Download at http://www.pin5i.com/

个很有用的变体，就是将特定类型的类主体的method声明为abstract，并为每个枚举值的method定义值所特有的实现。如果类型特有的method为abstract，编译器就会强迫你为类型里的每个枚举值实现那个method，要不小心省略这个实现是不可能的。请注意，即使类型特有的类主体包含了abstract method，但枚举类型的整体也并不是abstract（而且也可能不会被声明为abstract），因为每个值所特有的类主体实现了method。

以下的程序代码是大型范例的摘录，它使用枚举类型来代表基于堆栈的仿真CPU的操作码。Opcode枚举类型定义了一个abstract method perform()，它其后由类型的各个值的类主体所实现。此类型包含了一个构造函数来说明每个枚举值的完整语法：名称、构造函数自变量以及类主体。enum语法要求枚举值以及它的类主体要先出现。然而，如果略掉这些值先查看类型特有的类主体时，程序代码就会非常容易了解：

```
// 这些是我们的堆栈机器可以执行的操作码
public enum Opcode {
    // 将单一操作数推入堆栈
    PUSH(1) {
        public void perform(StackMachine machine, int[] operands) {
            machine.push(operands[0]);
        }
    },    // 要记得使用逗号来分隔枚举值

    // 将堆栈最上面的两个值加起来并推出结果
    ADD(0) {
        public void perform(StackMachine machine, int[] operands) {
            machine.push(machine.pop() + machine.pop());
        }
    },

    /* 为了让程序简单点，其他的opcode值已被省略 */

    // 如果等于零就进入分支
    BEZ(1) {
        public void perform(StackMachine machine, int[] operands) {
            if (machine.top() == 0) machine.setPC(operands[0]);
        }
    };    // 请记住在类主体之前要有分号

    // 这是类型的构造函数
    Opcode(int numOperands) { this.numOperands = numOperands; }

    int numOperands;  // 它预期会有多少个整数操作数?

    // 每个opcode常量都必须在值所特有的类主体中实现
    // 此abstract method用来执行它所代表的操作
    public abstract void perform(StackMachine machine, int[] operands);
}
```

Download at http://www.pin5i.com/

值所特有的类主体的使用时机

当每个枚举值都必须执行唯一的某种运算时，值所特有的类主体就会有非常强大的语言特性。但请记住，值所特有的类主体是高级的特性，它不常被使用，而且对于经验较少的程序员可能会造成混淆。在你决定要使用这个特性之前，请先确定那是必要的。

在使用值所特有的类主体之前，要确定你的设计对于此特性不会太简单也不会太复杂。首先，确认你真的需要值所特有的行为，而不是单纯的值所特有的数据。值所特有的数据可以被编码于构造函数自变量里，就像稍早的 Prefix 范例所显示的。例如，把那个范例改写为使用值所特有的 abbrev() method 版本是不必要也不恰当的。

接下来，想一下枚举类型是否充分符合你的需要。如果你的设计需要具有复杂实现的值所特有的 method 或每个值需要很多 method 时，你会发现把所有东西都写在单一类型中是很繁琐的。我们可以考虑使用传统的 class 与 interface 声明来声明你自己的自定义类型层次以及任何你需要的单独实例。

如果值所特有的行为在枚举类型的结构里真的需要，那么值所特有的类主体就会适合。值所特有的主体是很精致还是会令人混淆，这是见仁见智的问题，有些程序员会倾向于尽可能避免使用。有个对一些人具有吸引力的替代方案，就是在使用 switch 语句来将每个值作为个别 case 的类型特有的 method 中编码值特有的行为。以下的 enum 里的 compute() method 就是一个例子。此枚举类型的单纯性使得 switch 语句成为值所特有的类主体的替代方案：

```
public enum ArithmeticOperator {
    // 枚举值
    ADD, SUBTRACT, MULTIPLY, DIVIDE;

    // 使用了 switch 语句的值所特有的行为
    public double compute(double x, double y) {
        switch(this) {
        case ADD:       return x + y;
        case SUBTRACT:  return x - y;
        case MULTIPLY:  return x * y;
        case DIVIDE:    return x / y;
        default: throw new AssertionError(this);
        }
    }

    // 使用此 enum 的测试案例
    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for(ArithmeticOperator op : ArithmeticOperator.values())
            System.out.printf("%f %s %f = %f%n", x, op, y, op.compute(x,y));
    }
}
```

Download at http://www.pin5i.com/

使用switch语句的方法有个缺点，就是每次你增加新的枚举值时，就必须要记得在switch语句里加入相对应的case语句。而且如果有一个以上的method使用了switch语句，你就必须同时维护它们的switch语句。如果忘记使用switch语句来实现值所特有的行为，就会导致运行时的AssertionError。在值所特有的类主体覆盖类型特有的类主体内的abstract method时，同样的省略会造成编译错误，但可以被尽快修正。

值所特有的method和类型特有的method中的switch语句在性能上相当类似。在一个状况中的虚拟method调用的耗费，会与在另一个状况中switch语句的耗费平衡。值所特有的类主体会造成额外类的产生，就存储空间与加载时间来看，这些都会成为额外的负担。

enum类型上的限制

Java对于能在枚举类型里出现的程序代码作了一些限制。实际上，你不会常常遇到这些限制，但你还是应该要知道。

当你定义一个枚举类型时，编译器在私下做了很多的事情：它会创建一个扩展了java.lang.Enum的类，并且产生values()与valueOf() method以及保存枚举值的静态字段。如果你为类型加上类主体，就不应该加上名称会与自动产生的成员名称冲突或是与继承自Enum的final method冲突的成员。

enum类型不可以被声明为final。枚举类型是一个有效的final，而且编译器不允许你去扩展enum。但是如果enum包含了值所特有的类主体，那么针对enum所产生的类文件在技术上就不会被声明为final。

Java里的类型不可以同时为final与abstract。因为枚举类型为final，它们不可以被声明为abstract。如果enum声明的类型特有的类主体包含了abstract method，编译器就会要求每个枚举值要有值所特有的类主体，其中包含了abstract method的实现。就单独的个体来看，以这个方式定义的枚举类型并不是abstract。

构造函数（也就是实例字段的初始化程序）和枚举类型的实例初始化程序块受到了全面但模糊的限制：它们不可以使用类型的静态字段（包含枚举值本身）。这是因为枚举类型（以及所有的类型）的静态初化始的进行方式是由上而下。枚举值是出现在类型顶端的静态字段而且会先被初始化。因为它们都是自行输入（self-typed）的字段，所以它们会调用构造函数以及该类型的其他实例初始化程序代码。这代表实例初始化程序代码是在类的静态初始化完成之前被调用的。因为静态字段尚未被初始化，所以编译器并不允许它们被使用。唯一的例外是静态字段的值是编译器可解析的编译时常量表达式（例如integer与string）。

Download at http://www.pin5i.com/

如果你为枚举类型定义了构造函数，那么它不可以使用 super()关键字来调用超类构造函数，这是因为编译器会自动地将隐藏的 name 与 ordinal 自变量插入你所定义的构造函数里。如果你为该类型定义了多个构造函数，那么使用 this()来从一个构造函数调用另一个构造函数是可行的。请记住，个别的枚举值（如果你定义了）的类主体是匿名的，这代表它们根本不可能会有任何的构造函数。

具有类型安全性的 enum 模式

为了更深入地了解 enum 关键字的运作方式，或是要模拟 Java 5.0 之前的枚举类型，了解具有安全性的enum 模式是非常有用的。此模式在 Joshua Block（注 6）的《Effective Java Programming Language Guide》（Addison Wesley）一书中有精彩的描述，在这里我们不会介绍太多。

如果你想要使用 Java 5.0 之前的枚举类型 Prefix（在本章稍早介绍过），可以使用和以下类似的类来模拟它。但请注意，这个类的实例将无法和 switch 语句或 EnumSet 与 EnumMap 类一起运行。此外，这里所显示的程序代码并没有包含编译器为真正的枚举类型自动产生的 value()或 valueOf() method。像这样的类不会有像枚举类型所提供的特殊序列化（serialization）支持，所以如果你让它可序列化，就必须提供 readResolve() method来预防创建枚举值的多个实例时的反序列化(deserialization)。

```
public final class Prefix
    // 这些是自行输入的常量
    public static final Prefix MILLI = new Prefix("m",     .001);
    public static final Prefix CENTI = new Prefix("c",     .01);
    public static final Prefix DECI  = new Prefix("d",     .1);
    public static final Prefix DECA  = new Prefix("D",   10.0);
    public static final Prefix HECTA = new Prefix("h",  100.0);
    public static final Prefix KILO  = new Prefix("k", 1000.0);

    // 让这些成为专用字段，以让实例永远都不会改变
    private String name;
    private double multiplier;

    // 这是专用构造函数，所以除了上面的实例以外，没有实例会被创建
    private Prefix(String name, double multiplier) {
        this.name = name;
        this.multiplier = multiplier;
    }

    // 这些访问 method 是公共的
    public String toString() { return name; }
```

注 6： Josh 是 JSR 201 委员会里的联合主席，该委员会开发了 Java 5.0 里的许多新的语言特点。同时，他也是枚举类型的创造者与幕后推动者。

Download at http://www.pin5i.com/


```
    public double getMultiplier() { return multiplier; }
}
```

注释

注释（annotation）提供了一个将任意信息或元数据（metatdata）与程序元素结合在一起的方式。在语法上，注释的使用就像修饰符一样，可以用于包、类型、构造函数、method、字段、参数及局部变量的声明上。存储于注释中的信息采用了name=value对的格式，其类型是由注释类型所指定的。注释类型是一种interface，它们也可以通过Java Reflection API 来提供访问。

注释可被用来将任何你想要的信息与程序元素结合在一起。唯一的基本规则就是注释不会对程序的运行造成任何的影响：即使你增加或移除注释，程序代码还是一样地运行。换个方式说，就是Java解释器会忽略注释（虽然我们可以在运行时通过Java Reflection API 来取得注释）。因为 Java VM 会忽略注释，所以注释类型是没有用的，除非它伴随着可以将信息存储于注释类型里的工具。本章我们将会介绍标准的注释与 meta-annotation 类型，就像Override与Target一样。与这些类型相关的工具就是Java编译器，它必须以特定方式来处理它们（本节在稍后会做说明）。

我们可以很容易就猜想到，可以使用很多个注释（注7）。一个局部变量可能会用名为NonNull的类型来做注释，就像是在断言该变量不会有null值一样。相关（假设的）程序代码分析工具接着会解析程序代码而且会尝试验证该断言。JDK 包含了一个名为 *apt*（Annotation Processing Tool，注释处理工具）的工具，它为注释处理工具提供了一个框架：它会针对注释扫描该源代码并特别调用你所提供的已写好的注释处理器类（annotation processor class）。更多关于*apt*的说明请参阅第八章。你将会发现注释在企业级的程序设计里是最常被用到的，在那里它们可以取代像 *XDoclet* 这样的工具，它用来处理被内嵌在任意*javadoc* 注释里的元数据。

本节将从介绍与注释相关的术语开始。然后会介绍Java 5.0 里的标准注释类型，你可以立刻将*javac* 所提供的注释使用在你的程序代码里。接下来，我们会说明编写注释的语法，同时还会简短地说明在运行时使用Java Reflection API 来查询注释。之后，我们要介绍较难以理解的部分 —— 定义新的注释类型，不过很少有程序员会这么做。本章最后的部分则是讨论 meta-annotation。

注7： 这样的使用方式很快就会有。在编写这段时，JSR 250 通过了 Java Community Process 的方式来为 J2SE 与 J2EE 定义一些标准的常用注释。

Download at http://www.pin5i.com/

注释的概念与术语

注释的最主要的概念就是注释是将信息或元数据与程序元素结合在一起。注释并不会影响到Java程序的运行方式，但它们会影响到像是编译器的警告或辅助工具的行为，例如文件产生器、stub generator 等。

在讨论注释时以下的术语经常会被用到。特别重要的是注释（annotation）与注释类型（annotation type）之间的差异。

注释

注释把信息或元数据与 Java 程序元素结合在一起。注释使用了在 Java 5.0 里所介绍的新的语法与运行方式，就像修饰符一样，例如public或final。每一个注释都有一个名字以及零或多个成员。每一个成员都有一个名称和一个值，而这些name=value 对就带有注释的信息。

注释类型

注释的名称和被注释类型所定义的名称、类型以及它的成员的默认值一样。注释类型是一个 Java 接口，此接口对于它的成员有一些限制，同时在使用它的声明时也有一些新的语法。当你使用 Java Reflection API 来查询注释时，它的返回值是一个对象，该对象实现了注释类型的接口而且也允许查询各个注释成员。Java 5.0在java.lang包里包含了三个标准的注释类型，我们将在本章稍后的“使用标准注释”一节里看到这些注释。

注释成员

注释成员被声明在注释类型里就像无自变量的method一样。method名称与返回类型定义了该成员的名称与类型。特殊的default语法允许针对任一个注释成员声明默认值。注释会出现在含有name=value对的程序元素里，它为所有没有默认值的注释成员定义值，而且也包含了覆盖其他成员的默认值。

标志注释（marker annotation）

没有定义任何成员的注释类型就被称为标志注释（marker annotation）。这类注释所带有的信息就是它出现与否。

meta-annotation

用来声明注释类型的注释就称为 meta-annotation。Java 5.0 在 java.lang.annotation包里包含了几个标准的meta-annotaion类型，它们被用来指定该注释所适用的程序元素。

target

target 类型的注释是程序元素的注释。此注释适用于包、类型（类、接口、枚举类型甚至是注释类型）、类型成员（method、构造函数、字段与枚举值）、method 参

Download at http://www.pin5i.com/

数以及局部变量（包括循环变量以及catch参数）。注释类型的声明可以包含一个meta-annotation，它限制了该注释类型允许的目标。

retention

注释的 *retention* 说明包含在注释里的信息可以被保留多长的时间。有些注释会被编译器丢弃而且只会出现在源代码里，其余的都会被编译进类文件。这些被编译进类文件的注释中，有些会被虚拟机所忽略，而有些则会在包含注释的类被加载时被虚拟机所读取。注释类型的声明可以使用 meta-annotation 来为该类型的注释指定它的retention。可被虚拟机加载的注释在运行时是可见的（runtime-visible），而且可以被 java.lang.reflect 的 reflective API 所查询。

