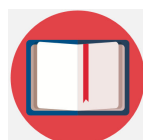


今日头条 Java 面试题

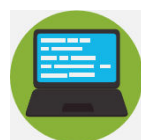
头条 Java 一面	2
头条 Java 二面	2
头条 Java 三面	3
头条 Java 一面参考答案:	3
MySQL 锁概述	3
行锁 和 表锁	4
悲观锁 和 乐观锁	4
共享锁	5
表锁和行锁应用场景:	6
乐观锁 VS 悲观锁	6
公平锁 VS 非公平锁	8
独享锁 VS 共享锁	9
分段锁	10
什么场景需要 JVM 调优	11
JVM 性能监控分析工具	11
VisualVM	11
Jconsole	13
MAT	18
JVM 内存泄漏分析	21
BAT 必考 JVM 系列专题	23
JVM 内存结构	24
堆内存 (Heap)	24
垃圾回收算法	28
垃圾回收机制	32
垃圾回收有两种类型: Minor GC 和 Full GC。	33
垃圾回收算法总结	33
头条 Java 二面参考答案:	34
哨兵 (sentinal)	34
Redis 复制 (Replication)	35
Redis 主从复制、哨兵和集群这三个有什么区别	37
Redis 的高并发和快速原因	38
为什么 Redis 是单线程的	38
Redis 单线程的优劣势	39
IO 多路复用技术	39
Redis 高并发快总结	40
数据同步一致性解决方案	41
缓存记录写 key 法	44
头条 Java 三面参考答案	45



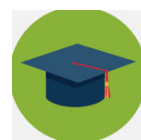
全网资源 点击即达



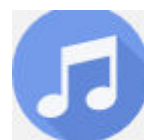
电子书



学习



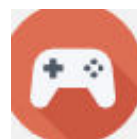
考研考公



音乐



影视剧集



游戏



软件



各种教程

更多免费资源

<https://link3.cc/8088>

全网资源 持续更新



如何解决 Redis 的并发竞争 key 问题.....	45
数据同步一致性解决方案.....	46
缓存记录写 key 法.....	49
缓存和数据库一致性解决方案.....	50
秒杀活动场景.....	52
秒杀背后的技术挑战.....	52
如何解决秒杀技术瓶颈.....	53
秒杀架构设计总结:	54

头条 Java 一面

1. 讲讲 jvm 运行时数据库区
2. 讲讲你知道的垃圾回收算法
3. jvm 内存模型 jmm
4. 内存泄漏与内存溢出的区别
5. select、epoll 的区别? 底层的数据结构是什么?
6. mysql 数据库默认存储引擎, 有什么优点
7. 优化数据库的方法, 从 sql 到缓存到 cpu 到操作系统, 知道多少说多少
8. 什么情景下做分表, 什么情景下做分库
9. linkedList 与 arrayList 区别 适用场景
10. array list 是如何扩容的
11. volatile 关键字的作用? Java 内存模型?
12. java lock 的实现, 公平锁、非公平锁
13. 悲观锁和乐观锁, 应用中的案例, mysql 当中怎么实现, java 中的实现

头条 Java 二面

- Java 内存分配策略? 多个线程同时请求内存, 如何分配?
- Redis 底层用到了哪些数据结构? 使用 Redis 的 set 来做过什么?
- Redis 使用过程中遇到什么问题? 搭建过 Redis 集群吗?
- 如何分析“慢查询”日志进行 SQL/索引 优化?

- MySQL 索引结构解释一下？（B+ 树）
- MySQL Hash 索引适用情况？举下例子？

头条 Java 三面

- 如何保证数据库与 redis 缓存一致的
- Redis 的并发竞争问题是什么？如何解决这个问题？了解 Redis 事务的 CAS 方案吗？
- 如何保证 Redis 高并发、高可用？
- Redis 的主从复制原理，以及 Redis 的哨兵原理？
- 如果让你写一个消息队列，该如何进行架构设计啊？说一下你的思路。
- MySQL 数据库主从同步怎么实现？
- 秒杀模块怎么设计的，如何压测，抗压手段

头条 Java 一面参考答案：

MySQL 锁概述

相对其他数据库而言，MySQL 的锁机制比较简单，其最显著的特点是不同的存储引擎支持不同的锁机制。

比如：

- MyISAM 和 MEMORY 存储引擎采用的是表级锁（table-level locking）；
- InnoDB 存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。

MySQL 主要的两种锁的特性可大致归纳如下：

	行锁	表锁	页锁
MyISAM		√	
BDB		√	√
InnoDB	√	√	

表级锁：开销小，加锁快；不会出现死锁（因为 MyISAM 会一次性获得 SQL 所需的全部锁）；锁定粒度大，发生锁冲突的概率最高，并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

页锁：开销和加锁速度介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发度一般

行锁 和 表锁

1. 主要是针对锁粒度划分的，一般分为：行锁、表锁、库锁

(1) 行锁：访问数据库的时候，锁定整个行数据，防止并发错误。

(2) 表锁：访问数据库的时候，锁定整个表数据，防止并发错误。

2. 行锁 和 表锁 的区别：

- 表锁：开销小，加锁快，不会出现死锁；锁定力度大，发生锁冲突概率高，并发度最低
- 行锁：开销大，加锁慢，会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高

悲观锁 和 乐观锁

(1) 悲观锁：顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会 block 直到它拿到锁。

传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

(2) **乐观锁**：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在**更新的时候会判断一下**在此期间别人有没有去更新这个数据，可以使用版本号等机制。

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 `write_condition` 机制的其实都是提供的乐观锁。

(3) 悲观锁 和 乐观锁的区别：

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行 `retry`，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

共享锁

共享锁指的就是对于多个不同的事务，对同一个资源共享同一个锁。相当于对于同一把门，它拥有多个钥匙一样。就像这样，你家有一个大门，大门的钥匙有好几把，你有一把，你女朋友有一把，你们都可能通过这把钥匙进入你们家，这个就是所谓的共享锁。

刚刚说了，对于悲观锁，一般数据库已经实现了，**共享锁也属于悲观锁的一种**，那么共享锁在 `mysql` 中是通过什么命令来调用呢。通过查询资料，了解到通过在执行语句后面加上 `lock in share mode` 就代表对某些资源加上共享锁了。

什么时候使用表锁

对于 InnoDB 表，在绝大部分情况下都应该使用行级锁，因为事务和行锁往往是我们之所以选择 InnoDB 表的理由。但在个别特殊事务中，也可以考虑使用表级锁。

- 第一种情况是：事务需要更新大部分或全部数据，表又比较大，如果使用默认的行锁，不仅这个事务执行效率低，而且可能造成其他事务长时间锁等待和锁冲突，这种情况下可以考虑使用表锁来提高该事务的执行速度。
- 第二种情况是：事务涉及多个表，比较复杂，很可能引起死锁，造成大量事务回滚。这种情况也可以考虑一次性锁定事务涉及的表，从而避免死锁、减少数据库因事务回滚带来的开销。

当然，应用中这两种事务不能太多，否则，就应该考虑使用 MyISAM 表了。

表锁和行锁应用场景：

- 表级锁使用与并发性不高，以查询为主，少量更新的应用，比如小型的 web 应用；
- 而行级锁适用于高并发环境下，对事务完整性要求较高的系统，如在线事务处理系统。

乐观锁 VS 悲观锁

乐观锁与悲观锁是一种广义上的概念，体现了看待线程同步的不同角度，在 Java 和数据库中都有此概念对应的实际应用。

1. 乐观锁

顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。

乐观锁适用于多读的应用类型，乐观锁在 Java 中是通过使用无锁编程来实现，最常采用的是 CAS 算法，Java 原子类中的递增操作就通过 CAS 自旋实现的。

CAS 全称 Compare And Swap（比较与交换），是一种无锁算法。在不使用锁（没有线程被阻塞）的情况下实现多线程之间的变量同步。java.util.concurrent 包中的原子类就是通过 CAS 来实现了乐观锁。

简单来说，CAS 算法有 3 个操作数：

需要读写的内存值 V。

进行比较的值 A。

要写入的新值 B。

当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则返回 V。这是一种乐观锁的思路，它相信在它修改之前，没有其它线程去修改它；而

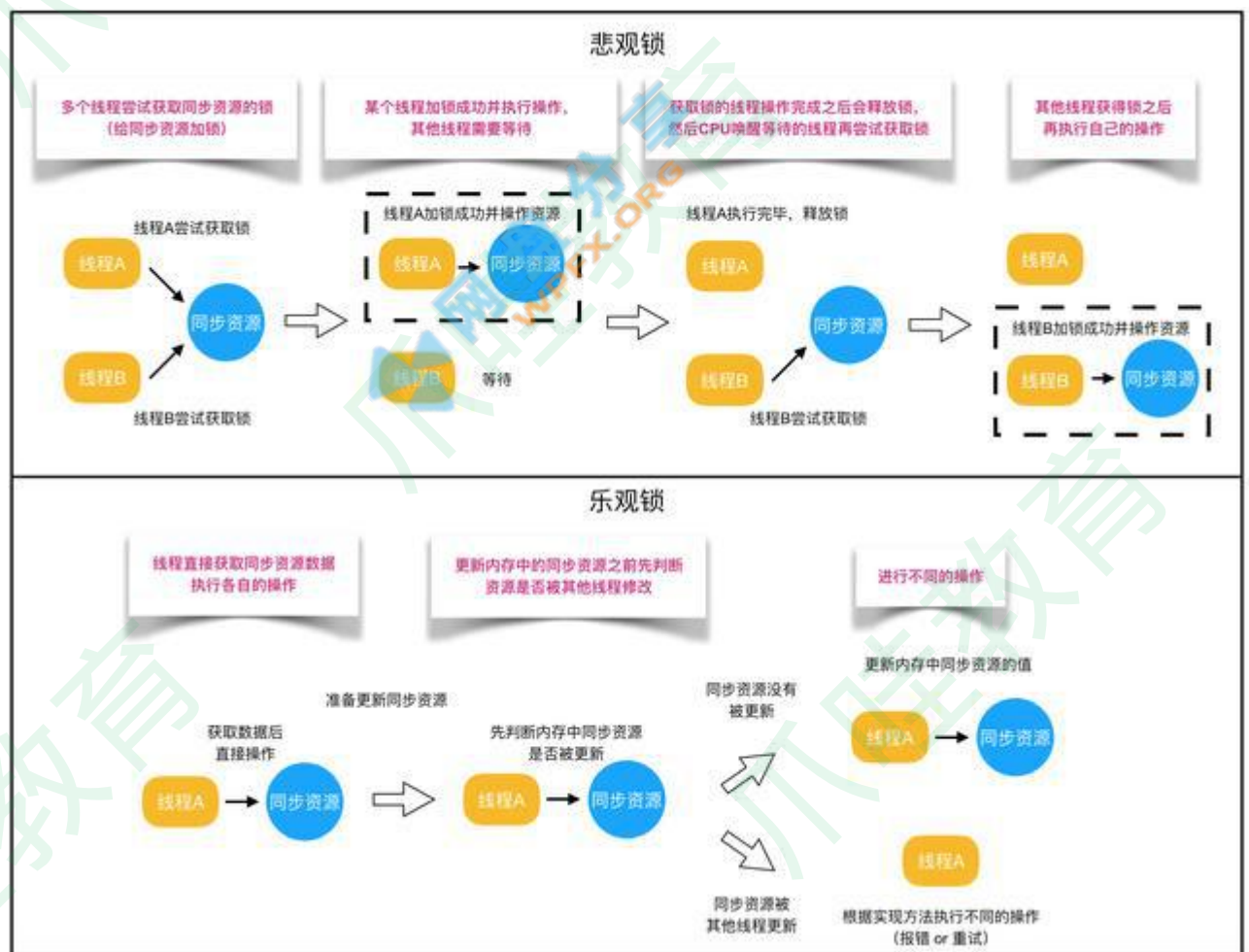
`Synchronized` 是一种悲观锁，它认为在它修改之前，一定会有其它线程去修改它，悲观锁效率很低。

2. 悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。

传统的 MySQL 关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。详情可以参考：[阿里 P8 架构师谈：MySQL 行锁、表锁、悲观锁、乐观锁的特点与应用](#)

再比如上面提到的 Java 的同步 `synchronized` 关键字的实现就是典型的悲观锁。



3. 总之：

- 悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。
- 乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

公平锁 VS 非公平锁

1. 公平锁

就是很公平，在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照 FIFO 的规则从队列中取到自己。

公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU 唤醒阻塞线程的开销比非公平锁大。

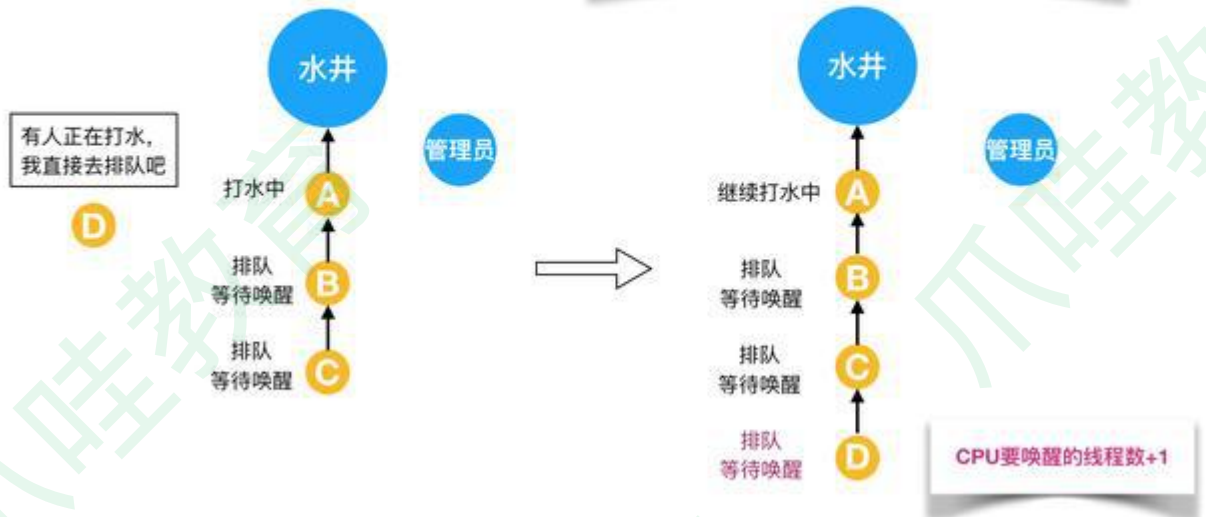
2. 非公平锁

上来就直接尝试占有锁，如果尝试失败，就再采用类似公平锁那种方式。

非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU 不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁。

