Java【虚拟机】面试题

以下面试题，基于网络整理，和自己编辑。具体参考的文章，会在文末给出所有的链接。

如果胖友有自己的疑问，欢迎在星球提问，我们一起整理吊吊的 Java【虚拟机】面试题的大保健。

而题目的难度，尽量按照从容易到困难的顺序，逐步下去。

因为 Java 并发涉及到的内容会非常多，本面试题可能很难覆盖到所有的知识点，所以推荐 《深入拆解 Java 虚拟机》 。并且，本文会将面试题和该书的章节，大体保持一致。嘻嘻~

另外，本文涉及的面试题会超级超级超级多，所以已经分了小节，胖友要注意哟。

对于大部分 JVM 的面试题，基本都是概念题，所以一些面试题如果和 《深入拆解 Java 虚拟机》 中，可以找到对应的章节，会进行注明。毕竟，看书可以理解更加深刻。

> ：因为和字节码、指令级，理解起来相对复杂，实际面试问的也相对少。所以，本文暂时先不写。当然，未来肯定要补的，这么有趣的东西，胖友说是不是。

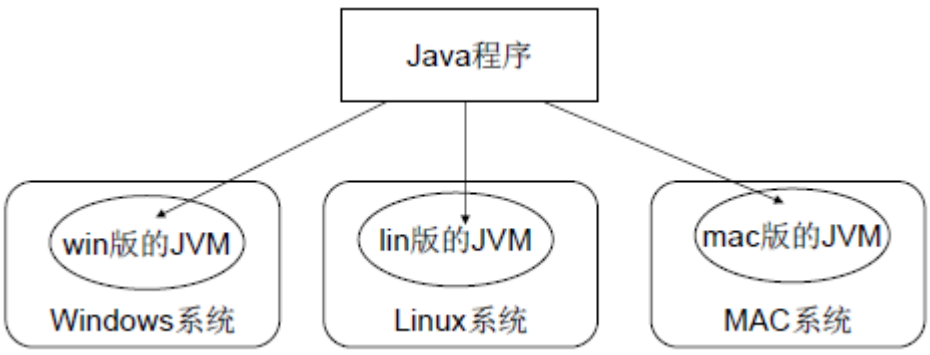
走近 Java

什么是虚拟机?

Java 虚拟机，是一个可以执行 Java 字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件（`.class`）。

Java 被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java 虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

但是，跨平台的是 Java 程序(包括字节码文件)，，而不是 JVM。JVM 是用 C/C++ 开发的，是编译后的机器码，不能跨平台，不同平台下需要安装不同版本的 JVM 。



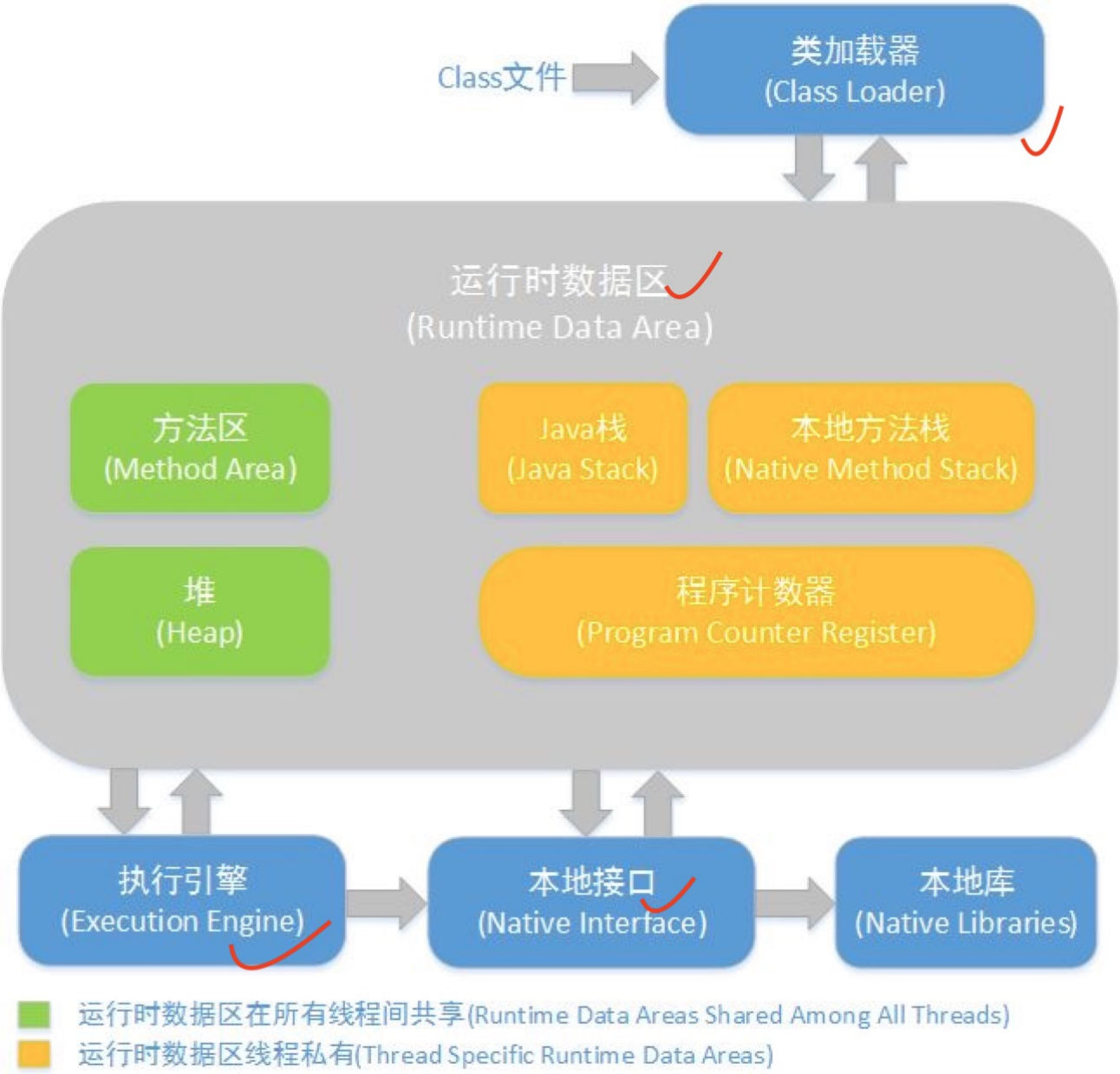
- 也就是说，JVM 能够跨计算机体系结构来执行 Java 字节码，主要是由于 JVM 屏蔽了与各个计算机平台相关的软件或者硬件之间的差异，使得与平台相关的耦合统一由 JVM 提供者来实现。

JVM 由哪些部分组成?

> 考察对 JVM 体系结构的掌握
>
> 《深入拆解 Java 虚拟机》 的 「2.2 运行时数据区域」 。

JVM 的结构基本上由 4 部分组成：

> 注意打钩的 4 个地方。



- **类加载器**，在 JVM 启动时或者类运行时将需要的 class 加载到 JVM 中。
- **内存区**，将内存划分成若干个区以模拟实际机器上的存储、记录和调度功能模块，如实际机器上的各种功能的寄存器或者 PC 指针的记录器等。

  > 关于这一块，我们在 「Java 内存区域与内存溢出异常」 也会细看。

- **执行引擎**，执行引擎的任务是负责执行 class 文件中包含的字节码指令，相当于实际机器上的 CPU 。
- **本地方法调用**，调用 C 或 C++ 实现的本地方法的代码返回结果。

怎样通过 Java 程序来判断 JVM 是 32 位 还是 64 位?

Sun 有一个 Java System 属性来确定JVM的位数：32 或 64。

- `sun.arch.data.model=32 // 32 bit JVM`
- `sun.arch.data.model=64 // 64 bit JVM`

我可以使用以下语句来确定 JVM 是 32 位还是 64 位：

```
System.getProperty("sun.arch.data.model")
```

📣 32 位 JVM 和 64 位 JVM 的最大堆内存分别是多数?

理论上说上 32 位的 JVM 堆内存可以到达 2^32，即 4GB ，但实际上会比这个小很多。不同操作系统之间不同，如 Windows 系统大约 1.5 GB，Solaris 大约 3GB。

64 位 JVM 允许指定最大的堆内存，理论上可以达到 2^64 ，这是一个非常大的数字，实际上你可以指定堆内存大小到 100GB 。甚至有的 JVM，如 Azul ，堆内存到 1000G 都是可能的。

📣 64 位 JVM 中，int 的长度是多数?

Java 中，int 类型变量的长度是一个固定值，与平台无关，都是 32 位或者 4 个字节。意思就是说，在 32 位 和 64 位 的Java 虚拟机中，int 类型的长度是相同的。

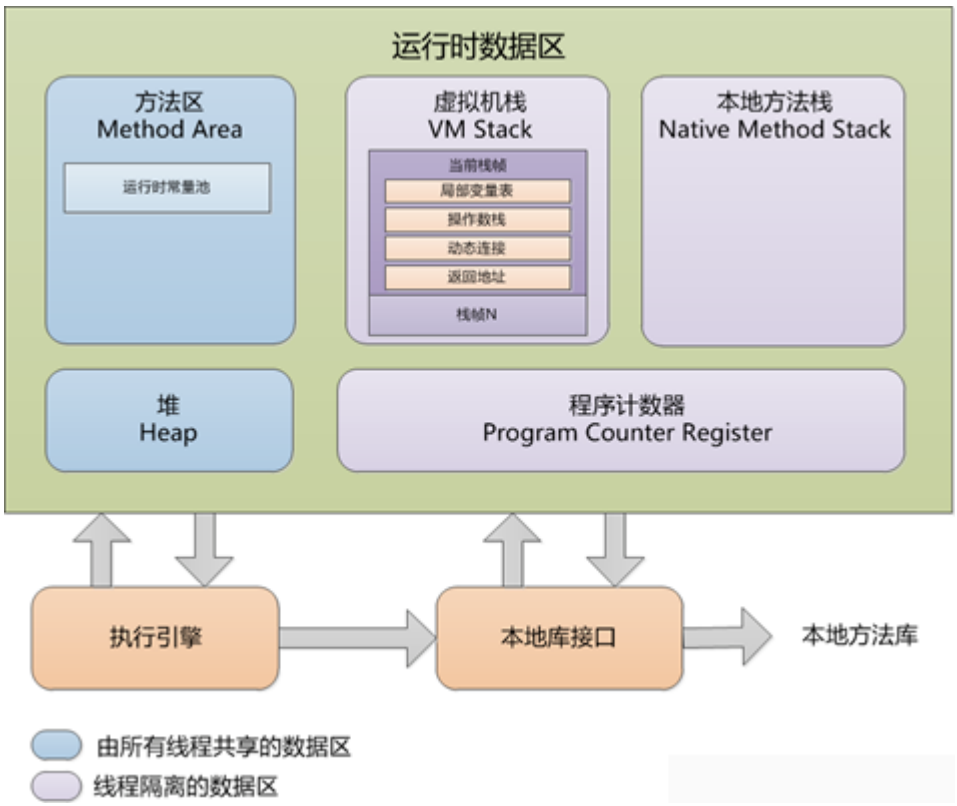
Java 内存区域与内存溢出异常

JVM 运行内存的分类?

