JVM

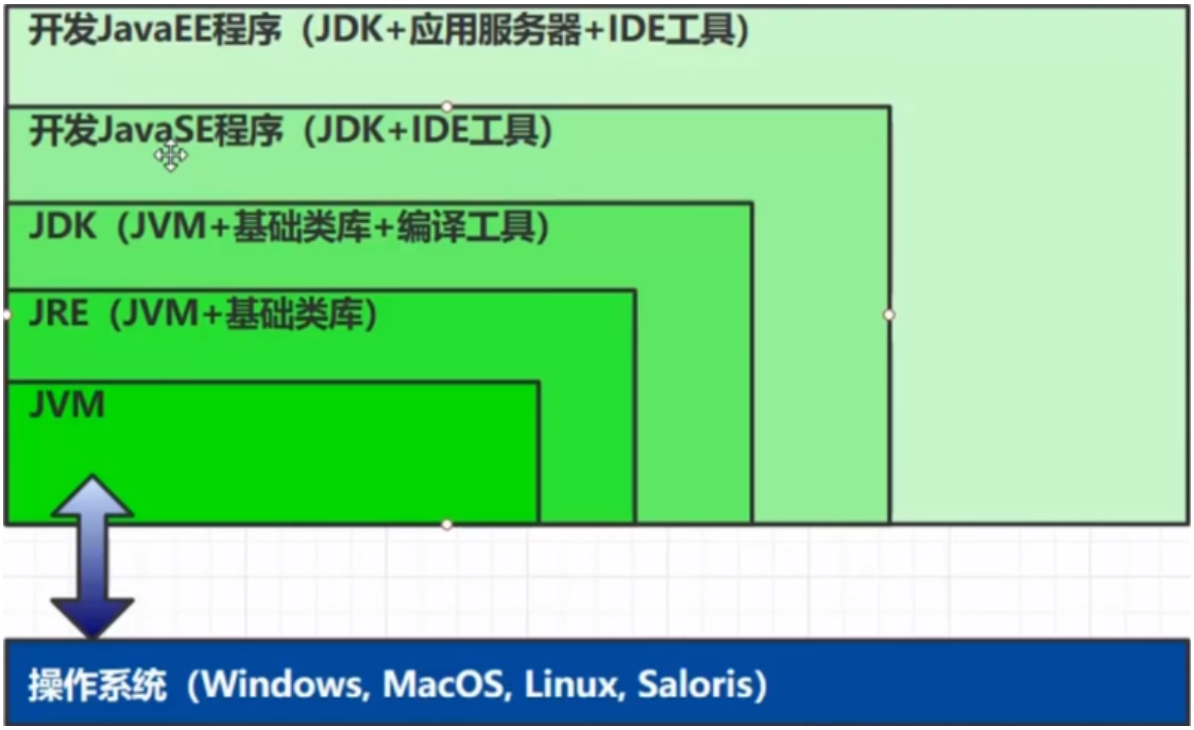
概述

定义：Java Vitual Mathince - java 程序的载体- 经过javac编译生成的 class字节码文件可以在其中运行

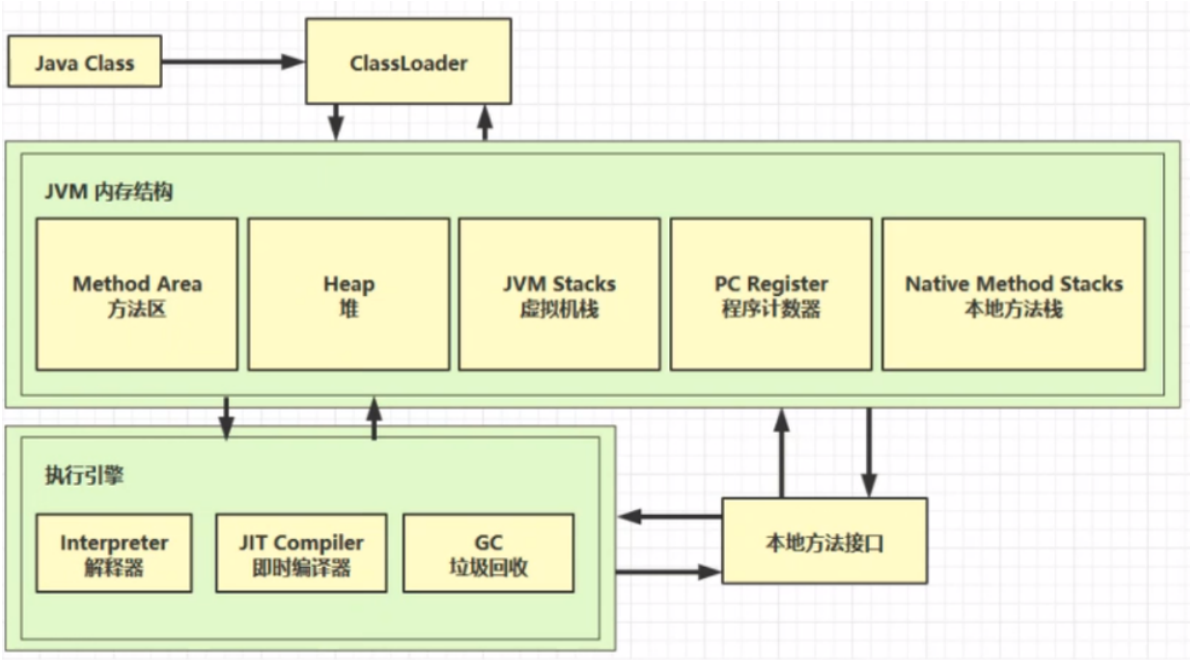
好处：

- 屏蔽了底层的操作系统，一次编写到处运行，也就是跨平台。
- 自动内存管理，垃圾回收功能，完爆了当年的C/C++,避免了程序员犯错误(没有释放导致内存泄露)。
- 一些严重问题的问题检查如数组下标越界，这比C/C++更安全。

JRE是java运行时环境包括了JVM和基本类库、JDK是开发工具包括了JRE和编译工具



JVM有多种实现，最常见的是Oracle实现的HotSpot。其核心部分包括：**类加载器**、**内存结构**、**执行引擎**



一个字节码文件通过类加载器加载到JVM方法区中，创建对象，执行方法会用到堆、栈、程序计数器等。方法执行时每行代码是由执行引擎解释，引擎还负责垃圾回收。一些java代码无法实现的功能，需要底层操作系统的介入，因此还有些本地方法的接口。

