目 录

1 初始化与清理

1.1 初始化

类似于 C++ 的方式，Java 也采用了构造器进行初始化。例子如下：

```
class Rock
{
    Rock()
    {
        System.out.println("Hello world");
    }
}
```

和 C++ 一样，Java 的构造器也可以进行重载。例子如下：

```
class Tree
{
    int height;
    Tree()
    {
        System.out.println("Planting a seeding");
        height = 0;
    }
    Tree(int inititalHeight)
    {
        height = inititalHeight;
        System.out.println("Creating new Tree that is " + height + " feer tall"
            );
    }
}
```

关于默认构造器，这里需要注意：如果程序员没有提供任何构造器，编译器将自动生成一个默认构造器；如果程序员写了一个构造器，编译器就不会自动生成一个默认构造器。

1.2 this 关键字

Java 的 this 关键字和 C++ 的 this 关键字有区别。C++ 的 this 关键字是该对象的地址，而 Java 中是没有指针的。Java 的 this 关键字表示该对象的引用。例子如下：

```
public class Leaf
{
    int i = 0;
    Leaf increment()
    {
        ++i;
        return this; // 相当于返回当前对象
    }
```

```
9       }
```

1.2.1 利用 this 调用构造器

在构造器中，如果为 this 添加了参数列表，将产生对符合此参数列表的某个构造器的明确调用。例子如下：

```
1   public class Flower
2   {
3       int petalCount = 0;
4       String s = "initial value";
5       Flower(int petals)
6       {
7           petalCount = petals;
8       }
9       Flower(String ss)
10      {
11          s = ss;
12      }
13      Flower(String s, int petals)
14      {
15          this(petals);
16          // 不能再写this(s)，否则会覆盖原有的内容
17          this.s = s;
18      }
19      void print()
20      {
21          // 在非构造函数中不能使用this(s)
22          System.out.println("Hello world!");
23      }
24  }
```

1.3 成员初始化

即使程序员没有对类的每个基本类型数据成员初始化，它们都会有一个初始值。在类中定义一个对象引用时，如果不将其初始化，该对象引用的初值就是 null。

与 C++ 不同的是，如果想为类中某个变量赋初值，只要在定义类成员变量的地方将其赋值即可，而 C++ 不允许这样的行为。

1.3.1 使用构造器进行初始化

可以使用构造器进行初始化，但是这无法阻止自动初始化的进行，它将在构造器被调用前发生。

在类的内部，变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于函数定义之间，它们也将在任何函数 (包括构造器) 被调用之前得到初始化。例子如下：

```
class Window
{
    Window(int order)
    {
        System.out.println("order is " + order);
    }
}

class House
{
    Window w1 = new Window(1);
    House()
    {
        w3 = new Window(4);
    }
    Window w2 = new Window(2);
    Window w3 = new Window(3);
}

public class OrderOfInitialization
{
    public static void main(String[] args)
    {
        House h = new House();
    }
}
```

Java 中 static 关键字不能应用于局部变量，只能作用于域。当类中的静态基本类型域没有初始化时，它的值会是基本类型的标准初值。如果它是一个对象引用，那么它的默认初始化值是 null。如果想在定义处进行初始化静态数据，采取的方法与非晶态数据没有什么不同。

静态变量只有在必要时刻才会初始化。只有类被第一次访问或使用时，类中静态成员才会被初始化，而且是类中所有静态成员变量一起被初始化。此后，静态对象不会再次被初始化。需要注意的是，静态成员变量在非静态成员变量之前被初始化。例子如下:

```
class Bowl
{
    Bowl(int marker)
    {
        System.out.println("order is " + marker);
    }
    void foo1()
    {
        System.out.println("foo1: Hello world");
    }
}

class Table
{
    static Bowl bowl1 = new Bowl(1);
    Table()
    {
```

```
18              System.out.println("Table()");
19              bowl2.foo();
20          }
21          void foo2()
22          {
23              System.out.println("foo2: Hello world");
24          }
25          static Bowl bowl2 = new Bowl(2);
26      }
27
28      class CupBoard
29      {
30          Bowl bowl3 = new Bowl(3);
31          static Bowl bowl4 = new Bowl(4);
32          CupBoard()
33          {
34              System.out.println("CupBoard()");
35              bowl4.foo1();
36          }
37          void foo3()
38          {
39              System.out.println("foo3: Hello world");
40          }
41          static Bowl bowl5 = new Bowl(5);
42      }
43
44      public class StaticInitialization
45      {
46          public static void main(String[] args)
47          {
48              System.out.println("Creating new CupBoard() in main");
49              new CupBoard();
50              System.out.println("Creating new CupBoard() in main");
51              new CupBoard();
52              table.foo2();
53              cupboard.foo3();
54          }
55          static Table table = new Table();
56          static CupBoard cupboard = new CupBoard();
57      }
```

假设有一个名为 Dog 的类，Dog 对象创建的步骤如下：

- 当首次创建类型为 Dog 的对象，或者 Dog 类的静态成员变量或静态成员函数 (构造器本身也是静态函数) 被访问到时，Java 解释器必须查找类路径，用于定位 Dog.class 文件，然后载入 Dog.class。
- 载入 Dog.class 之后，将执行静态初始化的所有动作。也就是说，静态初始化在类对象首次加载时进行。
- 使用 new 创建对象时，会在堆上为 Dog 对象分配足够的存储空间。然后将这块存储空间清零，这样一来，Dog 对象中所有基本类型都被设置成了默认值，而引用被设置为 null。

