



性能优化专题

JVM概念

主讲人：ROBERT: 2831742582

目录 /CONTENTS

01

虚拟机组成

02

运行原理

03

内存管理

04

垃圾回收策略

01

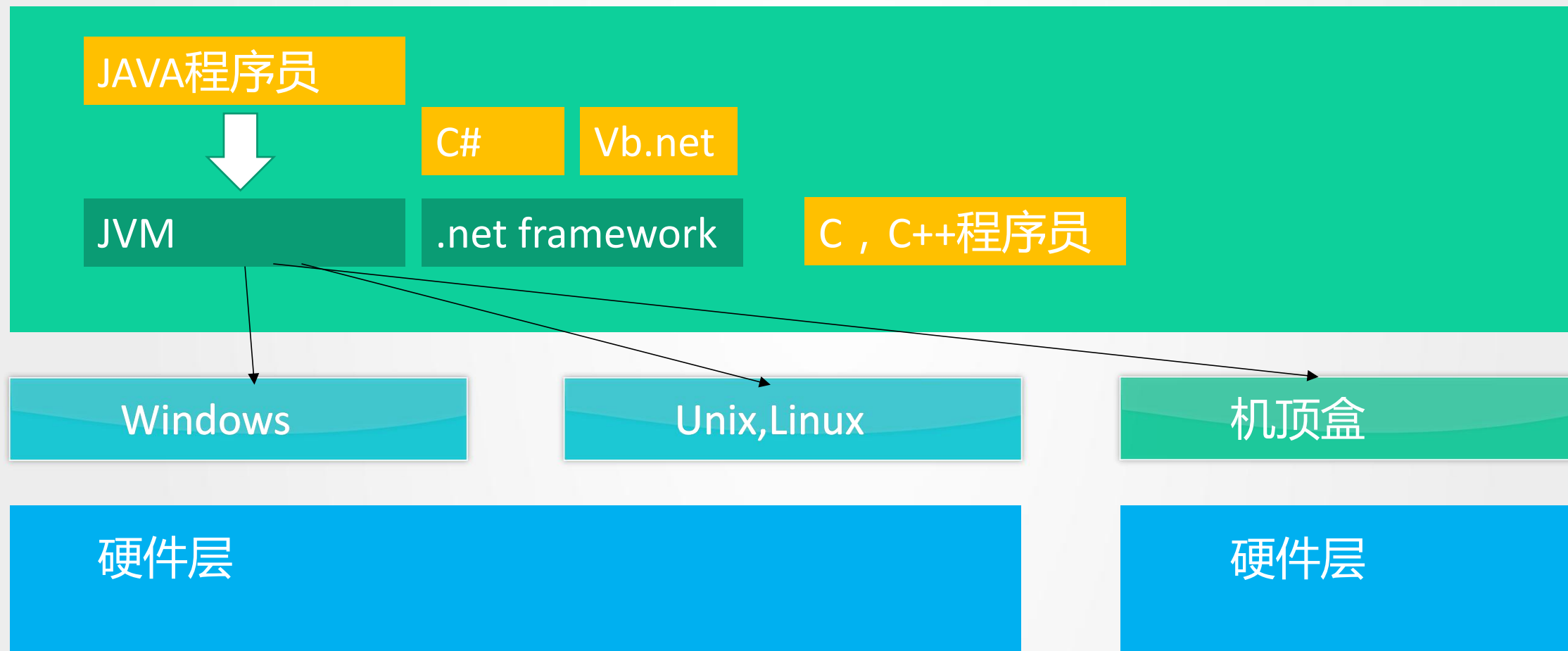
虚拟机组成

VIRTUAL MACHINE COMPOSITION

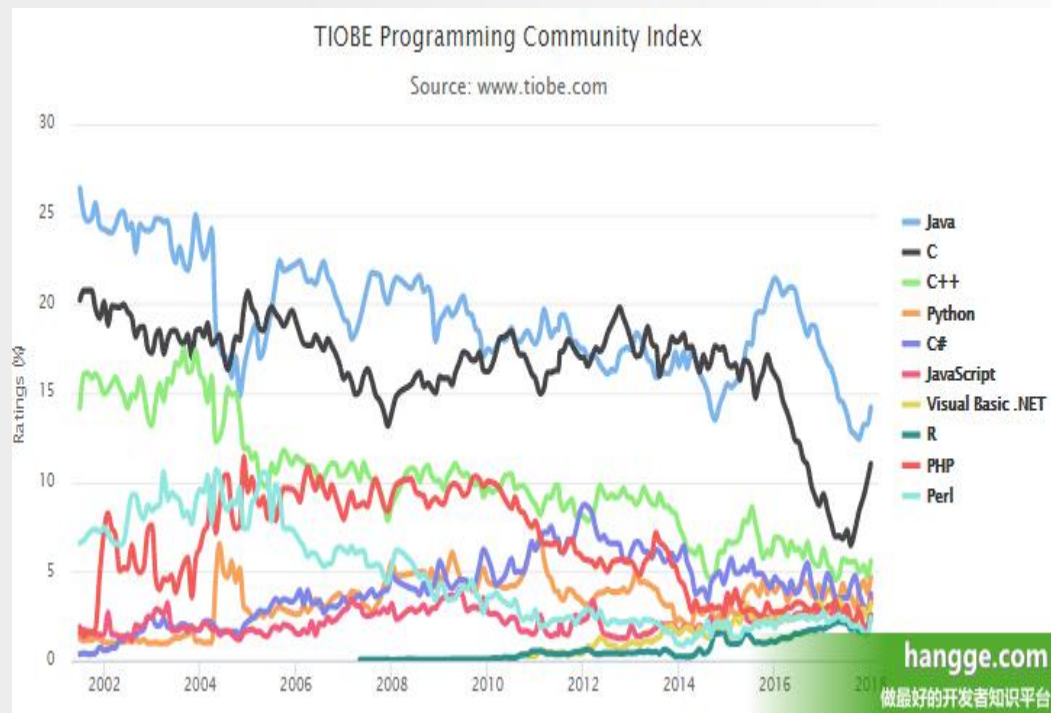


一次编写，到处运行

Java虚拟机是对操作系统的模拟，隔离差异



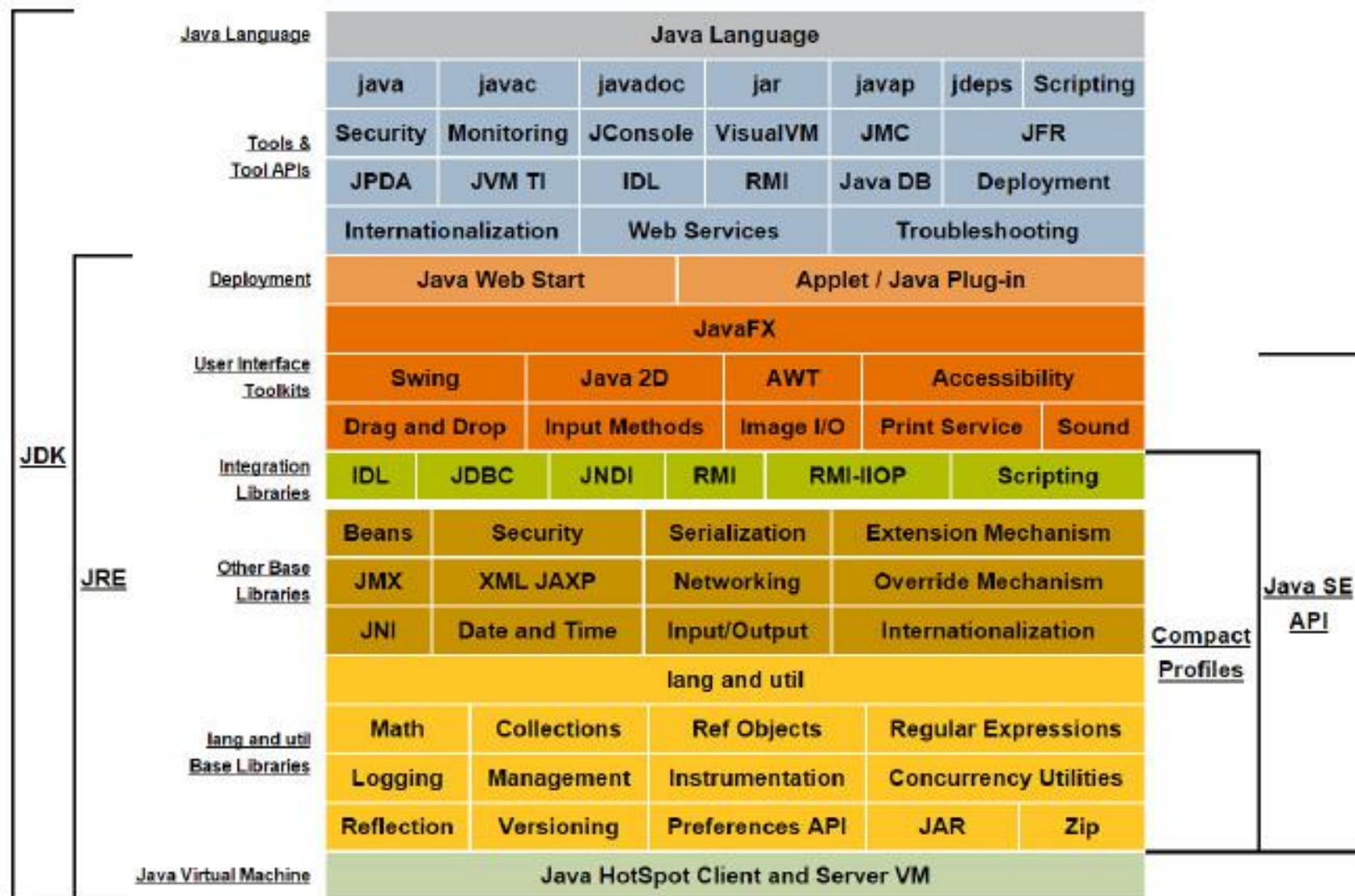
2018编程语言排行榜



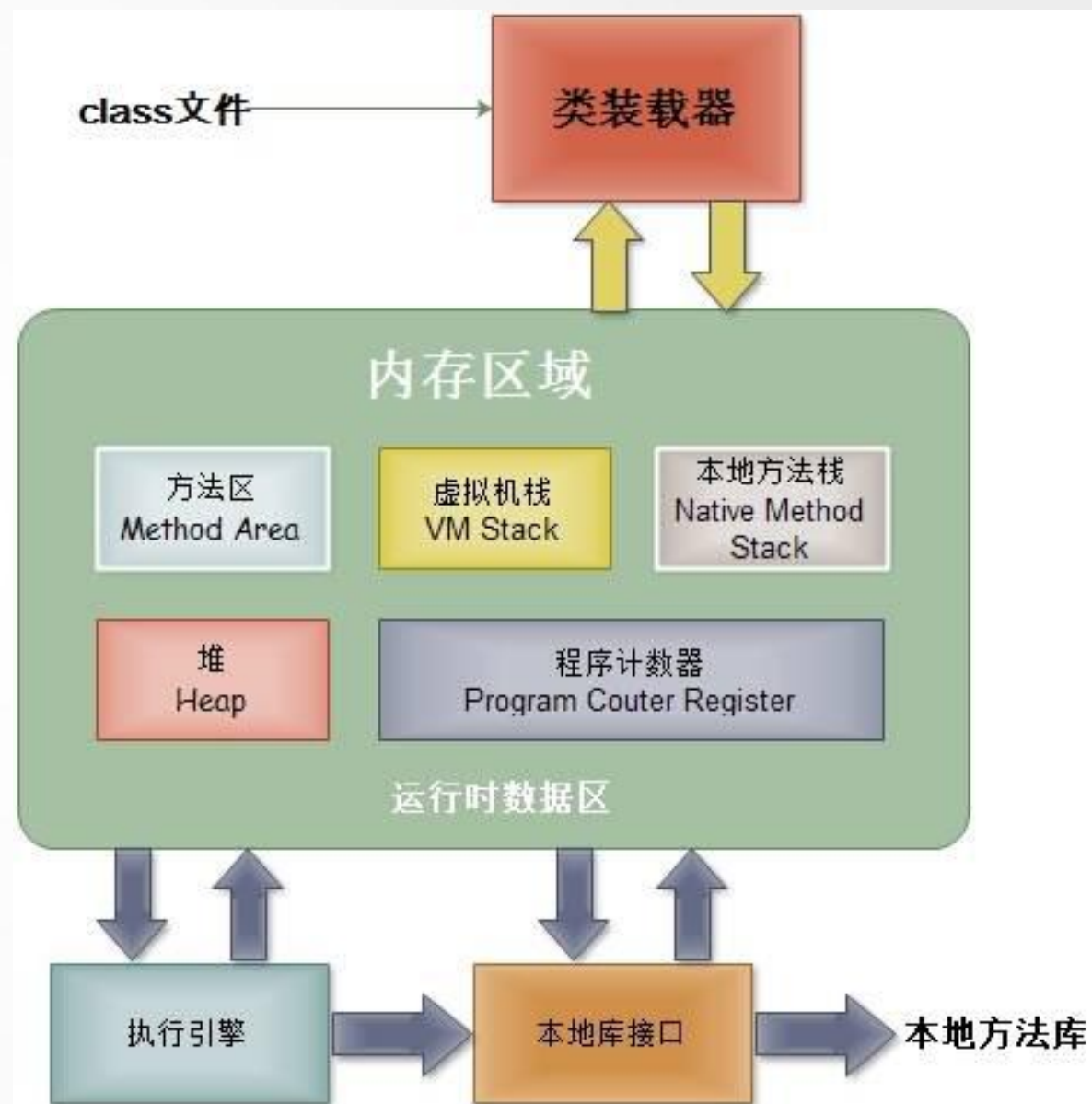
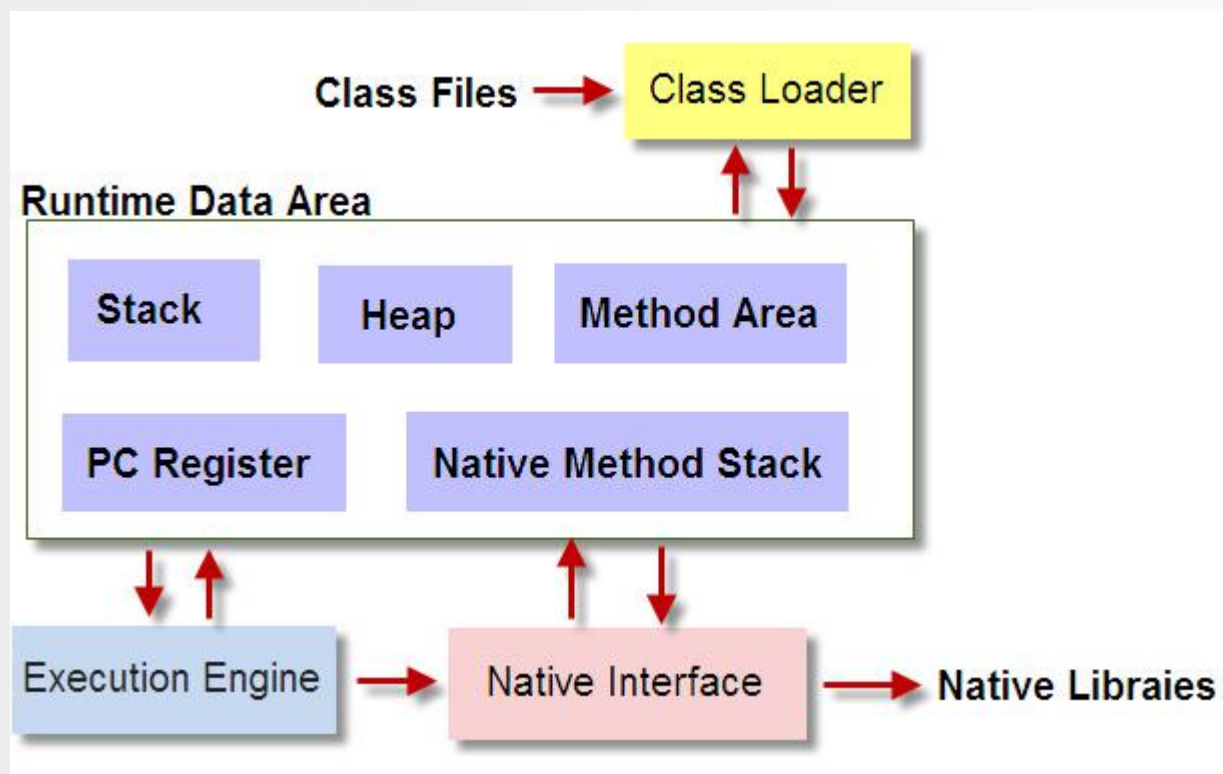
Jan 2018	Jan 2017	Change	Programming Language	Ratings	Change
1	1		Java	14.215%	-3.06%
2	2		C	11.037%	+1.69%
3	3		C++	5.603%	-0.70%
4	5	▲	Python	4.678%	+1.21%
5	4	▼	C#	3.754%	-0.29%
6	7	▲	JavaScript	3.465%	+0.62%
7	6	▼	Visual Basic .NET	3.261%	+0.30%
8	16	▲▲	R	2.549%	+0.76%
9	10	▲	PHP	2.532%	-0.03%
10	8	▼	Perl	2.419%	-0.33%
11	12	▲	Ruby	2.406%	-0.14%
12	14	▲	Swift	2.377%	+0.45%
13	11	▼	Delphi/Object Pascal	2.377%	-0.18%
14	15	▲	Visual Basic	2.314%	+0.40%
15	9	▼▼	Assembly language	2.056%	-0.65%
16	18	▲	Objective-C	1.860%	+0.24%
17	23	▲▲	Scratch	1.740%	+0.58%
18	19	▲	MATLAB	1.653%	+0.07%
19	13	▼▼	Go	1.569%	-0.76%
20	20		PL/SQL	1.429%	

