如果枚举中的switch语句不是在枚举中实现特定于常量的行为的一种很好的选择，那么它们还有什么用处呢？枚举中的**switch**语句适合于给外部的枚举类型增加特定于常量的行为。例如，假设Operation枚举不受你的控制，你希望它有一个实例方法来返回每个运算的反运算。你可以用下列静态方法模拟这种效果：

```
// Switch on an enum to simulate a missing method
public static Operation inverse(Operation op) {
    switch(op) {
        case PLUS:   return Operation.MINUS;
        case MINUS:  return Operation.PLUS;
        case TIMES:  return Operation.DIVIDE;
        case DIVIDE: return Operation.TIMES;
        default:  throw new AssertionError("Unknown op: " + op);
    }
}
```

一般来说，枚举会优先使用comparable而非int常量。与int常量相比，枚举有个小小的性能缺点，即装载和初始化枚举时会有空间和时间的成本。除了受资源约束的设备，例如手机和烤面包机之外，在实践中不必太在意这个问题。

那么什么时候应该使用枚举呢？每当需要一组固定常量的时候。当然，这包括“天然的枚举类型”，例如行星、一周的天数以及棋子的数目等等。但它也包括你在编译时就知道其所有可能值的其他集合，例如菜单的选项、操作代码以及命令行标记等。枚举类型中的常量集并不一定要始终保持不变。专门设计枚举特性是考虑到枚举类型的二进制兼容演变。

总而言之，与int常量相比，枚举类型的优势是不言而喻的。枚举要易读得多，也更加安全，功能更加强大。许多枚举都不需要显式的构造器或者成员，但许多其他枚举则受益于“每个常量与属性的关联”以及“提供行为受这个属性影响的方法”。只有极少数的枚举受益于将多种行为与单个方法关联。在这种相对少见的情况下，特定于常量的方法要优先于启用自有值的枚举。如果多个枚举常量同时共享相同的行为，则考虑策略枚举。

更多资源请访问稀酷客(www.ckook.com)

第31条：用实例域代替序数

许多枚举天生就与一个单独的int值相关联。所有的枚举都有一个ordinal方法，它返回每个枚举常量在类型中的数字位置。你可以试着从序数中得到关联的int值：

```
// Abuse of ordinal to derive an associated value - DON'T DO THIS
public enum Ensemble {
    SOLO,   DUET,   TRIO, QUARTET, QUINTET,
    SEXTET, SEPTET, OCTET, NONET,  DECTET;

    public int numberOfMusicians() { return ordinal() + 1; }
}
```

虽然这个枚举不错，但是维护起来就像一场恶梦。如果常量进行重新排序，numberOfMusicians方法就会遭到破坏。如果要再添加一个与已经用过的int值关联的枚举常量，就没那么走运了。例如，给双四重奏（**double quartet**）添加一个常量，它就像个八重奏一样，是由8位演奏家组成，但是没有办法做到。

要是没有给所有这些int值添加常量，也无法给某个int值添加常量。例如，假设想要添加一个常量表示三四重奏（**triple quartet**），它由12位演奏家组成。对于由11位演奏家组成的合奏曲并没有标准的术语，因此只好给没有用过的int值（11）添加一个虚拟（dummy）常量。这么做顶多就是不太好看。如果有许多int值都是从未用过的，可就不切实际了。

幸运的是，有一种很简单的方法可以解决这些问题。*永远不要根据枚举的序数导出与它关联的值，而是要将它保存在一个实例域中：*

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) { this.numberOfMusicians = size; }
    public int numberOfMusicians() { return numberOfMusicians; }
}
```

Enum规范中谈到ordinal时这么写道："大多数程序员都不需要这个方法。它是设计成用于像EnumSet和EnumMap这种基于枚举的通用数据结构的。"除非你在编写的是这种数据结构，否则最好完全避免使用ordinal方法。

更多资源请访问稀酷客(www.ckook.com)

第32条：用EnumSet代替位域

如果一个枚举类型的元素主要用在集合中，一般就使用int枚举模式（见第30条），将2的不同倍数赋予每个常量：

```
// Bit field enumeration constants - OBSOLETE!
public class Text {
    public static final int STYLE_BOLD          = 1 << 0;  // 1
    public static final int STYLE_ITALIC        = 1 << 1;  // 2
    public static final int STYLE_UNDERLINE     = 1 << 2;  // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3;  // 8

    // Parameter is bitwise OR of zero or more STYLE_ constants
    public void applyStyles(int styles) { ... }
}
```

这种表示法让你用OR位运算将几个常量合并到一个集合中，称作位域（**bit field**）：

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

位域表示法也允许利用位操作，有效地执行像union（联合）和intersection（交集）这样的集合操作。但位域有着int枚举常量的所有缺点，甚至更多。当位域以数字形式打印时，翻译位域比翻译简单的int枚举常量要困难得多。甚至，要遍历位域表示的所有元素也没有很容易的方法。

有些程序员优先使用枚举而非int常量，他们在需要传递多组常量集时，仍然倾向于使用位域。其实没有理由这么做，因为还有更好的替代方法。java.util包提供了EnumSet类来有效地表示从单个枚举类型中提取的多个值的多个集合。这个类实现Set接口，提供了丰富的功能、类型安全性，以及可以从任何其他Set实现中得到的互用性。但是在内部具体的实现上，每个EnumSet内容都表示为位矢量。如果底层的枚举类型有64个或者更少的元素——大多如此——整个EnumSet就是用单个long来表示，因此它的性能比得上位域的性能。批处理，如removeAll和retainAll，都是利用位算法来实现的，就像手工替位域实现得那样。但是可以避免手工位操作时容易出现的错误以及不太雅观的代码，因为EnumSet替你完成了这项艰巨的工作。

下面是前一个范例改成用枚举代替位域后的代码，它更加简短、更加清楚，也更加安全：

```
// EnumSet - a modern replacement for bit fields
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

    // Any Set could be passed in, but EnumSet is clearly best
    public void applyStyles(Set<Style> styles) { ... }
}
```

更多资源请访问稀酷客(www.ckook.com)

下面是将EnumSet实例传递给applyStyles方法的客户端代码。EnumSet提供了丰富的静态工厂来轻松创建集合，其中一个如这个代码所示：

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

注意applyStyles方法采用的是Set<Style>而非EnumSet<Style>。虽然看起来好像所有的客户端都可以将EnumSet传到这个方法，但是最好还是接受接口类型而非接受实现类型。这是考虑到可能会有特殊的客户端要传递一些其他的Set实现，并且没有什么明显的缺点。

总而言之，正是因为枚举类型要用在集合（Set）中，所以没有理由用位域来表示它。EnumSet类集位域的简洁和性能优势及第30条中所述的枚举类型的所有优点于一身。实际上EnumSet有个缺点，即截止Java 1.6发行版本，它都无法创建不可变的EnumSet，但是这一点很可能在即将出来的版本中得到修正。同时，可以用Collections.unmodifiableSet将EnumSet封装起来，但是简洁性和性能会受到影响。

更多资源请访问稀酷客(www.ckook.com)

第33条：用EnumMap代替序数索引

有时候，你可能会见到利用ordinal方法（见第31条）来索引数组的代码。例如下面这个过于简化的类，用来表示一种烹饪用的香草：

```
public class Herb {
    public enum Type { ANNUAL, PERENNIAL, BIENNIAL }

    private final String name;
    private final Type type;

    Herb(String name, Type type) {
        this.name = name;
        this.type = type;
    }

    @Override public String toString() {
        return name;
    }
}
```

现在假设有一个香草的数组，表示一座花园中的植物，你想要按照类型（一年生、多年生或者两年生植物）进行组织之后将这些植物列出来。如果要这么做的话，需要构建三个集合，每种类型一个，并且遍历整座花园，将每种香草放到相应的集合中。有些程序员会将这些集合放到一个按照类型的序数进行索引的数组中来实现这一点。

```
// Using ordinal() to index an array - DON'T DO THIS!
Herb[] garden = ... ;

Set<Herb>[] herbsByType =  // Indexed by Herb.Type.ordinal()
    (Set<Herb>[]) new Set[Herb.Type.values().length];
for (int i = 0; i < herbsByType.length; i++)
    herbsByType[i] = new HashSet<Herb>();

for (Herb h : garden)
    herbsByType[h.type.ordinal()].add(h);

// Print the results
for (int i = 0; i < herbsByType.length; i++) {
    System.out.printf("%s: %s%n",
                      Herb.Type.values()[i], herbsByType[i]);
}
```

这种方法的确可行，但是隐藏着许多问题。因为数组不能与泛型（见第25条）兼容，程序需要进行未受检的转换，并且不能正确无误地进行编译。因为数组不知道它的索引代表着什么，你必须手工标注（label）这些索引的输出。但是这种方法最严重的问题在于，当你访问一个按照枚举的序数进行索引的数组时，使用正确的int值就是你的职责了；int不能提供枚举的类型安全。你如果使用了错误的值，程序就会悄悄地完成错误的工作，或者幸运的话，会抛出ArrayIndexOutOfBoundException异常。

更多资源请访问稀酷客(www.ckook.com)

幸运的是，有一种更好的方法可以达到同样的效果。数组实际上充当着从枚举到值的映射，因此可能还要用到Map。更具体地说，有一种非常快速的Map实现专门用于枚举键，称作java.util.EnumMap。以下就是用EnumMap改写后的程序：

```
// Using an EnumMap to associate data with an enum
Map<Herb.Type, Set<Herb>> herbsByType =
    new EnumMap<Herb.Type, Set<Herb>>(Herb.Type.class);
for (Herb.Type t : Herb.Type.values())
    herbsByType.put(t, new HashSet<Herb>());
for (Herb h : garden)
    herbsByType.get(h.type).add(h);
System.out.println(herbsByType);
```

这段程序更简短、更清楚，也更加安全，运行速度方面可以与使用序数的程序相媲美。它没有不安全的转换；不必手工标注这些索引的输出，因为映射键知道如何将自身翻译成可打印字符串的枚举；计算数组索引时也不可能出错。EnumMap在运行速度方面之所以能与通过序数索引的数组相媲美，是因为EnuMap在内部使用了这种数组。但是它对程序员隐藏了这种实现细节，集Map的丰富功能和类型安全与数组的快速于一身。注意EnumMap构造器采用键类型的Class对象：这是一个有限制的类型令牌（bounded type token），它提供了运行时的泛型信息（见第29条）。

你还可能见到按照序数进行索引（两次）的数组的数组，该序数表示两个枚举值的映射。例如，下面这个程序就是使用这样一个数组将两个阶段映射到一个阶段过渡中（从液体到固体称作凝固，从液体到气体称作沸腾，诸如此类）。

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase { SOLID, LIQUID, GAS;
    public enum Transition {
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;
        // Rows indexed by src-ordinal, cols by dst-ordinal
        private static final Transition[][] TRANSITIONS = {
            { null,    MELT,     SUBLIME },
            { FREEZE,  null,     BOIL    },
            { DEPOSIT, CONDENSE, null    }
        };

        // Returns the phase transition from one phase to another
        public static Transition from(Phase src, Phase dst) {
            return TRANSITIONS[src.ordinal()][dst.ordinal()];
        }
    }
}
```

这段程序可行，看起来也比较优雅，但是事实并非如此。就像上面那个比较简单的香草花园的示例一样，编译器无法知道序数和数组索引之间的关系。如果在过渡表中出了错，或者在修改Phase或者Phase.Transition枚举类型的时候忘记将它更新，程序就会在运行时失败。这种失败的形式可能为ArrayIndexOutOfBoundsException、NullPointerException或者（更糟糕的是）没有任何提示的错误行为。这张表的大小是阶段个数的平方，即使非null项的数量比较少。

更多资源请访问稀酷客(www.ckook.com)

同样，利用EnumMap依然可以做得更好一些。因为每个阶段过渡都是通过一对阶段枚举进行索引的，最好将这种关系表示为一个map，这个map的键是一个枚举（起始阶段），值为另一个map，这第二个map的键为第二个枚举（目标阶段），它的值为结果（阶段过渡），即形成了Map（起始阶段，Map（目标阶段，阶段过渡））这种形式。一个阶段过渡所关联的两个阶段，最好通过“数据与阶段过渡枚举之间的关联”来获取，之后用该阶段过渡枚举来初始化嵌套的EnumMap。

```
// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
   SOLID, LIQUID, GAS;

   public enum Transition {
      MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
      BOIL(LIQUID, GAS),   CONDENSE(GAS, LIQUID),
      SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);

      private final Phase src;
      private final Phase dst;

      Transition(Phase src, Phase dst) {
         this.src = src;
         this.dst = dst;
      }

      // Initialize the phase transition map
      private static final Map<Phase, Map<Phase,Transition>> m =
        new EnumMap<Phase, Map<Phase,Transition>>(Phase.class);
      static {
         for (Phase p : Phase.values())
           m.put(p,new EnumMap<Phase,Transition>(Phase.class));
         for (Transition trans : Transition.values())
           m.get(trans.src).put(trans.dst, trans);
      }

      public static Transition from(Phase src, Phase dst) {
         return m.get(src).get(dst);
      }
   }
}
```

初始化阶段过渡map的代码看起来可能有点复杂，但是还不算太糟糕。map的类型为Map<Phase, Map<Phase, Transition>>，表示是由键为源Phase（即第一个phase）、值为另一个map组成的Map，其中组成值的Map是由键值对目标Phase（即第二个Phase）、Transition组成的。静态初始化代码块中的第一个循环初始化了外部map，得到了三个空的内容map。代码块中的第二个循环利用每个状态过渡常量提供的起始信息和目标信息初始化了内部map。

现在假设想要给系统添加一个新的阶段：**plasma**（离子）或者电离气体。只有两个过渡与这个阶段关联：电离化，它将气体变成离子；以及消电离化，将离子变成气体。为了更新基于数组的程序，必须给Phase添加一种新常量，给Phase.Transition添加两种新常量，用一种新的16个元素的版本取代原来9个元素的数组的数组。如果给数组添加的元素过多或者过少，或者元素放置不妥当，可就麻烦了：程序可以编译，但是会在运行时失败。为了更新基于

更多资源请访问稀酷客(www.ckook.com)

EnumMap的版本，所要做的就是必须将PLASMA添加到Phase列表，并将IONIZE（GAS，PLASMA）和DEIONIZE（PLASMA，GAS）添加到Phase.Transition的列表中。程序会自行处理所有其他的事情，你几乎没有机会出错。从内部来看，Map的Map被实现成了数组的数组，因此在提升了清楚性、安全性和易维护性的同时，在空间或者时间上还几乎不用任何开销。

总而言之，最好不要用序数来索引数组，而要使用**EnumMap**。如果你所表示的这种关系是多维的，就使用EnumMap<...，EnumMap<...>>。应用程序的程序员在一般情况下都不使用Enum.ordinal，即使要用也很少，因此这是一种特殊情况（见第31条）。

更多资源请访问稀酷客(www.ckook.com)

第34条：用接口模拟可伸缩的枚举

就几乎所有方面来看，枚举类型都优越于本书第一版中所述的类型安全枚举模式[Bloch01]。从表面上看，有一个异常与可伸缩性有关，这个异常可能处在原来的模式中，却没有得到语言构造的支持。换句话说，使用这种模式，就有可能让一个枚举类型去扩展另一个枚举类型；利用这种语言特性，则不可能这么做。这绝非偶然。枚举的可伸缩性最后证明基本上都不是什么好点子。扩展类型的元素为基本类型的实例，基本类型的实例却不是扩展类型的元素，这样很是混乱。目前还没有很好的方法来枚举基本类型的所有元素及其扩展。最终，可伸缩性会导致设计和实现的许多方面变得复杂起来。

也就是说，对于可伸缩的枚举类型而言，至少有一种具有说服力的用例，这就是操作码（**operation code**），也称作**opcode**。操作码是指这样的枚举类型：它的元素表示在某种机器上的那些操作，例如第30条中的Operation类型，它表示一个简单的计算器中的某些函数。有时候，要尽可能地让API的用户提供它们自己的操作，这样可以有效地扩展API所提供的操作集。

幸运的是，有一种很好的方法可以利用枚举类型来实现这种效果。由于枚举类型可以通过给操作码类型和（属于接口的标准实现的）枚举定义接口，来实现任意接口，基本的想法就是利用这一事实。例如，以下是第30条中的Operation类型的扩展版本：

```
// Emulated extensible enum using an interface
public interface Operation {
    double apply(double x, double y);
}

public enum BasicOperation implements Operation {
    PLUS("+") {
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        public double apply(double x, double y) { return x - y; }
    },
    TIMES("*") {
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y; }
    };
    private final String symbol;
    BasicOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}
```

虽然枚举类型（BasicOperation）不是可扩展的，但接口类型（Operation）则是可扩展的，

更多资源请访问稀酷客(www.ckook.com)

它是用来表示API中的操作的接口类型。你可以定义另一个枚举类型，它实现这个接口，并用这个新类型的实例代替基本类型。例如，假设你想要定义一个上述操作类型的扩展，由求幂（exponentiation）和求余（remainder）操作组成。你所要做的就是编写一个枚举类型，让它实现Operation接口：

```
// Emulated extension enum
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    };

    private final String symbol;
    ExtendedOperation(String symbol) {
        this.symbol = symbol;
    }
    @Override public String toString() {
        return symbol;
    }
}
```

在可以使用基础操作的任何地方，都可以使用新的操作，只要API是被写成采用接口类型（Operation）而非实现（BasicOperation）。注意，在枚举中，不必像在不可扩展的枚举中所做的那样，利用特定于实例的方法实现来声明抽象的apply方法。这是因为抽象的方法（apply）是接口（Operation）的一部分。

不仅可以在任何需要“基本枚举”的地方单独传递一个“扩展枚举”的实例，而且除了那些基本类型的元素之外，还可以传递完整的扩展枚举类型，并使用它的元素。例如，通过下面这个测试程序，体验一下上面定义过的所有扩展过的操作：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}
private static <T extends Enum<T> & Operation> void test(
        Class<T> opSet, double x, double y) {
    for (Operation op : opSet.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

注意扩展过的操作类型的类的字面文字（ExtendedOperation.class）从main被传递给了test方法，来描述被扩展操作的集合。这个类的字面文字充当有限制的类型令牌（见第29条）。opSet参数中公认很复杂的声明（<T extends Enum<T> & Operation> Class<T>）确保了Class

更多资源请访问稀酷客(www.ckook.com)

对象既表示枚举又表示Operation的子类型，这正是遍历元素和执行与每个元素相关联的操作时所需要的。

第二种方法是使用Collection<? Extends Operation>，这是个有限制的通配符类型（**bounded wildcard type**）（见第28条），作为opSet参数的类型：

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet,
        double x, double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n",
                          x, op, y, op.apply(x, y));
}
```

这样得到的代码没有那么复杂，test方法也比较灵活一些：它允许调用者将多个实现类型的操作合并到一起。另一方面，也放弃了在指定操作上使用EnumSet（见第32条）和EnumMap（见第33条）的功能，因此，除非需要灵活地合并多个实现类型的操作，否则可能最好使用有限制的类型令牌。

上面这两段程序用命令行参数2和4运行时，都会产生这样的输出：

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

用接口模拟可伸缩枚举有个小小的不足，即无法将实现从一个枚举类型继承到另一个枚举类型。在上述Operation的示例中，保存和获取与某项操作相关联的符号的逻辑代码，可以复制到BasicOperation和ExtendedOperation中。在这个例子中是可以的，因为复制的代码非常少。如果共享功能比较多，则可以将它封装在一个辅助类或者静态辅助方法中，来避免代码的复制工作。

总而言之，虽然无法编写可扩展的枚举类型，却可以通过编写接口以及实现该接口的基础枚举类型，对它进行模拟。这样允许客户端编写自己的枚举来实现接口。如果API是根据接口编写的，那么在可以使用基础枚举类型的任何地方，也都可以使用这些枚举。

更多资源请访问稀酷客(www.ckook.com)

第35条：注解优先于命名模式

Java 1.5发行版本之前，一般使用命名模式（**naming pattern**）表明有些程序元素需要通过某种工具或者框架进行特殊处理。例如，JUnit测试框架原本要求它的用户一定要用test作为测试方法名称的开头[Beck04]。这种方法可行，但是有几个很严重的缺点。首先，文字拼写错误会导致失败，且没有任何提示。例如，假设不小心将一个测试方法命名为tsetSafetyOverride而不是testSafetyOverride。JUnit不会出错，但也不会执行测试，造成错误的安全感（即测试方法没有执行，它没有报错的可能，从而给人以测试正确的假象）。

命名模式的第二个缺点是，无法确保它们只用于相应的程序元素上。例如，假设将某个类称作testSafetyMechanisms，是希望JUnit会自动地测试它所有的方法，而不管它们叫什么名称。JUnit还是不会出错，但也同样不会执行测试。

命名模式的第三个缺点是，它们没有提供将参数值与程序元素关联起来的好方法。例如，假设想要支持一种测试类别，它只在抛出特殊异常时才会成功。异常类型本质上是测试的一个参数。你可以利用某种具体的命名模式，将异常类型名称编码到测试方法名称中，但是这样的代码会很不雅观，也很脆弱（见第50条）。编译器不知道要去检验准备命名异常的字符串是否真正命名成功。如果命名的类不存在，或者不是一个异常，你也要到试着运行测试时才会发现。

注解[JLS，9.7]很好地解决了所有这些问题。假设想要定义一个注解类型来指定简单的测试，它们自动运行，并在抛出异常时失败。以下就是这样的一个注解类型，命名为Test：

```
// Marker annotation type declaration
import java.lang.annotation.*;

/**
 * Indicates that the annotated method is a test method.
 * Use only on parameterless static methods.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Test注解类型的声明就是它自身通过Retention和Target注解进行了注解。注解类型声明中的这种注解被称作元注解（**meta-annotation**）。@Retention(RetentionPolicy.RUNTIME)元注解表明，Test注解应该在运行时保留。如果没有保留，测试工具就无法知道Test注解。@Target(ElementType.METHOD)元注解表明，Test注解只在方法声明中才是合法的：它不能运用到类声明、域声明或者其他程序元素上。

注意Test注解声明上方的注释："Use only on parameterless static method（只用于无参的

更多资源请访问稀酷客(www.ckook.com)

静态方法)”。如果编译器能够强制这一限制最好，但是它做不到。编译器可以替你完成多少错误检查，这是有限制的，即使是利用注解。如果将Test注解放在实例方法的声明中，或者放在带有一个或者多个参数的方法中，测试程序还是可以编译，让测试工具在运行时来处理这个问题。

下面就是现实应用中的Test注解，称作标记注解（**marker annotation**），因为它没有参数，只是“标注”被注解的元素。如果程序员拼错了Test，或者将Test注解应用到程序元素而非方法声明，程序就无法编译：

```
// Program containing marker annotations
public class Sample {
    @Test public static void m1() { }  // Test should pass
    public static void m2() { }
    @Test public static void m3() {    // Test Should fail
        throw new RuntimeException("Boom");
    }
    public static void m4() { }
    @Test public void m5() { } // INVALID USE: nonstatic method
    public static void m6() { }
    @Test public static void m7() {    // Test should fail
        throw new RuntimeException("Crash");
    }
    public static void m8() { }
}
```

Sample类有8个静态方法，其中4个被注解为测试。这4个中有2个抛出了异常：m3和m7，另外两个则没有：m1和m5。但是其中一个没有抛出异常的被注解方法：m5，是一个实例方法，因此不属于注解的有效使用。总之，Sample包含4项测试：一项会通过，两项会失败，另一项无效。没有用Test注解进行标注的4个方法会被测试工具忽略。

Test注解对Sample类的语义没有直接的影响。它们只负责提供信息供相关的程序使用。更一般地讲，注解永远不会改变被注解代码的语义，但是使它可以通过工具进行特殊的处理，例如像这种简单的测试运行类：

```
// Program to process marker annotations
import java.lang.reflect.*;

public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class testClass = Class.forName(args[0]);
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(Test.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " failed: " + exc);
                } catch (Exception exc) {
```

更多资源请访问稀酷客(www.ckook.com)

```
                    System.out.println("INVALID @Test: " + m);
                }
            }
        }
        System.out.printf("Passed: %d, Failed: %d%n",
                          passed, tests - passed);
    }
}
```

测试运行工具在命令行上使用完全匹配的类名，并通过调用Method.invoke反射式地运行类中所有标注了Test的方法。isAnnotationPresent方法告知该工具要运行哪些方法。如果测试方法抛出异常，反射机制就会将它封装在InvocationTargetException中。该工具捕捉到了这个异常，并打印失败报告，包含测试方法抛出的原始异常，这些信息是通过getCause方法从InvocationTargetException中提取出来的。

如果尝试通过反射调用测试方法时抛出InvocationTargetException之外的任何异常，表明编译时没有捕捉到Test注解的无效用法。这种用法包括实例方法的注解，或者带有一个或者多个参数的方法的注解，或者不可访问的方法的注解。测试运行类中的第二个catch块捕捉到了这些Test用法错误，并打印出相应的错误消息。下面就是RunTests在Sample上运行时打印的输出：

```
public static void Sample.m3() failed: RuntimeException: Boom
INVALID @Test: public void Sample.m5()
public static void Sample.m7() failed: RuntimeException: Crash
Passed: 1, Failed: 3
```

现在我们要针对只在抛出特殊异常时才成功的测试添加支持。为此我们需要一个新的注解类型：

```
// Annotation type with a parameter
import java.lang.annotation.*;
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception> value();
}
```

这个注解的参数类型是Class<? extends Exception>。这个通配符类型无疑很绕口。它在英语中的意思是：某个扩展Exception的类的Class对象，它允许注解的用户指定任何异常类型。这种用法是有限制的类型令牌（见第29条）的一个示例。下面就是实际应用中的这个注解。注意类名称被用作了注解的参数值：

```
// Program containing annotations with a parameter
public class Sample2 {
    @ExceptionTest(ArithmeticException.class)
```

更多资源请访问稀酷客(www.ckook.com)

```
    public static void m1() {  // Test should pass
        int i = 0;
        i = i / i;
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m2() {  // Should fail (wrong exception)
        int[] a = new int[0];
        int i = a[1];
    }
    @ExceptionTest(ArithmeticException.class)
    public static void m3() { }  // Should fail (no exception)
}
```

现在我们要修改一下测试运行工具来处理新的注解。这其中包括将以下代码添加到main方法中：

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (InvocationTargetException wrappedEx) {
        Throwable exc = wrappedEx.getCause();
        Class<? extends Exception> excType =
            m.getAnnotation(ExceptionTest.class).value();
        if (excType.isInstance(exc)) {
            passed++;
        } else {
            System.out.printf(
                "Test %s failed: expected %s, got %s%n",
                m, excType.getName(), exc);
        }
    } catch (Exception exc) {
        System.out.println("INVALID @Test: " + m);
    }
}
```

这段代码类似于用来处理Test注解的代码，但有一处不同：这段代码提取了注解参数的值，并用它检验该测试抛出的异常是否为正确的类型。没有显式的转换，因此没有出现ClassCastException的危险。编译过的测试程序确保它的注解参数表示的是有效的异常类型，需要提醒一点：有可能注解参数在编译时是有效的，但是表示特定异常类型的类文件在运行时却不再存在。在这种希望很少出现的情况下，测试运行类会抛出TypeNotPresentException异常。

将上面的异常测试示例再深入一点，想像测试可以在抛出任何一种指定异常时都得到通过。注解机制有一种工具，使得支持这种用法变得十分容易。假设我们将ExceptionTest注解的参数类型改成Class对象的一个数组：

```
// Annotation type with an array parameter
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    Class<? extends Exception>[] value();
}
```

更多资源请访问稀酷客(www.ckook.com)

注解中数组参数的语法十分灵活。它是进行过优化的单元素数组。使用了ExceptionTest新版的数组参数之后，之前的所有ExceptionTest注解仍然有效，并产生单元素的数组。为了指定多元素的数组，要用花括号（{}）将元素包围起来，并用逗号（,）将它们隔开：

```
// Code containing an annotation with an array parameter
@ExceptionTest({ IndexOutOfBoundsException.class,
                 NullPointerException.class })
public static void doublyBad() {
    List<String> list = new ArrayList<String>();

    // The spec permits this method to throw either
    // IndexOutOfBoundsException or NullPointerException
    list.addAll(5, null);
}
```

修改测试运行工具来处理新的ExceptionTest相当简单。下面的代码代替了原来的代码：

```
if (m.isAnnotationPresent(ExceptionTest.class)) {
    tests++;
    try {
        m.invoke(null);
        System.out.printf("Test %s failed: no exception%n", m);
    } catch (Throwable wrappedExc) {
        Throwable exc = wrappedExc.getCause();
        Class<? extends Exception>[] excTypes =
            m.getAnnotation(ExceptionTest.class).value();
        int oldPassed = passed;
        for (Class<? extends Exception> excType : excTypes) {
            if (excType.isInstance(exc)) {
                passed++;
                break;
            }
        }
        if (passed == oldPassed)
            System.out.printf("Test %s failed: %s %n", m, exc);
    }
}
```

本条目中开发的测试框架只是一个试验，但它清楚地示范了注解之于命名模式的优越性。它这还只是揭开了注解功能的冰山一角。如果是在编写一个需要程序员给源文件添加信息的工具，就要定义一组适当的注解类型。既然有了注解，就完全没有理由再使用命名模式了。

也就是说，除了“工具铁匠（toolsmiths——特定的程序员）”之外，大多数程序员都不必定义注解类型。但是所有的程序员都应该使用**Java**平台所提供的预定义的注解类型（见第36和24条）。还要考虑使用IDE或者静态分析工具所提供的任何注解。这种注解可以提升由这些工具所提供的诊断信息的质量。但是要注意这些注解还没有标准化，因此如果变换工具或者形成标准，就有很多工作要做了。

更多资源请访问稀酷客(www.ckook.com)

第36条：坚持使用Override注解

随着Java 1.5发行版本中增加注解，类库中也增加了几种注解类型[JLS，9.6]。对于传统的程序员而言，这里面最重要的就是Override注解了。这个注解只能用在方法声明中，它表示被注解的方法声明覆盖了超类型中的一个声明。如果坚持使用这个注解，可以防止一大类的非法错误。考虑下面的程序，这里的类Bigram表示一个双字母组或者有序的字母对：

```
// Can you spot the bug?
public class Bigram {
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());
    }
}
```

主程序反复地将26个双字母组添加到集合中，每个双字母组都由两个相同的小写字母组成。随后它打印出集合的大小。你可能以为程序打印出的大小为26，因为集合不能包含重复。如果你试着运行程序，会发现它打印的不是26而是260。哪里出错了呢？

很显然，Bigram类的创建者原本想要覆盖equals方法（见第8条），同时还记得覆盖了hashCode。遗憾的是，不幸的程序员没能覆盖equals，而是将它重载了（见第41条）。为了覆盖Object.equals，必须定义一个参数为Object类型的equals方法，但是Bigram的equals方法的参数并不是Object类型，因此Bigram从Object继承了equals方法。这个equals方法测试对象的同一性，就像==操作符一样。每个bigram的10个备份中，每一个都与其余的9个不同，因此Object.equals认为它们不相等，这正解释了程序为什么会打印出260的原因。

幸运的是，编译器可以帮助你发现这个错误，但是只有当你告知编译器你想要覆盖Object.equals时才行。为了做到这一点，要用@Override标注Bigram.euqals，如下所示：

```
@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

更多资源请访问稀酷客(www.ckook.com)

如果插入这个注解，并试着重新编译程序，编译器就会产生一条像这样的错误消息：

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

你会立即意识到哪里错了，拍拍自己的头，恍然大悟，马上用正确的来取代出错的equals实现（见第8条）：

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Bigram))
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

因此，应该在你想要覆盖超类声明的每个方法声明中使用**Override**注解。这一规则有个小小的例外。如果你在编写一个没有标注为抽象的类，并且确信它覆盖了抽象的方法，在这种情况下，就不必将Override注解放在该方法上了。在没有声明为抽象的类中，如果没有覆盖抽象的超类方法，编译器就会发出一条错误消息。但是，你可能希望关注类中所有覆盖超类方法的方法，在这种情况下，也可以放心地标注这些方法。

现代的IDE提供了坚持使用Override注解的另一种理由。这种IDE具有自动检查功能，称作代码检验（**code inspection**）。如果启用相应的代码检验功能，当有一个方法没有Override注解，却覆盖了超类方法时，IDE就会产生一条警告。如果坚持使用Override注解，这些警告就会提醒你警惕无意识的覆盖。这些警告补充了编译器的错误消息，提醒你警惕无意识的覆盖失败。IDE和编译器，可以确保你覆盖任何你想要覆盖的方法，无一遗漏。

如果你使用的是Java 1.6或者更新的发行版本，Override注解在查找Bug方面还提供了更多的帮助。在Java 1.6发行版本中，在覆盖接口以及类的方法声明中使用Override注解变成是合法的了。在被声明为去实现某接口的具体类中，不必标注出你想要这些方法来覆盖接口方法，因为如果你的类没有实现每一个接口方法，编译器就会产生一条错误消息。当然，你可以选择只包括这些注解，来标明它们是接口方法，但是这并非绝对必要。

但是在抽象类或者接口中，还是值得标注所有你想要的方法，来覆盖超类或者超接口方法，无论是具体的还是抽象的。例如，Set接口没有给Collection接口添加新方法，因此它应该在它的所有方法声明中包括Override注解，以确保它不会意外地给Collection接口添加任何新方法。

总而言之，如果在你想要的每个方法声明中使用Override注解来覆盖超类声明，编译器就可以替你防止大量的错误，但有一个例外。在具体的类中，不必标注你确信覆盖了抽象方法声明的方法（虽然这么做也没有什么坏处）。

更多资源请访问稀酷客(www.ckook.com)

第37条：用标记接口定义类型

标记接口（**marker interface**）是没有包含方法声明的接口，而只是指明（或者“标明”）一个类实现了具有某种属性的接口。例如，考虑Serializable接口（见第11章）。通过实现这个接口，类表明它的实例可以被写到ObjectOutputStream（或者“被序列化”）。

你可能听说过标记注解（见第35条）使得标记接口过时了。这种断言是不正确的。标记接口有两点胜过标记注解。首先，也是最重要的一点是，标记接口定义的类型是由被标记类的实例实现的；标记注解则没有定义这样的类型。这个类型允许你在编译时捕捉在使用标记注解的情况下要到运行时才能捕捉到的错误。

就Serializable标记接口而言，如果它的参数没有实现该接口，ObjectOutputStream.write(Object)方法将会失败。令人不解的是，ObjectOutputStream API的创建者在声明write方法时并没有利用Serializable接口。该方法的参数类型应该为Serializable而非Object。因此，试着在没有实现Serializable的对象上调用ObjectOutputStream.write，只会在运行时失败，但也并不一定如此。

标记接口胜过标记注解的另一个优点是，它们可以被更加精确地进行锁定。如果注解类型利用@Target（ElementType.TYPE）声明，它就可以被应用到任何类或者接口。假设有一个标记只适用于特殊接口的实现。如果将它定义成一个标记接口，就可以用它将唯一的接口扩展成它适用的接口。

Set接口可以说就是这种有限制的标记接口（**restricted marker interface**）。它只适用于Collection子类型，但是它不会添加除了Collection定义之外的方法。一般情况下，不把它当作是标记接口，因为它改进了几个Collection方法的契约，包括add、equals和hashCode。但是很容易想像只适用于某种特殊接口的子类型的标记接口，它没有改进接口的任何方法的契约。这种标记接口可以描述整个对象的某个约束条件，或者表明实例能够利用其他某个类的方法进行处理（就像Serializable接口表明实例可以通过ObjectOutputStream进行处理一样）。

标记注解胜过标记接口的最大优点在于，它可以通过默认的方式添加一个或者多个注解类型元素，给已被使用的注解类型添加更多的信息[JLS，9.6]。随着时间的推移，简单的标记注解类型可以演变成更加丰富的注解类型。这种演变对于标记接口而言则是不可能的，因为它通常不可能在实现接口之后再给它添加方法（见第18条）。

标记注解的另一个优点在于，它们是更大的注解机制的一部分。因此，标记注解在那些支持注解作为编程元素之一的框架中同样具有一致性。

更多资源请访问稀酷客(www.ckook.com)

那么什么时候应该使用标记注解，什么时候应该使用标记接口呢？很显然，如果标记是应用到任何程序元素而不是类或者接口，就必须使用注解，因为只有类和接口可以用来实现或者扩展接口。如果标记只应用给类和接口，就要问问自己：我要编写一个还是多个只接受有这种标记的方法呢？如果是这种情况，就应该优先使用标记接口而非注解。这样你就可以用接口作为相关方法的参数类型，它真正可以为你提供编译时进行类型检查的好处。

如果你对第一个问题的回答是否定的，就要再问问自己：我要永远限制这个标记只用于特殊接口的元素吗？如果是，最好将标记定义成该接口的一个子接口。如果这两个问题的答案都是否定的，或许就应该使用标记注解。

总而言之，标记接口和标记注解都各有用处。如果想要定义一个任何新方法都不会与之关联的类型，标记接口就是最好的选择。如果想要标记程序元素而非类和接口，考虑到未来可能要给标记添加更多的信息，或者标记要适合于已经广泛使用了注解类型的框架，那么标记注解就是正确的选择。如果你发现自己在编写的是目标为**ElementType.TYPE**的标记注解类型，就要花点时间考虑清楚，它是否真的应该为注解类型，想想标记接口是否会更加合适呢。

从某种意义上说，本条目与第19条中“如果不想定义类型就不要使用接口”的说法相反。本条目最接近的意思是说：如果想要定义类型，一定要使用接口。

更多资源请访问稀酷客(www.ckook.com)

第7章

方　法

本章要讨论方法设计的几个方面：如何处理参数和返回值，如何设计方法签名，如何为方法编写文档。本章中大多数内容既适用于构造器，也适用于普通的方法。与第5章一样，本章的焦点也集中在可用性、健壮性和灵活性上。

