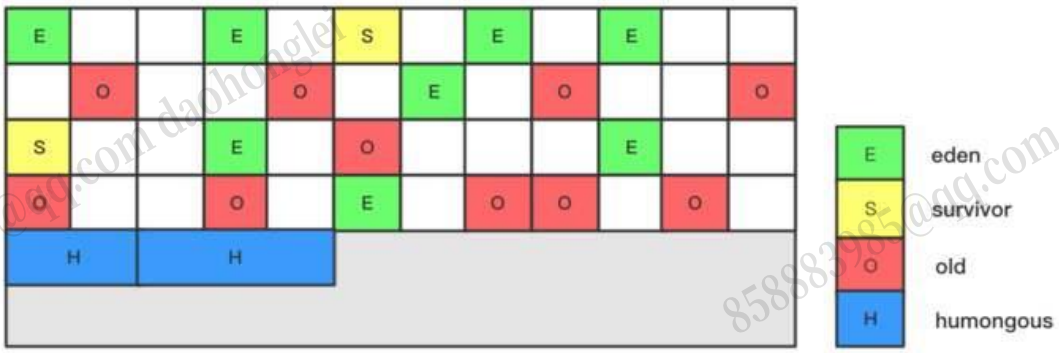
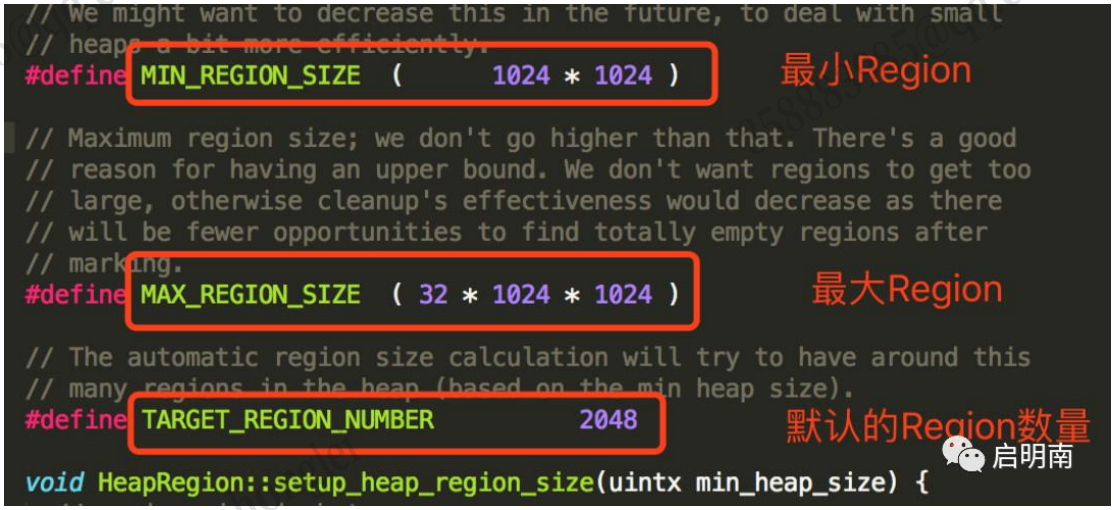
G1 及其后出现的垃圾收集器 ZGC、Shenandoah，它们都是基于 Region 的内存布局形式。它们垃圾收集的目标范围不再是整个新生代（Minor GC）、老年代（Majon GV）、整个堆（Full GC），而是一个一个的 Region。因为这样的内存布局，所以 G1 能做到面向局部收集。

每个 Region 都可以被标记为 E（Eden）、S（Survivor）、O（Old）、H（Humongous），但一个 Region 同一时刻只能是这四个中的一个。H 表示巨型对象，即超过 Region 大小的一半的对象，会直接进入老年代由多个连续的 Region 存储。

Region 的大小可以通过-XX:G1HeapRegionSize 参数指定，如果没有显示指定，则 G1 会计算出一个合理的大小。Region 的取值范围为 1M~32M，且应为 2 的 N 次幂，所以 Region 的大小只能是 1M、2M、4M、8M、16M、32M。比如-Xmx=16g -Xms=16g，则 Region 的大小等于 16G / 2048=8M。也可以推理出 G1 推荐的管理的最大堆内存是 64G





RSet（Remembered Set、记忆集）

在垃圾收集过程中，会存在一种现象，即跨代引用，在 G1 中，又叫跨 Region 引用。如果是年轻代指向老年代的引用我们不用关心，因为即使 Minor GC 把年轻代的对象清理掉了，程序依然能正常运行，而且随着引用链的断掉，无法被标记到的老年代对象会被后续的 Major GC 回收。如果是老年代指向年轻代的引用，那这个引用在 Minor GC 阶段是不能被回收掉的，那如何解决这个问题呢?

