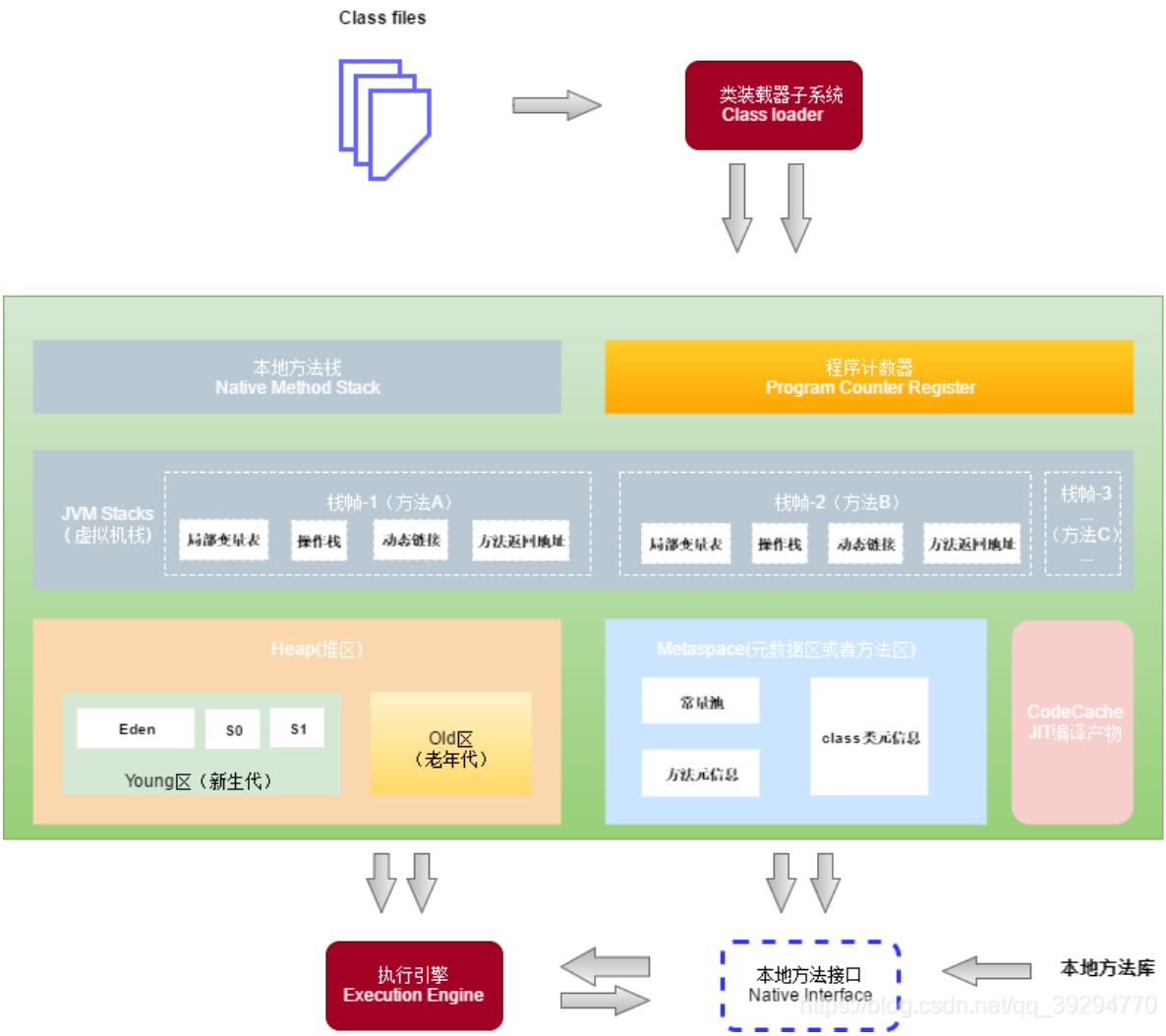
第一章 JVM与Java体系结构

1、JVM的整体结构



2、JVM的生命周期

- 虚拟机的启动
  java虚拟机的启动是通过引导类加载器（**boostrap class loader**）创建一个初始类来完成，这个类是由虚拟机的具体实现指定的。
- 虚拟机的运行
  执行一个java程序的时候，其实执行的是一个java虚拟机进程
- 虚拟机的退出，有如下几种情况：
  1. 程序正常执行退出
  2. 程序在执行中遇到异常或者错误而异常终止
  3. 由于操作系统出现错误导致Java虚拟机进程终止
  4. 某线程调用Runtime类或者System类的exit方法，或者Runtime类的halt方法，并且Java安全管理器也允许这次exit或者halt操作
  5. 除此之外，JNI规范描述用JNI Invocation API来加载或者卸载Java虚拟机时，Java虚拟机的退出情况

3、JVM的发展历程

Sun Classic VM

1、1996年发布java1.0版本的时候，Sun公司发布了一款叫Sun Classic VM的java虚拟机，它是世界上第一款商用java虚拟机，java1.4时被淘汰

2、这款虚拟机只提供解释器（执行性能低下），没有JIT编译器（即时编译器：寻找热点代码，把热点代码编译成本地机器指令，并缓存起来，这样就不用每次都逐行的解释代码，效率得到了很大提升），现在的主流的java虚拟机都有这两个

3、现在的虚拟机都是hotspot

Exact VM: 解决了上一个虚拟机的问题, 只在Solaris平台上短暂使用，其他平台还是classic vm

Hotspot:JDK1.3到现在，hotspot虚拟机成为默认的虚拟机

1、Sun/racle JDK 和OpenJDK的默认虚拟机

2、其他两款商用虚拟机（JRockit，J9）都没有方法区的概念，只有Hotspot有方法区

3、通过计数器找到最具编译价值代码，触发即时编译器或栈上替换()

4、通过编译器与解释器协同工作，在最优化的程序响应时间与最佳执行性能中取得平衡

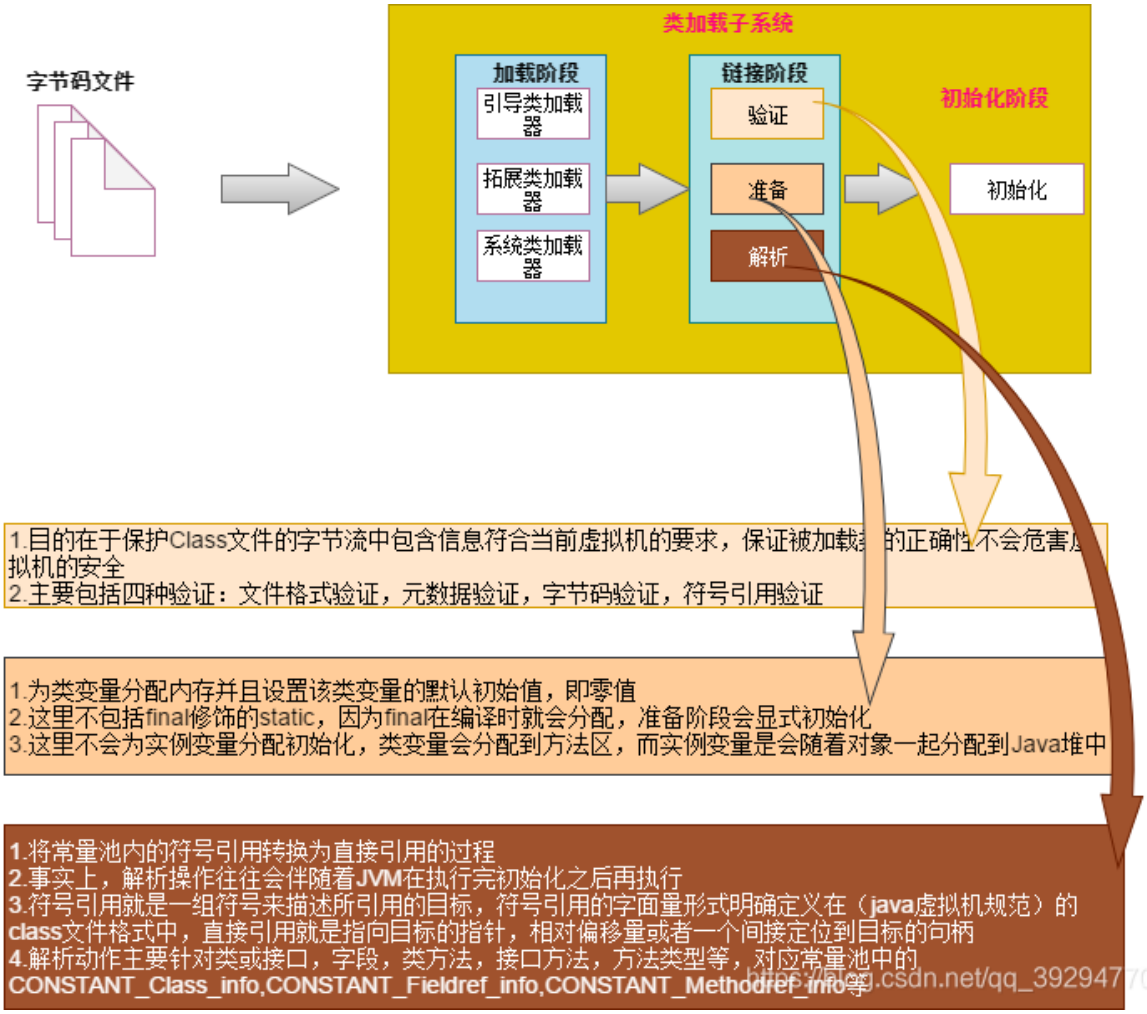
JRockit: 专注于服务器应用

JRockit JVM： 是世界上最快的JVM，适用财务，军事，电信网络

IBM的J9：广泛用于IBM的各种Java产品

第二章 类加载子系统

1、类加载过程



- 类加载器子系统只负责从文件系统或者网络中加载class文件，class文件在文件开头有特定的文件标识。

- ClassLoader只负责class文件的加载，至于它是否可以运行，则由Execution Engine决定
- 加载的类信息存放在方法区的内存空间。除了类的信息外，方法区中还会存放运行时常量池信息，可能还包括字符串字面量和数字常量（这部分常量信息是Class文件中常量池部分的内存映射）

2、加载 .class文件的方式

- 从本地系统中直接加载
- 通过网络获取，典型场景：Web Applet
- 从压缩包中获取，成为日后jar，war格式的基础
- 运行时计算生成：动态代理技术
- 由其他文件生成，典型场景：JSP应用
- 从专业数据库中提取.class文件，比较少见
- 从加密文件中获取，典型的仿class文件被反编译的保护措施