第38条：检查参数的有效性

绝大多数方法和构造器对于传递给它们的参数值都会有某些限制。例如，索引值必须是非负数，对象引用不能为null，等等，这些都是很常见的。你应该在文档中清楚地指明所有这些限制，并且在方法体的开头处检查参数，以强制施加这些限制。这是“应该在发生错误之后尽快检测出错误”这一普遍原则的一个具体情形。如果不能做到这一点，检测到错误的可能性就比较小，即使检测到错误了，也比较难以确定错误的根源。

如果传递无效的参数值给方法，这个方法在执行之前先对参数进行了检查，那么它很快就会失败，并且清楚地出现适当的异常（exception）。如果这个方法没有检查它的参数，就有可能发生几种情形。该方法可能在处理过程中失败，并且产生令人费解的异常。更糟糕的是，该方法可以正常返回，但是会悄悄地计算出错误的结果。最糟糕的是，该方法可以正常返回，但是却使得某个对象处于被破坏的状态，将来在某个不确定的时候，在某个不相关的点上会引发错误。

对于公有的方法，要用Javadoc的@throws标签（tag）在文档中说明违反参数值限制时会抛出的异常（见第62条）。这样的异常通常为IllegalArgumentException、IndexOutOfBoundsException或NullPointerException（见第60条）。一旦在文档中记录了对于方法参数的限制，并且记录了一旦违反这些限制将要抛出的异常，强加这些限制就是非常简单的事情了。下面是一个典型的例子：

更多资源请访问稀酷客(www.ckook.com)

```
/**
 * Returns a BigInteger whose value is (this mod m).  This method
 * differs from the remainder method in that it always returns a
 * non-negative BigInteger.
 *
 * @param  m the modulus, which must be positive
 * @return this mod m
 * @throws ArithmeticException if m is less than or equal to 0
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulus <= 0: " + m);
    ... // Do the computation
}
```

对于未被导出的方法（unexported method），作为包的创建者，你可以控制这个方法将在哪些情况下被调用，因此你可以，也应该确保只将有效的参数值传递进来。因此，非公有的方法通常应该使用断言（**assertion**）来检查它们的参数，具体做法如下所示：

```
// Private helper function for a recursive sort
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ... // Do the computation
}
```

从本质上讲，这些断言是在声称被断言的条件将会为真，无论外围包的客户端如何使用它。不同于一般的有效性检查，断言如果失败，将会抛出AssertionError。也不同于一般的有效性检查，如果它们没有起到作用，本质上也不会有成本开销，除非通过将-ea（或者-enableassertions）标记（flag）传递给Java解释器，来启用它们。关于断言的更多信息，请见Sun的教程[Asserts]。

对于有些参数，方法本身没有用到，却被保存起来供以后使用，检验这类参数的有效性尤为重要。例如，考虑第83页中的静态工厂方法，它的参数为一个int数组，并返回该数组的List视图。如果这个方法的客户端要传递null，该方法将会抛出一个NullPointerException，因为该方法包含一个显式的条件检查。如果省略了这个条件检查，它就会返回一个指向新建List实例的引用，一旦客户端企图使用这个引用，立即就会抛出NullPointerException。到那时，要想找到List实例的来源可能就非常困难了，从而使得调试工作极大地复杂化了。

如前所述，有些参数被方法保存起来供以后使用，构造器正是代表了这种原则的一种特殊情形。检查构造器参数的有效性是非常重要的，这样可以避免构造出来的对象违反了这个类的约束条件。

在方法执行它的计算任务之前，应该先检查它的参数，这一规则也有例外。一个很重要的例外是，在有些情况下，有效性检查工作非常昂贵，或者根本是不切实际的，而且有效性检查已隐含在计算过程中完成。例如，考虑一个为对象列表排序的方法：Collections.sort (List)。列表中的所有对象都必须是可以相互比较的。在为列表排序的过程中，列表中的每个对象将

更多资源请访问稀酷客(www.ckook.com)

与其他某个对象进行比较。如果这些对象不能相互比较，其中的某个比较操作就会抛出ClassCastException，这正是sort方法所应该做的事情。因此，提前检查列表中的元素是否可以相互比较，这并没有多大意义。然而，请注意，不加选择地使用这种方法将会导致失去失败原子性（failure atomicity）（见第64条）。

有时候，某些计算会隐式地执行必要的有效性检查，但是如果检查不成功，就会抛出错误的异常。换句话说，由于无效的参数值而导致计算过程抛出的异常，与文档中标明这个方法将抛出的异常并不相符。在这种情况下，应该使用第61条中讲述的异常转译（**exception translation**）技术，将计算过程中抛出的异常转换为正确的异常。

不要从本条目的内容中得出这样的结论：对参数的任何限制都是件好事。相反，在设计方法时，应该使它们尽可能地通用，并符合实际的需要。假如方法对于它能接受的所有参数值都能够完成合理的工作，对参数的限制就应该是越少越好。然而，通常情况下，有些限制对于被实现的抽象来说是固有的。

简而言之，每当编写方法或者构造器的时候，应该考虑它的参数有哪些限制。应该把这些限制写到文档中，并且在这个方法体的开头处，通过显式的检查来实施这些限制。养成这样的习惯是非常重要的。只要有效性检查有一次失败，你为必要的有效性检查所付出的努力便都可以连本带利地得到偿还了。

更多资源请访问稀酷客(www.ckook.com)

第39条：必要时进行保护性拷贝

使Java使用起来如此舒适的一个因素在于，它是一门安全的语言（**safe language**）。这意味着，它对于缓冲区溢出、数组越界、非法指针以及其他的内存破坏错误都自动免疫，而这些错误却困扰着诸如C和C++这样的不安全语言。在一门安全语言中，在设计类的时候，可以确切地知道，无论系统的其他部分发生什么事情，这些类的约束都可以保持为真。对于那些“把所有内存当作一个巨大的数组来看待”的语言来说，这是不可能的。

即使在安全的语言中，如果不采取一点措施，还是无法与其他的类隔离开来。假设类的客户端会尽其所能来破坏这个类的约束条件，因此你必须保护性地设计程序。实际上，只有当有人试图破坏系统的安全性时，才可能发生这种情形；更有可能的是，对你的API产生误解的程序员，所导致的各种不可预期的行为，只好由类来处理。无论是哪种情况，编写一些面对客户的不良行为时仍能保持健壮性的类，这是非常值得投入时间去做的事情。

没有对象的帮助时，虽然另一个类不可能修改对象的内部状态，但是对象很容易在无意识的情况下提供这种帮助。例如，考虑下面的类，它声称可以表示一段不可变的时间周期：

```
// Broken "immutable" time period class
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(
                start + " after " + end);
        this.start = start;
        this.end   = end;
    }

    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }

    ...  // Remainder omitted
}
```

乍一看，这个类似乎是不可变的，并且强加了约束条件：周期的起始时间（start）不能在结束时间（end）之后。然而，因为Date类本身是可变的，因此很容易违反这个约束条件：

更多资源请访问稀酷客(www.ckook.com)

```
// Attack the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
end.setYear(78);  // Modifies internals of p!
```

为了保护Period实例的内部信息避免受到这种攻击，对于构造器的每个可变参数进行保护性拷贝（**defensive copy**）是必要的，并且使用备份对象作为Period实例的组件，而不使用原始的对象：

```
// Repaired constructor - makes defensive copies of parameters
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end   = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
      throw new IllegalArgumentException(start +" after "+ end);
}
```

用了新的构造器之后，上述的攻击对于Period实例不再有效。注意，保护性拷贝是在检查参数的有效性（见第**38**条）之前进行的，并且有效性检查是针对拷贝之后的对象，而不是针对原始的对象。虽然这样做看起来有点不太自然，却是必要的。这样做可以避免在“危险阶段（window of vulnerability）”期间从另一个线程改变类的参数，这里的危险阶段是指从检查参数开始，直到拷贝参数之间的时间段。（在计算机安全社区中，这被称作**Time-Of-Check/Time-Of-Use**或者**TOCTOU**攻击[Viega01]。）

同时也请注意，我们没有用Date的clone方法来进行保护性拷贝。因为Date是非final的，不能保证clone方法一定返回类为java.util.Date的对象：它有可能返回专门出于恶意的目的而设计的不可信子类的实例。例如，这样的子类可以在每个实例被创建的时候，把指向该实例的引用记录到一个私有的静态列表中，并且允许攻击者访问这个列表。这将使得攻击者可以自由地控制所有的实例。为了阻止这种攻击，对于参数类型可以被不可信任方子类化的参数，请不要使用**clone**方法进行保护性拷贝。

虽然替换构造器就可以成功地避免上述的攻击，但是改变Period实例仍然是有可能的，因为它的访问方法提供了对其可变内部成员的访问能力：

```
// Second attack on the internals of a Period instance
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78);  // Modifies internals of p!
```

为了防御这第二种攻击，只需修改这两个访问方法，使它返回可变内部域的保护性拷贝即可：

```
// Repaired accessors - make defensive copies of internal fields
```

更多资源请访问稀酷客(www.ckook.com)

```
public Date start() {
    return new Date(start.getTime());
}

public Date end() {
    return new Date(end.getTime());
}
```

采用了新的构造器和新的访问方法之后，Period真正是不可变的了。不管程序员是多么恶意，或者多么不合格，都绝对不会违反“周期的起始时间不能落后于结束时间”这个约束条件。确实如此，因为除了Period类自身之外，其他任何类都无法访问Period实例中的任何一个可变域。这些域被真正封装在对象的内部。

访问方法与构造器不同，它们在进行保护性拷贝的时候允许使用clone方法。之所以如此，是因为我们知道，Period内部的Date对象的类是java.util.Date，而不可能是其他某个潜在的不可信子类。也就是说，基于第11条中所阐述的原因，一般情况下，最好使用构造器或者静态工厂。

参数的保护性拷贝并不仅仅针对不可变类。每当编写方法或者构造器时，如果它要允许客户提供的对象进入到内部数据结构中，则有必要考虑一下，客户提供的对象是否有可能是可变的。如果是，就要考虑你的类是否能够容忍对象进入数据结构之后发生变化。如果答案是否定的，就必须对该对象进行保护性拷贝，并且让拷贝之后的对象而不是原始对象进入到数据结构中。例如，如果你正在考虑使用由客户提供的对象引用作为内部Set实例的元素，或者作为内部Map实例的键（key），就应该意识到，如果这个对象在插入之后再被修改，Set或者Map的约束条件就会遭到破坏。

在内部组件被返回给客户端之前，对它们进行保护性拷贝也是同样的道理。不管类是否为不可变的，在把一个指向内部可变组件的引用返回给客户端之前，也应该加倍认真地考虑。解决方案是，应该返回保护性拷贝。记住长度非零的数组总是可变的。因此，在把内部数组返回给客户端之前，应该总要进行保护性拷贝。另一种解决方案是，给客户端返回该数组的不可变视图（immutable view）。这两种方法在第13条中都已经演示过了。

可以肯定地说，上述的真正启示在于，只要有可能，都应该使用不可变的对象作为对象内部的组件，这样就不必再为保护性拷贝（见第15条）操心。在前面的Period例子中，值得一提的是，有经验的程序员通常使用Date.getTime()返回的long基本类型作为内部的时间表示法，而不是使用Date对象引用。他们之所以这样做，主要因为Date是可变的。

保护性拷贝可能会带来相关的性能损失，这种说法并不总是正确的。如果类信任它的调用者不会修改内部的组件，可能因为类及其客户端都是同一个包的双方，那么不进行保护性拷贝也是可以的。在这种情况下，类的文档中就必须清楚地说明，调用者绝不能修改受到影响的参数或者返回值。

更多资源请访问稀酷客(www.ckook.com)

即使跨越包的作用范围，也并不总是适合在将可变参数整合到对象中之前，对它进行保护性拷贝。有一些方法和构造器的调用，要求参数所引用的对象必须有个显式的交接（**handoff**）过程。当客户端调用这样的方法时，它承诺以后不再直接修改该对象。如果方法或者构造器期望接管一个由客户端提供的可变对象，它就必须在文档中明确地指明这一点。

如果类所包含的方法或者构造器的调用需要移交对象的控制权，这个类就无法让自身抵御恶意的客户端。只有当类和它的客户端之间有着互相的信任，或者破坏类的约束条件不会伤害到除了客户端之外的其他对象时，这种类才是可以接受的。后一种情形的例子是包装类模式（wrapper class pattern）（见第16条）。根据包装类的本质特征，客户端只需在对象被包装之后直接访问它，就可以破坏包装类的约束条件，但是，这么做往往只会伤害到客户端自己。

简而言之，如果类具有从客户端得到或者返回到客户端的可变组件，类就必须保护性地拷贝这些组件。如果拷贝的成本受到限制，并且类信任它的客户端不会不恰当地修改组件，就可以在文档中指明客户端的职责是不得修改受到影响的组件，以此来代替保护性拷贝。

更多资源请访问稀酷客(www.ckook.com)

第40条：谨慎设计方法签名

本条目是若干API设计技巧的总结，它们都还不足以单独开设一个条目。综合来说，这些设计技巧将有助于使你的API更易于学习和使用，并且比较不容易出错。

谨慎地选择方法的名称。方法的名称应该始终遵循标准的命名习惯（见第56条）。首要目标应该是选择易于理解的，并且与同一个包中的其他名称风格一致的名称。第二个目标应该是选择与大众认可的名称（如果存在的话）相一致的名称。如果还有疑问，请参考Java类库的API。尽管Java类库的API中也有大量不一致的地方，考虑到这些Java类库的规模和范围，这是不可避免的，但它还是得到了相当程度的认可。

不要过于追求提供便利的方法。每个方法都应该尽其所能。方法太多会使类难以学习、使用、文档化、测试和维护。对于接口而言，这无疑是正确的，方法太多会使接口实现者和接口用户的工作变得复杂起来。对于类和接口所支持的每个动作，都提供一个功能齐全的方法。只有当一项操作被经常用到的时候，才考虑为它提供快捷方式（shorthand）。*如果不能确定，还是不提供快捷为好。*

避免过长的参数列表。目标是四个参数，或者更少。大多数程序员都无法记住更长的参数列表。如果你编写的许多方法都超过了这个限制，你的API就不太便于使用，除非用户不停地参考它的文档。现代的IDE会有所帮助，但最好还是使用简短的参数列表。*相同类型的长参数序列格外有害*。API的用户不仅无法记住参数的顺序，而且，当他们不小心弄错了参数顺序时，他们的程序仍然可以编译和运行，只不过这些程序不会按照作者的意图进行工作。

有三种方法可以缩短过长的参数列表。第一种是把方法分解成多个方法，每个方法只需要这些参数的一个子集。如果不小心，这样做会导致方法过多。但是通过提升它们的正交性（orthogonality），还可以*减少*（**reduce**）方法的数目。例如，考虑java.util.List接口。它并没有提供“在子列表（sublist）中查找元素的第一个索引和最后一个索引”的方法，这两个方法都需要三个参数。相反，它提供了subList方法，这个方法带有两个参数，并返回子列表的一个*视图*（**view**）。这个方法可以与indexOf或者lastIndexOf方法结合起来，获得期望的功能，而这两个方法都分别只有一个参数。而且，subList方法也可以与其他任何“针对List实例进行操作”的方法结合起来，在子列表上执行任意的计算。这样得到的API就有很高的“功能－重量”（power-to-weight）比。

缩短长参数列表的第二种方法是创建*辅助类*（**helper class**），用来保存参数的分组。这些辅助类一般为静态成员类（见第22条）。如果一个频繁出现的参数序列可以被看作是代表了某个独特的实体，则建议使用这种方法。例如，假设你正在编写一个表示纸牌游戏的类，你会

更多资源请访问稀酷客(www.ckook.com)

发现，经常要传递一个两参数的序列来表示纸牌的点数和花色。如果增加辅助类来表示一张纸牌，并且把每个参数序列都换成这个辅助类的单个参数，那么这个纸牌游戏类的API以及它的内部表示都可能会得到改进。

结合了前两种方法特征的第三种方法是，从对象构建到方法调用都采用Builder模式（请见第2条）。如果方法带有多个参数，尤其是当它们中有些是可选的时候，最好定义一个对象来表示所有参数，并允许客户端在这个对象上进行多次“setter”调用，每次调用都设置一个参数，或者设置一个较小的相关的集合。一旦设置了需要的参数，客户端就调用对象的“执行（execute）”方法，它对参数进行最终的有效性检查，并执行实际的计算。

对于参数类型，要优先使用接口而不是类（请见第52条）。只要有适当的接口可用来定义参数，就优先使用这个接口，而不是使用实现该接口的类。例如，没有理由在编写方法时使用HashMap类来作为输入，相反，应当使用Map接口作为参数。这使你可以传入一个Hashtable、HashMap、TreeMap、TreeMap的子映射表（submap），或者任何有待于将来编写的Map实现。如果使用的是类而不是接口，则限制了客户端只能传入特定的实现，如果碰巧输入的数据是以其他的形式存在，就会导致不必要的、可能非常昂贵的拷贝操作。

*对于**boolean**参数，要优先使用两个元素的枚举类型*。它使代码更易于阅读和编写，尤其当你在使用支持自动完成功能的IDE的时候。它也使以后更易于添加更多的选项。例如，你可能会有一个Thermometer类型，它带有一个静态工厂方法，而这个静态工厂方法的签名需要传入这个枚举的值：

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }
```

Thermometer.newInstance(TemperatureScale.CELSIUS)不仅比Thermometer.newInstance(true)更有用，而且你还可以在未来的发行版本中将KELVIN添加到TemperatureScale中，无需非得给Thermometer添加新的静态工厂。你还可以将依赖于温度刻度单位的代码重构到枚举常量的方法中（见第30条）。例如，每个刻度单位都可以有一个方法，它带有一个double值，并将它规格化成摄氏度。

更多资源请访问稀酷客(www.ckook.com)

第41条：慎用重载

下面这个程序的意图是好的，它试图根据一个集合（collection）是Set、List，还是其他的集合类型，来对它进行分类：

```
// Broken! - What does this program print?
public class CollectionClassifier {
    public static String classify(Set<?> s) {
        return "Set";
    }

    public static String classify(List<?> lst) {
        return "List";
    }

    public static String classify(Collection<?> c) {
        return "Unknown Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
            new HashSet<String>(),
            new ArrayList<BigInteger>(),
            new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections)
            System.out.println(classify(c));
    }
}
```

你可能期望这个程序会打印出“Set”，紧接着是“List”，以及“Unknown Collection”，但实际上不是这样。它是打印“Unknown Collection”三次。为什么会这样呢？因为classify方法被重载（**overloaded**）了，而要调用哪个重载（**overloading**）方法是在编译时做出决定的。对于for循环中的全部三次迭代，参数的编译时类型都是相同的：Collection<?>。每次迭代的运行时类型都是不同的，但这并不影响对重载方法的选择。因为该参数的编译时类型为Collection<?>，所以，唯一合适的重载方法是第三个：classify(Collection<?>)，在循环的每次迭代中，都会调用这个重载方法。

这个程序的行为有悖常理，因为对于重载方法（**overloaded method**）的选择是静态的，而对于被覆盖的方法（**overridden method**）的选择则是动态的。选择被覆盖的方法的正确版本是在运行时进行的，选择的依据是被调用方法所在对象的运行时类型。这里重新说明一下，当一个子类包含的方法声明与其祖先类中的方法声明具有同样的签名时，方法就被覆盖了。如果实例方法在子类中被覆盖了，并且这个方法是在该子类的实例上被调用的，那么子类中的覆盖方法（**overriding method**）将会执行，而不管该子类实例的编译时类型到底是什么。为了进行更具体的说明，考虑下面这个程序：

更多资源请访问稀酷客(www.ckook.com)

```
class Wine {
    String name() { return "wine"; }
}

class SparklingWine extends Wine {
    @Override String name() { return "sparkling wine"; }
}

class Champagne extends SparklingWine {
    @Override String name() { return "champagne"; }
}

public class Overriding {
    public static void main(String[] args) {
        Wine[] wines = {
            new Wine(), new SparklingWine(), new Champagne()
        };
        for (Wine wine : wines)
            System.out.println(wine.name());
    }
}
```

name方法是在类Wine中被声明的，但是在类SparklingWine和Champagne中被覆盖。正如你所预期的那样，这个程序打印出“wine，sparkling wine和champagne”，尽管在循环的每次迭代中，实例的编译时类型都为Wine。当调用被覆盖的方法时，对象的编译时类型不会影响到哪个方法将被执行；“最为具体的（most specific）”那个覆盖版本总是会得到执行。这与重载的情形相比，对象的运行时类型并不影响“哪个重载版本将被执行”；选择工作是在编译时进行的，完全基于参数的编译时类型。

在CollectionClassifier这个示例中，该程序的意图是：期望编译器根据参数的运行时类型自动将调用分发给适当的重载方法，以此来识别出参数的类型，就好像Wine的例子中的name方法所做的那样。方法重载机制完全没有提供这样的功能。假设需要有个静态方法，这个程序的最佳修正方案是，用单个方法来替换这三个重载的classify方法，并在这个方法中做一个显式的instanceof测试：

```
public static String classify(Collection<?> c) {
    return c instanceof Set  ? "Set" :
           c instanceof List ? "List" : "Unknown Collection";
}
```

因为覆盖机制是规范，而重载机制是例外，所以，覆盖机制满足了人们对于方法调用行为的期望。正如CollectionClassifier例子所示，重载机制很容易使这些期望落空。如果编写出来的代码的行为可能使程序员感到困惑，它就是很糟糕的实践。对于API来说尤其如此。如果API的普通用户根本不知道“对于一组给定的参数，其中的哪个重载方法将会被调用”，那么，使用这样的API就很可能导致错误。这些错误要等到运行时发生了怪异的行为之后才会显现出来，许多程序员无法诊断出这样的错误。因此，应该避免胡乱地使用重载机制。

到底怎样才算胡乱使用重载机制呢？这个问题仍有争议。安全而保守的策略是，永远不要

更多资源请访问稀酷客(www.ckook.com)

导出两个具有相同参数数目的重载方法。如果方法使用可变参数（**varargs**），保守的策略是根本不要重载它，除第42条中所述的情形之外。如果你遵守这些限制，程序员永远也不会陷入到“对于任何一组实际的参数，哪个重载方法是适用的”这样的疑问中。这项限制并不麻烦，因为你始终可以给方法起不同的名称，而不使用重载机制。

例如，考虑ObjectOutputStream类。对于每个基本类型，以及几种引用类型，它的write方法都有一种变形。这些变形方法并不是重载write方法，而是具有诸如writeBoolean(boolean)、writeInt(int)和writeLong(long)这样的签名。与重载方案相比较，这种命名模式带来的好处是，有可能提供相应名称的读方法，比如readBoolean()、readInt()和readLong()。实际上，ObjectInputStream类正是提供了这样的读方法。

对于构造器，你没有选择使用不同名称的机会；一个类的多个构造器总是重载的。在许多情况下，可以选择导出静态工厂，而不是构造器（见第1条）。对于构造器，还不用担心重载和覆盖的相互影响，因为构造器不可能被覆盖。或许你有可能导出多个具有相同参数数目的构造器，所以有必要了解一下如何安全地做到这一点。

如果对于“任何一组给定的实际参数将应用于哪个重载方法上”始终非常清楚，那么，导出多个具有相同参数数目的重载方法就不可能使程序员感到混淆。如果对于每一对重载方法，至少有一个对应的参数在两个重载方法中具有“根本不同（radically different）”的类型，就属于这种情形。如果显然不可能把一种类型的实例转换为另一种类型，这两种类型就是根本不同的。在这种情况下，一组给定的实际参数应用于哪个重载方法上就完全由参数的运行时类型来决定，不可能受到其编译时类型的影响，所以主要的混淆根源就消除了。例如，ArrayList有一个构造器带一个int参数，另一个构造器带一个Collection参数。难以想像在什么情况下，会不清楚要调用哪一个构造器。

在Java 1.5发行版本之前，所有的基本类型都根本不同于所有的引用类型，但是当自动装箱出现之后，就不再如此了，它会导致真正的麻烦。考虑下面这个程序：

```
public class SetList {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        List<Integer> list = new ArrayList<Integer>();

        for (int i = -3; i < 3; i++) {
            set.add(i);
            list.add(i);
        }
        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }
}
```

更多资源请访问稀酷客(www.ckook.com)

程序将－3至2之间的整数添加到了排好序的集合和列表中，然后在集合和列表中都进行3次相同的remove调用。如果像大多数人一样，希望程序从集合和列表中去除非整数值（0，1和2），并打印出[－3，－2，－1] [－3，－2，－1]。事实上，程序从集合中去除了非整数，还从列表中去除了奇数值，打印出[－3，－2，－1] [－2，0，2]。将这种行为称之为混乱，已是保守的说法。

实际发生的情况是：set.remove(i)调用选择重载方法remove(E)，这里的E是集合（Integer）的元素类型，将i从int自动装箱到Integer中。这是你所期待的行为，因此程序不会从集合中去除正值。另一方面，list.remove(i)调用选择重载方法remove(int i)，它从列表的指定位置上去除元素。如果从列表[－3，－2，－1，0，1，2]开始，去除第零个元素，接着去除第一个、第二个，得到的是[－2，0，2]，这个秘密被揭开了。为了解决这个问题，要将list.remove的参数转换成Integer，迫使选择正确的重载方法。另一种方法是，可以调用Integer.valueOf(i)，并将结果传给list.remove。这两种方法都如我们所料，打印出[－3，－2，－1] [－3，－2，－1]：

```
for (int i = 0; i < 3; i++) {
    set.remove(i);
    list.remove((Integer) i);  // or remove(Integer.valueOf(i))
}
```

前一个范例中所示的混乱行为在这里也出现了，因为List<E>接口有两个重载的remove方法：remove(E)和remove(int)。当它在Java 1.5发行版本中被泛型化之前，List接口有一个remove(Object)而不是remove(E)，相应的参数类型：Object和int，则根本不同。但是自从有了泛型和自动装箱之后，这两种参数类型就不再根本不同了。换句话说，Java语言中添加了泛型和自动装箱之后，破坏了List接口。幸运的是，Java类库中几乎再没有API受到同样的破坏，但是这种情形清楚地说明了，自动装箱和泛型成了Java语言的一部分之后，谨慎重载显得更加重要了。

数组类型和Object之外的类截然不同。数组类型和Serializable与Cloneable之外的接口也截然不同。如果两个类都不是对方的后代，这两个独特的类就是不相关的（**unrelated**）[JLS，5.5]。例如，String和Throwable就是不相关的。任何对象都不可能是两个不相关的类的实例，因此不相关的类是根本不同的。

还有其他一些“类型对”的例子也是不能相互转换的[JLS, 5.1.12]，但是，一旦超出了上述这些简单的情形，大多数程序员要想搞清楚“一组实际的参数应用于哪个重载方法上”就会非常困难。确定选择哪个重载方法的规则是非常复杂的。这些规则在语言规范中占了33页的篇幅[JLS, 15.12.1-3]，很少有程序员能够理解其中的所有微妙之处。

有时候，尤其在更新现有类的时候，可能会被迫违反本条目的指导原则。例如，自从Java 1.4发行版本以来，String类就已经有一个contentEquals（StringBuffer）方法。在Java 1.5发行

更多资源请访问稀酷客(www.ckook.com)

版本中，新增了一个称作CharSequence的接口，用来为StringBuffer、StringBuilder、String、CharBuffer以及其他类似的类型提供公共接口，为实现这个接口，对它们全都进行了改造。在Java平台中增加CharSequence的同时，String也配备了重载的contentEquals方法，即contentEquals（CharSequence）方法，这个方法表示，当且仅当此String表示与CharSequence序列相同的char值时，返回true。

尽管这样的重载很显然违反了本条目的指导原则，但是只要当这两个重载方法在同样的参数上被调用时，它们执行相同的功能，重载就不会带来危害。程序员可能并不知道哪个重载函数会被调用，但只要这两个方法返回相同的结果就行。确保这种行为的标准做法是，让更具体化的重载方法把调用转发给更一般化的重载方法：

```
public boolean contentEquals(StringBuffer sb) {
    return contentEquals((CharSequence) sb);
}
```

虽然Java平台类库很大程度上遵循了本条目中的建议，但是也有诸多的类违背了。例如，String类导出两个重载的静态工厂方法：valueOf(char[])和valueOf(Object)，当这两个方法被传递了同样的对象引用时，它们所做的事情完全不同。没有正当的理由可以解释这一点，它应该被看作是一种反常行为，有可能会造成真正的混淆。

简而言之，“能够重载方法”并不意味着就“应该重载方法”。一般情况下，对于多个具有相同参数数目的方法来说，应该尽量避免重载方法。在某些情况下，特别是涉及构造器的时候，要遵循这条建议也许是不可能的。在这种情况下，至少应该避免这样的情形：同一组参数只需经过类型转换就可以被传递给不同的重载方法。如果不能避免这种情形，例如，因为正在改造一个现有的类以实现新的接口，就应该保证：当传递同样的参数时，所有重载方法的行为必须一致。如果不能做到这一点，程序员就很难有效地使用被重载的方法或者构造器，他们就不能理解它为什么不能正常地工作。

更多资源请访问稀酷客(www.ckook.com)

第42条：慎用可变参数

Java 1.5发行版本中增加了可变参数（varargs）方法，一般称作**variable arity method**（可匹配不同长度的变量的方法）[JLS，8.4.1]。可变参数方法接受0个或者多个指定类型的参数。可变参数机制通过先创建一个数组，数组的大小为在调用位置所传递的参数数量，然后将参数值传到数组中，最后将数组传递给方法。

例如，下面就是一个可变参数方法，带有int参数的一个序列，并返回它们的总和。正如你所期望的，sum(1, 2, 3)的值为6，sum()的值为0：

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

有时候，有必要编写需要1个或者多个某种类型参数的方法，而不是需要0个或者多个。例如，假设想要计算多个int参数的最小值。如果客户端没有传递参数，那这个方法的定义就不太好了。你可以在运行时检查数组长度：

```
// The WRONG way to use varargs to pass one or more arguments!
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("Too few arguments");
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

这种解决方案有几个问题。其中最严重的问题是，如果客户端调用这个方法时，并没有传递参数进去，它就会在运行时而不是编译时失败。另一个问题是，这段代码很不美观。你必须在args中包含显式的有效性检查，除非将min初始化为Integer.MAX_VALUE，否则将无法使用for-each循环，这样的代码也不美观。

幸运的是，有一种更好的方法可以实现想要的这种效果。声明该方法带有两个参数，一个是指定类型的正常参数，另一个是这种类型的varargs参数。这种解决方案解决了前一个示例中的所有不足：

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
```

更多资源请访问稀酷客(www.ckook.com)

```
            min = arg;
    return min;
}
```

如你所见，当你真正需要让一个方法带有不定数量的参数时，可变参数就非常有效。可变参数是为printf而设计的，它是在Java 1.5发行版本中添加到平台中的，为了核心的反射机制（见第53条），在该发行版本中被改造成利用可变参数。printf和反射机制都从可变参数中极大地受益。

可以将以数组当作final参数的现有方法，改造成以可变参数代替，而不影响现有的客户端。但是可以并不意味着应该这么做！考虑Arrays.asList的情形。这个方法从来都不是设计成用来将多个参数集中到一个列表中的，但是当平台中增加了可变参数时，将它改造成这么做似乎是个好办法。因此，变成可以这样：

```
List<String> homophones = Arrays.asList("to", "too", "two");
```

这种用法有效，但是这么用就是个大错误。在Java 1.5发行版本之前，打印数组内容的常见做法如下：

```
// Obsolete idiom to print an array!
System.out.println(Arrays.asList(myArray));
```

这种做法在当时是必需的，因为数组从Object继承了它们的toString实现，因此直接在数组上调用toString，会产生没有意义的字符串，如[Ljava.lang.Integer;@3e25a5。这种做法只在对象引用类型的数组上才有用，但是如果不小心在基本类型的数组上尝试这么做，程序将无法编译。例如，这个程序：

```
public static void main(String[] args) {
    int[] digits = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 4 };
    System.out.println(Arrays.asList(digits));
}
```

在发行版本1.4中，这个程序会产生这条错误消息：

```
Va.java:6: asList(Object[]) in Arrays can't be applied to (int[])
        System.out.println(Arrays.asList(digits));
                                        ^
```

由于在Java 1.5发行版本中，令人遗憾地决定将Arrays.asList改造成可变参数方法，现在这个程序可以通过编译，并且没有错误或者警告。但是运行这个程序时，会输出无意识的也是无意义的结果：[[I@3e25a5]。Arrays.asList方法现在“增强”为使用可变参数，将int类型的数组digits的对象引用集中到数组的单个元素数组中，并忠实地将它包装到List<int[]>实例中。打印这个列表会导致在列表中调用toString，从而导致在它唯一的元素int数组上调用toString，产生上述令人遗憾的结果。

更多资源请访问稀酷客(www.ckook.com)

从好的方面看，将数组转变成字符串的Arrays.asList做法现在是过时的，当前的做法要健壮得多。也是在Java 1.5发行版本中，Arrays类得到了补充完整的Arrays.toString方法（不是可变参数方法！），专门为了将任何类型的数组转变成字符串而设计的。如果用Arrays.toString代替Arrays.asList，这个程序就会产生想要的结果：

```
// The right way to print an array
System.out.println(Arrays.toString(myArray));
```

如果不改造Arrays.asList，更好的办法则是给Collections添加一个新的方法，专门用来将它的参数集中到列表中：

```
public static <T> List<T> gather(T... args) {
    return Arrays.asList(args);
}
```

这种方法可以提供收集功能，而不会危及对现有Arrays.asList方法的类型检查。

这个教训很明显。*不必改造具有**final**数组参数的每个方法；只当确实是在数量不定的值上执行调用时才使用可变参数。*

有两个方法签名特别可疑：

```
ReturnType1 suspect1(Object... args) { }
<T> ReturnType2 suspect2(T... args) { }
```

带有上述任何一种签名的方法都可以接受任何参数列表。改造之前进行的任何编译时的类型检查都会丢失，Arrays.asList发生的情形正是说明了这一点。

在重视性能的情况下，使用可变参数机制要特别小心。可变参数方法的每次调用都会导致进行一次数组分配和初始化。如果凭经验确定无法承受这一成本，但又需要可变参数的灵活性，还有一种模式可以让你如愿以偿。假设确定对某个方法95%的调用会有3个或者更少的参数，就声明该方法的5个重载，每个重载方法带有0至3个普通参数，当参数的数目超过3个时，就使用一个可变参数方法：

```
public void foo() { }
public void foo(int a1) { }
public void foo(int a1, int a2) { }
public void foo(int a1, int a2, int a3) { }
public void foo(int a1, int a2, int a3, int... rest) { }
```

现在你知道了，所有调用中只有5%参数数量超过3个的调用需要创建数组。就像大多数的性能优化一样，这种方法通常不太恰当，但是一旦真正需要它时，它可就帮上大忙了。

EnumSet类对它的静态工厂使用这种方法，最大限度地减少创建枚举集合的成本。当时这

更多资源请访问稀酷客(www.ckook.com)

么做是有必要的，因为枚举集合为位域提供在性能方面有竞争力的替代方法，这是很重要的(见第32条)。

简而言之，在定义参数数目不定的方法时，可变参数方法是一种很方便的方式，但是它们不应该被过度滥用。如果使用不当，会产生混乱的结果。

更多资源请访问稀酷客(www.ckook.com)

第43条：返回零长度的数组或者集合，而不是null

像下面这样的方法并不少见：

```
private final List<Cheese> cheesesInStock = ...;

/**
 * @return an array containing all of the cheeses in the shop,
 *     or null if no cheeses are available for purchase.
 */
public Cheese[] getCheeses() {
    if (cheesesInStock.size() == 0)
        return null;
    ...
}
```

把没有奶酪（cheese）可买的情况当作是一种特例，这是不合常理的。这样做会要求客户端中必须有额外的代码来处理null返回值，例如：

```
Cheese[] cheeses = shop.getCheeses();
if (cheeses != null &&
    Arrays.asList(cheeses).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

而不是下面这段代码：

```
if (Arrays.asList(shop.getCheeses()).contains(Cheese.STILTON))
    System.out.println("Jolly good, just the thing.");
```

对于一个返回null而不是零长度数组或者集合的方法，几乎每次用到该方法时都需要这种曲折的处理方式。这样做很容易出错，因为编写客户端程序的程序员可能会忘记写这种专门的代码来处理null返回值。这样的错误也许几年都不会被注意到，因为这样的方法通常返回一个或者多个对象。返回null而不是零长度的数组也会使返回数组或者集合的方法本身变得更加复杂，这一点虽然不是特别重要，但是也值得注意。

有时候会有人认为：null返回值比零长度数组更好，因为它避免了分配数组所需要的开销。这种观点是站不住脚的，原因有两点。第一，在这个级别上担心性能问题是不明智的，除非分析表明这个方法正是造成性能问题的真正源头（见第55条）。第二，对于不返回任何元素的调用，每次都返回同一个零长度数组是有可能的，因为零长度数组是不可变的，而不可变对象有可能被自由地共享（见第15条）。实际上，当你使用标准做法（standard idiom）把一些元素从一个集合转存到一个类型化的数组（typed array）中时，它正是这样做的：

```
// The right way to return an array from a collection
private final List<Cheese> cheesesInStock = ...;

private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];
```

更多资源请访问稀酷客(www.ckook.com)

```
/**
 * @return an array containing all of the cheeses in the shop.
 */
public Cheese[] getCheeses() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

在这种习惯用法中，零长度数组常量被传递给toArray方法，以指明所期望的返回类型。正常情况下，toArray方法分配了返回的数组，但是，如果集合是空的，它将使用零长度的输入数组，Collection.toArray(T[])的规范保证：如果输入数组大到足够容纳这个集合，它就将返回这个输入数组。因此，这种做法永远也不会分配零长度的数组。

同样地，集合值的方法也可以做成在每当需要返回空集合时都返回同一个不可变的空集合。Collections.emptySet、emptyList和emptyMap方法提供的正是你所需要的，如下所示：

