option vmid [interval [s|ms] [count]]]
- 如果是本地虚拟机进程，VMID 与 LVMID 是一致的，如果是远程虚拟机进程，那 VMID 的格式应当是

  [protocol:] [//] lvmid [@hostname[:port]/servername]
- interval 和 count 代表查询间隔和次数，如果省略这两个参数，说明只查询一次
- jstat -gc 2764 250 20：每 250 毫秒查询一次进程 2764 垃圾收集情况，一共查询 20 次
- -class：监视类装载、卸载数量、总空间以及类装载所耗费的时间
- -gc：监视 Java 堆状况，包括 Eden 区、两个 survivor 区、老年代、永久代等的容量、已用空间、GC 时间合计等信息
- -gccapacity：监视内容与-gc 基本相同，但输出主要关注 Java 堆各个区域使用到的最大、最小空间
- -gcutil：监视内容与-gc 基本相同，但输出主要关注已使用空间占总空间的百分比
- -gccause：与-gcutil 功能一样，但是会额外输出导致上一次 GC 产生的原因
- -gcnew：监视新生代 GC 状况
- -gcnewcapacity：监视内容与-gcnew 基本相同，输出主要关注使用到的最大、最小空间
- -gcold：监视老年代 GC 状况
- -gcoldcapacity：监视内容与-gcold 基本相同，输出主要关注使用到的最大、最小空间
- -gcpermcapacity：输出永久代使用到的最大、最小空间
- -compiler：输出 JIT 编译器编译过的方法、耗时等信息
- -printcompilation：输出已经被 JIT 编译的方法

4、参数解释

- jstat -gcutil 1874
- E：新生代区 Eden
- S0\S1：Survivor0、Survivor1 这两个 Survivor 区
- O：老年代 Old
- P：永久代 Permanet

- YGC：Young GC，即 Minor GC 次数
- YGCT：Yount GC Time，即 Minor GC 耗时
- FGC：Full GC
- FTCG：Full GC Time，即 Full GC 耗时
- GCT：Minor GC 与 Full GC 总耗时

4.2.3. jinfo:Java 配置信息工具

1、jinfo(Configuration Info for Java)的作用是实时地查看和调整虚拟机各项参数
2、使用 jps 命令的-v 参数可以查看虚拟机启动时显示指定的参数列表，但如果向知道未被显式指定的参数的系统默认值，除了去查找资料外，就只能用 jinfo 的-flag 选项进行查询
3、如果 JDK1.6 或者以上版本，可以使用**-XX:+PrintFlagsFinal** 查看参数默认值
4、jinfo 还可以使用-sysprops 选项把虚拟机进程的 System.getProperties()的内容打印出来
5、jinfo [option] pid

- -flag：显式默认值
  - jinfo -flags 1874　　<==显式所有项的默认值
  - jinfo -flag CICompilerCount 1874　　<==显示指定项的默认值
- -sysprops：把虚拟机进程的 System.getProperties()的内容打印出来

4.2.4. jmap:Java 内存映像工具

1、jmap(Memory Map for Java)命令用于生成堆转储快照(一般称为 heapdump 或 dump 文件)
2、如果不适用 jmap 命令，要想获取 Java 堆转储快照，还有一些比较暴力的手段：譬如在第二章中用过的"**-XX:+HeapDumpOnOutOfMemoryError**"参数，可以让虚拟机在 OOM 异常出现之后自动生成 dump 文件，通过"**-XX:+HeapDumpOnCtrlBreak**"参数则可以使用[ctrl]+[break]键让虚拟机生成 dump 快照，又或者在 Linux 系统下通过 kill -3 命令发送进程退出信号"吓唬"一下虚拟机，也能拿到 dump 文件
3、jmap 的作用并不仅仅为了获取 dump 文件，它还可以查询 finalize 执行队列、Java 堆和永久代的详细信息，如空间使用率、当前用的是哪种收集器等
4、<jmap>

- jmap [option] vmid
- -dump：生成 Java 堆转储快照，格式为-dump:[live,]format=b, file=<filename>，其中 live 子参数说明是否只 dump 出存活对象
- -finalizerinfo：显示在 F-Queue 中等待 Finalizer 线程执行 finalize 方法的对象
- -heap：显示 Java 堆详细信息，如使用哪种回收器，参数配置，分代状况等
- -histo：显示堆中对象统计信息，包括类、实例数量、合计容量
- -permstat：以 ClassLoader 为统计口径显示永久代内存状态
- -F：当虚拟机进程对-dump 选项没有响应时，可使用这个选项强制生成 dump 快照

4.2.5. jhat：虚拟机堆转储快照分析工具

1、Sun JDK 提供 jhat(JVM Heap Analysis Tool)命令与 jmap 搭配使用，来分析 jmap 生成的堆转储快照。

2、jhat 内置了一个微型的 HTTP/HTML 服务器，生成 dump 文件的分析结果后，可以在浏览器中查看

3、不过在实际工作中，除非真的没有别的工具可用，否则一般不会直接使用 jhat 命令来分析 dump 文件，原因如下

- 一般不会再部署应用程序的服务器上直接分析 dump 文件，即使可以这样做，也会尽量将 dump 文件复制到其他机器上进行分析，因为分析工作是一个耗时而且消耗硬件资源的过程，既然都要在其他机器上进行，就没有必要受到命令工具的限制了
- jhat 的分析功能相对来说比较简陋，VisualVM，以及专业用于分析 dump 文件的 Eclipse Memory Analyzer、IBM HeapAnalyzer 等工具，都能实现比 jhat 更强大更专业的分析功能

4、配合 jmap 的例子

- jmap -dump:format=b,file=eclipse.bin 1874
- jhat eclipse.bin
- 在接下来的输出中会指定端口 7000
- 在浏览器中键入 http://localhost:7000/就可以看到分析结果

4.2.6. jstack：Java 堆栈跟踪工具

1、jstack(Stack Trace for Java)命令用于生成虚拟机当前时刻的线程快照(一般称为 trheaddump 或者 javacore 文件)

2、线程快照就是当前虚拟机每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程死锁、死循环、请求外部资源导致的长时间等待都是导致线程长时间停顿的常见原因

3、线程出现停顿的时候通过 jstack 来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做了什么，或者等待什么资源

4、jstack [option] vmid

- -F：当正常输出的请求不被响应时，强制输出线程堆栈
- -l：除堆栈外，显示关于锁的附加信息
- -m：如果调用本地方法的话，可以显示 C/C++的堆栈

5、在 JDK1.5 中，java.lang.Thread 类新增一个 getAllStackTraces()方法用于获取虚拟机中所有线程的 StackTraceElement 对象，使用这个对象可以通过简单的几行代码就能完成 jstack 的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈

6、经验代码 P111

4.2.7. HSDIS：JIT 生成代码反汇编

1、在 Java 虚拟机规范中，详细描述了虚拟机指令集中每条指令的执行过程、执行前后对操作数栈、局部变量表的影响细节

2、这些细节描述与 Sun 的早期虚拟机(Sun Classic VM)高度吻合，但随着技术发展，高性能虚拟机真正的细节实现方式已经渐渐与虚拟机规范所描述的内容产生了越来越大的差距，虚拟机规范中的描述逐渐成了虚拟机实现的"概念模型"--即

实现只能保证与规范描述等效
3、基于这个原因，我们分析程序的**执行语义问题(虚拟机做了什么)时**，在字节码层面上分析完全可行，但分析程序的**执行行为问题(虚拟机是怎样做的，性能如何)时**，在字节码上分析就没有意义了，需要通过其他方式
4、分析程序如何执行，通过软件调试工具(GDB、Windbg 等)来断点调试是最常见的手段，**但是这样的调试方式在 Java 虚拟机中会遇到很大困难，因为大量执行代码是通过 JIT 编译器动态生成到 CodeBuffer 中，没有很简单的手段来处理这种混合模式的调试**。因此不得不通过一些特别的手段来解决问题，基于这种背景，HSDIS 插件就登场了
5、HSDIS 是一个 Sun 官方推荐的 HotSpot 虚拟机 JIT 编译代码的反汇编插件，它包含在 HotSpot 虚拟机的源码之中，但没有提供编译后的程序

- "-XX:+PrintAssembly"把动态生成的本地代码还原为汇编代码输出，同时还生成了大量非常有价值的注释
- <未完成>：上述参数好像不能用，P113

4.3. JDK 的可视化工具

1、JDK 中除了提供大量的命令行工具外，还有两个功能强大的可视化工具：JConsole 和 VisualVM，这两个工具是 JDK 的正式成员
2、JConsole 是在 JDK 1.5 时期就已经提供的虚拟机监控工具，而 VisualVM 在 JDK 1.6 Update 7 中才首次发布，现在已经成为 Sun(Oracle)主力推动的多合一故障处理工具

4.3.1. JConsole：Java 监视与管理控制台

1、JConsole(Java Monitoring and Management Console)是一种基于 JMX 的可视化监视、管理工具。它管理部分的功能是针对 JMX MBean 进行管理，由于 MBean 可以使用代码、中间服务器的管理控制台或者所有符合 JMX 规范的软件进行访问，所以本节会着重介绍 JConsole 监视部分的功能
2、启动 JConsole
3、内存监控

- "内存"页签相当于可视化的 jstat 命令，用于监视收集器管理的虚拟机内存(Java 堆和永久代)变化趋势
- PS Old Gen：老年代
- PS Eden Space：新生代
- PS Survivor Space：新生代
- Metaspace：永久代
- Code Cache：方法区
- Compressed Class Space

4、线程监控

- "线程"页签相当于可视化的 jstack 命令，遇到线程停顿时可以使用这个页签进行监控分析
- 线程长时间停顿的原因：等待外部资源(数据库连接、网络资源、设备资源)、死循环、锁等待(活锁和死锁)

5、**注意只有程序还在跑的时候才能进行监控，当一个类的 main 函数运行结束，该进程就结束了**

4.3.2. VisualVM：多合一故障处理工具

1、VisualVM(All-in-One Java Troubleshooting Tool)是到目前为止随 JDK 发布的功能最强大的运行监视和故障处理程序，并且可以预见在未来一段时间内都是官方主力发展的虚拟机故障处理工具

2、"All-in-One"预示着它除了运行监视、故障处理外，还提供了很多其他方面的功能，如性能分析(Profiling)，VisualVM 的性能分析功能甚至比起 JProfiler、YourKit 等专业且收费的 Profiling 工具都不会逊色多少

3、**而且 VisualVM 的还有一个很大的优点：不需要被监视的程序基于特殊 Agent 运行，因此它对应用程序的实际性能影响很小，使得它能直接应用在生产环境中，这个优点是 JProfiler、YourKit 等工具无法媲美的**

4、VisualVM 兼容范围与插件安装

- VisualVM 是基于 NetBeans 平台开发的，因此它一开始就具备了插件扩展功能的特性，通过插件扩展支持，VisualVM 可以做到
  - 显示虚拟机进程以及进程的配置、环境信息(jps、jinfo)
  - 监视应用程序的 CPU、GC、堆、方法区以及线程的信息(jstat、jstack)
  - dump 以及分析堆转储快照(jmap、jhat)
  - 方法级的程序运行性能分析，找出被调用最多、运行时间最长的方法
  - 离线程序快照：收集程序的运行时间配置、线程 dump、内存 dump 等信息建立一个快照，可以将快照发送开发者处进行 Bug 反馈
  - 其他 plugins 的无限的可能性

5、插件可以进行手动安装，在 http://Visualvm java.net/pluginscenters.html 下载 *.nbm 包后，点击"工具"-"插件"-"已下载"菜单，然后在弹出的对话框中指定 nbm 包路径便可进行安装

6、生成、浏览堆转储快照

- 在"应用程序"窗口中右键单击应用程序节点，选择"堆 Dump"
- 在"应用程序"窗口双击应用程序节点打开应用程序标签，然后在"监视"标签中单击"堆 Dump"
- 生成了 dump 文件后，"应用程序"页签将在该堆的应用程序下增加一个以[heapdump]开头的子节点，并且在主页千重打开了该转储快照
- 如果要把 dump 文件保存或发送出去，要在 heapdump 节点上右键选择"另存为"菜单，否则当 VisualVM 关闭时，生成的 dump 文件会被当做临时文件删除掉
- 要打开一个已经存在的 dump 文件，通过文件菜单中的"装入"功能，选择硬盘上的 dump 文件即可
- 堆页签
  - 摘要：可以看到应用程序 dump 时的运行时参数
  - 类：以类为统计口径统计类的实例数量、容量信息
  - 实例："实例"面板不能直接使用，因为不能确定用户想查看哪个类的实例，所以需要通过"类"面板进入
  - OQL 控制台：运行 OQL 查询语句，同 jhat 中介绍的 OQL 功能一样

7、分析程序性能

- "Profiler"页签，VisualVM 提供了程序运行期间方法级的 CPU 执行时间分析以及内存分析，做 Profiling 分析肯定会对程序运行性能有比较大的影响，所以一般不在生产环境中使用这项功能
- VisualVM 会记录这段时间中应用程序执行过的方法
  - 若是 CPU 分析，会统计每个方法执行次数、执行耗时
  - 若是内存分析，则会统计每个方法关联的对象以及这些对象所占的空间
  - 分析结束后点击停止按钮结束监控过程
- **可以根据实际业务的复杂程度与方法的时间、调用次数做比较，找到最有优化价值的方法**

8、BTrace 动态日志跟踪

- BTrace 是一个很有趣的 VisualVM 插件，本身也是可以独立运行的程序
- **它的作用是在不停止调试目标程序运行的前提下，通过 HotSpot 虚拟机的 HotSwap 技术动态加入原本不存在的调试代码**
- 这项功能对实际生产中的程序很有意义：经常遇到程序出现问题，但是排查错误的一些必要信息，譬如方法参数，返回值等，在开发时并没有打印到日志中，以至于不得不停掉服务，通过调试增量来加入日志代码解决问题，当遇到生产环境服务无法随便停止时，缺一两句日志导致排错进行不下去是很郁闷的事情
- "在应用程序"面板右键点击要调试的程序，会出现"Trace Application..."菜单，点击将进入 BTrace 面板，这个面板看起来就像一个简单的 Java 程序开发环境，里面还有一小段 Java 代码
- <未完成>P129 例子无法通过

9、内存页签的参数解释

- heap 区
  - Eden Space：伊甸园(新生代)
  - Survivor Space：幸存者区(新生代)
  - Tenured Gen：养老区(老年代)
- 非 heap 区
  - Metaspace：就是以前的永久代(Perm Gen)
  - Code Cache：代码缓存区
  - JVM Stack：Java 虚拟机展
  - Local Method Stack：本地方法栈

Chapter 5. 调优案例分析与实战

5.1. 概述

1、在处理实际项目的问题时，除了知识与工具外，经验同样是一个很重要的因素

5.2. 案例分析

5.2.1. 高性能硬件上的程序部署策略

1、场景
- 15 万 PV/天左右的在线文档系统更换硬件
- 新硬件：4 个 CPU，16GB 内存
- 通过-Xmx 和-Xms 将堆固定在 12GB

2、在高性能硬件上部署程序，目前主要有两种方式
- 通过 64 位 JDK 来使用大内存
- 使用若干个 32 位虚拟机建立逻辑集群来利用硬件资源

3、可以给 Java 虚拟机分配超大堆的前提：
- 是有把握把应用程序的 Full GC 频率控制的足够低，至少要低到不会影响用户使用，譬如几十小时乃至一天才出现一个 Full GC
- 控制 Full GC 频率的关键：看应用中绝大多数对象能否符合"朝生夕灭"的规则，即大多数对象的生存时间不应太长，尤其是不能有成批量的、长生存时间的大对象产生，**这样才能保证老年代空间的稳定**

4、如果使用 64 位 JDK 来管理大内存，还应考虑以下问题：
- 回收内存导致长时间停顿
- 现阶段(具体时间???)，64 位 JDK 性能测试结果普遍低于 32 位 JDK
- 需要保证程序足够稳定，因为这种应用要是产生堆溢出几乎就无法产生堆转储快照(因为要产生几十 GB 乃至更大的 Dump 文件)，哪怕产生了也无法进行分析
- 相同程序在 64 位 JDK 小号的内存一般比 32 位 JDK 大，这是由于指针膨胀，以及数据类型对其补白等因素导致的

5、如果采用逻辑集群的方式来部署程序，还应考虑以下问题
- 尽量避免节点竞争全局资源，最典型的就是磁盘竞争，各个节点如果同时访问某个磁盘文件的话(尤其是并发写操作容易出现问题)，很容易导致 IO 异常
- 很难最高效率的利用某些资源池
- 各个节点仍然不可避免地受到 32 未内存限制

- 32 位 Windows 平台中每个进程只能使用 2GB 内存
- 在 Linux 或 UNIX 系统中，可以提升到 3G 乃至接近 4G 的内存，但是 32 位中仍然受最高 4GB(2^{32})内存的限制
  - 大量使用本地缓存的应用，在逻辑集群中会造成较大的内存浪费，因为每个逻辑节点上都有一份缓存，这时候可以考虑把本地缓存改为集中缓存

6、最终解决方案

- 5 个 32 位逻辑集群，每个进程 2GB 内存
- 另外建立一个 Apache 服务作为前段均衡代理访问门户
- 考虑到用户对相应速度比较关心，并且文档服务的主要压力集中在磁盘和内存访问，CPU 资源敏感度较低，因此改为 CMS 收集器进行垃圾回收

5.2.2. 集群间同步导致内存溢出

1、<未完成>

5.2.3. 堆外内存导致的溢出错误

1、<未完成>

2、垃圾收集进行时，虚拟机虽然会对 Direct Memory 进行回收，但是 Direct Memory 却不能像新生代、老年代那样，发现空间不足了就通知收集器进行垃圾回收，它只能等待老年代满了后 Full GC，然后"顺便地"帮它清理掉内存的废弃对象，否则它只能等到抛出内存异常时，先 catch 掉，然后在 catch 块里调用 System.gc()，要是虚拟机还是不 GC，那么就只能抛出内存溢出异常了

5.2.4. 外部命令导致系统变慢

1、问题分析

- 每个用户请求的处理都需要执行一个外部 shell 脚本来获得系统的一些信息
- 执行这个 shell 脚本是通过 Java 的 Runtime.getRuntime().exec()方法来调用的
- 这种调用方式可以达到目的，但是它在 Java 的虚拟机中是非常消耗资源的操作，即使外部命令本身能很快执行完毕，频繁调用时创建进程的开销也非常可观
- Java 虚拟机执行这个命令的过程：
  - 首先克隆一个和当前虚拟机拥有一样环境变量的进程
  - 再用这个新的进程去执行外部命令
  - 最后再退出这个进程
  - 如果频繁执行这个操作，系统的开销会很大，不仅是 CPU，内存负担也很重

2、解决方法：去掉 Shell 脚本执行的语句，改为用 Java 的 API 去获得这些信息后，系统恢复了正常

5.2.5. 服务器 JVM 进程崩溃

1、<未完成>：暂时看不懂

5.2.6. 不恰当数据结构导致内存占用过大

1、问题分析：

- 800MB 的 Eden 空间很快被填满从而引发 Minor GC，但是 Minor GC 后，新生代中绝大部分对象依然是存活的
- ParNew 收集器使用的是复制算法，这个算法的高效是建立在大部分对象是"朝生夕灭"的特性上，如果存活对象过多，把这些对象复制到 Survivor 并维持这些对象引用的正确就称为一个沉重的负担，因此导致 GC 暂停时

间明显变长

2、解决方法

- 如果不修改程序，仅从 GC 调优的角度去解决这个问题，可以考虑将 Survivor 空间去掉(加入参数-XX:SurvivorRatio=65536、-XX:MaxTenuringThreshold=0、或者-XX:+AlwaysTenure)让新生代中存活的对象在第一次 Minor GC 后立即进入老年代，等到 Major GC 的时候再清理掉它们
- 这种措施可以治标，但是副作用也很大，治本的方案需要修改程序
- 空间效率分析：
  - HashMap<Long,Long>结构中，只有 Key 和 Value 所存放的两个长整型数据是有效数据，共 16B
  - 这两个长整型数据包装成 java.lang.Long 对象后，就分别具有 8B 的 MarkWord，8B 的 Klass 指针，在加 8B 存储数据的 long 值
  - 在这两个 Long 对象组成 Map.Entry 后，又多了 16B 的对象头，然后一个 8B 的 next 字段和 4B 的 int 型的 hash 字段，为了对齐还必须添加 4B 的空白填充
  - 最后还有 HashMap 中对 Entry 的 8B 的引用
  - 实际耗费内存为(Long(24B)*2)+Entry(32B)+HashMap Ref(8B)=88B，空间效率为 16B/88B=18%

5.2.7. 由 Windows 虚拟内存导致长时间停顿

1、<未完成>

5.3. 实战：Eclipse 运行速度调优

1、很多 Java 开发人员都有这样的观念：系统调优工作都是针对服务端引用而言，规模越大的系统，就越需要专业的调优运维团队参与，这个观念不全对，其他应用也是需要调优的，在很多开发的时候也需要用到这些调优的知识

5.3.1. 调优前的程序运行状态

1、Eclipse 的配置文件 eclipse.ini 在/Applications/Eclipse.app/Contents/Eclipse 中

5.3.2. 升级 JDK 1.6 的性能变化及兼容问题

5.3.3. 编译时间和类加载时间的优化

1、编译时间是指虚拟机的 JIT 编译器(Just In Time Compiler)编译热点代码(Hot Spot Code)的耗时

- Java 语言为了实现跨平台的特性，Java 代码编译出来后形成的 Class 文件中存储的是字节码(ByteCode)，虚拟机通过解释方式执行字节码命令，比起 C++/C 编译成本地二进制代码来说，速度要慢不少
- 为了解决速度的问题，JDK 1.2 以后，虚拟机内置了两个运行时编译器，如果一段 Java 方法被调用次数达到一定程度，就会判定为热代码交给 JIT 编译器即时编译为本地代码，提高运行速度(这就是 HotSpot 虚拟机名字的由来)

- 甚至有可能在运行期动态编译比C/C++的编译期静态编译出来的代码更优秀，因为运行期可以收集很多编译器无法知道的信息，甚至可以采用一些很激进的优化手段，在优化条件不成立的时候再逆优化退回来
- 因此 Java 程序只要代码没问题(主要是泄露问题)，随着代码被编译得越来越彻底，运行速度应当是越运行越快的
- Java 运行期编译最大的缺点是它进行编译需要消耗程序正常的运行时间，也就是"编译时间"

5.3.4. 调整内存设置控制垃圾收集频率

5.3.5. 选择收集器降低延迟

Chapter 6. 类文件结构

1、代码编译的结果从本地机器码转为字节码，是存储格式发展的一小步，但却是编程语言发展的一大步

6.1. 概述

1、计算机只认识 0 和 1，但由于最近 10 年虚拟机以及大量建立在虚拟机之上的程序语言蓬勃发展，将编写的程序编译成二进制本地机器码(Native Code)已不再是唯一的选择
2、越来越多的程序语言选择了与操作系统和机器指令集无关的、平台中立的格式作为程序编译后的存储格式

6.2. 无关性的基石

1、如果计算机的 CPU 指令集只有 x86 一种，操作系统也只有 Windows 一种，那也许 Java 语言就不会出现
2、Java 的宣传口号：一次编写，到处运行(Write Once, Run Anywhere)
3、与平台无关的理想最终实现在操作系统的应用层上：Sun 公司以及其他虚拟机提供商发布了许多可以运行在不同平台上的虚拟机，这些虚拟机都可以载入和执行一种平台无关的字节码，从而实现了程序的"一次编写，到处执行"
4、各种不同平台的虚拟机与所有平台都统一使用的程序存储格式---字节码(ByteCode)是构成平台无关的基石
5、虚拟机的另外一种中立特性：语言无关性

- 目前，商业机构和开源机构已经在 Java 语言之外发展出一大批在 Java 虚拟机之上运行的语言，如 Clojure、Groovy、JRuby、Jython、Scala 等
- 实现语言无关性的基础仍然是虚拟机和字节码的存储格式
- Java 虚拟机不和包括 Java 在内的任何语言绑定，它只与"Class 文件"这种特定的二进制格式所关联，Class 文件中包含了 Java 虚拟机指令集和符号表示若干其他辅助信息
- 基于安全方面的考虑，Java 虚拟机规范要求在 Class 文件中使用许多强制性的语法和结构化约束，但任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 Class 文件
- 作为一个通用的、机器无关的执行平台，任何其他语言的实现都可以将 Java 虚拟机作为语言的产品交付媒介
  - Java 编译器可以把 Java 代码编译为存储字节码的 Class 文件
  - JRuby 等其他语言编译器一样可以把程序代码编译成 Class 文件
  - 虚拟机并不关心 Class 的来源是何种语言
- Java 语言中的各种变量、关键字和运算符号的语义最终都是由多条字节码命令组和而成，因此字节码命令所能提供的语义描述能力肯定会比 Java 语言本身强大，因此有一些 Java 语言本身无法有效支持的语言特性并不代表字节码本身无法有效支持

6.3. Class 类文件的结构

1、数据结构方面是了解虚拟机的重要基础之一
2、任何一个 Class 文件都对应着唯一一个类或接口等的定义信息，但反之，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通过类加载器直接生成)
3、基本格式

- Class 文件是一组 8 位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排列在 Class 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
- 当遇到需要占用 8 位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 位字节进行存储

4、根据 Java 虚拟机的规定

- 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类数据类型：**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 来分别代表一个字节、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，所有表都习惯性地以"_info"结尾
- 整个 Class 文件本质上就是一张表
- 无论是无符号数还是表，当需要描述**同一类型**但**数量不定**的多个数据时，经常会使用一个前置的容量计数器加若干个连续的数据项的形式，这时称这一系列连续的某一类型数据为某一类型的集合

表格 6-1 Class 文件格式

类型	名称	数量
u4	magic(魔数)	1
u2	minor_version(次版本号)	1
u2	major_version(主版本号)	1
u2	constant_pool_count(chang 常量池容量计数值)	1
cp_info	constant_pool(常量池)	constant_pool_count-1
u2	access_flags(访问标志)	1
u2	this_class(类索引)	1
u2	super_class(父类索引)	1
u2	interfaces_count(接口计数值)	1
u2	interfaces(接口索引集合)	interfaces_count
u2	fields_count(字段计数值)	1
field_info	fields(字段)	fields_count
u2	methods_count(方法计数值)	1
method_info	methods(方法)	methods_count
u2	attributes_count(属性计数值)	1
attribute_info	attributes(属性)	attributes_count

6.3.1. 魔数与 Class 文件的版本

1、每个 Class 文件的头 4 个字节称为魔数(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

- 很多文件存储标准中都使用魔数来进行身份识别，譬如图片格式，如 gif 或 jpeg 等在文件中都存有魔数
- 使用魔数而不是扩展名来进行识别主要是基于安全方面的考虑，因为文件扩展名可以随意地改动
- Class 文件的魔数：0xCAFEBABE(咖啡宝贝?)

