JVM中的垃圾回收

版本号:2018/09/27-1(23:55)

垃圾回收(42)

1、什么是垃圾回收?

> 1. 将已经分配出去，但是不再使用的内存回收起来。以便于再次分配。

2、在JVM中垃圾回收中的垃圾所占的内存空间具体是指什么?

> 死亡对象所占据的堆空间

3、JVM垃圾回收的关键?如何判断一个对象是存活还是死亡的?

> 1. 关键是：如何判断判断一个对象是存活还是死亡的
> 2. 有两个方法判断:
>   1. 引用计数法
>   1. 可达性分析

引用计数法

4、引用计数法是什么?

> 1. reference counting
> 2. 一种古老的辨别方法
> 3. 为每个对象添加引用计数器，记录指向该对象的引用的个数
> 4. 引用计数一旦为0，代表该对象已经死亡。

5、引用计数法的实现思路?

1. 一个引用指向一个对象，该对象的引用计数+1
2. 指向一个对象的引用，赋值为其他值，该对象的引用计数-1

6、引用计数法的缺点?

1. 需要为空间来存储计数器
2. 繁琐的更新操作：需要截获所有引用更新操作，并且相应加减对象的引用计数器。
3. 重要缺陷是无法处理循环引用的对象

7、什么是引用更新操作?

1. 将引用指向一个对象，或者赋值为其他值。就是在更新引用。

8、为什么无法处理循环引用的对象?

1. a和b之间相互引用，但是没有其他引用指向a和b。a和b实际上已经死了。可是因为计数器的值不为0，导致无法回收。
2. 最终导致内存泄漏

可达性分析

9、JVM主流的垃圾回收器采用可达性分析算法，可达性分析算法是什么?

1. 本质将一系列gc roots作为初始的存活对象集合(live set)
2. 会从该集合出发，探索所有能够被引用到的对象，并且将其加入到存活对象集合中。
3. 这就是标记过程(mark)，没有被标记的对象，就判定为死亡。

10、垃圾回收中的标记过程是干什么的?

1. 会从GC roots出发，探索所有能够被引用到的对象，并且将其加入到存活对象集合中。标记过的对象，就判定为存活。

11、GC roots是什么?

一种由堆外指向堆内的引用

12、GC roots有哪些?

1. Java方法栈帧中的局部变量(虚拟机栈的栈帧中的本地变量表,所引用的对象)
2. 已加载类的静态变量(方法区中类静态属性,所引用的对象)
3. 方法区中常量,所引用的对象
4. JNI Handles(本地方法栈中JNI-Native方法,所引用的对象)
5. 已启动且未停止的Java线程

13、可达性分析如何解决的循环引用问题?

1. 虽然a和b互相引用，直到GC Roots无法达到a和b，就不会将其添加到存活集合中。判定为死亡。

14、可达性分析算法在实践应用中的问题?

1. 多线程中的误报和漏报
2. xxx

15、可达性分析在多线程中遭遇的误报具体是什么?

1. 多线程中，可达性分析时已经访问到了该对象，认为该对象是存活的。然而这之后其他线程将其引用设置为null。
2. 会导致错过本次可以回收的机会。因为设置为了null应该回收对象，却被认为是存活的。
3. 误报这种情况问题不大，下次垃圾回收时依旧能进行回收。

16、可达性分析在多线程中遭遇的漏报具体是什么?

1. 多线程中，可达性分析时访问不到某对象，所以认为该对象是死亡的。与此同时其他线程又持有了该对象的引用。在垃圾回收后，对该对象的访问会导致JVM崩溃!
2. 漏报这种情况问题非常严重!

Stop-The-World

17、如何解决可达性分析算法在多线程中的问题?

1. 采用简单粗暴的方式， Stop the worlds(STW)
2. 在垃圾回收时，停止其他所有非垃圾回收的线程，直到垃圾回收完成。
3. 会造成垃圾回收中所谓的暂停时间-GC pause

18、STW有什么问题?

会造成GC pause

19、垃圾回收中的GC pause是什么?

垃圾回收中，需要停止所有非垃圾回收线程从而导致了暂停时间

20、JVM中的STW是如何实现的?

1. 通过安全点(safe point)来实现
2. JVM收到STW请求后，会等到所有线程达到安全点，才会让请求STW的线程开始独占工作

21、安全点是什么?

1. 一种机制，保证线程能找到一种稳定的执行状态(也就是JVM堆栈不会发生改变)

22、安全点的本质目的是为了让其他线程停下?

> 并不是。
>
> 1. 安全点是为了让线程找到一个稳定的执行状态
> 2. 在这个状态下，JVM的堆栈不会发生变化。
> 3. JVM会将处于稳定状态的代码作为一个安全点，只要不离开该安全点，JVM就能在垃圾回收时，继续运行这段本地代码。
> 4. 最终保证垃圾回收器能够安全地进行可达性分析。

23、怎样才是一个稳定的执行状态(JVM堆栈不会发生改变)?

> 1. 举个例子，当Java程序通过JNI执行本地代码时，如果这段代码 不访问Java对象 、 不调用Java方法 或者 不返回至原Java方法
> 2. 那么 Java 虚拟机的堆栈不会发生改变，也就代表着这段本地代 码可以作为同一个安全点。
> 3. 只要不离开这个安全点，Java 虚拟机便能够在垃圾回收的同时，继续运行这段本地代码。

24、Java线程具有哪几种执行状态?(4种)

> 1. 执行JNI本地代码
> 2. 解释执行字节码
> 3. 执行即时编译器生成的机器码
> 4. 线程阻塞

25、执行JNI本地代码如何进入安全点?

> 1. 访问Java对象 、 调用Java方法 或者 返回至原Java方法 这三个操作需要通过 JNI 的 API 来完成
> 2. JVM只需要在API的入口进行安全点检测(safepoint poll)
> 3. 测试是否有其他线程请求停留在安全点里(其他线程请求STW)，便可以在必要的时候挂起当前线程。

26、安全点检测是干什么的?

> 1. 测试是否有其他线程请求停留在安全点里(检测是否有其他线程发送STW请求)
> 2. 安全点检测位于哪些会造成执行状态不稳定的API入口处
> 3. 当其他线程有STW请求，表明不能去执行这些不稳定的API，因此该安全点检测会将线程挂起。
> 4. 当其他线程没有STW请求，因此去执行这些不稳定的API不会造成什么影响。

27、线程阻塞的安全点检测?

