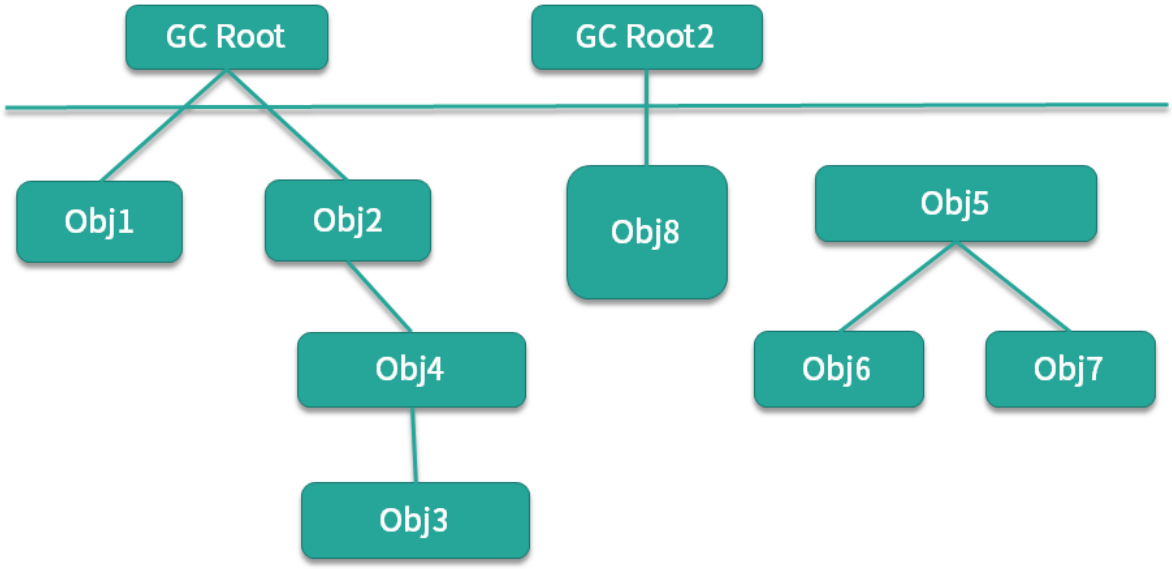
方法区会导致oom吗
年轻代的阈值默认

得心应手应对 OOM 的疑难杂症

什么是GC Roots：

垃圾回收就是围绕着 GC Roots 去做的，它也是很多内存泄露的根源，因为其他引用根本没有这样的权利。

如图所示，Obj5、Obj6、Obj7，由于不能和 GC Root 产生关联，发生 GC 时，就会被垃圾回收。



GC Roots 有哪些：

GC Roots 是一组必须活跃的引用。用通俗的话来说，就是程序接下来通过直接引用或者间接引用，能够访问到的**潜在被使用的对象**，对象是不能作为 GC Roots 的。

GCroot都有哪些?

> GC管理的主要区域是Java堆，一般情况下只针对堆进行垃圾回收。方法区、栈和本地方法区不被GC所管理,因而选择这些区域内的对象作为GC roots,被GC roots引用的对象不被GC回收。

1.虚拟机栈：栈帧中的本地变量表引用的对象(活动线程相关的各种引用)

2.native方法**引用**的对象

3.方法区中的静态变量和常量**引用**的对象（final 的常量值）

内存泄露的场景?

在java中内存泄漏的对象有两个特点：

- 该对象是可达的

- 该对象是无用的，即程序不再使用该对象

场景：

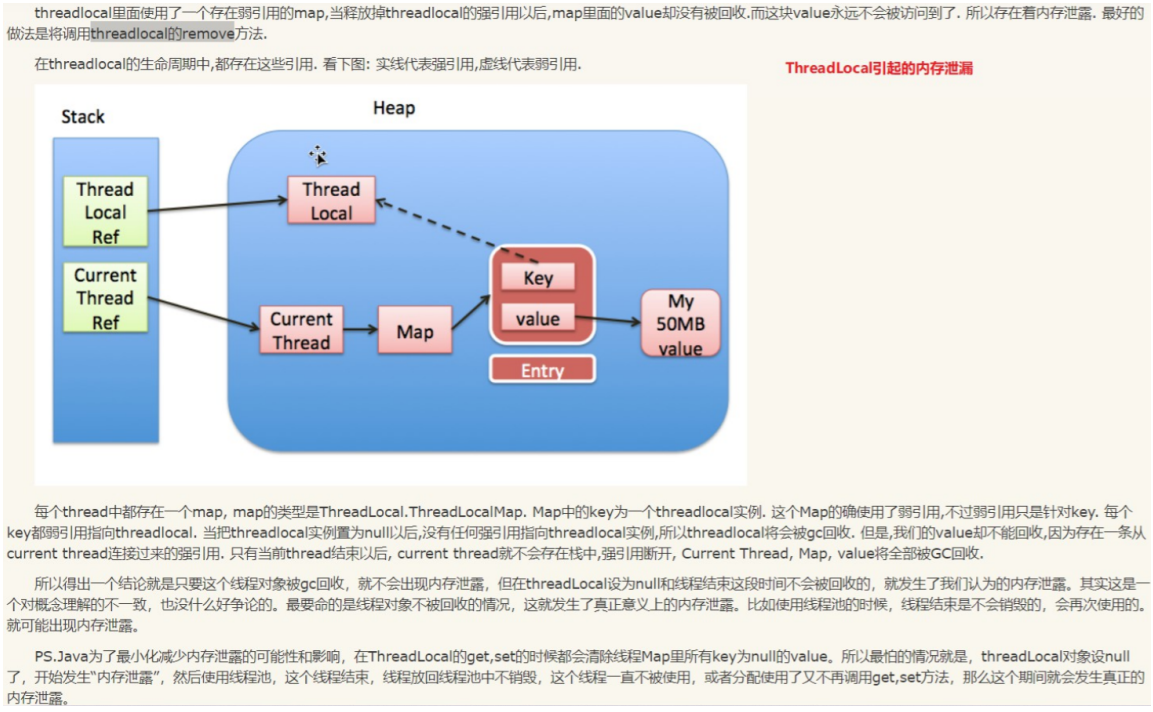
1. 创建大量无用对象：比如说当我们需要进行大量的字符串拼接时，使用的时 String构造，而不是Stringbulider。
2. 静态集合类的使用：像HashMap、Vector、List等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，所有的对象Object也不能被释放。大量的静态集合类容易造成内存泄漏。

   解决办法：最好不使用静态的集合类，如果使用的话，在不需要容器时要将其赋值为null。
3. 监听器：在java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

```
games.addValueEventListener(new ValueEventListener() {}) //添加监听器
games.removeEventListener(this); //删除监听器
```

4. 各种连接：
   比如**数据库连接（dataSourse.getConnection()）、网络连接(socket)和io连接、输入输出流**没有关闭，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。
5. ThreadLocal没有remove：

   ThreadLocal用完一定要remove，否则可能会造成内存泄漏。



6. 内部类和外部模块等的引用：
   内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如：
   public void registerMsg(Object b);
   这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B 是否提供相应的操作去除引用。

7. 单例模式

不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露，考虑下面的例子：

```
class A {
    public A() {
        B.getInstance().setA(this);
    }
....
}
//B类采用单例模式
class B {
    private A a;
    private static B instance = new B();
    public B() {
    }
    public static B getInstance() {
        return instance;
    }
    public void setA(A a) {
        this.a = a;
    }
//getter...
}
```

显然B采用singleton模式，它持有一个A对象的引用，而这个A类的对象将不能被回收。想象下如果A是个比较复杂的对象或者集合类型会发生什么情况。

解决办法：单例对象中持有的其他对象使用弱引用，弱引用对象在GC线程工作时，其占用的内存会被回收掉，如下示例：

```
public class SingleTon1 {
    private static final SingleTon1 mInstance = null;
    private WeakReference<Context> mContext;
    private SingleTon1(WeakReference<Context> context) {
    mContext = context;
    }
    public static SingleTon1 getInstance(WeakReference<Context> context) {
        if (mInstance == null) {
            synchronized (SingleTon1.class) {
                if (mInstance == null) {
                    mInstance = new SingleTon1(context);
                }
            }
        }
        return mInstance;
    }
}

public class MyActivity extents Activity {
    public void onCreate (Bundle savedInstanceState){
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
       SingleTon1 singleTon1 = SingleTon1.getInstance(new
WeakReference<Context>(this));
   }

}
```

8. 使用非静态内部类：

   非静态内部类对象的构建依赖于其外部类，内部类对象会持有外部类对象的this引用，即时外部类对象不再被使用了，其占用的内存可能不会被GC回收，因为内部类的生命周期可能比外部类的生命周期要长，从而造成外部类对象不能被及时回收。解决办法是尽量使用静态内部类，静态内部类只是形式上在外部类的里面，静态内部类不会持有外部类的引用，可以把静态内部类理解成是一个独立的类，和外部类没什么关系。

   ```
   1 引申：为什么非静态内部类对象会持有外部类对象的this引用？
   ```

   非静态内部类虽然和外部类写在同一个文件中， 但是编译完成后， 还是生成各自的class文件，通过如下三个步骤，内部类对象通过this访问外部类对象的成员。

   1） 编译器自动为内部类添加一个成员变量， 这个成员变量的类型和外部类的类型相同， 这个成员变量就是指向外部类对象(this)的引用；

   2） 编译器自动为内部类的构造方法添加一个参数， 参数的类型是外部类的类型， 在构造方法内部使用这个参数为内部类中添加的成员变量赋值；

   3）在调用内部类的构造函数初始化内部类对象时，会默认传入外部类的引用。

9. hashCode，equals重写错误：

   为何要重写hashCode?

   Java中的集合（Collection）有两类，一类是List，再有一类是Set。
   前者集合内的元素是有序的，元素可以重复；后者元素无序，但元素不可重复。 那么我们怎么判断两个元素是否重复呢？ 这就是Object.equals方法了。

   通常想查找一个集合中是否包含某个对象，就是逐一取出每个元素与要查找的元素进行比较，当发现某个元素与要查找的对象进行equals方法比较的结果相等时，则停止继续查找并返回肯定的信息，否则返回否定的信息，如果一个集合中有很多元素譬如成千上万的元素，并且没有包含要查找的对象时，则意味着你的程序需要从该集合中取出成千上万个元素进行逐一比较才能得到结论，于是，有人就发明了一种哈希算法来提高从集合中查找元素的效率，这种方式将集合分成若干个存储区域，每个对象可以计算出一个哈希码，可以将哈希码分组，每组分别对应某个存储区域，根据一个对象的哈希码就可以确定该对象应该存储的那个区域。

   hashCode方法可以这样理解：它返回的就是根据对象的内存地址换算出的一个值。这样一来，当集合要添加新的元素时，先调用这个元素的hashCode方法，就一下子能定位到它应该放置的物理位置上。如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。

   为何会泄露?

