JDJDK

Memory Management

毛宝龙

2019-03-20

咚咚群号：81523801

Agenda

01 JVM Memory Layout

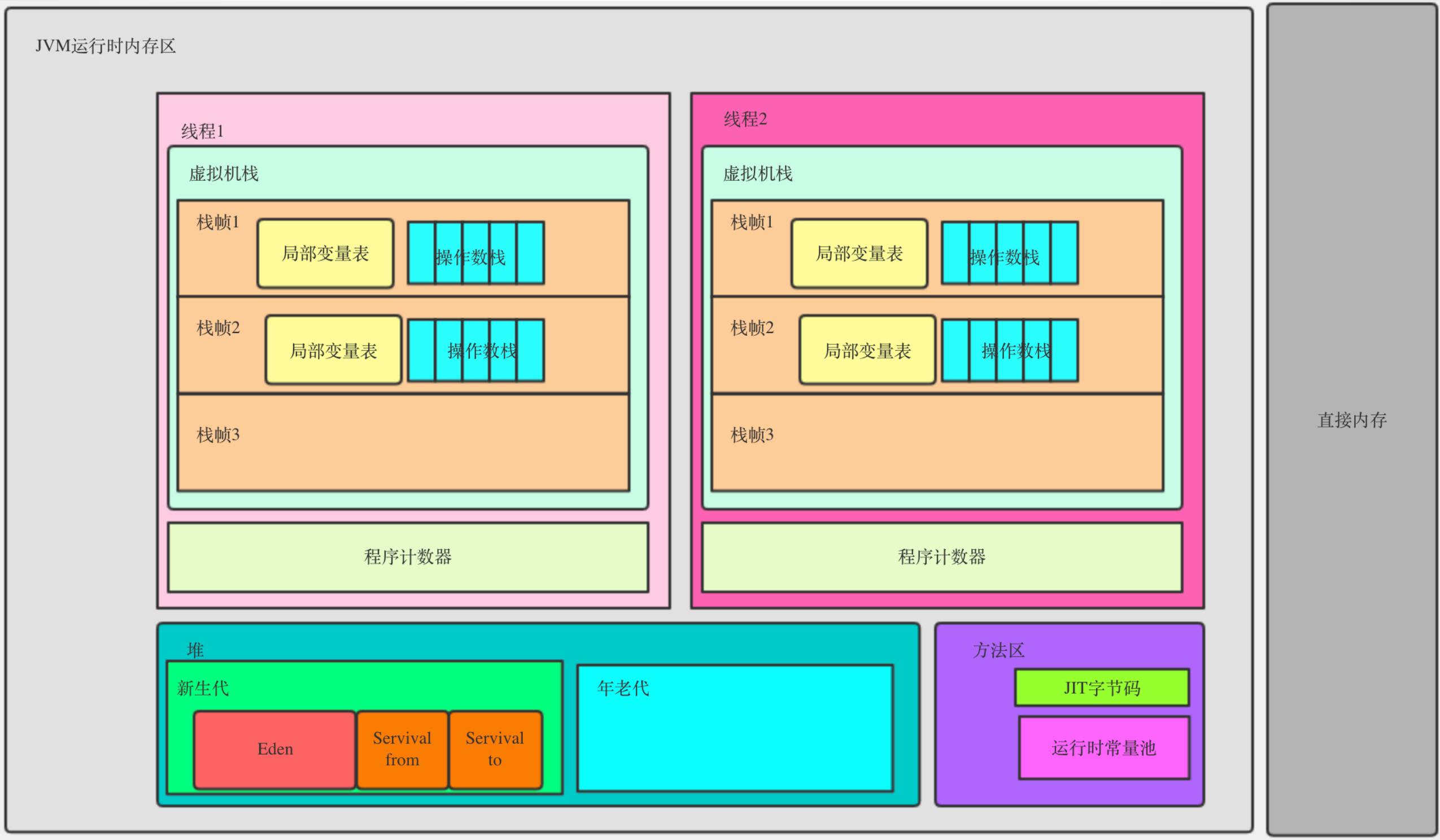
02 GarbageCollector

03 GC种类

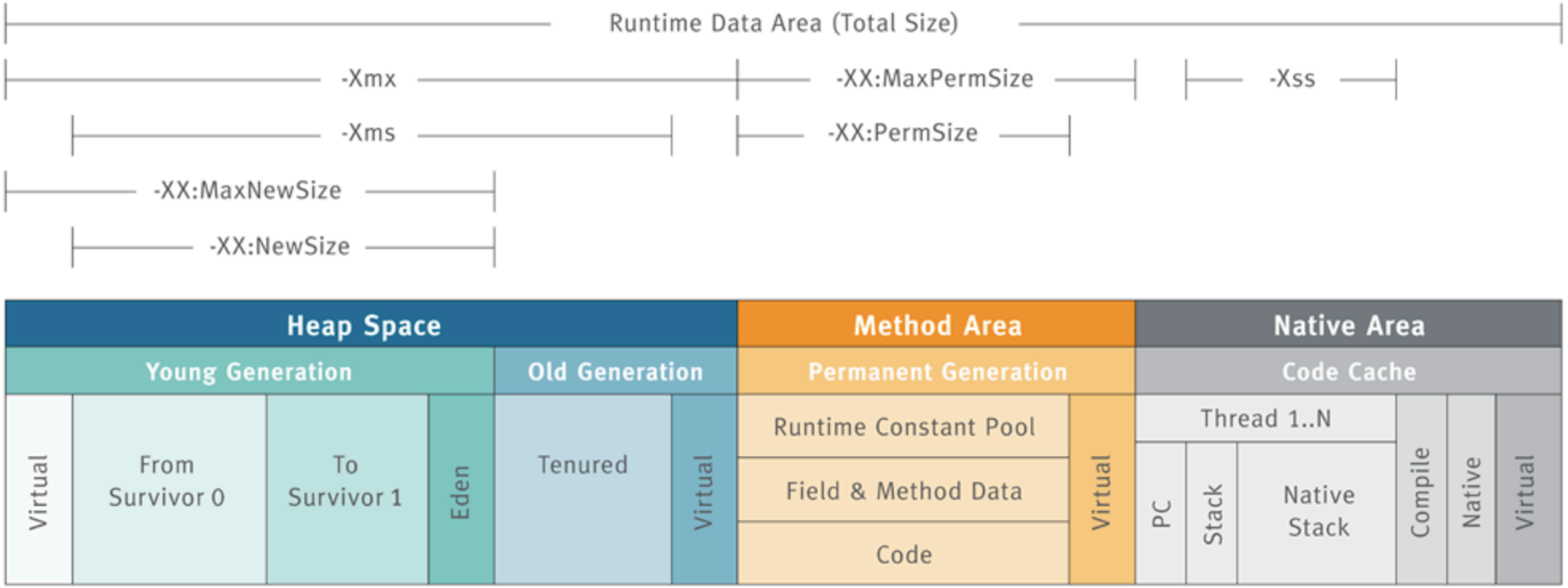
04 G1GC

JVM 内存布局

JVM内存布局


JVM运行时内存区
线程1
虚拟机栈
栈帧1
局部变量表
操作数栈
栈帧2
局部变量表
操作数栈
栈帧3
程序计数器
线程2
虚拟机栈
栈帧1
局部变量表
操作数栈
栈帧2
局部变量表
操作数栈
栈帧3
程序计数器
堆
新生代
Eden
Servival from
Servival to
年老代
方法区
