Java 基础：集合基础（5）

决定使用哪种集合

ArrayList，LinkedList 以及 Vector（大致等价于 ArrayList）都实现了 List 接口，所以无论选用哪一个，我们的程序都会得到类似的结果。然而， ArrayList（以及 Vector）是由一个数组后推得到的；而 LinkedList 是根据常规的双重链接列表方式实现的，因为每个单独的对象都包含了数据以及指向列表内前后元素的句柄。正是由于这个原因，假如想在一个列表中部进行大量插入和删除操作，那么 LinkedList 无疑是最恰当的选择（ LinkedList 还有一些额外的功能，建立于 AbstractSequentialList 中）。若非如此，就情愿选择 ArrayList，它的速度可能要快一些。

作为另一个例子， Set 既可作为一个 ArraySet 实现，亦可作为 HashSet 实现。ArraySet 是由一个 ArrayList 后推得到的，设计成只支持少量元素，特别适合要求创建和删除大量 Set 对象的场合使用。然而，一旦需要在自己的 Set 中容纳大量元素， ArraySet 的性能就会大打折扣。写一个需要 Set 的程序时，应默认选择 HashSet。而且只有在某些特殊情况下（对性能的提升有迫切的需求），才应切换到 ArraySet。

1. 决定使用何种 List

为体会各种 List 实施方案间的差异，最简便的方法就是进行一次性能测验。

```
public class ListPerformance {
    private static final int REPS = 100;
    private abstract static class Tester {
        String name;
        int size; // Test quantity
        Tester(String name, int size) {
            this.name = name;
            this.size = size;
        }
        abstract void test(List a);
    }
    private static Tester[] tests = { new Tester("get", 300) {
        void test(List a) {
            for (int i = 0; i < REPS; i++) {
                for (int j = 0; j < a.size(); j++)
                    a.get(j);
```

```
        }
    }
}, new Tester("iteration", 300) {
    void test(List a) {
        for (int i = 0; i < REPS; i++) {
            Iterator it = a.iterator();
            while (it.hasNext())
                it.next();
        }
    }
}, new Tester("insert", 1000) {
    void test(List a) {
        int half = a.size() / 2;
        String s = "test";
        ListIterator it = a.listIterator(half);
        for (int i = 0; i < size * 10; i++)
            it.add(s);
    }
}, new Tester("remove", 5000) {
    void test(List a) {
        ListIterator it = a.listIterator(3);
        while (it.hasNext()) {
            it.next();
            it.remove();
        }
    }
}, };
public static void test(List a) {
    // A trick to print out the class name:
    System.out.println("Testing " + a.getClass().getName());
    for (int i = 0; i < tests.length; i++) {
        Collection1.fill(a, tests[i].size);
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(a);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
}
public static void main(String[] args) {
```

```
        test(new ArrayList());
        test(new LinkedList());
    }
}
```

内部类 Tester 是一个抽象类,用于为特定的测试提供一个基础类。它包含了一个要在测试开始时打印的字串、一个用于计算测试次数或元素数量的 size 参数、用于初始化字段的一个构建器以及一个抽象方法 test()。 test()做的是最实际的测试工作。各种类型的测试都集中到一个地方： tests 数组。我们用继承于 Tester 的不同匿名内部类来初始化该数组。为添加或删除一个测试项目，只需在数组里简单地添加或移去一个内部类定义即可，其他所有工作都是自动进行的。

```
Type Get Iteration Insert Remove
A r r a y L i s t 110 490 3790 8730
LinkedList 1980 220 110 110
```

在 ArrayList 中进行随机访问（即 get()）以及循环反复是最划得来的；但对于 LinkedList 却是一个不小的开销。但另一方面，在列表中部进行插入和删除操作对于 LinkedList 来说却比 ArrayList 划算得多。**我们最好的做法也许是先选择一个 ArrayList 作为自己的默认起点。以后若发现由于大量的插入和删除造成了性能的降低，再考虑换成 LinkedList 不迟。**