   当一个对象被存储进hashset集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否则，对象修改后的哈希值与最初存储进hashset集合时的哈希值就不同了，这种情况下，即使在contains方法使用该对象的当前引用作为的参数去hashset集合中检索对象，也将返回找不到对象的结果，这也会导致无法从hashset集合中单独删除当前对象，从而造成内存泄露。

   简单来说，如果重写错误的hashcode，导致放进去的对象无法被找到，无法执行删除该元素，积压多了，内存就不断增长

引用级别：

GC ROOT的引用也分级别，不同级别回收的权限不一样。

1. **强引用 Strong references：**

Java程序中最常见的引用方式，程序创建一个对象，并赋给一个引用变量，这个引用变量就是强引用。当一个对象有一个或多个强引用时，Java垃圾收集器不会回收它。

2. **软引用 Soft references：**

   内存空间充足就不会回收，内存空间不足就会回收。**软引用通常用于对内存敏感的项目中，在此类项目中，软引用是强引用很好的替代者。**

3. **弱引用 Weak references：**

   弱引用与软引用类似，区别在于弱引用引用的对象生存期更短，不管内存是否足够，总是会被回收。

4. **虚引用 Phantom References：**

典型 OOM 场景：

OOM 的全称是 Out Of Memory

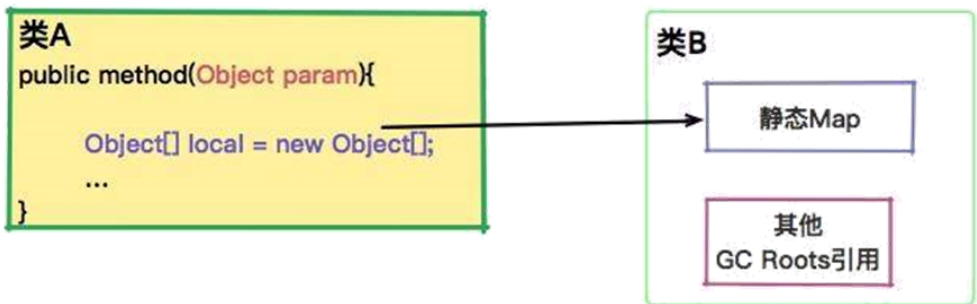
可以看到除了程序计数器，其他区域都有OOM溢出的可能。但是最常见的还是发生在堆上。

区域	是否线程私有	是否会发生OOM
程序计数器	是	否
虚拟机栈	是	是
本地方法栈	是	是
方法区	否	是
直接内存	否	是
堆	否	是

所以 OOM 到底是什么引起的呢？有几个原因：

- 内存的容量太小了，需要扩容，或者需要调整堆的空间。
- 错误的引用方式，发生了内存泄漏。没有及时的切断与 GC Roots 的关系。比如线程池里的线程，在复用的情况下忘记清理 ThreadLocal 的内容。
- 接口没有进行范围校验，外部传参超出范围。比如数据库查询时的每页条数等。
- 对堆外内存无限制的使用。这种情况一旦发生更加严重，会造成操作系统内存耗尽。

典型的内存泄漏场景，原因在于对象没有及时的释放自己的引用。比如一个局部变量，被外部的静态集合引用。



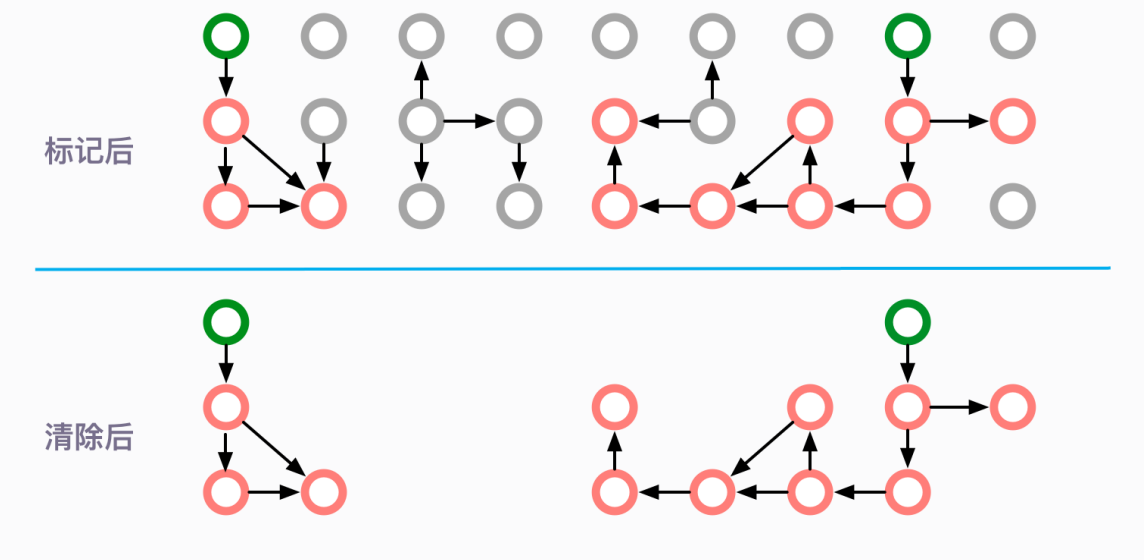
你在平常写代码时，一定要注意这种情况，千万不要为了方便把对象到处引用。即使引用了，也要在合适时机进行手动清理。关于这部分的问题根源排查，我们将在实践课程中详细介绍。

垃圾回收讲解：

垃圾回收的思想是先找到活跃的对象，不活跃的就是要回收的对象，**所以垃圾回收只与活跃的对象有关，和堆的大小无关。**

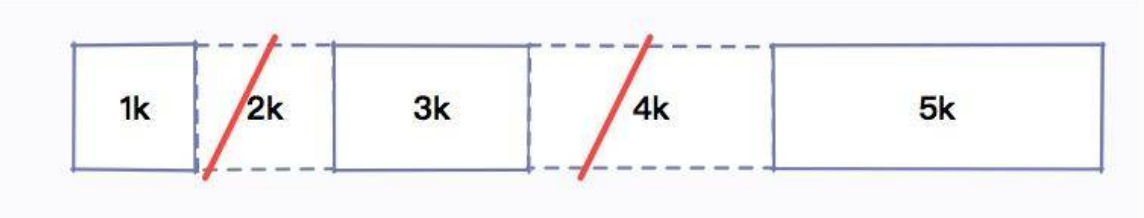
几种常见的内存回收算法：

1. **标记-清除：**



如上图，灰色的就是回收的对象，但是这种简单的清除方式，有一个明显的弊端，那就是碎片问题。比如我申请了 1k、2k、3k、4k、5k 的内存。

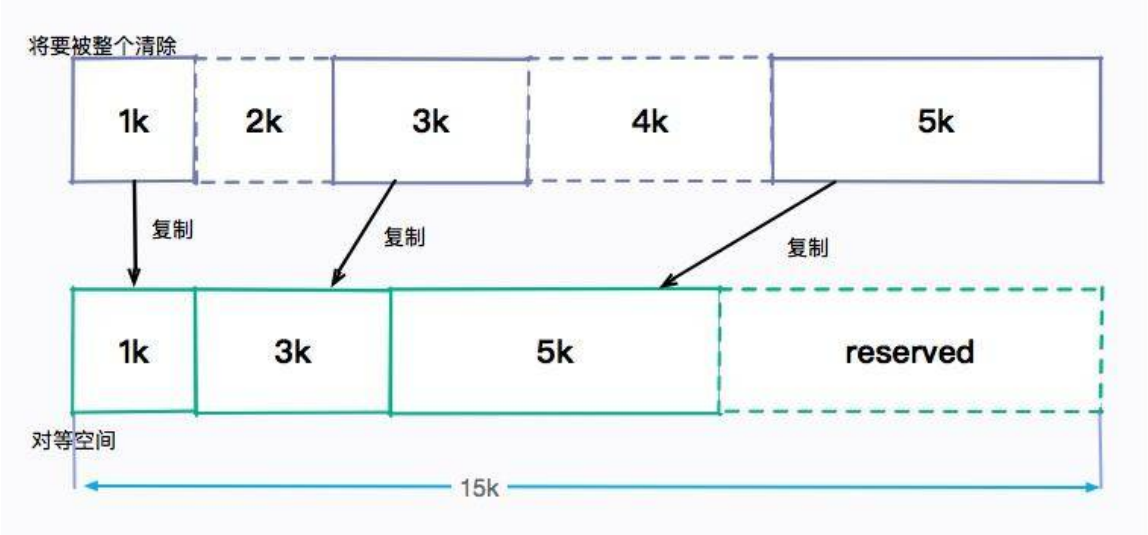
由于某种原因，2k 和 4k 的内存，我不再使用，就需要交给垃圾回收器回收。



这个时候，我应该有足足 6k 的空闲空间。接下来，我打算申请另外一个 5k 的空间，结果系统告诉我内存不足了。系统运行时间越长，这种碎片就越多。

2. **复制：**

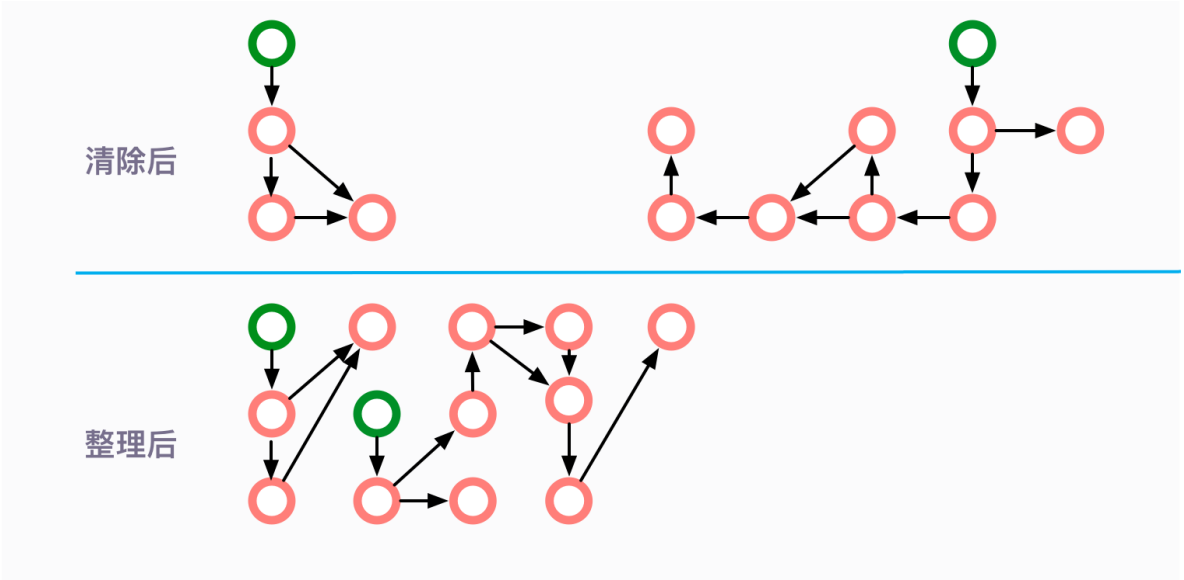
复制的方法就是为了改进碎片问题给出的方案，将碎片拼接在一起，如下图：



但是，它的弊端也非常明显。它浪费了几乎一半的内存空间来做这个事情，如果资源本来就很有限，这就是一种无法容忍的浪费。

3. **标记-整理：**

该方法不用分配一个对等的额外空间，该方法就是移动所有存活的对象，且按照内存地址顺序依次排列，然后将末端内存地址以后的内存全部回收。



我们可以用一个理想的算法来看一下这个过程。

```
last = 0
for(i=0;i<mems.length;i++){
  if(mems[i] != null){
      mems[last++] = mems[i]
      changeReference(mems[last])
  }
}
clear(mems,last,mems.length)
```

但是需要注意，这只是一个理想状态。对象的引用关系一般都是非常复杂的，我们这里不对具体的算法进行描述。你只需要了解，**从效率上来说，一般整理算法是要低于复制算法的。**