> 《深入拆解 Java 虚拟机》 的 「2.3.1 对象的创建」 。

JVM 运行内存的分类如下图所示：

> 注意，这个图是基于 JDK6 版本的运行内存的分类。



- 程序计数器

  ： Java 线程私有，类似于操作系统里的 PC 计数器，它可以看做是当前线程所执行的字节码的行号指示器。

  - 如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）。
  - 此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemoryError 情况的区域。
- 虚拟机栈（栈内存）

  ：Java线程私有，虚拟机栈描述的是 Java 方法执行的内存模型：

  - 每个方法在执行的时候，都会创建一个栈帧用于存储局部变量、操作数、动态链接、方法出口等信息。
  - 每个方法调用都意味着一个栈帧在虚拟机栈中入栈到出栈的过程。
- **本地方法栈**： 和 Java 虚拟机栈的作用类似，区别是该区域为 JVM 提供使用 Native 方法的服务。
- 堆内存

  （线程共享）：所有线程共享的一块区域，垃圾收集器管理的主要区域。

  - 目前主要的垃圾回收算法都是分代收集算法，所以 Java 堆中还可以细分为：新生代和老年代；再细致一点的有 Eden 空间、From Survivor 空间、To Survivor 空间等，默认情况下新生代按照 `8:1:1` 的比例来分配。
  - 根据 Java 虚拟机规范的规定，Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘一样。
- 方法区

  （线程共享）：各个线程共享的一个区域，用于存储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。

  - 虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 Non-Heap（非堆），目的应该是与 Java 堆区分开来。
  - 运行时常量池：是方法区的一部分，用于存放编译器生成的各种字面量和符号引用。

直接内存是不是虚拟机运行时数据区的一部分?

参见 《JVM 直接内存》 文章。

直接内存(Direct Memory)，并不是虚拟机运行时数据区的一部分，也不是 Java 虚拟机规范中农定义的内存区域。在 JDK1.4 中新加入了 NIO(New Input/Output) 类，引入了一种基于通道(Channel)与缓冲区（Buffer）的 I/O 方式，它可以使用 native 函数库直接分配堆外内存，然后通脱一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

- 本机直接内存的分配不会受到 Java 堆大小的限制，受到本机总内存大小限制。
- 配置虚拟机参数时，不要忽略直接内存，防止出现 OutOfMemoryError 异常。

📣 直接内存（堆外内存）与堆内存比较?

1. 直接内存申请空间耗费更高的性能，当频繁申请到一定量时尤为明显。
2. 直接内存 IO 读写的性能要优于普通的堆内存，在多次读写操作的情况下差异明显。

📣 实际上，后续的版本，主要对【方法区】做了一定的调整

- JDK7 的改变
  - 存储在永久代的部分数据就已经转移到了 Java Heap 或者是 Native Heap。但永久代仍存在于 JDK7 中，但是并没完全移除。
  - 常量池和静态变量放到 Java 堆里。
- JDK8 的改变
  - 废弃 PermGen（永久代），新增 Metaspace（元数据区）。
  - 那么方法区还在么？FROM 狼哥 的解答：方法区在 Metaspace 中了，方法区都是一个概念的东西。😈 狼哥通过撸源码获得该信息。

    > 因为，《Java 虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。
    >
    > 同时，大多数用的 JVM 都是 Sun 公司的 HotSpot 。在 HotSpot 上把 GC 分代收集扩展至方法区，或者说使用永久带来实现方法区。

- 参考文章
  - 《笔记之 JVM 方法区：永久带 VS 元空间》
  - 《JVM —— 移除永久代》
  - 《JDK 源码剖析五：JDK8 —— 废弃永久代（PermGen）迎来元空间（Metaspace）》
  - 《Java8 内存模型 —— 永久代(PermGen)和元空间(Metaspace)》

📣 JDK8 之后 Perm Space 有哪些变动? MetaSpace 大小默认是无限的么? 还是你们会通过什么方式来指定大小?

- JDK8 后用元空间替代了 Perm Space ； 字符串常量存放到堆内存中。
- MetaSpace 大小默认没有限制，一般根据系统内存的大小。JVM 会动态改变此值。
- 可以通过 JVM 参数配置
  - `-XX:MetaspaceSize` ： 分配给类元数据空间（以字节计）的初始大小（Oracle 逻辑存储上的初始高水位，the initial high-water-mark）。此值为估计值，MetaspaceSize 的值设置的过大会延长垃圾回收时间。垃圾回收过后，引起下一次垃圾回收的类元数据空间的大小可能会变大。
  - `-XX:MaxMetaspaceSize` ： 分配给类元数据空间的最大值，超过此值就会触发Full GC 。此值默认没有限制，但应取决于系统内存的大小，JVM 会动态地改变此值。

📣 为什么要废弃永久代?

1）现实使用中易出问题。

由于永久代内存经常不够用或发生内存泄露，爆出异常 `java.lang.OutOfMemoryError: PermGen` 。

- 字符串存在永久代中，容易出现性能问题和内存溢出。
- 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。

2）永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。

3）Oracle 可能会将HotSpot 与 JRockit 合二为一。

参照 JEP122： http://openjdk.java.net/jeps/122 ，原文截取：

> Motivation
>
> This is part of the JRockit and Hotspot convergence effort. JRockit customers do not need to configure the permanent generation (since JRockit does not have a permanent generation) and are accustomed to not configuring the permanent generation.

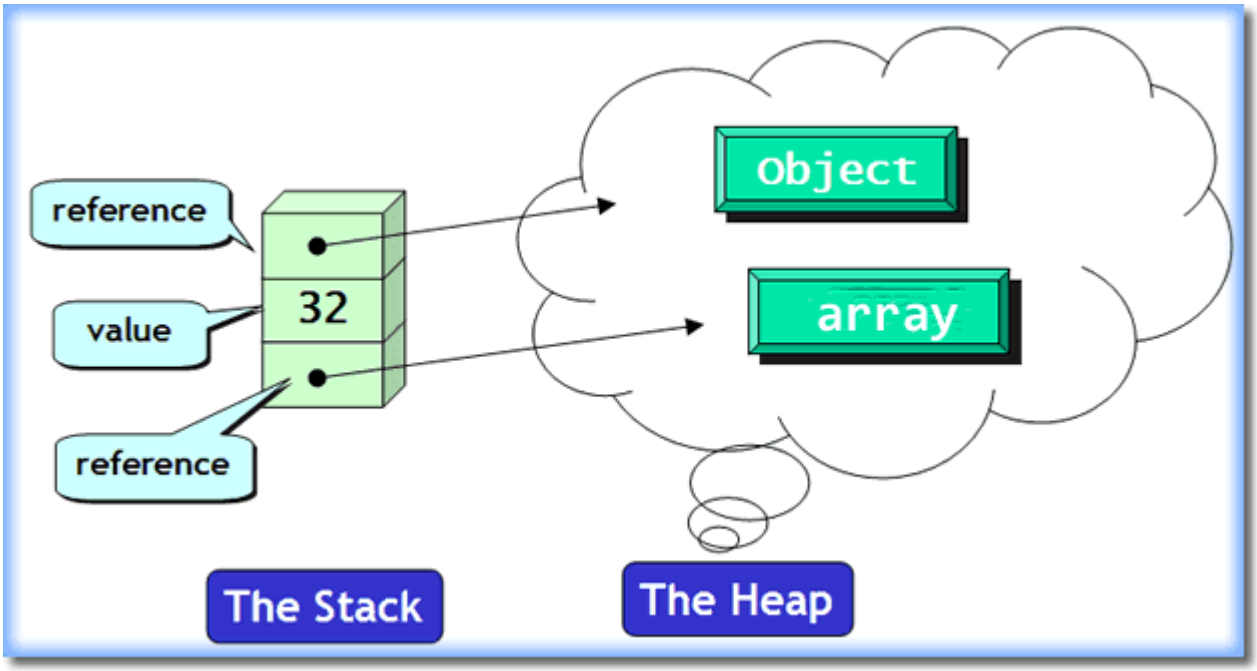
即：移除永久代是为融合 HotSpot JVM 与 JRockit VM 而做出的努力，因为 JRockit 没有永久代，不需要配置永久代。

Java 内存堆和栈区别?

- 栈内存用来存储基本类型的变量和对象的引用变量；堆内存用来存储Java中的对象，无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。
- 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存；堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。
- 如果栈内存没有可用的空间存储方法调用和局部变量，JVM 会抛出 `java.lang.StackOverFlowError` 错误；如果是堆内存没有可用的空间存储生成的对象，JVM 会抛出 `java.lang.OutOfMemoryError` 错误。
- 栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满。`-Xss` 选项设置栈内存的大小，`-Xms` 选项可以设置堆的开始时的大小。

当然，如果你记不住这个些，只要记住如下即可：

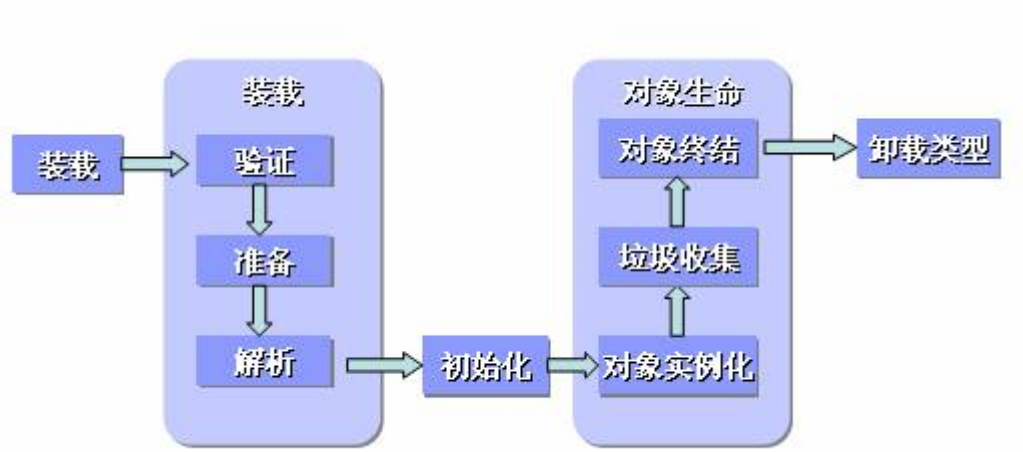
> JVM 中堆和栈属于不同的内存区域，使用目的也不同。栈常用于保存方法帧和局部变量，而对象总是在堆上分配。栈通常都比堆小，也不会在多个线程之间共享，而堆被整个 JVM 的所有线程共享。



JAVA 对象创建的过程?