JIT字节码
运行时常量池
直接内存

JVM内存布局

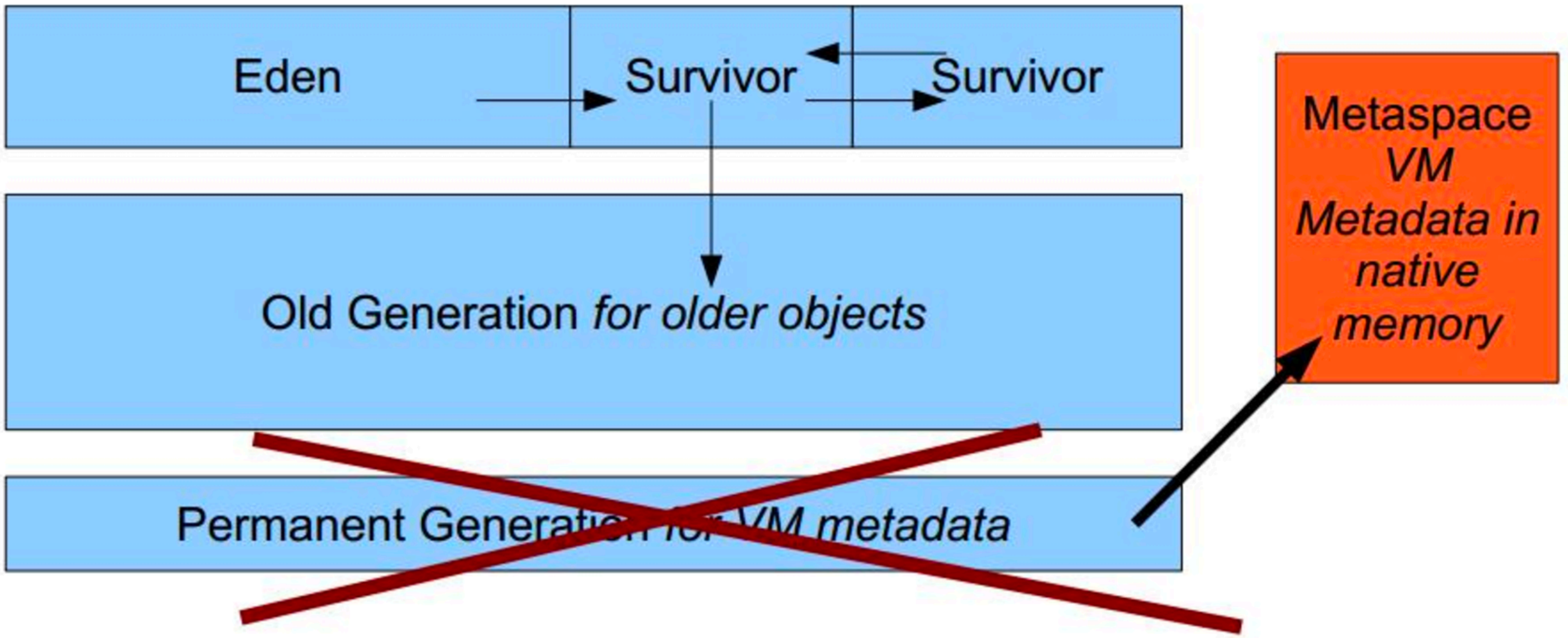


-Xmn=x 等价 -XX:NewSize= -XX:MaxNewSize=x 设置新生代大小。

**老年代空间大小= -Xmx　–　**-XX:MaxNewSize

JVM内存布局

- **Hotspot JDK8-废弃永久代（PermGen）新增元空间（Metaspace）**

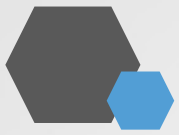


PermGen和Metaspace都是JVM规范中方法区的实现。
PermGen启动时固定大小，FGC会移动元数据信息。
MetaSpace在本地内存分配。包含Object 类元信息、静态属性、方法、常量
PermGen的字符串常量池移入Heap

