一、JDK 和 JRE 是什么?

JDK 是用于支持 Java 程序开发的最小环境。包括：Java 程序设计语言、Java 虚拟机、Java API 类库

JRE 是支持 Java 程序运行的标准环境。包括：Java SE API 子集、Java 虚拟机

二、运行时数据区域包括哪些?

1. 程序计数器（线程私有）

程序计数器（Program Counter Register）是一块**较小的内存空间**，可以看作是当前线程所执行字节码的**行号指示器**。分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器完成。

由于 Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。为了线程切换后能恢复到正确的执行位置，每条线程都需要一个**独立的程序计数器**，各线程之间的计数器互不影响，独立存储。

1. 如果线程正在执行的是一个 Java 方法，计数器记录的是正在执行的虚拟机字节码指令的地址；
2. 如果正在执行的是 Native 方法，这个计数器的值为空。

程序计数器是唯一一个没有规定任何 OutOfMemoryError 的区域。

2. Java 虚拟机栈（线程私有）

Java 虚拟机栈（Java Virtual Machine Stacks）是线程私有的，生命周期与线程相同。

虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会创建一个栈帧（Stack Frame），存储

1. 局部变量表
2. 操作栈
3. 动态链接
4. 方法出口

每一个方法被调用到执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

这个区域有两种异常情况：

1. StackOverflowError：线程请求的栈深度大于虚拟机所允许的深度
2. OutOfMemoryError：虚拟机栈扩展到无法申请足够的内存时

3. 本地方法栈（线程私有）

虚拟机栈为虚拟机执行 Java 方法（字节码）服务。

本地方法栈（Native Method Stacks）为虚拟机使用到的 Native 方法服务。

4. Java 堆（线程共享）

Java 堆（Java Heap）是 Java 虚拟机中**内存最大**的一块。Java 堆在虚拟机启动时创建，被所有线程共享。

作用：**存放对象实例**。垃圾收集器主要管理的就是 Java 堆。Java 堆在**物理上可以不连续**，只要逻辑上连续即可。

5. 方法区（线程共享）

方法区（Method Area）被所有线程共享，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。

6. 运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。保存 Class 文件中的符号引用、翻译出来的直接引用。运行时常量池可以在运行期间将新的常量放入池中。

三、Java 中对象访问是如何进行的?

```
Object obj = new Object();
```

对于上述最简单的访问，也会涉及到 Java 栈、Java 堆、方法区这三个最重要内存区域。

```
Object obj
```

如果出现在方法体中，则上述代码会反映到 Java 栈的本地变量表中，作为 reference 类型数据出现。

```
new Object()
```

反映到 Java 堆中，形成一块存储了 Object 类型所有对象实例数据值的内存。Java堆中还包含对象类型数据的地址信息，这些类型数据存储在方法区中。

四、如何判断对象是否“死去”?

1. 引用计数法

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+1,；当引用失效时，计数器就-1；任何时刻计数器都为0的对象就是不能再被使用的。

缺点：很难解决对象之间的循环引用问题。

2. 根搜索算法

通过一系列的名为“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 没有任何引用链相连（用图论的话来说就是从 GC Roots 到这个对象不可达）时，则证明此对象是不可用的。

五、Java 的4种引用方式?

1. 强引用

```
Object obj =  new  Object();
```

代码中普遍存在的，像上述的引用。只要强引用还在，垃圾收集器永远不会回收掉被引用的对象。

2. 软引用

用来描述一些**还有用，但并非必须**的对象。软引用所关联的对象，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范围，并进行**第二次回收。**如果这次回收还是没有足够的内存，才会抛出内存异常。提供了 **SoftReference** 类实现软引用。

3. 弱引用

描述非必须的对象，强度比软引用更弱一些，被弱引用关联的对象，只能生存到下一次垃圾收集发生前。当垃圾收集器工作时，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。提供了 **WeakReference** 类来实现弱引用。

4. 虚引用

一个对象是否有虚引用，完全不会对其生存时间够成影响，也无法通过虚引用来取得一个对象实例。为一个对象关联虚引用的唯一目的，就是希望在**这个对象被收集器回收时**，收到一个**系统通知。**提供了 **PhantomReference** 类来实现虚引用。

六、有哪些垃圾收集算法?

