Java 应用与开发
Java 内存模型与分配机制

王晓东
wangxiaodong@ouc.edu.cn

中国海洋大学

September 30, 2018

学习目标

1. 理解 JVM 内存模型，掌握 JVM 内存构成
2. 理解 Java 程序的运行过程，学会通过调试模式观察内存的变化
3. 了解 Java 内存管理，认识垃圾回收
4. 建立编程时高效利用内存、避免内存溢出的理念

大纲

接下来…

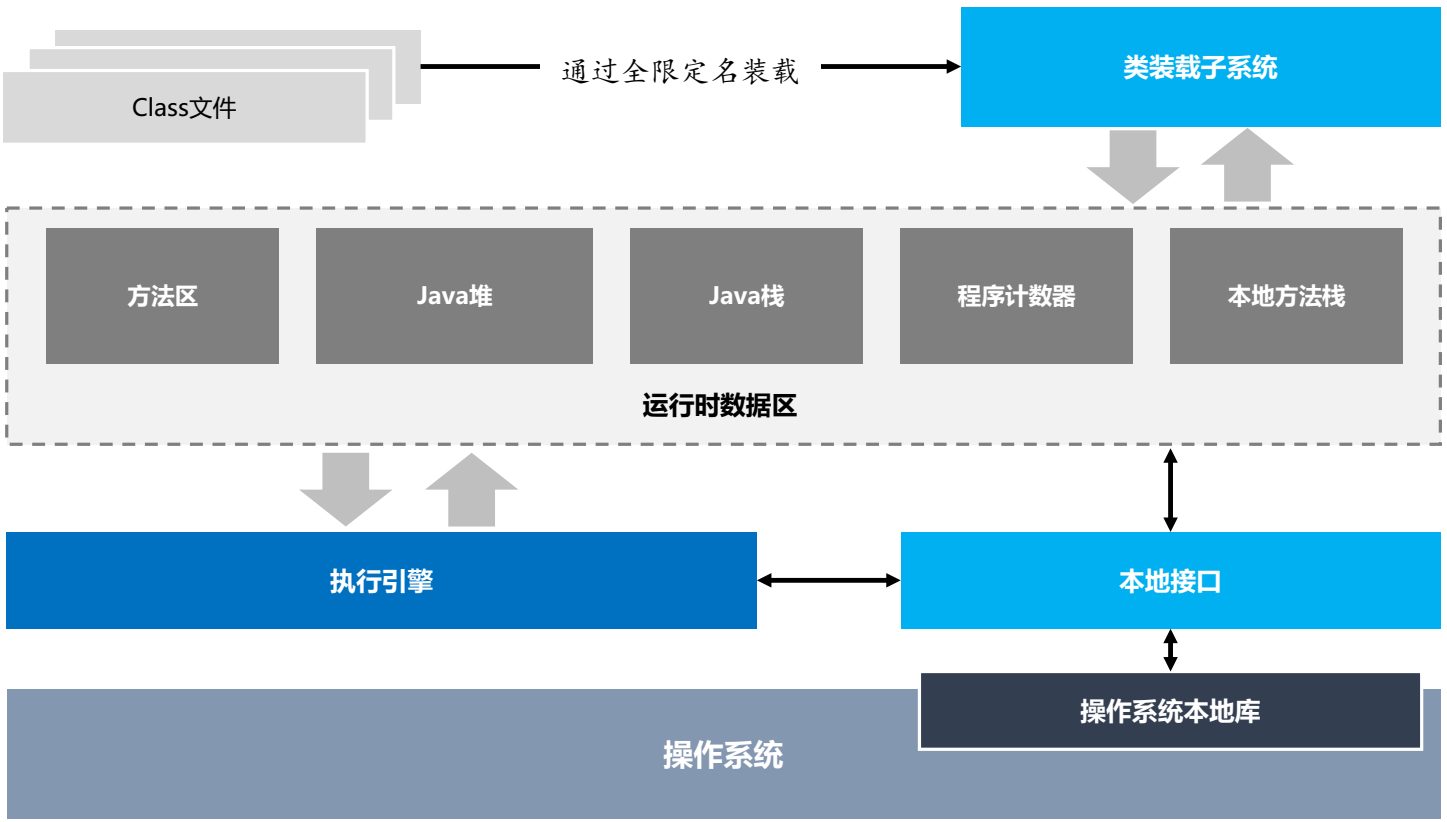
Java 内存模型

Java 程序内存运行分析

Java 内存管理建议

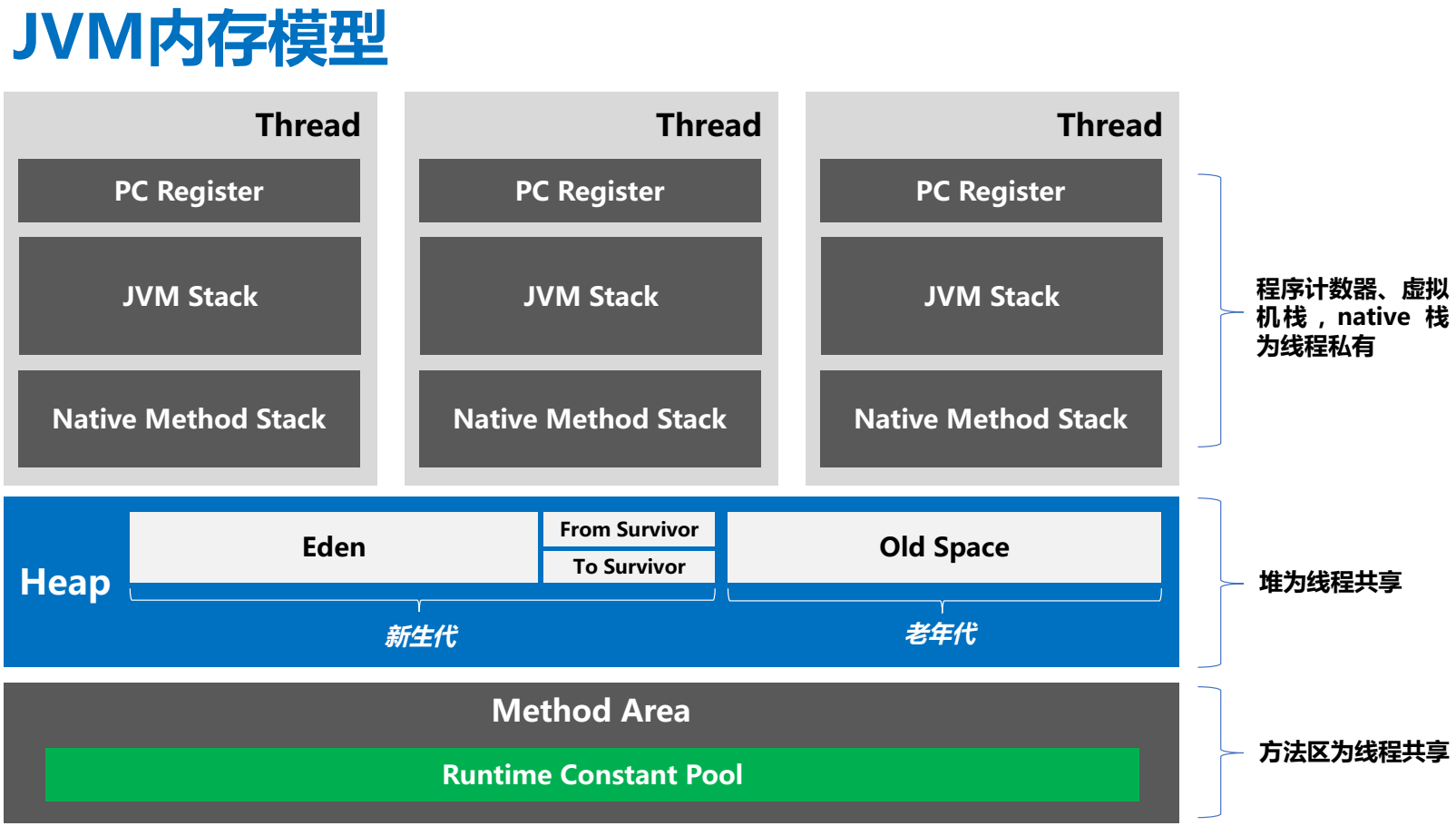
Java 虚拟机 (Java Virtual Machine, JVM)