内存回收算法总结：

- **标记-清除（Mark-Sweep）**

效率一般，缺点是会造成内存碎片问题。

- **复制算法（Copy）**

复制算法是所有算法里面效率最高的，缺点是会造成一定的空间浪费。

- **标记-整理（Mark-Compact）**

效率比前两者要差，但没有空间浪费，也消除了内存碎片问题。

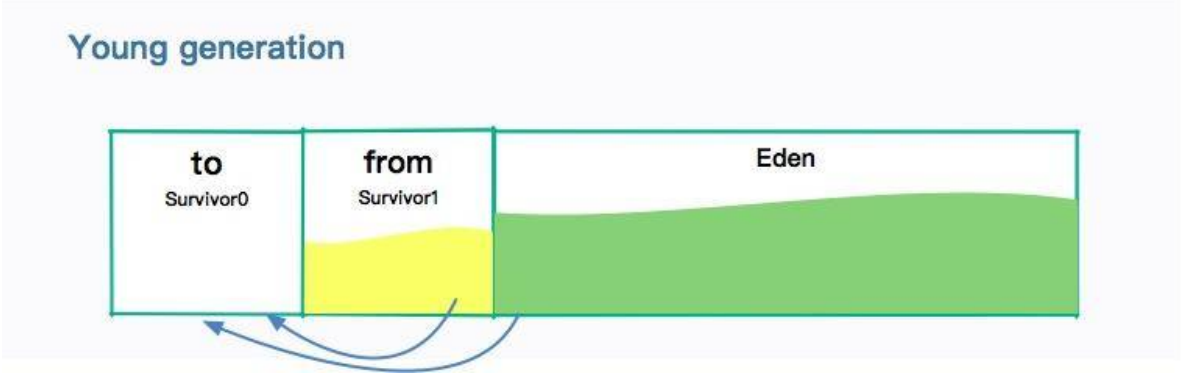
所以，没有最优的算法，只有最合适的算法。存活对象比较高使用标记-清除或整理，存活对象比较低使用复制算法这样空间不会浪费太多。

年轻代和老年代：

年轻代和老年代是逻辑上不同对象的区分，老年代是满足某些条件成为的。

- **年轻代：**

年轻代使用的垃圾回收算法是复制算法。因为年轻代发生 GC 后，只会有非常少的对象存活，复制这部分对象是非常高效的。



如上图，年轻代分为：一个伊甸园空间（Eden），两个幸存者空间（Survivor）

当Eden 区分配满的时候，就会触发年轻代的 GC（Minor GC）。具体过程如下：

1. 在 Eden 区执行了第一次 GC 之后，存活的对象会被移动到其中一个 Survivor 分区（图中的from）；
2. Eden 区再次 GC，这时会采用复制算法，将 Eden 和 from 区一起清理。存活的对象会被复制到 to 区；接下来，只需要清空 from 区就可以了。

所以在这个过程中，总会有一个 Survivor 分区是空置的。Eden、from、to 的默认比例是 8:1:1，所以只会造成 10% 的空间浪费，比例可以是由参数 **-XX:SurvivorRatio** 进行配置的（默认为 8）。

什么是TLAB：

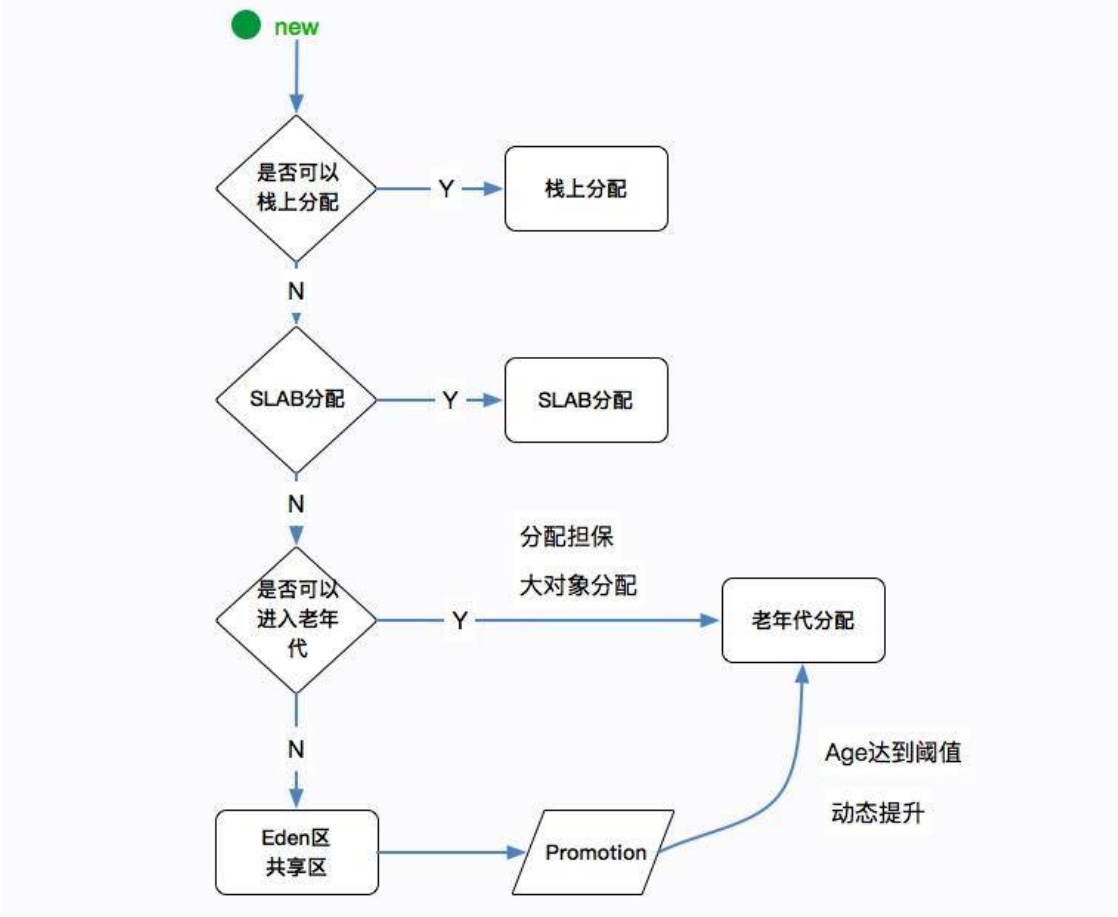
TLAB 是一种优化技术，TLAB 是**每个线程独有的buffer 区域，用来加速对象分配。**这个 buffer 就放在 Eden 区中。对象的分配优先在 TLAB上 分配，但 TLAB 通常都很小，所以对象相对比较大的时候，会在 Eden 区的共享区域进行分配。

- **老年代：**

  老年代一般使用“标记-清除”、“标记-整理”算法，因为老年代的对象存活率一般是比较高的，空间又比较大，拷贝起来并不划算，还不如采取就地收集的方式。

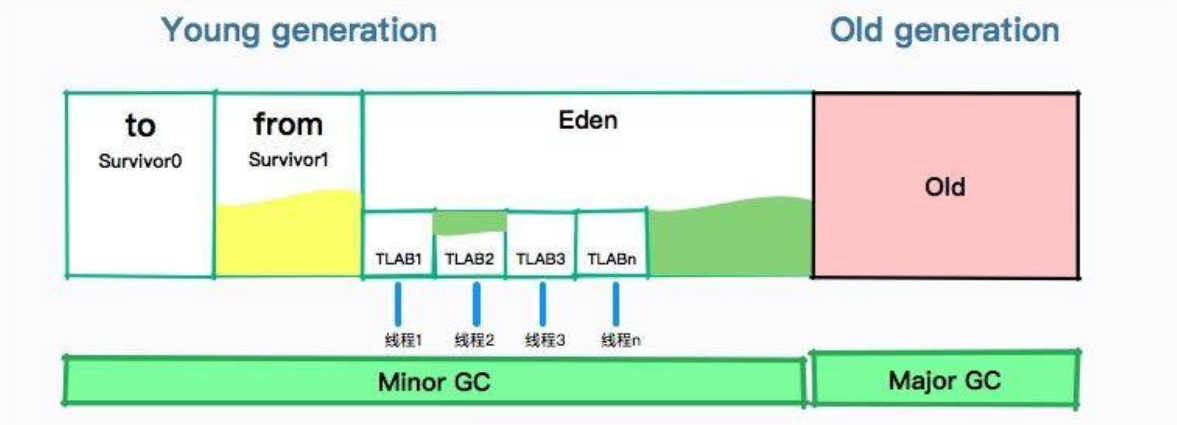
提升为老年代的几种途径：

1. **提升：**发生一次GC，存活的年龄加1，超过阈值可提升为老年代，可以通过参数-XX:+MaxTenuringThreshold进行配置，该阈值最大值为15，因为它是用4bit存储的。
2. **分配担保：**年轻代Survivor空间不够，多余的会变为老年代。
3. **大对象直接在老年代分配：**超过某个大小的对象直接在老年代分配，这个值是通过参数-XX:PretenureSizeThreshold进行配置的。默认为0，意思是全部首选Eden区进行分配。
4. **动态对象年龄判定：**会使用一些动态的计算方法判断是否进入老年代，比如幸存区中相同年龄对象大小的和，大于幸存区的一半，大于或等于age的对象将会直接进入老年代。（这些动态判定一般不受外部控制，我们知道有这么回事就可以了。通过下图可以看一下一个对象的分配逻辑。）



HotSpot几种垃圾回收器：

在此之前，我们把上面的分代垃圾回收整理成一张大图，在介绍下面的收集器时，你可以对应一下它们的位置。



年轻代垃圾回收器：

年轻代是 GC 的重灾区，大部分对象活不到老年代；

（1）Serial 垃圾收集器：

最简单的垃圾回收器，处理 GC 的只有一条线程，并且在垃圾回收的过程中暂停一切用户线程。

它通常用在客户端应用上。因为客户端应用不会频繁创建很多对象，用户也不会感觉出明显的卡顿。相反，它使用的资源更少，也更轻量级。

（2）ParNew 垃圾收集器：

由多条 GC 线程并行地进行垃圾清理。清理过程依然要停止用户线程。

ParNew 追求“低停顿时间”，与 Serial 唯一区别就是使用了多线程进行垃圾收集，在多 CPU 环境下性能比 Serial 会有一定程度的提升；**但线程切换需要额外的开销，因此在单 CPU 环境中表现不如 Serial。**

（3）Parallel Scavenge 垃圾收集器：

另一个多线程版本的垃圾回收器。它与 ParNew 的主要区别是：

- Parallel Scavenge：追求 CPU 吞吐量，能够在较短时间内完成指定任务，适合没有交互的后台计算。**弱交互强计算。**
- ParNew：追求降低用户停顿时间，适合交互式应用。**强交互弱计算。**

Parallel Scavenge收集器与ParNew收集器的一个重要区别是它具有自适应调节策略，Parallel Scavenge的自适应调节策略

Parallel Scavenge收集器有一个参数- XX：+UseAdaptiveSizePolicy当这个参数打开之后，**就不需要手动指定新生代的大小，Eden和Survivor区的比例，**晋升老年代对象等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大吞吐量，这种调节方式成为GC自适应的调节策略。

老年代垃圾回收器：

（1）Serial Old 垃圾收集器：

与年轻代的 Serial 垃圾收集器对应，都是单线程版本，同样适合客户端使用。

年轻代的 Serial，使用复制算法。

老年代的 Old Serial，使用标记-整理算法。

（2）Parallel Old：

Parallel Old 收集器是 Parallel Scavenge 的老年代版本，追求 CPU 吞吐量。

（3）CMS 垃圾收集器：

CMS（Concurrent Mark Sweep）收集器是以获取最短 GC 停顿时间为目标的收集器，它在垃圾收集时使得用户线程和 GC 线程能够并发执行，因此在垃圾收集过程中用户也不会感到明显的卡顿。我们会在后面的课时详细介绍它。

长期来看，CMS 垃圾回收器，是要被 G1 等垃圾回收器替换掉的。在 Java8 之后，使用它将会抛出一个警告。

```
Java HotSpot(TM) 64-Bit Server VM warning: Option UseConcMarkSweepGC was deprecated in version 9.0 and will likely be removed in a future release.
```

配置参数:

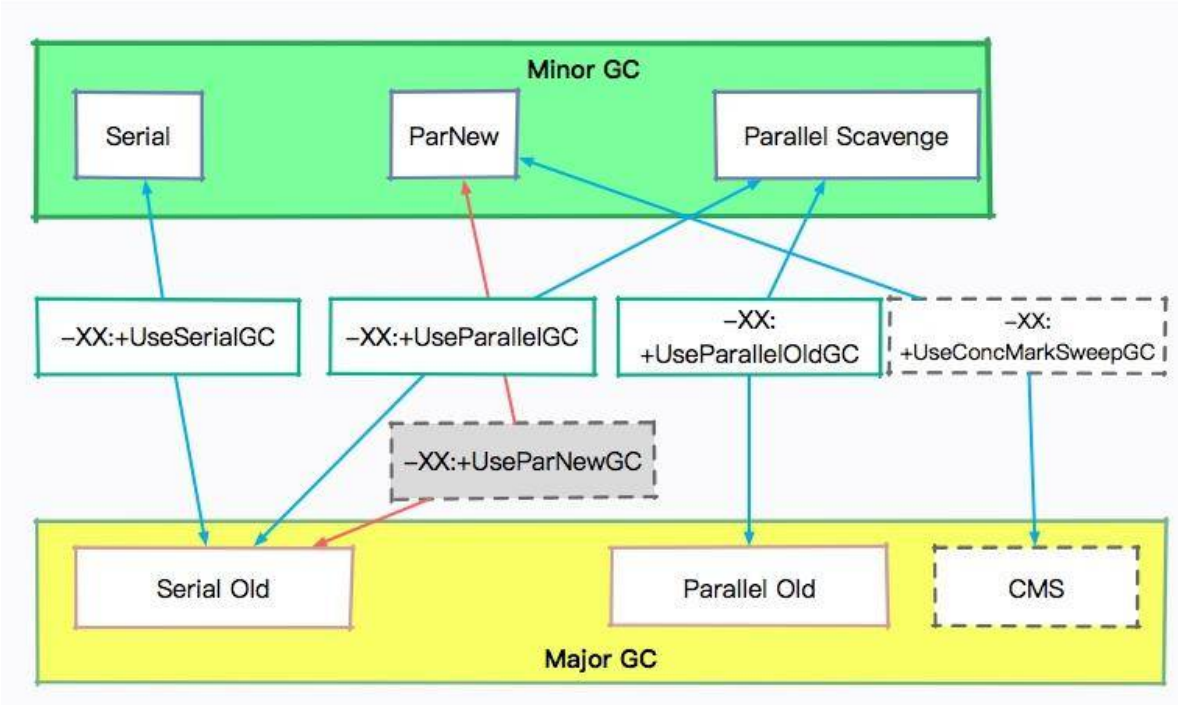
-XX:+PrintCommandLineFlags： 该参数可以查看当前 Java 版本默认使用的垃圾回收器。

```
java  -XX:+PrintCommandLineFlags  -version
//下面是结果
-XX:G1ConcRefinementThreads=4 -XX:GCDrainStackTargetSize=64 -
XX:InitialHeapSize=134217728 -XX:MaxHeapSize=2147483648 -XX:MinHeapSize=6815736 -
XX:+PrintCommandLineFlags -XX:ReservedCodeCacheSize=251658240 -
XX:+SegmentedCodeCache -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -
XX:+UseG1GC

java version "13.0.1" 2019-10-15

Java(TM) SE Runtime Environment (build 13.0.1+9)

Java HotSpot(TM) 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
```

以下是一些垃圾回收器的配置参数:

- -XX:+UseSerialGC 年轻代和老年代都用串行收集器
- -XX:+UseParNewGC 年轻代使用 ParNew，老年代使用 Serial Old
- -XX:+UseParallelGC 年轻代使用 ParallerGC，老年代使用 Serial Old
- -XX:+UseParallelOldGC 新生代和老年代都使用并行收集器
- -XX:+UseConcMarkSweepGC，表示年轻代使用 ParNew，老年代的用 CMS
- -XX:+UseG1GC 使用 G1垃圾回收器
- -XX:+UseZGC 使用 ZGC 垃圾回收器

为了让你有个更好的印象，请看下图。它们的关系还是比较复杂的。尤其注意 -XX:+UseParNewGC 这个参数，已经在 Java9 中就被抛弃了。很多程序（比如 ES）会报这个错误，不要感到奇怪。



有这么多垃圾回收器和参数，那我们到底用什么？在什么地方优化呢?

目前，虽然 Java 的版本比较高，但是使用最多的还是 Java8。从 Java8 升级到高版本的 Java 体系，是有一定成本的，所以 CMS 垃圾回收器还会持续一段时间。

线上使用最多的垃圾回收器，就有 CMS 和 G1，以及 Java8 默认的 Parallel Scavenge。

- CMS 的设置参数：-XX:+UseConcMarkSweepGC。
- Java8 的默认参数：-XX:+UseParallelGC。
- Java13 的默认参数：-XX:+UseG1GC。

我们的实战练习的课时中，就集中会使用这几个参数。

STW：

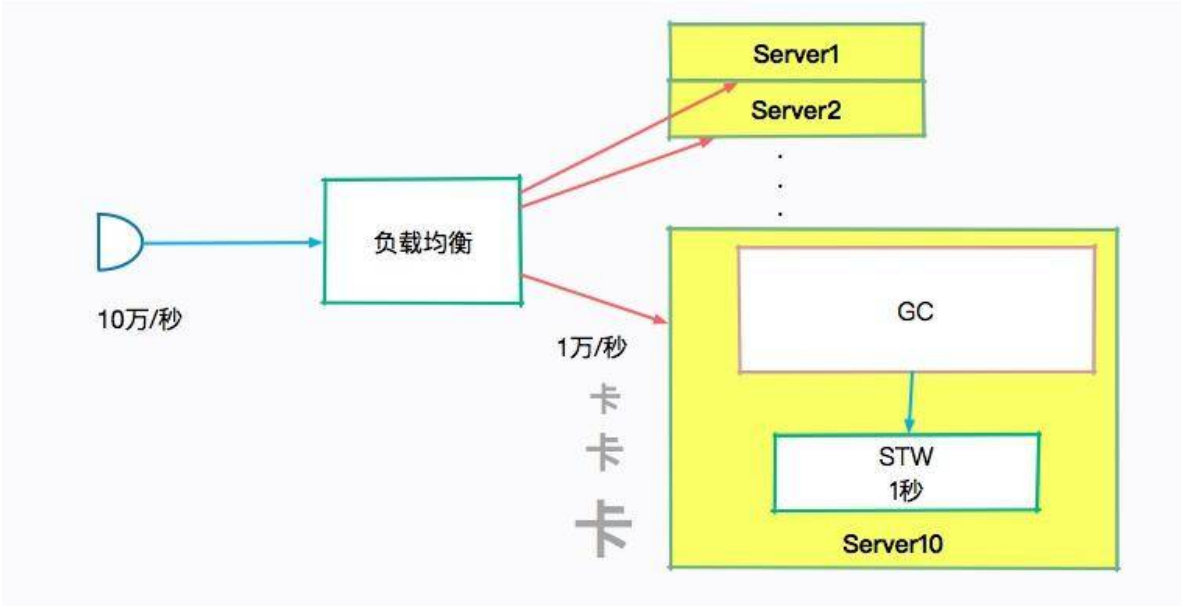
在垃圾回收时，有新对象进入，为了保证程序不会乱套，最好的办法就是暂停用户的一切线程。也就是在这段时间，你是不能 new 对象的，只能等待。表现在 JVM 上就是短暂的卡顿。这个头疼的现象，就叫作 Stop the world。简称 STW。

标记阶段，大多数是要 STW 的。如果不暂停用户进程，在标记对象的时候，有可能有其他用户线程会产生一些新的对象和引用，造成混乱。

现在的垃圾回收器，都会尽量去减少这个过程。但即使是最先进的 ZGC，也会有短暂的 STW 过程。我们要做的就是在现有基础设施上，尽量减少 GC 停顿。

你可能对 STW 的影响没有什么概念，我举个例子来说明下。

某个高并发服务的峰值流量是 10 万次/秒，后面有 10 台负载均衡的机器，那么每台机器平均下来需要 1w/s。假如某台机器在这段时间内发生了 STW，持续了 1 秒，那么本来需要 10ms 就可以返回的 1 万个请求，需要至少等待 1 秒钟。



在用户那里的表现，就是系统发生了卡顿。如果我们的 GC 非常的频繁，这种卡顿就会特别的明显，严重影响用户体验。

虽然说 Java 为我们提供了非常棒的自动内存管理机制，但也不能滥用，因为它是有 STW 硬伤的。

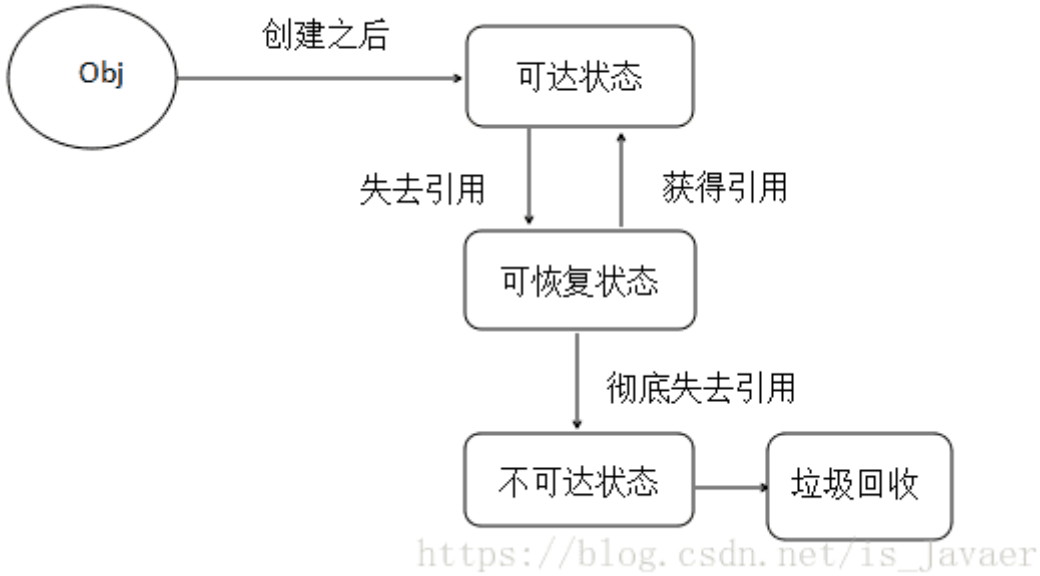
卡片标记：

对于是、否的判断，我们通常都会用 Bitmap（位图）和布隆过滤器来加快搜索的速度。如果你不知道这个概念就需要课后补补课了。？？？

老年代被分成众多的卡页（card page）的（一般数量是 2 的次幂），卡表是标记卡页状态的一个集合，每个卡表项对应一个卡页。如果年轻代有对象分配，而且**老年代有对象指向这个新对象， 那么这个老年代对象所对应内存的卡页，就会标识为 dirty**，卡表只需要非常小的存储空间就可以保留这些状态。垃圾回收时，就可以先读这个卡表，进行快速判断。（老年代卡页，卡表就是保留卡页的状态信息，如果老年代引用了新生代，就会标识为 dirty）

对象的可达状态、可恢复状态、不可达状态：

如下图所示，如果有一个以上的引用变量引用它，处于可达状态，如果没有引用，对象进入可恢复状态，系统会调用finalize（）方法进行资源清理，如果还是没引用，变为不可达状态，引用的话变为可达状态。对象如果是不可达状态，等待垃圾回收器回收。



https://blog.csdn.net/is_Javaer

CMS垃圾回收过程：

1. **初始标记：（标记老年代直接关联 GC root 的对象，还要标记年轻代中对象的引用，有STW）**

   标记直接关联 GC root 的对象，只是标记第一层，这一过程，还要标记年轻代中对象的引用，这也是 CMS 老年代回收，依然要扫描新生代的原因，**这个过程是 STW 的。**

