目录

[一．集合](#_Toc13264_WPSOffice_Level1) [2](#_Toc13264_WPSOffice_Level1)

[二．数据结构](#_Toc25301_WPSOffice_Level1) [4](#_Toc25301_WPSOffice_Level1)

[三． Java流](#_Toc22013_WPSOffice_Level1) [4](#_Toc22013_WPSOffice_Level1)

[四． 线程问题 \*（1/2）](#_Toc27149_WPSOffice_Level1) [6](#_Toc27149_WPSOffice_Level1)

[五． 并发问题 \*（1/2）](#_Toc28086_WPSOffice_Level1) [8](#_Toc28086_WPSOffice_Level1)

[六． 泛型问题](#_Toc18910_WPSOffice_Level1) [8](#_Toc18910_WPSOffice_Level1)

[七． Redis问题 \*](#_Toc19438_WPSOffice_Level1) [8](#_Toc19438_WPSOffice_Level1)

[八． 中间件MQ问题 \*](#_Toc24147_WPSOffice_Level1) [8](#_Toc24147_WPSOffice_Level1)

[九． FastDFS问题](#_Toc2262_WPSOffice_Level1) [9](#_Toc2262_WPSOffice_Level1)

[十． Spring 问题\*](#_Toc8675_WPSOffice_Level1) [9](#_Toc8675_WPSOffice_Level1)

[十一． Spring boot综合问题 \*](#_Toc5102_WPSOffice_Level1) [10](#_Toc5102_WPSOffice_Level1)

[十二． Spring Cloud问题 \*](#_Toc12649_WPSOffice_Level1) [10](#_Toc12649_WPSOffice_Level1)

[十三． Mybatis问题 \*](#_Toc15505_WPSOffice_Level1) [11](#_Toc15505_WPSOffice_Level1)

[十四． 前端Vue框架问题](#_Toc29951_WPSOffice_Level1) [11](#_Toc29951_WPSOffice_Level1)

[十五． 数据库问题 \*](#_Toc28666_WPSOffice_Level1) [11](#_Toc28666_WPSOffice_Level1)

[十六． 管理工具问题（SVN，Git）](#_Toc15465_WPSOffice_Level1) [13](#_Toc15465_WPSOffice_Level1)

[十七． Linux常用指令](#_Toc24678_WPSOffice_Level1) [13](#_Toc24678_WPSOffice_Level1)

[十八． 项目相关问题\*](#_Toc21567_WPSOffice_Level1) [13](#_Toc21567_WPSOffice_Level1)

[十九． JVM底层问题\*](#_Toc9783_WPSOffice_Level1) [13](#_Toc9783_WPSOffice_Level1)

一．集合

1. Conllection

List

ArrayList √

底层数组实现，初始长度0第一次添加增长为10，超出后已1.5倍增长（原数组+原数组>>1）,通过创建新数组，复制到新数组的方式进行扩容。检索速度快，增删慢（因为需要移动数组）。线程不安全。

LinkedList √

底层双向链表实现，特性类似双向链表，检索较慢需要遍历，但是增删快，线程不安全。

Vector（过时） √

线程安全，底层也是数组。

它的方法上都加了synchronized 关键字。

Set

HashSet √

基于hashmap实现使用map的可以进行存储，不允许重复元素，是无序的，允许null值

LinkedHashSet √

有序的基于linkedHashMap

TreeSet

基于treemap来实现的

1. Map

Hashmap √

底层数组链表红黑树初始长度是16加载因子0.75以两倍长度扩容（红黑树是1.8后才加入的为了在hash碰撞超过8时查询速率降低而改变的，防止链表被hash攻击导致速率太低，其实发生8次碰撞的概率很低了，可以达到十万分之一以上）

LinkedHashMap √

底层原理HashMap加上双向链表，节点过出before和after属性来负责前后加入的节点，heard属性固定头节点，从尾节点添加。

ConcurrentHashMap √

最开始使用hashtable但是公用一把锁效率低下，后使用ConcurrentHashMap分段锁，分成每段数据加一把锁，在多线程时减少锁竞争。其中包含Segment（桶）数组分段数组，也包含链表。

