1、说下JVM中类加载器分类与核心功能

Java里有如下几种类加载器

- 引导类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar等
- 扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包
- 应用程序类加载器：负责加载ClassPath路径下的类包，主要就是加载你自己写的那些类
- 自定义加载器：负责加载用户自定义路径下的类包

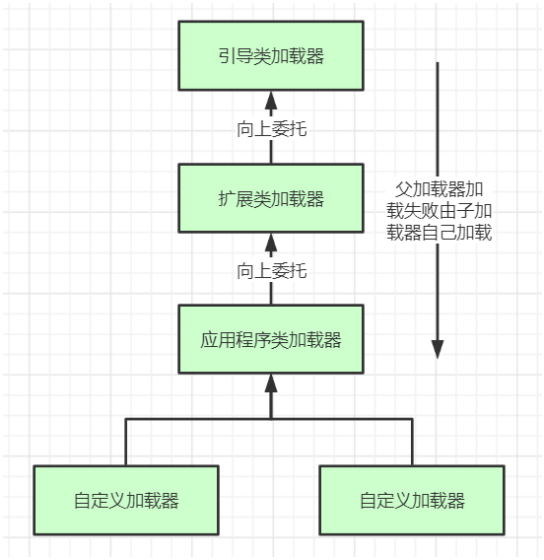
看一个**类加载器**示例：

```
public class TestJDKClassLoader {

 public static void main(String[] args) {
 System.out.println(String.class.getClassLoader());
 System.out.println(com.sun.crypto.provider.DESKeyFactory.class.getClassLoader().getClass().getName());
 System.out.println(TestJDKClassLoader.class.getClassLoader().getClass().getName());

 System.out.println();
 ClassLoader appClassLoader = ClassLoader.getSystemClassLoader();
  ClassLoader extClassloader = appClassLoader.getParent();
  ClassLoader bootstrapLoader = extClassloader.getParent();
  System.out.println("the bootstrapLoader : " + bootstrapLoader);
  System.out.println("the extClassloader : " + extClassloader);
  System.out.println("the appClassLoader : " + appClassLoader);

  System.out.println();
  System.out.println("bootstrapLoader加载以下文件：");
  URL[] urls = Launcher.getBootstrapClassPath().getURLs();
  for (int i = 0; i < urls.length; i++) {
  System.out.println(urls[i]);
  }

  System.out.println();
  System.out.println("extClassloader加载以下文件：");
  System.out.println(System.getProperty("java.ext.dirs"));

  System.out.println();
  System.out.println("appClassLoader加载以下文件：");
  System.out.println(System.getProperty("java.class.path"));

  }
}

运行结果：
null
sun.misc.Launcher$ExtClassLoader
sun.misc.Launcher$AppClassLoader

the bootstrapLoader : null
the extClassloader : sun.misc.Launcher$ExtClassLoader@3764951d
the appClassLoader : sun.misc.Launcher$AppClassLoader@14dad5dc

bootstrapLoader加载以下文件：
file:/D:/dev/Java/jdk1.8.0_45/jre/lib/resources.jar
file:/D:/dev/Java/jdk1.8.0_45/jre/lib/rt.jar
file:/D:/dev/Java/jdk1.8.0_45/jre/lib/sunrsasign.jar
file:/D:/dev/Java/jdk1.8.0_45/jre/lib/jsse.jar
file:/D:/dev/Java/jdk1.8.0_45/jre/lib/jce.jar
file:/D:/dev/Java/jdk1.8.0_45/jre/lib/charsets.jar
file:/D:/dev/Java/jdk1.8.0_45/jre/lib/jfr.jar
file:/D:/dev/Java/jdk1.8.0_45/jre/classes
```

```
52
53 extClassloader加载以下文件：
54 D:\dev\Java\jdk1.8.0_45\jre\lib\ext;C:\Windows\Sun\Java\lib\ext
55
56 appClassLoader加载以下文件：
57 D:\dev\Java\jdk1.8.0_45\jre\lib\charsets.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\deploy.jar;D:\dev\Java\jdk1.8.0_45\jre
\lib\ext\access-bridge-64.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\cldrdata.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\dns
ns.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\jaccess.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\jfxrt.jar;D:\dev\Java\jdk1.8
0_45\jre\lib\ext\localedata.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\nashorn.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\su
nec.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\sunjce_provider.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\sunmscapi.jar;D:\d
ev\Java\jdk1.8.0_45\jre\lib\ext\sunpkcs11.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\ext\zipfs.jar;D:\dev\Java\jdk1.8.0_45\jr
e\lib\javaws.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\jce.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\jfr.jar;D:\dev\Java\jdk1.8.0_
45\jre\lib\jfxswt.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\jsse.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\management-
agent.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\plugin.jar;D:\dev\Java\jdk1.8.0_45\jre\lib\resources.jar;D:\dev\Java\jdk1.8.
0_45\jre\lib\rt.jar;D:\ideaProjects\project-all\target\classes;C:\Users\zhuge\.m2\repository\org\apache\zookeeper\zook
eeper\3.4.12\zookeeper-3.4.12.jar;C:\Users\zhuge\.m2\repository\org\slf4j\slf4j-api\1.7.25\slf4j-api-1.7.25.jar;C:\Use
rs\zhuge\.m2\repository\org\slf4j\slf4j-log4j12\1.7.25\slf4j-log4j12-1.7.25.jar;C:\Users\zhuge\.m2\repository\log4j\lo
g4j\1.2.17\log4j-1.2.17.jar;C:\Users\zhuge\.m2\repository\jline\jline\0.9.94\jline-0.9.94.jar;C:\Users\zhuge\.m2\repos
itory\org\apache\yetus\audience-annotations\0.5.0\audience-annotations-0.5.0.jar;C:\Users\zhuge\.m2\repository\io\nett
y\netty\3.10.6.Final\netty-3.10.6.Final.jar;C:\Users\zhuge\.m2\repository\com\google\guava\guava\22.0\guava-
22.0.jar;C:\Users\zhuge\.m2\repository\com\google\code\findbugs\jsr305\1.3.9\jsr305-1.3.9.jar;C:\Users\zhuge\.m2\repos
itory\com\google\errorprone\error_prone_annotations\2.0.18\error_prone_annotations-2.0.18.jar;C:\Users\zhuge\.m2\repos
itory\com\google\j2objc\j2objc-annotations\1.1\j2objc-annotations-1.1.jar;C:\Users\zhuge\.m2\repository\org\codehaus\m
ojo\animal-sniffer-annotations\1.14\animal-sniffer-annotations-1.14.jar;D:\dev\IntelliJ IDEA 2018.3.2\lib\idea_rt.jar
58
```

2、类加载双亲委派机制是怎么回事

JVM类加载器是有亲子层级结构的，如下图



双亲委派机制说简单点就是，先找父亲加载加载，不行再由儿子加载器自己加载

比如我们自己写的一个类，最先会找应用程序类加载器加载，应用程序类加载器会先委托扩展类加载器加载，扩展类加载器再委托引导类加载器，顶层引导类加载器在自己的类加载路径里找了半天没找到你自己写的类，则向下退回加载类的请求，扩展类加载器收到回复就自己加载，在自己的类加载路径里找了半天也没找到，又向下退回类的加载请求给应用程序类加载器，应用程序类加载器于是在自己的类加载路径里找你自己写的类，结果找到了就自己加载了。

为什么会有双亲委派机制?

