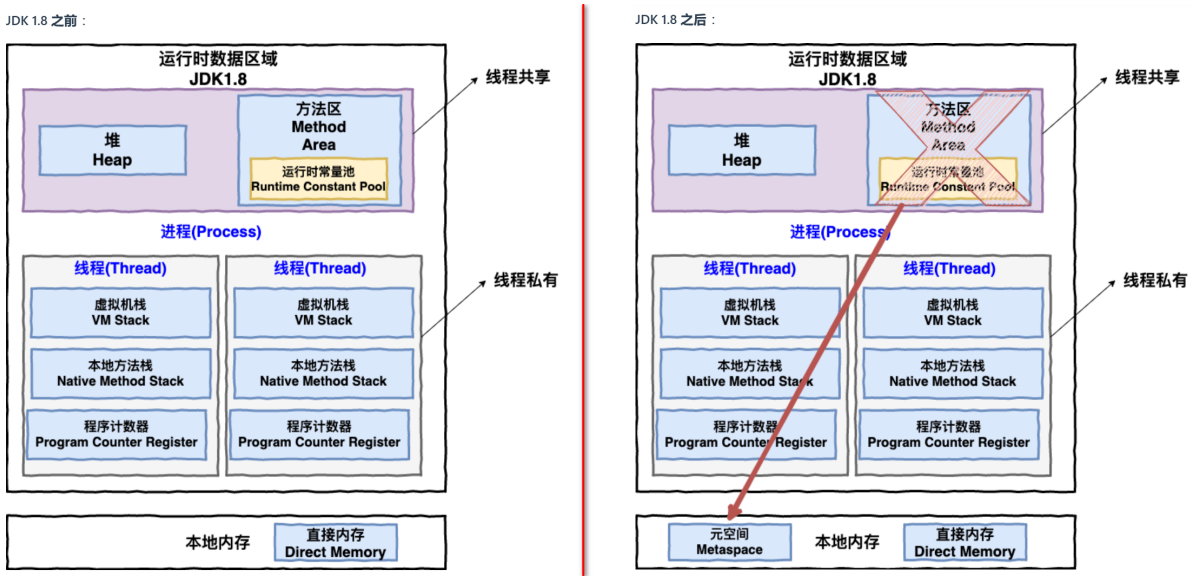
JVM

内存结构

JVM内存结构 (高)



- jdk1.6中, 方法区的实现是永久代, 而永久代是在堆中的. 方法区存储了类信息Class, 类加载器ClassLoader, 运行时常量池. 运行时常量池中存储了字符串常量池String Table
- java8中, 方法区的实现是元空间, 元空间在本地内存(操作系统内存)中. 方法区同样存储了类信息Class, 类加载器ClassLoader, 运行时常量池. 而字符串常量池String Table 被放在了堆区

JVM内存结构中哪些是线程私有的? 哪些是内存共享的? (高)

线程私有的:

- 程序计数器
  - 程序计数器保存了下一条指令的执行地址, 所以解释器才能读取下一条指令然后执行.

    - 为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器, 所以程序计数器是线程私有的
    - 因为只保存下一条指令的地址, 程序计数器是唯一一个不会出现 `OutOfMemoryError` (内存溢出)的内存区域
- 虚拟机栈
    - 与程序计数器一样，虚拟机栈也是线程私有的，它的生命周期和线程相同，随着线程的创建而创建，随着线程的死亡而死亡。
    - 每个方法执行时都会创建一个桢栈来存储方法的局部变量表、操作数栈、动态链接、方法返回地址等信息。栈的大小决定了方法调用的可达深度（递归多少层次，或嵌套调用多少层其他方法）
    - 虚拟机栈抛出的异常一般是 `StackOverflow`.
- 本地方法栈
    - 与虚拟机栈作用相似。虚拟机需要用到c或者c++写的一些本地方法(native修饰的方法), 这些本地方法运行时使用的内存就是本地方法栈.
    - 本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、方法返回地址信息。

线程共享的:

- 堆
    - 堆是线程共享的, 堆中的对象需要考虑线程安全问题.
    - java中几乎所有的对象实例以及数组存储到堆中. 但是随着逃逸分析技术不断成熟, java7已经默认开启了逃逸分析, 如果方法中某些对象没有被返回, 也没有被外界使用(就是没有逃逸出去), 那么对象可以直接在栈上分配内存.

