## 一、JVM参数设置：

* Xmx3550m：设置JVM**最大堆内存** 为3550M。
* -Xms3550m：设置JVM**初始堆内存** 为3550M。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。
* -Xss128k：设置**每个线程的栈** 大小。JDK5.0以后每个线程栈大小为1M，之前每个线程栈大小为256K。应当根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。
* -Xmn2g：设置**堆内存年轻代** 大小为2G。**整个堆内存大小 = 年轻代大小 + 年老代大小 + 持久代大小** 。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。
* -XX:PermSize=256M：设置**堆内存持久代** 初始值为256M。(貌似是Eclipse等IDE的初始化参数)
* -XX:MaxNewSize=size：新生成的对象能占用内存的最大值。
* -XX:MaxPermSize=512M：设置持久代最大值为512M。
* -XX:NewRatio=4：设置**堆内存年轻代（包括Eden和两个Survivor区）与堆内存年老代的比值（除去持久代）** 。设置为4，则年轻代所占与年老代所占的比值为1:4。
* -XX:SurvivorRatio=4：设置**堆内存年轻代中Eden区与Survivor区大小的比值** 。设置为4，则两个Survivor区（JVM堆内存年轻代中默认有2个Survivor区）与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6。
* -XX:MaxTenuringThreshold=7：表示一个对象如果在救助空间（Survivor区）移动7次还没有被回收就放入年老代。
* -XX:UserXXXGC：选择某种垃圾收集器。
* -XX：-UseCounterDecay 关闭热度衰减，只要系统运行时间足够长，绝大部方法都会被编译成本地代码。
* -XX:PretenureSizeThreshold：大于该设置值的参数将直接分配在老年代。
* -XX:+HeapDumpOnOutOfMemoryError：JVM会在遇到OutOfMemoryError时拍摄一个“堆转储快照”，并将其保存在一个文件中。

**二、java内存区域**

1. Java运行时数据区：
   1. 线程私有：虚拟机栈、本地方法栈、程序计数器
   2. 线程共享：java堆、方法区。（执行引擎、本地库接口）
2. 方法区存储的数据类型：类信息、常量、静态变量、即时编译器编译后的代码等。此外，运行时常量池也是方法区的一部分，class文件中的常量池在类加载后会存放在运行时常量池（注：常量池中内容：编译期生成的各种字面量和符号引用）。
3. 直接内存：在使用NIO时，可以使用Native函数库分配堆外内存，然后通过一个存储在java堆中的DirectByteBuffer作为这块内存的引用进行操作，这样能避免在Native堆和java堆中来回复制数据。
4. 对象创建（不包括数组和class对象）的过程
   1. 检查该指令的参数能否定位到常量池某个类的符号引用，并检查该类是否已经被加载、解析和初始化过，如果没有，则先进行类的加载过程。
   2. 类检查通过后，为新生对象分配内存，对象所需的内存大小在类加载完成后便可完全确定。
   3. 内存分配完成后，虚拟机需要将分配到的内存空间初始化为零值（不包括对象头）。这一步保证了对象的实例字段可以不赋初值直接使用。
   4. 设置对象头（见4）信息。
   5. 执行<init>方法，把对象安装程序员的意愿进行初始化。
5. 对象在内存中的布局：可分为三块区域：对象头、实例数据和对齐填充。
   1. 对象头：分为两部分信息：
      1. 存储对象自身的运行时数据：如哈希码、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。这部分长度一般和虚拟机位数相同。
      2. 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确定这个对象时哪个类的实例。

此外，如果对象是一个java数组，对象头还会有一块来记录数组长度的数据。

* 1. 实例数据：代码中定义的各种类型的字段内容，无论是从父类继承的，还是子类自己定义的。
  2. 对齐填充

1. 如何使用创建的对象：通过栈上的reference数据来操作堆上的具体对象。可使用句柄和直接指针两种对象的定位方式。
2. **垃圾回收**
3. 确定需要回收的对象的算法：
   1. 引用计数法，每引用一次计数器加1，计数器为0时进行垃圾回收。缺点：无法解决循环引用问题。
   2. 可达性分析算法：从GC Roots开始向下搜索，不可达的对象进行垃圾回收。GC Roots包括：
      1. 虚拟机栈（栈帧中的本地变量表）中引用的对象。
      2. 方法区中类静态属性引用的对象。
      3. 方法区中常量引用的对象。
      4. 本地方法栈中JNI（即一般说的Native）引用的对象。