内存结构

一、程序计数器

java代码 编译生成字节码文件其中的内容是jvm指令。Program Counter Register 程序计数器是一个寄存器，在指令执行的过程中记住下一条JVM指令的地址。

特点:

1. 根据pc取到jvm指令，解释器将jvm指令解释为机器码交给cpu后再pc+1，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 每个线程的指令序列和执行进度都不同，故每个线程有自己私有的程序计数器，彼此互不干扰，在线程切换时要切换pc。
3. 唯一一个不会存在内存溢出的区域

二、虚拟机栈

1、定义

栈-每个运行线程运行需要的内存空间，每个栈由多个栈帧组成。而栈帧是每个方法运行时需要的内存空间，里面有参数，局部变量，返回地址等。每个线程只能由一个活动栈帧，对应着当前线程正在执行的那个方法。实际上每个栈帧还可以细分为局部变量表、操作数栈等部分，指令的执行往往是将局部变量表中的内容压入操作数栈中计算后再写回局部变量表。

Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

2、问题辨析

垃圾回收是否会涉及栈内存?

不会。栈内存中的栈帧对应一次次方法调用，在每一次方法返回时栈内存会自动释放掉当前栈帧。垃圾回收针对的是堆内存中的无用对象。

Stack Space栈内存越大越好吗?

不是的。虚拟机总内存不变，栈内存越大的确可以支持更多次的方法调用更深的递归，但同样意味着可支持的线程总数减少了。linux，Solaris等默认是1MB,Window根据虚拟内存分配。

方法内的局部变量是否线程安全?

是。一个线程对应一个栈，它们执行同一个方法时会在各自的栈中开辟一个新的栈帧，局部变量在不同栈的栈帧中属于线程私有。但若是局部变量(引用类型)作为方法的参数或者返回值就不再是线程安全的了，因为很可能被发布到其他线程中去。static变量不是线程安全的，因为它是所有线程公有的。

栈内存溢出StackOverFlowError错误何时发生?

调用了大量的方法、递归调用没有出口时开辟的栈帧过多会导致栈内存溢出。栈帧过大也会导致。

注意，栈内存溢出不限于我们自己写的程序，第三方的工具包也可能导致。例如 两个类循环引用时，json解析会产生infinite recursive

3、线程运行诊断

案例1：cpu占用居高不下

在linux下通过**top**命令可以检测各个进程对cpu和内存的占用情况。它能够定位到进程号，但是定位不到线程

通过**ps** H -eo pid,tid,%cpu | grep 32655 能就看到进程63655的所有线程对cpu 的占用情况。

通过**jstack** 根据进程id32655显式进程32655中线程的状态详情，进一步定位到问题线程中问题代码所在的源码行数。

案例2：程序运行很久都没有结果

通过**jstack** 根据进程id32752显示其中的线程的状态情况，也许就会发现原来是有几个线程发生死锁deadlock，进一步定位到引发死锁的问题代码。

三、本地方法栈

本地方法是指不是由java编写的代码，即java代码会调用本地方法的（C/C++）native方法间接地与操作系统打交道。

本地方法栈是指这些本地方法运行时所用的空间，也是每个线程私有的。常见的本地方法：clone()、wait()、hasCode()、notify()

四、堆

1、介绍

Java 虚拟机所管理的内存中最大的一块，Java 堆是所有**线程共享**的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。

随着JIT即时编译的发展，所有对象都在堆上也不那么绝对了从jdk 1.7开始已经默认开启**逃逸分析**，如果某些方法中的对象引用没有被返回或者未被外面使用（也就是未逃逸出去），那么对象可以直接在栈上分配内存。

Java 堆是**垃圾收集**器管理的主要区域，因此也被称作GC 堆（Garbage Collected Heap）.从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代：新生代还有：伊甸园、幸存者空间等。

进一步划分的目的是更好地回收内存，或者更快地分配内存。

2、堆内存溢出

垃圾回收机制会回收掉堆中长时间没有被使用的对象，但如果堆中对象一直被使用无法被回收，并且持续创建新的对象，内存就可能溢出堆内存大小默认是4G。发生OutOfMemoryError：Java heap spce。

五、方法区

1、介绍

方法区与 Java 堆一样，是各个线程**共享**的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、编译后的指令等数据。

虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是在实现上可以选择与堆分开。很多JVM都没有把方法区放在堆中，HotSpot的JVM的方法区在JDK1.8之前位于堆内存中，用**永久代**作为实现方式。在1.8之后将永久代移除了，方法用**元空间**作为实现方式。元空间不再使用JVM的内存而是本地内存-操作系统的内存且不设上限。

2、方法区内存溢出

默认情况下，CMS不会对方法区（永久代或元空间）进行垃圾回收，但是如果方法区满了，就会启动一个Full GC 回收掉所有没有用到的类。垃圾回收器超过98%的时间用来垃圾回收，但回收不到2%的内存时，相当于癌症晚期GC会认定此时为内存溢出。

1.8之前会导致永久代内存溢出，PermGen space1.8之后会导致元空间内存溢出。
OutOfMemoryError：MetaSpace

整个永久代使用堆内存受限于JVM使用内存的上界，相比之下元空间使用的是计算机的直接内存，可用的空间更大，可以加载的类更多，发生方法区内存溢出的机会更小。

场景：Spring框架 mybatis框架 都会用cglib的技术生成大量的代理类或者Dao接口的实现类的字节码，完成类加载，导致方法区溢出

3、常量池与运行时常量池

常量池：是Class文件中的信息，用于存放编译期生成的各种字面量和符号引用。

运行时常量池：运行时常量池是方法区的一部分。当该类被加载到虚拟机之后，这个类的常量池信息就会放入方法区内存的运行时常量池。

案例：运行时常量池中的字符串常量池StringTable

```
//常量池中的信息都会被加载到运行时常量池中，加载后它们仅仅是常量池中的符号，还没有变为字符串对象----字符串加载的延迟特性
//等到具体的代码执行时，用a在StringTable(串池是个hashTable。不能扩展)中找是否存在“a”对象，如果没有就放入，如果有就什么都不做
String s1 = "a";//然后让s1引用指向这个对象。
String s2 = "b";
String s3 = "ab";
String s4 = s1 + s2;//new StringBuilder().append("a").append("b").toString() 相当于运行时的new String("ab");在堆中，串池中没有
System.out.println(s3==s4); //s3在串池中， s4在堆中,结果为false
String s5 = "a" + "b";//javac在编译期的优化，结果已经在编译期间确定为ab，并不是先找a再找b，而是直接在串池中找到"ab"
System.out.println(s3==s5);//结果为true
```

