

# JVM原理与实战

# JVM类加载流程和内存结构

```
【Java源文件】 public class Student {  
    private String name;  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public static void main(String[] args) {  
        Student student1 = new Student();  
        student1.setName("muse");  
    }  
}
```

类编译器  
(将java源码编译为class文件)

```
【Class文件】 cafe babe 0000 0031 0022 0a00  
0700 1c09  
0003 001d 0700 1e0a 0003 001c 0800 1f0a  
0003 0020 0700 2101 0004 6e61 6d65 0100  
124c 6a61 7661 2f6c 616e 672f 5374 7269  
6e67 3b01 0006 3c69 6e69 743e 0100 0328  
2956 0100 0443 6f64 6501 000f 4c69 6e65  
4e75 6d62 6572 5461 626c 6501 0012 4c6f  
... ..
```

类加载器ClassLoader  
(将class文件加载到JVM)

加载

验证

准备

解析

初始化

【线程私有】

程序计数器

本地方法栈

管理native方法的执行

虚拟机栈

栈帧  
(局部变量表、操作数栈、  
动态链接、方法出口)

Student student

内存管理

JVM内存管理

【线程公有】

方法区

被jvm加载的类  
信息

常量

即时编译器编  
译后的代码

静态变量

java堆

Student实例对象

本地内存

JVM垃圾回收

Eden  
8/10

S0  
1/10

S1  
1/10

新生代 1/3

老年代 2/3

堆内存 = Eden内存 + S0/S1内存 + 老年代内存

# 栈帧操作

```
public class StackFrameTest {  
    public static void main(String[] args) {  
        StackFrameTest stackFrameTest = new StackFrameTest();  
        stackFrameTest.method1(1);  
    }  
    public void method1(int i) {  
        int num1 = 2;  
        System.out.println("method1 i=" + i);  
        method2(i, num1);  
    }  
    public void method2(int i, int j) {  
        int num2 = 3;  
        System.out.println("method2 i=" + i + ",j=" + j);  
        method3(i, j, num2);  
    }  
    public void method3(int i, int j, int x) {  
        int num3 = 4;  
        System.out.println("method3 i=" + i + ",j=" + j + ",x=" + x);  
        method4(i, j, x, num3);  
    }  
    public void method4(int i, int j, int x, int y) {  
        System.out.println("method4 i=" + i + ",j=" + j + ",x=" + x + ",y=" + y);  
    }  
}
```

方法开始执行，进行入栈；  
方法执行的完毕，进行出栈；



# JVM垃圾回收算法

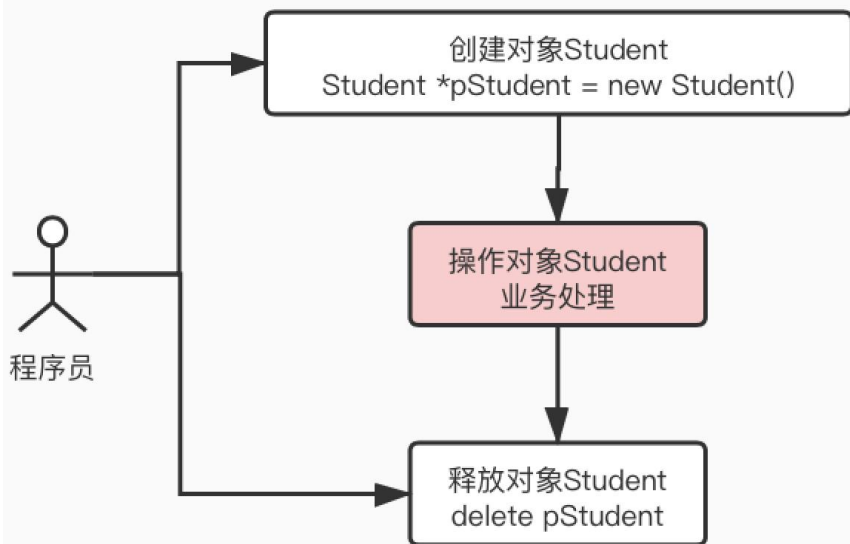
## 什么是垃圾回收

GC: 垃圾回收, 即: Garbage Collection。

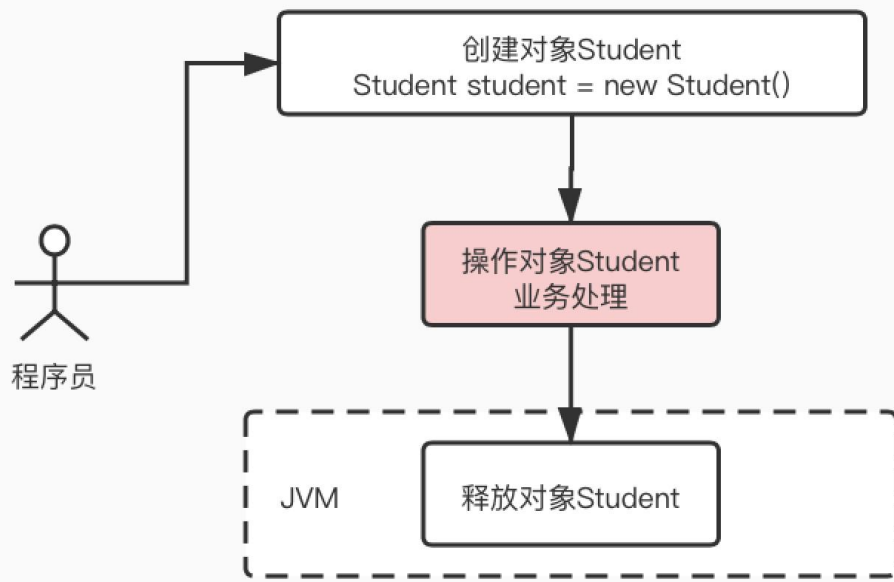
垃圾: 特指存在于内存中的、不会再被使用的对象。

回收: 清除内存中的“垃圾”对象。

C/C++



JAVA



# 主要的垃圾回收算法



## 引用计数法

增加引用+1，失去引用-1

## 复制算法

为了解决标记清除算法效率低的问题。  
该算法效率高，并且没有内存碎片，但是只能使用一半的系统内存。适用于新生代。

## 分代算法

将内存区间根据对象的生命周期分为两块，每块特点不同，使用回收算法也不同，从而提升回收效率。

## 标记清除法

但内存碎片多，对于大对象的内存分配。不连续的内存空间分配效率低于连续空间。  
是现代垃圾回收算法的思想基础。

## 标记压缩法

为了解决复制算法只能使用1/2内存的问题。  
适用于垃圾对象多的情况，适用于老年代。

## 分区算法

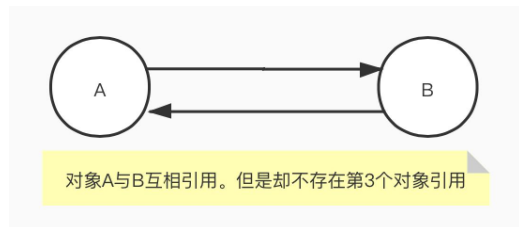
将这个堆空间划分成连续不同的小区，每个区间独立使用、独立回收。好处是：可以控制一次回收多少的小空间，避免GC时间过长，造成系统停顿。

