<https://blog.csdn.net/Butterfly_resting/article/details/89705057>

**JVM 几个重要的参数**：  
-server -Xmx3g -Xms3g -XX:MaxPermSize=128m  
-XX:NewRatio=1 新生代（Eden + 2\*S）与老年代（不包括永久区）的比值  
-XX:SurvivorRatio=8 2个Survivor区和Eden区的比值  
-XX:+UseParallelGC  
-XX:ParallelGCThreads=8  
-XX:+UseParallelOldGC 这个是JAVA 6出现的参数选项  
-XX:LargePageSizeInBytes=128m 内存页的大小， 不可设置过大， 会影响Perm的大小。  
-XX:+UseFastAccessorMethods 原始类型的快速优化  
-XX:+Disable‘’ExplicitGC 关闭System.gc()

**JVM调优**

查看堆空间大小分配（年轻代、年老代、持久代分配）  
垃圾回收监控（长时间监控回收情况）  
线程信息监控：系统线程数量  
线程状态监控：各个线程都处在什么样的状态下  
线程详细信息：查看线程内部运行情况，死锁检查  
CPU热点：检查系统哪些方法占用了大量CPU时间  
内存热点：检查哪些对象在系统中数量最大  
**jvm问题排查和调优**：  
jps主要用来输出JVM中运行的进程状态信息。  
jstat命令可以用于持续观察虚拟机内存中各个分区的使用率以及GC的统计数据  
jmap可以用来查看堆内存的使用详情。  
jstack可以用来查看Java进程内的线程堆栈信息。 jstack是个非常好用的工具，结合应用日志可以迅速定位到问题线程。  
**Java性能分析工具**  
jdk会自带JMC(JavaMissionControl)工具。可以分析本地应用以及连接远程ip使用。提供了实时分析线程、内存，CPU、GC等信息的可视化界面。

**JVM内存调优**

对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数。 过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。  
使用JDK提供的内存查看工具，比如JConsole和Java VisualVM。

1）监控GC的状态，使用各种JVM工具，查看当前日志，并且分析当前堆内存快照和gc日志，根据实际的情况看是否需要优化。  
2）通过JMX的MBean或者Java的jmap生成当前的Heap信息，并使用Visual VM或者Eclipse自带的Mat分析dump文件  
3）如果参数设置合理，没有超时日志，GC频率GC耗时都不高则没有GC优化的必要，如果GC时间超过1秒或者频繁GC，则必须优化  
4）调整GC类型和内存分配，使用1台和多台机器进行测试，进行性能的对比。再做修改，最后通过不断的试验和试错，分析并找到最合适的参数

**导致Full GC一般由于以下几种情况**：  
旧生代空间不足  
调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象  
**新生代设置过小**  
 一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC  
**2). 新生代设置过大**  
,.  
一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加  
**3). Survivor设置过小**  
导致对象从eden直接到达旧生代  
**4). Survivor设置过大**  
导致eden过小，增加了GC频率  
一般说来新生代占整个堆1/3比较合适

