## 收集理论

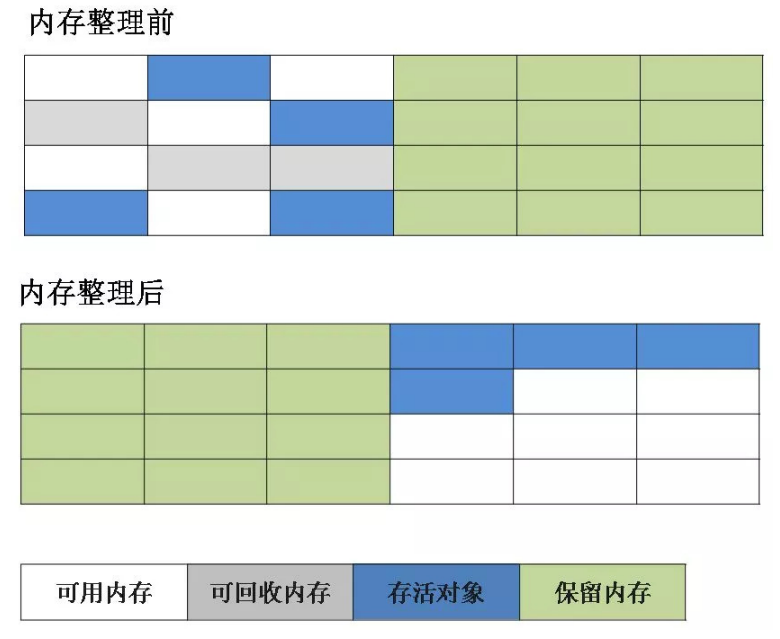
分代收集理论，java堆一般分为年轻代，老年代来区分收集、清除。

### 收集算法

收集算法包含**标记复制、标记-整理、标记-清除。**

#### 标记复制

内存分左右两块。标记左边存活对象然后复制到右边、整理排整齐，最后直接清空左边内存。

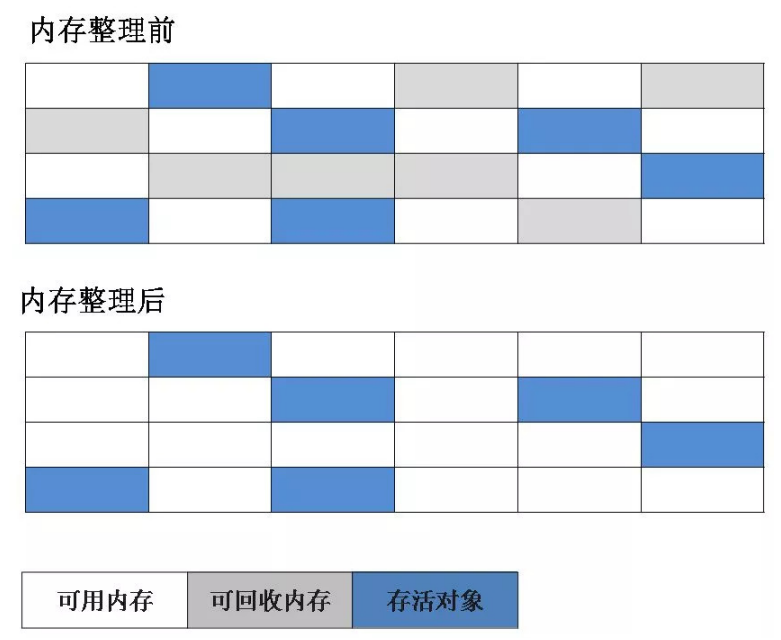


#### **标记清除**

分两步：标记、清除。

先标记过时对象(标记存活对象也可以)，然后清除对象。

问题：1，标记耗时。2，清除后导致内存碎片化。



#### **标记整理**

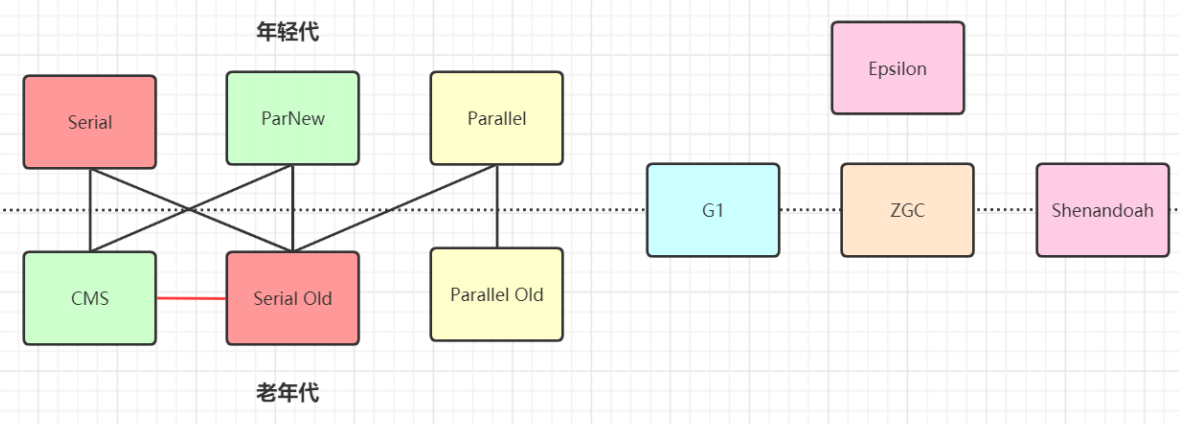