## 引用计数法 (Reference Counting)

对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1，当引用失效时，引用计数器就减1.只要对象A的引用计数器的值为0，则对象A就不可能再被使用。

但引用计数器有两个严重问题：

(1) 无法处理循环引用的情况。



(2) 引用计数器要求在每次因引用产生和消除的时候，需要伴随一个加减法操作，对系统性能会有一定的影响。

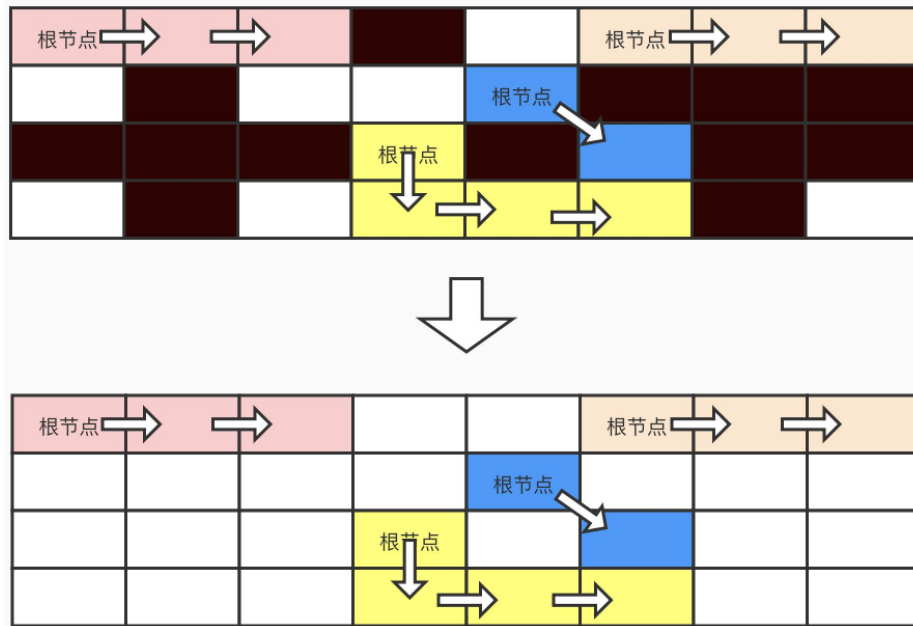
因此：JVM并未选择此算法作为垃圾回收算法。

## 标记清除法 (Mark-Sweep)

标记清除算法是现代垃圾回收算法的思想基础。

分为两个阶段：**标记**阶段和**清除**阶段。

标记清除算法产生最大的问题就是清除之后的**空间碎片**。

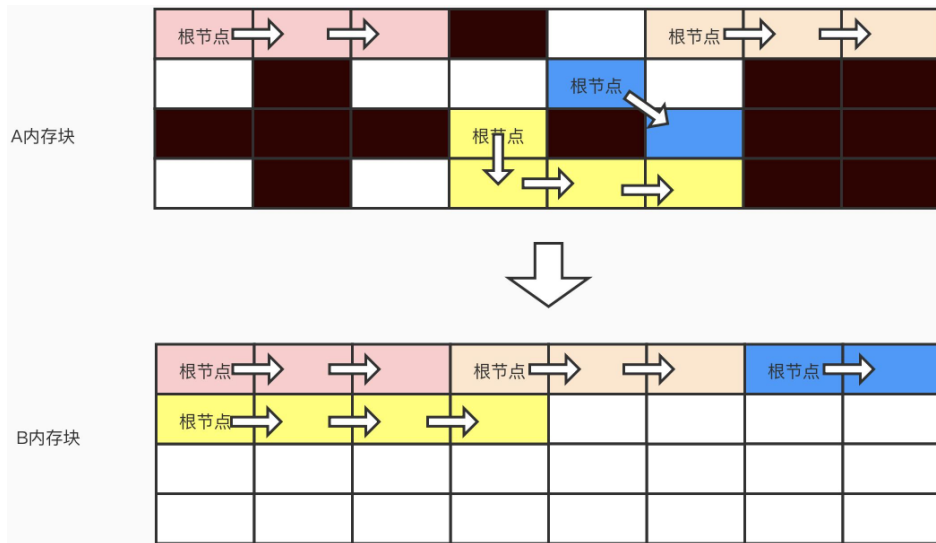




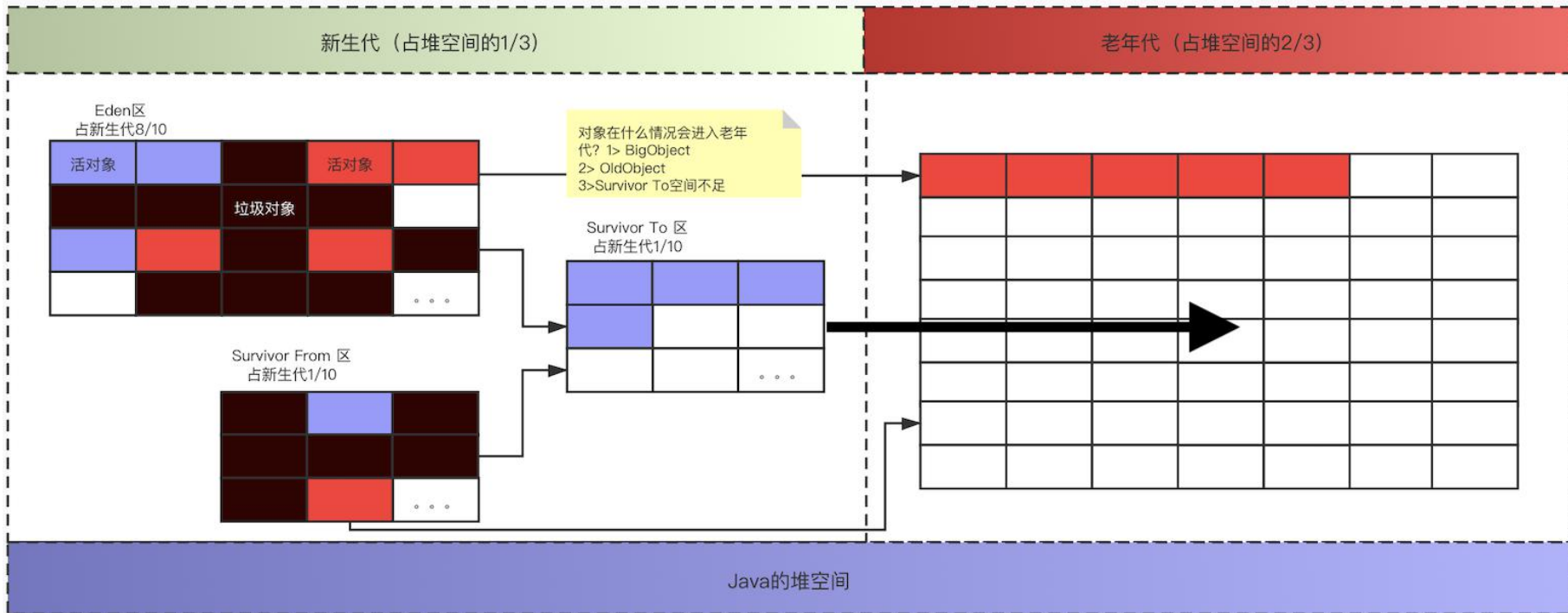
## 复制算法 (Copying)