元数据

当讨论到注释时，元数据（metadata）这个术语通常是指注释所隐含的信息或该注释本身。因为这个术语在计算机程序设计书籍里被使用在很多不同的地方，所以在本章中会尽量避免使用它。

使用标准注释

Java 5.0在java.lang包里定义了三个标准的注释类型。以下的小节就是说明这三个注释类型以及解释如何使用它们来注释你的程序代码。

Override

java.lang.Override是一个标志注释类型，可以用它来帮没有其他程序元素的method做注释。这种类型的注释是作为一个 assertion 提供的，这个有注释的 method 会覆盖超类的 method。如果你在 method 里使用了这个注释，且该 method 并没有覆盖其超类的 method，则编译器会对这种情形产生一个编译错误并告知你。

这个注释是针对一般的程序设计错误种类而设计的，这样的错误是当你试图覆盖超类 method 时发生 method 名称或签名错误。在这种情况下，你也许已经重载了 method 名称，但实际上并没有覆盖该 method，而你的程序代码并没被调用到。

要使用这个注释类型，只要在 method 放置修饰符的地方将 @Override包含进来就可以了。依据惯例，@Override出现在修饰符之前，而在 @ 字符与Override名称之间不可以有空白，虽然这在技术上是允许的。请注意，因为 java.lang包会自动地被导入，所以在使用这个注释类型时你并不需要将这个包包含在里面。这里有一个在 method 里使用 @Override 注释的范例，它并没有正确地覆盖其超类的 toString() method。

```
@Override
public String toSting() {    // 请注意，这儿有拼错的词!
```

Download at http://www.pin5i.com/


```
        // 用方括号括住超类的输出
        return "[" + super.toString() + "]";
    }
```

在没有注释的情况下，单词的拼写错误可能会被忽略，而我们也可能会遇到难以理解的错误：为什么 toString() method 不能正确运行？但因为有了注释，编译器给我们的答复是：toString() method 不会如预期般地运行，因为实际上它并没有被覆盖。

请注意，@Override 注释只可以应用于打算要覆盖其超类的 method 的 method，而不可用于实现定义于接口中的 method。如果你没有正确地实现接口的 method，编译器就会产生错误。

Deprecated

java.lang.Deprecated是一个标志注释，它和@deprecated这个javadoc tag非常的相似（请看第七章关于编写Java文件注释的更详细的介绍）。如果你使用@Deprecated来帮一个类型或类型成员做注释，那么它告诉编译器使用有注释的元素是不被允许的。如果你在程序代码中使用（或是扩展、覆盖）deprecated 类型或成员，而该类型或成员并不是它们自己所声明的@Deprecated，则编译器会产生警告。

请注意，@Deprecated 注释类型并不会反对 @deprecated 这个 javadoc tag。@Deprecated注释是为Java 编译器而设计的。就另一方面而言，javadoc tag 是提供给 *javadoc* 工具使用且作为说明文件提供的，它也包含了为什么程序元素不被使用以及它被什么东西所代替或可以用什么东西来取代它等描述。

在 Java 5.0 里，编译器会继续去寻找 @deprecated 这个 javadoc tag，同时使用它们来产生警告信息。然而，这样的用法已经过时了，除了 @deprecated 之外，你应该开始使用 @Deprecated 注释。

这里有个同时使用了注释与 javadoc tag 的范例：

```
/**
 * Sony Betamax 录像带格式
 * @deprecated 这种格式的播放器已经没有人有了，取而代之的是 VHS。
*/
@Deprecated public class Betamax { ... }
```

SuppressWarnings

@SuppressWarnings注释会选择性地关闭类、method 或字段以及变量初始化程序的编

Download at http://www.pin5i.com/

译器所产生的警告（compiler warning）（注 8）。在 Java 5.0 里，Sun 的 *javac* 编译器有一个很强大的 -Xlint 选项，它会使得你的程序代码产生关于“lint”的警告，这是合法的，但比较像是要表示程序设计上的错误。例如，当你使用一个没有指明类型参数值的泛型 collection 类时，这些包含了“未校验的警告（unchecked warning）”的警告就会出现；或是在 switch 语句里有一个 case 没有使用 break、return 或 throw 来结束而且允许控制权落到下一个 case 时，这样的警告也会出现。

一般来说，当你从编译器里看到了这些 lint 警告时，就应该要去调查产生这些警告的程序代码。如果真的是错误，就必须要加以更正。如果它是个很简略的程序，你可以依你自己的习惯重新编写程序代码，这样就不会再有警告了。例如，如果该警告信息是在告诉你并没有将枚举类型所有可能的值都表示在 siwtch 语句的 case 子句里，那么你可以在 switch 语句里加入具有保护作用的 default case 来防止警告的发生，即使你很确定该 default case 不会被调用到。

另一方面，有时候要避免这样的错误时你并不需要做任何的事情。例如，如果你在程序代码里使用了一个泛型 collection 类，而该类必须与之前的非泛型的程序代码交互时，你就无法避免这个未校验的警告信息的产生。这是 @SuppressWarnings 可以出现的地方：将这个注释加到最接近的相关修饰符（一般来说是 method 修饰符）的地方，以告诉编译器你意识到了这个问题，且它应该停止在该问题上对你的纠缠。

和 Override 与 Deprecated 不一样，SuppressWarnings 并不是个标志注释。它有一个名为 value 且类型为 String[] 的单一成员。此成员的值是被隐藏的警告的名称。SuppressWarings 注释并没有定义被允许的警告名称，这对编译器实现而言是一个问题。对 *javac* 编译器来说，可以被 -Xlint 选项所接受的警告名称对 @SuppressWarings 注释来说也是合法的。而指明任何你所想要的警告名称也是合法的，编译器会忽略（但可能会有警告）它们无法识别的警告名称。

所以，为了防止名称为 unchecked 与 fallthrough 的警告发生，你可以使用和下面一样的注释。注释的语法是在该注释类型的后面加上一个小括号，并使用逗号来将 name=value 对隔开。在这种情况下，SuppressWarnings 注释类型只定义了单一成员，所以在小括号内只有一对值。因为成员值是一个数组，所以使用大括号来将数组元素括在里面：

```
@SuppressWarnings(value={"unchecked","fallthrough"})
public void lintTrap() { /* 省略了简略的method主体 */ }
```

注 8： 在编写本章时，*javac* 编译器并没有支持 @SuppressWarnings 注释。此注释预计会在 Java 5.1 里得到支持。

Download at http://www.pin5i.com/

我们可以将这个注释缩写。当注释只有一个单一成员（single member）且该成员的名称为“value”时，你可以省略注释里的“value=”。所以以上的注释可以被重写成：

```
@SuppressWarnings({"unchecked","fallthrough"})
```

希望你在任何method里都不常遇到一些无法解决的lint警告，而只需要去防止一个单一名称的警告的发生。在这种情况下，还有另一种注释缩写式。当在写一个只包含单一成员的数组值时，你可以将大括号省略。在这种情况下，我们有像这样的注释写法：

```
@SuppressWarnings("unchecked")
```

注释语法

在标准注释类型的说明里，我们已经看过marker annotation与single-member annotation的语法了，还包括了当single member的名称为“value”时的缩写以及当数组类型（array-typed）成员只有单一数组元素时的缩写法。这节将会说明 annotation 的完整语法。

注释的组成是在@字符后加上注释类型的名称（可以包含一个包的名称）以及紧跟在之后的一对小括号，同时在小括号里将该注释类型所定义的每一个成员用逗号将name=value对的列表分隔开。如果该注释类型为该成员定义了默认值，则这些成员可能会出现在任何序列里或被忽略。每个 *value* 都必须是直接量或编译时常量、嵌套注释或数组。

在本章的结尾处，我们定义了一个名为Reviews的注释类型，该类型有一个单一的成员，且该成员是一个@Review注释的数组。此Review注释类型有三个成员：“reviewer”是个字符串，“comment”是个有默认值且非必要的字符串，而“grade”是个嵌套枚举类型Review.Grade的一个值。假设Reviews与Review类型都被正确地导入，则使用这些类型的注释看起来就会像这样（请注意这个注释里使用了嵌套注释、枚举类型以及数组）：

```
@Reviews({   // 只有一个值的注释，所以在这里“value=”被省略了
    @Review(grade=Review.Grade.EXCELLENT,
            reviewer="df"),
    @Review(grade=Review.Grade.UNSATISFACTORY,
            reviewer="eg",
            comment="This method needs an @Override annotation")
})
```

另一个很重要的注释语法规则是没有程序元素会有一个以上的相同注释的实例。这是不合法的，例如，在类里放入多个@Review注释。这就是为什么@Review注释被定义为允许@Review注释的数组。

Download at http://www.pin5i.com/

注释成员的类型与值

注释成员的值必须是非null的编译时常量表达式，它必须与成员的声明类型兼容。被允许的成员类型有基本类型、String、Class、枚举类型、注释类型以及以上任何类型的数组（但不可以是数组的数组）。例如，表达式2*Math.PI以及"hello"+"world"是合法的值，且成员的类型分别为double与String。

在本章结尾处我们定义了一个名为UncheckedExceptions的注释类型，它唯一的成员是一个扩展自RuntimeException的类的数组。此类型的注释看起来会像这样：

```
@UncheckedExceptions({
    IllegalArgumentException.class, StringIndexOutOfBoundsException.class
})
```

注释目标

注释最常被置于类型定义（例如类）以及它们的成员里面（例如method与字段），也可以出现在包、参数以及局部变量里。本节为这些较不常见的注释目标提供了较多的信息。

包的注释（package annotation）出现在名为*package-info.java*的文件里的package声明之前。这个文件不应该包含任何类型的声明（"package-info"并不是一个合法的Java标识符，所以它不能包含任何公共的类型定义），它应该包含选择性的javadoc comment、零个或多个注释以及package声明。例如：

```
/**
 * 此包拥有我们自定义的注释类型
 */
@com.davidflanagan.annotations.Author("David Flanagan")
package com.davidflanagan.annotations;
```

当*package-info.java*文件被编译后，它会产生一个名为*package-info.class*的类文件，该文件包含了一个虚构的接口声明。此接口没有成员且它的名称package-info并不是一个合法的Java标识符，所以它无法被使用于Java源代码里。它的存在对有类或运行时保留的包注释而言就像是一个占位符一样。

请注意，包注释出现在任何package或import声明的作用范围之外。这意味着该包注释应该包含该注释类型（除非该包是java.lang）的包名。

在method参数、catch子句参数以及局部变量中出现的注释，对这些程序元素来说就像是修饰符列表的一部分。Java类文件的格式并没有为在局部变量或catch子句参数上存储注释来做准备，所以这些注释都会被保留在源代码里。然而，method参数的注释可以被保留在类文件里且会拥有类或运行时保留。

Download at http://www.pin5i.com/

最后，请注意枚举类型定义的语法不允许为枚举值指定任何的修饰符。然而，它允许任何的值有注释。

注释与默认

注释必须为每一个被注释类型所定义且没有默认值的成员赋一个值。当然，注释也可以和其他的成员一样将值置于注释里面。

了解默认值是如何被处理的是一个很重要的细节。默认值会被存储于该注释类型的类文件里，而它并没有被编译进注释。如果你修改了注释类型以至于它的成员改变了默认值，则这个改变将会影响所有没有为该成员指定明确值的注释类型。而已被编译过的注释也会受到影响，即使在改变该类型后它们并没有被重新编译过。

注释与 Reflection

在 Java 5.0 里，`java.lang.reflect` 的 Reflection API 已经被扩展来支持运行时可见的注释（如果它的注释类型被指定为 runtime retention，那么注释只有在运行时是可见的，也就是说，注释被存储于类文件里，所以当类文件被加载时，注释可以被 Java VM 所读取）。本节会简短地说明新的反射能力。对注释来说，`AnnotatedElement` 表示一个可以被查询的程序元素。它可以由 `java.lang.Package`、`java.lang.Class` 实现，也可以间接地由 `Method`、`Constructor` 以及 `java.lang.reflect` 的 `Field` 类所实现。在 method 参数里的注释可以使用 `Method` 或 `Constructor` 类的 `getParameterAnnotations()` method 来查询。

以下的程序代码使用了 `AnnotatedElement` 的 `isAnnotationPresent()` method，以通过检查 `@Unstable` 注释来判定 method 是否稳定。它假设该 `Unstable` 注释类型（我们将在本章的稍后定义）具有运行时 retention。请注意，此程序代码使用了类直接量来指定被检查的类以及用来做检查的注释：

```
import java.lang.reflect.*;
Class c = WhizzBangClass.class;
Method m = c.getMethod("whizzy", int.class, int.class);
boolean unstable = m.isAnnotationPresent(Unstable.class);
```

对标志注释来说，`isAnnotationPresent()` 是有用的。一般来说，当它与拥有成员的注释一起运行时，我们会想要知道这些成员的值。针对这点，我们可以使用 `getAnnotation()` method。同时，在这里我们也看到了 Java 注释系统的优点：如果被指定的注释存在的话，通过此 method 返回的对象实现该注释类型的接口，而且你也可以通过调用定义该成员的注释类型 method 来查询任何成员的值。例如，考虑本章稍早

Download at http://www.pin5i.com/

所出现的@Reviews注释。如果注释类型被声明为runtime retention，那么你可以用以下的方式来查询：

```
AnnotatedElement target = WhizzBangClass.class; // 此类型是用来做查询的
// 要求@Reviews注释是一个实现Reviews的对象
Reviews annotation = target.getAnnotation(Reviews.class);
// Reviews只有一个单一的名为"value"的成员，它是由reviews组成的数组
Review[] reviews = annotation.value();
// 循环处理reviews
for(Review r : reviews) {
    Review.Grade grade = r.grade();
    String reviewer = r.reviewer();
    String comment = r.comment();
    System.out.printf("%s assigned a grade of %s and comment '%s'%n",
                      reviewer, grade, comment);
}
```