```
// The right way to return a copy of a collection
public List<Cheese> getCheeseList() {
  if (cheesesInStock.isEmpty())
    return Collections.emptyList(); // Always returns same list
  else
    return new ArrayList<Cheese>(cheesesInStock);
}
```

简而言之，返回类型为数组或集合的方法没理由返回**null**，而不是返回一个零长度的数组或者集合。这种习惯做法（指返回null）很有可能是从C程序设计语言中沿袭过来的，在C语言中，数组长度是与实际的数组分开返回的。在C语言中，如果返回的数组长度为零，再分配一个数组就没有任何好处。

更多资源请访问稀酷客(www.ckook.com)

第44条：为所有导出的API元素编写文档注释

如果要想使一个API真正可用，就必须为其编写文档。传统意义上的API文档是手工生成的，所以保持文档与代码同步是一件很繁琐的事情。Java语言环境提供了一种被称为Javadoc的实用工具，从而使这项任务变得很容易。Javadoc利用特殊格式的文档注释（**documentation comment**，通常被写作**doc comment**），根据源代码自动产生API文档。

如果你对文档注释的规范还不太熟悉，应该去了解这些规范。虽然这些规范还没有正式成为Java程序设计语言的一部分，但它们已经构成了每个程序员都应该知道的事实API。这些规范的内容在**Sun**公司关于如何编写文档注释（**How to Write Doc Comments**）的网页上进行了说明[Javadoc-guide]。虽然这个网页在Java 1.4发行版本之后还没有进行更新，但它仍然是个很有价值的资源。Java 1.5发行版本中，为Javadoc新增了两个重要的Javadoc标签：{@literal}和{@code}[Javadoc-5.0]。本条目中会讨论到这些标签。

为了正确地编写**API**文档，必须在每个被导出的类、接口、构造器、方法和域声明之前增加一个文档注释。如果类是可序列化的，也应该对它的序列化形式编写文档（见第75条）。如果没有文档注释，Javadoc所能够做的也就是重新生成该声明，作为受影响的API元素的唯一文档。使用没有文档注释的API是非常痛苦的，也很容易出错。为了编写出可维护的代码，还应该为那些没有被导出的类、接口、构造器、方法和域编写文档注释。

方法的文档注释应该简洁地描述出它和客户端之间的约定。除了专门为继承而设计的类中的方法（见第17条）之外，这个约定应该说明这个方法做了什么，而不是说明它是如何完成这项工作的。文档注释应该列举出这个方法的所有前提条件（**precondition**）和后置条件（**postcondition**），所谓前提条件是指为了使客户能够调用这个方法，而必须要满足的条件；所谓后置条件是指在调用成功完成之后，哪些条件必须要满足。一般情况下，前提条件是由@throws标签针对未受检的异常所隐含描述的；每个未受检的异常都对应一个前提违例（precondition violation）。同样地，也可以在一些受影响的参数的@param标记中指定前提条件。

除了前提条件和后置条件之外，每个方法还应该在文档中描述它的副作用（**side effect**）。所谓副作用是指系统状态中可以观察到的变化，它不是为了获得后置条件而明确要求的变化。例如，如果方法启动了后台线程，文档中就应该说明这一点。最后，文档注释也应该描述类或者方法的线程安全性（**thread safety**），正如第70条中所述。

为了完整地描述方法的约定，方法的文档注释应该让每个参数都有一个@param标签，以及一个@return标签（除非这个方法的返回类型为void），以及对于该方法抛出的每个异常，无论是受检的还是未受检的，都有一个@throws标签（见第62条）。按惯例，跟在@param标签

更多资源请访问稀酷客(www.ckook.com)

或者@return标签后面的文字应该是一个名词短语，描述了这个参数或者返回值所表示的值。跟在@throws标签之后的文字应该包含单词“if”（如果），紧接着是一个名词短语，它描述了这个异常将在什么样的条件下会被抛出。有时候，也会用算术表达式来代替名词短语。按惯例，@param、@return或者@throws标签后面的短语或者子句都不用句点来结束。下面这个简短的文档注释演示了所有这些习惯做法：

```
/**
 * Returns the element at the specified position in this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations it may run in time proportional
 * to the element position.
 *
 * @param  index index of element to return; must be
 *         non-negative and less than the size of this list
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index);
```

注意，这份文档注释中使用了HTML标签（<p>和<i>）。Javadoc工具会把文档注释翻译成HTML，文档注释中包含的任意HTML元素都会出现在结果HTML文档中。有时候，程序员会把HTML表格嵌入到它们的文档注释中，但是这种做法并不多见。

还要注意，@throws子句的代码片段中到处使用了Javadoc的{@code}标签。它有两个作用：造成该代码片段以代码字体（code font）进行呈现，并限制HTML标记和嵌套的Javadoc标签在代码片段中进行处理。后一种属性正是允许我们在代码片段中使用小于号（<）的东西，虽然它是一个HTML元字符。在Java 1.5发行版本之前，是通过使用HTML标签和HTML转义，将代码片段包含在文档注释中。现在再也没有必要在文档注释中使用**HTML <code>**或者**<tt>**标签了：**Javadoc {@code}**标签更好，因为它避免了转义**HTML**元字符。为了将多个代码示例包含在一个文档注释中，要使用包在HTML的<pre>标签里面的Javadoc {@code}标签。换句话说，是先在多行的代码示例前使用字符<pre>{@code，然后在代码后面加上}</pre>。

最后，要注意这个文档注释中用到了单词“this”。按惯例，当“this”被用在实例方法的文档注释中时，它应该始终是指方法调用所在的对象。

不要忘记，为了产生包含HTML元字符的文档，比如小于号（<）、大于号（>）以及“与”号（&），必须采取特殊的动作。让这些字符出现在文档中的最佳办法是用{@literal}标签将它们包围起来，这样就限制了HTML标记和嵌套的Javadoc标签的处理。除了它不以代码字体渲染文本之外，其余方面就像{@code}标签一样。例如，这个Javadoc片段：

```
 * The triangle inequality is {@literal |x + y| < |x| + |y|}.
```

更多资源请访问稀酷客(www.ckook.com)

产生了这样的文档："The triangle inequality is |x + y| < |x| + |y|."{@literal}标签也可以只是包住小于号，而不是整个不等式，所产生的文档是一样的，但是在源代码中见到的文档注释的可读性就会更差。这说明了一条通则：文档注释在源代码和产生的文档中都应该是易于阅读的。如果无法让两者都易读，产生的文档的可读性要优先于源代码的可读性。

每个文档注释的第一句话（如下所示）成了该注释所属元素的*概要描述*（**summary description**）。例如，第177页中文档注释中的概要描述为"返回这个列表中指定位置上的元素"。概要描述必须独立地描述目标元素的功能。为了避免混淆，*同一个类或者接口中的两个成员或者构造器，不应该具有同样的概要描述*。特别要注意重载的情形，在这种情况下，往往很自然地在描述中使用同样的第一句话（但在文档注释中这是不可接受的）。

注意所期待的概要描述中是否包括句点，因为句点会过早地终止这个描述。例如，一个以"A college degree, such as B.S., M.S., or Ph.D."开头的文档注释，会产生这样的概要描述："A college degree, such as B.S, M.S."问题在于，概要描述在后面接着空格、跳格或者行终结符的第一个句点处（或者在第一个块标签处）结束[Javadoc-ref]。在这种情况下，缩写"M.S."中的第二个句点就要接着用一个空格。最好的解决方法是，将讨厌的句点以及任何与{@literal}关联的文本都包起来，因此在源代码中，句点后面就不再是空格了：

```
/**
 * A college degree, such as B.S., {@literal M.S.} or Ph.D.
 * College is a fountain of knowledge where many go to drink.
 */
public class Degree { ... }
```

说概要描述是文档注释中的第一个*句子*（**sentence**），这似乎有点误导人。规范指出，概要描述很少是个完整的句子。对于方法和构造器而言，概要描述应该是个完整的动词短语（包含任何对象），它描述了该方法所执行的动作。例如：

- ArrayList(int initialCapacity)—Constructs an empty list with the specified initial capacity.（用指定的初始容量构造一个空的列表）
- Collection.size()—Returns the number of elements in this collection.（返回该集合中元素的数目）

对于类、接口和域，概要描述应该是一个名词短语，它描述了该类或者接口的实例，或者域本身所代表的事物。例如：

- TimerTask—A task that can be scheduled for one-time or repeated execution by a Timer.（可以调度一次的任务，或者被Timer重复执行的任务）

更多资源请访问稀酷客(www.ckook.com)

- Math.PI—The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.（非常接近于PI（圆周长度与直径的比值）的double值）

Java 1.5发行版本中增加的三个特性在文档注释中需要特别小心：泛型、枚举和注解。当为泛型或者方法编写文档时，确保要在文档中说明所有的类型参数。

```
/**
 * An object that maps keys to values.  A map cannot contain
 * duplicate keys; each key can map to at most one value.
 *
 * (Remainder omitted)
 *
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K, V> {
    ... // Remainder omitted
}
```

当为枚举类型编写文档时，要确保在文档中说明常量，以及类型，还有任何公有的方法。注意，如果文档注释很简短，可以将整个注释放在一行上：

```
/**
 * An instrument section of a symphony orchestra.
 */
public enum OrchestraSection {
    /** Woodwinds, such as flute, clarinet, and oboe. */
    WOODWIND,

    /** Brass instruments, such as french horn and trumpet. */
    BRASS,

    /** Percussion instruments, such as timpani and cymbals */
    PERCUSSION,

    /** Stringed instruments, such as violin and cello. */
    STRING;
}
```

为注解类型编写文档时，要确保在文档中说明所有成员，以及类型本身。带有名词短语的文档成员，就像是域一样。对于该类型的概要描述，要使用一个动词短语，说明当程序元素具有这种类型的注解时它表示什么意思。

```
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to succeed.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    /**
     * The exception that the annotated test method must throw
     * in order to pass. (The test is permitted to throw any
     * subtype of the type described by this class object.)
     */
```

更多资源请访问稀酷客(www.ckook.com)

```
    Class<? extends Exception> value();
}
```

从Java 1.5发行版本开始，包级私有的文档注释就应该放在一个称作package-info.java的文件中，而不是放在package.html中。除了包级私有的文档注释之外，package-info.java也可以（但并非必需）包含包声明和包注解。

类的导出API有两个特征经常被人忽视，即线程安全性和可序列化性。类是否是线程安全的，应该在文档中对它的线程安全级别进行说明，如第70条中所述。如果类是可序列化的，就应该在文档中说明它的序列化形式，如第75条中所述。

Javadoc具有“继承”方法注释的能力。如果API元素没有文档注释，Javadoc将会搜索最为适用的文档注释，接口的文档注释优先于超类的文档注释。搜索算法的细节可以在《*The Javadoc Reference Guide*》[Javadoc-ref]中找到。也可以利用{@inheritDoc}标签从超类型中继承文档注释的部分内容。这意味着，不说别的，类还可以重用它所实现的接口的文档注释，而不需要拷贝这些注释。这项机制有可能减轻维护多个几乎相同的文档注释的负担，但它使用起来比较需要一些小技巧（tricky），并具有一些局限性。关于这一点的详情超出了本书的范围，在此不做讨论。

为了降低文档注释中出错的可能性，一种简单的办法是通过一个**HTML有效性检查器**（**HTML validity checker**）来运行由Javadoc产生的HTML文件。这样可以检测出HTML标签的许多不正确用法，以及应该被转义的HTML元字符。Internet上有一些HTML有效性检查器可供下载，并且可以在线检验HTML[W3C-validator]。

关于文档注释有一点需要特别注意。虽然为所有导出的API元素提供文档注释是必要的，但是这样做并非永远就足够了。对于由多个相互关联的类组成的复杂API，通常有必要用一个外部文档来描述该API的总体结构，对文档注释进行补充。如果有这样的文档，相关的类或者包文档注释就应该包含一个对这个外部文档的链接。

本条目中所述的内容涵盖了基本的惯例。关于编写文档注解最权威的指导是Sun公司的《*How to Write Doc Comments*（如何编写文档注释）》[Javadoc-guide]。

简而言之，要为API编写文档，文档注释是最好、最有效的途径。对于所有可导出的API元素来说，使用文档注释应该被看作是强制性的。要采用一致的风格来遵循标准的约定。记住，在文档注释内部出现任何HTML标签都是允许的，但是HTML元字符必须要经过转义。

更多资源请访问稀酷客(www.ckook.com)

第8章

通用程序设计

本章主要讨论Java语言的具体细节，讨论了局部变量的处理、控制结构、类库的用法、各种数据类型的用法，以及两种不是由语言本身提供的机制（**reflection**和**native method**，反射机制和本地方法）的用法。最后讨论了优化和命名惯例。

第45条：将局部变量的作用域最小化

本条目与第13条（使类和成员的可访问性最小化）本质上是类似的。将局部变量的作用域最小化，可以增强代码的可读性和可维护性，并降低出错的可能性。

较早的程序设计语言（如C语言）要求局部变量必须在一个代码块的开头处进行声明，出于习惯，有些程序员们目前还是继续这样做。这个习惯应该改正。在此提醒，Java允许你在任何可以出现语句的地方声明变量。

要使局部变量的作用域最小化，最有力的方法就是在第一次使用它的地方声明。如果变量在使用之前进行声明，这只会造成混乱——对于试图理解程序功能的读者来说，这又多了一种只会分散他们注意力的因素。等到用到该变量的时候，读者可能已经记不起该变量的类型或者初始值了。

过早地声明局部变量不仅会使它的作用域过早地扩展，而且结束得也过于晚了。局部变量的作用域从它被声明的点开始扩展，一直到外围块（block）的结束处。如果变量是在“使用它的块”之外被声明的，当程序退出该块之后，该变量仍是可见的。如果变量在它的目标使用区域之前或者之后被意外地使用的话，后果将可能是灾难性的。

几乎每个局部变量的声明都应该包含一个初始化表达式。如果你还没有足够的信息来对一个变量进行有意义的初始化，就应该推迟这个声明，直到可以初始化为止。这条规则有个例外的情况与try-catch语句有关。如果一个变量被一个方法初始化，而这个方法可能会抛出一个

更多资源请访问稀酷客(www.ckook.com)

受检的异常（checked exception），该变量就必须在try块的内部被初始化。如果变量的值必须在try块的外部被使用到，它就必须在try块之前被声明，但是在try块之前，它还不能被“有意义地初始化”。请参照第202页中的例子。

循环中提供了特殊的机会来将变量的作用域最小化。（无论是传统的还是for-each形式的）for循环，都允许声明循环变量（**loop variable**），它们的作用域被限定在正好需要的范围之内。（这个范围包括循环体，以及循环体之前的初始化、测试、更新部分。）因此，如果在循环终止之后不再需要循环变量的内容，**for循环就优先于while循环**。

例如，下面是一种遍历集合的首选做法（见第46条）：

```
// Preferred idiom for iterating over a collection
for (Element e : c) {
    doSomething(e);
}
```

在Java 1.5发行版本之前，首选的做法如下（现在仍然有适用之处）：

```
// No for-each loop or generics before release 1.5
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething((Element) i.next());
}
```

为了弄清楚为什么这个for循环比while循环更好，请考虑下面的代码片断，它包含两个while循环，以及一个Bug：

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}
...

Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) {              // BUG!
    doSomethingElse(i2.next());
}
```

第二个循环中包含一个“剪切-粘贴”错误：它本来是要初始化一个新的循环变量i2，却使用了旧的循环变量i，遗憾的是，这时i仍然还在有效范围之内。结果代码仍然可以通过编译，运行的时候也不会抛出异常，但是它所做的事情却是错误的。第二个循环并没有在c2上迭代，而是立即终止，造成c2为空的假象。因为这个程序的错误是悄然发生的，所以可能在很长时间内都不会被发现。

如果类似的“剪切-粘贴”错误发生在前面任何一种for循环中，结果代码就根本不能通过编译。在第二个循环开始之前，第一个循环的元素（或者迭代器）变量已经不在它的作用域范围之内了：

更多资源请访问稀酷客(www.ckook.com)

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    doSomething(i.next());
}
    ...

// Compile-time error - cannot find symbol i
for (Iterator<Element> i2 = c2.iterator(); i.hasNext(); ) {
    doSomething(i2.next());
}
```

而且，如果使用for循环，犯这种“剪切-粘贴”错误的可能性就会大大降低，因为通常没有必要在两个循环中使用不同的变量名。循环是完全独立的，所以重用元素（或者迭代器）变量的名称不会有任何危害。实际上，这也是很流行的做法。

使用for循环与使用while循环相比还有另外一个优势：更简短，从而增强了可读性。

下面是另外一种对局部变量的作用域进行最小化的循环做法：

```
for (int i = 0, n = expensiveComputation(); i < n; i++) {
    doSomething(i);
}
```

关于这种做法要关注的一点是，它具有两个循环变量：i和n，两者具有完全相同的作用域。第二个变量n被用来保存第一个变量的极限值，从而避免在每次迭代中执行冗余计算的开销。通常，如果循环测试中涉及方法调用，它可以保证在每次迭代中都会返回同样的结果，就应该使用这种做法。

最后一种“将局部变量的作用域最小化”的方法是使方法小而集中。如果把两个操作（activity）合并到同一个方法中，与其中一个操作相关的局部变量就有可能会出现在执行另一个操作的代码范围之内。为了防止这种情况发生，只要把这个方法分成两个，每个方法各执行一个操作。

更多资源请访问稀酷客(www.ckook.com)

第46条：for-each循环优先于传统的for循环

在Java 1.5发行版本之前，对集合进行遍历的首选做法如下：

```
// No longer the preferred idiom to iterate over a collection!
for (Iterator i = c.iterator(); i.hasNext(); ) {
    doSomething((Element) i.next()); // (No generics before 1.5)
}
```

遍历数组的首选做法如下：

```
// No longer the preferred idiom to iterate over an array!
for (int i = 0; i < a.length; i++) {
    doSomething(a[i]);
}
```

这些做法都比while循环（见第45条）更好，但是它们也并不完美。迭代器和索引变量都会造成一些混乱。而且，它们也代表着出错的可能。迭代器和索引变量在每个循环中出现三次，其中有两次让你很容易出错。一旦出错，就无法保证编译器能够发现错误。

Java 1.5发行版本中引入的for-each循环，通过完全隐藏迭代器或者索引变量，避免了混乱和出错的可能。这种模式同样适用于集合和数组：

```
// The preferred idiom for iterating over collections and arrays
for (Element e : elements) {
    doSomething(e);
}
```

当见到冒号（:）时，可以把它读作“在…里面”。因此上面的循环可以读作“对于元素中的每个元素e。”注意，利用for-each循环不会有性能损失，甚至用于数组也一样。实际上，在某些情况下，比起普通的for循环，它还稍有些性能优势，因为它对数组索引的边界值只计算一次。虽然可以手工完成这项工作（见第45条），但程序员并不总会这么做。

在对多个集合进行嵌套式迭代时，for-each循环相对于传统for循环的这种优势还会更加明显。下面就是人们在试图对两个集合进行嵌套迭代时经常会犯的错误：

```
// Can you spot the bug?
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

更多资源请访问稀酷客(www.ckook.com)

如果之前没有发现这个Bug也不必难过。许多专家级的程序员偶尔也会犯这样的错误。问题在于，在迭代器上对外部的集合（suits）调用了太多次next方法了。它应该从外部的循环进行调用，以便每种花色调用一次，但它却是从内部循环调用，因此它是每张牌调用一次。在用完所有花色之后，循环就会抛出NoSuchElementException异常。

如果真的那么不幸，并且外部集合的大小是内部集合大小的几倍——可能因为它们是相同的集合——循环就会正常终止，但是不会完成你想要的工作。例如，下面是个考虑不周的尝试，要打印一对骰子的所有可能的滚法。

```
// Same bug, different symptom!
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = Arrays.asList(Face.values());

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

这个程序不会抛出异常，而是只打印6个重复的词（从“ONE ONE”到“SIX SIX”），而不是预计的36种组合。

为了修正这些示例中的Bug，必须在外部循环的作用域中添加一个变量来保存外部元素：

```
// Fixed, but ugly - you can do better!
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); ) {
    Suit suit = i.next();
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(suit, j.next()));
}
```

如果使用的是嵌套的for-each循环，这个问题就会完全消失。产生的代码就如你所希望得那样简洁。

```
// Preferred idiom for nested iteration on collections and arrays
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

for-each循环不仅让你遍历集合和数组，还让你遍历任何实现Iterable接口的对象。这个简单的接口由单个方法组成，与for-each循环同时被增加到Java平台中。下面就是这个接口的示例：

```
public interface Iterable<E> {
    // Returns an iterator over the elements in this iterable
    Iterator<E> iterator();
}
```

实现Iterable接口并不难。如果你在编写的类型表示的是一组元素，即使你选择不让它实现Collection，也要让它实现Iterable。这样可以允许用户利用for-each循环遍历你的类型，会令用户永远感激不尽的。

更多资源请访问稀酷客(www.ckook.com)

总之，for-each循环在简洁性和预防Bug方面有着传统的for循环无法比拟的优势，并且没有性能损失。应该尽可能地使用for-each循环。遗憾的是，有三种常见的情况无法使用for-each循环：

1. **过滤**——如果需要遍历集合，并删除选定的元素，就需要使用显式的迭代器，以便可以调用它的remove方法。

2. **转换**——如果需要遍历列表或者数组，并取代它部分或者全部的元素值，就需要列表迭代器或者数组索引，以便设定元素的值。

3. **平行迭代**——如果需要并行地遍历多个集合，就需要显式地控制迭代器或者索引变量，以便所有迭代器或者索引变量都可以得到同步前移（就如上述关于有问题的牌和骰子的示例中无意中所示范的那样）。

在以上任何一种情况下，就要使用普通的for循环，要警惕本条目中提到的陷阱，并且要确保做到最好。

更多资源请访问稀酷客(www.ckook.com)

第47条：了解和使用类库

假设你希望产生位于0和某个上界之间的随机整数。面对这个常见的任务，许多程序员会编写出如下所示的方法：

```
private static final Random rnd = new Random();

// Common but deeply flawed!
static int random(int n) {
    return Math.abs(rnd.nextInt()) % n;
}
```

这个方法看起来可能不错，但是却有三个缺点。第一个缺点是，如果n是一个比较小的2的乘方，经过一段相当短的周期之后，它产生的随机数序列将会重复。第二个缺点是，如果n不是2的乘方，那么平均起来，有些数会比其他的数出现得更为频繁。如果n比较大，这个缺点就会非常明显。这可以通过下面的程序直观地体现出来，它会产生一百万个经过细心指定的范围内的随机数，并打印出有多少个数字落在随机数取值范围的前半部分：

```
public static void main(String[] args) {
    int n = 2 * (Integer.MAX_VALUE / 3);
    int low = 0;
    for (int i = 0; i < 1000000; i++)
        if (random(n) < n/2)
            low++;

    System.out.println(low);
}
```

如果random方法工作正常的话，这个程序打印出来的数将接近于一百万的一半，但是如果真正运行这个程序，就会发现它打印出来的数接近于666 666。由random方法产生的数字有2/3落在随机数取值范围的前半部分。

random方法的第三个缺点是，在极少数情况下，它的失败是灾难性的，返回一个落在指定范围之外的数。之所以如此，是因为这个方法试图通过调用Math.abs，将rnd.nextInt()返回的值映射为一个非负整数int。如果nextInt()返回Integer.MIN_VALUE，那么Math.abs也会返回Integer.MIN_VALUE，假设n不是2的乘方，那么取模操作符（%）将返回一个负数。这几乎肯定会使程序失败，而且这种失败很难重现。

为了编写能修正这三个缺点的random方法，有必要了解关于伪随机数生成器、数论和2的求补算法的相关知识。幸运的是，你并不需要自己来做这些工作——已经有现成的成果可以为你所用。它被称为Random.nextInt(int)，自Java 1.2发行版本以来，它已经成了Java平台的一部分。

你无需关心nextInt(int)的实现细节（如果你有强烈的好奇心，可以研究它的文档或者源代

更多资源请访问稀酷客(www.ckook.com)

码)。具有算法背景的高级工程师已经花了大量的时间来设计、实现和测试这个方法，然后经过这个领域中的专家的审查，以确保它的正确性。然后，标准类库经过了Beta测试、发行和近十年的成千上万程序员的广泛使用。在这个方法中还没有发现过缺陷，但是，如果将来发现有缺陷，在下一个发行版本中就会修正这些缺陷。通过使用标准类库，可以充分利用这些编写标准类库的专家的知识，以及在你之前的其他人的使用经验。

使用标准类库的第二个好处是，不必浪费时间为那些与工作不太相关的问题提供特别的解决方案。就像大多数程序员一样，应该把时间花在应用程序上，而不是底层的细节上。

使用标准类库的第三个好处是，它们的性能往往会随着时间的推移而不断提高，无需你做任何努力。因为许多人在使用它们，被当作工业标准在使用，所以，提供这些标准类库的组织有强烈的动机要使它们运行得更快。这些年来，许多Java平台类库已经被重新编写了，有时候是重复编写，从而导致性能上有了显著的提高。

标准类库也会随着时间的推移而增加新的功能。如果类库中漏掉了某些功能，开发者社区(developer community)就会把这些缺点告示出来，漏掉的功能就会添加到后续的发行版本中。Java平台类库始终是在这个社区的推动下不断发展的。

使用标准类库的最后一个好处是，可以使自己的代码融入主流。这样的代码更易读、更易维护、更易被大多数的开发人员重用。

既然有那么多的优点，使用标准类库机制而不选择专门的实现，这显然是符合逻辑的，然而还是有相当一部分的程序员没有这样做。为什么呢？可能他们并不知道有这些类库机制的存在。在每个重要的发行版本中，都会有许多新的特性被加入到类库中，所以与这些新特性保持同步是值得的。每次Java平台有重要的发行时，Sun公司都会发布一个网页，说明新的特性。这些网页值得好好读一读[Java5-feat，Java6-feat]。这些标准类库太庞大了，以至于不可能去学习所有的文档[JavaSE6]，但是每个程序员都应该熟悉**java.lang**、**java.util**，某种程度上还有**java.io**中的内容。关于其他类库的知识可以根据需要随时学习。

本条目不可能总结类库中所有的便利工具，但是有两种工具值得特别一提。在1.2发行版本中，**Collections Framework**（集合框架）被加入到了java.util包中。它应该成为每个程序员基本工具箱中的一部分。Collections Framework是一个统一的体系结构，用来表示和操作集合，允许它们对集合进行独立于表示细节的操作。它减轻了编程的负担，同时还提升了性能。它考虑到不相关的API之间的互操作性，减少了为设计和学习新的API所要付出的努力，并且鼓励软件重用。如果想要了解更多这方面的细节，请参见Sun公司网站上的文章[Collections]，或者阅读有关的教程[Bloch06]。

1.5发行版本中，在**java.util.concurrent**包中增加了一组并发实用工具。这个包既包含高

更多资源请访问稀酷客(www.ckook.com)

级的并发工具来简化多线程的编程任务，还包含低级别的并发基本类型，允许专家们自己编写更高级的并发抽象。java.util.concurrent的高级部分，也应该是每个程序员基本工具箱中的一部分（见第68条和第69条）。

在有些情况下，一个类库工具并不能满足你的需要。你的需求越是特殊，这种情形就越有可能发生。虽然你的第一个念头应该是使用标准类库，但是，如果你在观察了它们在某些领域所提供的功能之后，确定它不能满足需要，你就得使用其他的实现。任何一组类库所提供的功能总是难免会有遗漏。如果你所需要的功能不存在，那么，就只能自己实现这些功能，别无选择。

总而言之，不要重新发明轮子。如果你要做的事情看起来是十分常见的，有可能类库中已经有某个类完成了这样的工作。如果确实是这样，就使用现成的；如果还不清楚是否存在这样的类，就去查一查。一般而言，类库的代码可能比你自己编写的代码更好一些，并且会随着时间的推移而不断改进。这并不是在影射你作为一个程序员的能力。从经济角度的分析表明：类库代码受到的关注远远超过大多数普通程序员在同样的功能上所能够给予的投入。

更多资源请访问稀酷客(www.ckook.com)

第48条：如果需要精确的答案，请避免使用float和double

float和double类型主要是为了科学计算和工程计算而设计的。它们执行二进制浮点运算(**binary floating-point arithmetic**)，这是为了在广泛的数值范围上提供较为精确的快速近似计算而精心设计的。然而，它们并没有提供完全精确的结果，所以不应该被用于需要精确结果的场合。**float和double类型尤其不适合用于货币计算**，因为要让一个float或者double精确地表示0.1（或者10的任何其他负数次方值）是不可能的。

例如，假设你的口袋中有$1.03，花掉了42¢之后还剩下多少钱呢？下面是一个很简单的程序片断，要回答这个问题：

```
System.out.println(1.03 - .42);
```

遗憾的是，它输出的结果是0.6100000000000001。这并不是个别的例子。假设你的口袋里有$1，你买了9个垫圈，每个为10¢。那么你应该找回多少零头呢？

```
System.out.println(1.00 - 9 * .10);
```

根据这个程序片断，你得到的是$0.09999999999999998。

你可能会认为，只要在打印之前将结果做一下舍入就可以解决这个问题，但遗憾的是，这种做法并不总是可行。例如，假设你的口袋里有$1，你看到货架上有一排美味的糖果，标价分别为10¢、20¢、30¢，等等，一直到$1。你打算从标价为10¢的糖果开始，每种买1颗，一直到不能支付货架上下一种价格的糖果为止，那么你可以买多少颗糖果？还会找回多少零头？下面是一个简单的程序，用来解决这个问题：

```
// Broken - uses floating point for monetary calculation!
public static void main(String[] args) {
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = .10; funds >= price; price += .10) {
        funds -= price;
        itemsBought++;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Change: $" + funds);
}
```

如果真正运行这个程序，你会发现你可以支付3颗糖果，并且还剩下$0.3999999999999999。这个答案是不正确的！解决这个问题的正确办法是使用**BigDecimal**、**int**或者**long**进行货币计算。

下面的程序是上一个程序的简单翻版，它使用BigDecimal类型代替double：

更多资源请访问稀酷客(www.ckook.com)

```
public static void main(String[] args) {
    final BigDecimal TEN_CENTS = new BigDecimal( ".10");

    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
    for (BigDecimal price = TEN_CENTS;
            funds.compareTo(price) >= 0;
            price = price.add(TEN_CENTS)) {
        itemsBought++;
        funds = funds.subtract(price);
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: $" + funds);
}
```

如果运行这个修改过的程序，就会发现你可以支付4颗糖果，还剩下$0.00。这才是正确的答案。

然而，使用BigDecimal有两个缺点：与使用基本运算类型相比，这样做很不方便，而且很慢。对于解决这样一个简单的问题，后一种缺点并不要紧，但是前一种缺点可能会让你很不舒服。

除了使用BigDecimal之外，还有一种办法是使用int或者long，到底选用int或者long要取决于所涉及数值的大小，同时要自己处理十进制小数点。在这个示例中，最明显的做法是以分为单位进行计算，而不是以元为单位。下面是这个例子的简单翻版，展示了这种做法：

```
public static void main(String[] args) {
    int itemsBought = 0;
    int funds = 100;
    for (int price = 10; funds >= price; price += 10) {
        itemsBought++;
        funds -= price;
    }
    System.out.println(itemsBought + " items bought.");
    System.out.println("Money left over: "+ funds + " cents");
}
```

总而言之，对于任何需要精确答案的计算任务，请不要使用float或者double。如果你想让系统来记录十进制小数点，并且不介意因为不使用基本类型而带来的不便，就请使用BigDecimal。使用BigDecimal还有一些额外的好处，它允许你完全控制舍入，每当一个操作涉及舍入的时候，它允许你从8种舍入模式中选择其一。如果你正通过法定要求的舍入行为进行业务计算，使用BigDecimal是非常方便的。如果性能非常关键，并且你又不介意自己记录十进制小数点，而且所涉及的数值又不太大，就可以使用int或者long。如果数值范围没有超过9位十进制数字，就可以使用int；如果不超过18位数字，就可以使用long。如果数值可能超过18位数字，就必须使用BigDecimal。

更多资源请访问稀酷客(www.ckook.com)

第49条：基本类型优先于装箱基本类型

Java有一个类型系统由两部分组成，包含基本类型（**primitive**），如int、double和boolean，和引用类型（**reference type**），如String和List。每个基本类型都有一个对应的引用类型，称作装箱基本类型（**boxed primitive**）。装箱基本类型中对应于int、double和boolean的是Integer、Double和Boolean。

Java 1.5发行版本中增加了自动装箱（**autoboxing**）和自动拆箱（**auto-unboxing**）。如第5条所述，这些特性模糊了但并没有完全抹去基本类型和装箱基本类型之间的区别。这两种类型之间真正是有差别的，要很清楚在使用的是哪种类型，并且要对这两种类型进行谨慎的选择，这些都非常重要。

在基本类型和装箱基本类型之间有三个主要区别。第一，基本类型只有值，而装箱基本类型则具有与它们的值不同的同一性。换句话说，两个装箱基本类型可以具有相同的值和不同的同一性。第二，基本类型只有功能完备的值，而每个装箱基本类型除了它对应基本类型的所有功能值之外，还有个非功能值：null。最后一点区别是，基本类型通常比装箱基本类型更节省时间和空间。如果不小心，这三点区别都会让你陷入麻烦之中。

考虑下面这个比较器，它被设计用来表示Integer值的递增数字顺序。（回想一下，比较器的compare方法返回的数值到底为负数、零还是正数，要取决于它的第一个参数是小于、等于还是大于它的第二个参数。）在实践中并不需要你编写这个在Integer中实现自然顺序的比较器，因为这是不需要比较器就可以得到的，但它展示了一个值得关注的例子：

```
// Broken comparator - can you spot the flaw?
Comparator<Integer> naturalOrder = new Comparator<Integer>() {
    public int compare(Integer first, Integer second) {
        return first < second ? -1 : (first == second ? 0 : 1);
    }
};
```

这个比较器表面看起来似乎不错，它可以通过许多测试。例如，它可以通过Collections.sort正确地给一个有一百万个元素的列表进行排序，无论这个列表中是否包含重复的元素。但是这个比较器有着严重的缺陷。如果你要让自己信服，只要打印naturalOrder.Compare(new Integer(42)和new Integer(42))的值。这两个Integer实例都表示相同的值(42)，因此这个表达式的值应该为0，但它输出的却是1，这表明第一个Integer值大于第二个。

问题出在哪呢？naturalOrder中的第一个测试工作得很好。对表达式first < second执行计算会导致被first和second引用的Integer实例被自动拆箱（**auto-unboxed**）；也就是说，它提取了它们的基本类型值。计算动作要检查产生的第一个int值是否小于第二个。但是假设答案

更多资源请访问稀酷客(www.ckook.com)

是否定的。下一个测试就是执行计算表达式first == second，它在两个对象引用上执行同一性比较（**identity comparison**）。如果first和second引用表示同一个int值的不同的Integer实例，这个比较操作就会返回false，比较器会错误地返回1，表示第一个Integer值大于第二个。对装箱基本类型运用==操作符几乎总是错误的。

修正这个问题最清楚的做法是添加两个局部变量，来保存对应于first和second的基本类型int值，并在这些变量上执行所有的比较操作。这样可以避免大量的同一性比较：

```
Comparator<Integer> naturalOrder = new Comparator<Integer>() {
    public int compare(Integer first, Integer second) {
        int f = first;   // Auto-unboxing
        int s = second;  // Auto-unboxing
        return f < s ? -1 : (f == s ? 0 : 1); // No unboxing
    }
};
```

接下来，考虑这个小程序：

```
public class Unbelievable {
    static Integer i;

    public static void main(String[] args) {
        if (i == 42)
            System.out.println("Unbelievable");
    }
}
```

它不是打印出Unbelievable——但是它的行为也是很奇怪的。它在计算表达式（i == 42）的时候抛出NullPointerException异常。问题在于，i是个Integer，而不是int，就像所有的对象引用域一样，它的初始值为null。当程序计算表达式(i == 42)时，它会将Integer与int进行比较。几乎在任何一种情况下，当在一项操作中混合使用基本类型和装箱基本类型时，装箱基本类型就会自动拆箱，这种情况无一例外。如果null对象引用被自动拆箱，就会得到一个NullPointerException异常。就如这个程序所示，它几乎可以在任何位置发生。修正这个问题很简单，声明i是个int而不是Integer就可以了。

最后，考虑第5条中的这个程序：

```
// Hideously slow program! Can you spot the object creation?
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

这个程序运行起来比预计的要慢一些，因为它不小心将一个局部变量（sum）声明为是装箱基本类型Long，而不是基本类型long。程序编译起来没有错误或者警告，变量被反复地装

更多资源请访问稀酷客(www.ckook.com)

箱和拆箱，导致明显的性能下降。

在本条目中所讨论的这三个程序中，问题是一样的：程序员忽略了基本类型和装箱基本类型之间的区别，并尝到了苦头。在前两个程序中，其结果是彻底的失败；在第三个程序中，则有严重的性能问题。

那么什么时候应该使用装箱基本类型呢？它们有几个合理的用处。第一个是作为集合中的元素、键和值。你不能将基本类型放在集合中，因此必须使用装箱基本类型。这是一种更通用的特例。在参数化类型（见第5章）中，必须使用装箱基本类型作为类型参数，因为Java不允许使用基本类型。例如，你不能将变量声明为ThreadLocal<int>类型，因此必须使用ThreadLocal<Integer>代替。最后，在进行反射的方法调用（见第53条）时，必须使用装箱基本类型。

总之，当可以选择的时候，基本类型要优先于装箱基本类型。基本类型更加简单，也更加快速。如果必须使用装箱基本类型，要特别小心！自动装箱减少了使用装箱基本类型的繁琐性，但是并没有减少它的风险。当程序用==操作符比较两个装箱基本类型时，它做了个同一性比较，这几乎肯定不是你所希望的。当程序进行涉及装箱和拆箱基本类型的混合类型计算时，它会进行拆箱，当程序进行拆箱时，会抛出NullPointerException异常。最后，当程序装箱了基本类型值时，会导致高开销和不必要的对象创建。

更多资源请访问稀酷客(www.ckook.com)

第50条：如果其他类型更适合，则尽量避免使用字符串

字符串被用来表示文本，它在这方面也确实做得很好。因为字符串很通用，并且Java语言也支持得很好，所以自然就会有这样一种倾向：即使在不适合使用字符串的场合，人们往往也会使用字符串。本条目就是讨论一些不应该使用字符串的情形。

字符串不适合代替其他的值类型。当一段数据从文件、网络，或者键盘设备，进入到程序中之后，它通常以字符串的形式存在。有一种自然的倾向是让它继续保留这种形式，但是，只有当这段数据本质上确实是文本信息时，这种想法才是合理的。如果它是数值，就应该被转换为适当的数值类型，比如int、float或者BigInteger类型。如果它是一个“是－或－否”这种问题的答案，就应该被转换为boolean类型。如果存在适当的值类型，不管是基本类型，还是对象引用，大多应该使用这种类型；如果不存在这样的类型，就应该编写一个类型。虽然这条建议是显而易见的，但却经常遭到违反。

字符串不适合代替枚举类型。正如第30条中所讨论的，枚举类型比字符串更加适合用来表示枚举类型的常量。

字符串不适合代替聚集类型。如果一个实体有多个组件，用一个字符串来表示这个实体通常是很不恰当的。例如，下面这行代码来自于真实的系统——标识符的名称已经被修改了，以免发生纠纷：

