11:对象的集合

如果程序的对象数量有限，且寿命可知，那么这个程序是相当简单的。

一般来说，程序都是根据具体情况在不断地创建新的对象，而这些情况又只有在程序运行的时候才能确定。不到运行时你是不会知道你到底需要多少对象，甚至是什么类型的对象。为了解决这种常见的编程问题，你得有办法能在任何时间，任何地点，创建任何数量的对象。所以你不能指望用命名的 reference 来持有每个对象：

```
MyObject myReference;
```

原因就在于，你不可能知道究竟需要多少这样的对象。

针对这个相当关键的问题，绝大多数语言都提供了某种解决办法。Java 也提供了好几种持有对象(或者更准确的说，是对象的 reference) 的方法。我们前面讨论的数组是语言内置的数据类型。此外，Java 的工具类库还包括一套比较完整的容器类(*container classes* 也被称为 *collection classes*，但是由于 **Collection** 被 Java 2 用来命名类库的某个子集，所以我还是用概括性更强的术语"container")。它提供了复杂而精致的方法来持有甚至是操控你的对象。

数组

我们已经在第 4 章的最后部分，对数组的绝大多数内容作了必要的介绍，此外我们还演示了如何定义和初始化一个数组。本章的所关注的问题是“持有对象”，而数组只是其中的一个方法。那么数组又是凭什么要我们如此重视的呢?

数组与其它容器的区别体现在三个方面：效率，类型识别以及可以持有 primitives。数组是 Java 提供的，能随机存储和访问 reference 序列的诸多方法中的，最高效的一种。数组是一个简单的线性序列，所以它可以快速的访问其中的元素。但是速度是有代价的；当你创建了一个数组之后，它的容量就固定了，而且在其生命周期里不能改变。也许你会提议先创建一个数组，等到快不够用的时候，再创建一个新的，然后将旧数组里的 reference 全部导到新的里面。其实(我们以后会讲的)**ArrayList** 就是这么做的。但是这种灵活性所带来的开销，使得 **ArrayList** 的效率比起数组有了明显下降。

C++的 **vector** 容器类确实能知道它到底持有了什么类型的对象，但是与 Java 的数组相比，它又有另一个缺点：C++ **vector** 的**[]**运算符不

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

作边界检查，所以你可能会不知不觉就过了界[1]。而 Java 对数组和容器都做边界检查；如果过了界，它就会给一个 **RuntimeException**。这种异常表明这个错误是由程序员造成的，这样你就用不着再在程序里面检查了(译者注：这里的原文有些模棱两可，我想他要表达的意思可能是以下两种中的一个。一是指，你不用再作边界检查了；二是指，通过异常信息，你就可以知道错误是由过界造成的，因而不用查源代码了)。顺便说一下，C++的 **vector** 不作边界检查是为了速度；而在 Java，不论是数组或是用容器，你都得面对边界检查所带来的固定的性能下降。

本章所探讨的其它泛型容器类还包括 **List**，**Set** 和 **Map**。它们处理对象的时候就好像这些对象都没有自己的具体类型一样。也就是说，容器将它所含的元素都看成是(Java 中所有类的根类)**Object** 的。这样你只需创建一种容器，就能把所有类型的对象全都放进去。从这个角度来看，这种做法很不错(只是苦了 primitive。如果是常量，你还可以用 Java 的 primitive 的 wrapper 类；如果是变量，那就只能放到你自己的类里了)。与其他泛型容器相比，这里体现出数组的第二个优势：创建数组的时候，你也同时指明了它所持有的对象的类型(这又引出了第三点 —— 数组可以持有 primitives，而容器却不行)。也就是说，它会在编译的时候作类型检查，从而防止你插入错误类型的对象，或者是在提取对象的时候把对象的类型给搞错了。Java 在编译和运行时都能阻止你将一个不恰当的消息传给对象。所以这并不是说使用容器就有什么危险，只是如果编译器能够帮你指定，那么程序运行会更快，最终用户也会较少受到程序运行异常的骚扰。

从效率和类型检查的角度来看，使用数组总是没错的。但是，如果你在解决一个更为一般的问题，那数组就会显得功能太弱了点。本章先讲数组，然后集中精力讨论 Java 的容器类。

数组是第一流的对象

不管你用的是那种类型的数组，数组的标识符实际上都是一个“创建在堆(heap)里的实实在在的对象的”reference。实际上是那个对象持有其他对象的 reference。你既可以用数组的初始化语句，隐含地创建这个对象，也可以用 new 表达式，明确地创建这个对象。只读的 **length** 属性能告诉你数组能存储多少元素。它是数组对象的一部分(实际上也是你唯一能访问的属性或方法)。‘**[]**’语法是另一条访问数组对象的途径。

下面这段程序演示了几种初始化数组的办法，以及如何将数组的 reference 赋给不同的数组对象。此外，它还显示了，对象数组和 primitives 数组在使用方法上几乎是完全相同。唯一的不同是，对象数组持有 reference，而 primitive 数组则直接持有值。

```
//: c11:ArraySize.java
```

[1]不过真的想知道 **vector** 的容量有多大，还是有办法的，更何况 **at( )**方法确实做边界检查。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
// Initialization & re-assignment of arrays.
import com.bruceeckel.simpletest.*;

class Weeble {} // A small mythical creature

public class ArraySize {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    // Arrays of objects:
    Weeble[] a; // Local uninitialized variable
    Weeble[] b = new Weeble[5]; // Null references
    Weeble[] c = new Weeble[4];
    for(int i = 0; i < c.length; i++)
      if(c[i] == null) // Can test for null
reference
        c[i] = new Weeble( );
    // Aggregate initialization:
    Weeble[] d = {
      new Weeble( ), new Weeble( ), new Weeble( )
    };
    // Dynamic aggregate initialization:
    a = new Weeble[] {
      new Weeble( ), new Weeble( )
    };
    System.out.println("a.length=" + a.length);
    System.out.println("b.length = " + b.length);
    // The references inside the array are
    // automatically initialized to null:
    for(int i = 0; i < b.length; i++)
      System.out.println("b[" + i + "]=" + b[i]);
    System.out.println("c.length = " + c.length);
    System.out.println("d.length = " + d.length);
    a = d;
    System.out.println("a.length = " + a.length);

    // Arrays of primitives:
    int[] e; // Null reference
    int[] f = new int[5];
    int[] g = new int[4];
    for(int i = 0; i < g.length; i++)
      g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Compile error: variable e not initialized:
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // The primitives inside the array are
    // automatically initialized to zero:
    for(int i = 0; i < f.length; i++)
      System.out.println("f[" + i + "]=" + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[] { 1, 2 };
    System.out.println("e.length = " + e.length);
    monitor.expect(new String[] {
      "a.length=2",
      "b.length = 5",
      "b[0]=null",
      "b[1]=null",
```

```
          "b[2]=null",
          "b[3]=null",
          "b[4]=null",
          "c.length = 4",
          "d.length = 3",
          "a.length = 3",
          "f.length = 5",
          "f[0]=0",
          "f[1]=0",
          "f[2]=0",
          "f[3]=0",
          "f[4]=0",
          "g.length = 4",
          "h.length = 3",
          "e.length = 3",
          "e.length = 2"
        });
    }
} ///:~
```

数组 **a** 是一个尚未初始化的局部变量，在你将其正确地初始化之前，编译器禁止你对这个 reference 作任何事情。数组 **b** 是一个已经进行了初始化的数组，它被连到了一个“**Weeble** 对象的 reference”的数组，只是这个数组里面还没有真正放上 **Weeble** 对象。但是由于 **b** 指向的是一个合法的对象，所以你已经可以查询其容量大小了。这就带来一个小问题：你没法知道数组里面究竟**放了**多少元素，因为 **length** 只是告诉你数组**能放**多少元素，也就是说是数组对象的容量，而不是它真正已经持有的元素的数量。但是，创建数组对象的时候，它所持有的 reference 都会被自动地初始化为 **null**，所以你可以通过检查数组的某个“槽位”是否为 **null**，来判断它是否持有对象。以此类推，primitive 的数组，会自动将数字初始化为零，字符初始化为**(char)0**，**boolean** 初始化为 **false**。

数组 **c** 演示了数组对象的创建，随后它直接用 **Weeble** 对象对数组各个“槽位”进行赋值。数组 **d** 就是所谓“总体初始化(aggregate initialization)”的语法，它只用一条语句，就创建了数组对象(隐含地使用了 **new**，就像对数组 **c**)，并且用 **Weeble** 对象进行了初始化。

下一个数组的初始化可以被理解为“动态的总体初始化(dynamic aggregate initialization)”。**d** 所使用的“总体初始化”语句，只能在定义 **d** 的时候用。但是用这种语法，你就可以在任何地方创建和初始化数组对象。比方说，假设 **hide( )**是一个使用 **Weeble** 对象的数组做参数的方法。那么，你可以这样调用：

```
hide(d);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

但是你也可以动态地创建一个数组，把它当作参数传给 **hide( )**：

```
hide(new Weeble[] { new Weeble( ), new Weeble( ) });
```

在很多情况下，这能给你的编程带来便利。

表达式：

```
a = d;
```

演示了如何将一个 reference 指向另一个数组对象。这么做跟使用其他对象的 refernce 没什么两样。现在 **a** 和 **d** 都指向堆中的同一个数组对象。

ArraySize.java 的第二部分应征了 primitive 数组的工作方式和对象数组的几乎一摸一样。**只是**它能直接持有 primitive 的值。

primitive 的容器

容器类只能持有 **Object** 对象的 reference。而数组除了能持有 **Objects** 的 reference 之外，还可以直接持有 primitive。当然**可以**使用诸如 **Integer**，**Double** 之类的 wrapper 类，把 primitive 的值放到容器中，但这样总有点怪怪的。此外，primitive 数组的效率要比 wrapper 类容器的高出许多。

当然，如果你使用 primitive 的时候，还需要那种“能随需要自动扩展的”容器类的灵活性，那就不能用数组了。你只能用容器来存储 primitive 的 wrapper 类。也许你会想，应该为每种 primitive 都提供一个 **ArrayList**，但是遗憾的是 Java 没为你准备。[2]

返回一个数组

假设你写了一个方法，它返回的不是一个而是一组东西。在 C 和 C++之类的语言里，这件事就有些难办了。因为你不能返回一个数组，你只能返回一个指向数组的指针。由于要处理“控制数组生命周期”之类的麻烦事，这么做很容易会出错，最后导致内存泄漏。

Java 采取了类似的解决方案，但是不同之处在于，它返回的“就是一个数组”。与 C++不同，你永远也不必为 Java 的数组操心——只要你还需要它，它就还在；一旦你用完了，垃圾回收器会帮你把它打扫干净。

[2]这就是 C++明显比 Java 强的地方了，因为它的 **template** 关键词支持参数化类型(*parameterized type*)。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

下面的例子演示了如何返回一个 **String** 数组：

```
//: c11:IceCream.java
// Returning arrays from methods.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class IceCream {
  private static Test monitor = new Test( );
  private static Random rand = new Random( );
  public static final String[] flavors = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
    "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  public static String[] flavorSet(int n) {
    String[] results = new String[n];
    boolean[] picked = new boolean[flavors.length];
    for(int i = 0; i < n; i++) {
      int t;
      do
        t = rand.nextInt(flavors.length);
      while(picked[t]);
      results[i] = flavors[t];
      picked[t] = true;
    }
    return results;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
      System.out.println(
        "flavorSet(" + i + ") = ");
      String[] fl = flavorSet(flavors.length);
      for(int j = 0; j < fl.length; j++)
        System.out.println("\t" + fl[j]);
      monitor.expect(new Object[] {
        "%% flavorSet\\(\\d+\\) = ",
        new TestExpression("%%
\\t(Chocolate|Strawberry|"
          + "Vanilla Fudge Swirl|Mint Chip|Mocha
Almond "
          + "Fudge|Rum Raisin|Praline Cream|Mud
Pie)", 8)
      });
    }
  }
} ///:~
```

flavorSet( ) 创建了一个名为 **results** 的 **String** 数组。这个数组的容量是由传给它的参数 **n** 所决定的。接下来它从 **flavors** 数组里面随机选择 flavors(译者注：表示冰激凌的口味)，放到 **results** 中，最后返回 **results**。返回数组跟返回对象没什么两样——都是一个 reference。因此数组是不是在 **flavorSet( )**或是其他什么地方创建的并不重要。只要

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

你还需要，它就不会消失；一旦你用完了，垃圾回收器负责帮你收拾干净。

再附带说一下，**flavorSet( )**随机选择 flavors 的时候，会检查那个 flavor 以前是不是没被选中过。这是由 **do** 循环做的。它不断的作随机选择，直到找到一个在 **picked** 数组中还没有的。(当然，也可以用“比较 **String**”的方式来检查随机选出的 flavor 是不是已经在 **results** 数组中了。)如果成功，它会添加这条记录，然后寻找下一个(递增 **i**)。

main( )会打印出 20 套 flavors，这样你就能看到，每次 **flavorSet( )** 都是随机选择 flavors 的。如果把输出导到文件你就能看得更清楚。只是记着看文件的时候，你只是想挑一个而不是真的想吃冰激凌。

Arrays 类

java.util 里面有一个 **Arrays** 类，它包括了一组可用于数组的 **static** 方法，这些方法都是一些实用工具。其中有四个基本方法：用来比较两个数组是否相等的 **equals( )**；用来填充数组的 **fill( )**；用来对数组进行排序的 **sort( )**；以及用于在一个已排序的数组中查找元素的 **binarySearch( )**。所有这些方法都对 primitive 和 **Object** 进行了重载。此外还有一个 **asList( )**方法，它接受一个数组，然后把它转成一个 **List** 容器。后面你会学到。

虽然 **Arrays** 还是有用的，但它的功能并不完整。举例来说，如果它能让我们不用写 **for** 循环就能直接打印数组，那就好了。此外，正如你所看到的，**fill( )**只能用一个值填数组。所以，如果如果你想把随机生成的数字填进数组的话，**fill( )**是无能为力的。

因此为 **Arrays** 类提供一些额外的功能还是有意义的。为方便起鉴，我把它放到 **package com.bruceeckel.util** 里。它可以打印任何类型的数组；并且用“你定义的 *generator* 对象生成的”值或对象填充一个数组。

由于要为各种 primitive 以及 **Object** 服务，程序里有大量的几乎是重复的代码。[3]比如，**next( )**必须根据不同的情况返回不同的类型，所以每种类型都需要一个“generator” 接口。

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;
public interface Generator { Object next( ); } ///:~
```

[3] C++的程序员会觉得，如果能用默认参数和模板的话，可以少写很多代码。而 Python 的程序员则会认为，这个类库本身就是多余的。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;
public interface BooleanGenerator { boolean
next( ); } ///:~
```

```
//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;
public interface ByteGenerator { byte next( ); }
///:~
```

```
//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
public interface CharGenerator { char next( ); }
///:~
```

```
//: com:bruceeckel:util:ShortGenerator.java
package com.bruceeckel.util;
public interface ShortGenerator { short next( ); }
///:~
```

```
//: com:bruceeckel:util:IntGenerator.java
package com.bruceeckel.util;
public interface IntGenerator { int next( ); } ///:~
```

```
//: com:bruceeckel:util:LongGenerator.java
package com.bruceeckel.util;
public interface LongGenerator { long next( ); }
///:~
```

```
//: com:bruceeckel:util:FloatGenerator.java
package com.bruceeckel.util;
public interface FloatGenerator { float next( ); }
///:~
```

```
//: com:bruceeckel:util:DoubleGenerator.java
package com.bruceeckel.util;
public interface DoubleGenerator { double next( ); }
///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Arrays2 包含了很多 **toString( )**方法以重载各种类型。这些方法能让你很方便地打印出一个数组。**toString( )**方法用了 **StringBuffer** 而不是 **String** 对象。这是出于运行效率的考虑；当你需要重复调用一个方法以组装字符串的时候，较为明智的选择还是使用效率更高的 **StringBuffer**，而不是更方便的 **String**。这里，创建 **StringBuffer** 的时候用了一个初始值，然后再它后面接 **String**。最后，把 **result** 转换成 **String** 再返回：

```
//: com:bruceeckel:util:Arrays2.java
// A supplement to java.util.Arrays, to provide
additional
// useful functionality when working with arrays.
Allows
// any array to be converted to a String, and to be
filled
// via a user-defined "generator" object.
package com.bruceeckel.util;
import java.util.*;

public class Arrays2 {
  public static String toString(boolean[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString( );
  }
  public static String toString(byte[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString( );
  }
  public static String toString(char[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString( );
  }
  public static String toString(short[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    result.append("]");
    return result.toString( );
  }
  public static String toString(int[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString( );
  }
  public static String toString(long[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString( );
  }
  public static String toString(float[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString( );
  }
  public static String toString(double[] a) {
    StringBuffer result = new StringBuffer("[");
    for(int i = 0; i < a.length; i++) {
      result.append(a[i]);
      if(i < a.length - 1)
        result.append(", ");
    }
    result.append("]");
    return result.toString( );
  }
  // Fill an array using a generator:
  public static void fill(Object[] a, Generator gen)
{
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(Object[] a, int from, int to, Generator gen)
{
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void
  fill(boolean[] a, BooleanGenerator gen) {
      fill(a, 0, a.length, gen);
  }
  public static void
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  fill(boolean[] a, int from, int
to,BooleanGenerator gen){
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void fill(byte[] a, ByteGenerator
gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(byte[] a, int from, int to, ByteGenerator gen)
{
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void fill(char[] a, CharGenerator
gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(char[] a, int from, int to, CharGenerator gen)
{
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void fill(short[] a, ShortGenerator
gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(short[] a, int from, int to, ShortGenerator
gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void fill(int[] a, IntGenerator gen)
{
      fill(a, 0, a.length, gen);
  }
  public static void
  fill(int[] a, int from, int to, IntGenerator gen)
{
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void fill(long[] a, LongGenerator
gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(long[] a, int from, int to, LongGenerator gen)
{
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void fill(float[] a, FloatGenerator
gen) {
    fill(a, 0, a.length, gen);
  }
  public static void
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  fill(float[] a, int from, int to, FloatGenerator
gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  public static void fill(double[] a,
DoubleGenerator gen){
    fill(a, 0, a.length, gen);
  }
  public static void
  fill(double[] a, int from, int to, DoubleGenerator
gen) {
    for(int i = from; i < to; i++)
      a[i] = gen.next( );
  }
  private static Random r = new Random( );
  public static class
  RandBooleanGenerator implements BooleanGenerator {
    public boolean next( ) { return
r.nextBoolean( ); }
  }
  public static class
  RandByteGenerator implements ByteGenerator {
    public byte next( ) { return
(byte)r.nextInt( ); }
  }
  private static String ssource =

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxy
z";
  private static char[] src = ssource.toCharArray( );
  public static class
  RandCharGenerator implements CharGenerator {
    public char next( ) {
      return src[r.nextInt(src.length)];
    }
  }
  public static class
  RandStringGenerator implements Generator {
    private int len;
    private RandCharGenerator cg = new
RandCharGenerator( );
    public RandStringGenerator(int length) {
      len = length;
    }
    public Object next( ) {
      char[] buf = new char[len];
      for(int i = 0; i < len; i++)
        buf[i] = cg.next( );
      return new String(buf);
    }
  }
  public static class
  RandShortGenerator implements ShortGenerator {
    public short next( ) { return
(short)r.nextInt( ); }
  }
  public static class
  RandIntGenerator implements IntGenerator {
    private int mod = 10000;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    public RandIntGenerator( ) {}
    public RandIntGenerator(int modulo) { mod =
modulo; }
    public int next( ) { return r.nextInt(mod); }
  }
  public static class
  RandLongGenerator implements LongGenerator {
    public long next( ) { return r.nextLong( ); }
  }
  public static class
  RandFloatGenerator implements FloatGenerator {
    public float next( ) { return r.nextFloat( ); }
  }
  public static class
  RandDoubleGenerator implements DoubleGenerator {
    public double next( ) {return r.nextDouble( );}
  }
} ///:~
```

要想用生成器(generator)来填满数组，就要传一个合适的 **interface** 的 reference 给 **fill( )**方法。这个接口应该有一个能生成正确类型的对象的 **next( )**方法(由接口如何实现来决定)。**fill( )**方法只是简单地调用 **next( )**，直到填满所需的范围。现在你可以通过实现合适的 **interface** 来创建 generator，然后用 **fill( )**来使用这个 generator。

测试的时候随机数生成器就会很有用了，所以除了用 **String** 生成器来代表 **Object** 之外，我们还创建了一整套内部类来实现所有 primitive 生成器的接口。你会看到 **RandStringGenerator** 用 **RandCharGenerator** 来填充一个 char 的数组，然后再把这个数组转换成 **String**。这个数组的大小是由构造函数的参数所决定的。

默认情况下，为了不让生成的数字太大，**RandIntGenerator** 会对 10，000 取模。但是重载的构造函数允许你选择一个更小的数值。

下面的程序在测试类库的同时还示范了该如何使用类库。

```
//: c11:TestArrays2.java
// Test and demonstrate Arrays2 utilities.
import com.bruceeckel.util.*;

public class TestArrays2 {
  public static void main(String[] args) {
    int size = 6;
    // Or get the size from the command line:
    if(args.length != 0) {
      size = Integer.parseInt(args[0]);
      if(size < 3) {
        System.out.println("arg must be >= 3");
        System.exit(1);
      }
    }
    boolean[] a1 = new boolean[size];
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        Arrays2.fill(a1, new
    Arrays2.RandBooleanGenerator( ));
        System.out.println("a1 = " +
    Arrays2.toString(a1));
        Arrays2.fill(a2, new
    Arrays2.RandByteGenerator( ));
        System.out.println("a2 = " +
    Arrays2.toString(a2));
        Arrays2.fill(a3, new
    Arrays2.RandCharGenerator( ));
        System.out.println("a3 = " +
    Arrays2.toString(a3));
        Arrays2.fill(a4, new
    Arrays2.RandShortGenerator( ));
        System.out.println("a4 = " +
    Arrays2.toString(a4));
        Arrays2.fill(a5, new
    Arrays2.RandIntGenerator( ));
        System.out.println("a5 = " +
    Arrays2.toString(a5));
        Arrays2.fill(a6, new
    Arrays2.RandLongGenerator( ));
        System.out.println("a6 = " +
    Arrays2.toString(a6));
        Arrays2.fill(a7, new
    Arrays2.RandFloatGenerator( ));
        System.out.println("a7 = " +
    Arrays2.toString(a7));
        Arrays2.fill(a8, new
    Arrays2.RandDoubleGenerator( ));
        System.out.println("a8 = " +
    Arrays2.toString(a8));
      }
    } ///:~
```

size 参数有一个默认的值，不过你也可以通过命令行来设定。

填满一个数组

Java 标准类库 **Arrays** 也包括了一个 **fill( )**方法，但是它太简单了；它只是简单的把一个的值复制到数组各个位置，如果是对象，则将相同的 reference 拷贝到每个位置。可以用 **Arrays2.toString( )**把 **Arrays.fill( )**的工作方式清清楚楚地显示出来：

```
//: c11:FillingArrays.java
// Using Arrays.fill( )
import com.bruceeckel.simpletest.*;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
import com.bruceeckel.util.*;
import java.util.*;

public class FillingArrays {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    int size = 6;
    // Or get the size from the command line:
    if(args.length != 0)
      size = Integer.parseInt(args[0]);
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays.fill(a1, true);
    System.out.println("a1 = " +
Arrays2.toString(a1));
    Arrays.fill(a2, (byte)11);
    System.out.println("a2 = " +
Arrays2.toString(a2));
    Arrays.fill(a3, 'x');
    System.out.println("a3 = " +
Arrays2.toString(a3));
    Arrays.fill(a4, (short)17);
    System.out.println("a4 = " +
Arrays2.toString(a4));
    Arrays.fill(a5, 19);
    System.out.println("a5 = " +
Arrays2.toString(a5));
    Arrays.fill(a6, 23);
    System.out.println("a6 = " +
Arrays2.toString(a6));
    Arrays.fill(a7, 29);
    System.out.println("a7 = " +
Arrays2.toString(a7));
    Arrays.fill(a8, 47);
    System.out.println("a8 = " +
Arrays2.toString(a8));
    Arrays.fill(a9, "Hello");
    System.out.println("a9 = " + Arrays.asList(a9));
    // Manipulating ranges:
    Arrays.fill(a9, 3, 5, "World");
    System.out.println("a9 = " + Arrays.asList(a9));
    monitor.expect(new String[] {
      "a1 = [true, true, true, true, true, true]",
      "a2 = [11, 11, 11, 11, 11, 11]",
      "a3 = [x, x, x, x, x, x]",
      "a4 = [17, 17, 17, 17, 17, 17]",
      "a5 = [19, 19, 19, 19, 19, 19]",
      "a6 = [23, 23, 23, 23, 23, 23]",
      "a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]",
      "a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]",
      "a9 = [Hello, Hello, Hello, Hello, Hello,
Hello]",
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        "a9 = [Hello, Hello, Hello, World, World,
Hello]"
    });
  }
} ///:~
```

你可以填满整个数组，或者像最后两句那样，只填其中的某个范围。相比只能用一个值的 **Arrays.fill( )**，**Arrays2.fill( )**的运行结果就更有趣一些了。

复制一个数组

Java 标准类库提供了一个 **System.arraycopy( )**的 **static** 方法。相比 **for** 循环，它能以更快的速度拷贝数组。**System.arraycopy( )**对所有类型都作了重载。下面就是一个用它来处理 **int** 数组的例子。