2、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(Major Version)
- Java 的版本号从 45 开始的(JDK 1.1 之后的每个 JDK 大版本发布，主版本号+1)
  - 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，即使文件格式没有任何变化，虚拟机也必须拒绝执行超过其版本号的 Class 文件
  - 例如 JDK 1.1 能支持版本号为 45.0~45.65535 的 Class 文件，但无法执行 46.0 以上的 Class 文件，JDK 1.2 则能支持 45.0~46.65535 的 Class 文件

6.3.2. 常量池

1、紧接着主次版本号之后的是常量池入口

- 常量池可以理解为 Class 文件之中的资源仓库
- 它是 Class 文件结构中与其他项目关联最多的数据类型
- 也是 Class 文件空间最大的数据项目之一
- 同时它还是在 Class 文件中第一个出现的表类型数据项目

2、由于产量池中常量的数量是不固定的，所以在常量池的入口需要放置一项 u2 类型的数据，代表常量池容量计数器(constant_pool_count)

- 与 Java 中语言习惯不一样的是，这个容量计数从 1 而不是从 0 开始，这样做是由特殊考虑的，这样做的目的是在于满足后面某些指向常量池的索引值的数据在特定情况下需要表达"不引用任何一个常量池"的含义，这种情况就可以把索引值置为 0 来表示
- Class 文件结构中，只有常量池的容量计数从 1 开始，对其他集合类型，包括接口索引集合，字段表集合，方法表集合等的容量计数都与一般习惯相同，从 0 开始
- 常量的数量=常量计数值-1(例如 0x0016=22，代表常量池中有 21 项常量，索引范围 1~21)

3、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、声明为 final 的常量值等
- <符号引用>则属于编译原理方面的概念，包括以下三类常量
  - 类和接口的全限定名(Fully Qualified Name)
  - 字段的名称和描述符(Descriptor)
  - 方法的名称和描述符

4、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

- ***Class 文件中不会保存各个方法、字段的最终内存布局信息，因为这些字段、方法的符号引用不经过运行期转换的话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***
- 当虚拟机运行时，需要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

5、常量池中每一项都是一个表

- 在 JDK 1.7 之前共有 11 中结构各不相同的表结构数据
- 在 JDK 1.7 中为了更好地支持动态语言调用，又额外增加了 3 种(CONSTANT_MethodHandle_info、CONSTANT_MethodType_info 和 CONSTANT_InvokeDynamic_info)
- 这 14 个表有一个共同特点，就是表开始的第一位是一个 u1 类型的标志位，代表当前这个常量属于哪种常量类型
- 这 14 种常量类型各自均有自己的结构

表格 6-2 常量池的项目类型

类型	标志	描述
CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	标志方法类型
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

表格 6-3 常量池中 14 种常量项的结构总表

常量	项目	类型	描述
CONSTANT_Utf8_info	tag	u1	值为 1
	length	u2	UTF-8 编码的字符串占用的字节数
	bytes	u1	长度为 length 的 UTF-8 编码的字符串
CONSTANT_Integer_info	tag	u1	值为 3
	bytes	u4	按照高位在前存储的 int 值
CONSTANT_Float_info	tag	u1	值为 4
	bytes	u4	按照高位在前存储的 float 值

CONSTANT_Long_info	tag	u1	值为 5
	bytes	u8	按照高位在前存储的 long 值
CONSTANT_Double_info	tag	u1	值为 6
	bytes	u8	按照高位在前存储的 double 值
CONSTANT_Class_info	tag	u1	值为 7
	index	u2	指向全限定名常量的索引(地址偏移)
CONSTANT_String_info	tag	u1	值为 8
	index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	u1	值为 9
	index	u2	指向声明字段的类或者接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向字段描述符 CONSTANT_NameAndType 的索引项
CONSTANT_Methodref_info	tag	u1	值为 10
	index	u2	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称即类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_InterfaceMethodref_info	tag	u1	值为 11
	index	u2	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType
CONSTANT_NameAndType_info	tag	u1	值为 12
	index	u2	指向该字段或方法名称常量项的索引
	index	u2	指向该字段或方法描述符常量的索引
CONSTANT_MethodHandle_info	tag	u1	值为 15
	reference_kind	u1	值必须在 1~9 之间(包括)，它决定了方法句柄的类型，方法句柄类型的值表示方法句柄的字节码行为
	reference_index	u2	值必须是对常量池的有效索引
CONSTANT_MethodType_info	tag	u1	值为 16
	descriptor_index	u2	值必须是对常量池的有效索引，常量池在该索引处的项必须是

			CONSTANT_Utf8_info 结构，表示方法的描述符
CONSTANT_InvokeDynamic_info	tag	u1	值为 18
	bootstrap_method_attrindex	u2	值必须是对当前 Class 文件中引导方法表的 bootstrap_methods[]数组的有效索引
	name_and_type_index	u2	值必须是对当前常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_NameAndType_info 结构，表示方法名和方法描述符

6.3.3. 访问标志

1、在常量池结束之后，紧接着的两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
- 这些含义可以组合使用，例如 0x0021 代表 ACC_PUBLIC 和 ACC_SUPER
- access_flags 共有 16 个标志位可用，当前只定义了 8 个，没有使用到的一律为 0

表格 6-4 访问标志

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 字节码指令的新语意，invokespecial 指令的语意在 JDK 1.0.2 发生过改变，为了区别这条指令使用哪种语意，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	标志这是一个接口
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	标志这个类并非由用户代码产生的
ACC_ANNOTATION	0x2000	标志这是一个注解
ACC_ENUM	0x4000	标志这是一个枚举

6.3.4. 类索引、父类索引与接口索引接口

1、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这个类的继承关系

- 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
  - Java 不允许多重继承，所以父类索引只有一个

- 除了 java.lang.Object 之外，所有类都有父类

➢ **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(如果这个类本身是一个接口，那么是 extends 关键字)语句后的接口顺序从左到右排列在接口索引集合中

4、数据结构

➢ 类索引和父类索引各自指向一个类型为 CONSTANT_Class_info 的类描述符常量，通过 CONSTANT_Class_info 类型的常量中索引值可以找到定义在 CONSTANT_Utf8_info 类型的常量中的全限定名字符串

➢ 对于接口索引集合，入口的第一项---u2 类型的数据为接口计数器(interfaces_count)，表示索引表的容量

➢ 如果该类没有实现任何接口，计数器为 0，后面接的索引表将不再占用任何字节

6.3.5. 字段表集合

1、字段表(field_info)用于描述接口或者类中声明的变量

➢ 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

2、字段包含的信息

➢ 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)

➢ 是否被序列化(transient 修饰符)

➢ 字段数据类型(基本类型、对象、数组)

➢ 字段名称

➢ **概括起来就是分为以下三类**

1) 字段名

2) 字段类型

3) 字段修饰符(public,private,protected,static,final,volatile,transient 的子集)

3、**至于字段叫什么名字，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**

4、字段表的格式，见下表

表格 6-5 字段表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

5、**access_flags 项目**可以设置的标志位和含义见下表

表格 6-6 字段访问标志

标志名称	标志值	含义
ACC_PUBLIC	0x0001	字段是否 public
ACC_PRIVATE	0x0002	字段是否 private

ACC_PROTECTED	0x0004	字段是否 protected
ACC_STATIC	0x0008	字段是否 static
ACC_FINAL	0x0010	字段是否 final
ACC_VOLATILE	0x0040	字段是否 volatile
ACC_TRANSIENT	0x0080	字段是否 transient
ACC_SYNTHETIC	0x1000	字段是否由编译器自动产生
ACC_ENUM	0x4000	字段是否 enum

6、跟随 access_flags 标志的是两项索引值：name_index 和 descriptor_index

- 它们都是对常量池的引用，分别代表字段的简单名称以及字段和方法的描述符

7、概念解析

- <全限定名>：
  - 含有包名的完整类名
  - 并且将'.'换成了'/'
  - 并且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一个";"表示全限定名结束
- <简单名称>：
  - 没有类型和参数修饰的方法或者字段名称
  - 例如 int a;中的 a
  - 例如 void f();中的 f
- <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**
  - 基本数据类型以及代表无返回值的 void 类型都用一个大写字符来表示，而对象则用字符 L 加对象的全限定名来表示

表格 6-7 描述符标志字符含义

标志字符	含义
B	基本类型 byte
C	基本类型 char
D	基本类型 double
F	基本类型 float
I	基本类型 int
J	基本类型 long
S	基本类型 short
Z	基本类型 boolean
V	特殊类型 void
L	对象类型，例如 Ljava/lang/Object

  - **对于数组类型**，每一维度将使用一个前置的"["字符来描述
    - 例如 java.lang.String[][] 类型的二维数组，被记录为:"[[Ljava/lang/String;"
    - 例如整型数组 int[]将被记录为"[I"
  - **描述符用来描述方法时**
    - 按照**先参数列表，后返回值**的顺序描述顺序
    - 参数列表按照参数的严格顺序放在一组小括号内"()"之内

- 例如 void inc()的描述符为"()V"
- 例如 java.lang.String toString()的描述符为()Ljava/lang/String
- 例如 int indexOf(char[]source ,int sourceOffset,int sourceCount,char[] target,int targetOffset,int targetCount,int fromIndex) 的描述符为"([CII[CIII)I"

8、**字段表都包含的固定数据项到 descriptor_index 为止就结束了，不过在 descriptor_index 之后跟随者一个属性表集合用于存储一些额外的信息，字段都可以在属性表中描述零至多项额外信息**

- 例如"final static int m=123"，需要一项名称为 ConstantValue 的属性，其值指向常量 123

9、**字段表中不会列出从超类或者父类接口中继承来的字段，但有可能列出原来 Java 代码中不存在的字段**，譬如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段

10、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，但是对于字节码来说，如果两个字段的描述符不一致，那字段重名就是合法的

6.3.6. 方法表集合

1、Class 文件存储格式中对方法的表述与对字段的描述几乎采用了完全一致的方式，方法表的结构如同字段一样，依次包括

- 访问标志(access_flags)，即方法修饰符
- 名称索引(name_index)
- 描述符索引(descriptor_index)
- 属性表集合(attributes)

表格 6-8 方法表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

表格 6-9 方法访问标志

标志名称	标志值	含义
ACC_PUBLIC	0x0001	方法是否为 public
ACC_PRIVATE	0x0002	方法是否为 private
ACC_PROTECTED	0x0004	方法是否为 protected
ACC_STATIC	0x0008	方法是否为 static
ACC_FINAL	0x0010	方法是否为 final
ACC_SYNCHRONIZED	0x0020	方法是否为 synchronized
ACC_BRIDGE	0x0040	方法是否是由编译器产生的桥接方法
ACC_VARARGS	0x0080	方法是否接受不定参数
ACC_NATIVE	0x0100	方法是否为 native
ACC_ABSTRACT	0x0400	方法是否为 abstract

ACC_STRICTFP	0x0800	方法是否为 strictfp
ACC_SYNTHETIC	0x1000	方法是否是由编译器自动产生的

2、方法的定义可以通过访问标志、名称索引、描述符索引表达清楚，**方法里的 Java 代码，经过编译器编译成字节码指令后，存放在方法属性表集合中一个名为"Code"的属性里面**，属性表作为 Class 文件格式中最具扩展性的一种数据项目

3、与字段表集合相对应，如果父类方法在子类中没有被重写，方法表集合中就不会出现来自父类的方法信息，但是有可能会出现编译器自动添加的方法，最典型的便是类构造器<clinit>方法和实例构造器<init>方法

4、在 Java 中，要重载一个方法，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，特征签名就是一个方法中各个参数在常量池中的字段符号引用集合

- **Java 语言无法依靠返回值的不同来对一个已有方法进行重载**
- **在 Class 文件格式中，特征签名的范围更大，只要描述符不是完全一致的两个方法也可以共存**

6.3.7. 属性表集合

1、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了一些，不再要求各个属性表具有严格的顺序，并且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

2、虚拟机规范预定义的属性，P180

3、**对于每个属性，它的名称需要从常量池中引用一个 CONSTANT_Utf8_info 类型的常量来表示，而属性值的结构是完全自定义的，只需要通过一个 u4 的长度属性去说明属性值所占用的位数即可**

表格 6-10 属性表结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

6.3.7.1. Code 属性

1、Java 程序方法体中的代码经过 javac 编译器处理后，最终变为字节码指令存储在 Code 属性内

2、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

表格 6-11 Code 属性表的结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1

exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count

3、Code 属性表参数解析

1) attribute_name_index：指向 CONSTANT_Utf8_info 型常量的索引，**常量值固定为"Code"**，代表了该属性的属性名称
2) attribute_length：指示了属性值的长度，由于属性名称索引与属性长度一共为 6 个字节，**所以属性值的长度固定为整个属性表长度减去 6 个字节**
3) max_stack：操作数栈(Operand Stacks)深度的最大值，虚拟机在运行的时候需要根据这个值来分配栈帧中操作栈深度
4) max_locals：代表了局部变量(**只能是基本类型，类类型都在堆中分配**)表所需的存储空间，单位是 Slot
   - Slot 是虚拟机为局部变量分配内存所使用的最小单位
   - 对于 byte、char、float、int、short、boolean 和 returnAddress 等长度不超过 32 位的数据类型，每个局部变量占用 1 个 Slot
   - 而 double 和 long 这两种 64 位的数据类型则需要 2 个 Slot 来存放
   - **方法参数(包括 this)、显式异常处理参数、方法体中定义的局部变量都需要使用局部变量表来存放**
   - 局部变量表中的 Slot 可以重用，当代码执行超出一个局部变量的作用域时，这个局部变量所占的 Slot 可以被其他局部变量使用
   - Javac 编译器会根据变量的作用域来分配 Slot 给各个变量使用，然后计算出 max_locals 的大小
5) code_length 和 code：用来存储 Java 源程序编译后生成的字节码指令，
   - code_length 代表字节码长度，code 是用于存储字节码指令的一些列字节流
   - **既然叫字节码指令，那么每个指令就是一个 u1 类型的单字节，当虚拟机读到 code 中的一个字节码时，就可以对应找出这个字节码代表的是什么指令，并且可以知道这条指令后面是否需要跟随参数，以及参数应当如何理解**
   - 一个 u1 数据类型的取值范围为 0x00~0xFF，对应十进制的 0~255，一共 256 条指令，目前 Java 虚拟机规范已经定义了其中约 200 条编码值对应的指令含义
   - 虽然 code_length 是一个 u4 类型的长度，理论上可以到达 $2^{32}-1$，但是虚拟机规范中明确限制了一个方法不允许超过 65535 条字节码指令，它实际只使用了 u2 的长度，如果超过这个限制，Javac 编译器也会拒绝编译，一般来说是很难超过这个限制的

4、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，那么整个 Class 文件中，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

5、**在任何实例方法里面，都可以通过"this"关键字访问到此方法所属的对象**

- 这个访问机制对 Java 程序的编写很重要，而它的实现却非常简单，仅仅通过 Javac 编译器编译的时候把对 this 关键字的访问转变为对一个普通方法参数的访问，然后虚拟机调用实例方法时自动传入此参数而已

- 在实例方法的局部变量表中，至少会存在一个指向当前对象实例的局部变量，局部变量表中也会预留出第一个 Slot 位来存放对象实例的引用

6、在字节码指令之后是这个方法的显式异常处理表集合，异常表对于 Code 属性来说并不是必须存在的

表格 6-12 属性表结构

类型	名称	数量
u2	start_pc	1
u2	end_pc	1
u2	handler_pc	1
u2	catch_type	1

7、异常表实际上是 Java 代码的一部分，编译器使用异常表而不是简单的跳转命令来实现 Java 异常及 finally 处理机制

8、异常表包含 4 个字段，这些字段的含义：

- 如果当字节码在 start_pc 行到 end_pc 行之间(不含第 end_pc 行)，出现了类型为 catch_type 或其子类的异常，转到 handler_pc 行继续处理
- 当 catch_type 的值为 0 时，代表任意异常情况都需要转向到 hander_pc 行进行处理

6.3.7.2. Exceptions 属性

1、这里的 Exceptions 属性是在方法表中与 Code 属性平级的一项属性，与前面讲的异常表是两个概念

2、Exceptions 的作用是列举出方法中可能抛出的受检查异常(Checked Exceptions)，也就是方法描述时在 throws 关键字后面列举的异常

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_exceptions	1
u2	exception_index_table	number_of_exceptions

1) number_of_exceptions 表示可能抛出受检查异常的种类，每个受检查的异常使用一个 exception_index_talbe 项表示
2) exception_intex_table：一个指向常量池 CONSTANT_Class_info 型常量的索引

6.3.7.3. LineNumberTable 属性

1、**用于描述 Java 源码行号与字节码行号(字节码偏移量)之间的对应关系**

2、并不是必须的属性，但默认会生成到 Class 文件之中

3、可以在 Javac 中分别使用-g:none 或-g:lines 选项来取消或要求生成这项信息

4、如果选择不生成 LineNumberTable 属性，对程序运行产生的最主要影响就是当抛出异常时，堆栈中将不会显示出错的行号，并且在调试程序的时候，也无法按照源码来设置断点

表格 6-13 LineNumberTable 属性结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1

u2	line_number_table_length	1
line_number_info	line_number_table	line_number_talbe_length

1) 其中 line_number_info 表包括了 start_pc 和 line_number 两个 u2 类型的数据项，前者是字节码行号，后者是 Java 源码行号

6.3.7.4. LocalVariableTable 属性

1、用于描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的关系
2、不是运行时必须的属性，但默认会生成到 Class 文件之中
3、可以在 Javac 中分别使用-g:none 或-g:vars 选项来取消或者要求生成这项信息
4、取消该信息，最大的影响就是当其他人引用这个方法时，所有的参数名称都将会丢失，IDE 将会使用诸如 arg0，arg1 之类的占位符代替原有的参数名，这对程序运行没有影响，但是会对代码编写带来较大不便

表格 6-14 LocalVariableTable 属性结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

表格 6-15 local_variable_info 项目结构

类型	名称	数量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

5、local_variable_info 参数解释

2) start_pc 和 length：分别代表这个局部变量的声明周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围
3) name_index 和 descriptor_index：指向常量池 CONSTANT_Utf8_info 类型常量的索引，分别代表了局部变量的名称以及这个局部变量的描述符
4) index：这个局部变量在栈帧局部变量表中 Slot 的位置，当这个变量的数据类型是 64 位类型(double 和 long)时，它占用 Slot 为 index 和 index+1

6、**在 JDK 1.5 引入泛型之后，LocalVariableTable 属性增加了一个姐妹属性：LocalVariableTypeTable**

1) 这个新增属性的结构与 LocalVariableTable 非常相似，仅仅是把记录的字段描述符 descriptor_index 替换成了字段的特征签名(Signature)
2) **对于非泛型类型来说，描述符和特征签名能描述的信息基本一致，但是泛型引入后，由于描述符中泛型的参数化类型被擦除掉，描述符就不能准确地描述泛型类型了**

6.3.7.5. SourceFile 属性

1、SourceFile 属性用于记录生成这个 Class 文件的源码文件名称
2、这个属性也是可选的

3、可以分别使用 Javac 的-g:none 或-g:source 选项来关闭或者要求生成这项信息

4、**在 Java 中，对于大多数类来说，类名和文件名是一致的，但是有一些特殊情况，例如内部类，如果不生成这项属性，当抛出异常时，堆栈中将不会显示出错代码所属的文件名**

5、这个属性是一个定长属性

表格 6-16SourceFile 属性结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

1) sourcefile_index 数据项是指向常量池中 CONSTANT_Utf8_info 型常量的索引

6.3.7.6. ConstantValue 属性

1、**该属性的作用是通知虚拟机自动为静态变量赋值，只有被 static 关键字修饰的变量(类变量)才可以使用这项属性**

2、对于静态和非静态变量的赋值方式和时刻

- 非 static 类型的变量(实例变量)的赋值是在实例构造器方法中进行的
- 对于类变量，则有两种方式可选：在类构造器<clinit>方法中或使用 ConstantValue 属性

3、目前 Sun Javac 编译器的选择是：

- 如果同时使用 final 和 static 来修饰一个变量，并且这个变量的数据类型是基本类型或者 java.lang.String，就生成 ConstantValue 属性来进行初始化
- 如果这个变量没有被 final 修饰，或者并非基本类型及字符串，则将会选择在<clinit>方法中进行初始化

表格 6-17ConstantValue 属性结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

1) constantvalue_index 数据项代表了常量池中一个字面量常量的引用

6.3.7.7. InnerClasses 属性

1、InnerClass 属性用于记录内部类与宿主类之间的关联，如果一个类中定义了内部类，那编译器将会为他以及它所包含的内部类生成 InnerClass 属性

表格 6-18InnerClass 属性结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_classes	1
inner_classes_info	inner_classes	number_of_class

1) number_of_classes：代表需要记录多少个内部类信息，每一个内部类信息都由一个 inner_classes_info 表进行描述

表格 6-19inner_classes_info 表的结构

类型	名称	数量
u2	inner_class_info_index	1
u2	outer_class_info_index	1
u2	inner_name_index	1
u2	inner_class_access_flags	1

1) inner_class_info_index 和 out_class_info_index 都是指向 CONSTANT_Class_info 型常量的引用，分别代表了内部类和宿主的符号引用
2) inner_name_index：指向常量 CONSTANT_Utf8_info 型常量的索引，代表这个内部类的名称，如果是匿名内部类，那么这项值是 0
3) inner_class_access_flags：内部类的访问标志，类似于类的 access_flags

标志名称	标志值	含义
ACC_PUBLIC	0x0001	内部类是否 public
ACC_PRIVATE	0x0002	内部类是否 private
ACC_PROTECTED	0x0004	内部类是否 protected
ACC_STATIC	0x0008	内部类是否 static
ACC_FINAL	0x0010	内部类是否 final
ACC_INTERFACE	0x0020	内部类是否为接口
ACC_ABSTRACT	0x0400	内部类是否 abstract
ACC_SYNTHETIC	0x1000	内部类是否并非由用户代码产生的
ACC_ANNOTATION	0x2000	内部类是否是一个注解
ACC_ENUM	0x4000	内部类是否是一个枚举

6.3.7.8. Deprecated 及 Synthetic 属性

1、这两个属性都属于标志类型的布尔值，只有存在和没有的区别，没有属性值的概念

2、**Deprecated 属性用于表示某个类、字段或者方法，已经被程序作者定为不再推荐使用，它可以通过在代码中使用@deprecated 注解进行设置**

3、Synthetic 属性代表此字段或者方法并不是由 Java 源码直接产生的，而是由编译器自行添加的

- 在 JDK 1.5 之后，标志一个类、字段或者方法是编译器自动产生的，也可以设置它们访问标志中的 ACC_SYNTHETIC 标志位，其中最典型的例子就是 Bridge Method
- 所有由非用户代码产生的类、方法及字段都应当至少设置 Synthetic 属性和 ACC_SYNTHETIC 标志位中的一项，唯一的例外是是实例构造器<init>和类构造器<clinit>方法

表格 6-20Deprecated 及 Synthetic 属性的结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1

1) 其中，attribute_length 数据项的值必须为 0x00000000，因为没有任何属性值需要设置

6.3.7.9. StackMapTable 属性

1、该属性在 JDK 1.6 发布后增加到了 Class 文件规范中，它是一个复杂的变长属性，位于 Code 属性的属性表中

2、这个属性会在虚拟机类加载的字节码验证阶段被新类型检查验证器(Type Checker)使用，目的在于代替以前比较消耗性能的基于数据流分析的类型推导验证器

3、<未完成>

6.3.7.10. Signature 属性

1、该属性在 JDK 1.5 发布后增加到了 Class 文件规范之中，它是一个可选的定长属性，可以出现于类、字段表和方法表结构的属性表中

2、在 JDK 1.5 中大幅增强了 Java 语言的语法，在此之后，任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量(Type Variables)或参数化类型(Parameterized Types)，则 Signature 属性会为它记录泛型签名信息

3、**之所以要专门使用这样一个属性去记录泛型类型，是因为 Java 语言的泛型采用的是擦除法实现的伪泛型，在字节码(Code 属性)中，泛型信息(类型变量、参数化类型)之后都通通被擦除掉**

4、**使用擦除的好处是实现简单(主要修改 Javac 编译器，虚拟机内部只做了很少的改动)**

5、***擦除实现的泛型坏处就是运行期无法像 C#等有真泛型支持的语言那样，将泛型类型与用于定义的普通类型同等对待，例如运行期做反射时无法或得到泛型信息，Signature 属性就是为了弥补这个缺陷而增设的***

表格 6-21Signature 属性的结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	signature_index	1

1) signature_index 项的值必须是一个对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info 结构，表示类签名、方法类型签名或字段类型签名

6.3.7.11. BootstrapMethods 属性

1、该属性在 JDK 1.7 发布后增加到了 Class 文件规范之中，它是一个复杂的变长属性，位于类文件的属性表中

2、这个属性用于保存 invokedynamic 指令引用的引导方法限定符

3、<未完成>

6.4. 字节码指令简介

1、Java 虚拟机就的指令由一个字节长度的、代表着某种特定操作含义的数字(称为操作码，Opcode)以及跟随其后的零个或多个代表此操作所需参数(称为操作数，Operands)而构成

2、由于 Java 虚拟机采用面向操作数栈而不是寄存器的架构，所以大多数指令都不包含操作数，只有一个操作码

3、字节码指令集是一种具有鲜明特点、优劣势都很突出的指令集架构

- 由于限制了 Java 虚拟机操作码的长度为一个字节(0~255)，这意味着操作码总数不可能超过 256 条
- 由于 Class 文件格式放弃了编码后代码的操作数长度对齐，这就意味着虚拟机处理那些超过一个字节数据的时候，不得不在运行时从字节中重建出具体数据的结构，如果要将一个 16 位长度的无符号整数使用两个无符号字节存储起来(byte1<<8)|byte2
- 这种操作在某种程度上会导致解释执行字节码时损失一些性能，但这样做的优势也很明显，放弃了操作数长度对齐，这就意味着可以省略很多填充和间隔符号
- 用一个字节来代表操作码，也是为了尽可能获得短小精干的编译代码，这种追求尽可能小数据量、高传输效率的设计是由 Java 语言设计之初面向网络、智能家电的技术背景所决定的，并一直沿用至今

4、Java 解释器的伪代码模型

```
do{
    自动计算 PC 寄存器值+1
    根据 PC 寄存器的指示位置，从字节码流中取出操作码
    if(字节码存在操作数) 从字节码流中取出操作数
    执行操作码所定义的操作
}while(字节码流长度>0)
```

6.4.1. 字节码与数据类型

1、Java 虚拟机的指令集中，大多数的指令都包含了其操作对应的数据类型

- 例如 iload 指用于从局部变量表中加载 int 型的数据到操作数栈中
- fload 指令加载的则是 float 类型的数据
- 这两条指令的操作在虚拟机内部可能会是由同一段代码实现的，但在 Class 文件中它们必须拥有各自独立的代码

2、**对于大部分与数据类型相关的字节码指令，它们的操作码助记符中都有特殊的字符来表明专门为哪种数据类型服务**

- i：int 类型
- l：long 类型
- s：short 类型
- b：byte 类型
- c：char 类型
- f：float 类型
- d：double 类型
- a：reference 类型

3、有一些指令的助记符中没有明确地指明操作类型的字母，如 arraylength 指令，它没有代表数据类型的特殊字符，但操作数永远只能是一个数组类型的对象

4、由于 Java 虚拟机的操作码长度只有一个字节，所以包含了数据类型的操作码就为指令集的设计带来了很大的压力：如果每一种与数据类型相关的指令都支持 Java 虚拟机所有运行时数据类型的话，那指令的数量恐怕就会超过一个字节所能表示的数量范围了