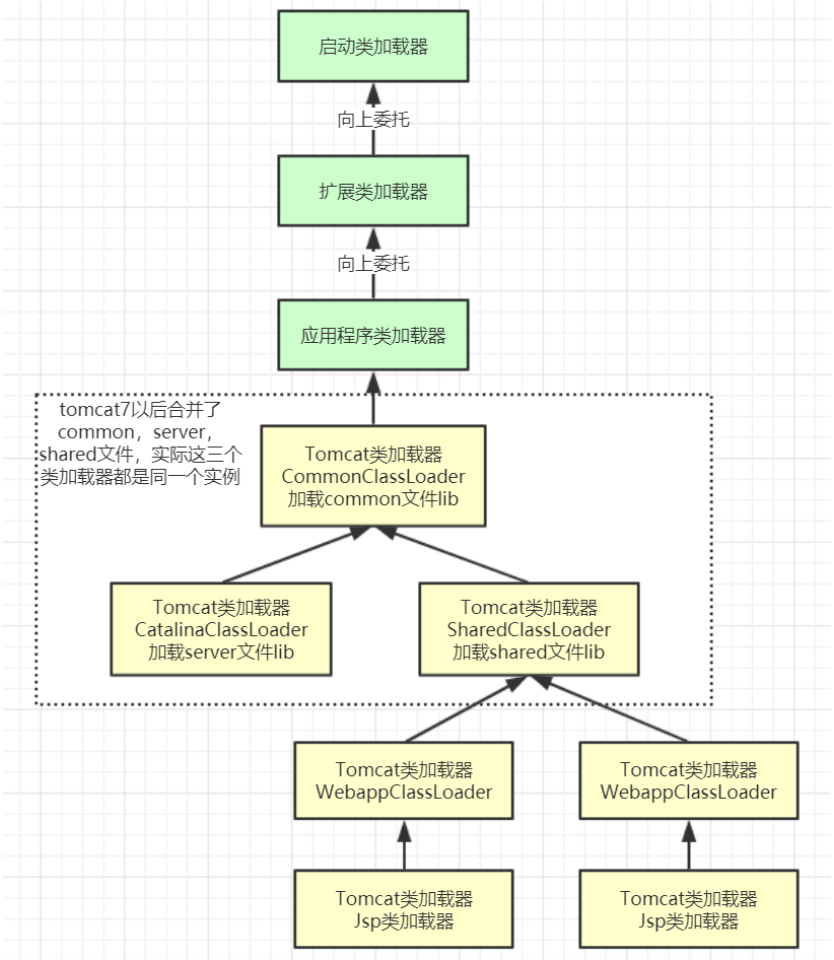
- 沙箱安全机制：自己写的java.lang.String.class类不会被加载，这样便可以防止核心API库被随意篡改
- 避免类的重复加载：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次，保证**被加载类的唯一性**

3、Tomcat底层类加载是用的双亲委派机制吗

不是！Tomcat底层类加载打破了双亲委派机制！

一个Tomcat Web容器可能需要部署两个应用程序，不同的应用程序可能会**依赖同一个第三方类库的不同版本**，不能要求同一个类库在同一个服务器只有一份，因此要保证每个应用程序的类库都是独立的，保证相互隔离。

Tomcat自定义加载器



tomcat的几个主要类加载器：

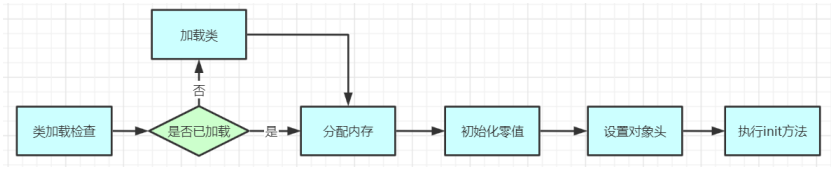
- commonLoader：Tomcat最基本的类加载器，加载路径中的class可以被Tomcat容器本身以及各个Webapp访问；
- catalinaLoader：Tomcat容器私有的类加载器，加载路径中的class对于Webapp不可见；
- sharedLoader：各个Webapp共享的类加载器，加载路径中的class对于所有Webapp可见，但是对于Tomcat容器不可见；
- WebappClassLoader：各个Webapp私有的类加载器，加载路径中的class只对当前Webapp可见，比如加载war包里相关的类，每个war包应用都有自己的WebappClassLoader，实现相互隔离，比如不同war包应用引入了不同的spring版本，这样实现就能加载各自的spring版本；

从图中的委派关系中可以看出：
CommonClassLoader能加载的类都可以被CatalinaClassLoader和SharedClassLoader使用，从而实现了公有类库的共用，而CatalinaClassLoader和SharedClassLoader自己能加载的类则与对方相互隔离。
WebAppClassLoader可以使用SharedClassLoader加载到的类，但各个WebAppClassLoader实例之间相互隔离。
而JasperLoader的加载范围仅仅是这个JSP文件所编译出来的那一个.Class文件，它出现的目的就是为了被丢弃：当Web容器检测到JSP文件被修改时，会替换掉目前的JasperLoader的实例，并通过再建立一个新的Jsp类加载器来实现JSP文件的热加载功能。

tomcat 这种类加载机制违背了java 推荐的双亲委派模型了吗？答案是：违背了。
很显然，tomcat 不是这样实现，tomcat 为了实现隔离性，没有遵守这个约定，**每个webappClassLoader加载自己的目录下的class文件，不会传递给父类加载器，打破了双亲委派机制。**

4、说下对象完整创建流程

对象创建的主要流程:



1.类加载检查

虚拟机遇到一条new指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

new指令对应到语言层面上讲是，new关键词、对象克隆、对象序列化等。

2.分配内存

在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需内存的大小在类加载完成后便可完全确定，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。

3.初始化零值

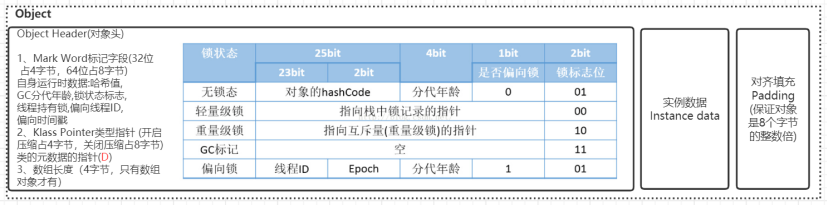
内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头）。

4.设置对象头

初始化零值之后，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的GC分代年龄等信息。这些信息存放在对象的对象头Object Header之中。

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头（Header）、 实例数据（Instance Data）和对齐填充（Padding）。 HotSpot虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的运行时数据， 如哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时 间戳等。对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

32位对象头



64位对象头

5.执行<init>方法

执行<init>方法，即对象按照程序员的意愿进行初始化。对应到语言层面上讲，就是为属性赋值（注意，这与上面的赋零值不同，这是由程序员赋的值），和执行构造方法。

5、对象分配内存时的指针碰撞与空闲列表机制知道吗

- 指针碰撞（Bump the Pointer）(默认用指针碰撞)

如果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。

- 空闲列表（Free List）

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。

6、对象分配内存时的并发问题解决CAS与TLAB机制知道吗

在并发情况下， 可能出现正在给对象A分配内存，指针还没来得及修改，对象B又同时使用了原来的指针来分配内存的情况。

解决并发问题的方法：

- CAS（compare and swap）

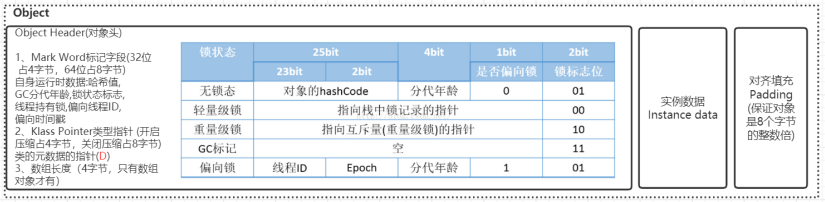
虚拟机采用CAS配上失败重试的方式保证更新操作的原子性来对分配内存空间的动作进行同步处理。

- 本地线程分配缓冲（Thread Local Allocation Buffer,TLAB）

把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在Java堆中预先分配一小块内存。通过-XX:+/-UseTLAB参数来设定虚拟机是否使用TLAB(JVM会默认开启-XX:+UseTLAB)，-XX:TLABSize 指定TLAB大小。