3、类加载器分类

- JVM支持两种类型的类加载器，分别为**引导类加载器**和**自定义加载器**
- 引导类加载器获取不到的
- 系统类加载器可以通过ClassLoader.getSystemClassLoader()方法获取
- 拓展类加载器可以通过ClassLoader.getSystemClassLoader().getParent()方法获取

4、双亲委派机制

Java虚拟机对class文件采用的是按需加载的方式。而且加载某个类的class文件时，Java虚拟机采用的是双亲委派模式，即把请求交由父类处理（而不是给自定义的类处理），它是一种任务委派模式。

优势

避免类的重复加载，只要一层层往上找到父类就不会去加载你自定义的类
保护程序安全，防止核心API被随意篡改（沙箱安全机制）

其他
在JVM中表示两个class对象是否为同一个类存在两个必要条件：
1.类的完整类名必须相同，包括包名
2.加载这个类的ClassLoader（指ClassLoader实例对象）必须相同

第三章 运行时数据区概述及线程

Java虚拟机定义了若干种程序运行期间会使用到的运行时数据区，其中有一些会随着虚拟机启动（进程）而启动，随着虚拟机的退出内销毁，另外一些则是与线程一一对应，这些与线程对应的数据区域会随着线程开始和结束而创建和销毁。

- 每个线程：独立的包括程序计数器，栈，本地栈
- 线程共享：堆，堆外内存（永久代或元空间，代码缓存）

1、程序计数器（PC寄存器）

- JVM中的PC寄存器是对物理PC寄存器的一种抽象模拟
- PC寄存器是用来存储指向下一条指令的地址，因为CPU需要不停的切换各个线程，这个时候切换回来的时候，就得知道接着从哪开始执行

2、虚拟机栈

1. **虚拟机栈的基本概述**

- 由于跨平台的设计，Java的指令都是根据栈来设计的，不同平台的CPU架构不同，所以不能设计为基于寄存器的。优点是跨平台，指令集小，编译器容易实现，缺点是性能下降，实现同样的功能需要更加多的指令
- 栈是运行时单位，而堆是存储的单位

2. **栈的存储单位**

- 一个线程对应一个虚拟机栈，一个个方法对应的是栈内的一个个栈帧（栈的基本单位），方法调用（Java方法有两种返回方式：一种是正常函数返回，使用return指令，一种是抛出异常，不管是哪种方式都会导致栈帧被弹出）对应就是栈帧的**出栈**，所以VM对栈的操作只有进栈和出栈
- 对**栈**来说是不存在垃圾回收的，但是会有**OOM**（异常），因为Java虚拟机规范允许Java栈的大小是动态或者固定不变的

```
package com.lzx;

/**
 * 演示栈中的异常：StackOverflowError
 * 默认情况下：count：9679
 * 设置栈大小：-Xss256k ： count：2472
 */
public class StackErrorTest {
    private static int count = 1;

    public static void main(String[] args) {
        System.out.println(count);
        count++;
        main(args);//自己调自己，比如递归中重复调用
    }
}
Exception in thread "main" java.lang.StackOverflowError
    at com.lzx.StackErrorTest.main(StackErrorTest.java:9)
    at com.lzx.StackErrorTest.main(StackErrorTest.java:9)
    at com.lzx.StackErrorTest.main(StackErrorTest.java:9)
    at com.lzx.StackErrorTest.main(StackErrorTest.java:9)
    at com.lzx.StackErrorTest.main(StackErrorTest.java:9)
    at com.lzx.StackErrorTest.main(StackErrorTest.java:9)
```

- 设置栈内存大小：使用-Xss选项来设置线程的最大栈空间，栈的大小直接决定函数调用的最大可达深度

  每个栈帧中存储着：

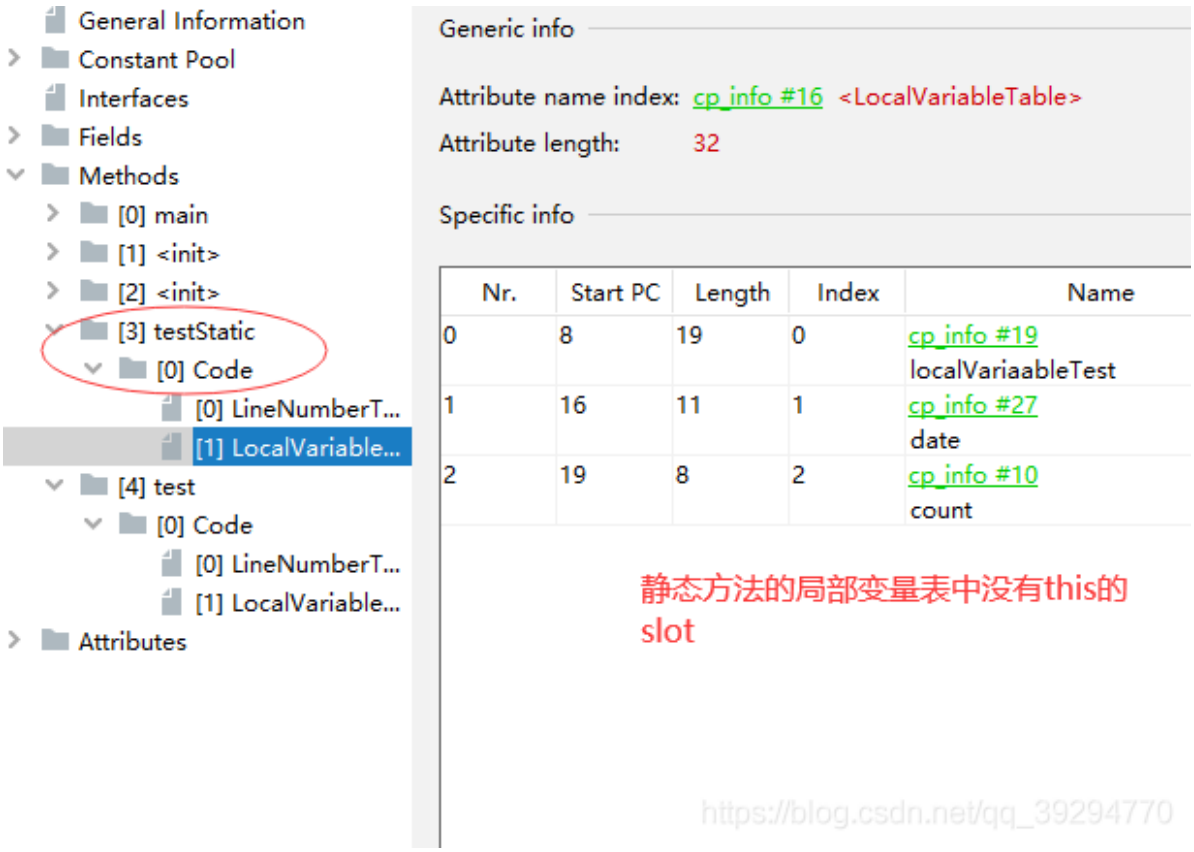
  - 局部变量表
  - 操作数栈（或表达式栈）
  - 动态链接（或指向运行时常量池的方法引用）
  - 方法返回地址（或方法正常退出或者异常退出的定义）

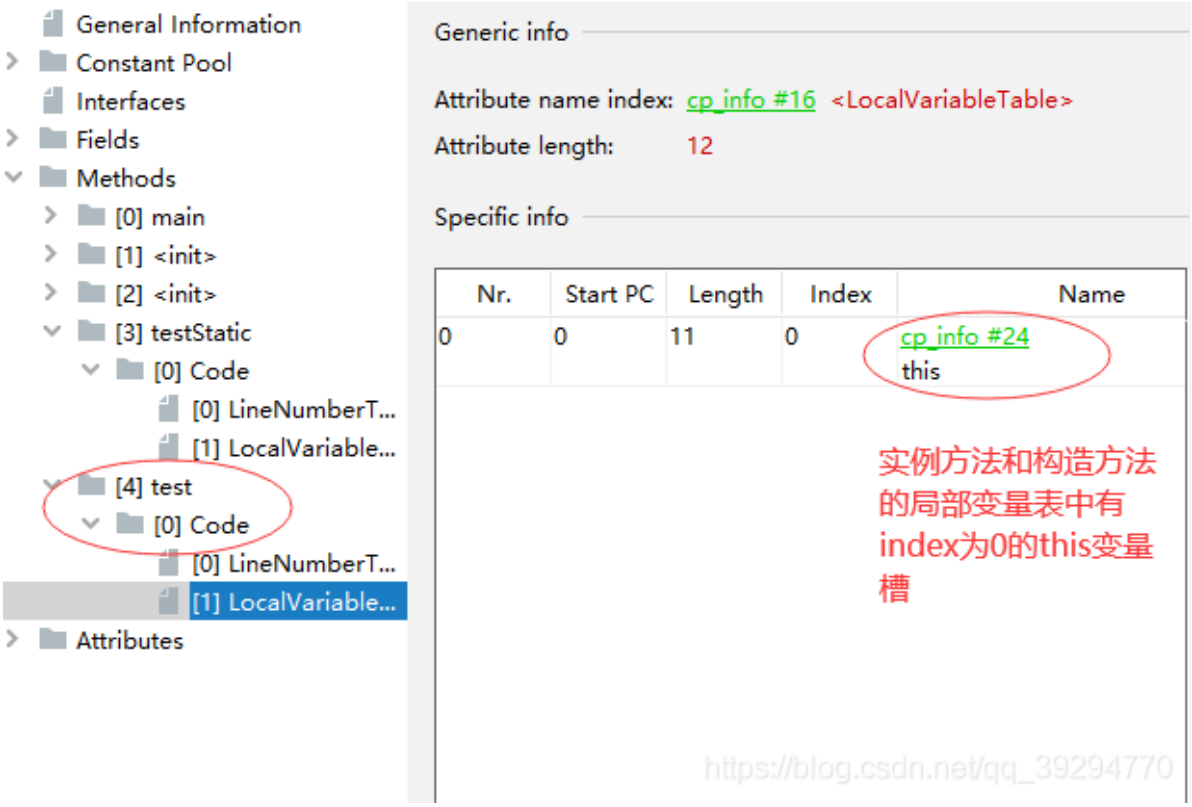
3. **局部变量表**

- 定义为一个数字数组，主要用于存储方法参数和定义方法体内的局部变量
- 由于局部变量表是建立在线程的栈上，是线程的私有数据，因此 **不存在数据安全问题**
- **局部变量表所需的容量大小是在编译期确定下来的**，并保存在Code属性的maximun local variables数据项中。在方法运行期间是不会改变局部变量表的大小的。
- 局部变量表，最基本的存储单元是**Slot（变量槽）**
- **局部变量表中存储的是编译器可知的各种基本数据类型，引用类型，returnAddress类型的变量**