常量池中的字符串仅仅是符号，第一次用到时才变为对象。利用串池的机制，可以避免重复创建字符串对象

字符串变量拼接的原理的StringBuilder(1.8以后)，在堆中；字符串常量拼接的原理是编译期优化，在串池中

只要是字符串可以确定下来，串池中必有对象；只要是new ，堆中必有对象。

```
//["a","b"]
String s = new String("a") + new String("b");
//new String("x")时会在串池生成一个"x"对象，又在堆中生成一个"x"对象,返回的是堆中的引用。
//堆中 “a” ，“b”，“ab”
String s2 = s.intern();//["ab","a","b"]
System.out.println(s2 == "ab");// true
System.out.println(s == "ab");//true
```

1.8以后可以使用intern方法，主动**尝试**将串池中还没有的字符串对象**移入**串池,最终返回串池中的对象

```
String x = "ab";//["ab"]
String s = new String("a") + new String("b");
//["ab","a","b"]
//堆中 “a” ，“b”，“ab”
String s2 = s.intern();//1.6的结果全都是false
System.out.println(s2 == x);// true
System.out.println(s == x);//false
```

StringTable的位置

是运行时常量池的一部分，1.6之前方法区以永久代实现在堆中，永久代的触发时机较晚，回收效率低。虽然1.7之后方法区的实现变为了元空间，但StringTable使用是非常多的，因此StringTable还在堆空间中，提高了回收效率，减轻了内存的负担。

符串常量非常多的情况下，应该尽量将字符串从堆的普通区域进入到池中，因为堆中的字符串对象可能存在大量的重复冗余占用过多内存，而池中的串有唯一性，值相同的字符串保证只占用一份内存从而节省大量内存。

其他的常量池

Java 基本类型的包装类的大部分都实现了常量池技术以节省运行期每次创建都消耗时间，即**Byte,Short,Integer,Long,Character,Boolean；**前面 4 种包装类在类加载阶段默认创建了数值[-128，127] 的相应类型的缓存数据，Character创建了数值在[0,127]范围的缓存数据，Boolean 直接返回True Or False。**浮点没有**

六、直接内存

定义：并不是JVM的内存，而是操作系统的内存。常见于NIO操作时，用于数据缓冲区（如DirectByteBuffer.allocateMemory）。它可以直接使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行直接操作。**避免了在 Java 堆和 Native 堆之间来回复制数据**，虽然分配回收成本较高，但读写性能更高速度更快，这样就能在一些场景中显著提高性能，不受JVM垃圾回收的管理。

直接内存回收

实际上DirectByteBuffer底层使用了Unsafe对象和其方法完成直接内存的分配和回收。一些分配allocateMemory、释放直接内存FreeMemory的事儿都是使用Unsafe对象实现的。Unsafe对象只能通过反射的方法得到。

ByeteBuffer的实现类内部，使用了Cleaner(虚引用)来监测ByteBuffer对象，这个对象一旦被垃圾回收了，那么就会有ReferenceHandler线程通过Cleaner的clean方法调用FreeMemeory来释放直接内存。

System.gc()这种显示的垃圾回收的意思是建议启动垃圾回收，Full GC 性能比较差，

垃圾回收

垃圾回收是JVM内存管理机制最具特色的部分。与运行在普通操作系统上的C不同，运行在JVM上的java不必我们手动地去回收使用的内存，因为内存泄漏而担惊受怕。

Java 堆是**垃圾收集**器管理的主要区域，因此也被称作GC 堆（Garbage Collected Heap）.从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以 Java 堆还可以细分为：新生代和老年代：新生代还有：伊甸园、幸存者空间等。

进一步划分的目的是更好地回收内存，或者更快地分配内存。

一、判断对象是否可回收

引用计数法：一个对象每被引用一次，计数加一，当引用为0时，触动垃圾回收。早期python虚拟机就是使用这种方式

可达性分析法：这个算法的基本思想就是通过一系列的称为 **“GC Roots”** 的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，引用链上的对象被判定为存活，除了它们其他对象都默认判定为死亡，需要进行垃圾回收。

哪些对象可以作为GC root？ MAT可视化工具观看

启动时就加载的Class对象、线程虚拟机栈上的引用变量、同步时被获取内置锁的对象、本地方法栈中引用的对象等

Accelerator的回答 - 知乎 https://www.zhihu.com/question/50381439/answer/120846441

五种引用

Java中提供这四种引用类型主要有两个目的：第一是可以让程序员通过代码的方式决定某些对象的生命周期；第二是有利于JVM进行垃圾回收。

https://blog.csdn.net/qq_39192827/article/details/85611873

```
public static void main(String[] args){
   List list = new ArrayList();
   for(int i = 0 ; i < 10 ; i ++){
       //无法回收强引用的对象,会内存溢出的问题
       list.add(new byte[1024*1024*1024]);
       System.out.println(list.size());
   }
}
```

```
List<SoftReference<byte[]>> list = new ArrayList<>();

  ReferenceQueue<byte[]> queue = new ReferenceQueue<>();
 for(int i = 0 ; i< 10 ; i++){
    //软引用指向的对象会在内存不足时被垃圾回收，软引用自身可以放入引用队列中
     SoftReference ref = new SoftReference(new byte[1024*1024*1024],queue);
     list.add(ref);
     System.out.println(list.size());
 }
 for(SoftReference<byte[]> ref : list){
    //发现很多ref都已经被回收，为null了
     System.out.println(ref.get());
 }
    //从队列中获取无用的软引用对象进行移除
    Reference< byte[]> poll = (Reference<byte[]>) queue.poll();
     while (poll!=null){
        list.remove(poll);
```

```
        poll = (Reference<byte[]>) queue.poll();
     }
    //弱引用指向的对象会被垃圾回收器的定期扫描所回收，
    weakReference ref = new WeakReference(new byte[1024*1024*1024]);
```