- 因此，Java 虚拟机的指令集对于特定的操作只提供了有限的类型相关指令

去支持它

- 换句话说，指令集将会故意被设计成非完全独立的，**Java 虚拟机规范中把这种特性称为"Not Orthogonal"，即并非每种数据类型和每一种操作数都有对应的指令**
- 有一些单独的指令可以在必要的时候用来将一些不支持的类型转换为可以被支持的类型

5、大部分的指令都没有支持整数类型 byte、char 和 short，甚至没有任何指令支持 boolean 类型

- 编译器会在编译期或运行期将 byte 或 shrot 类型的数据带符号扩展(Sign-Extend)为相应的 int 类型数据，将 boolean 和 char 类型数据零位扩展(Zero-Extend)为相应的 int 类型数据
- 因此大多数对于 boolean、byte、short、char 类型数据的操作，实际上都是使用相应的 int 类型作为运算类型(Computational Type)

表格 6-22 Java 虚拟机指令集所支持的数据类型

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

6.4.2. 加载和存储指令

1、加载和存储指令用于将数据在栈帧中的局部变量和操作数栈之间来回传输

- 将一个局部变量加载到操作数栈：iload、iload_<n>、lload、lload_<n>、fload、fload_<n>、dload、dload_<n>、aload、aload<n>
- 将一个数值从操作数栈存储到局部变量表：istore、istore_<n>、lstore、lstore_<n>、fstore、fstore_<n>、dstore、dstore_<n>、astore、astore_<n>
- 将一个常量加载到操作数栈：bipush、sipush、ldc、ldc_w、ldc2_w、aconst_null、iconst_ml、iconst_<i>、lconst_<l>、fconst_<f>、dconst_<d>
- 扩充局部变量表的访问索引的指令：wide
- 有一部分以尖括号结尾，例如 iload_<n>，代表了一组指令(iload_0、iload_1、iload_2、iload_3 这几条指令)
  - **对于 iload_1、iload_2、iload_3 这三个命令是没有操作数的**
  - **对于 iload　n(n>=4) 这个命令是有操作数的**
  - **其他也类似**

6.4.3. 运算指令

1、**运算或算数指令用于对操作数栈上的两个值(书上写的是两个操作数栈上的值???)进行某种特定运算，并把结果重新存入到操作数栈顶**

2、大体上算数指令可以分为两种：

- 对整数数据进行运算的指令与对浮点型数据进行运算的指令
- 无论是哪种算数指令，都是用 Java 虚拟机的数据类型
- 由于没有直接支持 byte、short、char、boolean 类型的算数指令，对于这些数据的运算，应使用操作 int 类型的指令代替

3、算数指令

- 加法指令：iadd、ladd、fadd、dadd
- 减法指令：isub、lsub、fsub、dsub
- 乘法指令：imul、lmul、fmul、dmul
- 触发指令：idiv、ldiv、fdiv、ddiv
- 求余指令：irem、lrem、frem、drem
- 取反指令：ineg、lneg、fneg、dneg
- 位移指令：ishl、ishr、iushr、lshl、lshr、lushr
- 按位或指令：ior、lor
- 按位与指令：iand、land
- 按位异或指令：ixor、lxor
- 局部变量自增指令：iinc
- 比较指令：dcmpg、dcmpl、fcmpg、fcmpl、lcmp

4、溢出

- Java 虚拟机规范没有明确定义过整型数据溢出的具体运算结果
- 仅规定了在处理整型数据时，**只有除法指令以及求余指令(irem lrem)中当出现除数为 0 时会导致虚拟机抛出 ArithmeticException 异常**
- **其余任何整数运算都不应该抛出运行时异常**

5、浮点数

- Java 虚拟机要求在进行浮点数运算时，所有的运算结果都必须舍入到适当的精度，非精确的结果必须舍入为可被表示的最接近的精确值

- 把浮点数转换为整数时，采用向零舍入模式，小数部分被直接丢掉，在目标数值类型中选择一个最接近但是不大于原值的数字来作为最精确的舍入结果???((int)-1.23 的结果是-1)

6.4.4. 类型转换指令

1、类型转换指令可以将两种不同的数值类型进行相互转换，这些转换操作一般用于实现用户代码中的显式类型转换操作

2、Java 虚拟机直接支持一下数值类型的宽化类型转换(Widening Numeric Conversions，即小范围向大范围类型的安全转换)

- int 类型到 long、float、double
- long 到 float、double 类型
- float 类型到 double 类型

3、处理窄化类型转换(Narrowing Numeric Conversions)时，必须显式地使用转换指令来完成，包括

- i2b
- i2c
- i2s
- l2i
- f2i
- f2l
- d2i
- d2l
- d2f

6.4.5. 对象创建于访问指令

1、虽然类实例和数组都是对象，但 Java 虚拟机对实例和数组的创建与操作使用了不同的字节码指令，对象创建后，就可以通过对象访问指令获取对象实例或者数组实例中的字段或者数组元素

- 创建类实例的指令：new
- 创建数组的指令：newarray、anewarray、multianewarray
- 访问类字段(static 字段、或称为类变量)和实例字段(非 static 字段，或者称为实例变量)的指令：getfield、putfield、getstatic、putstatic
- 把一个数组元素加载到操作数栈的指令：baload、caload、saload、iaload、laload、faload、daload、aaload
- 将一个操作数栈的值存储到数组元素中的指令：bastore、castore、sastore、iastore、fastore、dastore、aastore
- 取数组长度的指令：arraylength
- 检查类实例类型的指令：instanceof、checkcast

6.4.6. 操作数栈管理指令

1、如果操作一个普通数据结构中的堆栈那样，Java 虚拟机提供了一些用于直接操作操作数栈的指令

- 将操作数栈顶的一个或两个元素出栈：pop、pop2
- 赋值栈顶一个或两个数值并将赋值值或双份的赋值值重新压入栈顶：dup、

dup2、dup_x1、dup2_x1、dup_x2、dup2_x2

- 将栈最顶端的两个数值互换：swap

6.4.7. 控制转移指令

1、控制转义指令可以让 Java 虚拟机有条件或无条件地从指定位置指令而不是控制转移指令的下一条指令继续执行程序，从概念模型上理解，可以认为控制转义指令就是在有条件或无条件地修改 PC 寄存器的值

- 条件分支：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull、if_icmpeq、if_icmpne、if_icmplt、if_icmpgt、if_icmple、if_icmpge、if_acmpeq 和 if_acmpne
- 复合条件分支：tableswitch、lookupswitch
- 无条件分支：goto、goto_w、jsr、jsr_w、ret
- 对于 boolean、char、byte、shrot 类型的条件分支比较操作，都是使用 int 类型的比较指令来完成
- 对于 long 类型、float 类型和 double 类型的条件分支比较操作，则会先执行相应类型的比较运算指令(dcmpg、dcmpl、fcmpg、fcmpl、lcmp)，运算指令会返回一个整型值到操作数栈中，随后再执行 int 类型的条件分支比较操作来完成整个分支跳转
- **由于各种类型的比较最终都会转化为 int 类型的比较操作，int 类型比较是否方便完善就显得尤为重要，所以 Java 虚拟机提供的 int 类型的条件分支指令是最为丰富和强大的**

6.4.8. 方法调用和返回指令

1、方法调用指令介绍

- invokevirtual：用于调用对象的实例方法，根据对象的实际类型进行分派(虚方法分派)，这也是 Java 语言中最常见的方法分派方式(动态???)
- invokeinterface：用于调用接口方法，它会在运行时搜索一个实现了这个接口方法的对象，找出合适的方法进行调用
- invokespecial：用于调用一些需要特殊处理的实例方法，包括实力初始化方法、私有方法和父类方法
- invokestatic：用于调用类方法(static)
- invokedynamic：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 4 条指令的分派逻辑都固化在 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决定的

2、**方法调用指令**与数据类型无关，而**方法返回指令**是根据返回值的类型区分的，包括 ireturn(当返回值是 boolean、byte、char、short 和 int 类型时使用)、lreturn、freturn、dreturn 和 areturn，另外还有一条 return 指令供声明为 void 的方法、实例初始化方法以及类和接口的类初始化方法使用

6.4.9. 异常处理指令

1、在 Java 程序中显式抛出异常的操作(throw 语句)都由 athrow 指令来实现，除了用 throw 语句显式抛出异常情况之外，Java 虚拟机规范还规定了许多运行时异常会在其他 Java 虚拟机指令检测到异常状况时抛出

2、在 Java 虚拟机中，处理异常(catch 语句)不是由字节码指令来实现的，而是采用异常表来完成的

6.4.10. 同步指令

1、Java 虚拟机可以支持方法级的同步和方法内部一段指令序列的同步，这两种同步结构都是使用管理(Monitor)来支持的

2、方法级同步是隐式的，即无需通过字节码指令来控制，它实现在方法调用和返回操作之中

- 虚拟机可以从方法常量池的方法表结构中的 ACC_SYNCHRONIZED 访问标志得知一个方法是否声明为同步方法
- 当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程就要求先成功持有**管程???**，然后才能执行方法，最后当方法完成(无论是正常还是非正常完成)时释放管程
- 在方法执行期间，执行线程持有了管程，其他任何线程都无法在获取到同一个管程
- 如果一个同步方法执行期间抛出了异常，并且在方法内部无法处理此异常，那么这个同步方法所持有的管程将在异常抛到同步方法之外时自动释放

3、同步一段指令集序列通常是由 Java 语言中的 synchronized 语句块来表示的

- Java 虚拟机的指令集有 monitorenter 和 monitorexit 两条指令来支持 synchronized 关键字的语义
- 正确实现 synchronized 关键字需要 Javac 编译器与 Java 虚拟机两者共同协作支持
- 编译器必须确保无论方法通过何种方式完成，方法中调用过的每条 monitorenter 指令都必须执行其对应的 monitorexit 指令，而无论这个方法是正常结束还是异常结束

6.5. 公有设计和私有实现

1、Java 虚拟机规范描绘了 Java 虚拟机应有的共同程序存储格式：**Class 文件格式以及字节码指令集，这些内容与硬件、操作系统及具体的 Java 虚拟机实现之间是完全独立的**，虚拟机实现者可能更愿意把它们看做是程序在各种 Java 平台之间互相安全地交互的手段

2、理解共有设计与私有实现之间的分界线是非常有必要的，Java 虚拟机实现必须能够读取 Class 文件并精确实现包含在其中的 Java 虚拟机代码的语义

- 拿着 Java 虚拟机规范一成不变地逐字实现其中要求的内容当然是一种可行途径
- 但一个优秀的虚拟机实现，在满足虚拟机规范的约束下对具体实现做出修改和优化也是完全可行的，并且虚拟机规范中明确鼓励实现者这样做
  - 只要优化后 Class 文件依然可以被正确读取，并且包含在其中的语义能得到完整的保持，那实现者可以选择任何方式去实现这些语义
  - 虚拟机在后台如何处理 Class 文件完全是实现者自己的事情，只要它在外部接口上看起来与规范描述一致即可

3、虚拟机实现的方式主要有以下两种

- 将输入的 Java 虚拟机代码在加载或执行时翻译成另外一种虚拟机的指令集

- 将输入的 Java 虚拟机代码在加载或执行时翻译成宿主 CPU 的本地指令集(即 JIT 代码生成技术)

6.6.Class 文件结构的发展

1、Class 文件结构一直处于比较稳定的状态

- Class 文件的主体结构、字节码指令的语义和数量几乎没有出现过变动
- 所有对 Class 文件格式的改进都集中在访问标志、属性表这些设计上就可扩展的数据结构中添加内容

2、Class 文件格式所具备的平台中立(不依赖于特定硬件及操作系统)、紧凑、稳定和扩展的特点，是 Java 技术体系实现平台无关、语言无关两项特性的重要支柱

Chapter 7. 虚拟机类加载机制

1、代码编译的结果从本地机器码转变为字节码，是存储格式发展的一小步，确实编程语言发展的一大步

7.1. 概述

1、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，这就是虚拟机类加载机制

2、与那些在编译时需要进行连接工作的语言不同，在 Java 语言里，类型的加载、连接和初始化过程都是在程序运行期完成的

- 这种策略虽然会令类加载时稍微增加一些性能开销，但是会为 Java 应用提供高度的灵活性
- Java 里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，如果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 预定义的和自定义类加载器，让一个本地的应用程序可以在运行时从网络或其他地方加载一个二进制流作为程序代码的一部分
  - 这种组装应用程序的方式目前已广泛应用于 Java 程序之中

3、本章描述的两个约定

- 每个 Class 文件都有可能代表着 Java 语言中的一个类或接口
- 提到的 Class 文件并非某个存在于具体磁盘中的文件，而是指一串二进制的字节流

7.2. 类加载的时机

1、类从被加载到虚拟机内存中开始，到卸载出内存为止，整个生命周期包括：

- 加载(Loading)
- 验证(Verification)
- 准备(Preparation)
- 解析(Resolution)
- 初始化(Initialization)
- 使用(Using)和卸载(Unloading)
- **其中验证、准备、解析部分统称为连接(Linking)**

2、**加载、验证、准备、初始化和卸载这 5 个阶段的顺序是确定的，类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

- 解析在某些情况下可以在初始化阶段之后在开始，这是为了支持 Java 语言的运行时绑定(动态绑定)
- **注意：按部就班指的是按部就班地开始，而非按部就班地进行或完成，因为这些阶段通常都是互相交叉地混合式进行的，通常会在一个阶段执行过程中调用、激活另外一个阶段**

3、什么情况下需要开始类加载过程的第一个阶段：加载?

- 虚拟机规范中并没有强制约束，这点可以交由虚拟机的具体实现来自由把握
- **对于初始化阶段，虚拟机规范严格规定了有且仅有 5 种情况必须立即对类进行"初始化"(加载、验证、准备自然需要在此之前开始)**
  - 遇到 new、getstatic、putstatic 或 invokestatic 这四条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化**(使用 new 关键字实例化对象的时候，读取或设置一个类的静态字段时(被 final 修饰，已在编译期把结果放入常量池的静态字段除外)，以及调用一个类的静态方法的时候)**
  - 使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化
  - 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化
  - 当虚拟机启动时，用户需要指定一个要执行的主类(包含 main()方法的类，虚拟机会先初始化这个主类)
  - 当使用 JDK 1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后解析结果 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行初始化，则需要先触发初始化

4、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

5、Java 语言中对数组的访问比 C/C++相对安全是因为这个类(容纳相应类型的数组的类)封装了数组元素的访问方法，而 C/C++直接翻译为对数组指针的移动

- 在 Java 语言中，当检查到发生数组越界时会抛出 java.lang.ArrayIndexOutofBoundsException 异常

6、接口的加载过程与类加载过程稍微有些不同

- 接口也有初始化过程，这点与类一致
- 接口中不能有 static{}语句块
- 编译器仍然会为接口生成<clinit>类构造器
- **必须初始化的情况中的第三种与类有区别**：
  - 当一个类在初始化时，要求其父类全部都已经初始化过
  - 当一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候(如引用接口中定义的常量)才会初始化

7、神奇的例子

```
public class ConstClass{
    static{System.out.println("ConstClass.init!");}
    public static final String HELLOWORLD="Hello world";
}
public class NotInitialization{
    public static void main(String[] args){
        System.out.println(ConstClass.HELLOWORLD);
    }
```

}

- 结果不会输出"ConstClass init!"，因为虽然在 Java 源码中引用了 ConstClass 类中的常量，但是在编译阶段通过**常量传播优化**，已经将常量的值存储到了 NotInitialization 类的常量池中，即这两个类在编译成 Class 之后就不存在任何联系了

7.3. 类加载的过程

1、加载、验证、准备、解析、初始化这五个阶段所执行的具体动作

7.3.1. 加载

1、**"加载"是"类加载"过程的一个阶段，在该阶段，虚拟机需要完成以下 3 件事**

1) 通过一个类的全限定名来获取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

2、**虚拟机规范的这三点要求不算具体，并没有指明从哪里获取、怎样获取**

- 从 ZIP 包中读取，这很常见，最终成为日后的 JAR、EAR、WAR 格式的基础
- 从网络中获取，这种场景典型的应用就是 Applet
- 运行时计算生成，这种场景使用得最多的就是动态代理技术，在 java.lang.reflect.Proxy 中，就是用了 ProxyGenerator.generateProxyClass 来为特定接口生成形式为"*$Proxy"的代理类的二进制字节流
- 由其他文件生成，典型场景是 JSP 应用，即由 JSP 文件生成对应的 Class 类
- 从数据库中读取，这种场景相对少些，例如有些中间件服务器(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

3、**相对于类加载过程的其他阶段**

- **一个非数组类的加载阶段(准确的说，是加载阶段中获取二进制字节流的动作)是开发人员可控性最强的**
  - 因为加载阶段既可以使用系统提供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，开发人员可以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 loadClass()方法)
- 对于**数组类**而言
  - 数组类本身不通过类加载器创建，它是由 Java 虚拟机直接创建的，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)最终是要靠类加载器去创建
  - 一个数组类(下面简称 C)创建过程遵循以下原则
    - 如果**数组类的<组件类型>**(Component Type，指的是数组去掉第一个维度的类型)是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组 C 将在加载该组件的类加载器的类名称空间上被标识**(一个类必须与类加载器一起确定唯一性)**
    - 如果数组的组件不是引用类型，例如 int[]，Java 虚拟机会把数组 C 标

记为与引导类加载器关联

- 数组类的可见性与它的组件类型的可见一致性，如果组件类型不是引用类型，那数组类的可见性将默认为 public

4、加载阶段完成后

- 虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区中的数据存储格式由虚拟机实现自行定义，虚拟机规范未规定此区域的具体数据结构
- 然后在内存中实例化一个 java.lang.Class 类的对象**(并没有明确规定是在 Java 堆中，对于 HotSpot 虚拟机而言，Class 对象比较特殊，它虽然是对象，但是存放在方法区)**，这个 Class 对象将作为程序访问方法区中的这些类型数据的外部接口

5、加载阶段与连接阶段的部分内容是交叉进行的，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内容，这两个阶段的**开始时间**仍然保持着固定的先后顺序

7.3.2. 验证

1、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当前虚拟机的要求，并且不会危害虚拟机自身的安全**

2、Java 语言本身是相对安全的语言(相对于 C/C++来说)

- 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之类的事情，如果这样做了，编译器将拒绝编译
- **Class 文件并不一定要用 Java 源码编译而来，可以使用任何途径产生，甚至包括用 16 进制编辑器直接编写来产生 Class 文件，在字节码层面上，上述 Java 代码无法做到的事情都是可以实现的**
- 如果虚拟机不检查输入的字节流，对其完全信任的话，很可能会因载入了有害的字节流而导致系统崩溃，所以验证是虚拟机对自身保护的一项重要工作

4、验证阶段大致会完成下面 4 个动作

1) 文件格式验证：验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理
   - 是否以魔数 0xCAFEBABE 开头
   - 主次版本号是否在当前虚拟机的处理范围之内
   - 常量池中的常量是否有不被支持的常量类型(检查常量 tag 标志)
   - 指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量
   - CONSTANT_Utf8_info 型的长两种是否有不符合 UTF8 编码的数据
   - Class 文件中各个部分及文件本身是否有被删除的或附加的其他信息
   - .....
   - **这个阶段的主要目的是保证输入的字节流能正确地解析并且存储于方法区之内，格式上符合描述一个 Java 类型信息的要求**
   - 只有通过这个阶段的验证之后，字节流才会进入内存的**方法区**中进行存储

2) 元数据验证：对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求
   - 这个类是否有父类(除了 Object 之外，任何类都有父类)
   - 这个类的父类是否继承了不允许被继承的类(被 final 修饰的类)
   - 如果这个类不是抽象类，是否实现了其父类或接口中要求实现的所有方法
   - 类中的字段、方法、是否与父类产生矛盾
   - .....
   - **这个阶段的主要目的是对类的元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息**
3) 字节码验证：通过数据流和控制流分析，确定程序语义是合法的，符合逻辑的，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事件
   - 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，例如不会出现类似这样的情况：在操作栈放置了一个 int 类型的数据，使用时却按 long 类型来加载如本地变量表中
   - 保证跳转指令不会跳转到方法体以外的字节码指令上
   - 保证方法体中的类型转换是有效的
   - .....
   - 如果一个类方法体的字节码没有通过字节码验证，那肯定是有问题的，但通过了字节码验证也未必是安全的，即便做了大量的检查，也无法保证"**通过程序去校验程序逻辑无法做到绝对准确--不能通过程序准确地检查出程序是否能在有限时间之内结束运行**"
   - 虚拟机设计团队为了避免过多的时间消耗在字节码验证阶段，在 JDK 1.6 之后的 Javac 编译器和 Java 虚拟机中进行了一项优化，给方法体的 Code 属性表中增加了一项"StackMapTable"属性，该属性描述了方法体中所有基本块(Basic Block，按照控制流拆分的代码块)开始时本地变量表和操作栈应有的状态，在字节码验证期间，就不需要根据程序推导这些状态的合理性，只需要检查 StackMapTable 属性中的记录是否合法即可
   - 但理论上 StackMapTable 属性也存在错误或者被篡改的可能
4) 符号引用验证：对类自身以外(常量池中的各种符号引用)的信息进行匹配性校验
   - 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以及简单名称所描述的方法和字段
   - 符号引用中的类、字段、方法的访问性是否可被当前类访问
   - 符号引用验证的目的是：确保解析动作能正常执行，如果无法通过符号引用验证，将会抛出 java.lang.IncompatibleClassChangeError 异常的子类

➢ **对于虚拟机类加载机制来说，验证阶段是非常重要的，但不是一定必要的阶段**，如果所运行的全部代码已经被反复使用和验证过，在实施阶段就可以考虑用-Xverfy:none 来关闭大部分的类验证措施，以缩短虚拟机类加载的时间

7.3.3. 准备

1、准备阶段正式为类变量分配内存并设置类变量初始值的阶段，**这些变量所使**

用的内存都将在方法区中进行分配

2、概念明确

- ***这时候进行内存分配的仅包括类变量(被 static 修饰的变量)***，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中
- 这里提到的初始值，通常情况下是零值
  - 假设定义一个类变量

    public static int value=123
  - 变量 value 在准备阶段后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 123 的 putstatic 指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把 value 赋值为 123 的动作将在初始化阶段才会开始

表格 7-1 基本数据类型的零值

数据类型	零值
int	0
long	0L
short	(short)0
char	'\u0000'
byte	(byte)0
boolean	false
float	0.0f
double	0.0d
reference	null

- 有一些特殊情况，如果类字段的属性表中存在 ConstantValue 属性，那在准备阶段变量 value 就会被初始化为 ConstantValue 属性所指定的值

7.3.4. 解析

1、解析阶段是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，引用的目标不一定已经加载到内存中
  - 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一致，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)**：
  - **直接引用可以是指向目标的指针、相对偏移量或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存在内存中

2、虚拟机规范中并未规定解析阶段发生的具体时间，只要求了在执行 anewarray、checkcast、getfield、getstatic、instanceof、invokedynamic、

invokeinterface、invokespecial、invokestatic、invokevertual、ldc、ldc_w、multianewarray、new、putfield、putstatic 这 16 个用于操作符号引用的字节码指令之前，先对它们所使用的符号引用进行解析

- 因此，虚拟机实现可以根据需要来判断到底是在类被加载器加载时就对常量池中的符号引用进行解析还是等到一个符号引用将要被使用前才去解析它

3、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 外**

- 虚拟机实现可以对第一次解析的结果进行缓存(在运行时常量池中记录直接引用，并把常量标识为解析状态)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，虚拟机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，那么后续的引用请求就已应当一直成功，否则将会收到相同的异常

4、对于 invokedynamic 指令

- 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
- **动态的含义：必须等到程序实际运行到这条指令时，解析动作才能进行**
- 静态的含义：可以在刚刚完成加载阶段，还没有开始执行代码时就进行解析

5、解析动作针对的对象

- 类或接口(CONSTANT_Class_info)
- 字段(CONSTANT_Fieldref_info)
- 类方法(CONSTANT_Methodref_info)
- 接口方法(CONSTANT_InterfaceMethodref_info)
- 方法类型(CONSTANT_MethodType_info)
- 方法句柄(CONSTANT_MethodHandle_info)
- 调用点限定符(CONSTANT_InvokeDynamic_info)

6、**类或接口的解析**

- 如果当前代码所处的类为 D，如果要把一个从未解析过的符号引用 N 解析为一个类或接口 C 的直接引用，**那么虚拟机需要完成以下 3 个步骤**

1) 如果 C 不是一个数组类型
   - 那虚拟机将会把代表 N 的全限定名传给 D 的类加载器去加载这个类 C
   - 在加载过程中，由于元数据验证、字节码验证的需要，有可能触发其他相关类的加载动作，例如加载这个类的父类或实现的接口
   - 一旦这个加载过程出现任何异常，解析过程宣告失败
2) 如果 C 是一个数组类型，并且数组的**元素类型**为对象，也就是 N 的描述符是类似"[Ljava/lang/Integer]"的形式
   - 那将会按照第一点的规则加载数组元素类型
   - 接着由虚拟机生成一个代表此数组维度和元素的数组对象
3) **如果上面的步骤没有任何异常，那么 C 在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还要进行符号引用验证，确认 D 是否具备对 C 的访问权限，若不具备，则抛出 java.lang.IllegalAccessError 异常**

7、**字段解析**

- 要解析一个未被解析过的字段符号引用，首先将会对字段表内 class_index 项中索引的 CONSTANT_Class_info 符号引用进行解析，即字段所属类或接口的符号引用，要是在这个过程中出现了异常，都会导致字段符号引用解析的失败，如果解析成功完成，将这个字段所属的类或接口用 C 表示，**接下来虚拟机按照以下步骤对 C 进行后续字段的搜索**
- 如果 C 本身包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束
- 否则，如果在 C 中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束
- 否则，如果 C 不是 java.lang.Object 的话，将会按照继承关系从下往上递归搜索其父类，如果在父类中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束
- 否则，查找失败，抛出 java.lang.NoSuchFiledError 异常
- **如果查找过程成功返回了引用，将会对这个字段进行权限验证，如果发现不具备对字段的访问权限，将抛出 java.lang.IllegalAccessError**
- 问题：P223 代码
  - **类的 static final 字段是可以继承的???(可以通过"子类类名.字段名"来引用)**

8、类方法解析

- 类方法解析的第一个步骤与字段解析一样，也需要先解析出类方法表的 class_index 项中的方法所属的类或接口的符号引用，如果解析成功，用 C 来表示这个类，**接下来虚拟机会按照如下步骤进行后续类方法搜索**
- 类方法和接口方法符号引用的常量类型定义是分开的，如果类方法表中发现 class_index 中索引的 C 是个接口，就直接抛出 java.lang.IncompatibleClassChangeError 异常
- 否则，在 C 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束
- 否则，在 C 的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束
- 否则，在类 C 实现的接口列表及他们的父接口中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果存在匹配的方法，说明 C 是一个抽象类，这时候查找结束，抛出 java.lang.AbstractMethodError 异常
- 否则宣告方法查找失败，抛出 java.lang.NoSuchMethodError
- **最后，如果查找过程成功返回了直接引用，将会对这个方法进行权限验证，如果发现不具备对此方法的访问权限，将抛出 java.lang.IllegalAccessError 异常**