2

GC

1.查找内存中使用的对象（Live objects）。把部分不使用的释放。
2.复制、清理内存，获取更多连续内存空间。

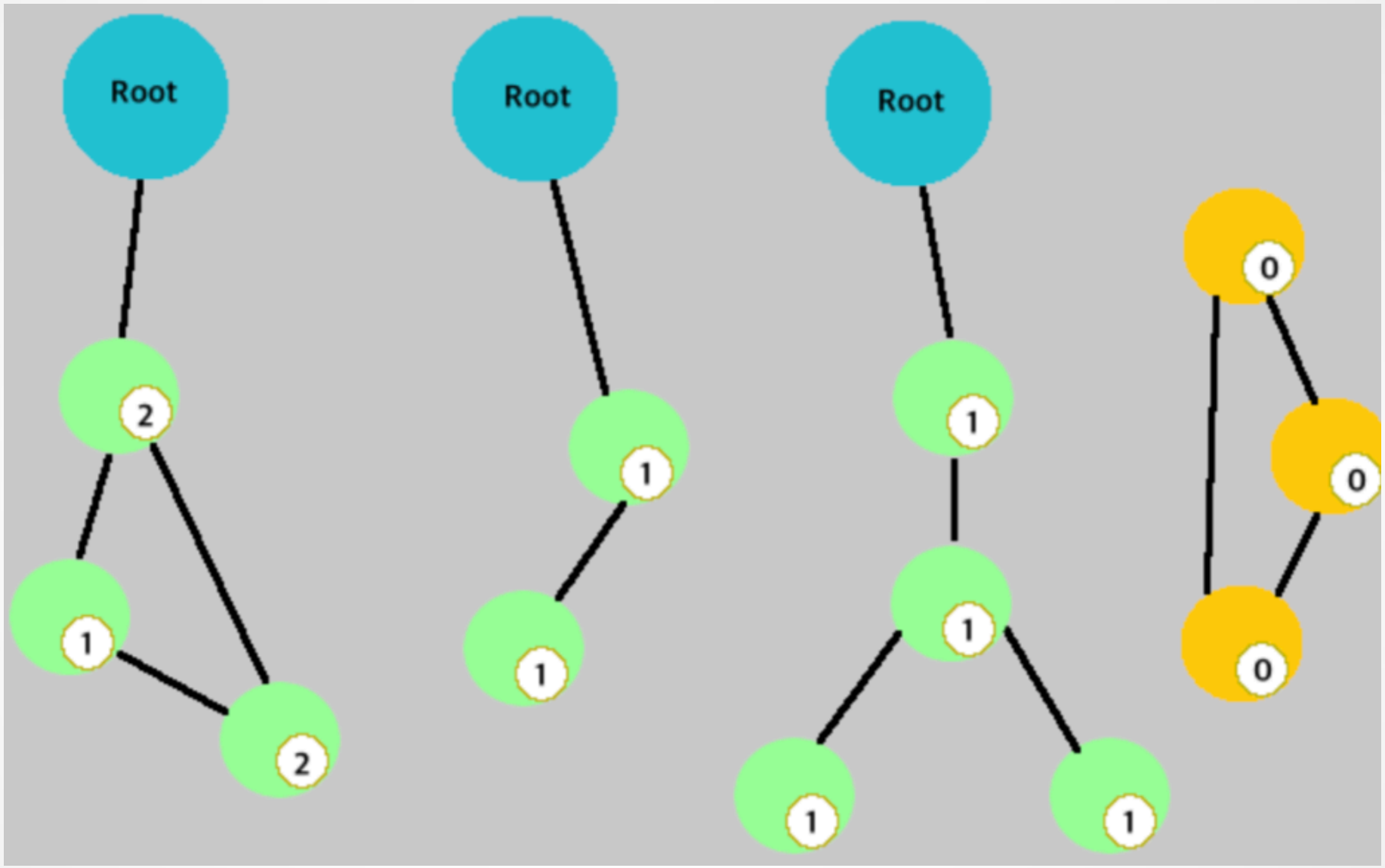


GC ROOTS

什么是GC ROOTS?

Class - 由系统类加载器(system class loader)加载的对象，这些类是不能够被回收的，他们可以以静态字段的方式保存持有其它对象。我们需要注意的一点就是，通过用户自定义的类加载器加载的类，除非相应的java.lang.Class实例以其它的某种（或多种）方式成为roots，否则它们并不是roots，.
Thread - 活着的线程
Stack Local - Java方法的local变量或参数
JNI Local - JNI方法的local变量或参数
JNI Global - 全局JNI引用
Monitor Used - 用于同步的监控对象
Held by JVM - 用于JVM特殊目的由GC保留的对象

GC ROOTS



GC ROOTS

```
public class TestGCRoots01 {
// 方法区中的静态变量引用的对象作为GCRoots
        private static  int _10MB = 10 * 1024 * 1024;
// 常量引用对象作为GC Roots
        private static final TestGCRoots03 t = new TestGCRoots03(8 * _10MB)
        private byte[] memory = new byte[8 * _10MB];

        public static void main(String[] args) {
                method01();
        }

        public static void method01() {
// 虚拟机栈（栈帧中的局部变量）中引用的对象作为GCRoots
                TestGCRoots01 t = new TestGCRoots01();
        }
}
```