2. **并发标记：（标记所有可达的对象，老年代对象所对应的卡页，会被标记为 dirty，没有STW）**

   标记所有可达的对象，这个过程会持续比较长的时间，但却**可以和用户线程并行。**在这个阶段的执行过程中，可能会产生很多变化：

   - 有些对象，从新生代晋升到了老年代；
   - 有些对象，直接分配到了老年代；
   - 老年代或者新生代的对象引用发生了变化。

   在这个阶段相应的老年代对象所对应的卡页，会被标记为 dirty，用于后续重新标记阶段的扫描。

3. **并发预清理：【重新标记老年代是dirty的卡页并清除dirty状态的卡页（只是清除卡页，因为被引用的新生代已经在survivor里面，所以卡页就需要删除了），没有STW】**

   不需要 STW，老年代中被标记为 dirty 的卡页中的对象，就会被重新标记，然后清除掉 dirty 的状态。

4. **并发可取消的预清理：（阶段是可选的，进行一次 Minor GC，没有STW）**

   “并发预清理”阶段的一种优化，**该阶段进行一次 Minor GC**。（因为标记动作是需要扫描年轻代的，如果年轻代的对象太多，肯定会严重影响标记的时间。如果在此之前能够进行一次 Minor GC，情况会不会变得好了许多？）

5. **最终标记：（标记老年代中所有存活对象，有STW）**

   CMS 会尝试在**年轻代尽可能空**的情况下运行 Final Remark 阶段，以免接连多次发生 STW 事件。这是 CMS 垃圾回收阶段的**第二次 STW 阶段**，目标是完成老年代中所有存活对象的标记。

6. **并发清除：（回收不可达对象的空间，没有STW）**

   目标是删掉不可达的对象，并回收它们的空间。

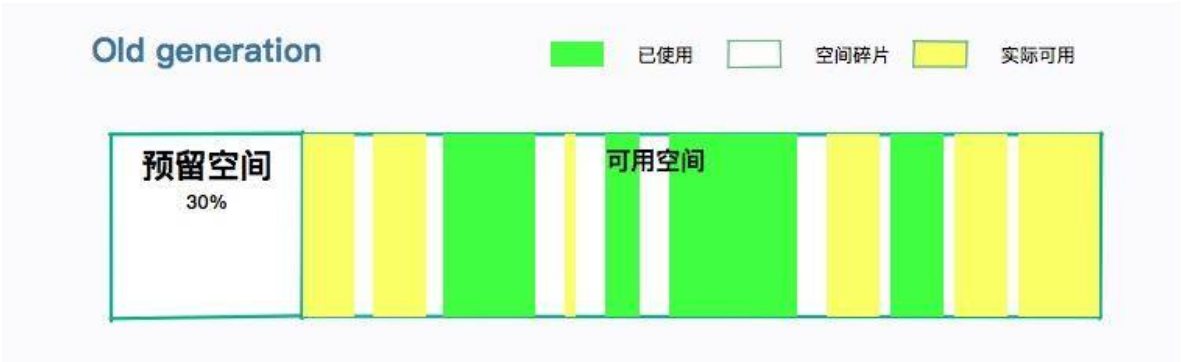
   浮动垃圾：因为清除过程没有STW，还有新的垃圾不断产生，CMS 无法在当次 GC 中处理掉它们，只好留待**下一次 GC 时再清理掉**。这一部分垃圾就称为“浮动垃圾”。

7. **Full GC**

   永久代空间耗尽时会触发，Full GC的持续时间较长。

CMS的预留空间：

如下图， CMS 在执行过程中，有些阶段用户线程还需要运行，老年代空间快满时，再开启这个回收过程，这样停顿时间就很长了（STW），为了防止这种情况，会预留部分空间（在老年代空间预留）。



这部分空间预留，一般在 30% 左右即可，参数 -XX:CMSInitiatingOccupancyFraction 用来配置这个比例（记得要首先开启参数UseCMSInitiatingOccupancyOnly）。也就是说，当老年代的使用率达到 70%，就会触发 GC 了。如果你的系统老年代增长不是太快，可以调高这个参数，降低内存回收的次数。

其实，这个比率非常不好设置。**一般在堆大小小于 2GB 的时候，都不会考虑 CMS 垃圾回收器。**

CMS 对老年代回收的时候，**并没有内存的整理阶段**，于是出现内存碎片情况。

CMS 提供了两个参数来解决内存碎片：

（1） UseCMSCompactAtFullCollection（默认开启），表示在要进行 Full GC 的时候，进行内存碎片整理。内存整理的过程是无法并发的，所以停顿时间会变长。

（2）CMSFullGCsBeforeCompaction，每隔多少次不压缩的 Full GC 后，执行一次带压缩的 Full GC。默认值为 0，表示每次进入 Full GC 时都进行碎片整理。

所以，预留空间加上内存的碎片，使用 CMS 垃圾回收器的老年代，留给我们的空间就不是太多，这也是 CMS 的一个弱点。

CMS优缺点：

优势：

大部分垃圾回收过程并发执行，有几个步骤是没有STW的。

劣势：

1. 内存碎片问题。Full GC 的整理阶段，会造成较长时间的停顿。
2. 需要预留空间，用来分配收集阶段产生的“浮动垃圾”。
3. 使用更多的 CPU 资源，在应用运行的同时进行堆扫描。

CMS 是一种高度可配置的复杂算法，因此给 JDK 中的 GC 代码库带来了很多复杂性。由于 G1 和 ZGC 的产生，CMS 已经在被废弃的路上。但是，目前仍然有大部分应用是运行在 Java8 及以下的版本之上，针对它的优化，还是要持续很长一段时间。

G1基础介绍：

使用 G1 垃圾回收器不得不设置的一个参数：

-XX:MaxGCPauseMillis=10

年轻代和老年代比例：

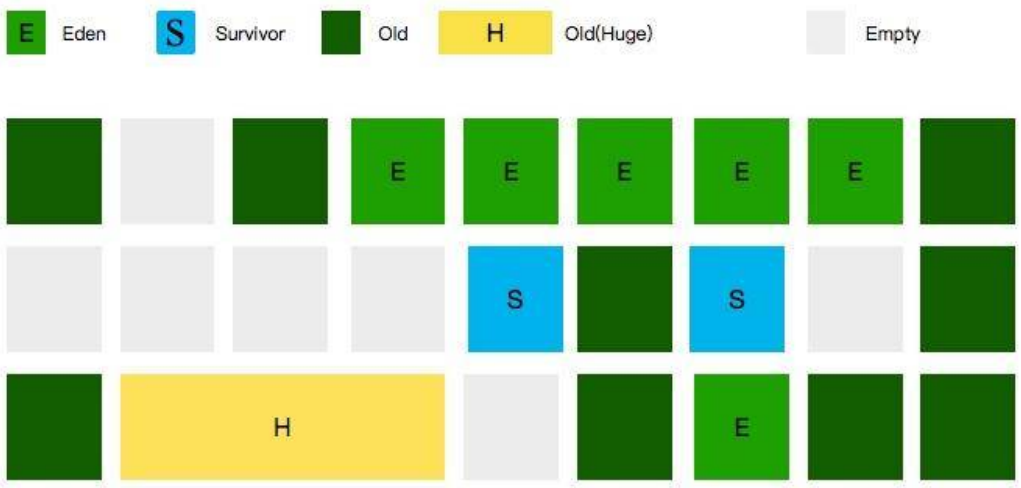
-XX:MaxGCPauseMillis

Region 的大小：

-XX:G1HeapRegionSize=M

为什么叫 G1：

G1 的参数只有 26 个，比CMS的72个简单很多。

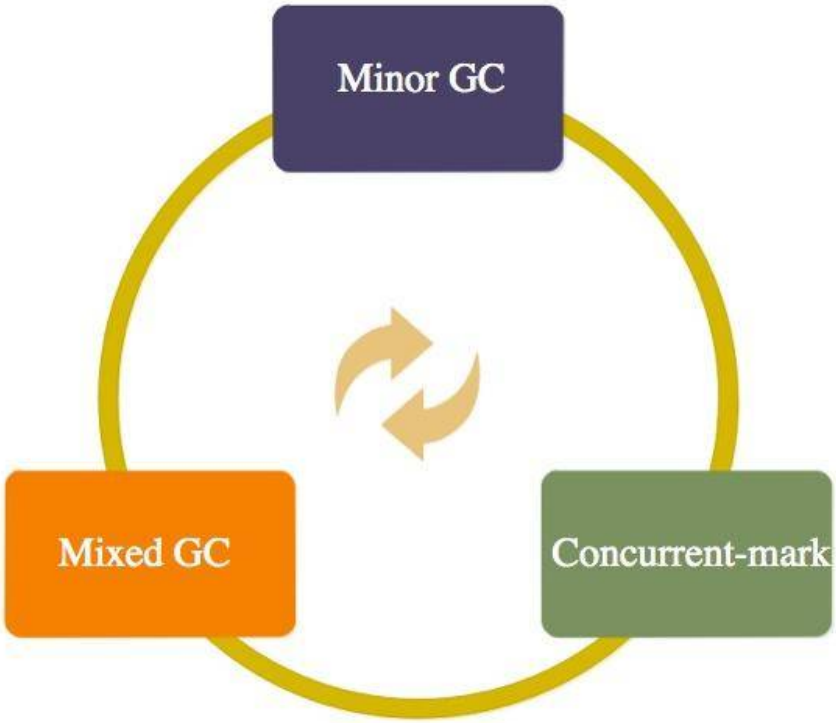


- 如上所示，G1把堆切成了很多份，每一份当做一个小目标，它们在内存上不是连续的，这些小块叫小堆区（Region）。
- 小堆区可以是 Eden 区，也可以是 Survivor 区，还可以是 Old 区。所以 G1 的年轻代和老年代的概念都是逻辑上的。
- 假如我的对象太大，一个 Region 放不下了怎么办?
  - 图中面积很大的黄色区域叫作 Humongous Region，大小超过 Region 50% 的对象，将会在这里分配
- 垃圾最多的Region区，会被优先收集。

G1 的垃圾回收过程：

G1 的回收过程主要分为 3 类:

