

# Java 虚拟机面试题全面解析

## 面试题

本文固定链接：<https://www.zybuluo.com/Yano/note/321063>

LeetCode题解：<https://github.com/LjyYano/LeetCode>

我的博客：[http://blog.csdn.net/yano\\_nankai](http://blog.csdn.net/yano_nankai)

周志明著的《深入理解 Java 虚拟机》的干货~如有错误，欢迎指出 O(n\_n)O 转载请保留以上信息。

---

## Java 虚拟机面试题全面解析

JDK 是什么？

JRE 是什么？

Java历史版本的特性？

- Java Version SE 5.0

- Java Version SE 6

- Java Version SE 7

- Java 8

运行时数据区域包括哪些？

- 程序计数器（线程私有）

- Java 虚拟机栈（线程私有）

- 本地方法栈（线程私有）

- Java 堆（线程共享）

- 方法区（线程共享）

- 运行时常量池

Java 中对象访问是如何进行的？

如何判断对象是否“死去”？

- 什么是引用计数法？

- 引用计数法的缺点？

- 什么是根搜索算法？

Java 的4种引用方式？

强引用

软引用

弱引用

虚引用

有哪些垃圾收集算法？

标记-清除算法 ( Mark-Sweep )

什么是标记-清除算法？

有什么缺点？

复制算法 ( Copying ) - 新生代

优点？

缺点？

应用？

标记-整理算法 ( Mark-Compact ) - 老年代

分代收集算法

Minor GC 和 Full GC有什么区别？

Java 内存

为什么要将堆内存分区？

堆内存分为哪几块？

分代收集算法

内存分配有哪些原则？

Young Generation Space ( 采用复制算法 )

Tenure Generation Space ( 采用标记-整理算法 )

Permanent Space

Class文件

Java虚拟机的平台无关性

Class文件的组成？

魔数与Class文件的版本

类加载器

类加载器的作用是什么？

类加载器有哪些？

类加载机制

什么是双亲委派模型？

为什么要使用双亲委派模型，组织类加载器之间的关系？

什么是类加载机制？

虚拟机和物理机的区别是什么？

运行时栈帧结构

Java 方法调用

什么是方法调用？

Java的方法调用，有什么特殊之处？

Java虚拟机调用字节码指令有哪些？

虚拟机是如何执行方法里面的字节码指令的？

解释执行

基于栈的指令集和基于寄存器的指令集

什么是基于栈的指令集？

什么是基于寄存器的指令集？

基于栈的指令集的优缺点？

Javac编译过程分为哪些步骤？

什么是即时编译器？

解释器和编译器

为什么要采用分层编译？

分层编译器有哪些层次？

编译对象与触发条件

热点代码有哪些？

如何判断一段代码是不是热点代码？

HotSpot虚拟机使用第二种，有两个计数器：

方法调用计数器统计方法

有哪些经典的优化技术（即时编译器）？

公共子表达式消除

数组边界检查消除

方法内联

逃逸分析

如果对象不会逃逸到方法或线程外，可以做什么优化？

Java与C/C++的编译器对比

物理机如何处理并发问题？

Java 内存模型

什么是Java内存模型？

Java内存模型的目标？

主内存与工作内存

内存间的交互操作

原子性、可见性、有序性

volatile

什么是volatile？

为什么基于volatile变量的运算在并发下不一定是安全的？

为什么使用volatile？

并发与线程

并发与线程的关系？

什么是线程？

实现线程有哪些方式？

Java线程的实现

Java线程调度

什么是线程调度？

线程调度有哪些方法？

线程安全的定义？

Java语言操作的共享数据，包括哪些？

不可变

如何实现线程安全？

阻塞同步（互斥同步）

非阻塞同步

锁优化是在JDK的那个版本？

为什么要提出自旋锁？

自旋锁的原理？

自旋的缺点？

什么是自适应自旋？

锁消除

锁粗化

轻量级锁

偏向锁

# JDK 是什么？

JDK 是用于支持 Java 程序开发的最小环境。

1. Java 程序设计语言
2. Java 虚拟机
3. Java API类库

# JRE 是什么？

JRE 是支持 Java 程序运行的标准环境。

1. Java SE API 子集
2. Java 虚拟机

# Java历史版本的特性？

## Java Version SE 5.0

- 引入泛型；
- 增强循环，可以使用迭代方式；
- 自动装箱与自动拆箱；
- 类型安全的枚举；
- 可变参数；
- 静态引入；
- 元数据（注解）；
- 引入Instrumentation。

## Java Version SE 6

- 支持脚本语言；
- 引入JDBC 4.0 API；
- 引入Java Compiler API；
- 可插拔注解；
- 增加对Native PKI(Public Key Infrastructure)、Java GSS(Generic Security Service)、Kerberos和LDAP(Lightweight Directory Access Protocol)的支持；
- 继承Web Services；
- 做了很多优化。

## Java Version SE 7

- switch语句块中允许以字符串作为分支条件；
- 在创建泛型对象时应用类型推断；
- 在一个语句块中捕获多种异常；
- 支持动态语言；
- 支持try-with-resources；
- 引入Java NIO.2开发包；
- 数值类型可以用2进制字符串表示，并且可以在字符串表示中添加下划线；
- 钻石型语法；
- null值的自动处理。

## Java 8

- 函数式接口
- Lambda表达式
- 接口的增强

## 运行时数据区域包括哪些？

1. 程序计数器
2. Java 虚拟机栈
3. 本地方法栈

4. Java 堆
5. 方法区
6. 运行时常量池
7. 直接内存

## 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。

字节码解释器工作时就是通过改变这个计数器的值来取下一条需要执行的字节码指令

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。

## 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

局部变量表所需的内存空间在编译期间完成分配，运行期间不会改变局部变量表的大小。

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

## 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。

## 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。

如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出 OutOfMemoryError

## 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。

和 Java 堆一样，不需要连续的内存，可以选择固定的大小，更可以选择不实现垃圾收集。

当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError

## 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。

当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError

## Java 中对象访问是如何进行的？

直接内存：不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域。但被频繁使用，也可能导致 OutOfMemoryError  
例子：NIO  
直接内存忽略也可能导致各个内存区域大于物理内存，将抛出 OutOfMemoryError



对象内存布局：

#### 1、对象头

第一部分：存储对象自身的运行时的数据(hashcode, GC分代年龄, 锁状态标志, 线程持有锁, 偏向线程ID, 偏向时间戳)第二部分：类型指针, 对象指向其类元数据的指针, 若是Java对象数组, 对象头还有一块用于记录数组长度的数据