2. 决定使用何种 Set

可在 ArraySet 以及 HashSet 间作出选择,具体取决于 Set 的大小(如果需要从一个 Set 中获得一个顺序列表，请用 TreeSet；)

```
public class SetPerformance {
    private static final int REPS = 200;
    private abstract static class Tester {
        String name;
        Tester(String name) {
            this.name = name;
        }
        abstract void test(Set s, int size);
    }
    private static Tester[] tests = { new Tester("add") {
        void test(Set s, int size) {
            for (int i = 0; i < REPS; i++) {
                s.clear();
                Collection1.fill(s, size);
            }
        }
```

```
    }, new Tester("contains") {
        void test(Set s, int size) {
            for (int i = 0; i < REPS; i++)
                for (int j = 0; j < size; j++)
                    s.contains(Integer.toString(j));
        }
    }, new Tester("iteration") {
        void test(Set s, int size) {
            for (int i = 0; i < REPS * 10; i++) {
                Iterator it = s.iterator();
                while (it.hasNext())
                    it.next();
            }
        }
    }, };
    public static void test(Set s, int size) {
        // A trick to print out the class name:
        System.out.println("Testing " + s.getClass().getName() + " size "
                + size);
        Collection1.fill(s, size);
        for (int i = 0; i < tests.length; i++) {
            System.out.print(tests[i].name);
            long t1 = System.currentTimeMillis();
            tests[i].test(s, size);
            long t2 = System.currentTimeMillis();
            System.out.println(": " + ((double) (t2 - t1) / (double) size));
        }
    }
    public static void main(String[] args) {
        // Small:
        test(new TreeSet(), 10);
        test(new HashSet(), 10);
        // Medium:
        test(new TreeSet(), 100);
        test(new HashSet(), 100);
        // Large:
        test(new HashSet(), 1000);
        test(new TreeSet(), 1000);
    }
}
```

进行 add()以及 contains()操作时，HashSet 显然要比 ArraySet 出色得多，而且性能明显与元素的多寡关系不大。一般编写程序的时候，几乎永远用不着使用 ArraySet。

3. 决定使用何种 Map

选择不同的 Map 实施方案时，注意 Map 的大小对于性能的影响是最大的，下面这个测试程序清楚地阐示了这一点：

```
public class MapPerformance {
    private static final int REPS = 200;
    public static Map fill(Map m, int size) {
        for (int i = 0; i < size; i++) {
            String x = Integer.toString(i);
            m.put(x, x);
        }
        return m;
    }
    private abstract static class Tester {
        String name;
        Tester(String name) {
            this.name = name;
        }
        abstract void test(Map m, int size);
    }
    private static Tester[] tests = { new Tester("put") {
        void test(Map m, int size) {
            for (int i = 0; i < REPS; i++) {
                m.clear();
                fill(m, size);
            }
        }
    }, new Tester("get") {
        void test(Map m, int size) {
            for (int i = 0; i < REPS; i++)
                for (int j = 0; j < size; j++)
                    m.get(Integer.toString(j));
        }
    }, new Tester("iteration") {
        void test(Map m, int size) {
            for (int i = 0; i < REPS * 10; i++) {
```

```
            Iterator it = m.entries().iterator();
            while (it.hasNext())
                it.next();
        }
    }
}, };
public static void test(Map m, int size) {
    // A trick to print out the class name:
    System.out.println("Testing " + m.getClass().getName() + " size "
            + size);
    fill(m, size);
    for (int i = 0; i < tests.length; i++) {
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(m, size);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + ((double) (t2 - t1) / (double) size));
    }
}
public static void main(String[] args) {
    // Small:
    test(new Hashtable(), 10);
    test(new HashMap(), 10);
    test(new TreeMap(), 10);
    // Medium:
    test(new Hashtable(), 100);
    test(new HashMap(), 100);
    test(new TreeMap(), 100);
    // Large:
    test(new HashMap(), 1000);
    test(new Hashtable(), 1000);
    test(new TreeMap(), 1000);
}
}
```

由于 Map 的大小是最严重的问题，所以程序的计时测试按 Map 的大小（或容量）来分割时间，以便得到令人信服的测试结果。下面列出一系列结果（在你的机器上可能不同）：

即使大小为 10， ArrayMap 的性能也要比 HashMap 差—— 除反复循环时以外。而在使用 Map 时，反复的作用通常并不重要（get()通常是我们时间花得最多的地方）。TreeMap 提供了出色的 put()以及反复时间，但 get()的性能并不佳。但是，我们为什么仍