9、接口方法解析

- 接口方法也需要先解析出接口方法表的 class_index 项中索引的方法所属的类或接口的符号引用，如果解析成功，依然用 C 表示这个接口，**接下来虚拟机会按以下步骤进行后续接口方法搜索**
- 与类方法解析不同，如果在接口方法表中发现 class_index 中索引 C 是个类而不是接口，那就直接抛出 java.lang.IncompatibleClassChangeError 异常

- 否则，在接口 C 中查找是否有简单名称和描述符都与目标相匹配的方法，如果有，则返回这个方法的直接引用，查找结束
- 否则，在接口 C 的父接口中递归查找，知道 java.lang.Object 类，看是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束
- 否则，宣告方法查找失败，抛出 java.lang.NoSuchMethodError 异常
- 由于接口中的所有方法默认都是 public 的，所以不存在访问权限的问题，因此接口方法的符号解析应当不会抛出 java.lang.IllegalAccessError

7.3.5. 初始化

1、类初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制，到了初始化阶段，才真正开始执行类中定义的 Java 代码(或者说字节码)

2、在准备阶段，变量已经赋值过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：**初始化阶段是执行类构造器<clinit>()方法的过程**

- <clinit>()方法是由编译器自动收集类中所有变量的赋值动作和静态语句块(static{})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**
- <clinit>()方法与类的构造器(或者说实例构造器<init>())不同
  - **它不需要显式地调用父类构造器**
  - **虚拟机会保证在子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕**
  - **因此在虚拟机中第一个被执行的<clinit>()方法肯定是 java.lang.Object**
- 由于父类的<clinit>()方法先执行，意味着父类中定义的静态语句块要优先于子类的变量赋值操作
- <clinit>()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句块，也就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

7.4. 类加载器

1、虚拟机设计团队把类加载阶段中"**通过一个类的全限定名来获取描述此类的二进制字节流**"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要的类，实现这个动作的代码模块称为"类加载器"

2、类加载器是 Java 语言的一项创新，也是 Java 流行的重要原因之一

- 类加载器最初是为了满足 Java Applet 的需求而开发的
- 虽然目前 Java Applet 技术基本上已经死掉，但是类加载器却在类层次划分、OSGi、热部署、代码加密等领域大放异彩

7.4.1. 类与类加载器

1、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远不限于类加载阶段

- 对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 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，通俗地说"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，被同一个虚拟机加载，只要加载它们的类加载器不同，那么这两个类就必定不等"
- 注意，上面提到的相等，包括代表类的 Class 对象的 equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果，也包括使用 instanceof 关键字做对象所属关系判定等情况

7.4.2. 双亲委派模型

1、从 Java 虚拟机的角度来讲，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
- 另外一种就是所有其他的类加载器，这些类加载器由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader

2、从 Java 开发人员的角度来看，类加载器还可以划分的更细致一些，绝大部分程序都会使用到以下 3 种系统提供的类加载器

- 启动类加载器(Bootstrap ClassLoader)
  - 负责将存放在<JAVA_HOME>\lib 目录中的，或者被-Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的(仅按文件名识别，例如 rt.jar，即名字不符合的类库即使放在 lib 目录中也不会被加载)类库加载到虚拟机内存中
  - 启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用 null 代替即可
- 扩展类加载器(Extension ClassLoader)：
  - 这个加载器由 sun.misc.Launcher$ExtClassLoader 实现，它负责加载<JAVA_HOME>\lib\ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库
  - 开发者可以直接使用扩展类加载器
- 应用程序类加载器(Application ClassLoader)：
  - 这个类加载器由 sun.misc.Launcher$AppClassLoader 实现
  - 由于这个类加载器是 ClassLoader 中的 getSystemClassLoader()方法的返回值，所以一般也称它为系统类加载器

3、类加载器之间的层次关系成为类加载器的双亲委派模型(parents Delegation Model)见图 7-1

- 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当有自己的父类加载器
- 这里类加载器之间的父子关系一般不会以继承(Inheritance)的关系来实现，而都是使用组合(Composition)关系来复用父加载器的代码

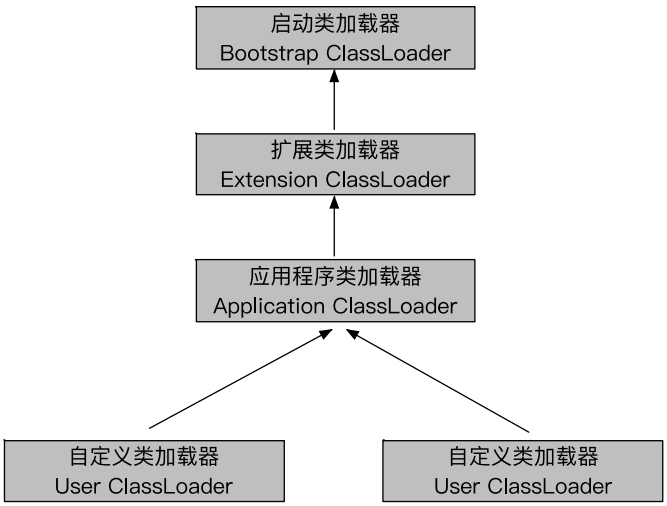

图 7-1 类加载器双亲委派模型

4、**双亲委派模型的工作过程**：

- 如果一个类加载器收到了加载请求，它首先不会自己去尝试加载这个类，而是把这个请求为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顶层的启动类加载器当中，只有父类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加载

5、双亲委派模型组织类加载器之间的关系的好处：Java 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

- 例如 java.lang.Object，它存放在 rt.jar 中，无论哪一个类加载器要加载这个类，最终都是会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是同一个类

Chapter 8. 虚拟机字节码执行引擎

8.1. 概述

1、执行引擎是 Java 虚拟机最核心的组成部分之一

2、**<虚拟机>是相对于<物理机>的概念**

- 这两种机器都有代码执行能力
- 物理机的执行引擎直接建立在处理器、硬件、指令集和操作系统层面上的
- 虚拟机的执行引擎则是由自己实现的，因此可以自行制定指令集与执行引擎的结构体系，并且能够执行那些不被硬件直接支持的指令集格式

3、在 Java 虚拟机规范中制定了虚拟机字节码执行引擎的概念模型，这个概念模型成为各种虚拟机执行引擎的统一外观(Facade)

- 在不同的虚拟机实现里面，执行引擎在执行 Java 代码的时候可能会有**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
- 从外观上来看，所有的 Java 虚拟机的执行引擎都是一致的：输入的是字节码文件，处理过程是字节码解析的等效过程，输出的是执行结果

8.2. 运行时栈帧结构

1、**<栈帧>**(Stack Frame)是用于支持虚拟机进行方法调用和方法执行的数据结构，它是虚拟机运行时数据区中的虚拟机栈(Virtual Machine Stack)

- 栈帧存储了方法的**局部变量表**、**操作数栈**、**动态链接**和**方法返回地址**等信息
- **每一个方法调用开始至执行完成的过程都对应着一个栈帧在虚拟机里面从入栈到出栈的过程**

2、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

- 在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的 Code 属性之中
- 一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现

3、一个线程中的方法调用链可能会很长，很多方法都同时处于执行状态

- 对于引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为**<当前栈帧>**(Current Stack Frame)
- 与这个栈相关两的方法称为**<当前方法>**(Current Method)
- **执行引擎运行的所有字节码指令都只对当前栈帧进行操作**

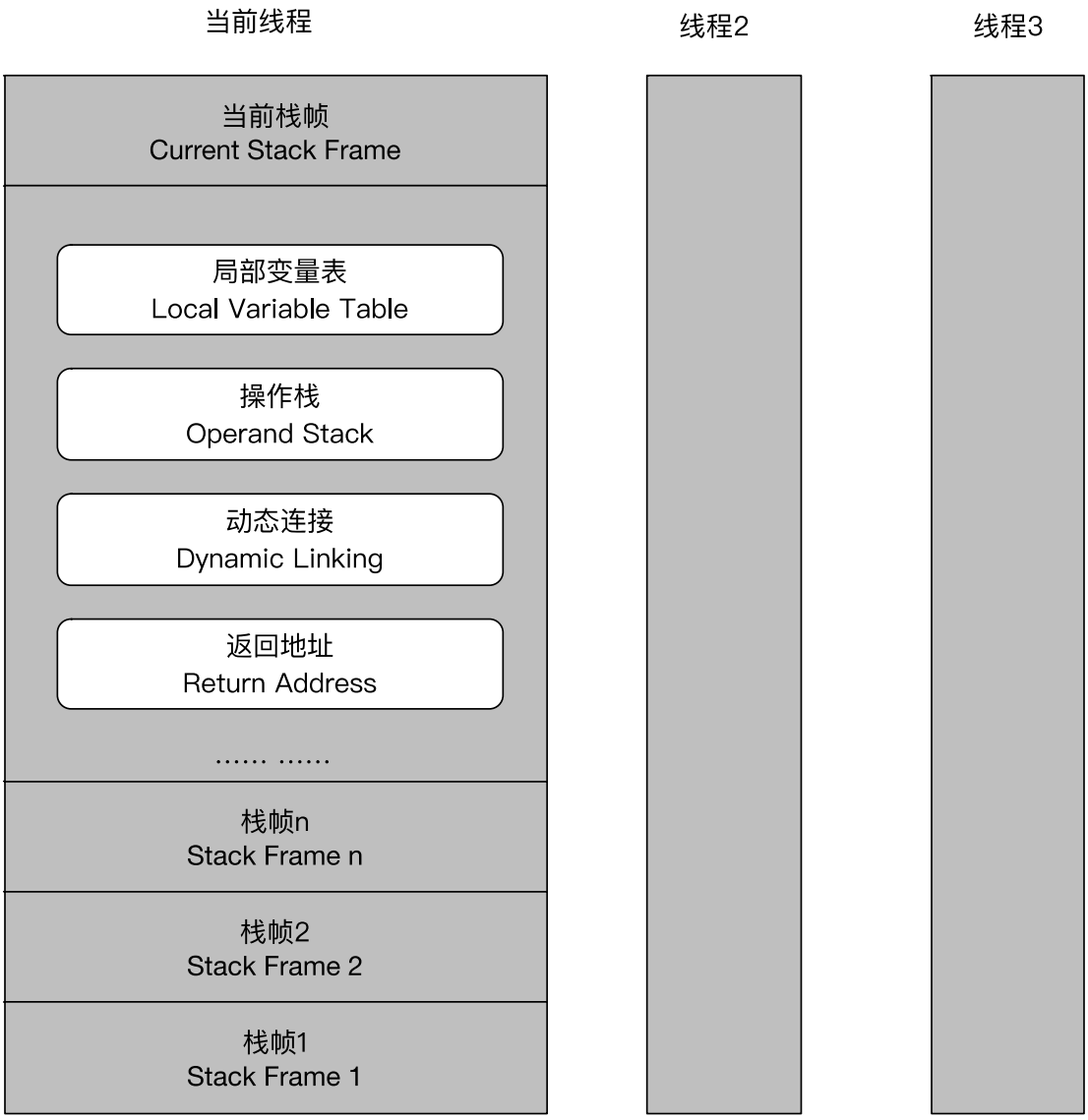

图 8-1 栈帧的概念结构

8.2.1. 局部变量表

1、局部变量表(Local Variable Table)是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量

2、在 Java 程序编译为 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要分配的局部变量表的最大容量

3、局部变量表的容量以变量槽(Variable Slot)为最小单位

- 虚拟机规范中并没有明确指明一个 Slot 应占用的内存空间大小，只是很有导向性地说到每个 Slot 都应该能存放一个 boolean、byte、char、short、int、float、reference 或 returnAddress 类型的数据
- 这八种数据类型，都可以用 32 位或更小的物理内存来存放，但这种描述与明确指出"每个 Slot 占用 32 位长度的内存空间"是有一定差别的
- 允许 Slot 的长度可以随着处理器、操作系统或虚拟机的不同而发生变化，只要保证在 64 位虚拟机中使用 64 位的物理内存空间去实现一个 Slot，虚拟机仍要用对齐和补白手段让 Slot 在外观上看起来与 32 位虚拟机中的一

致

4、**虚拟机的数据类型**

- 虚拟机中占 32 位以内的数据类型有 boolean、byte、char、short、int、float、reference、returnAddress 这 8 种类型
- 上述 8 种类型可以按照 Java 中语言对应的数据类型的概念去理解(**仅仅是理解而已，Java 语言与 Java 虚拟机中基本数据类型是存在本质差别的**)
- reference 类型表示一个对象实例的引用，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，但一般能通过这个引用做到两点
  - 从该引用中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索引
  - 此引用中直接或间接地查找到对象所属的数据类型在方法区中存储的类型信息
- returnAddress 类型目前已经很少了，它是为字节码指令 jsr、jsr_w、ret 服务的，指向了一条字节码指令的地址，很古老的 Java 虚拟机曾经使用这几条指令来实现异常处理，现在已经由异常表代替
- 对于 64 位数据类型，虚拟机会以高位对齐的方式为其分配两个连续的 Slot 空间
- Java 中明确的 64 位数据只有 long 和 double 两种，reference 可能是 32 或 64
- **由于局部变量表建立在线程的堆栈上，是线程私有的数据，无论读写两个连续的 Slot 是否为原子操作，都不会引起数据安全问题**

5、**虚拟机通过索引定位的方式使用局部变量表**，索引值的范围从 0 开始至局部变量表最大的 Slot 数量

- 如果访问的是 32 位数据类型的变量，索引 n 就代表了使用第 n 个 Slot
- 如果是 64 位数据类型的变量，则说明会同时使用 n 和 n+1 个 Slot
  - 不允许采用任何方式单独访问其中的某一个
  - Java 虚拟机规范中明确要求，如果遇到了进行这种操作的字节码，虚拟机应该在类加载的校验阶段抛出异常

6、方法执行时，**虚拟机使用局部变量表完成参数值到参数变量列表的传递过程**

- 如果执行的是实例方法(非 static 方法)，那局部变量表中第 0 位索引的 Slot 默认是用于传递方法所属对象实例的引用
- 在方法中可以通过关键字"this"来访问到这个隐含的参数
- 其余参数按照参数列表顺序，占用从 1 开始的局部变量 Slot，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 Slot

7、为了尽可能节省栈帧空间，局部变量表中的 Slot 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 Slot 就可以交给其他变量使用
- 不过这样的设计除了节省栈帧空间之外，还会伴随着一些额外的副作用

8、关于垃圾清理的例子

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

```
        }
        int a=0;
        System.gc();
    }
```

- 如果没有 int a=0 这句
  - 那么 System.gc()将不会对 placeholder 对应的内存进行回收
  - 虽然代码已经离开了 placeholder 的作用域，但在此之后，没有任何对局部变量表的读写操作，placeholder 原本所占用的 Slot 还没有被其他变量复用，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
  - 这种关联没有被及时打断，在大部分情况下影响都很轻微
  - **不用的变量，手动将其设置为 null 值是一个有意义的操作，这种操作在一种及特殊情形(对象占用内存大，此方法栈帧长时间不能被回收、方法调用次数还达不到 JIT 的编译条件)很有作用**
  - **但是没有必要把赋值为 null 作为普遍的编码规则来推广**
    - 从编码的角度来说，以恰当的变量作用域来控制变量回收时间才是最优雅的解决方案
    - 从执行角度来说，使用赋 null 值的操作来优化内存回收时建立在对字节码执行引擎概念模型的理解之上的
    - 虚拟机使用解释器执行时，通常与概念模型还比较接近，但经过 JIT 编译后，才是虚拟机执行代码的主要方式，赋 null 值的操作在经过 JIT 编译优化后就会被消除掉，对 GC Roots 的枚举也与解释器执行时期有巨大差别

9、局部变量表不像类变量那样存在"准备阶段"

- 类变量有两次赋初值的过程：一次在准备阶段，赋予系统初值，另外一次在初始化阶段，赋予程序员定义的初值
- **局部变量不同，如果一个局部变量定义了但没有赋初值是不能使用的**

8.2.2. 操作数栈

1、操作数栈(Operand Stack)也称为操作栈，它是一个后入先出(Last In First Out, FIFO)栈

- 与局部变量表一样，操作数栈的最大深度也在编译的时候写入到 Code 属性的 max_stacks 数据项中
- 操作数栈每个元素可以是任意的 Java 数据类型，包括 long 和 double
  - 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
- 在方法执行的任何时候，操作数栈的深度都不会超过 max_stacks 数据项中设定的最大值

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

3、操作数栈中元素的数据类型必须与字节码指令的序列严格匹配

4、在概念模型中，两个栈帧作为虚拟机栈的元素，是完全独立的，但在大多数虚拟机的实现里都会做一些优化处理，令两个栈帧出现一部分重叠

8.2.3. 动态连接

1、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

2、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中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转化为直接引用，这种转化称为<静态解析>
- 另外一部分将在每一次运行期间转化为直接引用，这部分称为<动态连接>

8.2.4. 方法返回地址

1、当一个方法开始执行后，只有两种方式可以退出这个方法

- 执行引擎遇到任意一个方法返回的字节码指令，这时可能会有返回值传递给上层的方法调用者，是否有返回值和返回值的类型将根据遇到何种方法返回值决定，这种退出方式称为正常完成出口(Normal Method Invocation Coompletion)
- 另一种退出方式，在方法在执行过程中遇到了异常(无论是虚拟机内部产生的异常，还是使用 athrow 字节码指令产生的异常)，并且这个异常没有在方体内得到处理，这种退出称为异常完成出口(Abrupt Method Invocation Completion)
  - 一个方法使用异常完成出口的方式退出，不会给它的上层调用者产生任何返回值

2、无论哪种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能正确执行

- 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方法的执行状态
- 一般来说，方法正常退出时，调用者的 PC 计数器(Program Counter)(程序计数器是用于存放下一条指令所在单元的地址的地方)可以作为返回地址，栈帧中很可能会保存这个计数器值
- 方法异常退出时，返回地址是要通过异常处理表来确定的，栈帧中一般不会保存这部分信息

3、方法退出的过程实际上就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
- 把返回值压入调用者栈帧的操作数栈
- 调整 PC 计数器的值以向方法调用指令后面的一条指令

8.2.5. 附加信息

1、虚拟机规范允许具体的虚拟机实现增加一些规范没有描述的信息到栈帧中，例如调试相关的信息，这部分信息完全取决于具体的虚拟机实现

2、实际开发中，一般会把动态连接、方法返回地址与其他附加信息归为一类，称为栈帧信息

8.3. 方法调用

1、***方法调用并不等同于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本，暂时还不涉及方法内部的具体运行过程***

2、Class 文件的编译过程中不包含传统编译中的连接步骤，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(相当于直接引用)

- 这个特性给 Java 带来了更强大的动态扩展能力
- 同时使得 Java 方法调用的过程变得相对复杂，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

8.3.1. 解析

1、所有方法调用中的目标方法在 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这种解析成立的前提：

- **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变的**
- 换句话说，调用目标在程序代码写好、编译器进行编译时就必须确定下来
- 这类的方法调用称为解析(Resolution)

2、在 Java 语言中符合"编译期可知，运行期不变"这个要求的方法，主要包括**静态方法**和**私有方法**两大类

- 前者与类直接相关联，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其他版本，因此它们都适合在类加载阶段进行解析(与静态绑定有关???)**

3、Java 虚拟机提供了 5 条方法调用字节码指令

- invokestatic：调用静态方法
- invokespecial：调用实例构造器<init>方法、私有方法和父类方法
- invokevirtual：调用所有的虚方法
- invokeinterface：调用接口方法，会在运行时再确定一个实现此接口的对象
- invokedynamic：现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 4 条调用指令，分派逻辑固化在 Java 虚拟机内部
- invokedynamic 指令的分派逻辑由用户所设定的引导方法决定

3、只要能被 invokestatic 和 invokespecial 指令调用的方法，都可以在解析阶段确定唯一的调用版本

- 包括：**静态方法**、**私有方法**、**实例构造器**、**父类方法**
- 它们在类加载的时候就会把符号引用解析为该方法的直接引用
- **这些方法称为非虚方法**

4、与上述相反，其他方法称为虚方法(除去 final 方法)

- 虽然 final 方法是使用 invokevirtual 指令来调用的，但是由于它无法被覆盖，没有其他版本，所以也无需对方法接受者进行多态选择，或者说多态的结果是唯一的
- 在 Java 语言规范中明确规定了 final 方法是非虚方法

5、解析调用时一个静态过程，在编译期就完全确定，在类装载的解析阶段就会把涉及的符号引用全部转换为可确定的直接引用，不会延迟到运行期才去确定
6、分配(Dispatch)调用则可能是静态的也可能是动态的

- 根据分配依据的宗量数可分为单分派和多分派
- 这两类分派方式的组合：~~静态单分派~~、静态多分派、动态单分派、~~动态多分派~~

8.3.2. 分派

1、Java 是一门面向对象的程序语言，因为 Java 具备 3 个特征

- 继承
- 封装
- 多态

2、分派调用过程揭示多态特征的一些最基本实现

- 例如重载和重写在 Java 虚拟机中是如何实现的
- 我们关心的不是 Java 语法该如何写，而是虚拟机如何确定正确的目标方法(<分派>)

8.3.2.1. 静态分派

1、静态类型和实际类型在程序中都可以发生一些变化，区别是

1) 静态类型的变化仅仅是在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是编译期可知的
2) 实际类型变化的结果在运行时才可确定，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

2、虚拟机(准确的说是编译器)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

- 虚拟机(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。因此，在编译阶段，Javac 编译器会根据参数的静态类型决定使用哪个重载版本

3、所以依赖静态类型来定位方法执行版本的分派动作称为静态分派
4、静态分派的典型应用是方法重载
5、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
6、在很多情况下确定重载版本并不是"唯一的"，往往只能确定一个"更加适合"的版本，这种模糊结论的主要原因是字面量不需要定义，所以字面量没有显式的静态类型，它的静态类型只能通过语言上的规则去理解和推断

8.3.2.2. 动态分派

1、动态分派和多态的另一个重要体现---重写(Override)有密切的关联
2、invokevirtual 指令的运行时解析过程

1) 找到操作数栈顶的第一个元素所指向的对象的*实例类型*，记作 C
2) 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，如果不通过，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) 若始终没有找到合适的方法，则抛出 java.lang.AbstractMethodError 异常

8.3.2.3. 单分派与多分派

1、方法的接受者与方法的参数统称为方法的<宗量>

2、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

- 单分派是根据一个宗量对目标方法进行选择
- 多分派是根据多于一个宗量对目标方法进行选择

3、例子说明

假设父类为 Father，子类为 Son(Father 有两个 eat 重载函数，分别接受 Apple 与 Banana，而子类 Son 重写了 Father 的这两个方法)

```
Father father=new Father();
Father son=new Son();
father.eat(new Apple());
son.eat(new Banana());
```

4、静态分派过程(编译器的选择过程)

- 选择目标方法的依据有以下两点
  - 方法的接受者的静态类型
  - 方法参数的静态类型
- 选择结果的最终产物是产生两条 invokevirtual 指令，分别指向常量池中 Father.eat(Apple)和 Father.eat(Banana)方法的符号引用。因为是根据两个宗量进行选择，所以 Java 语言的静态分派属于多分派类型

5、动态分派过程(虚拟机的选择过程)

- 在执行 son.eat(new Banan())这句代码时，更准确地说，是在执行这句代码所对应的 invokevirtual 指令时，由于编译期已经决定目标方法的签名必须为 eat(Banana)，虚拟机此时不会关心传递过来的参数到底是 Banana 还是 Apple，唯一可以影响虚拟机选择的因素只有此方法的接受者的实际类型是 Father 还是 Son。因为只有一个宗量作为选择依据，所以 Java 语言的动态分派属于但分派类型

6、Java 语言是一门静态多分派、动态单分派的语言

8.3.2.4. 虚拟机动态分派的实现

1、动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在类的方法元数据中搜索合适的目标方法

2、因此在虚拟机实际实现中基于性能的考虑，大部分实现都不会真正地进行频繁的搜索

3、最常用的"稳定优化"手段就是为类在方法区中建立一个虚方法表(Virtual Method Table，也称为 vtable，与此对应，在 invokeinterface 执行时也会用到接口方法表---Interface Method Table)

4、虚方法表中存放着各个方法的实际入口地址

5、如果某个方法在子类中没有被重写，那么子类的虚方法表里面的地址入口与父类相同方法的地址入口相同

6、如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类实现版本的入口

7、为了程序实现上的方便，具有相同签名的方法，**在父类、子类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，仅需要变更查找的方法表，就可以从不同的虚方法表中安索引转换出所需的入口地址

8、**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，虚拟机会把该类的方法表也初始化完毕

8.3.3. 动态类型语言支持

1、随着 JDK 1.7 的发布，字节码指令新增了 invokedynamic 指令，这条新增的指令是 JDK 7 实现"动态类型语言"(Dynamically Typed Language)支持而进行的改进之一，也是为 JDK 8 可以顺利实现 Lambda 表达式做技术准备

2、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
- 满足这个特征的语言有很多：APL、Clojure、Erlang、Groovy、JavaScript、Jython、Lisp、Lua、PHP、Prolog、Python、Ruby、Smalltalk 和 Tcl 等
- 在编译期就进行类型检查过程的语言(C++和 Java)就是最常用的静态类型语言
- 例子解释

  obj.println("hello world");
  - 假设在 Java 语言中，并且变量 obj 的静态类型为 java.io.PrintStream，那么 **obj 的实际类型就必须是 PrintStream 的子类才是合法的**，否则哪怕 obj 属于一个确实有 println(String)方法，但与 PrintStream 接口没有继承关系，那么代码依然不可运行---因为类型检查不合法
  - 假设在 JavaScript 中，无论 obj 具体是何种类型，只要这种类型定义中确实包含有 println(String)方法，那方法调用便可成功
  - 这种差别的原因：
    - **Java 语言在编译期已将 println(String)方法完整的符号引用生成出来，作为方法调用指令的参数存储到 Class 文件中**
    - 这个符号引用包含了此方法定义在哪个具体类型之中，方法的名字以及参数顺序、参数类型和方法返回值等信息
    - 通过这个符号引用，虚拟机可以翻译出这个方法的直接引用
    - **而在 JavaScript 等动态语言中，变量 obj 本身是没有类型的，变量 obj 的值才有具体类型**，编译时最多只能确定方法名称、参数、返回值这些信息，**而不会去确定方法所在的具体类型(即方法接受者不固定)，这个特点也是动态类型语言的重要特征**

3、静态类型与动态类型语言各自的优势

- 静态类型：在编译期确定类型，因此编译器可提供严谨的类型检查，这样与类型相关的问题能在编码时就及时发现，利于稳定性及代码达到更大规模
- 动态类型：在运行期确定类型，为开发人员提供更大的灵活性，在某些静态类型语言中需要大量"臃肿"代码来实现的功能，由动态类型语言来实现可能会更加清晰和简介，清晰和简介也就意味着开发效率的提升