请注意，这些反射method可以为你正确地解决默认注释值的问题。如果注释没有为成员提供默认值，则该默认值可以在注释类型里被查询到。

定义注释类型

注释类型是一个接口，但并不是正常的接口。注释类型与正常的接口有以下几个不同的地方：

- 注释类型是使用关键字@interface来定义，而不是interface。@interface的声明会隐含地扩展java.lang.annotation.Annotation接口，且在它的定义里不能有它自己的extends子句。
- 注释类型的method必须被声明为没有任何的自变量而且不可以抛出异常。这些method定义了注释成员：method的名称变成了成员的名称，同时method的返回类型变成了成员的类型。
- 注释method的返回值可以是基本类型、String、Class、枚举类型、另一个注释类型或是这些类型的一维数组。
- 注释类型的任何method后面可以紧跟着关键字default以及与该method返回类型兼容的值。这个新的语法指定了与该method相应的注释成员的默认值。在写注释的时候，默认值的语法与用来指定成员值的语法是一样的。null不是一个合法的默认值。
- 注释类型与它们的method不可以有类型参数注释类型（parameters-annotation type），而且它的成员不可以是泛型。在注释类型里唯一可以合法使用泛型的是返回类型为Class的method。这些method可以使用有限制的通配符来指明返回类的约束。

Download at http://www.pin5i.com/

在其他方式中，注释类型使用了@interface来声明就像一般的接口一样。它可以包含常量的定义以及静态成员类型的定义，例如枚举类型的定义。注释类型也可以像正常的接口一样被实现与扩展（然而，实现或扩展该类与接口的结果并不是它们自己的注释类型：注释类型只可以被创建于@interface的声明里）。

现在我们在自己的范例里定义注释类型。这些范例说明了注释类型声明的语法并显示了@interface与interface之间的多处不同点。我们从简单的标志注释类型Unstable开始，因为在本章稍早的一个reflection范例里使用到这个类型，它的定义包含了一个meta-annotation，会将它设定为runtime retention并让它可以使用reflection API。下面会对meta-annotation做说明。

```
package com.davidflanagan.annotations;
import java.lang.annotation.*;

/**
 * 指定被注释的元素是不稳定的而且它的API会改变
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Unstable {}
```

接下来的注释类型定义了一个单一的成员。通过命名成员value，我们就能对每个使用注释的程序使用更简短的语句。

```
/**
 * 指明程序元素的作者
 */
public @interface Author {
    /** 返回作者的名字 */
    String value();
}
```

再下来的一个范例就有点复杂了。Reviews注释类型有一个单一的成员，但成员类型有点复杂：它是Review注释组成的数组。Review注释类型有三个成员，其中一个具有被定义成Review类型本身的成员的枚举类型，另一个具有默认值。因为Reviews注释类型被使用在reflection范例里，所以我们也用meta-annotation来给予它runtime retention：

```
import java.lang.annotation.*;

/**
 * 此类型的注释为注释元素指定了一个或多个程序代码review的结果
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Reviews {
    Review[] value();
}
```

Download at http://www.pin5i.com/


```
/**
* 此类型的注释表示该注释元素的单一程序代码review。
* 每一个review必须指明reviewer的名称并为程序代码指定级别。
* review也可以包含一个注释（comment）字符串
*/
public @interface Review {
    // 嵌套的枚举类型
    public static enum Grade { EXCELLENT, SATISFACTORY, UNSATISFACTORY };

    // 这些method定义了注释成员
    Grade grade();                    // Grade类型里一个名为“grade”的成员
    String reviewer();
    String comment() default "";   // 请注意，默认值在这里
}
```

最后，假设我们想要为method做注释以列出它们可能抛出的未校验的异常（不是错误）。我们的注释类型将会有一个数组类型的单一成员，该数组中的每一个元素将会是异常的Class。为了只使用未校验的异常（uncheckecd exception）这个必要条件，我们在Class里使用了一个受限制的通配符：

```
public @interface UncheckedExceptions {
    Class<? extends RuntimeException>[] value();
}
```

Meta-Annotation

注释类型本身可以被注释。Java 5.0定义了四个标准的meta-annotation类型，它们提供了关于其他注释类型的使用与意义的信息。这些类型以及它们所支持的类都在java.lang.annotation包里。

Target

Target meta-annotation类型为注释类型指定了“目标（target）”，也就是它指定了哪一个程序元素可以有该类型的注释。如果一个注释类型没有Target meta-annotation，那么它就可以与稍早所介绍的任何程序成员一起使用。然而，有些注释类型只有在当它被用于某个程序元素里时才具有意义。Override就是一个例子：当它被使用于一个method时才具有意义。当@Target meta-annotation被用于Override类型的声明时会让它更明确，而且当它出现在不适当的地方时，编译器会拒绝@Override。

Target meta-annotation类型有一个名为value的单一成员。此成员的类型为java.lang.annotation.ElementType[]。ElementType是一个枚举类型，它的枚举值代表了可以被注释的程序元素。

Download at http://www.pin5i.com/

Retention

稍早我们在本章已经讨论过注释保留（annotation retention）了。它明确地指出注释是被编译器抛弃或被保留于类文件中，以及如果它被保留在类文件里，则当该类文件被加载时，它是否会被JVM所读取。在默认的状态下，注释是被存储于类文件里，但不能在运行时取得这些注释。这三个可选的保留值为source、class与runtime，说明于枚举类型java.lang.annotation.RetentionPolicy中。

Retention meta-annotation类型有一个名为value的单一成员，它的类型为RetentionPolicy。

Documented

Documented是个meta-annotation类型，用来说明其他类型的注释。它应该被视为该注释的程序元素的公共API的一部分，因此它可以被像*javadoc*这样的工具做成说明文件。Documented是个标记注释，它没有成员。

Inherited

@Inherited metat-annotation 是个标志注释，它说明了注释类型是会被继承的。也就是说，如果注释类型@Inherited被用来帮类做注释时，那么此注释一样适用于该类的子类。

请注意，@Inherited注释类型只可以被注释类的子类所继承。类并不会从它们所实现的接口里继承注释，而且 method 也不会从它们所覆盖的 method 里继承注释。

如果@Inherited注释类型也是一个有注释的@Retention(RetentionPolicy.RUNTIME)时，则 Reflection API 会去执行继承。如果你使用了 java.lang.reflect 来为一个@Inherited类型的注释查询类时，则reflection程序代码会去检查那个被指明的类以及它的每一个父类，直到那个被指明类型的注释被找到或已经到达类层次结构的最顶层。

Download at http://www.pin5i.com/

第五章

Java 平台

第二、三、四章说明了Java程序设计语言。本章转移到说明Java平台 —— 为每个Java程序提供了大量可用的预定义类，无论它在运行时所在的底层主机系统是哪一种。Java平台的类被组织成相关的组，也就是包（package）。本章从Java平台的包的概述开始，接着会以简短范例的形式来示范这些包中最有用的类。大部分的范例都只是程序代码片段，而不是可以编译并运行的完整程序。关于完全充实、真实世界的范例，请参阅《Java Examples in a Nutshell》（O'Reilly）。那本书大幅扩充了本章的内容，可作为本书的参考书籍。

Java 平台概述

表5-1归纳了本书中涉及的Java平台的重要包。

表5-1：Java平台的重要包

包	说明
`java.io`	用于输入和输出的类和接口。虽然此包中有一些类是用来直接处理文件，但大部分是用来处理字节或字节流
`java.lang`	Java语言的核心类，例如`String`、`Math`、`SystemThread`以及`Exception`
`java.lang.annotation`	annontation类型和其他用于Java 5.0 annotation特性的支持类型（请参阅第四章）
`java.lang.instrument`	针对Java虚拟机安装（instrumentation）代理程序的支持类，代理程序可修改JVM运行中程序的字节码。这是Java 5.0中新引入的

Download at http://www.pin5i.com/

表 5-1：Java 平台的重要包（续）

包	说明
java.lang.management	这是用于监控与管理运行中的Java 虚拟机的结构，是Java 5.0 中新引入的
java.lang.ref	这些是定义对对象弱引用（weak reference）的类。弱引用并无法避免被引用对象被内存回收
java.lang.reflect	能让 Java 程序通过查看类的构造函数、method 和字段来回顾其自身的类和接口
java.math	这是个小型的包，包含可用于任意精确度的整数和浮点运算的类
java.net	用于实现与其他系统建立网络连接的类和接口
java.nio	针对 New I/O API 的缓冲类。这是 Java 1.4 中新增的
java.nio.channels	针对高性能、非阻隔性 I/O 的 channel 和 selector 接口与类
java.nio.charset	用于在Unicode字符串和字节之间转换的字符组编码器与译码器
java.security	用于访问控制与认证的类与接口。此包及其子包支持加密消息摘要与数字签名
java.text	用于处理国际化应用程序中文本的类和接口
java.util	各种公用程序类，其中包括用于处理大量对象的强大收集框架
java.util.concurrent	用于并行程序设计的线程共享区和其他的公用程序类，子包支持 atomic 变量和锁定。这是 Java 5.0 新增的
java.util.jar	用于读取与写入 JAR 文件的类
java.util.logging	具有灵活性的登录工具。这是 Java 1.4 中新增的
java.util.prefs	用来读取与写入用户与系统的参数选择的 API。这是 Java 1.4 中新增的
java.util.regex	使用了正则表达式表达式的文本模式匹配。这是 Java 1.4 中新增的
java.util.zip	用于读取与写入 ZIP 文件的类
javax.crypto	针对数据加密与解密的类与接口
javax.net	定义用于建立 socket 和 server socket 的 factory class，以供建立有别于默认的 socket 类型
javax.net.ssl	使用 Secure Sockets Layer（SSL）来加密网络通信的类
javax.security.auth	用于认证与授权的JAAS API的顶层包，各种子包掌握了大部分的实际类。这是 Java 1.4 中新增的

Download at http://www.pin5i.com/

表 5-1：Java 平台的重要包（续）

包	说明
javax.xml.parsers	用于解析使用了可插拔 DOM 或 SAX 解析器的 XML 文件的高级 API
javax.xml.transform	这是个高级 API，用于转换使用了可插拔 XSLT 转换引擎和在流、DOM 树、SAX 事件之间转换 XML 文件，子包对 DOM、SAX 和流转换提供了支持。这是 Java 1.4 中新增的

表5-1并没有列出Java平台中的所有包，只列出本书中说明的最重要的包。Java也定义了许多针对制图和图形用户界面设计以及用于分布式、企业或计算的包。制图和GUI包是 java.awt 和 javax.swing 以及它们的许多子包。这些包的说明文档在《Java Foundation Classes in a Nutshell》和《Java Swing》中有介绍，这两本书都是由O'Reilly出版。Java的企业包包括了 java.rmi、java.sql、javax.jndi、org.omg.CORBA、org.omg.cosNaming、以及它们的所有子包。这些包以及对Java平台的好几种标准扩展在《Java Enterprise in a Nutshell》（O'Reilly）中有介绍。

文本

大部分的程序都会以多种形式来操作文本，而Java平台针对表示、格式化以及扫描文本定义了一些重要的类和接口。以下的章节提供了一个概述。

String 类

由文本组成的字符串是基本而常被使用的数据类型。但是在 Java 中，字符串并不像 char、int 和 float 这样的基本类型。字符串类型是由 java.lang.String 类来定义的，它定义了许多用于操作字符串的有用 method。String 对象是永远不变的：一旦 String 对象被创建，就无法修改它所表示的文本字符串。因此，在字符串上操作的每个 method 通常会返回一个保存已修改字符串的新 String 对象。

下面的程序代码展示了一些你可以在字符串上执行的基本操作：

```
// 创建字符串
String s = "Now";                    // String 对象具有特殊的直接量语法
String t = s + " is the time.";      // 使用 + 运算符来连接字符串
String t1 = s + " " + 23.4;          // 运算符 + 会把其他类型的值转换为字符串类型
t1 = String.valueOf('c');            // 取得对应每个字符的字符串
t1 = String.valueOf(42);             // 取得字符串类型的整数或任一个值
t1 = object.toString();              // 使用 toString() 将对象转为字符串
```

Download at http://www.pin5i.com/


```
// 字符串长度
int len = t.length();                // 字符串中的字符个数为16

// 字符串的子字符串
String sub = t.substring(4);         // 返回从位置4开始到结尾的字符串的
                                        子串"is the time."
sub = t.substring(4, 6);             // 返回位置4和5的字符"is"
sub = t.substring(0, 3);             // 返回位置0到2的字符"Now"
sub = t.substring(x, y);             // 返回介于位置x和y-1之间的字符串的子串
int numchars = sub.length();         // 子字符串的长度一定是(y-x)

// 从字符串中提取字符
char c = t.charAt(2);                // 取得字符串t的第3个字符w
char[] ca = t.toCharArray();         // 将字符串转换为字符数组
t.getChars(0, 3, ca, 1);             // 将字符串t的前三个字符放入ca[1]-ca[3]

// 大小写转换
String caps = t.toUpperCase();       // 转换为大写
String lower = t.toLowerCase();      // 转换为小写

// 字符串比较
boolean b1 = t.equals("hello");            // 若两字符串不相等，返回false
boolean b2 = t.equalsIgnoreCase(caps);     // 忽略大小写的比较，若相等则返回true
boolean b3 = t.startsWith("Now");          // 返回true
boolean b4 = t.endsWith("time.");          // 返回true
int r1 = s.compareTo("Pow");               // 返回值< 0，则s出现在"Pow"之前
int r2 = s.compareTo("Now");               // 返回0，则两字符串相等
int r3 = s.compareTo("Mow");               // 返回值>0，则s在"Mow"之后
r1 = s.compareToIgnoreCase("pow");         // 返回值<0（适用于Java 1.2及之后的版本）

// 查找字符与子字符串
int pos = t.indexOf('i');            // 第一个"i"的位置：4
pos = t.indexOf('i', pos+1);         // 下一个"i"的位置：12
pos = t.indexOf('i', pos+1);         // 字符串中不再有"i"，返回-1
pos = t.lastIndexOf('i');            // 字符串中最后一个"i"的位置：12
pos = t.lastIndexOf('i', pos-1);     // 从位置11的字符开始由后往前查找

pos = t.indexOf("is");               // 查找子字符串：返回4
pos = t.indexOf("is", pos+1);        // 只出现一次：返回-1
pos = t.lastIndexOf("the ");         // 由后往前查找字符串
String noun = t.substring(pos+4);    // 取出"the"之后的单词

// 以一个字符取代另一个字符
String exclaim = t.replace('.', '!');  // 只能处理字符，不能处理子字符串

// 清除字符串开头和结尾的空格
String noextraspaces = t.trim();

// 使用intern()取得字符串的唯一实例
String s1 = s.intern();              // 返回与s相等的字符串s1
String s2 = "Now";                   // 字符串直接量会自动被保留
boolean equals = (s1 == s2);         // 现在可以用==来比较是否相等
```