**针对老年代的标记算法，标记过程跟“标记-清除”一样，后续将存活对象向一端移动排齐，然后清空其他内存空间。**



### 垃圾收集器

垃圾收集器基于收集算法开发，包含Serial收集器 、Parallel Scavenge收集器 、ParNew收集器、 CMS收集器、G1 、ZGC等。

目前没完美算法，都是需要根据自身业务而定，下图是各收集器搭配使用情况。



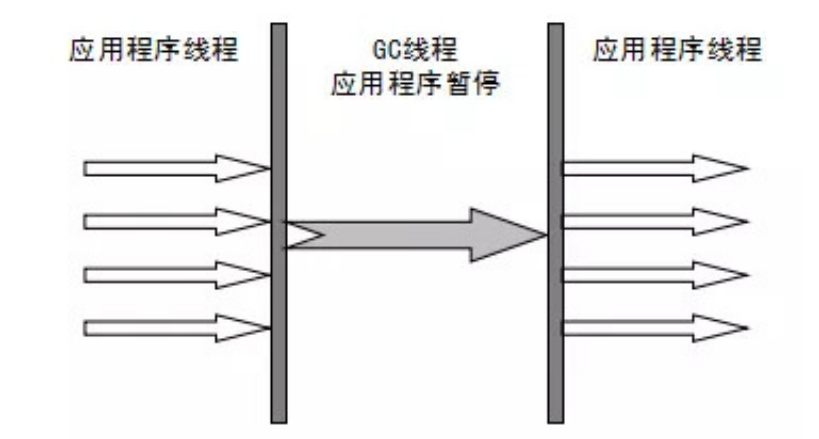
#### Serial收集器

**Serial(串行)收集器**，最古老收集器，单线程执行收集，且收集导致STW。

**新生代使用标记-复制，老年代使用标记-整理算法。**

**Serial Old收集器**是Serial收集器的老年代版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在JDK1.5以及以前的版本中与Parallel Scavenge收集器搭配使用，另一种用途是作为CMS收集器的后备方案。