- Java 程序运行在 JVM 上，JVM 是程序与操作系统之间的桥梁。
- JVM 实现了 Java 的平台无关性。
- JVM 是内存分配的前提。



JVM 内存模型

动画演示 JVM 内存模型



Java 程序运行过程会涉及的内存区域

程序计数器 当前线程执行的字节码的行号指示器。

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 出来的对象和数组。[1]。（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）

[1]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。

Java 程序运行过程会涉及的内存区域

程序计数器 当前线程执行的字节码的行号指示器。

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 出来的对象和数组。[1]。（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）

[1]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。

Java 程序运行过程会涉及的内存区域

程序计数器 当前线程执行的字节码的行号指示器。

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 出来的对象和数组。[1]。(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)

[1]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。

Java 程序运行过程会涉及的内存区域

程序计数器 当前线程执行的字节码的行号指示器。

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 出来的对象和数组。[1]。(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)

[1]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。

Java 程序运行过程会涉及的内存区域

程序计数器 当前线程执行的字节码的行号指示器。

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 出来的对象和数组。[1]。（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）

[1]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。

Java 程序运行过程会涉及的内存区域

程序计数器 当前线程执行的字节码的行号指示器。

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 出来的对象和数组。[1]。（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）

[1]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

Java 内存管理建议

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。
3. 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。

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。
3. 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。

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。
3. 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。**

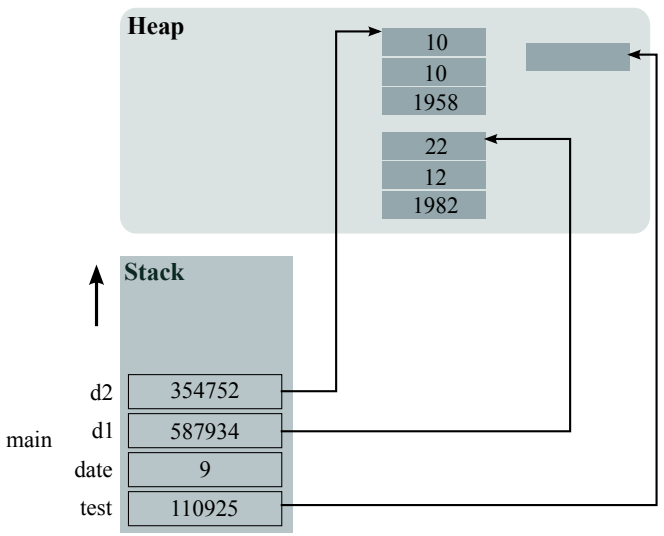
所用讲解程序示例

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 }
```

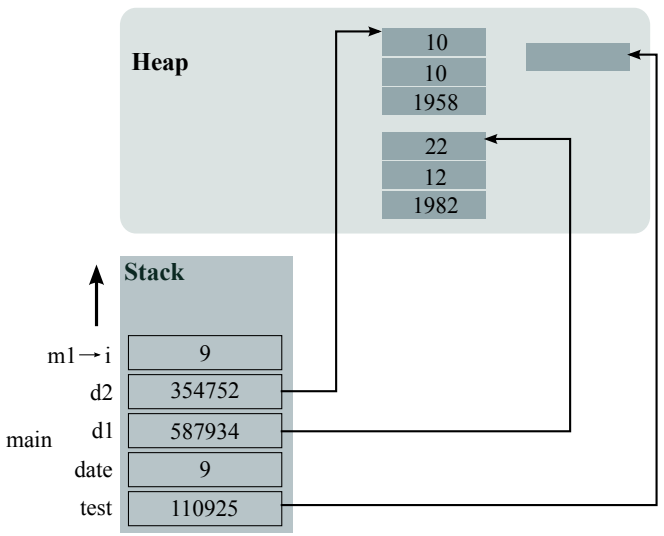
程序调用过程（一）




