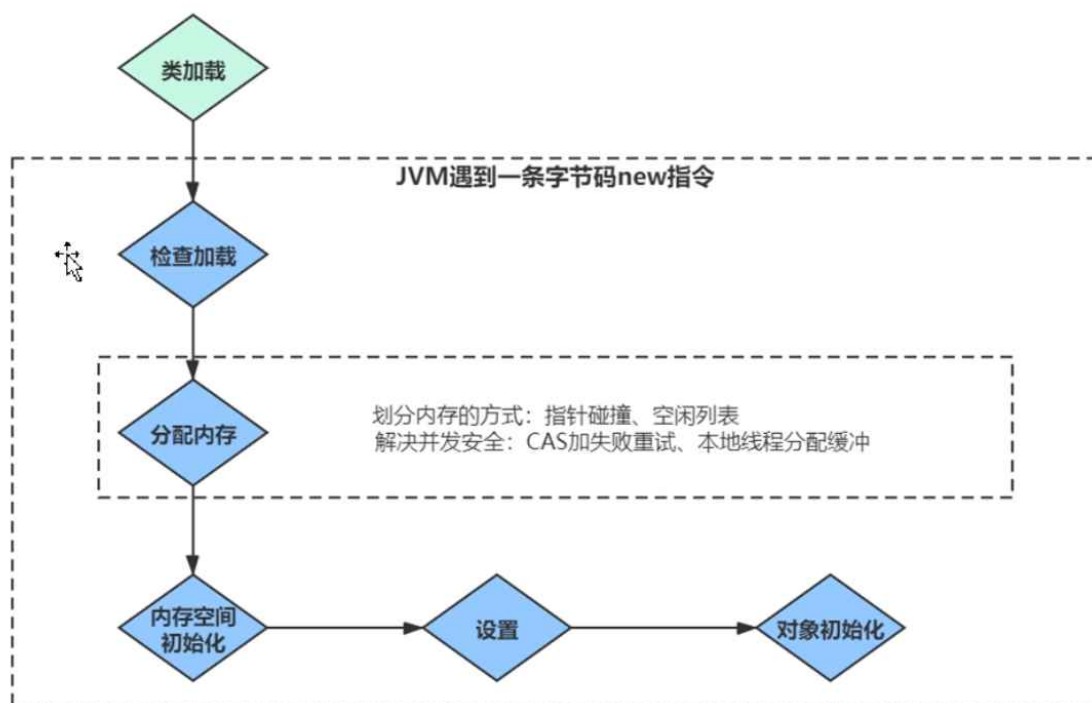


# 对象的创建过程

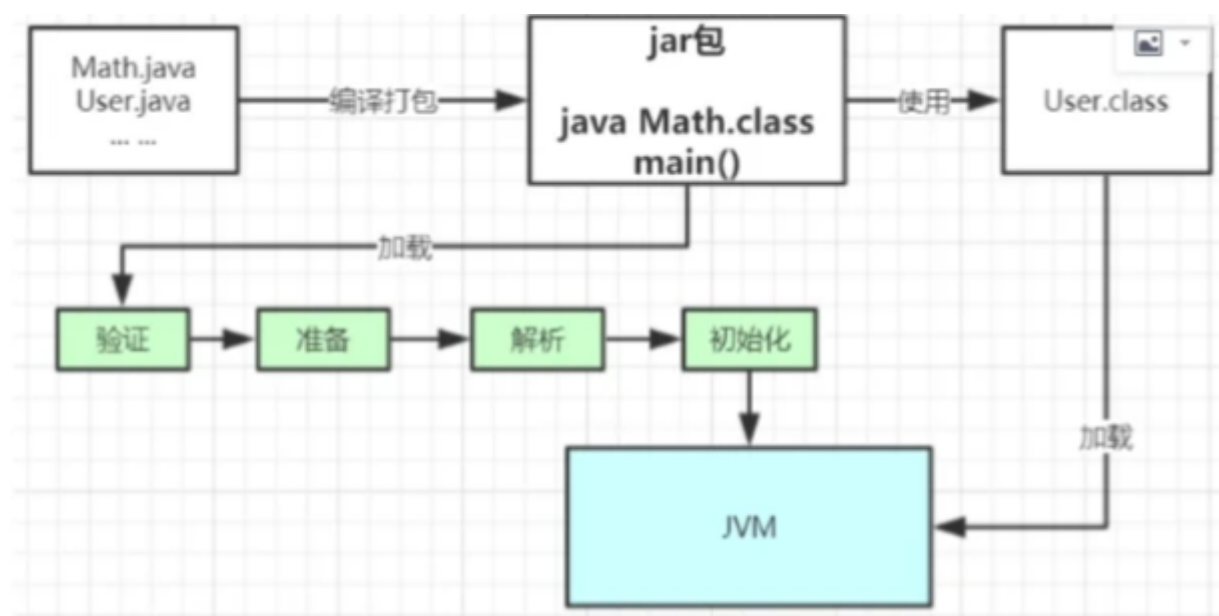


## 创建对象的流程图

- 检查加载
- 分配内存
  - ▣ 划分内存方式
  - ▣ 并发安全问题
- 初始化
  - ▣ “零” 值
- 设置
  - ▣ 对象头
- 对象初始化
  - ▣ 构造方法



## 检查类是否已经被加载





检查类是否已经被加载， 没有被加载则通过类加载子系统加载这个类

(生成对象需要依赖代码资源,而代码资源需要通过类加载进入 JVM 中才能被使用)