Download at http://www.pin5i.com/

Character 类

就如你所知道的，每个字符在Java中是以基本的char类型来表示的。此外，Java平台还义了Character类，它包含了用来检查字符类型和用于转换字符大小写的有用的类method。例如：

```
char[] text;   // 由字符组成的数组，会在其他地方被初始化
int p = 0;     // 当前我们在字符数组中的位置
// 略过前面的空白
while((p < text.length) && Character.isWhitespace(text[p])) p++;
// 将文本的第一个字改为大写
while((p < text.length) && Character.isLetter(text[p])) {
  text[p] = Character.toUpperCase(text[p]);
  p++;
}
```

StringBuffer 类

由于String对象是永远不变的，所以你不可以操作已实例化的String字符。如果你必须这么做，可以用java.lang.StringBuffer或java.lang.StringBuilder来代替。除了StringBuffer具有同步化method之外，这两个类是完全一样的。StringBuilder是在Java 5.0中引入的，你应该用它来取代StringBuffer，除非实际上有多个线程在操作StringBuffer。以下的程序代码示范了StringBuffer API，不过改变为使用StringBuilder也很容易：

```
// 根据字符串创建字符串缓冲区
StringBuffer b = new StringBuffer("Mow");

// 取得并修改StringBuffer的单独字符
char c = b.charAt(0);          // 返回"M"，就像String.charAt()
b.setCharAt(0, 'N');           // b包含子串"Now"，但是不能对字符串做这样的操作!

// 添加内容至StringBuffer
b.append(' ');                 // 添加一个字符
b.append("is the time.");      // 添加一个字符串
b.append(23);                  // 添加一个整数或任何其他类型的值

// 将String或其他类型的值插入StringBuffer中
b.insert(6, "n't");            // b现在包含子串"Now isn't the time.23"

// 用一个字符串来代替一组字符（适用于Java 1.2及之后的版本）
b.replace(4, 9, "is");         // 字会串b恢复为 "Now is the time.23"

// Delete characters
b.delete(16, 18);              // 删除一组字符"Now is the time"
b.deleteCharAt(2);             // 删除第2个字符，字符串b变为"No is the time"
b.setLength(5);                // 设定字符串的长度以截短字符串，返回值为"No is"
```

Download at http://www.pin5i.com/


```
// 其他有用的操作
b.reverse();                    // 逆序排列字符 "si oN"
String s = b.toString();        // 转换为不可变动的字符串
s = b.substring(1,2);           // 取得子字符串 "i"
b.setLength(0);                 // 清空缓冲区，现在它已可再被使用
```

CharSequence 接口

在 Java 1.4 中，String 和 StringBuffer 类实现了 java.lang.CharSequence 接口，那是个标准接口，用于从具有可读性的一连串字符中查询长度、提取字符及后续项目。此接口也是由 java.nio.CharBuffer 接口实现的，那是 Java 1.4 中引入的 New I/O API 的一部分。CharSequence 提供了在由字符组成的字符串上执行简单操作的方式，而不管那些东西的底层实现。例如：

```
/**
 * 返回特定 CharSequence 的前缀，由序列的第一个字符
 * 开始，扩展至（并包括）序列中第一个出现的字符 c。
 * 如果 c 没有出现就返回 null。s 可以是 String、
 * StringBuffer 或 java.nio.CharBuffer。
 */
public static CharSequence prefix(CharSequence s, char c) {
  int numChars = s.length();            // 此序列有多长?
  for(int i = 0; i < numChars; i++) {   // 逐个处理序列中的字符
    if (s.charAt(i) == c)               // 如果找到 c,
      return s.subSequence(0,i+1);      // 就返回前缀子序列
  }
  return null;                          // 否则，就返回 null
}
```

Appendable 接口

Appendable 是 Java 5.0 中的接口，它表示对象可以添加一个字符或 CharSequence。实现的类包括了 StringBuffer、StringBuilder、java.nio.CharBufer、java.io.PrintStream 和 java.io.Writer 及其字符输出流子类，其中包括 PrintWriter。因此，Appendable 接口表示文本缓冲区类和文本输出流类的可添加性。就如我们在下面看到的，Formatter 对象可以把它的输出送到任一个 Appendable 对象。

String 连接

+运算符可以连接两个 String 对象或一个 String 对象与一些其他类型的值并产生新的 String 对象。要知道，每当执行字符串连接而且结果被存储于变量中或被传送给 method 时，新的 String 对象就会被创建。在某些情况中，这可能会没有效率而且造成性能不佳。在循环中进行字符串连接时更是要多加小心。例如，以下的程序代码就没有效率：

Download at http://www.pin5i.com/


```
// 无效率，别这么做
public String join(List<String> words) {
    String sentence = "";
    // 每一轮都会创建新的 String 对象并舍弃旧的
    for(String word: words) sentence += word;
    return sentence;
}
```

当你发现自己写出了像是这样的程序代码时，就应使用StringBuffer或StringBuilder并使用 append() method：

```
// 这是正确的方法
public String join(List<String> words) {
    StringBuilder sentence = new StringBuilder();
    for(String word: words) sentence.append(word);
    return sentence.toString();
}
```

但是，对字符串连接有所偏见是不必要的。请记住，字符串直接量是由编译器连接的，而不是由Java 解释器连接的。此外，当字符串表达式包含多个字符串连接时，使用StringBuilder（或Java 5.0之前所提供的StringBuffer）在编译上较有效率，并且只会创建出一个新的String 对象。

字符串比较

由于字符串是对象而不是基本的值，所以一般说来，不能用 == 运算符来比较它们。== 会比较引用值，并且可以判定两个已引用的表达式是否为相同的字符串。它无法判定两个独立的字符串是否包含相同的文字。如果要这么做，可以用 equals() method。在Java 5.0中，你可以用 contentEquals() method 来将一个字符串的内容与任何其他的CharSequence比较。

同样地，< 和 > 关系运算符也不可用于字符串。如果要比较字符串的顺序，可使用compareTo() method，那是由Comparable<String>接口所定义的，并在上面的范例程序代码中说明过。如果在比较字符串时不考虑字母的大小写，可以用compareToIgnoreCase()。

请注意，StringBuffer和StringBuilder没有实现Comparable，而且没有覆写默认版本的 equals()和hashCode()，那是它们从Object 继承来的。这表示要对两个StringBuffer或StringBuilder包含的文字作相等性或顺序性比较是不可能的。

String类中有个很重要但很少被了解的method，那就是intern()。当字符串 s 被传递给这个 method时，它会返回保证与 s 具有相同内容的字符串。但重要的是，对于任

Download at http://www.pin5i.com/

意一个给定的字符串的内容，它一定会返回对相同 String 对象的引用。也就是说，如果 s 和 t 是两个如 s.equals(t) 这样的 String 对象，则：

```
s.intern() == t.intern()
```

这代表 intern() method 使用 == 提供了一个进行快速比较字符串的方法。重要的是，字符串直接量一定会由Java VM隐含地保留，所以如果你计划比较字符串 s 和一些字符串直接量，你或许会想要先保留 s，然后再用 == 作比较。

字符串类的 compareTo()和 equals() method 能让你比较字符串。compareTo()是以由 Unicode 编码的字符顺序作为比较基础，而 equal()把字符串的相等定义为严格的逐字符相等。但是在使用上，这些未必都是适当的method。在某些语言中，Unicode标准所定的字符排序，与以字母顺序排序字符串时的字排序不同。例如，在西班牙文中，字母“ch”会被认为是出现在“c”之后、“d”之前的独立字母。在国际化应用程序中比较人类可阅读的字符串时，你应该用 java.text.Collator 类来代替：

```
import java.text.*;

// 比较两个字符串，结果取决于运行程序的地方
// Collator.compare()的返回值具有与String.compareTo()相同的意义
Collator c = Collator.getInstance();        // 取得当前本地的Collator
int result = c.compare("chica", "coche"); // 用它来比较两个字符串
```

增补字符

Java 5.0 首次采用了 Unicode 4.0 标准，它定义了落在 char 类型的 16 位范围之外的码点（codepoint）。当处理这些“增补字符（supplementary characters）”时（这些主要是中文），你必须使用 int 值来代表各个字符。在 String 对象或是任何其他将文字表示成一连串 char 值的类型中，这些增补字符会被表示成一连串的双 char 值，也就是所谓的 *surrogate pair*。

虽然本书的英文版读者不太可能遇到增补字符，但如果你要处理的程序可能会被本地化以用于中国或其他使用中文的国家时，就应该要有所了解。为了帮助你处理增补字符，Character、String、StringBuffer 以及 StringBuilder 类已用新 method 来加以扩充，那些 method 是在 int 码点上操作，而不是在 char 值上。以下程序代码说明了其中的一些 method。你可以在在线的 javadoc 说明文件中阅读其他类似的 method 的相关信息。

```
int codepoint = 0x10001;  // 此码点无法被放入一个char中
// 取得此码点的char的UTF-16代理对
char[] surrogatePair = Character.toChars(codepoint);
// 将char转换为字符串
String s = new String(surrogatePair);
```

Download at http://www.pin5i.com/


```
// 列出以字符和码点计算的字符串长度
System.out.println(s.length());
System.out.println(s.codePointCount(0, s.length()-1));

// 先列出第一个字符的编码，接着列出第一个码点的编码
System.out.println(Integer.toHexString(s.charAt(0)));
System.out.println(Integer.toHexString(s.codePointAt(0)));

// 这是安全处理可能包含增补字符的字符串方式
String tricky = s + "Testing" + s + "!";
int i = 0, n = tricky.length();
while(i < n) {
    // 取得当前位置的码点
    int cp = tricky.codePointAt(i);
    if (cp < '\uffff') System.out.println((char) cp);
    else System.out.println("\\u" + Integer.toHexString(cp));

    // 以一个码点（一或两个字符）来递增字符串索引
    i = tricky.offsetByCodePoints(i, 1);
}
```

用 printf()和 format()来格式化文字

处理文字输出时的常见工作，就是把各种类型的值结合成人能看懂的独立文字块。完成此工作的方法之一，就是依赖Java字符串连接运算符的字符串转换能力。那会产生像这样的程序代码：

```
System.out.println(username + " logged in after " + numattempts +
                   "attempts. Last login at: " + lastLoginDate);
```

Java 5.0 引进了C程序员熟悉的替代方法：printf() method。“printf”是“print formatted”的缩写，它把打印与格式化功能结合在一个调用中。printf() method已被加到Java 5.0中的PrintWriter和PrintStream输出流类。它是个预期会有一个或多个自变量的varargs method。第一个自变量是“格式字符串”，它指定了要被打印的文字，通常包含了一个或多个“格式限定符（format specifier)”，那是以字符%开始的序列。printf()的其余自变量要被转换为字符串的值，并且会替换格式字符串中的格式限定符。格式限定符限制了其余自变量的类型，并且精确地指定了它们被转换为字符串的方式。上面显示的字符串连接在Java 5.0中可以被改写如下：

```
System.out.printf("%s logged in after %d attempts. Last login at: %tc%n",
                  username, numattempts, lastLoginDate);
```

格式限定符%s只是单纯替换一个字符串；%d预期相对应的自变量会是整数并加以显示；%tc预期相对应的值是Date、Calendar或毫秒数，并把那个值转换为完整日期和时间的文字表达式；%n不会执行转换，它只是输出平台特有的行结束字符，就如println() method 所做的。

Download at http://www.pin5i.com/

printf()所执行的转换都会被恰当地本地化。例如，时间和日期会以符合当地习惯的标点符号来显示。而且，如果你要求数值加上千分位符号显示，也会得到各地用的不同的标点符号（例如，在英国是逗号，在法国是句号）。

除了基本的printf() method，PrintWriter和PrintStream也定义了名为format()的同义method：它采用完全相同的自变量，行为方式也完全相同。在Java 5.0中，String类也有 format() method。除了把已格式化的字符串输出到串流外，此静态String.format() method 的行为就像 PrintWriter.format()一样，它只是返回：

```
// 使用两位小数和千分位符号来格式化一个字符串，将double
// 值转换为文字
double balance = getBalance();
String msg = String.format("Account balance: $%,.2f", balance);
```

java.util.Formatter类是个在printf()和format() method后面的多用途格式化类。它可以将文字格式化为任何 Appendable 对象或具名文件。以下程序代码使用了Formatter 对象来写文件：

```
public static void writeFile(String filename, String[] lines)
    throws IOException
{
    Formatter out = new Formatter(filename);  // format to a named file
    for(int i = 0; i < lines.length; i++) {
        // 写一行到文件
        out.format("%d: %s%n", i, lines[i]);
        // 检查异常
        IOException e = out.ioException();
        if (e != null) throw e;
    }
    out.close();
}
```

当你将对象与字符串连接时，调用对象的toString() method可以将它转换为字符串。这也是Formatter 类的默认动作。如果要对类的格式化做更精确的控制，除了实现toString()之外，还可以实现 java.util.Formattable接口。

在我们说明处理数值、日期以及时间的API时，会看到一些以printf()来进行格式化的额外范例。关于可使用的格式限定符和选项的完整列表，请参阅java.util.Formatter。

Logging

基于终端机的单纯程序可以用 System.out.println()或 System.out.print()来传送输出结果和错误信息到控制台。无人看管且长时间运行的服务器程序需要不同的输出方案：它们在运行时所用的硬件上可能没有连接终端机来显示，如果真是如此，就不太

Download at http://www.pin5i.com/

可能有任何人看着。这样的程序必须要有日志记录功能，输出信息要被传送至文件以供后续分析，或经由网络 socket 以供远程监控。Java 1.4 在 java.util.logging 包中提供了日志记录 API。

通常，应用程序开发者会使用与应用程序的类或包相结合的Logger对象来产生7种安全等级（severity level）的日志信息（请参阅 java.util.logging.Level）。这些信息可以报告错误和警告，或提供有关应用程序生命周期中令人感兴趣的事件的信息。它们可以包含调试信息，甚至追踪程序中重要 method 的执行。

