JVM笔记

第二部分 自动内存管理

第2章 Java内存区域与内存溢出异常

1）Java虚拟机管理的内存区域划分为以下几个：

- **程序计数器**：当前线程所执行的字节码的行号指示器，为了保证多线程环境下互不影响，该区域是线程私有的
- **Java虚拟机栈**：每个方法执行时，Java虚拟机栈都会同步创建一个栈帧用于存储局部变量表、操作数、动态连接、方法出口等信息。该区域同样为线程私有
- **本地方法栈**：与Java虚拟机栈所发挥功能几乎一致，只是为本地方法服务。
- **Java堆**：几乎所有的对象都在堆上分配，是垃圾收集器管理的内存区域，线程共享
- **方法区**：存储类型信息、常量、静态变量、运行时常量池（编译期生成的字面量和符号引用）等，线程共享

2）HotSpot虚拟机对象

- **对象创建**：检查类是否已被加载、为对象分配内存（分配方式：指针碰撞/空闲列表，取决于Java堆是否规整与使用的垃圾收集器算法相关，同步问题：分配内存时即同步/使用TLAB，缓冲不足分配新缓存区时同步）、内存空间置0（不包括对象头）、设置对象信息、构造函数。
- **对象布局**：对象头、实例数据、对齐填充
    - **对象头**：Mark Word（存储对象自身运行数据，如hashcode、GC分代年龄、锁状态标志等等）+ 类型指针 + 【数组长度】
    - **实例数据**：对象真正存储的有效信息
- **对象访问**：包括句柄访问和直接访问两种
    - **句柄访问**：Java堆中分配一块句柄区，Java栈上的对象引用reference中存的是句柄地址，句柄中包括了实例数据和类型数据的地址信息。优势是reference中存的是稳定的句柄地址，在对象移动时不需要修改本身，只需要修改句柄中实例数据地址（垃圾收集时移动对象很普遍）
    - **直接访问**：reference存的是对象地址，对象直接包含实例数据和类型指针。优势是速度更快，由于程序中对象访问频繁，积累下来的节省指针定位的时间开销相当可观。HotSpot虚拟机主要使用直接访问这种方式。

3）OutOfMemory异常与常用VM参数

- **Java堆溢出**：-Xms和-Xmx设置堆大小，-XX:+HeapDumpOnOutOfMemoryError可以让虚拟机在出现内存溢出异常时Dump出当前内存堆转储快照以进行事后分析。
- **虚拟机栈和本地方法栈溢出**：-Xss设置线程的栈内存容量
- **方法区和运行时常量池溢出**：-XX:PermSize和-XX:MaxPermSize设置永久代大小，在JDK7以前可间接限制常量池容量。JDK7以后常量池移至Java堆中，此时常量池容量只受堆大小影响；JDK8之后方法区使用元空间进行管理，移除永久代概念，-XX:MaxMetaspaceSize和-XX:MetaspaceSize和-XX:MinMetaspaceFreeRatio和-XX:MaxMetaspaceFreeRatio共同管理元空间
- **本机直接内存溢出**：-XX:MaxDirectMemorySize指定，若不指定则与-Xmx一致

第3章 垃圾收集器与内存分配策略

1）垃圾收集器概述

- **垃圾收集关注的三件事**：哪些内存需要回收？如何回收？什么时候回收？
- **Java虚拟机垃圾收集关注的内存**：由于程序计数器、虚拟机栈、本地方法栈这3个区域随着线程的生命周期结束而结束，内存分配和回收都具备确定性，因此垃圾收集器重点关注的是Java堆和方法区的内存回收。

2）对象如何判定死亡

- **引用计数算法**：在对象中添加一个引用计数器，每当有一个地方引用时，计数器就加1，垃圾回收计数器为0的对象。（不适用于Java，因为需要考虑很多例外情况，例如对象间相互循环引用）
- **可达性分析算法**：通过一系列称为“GC Roots”的根对象为起始节点集合，从它们开始按照引用关系向下搜索，若某个对象与GC Roots没有引用链相连则表示不可达，可以被回收。**Java中固定可作为GC Roots的对象包括以下几种**：
    - 在虚拟机栈中引用的对象，如线程的方法堆栈中用到的参数、局部变量、临时变量等
    - 在方法区中类静态属性引用的对象
    - 在方法区中常量引用的对象
    - 在本地方法栈中JNI（即Native方法）引用的对象
    - 在Java虚拟机内部的引用，如基本数据类型的Class对象，一些常驻的异常对象、系统加载类等
    - 所有被同步锁持有的对象
    - 反映Java虚拟机内部情况的JMXBean、JVM TI中注册的回调、本地代码缓存等
- **引用类型**：JDK1.2之后，Java将引用分为以下四类：
    - **强引用（Strong Reference）**：即代码中普遍存在的引用赋值，任何情况下，只要强引用关系存在，垃圾收集器就不会回收被引用的对象。
    - **软引用（Soft Reference）**：用于描述有用但非必须的对象，若系统将要发生内存溢出会将其纳入回收范围。
    - **弱引用（Weak Reference）**：同样用于描述非必须对象，但只要发生垃圾收集，就会被纳入回收范围。
    - **虚引用（Phantom Reference）**：无法通过虚引用来获取对象实例，只是用于对象被回收时能收到系统通知。
- **死亡对象回收过程**：当对象在可达性分析后被判定为不可达，则会被第一次标记，在之后的筛选中，会判断是否有必要执行finalize方法。若对象没有覆盖finalize方法或已经执行过finalize方法则无必要执行。
    - 虚拟机自动建立一个低调度优先级的Finalizer线程区去执行回收对象的finalize方法（仅触发，不保证一定等待它们执行完成）。finalize方法为对象逃离被回收的最后补救方法，通过覆盖重写finalize方法，并将对象赋予引用。任何对象的finalize方法只会被调用一次。
- **方法区的回收**：相对比于Java堆的回收，方法区的回收显得条件苛刻以及回收效率低。但大量使用反射、动态代理、CGLib加强类的框架以及动态生成JSP等情景，通常都需要虚拟机具备类型卸载的能力。方法区的垃圾回收主要聚集在两个部分：**废弃的常量和不再使用的类型。**常量是否废弃取决于是否存在引用，**而类型是否不再使用则取决于以下三个条件（需同时满足）**：
    - 该类的所有实例都已经被回收，包括子类实例
    - 加载该类的类加载器已经被回收，通常很难实现
    - 该类对应的Class对象没有被引用，无法通过反射访问该类
- **方法区回收相关参数**：-Xnoclassgc控制是否要对可回收类进行卸载、-verbose:class和-XX:+TraceClassLoading或-XX:+TraceClassUnLoading查看类加载与写在信息。

3）垃圾收集算法

- **垃圾收集算法分类**：从如何判定对象消亡的角度来说，垃圾收集算法可以分为“引用计数式垃圾收集（直接垃圾收集）”和“追踪式垃圾收集（间接垃圾收集）”。
- **分代假说**：包含弱分代假说、强分代假说和跨代引用假说。
  - **弱分代假说**：绝大多数对象都是朝生夕灭的
  - **强分代假说**：能熬过多次垃圾收集的对象越难以消亡
  - **跨代引用假说**：跨代引用相对于同代引用来说仅占极少数
- **常用垃圾收集器设计原则**：基于以上两个分代假说，收集器将Java堆分为不同的区域，将对象按照年龄分配到不同的区域，对于朝生夕灭的对象区域，每次回收时只关注是否保留少量存活的对象而不是分析是否标记需要消亡的对象；对于难消亡的对象区域，以较低的频率进行回收。
- **解决跨代引用扫面问题**：基于第三条假说，在收集新生代时，不需要去把整个老年代加入GC Roots以查看是否有引用新生代对象。而是通过在新生代上建立一个“记忆集”，将老年代划分为几块，标记出哪些块中的老年代对象存在跨代引用，在新生代收集时，仅将这几块的老年代对象加入GC Roots中。
- **分代收集术语**：
  - **部分收集（Partial GC）**：指目标不是整个Java堆的垃圾收集
  - **新生代收集（Minor/Young GC）**：指新生代垃圾收集
  - **老年代收集（Major/Old GC）**：指老年代垃圾收集
  - **混合收集（Mixed GC）**：指整个新生代和部分老年代的垃圾收集
  - **整堆收集（Full GC）**：指整个Java堆和方法区的垃圾收集
- **常见垃圾收集算法**：包括标记-清除算法、标记-复制算法、标记-整理算法
  - **标记-清除（Mark-Sweep）**：首先标记出需要消亡/存活的对象，之后再统一将标记对象消亡/保留。主要缺点为执行效率不稳定，若大量对象需要消亡，将产生大量的标记和消亡动作，使得执行效率随着对象数量增加而降低，其次是容易产生内存碎片。
  - **标记-复制（Mark-Copy）**：初始半区复制将可用内存一比一分为两块，每次只使用其中一块，在垃圾收集时将存活对象复制至另一块，之后再直接清空原始内存块，缺点是内存浪费太大。后面提出的Appel式回收将新生代分为一个Eden空间和两个Survivor空间（8：1：1），每次使用Eden空间和其中一块Survivor空间，将回收时存活的对象复制进入另一块Survivor空间，再清空Eden空间和已使用的那块Survivor空间。当Survivor不足以存放存活的对象时，将会分配至老年代空间。
  - **标记-整理（Mark-Compact）**：由于在对象存活率较高的时候标记-复制算法需要较多复制操作，以及需要担保空间，因此老年代一般不采用这种算法。标记-整理算法将存活对象移动到内存空间一端，然后将边界外的内存全部回收。
  - **对比清除与整理算法**：整理意味着移动对象，在老年代移动大量存活对象是繁琐的工作，且必须暂停用户线程，但优点是不会产生内存碎片；清除则意味着不需要暂停用户对象，但是对象分配时遇到空间碎片时只能依靠复杂的内存分配器和内存访问器来解决。因此考虑停顿时间的话采用清除算法，考虑吞吐量的话使用整理算法。也可以在正常情况下使用清除算法，在内存碎片达到一定程度后使用整理算法。