#### 2、实例数据

代码中所定义的各种类型的字段内容

#### 3、对齐填充

不必然存在, 起占位符作用

```
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。

## 如何判断对象是否“死去”？

1. 引用计数法
2. 根搜索算法

String.intern()是一个native方法, 作用为: 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对象的引用, 否则将String对象包含的字符串添加到常量池中, 并返回此对象的引用。

## 什么是引用计数法？

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。

### 引用计数法的缺点？

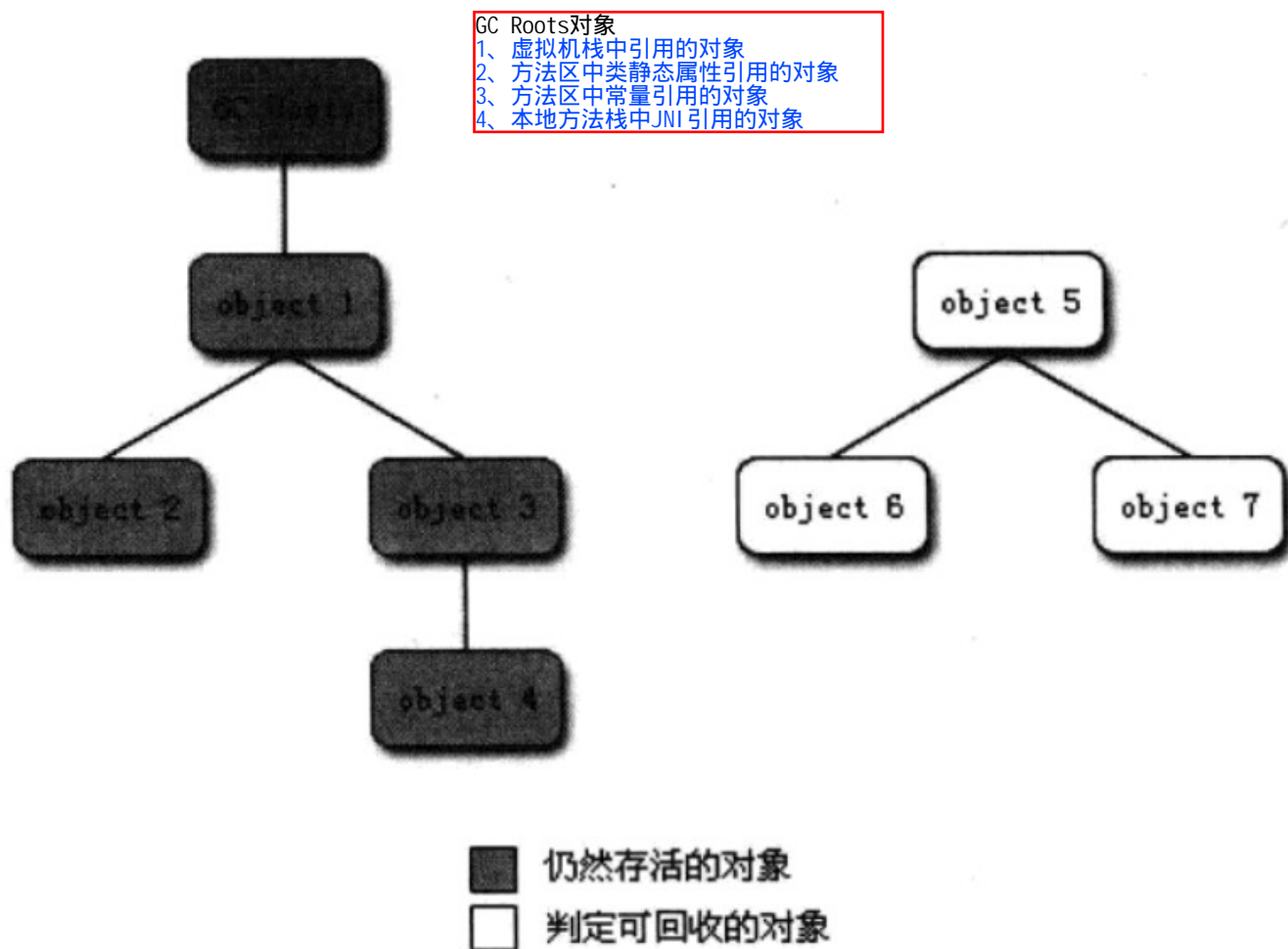
很难解决对象之间的循环引用问题。

互相循环引用问题

## 什么是根搜索算法？

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。



## Java 的4种引用方式？

在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为

1. 强引用 Strong Reference
2. 软引用 Soft Reference
3. 弱引用 Weak Reference
4. 虚引用 Phantom Reference

### 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。

## 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，有在系统将要**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。

## 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。

## 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。

## 有哪些垃圾收集算法？

1. **标记-清除算法**
2. **复制算法**
3. **标记-整理算法**
4. **分代收集算法**

### 标记-清除算法 ( Mark-Sweep )

## 什么是标记-清除算法？

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

## 有什么缺点？

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，不得不提前出发另一次垃圾收集动作。

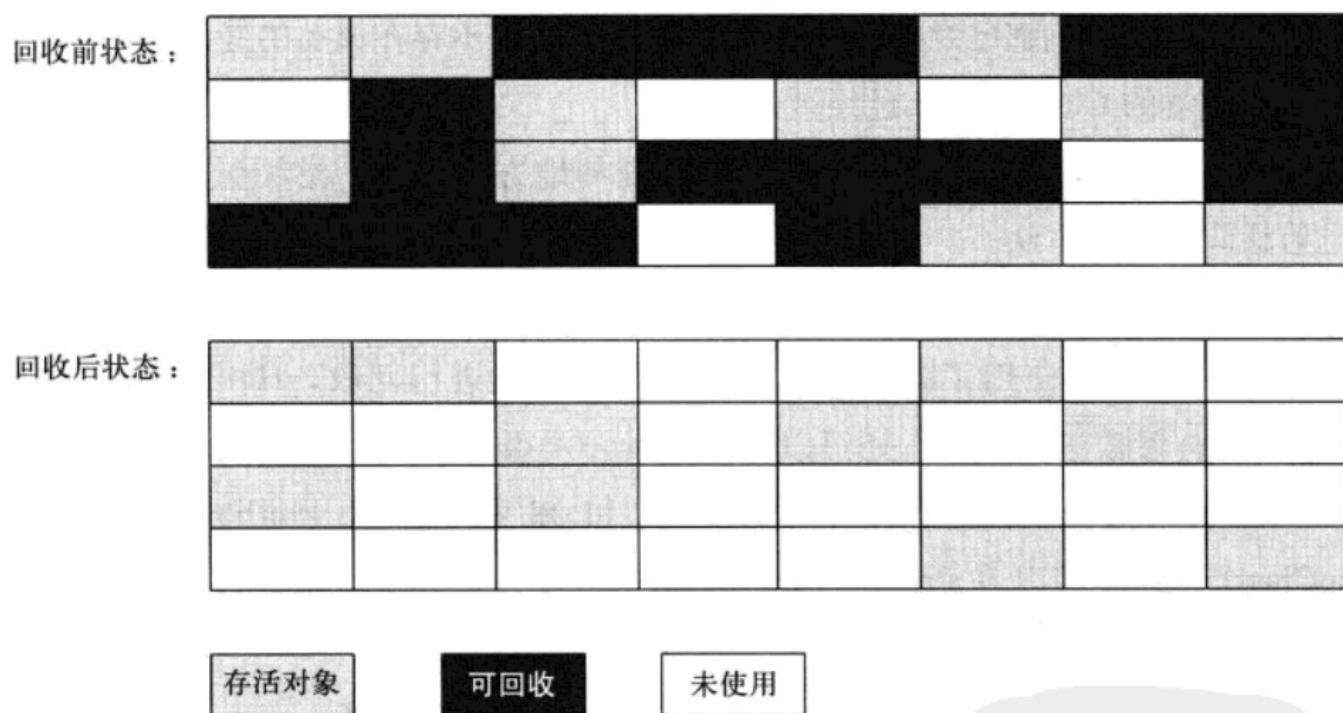


图 3-2 “标记 - 清除”算法示意图

## 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。

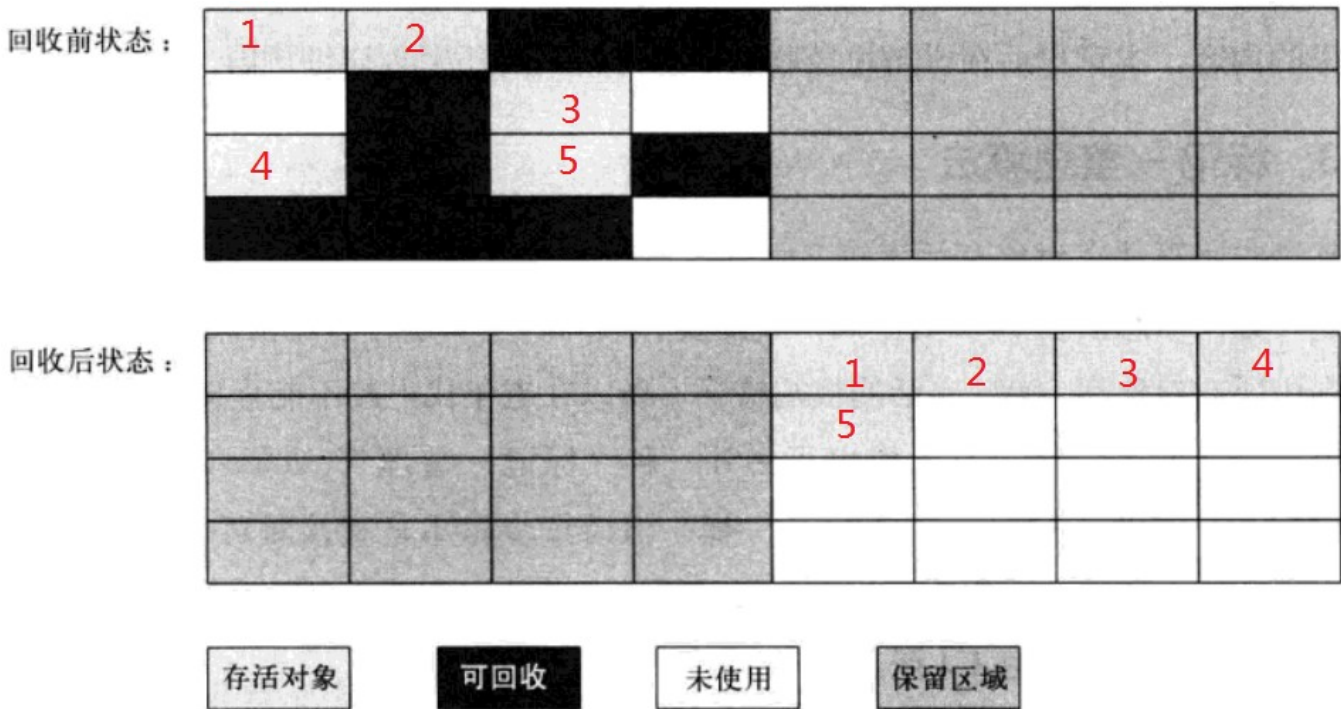


图 3-3 复制算法示意图

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。

若10%的survivor空间不足时，则需要老年代来担保

# 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。

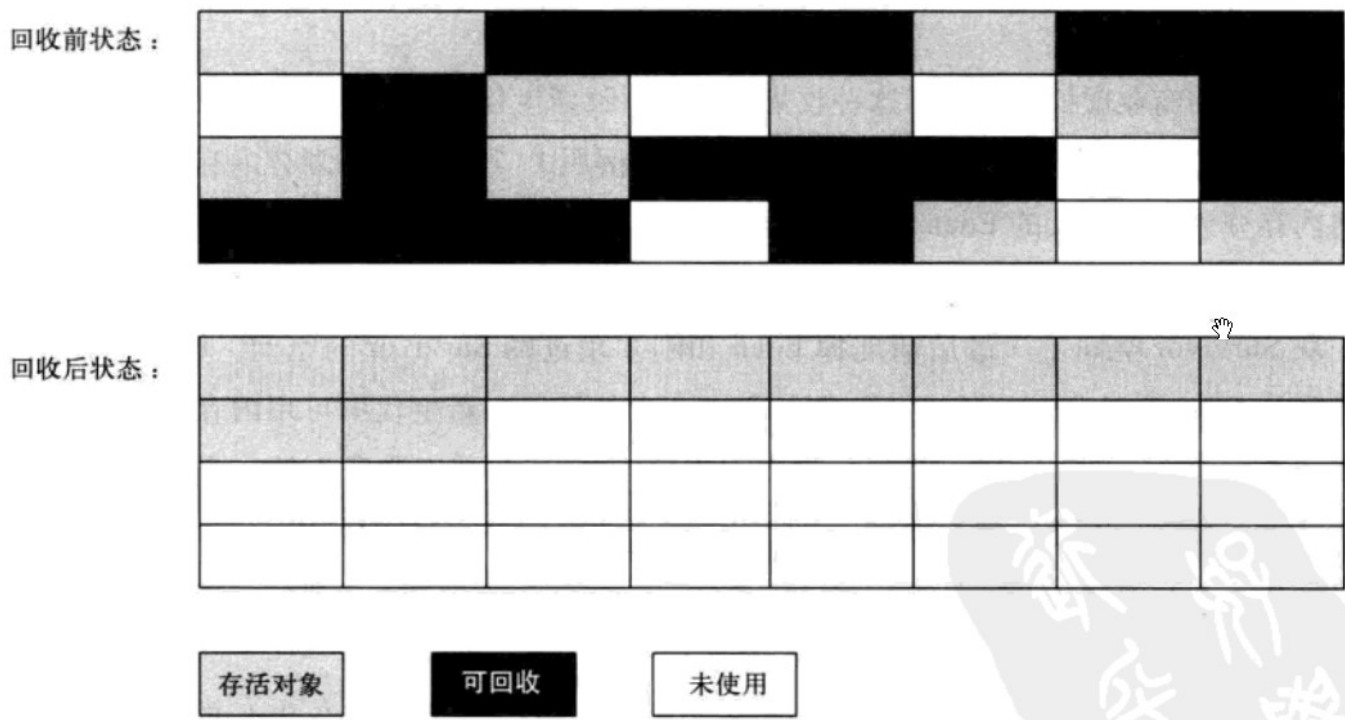


图 3-4 “标记 - 整理” 算法示意图