```
// Inappropriate use of string as aggregate type
String compoundKey = className + "#" + i.next();
```

这种方法有许多缺点。如果用来分隔域的字符也出现在某个域中，结果就会出现混乱。为了访问单独的域，必须解析该字符串，这个过程非常慢，也很繁琐，还容易出错。你无法提供equals、toString或者compareTo方法，只好被迫接受String提供的行为。更好的做法是，简单地编写一个类来描述这个数据集，通常是一个私有的静态成员类（见第22条）。

字符串也不适合代替能力表（capabilities）。有时候，字符串被用于对某种功能进行授权访问。例如，考虑设计一个提供线程局部变量（thread-local variable）的机制。这个机制提供的变量在每个线程中都有自己的值。自从Java 1.2发行版本以来，Java类库就有提供线程局部变量的机制，但在那之前，程序员必须自己完成。几年前面对这样的设计任务时，有些人自己提出了同样的设计方案：利用客户提供的字符串键，对每个线程局部变量的内容进行访问授权：

```
// Broken - inappropriate use of string as capability!
public class ThreadLocal {
    private ThreadLocal() { } // Noninstantiable
```

更多资源请访问稀酷客(www.ckook.com)

```
    // Sets the current thread's value for the named variable.
    public static void set(String key, Object value);

    // Returns the current thread's value for the named variable.
    public static Object get(String key);
}
```

这种方法的问题在于，这些字符串键代表了一个共享的全局命名空间。要使这种方法可行，客户端提供的字符串键必须是唯一的：如果两个客户端各自决定为它们的线程局部变量使用同样的名称，它们实际上就无意中共享了这个变量，这样往往会导致两个客户端都失败。而且，安全性也很差。恶意的客户端可能有意地使用与另一个客户端相同的键，以便非法地访问其他客户端的数据。

要修正这个API并不难，只要用一个不可伪造的键（unforgeable key，有时被称为能力（**capability**））来代替字符串即可：

```
public class ThreadLocal {
    private ThreadLocal() { }  // Noninstantiable

    public static class Key {  // (Capability)
        Key() { }
    }

    // Generates a unique, unforgeable key
    public static Key getKey() {
        return new Key();
    }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

虽然这解决了基于字符串的API的两个问题，但是你还可以做得更好。你实际上不再需要静态方法，它们可以被代之以键（Key）中的实例方法，这样这个键就不再是键，而是线程局部变量了。此时，这个不可被实例化的顶层类也不再做任何实质性的工作，因此可以删除这个顶层类，并将内层的嵌套类命名为ThreadLocal：

```
public final class ThreadLocal {
    public ThreadLocal() { }
    public void set(Object value);
    public Object get();
}
```

这个API不是类型安全的，因为当你从线程局部变量得到它时，必须将值从Object转换成它实际的值。不可能使原始的基于String的API为类型安全的，要使基于Key的API为类型安全的也很困难，但是，通过将ThreadLocal类泛型化（见第26条），使这个API变成类型安全的就是很简单的事情了：

```
public final class ThreadLocal<T> {
```

更多资源请访问稀酷客(www.ckook.com)

```
    public ThreadLocal() { }
    public void set(T value);
    public T get();
}
```

粗略地讲，这正是java.util.ThreadLocal提供的API。除了解决了基于字符串的API的问题之外，与前面的两个基于键的API相比，它还更快速、更优雅。

总而言之，如果可以使用更加合适的数据类型，或者可以编写更加适当的数据类型，就应该避免用字符串来表示对象。若使用不当，字符串会比其他的类型更加笨拙、更不灵活、速度更慢，也更容易出错。经常被错误地用字符串来代替的类型包括基本类型、枚举类型和聚集类型。

更多资源请访问稀酷客(www.ckook.com)

第51条：当心字符串连接的性能

字符串连接操作符（+, string concatenation operator）是把多个字符串合并为一个字符串的便利途径。要想产生单独一行的输出，或者构造一个字符串来表示一个较小的、大小固定的对象，使用连接操作符是非常合适的，但是它不适合运用在大规模的场景中。为连接n个字符串而重复地使用字符串连接操作符，需要n的平方级的时间。这是由于字符串不可变（见第15条）而导致的不幸结果。当两个字符串被连接在一起时，它们的内容都要被拷贝。

例如，考虑下面的方法，它通过反复连接每个项目行，构造出一个代表该对账单的字符串。代码如下：

```
// Inappropriate use of string concatenation - Performs horribly!
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i);  // String concatenation
    return result;
}
```

如果项目数量巨大，这个方法的执行时间就难以估算。为了获得可以接受的性能，请使用**StringBuilder**替代**String**，来存储建造中的对账单。（Java 1.5发行版本中增加了非同步StringBuilder类，代替了现在已经过时的StringBuffer类。）：

```
public String statement() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
        b.append(lineForItem(i));
    return b.toString();
}
```

上述两种做法的性能差别非常大。如果numItems返回100，并且lineForItem返回一个固定长度为80个字符的字符串，在我的机器上，第二种做法比第一种做法要快85倍。因为第一种做法的开销随项目数量而呈平方级增加，第二种做法则是线性增加，所以，项目数越大，性能的差别会越显著。注意，第二种做法预先分配了一个StringBuilder，使它大到足以容纳结果字符串。即使因为预先不知道字符串长度，使用了默认大小的StringBuilder，它仍然比第一种做法快50倍。

原则很简单：不要使用字符串连接操作符来合并多个字符串，除非性能无关紧要。相反，应该使用StringBuilder的append方法。另一种方法是，使用字符数组，或者每次只处理一个字符串，而不是将它们组合起来。

更多资源请访问稀酷客(www.ckook.com)

第52条：通过接口引用对象

第40条中有一个建议：应该使用接口而不是用类作为参数的类型。更一般地讲，应该优先使用接口而不是类来引用对象。如果有合适的接口类型存在，那么对于参数、返回值、变量和域来说，就都应该使用接口类型进行声明。只有当你利用构造器创建某个对象的时候，才真正需要引用这个对象的类。为了更具体地说明这一点，我们来考虑Vector的情形，它是List接口的一个实现。在声明变量的时候应该养成这样的习惯：

```
// Good - uses interface as type
List<Subscriber> subscribers = new Vector<Subscriber>();
```

而不是像这样的声明：

```
// Bad - uses class as type!
Vector<Subscriber> subscribers = new Vector<Subscriber>();
```

如果你养成了用接口作为类型的习惯，你的程序将会更加灵活。当你决定更换实现时，所要做的就只是改变构造器中类的名称（或者使用一个不同的静态工厂）。例如，第一个声明可以被改变为：

```
List<Subscriber> subscribers = new ArrayList<Subscriber>();
```

周围的所有代码都可以继续工作。周围的代码并不知道原来的实现类型，所以它们对于这种变化并不在意。

有一点值得注意：如果原来的实现提供了某种特殊的功能，而这种功能并不是这个接口的通用约定所要求的，并且周围的代码又依赖于这种功能，那么很关键的一点是，新的实现也要提供同样的功能。例如，如果第一个声明周围的代码依赖于Vector的同步策略，在声明中用ArrayList代替Vector就是不正确的。如果依赖于实现的任何特殊属性，就要在声明变量的地方给这些需求建立相应的文档说明。

那么，为什么要改变实现呢？因为新的实现提供了更好的性能，或者因为它提供了期望得到的额外功能。有个真实的例子与ThreadLocal类有关。在内部，这个类在Thread中使用了一个包级私有的Map域，将每个线程的值（per-thread values）与ThreadLocal实例关联起来。在1.3发行版本中，这个域被初始化为HashMap实例。在1.4发行版本中，Java平台增加了一个新的、被称为IdentityHashMap的专用Map实现。只需将初始化域的那一行代码改变为IdentityHashMap，代替原来的HashMap，ThreadLocal机制就会变快许多。ThreadLocal实现曾经一度发展为利用一个没有实现Map接口的高度优化过的存储结构，但是即使这样仍然不影响“通过接口引用对象”的这个观点。

更多资源请访问稀酷客(www.ckook.com)

如果把这个域声明为HashMap而不是Map，则无法保证只改变一行代码就足够了。如果客户端代码已经在Map接口之外使用了HashMap操作，或者把这个映射（Map）传递给了一个需要HashMap的方法，那么，若将该域改变为一个IdentityHashMap，代码就不再能通过编译。用接口类型声明域“让你保持诚实”。

如果没有合适的接口存在，完全可以用类而不是接口来引用对象。例如，考虑值类（**value class**），比如String和BigInteger。记住，值类很少会用多个实现编写。它们通常是final的，并且很少有对应的接口。使用这种值类作为参数、变量、域或者返回类型是再合适不过的了。更一般地讲，如果具体类没有相关联的接口，不管它是否表示一个值，你都没有别的选择，只有通过它的类来引用它的对象。Random类就属于这种情形。

不存在适当接口类型的第二种情形是，对象属于一个框架，而框架的基本类型是类，不是接口。如果对象属于这种基于类的框架（**class-based framework**），就应该用相关的基类（**base class**）（往往是抽象类）来引用这个对象，而不是用它的实现类。java.util.TimerTask抽象类就属于这种情形。

不存在适当接口类型的最后一种情形是，类实现了接口，但是它提供了接口中不存在的额外方法——例如LinkedHashMap。如果程序依赖于这些额外的方法，这种类就应该只被用来引用它的实例。它很少应该被用作参数类型（见第40条）。

以上这些例子并不全面，而只是代表了一些“适合于用类来引用对象”的情形。实际上，给定的对象是否具有适当的接口应该是很显然的。如果是，用接口引用对象就会使程序更加灵活；如果不是，则使用类层次结构中提供了必要功能的最基础的类。

更多资源请访问稀酷客(www.ckook.com)

第53条：接口优先于反射机制

核心反射机制（**core reflection facility**）java.lang.reflect，提供了“通过程序来访问关于已装载的类的信息”的能力。给定一个Class实例，你可以获得Constructor、Method和Field实例，分别代表了该Class实例所表示的类的Constructor（构造器）、Method（方法）和Field（域）。这些对象提供了“通过程序来访问类的成员名称、域类型、方法签名等信息”的能力。

而且，Constructor、Method和Field实例使你能够通过反射机制操作它们的底层对等体：通过调用Constructor、Method和Field实例上的方法，可以构造底层类的实例、调用底层类的方法，并访问底层类中的域。例如，Method.invoke使你可以调用任何类的任何对象上的任何方法（遵从常规的安全限制）。反射机制（reflection）允许一个类使用另一个类，即使当前者被编译的时候后者还根本不存在。然而，这种能力也要付出代价：

- **丧失了编译时类型检查的好处**，包括异常检查。如果程序企图用反射方式调用不存在的或者不可访问的方法，在运行时它将会失败，除非采取了特别的预防措施。
- **执行反射访问所需要的代码非常笨拙和冗长**。编写这样的代码非常乏味，阅读起来也很困难。
- **性能损失**。反射方法调用比普通方法调用慢了许多。具体慢了多少，这很难说，因为受到了多个因素的影响。在我的机器上，速度的差异可能小到2倍，也可能大到50倍。

核心反射机制最初是为了基于组件的应用创建工具而设计的。这类工具通常要根据需要装载类，并且用反射功能找出它们支持哪些方法和构造器。这些工具允许用户交互式地构建出访问这些类的应用程序，但是所产生出来的这些应用程序能够以正常的方式访问这些类，而不是以反射的方式。反射功能只是在设计时（**design time**）被用到。通常，普通应用程序在运行时不应该以反射方式访问对象。

有一些复杂的应用程序需要使用反射机制。这些示例中包括类浏览器、对象监视器、代码分析工具、解释型的内嵌式系统。在RPC（远程过程调用）系统中使用反射机制也是非常合适的，这样可以不再需要存根编译器（stub compiler）。如果你对自己的应用程序是否也属于这一类应用程序而感到怀疑，它很有可能就不属于这一类。

如果只是以非常有限的形式使用反射机制，虽然也要付出少许代价，但是可以获得许多好处。对于有些程序，它们必须用到在编译时无法获取的类，但是在编译时存在适当的接口或者超类，通过它们可以引用这个类（见第52条）。如果是这种情况，就可以以反射方式创建实例，然后通过它们的接口或者超类，以正常的方式访问这些实例。如果适当的构造器不带参数，

更多资源请访问稀酷客(www.ckook.com)

甚至根本不需要使用java.lang .reflect包；Class.newInstance方法就已经提供了所需的功能。

例如，下面的程序创建了一个Set<String>实例，它的类是由第一个命令行参数指定的。该程序把其余的命令行参数插入到这个集合中，然后打印该集合。不管第一个参数是什么，程序都会打印出余下的命令行参数，其中重复的参数会被消除掉。这些参数的打印顺序取决于第一个参数中指定的类。如果指定“java.util.HashSet”，显然这些参数就会以随机的顺序打印出来；如果指定“java.util.TreeSet”，则它们就会按照字母顺序打印出来，因为TreeSet中的元素是排好序的。相应的代码如下：

```
// Reflective instantiation with interface access
public static void main(String[] args) {
    // Translate the class name into a Class object
    Class<?> cl = null;
    try {
        cl = Class.forName(args[0]);
    } catch(ClassNotFoundException e) {
        System.err.println("Class not found.");
        System.exit(1);
    }

    // Instantiate the class
    Set<String> s = null;
    try {
        s = (Set<String>) cl.newInstance();
    } catch(IllegalAccessException e) {
        System.err.println("Class not accessible.");
        System.exit(1);
    } catch(InstantiationException e) {
        System.err.println("Class not instantiable.");
        System.exit(1);
    }

    // Exercise the set
    s.addAll(Arrays.asList(args).subList(1, args.length));
    System.out.println(s);
}
```

尽管这个程序就像一个“玩偶”，但是它所演示的这种方法是非常强大的。这个玩偶程序可以很容易地变成一个通用的集合测试器，通过侵入式地操作一个或者多个集合实例，并检查是否遵守Set接口的约定，以此来验证指定的Set实现。同样地，它也可以变成一个通用的集合性能分析工具。实际上，它所演示的这种方法足以实现一个成熟的服务提供者框架（**service provider framework**）（见第1条）。绝大多数情况下，使用反射机制时需要的也正是这种方法。

这个示例演示了反射机制的两个缺点。第一，这个例子会产生3个运行时错误，如果不使用反射方式的实例化，这3个错误都会成为编译时错误。第二，根据类名生成它的实例需要20行冗长的代码，而调用一个构造器可以非常简洁地只使用一行代码。然而，这些缺点还仅仅局限于实例化对象的那部分代码。一旦对象被实例化，它与其他的Set实例就难以区分。在实际的程序中，通过这种限定使用反射的方法，绝大部分代码可以不受影响。

更多资源请访问稀酷客(www.ckook.com)

如果试着编译这个程序，会得到下面的错误消息：

```
Note: SetEx.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

这条警告与程序中使用了泛型有关，但它并不能说明真正的问题。要了解禁止这种警告的最佳方法，请参见第24条。

另一个值得注意的附带问题是，这个程序使用了System.exit。很少有需要调用这个方法的时候，它会终止整个VM（虚拟机）。但是，它对于命令行有效性的非法终止是很合适的。

类对于在运行时可能不存在的其他类、方法或者域的依赖性，用反射法进行管理，这种用法是合理的，但是很少使用。如果要编写一个包，并且它运行的时候必须依赖其他某个包的多个版本，这种做法可能就非常有用。这种做法就是，在支持包所需要的最小环境下对它进行编译，通常是最老的版本，然后以反射方式访问任何更加新的类或者方法。如果企图访问的新类或者新方法在运行时不存在，为了使这种方法有效你还必须采取适当的动作。所谓适当的动作，可能包括使用某种其他可替换的办法来达到同样的目的，或者使用简化的功能进行处理。

简而言之，反射机制是一种功能强大的机制，对于特定的复杂系统编程任务，它是非常必要的，但它也有一些缺点。如果你编写的程序必须要与编译时未知的类一起工作，如有可能，就应该仅仅使用反射机制来实例化对象，而访问对象时则使用编译时已知的某个接口或者超类。

更多资源请访问稀酷客(www.ckook.com)

第54条：谨慎地使用本地方法

Java Native Interface（JNI）允许Java应用程序可以调用本地方法（**native method**），所谓本地方法是指用本地程序设计语言（比如C或者C++）来编写的特殊方法。本地方法在本地语言中可以执行任意的计算任务，并返回到Java程序设计语言。

从历史上看，本地方法主要有三种用途。它们提供了“访问特定于平台的机制”的能力，比如访问注册表（registry）和文件锁（file lock）。它们还提供了访问遗留代码库的能力，从而可以访问遗留数据（legacy data）。最后，本地方法可以通过本地语言，编写应用程序中注重性能的部分，以提高系统的性能。

使用本地方法来访问特定于平台的机制是合法的，但是随着Java平台的不断成熟，它提供了越来越多以前只有在宿主平台上才拥有的特性。例如，1.4发行版本中新增加的java.util.prefs包，提供了注册表的功能，1.6发行版本中增加了java.awt.SystemTray，提供了访问桌面系统托盘区的能力。使用本地方法来访问遗留代码也是合法的。

使用本地方法来提高性能的做法不值得提倡。在早期的发行版本中（1.3发行版本之前），这样做往往是很有必要的，但是JVM实现变得越来越快了。对于大多数任务，现在即使不使用本地方法也可以获得与之相当的性能。举例来说，当Java 1.1发行版本中增加了java.math时，BigInteger是在一个用C编写的快速多精度运算库的基础上实现的。在当时，为了获得足够的性能这样做是必要的。在1.3发行版本中，BigInteger则完全用Java重写了，并且进行了精心的性能调优。即便如此，新的版本还是比原来的版本更快，在这些年间，VM也已经变得更快了。

使用本地方法有一些严重的缺点。因为本地语言不是安全的（见第39条），所以，使用本地方法的应用程序也不再能免受内存毁坏错误的影响。因为本地语言是与平台相关的，使用本地方法的应用程序也不再是可自由移植的。使用本地方法的应用程序也更难调试。在进入和退出本地代码时，需要相关的固定开销，所以，如果本地代码只是做少量的工作，本地方法就可能降低（**decrease**）性能。最后一点，需要“胶合代码”的本地方法编写起来单调乏味，并且难以阅读。

总而言之，在使用本地方法之前务必三思。极少数情况下会需要使用本地方法来提高性能。如果你必须要使用本地方法来访问底层的资源，或者遗留代码库，也要尽可能少用本地代码，并且要全面进行测试。本地代码中的一个Bug就有可能破坏整个应用程序。

更多资源请访问稀酷客(www.ckook.com)

第55条：谨慎地进行优化

有三条与优化有关的格言是每个人都应该知道的。这些格言我们可能已经耳熟能详，但是，如果对它们还不太熟悉，请看下面：

很多计算上的过失都被归咎于效率（没有必要达到的效率），而不是任何其他的原因——甚至包括盲目地做傻事。

——William A. Wulf[Wulf72]

不要去计较效率上的一些小小的得失，在97%的情况下，不成熟的优化才是一切问题的根源。

——Donald E. Knuth[Knuth74]

在优化方面，我们应该遵守两条规则：

规则1：不要进行优化。

规则2（仅针对专家）：还是不要进行优化——也就是说，在你还没有绝对清晰的未优化方案之前，请不要进行优化。

——M. A. Jackson[Jackson75]

所有这些格言都比Java程序设计语言的出现早了20年。它们讲述了一个关于优化的深刻真理：优化的弊大于利，特别是不成熟的优化。在优化过程中，产生的软件可能既不快速，也不正确，而且还不容易修正。

不要因为性能而牺牲合理的结构。要努力编写好的程序而不是快的程序。如果好的程序不够快，它的结构将使它可以得到优化。好的程序体现了信息隐藏（**information hiding**）的原则：只要有可能，它们就会把设计决策集中在单个模块中，因此，可以改变单个决策，而不会影响到系统的其他部分（见第13条）。

这并不意味着，在完成程序之前就可以忽略性能问题。实现上的问题可以通过后期的优化而得到修正，但是，遍布全局并且限制性能的结构缺陷几乎是不可能被改正的，除非重新编写系统。在系统完成之后再改变设计的某个基本方面，会导致系统的结构很不好，从而难以维护和改进。因此，必须在设计过程中考虑到性能问题。

努力避免那些限制性能的设计决策。当一个系统设计完成之后，其中最难以更改的组件是那些指定了模块之间交互关系以及模块与外界交互关系的组件。在这些设计组件之中，最主

更多资源请访问稀酷客(www.ckook.com)

要的是API、线路层（wire-level）协议以及永久数据格式。这些设计组件不仅在事后难以甚至不可能改变，而且它们都有可能对系统本该达到的性能产生严重的限制。

*要考虑API设计决策的性能后果。*使公有的类型成为可变的（mutable），这可能会导致大量不必要的保护性拷贝（见第39条）。同样地，在适合使用复合模式的公有类中使用继承，会把这个类与它的超类永远地束缚在一起，从而人为地限制了子类的性能（见第16条）。最后一个例子，在API中使用实现类型而不是接口，会把你束缚在一个具体的实现上，即使将来出现更快的实现你也无法使用（见第52条）。

API设计对于性能的影响是非常实际的。考虑java.awt.Component类中的getSize方法。这个决定就是，这个注重性能的方法将返回Dimension实例，与此密切相关的决定是，Dimension实例是可变的，迫使这个方法的任何实现都必须为每个调用分配一个新的Dimension实例。尽管在现代VM上分配小对象的开销并不大，但是分配数百万个不必要的对象仍然会严重地损害性能。

在这种情况下，有几种可供选择的替换方案。理想情况下，Dimension应该是不可变的（见第15条）；另一种方案是，用两个方法来替换getSize方法，它们分别返回Dimension对象的单个基本组件。实际上，在1.2发行版本中，出于性能方面的原因，两个这样的方法已经被加入到Component API中。然而，原先的客户端代码仍然可以使用getSize方法，但是仍然要承受原始API设计决策所带来的性能影响。

幸运的是，一般而言，好的API设计也会带来好的性能。*为获得好的性能而对API进行包装，这是一种非常不好的想法。*导致你对API进行包装的性能因素可能会在平台未来的发行版本中，或者在将来的底层软件中不复存在，但是被包装的API以及由它引起的问题将永远困扰着你。

一旦谨慎地设计了程序，并且产生了一个清晰、简明、结构良好的实现，那么就到了该考虑优化的时候了，假定此时你对于程序的性能还不满意。

回想一下Jackson的两条优化规则："不要优化"以及"（仅针对专家）还是不要优化"。他可以再增加一条：*在每次试图做优化之前和之后，要对性能进行测量。*你可能会惊讶于自己的发现。试图做的优化通常对于性能并没有明显的影响，有时候甚至会使性能变得更差。主要的原因在于，要猜出程序把时间花在哪些地方并不容易。你认为程序慢的地方可能并没有问题，这种情况下实际上是在浪费时间去尝试优化。大多数人认为：程序把80%的时间花在20%的代码上了。

性能剖析工具有助于你决定应该把优化的重心放在哪里。这样的工具可以为你提供运行时的信息，比如每个方法大致上花费了多少时间、它被调用多少次。除了确定优化的重点之外，

更多资源请访问稀酷客(www.ckook.com)

它还可以警告你是否需要改变算法。如果一个平方级（或更差）的算法潜藏在程序中，无论怎么调整和优化都很难解决问题。你必须用更有效的算法来替换原来的算法。系统中的代码越多，使用性能剖析器就显得越发重要。这就好像要在一堆干草中寻找一根针：这堆干草越大，使用金属探测器就越有用。JDK带了简单的性能剖析器，现代的IDE也提供了更加成熟的性能剖析工具。

在Java平台上对优化的结果进行测量，比在其他的传统平台上更有必要，因为Java程序设计语言没有很强的性能模型（**performance model**）。各种基本操作的相对开销也没有明确定义。程序员所编写的代码与CPU执行的代码之间存在“语义沟（semantic gap）”，而且这条语义沟比传统编译语言中的更大，这使得要想可靠地预测出任何优化的性能结果都非常困难。大量流传的关于性能的说法最终都被证明为半真半假，或者根本就不正确。

不仅Java的性能模型未得到很好的定义，而且在不同的JVM实现，或者不同的发行版本，以及不同的处理器，在它们这些当中也都各不相同。如果将要在多个JVM实现和多种硬件平台上运行程序，很重要的一点是，需要在每个Java实现上测量优化效果。有时候，还必须在从不同JVM实现或者硬件平台上得到的性能结果之中进行权衡。

总而言之，不要费力去编写快速的程序——应该努力编写好的程序，速度自然会随之而来。在设计系统的时候，特别是在设计API、线路层协议和永久数据格式的时候，一定要考虑性能的因素。当构建完系统之后，要测量它的性能。如果它足够快，你的任务就完成了。如果不够快，则可以在性能剖析器的帮助下，找到问题的根源，然后设法优化系统中相关的部分。第一个步骤是检查所选择的算法：再多的低层优化也无法弥补算法的选择不当。必要时重复这个过程，在每次改变之后都要测量性能，直到满意为止。

更多资源请访问稀酷客(www.ckook.com)

第56条：遵守普遍接受的命名惯例

Java平台建立了一整套很好的命名惯例（**naming convention**），其中有许多命名惯例包含在了《*The Java Language Specification*》[JLS, 6.8]中。不严格地讲，这些命名惯例分为两大类：字面的（typographical）和语法的（grammatical）。

字面的命名惯例比较少，但也涉及包、类、接口、方法、域和类型变量。应该尽量不违反这些惯例，不到万不得已，千万不要违反。如果API违反了这些惯例，它使用起来可能会很困难。如果实现违反了它们，它可能会难以维护。在这两种情况下，违反惯例都会潜在地给使用这些代码的其他程序员带来困惑和苦恼，并且使他们做出错误的假设，造成程序出错。本条目将对这些惯例做简要的介绍。

包的名称应该是层次状的，用句号分隔每个部分。每个部分都包括小写字母和数字（很少使用数字）。任何将在你的组织之外使用的包，其名称都应该以你的组织的Internet域名开头，并且顶级域名放在前面，例如edu.cmu、com.sun、gov.nsa。标准类库和一些可选的包，其名称以java和javax开头，这属于这一规则的例外。用户创建的包的名称绝不能以java和javax开头。关于将Internet域名转换为包名称前缀的详细规则，请参见《*The Java Language Specification*》[JLS, 7.7]。

包名称的其余部分应该包括一个或者多个描述该包的组成部分。这些组成部分应该比较简短，通常不超过8个字符。鼓励使用有意义的缩写形式，例如，使用util而不是utilities。只取首字母的缩写形式也是可以接受的，例如awt。每个组成部分通常都应该由一个单词或者一个缩写词组成。

许多包的名称中除了Internet域名就只有一个组成部分。大型工具包可适当使用额外的组成部分，它们的规模决定了应该分割成非正式的层次结构。例如，javax.swing包有着非常丰富的包层次，如javax.swing.plaf.metal。这样的包通常被称为子包，尽管Java语言并没有提供对包层次的支持。

类和接口的名称，包括枚举和注解类型的名称，都应该包括一个或者多个单词，每个单词的首字母大写，例如Timer和TimerTask。应该尽量避免用缩写，除非是一些首字母缩写和一些通用的缩写，比如max和min。对于首字母缩写，到底应该全部大写还是只有首字母大写，没有统一的说法。虽然大写更常见一些，但还是强烈建议采用仅有首字母大写的形式：即使连续出现多个首字母缩写的形式，你仍然可以区分出一个单词的起始处和结束处。下面这两个类名你更愿意看到哪一个，HTTPURL还是HttpUrl？

方法和域的名称与类和接口的名称一样，都遵守相同的字面惯例，只不过方法或者域的名

更多资源请访问稀酷客(www.ckook.com)

称的第一个字母应该小写，例如remove、ensureCapacity。如果由首字母缩写组成的单词是一个方法或者域名称的第一个单词，它就应该是小写形式。

上述规则的唯一例外是“常量域”，它的名称应该包含一个或者多个大写的单词，中间用下划线符号隔开，例如VALUES或NEGATIVE_INFINITY。常量域是个静态final域，它的值是不可变的。如果静态final域有基本类型，或者有不可变的引用类型（见第15条），它就是个常量域。例如，枚举常量是常量域。如果静态final域有个可变的引用类型，若被引用的对象是不可变的，它也仍然可以是个常量域。注意，常量域是唯一推荐使用下划线的情形。

局部变量名称的字面命名惯例与成员名称类似，只不过它也允许缩写，单个字符和短字符序列的意义取决于局部变量所在的上下文环境，例如i、xref和houseNumber。

类型参数名称通常由单个字母组成。这个字母通常是以下五种类型之一：T表示任意的类型，E表示集合的元素类型，K和V表示映射的键和值类型，X表示异常。任何类型的序列可以是T、U、V或者T1、T2、T3。

为了快速查阅，表8-1列出了字面惯例的例子。

表8-1 字面惯例的例子

标识符类型	例 子
包	com.google.inject，org.joda.time.format
类或者接口	Timer, FutureTask, LinkedHashMap, HttpServlet
方法或者域	remove, ensureCapacity, getCrc
常量域	MIN_VALUE, NEGATIVE_INFINITY
局部变量	i, xref, houseNumber
类型参数	T, E, K, V, X, T1, T2

语法命名惯例比字面惯例更加灵活，也更有争议。对于包而言，没有语法命名惯例。类（包括枚举类型）通常用一个名词或者名词短语命名，例如Timer、BufferedWriter或者ChessPiece。接口的命名与类相似，例如Collection或Comparator，或者用一个以“-able”或“-ible”结尾的形容词来命名，例如Runnable、Iterable或者Accessible。由于注解类型有这么多用处，因此没有单独安排词类。名词、动词、介词和形容词都很常用，例如BindingAnnotation、Inject、ImplementedBy或者Singleton。

执行某个动作的方法通常用动词或者动词短语来命名，例如append或drawImage。对于返回boolean值的方法，其名称往往以单词“is”开头，很少用has，后面跟名词或名词短语，或者任何具有形容词功能的单词或短语，例如isDigit、isProbablePrime、isEmpty、isEnabled或者hasSiblings。

更多资源请访问稀酷客(www.ckook.com)

如果方法返回被调用对象的一个非boolean的函数或者属性，它通常用名词、名词短语，或者以动词“get”开头的动词短语来命名，例如size、hashCode或者getTime。有一种声音认为，只有第三种形式（以“get”开头）才可以接受，但是这种说法没有什么根据。前两种形式往往会产生可读性更好的代码，例如：

```
if (car.speed() > 2 * SPEED_LIMIT)
    generateAudibleAlert("Watch out for cops!");
```

如果方法所在的类是个Bean[JavaBeans]，就要强制使用以“get”开头的形式，而且，如果考虑将来要把这个类转变成Bean，这么做也是明智的。另外，如果这个类包含一个方法用于设置同样的属性，则强烈建议采用这种形式。在这种情况下，这两个方法应该被命名为getAttribute和setAttribute。

有些方法的名称值得专门提及。转换对象类型的方法、返回不同类型的独立对象的方法，通常被称为toType，例如toString和toArray。返回视图（**view**，见第5条，视图的类型不同于接收对象的类型）的方法通常被称为asType，例如asList。返回一个与被调用对象同值的基本类型的方法，通常被称为typeValue，例如intValue。静态工厂的常用名称为valueOf、of、getInstance、newInstance、getType和NewType（见第1条）。

域名称的语法惯例没有很好地建立起来，也没有类、接口和方法名称的惯例那么重要，因为设计良好的API很少会包含暴露出来的域。boolean类型的域命名与boolean类型的访问方法（accessor method）很类似，但是省去了初始的“is”，例如initialized和composite。其他类型的域通常用名词或者名词短语来命名，比如height、digits或bodyStyle。局部变量的语法惯例类似于域的语法惯例，但是更弱一些。

总而言之，把标准的命名惯例当作一种内在的机制来看待，并且学着用它们作为第二特性。字面惯例是非常直接和明确的；语法惯例则更复杂，也更松散。下面这句话引自《The Java Language Specification》[JLS, 6.8]：“如果长期养成的习惯用法与此不同，请不要盲目遵从这些命名惯例。”请运用常识。

更多资源请访问稀酷客(www.ckook.com)

第9章

异　常

充分发挥异常的优点，可以提高程序的可读性、可靠性和可维护性。如果使用不当，它们也会带来负面影响。本章提供了一些关于有效使用异常的指导原则。

第57条：只针对异常的情况才使用异常

某一天，如果你不走运的话，可能会碰到下面这样的代码：

```
// Horrible abuse of exceptions. Don't ever do this!
try {
    int i = 0;
    while(true)
        range[i++].climb();
} catch(ArrayIndexOutOfBoundsException e) {
}
```

这段代码有什么作用？看起来根本不明显，这正是它没有真正被使用的原因（见第55条）。事实证明，作为一个要对数组元素进行遍历的实现方式，它的构想是非常拙劣的。当这个循环企图访问数组边界之外的第一个数组元素时，用抛出（throw）、捕获（catch）、忽略ArrayIndexOutOfBoundsException的手段来达到终止无限循环的目的。假定它与数组循环的标准模式是等价的，对于任何一个Java程序员来说，下面的标准模式一看就会明白：

```
for (Mountain m : range)
    m.climb();
```

那么，为什么有人会优先使用基于异常的模式，而不是用行之有效的模式呢？这是被误导了，他们企图利用Java的错误判断机制来提高性能，因为VM对每次数组访问都要检查越界情况，所以他们认为正常的循环终止测试被编译器隐藏了，但在for-each循环中仍然可见，这无疑是多余的，应该避免。这种想法有三个错误：

- 因为异常机制的设计初衷是用于不正常的情形，所以很少会有JVM实现试图对它们进行

更多资源请访问稀酷客(www.ckook.com)

优化，使得与显式的测试一样快速。

- 把代码放在try-catch块中反而阻止了现代JVM实现本来可能要执行的某些特定优化。
- 对数组进行遍历的标准模式并不会导致冗余的检查。有些现代的JVM实现会将它们优化掉。

实际上，在现代的JVM实现上，基于异常的模式比标准模式要慢得多。在我的机器上，对于一个有100个元素的数组，基于异常的模式比标准模式慢了2倍。

基于异常的循环模式不仅模糊了代码的意图，降低了它的性能，而且它还不能保证正常工作！如果出现了不相关的Bug，这个模式会悄悄地失效，从而掩盖了这个Bug，极大地增加了调试过程的复杂性。假设循环体中的计算过程调用了一个方法，这个方法执行了对某个不相关数组的越界访问。如果使用合理的循环模式，这个Bug会产生未被捕捉的异常，从而导致线程立即结束，产生完整的堆栈轨迹。如果使用这个误导的基于异常的循环模式，与这个Bug相关的异常将会被捕捉到，并且被错误地解释为正常的循环终止条件。

这个例子的教训很简单：*顾名思义，异常应该只用于异常的情况下；它们永远不应该用于正常的控制流*。更一般地，应该优先使用标准的、容易理解的模式，而不是那些声称可以提供更好性能的、弄巧成拙的方法。即使真的能够改进性能，面对平台实现的不断改进，这种模式的性能优势也不可能一直保持。然而，由这种过度聪明的模式带来的微妙的Bug，以及维护的痛苦却依然存在。

这条原则对于API设计也有启发。*设计良好的**API**不应该强迫它的客户端为了正常的控制流而使用异常*。如果类具有“状态相关（state-dependent）”的方法，即只有在特定的不可预知的条件下才可以被调用的方法，这个类往往也应该有个单独的“状态测试（state-testing）”方法，即指示是否可以调用这个状态相关的方法。例如，Iterator接口有一个“状态相关”的next方法，和相应的状态测试方法hasNext。这使得利用传统的for循环（以及for-each循环，在这里，是在内部使用hasNext方法）对集合进行迭代的标准模式成为可能：

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {
    Foo foo = i.next();
    ...
}
```

如果Iterator缺少hasNext方法，客户端将被迫改用下面的做法：

```
// Do not use this hideous code for iteration over a collection!
try {
    Iterator<Foo> i = collection.iterator();
    while(true) {
        Foo foo = i.next();
        ...
```

更多资源请访问稀酷客(www.ckook.com)

```
    }
  } catch (NoSuchElementException e) {
  }
```

这应该非常类似于本条目刚开始时对数组进行迭代的例子。除了代码繁琐且令人误解之外，这个基于异常的模式可能执行起来也比标准模式更差，并且还可能掩盖系统中其他不相关部分中的Bug。