应用程序的系统管理员或终端用户有责任设定日志记录配置文件，其中设定了日志信息被导向何处（控制台、文件、网络 socket 或这些的组合）、格式化的方式（一般文本文件或 XML 文件）以及它们被记录时的安全门槛（severity threshold，具有规定门槛以下的安全信息会在资源极少的情况下被抛弃，而且不应该明显影响应用程序的性能）。日志记录安全门槛等级可以被独立地设定，以便与不同类或包结合的Logger对象能被“调入”或“调出”。由于具有可调性，你应该放心地在你的程序中自由使用日志记录输出。在正常运行中，大部分的日志信息会被有效率地自动抛弃。但是，在程序开发期间或在诊断已配置的应用程序的问题时，日志信息被证明非常有价值。

对于大部分的应用程序来说，使用Logging API是相当简单的。每当有需要时，调用静态Logging.getLogger() method、传送应用程序的类或包的名称作为logger名称，就能取得具名Logger 对象。接着，使用多个 Logger 实例 method 中的一个来产生日志信息。最易于使用的method，其名称是与安全等级对应的，例如servere()、warning()以及 info()。这里有一些范例程序代码：

```
import java.util.logging.*;

// 取得以当前包命名的Logger 对象
Logger logger = Logger.getLogger("com.davidflanagan.servers.pop");
logger.info("Starting server.");          // 记录具有信息的消息
ServerSocket ss;                          // 做一些事
try { ss = new ServerSocket(110); }
catch(Exception e) {                      // 记录异常
  logger.log(Level.SEVERE, "Can't bind port 110", e); // 复杂的日志信息
  logger.warning("Exiting");                          // 单纯的警告
  return;
}
logger.fine("got server socket"); // 详细（低安全性）的调试信息
```

使用正则表达式进行模式匹配

在Java 1.4及之后的版本中，你可以用正则表达式来执行文本模式匹配。正则表达式是由java.util.regex包的 Pattern和Matcher类提供支持的，但String类定义了一

Download at http://www.pin5i.com/

些方便的method，能让你在使用正则表达式时更为容易。正则表达式使用了相当复杂的文法来描述字符组成的模式。Java 的实现使用了与 Perl 5 程序设计语言相同的 regex 语法。关于此语法的更进一步的细节，请查阅适合的 Perl 程序设计书籍。至于 Perl 风格的正则表达式的完整教学，请参阅《Mastering Regular Expressions》（O'Reilly）。

接受正则表达式自变量的最简单的 `String` method 就是 `matches()`。如果字符串与特定正则表达式所定义的模式相匹配，就返回 `true`：

```
// 此字符串是个正则表达式，它描述了典型句子
// 的模式。在 Perl 风格的正则表达式语法中，
// 这指定了以大写字母开始并以句号、问号、
// 或惊叹号结束的字符串
String pattern = "^[A-Z].*[\\.?!]$";
String s = "Java is fun!";
s.matches(pattern);  // 此字符串与模式相同，所以会返回 true
```

`matches()` method 只有在整个字符串与指定的模式相匹配时才会返回 `true`。Perl 程序员应该注意，这与 Perl 的行为不同，在 Perl 中，相匹配只是代表字符串的一些部分与模式相匹配。如果要判定字符串或任何子字符串是否与模式相匹配，只要变动正则表达式以允许所需模式的前后有任意字符即可。在以下程序代码中，正则表达式字符 `.*` 与任何数量的任意字符相匹配：

```
s.matches(".*\\bJava\\b.*"); // 如果 s 有任何地方包含单词 "Java" 就是 true
                             // b 指定了单词界限
```

如果你已熟悉 Perl 的正则表达式语法，就会知道它依赖自由使用反斜线来输出特定字符。在 Perl 中，正则表达式是语言原有的，它们的语法是语言本身的一部分。但是在 Java 中，正则表达式是用字符串来描述的，而且通常会使用字符串直接量来内嵌在程序中。Java 的字符串直接量的语法也使用了反斜线作为转义字符，所以如果要在正则表达式中包含一个反斜线，就必须使用两个反斜线。因此，在 Java 程序设计中，你常会在正则表达式中看到双反斜线。

除了匹配之外，正则表达式还可以用于搜索与取代操作。`replaceFirst()` 和 `replaceAll()` method 会搜索字符串中与给定模式相匹配的第一个子字符串或所有子字符串，并以指定的替换文字来取代那些字符串，再返回包含有替换文字的新字符串。例如，你可以用下面这段程序代码来确保单词 "Java" 在字符串 s 中正确地使用了大写：

```
s.replaceAll("(?i)\\bjava\\b",// 模式：单词 "Java"，不区分大小写
             "Java");         // 替换字符串，正确地使用大写
```

传递给 `replaceAll()` 和 `replaceFirst()` 的替换字符串不必是单纯的字符串直接量，它也可以包含对与模式中加上括号的子表达式相匹配的文本的引用。这些引用采用的形式是美元符号后面加上一些子表达式。例如，如果要搜索如 JavaBean、JavaScript、

Download at http://www.pin5i.com/

JavaOS、和 Java VM 这样的词（不含 Java 或 Javanese）并用字母 J 取代 java，可以使用这样的程序代码：

```
s.replaceAll("\\bJava([A-Z]\\w+)",   // 模式
             "J$1");        // J 后面的字符与括号中的匹配
                            // 子表达式:[A-Z]\\w+
```

另一个使用了正则表达式的String method是split()，它会返回由字符串的子字符串所组成的数组，用来分隔的定界符与指定的模式相匹配。如果要取得字符串中由任意数量的空格、跳格符或换行分隔的字符串组成的数组，就这样做：

```
String sentence = "This is a\n\ttwo-line sentence";
String[] words = sentence.split("[ \t\n\r]+");
```

选择性的第二个自变量指定了返回数组中项目的最大数量。当你只使用一次正则表达式时，matches()、replaceFirst()、replaceAll()以及 split() method 很适合。如果你想把正则表达式用于多重匹配，就应该显式地使用 java.util.regex 包中的 Pattern 和 Matcher 类。首先，以静态 Pattern.compile() method 创建 Pattern 对象表示你的正则表达式（另一个显式地使用Pattern 类来代替String的method的原因就是 Pattern.compile()能让你指定如 Pattern.CASE_INSENSITIVE 这样的标记，来全部变动模式匹配的方式）。请注意，如果你输入无效的正则表达式字符串，则 compile() method可以被抛出PatternSyntaxException(此异常也会由各类String method抛出)。Pattern类定义了与String.split() method类似的split() method。但是，对于其他的匹配，你必须用matcher() method 创建 Matcher 对象并指定所要匹配的文字：

```
import java.util.regex.*;

Pattern javaword = Pattern.compile("\\bJava(\\w*)",
Pattern.CASE_INSENSITIVE);
Matcher m = javaword.matcher(sentence);
boolean match = m.matches();  // 如果文字与模式完全匹配就返回 true
```

一旦你有了 Matcher 对象，就可以用各种方法来比较字符串和模式。较复杂的一种方式，就是找出所有与模式相匹配的子字符串：

```
String text = "Java is fun; JavaScript is funny.";
m.reset(text);  // 开始与新字符串匹配
// 循环找出所有相匹配字符串，并列出每个相匹配部分的细节
while(m.find()) {
  System.out.println("Found '" + m.group(0) + "' at position " +
m.start(0));
  if (m.start(1) < m.end(1)) System.out.println("Suffix is " +
m.group(1));
}
```

Download at http://www.pin5i.com/

Matcher 类在 Java 5.0 中已用好几种方式来加以强化。这其中最重要的就是将最近的匹配结果存储在MatchResult对象中的能力。前面的找出字符串中所有相匹配部分的算法，在 Java 5.0 中可以改写如下：

```
import java.util.regex.*;
import java.util.*;

public class FindAll {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile(args[0]);
        String text = args[1];

        List<MatchResult> results = findAll(pattern, text);
        for(MatchResult r : results) {
            System.out.printf("Found '%s' at (%d,%d)%n",
                              r.group(), r.start(), r.end());
        }
    }

    public static List<MatchResult> findAll(Pattern pattern, CharSequence text)
    {
        List<MatchResult> results = new ArrayList<MatchResult>();
        Matcher m = pattern.matcher(text);
        while(m.find()) results.add(m.toMatchResult());
        return results;
    }
}
```

文字标记化

java.util.Scanner 是个通用的文字标记化程序，在 Java 5.0 中被加入以与本章前面所说明的 java.util.Formatter 类互补。Scanner 充分运用了 Java 正则表达式，并且可以从字符串、文件、流或任何实现了 java.lang.Readable接口的对象取得输入的文字。Readable 也是 Java 5.0 中新出现的，是 Appendable 接口的相反物。

Scanner 可以将输入的文字打散成为标记，这些标记以空白、任何你想要的分隔字符或正则表达式来进行分隔。它实现了 Iterator<String>接口，这能对返回的标记进行简单的逐一处理。Scanner 也定义了各种方便的 method，使用依地区识别数值的解析方式，把标记解析为布尔值、整数或浮点值。skip() method 会略过与指定模式相匹配的输入文字，也有可以针对输入文字搜索与指定模式相匹配的文字的 method。

这里是使用 Scanner 将 String 打散为以空格分隔的单词的方式：

```
public static List<String> getTokens(String line) {
    List<String> result = new ArrayList<String>();
    for(Scanner s = Scanner.create(line); s.hasNext(); )
        result.add(s.next());
```

Download at http://www.pin5i.com/


```
        return result;
    }
```

这里是使用 Scanner 将文件打散为行的方式：

```
    public static void printLines(File f) throws IOException {
        Scanner s = Scanner.create(f);
        // 使用 regex 将换行字符指定为标记分隔符
        s.useDelimiter("\r\n|\n|\r");
        while(s.hasNext()) System.out.println(s.next());
    }
```

以下 method 使用 Scanner 来解析具有 x + y = z 形式的输入，它示范了 Scanner 扫描数值的能力。请注意，Scanner 不只是解析 Java 风格的整数直接量：它支持千分位符号并且是以依地区识别的方式来进行。例如，它会针对美国的用户解析整数 1,234，针对法国的用户解析 1.234。这段程序代码也说明了 skip() method，并展示了 Scanner 可以从 InputStream 直接扫描文字。

```
    public static boolean parseSum() {
        System.out.print("enter sum> "); // 提示用户输入
        System.out.flush();              // 确定提示可立刻被看见

        try {
            // 读取并解析来自控制台的用户输入
            Scanner s = Scanner.create(System.in);
            s.useDelimiter("");      // 不要求标记间要有空格
            int x = s.nextInt();     // 解析整数
            s.skip("\\s*\\+\\s*"); // 略过非必要的空格和直接量 +
            int y = s.nextInt();     // 解析另一个整数
            s.skip("\\s*=\\s*");     // 略过非必要的空格和直接量 =
            int z = s.nextInt();     // 解析第三个整数

            return x + y == z;
        }
        catch(InputMismatchException e) { // 无匹配的模式
            throw new IllegalArgumentException("syntax error");
        }
        catch(NoSuchElementException e) { // 不再有输入可用
            throw new IllegalArgumentException("syntax error");
        }
    }
```

StringTokenizer

还有其他一些 Java 类会分隔字符串和字符。一个值得注意的类是 java.util.StringTokenizer，你可以用它将文字字符串打散为组成其的单词：

```
    String s = "Now is the time";
    java.util.StringTokenizer st = new java.util.StringTokenizer(s);
```

Download at http://www.pin5i.com/


```
while(st.hasMoreTokens()) {
  System.out.println(st.nextToken());
}
```

你甚至可以使用这个类来标记用空格以外的字符所分隔的单词：

```
String s = "a:b:c:d";
java.util.StringTokenizer st = new java.util.StringTokenizer(s, ":");
```

java.io.StreamTokenizer是另一个标记类，它具有比StringTokenizer更复杂的API并且具有更强大的性能。

数值与数学运算

Java提供了byte、short、int、long、float和double这些基本类型来表示数值。java.lang包包含了对应的Byte、Short、Integer、Long、Float以及Double类，这些都是Number的子类。这些类在作为其基本类型的对象封装程序时很有用，而且它们也定义了一些有用的常量：

```
// 整数范围的常量：Integer、Long以及Character也定义了这些
Byte.MIN_VALUE        // 最小（最负）的byte值
Byte.MAX_VALUE        // 最大的byte值
Short.MIN_VALUE       // 最小的short值
Short.MAX_VALUE       // 最大的short值

// 浮点数范围的常量：Double也定义了这些
Float.MIN_VALUE       // 最小（最接近零）的正浮点值
Float.MAX_VALUE       // 最大的正浮点值

// 其他有用的常量
Math.PI               // 3.14159265358979323846
Math.E                // 2.71828182845904523 54
```


数学函数

Math类定义了一些method，提供三角、对数、指数、四舍五入运算等。此类主要是在浮点值运算时很有用。针对三角函数，角度是以弧度制来表示。对数和指数函数是以 *e* 为底数，不是以10为底数。这有一些范例：

```
double d = Math.toRadians(27);        // 将角度值27度转为弧度值
d = Math.cos(d);                      // 取得余弦值
d = Math.sqrt(d);                     // 取得平方根
d = Math.log(d);                      // 取得自然对数
d = Math.exp(d);                      // 计算它的反函数：取e的d次方
d = Math.pow(10, d);                  // 取10的d次方
d = Math.atan(d);                     // 计算反正切值
d = Math.toDegrees(d);                // 再把弧度值转换为角度值
```

Download at http://www.pin5i.com/


```
double up = Math.ceil(d);              // 无条件进位
double down = Math.floor(d);           // 无条件舍去
long nearest = Math.round(d);          // 四舍五入至最近的整数
```

在 Java 5.0 中，Math 类中增加了几个新的函数，其中包括以下函数：

```
double d = 27;
d = Math.cbrt(d);        // 立方根
d = Math.log10(d);       // 以10为底的对数
d = Math.sinh(d);        // 双曲线正弦，还有cosh()和tanh()
d = Math.hypot(3, 4);    // 直角三角形的斜边 (Hypotenuse)
```