1、强引用：以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

2、软引用：如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。可以和一个引用队列（ReferenceQueue）联合使用

3、弱引用：不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。可以和一个引用队列（ReferenceQueue）联合使用来释放弱引用自身。

4、虚引用：必须配合引用队列（ReferenceQueue）使用。虚引就是形同虚设，与其他几种引用都不同，她并不会决定对象的生命周期。对象在任何时候都可能被垃圾回收。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序如果发现某个虚引用已经被加入到引用队列，那么就可以利用虚引用在对象的内存**被回收之前采取必要的行动。**

5、终结器引用：无需手动编码，但其内部配合引用队列使用，在第一次垃圾回收时终结器引用入队（被引用的对象尚未被回收），再由Finalizer线程桶过终结器引用找到对象并调用它的finalize方法，第二次GC时才能回收被引用对象。**类似与虚引用的具体实现**

finalize方法：当对象变成(GC Roots)不可达时，GC会判断该对象是否覆盖了finalize方法，若未覆盖，则直接将其回收。否则，若对象未执行过finalize方法，将其放入F-Queue队列，由一低优先级线程执行该队列中对象的finalize方法。执行finalize方法完毕后，GC会再次判断该对象是否可达，若不可达，则进行回收，否则，对象“复活”。

```
public class GC {
    public static GC SAVE_HOOK =null;
    public static void main(String[] args) throws InterruptedException{
        SAVE_HOOK = new GC();
        SAVE_HOOK = null;
        System.gc();
        Thread.sleep(500);
        if(null!= SAVE_HOOK){
            System.out.println("Yes , I am still alive");
        }else{
         System.out.println("No , I am dead");
        }
        SAVE_HOOK=null;
        System.gc();
        Thread.sleep(500);
        if(null!=SAVE_HOOK){
            System.out.println("Yes , I am still alive");
        }else{
            System.out.println("No , I am dead");
        }
    }
    @Override
    protected void finalize() throws Throwable {
        super.finalize();
```

```
            System.out.println("execute method finalize");
            SAVE_HOOK = this;
        }
    }
```

二、垃圾回收算法

标记&清除：Mark Sweep

- 从GC root出发首先标记出所有不需要回收的对象，在标记完成后统一清除所有没有被标记的对象
- 清楚是不是清零而是意味着将原有对象占用的地址放入空闲地址的列表里，随后这个地址可以供其他对象使用。

优点:清除操作简单，速度快。缺点：容易产生内存的碎片化，内存利用率降低

标记复制算法：Copy 适用于存活**较少**

- 将堆内存分为大小相同的两块，每次只使用其中一块。
- 触发垃圾回收时将仍存活的对象复制到另一块内存中，然后交换使用内存块如此循环往复

优点：清除非常高效，没有碎片化。缺点：只能使用原来的一半内存。

标记&整理：Mark Compact 适用于存活**较多**

因为前面的复制算法当对象的存活率比较高时，这样一直复制过来，复制过去，没啥意义，且浪费时间。

- 标记：从GC root 出发标记出所有存活的对象。
- 整理：让存活的对象，向内存的一端做移动紧缩，然后回收掉没有用的内存空间。

分代垃圾回收：HotSpot 为什么要分为新生代和老年代?

当前虚拟机的垃圾收集都采用分代收集算法，因为单一的回收算法无法适应对象生命周期的差异性，将它们分开使用不同的算法能够发挥不同算法各自的优势，大大提升效率。一般将 java 堆分为新生代和老年代，根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择"标记-复制"算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，所以我们选择"标记-清除"或"标记-整理"算法进行垃圾收集。

具体流程

对象诞生在新生代的伊甸园中，伊甸园占满时会触发垃圾回收，新生代的垃圾回收叫做Minor GC，使用可达性算法从GCroots找到存活的对象，**复制**到幸存区to。然后让这些幸存对象的寿命+1.**交换幸存区from和to的指针。**伊甸园又占满时，第二次Minor GC 将伊甸园和幸存区from中存活的对象**复制**到幸存区to 并讲寿命+1 再次交换幸存区from和to的指针。当幸存区中的对象寿命超过了一个阈值**15**，代表该对象比较重要，不应轻易回收掉。那么就讲这些对象从幸存区晋升到老年代。当老年代的内存不足时，触发Full GC 做一个整体的清理

minor GC/full GC都会触发一次 stop the world （**STW**）触发垃圾回收时 的Java中一种全局暂停现象，会暂停其他所有的用户线程，因为垃圾回收可能牵扯到对象的移动和删除，如果线程不停止还使用原来的地址会出错。

native代码可以执行，但不能与JVM交互。

minor GC的STW的时间非常短，因为新生代的对象大部分都是垃圾，存活的少因此复制的也少。Full GC 引发的STW的时间更长。

大对象直接进入老年代，当新生代不足以容纳大对象，大对象会直接进入老年代，不会触发GC，若是老年代也无法容纳该对象，会进行一个FULL GC，还是不够的话会爆出堆空间溢出。

三、垃圾回收器

1、串行的垃圾回收器

- 单线程负责垃圾回收
- 适用于堆内存较小、Cpu核数较少时，适用于**Client**

2、吞吐量优先

- 多线程
- 适用于堆内存较大、多核Cpu的情况。适合**Server**。如果没有多核Cpu(多把扫帚)，多线程还不如单线程效率高
- 让单位时间内总STW时间尽可能短。0.2 + 0.2 = 0.4

3、响应时间优先

- 多线程
- 适用于堆内存较大、多核Cpu的情况。**Server**
- 让每一次的STW的时间尽可能的短　　0.1+0.1+0.1+0.1+0.1 = 0.5

- **-XX:UseSerialGC**

  单线程垃圾回收器，它只会使用一根儿垃圾回收线程去完成GC工作，并且在该线程工作时其他工作的用户线程均阻塞，STW直到GC结束，Serial + SerialOld 分别工作在新生代和老年代，分别使用复制和标记整理算法