将原有内存空间分为**两块**。每次只使用其中**一块**内存，例如：A内存，GC时将存活的对象复制到B内存中。然后清除掉A内存所有对象。开始使用B内存。

复制算法没有内存碎片，并且如果垃圾对象很多，那么这种算法效率很高。但是它的缺点是系统**内存只能使用1/2**。



因为新生代大多对象都是“朝不保夕”，所以在新生代串行GC中，使用了复制算法。

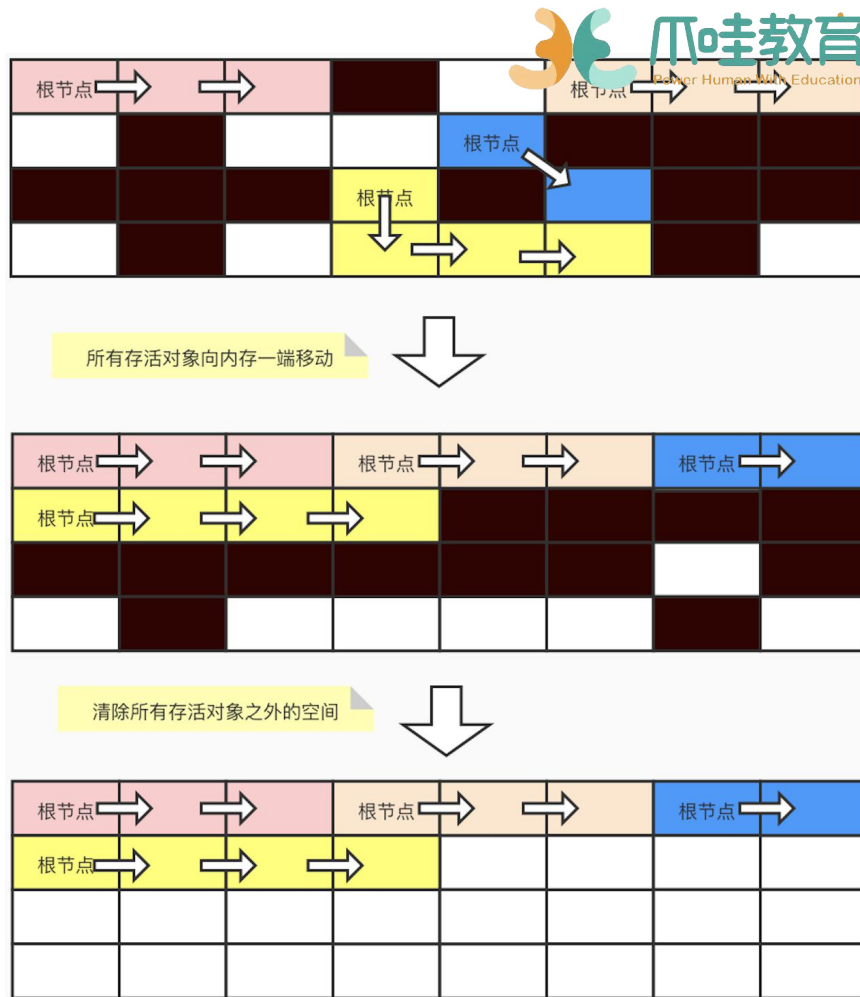


## 标记压缩法 (Mark-Compact)

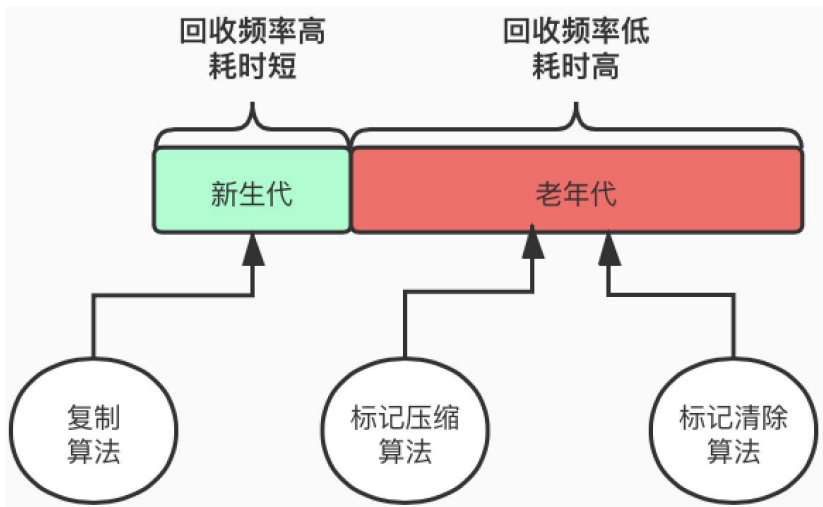
标记压缩算法是一种**老年代**的回收算法。

它首先标记存活的对象，然后将所有存活的对象压缩到内存的一端，然后在清理所有存活对象之外的空间。

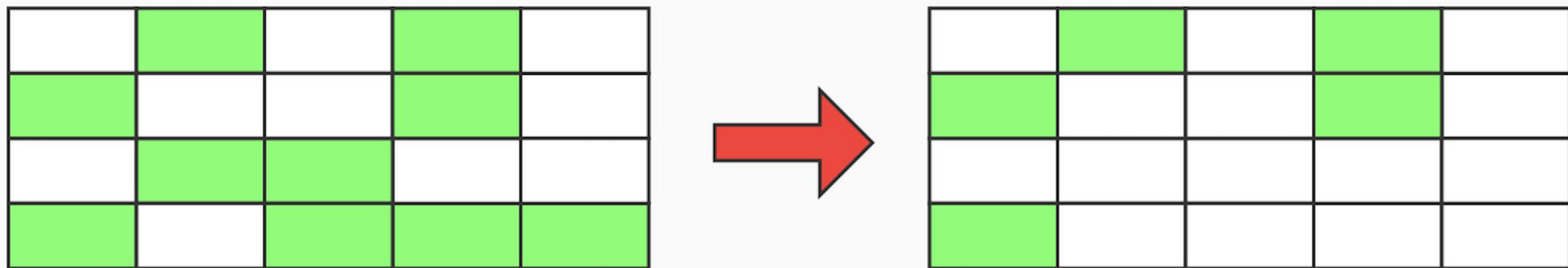
该算法不会产生内存碎片，并且也不用将内存一分为二。因此，其性价比比较高。



将堆空间划分为**新生代**和**老年代**，根据它们直接的不同特点，执行不同的回收算法，提升回收效率。



将堆空间划分成连续的不同**小区间**，每个区间**独立**使用、回收。由于当堆空间大时，一次GC的时间会非常耗时，那么可以控制每次回收多少个小区间，而不是整个堆空间，从而减少一次GC所产生的停顿。



