



# Android高级开发正式课

码牛学院-用代码码出牛逼人生

# android人员的专属JVM讲解

## 02-对象分配过程完全解析

技术点:

- 1.堆区结构详解
- 2.对象分配过程解析
- 3.对象创建过程解析
- 4.对象内存布局解析
- 5.MinorGc/MajorGc/FullGc对比与GC日志分析
- 6.透过逃逸分析完成代码编写优化



# 码牛学院Android讲师介绍

## 码牛学院-Kerwin老师 系统架构师、技术总监

◆10年互联网行业从业经验，架构师  
精通JAVA,C,C++,Android,IOS

前华为工程师，后出任两家公司技术总监，高校外聘讲师，省公安厅电子物证鉴定专家

拥有多个大型分布式系统架构设计与实施和移动终端系统架构设计经验

有丰富的分布式，高并发实战经验，  
开发过多套企业级自定义框架  
擅长系统底层架构，移动终端系统架构





## 课程安排



01

堆的核心结构解析



02

对象创建过程



03

GC日志分析



04

透过逃逸分析完成代码编写优化



01

---

## 堆的核心结构解析



# 运行时数据区



# 堆概述

1. 一个JVM进程存在一个堆内存，堆是JVM内存管理的核心区域
2. java 堆区在JVM启动是被创建，其空间大小也被确定，是JVM管理的最大一块内存（堆内存大小可以调整）
3. 本质上堆是一组在物理上不连续的内存空间，但是逻辑上是连续的空间（参考上节课HSDB分析的内存结构）
4. 所有线程共享堆，但是堆内对于线程处理还是做了一个线程私有的部分（TLAB）



# 堆的对象管理

在《JAVA虚拟机规范》中对Java堆的描述是：所有的对象示例以及数组都应当在运行时分配在堆上

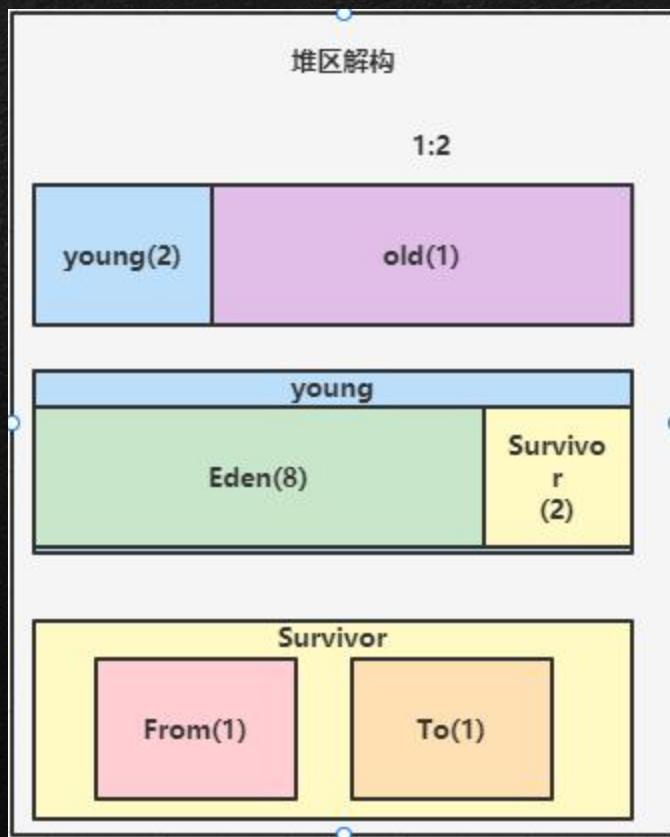
但是从实际使用角度来看，不是绝对，存在某些特殊情况下的对象产生是不在堆上分配

这里请注意，规范上是绝对、实际上是相对

方法结束后，堆中的对象不会马上移除，需要通过GC执行垃圾回收后才会回收



# 堆的内存细分



Java7之前内存逻辑划分为：

新生区+养老区+永久区

Java8之后内存逻辑划分为：

新生去+养老区+元空间

实际上不管永久代与元空间其实都是只方法区中对于长期存在的常量对象的保存



# 体会堆空间的分代思想

为什么需要分代？有什么好处？

经研究表明，不同对象的生命周期不一致，但是在具体使用过程中70%-90的对象是临时对象

分代唯一的理由是优化GC性能。如果没有分代，那么所有对象在一块空间，GC想要回收扫描他就必须扫描所有的对象，分代之后，长期持有的对象可以挑出，短期持有的对象可以固定在一个位置进行回收，省掉很大一部分空间利用