> 1. 阻塞的线程由于处于JVM线程调度器的掌控之下，因此属于安全点。
> 2. 不需要安全点检测

28、执行JNI本地代码、解释执行字节码、执行即时编译器生成的机器码这三种运行状态需要虚拟机保证在可预见的时间内进入安全点，不然会怎么样?

> 否则，垃圾回收线程可能长期处于等待所有线程进入安全点的状态，从而变相地提高了垃圾回收的暂停时间。

29、解释执行字节码时，需要如何进入安全点?

> 1. 对于解释执行来说，字节码与字节码之间皆可作为安全点。
> 2. JVM采取的做法是，当有安全点请求时，执行一条字节码便进行一次安全点检测。

30、执行即时编译器生成的机器码，需要如何进行安全点检测?

> 1. 这种情况比较复杂。这些代码直接运行在底层硬件之上，不受 JVM掌控，
> 2. 因此在生成机器码时，即时编译器需要插入安全点检测，以避免机器码长时间没有安全点检测的情况。
> 3. HotSpot 虚拟机的做法是在两处插入安全点检测：
>   1. 生成代码的方法出口处
>   1. 非计数循环的循环回边（back-edge）处

31、非计数循环的循环回边处是什么?

32、为什么不在每一条机器码或者每一个机器码基本块处插入安全点检测呢?

> 原因主要有两个:
>
> 1. 第一，安全点检测本身也有一定的开销。过多对性能有影响
> 2. 第二，即时编译器生成的机器码打乱了原本栈桢上的对象分布状况。因此会需要额外的信息，而这些信息需要很多空间来存储，即时编译器会尽量避免过多的安全点检测。

33、HotSpot虚拟机如何简化机器码中的安全点检测?

> 1. 已经将机器码中安全点检测简化为一个 内存访问操作 。
> 2. 在有安全点请求的情况下，JVM会将安全点检测访问的内存所在的页设置为不可读
> 3. 并且定义一个 segfault 处理器，来截获因访问该不可读内存而触发segfault的线程，并将它们挂起。

34、HotSpot如何解决即时编译器生成的机器码打乱了原本栈桢上的对象分布状况的这个问题?

> 1. 这个问题影响了安全点检测
> 2. 在进入安全点时，机器码需要提供一些额外的信息来帮助垃圾回收器能够枚举 GC Roots:
>   1. 这些信息能表明哪些寄存器，或者当前栈帧上的哪些内存空间存放着指向对象的引用。
> 3. 但是由于这些信息需要不少空间来存储，因此即时编译器会尽量避免过多的安全点检测。

35、不同的即时编译器插入安全点检测的位置也可能不同?有什么不同?

> 1. Graal 为例，除了生成代码的方法出口处、非计数循环的循环回边（back-edge）处外，还会在 计数循环的循环回边处 插入安全点检测。
> 2. 其他的虚拟机也可能选取方法入口而非方法出口来插入安全点检测。

3. 本质目的都是在可接受的性能开销以及内存开销之内，避免机器码长时间不进入安全点的情况，间接地减少垃圾回收的暂停时间。

36、安全点除了垃圾回收外还有哪些场景可以利用该机制?

1. JVM其他一些对堆栈内容的一致性有要求的操作时。

37、STW的机制非常不友好，有哪些解决之道？原理是什么?

1. 采用并行GC可以减少需要STW的时间
2. 它们会在即时编译器生成的代码中加入 写屏障 或者 读屏障

38、GC时有时候会突然出现较长的时间消耗，是为什么?

1. 这就是长暂停
2. 一般Full Gc会造成长暂停

39、Full GC有卡顿，对性能很不利，该如何避免?

1. 通过调整新生代大小，使对象在其生命周期内都处于新生代中。
2. 这样 `Minor GC` 就能收集完这些短命对象，而不需要去Full GC

40、多线程为什么会导致误报和漏报?

没有引入STW的时候才会出现。现代JVM引入STW后不存在该问题。

安全点检测的性能

41、有安全点和无安全点的性能差距有多少?

1. 无安全点检测的计数循环带来的长暂停:foo()运行时间为11884ms
2. 有安全点检测: bar()运行时间为3628ms

```
// time java SafepointTestp /
// 你还可以使用如下几个选项
// -XX:+PrintGC
// -XX:+PrintGCApplicationStoppedTime
// -XX:+PrintSafepointStatistics
// -XX:+UseCountedLoopSafepoints

public class SafepointTest {
    static double sum = 0;


    public static void foo() {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 0x77777777; i++) {
            sum += Math.sqrt(i);
        }
        long end = System.currentTimeMillis();
        System.out.println("end - start = " + (end - start));
    }


    public static void bar() {
        long start = System.currentTimeMillis();
        for (int i = 0; i < 50_000_000; i++) {
            new Object().hashCode();
        }
        long end = System.currentTimeMillis();
        System.out.println("end - start = " + (end - start));
    }

    public static void main(String[] args) {
      // 两次测试分别运行一个方法
        new Thread(SafepointTest::foo).start();
        //new Thread(SafepointTest::bar).start();
    }
}
```

42、foo中将int换为long，为什么就没有长暂停了？(具有了安全点)

> C2一个诡异的地方：

```
for(int i = start; i < limit; i++){
  ...
}
```

> 1. 对于int循环变量i，如果满足下列条件，C2会将其判定为计数循环(counted loop)，默认不插入安全点。
>   1. 该循环变量的循环出口只有一个，i < limit
>   2. 循环变量的增量为常数(i++), 且limit和循环无关。
> 2. 对于long类型循环变量，C2直接识别为非计数循环，需要插入安全点。

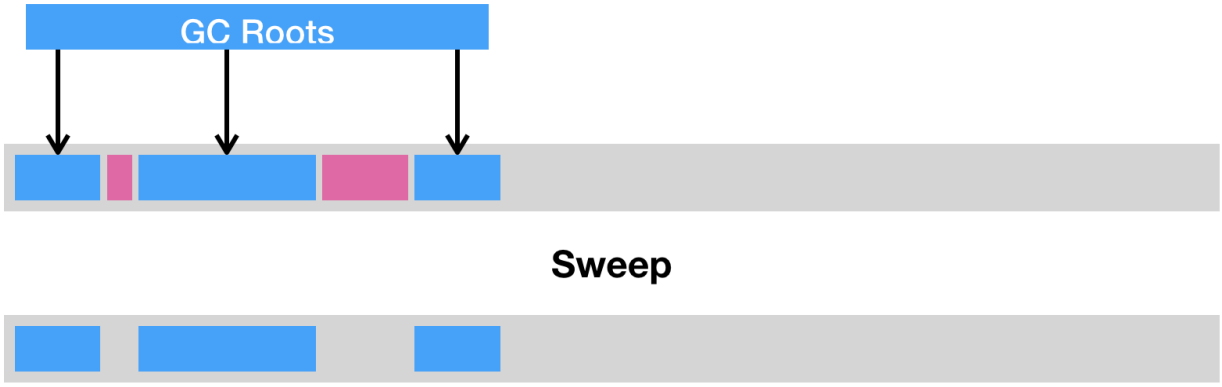
垃圾回收的三种方法(8)

1、主流的基本回收方式有哪几种(3种)?