- 在局部变量表中，32位内的类型只占一个slot（包括returnAddress类型），64位的类型（long，double）占两个slot。byte,short,char 在存储前被转换为init，boolean也被转换为int，0表示false，非0表示true.
- 如果当前栈帧是由构造方法或者实例方法创建的，那么该对象引用this将会存放在index为0的slot，其余参数会按照参数表顺序继续排序。

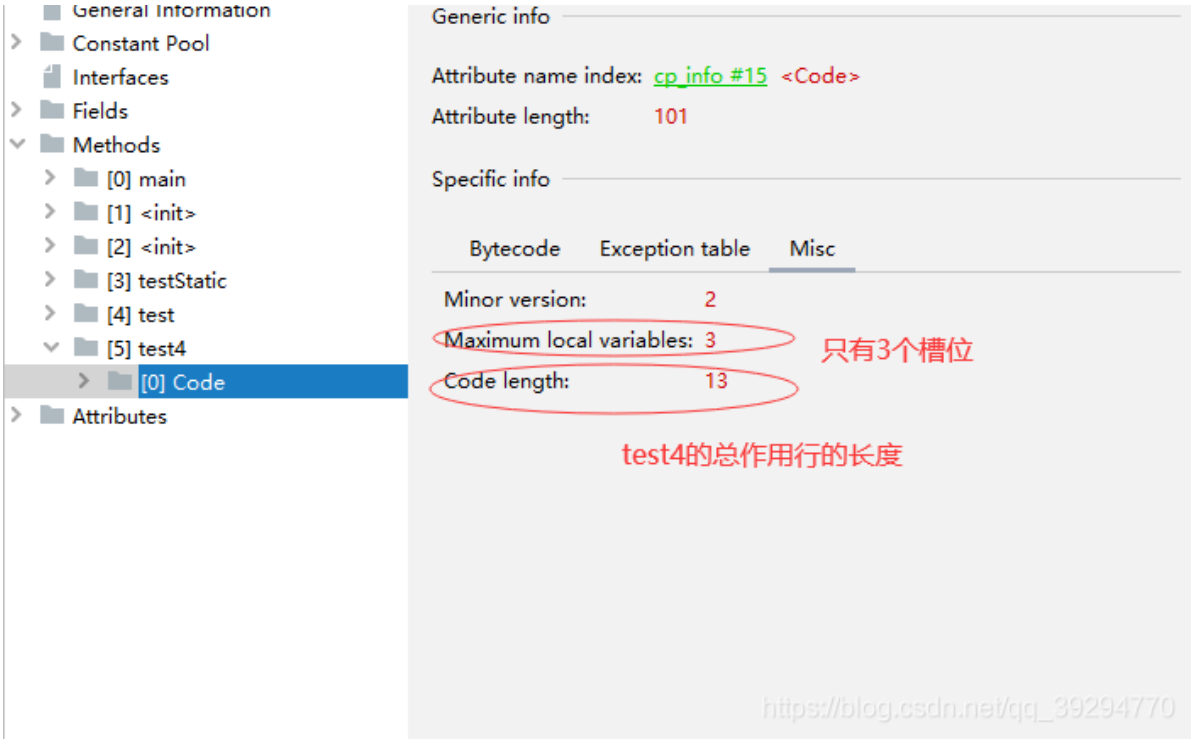
```
package com.lzx;

import java.util.Date;

public class LocalVariaableTest {

    private int count = 0;

    public static void main(String[] args) {
        LocalVariaableTest localVariaableTest = new LocalVariaableTest();
        int num = 10;
    }

    //  构造方法会将this存放在index为0的slot
    public LocalVariaableTest(int count) {
        this.count = count;
    }

    public LocalVariaableTest() {}


    public static void testStatic(){
        LocalVariaableTest localVariaableTest = new LocalVariaableTest();
        Date date = new Date();
        int count = 10;
        System.out.println(count);
        // 静态方法中不能使用this，因为this变量不存在于当前方法的局部变量表中！！！
        //System.out.println(this.count);

    }

    // 实例方法会将this存放在index为0的slot
    public void test(){
        this.count++; //this也是个变量
    }

}
```

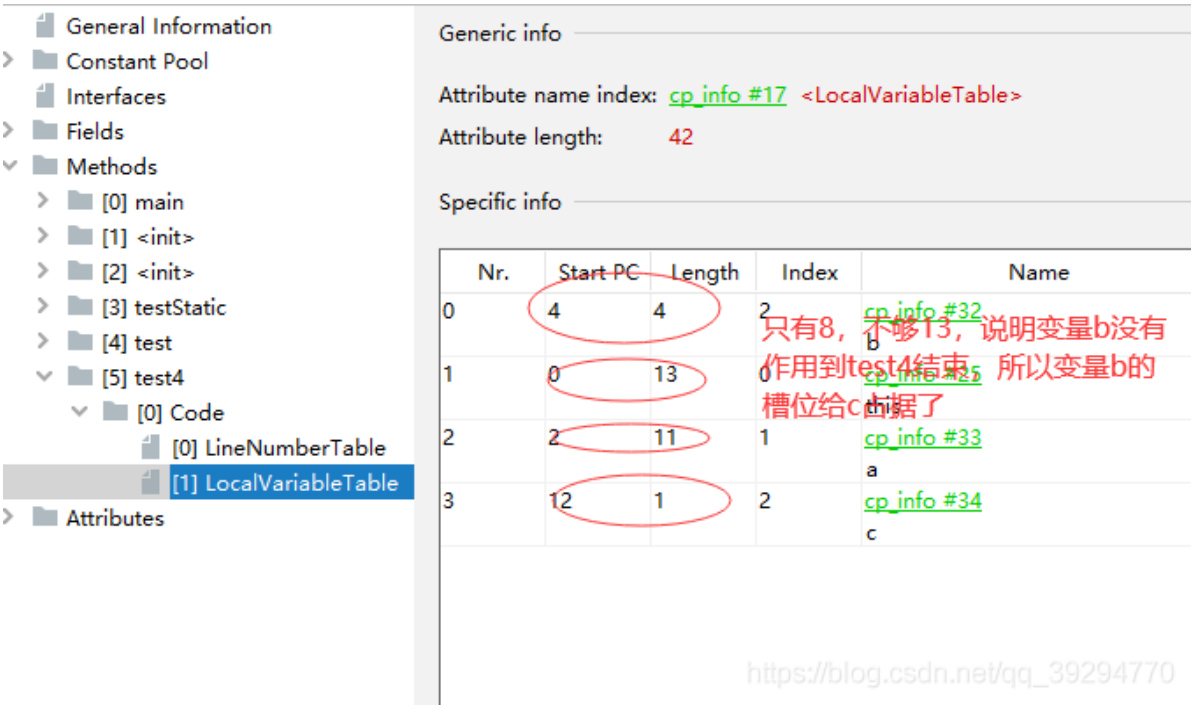




- 栈帧中的局部变量表中的槽位是可以重复利用的

```java
public void test4() {
    int a = 0;
    {
        int b = 0;
        b = a + 1;
    }
    // 变量c使用的是之前已经销毁的变量b占据的slot位置
    int c =  a + 1;
}
```





- **局部变量表中的变量也是重要的垃圾回收根节点，只要被局部变量表中直接或者间接引用的对象都不会被回收**

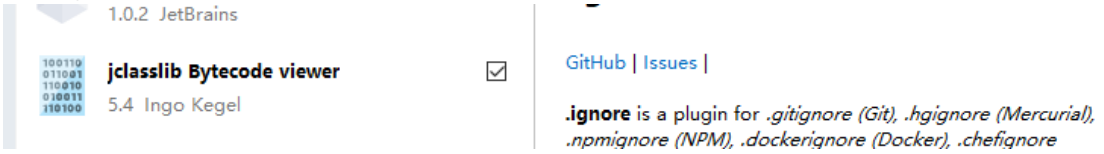
静态变量和局部变量的对比

```
/**
 * 变量的分类：按照数据类型分：①基本数据类型  ②引用类型
 *             按照在类中的声明的位置分：① 成员变量：在使用之前，都经历过默认初始化赋值
 *                                          类变量：linking的prepare阶段：给类变量默认赋值  ---->initial阶段：给类变量显示赋值，即静态代码块赋值
 *                                          实例变量：随着对象的创建，在堆空间分配实例变量空间，并进行默认赋值
 *                                      ② 局部变量：在使用前，必须进行显式赋值！否则，编译不通过
 */

public void test5() {
    int num;
    //System.out.println(num); //错误信息：Variable 'num' might not have been initialized
}
```

4. **操作数栈**

- 栈：可以使用数组或者链表 来实现的，操作数栈也是用数组结构实现的，但是还是栈，所以操作数栈并非采用访问索引的方式来进行数据的方式，而是入栈和出栈
- 每一个独立的栈帧中除了包含局部变量表以外，还包含一个后进先出的操作数栈，也可以称之为表达式栈
- **操作数栈，在方法执行过程中，根据字节码指令，往栈中写入数据或者提取数据，即入栈（push）和出栈（pop）**
- 操作数栈，主要是用于保存计算过程的中间结果，同时作为计算过程中变量的临时存储空间

1. **代码追踪**
   1.在生成字节码文件（.class）目录下执行javap -v xxx.class 查看
   2.安装jclass插件查看



2. **栈顶缓存技术**

- 由于操作数是存储在内存中的，因此频繁地执行内存读/写操作必然会影响执行速度。为了解决这个问题，HotSpot JVM设计者们提出了栈顶缓存技术，将栈顶元素全部缓存在物理CPU的寄存器中，以此降低对内存读写次数，提升执行引擎的执行效率