# 堆的默认大小

默认空间大小：

初始大小：物理电脑内存大小 / 64

最大内存大小：物理电脑内存大小 / 4



# 工具: jstat

```
C:\Users\Administrator>jstat -gc 238180
S0C    S1C    S0U    S1U      EC      EU      OC      OU      MC      MU      CCSC   CCSU   YGC     YGCT   FGC     FGCT     GCT
10752.0 10752.0  0.0     0.0   65536.0 6553.8 175104.0  0.0   4480.0 775.8   384.0   76.4     0     0.000   0     0.000   0.000
```

结尾C 代表总量

结尾U代表已使用量

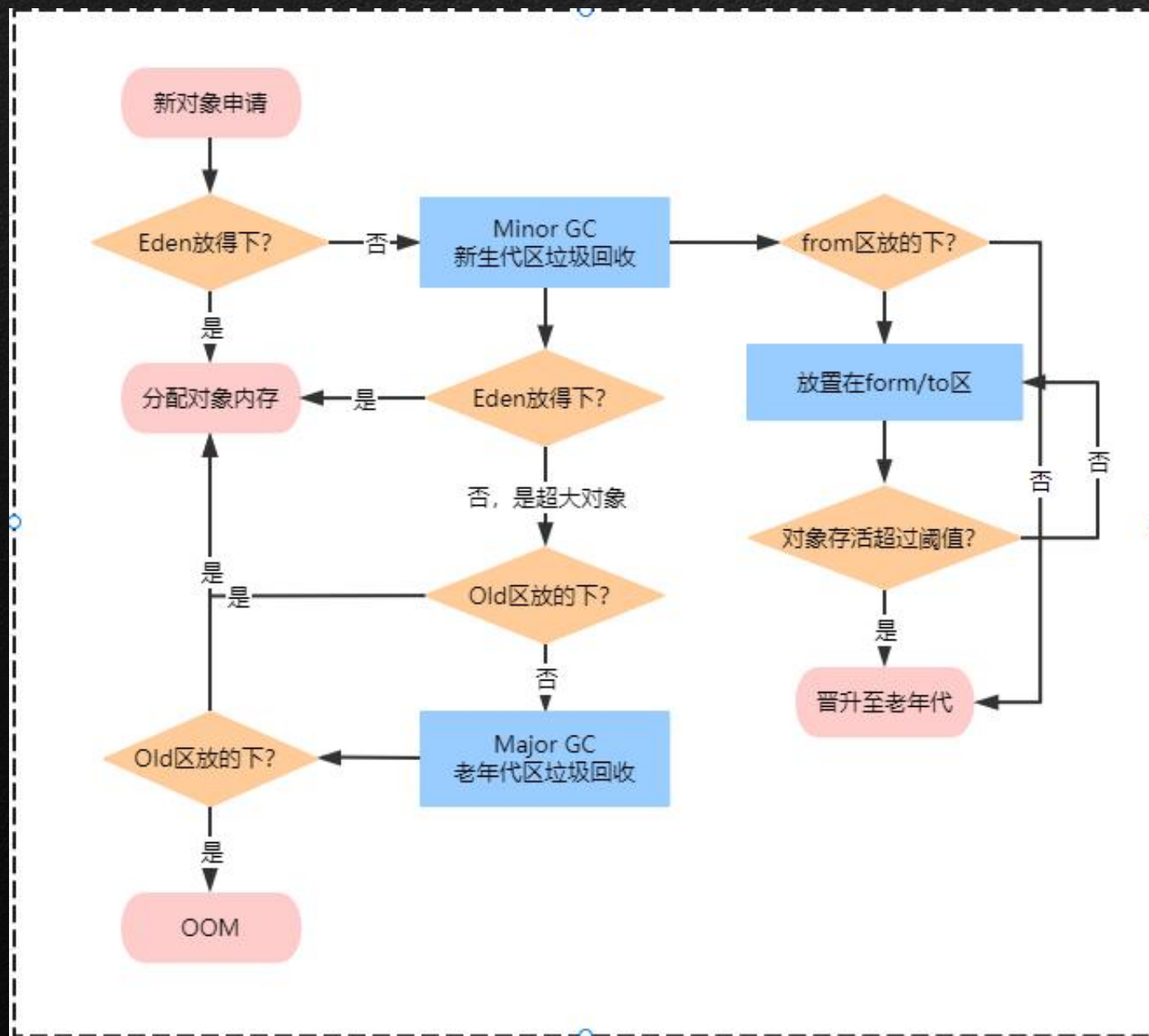
S0 S1代表 survivor区的From 与 To

E代表的是 Eden区

OC代表 老年总量 OU代表老年使用量