**GC策略的设置方式**  
1). 吞吐量优先 可由-XX:GCTimeRatio=n来设置  
2). 暂停时间优先 可由-XX:MaxGCPauseRatio=n来设置

## JVM内存管理：

1.JVM内存区域的划分：  
**程序计数器**（PC，Program Counter Register）。在 JVM 规范中，每个线程都有它自己的程序计数器，并且任何时间一个线程都只有一个方法在执行，也就是所谓的当前方法。程序计数器会存储当前线程正在执行的 Java 方法的 JVM 指令地址；或者，如果是在执行本地方法，则是未指定值（undefined）。（唯一不会抛出OutOfMemoryError）  
**第二，Java 虚拟机栈**（Java Virtual Machine Stack），早期也叫 Java 栈。每个线程在创建时都会创建一个虚拟机栈，其内部保存一个个的栈帧（Stack Frame），对应着一次次的 Java 方法调用。  
前面谈程序计数器时，提到了当前方法；同理，在一个时间点，对应的只会有一个活动的栈帧，通常叫作当前帧，方法所在的类叫作当前类。如果在该方法中调用了其他方法，对应的新的栈帧会被创建出来，成为新的当前帧，一直到它返回结果或者执行结束。JVM 直接对 Java 栈的操作只有两个，就是对栈帧的压栈和出栈。  
栈帧中存储着局部变量表、操作数（operand）栈、动态链接、方法正常退出或者异常退出的定义等。  
**第三，堆（Heap）**，它是 Java 内存管理的核心区域，用来放置 Java 对象实例，几乎所有创建的 Java 对象实例都是被直接分配在堆上。堆被所有的线程共享，在虚拟机启动时，我们指定的“Xmx”之类参数就是用来指定最大堆空间等指标。  
（ 编译器通过逃逸分析，确定对象是在栈上分配还是在堆上分配）  
理所当然，堆也是垃圾收集器重点照顾的区域，所以堆内空间还会被不同的垃圾收集器进行进一步的细分，最有名的就是新生代、老年代的划分。  
**第四，方法区（Method Area）**。这也是所有线程共享的一块内存区域，用于存储所谓的元（Meta）数据，例如类结构信息，以及对应的运行时常量池、字段、方法代码等。  
由于早期的 Hotspot JVM 实现，很多人习惯于将方法区称为永久代（Permanent Generation）。Oracle JDK 8 中将永久代移除，同时增加了元数据区（Metaspace）。  
**第五，运行时常量池**（Run-Time Constant Pool），这是方法区的一部分。如果仔细分析过反编译的类文件结构，你能看到版本号、字段、方法、超类、接口等各种信息，还有一项信息就是常量池。Java 的常量池可以存放各种常量信息，不管是编译期生成的各种字面量，还是需要在运行时决定的符号引用，所以它比一般语言的符号表存储的信息更加宽泛。  
**第六，本地方法栈**（Native Method Stack）。它和 Java 虚拟机栈是非常相似的，支持对本地方法的调用，也是每个线程都会创建一个。在 Oracle Hotspot JVM 中，本地方法栈和 Java 虚拟机栈是在同一块儿区域，这完全取决于技术实现的决定，并未在规范中强制。

**ＧＣ调优**：  
基本的调优思路可以总结为：  
１.理解应用需求和问题，确定调优目标。评估用户可接受的响应时间和业务量，将目标简化为，希望 GC 暂停尽量控制在 200ms 以内，并且保证一定标准的吞吐量。  
２.掌握 JVM 和 GC 的状态，定位具体的问题，确定真的有 GC 调优的必要。比如，通过 jstat 等工具查看 GC 等相关状态，可以开启 GC 日志，或者是利用操作系统提供的诊断工具等  
３.选择的 GC 类型是否符合我们的应用特征，如果是，具体问题表现在哪里，是 Minor GC 过长  
４.通过分析确定具体调整的参数或者软硬件配置。  
５.验证是否达到调优目标，如果达到目标，即可以考虑结束调优；否则，重复完成分析、调整、验证这个过程。

\*\*\* GC日志分析\*\*

* 调优命令
* 调优工具

**调优命令**  
Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

* jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。
* jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。
* jmap，JVM Memory Map命令用于生成heap dump文件
* jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看
* jstack，用于生成java虚拟机当前时刻的线程快照。
* jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

**调优工具**  
常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

* jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控

## GC触发的条件有两种

（1）程序调用System.gc时可以触发；（2）系统自身来决定GC触发的时机。

要完全回收一个对象，至少需要经过两次标记的过程。  
第一次标记：对于一个没有其他引用的对象，筛选该对象是否有必要执行finalize()方法，如果没有执行必要，则意味可直接回收。（筛选依据：是否复写或执行过finalize()方法；因为finalize方法只能被执行一次）。  
第二次标记：如果被筛选判定位有必要执行，则会放入FQueue队列，并自动创建一个低优先级的finalize线程来执行释放操作。如果在一个对象释放前被其他对象引用，则该对象会被移除FQueue队列。

## Minor GC ，Full GC 触发条件

Minor GC触发条件：当Eden区满时，触发Minor GC。  
Full GC触发条件：  
（1）调用System.gc时，系统建议执行Full GC，但是不必然执行  
（2）老年代空间不足  
（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存  
（5）由Eden区、From Space区向To Sp3ace区复制时，对象大小大于To Space可存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

## jvm怎样判断一个对象是否可回收，怎样的对象才能作为GC root

1)在Java中采取了 可达性分析法  
通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。  
2)虚拟机栈中引用的对象、方法区类静态属性引用的对象、方法区常量池引用的对象、本地方法栈JNI引用的对象

