第十章

数组

在 Java 编程语言中，数组是动态创建的对象(§4.3.1)，也可以赋值给 Object 类型的变量(§4.3.2)。 Object 类的所有方法都可以在数组上调用。

数组对象包含许多变量。变量的数量可以为零，在这种情况下，数组被称为空数组。数组中包含的变量没有名称;相反，它们由使用非负整数索引值的数组访问表达式引用。这些变量被称为数组的组件。如果一个数组有 n 个组件，我们说 n 是数组的长度;数组的组件使用从 0 到 n - 1(包括 n - 1)的整数索引来引用。

数组的所有组件都具有相同的类型，称为数组的组件类型。如果数组的组件类型为 T，则数组本身的类型为 T[]。

数组的组件类型本身可以是数组类型。这种数组的组件可以包含对子数组的引用。如果从任何数组类型开始，考虑它的组件类型，然后(如果也是数组类型)考虑该类型的组件类型，依此类推，最终必须得到一个不是数组类型的组件类型;这称为原始数组的元素类型，数据结构这一层的组件称为原始数组的元素。

在某些情况下，数组的元素可以是数组:如果元素类型是 Object 或 Cloneable 或 java.io.Serializable，则部分或全部元素可以是数组，因为任何数组对象都可以赋值给这些类型的任何变量。

10.1 数组类型

数组类型用于声明和强制转换表达式中(§15.16)。

数组类型写为元素类型的名称，后跟一些空方括号[]对。括号对的数量表示数组嵌套的深度。

数组类型中的每一个括号对都可以通过类型注解进行注解(§9.7.4)。注解应用于它后面的方括号对(或可变参数声明中的省略号)。

数组的元素类型可以是任何类型，无论是原生类型还是引用类型。特别地:

- 允许使用接口类型作为数组的元素类型。

  这种数组的元素可以使用空引用或实现该接口的任何类型的实例作为其值。

- 允许以抽象类类型作为数组的元素类型。

这种数组的元素可以使用空引用或本身不是抽象类的抽象类的任何子类的实例作为其值。

数组的长度不是其类型的一部分。

§4.10.3 规定了数组类型的超类型。

> 数组类型的超类型关系与超类关系不同。根据§4.10.3，Integer[]的直接超类型是 Numbers[]，但根据 Integer[]的 Class 对象(§10.8)，Integer[]的直接超类是 Object。这在实践中并不重要，因为 Object 也是所有数组类型的超类型。

10.2 数组变量

数组类型的变量保存对对象的引用。声明数组类型的变量不会创建数组对象，也不会为数组组件分配任何空间。它只创建变量本身，该变量可以包含对数组的引用。但是，声明符的初始化部分(§8.3、§9.3、§14.4.1)可以创建一个数组，对该数组的引用随后成为变量的初始值。

例子 10.2-1. 声明数组变量

```
int[]      ai;       // array of int
short[][] as;        // array of array of short
short      s,        // scalar short
           aas[][];  // array    of  array of short
Object[] ao,         // array    of  Object
           otherAo;  // array    of  Object
Collection<?>[] ca; // array of Collection of unknown type
```

上面的声明不创建数组对象。以下是创建数组对象的数组变量声明的示例:

```
Exception ae[] = new Exception[3];
Object aao[][]  = new Exception[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[]       = { 'n', 'o', 't', ' ', 'a', ' ',
                    'S', 't', 'r', 'i', 'n', 'g' };
String[] aas    = { "array", "of", "String", };
```

变量的数组类型取决于括号对，这些括号对可能作为变量声明开头的类型的一部分出现，或作为变量的声明符的一部分出现，或者两者兼而有之。具体地说，在字段、形式参数、局部变量或记录组件的声明中(§8.3, §8.4.1, §9.3, §9.4, §14.4.1, §14.14.2, §15.27.1, §8.10.1), 变量的数组类型表示为:

- 出现在声明开头的元素类型;然后,
- 声明符中变量标识符后面的任何方括号对(不适用于可变参数或记录组件);然后,
- 在类型声明开头出现的任何方括号对(其中可变参数或可变记录组件的省略号被视为方括号对)。