# 对象分配过程图示



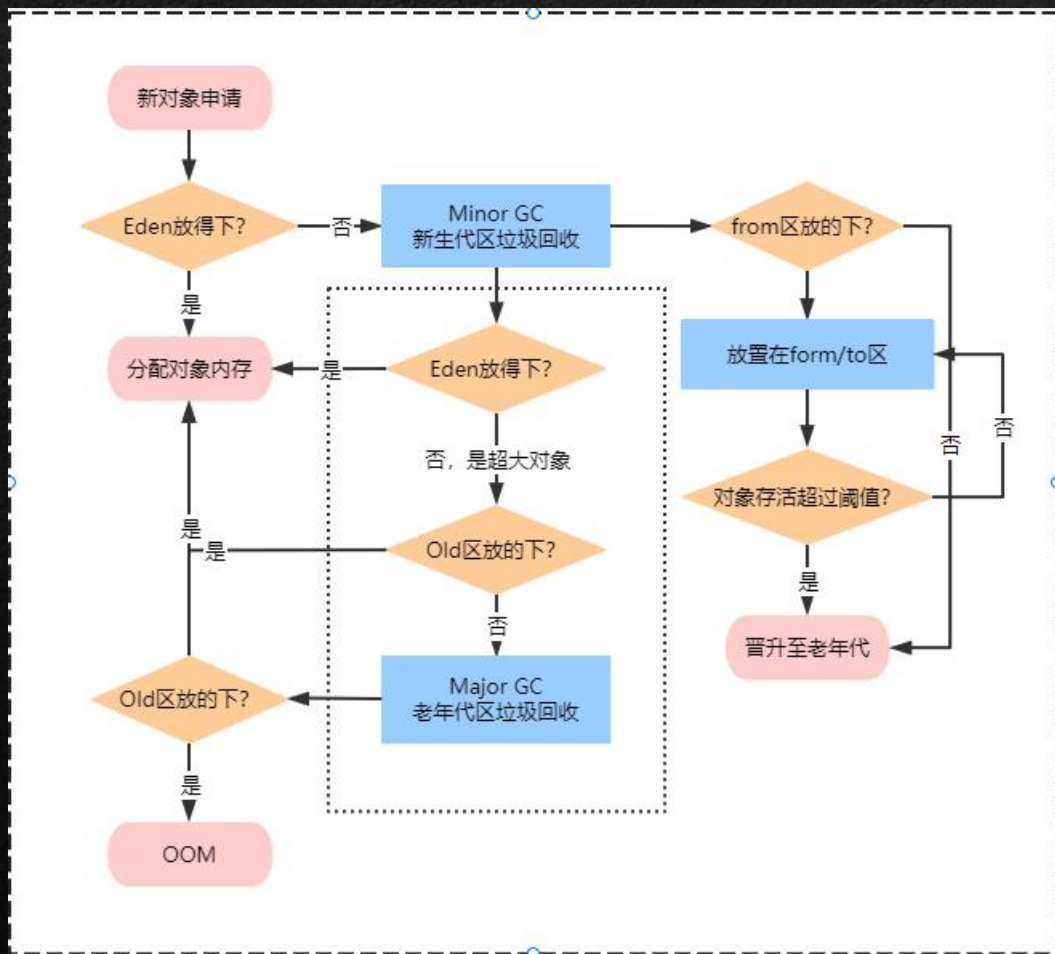


# Kerwin的对象生产过程自述

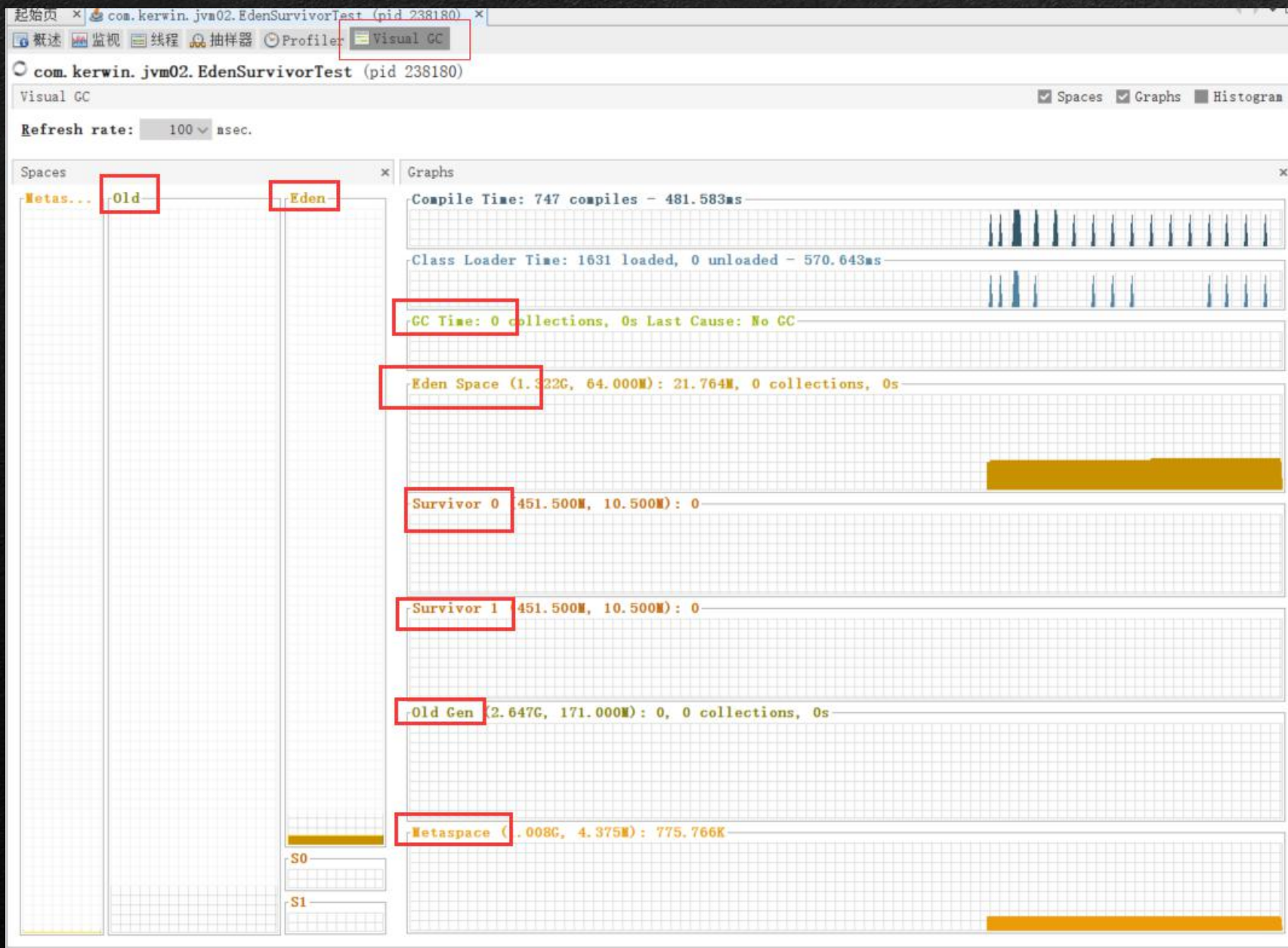
- 1.我是一个普通的java对象，我出生在Eden区，在Eden区我还看到和我长的很像的小兄弟，我们在Eden区中玩了挺长时间。
- 2.有一天Eden区中的人实在是太多了，我就被迫去了Survivor区的“From”区，自从去了Survivor区，我就开始了我漂泊的人生，有时候在Survivor的“From”区，有时候在Survivor的“To”区，居无定所。
- 3.直到我18岁的时候，爸爸说我成人了，该去社会上闯闯了。于是我就去了年老代那边，年老代里，人很多，并且年龄都挺大的，我在这里也认识了很多。在年老代里，我生活了20年(每次GC加一岁)，然后被回收。



# 对象分配的特殊情况



# JVisualVm演示对象分配过程





# MinorGc、MajorGC、FullGC的区别

JVM在进行GC时，并非每次都对上面三个内存区域一起回收，大部分的只会针对于Eden区进行  
在JVM标准中，他里面的GC按照回收区域划分为两种：

一种是部分采集（Partial GC）：

新生代采集（Minor GC / YongGC）：

只采集新生代数据

老年代采集（Major GC / Old GC）：

只采集老年代数据，目前只有CMS会单独采集老年代

混合采集（Mixed GC）：

采集新生代与老年代部分数据，目前只有G1使用

一种是整堆采集（Full GC）：

收集整个堆与方法区的所有垃圾



# GC触发策略

年轻代触发机制：

- 🔔 当年轻代空间不足时，就会触发MinorGc,这里年轻代满值得是Eden区中满了
- 🔔 因为Java大部分对象都是具备朝生熄灭的特性，所以MinorGC非常频繁，一般回收速度也快
- 🔔 MinorGc会出发STW行为，暂停其他用户的线程

老年代GC触发机制：

- 🔔 出现MajorGC经常会伴随至少一次MinorGC(非绝对，老年代空间不足时会尝试触发MinorGC如果空间还是不足则会出发MajorGC)
- 🔔 MajorGC比MinorGC速度慢10倍，如果MajorGC后内存还是不足则会出现OOM