4、JDK 1.7 与动态类型

- 能够在同一个虚拟机上可以达到静态类型语言的严谨性与动态类型语言

的灵活性，这是一件很美妙的事情

- JDK 1.7 以前的字节码指令中，***4 条方法调用指令(invokevirtual、invokestatic、invokespecial、invokeinterface)的第一个参数都是被调用方法的符号引用***(CONSTANT_Methodref_info 或者 CONSTANT_InterfaceMethodref_info 常量)，**而方法的符号引用在编译期产生**，而动态类型语言只有在运行期才能确定接受者类型
- JDK 1.7 新增了 invokedynamic 指令以及 java.lang.invoke 包

5、java.lang.invoke 包

- 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种方式以外，提供了一种新的动态确定目标方法的机制，称为 MethodHandle
- 可以与 C/C++中的 Function Pointer 以及 C#里面的 Delegate 类比一下
- C/C++中常用做法是把谓词定义为函数，用函数指针把谓词传递到泛型算法函数中，**但是 Java 做不到这一点，即没有办法单独把一个函数作为参数进行传递，普遍的做法是设计一个带有特定方法的接口，以实现了这个接口的对象作为参数，例如 Collections.sort()就是这样定义的**
- 在拥有 Method Handle 之后，Java 语言也可以拥有类似于函数指针或委托的方法别名的工具了

6、MethodHandle 的使用方法和效果与 Reflection 有众多相同之处，但他们有以下区别

- 从本质上讲，Reflection 和 MethodHandle 机制都是在模拟方法调用，但 Reflection 是在模拟 Java 代码层次的方法调用，而 MethodHandle 在模拟字节码层次的方法调用
  - MethodHandles.lookup 中的 3 个方法---findStatic()、findVirtual()、findSpecial()正是为了对应于 invokestatic、invokevirtual、invokeinterface 和 interspecial 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 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 对象所包含的信息多
  - 前者是方法在 Java 一端的全面映像，包含了方法签名、描述符以及方法属性表中各种属性的 Java 端表示方式，还包含执行权限等运行期信息
  - 后者仅仅包含与执行该方法的相关信息
  - 通俗来讲，Reflection 是重量级，MethodHandle 是轻量级
- 由于 MethodHandle 是对字节码的方法指令调用的模拟，所以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，而通过反射区调用方法则不行
- **Reflection API 的设计目标只是为 Java 语言服务的，而 MethodHandle 设计成可服务于所有 Java 虚拟机之上的语言，其中包括 Java 语言**

7、invokedynamic 指令

- 从某种程度上，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代码之中，让用户(包含其他语言的设计者)有更高的自由度**

- MethodHandle 采用 Java 代码和 API 来实现，invokedynamic 采用字节码和 Class 中的其他属性、常量来完成
- 每一处含有 invokedynamic 指令的位置都称作"动态调用点"(Dynamic 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 1.7 新加入的 CONSTANT_InvokeDynamic_info 常量，从这个常量可以得到三个信息：
  - 引导方法(Boostrap Method，此方法存放在新增的 BoostrapMethods 属性中)
  - 方法类型(MethodType)
  - 名称
  - 引导方法是固定的参数，并且返回值是 java.lang.invoke.CallSite 对象，这个代表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，虚拟机可以找到并执行引导方法，从而获得一个 CallSite 对象，最终调用要执行的目标方法
- <未完成>：代码清单不太对

8、掌控方法分派规则

- **invokedynamic 指令与前面 4 条"invoke*"最大的差别就是它的分派逻辑不是由虚拟机决定的，而是由程序员决定的**
- 在一个类中如何调用该类的祖父类的对应方法
  - 纯粹用 Java 语言很难处理这个问题，需要用到反射，**原因是当前类无法获取一个实际类型是祖父类型的对象的引用，而 invokevirtual 指令的分派逻辑是按照方法接受者的实际类型进行分派，这个逻辑是固化在虚拟机中的**
  - <未完成>：代码看不懂

8.4. 基于栈的字节码解释执行引擎

1、大部分的程序代码到物理机的目标代码或虚拟机能执行指令前需要进行以下步骤

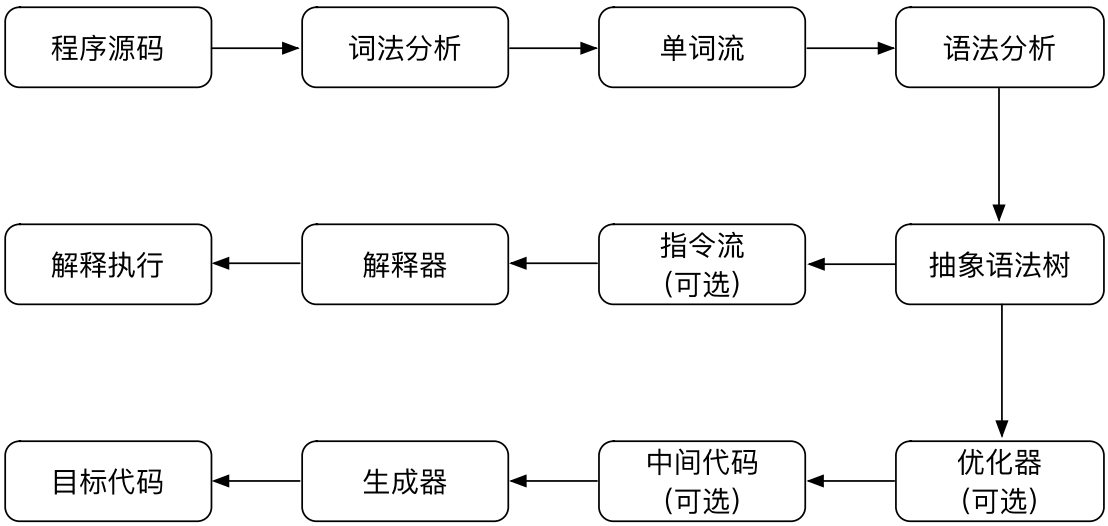

图 8-2 编译过程

2、对于一门具体语言的实现来说

- 词法分析、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++
- 也可以选择一部分步骤(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
- 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 执行器

3、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，因此 Java 程序的编译就是半独立的实现

8.4.1. 基于栈的指令集与基于寄存器的指令集

1、Java 编译器输出的指令流，基本上是一种基于栈的指令集架构(Instruction Set Architecture，ISA)，指令流中的指令大部分都是**零地址指令，它们依赖操作数栈进行工作**

2、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，说的通俗一点，就是现在我们主流 PC 机中直接支持的指令集架构，这些指令依赖寄存器进行工作

3、基于栈的指令集

- 优势
  - 可移植，而寄存器由硬件直接提供，如果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
  - 代码相对更紧凑(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)
  - 编译器实现更简单(不需要考虑空间分配的问题，所需空间都在栈上操作)
- 劣势
  - 执行速度相对较慢
  - 虽然代码更紧凑，但是完成相同功能所需的指令数量一般会比寄存器架构多
  - 频繁地访问栈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

Chapter 9. 类加载及执行子系统的案例与实战

9.1. 概述

1、在 Class 文件格式与执行引擎这部分中，用户的程序能直接影响的内容并不太多，Class 文件以何种格式存储，类型何时加载，如何连接，以及虚拟机何时执行字节码指令等都是由虚拟机直接控制的行为，用户程序无法对其进行改变
2、能通过程序进行操作的，主要是字节码生成与类加载器这两部分的功能

9.2. 案例分析

1、类加载器和字节码的案例各两个

9.2.1. Tomcat：正统的类加载器架构

1、主流的 Java Web 服务器，如 Tomcat、Jetty、WebLogic、WebSphere 或其他笔者没有列举的服务器，都实现了自己定义的类加载器(一般不止一个，双亲委派模型)，一个健全的 Web 服务器要解决以下几个问题

- 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以实现相互隔离，即服务器应当保证两个应用程序的类库可以相互独立使用
- 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以相互共享
- 服务器需要尽可能地保证自身的安全不受部署的 Web 应用程序影响，一般来说，基于安全考虑，服务器所使用的类库应该与应用程序的类库相互独立
- 支持 JSP 应用的 Web 服务器，大多数都需要支持 HotSwap(热替换)功能

2、由于存在上述问题，在部署 Web 应用时，单独的一个 ClassPath 就无法满足需求了

- 所以各种 Web 服务器都提供了好几个 ClassPath 路径供用户存放第三方类库，一般都以 lib 或 classes 命名
- 被放置到不同路径中的类库，具备不同的访问范围和服务对象，通常，每一个目录都会有一个相应的自定义的类加载器去加载放置在里面的 Java 类库

<未完成>：不知所云

Chapter 10. 早期(编译期)优化

10.1. 概述

1、Java 语言的"编译期"其实是一段"不确定"的操作过程

- 因为它可能是指一个**前端编译器**(其实叫做"编译器的前端"更准确一些)把*.java 文件转变为*.class 文件的过程
- 也可能是指**虚拟机的后端运行期编译器**(JIT 编译器，Just In Time Complier)把字节码转变成机器码的过程
- 还可能指使用**静态提前编译器**(<AOT>编译器，Ahead Of Time Compiler)直接把*.java 文件编译成本地机器代码的过程

2、比较有代表性的编译器

- 前端编译器：Sun 的 Javac、Eclipse JDT 中的增量式编译器(ECJ)
- JIT 编译器：HotSpot VM 的 C1、C2 编译器
- AOT 编译器：GNU Compiler for the Java(GCJ)、Excelsior JET

3、*本章提到的编译期和编译器都仅限于第一类编译过程*

- Javac 这类编译器对代码的运行效率几乎没有任何优化措施，虚拟机设计团队把对性能的优化集中到了后端的即时编译器中，这样可以让那些不是由 Javac 产生的 Class 文件也同样能享受到编译器优化所带来的好处
- Javac 做了许多针对 Java 语言编码过程的优化措施来改善程序员的编码风格和提高编码效率，**相当多新生的 Java 语法特性，都是靠编译器的"语法糖"来实现的，而不是依赖虚拟机的底层改进支持**
- **Java 中的即时编译器在运行期的优化过程对于程序运行来说更重要**
- **前端编译器在编译期的优化过程对于程序编码来说关系更加密切**

10.2. Javac 编译器

1、Javac 编译器不像 HotSpot 虚拟机那样使用 C++语言(包含少量 C 语言)实现，它本身就是一个由 Java 语言编写的程序，这为纯 Java 的程序员了解它的编译过程带来了很大的便利

10.2.1. Javac 的源码与调试

1、编译过程大致可以分为 3 个过程

1) 解析与填充符号表过程
2) 插入式注解处理器的注解处理过程
3) 分析与字节码生成过程

10.2.2. 解析与填充符号表

1、解析步骤包括了经典程序编译原理中的**词法分析**和**语法分析**两个过程

2、词法、语法分析

- 词法分析是将源代码的字符流转变为标记(Token)集合
  - **单个字符**是**程序编写过程**的**最小元素**
  - **标记**则是**编译过程**的**最小元素**
  - 关键字、变量名、字面量、运算符都可以称为标记

- 例如 int a=b+2 这句代码包含了 6 个标记，分别是 int、a、=、b、+、2
- 语法分析是根据 Token 序列构造抽象语法树的过程
  - 抽象语法树(Abstract Syntax Tree, AST)是一种用来描述程序代码语法结构的树形表示方式
  - 语法树的每一个节点都代表着程序代码中的一个语法结构(Construct)，例如包、类型、修饰符、运算符、接口、返回值甚至代码注释等

3、填充符号表

- 符号表(Symbol Table)是一组由符号地址和符号信息构成的表格
- 在目标代码生成阶段，当对符号名进行地址分配时，符号表是地址分配的依据

图 10-1Javac 的编译过程

10. 2. 3. 注解处理器

1、JDK 1.5 之后，Java 语言提供了对注解(Annotation)的支持，这些注解与普通的 Java 代码一样，**是在运行期间发挥作用的**

2、在 JDK 1.6 中实现了 JSR-269 规范(Pluggable Annotation Processing API)，提供了一组插入式注解处理器的标准 API 在编译期间对注解进行处理

- 我们可以把它看成是一组编译器的插件，在这些插件里面，可以读取、修改、添加抽象语法树中的任意元素
- 如果这些插件在处理注解期间对语法树进行了修改，编译器将回到解析及填充符号表的过程重新处理，直到所有插入式注解处理器都没有再对语法树进行修改为止，每一次循环称为一个 Round，见图 10-1

3、有了编译器注解处理的标准 API 后，我们的代码才有可能干涉编译器的行为，由于语法树中的任意元素，甚至包括代码注释都可以在插件中访问到，所以通过插入式注解处理器实现的插件在功能上有很大发挥空间

10. 2. 4. 语义分析与字节码生成

1、在语法分析后，编译器获得了程序代码的抽象语法树表示

- **语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的**
- **语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查，如类型审查**

2、在 Javac 的编译过程中，语义分析过程分为**标注检查**以及**数据及控制流分析**两个步骤

3、标注检查

- 标注检查步骤检查的内容
  - 变量使用前是否已被声明
  - 变量与赋值之间的数据类型是否能够匹配
  - 等等
- 常量折叠

- 例如 int a=1+2;经过常量折叠后，它们将会被折叠为字面量 3

4、数据及控制流分析

- **数据及控制流分析是对程序上下文逻辑更进一步的验证**
- 它可以检查诸如程序局部变量在使用前是否有赋值，方法的每条路径是否都有返回值、是否有的受检查异常都被正确处理等问题
- **编译时期**的数据及控制流分析与**类加载时**的数据及控制流分析的目的基本上是一致的，但校验范围有所区别
  - 例如方法参数列表的 final 修饰符，由于局部变量与字段(实例变量、类变量)是由区别的，它在常量池中没有 CONSTANT_Fieldref_info 的符号引用，自然就没有访问标志(Access_Flags)的信息，甚至连名字都不会保留下来，因此在 Class 文件中无法知道一个局部变量是不是声明为 final
  - 因此将局部变量声明为 fianl 对运行期是没有影响的，变量不变性仅仅由编译器在编译期间保障

5、解语法糖

- 语法糖(Syntactic Sugar)，也称糖衣语法，是由英国计算机科学家彼得·约翰·兰达(Peter J.Landin)发明的术语，指在计算机中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用
- 通常来说，使用语法糖能够增加程序的可读性，从而减少程序代码出错的机会
- Java 在现代变成语言之中属于"低糖语言"(相对于 C#及许多其他 JVM 语言来说)，尤其是 JDK 1.5 之前的版本，"低糖"语法也是 Java 语言被怀疑落后的表面理由
- Java 中最常用的语法糖主要是前面提到的泛型(泛型并不一定都是语法糖实现，如 C#的泛型就是直接由 **CLR(Common Language Runtime，公共语言运行库，也 Java 虚拟机一样也是一个运行环境)**支持的)、边长参数、自动装箱/拆箱等，**虚拟机运行时不支持这些语法糖，它们在编译阶段还原回简单的基础语法结构，这个过程称为解语法糖**

6、字节码生成

- **字节码生成是 Javac 编译过程的最后一个阶段**
- 字节码生成阶段不仅仅是把前面各个步骤所生成的信息(语法树、符号表)转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作
- 前面章节多次提到的实例构造器<init>()方法和类构造器<clinit>()方法就是在这个阶段添加到语法树之中的与当前类一致的默认构造函数，这个工作在填充符号表阶段就已经完成，这两个构造器的产生过程实际上是一个代码收敛的过程，编译器会把**语句块**(对于实例构造器而言是"{}";对于类构造器而言是"static{}")、**变量初始化**(实例变量和类变量)、**调用父类的*实例*构造器**(仅仅是实例构造器，<clinit>()中无需调用父类的<clinit>()方法，虚拟机会保证父类构造器的执行)**等操作**收敛到<init>()和<clinit>()方法之中，并且保证一定是按先执行父类的实例构造器，然后初始化变量，最后执行语句块的顺序进行
- **完成了对语法树的遍历和调整之后，生成最终的 Class 文件，到此为止整个编译过程宣告结束**

10.3. Java 语法糖的味道

1、几乎各种语言或多或少都提供过一些语法糖来方便程序员的代码开发

- 这些语法糖虽然不会提供实质性的功能改进，但是它们或许能提高效率，或能提高语法的严谨性，或能减少代码出错的机会
- 但是大量添加和使用含糖语法，容易让程序员产生依赖，无法看清语法糖的糖衣背后，程序代码的真实面目
- 总而言之，语法糖可以看做是编译器实现的一些"小把戏"，这些"小把戏"可能会使得效率"大提升"，但我们也应该去了解这些"小把戏"背后的真实世界，那样才能利用好它们，而不是被它们迷惑

10.3.1. 泛型与类型擦除

1、泛型是 JDK 1.5 的一项新增特性，它的本质是参数化类型(Parametersized Type)的应用，也就是说所有操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口和泛型方法

2、泛型思想早在 C++语言的模板(Template)中就开始生根发芽，在 Java 语言处于还没有出现泛型的版本时，只能通过 **Object 是所有类型的父类**和**类型强制转换**两个特点配合来实现类型泛化

- 但是只有程序员和 Java 虚拟机知道这个 Object 到底是个什么类型的对象
- 在编译期间，编译器无法检查这个 Object 的强制转型是否成功，如果仅仅依赖于程序员去保障这项操作的正确性，许多 ClassCastException 的风险就会转嫁到程序运行期之中

3、Java 的泛型只在源程序中存在，在编译后的字节码文件中就已经替换为原来的原生类型(Raw Type，也称为裸类型)了，**并在相应的地方插入了强制转型代码**，因此泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型称为伪泛型

4、由于 Java 泛型由擦除来实现，因此不能根据参数的泛型类型不同来对方法进行重载

```
public void f(List<Integer>);
public void f(List<String>);
```

- 上述两个方法在编译后，在 Class 文件中的方法签名是完全一致的，因此无法重载

```
public int f(List<Integer>);
public float f(List<String>);
```

- 上述两个方法在编译后，**在 Class 文件中的描述符是不同的，描述符还包括了方法返回值**，因此上述两个方法在 Class 文件中是可以共存的，**但并不意味着与 Java 方法重载(方法重载的要求是具备不同的特征签名)的基本认知矛盾**

5、从 Signature 属性的出现我们可以得出结论：**擦除法所谓的擦除，仅仅是对方法的 Code 属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据**

10.3.2. 自动装箱、拆箱与遍历循环

1、**从纯技术的角度来讲，自动装箱、自动拆箱与遍历循环(Foreach 循环)这些语**

法糖，无论是实现上还是思想上都不能与泛型相比，两者的难度和深度都有很大的差距，但它们是 Java 语言里使用得最多的语法糖

2、自动装箱、自动拆箱在编译之后被转化成了对应的包装和还原的方法，例如 Integer.valueOf()与 Integer.intValue()

10.3.3. 条件编译

1、许多程序设计语言都提供了条件编译的途径，如 C、C++中使用预处理器指示符(#ifdef)来完成条件编译

- C、C++预处理器最初的任务是解决编译时代码依赖关系(如常常用的 #include 预处理命令)
- 在 Java 语言中没有预处理器，因为 Java 语言天然的编译方式(编译并非一个个地编译 Java 文件，而是将所有编译单元的语法树顶级节点输入到待处理列表后再进行编译，因此各个文件之间能够相互提供符号信息)无须使用预处理器

2、Java 也可以进行条件编译，方法就是使用条件为常量的 if 语句，这类 if 语句不同于其他 Java 代码，它在编译阶段就会被运行

3、Java 语言中条件编译的实现，也是 Java 语言的一颗语法糖，根据布尔常量值的真假，编译器将会把分支中不成立的代码块消除掉，这一工作将在编译器解除语法糖阶段完成

- 由于这种条件编译的方式使用了 if 语句，所以它必须遵循最基本的 Java 语法，只能写在方法内部，因此它只能实现语句块基本(Block)级别的条件编译，而没有办法实现根据条件调整整个 Java 类的结构

10.4. 实战：插入式注解处理器

1、由于很少会有针对程序编译的需求，因此 JDK 的编译子系统里面，提供给用户直接控制的功能相对较少，除了虚拟机 JIT 编译的几个相关参数外，我们就只有使用 JSR-296 中定义的插入式注解处理器 API 来对 JDK 编译子系统的行为产生一些影响

2、一套编程语言中的编译子系统的优劣，很大程度上决定了程序运行性能的好坏和编码效率的高低，尤其在 Java 语言中，运行期即使编译与虚拟机执行子系统非常紧密地互相依赖、配合运作

10.4.1. 实战目标

1、编译器在把 Java 程序源码编译为字节码的时候，会对 Java 程序源码做各方面的检查校验，这些校验主要以程序"写得对不对"为出发点，但总体来讲还是较少会校验程序"写的好不好"

- 业界出了许多针对程序"写的好不好"的辅助校验工具，如 CheckStyle、FindBug、Klocwork 等

2、Java 程序命名应当符合下列规范

- 类(或接口)：符合驼式命名法，首字母大写
- 方法：符合驼式命名法，首字母小写
- 字段：
  - 类或实例变量：符合驼式命名法，首字母小写

- 常量：要求全部由大写字母或下划线构成，并且第一个字符不能是下划线

3、实战目标：为 Javac 编译器添加一个额外的功能，在编译程序时检查程序名是否符合上述对类或接口、方法、字段的命名要求

10.4.2. 代码实现

Chapter 11. 晚期(运行期)优化

11.1. 概述

1、在部分商用虚拟机(Sun HotSpot、IBM J9)中，Java 程序最初通过解释器(Interpreter)进行解释执行的，当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为"热点代码"(Hot Spot Code)
2、为了提高热点代码的执行效率，在运行时，**虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各种层次的优化，完成这个任务的编译器称为即时编译器(Just In Time Compiler，简称 JIT 编译器)**
3、即时编译器并不是虚拟机必须的部分，Java 虚拟机规范并没有规定 Java 虚拟机内必须要有即时编译器存在，更没有限定或指导即时编译器应该如何去实现，但是即时编译器性能的好坏，代码优化程度的高低却是衡量一款商用虚拟机优秀与否的最关键指标之一，它也是虚拟机中最核心且最能体现虚拟机技术水平的部分

11.2. HostSpot 虚拟机内的即时编译器

1、解决一下几个问题

- 为何 HotSpot 虚拟机要使用解释器与编译器并存的架构
- 为何 HotSpot 虚拟机要实现两个不同的即时编译器
- 程序何时使用解释器执行，何时使用即时编译器执行
- 哪些程序代码会被编译为本地代码，如何编译本地代码
- 如何从外部观察即时编译器的编译过程与编译结果

11.2.1. 解释器与编译器

1、许多主流的商用虚拟机，如 HotSpot、J9 等，都同时包含解释器与编译器
2、解释器与编译器各有优势

- 当程序需要迅速启动和执行的时候，解释器可以首先发挥作用，省去编译的时间，立即执行
- 在程序运行后，随着时间的推移，编译器逐渐发挥作用，把越来越多的代码编译成本地代码之后，可以获取更高的执行效率
- 当程序运行环境内存资源限制较大(如部分嵌入式系统)可以使用解释器执行节约内存，反之可以使用编译器来提升效率
- 同时，解释器还可以作为编译器激进优化时的一个逃生门
  - 让编译器根据概率选择一些大多数时候都能提升运行速度的手段
  - 当激进优化的假设不成立时，如加载了新类后类型继承结构出现变化，出现"罕见陷阱"时可以通过逆优化退回到解释状态继续执行

3、HotSpot 虚拟机中内置了两个即使编译器，分别称为 Client Compiler 和 Server Compiler 或简称为 C1 编译器和 C2 编译器(也叫 Opto 编译器)

- 目前主流的 HotSpot 虚拟机中，默认采用解释器与其中一个编译器直接配合的方式工作
- 程序使用哪个编译器，取决于虚拟机的运行模式

- HotSpot 虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用"-client"或"-server"参数去强制指定虚拟机运行在 Client 模式或 Server 模式
- 无论采用编译器是 Client Complier 还是 Server Compiler，解释器与编译器搭配使用的方式在虚拟机中称为"混合模式"(Mixed Mode)
  - 用户可以使用参数"-Xint"强制虚拟机运行于"解释模式"
  - 用户可以使用参数"-Xcomp"强制虚拟机运行于"编译模式"

4、代价

- 即时编译器编译本地代码需要占用程序运行时间；
- 编译出优化程度更高的代码，所花费的时间可能更长；
- 而且想要编译出优化程度更高的代码，解释器可能还要替编译器收集性能监控信息

5、为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot 虚拟机还会逐渐启用分层编译的策略

- 第 0 层：程序解释执行，解释器不开启性能监控功能(Profiling)，可触发第 1 层编译
- 第 1 层：也称为 C1 编译，将字节码编译为本地代码，进行简单、可靠的优化，如有必要将加入性能监控的逻辑
- 第 2 层(或 2 层以上)，也称为 C2 编译，也是将字节码编译为本地代码，但是会启用一些编译耗时较长的优化，甚至会根据性能监控信息进行一些不可靠的激进优化
- 实施分层编译后
  - Client Compiler 和 Server Compiler 将会同时同坐，许多代码都可能会被多次编译
  - 用 Client Compiler 获取更高的编译速度
  - 用 Server Compiler 来获取更好的编译质量
  - 在解释执行的时候也无需在承担收集性能监控信息的任务

11. 2. 2. 编译对象与触发条件

1、在运行过程中会被即时编译器编译的"热点代码"有两类

- 被多次调用的方法
- 被多次执行的循环体
- 对于方法调用触发的编译，编译器理所应当会以整个方法作为编译对象
- 对于循环体触发的编译，编译器**依然会以整个方法(而不是单个循环体)**作为编译对象，**这种编译方式因为编译发生在方法执行过程中，因此称为栈上替换(On Stack Replacement)，简称为 OSR 编译**

2、判断一段代码是不是热点代码，是不是需要触发即时编译，这样的行为称为热点探测(Hot Spot Detection)

- 基于采样的热点探测(Sample Based Hot Spot Detection)：采用这种方法的虚拟机会**周期性地检查各个线程的栈顶**，如果发现某个方法经常出现在栈顶，那这个方法就是"热点方法"
  - 优点：实现简单，高效，还可以很容易地获取方法调用关系(将调用堆栈展开即可)
  - 缺点：很难精确地确认一个方法的热度，容易因为受到线程阻塞或者别

的外界因素的影响而扰乱热点探测

- 基于计数器的热点探测(Counter Based Hot Spot Detection)：采用这种方法的虚拟机会为每个方法(甚至是代码块)建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值就认为它是"热点方法"
  - 优点：统计结果相对来说更加精确和严谨
  - 缺点：实现麻烦一些，需要为每个方法建立并维护计数器