- 随后执行所有出现于字段定义处的初始化动作。
- 执行构造器。

1.3.2 显式的静态初始化

Java 允许将多个静态初始化动作组织成一个特殊的“静态子句”。例子如下：

```
1  public class Spoon
2  {
3      static int i;
4      static
5      {
6          i = 47;
7      }
8  }
```

静态子句只会执行一次。当首次生成这个类的一个对象或者首次访问属于这个类的静态成员变量或静态成员函数时，该代码段将执行。

1.3.3 数组初始化

Java 不允许指定数组的大小。例子如下：

```
1  // 现在拥有的是对数组的一个引用
2  // 我们并没有给数组对象本身分配任何空间
3  // 我们只是为该引用分配了空间
4  int a[];
```

1.4 清理

Java 拥有一个垃圾回收器，用于释放由 new 分配的内存。在其他情况下，如果该内存区域不是由 new 分配，那么垃圾回收器就不知道该如何释放这块特殊内存。

为了解决这种情况，Java 允许在类中定义一个名为 finalize() 的函数。一旦垃圾回收器准备释放对象占用的存储空间，将首先调用其 finalize() 函数，并在垃圾回收动作发生时，真正地回收对象占用的内存。

finalize() 函数与 C++ 中的析构函数有区别，因为 C++ 中的对象一定会被销毁，而 Java 中的对象不一定总是被垃圾回收器回收，而且垃圾回收并不等于析构。需要知道的是，只要程序没有濒临存储空间用完的时刻，垃圾回收器就不会释放程序所创建的任何对象的存储空间。

1.4.1 finalize 函数的用途

finalize 函数不会负责释放对象所占有的内存。无论对象是如何创建的，垃圾回收器都会负责释放对象占据的所有内存。只有当 Java 程序中调用了本地方法分配空间时，finalize 函数才会派上用场。本地方法是一种在 Java 中调用非 Java 代码的方式。在非 Java 代码中，可能会调用 C 的 malloc 函数分配存储空间，此时只有使用了 free 函数，该内存空间才会释放。所以在对象释放时，需要在 finalize 函数中调用 free 函数来释放这块特殊的内存，否则将引起内存泄漏。

下面是使用 finalize() 函数的例子：

```
class Book
{
    boolean checkedOut = false;
    Book(boolean checkOut)
    {
        checkedOut = checkOut;
    }
    void checkIn()
    {
        checkedOut = false;
    }
    protected void finalize()
    {
        if(checkedOut)
            System.out.println("Error: checked out");
    }
}

public class TerminationCondition
{
    public static void main(String[] args)
    {
        Book novel = new Book(true);
        novel.checkIn();
        new Book(true);
        System.gc();
    }
}
```

以上程序通过 finalize() 函数确保所有 Book 对象在被当作垃圾回收钱都应该被 check in。当有一本书没有 check in 时，程序将输出错误情况。类似的，如果一个对象代表了一个打开的文件，在对象被回收前程序员应该关闭这个文件。finalize() 函数可以用来检查文件是否关闭。finalize 函数更多地被用来发现程序中隐晦的缺陷。

1.4.2 垃圾回收器的工作机制

首先需要意识到，Java 中除了基本类型，所有对象都在堆上分配空间。然而 Java 从堆分配空间的速度，可以和其他语言从堆栈上分配空间的速度相媲美。

对比一下 Java 与 C++ 的堆空间分配机制。C++ 的堆像一个院子，里面的每个对象都

负责管理自己的地盘。如果对象被销毁了，这个地盘必须加以重新利用。在 Java 中，堆更像一个堆栈，每分配一个新对象，它就往前移动一格。

Java 这样的实现方式会导致频繁的内存页面调度。为了避免这种情况的出现，Java 使用了垃圾回收器。垃圾回收器工作的时候，一边回收空间，一边讲堆中的对象排列紧凑。通过垃圾回收器对对象重新排列，实现了一种告诉的、有无限空间可供分配的堆模型。

Java 垃圾回收器依据的思想是：对任何活的对象，一定能追溯到其存活在堆栈或静态存储区之中的引用。那么，只要从堆栈或静态存储区之中的引用开始遍历，就能找到所有活的对象。

Java 垃圾回收器处理存活对象的方式为：先暂停程序的运行，将所有存活的对象从当前堆复制到另一个堆，没有复制的全都是垃圾。当对象被复制到新堆，它们是紧凑排列的。这就避免了内存页面调度频繁的发生。这种做法称为“停止-复制”。

如果 Java 虚拟机发现程序很少产生垃圾甚至不产生垃圾时，Java 会切换到另一种工作模式，叫做“标记-清扫”。“标记-清扫”的思路是：从堆栈和静态存储区出发，遍历所有引用，进而找出所有存活的对象。每当它找到一个存活对象，就会给对象设一个标记。当标记完所有存活的对象后，清理工作才开始进行。在清理过程中，没有标记的对象会被释放。这样一来，就不会有复制动作的发生，但是剩下的堆空间也会不连续。垃圾回收器要是希望得到连续空间，就要重新整理剩下的对象。“标记-清扫”工作必须在程序暂停的情况下才能进行。