注：static final修饰的变量就是常量，存在在常量存储区域（运行时常量池）。final修饰的变量在栈中，只不过它修饰的变量只能赋一次值。

1. 引用分为：强引用、软引用、弱引用和虚引用四种：
   1. 强引用：类似于Object c = new Object()。只要强引用还存在，垃圾收集器就永远不会回收掉被引用的对象。
   2. 软引用：系统在将要发生OOM之前，会对这些引用的对象进行第二次回收，如果这次回收还没有足够的内存，才会抛出OOM异常。使用SoftReference类实现软引用。
   3. 弱引用：被弱引用关联的对象只能生存到下一次GC之前。使用WeakReference实现弱引用。
   4. 虚引用：不会对对象的生存时间构成影响，设置虚引用的唯一目的是能在这个对象被垃圾回收时收到一个系统通知。使用PhantomReferenc实现虚引用。
2. 回收方法区中的类需要满足的条件：
   1. 该类所有实例都被回收。
   2. 加载该类的ClassLoader已经被回收。
   3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法。
3. 垃圾回收算法
   1. 标记-清理算法（Mark-Sweep）：先标记，后清理。缺点：
      1. 效率问题。两个过程效率都不高。
      2. GC后会产生大量不连续的内存碎片。
   2. 复制算法：将内存分为两块，一次使用一块，回收后将存活的对象复制到另一块，然后清理使用过的那一块。当前JVM新生代的收集都采用这种算法：将新生代分为一块较大的Eden区和两块较小的Survivor区，每次使用Eden和一块Survivor，GC后存活的对象放入另一块Survivor。当Survivor无法容纳存活的对象时，需要老年代进行分配担保。
   3. 标记-整理算法：先标记，然后将所有存活的对象移动到一段，再清理端边界以外的内存。
   4. 分代收集算法：即在不同代采用不同的收集算法。
4. 垃圾收集器
   1. 新生代收集器（均使用复制算法）：Serial、ParNew、Parallel Scavenge
      1. Serial：单线程收集器，回收时会暂停其他所有工作线程。
      2. ParNew：Serial的多线程版本，可以与CMS配合。
      3. Parallel Scavenge：多线程收集器，其目标是达到一个可控制的吞吐量。高吞吐量可以高效的利用CPU时间，因此适合后台计算。可以设置最大垃圾收集停顿时间，也可以直接设置吞吐量的大小。并且可以使用自适应的调节策略（虚拟机根据当前系统的运行情况动态调节新生代大小、Eden和Survivor比例、晋升老年代对象大小等参数）。无法与CMS配合。
   2. 老年代收集器：Serial Old、Parallel Old、CMS
      1. Serial Old：Serial的老年代版本，使用标记-整理算法，回收时会暂停其他所有工作线程。。
      2. Parallel Old：使用多线程和标记-整理算法。在注重吞吐量和CPU资源敏感的场合，可以考虑Parallel Scavenge+Parallel Old。
      3. CMS：以获取最短回收停顿时间为目标，使用标记-清除算法。可分为初始标记、并发标记、重新标记和并发清除四个步骤。其中，只有初始标记和重新标记需要Stop the world。缺点有三个：首先，CMS对CPU资源非常敏感，其在并发阶段会占用一部分CPU资源。其次，CMS无法处理浮动垃圾（并发清理阶段产生的垃圾），因此需要预留一部分空间。当CMS预留的空间无法满足程序需要时，会启用Serial Old进行GC。最后，标记清除算法会带来碎片，可以设置在多少次清除后，进行一次碎片清理。
   3. G1收集器：将整个JAVA堆分为多个大小相等的Region。有如下特点：
      1. 并行与并发。
      2. 分代收集：G1可独立管理整个GC堆，其在逻辑上分为老年代和新生代，两代不在物理隔离。
      3. 空间整合：整体上基于标记-整理算法，局部（两个Region）之间基于复制算法。
      4. 可预测的停顿：G1会跟踪每个Region中垃圾的价值大小，保证G1在有限的时间内获取尽可能高的收集效率。步骤和CMS相似：初始标记、并发标记、最终标记和筛选回收。
5. GC日志：

33.125: [GC [DefNew: 3324k->152k(3712k), 0.0025925 secs] 3324k->152k(11904k),0.0031680 secs]

103.125: [Full GC [Tenured: 0k->152k(10240k), 0.014925 secs] 4603k->152k(19904k),[Perm :2999k->2999k[21248k], 0.0150007 secs] [Times :user = 0.01 sys = 0.00, readl = 0.02 seces]

33.125表示GC发生的时间，数字是从JVM启动以来的秒数。

GC和Full GC表示GC的类型

[DefNew: 3324k->152k(3712k), 0.0025925 secs] ：DefNew表示GC发生的区域，这里显示的区域名与GC收集器相关。Perm表示永久代。3324k->152k(3712k)表示：GC前该区域已使用容量->GC后该区域已使用容量（该区域内存总容量）

3324k->152k(11904k),0.0031680 secs]：代表JAVA堆的回收情况（前->后（总），该内存区域GC时间）

[Times :user = 0.01 sys = 0.00, readl = 0.02 seces]：与linux time命令输出含义相同。分别表示用户态、内核态消耗的CPU时间和操作开始到结束的总时间（包含I/O、阻塞等时间）。多核CPU会叠加这些时间。

1. 两种GC：
   1. Minor GC：新生代GC，在新生代发送的垃圾回收动作，速度很快。大多数情况下，对象在新生代Eden区分配，当Eden区没有足够空间时，将发起Minor GC。
   2. Full GC/Major GC：老年代GC，速度会比较慢。
2. 晋入老年代的规则：
   1. 大对象直接进入老年代。应当避免大量朝生夕灭的大对象。
   2. 长期存活的对象进入老年代。
   3. 动态对象年龄判定：如果Survivor空间中相同年龄索引对象大小总和大于Survivor的一半，则年龄大于或等于该年龄对象直接进入老年代。
3. **几个JDK bin中的工具**
4. jstat：虚拟机统计信息监视工具：

Jstat -gc 2765 250 20 ：表示每250毫秒查询进程2764垃圾收集状况，一共查询20次。

1. jinfo 查看和调整虚拟机参数：

jinfo -flag [参数名] [pid]

1. jmap：java内存映像工具，用来生成堆转储快照（dump文件）：

jmap -dump:format=b,file=eclpse.bin 3500:表示为进程3500生成dump文件。

1. jhat:虚拟机转储快照分析工具，内置http/html服务器，能分析dump文件，并在浏览器上查看：

jhat eclipse.bin :分析jmap生成的dump文件。

1. jstack:堆栈跟踪工具：用于生成虚拟机当前时刻的线程快照。一般用来定位线程出现长时间等待的原因:

jstack [option] [pid]