最简单的实现方式当然是每个对象中记录这个跨 Region 引用记录，GC 时扫描所有老年代的对象，显然这是一个相当大的 Overhead。为什么呢？因为 IBM 做过这样的实验，发现绝大多数对象都是“朝生夕灭”，等不到进入老年代，能进入老年代的对象最多不到 5%。JVM 的新生代内存比例是 8:1:1 也是基于这个结论设定的。

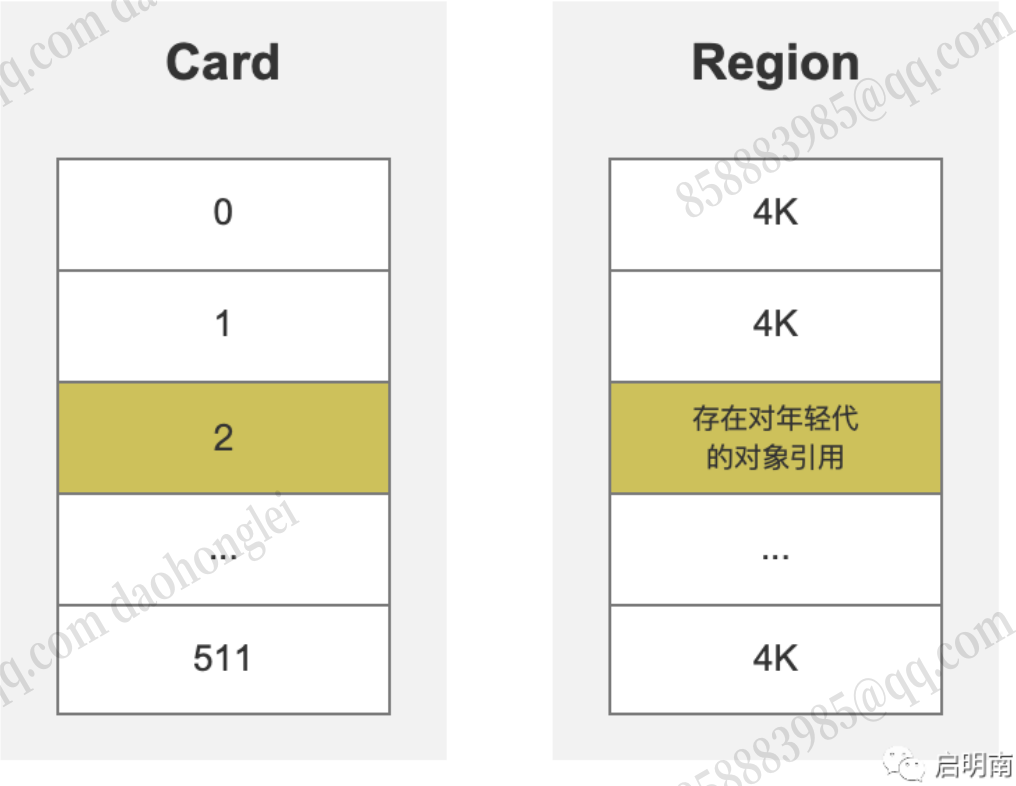
最合理的实现方式自然是记录哪些 Region 中的老年代的对象有指向年轻代的引用。GC 时扫描这些 Region 就行了。这就是 RSet 存在的意义。RSet 本质上是一种哈希表，Key 是 Region 的起始地址，Value 是一个集合，里面存储的元素是卡表的索引号（第几个 Card 的第几个元素）。

Card Table（卡表）

每个 Region 又被分成了若干个大小为 512 字节的 Card，这些 Card 都会记录在全局卡表

中。Card 中的每个元素对应着其标识的内存区域中一块特定大小的内存块，这个内存块被称为卡页。一个卡页的内存中通常不止一个对象，只有卡页中有一个及以上对象的字段存在着跨 Region 引用，这个对应的元素的值就标识为 1。

比如 G1 默认的 Region 有 2048 个，默认每个 Region 为 2M，那每个 Region 对应的 Card 的每个元素对应的卡页的大小为 2M / 512=4K，即这 4K 内存中只要有一个或一个以上的对象存在着跨 Region 对年轻代的引用，这个卡页对应的 Card 的元素值为 1。



这样在 Minor GC 时，只需要将变脏的 Region 中的那个卡页加入 GC Roots 一并扫描即可。比起扫描老年代的所有对象，大大减少了扫描的数据量，提升了效率。