## 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。

## 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。

**枚举根节点**：对象引用关系不再变化的时候“Stop the world”  
**安全点**：特定位置才记录OopMap数据结构的信息(对象引用的地址)，一般在指令序列发生复用的地方。然后在GC发生时，让所有线程跑到安全点。  
**安全区域**：一段代码片段之中，引用关系不会发生变化。

## Java 内存

### 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。

### 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（也称旧生代）
3. Permanent Space 永久存储区

### 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

若不够，发动Minor GC

启动本地线程分配缓冲的话，将按线程优先在TLAB上分配



并行：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：用户线程和垃圾收集线程同时执行(但不一定是并行的，可能会交替执行)，用户程序在继续执行，而垃圾收集程序运行于另一个CPU上

## Young Generation Space (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 Survivor Space，如果对象仍然存活，则复制到B Survivor Space，如果B Survivor Space已经满，则复制 Old Gen
- 扫描A Survivor 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 Survivor 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 Survivor Space。

## Tenure Generation Space (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。

## Permanent Space

存储不变的类定义、字节码和常量等。

## Class文件

## Java虚拟机的平台无关性

Serial收集器：单线程收集器，stop the world  
Serial Old收集器是老年代版本  
ParNew收集器：Serial收集器的多线程版本  
PS(Parallel Scavenge)收集器：控制吞吐量  
ParOld是PS的老年代版本

CMS收集器：以获取最短回收时间为目标(对CPU资源敏感、难以回收浮动垃圾、基于标记-清除，有大量空间碎片)  
1、初始标记: GC Root关联 快  
2、并发标记: GC Root tracing 最慢  
3、重新标记: 消除继续运行导致的引用变动 较慢  
4、并发清除

G1收集器：(并行与并发、分代收集、整体基于标记-整理，无空间碎片，可预测的停顿)  
1、初始标记: GC Root关联 快  
2、并发标记: GC Root tracing 最慢  
3、最终标记: 消除继续运行导致的引用变动 较慢  
4、筛选回收

Full GC 会stop the world



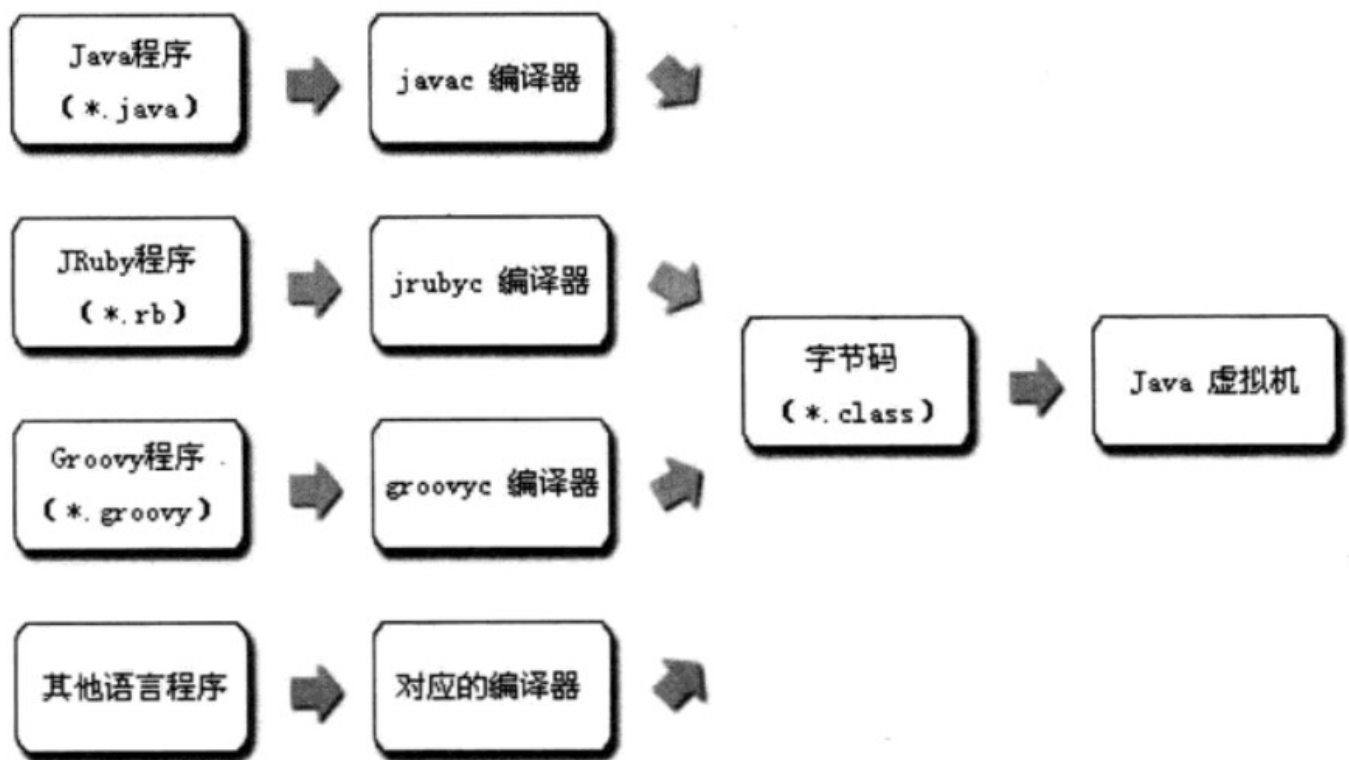


图 6-1 Java 虚拟机提供的语言无关性

## 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。

## 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。

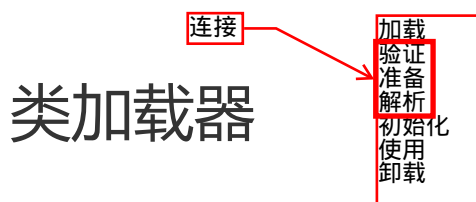
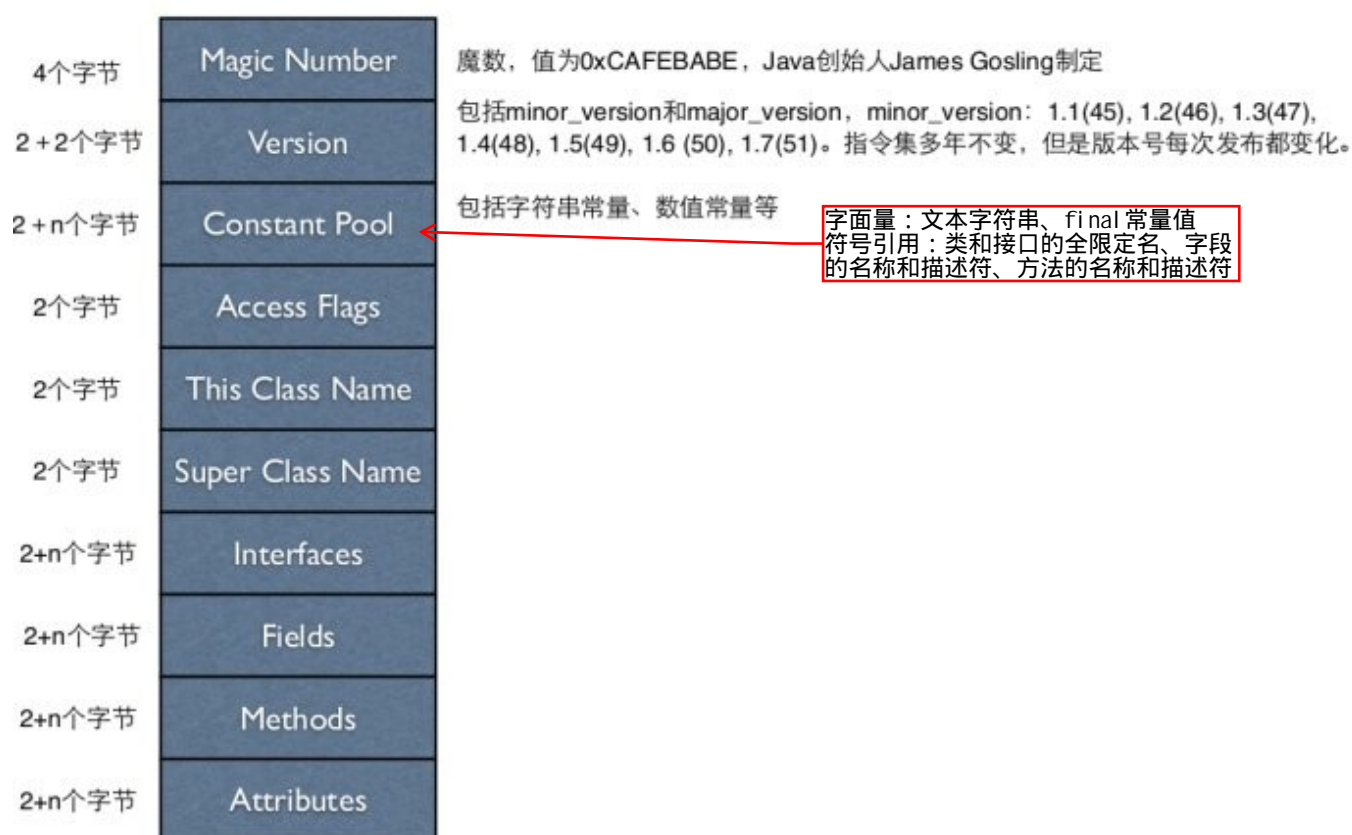