# JVM垃圾收集器

## 串行回收器 - Serial

串行回收器的特点:

1>只使用**单线程**进行GC

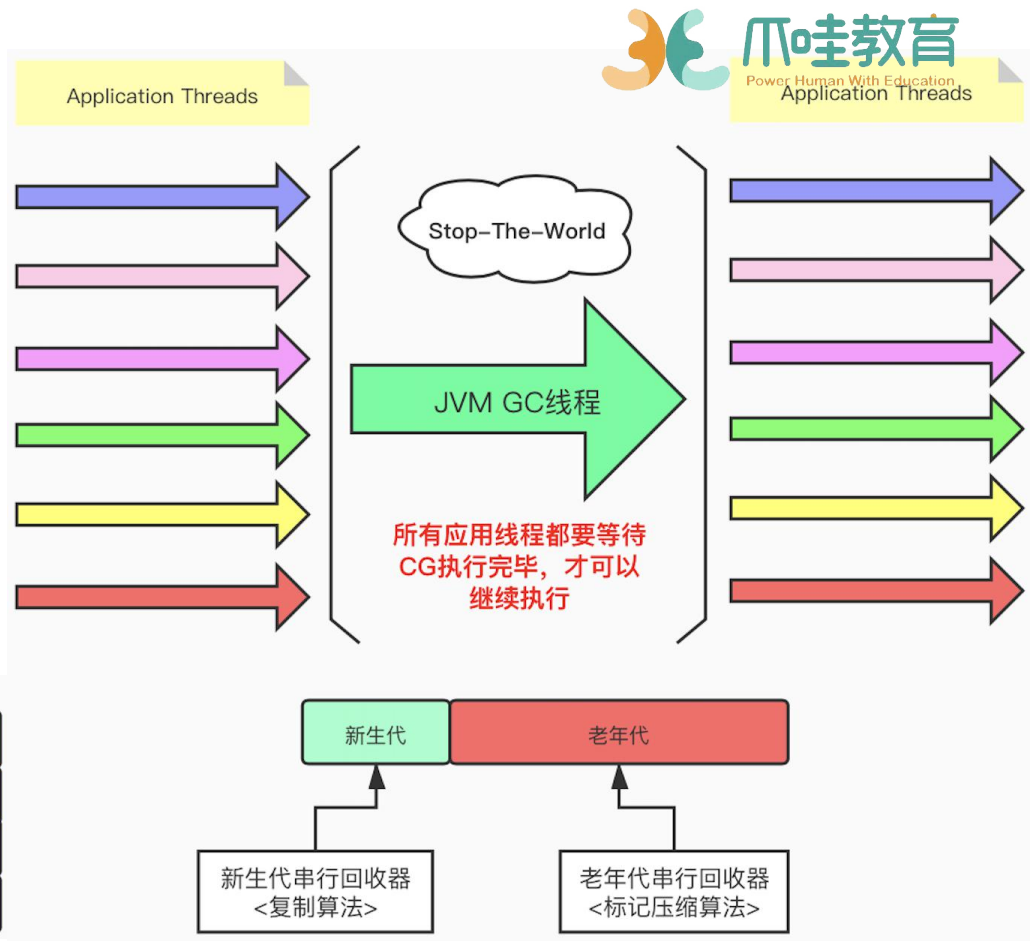
2>**独占式**的GC

串行收集器是JVM Client  
模式下默认的垃圾收集器

JVM参数	作用
-XX:SurvivorRatio	设置eden区与survivor区比例
-XX:PretenureSizeThreshold	设置大对象直接进入老年代的阈值
-XX:MaxTenuringThreshold	设置对象进入老年代的年龄阈值

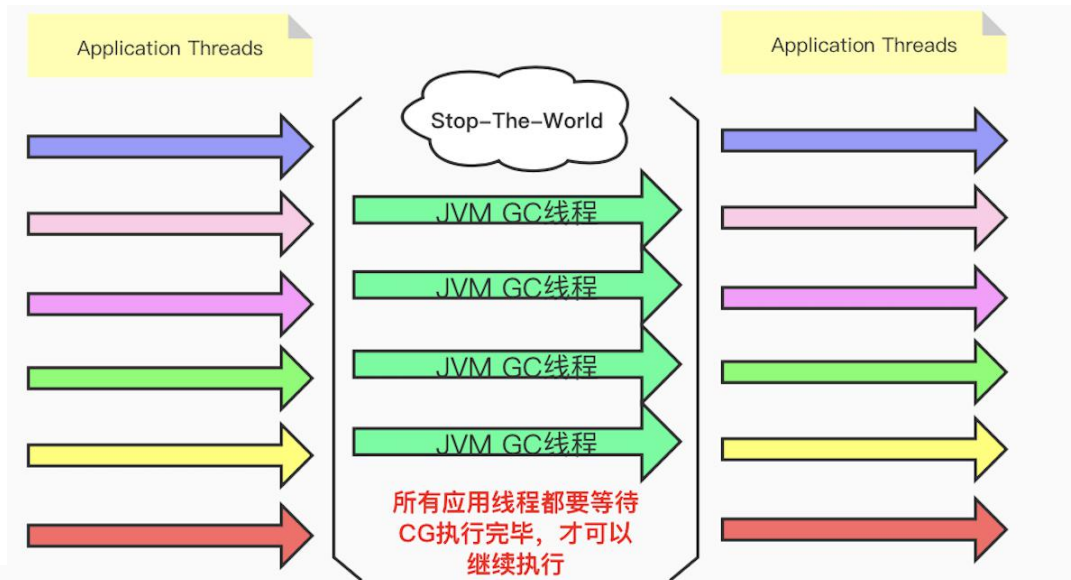
启用指定收集器

JVM参数	新生代	老年代
-XX:+UseSerialGC	串行回收器	串行回收器
-XX:+UseParNewGC	ParNew	串行回收器
-XX:+UseParallelGC	ParallelGC	串行回收器



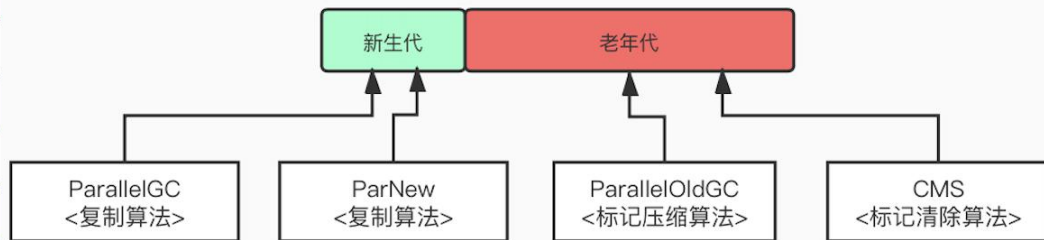
# 并行回收器 - ParNew & ParallelGC & ParallelOldGC