在1.8中有改变，取消了Segment字段，1.7中按段进行加锁1.8是行进行加锁（也就是对应一个hash地址加一把锁），也就是按照每个链表的头结点进行加锁

TreeMap

底层原理是红黑树。

<https://mp.weixin.qq.com/s/oNmUW-rUbTPPgUbFn9oddQ>

二．数据结构

1.数组 √

2.链表 √

3.树 √

红黑树学习连接：

<https://mp.weixin.qq.com/s/3_1oGgNrOvG3Qwh0lZPyOg>

4.哈希表 √

5.图（特定业务场景）

6.堆（基于二叉平衡树）

7.排序算法√（十大经典排序）

1. Java流

详解链接：

1. <https://mp.weixin.qq.com/s/mEahtWqeFqzzaETHKAWtzw>
2. AIO流 \*

NIO的升级，异步非阻塞流

1. NIO流 \*（需要熟悉Reactor模式和多线程-->盲区）

<https://mp.weixin.qq.com/s/jgC5cUwQtOZQgi0pLzoqxw>

<https://mp.weixin.qq.com/s/l8hodrbiBO_XHx8Jtbmpyw>

同步非阻塞流，相比于BIO流 一次处理一个字节数据，BI面向流，而NIO 一次处理一个数据块，面向块（或者说面向缓冲区）。NIO主要组成有选择器Selector、通道Channel和缓冲区Buffer。

Selector（Reactor模式）：

Selector运行单线程处理多个Channel，如果你的应用打开了多个通道，但每个连接的流量都很低，使用Selector就会很方便

通道Channel类型：

FileChannel：从文件中读写数据；

DatagramChannel：通过 UDP 读写网络中数据；

SocketChannel：通过 TCP 读写网络中数据；

ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。

缓冲区：发送给一个通道的所有数据都必须首先放到缓冲区中，同样地，从通道中读取的任何数据都要先读到缓冲区中。也就是说，不会直接对通道进行读写数据，而是要先经过缓冲区。

缓冲区实质上是一个数组，但它不仅仅是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。

类型：

ByteBuffer

CharBuffer

ShortBuffer

IntBuffer

LongBuffer

FloatBuffer

DoubleBuffer

缓冲区状态变量：

capacity：最大容量；

position：当前已经读写的字节数；

limit：还可以读写的字节数。

I/O用到了装饰者模式附带连接装饰者与代理模式的区别：

<https://blog.csdn.net/andong154564667/article/details/80258061>

1. 线程问题 \*（1/2）

三种创建方式：继承Thread，实现Runable，使用线程池（实际开发中最为常用）

使用线程池的目的：

1. 线程是稀缺资源，不能频繁的创建
2. 解耦作用，线程创建于执行完全分开，方便维护
3. 应当将其放进一个池子中，可以给其他任务进行复用

线程池：

<https://mp.weixin.qq.com/s/uxM2vMJghOOySbLa5BKoPA>

<https://mp.weixin.qq.com/s/qyRWTx47jpqtLh7QPfRNOw>

<https://mp.weixin.qq.com/s/ncjOig-BDKP4pv-PfT-mqA>

创建方式：

Executors.newCachedThreadPool()：无限线程池（缓存线程池）。

Executors.newFixedThreadPool(nThreads)：创建固定大小的线程池。

Executors.newSingleThreadExecutor()：创建单个线程的线程池。

创建参数解释：

public static ExecutorService newCachedThreadPool() {

return new ThreadPoolExecutor (0, Integer.MAX\_VALUE,

60L,TimeUnit.SECONDS, newSynchronousQueue<Runnable>());