公平锁中，线程会直接进入等待队列。

等待队列中除第一个线程以外的所有线程都会阻塞，都需要CPU去唤醒。线程D进入队列并阻塞，导致CPU开销增加（需要唤醒D）。



3. 典型应用：

java jdk 并发包中的 ReentrantLock 可以指定构造函数的 boolean 类型来创建公平锁和非公平锁（默认），比如：公平锁可以使用 `new ReentrantLock(true)` 实现。

独享锁 VS 共享锁

1. 独享锁

是指该锁一次只能被一个线程所持有。

2. 共享锁

是指该锁可被多个线程所持有。

3. 比较

对于 Java ReentrantLock 而言，其是独享锁。但是对于 Lock 的另一个实现类 ReadWriteLock，其读锁是共享锁，其写锁是独享锁。

读锁的共享锁可保证并发读是非常高效的，读写，写读，写写的过程是互斥的。

独享锁与共享锁也是通过 AQS 来实现的，通过实现不同的方法，来实现独享或者共享。

4. AQS

抽象队列同步器（AbstractQueuedSynchronizer，简称 AQS）是用来构建锁或者其他同步组件的基础框架，它使用一个整型的 volatile 变量（命名为 state）来维护同步状态，通过内置的 FIFO 队列来完成资源获取线程的排队工作。

高层类	Lock	同步器	阻塞队列	Executor	并发容器
基础类		AQS	非阻塞数据结构	原子变量类	
		volatile变量的读/写		CAS	

concurrent 包的实现结构如上图所示，AQS、非阻塞数据结构和原子变量类等基础类都是基于 volatile 变量的读/写和 CAS 实现，而像 Lock、同步器、阻塞队列、Executor 和并发容器等高层类又是基于基础类实现。

分段锁

分段锁其实是一种锁的设计，并不是具体的一种锁，对于 ConcurrentHashMap 而言，其并发的实现就是通过分段锁的形式来实现高效的并发操作。

我们以 ConcurrentHashMap 来说一下分段锁的含义以及设计思想，ConcurrentHashMap 中的分段锁称为 Segment，它即类似于 HashMap（JDK7 与 JDK8 中 HashMap 的实现）的结构，即内部拥有一个 Entry 数组，数组中的每个元素又是一个链表；同时又是一个 ReentrantLock（Segment 继承了 ReentrantLock）。

当需要 put 元素的时候，并不是对整个 hashmap 进行加锁，而是先通过 hashcode 来知道他要放在那一个分段中，然后对这个分段进行加锁，所以当多线程 put 的时候，只要不是放在一个分段中，就实现了真正的并行的插入。

但是，在统计 size 的时候，可就是获取 hashmap 全局信息的时候，就需要获取所有的分段锁才能统计。

分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

什么场景需要 JVM 调优

- OutOfMemoryError，内存不足
- 内存泄露
- 线程死锁
- 锁争用 (Lock Contention)
- Java 进程消耗 CPU 过高

这些问题出现的时候常常通过重启服务器或者调大内存来临时解决，实际情况，还需要尽量还原当时的业务场景，并分析内存、线程等数据，通过分析找到最终的解决方案，这就会涉及到性能分析工具。

JVM 性能监控分析工具

JDK 本身提供了很丰富的性能监控工具，除了集成式的 visualVM 和 jConsole 外，还有 jstat, jstack, jps, jmap, jhat 小工具，这些都是性能调优的常用工具。

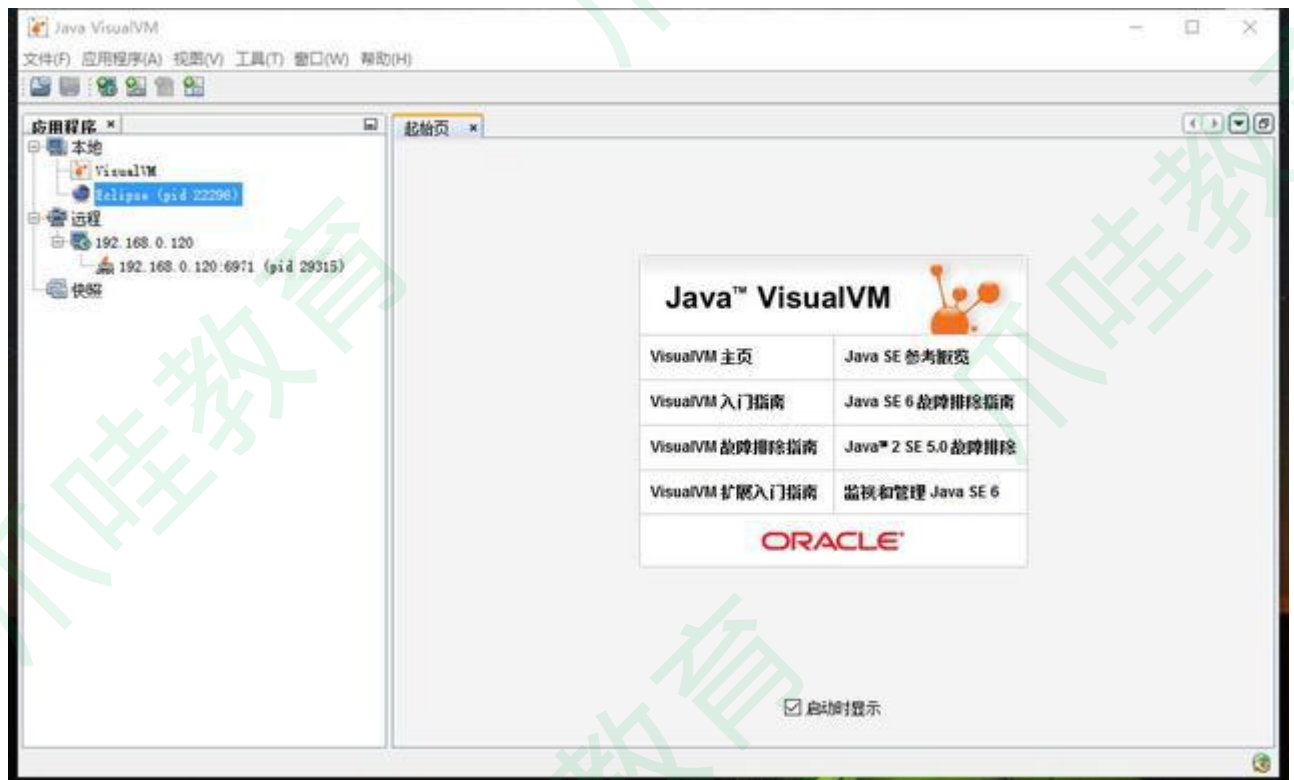
- **Jconsole**：jdk 自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。
- **JProfiler**：商业软件，功能强大。
- **VisualVM**：JDK 自带，功能强大，与 JProfiler 类似。
- **MAT**：MAT (Memory Analyzer Tool)，一个基于 Eclipse 的内存分析工具。

VisualVM

VisualVM 是 javajdk 自带的牛逼的调优工具，也是平时使用最多调优工具，几乎涉及了 jvm 调优的方方面面。启动起来后和 jconsole 一样同样可以选择本地和远程，如果需要监控远程同样需要配置相关参数。

1 打开 VisualVM

这个工具放在 JDK 安装目录的 bin 目录下，双击 jvisualvm.exe 即可打开，如下图所示



2. 监视页面主要展示 系统资源占用情况

CPU :展示 java 程序运行的时候占用的 cpu 资源

堆 :这里要说明下堆内存的组成部分,堆是由老年代和新生代组成,其中新生代有由”伊甸园”和”两个幸存者区组成”三部分组成,堆视图看到的资源占用实际是”老年代”、”伊甸园(Eden)”、”两个幸存者(Survivor)”的一个综合情况。

PermGen :Perm 区用来存放 java 类以及其他虚拟机自己的静态数据,(常被称为持久代或者方法区)

类 :此视图 主要展示 当前程序加载了多少个类

线程 :当前程序的线程启动情况

堆 Dump : 生产当前程序的内存快照 hprof 文件,对于分析内存溢出问题比较有帮助。

3. 线程页面(主要展示程序中所有的线程运行状态)

- **线程 dump** : 所有线程的快照(对分析线程死锁,比较有帮助)

- **时间线：** 展示每个线程的实时运行状态（不同颜色代表不同的状态）

VisualVM 可以根据需要安装不同的插件，每个插件的关注点都不同，有的主要监控 GC，有的主要监控内存，有的监控线程等。

Jconsole

JConsole 是一个 JMX（Java Management Extensions，即 Java 管理扩展）的 JVM 监控与管理工具，监控主要体现在：堆栈内存、线程、CPU、类、VM 信息这几个方面，而管理主要是对 JMX MBean（managed beans，被管理的 beans，是一系列资源，包含对象、接口、设备等）的管理，不仅能查看 bean 的属性和方法信息，还能够在运行时修改属性或调用方法。