```
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(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 }
```

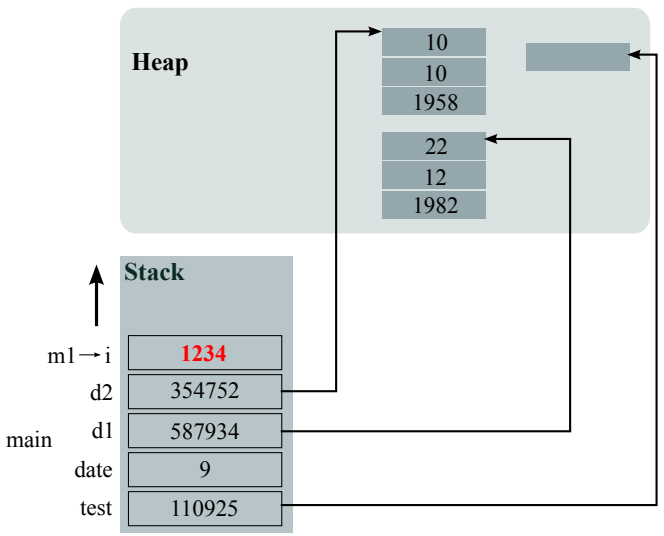
程序调用过程（二）




```
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(data); //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 }
```

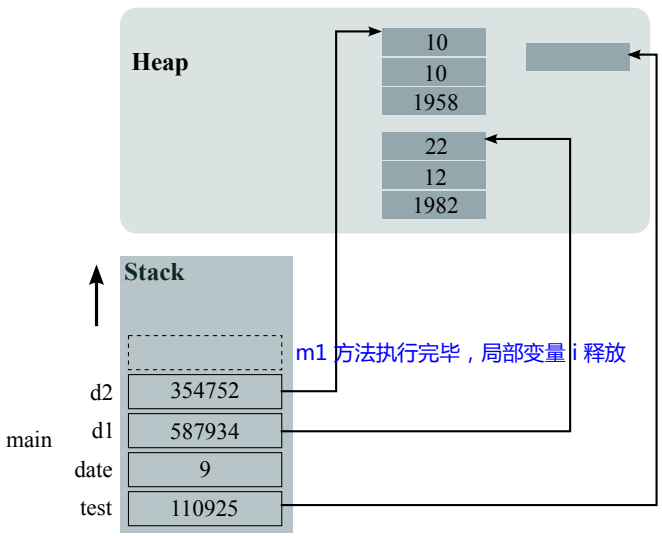
程序调用过程（三）




```
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(data);
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 }
```

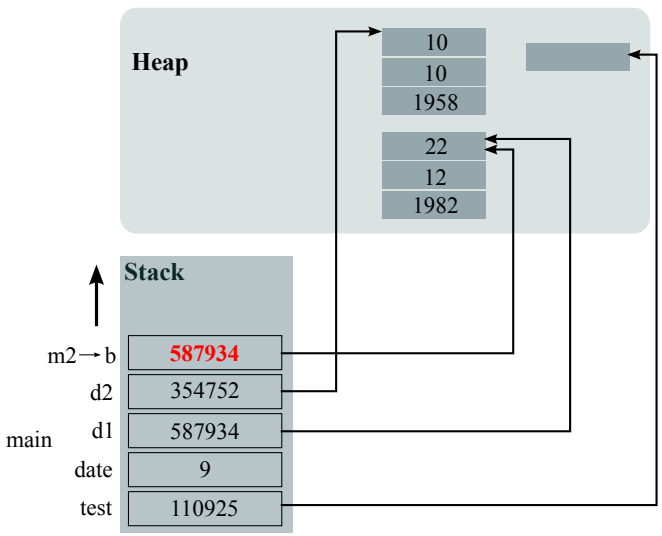
程序调用过程（四）