> 1. 第一种是清除（sweep）
> 2. 第二种是压缩（compact）
> 3. 第三种则是复制（copy）

2、清除的基本原理是什么?

> 1. 把死亡对象所占据的内存标记为空闲内存，并记录在一个空闲列表（free list）之中。
> 2. 当需要新建对象时， 内存管理模块 会从该空闲列表中寻找空闲内存，进行分配。

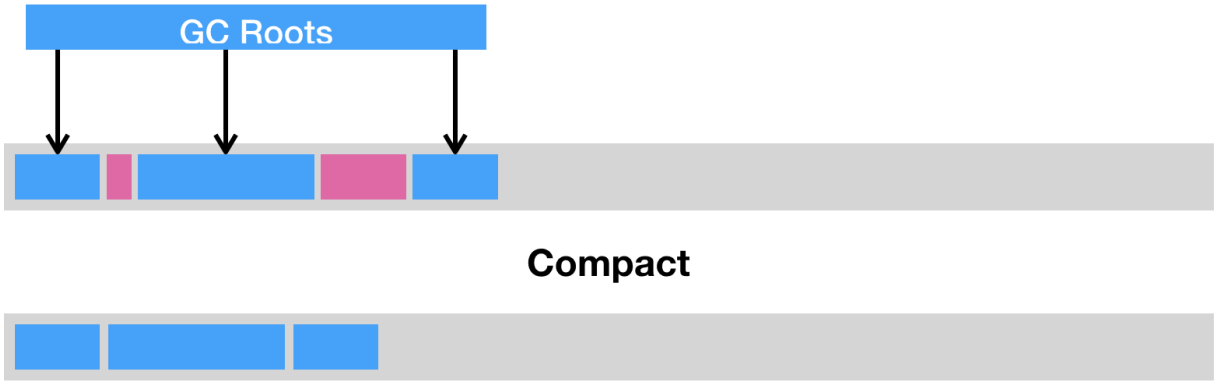


3、清除的优缺点?

> 1. 优点是: 这种回收方式的原理及其简单
> 2. 会造成内存碎片。
>   1. JVM的堆中对象必须是连续分布的，因此可能出现总空闲内存足够，但是无法分配的极端情况。
>   2. 比如：空闲内存不足以分配给一个对象，就会导致无法分配。
> 3. 分配效率较低。
>   1. 如果是一块连续的内存空间，可以通过指针加法（pointer bumping）来做分配。
>   2. 而对于空闲列表，JVM则需要逐个访问列表中的项，来查找大小足够的空闲内存。

4、压缩的基本原理?

> 1. 把存活的对象聚集到内存区域的起始位置，从而留下一段连续的内存空间。
> 2. 这种做法能够解决内存碎片化的问题，但代价是压缩算法的性能开销。

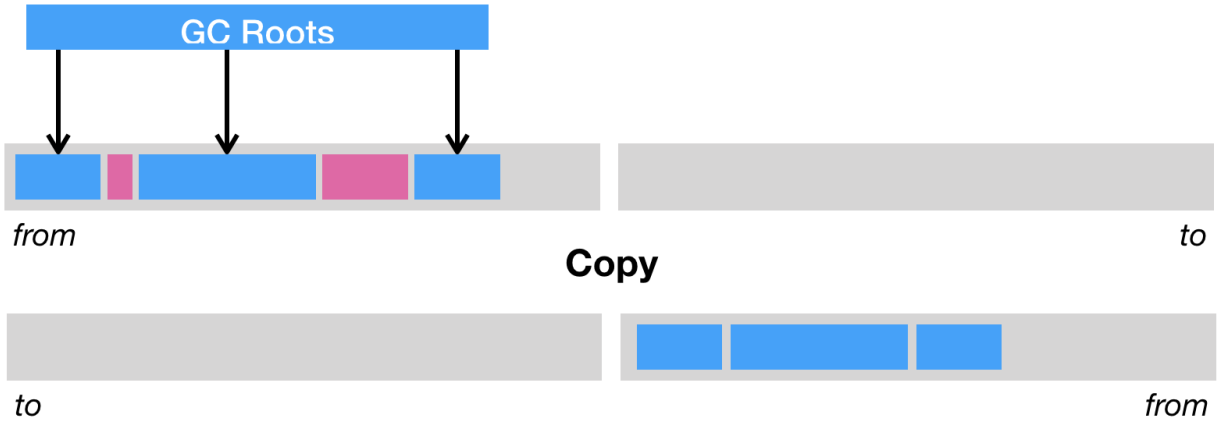


5、压缩的优缺点?

1. 优点：能够解决内存碎片化的问题
2. 缺点: 压缩算法的性能开销

6、复制的基本原理

1. 把内存区域分为两等分，分别用两个指针 from 和 to 来维护
2. 并且只是用 from 指针指向的内存区域来分配内存。
3. 当发生垃圾回收时，便把存活的对象复制到 to 指针指向的内存区域中
4. 并且交换 from 指针和 to 指针的内容



7、复制的优缺点?

1. 优点: 能够解决内存碎片化的问题
2. 缺点: 堆空间的使用效率极其低下

8、复制这种垃圾回收方式，为什么堆空间的使用效率极其低下?