## OOM说一下？怎么排查？哪些会导致OOM? OOM出现在什么时候

OOM，全称“Out Of Memory”，官方说明：当JVM因为没有足够的内存来为对象分配空间并且垃圾回收器也已经没有空间可回收时，就会抛出这个error。  
（没有空闲内存，并且垃圾收集器也无法提供更多内存。）  
怎么排查？  
首先可以查看服务器运行日志以及项目记录的日志，捕捉到内存溢出异常。  
核心系统日志文件  
**OOM出现在什么时候？哪些会导致OOM?**  
java堆内存溢出，此种情况最常见，一般由于内存泄露或者堆的大小设。置不当引起。 可以通过虚拟机参数-Xms,-Xmx等修改。  
（1）java永久代溢出，即方法区溢出了，因为永久代的大小是有限的，并且 JVM 对永久代垃圾回收（如，常量池回收、卸载不再需要的类型）非常不积极，所以当我们不断添加新类型的时候，永久代出现 OutOfMemoryError 也非常多见 ，尤其是在运行时存在大量动态类型生成的场合；（ＪＤＫ　８　已经没有方法区了，改为元数据区）  
（2）JAVA虚拟机栈溢出，不会抛OOM error，一般是由于程序中存在死循环或者深度递归调用造成的，栈大小设置太小也会出现此种溢出。可以通过虚拟机参数-Xss来设置栈的大小。程序不断的进行递归调用，而且没有退出条件，就会导致不断地进行压栈。类似这种情况，JVM 实际会抛出 StackOverFlowError；当然，如果 JVM 试图去扩展栈空间的的时候失败，则会抛出 OutOfMemoryError。  
（３）直接内存不足，也会导致 OOM

## 什么是Full GC？GC? major GC? stop the world

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC。  
Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC 是清理整个堆空间—包括年轻代和老年代。  
Full GC触发条件：  
（1）调用System.gc时，系统建议执行Full GC，但是不必然执行  
（2）老年代空间不足  
（3）方法去空间不足  
（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存  
（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

GC，即就是Java垃圾回收机制。GC触发的条件有两种。（1）程序调用System.gc时可以触发；（2）系统自身来决定GC触发的时机。  
两次标记的过程。  
第一次标记：对于一个没有其他引用的对象，筛选该对象是否有必要执行finalize()方法，如果没有执行必要，则意味可直接回收。（筛选依据：是否复写或执行过finalize()方法；因为finalize方法只能被执行一次）。  
第二次标记：如果被筛选判定位有必要执行，则会放入FQueue队列，并自动创建一个低优先级的finalize线程来执行释放操作。如果在一个对象释放前被其他对象引用，则该对象会被移除FQueue队列。

当程序运行到这些“安全点”(方法调用,循环跳转,异常跳转)的时候就会暂停所有当前运行的线程（Stop The World 所以叫STW）。  
在GC发生时，直接把所有线程都挂起，然后检测所有线程是否都在安全点，如果不在安全点则恢复线程的执行，等执行到安全点再挂起。  
VM会设置一个标志，当线程执行到安全点的时候会轮询检测这个标志，如果发现需要GC，则线程会自己挂起，直到GC结束才恢复运行

## 如何判断是否有内存泄露？

泄露可以对比不同时间点内存分配，一般看用户类型的分配情况，什么在增加。具体，比如用jmap -histo:live 多次快照，然后对比差异，或者用jmc之类profiling工具，都可以进行，对比会更加流畅一些  
**定位 Full GC 发生的原因，有哪些方式？**  
1,首先通过printgcdetail 查看fullgc频率以及时长  
2,通过dump 查看内存中哪些对象多，这些可能是引起fullgc的原因，看是否能优化  
3,如果堆大或者是生产环境，可以开起jmc 飞行一段时间，查看这期间的相关数据来订位问题