```
//: c11:CopyingArrays.java
// Using System.arraycopy( )
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class CopyingArrays {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    int[] i = new int[7];
    int[] j = new int[10];
    Arrays.fill(i, 47);
    Arrays.fill(j, 99);
    System.out.println("i = " + Arrays2.toString(i));
    System.out.println("j = " + Arrays2.toString(j));
    System.arraycopy(i, 0, j, 0, i.length);
    System.out.println("j = " + Arrays2.toString(j));
    int[] k = new int[5];
    Arrays.fill(k, 103);
    System.arraycopy(i, 0, k, 0, k.length);
    System.out.println("k = " + Arrays2.toString(k));
    Arrays.fill(k, 103);
    System.arraycopy(k, 0, i, 0, k.length);
    System.out.println("i = " + Arrays2.toString(i));
    // Objects:
    Integer[] u = new Integer[10];
    Integer[] v = new Integer[5];
    Arrays.fill(u, new Integer(47));
    Arrays.fill(v, new Integer(99));
    System.out.println("u = " + Arrays.asList(u));
    System.out.println("v = " + Arrays.asList(v));
    System.arraycopy(v, 0, u, u.length/2, v.length);
    System.out.println("u = " + Arrays.asList(u));
    monitor.expect(new String[] {
      "i = [47, 47, 47, 47, 47, 47, 47]",
      "j = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]",
      "j = [47, 47, 47, 47, 47, 47, 47, 99, 99, 99]",
      "k = [47, 47, 47, 47, 47]",
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        "i = [103, 103, 103, 103, 103, 47, 47]",
        "u = [47, 47, 47, 47, 47, 47, 47, 47, 47, 47]",
        "v = [99, 99, 99, 99, 99]",
        "u = [47, 47, 47, 47, 47, 99, 99, 99, 99, 99]"
      });
    }
} ///:~
```

传给 **arraycopy( )**的参数包括源数组，标识从源数组的哪个位置开始拷贝的偏移量，目标数组，标识从目标数组的哪个位置开始拷贝的偏移量，以及要拷贝的元素的数量。当然超出数组边界会引发异常。

这个例子告诉我们对象数组和 primitive 数组都能拷贝。但是如果你拷贝的是对象数组，那么你只拷贝了它们的 reference —— 对象本身不会被拷贝。这被称为浅拷贝(*shallow copy*) (见附录 A)。

数组的比较

为了能比较数组是否完全相等，**Arrays** 提供了经重载的 **equals( )**方法。当然，也是针对各种 primitive 以及 **Object** 的。两个数组要想完全相等，它们必须有相同数量的元素，而且数组的每个元素必须与另一个数组的相对应的位置上的元素相等。元素的相等性，用 **eqauls( )**判断。(对于 primitive，它会使用其 wrapper 类的 **equals( )**；比如 int 使用 **Integer.equals( )**。)例如：

```
//: c11:ComparingArrays.java
// Using Arrays.equals( )
import com.bruceeckel.simpletest.*;
import java.util.*;

public class ComparingArrays {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    int[] a1 = new int[10];
    int[] a2 = new int[10];
    Arrays.fill(a1, 47);
    Arrays.fill(a2, 47);
    System.out.println(Arrays.equals(a1, a2));
    a2[3] = 11;
    System.out.println(Arrays.equals(a1, a2));
    String[] s1 = new String[5];
    Arrays.fill(s1, "Hi");
    String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
    System.out.println(Arrays.equals(s1, s2));
    monitor.expect(new String[] {
      "true",
      "false",
      "true"
    });
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

起先 **a1** 和 **a2** 是完全相等的，所以输出结果是“true”，但是我们修改了一个元素，于是结果就变成“false”了。在后一个例子里，**s1** 的所有元素指向相同的对象，但是 **s2** 却有五个各自独立的对象。但是数组是否相等是基于其内容，(通过 **Object.equals( )**)，因此结果仍是 "true"。

数组元素的比较

Java 1.0 与 1.1 的类库缺少一些重要特性，其中之一就有没有提供算法，甚至连个简单的排序都没有。对那些期望能有一个像样的标准类库的人来说，这简直是太令人不解了。还好 Java2 作了补救，至少是解决了排序问题。

编写泛型排序程序的时候会遇到一个问题，就是它只能根据对象的实际类型进行比较。当然为每种类型都写一个的排序程序也不啻是一个办法，但是这样一来你就会发现，碰到新的类型的时候，要想复用代码就不那么容易了。

设计的一项重要目标就是“要将会变和不会变的东西分开来”。这里，不会变的东西就是通用的排序算法，而会变的东西就是对象是怎样比较大小的。所以与其在各种排序程序里面都插进比较算法，还不如使用回调(*callback*)技术。有了回调，你就能把“会根据情况的不同而改变”的代码分离出来，而用相同的代码来调用那些会变的代码。

Java 里面有两种能让你实现比较功能的方法。一是实现 **java.lang.Comparable** 接口，并以此实现类“自有的”比较方法。这是一个很简单的接口，它只有一个方法 **compareTo( )**。这个方法能接受另一个对象作为参数，如果现有对象比参数小，它会返回一个负数，如果相同则返回零，如果现有的对象比参数大，它就返回一个正数。

下面就是一个实现 **Comparable** 接口的类，此外它还用 Java 标准类库的 **Arrays.sort( )**方法演示了比较的结果：

```
//: c11:CompType.java
// Implementing Comparable in a class.
import com.bruceeckel.util.*;
import java.util.*;

public class CompType implements Comparable {
  int i;
  int j;
  public CompType(int n1, int n2) {
    i = n1;
    j = n2;
  }
  public String toString( ) {
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    return "[i = " + i + ", j = " + j + "]";
  }
  public int compareTo(Object rv) {
    int rvi = ((CompType)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
  }
  private static Random r = new Random( );
  public static Generator generator( ) {
    return new Generator( ) {
      public Object next( ) {
        return new
CompType(r.nextInt(100),r.nextInt(100));
      }
    };
  }
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, generator( ));
    System.out.println(
      "before sorting, a = " + Arrays.asList(a));
    Arrays.sort(a);
    System.out.println(
      "after sorting, a = " + Arrays.asList(a));
  }
} ///:~
```

一旦你定义了比较的方法，你就得负责确定这个对象应该依据什么同其它对象进行比较。这里我们只用到了 **i** 的值，而 **j** 的值则被忽略了。

static randInt( )方法会生成一个介于 0 到 100 之间的正数，而 **generator( )**方法则用“创建匿名内部类”的方法(参见第八章)，创建了一个实现 **Generator** 接口的对象。而这个对象又会用 **randInt( )**生成的随机数创建了多个 **CompType** 对象。**main( )**用这个 generator 把 **CompType** 型的数组填满。接下来，开始对数组排序。如果 **CompType** 没有实现 **Comparable** 接口，那么程序运行调用到 **sort( )**的时候，就会引发一个 **ClassCastException** 错误。这是因为 **sort( )**会把传给它的参数转换成 **Comparable**。

现在假设，有人给你一个没有实现 **Comparable** 接口的类，或者这个类实现了 **Comparable** 接口，但是你发现它的工作方式不是你所希望的，于是要重新定义一个新的比较方法。Java 没有强求你一定要把比较代码塞进类里，它的解决方案是使用“策略模式(strategy design pattern)[4]”。有了策略之后，你就能把会变的代码封装到它自己的类里(即所谓的策略对象 strategy object)。你把策略对象交给不会变的代码，然后由它运用策略完成整个算法。这样，你就可以用不同的策略对象来表示不同的比较方式，然后把它们都交给同一个排序程序了。接下来就要“通过实现 **Comparator** 接口”来定义策略对象了。这个接口有两个

[4] *Design Patterns*, Erich Gamma 等著, Addison-Wesley 1995 年出版。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

方法 **compare( )**和 **equals( )**。但是除非是有特殊的性能要求，否则你用不着去实现 **equals( )**。因为只要是类，它就都隐含地继承自 **Object**，而 **Object** 里面已经有了一个 **equals( )**了。所以你尽可以使用缺省的 **Object** 的 **equals( )**，这样就已经满足接口的要求了。

(我们要等一会儿才会讲到的)**Collections** 类里专门有一个会返回与对象自有的比较法相反的 **Comparator** 的方法。它能很轻易地被用到 **CompType** 上面。

```
//: c11:Reverse.java
// The Collecions.reverseOrder( ) Comparator
import com.bruceeckel.util.*;
import java.util.*;

public class Reverse {
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, CompType.generator( ));
    System.out.println(
      "before sorting, a = " + Arrays.asList(a));
    Arrays.sort(a, Collections.reverseOrder( ));
    System.out.println(
      "after sorting, a = " + Arrays.asList(a));
  }
} ///:~
```

Collections.reverseOrder( )返回了一个 **Comparator** 的 reference。

再举个例子，我们让 **Comparator** 根据 **j**，而不是 **i** 的值来比较 **CompType** 对象。

```
//: c11:ComparatorTest.java
// Implementing a Comparator for a class.
import com.bruceeckel.util.*;
import java.util.*;

class CompTypeComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    int j1 = ((CompType)o1).j;
    int j2 = ((CompType)o2).j;
    return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
  }
}

public class ComparatorTest {
  public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, CompType.generator( ));
    System.out.println(
      "before sorting, a = " + Arrays.asList(a));
    Arrays.sort(a, new CompTypeComparator( ));
    System.out.println(
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        "after sorting, a = " + Arrays.asList(a));
  }
} ///:~
```

compare( )方法会根据第一个参数是小于，等于还是大于第二个参数，分别返回负整数，零或是正整数。

数组的排序

有了内置的排序方法之后，你就能对任何数组排序了，不论是 primitive 的还是对象数组，只要它实现了 **Comparable** 接口或有一个与之相关的 **Comparator** 对象就行了。这个功能填补了 Java 类库的一个大漏洞。信不信由你，Java1.0 和 1.1 连字符串的排序都没有。下面是一个随机生成 **String** 并对其排序的程序：

```
//: c11:StringSorting.java
// Sorting an array of Strings.
import com.bruceeckel.util.*;
import java.util.*;

public class StringSorting {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa, new
Arrays2.RandStringGenerator(5));
    System.out.println(
      "Before sorting: " + Arrays.asList(sa));
    Arrays.sort(sa);
    System.out.println(
      "After sorting: " + Arrays.asList(sa));
  }
} ///:~
```

你可以从程序的输出看到，这个算法是按字典顺序进行排序的。就是把首字母大写的单词放在前面，小写字母开头的放在后面。(电话簿就是这么排的。)或许你想忽略大小写进行排序，那就自己定义一个 **Comparator** 类吧，只要覆写默认的 **String Comparable** 的行为就可以了。考虑到将来的复用，我们把这个类放进了“util”package：

```
//: com:bruceeckel:util:AlphabeticComparator.java
// Keeping upper and lowercase letters together.
package com.bruceeckel.util;
import java.util.*;

public class AlphabeticComparator implements
Comparator {
  public int compare(Object o1, Object o2) {
    String s1 = (String)o1;
    String s2 = (String)o2;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    return
s1.toLowerCase( ).compareTo(s2.toLowerCase( ));
  }
} ///:~
```

程序开头做了个类型转换，这样如果你传了个错误的类型，它就会抛出异常。字符串会先被转换成小写，再进行比较。接下来用 **String** 内置的 **compareTo( )**方法进行比较。

下面就是 **AlphabeticComparator** 的测试代码：

```
//: c11:AlphabeticSorting.java
// Keeping upper and lowercase letters together.
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSorting {
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa, new
Arrays2.RandStringGenerator(5));
    System.out.println(
      "Before sorting: " + Arrays.asList(sa));
    Arrays.sort(sa, new AlphabeticComparator( ));
    System.out.println(
      "After sorting: " + Arrays.asList(sa));
  }
} ///:~
```

Java 标准类库所用的排序算法已经作了优化——对 primitive，它用的是“快速排序(Quicksort)”，对对象，它用的是“稳定合并排序(stable merge sort)”。所以除非是 prolier 表明排序算法是瓶颈，否则你不用为性能担心。

查询有序数组

一旦数组排完序，你就能用 **Arrays.binarySearch( )**进行快速查询了。但是切忌对一个尚未排序的数组使用 **binarySearch( )**；因为这么做的结果是没意义的。接下来，我们用 **RandIntGenerator** 来填数组，然后用同一个 generator 生成一个随机数，再在数组里面找这个数字：

```
//: c11:ArraySearching.java
// Using Arrays.binarySearch( ).
import com.bruceeckel.util.*;
import java.util.*;

public class ArraySearching {
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  public static void main(String[] args) {
    int[] a = new int[100];
    Arrays2.RandIntGenerator gen =
      new Arrays2.RandIntGenerator(1000);
    Arrays2.fill(a, gen);
    Arrays.sort(a);
    System.out.println(
      "Sorted array: " + Arrays2.toString(a));
    while(true) {
      int r = gen.next( );
      int location = Arrays.binarySearch(a, r);
      if(location >= 0) {
        System.out.println("Location of " + r +
          " is " + location + ", a[" +
          location + "] = " + a[location]);
        break; // Out of while loop
      }
    }
  }
} ///:~
```

while 循环会不断地生成随机数，直到它找到为止。

如果 **Arrays.binarySearch( )**找到了，它就返回一个大于或等于 0 的值。否则它就返回一个负值，而这个负值要表达的意思是，如果你手动维护这个数组的话，这个值应该插在哪个位置。这个值就是：

```
-(插入点)-1
```

“插入点”就是，在所有“比要找的那个值”更大值中，最小的那个值的下标，或者，如果数组中所有的值都比要找的值小，它就是 **a.size( )**。

如果数组里面有重复元素，那它不能保证会返回哪一个。这个算法不支持重复元素，不过它也不报错。所以，如果你需要的是一个无重复元素的有序序列的话，那么可以考虑使用本章后面所介绍的 **TreeSet**(支持『排序顺序“sorted order”』)和 **LinkedHashSet**(支持『插入顺序“sorted order”』)。这两个类会帮你照看所有细节。只有在遇到性能瓶颈的时候，你才应该用手动维护的数组来代替这两个类。

如果排序的时候用到了 **Comparator** (针对对象数组，primitive 数组不允许使用 **Comparator**)，那么 **binarySearch( )**的时候，也必须使用同一个 **Comparator** (用这个方法的重载版)。比方说，我们修改了 **AlphabeticSorting.java**：

```
//: c11:AlphabeticSearch.java
// Searching with a Comparator.
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSearch {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa, new
Arrays2.RandStringGenerator(5));
    AlphabeticComparator comp = new
AlphabeticComparator( );
    Arrays.sort(sa, comp);
    int index = Arrays.binarySearch(sa, sa[10],
comp);
    System.out.println("Index = " + index);
    monitor.expect(new String[] {
      "Index = 10"
    });
  }
} ///:~
```

必须把 **Comparator** 当作 **binaraySearch( )**的第三个参数传给它。在这个例子里，由于要找的东西是从数组里面挑的，因此肯定能找到。

数组部分的总结

总而言之，如果你要持有一组对象，首选，同时也是效率最高的选择，应该是数组。而且，如果这是一组 primitive 的话，你也只能用数组。接下来，我们要讲一些更为一般的情况，也就是写程序的时候还不知道要用多少对象，或者要用一种更复杂方式来存储对象情况。为此，Java 提供了“容器类(*container class*)”。其基本类型有 **List**, **Set** 和 **Map**。有了这些工具，你就能解决很多问题了。

它们还有一些别的特性。比方说 **Set** 所持有的对象，个个都不同，**Map** 则是一个“关联性数组(associative array)”，它能在两个对象之间建立联系。此外，与数组不同，它们还能自动调整大小，所以你可以往里面放任意数量的对象。这样写程序的时候，就不用操心要开多大的空间了。

容器简介

就我个人的感受，容器类能极大地增强我的编程能力，是软件开发领域最得力的工具之一。Java2 的重新设计[5]了 1.0 和 1.1 里面那个表现差劲的容器类。新的设计更紧凑也更合理。同时它也补齐了容器类库的功能，提供了链表(linked list)、队列(queue)和双向队列(deques，读成“decks”)这几种数据结构的功能。

[5]由 Sun 的 Joshua Bloch 负责设计。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

要设计容器类库是很难的(对绝大多数类库，设计总是很难的)。C++的容器类库里包括了很多各色各样的类。相比原先什么都没有，这种设计当然好出不少，但是它并不适合 Java。也有走另一个极端的。我就曾看到过一个只包括“container”类的容器类库。它的工作方式既象线性序列，又象关联性数组。Java 2 的容器类库则取了个折中：它实现了“成熟的容器类库所应实现”的一切功能，同时它又比 C++或其它类似的容器类库更易学易用。所以这个类库可能会有些怪。不像早期的 Java 类库，这些怪异之处并不是什么设计缺陷，相反它是在仔细斟酌了各种复杂因素之后才作的决定。或许你得花一点时间来熟悉这个类库，但是我相信，你很快就会学会容器类，然后把它们派上用场。

Java2 的容器类要解决“怎样持有对象”，而它把这个问题分成两类：

1. **Collection**: 通常是一组有一定规律的独立元素。**List** 必须按照特定的顺序持有这些元素，而 **Set** 则不能保存重复的元素。(bag 没有这个限制，但是 Java 的容器类库没有实现它，因为 **List** 已经提供这种功能了。)
2. **Map:** 一组以“键——值”(key-value)形式出现的 pair。初看上去，它应该是一个 pair 的 **Collection**，但是真这么去做的话，它就会变得很滑稽，所以还是把这个概念独立列出来为好。退一步说，真的要用到 **Map** 的某个子集的时候，创建一个 **Collection** 也是很方便的。**Map** 可以返回“键(key)的”**Set**，值的 **Collection**，或者 pair 的 **Set**。和数组一样，**Map** 不需要什么修改，就能很容易地扩展成多维。你只要直接把 **Map** 的值设成 **Map** 就可以了(然后它的值再是 **Map**，以此类推)。

我们先来看看容器的一般特性，然后深入细节，最后再看为什么会有这么多版本，以及如何进行选择。

打印容器

不像数组，容器不需要借助别的类就能清清楚楚地把自己给打印出来。下面这个例子还介绍了几种基本的容器：

```
//: c11:PrintingContainers.java
// Containers print themselves automatically.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class PrintingContainers {
  private static Test monitor = new Test( );
  static Collection fill(Collection c) {
    c.add("dog");
    c.add("dog");
    c.add("cat");
    return c;
  }
  static Map fill(Map m) {
    m.put("dog", "Bosco");
    m.put("dog", "Spot");
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    m.put("cat", "Rags");
    return m;
  }
  public static void main(String[] args) {
    System.out.println(fill(new ArrayList( )));
    System.out.println(fill(new HashSet( )));
    System.out.println(fill(new HashMap( )));
    monitor.expect(new String[] {
      "[dog, dog, cat]",
      "[dog, cat]",
      "{dog=Spot, cat=Rags}"
    });
  }
} ///:~
```

正如前面所讲的，Java 的容器类分成两种基本类型。它们的区别就在，每个位置能放多少对象。**Collection** 只允许每个位置上放一个对象(这个名字有点误导，因为容器类库也常被统称为 collections)。它包括“以一定顺序持有一组对象”的 **List**，以及“只能允许添加不重复的对象”的 **Set**。**ArrayList** 是一种 **List**，而 **HashSet** 则是一种 **Set**。你可以用 **add( )**方法往 **Collection** 里面加对象。

Map 保存的是“键(key)—值”形式的 pair，很像是一个微型数据库。上面这段程序用了一种叫 **HashMap** 的 **Map**。如果你建了一个“州和首府”的 **Map**，然后想查一下 Ohio 的首府在哪里，你就可以用它来找了。用法和用下标查数组是一样的。(**Map** 又被称为关联性数组 *associative array*。)你可以用 **put( )**方法往 **Map** 里面加元素。它接受键—值形式 pair 作参数。例程只演示了怎样把元素加进去，它没做查询。这部分的内容我们以后再讲。

fill( )方法还为 **Collection** 和 **Map** 作了重载。如果你看过输出，就会发现默认情况下(使用容器类的 **toString( )**方法)的打印效果已经很不错了，所以我们就不再提供额外的打印支持了。打印出来的 **Collection** 会用方括号括起来，元素与元素之间用逗号分开。**Map** 会用花括号括起来，键和值之间用等号联起来(键在左边，值在右边)。

你能一眼就看出各种容器的基本行为。**List** 会老老实实地持有你所输入的所有对象，既不做排序也不做编辑。**Set** 则每个对象只接受一次，而且还要用它自己的规则对元素进行重新排序(一般情况下，你关心的只是 **Set** 包没包括某个对象，而不是它到底排在哪里——如果是那样，你最好还是用 **List**)。而 **Map** 也不接收重复的 pair，至于是不是重复，要由 key 来决定。此外，它也有它自己的内部排序规则，不会受输入顺序影响。如果插入顺序是很重要的，那你就只能使用 **LinkedHashSet** 或 **LinkedHashMap** 了。

填充容器

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

虽然容器的打印问题是解决了，但它的填充却还跟 **java.util.Arrays** 一样，有着相同的不足。和 **Arrays** 一样，**Collection** 也有一个叫 **Collections** 的辅助类，它包含了一些静态的实用工具方法，其中就有一个 **fill( )**。这个 **fill( )**也只是把同一个对象的 **reference** 复制到整个容器，而且它还只能为 **List**，不能为 **Set** 和 **Map** 工作。

```
//: c11:FillingLists.java
// The Collections.fill( ) method.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class FillingLists {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    List list = new ArrayList( );
    for(int i = 0; i < 10; i++)
      list.add("");
    Collections.fill(list, "Hello");
    System.out.println(list);
    monitor.expect(new String[] {
      "[Hello, Hello, Hello, Hello, Hello, " +
       "Hello, Hello, Hello, Hello, Hello]"
    });
  }
} ///:~
```

实际上这个方法还不如 **Arrays**，因为它只能在替换，而不是在往 **List** 里面加新的元素。

为了能创建一些有实际意义的例子，我补充了一个带 **fill( )**方法的 **Collections2** 类库(为方便起见，我把它放进了 **com.bruceeckel.util**)。它能用 generator 加元素，而且还能让你指定要 **add( )**多少元素。前面定义的 **Generator** 能同 **Collections** 一同工作，但是 **Map** 需要一个它自己的 generator **interface**，因为每次调用它的 **next( )**的时候要生成两个对象。下面就是这个 **Pair** 类：

```
//: com:bruceeckel:util:Pair.java
package com.bruceeckel.util;

public class Pair {
  public Object key, value;
  public Pair(Object k, Object v) {
    key = k;
    value = v;
  }
} ///:~
```

接下来是生成这个 **Pair** 的 generator 的 **interface**：

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
//: com:bruceeckel:util:MapGenerator.java
package com.bruceeckel.util;
public interface MapGenerator { Pair next( ); }
///:~
```

做完准备之后，我们就能开始编写容器类的实用工具了：

```
//: com:bruceeckel:util:Collections2.java
// To fill any type of container using a generator
object.
package com.bruceeckel.util;
import java.util.*;

public class Collections2 {
  // Fill an array using a generator:
  public static void
  fill(Collection c, Generator gen, int count) {
    for(int i = 0; i < count; i++)
      c.add(gen.next( ));
  }
  public static void
  fill(Map m, MapGenerator gen, int count) {
    for(int i = 0; i < count; i++) {
      Pair p = gen.next( );
      m.put(p.key, p.value);
    }
  }
  public static class
  RandStringPairGenerator implements MapGenerator {
    private Arrays2.RandStringGenerator gen;
    public RandStringPairGenerator(int len) {
      gen = new Arrays2.RandStringGenerator(len);
    }
    public Pair next( ) {
      return new Pair(gen.next( ), gen.next( ));
    }
  }
  // Default object so you don't have to create your
own:
  public static RandStringPairGenerator rsp =
    new RandStringPairGenerator(10);
  public static class
  StringPairGenerator implements MapGenerator {
    private int index = -1;
    private String[][] d;
    public StringPairGenerator(String[][] data) {
      d = data;
    }
    public Pair next( ) {
      // Force the index to wrap:
      index = (index + 1) % d.length;
      return new Pair(d[index][0], d[index][1]);
    }
    public StringPairGenerator reset( ) {
      index = -1;
      return this;
    }
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  }
  // Use a predefined dataset:
  public static StringPairGenerator geography =
    new StringPairGenerator(CountryCapitals.pairs);
  // Produce a sequence from a 2D array:
  public static class StringGenerator implements
Generator{
    private String[][] d;
    private int position;
    private int index = -1;
    public StringGenerator(String[][] data, int pos)
{
      d = data;
      position = pos;
    }
    public Object next( ) {
      // Force the index to wrap:
      index = (index + 1) % d.length;
      return d[index][position];
    }
    public StringGenerator reset( ) {
      index = -1;
      return this;
    }
  }
  // Use a predefined dataset:
  public static StringGenerator countries =
    new StringGenerator(CountryCapitals.pairs, 0);
  public static StringGenerator capitals =
    new StringGenerator(CountryCapitals.pairs, 1);
} ///:~
```

这两个 **fill( )**都要根据参数来决定要往容器里面加多少元素。此外，**Map** 有两个 generator：**RandStringPairGenerator** 和 **StringPairGenerator**。前者会生成随机的字符串 pair，其长度要由构造函数的参数决定；而后者会根据一个两维的字符串数组，生成字符串 pair。**StringGenerator** 也接受一个两维的字符串数组，但是它生成的是单个的，而不是成对的元素。**static** 的 **rsp**，**geography**，**countries** 以及 **capitals** 对象提供了预设的 generator，其中后面三个会用到国家及其首都。注意，如果你要创建更多的对象 pair，generator 就会循环到数组的开头，但是如果你把这些对象 pair 放进 **Map**，那么这些重复的东西就会被忽略掉。

下面就是预定义的数据集。它包括了国家的名字及其首都：