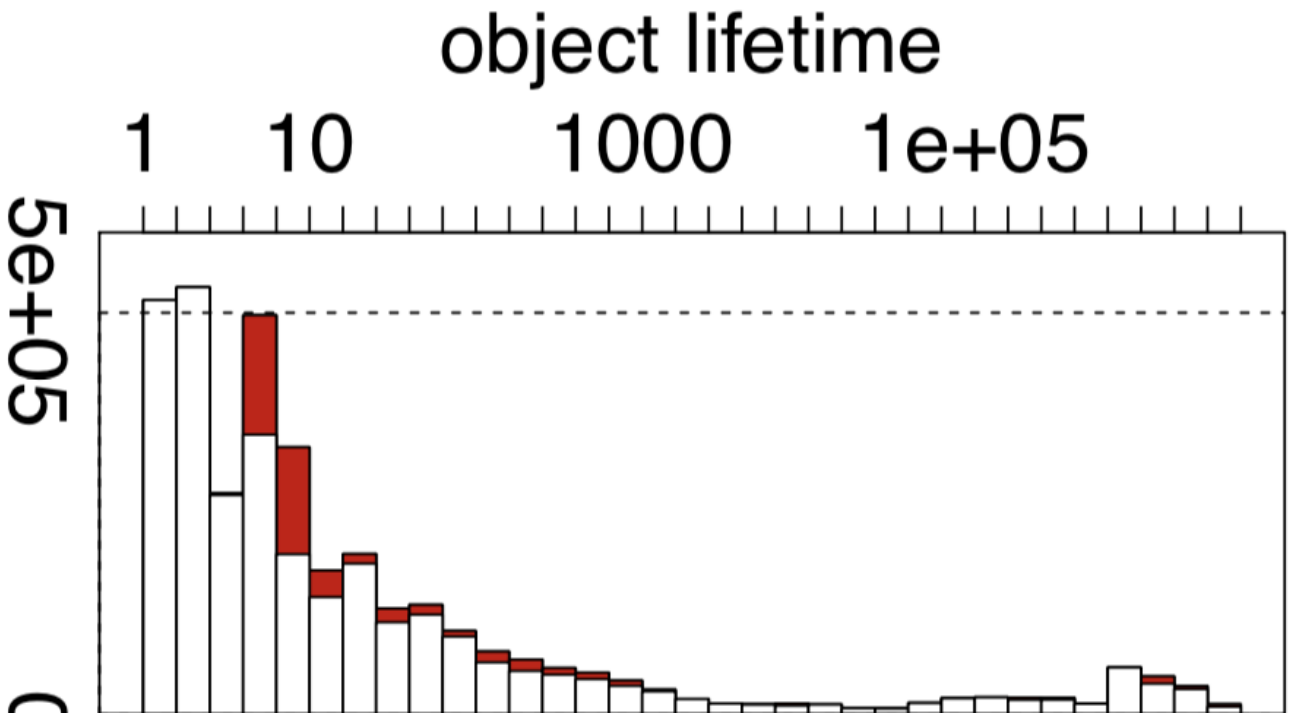
对象的生命周期(31)

1、Java对象生命周期的假设

二八法则的体现:

1. 大部分Java对象会存活一小段时间

2. 小部分Java对象会存活很长一段时间



2、JVM的分代回收思想是基于什么假设产生的?

> 基于Java对象生命周期的假设。

3、JVM的分代回收是指什么?

> 将对空间分为两代:
>
> 1. 新生代
> 2. 老年代

4、新生代是什么?

> 1. 用于存储新建的对象
> 2. 当存活时间足够长时，则移动到老年代。

5、JVM中新生代适合怎样的回收算法?

> 1. 适合频繁采用耗时较短的算法-Minor GC
> 2. 让大部分垃圾在新生代就能被回收

6、JVM中老年代适合采用什么回收算法?

> 1. 回收方式采用全堆扫描

7、JVM老年代出现垃圾回收操作的原因?

> 1. 误判了对象的存活时间(本质应该放置于新生代)
> 2. 堆空间已经耗尽

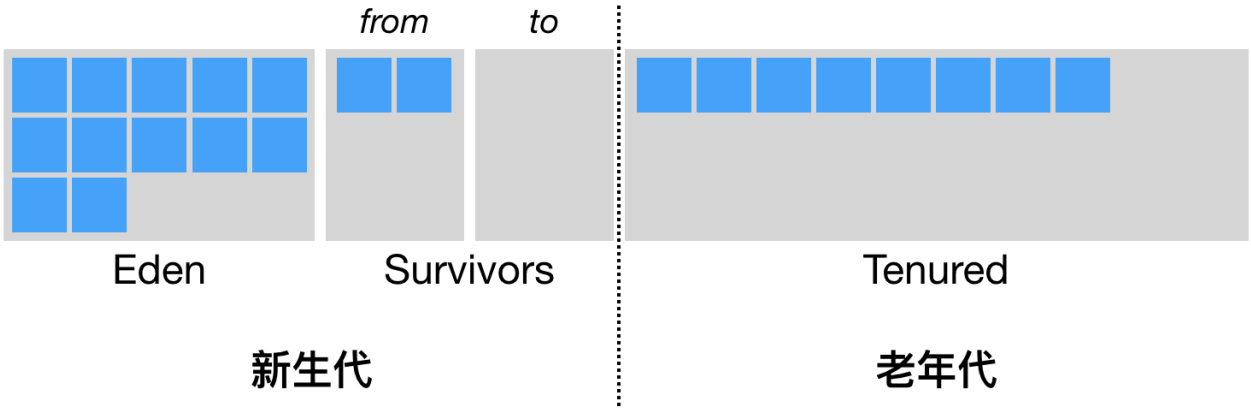
8、全堆扫描的问题和优化?

> 1. 全堆扫描是一种不计消耗时间的方法，性能低
> 2. 当代JVM也在努力发展并发回收，以减少全堆扫描的情况。

新生代

9、JVM的堆如何划分的？新生代分为哪些区域?

> 1. JVM堆划分为新生代和老年代
> 2. 新生代分为：Eden区、两个大小相同的Survivor区



10、JVM采用的动态分配策略是什么?

> 1. 默认情况下，Java 虚拟机采取的是一种动态分配的策略
> 2. 根据 生成对象的速率 和 `Survivor`区的使用情况 动态调整 `Eden` 区 和 `Survivor` 区 的比例。

11、是用什么虚拟机参数可以调整动态分配策略中的比例?

> 1. 对应 Java 虚拟机参数 `XX:+UsePSAdaptiveSurvivorSizePolicy`

12、如果动态分配策略中的比例(Eden/Survivoer)越低，浪费的堆空间将越多?

> 1. 是的
> 2. 两个Survivor区中一个Survivor区会一直为空，如果比率越低，导致Survivor区越高，浪费的堆空间就越多。

13、使用 `new` 指令时是在哪个区域中申请内存?

> 1. 新生代的Eden区域中划分内存，用于存储对象。

14、new指令申请内存时，为什么是在新生代的Eden区域中划分?而不是在新生代的Survivor区域或者老年代的区域中划分?

> 1. 新生代就是用于存储新建的对象
> 2. 采用Eden区域是JVM规范中规定的。

15、堆空间是线程共享的吗?