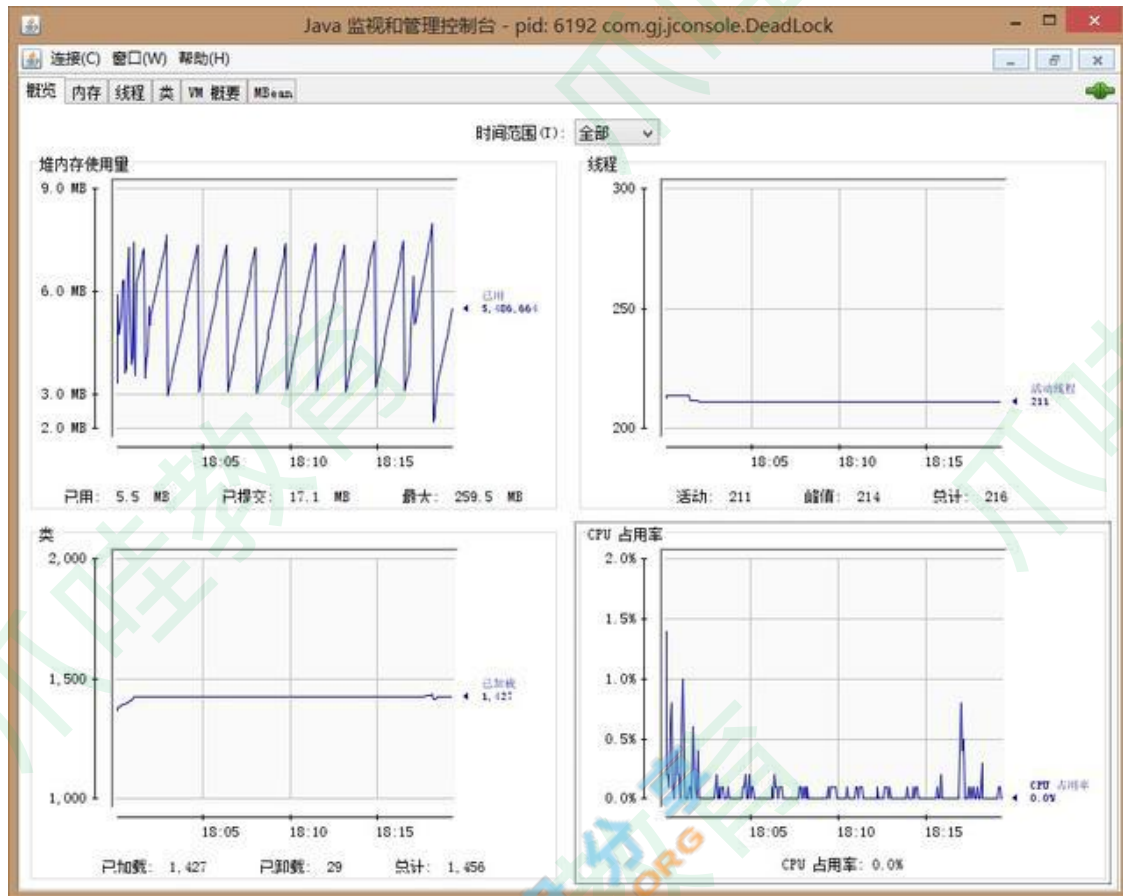
1. 打开 Jconsole

直接在 jdk/bin 目录下点击 jconsole.exe 即可启动，界面如下：



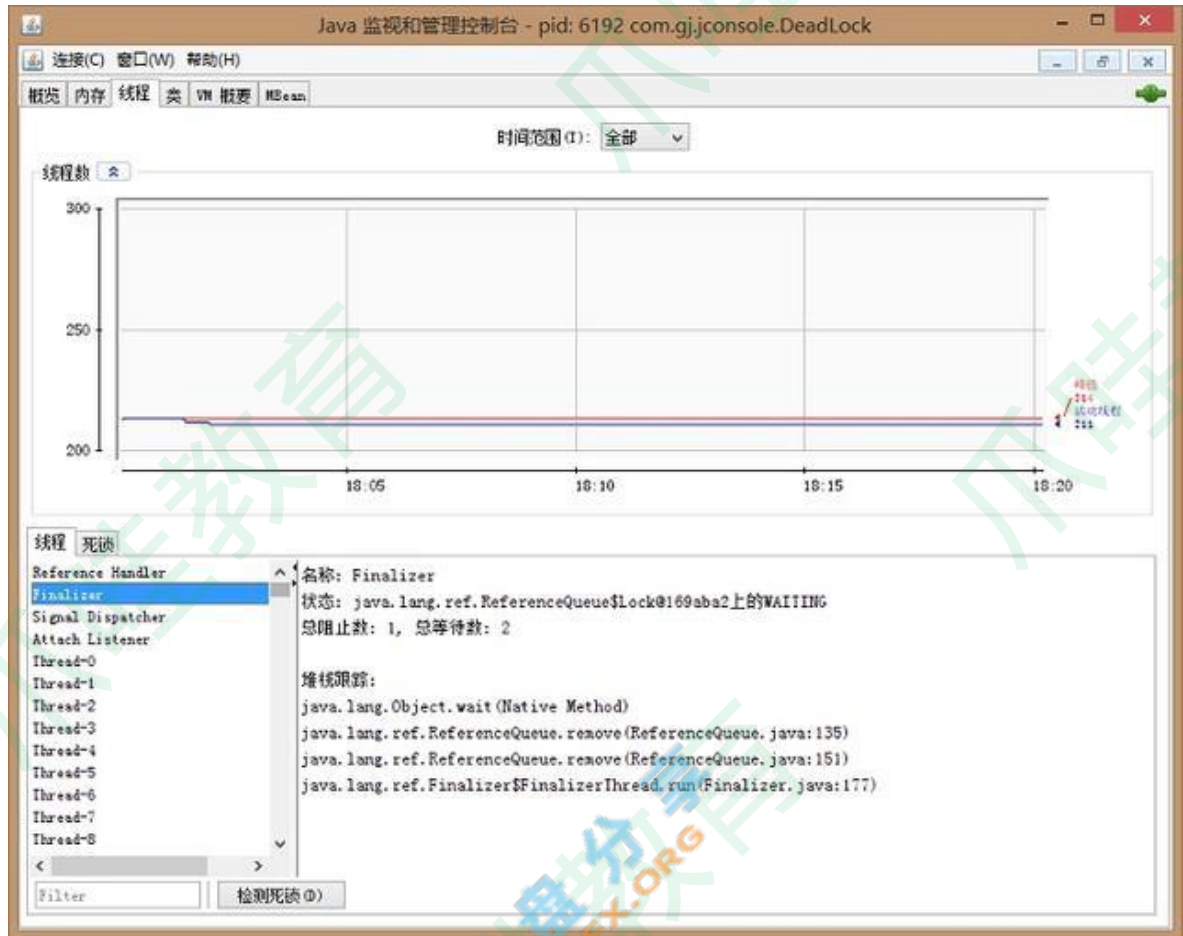
1、内存

在内存页我们可以看到程序运行期间 JVM 各个部分的内存状况，右下角是对应各个分区的内存使用柱状图，点击对应柱可查看详情，看图：



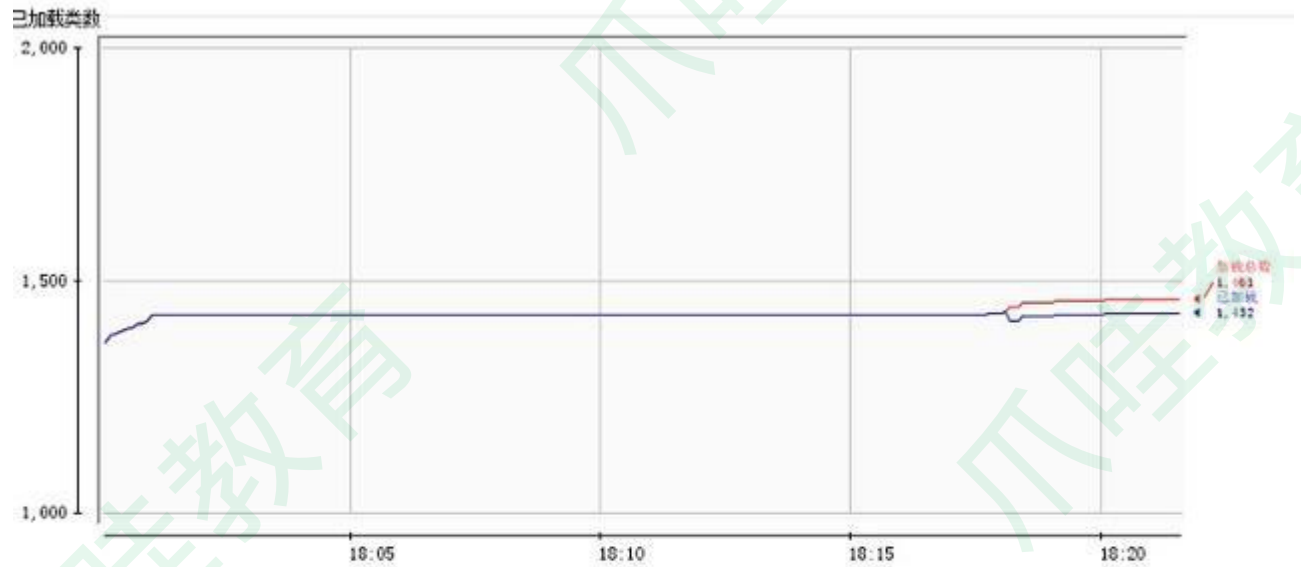
2、线程

该页面可以查看当前 JVM 进程启动了多少个线程，并能查看每个线程的状态及堆栈信息，此外还有一个功能就是能够自动检测死锁，见图：



3、类

该页面其实和线程页有些相似，不过显示的是 JVM 加载类的信息，见图：



详细资料

时间: 2014-03-30 18:21:34
已加载当前类: 1,432
已加载类总数: 1,461
已卸载类总数: 29

4、VM 概述

这个其实没必要细说，看图就明白，显示了当前 JVM 的各方面信息：

Java 监视和管理控制台 - pid: 6192 com.gjjconsole.DeadLock

连接(C) 窗口(W) 帮助(H)

概览 内存 线程 类 VM 概要 MBean

VM 概要

2014年3月30日 星期日 下午06时21分54秒 CST

连接名称: pid: 6192 com.gjjconsole.DeadLock	运行时间: 22 分钟
虚拟机: Java HotSpot(TM) Client VM版本 23.7-b01	进程 CPU 时间: 3.796 秒
厂商: Oracle Corporation	JIT 编译器: HotSpot Client Compiler
名称: 6192@gj	总编译时间: 0.200 秒

活动线程: 211	已加载当前类: 1,433
峰值: 214	已加载类总数: 1,462
守护程序线程: 10	已卸载类总数: 29
启动的线程总数: 216	

当前堆大小: 6,368 KB	提交的内存: 16,704 KB
最大堆大小: 253,440 KB	暂挂最终处理: 0对象

垃圾收集器: 名称 = 'Copy', 收集 = 43, 总花费时间 = 0.060 秒

垃圾收集器: 名称 = 'MarkSweepCompact', 收集 = 1, 总花费时间 = 0.014 秒

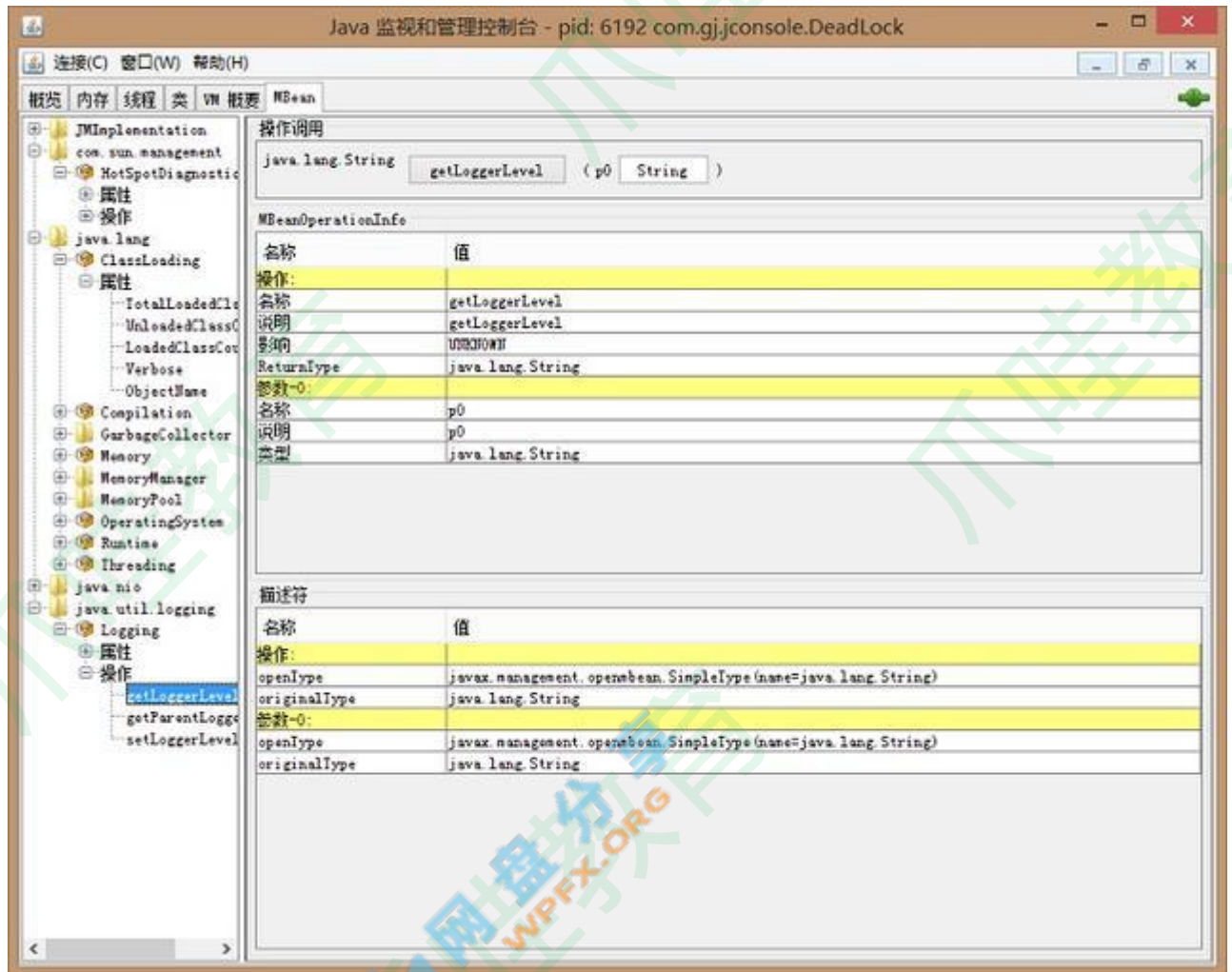
操作系统: Windows 8 6.2	总物理内存: 4,194,303 KB
体系结构: x86	空闲物理内存: 4,194,303 KB
处理程序数: 4	总交换空间: 4,194,303 KB
提交的虚拟内存: 69,544 KB	空闲交换空间: 4,194,303 KB

VM 参数: -Dfile.encoding=UTF-8

类路径: I:\java\win8\jtools-test\bin

库路径: I:\java\jdk1.7.0_17\bin\;C:\Windows\SunJava\bin;C:\Windows\system32;C:\Windows;I:\java\jdk1.7.0_17\bin\;jre\bin\client;I:\java\jdk1.7.0_17\bin\;jre\bin;
I:\java\jdk1.7.0_17\bin\;jre\lib\j386;C:\Program Files (x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files (x86)\Common Files\NetSarang\C

5、MBean 管理



MAT

MAT (Memory Analyzer Tool)，一个基于 Eclipse 的内存分析工具，是一个快速、功能丰富的 Java heap 分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。

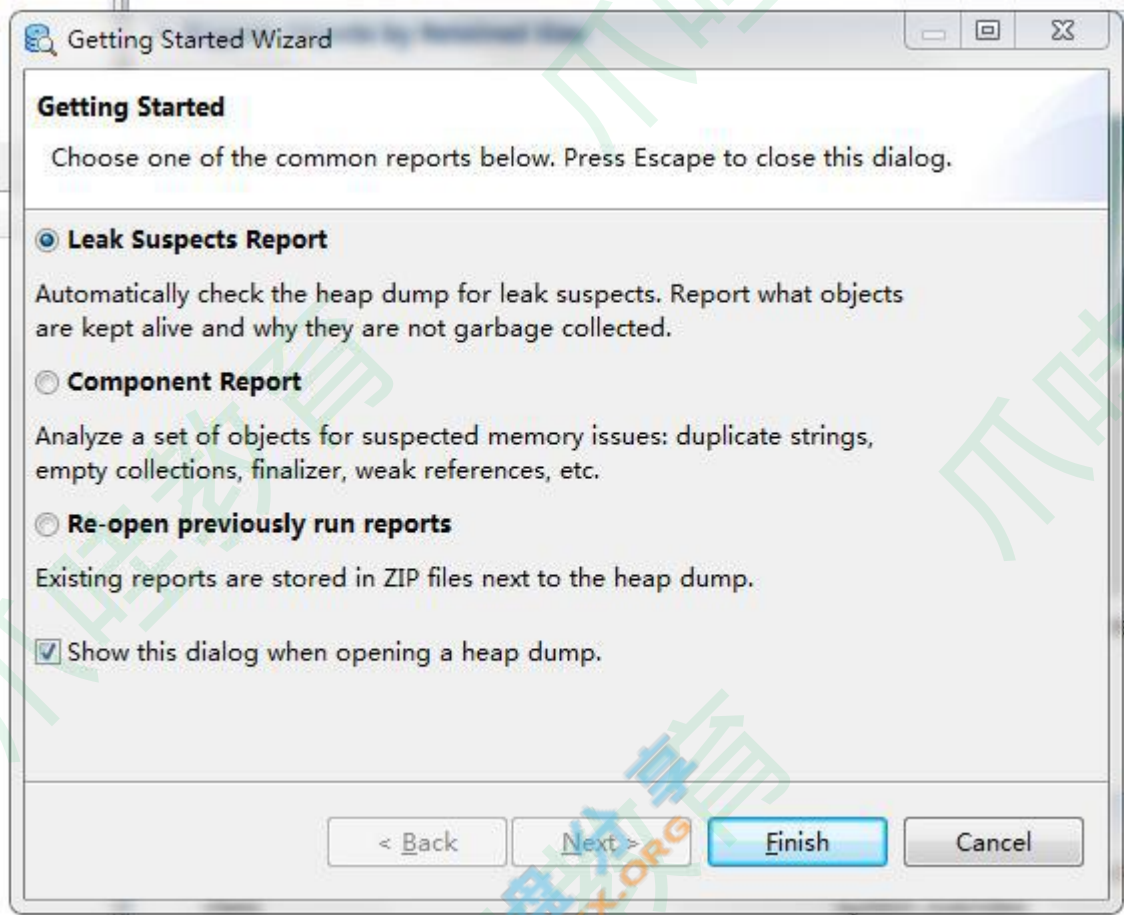
可以利用 visualvm 或者是 jmap 命令生产堆文件在进行内存分析。

1. 用 jmap 生成堆信息

```
D:\>jmap -dump:live,format=b,file=E:\jmap\map.bin 1464
Dumping heap to E:\jmap\map.bin ...
Heap dump file created
D:\>_
```

这样在 E 盘的 jmap 文件夹里会有一个 map.bin 的堆信息文件

2. 将堆信息导入到 mat 中分析



3. 生成分析报告

可以利用 `visualvm` 或者是 `jmap` 命令生产堆文件，导入 `eclipse mat` 中生成分析报告：



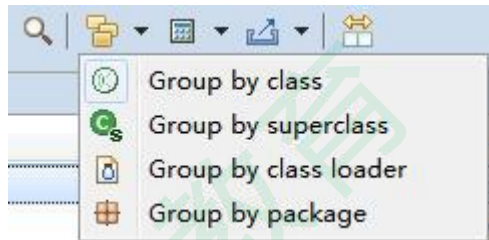
1) Histogram (直方图) 视图

Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	219,698	21,413,424	>= 21,413,424
byte[]	16,400	15,580,120	>= 15,580,120
java.lang.String	222,685	5,344,440	>= 23,295,888
org.objectweb.asm.Item	91,673	4,400,304	>= 8,159,792
org.objectweb.asm.Label	74,046	4,146,576	>= 8,682,248
org.objectweb.asm.Edge	146,529	3,516,696	>= 3,516,696
java.lang.reflect.Method	28,010	2,240,800	>= 2,938,472
org.objectweb.asm.Method...	11,664	2,239,488	>= 15,266,184
java.util.HashMap\$Entry[]	22,248	2,207,912	>= 29,975,144
java.lang.Object[]	38,582	1,790,064	>= 9,605,920
int[]	27,222	1,666,752	>= 1,666,752
java.util.LinkedHashMap\$Ent...	51,862	1,659,584	>= 5,654,480
java.util.HashMap\$Entry	53,719	1,289,256	>= 24,520,160