- **-XX：UserParalleGC 1.8默认的**

  并行的多线程收集器，它们的侧重点是**吞吐量**。触发GC时所有用户线程在安全点停下，多个垃圾回收线程多管齐下标记+清除(垃圾回收时CPU占用率几乎为100%。)，STW直到GC结束。Parallel Scavenge+ParallelOldGC分别工作再新生代和老年代，分别使用复制和标记整理算法。

- **-XX：+UseConcurrentMarkSweepGC　CMS老+Parnew新**

  并发的垃圾收集器，它们的侧重点是**响应时间**。流程包括：初始标记stw、并发标记、重新标记stw、并发清除，新生代和老年代分别采用标记复制和标记清除算法。由于该垃圾回收的某些阶段可以和用户线程并发执行会导致总的垃圾回收吞吐量降低，但每次STW的时间很短，因此CMS加快了响应的时间。

- **Garge First G1**

JDK1.9开始默认，Garge 1 取代了CMS

唯一一个可以同时用于年轻代和老年代的垃圾回收器，采用标记整理算法避免碎片，也是一种并发的垃圾回收器。它将整个堆分成了很多个固定大小的Region，化连续为离散。

具体流程包括：初始标记stw、并发标记、重新标记stw、筛选回收。即每次在用户设定的优先收集时间内，优先选择回收价值最大的region回收，而不是回收全部。因此这种垃圾回收器同时兼顾了吞吐量和响应时间。

- **重新标记**

由于并发标记时也伴随着其他用户线程的并发，对象之间的引用关系可能发生变化。当对象的引用状态发生了变化时，写屏障指令会触发将对象的引用加入到一个队列satb_mark_queue当中重点关注，即在并发标记处理完后会STW对队列中的所有对象再进行一次重新标记。

6）Young Collection跨代引用

若新生代对象的根对象是老年代，老年代对象很多，寻找的效率很低。我们将老年代区域进行细分成卡表，若是有引用时就将老年代对象所在的cut标记为脏cut，在之后的寻根时期只关注脏卡即可。

9）JDK 8u40 并发标记类卸载

所有对象都经过并发标记后，就能知道那些类不再被使用，当一个类所在的类加载器其中的所有类都不再使用（类的实例全部被回收掉了）），则卸载整个类加载器中的所有的类。条件比较苛刻，

10）JDK8u60回收巨型对象

当一个对象大于region 的一半时，称之为巨型对象，G1对巨型对象不会进行拷贝，回收时也被优先考虑，G1会跟踪它来自老年代的所有incoming引用，这样来自老年代incoming引用为0的巨型对象就可以再新生代垃圾回收时处理掉

11）JDK 还在不断为G1做出增强和修复

垃圾回收算法越来越智能，采样后调整，避免Full GC

四、GC调优

1）选择合适的 GC 回收器。根据应用对响应时间、吞吐量的要求，合理选用。推荐使用 G1 替换 CMS 垃圾回收器。Java 11 里新引入的 ZGC 垃圾回收器，基本可用做到全阶段并发标记和回收，因此出现 Stop The World 的情况会更少！

2）合理的堆内存大小设置，堆大小不要设置过大，建议不要超过系统内存的 75%，避免出现系统内存耗尽。响应时间和吞吐量，这两个指标其实是有冲突的。YONG GC占GC的很大一部分，因此新生代的调整是 GC 调优的核心，新生代过小会导致Young GC 频率高，单位时间内STW的时间就会长吞吐量不达标，新生代过大，每次Young GC 时间都长，每次STW时间长，导致响应时间不达标，应该根据实际情况进行折中取一个合理的新生代（大小、伊甸园与幸存区的比例）。

3）降低 Full GC 的频率。如果出现了频繁的 Full GC 或者 老年代 GC，很有可能是存在内存泄漏，导致对象被长期持有。除此之外，新生代和老年代的比例不合适，导致对象频频被直接分配到老年代，也有可能会造成 Full GC，这个时候需要结合业务代码和内存快照综合分析。

类加载与字节码技术

一、类文件结构

魔数信息占4字节，魔数 用于区分文件类型，字节码文件的魔数是cafebabe

类的版本号 00 34(52)表示jdk 8

常量池部分占据了很大比重{其他字段用到的常量都出自于它}，里面也有引用其他地方的内容

类中access flag 访问标识符号与this、super等继承信息 共6个字节

类中Field 成员变量信息(个数、详情)

方法字节码类中Method 方法信息（个数、详情（访问修饰符，名称，参数，局部变量等））linerTable 行号表 LocalVaiableTable作用(方法)范围

附加属性源文件的信息，Hello.java

https://snailclimb.gitee.io/javaguide/#/docs/java/jvm/%E7%B1%BB%E6%96%87%E4%BB%B6%E7%BB%93%E6%9E%84

二、字节码指令

2a b7 00 01 b1　　某一构造方法的二进制字节码指令

解释器将这些平台无关的字节码指令解释为机器码

反编译 javap -v Helloworld.class 会输出二进制字节码的友好阅读版本

类加载就是把字节码文件中的内容读入内存中。常量池内的内容放入方法区的运行时常量池，方法相关指令放入方法区。由解释器将指令字节码指令解释为机器码，指令执行引擎取值执行每一条指令进行线程栈内和堆内的内存使用。

注意，栈的内存还分为局部变量表(每个变量名都是代表了一个槽位)和操作数栈。 load是将槽位中的数读入操作数栈。

a++与++a的案例：区别在于iloac和iinc哪个先执行

```
int i = 0 ;      赋值在局部变量表中赋值
int x = 0 ;
while(i < 10 ){
    x = x ++; 自增是对局部变量表中进行自增，而本次赋值是将操作数栈中内容赋给了局部变量表
    i ++;
}
System.out.println(x);//结果是0
```

静态代码块和实例代码块的作用---编译期生成的方法

编译器会从**上至下**，收集类中所有static静态代码块和static静态成员赋值的代码，合并成一个特殊的方法()V：而该方法也是一系列字节码指令，会在**类加载阶段**被调用。静态变量的引用在方法区，值在堆区，静态的东西都只有一份。

编译器会从**上至下**，收集所有实例代码块和实例成员变量赋值的代码，形成一个新的构造方法init(但原始构造方法内的代码总是在**最后**),该方法在调用构造方法时被调用。