启用命令:**-XX:+UseSerialGC -XX:+UseSerialOldGC**



#### Parallel Scavenge收集器

**Parallel 收集器**就是**Serial(串行)收集器**的多线程版，在收集的时候使用多线程收集，其余余行为（控制参数、收集算法、回收策略等等）和Serial收集器类似。

默认线程数是cpu核心数，用参数(-XX:ParallelGCThreads)指定收集线程数，但是一般不推荐修改。

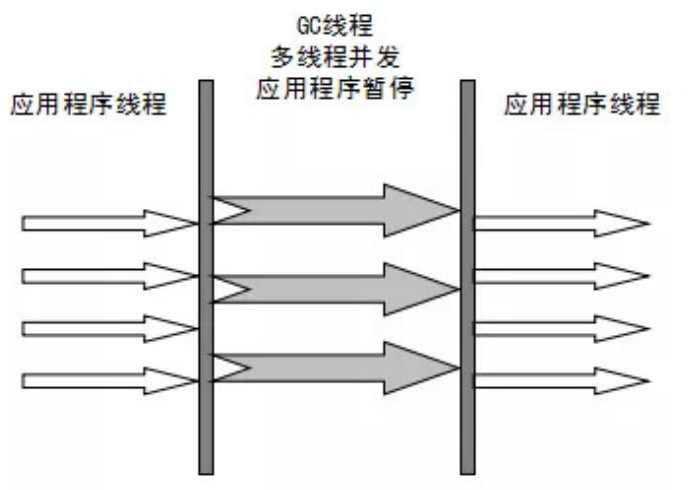
**Parallel Scavenge收集器**更注重吞吐量，而CMS收集器更关注用户体验，减少STM。

何为吞吐量?就是CPU中用于运行用户代码的时间与CPU总消耗时间的比值。

**Parallel Scavenge收集器新生代使用标记-复制，老年代使用标记-整理算法。**

**Parallel Old收集器**是Parallel Scavenge收集器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及CPU资源的场合，都可以优先考虑 Parallel Scavenge收集器和Parallel Old收集器(JDK8默认的新生代和老年代收集器)。