1. HSDIS:JIT生成代码反汇编。
2. JConsole:可视化的监视和管理控制台。
3. VisualVM:多合一故障处理工具。可安装各种插件。BTrace插件：能在不停止java程序的前提下加入调试代码（如输出程序中间变量等）
4. javap：是JDK自带的反汇编器，可以查看java编译器为我们生成的字节码。通过它，我们可以对照源代码和字节码，从而了解很多编译器内部的工作。

例：先编译：javac JAVAPTest.Java

执行反编译：javap -c JAVAPTest

1. **虚拟机类加载机制**
2. java语言里，类型的加载、连接和初始化都是在程序运行期间完成的。
3. 类的生命周期：
   1. 加载：
      1. 通过类名获得定义该类的二进制字节流。
      2. 将字节流代表的静态存储结构转化为方法区的运行时数据结构。
      3. 方法区（HotSpot虚拟机）中生成代表这个类的java.lang.Class对象。
   2. 连接：包含3个阶段：验证、准备、解析
      1. 验证：确保Class文件中的字节流信息符合当前虚拟机要求，且不会危害虚拟机自身的安全
      2. 准备：为类变量（static）分配内存并设置初始值（通常为零值）。如，Static int x=10，x=10是在初始化阶段执行。
      3. 解析：将常量池的符号引用替换为直接引用。
   3. 初始化：执行类构造器<clinit>()方法。<clinit>()方法方法的特点：
      1. 由所有类变量的赋值动作和static{}块中语句合并而成。如果两者都没有，则不会生成<clinit>()方法。（接口同理，但接口中本身不能使用静态语句块）
      2. 虚拟机会保证子类的<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。（执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口定义的变量使用时，才会初始化父接口。接口的实现类在初始化时也不会执行接口的<clinit>()方法）
      3. 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确的加锁和同步。只有一个线程会执行这个类的<clinit>()方法，其他线程阻塞等待。
   4. 使用
   5. 卸载
4. 不会触发类的初始化的例子：
   1. 通过子类引用父类的静态字段，不会导致子类初始化。对于静态字段，只有自己定义这个字段的类才会被初始化。
   2. 通过数组定义来引用类，不会触发此类的初始化，如：SuperClass[] a = new SuperClass[10];
   3. 使用常量时，不会触发定义常量的类的初始化，如：

Public class ConstClass{public staitc final String HELLO=”hello”;}

Public class Main{

public static void main(String args[]){

System.out.println(ConstClass.HELLO)}

}

1. 类加载器
   1. 对于任意一个类，都需要由加载他的类加载器和这个类本身一同确定其在JAVA虚拟机中的唯一性。每一个类加载器都有一个独立的类名称空间。这个唯一性会影响到代表类的Class对象的equals()方法、isInstance()方法和instanceof关键字的结果。
   2. 双亲委派模式：当类加载器收到类加载请求时，它会先把这个请求委托给父类加载器去完成。当父类加载器无法加载时，子加载器才会尝试自己加载。优点：使得如java.lang.Object类在各个类加载环境中都是同一个类。
   3. 破坏双亲委派模式：
      1. 可以通过java.lang.Thread类的setContextClassLoader()设置。默认从父线程继承，如果全局都没有设置，则为应用程序类加载器。
      2. 代码热替换、模块热部署。OSGI。
   4. 类加载器的种类：按父子关系排序：
      1. 启动类加载器（BootStrap ClassLoader）：加载<JAVA\_HOME>\lib（也可用-Xbootclasspath参数指定路径）目录中的类库。无法被JAVA程序直接引用。
      2. 扩展类加载器（Extension ClassLoader）：负责加载<JAVA\_HOME>\lib\ext（也可用java.ext.dirs系统变量指定路径）目录中的类库。由sun.misc.launcher$ExtClassLoader实现。
      3. 应用程序类加载器（Application ClassLoader）：负责加载ClassPath上的类库。也称为系统类加载器（是ClassLoader.getSystemClassLoder()方法的返回值）。为程序中默认的类加载器。由sun.misc.launcher$AppClassLoader实现。
      4. 自定义类加载器。
2. 书中的一个问题：如果有10个WEB应用程序都是用Spring来进行组织管理的话，可以把Spring放到Common或Shared目录下（Tomcat5.0）让这些程序共享。Spring要对用户程序的类进行管理，自然要能访问到用户程序的类，而用户程序显然是放在/WEB-INF目录中的。那么被CommonClassLoader或SharedClassLoader加载的Spring如何访问并不在其加载范围内的用户程序呢？

答案是使用线程上下文类加载器来实现的，仔细看源码会发现，spring加载类所用的classloader都是通过Thread.currentThread().getContextClassLoader()来获取的，而当线程创建时会默认 setContextClassLoader(AppClassLoader)，即spring中始终可以获取到这个AppClassLoader(在tomcat里就是WebAppClassLoader)子类加载器来加载bean，以后任何一个线程都可以通过getContextClassLoader()获取到WebAppClassLoader来getbean了