> 是

16、new指令在堆空间划分空间需要进行同步吗?

> 需要
>
> 1. 因为堆空间是线程共享的
> 2. 不然会出现剐蹭事故-两个线程将对象存放到同一个堆空间中

17、JVM如何解决堆空间中划分空间的同步问题?

> 1. JVM解决办法是为每个线程预先申请一片空间，也就是 `TLAB技术` (Thread Local Allocation Buffer)
> 2. new指令去申请内存时，直接在该内存空间中划分。

TLAB

18、TLAB技术是什么?

> 1. 线程局部分配缓存区-Thread Local Allocation Buffer
> 2. 用于解决在堆空间中划分空间的同步问题
> 3. 每个线程可以向JVM申请一段连续的内存，如2048byte，作为线程私有的TLAB
> 4. 该申请操作需要加锁
> 5. 线程需要重点维护两个指针
>   1. 一个指向TLAB中空余内存的起始地址
>   2. 一个指向TLAB的末尾

19、如果TLAB的空间不够用时该怎么办?

> 线程继续向JVM申请更大的空间

20、TLAB技术下，new指令申请内存的流程?

> 1. new指令直接采用 `指针加法` (bump the pointer)来实现
> 2. 在指向空余内存的起始位置的基础上，将该指针加上 `所请求的字节数`
> 3. 如果加法后空余内存指针的值仍小于或等于指向末尾的指针，则代表分配成功。
> 4. 否则，TLAB已经没有足够的空间来满足本次新建操作。便需要当前线程重新申请新的TLAB。

21、TLAB的虚拟机参数是什么?

> 对应虚拟机参数 `-XX:+UseTLAB` ，默认开启

Minor GC

22、当Eden区的空间耗尽了怎么办?

> 1. 每当这个时候 Java 虚拟机便会触发一次 Minor GC，来收集新生代的垃圾。
> 2. 存活下来的对象，则会被送到Survivor区。

23、Minor GC的内部机制流程

> 1. 新生代共有两个 Survivor 区，分别用from和to来指代。其中to指向的Survivior区是空的。
> 2. Minor GC时， `Eden`区 和 `from`指向的 `Survivor` 区 中的存活对象会被复制到to指向的 Survivor 区中
> 3. 然后交换 from 和 to 指针，以保证下一次Minor GC时，to指向的 Survivor 区还是空的。
> 4. JVM会记录 Survivor区中的对象的复制次数。如果复制的次数为15, 那么该对象将被晋升（promote）至 老年代。
> 5. 另外，如果单个Survivor区已经被占用了50%，那么较高复制次数的对象也会被晋升至老年代。

24、Minor GC采用的什么算法？采用这种算法的好处?

> 1. 标记-复制算法(垃圾回收的第三种方法)
> 2. 复制的数据将非常少： 将Survivor区中的老存活对象晋升到老年代，然后将剩下的存活对象和Eden区的存活对象复制到另一个 Survivor 区中。理 想情况下，Eden 区中的对象基本都死亡了，那么需要复制的数据将非常少，采用该算法效果极好。
> 3. 另外一个好处是不用对整个堆进行垃圾回收。

25、Minor GC算法的缺点

> 1. 老年代的对象可能引用新生代的对象。
> 2. 也就是说，在标记存活对象的时候，我们需要扫描老年代中的对象。
> 3. 如果某老年代的对象拥有对新生代对象的引用，那么这个引用也会被作为 GC Roots。

26、Survivor区中对象能晋级到老年代的复制次数用什么虚拟机参数可以设置(默认15)?

> `-XX:+MaxTenuringThreshold`

27、Survivor区达到一定占用比时，那么较高复制次数的对象也会被晋升至老年代，该占用比通过什么虚拟机参数进行设置(默认50%)?

> `XX:TargetSurvivorRatio`

28、Minor GC算法中出现全堆扫描的问题?

> 1. 是的，老年代的对象可能引用新生代的对象。
> 2. 我们需要扫描老年代中所有对象。

29、什么是全堆扫描?

> 扫描堆中所有对象

30、为什么JVM分代回收时新生代对象进入老年代，年龄为什么默认是15而不是其他的?

1. HotSpot会在对象头中的标记字段里记录年龄
2. 分配到的空间只有4位，最多只能记录到15

31、JVM分代回收时，是根据对象复制次数是否达到15来决定对象是否进入老年代，这里的年龄可以通过参数设置达到15以上吗?

1. 不可以
2. 通过参数可以设置年龄<=15，但是不可以超过15，因为对象头中标记字段里记录年龄的部分只有4位

卡表(10)

1、如何解决Minor GC算法中老年代对象引用新生代对象导致的扫描老年代中所有对象的问题(全堆扫描)?

1. HotSpot中采用一种 卡表-Card Table 的技术

2、卡表技术是什么?

1. Card Table
2. HotSpot中一种用于解决Minor GC的全堆扫描的问题
3. 该技术将整个堆划分为一个个大小为 512 字节的卡，并且维护一个卡表，用来存储每张卡的一个标识位。
4. 这个标识位代表对应的卡是否可能存有指向新生代对象的引用。
5. 如果可能存在，那么我们就认为这张卡是脏的。
6. 在进行 Minor GC 的时候，我们便可以不用扫描整个老年代，而是在卡表中寻找脏卡，并将脏 卡中的对象加入到 Minor GC 的 GC Roots 里。
7. 当完成所有脏卡的扫描之后，Java 虚拟机便会 将所有脏卡的标识位清零。

3、Minor GC时，为什么可以确保脏卡中必定包含指向 新生代对象的引用?

1. Minor GC时会进行存活对象的复制，而复制需要更新指向该对象的引用。
2. 因此，在更新引用时，会设置引用所在的卡的标识位。
3. 这样就可以确保脏卡中必定包含指向新生代对象的引用。

4、Minor GC之前，为什么不能确保脏卡中包含指向新生代对象的引用?

1. 其原因和如何设置卡 的标识位有关。
2. 首先，如果想要保证每个可能有指向新生代对象引用的卡都被标记为脏卡，那么JVM需要截获每个引用型实例变量的写操作，并作出对应的写标识位操作。
3. 这个操作在解释执行器中比较容易实现。
4. 但是在即时编译器生成的机器码中，则需要插入额外的 逻辑。这也就是所谓的写屏障（write barrier，注意不要和 volatile 字段的写屏障混淆）。

5. 写屏障需要尽可能地保持简洁。这是因为我们并不希望在每条引用型实例变量的写指令后跟着一 大串注入的指令。
6. 因此，写屏障并不会判断更新后的引用是否指向新生代中的对象，而是宁可错杀，不可放过，一 律当成可能指向新生代对象的引用。

5、写屏障的特点

1. 写屏障-write barrier，和volatile字段的写屏障不同。
2. 写屏障需要尽可能地保持简洁。避免在每条引用型实例变量的写指令后跟着一大串注入的指令。
3. 写屏障并不会判断更新后的引用是否指向新生代中的对象，而是宁可错杀，不可放过，一律当成可能指向新生代对象的引用。

6、volatile字段的写屏障是什么?