**启用命令:-XX:+UseParallelGC(年轻代),-XX:+UseParallelOldGC(老年代))**



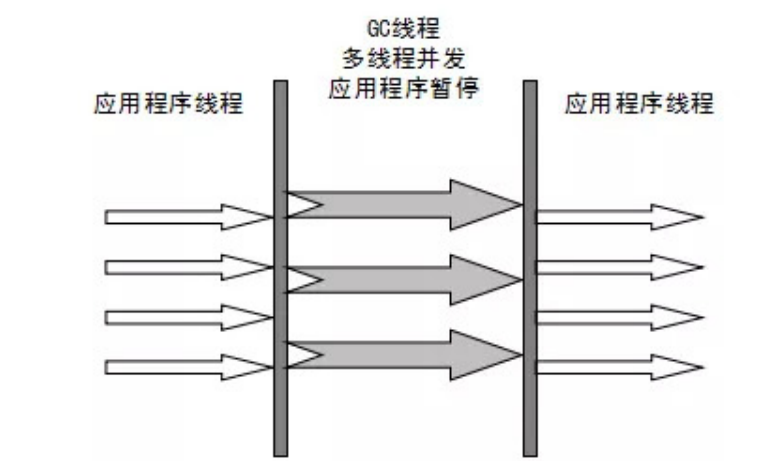
#### ParNew收集器

ParNew收集器其实跟Parallel收集器很类似，区别主要在于它可以和CMS收集器配合使用。

**新生代采用复制算法，老年代采用标记-整理算法。**

它是许多运行在Server模式下的虚拟机的首要选择，除了Serial收集器外，只有它能与CMS收集器（真正意义上的并发收集器）配合工作。

**启用命令:-XX:+UseParNewGC**

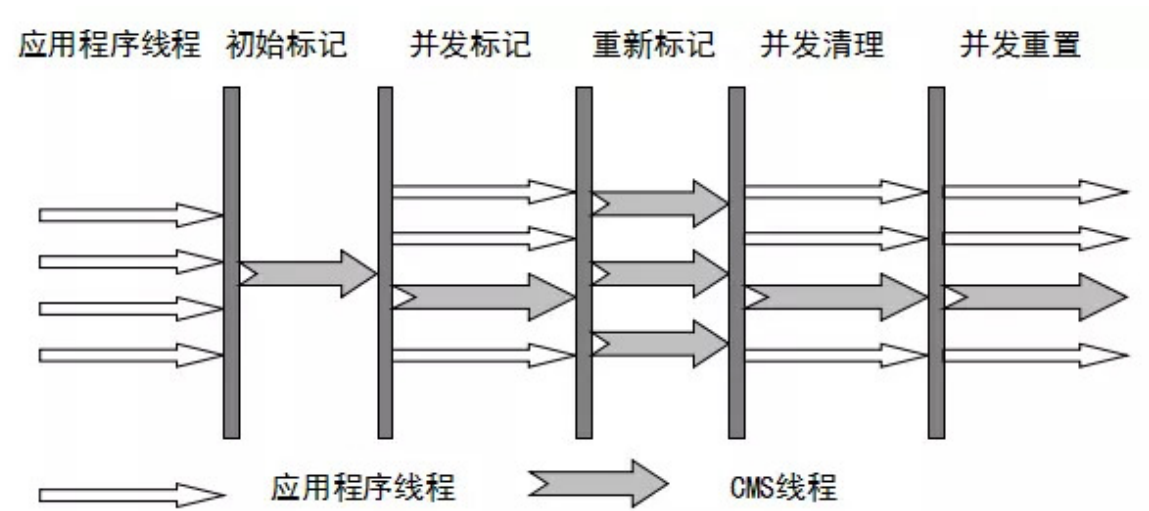
****

#### CMS收集器

**CMS（Concurrent Mark Sweep）收集器**是目标尽量减少STW的收集器。适合用于注重用户体验的应用程序中，真正实现用户线程、收集线程(基本上)并发执行的收集器。

CMS收集器使用标识-清除算法，具体包含四步：

1. 初始标记，会STW，并记录gc root能够直接引用的对象，速度很快；
2. 并发标记，收集线程从gc root直接关联的对象开始遍历整个对象图的过程，耗时长但无需暂停用户线程。由于是并发执行，会导致已标记对象但又发生了变化。
3. 重新标记，补救2的，会stw，时间比1稍长，但是远低于2。主要用到**三色标记**里的增量更新算法做重新标记。
4. 并发清理，用户线程正常开启，gc线程对未标记区域继续做清扫。新增对象会标成黑色，不做处理。
5. 并发重置，重置本次gc标记数据。



CMS优点：

**并发高，低停顿。**

CMS缺点：

* 对cpu资源敏感(容易跟服务抢资源)
* 无法处理浮动垃圾(并发标记、并发清理阶段产生的垃圾需要下次gc处理)
* 标记-清除算法会导致内存碎片化。

当然通过参数**-XX:+UseCMSCompactAtFullCollection**可以让jvm在执行完标记清除后再做整理。

* 执行不确定性。上次回收没有完成，又触发gc会导致。特别容易发生在并发标记、并发清理的阶段，也就是"concurrent mode failure"，此时会进入stop the world，用serial old垃圾收集器来回收。

**备注：CMS的相关核心参数**

1. **-XX:+UseConcMarkSweepGC**：启用cms

2. **-XX:ConcGCThreads**：并发的GC线程数

3. **-XX:+UseCMSCompactAtFullCollection**：FullGC之后做压缩整理（减少碎片）

4. **-XX:CMSFullGCsBeforeCompaction**：多少次FullGC之后压缩一次，默认是0，代表每次FullGC后都会压缩一次

5. **-XX:CMSInitiatingOccupancyFraction**: 当老年代使用达到该比例时会触发FullGC（默认是92，这是百分比）

6. **-XX:+UseCMSInitiatingOccupancyOnly**：只使用设定的回收阈值(**-XX:CMSInitiatingOccupancyFraction**设定的值)，如果不指定，JVM仅在第一次使用设定值，后续则会自动调整

