# Java 应用程序设计
## Java 内存分配机制

KevinW@OUC
wangxiaodong@ouc.edu.cn

中国海洋大学

September 21, 2017


# 参考文献

1.

# 大纲

Java 内存模型

Java 程序内存运行分析

常量池技术

Java 内存建议

# 接下来…

Java 内存模型

Java 程序内存运行分析

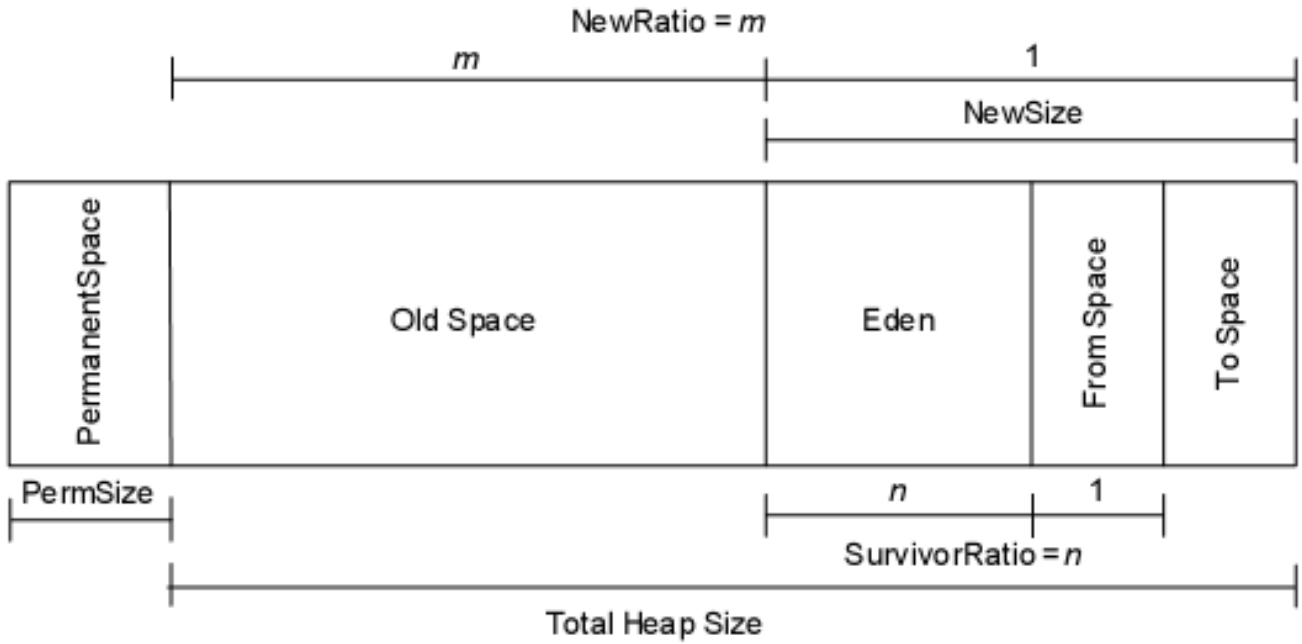
常量池技术

Java 内存建议

# 以 JVM 的视角

Java 程序运行在 JVM（Java Virtual Machine，Java 虚拟机）上，可以把 JVM 理解成 Java 程序和操作系统之间的桥梁，JVM 实现了 Java 的平台无关性。JVM 是内存分配原理的前提。

# JVM 内存模型




从大方面讲，JVM 的内存模型分为两块：**永久区内存**（Permanent space）**和堆内存**（Heap space）。栈内存（stack space）一般不归在 JVM 内存模型中，因为栈内存属于线程级别，每个线程都有个独立的栈内存空间。

