Java 基础：集合总结（6）

排序和搜索

数组

Arrays 类为所有基本数据类型的数组提供了一个过载的 sort()和 binarySearch()，它们亦可用于 String 和 Object。

```
public class Array1 {
    static Random r = new Random();
    static String ssource = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
            + "abcdefghijklmnopqrstuvwxyz";
    static char[] src = ssource.toCharArray();
    // Create a random String
    public static String randString(int length) {
        char[] buf = new char[length];
        int rnd;
        for (int i = 0; i < length; i++) {
            rnd = Math.abs(r.nextInt()) % src.length;
            buf[i] = src[rnd];
        }
        return new String(buf);
    }
    // Create a random array of Strings:
    public static String[] randStrings(int length, int size) {
        String[] s = new String[size];
        for (int i = 0; i < size; i++)
            s[i] = randString(length);
        return s;
    }
    public static void print(byte[] b) {
        for (int i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();
    }
    public static void print(String[] s) {
```

```
        for (int i = 0; i < s.length; i++)
            System.out.print(s[i] + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        byte[] b = new byte[15];
        r.nextBytes(b); // Fill with random bytes
        print(b);
        Arrays.sort(b);
        print(b);
        int loc = Arrays.binarySearch(b, b[10]);
        System.out.println("Location of " + b[10] + " = " + loc);
        // Test String sort & search:
        String[] s = randStrings(4, 10);
        print(s);
        Arrays.sort(s);
        print(s);
        loc = Arrays.binarySearch(s, s[4]);
        System.out.println("Location of " + s[4] + " = " + loc);
    }
}
```

在 main()中，Random.nextBytes() 用随机选择的字节填充数组自变量（没有对应的 Random 方法用于创建其他基本数据类型的数组）。获得一个数组后，便可发现为了执行 sort()或者 binarySearch()，只需发出一次方法调用即可。**与 binarySearch()有关的还有一个重要的警告：若在执行一次 binarySearch()之前不调用 sort()，便会发生不可预测的行为，其中甚至包括无限循环。**

对 String 的排序以及搜索是相似的，但在运行程序的时候，我们会注意到一个有趣的现象：排序遵守的是字典顺序，亦即大写字母在字符集中位于小写字母的前面。因此，所有大写字母都位于列表的最前面，后面再跟上小写字母—— Z 居然位于 a 的前面。似乎连电话簿也是这样排序的。

- **可比较与比较器**

若想对一个 Object 数组进行排序，那么必须解决一个问题。根据什么来判定两个 Object 的顺序呢？不幸的是，最初的 Java 设计者并不认为这是一个重要的问题，否则就已经在根类 Object 里定义它了。这样造成的一个后果便是：必须从外部进行 Object 的排序，而且新的集合库提供了实现这一操作的标准方式（最理想的是在 Object 里定义它）。

针对 Object 数组（以及 String，它当然属于 Object 的一种），可使用一个 sort()，并令其接纳另一个参数：实现了 Comparator 接口（即“比较器”接口，新集合库的一部分）的一个对象，并用它的单个 compare()方法进行比较。这个方法将两个准备比较的对象作为自己的参数使用—— **若第一个参数小于第二个，返回一个负整数；若相等，返回零；若第一**

个参数大于第二个，则返回正整数。基于这一规则，上述例子的 String 部分便可重新写过，令其进行真正按字母顺序的排序：

通过造型为 String， compare()方法会进行“暗示”性的测试，保证自己操作的只能是 String 对象—— 运期系统会捕获任何差错。将两个字串都强迫换成小写形式后，String.compareTo()方法会产生预期的结果若用自己的 Comparator 来进行一次 sort()，那么在使用 binarySearch()时必须使用那个相同的 Comparator。