随机数

Math类也定义了用于产生虚拟随机数的基本method，但java.util.Random类较具灵活性。如果你需要随机性很强的虚拟随机数，可以使用java.security.SecureRandom类：

```
// 简单的随机数
double r = Math.random();      // 返回：0.0 <= d < 1.0

// 创建一个新的Random对象，以当前时间作为参数
java.util.Random generator = new
java.util.Random(System.currentTimeMillis());
double d = generator.nextDouble();   // 0.0 <= d < 1.0
float f = generator.nextFloat();     // 0.0 <= f < 1.0
long l = generator.nextLong();       // 从整个long范围内选择
int i = generator.nextInt();         // 从整个int范围内选择
i = generator.nextInt(limit);        // 0 <= i < limit (适用于Java 1.2及之后的
                                        版本)
boolean b = generator.nextBoolean();  // true或false (适用于Java 1.2及之后的版本)
d = generator.nextGaussian();        // 平均值为0.0，标准方差为1.0
byte[] randomBytes = new byte[128];
generator.nextBytes(randomBytes);    // 把随机byte填入数组

// 针对加密的随机数，使用SecureRandom子类
java.security.SecureRandom generator2 = new java.security.SecureRandom();
// 让产生器产生它自己的16字节参数，这会花很长的时间
generator2.setSeed(generator2.generateSeed(16)); // 额外的随机16字节参数
// 接着就像其他的Random对象一样使用SecureRandom
generator2.nextBytes(randomBytes);                // 产生更多的随机字节
```

数值很大的数

java.math包包含了BigInteger和BigDecimal类，这些类能让你处理任意大小与任意精确度的整数和浮点数值。例如：

```
import java.math.*;

// 计算1000的阶乘
BigInteger total = BigInteger.valueOf(1);
```

Download at http://www.pin5i.com/


```
for(int i = 2; i <= 1000; i++)
  total = total.multiply(BigInteger.valueOf(i));
System.out.println(total.toString());
```

在Java 1.4中，BigInteger有一个method来随机地产生很大的原始数值，这在许多的密码应用中很有用：

```
BigInteger prime =
  BigInteger.probablePrime(1024,          // 1024位长
                           generator2); // 随机数的来源，来自于上文
```

BigDecimal类在Java 5.0中已被彻底修改，在这版中更有用。它除了能代表非常大或非常精确的浮点数之外，在财务计算中也很有用，因为财务计算是用十进制的表示方式，而不是二进制的表示方式。float和double值无法精确地将数值简单地表示为0.1，这会造成四舍五入错误，通常在表示货币值时是不能接受的。BigDecimal与其相关的MathConetxt和RoundingMode类型提供了解决方案。例如：

```
// 计算贷款所需付的月利率
public static BigDecimal monthlyPayment(int amount, // 贷款金额
                                        int years,  // 年限
                                        double apr) // 年利率 %
{
  // 将贷款金额转换为BigDecimal
  BigDecimal principal = new BigDecimal(amount);

  // 将以年计算的贷款期限转换为月
  int payments=years*12;

  // 将利息从年利率转换为月利率
  BigDecimal interest = BigDecimal.valueOf(apr);
  interest = interest.divide(new BigDecimal(100));      // 成为小数
  interest = interest.divide(new BigDecimal(12));       // 每月

  // 月付款计算
  BigDecimal x = interest.add(BigDecimal.ONE).pow(payments);
  BigDecimal y = principal.multiply(interest).multiply(x);
  BigDecimal monthly = y.divide(x.subtract(BigDecimal.ONE),
                                MathContext.DECIMAL64);  // 请注意来龙去脉

  // 转换为两位小数
  monthly = monthly.setScale(2, RoundingMode.HALF_EVEN);

  return monthly;
}
```

在数字与字符串之间转换

在数字上进行操作的Java程序必须从某些地方取得输入值。通常，这样的程序会读取文本形式的数字，而且必须将它转换成数字形式。各种Number子类定义了很多有用的转换method：

Download at http://www.pin5i.com/


```
String s = "-42";
byte b = Byte.parseByte(s);          // s作为byte
short sh = Short.parseShort(s);      // s作为short
int i = Integer.parseInt(s);         // s作为int
long l = Long.parseLong(s);          // s作为long
float f = Float.parseFloat(s);       // s作为float（适用于Java 1.2及之后的版本）
f = Float.valueOf(s).floatValue();   // s作为float（适用于Java 1.2及之后的版本）
double d = Double.parseDouble(s);    // s作为double（适用于Java 1.2及之后的版本）
d = Double.valueOf(s).doubleValue(); // s作为double（适用于Java 1.2之前的版本）

// 整数转换规则会处理使用其他进制的数字
byte b = Byte.parseByte("1011", 2);       // 二进制中的1011是十进制中的11
short sh = Short.parseShort("ff", 16);    // 十六进制中的ff是十进制中的255

// valueOf() method可以处理从2到36的任意基数
int i = Integer.valueOf("egg", 17).intValue();    // 17进制!

// decode() method能处理八进制、十进制或十六进制的数字，具体使用的是哪种方式取决
// 于字符串的数字前缀
short sh = Short.decode("0377").byteValue();    // 以0开头代表八进制
int i = Integer.decode("0xff").shortValue();    // 以0x开头代表十六进制
long l = Long.decode("255").intValue();         // 其他的数字代表十进制

// Integer类可以将数字转换为字符串
String decimal = Integer.toString(42);
String binary = Integer.toBinaryString(42);
String octal = Integer.toOctalString(42);
String hex = Integer.toHexString(42);
String base36 = Integer.toString(42, 36);
```

格式化数字

本章前面所说明的Java 5.0的printf()和format() method也能用于格式化数字。%d格式限定符用于将整数格式化为十进制格式：

```
// 将int、long和BigInteger格式化为字符串“1 10 100”
String s = String.format("%d %d %d", 1, 10L, BigInteger.TEN.pow(2));
// 增加千分位符号
s = String.format("%,d", Integer.MAX_VALUE); //  “2,147,483,647”
// 将输出值向右对齐放在在八个字符宽度的字段中
s = String.format("%8d", 123);               //  “123”
// 以 0 填充左边以让数字总共有5个
s = String.format("%05d", 123);              //  “00123”"
```

浮点数可以用%f、%e或%g格式限定符来格式化，这在是否使用指数符号时会有差别：

```
double x = 1.234E9; // 1.234 billion, 12.34 亿
// 返回"1234000000.000000 1.234000e+09 1.234000e+09 1234.000000"
s = String.format("%f %e %g %g", x, x, x, x/1e6);
```

你会注意到以上的数字在小数点后都有六位。这样的默认值可以通过变更格式字符串中的精确度来更改：

Download at http://www.pin5i.com/


```
// 显示具有两位小数的BigDecimal
s = String.format("%.2f", new BigDecimal("1.234"));  // "1.23"
```

其他的标记也可以应用至浮点转换。以下程序代码将数字栏格式化为宽为10个字符、向右对齐的格式。每个数字在小数点后都有两位，并且在有需要时加上千分位符号。负值会被格式化为加上括号，这是会计上常见的格式惯例。

```
// 由4个数字组成的栏。%n是换行字符
s = String.format("%(,10.2f%n%(,10.2f%n%(,10.2f%n%(,10.2f%n",
                  BigDecimal.TEN,                    //      10.00
                  BigDecimal.TEN.movePointRight(3),  //  10,000.00
                  BigDecimal.TEN.movePointLeft(3),   //       0.01
                  BigDecimal.TEN.negate());          //     (10.00)
```

在Java 5.0之前，可以使用java.text.umberFormat类来格式化数字：

```
import java.text.*;

// 使用NumberFormat来格式化与解析当前locale的数字

NumberFormat nf = NumberFormat.getNumberInstance();  // 取得NumberFormat
System.out.println(nf.format(9876543.21)); // 针对当前locale来格式化数字
try {
  Number n = nf.parse("1.234.567,89");      // 依据locale来解析字符串
} catch (ParseException e) { /* Handle exception */ }

// 货币值的格式化有时会与其他数字不同
NumberFormat moneyFmt = NumberFormat.getCurrencyInstance();
System.out.println(moneyFmt.format(1234.56)); // 列出U.S.中的$1,234.56
```


日期与时间

Java允许以三种形式来表现并处理日期和时间：作为long值、作为java.util.Date或作为java.util.Calendar对象。Java 5.0引入了enumerated类型java.util.concurrent.TimeUnit。这个类型的值代表了时间的精确度或单位：秒（second）、毫秒（millisecond）、微秒（microsecond，一百万分之一秒）及纳秒（nanosecond，十亿分之一秒）。它们具有方便有用的method，但其本身不代表时间。

毫秒与纳秒

在最低层，日期和时间被表示为long值，这个值包括自1970年1月1日午夜开始的正或负毫秒数。此特殊的日期和时间也就是新纪元（epoch），并且是格林威治标准时间（Greenwich Mean Time，GMT）或世界时（Universal Time，UTC）。如果要查询以此毫秒表示的当前时间，可以使用System.currentTimeMillis()：

Download at http://www.pin5i.com/


```
long now = System.currentTimeMillis();
```

在Java 5.0及之后的版本中，你可以使用System.nanoTime()来查询以纳秒为单位的时间。此method会返回纳秒长度的long数字。与currentTimeMillis()不同，nanoTime()不会返回任何相对于已定义的新纪元的时间。nanoTime()很适合用于测量相对或占用时间（只要占用时间不大于292年），但不适合用于绝对时间：

```
long start = System.nanoTime();
doSomething();
long end = System.nanoTime();
long elapsedNanoSeconds = end - start;
```

Date类

java.util.Date是围绕着long的对象封装程序，它包含了自新纪元开始的毫秒数。使用Date对象来代替long，就能用toString method进行简单的转换，以成为非本地化的字符串。Date对象可以用equals()进行相等性比较，而且可以用compareTo() method或before()和after() method比较顺序。

无自变量版本的Date()构造函数会创建一个表示当前时间的Date，你也可以传递毫秒的long数值来创建表示其他时间的Date。getTime()会返回Date的毫秒表达式。Date是可变（mutable）类，所以你也可以传递毫秒数给setTime()。

Date有一些用于查询和设定年、月、日、时、分和秒的method。但是，现在都不赞成使用这些method，而是支持使用Calendar类，如下段所述。

Calendar类

java.util.Calendar类是个适当地本地化版本的Date。它只是围绕于毫秒数的long值的封装程序，但可以表示依据当前本地的日历（通常是新历（Gregorian calendar））的时间。此外，它具有用于在各种日期和时间字段上查询、设定以及计算的method。

以下程序代码展示了Calendar类的常见用法。请注意，set()、get()以及add() method都会接受初始自变量，它指定了日期或时间的哪个字段要被设定、查询或加减。例如年、一月天次、一周天次、时、分和秒这样的字段，会在类中以整数常量被定义。其他的整数常量定义了新历的月和周的值，月常量UNDECIMBER代表阴历的第13个月。

```
// 取得当前地区和时区的Calendar
Calendar cal = Calendar.getInstance();

// 算出今天是今年的第几天
cal.setTimeInMillis(System.currentTimeMillis()); // 设定当前时间
int dayOfYear = cal.get(Calendar.DAY_OF_YEAR);   // 今天是今年的第几天?
```

Download at http://www.pin5i.com/


```
// 2008 年的 2 月 29 日是星期几?
cal.set(2008, Calendar.FEBRUARY, 29);              // 设定年、月、日字段
int dayOfWeek = cal.get(Calendar.DAY_OF_WEEK);     // 查询另一个字段

// 2005 年 5 月的第 3 个星期四是当月的第几天?
cal.set(Calendar.YEAR, 2005);                      // 设定年
cal.set(Calendar.MONTH, Calendar.MAY);             // 设定月
cal.set(Calendar.DAY_OF_WEEK,Calendar.THURSDAY);   // 设定星期几
cal.set(Calendar.DAY_OF_WEEK_IN_MONTH, 3);         // 设定周
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH);   // 查询那天是当月的第几天

// 取得表示从现在开始的三个月的 Date 对象
cal.setTimeInMillis(System.currentTimeMillis());   // 当前时间
cal.add(Calendar.MONTH, 3);                        // 增加三个月
Date expiration = cal.getTime();                   // 取得 Date 形式的结果
long millis = cal.getTimeInMillis();               // 或使用 long 形式
```

格式化日期与时间

Date的toString() method会产生文字形式的日期和时间，但不会做本地化，而且不允许指定显示哪些字段（例如，日、月和年或时、分）。toString() method应该只用于产生机器可读取的时间，而不是人所读取的字符串。

就和数字一样，日期和时间可以用String.format() method和Java 5.0相关的java.util.Formatter类来转换为字符串。用于显示日期和时间的格式字符串都是以字母t开头的双字符序列。每个序列的第二个字母指定或设定了要被显示的日期或时间字段。例如%tR显示使用24小时制的时和分字段，而%tD显示以斜线分隔的月、日以及年字段。String.format()可以将指定的日期或时间格式化为long、Date或Calendar：

```
// 现在的时和分
long now = System.currentTimeMillis();
String s = String.format("%tR", now);          // "15:12"

// 现在的月 / 日 / 年
Date d = new Date(now);
s = String.format("%tD", d);                   // "07/13/04"

// 使用 12 小时制的时和分
Calendar c = Calendar.getInstance();
c.setTime(d);
s = String.format("%tl:%tM %tp", now, d, c);   // "3:12 pm"
```

在Java 5.0及其Formatter类之前，你可以用java.text.DateFormat类来格式化日期和时间，它会针对日期和时间格式化并自动处理与地区相关的惯例的转换。甚至在使用了与全世界大部分地区常使用的阳历不同的日历的地区，DateFormat也能正确处理：

Download at http://www.pin5i.com/


```
import java.util.Date;
import java.text.*;

// 使用默认的本地格式来显示当前的日期
DateFormat defaultDate = DateFormat.getDateInstance();
System.out.println(defaultDate.format(new Date()));

// 使用本地的短时间格式来显示当前时间
DateFormat shortTime = DateFormat.getTimeInstance(DateFormat.SHORT);
System.out.println(shortTime.format(new Date()));

// 使用长格式同时显示日期和时间
DateFormat longTimestamp =
  DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL);
System.out.println(longTimestamp.format(new Date()));

// 使用 SimpleDateFormat 来定义你自己的格式化模板 (template)
// 关于模板的语法请参阅 java.text.SimpleDateFormat
DateFormat myformat = new SimpleDateFormat("yyyy.MM.dd");
System.out.println(myformat.format(new Date()));
try {   // DateFormat 也可以解析日期
  Date leapday = myformat.parse("2000.02.29");
}
catch (ParseException e) { /* Handle parsing exception */ }
```