}

ThreadPoolExecutor (int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)

corePoolSize 为线程池的基本大小。

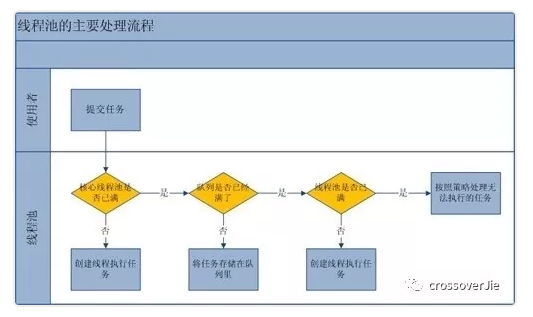
maximumPoolSize 为线程池最大线程大小。

keepAliveTime 和 unit（时间单位） 则是线程空闲后的存活时间。

workQueue 用于存放任务的阻塞队列。

handler 当队列和最大线程池都满了之后的饱和策略（拒绝策略）。

线程池执行流程：



1. 并发问题 \*（1/2）
2. 泛型问题
3. Redis问题 \*

类型：String，Hash，List，Set，SortSet，

BItmaps（位图），HyperLoglog，GEO（地理信息定位）

可实现的功能：

1.缓存：毫无疑问这是Redis当今最为人熟知的使用场景。再提升服务器性能方面非常有效；

2.排行榜：如果使用传统的关系型数据库来做这个事儿，非常的麻烦，而利用Redis的SortSet数据结构能够非常方便搞定；

3.计算器/限速器：利用Redis中原子性的自增操作，我们可以统计类似用户点赞数、用户访问数等，这类操作如果用MySQL，频繁的读写会带来相当大的压力；限速器比较典型的使用场景是限制某个用户访问某个API的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力；

4.好友关系：利用集合的一些命令，比如求交集、并集、差集等。可以方便搞定一些共同好友、共同爱好之类的功能；

5.简单消息队列：除了Redis自身的发布/订阅模式，我们也可以利用List来实现一个队列机制，比如：到货通知、邮件发送之类的需求，不需要高可靠，但是会带来非常大的DB压力，完全可以用List来完成异步解耦；

6.Session共享：以PHP为例，默认Session是保存在服务器的文件中，如果是集群服务，同一个用户过来可能落在不同机器上，这就会导致用户频繁登陆；采用Redis保存Session后，无论用户落在那台机器上都能够获取到对应的Session信息。

Redis为什么速度那么快：

1. 操作对象时内存，不需要加载到内存之中，
2. 单线程操作，（没有CPU竞争时间，线程切换时间）
3. 采用I/O多路复用技术，一个线程管理多个IO通道
4. 中间件MQ问题 \*

<https://mp.weixin.qq.com/s/tiN6Z_VjTKZ5voFu4TFElg>

<https://mp.weixin.qq.com/s/DsowfyzYXcD-OyiL1JOFZw>

<https://mp.weixin.qq.com/s/ZAWPRToPQFcwzHBf47jZ-A>

<https://mp.weixin.qq.com/s/AEn3j2lVJOHZx9yegwTsvwKafka>

需要整合zookeeper框架，特点：高吞吐低延迟高并发，高性能的消息中间件，设置可以做到几十上百万的找高并发。

其原理流程：接收到的数据会往磁盘上写，但是在这个过程中会先写到os cache缓存中在刷到磁盘上，读取数据是kafka有个叫零拷贝技术（不用将数据读到应用程序缓存中，再读到socket缓存，再发送到网卡入喉由网卡发出），零拷贝技术会从OScache缓存中直接将数据读取到网卡，这样效率就增加了许多，再此之前会先去看数据是否在OScache中存在，存在直接读取，不存在就从磁盘中读取

1. FastDFS问题
2. Spring 问题\*

Spring

AOP：

<https://mp.weixin.qq.com/s/lk42ijGVXBPZht4teJ8HWQ>

是一种切面编程，底层动态代理JDK和cglib实现。代理分两种动态代理静态和动态代理，静态代理需要自己去创建代理类，动态代理不需要，JDK动态代理，被代理类需要实现接口，而cglib通过继承来实现不需要被代理类实现接口，单例模式建议使用cglib，其他的则选用JDK。