方法(§8.4.5)的返回类型可以是数组类型。精确的数组类型取决于可能作为类型的一部分出

现在方法声明的开头、方法的形式参数列表之后或两者都出现的括号对。数组类型由以下内容表示：

- Result 中出现的元素类型; 然后，
- 形式参数列表后面的任何括号对；然后，
- Result 中出现的任何括号对。

我们不建议在数组变量声明中使用“混合表示法”，其中括号对同时出现在类型和声明符中；也不建议在方法声明中使用，其中括号对同时出现在形参列表的前后。

例子 10.2-2. 数组变量和数组类型

局部变量声明语句:

```
byte[] rowvector, colvector, matrix[];
```

等价与:

```
byte rowvector[], colvector[], matrix[][];
```

因为每个局部变量的数组类型是不变的。同样，局部变量声明语句:

```
int a, b[], c[][];
```

等价于一系列声明语句:

```
int a;
int[] b;
int[][] c;
```

声明符中允许使用方括号，这是对 C 和 C++传统的认可。然而，变量声明的一般规则允许在类型和声明符中同时出现括号，因此局部变量声明语句:

```
float[][] f[][], g[][][], h[];  // Yechh!
```

等价于一系列声明:

```
float[][][][] f;
float[][][][][] g;
float[][][] h;
```

由于数组类型的形成方式，以下参数声明具有相同的数组类型:

```
void m(int @A [] @B [] x) {}
void n(int @A [] @B ... y) {}
```

也许令人惊讶的是，以下字段声明具有相同的数组类型:

```
int @A [] f @B [];
int @B [] @A [] g;
```

创建数组对象后，其长度永远不会改变。要使数组变量引用不同长度的数组，必须将对不同数组的引用赋给该变量。

数组类型的单个变量可以包含对不同长度数组的引用，因为数组的长度不是其类型的一部分。

如果数组变量 v 的类型为 A[]，其中 A 是引用类型，则 v 可以包含对任何数组类型 B[]的实例的引用，前提是 B 可以赋给 A(§5.2)。这可能会导致在以后的分配中出现运行时异常；有关讨论，请参阅§10.5。

10.3 数组创建

数组由数组创建表达式(§15.10.1)或数组初始化器(§10.6)创建。

数组创建表达式指定元素类型、嵌套数组的层数以及至少一个嵌套级别的数组的长度。数组的长度存储为一个 final 实例变量 length。

数组初始化器创建数组并为其所有组件提供初始值。

10.4 数组访问

数组的组件由数组访问表达式(§15.10.3)访问，该表达式由一个值为数组引用的表达式组成，后跟一个由[和]括起来的索引表达式，如 A[i]。

所有数组都以 0 为起点。长度为 n 的数组可以由 0 到 n-1 的整数索引。

例子 10.4-1. 数组访问

```
class Gauss {
    public static void main(String[] args) {
        int[] ia = new int[101];
        for (int i = 0; i < ia.length; i++) ia[i] = i;
        int sum = 0;
        for (int e : ia) sum += e;
        System.out.println(sum);
```

此程序生成以下输出：

```
5050
```

程序声明了一个类型为 int 的数组变量 ia，即 int[]。变量 ia 被初始化为引用由数组创建表达式创建的新创建的数组对象(§15.10.1)。数组创建表达式指定该数组应具有 101 个组件。可以使用字段 length 获得数组的长度，如程序所示。程序用 0 到 100 之间的整数填充数组，对这些整数求和，然后打印结果。

数组必须使用 int 类型的值作为索引值，short、byte 或 char 类型的值也可以用作索引值，因为它们经过一元数值提升 (§5.6)成为 int 值。

尝试用 long 类型的索引访问数组组件会导致编译时错误。

所有数组访问都在运行时进行检查；尝试使用小于零或大于或等于数组长度的索引会引发

ArrayIndexOutOfBoundsException (§15.10.4)。

10.5 数组存储异常

对于类型为 A[]的数组，其中 A 是引用类型，在运行时检查数组组件的赋值，以确保被赋值的值可赋值给该组件。

如果赋值的值的类型与组件类型不是赋值兼容的(§5.2)，则抛出 ArrayStoreException。

如果数组的组件类型不可具体化(§4.7)，则 Java 虚拟机不能执行上一段中描述的存储检查。这就是为什么禁止使用不可具体化元素类型的数组创建表达式(§15.10.1)。可以声明元素类型为不可具体化的数组类型的变量，但将数组创建表达式的结果赋值给该变量必然会导致未检查的警告(§5.1.9)。