7. **动态链接**

- 每一个栈帧内部都包含一个指向运行时常量池（方法区中）中该栈帧所属方法的引用。包含这个引用的目的就是为了支持当前方法的代码可以实现动态链接。动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用。

```
package com.lzx;

public class DynamicLinkTest {

    int num = 10;

    public void methodA(){
        System.out.println("methodA...");
    }
```

```
    public void methodB(){
        System.out.println("methodB...");
        methodA();
        num++;
    }

}
```

```
Constant pool:
   #1 = Methodref          #9.#23         // java/lang/Object."<init>":()V
   #2 = Fieldref           #8.#24         // com/lzx/DynamicLinkTest.num:I
   #3 = Fieldref           #25.#26        // java/lang/System.out:Ljava/io/PrintStream;
   #4 = String             #27            // methodA...
   #5 = Methodref          #28.#29        // java/io/PrintStream.println:(Ljava/lang/Str
   #6 = String             #30            // methodB...
   #7 = Methodref          #8.#31         // com/lzx/DynamicLinkTest.methodA:()V
   #8 = Class              #32            // com/lzx/DynamicLinkTest
   #9 = Class              #33            // java/lang/Object
  #10 = Utf8               num
  #11 = Utf8               I
  #12 = Utf8               <init>
  #13 = Utf8               ()V
  #14 = Utf8               Code
  #15 = Utf8               LineNumberTable
  #16 = Utf8               LocalVariableTable
  #17 = Utf8               this
  #18 = Utf8               Lcom/lzx/DynamicLinkTest;
  #19 = Utf8               methodA
  #20 = Utf8               methodB
  #21 = Utf8               SourceFile
```

方法区中的运行时常量池

这个是栈帧所属方法的引用，在每个栈帧内部都会包含这个引用，所以调用方法时就是栈帧中的这个动态链接将这些符号引用转换成调用方法的直接引用，从而实现动态调用

https://blog.csdn.net/qq_39294770

```
      Start  Length  Slot  Name   Signature
          0      11     0  this   Lcom/lzx/DynamicLinkTest;

public void methodA();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream
       3: ldc           #4                  // String methodA...
       5: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/
       8: return
    LineNumberTable:
      line 8: 0
      line 9: 8
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  this   Lcom/lzx/DynamicLinkTest;

public void methodB();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=3, locals=1, args_size=1
       0: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream
```

这个是栈帧中的符号引用，对应上面运行时常量池中的直接引用，这个是通过动态链接来转换的

8. **方法的调用**

- 在JVM中，当一个字节码文件被装载进JVM内部时，如果被调用的目标方法在编译期就可以知道，且运行期保持不变，这种情况下将调用方法的符号转换成直接引用的过程称为静态链接。如果被调用的方法在编译期无法确定下来，只能够在程序运行期将调用方法的符号转换为直接引用的过程称为动态链接。

9. **方法返回地址**

- 存放调用该方法的pc寄存器的值
- 正常完成出口和异常完成出口的区别在于：通过异常完成出口退出不会给他的上层调用者产生任何的返回值

栈的面试题

- 举例栈溢出的情况？（StackOverflowError）
  1.当栈空间一个个去加载栈帧不足的时候
  2.通过-Xss设置栈的大小：OOM
- 调整栈大小，就能保证不出现溢出吗？
  不能，当我们调整栈的大小的时候，栈的大小越大，递归时的深度就可以更深，同样的如果递归的深度过大或者递归时重复调用时就会导致栈空间不足
- 分配的栈内存越大越好吗？
  在一定的情况下来讲，栈内存越大，在单位时间内发生OOM的概率就越低，但是栈内存越大那每一个线程用的栈内存就多，那线程数就会相应减少
- 垃圾回收是否会涉及到虚拟机栈？
  不会的，程序计数器时不会Error的，也不会GC，虚拟机栈会Error，不会GC，因为栈的话直接出栈就行了，不用显式的回收
- 方法中定义的局部变量是否线程安全？
  具体问题具体分析，如果局部变量是内部产生且方法内部就消亡的，那就是线程安全的，如果不是就会可能存在线程安全问题（逃逸分析）

```
package com.lzx;

/**
 * 何为线程安全？
 *     如果只有一个线程才可以操作此数据，则必然是线程安全的
 *     如果有多个线程操作此数据，则此数据是共享数据，如果不考虑同步机制，会存在线程安全问题
 */

public class StringBuilderTest {


    //s1的声明是线程安全的
    public static void method1(){
        //StringBuilder:线程不安全1
        StringBuilder s1 = new StringBuilder();
        s1.append("a");
        s1.append("b");
    }

    // stringBuilder的操作过程：是线程不安全的，因为method2可能被多个线程调用，但是method2没有做任何处理（严格上来说stringBuilder不算局部变量，只是个形参）
    public static void method2(StringBuilder stringBuilder){
        //StringBuilder:线程不安全1
        stringBuilder.append("a");
        stringBuilder.append("b");
    }
```

```
    //s1的声明是线程不安全的，当把s1返回出去时被多个线程操作时就会出现线程不安全问题
    public static StringBuilder method3(){
        StringBuilder s1 = new StringBuilder();
        s1.append("a");
        s1.append("b");
        return s1;
    }

    //s1的声明是线程安全的，因为s1返回出去前死了，所以这里s1是安全的
    public static String method4(){
        StringBuilder s1 = new StringBuilder();
        s1.append("a");
        s1.append("b");
        return s1.toString();
    }

    public static void main(String[] args) {
        StringBuilder s = new StringBuilder();
        new Thread(() -> {
            s.append("a");
            s.append("b");
        }).start();
        method2(s);
    }
}
```

3、堆

1. **堆的核心概述**

- 一个JVM实例只有一个堆内存，并在JVM启动的时候即被创建，其空间大小也就确定了（堆空间是可以被调节的）
- 所有的线程共享Java堆，在这里还可以划分线程私有的缓冲区（ **TLAB**）
- 所有（“几乎”）的对象实例以及数组都应当在运行时分配在堆上
- 在方法结束后，堆中的对象不会马上被移除，仅仅在垃圾收集的时候才会被移除
- 现代垃圾收集器大部分都是基于分代收集理论设计，堆空间细分为：（也是JDK8和之前JDK版本的最大区别）
  Java7及之前堆内存逻辑上分为三部分：新生区+养老区+永久区
  Java8及之后堆内存逻辑上分为三部分：新生区+养老区+元空间
  约定：新生区=新生代=年轻代 养老区=老年区=老年代 永久区=永久代

2. **设置堆内存大小与OOM**

- “-Xms”用于表示堆区（新生区+养老区）的起始内存，等价于-XX：InitialHeapSize，默认是物理电脑内存大小 / 64
- “-Xmx”用于表示堆区（新生区+养老区）的最大内存，等价于-XX：MaxHeapSize，默认是物理电脑内存大小 / 4
- 一旦堆区中的内存大小超过“-Xmx”，将会抛出OutOfMemoryError异常

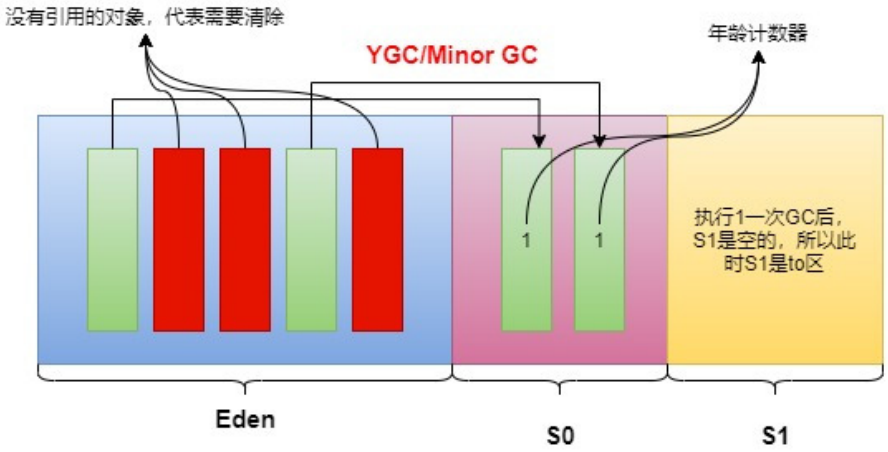
3. **年轻代与老年代**

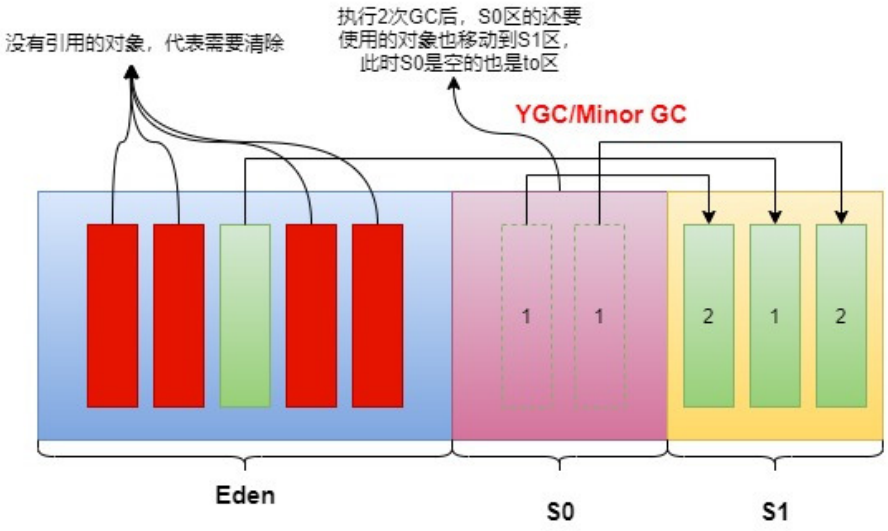
- 年轻代可以划分为Eden空间，Survivor0空间和Survivor1空间（有时也叫from区，to区）
- 配置新生代和老年代在堆结构的占比

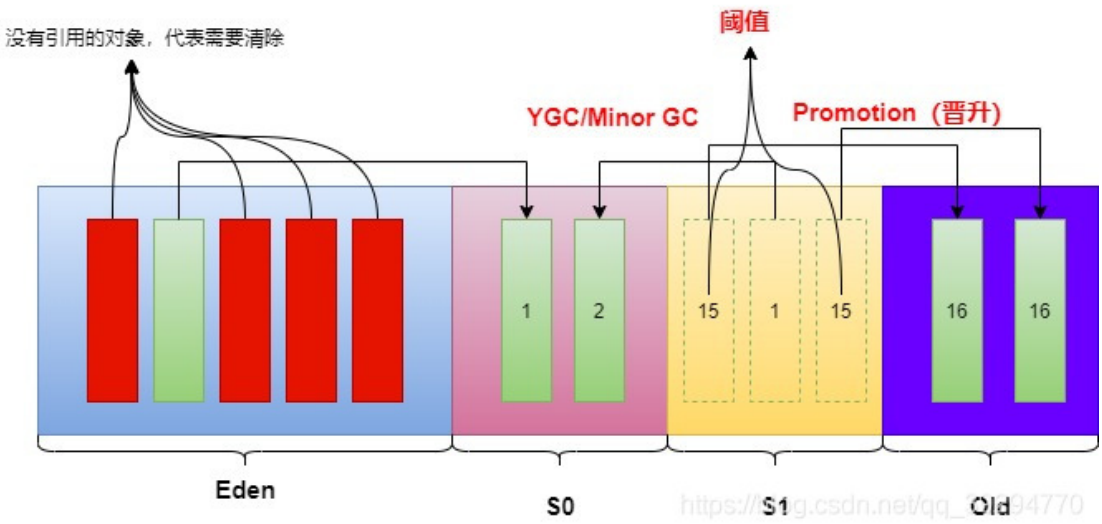
```
默认-XX:NewRatio=2,表示新生代占1，老年代占2，新生代占整个堆的1/3

可以修改-XX:NewRatio=4，表示新生代占1，老年代占4，新生代占整个堆的1/5

-XX:SurvivorRatio=8,设置新生代中Eden区和survivor区的比例，默认是8：1：1，但是实际是6：1：1

-XX:-UseAdaptiveSizePolicy，关闭自适应的内存分配策略（暂时用不到）-代表关闭

-Xmn：设置新生代的空间大小（和-XX:NewRatio冲突下以 -Xmn为主）
```