4）HotSpot垃圾算法实现细节

- **根节点枚举**：由于HotSpot虚拟机是准确式内存管理的，可以知道内存中某个位置的数据类型是不是引用类型。因此HotSpot虚拟机在类加载时，使用一组称为OopMap的数据结构来存放引用类型的对象，这样就不需要扫描时扫描所有方法区。目前所有垃圾收集器在枚举根节点GC Roots的时候都需要暂停用户线程。
- **安全点**：由于导致引用关系变化的指令很多，如果为每一条这样的指令生成对应的OopMap将是不小的负担。HotSpot采用在安全点时才记录这些信息，当用户程序执行到安全点时才能够暂停以执行垃圾收集。安全点一般选在能让程序长时间执行的过程中。而如何在垃圾收集时让所有线程都跑到安全点附近，有以下两种方式：

    - **抢先式中断**：垃圾收集时暂停所有用户线程，发现有线程未跑到安全点时则恢复该线程执行，直到它到达安全点。现在几乎没有垃圾收集器采用这种方式。
    - **主动式中断**：垃圾收集时仅设置一个标志位（标志位与安全点重合），让线程执行时去轮询这个标志，一旦发现标志为真则在最近的安全点主动中断挂起。为了保证轮询操作高效，HotSpot将使用内存保护的方式，使产生自陷异常信息。
- **安全区域**：由于存在程序不执行的情况，如线程处于Sleep或Blocked状态时，用户线程走不到安全点。因此引入安全区域的概念保证在某代码片段中引用关系不会发生改变。垃圾收集时就不必管已进入安全区域的线程，当线程离开安全区域时，若虚拟机已完成根节点枚举则当作无事发生，若未完成则等待其完成再走出安全区域。
- **记忆集与卡表**：记忆集是用于记录从非收集区域指向收集区域的指针集合的抽象数据结构。常见的记录精度包括字长精度（精确到字段）、对象精度、卡精度（精确到某块内存，也称卡表）。HotSpot中卡表是用字节数组实现的，每一个元素相当于一块内存区域（卡页）。若某块内存区域内存在跨代引用，将对应的卡表元素标识为1（变脏），块内元素加入GC Roots中。
- **写屏障**：卡表元素变脏时间原则上是在引用类型字段赋值那一刻，HotSpot是通过写屏障（参考AOP）来维护卡表状态的。伪共享问题：中央处理器用缓存行为单位存储，当并发修改的变量恰好共享一个缓存行时，就会彼此影响，如多个卡表在同一个缓存行时，*-XX:+UseCondCardMark*来决定是否添加条件判断，当卡表元素未被标记时才进行标记过程。
- **可达性分析的并发难题**：根据三色法原则，若进行根节点的可达性分析不暂停用户线程，则可能导致漏标或将存活对象标为消亡。**造成对象消失问题的2个条件如下（需同时满足）：**
    - 赋值器插入了一条或多条从黑色对象到白色对象的新引用
    - 赋值器删除了全部从灰色对象到该白色对象的直接或间接引用

  当破坏这两个条件之中的一个时，则可以在进行可达性分析时无需暂停用户线程：
    - **增量更新**：当黑色对象插入白色对象时，将黑色对象记录下来，并发扫描结束后，再以黑色对象为根再扫描一次（破坏第一个条件）
    - **原始快照**：当灰色对象要删除指向白色对象的引用时，将白色对象记录下来，并发扫描结束后，以白色对象为根再扫描一次（破坏第二个条件）

5）经典垃圾收集器

- **新生代垃圾收集器**：专注于新生代的垃圾收集，常见的有以下几种：
    - **Serial**：单线程工作的收集器，**进行垃圾收集时必须停止用户线程**，采用**标记-复制算法。**是所有垃圾收集器里内存消耗最小的，适用于系统资源较低的环境，是目前HotSpot在客户端模式下的默认新生代垃圾收集器。**可搭配的老年代垃圾收集器有Serial Old、CMS（JDK9以后禁止搭配）。此垃圾收集新生代收集全程Stop the world。**
    - **ParNew**：Serial的多线程版本，**进行垃圾收集时必须停止用户线程**，采用**标记-复制算法。**可**搭配的老年代垃圾收集器有CMS、Serial Old（JDK9禁止）。**由于后续DK版本只允许CMS与ParNew搭配，因此仍然是不少系统的新生代收集器选择。在单核心处理器的环境不见得比Serial效果好。**此垃圾收集新生代收集全程Stop the world。**
    - **Parallel Scavenge**：**关注吞吐量**（即用户代码运行时间/用户时间+垃圾收集时间）的垃圾收集器，**进行垃圾收集时必须暂停用户线程**，并发收集，采用**标记-复制算法。**提供-XX:MaxGCPauseMillis设置最大垃圾收集停顿时间和-XX:GCTimeRatio设置垃圾收集时间占总时间比例（垃圾收集时间比例 1/1+参数值，默认99）。除此之外，还有-XX:+UseAdaptiveSizePolicy用于自动调节新生代大小(-Xmn)，Eden与Survivor比例(-XX:SurvivorRatio)、晋升老年代对象大小(-XX:PretenureSizeThreshold)等参数。**可搭配的老年代垃圾收集器有Serial Old、Parallel Old。此垃圾收集新生代收集全程Stop the world。**
- **老年代垃圾收集器**：专注于老年代的垃圾收集，常见的有以下几种：
    - **Serial Old**：Serial的老年代版本，同样单线程，**进行垃圾收集时必须暂停用户线程**，采用**标记-整理算法**，同样也是使用在HotSpot客户端模式下的老年代垃圾收集器。除此之外，它还可以与Parallel Scavenge搭配使用，或者在CMS发生并发收集失败时作为后备预案。**此垃圾收集老年代收集全程Stop the world。**

- **Parallel Old**：Parallel Scavenge的老年代版本，并发收集，**进行垃圾收集时必须暂停用户线程**，采用**标记-整理算法**。可以Parallel Scavenge搭配使用组成吞吐量优先的垃圾收集器整体。**此垃圾收集老年代收集全程Stop the world。**
- **CMS（Concurrent Mark Sweep）：关注于获取最短垃圾回收停顿时间**，采用**标记-清除算法**。CMS的垃圾收集分为四个阶段，都基于获取最短停顿时间考量：
  - **初始标记**：仅标记GC Roots直接关联的对象，需短暂暂停用户线程
  - **并发标记**：从上一步的直接关联对象开始遍历，可以用户线程并发执行
  - **重新标记**：采用增量更新解决可达性分析的并发难题，需暂停用户线程
  - **并发清除**：由于采用标记-清除算法，此步骤不需要暂停用户线程

  CMS的三个主要缺点：
  - 并发标记时占用线程资源，导致应用变慢，降低总吞吐量。后考虑使用i-CMS让会回收线程和用户线程交替执行，减少对用户线程的影响。但效果一般，JDK9后被废除。
  - 标记-清除意味着无法处理浮动垃圾，当老年代剩余空间无法分配新对象时，将导致并发失败，进而启动预备方案Serial Old，停顿时间更长。通过-XX:CMSInitiatingOccupancyFraction参数设置老年代使用的内存达到一定比例后就触发CMS收集，而不是等待老年代存满。
  - 同样由于标记-清除算法，空间碎片太多时将导致老年代存放大对象困难，不得以采用Full GC造成更长时间停顿。通过-XX:+UseCmsCompactAtFullColleation设置CMS必须进行Full GC时进行内存碎片的整合，-XX:CMSFullGCsBeforeCompaction设置经过多少次Full GC才进行碎片整理。但都在JDK9后被废除。
- **全堆收集器**：同时关注新生代和老年代区域的垃圾收集，或者说无分代概念
  - **G1（Garbage First）**：基于Region的内存布局形式与面向局部收集的垃圾收集器，旨在建立停顿时间模型，将停顿时间控制在一定范围内。G1将内存进行分块，对任意需要回收的部分组成回收集，以它为回收单位，通过分析内存块的回收收益为标准。**具体细节如下：**
    - G1同样使用分代收集，它将内存分为大小相等的Region，每块Region可以为新生代的Eden、Survivor空间或老年代空间，收集器根据其角色进行不同处理。大对象（超过Region一半）将被存在Humongous区域，超大对象会被存在连续的Humongous Region中。通过-XX:G1HeapRegionSize设置大小。
    - G1以多个Region组成的回收集作为回收单位，而不是整个堆，同时分析每个Region垃圾回收的收益，这是它能控制停顿时间范围的原因。通过-XX:MaxGcPauseMillis设置用于允许的最大垃圾收集停顿时间。

    G1垃圾收集器在设计时解决的难题：
    - 跨代引用：每个Region维护自己的记忆集，记录下别的Region指向自己的指针，并标记指针的卡页范围。比起传统的卡表，这种设计无疑占用更多内存，因此G1至少耗费Java堆10%-20%的内存来维持收集器工作
    - 用户线程和标记线程并发：G1使用原始快照避免标记错误，同时为了在回收过程中能正常分配新对象内存，G1为每个Region维持两个TAMS（Top at mark start指针），新分配对象地址会被分在这两个指针之上，同时垃圾回收时默认这里的对象都是存活的。同样分配内存失败会导致Full GC的执行。
    - 可靠停顿预测模型：G1在垃圾收集时会记录每个Region的回收耗时、脏卡数量等，计算出平均值、置信度等统计信息，根据衰减平均值决定哪些Region组成的回收集能在不超过停顿时间的约束下获得最高收益。

    G1垃圾收集的步骤：
    - 初始标记：标记GC Roots能直接关联的对象，同时修改TAMS的值，需要短暂暂停用户线程
    - 并发标记：从上一步的结果开始进行扫描，使用原始快照，此阶段可与用户线程并发
    - 最终标记：处理原始快照SATB记录，需要短暂暂停用户线程
    - 筛选回收：统计分析Region，筛选需要回收的Region，将它们上面的存活对象复制到新的Region上，再清除Region。由于涉及对象移动，需要暂停用户线程。