另一种提供单独的状态测试方法的做法是，如果“状态相关的”方法被调用时，该对象处于不适当的状态之中，它就会返回一个可识别的值，比如null。这种方法对于Iterator而言并不合适，因为null是next方法的合法返回值。

对于“状态测试方法”和“可识别的返回值”这两种做法，有些指导原则可以帮助你在两者之中做出选择。如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，使用可被识别的返回值可能是很有必要的，因为在调用“状态测试”方法和调用对应的“状态相关”方法的时间间隔之中，对象的状态有可能会发生变化。如果单独的“状态测试”方法必须重复“状态相关”方法的工作，从性能的角度考虑，就应该使用可被识别的返回值。如果所有其他方面都是等同的，那么“状态测试”方法则略优于可被识别的返回值。它提供了更好的可读性，对于使用不当的情形，可能更加易于检测和改正：如果忘了去调用状态测试方法，状态相关的方法就会抛出异常，使这个Bug变得很明显；如果忘了去检查可识别的返回值，这个Bug就很难会被发现。

总而言之，异常（exception）是为了在异常情况下使用而设计的。不要将它们用于普通的控制流，也不要编写迫使它们这么做的API。

更多资源请访问稀酷客(www.ckook.com)

第58条：对可恢复的情况使用受检异常，对编程错误使用运行时异常

Java程序设计语言提供了三种可抛出结构（throwable）：受检的异常（**checked exception**）、运行时异常（**run-time exception**）和错误（**error**）。关于什么时候适合使用哪种可抛出结构，程序员中间存在一些困惑。虽然这项决定并不总是那么清晰，但还是有些一般性的原则提出了强有力的指导。

在决定使用受检的异常或是未受检的异常时，主要的原则是：*如果期望调用者能够适当地恢复，对于这种情况就应该使用受检的异常*。通过抛出受检的异常，强迫调用者在一个catch子句中处理该异常，或者将它传播出去。因此，方法中声明要抛出的每个受检的异常，都是对API用户的一种潜在指示：与异常相关联的条件是调用这个方法的一种可能的结果。

API的设计者让API用户面对受检的异常，以此强制用户从这个异常条件中恢复。用户可以忽视这样的强制要求，只需捕获异常并忽略即可，但这往往不是个好办法（见第65条）。

有两种未受检的可抛出结构：运行时异常和错误。在行为上两者是等同的：它们都是不需要也不应该被捕获的可抛出结构。如果程序抛出未受检的异常或者错误，往往就属于不可恢复的情形，继续执行下去有害无益。如果程序没有捕捉到这样的可抛出结构，将会导致当前线程停止（halt），并出现适当的错误消息。

用运行时异常来表明编程错误。大多数的运行时异常都表示前提违例（**precondition violation**）。所谓前提违例是指API的客户没有遵守API规范建立的约定。例如，数组访问的约定指明了数组的下标值必须在零和数组长度减1之间。ArrayIndexOutOfBoundsException表明这个前提被违反了。

虽然JLS（Java语言规范）并没有要求，但是按照惯例，错误往往被JVM保留用于表示资源不足、约束失败，或者其他使程序无法继续执行的条件。由于这已经是个几乎被普遍接受的惯例，因此最好不要再实现任何新的Error子类。因此，*你实现的所有未受检的抛出结构都应该是**RuntimeException**的子类*（直接的或者间接的）。

要想定义一个抛出结构，它不是Exception、RuntimeException或Error的子类，这也是可能的。JLS并没有直接规定这样的抛出结构，而是隐式地指定了：从行为意义上讲它们等同于普通的受检异常（即Exception的子类，但不是RuntimeException的子类）。那么，什么时候应该使用这样的抛出结构呢？总之，永远也不会用到。它与普通的受检异常相比没有任何益处，只会困扰API的用户。

总而言之，对于可恢复的情况，使用受检的异常；对于程序错误，则使用运行时异常。当

更多资源请访问稀酷客(www.ckook.com)

然，情况并不总是那么黑白分明。例如，考虑资源枯竭的情形，这可能是由于程序错误而引起的，比如分配了一块不合理的过大的数组，也可能确实是由于资源不足而引起。如果资源枯竭是由于临时的短缺，或是临时需求太大所造成的，这种情况可能就是可恢复的。API设计者需要判断这样的资源枯竭是否允许恢复。如果你相信一种情况可能允许恢复，就使用受检的异常；如果不是，则使用运行时异常。如果不清楚是否有可能恢复，最好使用未受检的异常，原因请参见第59条的讨论。

API的设计者往往会忘记，异常也是个完全意义上的对象，可以在它上面定义任意的方法。这些方法的主要用途是为捕获异常的代码而提供额外的信息，特别是关于引发这个异常条件的信息。如果没有这样的方法，程序员必须要懂得如何解析“该异常的字符串表示法”，以便获得这些额外信息。这是极为不好的做法（见第10条）。类很少会指定它们的字符串表示法中的细节，因此，不同的实现，不同的版本，字符串表示法会大相径庭。因此，“解析异常的字符串表示法”的代码可能是不可移植的，也是非常脆弱的。

因为受检的异常往往指明了可恢复的条件，所以，对于这样的异常，提供一些辅助方法尤其重要，通过这些方法，调用者可以获得一些有助于恢复的信息。例如，假设因为用户没有储存足够数量的钱，他企图在一个收费电话上进行呼叫就会失败，于是抛出受检的异常。这个异常应该提供一个访问方法，以便允许客户查询所缺的费用金额，从而可以将这个数值传递给电话用户。

更多资源请访问稀酷客(www.ckook.com)

第59条：避免不必要地使用受检的异常

受检的异常是Java程序设计语言的一项很好的特性。与返回代码不同，它们强迫程序员处理异常的条件，大大增强了可靠性。也就是说，过分使用受检的异常会使API使用起来非常不方便。如果方法抛出一个或者多个受检的异常，调用该方法的代码就必须在一个或者多个catch块中处理这些异常，或者它必须声明它抛出这些异常，并让它们传播出去。无论哪种方法，都给程序员增添了不可忽视的负担。

如果正确地使用API并不能阻止这种异常条件的产生，并且一旦产生异常，使用API的程序员可以立即采取有用的动作，这种负担就被认为是正当的。除非这两个条件都成立，否则更适合于使用未受检的异常。作为一个“石蕊”测试[⊖]，你可以试着问自己：程序员将如何处理该异常。下面的做法是最好的吗？

```
} catch(TheCheckedException e) {
    throw new AssertionError(); // Can't happen!
}
```

下面这种做法如何？

```
} catch(TheCheckedException e) {
    e.printStackTrace();        // Oh well, we lose.
    System.exit(1);
}
```

如果使用API的程序员无法做得比这更好，那么未受检的异常可能更为合适。这种例子就是CloneNotSupportedException。它是被Object.clone抛出来的，而Object.clone应该只是在实现了Cloneable的对象上才可以被调用（见第11条）。在实践中，catch块几乎总是具有断言（assertion）失败的特征。异常受检的本质并没有为程序员提供任何好处，它反而需要付出努力，还使程序更为复杂。

被一个方法单独抛出的受检异常，会给程序员带来非常高的额外负担。如果这个方法还有其他的受检异常，它被调用的时候一定已经出现在一个try块中，所以这个异常只需要另外一个catch块。如果方法只抛出单个受检的异常，仅仅一个异常就会导致该方法不得不外于try块中。在这些情况下，应该问自己，是否有别的途径来避免使用受检的异常。

“把受检的异常变成未受检的异常”的一种方法是，把这个抛出异常的方法分成两个方法，其中第一个方法返回一个boolean，表明是否应该抛出异常。这种API重构，把下面的调用序列：

⊖ 石蕊测试指简单而具有决定性的测试。——编辑注

更多资源请访问稀酷客(www.ckook.com)

```
// Invocation with checked exception
try {
    obj.action(args);
} catch(TheCheckedException e) {
    // Handle exceptional condition
    ...
}
```

重构为：

```
// Invocation with state-testing method and unchecked exception
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    // Handle exceptional condition
    ...
}
```

这种重构并不总是恰当的，但是，凡是在恰当的地方，它都会使API用起来更加舒服。虽然后者的调用序列没有前者的漂亮，但是这样得到的API更加灵活。如果程序员知道调用将会成功，或者不介意由于调用失败而导致的线程终止，这种重构还允许以下这个更为简单的调用形式：

```
obj.action(args);
```

如果你怀疑这个简单的调用序列是否合乎要求，这个API重构可能就是恰当的。这种重构之后的API在本质上等同于第57条中的“状态测试方法”，并且，同样的告诫依然适用：如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，这种重构就是不恰当的，因为在actionPermitted和action这两个调用的时间间隔之中，对象的状态有可能会发生变化。如果单独的actionPermitted方法必须重复action方法的工作，出于性能的考虑，这种API重构就不值得去做。

更多资源请访问稀酷客(www.ckook.com)

第60条：优先使用标准的异常

专家级程序员与缺乏经验的程序员一个最主要的区别在于，专家追求并且通常也能够实现高度的代码重用。代码重用是值得提倡的，这是一条通用的规则，异常也不例外。Java平台类库提供了一组基本的未受检的异常，它们满足了绝大多数API的异常抛出需要。本条目中，我们将讨论这些常见的可重用异常。

重用现有的异常有多方面的好处。其中最主要的好处是，它使你的API更加易于学习和使用，因为它与程序员已经熟悉的习惯用法是一致的。第二个好处是，对于用到这些API的程序而言，它们的可读性会更好，因为它们不会出现很多程序员不熟悉的异常。最后（也是最不重要的）一点是，异常类越少，意味着内存印迹（footprint）就越小，装载这些类的时间开销也越少。

最经常被重用的异常是IllegalArgumentException。当调用者传递的参数值不合适的时候，往往就会抛出这个异常。例如，假设一个参数代表了“某个动作的重复次数”，如果程序员给这个参数传递了一个负数，就会抛出这个异常。

另一个经常被重用的异常是IllegalStateException。如果因为接收对象的状态而使调用非法，通常就会抛出这个异常。例如，如果在某个对象被正确地初始化之前，调用者就企图使用这个对象，就会抛出这个异常。

可以这么说，所有错误的方法调用都可以被归结为非法参数或者非法状态，但是，其他还有一些标准异常也被用于某些特定情况下的非法参数和非法状态。如果调用者在某个不允许null值的参数中传递了null，习惯的做法就是抛出NullPointerException，而不是IllegalArgument-Exception。同样地，如果调用者在表示序列下标的参数中传递了越界的值，应该抛出的就是IndexOutOfBoundsException，而不是IllegalArgumentException。

另一个值得了解的通用异常是ConcurrentModificationException。如果一个对象被设计为专用于单线程或者与外部同步机制配合使用，一旦发现它正在（或已经）被并发地修改，就应该抛出这个异常。

最后一个值得注意的通用异常是UnsupportedOperationException。如果对象不支持所请求的操作，就会抛出这个异常。与本条目中讨论的其他异常相比，它很少用到，因为绝大多数对象都会支持它们实现的所有方法。如果接口的具体实现没有实现该接口所定义的一个或者多个可选操作，它就可以使用这个异常。例如，对于只支持追加操作的List实现，如果有人试图从列表中删除元素，它就会抛出这个异常。

表9-1概括了最常见的可重用异常。

更多资源请访问稀酷客(www.ckook.com)

表9-1 常用的异常

异 常	使用场合
IllegalArgumentException	非null的参数值不正确
IllegalStateException	对于方法调用而言，对象状态不合适
NullPointerException	在禁止使用null的情况下参数值为null
IndexOutOfBoundsException	下标参数值越界
ConcurrentModificationException	在禁止并发修改的情况下，检测到对象的并发修改
UnsupportedOperationException	对象不支持用户请求的方法

虽然它们是Java平台类库中迄今为止最常被重用的异常，但是，在条件许可的情况下，其他的异常也可以被重用。例如，如果要实现诸如复数或者有理数之类的算术对象，也可以重用ArithmeticException和NumberFormatException。如果某个异常能够满足你的需要，就不要犹豫，使用就是，不过，一定要确保抛出异常的条件与该异常的文档中描述的条件一致。这种重用必须建立在语义的基础上，而不是建立在名称的基础之上。而且，如果希望稍微增加更多的失败－捕获（failure-capture）信息（见第63条），可以放心地把现有的异常进行子类化。

最后，一定要清楚，选择重用哪个异常并不总是那么精确，因为上表中的“使用场合”并不是相互排斥的。例如，考虑表示一副纸牌的对象。假设有个处理发牌操作的方法，它的参数是发一手牌的纸牌张数。假设调用者在这个参数中传递的值大于整副纸牌的剩余张数。这种情形既可以被解释为IllegalArgumentException（handSize参数的值太大），也可以被解释为IllegalStateException（相对于客户的请求而言，纸牌对象包含的纸牌太少）。在这个例子中，感觉IllegalArgumentException要好一些，不过，这里并没有严格的规则。

更多资源请访问稀酷客(www.ckook.com)

第61条：抛出与抽象相对应的异常

如果方法抛出的异常与它所执行的任务没有明显的联系，这种情形将会使人不知所措。当方法传递由低层抽象抛出的异常时，往往会发生这种情况。除了使人感到困惑之外，这也让实现细节污染了更高层的API。如果高层的实现在后续的发行版本中发生了变化，它所抛出的异常也可能会跟着发生变化，从而潜在地破坏现有的客户端程序。

为了避免这个问题，更高层的实现应该捕获低层的异常，同时抛出可以按照高层抽象进行解释的异常。这种做法被称为异常转译（**exception translation**），如下所示：

```
// Exception Translation
try {
    // Use lower-level abstraction to do our bidding
    ...
} catch(LowerLevelException e) {
    throw new HigherLevelException(...);
}
```

下面的异常转译例子取自于AbstractSequentialList类，该类是List接口的一个骨架实现（skeletal implementation）（见第18条）。在这个例子中，按照List<E>接口中get方法的规范要求，异常转译是必需的：

```
/**
 * Returns the element at the specified position in this list.
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= size()}).
 */
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    } catch(NoSuchElementException e) {
        throw new IndexOutOfBoundsException("Index: " + index);
    }
}
```

一种特殊的异常转译形式称为异常链（**exception chaining**），如果低层的异常对于调试导致高层异常的问题非常有帮助，使用异常链就很合适。低层的异常（原因）被传到高层的异常，高层的异常提供访问方法（Throwable.getCause）来获得低层的异常：

```
// Exception Chaining
try {
    ...  // Use lower-level abstraction to do our bidding
} catch (LowerLevelException cause) {
    throw new HigherLevelException(cause);
}
```

高层异常的构造器将原因传到支持链（**chaining-aware**）的超级构造器，因此它最终将被

更多资源请访问稀酷客(www.ckook.com)

传给Throwable的其中一个运行异常链的构造器，例如Throwable（Throwable）：

```
// Exception with chaining-aware constructor
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(cause);
    }
}
```

大多数标准的异常都有支持链的构造器。对于没有支持链的异常，可以利用Throwable的initCause方法设置原因。异常链不仅让你可以通过程序（用getCause）访问原因，它还可以将原因的堆栈轨迹集成到更高层的异常中。

尽管异常转译与不加选择地从低层传递异常的做法相比有所改进，但是它也不能被滥用。如有可能，处理来自低层异常的最好做法是，在调用低层方法之前确保它们会成功执行，从而避免它们抛出异常。有时候，可以在给低层传递参数之前，检查更高层方法的参数的有效性，从而避免低层方法抛出异常。

如果无法避免低层异常，次选方案是，让更高层来悄悄地绕开这些异常，从而将高层方法的调用者与低层的问题隔离开来。在这种情况下，可以用某种适当的记录机制（如java.util.logging）将异常记录下来。这样有助于管理员调查问题，同时又将客户端代码和最终用户与问题隔离开来。

总而言之，如果不能阻止或者处理来自更低层的异常，一般的做法是使用异常转译，除非低层方法碰巧可以保证它抛出的所有异常对高层也合适才可以将异常从低层传播到高层。异常链对高层和低层异常都提供了最佳的功能：它允许抛出适当的高层异常，同时又能捕获底层的原因进行失败分析（见第63条）。

更多资源请访问稀酷客(www.ckook.com)

第62条：每个方法抛出的异常都要有文档

描述一个方法所抛出的异常，是正确使用这个方法时所需文档的重要组成部分。因此，花点时间仔细地为每个方法抛出的异常建立文档是特别重要的。

始终要单独地声明受检的异常，并且利用**Javadoc**的**@throws**标记，准确地记录下抛出每个异常的条件。如果一个方法可能抛出多个异常类，则不要使用“快捷方式”声明它会抛出这些异常类的某个超类。永远不要声明一个方法“throws Exception”，或者更糟糕的是声明它“throws Throwable”，这是非常极端的例子。这样的声明不仅没有为程序员提供关于“这个方法能够抛出哪些异常”的任何指导信息，而且大大地妨碍了该方法的使用，因为它实际上掩盖了该方法在同样的执行环境下可能抛出的任何其他异常。

虽然Java语言本身并不要求程序员为一个方法声明它可能会抛出的未受检异常，但是，如同受检异常一样，仔细地为它们建立文档是非常明智的。未受检的异常通常代表编程上的错误（见第58条），让程序员了解所有这些错误都有助于帮助他们避免犯这样的错误。对于方法可能抛出的未受检异常，如果将这些异常信息很好地组织成列表文档，就可以有效地描述出这个方法被成功执行的前提条件（**precondition**）。每个方法的文档应该描述它的前提条件，这是很重要的，在文档中记录下未受检的异常是满足前提条件的最佳做法。

对于接口中的方法，在文档中记录下它可能抛出的未受检异常显得尤为重要。这份文档构成了该接口的通用约定（**general contract**）的一部分，它指定了该接口的多个实现必须遵循的公共行为。

使用**Javadoc**的**@throws**标签记录下一个方法可能抛出的每个未受检异常，但是不要使用**throws**关键字将未受检的异常包含在方法的声明中。使用API的程序员必须知道哪些异常是需要受检的，哪些是不需要受检的，这很重要，因为这两种情况下他们的责任是不同的。当缺少由throws声明产生的方法标头时，由Javadoc的@throws标签所产生的文档就会提供明显的提示信息，以帮助程序员区分受检的异常和未受检的异常。

应该注意到，为每个方法可能抛出的所有未受检异常建立文档是很理想的，但是在实践中并非总能做到这一点。当类被修订之后，如果有个导出方法被修改了，它将会抛出额外的未受检异常，这不算违反源代码或者二进制兼容性。假设一个类调用了另一个独立类中的方法。第一个类的编写者可能会为每个方法抛出的未受检异常仔细地建立文档，但是，如果第二个类被修订了，抛出了额外的未受检异常，很有可能第一个类（它并没有被修订）就会把新的未受检异常传播出去，尽管它并没有声明这些异常。

如果一个类中的许多方法出于同样的原因而抛出同一个异常，在该类的文档注释中对这个

更多资源请访问稀酷客(www.ckook.com)

异常建立文档，这是可以接受的，而不是为每个方法单独建立文档。一个常见的例子是NullPointerException。如果类的文档注释中有这样的描述："(All methods in this class throw a NullPointerException if a null object reference is passed in any parameter) 如果null对象引用被传递到任何一个参数中，这个类中的所有方法都会抛出NullPointerException"，或者有其他类似的语句，这是可以的。

总而言之，要为你编写的每个方法所能抛出的每个异常建立文档。对于未受检和受检的异常，以及对于抽象的和具体的方法也都一样。要为每个受检异常提供单独的throws子句，不要为未受检的异常提供throws子句。如果没有为可以抛出的异常建立文档，其他人就很难或者根本不可能有效地使用你的类和接口。

更多资源请访问稀酷客(www.ckook.com)

第63条：在细节消息中包含能捕获失败的信息

当程序由于未被捕获的异常而失败的时候，系统会自动地打印出该异常的堆栈轨迹。在堆栈轨迹中包含该异常的字符串表示法（**string representation**），即它的toString方法的调用结果。它通常包含该异常的类名，紧随其后的是细节消息（**detail message**）。通常，这只是程序员或者域服务人员（field service personnel，指检查软件失败的人）在调查软件失败原因时必须检查的信息。如果失败的情形不容易重现，要想获得更多的信息会非常困难，甚至是不可能的。因此，异常类型的toString方法应该尽可能多地返回有关失败原因的信息，这一点特别重要。换句话说，异常的细节消息应该捕获住失败，便于以后分析。

为了捕获失败，异常的细节信息应该包含所有“对该异常有贡献”的参数和域的值。例如，IndexOutOfBoundsException异常的细节消息应该包含下界、上界以及没有落在界内的下标值。该细节消息提供了许多关于失败的信息。这三个值中任何一个或者全部都有可能是错的。实际的下标值可能小于下界或等于上界（“越界错误”），或者它可能是个无效值，太小或太大。下界也有可能大于上界（严重违反内部约束条件的一种情况）。每一种情形都代表了不同的问题，如果程序员知道应该去查找哪种错误，就可以极大地加速诊断过程。

虽然在异常的细节消息中包含所有相关的“硬数据（hard data）”是非常重要的，但是包含大量的描述信息往往没有什么意义。堆栈轨迹的用途是与源文件结合起来进行分析，它通常包含抛出该异常的确切文件和行数，以及堆栈中所有其他方法调用所在的文件和行数。关于失败的冗长描述信息通常是不必要的，这些信息可以通过阅读源代码而获得。

异常的细节消息不应该与“用户层次的错误消息”混为一谈，后者对于最终用户而言必须是可理解的。与用户层次的错误消息不同，异常的字符串表示法主要是让程序员或者域服务人员用来分析失败的原因。因此，信息的内容比可理解性要重要得多。

为了确保在异常的细节消息中包含足够的能捕获失败的信息，一种办法是在异常的构造器而不是字符串细节消息中引入这些信息。然后，有了这些信息，只要把它们放到消息描述中，就可以自动产生细节消息。例如，IndexOutOfBoundsException并不是有个String构造器，而是有个这样的构造器：

```
/**
 * Construct an IndexOutOfBoundsException.
 *
 * @param lowerBound the lowest legal index value.
 * @param upperBound the highest legal index value plus one.
 * @param index      the actual index value.
 */
public IndexOutOfBoundsException(int lowerBound, int upperBound,
                                 int index) {
    // Generate a detail message that captures the failure
```

更多资源请访问稀酷客(www.ckook.com)

```
        super("Lower bound: "   + lowerBound +
              ", Upper bound: " + upperBound +
              ", Index: "       + index);

        // Save failure information for programmatic access
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
        this.index = index;
    }
```

遗憾的是，Java平台类库并没有广泛地使用这种做法，但是，这种做法仍然值得大力推荐。它使程序员更加易于抛出异常以捕获失败。实际上，这种做法使程序员不想捕获失败都难！这种做法可以有效地把代码集中起来放在异常类中，由这些代码对异常类自身中的异常产生高质量的细节消息，而不是要求类的每个用户都多余地产生细节消息。

正如第58条中所建议的，为异常的“失败捕获”信息提供一些访问方法是合适的（在上述例子中的lowerBound、upperBound和index方法）提供一些访问方法是合适的。提供这样的访问方法对于受检的异常，比对于未受检的异常更为重要，因为失败——捕获信息对于从失败中恢复是非常有用的。程序员希望通过程序的手段来访问未受检异常的细节，这很少见（尽管也是可以想像得到的）。然而，即使对于未受检的异常，作为一般原则提供这些访问方法也是明智的（见第44页中的第10条）。

更多资源请访问稀酷客(www.ckook.com)

第64条：努力使失败保持原子性

当对象抛出异常之后，通常我们期望这个对象仍然保持在一种定义良好的可用状态之中，即使失败是发生在执行某个操作的过程中间。对于受检的异常而言，这尤为重要，因为调用者期望能从这种异常中进行恢复。一般而言，失败的方法调用应该使对象保持在被调用之前的状态。具有这种属性的方法被称为具有失败原子性（**failure atomic**）。

有几种途径可以实现这种效果。最简单的办法莫过于设计一个不可变的对象（见第15条）。如果对象是不可变的，失败原子性就是显然的。如果一个操作失败了，它可能会阻止创建新的对象，但是永远也不会使已有的对象保持在不一致的状态之中，因为当每个对象被创建之后它就处于一致的状态之中，以后也不会再发生变化。

对于在可变对象上执行操作的方法，获得失败原子性最常见的办法是，在执行操作之前检查参数的有效性（见第38条）。这可以使得在对象的状态被修改之前，先抛出适当的异常。例如，考虑第6条中的Stack.pop方法：

```
public Object pop() {
    if (size == 0)
        throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; // Eliminate obsolete reference
    return result;
}
```

如果取消对初始大小（size）的检查，当这个方法企图从一个空栈中弹出元素时，它仍然会抛出异常。然而，这将会导致size域保持在不一致的状态（负数）之中，从而导致将来对该对象的任何方法调用都会失败。此外，那时候pop方法抛出的异常也不适于抽象（见第61条）。

一种类似的获得失败原子性的办法是，调整计算处理过程的顺序，使得任何可能会失败的计算部分都在对象状态被修改之前发生。如果对参数的检查只有在执行了部分计算之后才能进行，这种办法实际上就是上一种办法的自然扩展。例如，考虑TreeMap的情形，它的元素被按照某种特定的顺序做了排序。为了向TreeMap中添加元素，该元素的类型就必须是可以利用TreeMap的排序准则与其他元素进行比较的。如果企图增加类型不正确的元素，在tree以任何方式被修改之前，自然会导致ClassCastException异常。

第三种获得失败原子性的办法远远没有那么常用，做法是编写一段恢复代码（**recovery code**），由它来拦截操作过程中发生的失败，以及使对象回滚到操作开始之前的状态上。这种办法主要用于永久性的（基于磁盘的（disk-based））数据结构。

最后一种获得失败原子性的办法是，在对象的一份临时拷贝上执行操作，当操作完成之后

更多资源请访问稀酷客(www.ckook.com)

再用临时拷贝中的结果代替对象的内容。如果数据保存在临时的数据结构中，计算过程会更加迅速，使用这种办法就是件很自然的事。例如，Collections.sort在执行排序之前，首先把它的输入列表转到一个数组中，以便降低在排序的内循环中访问元素所需要的开销。这是出于性能考虑的做法，但是，它增加了一项优势：即使排序失败，它也能保证输入列表保持原样。

虽然一般情况下都希望实现失败原子性，但并非总是可以做到。例如，如果两个线程企图在没有适当的同步机制的情况下，并发地修改同一个对象，这个对象就有可能被留在不一致的状态之中。因此，在捕获了ConcurrentModificationException异常之后再假设对象仍然是可用的，这就是不正确的。错误（相对于异常）通常是不可恢复的，当方法抛出错误时，它们不需要努力保持失败原子性。

即使在可以实现失败原子性的场合，它也并不总是人们所期望的。对于某些操作，它会显著地增加开销或者复杂性。但一旦意识到这个问题，实现失败原子性往往轻松自如。

一般而言，作为方法规范的一部分，产生的任何异常都应该让对象保持在该方法调用之前的状态。如果违反这条规则，API文档就应该清楚地指明对象将会处于什么样的状态。遗憾的是，大量现有的API文档都未能做到这一点。

更多资源请访问稀酷客(www.ckook.com)

第65条：不要忽略异常

尽管这条建议看上去是显而易见的，但是它却常常被违反，因而值得再次提出来。当API的设计者声明一个方法将抛出某个异常的时候，他们等于正在试图说明某些事情。所以，请不要忽略它！要忽略一个异常非常容易，只需将方法调用通过try语句包围起来，并包含一个空的catch块：

```
// Empty catch block ignores exception - Highly suspect!
try {
    ...
} catch (SomeException e) {
}
```

空的catch块会使异常达不到应有的目的，即强迫你处理异常的情况。忽略异常就如同忽略火警信号一样——若把火警信号器关掉了，当真正的火灾发生时，就没有人能看到火警信号了。或许你会侥幸逃过劫难，或许结果将是灾难性的。每当见到空的catch块时，应该警钟长鸣。至少，catch块也应该包含一条说明，解释为什么可以忽略这个异常。

有一种情形可以忽略异常，即关闭FileInputStream的时候。因为你还没有改变文件的状态，因此不必执行任何恢复动作，并且已经从文件中读取到所需要的信息，因此不必终止正在进行的操作。即使在这种情况下，把异常记录下来还是明智的做法，因为如果这些异常经常发生，你就可以调查异常的原因。

本条目中的建议同样适用于受检异常和未受检的异常。不管异常代表了可预见的异常条件，还是编程错误，用空的catch块忽略它，将会导致程序在遇到错误的情况下悄然地执行下去。然后，有可能在将来的某个点上，当程序不能再容忍与错误源明显相关的问题时，它就会失败。正确地处理异常能够彻底挽回失败。只要将异常传播给外界，至少会导致程序迅速地失败，从而保留了有助于调试该失败条件的信息。

更多资源请访问稀酷客(www.ckook.com)

第10章

并　发

线程（Thread）机制允许同时进行多个活动。并发程序设计比单线程程序设计要困难得多，因为有更多的东西可能出错，也很难以重现失败。但是你无法避免并发，因为我们所做的大部分事情都需要并发，而且并发也是能否从多核的处理器中获得好的性能的一个条件，这些现在都是很平常的事了。本章阐述的建议可以帮助你编写出清晰、正确、文档组织良好的并发程序。

第66条：同步访问共享的可变数据

关键字synchronized可以保证在同一时刻，只有一个线程可以执行某一个方法，或者某一个代码块。许多程序员把同步的概念仅仅理解为一种互斥的方式，即，当一个对象被一个线程修改的时候，可以阻止另一个线程观察到对象内部不一致的状态。按照这种观点，对象被创建的时候处于一致的状态（见第15条），当有方法访问它的时候，它就被锁定了。这些方法观察到对象的状态，并且可能会引起状态转变（**state transition**），即把对象从一种一致的状态转换到另一种一致的状态。正确地使用同步可以保证没有任何方法会看到对象处于不一致的状态中。

这种观点是正确的，但是它并没有说明同步的全部意义。如果没有同步，一个线程的变化就不能被其他线程看到。同步不仅可以阻止一个线程看到对象处于不一致的状态之中，它还可以保证进入同步方法或者同步代码块的每个线程，都看到由同一个锁保护的之前所有的修改效果。

Java语言规范保证读或者写一个变量是原子的（**atomic**），除非这个变量的类型为long或者double[JLS，17.4.7]。换句话说，读取一个非long或double类型的变量，可以保证返回的值是某个线程保存在该变量中的，即使多个线程在没有同步的情况下并发地修改这个变量也是如此。

更多资源请访问稀酷客(www.ckook.com)

你可能听说过，为了提高性能，在读或写原子数据的时候，应该避免使用同步。这个建议是非常危险而错误的。虽然语言规范保证了线程在读取原子数据的时候，不会看到任意的数值，但是它并不保证一个线程写入的值对于另一个线程将是可见的。为了在线程之间进行可靠的通信，也为了互斥访问，同步是必要的。这归因于Java语言规范中的内存模型（**memory model**），它规定了一个线程所做的变化何时以及如何变成对其他线程可见[JLS, 17, Goetz06 16]。

如果对共享的可变数据的访问不能同步，其后果将非常可怕，即使这个变量是原子可读写的。考虑下面这个阻止一个线程妨碍另一个线程的任务。Java的类库中提供了Thread.stop方法，但是这个方法在很久以前就不提倡使用，因为它本质上是不安全的（**unsafe**）——使用它会导致数据遭到破坏。不要使用**Thread.stop**。要阻止一个线程妨碍另一个线程，建议做法是让第一个线程轮询（poll）一个boolean域，这个域一开始为false，但是可以通过第二个线程设置为true，以表示第一个线程将终止自己。由于boolean域的读和写操作都是原子的，程序员在访问这个域的时候不再使用同步：

```
// Broken! - How long would you expect this program to run?
public class StopThread {
    private static boolean stopRequested;

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested)
                    i++;
            }
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

你可能期待这个程序运行大约一秒钟左右，之后主线程将stopRequested设置为true，致使后台线程的循环终止。但是在我的机器上，这个程序永远不会终止：因为后台线程永远在循环！

问题在于，由于没有同步，就不能保证后台线程何时“看到”主线程对stopRequested的值所做的改变。没有同步，虚拟机将这个代码：

```
while (!done)
    i++;
```

转变成这样：

更多资源请访问稀酷客(www.ckook.com)

```
if (!done)
    while (true)
        i++;
```

这是可以接受的。这种优化称作提升（**hoisting**），正是HopSpot Server VM的工作。结果是个活性失败（**liveness failure**）：这个程序无法前进。修正这个问题的一种方式是同步访问stopRequested域。这个程序会如预期般在大约一秒钟之内终止：

```
// Properly synchronized cooperative thread termination
public class StopThread {
    private static boolean stopRequested;
    private static synchronized void requestStop() {
        stopRequested = true;
    }
    private static synchronized boolean stopRequested() {
        return stopRequested;
    }

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested())
                    i++;
            }
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        requestStop();
    }
}
```

注意写方法（**requestStop**）和读方法（**stopRequested**）都被同步了。只同步写方法还不够！实际上，如果读和写操作没有都被同步，同步就不会起作用。

StopThread中被同步方法的动作即使没有同步也是原子的。换句话说，这些方法的同步只是为了它的通信效果，而不是为了互斥访问。虽然循环的每个迭代中的同步开销很小，还是有其他更正确的替代方法，它更加简洁，性能也可能更好。如果stopRequested被声明为volatile，第二种版本的StopThread中的锁就可以省略。虽然volatile修饰符不执行互斥访问，但它可以保证任何一个线程在读取该域的时候都将看到最近刚刚被写入的值：

```
// Cooperative thread termination with a volatile field
public class StopThread {
    private static volatile boolean stopRequested;

    public static void main(String[] args)
            throws InterruptedException {
        Thread backgroundThread = new Thread(new Runnable() {
            public void run() {
                int i = 0;
                while (!stopRequested)
                    i++;
            }
```

更多资源请访问稀酷客(www.ckook.com)

```
        });
        backgroundThread.start();

        TimeUnit.SECONDS.sleep(1);
        stopRequested = true;
    }
}
```

在使用volatile的时候务必要小心。考虑下面的方法，假设它要产生序列号：

```
// Broken - requires synchronization!
private static volatile int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

这个方法的目的是要确保每个调用都返回不同的值（只要不超过2^{32}个调用）。这个方法的状态只包含一个可原子访问的域：nextSerialNumber，这个域的所有可能的值都是合法的。因此，不需要任何同步来保护它的约束条件。然而，如果没有同步，这个方法仍然无法正常工作。

问题在于，增量操作符（++）不是原子的。它在nextSerialNumber域中执行两项操作：首先它读取值，然后写回一个新值，相当于原来的值再加上1。如果第二个线程在第一个线程读取旧值和写回新值期间读取这个域，第二个线程就会与第一个线程一起看到同一个值，并返回相同的序列号。这就是安全性失败（**safety failure**）：这个程序会计算出错误的结果。

修正generateSerialNumber方法的一种方法是在它的声明中增加synchronized修饰符。这样可以确保多个调用不会交叉存取，确保每个调用都会看到之前所有调用的效果。一旦这么做，就可以且应该从nextSerialNumber中删除volatile修饰符。为了让这个方法更可靠，要用long代替int，或者在nextSerialNumber快要重叠时抛出异常。

最好还是遵循第47条中的建议，使用类AtomicLong，它是java.util.concurrent.atomic的一部分。它所做的工作正是你想要的，并且有可能比同步版的generateSerialNumber执行得更好：

```
private static final AtomicLong nextSerialNum = new AtomicLong();

public static long generateSerialNumber() {
    return nextSerialNum.getAndIncrement();
}
```

避免本条目中所讨论到的问题的最佳办法是不共享可变的数据。要么共享不可变的数据（见第15条），要么压根不共享。换句话说，将可变数据限制在单个线程中。如果采用这一策略，对它建立文档就很重要，以便它可以随着程序的发展而得到维护。深刻地理解正在使用的框架和类库也很重要，因为它们引入了你所不知道的线程。

更多资源请访问稀酷客(www.ckook.com)

让一个线程在短时间内修改一个数据对象，然后与其他线程共享，这是可以接受的，只同步共享对象引用的动作。然后其他线程没有进一步的同步也可以读取对象，只要它没有再被修改。这种对象被称作事实上不可变的（**effectively immutable**）[Goetz06 3.5.4]。将这种对象引用从一个线程传递到其他的线程被称作安全发布（**safe publication**）[Goetz06 3.5.3]。安全发布对象引用有许多种方法：可以将它保存在静态域中，作为类初始化的一部分；可以将它保存在volatile域、final域或者通过正常锁定访问的域中；或者可以将它放到并发的集合中（见第69条）。

简而言之，当多个线程共享可变数据的时候，每个读或者写数据的线程都必须执行同步。如果没有同步，就无法保证一个线程所做的修改可以被另一个线程获知。未能同步共享可变数据会造成程序的活性失败（**liveness failure**）和安全性失败（**safety failure**）。这样的失败是最难以调试的。它们可能是间歇性的，且与时间相关，程序的行为在不同的VM上可能根本不同。如果只需要线程之间的交互通信，而不需要互斥，volatile修饰符就是一种可以接受的同步形式，但要正确地使用它可能需要一些技巧。

更多资源请访问稀酷客(www.ckook.com)

第67条：避免过度同步

第66条告诫我们缺少同步的危险性。本条目则关注相反的问题。依据情况的不同，过度同步可能会导致性能降低、死锁，甚至不确定的行为。

为了避免活性失败和安全性失败，在一个被同步的方法或者代码块中，永远不要放弃对客户端的控制。换句话说，在一个被同步的区域内部，不要调用设计成要被覆盖的方法，或者是由客户端以函数对象的形式提供的方法（见第21条）。从包含该同步区域的类的角度来看，这样的方法是外来的（alien）。这个类不知道该方法会做什么事情，也无法控制它。根据外来方法的作用，从同步区域中调用它会导致异常、死锁或者数据损坏。