    - 堆是垃圾收集器管理的主要区域，由于现在收集器基本都采用分代垃圾收集算法，从垃圾回收的角度来看, Java 堆还可以细分为：新生代和老年代. 在java8以前, 还有堆中还有永久代, java8之后就永久代被元空间取代, 位置也放到了本地内存中.
- 方法区
    - jdk1.6中, 方法区的实现是永久代, 而永久代是在堆中的. 方法区存储了类信息Class, 类加载器ClassLoader, 运行时常量池. 运行时常量池中存储了字符串常量池String Table
    - java8中, 方法区的实现是元空间, 元空间在本地内存(操作系统内存)中. 方法区同样存储了类信息Class, 类加载器ClassLoader, 运行时常量池. 而字符串常量池String Table 被放在了堆区
- 直接内存 (非运行时数据区的一部分)

JVM内存模型里的堆和栈有什么区别 (高)

- 用途
    - 栈主要用于存储局部变量、方法调用的参数、方法返回地址以及一些临时数据。每当一个方法被调用，一个栈帧（stack frame)就会在栈中创建，用于存储该方法的信息，当方法执行完毕，栈帧也会被移除。
    - 堆用于存储对象的实例。当你使用new关键字创建一个对象时，对象的实例就会在堆上分配空间。
- 生命周期
    - 栈中的数据具有确定的生命周期，当一个方法调用结束时，其对应的栈帧就会被销毁，栈中存储的局部变量也会随之消失。
    - 堆中的对象生命周期不确定，对象会在垃圾回收机制检测到对象不再被引用时才被回收。

- 存储
  - 栈的空间相小. 当栈溢出时，通常是因为递归过深
  - 堆往往比较大. 堆溢出通常是由于创建了太多的大对象或未能及时收不再使用的对象
- 可见性
  - 栈中的数据对线程是私有的，每个线程有自己的栈空间。
  - 堆中的数据对线程是共享的，所有程都可以访问堆上的对象。

JVM内存结构的常见问题 (中)

- 垃圾回收是否涉及栈内存?
  - 方法运行结束, 自动出栈, 不需要垃圾回收.
- 栈内存分配越大越好吗?
  - 栈内存分配太大, 只能让方法递归的次数变多. 而且会让的线程数变少.
- 方法内的局部变量是否线程安全?
  - 栈是线程私有的, 而方法是放到栈里的, 那么多线程里的栈都是不同的, 里面的方法和方法的局部变量不共享, 自然是线程安全的.
- 如何判断方法中的一个变量是否为线程安全的? (逃逸分析)
  - 方法逃逸： 指在某一个方法中的对象，在该方法外部可以继续访问这个对象, 可以理解成对象跳出了方法。
    - 比如, 该方法的中的一个对象是通过参数传递过来的或者该方法将某对象return出去, 那么该方法的外边就能访问到这个对象, 从而造成线程安全问题.

- 线程逃逸：这个对象被其他线程访问到，比如赋值给了全局变量(类属性, 独立于方法之外)，并被其他线程访问到了。对象逃出了当前线程。

如果有个大对象一般是在哪个区域？ (中)

- 大对象通常会直接分配到老年代。
- 新生代主要用于存放生命周期较短的对象，并且其内存空间相对较小。如果将大对象分配到新生代，可能
  会很快导致新生代空间不足，从而频繁触发MinorGC
- 将大对象直接分配到老年代，可以减少新生代的内存压力，降低MinorGC的频率。
- 大对象通常需要连续的内存空间，如果在新生代中频繁分配和回收大对象，容易产生内存碎片，导致后续分配大对象时可能因为内存不连续而失败。
- 老年代的空间相对较大，更适合存储大对象，有助于减少内存碎片的产生。

字符串常量池是什么 (中)

字符串常量池 是 JVM 为了提升性能和减少内存消耗针对字符串（String 类）专门开辟的一块区域，主要目的是为了避免字符串的重复创建。

JDK1.7 之前，字符串常量池存放在永久代。JDK1.7 字符串常量池和静态变量从永久代移动了 Java 堆中。

内存泄漏和内存溢出有什么不同 (高)

内存泄漏是指程序在运行过程中不再使用的对象仍然被引用，从而无法被垃圾收集器回收，导致可用内存逐渐减少

内存泄露常见原因：

- 静态集合：使用静态数据结构（如HashMap或ArrayList)存储对象，且未清理
- 线程：未停止的线程可能持有对象引用，无法被回收
- ThreadLocal: ThreadLocal使用后不调用remove, 就可能发送内存泄漏

内存溢出是指JVM在申请内存时，没有足够的内存，最终引发OutofMemoryError。往往发生在堆内存不足以存放新创建的对象时.