## 分配内存



当类被加载完之后，虚拟机就会给对象分配内存，此时对象的大小已经确定，虚拟机会从堆（栈）的内存区域中划分出该对象大小空间的区域供存放该对象。

### 分配内存方式:

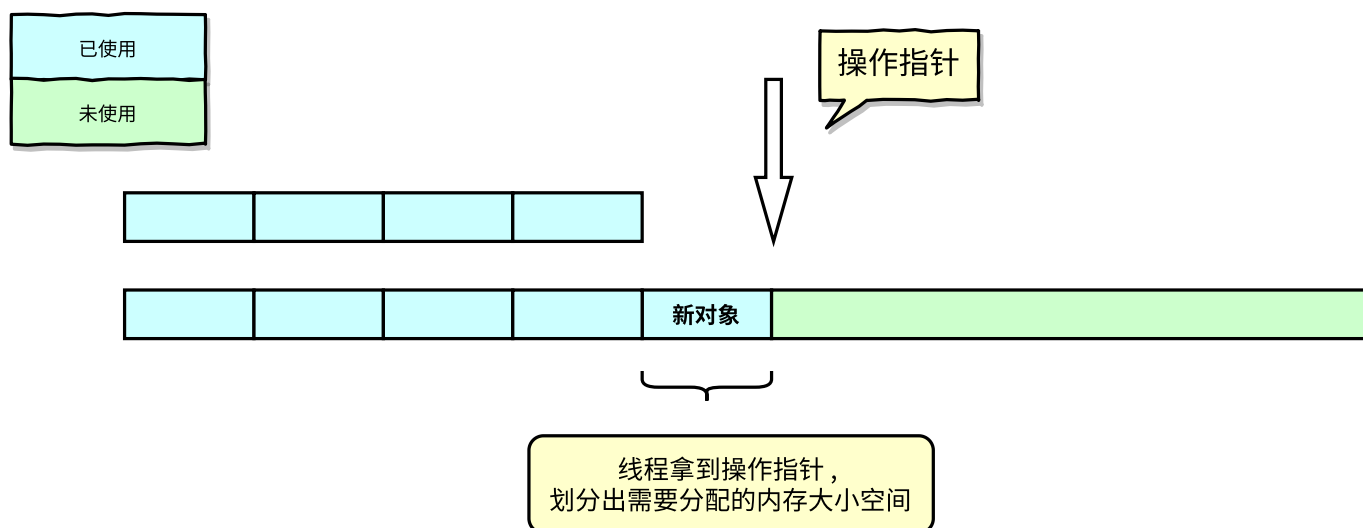
#### 1、指针碰撞



指针碰撞时jvm虚拟机默认的给对象分配内存的方式

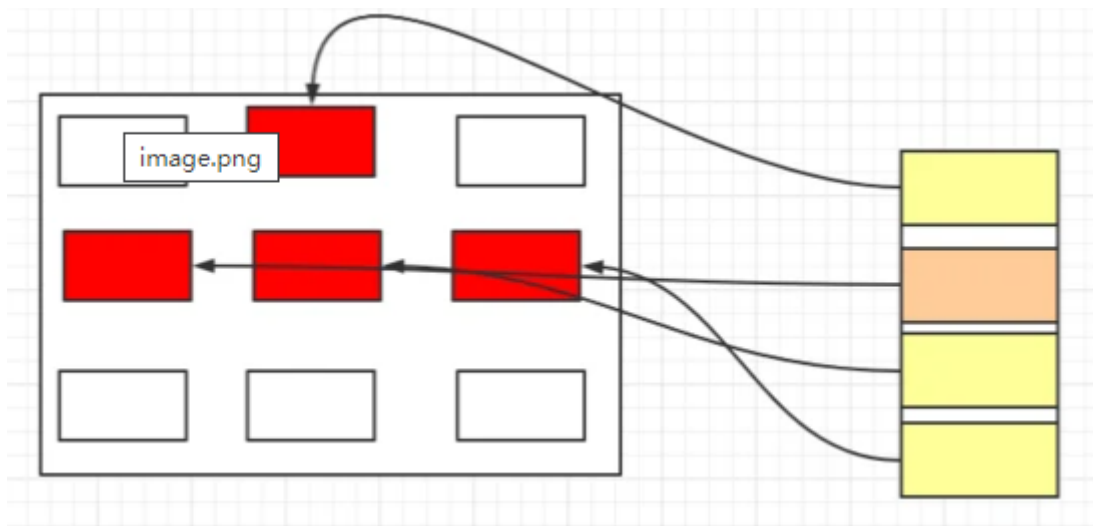
即：把堆（或栈）上的一部分内存分成两部分，一部分依次放满了对象，此时对象在这部分内存上是整齐排放着，另一部分是空白内存区域，等待放对象。

两块区域的间隔点是用一个指针进行标记，如果在这块内存中又创建了对象，则指针会向后移动这个对象的大小的地址，供这个对象存放