【2】

<https://www.jianshu.com/p/49e76079243d>

【3】

## 13 张图解 Java 中的内存模型

**前言**

* 了解Java中的对象、变量等存放的内存区域十分重要
* 本文将全面讲解Java虚拟机中的内存模型 & 分区，希望你们会喜欢

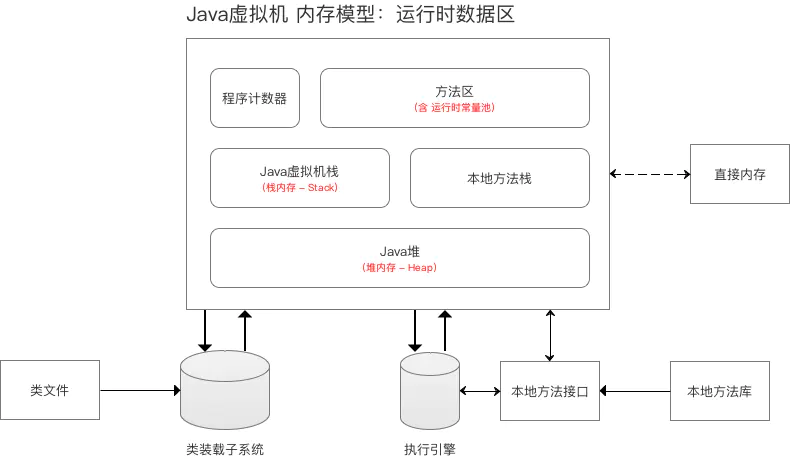
# ****目录****

# ****1. 内存模型 & 分区****

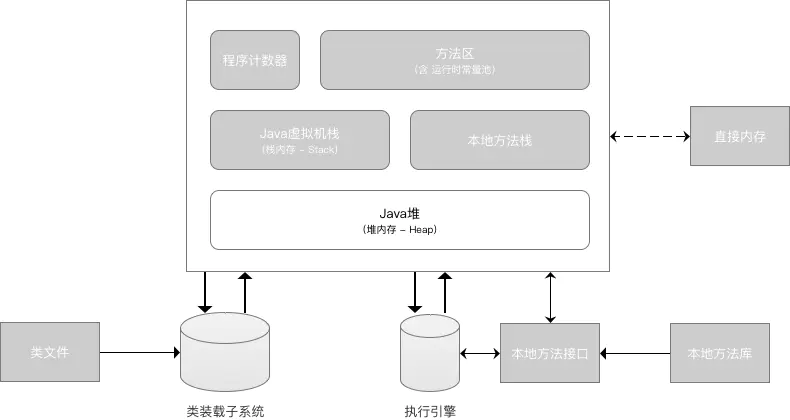
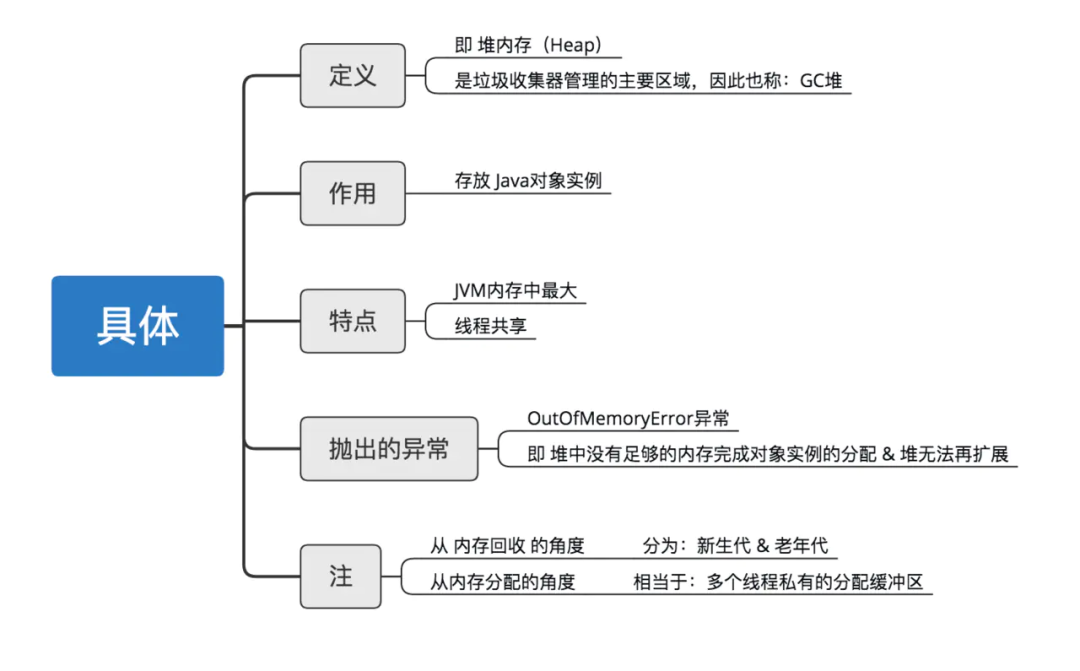
Java虚拟机在运行Java程序时，会管理着一块内存区域：运行时数据区