例子 10. 5-1. `ArrayStoreException`

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    public static void main(String[] args) {
        ColoredPoint[] cpa = new ColoredPoint[10];
        Point[] pa = cpa;
        System.out.println(pa[1] == null);
        try {
            pa[0] = new Point();
        } catch (ArrayStoreException e) {
            System.out.println(e);
        }
    }
```

此程序生成以下输出：

```
true
java.lang.ArrayStoreException: Point
```

变量 pa 的类型为 Point[]，变量 cpa 的值为对类型为 ColoredPoint[]的对象的引用。可以将 ColoredPoint 分配给 Point；因此，可以将 cpa 的值分配给 pa。

对此数组 pa 的引用(例如，测试 pa[1]是否为空)不会导致运行时类型错误。这是因为类型为 ColoredPoint[]的数组的元素是一个 ColoredPoint，并且每个 ColoredPoint 都可以代表一个 Point，因为 Point 是 ColoredPoint 的超类。

另一方面，对数组 pa 的赋值可能导致运行时错误。在编译时，检查对 pa 的元素的赋值，以确保赋值的值是一个 Point。但是，由于 pa 包含对 ColoredPoint 数组的引用，因此只有在运行时分配的值的类型是 ColoredPoint 时，赋值才有效。

Java 虚拟机在运行时检查这种情况，以确保分配有效；如果不是，则抛出 ArrayStoreException 异常。

10.6 数组初始化器

数组初始化器可在字段声明(§8.3、§9.3)或局部变量声明(§14.4)中指定，或作为数组创建表达式(§15.10.1)的一部分来指定，以创建数组并提供一些初始值。

ArrayInitializer:
{ *[VariableInitializerList] [,]* }

VariableInitializerList:
VariableInitializer {, VariableInitializer}

为方便起见，此处显示了§8.3 中的以下产品：

VariableInitializer:
Expression
ArrayInitializer

数组初始化器写作逗号分隔的表达式列表，用大括号{和}括起来。

在数组初始化器中，尾随逗号可能出现在最后一个表达式之后，并被忽略。

每一个变量初始化器必须和数组组件类型是赋值兼容的 (§5.2),否则会发生编译时错误。

如果被初始化的数组的组件类型不可具体化，则为编译时错误(§4.7)。

要构造的数组的长度等于数组初始化器的大括号中立即包含的变量初始化器的数量。为该长度的新数组分配空间。如果没有足够的空间分配数组，则数组初始化器的求值会突然结束，并抛出 OutOfMemoryError。否则，一个一维数组就会按照指定的长度被创建，数组的每个组件都被初始化为其默认值(§4.12.5)。

然后，立即由数组初始化器的大括号括起来的变量初始化器按照它们在源代码中出现的文本顺序从左到右执行。第 n 个变量初始化器指定了第 n-1 个数组组件的值。如果变量初始化器的执行突然结束，那么数组初始化器的执行也会因为同样的原因突然结束。如果所有变量初始化式表达式都正常完成，那么数组初始化器也会正常完成，并带有新初始化的数组的值。

如果组件类型是数组类型，那么指定组件的变量初始化器本身可能就是数组初始化器;也就是说，数组初始化器可以嵌套。在这种情况下，嵌套数组初始化器的执行将通过递归应用上述算法构造并初始化数组对象，并将其赋值给组件。

例子 10.6-1. 数组初始化器

```
class Test {
    public static void main(String[] args) {
       int[][] ia = { { 1, 2 }, null };
       for (int[] ea : ia) {
          for (int e: ea) {
             System.out.println(e);
          }
       }
    }
}
```

这个程序产生输出:

```
1
2
```

在试图索引数组 ia 的第二个组件(这是一个空引用)时导致 NullPointerException。

10.7 数组成员

数组类型的成员如下所示:

- 公共 final 字段 length，它包含数组的组件数。length 可以为正，也可以为零。
- 公共方法 clone，它重写 Object 类中的同名方法，不引发检查异常。数组类型 T[]的 clone 方法的返回类型是 T[]。

  多维数组的克隆是浅拷贝的，也就是说，它只创建单个新数组。子数组是共享的。
- 从类 Object 继承的所有成员；Object 的唯一未继承的方法是其 clone 方法。

有关 Object 的公共方法和非公共方法之间的差异需要特别注意的另一种情况，请参见§9.6.4.4。

因此，数组具有与以下类相同的公共字段和方法:

```
class A<T> implements Cloneable, java.io.Serializable {
public final int length = X;
    public T[] clone() {
        try {
            return (T[])super.clone();
        } catch (CloneNotSupportedException e) {
        throw new InternalError(e.getMessage());
        }
    }
}
```

请注意，如果数组确实是以这种方式实现的，则上述代码中 T[]的强制转换将生成一个未经检查的警告(§5.1.9)。

例子 10.7-1. 数组是可克隆的

```
class Test1 {
    public static void main(String[] args) {
        int[] ia1 = { 1, 2 };
        int[] ia2 = ia1.clone();
        System.out.print((ia1 == ia2) + " ");
        ia1[1]++;
        System.out.println(ia2[1]);
    }
}
```

此程序生成以下输出:

```
false 2
```

表明由 ia1 和 ia2 引用的数组的组件是不同的变量。

例子 10.7-2. 克隆后的共享子数组

此程序显示了在克隆多维数组时共享子数组的事实:

```
class Test2 {
    public static void main(String[] args) throws Throwable {
        int[][] ia = { { 1, 2 }, null };
        int[][] ja = ia.clone();
        System.out.print((ia == ja) + " ");
        System.out.println(ia[0] == ja[0] && ia[1] == ja[1]);
    }
}
```

此程序生成以下输出：

```
false true
```

表示名为 ia[0]的 int[]数组和名为 ja[0]的 int[]数组是相同的数组。

10.8 数组的 Class 对象

每个数组都有一个关联的 Class 对象，与具有相同组件类型的所有其他数组共享。

尽管数组类型不是类，但每个数组的 Class 对象的行为类似于：

- 每个数组类型的直接超类是 Object。
- 每个数组类型实现接口 Cloneable 和 java.io.Serializable。

例子 10.8-1. 数组的 Class 对象

```
class Test1 {
    public static void main(String[] args) {
       int[] ia = new int[3];
       System.out.println(ia.getClass());
       System.out.println(ia.getClass().getSuperclass());
       for (Class<?> c : ia.getClass().getInterfaces())
           System.out.println("Superinterface: " + c);
       }
}
```

此程序生成以下输出：

```
class [I
class java.lang.Object
Superinterface: interface java.lang.Cloneable
Superinterface: interface java.io.Serializable
```

字符串"[I" 是 Class 对象"组件类型为 int 的数组"的运行时类型签名。

例子 10.8-2. 数组 Class 对象是共享的

```
class Test2 {
    public static void main(String[] args) {
       int[] ia = new int[3];
       int[] ib = new int[6];
       System.out.println(ia == ib);
       System.out.println(ia.getClass() == ib.getClass());
    }
```

```
}
```

此程序生成以下输出：

```
false true
```

虽然 ia 和 ib 引用了不同的数组，但 Class 对象的比较结果表明，组件类型为 int 的所有数组都是相同数组类型(即 int[])的实例。

10.9 字符数组不是字符串

在 Java 编程语言中，与 C 不同的是，字符数组不是字符串，字符串和字符数组都不以‘\u0000’(NUL 字符)结尾。

字符串对象是不可变的，也就是说，它的内容永远不会改变，而字符数组具有可变的元素。

String 类中的方法 toCharArray 返回一个字符数组，该数组包含与 String 相同的字符序列。类 StringBuffer 在可变字符数组上实现了有用的方法。