## 类加载机制

### 基础

|  |
| --- |
|  |
| 上面的那个图不完整，下图接上代码和代码输出的部分 |
|  |

### 双亲委派机制

|  |
| --- |
| 为什么要涉及双亲委派机制： |
| 当启动类加载器加载上面的这个类的时候，启动类加载器发现java.lang包下有一个String.clas，也就是说，它找的是类库种的String.class，而不是自己定义的这个String.class，如果想要让启动类加载器加载到这个类，那么需要通过自定义加载器来加载(需要自己定义一个加载自定义的类的自定义类加载器) |
| **A a = null;这段代码会被加载吗？**  这里是不会被加载的，因为这个既不是main方法，也不是new的类  当a对象在下面用到了的时候才会被加载。 |

### 打破双亲委派机制

|  |
| --- |
| **实现思路：**  在自定义类加载器种重写loadClass()方法(将双亲委派的东西去掉或者改写一下)    **Tomcat自定义类加载器** |

## JVM整体结构解析

|  |
| --- |
|  |
| ***栈***：  **FILO: first in last out: 先进后出**  **局部变量表**：存放方法种的局部变量  **操作数栈**：栈空间种对于操作数的临时存放的区域  **动态链接**：  **方法出口**：方法执行完后返回上一级方法的执行位置(也可以简单的理解为上一级方法的程序计数器)    **程序计数器**：计算当前线程执行到的位置(可以简单的理解成行号)，每个线程都会拥有一个程序计数器。 |
| **一般生产上服务器对堆栈的分区**：  对于双核4G的服务器一般是这样设置的：    **但是要注意：**  **对于JVM参数大小的设置，没有固定的标准，需要根据项目的实际情况进行分析。** |
| ***方法区：***  常量、静态变量、类元信息存放处  类元信息：一个类的组成部分  在new一个对象的时候，会生成一个对象头指针，是通过对象指到对象的类元信息的，如果静态变量是一个对象类型的，还会有一个指针指向堆中的对象(下图中的从方法区指向堆的黑色箭头)，例如：  public static User user = new User(); |
| 本地方法栈：  本地方法(native修饰的方法)存放的地方 |

### 堆(最重要的部分)

|  |
| --- |
|  |
| 新new出来的对象是放多数在伊甸园区的(上图中的Eden的位置)。  伊甸园区(Eden)的内存被放满，会触发一次**minor gc**， 将Eden中和Survivor中没有引用的垃圾对象进行回收(主要是靠字节码执行引擎开启的专用线程来进行)，还有一部分存活的对象，会被移动到Survivor的From区，然后再等伊甸园区满了，又会触发一次minor gc，将垃圾清理，然后将存活的对象放入To区域，如此循环。当对象的年龄到了15之后，会被一次性移动到老年代。  终有一天，老年代会被放满，这时候会执行一次full gc，这时会堆整个堆甚至方法区进行清理收集。此时会触发STW(stop the world)来停止应用程序的所有线程的执行。  **如果不执行STW会怎么样？**:  会不停的产生垃圾数据，不断进行full gc，然后几乎所有的对象都会被回收，然后就很可能会导致各种空指针以及其他类型的报错使得程序挂掉。 |

### 逃逸分析

|  |
| --- |
|  |
|  |

## 垃圾收集机制与算法详解

### JVM内存分配与回收

1. 对象优先在Eden区分配

|  |
| --- |
|  |
|  |

1. 大对象直接进入老年代

|  |
| --- |
|  |

1. 长期存活的对象将进入老年代

|  |
| --- |
|  |

1. 对象动态年龄判断

|  |
| --- |
|  |

1. Minor gc后存活的对象在Survivor区放不下

|  |
| --- |
|  |

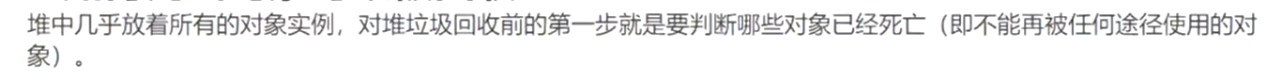
1. 老年代空降分配担保机制(很重要)

|  |
| --- |
| 图示： |

1. Eden于Survivor区默认空间分配8:1:1

|  |
| --- |
|  |

### 如何判断对象可以被回收



1. 引用计数法

|  |
| --- |
| **这个引用计数器法有一个问题：**  如上面的代码，会导致objA和objB相互引用，计数器的值都是1，这样就不会被回收，最终可能会导致内存溢出。所以现在不怎么用了 |