在运行时数据区里，会根据用途进行划分：

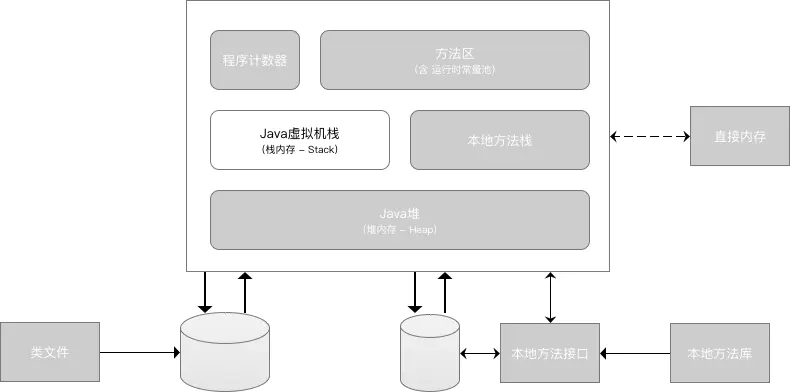
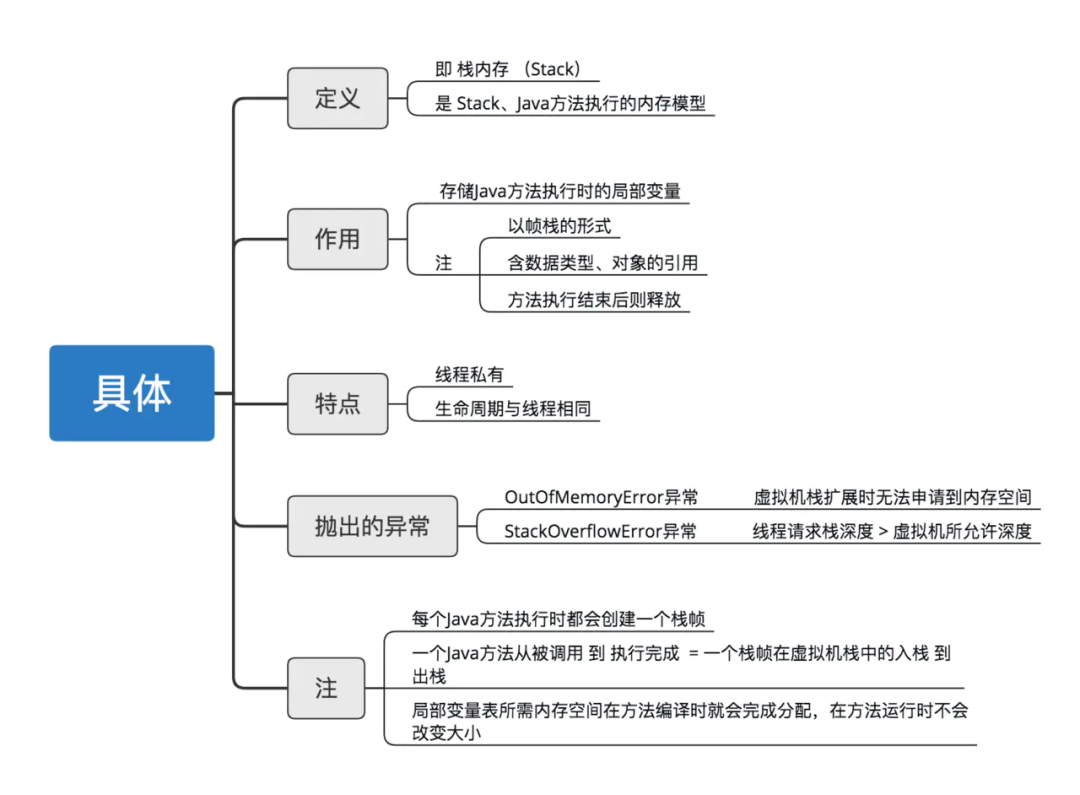
1. Java虚拟机栈（栈区）
2. 本地方法栈
3. Java堆（堆区）
4. 方法区
5. 程序计数器