数组

java.lang.System类定义了arraycopy() method，它对于把第一个数组中的特定元素复制到第二个数组中的指定位置很有用。第二个数组必须与第一个数组有相同的类型，甚至可以是同一个数组：

```
char[] text = "Now is the time".toCharArray();
char[] copy = new char[100];
// 从 text 中的元素 4 开始复制 10 个字符到 copy，从 copy[0]开始
System.arraycopy(text, 4, copy, 0, 10);

// 将 text 的一部分移到较后面的元素以让出地方供插入
System.arraycopy(copy, 3, copy, 6, 7);
```

在Java 1.2和之后的版本中，java.util.Arrays类定义了有用的数组操作method，其中包括了用于排序和查找数组的method：

```
import java.util.Arrays;

int[] intarray = new int[] { 10, 5, 7, -3 }; // 整个数组
Arrays.sort(intarray);                       // 进行排序
int pos = Arrays.binarySearch(intarray, 7);  // 值 7 是在位置 2 找到的
pos = Arrays.binarySearch(intarray, 12);     // 未找到：返回负值
```

Download at http://www.pin5i.com/


```
// 由对象组成的数组也能被排序和查找
String[] strarray = new String[] { "now", "is", "the", "time" };
Arrays.sort(strarray);   // sorted to: { "is", "now", "the", "time" }

// Arrays.equals()会比较两数组中的所有元素
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone);  // 没错，它们相等

// Arrays.fill()会初始化数组元素
byte[] data = new byte[100];            // 空数组，元素都被设定为0
Arrays.fill(data, (byte) -1);            // 将它们都设定为-1
Arrays.fill(data, 5, 10, (byte) -2);    // 将元素5, 6, 7, 8, 9设定为-2
```

在Java中，数组可以被当成对象来处理与操作。给定任意对象o，你可以使用如下的程序代码来查明此对象是否是数组，如果是，又是什么类型的数组：

```
Class type = o.getClass();
if (type.isArray()) {
  Class elementType = type.getComponentType();
}
```

集合

Java集合结构（Collection Framework）是java.util包中的一组重要的实用程序类和接口，用于处理对象的集合。Collection Framework定义了两个基本的集合类型。Clooection是一组对象，而Map是对象之间的一组映射或关联。Set是一种不具有重复项目的Collection类型，而List是其元素已被排序的Collection。SortedSet和SortedMap是特殊化的set和map，以排序的方式来维护它们的元素。Collection、Set、List、Map、SortedSet以及SortedMap都是接口，但java.util包也定义了各种具体的实现，例如基于数组和链接列表的list以及基于哈希表或二元树的map和set。其他的重要接口是Iterator和ListIterator，能让你逐个处理集合中的对象。Collection Framework是在Java 1.2中加入的，但在那个版本之前，你可以使用Vector和Hashtable，它们几乎与ArrayList和HashMap相同。

在Java 1.4中，Collection API增加了RandomAccess这个marker inteface，它是由List的实现来完成，支持了有效率的随机访问（也就是说，它是由ArrayList和Vecctor实现，而不是由LinkedList）。Java 1.4也引入了LinkedHashMap和LinkedHashSet，它是基于哈希表的map和set，会保留元素的插入顺序。最后，IdentityHashMap是个基于哈希表的Map实现，它使用==运算符来比较关键对象，而不是使用equals() method来进行比较。

Collection Framework在Java 5.0中已被彻底修正为使用generics（请参阅第四章）。Java 5.0还增加了EnumSet和EnumMap类，它们被特殊化以处理枚举值（请参阅第四章）以

Download at http://www.pin5i.com/

及新的for/in循环语句所使用的java.lang.Iterable接口。最后，Java 5.0增加了Queue接口。大部分令人感兴趣的Queue实现是java.util.concurrent中的BlockingQueue实现。

Collection 接口

Collection<E>是个参数化接口，表示由类型E的对象所组成的泛型组。此组有可能允许重复元素，也有可能会对元素排序。有一些method被定义来从组增加或删除对象、测试对象在组中的成员资格以及逐一处理组中的所有元素。另有一些method会以数组形式返回组的元素并返回集合的大小。

Java Collection Framework不提供任何Collection的实现，但此接口仍然非常重要，因为它定义了所有针对Set、List以及Queue实现的共同特性。以下程序代码说明在Collection对象上能执行的操作：

```
// 创建一些collection来运行
Collection<String> c = new HashSet<String>();  // 一个空的set
// 我们稍后会看到这些实用程序method
Collection<String> d = Arrays.asList("one", "two");    // 固定不变
Collection<String> e = Collections.singleton("three"); // 固定不变

// 增加collection的元素。如果collection变动，这些method就会返回
// true，这对于不允许重复项目的Set很有用
c.add("zero");              // 增加单一元素
c.addAll(d);                // 增加许多元素

// 复制collection：大部分的实现会有个copy构造函数
Collection<String> copy = new ArrayList<String>(c);

// 删除collection的元素
// 如果collection有变更，则除了Clear()之外，全部都会返回true
c.remove("zero");           // 删除一个元素
c.removeAll(e);             // 删除许多元素
c.retainAll(d);             // 删除所有不在e中的元素
c.clear();                  // 删除collection的所有元素

// 查询collection的大小
boolean b = c.isEmpty(); // collection现在是空的
int s = c.size();        // collection的大小现在是0

// 从我们制做的副本来复原collection
c.addAll(copy);

// 测试collection中的成员数据，成员数据是基于equals()
// method，而不是==运算符
b = c.contains("zero");  // true
b = c.containsAll(d);    // true
```

Download at http://www.pin5i.com/


```
// 以while循环逐一处理collection的元素
// 有些实现（例如list）会保证处理的顺序
// 其他的则不保证
Iterator<String> iterator = c.iterator();
while(iterator.hasNext()) System.out.println(iterator.next());

// 使用for循环来处理
for(Iterator<String> i = c.iterator(); i.hasNext(); )
    System.out.println(i.next());

// Java 5.0的迭代使用了for/in循环
for(String word : c) System.out.println(word);

// 大部分的Collection实现有有用的toString() method
System.out.println(c);    // 作为以上迭代的替代方案

// 取得由Collection元素组成的数组。如果iterator
// 保证顺序，此数组就会有相同顺序。此数组是个副
// 本，不是对内部数据结构的引用
Object[] elements = c.toArray();

// 如果我们要让String[]中有元素，就必须先传一个进去
String[] strings = c.toArray(new String[c.size()]);

// 或者我们可以传递一个只有指定类型的空String[]，
// 而toArray() method会为我们分配一个数组
strings = c.toArray(new String[0]);
```

要记住，你可以配合任何Set、List或Queue来使用以上所示的任一个method。这些子接口（subinterface）可能会影响collection的元素上的成员数据限制或顺序限制，但仍提供了相同的基本method。例如add()、remove()、clear()以及retainAll()这些会变动collection的method是选择性的，而只读的实现可能会抛出UnsupportedOperationException。

Collection、Map以及其他的subinterface没有扩展Cloneable或Serializable interface。但是，所有Java Collection Framework中提供的collection和map实现类的确实现了这些interface。

有些collection实现会在可包含的元素上设定限制。例如，有个实现可能会禁止用null作为元素。EnumSet将成员数据限定为属于特定枚举类型的值，试图在collection中加入被禁止的元素，一定会抛出例如NullPointerException或ClassCastException的未检查异常。检查collection中是否包含被禁止的元素，也有可能会抛出这样的异常或是返回false。

Set接口

*set*是由对象组成的collection，其中不允许有重复项目，它不可以包含两个对相同对象

Download at http://www.pin5i.com/

的引用、两个对 null 的引用或对两个对象 a 与 b 而且 a.equals(b) 的引用。大部分的通用 Set 实现不会对 set 的元素排序，但有序的 set 并未被禁止（请参阅 SortedSet 和 LinkedHashSet）。依据一般预期，set 优于像是 list 这样的有序 collection，set 具有一个有效率的在常量或对数时间中运行的 contains() method。

Set 没有定义由 Collection 定义的 method 之外的额外 method，但在那些 method 上增加了额外的限制。如果要强制执行无重复项目规则，Set 的 add() 和 addAll() method 是必要的：如果 set 已包含了那个元素，就不可以再把它加到 Set 中。回想一下，如果调用会造成 collection 改变，则 Collection interface 所定义的 add() 和 addAll() method 就会返回 true，否则就返回 false。此返回值与 Set 对象有关，因为无重复项目的限制表示增加元素未必一定会造成 set 的改变。

表 5-2 列出了 Set interface 的实现，并归纳它们的内部表示方式、排序特性、成员限制、基本 add()、remove() 和 contains() 运算以及迭代的性能。请注意，CopyOnWriteArraySet 是在 java.util.concurrent 包中，其他的实现都是 java.util 的一部分。也请注意，java.util.BitSet 不是一个 Set 实现。这个原有类在作为紧密且有效率的布尔值列表时很有用，但它不是 Java Collection Framwork 的一部分。

表 5-2：Set 实现

类	内部表示方式	元素排序	成员限制	基本运作	迭代性能	注意事项
HashSet	哈希表	无	无	O(1)	O(capacity)	最佳的通用实现
LinkedHashSet	链接哈希表	插入顺序	无	O(1)	O(n)	保留插入顺序
EnumSet	位字段	enum 声明	enum 值	O(1)	O(n)	只包含非 null 的 enum 值
TreeSet	red-black 树形结构	升幂排列	可比较	O(log(n))	O(n)	可比较的元素或 Comparator
CopyOnWrite ArraySet	队列	插入顺序	无	O(n)	O(n)	没有同步method 的线性安全性

TreeSet 实现使用了树形数据结构来维护依升幂排列迭代的 set，排序是依据 Comparable 对象的自然排序或依据由 Comparator 对象所指定的顺序。TreeSet 实际上实现了 SortdSet 接口，那是 Set 的子接口。

SortedSet 提供了好几个令人感兴趣的 method，这些 method 运用了它的已排序本质。以下程序代码作了说明：

Download at http://www.pin5i.com/


```
public static void testSortedSet(String[] args) {
    // 创建 SortedSet
    SortedSet<String> s = new TreeSet<String>(Arrays.asList(args));

    // 迭代 set：元素会被自动排序
    for(String word : s) System.out.println(word);

    // 特别的元素
    String first = s.first();  // 第一个元素
    String last = s.last();  // 最后一个元素
    // set 的子范围查看
    SortedSet<String> tail = s.tailSet(first+'\0'); // 除了第一个元素之外的所有元素
    SortedSet<String> head = s.headSet(last);       // 除了最后一个元素之外的所有元素
    SortedSet<String> middle = s.subSet(first+'\0', last )
                                        // 除了第一个元素和最后一个元素之外的所有元素
                                        ;
}
```

List 接口

List 是有序的对象集合。list 的每个元素在 list 中都有特定位置，而且 List interface 定义了 method 来查询或设定特定位置（或索引）的元素。在这方面，List 就像是大小会变动的数组，会随需要来容纳它所包含的元素数量。与 set 不同的是，list 允许重复的元素。

除了基于索引的 get()和 set() method 之外，List interface 还定义了 method 来增加或删除特定索引中的元素，也定义了 method 来返回特定值在 list 中的第一个或最后一个出现位置的索引。继承自 Collection 的 add()和 remove() method 被定义来添加数据至 list 或从 list 删除与指定值相符的第一个元素。retainAll()和 removeAll() method 的行为就和针对任一种 Collection 的一样，在需要时保留或删除相同的多个值。

List interface 没有定义在一段 list 索引范围内进行操作的 method，但它定义了一个 subList method，它会返回只表示原来 list 的指定范围的 List 对象。子 list 会回溯至父 list，对子 list 所做的任何改变都会立即出现在父 list 中。subList()的范例及其他基本的 List 操作 method 如下。

```
// 创建 list 以运用
List<String> l = new ArrayList<String>(Arrays.asList(args));
List<String> words = Arrays.asList("hello", "world");

// 用索引来查询和设定元素
String first = l.get(0);                // list 的第一个元素
String last = l.get(l.size()-1);        // list 的最后一个元素
l.set(0, last);                         // 将第一个元素变为最后一个

// 增加并插入元素。add()可以添加或插入
l.add(first);        // 将 first 添加在 list 的结尾
```

Download at http://www.pin5i.com/


```
l.add(0, first);     // 将first再次插入list的开头
l.addAll(words);     // 将一个collection添加在list的结尾
l.addAll(1, words); // 将collection插在第一个单词后面

// 子list：回溯至原始list
List<String> sub = l.subList(1,3);  // 第二及第三个元素
sub.set(0, "hi");                   // 修改l的第二个元素
// subList可以将操作限制在list的子范围内
String s = Collections.min(l.subList(0,4));
Collections.sort(l.subList(0,4));
// 子list的独立副本不会影响父list
List<String> subcopy = new ArrayList<String>(l.subList(1,3));

// 搜索list
int p = l.indexOf(last);  // 最后一个单词出现在哪里?
p = l.lastIndexOf(last);  // 逆向搜索

// 列出l中所有与last相符的索引。请注意subList()
int n = l.size();
p = 0;
do {
    // 查看只包含我们尚未搜索的元素的list
    List<String> list = l.subList(p, n);
    int q = list.indexOf(last);
    if (q == -1) break;
    System.out.printf("Found '%s' at index %d%n", last, p+q);
    p += q+1;
} while(p < n);

// 从list删除元素
l.remove(last);           // 删除第一个相符的元素
l.remove(0);              // 删除特定索引中的元素
l.subList(0,2).clear();   // 使用subList()删除一段范围内的元素
l.retainAll(words);       // 删除words中除了元素之外的所有东西
l.removeAll(words);       // 删除words中的所有元素
l.clear();                // 删除所有东西
```

对List实现的一般期待就是它们能被有效率地迭代，通常所需的时间会与list的大小成比例。但是，list并未对任一个索引的元素都提供有效率的随机访问。循序访问的列表（例如LinkedList类）以随机访问性能为代价，提供了有效率的插入和删除操作。在Java 1.4及之后的版本中，提供了有效率的随机访问的实现，包括RandomAccess这个marker interface，如果你需要确认操作的列表效率，可以用instanceof来测试此interface：

```
List<?> l = ...;  // 我们传递供操作的任意list
// 确定我们可以做到有效率的随机访问，否则，就在进行操作
// 前，使用copy构造函数建立随机访问的副本
if (!(l instanceof RandomAccess)) l = new ArrayList<?>(l);
```