3、HotSpot 采用的是第二种---基于计数器的热点探测方法，它为每个方法准备了两类计数器：

- 方法调用计数器(Invocation Counter)
  - **统计方法被调用的次数**
  - 默认阈值在 Client 模式下为 1500，Server 模式下为 10000
  - 可以通过-XX:CompileThreshold 来设定
  - 当一个方法被调用时，会先检查该方法是否存在被 JIT 编译过的版本，如果存在，则优先使用编译后的本地代码来执行，否则计数器加 1，然后判断方法调用计数器和回边计数器值之和是否超过**方法调用计数器的阈值**，如果超过阈值，那么会向即使编译器提交一个该方法的代码编译请求
  - 如果不做任何设置，方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间内方法被调用的次数，当超过一定的时间限度，如果方法调用次数仍然不足以让它提交给即使编译器，那这个方法的调用计数器就会减少一半，这个过程称为**方法调用计数器热度的衰减(Counter Decay)，这段时间就称为此方法统计的半衰周期(Counter Half Life Time)**
  - 进行热度衰减的动作是在虚拟机进行垃圾收集时顺便进行的，可以使用虚拟机参数-XX:-UseCounterDecay 来关闭热度衰减，让方法计数器统计方法调用的绝对次数，这样只要系统运行时间足够长，绝大部分的方法都会被编译成为本地代码
  - 可以使用-XX:CounterHalfLifeTime 参数设置半衰周期的时间，单位秒
- 回边计数器(Back Edge Counter)
  - **统计一个方法中循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为"回边"(Back Edge)**
  - HotSpot 提供了类似于方法调用计数器阈值的参数-XX:BackEdgeThreadshold 供用户设置，但是当前的虚拟机并未使用此参数，因此我们需要设置另外一个参数-XX:OnStackReplacePercentage 来间接调整回边计数器的阈值
    - 虚拟机在 Client 模式下，回边计数器阈值计算公式为
      方法调用计数器阈值 xOSR 比率(OnStackReplacePercentage)/100
      - 其中 OnStackReplacePercentage 默认值为 933，若为默认值，则回边计数器阈值为 13995
    - 虚拟机在 Server 模式下，回边计数器阈值计算公式为
      方法调用计数器阈值 x (OSR 比率-解释器监控比率(InterpreterProfilePercentage))/100
      - 其中 OnStackReplacePercentage 默认值为 140，InterpreterProfilePercentage 默认值为 33，若为默认值，则回边计数

器阈值为 10700

- 当解释器遇到一条回边指令时，会先检查将要执行的代码片段是否已经有编译好的版本，若有，则执行已编译的代码，否则把回边计数器加 1，然后判断方法调用计数器与回边计数器之和是否超过**回边计数器阈值**。当超过阈值时，将会提交一个 **OSR 编译请求**，并且把回边计数器的值降低一些，以便继续在解释器中执行循环，等待编译器输出编译结果
- **与方法计数器不同，回边计数器没有计数热度衰减过程**，**因此这个计数器统计的就是该方法循环执行的绝对次数**，当计数器溢出的时候，它还会把方法计数器的值也调整到溢出状态，这样下次再进入该方法的时候就会执行标准编译过程

➢ 这两个计数器都有一个确定的阈值，当计数器超过阈值，就会触发 JIT 编译

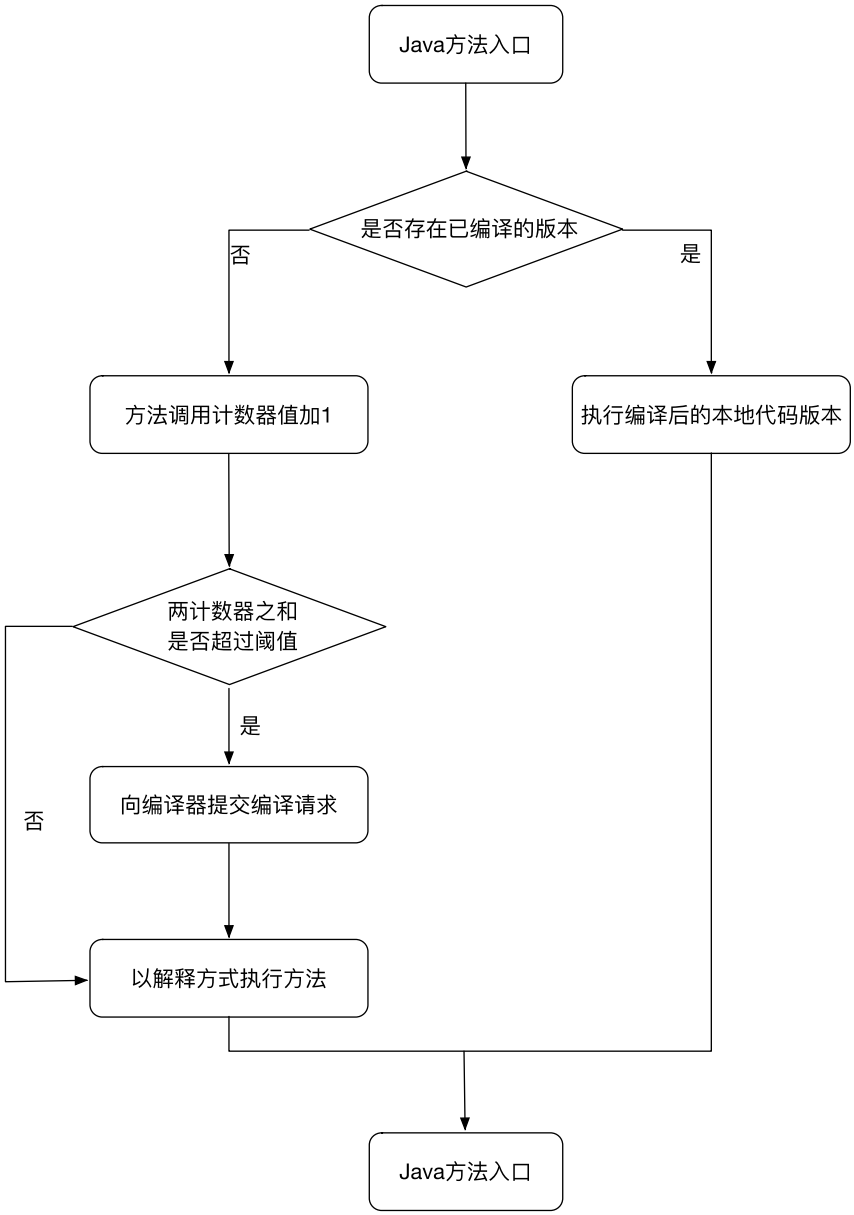

图 11-1 方法调用计数器触发即时编译器

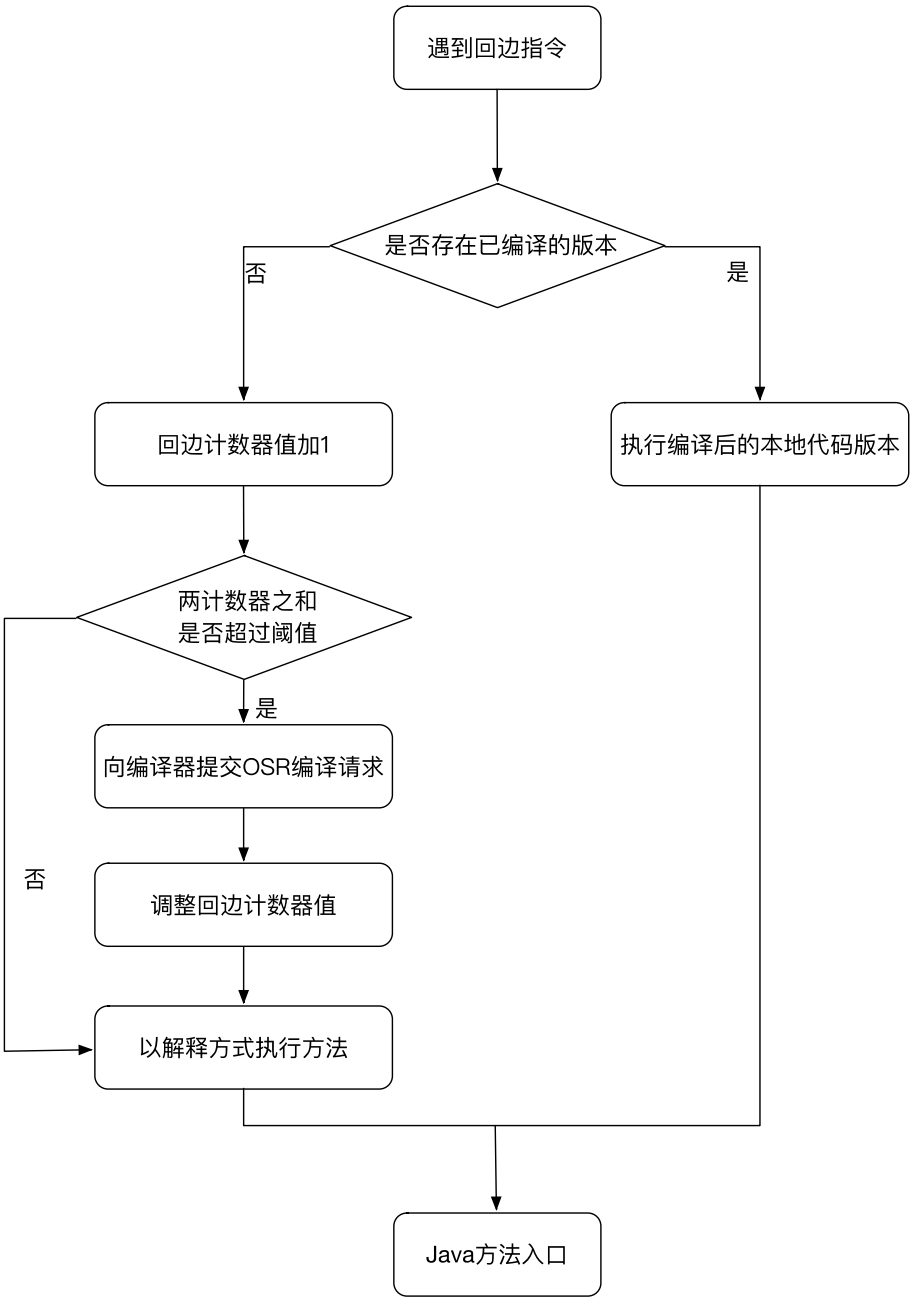

图 11-2 回边计数器触发即时编译

11. 2. 3. 编译过程

1、在默认设置下，无论是方法调用产生的即时编译请求，还是 OSR 编译请求，虚拟机在代码编译尚未完成之前，都仍然将按照解释方式继续执行，**而编译的动作在后台的编译线程中进行**

- 用户可以通过参数-XX:-BackgroudCompilation 来禁止后台编译，一旦达到 JIT 编译条件，执行线程向虚拟机提交编译请求后会一直等待，直到编译过程完成后再开始执行编译器输出的本地代码

2、对于 Client Compiler 来说，它是一个简单快速的三段式编译器，主要的关注点在于局部性的优化，而放弃了许多耗时较长的全局优化手段

- 第一个阶段：一个平台独立的前端字节码构造成一种**高级中间码**表示**(High-Level Intermediate Representation,HIR)**
  - HIR 使用静态单分配(Static Single Assignment,SSA)的形式来代表码值，这可以使得一些在 HIR 的构造过程之中和之后进行的优化动作更容易实现
  - 在此之前编译器会在字节码上完成一部分基础优化，如方法内联，常量传播等优化将会在字节码被构造成 HIR 之前完成
- 第二个阶段：一个平台相关的后端从 HIR 中产生低级中间码表示(Low-Level Intermediate Representation,LIR)，而在此之前会在 HIR 上完成另外一些优化，如空值检查消除、范围检查消除等，以便让 HIR 达到更高效的代码表示形式
- 最后阶段：在平台相关的后端使用线性扫描算法(Linear Scan Register Allocation)在 LIR 上分配寄存器，并在 LIR 上做窥孔(Peephole)优化，然后产生机器代码

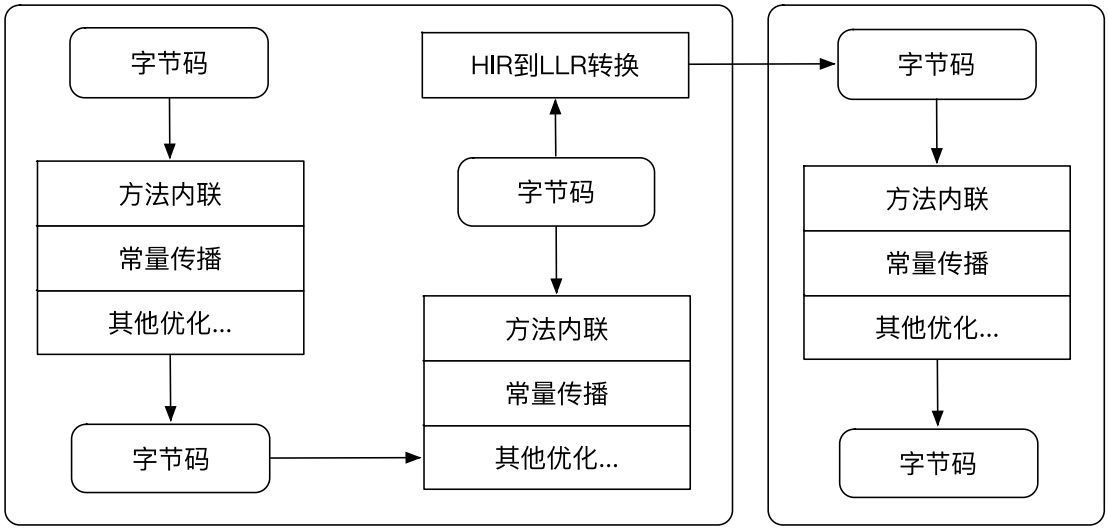

图 11-3 Client Compiler 架构

3、Server Compiler 则是专门面向服务端的典型应用并为服务端的性能配置特别调整过的编译器，也是一个充分优化过的高级编译器，几乎能达到 GUN C++编译器使用-O2 参数时的优化强度，它会执行所有经典的优化动作

- 如无用代码消除(Dead Code Elimination)
- 循环展开(Loop Unrolling)
- 循环表达式外提(Loop Expression Hoisting)
- 消除公共自表达式(Common Subexpression Elimination)
- 常量传播(Constant Propagation)
- 基本块重排序(Basic Blcok Reordering)
- 还会实施一些与 Java 语言特性密切相关的优化技术，如范围检查消除(Range Check Elimination)、空值检查消除(Null Check Elimination)
- 还可能根据解释器或 Client Compiler 提供的性能监控信息，进行一些不稳定的激进优化，如守护内联(Guarded Inlining)、分支频率预测(Branch Frequency Prediction)等

4、Server Compiler 的寄存器分配器是一个全局图着色分配器，它可以充分利用某些处理器架构(RISC)上的大寄存器集合

- 从即时编译的标准来看，Server Compiler 无疑是比较缓慢的
- 但它的编译速度依然远远超过传统的静态优化编译器
- 而且它相对于 Client Compiler 编译器输出的代码质量有所提高，可以减少本地代码的执行时间，从而抵消了编译开销

11.2.4. 查看及分析即时编译结果

1、**一般来说，虚拟机的即时编译过程对于用户程序是完全透明的**，虚拟机通过解释执行代码还是编译执行代码，对于用户来说并没有什么影响(执行结果没有影响，速度上会有很大差别)

2、虚拟机提供了一些参数用于输出即时编译和某些优化手段的执行情况

3、除了查看即时编译器生成的机器码内容，不过虚拟机输出的 01 对于我们的阅读来说是没有意义的，必须反汇编成基本的汇编语言才可能被阅读

- 虚拟机提供了一组通用的反汇编接口

11.3. 编译优化技术

1、Java 程序员有一个共识，以编译方式执行本地代码比解释器执行更快

- 之所以有这样的共识，除去虚拟机解释执行字节码时额外消耗时间的原因外还有一个很重要的原因就是**虚拟机设计团队几乎把对代码的所有优化措施都集中在了即时编译器之中**(在 JDK1.3 之后，不会生成任何字节码级别的优化代码了)
- 因此一般来说，即时编译器产生的本地代码会比 Javac 产生的字节码更加优秀

2、即时编译器优化技术

- 编译器策略(compiler tactics)
  - 延迟编译(delayed compilation)
  - 分层编译(tiered compilation)
  - 栈上替换(on-stack replacement)
  - 延迟优化(delayed reoptimization)
  - 程序依赖图表示(program dependence graph representation)
  - 静态单赋值表示(static single assignment representation)
- 基于性能监控的优化技术(profile-based techniques)
  - 乐观空值断言(optimistic nullness assertions)
  - 乐观类型断言(optimistic type assertions)
  - 乐观类型增强(optimistic type strengthening)
  - 乐观数组长度增强(optimistic array length strengthening)
  - 裁剪未被选择的分支(untaken branch pruning)
  - 乐观的多态内联(optimistic N-morphic inlining)
  - 分支频率预测(branch frequency prediction)
  - 调用频率预测(call frequency prediction)
- 基于证据的优化技术(proof-based techniques)
  - 精确类型推断(exact type inference)

- 内存值推断(memory value inference)
- 内存值跟踪(memory value tracking)
- 常量折叠(constant folding)
- 重组(reassociation)
- 操作符退化(operator strength reduction)
- 空值检查消除(null check elimination)
- 类型检查退化(type test strength reduction)
- 类型检测消除(type test elimination)
- 代数化简(algebraic simplification)
- 公共子表达式消除(common subexpression elimination)

➢ 数据流敏感重写(flow-sensitive rewrites)
- 条件常量传播(conditional constant propagation)
- 基于流承载的类型缩减转换(flow-carried type narrowing)
- 无用代码消除(dead code elimination)

➢ 语言相关的优化技术(language-specific techniques)
- 类型继承关系分析(class hierarchy analysis)
- 去虚拟化(devirtualization)
- 符号常量传播(symbolic constant propagation)
- 自动装箱消除(autobox elimination)
- 逃逸分析(escape analysis)
- 锁消除(lock elision)
- 锁膨胀(lock coarsening)
- 消除反射(de-reflection)

➢ 内存及代码位置变换(memory and placement transformation)
- 表达式提升(expression hoisting)
- 表达式下沉(expression sinking)
- 冗余存储消除(redundant store elimination)
- 相邻存储合并(adjacent store fusion)
- 交汇点分离(merge-point splitting)

➢ 循环变换(loop transformation)
- 循环展开(loop unrolling)
- 循环剥离(loop peeling)
- 安全点消除(safepoint elimination)
- 迭代范围分离(iteration range splitting)
- 范围检查消除(range check elimination)
- 循环向量化(loop vectorzation)

➢ 全局代码调整(global code shaping)
- 内联(inlining)
- 全局代码外提(global code motion)
- 基于热度的代码布局(heat-based code layout)
- Switch 调整(switch balancing)

➢ 控制流图变换(control flow graph transformation)
- 本地代码编排(local code scheduling)
- 本地代码封包(local code bundling)

- 延迟槽填充(delay slot filling)
- 着色图寄存器分配(graph-coloring register allocation)
- 线性扫描寄存器分配(linear scan register allocation)
- 复写聚合(copy coalescing)
- 常量分裂(constant splitting)
- 复写移除(copy removal)
- 地址模式匹配(address mode matching)
- 指令窥孔优化(instruction peepholing)
- 基于确定有限状态机的代码生成(DFA-based code generator)

3、**方法内联的重要性要高于其他优化措施**

- 去除方法调用的成本(建立栈帧等)
- 为其他优化建立良好的基础，方法内联膨胀之后可以便于在更大范围上采取后续的优化手段，从而获得更好的优化效果

4、几项最有代表性的优化技术

- 语言无关的经典优化技术之一：公共子表达式消除
- 语言相关的经典优化技术之一：数组范围检查消除
- 最重要的优化技术之一：方法内联
- 最前沿的优化技术之一：逃逸分析

11.3.1. 公共子表达式消除

1、如果一个表达式 E 已经计算过了，并且从先前的计算到现在 E 中所有变量的值都没有发生变化，那么 E 的这次出现就成为了公共子表达式

- 对于这种表达式，没有必要话费时间再对它进行计算，只需要直接用前面计算过的表达式结果代替 E 就可以了
- 如果这种优化仅限于程序的基本块内，便成为局部公共子表达式消除(Local Common Subexpression Elimination)
- 如果这种优化的范围涵盖了多个基本的块，就称为全局公共子表达式消除(Global Common Subexpression Elimination)

11.3.2. 数组边界检查消除

1、数组边界检查消除(Array Bounds Checking Elimination)是即时编译器中一项语言相关的经典优化技术

2、Java 语言是一门动态安全的语言，对数组的读写访问也不像 C 和 C++那样在本质上是裸指针操作

- Java 语言中访问数组元素时系统将会自动进行上下界的范围检查
- 这对软件开发者来说是一件很好的事
- 对于虚拟机执行子系统来说，每次数组元素的读写都带有一次隐含的条件判定操作，对于拥有大量数组访问的程序代码，无疑是性能负担

3、无论如何，数组边界检查肯定是必须的，但是不是必须在运行期间一次不漏地检查则是可以商量的事情

4、安全检查成为一种隐式的开销，如何处理不好，可能就成为 Java 语言比 C/C++更慢的因素

- **尽可能把运行期检查提到编译期完成**
  - 例如一个循环中有对数组的访问(数组的访问就是利用该循环的循环变

量)，可以根据循环的边界条件来判断数组访问是否可能发生越界

- **隐式异常处理**，由于一次异常处理的代价要高于一次判断，当异常极少出现时，隐式异常优化是值得的，**HotSpot 足够聪明，它会根据运行期收集到的 Profile 信息自动选择最优方案**

11. 3. 3. 方法内联

1、方法内敛是编译器最重要的优化手段

- 消除方法调用成本
- 为其他优化手段建立良好的基础

2、**方法内联不过是把目标方法的代码"复制到"发起调用的方法之中，避免发生真实的方法调用而已**

3、实际上，Java 虚拟机中内联过程远远没有那么简单，因为如果不是即时编译器做了一些特别的努力，按照经典编译原理的优化理论，大多数 Java 方法都无法进行内联

- **只有使用 invokespecial 指令调用私有方法、实例构造器、父类方法以及使用 invokestatic 指令调用静态方法才是在编译期进行解析的**，除这 4 种方法，其他方法都需要在运行时进行方法接受者的多态选择
- 对于一个虚方法，编译期做内联的时候根本无法确定应该使用哪个方法版本

4、Java 对象默认就是虚方法，因此 Java 间接鼓励了程序员使用大量的虚方法来完成程序逻辑

- 如果内联与虚方法之间产生矛盾，是不是为了提高执行性能，就要导出使用 final 关键字去修饰方法呢?(这就是早期 final 能提高效率的原因吗)

5、**类型继承关系分析(Class Hierarchy Analysis，CHA)技术**

- 基于整个应用程序的类型分析技术
- 用于确定在目前已加载的类中，某个接口是否有多于一种的实现，某个类是否存在子类，子类是否为抽象类等信息

6、编译器在进行内联时

- 如果是非虚方法，那么直接进行内联就可以了，这时候内联是由稳定前提保障的
- 如果是虚方法，则会向 CHA 查询此方法在当前程序下是否有多个目标版本可供选择，**如果查询结果只有一个版本，那也可以进行内联，不过这种内联就属于激进优化**，需要预留一个"逃生门"(Guard 条件不成立时的 Slow Path)，称为**守护内联(Guarded Inlining)**
  - 如果在后续的执行过程中，虚拟机一直没有加载到会令这个方法的接受者的继承关系发生变化的类，那么这个内联优化的代码就可以一直使用下去
  - 如果加载了导致继承关系发生变化的新类，那就需要抛弃已经编译的代码，退回到解释状态执行，或者重新进行编译
- 如果是虚方法，并且 CHA 查询结果有多个版本的目标方法可供选择，编译器还会进行最后一次努力，使用内联缓存(Inline Cache)来完成方法内联，其工作原理大致是
  - 在未发生方法调用之前，内联缓存状态为空，**当第一次调用发生后，缓存记录下方法接受者的版本信息**，并且每次进行方法调用时都比较接受

者版本，如果以后进来的每次调用方法的接受者版本都是一样的，那这个内联还可以一直用下去，如果发生了方法接受者不一致的情况，说明程序真正使用了虚方法的多态特性，这时才会取消内联，查找虚方法表进行方法分派

11. 3. 4. 逃逸分析

1、逃逸分析是目前 Java 虚拟机中比较前沿的技术，**它与类型继承关系分析(CHA)一样，并不是直接优化代码的手段，而是为其他优化手段提供依据的分析技术**

2、逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用例如作为调用参数传递到其他方法中，称为**方法逃逸**，甚至还有可能被外部线程访问到，譬如赋值给类变量或可以在其他线程中访问的实例变量，称为**线程逃逸**

3、如果能证明一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问这个对象，则可能为这个变量进行一些高效的优化

4、**栈上分配(Stack Allocation)**：

- Java 虚拟机中，在 Java 堆上分配创建对象的内存空间几乎是 Java 程序员都清楚的常识了
- Java 堆中的对象对于各个线程都是共享可见的，只要持有这个对象的引用，就可以访问堆中存储的对象数据
- 虚拟机的垃圾收集系统可以回收堆中不再使用的对象，但回收动作无论是筛选可回收对象，还是回收和整理内存都需要耗费时间
- **如果确定一个对象不会逃出方法之外，那让这个对象在*栈上分配内存*将会是一个很不错的主意，对象所占的内存空间就可以随栈帧出栈而销毁**
- 在一般应用中，不会逃逸的局部对象所占的比例很大，如果能使用栈上分配，那大量的对象就会随着方法的结束而自动销毁了，垃圾收集系统的压力将会小很多

5、**同步消除(Synchronization Elimination)**：

- 线程同步本身是一个相对耗时的过程，如果逃逸分析能够确定一个变量不会逃逸出线程，无法被其他线程访问，那这个变量的读写肯定不会有竞争，对这个变量实施的同步措施也就可以消除

6、**标量替换(Scalar Replacement)**：

- 标量(Scalar)是指一个数据已经无法再分解成更小的数据来表示了
- Java 虚拟机中的原始数据类型(int、long 等数值类型以及 reference 类型等)都不能再进一步分解，它们就可以称为标量
- 相对的，如果一个数据可以继续分解，那它就称为聚合量(Aggregate)，Java 中的对象就是最典型的聚合量
- 如果把一个 Java 对象拆散，根据程序访问情况，将其使用到的成员变量恢复原始类型来访问就叫标量替换
- **如果逃逸分析证明一个对象不会被外部访问，并且这个对象可以被拆散的话，那程序真正执行的时候将可能不创建这个对象，而改为直接创建它的若干个被这个方法使用到的成员变量来代替**
- 将对象拆分后，除了可以让对象的成员变量在栈上分配和读写之外，还可以为后续进一步的优化手段创建条件

7、逃逸分析的论文在 1999 年就已经发表，但是到 JDK 1.6 才实现了逃逸分析

- 不能保证逃逸分析的性能受益必定高于它的消耗
- 要完全准确地判断一个对象是否会逃逸，需要进行数据流敏感的一系列复杂分析，从而确定程序各个分支执行时对此对象的影响，这是一个相对高耗时的过程，如果分析完发现没有几个对象不逃逸，那么运行时耗用的时间就白白浪费了
- 目前虚拟机只能采用不那么准确，但时间压力相对较小的算法来完成逃逸分析
- 基于逃逸分析的优化手段，例如栈上分配，由于 HotSpot 虚拟机目前的实现方式导致栈上分配实现起来比较复杂，暂时不支持这项优化

8、目前逃逸分析默认不开启

- 可以使用参数-XX:+DoEscapeAnalysis 来手动启动逃逸分析
- 还可以通过参数-XX:+PrintEscapeAnalysis 来查看分析结果
- **开启逃逸分析后**
- -XX:+EliminateAllocations 开启标量替换
- -XX:+EliminateLocks 开启同步消除
- -XX:+PrintEliminateAllocation 查看标量替换情况