  
下面，我将详细介绍每个内存模型分区

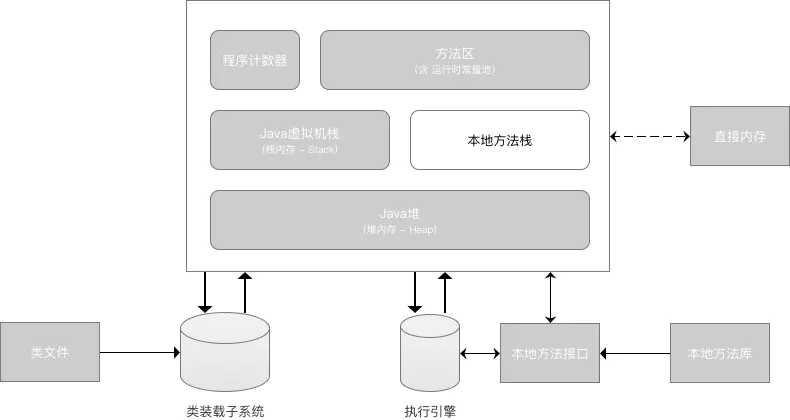
# ****2. Java堆****

简介

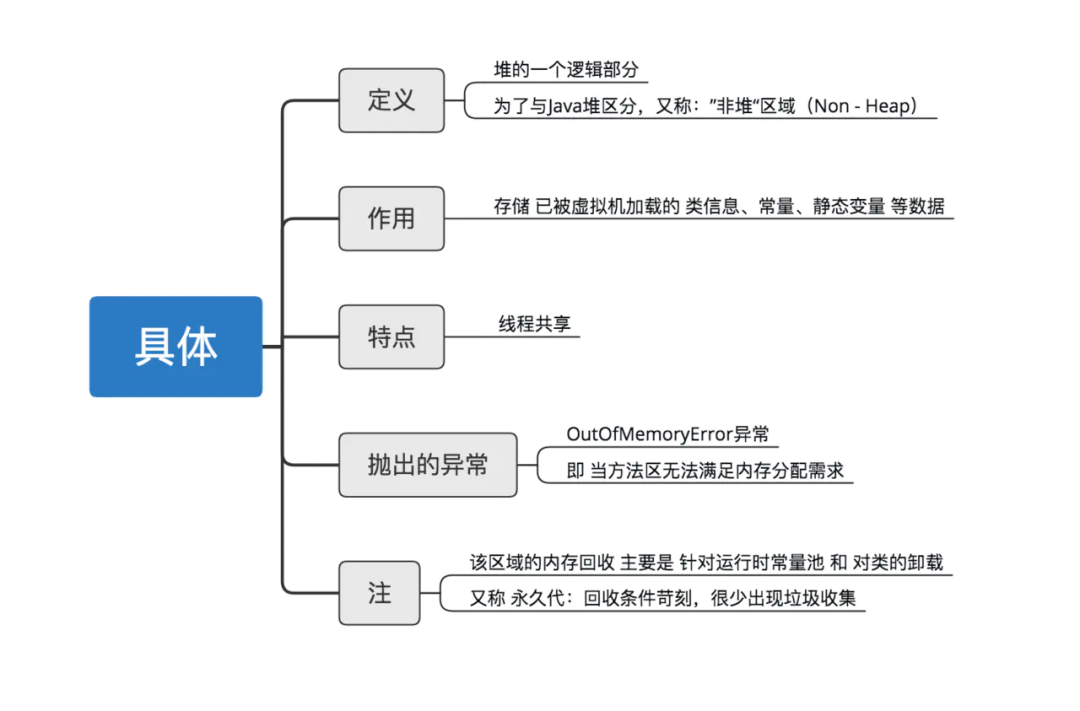
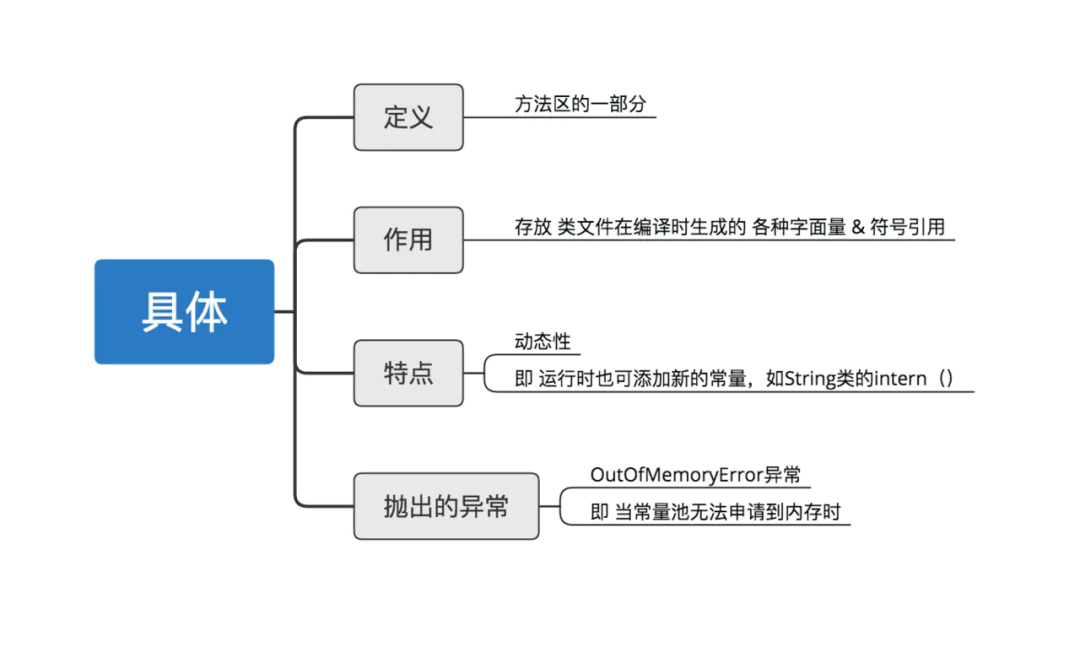
# ****3. Java虚拟机栈****