方法调用：不同方法的字节码

```
public class Demo3_9 {
    public Demo3_9(){ }
    private void test1(){ }
    private final void test2(){ }
    public void test3() { }
    public static void test4(){ }

    public static void main(String[] args) {
        //invoke special 静态绑定：在字节码指令生成的时候，就完了绑定，因为这些方法一定当前类调用的。
        //编译时直接确定方法的入口地址
        Demo3_9 d = new Demo3_9();//new 首先会在堆中分配对象内存，成功后将对象引用放入操作数栈 *2份；invoke special 会用栈顶的那份引用调用构造方法，剩下的那一份引用会赋给操作数d。
        d.test1();//从局部变量表中load d 调用test1
        d.test2();
        //invoke virtual 普通的public ，有多态，在编译期间不能确定自己调用的是谁(子、父)的方法，
        //因此需要在运行时动态绑定，确定方法的入口地址
        d.test3();
        //invoke static 静态绑定
        d.test4();//load d到操作数栈后 发现这是静态方法，又pop出去了，再调用invokestatic
        Demo3_9.test4();
    }
}
```

对象底层其实都是C++的Structure，里面有对象头16字节（8markword包括哈希码，锁标记，8klass对象的类型指针64位不压缩就是8），对象其他的数据。通过类型指针能找到方法区的类的详细信息

多态的原理

类的结构，字节码文件中有一张虚方法表，之所以称为虚方法是因为在编译过后并没有确定方法具体的执行指令，即编译过后只有个方法的头衔，而头衔下面没有绑定方法的入口地址，需要运行时通过invokevirtual根据虚方法表中的映射关系找到具体的入口地址进而执行。

多态的实现就是基于这种机制，完成了运行时通过父类引用调用其子类的方法。当执行invokevirtual指令时：

1）先通过栈帧中的对象引用找到堆中的对象

2）分析对象头、找到对象实际的Class

3）Class结构中有vtable，它在类加载的链接阶段就已经根据方法重写规则生成好嘞

4）查表得到具体方法指令的入口地址

5）执行方法的字节码

步骤繁琐因此，效率细微地比static、spectial慢，但是JVM对虚方法表也会做出优化，如果多次反复调用对象的虚方法，那么会从缓存中找到入口地址，不必每次都多次寻址。

异常的处理

字节码中的Exception table 会监测try代码块中的代码（from....to.....之间的字节码指令），如果发生了某种异常，会跳转到target异常处理的指令位置。执行异常处理指令时会将异常对象的引用放到局部变量表中的槽位用于后续使用、做catch块中的操作。

当多个single catch 时，它们发生异常后跳入的target不同，有各自的处理方式。

multi-catch语法，catch（异常|异常|异常），它们发生异常之后跳入的target相同。

finally块对应的字节码，会出现多份在每个分支，在try块指令末尾和catch块指令末尾保证自身必然会执行。注意，catch未必能捕获所有的异常和错误，对于它们Exception table 中仍然会为其设置跳转（catch any）的指令行数，即仍要保证finally块被执行。

案例

```
public static void main(String[] args) {
    int result = test();
    System.out.println(result);
    //无论结果有没有异常，操作数栈顶的都是20，return做的就是返回栈顶
}

    private static int test() {
    try{
        return 10;//10进入操作数栈，20也进入操作数栈。
    }finally {
        return 20;//不要再finally中写return ，会吞掉athrow指令。
        //由于未捕获的异常进入带有return的finllay块后，异常不会被再度抛出。
    }

     private static int test() {
        try{
            int i = 1/0;
        }finally {
            return 10;
        }
    }
        //try中return了，再在finally中进行修改不会影响到返回结果。它在try块中对return的值进行了暂存，再去执行finally中的代码，执行完之后再将暂存的值恢复到栈顶。此时操作数栈栈顶是10，slot槽位中是20，最终返回栈顶。
```

```
    private static int test() {
        int i = 10;
        try{
            return i;//返回10
        }finally {
            i+=10;
            System.out.println(i);//输出20
        }
    }
}
```

synchronized如何加锁解锁

同步代码块经过编译，会展成在含有开始执行monitorenter指令对某个对象进行加锁的操作，结束时含有monitorexit对某个对象进行解锁的操作。这是不发生异常的情况。

保证解锁：如果出现了异常，后面的异常表会指引指令流去执行异常处理操作，其中就含有monitorexit的解锁操作，并且会抛出异常。

锁对象的引用始终留有一份在局部变量表中等待monitor指令的调用（进入操作数栈）。

三、编译器处理

语法糖：编译器在把.java文件编译成.class字节码的过程中，自动生成和转换了一些代码，做出了一些优化。

1、编译器为我们生成默认的无参的构造器，调用父类的无参构造方法。

2、JDK5加入，基本类型到包装类型的自动的拆装箱。在这之前需要手动.，泛型也是JDK5以后加入的

3、泛型擦除是在编译器将泛型的有关信息去掉，在字节码中首先将它们都作为Object处理，随后又加上了强制类型转换(转成泛型)的指令。

```
int x  = ((Integer)List.get(0)).intValue();
```

4、局部变量的泛型信息做了擦除，因此通过反射机制拿不到局部变量的的泛型信息。但是方法的返回值和参数的泛型信息可以。

5、可变参数，String...args 会被编译器根据实参的数量自动转化为一个长度正好匹配的 String[] args

6、foreach对数组遍历时会被编译器自动转化为按下标遍历的 for（int i = 0 ; i < len ; i++）。而对集合遍历时会被编译器优化为迭代器式的遍历。

7、switch从JDK7开始支持了枚举类和字符串，底层实现共执行了两遍switch，第一遍根据字符串对象的hasCode方法和equals方法判断分支将对象转换为byte类型的临时变量，再根据临时变量的值进行第二便switch决定是哪一个分支。使用hasCode()是为了快速定位大概的范围提高比较效率，equals是为了在hasCode冲突后需要做出进一步的比较判断。

枚举类的switch会被编译器优化，在原有的类中定义一个内部的合成类（JVM使用，我们不可见），在该静态内部类中完成枚举类序号ordianl到整数的映射，正式的switch是用各个不同名的枚举成员所对应的整数完成的。