将串行回收器 **多线程化**。  
与串行回收器有相同的回收策略、  
算法、参数。



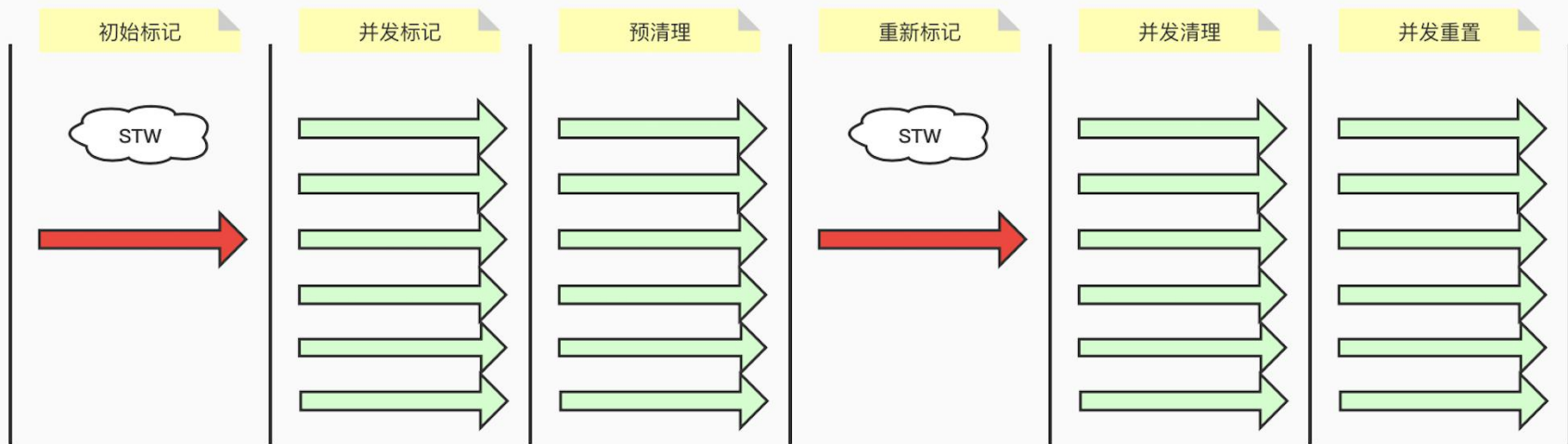
启用指定收集器

JVM参数	新生代	老年代
-XX:+UseParNewGC	ParNew	串行回收器
-XX:+UseConcMarkSweepGC	ParNew	CMS
-XX:+UseParallelGC	ParallelGC	串行回收器
-XX:+UseParallelOldGC	ParallelGC	ParallelOldGC





## CMS垃圾回收步骤



## 并行回收器jvm参数

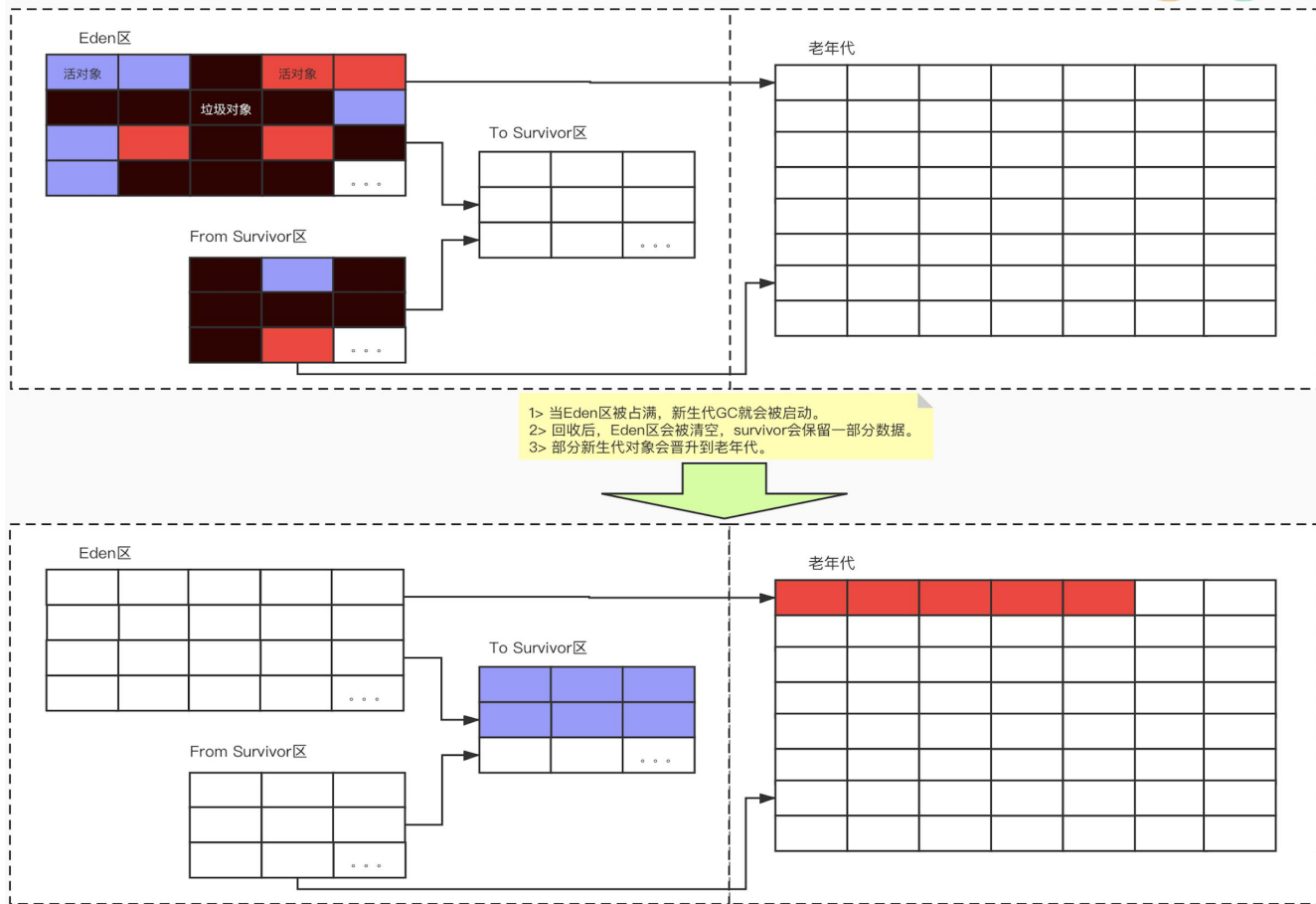
收集器JVM参数

收集器	JVM参数	作用
ParNew	-XX:ParallelGCThreads	指定GC时工作的线程数量
ParallelGC	-XX:MaxGCPauseMillis	最大的垃圾收集暂停时间
	-XX:GCTimeRatio	设置垃圾收集吞吐量
	-XX:+UseAdaptiveSizePolicy	打开自适应垃圾收集策略
ParallelOldGC	-XX:ParallelGCThreads	指定GC时工作的线程数量
CMS	-XX:-CMSPrecleaningEnabled	禁用预清理操作
	-XX:ConcGCThreads	设置并发线程数量
	-XX:ParallelCMSThreads	设置并发线程数量
	-XX:CMSInitiatingOccupancyFraction	当老年代空间使用量达到某百分比时，会执行CMS。默认68
	-XX:+CMSCompactAtFullCollection	GC后，进行一次碎片整理
	-XX:CMSFullGCsBeforeCompaction	指定执行多少次GC后，进行一次碎片整理