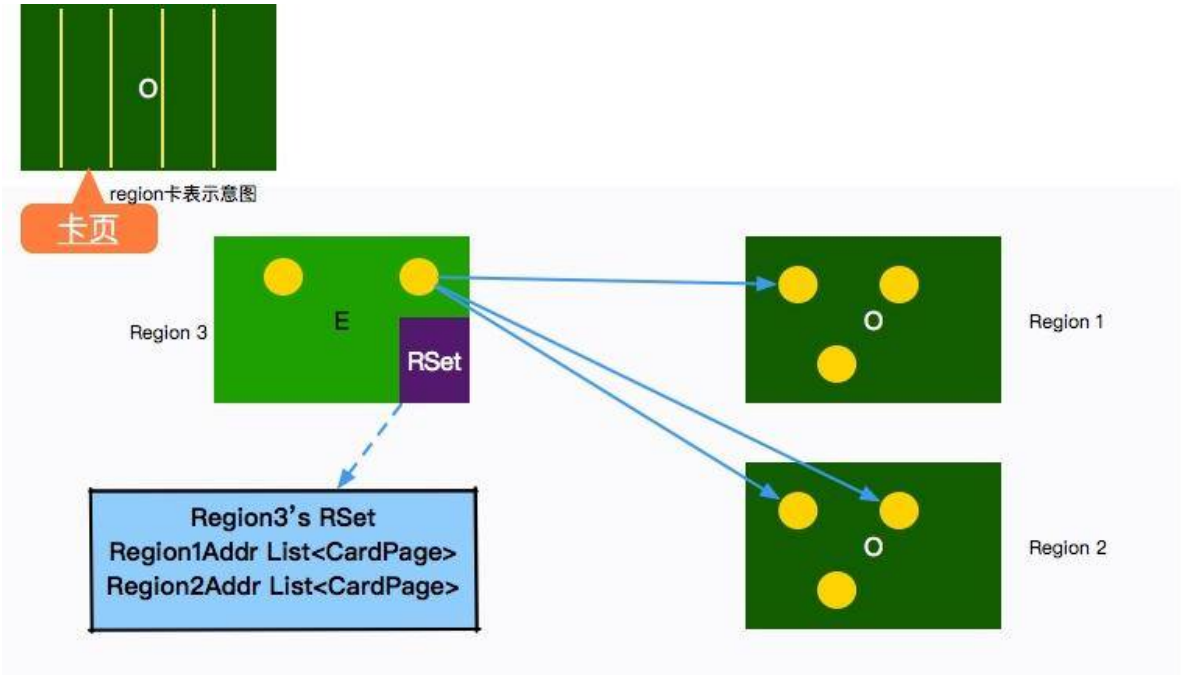
1. G1“年轻代”的垃圾回收
2. 老年代的垃圾收集，严格上来说其实不算是收集，它是一个“并发标记”的过程，顺便清理了一点点对象。
3. 混合模式清理，不止清理年轻代，还会将老年代的一部分区域进行清理。



如上图，这三种模式之间的间隔也是不固定的。比如，1 次 Minor GC 后，发生了一次并发标记，接着发生了 9 次 Mixed GC。

RSet数据结构：

和之前说的卡表（Card Table）类似，只不过记录的是相反的，Card Table 是一种 points-out（我引用了谁的对象）的结构。而 RSet 记录了其他 Region 中的对象引用本 Region 中对象的关系。（用于记录和维护 Region 之间的对象引用关系）



如图，有了RSet，就不必对整个堆内存的对象进行扫描了，它使得部分收集成为了可能。

RSet 通常会占用很大的空间，大约 5% 或者更高，计算开销也是比较大的。（事实上，为了维护 RSet，程序运行的过程中，写入某个字段就会产生一个 post-write barrier 。为了减少这个开销，将内容放入 RSet 的过程是异步的，而且经过了很多的优化：Write Barrier 把脏卡信息存放到本地缓冲区（local buffer），有专门的 GC 线程负责收集，并将相关信息传给被引用 Region 的 RSet。参数 -

XX:G1ConcRefinementThreads 或者 -XX:ParallelGCThreads 可以控制这个异步的过程。如果并发优化线程跟不上缓冲区的速度，就会在用户进程上完成。）

因为老年代回收之前，会先对年轻代进行回收，这时，Eden 区变空了，所以老年代就必要保存来自年轻代的引用，只需保存老年代的引用。

CSet 数据结构：

全称是 Collection Set，即收集集合，将所有存活对象存到该数据结构进行转移。

G1 的具体回收过程：

1. **年轻代回收：（是一个STW过程，这里主要是回收RSet）**

   （1）扫描根：和CMS第一步一致， RSet 记录的其他 Region 的外部引用。

   （2）更新 RS：补充RSet引用关系。

   （3）处理 RS：识别被老年代对象指向的 Eden 中的对象，这些被指向的 Eden 中的对象被认为是存活的对象。

   （4）复制对象：Eden 区内存段中存活的对象复制到Survivor 区中空的 Region，使用**复制算法**收集。

   （5）处理引用：处理 Soft、Weak、Phantom、Final、JNI Weak 等引用

2. **并发标记：（为更加复杂的 Mixed GC 阶段做足了准备）**

   （1）初始标记（Initial Mark）：和年轻代回收扫描根一致。（是一个STW过程，但是时间很短暂）

   （2）Root 区扫描（Root Region Scan）

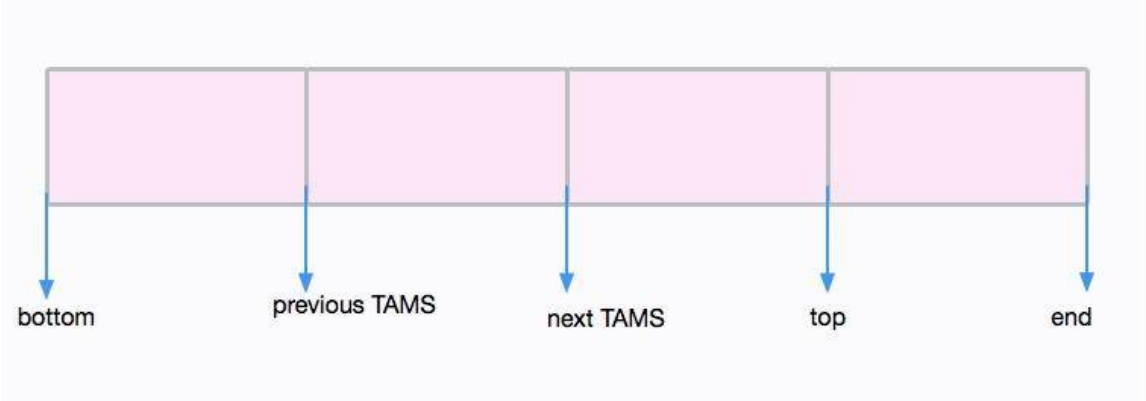
   （3）并发标记（ Concurrent Mark）：收集各个 Region 的存活对象信息。

   （4）重新标记（Remaking）：标记那些在并发标记阶段发生变化的对象。（是一个STW过程）

   （5）清理阶段（Cleanup）：Region 里全是垃圾，在这个阶段会立马被清除掉。不全是垃圾的 Region，并不会被立马处理，它会在 Mixed GC 阶段，进行收集。

   如果在并发标记阶段，又有新的对象变化，该怎么办？

   这是由算法 SATB 保证的。SATB 的全称是 Snapshot At The Beginning，它作用是保证在并发标记阶段的正确性。

   这个快照是逻辑上的，主要是有几个指针，将 Region 分成个多个区段。如图所示，**并发标记期间分配的对象，都会在 next TAMS 和 top 之间，在这个区间的说明是并发标记阶段产生的新对象。**

3. **混合回收：**

   不只清理年轻代，还会将一部分老年代区域也加入到 CSet 中，该过程是并发执行的。通过阈值来触发 混合回收。阈值由 -XX:G1HeapWastePercent 参数进行设置（默认是堆大小的 5%）。

   还有参数 G1MixedGCCountTarget，用于控制一次并发标记之后，最多执行 Mixed GC 的次数。

ZGC介绍：

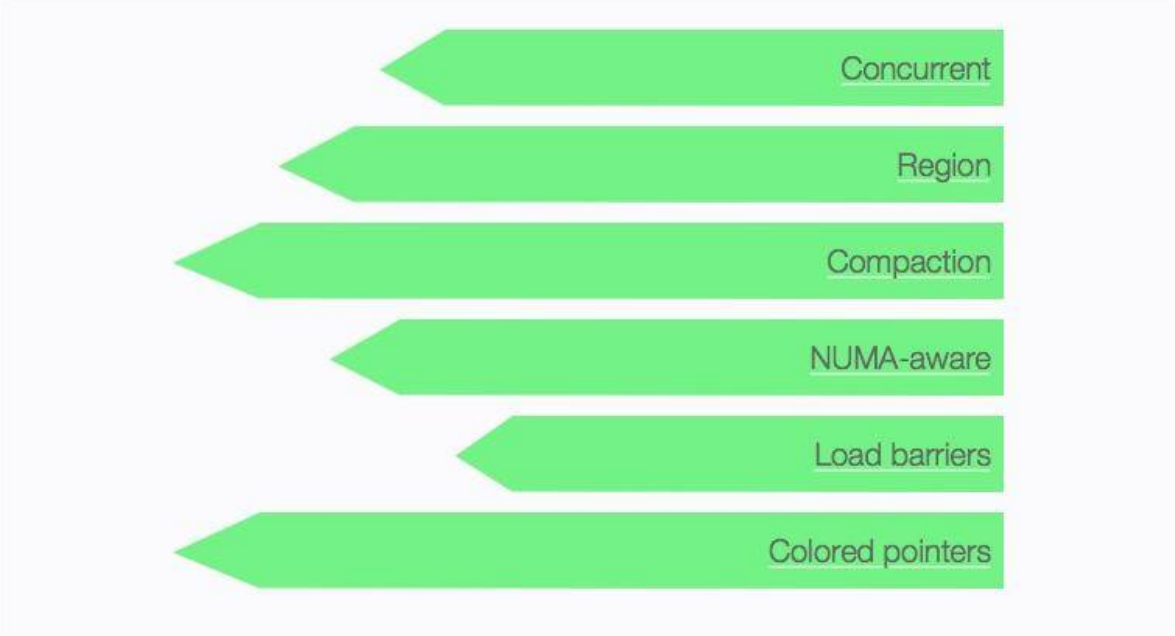
你有没有感觉，在系统切换到 G1 垃圾回收器之后，线上发生的严重 GC 问题已经非常少了？

这归功于 G1 的预测模型和它创新的分区模式。但预测模型也会有失效的时候，它并不是总如我们期望的那样运行，尤其是你给它定下一个苛刻的目标之后。

另外，如果应用的内存非常吃紧，对内存进行部分回收根本不够，始终要进行整个 Heap 的回收，那么 G1 要做的工作量就一点也不会比其他垃圾回收器少，而且因为本身算法复杂了，还可能比其他回收器要差。

所以垃圾回收器本身的优化和升级，从来都没有停止过。最新的 ZGC 垃圾回收器，就有 3 个令人振奋的 Flag：

1. 停顿时间不会超过 10ms；
2. 停顿时间不会随着堆的增大而增大（不管多大的堆都能保持在 10ms 以下）；
3. 可支持几百 M，甚至几 T 的堆大小（最大支持 4T）。



在 ZGC 中，连逻辑上的年轻代和老年代也去掉了，只分为一块块的 page，每次进行 GC 时，都会对 page 进行压缩操作，所以没有碎片问题。ZGC 还能感知 NUMA 架构，提高内存的访问速度。与传统的收集算法相比，ZGC 直接在对象的引用指针上做文章，用来标识对象的状态，所以它只能用在 64 位的机器上。

现在在线上使用 ZGC 的还非常少。即使是用，也只能在 Linux 平台上使用。等待它的普及，还需要一段时间。

CMS和G1的垃圾回收总结：

年轻代都是将Eden区的存活转移到Suvivor区，将老年代和新生代的引用关系（CMS是dirty，G1是RSet）清除，再将老年代和新生代的非可达对象清除。【CMS是并发清除+FullGC（只有老年代空间不足会触发），G1是混合模式清理】

案例实战：亿级流量高并发下如何进行估算和调优

前言：

我们知道，垃圾回收器一般使用默认参数，就可以比较好的运行。但如果用错了某些参数，那么后果可能会比较严重，我不只一次看到有同学想要验证某个刚刚学到的优化参数，结果引起了线上 GC 的严重问题。

所以你的应用程序如果目前已经满足了需求，那就不要再随便动这些参数了。另外，优化代码获得的性能提升，远远大于参数调整所获得的性能提升，你不要纯粹为了调参数而走了弯路。