```
//: com:bruceeckel:util:CountryCapitals.java
package com.bruceeckel.util;

public class CountryCapitals {
  public static final String[][] pairs = {
    // Africa
    {"ALGERIA","Algiers"}, {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"}, {"BOTSWANA","Gaberone"},
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        {"BURKINA FASO","Ouagadougou"},
        {"BURUNDI","Bujumbura"},
        {"CAMEROON","Yaounde"}, {"CAPE VERDE","Praia"},
        {"CENTRAL AFRICAN REPUBLIC","Bangui"},
        {"CHAD","N'djamena"},  {"COMOROS","Moroni"},
        {"CONGO","Brazzaville"},
    {"DJIBOUTI","Dijibouti"},
        {"EGYPT","Cairo"}, {"EQUATORIAL
    GUINEA","Malabo"},
        {"ERITREA","Asmara"}, {"ETHIOPIA","Addis Ababa"},
        {"GABON","Libreville"}, {"THE GAMBIA","Banjul"},
        {"GHANA","Accra"}, {"GUINEA","Conakry"},
        {"GUINEA","-"}, {"BISSAU","Bissau"},
        {"COTE D'IVOIR (IVORY COAST)","Yamoussoukro"},
        {"KENYA","Nairobi"}, {"LESOTHO","Maseru"},
        {"LIBERIA","Monrovia"}, {"LIBYA","Tripoli"},
        {"MADAGASCAR","Antananarivo"},
    {"MALAWI","Lilongwe"},
        {"MALI","Bamako"}, {"MAURITANIA","Nouakchott"},
        {"MAURITIUS","Port Louis"}, {"MOROCCO","Rabat"},
        {"MOZAMBIQUE","Maputo"}, {"NAMIBIA","Windhoek"},
        {"NIGER","Niamey"}, {"NIGERIA","Abuja"},
        {"RWANDA","Kigali"},
        {"SAO TOME E PRINCIPE","Sao Tome"},
        {"SENEGAL","Dakar"}, {"SEYCHELLES","Victoria"},
        {"SIERRA LEONE","Freetown"},
    {"SOMALIA","Mogadishu"},
        {"SOUTH AFRICA","Pretoria/Cape Town"},
        {"SUDAN","Khartoum"},
        {"SWAZILAND","Mbabane"}, {"TANZANIA","Dodoma"},
        {"TOGO","Lome"}, {"TUNISIA","Tunis"},
        {"UGANDA","Kampala"},
        {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",
         "Kinshasa"},
        {"ZAMBIA","Lusaka"}, {"ZIMBABWE","Harare"},
        // Asia
        {"AFGHANISTAN","Kabul"}, {"BAHRAIN","Manama"},
        {"BANGLADESH","Dhaka"}, {"BHUTAN","Thimphu"},
        {"BRUNEI","Bandar Seri Begawan"},
        {"CAMBODIA","Phnom Penh"},
        {"CHINA","Beijing"}, {"CYPRUS","Nicosia"},
        {"INDIA","New Delhi"}, {"INDONESIA","Jakarta"},
        {"IRAN","Tehran"}, {"IRAQ","Baghdad"},
        {"ISRAEL","Tel Aviv"}, {"JAPAN","Tokyo"},
        {"JORDAN","Amman"}, {"KUWAIT","Kuwait City"},
        {"LAOS","Vientiane"}, {"LEBANON","Beirut"},
        {"MALAYSIA","Kuala Lumpur"}, {"THE
    MALDIVES","Male"},
        {"MONGOLIA","Ulan Bator"},
        {"MYANMAR (BURMA)","Rangoon"},
        {"NEPAL","Katmandu"}, {"NORTH
    KOREA","P'yongyang"},
        {"OMAN","Muscat"}, {"PAKISTAN","Islamabad"},
        {"PHILIPPINES","Manila"}, {"QATAR","Doha"},
        {"SAUDI ARABIA","Riyadh"},
    {"SINGAPORE","Singapore"},
        {"SOUTH KOREA","Seoul"}, {"SRI LANKA","Colombo"},
        {"SYRIA","Damascus"},
        {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
        {"THAILAND","Bangkok"}, {"TURKEY","Ankara"},
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    {"UNITED ARAB EMIRATES","Abu Dhabi"},
    {"VIETNAM","Hanoi"}, {"YEMEN","Sana'a"},
    // Australia and Oceania
    {"AUSTRALIA","Canberra"}, {"FIJI","Suva"},
    {"KIRIBATI","Bairiki"},
    {"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
    {"MICRONESIA","Palikir"}, {"NAURU","Yaren"},
    {"NEW ZEALAND","Wellington"}, {"PALAU","Koror"},
    {"PAPUA NEW GUINEA","Port Moresby"},
    {"SOLOMON ISLANDS","Honaira"},
{"TONGA","Nuku'alofa"},
    {"TUVALU","Fongafale"}, {"VANUATU","< Port-
Vila"},
    {"WESTERN SAMOA","Apia"},
    // Eastern Europe and former USSR
    {"ARMENIA","Yerevan"}, {"AZERBAIJAN","Baku"},
    {"BELARUS (BYELORUSSIA)","Minsk"},
    {"GEORGIA","Tbilisi"},
    {"KAZAKSTAN","Almaty"}, {"KYRGYZSTAN","Alma-
Ata"},
    {"MOLDOVA","Chisinau"}, {"RUSSIA","Moscow"},
    {"TAJIKISTAN","Dushanbe"},
{"TURKMENISTAN","Ashkabad"},
    {"UKRAINE","Kyiv"}, {"UZBEKISTAN","Tashkent"},
    // Europe
    {"ALBANIA","Tirana"}, {"ANDORRA","Andorra la
Vella"},
    {"AUSTRIA","Vienna"}, {"BELGIUM","Brussels"},
    {"BOSNIA","-"}, {"HERZEGOVINA","Sarajevo"},
    {"CROATIA","Zagreb"}, {"CZECH
REPUBLIC","Prague"},
    {"DENMARK","Copenhagen"}, {"ESTONIA","Tallinn"},
    {"FINLAND","Helsinki"}, {"FRANCE","Paris"},
    {"GERMANY","Berlin"}, {"GREECE","Athens"},
    {"HUNGARY","Budapest"}, {"ICELAND","Reykjavik"},
    {"IRELAND","Dublin"}, {"ITALY","Rome"},
    {"LATVIA","Riga"}, {"LIECHTENSTEIN","Vaduz"},
    {"LITHUANIA","Vilnius"},
{"LUXEMBOURG","Luxembourg"},
    {"MACEDONIA","Skopje"}, {"MALTA","Valletta"},
    {"MONACO","Monaco"}, {"MONTENEGRO","Podgorica"},
    {"THE NETHERLANDS","Amsterdam"},
{"NORWAY","Oslo"},
    {"POLAND","Warsaw"}, {"PORTUGAL","Lisbon"},
    {"ROMANIA","Bucharest"}, {"SAN MARINO","San
Marino"},
    {"SERBIA","Belgrade"}, {"SLOVAKIA","Bratislava"},
    {"SLOVENIA","Ljujiana"}, {"SPAIN","Madrid"},
    {"SWEDEN","Stockholm"}, {"SWITZERLAND","Berne"},
    {"UNITED KINGDOM","London"}, {"VATICAN CITY","--
-"},
    // North and Central America
    {"ANTIGUA AND BARBUDA","Saint John's"},
    {"BAHAMAS","Nassau"},
    {"BARBADOS","Bridgetown"}, {"BELIZE","Belmopan"},
    {"CANADA","Ottawa"}, {"COSTA RICA","San Jose"},
    {"CUBA","Havana"}, {"DOMINICA","Roseau"},
    {"DOMINICAN REPUBLIC","Santo Domingo"},
    {"EL SALVADOR","San Salvador"},
    {"GRENADA","Saint George's"},
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    {"GUATEMALA","Guatemala City"},
    {"HAITI","Port-au-Prince"},
    {"HONDURAS","Tegucigalpa"},
{"JAMAICA","Kingston"},
    {"MEXICO","Mexico City"},
{"NICARAGUA","Managua"},
    {"PANAMA","Panama City"}, {"ST. KITTS","-"},
    {"NEVIS","Basseterre"}, {"ST. LUCIA","Castries"},
    {"ST. VINCENT AND THE GRENADINES","Kingstown"},
    {"UNITED STATES OF AMERICA","Washington, D.C."},
    // South America
    {"ARGENTINA","Buenos Aires"},
    {"BOLIVIA","Sucre (legal)/La
Paz(administrative)"},
    {"BRAZIL","Brasilia"}, {"CHILE","Santiago"},
    {"COLOMBIA","Bogota"}, {"ECUADOR","Quito"},
    {"GUYANA","Georgetown"}, {"PARAGUAY","Asuncion"},
    {"PERU","Lima"}, {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"},
{"VENEZUELA","Caracas"},
  };
} ///:~
```

这只是一个两维的字符串数组。[6]下面是 **fill( )**方法和 generator 的测试：

```
//: c11:FillTest.java
import com.bruceeckel.util.*;
import java.util.*;

public class FillTest {
  private static Generator sg =
    new Arrays2.RandStringGenerator(7);
  public static void main(String[] args) {
    List list = new ArrayList( );
    Collections2.fill(list, sg, 25);
    System.out.println(list + "\n");
    List list2 = new ArrayList( );
    Collections2.fill(list2, Collections2.capitals,
25);
    System.out.println(list2 + "\n");
    Set set = new HashSet( );
    Collections2.fill(set, sg, 25);
    System.out.println(set + "\n");
    Map m = new HashMap( );
    Collections2.fill(m, Collections2.rsp, 25);
    System.out.println(m + "\n");
    Map m2 = new HashMap( );
    Collections2.fill(m2, Collections2.geography,
25);
    System.out.println(m2);
  }
} ///:~
```

[6]这些数据是在 Internet 上找的，我用 Python 程序作了一下处理 (见 *www.Python.org*)。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

有了这些工具，你就能用有实际意义的数据来测试容器了。

容器的缺点：不知道对象的类型

Java 的容器有个缺点，就是往容器里面放对象的时候，会把对象的类型信息给弄丢了。这是因为开发容器类的程序员不会知道你要用它来保存什么类型的对象，而让容器仅只保存特定类型的对象又会影响它的通用性。所以容器被做成只持有 **Object**，也就是所有对象的根类的 reference，这样它就能持有任何类型的对象了。(当然不包括 primitive，因为它们不是对象，也没有继承别的对象。)这是一个很了不起的方案，只是：

1. 由于在将对象放入容器的时候，它的类型信息被扔掉了，所以容器对“能往里面加什么类型的对象”没有限制。比方说，即使你想让它只持有 cat，别人也能很轻易地把 dog 放进去。
2. 由于对象的类型信息没了，容器只知道它持有的 **Object** 的 reference，所以对象在使用之前还必须进行类型转换。

好的一面是， Java 不会让你误用放进容器里的对象。假设你往 cat 的容器里面扔了个 dog，然后要把这个容器里的所有对象都当 cat 来用，当你把 dog 的 reference 从 cat 的容器里面拉出来，并且试图将它转换成 cat 的时候，就会引发一个 **RuntimeException**。

接下来我们就用大家最常用 **ArrayList** 容器类来举一个例子。对初学者来说，你可以把 **ArrayList** 想成“一个能够自动扩展的数组”。**ArrayList** 的用法也是很简单：先创建一个，用 **add( )**把对象放进去，要用的时候再给 **get( )**传一个下标——就跟用数组差不多，只是不需要用方括号了。[7]**ArrayList** 也有一个 **size( )**方法，它会告诉你容器里面有多少对象，这样你就不会粗心大意地过了界然后引发异常了。

先创建 **Cat** 和 **Dog**：

```
//: c11:Cat.java
package c11;

public class Cat {
  private int catNumber;
  public Cat(int i) { catNumber = i; }
  public void id( ) {
    System.out.println("Cat #" + catNumber);
  }
} ///:~
```

[7]这种地方能重载运算符就好了。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
//: c11:Dog.java
package c11;

public class Dog {
  private int dogNumber;
  public Dog(int i) { dogNumber = i; }
  public void id( ) {
    System.out.println("Dog #" + dogNumber);
  }
} ///:~
```

先把 **Cat** 和 **Dog** 放进容器，然后再把它们捞出来：

```
//: c11:CatsAndDogs.java
// Simple container example.
// {ThrowsException}
package c11;
import java.util.*;

public class CatsAndDogs {
  public static void main(String[] args) {
    List cats = new ArrayList( );
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Not a problem to add a dog to cats:
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size( ); i++)
      ((Cat)cats.get(i)).id( );
      // Dog is detected only at run time
  }
} ///:~
```

Cat 和 **Dog** 是两个不同的类；除了都是对象之外，它们没有什么相同之处。(如果你不明确地说明这个类是继承谁的，那么它就自动继承 **Object**。)由于 **ArrayList** 持有的是 **Object**，所以你不但能用 **ArrayList** 的 **add( )**来加 **Cat**，也可以用它来加 **Dog**。这么做在编译和运行的时候都不会报错。但是，当你用 **get( )**方法把你认为是 **Cat** 的对象取出来的时候，你得到的是一个要转换成 **Cat** 的 **Object** 的 **reference**。于是，在你能调用 **Cat** 的 **id( )**方法之前，你还要用括号来做强制的类型转换；不然的话就是一个语法错误。这样，到程序运行的时候，当你把 **Dog** 转换成 **Cat** 的时候，就会得到一个异常了。

这还不只是麻烦。它还会制造一些很难发现的 bug。假如程序在什么地方(或好几个地方)往容器里面插了对象，然后你发现，只要程序执行到了这个地方，就会有一个异常跑出来，告诉你容器里面有一个错误的对象，于是你就得把这个坏的插入点找出来。在绝大多数情况下，这不是什么问题，但是你还是应该对这种可能性保持警惕。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

有时即使不正确它也能运行

有时，即便不把对象转换成原先的类型，它好像也能正常工作。有一种情况比较特殊：**String** 能从编译器那里得到一些能使之平稳工作的特殊帮助。只要编译器没能得到它所期望的 **String** 对象，它就会调用 **toString( )**。这个方法由 **Object** 定义，能被任何 **Java** 类覆写。它所返回的 **String** 对象，会被用到任何要用它的地方。

于是只要覆写了类的 **toString( )**方法，你就能打印对象了，就像下面这样：

```
//: c11:Mouse.java
// Overriding toString( ).

public class Mouse {
  private int mouseNumber;
  public Mouse(int i) { mouseNumber = i; }
  // Override Object.toString( ):
  public String toString( ) {
    return "This is Mouse #" + mouseNumber;
  }
  public int getNumber( ) { return mouseNumber; }
} ///:~



//: c11:MouseTrap.java

public class MouseTrap {
  static void caughtYa(Object m) {
    Mouse mouse = (Mouse)m; // Cast from Object
    System.out.println("Mouse: " +
mouse.getNumber( ));
  }
} ///:~



//: c11:WorksAnyway.java
// In special cases, things just seem to work
correctly.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class WorksAnyway {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    List mice = new ArrayList( );
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size( ); i++) {
      // No cast necessary, automatic
      // call to Object.toString( ):
      System.out.println("Free mouse: " +
mice.get(i));
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      MouseTrap.caughtYa(mice.get(i));
    }
    monitor.expect(new String[] {
      "Free mouse: This is Mouse #0",
      "Mouse: 0",
      "Free mouse: This is Mouse #1",
      "Mouse: 1",
      "Free mouse: This is Mouse #2",
      "Mouse: 2"
    });
  }
} ///:~
```

你会发现 **Mouse** 的 **toString( )**被覆写了。**main( )**的第二个 **for** 循环里有下面这句：

```
System.out.println("Free mouse: " + mice.get(i));
```

编译器预计‘**+**’后面跟着的是一个 **String** 对象。而 **get( )**返回了一个 **Object**，所以为了得到它所期望的 **String**，编译器会偷偷地调用 **toString( )**。遗憾的是，你只能对 **String** 玩这套把戏；其它类不行。

第二种解决方案是把类型转换藏起来，把它放到 **MouseTrap** 里面去。**caughtYa( )**方法接受的不是 **Mouse**，而是一个要转换成 **Mouse** 的 **Object**。当然，这种做法是相当专横的，由于它拿的是 **Object**，因此什么东西都能传给它了。但是如果类型转换失败了，假设你传了一个错误的类型，程序运行的时候就会抛出异常。这当然不及编译时的检查，但也还算健壮。注意，如果你用这种方法：

```
MouseTrap.caughtYa(mice.get(i));
```

类型转换就不再需要了。

做一个类型敏感的 ArrayList

可能你不会就这样放弃这个问题。还有一个更无坚不摧的解决方案，就是用 **ArrayList** 来建一个新的类，让它只能处理你规定的类型：

```
//: c11:MouseList.java
// A type-conscious List.
import java.util.*;

public class MouseList {
  private List list = new ArrayList( );
  public void add(Mouse m) { list.add(m); }
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  public Mouse get(int index) {
    return (Mouse)list.get(index);
  }
  public int size( ) { return list.size( ); }
} ///:~
```

下面就是对这个新容器做的测试：

```
//: c11:MouseListTest.java
import com.bruceeckel.simpletest.*;

public class MouseListTest {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    MouseList mice = new MouseList( );
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size( ); i++)
      MouseTrap.caughtYa(mice.get(i));
    monitor.expect(new String[] {
      "Mouse: 0",
      "Mouse: 1",
      "Mouse: 2"
    });
  }
} ///:~
```

除了有一个 **private** 的 **ArrayList**，以及两个很像 **ArrayList** 的方法之外，新的 **MouseList** 同前面那个例子非常像。但是它不再接收和返回泛型的 **Object** 对象了，它只处理 **Mouse** 对象。

注意，如果 **MouseList** 是继承 **ArrayList** 的，那么 **add(Mouse)**就变成了对 **add(Object)**的重载了，于是“容器能放什么对象”还是没有限制，而你也不会得到你所希望的效果。而用了合成之后，**MouseList** 只是单纯地利用了 **ArrayList**。它会先做一些工作，然后再把下一步的任务交给 **ArrayList**。

因为 **MouseList** 只接受 **Mouse**，所以如果你说：

```
mice.add(new Pigeon( ));
```

编译的时候就会报错。虽然仅从代码的角度来讲，这种方案比较冗长，但是如果你用错了类型，它会立刻告诉你。

记住，**get( )**的时候不需要做类型转换；它永远都是 **Mouse**。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

参数化类型 （Parameterized types）

这并不是一个孤立的问题。很多时候，你都会碰到要利用其它类型创建新的类型，而编译的时候这种类型的信息又是极为有用的情况。这就是所谓的“参数化类型(*parameterized types*)”的概念了。在 C++中，它是用 *templates* 直接由语言提供支持的。Java 很可能会在 JDK 1.5 里面提供它自己的参数化类型，*generics*。

迭代器

无论是哪种容器，你都得有办法既能放东西进去，也能拿东西出来。毕竟，容器的主要任务就是存放对象。**ArrayList** 的 **add( )**就是用来放东西的，而 **get( )**则是把对象拿出来的办法。**ArrayList** 很灵活；你可以随时提取任何东西，并且换一个下标，马上就能选择另一个元素。

但是，如果你深入下去，就会发现这么做会有一个缺点：要这样用的话，你必须预先知道容器的确切类型。可能刚开始的时候，你会觉得这并不是什么问题，但是假设，你用 **ArrayList** 做了个设计，然后发觉，在这种情况下，选 **Set** 才对，那你该怎么办呢？或者，你想写一段泛型程序，为了让它能“不经重写就能同各种容器一同工作”，它应该既不知道也不关心它要处理的是哪种容器，那么你又该怎么办呢？

“迭代器(*iterator*)”的概念(又是一个设计模式)就是用来达成这种抽象的。迭代器是一个对象，它的任务是，能在让“客户程序员在不知道，或者不关心他所处理的是什么样的底层序列结构”的情况下，就能在一个对象序列中前后移动，并选取其中的对象。此外迭代器还是一种通常所说的“轻量级”的对象，即创建代价很小的对象。所以，你常会发现迭代器有一些看上去很奇怪的限制：比如有些迭代器只能按一个方向移动。

Java 的 **Iterator** 就属于有这种限制的迭代器。它做不了很多事情，除了：

1. 用 **iterator( )**方法叫容器传给你一个 **Iterator** 对象。第一次调用 **Iterator** 的 **next( )**方法的时候，它就会传给你序列中的第一个元素。
2. 用 **next( )**方法获取序列中的下一个对象。
3. 用 **hasNext( )**方法查询序列中是否还有其他对象。
4. 用 **remove( )**方法删除迭代器所返回的最后一个元素。

就这么多了。这只是迭代器的一个很简单的实现，不过还是很强大(对 **List** 来说，还有一个更精巧的 **ListIterator**)。为了能看到它是怎样工作的，我们要重写一遍本章前面所讲的 **CatsAndDogs.java**。原来我们是用 **get( )**来选取元素的，现在我们要用 **Iterator**：

```
//: c11:CatsAndDogs2.java
// Simple container with Iterator.
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
package c11;
import com.bruceeckel.simpletest.*;
import java.util.*;

public class CatsAndDogs2 {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    List cats = new ArrayList( );
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    Iterator e = cats.iterator( );
    while(e.hasNext( ))
      ((Cat)e.next( )).id( );
  }
} ///:~
```

可以看到，最后这几行是用 **Iterator**，而不是 **for** 循环来遍历整个序列的。有了 **Iterator**，你就不用操心容器里还有几个元素。**hasNext( )** 和 **next( )**已经帮你打理好了。

再举一个例程，想想怎样写一个通用的打印程序：

```
//: c11:Printer.java
// Using an Iterator.
import java.util.*;

public class Printer {
  static void printAll(Iterator e) {
    while(e.hasNext( ))
      System.out.println(e.next( ));
  }
} ///:~
```

仔细看 **printAll( )**。注意一下，它里面没有要打印哪种序列的信息。它要的只是一个 **Iterator** 的对象，对一个序列来说这已经够了：你可以用它来获取容器中的下一个对象，它也会告诉你是不是到界了。这种“一下子就拿走容器里的所有对象，然后再一个一个地进行处理”的思路是非常强大的，也是贯穿本书的。

实际上这段程序还要泛化，因为它还隐含地调用了 **Object.toString( )** 方法。**println( )**对所有的 primitve 和 **Object** 都做了重载；在各种情况下，它都会自动调用合适的 **toString( )**方法来生成它所需的 **String** 对象。

尽管不是必须的，但你还是可以明确地使用类型传递。结果就是调用 **toString( )**:

```
System.out.println((String)e.next( ));
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

总之，除了是调用 **Object** 的方法，一般情况下，你总还要做一些其他事情，否则又会撞上类型传递的问题了。你要把它当作你感兴趣的那种对象的序列的 **Iterator**，然后把它转换成目标类型(如果出了错，又会得到一个运行时异常)。

我们可以用它来打印 **Hamsters**，并以此作一个测试：

```
//: c11:Hamster.java

public class Hamster {
  private int hamsterNumber;
  public Hamster(int hamsterNumber) {
    this.hamsterNumber = hamsterNumber;
  }
  public String toString( ) {
    return "This is Hamster #" + hamsterNumber;
  }
} ///:~
```

```
//: c11:HamsterMaze.java
// Using an Iterator.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class HamsterMaze {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    List list = new ArrayList( );
    for(int i = 0; i < 3; i++)
      list.add(new Hamster(i));
    Printer.printAll(list.iterator( ));
    monitor.expect(new String[] {
      "This is Hamster #0",
      "This is Hamster #1",
      "This is Hamster #2"
    });
  }
} ///:~
```

你可以写一个拿 **Collection** 对象当参数的 **printAll( )**，但是 **Iterator** 能提供更好的“脱钩机制(decoupling)”。

不经意的递归　(Unintended recursion)

由于 Java 的标准容器类(同其它类一样)也是继承 **Object** 的，因此它们也有一个 **toString( )**方法。这个方法已经被覆写了，所以它能生成一个表示它自己以及所有它所保存的对象的 **String**。比如 **ArrayList** 的

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

toString( )方法就会遍历 **ArrayList** 的每个元素，然后调用它们的 **toString( )**方法。假设你要打印类的地址。好像最直接的办法就是使用 **this**(C++的程序员尤其喜欢用这种办法)：

```
//: c11:InfiniteRecursion.java
// Accidental recursion.
// {RunByHand}
import java.util.*;

public class InfiniteRecursion {
  public String toString( ) {
    return " InfiniteRecursion address: " + this +
"\n";
  }
  public static void main(String[] args) {
    List v = new ArrayList( );
    for(int i = 0; i < 10; i++)
      v.add(new InfiniteRecursion( ));
    System.out.println(v);
  }
} ///:~
```

如果你直接创建一个 **InfiniteRecursion**，然后把它打印出来，你就会得到一串无穷无尽的异常。把它放到 **ArrayList** 也一样。这是 **toString( )**在作怪。当你写：