控制反转IOC：

<https://mp.weixin.qq.com/s/f_XEMSCFKtPXFLZLlLXnjQ>

大概实现原理：工厂模式加反射，他有两个容器BeanFactory和ApplicationContext（高级容器）

1. Spring boot综合问题 \*

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置

1. Spring Cloud问题 \*

Spring Cloud是一系列框架的有序集合。它利用[Spring Boot](https://baike.baidu.com/item/Spring Boot/20249767" \t "https://baike.baidu.com/item/_blank)的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

Spring Cloud Netflix（可关注微信公众号：程序员私房菜 里面有讲解）

<https://mp.weixin.qq.com/s/DabOWe0GKY-HU49jtWvdTw>

是对Netflix开发的一套分布式服务框架的封装，包括服务的发现和注册（eureka），负载均衡（ribbon）、断路器（Hystrix ）、REST客户端（feign）、请求路由等（zuul）。

Spring Cloud Config

　　将配置信息中央化保存, 配置Spring Cloud Bus可以实现动态修改配置文件

Spring Cloud Bus

　　分布式消息队列，是对Kafka, MQ的封装

Spring Cloud Security

　　对Spring Security的封装，并能配合Netflix使用

Spring Cloud Zookeeper

　　对Zookeeper的封装，使之能配置其它Spring Cloud的子项目使用

Spring Cloud Eureka

Spring Cloud Eureka 是 Spring Cloud Netflix 微服务套件中的一部分，它基于Netflix Eureka 做了二次封装，主要负责完成微服务架构中的服务治理功能。

1. Mybatis问题 \*
2. 前端Vue框架问题
3. 数据库问题MySQL \*

读多写少用缓存，写多读少用缓冲！

1. 事务

事务具有四个特征：原子性（ Atomicity ）、一致性（ Consistency ）、隔离性（ Isolation ）和持续性（ Durability ）。这四个特性简称为 ACID 特性。

原子性：事务是数据库的逻辑工作单位，事务中包含的各操作要么都做，要么都不做

一致性：事 务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。因此当数据库只包含成功事务提交的结果时，就说数据库处于一致性状态。如果数据库系统 运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态，或者说是 不一致的状态。

隔离性：一个事务的执行不能其它事务干扰。即一个事务内部的操作及使用的数据对其它并发事务是隔离的，并发执行的各个事务之间不能互相干扰。

持续性：也称永久性，指一个事务一旦提交，它对数据库中的数据的改变就应该是永久性的。接下来的其它操作或故障不应该对其执行结果有任何影响。

1. 隔离级别

Read Uncommitted（读取未提交内容也称之为脏读）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。

Read Committed（读取提交内容也称之为幻读）

这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。

Repeatable Read（可重读）

这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题。

Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

1. 三大范式

第一范式：一范式（1NF）是指在[关系模型](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%A8%A1%E5%9E%8B" \t "https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%8C%83%E5%BC%8F/_blank)中，对于添加的一个规范要求，所有的域都应该是原子性的，即数据库表的**每一列都是不可分割的原子数据项。**

第二范式：第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）**要求实体的属性完全依赖于主关键字**

第三范式：第三范式（3NF）是第二范式（2NF）的一个子集，即满足第三范式（3NF）必须满足第二范式（2NF）在实体中不存在非主键属性传递函数依赖于主键属性。（表中字段[非主键]不存在对主键的传递依赖）