6）低延迟垃圾收集器

- **垃圾收集器三大指标**：内存占用、吞吐量、延迟。一款优秀的收集器通常具备两种，无法达到三种。
- **Shenandoah**：为了实现在任意堆大小下低延迟，相比于CMS和G1，Shenanodoah不仅要进行并发标记，还要并发地进行对象清理后的整理。Shenandoah更像是G1的继承者，使用了同样的设计理念。**但它相比较于G1，有以下的改进：**
  - Shenandoah同样基于Region分内存区域，使用Humongous Region分配大对象，统计分析Region的回收效益。但是最重要的区别是**并发整理时，Shenandoah可以与用户线程并发。**
  - **Shenondoah目前默认不适用分代收集，即Region不区分里面存放的是什么代的对象**
  - Shenondoah舍弃了为每个Region维护一个记忆集，而是**使用连接矩阵的全局数据来记录跨Region的引用关系**，降低维护记忆集的消耗，减少伪共享发生概率。

  Shenandoah垃圾收集步骤：
  - 初始标记：标记GC Roots能直接关联的对象，需短暂暂停用户线程
  - 并发标记：从上一步结果开始遍历，与用户线程并发
  - 最终标记：处理剩余原始快照记录，统计Region回收价值，组成回收集。需短暂暂停用户线程
  - 并发清理：用于清理整个区域一个存活对象都没有的Region，与用户线程并发
  - 并发回收：与其他垃圾收集器的核心差异。将回收集中的存活对象复制到其他未使用的Region中，但是它通过增加"Brooks Pointers"的转发指针，使得移动对象过程能与用户线程并发执行。
  - 初始引用更新：建立线程集合点，确认并发回收阶段完成。需短暂暂停用户线程
  - 并发引用更新：与用户线程并发，修改引用对象，将旧值改为新值
  - 最终引用更新：修正存在于GC Roots中的引用，需暂停用户线程
  - 并发清理：经过上述阶段后，回收集中已不存在存活对象，进行直接回收，与用户线程并发

  Brooks Pointer介绍：
  - 在原有对象布局结构前面统一增加一个新的引用字段，在正常情况下指向自己，当进行移动时指向新地址。用户线程访问引用对象时，通过地址访问到对象，根据头顶的Brooks Pointer决定真实访问地址。
  - 为了避免用户线程修改引用对象的字段值时，正好处于并发移动的中间区域，导致用户线程修改了旧地址的值，因此并发对象的访问会加入同步措施，保证收集器线程和用户线程只有一个可以访问成功。
  - 由于使用了转发指针，不得不增加代码中的读屏障，造成性能开销
- **ZGC**：同样致力于在任何堆大小的情况下把垃圾收集停顿时间控制低延迟，ZGC采用了PGC和C4的优点进行设计。虽然同样基于Region管理内存，但**它与Shenandoah与G1有较大差异：**
  - **ZGC的Region具有动态的大小**，小型Region固定容量2MB，用于存放小于256KB对象；中型Region固定容量32MB，用于存放大于等于256KB小于4MB对象；大型Region容量不固定，只存放一个大于等于4MB的大对象，并且大型Region不会被重分配（移动）。
  - 采用染色指针解决并发整理的问题，如Linux64位指针高18位无法用于寻址，剩下46位中，取前4位用于染色指针技术。分别存储是否只能通过finalize方法才能访问（Finalizable）、是否进入重分配集合（即被移动过，Remapped）、是否被标记（Marked1、Marked0）。当然这也导致了Linux下ZGC能管理的堆内存不超过2^42=4TB。

  染色指针的三大优势：
  - 一旦某个Region的存活对象被移走了，它就能被立即清除，不需要等待其他所有执行该Region的引用修改值之后才能被清除，理论上只要有一个空闲Region，ZGC就能完成收集，而Shenandoah在复制阶段若几乎所有对象都存活则需要1：1的新Region，即必须有一半空闲Region。
  - 由于对象引用的变动直接写在指针上，所以因此不需要设置内存屏障如写屏障来维护这些信息，能降低对吞吐量的影响

- 染色指针存在可扩展性，地址的前18位还处于未使用状态。

ZGC的垃圾回收步骤：（省去对初始标记、最终标记的介绍，以下步骤均可以与用户线程并发）

- 并发标记：ZGC遍历对象，在指针上进行标记Marked1、Marked0
- 并发预备重分配：根据特定条件统计出需要回收的Region，组成重分配集。与G1的回收收益优先不同，ZGC会扫描所有的Region，省去维护记忆集的成本。重分配集决定了内部的存活对象会被复制到新Region上，之后集合中的Region会被清理
- 并发重分配：把重分配集中Region的存活对象复制到新的Region上，并为重分配集中的每一个Region维护一个转发表，记录旧对象到新对象的转型关系。得益于染色指针，ZGC能清除知道某个对象是否处于重分配集中，若用户线程访问到处于重分配集合中的对象，将会被预置的内存屏障截获，根据转发表修改引用的值（自愈）。这样做使得只有第一次访问旧对象时才需要转发，同时Region在复制完存活对象后，就可以被清理。
- 并发重映射：将堆中指向重分配集中对象的引用修改为新值，优先级不高，可与下一次垃圾收集的并发标记同时进行。

7）选择合适的垃圾收集器

- **Epsilon收集器**：不做任何垃圾处理的收集器，适用于负载小、运行时间短的应用
- **垃圾收集器的选择**：设备硬件情况（内存，系统等）、使用JDK的版本号等
- **虚拟机运行日志**：-Xlog，常用参数包括-Xlog:gc、-Xlog:gc+heap、-Xlog:safepoint等，注意JDK9前后版本参数的改变。
- **垃圾收集器参数**：具体参数建议上网参照文档使用

8）内存分配与回收策略

- 对象优先在Eden上分配，当Eden区没有足够空间时会执行一次Minor GC，对象无法分配时会进入担保区域即老年代中
- 大对象直接进入老年代，通过-XX:PretenureSizeThreshold设定对象进入老年代的大小
- 长期存活的对象进入老年代，通过对象头的年龄计数器，当Eden中的对象熬过第一次Minor GC时将进入Survivor区域，年龄+1。当年龄到达-XX:MaxTenuringThreshold设定的大小（默认15），进入老年代区域
- 当Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半时，大于或等于该年龄的对象直接进入老年代
- 空间分配担保：发生Minor GC前的检查
  - 老年代最大可用连续空间足以存放新生代所有对象，则确保接下来的Minor GC安全
  - 若以上条件不成立，查看-XX:HandlePromotionFailure参数是否允许担保失败，若允许则检查老年代最大可用连续空间是否大于历次晋升老年代对象的平均大小，若大于则进行冒险的Minor GC
  - 若不允许担保失败或值小于历次大小，则Minor GC改为Full GC
  - 风险是当新生代所有对象都存活时，Eden的对象无法放入小容量的Survivor，只能放入老年代，若老年代无法容纳，则还是会触发Full GC

第4章 虚拟机性能监控、故障处理工具

1）基础故障处理工具

- **根据软件可用性和授权不同，进行分类**：
  - **商业授权工具**：主要指JMC（Java Mission Control）以及它依赖JFR（Java Flight Recorder），商业环境下需要付费，但个人使用免费
  - **正式支持工具**：长期支持
  - **实验性工具**：可能会在某个JDK版本中消失，但大多数都非常稳定且功能强大

- **常用工具介绍**：下列展示常用的几种工具以及使用场景，后续JDK版本中**jcmd和jhsdb**已包含内部大多数工具，且使用起来更方便。
    - **jps**：虚拟机进程状况，展示本地虚拟机唯一ID（LVMID）和执行主类等，作为其他工具的前置工具使用（常用参数-q、-m、-l、-v）
    - **jstat**：虚拟机统计信息，根据LVMID显示对应虚拟机进程中的类加载、内存、垃圾收集、即时编译等运行数据，适用于没有图形GUI界面的服务器上
    - **jinfo**：Java配置信息，实时查看和调整虚拟机进程的各项参数
    - **jmap**：Java内存映像工具，用于生成堆转储快照（heapdump或dump文件）。除此之外，还可以查询finalize执行队列、Java堆和方法区的详细信息
    - **jhat**：堆转储快照分析工具，由于分析工作耗时且耗费硬件资源且jhat分析功能简陋，除非没有其他方法，否则不建议使用
    - **jstack**：Java堆栈跟踪工具，用于生成当前时刻的线程快照（threaddump或javacore文件），即虚拟机中每一个线程所执行的方法堆栈的集合，常用于分析线程停顿时间长的原因。

2）可视化故障处理工具

- **JHSDB**：基于服务性代理的调试工具，服务性代理是HotSpot虚拟机中一组用户映射Java虚拟机运行信息的、主要是基于Java语言实现的API集合。通过这些API，可以在一个独立的Java虚拟机进程里分析其他虚拟机的内部数据或还原堆转储快照的运行细节。**常用于查看各代大小与根据地址查看变量。**
- **JConsole**：监视与管理平台，可以查看内存、线程使用情况等，可以找出死锁
- **VisualVM**：查看虚拟机进程的信息、内存、堆转储分析等效果
- **JMC（Java Mission Control）**：类似VisualVM

第5章 调优案例分析与实战

1）大内存硬件上的程序部署策略

- **案例详情**：在线文档类型网站，使用16G物理内存和64位系统，虚拟机运行参数为-Xmx12g -Xms12g。
- **问题描述**：经常不定期出现长时间失去响应
- **问题分析**：监控后发现是由于垃圾收集停顿造成的，虚拟机使用的吞吐量优先的收集器，一次Full GC回收12G的堆停顿时间高达14秒；同时由于程序设计，访问文档会将文档序列化尝生大对象，这些大对象会直接进入老年代，就算是12G堆内存，也很容易达到满额，使得不得已频繁进行Full GC回收空间。
- **问题解决**：调整为5个32位JDK的集群，每个进程使用2GB内存，另外使用一个Apache服务器作为均衡代理。并且文档型网站主要压力在于磁盘访问和内存访问，对处理器资源不敏感，采用CMS垃圾收集器。
- **大内存单体应用的主要部署方式——单独Java虚拟机**：需要解决的问题
    - 控制Full GC的频率，关键是老年代的相对稳定，即控制代码层面上不能有大量的长生存时间的大对象产生。
    - 回收大块堆内存而导致的长时间停顿，最好使用那些不随着堆内存变大停顿时间相应变长的垃圾收集器，优先选择G1、Shenandoah、ZGC。
    - 大内存通常需要64位Java虚拟机的支持，但由于压缩指针、处理器缓存行容量等因素，64位虚拟机的性能测试通常普遍低于相同版本的32位虚拟机
    - 必须保证应用程序足够稳定，因为大型单体应用一旦发生内存溢出，生成堆转储快照是很难的，相应的分析十几G的快照文件也很困难
    - 相同的程序在64位虚拟机中消耗的内存一般高于32位虚拟机，这是由于指针膨胀以及数据类型对齐补白等造成的，可使用压缩指针（默认开启）来缓解
- **大内存单体应用的主要部署方式——多个Java虚拟机**：需要解决的问题
    - 节点之间竞争全局资源，如磁盘竞争，很容易导致I/O异常