接下来是Class文件的**版本号**：第5,6字节是次版本号（Minor Version），第7,8字节是主版本号（Major Version）。

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**。

- 33 : JDK1.7
- 32 : JDK1.6
- 31 : JDK1.5
- 30 : JDK1.4
- 2F : JDK1.3



类加载器的作用是什么？

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。

## 类加载器有哪些？

1. **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。
2. **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。
3. **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。

## 类加载机制

### 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。

### 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。

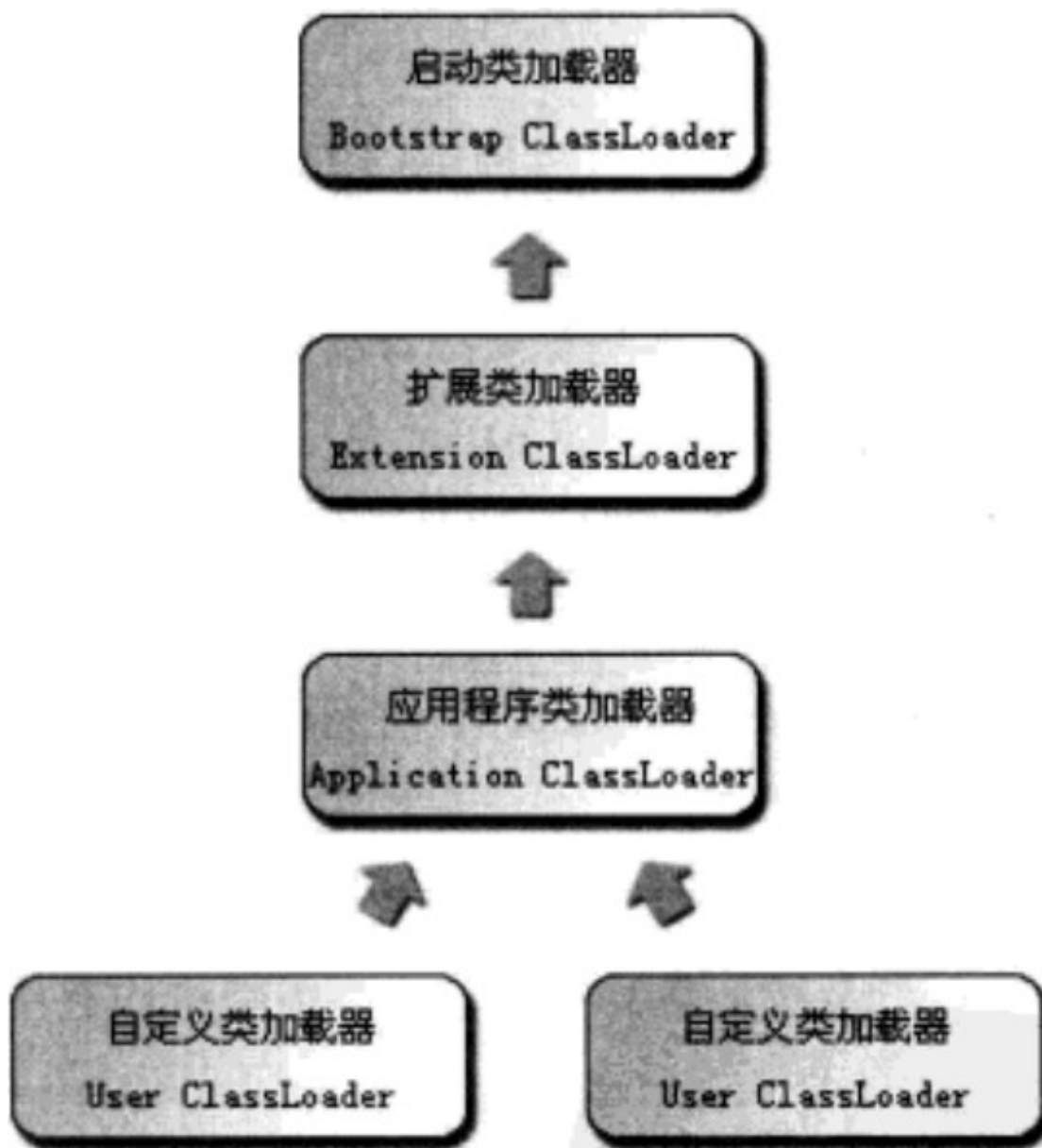


图 7-2 类加载器双亲委派模型

什么是**类加载机制**？

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

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。

## 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。

## 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。

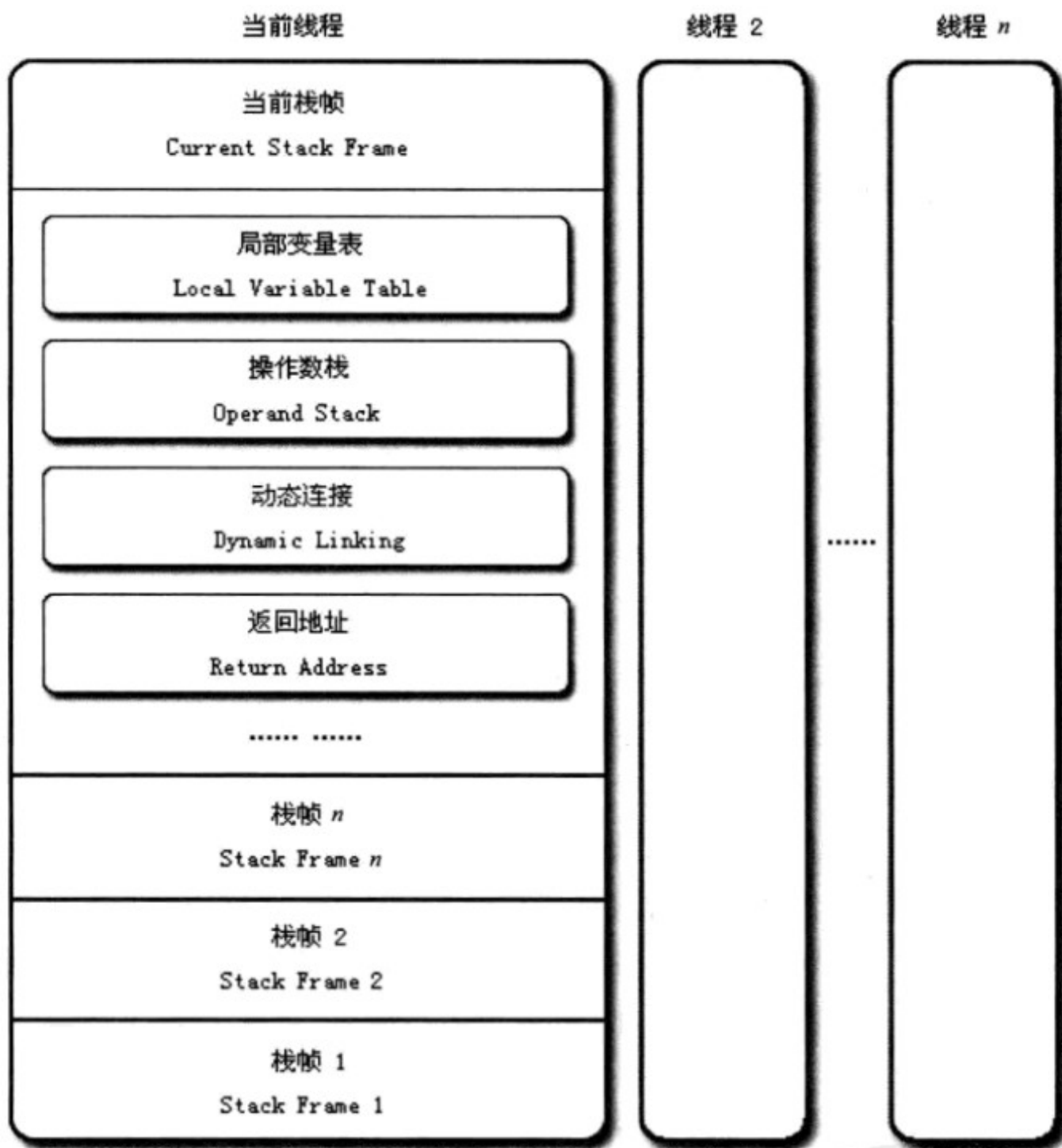


图 8-1 栈帧的概念结构

## Java 方法调用

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。

## 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。

## 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

## 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）

## 解释执行

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。

**自动装箱：**  
Java虚拟机自动调用封装类型的valueOf()方法  
**自动拆箱：**  
Java虚拟机会自动调用封装类型的intValue()方法等  
Byte、Short、Integer、Long、Char这几个装箱类的  
valueOf()方法是以128位分界线做了缓存的，假如是  
128以下且-128以上的值是会取缓存里面的引用的  
Float和Double则不会

# 基于栈的指令集和基于寄存器的指令集

## 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。

## 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。

## 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熟练多

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。