```
"InfiniteRecursion address: " + this
```

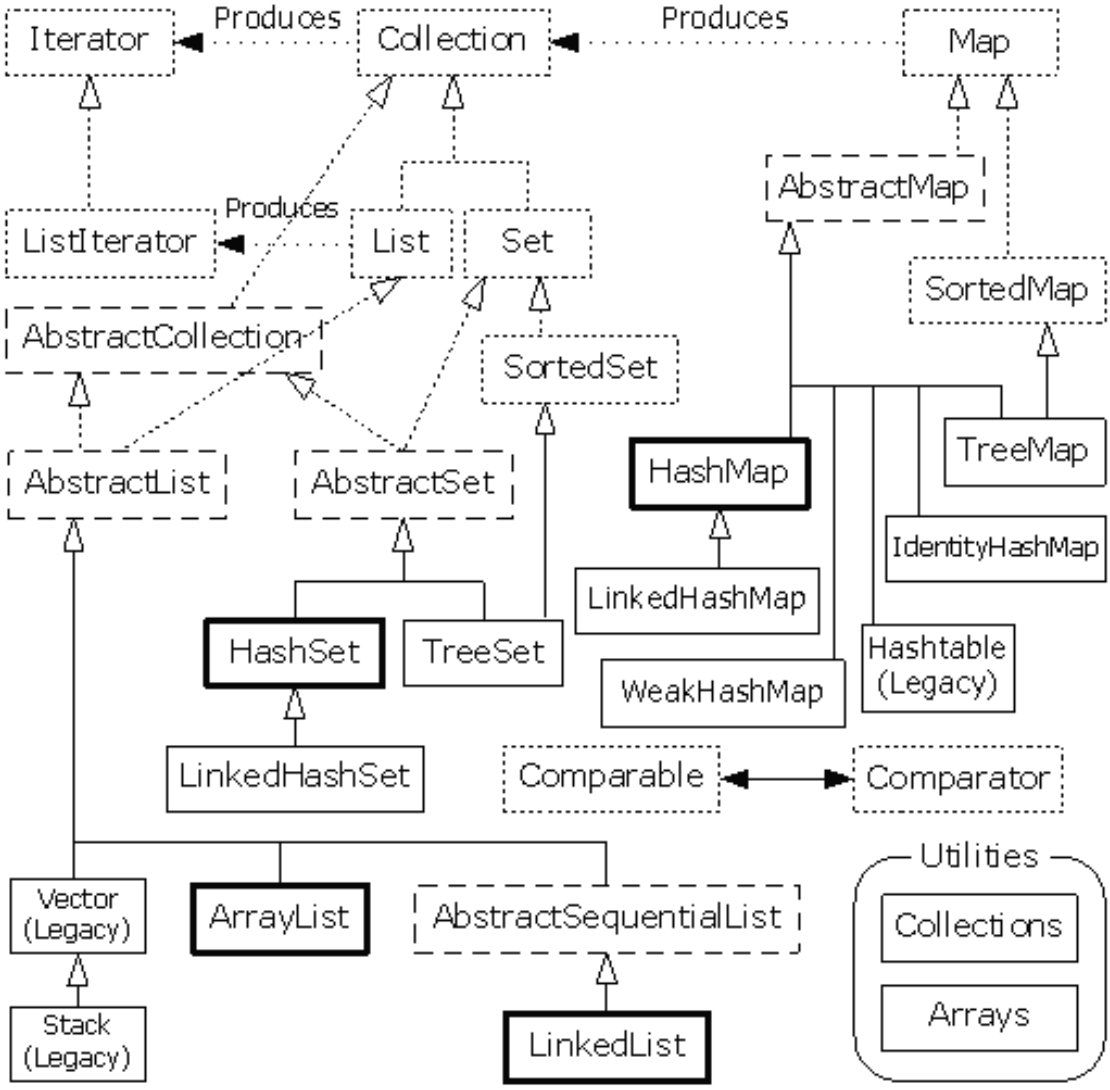
的时候，编译器看到 **String** 和‘+’后面跟着的不是 **String**。于是它试着将 **this** 转换成 **String**。转换要用到 **toString( )**，于是就变成递归调用了。

如果你真的想打印对象的地址，解决办法就是去调用 **Object** 的 **toString( )**方法，它就是干这活的。所以不要用 **this**，应该写 **super.toString( )**。

容器分类学

根据编程的需要，**Collection** 和 **Map** 分别有好几个实现。所以看懂 Java 容器类(JDK 1.4)的关系图是很有用的。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com



第一眼看到这张图的时候，你会觉得很震撼。不过你马上就会知道，实际上只有三种容器组件——**Map**，**List** 和 **Set**，而每种又有两到三个实现。最常用的几个容器已经用粗黑线框了起来。看到这里，这张图就不再那么令人望而生畏了。

用点号框起来的是 **interface**，用虚线框起来的是 **abstract** 类，实线则表示普通的(“实体 concrete”)类。点线的箭头表示类实现了这个 **interface**(或者，**abstract** 类表示部分实现了这个 **interface**)。实线箭头表示这个类可以制造箭头所指的那个类的对象。比如，**Collection** 能制造 **Iterator**，而 **List** 还能制造 **ListIterator**(也能制造 **Iterator**，因为 **List** 是继承自 **Collection** 的)。

与存放对象有关的接口包括 **Collection**，**List**，**Set** 和 **Map**。在理想情况下，绝大多数代码应该只同这些接口打交道，只是在创建容器的时候才要精确地指明它的确切类型。所以你可以这样创建一个 **List**。

```
List x = new LinkedList( );
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

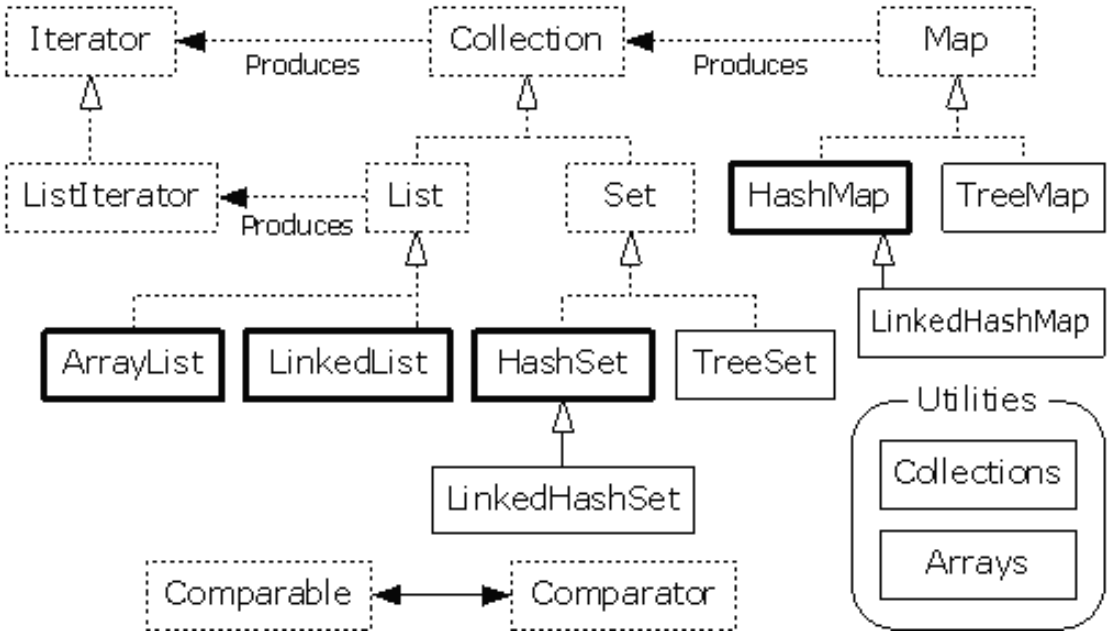
当然，你也可以选择让 **x** 成为 **LinkedList**(而不是泛型的 **List**)，这样 **x** 就带上了准确的类型信息。**interface** 的优雅 (同时也是它的本意)就在于，你想修改具体的实现的时候，只要改一下创建的声明就可以了，就像这样：

```
List x = new ArrayList( );
```

无需惊动其它代码(用迭代器也能获得一些这种泛型性)。

这个类系里面有很多以 **“Abstract”** 开头的类，初看起来这可能会让人有点不明白。实际上它们只是一些部分实现某个接口的办成品。假如你要编一个你自己的 **Set**，不要从 **Set** 接口开始挨个实现它的方法；相反你最好继承 **AbstractSet**，这样就能把编程的工作量压缩到最低了。但是，实际上容器类库的功能已经够强的了，我们要求的事情它几乎都能做到。所以对我们来说，你完全可以忽略以 **“Abstract”** 开头的类。

因此看这张图的时候，真正需要关心只是顶层的 **interface** 和那些实体类(用实线框起来的那些)。通常你会创建实体类的对象，然后把它上传到相应的 **interface**，再在程序里面使用这个 **interface**。此外，写新代码的时候不用考虑用已经淘汰的东西。所以这张图能被简化为如下所示：



现在它只包括那些你平常会用到的接口和类了。这些东西也是本章要着重探讨的。注意，这张图里没有包括 **WeakHashMap** 和 JDK 1.4 的 **IdentityHashMap**。这是因为它们是有特殊用途的，你不太可能会用到它们。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

下面就是一个用 **String** 对象来填充 (用 **ArrayList** 表示的) **Collection**，并且打印其所有元素的例子：

```
//: c11:SimpleCollection.java
// A simple example using Java 2 Collections.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class SimpleCollection {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    // Upcast because we just want to
    // work with Collection features
    Collection c = new ArrayList( );
    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator( );
    while(it.hasNext( ))
      System.out.println(it.next( ));
    monitor.expect(new String[] {
      "0",
      "1",
      "2",
      "3",
      "4",
      "5",
      "6",
      "7",
      "8",
      "9"
    });
  }
} ///:~
```

main( )的第一行创建了一个 **ArrayList** 对象，然后把它上传给了 **Collection**。由于程序只用到了 **Collection** 的方法，因此只要是继承 **Collection** 的都能正常工作，不过 **ArrayList** 是我们最常用的 **Collection**。

add( )，就像它的名字告诉我们的，会把新的元素放进 **Collection**。但是文档里面特别仔细地声明，“**add( )**会确保容器包含指定的元素”。这句话是说给 **Set** 的，因为它只添加原先没有的元素。对 **ArrayList** 或其它 **List**，**add( )**总是“把它放进去”，因为 **List** 并不关心它是不是保存了相同的元素。

Collection 都能用 **iterator( )**方法产生一个 **Iterator**。这里，我们用 **Iterator** 来遍历整个 **Collection**，然后把它们打印出来。

Collection 的功能

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

下面这张表给出了 **Collection** 的所有功能，也就是你能用 **Set** 和 **List** 做什么事(不包括从 **Object** 自动继承过来的方法)。(**List** 还有一些额外的功能。)**Map** 不是继承 **Collection** 的，所以我们会区别对待。

boolean add(Object)	确保容器能持有你传给它的那个参数。如果没能把它加进去，就返回 **false**。(这是个“可选”的方法，本章稍后会再作解释。)
boolean addAll(Collection)	加入参数 **Collection** 所含的所有元素。只要加了元素，就返回 **true**。(“可选”)
void clear( )	清除容器所保存的所有元素。(“可选”)
boolean contains(Object)	如果容器持有参数 **Object**，就返回 **true**。
boolean containsAll(Collection)	如果容器持有参数 **Collection** 所含的全部元素，就返回 **true**。
boolean isEmpty( )	如果容器里面没有保存任何元素，就返回 **true**。
Iterator iterator( )	返回一个可以在容器的各元素之间移动的 **Iterator**。
boolean remove(Object)	如果容器里面有这个参数 **Object**，那么就把其中的某一个给删了。只要删除过东西，就返回 **true**。(“可选”)
boolean removeAll(Collection)	删除容器里面所有参数 **Collection** 所包含的元素。只要删过东西，就返回 **true**。(“可选”)
boolean retainAll(Collection)	只保存参数 **Collection** 所包括的元素(集合论中“交集”的概念)。如果发生过变化，则返回 **true**。(“可选”)
int size( )	返回容器所含元素的数量。
Object[] toArray( )	返回一个包含容器中所有元素的数组。
Object[] toArray(Object[] a)	返回一个包含容器中所有元素的数组，且这个数组不是普通的 **Object** 数组，它的类型应该同参数数组 **a** 的类型相同(要做类型转换)。

注意，这里没有能进行随机访问的 **get( )**方法。这是因为 **Collection** 还包括 **Set**。而 **Set** 有它自己的内部顺序(因此随机访问是毫无意义的)。所以如果你要检查 **Collection** 的元素，你就必须使用迭代器。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

下面这段程序把这些方法都演示了一遍。再讲一下，实现了 **Collection** 接口的容器都有这些功能，只是 **ArrayList** 是用着最顺手的：

```
//: c11:Collection1.java
// Things you can do with all Collections.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class Collection1 {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    Collection c = new ArrayList( );
    Collections2.fill(c, Collections2.countries, 5);
    c.add("ten");
    c.add("eleven");
    System.out.println(c);
    // Make an array from the List:
    Object[] array = c.toArray( );
    // Make a String array from the List:
    String[] str = (String[])c.toArray(new
String[1]);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println("Collections.max(c) = " +
      Collections.max(c));
    System.out.println("Collections.min(c) = " +
      Collections.min(c));
    // Add a Collection to another Collection
    Collection c2 = new ArrayList( );
    Collections2.fill(c2, Collections2.countries, 5);
    c.addAll(c2);
    System.out.println(c);
    c.remove(CountryCapitals.pairs[0][0]);
    System.out.println(c);
    c.remove(CountryCapitals.pairs[1][0]);
    System.out.println(c);
    // Remove all components that are
    //  in the argument collection:
    c.removeAll(c2);
    System.out.println(c);
    c.addAll(c2);
    System.out.println(c);
    // Is an element in this Collection?
    String val = CountryCapitals.pairs[3][0];
    System.out.println("c.contains(" + val  + ") = "
      + c.contains(val));
    // Is a Collection in this Collection?
    System.out.println(
      "c.containsAll(c2) = " + c.containsAll(c2));
    Collection c3 = ((List)c).subList(3, 5);
    // Keep all the elements that are in both
    // c2 and c3 (an intersection of sets):
    c2.retainAll(c3);
    System.out.println(c);
    // Throw away all the elements
    // in c2 that also appear in c3:
    c2.removeAll(c3);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    System.out.println("c.isEmpty( ) = " +
c.isEmpty( ));
    c = new ArrayList( );
    Collections2.fill(c, Collections2.countries, 5);
    System.out.println(c);
    c.clear( ); // Remove all elements
    System.out.println("after c.clear( ):");
    System.out.println(c);
    monitor.expect(new String[] {
      "[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA
FASO, " +
        "ten, eleven]",
      "Collections.max(c) = ten",
      "Collections.min(c) = ALGERIA",
      "[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA
FASO, " +
      "ten, eleven, BURUNDI, CAMEROON, CAPE VERDE, "
+
      "CENTRAL AFRICAN REPUBLIC, CHAD]",
      "[ANGOLA, BENIN, BOTSWANA, BURKINA FASO, ten,
" +
      "eleven, BURUNDI, CAMEROON, CAPE VERDE, " +
      "CENTRAL AFRICAN REPUBLIC, CHAD]",
      "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven,
" +
      "BURUNDI, CAMEROON, CAPE VERDE, " +
      "CENTRAL AFRICAN REPUBLIC, CHAD]",
      "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven]",
      "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven,
" +
      "BURUNDI, CAMEROON, CAPE VERDE, " +
      "CENTRAL AFRICAN REPUBLIC, CHAD]",
      "c.contains(BOTSWANA) = true",
      "c.containsAll(c2) = true",
      "[BENIN, BOTSWANA, BURKINA FASO, ten, eleven,
" +
      "BURUNDI, CAMEROON, CAPE VERDE, " +
      "CENTRAL AFRICAN REPUBLIC, CHAD]",
      "c.isEmpty( ) = false",
      "[COMOROS, CONGO, DJIBOUTI, EGYPT, " +
      "EQUATORIAL GUINEA]",
      "after c.clear( ):",
      "[]"
    });
  }
} ///:~
```

我们把数据放在 **ArrayList** 里面，然后把它传给 **Collection**，所以很清楚，除了 **Collection** 接口，我们什么都没用。**main( )**只是简单地演示了一遍 **Collection** 的方法。

接下来我们要讲 **List**，**Set** 和 **Map** 的各种实现了，每讲一种容器，我都会(用星号)告诉你默认情况下应该选用哪种实现。你会发现，我们**没**把老式的 **Vector**，**Stack** 和 **Hashtable** 包括进来。这是因为，无论是哪种容器，**Java 2** 的类库里都有更好的选择。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

List 的功能

正如你从 **ArrayList** 那里所看到的，**List** 的基本用法是相当简单的。虽然绝大多数时候，你只是用 **add( )**加对象，用 **get( )**取对象，用 **iterator( )**获取这个序列的 **Iterator**，但 **List** 还有一些别的很有用的方法。

实际上有两种 **List**：擅长对元素进行随机访问的，较常用的 **ArrayList**，和更强大的 **LinkedList**。**LinkedList** 不是为快速的随机访问而设计的，但是它却有一组更加通用的方法。

List (接口)	**List** 的最重要的特征就是有序；它会确保以一定的顺序保存元素。**List** 在 **Collection** 的基础上添加了大量方法，使之能在序列中间插入和删除元素。(只对 **LinkedList** 推荐使用。)**List** 可以制造 **ListIterator** 对象，你除了能用它在 **List** 的中间插入和删除元素之外，还能用它沿两个方向遍历 **List**。
ArrayList*	一个用数组实现的 **List**。能进行快速的随机访问，但是往列表中间插入和删除元素的时候比较慢。**ListIterator** 只能用在反向遍历 **ArrayList** 的场合，不要用它来插入和删除元素，因为相比 **LinkedList**，在 **ArrayList** 里面用 **ListIterator** 的系统开销比较高。
LinkedList	对顺序访问进行了优化。在 **List** 中间插入和删除元素的代价也不高。随机访问的速度相对较慢。(用 **ArrayList** 吧。)此外它还有 **addFirst( )**，**addLast( )**，**getFirst( )**，**getLast( )**，**removeFirst( )**和 **removeLast( )**等方法(这些方法，接口和基类均未定义)，你能把它当成栈(stack)，队列(queue)或双向队列(deque)来用。

下面这段程序把各种操作都集中到方法里面：**List** 都能作的事(**basicTest( )**)，用 **Iterator** 在列表中移动(**iterMotion( )**)，修改列表的元素(**iterManipulation( )**)，查看 **List** 的操作结果(**testVisual( )**)，以及 **LinkedList** 所独有的方法。

```
//: c11:List1.java
// Things you can do with Lists.
import java.util.*;
import com.bruceeckel.util.*;

public class List1 {
  public static List fill(List a) {
    Collections2.countries.reset( );
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    Collections2.fill(a, Collections2.countries, 10);
    return a;
  }
  private static boolean b;
  private static Object o;
  private static int i;
  private static Iterator it;
  private static ListIterator lit;
  public static void basicTest(List a) {
    a.add(1, "x"); // Add at location 1
    a.add("x"); // Add at end
    // Add a collection:
    a.addAll(fill(new ArrayList( )));
    // Add a collection starting at location 3:
    a.addAll(3, fill(new ArrayList( )));
    b = a.contains("1"); // Is it in there?
    // Is the entire collection in there?
    b = a.containsAll(fill(new ArrayList( )));
    // Lists allow random access, which is cheap
    // for ArrayList, expensive for LinkedList:
    o = a.get(1); // Get object at location 1
    i = a.indexOf("1"); // Tell index of object
    b = a.isEmpty( ); // Any elements inside?
    it = a.iterator( ); // Ordinary Iterator
    lit = a.listIterator( ); // ListIterator
    lit = a.listIterator(3); // Start at loc 3
    i = a.lastIndexOf("1"); // Last match
    a.remove(1); // Remove location 1
    a.remove("3"); // Remove this object
    a.set(1, "y"); // Set location 1 to "y"
    // Keep everything that's in the argument
    // (the intersection of the two sets):
    a.retainAll(fill(new ArrayList( )));
    // Remove everything that's in the argument:
    a.removeAll(fill(new ArrayList( )));
    i = a.size( ); // How big is it?
    a.clear( ); // Remove all elements
  }
  public static void iterMotion(List a) {
    ListIterator it = a.listIterator( );
    b = it.hasNext( );
    b = it.hasPrevious( );
    o = it.next( );
    i = it.nextIndex( );
    o = it.previous( );
    i = it.previousIndex( );
  }
  public static void iterManipulation(List a) {
    ListIterator it = a.listIterator( );
    it.add("47");
    // Must move to an element after add( ):
    it.next( );
    // Remove the element that was just produced:
    it.remove( );
    // Must move to an element after remove( ):
    it.next( );
    // Change the element that was just produced:
    it.set("47");
  }
  public static void testVisual(List a) {
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    System.out.println(a);
    List b = new ArrayList( );
    fill(b);
    System.out.print("b = ");
    System.out.println(b);
    a.addAll(b);
    a.addAll(fill(new ArrayList( )));
    System.out.println(a);
    // Insert, remove, and replace elements
    // using a ListIterator:
    ListIterator x = a.listIterator(a.size( )/2);
    x.add("one");
    System.out.println(a);
    System.out.println(x.next( ));
    x.remove( );
    System.out.println(x.next( ));
    x.set("47");
    System.out.println(a);
    // Traverse the list backwards:
    x = a.listIterator(a.size( ));
    while(x.hasPrevious( ))
      System.out.print(x.previous( ) + " ");
    System.out.println( );
    System.out.println("testVisual finished");
  }
  // There are some things that only LinkedLists can
do:
  public static void testLinkedList( ) {
    LinkedList ll = new LinkedList( );
    fill(ll);
    System.out.println(ll);
    // Treat it like a stack, pushing:
    ll.addFirst("one");
    ll.addFirst("two");
    System.out.println(ll);
    // Like "peeking" at the top of a stack:
    System.out.println(ll.getFirst( ));
    // Like popping a stack:
    System.out.println(ll.removeFirst( ));
    System.out.println(ll.removeFirst( ));
    // Treat it like a queue, pulling elements
    // off the tail end:
    System.out.println(ll.removeLast( ));
    // With the above operations, it's a dequeue!
    System.out.println(ll);
  }
  public static void main(String[] args) {
    // Make and fill a new list each time:
    basicTest(fill(new LinkedList( )));
    basicTest(fill(new ArrayList( )));
    iterMotion(fill(new LinkedList( )));
    iterMotion(fill(new ArrayList( )));
    iterManipulation(fill(new LinkedList( )));
    iterManipulation(fill(new ArrayList( )));
    testVisual(fill(new LinkedList( )));
    testLinkedList( );
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

为了示范正确的语法，我们让 **basicTest( )**和 **iterMotion( )**调用了一些方法。虽然我们拿了返回的值，但是却没有用它。有时，程序根本就不去拿返回值。写程序之前，你应该到 *java.sun.com* 去看看 JDK 的文档，查一查这些方法的完整用法。

记住，容器只是一个存储对象的盒子。如果这个小盒子能帮你解决所有的问题，那你就用不着去管它是怎么实现的(在绝大多数情况下，这是使用对象的基本概念)。如果开发环境里面还有一些别的，会造成固定的性能开销的因素存在，那么 **ArrayList** 和 **LinkedList** 之间的性能差别就会变得不那么重要了。你只需要它们中的一个。你甚至可以想像有这样一种“完美”的抽象容器：它能根据用途，自动地切换其底层的实现。

用 LinkedList 做一个栈

“栈(stack)”有时也被称为“后进先出”(LIFO)的容器。就是说，最后一个被“压”进栈中的东西，会第一个“弹”出来。同其他 Java 容器一样，压进去和弹出来的东西都是 **Object**，所以除非你只用 **Object** 的功能，否则就必须对弹出来的东西进行类型转换。

LinkedList 的方法能直接实现栈的功能，所以你完全可以不写 **Stack** 而直接使用 **LinkedList**。但是有一个叫“栈”的类能让我们把故事讲的更好：

```
//: c11:StackL.java
// Making a stack from a LinkedList.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class StackL {
  private static Test monitor = new Test( );
  private LinkedList list = new LinkedList( );
  public void push(Object v) { list.addFirst(v); }
  public Object top( ) { return list.getFirst( ); }
  public Object pop( ) { return
list.removeFirst( ); }
  public static void main(String[] args) {
    StackL stack = new StackL( );
    for(int i = 0; i < 10; i++)
      stack.push(Collections2.countries.next( ));
    System.out.println(stack.top( ));
    System.out.println(stack.top( ));
    System.out.println(stack.pop( ));
    System.out.println(stack.pop( ));
    System.out.println(stack.pop( ));
    monitor.expect(new String[] {
      "CHAD",
      "CHAD",
      "CHAD",
      "CENTRAL AFRICAN REPUBLIC",
      "CAPE VERDE"
    });
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  }
} ///:~
```

如果你只想要栈的功能，那么继承就不太合适了。因为继承出来的是一个拥有 **LinkedList** 的所有方法的类(后面你就会看到 Java 1.0 类库的设计者们是怎样栽在 **Stack** 上的)。

用 LinkedList 做一个队列

队列(queue)是一个“先进先出”(FIFO)容器。也就是，你从一端把东西放进去，从另一端把东西取出来。所以你放东西的顺序也就是取东西的顺序。**LinkedList** 有支持队列的功能的方法，所以它也能被当作 **Queue** 来用。

```
//: c11:Queue.java
// Making a queue from a LinkedList.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Queue {
  private static Test monitor = new Test( );
  private LinkedList list = new LinkedList( );
  public void put(Object v) { list.addFirst(v); }
  public Object get( ) { return list.removeLast( ); }
  public boolean isEmpty( ) { return
list.isEmpty( ); }
  public static void main(String[] args) {
    Queue queue = new Queue( );
    for(int i = 0; i < 10; i++)
      queue.put(Integer.toString(i));
    while(!queue.isEmpty( ))
      System.out.println(queue.get( ));
    monitor.expect(new String[] {
      "0",
      "1",
      "2",
      "3",
      "4",
      "5",
      "6",
      "7",
      "8",
      "9"
    });
  }
} ///:~
```

还能很轻易地用 **LinkedList** 做一个 *deque*(双向队列)。它很像队列，只是你可以从任意一端添加和删除元素。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Set 的功能

Set 的接口就是 **Collection** 的，所以不像那两个 **List**，它没有额外的功能。实际上 **Set** 确确实实就是一个 **Collection**——只不过行为方式不同罢了。(这是继承和多态性的完美运用：表达不同地行为。)**Set** 会拒绝持有多个具有相同值的对象的实例(对象的“值”又是由什么决定的呢？这个问题比较复杂，我们以后会讲的)。

Set (接口)	加入 **Set** 的每个元素必须是唯一的；否则，**Set** 是不会把它加进去的。要想加进 **Set**，**Object** 必须定义 **equals( )**，这样才能标明对象的唯一性。**Set** 的接口和 **Collection** 的一模一样。**Set** 的接口不保证它会用哪种顺序来存储元素。
HashSet*	为优化查询速度而设计的 **Set**。要放进 **HashSet** 里面的 **Object** 还得定义 **hashCode( )**。
TreeSet	是一个有序的 **Set**，其底层是一棵树。这样你就能从 **Set** 里面提取一个有序序列了。
LinkedHashSet (JDK 1.4)	一个在内部使用链表的 **Set**，既有 **HashSet** 的查询速度，又能保存元素被加进去的顺序(插入顺序)。用 **Iterator** 遍历 **Set** 的时候，它是按插入顺序进行访问的。

下面这段程序**没有**列出 **Set** 的所有方法，由于它的接口和 **Collection** 的是完全相同的，所以前面那个例子已经讲过了。现在，它要演示的是 **Set** 与众不同的地方：