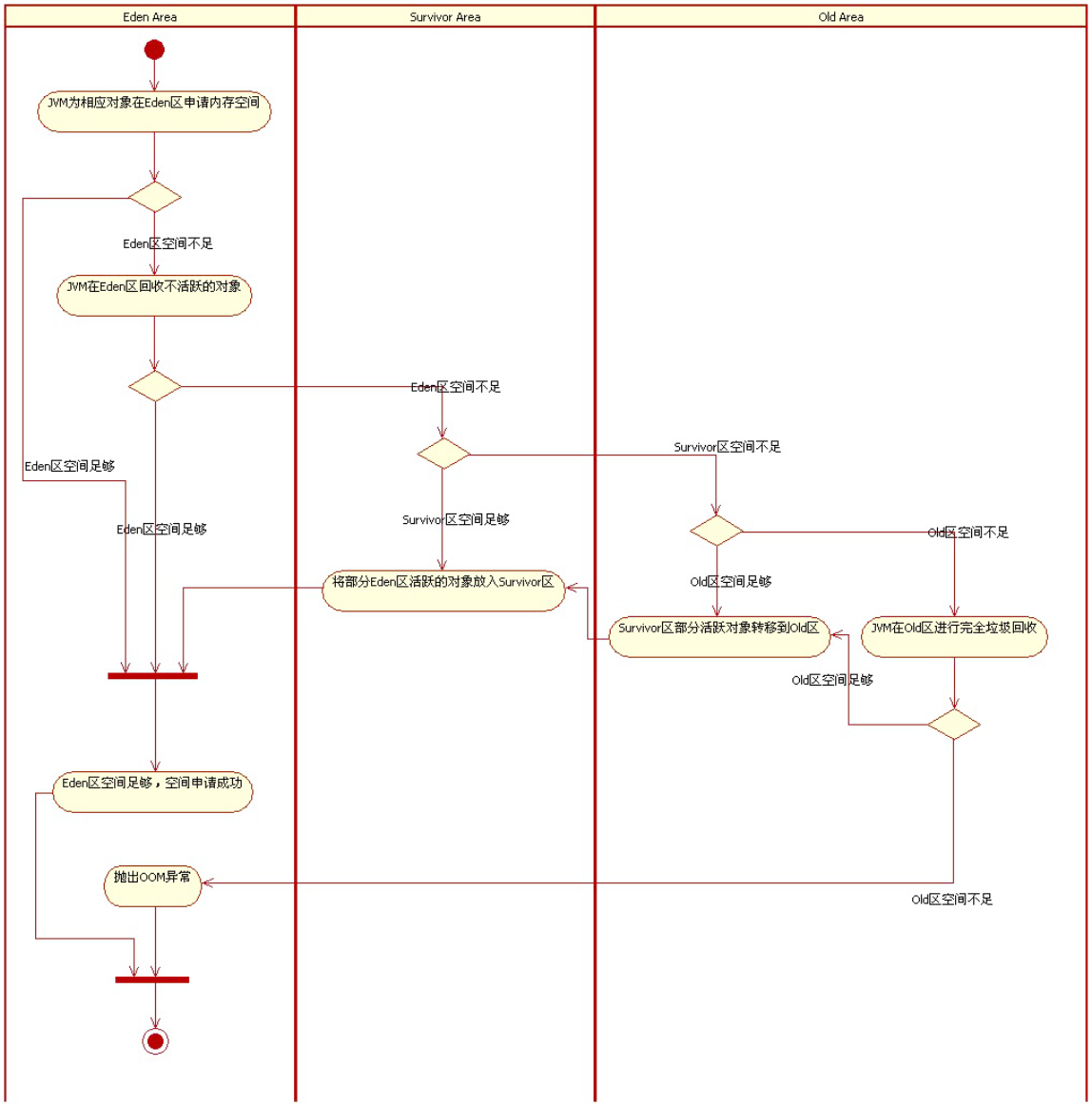
# Java 程序运行过程会涉及的内存区域

- Permanent space 里存放加载的 Class 类级对象如 class 本身、method、field 等。
- Heap space 主要存放对象和数组。

# Java 程序运行过程会涉及的内存区域

- Permanent space 里存放加载的 Class 类级对象如 class 本身、method、field 等。
- Heap space 主要存放对象和数组。
  Heap space 由 Old Generation 和 New Generation 组成，Old Generation 存放生命周期长久的实例对象，而新的对象实例一般放在 New Generation。New Generation 还可以再分为 Eden 区和 Survivor 区，新的对象实例总是首先放在 Eden 区，Survivor 区作为 Eden 区和 Old 区的缓冲，可以向 Old 区转移活动的对象实例。