1. 为什么要优化数据库

1、系统吞吐量瓶颈往往是数据库访问速度造成的。

2、随着应用程序的运行，数据库的数据越俩越多，处理速度相应变慢。

3、数据库存放在磁盘上，读写速度无法与内存相比。

5.如何优化数据库

1、设计数据库时：数据库表，字段的设计，存储引擎。

2、利用好MySQL自身提供的功能，如索引等。

3、横向扩展：MySQL集群，负载均衡，读写分离等。

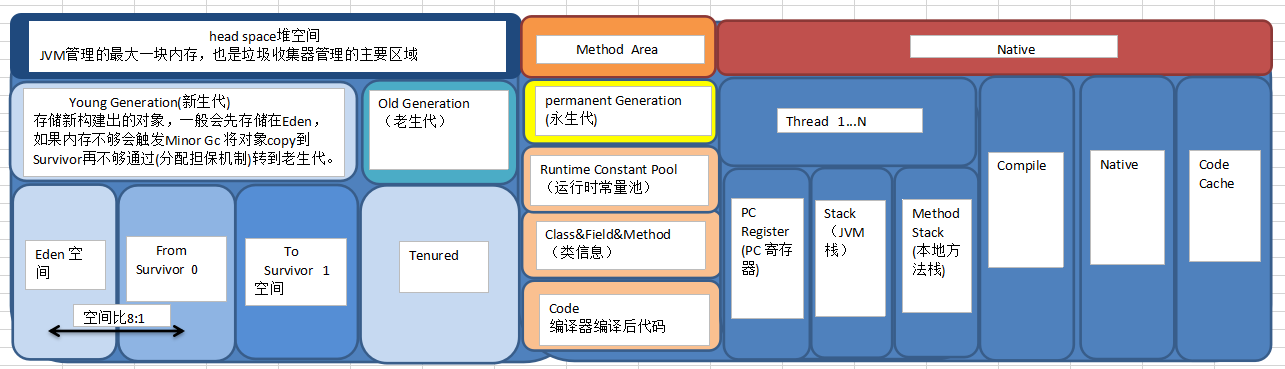
4、SQL语句优化（效果不高）

1. 管理工具问题（SVN，Git）
2. Linux常用指令
3. 项目相关问题\*
4. JVM底层问题\*

<https://mp.weixin.qq.com/s/8hi3XyVhvspjO_Vp-qs-mg>

<https://mp.weixin.qq.com/s/Q3KdziNqWVlcmg7yq5sKaw>