> 《深入拆解 Java 虚拟机》 的 「2.3.1 对象的创建」 。
>
> 注意，加粗的文字部分。

JAVA 对象创建的过程，如下图所示：



- Java 中对象的创建就是在堆上分配内存空间的过程，此处说的对象创建仅限于 new 关键字创建的普通 Java 对象，不包括数组对象的创建。

1）检测类是否被加载

当虚拟机遇到 `new` 指令时，首先先去检查这个指令的参数是否能在常量池中**定位到一个类的符号引用**，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，就执行类加载过程。

2）为对象分配内存

类加载完成以后，虚拟机就开始为对象分配内存，此时所需内存的大小就已经确定了。只需要在堆上分配所需要的内存即可。

具体的分配内存有两种情况：第一种情况是内存空间绝对规整，第二种情况是内存空间是不连续的。

- 对于内存绝对规整的情况相对简单一些，虚拟机只需要在被占用的内存和可用空间之间移动指针即可，这种方式被称为“**指针碰撞**”。
- 对于内存不规整的情况稍微复杂一点，这时候虚拟机需要维护一个列表，来记录哪些内存是可用的。分配内存的时候需要找到一个可用的内存空间，然后在列表上记录下已被分配，这种方式成为“**空闲列表**”。

多线程并发时会出现正在给对象 A 分配内存，还没来得及修改指针，对象 B 又用这个指针**分配内存**，这样就出现问题了。解决这种问题有两种方案：

- 第一种，是采用同步的办法，使用 CAS 来保证操作的原子性。
- 另一种，是每个线程分配内存都在自己的空间内进行，即是每个线程都在堆中预先分配一小块内存，称为本地线程分配缓冲（Thread Local Allocation Buffer, TLAB），分配内存的时候再TLAB上分配，互不干扰。可以通过 `-XX:+/-UseTLAB` 参数决定。

3）为分配的内存空间初始化零值

对象的内存分配完成后，还需要将对象的内存空间都初始化为零值，这样能保证对象即使没有赋初值，也可以直接使用。

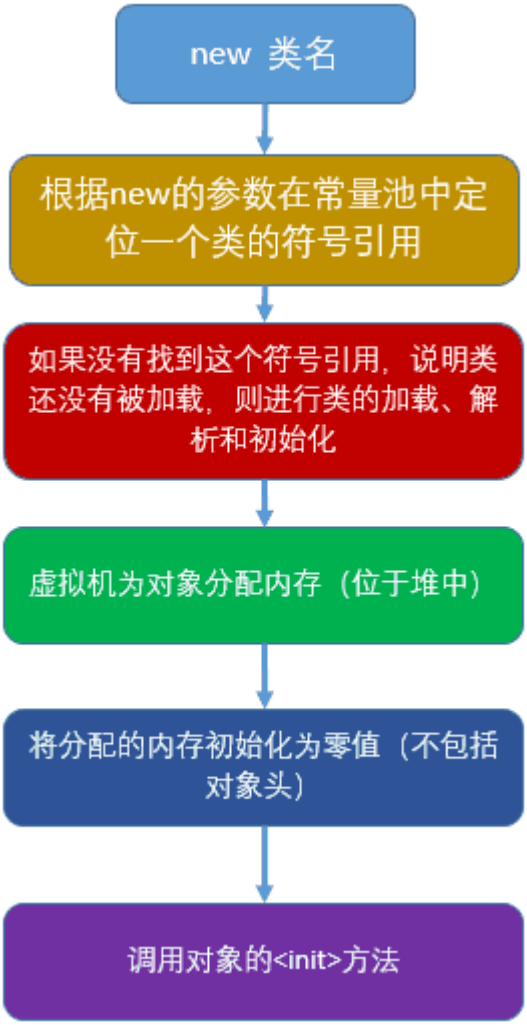
4）对对象进行其他设置

分配完内存空间，初始化零值之后，虚拟机还需要对对象进行其他必要的设置，设置的地方都在对象头中，包括这个对象所属的类，类的元数据信息，对象的 hashcode ， GC 分代年龄等信息。

5）执行 init 方法

执行完上面的步骤之后，在虚拟机里这个对象就算创建成功了，但是对于 Java 程序来说还需要执行 init 方法才算真正的创建完成，因为这个时候对象只是被初始化零值了，还没有真正的去根据程序中的代码分配初始值，调用了 init 方法之后，这个对象才真正能使用。

到此为止一个对象就产生了，这就是 new 关键字创建对象的过程。过程如下：



> 另外，这个问题，面试官可能引申成“`A a = new A()` 经历过什么过程”的问题。

对象的内存布局是怎样的?

> 《深入拆解 Java 虚拟机》 的 「2.3.2 对象的内存布局」 。

对象的内存布局包括三个部分：

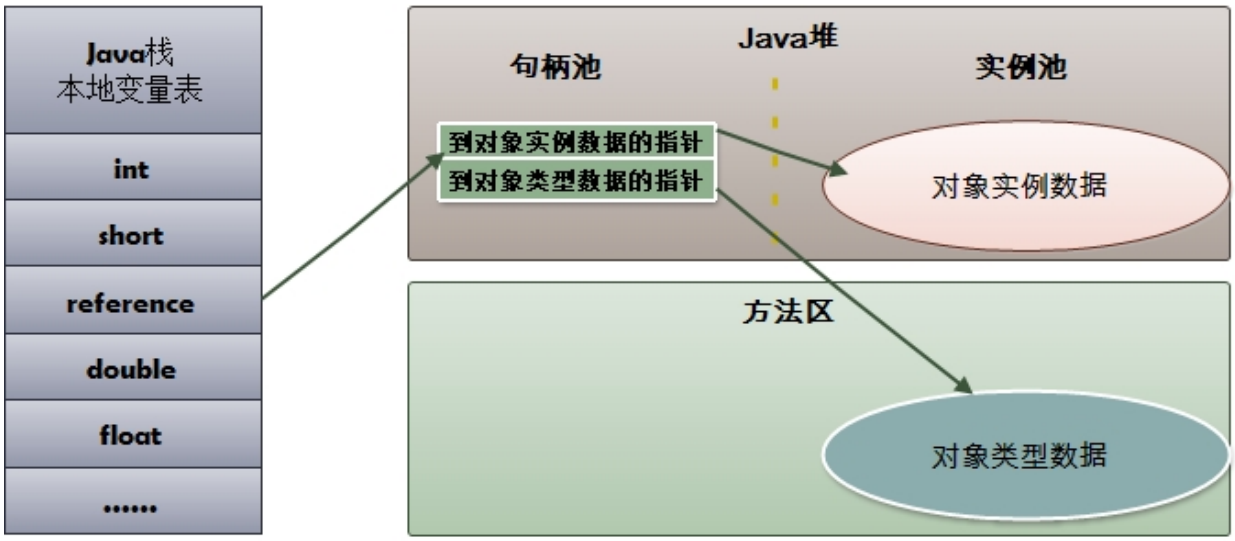
- 对象头：对象头包括两部分信息。
  - 第一部分，是存储对象自身的运行时数据，如哈希码，GC 分代年龄，锁状态标志，线程持有的锁等等。
  - 第二部分，是类型指针，即对象指向类元数据的指针。
- 实例数据：就是数据。
- 对齐填充：不是必然的存在，就是为了对齐。

对象是如何定位访问的?

> 《深入拆解 Java 虚拟机》 的 「2.3.3 对象的访问定位」 。

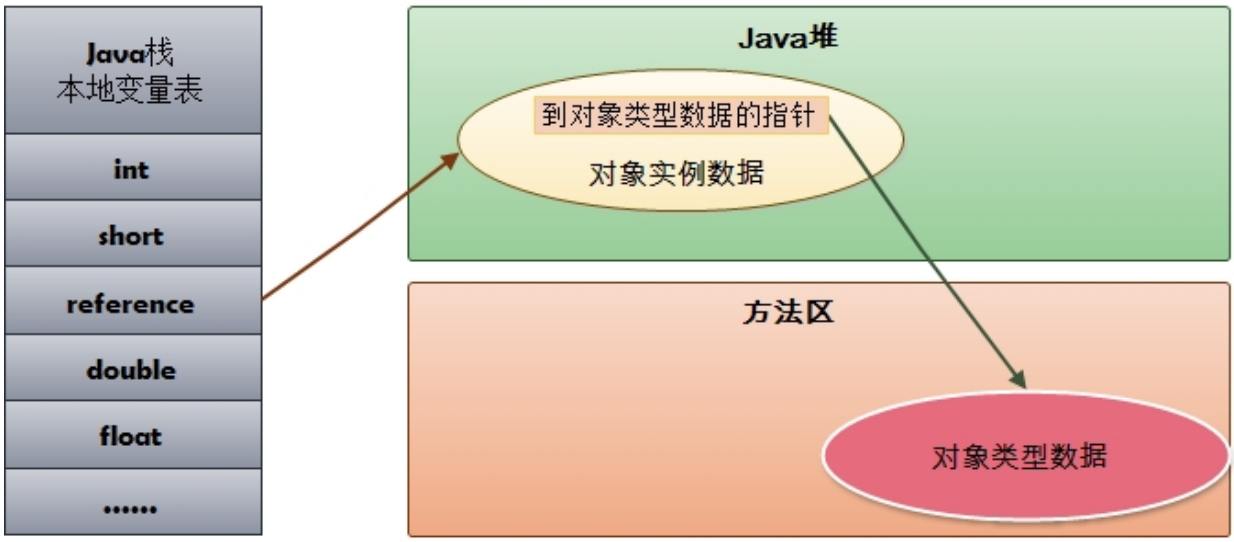
对象的访问定位有两种：

- 句柄定位：Java 堆会画出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。



句柄方式访问对象

- 直接指针访问：Java 堆对象的不居中就必须考虑如何放置访问类型数据的相关信息，而 reference 中存储的直接就是对象地址。



直接指针方式访问对象

对比两种方式?