```
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 }
```

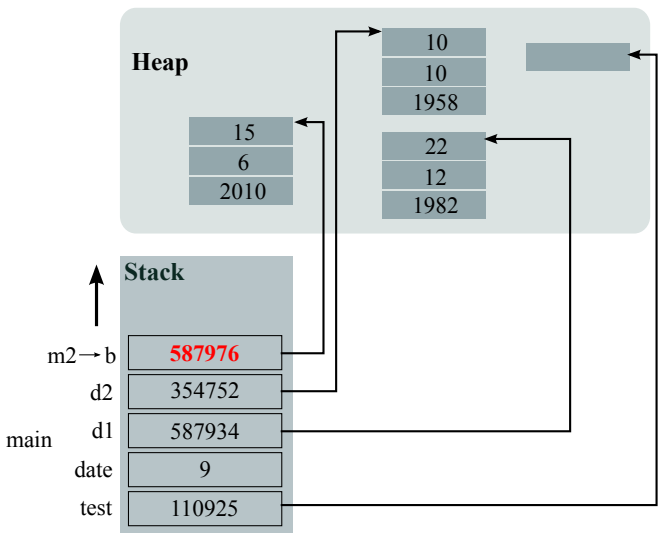
程序调用过程（五）




```
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(data);
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（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(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); //8
17   }
18   public void m3(BirthDate b) {
19     b.setDay(18);
20   }
21 }
```

Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中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    MyClass a = new MyClass();
```

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。

Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中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    MyClass a = new MyClass();
```

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。

Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中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    MyClass a = new MyClass();
```

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。

Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中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    MyClass a = new MyClass();
```

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。

Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中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    MyClass a = new MyClass();
```

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。

Java 程序运行内存分析小结

- 栈中的数据和堆中的数据销毁并不是同步的。方法一旦执行结束，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。
- 栈、堆、代码段、数据段等都是相对于应用程序而言的。

Java 程序运行内存分析小结

- 栈中的数据和堆中的数据销毁并不是同步的。
- 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该 JVM 实例所有线程共享的。

接下来…

Java 内存模型

Java 程序内存运行分析

Java 内存管理建议

Java 垃圾回收机制

JVM 的垃圾回收机制（GC）决定对象是否是垃圾对象，并进行回收。

❖ 垃圾回收机制的特点

- 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。
- 垃圾回收工作本身需要消耗资源，同样会产生内存浪费。

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。

JVM 内存溢出和参数调优

☛ 当遇到 OutOfMemoryError 时该如何做？

- 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。
- 栈内存也会溢出，但是更加少见。

处理方法 ❶ 调整 JVM 内存配置；❷ 优化代码

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。

JVM 内存溢出和参数调优

☛ 当遇到 OutOfMemoryError 时该如何做？

- 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。
- 栈内存也会溢出，但是更加少见。

处理方法 ❶ 调整 JVM 内存配置；❷ 优化代码

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。

JVM 内存溢出和参数调优

☛ 当遇到 OutOfMemoryError 时该如何做？

- 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。
- 栈内存也会溢出，但是更加少见。

处理方法 ❶ 调整 JVM 内存配置；❷ 优化代码

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。

JVM 内存溢出和参数调优

☛ 当遇到 OutOfMemoryError 时该如何做？

- 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。
- 栈内存也会溢出，但是更加少见。

处理方法 ❶ 调整 JVM 内存配置；❷ 优化代码

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。

JVM 内存溢出和参数调优

☛ 当遇到 OutOfMemoryError 时该如何做？

- 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。
- 栈内存也会溢出，但是更加少见。

处理方法 ❶ 调整 JVM 内存配置；❷ 优化代码

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。

内存优化的小示例

❖ 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    }
```

内存性能分析

Test 2 比 Test 1 的性能要好。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 则只在栈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。

内存优化的小示例

❖ 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    }
```

内存性能分析

Test 2 比 Test 1 的性能要好。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 则只在栈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。

内存优化的小示例

❖ 不要对同一对象初始化多次

```
1    public class A {
2      private Hashtable table = new Hashtable();
3      public A() {
4        table = new Hashtable();
5      }
6    }
```

内存性能分析

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。

内存优化的小示例

❖ 不要对同一对象初始化多次

```
1    public class A {
2      private Hashtable table = new Hashtable();
3      public A() {
4        table = new Hashtable();
5      }
6    }
```

内存性能分析

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。

本节习题

❖ 简答题

1. 请描述 JVM 的内存模型。
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。
3. 搜索关于 Java 常量池技术的相关文档和资源，并进行总结。（选做）

❖ 小编程

1. 编写 Java 程序内存运行分析部分的程序，通过调试模式跟踪查看内存的变化情况。

THE END

wangxiaodong@ouc.edu.cn