Arrays 类提供了另一个 sort()方法，它会采用单个自变量：一个 Object 数组，但没有 Comparator。这个 sort()方法也必须用同样的方式来比较两个 Object。**通过实现 Comparable 接口，它采用了赋予一个类的“自然比较方法”。 这个接口含有单独一个方法 —— compareTo()，能分别根据它小于、等于或者大于自变量而返回负数、零或者正数，从而实现对象的比较。**

```
public class CompClass implements Comparable {
    private int i;
    public CompClass(int ii) {
        i = ii;
    }
    public int compareTo(Object o) {
        // Implicitly tests for correct type:258
        int argi = ((CompClass) o).i;
        if (i == argi)
            return 0;
        if (i < argi)
            return -1;
        return 1;
    }
    public static void print(Object[] a) {
        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
    }
    public String toString() {
        return i + "";
    }
    public static void main(String[] args) {
        CompClass[] a = new CompClass[20];
        for (int i = 0; i < a.length; i++)
            a[i] = new CompClass((int) (Math.random() * 100));
        print(a);
        Arrays.sort(a);
```

```
        print(a);
        int loc = Arrays.binarySearch(a, a[3]);
        System.out.println("Location of " + a[3] + " = " + loc);
    }
}
```

- **列表**

可用与数组相同的形式排序和搜索一个列表(List)。用于排序和搜索列表的静态方法包含在类 Collections 中,但它们拥有与 Arrays 中差不多的签名： sort(List)用于对一个实现了 Comparable 的对象列表进行排序； binarySearch(List,Object)用于查找列表中的某个对象； sort(List,Comparator)利用一个“比较器”对一个列表进行排序；而 binarySearch(List,Object,Comparator)则用于查找那个列表中的一个对象。

```
public class ListSort {
    public static void main(String[] args) {
        final int SZ = 20;
        // Using "natural comparison method":
        List a = new ArrayList();
        for(int i = 0; i < SZ; i++)
        a.add(new CompClass(
        (int)(Math.random() *100)));
        Collection1.print(a);
        Collections.sort(a);
        Collection1.print(a);
        Object find = a.get(SZ/2);259
        int loc = Collections.binarySearch(a, find);
        System.out.println("Location of " + find +
        " = " + loc);
        // Using a Comparator:
        List b = new ArrayList();
        for(int i = 0; i < SZ; i++)
        b.add(Array1.randString(4));
        Collection1.print(b);
        AlphaComp ac = new AlphaComp();
        Collections.sort(b, ac);
        Collection1.print(b);
        find = b.get(SZ/2);
        // Must use the Comparator to search, also:
        loc = Collections.binarySearch(b, find, ac);
        System.out.println("Location of " + find +
        " = " + loc);
```

```
    }
}
```

这些方法的用法与在 Arrays 中的用法是完全一致的，只是用一个列表代替了数组。

TreeMap 也必须根据 Comparable 或者 Comparator 对自己的对象进行排序。

Collections 类中的实用工具：

enumeration(Collection) 为自变量产生原始风格的 Enumeration（枚举）

max(Collection)， min(Collection) 在自变量中用集合内对象的自然比较方法产生最大或最小元素

max(Collection,Comparator)， min(Collection,Comparator) 在集合内用比较器产生最大或最小元素

nCopies(int n, Object o) 返回长度为 n 的一个不可变列表，它的所有句柄均指向 o

subList(List,int min,int max) 返回由指定参数列表后推得到的一个新列表。可将这个列表想象成一个“窗口”，它自索引为 min 的地方开始，正好结束于 max 的前面。

注意 min()和 max()都是随同 Collection 对象工作的，而非随同 List，所以不必担心 Collection 是否需要排序（就象早先指出的那样，在执行一次 binarySearch()—— 即二进制搜索—— 之前，必须对一个 List 或者一个数组执行 sort()）。