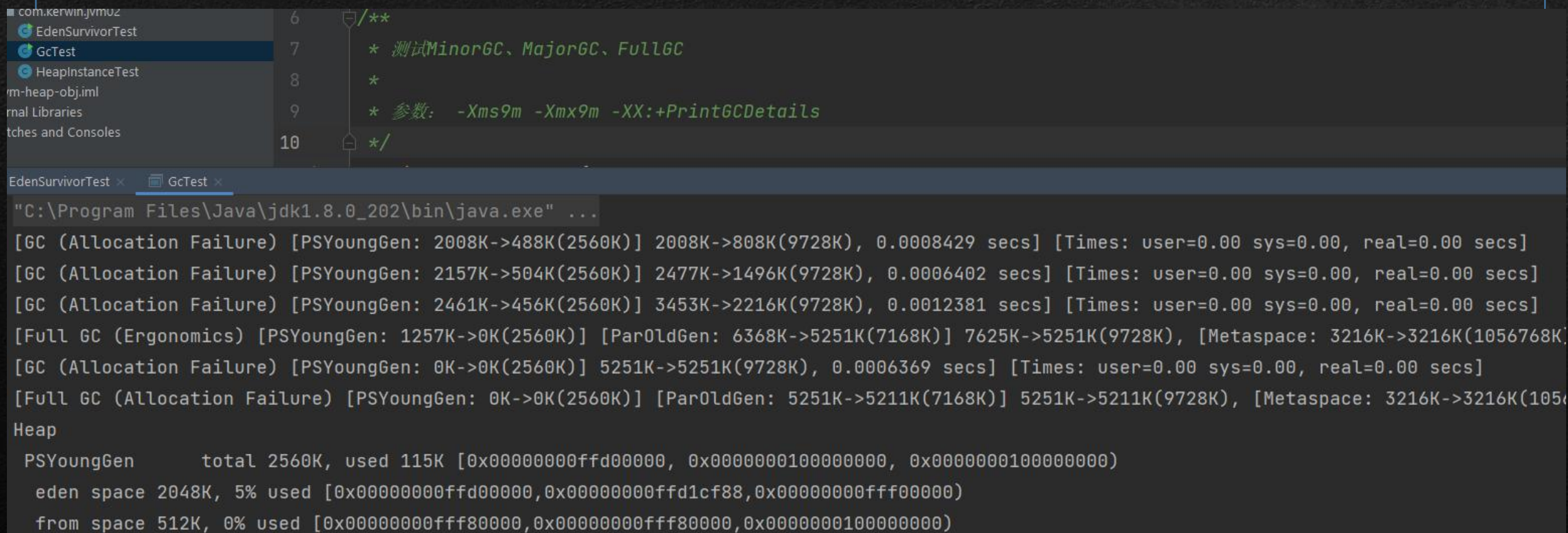
# FullGC触发

- ⌘调用System.gc()时
  - ⌘老年代空间不足时
  - ⌘方法区空间不足时
  - ⌘通过MinorGC进入老年代的平均大小大于老年代的可用内存
  - ⌘在Eden使用Survivor进行复制时，对象大小大于Survivor的可用内存，则该对象转入老年代，且老年代的可用内存小于该对象
- ⌘Full GC 是开发或者调优中尽量要避开的



# GC日志查看

-Xms9m -Xmx9m -XX:+PrintGCDetails



The screenshot shows an IDE with a project named 'com.kerwin.jvmuz'. The 'GcTest' class is selected in the project explorer. The code in 'GcTest.java' is as follows:

```
6  /**
7   * 测试MinorGC、MajorGC、FullGC
8   *
9   * 参数: -Xms9m -Xmx9m -XX:+PrintGCDetails
10  */
```

The output window shows the GC log for the 'GcTest' class. The log starts with the command 'C:\Program Files\Java\jdk1.8.0\_202\bin\java.exe' ... and displays the following GC events:

```
[GC (Allocation Failure) [PSYoungGen: 2008K->488K(2560K)] 2008K->808K(9728K), 0.0008429 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 2157K->504K(2560K)] 2477K->1496K(9728K), 0.0006402 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 2461K->456K(2560K)] 3453K->2216K(9728K), 0.0012381 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 1257K->0K(2560K)] [ParOldGen: 6368K->5251K(7168K)] 7625K->5251K(9728K), [Metaspace: 3216K->3216K(1056768K)] 0.0012381 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] 5251K->5251K(9728K), 0.0006369 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(2560K)] [ParOldGen: 5251K->5211K(7168K)] 5251K->5211K(9728K), [Metaspace: 3216K->3216K(1056768K)] 0.0012381 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

The log also shows the heap status:

```
Heap
PSYoungGen      total 2560K, used 115K [0x00000000ffd00000, 0x0000000100000000, 0x0000000100000000)
  eden space 2048K, 5% used [0x00000000ffd00000, 0x00000000ffd1cf88, 0x00000000fff00000)
  from space 512K, 0% used [0x00000000fff80000, 0x00000000fff80000, 0x0000000100000000)
```



# TLAB (Thread Local Allocation Buffer)

## 什么是TLAB?

- 🔗 堆区是线程共享区，任何线程都可以访问堆中共享数据
- 🔗 由于对象实例的创建很频繁，在并发环境下对重划分内存空间是线程不安全的，如果需要避免多个线程对于同一地址操作，需要加锁，而加锁则会影响分配速度
- 🔗 所以JVM默认在堆区中开辟了一块空间，专门服务于每一个线程。他为每个线程分配了一个私有缓存区域，包含在Eden中，这就是TLAB
- 🔗 多线程同时分配内存是，使用TLAB可以避免一系列的非线程安全问题
- 🔗 TLAB会作为内存分配的首选，TLAB总空间只会占用EDEN空间的1%
- 🔗 一旦对象在TLAB空间分配失败，JVM会尝试使用加锁来保证数据操作的原子性，从而直接在Eden中分配







02

---

## 对象逃逸与代码优化



堆是分配对象存储的唯一选择吗？



# 堆是分配对象存储的唯一选择吗？

在《深入理解JAVA虚拟机》一书中，有一段这样的描述：

随着JIT编译器的发展与**逃逸分析技术**逐渐成熟，栈上分配、标量替换优化技术将会导致一些微妙的变化，所有的对象分配到堆上也渐渐地变得不那么“绝对”了。

？ ？ ？ ？

什么是**栈上分配**？ ？    什么是**标量替换**？ ？    什么叫**逃逸分析技术**？ ？ ？



# 逃逸分析

逃逸：

- 🔥 一个对象的作用域仅限于方法区域内部在使用的情况下，此种状况加做非逃逸
- 🔥 一个对象如果被外部其他类调用，或者是作用于属性中，则此种现象被称之为对象逃逸
- 🔥 此种行为发生在字节码被编译后JIT对于代码的进一步优化



# 逃逸分析案例

```
/**
 * 未发生逃逸
 */
public void method1(){
    Point p = new Point();
    //.....
    p = null;
}
```

逃逸：

🔥 一个对象的作用域仅限于方法区域内部在使用的情况下，此种状况加做非逃逸



# 逃逸分析案例

```
/**
 * 产生逃逸
 */
public static StringBuffer method2(String s1,String s2){
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb;
}

/**
 * 未产生逃逸
 */
public static String method3(String s1,String s2){
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toString();
}
```

逃逸：

method2 因为将当前sb返回出去进行使用，所以发生逃逸，

变更为methd3之后是构建一个新的String对象，而StringBuffer对象未产生逃逸现象



# 逃逸分析：代码优化

使用逃逸分析，编译器可以堆代码做如下优化：

1. **栈上分配**：JIT编译器在编译期间根据逃逸分析计算结果，如果发现当前对象没有发生逃逸现象，那么当前对象就可能被优化成栈上分配，会将对象直接分配在栈中
2. **标量替换**：有的对象可能不需要作为一个连续的内存结构存在也能被访问到，那么对象部分可以不存储在内存，而是存储在CPU寄存器中



# 标量替换

**标量 (Scalar)**：指一个无法再分解成更小的数据的数据。Java中的原始数据类型就是标量

**聚合量 (Aggregate)**：Java中的聚合量指的是类，封装的行为就是聚合

标量替换：指的是，在未发生逃逸的情况下，函数内部生成的聚合量在经过JIT优化后会将其拆解成标量。



## 逃逸分析弊端

逃逸分析技术在99年以及发布，到JDK1.6版本后退出，但是这个技术至今还未完全成熟，原因是无法保证逃逸分析的性能消耗一定高于他的实际消耗，虽然经过逃逸分析可以做标量替换，栈上分配，锁消除等操作。但是逃逸分析自身也需要进行一系列的复杂分析算法的运算，这也是一个相对耗时过程





03

---

对象的生产与对象内存分布



# 大厂最喜欢为的几个问题

1. 对象在JVM当中的内存结构是怎样的？

2. 对象头当中有什么信息？



# 对象创建的几种实例化方案

1. new

最常见方式

2. Class.newInstance

反射

3. Cpmstructor.newInstance(xx)

反射

4. obj.clone

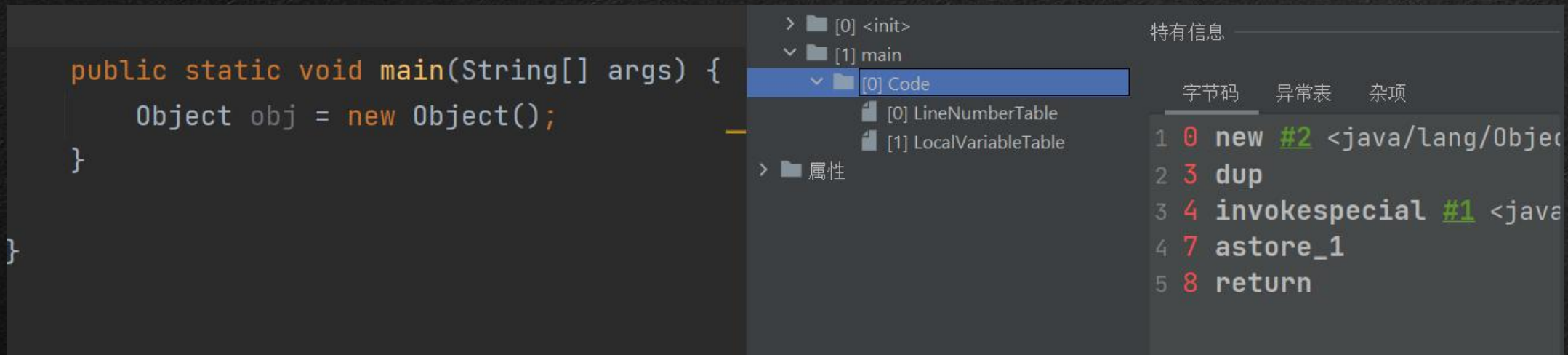
克隆数据

5. 反序列化

从文件、网络中获取一个对象流



# 字节码看对象内存创建过程