# JVM 堆内存空间申请流图



# JVM 内存溢出和参数调优[1]

- 常见的 OOM（Out Of Memory）内存溢出异常，就是堆内存空间不足以存放新对象实例时导致。
- 永久区内存溢出相对少见，一般是由于需要加载海量的 Class 数据，超过了非堆内存的容量导致。通常出现在 Web 应用刚刚启动时，因此 Web 应用推荐使用预加载机制，方便在部署时就发现并解决该问题。
- 栈内存也会溢出，但是更加少见。

| | |
|---|---|
| 堆内存优化 | 调整 JVM 启动参数 -Xms -Xmx -XX:newSize -XX:MaxNewSize，如调整初始堆内存和最大对内存 -Xms256M -Xmx512M。或者调整初始 New Generation 的初始内存和最大内存 -XX:newSize=128M -XX:MaxNewSize=128M。 |
| 永久区内存优化 | 调整 PermSize 参数如 -XX:PermSize=256M -XX:MaxPermSize=512M。 |
| 栈内存优化 | 调整每个线程的栈内存容量如 -Xss2048K。 |

[1]关于参数调优需勘误。

# Java 程序运行过程会涉及的内存区域

寄存器 JVM 内部虚拟寄存器，存取速度非常快，程序不可控制。

栈 保存局部变量的值，包括：用来保存基本数据类型的值；保存类的实例，即堆区对象的引用（指针），也可以用来保存加载方法时的帧。（Stack）

堆 用来存放动态产生的数据，如 new 出来的对象[2]。（Heap）

常量池 JVM 为每个已加载的类型维护一个常量池，常量池就是这个类型用到的常量的一个有序集合。包括直接常量（基本类型，String）和对其他类型、方法、字段的符号引用。池中的数据和数组一样通过索引访问，常量池在 Java 程序的动态链接中起了核心作用。（Perm）

代码段 存放从硬盘上读取的源程序代码。（Perm）

数据段 存放 static 定义的静态成员。（Perm）

[2]注意创建出来的对象只包含属于各自的成员变量，并不包括成员方法。因为同一个类的对象拥有各自的成员变量，存储在各自的堆内存中，但是他们共享该类的方法，并不是每创建一个对象就把成员方法复制一次。

# 接下来…

Java 内存模型

Java 程序内存运行分析

常量池技术

Java 内存建议

# 预备知识

1. 一个 Java 文件，只要有 main 入口方法，即可认为这是一个 Java 程序，可以单独编译运行。
2. 无论是普通类型的变量还是引用类型的变量（俗称实例），都可以作为局部变量，他们都可以出现在栈中。只不过普通类型的变量在栈中直接保存它所对应的值，而引用类型的变量保存的是一个指向堆区的指针。通过这个指针，就可以找到这个实例在堆区对应的对象。因此，**普通类型变量只在栈区占用一块内存，而引用类型变量要在栈区和堆区各占一块内存。**