- Class Name : 类名称, java 类名
- Objects : 类的对象的数量, 这个对象被创建了多少个
- Shallow Heap : 一个对象内存的消耗大小, 不包含对其他对象的引用

- Retained Heap : 是 shallow Heap 的总和，也就是该对象被 GC 之后所能回收到内存的总和

通过直方图视图可以很容易找到占用内存最多的几个类（通过 Retained Heap 排序），还可以通过其他方式进行分组（见下图）。



如果存在内存溢出，时间久了溢出类的实例数量或者内存占比会越来越多，排名也越来越靠前。

图标进行对比，通过多次对比不同时间点下的直方图对比就很容易把溢出的类找出来。

JVM 内存泄漏分析

造成 OutOfMemoryError 内存泄露典型原因：对象已经死了，无法通过垃圾收集器进行自动回收，需要通过找出泄露的代码位置和原因，才好确定解决方案。

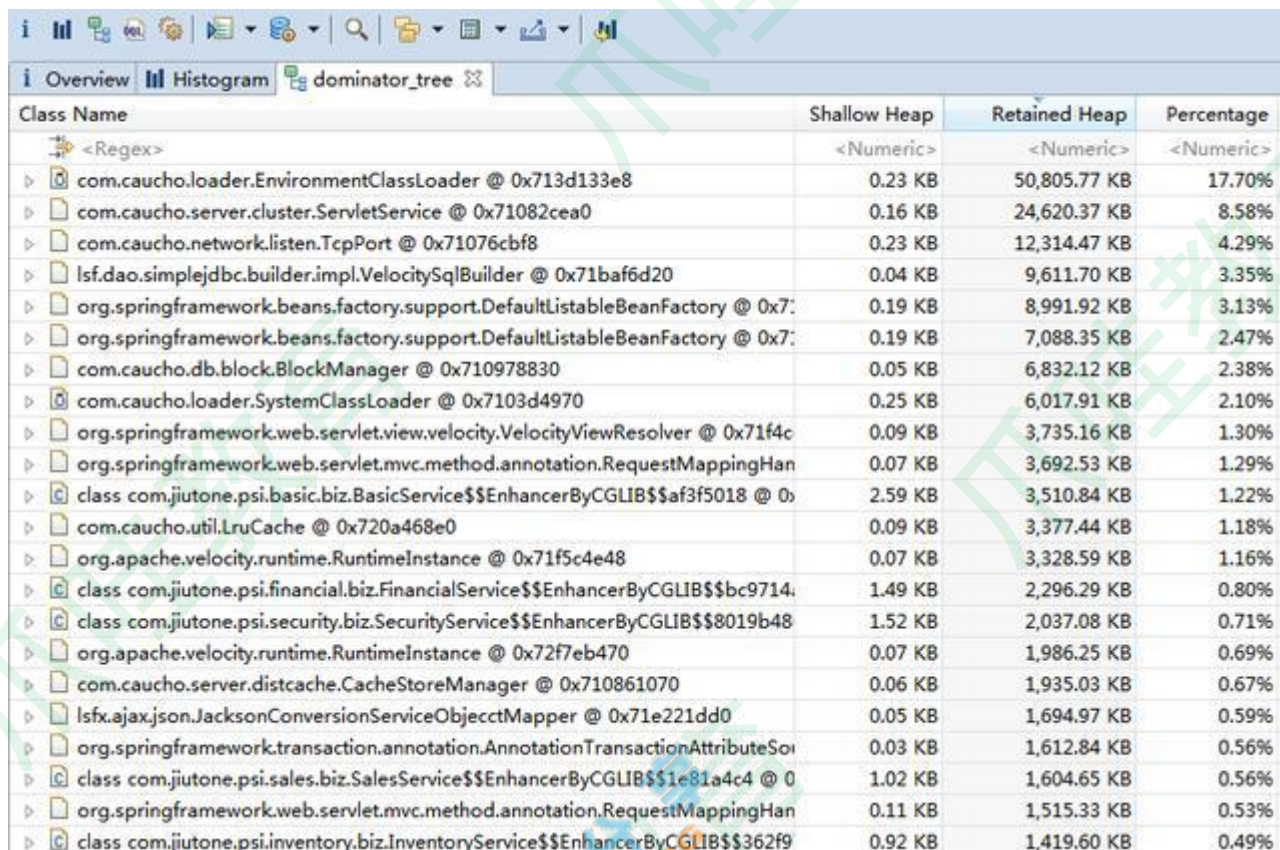
分析步骤：

1. 用工具生成 java 应用程序的 heap dump（如 jmap）
2. 使用 Java heap 分析工具（如 MAT），找出内存占用超出预期的嫌疑对象
3. 根据情况，分析嫌疑对象和其他对象的引用关系。
4. 分析程序的源代码，找出嫌疑对象数量过多的原因。

i Overview Histogram		
Class Name	Objects	Shallow Heap
<Regex>	<Numeric>	<Numeric>
char[]	+394,314	+24,512.46 KB
java.util.HashMap\$Entry	+432,590	+13,518.44 KB
com.mysql.jdbc.ConnectionPropertiesImpl\$Bo...	+178,080	+11,130.00 KB
java.lang.String	+366,765	+8,596.05 KB
java.util.Hashtable\$Entry	+263,348	+8,229.62 KB
com.mysql.jdbc.ConnectionPropertiesImpl\$Str...	+58,830	+3,676.88 KB
com.mysql.jdbc.ConnectionPropertiesImpl\$Int...	+44,520	+3,130.31 KB
java.lang.Integer	+199,512	+3,117.38 KB
java.util.HashMap\$Entry[]	+11,430	+2,971.09 KB
java.util.Hashtable\$Entry[]	+5,113	+2,637.67 KB
int[]	+20,054	+2,468.73 KB
com.caucho.server.dispatch.Invocation	+25,193	+2,361.84 KB
com.mysql.jdbc.JDBC4Connection	+1,590	+1,751.48 KB
net.sf.cglib.asm.Edge	+46,380	+1,087.03 KB
net.sf.cglib.asm.Item	+19,714	+1,078.11 KB
java.lang.reflect.Method	+13,092	+1,022.81 KB
com.caucho.server.webapp.WebAppFilterChain	+25,193	+984.10 KB
java.util.HashMap	+19,801	+928.17 KB
com.caucho.server.http.InvocationKey	+25,193	+787.28 KB
org.apache.velocity.runtime.parser.Token	+16,355	+766.64 KB

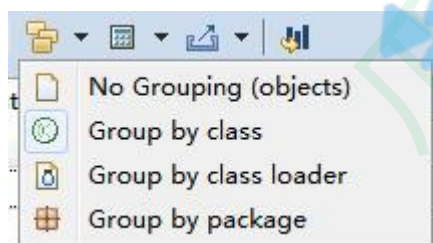
2) 支配树 (Dominator Tree)

MAT 提供了一个称为支配树 (Dominator Tree) 的对象图。支配树体现了对象实例间的支配关系，在此视图中列出了每个对象 (Object Instance) 与其引用关系的树状结构，同时包含了占用内存的大小和百分比。



Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
com.caucho.loader.EnvironmentClassLoader @ 0x713d133e8	0.23 KB	50,805.77 KB	17.70%
com.caucho.server.cluster.ServletService @ 0x71082cea0	0.16 KB	24,620.37 KB	8.58%
com.caucho.network.listen.TcpPort @ 0x71076cbf8	0.23 KB	12,314.47 KB	4.29%
lsf.dao.simplejdbc.builder.impl.VelocitySqlBuilder @ 0x71baf6d20	0.04 KB	9,611.70 KB	3.35%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0x71076cbf8	0.19 KB	8,991.92 KB	3.13%
org.springframework.beans.factory.support.DefaultListableBeanFactory @ 0x71076cbf8	0.19 KB	7,088.35 KB	2.47%
com.caucho.db.block.BlockManager @ 0x710978830	0.05 KB	6,832.12 KB	2.38%
com.caucho.loader.SystemClassLoader @ 0x7103d4970	0.25 KB	6,017.91 KB	2.10%
org.springframework.web.servlet.view.velocity.VelocityViewResolver @ 0x71f4c	0.09 KB	3,735.16 KB	1.30%
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping @ 0x71f4c	0.07 KB	3,692.53 KB	1.29%
class com.jiutone.psi.basic.biz.BasicService\$\$EnhancerByCGLIB\$\$af3f5018 @ 0x71f4c	2.59 KB	3,510.84 KB	1.22%
com.caucho.util.LruCache @ 0x720a468e0	0.09 KB	3,377.44 KB	1.18%
org.apache.velocity.runtime.RuntimeInstance @ 0x71f5c4e48	0.07 KB	3,328.59 KB	1.16%
class com.jiutone.psi.financial.biz.FinancialService\$\$EnhancerByCGLIB\$\$bc9714	1.49 KB	2,296.29 KB	0.80%
class com.jiutone.psi.security.biz.SecurityService\$\$EnhancerByCGLIB\$\$8019b48	1.52 KB	2,037.08 KB	0.71%
org.apache.velocity.runtime.RuntimeInstance @ 0x72f7eb470	0.07 KB	1,986.25 KB	0.69%
com.caucho.server.distcache.CacheStoreManager @ 0x710861070	0.06 KB	1,935.03 KB	0.67%
lsfx.ajax.json.JacksonConversionServiceObjectMapper @ 0x71e221dd0	0.05 KB	1,694.97 KB	0.59%
org.springframework.transaction.annotation.AnnotationTransactionAttributeSource @ 0x71e221dd0	0.03 KB	1,612.84 KB	0.56%
class com.jiutone.psi.sales.biz.SalesService\$\$EnhancerByCGLIB\$\$1e81a4c4 @ 0x71e221dd0	1.02 KB	1,604.65 KB	0.56%
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping @ 0x71e221dd0	0.11 KB	1,515.33 KB	0.53%
class com.jiutone.psi.inventory.biz.InventoryService\$\$EnhancerByCGLIB\$\$362f9	0.92 KB	1,419.60 KB	0.49%

通过 Dominator Tree 视图可以很容易的找出占用内存最多的几个对象（根据 Retained Heap 或 Percentage 排序），和 Histogram 类似，可以通过不同的方式进行分组显示：



Histogram 视图和 Dominator Tree 视图的角度不同，前者是基于类的角度，后者是基于对象实例的角度，并且可以更方便的看出其引用关系。

以上只是一个初步的介绍，mat 还有更强大的使用，比如对比堆内存，在生产环境中往往为了定位问题，每隔几分钟 dump 出一下内存快照，随后在对比不同时间的堆内存的变化来发现问题。

BAT 必考 JVM 系列专题

1. JVM 内存模型
2. JVM 垃圾回收算法
3. JVM 垃圾回收器

- 4. JVM 参数详解
- 5. JVM 性能调优

JVM 内存结构



由上图可以清楚的看到 JVM 的内存空间分为 3 大部分：

- 堆内存
- 方法区
- 栈内存

其中栈内存可以再细分为 java 虚拟机栈和本地方法栈，堆内存可以划分为新生代和老年代，新生代中还可以再次划分为 Eden 区、From Survivor 区和 To Survivor 区。

其中一部分是线程共享的，包括 Java 堆和方法区；另一部分是线程私有的，包括虚拟机栈和本地方法栈，以及程序计数器这一小部分内存。

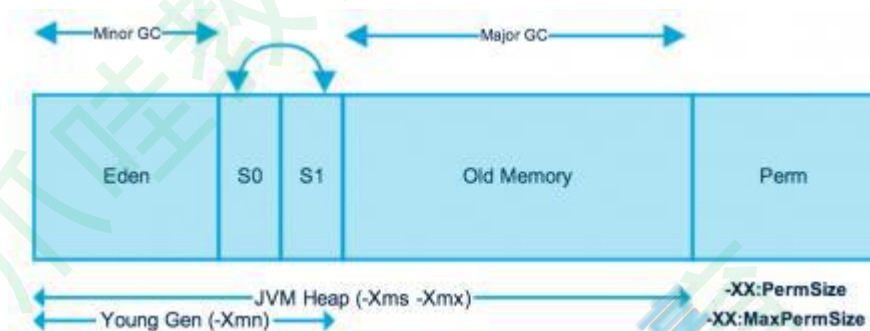
堆内存（Heap）

java 堆（Java Heap）是 Java 虚拟机所管理的内存中最大的一块。堆是被所有线程共享的区域，实在虚拟机启动时创建的。堆里面存放的都是对象的实例（new 出来的对象都存在堆中）。

此内存区域的唯一目的就是存放对象实例（new 的对象），几乎所有的对象实例都在这里分配内存。

堆内存分为两个部分：年轻代和老年代。我们平常所说的垃圾回收，主要回收的就是堆区。更细一点划分新生代又可划分为 Eden 区和 2 个 Survivor 区（From Survivor 和 To Survivor）。

下图中的 Perm 代表的是永久代，但是注意永久代并不属于堆内存中的一部分，同时 jdk1.8 之后永久代已经被移除。



新生代（Young）与老年代（Old）的比例的值为 1:2（该值可以通过参数 -XX:NewRatio 来指定）

默认的，Eden : from : to = 8 : 1 : 1（可以通过参数 -XX:SurvivorRatio 来设定），即：Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。

方法区（Method Area）

方法区也称“永久代”，它用于存储虚拟机加载的类信息、常量、静态变量、是各个线程共享的内存区域。

在 JDK8 之前的 HotSpot JVM，存放这些“永久的”的区域叫做“永久代(permanent generation)”。永久代是一片连续的堆空间，在 JVM 启动之前通过在命令行设置参数 -XX:MaxPermSize 来设定永久代最大可分配的内存空间，默认大小是 64M（64 位 JVM 默认是 85M）。

随着 JDK8 的到来，JVM 不再有永久代(PermGen)。但类的元数据信息(metadata)还在，只不过不再是存储在连续的堆空间上，而是移动到叫做“Metaspace”的本地内存(Native memory)。