一个复杂的构架

Description of Java Conceptual Diagram



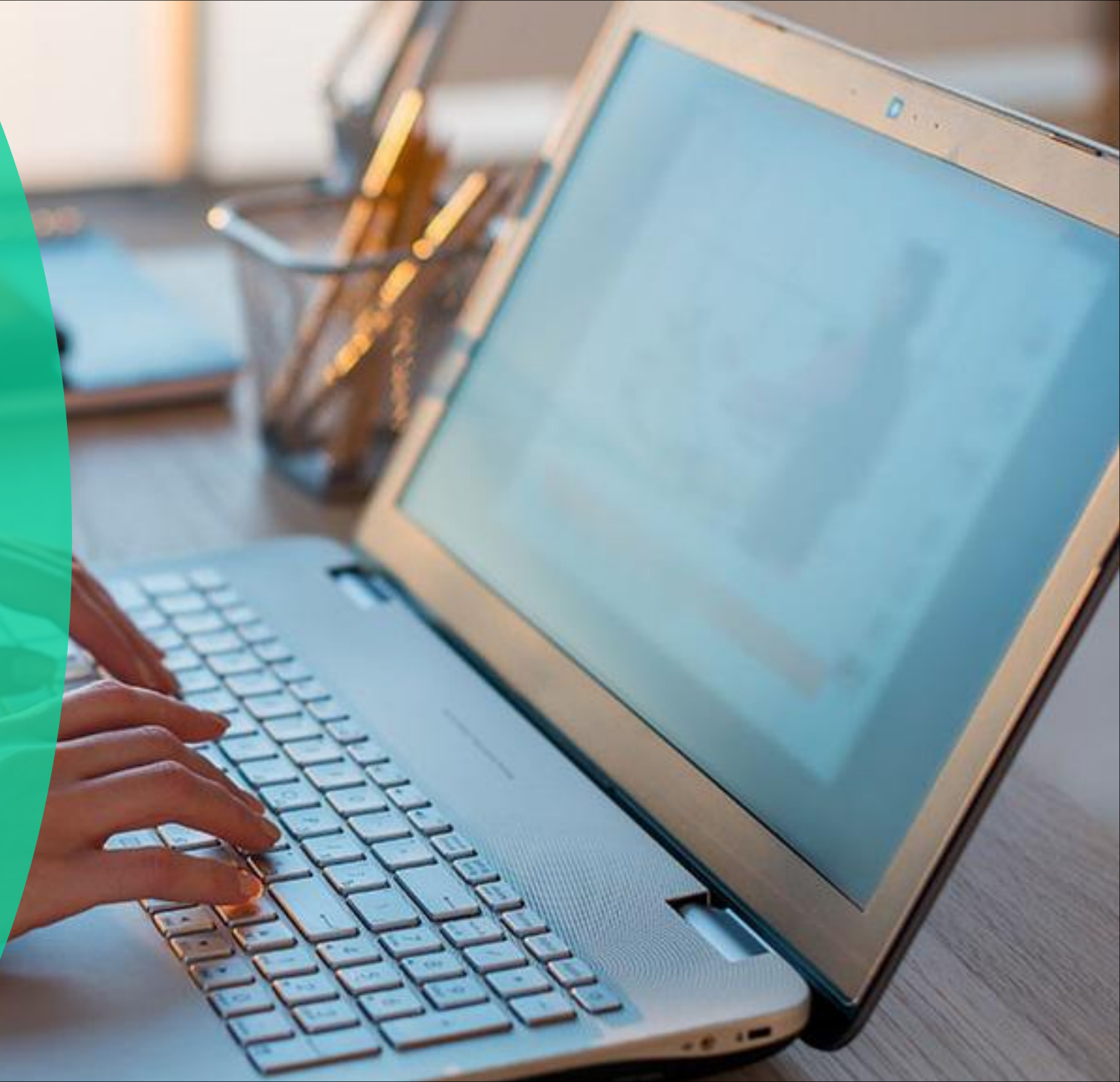
虚拟机的内部概念



02

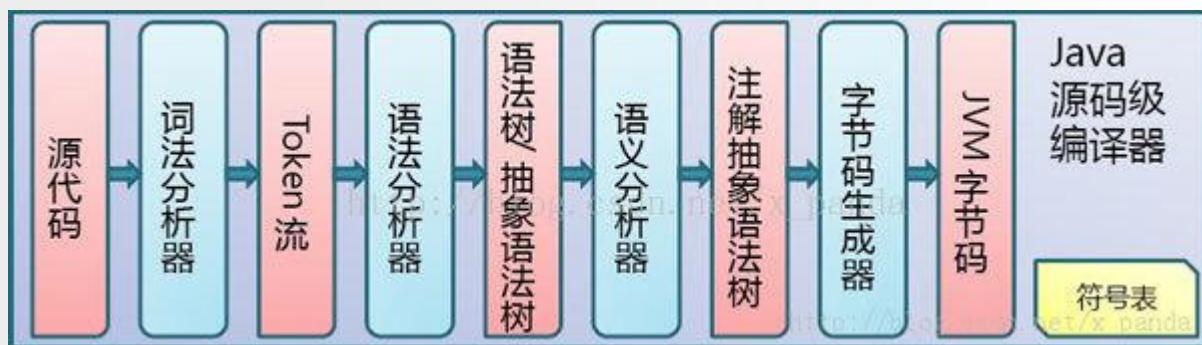
运行原理

OPERATING PRINCIPLE



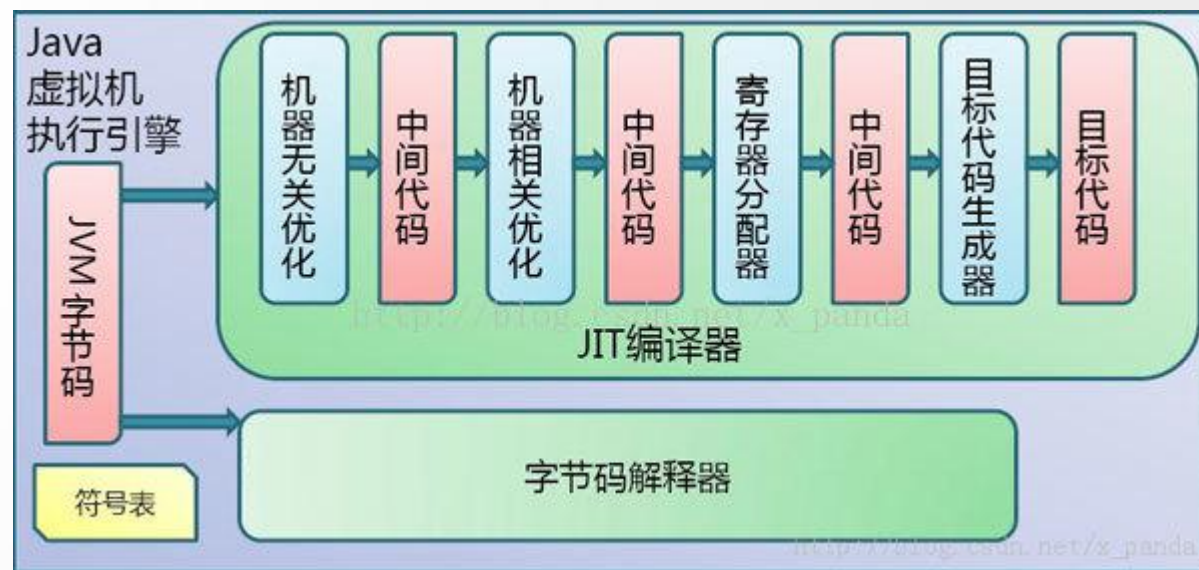
编译器，解释器执行流程

编译流程



编译器

解释器



内存分配-线程模型

每个线程一套

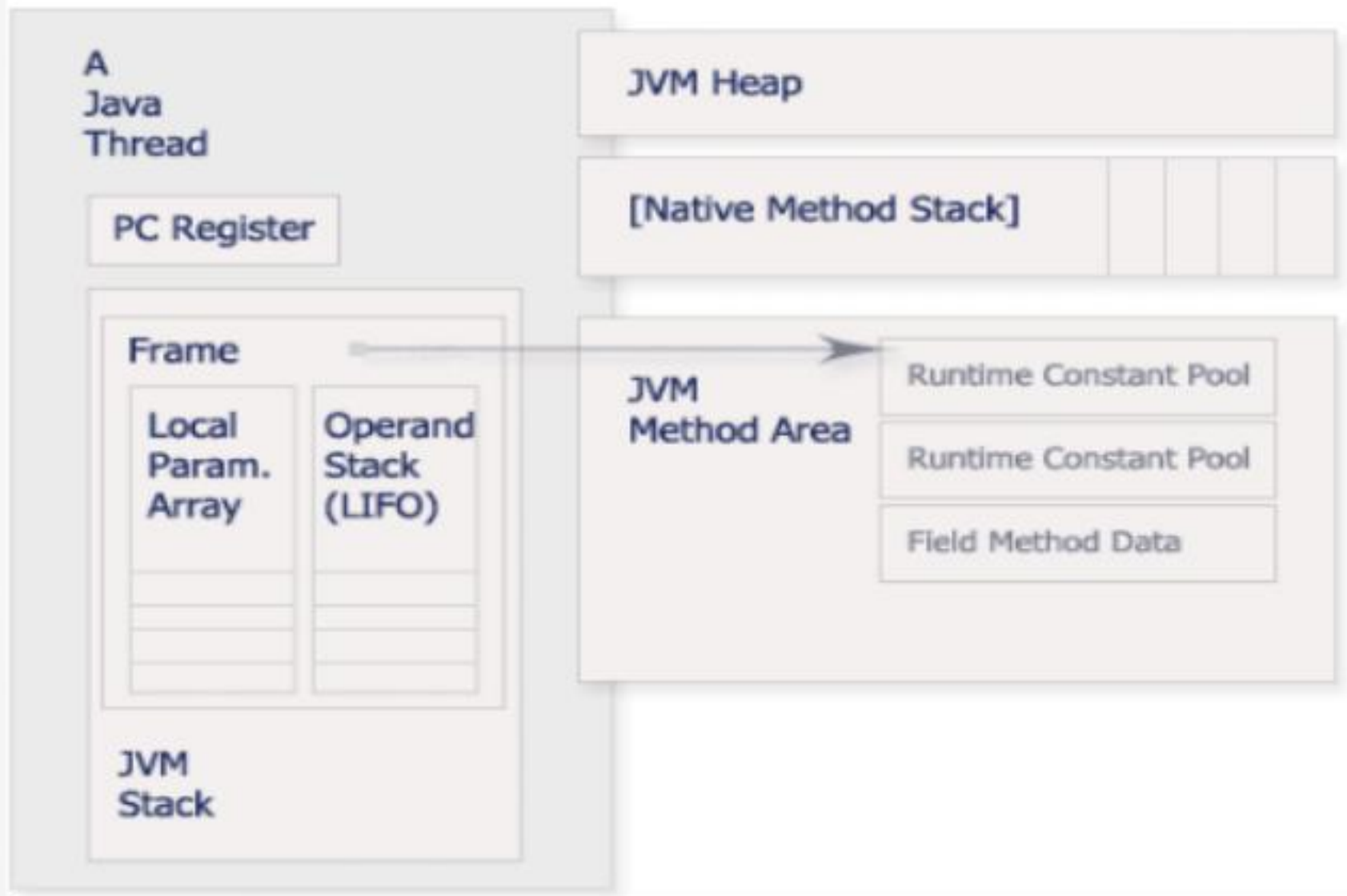