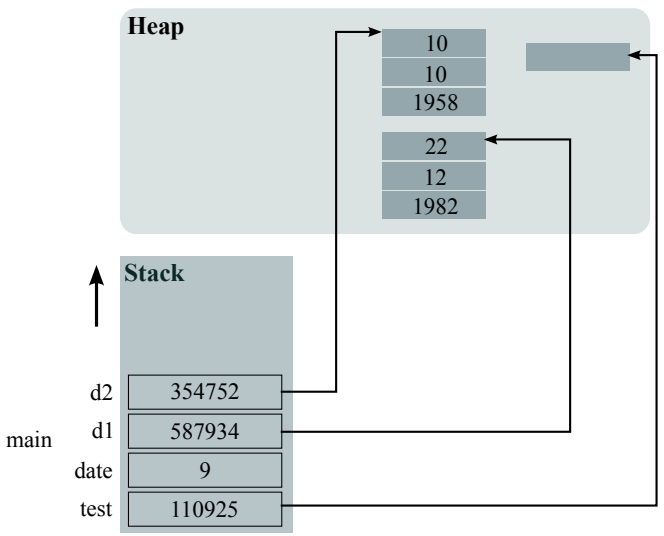
# 所用讲解程序实例

CODE ◗ Test.java

```
1  public class Test {
2    public static void main(String[] args) {
3      Test test = new Test();
4      int data = 9;
5      BirthDate d1 = new BirthDate(22, 12, 1982);
6      BirthDate d2 = new BirthDate(10, 10, 1958);
7      test.m1(data);
8      test.m2(d1);
9      test.m3(d2);
10   }

12   public void m1(int i) {
13     i = 1234;
14   }
15   public void m2(BirthDate b) {
16     b = new BirthDate(15, 6, 2010);
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

# 程序调用过程（一）



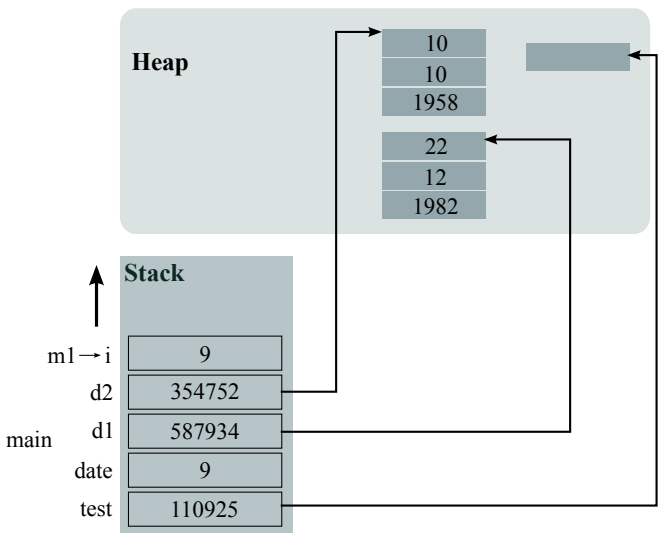

```
1  public class Test {
2    public static void main(String[] args) {
3      Test test = new Test(); //1
4      int data = 9; //2
5      BirthDate d1 = new BirthDate(22, 12, 1982); //3
6      BirthDate d2 = new BirthDate(10, 10, 1958); //4
7      test.m1(date);
8      test.m2(d1);
9      test.m3(d2);
10   }

12   public void m1(int i) {
13     i = 1234;
14   }
15   public void m2(BirthDate b) {
16     b = new BirthDate(15, 6, 2010);
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

# 程序调用过程（一）

- JVM 自动寻找 main 方法，执行第一句代码，创建一个 Test 类的实例，在栈中分配一块内存，存放一个指向堆区对象的指针 110925。
- 创建一个 int 型的变量 date，由于是基本类型，直接在栈中存放 date 对应的值 9。
- 创建两个 BirthDate 类的实例 d1、d2，在栈中分别存放了对应的指针指向各自的对象。它们在实例化时调用了有参数的构造方法，因此对象中有自定义初始值。

# 程序调用过程（二）



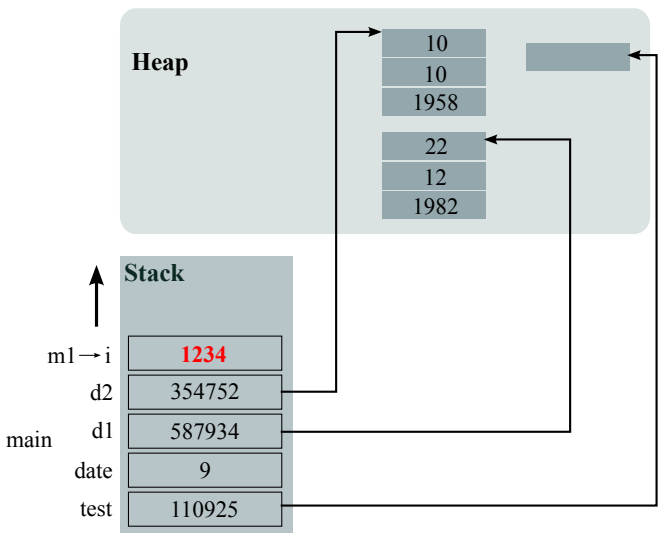

```
1  public class Test {
2    public static void main(String[] args) {
3      Test test = new Test();
4      int data = 9;
5      BirthDate d1 = new BirthDate(22, 12, 1982);
6      BirthDate d2 = new BirthDate(10, 10, 1958);
7      test.m1(date); //5
8      test.m2(d1);
9      test.m3(d2);
10   }

12   public void m1(int i) {
13     i = 1234;
14   }
15   public void m2(BirthDate b) {
16     b = new BirthDate(15, 6, 2010);
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

# 程序调用过程（二）

- 调用 test 对象的 m1 方法，以 date 为参数。JVM 读取这段代码时，检测到 i 是局部变量，则会把 i 放在栈中，并且把 date 的值赋给 i。

# 程序调用过程（三）



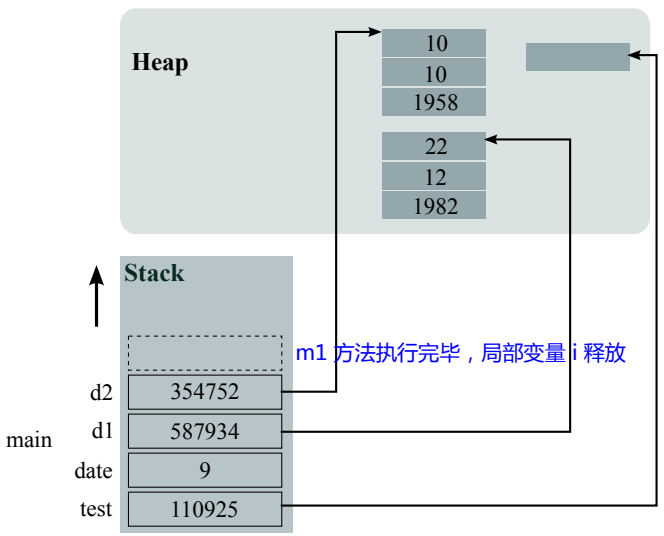

