# 第2章 Java内存区域与内存溢出异常

2.1 概述

2.2 运行时数据区域

2.3 HotSpot虚拟机对象探秘

2.4 实战：OutOfMemoryError异常

第四部分 程序编译与代码优化

# 第10章 前端编译与优化

1. 什么是编译器(编译过程)的前端和后端？
   1. 前端——把\*.java文件转变成\*.class文件。Javac就是做这个的。
   2. 后端——把Class文件转换成与本地基础设施（硬件指令集、操作系统）相关的二进制机器码
2. 以javac为例，**编译的过程**为三步：
   1. 解析与填充符号表
      1. 词法、语法分析 -> 字符流转为标记集合，生成语法树
      2. 填充符号表 -> 产生符号地址和符号信息
   2. 插入式注解处理器的注解处理过程 \*在本章实战部分，我还没看
   3. 分析与字节码的生成过程，包括：
      1. **标注检查**——检查静态信息（如：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匹配）
      2. **数据流及控制流分析**——检查程序动态运行过程（如：局部变量在使用前是否已被赋值、方法的每条路径是否都有返回值、是否所有受检异常都被处理）
      3. **解语法糖**——将语法糖还原成原有形式（也是代码）
      4. **字节码生成**
3. 谈一谈Java语法糖
   1. 泛型
   2. 自动装箱、自动拆箱、遍历循环
   3. 条件编译

# 第11章 后端编译与优化

1. **即时编译器**JIT（Just In Time）
   1. 概述/总结：Java程序最初都是通过解释器Interpreter进行解释执行的，当虚拟机发现某个方法或代码块的运行非常频繁时，就会把这些代码认定为“热点代码”（Hot Spot Code），在运行时，*虚拟机*将会把这些代码编译成**本地机器码**，并以各种手段尽可能地进行**代码优化**，运行时完成这个任务的后端编译器被称为即时编译器。
   2. 解释器&编译器：
      1. 当前主流的JVM,如HotSpot，内部都同时包含解释器与编译器。
      2. 当程序需要迅速启动和执行的时候，解释器省去了编译的时间，立即运行；随着时间的推迟，编译器逐渐发挥作用，越来越多的代码编译成本地代码，减少解释器的中间损耗。
      3. 应用场景：当程序运行环境中内存资源限制较大，可以使用解释执行节约内存，反之使用编译执行节约内存。
   3. 编译对象与触发条件
      1. “热点代码”=被多次调用的方法+被多次执行的循环体
      2. 对于这两种情况，编译的都是整个方法体，而不是单独的循环体
2. **编译器优化**技术
   1. **方法内联**——最重要的优化手段。
      1. 语义上，内联是把目标方法上的代码原封不动地“复制”到发起调用的方法之中，避免发生真正的方法调用。
      2. 实现上，很难for Java。Java默认的方法是虚方法（for多态），动态的，难以在编译器进行解析。解决的看不懂。
   2. **逃逸分析**
      1. 什么是“逃逸”？
         1. 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，e.g.作为调用参数传递到其他方法中 -> **方法逃逸**
         2. Even more，被外部线程访问到，e.g.赋值给可以在其他线程中访问到的实例变量 -> **线程逃逸**
      2. 如果能证明一个对象不会逃逸到方法或者线程之外，或者逃逸程度比较低（只方法无线程），那么可以进行以下优化：
         1. 栈上分配——相比GC堆回收对象，在栈上分配的对象可以随着方法的结束而自动销毁。支持方法逃逸，不支持线程逃逸。
         2. 标量替换——标量is一个数据无法再分解成更小的数据（int、long等数值类型以及reference类型）。若对象不逃逸，将它（聚合量）拆散成若干个成员变量，有利于栈上分配和进一步优化。
         3. 同步消除——如果不逃逸线程，直接取消对这个变量实施的同步措施。
   3. **公共子表达式消除**：一个计算结果，重复使用
   4. **数组边界检查消除**
      1. Bg，相较于C/C++,Java对于数组的访问并非裸指针操作，而是会自动进行上下界的范围检查
      2. 但是，边界数组检查是不是在运行期间进行“一次不落”的进行是一件可以“商量”的事 -> 数组访问常发生在循环中，通过数据流分析就可以完成 -> 把运行期检查提前到编译期检查