- **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（但不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集程序运行于另一个CPU上。
- **新生代GC（Minor GC）**：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。
- **老年代GC（Major GC）**：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。
- Full GC: STW
- **吞吐量**：吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）。
虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。
- **STW = Stop The World.** 可达性分析必须在一个能确保一致性的内存快照中进行（ 标记阶段）

循环引用

- 引用计数算法(Reference Counting)
- 标记-清除算法(Mark-Sweep)

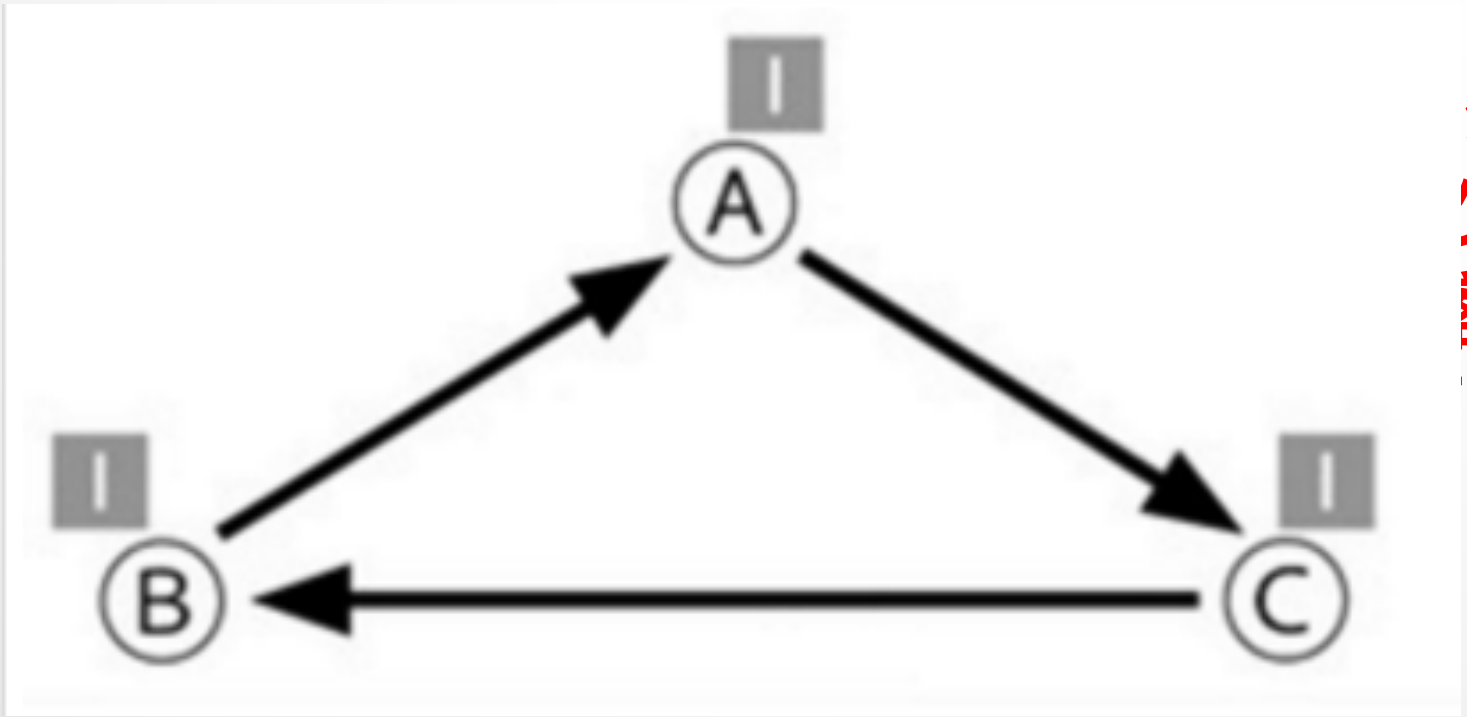
空间碎片

- 复制算法(Copying)

Survivor：S0，S1

老年代：让存活对象都向一端移动

- 标记-整理(Mark-Compact)
- 分代收集算法(Generational Collection)



死去，只有少量存活

少量存活的对象

额外空间做分配担保

种算法

GC hotspot种类

GC 种类

名称		区域	算法	适用情况
Serial	串行	年轻代	复制	单CPU（或CPU较少）、小型客户端应用
Parallel Scavenge	并行	年轻代	复制	多CPU、吞吐量优先（后台处理、科学计算）
ParNew Parallel New Generation	并行	年轻代	复制	多CPU、响应优先（web服务器等）
CMS	并发	年老代	标记-清除	响应优先（web服务器等）
Serial Old	串行	年老代	标记-整理	单CPU、小型客户端应用
Parallel Old	并行	年老代	标记-整理	多CPU、吞吐量优先（后台处理、科学计算）
G1		老少通吃		支持很大的堆，高吞吐量 支持多CPU和垃圾回收线程

GC 种类

参数	说明
-XX:+UseSerialGC	相当于”Serial” + “SerialOld”
-XX:+UseParallelGC	相当于” Parallel Scavenge” + “SerialOld”，也就是说，在young generation中是多线程处理，但是在tenured generation中则是单线程；
-XX:+UseParallelOldGC	相当于” Parallel Scavenge” + “ParallelOld”，都是多线程并行处理；
-XX:+UseConcMarkSweepGC	相当于“ParNew” + “CMS” + “Serial Old”，即在young generation中采用ParNew，多线程处理；在tenured generation中使用CMS，以求得到最低的暂停时间，但是，采用CMS有可能出现”Concurrent Mode Failure”，如果出现了，就只能采用”SerialOld”模式了；
-XX:+UseG1GC	