方法区或永生代相关设置

-XX:PermSize=64MB 最小尺寸，初始分配

-XX:MaxPermSize=256MB 最大允许分配尺寸，按需分配

XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled 设置垃圾不回收

默认大小

-server 选项下默认 MaxPermSize 为 64m

-client 选项下默认 MaxPermSize 为 32m

虚拟机栈(JVM Stack)

java 虚拟机栈是**线程私有**，生命周期与线程相同。创建线程的时候就会创建一个 java 虚拟机栈。

虚拟机执行 java 程序的时候，每个方法都会创建一个栈帧，栈帧存放在 java 虚拟机栈中，通过压栈出栈的方式进行方法调用。

栈帧又分为一下几个区域：**局部变量表、操作数栈、动态连接、方法出口等**。平时我们所说的变量存在栈中，这句话说的不太严谨，应该说局部变量存放在 java 虚拟机栈的局部变量表中。

java 的 8 中基本类型的局部变量的值存放在虚拟机栈的局部变量表中，如果是引用型的变量，则只存储对象的引用地址。

本地方法栈(Native Stack)

本地方法栈(Native Method Stacks)与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而**本地方法栈则是为虚拟机使用到的 Native 方法服务**。

程序计数器(PC Register)

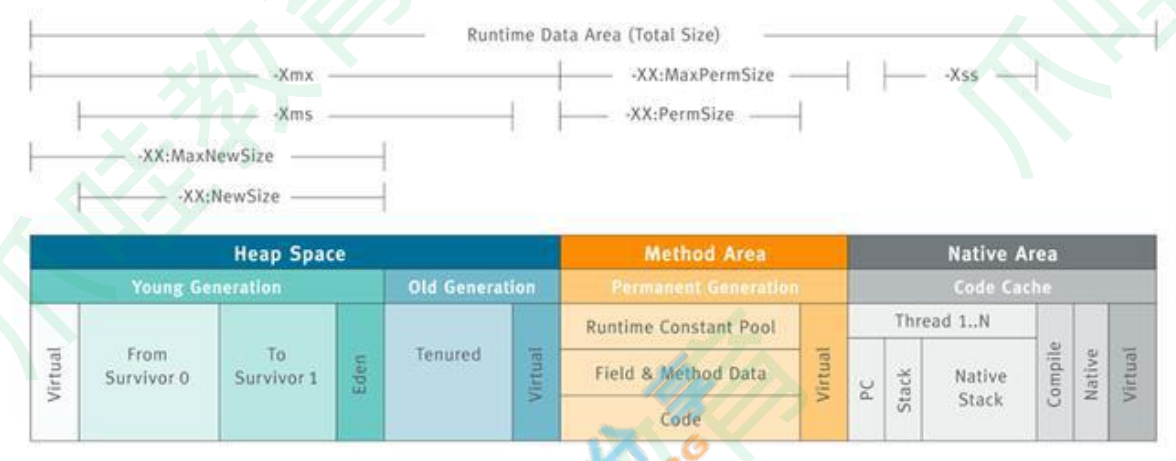
程序计数器就是记录当前线程执行程序的位置，改变计数器的值来确定执行的下一条指令，比如循环、分支、方法跳转、异常处理，线程恢复都是依赖程序计数器来完成。

Java 虚拟机多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。为了线程切换能恢复到正确的位置，每条线程都需要一个独立的程序计数器，所以它是**线程私有**的。

直接内存

直接内存并不是虚拟机内存的一部分，也不是 Java 虚拟机规范中定义的内存区域。jdk1.4 中新加入的 NIO，引入了通道与缓冲区的 IO 方式，它可以调用 Native 方法直接分配堆外内存，这个堆外内存就是本机内存，不会影响到堆内存的大小。

JVM 内存参数设置



- -Xms 设置堆的最小空间大小。
- -Xmx 设置堆的最大空间大小。
- -Xmn: 设置年轻代大小
- -XX:NewSize 设置新生代最小空间大小。
- -XX:MaxNewSize 设置新生代最大空间大小。
- -XX:PermSize 设置永久代最小空间大小。
- -XX:MaxPermSize 设置永久代最大空间大小。
- -Xss 设置每个线程的堆栈大小
- -XX:+UseParallelGC: 选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下, 年轻代使用并发收集, 而年老代仍旧使用串行收集。
- -XX:ParallelGCThreads=20: 配置并行收集器的线程数, 即: 同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

典型 JVM 参数配置参考:

- java-Xmx3550m-Xms3550m-Xmn2g-Xss128k

- `-XX:ParallelGCThreads=20`
- `-XX:+UseConcMarkSweepGC-XX:+UseParNewGC`

`-Xmx3550m`: 设置 JVM 最大可用内存为 3550M。

`-Xms3550m`: 设置 JVM 促使内存为 3550m。此值可以设置与 `-Xmx` 相同, 以避免每次垃圾回收完成后 JVM 重新分配内存。

`-Xmn2g`: 设置年轻代大小为 2G。整个堆大小=年轻代大小+年老代大小+持久代大小。持久代一般固定大小为 64m, 所以增大年轻代后, 将会减小年老代大小。此值对系统性能影响较大, 官方推荐配置为整个堆的 3/8。

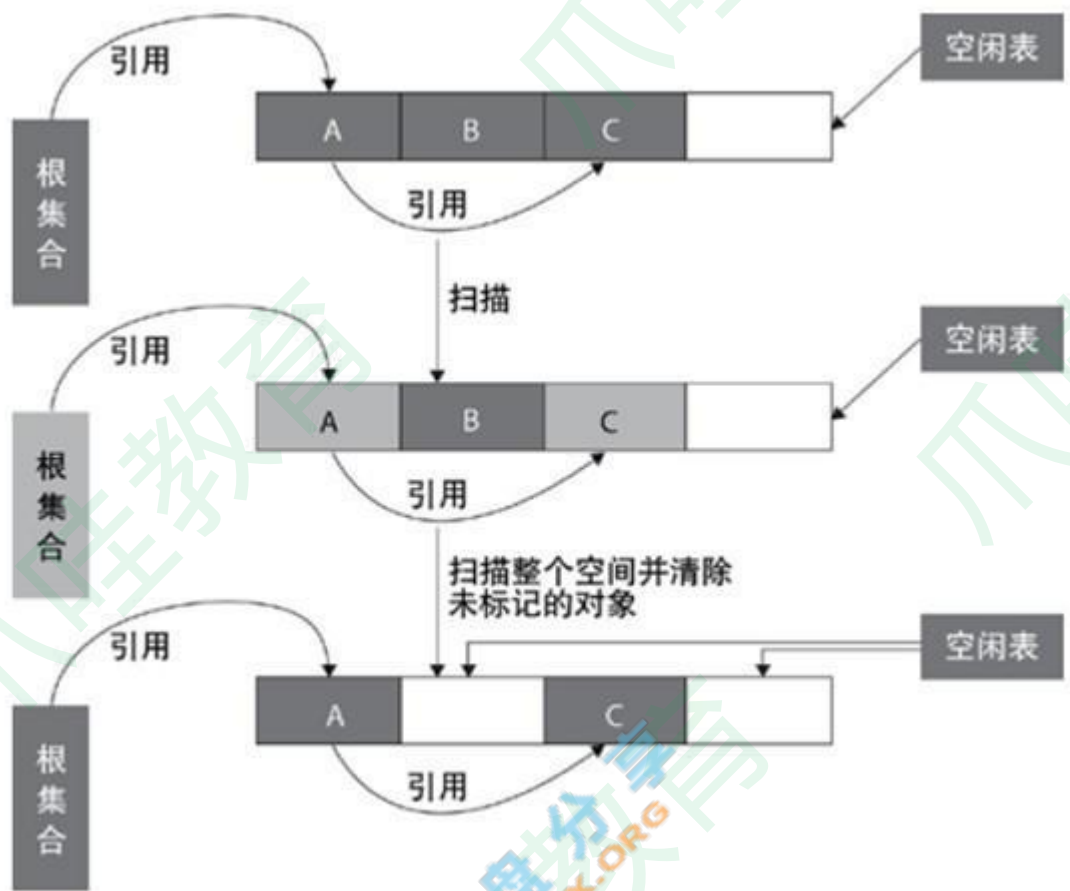
`-Xss128k`: 设置每个线程的堆栈大小。JDK5.0 以后每个线程堆栈大小为 1M, 以前每个线程堆栈大小为 256K。更具应用的线程所需内存大小进行调整。在相同物理内存下, 减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的, 不能无限生成, 经验值在 3000~5000 左右。

垃圾回收算法

1. 标记清除

标记-清除算法将垃圾回收分为两个阶段：**标记阶段**和**清除阶段**。

在标记阶段首先通过**根节点 (GC Roots)**, 标记所有从根节点开始的对象, 未被标记的对象就是未被引用的垃圾对象。然后, 在清除阶段, 清除所有未被标记的对象。



适用场合：

存活对象较多的情况下比较高效

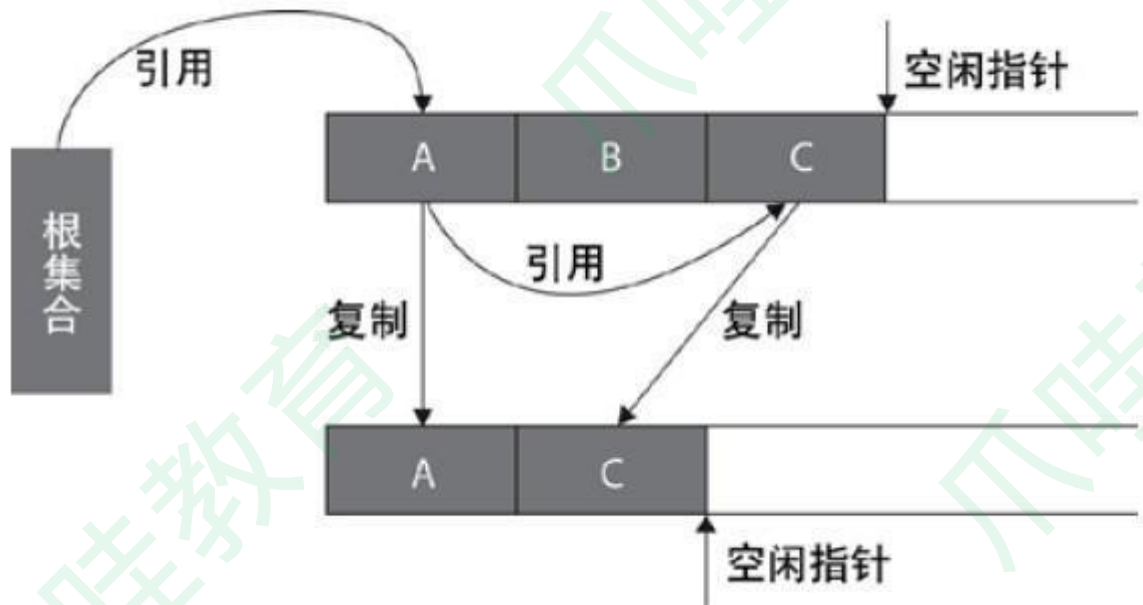
适用于年老代（即旧生代）

缺点：

- 容易产生内存碎片，再来一个比较大的对象时（典型情况：该对象的大小大于空闲表中的每一块儿大小但是小于其中两块儿的和），会提前触发垃圾回收
- 扫描了整个空间两次（第一次：标记存活对象；第二次：清除没有标记的对象）

2. 复制算法

从根集合节点进行扫描，标记出所有的存活对象，并将这些存活的对象复制到一块儿新的内存（图中下边的那一块儿内存）上去，之后将原来的那一块儿内存（图中上边的那一块儿内存）全部回收掉



现在的商业虚拟机都采用这种收集算法来回收新生代。

适用场合：

- 存活对象较少的情况下比较高效
- 扫描了整个空间一次（标记存活对象并复制移动）
- 适用于年轻代（即新生代）：基本上 98% 的对象是“朝生夕死”的，存活下来的会很少