```
1  public class Test {
2    public static void main(String[] args) {
3      Test test = new Test();
4      int data = 9;
5      BirthDate d1 = new BirthDate(22, 12, 1982);
6      BirthDate d2 = new BirthDate(10, 10, 1958);
7      test.m1(date);
8      test.m2(d1);
9      test.m3(d2);
10   }

12   public void m1(int i) {
13     i = 1234; //6
14   }
15   public void m2(BirthDate b) {
16     b = new BirthDate(15, 6, 2010);
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

# 程序调用过程（三）

- 把 1234 赋值给 i。

# 程序调用过程（四）



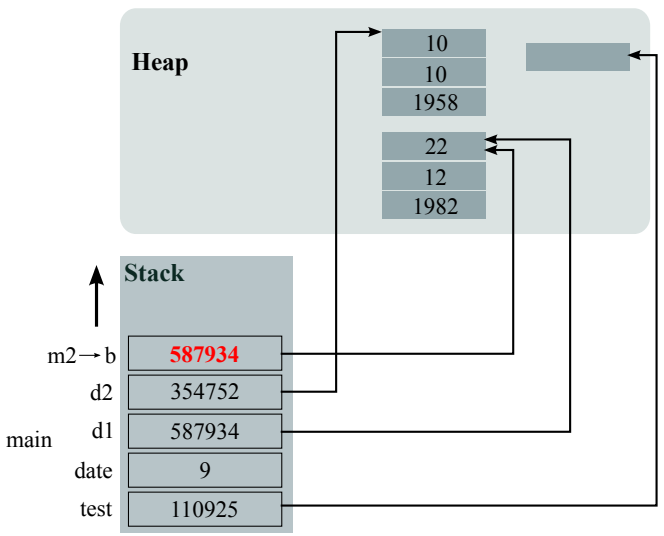

```
1  public class Test {
2    public static void main(String[] args) {
3      Test test = new Test();
4      int data = 9;
5      BirthDate d1 = new BirthDate(22, 12, 1982);
6      BirthDate d2 = new BirthDate(10, 10, 1958);
7      test.m1(date);
8      test.m2(d1);
9      test.m3(d2);
10   }

12   public void m1(int i) {
13     i = 1234;
14   }
15   public void m2(BirthDate b) {
16     b = new BirthDate(15, 6, 2010);
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

# 程序调用过程（四）

- m1 方法执行完毕，立即释放局部变量 i 所占用的栈空间。。

# 程序调用过程（五）



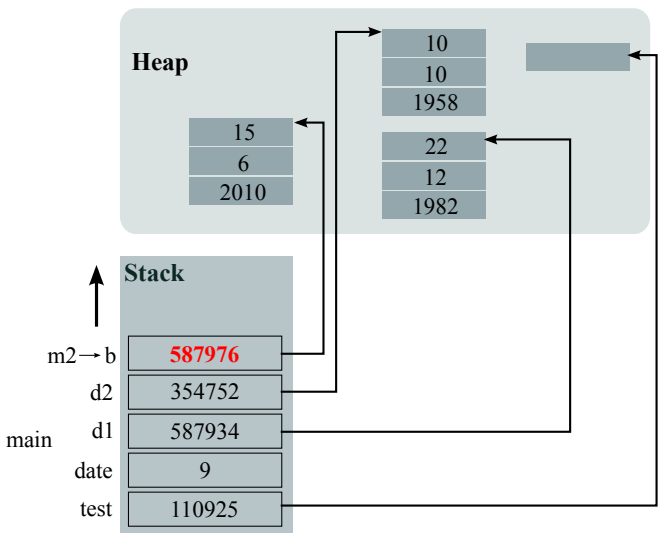