第五部分 高效并发

# 第12章 Java内存模型与线程

12.1 概述

12.2 硬件的效率与一致性

12.3 Java内存模型

12.4 Java与线程

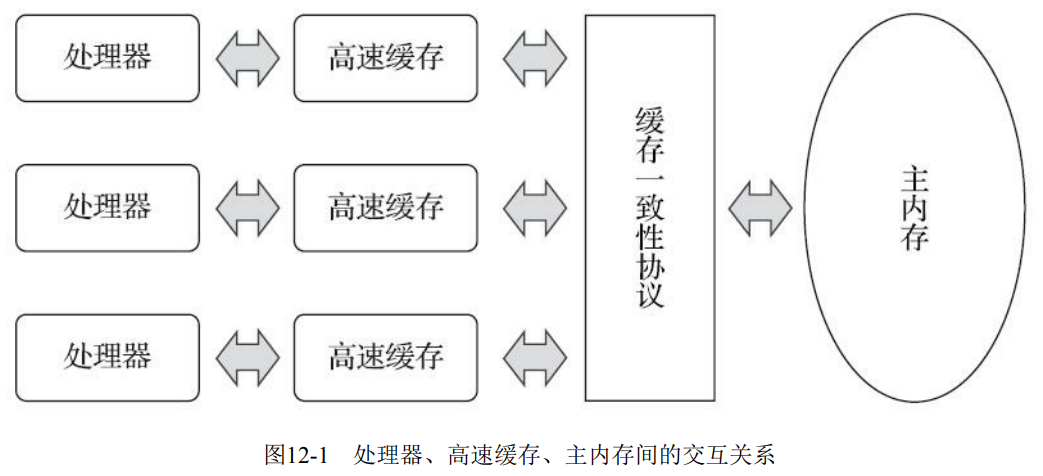
12.5 Java与协程

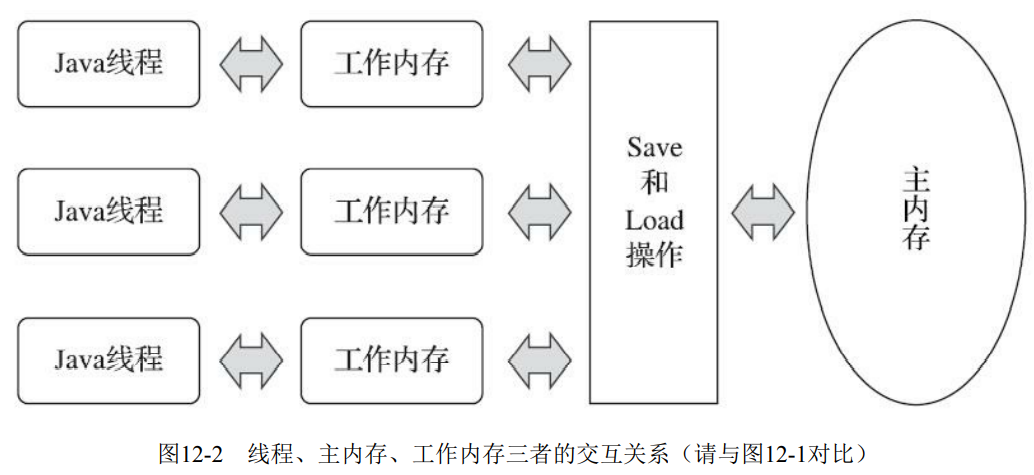
1. 并发的理解：a.压榨处理器的计算能力 b.一个服务端同时对多个客户端提供服务，则是一个更具体的并发场景

2. JMM的出现可以与缓存一致性相类比

a. 为了解决cpu与内存交互I/O之间的巨大差距，引入了缓存。加一层，出现缓存。但是也引入了新问题——缓存一致性——在多路处理器系统中，每个处理器有自己的高速缓存，而它们又共享同一主内存。为了解决一致性问题，各个**处理器**访问缓存时都遵循一些协议。 ->***JMM的***内存访问操作类似于硬件的缓存访问操作

b. 除了cache，为了使处理器内部的运算单元能尽量被充分利用，**处理器**可能对输入代码进行乱序执行优化。 -> 与乱序执行优化类似，***Java虚拟机***的即时编译器中也有指令重排序优化。