```
//: c11:Set1.java
// Things you can do with Sets.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Set1 {
  private static Test monitor = new Test( );
  static void fill(Set s) {
    s.addAll(Arrays.asList(
      "one two three four five six seven".split("
")));
  }
  public static void test(Set s) {
    // Strip qualifiers from class name:
    System.out.println(
      s.getClass( ).getName( ).replaceAll("\\w+\\.",
""));
    fill(s); fill(s); fill(s);
    System.out.println(s); // No duplicates!
    // Add another set to this one:
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    s.addAll(s);
    s.add("one");
    s.add("one");
    s.add("one");
    System.out.println(s);
    // Look something up:
    System.out.println("s.contains(\"one\"): " +
      s.contains("one"));
  }
  public static void main(String[] args) {
    test(new HashSet( ));
    test(new TreeSet( ));
    test(new LinkedHashSet( ));
    monitor.expect(new String[] {
      "HashSet",
      "[one, two, five, four, three, seven, six]",
      "[one, two, five, four, three, seven, six]",
      "s.contains(\"one\"): true",
      "TreeSet",
      "[five, four, one, seven, six, three, two]",
      "[five, four, one, seven, six, three, two]",
      "s.contains(\"one\"): true",
      "LinkedHashSet",
      "[one, two, three, four, five, six, seven]",
      "[one, two, three, four, five, six, seven]",
      "s.contains(\"one\"): true"
    });
  }
} ///:~
```

我们往 **Set** 里面加了些重复的元素，但是打印的时候，你就会看到，同一个值，**Set** 只接收一个实例。

程序运行的时候，你就会发现 **HashSet** 保存对象的顺序是和 **TreeSet** 和 **LinkedHashSet** 不一样的。这是因为它们是用不同的方式来存储和查找元素的。(**TreeSet** 用了一种叫红黑树的数据结构『red-black tree data structure』来为元素排序，而 **HashSet** 则用了“专为快速查找而设计”的散列函数。**LinkedHashSet** 在内部用散列来提高查询速度，但是它**看上去**像是用链表来保存元素的插入顺序。)你写自己的类的时候，一定要记住，**Set** 要有一个判断以什么顺序来存储元素的标准，也就是说你必须实现 **Comparable** 接口，并且定义 **compareTo( )**方法。下面就是举例：

```
//: c11:Set2.java
// Putting your own type in a Set.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Set2 {
  private static Test monitor = new Test( );
  public static Set fill(Set a, int size) {
    for(int i = 0; i < size; i++)
      a.add(new MyType(i));
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    return a;
  }
  public static void test(Set a) {
    fill(a, 10);
    fill(a, 10); // Try to add duplicates
    fill(a, 10);
    a.addAll(fill(new TreeSet( ), 10));
    System.out.println(a);
  }
  public static void main(String[] args) {
    test(new HashSet( ));
    test(new TreeSet( ));
    test(new LinkedHashSet( ));
    monitor.expect(new String[] {
      "[2 , 4 , 9 , 8 , 6 , 1 , 3 , 7 , 5 , 0 ]",
      "[9 , 8 , 7 , 6 , 5 , 4 , 3 , 2 , 1 , 0 ]",
      "[0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ]"
    });
  }
} ///:~
```

我们会在本章的后面部分讲解怎样定义 **equals( )**和 **hashCode( )**。无论是用哪种 **Set**，你都应该定义 **equals( )**，但是只有在“要把对象放进 **HashSet**”的情况下，你才需要定义 **hashCode( )**(最好还是定义一个，因为通常情况下 **HashSet** 是 **Set** 的首选)。但是作为一种编程风格，你应该在覆写 **equals( )**的同时把 **hashCode( )**也覆写了。这个过程将在本章的后面部分做再作深入探讨。

注意一下，我没在 **compareTo( )**里面用“简单明了”的 **return i-i2**。虽然这是一个很常见的编程错误，但是如果 **i** 和 **i2** 都是“unsigned” **int** 的话(Java 里面没有，但是假设它有)，那它还是能正常工作的。可是 Java 的 singed **int** 把它给毁了，因为它太小了，不能用来表示两个 signed **int** 的差。假如 **i** 是一个很大的正整数而 **j** 是一个很大的负整数，那么 **i—j** 就会溢出，并返回一个负值，于是程序就出错了。

SortedSet

SortedSet(只有 **TreeSet** 这一个实现可用)中的元素一定是有序的。这使得 **SortedSet** 接口多了一些方法：

Comparator comparator( )：返回 **Set** 所使用的 **Comparator** 对象，或者用 null 表示它使用 **Object** 自有的排序方法。

Object first( ): 返回最小的元素。

Object last( ): 返回最大的元素。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

SortedSet subSet(fromElement, toElement): 返回 **Set** 的子集，其中的元素从 **fromElement** 开始到 **toElement** 为止(包括 **fromElement**，不包括 **toElement**)。

SortedSet headSet(toElement)：返回 **Set** 的子集，其中的元素都应小于 **toElement**。

SortedSet headSet(toElement)：返回 **Set** 的子集，其中的元素都应大于 **fromElement**。

下面是一个简单的示例：

```
//: c11:SortedSetDemo.java
// What you can do with a TreeSet.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class SortedSetDemo {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    SortedSet sortedSet = new TreeSet(Arrays.asList(
    "one two three four five six seven
eight".split(" ")));
    System.out.println(sortedSet);
    Object
      low = sortedSet.first( ),
      high = sortedSet.last( );
    System.out.println(low);
    System.out.println(high);
    Iterator it = sortedSet.iterator( );
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next( );
      if(i == 6) high = it.next( );
      else it.next( );
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedSet.subSet(low, high));
    System.out.println(sortedSet.headSet(high));
    System.out.println(sortedSet.tailSet(low));
    monitor.expect(new String[] {
      "[eight, five, four, one, seven, six, three,
two]",
      "eight",
      "two",
      "one",
      "two",
      "[one, seven, six, three]",
      "[eight, five, four, one, seven, six, three]",
      "[one, seven, six, three, two]"
    });
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

注意，**SortedSet** 意思是“根据对象的比较顺序”，而不是“插入顺序” 进行排序。

Map 的功能

ArrayList 能让你用数字在一个对象序列里面进行选择，所以从某种意义上讲，它是将数字和对象关联起来。但是，如果你想根据其他条件在一个对象序列里面进行选择的话，那又该怎么做呢？栈就是一个例子。它的标准是“选取最后一个被压入栈的对象”。我们常用的术语 *map*，*dictionary*，或 *associative array* 就是一种非常强大的，能在序列里面进行挑选的工具(前一章的 **AssociativeArray.java** 里面已经有一个简单的例子了)。从概念上讲，它看上去像是一个 **ArrayList**，但它不用数字，而是用另一个对象来查找对象！这是一种至关重要的编程技巧。

这一概念在 Java 中表现为 **Map**。**put(Object key, Object value)** 方法会往 **Map** 里面加一个值，并且把这个值同键(你查找时所用的对象)联系起来。给出键之后，**get(Object key)**就会返回与之相关联的值。你也可以用 **containsKey( )** 和 **containsValue( )**测试 **Map** 是否包含有某个键或值。

Java 标准类库里有好几种 **Map**：**HashMap**，**TreeMap**，**LinkedHashMap**，**WeakHashMap**，以及 **IdentityHashMap**。它们都实现了 **Map** 的基本接口，但是在行为方式方面有着明显的差异。这些差异体现在，效率，持有和表示对象 pair 的顺序，持有对象的时间长短，以及如何决定键的相等性。

性能是 **Map** 所要面对的一个大问题。如果你知道 **get( )**是怎么工作的，你就会发觉(比方说)在 **ArrayList** 里面找对象会是相当慢的。而这正是 **HashMap** 的强项。它不是慢慢地一个个地找这个键，而是用了一种被称为 *hash code* 的特殊值来进行查找的。散列(hash)是一种算法，它会从目标对象当中提取一些信息，然后生成一个表示这个对象的“相对独特”的 **int**。**hashCode( )**是 **Object** 根类的方法，因此所有 Java 对象都能生成 hash code。**HashMap** 则利用对象的 **hashCode( )**来进行快速的查找。这样性能就有了急剧的提高。[8]

Map (接口)	维持键一值的关联(即 pairs)，这样就能用键来找值了。
HashMap*	基于 hash 表的实现。(用它来代替

[8]如果这一性能仍不能满足你的要求，你可以为你自己的类定制一个 **Map**，这样就能避免类型传递所引起的延时，从而进一步提高查询的速度。如果还想获得更高的性能，速度狂们可以参考 Donald Knuth 的 *The Art of Computer Programming*，第二版的第三卷：*Sorting and Searching*，用数组来代替“溢出的桶列表(overflow bucket lists)”；这样做会有两个好处：一是能针对磁盘存储做优化，二是在创建和回收单独记录的时候能节省很多时间。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

	Hashtable。)提供时间恒定的插入与查询。在构造函数中可以设置 hash 表的 *capacity* 和 *load factor*。可以通过构造函数来调节其性能。
LinkedHashMap (JDK 1.4)	很像 **HashMap**，但是用 **Iterator** 进行遍历的时候，它会按插入顺序或最先使用的顺序(least-recently-used (LRU) order)进行访问。除了用 **Iterator** 外，其他情况下，只是比 **HashMap** 稍慢一点。用 **Iterator** 的情况下，由于是使用链表来保存内部顺序，因此速度会更快。
TreeMap	基于红黑树数据结构的实现。当你查看键或 pair 时，会发现它们是按顺序 (根据 **Comparable** 或 **Comparator**，我们过一会讲)排列的。**TreeMap** 的特点是，你所得到的是一个有序的 **Map**。**TreeMap** 是 **Map** 中唯一有 **subMap( )**方法的实现。这个方法能让你获取这个树中的一部分。
WeakHashMap	一个 *weak key* 的 **Map**，是为某些特殊问题而设计的。它能让 **Map** 释放其所持有的对象。如果某个对象除了在 **Map** 当中充当键之外，在其它地方都没有其 reference 的话，那它将被当作垃圾回收。
IdentityHashMap (JDK 1.4)	一个用**==**，而不是 **equals( )**来比较键的 hash map。不是为我们平常使用而设计的，是用来解决特殊问题的。

散列是往 **Map** 里存数据的常用算法。有时你会需要知道散列算法的工作细节，所以我们会稍后再讲。

下面这段程序用到了 **Collections2.fill( )**方法，以及我们前面准备的测试数据：

```
//: c11:Map1.java
// Things you can do with Maps.
import java.util.*;
import com.bruceeckel.util.*;

public class Map1 {
  private static Collections2.StringPairGenerator
geo =
    Collections2.geography;
  private static
Collections2.RandStringPairGenerator
    rsp = Collections2.rsp;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  // Producing a Set of the keys:
  public static void printKeys(Map map) {
    System.out.print("Size = " + map.size( ) + ", ");
    System.out.print("Keys: ");
    System.out.println(map.keySet( ));
  }
  public static void test(Map map) {
    // Strip qualifiers from class name:
    System.out.println(

map.getClass( ).getName( ).replaceAll("\\w+\\.",
""));
    Collections2.fill(map, geo, 25);
    // Map has 'Set' behavior for keys:
    Collections2.fill(map, geo.reset( ), 25);
    printKeys(map);
    // Producing a Collection of the values:
    System.out.print("Values: ");
    System.out.println(map.values( ));
    System.out.println(map);
    String key = CountryCapitals.pairs[4][0];
    String value = CountryCapitals.pairs[4][1];
    System.out.println("map.containsKey(\"" + key +
      "\"): " + map.containsKey(key));
    System.out.println("map.get(\"" + key + "\"): "
      + map.get(key));
    System.out.println("map.containsValue(\""
      + value + "\"): " + map.containsValue(value));
    Map map2 = new TreeMap( );
    Collections2.fill(map2, rsp, 25);
    map.putAll(map2);
    printKeys(map);
    key =
map.keySet( ).iterator( ).next( ).toString( );
    System.out.println("First key in map: " + key);
    map.remove(key);
    printKeys(map);
    map.clear( );
    System.out.println("map.isEmpty( ): " +
map.isEmpty( ));
    Collections2.fill(map, geo.reset( ), 25);
    // Operations on the Set change the Map:
    map.keySet( ).removeAll(map.keySet( ));
    System.out.println("map.isEmpty( ): " +
map.isEmpty( ));
  }
  public static void main(String[] args) {
    test(new HashMap( ));
    test(new TreeMap( ));
    test(new LinkedHashMap( ));
    test(new IdentityHashMap( ));
    test(new WeakHashMap( ));
  }
} ///:~
```

PrintKeys( )和 **printValues( )**不但是非常实用的工具，而且还演示了怎样将 **Map** 转换成 **Collection**。**KeySet( )**方法返回了一个由

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Map 的键所组成的 **Set**。**values( )**也差不多，它返回的是由 **Map** 的值所组成的 **Collection**。(注意，键必须是唯一的，而值却可以有重复。)由于这些 **Collection** 的后台都是 **Map**，因此对 **Collection** 的任何修改都会反映到 **Map** 上。

接下来，我们简单地示范了一下各种 **Map** 的操作，然后挨个测试了一下 **Map**。

举一个应用 **HashMap** 的例子，想一想，怎样才能写一个检测 Java **Random** 类的随机性的程序。在理想情况下，它应该能生成一个分布均匀的随机数。不过这只是一个测试，所以我们只要数一下随机数落在各个区域的次数就可以了。要把两个对象关联起来，用 **HashMap** 正合适。(这里要 map 的就是 **Math.random( )**所返回的随机数，以及这个数字出现的次数)：

```
//: c11:Statistics.java
// Simple demonstration of HashMap.
import java.util.*;

class Counter {
  int i = 1;
  public String toString( ) { return
Integer.toString(i); }
}

public class Statistics {
  private static Random rand = new Random( );
  public static void main(String[] args) {
    Map hm = new HashMap( );
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      Integer r = new Integer(rand.nextInt(20));
      if(hm.containsKey(r))
        ((Counter)hm.get(r)).i++;
      else
        hm.put(r, new Counter( ));
    }
    System.out.println(hm);
  }
} ///:~
```

main( )会把随机数转换成 **Integer**，这样 **HashMap** 就有 reference 可用了。(容器不能存储 primitive，只能存储对象的 reference。)**containsKey( )**会检查这个键是不是已经在容器里了(也就是说，找没找到这个数)。如果找到了，**get( )**方法就会返回与这个键相关联的值，也就是 **Counter** 对象。**Counter** 是一个计数器，每找到一个随机数的时候，它就对 **i** 做递增。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

如果没有找到这个键，**put( )**方法就会往 **HashMap** 里面插一个新的 pair。**Counter** 会自动地将变量 **i** 的值初始化为 1，这表示第一次插入了这个随机数。

如果想观察 **HashMap**，直接打印就是了。**HashMap** 的 **toString( )** 方法会遍历每一个 pair，然后调用它们的 **toString( )**方法。**Integer.toString( )**方法已经预先定义过了，这样你就能看懂 **Counter** 的 **toString( )**了。下面就是某次运行的输出：

```
{15=529, 4=488, 19=518, 8=487, 11=501, 16=487,
18=507, 3=524, 7=474, 12=485, 17=493, 2=490, 13=540,
9=453, 6=512, 1=466, 14=522, 10=471, 5=522, 0=531}
```

也许你会觉得奇怪，为什么要用 **Counter** 类呢，好像它的功能还没有 **Integer** 的 wrapper 强，为什么不用 **int** 或 **Integer** 呢？好，首先我们不能使用 **int**，因为容器只能持有 **Object** 对象。知道了这点，你或许会认为应该用 wrapper 类，因为你可以通过它把 primitive 放入容器。但是对 Java 的 wrapper 类来说，你唯一能做的就是用某个值来对它进行初始化，然后把这个值读出来。也就是说一旦创建了一个 wrapper 类对象，你就再也不能修改它的值了。就这个问题而言，**Integer** 的 wrapper 类是无能为力的。所以我们只能创建一个类来解决这个问题。

SortedMap

SortedMap(只有 **TreeMap** 这一个实现)的键肯定是有序的，因此这个接口里面就有一些附加功能的方法了。

Comparator comparator( )：返回 **Map** 所使用的 **comparator**，如果是用 **Object** 内置的方法的话，则返回 **null**。

Object firstKey( ): 返回第一个键。

Object lastKey( ): 返回最后一个键。

SortedMap subMap(fromKey, toKey): 返回这个 **Map** 的一个子集，其键从 **fromKey** 开始到 **toKey** 为止，包括前者，不包括后者。

SortedMap headMap(toKey): 返回这个 **Map** 的一个子集，其键均小于 **toKey**。

SortedMap tailMap(fromKey): 返回这个 **Map** 的一个子集，其键均大于等于 **fromKey**。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

下面这个例子，有点像 **SortedSetDemo.java**，它演示了 **TreeMap** 的这些附加功能：

```
//: c11:SimplePairGenerator.java
import com.bruceeckel.util.*;
//import java.util.*;

public class SimplePairGenerator implements
MapGenerator {
  public Pair[] items = {
    new Pair("one", "A"), new Pair("two", "B"),
    new Pair("three", "C"), new Pair("four", "D"),
    new Pair("five", "E"), new Pair("six", "F"),
    new Pair("seven", "G"), new Pair("eight", "H"),
    new Pair("nine", "I"), new Pair("ten", "J")
  };
  private int index = -1;
  public Pair next( ) {
    index = (index + 1) % items.length;
    return items[index];
  }
  public static SimplePairGenerator gen =
    new SimplePairGenerator( );
} ///:~



//: c11:SortedMapDemo.java
// What you can do with a TreeMap.
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class SortedMapDemo {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    TreeMap sortedMap = new TreeMap( );
    Collections2.fill(
      sortedMap, SimplePairGenerator.gen, 10);
    System.out.println(sortedMap);
    Object
      low = sortedMap.firstKey( ),
      high = sortedMap.lastKey( );
    System.out.println(low);
    System.out.println(high);
    Iterator it = sortedMap.keySet( ).iterator( );
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next( );
      if(i == 6) high = it.next( );
      else it.next( );
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedMap.subMap(low, high));
    System.out.println(sortedMap.headMap(high));
    System.out.println(sortedMap.tailMap(low));
    monitor.expect(new String[] {
      "{eight=H, five=E, four=D, nine=I, one=A,
seven=G," +
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        " six=F, ten=J, three=C, two=B}",
        "eight",
        "two",
        "nine",
        "ten",
        "{nine=I, one=A, seven=G, six=F}",
        "{eight=H, five=E, four=D, nine=I, " +
        "one=A, seven=G, six=F}",
        "{nine=I, one=A, seven=G, six=F, " +
        "ten=J, three=C, two=B}"
      });
  }
} ///:~
```

这里，pair 是按 key 的顺序存储的。由于 **TreeMap** 有顺序的概念，因此“位置”是有意义的，所以你可以去获取它的第一个和最后一个元素，以及它的子集。

LinkedHashMap

为了提高速度，**LinkedHashMap** 对所有东西都作了 hash，而且遍历的时候(**println( )**会遍历整个 **Map**，所以你能看到这个过程)还会按插入顺序返回 pair。此外，你还可以在 **LinkedHashMap** 的构造函数里面进行配置，让它使用基于访问的 LRU(*least-recently-used*)算法，这样还没被访问过的元素(同时也是要删除的候选对象)就会出现在队列的最前头。这样，为节省资源而写一个定时清理的程序就变得很简单了。下面这段程序就演示了这两个特性：

```
//: c11:LinkedHashMapDemo.java
// What you can do with a LinkedHashMap.
import com.bruceeckel.simpletest.*;
import com.bruceeckel.util.*;
import java.util.*;

public class LinkedHashMapDemo {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    LinkedHashMap linkedMap = new LinkedHashMap( );
    Collections2.fill(
      linkedMap, SimplePairGenerator.gen, 10);
    System.out.println(linkedMap);
    // Least-recently used order:
    linkedMap = new LinkedHashMap(16, 0.75f, true);
    Collections2.fill(
      linkedMap, SimplePairGenerator.gen, 10);
    System.out.println(linkedMap);
    for(int i = 0; i < 7; i++) // Cause accesses:

linkedMap.get(SimplePairGenerator.gen.items[i].key);
    System.out.println(linkedMap);

linkedMap.get(SimplePairGenerator.gen.items[0].key);
    System.out.println(linkedMap);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      monitor.expect(new String[] {
        "{one=A, two=B, three=C, four=D, five=E, " +
         "six=F, seven=G, eight=H, nine=I, ten=J}",
        "{one=A, two=B, three=C, four=D, five=E, " +
         "six=F, seven=G, eight=H, nine=I, ten=J}",
        "{eight=H, nine=I, ten=J, one=A, two=B, " +
         "three=C, four=D, five=E, six=F, seven=G}",
        "{eight=H, nine=I, ten=J, two=B, three=C, " +
         "four=D, five=E, six=F, seven=G, one=A}"
      });
    }
} ///:~
```

你可以从程序的输出看到，遍历的顺序真的就是 pair 插入的顺序，甚至 LRU 版的也是。但是仅仅访问过前七个元素之后，后面三个就被移到了队列的最前头。但是当我们访问了其中的某个之后，它又被移到队列的后面去了。

散列算法与 Hash 数

Statistics.java 用标准类库的 **Integer** 来充当 **HashMap** 的键。由于它具备了键所应必备的一切功能，因此这个程序能很正常地运行。但是当你用自己写的类来充当键的时候，就会有一些问题了。比如在一个气象预报系统里面，你把 **Groundhog** 和 **Prediction** 对象配对。这种设计似乎很简单，你创建两个类，然后把 **Groundhog** 当作键，把 **Prediction** 当作值。

```
//: c11:Groundhog.java
// Looks plausible, but doesn't work as a HashMap
key.

public class Groundhog {
  protected int number;
  public Groundhog(int n) { number = n; }
  public String toString( ) {
    return "Groundhog #" + number;
  }
} ///:~
```

```
//: c11:Prediction.java
// Predicting the weather with groundhogs.

public class Prediction {
  private boolean shadow = Math.random( ) > 0.5;
  public String toString( ) {
    if(shadow)
      return "Six more weeks of Winter!";
    else
      return "Early Spring!";
  }
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
} ///:~
```

```
//: c11:SpringDetector.java
// What will the weather be?
import com.bruceeckel.simpletest.*;
import java.util.*;
import java.lang.reflect.*;

public class SpringDetector {
  private static Test monitor = new Test( );
  // Uses a Groundhog or class derived from
Groundhog:
  public static void
  detectSpring(Class groundHogClass) throws
Exception {
    Constructor ghog =
groundHogClass.getConstructor(
      new Class[] {int.class});
    Map map = new HashMap( );
    for(int i = 0; i < 10; i++)
      map.put(ghog.newInstance(
        new Object[]{ new Integer(i) }), new
Prediction( ));
    System.out.println("map = " + map + "\n");
    Groundhog gh = (Groundhog)
      ghog.newInstance(new Object[]{ new
Integer(3) });
    System.out.println("Looking up prediction for "
+ gh);
    if(map.containsKey(gh))
      System.out.println((Prediction)map.get(gh));
    else
      System.out.println("Key not found: " + gh);
  }
  public static void main(String[] args) throws
Exception {
    detectSpring(Groundhog.class);
    monitor.expect(new String[] {
      "%% map = \\{(Groundhog #\\d=" +
      "(Early Spring!|Six more weeks of Winter!)" +
      "(, )?){10}\\}",
      "",
      "Looking up prediction for Groundhog #3",
      "Key not found: Groundhog #3"
    });
    }
} ///:~
```

每个 **Groundhog** 都有一个标识值，这样你就可以用这种方式，“给我与 **Groundhog #3** 相关联的 **Prediction**”，在 **HashMap** 里面查找 **Prediction** 了。**Prediction** 类包含了一个用 **Math.random( )** 初始化的 **boolean** 值，和一个把结果翻译出来的 **toString( )**方法。**detectSpring( )**要用 reflection 的方式创建 **Groundhog** 或是其派

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

生类的实例，并且调用其方法。这所以要用这种方法是因为，就这个问题而言，如果要继承 **Groundhog** 的话，用这种方法会很顺手。我们用 **Groundhog** 和与之相关的 **Prediciton** 填 **HashMap**。接下来我们要打印 **HashMap**，这样你就能看到它真的是被填满了。然后我们就要用标识为#3 的 **Groundhog** 来查找 **Prediction** 了(它肯定在 **Map** 里面)。

好像很简单，但是就是不能运行。毛病就出在，**Groundhog** 是继承 **Object** 根类的(如果你不指明它的父类，它就自动继承根类，而最终所有的类都继承 **Object**)。这样，对象的 **hash** 数是由 **Object** 的 **hashCode( )**生成的，缺省情况下这就是对象的内存地址。这样 **Groundhog(3)**的第一个实例的 **hash** 数会与其第二个实例的 **hash** 数不相符。而后者正是我们用来查找的键。

或许你会认为，你所要做的就是覆写一个合适的 **hashCode( )**。但是除非你再作另一件事，把同属 **Object** 的 **equals( )**方法也覆写了，否则还是不行。**HashMap** 要用 **equals( )**来判断查询用的键是不是与表里面的其它键相等。

一个合适的 **equals( )**必须做到以下五点：

1. 反身性：对任何 **x**，**x.equals(x)**必须是 **true** 的。
2. 对称性：对任何 **x** 和 **y**，如果 **y.equals(x)**是 **true** 的，那么 **x.equals(y)**也必须是 **true** 的。
3. 传递性：对任何 **x**，**y** 和 **z**，如果 **x.equals(y)**是 **true** 的，且 **y.equals(z)**也是 **true** 的，那么 **x.equals(z)**也必须是 **true** 的。
4. 一致性：对任何 **x** 和 **y**，如果对象里面用来判断相等性的信息没有修改过，那么无论调用多少次 **x.equals(y)**，它都必须一致地返回 **true** 或 **false**。
5. 对于任何非空的 **x**，**x.equals(null)**必须返回 **false**。