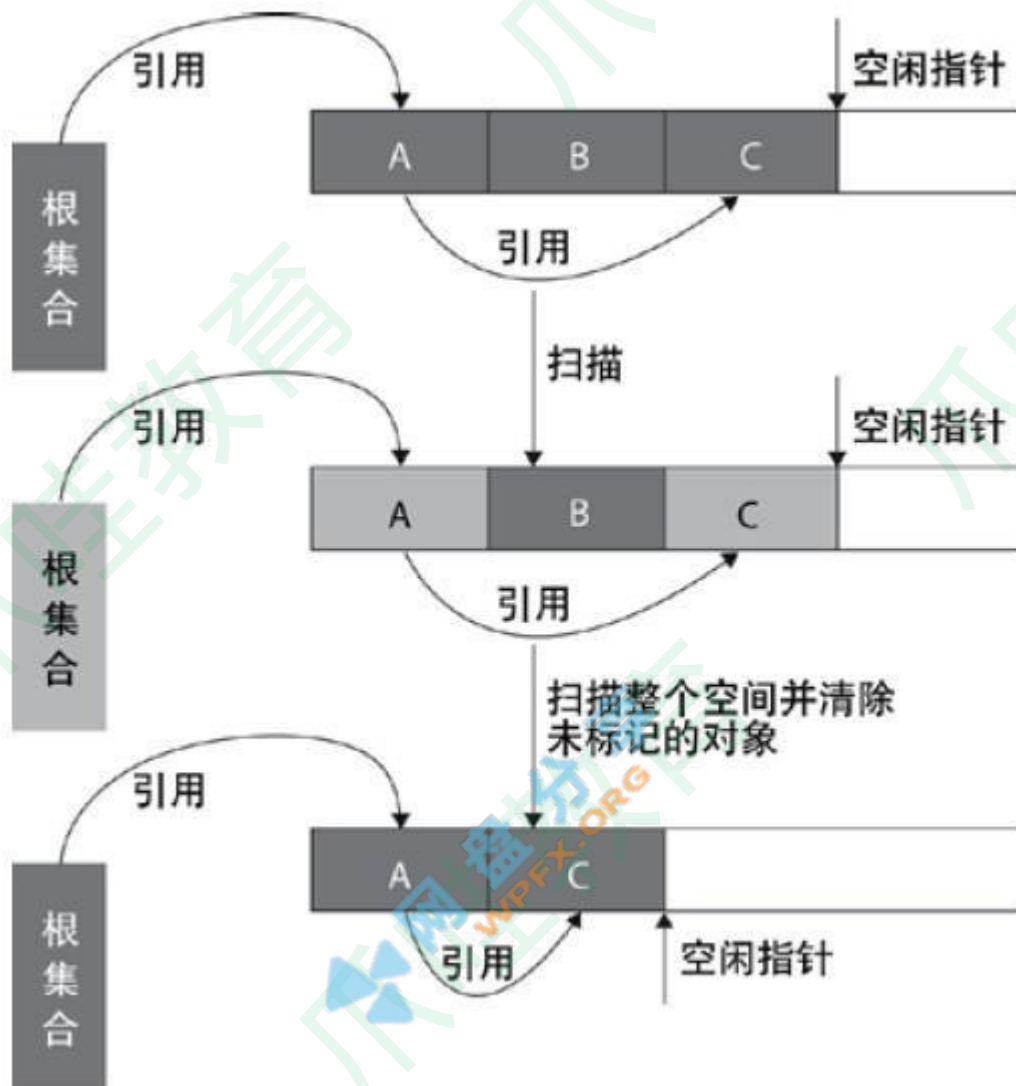
缺点：

- 需要一块儿空的内存空间
- 需要复制移动对象

3. 标记整理

复制算法的高效性是建立在存活对象少、垃圾对象多的前提下的。

这种情况在新生代经常发生，但是在老年代更常见的情况是大部分对象都是存活对象。如果依然使用复制算法，由于存活的对象较多，复制的成本也将很高。



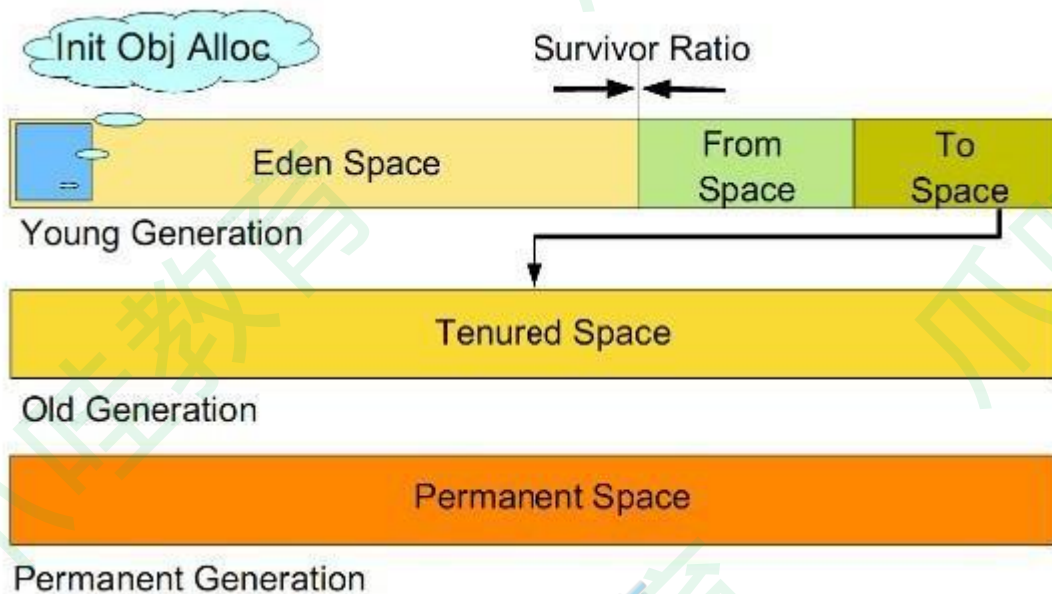
标记-压缩算法是一种老年代的回收算法，它在标记-清除算法的基础上做了一些优化。

首先也需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

4. 分代收集算法

分代收集算法就是目前虚拟机使用的回收算法，它解决了标记整理不适用于老年代的问题，将内存分为各个年代。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），在堆区之外还有一个代就是永久代（Permanet Generation）。

在不同年代使用不同的算法，从而使用最合适的算法，新生代存活率低，可以使用复制算法。而老年代对象存活率搞，没有额外空间对它进行分配担保，所以只能使用标记清除或者标记整理算法。



垃圾回收机制

年轻代分为 Eden 区和 survivor 区（两块儿：from 和 to），且 $\text{Eden}:\text{from}:\text{to}=8:1:1$ 。



jvm 内存结构

- 1) 新产生的对象优先分配在 Eden 区（除非配置了 `-XX:PretenureSizeThreshold`，大于该值的对象会直接进入年老代）；
- 2) 当 Eden 区满了或放不下了，这时候其中存活的对象会复制到 from 区。

这里，需要注意的是，如果存活下来的对象 from 区都放不下，则这些存活下来的对象全部进入年老代。之后 Eden 区的内存全部回收掉。

- 3) 之后产生的对象继续分配在 Eden 区，当 Eden 区又满了或放不下了，这时候将会把 Eden 区和 from 区存活下来的对象复制到 to 区（同理，如果存活下来的

对象 to 区都放不下, 则这些存活下来的对象全部进入年老代), 之后回收掉 Eden 区和 from 区的所有内存。

4) 如上这样, 会有很多对象会被复制很多次 (每复制一次, 对象的年龄就+1), 默认情况下, 当对象被复制了 15 次 (这个次数可以通过: -XX:MaxTenuringThreshold 来配置), 就会进入年老代了。

5) 当年老代满了或者存放不下将要进入年老代的存活对象的时候, 就会发生一次 Full GC (这个是我们最需要减少的, 因为耗时很严重)。

垃圾回收有两种类型: Minor GC 和 Full GC。

1. Minor GC

对新生代进行回收, 不会影响到年老代。因为新生代的 Java 对象大多死亡频繁, 所以 Minor GC 非常频繁, 一般在这里使用速度快、效率高的算法, 使垃圾回收能尽快完成。

2. Full GC

也叫

Major GC, 对整个堆进行回收, 包括新生代和老年代。由于 Full GC 需要对整个堆进行回收, 所以比 Minor GC 要慢, 因此应该尽可能减少 Full GC 的次数, 导致 Full GC 的原因包括: 老年代被写满、永久代 (Perm) 被写满和 System.gc() 被显式调用等。

垃圾回收算法总结

1. 年轻代: 复制算法

1) 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

2)

新生代内存按照 8:1:1 的比例分为一个 eden 区和两个 survivor (survivor0, survivor1) 区。一个 Eden 区, 两个 Survivor 区 (一般而言)。大部分对象在 Eden 区中生成。回收时先将 eden 区存活对象复制到一个 survivor0 区, 然后清空 eden 区, 当这个 survivor0 区也存放满了时, 则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区, 然后清空 eden 和这个 survivor0 区, 此时 survivor0 区是空的, 然后将 survivor0 区和 survivor1 区交换, 即保持 survivor1 区为空, 如此往复。

3) 当 survivor1 区不足以存放 eden 和 survivor0 的存活对象时, 就将存活对象直接存放到老年代。若是老年代也满了就会触发一次 Full GC (Major GC), 也就是新生代、老年代都进行回收。

4) 新生代发生的 GC 也叫做 Minor GC, MinorGC 发生频率比较高(不一定等 Eden 区满了才触发)。

2. 年老代: 标记-清除或标记-整理

1) 在年轻代中经历了 N 次垃圾回收后仍然存活的对象, 就会被放到年老代中。因此, 可以认为年老代中存放的都是一些生命周期较长的对象。

2) 内存比新生代也大很多(大概比例是 1:2), 当老年代内存满时触发 Major GC 即 Full GC, Full GC 发生频率比较低, 老年代对象存活时间比较长, 存活率标记高。

以上这种年轻代与年老代分别采用不同回收算法的方式称为”分代收集算法”, 这也是当下企业使用的一种方式

3. 每一种算法都会有很多不同的垃圾回收器去实现, 在实际使用中, 根据自己的业务特点做出选择就好。

头条 Java 二面参考答案:

哨兵 (sentinel)

哨兵是 Redis 集群架构中非常重要的一个组件, 哨兵的出现主要是解决了主从复制出现故障时需要人为干预的问题。

1. Redis 哨兵主要功能

(1) **集群监控**: 负责监控 Redis master 和 slave 进程是否正常工作

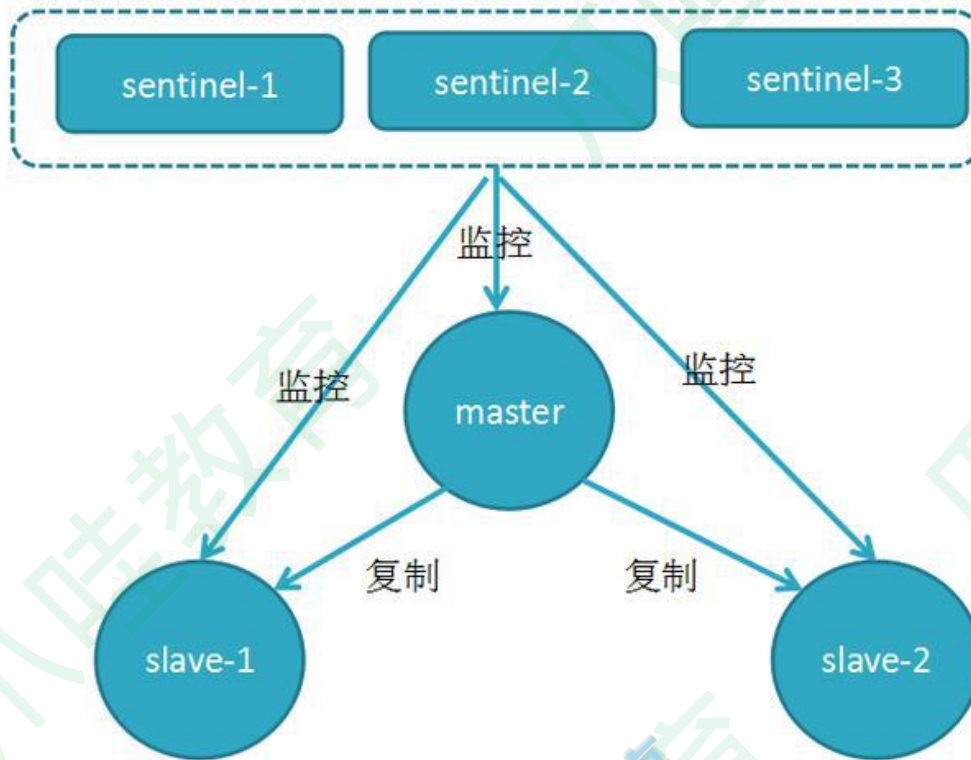
(2) **消息通知**: 如果某个 Redis 实例有故障, 那么哨兵负责发送消息作为报警通知给管理员

(3) **故障转移**: 如果 master node 挂掉了, 会自动转移到 slave node 上

(4) **配置中心**: 如果故障转移发生了, 通知 client 客户端新的 master 地址

2. Redis 哨兵的高可用

原理: 当主节点出现故障时, 由 Redis Sentinel 自动完成故障发现和转移, 并通知应用方, 实现高可用性。



- 哨兵机制建立了多个哨兵节点(进程)，共同监控数据节点的运行状况。
- 同时哨兵节点之间也互相通信，交换对主从节点的监控状况。
- 每隔 1 秒每个哨兵会向整个集群：Master 主服务器+Slave 从服务器+其他 Sentinel（哨兵）进程，发送一次 ping 命令做一次心跳检测。

这个就是哨兵用来判断节点是否正常的重要依据，涉及两个新的概念：**主观下线**和**客观下线**。

1. **主观下线**：一个哨兵节点判定主节点 down 掉是主观下线。
2. **客观下线**：只有半数哨兵节点都主观判定主节点 down 掉，此时多个哨兵节点交换主观判定结果，才会判定主节点客观下线。
3. **原理**：基本上哪个哨兵节点最先判断出这个主节点客观下线，就会在各个哨兵节点中发起投票机制 Raft 算法（选举算法），最终被投为领导者的哨兵节点完成主从自动化切换的过程。

Redis 复制(Replication)

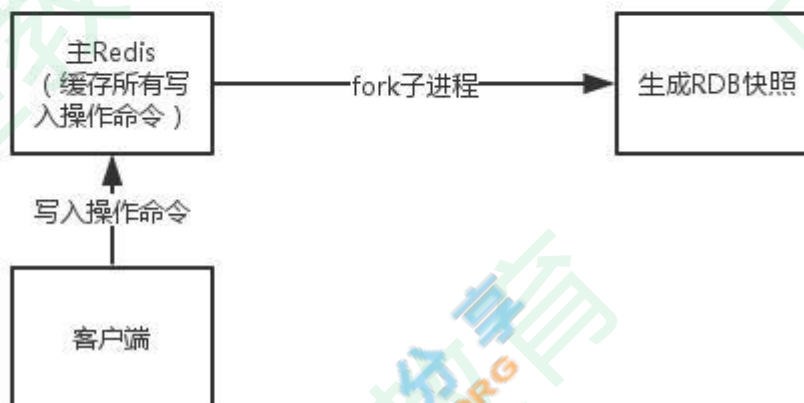
Redis 为了解决单点数据库问题，会把数据复制多个副本部署到其他节点上，通过复制，实现 Redis 的高可用性，实现对数据的冗余备份，保证数据和服务的高度可靠性。