1. 自定义类加载器
   1. web服务器自定义类加载器的原因：
      1. 使得同一服务器上两个web应用所使用的java类库相互隔离。因为两个不同的应用可能会依赖同一个第三方类库的不同版本。这样就可以使两个应用程序的类库可以互相独立使用。
      2. 同一服务器上的两个web应用所使用的java类库可以共享。
      3. 服务器需要尽可能保证自身的安全不受部署的web应用程序影响。一般地，服务器所使用的类库与应用程序的类库相互隔离。
      4. 支持JSP应用的HotSwap功能。因为JSP运行时修改概率远大于第三方类库和自身的Class文件。
   2. 自定义类加载器：需要继承ClassLoader并重写findClass方法（见代码实例1）
2. 注意：ClassLoader会持有它加载的类的Class对象，也就是会持有这些类的静态字段，这在某些时候可能会导致内存泄漏。java.lang.ClassLoader中持有这些对象的字段：

// The classes loaded by this class loader. The only purpose of this table

// is to keep the classes from being GC'ed until the loader is GC'ed.

private final Vector<Class<?>> classes = new Vector<>();

1. **虚拟机字节码执行引擎**
2. 执行引擎的工作过程：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。
3. 栈帧：虚拟机栈的栈元素。每一次方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。存储着：局部变量表、操作数栈、动态连接和方法的返回地址等。栈帧的大小编码结束就已经确定了。
4. 局部变量在未初始化不能直接使用（并没有赋初值），这点不同于类变量。
5. 方法调用（不同于方法执行，其唯一任务确定调用哪一个方法）：
   1. 解析：在编译期完成的，即在编译期就能确定调用版本的方法。包括静态方法、私有方法、实例构造器、父类方法以及final修饰的方法。
   2. 分派：

Human woman = new Woman();

Animal woman = new Woman();

Human 和 Animal称为变量的静态类型，Woman称为变量的实际类型。

* + 1. 静态分派：依赖静态类型定位方法的执行版本。典型事例：方法重载（overload）

（见代码实例2）。

虚拟机在重载时，根据参数的静态类型而不是实际类型作为判定依据。

* + 1. 动态分派：根据实际类型定位方法的执行版本。典型事例：方法重写（Override）
    2. 动态分派的实现：为类在方法区建立虚方法表。虚方法表存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，则子类的虚方法表里的入口地址与父类相同。如果被重写，则子类的虚方法表里的入口地址替换为指向子类实现版本的入口地址。方法表的初始化一般在类加载的连接阶段，准备完类的变量初始值后，虚拟机会把该类的方法表也初始完毕。

5.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的字节码指令（基于栈，依赖于操作数栈、大部分为零地址、方便移植，但速度稍慢的栈架构指令集。）的过程。这部分在JVM之外进行，而解释器在JVM内部。

6.动态代理：

a) 使用（见实例代码3）

b) 原理：Proxy.newProxyInstance()返回了一个实现Ihello,Ibye接口，代理了origin（new Hello()）实例行为的对象。这个对象就是一个代理类（该类由sun.misc.ProxyGenerator.generatorProxyClass()直接生成字节码）的实例。该类为传入接口中的每一个方法，以及从java.lang.Object中继承来的equals()、hashCode()、toString()方法都生成了对应的实现（这是代理模式的基本实现），并且统一调用了InvocationHandler对象的invoke()方法，各个方法的区别仅仅是传入的参数和Method对象有所不同。例如，代理类中sayHello()方法的实现（省略了throw和catch）：

public final void sayHello() throws {

try{

this.h.invoke(this,m3,null);

return;

}catch(Exception){}

}

其中，m3（private static Method m3）的初始化在该类的static块内

m3 = Class.forName(“zhangxiao.DynamicProxyTest$Ihello ”).getMethod(“sayHello”,new Class[0]);（new class[0]代表参数为空.

1. **java相关语法糖**
2. 泛型和类型擦除
   1. 使用：泛型可以用于类、接口和方法。
   2. 原理：java中的泛型只在程序源码中存在。在编译后的字节码文件中，就已经替换为原来的原生类型（裸类型）了，并在相应的地方插入了强制转型的代码。对于运行期的JAVA语言来说，ArrayList<Integer>和ArrayList<String>就是同一个类。为了支持泛型，引入了Signature（等）属性，其作用是存储一个方法在字节码层面的特征签名，包括了参数化类型的信息。此外，从Signature属性可以看出，擦除法的擦除，只是对方法的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，这样我们就能通过反射手段取得参数和类型。

注：code属性（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属性中）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，方法体里面的代码）和元数据（Metadata，包括类、字段、方法定义以及其他信息）两部分，那么在整个Class文件中，code属性用于描述代码，所有的其他数据都用来描述元数据。

1. 自动装箱、拆箱；循环遍历；条件编译:

自动装箱时编译器调用valueOf将原始类型值转换成对象，同时自动拆箱时，编译器通 过调用类似intValue(),doubleValue()这类的方法将对象转换成原始类型值。

自动装箱和拆箱的注意事项：“==”可以用于原始值进行比较，也可以用于对象进行比较，当用于对象与对象之间比较时，比较的不是对象代表的值，而是检查两个对象是否是同一对象，这个比较过程中没有自动装箱发生。进行对象值比较不应该使用”==“，而应该使用对象对应的equals方法。

此外,出于节省内存的考虑，JVM会缓存-128到127的Integer对象。因次obj1和obj2实际上是同一个对象。（ Integer obj1 = 1; Integer obj2 = 1;）（见代码实例4，代码实例5）