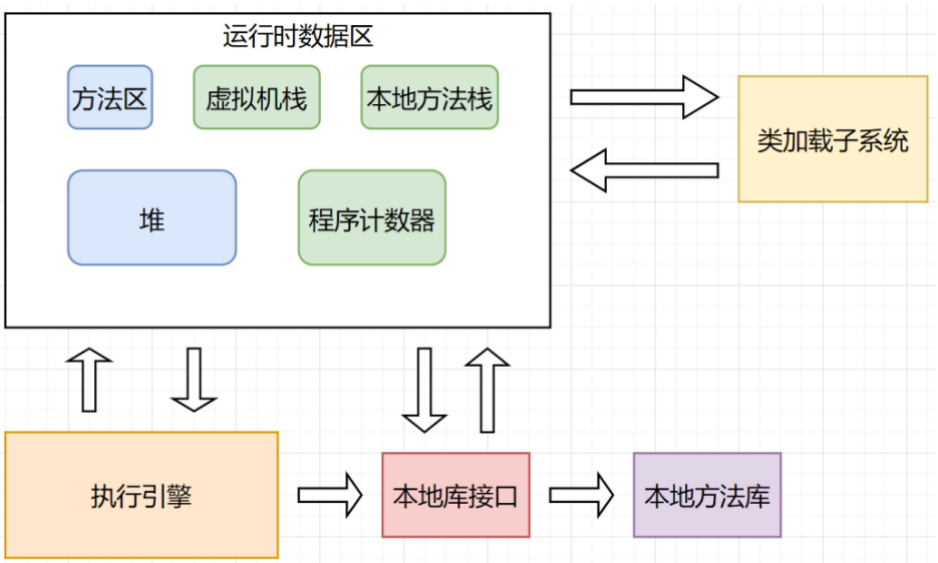
内存溢出常见原因：

- 大量对象创建：短时间程序中不断创建大量对象, 且无法回收, 超出JVM堆的限制就会OOM.
- 内存泄漏: 内存泄漏之后, 可用内存减少, 一直泄漏下去, 就很容易出现OOM.
- 持久引用：大型数据结构（如缓存、集合等）长时间持有对象引用，导致内存累积.
- 递归调用：深度递归导致栈溢出, 栈溢出也是内存溢出的一种.

jvm有哪几种内存溢出的情况 (中)

- 堆内存溢出：当出现Java.lang.OutOfMemoryError:Java heap space异常时，就是堆内存溢出了。原因是代码中可能存在大对象分配，或者发生了内存泄露，导致在多次GC之后，还是无法找到一块足够大的内存容纳当前对象
- 栈溢出：如果我们写一段程序不断的进行递归调用，而且没有退出条件，就会导致不断地进行压栈。类似这种情况，JVM实际会抛出StackOverFlowError;当然，如果JVM试图去扩展栈空间的的时候失败，则会抛出OutOfMemoryError
- 元空间溢出：元空间的溢出，系统会抛出Java.lang.OutOfMemoryError:Metaspace。出现这个异常的问题的原因是系统的代码非常多或引用的第三方包非常多或者通过动态代码生成类加载等方法，导致元空间的内存占用很大。
- 直接内存内存溢出：在使用ByteBuffer中的allocateDirect()的时候会用到，很多JavaNIO(像netty)的框架中被封装为其他的方法，出现该问题时会抛出Java.lang.OutOfMemoryError:Direct buffer memory异常。

JVM 的主要组成部分及其作用 (中)



JVM包含两个子系统和两个组件，分别为

- Class loader(类装载子系统)：根据给定的全类名来装载class文件到运行时数据区的方法区中
- Execution engine(执行引擎子系统)：执行字节码文件中的指令
- Runtime data area(运行时数据区组件): 即我们常说的JVM的内存
- Native Interface(本地接口组件)：与native lib交互，是与其它编程语言的本地库交互的接口