这两种对象访问方式各有优势。

- 使用句柄来访问的最大好处，就是 reference 中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。
- 使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。

我们目前主要虚拟机 Sun HotSpot 而言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

有哪些 OutOfMemoryError 异常?

> 《深入拆解 Java 虚拟机》 的 「2.4 实战：OutOfMemoryError 异常」 。

在 Java 虚拟机中规范的描述中，除了程序计数器外，虚拟机内存的其它几个运行时区域都有发生的 OutOfMemoryError(简称为“OOM”) 异常的可能。

- Java 堆溢出
- 虚拟机栈和本地方法栈溢出
- 方法区和运行时常量池溢出

  > 从 JDK8 开始，就变成元数据区的内存溢出。

- 本机直接内存溢出

1）Java 堆溢出

重现方式，参见《Java 堆溢出》文章。

另外，Java 堆溢出的原因，有可能是内存泄露，可以使用 MAT 进行分析。

2）虚拟机栈和本地方法栈溢出

由于在 HotSpot 虚拟机中并不区分虚拟机栈和本地方法栈，因此，对于 HotSpot 来说，虽然 `-Xoss` 参数（设置本地方法栈大小）存在，但实际上是无效的，栈容量只由 `-Xss` 参数设定。

关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常。

  > StackOverflowError 不属于 OOM 异常哈。

- 如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常。

重现方式，参见《OutOfMemoryError 异常 —— 虚拟机栈和本地方法栈溢出》文章。

3）运行时常量池溢出

因为 JDK7 将常量池和静态变量放到 Java 堆里，所以无法触发运行时常量池溢出。如果想要触发，可以使用 JDK6 的版本。

重现方式，参见《JVM 内存溢出 - 方法区及运行时常量池溢出》文章。

4）方法区的内存溢出

因为 JDK8 将方法区溢出，所以无法触发方法区的内存溢出溢出。如果想要触发，可以使用 JDK7 的版本。

重现方式，参见《Java 方法区溢出》文章。

5）元数据区的内存溢出

实际上，方法区的内存溢出在 JDK8 中，变成了元数据区的内存溢出。所以，重现方式，还是参见《Java 方法区溢出》文章，只是说，需要增加 `-XX:MaxMetaspaceSize=10m` VM 配置项。

6）本机直接内存溢出

重现方式，参见《JVM 内存溢出 —— 直接内存溢出》文章。

另外，非常推荐一篇文章，胖友耐心阅读，提供了更多有趣的案例，《Java 内存溢出(OOM)异常完全指南》。

📣 当出现了内存溢出，你怎么排错？

- 1、首先，控制台查看错误日志。

- 2、然后，使用 JDK 自带的 jvisualvm 工具查看系统的堆栈日志。
- 3、定位出内存溢出的空间：堆，栈还是永久代（JDK8 以后不会出现永久代的内存溢出）。
    - 如果是堆内存溢出，看是否创建了超大的对象。
    - 如果是栈内存溢出，看是否创建了超大的对象，或者产生了死循环。

Java 中会存在内存泄漏吗?

理论上 Java 因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是 Java 被广泛使用于服务器端编程的一个重要原因）。然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被 GC 回收也会发生内存泄露。例如说：

- Hibernate 的 Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象。
- 使用 Netty 的堆外的 ByteBuf 对象，在使用完后，并未归还，导致使用的一点一点在泄露。

垃圾收集器与内存分配策略

什么是垃圾回收机制?

> 《深入拆解 Java 虚拟机》 的 「3.1 概述」 。

- Java 中对象是采用 `new` 或者反射的方法创建的，这些对象的创建都是在堆(Heap)中分配的，所有对象的回收都是由 Java 虚拟机通过垃圾回收机制完成的。GC 为了能够正确释放对象，会监控每个对象的运行状况，对他们的申请、引用、被引用、赋值等状况进行监控。
- Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。
- 可以调用下面的方法之一：`System#gc()` 或 `Runtime#getRuntime()#gc()` ，但 JVM 也可以屏蔽掉显示的垃圾回收调用。

📣 为什么不建议在程序中显式的声明 System.gc() ?

因为显式声明是做堆内存全扫描，也就是 Full GC ， 是需要停止所有的活动的(Stop The World Collection)，对应用很大可能存在影响。

另外，调用 `System.gc()` 方法后，不会立即执行 Full GC ， 而是虚拟机自己决定的。

📣 如果一个对象的引用被设置为 null , GC 会立即释放该对象的内存么?

不会, 这个对象将会在下一次 GC 循环中被回收。

📣 #finalize() 方法什么时候被调用？它的目的是什么?

> 《深入拆解 Java 虚拟机》 的 「3.2.4 生存还是死亡」 。

`#finallize()` 方法，是在释放该对象内存前由 GC (垃圾回收器)调用。

- 通常建议在这个方法中释放该对象持有的资源，例如持有的堆外内存、和远程服务的长连接。
- 一般情况下，不建议重写该方法。
- 对于一个对象，该方法有且仅会被调用一次。

如何判断一个对象是否已经死去?

> 《深入拆解 Java 虚拟机》 的 「3.2 对象已死吗」 。

有两种方式：

1. 引用计数
2. 可达性分析

1）引用计数

每个对象有一个引用计数属性，新增一个引用时计数加 1 ，引用释放时计数减 1 ，计数为 0 时可以回收。此方法简单，无法解决对象相互循环引用的问题。目前在用的有 Python、ActionScript3 等语言。

2）可达性分析（Reachability Analysis）

从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。不可达对象。目前在用的有 Java、C# 等语言。

📣 如果 A 和 B 对象循环引用，是否可以被 GC?

可以，因为 Java 采用可达性分析的判断方式。

📣 在 Java 语言里，可作为 GC Roots 的对象包括以下几种?

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象。
2. 方法区中的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 方法)中引用的对象。

📣 方法区是否能被回收?

方法区可以被回收，但是价值很低，主要回收废弃的常量和无用的类。

如何判断无用的类，需要完全满足如下三个条件：

1. 该类所有实例都被回收（Java 堆中没有该类的对象）。
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，无法在任何地方利用反射访问该类。

Java 对象有哪些引用类型?

> 《深入拆解 Java 虚拟机》 的 「3.2.3 再谈引用类型」 。

Java 一共有四种引用类型：

- 强引用
- 软引用（SoftReference）
- 弱引用（WeakReference）
- 虚引用（PhantomReference）

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。

> ：不然，代码都没法写了 😺

2）软引用（SoftReference）

如果一个对象只具有软引用，那就类似于可有可物的生活用品。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。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

使用示例，见 《精尽 MyBatis 源码分析 —— 缓存模块》 的 「2.10 SoftCache」 小节。

3）弱引用（WeakReference）

如果一个对象只具有弱引用，那就类似于可有可物的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，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。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

使用示例，见

- 《精尽 MyBatis 源码分析 —— 缓存模块》 的 「2.9 WeakCache」 小节。
- 《精尽 Netty 源码解析 —— Buffer 之 ByteBuf（三）内存泄露检测》 的 「4. ResourceLeakDetector」 小节。

4）虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。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。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

如果胖友想看看各种引用在 GC 下的效果，可以看看 《Java 中的四种引用类型》 提供的代码示例。

🚀 WeakReference 与 SoftReference的区别?

虽然 WeakReference 与 SoftReference 都有利于提高 GC 和 内存的效率。

- 但是 WeakReference 一旦失去最后一个强引用，就会被 GC 回收
- 而 SoftReference 虽然不能阻止被回收，但是可以延迟到 JVM 内存不足的时候。

🚀 为什么要有不同的引用类型?

不像 C 语言，我们可以控制内存的申请和释放，在 Java 中有时候我们需要适当的控制对象被回收的时机，因此就诞生了不同的引用类型，可以说不同的引用类型实则是对 GC 回收时机不可控的妥协。有以下几个使用场景可以充分的说明：

- 利用软引用和弱引用解决 OOM 问题。用一个 HashMap 来保存图片的路径和相应图片对象关联的软引用之间的映射关系，在内存不足时，JVM 会自动回收这些缓存图片对象所占用的空间，从而有效地避免了 OOM 的问题.

- 通过软引用实现 Java 对象的高速缓存。比如我们创建了一 Person 的类，如果每次需要查询一个人的信息，哪怕是几秒中之前刚刚查询过的，都要重新构建一个实例，这将引起大量 Person 对象的消耗，并且由于这些对象的生命周期相对较短，会引起多次 GC 影响性能。此时，通过软引用和 HashMap 的结合可以构建高速缓存，提供性能。

JVM 垃圾回收算法?

> 《深入拆解 Java 虚拟机》 的 「3.3 垃圾回收算法」 。

有四种算法：

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

1）标记-清除算法

标记-清除（Mark-Sweep）算法，是现代垃圾回收算法的思想基础。

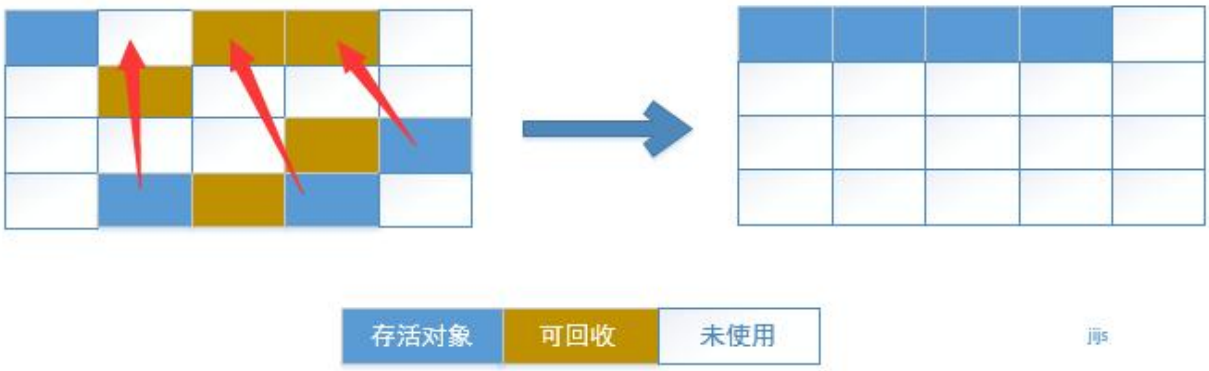
标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。

一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象（好多资料说标记出要回收的对象，其实明白大概意思就可以了）。然后，在清除阶段，清除所有未被标记的对象。

- 缺点：
  - 1、效率问题，标记和清除两个过程的效率都不高。
  - 2、空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