1. **运行期优化**
2. 即时编译期（JIT编译器）：当JVM发现某个方法或者代码块运行特别频繁时，就会将这些代码作为热点代码，使用JIT编译器将其编译成与本地平台相关的机器码，并进行优化。
3. 热点代码分为两类：多次调用的方法和多次执行的循环体。可以采用-XX：-UseCounterDecay 来关闭热度衰减，这样，只要系统运行时间足够长，绝大部方法都会被编译成本地代码。
4. 解释器和编译器各自的优势：解释器启动迅速，编译器执行效率高。
5. Hotspot虚拟机内置了两个即时编译器：C1（Client Compiler）和C2（Server Compiler）。默认采用解释器和其中一个编译器直接配合的方式工作。可以使用”-client”或”-server”参数去强制指定虚拟机运行的模式。
6. 逃逸分析：为其他优化收到提供依据。其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；甚至有可能被外部线程访问到（例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。

如果一个对象经过逃逸分析，确定其不会被别的方法或者线程访问到，则可对这个对象进行如下的优化：

* 1. 栈上分配：一般对象的创建都是在JAVA堆，如果把不会逃逸的对象分配到栈上，对象占用的空间就可以随栈帧出栈而销毁，减少GC压力。
  2. 同步消除：因为这个对象不会被其他线程访问到，因此对这个对象的同步措施就可以消除掉。
  3. 标量替换：把一个对象拆散，根据程序访问的情况，将其使用的成员变量恢复原始类型来访问就叫做标量替换。这样，程序真正执行的时候可能不创建这个对象。

1. **java内存模型和线程**
2. 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，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节（可以认为是规定了虚拟机栈和堆之间如何进行交互）。
3. Java内存模型的定义：
   1. 所有的变量都存储在主内存。
   2. 每条线程有自己的工作内存，其中保存了该线程使用到的变量的主内存副本拷贝。（这里的拷贝不一定是拷贝整个对象，可能只是拷贝这个对象的引用、对象在某个线程中访问到的字段）
   3. 线程对变量的所有操作（读取、赋值等）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。
   4. 不同线程之间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。

这里的主内存主要对应于java堆中的对象实例数据部分，而工作内存主要对应于虚拟机栈中的部分区域。

1. volatile变量:
   1. 特性：
      1. 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可以立即得知的。
      2. 禁止指令重排序优化（java会在保证“线程内表现为串行语义的前提下”对普通变量进行重排序优化）。
   2. 使用原则：因为volatile变量值保证了可见性，因此在不符合以下两条规则的场景下，volatile仍然需要加锁：
      1. 运算结果不依赖于当前值，或者能够确保只有单一的线程修改变量的值
      2. 变量不需要与其他的状态变量共同参与不变约束（即如果使用volatile变量来标识状态，则只能用一个volatile变量来标识，多个变量依然需要加锁）。
   3. 原理：
      1. volatile变量在使用前必须先从主内存中刷新最新的值，在修改后必须立即同步回主内存。（即read,load,use 操作必须在连续一起出现，assign,store,write操作必须连续一起出现。每个操作本身都是原子的）。
      2. 禁止指令重排序优化原理：如果你的字段是volatile，Java内存模型将在写操作后插入一个写屏障指令，在读操作前插入一个读屏障指令。内存屏障，指重排序时不能把后面的指令重排序到内存屏障之前的位置。内存屏障另一个作用是强制更新一次不同CPU的缓存。例如，一个写屏障会把这个屏障前写入的数据刷新到缓存，这样任何试图读取该数据的线程将得到最新值，而不用考虑到底是被哪个cpu核心或者哪颗CPU执行的。指令重排优化既可以发生在编译期，也可能发生在运行期。
   4. 使用实例：
      1. 使用DCL单例模式时，应该将实例变量声明为volatile（实例和原理见代码实例6）。
      2. volatile变量的自增操作（++）并不是原子的。例如public static volatile int race = 0;race++的字节码如下：

0: getstatic

3: iconst\_1

4: iadd

5: putstaic

volatile关键字只能保证getstatic取得的race值是最新的，但在执行3,4时，其他线程可能已经修改了race的值，因此5时就可能覆盖掉其他线程的修改。