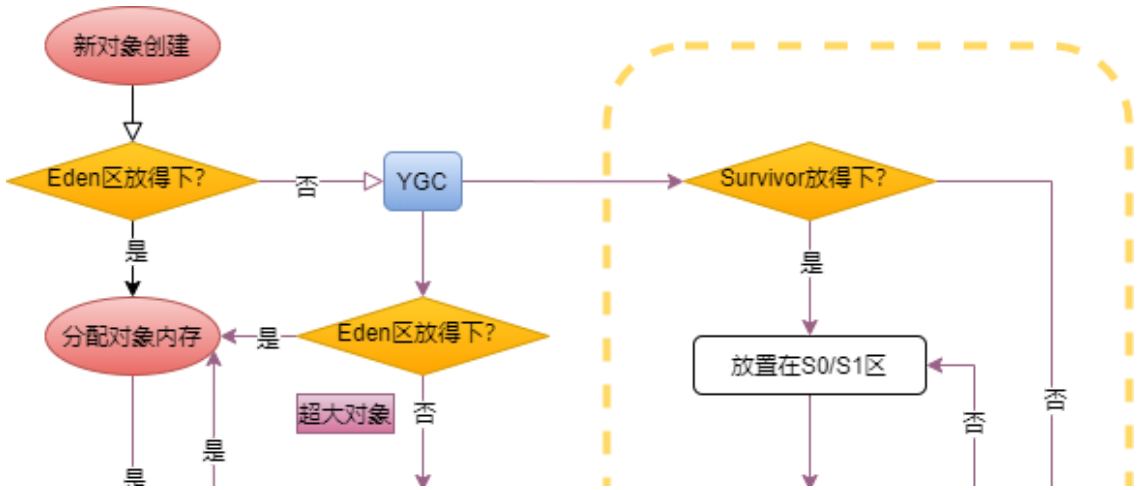
- 堆中新生代和老年代的占比是1：2，可以通过-Xmn：设置新生代的空间大小；新生代中Eden和Survivor0空间和Survivor1空间的占比实际是6：1：1

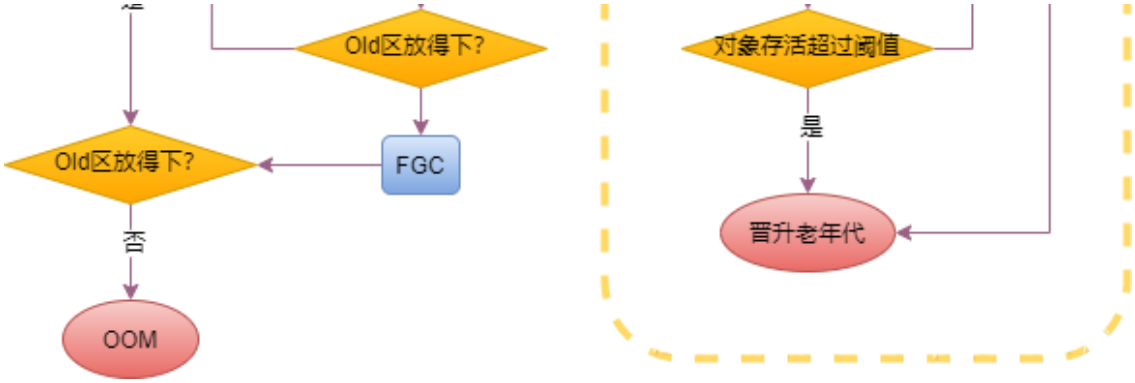
4. **图解对象分配过程**


没有引用的对象，代表需要清除
YGC/Minor GC
年龄计数器
1
1
执行1一次GC后，
S1是空的，所以此
时S1是to区
Eden
S0
S1


没有引用的对象，代表需要清除
执行2次GC后，S0区的还要
使用的对象也移动到S1区，
此时S0是空的也是to区
YGC/Minor GC
1
1
2
1
2
Eden
S0
S1


没有引用的对象，代表需要清除
阈值
YGC/Minor GC
Promotion（晋升）
1
2
15
1
15
16
16
Eden
S0
S1
Old


新对象创建
Eden区放得下?
否
YGC
Survivor放得下?
是
是
分配对象内存
是
Eden区放得下?
放置在S0/S1区
超大对象
否
否
否
是
是



- 进入老年代的对象可能发生的情况：
  - 大对象就直接进入old区了
  - 在Survivor区的对象超过了阈值，从Survivor区晋升到老年代
  - 在Eden区进入Survivor区的对象，在Survivor区满50%（动态年龄判断）的时候，Eden区的对象会直接进入old区

5. **Minor GC 和Major GC 和 Full GC**

- GC按照回收区域分为两大种类型：一种是 **部分收集**（Partial GC），一种是 **整堆收集**（Full GC）
  1.部分收集：不是完整收集整个Java堆的垃圾收集。其中分为：
  新生代收集（Minor GC / Young GC）：只是新生代的垃圾收集
  老年代收集（Major GC / Old GC）：只是老年代的收集。 注意很多时候Major GC 会和Full GC混淆使用，需要具体分辨是老年代回收还是整堆回收
  2.整堆收集（Full GC）：收集整个Java堆和方法区的垃圾回收。
- Minor GC 只在Eden区满的时候触发，Survivor区满并不会触发Minor GC，但是并不是说Survivor区不会被垃圾回收，而是说在Eden区满时触发Minor GC然后Eden区和Survivor区一起被垃圾回收，可以说Survivor区时被动垃圾回收的
- 在老年代空间不足的时候，会先尝试触发 **Minor GC**，如果之后空间还不足，就会执行 **Major GC**。
- Full GC触发机制
  （1）调用System.gc()时，系统建议执行Full GC，但是不必然执行
  （2）老年代不足的时候
  （3）方法区不足的时候
  （4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存
  （5）由Eden区，survivor0（From Space）区先survivor1（To Space）区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小
  说明：Full GC是开发或者调优中尽量要避免的，这样暂停时间会短一些

6. **为对象分配内存：TLAB**

- **TLAB（Thread Local Allocation Buffer）是指JVM为每个线程分配了一个私有缓存区域，它包含在Eden空间内。**
- 多线程同时分配内存时，使用TLAB可以避免一系列的非线程安全问题，同时还能提升内存分配的吞吐量，因此可以将这种内存分配方式称为快速分配策略
- 尽管不是所有的对象实例都可以在TLAB中成功分配内存（默认情况下，TLAB的空间内存非常小，仅占整个Eden空间的1%），但JVM确实是将TLAB作为内存分配的首选。
- 在程序中，可以通过“ **-XX:UseTLAB**”设置是否开启TLAB空间（默认开启），通过“-XX:TLABWasteTargetPercent”设置TLAB空间所占Eden空间的百分比大小。
- 一旦对象空间在TLAB空间分配内存失败，JVM就会尝试通过使用加锁机制确保数据操作原子性，从而直接在Eden空间分配内存

7. **堆空间的参数设置**

```
-XX:+PrintFlagsInitial ：查看所有参数的默认初始值
```

-XX:+PrintFlagsFinal ：查看所有参数的最终值（可能会存在修改，不再是初始值）

-Xms ：初始堆空间内存（默认为物理内存的1/64）

-Xmx ：最大堆空间内存（默认为物理内存的1/4）

-Xmm ：设置新生代的大小（初始值和最大值）

-XX:NewRatio：配置新生代和老年代在堆结构的占比

-XX:SurvivorRatio ：设置新生代中Eden和S0/S1空间的比例

-XX:MaxTenuringThreshold：设置新生代垃圾的最大年龄

-XX:+PrintGCDetails ：输出详细的GC处理日志

-XX:+PrintGC ：打印简要的GC信息

-verbose:gc ：打印简要的GC信息

-XX:HandlePromotionFailure：是否设置空间分配担保

8. **堆不是分配对象的唯一选择**

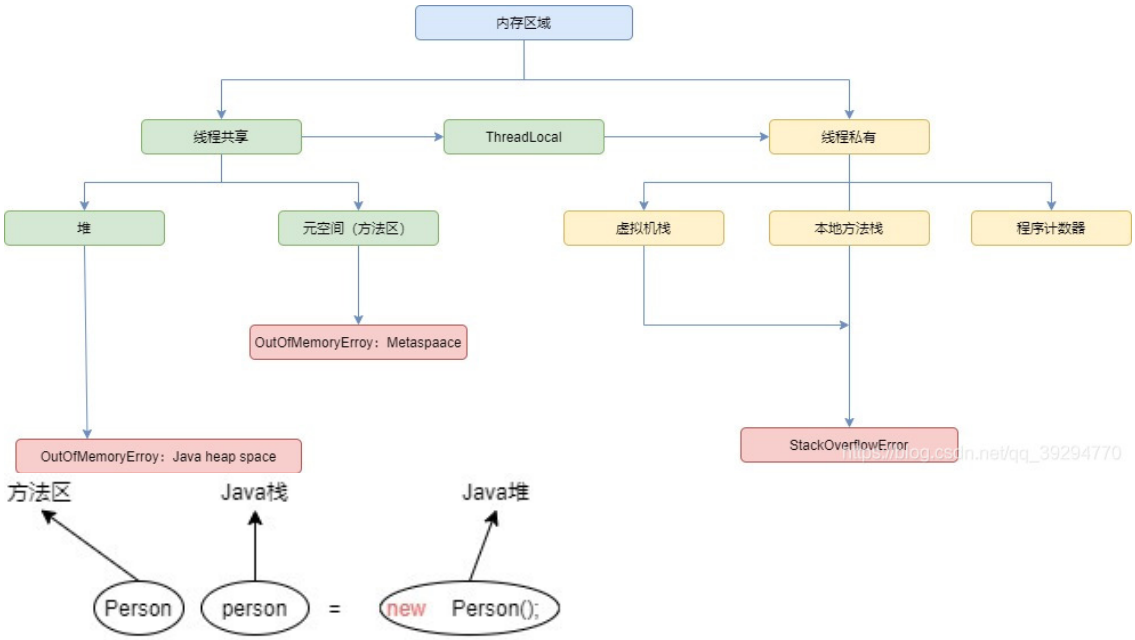
- 对象还可以分配到栈，需要使用逃逸分析手段
- 逃逸分析的基本行为就是分析对象动态作用域：
  （1）当一个对象在方法中被定义后，对象只在方法内部使用，则认为没有发生逃逸（能够在栈上分配，线程安全）
  （2）当一个对象在方法中被定义后，它被外部方法使用，则认为发生逃逸（不能够在栈上分配）。例如作为调用参数传递到其他地方中