首先通过编译器把 Java源代码转换成字节码，Class loader(类装载)再把字节码加载到内存 中，将其放在运行时数据区的方法区内，而字节码文件只是 JVM 的一套指令集规范，并不能直 接交给底层操作系统去执行，因此需要特定的命令解析器执行引擎（Execution Engine），将 字节码翻译成底层系统指令，再交由 CPU 去执行，而这个过程中需要调用其他语言的本地库 接口（Native Interface）来实现整个程序的功能。

强引用、软引用、弱引用、虚引用（虚引用与软引用和弱引用的区别、使用软引用能带来的好处）（中）

强引用

```
String str = "123";
```

这是我们日常用的引用, **只要对象是强引用, 如果内存不足时, JVM宁愿抛出OutOfMemoryError内存溢出错误也不会回收强引用**

如果想要JVM回收强引用类型的对象,将其引用更改为null, JVM会在合适的时间回收这个null引用对象.

软引用

描述有些还有用但并非必需的对象。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进行二次回收。如果这次回收后还没有足够的内存，才会抛出内存溢出异常。Java中的类SoftReference表示软引用。

```
SoftReference<String> srs = new SoftReference<String>("123");
System.out.println(srs.get());  // 123
System.gc();
System.out.println(srs.get()); // 123
//因为内存充足所以无法回收
```

被软引用的对象, 内存充足时, gc不会回收. 内存不足时, gc会将其回收.

软引用适合做缓存

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。

弱引用

java中使用WeackReference来表示弱引用, 描述非必需对象。被弱引用关联的对象只能生存到下一次垃圾回收之前，垃圾收集器工作之后，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

```
WeakReference<String> str = new WeakReference<String>(new String("123"));
System.out.println(str.get()); // 123
System.out.println(str.getClass()); // class java.lang.ref.WeakReference
System.gc();
System.out.println(str.get()); //null
```

可以看出, **一旦垃圾回收线程发现了只具被弱引用关联的对象，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。

虚引用

java中使用PhantomReference来表示虚引用, 引用就好事虚设一样, 就像某个对象没有引用关联一样, 若某个对象与虚引用关联, 那么任何时候都可能被虚拟机回收, 虚引用不能单独使用, 必须配合引用队列一起使用.

```
1 PhantomReference<String> ph = new
  PhantomReference<String>("123", new
  ReferenceQueue<String>());
2 System.out.println(ph.get()); // null
3 // get方法没有获取到值
```

当垃圾回收器准备回收一个对象时, 如果发现它与虚引用关联, 就会在他回收以前将这个虚引用加入到引用队列中(**要被回收的虚引用对象会被放入引用队列中**),程序可以判断引用队列中是否加入了虚引用, 来了解被引用的对象是否将要被回收, 如果确实要被回收, 就可以做一些回收之前的收尾工作.

虚引用主要用来跟踪对象被垃圾回收的活动，主要用来实现比较精细的内存使用控制， **当某个对象被回收时, 做事一些事情.** 比如ByteBuffer的实现类内部，使用了Cleaner(虚引用）来监测ByteBuffer对象，一旦ByteBuffer对象被垃圾回收，那么就会由ReferenceHandler线程通过Cleaner的clean方法调用 `unsafe.freeMemory()` 来释放直接内存

垃圾回收

垃圾回收算法是什么，为了解决了什么问题？（高）

- 垃圾回收算法是为了解决内存管理的问题。
- 在传统的编程语言中，开发人员需要手动分配和释放内存，这可能导致内存泄漏、内存溢出等问题。
- Java为了提供更简单、更安全的编程环境，因此引入了垃圾回收机制来自动管理内存。
- 垃圾回收机制的主要目标是自动检测和回收不再使用的对象，从而释放它们所占用的内存空间。这样可以避免内存泄漏。
- 同时，垃圾回收机制还可以防止内存溢出（即程序需要的内存超过了可用内存的情况）。
- 通过垃圾回收机制，JVM可以在程序运行时自动识别和清理不再使用的对象，使得开发人员无需手动管理内存

如何判断对象是否死亡/如何判断对象可以被回收 (高)

- 引用计数法
  - 引用计数法就是当一个对象被引用一次, 计数就+1, 再被其他对象引用一次, 计数就为2. 如果一个对象计数为1, 那么它就可以被回收了.
  - 引用计数法存在的问题是, 当出现循环引用, a引用b, b引用a. 那么他们的引用计数都等于1, 则永远无法被回收. 但其实这时他们已经不被需要应该被回收.
- 可达性分析

- 从GC Roots开始向下搜索，搜索所走过的路径称为引用链. GC Root直接或者间接引用的对象就是不可被回收的. 而根对象没有直接或间接引用的对象就是要被回收的对象.