1. 先行发生原则
   1. 含义：先行发生是java内存模型中定义 两项操作之间的偏序关系，如果说操作A先行发生于操作B，其实就是说在发生操作B之前，操作A产生的影响（包括修改共享变量的值、发送了消息、调用了方法等）能被B观察到。
   2. Java中一些天然的先行发生关系，如果两个操作之间的关系不在此列，且无法从下列规则中推导出来的话，他们就没有顺序性保障，虚拟机可以对他们随意进行重排序：
      1. 程序次序规则：在一个线程内，按程序代码顺序（控制流顺序），书写在前的操作先行发生于书写在后的操作。
      2. 管程锁定规则：一个unlock操作先行发生于后面对同一个锁的lock操作。（这里的后面是指时间上的先后顺序）
      3. volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作。（这里的后面意义同上）。
      4. 线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作。
      5. 线程终止规则：线程中所有操作都先行发生于对此线程的终止检测。
      6. 线程中断规则。对线程interrupt()方法的调用先行发生于被中断线程的代码所检测到的中断事件。
      7. 对象终结规则：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fianlize()方法的开始。
      8. 传递性：如果A先行发生于B，B先行发生于C，则A先行发生于C。
2. java线程的实现：对于sun JDK来说，它的windows版和linux版都是使用一对一的线程模型实现的，一条java线程就映射到一条轻量级进程中。轻量级进程是操作系统内核支持的线程（这种线程由内核来完成线程切换）的一种高级接口，每个轻量级进程都由一个内核线程支持（1：1）。这种实现方式的优点：每个轻量级进程都是一个独立的调度单元，即使有一个轻量级进程在系统调用中阻塞，也不会影响整个进程的工作。缺点在于：线程的各种操作需要进行系统调用，代价较高。
3. java的线程调度：采用抢占式调度，每个线程由系统分配执行时间，线程的切换不由线程本身来决定（可以使用Thread.yield()让出时间，但没办法主动获取时间）。java线程的优先级不靠谱的原因：不一定能和操作系统本身的优先级一一对应，可能会有多个java线程的优先级映射到同一个系统本身优先级；此外，优先级可能会被系统自行改变。
4. Java线程的几种状态：
   1. 新建（New）：创建后尚未启动。
   2. 运行（Runnable）：包括了操作系统线程状态中的Running和Ready，此状态的线程有可能正在执行，也有可能正等待CPU分配执行时间。
   3. 无限期等待（Waiting）：该状态的线程不会被CPU分配执行时间，他们要等待被其他线程显式唤醒。
   4. 限期等待（Timed Waiting）：该状态的线程不会被CPU分配执行时间，无须等待其他线程显式唤醒，一定时间会它们会由系统自动唤醒。
   5. 阻塞（Blocked）：线程被阻塞了，在等待着获取到一个排他锁。
   6. 结束（Terminated）：已经终止的线程。
5. **线程安全与锁优化**
6. 什么是线程安全：
   1. 绝对线程安全：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，那么这个对象时线程安全的。
   2. 相对线程安全：保证对这个对象单独的操作是线程安全的，但对于一些特定顺序的连续调用，就可能需要在调用时进行一些额外的同步手段来保证调用的正确性。java中大部分线程安全类都属于这种类型，如Vector，HashTable等。
7. synchronized关键字：
   1. 原理：会在同步块前后分别形成monitorenter和monitorexit两个字节码指令，这两个字节码都需要一个reference类型的参数指明锁定和解锁的对象。如果java程序中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，那就根据修饰的是实例方法还是类方法，去取相应的对象实例或者Class对象来作为锁对象。
   2. Monitorenter：每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：
      1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
      2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
      3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。
   3. Monitorexit：执行monitorexit的线程必须是objectref所对应的monitor的所有者。指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。
   4. 其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。（Java并发编程：Synchronized及其实现原理：原文地址：<http://www.cnblogs.com/paddix/p/5367116.html）>
8. ReentrantLock：API层面的互斥锁，相对于synchronized高级功能：
   1. 等待可中断：当持有锁的线程长期不释放锁时，正在等待的线程可以选择放弃等待，改为处理其他事情（而使用synchronized锁的线程在等待时处于阻塞状态，不响应中断）。
   2. 公平锁：是指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取锁；而非公平锁不保证这一点，在锁释放时，任何一个等待锁的线程都有机会获得锁。synchronized和默认情况下的ReentrantLock都是非公平锁。
   3. 锁可以绑定多个条件：一个ReentrantLock可以绑定多个Condition对象，使得可以只唤醒等待同一个锁的某一部分线程（而不是全部）。
   4. 读写锁：ReentrantReadWriteLock rwl=new ReentrantReadWriteLock();
      1. rwl.readLock():得到一个可以被多个读操作共用的读锁，排斥所有写操作。
      2. rwl.writeLock()得到一个写锁，排斥所有其他读操作和写操作。
9. 非阻塞同步：基于冲突检测的乐观并发策略，也就是乐观锁，即先进行操作，如果没有其他线程争用共享数据，那么就操作成功；如果共享数据有争用，产生了冲突，那就采取其他的补偿措施（最常见的补偿措施就是不断重试），这种乐观的并发策略的许多实现都不需要将线程挂起。而悲观的并发策略，悲观锁，认为只要不去做正确的同步措施，那就肯定会出现问题，因此无论共享数据是否真的会出现竞争，都会进行加锁、用户态核心态转换、维护锁计数器、检查是否有被阻塞的需要唤醒的线程等操作（事实上JVM会优化掉很大一部分不必要的加锁）。
10. CAS指令（Compare-and-Swap，比较并交换）：该处理器指令需要三个操作数，分别是内存位置、旧的预期值和新值。当且仅当内存中的值符合旧的预期值时，处理器用新值更新内存中的值，否则不执行更新。上述的处理过程是一个原子操作。

CAS指令使用的实例：AtomicInteger的incrementAndGet()，其源码如下：

public final int incrementAndGet(){

for(;;){

int current = get();

int next = current+1;

if(compareAndSet(current,next))

Return next;

}

}

该方法就是在一个无限循环中，不断尝试将一个比当前值大1的新值赋给自己。如果失败了，说明在执行“获取-设置”操作的时候值已经有了修改，于是再次循环进行下一次操作，直到设置成功为止。