7、写屏障的伪代码

1. 这里右移 9 位相当于除以 512
2. JVM便是通过这种方式来从地址映射到卡表中的索引的。
3. 最终，这段代码会被编译成一条移位指令和 一条存储指令。

```
CARD_TABLE [this address >> 9] = DIRTY;
```

8、写屏障的优缺点

1. 虽然写屏障不可避免地带来一些开销
2. 但是能够加大 Minor GC 的吞吐率- 应用运行时间 /(应用运行时间 + 垃圾回收时间) 。
3. 但在高并发环境下，写屏障又带来了虚共享（false sharing）问题。

9、Minor GC的吞吐量是什么?

```
应用运行时间 /(应用运行时间 + 垃圾回收时间)
```

10、写屏障的虚共享问题

1. 虚共享-(false sharing)
2. 对象内存布局中的虚共享问题，是几个volatile字段出现在同一缓存行里造成的虚共享。
3. 写屏障的虚共享是卡表中不同卡的标识位之间的虚共享问题。
4. 在 HotSpot 中，卡表是通过 byte 数组来实现的。对于一个 64 字节的缓存行来说，如果用它来 加载部分卡表，那么它将对应 64 张卡，也就是 32KB 的内存。
5. 如果同时有两个 Java 线程，在这 32KB 内存中进行引用更新操作，那么也将造成存储卡表的同 一部分的缓存行的写回、无效化或者同步操作，因而间接影响程序性能。
6. 为此，HotSpot 引入了一个新的参数 -XX:+UseCondCardMark，来尽量减少写卡表的操作。其伪代码如下所示：

```
if (CARD_TABLE [this address >> 9] != DIRTY)
  CARD_TABLE [this address >> 9] = DIRTY;
```

具体垃圾回收器(10)

1、新生代的垃圾回收器共有哪三个?

> 1. Serial-单线程的
> 2. Parallel New-多线程版本
> 3. Parallel Scavenge-和Parallel New 类似，但更加注重吞吐率。

2、新生代的垃圾回收器采用什么算法?

> 都是标记 - 复制算法。

3、Parallel Scavenge 不能与CMS一起使用？为什么?

4、老年代的垃圾回收器共有哪几种?

> 1. Serial Old-单线程，标记 - 压缩算法(没有碎片、压缩算法代价高)
> 2. Parallel Old-多线程版本，标记 - 压缩算法
> 3. CMS-多线程，采用标记 - 清除算法(会产生碎片)

5、垃圾回收三种算法在垃圾回收器中的应用?

> 1. 标记 - 复制算法: Serial、Parallel New、Parallel Scavenge
> 2. 标记 - 压缩算法：Serial Old、Parallel Old
> 3. 标记 - 清除算法：CMS

6、CMS垃圾回收器的特点?

> 1. 采用的是标记 - 清除算法
> 2. 是并发的
> 3. 除了少数几个操作需要STW，可以在应用程序运行过程中进行垃圾回收。在并发收集失败的情况下，JVM会使用其他两个压缩型垃圾回收器进行一次垃圾回收。

7、老年代在Java8以及之前是如何进行垃圾回收的?

> 1. 先使用CMS并发进行垃圾回收。除了少数几个操作需要STW，可以在应用程序运行过程中进行垃圾回收。
> 2. 在并发收集失败的情况下，JVM会采用Serial Old和Parallel Old进行垃圾回收

8、CMS 在 Java 9 中已被废弃? G1是什么?

> 1. 由于 G1 的出现，CMS 在 Java 9 中已被废弃 [3]。
> 2. G1（Garbage First）是一个横跨新生代和老年代的垃圾回收器。

9、G1的特点?

> 1. 采用的是标记 - 压缩算法，
> 2. 和CMS 一样都能够在应用程序运行过程中并 发地进行垃圾回收。
> 3. G1已经打乱了常规的堆结构，直接将堆分成极其多个区域。每个区域都可以充当 Eden 区、Survivor 区或者老 年代中的一个。
> 4. G1 能够针对每个细分的区域来进行垃圾回收。
> 5. G1 会优先回收死 亡对象较多的区域。(Garbage First)

10、Java11中引入的ZGC是什么?

> 1. Java 11 引入了 ZGC
> 2. 宣称暂停时间不超过 10ms。
> 3. 参考资料中有 R 大的文章。

知识扩展(16)

1、Java是跨平台的语言，可是又能调用本地方法，而本地方法用C实现，不就破坏了其平台无关性?

> 1. 一般需要C语言实现的代码，都是Java层面无法实现的内容。
> 2. 因此没有办法，只能牺牲平台无关性。

2、缓存行是什么？虚共享是什么?

JVM的数据分布

3、程序计数器(Program Conuter Register)是什么?

> 1. 是一块较小的内存空间
> 2. 是当前线程执行字节码的行号指示器
> 3. 字节码解释工作器就是通过改变这个计数器的值来选取下一条需要执行的指令
> 4. 它是线程私有的内存
> 5. 也是唯一一个没有OOM异常的区域。

4、Java虚拟机栈区(Java Virtual Machine Stacks)是什么?

> 1. 通常所说的栈区
> 2. 它描述的是Java方法执行的内存模型
> 3. 每个方法被执行的时候都创建一个栈帧(Stack Frame)---用于存储局部变量表、操作数栈、动态链接、方法出口等。
> 4. 每个方法被调用到完成，相当于一个栈帧在虚拟机栈中从入栈到出栈的过程。
> 5. 此区域也是线程私有的内存，可能抛出两种异常：
>   1. 如果线程请求的栈深度大于虚拟机允许的深度将抛出StackOverflowError；

> 2. 如果虚拟机栈可以动态的扩展，扩展到无法动态的申请到足够的内存时会抛出OOM异常。

5、本地方法栈(Native Method Stacks)是什么?

> 1. 本地方法栈与虚拟机栈发挥的作用非常相似
> 2. 区别就是虚拟机栈为虚拟机执行Java方法
> 3. 本地方法栈则是为虚拟机使用到的Native方法服务。

6、堆区(Heap)是什么?