```
1  public class Test {
2    public static void main(String[] args) {
3      Test test = new Test();
4      int data = 9;
5      BirthDate d1 = new BirthDate(22, 12, 1982);
6      BirthDate d2 = new BirthDate(10, 10, 1958);
7      test.m1(date);
8      test.m2(d1); //7
9      test.m3(d2);
10   }

12   public void m1(int i) {
13     i = 1234;
14   }
15   public void m2(BirthDate b) {
16     b = new BirthDate(15, 6, 2010);
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

# 程序调用过程（五）

- 调用 test 对象的 m2 方法，以实例 d1 为参数。JVM 检测到 m2 方法中的 b 参数为局部变量，立即加入到栈中，由于是引用类型的变量，所以 b 中保存的是 d1 中的指针，此时 b 和 d1 指向同一个堆中的对象。在 b 和 d1 之间传递是指针。

# 程序调用过程（六）




```
1  public class Test {
2    public static void main(String[] args) {
3      Test test = new Test();
4      int data = 9;
5      BirthDate d1 = new BirthDate(22, 12, 1982);
6      BirthDate d2 = new BirthDate(10, 10, 1958);
7      test.m1(date);
8      test.m2(d1);
9      test.m3(d2);
10   }

12   public void m1(int i) {
13     i = 1234;
14   }
15   public void m2(BirthDate b) {
16     b = new BirthDate(15, 6, 2010); //8
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

# 程序调用过程（六）

- m2 方法中又实例化了一个 BirthDate 对象，并且赋给 b。在内部执行过程是：在堆区 new 了一个对象，并且把该对象的指针保存在栈中 b 对应空间，此时实例 b 不再指向实例 d1 所指向的对象，但是实例 d1 所指向的对象并无变化，未对 d1 造成任何影响。

# 程序调用过程（七）

- m2 方法执行完毕，立即释放局部引用变量 b 所占的栈空间，注意只是释放了栈空间，堆空间要等待自动回收。

# 程序调用过程（八）

- 调用 test 实例的 m3 方法，以实例 d2 为参数。JVM 会在栈中为局部引用变量 b 分配空间，并且把 d2 中的指针存放在 b 中，此时 d2 和 b 指向同一个对象。再调用实例 b 的 setDay 方法，其实就是调用 d2 指向的对象的 setDay 方法。
- 调用实例 b 的 setDay 方法会影响 d2，因为二者指向的是同一个对象。
- m3 方法执行完毕，立即释放局部引用变量 b。

# 小结

- 基本类型和引用类型，二者作为局部变量都存放在栈中。基本类型直接在栈中保存值，引用类型只保存一个指向堆区的指针，真正的对象存放在堆中。作为参数时基本类型就直接传值，引用类型传指针。
- 注意什么是对象。

```
1  Class a = new Class();
```

此时 a 是指向对象的指针，而不能说 a 是对象。指针存储在栈中，对象存储在堆中，操作实例实际上是通过指针间接操作对象。多个指针可以指向同一个对象。

# 小结（续）

- 栈中的数据和堆中的数据销毁并不是同步的。方法一旦结束，栈中的局部变量立即销毁，但是堆中对象不一定销毁。因为可能有其他变量也指向了这个对象，直到栈中没有变量指向堆中的对象时，它才销毁；而且还不是马上销毁，要等垃圾回收扫描时才可以被销毁。
- 以上的栈、堆、代码段、数据段等都是相对于应用程序而言的。每一个应用程序都对应唯一的一个 JVM 实例，每一个 JVM 实例都有自己的内存区域，互不影响。并且这些内存区域是所有线程共享的。
- 类的成员变量在不同对象中各不相同，都有自己的存储空间（成员变量在堆中的对象中）。而类的方法却是该类的所有对象共享的。

# 接下来…

Java 内存模型

Java 程序内存运行分析

常量池技术

Java 内存建议

# 基本类型和基本类型的包装类

## ❖ 基本类型

byte、short、char、int、long、boolean

## ❖ 基本类型的包装类

Byte、Short、Character、Integer、Long、Boolean

## ❖ 二者的区别

基本类型体现在程序中是普通变量，基本类型的包装类是类，体现在程序中是引用变量。因此二者在内存中的存储位置不同：基本类型存储在栈中，而基本类型包装类存储在堆中。

**上述包装类均实现了常量池技术，另外两种浮点数类型的包装类则没有实现。String 类型也实现了常量池技术。**

# 代码实例