1. 线程本地存储：java中可以使用java.lang.ThreadLocal来实现线程本地存储的功能。每一个线程的Thread对象中都有一个ThreadLocalMap对象，这个对象存储了一组以ThreadLocal.threadLocalHashCode为键，以本地线程变量为值的K-V值对，ThreadLocal对象就是当前线程的ThreadLocalMap的访问入口，每一个ThreadLocal对象都包含了一个独一无二的threadLocalHashCode值，使用这个值就可以在线程K-V值对中找回对应的本地线程变量。（见代码实例7）
2. 锁的优化
   1. 自旋锁：在一个线程在请求锁而需要等待时，先不放弃处理器的执行时间，而是执行一个忙循环（自旋），来看持有锁的线程是否会释放锁。优点在于，如果等待后成功得到锁，则能避免线程切换的开销；缺点在于，忙循环是在白白的消耗处理器资源。
   2. 锁消除：虚拟机会忽略掉那些经过逃逸分析，判断对象不会出现逃逸的同步。
   3. 锁粗化：如果JVM检查到一串零碎的操作不断的对同一对象重复的加锁和解锁（例如循环中），JVM将会把加锁同步的范围粗化到整个操作序列的外部。
   4. 轻量级锁：
      1. 目标：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损耗。
      2. 实现：在代码进入同步块时，如果此同步对象没有被锁定，JVM首先将在当前线程的栈帧建立一个名为锁记录（Lock Record）的空间，用于存储对象目前的Mark Word（即对象头中存储对象自身的运行时数据的部分，详见二、java内存区域）的拷贝。然后JVM会使用CAS操作尝试将对象的Mark Word更新为指向Lock Record指针，如果这个更新操作成功了，那么这个线程就拥有了该对象的锁（相应的锁标志位变为00，之前为未锁定-01）。如果失败了，JVM会先检查对象的Mark Word是否指向当前线程的栈帧，如果是，则说明当前线程已经拥有这个对象的锁。否则，则说明这个锁对象被其他线程抢占了。如果两条以上线程争用同一个锁，则轻量级锁膨胀为重量级锁，此时Mark Word存储指向重量级锁（互斥量）的指针。轻量级锁的解锁也是通过CAS操作进行的，将Lock Record中的值替换回对象的Mark Word。如果CAS替换失败，说明有其他线程尝试过获取该锁，那么就得在释放锁的同时，唤醒被挂起的线程。
      3. 优化依据：对于绝大部分锁，整个同步期间都是不存在竞争的。
   5. 偏向锁：在**无竞争**的情况下消除整个同步。当锁对象第一次被线程获取的时候，JVM会将对象头中的标志位设为01-偏向模式，同时使用CAS把获取到这个锁的线程ID记录在对象的Mark Word。如果成功，那么持有偏向锁的线程以后每次进入这个锁相关的同步块时，虚拟机都可以不再进行任何同步操作。当有其他线程尝试获取这个锁时，偏向模式宣告结束。后续操作就如上述的轻量级锁那样执行。

**代码实例**

1.

**MyClassLoder.java:**

import java.io.ByteArrayOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

public class MyClassLoader extends ClassLoader{

private String path="C:/Users/zhangxiao/Desktop";

@Override

protected Class<?> findClass(String name) throws ClassNotFoundException {

// TODO Auto-generated method stub

byte[] classData = loadClassData(name);

Class<?> clazz = super.defineClass(name, classData, 0, classData.length);

return clazz;

}

private byte[] loadClassData(String name){

String path = name.replace(".", "/");

try {

FileInputStream is = new FileInputStream(new File((this.path+File.separator+path+".class")));

ByteArrayOutputStream bout=new ByteArrayOutputStream();

int b;

while((b=is.read())!=-1){

bout.write(b);

}

return bout.toByteArray();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

return null;

}

}

**ClassLoaderTest.java:**

import java.lang.reflect.InvocationTargetException;

import java.lang.reflect.Method;

public class ClassLoaderTest {

public static void main(String args[]) throws ClassNotFoundException, InstantiationException, IllegalAccessException, NoSuchMethodException, SecurityException, IllegalArgumentException, InvocationTargetException{

ClassLoader c = new MyClassLoader();

Class<?> clazz = c.loadClass("Test");

Object obj = clazz.newInstance();

Method method = clazz.getDeclaredMethod("say");

method.invoke(obj);

System.out.println(clazz.getClassLoader());

}

}

**Test.java:**

public class Test{

static{

System.out.println("load Test");

}

public void say(){

System.out.println("hello");

}

}

**输出：**

load Test

hello

classLoader.MyClassLoader@6d06d69c

2.

public class Test {

static abstract class Animal{}

static class Human extends Animal{}

static class Man extends Human{}

static class Woman extends Human{}

public void SayHello(Animal guy){

System.out.println("hello animal");

}

public void SayHello(Human guy){

System.out.println("hello Human");

}

public void SayHello(Man guy){

System.out.println("hello Man");

}

public void SayHello(Woman guy){

System.out.println("hello Woman");

}

public static void main(String args[]){

Human man = new Man();

Animal Woman = new Woman();

Test test = new Test();

test.SayHello(man);

test.SayHello(Woman);

}

}

**输出：**

hello Human

hello animal

3.

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

import java.lang.reflect.Proxy;

public class DynamicProxyTest {

interface Ihello{

void sayHello();

}

interface Ibye{

void sayBye();

}

static class Hello implements Ihello,Ibye{

@Override

public void sayHello() {

// TODO Auto-generated method stub

System.out.println("hello world");

}

@Override

public void sayBye(){

System.out.println("Bye");

}

}

static class DynamicProxy implements InvocationHandler{

Object originalObj;

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

// TODO Auto-generated method stub

System.out.println("wlecome");

return method.invoke(originalObj, args);

}

Object bind(Object originalObj){

this.originalObj = originalObj;

return Proxy.newProxyInstance(originalObj.getClass().getClassLoader(),

originalObj.getClass().getInterfaces(), this);

}

}

public static void main(String args[]){

Hello orgin = new Hello();

Object proxy = new DynamicProxy().bind(orgin);

Ihello hello = (Ihello) proxy;

hello.sayHello();

Ibye bye = (Ibye) proxy;

bye.sayBye();

}

}

**输出:**

wlecome

hello world

wlecome

Bye

4.

public class WrapperClassTest {

public static void main(String args[]){

Integer a = 1,b = 2,c = 3,d = 3,e = 321,f = 321;

Long g = 3L;

System.out.println(c == d);