## 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<sup>①</sup>

## 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）。

## 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。

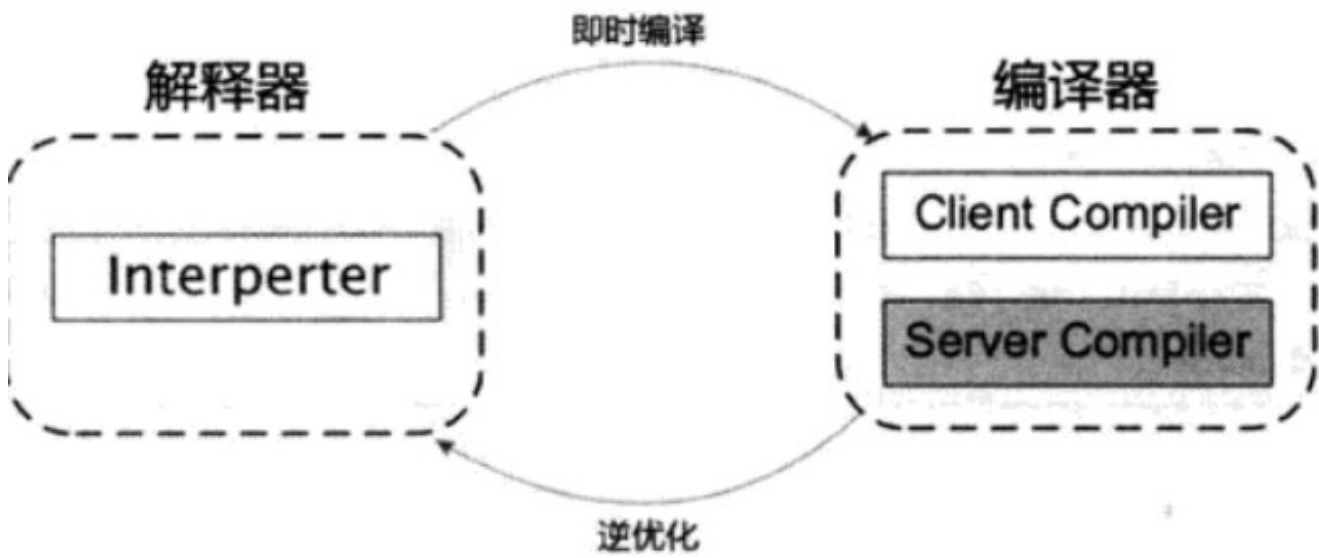


图 11-1 解释器与编译器的交互

## 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。

## 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，可出发第1层编译。
- 第1层：也成为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：也成为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。

## 编译对象与触发条件

热点代码有哪些？

- 被多次调用的方法
- 被多次执行的循环体

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）

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**。

## 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**

### 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。

### 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。

### 方法内联

**方法内联：**  
把目标方法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

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。

## 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**。

## 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。

## 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。

## 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。

## Java 内存模型

### 什么是Java内存模型？

Java虚拟机的规范，用来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各个平台下都能达到一致的并发效果。

### Java内存模型的目标？

定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出这样的底层细节。此处的变量包括实例字段、静态字段和构成数组对象的元素，但是不包括局部变量和方法

法参数，因为这些是线程私有的，不会被共享，所以不存在竞争问题。

## 主内存与工作内存

所有的变量都存储在主内存，每条线程还有自己的工作内存，保存了被该线程使用到的变量的主内存副本拷贝。线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，不能直接读写主内存的变量。不同的线程之间也无法直接访问对方工作内存的变量，线程间变量值的传递需要通过主内存。

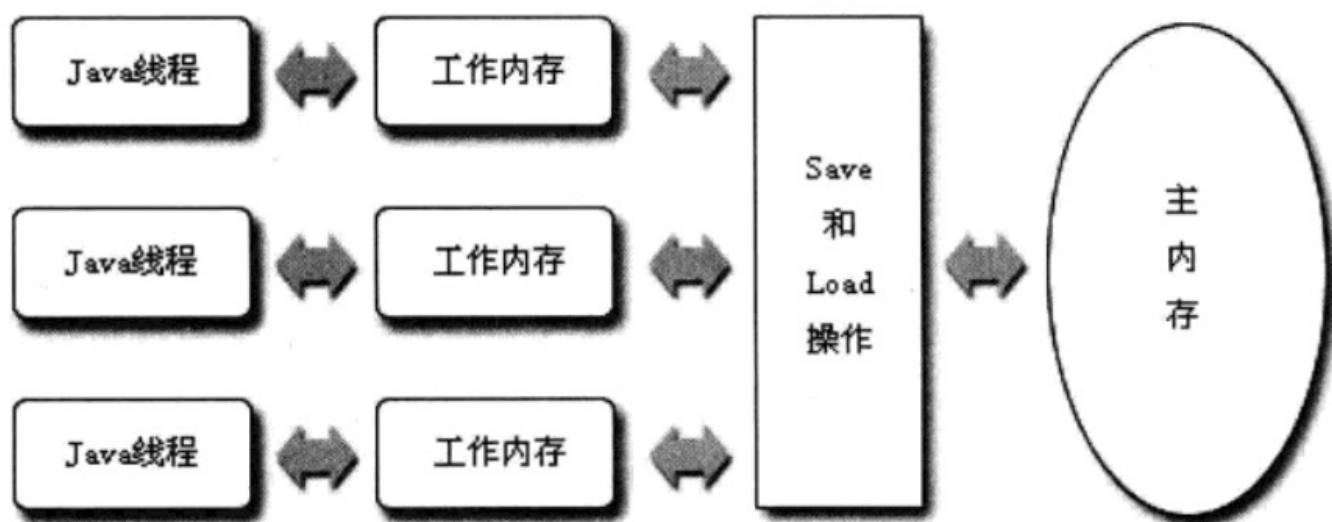


图 12-2 线程、主内存、工作内存三者的交互关系（请与图 12-1 对比）