那么，GC 优化有没有可遵循的一些规则呢？这些“需求”又是指的什么？我们可以将目标归结为三点：

- **系统容量（Capacity）：** 领导要求你每个月的运维费用不能超过 x 万，那就决定了你的机器最多是 2C4G 的。
- **延迟（Latency）：** 请求时等待响应的，涉及影响顾客的满意度。（响应能力是以最大的延迟时间来判断的，比如：一个桌面按钮对一个触发事件响应有多快；需要多长时间返回一个网页；查询一行 SQL 需要多长时间，等等。）
- **吞吐量（Throughput）：** 也就是每天制作的面包数量。（吞吐量大不代表响应能力高，吞吐量一般这么描述：在一个时间段内完成了多少个事务操作；在一个小时之内完成了多少批量操作。）

选择垃圾回收器：

- 如果你的**堆大小不是很大**（比如 100MB），选择串行收集器一般是效率最高的。参数：-XX:+UseSerialGC。
- 如果你的应用运行在**单核的机器上**，或者你的虚拟机核数只有 1C，选择串行收集器依然是合适的，这时候启用一些并行收集器没有任何收益。参数：-XX:+UseSerialGC。
- 如果你的应用是“**吞吐量”优先**的，并且对较长时间的停顿**没有**什么特别的要求。选择并行收集器是比较好的。参数：-XX:+UseParallelGC。
- 如果你的应用对**响应时间要求较高，想要较少的停顿**。甚至 1 秒的停顿都会引起大量的请求失败，那么选择 G1、ZGC、CMS 都是合理的。虽然这些收集器的 GC 停顿通常都比较短，但它需要一些额外的资源去处理这些工作，通常吞吐量会低一些。参数：-XX:+UseConcMarkSweepGC、-XX:+UseG1GC、-XX:+UseZGC 等。

从上面这些出发点来看，我们平常的 Web 服务器，都是对响应性要求非常高的。选择性其实就集中在 CMS、G1、ZGC 上。

大流量应用特点：

这是一类对**延迟非常敏感**的系统。吞吐量一般可以通过**堆机器解决。**

如果一项业务有价值，客户很喜欢，那亿级流量很容易就能达到了。假如某个接口一天有 10 亿次请求，每秒的峰值大概也就 5~6 w/秒，虽然不算是很大，但也不算小。最直接的影响就是：可能你发个版，几万用户的请求就抖一抖。

一般达到这种量级的系统，承接请求的都不是一台服务器，接口都会要求快速响应，一般不会超过 **100ms。**

这种系统，一般都是社交、电商、游戏、支付场景等，要求的是短、平、快。长时间停顿会堆积海量的请求，所以在停顿发生的时候，表现会特别明显。我们要考量这些系统，有很多指标。

- 每秒处理的事务数量（TPS）；
- 平均响应时间（AVG）；
- TP 值，比如 TP90 代表有 90% 的请求响应时间小于 x 毫秒。

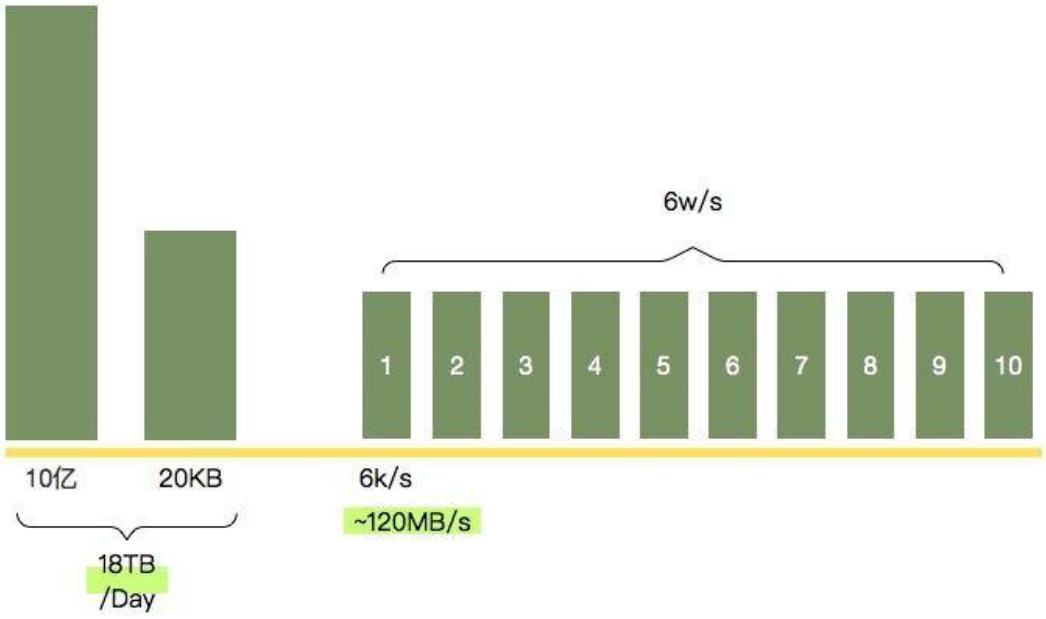
可以看出来，它和 JVM 的某些指标很像。

尤其是 TP 值，最能代表系统中到底有多少长尾请求，这部分请求才是影响系统稳定性的元凶。大多数情况下，GC 增加，长尾请求的数量也会增加。

我们的目标，就是减少这些停顿。本课时假定使用的是 CMS 垃圾回收器。

流量估算：

假如是查询用户在社交网站上发送的帖子，还需要查询第一页的留言（大概是 15 条）这种类型的数据结构，一般返回体都比较大，大概会有几 KB 到几十 KB 不等。我们就可以对这些数据进行以大体估算。具体的数据来源可以看日志，也可以分析线上的请求。



这个接口每天有 10 亿次请求，假如每次请求的大小有 20KB（很容易达到），那么一天的流量就有 18TB 之巨。假如高峰请求 6w/s，我们部署了 10 台机器，那么每个 JVM 的流量就可以达到 120MB/s，这个速度算是比较快的了。

如果你实在不知道怎么去算这个数字，那就按照峰值的 2 倍进行准备，一般都是 OK 的。

对以上场景调优：

问题是这样的，我们的机器是 4C8GB 的，分配给了 JVM 1024x8GB/3x2= 5460MB（这里计算的是JVM占用物理内存2/3的情况，是推荐的比例，默认是物理内存的1/4） 的空间。那么年轻代大小就有 5460MB/3=1820MB。进而可以推断出，Eden 区的大小约 1456MB，那么大约只需要 12 秒，就会发生一次 Minor GC。不仅如此，每隔半个小时，会发生一次 Major GC。

不管是年轻代还是老年代，这个 GC 频率都有点频繁了。

提醒一下，你可以算一下我们的 Survivor 区大小，大约是 182MB 左右，如果稍微有点流量偏移，或者流量突增，再或者和其他接口共用了 JVM，那么这个 Survivor 区就已经装不下 Minor GC 后的内容了。总有一部分超出的容量，需要老年代来补齐。这些垃圾信息就要保存更长时间，直到老年代空间不足。

我们发现，用户请求完这些信息之后，很快它们就会变成垃圾。所以每次 MinorGC 之后，剩下的对象都很少。

也就是说，我们的流量虽然很多，但大多数都在年轻代就销毁了。如果我们加大年轻代的大小，由于 GC 的时间受到活跃对象数的影响，回收时间并不会增加太多。

如果我们把一半空间给年轻代。也就是下面的配置：

-XX:+UseConcMarkSweepGC -Xmx5460M -Xms5460M -Xmn2730M

重新估算一下，发现 Minor GC 的间隔，由 12 秒提高到了 18 秒。

线上观察：

[ParNew: 2292326K->243160K(2795520K), 0.1021743 secs]

3264966K->10880154K(1215800K), 0.1021417 secs]

[Times: user=0.52 sys=0.02, real=0.2 secs]

Minor GC 有所改善，但是并没有显著的提升。相比较而言，Major GC 的间隔却增加到了 3 小时，是一个非常大的性能优化。这就是在容量限制下的初步调优方案。

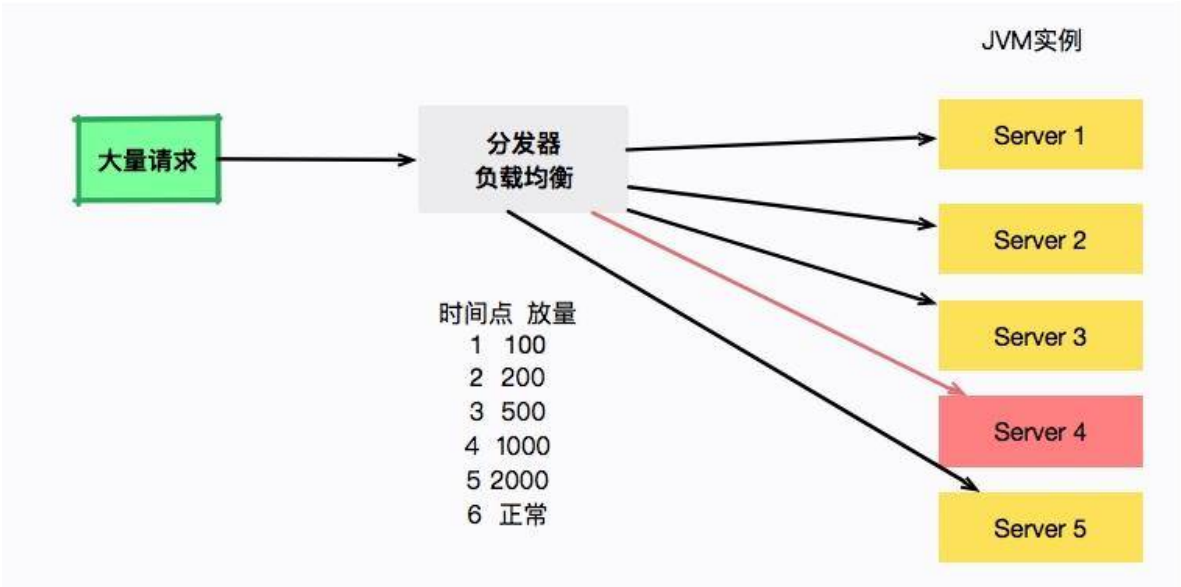
此种场景，我们可以更加激进一些，调大年轻代（顺便调大了幸存区），让对象在年轻代停留的时间更长一些，有更多的 buffer 空间。这样 Minor GC 间隔又可以提高到 23 秒。参数配置：

-XX:+UseConcMarkSweepGC -Xmx5460M -Xms5460M -Xmn3460M

一切看起来很美好，但还是有一个瑕疵。

问题如下：由于每秒的请求都非常大，如果应用重启或者更新，流量瞬间打过来，JVM 还没预热完毕，这时候就会有大量的用户请求超时、失败。

为了解决这种问题，通常会逐步的把新发布的机器进行放量预热。比如第一秒 100 请求，第二秒 200 请求，第三秒 5000 请求。大型的应用都会有这个预热过程。



如图所示，负载均衡器负责服务的放量，server4 将在 6 秒之后流量正常流通。但是奇怪的是，每次重启大约 20 多秒以后，就会发生一次诡异的 Full GC。

注意是 Full GC，而不是老年代的 Major GC，也不是年轻代的 Minor GC。

事实上，经过观察，此时年轻代和老年代的空间还有很大一部分，那 Full GC 是怎么产生的呢？

一般，Full GC 都是在老年代空间不足的时候执行。但不要忘了，我们还有一个区域叫作 Metaspace，它的容量是没有上限的，但是**每当它扩容时，就会发生 Full GC**。

使用下面的命令可以看到它的默认值：

java -XX:+PrintFlagsFinal 2>&1 | grep Meta

默认值如下：

size_t MetaspaceSize = 21807104 {pd product} {default}

size_t MaxMetaspaceSize = 18446744073709547520 {product} {default}

可以看到 MetaspaceSize 的大小大约是 20MB。这个初始值太小了。

现在很多类库，包括 Spring，都会大量生成一些动态类，20MB 很容易就超了，我们可以试着调大这个数值。

按照经验，一般调整成 256MB 就足够了。同时，为了避免无限制使用造成操作系统内存溢出，我们同时设置它的上限。配置参数如下：

-XX:+UseConcMarkSweepGC -Xmx5460M -Xms5460M -Xmn3460M -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M