1. 标记-清除算法（Mark-Sweep）

分为**标记**和**清除**两个阶段。首先标记出所有需要回收的对象，在标记完成后统一回收被标记的对象。

缺点：

1. 效率问题。标记和清除过程的效率都不高。
2. 空间问题。标记清除之后会产生大量**不连续的内存碎片**，空间碎片太多可能导致，程序**分配较大对象时**无法找到足够的连续内存，不得不提前发出另一次垃圾收集动作。

2. 复制算法（Copying）- 新生代

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。当这一块的内存用完了，就**将存活着的对象复制到另一块上面**，然后再把已经使用过的内存空间一次清理掉。

优点：复制算法使得每次都是针对其中的一块进行内存回收，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。

缺点：将内存缩小为原来的一半。在对象存活率较高时，需要执行较多的复制操作，效率会变低。

应用：

商业的虚拟机都采用复制算法来**回收新生代。**因为新生代中的对象容易死亡，所以并不需要按照1:1的比例划分内存空间，而是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 和其中的一块 Survivor。

当回收时，将 Eden 和 Survivor 中还存活的对象一次性拷贝到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。Hotspot 虚拟机默认 Eden 和 Survivor 的大小比例是**8:1**，也就是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%的内存是会被“浪费”的。

3. 标记-整理算法（Mark-Compact）- 老年代

标记过程仍然与“标记-清除”算法一样，但不是直接对可回收对象进行清理，而是**让所有存活的对象向一端移动，然后直接清理掉边界以外的内存。**

4. 分代收集算法

根据对象的存活周期，将内存划分为几块。一般是把 **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。

- **新生代**：每次垃圾收集时会有大批对象死去，只有少量存活，所以选择**复制算法**，只需要少量存活对象的复制成本就可以完成收集。
- **老年代**：对象存活率高、没有额外空间对它进行分配担保，必须使用“**标记-清理**”或“**标记-整理**”算法进行回收。

七、Minor GC 和 Full GC有什么区别?

Minor GC：新生代 GC，指发生在新生代的垃圾收集动作，因为 Java 对象大多死亡频繁，所以 Minor GC 非常频繁，一般回收速度较快。 **Full GC**：老年代 GC，也叫 Major GC，速度一般比 Minor GC 慢 10 倍以上。

八、Java 内存

1. 为什么要将堆内存分区?

对于一个大型的系统，当创建的对象及方法变量比较多时，即堆内存中的对象比较多，如果逐一分析对象是否该回收，效率很低。分区是为了进行**模块化管理**，管理不同的对象及变量，以提高 JVM 的执行效率。

2. 堆内存分为哪几块?

1. Young Generation Space 新生区（也称**新生代**）
2. Tenure Generation Space养老区（也称**老年代**）
3. Permanent Space 永久存储区（也称**永生代**）

3. 分代收集算法

内存分配有哪些原则?

1. 对象优先分配在 Eden
2. 大对象直接进入老年代
3. 长期存活的对象将进入老年代
4. 动态对象年龄判定
5. 空间分配担保

新生代采用复制算法

主要用来**存储新创建的对象**，内存较小，垃圾回收频繁。这个区又分为三个区域：**一个 Eden Space 和两个 Survivor Space。**

- 当对象在堆创建时，将进入年轻代的Eden Space。
- 垃圾回收器进行垃圾回收时，扫描Eden Space和A Suvivor Space，如果对象仍然存活，则复制到B Suvivor Space，如果B Suvivor Space已经满，则复制 Old Gen
- 扫描A Suvivor Space时，如果对象已经经过了几次的扫描仍然存活，JVM认为其为一个Old对象，则将其移到Old Gen。
- 扫描完毕后，JVM将Eden Space和A Suvivor Space清空，然后交换A和B的角色（即下次垃圾回收时会扫描Eden Space和B Suvivor Space。