```
public static void main(String[] args) {  
    Object obj = new Object();  
}
```

特有信息

	字节码	异常表	杂项
1	0 new #2 <java/lang/Object>		
2	3 dup		
3	4 invokespecial #1 <java/lang/Object.<init>()V>		
4	7 astore_1		
5	8 return		

new = 方法调用

dup = 复制--》作用句柄

invokeSpecial = 调用构造器



# 字节码看对象创建

```
public static void main(String[] args) {  
    Object obj = new Object();  
}
```

特有关信息

	字节码	异常表	杂项
1	0 new #2 <java/lang/Object>		
2	3 dup		
3	4 invokespecial #1 <java/lang/Object.<init>()V>		
4	7 astore_1		
5	8 return		

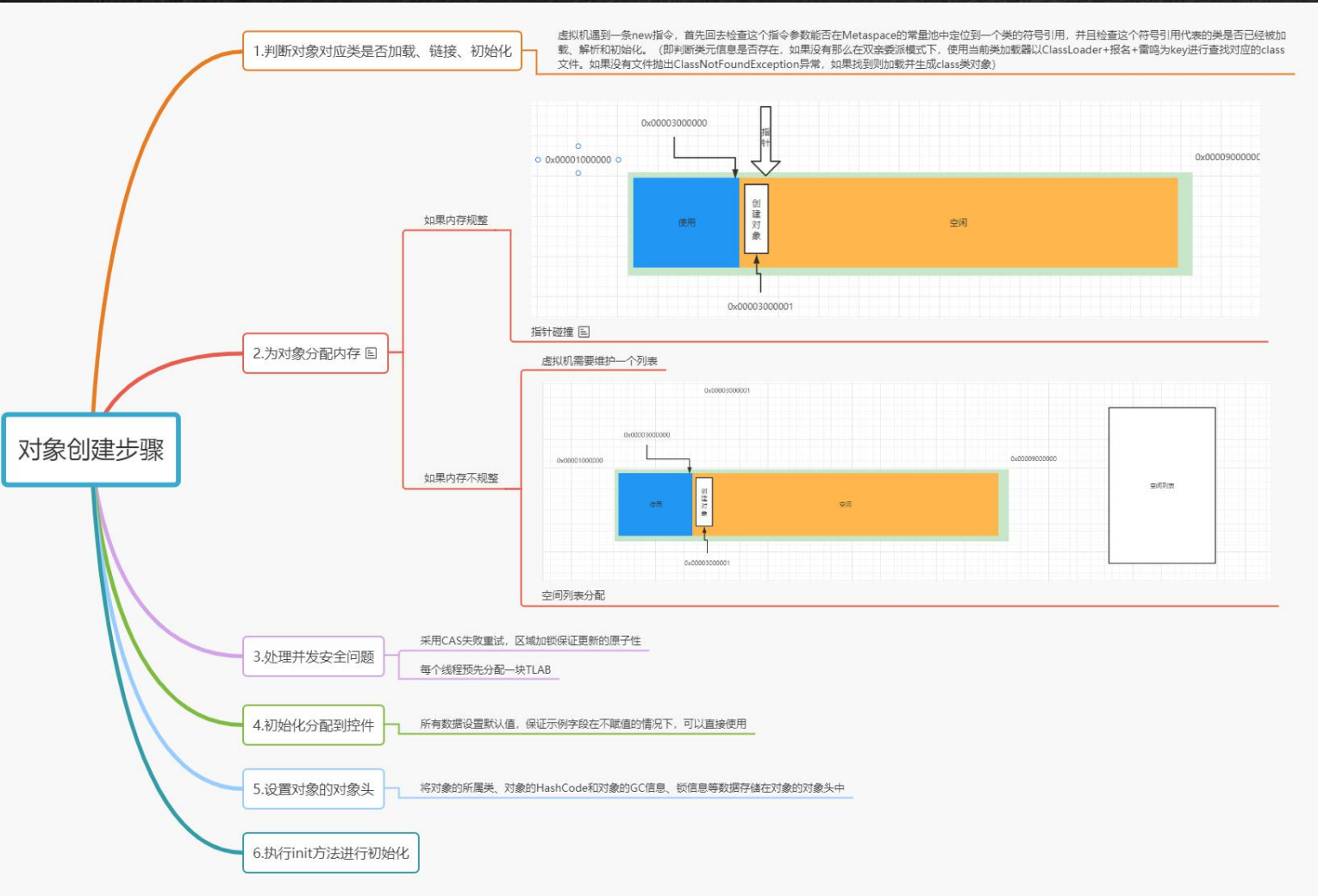
new = 方法调用

dup = 复制--》作用句柄

invokeSpecial = 调用构造器

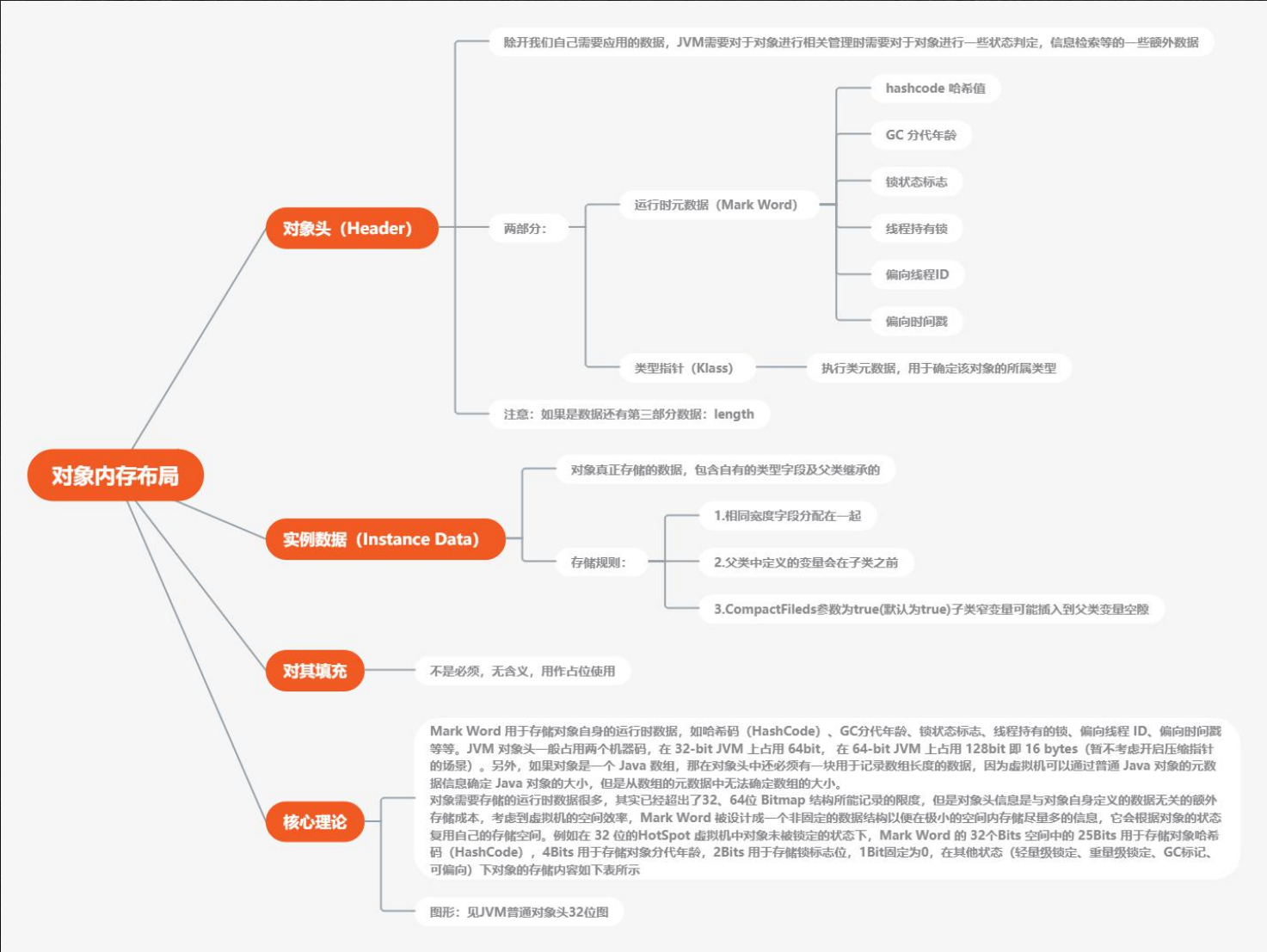


# 对象创建步骤





# 对象内存布局







# THANK YOU

码牛学院-用代码码出牛逼人生



谢谢观看

---