11.4. Java 与 C/C++编译器对比

1、Java 与 C/C++的编译器对比实际上代表了最典型的即时编译器与静态编译器的对比，很大程度上决定了 Java 与 C/C++的性能对比结果

- **无论是 C/C++还是 Java 代码，最终编译之后被机器执行的都是本地机器码(Java 字节码是???)**，哪种语言的性能更高，除了它们自身的 API 库实现得好坏之外，其余的比较就成了一场"拼编译器"和"拼输出代码质量"的游戏

2、Java 虚拟机的即时编译器与 C/C++的静态优化编译器相比，可能会由于这些原因而导致输出的本地代码有一些劣势

- **即时编译器运行占用的是用户程序的运行时间，具有很大的时间压力，它能提供的优化手段也严重受制于编译成本**
  - 如果编译速度不能达到要求，那用户将在启动程序或程序的某部分察觉到重大延迟
  - 这使得即时编译器不敢随便引入大规模的优化技术，而编译的时间成本在静态优化编译器中并不是主要的关注点
- **Java 语言是动态的类型安全语言**
  - 这就意味着需要由虚拟机来确保程序不会违反语言语义或访问非结构化内存
  - 这就意味着虚拟机必须频繁地进行动态检查，例如实例方法访问时检查空指针、数组元素访问时上下界范围的检查、类型转换时检查继承关系等
  - 对于这类程序代码没有明确写出的检查行为，尽管编译器会努力进行优化，但总体上还是要消耗不少的运行时间
- **Java 语言虽然没有 virtual 关键字，但是使用虚方法的频率远远大于 C/C++语言**
  - 这意味着运行时对方法接受者进行多态选择的频率要远远大于 C/C++语

言

- 也意味着即时编译器在进行一些优化(例如方法内联)时的难度要远大于 C/C++的静态优化编译器

➢ **Java 语言是可以动态扩展的语言**

- 运行时加载新的类可能改变程序类型的继承关系，这使得很多全局的优化都难以进行
- **因为编译器无法看清程序的全貌，许多全局的优化措施只能以激进的方式来完成**，编译器不得不时刻注意着类型的变化而在运行时撤销或重新进行一些优化

➢ **Java 语言中对象的内存分配都是在堆上进行的，只有方法中的局部变量才能在栈上分配**

- C/C++的对象则有多种内存分配方式，既可能在堆上分配，又可能在栈上分配，如果可以在栈上分配线程私有的对象，则将减轻内存回收的压力
- C/C++主要由用户程序代码来回收分配的内存，这就不存在无用对象筛选过程，因此效率上也比垃圾收集机制要高

3、Java 语言的这些性能上的劣势都是为了换取开发效率上的优势而付出的代价，动态安全、动态扩展、垃圾回收这些特性都是为 Java 的开发效率做了很大的贡献

4、另外，还有许多优化时 Java 即时编译器能做而 C/C++静态优化编译器不能做或者不好做的

➢ 例如 C/C++中，别名分析(Alias Analysis)的难度就要远高于 Java

5、Java 编译器的另外一个红利是动态性所带来的，由于 C/C++编译器所有优化都在编译期完成，以运行期性能监控为基础的优化措施它都无法进行，如调用频率预测(Call Frequency Prediction)、分支频率预测(Branch Frequency Prediction)、裁剪未被选择的分支(Untaken Branch Pruning)等，都会成为 Java 语言独有的性能优势

Chapter 12. Java 内存模型与线程

1、并发处理的广泛应用使得 Amdahl 定律代替摩尔定律称为计算机性能发展原动力的根本原因，也是人类压榨计算机运算能力的最有力武器

12.1. 概述

1、在许多情况下，让计算机同时去做几件事情，不仅是因为计算机的运算能力强大了，还有一个重要的原因是计算机的运算速度与它的存储和通信子系统速度差距太大，大量的时间都花费在磁盘 I/O、网络通信或者数据库访问上
2、除了充分利用计算机处理器的能力外，一个服务端同时对多个客户端提供服务则是另一个更具体的并发应用场景

- 衡量一个服务性能的好坏，每秒事务处理数(Transactions Per Second，TPS)是最重要的指标之一，它代表着一秒内服务端平均能响应的请求总数
- TPS 值与程序的并发能力又有非常密切的关系

3、服务端是 Java 语言最擅长的领域之一

- 如何写好并发应用程序却又是服务端程序开发的难点之一，处理好并发方面的问题通常需要更多的编码经验来支持
- Java 语言和虚拟机提供了许多工具，把并发编程的门槛降低了不少，使得程序员编码时更关注业务逻辑，而不是花费大部分时间去关注此服务同时会被多少人调用、如何协调硬件资源等
- **无论语言、中间件和框架如何先进，开发人员都不能期望它们能独立完成所有并发处理的事情，了解并发内幕也是成为一个高级程序员不可缺少的课程**

12.2. 硬件的效率与一致性

1、"让计算机并发执行若干个运算任务"与"更充分地利用计算机处理器的效能"之间的关系并没有想象中那么简单

- 其中一个重要的复杂性来源是绝大多数运算任务都不可能只靠处理器"计算"就能完成，处理器至少要与内存交互，如读取运算数据、存储运算结果等，这个 I/O 操作是很难消除的
- 由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的**高速缓存(Cache)**来作为内存与处理器之间的缓冲：**将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存当中**。这样处理器就无需等待慢慢的内存读写了

2、基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也为计算机系统带来更高的复杂度，因为它引入了一个新问题：**缓存一致性(Cache Coherence)**

- 在多处理器系统中，每个处理器都有自己的高速缓存，它们又共享同一主内存
- 当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致，如果真的发生这种情况，那同步回到主内存时以谁的缓

存数据为准呢?

- 为了解决一致性的问题，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议来进行操作，这类协议有 MSI、MESI、MOSI、Synapse、Firefly 及 Dragon Protocol 等

3、内存模型：

- 可以理解为在特定的操作协议下，对特定的内存或高速缓存进行读写访问的抽象过程
- 不同架构的物理机器可以拥有不一样的内存模型
- Java 虚拟机也有自己的内存模型

4、除了增加高速缓存之外，为了使得处理器内部的运算单元能尽量被充分利用，处理器可能会对输入代码进行**乱序执行(Out-Of-Order Execution)优化**

- 处理器会在计算之后将乱序执行的结果重组，保证该结果与顺序执行的结果是一致的，但并不保证程序中各个语句计算的先后顺序与输入代码中的顺序一致
- 因此，如果存在一个计算任务依赖另一个计算任务的中间结果，那么其顺序性并不能靠代码的先后顺序来保证
- 与处理器的乱序执行优化类似，Java 虚拟机的即时编译器中也有类似的指令重排序(Instruction Reorder)优化

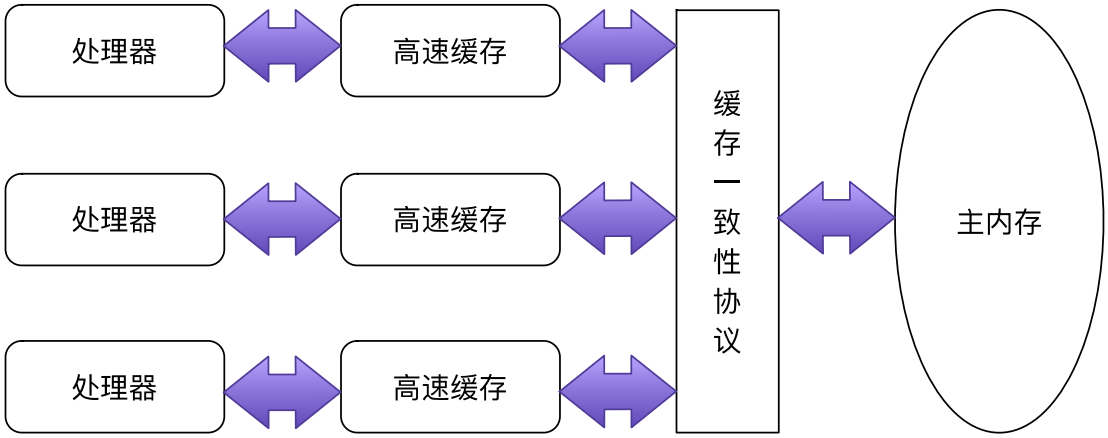

图 12-1 处理器、高速缓存、主内存间的交互关系

12.3. Java 内存模型

1、Java 虚拟机规范中试图定义一种 Java 内存模型(Java Memory Model,JVM)来屏蔽掉各种硬件和操作系统内存访问差异，以实现让 Java 程序在各种平台下都能达到一致性的内存访问效果

2、主流程序语言如 C/C++直接使用武力硬件和操作系统的内存模型

- 因此，会由于不同平台上内存模型的差异，有可能导致程序在一套平台上并发完全正常，而在另外一套平台上并发访问却经常出错
- 因此，在某些场景就必须针对不同平台来编写程序

3、定义 Java 内存模型并非一件容易的事情

- 这个模型必须定义得足够严谨，才能让 Java 的并发访问操作不会产生歧义

- 同时也必须定义得足够宽松，使得虚拟机的实现有足够的自由空间去利用硬件的各种特性(寄存器、高速缓存和指令集中某些特有的指令)来获取更好的执行速度

12. 3. 1. 主内存与工作内存

1、Java 内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将<注意>变量存储到内存和从内存中取出变量这样底层的细节

- 此处的变量与 Java 编程中的变量有所区别，此处的变量包括了实例字段、静态字段、和构成数组对象的元素，但不包括局部变量与方法参数，因为后者是线程私有的，不会被共享，自然就不会存在竞争的问题

2、为了获取较好的执行效能，Java 内存模型并没有限制执行引擎使用处理器的特定寄存器或缓存来和主内存进行交互，也没有限制即时编译器进行调整代码执行顺序这类优化措施

3、Java 内存模型中规定了所有的变量都存储在主内存(Main Memory)中(此处的主内存与物理硬件的主内存名字一样，两者也可以相互类比，但此处仅是虚拟机内存的一部分(Java 堆))

4、每条线程还有自己的工作内存(Working Memory，可与处理器的高速缓存类比)

- 线程的工作内存中保存了该线程使用到的变量的主内存副本拷贝，线程对变量所有操作(读取、赋值)都必须在工作内存中进行，而不能直接读写主内存中的变量
- 不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成

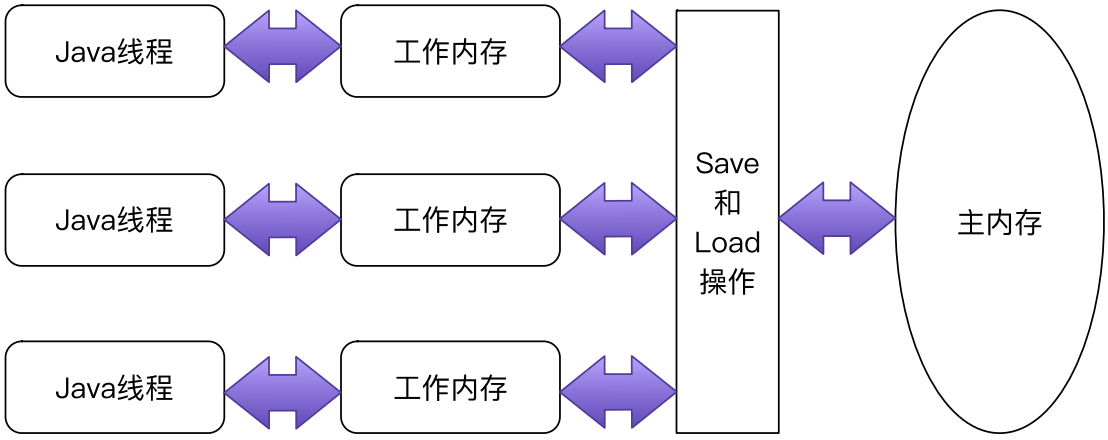

图 12-2 线程、主内存、工作内存三者的交互关系

5、<注意>

- 这里讲的主内存、工作内存与第二章所讲的 Java 内存区域中的 Java 堆、栈、方法区并不是同一个层次的内存划分，这两者基本是没有关系的
- 硬要勉强对应的话
  - 主内存主要对应于 Java 堆中的对象实例数据部分
  - 工作内存主要对应于虚拟机栈中的部分区域
- 从更低层次来说，主内存就直接对应于物理硬件的内存，而为了获取更好的运行速度，虚拟机(甚至是硬件系统本身的优化措施)可能会让工作内存优先存储于寄存器和高速缓存中，因为程序运行时主要访问读写的是工作内存

12. 3. 2. 内存建交互操作

1、关于主内存与工作内存之间的交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步回主内存之类的实现细节，Java 内存模型中定义了以下 8 种操作来完成，**虚拟机必须保证下面提及的每一种操作都是原子的、不可再分的(对 double 和 long 类型的变量来说，load、store、read、和 write 操作在某些平台上允许有例外)**

- lock(锁定)：用于主内存的变量，它把一个变量标识为一条线程独占的状态
- unlock(解锁)：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定
- read(读取)：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的 load 动作使用
- load(载入)：作用于工作内存的变量，它把 read 操作从主内存中得到的变量值放入工作内存的变量副本中
- use(使用)：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的值的字节码指令时将会执行这个操作
- assign(赋值)：作用于工作内存的变量，它把一个从执行引擎接收到的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作
- store(存储)：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的 write 操作使用
- write(写入)：作用于主内存中的变量，它把 store 操作从内存中得到的变量的值放入主内存的变量中

2、如果要把一个变量从主内存复制到工作内存，那就要顺序地执行 read 和 load 操作，如果要把变量从工作内存同步回主内存，就要顺序地执行 store 和 write 操作

- Java 内存模型只要求上述两个操作必须顺序执行，但没有保证是连续执行

3、Java 内存模型还规定了在执行上述 8 种基本操作时必须满足如下规则

- 不允许 read 和 load、store 和 write 操作之一单独出现，即不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起写回了但主内存不接受的情况
- 不允许一个线程丢弃它的最近的 assing 操作，即变量在工作内存中改变了之后必须把该变化同步回主内存
- 不允许一个线程无原因地(没有发生任何 assign 操作)把数据从线程的工作内存同步回主内存中
- 一个新的变量只能在主内存中"诞生"，不允许在工作内存中直接使用一个未被初始化(load 或 assign)的变量，换句话说，就是对一个变量实施 use、store 之前必须先执行过 assign 和 load 操作
- 一个变量在同一时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁

- 如果对一个变量执行 lock 操作，那将会清空工作内存中此变量的值，在执行引擎使用这个变量之前，需要重新执行 load 或 assign 操作初始化变量的值
- 如果一个变量事先没有被 lock 操作锁定，不允许对它执行 unlock 操作，也不允许去 unlock 一个被其他线程锁住的变量
- 对一个变量执行 unlock 之前，必须把变量同步回主内存中(执行 store、write 操作)

12.3.3. 对于 volatile 类型变量的特殊规则

1、关键字 volatile 可以说是 Java 虚拟机提供的最轻量级的同步机制，但它并不容易被正确、完整地理解

2、当一个变量定义为 volatile 之后，它将具备两种特性

- 一是保证此变量对所有线程的**可见性**：可见性指当一条线程修改了这个变量的值，新值对于其他线程来说是立即得知的
  - **普通变量做不到这一点，普通变量的值在线程间传递均需要通过主内存来完成**
  - volatile 的可见性经常会被开发人员误解，认为以下描述成立"volatile 变量对所有线程是立即可见的，对 volatile 变量所有的写操作都能立刻反应到其他线程之中，~~换句话说，volatile 变量在各个线程中是一致的，所以基于 volatile 变量的运算在并发下是安全的~~"
    - 这句话论据部分没有错，但其论据不能得出"~~基于 volatile 变量的运算在并发下是安全的~~"这个结论
    - volatile 变量在各个线程的工作内存中不存在一致性的问题
    - 但是 Java 里面运算并非原子性操作，导致 volatile 变量的运算在并发下一样是不安全的
    - 例如 race++运算符(假设 race 是 volatile 的)

      public static void increase(){race++;}可以分解为以下四条字节码

      ```
      getstatic
      iconst_1
      iadd
      putstatic
      return
      ```

      - getstatic 保证把 race 的值取到操作栈顶时，volatile 关键字保证了 race 的值在此时是正确的，但是在执行 iconst_1、iadd 时，其他线程可能已经把 race 的值增大了，而在操作栈顶的值就变成了过期的数据，所以 putstatic 指令执行后就可能把较小的 race 值同步回主内存之中
    - **即便只有一条字节码指令，也并不意味着这条指令就是一个原子操作，因为一条字节码指令在解释执行时，解释器将要运行许多代码才能实现它的语义，如果是编译执行，一条字节码指令也可能转化成若干条本地机器码指令**
  - 由于 volatile 变量只能保证可见性，在不符合以下两条规则的运算场景中，我们仍然要通过加锁(使用 synchronized 或 java.util.concurrent 中的原子类)来保证原子性

        - 运算结果并不依赖变量当前值，或者能够确保的只有单一的线程修改变量的值
        - 变量不需要与其他状态变量共同参与不变约束
    - volatile 第二个语义是禁止指令重排序优化
        - 普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中的执行顺序一致

3、volatile 选用建议

- 在某些情况下，volatile 的同步机制的性能确实要优于锁，但是由于虚拟机对锁实行的许多消除和优化，使得我们很难量化地认为 volatile 就会比 synchronized 快多少
- volatile 变量读操作的性能消耗与普通变量几乎没有什么差别，但是写操作则可能会慢一点，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行
- 不过即便如此，大多数场景下，volatile 的总开销仍然要比锁低
- 我们在 volatile 与锁之中选择的唯一依据仅仅是 volatile 的语义能否满足使用场景的需求

4、Java 内存模型中对 volatile 变量定义的特殊规则

- <未完成>

12.3.4. 对于 long 和 double 型变量的特殊规则

1、Java 内存模型要求 lock、unlock、read、load、assign、use、store、write 这 8 个操作具有原子性，但是对于 64 位的数据类型(long 和 double)，在模型中特别定义了一条相对宽松的规定：

- 允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作划分为两次 32 位的操作来进行
- 即允许虚拟机实现选择可以不保证 64 位数据类型 load、store、read 和 write 这 4 个操作的原子性
- 这就是 long 和 double 的非原子性协议

2、如果多个线程共享一个并未声明为 volatile 的 long 或 double 类型的变量，并且同时对它们进行读取和修改操作，那么某些线程可能会读取到一个既非原值，也不是其他线程修改值的代表了"半个变量"的数值

3、这种读取到半个变量的情况非常罕见(在商用 Java 虚拟机不会出现)

- Java 内存模型虽然允许虚拟机不把 long 和 double 变量的读写实现成原子操作，但允许虚拟机选择把这些操作实现为具有原子性的操作，而且还"强烈建议"虚拟机这样实现
- 实际开发中，各种平台下的商用虚拟机几乎都选择把 64 位数据的读写操作作为原子操作来对待，因此编写现代代码时一般不需要把用到的 long 和 double 变量专门声明为 volatile

12.3.5. 原子性、可见性与有序性

1、Java 内存模型是围绕着在并发过程中如何处理原子性、可见性与有序性这 3 个特征来建立的

2、**原子性(Atomicity)**

- 由 Java 内存模型来直接保证的原子性变量操作包括 read、load、assign、use、store、write
- 我们大致可以认为基本数据类型的访问读写是具备原子性的(例外就是 long 和 double 的非原子性协定(但 HotSpot 实现过程却是保证其原子性的)，但是无序太过在意这些几乎不会发生的例外情况)
- 如果应用场景需要一个更大范围的原子性保证，Java 内存模型提供了 lock 和 unlock 操作来满足这种需求
  - 虚拟机未把 lock 和 unlock 操作直接开放给用户使用，但是却提供了更高层次的字节码指令 monitorenter 和 monitorexit 来隐式地使用这两个操作
  - **这两个字节码反映到 Java 代码中就是同步块 synchronized 关键字，因此在 synchronized 块之间的操作也具备原子性**

3、可见性(Visibility)：

- 可见性是指一个线程修改了共享变量的值，其他线程能够立即得知这个修改值
- Java 内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值**这种依赖主内存作为传递媒介的方式**来实现可见性的，无论是普通变量还是 volatile 变量都是如此
- 普通变量与 volatile 变量的区别是：**volatile 的特殊规则保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新**
- 因此可以说，volatile 保证了多线程操作时变量的可见性，而普通变量则不能保证这一点
- **至于从主内存读取到将新值同步回主内存中间这段时间，其余线程仍然可以更改该变量在主内存的值，导致当前线程将该变量同步回主内存时，该变量可能已经被别的线程修改过了，而并非其之前读取的值**
- 除了 volatile 之外，Java 还有两个关键字能实现可见性，即 synchronized 和 final
- 同步块的可见性是由："对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前，需要重新执行 load 和 assign 操作初始化变量的值"和"对一个变量执行 unlock 操作之前，必须先把此变量同步回主内存中(执行 store、write)操作"这两条规则获得的
- final 的可见性是指：被 final 修饰的字段在构造器中一旦初始化完成，并且构造器没有把 this 的引用传递出去(this 引用逃逸是一件很文献的事情，其他线程有可能通过这个引用访问到"初始化了一半"的对象)，那在其他线程中就能看见 final 字段的值

4、有序性(Ordering)：

- Java 程序中天然的有序性可以总结为一句话：如果在本线程内观察，所有的操作都是有序的；如果在一个线程中观察另一个线程，所有的操作都是无序的
- 前半句是指：线程内表现为串行的语义
- 后半句是指：指令重排序现象和工作内存与主内存同步延迟现象
- Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性
  - volatile 关键字本身包含了禁止重排序的语义

- synchronized 则是由"一个变量在同一时刻只允许一条线程对其进行 lock 操作"这条规则获得的

➢

5、synchronized 关键字可以同时保证原子性、可见性与有序性

➢ 大量使用 synchronized 可能会导致性能的降低

12.3.6. 先行发生原则

1、如果 Java 内存模型中所有的有序性都仅仅靠 volatile 和 synchronized 来完成，那么有一些操作将会很繁琐，但是编写 Java 并发代码时并没有感觉到这一点

2、Java 语言有一个"先行发生(happens-before)的原则"

➢ **这个原则是判断数据是否存在竞争、线程是否安全的主要依据**

➢ 依靠这个原则，我们可以通过几条规则一揽子地解决并发环境下两个操作之间是否可能存在冲突的所有问题

3、Java 内存模型下一些"天然的"线性发生关系

➢ **程序次序规则(Program Order Rule)**：在**一个线程内**，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作，准确的说，应该是控制流顺序而不是程序代码顺序

➢ **管程锁定规则(Monitor Lock Rule)**：一个 unlock 操作线性发生于后面对**同一个锁**的 lock 操作

➢ **volatile 变量规则(Volatile Variable Rule)**：对一个 volatile 变量的写操作先行发生于后面这个变量的读操作，后面指时间上的先后顺序

➢ **线程启动规则(Thread Start Rule)**：Thread 对象的 start()方法先行发生于此线程的每一个动作

➢ **线程终止规则(Thread Termination Rule)**：线程中所有操作都先行发生于对此线程的终止检测，我们可以通过 Thread.join()方法结束，Thread.isAlive()的返回值等手段检测到线程已经终止执行

➢ **线程中断规则(Thread Interruption Rule)**：对线程 interrupt()方法调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 Thread.interrupted()方法检测到是否有中断发生

➢ **对象终结规则(Finalizer Rule)**：一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize()方法的开始

➢ **传递性(Transitivity)**：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C 的结论

4、时间先后与先行发生原则之间基本没有太大关系，所以我们衡量并发安全问题的时候不要受到时间顺序的干扰

12.4. Java 与线程

1、并发不一定要依赖多线程(如 PHP 中常见的多进程并发)，但是在 Java 里面谈论并发，大多都与线程脱不开关系

12.4.1. 线程的实现

1、线程是比进程更轻量级的调度执行单位，线程的引入，可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源(内存地址、文件 I/O)，又

可以独立调度

2、主流的操作系统都提供了线程实现，Java 语言则提供了在不同硬件和操作系统平台下对线程操作的统一处理

- 每个已经执行 start()且还未结束的 java.lang.Thread 类的实例就代表了一个线程
- Thread 类与大部分 Java API 有显著的差别，它的所有关键方法都是声明为 Native 的
- 在 Java API 中，一个 Native 方法往往意味着这个方法没有使用或无法使用平台无关的手段来实现(当然也可能是为了执行效率而使用 Native 方法，不过，通常最高效的手段也就是平台相关的手段)

3、实现线程主要有三种方式

- 使用内核线程实现
- 使用用户线程实现
- 使用用户线程加轻量级进程混合实现

4、**使用内核线程实现**

- 内核线程(Kernel-Level Thread, KTL)就是直接由操作系统内核(Kernel，下称内核)支持的线程
- 这种线程由内核来完成线程切换，内核通过操纵调度器(Scheduler)对线程进行调度，并负责将线程的任务映射到各个处理器上
- 每个内核线程可以视为内核的一个分身，这样操作系统就有能力同时处理多件事情
- 支持多线程的内核就叫做多线程内核(Multi-Threads Kernel)
- 程序一般不会直接去使用内核线程，而是去使用内核线程的一种高级接口---轻量级进程(Light Weight Process, LWP)，轻量级进程就是我们通常意义上所讲的线程
- 由于每个轻量级进程都由一个内核线程支持，因此只有先支持内核线程，才能有轻量级进程
- 这种轻量级进程与内核线程之间 1：1 的关系成为一对一的线程模型如下

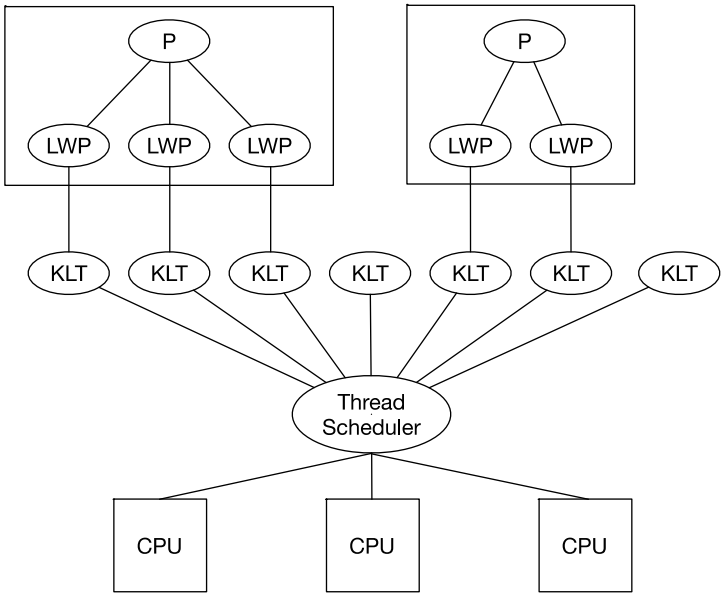

图 12-3 轻量级进程与内核线程之间 1:1 的关系

- 由于内核的支持，每个轻量级进程都成为一个独立的调度单元，即使有一个轻量级进程在系统中阻塞了，也不会影响整个进程继续工作
- 轻量级进程具有局限性
  - 由于是基于内核线程实现的，所以各种线程操作，如创建、析构及同步，都需要进行系统调用
  - 系统调用的代价相对较高，需要在用户态(User Mode)和内核态(Kernel Mode)中来回切换
  - 轻量级进程都需要有一个内核线程的支持，因此轻量级进程需要消耗一定的内核资源(如内核线程的栈空间)，因此一个系统支持轻量级进程的数量是有限的