```
1  public class Test {
2    public static void main(String[] args) {
3      objPoolTest();
4    }
5
6    public static void objPoolTest() {
7      int i = 40;
8      int i0 = 40;
9      Integer i1 = 40;
10     Integer i2 = 40;
11     Integer i3 = 0;
12     Integer i4 = new Integer(40);
13     Integer i5 = new Integer(40);
14     Integer i6 = new Integer(0);
15     Double d1 = 1.0;
16     Double d2 = 1.0;
17
18     System.out.println("i=i0\t" + (i == i0));
19     System.out.println("i1=i2\t" + (i1 == i2));
20     System.out.println("i1=i2+i3\t" + (i1 == i2 + i3));
21     System.out.println("i4=i5\t" + (i4 == i5));
22     System.out.println("i4=i5+i6\t" + (i4 == i5 + i6));
23     System.out.println("d1=d2\t" + (d1==d2));
24     System.out.println();
25   }
26 }
```

# Output

output

```
i=i0    true
i1=i2   true
i1=i2+i3    true
i4=i5   false
i4=i5+i6    true
d1=d2   false
```

# Output analysis 1

- i 和 i0 均是普通类型（int）的变量，所以数据直接存储在栈中，而栈有一个很重要的特性：栈中的数据可以共享。当我们定义了 int i = 40;，再定义 int i0 = 40;，这时候会自动检查栈中是否有 40 这个数据，如果有，i0 会直接指向 i 的 40，不会再添加一个新的 40。
- i1 和 i2 均是引用类型，在栈中存储指针，因为 Integer 是包装类，实现了常量池技术，因此 i1、i2 的 40 均是从常量池中获取的，均指向同一个地址，因此 i1 = i2。
- 很明显这是一个加法运算，Java 的数学运算都是在栈中进行的，Java 会自动对 i1、i2 进行拆箱操作转化成整型，因此 i1 在数值上等于 i2 + i3。

# Output analysis 2

- i4 和 i5 均是引用类型，在栈中存储指针，因为 Integer 是包装类。但是由于他们各自都是 new 出来的，因此不再从常量池寻找数据，而是从堆中各自 new 一个对象，然后各自保存指向对象的指针，所以 i4 和 i5 不相等，因为他们所存指针不同，所指向对象不同。
- 也是一个加法运算，同理 3。
- d1 和 d2 均是引用类型，在栈中存储指针，因为 Double 是包装类。但Double 包装类没有实现常量池技术，因此 Double d1 = 1.0; 相当于 Double d1 = new Double(1.0);，是在堆中 new 一个对象，d2 同理。因此 d1 和 d2 存放的指针值不同，指向的对象不同，所以不相等。

# Output analysis 3

- 以上提到的几种基本类型包装类均实现了常量池技术，但他们维护的常量仅仅是从 -128 至 127 这个范围内，如果常量值超过这个范围，就会从堆中创建对象，不再从常量池中取。比如，把上边例子改成 Integer i1 = 400; Integer i2 = 400;，很明显超过了 127，无法从常量池中获取常量，就要从堆中 new 新的 Integer 对象，这时 i1 和 i2 就不相等。
- String 类型也实现了常量池技术，但是稍有不同。String 型是先检测常量池中有没有对应字符串，如果有则取出来；如果没有则把当前的添加进去。