> 1. 所有对象实例和数组都在堆区上分配? 不准确，也可以在堆外分配对象
> 2. 堆区是GC主要管理的区域
> 3. 堆区还可以细分为新生代、老年代，新生代还分为一个Eden区和两个Survivor区
> 4. 此块内存为所有线程共享区域
> 5. 当堆中没有足够内存完成实例分配时会抛出OOM异常。

7、方法区(Method Area)是什么?

> 1. 方法区也是所有线程共享区
> 2. 用于存储已被虚拟机加载的:
>   1. 类信息
>   2. 常量
>   3. 静态变量
>   4. 即时编译后的代码等数据。
> 3. GC在这个区域很少出现，主要是对常量池的回收和类型的卸载
> 4. 回收的内存比较少, 所以也有称这个区域为永久代(Permanent Generation)。
> 5. 当方法区无法满足内存分配时抛出OOM异常。

8、运行时常量池(Runtime Constant Pool)是什么?

> 1. 运行时常量池是方法区的一部分
> 2. 用于存放编译期生成的各种字面量和符号引用。

GC Roots

9、为什么一些特定对象可以作为GC roots?

> 因为这些东西被认为是在被使用。根据JVM规范将他们作为GC Roots 。

10、到底GC Roots是什么?

> GC Roots是一个统称，是所有可以用作“可达性算法”中的根。

11、GC Roots存放在哪里?

1. GC Roots 本身是没有所谓的存储位置
2. GC Roots 都是加载到JVM中的一些普通对象，只不过被认为是GC Roots 。

12、GC Roots是引用还是对象?

1. 是对象，但是该对象一般都有个引用。
2. 对于Java语言(非字节码)来说单独的引用(没有指向对象的引用)没有意义。

13、GC Roots是放在堆里的还是方法栈还是哪个地方?

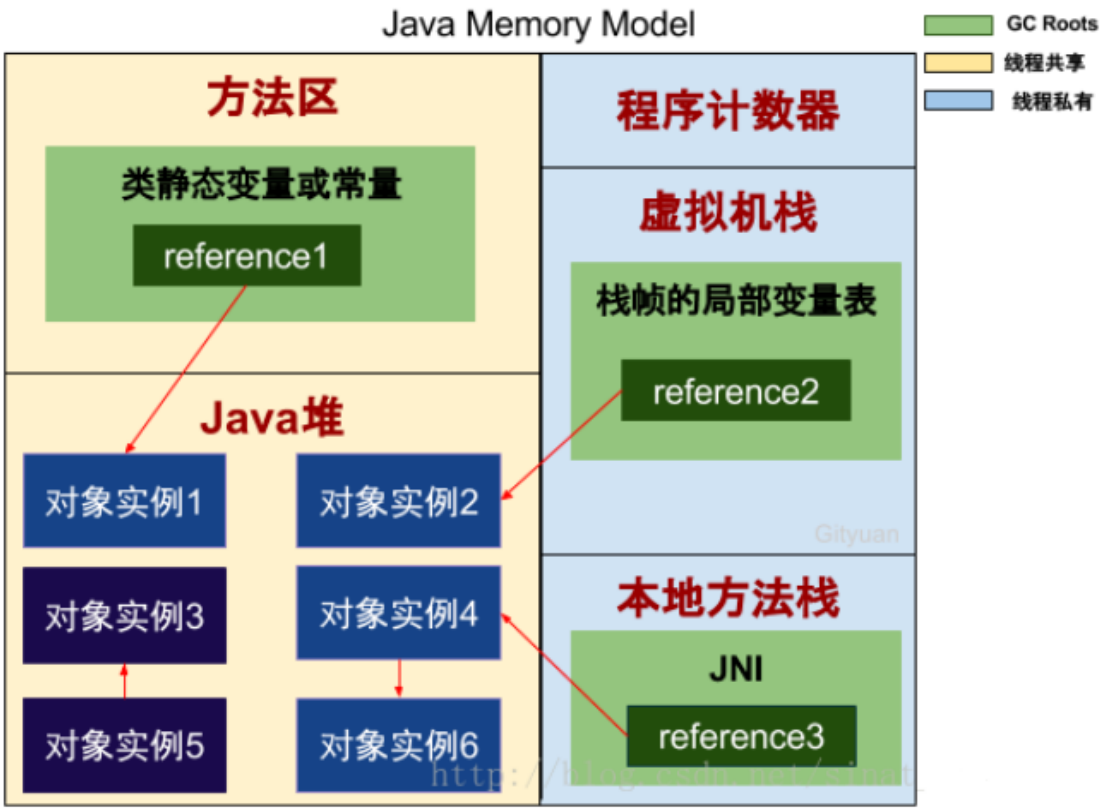
1. 引用一般是存储在运行时常量池(方法区)、方法区、栈中。但是其引用的对象是存储在堆中的。

14、虚拟机会回收GC Roots吗?

1. 不会但不保证绝对不会，因为JVM有几十种，无法保证以后会不会加入回收GC Roots 的机制
2. 但就HotSpot而言，不会回收GC Roots。

15、如何根据GC Roots进行可达性分析?

1. 可达性分析就是去标记能和 `GC Roots` 构成连通图的对象
2. 从根节点开始，搜索所走过的路径称为引用链(Reference Chain)，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。
3. 如图: 对象实例3、对象实例5就是 垃圾对象



16、对象的成员变量存储在哪里？如何进行可达性分析?

> 1. 对象的成员变量不属于方法的局部变量，因此不能存在 栈 中
> 2. 对象的成员变量追随对象一起存储在 堆 中
> 3. 可达性分析时，对象是可达的，那么成员变量所引用的对象，也就是可达的。