2）标记-整理算法

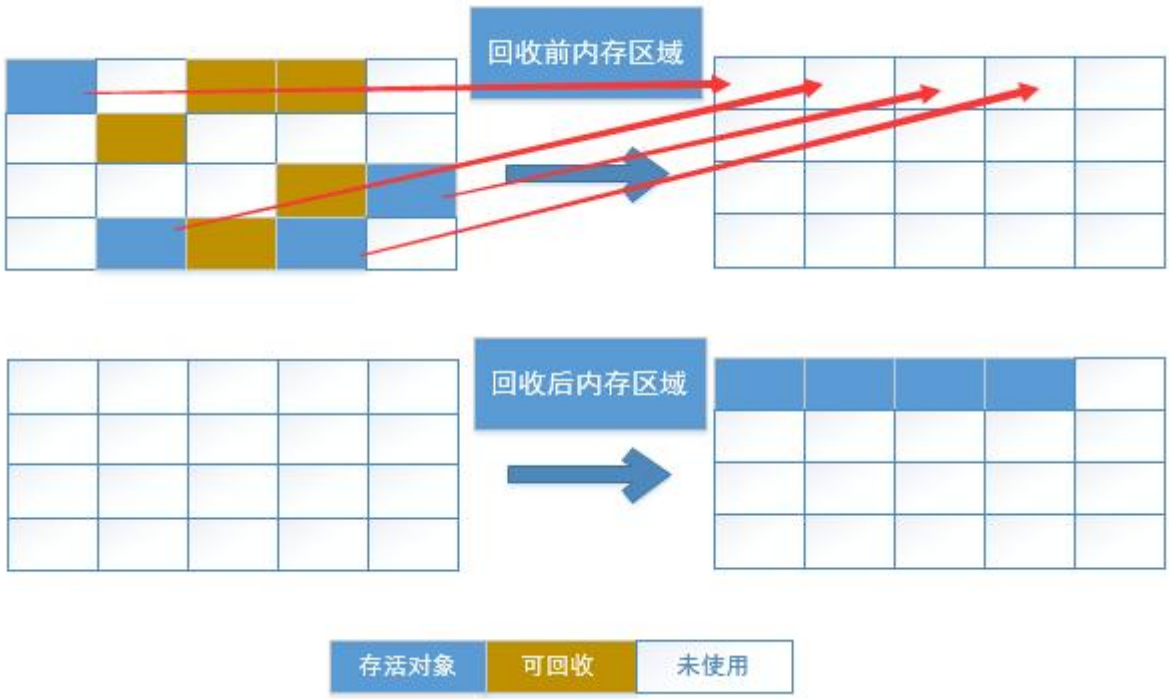
标记整理算法，类似与标记清除算法，不过它标记完对象后，不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉边界以外的内存。



- 优点：
  - 1、相对标记清除算法，解决了内存碎片问题。
  - 2、没有内存碎片后，对象创建内存分配也更快速了（可以使用TLAB进行分配）。
- 缺点：
  - 1、效率问题，（同标记清除算法）标记和整理两个过程的效率都不高。

3）复制算法

复制算法，可以解决效率问题，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块，当这一块内存用完了，就将还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，这样使得每次都是对整个半区进行内存回收，内存分配时也就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（还可使用TLAB进行高效分配内存）。



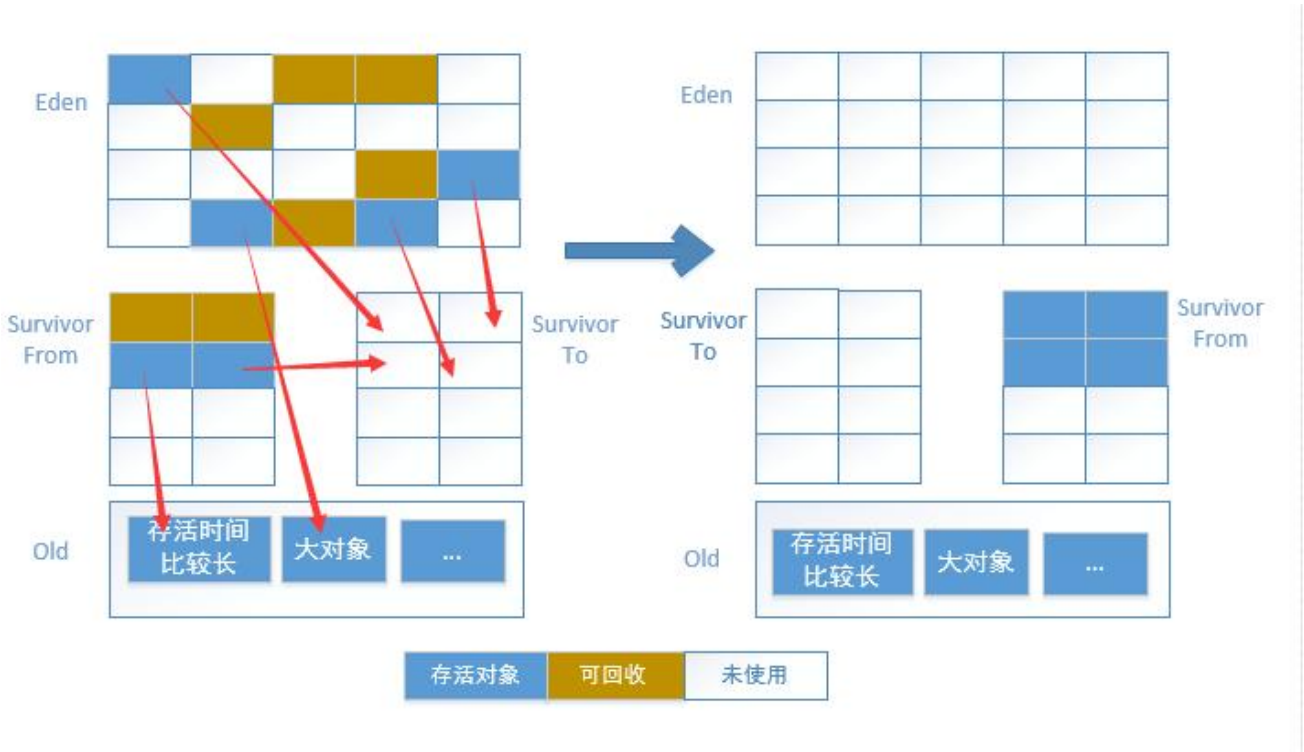
- 图的上半部分是未回收前的内存区域，图的下半部分是回收后的内存区域。通过图，我们发现不管回收前还是回收后都有一半的空间未被利用。
- 优点：
  - 1、效率高，没有内存碎片。

- 缺点：
  - 1、浪费一半的内存空间。
  - 2、复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。

4）分代收集算法

当前商业虚拟机都是采用分代收集算法，它根据对象存活周期的不同将内存划分为几块，一般是把 Java 堆分为新生代和老年代，然后根据各个年代的特点采用最适当的收集算法。

- 在新生代中，每次垃圾收集都发现有大批对象死去，只有少量存活，就选用复制算法。
- 而老年代中，因为对象存活率高，没有额外空间对它进行分配担保，就必须使用“标记清理”或者“标记整理”算法来进行回收。



- 图的左半部分是未回收前的内存区域，右半部分是回收后的内存区域。
- 对象分配策略：
  - 对象优先在 Eden 区域分配，如果对象过大直接分配到 Old 区域。
  - 长时间存活的对象进入到 Old 区域。
- 改进自复制算法
  - 现在的商业虚拟机都采用这种收集算法来回收新生代，IBM 公司的专门研究表明，新生代中的对象 98% 是“朝生夕死”的，所以并不需要按照 `1:1` 的比例来划分内存空间，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 和其中一块 Survivor 。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性地复制到另外一块 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 2 块 Survivor 的大小比例是 `8:1:1`，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% 的内存会被“浪费”。当然，98% 的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于 10% 的对象存活，当 Survivor 空间不够用时，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

什么是安全点?

> 《深入拆解 Java 虚拟机》 的 「3.4.2 安全点」 。

SafePoint 安全点，顾名思义是指一些特定的位置，当线程运行到这些位置时，线程的一些状态可以被确定(the thread's representation of it's Java machine state is well described)，比如记录OopMap 的状态，从而确定 GC Root 的信息，使 JVM 可以安全的进行一些操作，比如开始 GC 。

SafePoint 指的特定位置主要有：

1. 循环的末尾 (防止大循环的时候一直不进入 Safepoint ，而其他线程在等待它进入 Safepoint)。
2. 方法返回前。
3. 调用方法的 Call 之后。
4. 抛出异常的位置。

详细的内容，可以看看 《深入学习 JVM-JVM 安全点和安全区域》 。

- 如何使线程中断
  - 主动式

    > 主动式 JVM 设置一个全局变量，线程去按照某种策略检查这个变量一旦发现是 SafePoint 就主动挂起。
    >
    > HostSpot 虚拟机采用的是主动式使线程中断。

  - 被动式

    > 被动式就是发个信号，例如关机、Control+C ，带来的问题就是不可控，发信号的时候不知道线程处于什么状态。

- 安全区域

  > 如果程序长时间不执行，比如线程调用的 sleep 方法，这时候程序无法响应 JVM 中断请求这时候线程无法到达安全点，显然 JVM 也不可能等待程序唤醒，这时候就需要安全区域了。
  >
  > 安全区域是指一段代码片中，引用关系不会发生变化，在这个区域任何地方 GC 都是安全的，安全区域可以看做是安全点的一个扩展。
  >
  > - 线程执行到安全区域的代码时，首先标识自己进入了安全区域，这样 GC 时就不用管进入安全区域的线程了.
  > - 线程要离开安全区域时就检查 JVM 是否完成了 GC Roots 枚举（或者整个 GC 过程），如果完成就继续执行，如果没有完成就等待直到收到可以安全离开的信号。

JVM 垃圾收集器有哪些?

> 《深入拆解 Java 虚拟机》 的 「3.5 垃圾收集器」 。

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

- 新生代收集器
  - Serial 收集器
  - ParNew 收集器

    > ParNew 收集器，是 Serial 收集器的多线程版。

  - Parallel Scavenge 收集器
- 老年代收集器

    - Serial Old 收集器
        - Serial Old 收集器，是 Serial 收集器的老年代版本。
    - Parallel Old 收集器
        - Parallel Old 收集器，是 Parallel Scavenge 收集器的老年代版本。
    - CMS 收集器
- 新生代 + 老年代收集器
    - G1 收集器
    - ZGC 收集器

小结表格如下：

收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

关于每种垃圾收集器的说明，请看 如下文章：

- 《深入理解 JVM(3) —— 7 种垃圾收集器》
- 《一文读懂 Java 11 的 ZGC 为何如此高效》

🚀 G1 和 CMS 的区别?