1. 数据复制原理（执行步骤）

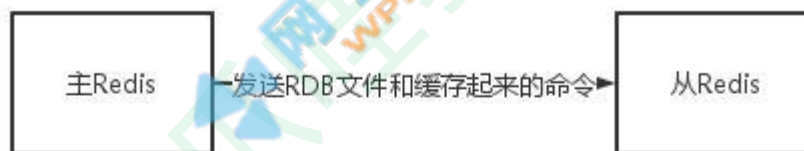
(1) 从Redis服务器启动



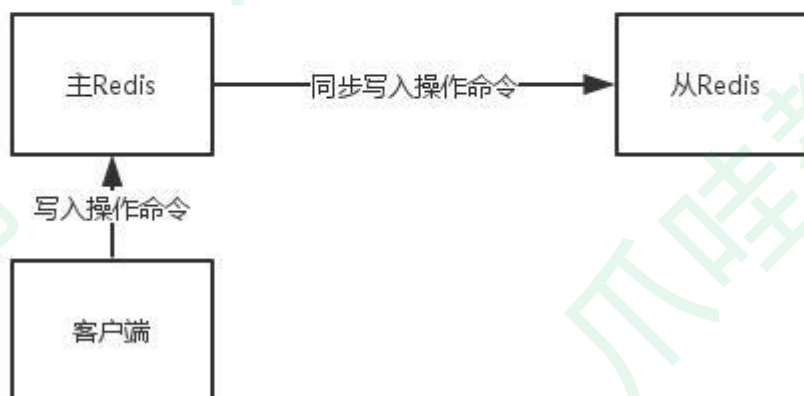
(2) 主Redis服务器接收到SYNC命令后



(3) RDB持久化完成后



(4) 复制初始化完成后



①从数据库向主数据库发送 sync(数据同步) 命令。

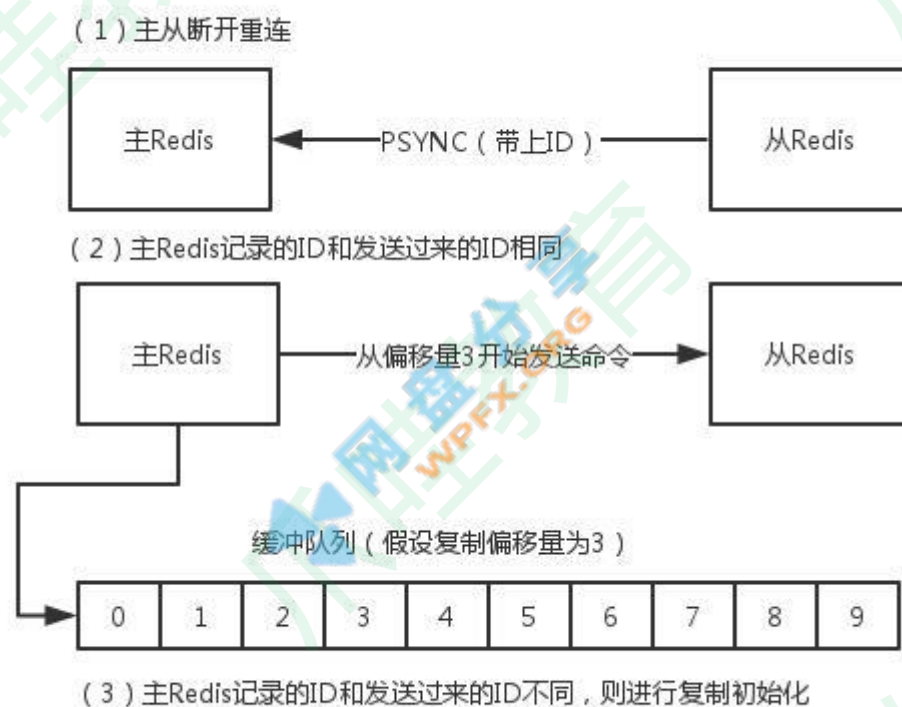
②主数据库接收同步命令后，会保存快照，创建一个 RDB 文件。

③当主数据库执行完保持快照后，会向从数据库发送 RDB 文件，而从数据库会接收并载入该文件。

④主数据库将缓冲区的所有写命令发给从服务器执行。

⑤以上处理完之后，之后主数据库每执行一个写命令，都会将被执行的写命令发送给从数据库。

注意：在 Redis2.8 之后，主从断开重连后会根据断开之前最新的命令偏移量进行增量复制。



Redis 主从复制、哨兵和集群这三个有什么区别

1. **主从模式**：读写分离，备份，一个 Master 可以有多个 Slaves。
2. **哨兵 sentinel**：监控，自动转移，哨兵发现主服务器挂了后，就会从 slave 中重新选举一个主服务器。
3. **集群**：为了解决单机 Redis 容量有限的问题，将数据按一定的规则分配到多台机器，内存/QPS 不受限于单机，可受益于分布式集群高扩展性。

Redis 的高并发和快速原因

1. redis 是基于内存的，内存的读写速度非常快；
2. redis 是单线程的，省去了很多上下文切换线程的时间；
3. redis 使用多路复用技术，可以处理并发的连接。非阻塞 IO 内部实现采用 epoll，采用了 epoll+自己实现的简单的事件框架。epoll 中的读、写、关闭、连接都转化成了事件，然后利用 epoll 的多路复用特性，绝不在 io 上浪费一点时间。

下面重点介绍单线程设计和 IO 多路复用核心设计快的原因。

为什么 Redis 是单线程的

1. 官方答案

因为 Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且 CPU 不会成为瓶颈，那就顺理成章地采用单线程的方案了。

2. 性能指标

关于 redis 的性能，官方网站也有，普通笔记本轻松处理每秒几十万的请求。

3. 详细原因

1) 不需要各种锁的性能消耗

Redis 的数据结构并不全是简单的 Key-Value，还有 list，hash 等复杂的结构，这些结构有可能会进行很细粒度的操作，比如在很长的列表后面添加一个元素，在 hash 当中添加或者删除

一个对象。这些操作可能就需要加非常多的锁，导致的结果是同步开销大大增加。

总之，在单线程的情况下，就不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗。

2) 单线程多进程集群方案

单线程的威力实际上非常强大，每核心效率也非常高，多线程自然是可以比单线程有更高的性能上限，但是在今天的计算环境中，即使是单机多线程的上限也往往不能满足需要了，需要进一步摸索的是多服务器集群化的方案，这些方案中多线程的技术照样是用不上的。

所以单线程、多进程的集群不失为一个时髦的解决方案。

3) CPU 消耗

采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU。

但是如果 CPU 成为 Redis 瓶颈，或者不想让服务器其他 CPU 核闲置，那怎么办？

可以考虑多起几个 Redis 进程，Redis 是 key-value 数据库，不是关系数据库，数据之间没有约束。只要客户端分清哪些 key 放在哪个 Redis 进程上就可以了。

Redis 单线程的优劣势

1. 单进程单线程优势

- 代码更清晰，处理逻辑更简单
- 不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗
- 不存在多进程或者多线程导致的切换而消耗 CPU

2. 单进程单线程弊端

- 无法发挥多核 CPU 性能，不过可以通过在单机开多个 Redis 实例来完善；

IO 多路复用技术

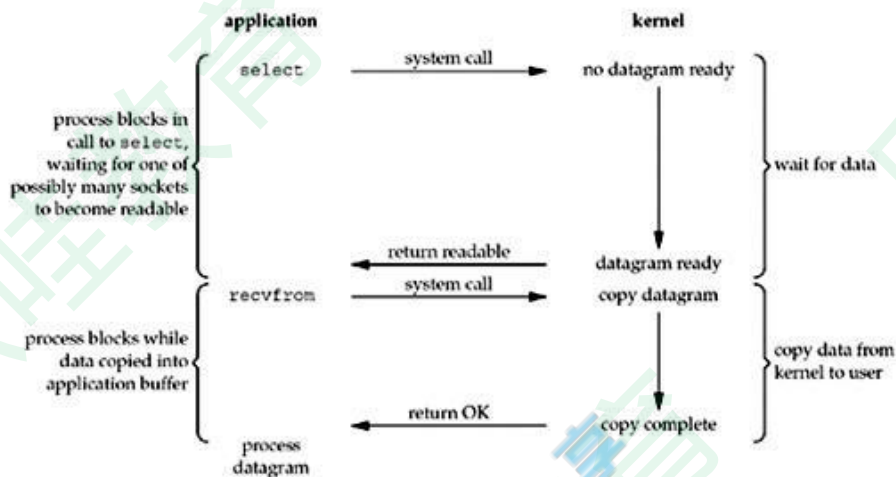
redis 采用网络 IO 多路复用技术来保证在多连接的时候，系统的高吞吐量。

多路-指的是多个 socket 连接，复用-指的是复用一个线程。多路复用主要有三种技术：select, poll, epoll。epoll 是最新的也是目前最好的多路复用技术。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O

复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 I/O 的时间消耗），且 Redis 在内存中操作数据的速度非常快（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了 Redis 具有很高的吞吐量。

Figure 6.3. I/O multiplexing model.



Redis 高并发快总结

1. Redis 是纯内存数据库，一般都是简单的存取操作，线程占用的时间很多，时间的花费主要集中在 I/O 上，所以读取速度快。
2. 再说一下 I/O，Redis 使用的是非阻塞 I/O，I/O 多路复用，使用了单线程来轮询描述符，将数据库的开、关、读、写都转换成了事件，减少了线程切换时上下文的切换和竞争。
3. Redis 采用了单线程的模型，保证了每个操作的原子性，也减少了线程的上下文切换和竞争。
4. 另外，数据结构也帮了不少忙，Redis 全程使用 hash 结构，读取速度快，还有一些特殊的数据结构，对数据存储进行了优化，如压缩表，对短数据进行压缩存储，再如，跳表，使用有序的数据结构加快读取的速度。
5. 还有一点，Redis 采用自己实现的事件分离器，效率比较高，内部采用非阻塞的执行方式，吞吐能力比较大。

数据同步一致性解决方案

1. 半同步复制

办法就是等主从同步完成之后，等主库上的写请求再返回，这就是常说的“半同步复制”。

实现方案

mysql 的半同步复制方案，下面我以 mysql 为例介绍。



MySQL 半同步复制

MySQL 的 Replication 默认是一个异步复制的过程，从 MySQL5.5 开始，MySQL 以插件的形式支持半同步复制，我先谈下异步复制，这样可以更好的理解半同步复制。

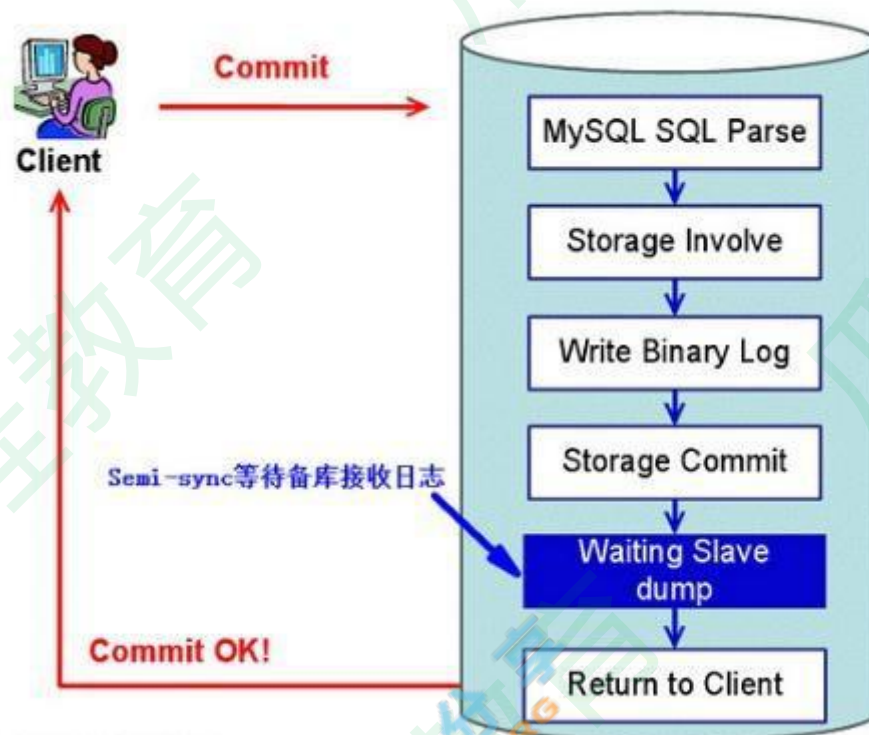
1) 异步复制

MySQL 默认的复制是异步的，主库在执行完客户端提交的事务后会立即将结果返回给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题，主如果 crash 掉了，此时主上已经提交的事务可能并没有传到从库上。

2) 半同步复制

介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到并写到 relay log 中才返回给客户端。相对于异步复制，半同步复制提高了数据的安全性，同

时它也造成了一定程度的延迟，这个延迟最少是一个 TCP/IP 往返的时间。所以，半同步复制最好在低延时的网络中使用。



半同步复制原理：

事务在主库写完 binlog 后需要从库返回一个已接受，才放回给客户端

mysql 5.5 版本以后，以插件的形式存在，需要单独安装

确保事务提交后 binlog 至少传输到一个从库

不保证从库应用完成这个事务的 binlog

性能有一定的降低

网络异常或从库宕机，卡主库，直到超时或从库恢复

该方案优点：

利用数据库原生功能，比较简单

该方案缺点：

主库的写请求时延会增长，吞吐量会降低

2. 数据库中间件



流程：

- 1) 所有的读写都走数据库中间件，通常情况下，写请求路由到主库，读请求路由到从库
- 2) 记录所有路由到写库的 key，在主从同步时间窗口内（假设是 500ms），如果有读请求访问中间件，此时有可能从库还是旧数据，就把这个 key 上的读请求路由到主库。
- 3) 在主从同步时间过完后，对应 key 的读请求继续路由到从库。

相关的中间件有：

- 1) canal：是阿里巴巴旗下的一款开源项目，纯 Java 开发，基于数据库增量日志解析，提供增量数据订阅&消费，目前主要支持了 MySQL。
- 2) otter：也是阿里开源的一个分布式数据库同步系统，尤其是在跨机房数据库同步方面，有很强大的功能。它是基于数据库增量日志解析，实时将数据同步到本机房或跨机房的 `mysql/oracle` 数据库。

两者的区别在于：

otter 目前嵌入式依赖 canal，部署为同一个 jvm，目前设计为不产生 Relay Log。

otter 目前允许自定义同步逻辑，解决各类需求。

该方案优点

能保证绝对一致

该方案缺点：

数据库中间件的成本较高

缓存记录写 key 法



写流程：

1) 如果 key 要发生写操作，记录在 cache 里，并设置“经验主从同步时间”的 cache 超时时间，例如 500ms

2) 然后修改主数据库

读流程：

1) 先到缓存里查看，对应 key 有没有相关数据

2) 有相关数据，说明缓存命中，这个 key 刚发生过写操作，此时需要将请求路由到主库读最新的数据。

3) 如果缓存没有命中，说明这个 key 上近期没有发生过写操作，此时将请求路由到从库，继续读写分离。

该方案优点：

相对数据库中间件，成本较低

该方案缺点：

为了保证“一致性”，引入了一个 cache 组件，并且读写数据库时都多了缓存操作。

头条 Java 三面参考答案

如何解决 Redis 的并发竞争 key 问题

第一种方案：分布式锁

1. 整体技术方案

这种情况，主要是准备一个分布式锁，大家去抢锁，抢到锁就做 set 操作。

2. 为什么是分布式锁

因为传统的加锁的做法（如 java 的 synchronized 和 Lock）这里没用，只适合单点。因为这是分布式环境，需要的是分布式锁。

当然，分布式锁可以基于很多种方式实现，比如 zookeeper、redis 等，不管哪种方式实现，基本原理是不变的：用一个状态值表示锁，对锁的占用和释放通过状态值来标识。

3. 分布式锁的要求

互斥性：在任意一个时刻，只有一个客户端持有锁。

无死锁：即便持有锁的客户端崩溃或者其他意外事件，锁仍然可以被获取。

容错：只要大部分 Redis 节点都活着，客户端就可以获取和释放锁

4. 分布式锁的实现方式

- 数据库
- Memcached (add 命令)
- Redis (setnx 命令)
- Zookeeper (临时节点)