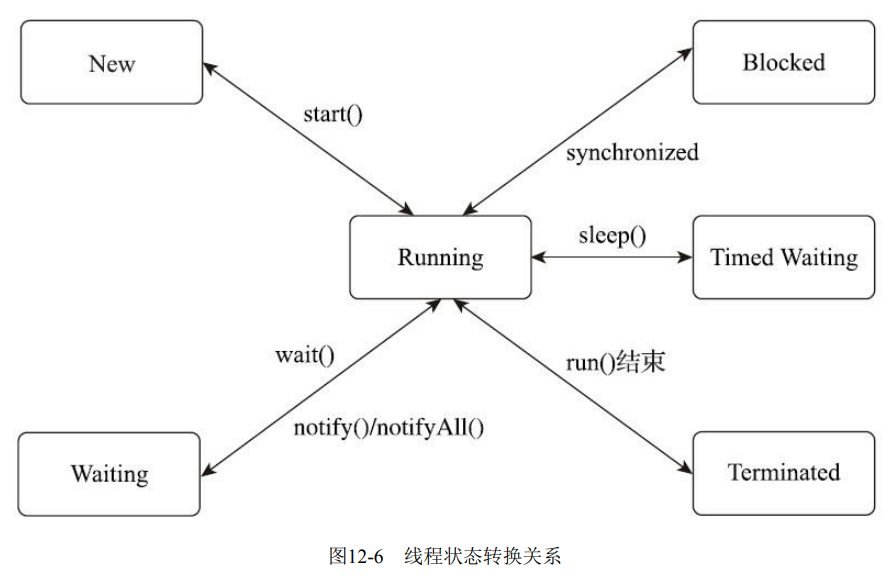
3. 我自己总结的：

每种JVM的实现都必须遵守JMM，which规定了对内存或cache的访问抽象，by定义各种变量的访问规则，即关注虚拟机中把变量存储到内存和从内存中取出变量值，具体定义了8种操作（lock、unlock、read、load、use\assign\store\write）。另外有happens-before表示有些偏序关系无须任何同步器协助就已经存在，可以在编码中直接使用。

4. 关于线程：

“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等），又可以独立调度。”

Java语言定义了6种线程状态：



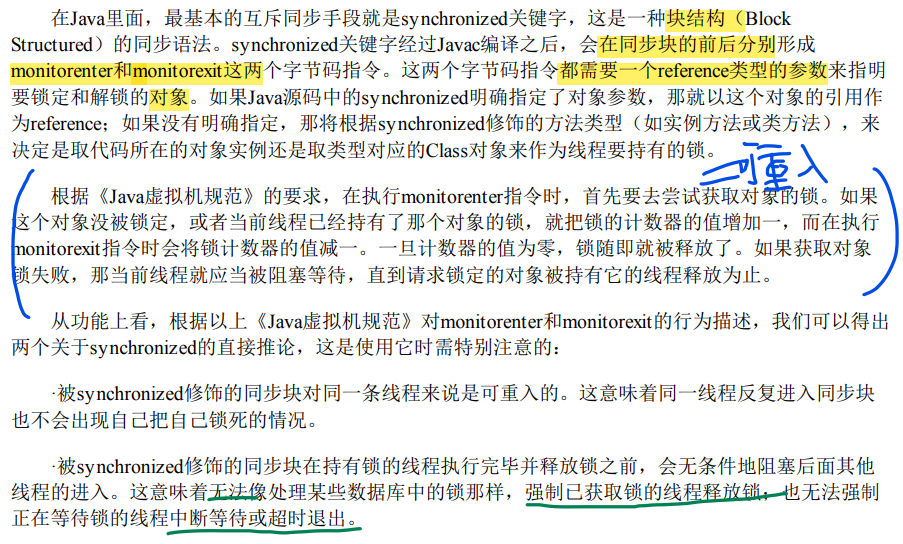
5. 谈谈协程

现在的协程是“有栈协程”

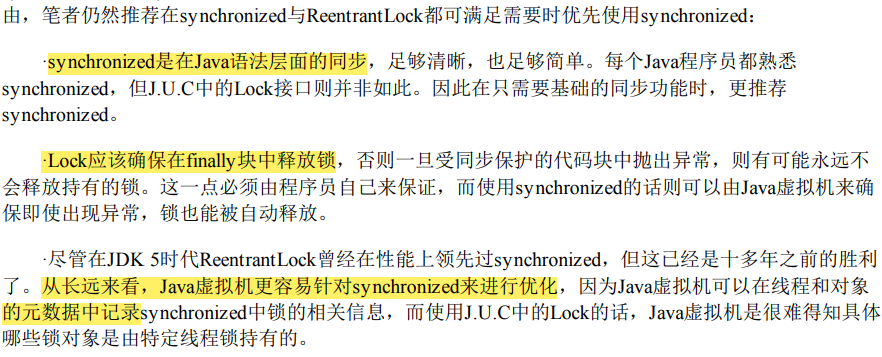
# 第13章 线程安全与锁优化

笔者认为《Java并发编程实战（Java Concurrency In Practice）》的作者Brian Goetz为“线程安全”做 出了一个比较恰当的**定义**：“当多个线程同时访问一个对象时，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，那就称这个对象是线程安全的。”