7. **-XX:+CMSScavengeBeforeRemark**：在CMS GC前启动一次minor gc，目的在于减少老年代对年轻代的引用，降低CMS GC的标记阶段时的开销，一般CMS的GC耗时 80%都在标记阶段

8. **-XX:+CMSParallellnitialMarkEnabled**：表示在初始标记的时候多线程执行，缩短STW

9. **-XX:+CMSParallelRemarkEnabled**：在重新标记的时候多线程执行，缩短STW;

### 如何选择收集器

* 优先调整堆的大小让服务器自己来选择
* 如果内存小于100M，使用串行收集器
* 如果是单核，并且没有停顿时间的要求，串行或JVM自己选择
* 如果允许停顿时间超过1秒，选择并行或者JVM自己选
* 如果响应时间最重要，并且不能超过1秒，使用并发收集器
* 4G以下可以用parallel，4-8G可以用ParNew+CMS，8G以上可以用G1，几百G以上用ZGC
* 备注：

JDK 1.8默认使用 Parallel(年轻代和老年代都是)

JDK 1.9默认使用 G1

### 优化实践

分享一个下单系统的优化总结。

场景是下单，对象不会长期存活，但是要求少STW,用户体验好，所以：

|  |  |
| --- | --- |
| 参数 | 说明 |
| -Xms3072M | Jvm最小堆内存 |
| -Xmx3072M | Jvm最大堆内存 |
| -Xmn2048M | 新生代内存 |
| -Xss1M | 线程栈内存 |
| -XX:MetaspaceSize=256M | 元空间内存 |
| -XX:MaxMetaspaceSize=256M | 元空间最大内存 |
| -XX:SurvivorRatio=8 | Eden占了新生代80%，s0、s1共占20% |
| -XX:MaxTenuringThreshold=5 | 设置垃圾最大年龄，默认15次 |
| -XX:PretenureSizeThreshold=1M | 超过1m的对象直接进入老年代 |
| -XX:+UseParNewGC | 新生代使用ParNew收集器 |
| -XX:+UseConcMarkSweepGC | 老年代使用CMS收集器 |
| -XX:CMSInitiatingOccupancyFraction=92 | 当老年代使用达到该比例时会触发FullGC（默认是92，这是百分比） |
| -XX:+UseCMSCompactAtFullCollection | CMS在FullGC之后做压缩整理（减少碎片） |
| -XX:CMSFullGCsBeforeCompaction=0 | 多少次FullGC之后压缩一次，默认是0 |

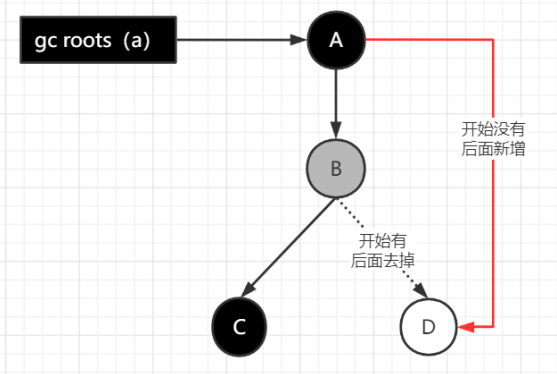
### 垃圾回收底层算法

#### 三色标记法

在cms这些并发标记的收集器工作过程中，由于用户线程也在工作可能导致对象间引用变化，多标、漏标的情况常常发生。

Jvm引入三色标记法，把GCroots可达性分析遍历对象过程中遇到的对象按照“是否访问过”标成三种颜色：

* 黑色：表示独享已经被垃圾收集器访问过，且他所有引用已经扫描过。黑色对象安全存活，其他对象引用此黑色对象，也无需再扫面。黑色对象不可以直接指向白色对象。
* 灰色：表示此对象已经被垃圾收集器访问过，但此对象至少存在一个没扫描对象。
* 白色：表示此对象没有被垃圾收集器访问过。如果访问过还是白色，表示此对象是不可达。