- 很难高效利用某些资源池，如连接池，就算使用集中式的JNDI来解决，也有附加的性能代价
- 若使用32位虚拟机作为节点的话，会受到32位的内存限制，如Linux系统最高4G内存
- 大量使用本地缓存会造成较大的内存浪费，应考虑改为集中式缓存

2）集群间同步导致的内存溢出

- **案例详情**：两台双路处理器、8G内存HP小型机，共启用6个webLogic组成集群。由于未进行Session同步，一些需要全局共享的数据存放于数据库中，后由于读写频繁、竞争激烈，采用JBossCache构建全局缓存。
- **问题描述**：改用全局缓存后，出现不定期的多次内存溢出
- **问题分析**：内存回收状态正常，且每次回收后空间容量都达到稳定可用。代码层面未出现内存泄漏问题。使用-XX:HeapDumpOnOutOfMemoryError生成堆转储文件，最近一次内存溢出时分析发现存在大量org.jgroups.protocols.pbcast.NAKACK对象。这是由于JBossCache使用NACAKA对象用于集群间数据通信的数据包的有效顺序和重发。由于信息存在重发可能性，因此在确定发送成功前，信息会保存在内存中，且每次收到请求时，都有会更新最后操作时间的操作，当网络情况不能满足传输要求时，就会导致重发数据堆积，直到内存溢出
- **问题解决**：造成问题出现的根本原因在于JBossCache的缺陷，建议采取别的全局缓存方式代替

3）堆外内存导致的溢出错误

- **案例详情**：普通PC机，4GB内存，32位Windows系统，Core i5CPU，电子考试系统。
- **问题描述**：服务端不定时抛出内存溢出异常，由于系统敏感性，应不出现异常为好。调整内存为32位Windows系统最大堆内存1.6GB，仍然出现此异常。使用-XX:HeapDumpOnOutOfMemoryError检测不到任何堆内存溢出情况。使用jstat发现垃圾收集也不频繁，各区均稳定且压力小。在内存溢出后的日志发现出现了直接内存溢出。
- **问题分析**：直接内存并不算入堆内存之内，且虚拟机只有在老年代满了之后使用Full GC时才会顺便回收直接内存，否则就只能等待直接内存溢出时在catch块进行System.gc手动触发。若启动时-XX:+DisableExplicitGC禁止了手动触发，则只能等着直接内存溢出。
- **问题解决**：在处理小内存或者32位系统时，应该注意其他占有较多内存的区域设计
  - 直接内存：可通过-XX:MaxDirectMemorySize调整大小，内存不足时会抛出内存溢出异常
  - 线程堆栈：可通过-Xss调整大小，内存不足时抛出StackOverflowError或内存溢出异常
  - Socket缓存区：每个Socket都有Receive和Send缓存区，连接较多的话内存占用同样很高，若无法分配可能抛出IOException
  - JNI代码：若代码中使用了JNI调用本地库，本地库使用的内存也不在堆内存中，则是占用Java虚拟机的本地方法栈和本地内存
  - 虚拟机和垃圾收集器：虚拟机、垃圾收集器等工作也是消耗一定内存的

4）外部命令导致系统缓慢

- **案例详情**：四路处理器的Solaris 10操作系统机器，中间件位GlassFish服务器
- **问题描述**：系统在做大并发压力测试时，发现请求响应较慢，使用系统的mpstat工具发现处理器使用率较高，但占有较多处理器资源的却不是应用本身，说明出现异常占用处理器资源的情况
- **问题分析**：通过分析请求的处理发现，每个请求都会执行一个外部Shell脚本来获取系统信息，而Shell脚本的执行是通过Runtime.getRuntime().exec()来调用的，它十分消耗处理器资源
- **问题解决**：去掉Shell脚本执行，改用Java API调用的方式后恢复正常

5）服务器虚拟机进程奔溃

- **案例详情**：两台两路处理器、8G内存的HP系统，服务器是WebLogic
- **问题描述**：正常运行一段时间后，发现运行期频繁出现集群节点的虚拟机进程自动关闭，留下hs_error_pid###.log文件后，虚拟机进程就消失了
- **问题分析**：从系统日志中注意到，每个节点在虚拟机进程奔溃前，都发生过大量相同异常，Socket断开连接异常。了解到系统与OA门户做集成，进行相关业务时需要通过Web服务通知OA系统，且

Web调用响应长达3分钟。由于两边服务速度不对等，导致等待的Socket越来越多，超过虚拟机承受能力后导致虚拟机进程奔溃。

- **问题解决**：后续通知OA处理调用时间长的问题，并将异步调用改为消息队列调用后，恢复正常

6）不恰当数据结构导致内存占用过大

- **案例详情**：64位Java虚拟机，参数为-Xms4g -Xmx8g -Xmn1g，使用Par New + CMS
- **问题描述**：日常对外服务Minor GC时间30毫秒可接受，但业务上每10分钟加载一个80MB的数据文件进行分析，生成超过100万个HashMap<Long, Long>，会导致Minor GC超过500毫秒停顿。
- **问题分析**：Eden区域800M的空间在数据分析时很快被占满，且Minor GC并不能回收多少空间，在大量对象存活的情况下，Par New收集器采用的复制算法与维持引用将造成长时间停顿
- **问题解决**：在不修改代码的情况下，可以去除Survivor空间（-XX:SurvivorRatio=65536、-XX:MaxTenuringThreshold=0或者-XX:+Always-Tenure）让新生代中存活的对象在第一次Minor GC后直接进去老年代，等到Major/Full GC再清理。但根本原因还是在于Hash<Long， Long>空间效率太低，两个Long共16字节，而包装成Long对象又加入8字节Mark Word和8字节Klass指针，再包装成Map.Entry后又加入16字节对象头、8字节next字段和8字节hash字段（4字节int+4字节空白补齐），最后加上对Entry的8字节引用，实际占有88字节，有效率为16/88=18%。考虑使用其他数据结构存储。

7）由Windows虚拟内存导致的长时间停顿

- **案例详情**：一个带心跳检测功能的GUI桌面程序，每15秒发送一次心跳检测信号，若30秒内没有发现信号返回则认为断开连接
- **问题描述**：程序上线后发现心跳检测有误报的可能，查询日志发现误报时程序偶尔出现一分钟的时间完全无日志输出，停顿状态
- **问题分析**：使用-XX:+PrintGCApplicationStoppedTime、-XX:+PrintGCDateStamps、-Xloggc:gclog.log，从收集器中获取日志后发现，停顿确实是由于垃圾收集导致的，大部分垃圾收集都处于100毫秒内，但偶尔有一次接近1分钟。使用-XX:+PrintReferenceGC获取长时间停顿的具体日志信息后发现，真正执行垃圾收集的动作时间不是很长，但从准备开始，到真正开始收集则消耗了大多数时间。且观察到GUI程序最小化时，占用内存大幅度减小，但虚拟内存没有改变。
- **问题解决**：综上所述，怀疑是最小化时它的工作内存被自动交换到磁盘的页面文件之中了，导致垃圾收集时不得不恢复页面文件而产生不正常的停顿，后续使用-Dsun.awt.keepWorkingSetOnMinimize=true保持最小化运行

8）由安全点导致长时间停顿

- **案例详情**：一个比较大的承担公共计算任务的离线HBase集群，运行在JDK8上，使用G1垃圾收集器。每天有大量的MapReduce或Spark离线分析任务对其进行访问，同时有很多在线集群Relication过来的数据写入，由于集群读写压力大，且离线分析对于延迟不敏感，因此将-XX:MaxGCPauseMillis设置为500毫秒
- **问题描述**：运行一段时间后发现垃圾收集的停顿经常到达3秒以上，且实际垃圾收集器进行回收的动作只占其中几百毫秒。
- **问题分析**：使用-XX:+PrintSafepointStatistics和-XX:PrintSafepointStatisticsCount=1查看安全点日志发现有两个线程特别慢，自旋等待2255毫秒后才走到安全点。使用-XX:+SafepointTimeout和-XX:SafepointTimeoutDelay=2000查出超时2000毫秒的线程。
- **HotSpot安全点**：为避免安全点过多带来负担，HotSpot认为循环次数太少，如使用int类型或范围更小的类型作为索引值的循环默认是不会放在安全点的，这类循环称为可数循环；相应的使用long或者更大范围的类型作为索引的循环成为不可数循环。但是需要注意的是，这种情况下潜意识认为循环次数少的就执行的快，但现实中循环单体每次执行的时间可能很长，例如导致上面的线程很久才走到安全点。
- **问题解决**：使用-XX:+UseCountedLoolSafepoints参数去强制使用可数循环放置安全点，但是这个参数在JDK8下容易导致BUG，后续将代码中的可数循环容易长时间执行的循环，将索引类型改为long放置安全点。

9）对Eclipse运行速度调优

- **升级JDK版本**：JDK版本升级在性能上有一定的提高，但需要注意不同版本启动参数不同，可能导致启动失败
- **调整内存设置控制垃圾收集频率**：根据visualVM查看内存各代的使用情况与垃圾收集频率，设定合理的内存分区，降低垃圾收集频率
- **选择低延迟垃圾收集器**：升级JDK版本后，可考虑更换垃圾收集器降低延迟

第三部分 虚拟机执行子系统

第6章 类文件结构

1）无关性的基石

- **平台无关性**和**语言无关性**的基石都是**字节码Byte Code**。语言编辑器将代码编译为Class文件，虚拟机并不关心Class文件的来源是什么语言，只要遵循Java虚拟机规范即可。