默认的 **Object.equals( )**只是简单地比较两个对象的地址，所以一个 **Groundhog(3)**会不等于另一个 **Groundhog(3)**。因此，如果你想把你自己写的类当 **HashMap** 的键来用的话，你就必须把 **hashCode( )**和 **equals( )**都给覆写了，就像下面这个程序：

```
//: c11:Groundhog2.java
// A class that's used as a key in a HashMap
// must override hashCode( ) and equals( ).

public class Groundhog2 extends Groundhog {
  public Groundhog2(int n) { super(n); }
  public int hashCode( ) { return number; }
  public boolean equals(Object o) {
    return (o instanceof Groundhog2)
      && (number == ((Groundhog2)o).number);
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
//: c11:SpringDetector2.java
// A working key.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class SpringDetector2 {
  private static Test monitor = new Test( );
  public static void main(String[] args) throws
Exception {
    SpringDetector.detectSpring(Groundhog2.class);
    monitor.expect(new String[] {
      "%% map = \\{(Groundhog #\\d=" +
      "(Early Spring!|Six more weeks of Winter!)" +
      "(, )?){10}\\}",
      "",
      "Looking up prediction for Groundhog #3",
      "%% Early Spring!|Six more weeks of Winter!"
    });
  }
} ///:~
```

Groundhog2.hashCode( ) 会以 groundhog 的序号为 **hash** 数，并且返回这个数字。在这个程序里，程序员要责任两个 groundhog 不能拥有相同的序号。**HashCode( )**并不要求你一定要返回一个唯一的标识符(这章学完之后你就会有更深的认识了)，可是 **equals( )**方法就必须能准确地判断两个对象是否相等了。这个 **equals( )**是根据 **groundhog** 的序号来作判断的，所以如果 **HashMap** 的键里有两个序号相同的 **Groundhog2** 的话，程序就出错了。

虽然 **equals( )**好像只检查了参数是不是 **Groundhog2** 类型的(用了第十章讲的 **instanceof** 关键词)，但实际上它还悄悄地作了一个健全性检查，也就是检查一下这个对象是不是 **null** 的，因为如果等号的左边是 **null** 的话，**instanceof** 会返回 **false**。如果类型正确且不为空，**equals( )**就要根据 **ghNumber** 进行比较了。你可以从程序的输出看到，现在运行正常了。

在 **HashSet** 中使用自建对象所应注意的问题，同把它用于 **HashMap** 的键是相同的。

理解 hashCode( )

前面那个例子只是在解决问题的正确方向上迈出了一小步。它告诉我们，如果你不覆写键的 **hashCode( )**和 **equals( )**的话，散列数据结构(**HashSet**，**HashMap**，**LinkedHashSet**，或 **LinkedHashMap**)就没法正确地处理键。但是要想提供一个好的解决方案，你还必须知道散列数据结构究竟是怎样运行的。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

首先想想我们为什么要用散列：要通过一个对象来查找另一个对象。不过 **TreeSet** 或 **TreeMap** 也能做这件事。当然，你也可以实现一个你自己的 **Map**。这么做的前提是，先得定义一个会返回 **Map.Entry** 对象集合的 **Map.entrySet( )**方法。我们为 **Map.Entry** 定义一个新的 **MPair** 类。要想把它放到 **TreeSet** 里面，就得定义它的 **equals( )**，并且实现 **Comparable**:

```
//: c11:MPair.java
// A new type of Map.Entry.
import java.util.*;

public class MPair implements Map.Entry, Comparable
{
  private Object key, value;
  public MPair(Object k, Object v) {
    key = k;
    value = v;
  }
  public Object getKey( ) { return key; }
  public Object getValue( ) { return value; }
  public Object setValue(Object v) {
    Object result = value;
    value = v;
    return result;
  }
  public boolean equals(Object o) {
    return key.equals(((MPair)o).key);
  }
  public int compareTo(Object rv) {
    return
((Comparable)key).compareTo(((MPair)rv).key);
  }
} ///:~
```

注意，它只对键进行比较，因此值完全可以是重复的。

下面用一对 **ArrayList** 实现一个 **Map**。

```
//: c11:SlowMap.java
// A Map implemented with ArrayLists.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class SlowMap extends AbstractMap {
  private static Test monitor = new Test( );
  private List
    keys = new ArrayList( ),
    values = new ArrayList( );
  public Object put(Object key, Object value) {
    Object result = get(key);
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      } else
        values.set(keys.indexOf(key), value);
      return result;
    }
    public Object get(Object key) {
      if(!keys.contains(key))
        return null;
      return values.get(keys.indexOf(key));
    }
    public Set entrySet( ) {
      Set entries = new HashSet( );
      Iterator
        ki = keys.iterator( ),
        vi = values.iterator( );
      while(ki.hasNext( ))
        entries.add(new MPair(ki.next( ), vi.next( )));
      return entries;
    }
    public String toString( ) {
      StringBuffer s = new StringBuffer("{");
      Iterator
        ki = keys.iterator( ),
        vi = values.iterator( );
      while(ki.hasNext( )) {
        s.append(ki.next( ) + "=" + vi.next( ));
        if(ki.hasNext( )) s.append(", ");
      }
      s.append("}");
      return s.toString( );
    }
    public static void main(String[] args) {
      SlowMap m = new SlowMap( );
      Collections2.fill(m, Collections2.geography, 15);
      System.out.println(m);
      monitor.expect(new String[] {
        "{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-
Novo,"+
        " BOTSWANA=Gaberone, BURKINA FASO=Ouagadougou,
" +
        "BURUNDI=Bujumbura, CAMEROON=Yaounde, " +
        "CAPE VERDE=Praia, CENTRAL AFRICAN
REPUBLIC=Bangui,"+
        " CHAD=N'djamena, COMOROS=Moroni, " +
        "CONGO=Brazzaville, DJIBOUTI=Dijibouti, " +
        "EGYPT=Cairo, EQUATORIAL GUINEA=Malabo}"
      });
    }
} ///:~
```

put( )只是简单地把键和值放入相应的 **ArrayList**。**main( )**会装载并打印一个 **SlowMap**，并以此证明它真的能工作。

这段程序告诉我们，造一个新的 **Map** 并不是很难。但是就像它的名字所说的，**SlowMap** 不可能快，所以如果还有其它选择，最好不要用它。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

问题就出在查找键的算法上；键的排列是无序的，所以只能按照顺序一个一个地找，而这是最慢的方法。

散列的价值就在于速度：散列算法能很快地找出东西。由于问题是出在键的查找速度上，那么我们就可以用下面这个办法，把键按顺序排好，然后用 **Collections.binarySearc( )**进行查找(本章的最后会有一个练习，它会带着你走完这个过程)。

散列则走得更远，它的意思是，你不用管了，我会帮你把键存到**某个**你能很快找到的地方。正如本章前面所说的，数组是最快的数据结构，所以我们用它表示键的信息(注意我说“键的信息”，而不是键本身)。此外，本章还说了，数组一经分配就不能调整大小，所以这里又有一个问题：**Map** 要能存储任意数量的 pair，而键的数量又被数组的大小给定死了，该怎么办呢？

答案就是，不用数组来存储键。键对象会生成一个数字，而我们要用这个数字作下标来访问数组。这个数字就是所谓的 **hash** 数。它是由 **Object** 定义的 **hashCode( )**生成的(用计算机科学的术语，就是散列函数)，而先前我们已经要求你在类的定义里面覆写这个方法了。要想解决定长数组的问题，就得允许多个键生成同一个 hash 数。也就是说会有**冲突**。于是数组多大就变得无关紧要了；每个键对象都会落到数组的某个位置上了。

所以查找过程是从计算 hash 数开始的，算完之后再用这个数在数组里定位。如果散列函数能够确保不产生冲突(如果对象数量是固定的话，那么这是有可能的)，那么它就被称为“完全散列函数”，不过这只是特例。通常情况下，冲突是由“外部链(external chaining)”处理的：数组并不是直接指向对象，而是指向一个对象的列表。然后再用 **equals( )**在这个列表中一个一个地找。当然，这一步是比较慢的，但如果散列函数定义得好，每个 hash 数就会只对应几个对象。这样，与搜寻整个序列相比，你能很快地跳到这个组，然后只比较几个对象。这样会快许多，而这也是 **HashMap** 为什么这么快的原因了。

知道这些基础知识之后，你就能用 hash 实现一个简单 **Map** 了：

```
//: c11:SimpleHashMap.java
// A demonstration hashed Map.
import java.util.*;
import com.bruceeckel.util.*;

public class SimpleHashMap extends AbstractMap {
  // Choose a prime number for the hash table
  // size, to achieve a uniform distribution:
  private static final int SZ = 997;
  private LinkedList[] bucket = new LinkedList[SZ];
  public Object put(Object key, Object value) {
    Object result = null;
    int index = key.hashCode( ) % SZ;
    if(index < 0) index = -index;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      if(bucket[index] == null)
        bucket[index] = new LinkedList( );
      LinkedList pairs = bucket[index];
      MPair pair = new MPair(key, value);
      ListIterator it = pairs.listIterator( );
      boolean found = false;
      while(it.hasNext( )) {
        Object iPair = it.next( );
        if(iPair.equals(pair)) {
          result = ((MPair)iPair).getValue( );
          it.set(pair); // Replace old with new
          found = true;
          break;
        }
      }
      if(!found)
        bucket[index].add(pair);
      return result;
    }
    public Object get(Object key) {
      int index = key.hashCode( ) % SZ;
      if(index < 0) index = -index;
      if(bucket[index] == null) return null;
      LinkedList pairs = bucket[index];
      MPair match = new MPair(key, null);
      ListIterator it = pairs.listIterator( );
      while(it.hasNext( )) {
        Object iPair = it.next( );
        if(iPair.equals(match))
          return ((MPair)iPair).getValue( );
      }
      return null;
    }
    public Set entrySet( ) {
      Set entries = new HashSet( );
      for(int i = 0; i < bucket.length; i++) {
        if(bucket[i] == null) continue;
        Iterator it = bucket[i].iterator( );
        while(it.hasNext( ))
          entries.add(it.next( ));
      }
      return entries;
    }
    public static void main(String[] args) {
      SimpleHashMap m = new SimpleHashMap( );
      Collections2.fill(m, Collections2.geography, 25);
      System.out.println(m);
    }
  } ///:~
```

由于 hash 表的“槽位”常常被称为 *bucket*(桶)，因此代表这个 hash 表的数组就被命名为 **bucket**。为了提高分配的平均性，bucket 的数目

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

通常是一个质数。[9]注意它是一个 **LinkedList** 的数组，而这个 **LinkedList** 是为冲突准备的；新的对象会被直接加到 list 的最后。

put( )的返回值要么是 **null**，要么，如果键已经在 list 里面的话，就是原先那个与键相关联的值。返回值 **result** 会被先初始化为 **null**，但是如果在 list 里发现键的话，它会重新设置这个键的值。

put( )和 **get( )**做的第一件事都是调用键的 **hashCode( )**。如果 **hashCode( )**返回的是负数的话，它还要把这个负数转换成正数。接着，它用数组的大小对这个数取模，并以此确定要把这对 pair 放到 **bucket** 的哪个位置上。如果这个位置还是 **null** 的话，就说明此前还没有别的元素被 **hash** 到这里，因此要创建了一个新的 **LinkedList** 来持有这对 pair。但是一般情况下是先搜索 list，看看是否有重复元素，如果有，就把原先那个值赋给 **result**，再用新的值来代替旧的。**found** 负责跟踪是否发现旧的 pair，如果没发现，它就会把新 pair 会加到 list 的最后。

你会发现，**get( )**的代码同 **put( )**的很相似，只是稍许简单了一些。index 也是用来计算 **bucket** 数组里的位置的，如果这里有 **LinkedList**，它就进行搜索，然后找出一个匹配的。

entrySet( ) 会找出所有的 list，并且一个一个地进行遍历，然后把结果加进要返回的 **Set**。写完这个方法之后，你就能往 **Map** 里面填数据了，于是也能进行打印和测试了。

影响 HashMap 性能的因素

要想听懂这个问题，显得了解下面几个术语：

Capacity：hash 表里面的 bucket 的数量。

Initial capacity：创建 hash 表时，bucket 的数量。**HashMap** 和 **HashSet** 都有能让你指定 Initial capacity 的构造函数。

Size：当前 hash 表的记录的数量。

Load factor：size/capacity。load factor 为 0 表示这是一个空表，0.5 表示这是一个半满的表，以此类推。一个负载较轻表会有较少的冲突，因此插入和查找的速度会比较快(但是在用迭代器遍历的时候会比较慢)。**HashMap** 和 **HashSet** 都提供了能指定 load factor 的构造函

[9]事实证明质数实际上并不是一个理想的 hash buckets 的数目，(经过广泛测试)Java 最新的 hash 实现使用二的整数次方。对处理器来说，除法和取模是最慢的两种操作，所以如果 hash 表的长度是二的整数次方的话，它就能用掩码来代替除法了。由于 get( )的使用频率要比其它操作多的多，因此很大一部分操作都要牵涉到取模(%)，而这种方法能限制它对性能所产生的消极影响 (也有可能会对 hashCode( )方法产生一些影响。)

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

数，也就是说当 load factor 达到这个阀值的时候，容器会自动地将 capacity(bucket 的数量) 增加大约一倍，然后将现有的对象分配到新的 bucket 里面(这就是所谓的 rehash。)

缺省情况下**HashMap**会使用0.75的load factor(也就是说，当表的 3/4 被填满的时候，它再开始 rehash)。这看上去是一个不错的折中。laod factor 再高上去的话，确实会降低对存储空间的要求，但是这样做会使查询的速度变慢。这个影响就大了，因为绝大多数情况下，你都要用到查询(包括 **get( )**和 **put( )**)。

如果你知道要在 **HashMap** 里面存储很多记录，那么创建的时候就应该把 initial capacity 设成一个比较合适的大小，这样就能预防自动的 rehash 所引起的性能开销了。[10]

覆写 hashCode( )

现在你已经知道 **HashMap** 都涉及了哪些功能，因此我们可以讲 **hashCode( )**了。

首先，你控制的不是那个在 bucket 数组里面进行检索的值。这个值要取决于 **HashMap** 的 capacity，load factor，以及随容器存储对象的数量变化而引起的 capacity 的变化。**hashCode( )**所返回的值还要做进一步处理，这样才能得到 bucket 数组的下标(在 **SimpleHashMap** 例程中，我们只是简单地对 bucket 数组的容量取模)。

创建 **hashCode( )**最重要的一点就是，对同一个对象，无论在什么时候调用 **hashCode( )**，它都应该返回同一个值。如果对象被 **put( )**进 **HashMap** 的时候，**hashCode( )**是一个值，**get( )**出来的时候，**hashCode( )**是另一个值，那么你就没法提取对象了。所以，如果 **hashCode( )**用到了对象的可变数据的话，你就应该让用户知道，由于 **hashCode( )**的缘故，修改这些数据实际上是在创建一个不同的键。

此外，你大概也**不会**根据对象的“唯一性信息(unique object information)”来生成 **hashCode( )**——特别是 **this**，这会是一个很糟糕的 **hashCode( )**。因为一般情况下，你会把一个『键－值 pair』直接 **put( )**进 **HashMap**，而用了这种 **hashCode( )**之后，你就不能这么做了。**SpringDetector.java** 讲的就是这个问题。由于缺省的

[10]私下里，Joshua Bloch 写道:"...我相信，让用户通过 API 来调整实现的细节，(比如 hash 表的大小，load factor 是多少等等)，是在犯错误。可能是应该让客户来决定他要多大的容量，我们应该在这种地方听取他们的要求。要叫客户去挑选参数，那他们多半会选错。举个极端的例子，**Vector** 的 capacityIncrement。没人会去动它，我们也不提供工具。但是如果你把它设成非零的值的话，顺序添加操作的边际代价会从线性的变成两次的(asymptotic cost of a sequence of appends goes from linear to quadratic)。简而言之，性能受损了。时间让我们在这种事情上变聪明了。如果你再去看 **IdentityHashMap**，就会发现，你已经不能调整它的底层实现的参数了。"

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

hashCode( )用的就是对象的地址，因此你应该在 **hashCode( )**里面用一些能标识对象的有意义的信息。

用 **String** 举个例。**String** 有个特点，如果程序中的多个内容相同的 **String** 对象会被映射到同一块内存(附录 A 讲的就是这个机制)。所以两个用 **new String("hello")**创建的 **String** 对象，应该返回相同的 **hashCode( )**，这是天经地义的。下面这段程序就是讲这件事的：

```
//: c11:StringHashCode.java
import com.bruceeckel.simpletest.*;

public class StringHashCode {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    System.out.println("Hello".hashCode( ));
    System.out.println("Hello".hashCode( ));
    monitor.expect(new String[] {
      "69609650",
      "69609650"
    });
  }
} ///:~
```

很明显，**String** 是根据其内容计算 **hashCode( )**的。

所以要想让 **hashCode( )**充分发挥作用，它必须既快又有意义；也就是说它必须根据其内容生成值。记住，这个值可以是不唯一的，——在速度和唯一性之间，你应该更倾向于速度——但是经过 **hashCode( )**和 **equals( )**这两道处理，对象的身份应该被完全确认下来。

由于 **hashCode( )**在生成 bucket 数组的下标之前还要进行进一步的处理，因此它的取值范围并不重要；只要是 **int** 就行了。

还有一点：好的 **hashCode( )**应该能生成均匀分布的值。如果这些值都结成了块，那么 **HashMap** 和 **HashSet** 的负载就会被集中在一些特定的区域，相比均匀分布的 hash 函数，其性能自然会打一些折扣。

Joshua Block 在 *Effective Java* 里面(Addison-Wesley，2001 年出版)给怎样写像样的 **hashCode( )**出了个方子：

1. 给 **int** 变量 **result** 赋一个非零的常量，比如 17。
2. 对每个重要的数据成员 **f**，(也就是 **equals( )**要会用到的所有的数据成员)，分别计算其 **int** 型的 hash 值 **c**：

数据字段类型	计算方法
Boolean	**c = (f ? 0 : 1)**

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Byte, char, short, 或 int	**c = (int)f**
Long	**c = (int)(f ^ (f >>>32))**
Float	**c = Float.floatToIntBits(f);**
Double	**long l = Double.doubleToLongBits(f); c = (int)(l ^ (l >>> 32))**
它的 **equals( )**调用了其中的数据字段的 **equals( )**的 **Object**	**c = f.hashCode( )**
数组	对于其中每个元素都使用上述规则

1. 结合上面的 hash 值，计算:
   result = 37 * result + c;
2. 返回 **result**.
3. 再检查一下这个 **hashCode( )**函数，要确保相同的实例会生成相同的 hash 值。

下面就是照这个方抓的药：

```
//: c11:CountedString.java
// Creating a good hashCode( ).
import com.bruceeckel.simpletest.*;
import java.util.*;

public class CountedString {
  private static Test monitor = new Test( );
  private static List created = new ArrayList( );
  private String s;
  private int id = 0;
  public CountedString(String str) {
    s = str;
    created.add(s);
    Iterator it = created.iterator( );
    // Id is the total number of instances
    // of this string in use by CountedString:
    while(it.hasNext( ))
      if(it.next( ).equals(s))
        id++;
  }
  public String toString( ) {
    return "String: " + s + " id: " + id +
      " hashCode( ): " + hashCode( );
  }
  public int hashCode( ) {
    // Very simple approach:
    // return s.hashCode( ) * id;
    // Using Joshua Bloch's recipe:
    int result = 17;
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    result = 37*result + s.hashCode( );
    result = 37*result + id;
    return result;
  }
  public boolean equals(Object o) {
    return (o instanceof CountedString)
      && s.equals(((CountedString)o).s)
      && id == ((CountedString)o).id;
  }
  public static void main(String[] args) {
    Map map = new HashMap( );
    CountedString[] cs = new CountedString[10];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CountedString("hi");
      map.put(cs[i], new Integer(i));
    }
    System.out.println(map);
    for(int i = 0; i < cs.length; i++) {
      System.out.println("Looking up " + cs[i]);
      System.out.println(map.get(cs[i]));
    }
    monitor.expect(new String[] {
      "{String: hi id: 4 hashCode( ): 146450=3," +
      " String: hi id: 10 hashCode( ): 146456=9," +
      " String: hi id: 6 hashCode( ): 146452=5," +
      " String: hi id: 1 hashCode( ): 146447=0," +
      " String: hi id: 9 hashCode( ): 146455=8," +
      " String: hi id: 8 hashCode( ): 146454=7," +
      " String: hi id: 3 hashCode( ): 146449=2," +
      " String: hi id: 5 hashCode( ): 146451=4," +
      " String: hi id: 7 hashCode( ): 146453=6," +
      " String: hi id: 2 hashCode( ): 146448=1}",
      "Looking up String: hi id: 1 hashCode( ):
146447",
      "0",
      "Looking up String: hi id: 2 hashCode( ):
146448",
      "1",
      "Looking up String: hi id: 3 hashCode( ):
146449",
      "2",
      "Looking up String: hi id: 4 hashCode( ):
146450",
      "3",
      "Looking up String: hi id: 5 hashCode( ):
146451",
      "4",
      "Looking up String: hi id: 6 hashCode( ):
146452",
      "5",
      "Looking up String: hi id: 7 hashCode( ):
146453",
      "6",
      "Looking up String: hi id: 8 hashCode( ):
146454",
      "7",
      "Looking up String: hi id: 9 hashCode( ):
146455",
      "8",
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        "Looking up String: hi id: 10 hashCode( ):
146456",
        "9"
      });
    }
} ///:~
```

CountedString 包含一个 **String** 和一个 **id**。这个 **id** 的意思是，有多少 **CountedString** 对象包含了与这个对象相同的 **String**。计数过程是通过“让构造函数去遍历那个保存着所有 **String** 的 **static ArrayList**”来实现的。

hashCode( )和 **equals( )**要用这两个数据来生成返回值；如果你只使用 **String** 或 **id** 的话，不同的对象就会产生相同的值了。

main( )用同一个 **String** 创建了一大堆 **CountedString**。之所以要用相同的 **String**，是想以此证明，由于 **id** 的不同，**hashCode( )**产生了不同的值。我们把 **HashMap** 打印了出来，这样你就能看到它的内部存储顺序了(没什么规律)。接下来，我们要一个一个地查找键，并以此证明它能正常工作。

为类写一个合适的 **hashCode( )**和 **equals( )**，是有一件技巧性很强的事。Apache 的“Jakarta Commons”项目里有很多很实用的工具。就在 *jakata.apache.org/commons* 的“lang”下面(这个项下还包括了一些可能会非常有用的类库，而且它像是 Java 社区对 C++的 *www.boost.org* 作出的应战)。

持有 reference

java.lang.ref 类库里有一套能增进垃圾回收器工作的灵活性的类。一旦碰到了“对象大到要耗光内存”的时候，这些类就会显得格外有用。有三个类是继承抽象类 **Reference** 的：**SoftReference**，**WeakReference** 和 **PhantomReference**。如果待处理的对象**只能**通过这些 **Reference** 进行访问的话，那么这些 **Reference** 对象就会向垃圾回收器提供一些不同级别的暗示。

如果对象**还能访问的到**，那么在程序的某个地方应该还能找到这个对象。或许栈里还有一个普通的 reference 直接指着这个对象，或许在你“引用(reference)”的对象里面还有一个指向那个要找的对象的 reference；这中间可能会有很多层。但是，只要对象还能访问的到，也就是说程序还要用，垃圾回收器就不能回收。如果对象已经访问不到了，程序也就无从使用了，因此回收就应该是安全的了。

你可以用 **Reference** 对象来持有那个你想继续持有的那个对象的 reference；你要能访问那个对象，但是有允许垃圾回收器回收它。于是，你就有了一种“能继续使用那个对象，但是当内存即将耗尽的时候，又能释放那个对象”的方法了。

要达到这个目的，你可以把 **Reference** 对象当作你和『普通的 reference』之间的中介，**此外**那个对象上面还不能附有其它『普通的 reference』(指没有用 **Reference** 类包覆的 reference)。如果垃圾回收器发现你还可以通过『普通的 reference』访问某个对象，那它就不会释放那个对象了。

从 **SoftReference** 到 **WeakReference**，到 **PhantomRefernce**，它们的功能依次减弱，而“访问级别(level of reachability)”又各自不同。**SoftReference** 是为内存敏感的缓存而实现的。**WeakReference** 是为了“规范化映射(canonical mappings)”而实现的，也就是为了节省存储空间，对象的实例可以被同时用于程序的多个地方，这样你就不用重新申请它的键(或值)了。**PhantomReference** 则用于调度“回收前的清理工作(premortem cleanup action)”，这种清理可以比 Java 的 finalization 的机制更为灵活。

对于 **SoftReference** 和 **WeakReference**，你可以选择是不是把它们放进 **ReferenceQueue**(一个用于“回收前的清理工作”的工具)，但是对于 **PhantomReference**，你只能把它放进 **ReferenceQueue**。下面就是一个简单的演示：

```
//: c11:References.java
// Demonstrates Reference objects
import java.lang.ref.*;

class VeryBig {
  private static final int SZ = 10000;
  private double[] d = new double[SZ];
  private String ident;
  public VeryBig(String id) { ident = id; }
  public String toString( ) { return ident; }
  public void finalize( ) {
    System.out.println("Finalizing " + ident);
  }
}

public class References {
  private static ReferenceQueue rq = new
ReferenceQueue( );
  public static void checkQueue( ) {
    Object inq = rq.poll( );
    if(inq != null)
      System.out.println("In queue: " +
        (VeryBig)((Reference)inq).get( ));
  }
  public static void main(String[] args) {
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    int size = 10;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.parseInt(args[0]);
    SoftReference[] sa = new SoftReference[size];
    for(int i = 0; i < sa.length; i++) {
      sa[i] = new SoftReference(
        new VeryBig("Soft " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)sa[i].get( ));
      checkQueue( );
    }
    WeakReference[] wa = new WeakReference[size];
    for(int i = 0; i < wa.length; i++) {
      wa[i] = new WeakReference(
        new VeryBig("Weak " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)wa[i].get( ));
      checkQueue( );
    }
    SoftReference s =
      new SoftReference(new VeryBig("Soft"));
    WeakReference w =
      new WeakReference(new VeryBig("Weak"));
    System.gc( );
    PhantomReference[] pa = new
PhantomReference[size];
    for(int i = 0; i < pa.length; i++) {
      pa[i] = new PhantomReference(
        new VeryBig("Phantom " + i), rq);
      System.out.println("Just created: " +
        (VeryBig)pa[i].get( ));
      checkQueue( );
    }
  }
} ///:~
```