#### 多标-浮动垃圾

并发标记过程中，由于方法执行完毕导致部分局部变量(gcroot)被销毁，这个gcroot引用的对象之前又被扫描并标记成非垃圾对象，那么本轮gc就不会回收此部分内存，而留到下一次。**本该回收但是没有回收的内存就是浮动垃圾。**

**针对并发标记(并发清理)开始后产生的对象，通常标成黑色，等下一轮再回收。**这些对象有可能也是垃圾也算浮动垃圾。

#### 漏标-读写屏障

漏标会导致被引用对象当成垃圾直接清理。是严重的bug需要通过以下两种方式解决：

* **增量更新**

当黑色对象插入指向白色的对象的引用关系时，并发扫描后，黑色对象将被标成灰色，重新执行扫描。

* **原始快照**

灰色对象要删除执行白色对象A的引用关系时，A将被记录下来，等并发扫描后，A作为灰色根重新扫描一次，这样能扫描出来的白色对象都标成黑色。

目标是将这些对象能在本轮gc中存活然后下一轮gc重新扫描，这些对象也可能是浮动垃圾。

#### 读与写屏障

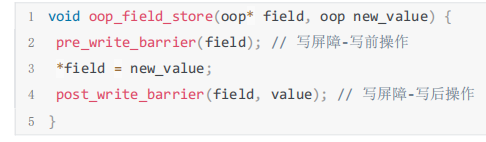
虚拟机的引用关系记录的插入和删除都是通过写屏障实现。

* **写屏障**

给某个对象的成员**变量赋值**时，其底层代码大概长这样：



所谓的写屏障，其实就是指在赋值操作前后，加入一些处理（可以参考AOP的概念）：



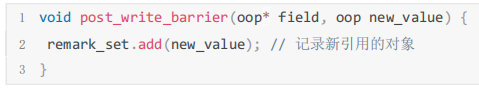
* **写屏障实现SATB**

当对象B的成员变量的引用发生变化时，比如引用消失（a.b.d = null），我们可以利用写屏障，将B原来成员变量的引用对象D记录下来：



* **写屏障实现增量更新**

当对象A的成员变量的引用发生变化时，比如新增引用（a.d = d），我们可以利用写屏障，将A新的成员变量引用对象D记录下来：



* **读屏障**

oop oop\_field\_load(oop\* field) {

pre\_load\_barrier(field); // 读屏障‐读取前操作

return \*field;

}

读屏障是直接针对第一步：D d = a.b.d，当读取成员变量时，一律记录下来



* **综述**

现代追踪式（可达性分析）的垃圾回收器几乎都借鉴了三色标记的算法思想，尽管实现的方式不尽相同：比如白色/黑色集合一般都不会出现（但是有其他体现颜色的地方）、灰色集合可以通过栈/队列/缓存日志等方式进行实现、遍历方式可以是广度/深度遍历等等。

对于读写屏障，以Java HotSpot VM为例，其并发标记时对漏标的处理方案如下：

* CMS：写屏障 + 增量更新
* G1，Shenandoah：写屏障 + SATB
* ZGC：读屏障

工程实现中，读写屏障还有其他功能，比如写屏障可以用于记录跨代/区引用的变化，读屏障可以用于支持移动对象的并发执行等。功能之外，还有性能的考虑，所以对于选择哪种，每款垃圾回收器都有自己的想法。

为什么G1用原始快照SATB？CMS用增量更新？

我的理解：SATB相对增量更新效率会高(当然SATB可能造成更多的浮动垃圾)，因为不需要在重新标记阶段再次深度扫描被删除引用对象，而CMS对增量引用的根对象会做深度扫描，G1因为很多对象都位于不同的region，CMS就一块老年代区域，重新深度扫描对象的话G1的代价会比CMS高，所以G1选择SATB不深度扫描对象，只是简单标记，等到下一轮GC再深度扫描。