垃圾收集有哪些算法，各自的特点？（高）

GC最基础的算法有三种： 标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

- 标记 -清除算法
  - 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掉所有被标记的对象。
  - 适用于**存活对象多，垃圾较少**的情况.
  - 缺点是会产生内存碎片, 需要维护一个空闲列表来进行分配.
- 标记整理算法/标记压缩算法
  - 算法分为“标记”和“整理”两个阶段：首先标记出所有需要回收的对象，在标记完成后让所有存活的对象都向一端移动，然后直接清理掉另一端的内存.
  - 标记整理算法适用于**存活对象多，垃圾较少**的情况.
  - 优点是不会产生内存碎片, 消除了复制算法中内存减半的缺点
  - 缺点是效率不如复制算法, 需要STW
- 复制算法
  - 执行过程
    - 将原有的内存空间一分为二，每次只用其中的一块
    - 在垃圾回收时，将正在使用的对象复制到另外一个内存空间中，然后将该内存空间清空，交换两个内存的角色，完成垃圾回收。
  - 优点

        - 没有标记和清除过程，**实现简单，运行高效。**
        - 复制过去以后**保证空间的连续性**，**不会出现“碎片”**问题。
    - 缺点
        - 需要**两倍的内存空间, 或者需要将原本的内存一分为二。**
        - 复制而不是移动，意味着 GC 需要维护对象引用关系，不管是内存占用或者时间开销也不小。
    - 复制算法的高效性是建立在**存活对象少、垃圾对象多**的前提下的。
        - 这种情况在**新生代经常发生**，但是在老年代，更常见的情况是大部分对象都是存活对象。
        - 如果依然使用复制算法由于存活对象较多，复制的成本也将很高。因此，基于老年代垃圾回收的特性，更适合标记整理算法.
- 分代收集算法

分代收集算法 (高)

目前java采用分代收集算法, 将分为堆内存分为新生代和老年代.

新生代

所有新生成的对象首先都是放在新生代的, 新生代的目标就是尽可能快速的收集掉那些生命周期短的对象。新生代代分三个区。一个Eden区(伊甸园)，两个Survivor区(from和to, 有的也叫s1和s0)

- 首先, 大部分对象都会在Eden区生成

- 当Eden区空间不足时, 就会触发一次minorGC, 此时会采用复制算法, 将存活的对象从Eden区复制到to区, 存活对象的寿命+1, 清除Eden区的垃圾. 然后将from区和to区交换, from区就保留了存活的对象.
- 此时, 新的对象又在Eden区生成. 当Eden区空间不足时, 又会触发MinorGC, 此时会采用复制算法, 将Eden区和from区的存活对象复制到to区, 且寿命+1. 清除Eden区的垃圾, 然后将from和to区交换, from区保留了存活的对象.
- 以此类推, 每当Eden区空间不足时, 就触发MinorGC, 采用复制算法, 将Eden和from区的存活对象复制到to区, 且寿命+1, 清除Eden区的垃圾, 然后将from和to区交换, from区保留了存活的对象.
- Minor GC会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。
- 当存活对象的寿命超过阈值时, 就将该对象从新生代转移到老年代. 因为该对象一直存活说明十分重要, 应该放到gc不频繁的老年代.

注意: 在minor gc中, 会引发stop the world, 垃圾回收线程会暂停其他线程, 当垃圾回收完后, 其他线程才能执行. 因为gc时, 对象地址有可能改变, 此时用户线程运行会出问题.