G1全称Garbage First Garbage Collector。优先回收垃圾**比例最高**的区域。  
G1收集器将堆划分为**多个区域**，每次收集**部分区域**来减少GC产生的停顿时间。

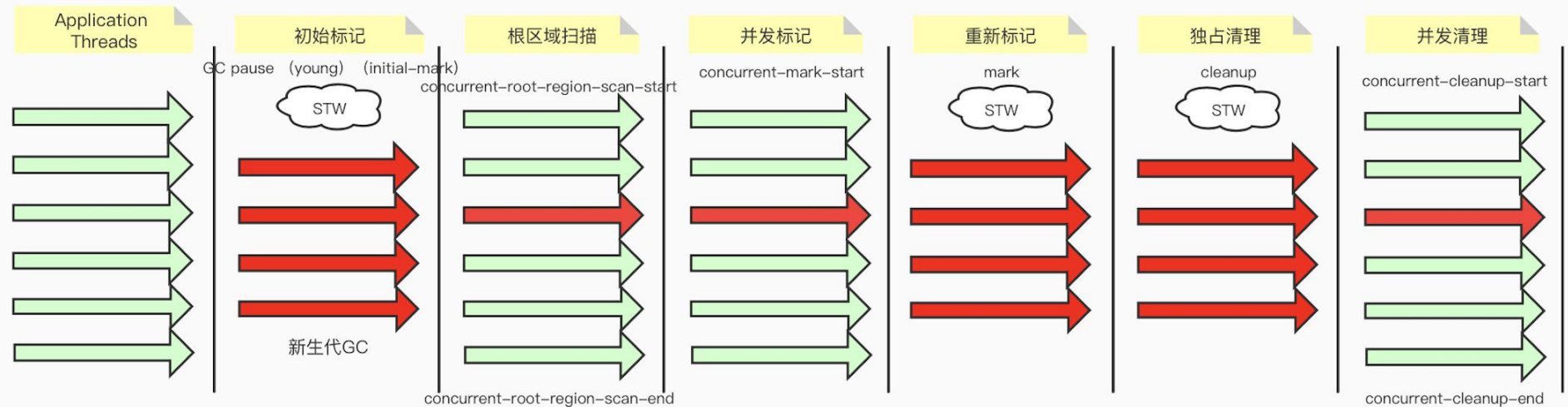


# G1 —— 新生代GC



## G1 —— 并发标记周期

### CMS垃圾回收步骤



# G1 —— 混合收集

G1收集过程

第一阶段  
新生代GC

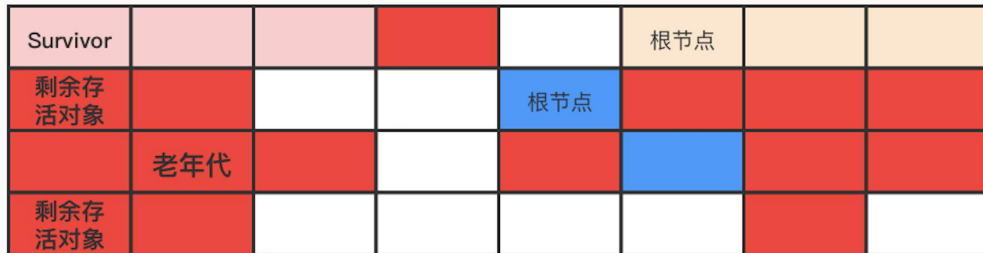
第二阶段  
并发标记周期

第三阶段  
混合回收

JVM参数	作用
-XX:+UseG1GC	打开G1收集器开关
-XX:MaxGCPauseMillis	指定目标最大停顿时间
-XX:ParallelGCThreads	设置并发线程数量
-XX:InitiatingHeapOccupancyPercent	指定堆的使用率是多少事，触发并发标记周期（默认45）



- 1> 由于触发年轻代GC，eden区域会被清空。
- 2> 被标记的年轻代和老年代都会被回收。
- 3> 垃圾比例高的会被清理，剩余存活对象会被移动其他区域，减少内存碎片。



# JVM常用参数

执行语法:

Java [-options] [package+className] [arg1,arg2,...,argN]

options:

-Xms128m	设置初始化堆内存为128M
-Xmx512m	设置最大堆内存为512M
-Xmn160m	设置新生代大小为-Xmn160M (堆空间1/4~1/3)
-Xss128m	设置最大栈内存为128M
-XX:SurvivorRatio	设置新生代eden区与from/to空间的比例关系
-XX:PermSize=64M	设置初始永久区64M
-XX:MaxPermSize=128M	设置最大永久区128M
-XX:MaxMetaspaceSize	设置元数据区大小 (JDK1.8 取代永久区)
-XX:+DoEscapeAnalysis	启用逃逸分析 (Server模式)
-XX:+EliminateAllocations	开启标量替换 (默认开启)
-XX:+TraceClassLoading	跟踪类的加载
-XX:+TraceClassUnloading	跟踪类的卸载
-Xloggc:gc.log	将gc日志信息打印到gc.log文件中



options:

-XX:+PrintGC	打印GC日志
-XX:+PrintGCDetails	打印GC详细日志
-XX:+PrintGCTimeStamps	输出GC发生的时间
-XX:+PrintGCApplicationStoppedTime	GC产生停顿的时间
-XX:+PrintGCApplicationConcurrentTime	应用执行的时间
-XX:+PrintHeapAtGC	在GC发生前后，打印堆栈日志
-XX:+PrintReferenceGC	打印对象引用信息
-XX:+PrintVMOptions	打印虚拟机参数
-XX:+PrintCommandLineFlags	打印虚拟机显式和隐式参数
-XX:+PrintFlagsFinal	打印所有系统参数
-XX:+PrintTLAB	打印TLAB相关分配信息
-XX:+UseTLAB	打开TLAB
-XX:TLABSize	设置TLAB大小
-XX:+ResizeTLAB	自动调整TLAB大小
-XX:+DisableExplicitGC	禁用显示GC (System.gc())
-XX:+ExplicitGCInvokesConcurrent	使用并发方式处理显式GC



# JVM监控优化

### Linux

top

vmstat

iostat

pidstat

### JDK

jps

jstat

jinfo

jmap

hprof

jhat

jstack

jstatd

jcmd

### 工具

JConsole

Visual VM

MissionControl

## Linux——top 命令

能够实时显示系统中各个进程的资源占用情况。  
分为两部分：**系统统计信息**&**进程信息**。

系统统计信息：

Line1:**任务队列信息**，从左到右依次表示：系统当前时间、系统运行时间、当前登录用户数。

**Load average**表示系统的**平均负载**，即任务队列的平均长度——1分钟、5分钟、15分钟到现在的平均值。

Line2:**进程统计信息**，分别是：正在运行进程数、睡眠进程数、停止的进程数、僵尸进程数。

Line3:**CPU统计信息**。us表示用户空间CPU占用率、sy表示内核空间CPU占用率、ni表示用户进程空间改变过优先级的进程CPU占用率。id表示空闲CPU占用率、wa表示待输入输出的CPU时间百分比、hi表示硬件中断请求、si表示软件中断请求。