经观察，启动后停顿消失。

这种方式通常是行之有效的，但也可以通过扩容机器内存或者扩容机器数量的办法，显著地降低 GC 频率。这些都是在估算容量后的优化手段。

我们把部分机器升级到 8C16GB 的机器，使用如下的参数:

-XX:+UseConcMarkSweepGC -Xmx10920M -Xms10920M -Xmn5460M -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M

相比较其他实例，系统运行的特别棒，系统平均 1 分钟左右发生一次 MinorGC，老年代观察了一天才发生 GC，响应水平明显提高。

这是一种非常简单粗暴的手段，但是有效。我们看到，对 JVM 的优化，不仅仅是优化参数本身。我们的目的是解决问题，寻求多种**有用手段**。

调优总结：

如果没有明显的内存泄漏问题和严重的性能问题，专门调优一些 JVM 参数是非常没有必要的，优化空间也比较小。

所以，我们一般优化的思路有一个重要的顺序：

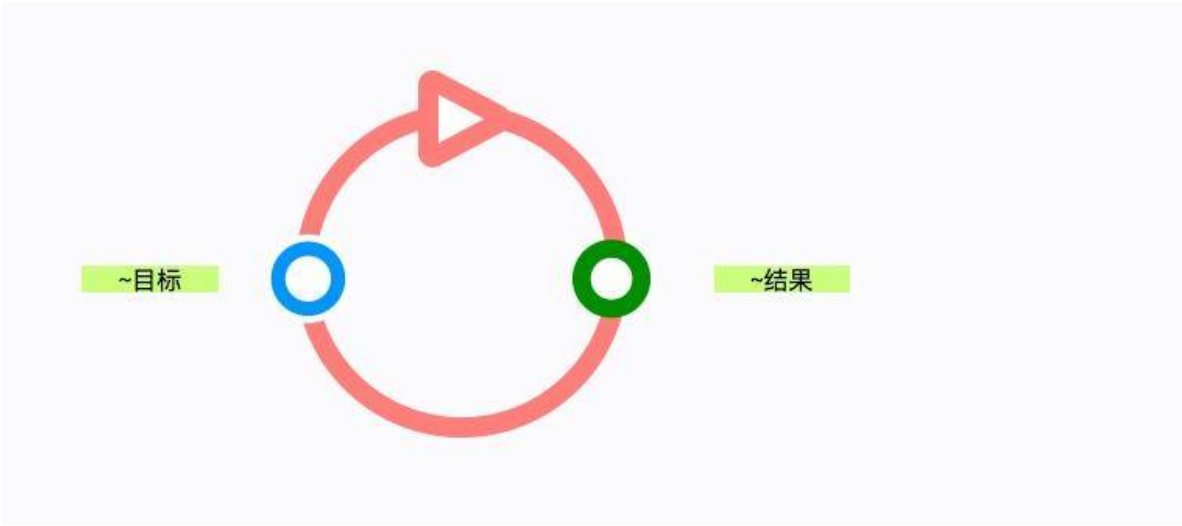
1. 程序优化，效果通常非常大；
2. 扩容，如果金钱的成本比较小，不要和自己过不去；
3. 参数调优，在成本、吞吐量、延迟之间找一个平衡点。

本课时主要是在第三点的基础上，一步一步地增加 GC 的间隔，达到更好的效果。

我们可以再加一些原则用以辅助完成优化。

1. 一个长时间的压测是必要的，通常我们使用 JMeter 工具。

2. 如果线上有多个节点，可以把我们的优化在其中几个节点上生效。等优化真正有效果之后再全面推进。
3. 优化过程和目标之间可能是循环的，结果和目标不匹配，要推翻重来。



我们的业务场景是高并发的。对象诞生的快，死亡的也快，对**年轻代的利用直接影响了整个堆的垃圾收集。**（调优来说，年轻代是比较重要的）

1. 足够大的年轻代，会增加系统的吞吐，但不会增加 GC 的负担。
2. 容量足够的 Survivor 区，能够让对象尽可能的留在年轻代，减少对象的晋升，进而减少 Major GC。

我们还看到了一个元空间引起的 Full GC 的过程，这在高并发的场景下影响会格外突出，尤其是对于使用了大量动态类的应用来说。通过调大它的初始值，可以解决这个问题。

面试题：

假如拿到锁之后java刚刚好执行gc，执行了4、5秒，而锁的过期时间为3秒，怎么办？

1. 加长持有锁时间
2. 派生子线程每隔500毫秒检查锁的状态及执行结果，如果未执行完，自动延期
3. 锁保存version值，业务执行完比较本地和远程的version，如果本地小，则抛异常或业务回滚
4. 优化jvm缩短stw时间

JVM是不是所有对象都在堆上分配？

不是，JVM涉及到**逃逸分析**和**标量替换**，这两种优化分析情况会在栈上分配

- 逃逸分析：

  逃逸分析分为：全局变量逃逸、方法返回值逃逸、实例引用发生逃逸、线程逃逸（赋值给类变量或可以在其他线程中访问的变量）

  我们通过JVM内存分配可以知道JAVA中的对象都是在堆上进行分配，当对象没有被引用的时候，需要依靠GC进行回收内存，如果对象数量较多的时候，会给GC带来较大压力，也间接影响了应用的性能。为了减少临时对象在堆内分配的数量，JVM通过逃逸分析确定该对象不会被外部访问。那就通过标量替换将该对象分解在栈上分配内存，这样该对象所占用的内存空间就可以随栈帧出栈而销毁，就减轻了垃圾回收的压力。

  ```
  1 - 通过-XX:-DoEscapeAnalysis关闭逃逸分析
  ```

- 标量替换：

  1.标量和聚合量

标量即不可被进一步分解的量，而JAVA的基本数据类型就是标量（如：int，long等基本数据类型以及reference类型等），标量的对立就是可以被进一步分解的量，而这种量称之为聚合量。而在JAVA中对象就是可以被进一步分解的聚合量。

2.替换过程

通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，JVM不会创建该对象，而会将该对象成员变量分解若干个被这个方法使用的成员变量所代替。这些代替的成员变量在栈帧或寄存器上分配空间。

> 通过-XX:+EliminateAllocations可以开启标量替换， -XX:+PrintEliminateAllocations查看标量替换情况（Server VM 非Product版本支持）

- 同步消除：

  同步消除是java虚拟机提供的一种优化技术。通过逃逸分析，可以确定一个对象是否会被其他线程进行访问。

  如果你定义的类的方法上有同步锁，但在运行时，却只有一个线程在访问，此时逃逸分析后的机器码，会去掉同步锁运行，这就是没有出现线程逃逸的情况。那该对象的读写就不会存在资源的竞争，不存在资源的竞争，则可以消除对该对象的同步锁。

  > 通过-XX:+EliminateLocks可以开启同步消除,进行测试执行的效率

什么是进程、线程，它们的区别?

1）进程是一个独立的运行环境，它可以被看作是一个程序或者一个应用。而线程是在进程中执行的一个任务。eg:打开360安全卫士，它本身是一个程序，也是一个进程，它里面有杀毒，清理垃圾，电脑加速等功能，当你点击杀毒的时候，杀毒任务就相当于一个线程。

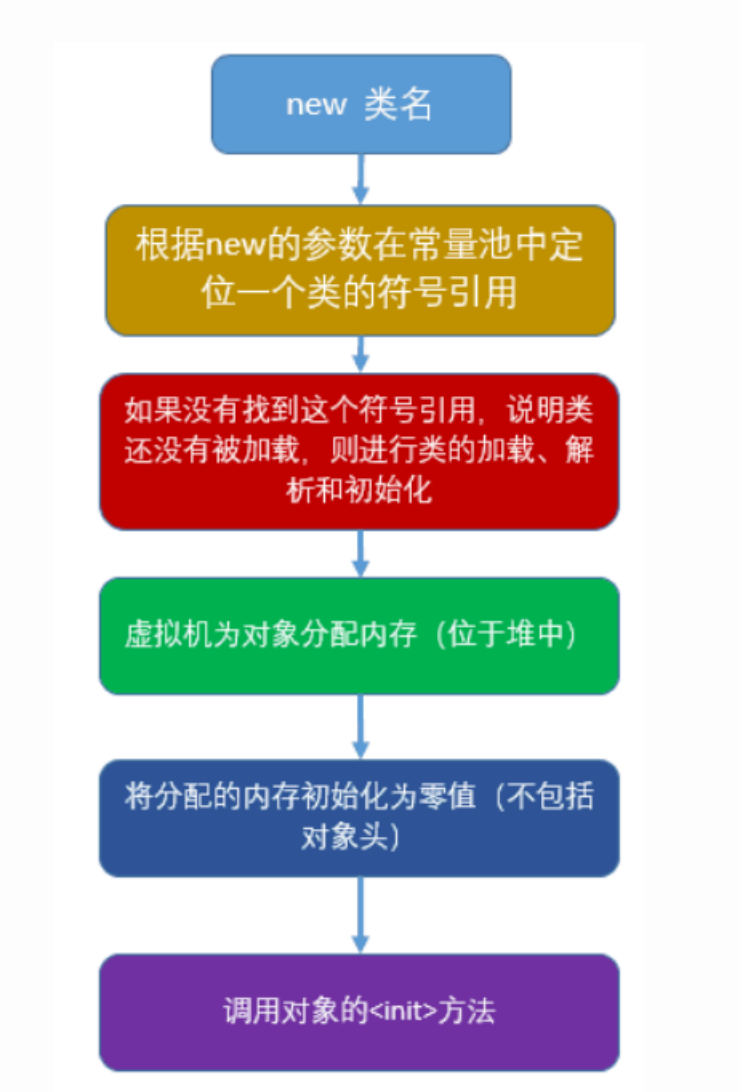
2）进程是操作系统进行资源分配的基本单位，而线程是操作系统进行调度的基本单位。

3）进程让操作系统的并发性成为可能，而线程让进程的内部并发成为可能

局部变量和全局变量在内存中有什么区别?

- 全局变量保存在内存的全局存储区中，占用静态的存储单元；
- 局部变量保存在栈中，只有在所在函数被调用时才动态地为变量分配存储单元。

创建对象的过程



虚拟机栈不用垃圾回收器?

虚拟机栈里的栈帧即对应代码中的一个方法。代码运行的过程，即栈帧入栈出栈的过程。

一个方法执行完，栈帧出栈后，即被销毁。只有入栈出栈这样简单的操作，不需要设计复杂的垃圾回收算法来回收。随着方法的执行，线程的结束正常回收即可。

在递归函数中，该方法还没有结束，就一直不会出栈，如果循环的次数过多，栈空间有被挤爆的可能。会出现StackOverflowError 栈溢出。栈溢出也是内存溢出的一种情况。可通过-Xss （stack size）设置栈大小。

程序计数器没有OOM?

程序计数器（Program Counter Register）也称PC寄存器。是运行时数据区里唯一一块没有Out of Memory的区域。

只存下一个字节码指令的地址，消耗内存小且固定，无论方法多深，他只存一条。

只针对一个线程，随着线程的结束而销毁。

方法区会导致oom吗

创建类过多就会

年轻代的阈值默认

虚拟机给每个对象定义了一个对象年龄计数器。如果对象在Eden出生并经过第一次MinorGC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间，并将对象年龄设为1.对象在Survivor区每熬过一次MinorGC年龄就加一岁，当它的年龄增加到一定程度（默认为15岁）时，就会晋升到老年代中，对象晋升老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold来设置。