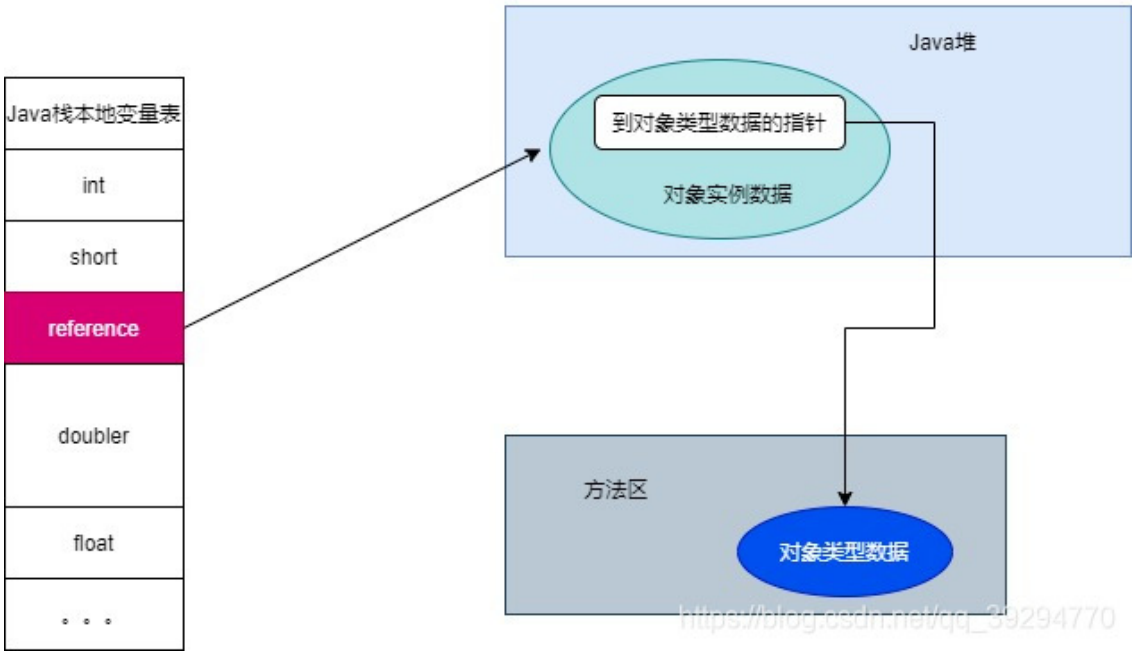
```
/**
 * stringBuffer发生了逃逸，快速判断是否发生了逃逸分析，就看new出来的对象是否可能在方法外被调用
 * @param s1
 * @param s2
 * @return
 */
public static StringBuffer createStringBuffer (String s1, String s2) {
    StringBuffer stringBuffer = new StringBuffer();
    stringBuffer.append(s1);
    stringBuffer.append(s2);
    return stringBuffer;
}

/**
 * stringBuffer没有发生逃逸，传递出去的是一个字符串
 * @param s1
 * @param s2
 * @return
 */
public static String createStringBuffer2 (String s1, String s2) {
    StringBuffer stringBuffer = new StringBuffer();
    stringBuffer.append(s1);
    stringBuffer.append(s2);
    return stringBuffer.toString();
```

```
}
```

4、方法区

1. **栈，堆，方法区的交互关系**





2. **方法区（元空间）的理解**

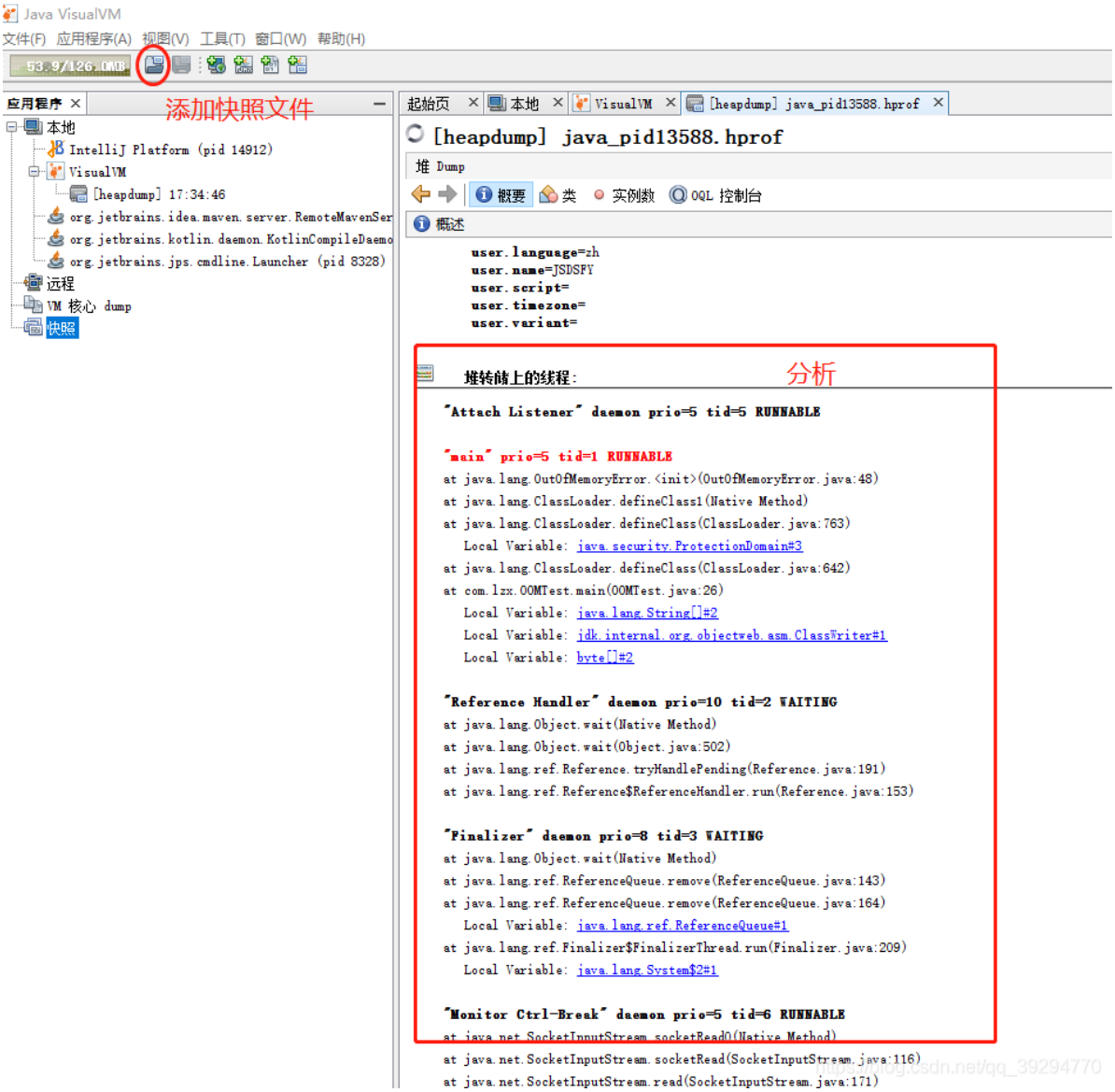
- 方法区看作是一块独立于Java堆的内存空间，而且是使用**本地内存**
- 方法区与java堆一样，是各个线程共享的内存区域，并且它的实际物理内存和Java堆一样是可以不连续的
- 方法区的大小可以是固定的，也可以扩展，方法区的大小决定了系统可以保存多少个类，如果系统定义太多类，导致方法区溢出，方法区异常：**java.lang.OutOfMemoryError:PermGen space**（JDK1.7之前），**java.lang.OutOfMemoryError:Metaspace**（JDK1.7之后）
- 关闭JVM就会释放这个区域的内存

3. **设置方法区大小与OOM**

- JDK7及以前

    - 通过-XX:PermSize来设置永久代初始分配空间。默认值是**20.75M**
    - **-XX:MaxPermSize**来设定永久代最大可分配空间。32位机默认是**64M**，64位机默认是**82M**
- JDK8及以后
    - **-XX:MetaspaceSize**来设置永久代初始分配空间。默认值是21M
    - **-XX:MaxMetaspaceSize**来设定永久代最大可分配空间。默认是-1，即没有限制
- OOM异常解决：
    - 首先通过对堆转储快照进行分析，确认确认内存中的对象是否是必要的，也就是分清楚是出现了内存泄漏还是内存溢出。如果是内存泄漏，可以通过JVM工具查看泄漏对象到GC Roots的引用链找到泄漏对象是通过怎样的路径于GC Roots关联导致垃圾收集器无法自动回收对象的；如果不存在内存泄漏，代表内存中的对象还存活着，就可以检查JVM的堆参数，看是否可以调大
    - dump堆转储快照文件的方式：
        - ①配置JMM参数OOM异常时打印堆转储快照文件
          -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=${目录bai}
        - ②使用命令**JPS和jmap -dump:file=a pid**

```
package com.lzx;

import jdk.internal.org.objectweb.asm.ClassWriter;
import jdk.internal.org.objectweb.asm.Opcodes;

import java.util.concurrent.TimeUnit;

/**
 * JDK8
 * 设置方法区大小：-XX:MetaspaceSize=10m -XX:MaxMetaspaceSize=10m
 * 打印OOMError文件： -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=${目录bai}
 */
public class OOMTest extends ClassLoader{
    public static void main(String[] args) throws InterruptedException {
        int j = 0;
        try{
            OOMTest oomTest = new OOMTest();
            for (int i = 0; i < 10000; i++) {
                // 创建classWriter对象，用于生成类的二进制字节码
                ClassWriter classWriter = new ClassWriter(0);
                //指明版本，修饰符，类名，包名，父类，接口
                classWriter.visit(Opcodes.V1_8,Opcodes.ACC_PUBLIC, "Class" + i,
null, "java/lang/Object", null);
                // 返回byte[]
                byte[] code = classWriter.toByteArray();
                //加载类
                oomTest.defineClass("Class" + i , code, 0, code.length);
                j++;
            }
        }finally {
            System.out.println(j);
        }
    }
}
```