### 记忆集与卡表

在新生代做GCRoots可达性扫描过程中可能会碰到跨代引用的对象，这种如果又去对老年代再去扫描效率太低了。

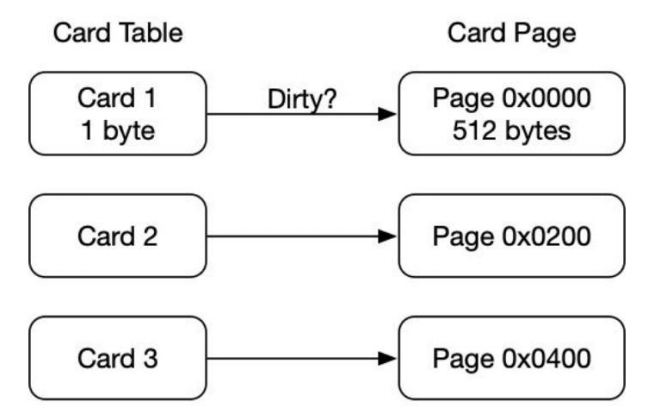
为此，在新生代可以引入记录集（Remember Set）的数据结构（记录从非收集区到收集区的指针集合），避免把整个老年代加入GCRoots扫描范围。事实上并不只是新生代、 老年代之间才有跨代引用的问题， 所有涉及部分区域收集（Partial GC） 行为的垃圾收集器， 典型的如G1、 ZGC和Shenandoah收集器， 都会面临相同的问题。

垃圾收集场景中，收集器只需通过记忆集判断出某一块非收集区域是否存在指向收集区域的指针即可，无需了解跨代引用指针的全部细节。

hotspot使用一种叫做“卡表”(cardtable)的方式实现记忆集，也是目前最常用的一种方式。关于卡表与记忆集的关系，可以类比为Java语言中HashMap与Map的关系。

卡表是使用一个字节数组实现：CARD\_TABLE[ ]，每个元素对应着其标识的内存区域一块特定大小的内存块，称为“卡页”。

hotSpot使用的卡页是2^9大小，即512字节。



一个卡页中可包含多个对象，只要有一个对象的字段存在跨代指针，其对应的卡表的元素标识就变成1，表示该元素变脏，否则为0.

GC时，只要筛选本收集区的卡表中变脏的元素加入GCRoots里。

**卡表的维护**

卡表变脏上面已经说了，但是需要知道如何让卡表变脏，即发生引用字段赋值时，如何更新卡表对应的标识为1。

Hotspot使用写屏障维护卡表状态。

### 安全点与安全区域

**安全点**就是指代码中一些特定的位置,当线程运行到这些位置时它的状态是确定的,这样JVM就可以安全的进行一些操作,比如GC等，所以GC不是想什么时候做就立即触发的，是需要等待所有线程运行到安全点后才能触发。

这些特定的安全点位置主要有以下几种:

1. 方法返回之前

2. 调用某个方法之后

3. 抛出异常的位置

4. 循环的末尾

大体实现思想是当垃圾收集需要中断线程的时候， 不直接对线程操作， 仅仅简单地设置一个标志位， 各个线程执行过程时会不停地主动去轮询这个标志， 一旦发现中断标志为真时就自己在最近的安全点上主动中断挂起。 轮询标志的地方和安全点是重合的。

**安全区域**又是什么？

Safe Point 是对正在执行的线程设定的。

如果一个线程处于 Sleep 或中断状态，它就不能响应 JVM 的中断请求，再运行到 Safe Point 上。

因此 JVM 引入了 Safe Region。

Safe Region 是指在一段代码片段中，引用关系不会发生变化。在这个区域内的任意地方开始 GC 都是安全的。