问题汇总

1. 什么是垃圾回收?
2. 在JVM中垃圾回收中的垃圾所占的内存空间具体是指什么?
3. JVM垃圾回收的关键?如何判断一个对象是存活还是死亡的?
4. 引用计数法是什么?
5. 引用计数法的实现思路?
6. 引用计数法的缺点?
7. 什么是引用更新操作?
8. 为什么无法处理循环引用的对象?
9. 垃圾回收中的标记过程是干什么的?
10. GC roots是什么?
11. GC roots有哪些?
12. 可达性分析如何解决的循环引用问题?
13. 可达性分析算法在实践应用中的问题?
14. 可达性分析在多线程中遭遇的误报具体是什么?
15. 可达性分析在多线程中遭遇的漏报具体是什么?
16. 如何解决可达性分析算法在多线程中的问题?
17. STW有什么问题?
18. 垃圾回收中的GC pause是什么?
19. JVM中的STW是如何实现的?
20. 安全点是什么?
21. 安全点的本质目的是为了让其他线程停下?
22. 怎样才是一个稳定的执行状态(JVM堆栈不会发生改变)?
23. Java线程具有哪几种执行状态?(4种)
24. 执行JNI本地代码如何进入安全点?
25. 安全点检测是干什么的?
26. 线程阻塞的安全点检测?
27. 执行JNI本地代码、解释执行字节码、执行即时编译器生成的机器码这三种运行状态需要虚拟机保证在可预见的时间内进入安全点，不然会怎么样?
28. 解释执行字节码时，需要如何进入安全点?
29. 执行即时编译器生成的机器码，需要如何进行安全点检测?
30. 非计数循环的循环回边处是什么?
31. 为什么不在每一条机器码或者每一个机器码基本块处插入安全点检测呢?
32. HotSpot虚拟机如何简化机器码中的安全点检测?
33. HotSpot如何解决即时编译器生成的机器码打乱了原本栈桢上的对象分布状况的这个问题?
34. 不同的即时编译器插入安全点检测的位置也可能不同?有什么不同?

35. 安全点除了垃圾回收外还有哪些场景可以利用该机制?
36. STW的机制非常不友好，有哪些解决之道？原理是什么?
37. GC时有时候会突然出现较长的时间消耗，是为什么?
38. Full GC有卡顿，对性能很不利，该如何避免?
39. 多线程为什么会导致误报和漏报?
40. 有安全点和无安全点的性能差距有多少?
41. foo中将int换为long，为什么就没有长暂停了？(具有了安全点)
42. 主流的基本回收方式有哪几种(3种)?
43. 清除的基本原理是什么?
44. 清除的优缺点?
45. 压缩的基本原理?
46. 压缩的优缺点?
47. 复制的基本原理
48. 复制的优缺点?
49. 复制这种垃圾回收方式，为什么堆空间的使用效率极其低下?
50. Java对象生命周期的假设
51. JVM的分代回收思想是基于什么假设产生的?
52. JVM的分代回收是指什么?
53. 新生代是什么?
54. JVM中新生代适合怎样的回收算法?
55. JVM中老年代适合采用什么回收算法?
56. JVM老年代出现垃圾回收操作的原因?
57. 全堆扫描的问题和优化?
58. JVM的堆如何划分的？新生代分为哪些区域?
59. JVM采用的动态分配策略是什么?
60. 是用什么虚拟机参数可以调整动态分配策略中的比例?
61. 如果动态分配策略中的比例(Eden/Survivoer)越低，浪费的堆空间将越多?
62. 使用 `new` 指令时是在哪个区域中申请内存?
63. new指令申请内存时，为什么是在新生代的Eden区域中划分?而不是在新生代的Survivor区域或者老年代的区域中划分?
64. 堆空间是线程共享的吗?
65. new指令在堆空间划分空间需要进行同步吗?
66. JVM如何解决堆空间中划分空间的同步问题?
67. TLAB技术是什么?
68. 如果TLAB的空间不够用时该怎么办?
69. TLAB技术下，new指令申请内存的流程?
70. TLAB的虚拟机参数是什么?
71. 当Eden区的空间耗尽了怎么办?
72. Minor GC的内部机制流程
73. Minor GC采用的什么算法？采用这种算法的好处?
74. Minor GC算法的缺点

75. Survivor区中对象能晋级到老年代的复制次数用什么虚拟机参数可以设置(默认15)?
76. Survivor区达到一定占用比时，那么较高复制次数的对象也会被晋升至老年代，该占用比通过什么虚拟机参数进行设置(默认50%)?
77. Minor GC算法中出现全堆扫描的问题?
78. 什么是全堆扫描?
79. 为什么JVM分代回收时新生代对象进入老年代，年龄为什么默认是15而不是其他的?
80. JVM分代回收时，是根据对象复制次数是否达到15来决定对象是否进入老年代，这里的年龄可以通过参数设置达到15以上吗?
81. 如何解决Minor GC算法中老年代对象引用新生代对象导致的扫描老年代中所有对象的问题(全堆扫描)?
82. 卡表技术是什么?
83. Minor GC时，为什么可以确保脏卡中必定包含指向 新生代对象的引用?
84. Minor GC之前，为什么不能确保脏卡中包含指向新生代对象的引用?
85. 写屏障的特点
86. volatile字段的写屏障是什么?
87. 写屏障的伪代码
88. 写屏障的优缺点
89. Minor GC的吞吐量是什么?
90. 写屏障的虚共享问题
91. 新生代的垃圾回收器共有哪三个?
92. 新生代的垃圾回收器采用什么算法?
93. Parallel Scavenge 不能与CMS一起使用？为什么?
94. 老年代的垃圾回收器共有哪几种?
95. 垃圾回收三种算法在垃圾回收器中的应用?
96. CMS垃圾回收器的特点?
97. 老年代在Java8以及之前是如何进行垃圾回收的?
98. CMS 在 Java 9 中已被废弃? G1是什么?
99. G1的特点?
100. Java11中引入的ZGC是什么?
101. Java是跨平台的语言，可是又能调用本地方法，而本地方法用C实现，不就破坏了其平台无关性?
102. 缓存行是什么？虚共享是什么?
103. 程序计数器(Program Conuter Register)是什么?
104. Java虚拟机栈区(Java Virtual Machine Stacks)是什么?
105. 本地方法栈(Native Method Stacks)是什么?
106. 堆区(Heap)是什么?
107. 方法区(Method Area)是什么?
108. 运行时常量池(Runtime Constant Pool)是什么?
109. 为什么一些特定对象可以作为GC roots?
110. 到底GC Roots是什么?
111. GC Roots存放在哪里?

112. GC Roots是引用还是对象?
113. GC Roots是放在堆里的还是方法栈还是哪个地方?
114. 虚拟机会回收GC Roots吗?
115. 如何根据GC Roots进行可达性分析?
116. 对象的成员变量存储在哪里？如何进行可达性分析?

参考资料

1. 极客时间-JVM垃圾回收(下)
2. 垃圾收集器GC中parallel scavenge收集器为什么不能CMS配合使用?
3. ZGC 原理是什么，它为什么能做到低延时?
4. Java GC 内存回收机制详解（二）GC Roots 和 可达链
5. Java GC如何判断对象是否为垃圾