## 内存间的交互操作

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，Java内存模型定义了8种操作：



lock ( 锁定 )	主内存	把一个变量标识为一条线程独占的状态
unlock ( 解锁 )	主内存	把一个处于锁定状态的变量释放出来，释放之后的变量才能被其他线程锁定
read ( 读取 )	主内存	把一个变量值从主内存传输到线程的工作内存，以便load
load ( 载入 )	工作内存	把read操作从主内存得到的变量值放入工作内存的变量副本中
use ( 使用 )	工作内存	将工作内存变量值传递给执行引擎
assign ( 赋值 )	工作内存	将执行引擎值传递给工作内存变量值
store ( 存储 )	工作内存	把工作内存的变量值传送到主内存，以便write
write ( 写入 )	主内存	把store操作从工作内存得到的变量的值，放入主内存的变量中

## 原子性、可见性、有序性

- **原子性**：对**基本数据类型**的访问和读写是**具备原子性**的。对于**更大范围的原子性保证**，可以使用字节码指令monitorenter和monitorexit来隐式使用lock和unlock操作。这两个字节码指令反映到Java代码中就是**同步块**——**synchronized**关键字。因此synchronized块之间的操作也具有原子性。
- **可见性**：当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**。Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取之前从主内存刷新变量值来实现可见性的。volatile的特殊规则保证了新值能够立即同步到主内存，每次使用前立即从主内存刷新。**synchronized**和**final**也能实现可见性。final修饰的字段在构造器中一旦被初始化完成，并且构造器没有把this的引用传递出去，那么其他线程中就能看见final字段的值。  
 this引用逃逸时一件很危险的事情，其他线程就可能通过这个引用访问到了"初始化了一半"的对象
- **有序性**：Java程序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（线程内表现为串行的语义）；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（指令重排序和工作内存与主内存同步延迟线性）。



# volatile

## 什么是volatile？

关键字volatile是Java虚拟机提供的**轻量级的同步机制**。当一个变量被定义成volatile之后，具备两种特性：

1. **保证此变量对所有线程的可见性**。当一条线程修改了这个变量的值，新值对于其他线程是可以立即得知的。而普通变量做不到这一点。
2. **禁止指令重排序优化**。普通变量仅仅能保证在该方法执行过程中，得到正确结果，但是不保证程序代码的执行顺序。

## 为什么基于volatile变量的运算在并发下不一定是安全的？

volatile变量在各个线程的工作内存，不存在一致性问题（**各个线程的工作内存中volatile变量，每次使用前都要刷新到主内存**）。但是Java里面的运算并非原子操作，导致volatile变量的运算在并发下一样是不安全的。

## 为什么使用volatile？

在某些情况下，volatile同步机制的性能要优于锁（synchronized关键字），但是由于虚拟机对锁实行的许多消除和优化，所以并不是很快。

volatile变量读操作的性能消耗与普通变量几乎没有差别，但是**写操作则可能慢**一些，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**。