老年代

- 当老年代空间不足, 会先尝试触发minor gc清理新生代, 如果空间依旧不足, 那么就会触发full gc, 清理老年代.
- full gc也会触发SWT(stop the world), 但是相比于minor gc, full gc的暂停时间更长.
- 如果full gc后空间足够, 那就继续运行. 如果full gc后空间不足, 那么抛出out of memory error

常见的垃圾回收器有哪些？（中）

这里先讲一下并发和并行的区别

- **并行（Parallel）**：指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（Concurrent）**：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

串行垃圾收集器

- Serial收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效； **全程STW**
- Serial Old收集器 (标记-整理算法): 老年代单线程收集器，Serial收集器的老年代版本；**全程STW**

并行垃圾收集器

- ParNew收集器 (复制算法): 新生代收并行集器，实际上是Serial收集器的多线程版本，在多核 CPU环境下有着比Serial更好的表现； **全程STW**
- Parallel Scavenge收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC线程时间)，高吞吐量可以高效率的利用CPU时间，尽 快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景； **全程STW**
- Parallel Old收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge收集器的老年代版本； **全程STW**

并发垃圾收集器

- CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有**并发收集、低停顿**的特点，追求**最短GC回收停顿时间。**
- G1(Garbage First)收集器 (标记-整理算法)： Java堆并行收集器，G1收集器是JDK1.7提供的一 个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收 集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年 代)，而前六种收集器回收的范围仅限于新生代或老年代。

新生代收集器：Serial、 ParNew 、 Parallel Scavenge

老年代收集器： CMS 、Serial Old、Parallel Old

整堆收集器： G1

介绍一下 CMS (高)

CMS是老年代并行垃圾收集器, 采取标记清除算法, 追求最短GC回收停顿时间, 优点是并发收集, 低停顿.

CMS 处理过程有四个步骤：

1. **初始标记，指的是寻找所有被 GCRoots 引用的对象，该阶段需要STW。** 这个步骤仅仅只是标记一下 GC Roots 能直接关联到的对象，并不需要做整个引用的扫描，因此速度很快。
2. **并发标记，指的是对「初始标记阶段」标记的对象进行整个引用链的扫描，该阶段与用户线程同时运行, 不需要STW**
3. **重新标记，指的是对「并发标记」阶段出现的问题进行校正，该阶段需要STW.** 由于垃圾回收算法和用户线程并发执行，虽然能降低响应时间，但是会发生漏标和多标的问题。
4. **并发清除，指的是将标记为垃圾的对象进行清除，该阶段与用户线程同时运行, 不需要STW.**

CMS 之所以能极大地降低 GC 停顿时间，本质上是将原本冗长的引用链扫描进行切分。通过 GC 线程与用户线程并发执行，加上重新标记校正的方式，减少了垃圾回收的时间。

CMS缺点是:

- 多线程并发处理, 吃CPU资源
- 无法处理浮动垃圾, 在 CMS 进行并发清理的时候，用户线程同时在运行，也会产生一些浮动垃圾。
- 采用标记清除算法会产生空间碎片

介绍一下 G1 (高)

特点

- 同时注重吞吐量(Throughput)和低延迟(Low latency), 默认的暂停目标是200ms
- 将堆内存化整为零，会将堆划分为多个大小相等的Region，内存的回收是以region作为基本单位的
- 整体上看是标记整理算法，微观上看两个区域之间是复制算法

G1 垃圾收集过程主要分为4个阶段: (这里以深入理解java虚拟机为标准)

1. 初始标记
    - 标记了从GC Root开始直接关联可达的对象
    - 触发STW
2. 并发标记

    - 在老年代占用堆空间比例到达阈值时, 从GC Root开始对堆中对象进行可达性分析, 递归扫描整个堆里的对象图, 标记出所有回收对象
    - 和用户程序并发执行, 不会STW
3. 最终标记
    - 并发标记时, 其他 用户线程也会工作, 会产生新的对象, 遗弃旧对象 (处理漏标, 错标)
    - 触发STW
4. 筛选回收
    - 先对Region的回收价值进行排序，然后根据期望暂停时间，选择性回收Region
    - 不追求一次全部清理完, 优先回收垃圾最多的区域
    - 回收时采用复制算法，多条收集器线程并发执行
    - 触发STW

如何选择垃圾收集器 (中)

- 较小的JVM内存用CMS, 因为内存大CMS回收垃圾时会比较慢.
- 较大的JVM内存用G1, 因为G1需要用卡表和RSet存储新生代老年代跨代引用关系, 内存占用比较大.
    - 卡表记录了当前regin引用了哪个Regin的对象
    - RSet记录了其他Region中的对象引用本Region中对象的关系

什么情况下使用CMS,什么情况使用G1? (中)