Line4:**内存统计信息**。从左到右依次表示：**物理内存总量**、**已使用的物理内存**、**空闲物理内存**、**内核缓冲使用量**。

Line5:从左到右表示：交换区总量、已使用交换区大小、空闲交换区大小、缓冲交换区大小。



```
top - 00:04:17 up 219 days, 7:22, 2 users, load average: 0.00, 0.00, 0.00
Tasks: 348 total, 2 running, 346 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.6% us, 2.1% sy, 2.2% ni, 95.1% id, 0.0% wa, 0.0% hi, 0.1% si
Mem: 12201288k total, 11561828k used, 639460k free, 704188k buffers
Swap: 12582904k total, 204k used, 12582700k free, 8557552k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
31252	nobody	30	10	870m	35m	12m	S	12.9	0.3	1255:47	minos-agent-tk
31111	nobody	30	10	870m	35m	12m	S	12.3	0.3	1258:51	minos-agent
7223	root	39	19	733m	72m	9124	S	2.7	0.6	1813:15	baas_agent
32021	root	39	19	1388m	13m	11m	S	1.3	0.1	2355:44	gko3
18462	root	20	0	1242m	174m	7796	S	0.7	1.5	26:05:17	naming-agent
6460	root	39	19	1268m	85m	18m	S	0.3	0.7	1552:20	webdir-agent
7029	work	20	0	14724	1480	920	R	0.3	0.0	0:00.04	top
7177	root	20	0	700m	5624	4296	S	0.3	0.0	141:51.34	monitor_baas_ag
8415	root	39	19	326m	4292	3148	S	0.3	0.0	181:03.15	webfoot-agent
10890	root	39	19	1891m	9176	5704	S	0.3	0.1	849:57.96	drct
12290	root	20	0	700m	5736	4272	S	0.3	0.0	655:34.36	monitor_gianod.
12349	root	39	19	533m	45m	6364	S	0.3	0.4	961:41.36	gianod
20689	nobody	39	19	661m	44m	5772	S	0.3	0.4	1289:02	bsdc_agent

# Linux——top 命令



进程信息：

**PID：**进程id

**USER：**进程所有者

**PR：**优先级

**NI：**nice值，负值→高优先级，正值→低优先级

**VIRT：**进程使用虚拟内存总量 VIRT=SWAP+RES

**RES：**进程使用并未被换出的内存。CODE+DATA

**SHR：**共享内存大小

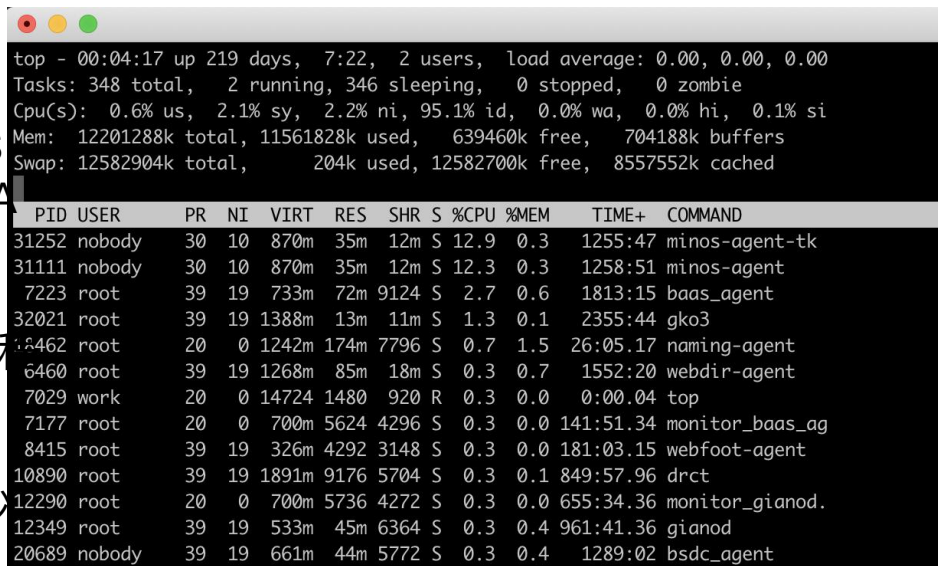
**S：**进程状态。D=不可中断的睡眠状态 R=运行  
S=睡眠 T=跟踪/停止 Z=僵尸进程

**%CPU：**上次更新到现在的CPU时间占用百分比

**%MEM：**进程使用的物理内存百分比

**TIME+：**进程使用的CPU时间总计，单位 1/100秒

**COMMAND：**命令行



```
top - 00:04:17 up 219 days, 7:22, 2 users, load average: 0.00, 0.00, 0.00
Tasks: 348 total, 2 running, 346 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.6% us, 2.1% sy, 2.2% ni, 95.1% id, 0.0% wa, 0.0% hi, 0.1% si
Mem: 12201288k total, 11561828k used, 639460k free, 704188k buffers
Swap: 12582904k total, 204k used, 12582700k free, 8557552k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
31252	nobody	30	10	870m	35m	12m	S	12.9	0.3	1255:47	minos-agent-tk
31111	nobody	30	10	870m	35m	12m	S	12.3	0.3	1258:51	minos-agent
7223	root	39	19	733m	72m	9124	S	2.7	0.6	1813:15	baas_agent
32021	root	39	19	1388m	13m	11m	S	1.3	0.1	2355:44	gko3
16462	root	20	0	1242m	174m	7796	S	0.7	1.5	26:05.17	naming-agent
6460	root	39	19	1268m	85m	18m	S	0.3	0.7	1552:20	webdir-agent
7029	work	20	0	14724	1480	920	R	0.3	0.0	0:00.04	top
7177	root	20	0	700m	5624	4296	S	0.3	0.0	141:51.34	monitor_baas_ag
8415	root	39	19	326m	4292	3148	S	0.3	0.0	181:03.15	webfoot-agent
10890	root	39	19	1891m	9176	5704	S	0.3	0.1	849:57.96	drct
12290	root	20	0	700m	5736	4272	S	0.3	0.0	655:34.36	monitor_gianod.
12349	root	39	19	533m	45m	6364	S	0.3	0.4	961:41.36	gianod
20689	nobody	39	19	661m	44m	5772	S	0.3	0.4	1289:02	bsdc_agent