G1GC

G1是目前技术发展的**最前沿可产品化的**成果，HotSpot开发团队JDK9作为默认GC，替换掉**JDK1.5**中发布的CMS收集器。

Garbage First Garbage Collector

首先收集尽可能多的垃圾(Garbage First)

适合多处理器和大容量内存的服务端环境中

Region:

所有Region大小相同，最大32M（openjdk）,2的指数。

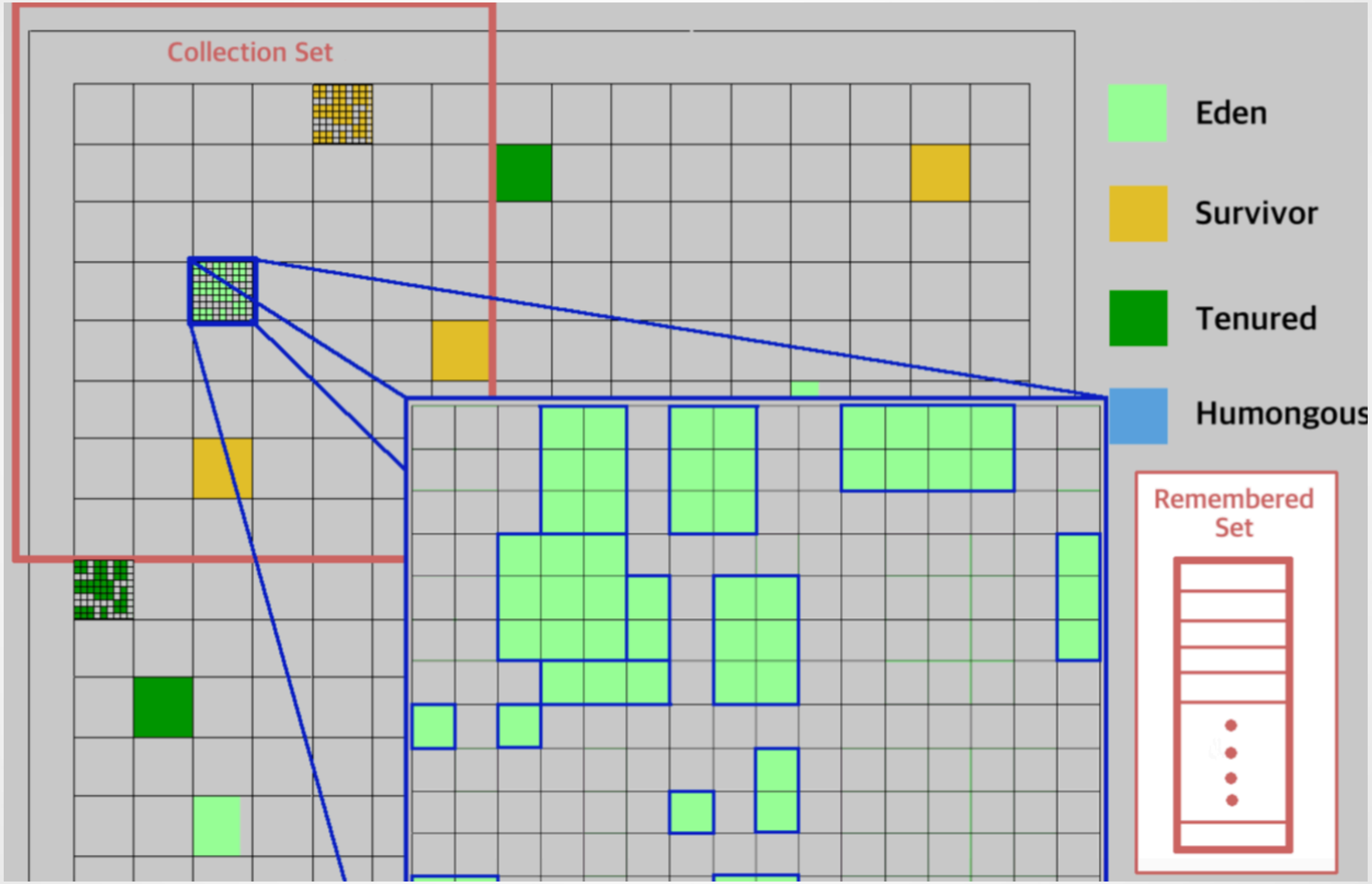
每个Region是独立的

老年代和年轻代都由若干个Region组成

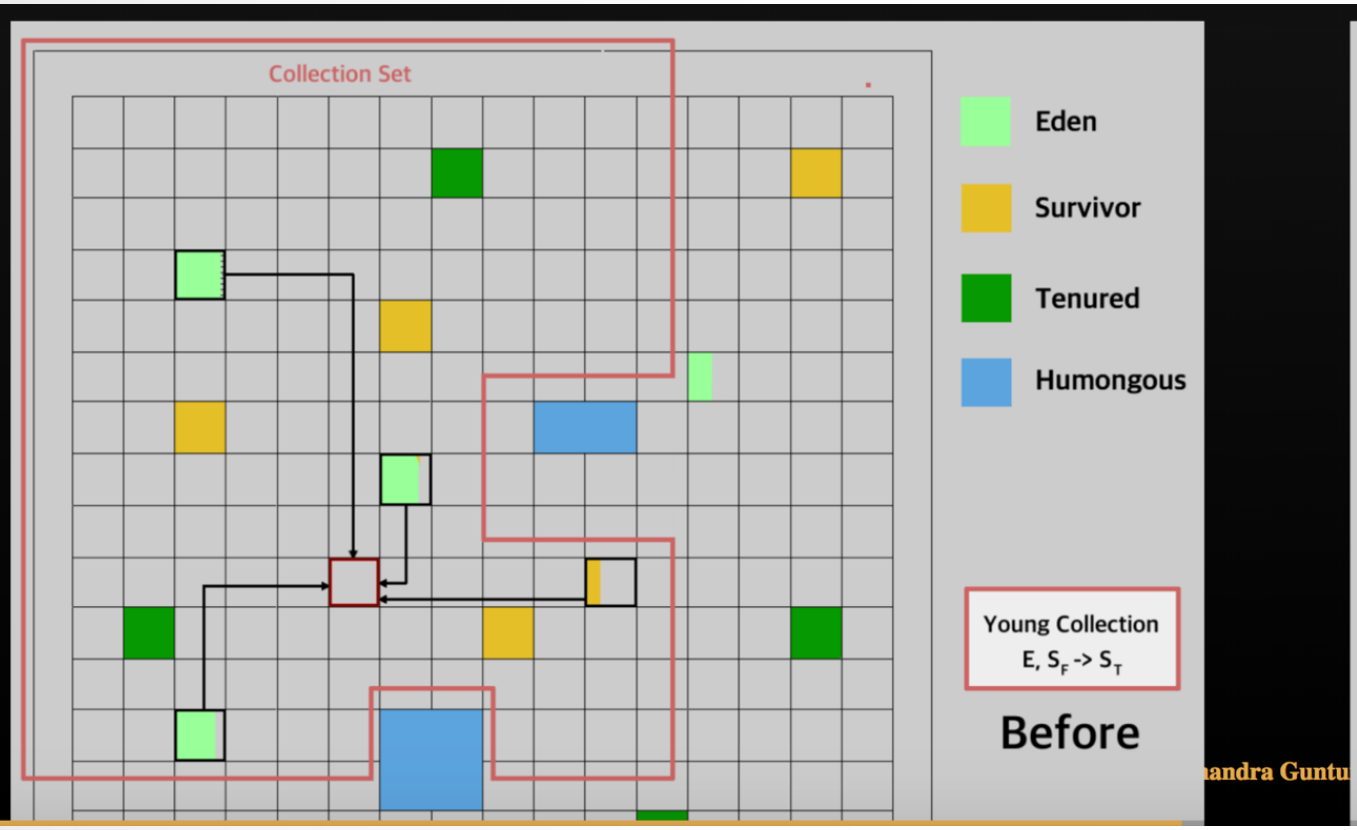
G1GC —— region size

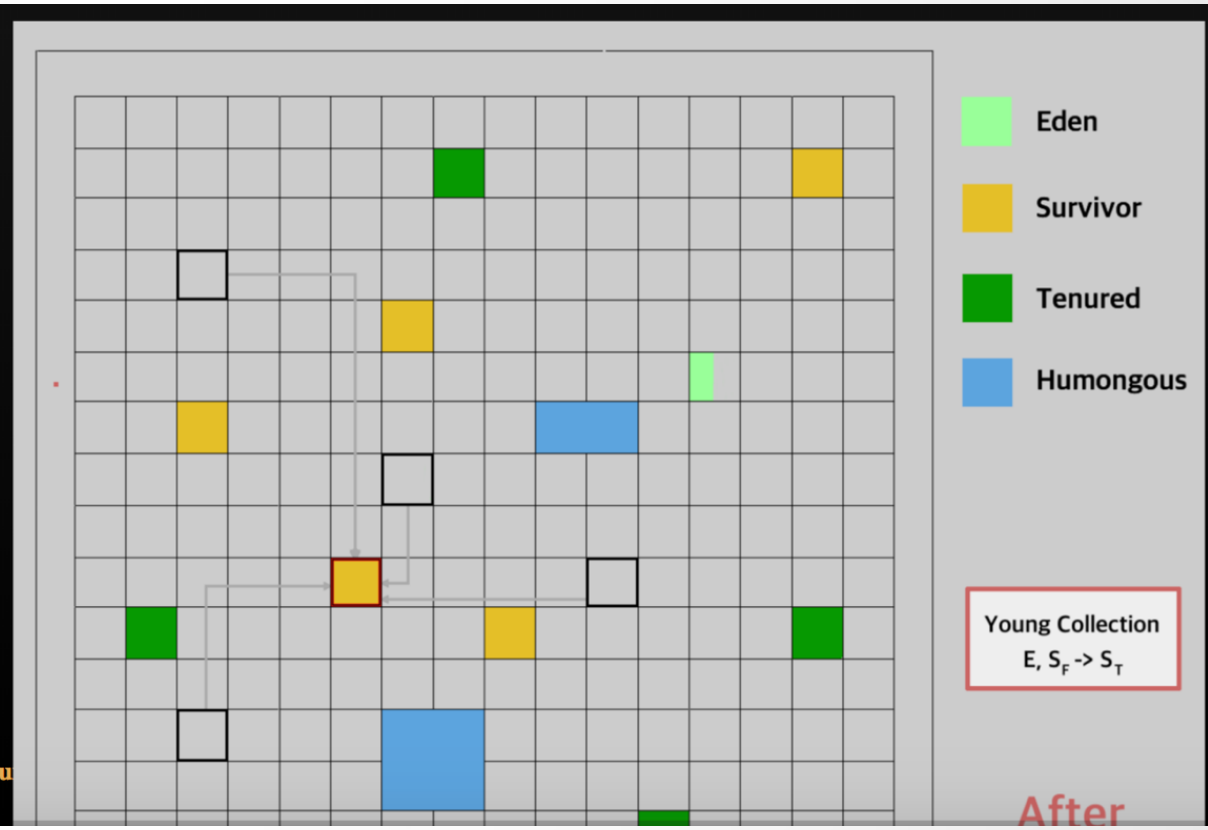
```
static const size_t MIN_REGION_SIZE = 1024 * 1024;
static const size_t MAX_REGION_SIZE = 32 * 1024 * 1024;
static const size_t TARGET_REGION_NUMBER = 2048;

size_t region_size = G1HeapRegionSize;
if (FLAG_IS_DEFAULT(G1HeapRegionSize)) {
  size_t average_heap_size = (initial_heap_size + max_heap_size) / 2;
  region_size = MAX2(average_heap_size / HeapRegionBounds::target_number(),
                     HeapRegionBounds::min_size());
}
```

```
int region_size_log = log2_long((jlong) region_size);
// Recalculate the region size to make sure it's a power of
// 2. This means that region_size is the largest power of 2 that's
// <= what we've calculated so far.
region_size = ((size_t)1 << region_size_log);

// Now make sure that we don't go over or under our limits.
if (region_size < HeapRegionBounds::min_size()) {
  region_size = HeapRegionBounds::min_size();
} else if (region_size > HeapRegionBounds::max_size()) {
  region_size = HeapRegionBounds::max_size();
}
```