CMS适用场景：

- 低延迟需求：适用于对停顿时间要求敏感的应用程序。

- 老生代收集：主要针对老年代的垃圾回收。
- 碎片化管理：容易出现内存碎片，可能需要定期进行FullGC来压缩内存空间。

G1适用场景：

- 大堆内存：适用于需要管理大内存堆的场景，能够有效处理数GB以上的堆内存。
- 对内存碎片敏感：G1通过紧凑整理来减少内存碎片，降低了碎片化对性能的影响。
- 比较平衡的性能：G1在提供较低停顿时间的同时，也保持了相对较高的吞吐量。

G1回收器的特色是什么？（中）

G1的特点：

- G1最大的特点是引入分区的思路，弱化了分代的概念。
- 合理利用垃圾收集各个周期的资源，解决了其他收集器、甚至CMS的众多缺陷

G1相较其他收集器的改进：

- 算法：G1基于标记--整理算法，不会产生空间碎片，在分配大对象时，不会因无法得到连续的空间，而提前触发一次FULLGC。
- 停顿时间可控：G1可以设置预期停顿时间。
- 并行与并发：G1能更充分的利用CPU多核环境下的硬件优势，来缩短stw的停顿时间。

Minor Gc 和 Full GC 有什么不同呢？Minor GC与Full GC分别在什么时候发生？ (高)

MinorGC是发生在新生代的垃圾收集动作，所以Minor GC非常频繁，一般回收速度也比较快。

Full GC 指的是针对新生代、老年代、永久代的全体内存空间的垃圾回收. FullGC回收速度更慢, 发生频率也更低.

什么时候出发MinorGC

- Eden区域满了
- 新创建的对象大小 > Eden所剩空间

什么时候会触发fullGC (高)

- System.gc()方法的调用
- 老年代空间不足
- 永久代空间不足 (元空间时再本地内存, 由OS进行管理, 不会发生垃圾回收)
- 统计得到的Minor GC晋升到老年代的平均大小大于老年代的剩余空间
- 堆中分配很大的对象
    - 所谓大对象，是指需要大量连续内存空间的java对象，例如很长的数组，此种对象会直接进入老年代，而老年代虽然有很大的剩余空间，但是无法找到足够大的连续空间来分配给当前对象，此种情况就会触发JVM进行Full GC。

类加载

什么是类加载器? 常见的类加载器? (中)

名称	加载哪儿的类	说明
Bootstrap ClassLoader	JAVA_HOME/jre/lib	无法直接访问
Extension ClassLoader	JAVA_HOME/jre/lib/ext	上级为Bootstrap, 显示为null
Application ClassLoader	classpath	上级为Extension
自定义类加载器	自定义	上级为Application

Bootstrap ClassLoader：启动类加载器

- 最顶层的加载类，由C++实现，负责加载`JAVA_HOME/jre/lib`目录下的jar包和类，(如String、System等)

Extension ClassLoader：扩展类加载器

- 它负责加载JRE的扩展目录（`%JAVA_HOME%/jre/lib/ext`）中JAR包的类

Application ClassLoader：系统类加载器

- 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

双亲委派模式 (高)

双亲委派模式:

- 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。
- 加载器加载类时先把请求委托给自己的父类加载器执行, 直到顶层的启动类加载器. 父类加载器能够完成加载则成功返回, 不能则子类加载器才自己尝试加载.
- 简而言之: **自底向上检查类是否被加载, 自顶向下尝试加载类**

优点:

- 避免类的重复加载
- 避免Java的核心API被篡改, 保证安全

为什么要打破双亲委派模型 (高)

但是使用双亲委派也存在一定的局限性，在正常情况下，用户代码是依赖核心类库的，所以按照正常的双亲委派加载流程是没问题的；

但是在加载核心类库时，如果需要使用用户代码，双亲委派流程就无法满足；

比如在使用JDBC时， 利用DriverManager.getConnection获取连接时，就会存在这样的问题。

- DriverManager是由Bootstrap ClassLoader加载的，在加载DriverManager时，会执行其静态方法，加载初始驱动程序，也就是Driver接口的实现类；
- 但是这些实现类基本都是第三方厂商提供的，根据双亲委派原则，第三方的类不可能被Bootstrap ClassLoader加载。

- 所以这时候就需要打破双亲委派模型来进行加载.