**性能监测工具**，显示单位均为kb。它可以统计CPU、内存使用情况、swap使用情况等信息，也可以指定采样周期和采用次数。例如：每秒采样一次，共计3次。

### vmstat 1 3

procs		memory				swap		io		system		cpu				
r	b	swpd	free	buff	cache	si	so	bi	bo	in	cs	us	sy	id	wa	st
3	0	204	692460	704080	8511796	0	0	1	18	0	0	2	2	96	0	0
1	0	204	692576	704080	8511796	0	0	0	4	28110	54114	2	2	96	0	0
2	0	204	692064	704080	8511796	0	0	0	0	28328	54483	3	2	95	0	0

procs列：r表示等待运行的进程数。b表示处于非中断睡眠状态的进程数。

memory列：swpd表示虚拟内存使用情况。free表示空闲内存量。buff表示被用来作为缓存的内存。

swap列：si表示从磁盘交换到内存的交换页数量。so表示从内存交换到磁盘的交换页数量。

io列：bi表示发送到块设备的块数，单位：块/秒。bo表示从块设备接收到的块数。

system列：in表示每秒的中断数，包括时钟中断。cs表示每秒的上下文切换次数。

cpu列：us表示用户cpu使用时间。sy表示内核cpu系统使用时间。id表示空闲时间。

wa表示等待io时间。

可以提供详尽的I/O信息。

如果只看磁盘信息，可以使用-d参数。即：iostat -d 1 3 （每1秒采集一次持续3次）

tps列表示该设备每秒的传输次数。

Blk\_read/s列表示每秒读取块数。

Blk\_wrtn/s列表示每秒写入块数。

Blk\_read列表示读取块数总量。

Blk\_wrtn列表示写入块数总量。

### iostat 1 1

avg-cpu:	%user	%nice	%system	%iowait	%steal	%idle
	0.74	0.93	2.07	0.02	0.16	96.08
Device:	tps	Blk_read/s	Blk_wrtn/s	Blk_read	Blk_wrtn	
vda	2.75	0.19	93.32	3512258	1768795696	
vdb	2.87	<u>11.04</u>	51.08	209298690	968318672	
vdc	0.00	0.00	0.00	1656	408	

用于列出Java的进程。

执行语法：`jps [-options]`

`jps` 列出java进程id和类名

`91275 FireLOTest`

`jps -q` 仅列出java进程id

`91275`

`jps -m` 输出java进程的入参

`91730 FireLOTest a b`

`jps -l` 输出主函数的完整路径

`91730 day1.FireLOTest`

`jps -v` 显示传递给JVM的参数

`91730 FireLOTest -Xmx512m -XX:+PrintGC -javaagent:/Applications/IntelliJ  
IDEA.app/Contents/lib/idea_rt.jar=51673:/Applications/IntelliJ IDEA.app/Contents/bin -  
Dfile.encoding=UTF-8`



## JDK工具——jstat

用于查看堆中的运行信息。

执行语法: `jstat -help` `jstat -options`

`jstat <-option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]`

`jstat -class -t 73608 1000 5` 查看进程73608的ClassLoader相关信息, 每1000毫秒打印1次, 一共

打印5次, 并输出程序启动到此刻的Timestamp数。

`jstat -compiler -t 73608` 查看指定进程的编译信息。

`jstat -gc 73608` 查看指定进程的堆信息。

`jstat -gccapacity 73608` 查看指定进程中每个代的容量与使用情况

`jstat -gccause 73608` 显示最近一次gc信息

`jstat -gcmetacapacity 73608` 查看指定进程的元空间使用信息

`jstat -gcnew 73608` 查看指定进程的新生代使用信息

`jstat -gcnewcapacity 73608` 查看指定进程的新生代各区大小信息

`jstat -gcold 73608` 查看指定进程的老年代使用信息

`jstat -gcoldcapacity 73608` 查看指定进程的老年代各区大小信息

`jstat -gcutil 73608` 查看指定进程的GC回收信息

`jstat -printcompilation 73608` 查看指定进程的JIT编译方法统计信息

用于查看运行中java进程的虚拟机参数。

执行语法：

`jinfo [option] <pid>`

`jinfo -flag MaxTenuringThreshold 73608`      查看进程73608的虚拟机参数MaxTenuringThreshold的值

`jinfo -flag +PrintGCDetails 73608`      动态添加进程73608的虚拟机参数+PrintGCDetails，开启GC日志打印。

`jinfo -flag -PrintGCDetails 73608`      动态添加进程73608的虚拟机参数+PrintGCDetails，关闭GC日志打印。

命令用于生成指定java进程的dump文件；可以查看堆内对象实例的统计信息，查看ClassLoader信息和finalizer队列信息。

执行语法：

```
jmap [option] <pid>
```

jmap -histo 73608 > /Users/muse/a.txt 输出进程73608的实例个数与合计到文件a.txt中

jmap -dump:format=b,file=/Users/muse/b.hprof 73608 输出进程73608的堆快照，可使用jhat、visual VM等进行分析

命令用于分析jmap生成的堆快照。

执行语法：

jhat [-stack <bool>] [-refs <bool>] [-port <port>] [-baseline <file>] [-debug <int>] [-version] [-h|-help] <file>

jhat b.hprof      分析jmap生成的堆快照b.hprof,  
<http://127.0.0.1:7000>通过这个地址查  
OQL (Object Query Language)

```
Reading from b.hprof...
Dump file created Thu Aug 13 14:20:09 CST 2020
Snapshot read, resolving...
Resolving 12746 objects...
Chasing references, expect 2 dots..
Eliminating duplicate references..
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

## Object Query Language (OQL) query

[All Classes \(excluding platform\)](#) [OQL Help](#)

select file.path.value.toString() from java.io.File file

Execute

/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/rt.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/charsets.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/ext/jfxrt.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/ext/sunpkcs11.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/ext/zipfs.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/ext/sunec.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/resources.jar
/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/jre/lib/ext/localedata.jar

命令用于导出指定java进程的堆栈信息。

执行语法：

```
jstack [-l] <pid>
```

`jstack -l 73608 > /Users/muse/d.txt` 输出进程73608的实例个数与合计到文件a.txt中

`cat /Users/muse/d.txt`

DeadLockTest.java

```
"Thread-1" #12 prio=5 os_prio=31 tid=0x00007fd0ef8a4000 nid=0x5603 waiting for monitor entry [0x00007000ef9c000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at day1.DeadLockTest.lambda$main$1(DeadLockTest.java:38)
    - waiting to lock <0x0000000079569a190> (a java.lang.String)
    - locked <0x0000000079569a1c0> (a java.lang.String)
    at day1.DeadLockTest$$Lambda$2/381259350.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)

Locked ownable synchronizers:
- None

"Thread-0" #11 prio=5 os_prio=31 tid=0x00007fd0f094b800 nid=0xa803 waiting for monitor entry [0x00007000ee99000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at day1.DeadLockTest.lambda$main$0(DeadLockTest.java:25)
    - waiting to lock <0x0000000079569a1c0> (a java.lang.String)
    - locked <0x0000000079569a190> (a java.lang.String)
    at day1.DeadLockTest$$Lambda$1/931919113.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)
```

命令用于导出指定java进程的堆栈信息，查看进程，GC等。

执行语法：

```
jcmd <pid | main class> <command ...|PerfCounter.print|-f file>
```

jcmd -l	列出java进程列表
jcmd 26586 help	输出进程java进程为26586所支持的jcmd指令
jcmd 26586 VM.uptime	查看java进程启动时间
jcmd 26586 Thread.print	打印线程栈信息
jcmd 26586 GC.class_histogram	查看系统中类的统计信息
jcmd 26586 GC.heap_dump /Users/muse/a.txt	导出堆信息
jcmd 26586 VM.system_properties	获得系统的Properties内容
jcmd 26586 VM.flags	获得启动参数
jcmd 26586 PerfCounter.print	获得性能统计相关数据