1. **是什么？:”**线程安全不是一个非真即假的二元排他选项，根据安全的由强至弱分为五类
   1. 不可变——*最安全、最纯粹*
      1. 不妨类比java.lang.String类的对象实例，它是一个典型的不可变对象，用户调用它的 substring()、replace()和concat()这些方法都不会影响它原来的值，只会返回一个新构造的字符串对象。
      2. 保证对象行为不影响自己状态的途径有很多种，最简单的一种就是把对象里面带有状态的变量都声明为final，这样在构造函数结束之后，它就是不可变的
   2. 绝对线程安全
      1. 如果说java.util.Vector是一个线程安全的容器，相信所有的Java程序员对此都不会有异议，因为它 的add()、get()和size()等方法都是被synchronized修饰的，尽管这样效率不高，但保证了具备原子性、 可见性和有序性。不过，即使它所有的方法都被修饰成synchronized，也不意味着调用它的时候就永远 都不再需要同步手段了
      2. 假如Vector一定要做到绝对的线程安全，那就必须在它内部维护一组一致性的快照访问才行，每次 对其中元素进行改动都要产生新的快照，这样要付出的时间和空间成本都是非常大的。
   3. 相对线程安全——*通常意义上的线程安全*
      1. 相对线程安全就是我们通常意义上所讲的线程安全，它需要保证对这个对象单次的操作是线程安全的，我们在调用的时候不需要进行额外的保障措施，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。
      2. 在Java语言中，大部分声称线程安全的类都属于这种类型，例如Vector、HashTable、Collections的 synchronizedCollection()方法包装的集合等。
   4. 线程兼容——*通常意义上，线程不安全的*
      1. 线程兼容是指对象本身并不是线程安全的，但是可以通过在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地使用。我们平常说一个类不是线程安全的，通常就是指这种情况。
      2. Java类库API中大部分的类都是线程兼容的，如与前面的Vector和HashTable相对应的集合类ArrayList和 HashMap等。
   5. 线程对立——有害的、尽量避免
      1. 一个线程对立的例子是Thread类的suspend()和resume()方法。一个尝试中断线程，一个尝试恢复线程，存在死锁风险。
2. **How？：线程安全的实现方法：**
   1. 互斥同步
      1. 同步是指多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一条/一些线程使用。 || 互斥是实现同步的一种手段，有：临界区、互斥量mutex、信号量semaphore
      2. **Java最基本的、最常见的互斥同步手段就是synchronized关键字**



* + 1. **为了弥补synchronized局限性**（绿线），Java类库中新提供了java.util.concurrentLock**.Lock接口**，用户可基于它，以非块结构来实现互斥同步。（synchronized：语言特性；Lock：类库层面）。
    2. **ReentrantLock**是Lock接口最常见的实现（also：ReentrantReadWriteLock）。相比synchronized，增加了一下高级功能：
       1. 等待可中断
       2. 公平锁
       3. 锁绑定多个条件（多个Condition对象）
    3. 推荐：优先使用synchronised