老年代采用标记-整理算法

主要用来**存储长时间被引用的对象。**它里面存放的是经过**几次在 Young Generation Space 进行扫描判断过仍存活的对象**，内存较大，垃圾回收频率较小。

永生代

存储不变的类定义、字节码和常量等。

九、Class文件

1. Java虚拟机的平台无关性

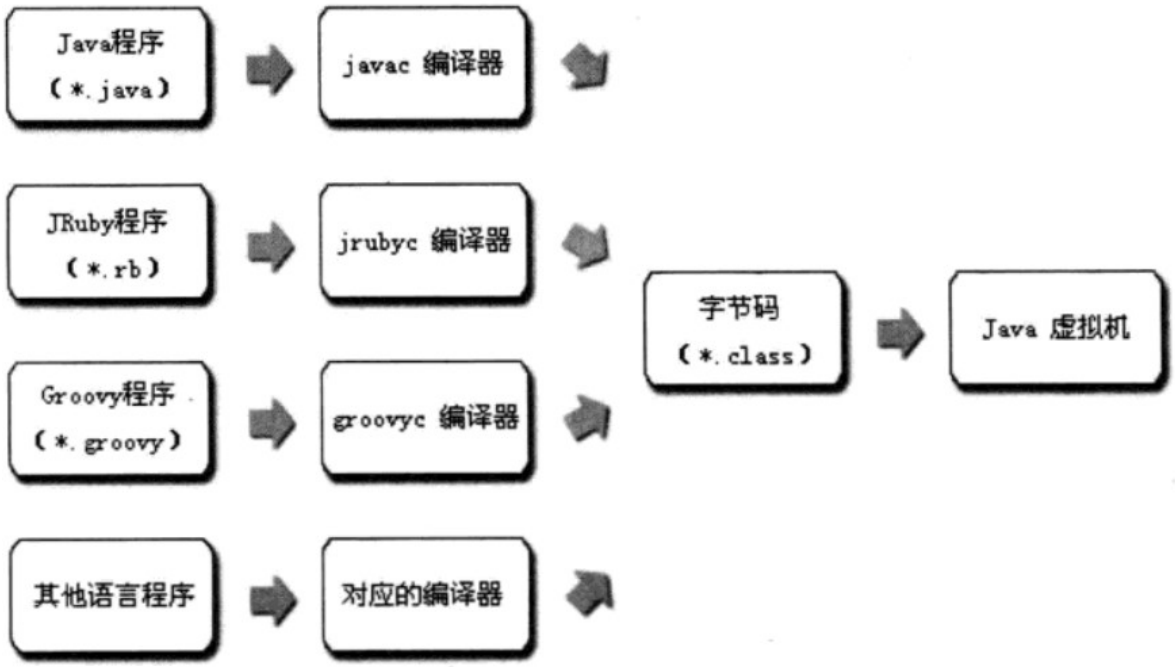

图 6-1　Java 虚拟机提供的语言无关性

2. Class文件的组成

Class文件是一组以**8位字节**为基础单位的**二进制流**，各个数据项目间没有任何分隔符。当遇到8位字节以上空间的数据项时，则会按照**高位在前**的方式分隔成若干个8位字节进行存储。

3. 魔数与Class文件的版本

每个Class文件的头4个字节称为**魔数**（Magic Number），它的唯一作用是用于确定这个文件**是否为一个能被虚拟机接受的Class文件**。OxCAFEBABE。

接下来是Class文件的**版本号**：第5,6字节是次版本号（Minor Version），第7,8字节是主版本号（Major Version）。

使用JDK 1.7编译输出Class文件，格式代码为：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	CA	FE	BA	BE	00	00	00	33	00	16	07	00	02	01	00	09
00000010	54	65	73	74	43	6C	61	73	73	07	00	04	01	00	10	6A