4. **方法区的内部结构**

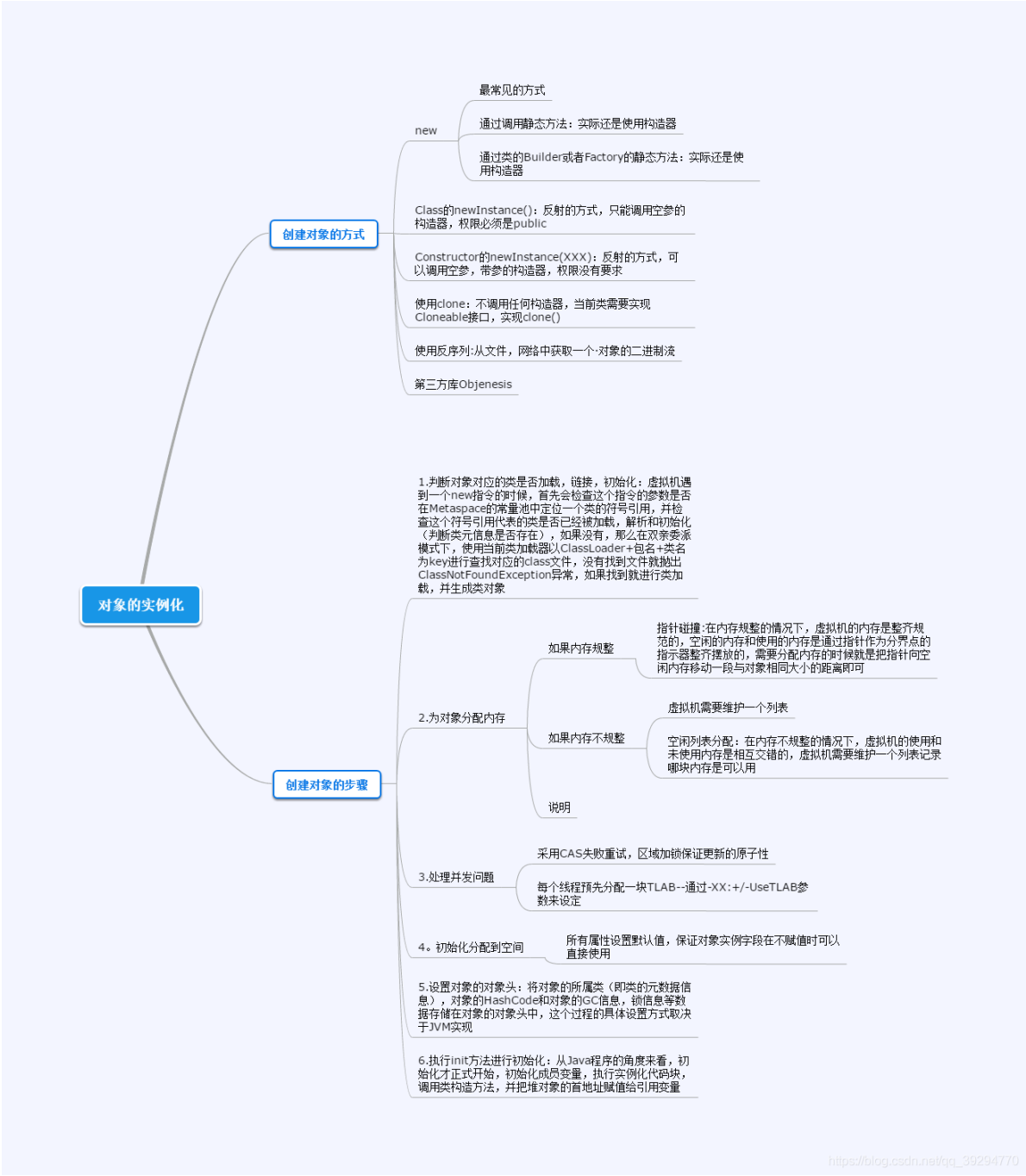
- **方法区里存储着类型信息（类，接口，枚举，注解），方法信息，域信息和运行时常量池，即时编译器编译后的代码缓存**
- **运行时常量池是方法区的一部分也是非常重要的一部份，存放的是Class文件中的常量池表（用于存放编译期生成的各种字面量与符号引用）在类加载后存放到方法区的内容**
- **jdk1.8和之前最大的改变就是没有了永久代，但是字符串常量池（堆中），静态变量的实例对象始终还是和原来的jdk1.7版本一样，仍然保存在堆内**

5. **方法区的垃圾回收**

- **方法区的回收条件是相当苛刻的，回收效果也是不怎么样**
- **方法区的垃圾回收主要是：常量池中废弃的常量和不再使用的类型**

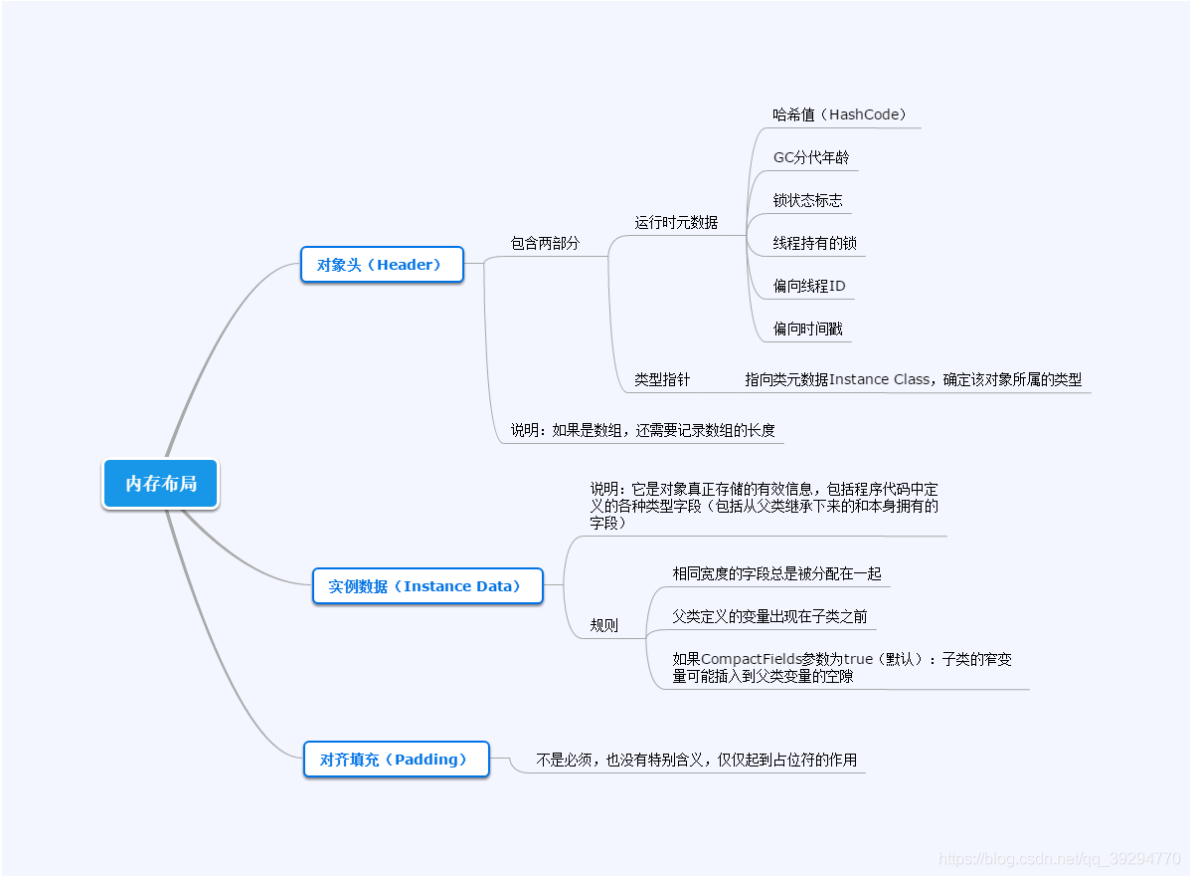
第四章 对象的实例化和内存布局

1、对象的实例化



https://blog.csdn.net/qq_39294770

2、对象的内存布局



第五章 StringTable

1、String的基本特性

- 字符串有两种声明方式：①字面量，②new 的方式
- String在JDK8及以前内部定义了final char[ ] value 用于存储字符串数据。jdk9时改为byte[ ]
- **String代表不可变字符序列（不可变性）：一旦定义好一个字符串之后，如果试图在字符串后面去补一个字符，只能重新去字符串常量池里创建一个新的字符串，这就体现了字符串的不可变性**
- **字符串常量池中是不会存储相同内容的字符串的。String之所以可以实现不可变性就是因为String的String Pool是一个固定大小的HashTable。**
- String Pool默认值大小长度是1009（JDK6，JDK7和JDK8是60013），如果放进String Pool的String非常多，就会造成Hash冲突严重，从而导致链表很长，而链表长了之后会直接造成当调用String.intern时性能会大幅下降
- **通过new String(“XXX”) 的方式创建的字符串会创建两个对象。一个对象是：new关键字在堆空间创建的（new String()）；另一个对象是：字符串常量池中的对象（常量池里的"XXX",常量池里之前没有的情况下创建，有就直接指向）但是返回的是堆空间的对象。**

2、String的内存分配

- **JDK6及以前，字符串常量池时放在永久代，JDK7就把字符串常量池的位置调整到堆内，JDK8之后没有永久代了，改成了元空间，但是字符串常量池还是在堆内**
- **字符串都保存在堆内（不管是在字符串常量池还是堆的其他区域），所有调优应用时仅需要调整堆的大小即可**