- CMS ：并发标记清除。他的主要步骤有：初始收集，并发标记，重新标记，并发清除（删除）、重置。
- G1：主要步骤：初始标记，并发标记，重新标记，复制清除（整理）
- CMS 的缺点是对 CPU 的要求比较高。G1是将内存化成了多块，所有对内段的大小有很大的要求。
- CMS是清除，所以会存在很多的内存碎片。G1是整理，所以碎片空间较小。
- G1 和 CMS 都是响应优先把，他们的目的都是尽量控制 STW 时间。

> G1 和 CMS 的 Full GC 都是单线程 mark sweep compact 算法，直到 JDK10 才优化为并行的。

感兴趣的胖友，可以看看 《GC 优化的一些总结》 的分析。

CMS 算法的过程，CMS 回收过程中 JVM 是否需要暂停?

会有短暂的停顿。详细的，可以看看 《[jvm][面试] 并发收集器 CMS(Concurrent Mark-Sweep)》 。

如何使用指定的垃圾收集器

配置	描述
-XX:+UserSerialGC	串行垃圾收集器
-XX:+UserParrallelGC	并行垃圾收集器
-XX:+UseConcMarkSweepGC	并发标记扫描垃圾回收器
-XX:ParallelCMSThreads	并发标记扫描垃圾回收器 =为使用的线程数量
-XX:+UseG1GC	G1垃圾回收器

对象分配规则是什么?

> 《深入拆解 Java 虚拟机》 的 「3.6 对象分配与回收策略」 。

- 对象优先分配在 Eden 区。

  > 如果 Eden 区无法分配，那么尝试把活着的对象放到 Survivor0 中去（Minor GC）
  >
  > - 如果 Survivor0 可以放入，那么放入之后清除 Eden 区。
  > - 如果 Survivor0 不可以放入，那么尝试把 Eden 和 Survivor0 的存活对象放到 Survivor1 中。
  >   - 如果 Survivor1 可以放入，那么放入 Survivor1 之后清除 Eden 和 Survivor0 ，之后再把 Survivor1 中的对象复制到 Survivor0 中，保持 Survivor1 一直为空。
  >   - 如果 Survivor1 不可以放入，那么直接把它们放入到老年代中，并清除 Eden 和 Survivor0 ，这个过程也称为**分配担保**。
  >
  > ps：清除 Eden、Survivor 区，就是 Minor GC 。
  >
  > 总结来说，分配的顺序是：新生代（Eden => Survivor0 => Survivor1）=> 老年代

- 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。

  > 这样做的目的是，避免在 Eden 区和两个 Survivor 区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。

- 长期存活的对象进入老年代。

  > 虚拟机为每个对象定义了一个年龄计数器，如果对象经过了 1 次 Minor GC 那么对象会进入 Survivor 区，之后每经过一次 Minor GC 那么对象的年龄加 1 ，知道达到阀值对象进入老年区。

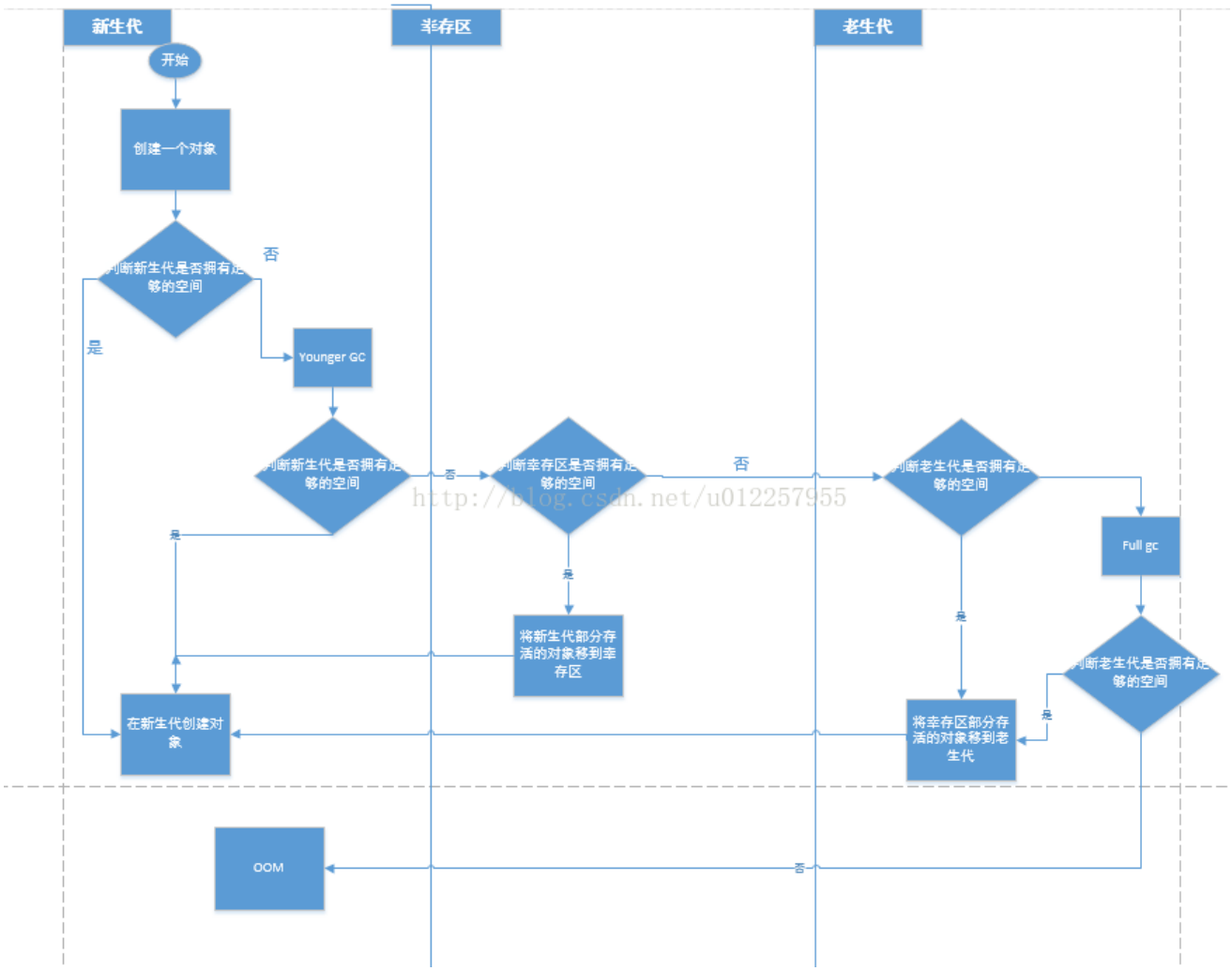
- 动态判断对象的年龄。

  > 为了更好的适用不同程序的内存情况，虚拟机并不是永远要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代。
  >
  > 如果 Survivor 区中相同年龄的所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。

- 空间分配担保。

> 每次进行 Minor GC 时，JVM 会计算 Survivor 区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次 Full GC ，如果小于检查 HandlePromotionFailure 设置，如果 `true` 则只进行 Monitor GC ，如果 `false` 则进行 Full GC 。

如下是一张对象创建时，分配内存的图：



为什么新生代内存需要有两个 Survivor 区?

详细的原因，可以看 《为什么新生代内存需要有两个 Survivor 区》 文章。

什么是新生代 GC 和老年代 GC?

GC 经常发生的区域是堆区，堆区还可以细分为

- 新生代
  - 一个 Eden 区
  - 两个 Survivor 区
- 老年代

> 默认新生代(Young)与老年代(Old)的比例的值为 `1:2` (该值可以通过参数 `-XX:NewRatio` 来指定)。
>
> 默认的 `Eden:from:to=8:1:1` (可以通过参数 `-XX:SurvivorRatio` 来设定)。

新生代GC（MinorGC/YoungGC）：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 MinorGC 非常频繁，一般回收速度也比较快。

老年代GC（MajorGC/FullGC）：指发生在老年代的 GC，出现了 MajorGC，经常会伴随至少一次的 MinorGC（但非绝对的，在 Parallel Scavenge 收集器的收集策略里就有直接进行 MajorGC 的策略选择过程）。MajorGC 的速度一般会比 MinorGC 慢 10 倍以上。

📣 什么情况下会出现 Young GC?

对象优先在新生代 Eden 区中分配，如果 Eden 区没有足够的空间时，就会触发一次 Young GC 。

📣 什么情况下回出现 Full GC?

Full GC 的触发条件有多个，FULL GC 的时候会 STOP THE WORD 。

- 1、在执行 Young GC 之前，JVM 会进行空间分配担保——如果老年代的连续空间小于新生代对象的总大小（或历次晋升的平均大小），则触发一次 Full GC 。
- 2、大对象直接进入老年代，从年轻代晋升上来的老对象，尝试在老年代分配内存时，但是老年代内存空间不够。
- 3、显式调用 `System#gc()` 方法时。

虚拟机性能监控与故障处理工具

JDK 的命令行工具有哪些可以监控虚拟机?

> 《深入拆解 Java 虚拟机》 的 「4.2 JDK 的命令行工具」 。

- jps： 虚拟机进程状况工具

  > JVM Process Status Tool ， 显示指定系统内所有的HotSpot虚拟机进程。

- jstat： 虚拟机统计信息监控工具

  > JVM statistics Monitoring，是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

- jinfo： Java 配置信息工具

  > JVM Configuration info，这个命令作用是实时查看和调整虚拟机运行参数。

- jmap： Java 内存映射工具

  > JVM Memory Map，命令用于生成 heap dump 文件。

- jhat： 虚拟机堆转储快照分析工具

  > JVM Heap Analysis Tool，命令是与 jmap 搭配使用，用来分析 jmap 生成的 dump 文件。jhat 内置了一个微型 的HTTP/HTML 服务器，生成 dump 的分析结果后，可以在浏览器中查看。

- jstack： Java 堆栈跟踪工具

  > Java Stack Trace，用于生成 Java 虚拟机当前时刻的线程快照。

- HSDIS： JIT 生成代码反编译

JDK 的可视化工具有哪些可以监控虚拟机?

> 《深入拆解 Java 虚拟机》 的 「4.3 JDK 的可视化工具」 。

- Java 自带
  - JConsole： Java 监视与管理控制台

    > Java Monitoring and Management Console 是从 Java5 开始，在 JDK 中自带的 Java 监控和管理控制台，用于对 JVM 中内存，线程和类等的监控。

  - VisualVM： 多合一故障处理工具

    > JDK 自带全能工具，可以分析内存快照、线程快照、监控内存变化、GC变化等。
    >
    > 特别是 BTrace 插件，动态跟踪分析工具。

- 第三方
  - MAT： 内存分析工具

    > Memory Analyzer Tool，一个基于 Eclipse 的内存分析工具，是一个快速、功能丰富的 Java heap 分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。

  - GChisto： 一款专业分析 GC 日志的工具。

另外，一些开源项目，例如 SkyWalking、Cat，也提供了 JVM 监控的功能，更加适合生产环境，对 JVM 的监控。

怎么获取 Java 程序使用的内存?