7、说下对象头是怎么回事

32位对象头



64位对象头

64位虚拟机

锁状态	56bit		1bit	4bit	1bit (是否偏向锁)	2bit (锁标志位)
	25bit	31bit				
无锁	unused	对象 hashCode	Cms_free	对象分代年龄	0	01
偏向锁	threadId(54bit)(偏向锁的线程ID)	Epoch(2bit)	Cms_free	对象分代年龄	1	01
轻量级锁	指向栈中锁的记录的指针					00
重量级锁	指向重量级锁的指针					10
GC 标志	空					11

8、如何计算对象占用内存大小

对象大小可以用jol-core包查看，引入依赖

```
<dependency>
 <groupId>org.openjdk.jol</groupId>
 <artifactId>jol-core</artifactId>
 <version>0.9</version>
</dependency>
```

```
import org.openjdk.jol.info.ClassLayout;

/**
 * 计算对象大小
 */
public class JOLSample {

 public static void main(String[] args) {
 ClassLayout layout = ClassLayout.parseInstance(new Object());
  System.out.println(layout.toPrintable());

  System.out.println();
  ClassLayout layout1 = ClassLayout.parseInstance(new int[]{});
  System.out.println(layout1.toPrintable());

  System.out.println();
  ClassLayout layout2 = ClassLayout.parseInstance(new A());
  System.out.println(layout2.toPrintable());
  }

  // -XX:+UseCompressedOops 默认开启的压缩所有指针
  // -XX:+UseCompressedClassPointers 默认开启的压缩对象头里的类型指针Klass Pointer
  // Oops : Ordinary Object Pointers
  public static class A {
  //8B mark word
  //4B Klass Pointer 如果关闭压缩-XX:-UseCompressedClassPointers或-XX:-UseCompressedOops，则占用8B
  int id; //4B
  String name; //4B 如果关闭压缩-XX:-UseCompressedOops，则占用8B
  byte b; //1B
  Object o; //4B 如果关闭压缩-XX:-UseCompressedOops，则占用8B
  }
```

```
}


运行结果:
java.lang.Object object internals:
 OFFSET SIZE TYPE DESCRIPTION VALUE
 0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000) (1) //mark word
 4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000) (0) //mark word
 8 4 (object header) e5 01 00 f8 (11100101 00000001 00000000 11111000) (-134217243) //Klass Pointer
 12 4 (loss due to the next object alignment)
Instance size: 16 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total


[I object internals:
 OFFSET SIZE TYPE DESCRIPTION VALUE
 0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000) (1)
 4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000) (0)
 8 4 (object header) 6d 01 00 f8 (01101101 00000001 00000000 11111000) (-134217363)
 12 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000) (0)
 16 0 int [I.<elements> N/A
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total


com.tuling.jvm.JOLSample$A object internals:
 OFFSET SIZE TYPE DESCRIPTION VALUE
 0 4 (object header) 01 00 00 00 (00000001 00000000 00000000 00000000) (1)
 4 4 (object header) 00 00 00 00 (00000000 00000000 00000000 00000000) (0)
 8 4 (object header) 61 cc 00 f8 (01100001 11001100 00000000 11111000) (-134165407)
 12 4 int A.id 0
 16 1 byte A.b 0
 17 3 (alignment/padding gap)
 20 4 java.lang.String A.name null
 24 4 java.lang.Object A.o null
 28 4 (loss due to the next object alignment)
Instance size: 32 bytes
Space losses: 3 bytes internal + 4 bytes external = 7 bytes total
```

9、对象指针压缩是怎么回事

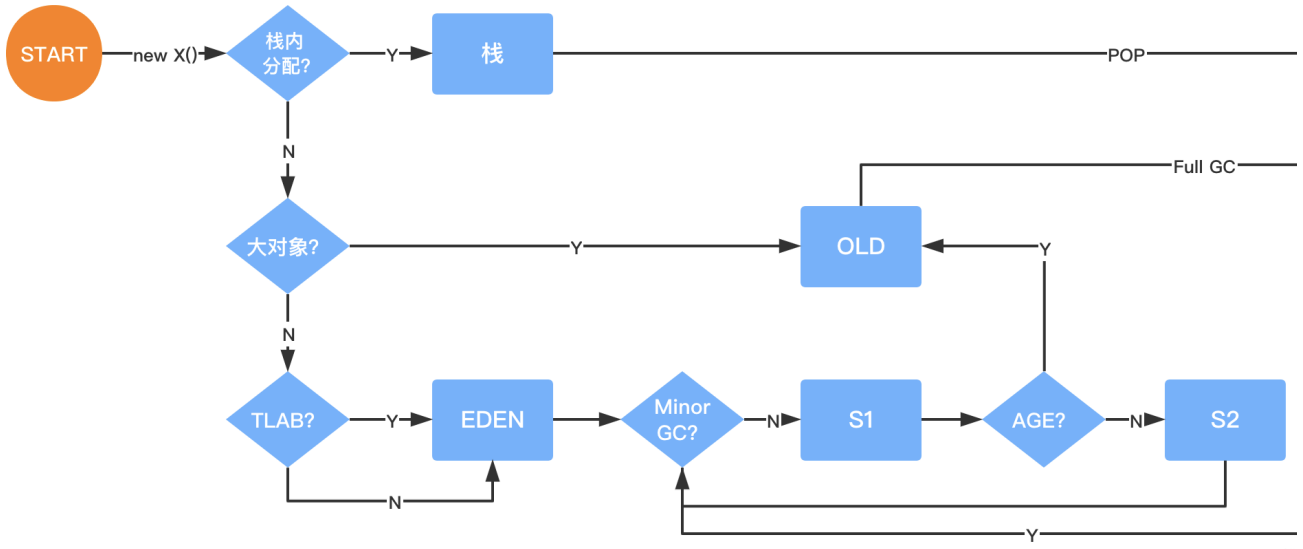
什么是java对象的**指针压缩**?

1.jdk1.6 update14开始，在64bit操作系统中，JVM支持指针压缩

2.jvm配置参数:UseCompressedOops，compressed--压缩、oop(ordinary object pointer)--对象指针

3.启用指针压缩:-XX:+UseCompressedOops(**默认开启**)，禁止指针压缩:-XX:-UseCompressedOops

10、对象内存分配流程是怎样的



11、解释下对象栈上分配、逃逸分析与标量替换

对象栈上分配

我们通过JVM内存分配可以知道JAVA中的对象都是在堆上进行分配，当对象没有被引用的时候，需要依靠GC进行回收内存，如果对象数量较多的时候，会给GC带来较大压力，也间接影响了应用的性能。为了减少临时对象在堆内分配的数量，JVM通过**逃逸分析**确定该对象不会被外部访问。如果不会逃逸可以将该对象在**栈上分配**内存，这样该对象所占用的内存空间就可以随栈帧出栈而销毁，就减轻了垃圾回收的压力。

对象逃逸分析：就是分析对象动态作用域，当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他地方中。

```
public User test1() {
 User user = new User();
 user.setId(1);
 user.setName("zhuge");
 //TODO 保存到数据库
 return user;
}

public void test2() {
 User user = new User();
 user.setId(1);
 user.setName("zhuge");
 //TODO 保存到数据库
}
```

很显然test1方法中的user对象被返回了，这个对象的作用域范围不确定，test2方法中的user对象我们可以确定当方法结束这个对象就可以认为是无效对象了，对于这样的对象我们其实可以将其分配在栈内存里，让其在方法结束时跟随栈内存一起被回收掉。

JVM对于这种情况可以通过开启逃逸分析参数(-XX:+DoEscapeAnalysis)来优化对象内存分配位置，使其通过**标量替换**优先分配在栈上(**栈上分配**)，**JDK7之后默认开启逃逸分析**，如果要关闭使用参数(-XX:-DoEscapeAnalysis)