前四个字节为魔数，次版本号是0x0000，主版本号是0x0033，说明本文件是**可以被1.7及以上版本的虚拟机执行的文件**。

十、类加载器

1. 类加载器的作用

类加载器实现类的加载动作，同时用于确定一个类。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。**即使两个类来源于同一个Class文件，只要加载它们的类加载器不同，这两个类就不相等。

2. 类加载器有哪些

- **启动类加载器**（Bootstrap ClassLoader）：使用C++实现（仅限于HotSpot），是虚拟机自身的一部分。负责将存放在**\lib目录中的类库**加载到虚拟机中。其无法被Java程序直接引用。
- **扩展类加载器**（Extention ClassLoader）由ExtClassLoader实现，负责加载**\lib\ext目录中的所有类库**，开发者可以直接使用。
- **应用程序类加载器**（Application ClassLoader）：由APPClassLoader实现。负责加载**用户类路径（ClassPath）上所指定的类库。**

十一、类加载机制

1. 什么是双亲委派模型?

双亲委派模型（Parents Delegation Model）要求除了顶层的启动类加载器外，其余加载器都应当**有自己的父类加载器。**类加载器之间的父子关系，通过组合关系复用。

工作过程：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求**委派给父类加载器**完成。每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中，只有到父加载器反馈自己无法完成这个加载请求（它的搜索范围没有找到所需的类）时，子加载器才会尝试自己去加载。

2. 为什么要使用双亲委派模型，组织类加载器之间的关系?

Java类随着它的类加载器一起具备了一种带优先级的层次关系。比如java.lang.Object，它存放在rt.jar中，无论哪个类加载器要加载这个类，最终都是委派给启动类加载器进行加载，因此Object类在程序的各个类加载器环境中，都是同一个类。

如果没有使用双亲委派模型，让各个类加载器自己去加载，那么Java类型体系中最基础的行为也得不到保障，应用程序会变得一片混乱。

3. 什么是类加载机制?

Class文件描述的各种信息，都需要**加载到虚拟机**后才能运行。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，这就是虚拟机的类加载机制。

十二、虚拟机和物理机的区别是什么?

这两种机器都有代码执行的能力，但是：

- **物理机**的执行引擎是**直接建立在处理器、硬件、指令集和操作系统层面**的。
- **虚拟机**的执行引擎是自己实现的，因此可以**自行制定**指令集和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。

十三、运行时栈帧结构

栈帧是用于支持虚拟机进行**方法调用和方法执行**的数据结构， 存储了方法的

- 局部变量表
- 操作数栈
- 动态连接

- 方法返回地址

每一个方法从调用开始到执行完成的过程，就对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

十四、Java 方法调用

1. 什么是方法调用?

方法调用唯一的任务是**确定被调用方法的版本**（调用哪个方法），暂时还不涉及方法内部的具体运行过程。

2. Java的方法调用，有什么特殊之处?

Class文件的编译过程**不包含**传统编译的**连接步骤**，一切方法调用在Class文件里面存储的都只是**符号引用**，而不是方法在实际运行时内存布局中的入口地址。这使得Java有强大的动态扩展能力，但使Java方法的调用过程变得相对复杂，需要在类加载期间甚至到运行时才能确定目标方法的直接引用。

3. Java虚拟机调用字节码指令有哪些?

- invokestatic：调用静态方法
- invokespecial：调用实例构造器方法、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

4. 虚拟机是如何执行方法里面的字节码指令的?

- 解释执行（通过解释器执行）
- 编译执行（通过即时编译器产生本地代码）

当主流的虚拟机中都包含了即时编译器后，Class文件中的代码到底会被解释执行还是编译执行，只有虚拟机自己才能准确判断。

Javac编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生成线性的字节码指令流的过程。因为这一动作是在Java虚拟机之外进行的，而解释器在虚拟机的内部，所以Java程序的编译是半独立的实现。

十五、基于栈的指令集和基于寄存器的指令集

1. 什么是基于栈的指令集?