3、字符串拼接操作

- 字符串常量与字符串常量的拼接结果在字符串常量池，原理是编译期优化
- 字符串常量池中不会存在相同内容的字符串常量

```
String s1 = "a"+"b"+"c"; // 编译期时就直接给转成了"abc"，而字符串字面量就是存在字符串常量池中的
String s2 = "abc"; //"abc"一定是放在字符串常量池中，并将地址赋给s2
System.out.println(s1 == s2); //  true,因为字符串常量池中不会存在相同内容的字符串常量，所以s1的"abc"和s2的"abc"是同一个
String newString = new String("abc");
System.out.println(s2 == newString);// false,因为newString是在非常字符串量池的堆空间内
```

- 只要其中有一个是字符串变量，结果就在堆中，字符串变量的拼接原理是StringBuilder（只要是拼接，底层都会new StringBuider()），拼接结束后底层会调用StringBuilder的toString(),但是toString()并不会在字符串常量池里创建对象

```
String s3 = "javaEE";
String s4 = "hadoop";

String s5 = "javaEEhadoop";
String s6 = "javaEE" + "hadoop"; //编译期优化
// 如果拼接字符串的前后出现了变量，则相当于在堆空间中new String(),具体内容就为拼接的结果
String s7 = s3 + "hadoop";
String s8 = "javaEE" + s4;
String s9 = s1 + s2;

System.out.println(s5 == s6); // true s5和s6的值都在字符串常量池
System.out.println(s5 == s7); // false s5值在字符串常量池，s7由于拼接了变量s3，所以s7在堆中
System.out.println(s5 == s8); // false s5值在字符串常量池，s8由于拼接了变量s4，所以s8在堆中
System.out.println(s5 == s9); // false s5值在字符串常量池，s9是由变量s3和变量s4拼接，所以s9在堆中
System.out.println(s7 == s8); // false 虽然s7和s8都在堆中，但是是在不同的引用空间，即地址是不一样的
System.out.println(s7 == s9); // false 虽然s7和s9都在堆中，但是是在不同的引用空间，即地址是不一样的
System.out.println(s8 == s9); // false 虽然s8和s9都在堆中，但是是在不同的引用空间，即地址是不一样的

final String s10 = "javaEE";
final String s11 = "hadoop";
String s12 = s10 + s11;
/**
 * 由于s10和s11使用了final修饰，即是常量引用，所以s5和s12都是在字符串常量池里，
 * 即字符串拼接也不一定使用StringBuilder,如果拼接符号左右都是字符串常量或者常量引用，则仍使用编译期优化
 */
System.out.println(s5 == s12); // true
```

- 如果拼接的结果是调用intern()方法，则主动将常量池中还没有的字符串对象放入池中，并返回此对象地址

```
String s4 = "hadoop";
String s8 = "javaEE" + s4;
String s13 = s8.intern();
System.out.println(s13 == s8); // true
```

4、intern()解析

```
String s = new String("1"); //造了两个对象，new String()和字符串常量池里的"1"，返回的是堆空间的对象
s.intern(); //此时字符串常量池中已经存在"1"
String s2 = "1"; // 返回的是字符串常量池里的"1"
System.out.println(s == s2); // jdk6: false , jdk7/8: false

/**
 * s3这个生成的对象包括：
 * 对象1：new StringBuilder() 只要是拼接操作都会生成一个new StringBuilder()对象
 * 对象2：new String("1")
 * 对象3：字符串常量池中的"1"
 * 对象4：和对象2在不同非字符串常量池的堆空间的new String("1")
 * 因为字符串常量池已经存在"1"，所以拼接后的new String("1")就没有再在字符串常量池中创建"1"了，而是直接返回已经存在的"1"
 * 对象5: 拼接的结果：new String("11"),但是不会在字符串常量池中创造"11"
 */
String s3 = new String("1") + new String("1");
/**
 * 字符串常量池创造"11"，
 * JDK6：创建一个新的对象"11",也就是新的地址；
 * JDK7/8：此时的intern()操作会把堆空间中的对象引用地址复制一份放到常量池里（此时常量池中并没有"11",有就不会复制），而不会再创建一个新的对象"11"
 */
s3.intern();
String s4 = "11"; // 返回s3.intern()操作创建的符串常量池"11"
System.out.println(s3 == s4); // jdk6: false , jdk7/8: true

/**
 * 把 s3.intern(); 和String s4 = "11";换个位置会怎么样呢？
 */

String s5 = new String("2") + new String("2");
String s6 = "22"; // 在字符串常量池中创建"22"
/**
 * 此时的操作不会把堆空间中的对象引用地址复制一份放到常量池里，而是直接返回s6在常量池中创建的对象
 */
s5.intern();
System.out.println(s5 == s6); // false
String s7 = s5.intern();
System.out.println(s7 == s6); // true
```

- 对于程序中存在大量的字符串，尤其其中存在很多重复的字符串时，使用intern()可以节省内存空间效率

第六章 垃圾回收的相关算法

1、标记阶段：引用计数算法，可达性分析算法

- 标记内存中哪些对象是存活的，哪些是已经死亡的对象的过程就是**垃圾标记阶段**
- **判断对象是否存活一般有两种方式：引用计数算法和可达性分析算法**
- 引用计数算法（Java不使用，Python使用）
  - 对每一个对象保存一个整型的引用计数器属性，用于记录对象被引用的情况
  - 优点：实现简单，垃圾对象便于辨识；判断效率高，回收没有延迟性

    - 缺点：它需要单独的字段存储计数器，增加了存储空间的开销；每次赋值都要更新计数器，增加了时间开销；**无法处理循环引用的情况**
- **可达性分析算法（Java使用）**
    - 以根对象集合为起始点，按照从上至下的方式搜索被根对象集合所连接的目标对象是否可达
    - 优点：同样具备实现简单和执行效率高的特点，**而且有效解决引用计数算法中循环引用的问题，防止内存泄漏的发生**
    - GC Roots包括几类元素：①虚拟机栈中引用的对象；②本地方法栈内JNI（本地方法）引用的对象；③方法区中类静态属性引用的对象；④方法区中常量引用对象；⑤所有被同步锁synchronization持有的对象；⑥Java虚拟机内部的引用 （**小技巧：由于Root采用栈方式存放变量和指针，所以如果一个指针，它保存了堆内存里面的对象，但是自己本身又不存放在堆内存里面，那它就是一个Root**）

2、对象的finalization机制

- Java语言提供了对象终止（finalization）机制来允许开发人员提供对象被销毁之前的自定义处理逻辑（finalize()方法）
- finalize()方法允许重写，用于在对象被回收时进行资源释放

3、增量收集算法，分区算法

- 增量收集算法和分区算法都是为了解决问题：垃圾回收时间过长而导致系统长时间的停顿
- 增量收集算法
    - 解决思路：让垃圾回收线程和应用线程交替执行，每次，垃圾回收线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。
    - 解决思路支撑算法：增量收集算法通过对线程间冲突的妥善处理，允许垃圾收集线程以分阶段的方式完成标记，清除或者复制工作。
    - 增量收集算法缺点：线程切换和上下文切换的消耗，造成系统吞吐量的下降
- 分区算法
    - 分区算法将整个堆空间划分为连续的不同小区间，每个小区间都独立使用，独立回收。

第七章 垃圾回收的相关概念

1、System.gc

```
package com.lzx;

public class SystemGCTest {

    public static void main(String[] args) {

        new SystemGCTest();
        System.gc(); //提醒jvm进行垃圾回收，但是不确定是否马上执行gc
        System.runFinalization(); // 强制调用使用引用对象的finalize()方法

    }

    @Override
    protected void finalize() throws Throwable {
        super.finalize();
        System.out.println("SystemGCTest 重写了finalize()");
    }
}
```

2、内存溢出（OOM）

- Java虚拟机的堆内存设置不够
- 代码中创建了大量大对象，并且长时间不能被垃圾收集器收集（存在引用）

3、内存泄漏（Memory Leak）

- 只有对象不会再被程序用到，但是GC又不能回收他们的情况，才叫内存泄漏
- 内存泄漏并不会立刻引起程序崩溃，但是一旦发生内存泄漏，程序中的可用内存就会被逐步蚕食，直至内存耗尽，最终出现OOM，导致程序崩溃
- 内存泄漏的场景：①单例模式：单例的生命周期和应用程序是一样长的，所以在单例程序中，如果持有对外部对象的引用，那么这个对象是无法被回收的，否则会导致内存泄漏的产生；②一些提供close的资源未关闭导致内存泄漏：数据库连接，网络连接

4、引用

- 强引用(StrongReference)
  - 普遍存在的引用赋值，即类似“Object obj = new Object()”这种引用关系。无论任何情况下，只要强引用关系还在，垃圾收集器就永远不会回收掉被引用的对象。
- 软引用(SoftReference)
  - 在系统将要发生内存溢出之前，将会把这些对象列入回收范围之中进行二次回收。如果这次回收之后还是没有足够的内存，才会抛出内存溢出异常。
    软引用通常用来实现内存敏感的缓存。比如：高速缓存

```
package com.lzx.reference;

import java.lang.ref.SoftReference;

/**
 * -Xms10m -Xmx10m
 */

public class SoftReferenceTest {


    public static void main(String[] args) {
        //创建对象，建立软引用
        SoftReference<User> userSoftReference = new SoftReference(new
User(1,"test"));
        //从软引用中重新获得强引用对象
        System.out.println(userSoftReference.get().toString()); // User{age=1,
name='test'}

        System.gc();
        System.out.println("After GC: " + userSoftReference.get().toString());
//After GC: User{age=1, name='test'}

        try {
            // 让系统认为内存资源紧张
            byte[] bytes = new byte[1024 * 1024 * 7];
        } catch (Throwable e) {
            e.printStackTrace(); //java.lang.OutOfMemoryError: Java heap space
        } finally {
            System.out.println("资源紧张后: " + userSoftReference.get()); //资源紧张
后: null
```

```
        }
    }
}


class User {
    private int age;
    private String name;
    public User(int age, String name) {
        this.age = age;
        this.name = name;
    }

    @Override
    public String toString() {
        return "User{" +
                "age=" + age +
                ", name='" + name + '\'' +
                '}';
    }
}
```

- 弱引用(WeakReference)
    - 只被弱引用关联的对象只能生存到下一次垃圾收集之前。当垃圾收集器工作时，无论内存空间是否足够，都会回收掉被弱引用关联的对象
    - 软引用和弱引用都适合保存那些可有可无的缓存数据

```
Object obj = new Object(); //声明强引用
WeakReference<Object> weakReference = new WeakReference<>(obj); //实现弱引用
obj = null; // 销毁强引用
```

- 虚引用(PhantomReference)
    - 一个对象无法通过虚引用来获得一个对象的实例，为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知
    - 由于虚引用可以跟踪对象的回收时间，因此，也可以将一些资源释放操作放置在虚引用中执行和记录