2）Class类文件结构介绍

- **简介**：Class文件是一组以字节为基础单位的二进制流，各个数据项目严格按照顺序排列，之间没有分隔符。多字节的数据项按照高位在前Big-Endian的方式分割。
- **数据类型**：无符号数、表（多个无符号数或表组成的复合数据类型）
- **Class文件格式**：魔数(u4) →次版本号(u2)→主版本号(u2)→常量池数据个数→常量池→访问标志→本类索引→父类索引→接口数量→接口→字段数量→字段→方法数量→方法→属性数量→属性

3）Class类文件各部分介绍（辅助javap -verbose查看字节码内容）

- **魔数Magic**：固定为0xCAFEBABE，唯一作用是确定这个文件是不是可以被虚拟机接受的Class文件。不采用扩展名是因为可以随便改动。
- **版本号Major/Minor Version**：JDK主版本号从45开始，1.1版本之后每个大版本+1。用于标识高版本JDK可以向下兼容，但不允许向上兼容，即使Class文件格式没有明显变化，也要明确拒绝执行。JDK12之后次版本号被用于表示技术预览版的支持，即表示该Class文件使用了未被正式加入JDK特性清单的预览功能，必须把次版本号标为65535
- **常量池Constant Pool**：占用Class文件较多容量的数据，与其他项目关联最多。常量池容量的计数不同其他标准，是从1开始计算的。如0x0016表示22，代表常量池有21个常量，索引从1~21。0号索引单独空出用于表示不引用任何一个常量池项目的含义。常量池中主要存放两类数据：字面量和符号引用。常量池中的每一个常量都是一个表，目前总共有17种常量，常量结构种中除了前面的u1表示类型，其余的部分17种几乎都不一样。
- **访问标志Access Flag**：用于识别这个Class是类还是接口、是否public、是否abstract、是否被final定义等等
- **类索引、父类索引、接口索引集合**：这三个数据共同确定该类型的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定父类的全限定名，接口索引集合则表示该类实现的接口。
- **字段表集合Field**：用于描述接口或者类中声明的变量，包括字段访问标志、名称索引、标识符索引、属性集合。不会列出父类或者父接口的字段。
- **方法表集合Method**：同字段表集合类似，用于描述接口或者类种声明的方法，包括方法访问标志、名称索引、标识符索引、属性集合。同样不会列出父类或者父接口的方法。
- **属性表集合Attribute**：包含多种属性，其中最常用的是Code属性，用于存放方法体中代码的字节码指令、名称、栈帧等。

4）字节码指令简介

- Java虚拟机操作码的长度为一个字节，即操作码总类不超过256条。受此限制，不是所有指令都针对所有数据类型有独立指令，部分数据类型如byte、short、Boolean等独立指令不多，使用时将其转为Integer类型，再使用integer类型的独立指令。
- 指令集分类：加载和存储指令、运算指令、类型转换指令、对象创建与访问指令、操作数栈管理指令、控制转移指令、方法调用与返回指令、异常处理指令、同步指令等。

第7章 虚拟机类加载机制

1）类的生命周期

- 类的整个生命周期包括**加载、验证、准备、解析、初始化、使用、卸载**7个阶段。其中**验证、准备、解析统称为连接阶段。**其中，**加载、验证、准备、初始化、卸载这5个阶段的顺序是确定的。**而解析阶段在某些情况下可以在初始化阶段之后再开始。
- **有且仅有六种情况必须立即对类进行初始化**：
    - 遇到new、getstatic、putstatic、invokestatic这四个字节码指令时，若类还未初始化则必须触发。常见场景有new实例化对象、读取或设置类的静态字段（static final修饰的常量不算）、调用类的静态方法等等
    - 使用java.lang.reflect对类型进行反射调用时，若类未初始化则进行触发
    - 当初始化时发现父类还未初始化，则会触发父类的初始化；接口稍微不同，在接口初始化时，不会要求父接口完成初始化，只有在真正使用父类接口的时候（如使用了父类接口定义的常量），才会初始化父接口。
    - 当虚拟机启动时，需要指定一个主类，即包含main()方法的那个，会先初始化它
    - 当使用动态语言支持时，若java.lang.invoke.MethodHandle实例最后的解析结果未REF_getStatic、REF_putStatic、REF_invokeStatic、REF_newInvokeSpecial四种类型的方法句柄，且这个方法句柄对应的类没有进行初始化时触发
    - 当接口定义了默认方法，若这个接口的实现类发生了初始化，也会先触发接口的初始化
- **被动引用**：上述六种情况被称为对类型的主动引用，除此之外的其他类型引用都不会触发初始化，剩余的这些引用称为被动引用。常见的情况如下：
    - 通过子类引用父类的静态字段，不会导致子类被初始化
    - 通过数组定义来引用类，不会触发类的初始化，虚拟机会触发当前类的数组类的初始化，如“[java.long.Integer”整形数组类
    - 引用静态常量字段（static final修饰），不会触发类的初始化。因为常量在编译时会存入调用类的常量池中，如TestClass调用了ConstClass的静态常量，编译时通过常量传播优化，会将常量值存入TestClass类的常量池中，后续引用相当于对自己常量池的引用。

2）类加载过程——加载

- **加载阶段是类加载过程的第一个阶段，主要完成以下三件事：**
    - 通过一个类的全限定名去获取定义此类的二进制字节流
    - 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
    - 在内存中生成一个代表此类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口
- **虚拟机规范对这三点的要求并不具体，因此虚拟机实现和Java应用的灵活度很高**，例如可以从任何地方获取二进制字节流，如Class文件、Jar包、网络中等等。
- **相对于类加载过程的其他阶段，非数组类型的加载过程是开发人员可控性最强的阶段。**加载过程既可以使用虚拟机内置的引导类加载器，也可以由用户自定义类加载器完成，用户可自定义类加载器获取字节流的方式。
- 对于数组类来说，本身不由类加载器创建，而是由虚拟机直接在内存中动态构造出来的，但其元素类型仍然靠类型加载器完成。**数组类创建过程遵循以下原则**：

- 如果数组类的组件类型（去除一个维度的类型）是引用类型，则递归使用加载过程去加载这个组件类型，数组类将被表示在加载该组件类型的类加载器的类名称空间上
- 如果数组类的组件类型不是引用类型，则虚拟机将数组类标记为与引导类加载器关联
- 数组类的可访问性与组件类型一致，若组件类型不是引用类型，则默认public

3）类加载过程——验证

- **验证是连接阶段的第一步，目的是确保Class文件的字节流包含的信息符合Java虚拟机规范，且这些信息被当作代码运行后不会危害虚拟机安全。**验证阶段在类加载过程中占了相当大的比重，虽然纯粹的Java代码相对安全，但是通过其他方式产生的Class文件是有可能直接载入恶意字节码流的。
- **验证阶段主要包含四个阶段的检验动作**：
  - **文件格式验证**：**验证字节流是否符合Class文件格式规范，是否能被当前版本的虚拟机处理。**包括但不限于魔数验证、主次版本号、常量池类型是否支持等等。**该阶段主要目的是保证输入的字节流能被正确解析并存储于方法区之内，通过这个阶段后字节流才被允许进入方法区存储，后面三个检验阶段都基于方法区结构上进行，不再读取字节流。**
  - **元数据验证**：**对字节码描述的信息进行语义分析**，包括这个类是否有父类（除了Object，其他类必须有父类）、这个类的父类是否继承了不允许被继承的类、这个类不是抽象类的话是否实现了父类和接口的抽象方法、类中的字段和方法是否与父类冲突等等。**该阶段主要目的是对类的元数据信息进行语义校验。**
  - **字节码验证**：**主要目的是通过数据流分析和控制流分析，确定程序语义是合法的、符合逻辑的。**这个阶段主要**对类的方法体（Class文件的Code属性）进行校验分析，保证校验类的方法允许时不会危害虚拟机。**JDK6之后Code里增加了StackMapTable属性，描述方法体基本块（根据控制流分的代码块）开始时本地变量表和操作栈应有的状态，虚拟机验证时就只需要检查这些属性是否合法即可。
  - **符号引用验证**：**该阶段校验行为发生在虚拟机将符号引用转化为直接引用的时候，符号引用验证可以看作是类对自身之外的各类信息进行匹配性校验**，检查是否缺少或者被禁止访问它以来的某些外部类、方法、字段等资源。**该阶段很重要但不是必须执行的，若程序全部代码已经经过多次验证，则可以考虑使用-Xverify：none来关闭大部分的类验证，缩短虚拟机加载类时间。**

4）类加载过程——准备

- **准备阶段是正式为类中定义的变量（静态变量，而非静态常量）分配内存并设置类变量初始值的阶段。**通常情况下静态变量初始化时会被赋予数据类型的零值，代码中赋的值在类构造器<clinit>()方法中才会执行，即类的初始化阶段。
- 静态常量在编译时会将value生成ConstantValue属性，在准备阶段虚拟机就根据ConstantValue的设置来赋值静态常量。

5）类加载过程——解析

- **解析阶段是虚拟机将常量池中的符号引用替换为直接引用的过程。**关于符号引用和直接引用概念如下：
  - **符号引用**：用一组符号来描述所引用的目标，在Class文件中以CONSTANT_Class_info、CONSTANT_Fieldref_info等类型常量出现。
  - **直接引用**：可以直接指向目标的指针、相对偏移量或者能间接定位到目标的句柄，与虚拟机实现的内存布局直接相关。
- **Java虚拟机并未规定解析发生时间，只要求在操作符号引用的字节码指令之前，对所使用的符号引用进行解析。**对同一个符号引用的多次解析，虚拟机可以将第一次解析结果存于常量池中，若第一次解析成功则后续解析请求均视为成功，反之亦然。而对于invokedynamic指令触发的解析，则每次都进行动态解析。