<https://mp.weixin.qq.com/s/hsgTHVQLApPFgI1sHzpgjQ>



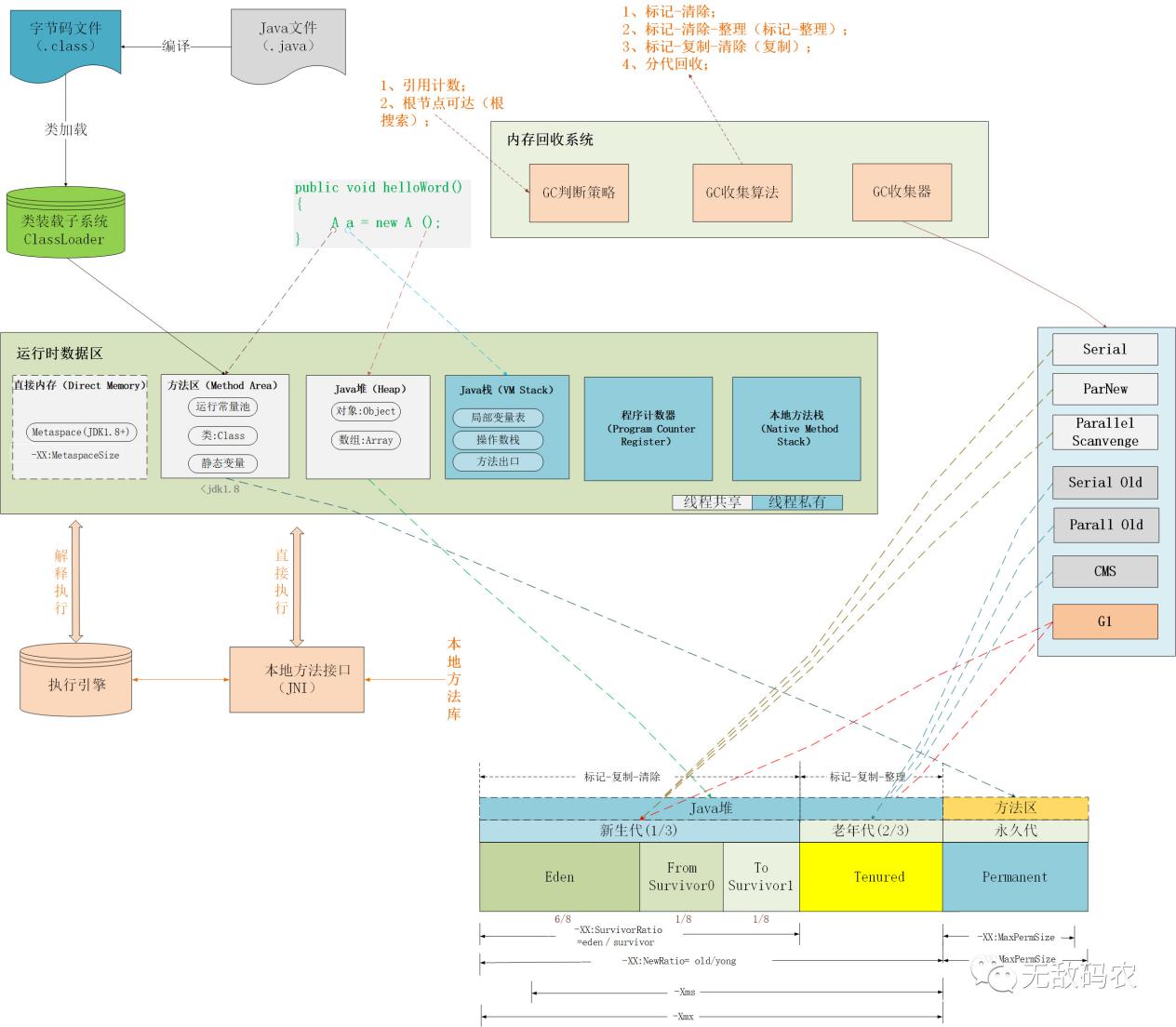
常量池:顾名思义主要用于存放两大类常量字面量和符号引用量，字面量相当于Java语言层面常量的概念，比如：字符串常量、声明为final的常量等等

分配担保机制:在发送Minor GC前，JVM会检查老年代最大可用的连续空间是否大于新生代所有对象的总空间，如果是，那么可以确保Minor GC是安全的，如果不是，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果小于，直接进行Full GC，如果大于，将尝试着进行一次Minor GC，Minor GC失败才会触发Full GC。注：不同版本的JDK，流程略有不同