然需要使用 TreeMap 呢？这样一来，我们可以不把它作为 Map 使用，而作为创建顺序列表的一种途径。**一旦填充了一个 TreeMap，就可以调用 keySet()来获得键的一个 Set“景象”。然后用 toArray()产生包含了那些键的一个数组。随后，可用 static 方法 Array.binarySearch()快速查找排好序的数组中的内容。**当然，也许只有在 HashMap 的行为不可接受的时候，才需要采用这种做法。因为 HashMap 的设计宗旨就是进行快速的检索操作。最后，**当我们使用 Map 时，首要的选择应该是 HashMap。只有在极少数情况下才需要考虑其他方法。**

```
public class MapCreation {
    public static void main(String[] args) {
        final long REPS = 100000;
        long t1 = System.currentTimeMillis();
        System.out.print("Hashtable");
        for (long i = 0; i < REPS; i++)
            new Hashtable();
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
        t1 = System.currentTimeMillis();
        System.out.print("TreeMap");
        for (long i = 0; i < REPS; i++)
            new TreeMap();
        t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
        t1 = System.currentTimeMillis();
        System.out.print("HashMap");
        for (long i = 0; i < REPS; i++)
            new HashMap();
        t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
}
```

TreeMap 的创建速度比其他两种类型明显快得多（但你应亲自尝试一下，因为据说新版本可能会改善 ArrayMap 的性能）。考虑到这方面的原因，同时由于前述 TreeMap 出色的 put()性能，所以如 果需要创建大量 Map，而且只有在以后才需要涉及大量检索操作，那么最佳的策略就是：**创建和填充 TreeMap；以后检索量增大的时候，再将重要的 TreeMap 转换成 HashMap—— 使用 HashMap(Map)构建器。**

未支持的操作

利用 static（静态）数组 Arrays.toList()，也许能将一个数组转换成 List。

```
public class Unsupported {
    private static String[] s = { "one", "two", "three", "four", "five", "six",
            "seven", "eight", "nine", "ten", };
    static List a = Arrays.toList(s);
    static List a2 = Arrays.toList(new String[] { s[3], s[4], s[5] });
    public static void main(String[] args) {
        Collection1.print(a); // Iteration
        System.out.println("a.contains(" + s[0] + ") = " + a.contains(s[0]));
        System.out.println("a.containsAll(a2) = " + a.containsAll(a2));
        System.out.println("a.isEmpty() = " + a.isEmpty());
        System.out.println("a.indexOf(" + s[5] + ") = " + a.indexOf(s[5]));
        // Traverse backwards:
        ListIterator lit = a.listIterator(a.size());
        while (lit.hasPrevious())
            System.out.print(lit.previous());
        System.out.println();
        // Set the elements to different values:
        for (int i = 0; i < a.size(); i++)
            a.set(i, "47");
        Collection1.print(a);
        // Compiles, but won't run:
        lit.add("X"); // Unsupported operation
        a.clear(); // Unsupported
        a.add("eleven"); // Unsupported
        a.addAll(a2); // Unsupported
        a.retainAll(a2); // Unsupported
        a.remove(s[0]); // Unsupported
        a.removeAll(a2); // Unsupported
    }
}
```

从中可以看出，实际只实现了 Collection 和 List 接口的一部分。剩余的方法导致了不受欢迎的一种情况，名为 UnsupportedOperationException。

在实现那些接口的集合类中，或者提供、或者没有提供对那些方法的支持。若调用一个未获支持的方法，就会导致一个 UnsupportedOperationException（操作未支持违例），这表明出现了一个编程错误。

Arrays.toList()产生了一个 List（列表），该列表是由一个固定长度的数组后推出来的。因此唯一能够支持的就是那些不改变数组长度的操作。在另一方面，若请求一个新接口表达不同种类的行为（可能叫作“ FixedSizeList” —— 固定长度列表），就有遭遇更大的复杂程度的危险。这样一来，以后试图使用库的时候，很快就会发现自己不知从何处下手。

对那些采用 Collection，List，Set 或者 Map 作为参数的方法，它们的文档应当指出哪些可选的方法是必须实现的。举个例子来说，排序要求实现 set()和 Iterator.set()方法，但不包括 add()和 remove()。