#### 2、空闲列表:

💡 当内存上存放的对象**不是整齐规整**存放的话，此时内存空间就会碎片化，就不能使用指针碰撞的方式去分配内存，空闲列表的方式就是维护一个可以存放对象的内存地址的**集合列表**，即记录哪些内存区域是可以存放对象的，当创建对象时，在列表中找一个空间合适的一块区域去存放，然后再更新列表。



### 3、分配的并发问题

💡 在以上分配内存的过程中，**由于堆空间是线程共享的**，如果在给对象a分配内存的同时指针还没来得及修改，对象b同时也使用了原来的这个指针来分配内存。

就会出现一块内存分配给了两个对象，**即并发分配问题**

#### 解决方案:

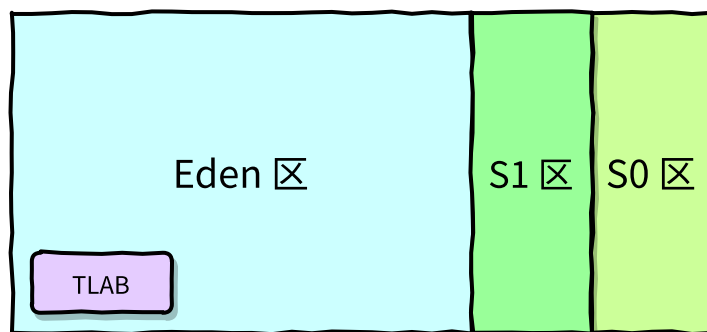
**通过CAS保证只有一个线程能分配正确**，对象在进行分配内存之前会将该线程期望的指针值与内存上的原有的指针值进行比较，如果相同，则先修改内存上的指针值，即向后移动该对象大小的空间，然后存放对象。如果不相同，则返回现在的指针值，然后重试以上操作