Java编译器输出的指令流，里面的指令大部分都是零地址指令，它们依赖**操作数栈**进行工作。

计算“1+1=2”，基于栈的指令集是这样的：

```
iconst_1
iconst_1
iadd
istore_0
```

两条iconst_1指令连续地把两个常量1压入栈中，iadd指令把栈顶的两个值出栈相加，把结果放回栈顶，最后istore_0把栈顶的值放到局部变量表的第0个Slot中。

2. 什么是基于寄存器的指令集?

最典型的是x86的地址指令集，依赖寄存器工作。

计算“1+1=2”，基于寄存器的指令集是这样的：

```
mov eax, 1
add eax, 1
```

mov指令把EAX寄存器的值设为1，然后add指令再把这个值加1，结果就保存在EAX寄存器里。

3. 基于栈的指令集的优缺点?

优点：

- 可移植性好：用户程序不会直接用到这些寄存器，由虚拟机自行决定把一些访问最频繁的数据（程序计数器、栈顶缓存）放到寄存器以获取更好的性能。
- 代码相对紧凑：字节码中每个字节就对应一条指令
- 编译器实现简单：不需要考虑空间分配问题，所需空间都在栈上操作

缺点：

- 执行速度稍慢
- 完成相同功能所需的指令数多

频繁的访问栈，意味着频繁的访问内存，相对于处理器，内存才是执行速度的瓶颈。

十六、Javac编译过程分为哪些步骤?

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

图 10-4　Javac 的编译过程[①]

十七、什么是即时编译器?

Java程序最初是通过解释器进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会把这些代码认定为“**热点代码**”（Hot Spot Code）。

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成为**即时编译器**（Just In Time Compiler，JIT编译器）。

十八、解释器和编译器

许多主流的商用虚拟机，都同时包含解释器和编译器。

- 当程序需要**快速启动和执行**时，**解释器**首先发挥作用，省去编译的时间，立即执行。
- 当程序运行后，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，可以**提高执行效率**。

如果内存资源限制较大（部分嵌入式系统），可以使用解释执行节约内存，反之可以使用编译执行来提升效率。同时编译器的代码还能退回成解释器的代码。

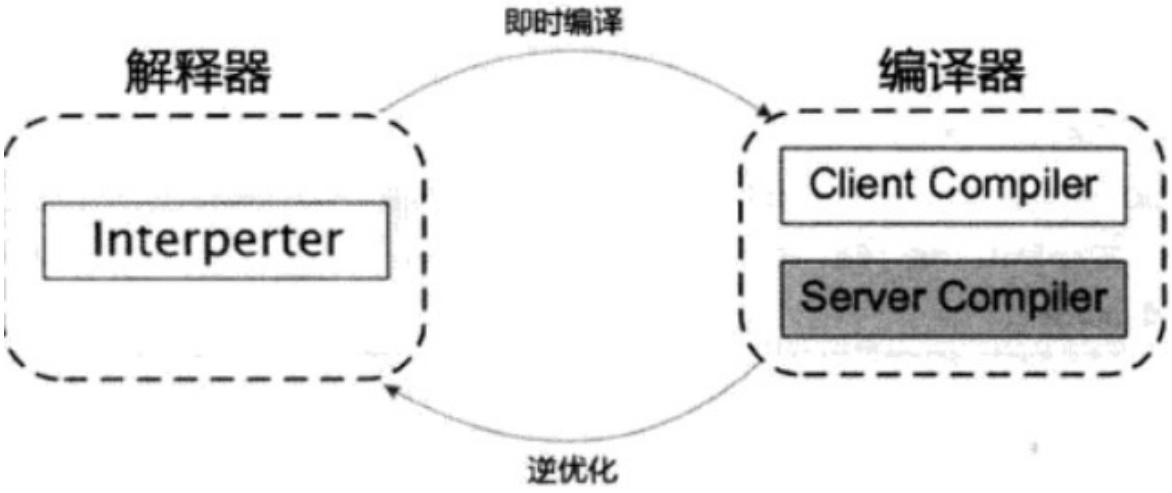

图 11-1　解释器与编译器的交互

1. 为什么要采用分层编译?

因为即时编译器编译本地代码需要占用程序运行时间，要编译出优化程度更高的代码，所花费的时间越长。

2. 分层编译器有哪些层次?