为了对这个过程进行更具体的说明，来考虑下面的类，它实现了一个可以观察到的集合包装（set wrapper）。该类允许客户端在将元素添加到集合中时预订通知。这就是观察者（Observer）模式[Gamma95, p.293]。为了简洁起见，类在从集合中删除元素时没有提供通知，但要提供通知也是件很容易的事情。这个类是在第73页第16条中可重用的ForwardingSet上实现的：

```
// Broken - invokes alien method from synchronized block!
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers =
        new ArrayList<SetObserver<E>>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }
    public boolean removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            return observers.remove(observer);
        }
    }
    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }
    @Override public boolean add(E element) {
        boolean added = super.add(element);
        if (added)
            notifyElementAdded(element);
        return added;
    }

    @Override public boolean addAll(Collection<? extends E> c) {
        boolean result = false;
        for (E element : c)
            result |= add(element);  // calls notifyElementAdded
```

更多资源请访问稀酷客(www.ckook.com)

```
        return result;
    }
}
```

Observer通过调用addObserver方法预订通知，通过调用removeObserver方法取消预订。在这两种情况下，这个回调接口的实例都会被传递给方法：

```
public interface SetObserver<E> {
    // Invoked when an element is added to the observable set
    void added(ObservableSet<E> set, E element);
}
```

如果只是粗略地检验一下，ObservableSet会显得很正常。例如，下面的程序打印出0～99的数字：

```
public static void main(String[] args) {
    ObservableSet<Integer> set =
        new ObservableSet<Integer>(new HashSet<Integer>());

    set.addObserver(new SetObserver<Integer>() {
        public void added(ObservableSet<Integer> s, Integer e) {
            System.out.println(e);
        }
    });

    for (int i = 0; i < 100; i++)
        set.add(i);
}
```

现在我们来尝试一些更复杂点的例子。假设我们用一个addObserver调用来代替这个调用，用来替换的那个addObserver调用传递了一个打印Integer值的观察者，这个值被添加到该集合中，如果值为23，这个观察者要将自身删除：

```
set.addObserver(new SetObserver<Integer>() {
    public void added(ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) s.removeObserver(this);
    }
});
```

你可能以为这个程序会打印0～23的数字，之后观察者会取消预订，程序会悄悄地完成它的工作。实际上却是打印出0～23的数字，然后抛出ConcurrentModificationException。问题在于，当notifyElementAdded调用观察者的added方法时，它正处于遍历observers列表的过程中。added方法调用可观察集合的removeObserver方法，从而调用observers.remove。现在我们有麻烦了。我们正企图在遍历列表的过程中，将一个元素从列表中删除，这是非法的。notifyElementAdded方法中的迭代是在一个同步的块中，可以防止并发的修改，但是无法防止迭代线程本身回调到可观察的集合中，也无法防止修改它的observers列表。

现在我们要尝试一些比较奇特的例子：我们来编写一个试图取消预订的观察者，但是不直接调用removeObserver，它用另一个线程的服务来完成。这个观察者使用了一个**executor**

更多资源请访问稀酷客(www.ckook.com)

service（见第68条）：

```
// Observer that uses a background thread needlessly
set.addObserver(new SetObserver<Integer>() {
    public void added(final ObservableSet<Integer> s, Integer e) {
        System.out.println(e);
        if (e == 23) {
            ExecutorService executor =
                Executors.newSingleThreadExecutor();
            final SetObserver<Integer> observer = this;
            try {
                executor.submit(new Runnable() {
                    public void run() {
                        s.removeObserver(observer);
                    }
                }).get();
            } catch (ExecutionException ex) {
                throw new AssertionError(ex.getCause());
            } catch (InterruptedException ex) {
                throw new AssertionError(ex.getCause());
            } finally {
                executor.shutdown();
            }
        }
    }
});
```

这一次我们没有遇到异常，而是遭遇了死锁。后台线程调用s.removeObserver，它企图锁定observers，但它无法获得该锁，因为主线程已经有锁了。在这期间，主线程一直在等待后台线程来完成对观察者的删除，这正是造成死锁的原因。

这个例子是刻意编写用来示范的，因为观察者实际上没理由使用后台线程，但这个问题却是真实的。从同步区域中调用外来方法，在真实的系统中已经造成了许多死锁，例如GUI工具箱。

在前面这两个例子中（异常和死锁），我们都还算幸运的。调用外来方法（**added**）时，同步区域（**observers**）所保护的资源处于一致的状态。假设当同步区域所保护的约束条件暂时无效时，你要从同步区域中调用一个外来方法。由于Java程序设计语言中的锁是可重入的（reentrant），这种调用不会死锁。就像在第一个例子中一样，它会产生一个异常，因为调用线程已经有这个锁了，因此当该线程试图再次获得该锁时会成功，尽管概念上不相关的另一项操作正在该锁所保护的数据上进行着。这种失败的后果可能是灾难性的。从本质上来说，这个锁没有尽到它的职责。可再入的锁简化了多线程的面向对象程序的构造，但是它们可能会将活性失败（lireness failure）变成安全性失败（safety failure）。

幸运的是，通过将外来方法的调用移出同步的代码块来解决这个问题通常并不太困难。对于notifyElementAdded方法，这还涉及给observers列表拍张“快照”，然后没有锁也可以安全地遍历这个列表了。经过这一修改，前两个例子运行起来便再也不会出现异常或者死锁了：

```
// Alien method moved outside of synchronized block - open calls
private void notifyElementAdded(E element) {
```

更多资源请访问稀酷客(www.ckook.com)

```
    List<SetObserver<E>> snapshot = null;
    synchronized(observers) {
        snapshot = new ArrayList<SetObserver<E>>(observers);
    }
    for (SetObserver<E> observer : snapshot)
        observer.added(this, element);
}
```

事实上，要将外来方法的调用移出同步的代码块，还有一种更好的方法。自从Java 1.5发行版本以来，Java类库就提供了一个并发集合（**concurrent collection**），见第69条，称作CopyOnWriteArrayList，这是专门为此定制的。这是ArrayList的一种变体，通过重新拷贝整个底层数组，在这里实现所有的写操作。由于内部数组永远不改动，因此迭代不需要锁定，速度也非常快。如果大量使用，CopyOnWriteArrayList的性能将大受影响，但是对于观察者列表来说却是很好的，因为它们几乎不改动，并且经常被遍历。

如果这个列表改成使用CopyOnWriteArrayList，就不必改动ObservableSet的add和addAll方法。下面是这个类的其余代码。注意其中并没有任何显式的同步。

```
// Thread-safe observable set with CopyOnWriteArrayList
private final List<SetObserver<E>> observers =
    new CopyOnWriteArrayList<SetObserver<E>>();

public void addObserver(SetObserver<E> observer) {
    observers.add(observer);
}
public boolean removeObserver(SetObserver<E> observer) {
    return observers.remove(observer);
}
private void notifyElementAdded(E element) {
    for (SetObserver<E> observer : observers)
        observer.added(this, element);
}
```

在同步区域之外被调用的外来方法被称作“开放调用（**open call**）”[Lea00, 2.4.1.3]。除了可以避免死锁之外，开放调用还可以极大地增加并发性。外来方法的运行时间可能会任意长。如果在同步区域内调用外来方法，其他线程对受保护资源的访问就会遭到不必要的拒绝。

通常，你应该在同步区域内做尽可能少的工作。获得锁，检查共享数据，根据需要转换数据，然后放掉锁。如果你必须要执行某个很耗时的动作，则应该设法把这个动作移到同步区域的外面，而不违背第66条中的指导方针。

本条目的第一部分是关于正确性的。接下来，我们要简单地讨论一下性能。虽然自从Java平台早期以来，同步的成本已经下降了，但更重要的是，永远不要过度同步。在这个多核的时代，过度同步的实际成本并不是指获取锁所花费的CPU时间；而是指失去了并行的机会，以及因为需要确保每个核都有一个一致的内存视图而导致的延迟。过度同步的另一项潜在开销在于，它会限制VM优化代码执行的能力。

如果一个可变的类要并发使用，应该使这个类变成是线程安全的（见第70条），通过内部

更多资源请访问稀酷客(www.ckook.com)

同步，你还可以获得明显比从外部锁定整个对象更高的并发性。否则，就不要在内部同步。让客户在必要的时候从外部同步。在Java平台出现的早期，许多类都违背了这些指导方针。例如，StringBuffer实例几乎总是被用于单个线程之中，而它们执行的却是内部同步。为此，StringBuffer基本上都由StringBuilder代替，它在Java 1.5发行版本中是个非同步的StringBuffer。当你不确定的时候，就不要同步你的类，而是应该建立文档，注明它不是线程安全的（见第70条）。

如果你在内部同步了类，就可以使用不同的方法来实现高并发性，例如分拆锁（lock splitting）、分离锁（lock striping）和非阻塞（nonblocking）并发控制。这些方法都超出了本书的讨论范围，但有其他著作对此进行了阐述[Goetz06, Lea00]。

如果方法修改了静态域，那么你也必须同步对这个域的访问，即使它往往只用于单个线程。客户要在这种方法上执行外部同步是不可能的，因为不可能保证其他不相关的客户也会执行外部同步。第232页中的generateSerialNumber方法就是这样的一个例子。

简而言之，为了避免死锁和数据破坏，千万不要从同步区域内部调用外来方法。更为一般地讲，要尽量限制同步区域内部的工作量。当你在设计一个可变类的时候，要考虑一下它们是否应该自己完成同步操作。在现在这个多核的时代，这比永远不要过度同步来得更重要。只有当你有足够的理由一定要在内部同步类的时候，才应该这么做，同时还应该将这个决定清楚地写到文档中（见第70条）。

更多资源请访问稀酷客(www.ckook.com)

第68条：executor和task优先于线程

本书第1版中阐述了简单的工作队列（**work queue**）[Bloch01，见第50条]的代码。这个类允许客户将后台线程异步处理的工作项目加入队列（enqueue）。当不再需要这个工作队列时，客户端可以调用一个方法，让后台线程在完成了已经在队列中的所有工作之后，优雅地终止自己。这个实现几乎就像件玩具，但即使如此，它还是需要一整页精细的代码，一不小心，就容易出现安全问题或者导致活性失败（liveness failure）。幸运的是，你再也不需要编写这样的代码了。

在Java 1.5发行版本中，Java平台中增加了java.util.concurrent。这个包中包含了一个**Executor Framework**，这是一个很灵活的基于接口的任务执行工具。它创建了一个在各方面都比本书第一版更好的工作队列，却只需要这一行代码：

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

下面是为执行提交一个runnable的方法：

```
executor.execute(runnable);
```

下面是告诉executor如何优雅地终止（如果做不到这一点，虚拟机可能将不会退出）：

```
executor.shutdown();
```

你可以利用executor service完成更多的事情。例如，可以等待完成一项特殊的任务（就如第236页第67条中的“后台线程SetObserver”中的一样），你可以等待一个任务集合中的任何任务或者所有任务完成（利用invokeAny或者invokeAll方法），你可以等待executor service优雅地完成终止（利用awaitTermination方法），你可以在任务完成时逐个地获取这些任务的结果（利用ExecutorCompletionService），等等。

如果想让不止一个线程来处理来自这个队列的请求，只要调用一个不同的静态工厂，这个工厂创建了一种不同的executor service，称作线程池（**thread pool**）。你可以用固定或者可变数目的线程创建一个线程池。java.util.concurrent.Executors类包含了静态工厂，能为你提供所需的大多数executor。然而，如果你想来点特别的，可以直接使用ThreadPoolExecutor类。这个类允许你控制线程池操作的几乎每个方面。

为特殊的应用程序选择executor service是很有技巧的。如果编写的是小程序，或者是轻载的服务器，使用Executors.newCachedThreadPool通常是个不错的选择，因为它不需要配置，并且一般情况下能够正确地完成工作。但是对于大负载的服务器来说，缓存的线程池就不是很好的选择了！在缓存的线程池中，被提交的任务没有排成队列，而是直接交给线程执行。如果没有线程可用，就创建一个新的线程。如果服务器负载得太重，以致它所有的CPU都完

更多资源请访问稀酷客(www.ckook.com)

全被占用了，当有更多的任务时，就会创建更多的线程，这样只会使情况变得更糟。因此，在大负载的产品服务器中，最好使用Executors.newFixedThreadPool，它为你提供了一个包含固定线程数目的线程池，或者为了最大限度地控制它，就直接使用ThreadPoolExecutor类。

你不仅应该尽量不要编写自己的工作队列，而且还应该尽量不直接使用线程。现在关键的抽象不再是Thread了，它以前可是既充当工作单元，又是执行机制。现在工作单元和执行机制是分开的。现在关键的抽象是工作单元，称作任务（task）。任务有两种：Runnable及其近亲Callable（它与Runnable类似，但它会返回值）。执行任务的通用机制是executor service。如果你从任务的角度来看问题，并让一个executor service替你执行任务，在选择适当的执行策略方面就获得了极大的灵活性。从本质上讲，Executor Famework所做的工作是执行，犹如Collections Framework所做的工作是聚集（aggregation）一样。

Executor Framework也有一个可以代替java.util.Timer的东西，即ScheduledThreadPool-Executor。虽然timer使用起来更加容易，但是被调度的线程池executor更加灵活。timer只用一个线程来执行任务，这在面对长期运行的任务时，会影响到定时的准确性。如果timer唯一的线程抛出未被捕获的异常，timer就会停止执行。被调度的线程池executor支持多个线程，并且优雅地从抛出未受检异常的任务中恢复。

Executor Framework的完整处理方法超出了本书的讨论范围，但是有兴趣的读者可以参阅《Java Concurrency in Practice》[⊖]一书[Goetz06]。

⊖ 中文版为《Java并发编程实践》书号为978-7-121-04316-1。——译者注

更多资源请访问稀酷客(www.ckook.com)

第69条：并发工具优先于wait和notify

本书第1版中专门用了一个条目来说明如何正确地使用wait和notify（Bloch01，见第50条）。它提出的建议仍然有效，并且在本条目的最后也对此做了概述，但是这条建议现在远远没有之前那么重要了。这是因为几乎没有理由再使用wait和notify了。自从Java 1.5发行版本开始，Java平台就提供了更高级的并发工具，它们可以完成以前必须在wait和notify上手写代码来完成的各项工作。既然正确地使用**wait**和**notify**比较困难，就应该用更高级的并发工具来代替。

java.util.concurrent中更高级的工具分成三类：Executor Framework、并发集合（Concurrent Collectioin）以及同步器（Synchronizer），Executor Framework只在第68条中简单地提到过。并发集合和同步器将在本条目中进行简单的阐述。

并发集合为标准的集合接口（如List、Queue和Map）提供了高性能的并发实现。为了提供高并发性，这些实现在内部自己管理同步（见第67条）。因此，并发集合中不可能排除并发活动；将它锁定没有什么作用，只会使程序的速度变慢。

这意味着客户无法原子地对并发集合进行方法调用。因此有些集合接口已经通过依赖状态的修改操作（state-dependent modify operation）进行了扩展，它将几个基本操作合并到了单个原子操作中。例如，ConcurrentMap扩展了Map接口，并添加了几个方法，包括putIfAbsent(key, value)，当键没有映射时会替它插入一个映射，并返回与键关联的前一个值，如果没有这样的值，则返回null。这样使得实现线程安全的标准Map就很容易了。例如，下面这个方法模拟了String.intern的行为：

```
// Concurrent canonicalizing map atop ConcurrentMap - not optimal
private static final ConcurrentMap<String, String> map =
    new ConcurrentHashMap<String, String>();

public static String intern(String s) {
    String previousValue = map.putIfAbsent(s, s);
    return previousValue == null ? s : previousValue;
}
```

事实上，你还可以做得更好。ConcurrentHashMap对获取操作（如get）进行了优化。因此，只有当get表明有必要的时候，才值得先调用get，再调用putIfAbsent：

```
// Concurrent canonicalizing map atop ConcurrentMap - faster!
public static String intern(String s) {
    String result = map.get(s);
    if (result == null) {
        result = map.putIfAbsent(s, s);
        if (result == null)
            result = s;
    }
    return result;
}
```

更多资源请访问稀酷客(www.ckook.com)

ConcurrentHashMap除了提供卓越的并发性之外，速度也非常快。在我的机器上，上面这个优化过的intern方法比String.intern速度快了超过6倍（但是记住，String.intern必须使用某种弱引用，来避免随着时间的推移而发生内存泄露）。除非不得已，否则应该优先使用**ConcurrentHashMap**，而不是使用**Collections.synchronizedMap**或者**Hashtable**。只要用并发Map替换老式的同步Map，就可以极大地提升并发应用程序的性能。更一般地，应该优先使用并发集合，而不是使用外部同步的集合。

有些集合接口已经通过阻塞操作（**blocking operation**）进行了扩展，它们会一直等待（或者阻塞）到可以成功执行为止。例如，BlockingQueue扩展了Queue接口，并添加了包括take在内的几个方法，它从队列中删除并返回了头元素，如果队列为空，就等待。这样就允许将阻塞队列用于工作队列（**work queue**），也称作生产者-消费者队列（**producer-consumer queue**），一个或者多个生产者线程（**producer thread**）在工作队列中添加工作项目，并且当工作项目可用时，一个或者多个消费者线程（**consumer thread**）则从工作队列中取出队列并处理工作项目。不出所料，大多数ExecutorService实现（包括ThreadPoolExecutor）都使用BlockingQueue（见第68条）。

同步器（Synchronizer）是一些使线程能够等待另一个线程的对象，允许它们协调动作。最常用的同步器是CountDownLatch和Semaphore。较不常用的是CyclicBarrier和Exchanger。

倒计数锁存器（Countdown Latch）是一次性的障碍，允许一个或者多个线程等待一个或者多个其他线程来做某些事情。CountDownLatch的唯一构造器带有一个int类型的参数，这个int参数是指允许所有在等待的线程被处理之前，必须在锁存器上调用countDown方法的次数。

要在这个简单的基本类型之上构建一些有用的东西，做起来是相当的容易。例如，假设想要构建一个简单的框架，用来给一个动作的并发执行定时。这个框架中包含单个方法，这个方法带有一个执行该动作的executor，一个并发级别（表示要并发执行该动作的次数），以及表示该动作的runnable。所有的工作线程（worker thread）自身都准备好，要在timer线程启动时钟之前运行该动作（为了实现准确的定时，这是必需的）。当最后一个工作线程准备好运行该动作时，timer线程就“发起头炮”，同时允许工作线程执行该动作。一旦最后一个工作线程执行完该动作，timer线程就立即停止计时。直接在wait和notify之上实现这个逻辑至少来说会很混乱，而在CountDownLatch之上实现则相当简单：

```
// Simple framework for timing concurrent execution
public static long time(Executor executor, int concurrency,
        final Runnable action) throws InterruptedException {
    final CountDownLatch ready = new CountDownLatch(concurrency);
    final CountDownLatch start = new CountDownLatch(1);
    final CountDownLatch done = new CountDownLatch(concurrency);
    for (int i = 0; i < concurrency; i++) {
        executor.execute(new Runnable() {
```

更多资源请访问稀酷客(www.ckook.com)

```
                public void run() {
                    ready.countDown(); // Tell timer we're ready
                    try {
                        start.await(); // Wait till peers are ready
                        action.run();
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    } finally {
                        done.countDown();  // Tell timer we're done
                    }
                }
            });
        }
        ready.await();      // Wait for all workers to be ready
        long startNanos = System.nanoTime();
        start.countDown(); // And they're off!
        done.await();       // Wait for all workers to finish
        return System.nanoTime() - startNanos;
    }
```

注意这个方法使用了三个倒计数锁存器。第一个是ready，工作线程用它来告诉timer线程它们已经准备好了。然后工作线程在第二个锁存器上等待，也就是start。当最后一个工作线程调用ready.countDown时，timer线程记录下起始时间，并调用start.countDown，允许所有的工作线程继续进行。然后timer线程在第三个锁存器（即done）上等待，直到最后一个工作线程运行完该动作，并调用done.countDown。一旦调用这个，timer线程就会苏醒过来，并记录下结束的时间。

还有一些细节值得注意。传递给time方法的executor必须允许创建至少与指定并发级别一样多的线程，否则这个测试就永远不会结束。这就是*线程饥饿死锁*（**thread starvation deadlock**）[Goetz06 8.1.1]。如果工作线程捕捉到InterruptedException，就会利用习惯用法Thread.currentThread().interrupt()重新断言中断，并从它的run方法中返回。这样就允许executor在必要的时候处理中断，事实上也理当如此。最后，注意利用了System.nanoTime来给活动定时，而不是利用System.currentTimeMillis。*对于间歇式的定时，始终应该优先使用* **System.nanoTime**，*而不是使用* **System.currentTimeMills**。System.nanoTime更加准确也更加精确，它不受系统的实时时钟的调整所影响。

本条目仅仅触及了并发工具的一些皮毛。例如，前一个例子中的那三个倒计数锁存器（CountdownLatch）其实可以用一个CyclicBarrier来代替。这样产生的代码更加简洁，但是理解起来比较困难。更多信息，请参阅《*Java Concurrency in Practice*》一书[Goetz06]。

虽然你始终应该优先使用并发工具，而不是使用wait和notify，但可能必须维护使用了wait和notify的遗留代码。wait方法被用来使线程等待某个条件。它必须在同步区域内部被调用，这个同步区域将对象锁定在了调用wait方法的对象上。下面是使用wait方法的标准模式：

```
// The standard idiom for using the wait method
synchronized (obj) {
    while (<condition does not hold>)
```

更多资源请访问稀酷客(www.ckook.com)

```
        obj.wait(); // (Releases lock, and reacquires on wakeup)
    ... // Perform action appropriate to condition
}
```

始终应该使用wait循环模式来调用wait方法；永远不要在循环之外调用wait方法。循环会在等待之前和之后测试条件。

在等待之前测试条件，当条件已经成立时就跳过等待，这对于确保活性（liveness）是必要的。如果条件已经成立，并且在线程等待之前，notify（或者notifyAll）方法已经被调用，则无法保证该线程将会从等待中苏醒过来。

在等待之后测试条件，如果条件不成立的话继续等待，这对于确保安全性（safety）是必要的。当条件不成立的时候，如果线程继续执行，则可能会破坏被锁保护的约束关系。当条件不成立时，有下面一些理由可使一个线程苏醒过来：

- 另一个线程可能已经得到了锁，并且从一个线程调用notify那一刻起，到等待线程苏醒过来的这段时间中，得到锁的线程已经改变了受保护的状态。
- 条件并不成立，但是另一个线程可能意外地或恶意地调用了notify。在公有可访问的对象上等待，这些类实际上把自己暴露在了这种危险的境地中。公有可访问对象的同步方法中包含的wait都会出现这样的问题。
- 通知线程（notifying thread）在唤醒等待线程时可能会过度“大方”。例如，即使只有某一些等待线程的条件已经被满足，但是通知线程可能仍然调用notifyAll。
- 在没有通知的情况下，等待线程也可能（但很少）会苏醒过来。这被称为“伪唤醒（spurious wakeup）”[Posix, 11.4.3.6.1；JavaSE6]。

一个相关的话题是，为了唤醒正在等待的线程，你应该使用notify还是notifyAll（回忆一下，notify唤醒的是单个正在等待的线程，假设有这样的线程存在，而notifyAll唤醒的则是所有正在等待的线程）。一种常见的说法是，你总是应该使用notifyAll。这是合理而保守的建议。它总会产生正确的结果，因为它可以保证你将会唤醒所有需要被唤醒的线程。你可能也会唤醒其他一些线程，但是这不会影响程序的正确性。这些线程醒来之后，会检查它们正在等待的条件，如果发现条件并不满足，就会继续等待。

从优化的角度来看，如果处于等待状态的所有线程都在等待同一个条件，而每次只有一个线程可以从这个条件中被唤醒，那么你就应该选择调用notify，而不是notifyAll。

即使这些条件都是真的，也许还是有理由使用notifyAll而不是notify。就好像把wait调用放在一个循环中，以避免在公有可访问对象上的意外或恶意的通知一样，与此类似，使用

更多资源请访问稀酷客(www.ckook.com)

notifyAll代替notify可以避免来自不相关线程的意外或恶意的等待。否则，这样的等待会“吞掉”一个关键的通知，使真正的接收线程无限地等待下去。

简而言之，直接使用wait和notify就像用“并发汇编语言”进行编程一样，而java.util.concurrent则提供了更高级的语言。没有理由在新代码中使用**wait**和**notify**，即使有，也是极少的。如果你在维护使用wait和notify的代码，务必确保始终是利用标准的模式从while循环内部调用wait。一般情况下，你应该优先使用notifyAll，而不是使用notify。如果使用notify，请一定要小心，以确保程序的活性（liveness）。

更多资源请访问稀酷客(www.ckook.com)

第70条：线程安全性的文档化

当一个类的实例或者静态方法被并发使用的时候，这个类的行为如何，是该类与其客户端程序建立的约定的重要组成部分。如果你没有在一个类的文档中描述其行为的并发性情况，使用这个类的程序员将不得不做出某些假设。如果这些假设是错误的，这样得到的程序就可能缺少足够的同步（见第66条），或者过度同步（见第67条）。无论属于这其中的哪种情况，都可能会发生严重的错误。

你可能听到过这样的说法：通过查看文档中是否出现synchronized修饰符，可以确定一个方法是否是线程安全的。这种说法从几个方面来说都是错误的。在正常的操作中，Javadoc并没有在它的输出中包含synchronized修饰符，这是有理由的。*因为在一个方法声明中出现**synchronized**修饰符，这是个实现细节，并不是导出的**API**的一部分。*它并不一定表明这个方法是线程安全的。

而且，“出现了synchronized关键字就足以用文档说明线程安全性”的这种说法隐含了一个错误的观念，即认为线程安全性是一种“要么全有要么全无”的属性。实际上，线程安全性有多种级别。*一个类为了可被多个线程安全地使用，必须在文档中清楚地说明它所支持的线程安全性级别。*

下面的列表概括了线程安全性的几种级别。这份列表并没有涵盖所有的可能，而只是些常见的情形：

- **不可变的**（immutable）——这个类的实例是不变的。所以，不需要外部的同步。这样的例子包括String、Long和BigInteger（见第15条）。
- **无条件的线程安全**（unconditionally thread-safe）——这个类的实例是可变的，但是这个类有着足够的内部同步，所以，它的实例可以被并发使用，无需任何外部同步。其例子包括Random和ConcurrentHashMap。
- **有条件的线程安全**（conditionally thread-safe）——除了有些方法为进行安全的并发使用而需要外部同步之外，这种线程安全级别与无条件的线程安全相同。这样的例子包括Collections.synchronized包装返回的集合，它们的迭代器（iterator）要求外部同步。
- **非线程安全**（not thread-safe）——这个类的实例是可变的。为了并发地使用它们，客户必须利用自己选择的外部同步包围每个方法调用（或者调用序列）。这样的例子包括通用的集合实现，例如ArrayList和HashMap。

更多资源请访问稀酷客(www.ckook.com)

- **线程对立的（thread-hostile）**——这个类不能安全地被多个线程并发使用，即使所有的方法调用都被外部同步包围。线程对立的根源通常在于，没有同步地修改静态数据。没有人会有意编写一个线程对立的类；这种类是因为没有考虑到并发性而产生的后果。幸运的是，在Java平台类库中，线程对立的类或者方法非常少。System.runFinalizersOnExit方法是线程对立的，但已经被废除了。

这些分类（除了线程对立的之外）粗略对应于《*Java Concurrency in Practice*》一书中的*线程安全注解*（thread safety annotation），分别为Immutable、ThreadSafe和NotThreadsafe [Goetz06，Appendix A]。上述分类中无条件和有条件的线程安全类别都涵盖在ThreadSafe注解中了。

在文档中描述一个有条件的线程安全类要特别小心。你必须指明哪个调用序列需要外部同步，还要指明为了执行这些序列，必须获得哪一把锁（极少的情况下是指哪几把锁）。通常情况下，这是指作用在实例自身上的那把锁，但也有例外。如果一个对象代表了另一个对象的一个*视图*（view），客户通常就必须在后台对象上同步，以防止其他线程直接修改后台对象。例如，Collections.synchronizedMap的文档应该有这样的说明：

It is imperative that the user manually synchronize on the returned map when

iterating over any of its collection views:

（当遍历任何被返回Map的集合视图时，用户必须手工对它们进行同步：）

```
Map<K, V> m = Collections.synchronizedMap(new HashMap<K, V>());
    ...
Set<K> s = m.keySet();  // Needn't be in synchronized block
    ...
synchronized(m) {  // Synchronizing on m, not s!
    for (K key : s)
        key.f();
}
```

如果没有遵循这样的建议，就可能造成不确定的行为。

类的线程安全说明通常放在它的文档注释中，但是带有特殊线程安全属性的方法则应该在它们自己的文档注释中说明它们的属性。没有必要说明枚举类型的不可变性。除非从返回类型来看已经很明显，否则静态工厂必须在文档中说明被返回对象的线程安全性，如Collections.synchronizedMap（上述）所示。

当一个类承诺了“使用一个公有可访问的锁对象”时，就意味着允许客户端以原子的方式执行一个方法调用序列，但是，这种灵活性是要付出代价的。并发集合（如ConcurrentHashMap和ConcurrentLinkedQueue）使用的那种并发控制，并不能与高性能的内部并发控制相兼容。客户端还可以发起拒绝服务（denial-of service）攻击，他只需超时地保持公有可访问锁即可。

更多资源请访问稀酷客(www.ckook.com)

这有可能是无意的，也可能是有意的。

为了避免这种拒绝服务攻击，应该使用一个私有锁对象（private lock object）来代替同步的方法（隐含着一个公有可访问锁）：

```
// Private lock object idiom - thwarts denial-of-service attack
private final Object lock = new Object();

public void foo() {
    synchronized(lock) {
        ...
    }
}
```

因为这个私有锁对象不能被这个类的客户端程序所访问，所以它们不可能妨碍对象的同步。实际上，我们正是在应用第13条的建议，把锁对象封装在它所同步的对象中。

注意lock域被声明为final的。这样可以防止不小心改变它的内容，而导致不同步访问包含对象的悲惨后果（见第66条）。我们这是在应用第15条的建议，将lock域的可变性减到最小。

重申一下，私有锁对象模式只能用在无条件的线程安全类上。有条件的线程安全类不能使用这种模式，因为它们必须在文档中说明：在执行某些方法调用序列时，它们的客户端程序必须获得哪把锁。

私有锁对象模式特别适用于那些专门为继承而设计的类（见第17条）。如果这种类使用它的实例作为锁对象，子类可能很容易在无意中妨碍基类的操作，反之亦然。出于不同的目的而使用相同的锁，子类和基类可能会“相互绊住对方的脚”。这不只是一个理论意义上的问题。例如，这种现象在Thread类上就出现过[Bloch05，Puzzle 77]。

简而言之，每个类都应该利用字斟句酌的说明或者线程安全注解，清楚地在文档中说明它的线程安全属性。synchronized修饰符与这个文档毫无关系。有条件的线程安全类必须在文档中指明“哪个方法调用序列需要外部同步，以及在执行这些序列的时候要获得哪把锁”。如果你编写的是无条件的线程安全类，就应该考虑使用私有锁对象来代替同步的方法。这样可以防止客户端程序和子类的不同步干扰，让你能够在后续的版本中灵活地对并发控制采用更加复杂的方法。

更多资源请访问稀酷客(www.ckook.com)

第71条：慎用延迟初始化

延迟初始化（lazy initialization）是延迟到需要域的值时才将它初始化的这种行为。如果永远不需要这个值，这个域就永远不会被初始化。这种方法既适用于静态域，也适用于实例域。虽然延迟初始化主要是一种优化，但它也可以用来打破类和实例初始化中的有害循环[Bloch05，Puzzle 51]。

就像大多数的优化一样，对于延迟初始化，最好建议“除非绝对必要，否则就不要这么做”（见第55条）。延迟初始化就像一把双刃剑。它降低了初始化类或者创建实例的开销，却增加了访问被延迟初始化的域的开销。根据延迟初始化的域最终需要初始化的比例、初始化这些域要多少开销，以及每个域多久被访问一次，延迟初始化（就像其他的许多优化一样）实际上降低了性能。

也就是说，延迟初始化有它的好处。如果域只在类的实例部分被访问，并且初始化这个域的开销很高，可能就值得进行延迟初始化。要确定这一点，唯一的办法就是测量类在用和不用延迟初始化时的性能差别。

当有多个线程时，延迟初始化是需要技巧的。如果两个或者多个线程共享一个延迟初始化的域，采用某种形式的同步是很重要的，否则就可能造成严重的Bug（见第66条）。本条目中讨论的所有初始化方法都是线程安全的。

在大多数情况下，正常的初始化要优先于延迟初始化。下面是正常初始化的实例域的一个典型声明。注意其中使用了final修饰符（见第15条）：

```
// Normal initialization of an instance field
private final FieldType field = computeFieldValue();
```

如果利用延迟优化来破坏初始化的循环，就要使用同步访问方法，因为它是最简单、最清楚的替代方法：

```
// Lazy initialization of instance field - synchronized accessor
private FieldType field;

synchronized FieldType getField() {
    if (field == null)
        field = computeFieldValue();
    return field;
}
```

这两种习惯模式（正常的初始化和使用了同步访问方法的延迟初始化）应用到静态域上时保持不变，除了给域和访问方法声明添加了static修饰符之外。

更多资源请访问稀酷客(www.ckook.com)

如果出于性能的考虑而需要对静态域使用延迟初始化，就使用**lazy initialization holder class**模式。这种模式（也称作**initialize-on-demand holder class idiom**）保证了类要到被用到的时候才会被初始化[JLS，12.4.1]。如下所示：

```
// Lazy initialization holder class idiom for static fields
private static class FieldHolder {
    static final FieldType field = computeFieldValue();
}
static FieldType getField() { return FieldHolder.field; }
```

当getField方法第一次被调用时，它第一次读取FieldHolder.field，导致FieldHolder类得到初始化。这种模式的魅力在于，getField方法没有被同步，并且只执行一个域访问，因此延迟初始化实际上并没有增加任何访问成本。现代的VM将在初始化该类的时候，同步域的访问。一旦这个类被初始化，VM将修补代码，以便后续对该域的访问不会导致任何测试或者同步。

如果出于性能的考虑而需要对实例域使用延迟初始化，就使用双重检查模式（double-check idiom）。这种模式避免了在域被初始化之后访问这个域时的锁定开销（见第67条）。这种模式背后的思想是：两次检查域的值[因此名字叫双重检查（double-check）]，第一次检查时没有锁定，看看这个域是否被初始化了；第二次检查时有锁定。只有当第二次检查时表明这个域没有被初始化，才会调用computeFieldValue方法对这个域进行初始化。因为如果域已经被初始化就不会有锁定，域被声明为volatile很重要（见第66条）。下面就是这种习惯模式：

```
// Double-check idiom for lazy initialization of instance fields
private volatile FieldType field;
FieldType getField() {
    FieldType result = field;
    if (result == null) {  // First check (no locking)
        synchronized(this) {
            result = field;
            if (result == null)  // Second check (with locking)
                field = result = computeFieldValue();
        }
    }
    return result;
}
```

这段代码可能看起来似乎有些费解。尤其对于需要用到局部变量result可能有点不解。这个变量的作用是确保field只在已经被初始化的情况下读取一次。虽然这不是严格需要，但是可以提升性能，并且因为给低级的并发编程应用了一些标准，因此更加优雅。在我的机器上，上述的方法比没用局部变量的方法快了大约25%。

在Java 1.5发行版本之前，双重检查模式的功能很不稳定，因为volatile修饰符的语义不够强，难以支持它[Pugh01]。Java 1.5发行版本中引入的内存模式解决了这个问题[JLS，17，Goetz06 16]。如今，双重检查模式是延迟初始化一个实例域的方法。虽然你也可以对静态域应用双重检查模式，但是没有理由这么做，因为lazy initialization holder class idiom是更好的

更多资源请访问稀酷客(www.ckook.com)

选择。

双重检查模式的两个变量值得一提。有时候，你可能需要延迟初始化一个可以接受重复初始化的实例域。如果处于这种情况，就可以使用双重检查惯用法的一个变形，它省去了第二次检查。没错，它就是单重检查模式（**single-check idiom**）。下面就是这样的一个例子。注意field仍然被声明为volatile：

```
// Single-check idiom - can cause repeated initialization!
private volatile FieldType field;

private FieldType getField() {
    FieldType result = field;
    if (result == null)
        field = result = computeFieldValue();
    return result;
}
```

本条目中讨论的所有初始化方法都适用于基本类型的域，以及对象引用域。当双重检查模式（double-check idiom）或者单重检查模式（single-check idiom）应用到数值型的基本类型域时，就会用0来检查这个域（这是数值型基本变量的默认值），而不是用null。

如果你不在意是否每个线程都重新计算域的值，并且域的类型为基本类型，而不是long或者double类型，就可以选择从单重检查模式的域声明中删除volatile修饰符。这种变体称之为racy single-check idiom。它加快了某些架构上的域访问，代价是增加了额外的初始化（直到访问该域的每个线程都进行一次初始化）。这显然是一种特殊的方法，不适合于日常的使用。然而，String实例却用它来缓存它们的散列码。

简而言之，大多数的域应该正常地进行初始化，而不是延迟初始化。如果为了达到性能目标，或者为了破坏有害的初始化循环，而必须延迟初始化一个域，就可以使用相应的延迟初始化方法。对于实例域，就使用双重检查模式（double-check idiom）；对于静态域，则使用lazy initialization holder class idiom。对于可以接受重复初始化的实例域，也可以考虑使用单重检查模式（single-check idiom）。

更多资源请访问稀酷客(www.ckook.com)

第72条：不要依赖于线程调度器

当有多个线程可以运行时，由线程调度器（thread scheduler）决定哪些线程将会运行，以及运行多长时间。任何一个合理的操作系统在做出这样的决定时，都会努力做到公正，但是所采用的策略却大相径庭。因此，编写良好的程序不应该依赖于这种策略的细节。任何依赖于线程调度器来达到正确性或者性能要求的程序，很有可能都是不可移植的。