#### 分配优化:

**获取操作指针这个操作，可能是多线程并发的，为了保证成功会采用CAS的方式去执行，但是CAS多次失败会有性能瓶颈，会先使用TLAB的分配方式去分配内存**

1. TLAB：在内存中分配一块很小的空间作为线程私有的，直接就可以分配了，
2. 条件: 需要对象小于eden区的1%（默认），可通过配置TLABsize配置大小

使用**TLAB**可以避免一系列的非线程安全问题，同时还能够提升内存分配的吞吐量，因此我们可以讲这种内存分配方式成为**快速分配策略**



小于 Eden 区 1% 的时候通过 TLAB 分配方式给线程分配一片私有空间



## 初始化

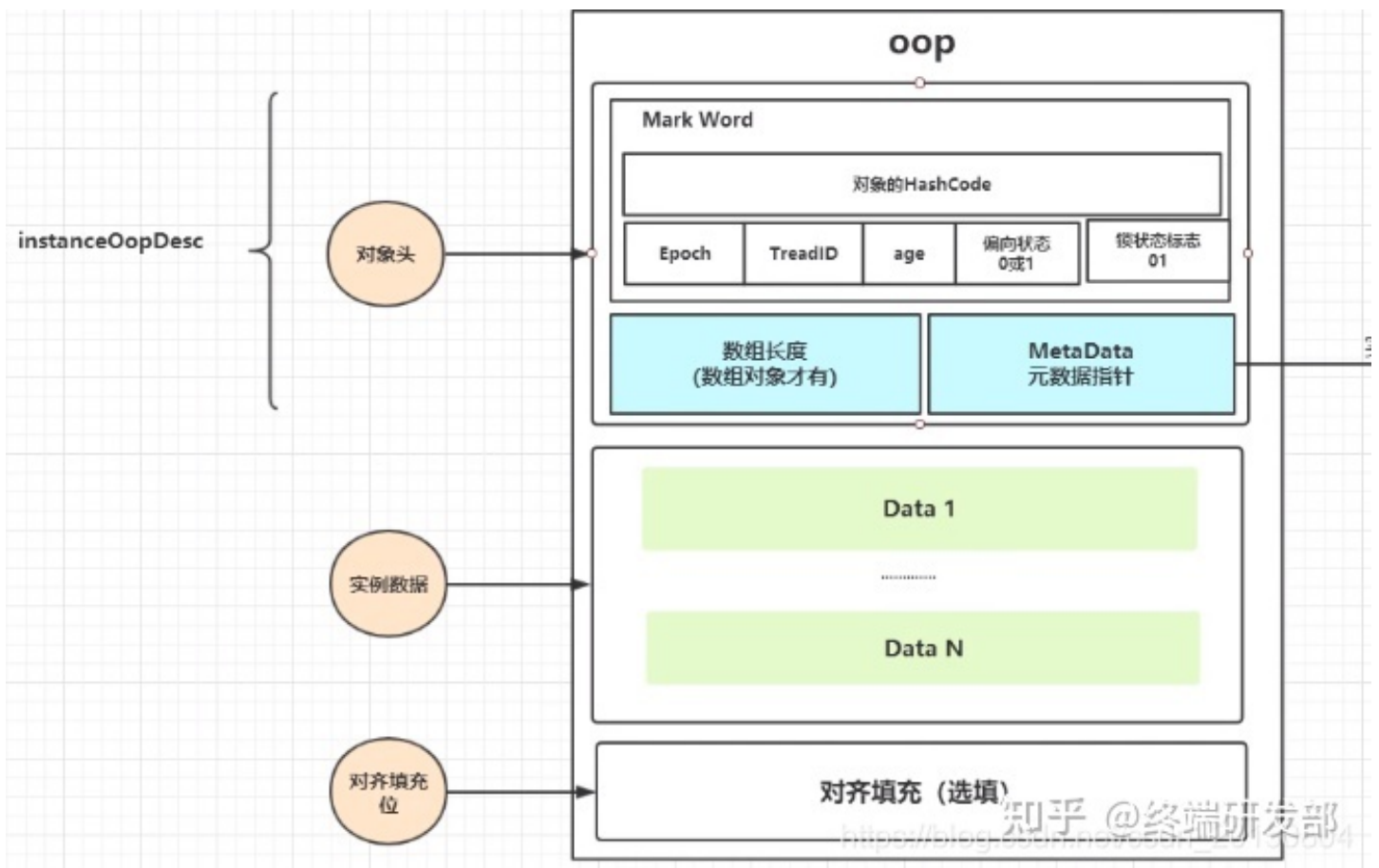


(这里的初始化不是链接里面的初始化)

给对象赋默认值。如果不赋默认值，这个对象的该字段如果不set值的话，就无法get该字段的内容，程序会出错。如果赋默认值之后，就可以返回默认值



## 设置对象头



## Java对象内存布局



- 💡 对象由对象头 (Header) 8(对象头) + 4(指针) + 4(数组) 个字节、实例数据 (Instance Data) 对齐填充 (Padding) 三部分组成
- Mark Word的32个比特存储空间中的25个比特用于存储对象的hashCode, 4个比特存储对象分代年龄, 2个比特存储锁标志位, 一个比特固定为0
1. 对象头: 包含了该对象的必要信息, 比如对象的**分代年龄**, **锁信息**, **hashCode**, **偏向锁Id**, **类型指针 (压缩4字节, 没压8字节)**, **数组长度**

2. 实例数据：对象中各个实例字段的数据

3. 对齐填充：其中为了保持对象内存的整齐性，还包括了对齐指针来进行对对象的内存大小的填充,能被8整除,为了保证数据在同一个缓存行,计算寻址最优

年龄分代最大值是15（4个比特只能存储到15）设置超过15会报错：must be between 0 and 15

根据对32位和64位的Hot Spot的对象头分析，分代年龄都占4bit。

所以在GC中，如果对象在Survivor区复制一次，年龄增加1。当对象达到设定的阈值时，将会晋升到老年代。默认情况下，并行GC的年龄阈值为15，并发GC的年龄阈值为6。由于age只有4位，所以最大值为15。