- **解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符这7类符号引用**，分别对应常量池中的CONSTANT_Class| Fieldref | Methodref | InterfaceMethodref | MethodType | MethodHandle | Dynamic | InvokeDynamic _info这8中常量类型。
- **类或接口的解析3个步骤**：（当前代码类为D，符号引用为N，类或接口为C）
  - 若C不是数组类型，则虚拟机将代表N的全限定名传递给D的类加载器去加载类C。在这个阶段可能触发其他相关类的加载，若加载过程异常则解析失败。
  - 若C是数组类型，且数组的元素类型为对象，则N的描述符类似“[java.lang.Integer”，那么根据第一点规则加载数组元素类型
  - 若以上两步无异常，则C在虚拟机中已成为一个有效的类或接口，在解析完成前确认D是否对C有访问权限。JDK9之后，public不意味着拥有访问权，还需要检查模块间的访问权限
- **字段解析**：对字段符号引用的解析，会先触发字段表中索引的Class_info符号引用解析，后续步骤如下:
  - 若C本身包含了简单名称和字段描述符与目标匹配的字段，则返回字段的直接引用
  - 否则，若C中实现了接口，则按照继承关系从下往上搜索接口中是否有满足匹配的字段
  - 否则，则按照继承关系从下往上搜索父类中是否有满足匹配的字段
  - 否则，报NoSuchFieldError异常。

  通过之后需要进行权限验证。编译器采取更严格的规范，若同名字段同时出现在父类或接口中，将会被拒绝编译。
- **方法解析**：同样先触发方法表中索引的Class_info符号引用解析，后续步骤如下：
  - 若在类的方法表中发现class_index中索引的C是个接口的话，报异常
  - 否则，若C本身包含了简单名称和字段描述符与目标匹配的方法，则返回方法的直接引用
  - 否则，则按照继承关系从下往上搜索父类中是否有满足匹配的方法
  - 否则，若C中实现了接口，则按照继承关系从下往上搜索接口中是否有满足匹配的方法，若存在匹配，则说明C是抽象类，报AbstractMethodError异常
  - 否则，报NoSuchMethodError异常

  通过之后需要进行权限验证。
- **接口方法解析**：同样先触发方法表中索引的Class_info符号引用解析，后续步骤如下：
  - 若在类的方法表中发现class_index中索引的C是个类的话，报异常
  - 否则，若C本身包含了简单名称和字段描述符与目标匹配的方法，则返回方法的直接引用
  - 否则，则按照继承关系从下往上搜索父接口中是否有满足匹配的方法
  - 若C实现了多个接口，同时存在满足匹配的方法，则虚拟机会选择其中一个返回。为了严格规范，目前大多数虚拟机都会拒绝这种情况的编译
  - 否则，报NoSuchMethodError异常

  通过之后需要进行权限验证。

6）类加载过程——初始化

- **初始化阶段是类加载过程的最后一个阶段，虚拟机真正开始执行类中的代码，将主导权交给应用程序。初始化阶段就是执行类构造器<clinit>()方法的过程。**
- **<clinit>()方法是由编译器自动收集类中的所有类变量赋值动作和静态语句块中的语句合并产生的（非必需）。**编译器收集的顺序由语句出现顺序决定，**静态语句块只能访问到定义在静态语句块之前的变量，定义在之后的变量，静态语句块只可以赋值，但不能访问。**
- <clinit>()方法与类的构造方法不同，它不需要显式的调用父类构造器，**虚拟机会保证父类的构造器在子类构造器之前已经执行完毕了。**
- 接口中不能使用静态语句块，但仍然可以进行变量初始化赋值，仍然会产生<clinit>()方法，但是接**口不需要父接口的<clinit>()方法先执行，除非父类中的变量被使用了。同时接口的实现类也不执行接口的<clinit>()方法。**

- 虚拟机会进行<clinit>()方法的同步，以确定只有一个线程真正在执行类的<clinit>()方法，同时，在用一个类加载器下，一个类型只会被初始化一次。

7）类加载器

- **类加载器是用于完成"通过一个类的全限定名获取该类的二进制字节流"操作的。**对于任何一个类，都需要由它的类加载器和这个类本身来确定它在虚拟机中的唯一性，每一个类加载器都拥有一个独立的类名称空间。因此，比较两个类是否相等，只有在同一个类加载器下才有意义。
- **双亲委派模型**：自从JDK1.2以来，虚拟机就保持着三层类加载器、双亲委派模型的类加载架构：
  - **启动类加载器（Bootstrap ClassLoader）**：以C++语言实现，负责加载存放在JAVA_HOME\lib目录或者被-Xbootclasspath参数指定路径的可被虚拟机识别的类库。启动类加载器无法直接被Java程序直接引用，若需要把加载请求委派给引导类加载器去处理，则以null代替。
  - **扩展类加载器（Extension ClassLoader）**：以ExtClassLoader的Java代码形式实现，负责加载存放在JAVA_HOME\lib\ext目录中或者java.ext.dirs系统变量指定的路径中存放的类库。用于将通用性类库放置在ext目录下以扩展JAVA SE的功能，JDK9之后被模块化扩展能力取代。
  - **应用程序类加载器（Application ClassLoader）**：以AppClassLoader的Java代码形式实现，由于它是ClassLoader类中的getSystemClassLoader返回值，也成为系统类加载器，**负责加载用户类路径ClassPath上的所有类库，为程序中默认的类加载器。**
  - **双亲委派模型（Parents Delegation Model）**：除了顶层的启动类加载器外，其余类加载器都应该有自己的父类加载器，通过组合而不是继承的关系来实现。若类加载器收到类加载请求，会将请求委派给父类加载器去完成，除非递层向上的所有父类加载器都无法加载这个类，类加载器才会自己去加载。使用双亲委派模型的好处是确定了类的优先级，如Object保证会最先被加载，而不用担心用户也书写了Object类造成混乱。
  - **双亲委派模型的破坏**：双亲委派模型保证了基础类型的一致性问题，但若基础类型要调用回用户的代码，由于加载顺序问题，它肯定不认识这些代码（既父类加载器加载的类不认识子类加载器加载的类），Java虚拟机使用了线程上下文类加载器来解决这个问题（由父类加载器去请求子类加载器完成类加载）；OSGi为了实现热部署，每一个程序模块都有自己的类加载器，需要更换模块时会将模块与类加载器一起换掉以实现代码热部署。

8）Java模块化系统

- **JDK9之后引入的模块化系统实现了可配置的封装隔离机制**，模块不仅充当代码容器，还包括依赖其他模块的列表、导出的包列表（其他模块可访问）、开放的包列表（其他模块可反射访问）、使用的服务列表、提供服务的实现列表等。JDK9之后，模块通过声明的依赖关系，在验证阶段即可发现异常。
- **为了兼容传统类路径查找机制，JDK9提出了模块路径概念**，保证使用传统类路径依赖的Java程序可以直接运行在JDK9上。**访问规则如下**：
  - **JAR文件在类路径的访问规则**：所有类路径下的资源（包括JAR）都被自动打包成匿名模块，可以正常访问类路径上的所有包、JDK系统模块的所有导出包、模块路径上所有模块的导出包。
  - **模块在模块路径的访问规则**：具名模块只能访问到它依赖的模块和包，看不到匿名模块内容，即看不到传统JAR包的内容。
  - **JAR文件在模块路径的访问规则**：将一个传统且不包含模块定义的JAR包放在模块路径下，将变成自动模块，默认依赖于整个模块路径的所有模块、导出自己所有的包。
- **模块化下类加载器的改变**：
  - 首先是由于JDK已经实现了模块化构建，天然支持可扩展需求，无需ext目录，扩展类加载器被平台类加载器（Platform ClassLoader）取代。
  - 平台类加载器和应用程序加载器不再派生子URLClassLoader，所有类加载器全部继承于BuiltinClassLoader。

- 启动类加载器由虚拟机内部和Java类库共同实现。

第8章 虚拟机字节码执行引擎

1）运行时栈帧结构

- **Java虚拟机以方法作为最基本的执行单元，栈帧则是用于支持虚拟机进行方法调用和方法执行背后的数据结构，也是虚拟机运行时数据区中虚拟机栈的栈元素。**栈帧中存放了**局部变量表、操作数栈、动态连接和方法返回地址**等信息。在编译时局部变量表、操作数栈的容量大小已被写入Code属性中。对于执行引擎而言，在活动线程中，只有为了栈顶的方法才是正在运行的，该栈帧称为当前栈帧。
- **局部变量表**：
  - 用于存放**方法参数和方法内部定义的局部变量**，编译时Code属性中的max_locals确定了该表的最大容量。
  - 变量表的单位为变量槽，一个变量槽可以存放32位数据类型的变量，而64位的则以高位对齐的方式存放在两个连续的变量槽中。为了节省栈帧消耗的内存空间，变量槽是可重用的，方法体中的变量不一定会覆盖整个方法体。
  - 按照方法参数到方法内部变量顺序存放，若是实例方法，则第0位为this变量。
- **操作数栈**：
  - 用于进行方法执行时的运算操作，编译时Code属性中的max_stacks确定了最大深度。
  - 32位数据类型占用一个栈容量，64位占用两个连续的站容量。
  - 大多数虚拟机会令两个栈帧一部分重叠，将下面栈帧的操作数栈与上面的局部量表重叠，节省空间之余，在进行方法调用时可以直接公用，无需进行参数复制传递。
- **动态连接：**每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，以支持方法调用过程中的动态连接。符号引用在**运行期间**转化为直接引用的过程称为动态连接。
- **方法返回地址**：方法执行后退出有2种方式，一个是正常退出，将返回值传递给上层调用者，另一种是遇到异常且没有得到内部处理，将形成异常调用完成，不会给上层调用者任何返回值。无论哪种方式都需要返回被调用的位置，帮助当前方法退出当前栈帧，恢复上层方法执行状态。