可以通过 `java.lang.Runtime` 类中与内存相关方法来获取剩余的内存，总内存及最大堆内存。通过这些方法你也可以获取到堆使用的百分比及堆内存的剩余空间。

- `Runtime#freeMemory()` 方法，返回剩余空间的字节数。
- `Runtime#totalMemory()` 方法，总内存的字节数。

- `Runtime#maxMemory()` 方法，返回最大内存的字节数。

调优案例分析与实战

在 《深入拆解 Java 虚拟机》 的 「第5章 调优案例分析与实战」 中，已经提供了一些案例，建议胖友可以看看。

常见 GC 的优化配置?

配置	描述
-Xms	初始化堆内存大小
-Xmx	堆内存最大值
-Xmn	新生代大小
-XX:PermSize	初始化永久代大小
-XX:MaxPermSize	永久代最大容量
-XX:SurvivorRatio	设置年轻代中 Eden 区与 Survivor 区的比值
-XX:Xmn	设置年轻代大小

另外，也可以看看 《JVM 调优》 文章。

如何排查线程 Full GC 频繁的问题?

- 《线上 Full GC 频繁的排查》
- 《触发 JVM 进行 Full GC 的情况及应对策略》

JVM 的永久代中会发生垃圾回收么?

- **Young GC** 不会发生在永久代。
- 如果永久代满了或者是超过了临界值，会触发完全垃圾回收(**Full GC**)。如果我们仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免 Full GC 是非常重要的原因。

  > Java8：从永久代到元数据区 (注：Java8 中已经移除了永久代，新加了一个叫做元数据区的 native 内存区)。

有看过 GC 日志么?

> ：这个问题，一般面试不会问，加进来，主要让胖友知道，有这么个知识点。

参见文章如下：

- [《JVM]理解GC日志》
- 《GC 日志查看分析》

TODO JVM 线程案例

TODO 问晓峰 TODO 问阿牛 TODO 问狼哥 TODO 问闪电侠 TODO 问超哥

TODO 类文件结构

TODO 虚拟机类加载机制

> 类加载器，是面试的重点，所以要掌握好。当然，相对来说难度也不算上~

类加载器是有了解吗?

> 《深入拆解 Java 虚拟机》 的 「7.4 类加载器」 。

类加载器(ClassLoader)，用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序(`.java` 文件)在经过 Java 编译器编译之后就被转换成 Java 字节代码(`.class` 文件)。

类加载器，负责读取 Java 字节代码，并转换成 `java.lang.Class` 类的一个实例。

- 每个这样的实例用来表示一个 Java 类。通过此实例的 `Class#newInstance(...)` 方法，就可以创建出该类的一个对象。
- 实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。

类加载发生的时机是什么时候?

> 《深入拆解 Java 虚拟机》 的 「7.2 类加载的时机」 。

虚拟机严格规定，有且仅有 5 种情况必须对类进行加载：

> 注意，有些文章会称为对类进行“初始化”。

- 1、遇到 `new` 、 `getstatic` 、 `putstatic` 、 `invokestatic` 这四条字节码指令时，如果类还没进行初始化，则需要先触发其初始化。
- 2、使用 `java.lang.reflect` 包的方法对类进行反射调用的时候，如果类还没进行初始化，则需要先触发其初始化。
- 3、当初始化了一个类的时候，如果发现其父类还没进行初始化，则需要先触发其父类的初始化。
- 4、当虚拟机启动时，用户需要指定一个执行的主类，即调用其 `#main(String[] args)` 方法，虚拟机则会先初始化该主类。
- 5、当使用 JDK7 的动态语言支持时，如果一个 `java.lang.invoke.MethodHandle` 实例最后的解析结果为 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

类加载器是如何加载 Class 文件的?

> 《深入拆解 Java 虚拟机》 的 「7.2 类加载的时机」 。

下图所示是 ClassLoader 加载一个 `.class` 文件到 JVM 时需要经过的步骤：

- 第一个阶段，加载(Loading)，是找到 `.class` 文件并把这个文件包含的字节码加载到内存中。
- 第二阶段，连接(Linking)，又可以分为三个步骤，分别是字节码验证、Class 类数据结构分析及相应的内存分配、最后的符号表的解析。
- 第三阶段，Initialization(类中静态属性和初始化赋值)，以及Using(静态块的执行)等。

> 注意，不包括卸载(Unloading)部分。

1）加载

> 加载是“类加载”过程的第一阶段，胖友不要混淆这两个名字。

在加载阶段，虚拟机需要完成以下三件事情：

- 通过一个类的全限定名来获取其定义的二进制字节流。
- 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 在Java堆中生成一个代表这个类的 `java.lang.Class` 对象，作为对方法区中这些数据的访问入口。

相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的类加载器来完成加载，也可以自定义自己的类加载器来完成加载。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在Java堆中也创建一个 `java.lang.Class` 类的对象，这样便可以通过该对象访问方法区中的这些数据。

2）加载

2.1 验证：确保被加载的类的正确性

验证是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

验证阶段大致会完成4个阶段的检验动作：

- 文件格式验证：验证字节流是否符合 Class 文件格式的规范。例如：是否以 `0xCAFEBABE` 开头、主次版本号是否在当前虚拟机的处理范围之内、常量池中的常量是否有不被支持的类型。
- 元数据验证：对字节码描述的信息进行语义分析（注意：对比 javac 编译阶段的语义分析），以保证其描述的信息符合 Java 语言规范的要求。例如：这个类是否有父类，除了 `java.lang.Object` 之外。
- 字节码验证：通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。
- 符号引用验证：确保解析动作能正确执行。

验证阶段是非常重要的，但不是必须的，它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用 `-Xverifynone` 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

2.2 准备：为类的静态变量分配内存，并将其初始化为默认值

准备阶段，是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。对于该阶段有以下几点需要注意：

- 1、这时候进行内存分配的仅包括类变量(`static`)，而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在 Java 堆中。

  > 思考下，对于类本身，静态变量就是其属性。

- 2、这里所设置的初始值通常情况下是数据类型默认的零值(如`0`、`0L`、`null`、`false`等)，而不是被在 Java 代码中被显式地赋予的值。

  假设一个类变量的定义为：`public static int value = 3`。那么静态变量`value`在准备阶段过后的初始值为`0`，而不是`3`。因为这时候尚未开始执行任何 Java 方法，而把`value`赋值为`3`的`public static`指令是在程序编译后，存放于**类构造器**`<clinit>()`方法之中的，所以把`value`赋值为`3`的动作将在初始化阶段才会执行。

  > 这里还需要注意如下几点：
  >
  > - 对基本数据类型来说，对于类变量(`static`)和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。
  > - 对于同时被`static`和`final`修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被`final`修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。
  > - 对于引用数据类型 reference 来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的空值，即`null`。
  > - 如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的“空”值。

- 3、如果类字段的字段属性表中存在 ConstantValue 属性，即同时被`final`和`static`修饰，那么在准备阶段变量`value`就会被初始化为 ConstValue 属性所指定的值。

  假设上面的类变量`value`被定义为：`public static final int value = 3`。编译时，`javac`将会为`value`生成 ConstantValue 属性。在准备阶段虚拟机就会根据 ConstantValue 的设置将`value`赋值为 3。我们可以理解为`static final`常量在编译期就将其结果放入了调用它的类的常量池中。

2.3 解析：把类中的符号引用转换为直接引用

> 这个步骤，看的也有点懵逼。
>
> R 大在《JVM 符号引用转换直接引用的过程?》和《JVM 里的符号引用如何存储？》做过解答，看的还是懵逼。

解析阶段，是虚拟机将常量池内的符号引用替换为直接引用的过程。解析动作，主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符 7 类符号引用进行。

- 符号引用，就是一组符号来描述目标，可以是任何字面量。
- 直接引用，就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。

📢 3）初始化

初始化，为类的静态变量赋予正确的初始值，JVM 负责对类进行初始化，主要对类变量进行初始化。在 Java 中对类变量进行初始值设定有两种方式：

- 1、声明类变量是指定初始值。
- 2、使用静态代码块为类变量指定初始值。

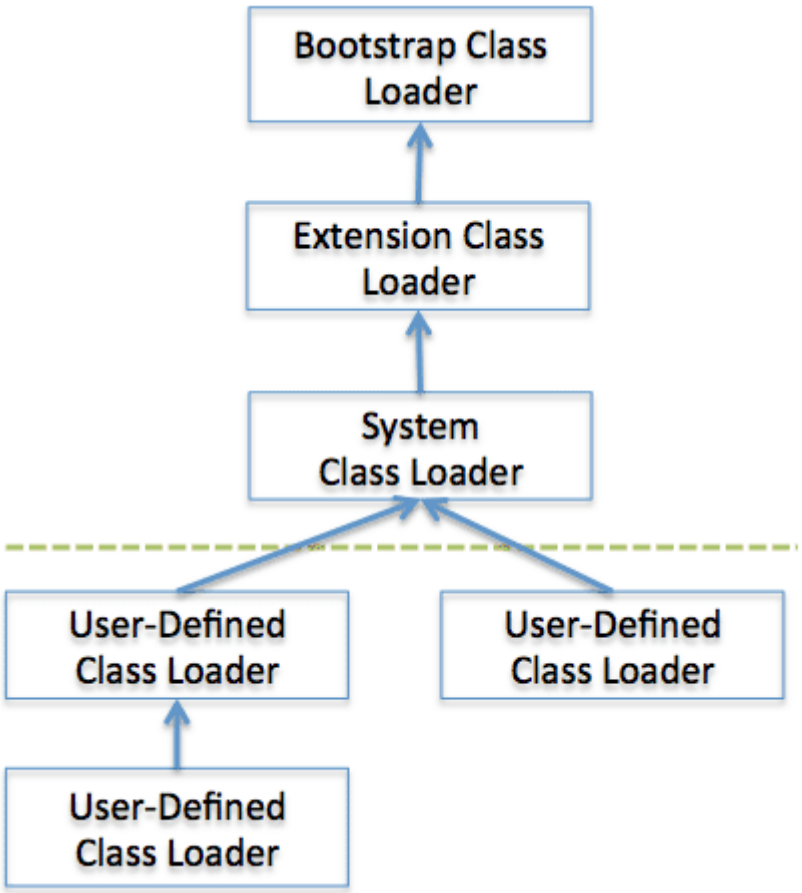
JVM 初始化步骤：

- 1、假如这个类还没有被加载和连接，则程序先加载并连接该类。
- 2、假如该类的直接父类还没有被初始化，则先初始化其直接父类。
- 3、假如类中有初始化语句，则系统依次执行这些初始化语句。