这也是 -XX:MaxTenuringThreshold 指令设置最大值为15的原因



## 执行client<> () 方法



为对象赋用户自己的值，执行构造方法,给变量赋最终值



## 对象的类元信息解析



对象在堆中创建

其对象头中存在指向该对象的类元信息的指针 Class pointer

也就是说，如果对象想要找到该对象的成员方法的具体方法内容（代码），就是通过这个对象头内的指针找的

比如： 对象.add(); ， add () 方法，需要通过这个类指针去找到对应的add方法相关信息

Class对象，也是在堆的，是提供给程序员去访问方法区类信息的入口对象

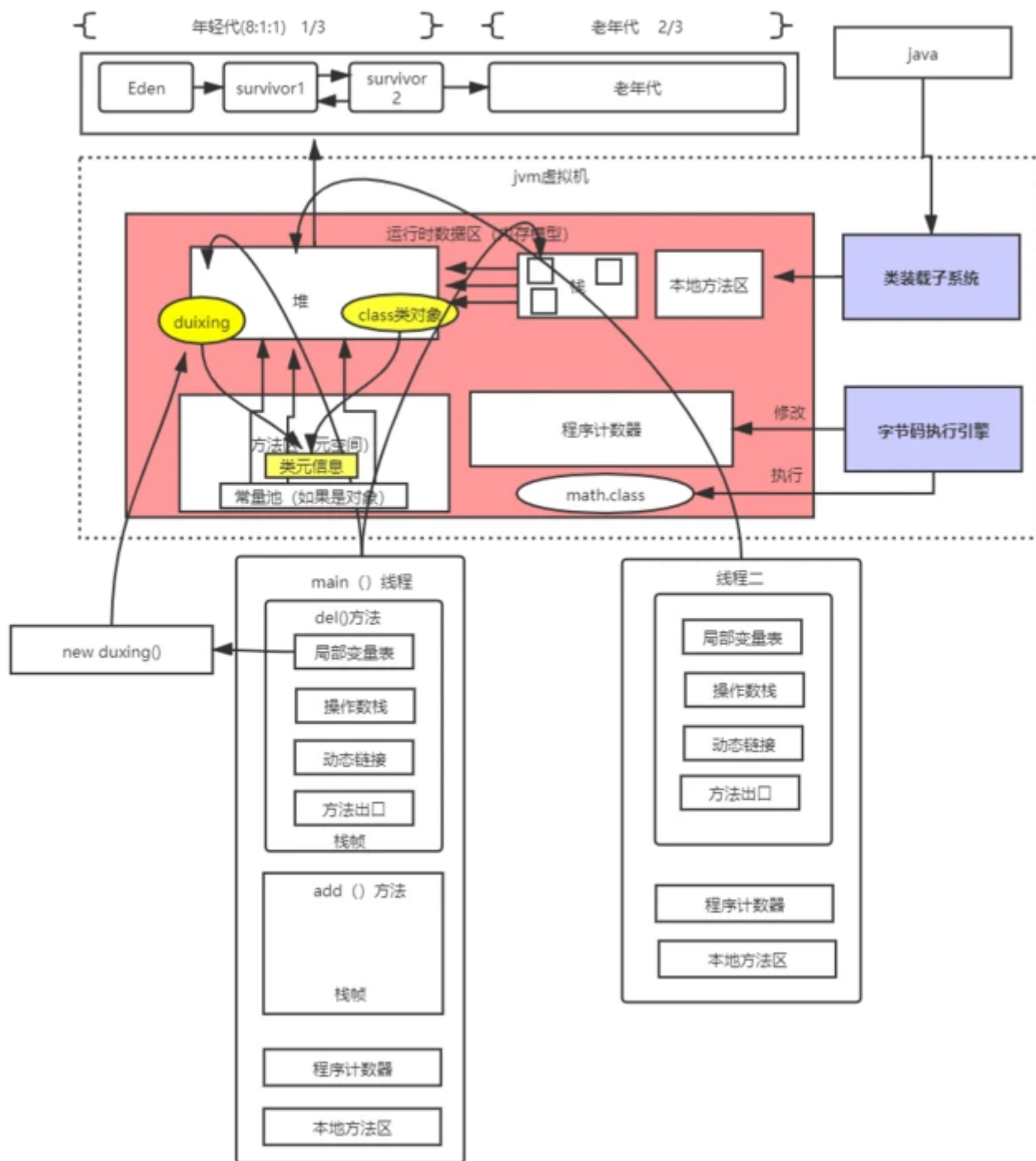
## 区分 对象、类对象、类元信息:

**对象**: 是这个类的其中一个实体,可以有多个

**类对象**: `class<?extends Duixiang> duixiangClass = 对象.getClass();`

类加载完成之后,是java虚拟机为方便用户操作该类的类元信息而创建的镜像对象,里面没有该类的信息,但是可以通过该类对象可以访问该类的信息。Class存放在堆中,类信息在方法区中