Collection Set
Eden
Survivor
Tenured
Humongous
Remembered Set

JD.京东.COM


Collection Set
Eden
Survivor
Tenured
Humongous
Young Collection
E, S_F -> S_T
Before


Eden
Survivor
Tenured
Humongous
Young Collection
E, S_F -> S_T
After

- **Remembered Set** 简称 Rset。 points-into结构
  - 跟踪对象跨带引用（哪个Region引用了我的对象），YGC时，找到Y Region里边的live objects。
  - 每个region都有一个Rset。
  - Rset支持在单一区进行并行和独立收集
- **Collection Set** 简称 Cset
  - 待清除的Region集合
  - GC期间清除
- **Humongous** 特殊的Old。 当新建对象大小超过Region大小一半时，直接在新的一个或多个连续Region中分配，并标记为H。

RSet 作用：

在做**YGC**(Minor GC)的时候，只需要选定young generation region的RSet作为根集，这些RSet记录了**old->young**的跨代引用，避免了扫描整个old generation。

而**mixed gc**的时候，old generation中记录了old->old的RSet，young->old的引用由扫描全部young generation region得到，这样也不用扫描全部old generation region。

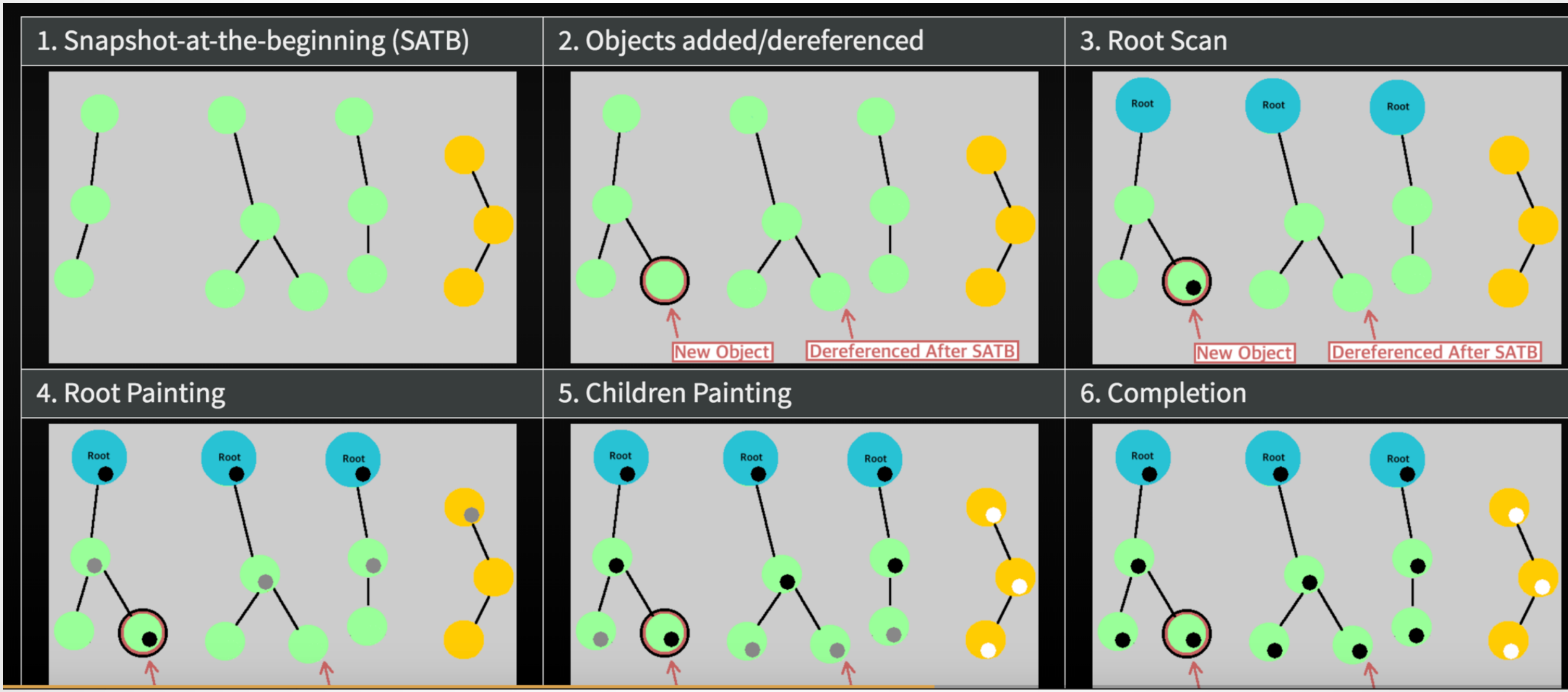
所以RSet的引入大大减少了GC的工作量

CSet 作用：

- 在任意一次收集暂停中，CSet**所有分区**都会被释放
- 年轻代收集CSet**只容纳年轻代**分区
- 而混合收集会通过启发式算法，在老年代候选回收分区中，筛选出**回收收益最高的分区**添加到CSet中。
- 候选老年代分区的CSet准入条件，可以通过活跃度阈值-XX:G1MixedGCLiveThresholdPercent(默认85%)进行设置，从而拦截那些回收开销巨大的对象；同时，每次混合收集可以包含候选老年代分区，可根据CSet对堆的总大小占比-XX:G1OldCSetRegionThresholdPercent(默认10%)设置数量上限。

JD.京东.COM

Gray = To check, **Black** = Live, **White** = Dead



Old gen收集步骤：

Initial mark (stop_the_world事件)	这是一个stop_the_world的过程，是随着年轻代GC做的，标记survivor区域(根区域),这些区域可能含有对老年代对象的引用。
Root region scan	扫描survivor区域中对老年代的引用，这个过程和应用程序一起执行的，这个阶段必须在年轻代GC发生之前完成。
Concurrent marking	查找整个堆中存活的对象，这也是和应用程序一起执行的。这个阶段可以被年轻代的垃圾收集打断。
Remark (stop-the-world事件)	完成堆内存活对象的标记。使用了一个叫开始前快照snapshot-at-the-beginning (SATB)的算法，这个会比CMS collector使用的算法快。
Cleanup(stop-the-world事件，并且是并发的)	•对存活的对象和完全空的区域进行统计(stop-the-world) •刷新Remembered Sets(stop-the-world) •重置空的区域，把他们放到free列表(并发)(译者注：大体意思就是统计下哪些区域又空了，可以拿去重新分配了)