标量替换：通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，**JVM不会创建该对象**，而是将该对象成员变量分解若干个被这个方法使用的成员变量所代替，这些代替的成员变量在栈帧或寄存器上分配空间，这样就不会因为没有一大块连续空间导致对象内存不够分配。开启标量替换参数(-XX:+EliminateAllocations)，**JDK7之后默认开启。**

标量与聚合量：标量即不可被进一步分解的量，而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。

栈上分配示例：

```java
/**
 * 栈上分配，标量替换
 * 代码调用了1亿次alloc()，如果是分配到堆上，大概需要1GB以上堆空间，如果堆空间小于该值，必然会触发GC。
 *
 * 使用如下参数不会发生GC
 * -Xmx15m -Xms15m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:+EliminateAllocations
 * 使用如下参数都会发生大量GC
 * -Xmx15m -Xms15m -XX:-DoEscapeAnalysis -XX:+PrintGC -XX:+EliminateAllocations
 * -Xmx15m -Xms15m -XX:+DoEscapeAnalysis -XX:+PrintGC -XX:-EliminateAllocations
 */
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("zhuge");
 }
}
```

结论：栈上分配依赖于逃逸分析和标量替换

12、判断对象是否是垃圾的引用计数法有什么问题

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器就减1；任何时候计数器为0的对象就是不可能再被使用的。

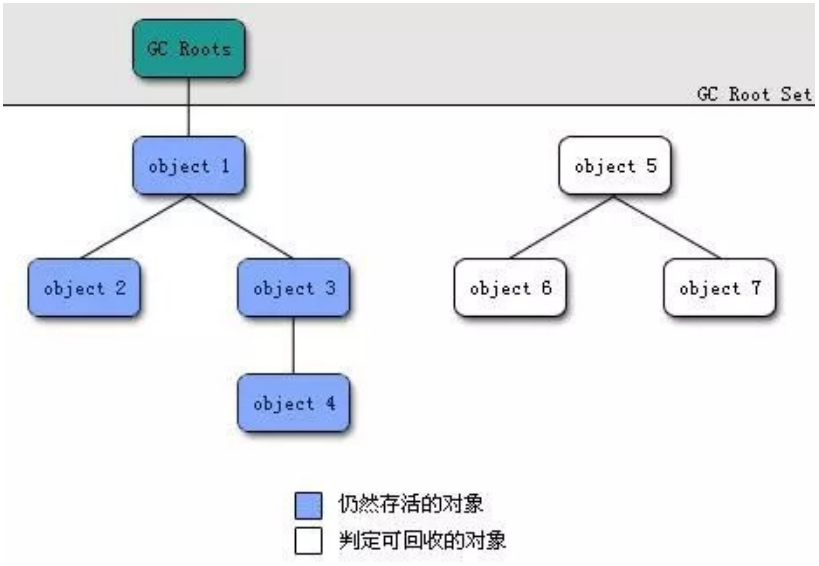
这个方法实现简单，效率高，但是目前主流的虚拟机中并没有选择这个算法来管理内存，其最主要的原因是它很难解决对象之间相互循环引用的问题。 所谓对象之间的相互引用问题，如下面代码所示：除了对象objA 和 objB 相互引用着对方之外，这两个对象之间再无任何引用。但是他们因为互相引用对方，导致它们的引用计数器都不为0，于是引用计数算法无法通知 GC 回收器回收他们。

```java
public class ReferenceCountingGc {
 Object instance = null;

 public static void main(String[] args) {
 ReferenceCountingGc objA = new ReferenceCountingGc();
 ReferenceCountingGc objB = new ReferenceCountingGc();
 objA.instance = objB;
 objB.instance = objA;
 objA = null;
 objB = null;
 }
}
```

13、画图解释下JVM底层可达性分析算法如何找垃圾对象

将“GC Roots” 对象作为起点，从这些节点开始向下搜索引用的对象，找到的对象都标记为**非垃圾对象**，其余未标记的对象都是垃圾对象

GC Roots根节点：线程栈的本地变量、静态变量、本地方法栈的变量等等



15、可达性分析算法中不可达的对象还有机会存活吗

finalize()方法最终判定对象是否存活

即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历再次标记过程。

标记的前提是对象在进行可达性分析后发现没有与GC Roots相连接的引用链。

1. 第一次标记并进行一次筛选。

筛选的条件是此对象是否有必要执行finalize()方法。

当对象没有覆盖finalize方法，对象将直接被回收。

2. 第二次标记

如果这个对象覆盖了finalize方法，finalize方法是对象脱逃死亡命运的最后一次机会，如果对象要在finalize()中成功拯救自己，只要重新与引用链上的任何的一个对象建立关联即可，譬如把自己赋值给某个类变量或对象的成员变量，那在第二次标记时它将移除出“即将回收”的集合。如果对象这时候还没逃脱，那基本上它就真的被回收了。

注意：一个对象的finalize()方法只会被执行一次，也就是说通过调用finalize方法自我救命的机会就一次。

示例代码：

```
public class OOMTest {

 public static void main(String[] args) {
 List<Object> list = new ArrayList<>();
 int i = 0;
 int j = 0;
 while (true) {
 list.add(new User(i++, UUID.randomUUID().toString()));
 new User(j--, UUID.randomUUID().toString());
  }
  }
}


//User类需要重写finalize方法
@Override
protected void finalize() throws Throwable {
  OOMTest.list.add(this);
  System.out.println("关闭资源，userid=" + id + "即将被回收");
}
```

finalize()方法的运行代价高昂， 不确定性大， 无法保证各个对象的调用顺序， 如今已被官方明确声明为不推荐使用的语法。 有些资料描述它适合做“关闭外部资源”之类的清理性工作， 这完全是对finalize()方法用途的一种自我安慰。finalize()能做的所有工作， 使用try-finally或者其他方式都可以做得更好、更及时， 所以建议大家完全可以忘掉Java语言里面的这个方法。

16、什么样的类能被回收

- 该类所有的对象实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

17、解释下JVM内部各种垃圾收集算法

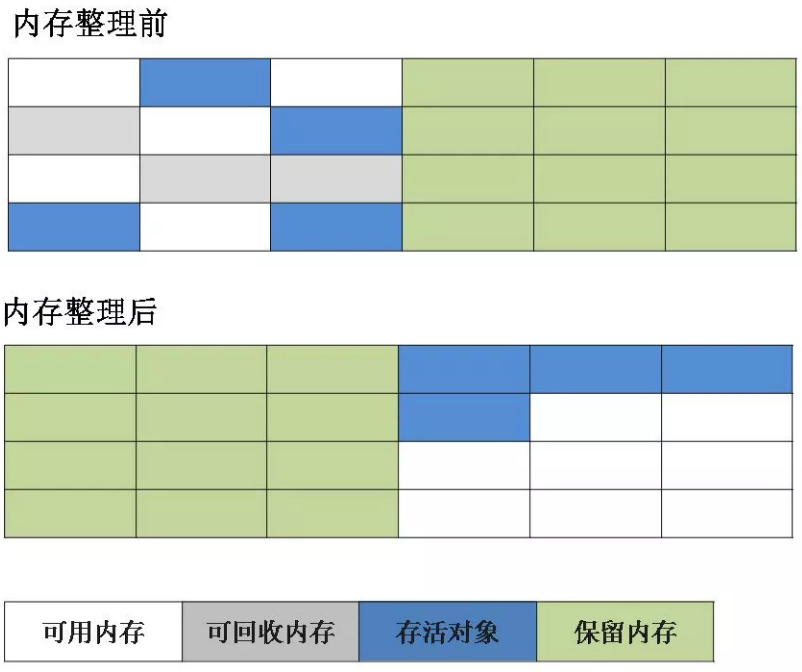
分代收集理论

当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将java堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象(近99%)死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。注意， “标记-清除”或“标记-整理”算法会比复制算法慢10倍以上。