**类元信息**: 存放类的相关信息,比如变量与成员方法,在方法区中,通过 Class 或者 klass 指针访问操作



## 对象怎么定位

### 句柄池



- 句柄池映射对象和引用之间的的关系
- 优点:垃圾回收的时候只要修改句柄对应的地址,更好的统一管理,有稳定性



- 对象移动了地址，不需要重新改变指针，只需要改变句柄池
- 缺点:效率更低,需要转换关系

## 直接指针

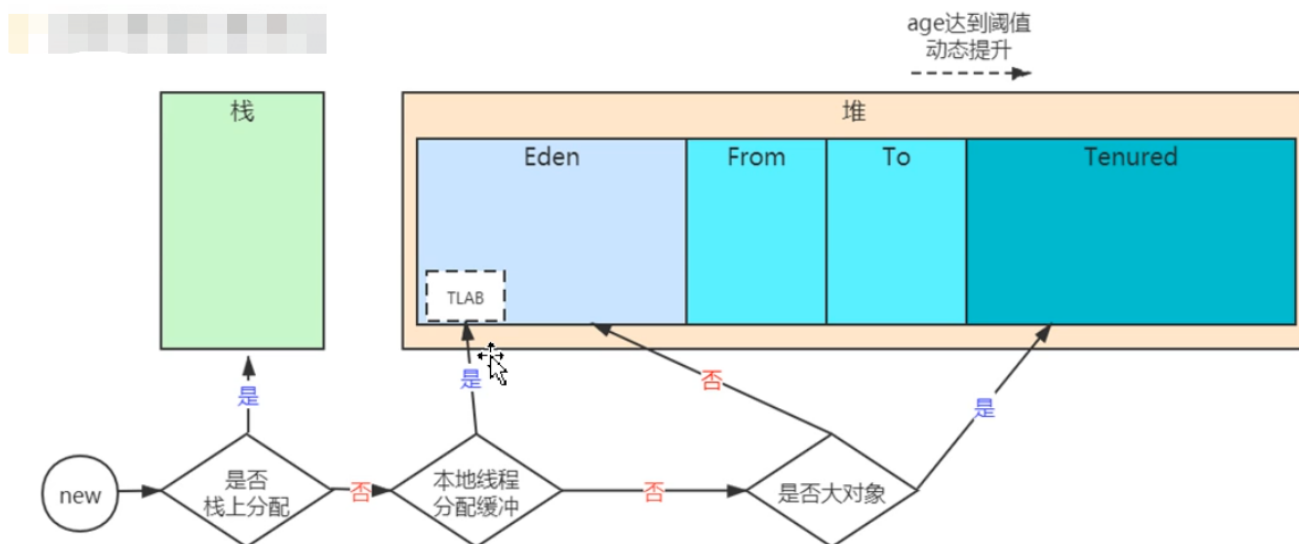
JVM的Hotspot用的是直接指针



- 优点:查询效率更快,减少了类似句柄的转换关系的开销
- 缺点:垃圾回收的时候,引用需要修改,被改了,其他在用的时候可能出问题.



## 对象的分配策略




1. 是否能在栈上分配
2. 对象优先在Eden区分配
  - 是否能够通过TLAB方式分配在 eden 区
3. 大对象直接进入老年代 ( **-XX:PretenureSizeThreshold** 配置大小, 默认是0,空间不够了才触发)
4. 长期存活的对象将进入老年代
5. 动态年龄判断( **相同年龄占单个S区一半** )
6. 空间分配担保( **利用young GC 来防止频繁FullGC** )

## 1. 对象在栈中分配原理

- 通过逃逸分析算法，判断这个对象的作用域，如果该对象只在该栈帧中调用，即方法运行完成即回收，则在栈的该线程栈的栈帧上创建，此方法运行完成，即在对应栈帧上出栈，然后立即被回收，避免了在堆上创建，然后进行gc后回收的耗时与浪费空间。
- 上边说是在栈上创建对象，其实是将该对象分解为一个个标量进行创建在栈上，并没有真的在栈上创建对象，因为对象的创建需要一块连续的空间，栈上不一定有。标量即不可被再次分解的量JAVA的基本数据类型就是标量（如：int，long等基本数据类型以及reference类型等），标量的对立就是可以被进一步分解的量，而这种量称之为聚合量。而在JAVA中对象就是可以被进一步分解的聚合量

## 2. 是否能在TLAB分配

不管是**指针碰撞**还是**空闲散列表**分配的方法,都需要先获得操作指针,获取操作指针可能是并发操作,会通过CAS去进行,**CAS的瓶颈在于多次咨询会影响CPU空消耗**,

 如果对象大小小于**1%Eden**区大小,JVM会在堆内分配一块**线程私有**的内存地址,给到线程进行**TLAB**分配

## 3. 优先在 Eden 分配

大多数情况下，对象在新生代 Eden 区分配，当 Eden 区空间不够时，发起 Minor GC

关于 Minor GC 和 Full GC：

- a. Minor GC：发生在新生代上，因为新生代对象存活时间很短，因此 Minor GC 会频繁执行，执行的速度一般 也会比较快。
- b. Full GC：发生在老年代上，老年代对象和新生代的相反，其存活时间长，因此 Full GC 很少执行，而且执行速度会比 Minor GC 慢很多