Copy (stop-the-world事件)	这个stop-the-world的阶段是来移动和复制存活对象到一个未被使用的区域，这个可以是年轻代区域,打日志的话就标记为 [GC pause (young)]。或者老年代和年轻代都用到了，打日志就会标记为[GC Pause (mixed)]。

g1 为什么能建立可预测的停顿时间模型?

- Region大小相同，每个region 的Pause time大致相同。
- G1并不会等内存耗尽，而是在内部采用了**启发式算法**，在老年代找出具有**高收集收益的分区**进行收集。
- G1可以根据用户设置的**暂停时间目标**自动调整年轻代和总堆大小，暂停目标越短年轻代空间越小、总空间就越大

G1GC —— 常用参数

- -XX:G1HeapRegionSize=32m。值是 2 的幂，[1M,32M] 。支持的Heap总大小官方说32m*2048=64G。但实验可以-Xmx180g没问题，openjdk代码也没有region个数的限制。
- -XX:MaxGCPauseMillis=200设置并行收集最大暂停时间
- 避免使用 -Xmn 选项或 -XX:NewRatio 等其他相关选项显式设置年轻代大小，因为固定年轻代的大小会覆盖暂停时间目标
- -XX:ParallelGCThreads=n. 设置 STW 工作线程数的值。将 n 的值设置为逻辑处理器的数量。n 的值与逻辑处理器的数量相同，最多为 8。
- -XX:ConcGCThreads=n .设置并行标记的线程数。默认将 n 设置为并行垃圾回收线程数 (ParallelGCThreads+2) /4。
- -XX:G1PrintRegionLivenessInfo -XX:+UnlockDiagnosticVMOptions

参考

- 深入理解 Java G1 垃圾收集器

http://blog.jobbole.com/109170/

- WebLogic Server 性能研讨会 Garbage First — G1

https://wenku.baidu.com/view/5a5e6e76f01dc281e53af0aa.html

- G1垃圾收集器入门
- **垃圾优先型垃圾回收器调优**

http://www.oracle.com/technetwork/cn/articles/java/g1gc-1984535-zhs.html

- openjdk

http://hg.openjdk.java.net/jdk8/jdk8/hotspot/

- **Java Hotspot G1 GC的一些关键技术**

https://tech.meituan.com/g1.html

- Understanding G1 GC Logs

https://blogs.oracle.com/poonam/understanding-g1-gc-logs

- 深入理解JVM&G1GC
- GARBAGE COLLECTION - THE JOURNEY UNTIL JAVA 11

https://c-guntur.github.io/java9-gc/#/

- **通过源码学习G1GC —— Pause Young (G1 Evacuation Pause)**

https://www.jianshu.com/p/455361403079

- **自制（OpenJDK）垃圾收集器**

https://www.jianshu.com/p/46105863ed9d

通过源码学习G1GC —— Pause Initial Mark (G1 Evacuation Pause)

https://www.jianshu.com/p/0823d3975628

THANKS

咚咚

81523801