System.out.println(e == f);

System.out.println(c == (a + b));

System.out.println(c.equals(a + b));

System.out.println(g == (a+b));

System.out.println(g.equals(a + b));

}

}

**输出：**

true

false

true

true

true

False

5.

public class AutoboxingTest {

public static void main(String args[]) {

// Example 1: == comparison pure primitive – no autoboxing

int i1 = 1;

int i2 = 1;

System.out.println("i1==i2 : " + (i1 == i2)); // true

// Example 2: equality operator mixing object and primitive

Integer num1 = 1; // autoboxing

int num2 = 1;

System.out.println("num1 == num2 : " + (num1 == num2)); // true

// Example 3: special case - arises due to autoboxing in Java

Integer obj1 = 1; // autoboxing will call Integer.valueOf()

Integer obj2 = 1; // same call to Integer.valueOf() will return same

// cached Object

System.out.println("obj1 == obj2 : " + (obj1 == obj2)); // true

// Example 4: equality operator - pure object comparison

Integer one = new Integer(1); // no autoboxing

Integer anotherOne = new Integer(1);

System.out.println("one == anotherOne : " + (one == anotherOne)); // false

}

}

**输出：**

i1==i2 : true

num1 == num2 : true

obj1 == obj2 : true

one == anotherOne : false

6.

public class SingletonKerriganD {

/\*\*

\* 单例对象实例

\*/

private static SingletonKerriganD instance = null;

public static SingletonKerriganD getInstance() {

if (instance == null) {

synchronized (SingletonKerriganD.class) {

if (instance == null) {

instance = new SingletonKerriganD();

}

}

}

return instance;

}

}

假设线程一执行到instance = new SingletonKerriganD()这句，这里看起来是一句话，但实际上它并不是一个原子操作（原子操作的意思就是这条语句要么就被执行完，要么就没有被执行过，不能出现执行了一半这种情形）。事实上高级语言里面非原子操作有很多，我们只要看看这句话被编译后在JVM执行的对应汇编代码就发现，这句话被编译成8条汇编指令，大致做了3件事情：

1.给Kerrigan的实例分配内存。

2.初始化Kerrigan的构造器

3.将instance对象指向分配的内存空间（注意到这步instance就非null了）。

但是，由于Java编译器允许处理器乱序执行（out-of-order），以及JDK1.5之前JMM（Java Memory Medel）中Cache、寄存器到主内存回写顺序的规定，上面的第二点和第三点的顺序是无法保证的，也就是说，执行顺序可能是1-2-3也可能是1-3-2，如果是后者，并且在3执行完毕、2未执行之前，被切换到线程二上，这时候instance因为已经在线程一内执行过了第三点，instance已经是非空了，所以线程二直接拿走instance，然后使用，然后顺理成章地报错，而且这种难以跟踪难以重现的错误估计调试上一星期都未必能找得出来，真是一茶几的杯具啊。

DCL的写法来实现单例是很多技术书、教科书（包括基于JDK1.4以前版本的书籍）上推荐的写法，实际上是不完全正确的。的确在一些语言（譬如C语言）上DCL是可行的，取决于是否能保证2、3步的顺序。在JDK1.5之后，官方已经注意到这种问题，因此调整了JMM、具体化了volatile关键字，因此如果JDK是1.5或之后的版本，只需要将instance的定义改成“private volatile static SingletonKerriganD instance = null;”就可以保证每次都去instance都从主内存读取（volatile 变量在修改后必须立即同步回主内存），就可以使用DCL的写法来完成单例模式。

7.

public class TestNum {

// ①通过匿名内部类覆盖ThreadLocal的initialValue()方法，指定初始值

private static ThreadLocal<Integer> seqNum = new ThreadLocal<Integer>() {

public Integer initialValue() {

return 0;

}

};

// ②获取下一个序列值

public int getNextNum() {

seqNum.set(seqNum.get() + 1);

return seqNum.get();

}

public static void main(String[] args) {

TestNum sn = new TestNum();

// ③ 3个线程共享sn，各自产生序列号

TestClient t1 = new TestClient(sn);

TestClient t2 = new TestClient(sn);

TestClient t3 = new TestClient(sn);

t1.start();

t2.start();

t3.start();

}

private static class TestClient extends Thread {

private TestNum sn;

public TestClient(TestNum sn) {

this.sn = sn;

}

public void run() {

for (int i = 0; i < 3; i++) {

// ④每个线程打出3个序列值

System.out.println("thread[" + Thread.currentThread().getName() + "] --> sn["

+ sn.getNextNum() + "]");

}

}

}

}

**输出：**

thread[Thread-0] --> sn[1]

thread[Thread-2] --> sn[1]

thread[Thread-1] --> sn[1]

thread[Thread-2] --> sn[2]

thread[Thread-0] --> sn[2]

thread[Thread-2] --> sn[3]

thread[Thread-1] --> sn[2]

thread[Thread-0] --> sn[3]

thread[Thread-1] --> sn[3]

//代码原文地址http://blog.csdn.net/lufeng20/article/details/24314381/

注意：这里将ThreadLocal变量声明为静态的原因：

由于每个线程维护的Hash表是独立的，因此在不同的Hash表中，key值即使相同也是没问题的（注意上文提到的，ThreadLocal对象都包含了一个独一无二的threadLocalHashCode值，它其实是一个key，其对应的值要到线程中ThreadLoalMap中去取）。如果把ThreadLocal变量声明为非静态，则在类TestNum的每个实例中都会产生一个新对象，这是毫无意义的，只是增加了内存消耗。