在Java中一个线程就会相应有一个线程栈与之对应。

堆是所有线程共享的。

栈是运行单位，信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；

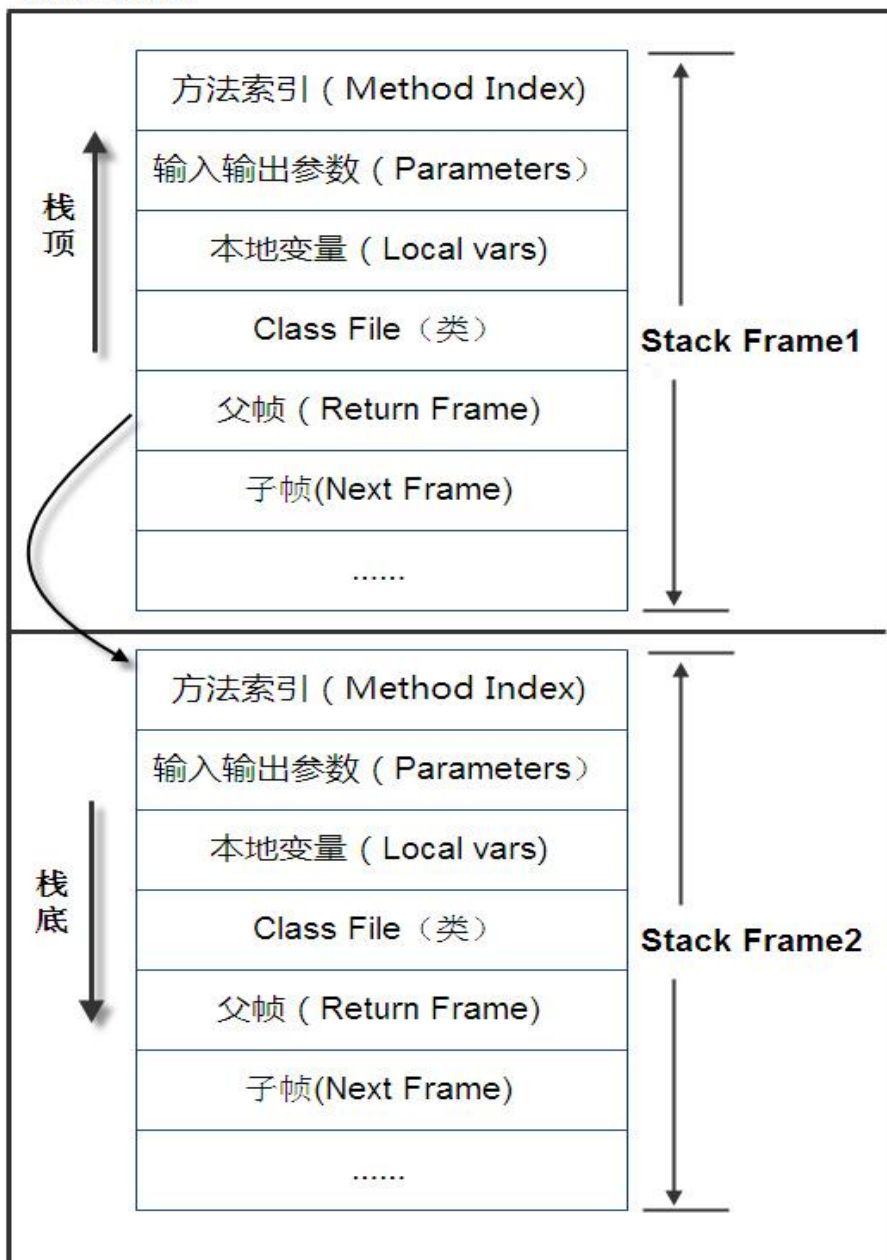
而堆只负责存储对象信息。

JVM Memory Structure

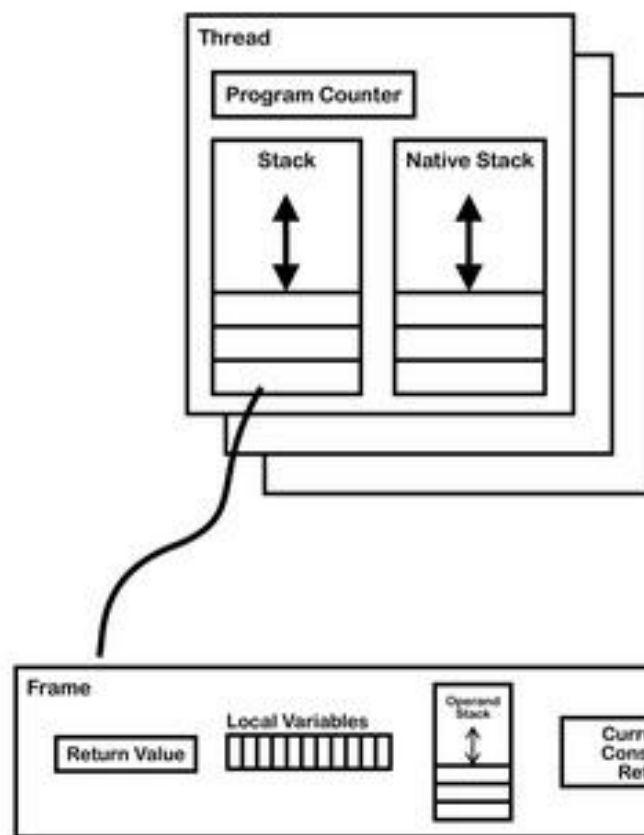


栈帧模型

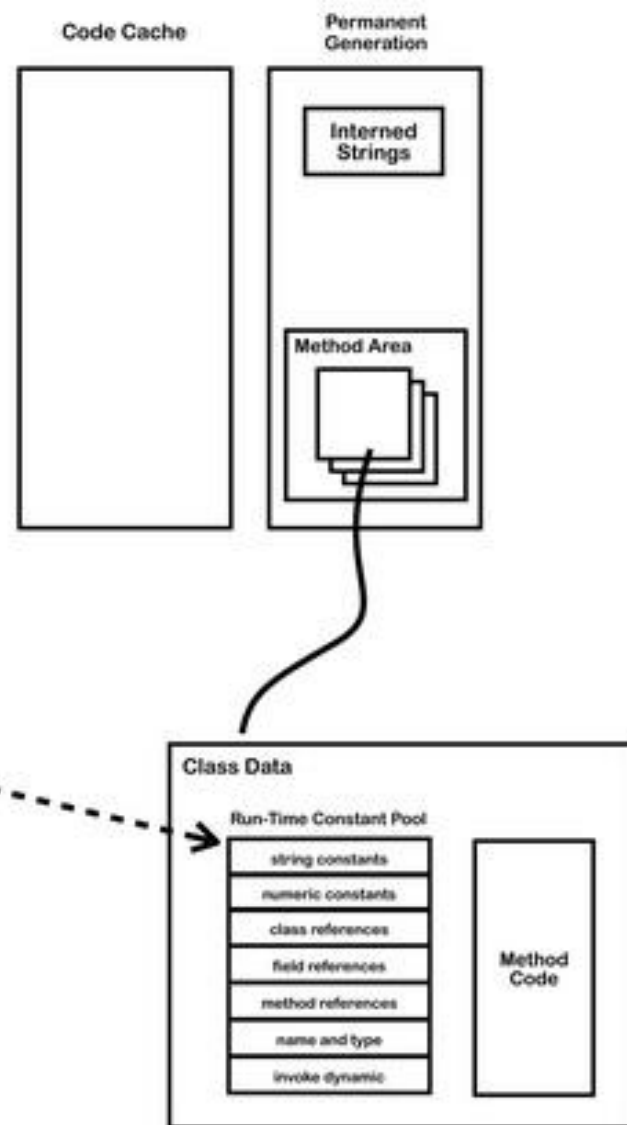
Java Stack



Stack



Non Heap