1. 可达性分析算法(用的比较多的算法)

|  |
| --- |
| 从栈内存里面遍历所有的局部变量表的局部变量，看里面是否引用，有引用的标记为非垃圾对象，没有引用 的标记为垃圾对象，回收 |

1. 常见引用类型

|  |
| --- |
|  |

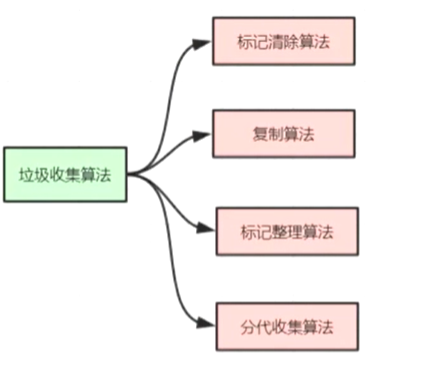
1. finalize()方法最终判断对象是否存活(用的比较少)

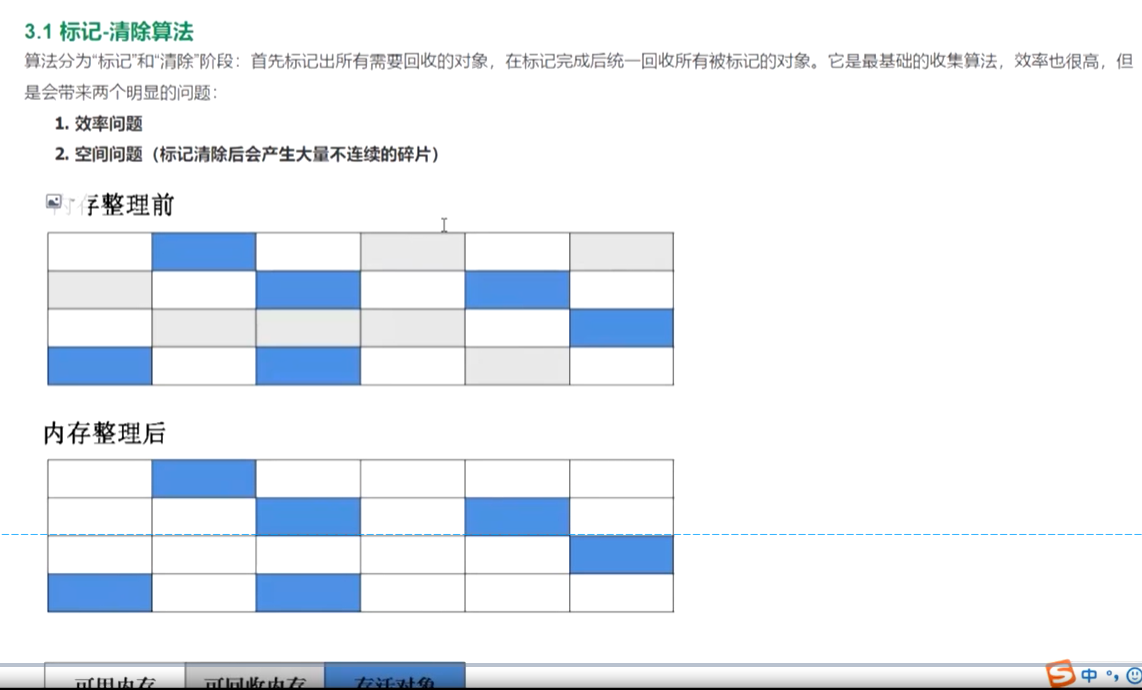
|  |
| --- |
|  |

1. 如何判断一个类是无用的类

|  |
| --- |
|  |

### 垃圾回收算法



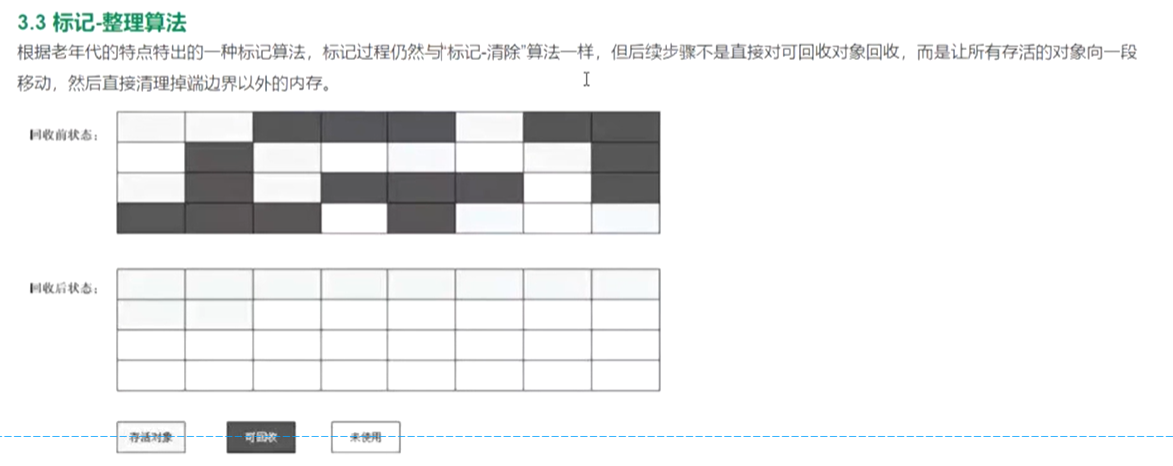




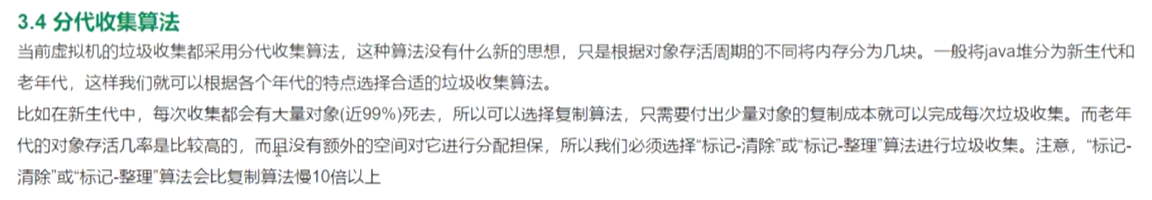
1. 复制算法



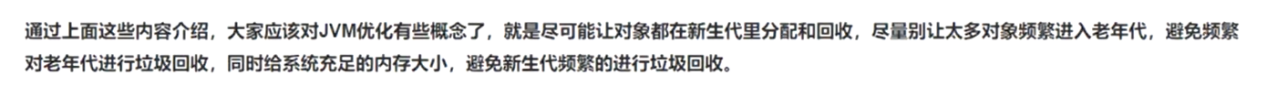
1. 标记-整理算法



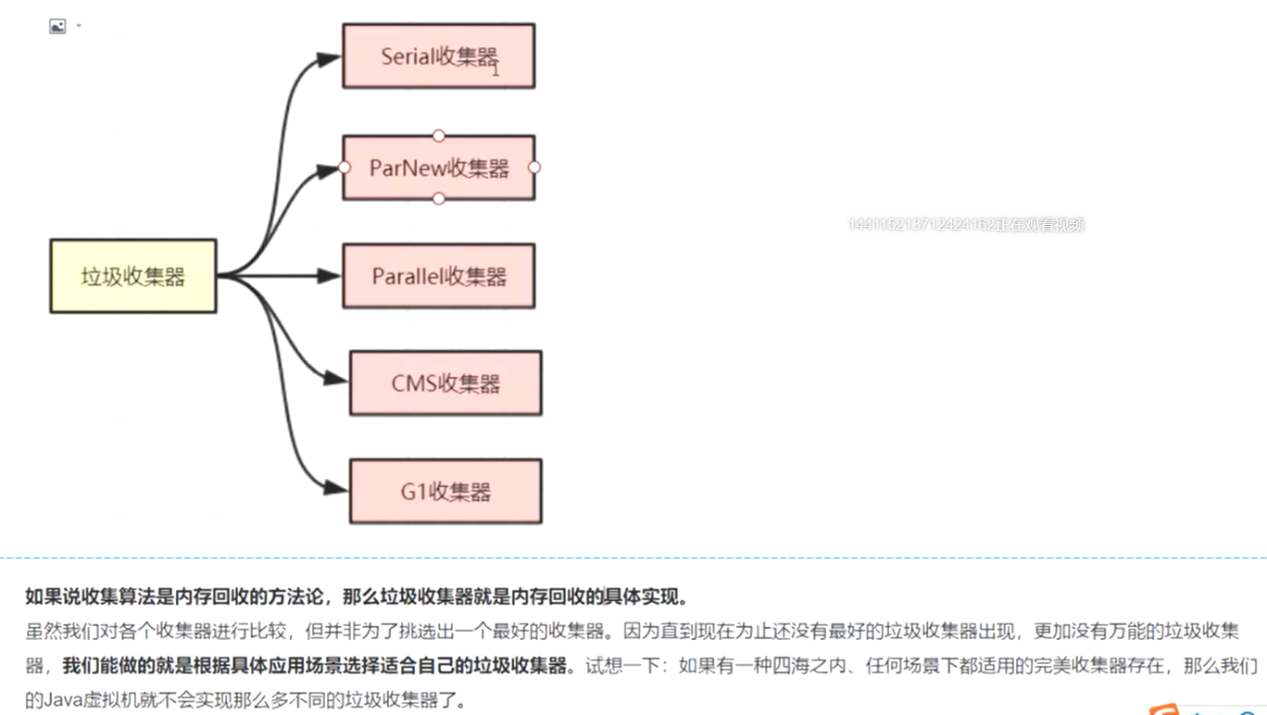
1. 分代收集算法



总结：



## 垃圾收集器



**注意：**没有那种垃圾收集器是最强大的，需要对症下药。

### Serial收集器(老年代)

|  |
| --- |