8、枚举类的底层实现是final的，因此无法被继承。枚举成员也会被声明为static final的，它们是在static代码块中赋的初值，初值包括它的名字，和ordinal枚举成员的序号。可以通过枚举成员的名称得到想获取的枚举的实例对象。

9、JDK7新增 （**twr**）try with resources允许我们不必显式地对资源对象进行释放，如输入输出流、连接、statement，前提是它们需要都实现AutoCloseable接口。编译器会为我们在末尾自动生成finally代码块对资源进行关闭和释放。

10、方法重写时，父子类的返回值一致/子类的返回值是父类返回值的子类，后者实现时java编译器会生成合成方法，作为桥接手段使其符合重写的规则。

11、匿名内部类在编译之后，会额外生成一个独立的类，和非内部类同级，原内部类引用的外部类中的局部变量（final防止局部变量和匿名内部类的属性不一致）会作为参数传给这个额外生成类的构造方法。

四、类加载

1、加载阶段

1. 通过全类名获取定义此类的二进制字节流，即将类的字节码文件，可从jar\war中得到
2. 将字节流所代表的静态存储结构转换为方法区的运行时数据结构，本质上是C++的instanceKlass
3. 因为java代码无法直接访问C++对象，因此在内存中生成一个代表该类的 java 对象即Class对象，作为这些数据的访问入口。

   类镜像就是将klass暴露给java程序如String.class 它和instaceKlass互相持有对方的指针。

2、链接阶段：加载阶段和连接阶段的部分内容是交叉进行的，加载阶段尚未结束，连接阶段可能就已经开始了。

1、验证：检查类的字节码文件格式、语义、符号引用是否符合JVM规范。

2、准备：为静态变量分配空间,设置初始值，通常情况下的初始值是数据类型默认的零值，只有在编译器就确定好的常量（final或String）才能在准备阶段直接赋值。

3、解析：将常量池中的符号引用替换解析为直接引用。在程序实际运行时只有符号引用是不够的，系统必须需要明确知道属性字段或者方法指令所在的位置。

3、初始化阶段

初始化阶段调用`<clinit> ()`为静态变量完成初始化赋值操作，对于`<clinit> ()`方法的调用，虚拟机会自己确保其在多线程环境中的安全性。

发生的时机： new 、 getstatic、putstatic或invokestatic

- new一个对象的时候，如果类还没有初始化
- 访问一个类的静态变量或者静态方法
- 利用反射机制获得对象时
- Main主线程所需要的类，子类的父类最先被初始化。

不发生的时机：

- 当访问一个静态域时，只有真正声明这个域的类才会被初始化。例如：通过子类引用父类的静态变量，不会导致子类初始化。
- 引用常量不会触发此类的初始化（常量在编译阶段就存入常量池中了）。
- 通过数组定义类引用，不会触发此类的初始化。

总之，初始化是懒惰的。

如果采用**饿汉模式**，单实例对象便会在类加载完成之时，常驻堆中，后续访问时本质上是通过该类的Class对象嵌入的intance指针寻址，找到单实例对象的所在。好处在于：

1、通过空间换取时间，避免了后续访问在构造对象上花费时间

2、无需考虑多线程并发问题，JVM在类加载过程中，通过内部加锁机制保证加载类的全局唯一性。

不好的地方就是，无论是否使用，只要完成了类加载就占用了内存空间，也是一种内存泄露的体现。

懒加载的单例模式

```
class Singleton{
    public static void test(){
        System.out.println("test");
    }
    static{System.out.println("hhhhh");}
    private Singleton(){  }
    private static class LazyHolder{
        static final Singleton SINGLETON = new Singleton();
        static{
            System.out.println("lazy holder init");
        }
    }
    public static Singleton getInstance(){
        return LazyHolder.SINGLETON;
    }
}
```

五、类加载器

Bootstrap ClassLoader 启动类加载器

是最顶层的类加载器，C++实现，除了 BootstrapClassLoader 其他类加载器均由 Java 实现且全部继承自`java.lang.ClassLoader`

负责加载JAVA_HOME/jre/lib下的jar包类，但也可以把其他类交给它加载

Extension Classloader 扩展类加载器

加载JAVA_HOME/jre/lib/ext下的jar包和类，当然也可以把其他类交给它

Application Classloader 应用类加载器、自定义类加载器

面向我们用户的加载器，负责加载当前应用classpath下的所有jar包和类。

双亲(上级)委派模式

系统中的 ClassLoder 在协同工作的时候会默认使用 **双亲委派模型** 。即在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。加载的时候，首先会把该请求委派该父类加载器的 `loadClass()` 处理，因此所有的请求最终都应该传送到顶层的启动类加载器 `BootstrapClassLoader` 中。当父类加载器无法处理时，才由自己来处理。当父类加载器为null时，会使用启动类加载器 `BootstrapClassLoader` 作为父类加载器。

`AppClassLoader`的父类加载器为`ExtClassLoader`， `ExtClassLoader`的父类加载器为null，**null并不代表`ExtClassLoader`没有父类加载器，而是`BootstrapClassLoader`** 。

双亲委派模型最大的好处就是保证稳定运行，可以避免类的重复加载。因为java中的类使用包名、类名、类加载器联合作为唯一性约束。如果没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现类名相同的类。比如我们自己写一个 `java.lang.Object` 类的话，再用自己的加载器加载，系统就会出现多个不同的 `Object` 类。

六、运行时优化

在运行期间，即时编译器C1与C2会对代码进行优化：将少部分的热点代码（方法调用、循环遍历的次数达到一定阈值）编译为机器码，以达到理想的运行速度。Hotspot指的就是这种特性

解释器< C1 五倍<C2 十~一百倍的优化

逃逸分析：C2的优化分析new Object对象创建之后会不会被外部使用，优化过的字机器码可能已经面目全非，不区创建无用的对象了

方法内联：将不太长的热点方法的代码，内联（代码拷贝）到该方法调用者的位置从而不必进行方法调用时的栈操作，可能还会**常量折叠**

字段优化：使用局部变量要比使用成员遍历或者静态变量更快

反射优化：通过反射机制调用方法执行了15次以时，JVM会在运行期间动态生成新的类，将反射方法变为正常方法调用性能提升显著。

JMM内存模型

JMM是内存模型与内存结构不同，JMM定义了一套在多线程读写共享数据时，对数据的可见性、有序性和原子性的保障。

sychronized关键字 是一种线程同步的方式，可以起到将一组复合操作原子化进而实现线程安全。

放在代码块、实例方法或者静态方法上意味着对不同的对象上锁。

为对象上锁的机制实际上是由于每个对象都有一个唯一的monitor，monitor还可以分成 owner、entryset、waitset三个区域，owner区域代表当前对象的拥有者（是唯一的），entryset是因当前对象而阻塞的线程的集合，waitset是等待线程的集合。