运行这个程序的时候(你应该将用“more”把输出重定向到一个管道里，这样就能看到分页的输出了)，你会发现，尽管你还能通过 **Reference** 对象进行访问(要想获取真实对象的 reference，你得用 **get( )**)，但对象还是被回收了。你还会看到 **ReferenceQueue** 总是会返回保存 **null** 对象的 **Reference** 对象。要利用它，你可以继承某个你感兴趣的 **Reference** 类，并且给新的 **Reference** 类型加上一些有用的方法。

WeakHashMap

容器类库里面还有一种特殊的，持有 weak reference 的 **Map**：**WeakHashMap**。这个类是为“简化创建规范化映射”而设计的。在这种映射中，你可以用“只创建某个值的一个实例”来节省存储空间。当程序需要使用这个值的时候，它会在映射表中查找已有的对象(而不是从头开始创建)，并且使用那个对象。可以在对映射进行初始化的时候就把值设置好，但是常见的还是按需来设置。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

由于这是一种节省内存空间的技术，所以让垃圾回收器来自动清理 **WeakHashMap** 的键和值是很容易的。你不必对放进 **WeakHashMap** 的键和值作什么特殊处理；它们会由 map 自动包装成 **WeakReference**。清理的触发条件是，不需要再使用这个键了，就像这样：

```
//: c11:CanonicalMapping.java
// Demonstrates WeakHashMap.
import java.util.*;
import java.lang.ref.*;

class Key {
  private String ident;
  public Key(String id) { ident = id; }
  public String toString( ) { return ident; }
  public int hashCode( ) { return
ident.hashCode( ); }
  public boolean equals(Object r) {
    return (r instanceof Key)
      && ident.equals(((Key)r).ident);
  }
  public void finalize( ) {
    System.out.println("Finalizing Key "+ ident);
  }
}

class Value {
  private String ident;
  public Value(String id) { ident = id; }
  public String toString( ) { return ident; }
  public void finalize( ) {
    System.out.println("Finalizing Value " + ident);
  }
}

public class CanonicalMapping {
  public static void main(String[] args) {
    int size = 1000;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.parseInt(args[0]);
    Key[] keys = new Key[size];
    WeakHashMap map = new WeakHashMap( );
    for(int i = 0; i < size; i++) {
      Key k = new Key(Integer.toString(i));
      Value v = new Value(Integer.toString(i));
      if(i % 3 == 0)
        keys[i] = k; // Save as "real" references
      map.put(k, v);
    }
    System.gc( );
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

正如前面所讲的，由于要用于 hash 数据结构，**Key** 必须有 **hashCode( )**和 **equals( )**方法。

运行程序的时候，你会看到垃圾回收器会每隔三个跳开一个，这是因为那个键的 reference 已经被放进 **keys** 数组，因此那个对象是不能回收的。

重访 Iterator

现在我们可以来看 **Iterator** 的真正威力了：把序列的遍历过程同这个序列的底层结构分隔开来。**PrintData**(本章的前面定义的)用 **Iterator** 遍历了一个序列，并且对每个对象都使用 **toString( )**方法。下面这段程序创建了两个容器——**ArrayList** 和 **HashMap**，然后分别用 **Mouse** 和 **Hamster** 进行填充。(这两个类是在本章的前面部分定义的。)由于 **Iterator** 隐藏了在其背后的那个容器的结构，因此 **Printer.printAll( )**既不知道也不关心它到底是哪个容器的 **Iterator**：

```
//: c11:Iterators2.java
// Revisiting Iterators.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class Iterators2 {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    List list = new ArrayList( );
    for(int i = 0; i < 5; i++)
      list.add(new Mouse(i));
    Map m = new HashMap( );
    for(int i = 0; i < 5; i++)
      m.put(new Integer(i), new Hamster(i));
    System.out.println("List");
    Printer.printAll(list.iterator( ));
    System.out.println("Map");
    Printer.printAll(m.entrySet( ).iterator( ));
    monitor.expect(new String[] {
      "List",
      "This is Mouse #0",
      "This is Mouse #1",
      "This is Mouse #2",
      "This is Mouse #3",
      "This is Mouse #4",
      "Map",
      "4=This is Hamster #4",
      "3=This is Hamster #3",
      "2=This is Hamster #2",
      "1=This is Hamster #1",
      "0=This is Hamster #0"
    }, Test.IGNORE_ORDER);
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

HashMap 的 **entrySet( )**方法会返回 **Map.entry** 对象的 **Set**，这个对象既包括键也包括值，因此你会看到这两者都被打印出来了。

注意 **PrintData.print( )**用到了下面这条有利条件，“容器里的对象都是 **Object**，所以 **System.out.println( )**会自动调用它的 **toString( )**方法”。但是在解决实际问题的时候，你很可能会面对“**Iterator** 所访问的是某种类型的容器”的情况。比方说，容器里的对象都是能 **draw( )**出来的 **Shape**。于是你还得先把 **Iterator.next( )** 所返回的 **Object** 下传给 **Shape**。

选择实现

现在你应该明白了，实际上只有三种容器组件：**Map**，**List** 和 **Set**，但是每种组件又有多个实现。所以，如果你要用到某个组件的话，又该选择其中的哪个实现呢？

要想回答这个问题，你必须先了解，各种实现都有它自己的特性，它所独有的强项和弱点。比方说，你可以从图表得知，**Hashtable**，**Vector** 和 **Stack** 属于老版本留下来的类，目的是让老代码还能运行下去。所以，写新程序的时候就不应该再用了。

容器与容器的差别，归根结蒂还是在其“背后”的实现； 也就是说，真正实现这个 **interface** 的数据结构是什么。比方说，**ArrayList** 和 **LinkedList** 都实现了 **List** 接口，所以不论你用它们中的哪个，其基本的功能都是一样的。但是，**ArrayList** 的背后是数组，而 **LinkedList** 是用所谓的双向链表来实现的，也就是每个对象，除了保存数据之外，还保存着在它前面和后面的那两个对象的 reference。所以，如果你要在 **List** 的中间做很多插入和删除的话，**LinkedList** 就比较合适了。(**LinkedList** 也还有一些 **AbstractSequenialList** 的附加的功能。) 否则，**ArrayList** 会更快一些。

再举一个例子，**Set** 有三个实现，**TreeSet**，**HashSet** 和 **LinkedHashSet**。它们的工作方式都不一样：**HashSet** 是我们通常所用的 **Set**，我们会把用当“查询性能的基准(raw speed on lookup)”，**LinkedHashSet** 会按插入顺序保存 pair，而 **TreeSet** 的背后是 **TreeMap**，因此它能提供恒定有序的 **Set**。之所以要这么设计，是想让你能根据需要选用具体的实现。绝大多数情况下，**HashSet** 就够用了，所以缺省情况下，你应该用它。

如何挑选 List

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

要想观察 **List** 的各种实现之间的区别，最具说服力的办法还是做一个性能测试。下面这段程序创建了一个被用作测试框架的内部类基类，然后创建了一个代表各种测试的匿名内部类的数组。你可以调用这些内部类的 **test( )**方法来启动测试，这样就能很方便地添加和删除新的测试了。

```
//: c11:ListPerformance.java
// Demonstrates performance differences in Lists.
// {Args: 500}
import java.util.*;
import com.bruceeckel.util.*;

public class ListPerformance {
  private static int reps = 10000;
  private static int quantity = reps / 10;
  private abstract static class Tester {
    private String name;
    Tester(String name) { this.name = name; }
    abstract void test(List a);
  }
  private static Tester[] tests = {
    new Tester("get") {
      void test(List a) {
        for(int i = 0; i < reps; i++) {
          for(int j = 0; j < quantity; j++)
            a.get(j);
        }
      }
    },
    new Tester("iteration") {
      void test(List a) {
        for(int i = 0; i < reps; i++) {
          Iterator it = a.iterator( );
          while(it.hasNext( ))
            it.next( );
        }
      }
    },
    new Tester("insert") {
      void test(List a) {
        int half = a.size( )/2;
        String s = "test";
        ListIterator it = a.listIterator(half);
        for(int i = 0; i < reps * 10; i++)
          it.add(s);
      }
    },
    new Tester("remove") {
      void test(List a) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext( )) {
          it.next( );
          it.remove( );
        }
      }
    },
  };
  public static void test(List a) {
    // Strip qualifiers from class name:
    System.out.println("Testing " +
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      a.getClass( ).getName( ).replaceAll("\\w+\\.",
""));
    for(int i = 0; i < tests.length; i++) {
      Collections2.fill(a,
Collections2.countries.reset( ),
        quantity);
      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 testArrayAsList(int reps) {
    System.out.println("Testing array as List");
    // Can only do first two tests on an array:
    for(int i = 0; i < 2; i++) {
      String[] sa = new String[quantity];
      Arrays2.fill(sa,
Collections2.countries.reset( ));
      List a = Arrays.asList(sa);
      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) {
    // Choose a different number of
    // repetitions via the command line:
    if(args.length > 0)
      reps = Integer.parseInt(args[0]);
    System.out.println(reps + " repetitions");
    testArrayAsList(reps);
    test(new ArrayList( ));
    test(new LinkedList( ));
    test(new Vector( ));
  }
} ///:~
```

考虑到内部类 **Tester** 是各项具体测试的基类，因此它被定义成 **abstract** 的。它包含了一个“要在测试开始的时候被打印出来”的 **String**，以及真正用于测试的 **abstract** 的 **test( )**方法。所有测试都被集中在 **tests** 数组里面。我们用继承 **Tester** 的匿名内部类来对数组进行初始化。要想增加或删除测试，直接往数组里加减内部类就可以了，接下来的东西都是自动的。

为了在数组和容器之间进行比较(主要是针对 **ArrayList** 的)，我们用 **Arrays.asList( )**把数组包装成 **List**，并以此为数组创建了一个特殊的测试。注意，由于你没法在数组里面插入和删除元素，因此你只能进行前两个课目的测试。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

我们先把 **List** 填满，再传给 **test( )**，然后程序再为各项测试标注时间。测试结果会随机器的不同而不同；因此我们只关心它们在顺序，以及大数方面的差异。下面是某次运行的结果：

Type	Get	Iteration	Insert	Remove
array	172	516	na	na
ArrayList	281	1375	328	30484
LinkedList	5828	1047	109	16
Vector	422	1890	360	30781

结果同我们的估计差不多，数组在随机访问和顺序访问方面比任何容器都快。**ArrayList** 的随机访问(**get( )**)要比 **LinkedList** 快。(但奇怪的是 **LinkedList** 的顺序访问居然会比 **ArrayList** 的快，真是有点不可思议。)另一方面，**LinkedList** 的插入和删除，**特别**是删除，要比 **ArrayList** 的快得多的多。通常情况下，**Vector** 的速度比不上 **ArrayList** 的，所以你就不要再用了；它之所以还呆在类库里面，只是为了要对遗留下来的老代码提供支持(这里还能测试 **Vector**，也只是因为它在 Java 2 里摇身一变为 **List** 了)。也许最佳的做法就是，先选用 **ArrayList**，当发现“在列表的中间进行插入和删除的操作太多所引发的”性能问题时，把它改成 **LinkedList**。当然，处理固定数量的元素时，还是用数组。

如何挑选 Set　　(Choosing between Sets)

你可以根据需要，在 **TreeSet**，**HashSet** 和 **LinkedHashSet** 中间选一个择。下面这项测试揭示了这几种实现在性能方面的侧重点：

```
//: c11:SetPerformance.java
// {Args: 500}
import java.util.*;
import com.bruceeckel.util.*;

public class SetPerformance {
  private static int reps = 50000;
  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( );
          Collections2.fill(s,
            Collections2.countries.reset( ),size);
        }
      }
    },
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      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) {
      // Strip qualifiers from class name:
      System.out.println("Testing " +
        s.getClass( ).getName( ).replaceAll("\\w+\\.",
"") +
        " size " + size);
      Collections2.fill(s,
        Collections2.countries.reset( ), 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) {
      // Choose a different number of
      // repetitions via the command line:
      if(args.length > 0)
        reps = Integer.parseInt(args[0]);
      System.out.println(reps + " repetitions");
      // Small:
      test(new TreeSet( ), 10);
      test(new HashSet( ), 10);
      test(new LinkedHashSet( ), 10);
      // Medium:
      test(new TreeSet( ), 100);
      test(new HashSet( ), 100);
      test(new LinkedHashSet( ), 100);
      // Large:
      test(new TreeSet( ), 1000);
      test(new HashSet( ), 1000);
      test(new LinkedHashSet( ), 1000);
    }
} ///:~
```

下面的表格演示了运行的结果。(当然，它会随你所使用的计算机和 JVM 的不同而不同；你也应该自己运行一下):

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

Type	Test size	Add	Contains	Iteration
	10	25.0	23.4	39.1
TreeSet	100	17.2	27.5	45.9
	1000	26.0	30.2	9.0
	10	18.7	17.2	64.1
HashSet	100	17.2	19.1	65.2
	1000	8.8	16.6	12.8
	10	20.3	18.7	64.1
LinkedHashSet	100	18.6	19.5	49.2
	1000	10.0	16.3	10.0

总的说来，**HashSet** 的各项性能都比 **TreeSet** 的好 (尤其是在“加入”和“查询”这两个最重要的方面)。而 **TreeSet** 的意义在于，它会按顺序保存元素，因此只有在需要有序的 **Set** 时，你才应该用它。

注意 **LinkedHashSet** 的插入比 **HashSet** 的稍慢一些。这是因为它要承担维护链表和 hash 容器的双重代价。但是由于链接表的缘故，**LinkedHashSet** 的遍历比较快。

如何挑选 Maps

对 **Map** 来说，容量是影响其性能的最重要的因素，下面我们用这个程序测试一下它对性能的影响：

```
//: c11:MapPerformance.java
// Demonstrates performance differences in Maps.
// {Args: 500}
import java.util.*;
import com.bruceeckel.util.*;

public class MapPerformance {
  private static int reps = 50000;
  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( );
          Collections2.fill(m,
            Collections2.geography.reset( ), size);
        }
      }
    },
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      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.entrySet( ).iterator( );
            while(it.hasNext( ))
              it.next( );
          }
        }
      },
    };
    public static void test(Map m, int size) {
      // Strip qualifiers from class name:
      System.out.println("Testing " +
        m.getClass( ).getName( ).replaceAll("\\w+\\.",
"") +
        " size " + size);
      Collections2.fill(m,
        Collections2.geography.reset( ), 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) {
      // Choose a different number of
      // repetitions via the command line:
      if(args.length > 0)
        reps = Integer.parseInt(args[0]);
      System.out.println(reps + " repetitions");
      // Small:
      test(new TreeMap( ), 10);
      test(new HashMap( ), 10);
      test(new LinkedHashMap( ), 10);
      test(new IdentityHashMap( ), 10);
      test(new WeakHashMap( ), 10);
      test(new Hashtable( ), 10);
      // Medium:
      test(new TreeMap( ), 100);
      test(new HashMap( ), 100);
      test(new LinkedHashMap( ), 100);
      test(new IdentityHashMap( ), 100);
      test(new WeakHashMap( ), 100);
      test(new Hashtable( ), 100);
      // Large:
      test(new TreeMap( ), 1000);
      test(new HashMap( ), 1000);
      test(new LinkedHashMap( ), 1000);
      test(new IdentityHashMap( ), 1000);
      test(new WeakHashMap( ), 1000);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
      test(new Hashtable( ), 1000);
    }
} ///:~
```

由于要考虑 **Map** 的大小对性能的影响，因此我们用除以容量的方法对测试数据做了一些修正。下面是某次测试的运行结果。(你的测试结构可能会不同。)

Type	Test size	Put	Get	Iteration
	10	26.6	20.3	43.7
TreeMap	100	34.1	27.2	45.8
	1000	27.8	29.3	8.8
	10	21.9	18.8	60.9
HashMap	100	21.9	18.6	63.3
	1000	11.5	18.8	12.3
	10	23.4	18.8	59.4
LinkedHashMap	100	24.2	19.5	47.8
	1000	12.3	19.0	9.2
	10	20.3	25.0	71.9
IdentityHashMap	100	19.7	25.9	56.7
	1000	13.1	24.3	10.9
	10	26.6	18.8	76.5
WeakHashMap	100	26.1	21.6	64.4
	1000	14.7	19.2	12.4
	10	18.8	18.7	65.7
Hashtable	100	19.4	20.9	55.3
	1000	13.1	19.9	10.8

正如你所预料的，**Hashtable** 的性能同 **HashMap** 的不相上下。(可能你也注意到了，一般情况下 **HashMap** 会稍快些；**HashMap** 是用来代替 **Hashtable** 的。)**TreeMap** 通常要比 **HashMap** 慢，那么为什么还要它呢？答案是，它是用来创建有序列表的。树总是有序的，所以根本用不着为它去做排序。往 **TreeMap** 里面填完数据之后，你就能用 **keySet( )**获取包含这个 **Map** 的键的 **Set** 了，接下来用 **toArray()**把这个 **Set** 转换成数组。然后就能用 **static** 的 **Arrays.binarySearch( )**方法(下面再讲)在有序数组里面进行快速查找对象了。当然，这一切是在无法使用 **HashMap** 的情况下做的，因为 **HashMap** 就是为快速查找而设计的。此外，你还能轻而易举地用

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

TreeMap 创建一个 **HashMap**。结论是，选择 **Map** 的时候，首选应该是 **HashMap**，只有在要用恒定有序的 **Map** 的情况下，你才应该选用 **TreeMap**。

LinkedHashMap 比 **HashMap** 稍慢一些，这是因为它除了要保存 hash 数据结构之外，它还要保存链表。**IdentityHashMap** 和上面没法作比较，因为它是用 **==** 而不是 **equals( )**来比较对象的相等性的。

List 的排序与查询

用于 **List** 的排序和查询工具和用于数组的有着相同的名称和特征签名，只是它们不是 **Arrays** 的，而是 **Collections** 的 **static** 方法。下面我们就用这些方法来改写 **ArraySearching.java**：

```
//: c11:ListSortSearch.java
// Sorting and searching Lists with 'Collections.'
import com.bruceeckel.util.*;
import java.util.*;

public class ListSortSearch {
  public static void main(String[] args) {
    List list = new ArrayList( );
    Collections2.fill(list, Collections2.capitals,
25);
    System.out.println(list + "\n");
    Collections.shuffle(list);
    System.out.println("After shuffling: " + list);
    Collections.sort(list);
    System.out.println(list + "\n");
    Object key = list.get(12);
    int index = Collections.binarySearch(list, key);
    System.out.println("Location of " + key +
      " is " + index + ", list.get(" +
      index + ") = " + list.get(index));
    AlphabeticComparator comp = new
AlphabeticComparator( );
    Collections.sort(list, comp);
    System.out.println(list + "\n");
    key = list.get(12);
    index = Collections.binarySearch(list, key,
comp);
    System.out.println("Location of " + key +
      " is " + index + ", list.get(" +
      index + ") = " + list.get(index));
  }
} ///:~
```

这些方法的用法同 **Arrays** 的完全相同，只不过是用于 **List** 而不是数组。同数组一样，如果你用 **Comparator** 进行排序，那么你得用一个 **Comparator** 来 **binarySearch( )**。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

这个程序还演示了 **Collections** 的 **shuffle( )**方法，它的功能就是把 **List** 的顺序打乱。

实用工具

Collections 类还有很多很实用的工具:

max(Collection) **min(Collection)**	用自然对象内置的算法进行比较，返回 **Collection** 中最大和最小的元素。
max(Collection, Comparator) **min(Collection, Comparator)**	用 **Comparator** 进行比较，返回最大或最小的元素。
indexOfSubList(List source, List target)	获取 **target** 第一次出现在 **source** 中的位置。
lastIndexOfSubList(List source, List target)	返回 **target** 最后一次出现在 **source** 中的位置。
replaceAll(List list, Object oldVal, Object newVal)	将所有的 **oldVal** 替换成 **newVal**.
reverse( )	颠倒 **List** 的顺序。
rotate(List list, int distance)	把所有的元素向后移 **distance** 位，将最后面的元素接到最前面。
copy(List dest, List src)	将 **src** 的元素拷贝到 **dest**。
swap(List list, int i, int j)	互换 **list** 的 **i** 和 **j** 位置上的元素。可能会比你写代码要快。
fill(List list, Object o)	把 **list** 里面的全部元素全都替换成 **o**。
nCopies(int n, Object o)	返回一个有 **n** 个元素的不可变的 **List**，而且这个 **List** 中的所有元素全都指向 **o**。
enumeration(Collection)	返回一个老式的 **Enumeration**。
list(Enumeration e)	用这个 **Enumeration** 生成一个 **ArrayList**，并且返回这个 **ArrayList**。是用来处理遗留下来的老代码的。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

注意，**min( )**和 **max( )**是针对 **Collection** 的，它不是只为 **List** 服务的，所以你不用担 **Collection** 是不是有序。(我们前面已经提到过了，在 **binarySearch( )**之前，你一定得先 **sort( )**这个 **List** 或数组。)

```
//: c11:Utilities.java
// Simple demonstrations of the Collections
utilities.
import com.bruceeckel.simpletest.*;
import java.util.*;
import com.bruceeckel.util.*;

public class Utilities {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    List list = Arrays.asList(
      "one Two three Four five six one".split(" "));
    System.out.println(list);
    System.out.println("max: " +
Collections.max(list));
    System.out.println("min: " +
Collections.min(list));
    AlphabeticComparator comp = new
AlphabeticComparator( );
    System.out.println("max w/ comparator: " +
      Collections.max(list, comp));
    System.out.println("min w/ comparator: " +
      Collections.min(list, comp));
    List sublist =
      Arrays.asList("Four five six".split(" "));
    System.out.println("indexOfSubList: " +
      Collections.indexOfSubList(list, sublist));
    System.out.println("lastIndexOfSubList: " +
      Collections.lastIndexOfSubList(list, sublist));
    Collections.replaceAll(list, "one", "Yo");
    System.out.println("replaceAll: " + list);
    Collections.reverse(list);
    System.out.println("reverse: " + list);
    Collections.rotate(list, 3);
    System.out.println("rotate: " + list);
    List source =
      Arrays.asList("in the matrix".split(" "));
    Collections.copy(list, source);
    System.out.println("copy: " + list);
    Collections.swap(list, 0, list.size( ) - 1);
    System.out.println("swap: " + list);
    Collections.fill(list, "pop");
    System.out.println("fill: " + list);
    List dups = Collections.nCopies(3, "snap");
    System.out.println("dups: " + dups);
    // Getting an old-style Enumeration:
    Enumeration e = Collections.enumeration(dups);
    Vector v = new Vector( );
    while(e.hasMoreElements( ))
      v.addElement(e.nextElement( ));
    // Converting an old-style Vector
    // to a List via an Enumeration:
    ArrayList arrayList =
Collections.list(v.elements( ));
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    System.out.println("arrayList: " + arrayList);
    monitor.expect(new String[] {
      "[one, Two, three, Four, five, six, one]",
      "max: three",
      "min: Four",
      "max w/ comparator: Two",
      "min w/ comparator: five",
      "indexOfSubList: 3",
      "lastIndexOfSubList: 3",
      "replaceAll: [Yo, Two, three, Four, five, six,
Yo]",
      "reverse: [Yo, six, five, Four, three, Two,
Yo]",
      "rotate: [three, Two, Yo, Yo, six, five,
Four]",
      "copy: [in, the, matrix, Yo, six, five, Four]",
      "swap: [Four, the, matrix, Yo, six, five, in]",
      "fill: [pop, pop, pop, pop, pop, pop, pop]",
      "dups: [snap, snap, snap]",
      "arrayList: [snap, snap, snap]"
    });
  }
} ///:~
```

程序的输出已经讲解了这些工具的用法。注意一下 **AlphabeticComparator** 对 **min( )** 和 **max( )**的影响，这是由于大小写的缘故。

把 Collection 和 Map 设成不可修改的

通常情况下，创建只读的 **Collection** 或 **Map** 是很方便的。**Collections** 有专门的方法，你可以传一个容器给它，它会返回这个容器的只读版。这个方法有四种变形，**Collection**(如果你没法明确指明它是哪种 **Collection** 的话)，**List**，**Set** 和 **Map** 各一个。下面我们用一个例子来演示一下如何创建只读的容器：

```
//: c11:ReadOnly.java
// Using the Collections.unmodifiable methods.
import java.util.*;
import com.bruceeckel.util.*;

public class ReadOnly {
  private static Collections2.StringGenerator gen =
    Collections2.countries;
  public static void main(String[] args) {
    Collection c = new ArrayList( );
    Collections2.fill(c, gen, 25); // Insert data
    c = Collections.unmodifiableCollection(c);
    System.out.println(c); // Reading is OK
    //! c.add("one"); // Can't change it

    List a = new ArrayList( );
    Collections2.fill(a, gen.reset( ), 25);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    a = Collections.unmodifiableList(a);
    ListIterator lit = a.listIterator( );
    System.out.println(lit.next( )); // Reading is
OK
    //! lit.add("one"); // Can't change it

    Set s = new HashSet( );
    Collections2.fill(s, gen.reset( ), 25);
    s = Collections.unmodifiableSet(s);
    System.out.println(s); // Reading is OK
    //! s.add("one"); // Can't change it

    Map m = new HashMap( );
    Collections2.fill(m, Collections2.geography, 25);
    m = Collections.unmodifiableMap(m);
    System.out.println(m); // Reading is OK
    //! m.put("Ralph", "Howdy!");
  }
} ///:~
```