什么是双亲委派模型（Parent Delegation Model）？

> 《深入拆解 Java 虚拟机》 的 「7.4.2 双亲委派模型」 。

类加载器 ClassLoader 是具有层次结构的，也就是父子关系，如下图所示：



- Bootstrap ClassLoader： 根类加载器，负责加载 Java 的核心类，它不是 `java.lang.ClassLoader` 的子类，而是由 JVM 自身实现。

  > 此处，说的是 Hotspot 的情况下。

- Extension ClassLoader： 扩展类加载器，扩展类加载器的加载路径是 JDK 目录下 `jre/lib/ext` 。扩展加载器的 `#getParent()` 方法返回 `null` ，实际上扩展类加载器的父类加载器是根加载器，只是根加载器并不是 Java 实现的。
- System ClassLoader： 系统(应用)类加载器，它负责在 JVM 启动时加载来自 Java 命令的 `-classpath` 选项、 `java.class.path` 系统属性或 `CLASSPATH` 环境变量所指定的 jar 包和类路径。程序可以通过 `#getSystemClassLoader()` 来获取系统类加载器。系统加载器的加载路径是程序运行的当前路径。

- 该模型要求除了顶层的 Bootstrap 启动类加载器外，其余的类加载器都应当有自己的父类加载器。子类加载器和父类加载器不是以继承（Inheritance）的关系来实现，而是通过组合（Composition）关系来复用父加载器的代码。简略代码如下：

```
// java.lang.ClassLoader

public abstract class ClassLoader {

    // ... 省略其它代码

    // The parent class loader for delegation
    // Note: VM hardcoded the offset of this field, thus all new fields
    // must be added *after* it.
    private final ClassLoader parent;

}
```

- 每个类加载器都有自己的命名空间（由该加载器及所有父类加载器所加载的类组成。
  - 在同一个命名空间中，不会出现类的完整名字（包括类的包名）相同的两个类。
  - 在不同的命名空间中，有可能会出现类的完整名字（包括类的包名）相同的两个类）。
  - 类加载器负责加载所有的类，同一个类(一个类用其全限定类名(包名加类名)标志)只会被加载一次。

📣 Java 虚拟机是如何判定两个 Java 类是相同的?

Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。**只有两者都相同的情况，才认为两个类是相同的。**即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。

比如一个 Java 类 `com.example.Sample` ，编译之后生成了字节代码文件 `Sample.class` 。两个不同的类加载器 ClassLoaderA 和 ClassLoaderB 分别读取了这个 `Sample.class` 文件，并定义出两个 `java.lang.Class` 类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 ClassCastException 。

📣 双亲委派模型的工作过程?

- 1、当前 ClassLoader 首先从自己已经加载的类中，查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。

> 每个类加载器都有自己的加载缓存，当一个类被加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

- 2、当前 ClassLoader 的缓存中没有找到被加载的类的时候
  - 委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到 bootstrap ClassLoader。
  - 当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

让我们来简单撸下源码。代码如下：

> ：要不要跟面试官吹下，自己看过源码得知~

```java
// java.lang.ClassLoader
// 删除部分无关代码

protected Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
    synchronized (getClassLoadingLock(name)) {
        // 首先，从缓存中获得 name 对应的类
        Class<?> c = findLoadedClass(name);
        if (c == null) { // 获得不到
            try {
                // 其次，如果父类非空，使用它去加载类
                if (parent != null) {
                    c = parent.loadClass(name, false);
                // 其次，如果父类为空，使用 Bootstrap 去加载类
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
            }

            if (c == null) { // 还是加载不到
                // 最差，使用自己去加载类
                c = findClass(name);
            }
        }
        // 如果要解析类，则进行解析
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

为什么优先使用父 ClassLoader 加载类?

- 1、共享功能：可以避免重复加载，当父亲已经加载了该类的时候，子类不需要再次加载，一些 Framework 层级的类一旦被顶层的 ClassLoader 加载过就缓存在内存里面，以后任何地方用到都不需要重新加载。
- 2、隔离功能：主要是为了安全性，避免用户自己编写的类动态替换 Java 的一些核心类，比如 String，同时也避免了重复加载，因为 JVM 中区分不同类，不仅仅是根据类名，相同的 class 文件被不同的 ClassLoader 加载就是不同的两个类，如果相互转型的话会抛 `java.lang.ClassCaseException`。

> 这也就是说，即使我们自己定义了一个 `java.util.String` 类，也不会被重复加载。

什么是破坏双亲委托模型?

> 《深入拆解 Java 虚拟机》的「7.4.3 破坏双亲委派模型」。

正如我们上面看到的源码，破坏双亲委托模型，需要做的是，`#loadClass(String name, boolean resolve)` 方法中，不调用父 `parent` ClassLoader 方法去加载类，那么就成功了。那么我们要做的仅仅是，错误的覆盖 `##loadClass(String name, boolean resolve)` 方法，不去使用父 `parent` ClassLoader 方法去加载类即可。

想要深入的胖友，可以深入看看如下文章：

- 《Tomcat 类加载器之为何违背双亲委派模型》
- 《真正理解线程上下文类加载器（多案例分析）》提供了多种打破双亲委托模型的案例。
- 《深入拆解 Java 虚拟机》的「第 9 章 类加载及执行子系统的案例与实战」

🚀 如何自定义 ClassLoader 类?

直接参考《Java 自定义 ClassLoader 实现 JVM 类加载》文章即可。

🚀 OSGI 如何实现模块化热部署?

> ：了解即可。

OSGI 实现模块化热部署的关键，是它自定义的类加载器机制的实现。每一个程序模块都有一个自己的类加载器，当需要等换一个模块时，就把模块连同类加载器一起换掉以实现代码的热替换。

TODO 虚拟机字节码执行引擎

TODO 早期（编译期）优化

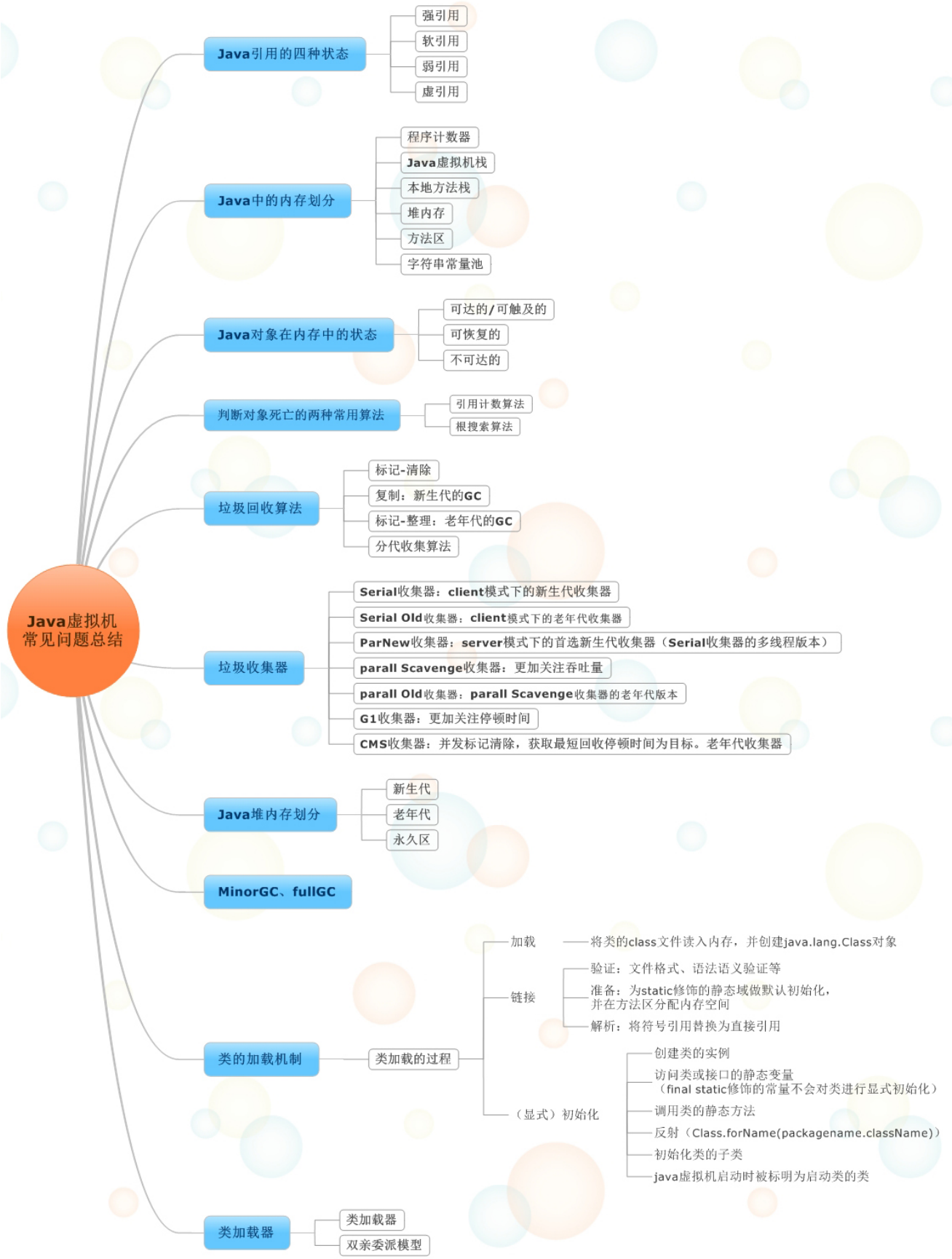
TODO JIT

有时我们会听到 JIT 这个概念，并说它是 JVM 的一部分，这让我们很困惑。JIT 是 JVM 的一部分，它可以在同一时间编译类似的字节码来优化将字节码转换为机器特定语言的过程相似的字节码，从而将优化字节码转换为机器特定语言的过程，这样减少转换过程所需要花费的时间。

TODO 晚期（运行期）优化

666. 彩蛋

总的来说，JVM 主要提问的点，如下脑图：



参考与推荐如下文章：

- 《Java 面试知识点解析(三)——JVM篇》

- 《总结的 JVM 面试题》
- 《JAVA 对象创建的过程》
- 《Java 虚拟机详解 —— JVM 常见问题总结》