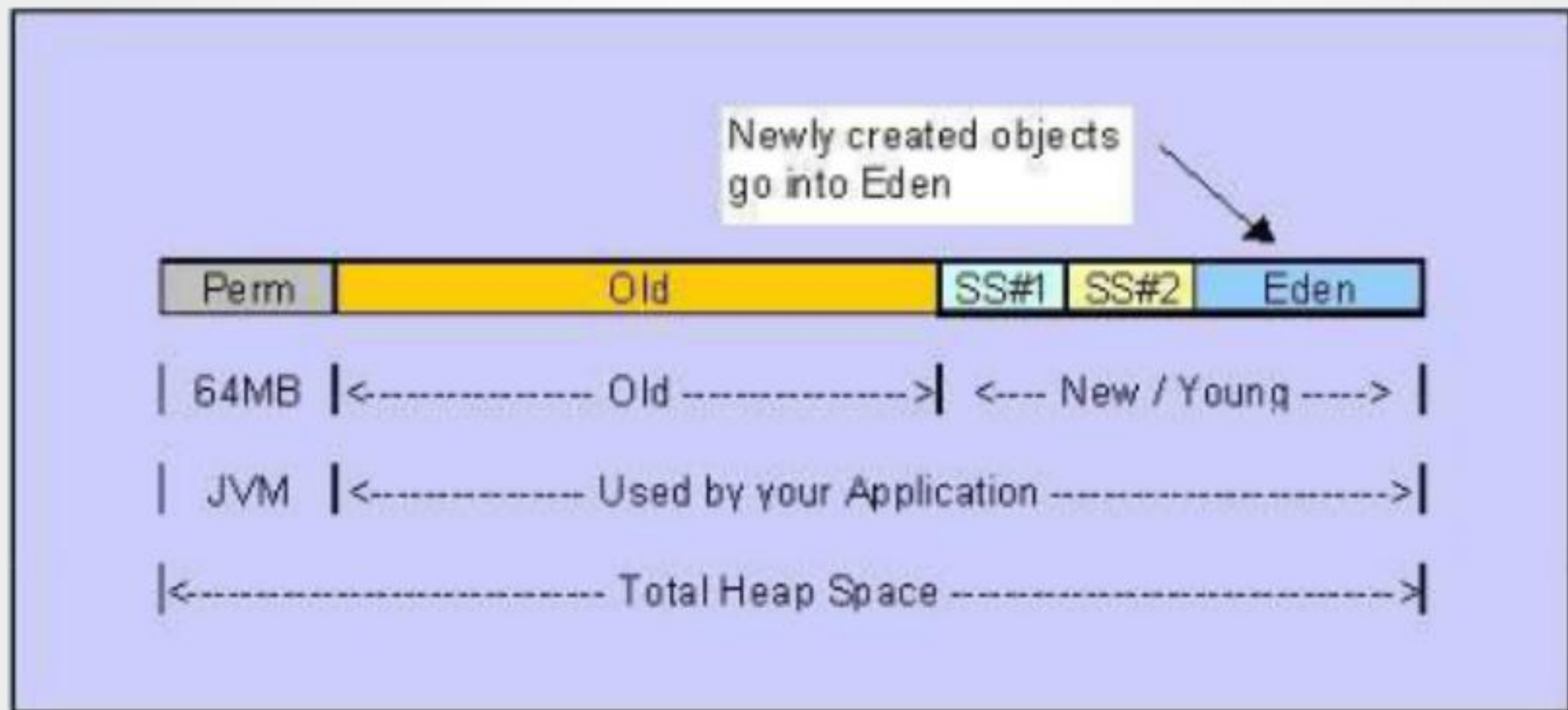


03

内存管理

MEMORY MANAGEMENT





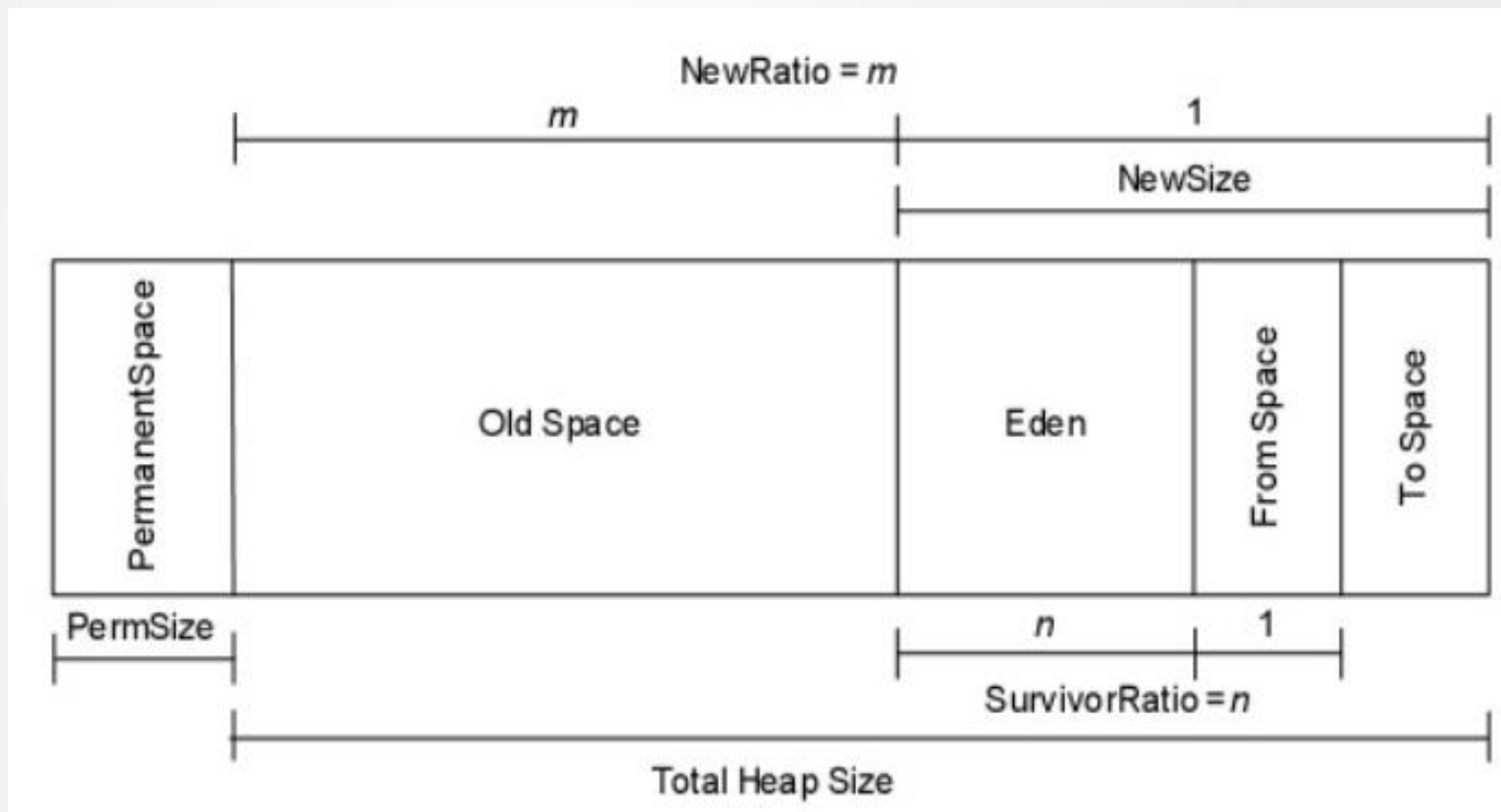
堆内存分配

Perm 默认大小64M

NewRatio配比

SurvivorRatio配比

Xmx, Xms, Xmn



04

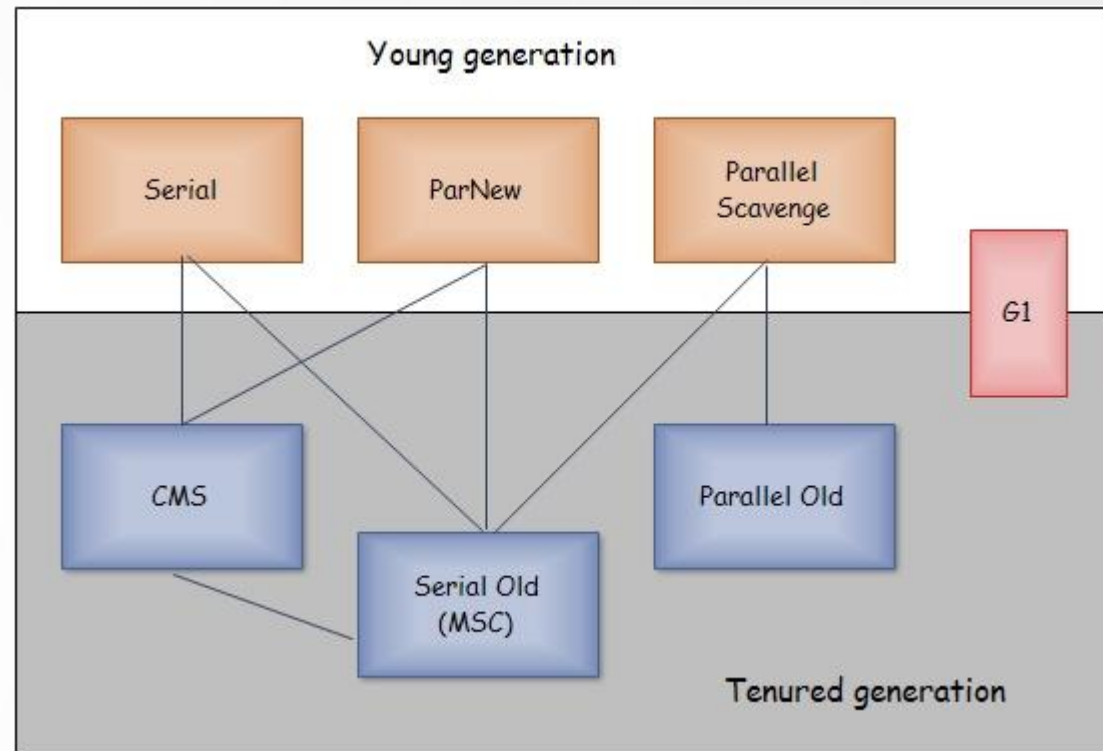
垃圾回收策略

GARBAGE COLLECTION STRATEGY



收集器

- 1.Serial GC。单线程，所有的线程暂停。一般用于Client模式的JVM中。
- 2.ParNew GC。是在SerialGC的基础上，增加了多线程机制。
- 3.Parallel Scavenge GC。吞吐量优先收集器，吞吐量=程序运行时间/(JVM执行回收的时间+程序运行时间)，运行100分钟，GC占用1分钟，吞吐量=99%。server模式JVM默认配置。
- 4.ParallelOld。老年代并行收集器的一种，使用了标记整理算法，是JDK1.6中引进的。
- 5.Serial Old，CMS收集器失败后的备用收集器。
- 6.CMS又称响应时间优先回收器，使用标记清除算法。他的回收线程数为 $(\text{CPU核心数}+3)/4$ ，所以当CPU核心数为2时比较高效些。CMS分为4个过程：初始标记、并发标记、重新标记、并发清除。



