**引言**

　　我们知道，一般在java程序中，new的对象是分配在堆空间中的，但是实际的情况是，大部分的new对象会进入堆空间中，而并非是全部的对象，还有另外两个地方可以存储new的对象，我们称之为栈上分配以及TLAB

**栈上分配**

**为什么需要栈上分配？**

　　在我们的应用程序中，其实有很多的对象的作用域都不会逃逸出方法外，也就是说该对象的生命周期会随着方法的调用开始而开始，方法的调用结束而结束，对于这种对象，是不是该考虑将对象不在分配在堆空间中呢？

　　因为一旦分配在堆空间中，当方法调用结束，没有了引用指向该对象，该对象就需要被gc回收，而如果存在大量的这种情况，对gc来说无疑是一种负担。

**什么是栈上分配？**

　　因此，JVM提供了一种叫做栈上分配的概念，针对那些作用域不会逃逸出方法的对象，在分配内存时不在将对象分配在堆内存中，而是将对象属性打散后分配在栈（线程私有的，属于栈内存）上，这样，随着方法的调用结束，栈空间的回收就会随着将栈上分配的打散后的对象回收掉，不再给gc增加额外的无用负担，从而提升应用程序整体的性能

　　本质：Java虚拟机提供的一项优化技术

　　基本思想： 将线程私有的对象打散分配在栈上

　　优点：

　　　　1）可以在函数调用结束后自行销毁对象，不需要垃圾回收器的介入，有效避免垃圾回收带来的负面影响

　　　　2）栈上分配速度快，提高系统性能

　　局限性： 栈空间小，对于大对象无法实现栈上分配

　　技术基础： 逃逸分析

　　逃逸分析的目的： 判断对象的作用域是否超出函数体[即:判断是否逃逸出函数体

逃逸分析例子：

[复制代码](javascript:void(0);)

//user的作用域超出了函数setUser的范围,是逃逸对象

//当函数结束调用时，不会自行销毁user

private User user;

public void setUser(){

user = new User();

user.setId(1);

user.setName("blueStarWei");

}

//u只在函数内部生效，不是逃逸对象

//当函数调用结束，会自行销毁对象u

public void createUser(){

User u = new User();

u.setId(2);

u.setName("JVM");

}

[复制代码](javascript:void(0);)

栈上分配示例：

[复制代码](javascript:void(0);)

public class AllotOnStack {

public static void main(String[] args) {

long start = System.currentTimeMillis();

for (int i = 0; i < 100000000; i++) {

alloc();

}

long end = System.currentTimeMillis();

System.out.println(end - start);

}

private static void alloc() {

User user = new User();

user.setId(1);

user.setName("blueStarWei");

}

}

[复制代码](javascript:void(0);)

上述代码调用了1亿次alloc()，如果是分配到堆上，大概需要1.5GB的堆空间，如果堆空间小于该值，必然会触发GC。

使用如下参数运行，发现不会触发GC

// 使用server模式 最大堆空间为15m 初始堆空间为15m 启用逃逸分析 打印ＧＣ日志 关闭TLAB 启用标量替换，允许对象打散分配到栈上  
-server -Xmx15m -Xms15m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:-UseTLAB -XX:+EliminateAllocations

使用如下参数（任意一行）运行，会发现触大量ＧＣ

//关闭逃逸分析 关闭TLAB 启用标量替换，允许对象打散分配到栈上

-server -Xmx15m -Xms15m -XX:－DoEscapeAnalysis -XX:+PrintGC -XX:-UseTLAB -XX:+EliminateAllocations

//启用逃逸分析 关闭TLAB 关闭标量替换

-server -Xmx15m -Xms15m -XX:＋DoEscapeAnalysis -XX:+PrintGC -XX:-UseTLAB -XX:－EliminateAllocations

可以发现：栈上分配依赖于逃逸分析和标量替换

GC日志

[GC (Allocation Failure) 4095K->528K(15872K), 0.0025208 secs]

[GC (Allocation Failure) 4624K->552K(15872K), 0.0012518 secs]

[GC (Allocation Failure) 4648K->608K(15872K), 0.0009262 secs]

......(省略)

3718

GC日志解析

[A screenshot of a computer

Description automatically generated](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513183518908-2010958762.png)

JVM参数解析

[A screenshot of a computer

Description automatically generated](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513183542060-795704865.png)