复制算法

为了解决效率问题， “复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

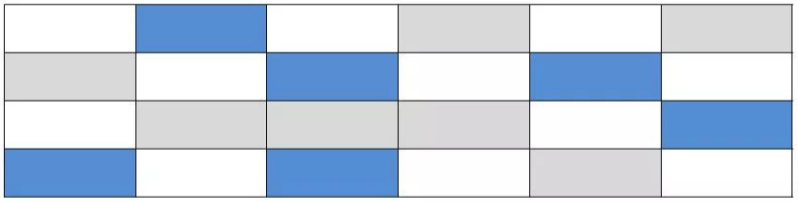


标记-清除算法

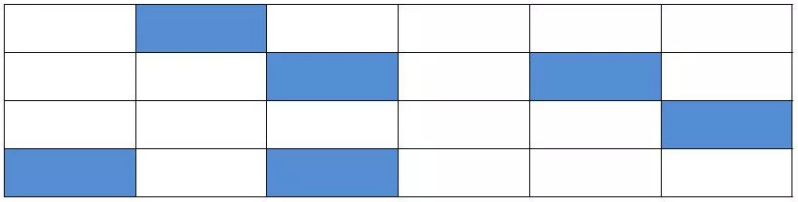
算法分为“标记”和“清除”阶段：标记存活的对象， 统一回收所有未被标记的对象(一般选择这种)；也可以反过来，标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象 。它是最基础的收集算法，比较简单，但是会带来两个明显的问题：

1. **效率问题（如果需要标记的对象太多，效率不高）**
2. **空间问题（标记清除后会产生大量不连续的碎片）**

内存整理前

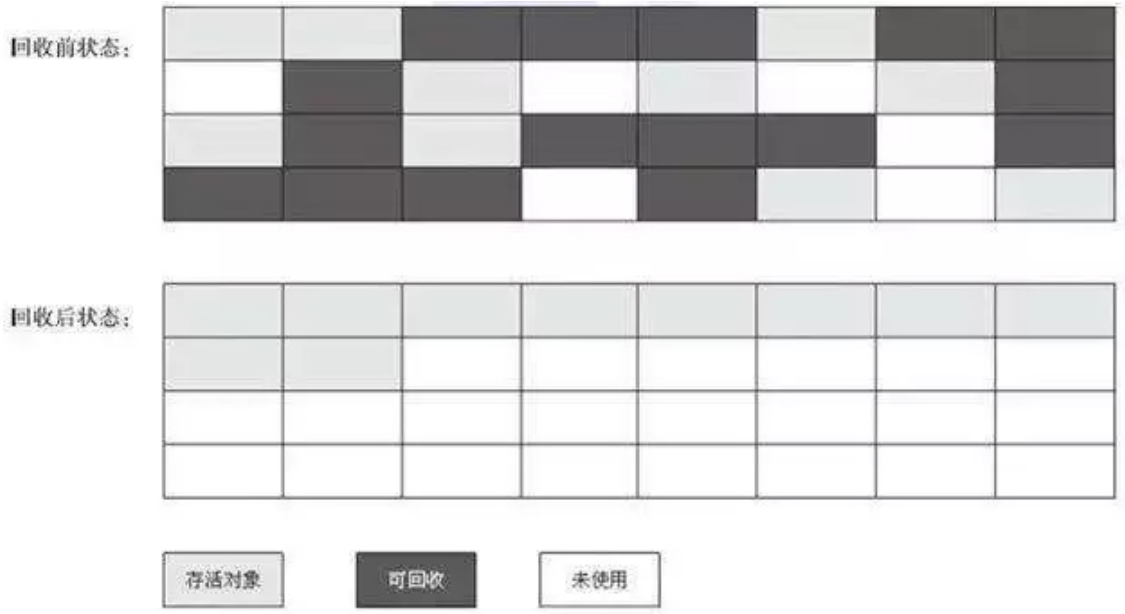


内存整理后



标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。



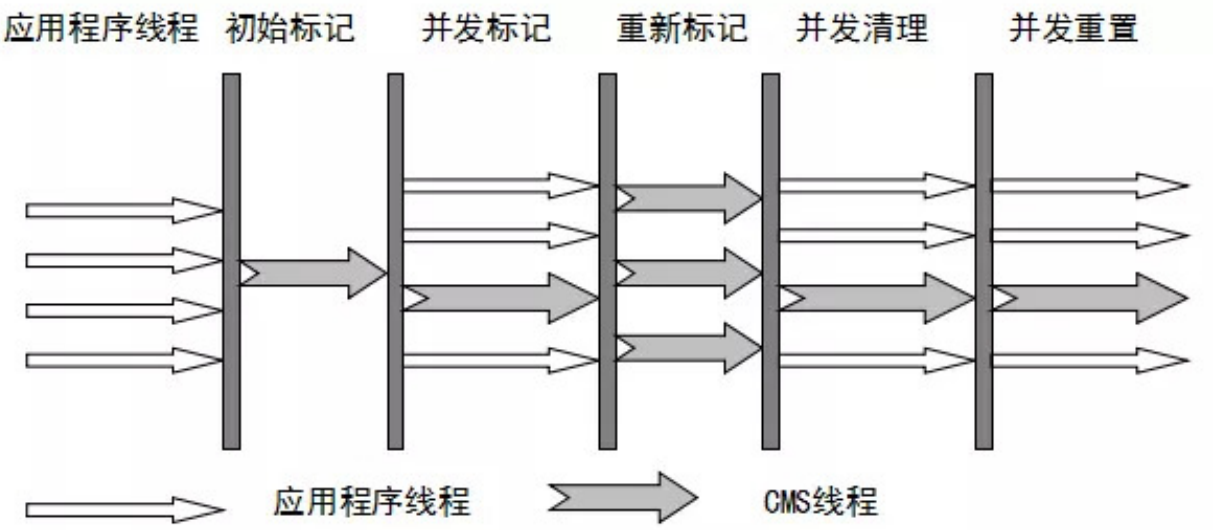
18、解释下CMS收集器垃圾收集过程

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。它非常符合在注重用户体验的应用上使用，它是HotSpot虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

从名字中的**Mark Sweep**这两个词可以看出，CMS收集器是一种 **“标记-清除”算法**实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- **初始标记：** 暂停所有的其他线程(STW)，并记录下gc roots**直接能引用的对象，速度很快。**

- **并发标记：** 并发标记阶段就是从GC Roots的直接关联对象开始遍历整个对象图的过程， 这个过程耗时较长但是不需要停顿用户线程， 可以与垃圾收集线程一起并发运行。因为用户程序继续运行，可能会有导致已经标记过的对象状态发生改变。
- **重新标记：** 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录(**主要是处理漏标问题**)，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短。**主要用到三色标记里的增量更新算法(见下面详解)做重新标记。**
- **并发清理：** 开启用户线程，同时GC线程开始对未标记的区域做清扫。这个阶段如果有新增对象会被标记为黑色不做任何处理(见下面三色标记算法详解)。
- **并发重置：**重置本次GC过程中的标记数据。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面几个明显的缺点：

- 对CPU资源敏感（会和服务抢资源）；
- 无法处理**浮动垃圾**(在并发标记和并发清理阶段又产生垃圾，这种浮动垃圾只能等到下一次gc再清理了)；
- 它使用的回收算法-**“标记-清除”算法**会导致收集结束时会有**大量空间碎片**产生，当然通过参数-XX:+UseCMSCompactAtFullCollection可以让jvm在执行完标记清除后再做整理
    - 执行过程中的不确定性，会存在上一次垃圾回收还没执行完，然后垃圾回收又被触发的情况，**特别是在并发标记和并发清理阶段会出现**，一边回收，系统一边运行，也许没回收完就再次触发full gc，也就是"**concurrent mode failure**"，**此时会进入stop the world，用serial old垃圾收集器来回收**