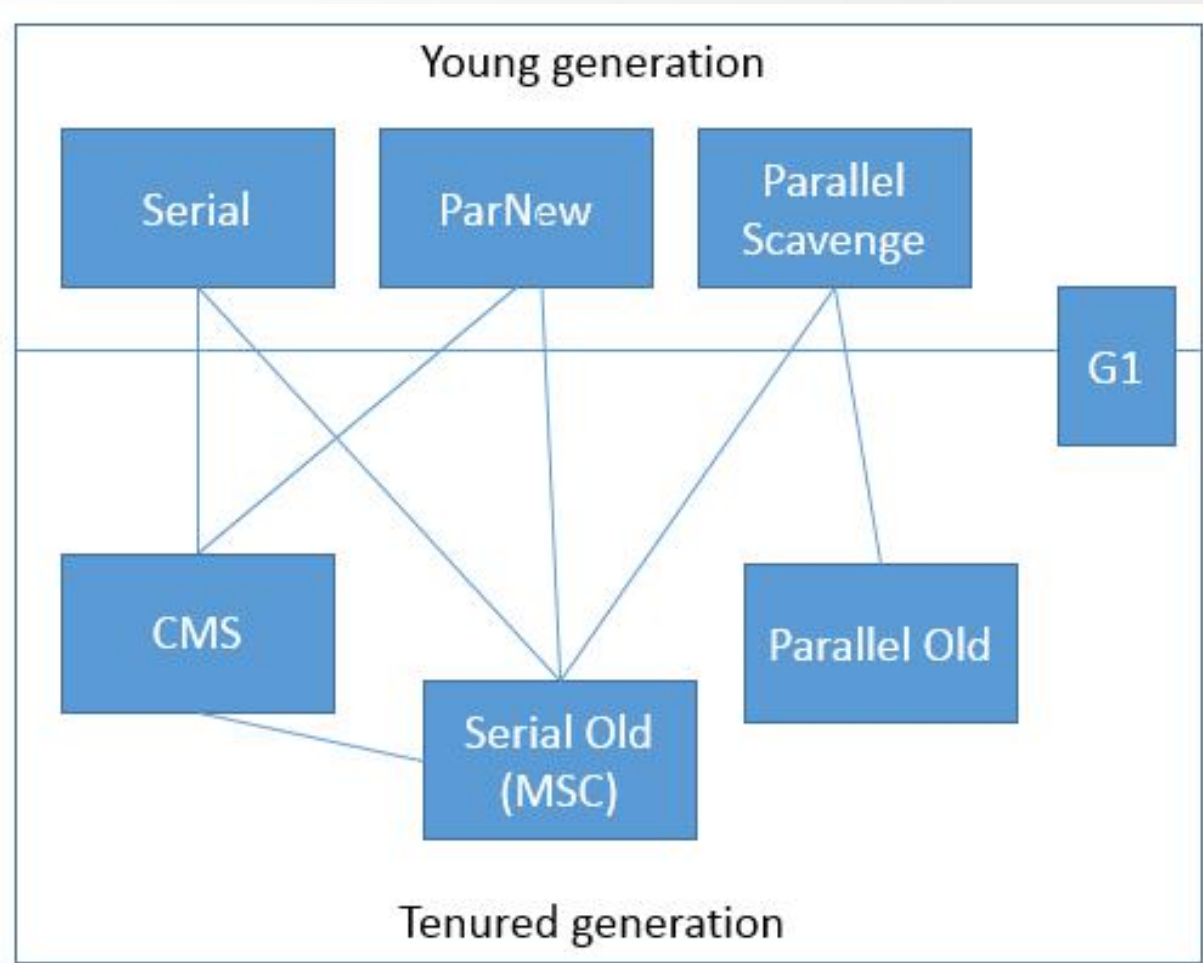
G1收集器=洋气

面向服务器- server

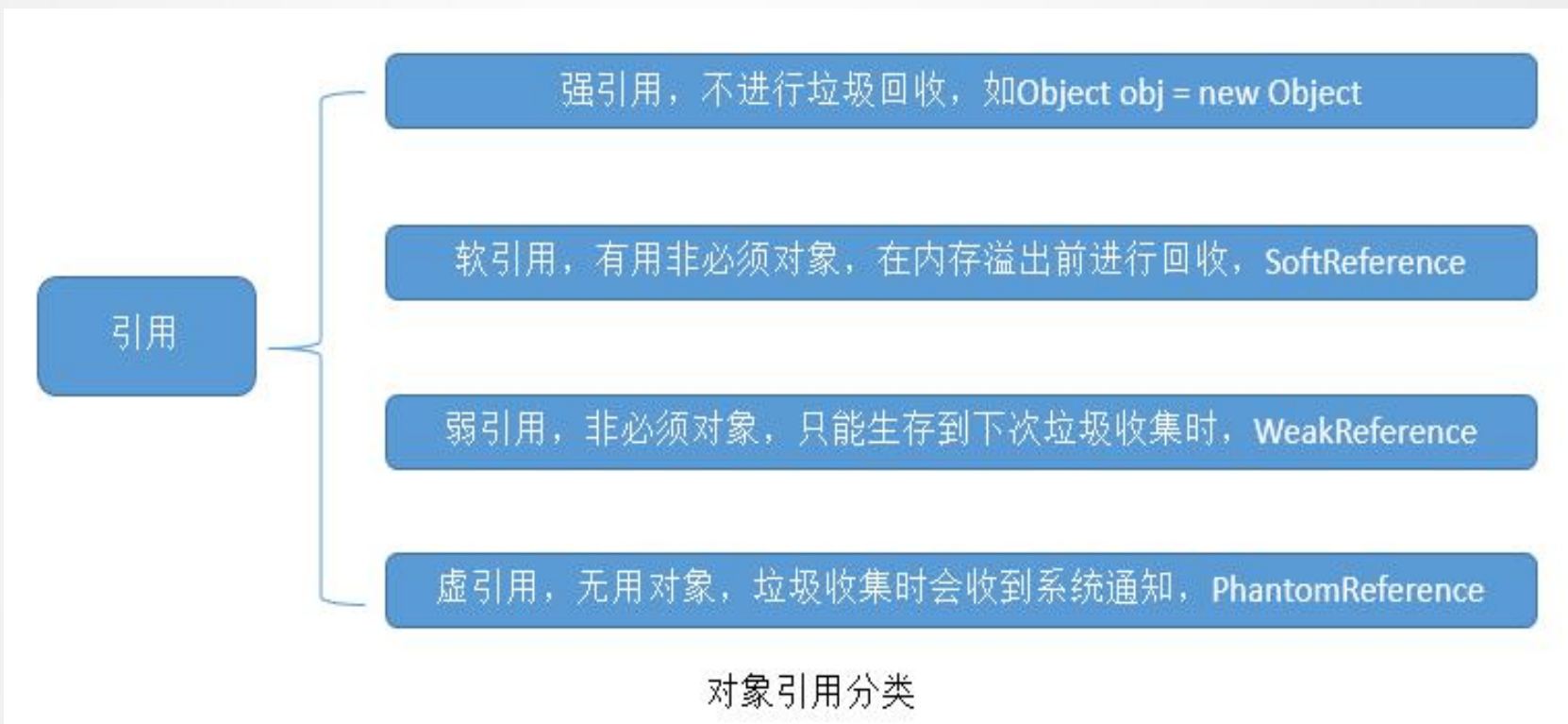
多CPU，多核

并行与并发

建立可预测的停顿时间模型



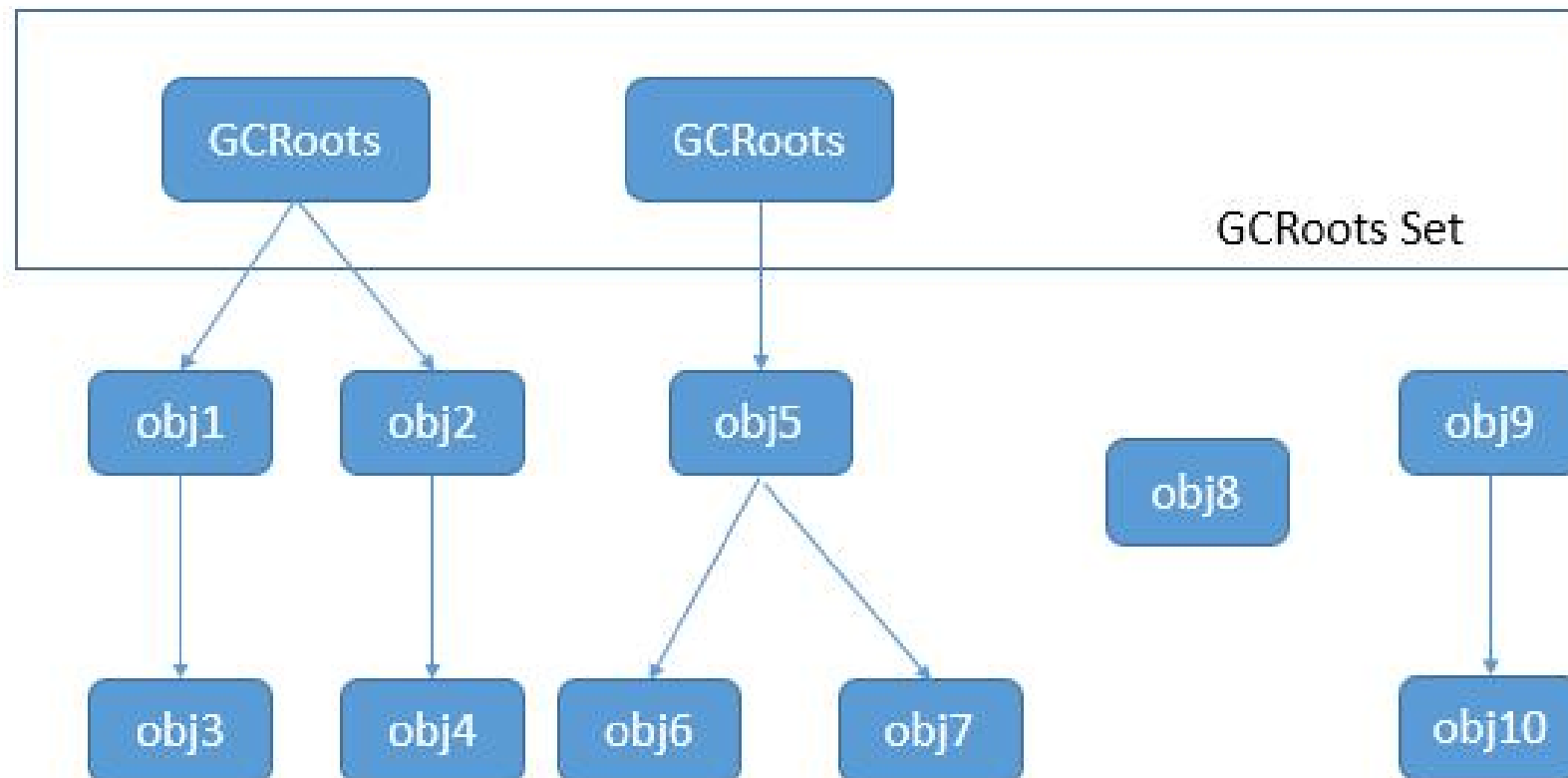
JVM不需要



JVM使用

哪行开始引起收集

```
Object a = new Object ( ) ;  
Object b = new Object ( ) ;  
Object c = new Object ( ) ;  
a = b;  
a = c;  
c = null;  
a = null;
```



1、标记-清除（Mark-Sweep）算法

标记-清除算法将垃圾回收分为两个阶段：

①.标记阶段：首先标记出所有需要回收的对象。

②.清除阶段：标记完成后，统一回收被标记的对象

缺点：

①.效率问题：标记清除过程效率都不高。

②.空间问题：标记清除之后会产生大量的不连续的内存碎片(空间碎片太多可能会导致以后在程序运行过程中需要分配较大的对象时，无法找到足够的连续的内存空间而不得不提前触发另一次垃圾收集动作。)