编译器不会因为你调用了“unmodifiable”方法而去做额外的检查，但是转换完毕之后，你再想去修改这个容器的时候，它就会抛出 **UnsupportedOperationException** 了。

任何情况下，你都应该**先**准备好数据，**再**把容器设成只读的。做完之后，你就应该用“unmodifiable”方法所返回的 reference 来替换原先那个 reference 了。这样你就不会在无意之中把不该改的东西给改了。此外，你还能利用这个方法，在类里以 **private** 的权限保存可修改的容器。这样，你可以修改，而其他人就只能读了。

Collection 和 Map 的同步

synchronized 关键词是多线程的一个重要组成部分，我们要到第 13 章才做更详细的讲解。这里，我只想指出，**Collections** 里面也有一个能自动对容器做同步的方法。它的语法与“unmodifiable”方法的有些相似：

```
//: c11:Synchronization.java
// Using the Collections.synchronized methods.
import java.util.*;

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( ));
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
  }
} ///:~
```

这样，你就能通过合适的“synchronized”方法来传递容器了；于是，你就再也不用担心会泄漏尚未同步的东西了。

Fail fast

Java 容器还有一种能防止多个进程同时修改容器内容的机制。假设你正在遍历某个容器，这时另一个进程插了进来，对容器作了插入，删除或是修改里边的对象，于是这个问题就来了。或许你已经把对象传出去了，但是它抢在你头里把它给删了；或许你调用了 **size( )**，但是容器已经缩水了——会引发灾难的可能性太多了。Java 容器类库集成了一个叫 *fail-fast*(及早报告错误)机制，它能找出所有不应由进程负责的容器的变化。如果它发现有人在修改容器，它会立即返回一个 **ConcurrentModificationException**。这就是它“fail-fast”的地方，它不会等出了问题之后再去用很复杂的算法去找问题了。

要想观察 fail-fast 很容易——只要创建一个迭代器，然后在 iterator 的位置上往 **Collection** 里面加东西就行了，就像这样：

```
//: c11:FailFast.java
// Demonstrates the "fail fast" behavior.
// {ThrowsException}
import java.util.*;

public class FailFast {
  public static void main(String[] args) {
    Collection c = new ArrayList( );
    Iterator it = c.iterator( );
    c.add("An object");
    // Causes an exception:
    String s = (String)it.next( );
  }
} ///:~
```

之所以会有这种异常，是因为你是在已经获取容器的 iterator 的情况下往里面加对象的。程序的两个部分会修改同一个容器的这种可能性，会导致程序处于不确定的状态，因此它抛出一个异常来通知你，你应该修改代码了——碰到这种情况，你应该先往容器里面加元素，再去获取容器的 iterator。

提示一下，如果你是在用 **get( )**访问 **List**，那么 fail-fast 就帮不上什么忙了。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

可以不支持的操作

可以用 **Arrays.asList( )**方法把数组改造成 **List**：

```
//: c11:Unsupported.java
// Sometimes methods defined in the
// Collection interfaces don't work!
// {ThrowsException}
import java.util.*;

public class Unsupported {
  static List a = Arrays.asList(
    "one two three four five six seven
eight".split(" "));
  static List a2 = a.subList(3, 6);
  public static void main(String[] args) {
    System.out.println(a);
    System.out.println(a2);
    System.out.println("a.contains(" + a.get(0) + ")
= " +
      a.contains(a.get(0)));
    System.out.println("a.containsAll(a2) = " +
      a.containsAll(a2));
    System.out.println("a.isEmpty( ) = " +
a.isEmpty( ));
    System.out.println("a.indexOf(" + a.get(5) + ")
= " +
      a.indexOf(a.get(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");
    System.out.println(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(a.get(0)); // Unsupported
    a.removeAll(a2); // Unsupported
  }
} ///:~
```

你会发现，实际上它只是部分地实现了 **Collection** 和 **List** 接口。调用其它方法会引发一个 **UnsupportedOperationException** 异常。**Collection** 接口——以及 Java 容器类库的其它接口——都包含了一些“可选的”方法，也就是说在 **implements** 这个 **interface** 的实体类

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

里可以“支持”也可以不“支持”这些方法。如果调用了不支持的方法，它就会用 **UnsupportedOperationException** 来表示错误。

“什么?!?”，你一定会觉得真是不可思议。“**interface** 和基类的意义就在于，它们能确保这些方法会做一些有意义的事情！而这一点打破了这一条；它的意思是，这些方法不但不能作有意义的事情，而且还会让程序停下来！类型安全则被撇在一边了！”

还不至于那么糟糕。就是对 **Collection**，**List**，**Set** 或 **Map**，编译器也只允许你去调用它们的 **interface** 的方法，所以这一点同 Smalltalk 不太一样(它允许调用任何对象的任何方法，因此只有在运行时才能知道这个调用是不是管用)。此外，绝大多数拿 **Collection** 当参数的方法只是要从里面读东西——而 **Collection** 的“read”不属于可选的。

这个方法能防止接口数量的爆炸。其它容器类都是用一个接口去描述一种容器的变形，最后总是弄得接口多得不得了，于是变得很难学。但是要让 **interface** 去概括所有的具体实现又是不太可能的，于是总会有人去发明新的 **interface**。Java 用“不支持的操作”达成了一项重要目的：使容器类变得易学易用；不支持的操作属于特例，可以以后再学。但是要让这种设计能起作用：

1. **UnsupportedOperationException** 只能偶尔为之。也就是说，绝大多数类应该具备所有功能，只有在特殊情况下，才可以不提供某些功能。对于 Java 容器类库就属于这种情况因为百分之九十九的情况下，你要用到的那些类—**ArrayList**，**LinkedList**，**HashSet** 和 **HashMap** 以及其它实现，都是全功能的。如果你想不去定义 **Collection interface** 里的全部方法就创建一个新的 **Collection**，同时又要让它融入现有的类库里面，那么这种设计确实能为你提供了一扇“后门”就能让它融入现有的类库。
2. 如果这是一种不受支持的操作，那么 **UnsupportedOperationException** 最好是出现在实现的时候，而不是产品交给客户之后。毕竟这是一个编程错误：你用错了实现。这一点不是那么经得起推敲，因此是它实验性的一面。只有时间才能告诉我们它的效果怎么样。

在上述例程中，**Arrays.asList( )**返回的是一个由固定容量的数组支撑的 **List**。因此，你就明白了，为什么它支持的都是那些不改变数组容量的操作了。但是换一个角度看，如果要用新的 **interface** 来表述这种特殊行为的话(或许可以称为“**FixedSizeList**”)，复杂性就有机可乘了，这样用不了多久，再用到这个类库的时候，你就不知道从何开始了。

注意，如果你要想创建普通容器，随时都可以把 **Arrays.asList( )**的结果当作构造函数的参数传给 **List** 或 **Set**，这样就能使用它的完整接口了。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

在为“用 **Collection**，**List**，**Set** 或 **Map** 作参数的方法”写文档的时候，必须指明，一定要实现那些可选的方法。比如排序会用到 **set( )**和 **Iterator.set( )**方法，但是不会用到 **add( )**和 **remove( )**。

Java 1.0/1.1 容器

不幸的是很多代码都是用 Java 1.0/1.1 的容器写的，有时甚至写新代码的时候也用到了这些库。所以尽管你可以不用老容器去写新代码，但是还是应该对它们有点了解。不过旧容器是相当简陋 的，所以也没有更多可说的了。(因为都过去了，所以我也不想再去强调它的设计有多糟糕。)

Vector 和 Enumeration

Java 1.0/1.1 里面，唯一可以自动扩展的容器就只有 **Vector** 了，所以用得最多的也是它。它的缺点多到这里都没法讲的地步(*www.BruceEckel.com* 有本书的第一版可供下载)。基本上你可以把它理解成是一个 **ArrayList**，只是它的方法的名字都很长，很滑稽。Java 2 对 **Vector** 作了些修改，把它归到 **Collection** 和 **List** 里面，所以 **Collections2.fill( )**方法也可以用于下面这些程序。这样做不是太合适，因为会让人误认为 **Vector** 可能更好，实际上这么做只是为了支持 Java 2 以前的代码。

Java 1.0/1.1 用一个新发明的“enumeration”来表示大家都已经很熟悉的 iterator。**Enumeration** 接口比 **Iterator** 的小，它只有两个名字很长方法。一个是“只要 enumeration 还有其它的元素，就会返回 **true**”的 **boolean hasMoreElements( )**，另一个是“只要 enumeration 里面还有下一个元素，它就会返回这个元素”的 **Object nextElement( )**(反之则抛出异常)。

Enumeration 是接口而不是实现，所以有时新的类库仍然会用旧的 **Enumeration**，这真是太糟了，但是也没什么关系。虽然你应该尽量在新代码里使用 **Iterator**，但也要对“类库可能会交给你一个 **Enumeration**”有所准备。

此外，你还可以用 **Collections.enumeration( )**方法从 **Collection** 那里获取一个 **Enumeration**，就像下面这段程序：

```
//: c11:Enumerations.java
// Java 1.0/1.1 Vector and Enumeration.
import java.util.*;
import com.bruceeckel.util.*;

public class Enumerations {
  public static void main(String[] args) {
    Vector v = new Vector( );
    Collections2.fill(v, Collections2.countries,
100);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    Enumeration e = v.elements( );
    while(e.hasMoreElements( ))
      System.out.println(e.nextElement( ));
    // Produce an Enumeration from a Collection:
    e = Collections.enumeration(new ArrayList( ));
  }
} ///:~
```

Java 1.0/1.1 的 **Vector** 只有一个 **addElement( )**方法，但是 **fill( )** 用的却是 **add( )**。这是 **Vector** 在转换成 **List** 的过程中带过去的。调用 **elements( )**会返回一个 **Enumeration**，然后用它来进行遍历。

最后一行创建了一个 **ArrayList**，然后用 **enumeration( )**把 **ArrayList** 的 **Iterator** 转化成 **Enumeration**。 这样，即便你有“要用 **Enumeration** 的旧代码”，仍然可以使用新的容器。

Hashtable

正如你在性能比较里面所看到的，**Hashtable** 和 **HashMap** 基本相同，甚至是在方法的名字上。没理由再在新代码里用 **Hashtable**，而不是 **HashMap** 了。

Stack

我们在讲 **LinkedList** 的时候已经讲过栈了。但是 Java 1.0/1.1 的 **Stack** 有一个非常奇怪的地方，那就是它不是把 **Vector** 用作栈的内部，而是**继承**了 **Vector**。由此 **Stack** 拥有了 **Vector** 的全部特征和功能，并且再加上一点 **Stack** 的功能。真不知道设计者们是很清醒地认为这是一种特殊设计，或者只是一种幼稚的设计；但不管怎么说，很明显，在发布之前，这个方案没有被认真地检讨过。造成的恶果就是，直到现在它还吊在那里(但是你永远也别去用它)。

下面我们简单地演示一下 **Stack**，它会逐行把 **String** 数组的内容压入栈：

```
//: c11:Stacks.java
// Demonstration of Stack Class.
import com.bruceeckel.simpletest.*;
import java.util.*;
import c08.Month;

public class Stacks {
  private static Test monitor = new Test( );
  public static void main(String[] args) {
    Stack stack = new Stack( );
    for(int i = 0; i < Month.month.length; i++)
      stack.push(Month.month[i] + " ");
    System.out.println("stack = " + stack);
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
        // Treating a stack as a Vector:
        stack.addElement("The last line");
        System.out.println("element 5 = " +
          stack.elementAt(5));
        System.out.println("popping elements:");
        while(!stack.empty( ))
          System.out.println(stack.pop( ));
        monitor.expect(new String[] {
          "stack = [January , February , March , April ,
May "+
            ", June , July , August , September ,
October , " +
            "November , December ]",
          "element 5 = June ",
          "popping elements:",
          "The last line",
          "December ",
          "November ",
          "October ",
          "September ",
          "August ",
          "July ",
          "June ",
          "May ",
          "April ",
          "March ",
          "February ",
          "January "
        });
      }
    } ///:~
```

months 数组里的每一行都会被 **push( )**进 **Stack**，然后再 **pop( )**出来。要指出一点，**Stack** 拥有 **Vector** 的全部功能。这是完全可能的，因为继承的意思是，**Stack 就是 Vector**。所以 **Vector** 都做的事 **Stack** 都能做，就像 **elementAt( )**。

正如前面所说的，需要栈的时候，你应该使用 **LinkedList**。

BitSet

如果你想高效地存储许多“是非(on-off)”信息的话，可以使用 **BitSet**。高效只是对容量说的；如果指访问速度的话，primitive 的数组会比较快。

此外，**BitSet** 的最小容量跟 **long** 一样：64 位。也就是说，如果你要存储更小的数据，比如说 8 位的，**BitSet** 就会很浪费；所以如果容量是个问题，你最好还是创建一个自己的类，或者用数组来存储标志信息。

普通的容器会随元素的添加而扩展，这点 **BitSet** 作得很好。下面这段程序演示了 **BitSet** 是如何工作的：

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
//: c11:Bits.java
// Demonstration of BitSet.
import java.util.*;

public class Bits {
  public static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    String bbits = new String( );
    for(int j = 0; j < b.size( ) ; j++)
      bbits += (b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
  }
  public static void main(String[] args) {
    Random rand = new Random( );
    // Take the LSB of nextInt( ):
    byte bt = (byte)rand.nextInt( );
    BitSet bb = new BitSet( );
    for(int i = 7; i >= 0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("byte value: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt( );
    BitSet bs = new BitSet( );
    for(int i = 15; i >= 0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
      else
        bs.clear(i);
    System.out.println("short value: " + st);
    printBitSet(bs);

    int it = rand.nextInt( );
    BitSet bi = new BitSet( );
    for(int i = 31; i >= 0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("int value: " + it);
    printBitSet(bi);

    // Test bitsets >= 64 bits:
    BitSet b127 = new BitSet( );
    b127.set(127);
    System.out.println("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    System.out.println("set bit 1023: " + b1023);
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

我们用随机数生成器生成了 **byte**，**short** 和 **int**，然后把它们转换成相应的 bit 形式，再存储到 **BitSet** 里面。这种做法很不错，因为 **BitSet** 是 64 位的，而它们都不会超出这个范围。接着它创建了一个 512 位的 **BitSet**。构造函数会分配两倍的大小的存储空间。但是你也可以设置 1024 位或者更多。

总结

总结 Java 标准类库的容器类；

1. 数组把对象和数字形式的下标联系起来。它持有的是类型确定的对象，这样提取对象的时候就不用再作类型传递了。它可以是多维的，也可以持有 primitive。但是创建之后它的容量不能改了。
2. **Collection** 持有单个元素，而 **Map** 持有相关联的 pair。
3. 和数组一样，**List** 也把数字下标同对象联系起来，你可以把数组和 **List** 想成有序的容器。**List** 会随元素的增加自动调整容量。但是 **List** 只能持有 **Object reference**，所以不能存放 primitive，而且把 **Object** 提取出来之后，还要做类型传递。
4. 如果要作很多随机访问，那么请用 **ArrayList**，但是如果要在 **List** 的中间作很多插入和删除的话，就应该用 **LinkedList** 了。
5. **LinkedList** 能提供队列，双向队列和栈的功能。
6. **Map** 提供的不是对象与数组的关联，而是对象和对象的关联。**HashMap** 看重的是访问速度，而 **TreeMap** 更看重键的顺序，因而它不如 **HashMap** 那么快。而 **LinkedHashMap** 则保持对象插入的顺序，但是也可以用 LRU 算法为它重新排序。
7. **Set** 只接受不重复的对象。**HashSet** 提供了最快的查询速度，而 **TreeSet** 则保持元素有序。**LinkedHashSet** 保持元素的插入顺序。
8. 没必要再在新代码里使用旧类库留下来的 **Vector**，**Hashtable** 和 **Stack** 了。

容器类库是你每天都会用到的工具，它能使程序更简洁，更强大并且更高效。

练习

只要付很小一笔费用就能从 *www.BruceEckel.com* 下载名为 *The Thinking in Java Annotated Solution Guide* 的电子文档，这上面有一些习题的答案。

1. 创建一个 **double** 类型的数组，然后用 **RandDoubleGenerator** 和 **fill( )**把它填满。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

2. 创建一个新的，带 **int gerbilNumber** 的 **Gerbil** 类，然后在构造函数里面进行初始化 (就像本章的 **Mouse.java**)。定义一个 **hop( )**方法，让它打印 **gerbilNumber**，以及“它正在跳”的信息。创建一个 **ArrayList**，再往里面加一串 **Gerbil** 对象。接下来用 **get( )**方法遍历一遍 **List**，再调用每个 **Gerbil** 对象的 **hop( )**。
3. 修改练习 2，改用 **Iterator** 来遍历 **List**。
4. 把练习 2 的 **Gerbil** 类放入 **Map**，把表示 **Gerbil** 对象名字的 **String**(也就是键)，与这个 **Gerbil** 对象(值)关联起来。获取 **keySet( )** 的 **Iterator**，然后用它来遍历 **Map**，根据键查找各个 **Gerbil**，再调用其 **hop( )**。
5. 创建一个 **List**(**ArrayList** 和 **LinkedList** 都尝试一下)，然后用 **Collections2.countries** 来进行填充。对 **List** 进行排序，然后把它打印出来，接下来再用 **Collections.shuffle( )**把顺序打乱，再打印这个 **List**，多试几次，看看每次调用 **shuffle( )**的时候，它都是怎么把顺序打乱的。
6. 证明一下，除了 **Mouse** 之外，**MouseList** 不接受任何对象。
7. 修改 **MouseList.java**，让它不是通过合成使用 **ArrayList**，而是继承 **ArrayList**。证明一下这么做是有问题的。
8. 创建一个只接收和返回 Cat 对象的 Cats 容器(使用 ArrayList)，并以此来修补 CatsAndDogs.java。
9. 用键值的 pair 来填充 **HashMap**。把这个 **HashMap** 打印出来，印证一下它是按 hash 数排序的，然后把它放入 **LinkedHashMap**。印证一下，它是按插入顺序排序的。
10. 用 **HashSet** 和 **LinkedHashSet** 重做上面那个练习。
11. 创建一种新的容器，用 **private ArrayList** 来保存对象。用 **Class** reference 来判断容器中的第一个对象的类型，然后只允许用户插入那种类型的对象。
12. 用 **String** 数组创建一个只能存取 **String** 的容器，这样使用的时候就没有类型转换的问题了。当容器发现数组不够大的时候，应该能自动调整其内部数组的大小。用 **main( )**作一下测试，看看是你自制的容器的性能好，还是 **ArrayList** 的性能好。
13. 重做练习 12，这次是做一个 **int** 的容器，然后比较它和保存 **Integer** 对象的 **ArrayList** 的性能。性能测试应该包括“对容器中的每个对象都做递增”的操作。
14. 使用 **com.bruceeckel.util** 中的实用工具，为每种 primitive，以及 **String** 对象各创建一个数字，然后用 generator 填充这个数组，再使用合适的 **print( )**方法打印这个数组。
15. 创建一个能生成你最喜欢的电影的名字的 generator(实在找不到，就用白雪公主，星球大战之类的)，如果名字用光了，就绕到最前面去。使用

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

com.bruceeckel.util 里面的实用工具来填补数组，**ArrayList**，**LinkedList**，以及两种 **Set**，然后把这些容器打印出来。

16. 创建一个包括两个 **String** 对象的类，然后做一个只比较第一个字符串的 **Comparable**。用 **geography** 的 generator 来生成这种对象，然后用这种对象来填充数组和 **ArrayList**。验证一下，排序能正常工作。再做一个只比较第二个 **String** 的 **Comparator**，然后验证一下排序也能正常工作。然后用 **Comparator** 进行以此 **binarySearch( )**。
17. 修改练习 16，让它按字母顺序排序。
18. 用 **Arrays2.RandStringGenerator** 生成的字符串，按字母顺序填充 **TreeSet**。把 **TreeSet** 打印出来，看看它是按什么顺序排列的。
19. 分别创建一个 **ArrayList** 和 **LinkedList**，用 **Collections2.captials** generator 来填充这个容器。用普通的 **Iterator** 打印这个列表，然后用 **ListIterator**，按照隔一个位置插一个对象的方式，把两个列表合并起来。然后从列表的末尾开始向前移动，并且执行插入操作。
20. 写一个用 **Iterator** 遍历 **Collection**，并且打印其中每个对象的 **hashCode( )**的方法。把对象填到各种 **Collection** 里面，然后测试这个方法。
21. 修复 **InfiniteRecursion.java** 的问题。
22. 先创建一个类，再创建一个用这个类的对象进行初始化的数组。把数组里的对象填到 **List** 里面，再用 **subList( )**创建一个这个 **List** 的子集，然后用 **removeAll( )**子集从 **List** 里面删除。
23. 修改第七章的练习 6，用 **ArrayList** 保存 **Rodent**，用 **Iterator** 遍历这个序列。记住，**ArrayList** 只保存 **Object**，所以要想访问 **Rodent**，先得做类型转换。
24. 模仿 **Queue.java**，创建一个 **Deque** 类，然后再测试一下。
25. 用 **TreeMap** 改写 **Statistics.java**。再加入测试代码，比较一下 **HashMap** 和 **TreeMap** 的性能。
26. 创建一个保存名字以“A”开头的国家的 **Map** 和 **Set**。
27. 用 **Collections2.countries** 重复填补 **Set**，并以此来验证，**Set** 不接受重复的对象。两种 **Set** 各试一次。
28. 修改 **Statistics.java**，创建一个重复进行测试的程序，看看一个数是不是比另一个数出现的频率高。
29. 用 **Couter** 对象的 **HashSet** 重写 **Statistics.java**(要对 Counter 类进行修改，这样它才能被放入 **HashSet** 里面)。比较一下，哪种方法更好。
30. 用 **String** 当键，你自己选择的对象当值，填充 **LinkedHashMap**。把键值 pair 提取出来，根据键排序，然后把它们重新插到 **Map** 里面。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

31. 修改练习 16 的类，让它能适用于 **HashSet**，并且能在 **HashMap** 里面充当键。
32. 参考 **SlowMap.java**，创建一个 **SlowSet**。
33. 创建一个 **FastTraversalLinkedList**，让它在内部用 **LinkedList** 进行快速的插入和删除，用 **ArrayList** 进行快速的遍历和 **get( )**操作。参考 **ArrayPerformance.java** 编写程序进行测试。
34. 对 **SlowMap** 进行 **Map1.java** 的测试，看看它是不是能正常工作。修改 **SlowMap**，让它能进行通过这个测试。
35. 修改 **SlowMap**，让它实现 **Map** 的全部接口。
36. 修改 **MapPerformance.java**，用它来测试一下 **SlowMap** 的性能。
37. 修改 **SlowMap**，改用 **MPair** 对象的 **ArrayList**，而不是两个 **ArrayList** 来实现。看看修改后的版本是不是也能正常工作。用 **MapPerformance.java** 测试一下新 **Map** 的速度。修改 **put( )**方法，让它先 **sort( )**再插入，然后修改 **get( )**，让它用 **Collections.binarySearch( )**来查询键。再比较一下新旧两个版本的速度。
38. 往 **CountedString** 里面加入一个也是用构造函数进行初始化的 **char** 成员，修改 **hashCode( )**和 **equals( )**方法，让它们把这个 **char** 也包括进去。
39. 修改 **SimpleHashMap**，让它报告冲突，然后往里面加重复的对象，并且观察冲突。
40. 修改 **SimpleHashMap**，让它报告需要经过几次测试才能发现冲突。也就是说，为了寻找相同的对象，遍历 **LinkedList** 的时候要调用几次 **Iterator** 的 **next( )**。
41. 实现 **SimpleHashMap** 的 **clear( )**和 **remove( )**方法。
42. 实现 **SimpleHashMap** 的其它 **Map** 接口。
43. 为 **SimpleHashMap** 加一个 **private rehash( )**方法，当 load factor 超过 0.75 的时候，就自动调用这个方法。rehash 的时候，要先对 bucket 的数量乘以二，再找出第一个比这个数大的质数，这个质数就是新的 bucket 的数量了。
44. 参照 **SimpleHashMap.java**，创建一个 **SimpleHashSet**，并进行测试。
45. 修改 **SimpleHashMap**，让它使用 **ArrayList**，而不是 **LinkedList**。修改 **MapPerformance.java**，让它比较这两种实现的性能。
46. 查阅 JDK 文档中 **HashMap** 的内容。创建一个 **HashMap**，设定其 load factor，再用各种元素进行填充。测试一下这个 **Map** 的查询速度。

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

然后创建一个新的 **HashMap**，并且把它的 initial capacity 加大，再把相同的元素加入这个 **Map**，测试一下查询速度，看看是不是更快了。

47. 找出第八章的 **GreenhouseController.java**，注意，总共有四个文件。**Controller.java** 用了 **ArrayList**，把它改成 **LinkedList**，并且用 **Iterator** 来处理事件。
48. (挑战级) 写一个你自己的，专为特定类型的键(比方说 **String**)定制的 hashed map。不许继承 **Map**。而且还要让 **put( )**和 **get( )**拿 **String** 而不是 **Object** 当键。牵涉到键的一切操作都不能使用泛型；相反，还要让它只适用于 **String**，这样就能省下上传和下传的开销了。目标是要写出最快的定制的实现。修改 **MapPerformance.java**，然后进行测试。
49. (挑战级) Java 的发布版里带着源代码。找到 **List** 的源代码，拷贝一份，再把它改写成专门保存 **int** 的 **intList**。思考一下，如果要写一个能保存所有 primitive 的 **List** 的话，需要考虑哪些问题。接下来再想想，要写一个能保存所有 primitive 数据的 linked list 的话，又要考虑哪些事情。
50. 修改 **c08:Month.java**，让它实现 **Comparable** 接口。
51. 修改 **CountedString.java** 的 **hashCode( )**，用 id 来代替乘法运算，证明一下，**CountedString** 仍然能用作键。但是这种做法有什么问题？

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com