要编写健壮的、响应良好的、可移植的多线程应用程序，最好的办法是确保可运行线程的平均数量不明显多于处理器的数量。这使得线程调度器没有更多的选择：它只需要运行这些可运行的线程，直到它们不再可运行为止。即使在根本不同的线程调度算法下，这些程序的行为也不会有很大的变化。注意可运行线程的数量并不等于线程的总数量，前者可能更多。在等待的线程并不是可运行的。

保持可运行线程数量尽可能少的主要方法是，让每个线程做些有意义的工作，然后等待更多有意义的工作。如果线程没有在做有意义的工作，就不应该运行。根据Executor Framework（见第68条），这意味着适当地规定了线程池的大小[Goetz06 8.2]，并且使任务保持适当地小，彼此独立。任务不应该太小，否则分配的开销也会影响到性能。

线程不应该一直处于忙－等（**busy-wait**）的状态，即反复地检查一个共享对象，以等待某些事情发生。除了使程序易受到调度器的变化影响之外，忙－等这种做法也会极大地增加处理器的负担，降低了同一机器上其他进程可以完成的有用工作量。作为一个极端的反面例子，考虑下面这个CountDownLatch的不正当的重新实现：

```
// Awful CountDownLatch implementation - busy-waits incessantly!
public class SlowCountDownLatch {
    private int count;
    public SlowCountDownLatch(int count) {
        if (count < 0)
            throw new IllegalArgumentException(count + " < 0");
        this.count = count;
    }

    public void await() {
        while (true) {
            synchronized(this) {
                if (count == 0) return;
            }
        }
    }
    public synchronized void countDown() {
        if (count != 0)
            count--;
    }
}
```

在我的机器上，当1000个线程在锁存器（latch）中等待的时候，SlowCountDownLatch比

更多资源请访问稀酷客(www.ckook.com)

CountDownLatch慢了大约2000倍。虽然这个例子可能显得有点牵强，但是系统中有一个或者多个线程处于不必要的可运行状态，这种现象并不少见。结果虽然不像SlowCountDownLatch那么悲惨，但是性能和可移植性都可能受到损害。

如果某一个程序不能工作，是因为某些线程无法像其他线程那样获得足够的CPU时间，那么，*不要企图通过调用**Thread.yield**来“修正”该程序*。你可能好不容易成功地让程序能够工作，但这样得到的程序仍然是不可移植的。同一个yield调用在一个JVM实现上能提高性能，而在另一个JVM实现上却有可能会更差，在第三个JVM实现上则可能没有影响。**Thread.yield**没有可测试的语义（testable semantic）。更好的解决办法是重新构造应用程序，以减少可并发运行的线程数量。

有一种相关的方法是调整线程优先级（thread priority），同样有类似的警告。*线程优先级是**Java**平台上最不可移植的特征了*。通过调整某些线程的优先级来改善应用程序的响应能力，这样做并非不合理，却是不必要的，也是不可移植的。通过调整线程的优先级来解决严重的活性问题是不合理的。在你找到并修正底层的真正原因之前，这个问题可能会再次出现。

在本书第一版中说过，对于大多数程序员来说，Thread.yield的*唯一用途是在测试期间人为地增加程序的并发性*。意思就是，通过探查程序中更大部分的状态空间，可以发现一些隐蔽的Bug。这种方法曾经十分奏效，但从来不能保证一定可行。在Java语言规范中，Thread.yield根本不做实质性的工作，只是将控制权返回给它的调用者。有些现代的VM实际上正是这种功能。因此，应该使用Thread.sleep(1)代替Thread.yield来进行并发测试。千万不要使用Thread.sleep(0)，它会立即返回。

简而言之，不要让应用程序的正确性依赖于线程调度器。否则，结果得到的应用程序将既不健壮，也不具有可移植性。作为推论，不要依赖Thread.yield或者线程优先级。这些设施仅仅对调度器作些暗示。线程优先级可以用来提高一个已经能够正常工作的程序的服务质量，但永远不应该用来“修正”一个原本并不能工作的程序。

更多资源请访问稀酷客(www.ckook.com)

第73条：避免使用线程组

除了线程、锁和监视器之外，线程系统还提供了一个基本的抽象，即线程组（thread group）。线程组的初衷是作为一种隔离applet（小程序）的机制，当然是出于安全的考虑。但是它们从来没有真正履行这个承诺，它们的安全价值已经差到根本不在Java安全模型的标准工作中提及的地步[Gong03]。

既然线程组并没有提供所提及的任何安全功能，那么它们到底提供了什么功能呢？不多。它们允许你同时把Thread的某些基本功能应用到一组线程上。其中有一些基本功能已经被废弃了，剩下的也很少使用。

具有讽刺意味的是，从线程安全性的角度来看，ThreadGroup API非常弱。为了得到一个线程组中的活动线程列表，你必须调用enumerate方法，它有一个数组参数，并且数组的容量必须足够大，以便容纳所有的活动线程。activeCount方法返回一个线程组中活动线程的数量，但是，一旦这个数组进行了分配，并传递给了enumerate方法，就不保证原先得到的活动线程数仍是正确的。如果线程数增加了，而数组太小，enumerate方法就会悄然地忽略掉无法在数组中容纳的线程。

列出线程组中子组的API也有类似的缺陷。虽然通过增加新的方法，这些问题都有可能得到修正，但是，它们目前还没有被修正，因为线程组已经过时了，所以实际上根本没有必要修正。

在Java 1.5发行版本之前，有一种小功能只有ThreadGroup API才有：当线程抛出未被捕捉的异常时，ThreadGroup.uncaughtException方法是获得控制权的唯一方式。这项功能很有用，例如，为了把堆栈轨迹定向到一个特定于应用程序的日志中。然而，自从Java 1.5发行版本之后，Thread的setUncaughtExceptionHandler方法也提供了同样的功能。

总而言之，线程组并没有提供太多有用的功能，而且它们提供的许多功能还都是有缺陷的。我们最好把线程组看作是一个不成功的试验，你可以忽略掉它们，就当它们根本不存在一样。如果你正在设计的一个类需要处理线程的逻辑组，或许就应该使用线程池executor（见第68条）。

更多资源请访问稀酷客(www.ckook.com)

第11章

序列化

本章关注对象序列化（**object serialization**）API，它提供了一个框架，用来将对象编码成字节流，并从字节流编码中重新构建对象。“将一个对象编码成一个字节流”，称作将该对象序列化（**serializing**）；相反的处理过程被称作反序列化（**deserializing**）。一旦对象被序列化后，它的编码就可以从一台正在运行的虚拟机被传递到另一台虚拟机上，或者被存储到磁盘上，供以后反序列化时用。序列化技术为远程通信提供了标准的线路级（wire-level）对象表示法，也为JavaBeans组件结构提供了标准的持久化数据格式。本章中有一项值得特别提及的特性，就是序列化代理（**serialization proxy**）模式（见第78条），它可以帮助你避免对象序列化的许多缺陷。

第74条：谨慎地实现Serializable接口

要想使一个类的实例可被序列化，非常简单，只要在它的声明中加入“implements Serializable”字样即可。正因为太容易了，所以普遍存在这样一种误解，认为程序员毫不费力就可以实现序列化。实际的情形要复杂得多。虽然使一个类可被序列化的直接开销非常低，甚至可以忽略不计，但是为了序列化而付出的长期开销往往是实实在在的。

实现Serializable接口而付出的最大代价是，一旦一个类被发布，就大大降低了“改变这个类的实现”的灵活性。如果一个类实现了Serializable接口，它的字节流编码（或者说序列化形式，serialized form）就变成了它的导出的API的一部分。一旦这个类被广泛使用，往往必须永远支持这种序列化形式，就好像你必须要支持导出的API的所有其他部分一样。如果你不努力设计一种自定义的序列化形式（**custom serialized form**），而仅仅接受了默认的序列化形式，这种序列化形式将被永远地束缚在该类最初的内部表示法上。换句话说，如果你接受了默认的序列化形式，这个类中私有的和包级私有的实例域将都变成导出的API的一部分，这不符合“最低限度地访问域”的实践准则（见第13条），从而它就失去了作为信息隐藏工具的有效性。

更多资源请访问稀酷客(www.ckook.com)

如果你接受了默认的序列化形式，并且以后又要改变这个类的内部表示法，结果可能导致序列化形式的不兼容。客户端程序企图用这个类的旧版本来序列化一个类，然后用新版本进行反序列化，结果将导致程序失败。在改变内部表示法的同时仍然维持原来的序列化形式（使用ObjectOutputStream.putFields和ObjectInputStream.readFields），这也是可能的，但是做起来比较困难，并且会在源代码中留下一些明显的隐患。因此，你应该仔细地设计一种高质量的序列化形式，并且在很长时间内都愿意使用这种形式（见第75，78条）。这样做将会增加开发的初始成本，但这是值得的。设计良好的序列化形式也许会给类的演变带来限制；但是设计不好的序列化形式则可能会使类根本无法演变。

序列化会使类的演变受到限制，这种限制的一个例子与流的唯一标识符（**stream unique identifier**）有关，通常它也被称为序列版本**UID**（**serial version UID**）。每个可序列化的类都有一个唯一标识号与它相关联。如果你没有在一个名为serialVersionUID的私有静态final的long域中显式地指定该标识号，系统就会自动地根据这个类来调用一个复杂的运算过程，从而在运行时产生该标识号。这个自动产生的值会受到类名称、它所实现的接口的名称、以及所有公有的和受保护的成员的名称所影响。如果你通过任何方式改变了这些信息，比如，增加了一个不是很重要的工具方法，自动产生的序列版本UID也会发生变化。因此，如果你没有声明一个显式的序列版本UID，兼容性将会遭到破坏，在运行时导致InvalidClassException异常。

实现Serializable的第二个代价是，它增加了出现Bug和安全漏洞的可能性。通常情况下，对象是利用构造器来创建的；序列化机制是一种语言之外的对象创建机制（extralinguistic mechanism）。无论你是接受了默认的行为，还是覆盖了默认的行为，反序列化机制（deserialization）都是一个“隐藏的构造器”，具备与其他构造器相同的特点。因为反序列化机制中没有显式的构造器，所以你很容易忘记要确保：反序列化过程必须也要保证所有“由真正的构造器建立起来的约束关系”，并且不允许攻击者访问正在构造过程中的对象的内部信息。依靠默认的反序列化机制，很容易使对象的约束关系遭到破坏，以及遭受到非法访问（见第76条）。

实现Serializable的第三个代价是，随着类发行新的版本，相关的测试负担也增加了。当一个可序列化的类被修订的时候，很重要的一点是，要检查是否可以“在新版本中序列化一个实例，然后在旧版本中反序列化”，反之亦然。因此，测试所需要的工作量与“可序列化的类的数量和发行版本号”的乘积成正比，这个乘积可能会非常大。这些测试不可能自动构建，因为除了二进制兼容性（**binary compatibility**）以外，你还必须测试语义兼容性（**semantic compatibility**）。换句话说，你必须既要确保“序列化－反序列化”过程成功，也要确保结果产生的对象真正是原始对象的复制品。可序列化类的变化越大，它就越需要测试。如果在最初编写一个类的时候，就精心设计了自定义的序列化形式，测试的要求就可以有所降低，但是也不能完全没有测试。

实现**Serializable**接口并不是一个很轻松就可以做出的决定。它提供了一些实在的益处：

更多资源请访问稀酷客(www.ckook.com)

如果一个类将要加入到某个框架中，并且该框架依赖于序列化来实现对象传输或者持久化，对于这个类来说，实现Serializable接口就非常有必要。更进一步来看，如果这个类要成为另一个类的一个组件，并且后者必须实现Serializable接口，若前者也实现了Serializable接口，它就会更易于被后者使用。然而，有许多实际的开销都与实现Serializable接口有关。每当你实现一个类的时候，都需要权衡一下所付出的代价和带来的好处。根据经验，比如Date和BigInteger这样的值类应该实现Serializable，大多数的集合类也应该如此。代表活动实体的类，比如线程池（thread pool），一般不应该实现Serializable。

*为了继承而设计的类（见第17条）应该尽可能少地去实现**Serializable**接口，用户的接口也应该尽可能少地继承Serializable接口。*如果违反了这条规则，扩展这个类或者实现这个接口的程序员就会背上沉重的负担。然而在有些情况下违反这条规则却是合适的。例如，如果一个类或者接口存在的目的主要是为了参与到某个框架中，该框架要求所有的参与者都必须实现Serializable接口，那么，对于这个类或者接口来说，实现或者扩展Serializable接口就是非常有意义的。

在为了继承而设计的类中，真正实现了Serializable接口的有Throwable类、Component和HttpServlet抽象类。因为Throwable类实现了Serializable接口，所以RMI的异常可以从服务器端传到客户端。Component实现了Serializable接口，因此GUI可以被发送、保存和恢复。HttpServlet实现了Serializable接口，因此会话状态（session state）可以被缓存。

如果你实现了一个带有实例域的类，它是可序列化和可扩展的，你就应该担心这样一条告诫。如果类有一些约束条件，当类的实例域被初始化成它们的默认值（整数类型为0，boolean为false，对象引用类型为null）时，就会违背这些约束条件，这时候你就必须给这个类添加这个readObjectNoData方法：

```
// readObjectNoData for stateful extendable serializable classes
private void readObjectNoData() throws InvalidObjectException {
    throw new InvalidObjectException("Stream data required");
}
```

Java 1.4的版本中就增加了这个readObjectNoData方法，还包含了一些冷僻的用例，包括给现有的可序列化类添加可序列化的超类。如果你有兴趣，可以在序列化规范中找到详细的信息[Serialization，3.5]。[⊖]

有一条告诫与“不要实现Serializable接口”有关。如果一个专门为了继承而设计的类不是可序列化的，就不可能编写出可序列化的子类。特别是，如果超类没有提供可访问的无参构造器，子类也不可能做到可序列化。因此，*对于为继承而设计的不可序列化的类，你应该考虑提供一个无参构造器。*这通常并不需要付出特别的努力，因为许多为继承而设计的类都不具有状态，但是情况并不总是这样的。

⊖ Java序列化规范的地址为：http://java.sun.com/jzse/1.5/pdf/serial-1.5.0.pdf或http://java.sun.com/javase/6/docs/platform/serialization/spec/serialToc.heml。——译者注

更多资源请访问稀酷客(www.ckook.com)

最好在所有的约束关系都已经建立的情况下再创建对象（见第15条）。如果为了建立这些约束关系而要求客户端提供一些数据，这实际上就排除了使用无参构造器的可能性。盲目地为一个类增加无参构造器和单独的初始化方法，而它的约束关系仍由其他的构造器来建立，这样做会使该类的状态空间更加复杂，并且增加出错的可能性。

有一种办法可以给“不可序列化但可扩展的类”增加无参构造器，同时避免以上的不足。假设该类有这样一个构造器：

```
public AbstractFoo(int x, int y) { ... }
```

下面的转换增加了一个受保护的无参构造器，和一个初始化方法。初始化方法与正常的构造器具有相同的参数，并且也建立起同样的约束关系。注意保存对象状态（x和y）的变量不能是final的，因为它们是由initialize方法设置的：

```
// Nonserializable stateful class allowing serializable subclass
public abstract class AbstractFoo {
    private int x, y;  // Our state

    // This enum and field are used to track initialization
    private enum State { NEW, INITIALIZING, INITIALIZED };
    private final AtomicReference<State> init =
        new AtomicReference<State>(State.NEW);

    public AbstractFoo(int x, int y) { initialize(x, y); }

    // This constructor and the following method allow
    // subclass's readObject method to initialize our state.
    protected AbstractFoo() { }
    protected final void initialize(int x, int y) {
        if (!init.compareAndSet(State.NEW, State.INITIALIZING))
            throw new IllegalStateException(
                "Already initialized");
        this.x = x;
        this.y = y;
        ... // Do anything else the original constructor did
        init.set(State.INITIALIZED);
    }

    // These methods provide access to internal state so it can
    // be manually serialized by subclass's writeObject method.
    protected final int getX() { checkInit(); return x; }
    protected final int getY() { checkInit(); return y; }
    // Must call from all public and protected instance methods
    private void checkInit() {
        if (init.get() != State.INITIALIZED)
            throw new IllegalStateException("Uninitialized");
    }
    ... // Remainder omitted
}
```

AbstractFoo中所有公有的和受保护的实例方法在开始做任何其他工作之前都必须先调用checkInit。这样可以确保如果有编写不好的子类没有初始化实例，该方法调用就可以快速而干净地失败。注意init域是一个原子引用（**atomic reference**）（java.util. concurrent.atomic.

更多资源请访问稀酷客(www.ckook.com)

AtomicReference)。在遇到特定的情况时，确保对象的完整性是很有必要的。如果没有这样的防范机制，万一有个线程要在某一个实例上调用initialize，而另一个线程又要企图使用这个实例，第二个线程就有可能看到这个实例处于不一致的状态。这种模式利用compareAndSet方法来操作枚举的原子引用，这是一个很好的线程安全状态机（**thread-safe state machine**）的通用实现。如果有了这样的机制做保证，实现一个可序列化的子类就非常简单明了：

```
// Serializable subclass of nonserializable stateful class
public class Foo extends AbstractFoo implements Serializable {
    private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
        s.defaultReadObject();

        // Manually deserialize and initialize superclass state
        int x = s.readInt();
        int y = s.readInt();
        initialize(x, y);
    }
    private void writeObject(ObjectOutputStream s)
            throws IOException {
        s.defaultWriteObject();

        // Manually serialize superclass state
        s.writeInt(getX());
        s.writeInt(getY());
    }

    // Constructor does not use the fancy mechanism
    public Foo(int x, int y) { super(x, y); }

    private static final long serialVersionUID = 1856835860954L;
}
```

内部类（**inner class**）（见第22条）不应该实现**Serializable**。它们使用编译器产生的合成域（**synthetic field**）来保存指向外围实例（**enclosing instance**）的引用，以及保存来自外围作用域的局部变量的值。“这些域如何对应到类定义中”并没有明确的规定，就好像没有指定匿名类和局部类的名称一样。因此，内部类的默认序列化形式是定义不清楚的。然而，静态成员类（**static member class**）却可以实现Serializable接口。

简而言之，千万不要认为实现Serializable接口会很容易。除非一个类在用了一段时间之后就会被抛弃，否则，实现Serializable接口就是个很严肃的承诺，必须认真对待。如果一个类是为了继承而设计的，则更加需要加倍小心。对于这样的类而言，在“允许子类实现Serializable接口”或“禁止子类实现Serializable接口”两者之间的一个折衷设计方案是，提供一个可访问的无参构造器。这种设计方案允许（但不要求）子类实现Serializable接口。

更多资源请访问稀酷客(www.ckook.com)

第75条：考虑使用自定义的序列化形式

当你在时间紧迫的情况下设计一个类时，一般合理的做法是把工作重心集中在设计最佳的API上。有时候，这意味着要发行一个“用完后即丢弃”的实现，因为你知道以后会在新版本中将它替换掉。正常情况下，这不成问题，但是，如果这个类实现了Serializable接口，并且使用了默认的序列化形式，你就永远无法彻底摆脱那个应该丢弃的实现了。它将永远牵制住这个类的序列化形式。这不只是一个纯理论的问题，在Java平台类库中已经有几个类出现了这样的问题，比如BigInteger。

*如果没有先认真考虑默认的序列化形式是否合适，则不要贸然接受。*接受默认的序列化形式是一个非常重要的决定，你需要从灵活性、性能和正确性多个角度对这种编码形式进行考察。一般来讲，只有当你自行设计的自定义序列化形式与默认的序列化形式基本相同时，才能接受默认的序列化形式。

考虑以一个对象为根的对象图，相对于它的物理表示法而言，该对象的默认序列化形式是一种比较有效的编码形式。换句话说，默认的序列化形式描述了该对象内部所包含的数据，以及每一个可以从这个对象到达的其他对象的内部数据。它也描述了所有这些对象被链接起来后的拓扑结构。对于一个对象来说，理想的序列化形式应该只包含该对象所表示的逻辑数据，而逻辑数据与物理表示法应该是各自独立的。

*如果一个对象的物理表示法等同于它的逻辑内容，可能就适合于使用默认的序列化形式。*例如，对于下面这些仅仅表示人名的类，默认的序列化形式就是合理的：

```
// Good candidate for default serialized form
public class Name implements Serializable {
    /**
     * Last name. Must be non-null.
     * @serial
     */
    private final String lastName;

    /**
     * First name. Must be non-null.
     * @serial
     */
    private final String firstName;
    /**
     * Middle name, or null if there is none.
     * @serial
     */
    private final String middleName;

    ... // Remainder omitted
}
```

从逻辑的角度而言，一个名字包含三个字符串，分别代表姓、名和中间名。Name中的实

更多资源请访问稀酷客(www.ckook.com)

例域精确地反映了它的逻辑内容。

即使你确定了默认的序列化形式是合适的，通常还必须提供一个**readObject**方法以保证约束关系和安全性。对于Name这个类而言，readObject方法必须确保lastName和firstName是非null的。第76条和第78条将详细地讨论这个问题。

注意，虽然lastName、firstName和middleInitial域是私有的，但是它们仍然有相应的注释文档。这是因为，这些私有域定义了一个公有的API，即这个类的序列化形式，并且该公有的API必须建立文档。@serial标签告诉Javadoc工具，把这些文档信息放在有关序列化形式的特殊文档页中。

下面的类与Name不同，它是另一个极端，该类表示了一个字符串列表（此刻我们暂时忽略关于“最好使用标准类库中List实现”的建议）：

```
// Awful candidate for default serialized form
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry  next;
        Entry  previous;
    }

    ... // Remainder omitted
}
```

从逻辑意义上讲，这个类表示了一个字符串序列。但是从物理意义上讲，它把该序列表示成一个双向链表。如果你接受了默认的序列化形式，该序列化形式将不遗余力地镜像出（mirror）链表中的所有项，以及这些项之间的所有双向链接。

当一个对象的物理表示法与它的逻辑数据内容有实质性的区别时，使用默认序列化形式会有以下4个缺点：

- **它使这个类的导出API永远地束缚在该类的内部表示法上。**在上面的例子中，私有的StringList.Entry类变成了公有API的一部分。如果在将来的版本中，内部表示法发生了变化，StringList类仍将需要接受链表形式的输入，并产生链表形式的输出。这个类永远也摆脱不掉维护链表项所需要的所有代码，即使它不再使用链表作为内部数据结构了，也仍然需要这些代码。

- **它会消耗过多的空间。**在上面的例子中，序列化形式既表示了链表中的每个项，也表示了所有的链接关系，这是不必要的。这些链表项以及链接只不过是实现细节，不值得记录在序列化形式中。因为这样的序列化形式过于庞大，所以，把它写到磁盘中，或者在

更多资源请访问稀酷客(www.ckook.com)

网络上发送都将非常慢。

- **它会消耗过多的时间**。序列化逻辑并不了解对象图的拓扑关系，所以它必须要经过一个昂贵的图遍历（traversal）过程。在上面的例子中，沿着next引用进行遍历是非常简单的。

- **它会引起栈溢出**。默认的序列化过程要对对象图执行一次递归遍历，即使对于中等规模的对象图，这样的操作也可能会引起栈溢出。在我的机器上，如果StringList实例包含1258个元素，对它进行序列化就会导致栈溢出。到底多少个元素会引发栈溢出，这要取决于JVM的具体实现以及Java启动时的命令行参数，（比如Heap Size的-Xms与-Xmx的值）有些实现可能根本不存在这样的问题。

对于StringList类，合理的序列化形式可以非常简单，只需先包含链表中字符串的数目，然后紧跟着这些字符串即可。这样就构成了StringList所表示的逻辑数据，与它的物理表示细节脱离。下面是StringList的一个修订版本，它包含writeObject和readObject方法，用来实现这样的序列化形式。顺便提醒一下，transient修饰符表明这个实例域将从一个类的默认序列化形式中省略掉：

```
// StringList with a reasonable custom serialized form
public final class StringList implements Serializable {
    private transient int size   = 0;
    private transient Entry head = null;

    // No longer Serializable!
    private static class Entry {
        String data;
        Entry  next;
        Entry  previous;
    }

    // Appends the specified string to the list
    public final void add(String s) { ... }

    /**
     * Serialize this {@code StringList} instance.
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted ({@code int}), followed by all of
     * its elements (each a {@code String}), in the proper
     * sequence.
     */
    private void writeObject(ObjectOutputStream s)
            throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }

    private void readObject(ObjectInputStream s)
            throws IOException, ClassNotFoundException {
```

更多资源请访问稀酷客(www.ckook.com)

```
        s.defaultReadObject();
        int numElements = s.readInt();

        // Read in all elements and insert them in list
        for (int i = 0; i < numElements; i++)
            add((String) s.readObject());
    }

    ... // Remainder omitted
}
```

注意，尽管StringList的所有域都是瞬时的（transient），但writeObject方法的首要任务仍是调用defaultWriteObject，readObject方法的首要任务则是调用defaultReadObject。如果所有的实例域都是瞬时的，从技术角度而言，不调用**defaultWriteObject**和**defaultReadObject**也是允许的，但是不推荐这样做。即使所有的实例域都是transient的，调用defaultWriteObject也会影响该类的序列化形式，从而极大地增强灵活性。这样得到的序列化形式允许在以后的发行版本中增加非transient的实例域，并且还能保持向前或者向后兼容性。如果某一个实例将在未来的版本中被序列化，然后在前一个版本中被反序列化，那么，后增加的域将被忽略掉。如果旧版本的readObject方法没有调用defaultReadObject，反序列化过程将失败，引发StreamCorrupted Exception异常。

注意，尽管writeObject方法是私有的，它也有文档注释。这与Name类中私有域的文档注释是同样的道理。该私有方法定义了一个公有的API，即序列化形式，并且这个公有的API应该建立文档。如同域的@serial标签一样，方法的@serialData标签也告知Javadoc工具，要把该文档信息放在有关序列化形式的文档页上。

套用以前对性能的讨论形式，如果平均字符串长度为10个字符，StringList修订版本的序列化形式就只占用原序列化形式一半的空间。在我的机器上，同样是10个字符长度的情况下，StringList修订版的序列化速度比原版本的快2倍。最终，修订版中不存在栈溢出的问题，因此，对于可被序列化的StringList的大小也没有实际的上限。

虽然默认的序列化形式对于StringList类来说只是不适合而已，对于有些类，情况却变得更加糟糕。对于StringList，默认的序列化形式不够灵活，并且执行效果不佳，但是序列化和反序列化StringList实例会产生对原始对象的忠实拷贝，它的约束关系没有被破坏，从这个意义上讲，这个序列化形式是正确的。但是，如果对象的约束关系要依赖于特定于实现的细节，对于它们来说，情况就不是这样了。

例如，考虑散列表的情形。它的物理表示法是一系列包含“键－值（key-value）”项的散列桶。到底一个项将被放在哪个桶中，这是该键的散列码的一个函数，一般情况下，不同的JVM实现不保证会有同样的结果。实际上，即使在同一个JVM实现中，也无法保证每次运行都会一样。因此，对于散列表而言，接受默认的序列化形式将会构成一个严重的Bug。对散列

更多资源请访问稀酷客(www.ckook.com)

表对象进行序列化和反序列化操作所产生的对象，其约束关系会遭到严重的破坏。

无论你是否使用默认的序列化形式，当defaultWriteObject方法被调用的时候，每一个未被标记为transient的实例域都会被序列化。因此，每一个可以被标记为transient的实例域都应该做上这样的标记。这包括那些冗余的域，即它们的值可以根据其他“基本数据域”计算而得到的域，比如缓存起来的散列值。它也包括那些“其值依赖于JVM的某一次运行”的域，比如一个long域代表了一个指向本地数据结构的指针。*在决定将一个域做成非transient的之前，请一定要确信它的值将是该对象逻辑状态的一部分。*如果你正在使用一种自定义的序列化形式，大多数实例域，或者所有的实例域则都应该被标记为transient，就像上面例子中的StringList那样。

如果你正在使用默认的序列化形式，并且把一个或者多个域标记为transient，则要记住，当一个实例被反序列化的时候，这些域将被初始化为它们的*默认值*（**default value**）：对于对象引用域，默认值为null；对于数值基本域，默认值为0；对于boolean域，默认值为false[JLS, 4.12.5]。如果这些值不能被任何transient域所接受，你就必须提供一个readObject方法，它首先调用defaultReadObject，然后把这些transient域恢复为可接受的值（见第76条）。另一种方法是，这些域可以被延迟到第一次被使用的时候才真正被初始化（见第71条）。

无论你是否使用默认的序列化形式，*如果在读取整个对象状态的任何其他方法上强制任何同步，则也必须在对象序列化上强制这种同步*。因此，如果你有一个线程安全的对象（见第70条），它通过同步每个方法实现了它的线程安全，并且你选择使用默认的序列化形式，就要使用下列的writeObject方法：

```
// writeObject for synchronized class with default serialized form
private synchronized void writeObject(ObjectOutputStream s)
        throws IOException {
    s.defaultWriteObject();
}
```

如果你把同步放在writeObject方法中，就必须确保它遵守与其他动作相同的锁排列（lock-ordering）约束条件，否则就有遭遇资源排列（resource-ordering）死锁的危险[Goetz06, 10.1.5]。

*不管你选择了哪种序列化形式，都要为自己编写的每个可序列化的类声明一个显式的序列版本*UID（**serial version UID**）。这样可以避免序列版本UID成为潜在的不兼容根源（见第74条）。而且，这样做也会带来小小的性能好处。如果没有提供显式的序列版本UID，就需要在运行时通过一个高开销的计算过程产生一个序列版本UID。

要声明一个序列版本UID非常简单，只要在你的类中增加下面一行：

```
private static final long serialVersionUID = randomLongValue ;
```

更多资源请访问稀酷客(www.ckook.com)

在编写新的类时，为randomLongValue选择什么值并不重要。通过在该类上运行serialver工具，你就可以得到一个这样的值，但是，如果你凭空编造一个数值，那也是可以的。如果你想修改一个没有序列版本UID的现有的类，并希望新的版本能够接受现有的序列化实例，就必须使用那个自动为旧版本生成的值。如通过在旧版的类上运行serialver工具[⊖]，可以得到这个数值——被序列化的实例为之存在的那个数值。

如果你想为一个类生成一个新的版本，这个类与现有的类不兼容（**incompatible**），那么你只需修改序列版本UID声明中的值即可。结果，前一版本的实例经序列化之后，再做反序列化时会引发InvalidClassException异常而失败。

总而言之，当你决定要将一个类做成可序列化的时候（见第74条），请仔细考虑应该采用什么样的序列化形式。只有当默认的序列化形式能够合理地描述对象的逻辑状态时，才能使用默认的序列化形式；否则就要设计一个自定义的序列化形式，通过它合理地描述对象的状态。你应该分配足够多的时间来设计类的序列化形式，就好像分配足够多的时间来设计它的导出方法一样（见第40条）。正如你无法在将来的版本中去掉导出方法一样，你也不能去掉序列化形式中的域；它们必须被永久地保留下去，以确保序列化兼容性（serialization compalibility）。选择错误的序列化形式对于一个类的复杂性和性能都会有永久的负面影响。

⊖ serialver用法：serialver [-classpath类路径][-show][类名称…]。——译者注

更多资源请访问稀酷客(www.ckook.com)

第76条：保护性地编写readObject方法

第39条介绍了一个不可变的日期范围类，它包含可变的私有Date域。该类通过在其构造器和访问方法（accessor）中保护性地拷贝Date对象，极力地维护其约束条件和不可变性。下面就是这个类：

```
// Immutable class that uses defensive copying
public final class Period {
    private final Date start;
    private final Date end;

    /**
     * @param  start the beginning of the period
     * @param  end the end of the period; must not precede start
     * @throws IllegalArgumentException if start is after end
     * @throws NullPointerException if start or end is null
     */
    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end   = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0)
            throw new IllegalArgumentException(
                                start + " after " + end);
    }

    public Date start () { return new Date(start.getTime()); }

    public Date end ()   { return new Date(end.getTime()); }

    public String toString() { return start + " - " + end; }

    ... // Remainder omitted
}
```

假设你决定要把这个类做成可序列化的。因为Period对象的物理表示法正好反映了它的逻辑数据内容，所以，使用默认的序列化形式并没有什么不合理的（见第75条）。因此，为了使这个类成为可序列化的，似乎你所需要做的也就是在类的声明中增加“implements Serializable”字样。然而，如果你真的这样做，那么这个类将不再保证它的关键约束了。

问题在于，readObject方法实际上相当于另一个公有的构造器，如同其他的构造器一样，它也要求注意同样的所有注意事项。构造器必须检查其参数的有效性（见第38条），并且在必要的时候对参数进行保护性拷贝（见第39条），同样地，readObject方法也需要这样做。如果readObject方法无法做到这两者之一，对于攻击者来说，要违反这个类的约束条件相对就比较简单了。

不严格地说，readObject是一个“用字节流作为唯一参数”的构造器。在正常使用的情况下，对一个正常构造的实例进行序列化可以产生字节流。但是，当面对一个人工仿造的字节流时，readObject产生的对象会违反它所属的类的约束条件，这时问题就产生了。假设我们仅

更多资源请访问稀酷客(www.ckook.com)

仅在Period类的声明中加上了“implements Serializable”字样。那么，这个不完整的程序将产生一个Period实例，它的结束时间比起始时间还要早：

```
public class BogusPeriod {
    // Byte stream could not have come from real Period instance!
    private static final byte[] serializedForm = new byte[] {
     (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
     0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
     0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
     0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
     0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
     0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
     0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
     0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
     0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
     (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
     0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
     0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
     0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
     0x00, 0x78 };

    public static void main(String[] args) {
        Period p = (Period) deserialize(serializedForm);
        System.out.println(p);
    }

    // Returns the object with the specified serialized form
    private static Object deserialize(byte[] sf) {
        try {
            InputStream is = new ByteArrayInputStream(sf);
            ObjectInputStream ois = new ObjectInputStream(is);
            return ois.readObject();
        } catch (Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

被用来初始化serializedForm的byte数组常量是这样产生的：首先对一个正常的Period实例进行序列化，然后对得到的字节流进行手工编辑。对于这个例子而言，字节流的细节并不重要，但是如果你很好奇的话，可以在《Java™ Object Serialization Specification》[Serialization, 6]中查到有关序列化字节流格式的描述信息。如果你运行这个程序，它会打印出“Fri Jan 01 12:00:00 PST 1999 - Sun Jan 01 12:00:00 PST 1984”。只要把Period声明成可序列化的，就会使我们创建出违反其类约束条件的对象。

为了修正这个问题，你可以为Period提供一个readObject方法，该方法首先调用defaultReadObject，然后检查被反序列化之后的对象的有效性。如果有效性检查失败，readObject方法就抛出一个InvalidObjectException异常，使反序列化过程不能成功地完成：

```
// readObject method with validity checking
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Check that our invariants are satisfied
```

更多资源请访问稀酷客(www.ckook.com)

```
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start +" after "+ end);
}
```

尽管这样的修正避免了攻击者创建无效的Period实例，但是，这里仍然隐藏着一个更为微妙的问题。通过伪造字节流，要想创建可变的Period实例仍是有可能的，做法是：字节流以一个有效的Period实例开头，然后附加上两个额外的引用，指向Period实例中的两个私有的Date域。攻击者从ObjectInputStream中读取Period实例，然后读取附加在其后面的“恶意编制的对象引用”。这些对象引用使得攻击者能够访问到Period对象内部的私有Date域所引用的对象。通过改变这些Date实例，攻击者可以改变Period实例。下面的类演示了这种攻击：

```
public class MutablePeriod {
    // A period instance
    public final Period period;

    // period's start field, to which we shouldn't have access
    public final Date start;

    // period's end field, to which we shouldn't have access
    public final Date end;
    public MutablePeriod() {
        try {
            ByteArrayOutputStream bos =
                new ByteArrayOutputStream();
            ObjectOutputStream out =
                new ObjectOutputStream(bos);

            // Serialize a valid Period instance
            out.writeObject(new Period(new Date(), new Date()));

            /*
             * Append rogue "previous object refs" for internal
             * Date fields in Period. For details, see "Java
             * Object Serialization Specification," Section 6.4.
             */
            byte[] ref = { 0x71, 0, 0x7e, 0, 5 }; // Ref #5
            bos.write(ref); // The start field
            ref[4] = 4;     // Ref # 4
            bos.write(ref); // The end field

            // Deserialize Period and "stolen" Date references
            ObjectInputStream in = new ObjectInputStream(
            new ByteArrayInputStream(bos.toByteArray()));
            period = (Period) in.readObject();
            start  = (Date)   in.readObject();
            end    = (Date)   in.readObject();
        } catch (Exception e) {
            throw new AssertionError(e);
        }
    }
}
```

运行下面的程序，可以看到攻击的效果：

```
public static void main(String[] args) {
    MutablePeriod mp = new MutablePeriod();
    Period p = mp.period;
    Date pEnd = mp.end;
```

更多资源请访问稀酷客(www.ckook.com)

```
    // Let's turn back the clock
    pEnd.setYear(78);
    System.out.println(p);

    // Bring back the 60s!
    pEnd.setYear(69);
    System.out.println(p);
}
```

运行这个程序，产生如下的输出结果：

```
Wed Apr 02 11:04:26 PDT 2008 - Sun Apr 02 11:04:26 PST 1978
Wed Apr 02 11:04:26 PDT 2008 - Wed Apr 02 11:04:26 PST 1969
```

虽然Period实例被创建之后，它的约束条件没有被破坏，但是要随意地修改它的内部组件仍然是有可能的。一旦攻击者获得了一个可变的Period实例，他就可以将这个实例传递给一个“安全性依赖于Period的不可变性”的类，从而造成更大的危害。这种推断并不牵强：实际上，有许多类的安全性就是依赖于String的不可变性。

问题的根源在于，Period的readObject方法并没有完成足够的保护性拷贝。当一个对象被反序列化的时候，对于客户端不应该拥有的对象引用，如果哪个域包含了这样的对象引用，就必须要做保护性拷贝，这是非常重要的。因此，对于每个可序列化的不可变类，如果它包含了私有的可变组件，那么在它的readObject方法中，必须要对这些组件进行保护性拷贝。下面的readObject方法可以确保Period的约束条件不会遭到破坏，以保持它的不可变性：