回收前

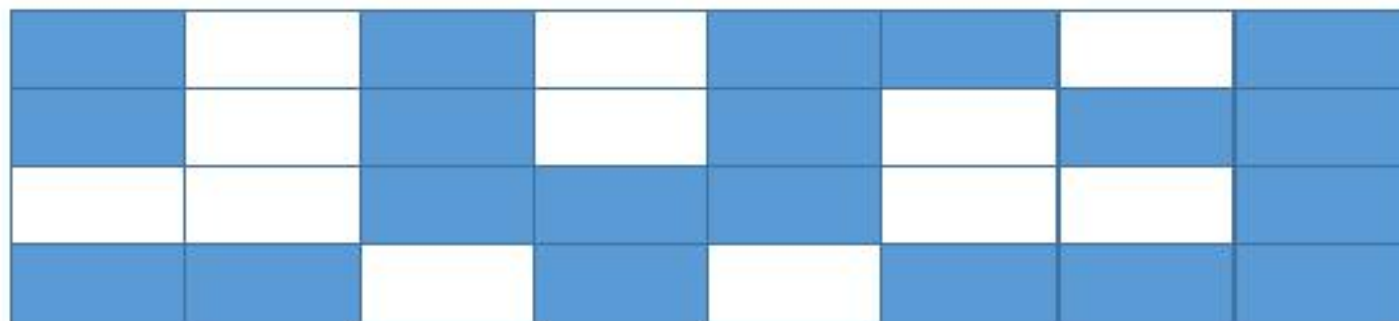


可回收

存活对象

未使用

回收后



可回收

存活对象

未使用

2、复制（Copying）算法

2.1.算法思想：

- 1).将现有的内存空间分为两块，每次只使用一块。
- 2).当其中一块用完的时候，就将还存活的对象复制到另外一块上去。
- 3).再把已使用过的内存空间一次清理掉。

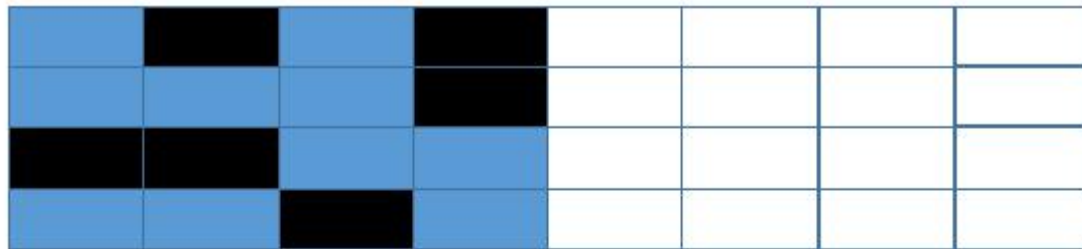
2.2.优点：

- 1).由于是每次都对整个半区进行内存回收，内存分配时不必考虑内存碎片问题。
- 2).只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

2.3.缺点：

- 1).内存减少为原来的一半，太浪费了。
- 2).对象存活率较高的时候就要执行较多的复制操作，效率变低。
- 3).如果不使用50%的对分策略，老年代需要考虑的空间担保策略。

回收前

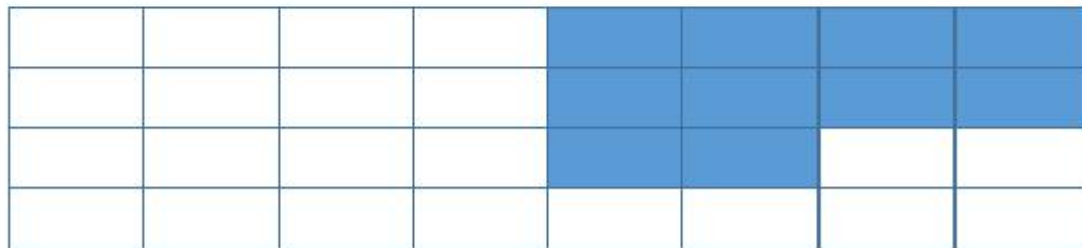


可回收

存活对象

未使用

回收后



可回收

存活对象

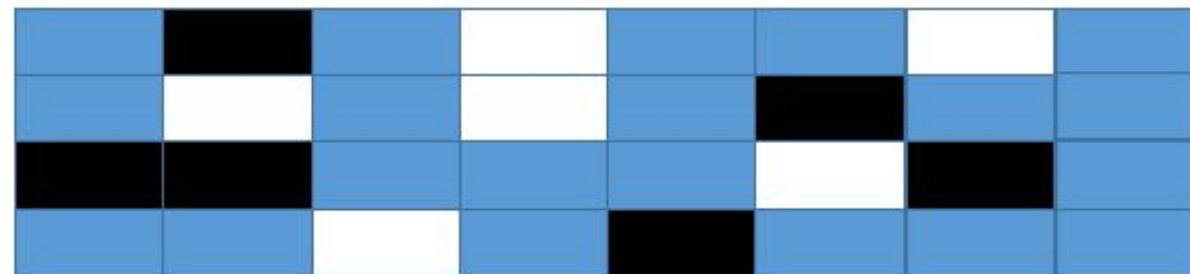
未使用

3、标记-整理（Mark-Compact）算法

- 1).标记阶段：首先标记出所有需要回收的对象。与“标记-清除”一样
- 2).让存活的对象向内存的一段移动。而不跟“标记-清除”直接对可回收对象进行清理
- 3).再清理掉边界以外的内存。

由于老年代存活率高，没有额外内存对老年代进行空间担保，那么老年代只能采用标记-清理算法或者标记整理算法。

回收前

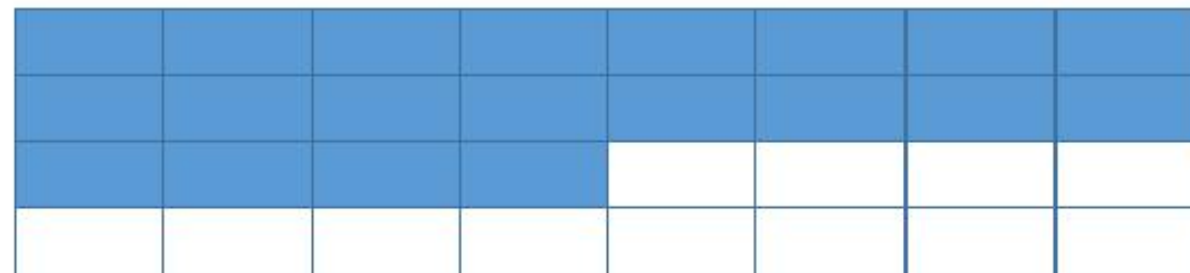


可回收

存活对象

未使用

回收后



可回收

存活对象

未使用

4、分代收集算法

以上三种算法的综合

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，选用：复制算法

在老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清除”或者“标记-整理”算法来进行回收。

- jps
- jmap
- Jstat
- Jvisualvm: window下启动远程监控, 并在被监控服务端, 启动jstatd服务。

Mark Sweep Compact GC

Heap Configuration: #堆内存初始化配置

MinHeapFreeRatio = 40 # -XX:MinHeapFreeRatio 设置JVM堆最小空闲比率

MaxHeapFreeRatio = 70 # -XX:MaxHeapFreeRatio 设置JVM堆最大空闲比率

MaxHeapSize = 100663296 (96.0MB) # -XX:MaxHeapSize= 设置JVM堆的最大大小

NewSize = 1048576 (1.0MB) # -XX:NewSize= 设置JVM堆的‘新生代’的默认大小

MaxNewSize = 4294901760 (4095.9375MB) # -XX:MaxNewSize= 设置JVM堆的‘新生代’的最大大小

OldSize = 4194304 (4.0MB) # -XX:OldSize= 设置JVM堆的‘老生代’的大小

NewRatio = 2 # -XX:NewRatio= ‘新生代’和‘老生代’的大小比率

SurvivorRatio = 8 # -XX:SurvivorRatio= 设置年轻代中Eden区与Survivor区的大小比值

PermSize = 12582912 (12.0MB) # -XX:PermSize=<value>: 设置JVM堆的‘持久代’的初始大小

MaxPermSize = 67108864 (64.0MB) # -XX:MaxPermSize=<value>: 设置JVM堆的‘持久代’的最大大小

Heap Usage:

New Generation (Eden + 1 Survivor Space): #新生代区内存分布，包含伊甸园区+1个Survivor区

capacity = 30212096 (28.8125MB)

used = 27103784 (25.848182678222656MB)

free = 3108312 (2.9643173217773438MB)

89.71169693092462% used

Eden Space: #Eden区内存分布

capacity = 26869760 (25.625MB)

used = 26869760 (25.625MB)

free = 0 (0.0MB)

100.0% used

From Space: #其中一个Survivor区的内存分布

capacity = 3342336 (3.1875MB)

used = 234024 (0.22318267822265625MB)

free = 3108312 (2.9643173217773438MB)

7.001809512867647% used

To Space: #另一个Survivor区的内存分布

capacity = 3342336 (3.1875MB)

used = 0 (0.0MB)

free = 3342336 (3.1875MB)

0.0% used

tenured generation: #当前的Old区内存分布

capacity = 67108864 (64.0MB)

used = 67108816 (63.99995422363281MB)

free = 48 (4.57763671875E-5MB)

99.99992847442627% used

Perm Generation: #当前的“持久代”内存分布

capacity = 14417920 (13.75MB)

used = 14339216 (13.674942016601562MB)

free = 78704 (0.0750579833984375MB)

99.45412375710227% used