  
简介

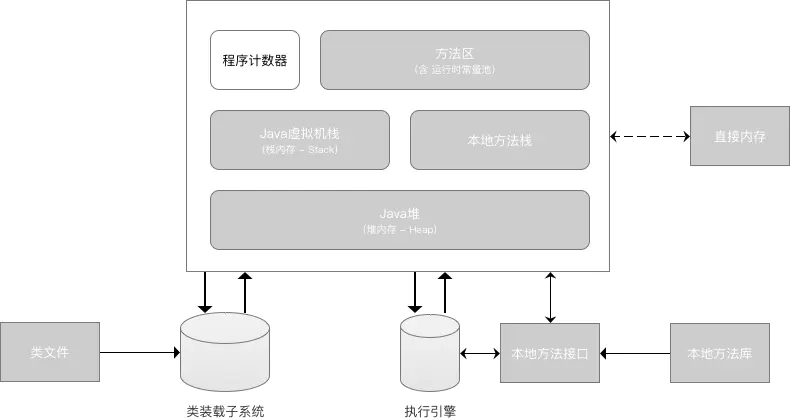
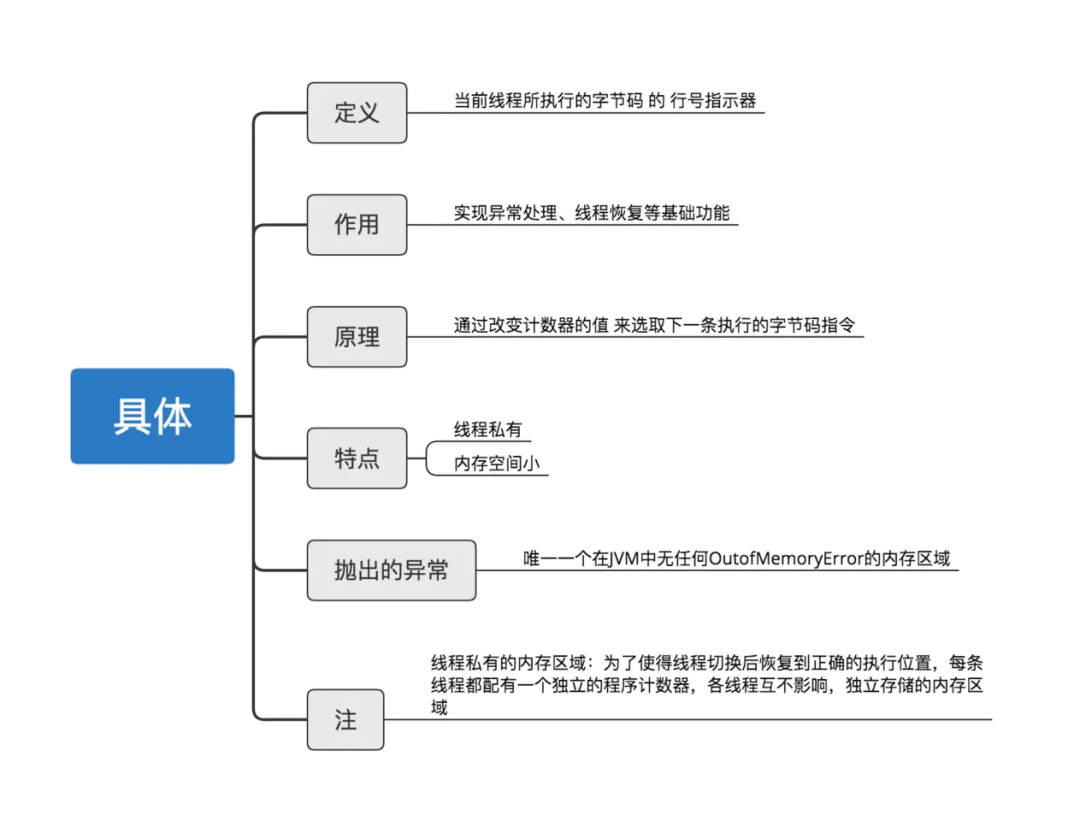
# ****4. 本地方法栈****