## 4. 大对象直接进入老年代

大对象是指需要连续内存空间的对象，最典型的大对象是那种很长的字符串以及数组。经常出现大对象会提前触发垃圾收集以获取足够的连续空间分配给大对象。

提供 **-XX:PretenureSizeThreshold** 参数，大于此值的对象或者是大于eden区的剩余空间直接在老年代分配，避免在 Eden 区和 Survivor 区之间的大量内存复制

## 5. 长期存活的对象进入老年代

JVM 为对象定义年龄计数器，经过 Minor GC 依然存活，并且能被 Survivor 区容纳的，移被移到 Survivor 区，年龄就增加 1 岁，增加到一定年龄则移动到老年代中（默认 15 岁，通过 -XX:MaxTenuringThreshold 设置）。

## 6. 动态对象年龄判定

JVM 并不是永远地要求对象的年龄必须达到 MaxTenuringThreshold 才能晋升老年代，如果在 Survivor 区中**相同年龄**所有对象大小的总和大于 **单个Survivor 空间的一半**，则年龄大于或等于该年龄的对象可以直接进入老年代，无需等待 MaxTenuringThreshold 中要求的年龄

**原因分析：**因为一半的对象占了其中一个 S 区，理论上存活的次数越久，而这些对象后续大概率还是存活，这样挪来挪去会浪费性能，并且如果这个时候有大量的新对象进来，空间不够用了，会把新来的对象直接放到老年代，这些新对象本来可能是马上死亡的，但是由于已经放入老年代，必须等下一次 full gc 才能把已经无用的新对象清理掉

## 7. 空间分配担保机制



当出现大量对象在一次 Minor GC 后仍然存活的情况时，Survivor 区可能容纳不下这么多对象，此时，就需要老年代进行分配担保，即将 Survivor 无法容纳的对象直接进入老年代。

这么做有一个前提，就是老年代得装得下这么多对象。可是在一次 GC 操作前，虚拟机并不知道到底会有多少对象存活，所以空间分配担保有这样一个判断流程

### 【为了避免经常full GC 该参数建议打开】

- 发生 Minor GC 前，虚拟机先检查老年代的最大可用连续空间是否大于新生代所有对象的总空间；
  - 如果大于，Minor GC 一定是安全的；
  - 如果小于，虚拟机会查看 HandlePromotionFailure 参数，看看是否允许担保失败；
    - 允许失败：尝试着进行一次 Minor GC；
    - 不允许失败：进行一次 Full GC；
- 不过 JDK 6 Update 24 后，HandlePromotionFailure 参数就没有用了，规则变为只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行 Minor GC，否则将进行 Full GC。



## new一个对象的过程

### 1: 对象的创建过程

源码:

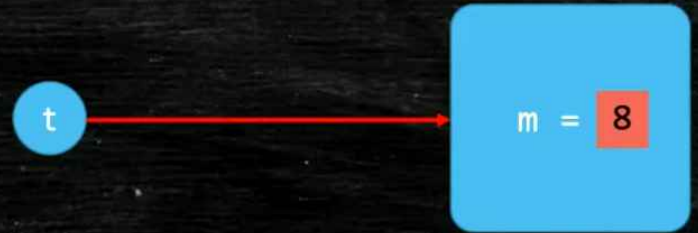
```
class T {  
    int m = 8;  
}
```

```
T t = new T();
```

汇编码:

```
0 new #2 <T>  
3 dup  
4 invokespecial #3 <T.<init>>  
7 astore_1  
8 return
```

144115200851757250正在观看直播



0.先开辟一个空间

4初始化值

7.建立关联关系

(可能发生重排序, 是可能把7和4换一个顺序的, 因为不影响最终一致性)



## 如何计算一个对象大小

```
public class MyOrder {  
    private long orderId;  
    private long userId;  
    private byte state;  
    private long createTime;  
}
```