1. 使 Collection 或 Map 不可修改

通常，创建 Collection 或 Map 的一个“只读”版本显得更有利一些。 Collections 类允许我们达到这个目标 ,方法是将原始容器传递进入一个方法 ,并令其传回一个只读版本。这个方法共有四种变化形式，分别用于 Collection（如果不想把集合当作一种更特殊的类型对待）、 List、 Set 以及 Map。

```
public class ReadOnly {
    public static void main(String[] args) {
        Collection c = new ArrayList();
        Collection1.fill(c); // Insert useful data
        c = Collections.unmodifiableCollection(c);
        Collection1.print(c); // Reading is OK
        // ! c.add("one"); // Can't change it
        List a = new ArrayList();
        Collection1.fill(a);
        a = Collections.unmodifiableList(a);
        ListIterator lit = a.listIterator();
        System.out.println(lit.next()); // Reading OK
        // ! lit.add("one"); // Can't change it
        Set s = new HashSet();
        Collection1.fill(s);
        s = Collections.unmodifiableSet(s);
```

```
        Collection1.print(s); // Reading OK
        // ! s.add("one"); // Can't change it
        Map m = new HashMap();
        Map1.fill(m, Map1.testData1);
        m = Collections.unmodifiableMap(m);
        Map1.print(m); // Reading OK
        // ! m.put("Ralph", "Howdy!");
    }
}
```

对于每种情况，在将其正式变为只读以前，都必须用有有效的数据填充容器。一旦载入成功，最佳的做法就是用“不可修改”调用产生的句柄替换现有的句柄。这样做可有效避免将其变成不可修改后不慎改变其中的内容。

在另一方面，该工具也允许我们在一个类中将能够修改的容器保持为 private 状态，并可从一个方法调用中返回指向那个容器的一个只读句柄。这样一来，虽然我们可在类里修改它，但其他任何人都只能读。

为特定类型调用“不可修改”的方法不会造成编译期间的检查，但一旦发生任何变化，对修改特定容器的方法的调用便会产生一个 UnsupportedOperationException 违例。

2. Collection 或 Map 的同步

在这儿，大家只需注意到 Collections 类提供了对整个容器进行自动同步的一种途径。它的语法与“不可修改”的方法是类似的：

```
public class Synchronization {
    public static void main(String[] args) {
        Collection c = Collections.synchronizedCollection(new ArrayList());
        List list = Collections.synchronizedList(new ArrayList());
        Set s = Collections.synchronizedSet(new HashSet());
        Map m = Collections.synchronizedMap(new HashMap());
    }
}
```

总结

1) 数组包含了对象的数字化索引。它容纳的是一种已知类型的对象，所以在查找一个对象时，不必对结果进行造型处理。数组可以是多维的，而且能够容纳基本数据类型。但是，一旦把它创建好以后，大小便不能变化了。
2) Vector(矢量)也包含了对象的数字索引—— 可将数组和 Vector 想象成随机访问集合。当我们加入更多的元素时， Vector 能够自动改变自身的大小。但 Vector 只能容纳对

象的句柄，所以它不可包含基本数据类型；而且将一个对象句柄从集合中取出来的时候，必须对结果进行造型处理。

3) Hashtable（散列表）属于 Dictionary（字典）的一种类型，是一种将对象（而不是数字）同其他对象关联到一起的方式。散列表也支持对对象的随机访问，事实上，它的整个设计方案都在突出访问的“高速度”。
4) Stack（堆栈）是一种“后入先出”（ LIFO）的队列；对于 Hashtable，可将任何东西置入其中，并以非常快的速度检索；对于 Enumeration（枚举），可遍历一个序列，并对其中的每个元素都采取一个特定的操作。那是一种功能足够强劲的工具。

但 Hashtable 没有“顺序”的概念。 Vector 和数组为我们提供了一种线性顺序，但若要把一个元素插入它们任何一个的中部，一般都要付出“惨重”的代价。除此以外，队列、拆散队列、优先级队列以及树都涉及到元素的“排序” —— 并非仅仅将它们置入，以便以后能按线性顺序查找或移动它们。