  简介：十分类似Java虚拟机栈，与Java虚拟机区别在于：服务对象，即Java虚拟机栈为执行 Java 方法服务；本地方法栈为执行 Native方法服务

# ****5. 方法区****

简介注：其内部包含一个运行时常量池，具体介绍如下：

# ****6. 程序计数器****

简介

# ****7. 额外知识：直接内存****

* 定义：NIO类（JDK1.4引入）中基于通道和缓冲区的I/O方式 通过使用Native函数库 直接分配 的堆外内存
* 特点：不受堆大小限制

不属于虚拟机运行时数据区的一部分 & 不在堆中分配

* 应用场景：适用于频繁调用的场景

通过一个 存储在Java堆中的DirectByteBuffer对象 作为这块内存的引用 进行操作，从而避免在 Java 堆和 Native堆之间来回复制数据，提高使用性能

* 抛出的异常：OutOfMemoryError，即与其他内存区域的总和 大于 物理内存限制

# ****8. 总结****

本文全面讲解JVM中的内存模型 & 分区，总结如下



作者：Carson\_Ho

链接：juejin.im/post/6844903677279338509

**热门内容：**

* [昨天还在 for 循环里写加号拼接字符串的那个同事，今天已经不在了](http://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247492884&idx=1&sn=f820a857ffe34a3f8a4246b9fd47bc23&chksm=9beec466ac994d70134d16116a5a0c5ec3fd97b8e9cc2926923f63b949cb3801aaf55b602bac&scene=21#wechat_redirect)
* [推荐个 Java 开源商城项目，这个是真的好！](http://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247492865&idx=1&sn=2c1f894fc0aa9fc78390792af4fe6505&chksm=9beec473ac994d6571e7b1e8bf9f8d2161220467831413e2814094b7d5c5be040e9a25bdaa8d&scene=21#wechat_redirect)
* [记一次订单号重复的事故，快看看你的 uuid 在并发下还正确吗？](http://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247492848&idx=1&sn=b33a3b81b78a4dbdb95916a8368c5897&chksm=9beec582ac994c946dd8041f19402a76c664adf46ac734e78029a9415d202f1e6fbde5051d7a&scene=21#wechat_redirect)
* [为什么程序员都不喜欢使用 switch ，而是大量的 if……else if ？](http://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247492832&idx=2&sn=e93641d51b383b98a101a5b79f7b4519&chksm=9beec592ac994c84cbdfd3604306b0202587c2fa9db3d5d0957a4a5fce98d3ca303492317bab&scene=21#wechat_redirect)
* [终于有人把 Docker 讲清楚了，别再说不会 Docker 了！](http://mp.weixin.qq.com/s?__biz=MzAxNjk4ODE4OQ==&mid=2247492818&idx=2&sn=0fcc730fde0dd1af495845d5bc07073e&chksm=9beec5a0ac994cb67f157437623cc8c275db0a4979329bd2a02cae43e8fb671d96921232d487&scene=21#wechat_redirect)

最近面试BAT，整理一份面试资料《**Java面试BAT通关手册**》，覆盖了Java核心技术、JVM、Java并发、SSM、微服务、数据库、数据结构等等。

获取方式：点“**在看**”，关注公众号并回复 **666** 领取，更多内容陆续奉上。

**明天见(｡･ω･｡)**

阅读 323

赞在看1



写下你的留言