2）方法调用

- 方法调用并不等于代码执行，而是确定被调用的方法是哪一个。由于Class文件中存的是方法的符号引用，因此实际方法入口地址，即直接引用，在类加载期间，甚至于运行期间才能确定直接引用。
- **解析**：方法在程序真正运行前就有可确定的调用版本，且运行期内不会改变。即调用目标在编译那一刻就已经确定下来的方法的调用称为解析。 目前可以在解析阶段确定唯一调用版本的方法有**静态方法、私有方法、实例构造器、父类方法、final修饰方法**5种，它们在类加载时可以直接将符号引用解析为直接引用，统称为非虚方法。
- **分派**：方法调用的另一种
  - **静态分派**：所有依赖静态类型来决定方法执行版本的分派动作都称为静态分派。最典型表现就是重载，只取决于参数的静态类型来决定方法版本，发生在编译阶段。重载的匹配规则优先级为静态类型（如char）→自动转换类型（int、long、float、double）→封装类型（Character）→实现接口类型（Serializable）→Object→变长参数
  - **动态分派**：运行期根据实际类型确定方法执行版本的分派动作称为动态分派。最典型表现就是重写，根据实际类型决定方法版本。**方法才有虚的概念，字段没有。**
  - **单分派和多分派**：根据方法的接收者与方法的参数称为宗量，分派基于宗量的多少分为单/多分派。**静态分派属于多分派，动态分派属于单分派。**
  - **动态分派的实现**：在方法区建立虚方法表或接口方法表，存放各个方法的实际入口地址，若方法在子类中没有被重写，则地址与父类的一致。则方法选择过程不需要按照接收者类型的方法

元数据进行频繁搜索合适目标，而根据虚方法索引表来实现。

3）动态类型语言支持

- **动态类型语言的关键特征是类型检查是在运行期进行而不是编译器进行的。**
- **JDK7之后引入了java.lang.invoke包实现了另一种动态确定目标方法的机制，即方法句柄Method Handle。**通过方法句柄之后Java也可以拥有类似于函数指针或委托的方法别名工具。
- **Method Hanle和Reflecting区别**：
  - 反射与方法句柄都是在模拟方法调用，但反射是模拟Java代码层次的方法调用，而方法句柄是模拟字节码层次的方法调用，还需要对字节码指令的执行权限进行校验，而反射是不需要的。
  - 反射中的java.lang.reflect.Method对象比方法句柄中的java.lang.invoke.MethodHandle对象包含的信息要更多，包括方法签名、描述以及方法属性表等，而后者仅包含执行该方法的相关信息。即Reflecting是重量级的，MethodHandle是轻量级的。
  - 方法句柄是对字节码的方法指令调用模拟，则理论上虚拟机对此做出的优化在方法句柄上也可以支持，但反射是不可能直接实现这些调用优化措施的。
  - 反射是只为Java服务的，而方法句柄则设计于为Java虚拟机上的所有语言
- invokedynamic指令把查找目标方法的决定权从虚拟机转嫁到具体用户代码中，与方法句柄作用一致，只是一个用上层代码和API实现，一个用字节码和Class属性实现。

4）基于栈的字节码解析执行引擎

- **基于栈的指令集与基于寄存器的指令集**：Javac编译器输出的字节码指令流，实际上是一种基于栈的指令集架构ISA，字节码中的指令大部分是零地址指令，依赖操作数栈进行工作。在解释执行时，栈架构指令集的指令会比寄存器架构的更多，且栈实现于内存中，频繁内存访问使得执行速度受限于内存，导致了栈架构指令集执行速度慢于寄存器指令集。
- 因此编译执行和解释执行的区别在于，经过即时编译的汇编指令流与解释代码形成的虚拟机指令流，采用的分别是基于寄存器的指令集和基于栈的指令集。

第9章 类加载及执行子系统的案例与实战

1）Tomcat

- **功能健全的web服务器需要解决的问题**：
  - 部署在同一个服务器上的两个web应用程序所使用的Java类库互相隔离
  - 部署在同一个服务器上的两个web应用程序所使用的Java类库可以共享
  - 服务器需要保证自身的安全不受部署的web应用程序所影响
  - 支持JSP的服务器需要支持hotswap功能
- **Tomcat解决上述方法的途径**：tomcat中不只有一个ClassPath，可设置common（共享类库目录）、server（Tomcat独有类库）、shared（web程序共享目录）以及common类加载器、shared类加载器、webapp类加载器、jsp类加载器等。Tomcat6之后将三个目录默认合并为lib目录

2）OSGi——灵活的类加载结构

- **OSGi（Open Service Gateway Initiative）是OSGi联盟制定的一个基于Java语言的动态模块化规范。**OSGi中的每个模块Bundle与普通Java类库区别不大，都是以jar的格式进行封装，但Bundle允许声明依赖和导出，可以实现模块级的热插拔更新，当程序升级或者调试排除错误时，可以只停用或重新安装应用程序其中一部分，而这都归功于它的类加载器架构。
- OSGi的Bundle类加载器之间只有规则，没有固定的委派关系，各个Bundle加载器都是平级的。同时，一个Bundle类加载器为其他Bundle提供服务时，会根据Export-Package严格控制访问范围。

- 但是OSGi同时也可能导致死锁，如Bundle A与B互相依赖又同时请求对方加载时。JDK7后使用的并行类加载避免死锁出现。

第四部分 程序编译与代码优化

第10章 前端编译与优化（Java文件转字节码）

1）三类编译器

- **前端编译器**：把.java转变成.class文件的过程，如JDK的Javac
- **即时编译器**：运行期把字节码转变成本地机器码的过程，如HotSpot的C1、C2编译器，Graal编译器
- **提前编译器**：直接程序编译成与目标机器指令集相关的二进制代码的过程，如JDK的额Jaotc，GNU Complie For Java
- **Java即时编译器在运行期的优化过程，支撑了程序执行效率的不断提升；而前端编译器在编译器的优化过程，支撑着程序员的编码效率和语言使用者的快捷与幸福感提高。**

2）Javac编译器

- **从Javac代码的总体结构来看，编译过程可分为1个准备和3个处理过程：**
  - **准备过程**：初始化插入式注解处理器
  - **解析与填充符号表过程**：包括词法、语法分析和填充符号表
  - **插入式注解处理器的注解处理过程**
  - **分析与字节码生成过程**：包括标注检查、数据流和控制流分析、解语法糖、字节码生成。
- **解析与填充符号表过程**：
  - **词法、语法分析**：词法分析是将源代码的字符流转变成标记集合的过程，标记是编译时的最小元素；语法分析是根据标记序列构造抽象语法树的过程，抽象语法树是一种用来描述程序代码语法结构的树形表示方式，每一个节点代表程序代码的一个语法结构，如包、类型、修饰符等。
  - **填充符号表**：符号表是由一组符号地址和符号信息构成的数据结构，内部所登记的信息在编译的不同阶段都要被用到
- **注解处理器**：JDK5之后提供的注解在运行期发挥作用，JDK6提出插入式注解处理器，提前至编译器对代码中的特定注解进行处理，影响前端编译器的工作过程。当这些插件在处理注解期间对语法树进行过修改时，编译器将回到解析与填充符号表过程，直到所有插入式注解处理器都没有再修改语法树。由于语法树中的任意元素甚至代码注释都可以在注解处理器插件中访问到，程序员的代码才有可能干涉编译器的行为，如Lombok。
- **语义分析**：语义分析的主要任务是对结构上正确的源代码进行上下文相关性质的检查，包括**标注检查和数据及控制流分析**，标注检查包括变量使用前是否被声明、变量与赋值的数据类型是否匹配等等，还有进行代码优化进行常量折叠，如int a = 1 + 2，会直接存常量3；数据流分析和控制流分析是对程序上下文逻辑进一步验证，包括局部变量使用前是否赋值、方法每条路径是否有返回值、所有受检异常是否都被处理等。**编译期的数据流分析和控制流分析与类加载时的数据控制流分析目的基本一致，但是检验范围不同，如局部常量的不变性不会存在class文件中，由编译器或运行期来保障。**
- **解语法糖**：语法糖是用于减少代码量、增加程序可读性，减少程序代码出错的可能。Java语法糖有泛型、变长参数、自动装箱拆箱等。**Java虚拟机并不直接支持这些语法，是在编译阶段被还原回原始的基础语法结构的，即解语法糖。**

- **字节码生成**：将语法树、符号表等信息转化为字节码指令，**编译器还进行少量的代码添加和转换工作。**如添加实例构造器<init>()和类构造器<clinit>()，这里指的不是默认构造函数，默认构造函数是在填充符号表的时候写入的，这里指的是收敛代码的过程，如将语句块、变量初始化、调用父类实例构造器等过程收敛到实例构造器<init>()和类构造器<clinit>()中；转换工作则包括将字符串的加操作替换为StringBuffer或StringBuilder的append操作等。

3）Java语法糖

- **泛型**：C#和Java都有泛型的语法特性，但两者的实现方式不同。**C#采用的是具现化式泛型，无论在何时泛型类型都是真实存在的类型，C#的List<Integer>和List<String>就是两种类型；而Java采用的是类型擦除式泛型，List<Integer>和List<String>在运行期是同一种类型。**除了Java泛型在编码阶段产生的不良影响，最重要的是性能较低，需要进行频繁拆箱装箱。但是Java使用这种方式是基于不需要改动字节码、不需要改动虚拟机、保证以前没有使用泛型的库也可以直接运行在JDK5上。
- **类型擦除**：Java泛型在编译时进行类型擦除，将泛型类转化为裸类型，如ArrayList<String>转为ArrayList，这也是为什么上面属于同一种类型的原因。使用时自动插入强制类型转换和检查指令。同时，由于不支持int、long等基础类型与Object的强制转型，因此泛型无法使用原生基础类型。而且，运行期间无法获取泛型真正类型信息，导致部分代码需要指定真实类型信息。
- **自动拆箱装箱**：包装类的==运算在没有算法运算的时候是不会自动拆箱的，以及equals方法不处理数据转型关系。
- **条件编译**：编译器会把条件分支中不成立的代码块消除。