分层编译根据编译器编译、优化的规模和耗时，划分不同的编译层次，包括：

- **第0层**：程序解释执行，解释器不开启性能监控功能，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，如有必要加入性能监控的逻辑。
- **第2层**：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。

用Client Compiler和Server Compiler将会同时工作。用Client Compiler获取更高的编译速度，用Server Compiler获取更好的编译质量。

十九、编译对象与触发条件

1. 热点代码有哪些?

- 被多次调用的方法
- 被多次执行的循环体

2. 如何判断一段代码是不是热点代码?

要知道一段代码是不是热点代码，是不是需要触发即时编译，这个行为称为**热点探测**。主要有两种方法：

- **基于采样的热点探测**，虚拟机周期性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”。实现简单高效，但是很难精确确认一个方法的热度。
- **基于计数器的热点探测**，虚拟机会为每个方法建立计数器，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。

HotSpot虚拟机使用第二种，有两个计数器：

- 方法调用计数器
- 回边计数器（判断循环代码）

3. 方法调用计数器统计方法

统计的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**，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器的**热度衰减**，这个时间就被称为**半衰周期**。

二十、有哪些经典的优化技术（即时编译器）？

- 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

1. 公共子表达式消除

普遍应用于各种编译器的经典优化技术，它的含义是：

如果一个表达式 E 已经被计算过了，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 的这次出现就成了公共子表达式。没有必要重新计算，直接用结果代替 E 就可以了。

2. 数组边界检查消除

因为Java会自动检查数组越界，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，这无疑是一种性能负担。

如果数组访问发生在循环之中，并且使用循环变量来进行数组访问，如果编译器只要通过数据流分析就可以判定循环变量的取值范围永远在数组区间内，那么整个循环中就可以把数组的上下界检查消除掉，可以节省很多次的条件判断操作。

3. 方法内联

内联消除了方法调用的成本，还为其他优化手段建立良好的基础。

编译器在进行内联时，如果是非虚方法，那么直接内联。如果遇到虚方法，则会查询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，那么也可以内联，不过这种内联属于**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**。

如果程序的后续执行过程中，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现变化的类，那么内联优化的代码可以一直使用。否则需要抛弃掉已经编译的代码，退回到解释状态执行，或者重新进行编译。

4. 逃逸分析

逃逸分析的基本行为就是**分析对象动态作用域**：当一个对象在方法里面被定义后，它可能被外部方法所引用，这种行为被称为**方法逃逸**。被外部线程访问到，被称为**线程逃逸**。

二十一、如果对象不会逃逸到方法或线程外，可以做什么优化?

- **栈上分配**：一般对象都是分配在Java堆中的，对于各个线程都是共享和可见的，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。但是垃圾回收和整理都会耗时，如果一个对象不会逃逸出方法，可以让这个对象在栈上分配内存，对象所占用的内存空间就可以随着栈帧出栈而销毁。如果能使用栈上分配，那大量的对象会随着方法的结束而自动销毁，垃圾回收的压力会小很多。
- **同步消除**：线程同步本身就是很耗时的过程。如果逃逸分析能确定一个变量不会逃逸出线程，那这个变量的读写肯定就不会有竞争，同步措施就可以消除掉。

- **标量替换**：不创建这个对象，直接创建它的若干个被这个方法使用到的成员变量来替换。

二十二、Java与C/C++的编译器对比

1. 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。
2. Java语言虽然没有virtual关键字，但是使用虚方法的频率远大于C++，所以即时编译器进行优化时难度要远远大于C++的静态优化编译器。
3. 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，使得全局的优化难以进行，因为编译器无法看见程序的全貌，编译器不得不时刻注意并随着类型的变化，而在运行时撤销或重新进行一些优化。
4. Java语言对象的内存分配是在堆上，只有方法的局部变量才能在栈上分配。C++的对象有多种内存分配方式。

二十三、物理机如何处理并发问题?