具体的分布式锁实现，请参考：[阿里 P8 架构师谈：分布式锁的 3 种实现（数据库、缓存、Zookeeper）](#)

第二种方案：利用消息队列

在并发量过大的情况下, 可以通过消息中间件进行处理, 把并行读写进行串行化。

把 Redis.set 操作放在队列中使其串行化, 必须的一个一个执行。

这种方式在一些高并发的场景中算是一种通用的解决方案。

数据同步一致性解决方案

1. 半同步复制

办法就是等主从同步完成之后, 等主库上的写请求再返回, 这就是常说的“半同步复制”。

实现方案

mysql 的半同步复制方案, 下面我以 mysql 为例介绍。



MySQL 半同步复制

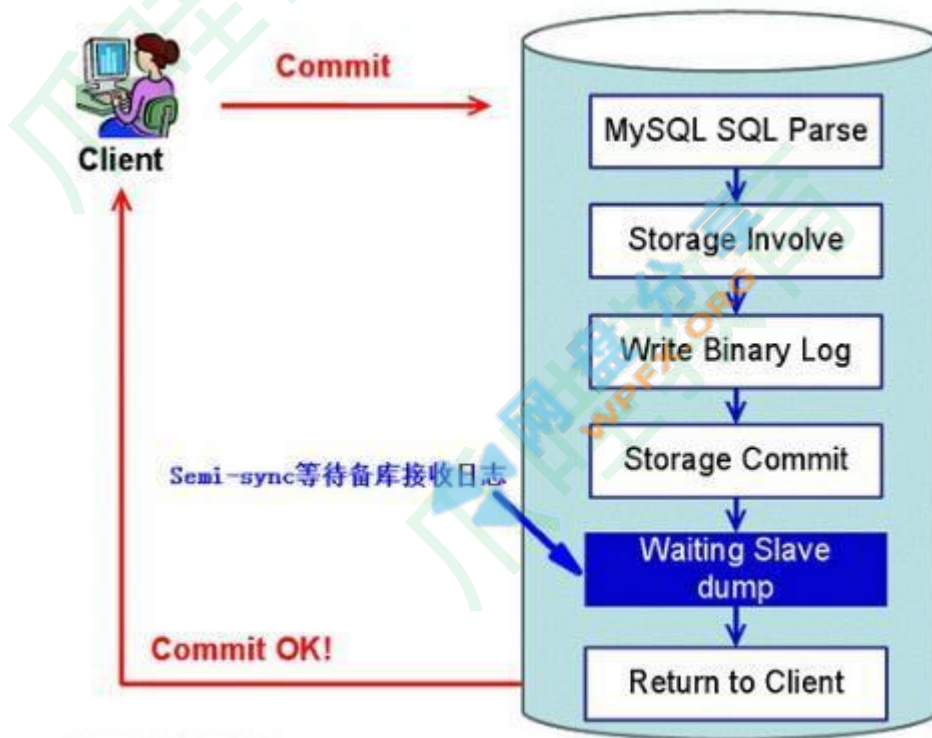
MySQL 的 Replication 默认是一个异步复制的过程, 从 MySQL5.5 开始, MySQL 以插件的形式支持半同步复制, 我先谈下异步复制, 这样可以更好的理解半同步复制。

1) 异步复制

MySQL 默认的复制是异步的，主库在执行完客户端提交的事务后会立即将结果返回给客户端，并不关心从库是否已经接收并处理，这样就会有一个问题，主如果 crash 掉了，此时主上已经提交的事务可能并没有传到从库上。

2) 半同步复制

介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待至少一个从库接收到并写到 relay log 中才返回给客户端。相对于异步复制，半同步复制提高了数据的安全性，同时它也造成了一定程度的延迟，这个延迟最少是一个 TCP/IP 往返的时间。所以，半同步复制最好在低延时的网络中使用。



半同步复制原理：

事务在主库写完 binlog 后需要从库返回一个已接受，才放回给客户端

mysql 5.5 版本以后，以插件的形式存在，需要单独安装

确保事务提交后 binlog 至少传输到一个从库

不保证从库应用完成这个事务的 binlog

性能有一定的降低

网络异常或从库宕机，卡主库，直到超时或从库恢复

该方案优点：

利用数据库原生功能，比较简单

该方案缺点：

主库的写请求时延会增长，吞吐量会降低

2. 数据库中间件



流程：

- 1) 所有的读写都走数据库中间件，通常情况下，写请求路由到主库，读请求路由到从库
- 2) 记录所有路由到写库的 key，在主从同步时间窗口内（假设是 500ms），如果有读请求访问中间件，此时有可能从库还是旧数据，就把这个 key 上的读请求路由到主库。
- 3) 在主从同步时间过完后，对应 key 的读请求继续路由到从库。

相关的中间件有：

- 1) canal：是阿里巴巴旗下的一款开源项目，纯 Java 开发，基于数据库增量日志解析，提供增量数据订阅&消费，目前主要支持了 MySQL。

2) otter: 也是阿里开源的一个分布式数据库同步系统, 尤其是在跨机房数据库同步方面, 有很强大的功能。它是基于数据库增量日志解析, 实时将数据同步到本机房或跨机房的 `mysql/oracle` 数据库。

两者的区别在于:

otter 目前嵌入式依赖 canal, 部署为同一个 jvm, 目前设计为不产生 Relay Log。

otter 目前允许自定义同步逻辑, 解决各类需求。

该方案优点

能保证绝对一致

该方案缺点:

数据库中间件的成本较高

缓存记录写 key 法



写流程:

1) 如果 key 要发生写操作, 记录在 cache 里, 并设置 “经验主从同步时间” 的 cache 超时时间, 例如 500ms

2) 然后修改主数据库

读流程:

- 1) 先到缓存里查看，对应 key 有没有相关数据
- 2) 有相关数据，说明缓存命中，这个 key 刚发生过写操作，此时需要将请求路由到主库读最新的数据。
- 3) 如果缓存没有命中，说明这个 key 上近期没有发生过写操作，此时将请求路由到从库，继续读写分离。

该方案优点：

相对数据库中间件，成本较低

该方案缺点：

为了保证“一致性”，引入了一个 cache 组件，并且读写数据库时都多了缓存操作。

缓存和数据库一致性解决方案

第一种方案：采用延时双删策略

在写库前后都进行 `redis.del(key)` 操作，并且设定合理的超时时间。

伪代码如下

```
public void write(String key, Object data) {  
    redis.delKey(key);  
    db.updateData(data);  
    Thread.sleep(500);  
    redis.delKey(key);  
}
```

2. 具体的步骤就是：

- 1) 先删除缓存
- 2) 再写数据库
- 3) 休眠 500 毫秒
- 4) 再次删除缓存

那么，这个 500 毫秒怎么确定的，具体该休眠多久呢？

需要评估自己的项目的读数据业务逻辑的耗时。这么做的目的，就是确保读请求结束，写请求可以删除读请求造成的缓存脏数据。

当然这种策略还要考虑 redis 和数据库主从同步的耗时。最后的的写数据的休眠时间：则在读数据业务逻辑的耗时基础上，加几百 ms 即可。比如：休眠 1 秒。

3. 设置缓存过期时间

从理论上来说，给缓存设置过期时间，是保证最终一致性的解决方案。所有的写操作以数据库为准，只要到达缓存过期时间，则后面的读请求自然会从数据库中读取新值然后回填缓存。

4. 该方案的弊端

结合双删策略+缓存超时设置，这样最差的情况就是在超时时间内数据存在不一致，而且又增加了写请求的耗时。

第二种方案：异步更新缓存(基于订阅 binlog 的同步机制)

1. 技术整体思路：

MySQL binlog 增量订阅消费+消息队列+增量数据更新到 redis

- 1) 读 Redis：热数据基本都在 Redis
- 2) 写 MySQL：增删改都是操作 MySQL
- 3) 更新 Redis 数据：MySQL 的数据操作 binlog，来更新到 Redis

2. Redis 更新

1) 数据操作主要分为两大块：

一个是全量(将全部数据一次写入到 redis)

一个是增量(实时更新)

这里说的是增量，指的是 mysql 的 update、insert、delete 变更数据。

2) 读取 binlog 后分析，利用消息队列, 推送更新各台的 redis 缓存数据。

这样一旦 MySQL 中产生了新的写入、更新、删除等操作，就可以把 binlog 相关的消息推送至 Redis，Redis 再根据 binlog 中的记录，对 Redis 进行更新。

其实这种机制，很类似 MySQL 的主从备份机制，因为 MySQL 的主备也是通过 binlog 来实现的数据一致性。

这里可以结合使用 canal (阿里的一款开源框架)，通过该框架可以对 MySQL 的 binlog 进行订阅，而 canal 正是模仿了 mysql 的 slave 数据库的备份请求，使得 Redis 的数据更新达到了相同的效果。

当然，这里的消息推送工具你也可以采用别的第三方：kafka、rabbitMQ 等来实现推送更新 Redis。

秒杀活动场景

淘宝双 11 秒杀场景，大量的用户短时间内涌入，瞬间流量巨大（高并发），比如：1000 万人同一时间抢购 100 件商品。秒杀活动是一个特别考验后台数据库、缓存服务的业务，对于数据库、缓存的性能要求特别严格。

秒杀背后的技术挑战

1、突增的服务器及网络需求

通常情况下，双 11 的服务器使用是平时的 3-5 倍，网络带宽是平时 N 倍。

2、业务高并发，服务负载重

我们通常衡量一个 Web 系统的吞吐率的指标是 QPS (Query Per Second, 每秒处理请求数)，解决每秒数万次的高并发场景，这个指标非常关键。

假设处理一个业务请求平均响应时间为 100 ms，同时，系统内有 20 台 Web 服务器，配置最大连接数为 500 个，Web 系统的理论峰值 QPS 为（理想化的计算方式）：100000（10 万 QPS）意味着 1 秒钟可以处理完 10 万的请求，而“秒杀”的那 5w/s 的秒杀似乎是“纸老虎”。

实际情况，在高并发的实际场景下，服务器处于高负载的状态，网络带宽被挤满，在这个时候平均响应时间会被大大增加。随着用户数量的增加，数据库连接进程增加，需要处理的上下文切换也越多，服务器造成负载压力越来越重。

3、业务耦合度高，引起系统“雪崩”

更可怕的问题是，当系统上某个应用因为延迟而变得不可用，用户的点击越频繁，恶性循环最终导致“雪崩”，因为其中一台服务器挂了，导致流量分散到其他正常工作的机器上，再导致正常的机器也挂，然后恶性循环，将整个系统拖垮。

如何解决秒杀技术瓶颈

秒杀架构设计思路：

将请求拦截在系统上游，降低下游压力：秒杀系统特点是并发量极大，但实际秒杀成功的请求数量却很少，所以如果不在前端拦截很可能造成数据库读写锁冲突，甚至导致死锁，最终请求超时。

充分利用缓存(redis)：利用缓存可极大提高系统读写速度。

消息中间件(ActiveMQ、Kafka 等)：消息队列可以削峰，将拦截大量并发请求，这也是一个异步处理过程，后台业务根据自己的处理能力，从消息队列中主动的拉取请求消息进行业务处理。

前端设计方案

页面静态化：将活动页面上的所有可以静态的元素全部静态化，并尽量减少动态元素。通过 CDN 来抗峰值。

禁止重复提交：用户提交之后按钮置灰，禁止重复提交

用户限流：在某一时间段内只允许用户提交一次请求，比如可以采取 IP 限流

后端设计方案

- **服务端控制器层(网关层)**
- **限制 uid (UserID) 访问频率：**我们上面拦截了浏览器访问的请求，但针对某些恶意攻击或其它插件，在服务端控制层需要针对同一个访问 uid，限制访问频率。
- **服务层**

上面只拦截了一部分访问请求，当秒杀的用户量很大时，即使每个用户只有一个请求，到服务层的请求数量还是很大。比如我们有 100W 用户同时抢 100 台手机，服务层并发请求压力至少为 100W。

- **采用消息队列缓存请求：**既然服务层知道库存只有 100 台手机，那完全没有必要把 100W 个请求都传递到数据库啊，那么可以先把这些请求都写到消息队列缓存一下，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。

- **利用缓存应对读请求：**比如双 11 秒杀抢购，是典型的读多写少业务，大部分请求是查询请求，所以可以利用缓存分担数据库压力。
- **利用缓存应对写请求：**缓存也是可以应对写请求的，比如我们就可以把数据库中的库存数据转移到 Redis 缓存中，所有减库存操作都在 Redis 中进行，然后再通过后台进程把 Redis 中的用户秒杀请求同步到数据库中。

数据库层

数据库层是最脆弱的一层，一般在应用设计时在上游就需要把请求拦截掉，数据库层只承担“能力范围内”的访问请求。所以，上面通过在 service 层引入队列和缓存，让最底层的数据库高枕无忧。

比如：利用消息中间件和缓存实现简单的秒杀系统

Redis 是一个分布式缓存系统，支持多种数据结构，我们可以利用 Redis 轻松实现一个强大的秒杀系统。

我们可以采用 Redis 最简单的 key-value 数据结构，用一个原子类型的变量值 (AtomicInteger) 作为 key，把用户 id 作为 value，库存数量便是原子变量的最大值。对于每个用户的秒杀，我们使用 RPush key value 插入秒杀请求，当插入的秒杀请求数达到上限时，停止所有后续插入。

然后我们可以在后台启动多个工作线程，使用 LPop key 读取秒杀成功者的用户 id，然后再操作数据库做最终的下订单减库存操作。

当然，上面 Redis 也可以替换成消息中间件如 ActiveMQ、Kafka 等，也可以将缓存和消息中间件 组合起来，缓存系统负责接收记录用户请求，消息中间件负责将缓存中的请求同步到数据库。

秒杀架构设计总结：

限流：鉴于只有少部分用户能够秒杀成功，所以要限制大部分流量，只允许少部分流量进入服务后端。

削峰：对于秒杀系统瞬时会有大量用户涌入，所以在抢购一开始会有很高的瞬间峰值。高峰值流量是压垮系统很重要的原因，所以如何把瞬间的高流量变成一段时间平稳的流量也是设计秒杀系统很重要的思路。实现削峰的常用的方法有利用缓存和消息中间件等技术。

异步处理：秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式。

内存缓存：秒杀系统最大的瓶颈一般都是数据库读写，由于数据库读写属于磁盘IO，性能很低，如果能够把部分数据或业务逻辑转移到内存缓存，效率会有极大地提升。

可拓展：当然如果我们想支持更多用户，更大的并发，最好就将系统设计成弹性可拓展的，如果流量来了，拓展机器就好了。像淘宝、京东等双十一活动时会增加大量机器应对交易高峰。