第11章 后端编译与优化（字节码转机器指令）

1）即时编译器

- 目前的主流虚拟机HotSpot与J9，**Java程序最初都是通过解释器解释执行的，当虚拟机发现某个方法或代码块运行频率高时，将认定其为热点代码，运行时将其编译成本地机器码，并进行代码优化**，此时完成这项任务的后端编译器被称为即时编译器。
- **即时编译器需要解决的问题（以HotSpot为例）**：
  - 为何HotSpot虚拟机要使用解释器和编译器并存的架构
  - 为何HotSpot虚拟机要实现两个或三个不同的即时编译器
  - 程序何时使用解释器执行？何时使用编译器执行？
  - 哪些程序代码会被编译成本地代码？如何编译本地代码？
  - 如何从外部观察到即时编译器的编译过程和编译结果？
- **解释器与编译器的优势**：
  - 当程序需要快速启动和执行时，解释器可以首先发挥作用，省去编译的时间，立即运行；当程序启动后，随着时间推移，编译器逐渐将越来越多的代码编译成本地代码，以减少解释器的中间损耗，获得更高执行效率。
  - 当程序运行内存资源限制较大时，可以使用解释执行节约内存，反之可以使用编译器提高效率
  - 解释器还可以作为编译器激进优化时后备的逃生门
- **虚拟机运行模式**：
  - **混合模式**：解释器与编译器搭配使用
  - **解释模式**：只使用解释器（-Xint）
  - **编译模式**：优先使用编译方式执行程序，但编译无法进行时会使用解释器（-Xcomp）
- **分层编译**：分层编译出现前，虚拟机采用解释器和其中一个编译器直接搭配工作，可选择客户端编译器C1（-client）或者服务端编译器（-server）。分层编译出现后，根据编译器编译、优化的规模和耗时划分出**不同的编译层次**：
  - 第0层：程序纯解释执行，解释器不启用性能监控功能（Profiling）
  - 第1层：使用客户端编译器编译执行，进行简单可靠的稳定优化，不开启性能监控功能

    - 第2层：使用客户端编译器编译执行，仅开启方法与回边次数统计等性能监控功能
    - 第3层：使用客户端编译器编译执行，开启全部性能监控，统计所有包括分支调整、虚方法调用版本等全部统计信息
    - 第4层：使用服务端编译器编译执行，启用更多编译耗时更长的优化，并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，编译器、客户端编译器和服务端编译器是**同时工作**的。热点代码可能被多次编译，用客户端可以获取更高编译速度，服务端可以获取更好编译质量，解释执行则可以无需额外监控，**在进行服务端高复杂度的优化时，可以先采用客户端简单优化争取更多编译时间。**

- **编译对象**：热点代码主要有两类：**被多次调用的方法和多次执行的循环体。**这两种情况的编译对象都是整个方法体，后者的执行入口即字节码指令执行开始行稍有不同，当方法的栈帧还在栈上时，方法就被替换了，称为栈上替换。
- **编译触发条件**：**分析某段代码是否热点代码的热点探测方式有两种**：
    - 基于采样的热点探测：周期性检查各个线程的调用栈顶，若发现某个方法经常在栈顶，则认定为热点代码。实现简单，但容易受到线程阻塞等其他外界因素影响
    - 基于计数器的热点探测：为每个方法体甚至代码块建立计数器，统计执行次数，达到阈值后判定为热点方法。需要为每个方法维护计数器，但结果更精准。

  HotSpot使用第二种方式，并为此提供了两种计数器：**方法调用计数器和回边计数器。**当这两个计数器达到阈值后就会触发即时编译。
- **HotSpot计数器规则**：
    - **方法调用计数器**：方法被调用时检测是否有编译版本，否则计数器+1，判断与回边计数器的值之和超过阈值（-XX:CompileThreshold）时请求后台编译，默认不会同步等待编译完成才执行，达到阈值后，没完成编译前还是使用解释执行；默认设置下，方法调用计数器统计的是一段时间内的方法调用次数，若超过一定的时间限度（方法统计的半衰周期）后仍未超过阈值，将进行减半处理。
    - **回边计数器**：虚拟机在客户端模式与服务端模式下的回边计数器阈值计算方式不同。当回边计数器与方法调用计数器之和超过回边计数器阈值时，将会提交栈上替换编译请求，并且将回边计数器值降低以便循环可以继续进行。回边计数器没有半衰概念，达到阈值后，将会同时将方法调用计数器也调整到溢出状态。
- **编译过程**：服务端编译器与客户端编译器的编译过程有所差别
    - 客户端编译：相对简单快速的三段式编译，主要关注局部性优化
    - 客户端编译阶段一：前端将字节码编译成高级中间代码HIR，在此之前在字节码上进行过方法内联、常量传播等优化
    - 客户端编译阶段二：平台相关的后端将从HIR产生低级中间代码LIR，在此之前在HIR上进行如空值检查消除、范围检查消除等优化
    - 客户端编译阶段三：平台相关的后端使用线性扫描算法在LIR上分配寄存器，并在LIR上做窥孔优化，然后产生机器代码。
    - 服务端编译：服务端编译能执行大部分经典优化动作，如无用代码消除、消除公共子表达式、常量传播、循环表达式外提、基本块重排序、范围检查消除、空值检查消除等。虽然以即时编译标准看，服务端编译比较缓慢，但是相对于客户端编译器输出的代码有很大质量提高，能大幅度减少本地代码执行时间，从而抵消编译时间开销。
- **查看与分析即时编译结果**：以下部分运行参数需要FastDebug或SlowDebug优化级别HotSpot虚拟机
    - **确认代码是否触发即时编译**：-XX:+PrintCompliation，输出即时编译时被编译成本地代码的方法名称，带%的是由于回边计数器触发的栈上替换编译
    - **输出方法内联信息**：-XX:+PrintInlining
    - **输出编译后汇编代码**：-XX:+PrintAssembly（汇编代码与平台相关，使用前需要安装匹配的反汇编适配器）
    - **输出比较接近最终结果的中间代码**：-XX:+PrintOptoAssembly服务端模式虚拟机，-XX:+PrintLIR客户端模式虚拟机（不需要HSDIS插件支持）

- **跟踪本地代码生成具体过程**：-XX:+PrintCFGToFile客户端编译器、-XX:+PrintIdealGraphFile（服务端编译器），将编译过程各个阶段的数据如字节码、HLR、LIR、寄存器分配过程等输出到文件，再搭配HotSpot Client Compiler Visualizer（客户端）或Ideal Graph Visualizer（服务端）进行分析。

2）提前编译器

- Java核心优势是平台中立性，即一次编译，到处运行。因此当时平台相关的提前编译并未受到较大关注。直到Android上使用提前编译的ART出现后打败了使用即时编译的Dalvik虚拟机，提前编译再次受到大家关注。
- **提前编译的分支**：
  - **一种分支与传统C、C++编译器类似，在程序运行前将代码编译成机器代码的静态编译工作。**由于即时编译会消耗程序真正运行的时间和资源，因此很多耗时的优化措施都只是采用相对激进的优化或者不太精确的最可能状态来优化。提前编译的话，如Graal VM的Substrate VM就可以在创建本地镜像时采取全程序优化措施来获得更好的运行时性能，毕竟创建镜像时间再慢也不影响程序运行。
  - **另一种分支是把原本即时编译器在运行时的编译工作提前做好并保存下来，下次运行时加载使用，本质就是给即时编译器做缓存加速。**通过这种方式来改善Java程序启动时间和需要一段时间预热后才能达到最高性能的问题，这种提前编译也被称为**动态提前编译或即时编译缓存。**基于Graal VM的Jaotc提前编译器则可以针对目标机器对应用程序进行提前编译，HotSpot运行时可以直接加载这些编译的结果，但需要注意的是，这种提前编译不仅与目标机器相关，还与HotSpot运行参数绑定。
- **提前编译器相对于即时编译器的天然劣势**：
  - **性能分析制导优化**：静态编译无法获取运行时的性能监控信息，只能按照一些启发性条件去猜测，而运行时才能知道程序信息的偏好。因此无法做到像即时编译器那样可以更精确的分析优化
  - **激进预测性优化**：这是很多即时编译器优化措施的基础，但静态化编译为了保证优化后所有程序外部可见影响和优化前等效，优化无法采用激进措施。而即时编译器则可以采用激进措施进行优化，即使走进罕见分支也可以使用低级编译器或者解释器执行。
  - **链接时优化**：Java的Class文件是可以在运行时才加载到虚拟机内存中的，当出现跨链接库边界的调用时，一些例如内联等优化操作将变得困难。

3）编译器优化技术

- **最重要的优化技术之一：方法内联**
  - 内联除了消除方法调用的成本，还为其他优化手段建立良好基础。由于Java实例方法都是虚方法，直到运行时才能知道才能知道方法的实际版本是什么，因此虚方法和内联本身是矛盾的。
  - 为了解决虚方法内联优化困难，Java引入了类型继承关系分析CHA，用于确定目前已加载的类、接口是否有多于一种实现、是否存在子类、子类是否重写了父类方法等。编译器在内联时，遇到非虚方法则直接内联，此时保证安全；若遇到虚方法，会向CHA查询该方法是否有多个目标版本可选择，若只有一个版本，则假设应用程序此时全貌如现在运行的样子，进行守护内联；若存在多个版本，则进行内联缓存减少方法调用开销。由于Java动态链接，因此CHA随时可能改变，这种激进的优化措施需要逃生门。
- **最前沿的优化技术之一：逃逸分析**
  - **逃逸程度**：分析对象动态作用域，若对象在方法内被定义后可能被其他方法引用，则成为方法逃逸；若可能被外部线程引用，则称为线程逃逸。还有不逃逸。
  - **栈上分配**：若对象不逃逸或仅存在方法逃逸，则可以考虑在栈上分配内存，以减少垃圾回收器的工作，对象的内存将随着栈帧出栈而销毁。
  - **标量替换**：若对象不逃逸，则可以将对象的成员变量恢复成原始类型来访问，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优化创建条件。

    - **同步消除**：若对象不逃逸或仅存在方法逃逸，则不需要对变量进行同步措施，可以将同步措施消除。
    - **缺陷**：逃逸分析计算成本高，且不能保证逃逸分析带来的性能收益高于消耗
- **语言无关的经典优化技术之公共子表达式消除**：若一个表达式之前已经被计算过了，且先前的计算到现在表达式中所有变量值都没有发生改变，则不需要再重新计算。
- **语言无关的经典优化技术之数组边界检查消除**：Java在进行数组读写访问时将会自动进行上下界的范围检查，每次数组访问都隐含一次条件判定操作，将带来一定的性能负担。则可以将运行期的检查提前到编译期完成，在编译器根据数据流分析是否越界，执行时无须再次判断。