# 并发与线程

## 并发与线程的关系？

并发不一定要依赖多线程，PHP中有多进程并发。但是Java里面的并发是多线程的。

## 什么是线程？

线程是比进程更轻量级的调度执行单位。线程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（线程是CPU调度的最基本单位）。

## 实现线程有哪些方式？

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

## Java线程的实现

操作系统支持怎样的线程模型，在很大程度上就决定了Java虚拟机的线程是怎样映射的。

# Java线程调度

## 什么是线程调度？

线程调度是系统为线程分配处理器使用权的过程。

## 线程调度有哪些方法？

- 协同式线程调度：实现简单，没有线程同步的问题。但是线程执行时间不可控，容易系统崩溃。
- 抢占式线程调度：每个线程由系统来分配执行时间，不会有线程导致整个进程阻塞的问题。

虽然Java线程调度是系统自动完成的，但是我们可以建议系统给某些线程多分配点时间——设置线程优先级。Java语言有10个级别的线程优先级，优先级越高的线程，越容易被系统选择执

行。

但是并不能完全依靠线程优先级。因为Java的线程是被映射到系统的原生线程上，所以线程调度最终还是由操作系统说了算。如Windows中只有7种优先级，所以Java不得不出现在几个优先级相同的情况。同时优先级可能会被系统自行改变。Windows系统中存在一个“**优先级推进器**”，当系统发现一个线程执行特别勤奋，可能会越过线程优先级为它分配执行时间。

## 线程安全的定义？

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，或者在调用方法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，那这个对象就是线程安全的。

## Java语言操作的共享数据，包括哪些？

- 不可变
- 绝对线程安全
- 相对线程安全
- 线程兼容
- 线程对立

### 不可变

在Java语言里，不可变的对象一定是线程安全的，只要一个不可变的对象被正确构建出来，那其外部的可见状态永远也不会改变，永远也不会多个线程中处于不一致的状态。

## 如何实现线程安全？

虚拟机提供了**同步**和**锁**机制。

- 阻塞同步（互斥同步）
- 非阻塞同步

## 阻塞同步（互斥同步）

**互斥**是实现同步的一种手段，**临界区**、**互斥量**和**信号量**都是主要的互斥实现方式。Java中最基本的同步手段就是**synchronized**关键字，其编译后会在同步块的前后分别形成**monitorenter**和**monitorexit**两个字节码指令。这两个字节码都需要一个Reference类型的参数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，那么这个对象就是Reference；如果没有明确指定，那就根据synchronized修饰的是实例方法还是类方法，去获取对应的对象实例或Class对象作为锁对象。在执行monitorenter指令时，首先要尝试获取对象的锁。

- 如果这个对象没有锁定，或者当前线程已经拥有了这个对象的锁，把锁的计数器+1；当执行monitorexit指令时将锁计数器-1。当计数器为0时，锁就被释放了。
- 如果获取对象失败了，那当前线程就要阻塞等待，知道对象锁被另外一个线程释放为止。

除了synchronized之外，还可以使用java.util.concurrent包中的**重入锁**（**ReentrantLock**）来实现同步。ReentrantLock比synchronized增加了高级功能：等待可中断、可实现公平锁、锁可以绑定多个条件。