20、CMS比较严重的问题并发收集阶段再次触发Full gc怎么处理

CMS垃圾收集器收集垃圾过程中，会存在上一次垃圾回收还没执行完，然后垃圾回收又被触发的情况，**特别是在并发标记和并发清理阶段会出现**，一边回收，系统一边运行，也许没回收完就再次触发full gc，也就是"**concurrent mode failure**"，**此时会进入stop the world，用serial old垃圾收集器来回收**

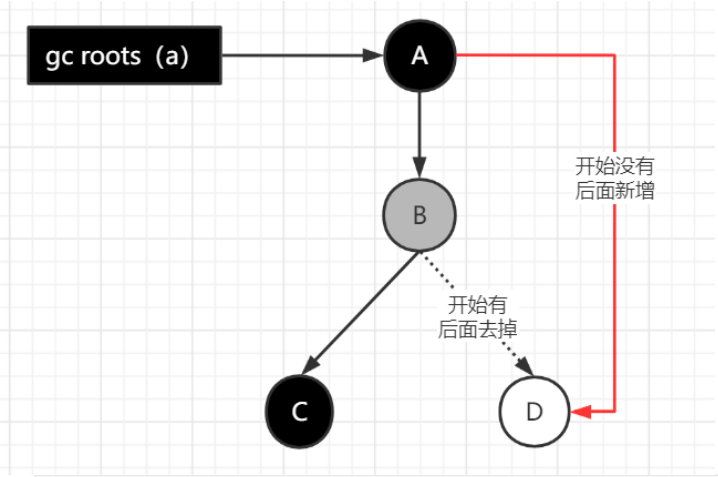
21、解释下垃圾收集底层三色标记算法

在并发标记的过程中，因为标记期间应用线程还在继续跑，对象间的引用可能发生变化，多标和漏标的情况就有可能发生。漏标的问题主要引入了三色标记算法来解决。

三色标记算法把Gc roots可达性分析遍历对象过程中遇到的对象， 按照“是否访问过”这个条件标记成以下三种颜色：

- **黑色**： 表示对象已经被垃圾收集器访问过， 且这个对象的所有引用都已经扫描过。 黑色的对象代表已经扫描过， 它是安全存活的， 如果有其他对象引用指向了黑色对象， 无须重新扫描一遍。 黑色对象不可能直接（不经过灰色对象） 指向某个白色对象。
- **灰色**： 表示对象已经被垃圾收集器访问过， 但这个对象上至少存在一个引用还没有被扫描过。

- **白色**：表示对象尚未被垃圾收集器访问过。显然在可达性分析刚刚开始的阶段，所有的对象都是白色的，若在分析结束的阶段，仍然是白色的对象，即代表不可达。




```
/**
 * 垃圾收集算法细节之三色标记
 * 为了简化例子，代码写法可能不规范，请忽略
 * Created by 诸葛老师
 */
public class ThreeColorRemark {

 public static void main(String[] args) {
 A a = new A();
  //开始做并发标记
  D d = a.b.d; // 1.读
  a.b.d = null; // 2.写
  a.d = d; // 3.写
  }
}

class A {
 B b = new B();
 D d = null;
}

class B {
 C c = new C();
 D d = new D();
}

class C {
}

class D {
}
```

22、解释下对象漏标的处理方案增量更新与原始快照(SATB)

漏标会导致被引用的对象被当成垃圾误删除，这是严重bug，必须解决，有两种解决方案：**增量更新（Incremental Update）和原始快照（Snapshot At The Beginning，SATB）。**

增量更新就是当黑色对象插入新的指向白色对象的引用关系时，就将这个新插入的引用记录下来，等并发扫描结束之后，再将这些记录过的引用关系中的黑色对象为根，重新扫描一次。这可以简化理解为，**黑色对象一旦新插入了指向白色对象的引用之后，它就变回灰色对象了。**

原始快照就是当灰色对象要删除指向白色对象的引用关系时，就将这个要删除的引用记录下来，在并发扫描结束之后，再将这些记录过的引用关系中的灰色对象为根，重新扫描一次，这样就能扫描到白色的对象，将白色对象直接标记为黑

色(**目的就是让这种对象在本轮gc清理中能存活下来，待下一轮gc的时候重新扫描，这个对象也有可能是浮动垃圾**)

23、CMS、G1与ZGC对于对象漏标问题的处理区别

现代追踪式（可达性分析）的垃圾回收器几乎都借鉴了三色标记的算法思想，尽管实现的方式不尽相同：比如白色/黑色集合一般都不会出现（但是有其他体现颜色的地方）、灰色集合可以通过栈/队列/缓存日志等方式进行实现、遍历方式可以是广度/深度遍历等等。

对于读写屏障，以Java HotSpot VM为例，其并发标记时对漏标的处理方案如下：

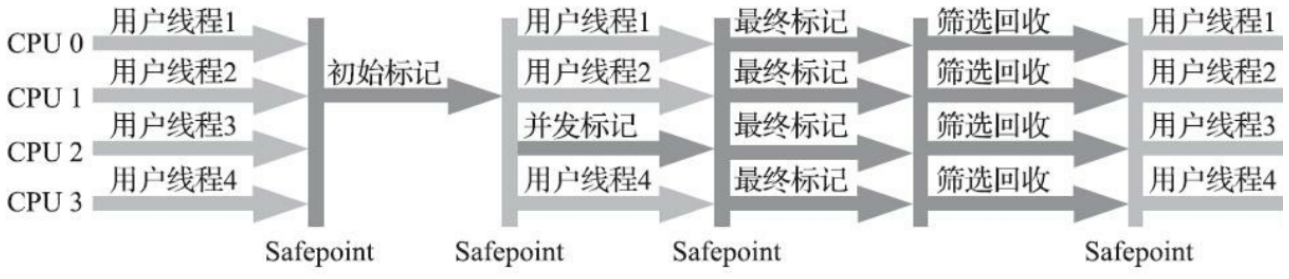
- **CMS：写屏障 + 增量更新**
- **G1，Shenandoah：写屏障 + SATB**
- **ZGC：读屏障**

工程实现中，读写屏障还有其他功能，比如写屏障可以用于记录跨代/区引用的变化，读屏障可以用于支持移动对象的并发执行等。功能之外，还有性能的考虑，所以对于选择哪种，每款垃圾回收器都有自己的想法。

24、解释下G1收集器垃圾收集过程

G1收集器一次GC(主要值Mixed GC)的运作过程大致分为以下几个步骤：

- **初始标记**（initial mark，STW）：暂停所有的其他线程，并记录下gc roots直接能引用的对象，**速度很快**；
- **并发标记**（Concurrent Marking）：同CMS的并发标记
- **最终标记**（Remark，STW）：同CMS的重新标记
- **筛选回收**（Cleanup，STW）：筛选回收阶段首先对各个Region的**回收价值和成本进行排序**，**根据用户所期望的GC停顿STW时间(可以用JVM参数 -XX:MaxGCPauseMillis指定)来制定回收计划**，比如说老年代此时有1000个Region都满了，但是因为根据预期停顿时间，本次垃圾回收可能只能停顿200毫秒，那么通过之前回收成本计算得知，可能回收其中800个Region刚好需要200ms，那么就只会回收800个Region(**Collection Set**，要回收的集合)，尽量把GC导致的停顿时间控制在我们指定的范围内。这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。不管是年轻代或是老年代，**回收算法主要用的是复制算法，将一个region中的存活对象复制到另一个region中，这种不会像CMS那样回收完因为有很多内存碎片还需要整理一次，G1采用复制算法回收几乎不会有太多内存碎片。**(注意：CMS回收阶段是跟用户线程一起并发执行的，G1因为内部实现太复杂暂时没实现并发回收，不过到了ZGC，Shenandoah就实现了并发收集，Shenandoah可以看成是G1的升级版本)