5、**使用用户线程实现**

- 广义上讲，一个线程只要不是内核线程，就可以认为是用户线程(User Thread, UT)
- 从这个定义上讲，轻量级进程也属于用户线程，但轻量级进程的实现始终是建立在内核之上的，许多操作都要进行系统调用，效率会受到限制
- 狭义上的用户线程指的是完全建立在用户空间的线程库上，系统内核不能感知线程存在的实现
- 用户线程的建立、同步、销毁和调度完全在用户态中完成，不需要内核的帮助
- 这种线程不需要切换到内核态，因此操作可以是非常快速且低消耗的，也可以支持更大规模的线程数量
- 用户线程的优势在于不需要系统内核支援
- **劣势也在于没有系统内核的支援，所有的线程操作都需要用户程序自己处理**
  - 线程的创建、切换和调度都是要考虑的问题
  - 使用用户线程实现的程序一般比较复杂
- 除了以前在不支持多线程的操作系统中(如 DOS)的多线程程序与少数有特殊需求的程序外，现在使用用户线程的程序越来越少了，Java、Ruby 等语言都曾经使用过用户线程，最终又都放弃使用它

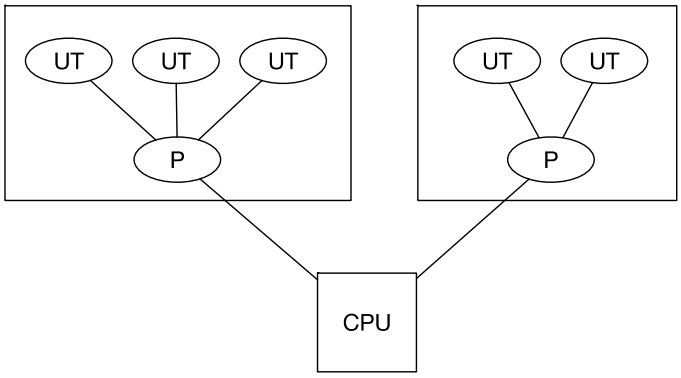

图 12-4 进程与用户线程之间 1：N 的关系

6、**使用用户线程加轻量进程混合实现**

- 在这种混合实现下，既存在用户线程，也存在轻量级进程
- 用户线程还是完全建立在用户空间中，因此用户线程的创建、切换、析构等操作依然廉价，并且可以支持大规模的用户线程并发
- 操作系统提供支持的轻量级进程则作为用户线程和内核线程之间的桥梁，

这样可以使用内核提供的线程调度功能及处理器映射，并且用户线程的系统调用要通过轻量级线程来完成，大大降低了整个进程被完全阻塞的风险

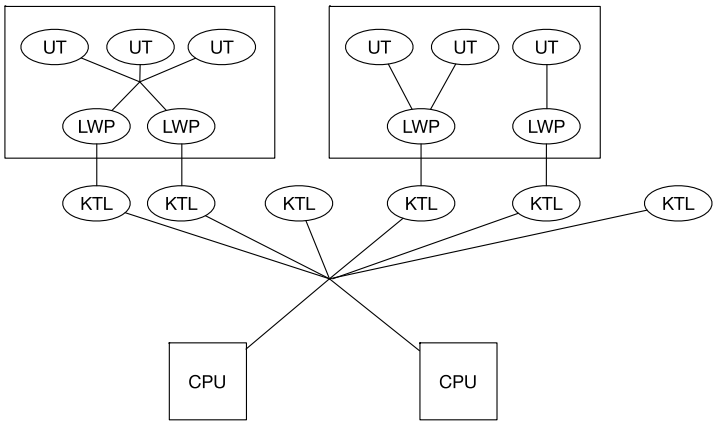

图 12-5 用户线程与轻量级进程之间 N:M 的关系

7、Java 线程的实现

- Java 线程在 JDK 1.2 之前，是基于称为"绿色线程"(Green Threads)的用户线程实现的
- **在 JDK 1.2 中，线程模型替换为基于操作系统原生线程模型来实现**
- 在目前 JDK 版本中，操作系统支持怎样的线程模型，在很大程度上决定了 Java 虚拟机的线程是怎样映射的，这点在不同平台上无法达成一致，Java 虚拟机也并未限定 Java 线程需要使用哪种线程模型来实现
- 线程模型只对线程的并发规模和操作成本产生影响，对 Java 程序编码和运行过程而言，这些差异都是透明的
- 对 Sun JDK 来说，它的 Windows 版与 Linux 版都是使用一对一线程模型实现的，一条 Java 线程就映射到一条轻量级进程之中，因为 Windows 和 Linux 系统提供的线程模型就是一对一的
- 在 Solaris 平台中，由于操作系统的线程特性可同时支持一对以(通过 Bound Threads 或 Alternate Libthread 实现)及多对多(通过 LWP/Thread Based Synchronization 实现)的线程模型，因此在 Solaris 版的 JDK 中也提供了两个该平台专有的虚拟机参数
  - -XX:+UseLWPSynchronization(默认值)
  - -XX:+UseBoundThreads

12. 4. 2. Java 线程调度

1、**线程调度指系统为线程分配处理器使用权的过程**，主要的调度方式有两种

- 协同式调度(Cooperative Threads-Scheduling)
- 抢占式线程调度(Preemptive ThreadsScheduling)

2、协同式(协作式)调度

- 线程的执行时间由线程本身控制，线程把自己的工作执行完了之后，要主动通知系统切换到另一个线程上
- 协同式调度最大的好处是实现简单，而且由于线程要把自己的事情干完后才会进行线程切换，切换操作对于线程自己是可知的，所以没有什么线程同步的问题

- 坏处：线程执行时间不可控制，甚至如果一个线程编写有问题，一直不告知系统进行线程切换，那么程序就会一直阻塞在那里

3、抢占式调度

- 每个线程将由系统来分配执行时间，线程的切换不由线程本身来决定(Java 中，Thread.yiels()可以让出执行时间，但是要获取执行时间的话，线程本身也没有什么办法)
- 线程的执行时间是系统控制的，也不会有一个线程导致整个进程阻塞的问题
- Java 使用的线程调度方式就是抢占式调度
- 虽然 Java 线程调度是系统自动完成的，但是我们还是可以"建议"系统给某些线程多分配一点时间，另外一些线程少分配一点时间---这箱操作可以通过设置线程优先级来完成
  - Java 语言一共设置了 10 个级别的线程优先级
  - 在两个线程同时处于 Ready 状态时，优先级越高的线程越容易被系统选择执行
  - 线程优先级并不太靠谱，原因是 Java 的线程是通过映射到系统的原生线程上来实现的，所以线程调度最终还是取决于操作系统
  - 虽然很多操作系统都提供线程优先级的概念，但并不见得能与 Java 线程的优先级一一对应
    - Solaris 中有 2147483648(2^{32})种优先级
    - Windows 中只有 7 种
    - 比 Java 线程优先级多的系统还好办，中间留下一点空位就行
    - 比 Java 线程优先级少的系统，如 Windows，就不得不出现几个优先级相同的情况了
- 我们不能太依赖优先级，因为它并不靠谱，优先级可能会被系统自行更改

12.4.3. 状态转换

1、Java 语言定义了 5 种线程状态，在任意一个时间点，一个线程只能有且只有其中一个状态

- 新建(New)：创建后尚未启动的线程处于这种状态
- 运行(Runable)：Runable 包括了操作系统线程状态中的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，有可能正在等待 CPU 为它分配执行时间
- 无期限等待(Waiting)：处于这种状态的线程不会被分配 CPU 执行时间，它们要等待被其他线程显式地唤醒
  - 以下方法会让线程进入无限期的等待状态
    - 没有设置 Timeout 参数的 Object.wait()方法
    - 没有设置 Timeout 参数的 Tread.join()方法
    - LockSupport.park()方法
- 限期等待(Timed Waiting)：处于这种状态的线程也不会被分配 CPU 执行时间，不过无须等待被其他线程显式地唤醒，在一定时间之后它们会由系统自动唤醒
  - 以下方法会让线程进入限期等待状态

- Thread.sleep()方法
- 设置了 Timeout 参数的 Object.wait()方法
- 设置了 Timeout 参数的 Thread.join()方法
- LockSupport.parkNanos()方法
- LockSupport.parkUntil()方法

➢ 阻塞(Blocked)：线程被阻塞了
- "阻塞状态"：在等待着获取到一个排他锁，这个时间将在另外一个线程放弃这个锁的时候发生
- "等待状态"：等待一段时间，或者唤醒动作的发生

➢ 结束(Terminated)：已经终止线程的线程状态，线程已经结束执行

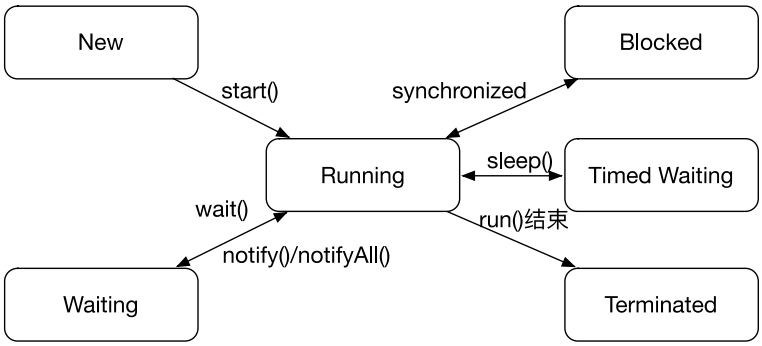

图 12-6 线程状态转换关系

Chapter 13. 线程安全与锁优化

13.1. 概述

1、软件开发初期，程序编写都是以算法为核心的，程序员会把数据和过程分别作为独立的部分来考虑，数据代表问题空间中的客体，**程序代码则用于处理这些数据**，这种思维方式直接站在计算机的角度去抽象问题和解决问题，称为**面向过程的编程思想**

2、与面向过程相对，面向对象的编程思想是站在实现世界的角度去抽象和解决问题，它把数据和行为都看做是对象的一部分，这样可以让程序员能以符合现实世界的思维方式来编写和组织程序

13.2. 线程安全

1、比较恰当的定义：当多个线程访问一个对象时，如果不用考虑这些线程在运行时环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象是线程安全的

2、上述定义比较严谨，它要求线程安全的代码都具备一个特征：**代码本身封装了所有必要的正确性保障手段(如互斥同步等)**，令调用者无须关心多线程的问题，更无须自己采取任何措施来保证多线程的正确调用

13.2.1. Java 语言中的线程安全

1、按照线程安全的"安全程度"由强至弱来排序，我们可以将 Java 语言中各种操作共享的数据分为以下 5 类：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立

2、**不可变**

- **Java 语言**特指 JDK 1.5 以后，即 Java 内存模型被修正之后的 Java 语言
- 不可变(Immutable)的对象一定是线程安全的，无论对象的方法实现还是方法的调用者，都不需要采取任何的线程安全保障措施
- 只要一个不可变对象被正确地构造出来(没有发生 this 逃逸的情况)，那其外部的可见状态永远也不会改变，永远也不会看到它在多线程之中处于不一致的状态
- 不可变带来的安全性是最简单和最纯粹的
- Java 语言中
  - 如果共享数据是一个基本类型，那么只要在定义时使用 final 关键字修饰它就可以保证它是不可变的
  - 如果共享数据是一个对象，那就需要保证对象的行为不会对其状态产生任何影响才行
  - 例如 java.lang.String 类的对象就是不可变对象，它的一系列方法都不会影响原有的值，只会返回一个新构造的字符串对象
  - 保证对象行为不影响自己状态的途径有多种
    - 最简单的就是把对象中带有状态的变量都声明为 fianl，这样在构造函数结束后，它就是不可变的

- Java API 中符合不可变要求的类型，除了 String 外，还有枚举类型，以及 java.lang.Number 的部分子类，例如 Long 和 Double 等数值包装类型，BigInteger 和 BigDecimal 等大数据类型

3、**绝对线程安全**

- 一个类要达到"不管运行时环境如何，调用者都不需要任何额外的同步措施"通常需要付出很大，甚至不切实际的代价
- 在 Java API 中标注自己是线程安全的类，大多数都不是绝对的线程安全
- 举个例子
  - java.util.Vector 是一个相对线程安全的容器，但是并不包括多个操作之间的"原子性"支持，因此仍然在必要的时候需要同步手段，详见 JavaPoint 第 11 章总结部分例子
  - 第一个操作是利用 Vector.size()获取最后一个元素的索引 i
  - 第二个操作是利用 Vector.get(i)获取最后一个元素
  - 但是由于这两个操作的组合操作不具有原子性，可能存在另一个线程在上述两个操作中间，执行了一次 Vector.remove()
  - 对于每个方法而言都是线程安全的，但是对于这一串操作而言，确是会出现这种线程非安全的情况，因此对方法的同步对于这种场景是没有作用的

4、**相对线程安全**

- 相对线程安全就是通常意义上所讲的线程安全，它需要保证对这个对象单独的操作是线程安全的，我们在调用的时候不需要做额外的保障措施
- 但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性
- Java 语言中，大部分线程安全类都属于这种类型，例如 Vector、Hashtable、Collections 的 synchronizedCollection()方法包装的集合等

5、**线程兼容**

- 线程兼容指对象本身并不是线程安全的，但是可以通过调用端正确地使用同步手段来保证对象在并发环境中可以安全使用
- 我们平时说一个类是线程不安全的，绝大多数时候是指这一种情况
- Java API 大部分类属于线程兼容的，例如 ArrayList、HashMap 等等

6、**线程对立**

- 线程对立是指无论调用端是否采用了同步措施，都无法在多线程环境中并发使用的代码
- 由于 Java 语言天生具备多线程特性，线程堆里这种排斥多线程的代码是很少见的，通常是有害的，尽量避免
- 线程对立的例子是：Thread 类的 suspend()和 resume()方法，如果有两个线程同时持有一个线程对象，一个尝试去中断线程，另一个尝试恢复线程，如果并发进行的话，无论调用时是否进行了同步，目标线程都存在死锁风险，如果 suspend()中断的线程就是将要执行 resume()的那个线程，那肯定就要产生死锁了
- 正是由于这个原因，suspend()和 resume()方法已被 JDK 声明废弃
- 常见的线程对立操作还有 System.setIn()、System.setOut()和 System.runFinalizersOnExit()等

13. 2. 2. 线程安全的实现方法

1、互斥同步机制

- 互斥同步(Mutual Exclusion & Synchronization)是常见的一种并发正确性保障手段
- **同步是指在多个线程并发访问共享数据时，保证共享数据在同一时刻只被一个(或者是一些，使用信号量的时候)线程使用**
- 互斥是实现同步的一种手段，临界区(Critical Section)、互斥量(Mutex)和信号量(Semaphore)都是主要的互斥实现方式
  - 互斥是因，同步是果
  - 互斥是方法，同步是目的
- 在 Java 中，最基本的互斥同步手段就是 synchronized 关键字
  - synchronized 关键字经过编译后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令
  - **这两个字节码指令都需要一个 reference 类型的参数来指明要锁定和解锁的对象**
    - 如果 Java 程序中的 synchronized 明确指定了对象参数，那就是这个对象的 reference
    - 如果没有明确指定，**那就是根据 synchronized 修饰的实例方法还是类方法，去取对应的对象实例或者 Class 对象来作为锁对象**
  - 在执行 monitorenter 指令时，首先要尝试获取对象的锁
    - 如果对象没有被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加 1
    - 如果获取对象锁失败，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止
  - 在执行 monitorexit 指令时会将锁计数器减 1
    - 当计数器为 0 时，锁就被释放
- 虚拟机规范对 monitorenter 和 monitorexit 的行为描述中，两点需要注意
  - 首先，synchronized 同步块对同一条线程来说是可重入的，不会出现自己把自己锁死的问题
  - 其次，同步块在已进入线程执行完之前，会阻塞后面其他线程的进入
- **Java 的线程是映射到操作系统的原生线程之上的**，如果要阻塞或唤醒一个线程，都需要操作系统来帮忙完成，**这就需要从用户态转换到核心态中**，因此状态转换需要耗费很多的处理器时间
  - 对于简单的同步块，状态转换需要的时间可能比用户代码执行的时间还要长
  - 因此 synchronized 是 Java 语言中一个重量级(Heavyweight)的操作
  - 因此必要的时候才会使用这种操作
- 还可以使用 java.util.concurrent(简称 J.U.C)包中的重入锁(ReentrantLock)来实现同步
  - 基本用法上，ReentrantLock 与 synchronized 很相似，它们都具备一样的线程重入特性，只是代码写法上有区别，一个表现为 API 层面的互斥锁(lock()和 unlock()方法配合 try/finally 语句块来完成)，另一个表现为原生语法层面的互斥锁
  - 相比 synchronized，ReentrantLock 增加了一些高级功能

- **等待可中断**：当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情
- **可实现公平锁**：当多个线程在等待同一个锁时，必须按照申请锁的时间来依次获得锁，而非公平锁不能保证这一点，非公平锁释放时，任何一个等待该锁的线程都有机会获得锁
  - synchronized 中锁是非公平的
  - ReentrantLock 默认情况下也是非公平的，但可以通过带布尔值的构造函数要求使用公平锁
- **锁可以绑定多个条件**：一个 ReentrantLock 对象可以同时绑定多个 Condition 对象，而在 synchronized 中，锁对象的 wait()和 notify()或 notifyAll()方法可以实现一个隐含的条件，如果要和多于一个的条件关联的时候，就不得不额外地添加一个锁，而 ReentrantLock 则无须这样做，只需要多次调用 newCondition()即可

- JDK 1.5 中，多线程环境下 synchronized 的吞吐量下降的非常严重(随线程数量增多)，而 ReentrantLock 则能基本保持在同一个比较稳定的水平上
- JDK 1.6 后，加入了很多针对锁的优化措施，synchronized 和 ReentrantLock 的性能基本持平了，因此 JDK 1.6 以后，性能因素就不再是选择 ReentrantLock 的理由了
- 虚拟机在未来性能改进中，肯定也会更加偏向于原生的 synchronized，所以还是提倡在 synchronized 能实现需求的情况下，优先考虑使用 synchronized 来进行同步

2、非阻塞同步

- 互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步也称为阻塞同步
- 互斥同步属于一种悲观的并发策略，总认为只要不做正确的同步措施(加锁)，那就肯定会出现问题，无论共享数据是否真的会出现竞争，都要进行加锁(这里讨论的是概念模型，虚拟机实际上会优化掉很大一部分不必要的加锁)、用户态核心转换、维护锁计数器和检查是否有被阻塞的线程需要唤醒等操作
- **随着硬件指令集的发展**，另一个选择：基于冲突检测的乐观并发策略，就是先进行操作
  - 如果没有其他线程争用共享数据，那操作就成功了
  - 如果共享数据有争用，产生了冲突，那就再采取其他的补偿措施
  - 这种乐观的并发策略的许多实现都不需要把线程挂起，因此这种同步称为非阻塞同步
  - 为什么要强调硬件指令集的发展，因为需要**操作和冲突检测这两个步骤具备原子性**，硬件保证一个从语义上看起来需要多次操作的行为只通过一条处理器指令就能完成
    - 测试并设置(Test-and-Set)
    - 获取并增加(Fetch-and-Increment)
    - 交换(Swap)
    - 比较并交换(Compare-and-Swap，简称 CAS)
    - 加载连接/条件存储(Load-Linked/Store-Conditional，简称 LL/SC)

- 前 3 条是 20 实际就存在于大多数指令集之中的处理器指令，后面的两条是现代处理器新增的
- 在 JDK 1.5 之后，Java 程序中才可以使用 CAC 操作，该操作由 sun.misc.Unsafe 类里面的 compareAndSwapInt() 和 compareAndSwapLong()等几个方法包装提供，虚拟机在内部对这些方法做了特殊的处理，即时编译出来的结果就是一条平台相关的处理器 CAS 指令，没有方法调用过程，可以无条件认为内联进去了
- 但是 Unsafe 类不是给用户程序调用的类(Unsafe.getUnsafe()的代码中限制了只有启动类加载器(Bootstrap ClassLoader)加载的 Class 才能访问它
- 我们只能采用其他的 Java API 来间接使用，如 J.U.C 里面的整数原子类，其中的 compareAndSet()和 getAndIncrement()等方法都使用了 Unsafe 类的 CAS 操作
- 尽管 CAS 看起来很美，但存在一个逻辑漏洞
  - 如果一个变量 V 除此读取的时候是 A 值，并且在准备赋值的时候检查到它仍然为 A 值，那就能说它的值没有被其他线程改变过了吗？
  - 如果在这段时间被改成了 B，然后又该回 A，那 CAS 操作就会误认为它从来没有改变过
  - 为了解决这个问题，提供了一个带有标记的原子引用类，它可以通过控制变量值的版本来保证 CAS 的正确性
  - 大部分情况下 ABA 问题不会影响程序并发的正确性，如果需要解决 ABA 问题，传统的互斥同步锁可能会比原子类更高效

3、无同步方案

- 要保证线程安全，并不是一定就要进行同步，两者没有因果关系，同步只是保证共享数据争用时的正确性的手段，如果一个方法本来就不涉及共享数据，它就自然无须任何同步措施去保证正确性，因此会有一些代码天生就是安全的
- **可重入代码**：这种代码也叫作纯代码(Pure Code)，可以在代码执行的任何时刻中断它，转而去执行另外一段代码，在控制权返回后，原来的程序不会出现任何错误
  - 相对线程安全来说，重入性是更基本的特征，他可以保证线程安全
  - 所有的可重入的代码都是线程安全的
  - 但并非所有线程安全的代码都是可重入的
  - 通过一个简单的原则来判断代码是否具备可重入性：如果一个方法，它的返回值是可预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的
- **线程本地存储**：如果一段代码中所有需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中进行，如果能保证，我们就可以把共享数据的可见范围限制在同一个线程之内，这样无须同步也能保证线程之间不能出现数据争用的问题
  - 符合这种特点的应用很多，大部分使用消费队列的架构模式(生产者-消费者模式)都会讲产品的消费过程尽量在一个线程中消费完，其中一个最重要的实例就是经典 Web 交互模型中的"一个请求对应一个服务器线

程(Thread-per-Request)"的处理方式，这种处理方式的广泛应用使得很多 Web 服务端应用都可以使用线程本地存储来解决线程安全问题

13. 3. 锁优化

1、高效并发是从 JDK 1.5 到 JDK 1.6 的一个重要改进，实现了各种锁优化技术

- 适应性自旋(Adaptive Spinning)
- 锁消除(Lock Elimination)
- 锁粗化(Lock Coarsening)
- 轻量级锁(Lightweight Locking)
- 偏向锁(Biased Locking)

13. 3. 1. 自旋锁与自适应自旋

1、互斥同步对性能最大的影响就是阻塞的实现，挂起线程和恢复线程的操作都需要转入内核态中完成，这些操作给系统的并发性能带来了很大的压力

2、共享数据的锁定状态只会持续很短一段时间，为了这段时间去挂起和恢复线程并不值得

- 我们可以让请求锁的线程"稍等一下"，但**不放弃处理器的执行时间**，看看持有锁的线程是否很快就会释放锁
- **为了让线程等待，只需要让线程执行一个忙循环，这项技术就是自旋锁**

3、自旋锁在 JDK 1.4.2 中就引入，只不过默认关闭，在 JDK 1.6 中已经改为默认开启

- 自旋等待不能代替阻塞，自旋等待避免了线程切换的开销，但是它要占用处理器的时间
- 因此如果锁被占用时间很短，自旋等待的效果就会非常好
- 如果锁被占用的时间很长，那么自旋的线程只会白白消耗处理器资源，反而带来性能上的差异
- **因此自旋等待的时间就必须要有一定的限度(默认 10 次)，如果自旋超过了限定的次数仍然没有成功获得锁，就应当使用传统的方式去挂起线程了**

4、在 JDK 1.6 中引入了自适应自旋锁，意味着自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间锁的拥有者的状态来决定

- 如果在同一个锁上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，它将允许自旋等待持续相对更长的时间
- 如果对于某个锁，自旋等待很少成功获得过，在以后要获取这个锁时将可能省略掉自旋的过程，以避免浪费处理器资源
- 随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确

13. 3. 2. 锁消除

1、锁消除是指即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除

2、锁消除的主要判定依据来源于逃逸分析的数据支持

- 如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁自然就无需进行
- 变量是否逃逸，程序员自己应该很清楚，为什么明知道不存在竞争的情况下还会加锁呢？
  - 许多同步措施不是程序员自己加入的，例如 String 的相加，在 JDK 1.5 之前，会转化为 StringBuffer 来处理

13. 3. 3. 锁粗化

1、原则上，我们在编写代码的时候，总是推荐奖同步块的作用范围限制得尽量小---只在共享数据的实际作用域中才进行同步，这样是为了使得需要同步的操作数量尽可能编小，如果存在竞争，那等待锁的线程也能尽快拿到锁

2、大部分情况下，上面的原则是正确的，但是如果一系列连续操作都对同一个对象反复加锁和解锁，甚至加锁操作出现在循环体中，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能损耗

3、例如连续的 append 操作，如果虚拟机探测到右这样一串零碎的操作都对同一个对象加锁，将会把锁同步的范围(粗化)到整个操作序列的外部，这样就只需要加锁一次即可

13. 3. 4. 轻量级锁

1、轻量级锁是 JDK 1.6 之中加入的新型锁机制

- 轻量级是相对于使用操作系统互斥量来实现的传统锁而言
- 传统的锁机制就被称为"重量级"锁
- **轻量级锁并不是用来代替重量级锁的，它的本意是，在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗**

2、HotSpot 虚拟机对象头(Object Header)分为两部分信息

- 第一部分用于存储对象自身的运行时数据，如哈希码，GC 分代年龄，这部分数据的长度在 32 位和 64 为虚拟机中分别为 32bit 和 64bit，官方称为"Mark Word"，它是实现抢良机锁和偏向锁的关键
- 另一部用于存储指向方法区对象类型数据的指针，如果是对象数组的话，还会有一个额外的部分用于存储数组长度

3、对象头信息是与对象自身定义的数据无关的额外存储成本，考虑到虚拟机的空间效率，Mark Word 被设计成一个非固定的数据结构以便在极小的空间内存储尽量多的信息

4、轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量

- "对于绝大部分的锁，在整个同步周期内都是不存在竞争的"，这是一个经验数据
- 如果没有竞争，轻量级锁使用 CAS 操作避免了使用互斥量的开销
- 但如果存在锁竞争，除了互斥量的开销外，还额外发生了 CAS 操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢

13. 3. 5. 偏向锁

1、偏向锁也是 JDK 1.6 引入的一项锁优化，它的目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能

2、如果轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把整个同步锁都消除掉，连 CAS 也不做

标记
<未完成>
???
泛型参数为什么设计成不支持基本类型