TLAB 分配

　　JVM在内存新生代Eden Space中开辟了一小块线程私有的区域，称作TLAB（Thread-local allocation buffer）。默认设定为占用Eden Space的1%。在Java程序中很多对象都是小对象且用过即丢，它们不存在线程共享也适合被快速GC，所以对于小对象通常JVM会优先分配在TLAB上，并且TLAB上的分配由于是线程私有所以没有锁开销。因此在实践中分配多个小对象的效率通常比分配一个大对象的效率要高。  
　　也就是说，Java中每个线程都会有自己的缓冲区称作TLAB（Thread-local allocation buffer），每个TLAB都只有一个线程可以操作，TLAB结合bump-the-pointer技术可以实现快速的对象分配，而不需要任何的锁进行同步，也就是说，在对象分配的时候不用锁住整个堆，而只需要在自己的缓冲区分配即可。  
　　关于对象分配的JDK源码可以参见 [JVM 之 Java对象创建[初始化]](http://blog.hesey.net/2011/07/object-allocation-on-non-heap.html) 中对OpenJDK源码的分析。

为什么需要TLAB？

　　这是为了加速对象的分配。由于对象一般分配在堆上，而堆是线程共用的，因此可能会有多个线程在堆上申请空间，而每一次的对象分配都必须线程同步，会使分配的效率下降。考虑到对象分配几乎是Java中最常用的操作，因此JVM使用了TLAB这样的线程专有区域来避免多线程冲突，提高对象分配的效率。

　　局限性： TLAB空间一般不会太大（占用eden区），所以大对象无法进行TLAB分配，只能直接分配到堆上。

　　分配策略：

　　　　一个100KB的TLAB区域，如果已经使用了80KB，当需要分配一个30KB的对象时，TLAB是如何分配的呢？

　  　　 此时，虚拟机有两种选择：第一，废弃当前的TLAB（会浪费20KB的空3.4 间）；第二，将这个30KB的对象直接分配到堆上，保留当前TLAB（当有小于20KB的对象请求TLAB分配时可以直接使用该TLAB区域）。

　　　　JVM选择的策略是：在虚拟机内部维护一个叫refill\_waste的值，当请求对象大于refill\_waste时，会选择在堆中分配，反之，则会废弃当前TLAB，新建TLAB来分配新对象。

　　　　【默认情况下，TLAB和refill\_waste都是会在运行时不断调整的，使系统的运行状态达到最优。】

JVM参数解析

[](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513184519103-1205351241.png)

**栈上分配和TLAB对比**

[A screenshot of a computer

Description automatically generated](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513185143309-1578594042.png)

**Java对象分配的过程**

1. 编译器通过逃逸分析，确定对象是在栈上分配还是在堆上分配。如果是在堆上分配，则进入选项2.
2. 如果tlab\_top + size <= tlab\_end，则在在TLAB上直接分配对象并增加tlab\_top 的值，如果现有的TLAB不足以存放当前对象则3.
3. 重新申请一个TLAB，并再次尝试存放当前对象。如果放不下，则4.
4. 在Eden区加锁（这个区是多线程共享的），如果eden\_top + size <= eden\_end则将对象存放在Eden区，增加eden\_top 的值，如果Eden区不足以存放，则5.
5. 执行一次Young GC（minor collection）。
6. 经过Young GC之后，如果Eden区任然不足以存放当前对象，则直接分配到老年代。

　　对象不在堆上分配主要的原因还是堆是共享的，在堆上分配有锁的开销。无论是TLAB还是栈都是线程私有的，私有即避免了竞争（当然也可能产生额外的问题例如可见性问题），这是典型的用空间换效率的做法。

对象内存分配的两种方法  
　　为对象分配空间的任务等同于把一块确定大小的内存从Java堆中划分出来。

指针碰撞(Serial、ParNew等带Compact过程的收集器)

　　假设Java堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“指针碰撞”（Bump the Pointer）。

空闲列表(CMS这种基于Mark-Sweep算法的收集器)   
　　如果Java堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”（Free List）。 

总结

总体流程

[A diagram of a flowchart

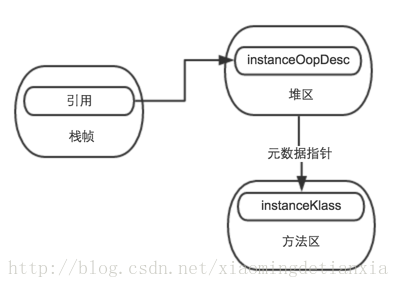
Description automatically generated](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513184933035-187311514.png)

对象分配流程   
 [A diagram of a flowchart

Description automatically generated](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513184946035-1879109821.png)

　　如果开启栈上分配，JVM会先进行栈上分配，如果没有开启栈上分配或则不符合条件的则会进行TLAB分配，如果TLAB分配不成功，再尝试在eden区分配，如果对象满足了直接进入老年代的条件，那就直接分配在老年代。

对象在内存的引用方式

[](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513185001186-869262329.png)

对象在内存中的结构

[A diagram of a computer

Description automatically generated](https://img2020.cnblogs.com/blog/885859/202005/885859-20200513185005668-954611212.png)