25、G1垃圾收集器最大停顿时间是如何实现的

G1收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的Region(这也就是它的名字Garbage-First的由来)，比如一个Region花200ms能回收1M垃圾，另外一个Region花50ms能回收2M垃圾，在回收时间有限情况下，G1当然会优先选择后面这个Region回收。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限时间内可以尽可能高的收集效率。

26、内存泄露到底是怎么回事，怎么快速排查与解决

再给大家讲一种情况，一般电商架构可能会使用多级缓存架构，就是redis加上JVM级缓存，大多数同学可能为了图方便对于JVM级缓存就简单使用一个hashmap，于是不断往里面放缓存数据，但是很少考虑这个map的容量问题，结果这个缓存map越来越大，一直占用着老年代的很多空间，时间长了就会导致full gc非常频繁，这就是一种内存泄漏，对于一些老旧数据没有及时清理导致一直占用着宝贵的内存资源，时间长了除了导致full gc，还有可能导致OOM。

这种情况完全可以考虑采用一些成熟的JVM级缓存框架来解决，比如ehcache等自带一些LRU数据淘汰算法的框架来作为JVM级的缓存。

27、GC是什么时候都能做吗？知道GC安全点与安全区域是怎么回事吗？

GC不是任何时候都能做的，需要代码运行到安全点或安全区域才能做。

安全点就是指代码中一些特定的位置,当线程运行到这些位置时它的状态是确定的,这样JVM就可以安全的进行一些操作,比如GC等，所以GC不是想什么时候做就立即触发的，是需要等待所有线程运行到安全点后才能触发。

这些特定的安全点位置主要有以下几种:

1. 方法返回之前
2. 调用某个方法之后
3. 抛出异常的位置
4. 循环的末尾

大体实现思想是当垃圾收集需要中断线程的时候， 不直接对线程操作， 仅仅简单地设置一个标志位， 各个线程执行过程时会不停地主动去轮询这个标志， 一旦发现中断标志为真时就自己在最近的安全点上主动中断挂起。 轮询标志的地方和安全点是重合的。

安全区域又是什么?

Safe Point 是对正在执行的线程设定的。

如果一个线程处于 Sleep 或中断状态，它就不能响应 JVM 的中断请求，再运行到 Safe Point 上。

因此 JVM 引入了 Safe Region。

Safe Region 是指在一段代码片段中，引用关系不会发生变化。在这个区域内的任意地方开始 GC 都是安全的。

28、解释下字符串常量池

字符串常量池的设计思想

1. 字符串的分配，和其他的对象分配一样，耗费高昂的时间与空间代价，作为最基础的数据类型，大量频繁的创建字符串，极大程度地影响程序的性能
2. JVM为了提高性能和减少内存开销，在实例化字符串常量的时候进行了一些优化

- 为字符串开辟一个字符串常量池，类似于缓存区
- 创建字符串常量时，首先查询字符串常量池是否存在该字符串
- 存在该字符串，返回引用实例，不存在，实例化该字符串并放入池中

三种字符串操作(Jdk1.7 及以上版本)

- 直接赋值字符串

```
String s = "zhuge"; // s指向常量池中的引用

```

这种方式创建的字符串对象，只会在常量池中。

因为有"zhuge"这个字面量，创建对象s的时候，JVM会先去常量池中通过 equals(key) 方法，判断是否有相同的对象如果有，则直接返回该对象在常量池中的引用；

如果没有，则会在常量池中创建一个新对象，再返回引用。

- new String();

```
String s1 = new String("zhuge"); // s1指向内存中的对象引用
```

这种方式会保证字符串常量池和堆中都有这个对象，没有就创建，最后返回堆内存中的对象引用。
步骤大致如下：
因为有"zhuge"这个字面量，所以会先检查字符串常量池中是否存在字符串"zhuge"
不存在，先在字符串常量池里创建一个字符串对象；再去内存中创建一个字符串对象"zhuge"；
存在的话，就直接去堆内存中创建一个字符串对象"zhuge"；
最后，将内存中的引用返回。

- intern方法

```
String s1 = new String("zhuge");
String s2 = s1.intern();

System.out.println(s1 == s2); //false
```

String中的intern方法是一个 native 的方法，当调用 intern方法时，如果池已经包含一个等于此String对象的字符串（用equals(oject)方法确定），则返回池中的字符串。**否则，将intern返回的引用指向当前字符串 s1(jdk1.6版本需要将s1 复制到字符串常量池里)**。

字符串常量池位置

Jdk1.6及之前： 有永久代, 运行时常量池在永久代，运行时常量池包含字符串常量池
Jdk1.7：有永久代，但已经逐步“去永久代”，字符串常量池从永久代里的运行时常量池分离到堆里
Jdk1.8及之后： 无永久代，运行时常量池在元空间，字符串常量池里依然在堆里
用一个程序证明下字符串常量池在哪里：

```
/**
 * jdk6: -Xms6M -Xmx6M -XX:PermSize=6M -XX:MaxPermSize=6M
 * jdk8: -Xms6M -Xmx6M -XX:MetaspaceSize=6M -XX:MaxMetaspaceSize=6M
 */
public class RuntimeConstantPoolOOM{
 public static void main(String[] args) {
 ArrayList<String> list = new ArrayList<String>();
 for (int i = 0; i < 10000000; i++) {
 String str = String.valueOf(i).intern();
  list.add(str);
  }
  }
}

运行结果:
jdk7及以上: Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
jdk6: Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

字符串常量池设计原理

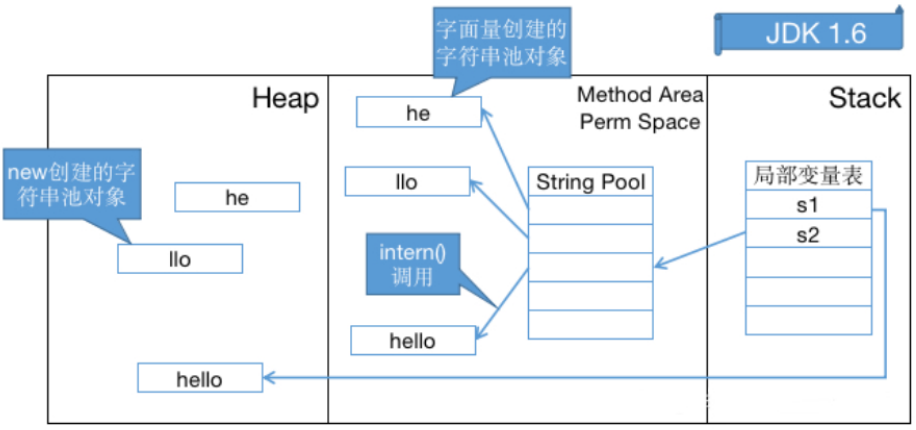
字符串常量池底层是hotspot的C++实现的，底层类似一个 HashTable， 保存的本质上是字符串对象的引用。
看一道比较常见的面试题，下面的代码创建了多少个 String 对象?