运算任务，除了需要**处理器计算**之外，还需要与**内存交互**，如读取运算数据、存储运算结果等（不能仅靠寄存器来解决）。

计算机的存储设备和处理器的运算速度差了几个数量级，所以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），作为内存与处理器之间的缓冲：将运算需要的数据复制到缓存中，让运算快速运行。当运算结束后再从缓存同步回内存，这样处理器就无需等待缓慢的内存读写了。

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，但是引入了一个新的问题：**缓存一致性。**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时，可能导致各自的缓存数据不一致。

为了解决一致性的问题，需要各个处理器访问缓存时遵循**缓存一致性协议。**同时为了使得处理器充分被利用，处理器可能会对输出代码进行**乱序执行优化。**Java虚拟机的即时编译器也有类似的**指令重排序优化。**

二十四、讲讲什么情况下回出现内存溢出，内存泄漏?

内存泄漏的原因：对象是可达的(一直被引用)，但是对象不会被使用。

内存溢出的原因：

- 内存泄露导致堆栈内存不断增大，从而引发内存溢出。
- 大量的jar，class文件加载，装载类的空间不够，溢出
- 操作大量的对象导致堆内存空间已经用满了，溢出
- nio直接操作内存，内存过大导致溢出

解决：

- 查看程序是否存在内存泄漏的问题
- 设置参数加大空间
- 代码中是否存在死循环或循环产生过多重复的对象实体、
- 查看是否使用了nio直接操作内存。

二十五、JVM 年轻代到年老代的晋升过程的判断条件是什么呢?

1. 部分对象会在From和To区域中复制来复制去,**如此交换15次**(由JVM参数MaxTenuringThreshold决定,这个参数默认是15),最终如果还是存活,就存入到老年代。
2. 如果**对象的大小大于Eden的二分之一会直接分配在old**，如果old也分配不下，会做一次majorGC，如果小于eden的一半但是没有足够的空间，就进行minorgc也就是新生代GC。

3. minor gc后，survivor仍然放不下，则放到老年代
4. 动态年龄判断， 大于等于某个年龄的对象超过了survivor空间一半， 大于等于某个年龄的对象直接进入老年代

二十六、JVM 出现 fullGC 很频繁，怎么去线上排查问题

这题就依据full GC的触发条件来做：

- 如果有perm gen的话(jdk1.8就没了)，**要给perm gen分配空间，但没有足够的空间时**，会触发full gc。

  所以看看是不是perm gen区的值设置得太小了。

- `System.gc()`方法的调用

  这个一般没人去调用吧~~~

- 当**统计**得到的Minor GC晋升到旧生代的平均大小**大于老年代的剩余空间**，则会触发full gc(这就可以从多个角度上看了)

  是不是**频繁创建了大对象(也有可能eden区设置过小)**(大对象直接分配在老年代中，导致老年代空间不足--->从而频繁gc) 是不是**老年代的空间设置过小了**(Minor GC几个对象就大于老年代的剩余空间了)

二十七、类的实例化顺序

1. 父类静态成员和静态初始化块， 按在代码中出现的顺序依次执行
2. 子类静态成员和静态初始化块， 按在代码中出现的顺序依次执行
3. 父类实例成员和实例初始化块， 按在代码中出现的顺序依次执行
4. 父类构造方法
5. 子类实例成员和实例初始化块， 按在代码中出现的顺序依次执行
6. 子类构造方法

二十八、JVM 中一次完整的 GC 流程（从 ygc 到 fgc）是怎样的

- YGC： **对新生代堆进行gc。**频率比较高，因为大部分对象的存活寿命较短，在新生代里被回收。性能耗费较小。
- FGC： **全堆范围的gc。**默认堆空间使用到达80%(可调整)的时候会触发fgc。以我们生产环境为例，一般比较少会触发fgc，有时10天或一周左右会有一次。

什么时候执行YGC和FGC

- eden空间不足,执行 young gc
- old空间不足，perm空间不足，调用方法`System.gc()`， ygc时的悲观策略, dump live的内存信息时(jmap –dump:live)，都会执行full gc