| **弊端：**多核服务器的话，效率会比较低 |

### ParNew收集器(可以理解成1.1的多线程版本)(年轻代)

|  |
| --- |
|  |

### Parallel Scavenge收集器

|  |
| --- |
|  |

### CMS垃圾收集器(重要)(只用在老年代)

|  |
| --- |
|  |
| **优缺点：** |
| **CMS相关参数：** |

### G1垃圾收集器(大型服务器，重要)

#### 概念

|  |
| --- |
|  |
| **（上面的内容类比下*快递柜*）** |

#### 特点

|  |
| --- |
|  |

#### 参数设置

|  |
| --- |
|  |

#### G1垃圾收集器分类

|  |
| --- |
|  |

#### G1垃圾收集器优化建议：

|  |
| --- |
|  |

#### 什么场景适合使用G1：

|  |
| --- |
|  |

#### 每秒几十万并发的系统如何优化JVM

|  |
| --- |
|  |

### 如何选择垃圾收集器(这里是建议)

|  |
| --- |
|  |

## 调优工具

### Jmap

从视频的1小时25分钟左右开始

|  |
| --- |
|  |
|  |

### jstack(排查死锁)

|  |
| --- |
|  |

jstat(重要)

|  |
| --- |
| jstat -gc 端口号  1590205994(1) |

JVM运行情况预估

|  |
| --- |
|  |

系统频繁Full GC导致卡顿是咋回事

|  |
| --- |
|  |
|  |

### 内存泄漏

|  |
| --- |
|  |

## GC调优实战二

### GC日志详解

|  |
| --- |
|  |

如何分析GC日志

|  |
| --- |
|  |

日志工具

GCeasy

|  |
| --- |
| https://gcwasy.io |
| 里面有分析以及解决方案(解决方案现在收费了) |

## Class常量池

|  |
| --- |
|  |
| 1590216175(1) |

静态常量池和运行时常量池

静态常量池：编译出来的字节码文件中的常量池的代码部分；

运行时常量池：静态常量池的代码放到内存中运行后是运行时常量池

字符串常量池

|  |
| --- |
|  |

面试题

|  |
| --- |
|  |

操作字符串常量池的方式

|  |
| --- |
|  |

八种基本类型的包装类和对象池

|  |
| --- |
|  |

安全点与安全区域

|  |
| --- |
|  |