```
String s1 = new String("he") + new String("llo");
String s2 = s1.intern();

System.out.println(s1 == s2);
// 在 JDK 1.6 下输出是 false，创建了 6 个对象
```

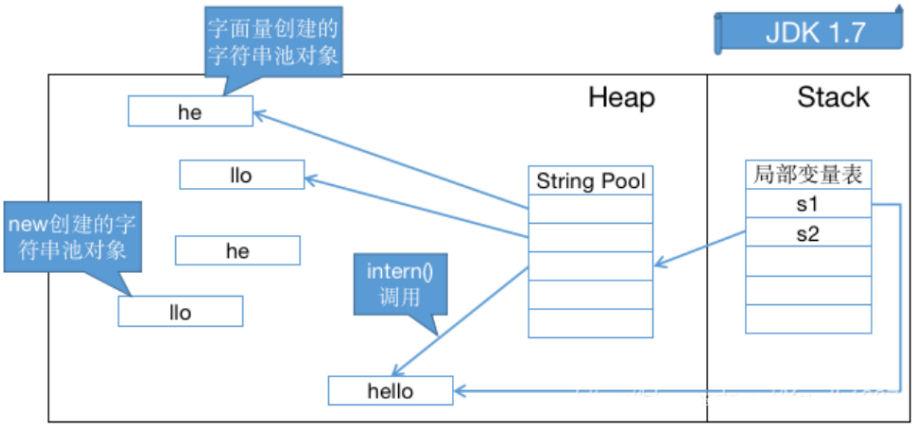
```
// 在 JDK 1.7 及以上的版本输出是 true，创建了 5 个对象
// 当然我们这里没有考虑GC，但这些对象确实存在或存在过

```

为什么输出会有这些变化呢？主要还是字符串池从永久代中脱离、移入堆区的原因， intern() 方法也相应发生了变化：

1、在 JDK 1.6 中，调用 intern() 首先会在字符串池中寻找 equal() 相等的字符串，假如字符串存在就返回该字符串在字符串池中的引用；假如字符串不存在，虚拟机会重新在永久代上创建一个实例，将 StringTable 的一个表项指向这个新创建的实例。



2、在 JDK 1.7 (及以上版本)中，由于字符串池不在永久代了，intern() 做了一些修改，更方便地利用堆中的对象。字符串存在时和 JDK 1.6一样，但是字符串不存在时不再需要重新创建实例，可以直接指向堆上的实例。



由上面两个图，也不难理解为什么 JDK 1.6 字符串池溢出会抛出 OutOfMemoryError: PermGen space ，而在 JDK 1.7 及以上版本抛出 OutOfMemoryError: Java heap space 。

String常量池问题的几个例子

示例1：

```
String s0="zhuge";
String s1="zhuge";
String s2="zhu" + "ge";
System.out.println( s0==s1 ); //true
System.out.println( s0==s2 ); //true

```

分析：因为例子中的 s0和s1中的”zhuge”都是字符串常量，它们在编译期就被确定了，所以s0==s1为true；

而”zhu”和”ge”也都是字符串常量，当一个字 符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，所

以s2也同样在编译期就被优化为一个字符串常量"zhuge"，所以s2也是常量池中” zhuge”的一个引用。所以我们得出s0==s1==s2；

示例2：

```
String s0="zhuge";
String s1=new String("zhuge");
String s2="zhu" + new String("ge");
System.out.println( s0==s1 );    // false
System.out.println( s0==s2 );    // false
System.out.println( s1==s2 );    // false

```

分析：用new String() 创建的字符串不是常量，不能在编译期就确定，所以new String() 创建的字符串不放入常量池中，它们有自己的地址空间。
s0还是常量池 中"zhuge”的引用，s1因为无法在编译期确定，所以是运行时创建的新对象”zhuge”的引用，s2因为有后半部分 new String(”ge”)所以也无法在编译期确定，所以也是一个新创建对象”zhuge”的引用;明白了这些也就知道为何得出此结果了。

示例3：

```
 String a = "a1";
 String b = "a" + 1;
 System.out.println(a == b); // true

 String a = "atrue";
 String b = "a" + "true";
 System.out.println(a == b); // true

 String a = "a3.4";
  String b = "a" + 3.4;
  System.out.println(a == b); // true

```

分析：JVM对于字符串常量的"+"号连接，将在程序编译期，JVM就将常量字符串的"+"连接优化为连接后的值，拿"a" + 1来说，经编译器优化后在class中就已经是a1。在编译期其字符串常量的值就确定下来，故上面程序最终的结果都为true。

示例4：

```
String a = "ab";
String bb = "b";
String b = "a" + bb;

System.out.println(a == b); // false

```

分析：JVM对于字符串引用，由于在字符串的"+"连接中，有字符串引用存在，而引用的值在程序编译期是无法确定的，即"a" + bb无法被编译器优化，只有在程序运行期来动态分配并将连接后的新地址赋给b。所以上面程序的结果也就为false。

示例5：

```
String a = "ab";
final String bb = "b";
String b = "a" + bb;

System.out.println(a == b); // true

```

分析：和示例4中唯一不同的是bb字符串加了final修饰，对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中。所以此时的"a" + bb和"a" + "b"效果是一样的。故上面程序的结果

为true。

示例6：

```
String a = "ab";
final String bb = getBB();
String b = "a" + bb;

System.out.println(a == b); // false

private static String getBB()
{
 return "b";
  }

```

分析：JVM对于字符串引用bb，它的值在编译期无法确定，只有在程序运行期调用方法后，将方法的返回值和"a"来动态连接并分配地址为b，故上面 程序的结果为false。

关于String是不可变的

通过上面例子可以得出得知：

```
String  s  =  "a" + "b" + "c"; //就等价于String s = "abc";
String  a  =  "a";
String  b  =  "b";
String  c  =  "c";
String  s1  =   a  +  b  +  c;

```

s1 这个就不一样了，可以通过观察其**JVM指令码**发现s1的"+"操作会变成如下操作：

```
StringBuilder temp = new StringBuilder();
temp.append(a).append(b).append(c);
String s = temp.toString();

```

最后再看一个例子：

```
//字符串常量池："计算机"和"技术" 堆内存：str1引用的对象"计算机技术"
//堆内存中还有个StringBuilder的对象，但是会被gc回收，StringBuilder的toString方法会new String()，这个String才是真正返回的对象引用
String str2 = new StringBuilder("计算机").append("技术").toString(); //没有出现"计算机技术"字面量，所以不会在常量池里生成"计算机技术"对象
System.out.println(str2 == str2.intern()); //true
//"计算机技术" 在池中没有，但是在heap中存在，则intern时，会直接返回该heap中的引用

//字符串常量池："ja"和"va" 堆内存：str1引用的对象"java"
//堆内存中还有个StringBuilder的对象，但是会被gc回收，StringBuilder的toString方法会new String()，这个String才是真正返回的对象引用
String str1 = new StringBuilder("ja").append("va").toString(); //没有出现"java"字面量，所以不会在常量池里生成"java"对象
 System.out.println(str1 == str1.intern()); //false
 //java是关键字，在JVM初始化的相关类里肯定早就放进字符串常量池了

 String s1=new String("test");
 System.out.println(s1==s1.intern()); //false
 //"test"作为字面量，放入了池中，而new时s1指向的是heap中新生成的string对象，s1.intern()指向的是"test"字面量之前在池中生成的字符串对象

 String s2=new StringBuilder("abc").toString();
 System.out.println(s2==s2.intern()); //false
 //同上

```

29、八种基本类型包装类的常量池是如何实现的

java中基本类型的包装类的大部分都实现了常量池技术(严格来说应该叫**对象池**，在堆上)，这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另外Byte,Short,Integer,Long,Character这5种整型的包装类也只是在对应值小于等于127时才可使用对象池，也即对象不负责创建和管理大于127的这些类的对象。因为一般这种比较小的数用到的概率相对较大。

```
public class Test {

 public static void main(String[] args) {
 //5种整形的包装类Byte,Short,Integer,Long,Character的对象，
 //在值小于127时可以使用对象池
 Integer i1 = 127; //这种调用底层实际是执行的Integer.valueOf(127)，里面用到了IntegerCache对象池
 Integer i2 = 127;
 System.out.println(i1 == i2);//输出true

 //值大于127时，不会从对象池中取对象
 Integer i3 = 128;
 Integer i4 = 128;
 System.out.println(i3 == i4);//输出false

 //用new关键词新生成对象不会使用对象池
 Integer i5 = new Integer(127);
 Integer i6 = new Integer(127);
 System.out.println(i5 == i6);//输出false

 //Boolean类也实现了对象池技术
 Boolean bool1 = true;
 Boolean bool2 = true;
 System.out.println(bool1 == bool2);//输出true

 //浮点类型的包装类没有实现对象池技术
 Double d1 = 1.0;
 Double d2 = 1.0;
 System.out.println(d1 == d2);//输出false
 }
}
```