含有synchronized关键字的java代码在编译过后能展成含有对monitor操作的指令。通过monitorenter指令，线程可以称为对象的owner，即获取该对象锁。monitorexit指令，当前线程不再是对象的owner，即释放该对象锁。

synchronized优化

从1.6开始，对synchronzied的底层进行了大量的优化，其性能已经可以与其他轻量级并发策略相媲美甚至更好。

在Hotspot虚拟机中，每个对象都有对象头（markword,klass）,Mark Word平时存储这个对象的 哈希码、分代年龄，当加锁时这些信息就根据情况替换为**标记**位。原来的markword会保存起来，存在线程的栈中**锁记录**的结构中，内部可以存储锁定对象的Mark word，解锁时再将这些信息还原到其Markword中，

- 轻量级锁

当线程访问的时间是错开的（没有竞争），那么可以使用轻量级锁来优化。线程执行时将对象mark word中的01标记变为00，使用CAS将锁记录地址写入mark word，表示上了轻量级锁。在CAS修改失败时，不一定是其他的线程竞争，也可能是本线程的重入。释放时将00在改会01；

如果是由于其他线程访问导致CAS（修改锁记录地址）的失败，会带来锁膨胀--将轻量级锁膨胀为重量级锁锁标记，从00变为10。将锁记录地址改为重量级锁指针。此时其他线程要访问只能进入monitor。每个对象都对应一个ObjectMonitor，代表其重量级锁，底层是基于操作系统的mutex Lock实现互斥的。多线程发生竞争时，某个线程成为了monitor 的owner，其他那些线程会被放入EntrySet（）的阻塞队列中，如果线程调用了wait（）方法，那么该线程会释放掉所有持有的mutex，并且进入到WaitSet中，等待下一次被其他线程调用notify唤醒。重量级锁的实现方法依赖于Monior，即底层的操作系统的mutex实现，这样就涉及到内核态和用户态之间的切换，线程的挂起和恢复，开销还是很大的；

- 重量级锁的优化

由于原来的重量级锁涉及操作系统级别的线程的挂起和恢复开销很大，引入了自旋重试的概念。当线程获取不到锁的时候，并不直接陷入阻塞，而是重试几次，有可能重试两次很快就获取到了锁。自旋锁是自适应的，当对象刚刚自旋操作成功了，那么这次也会多自旋几次。如果对象之前自旋失败了，这次就少自旋甚至不自旋，比较智能的算法。

自旋会占用cpu的时间，cpu是多核的自旋才有意义。“红灯”该不该熄火

- 偏向锁

只有第一次使用CAS将线程ID设置在MarkWord头部，之后发现这个线程ID是自己的就表示没有竞争，不用重新CAS修改锁记录地址。但是撤销偏向锁是频繁的，升级为轻量级锁时要STW，开销不小。

其他优化

1、减少上锁时间

2、将一个锁拆分为多个锁，减小锁的粒度。ConcurrentHashMap、LongAdder，linkedBolckingQueue

3、锁粗化 new StringBuffer（).appedn("a").appedn("a").appedn("a").....相当于重入锁多次也是会消耗性能的。循环进入锁->锁循环

4、锁消除，JVM会进行代码逃逸分析，如果对象不会被发布到其他线程，这是jit会忽略到其所有的同步机制。

5、读写分离，CopyOnWirteArrayList，读原始内容，写时复制在一个新数组上。读就不用同步操作了，只对并发写同步即可。

对象创建的过程

对象的内存布局

对象在内存中的布局可以分为 3 块区域：对象头、实例数据和对齐填充。

Hotspot 虚拟机的**对象头**包括两部分信息，第一部分markword用于存储对象自身的运行时数据（哈希码、GC 分代年龄、锁状态标志等等），另一部分是类型指针，虚拟机通过这个指针来确定这个对象是那个类的实例。

实例数据部分是对象真正存储的有效信息，也是在程序中所定义的各种类型的字段内容。

对齐填充部分不是必然存在的，仅仅起占位作用。 因为 Hotspot 虚拟机的自动内存管理系统要求对象起始地址必须是 8 字节的整数倍，换句话说就是对象的大小必须是 8 字节的整数倍。因此当对象头加对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

Step1:类加载检查

虚拟机遇到一条 new 指令时，首先将去检查这个指令是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、链接和初始化过。如果没有，那必须先执行相应的类加载过程。

Step2:分配内存

在类加载检查通过后，接下来虚拟机将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确定，分配内存只需把一块确定大小的内存从 Java 堆中划分出来。

分配方式

- 指针碰撞：适用于堆内存规整，没有碎片的场景。垃圾回收算法采用标记整理，因此可以将堆内存分为使用中和未使用，中间有一个边界指针，只要在该指针处分配对象的大小的内存，指针随之往后移动即可。
- 空闲列表：适用于堆内存不规整，有碎片的场景，垃圾回收算法采用标记清除，JVM会维护一个列表，其中记录着堆中可用的内存区域地址，到时候只需在列表中寻找一个足够大的区域分配给实例对象即可。

内存分配并发问题

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证分配对象是是线程安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试：** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。
- **TLAB写分离：** 为每一个线程预先在 堆中Eden 区分配一块儿内存，JVM 在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

Step3:初始化零值

内存分配完成后，虚拟机需要将实例对象中的属性字段都初始化为零值，这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，我们访问到的是这些字段的数据类型所对应的零值。

Step4:设置对象头

初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。 这些信息存放在对象头中。 另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

Step5:执行 init 方法

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚开始，`<init>` 方法还没有执行，所有的字段都还为零值。紧接着会根据构造方法和实例代码块的内容执行 `<init>` 方法进行初始化赋值，这样一个真正可用的对象才算完全产生出来。