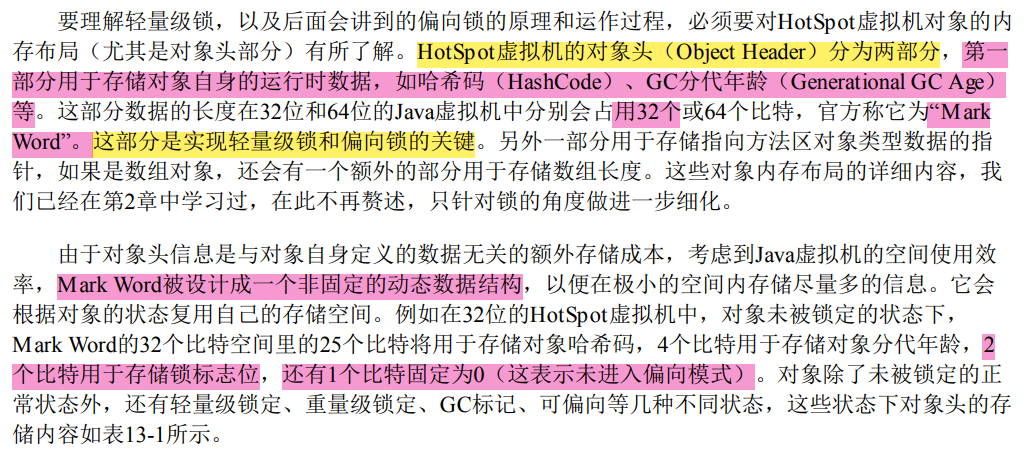
* 1. 非阻塞同步

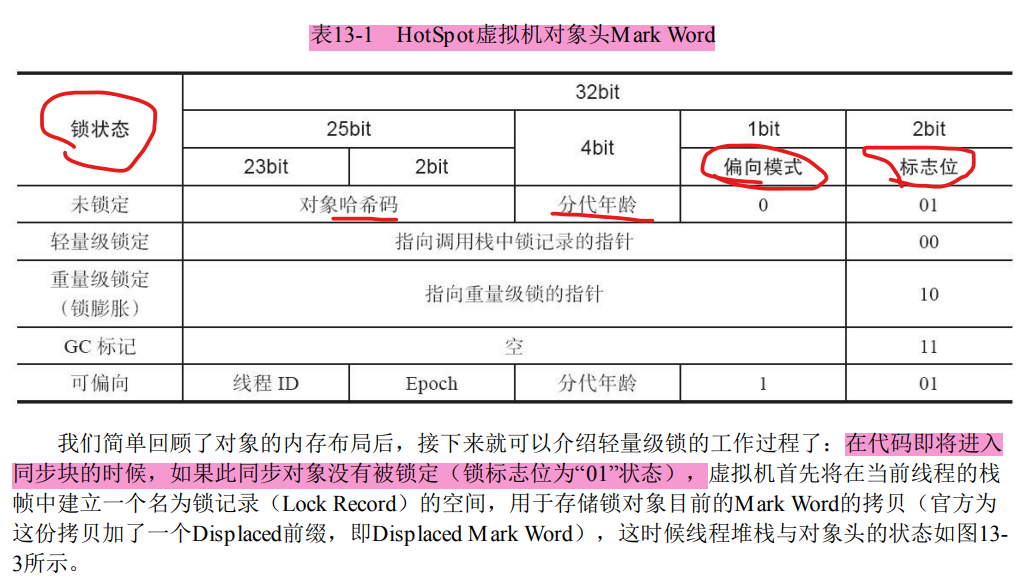
CAS、“ABA”问题，略

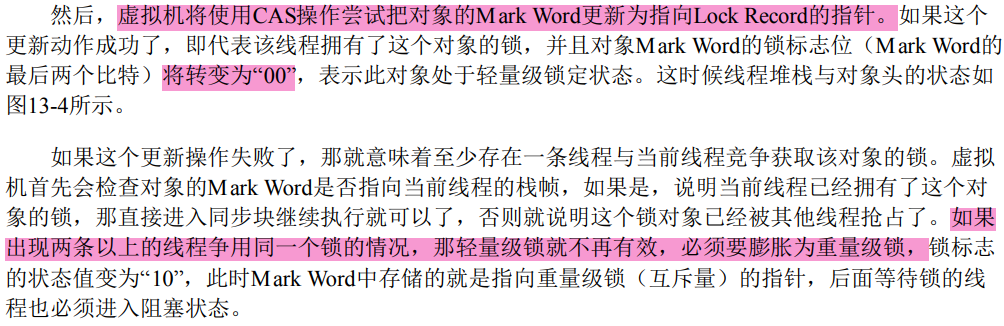
* 1. 无同步代码

有一些代码天生就是线程安全的。1.可重入代码 2.线程本地存储

1. **锁优化**
   1. 自旋锁与自适应自旋
      1. 自适应意味着自旋的时间不再是固定的了，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决定。
   2. 锁消除
      1. 即时编译器中，对不可能存在数据竞争的锁进行消除。源于逃逸分析的支持。
   3. 锁粗化
      1. 看起来反常识。适用于：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之中。
   4. 轻量级锁
      1. JDK6引入。“轻量”是相对于使用操作系统互斥量来实现的传统锁，which is“重量级”锁。
      2. 轻量级锁和偏向锁的实现原理和运作过程：







* 1. 偏向锁
     1. JDK6引入，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。如果说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**把整个同步都消除掉**，连CAS操作都不去做了。
     2. 原理：