```
// readObject method with defensive copying and validity checking
private void readObject(ObjectInputStream s)
        throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    // Defensively copy our mutable components
    start = new Date(start.getTime());
    end   = new Date(end.getTime());

    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start +" after "+ end);
}
```

注意，保护性拷贝是在有效性检查之前进行的，而且，我们没有使用Date的clone方法来执行保护性拷贝。这两个细节对于保护Period免受攻击是必要的（见第39条）。同时也要注意到，对于final域，保护性拷贝是不可能的。为了使用readObject方法，我们必须要将start和end域做成非final的。这是很遗憾的，但是这还算是相对比较好的做法。有了这个新的readObject方法，并去掉了start和end域的final修饰符之后，MutablePeriod类将不再有效。此时，上面的攻击程序会产生这样的输出：

```
Wed Apr 02 11:05:47 PDT 2008 - Wed Apr 02 11:05:47 PDT 2008
Wed Apr 02 11:05:47 PDT 2008 - Wed Apr 02 11:05:47 PDT 2008
```

更多资源请访问稀酷客(www.ckook.com)

在Java 1.4发行版本中，为了阻止恶意的对象引用攻击，同时节省保护性拷贝的开销，在ObjectOutputStream中增加了writeUnshared和readUnshared方法[Serialization]。遗憾的是，这些方法都很容易受到复杂的攻击，即本质上与第77条中所述的ElvisStealer攻击相似的攻击。不要使用**writeUnshared**和**readUnshared**方法。它们通常比保护性拷贝更快，但是它们不提供必要的安全性保护。

有一个简单的“石蕊”测试，可以用来确定默认的readObject方法是否可以被接受。测试方法：增加一个公有的构造器，其参数对应于该对象中每个非transient的域，并且无论参数的值是什么，都是不进行检查就可以保存到相应的域中的。对于这样的做法，你是否会感到很舒适？如果你对这个问题的回答是否定的，就必须提供一个显式的readObject方法，并且它必须执行构造器所要求的所有有效性检查和保护性拷贝。另一种方法是，可以使用序列化代理模式（**serialization proxy pattern**），见第78条。

对于非final的可序列化的类，在readObject方法和构造器之间还有其他类似的地方。readObject方法不可以调用可被覆盖的方法，无论是直接调用还是间接调用都不可以（见第17条）。如果违反了这条规则，并且覆盖了该方法，被覆盖的方法将在子类的状态被反序列化之前先运行。程序很可能会失败[Bloch05，Puzzle91]。

总而言之，每当你编写readObject方法的时候，都要这样想：你正在编写一个公有的构造器，无论给它传递什么样的字节流，它都必须产生一个有效的实例。不要假设这个字节流一定代表着一个真正被序列化过的实例。虽然在本条目的例子中，类使用了默认的序列化形式，但是，所有讨论到的有可能发生的问题也同样适用于使用自定义序列化形式的类。下面以摘要的形式给出一些指导方针，有助于编写出更加健壮的readObject方法：

- 对于对象引用域必须保持为私有的类，要保护性地拷贝这些域中的每个对象。不可变类的可变组件就属于这一类别。
- 对于任何约束条件，如果检查失败，则抛出一个InvalidObjectException异常。这些检查动作应该跟在所有的保护性拷贝之后。
- 如果整个对象图在被反序列化之后必须进行验证，就应该使用ObjectInputValidation接口[JavaSE6，Serialization]。
- 无论是直接方式还是间接方式，都不要调用类中任何可被覆盖的方法。

更多资源请访问稀酷客(www.ckook.com)

第77条：对于实例控制，枚举类型优先于readResolve

第3条讲述了Singleton模式，并且给出了以下这个Singleton类的示例。这个类限制了对其构造器的访问，以确保永远只创建一个实例：

```
public class Elvis {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { ... }

    public void leaveTheBuilding() { ... }
}
```

正如在第3条中提到的，如果这个类的声明中加上了“implements Serializable”的字样，它就不再是一个Singleton。无论该类使用了默认的序列化形式，还是自定义的序列化形式（见第75条），都没有关系；也跟它是否提供了显式的readObject方法（见第76条）无关。任何一个readObject方法，不管是显式的还是默认的，它都会返回一个新建的实例，这个新建的实例不同于该类初始化时创建的实例。

readResolve特性允许你用readObject创建的实例代替另一个实例[Serialization, 3.7]。对于一个正在被反序列化的对象，如果它的类定义了一个readResolve方法，并且具备正确的声明，那么在反序列化之后，新建对象上的readResolve方法就会被调用。然后，该方法返回的对象引用将被返回，取代新建的对象。在这个特性的绝大多数用法中，指向新建对象的引用不需要再被保留，因此立即成为垃圾回收的对象。

如果Elvis类要实现Serializable接口，下面的readResolve方法就足以保证它的Singleton属性：

```
// readResolve for instance control - you can do better!
private Object readResolve() {
    // Return the one true Elvis and let the garbage collector
    // take care of the Elvis impersonator.
    return INSTANCE;
}
```

该方法忽略了被反序列化的对象，只返回该类初始化时创建的那个特殊的Elvis实例。因此，Elvis实例的序列化形式并不需要包含任何实际的数据；所有的实例域都应该被声明为transient的。事实上，如果依赖**readResolve**进行实例控制，带有对象引用类型的所有实例域则都必须声明为**transient**的。否则，那种破釜沉舟式的攻击者，就有可能在readResolve方法被运行之前，保护指向反序列化对象的引用，采用的方法类似于在第76条中提到过的MutablePeriod攻击。

这种攻击有点复杂，但是背后的思想却很简单。如果Singleton包含一个非transient的对象

更多资源请访问稀酷客(www.ckook.com)

引用域，这个域的内容就可以在Singleton的readResolve方法运行之前被反序列化。当对象引用域的内容被反序列化时，它就允许一个精心制作的流“盗用”指向最初被反序列化的Singleton的引用。

以下是它更详细的工作原理。首先，编写一个“盗用者”类，它既有readResolver方法，又有实例域，实例域指向被序列化的Singleton的引用，“盗用者”类就“潜伏”在其中。在序列化流中，用“盗用者”类的实例代替Singleton的非transient时域。你现在就有了一个循环：Singleton包含“盗用者”类，“盗用者”类则引用该Singleton。

由于Singleton包含“盗用者”类，当这个Singleton被反序列化时，“盗用者”类的readResolve方法先运行。因此，当“盗用者”的readResolve方法运行时，它的实例域仍然引用被部分反序列化（并且也还没有被解析）的Singleton。

“盗用者”的readResolve方法从它的实例域中将引用复制到静态域中，以便该引用可以在readResolve方法运行之后被访问到。然后这个方法为它所藏身的那个域返回一个正确的类型值。如果没有这么做，当序列化系统试着将“盗用者”引用保存到这个域中时，VM就会抛出ClassCastException。

为了更具体地说明这一点，我们来考虑下面这个有问题的Singleton：

```
// Broken singleton - has nontransient object reference field!
public class Elvis implements Serializable {
    public static final Elvis INSTANCE = new Elvis();
    private Elvis() { }

    private String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
        System.out.println(Arrays.toString(favoriteSongs));
    }

    private Object readResolve() {
        return INSTANCE;
    }
}
```

下面是个“盗用者”类，是根据上述的描述构造的：

```
public class ElvisStealer implements Serializable {
    static Elvis impersonator;
    private Elvis payload;

    private Object readResolve() {
        // Save a reference to the "unresolved" Elvis instance
        impersonator = payload;

        // Return an object of correct type for favorites field
        return new String[] { "A Fool Such as I" };
    }
    private static final long serialVersionUID = 0;
}
```

更多资源请访问稀酷客(www.ckook.com)

最后，这是一个不完整的程序，它反序列化一个手工制作的流，为那个有缺陷的Singleton产生两个截然不同的实例。这个程序中省略了反序列化方法，因为它与第267页中的一样：

```
public class ElvisImpersonator {
    // Byte stream could not have come from real Elvis instance!
    private static final byte[] serializedForm = new byte[] {
     (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x05,
     0x45, 0x6c, 0x76, 0x69, 0x73, (byte)0x84, (byte)0xe6,
     (byte)0x93, 0x33, (byte)0xc3, (byte)0xf4, (byte)0x8b,
     0x32, 0x02, 0x00, 0x01, 0x4c, 0x00, 0x0d, 0x66, 0x61, 0x76,
     0x6f, 0x72, 0x69, 0x74, 0x65, 0x53, 0x6f, 0x6e, 0x67, 0x73,
     0x74, 0x00, 0x12, 0x4c, 0x6a, 0x61, 0x76, 0x61, 0x2f, 0x6c,
     0x61, 0x6e, 0x67, 0x2f, 0x4f, 0x62, 0x6a, 0x65, 0x63, 0x74,
     0x3b, 0x78, 0x70, 0x73, 0x72, 0x00, 0x0c, 0x45, 0x6c, 0x76,
     0x69, 0x73, 0x53, 0x74, 0x65, 0x61, 0x6c, 0x65, 0x72, 0x00,
     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0x00, 0x01,
     0x4c, 0x00, 0x07, 0x70, 0x61, 0x79, 0x6c, 0x6f, 0x61, 0x64,
     0x74, 0x00, 0x07, 0x4c, 0x45, 0x6c, 0x76, 0x69, 0x73, 0x3b,
     0x78, 0x70, 0x71, 0x00, 0x7e, 0x00, 0x02
    };
    public static void main(String[] args) {
        // Initializes ElvisStealer.impersonator and returns
        // the real Elvis (which is Elvis.INSTANCE)
        Elvis elvis = (Elvis) deserialize(serializedForm);
        Elvis impersonator = ElvisStealer.impersonator;

        elvis.printFavorites();
        impersonator.printFavorites();
    }
}
```

运行这个程序会产生下列输出，最终证明可以创建两个截然不同的Elvis实例（包含两种不同的音乐品位）：

```
[Hound Dog, Heartbreak Hotel]
[A Fool Such as I]
```

通过将favorites域声明为transient，可以修正这个问题，但是最好把Elvis做成是一个单元素的枚举类型（见第3条）进行修正。从历史上看，readResolve方法被用于所有可序列化的实例受控（instance-Controlled）的类。自从Java 1.5发行版本以来，它就不再是在可序列化的类中维持实例控制的最佳方法了。就如ElvisStealer攻击所示范的，这种方法很脆弱，需要万分谨慎。

如果反过来，你将一个可序列化的实例受控的类编写成枚举，就可以绝对保证除了所声明的常量之外，不会有别的实例。JVM对此提供了保障，这一点你可以确信无疑。从你这方面来讲，并不需要特别注意什么。以下是把Elvis写成枚举的例子：

```
// Enum singleton - the preferred approach
public enum Elvis {
    INSTANCE;
    private String[] favoriteSongs =
        { "Hound Dog", "Heartbreak Hotel" };
    public void printFavorites() {
```

更多资源请访问稀酷客(www.ckook.com)

```
        System.out.println(Arrays.toString(favoriteSongs));
    }
}
```

用readResolve进行实例控制并不过时。如果必须编写可序列化的实例受控的类，它的实例在编译时还不知道，你就无法将类表示成一个枚举类型。

readResolve的可访问性（accessibility）很重要。如果把readResolve方法放在一个final类上，它就应该是私有的。如果把readResolver方法放在一个非final的类上，就必须认真考虑它的可访问性。如果它是私有的，就不适用于任何子类。如果它是包级私有的，就只适用于同一个包中的子类。如果它是受保护的或者公有的，就适用于所有没有覆盖它的子类。如果readResolve方法是受保护的或者公有的，并且子类没有覆盖它，对序列化过的子类实例进行反序列化，就会产生一个超类实例，这样有可能导致ClassCastException异常。

总而言之，你应该尽可能地使用枚举类型来实施实例控制的约束条件。如果做不到，同时又需要一个既可序列化又是实例受控（instance-controlled）的类，就必须提供一个readResolver方法，并确保该类的所有实例域都为基本类型，或者是transient的。

更多资源请访问稀酷客(www.ckook.com)

第78条：考虑用序列化代理代替序列化实例

正如第74条中提到以及本章中所讨论的，决定实现Serializable接口，会增加出错和出现安全问题的可能性，因为它导致实例要利用语言之外的机制来创建，而不是用普通的构造器。然而，有一种方法可以极大地减少这些风险。这种方法就是*序列化代理模式*（**serialization proxy pattern**）。

序列化代理模式相当简单。首先，为可序列化的类设计一个私有的静态嵌套类，精确地表示外围类的实例的逻辑状态。这个嵌套类被称作*序列化代理*（**serialization proxy**），它应该有一个单独的构造器，其参数类型就是那个外围类。这个构造器只从它的参数中复制数据：它不需要进行任何一致性检查或者保护性拷贝。从设计的角度来看，序列化代理的默认序列化形式是外围类最好的序列化形式。外围类及其序列代理都必须声明实现Serializable接口。

例如，考虑第39条中编写的不可变的Period类，并在第76条中做成可序列化的。以下是这个类的一个序列化代理。Period是如此简单，以致它的序列化代理有着与类完全相同的域：

```
// Serialization proxy for Period class
private static class SerializationProxy implements Serializable {
    private final Date start;
    private final Date end;

    SerializationProxy(Period p) {
        this.start = p.start;
        this.end = p.end;
    }

    private static final long serialVersionUID =
        234098243823485285L; // Any number will do (Item 75)
}
```

接下来，将下面的writeReplace方法添加到外围类中。通过序列化代理，这个方法可以被逐字地复制到任何类中：

```
// writeReplace method for the serialization proxy pattern
private Object writeReplace() {
    return new SerializationProxy(this);
}
```

这个方法的存在导致序列化系统产生一个SerializationProxy实例，代替外围类的实例。换句话说，writeReplace方法在序列化之前，将外围类的实例转变成了它的序列化代理。

有了这个writeReplace方法之后，序列化系统永远不会产生外围类的序列化实例，但是攻击者有可能伪造，企图违反该类的约束条件。为了确保这种攻击无法得逞，只要在外围类中添加这个readObject方法即可：

更多资源请访问稀酷客(www.ckook.com)

```
// readObject method for the serialization proxy pattern
private void readObject(ObjectInputStream stream)
        throws InvalidObjectException {
    throw new InvalidObjectException("Proxy required");
}
```

最后，在SerializationProxy类中提供一个readResolve方法，它返回一个逻辑上相当的外围类的实例。这个方法的出现，导致序列化系统在反序列化时将序列化代理转变回外围类的实例。

这个readResolve方法仅仅利用它的公有API创建外围类的一个实例，这正是该模式的魅力之所在。它极大地消除了序列化机制中语言本身之外的特征，因为反序列化实例是利用与任何其他实例相同的构造器、静态工厂和方法而创建的。这样你就不必单独确保被反序列化的实例一定要遵守类的约束条件。如果该类的静态工厂或者构造器建立了这些约束条件，并且它的实例方法在维持着这些约束条件，你就可以确信序列化也会维持这些约束条件。

以下是上述Period.SerializationProxy的readResolve方法：

```
// readResolve method for Period.SerializationProxy
private Object readResolve() {
    return new Period(start, end);  // Uses public constructor
}
```

正如保护性拷贝方法一样（见第269页），序列化代理方法可以阻止伪字节流的攻击（见第267页）以及内部域的盗用攻击（见第268页）。与前两种方法不同，这种方法允许Period的域为final的，为了确保Period类真正是不可变的（见第15条），这一点很有必要。与前两种方法不同的还有，这种方法不需要太费心思。你不必知道哪些域可能受到狡猾的序列化攻击的威胁，你也不必显式地执行有效性检查，作为反序列化的一部分。

还有一种方法，利用这种方法时，序列化代理模式的功能比保护性拷贝的更加强大。序列化代理模式允许反序列化实例有着与原始序列化实例不同的类。你可能认为这在实际应用中没有什么作用，其实不然。

考虑EnumSet的情况（见第32条）。这个类没有公有的构造器，只有静态工厂。从客户的角度来看，它们返回EnumSet实例，但是实际上，它们是返回两种子类之一，具体取决于底层枚举类型的大小（见第1条，第6页）。如果底层的枚举类型有64个或者少于64个的元素，静态工厂就返回一个RegularEnumSet；否则，它们就返回一个JumboEnumSet。现在考虑这种情况：如果序列化一个枚举集合，它的枚举类型有60个元素，然后给这个枚举类型再增加5个元素，之后反序列化这个枚举集合。当它被序列化的时候，是一个RegularEnumSet实例，但是一旦它被反序列化，它最好是一个JumboEnumSet实例。实际发生的情况正是如此，因为EnumSet使用序列化代理模式。如果你有兴趣，可以看看EnumSet的这个序列化代理，它实际上就这么简单：

更多资源请访问稀酷客(www.ckook.com)

```
// EnumSet's serialization proxy
private static class SerializationProxy <E extends Enum<E>>
        implements Serializable {
    // The element type of this enum set.
    private final Class<E> elementType;

    // The elements contained in this enum set.
    private final Enum[] elements;

    SerializationProxy(EnumSet<E> set) {
        elementType = set.elementType;
        elements = set.toArray(EMPTY_ENUM_ARRAY);  // (Item 43)
    }

    private Object readResolve() {
        EnumSet<E> result = EnumSet.noneOf(elementType);
        for (Enum e : elements)
            result.add((E)e);
        return result;
    }
    private static final long serialVersionUID =
        362491234563181265L;
}
```

序列化代理模式有两个局限性。它不能与可以被客户端扩展的类兼容（见第17条）。它也不能与对象图中包含循环的某些类兼容：如果你企图从一个对象的序列化代理的readResolve方法内部调用这个对象中的方法，就会得到一个ClassCastException异常，因为你还没有这个对象，只有它的序列化代理。

最后，序列化代理模式所增强的功能和安全性并不是没有代价的。在我的机器上，通过序列化代理来序列化和反序列化Period实例的开销，比用保护性拷贝进行的开销增加了14%。

总而言之，每当你发现自己必须在一个不能被客户端扩展的类上编写readObject或者writeObject方法的时候，就应该考虑使用序列化代理模式。要想稳健地将带有重要约束条件的对象序列化时，这种模式可能是最容易的方法。

更多资源请访问稀酷客(www.ckook.com)

附　录

第1版与第2版条目对照

第1版条目	第2版条目
第1条	第1条：考虑用静态工厂方法代替构造器
第2条	第3条：用私有构造器或者枚举类型强化Singleton属性
第3条	第4条：通过私有构造器强化不可实例化的能力
第4条	第5条：避免创建不必要的对象
第5条	第6条：消除过期的对象引用
第6条	第7条：避免使用终结方法
第7条	第8条：覆盖equals时请遵守通用约定
第8条	第9条：覆盖equals时总要覆盖hashCode
第9条	第10条：始终要覆盖toString
第10条	第11条：谨慎地覆盖clone
第11条	第12条：考虑实现Comparable接口
第12条	第13条：使类和成员的可访问性最小化
第13条	第15条：使可变性最小化
第14条	第16条：复合优先于继承
第15条	第17条：要么为继承而设计，并提供文档说明，要么就禁止继承
第16条	第18条：接口优于抽象类
第17条	第19条：接口只用于定义类型
第18条	第22条：优先考虑静态成员类
第19条	第14条：在公有类中使用访问方法而非公有域
第20条	第20条：类层次优于标签类
第21条	第30条：用enum代替int常量
第22条	第21条：用函数对象表示策略
第23条	第38条：检查参数的有效性
第24条	第39条：必要时进行保护性拷贝
第25条	第40条：谨慎设计方法签名
第26条	第41条：慎用重载
第27条	第43条：返回零长度的数组或者集合，而不是null
第28条	第44条：为所有导出的API元素编写文档注释
第29条	第45条：将局部变量的作用域最小化

更多资源请访问稀酷客(www.ckook.com)

（续）

第1版条目	第2版条目
第30条	第47条：了解和使用类库
第31条	第48条：如果需要精确的答案，请避免使用float和double
第32条	第50条：如果其他类型更适合，则尽量避免使用字符串
第33条	第51条：当心字符串连接的性能
第34条	第52条：通过接口引用对象
第35条	第53条：接口优先于反射机制
第36条	第54条：谨慎地使用本地方法
第37条	第55条：谨慎地进行优化
第38条	第56条：遵守普遍接受的命名惯例
第39条	第57条：只针对异常的情况才使用异常
第40条	第58条：对可恢复的情况使用受检异常，对编程错误使用运行时异常
第41条	第59条：避免不必要地使用受检的异常
第42条	第60条：优先使用标准的异常
第43条	第61条：抛出与抽象相对应的异常
第44条	第62条：每个方法抛出的异常都要有文档
第45条	第63条：在细节消息中包含能捕获失败的信息
第46条	第64条：努力使失败保持原子性
第47条	第65条：不要忽略异常
第48条	第66条：同步访问共享的可变数据
第49条	第67条：避免过度同步
第50条	第69条：并发工具优先于wait和notify
第51条	第72条：不要依赖于线程调度器
第52条	第70条：线程安全性的文档化
第53条	第73条：避免使用线程组
第54条	第74条：谨慎地实现Serializable接口
第55条	第75条：考虑使用自定义的序列化形式
第56条	第76条：保护性地编写readObject方法
第57条	第77条：对于实例控制，枚举类型优先于readResolve
	第78条：考虑用序列化代理代替序列化实例

更多资源请访问稀酷客(www.ckook.com)

中英文术语对照

access control　访问控制
accessibility　可访问能力，可访问性
accessor method　访问方法
adapter pattern　适配器模式
annotation type　注解类型
anonymous class　匿名类
antipattern　反模式
API (Application Programming Interface)　应用编程接口
API element　API元素
array　数组
assertion　断言
binary compatibility　二进制兼容性
bit field　位域
bounded wildcard type　有限制的通配符类型
boxed primitive type　基本包装类型
callback　回调
callback framework　回调框架
checked exception　受检异常
class　类
client　客户端
code inspection　代码检验
comparator　比较器
composition　复合
concrete strategy　具体策略
constant interface　常量接口
constant-specific class body　特定于常量的类主体
constant-specific method implementation　特定于常量的方法实现
copy constructor　拷贝构造器
covariant　协变的
covariant return type　协变返回类型
custom serialized form　自定义的序列化形式
decorator pattern　装饰模式
default access　缺省访问
default constructor　缺省构造器
defensive copy　保护性拷贝
delegation　委托
deserializing　反序列化
design pattern　设计模式
documentation comment　文档注释
double-check idiom　双重检查模式，双检法
dynamically cast　动态地转换
encapsulation　封装
enclosing instance　外围实例
enum type　枚举类型
erasure　擦除
exception　异常
exception chaining　异常链
exception translation　异常转换
explicit type parameter　显式的类型参数
exponentiation　求幂
exported API　导出的API
extend　扩展

更多资源请访问稀酷客(www.ckook.com)

failure atomicity 失败原子性
field 域
finalizer guardian 终结方法守卫者
forwarding 转发
forwarding method 转发方法
function object 函数对象
function pointer 函数指针
general contract 通用约定
generic 泛型
generic array creation 泛型数组创建
generic method 泛型方法
generic singleton factory 泛型单例工厂
generic static factory method 泛型静态工厂方法
generification 泛型化
heterogeneous 异构的
idiom 习惯用法，模式
immutable 不可变的
implement 实现（用作动词）
implementation 实现（用作名词）
implementation inheritance 实现继承
information hiding 信息隐藏
inheritance 继承
inner class 内部类
int enum pattern int枚举模式
interface 接口
interface inheritance 接口继承
invariant 不可变的
lazy initialization 延迟初始化
local class 局部类
marker annotation 标记注解
marker interface 标记接口
member 成员
member class 成员类
member interface 成员接口
memory footprint 内存占用
memory model 内存模型
meta-annotation 元注解
method 方法
migration compatibility 移植兼容性
mixin 混合类型
module 模块
mutator 设值方法
naming convention 命名惯例
naming pattern 命名模式
native method 本地方法
native object 本地对象
nested class 嵌套类
non-reifiable 不可具体化的
nonstatic member class 非静态的成员类
object 对象
object pool 对象池
object serialization 对象序列化
obsolete reference 过期引用
open call 开放调用
operation code 操作码
overload 重载
override 覆盖
package-private 包级私有
parameterized type 参数化的类型
performance model 性能模型
postcondition 后置条件
precondition 前提条件
precondition violation 前提违例
primitive 基本类型
private 私有的
public 公有的
raw type 原生态类型
recursive type bound 递归类型限制
redundant field 冗余域
reference type 引用类型
reflection 反射机制
register 注册
reifiable 可具体化的

更多资源请访问稀酷客(www.ckook.com)

reified 具体化的
remainder 求余
restricted marker interface 有限制的标记接口
rounding mode 舍入模式
runtime exception 运行时异常
safety 安全性
scalar type 标量类型
semantic compatibility 语义兼容性
serial version UID 序列版本UID
serialization proxy 序列化代理
serialized form 序列化形式
serializing 序列化
service provider framework 服务提供者框架
signature 签名
singleton 单例
singleton pattern 单例模式
skeletal implementation 骨架实现
state transition 状态转变
stateless 无状态的
static factory method 静态工厂方法
static member class 静态成员类
storage pool 存储池
strategy enum 策略枚举
strategy interface 策略接口
strategy pattern 策略模式
stream unique identifier 流的唯一标识符
subclassing 子类化
subtyping 子类型化
synthetic field 合成域
thread group 线程组
thread safety 线程安全性
thread-safe 线程安全的
top-level class 顶级类，顶层类
type inference 类型推导
type parameter 类型参数
typesafe 类型安全
typesafe enum pattern 类型安全的枚举模式
typesafe heterogeneous container 类型安全的异构容器
unbounded wildcard type 无限制的通配符类型
unchecked exception 未受检异常
unintentional object retention 无意识的对象保持
utility class 工具类
value class 值类
value type 值类型
view 视图
virgin state 空白状态
worker thread 工作线程
wrapper class 包装类

更多资源请访问稀酷客(www.ckook.com)

参考文献

[Arnold05] Arnold, Ken, James Gosling, and David Holmes. *The Java™ Programming Language, Fourth Edition*. Addison-Wesley, Boston, 2005. ISBN: 0321349806.

[Asserts] *Programming with Assertions*. Sun Microsystems. 2002. <http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html>

[Beck99] Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999. ISBN: 0201616416.

[Beck04] Beck, Kent. *JUnit Pocket Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2004. ISBN: 0596007434.

[Bloch01] Bloch, Joshua. *Effective Java™ Programming Language Guide*. Addison-Wesley, Boston, 2001. ISBN: 0201310058.

[Bloch05] Bloch, Joshua, and Neal Gafter. *Java™ Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, Boston, 2005. ISBN: 032133678X.

[Bloch06] Bloch, Joshua. Collections. In *The Java™ Tutorial: A Short Course on the Basics, Fourth Edition*. Sharon Zakhour et al. Addison-Wesley, Boston, 2006. ISBN: 0321334205. Pages 293–368. Also available as <http://java.sun.com/docs/books/tutorial/collections/index.html>.

[Bracha04] Bracha, Gilad. *Generics in the Java Programming Language*. 2004. <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>

更多资源请访问稀酷客(www.ckook.com)

[Burn01] Burn, Oliver. *Checkstyle*. 2001–2007. <http://checkstyle.sourceforge.net>

[Collections] *The Collections Framework*. Sun Microsystems. March 2006. <http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html>

[Gafter07] Gafter, Neal. *A Limitation of Super Type Tokens*. 2007. <http://gafter.blogspot.com/2007/05/limitation-of-super-type-tokens.html>

[Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN: 0201633612.

[Goetz06] Goetz, Brian, with Tim Peierls et al. *Java Concurrency in Practice*. Addison-Wesley, Boston, 2006. ISBN: 0321349601.

[Gong03] Gong, Li, Gary Ellison, and Mary Dageforde. *Inside Java™ 2 Platform Security, Second Edition*. Addison-Wesley, Boston, 2003. ISBN: 0201787911.

[HTML401] *HTML 4.01 Specification*. World Wide Web Consortium. December 1999. <http://www.w3.org/TR/1999/REC-html401-19991224/>

[Jackson75] Jackson, M. A. *Principles of Program Design*. Academic Press, London, 1975. ISBN: 0123790506.

[Java5-feat] *New Features and Enhancements J2SE 5.0*. Sun Microsystems. 2004. <http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html>

[Java6-feat] *Java™ SE 6 Release Notes: Features and Enhancements*. Sun Microsystems. 2008. <http://java.sun.com/javase/6/webnotes/features.html>

[JavaBeans] *JavaBeans™ Spec*. Sun Microsystems. March 2001. <http://java.sun.com/products/javabeans/docs/spec.html>

更多资源请访问稀酷客(www.ckook.com)

[Javadoc-5.0] *What's New in Javadoc 5.0*. Sun Microsystems. 2004. <http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/whatsnew-1.5.0.html>

[Javadoc-guide] *How to Write Doc Comments for the Javadoc Tool*. Sun Microsystems. 2000–2004. <http://java.sun.com/j2se/javadoc/writingdoccomments/index.html>

[Javadoc-ref] *Javadoc Reference Guide*. Sun Microsystems. 2002–2006. <http://java.sun.com/javase/6/docs/technotes/tools/solaris/javadoc.html> <http://java.sun.com/javase/6/docs/technotes/tools/windows/javadoc.html>

[JavaSE6] *Java™ Platform, Standard Edition 6 API Specification*. Sun Microsystems. March 2006. <http://java.sun.com/javase/6/docs/api/>

[JLS] Gosling, James, Bill Joy, and Guy Steele, and Gilad Bracha. *The Java™ Language Specification, Third Edition*. Addison-Wesley, Boston, 2005. ISBN: 0321246780.

[Kahan91] Kahan, William, and J. W. Thomas. *Augmenting a Programming Language with Complex Arithmetic*. UCB/CSD-91-667, University of California, Berkeley, 1991.

[Knuth74] Knuth, Donald. Structured Programming with `go to` Statements. In *Computing Surveys* 6 (1974): 261–301.

[Langer08] Langer, Angelika. *Java Generics FAQs — Frequently Asked Questions*. 2008. <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>

[Lea00] Lea, Doug. *Concurrent Programming in Java™: Design Principles and Patterns, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0201310090.

[Lieberman86] Lieberman, Henry. Using Prototypical Objects to Implement

更多资源请访问稀酷客(www.ckook.com)

Shared Behavior in Object-Oriented Systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223, Portland, September 1986. ACM Press.

[Liskov87] Liskov, B. Data Abstraction and Hierarchy. In *Addendum to the Proceedings of OOPSLA '87* and *SIGPLAN Notices,* Vol. 23, No. 5: 17–34, May 1988.

[Meyers98] Meyers, Scott. *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, Reading, MA, 1998. ISBN: 0201924889.

[Naftalin07] Naftalin, Maurice, and Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., Sebastopol, CA, 2007. ISBN: 0596527756.

[Parnas72] Parnas, D. L. On the Criteria to Be Used in Decomposing Systems into Modules. In *Communications of the ACM* 15 (1972): 1053–1058.

[Posix] 9945-1:1996 (ISO/IEC) [IEEE/ANSI Std. 1003.1 1995 Edition] Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) C Language] (ANSI), IEEE Standards Press, ISBN: 1559375736.

[Pugh01] *The "Double-Checked Locking is Broken" Declaration*. Ed. William Pugh. University of Maryland. March 2001. <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>

[Serialization] *Java™ Object Serialization Specification*. Sun Microsystems. March 2005. <http://java.sun.com/javase/6/docs/platform/serialization/spec/serialTOC.html>

[Sestoft05] Sestoft, Peter. *Java Precisely, Second Edition*. The MIT Press, Cambridge, MA, 2005. ISBN: 0262693259.

[Smith62] Smith, Robert. Algorithm 116 Complex Division.

更多资源请访问稀酷客(www.ckook.com)

In *Communications of the ACM*, 5.8 (August 1962): 435.

[Snyder86] Snyder, Alan. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings*, 38–45, 1986. ACM Press.

[Thomas94] Thomas, Jim, and Jerome T. Coonen. Issues Regarding Imaginary Types for C and C++. In *The Journal of C Language Translation*, 5.3 (March 1994): 134–138.

[ThreadStop] *Why Are `Thread.stop`, `Thread.suspend`, `Thread.resume` and `Runtime.runFinalizersOnExit` Deprecated?* Sun Microsystems. 1999.
<http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html>

[Viega01] Viega, John, and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, Boston, 2001. ISBN: 020172152X.

[W3C-validator] *W3C Markup Validation Service*. World Wide Web Consortium. 2007.
<http://validator.w3.org/>

[Wulf72] Wulf, W. A Case Against the GOTO. In *Proceedings of the 25th ACM National Conference* 2 (1972): 791–797.

更多资源请访问稀酷客(www.ckook.com)

[General Information]
书名=Effective Java中文版
作者=（美）布洛克（Bloch, J.）著
页数=287
出版社=机械工业出版社
出版日期=2009
SS号=12143064
DX号=000006674494
URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000006674494&d=64805EEE8CB92EEFF1A499E975E74CA5

更多资源请访问稀酷客(www.ckook.com)

封面
书名
版权
前言
目录
第１章　引言
第２章　创建和销毁对象
　第１条：考虑用静态工厂方法代替构造器
　第２条：遇到多个构造器参数时要考虑用构建器
　第３条：用私有构造器或者枚举类型强化Ｓｉｎｇｌｅｔｏｎ属性
　第４条：通过私有构造器强化不可实例化的能力
　第５条：避免创建不必要的对象
　第６条：消除过期的对象引用
　第７条：避免使用终结方法
第３章　对于所有对象都通用的方法
　第８条：覆盖ｅｑｕａｌｓ时请遵守通用约定
　第９条：覆盖ｅｑｕａｌｓ时总要覆盖ｈａｓｈＣｏｄｅ
　第１０条：始终要覆盖ｔｏＳｔｒｉｎｇ
　第１１条：谨慎地覆盖ｃｌｏｎｅ
　第１２条：考虑实现Ｃｏｍｐａｒａｂｌｅ接口
第４章　类和接口
　第１３条：使类和成员的可访问性最小化
　第１４条：在公有类中使用访问方法而非公有域
　第１５条：使可变性最小化
第１６条：复合优先于继承
　第１７条：要么为继承而设计，并提供文档说明，要么就禁止继承
　第１８条：接口优于抽象类
　第１９条：接口只用于定义类型
　第２０条：类层次优于标签类
　第２１条：用函数对象表示策略
　第２２条：优先考虑静态成员类
第５章　泛型
　第２３条：请不要在新代码中使用原生态类型
　第２４条：消除非受检警告
　第２５条：列表优先于数组
　第２６条：优先考虑泛型
　第２７条：优先考虑泛型方法
　第２８条：利用有限制通配符来提升ＡＰＩ的灵活性
　第２９条：优先考虑类型安全的异构容器
第６章　枚举和注解
　第３０条：用ｅｎｕｍ代替ｉｎｔ常量
　第３１条：用实例域代替序数
　第３２条：用ＥｎｕｍＳｅｔ代替位域
　第３３条：用ＥｎｕｍＭａｐ代替序数索引
第３４条：用接口模拟可伸缩的枚举
第３５条：注解优先于命名模式
第３６条：坚持使用Ｏｖｅｒｒｉｄｅ注解
第３７条：用标记接口定义类型
　第７章　方法
　　第３８条：检查参数的有效性
　　第３９条：必要时进行保护性拷贝
　　第４０条：谨慎设计方法签名
　　第４１条：慎用重载
　　第４２条：慎用可变参数
　　第４３条：返回零长度的数组或者集合，而不是ｎｕｌｌ
　　第４４条：为所有导出的ＡＰＩ元素编写文档注释
　第８章　通用程序设计
　　第４５条：将局部变量的作用域最小化
　　第４６条：ｆｏｒ－ｅａｃｈ循环优先于传统的ｆｏｒ循环
　　第４７条：了解和使用类库
　　第４８条：如果需要精确的答案，请避免使用ｆｌｏａｔ和ｄｏｕｂｌｅ
　　第４９条：基本类型优先于装箱基本类型
　　第５０条：如果其他类型更适合，则尽量避免使用字符串
　　第５１条：当心字符串连接的性能
　　第５２条：通过接口引用对象
　　第５３条：接口优先于反射机制
　　第５４条：谨慎地使用本地方法

更多资源请访问稀酷客(www.ckook.com)

第５５条：谨慎地进行优化
第５６条：遵守普遍接受的命名惯例
第９章　异常
第５７条：只针对异常的情况才使用异常
第５８条：对可恢复的情况使用受检异常，对编程错误使用运行时异常
第５９条：避免不必要地使用受检的异常
第６０条：优先使用标准的异常
第６１条：抛出与抽象相对应的异常
第６２条：每个方法抛出的异常都要有文档
第６３条：在细节消息中包含能捕获失败的信息
第６４条：努力使失败保持原子性
第６５条：不要忽略异常
第１０章　并发
第６６条：同步访问共享的可变数据
第６７条：避免过度同步
第６８条：ｅｘｅｃｕｔｏｒ和ｔａｓｋ优先于线程
第６９条：并发工具优先于ｗａｉｔ和ｎｏｔｉｆｙ
第７０条：线程安全性的文档化
第７１条：慎用延迟初始化
第７２条：不要依赖于线程调度器
第７３条：避免使用线程组
第１１章　序列化
第７４条：谨慎地实现Ｓｅｒｉａｌｉｚａｂｌｅ接口
第７５条：考虑使用自定义的序列化形式
第７６条：保护性地编写ｒｅａｄＯｂｊｅｃｔ方法
第７７条：对于实例控制，枚举类型优先于ｒｅａｄＲｅｓｏｌｖｅ
第７８条：考虑用序列化代理代替序列化实例
附录　第１版与第２版条目对照
中英文术语对照
参考文献

更多资源请访问稀酷客(www.ckook.com)