```
1 // s1, s2 分别位于堆中不同空间
2 String s1 = new String("hello");
3 String s2 = new String("hello");
4 System.out.println(s1 == s2); // 输出 false
5 // s3, s4 位于池中同一空间
6 String s3 = "hello";
7 String s4 ="hello";
8 System.out.println(s3 == s4);// 输出 true
```

# 接下来…

Java 内存模型

Java 程序内存运行分析

常量池技术

Java 内存建议

# Java 人为的内存管理是必要的

Java 需要内存管理。虽然 JVM 已经代替开发者完成了对内存的管理，但是硬件本身的资源是有限的，如果 Java 的开发人员不注意内存的使用依然会造成较高的内存消耗，导致性能的降低。

# Java Garbage Collection

**JVM 决定对象是否是垃圾对象，并进行回收。**

垃圾内存并不是用完了马上就被释放，所以会产生内存释放不及时的现象，从而降低内存的使用效率。而当程序庞大的时候，这种现象更为明显，并且垃圾回收（GC）工作需要消耗资源，同样会产生内存浪费。

## ❖ JVM 中的对象生命周期

对象的生命周期一般分为 7 个阶段：❶创建阶段、❷应用阶段、❸不可视阶段、❹不可到达阶段、❺可收集阶段、❻终结阶段、❼释放阶段。

# 创建阶段

## ❖ 减少无谓的对象引用创建

CODE ◗ Test 1

```
1 for( int i=0; i<10000; i++) {
2   Object obj=new Object();
3 }
```

CODE ◗ Test 2

```
1 Object obj=null;
2 for( int i=0; i<10000; i++) {
3   obj=new Object();
4 }
```

分析

Test 2 的性能要比 Test 1 性能要好，内存使用率要高。两段程序每次执行 for 循环都要创建一个 Object 的临时对象，JVM 的垃圾回收不会马上销毁但这些临时对象。相对于 Test 1，Test 2 则只在内存中保存一份对象的引用，而不必创建大量新临时变量，从而降低了内存的使用。

# 创建阶段

## ❖ 不要对同一对象初始化多次

CODE ⏵ Test

```
1  public class A {
2    private Hashtable table = new Hashtable();
3    public A() {
4      table = new Hashtable(); // 应该去掉，因为table已被初始化
5    }
```

### 分析

上述代码 new 了两个 Hashtable，但是却只使用了一个，另外一个则没有被引用而被忽略掉，浪费了内存。并且由于进行了两次 new 操作，也影响了代码的执行速度。另外，不要提前创建对象，尽量在需要的时候创建对象。

# 其他阶段

应用 即该对象至少有一个引用在维护它。

不可视 即超出该变量的作用域。
因为 JVM GC 并不是马上进行回收，而是要判断对象是否被其他引用维护。所以，如果我们在使用完一个对象以后对其进行 obj = null 或者 obj.doSomething() 操作，将其标记为空，则帮助 JVM 及时发现这个垃圾对象。

不可到达 即在 JVM 中找不到对该对象的直接或者间接的引用。

可收集，终结，释放 垃圾回收器发现该对象不可到达，finalize 方法已经被执行，或者对象空间已被重用的时候。

# Java 的 finalize() 方法

Java 所有类都继承自 Object 类，而 finalize() 是 Object 类的一个函数，该函数在 Java 中类似于 C++ 的析构函数（仅仅是类似）。一般来说可以通过重载 finalize() 的形式来释放类中对象。

```
1  public class A {
2    public Object a;
3
4    public A() {
5      a = new Object() ;
6    }
7
8    protected void finalize() throws java.lang.Throwable {
9      a = null; // 标记为空，释放对象
10     super.finalize(); // 递归调用超类中的 finalize 方法
11   }
12 }
```

什么时候 finalize() 被调用由 JVM 来决定。尽量少用 finalize() 函数，finalize() 函数是 Java 提供给程序员一个释放对象或资源的机会。但它会加大 GC 的工作量，因此尽量少采用 finalize 方式回收资源。

# Java 的 finalize() 方法

- 一般的，纯 Java 编写的 Class 不需要重写 finalize() 方法，因为 Object 已经实现了一个默认的，除非我们要实现特殊的功能。
- 用 Java 以外的代码编写的 Class(比如 JNI、C++ 的 new 方法分配的内存)，垃圾回收器并不能对这些部分进行正确的回收，这就需要我们覆盖默认的方法来实现对这部分内存的正确释放和回收。

# 课后作业

1. 搜索关于 Java 常量池技术的相关文档和资源，并进行总结。
2. 搜索关于 Java 中比较操作“==”和“equals()”的相关文档，并进行总结，加深对 Java 内存模型的理解。

# THE END

wxd2870@163.com