**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，对处理执行时间非常长的同步块很有用。

**公平锁**：多个线程在等待同一个锁时，必须**按照申请锁的时间顺序**来依次获得锁。**synchronized**中的锁是**非公平**的。

## 非阻塞同步

**互斥同步**最大的问题，就是进行**线程阻塞和唤醒所带来的性能问题**，是一种**悲观的并发策略**。总是认为只要不去做正确的同步措施（加锁），那就肯定会出问题，无论共享数据是否真的会出现竞争，它都要进行加锁、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。

随着硬件指令集的发展，我们可以使用**基于冲突检测的乐观并发策略**。**先进行操作**，如果没有其他线程征用数据，那操作就成功了；如果共享数据有征用，产生了冲突，那就再进行其他的补偿措施。这种乐观的并发策略的许多实现**不需要线程挂起**，所以被称为**非阻塞同步**。

## 锁优化是在JDK的那个版本？

JDK1.6的一个重要主题，就是**高效并发**。HotSpot虚拟机开发团队在这个版本上，实现了各种锁优化：

- 适应性自旋
- 锁消除
- 锁粗化
- 轻量级锁
- 偏向锁

## 为什么要提出自旋锁？

互斥同步对性能最大的影响是阻塞的实现，**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带来很大压力。同时很多应用共享数据的锁定状态，只会**持续很短的一段时间**，为了这段时间去挂起和恢复线程并不值得。先不挂起线程，等一会儿。

## 自旋锁的原理？

如果物理机器有一个以上的处理器，能让两个或以上的线程同时并行执行，让后面请求锁的线程稍等一会，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。**为了让线程等待，我们只需让线程执行一个忙循环（自旋）。

## 自旋的缺点？

自旋等待本身虽然避免了线程切换的开销，但它要占用处理器时间。所以如果锁被占用的时间很短，自旋等待的效果就非常好；如果时间很长，那么自旋的线程只会白白消耗处理器的资

源。所以自旋等待的时间要有一定的限度，如果自旋超过了限定的次数仍然没有成功获得锁，那就应该使用传统的方式挂起线程了。

## 什么是自适应自旋？

自旋的时间不固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

- 如果一个锁对象，自旋等待刚刚成功获得锁，并且持有锁的线程正在运行，那么虚拟机认为这次自旋仍然可能成功，进而运行自旋等待更长的时间。
- 如果对于某个锁，自旋很少成功，那在以后要获取这个锁，可能省略掉自旋过程，以免浪费处理器资源。

有了自适应自旋，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就会越来越准确，虚拟机也会越来越聪明。

## 锁消除

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但被检测到不可能存在共享数据竞争的锁进行消除。主要根据逃逸分析。

程序员怎么会在明知道不存在数据竞争的情况下使用同步呢？很多不是程序员自己加入的。

## 锁粗化

原则上，同步块的作用范围要尽量小。但是如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作在循环体内，频繁地进行互斥同步操作也会导致不必要的性能损耗。

锁粗化就是增大锁的作用域。

## 轻量级锁

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。

## 偏向锁

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。即在无竞争的情况下，**把整个同步都消除掉**。这个锁会偏向于第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要同步。