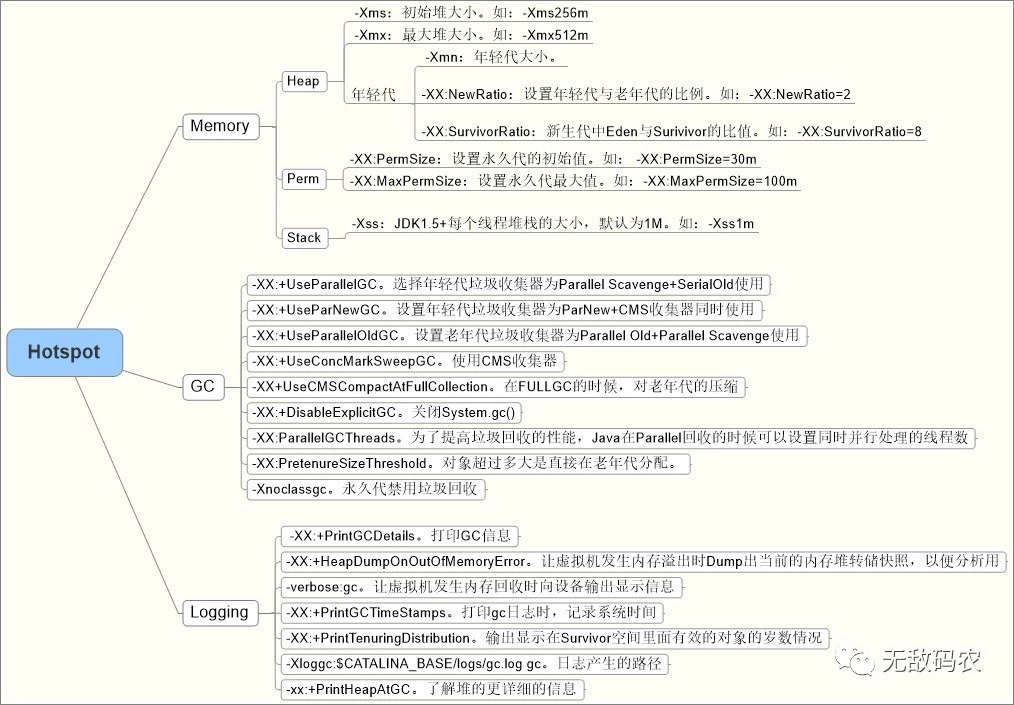
虚拟机栈:与本地方法栈都属于线程私有，它们的生命周期与线程相同。虚拟机栈用于描述Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧(Stack Frame)用于存储局部变量表、操作数栈、动态连接、方法出口等信息

堆外内存:堆外内存又被称为直接内存(Direct Memory)，它并不是虚拟机运行时数据区的一部分，Java虚拟机规范中也没有定义这部分内存区域，使用时由Java程序直接向系统申请，访问直接内存的速度要优于Java堆，因此，读写频繁的场景下使用直接内存，性能会有提升，比如Java NIO库，就是使用Native函数直接分配堆外内存，然后通过一个存储在Java堆中的DirectBytedBuffer对象作为这块内存的引用进行操作。

由于直接内存在Java堆外，其大小不会直接受限于Xmx指定的堆大小，但它肯定会受到本机总内存大小以及处理器寻址空间的限制，因此我们在配置JVM参数时，特别是有大量网络通讯场景下，要特别注意，防止各个内存区域的总内存大于物理内存限制 (包括物理的和OS的限制)。







JVM的垃圾回收机制：

YGC耗时分为三个组成部分：

1. 标记存活耗时
2. 复制耗时（从Eden和（form）survivor—>（to）survivor）
3. 记录日志耗时，并且YGC耗时与Eden的空间没有太大关系。

JVM的堆内存实现为什么采用分代思想：

1、清空垃圾对象，保留存活对象。

这种方式，不会涉及对象复制，应该效率还不错，但是存在致命的问题，可能发生GC几次之后，就出现大量的内存碎片，而这些碎片已经小到不足以分配任何对象

2、移动存活对象。

这种方式，需要把这些对象移动到内存的一端，可以完美解决内存碎片问题，但是，对于长期活跃的对象，每次GC都要进行移动，特别是对于大对象来说，这个效率实在可怕。

**JVM对新生代进行了更细的划分，分成Eden区和2个Survivor，新创建的小对象直接在Eden进行分配，大对象可以选择在老年代进行分配（这样可以有效的防止在YGC的时候进行移动）。当Eden分配满了之后，就会触发我们熟悉的YGC进行垃圾回收，先标记存活对象，再复制到其中一个Survivor区，如果Survivor区装不下存活对象，那说明JVM参数设置的不太合理，因为Eden区和2个Survivor的默认比例是8:1:1**

总起来讲就是为了解决空间浪费和耗时效率才采用分代思想的。

GC算法：

1. Mark-Sweep（标记-清除）算法

缺点：会产生内存碎片

1. Copying（复制）算法

缺点：浪费内存太大

1. Mark-Compact（标记-整理）算法
2. Generational Collection（分代收集）算法

分代使用不同的算法，新生代垃圾多使用复制算法，老年代使用整理算法