由List的iterator() method所返回的Iterator，会依list元素在list中出现的顺

Download at http://www.pin5i.com/

序来迭代。List 实现了 Iterable，而列表就像其他的collection一样，可以使用 for/in 循环来迭代。

如果只要迭代 list 的一部分，可以使用 subList() method 来创建子 list：

```
List<String> words = ...;  // Get a list to iterate

// 迭代除了第一个元素之外的所有元素
for(String word : words.subList(1, words.size()))
    System.out.println(word);
```

除了标准的迭代之外，列表也提供强化的双向迭代，那是使用由 ListIterator() method 所返回的 ListIterator 对象。例如，如果要逆向迭代 List，可以将 ListIterator 的光标定位在列表的结尾来开始 ListIterator：

```
ListIterator<String> li = words.listIterator(words.size());
while(li.hasPrevious()) {
    System.out.println(li.previous());
}
```

表 5-1 归纳了五个在 Java 平台中的通用 List 实现。Vector 和 Stack 是 Java 1.0 留下的原有实现，CopyOnWriteArrayList 是 Java 5.0 中新出现的并且是 java.util.concurrent 包的一部分。

表 5-3：List 实现

类	代表	随机访问	备注
ArrayList	数组	是	最佳的全能实现
LinkedList	双链接列表	否	在新增和插入上有效率
CopyOnWriteArrayList	数组	是	具有线程安全性；追踪（traversal）时快捷，但变动时缓慢
Vector	数组	是	原有类；同步化的 method
Stack	数组	是	扩展了 Vector，增加 push()、pop()、peek()

Map 接口

map 是由一些键对象（key object）和那些键对象与值对象（value object）的映射（mapping）组成的。Map inteface 定义了一个用于定义和查询映射的 API。Map 是 Java Collection Framework 的一部分，但它并没有扩展 Collection interface，所以 Map 是小c的collection，而不是大C的 Collection。Map 是具有两个类型变量的参数化类型。类型变量 K 代表 map 包含的键的类型，而类型变量 V 代表键所映射的值的类型。例如，从 String 键到 Integer 值的映射，可以用 Map<String,Integer> 来表示。

Download at http://www.pin5i.com/

最重要的Map method有：put()，定义了map中的键/值对；get()，查询与特定键结合的值；remove()，从map中删除特定键及与其关联的值。针对Map实现的一般性能期望就是这三个基本method相当有效率：它们应该在常量时间（constant time）内运行，而且不可以比在对数时间（logarithmic time）内的状况糟。

Map有个重要特性，就是它对“集合观点（collection view）”的支持。虽然Map不是个Collection，但它的键可以被看成Set，它的值可以被看成Collection，而它的映射可以被看成由Map.Entry对象组成的Set（Map.Entry是个定义于Map中的嵌套interface：它只表示单一的键/值对）。

以下的程序代码显示了Map的get()、put()、remove()以及其他method，也说明了Map的collection view的一些常见用法：

```
// 创建map以运用
Map<String,Integer> m = new HashMap<String,Integer>();  // 新的空map
// 不可改变的Map包含了单一键－值对
Map<String,Integer> singleton = Collections.singletonMap("testing", -1);
// 请注意，这个很少用的语法是在明确地指定generic emptyMap() method的参数类型。
// 所返回的map是不可变更的
Map<String,Integer> empty = Collections.<String,Integer>emptyMap();

// 使用put()填入map来定义从数组元素到各个元素
//出现位置的索引的映射
String[] words = { "this", "is", "a", "test" };
for(int i = 0; i < words.length; i++)
    m.put(words[i], i);  // 请注意从int到Integer的autoboxing

// 每个键都必须映射到一个值，但多个键可以映射到同一个值
for(int i = 0; i < words.length; i++)
    m.put(words[i].toUpperCase(), i);

// putAll() method会从另一个Map复制映射
m.putAll(singleton);

// 使用get() method查询映射
for(int i = 0; i < words.length; i++)
    if (m.get(words[i]) != i) throw new AssertionError();

// 键和值的成员关系测试
m.containsKey(words[0]);        // true
m.containsValue(words.length);  // false

//  Map的键、值以及映射项目可以被当成collection来查看
Set<String> keys = m.keySet();
Collection<Integer> values = m.values();
Set<Map.Entry<String,Integer>> entries = m.entrySet();

// Map及其collection view通常会有有用的toString() method
System.out.printf("Map: %s%nKeys: %s%nValues: %s%nEntries: %s%n",
                  m, keys, values, entries);
```

Download at http://www.pin5i.com/


```
// 这些collection可以被迭代
// 大部分的map都具有未定义的迭代顺序（请参阅SortedMap）
for(String key : m.keySet()) System.out.println(key);
for(Integer value: m.values()) System.out.println(value);

// Map.Entry<K,V>类型代表map中的单一键/值配对
for(Map.Entry<String,Integer> pair : m.entrySet()) {
    // 列出映射
    System.out.printf("'%s' ==> %d%n", pair.getKey(), pair.getValue());
    // 并递增每个项目的值
    pair.setValue(pair.getValue() + 1);
}

// 删除映射
m.put("testing", null);    // 映射至null就可以"消除"映射
m.get("testing");          // 返回null
m.containsKey("testing"); // 返回true：映射仍然存在
m.remove("testing");       // 删除映射
m.get("testing");          // 仍然返回null
m.containsKey("testing"); // 现在返回false

// 通过map的collection view也可以进行删除
// 但是，不能用这个方式新增map
m.keySet().remove(words[0]);   // 与m.remove(words[0])相同
m.values().remove(2);          // 删除一个对值2的映射
m.values().removeAll(Collections.singleton(4)); // 删除所有对值4的映射
m.values().retainAll(Arrays.asList(2, 3));      // 只保留对2和3的映射

// 通过iterator也可以进行删除操作
Iterator<Map.Entry<String,Integer>> iter = m.entrySet().iterator();
while(iter.hasNext()) {
    Map.Entry<String,Integer> e = iter.next();
    if (e.getValue() == 2) iter.remove();
}

// 找出同时出现在两个map中的值。通常，addAll()和retainAll()配
// 合keySet()和values()能做到并集和交集
Set<Integer> v = new HashSet<Integer>(m.values());
v.retainAll(singleton.values());

// 各种method
m.clear();                 // 删除所有的映射
m.size();                  // 返回映射数量：当前是0
m.isEmpty();               // 返回true
m.equals(empty);           // true：Map实现覆盖了equals
```

Map interface包含了各种一般用途和特殊用途的实现，这些归纳于表5-4。除了ConcurrentHashMap（它是java.util.concurrent的一部分）之外，表5-4中的所有类都是在java.util包中。

Download at http://www.pin5i.com/

表 5-4：Map 实现

类	代表	开始有的版本	null 键	null 值	备注
HashMap	哈希表	1.2	是	是	一般用途的实现
Concurrent-HashMap	哈希表	5.0	否	否	一般用途、具有线程安全性的实现。请参阅 ConcurrentMap interface
EnumMap	array	5.0	否	是	键是 enum 的 instance
LinkedHashMap	哈希表加上列表	1.4	是	是	会保存插入或访问的顺序
TreeMap	red-black 树形结构	1.2	否	是	依键值排序。操作为O(log(n))。请参阅 SortedMap
IdentityHashMap	哈希表	1.4	是	是	使用 = = 来比较，而不是用 equals()
WeakHashMap	哈希表	1.2	是	是	不会阻止键的内存回收
Hashtable	哈希表	1.0	否	否	原有类；同步化的 method
Properties	哈希表	1.0	否	否	以 String method 扩展 Hashtable

`java.util.concurrent`包的`ConcurrentHashMap`类实现了同一个包的`ConcurrentMap` interface。`ConcurrentMap`扩展了`Map`并定义了一些额外的基本操作，这些在多线程程序设计中是很重要的。例如，`putIfAbsent()` method 就像是`put()`，但只有在键尚未被映射时才会对 map 增加键 / 值对。

`TreeMap`实现了`SortedMap` interface，它扩展了`Map`来加入能运用 map 的排序本质的 method。`SortedMap`与`SortedSet` interface 相当类似。`firstKey()`和`lastKey()` method 会返回`keySet()`中的第一个和最后一个键，而`headMap()`、`tailMap()`和`subMap()`会返回限定范围的原始 map。

Queue 与 BlockingQueue 接口

队列（queue）是具有顺序性的元素集合，包含从队列的开头依序提取元素的 method。队列的实现通常是基于如先进先出（FIFO）队列或后进先出队列（LIFO，队列也就是堆栈(stack)）的插入顺序。但是，其他的顺序也是有可能的：优先权队列(priority queue)会依外部的`Comparator`对象或依`Comparable`元素的天然顺序来对其元素排序。与`Set`不同的是，`Queue`的实现通常会允许重复的元素。而与`List`不同的是，`Queue` interface没有定义用于操作任意位置的队列元素的method，只有在队列前端的元素才能被查看。对于`Queue`实现来说，拥有固定的容量是很常见的：当队列已满时，就不能再加入元素。同样地，当队列是空的，就不能再移除任何元素。因为满和空的状况是许多

Download at http://www.pin5i.com/

以队列为基础的算法的基础部分，所以 Queue interface 定义了一些 method，以返回值来表示这些状况，而不是抛出异常。尤其是 peek() 和 poll() method 会返回 null 来表示队列是空的。基于此原因，大部分的 Queue 实现不允许 null 元素。

阻塞型队列（blocking queue）是一种定义了具阻塞性的 put() 和 take() method 的队列。put() method 会增加一个元素到队列，并在必要时等待，直到队列中有空间提供给此元素。而 take() method 会从队列的前端移除一个元素，并在必要时等待，直到有元素可供移除。阻塞型队列是许多线程算法的重要部分，而 BlockingQueue interface（扩展了 Queue）被定义为 java.util.concurrent 包的一部分。Queue、BlockingQueue 以及它们的实现是 Java 5.0 中新出现的。关于 BlockingQueue 实现的列表，请参阅本章稍后的“阻塞型队列”一节。

除了特定的多线程程序设计风格之外，队列在使用上远不如 set、list 以及 map。我们在这不使用程序代码范例，而是试着厘清大量令人困惑的队列插入和移除操作：

- 增加元素至队列

  add()

  此 Collection method 只是以一般方式来增加元素。在有限队列中，如果队列已满，则此 method 可能会抛出异常。

  offer()

  此 Queue method 就像 add()，但在元素因为有限队列已满而无法被加入时就会返回 false，而不是抛出异常。

  BlockingQueue 定义了超时版本的 offer()，它会等待特定的时间，以在已满的队列中得到空间。就和基本版本的 method 一样，如果元素已被插入，就返回 true，否则返回 false。

  put()

  此 BlockingQueue method 会阻塞：如果元素因为队列已满而无法被插入，put() 就会等待，直到有其他的线程从队列中移除一个元素，使得新元素有空间可用为止。

- 从队列移除元素

  remove()

  除了 Collection.remove() method 之外，它会从队列中移除特定的元素。Queue interface 定义了无自变量版本的 remove()，会移除并返回在队列前端的元素。如果队列是空的，则此 method 会抛出 NoSuchElementException。

Download at http://www.pin5i.com/

poll()

此Queue method会移除并返回在队列前端的元素，就和remove()所做的一样，但如果队列是空的，就返回null，而不是抛出异常。

BlockingQueue定义了超时版本的poll()，它会针对元素被加入空队列而等待特定时间。

take()

此BlockingQueue method会移除并返回队列前端的元素。如果队列是空的，它就会停止，直到有其他的线程增加一个元素到此队列为此。

drainTo()

此BlockingQueue method会移除队列中的所有元素，并把它们加到指定的Collection中。它不会停下来等待要被加到队列中的元素。此method的变体会接收最大数量的元素并移出。

- 查询前端的元素，而不将它从队列中删除

element()

此Queue method会返回队列前端的元素，但不会把那个元素从队列删除。如果队列是空的，它就会抛出NoSuchElementException。

peek()

此Queue method就像element()，但如果队列是空的，就会返回null。

在Java 5.0中，LinkedList类已被更新来实现Queue。它提供了无限制的FIFO（fist in, first out）顺序，而且插入和删除只需要常量时间。LinkedList允许null元素，但在list被用来作为队列时，最好不要使用null元素。

java.util包的另一个Queue实现是PriorityQueue，它是依据Comparator来排序其元素，或是依据由compareTo() method所定义的顺序来排序Comparable元素。PriorityQueue的前端总是依据已定义顺序所列出的最小元素。

java.util.concurrent包包含了一些BlockingQueue实现，在本章稍后会作说明。此包也包含了ConcurrentLinkedQueue，那是个有效率的具有线程安全性的Queue实现，不必有同步化method的额外耗费。

Collection封装程序

对于相当多的设计出来配合collection使用的静态实用程序method，java.util.Collections类是其发源地。在这些method中，有一组很重要的就是collection的*wrapper* method：它们会返回特殊用途的collection，包含你所指定的

Download at http://www.pin5i.com/

collection。wrapper collection 的用途是将额外的功能封装至本身未提供那些功能的 collection 中。wrapper 是用来提供线程安全性、写保护和运行时类型检查。wrapper collection 一定会由原始的 collection 支持，意味着 wrapper 的 method 只是发送被封装 collection 的相等 method。这表示 wrapper 对 collection 所做的改动，可以通过被封装的 collection 看到，反之亦然。

第一组 wrapper method 提供了在 collection 之外具有线程安全性的 wrapper。除了原有的 Vector 和 Hashtable 类之外，java.util 中的 collection 实现不具有同步化的 method，而且无法避免多个线程的并行访问。如果你需要具有线程安全性的 collection，可以用如下的程序代码来创建：

```
List<String> list = Collections.synchronizedList(new ArrayList<String>());
Set<Integer> set = Collections.synchronizedSet(new HashSet<Integer>());
Map<String,Integer> map =
    Collections.synchronizedMap(new HashMap<String,Integer>());
```

第二组 wrapper method 提供了 collection 对象，底层的 collection 无法通过这些对象而被修改。它们会返回只读的 collection 以供查找：任何尝试改变 collection 的内容，都会造成 UnsupportedOperationException。当你传递 collection 给 method 而且无论如何都不允许 collection 的内容被变动或改变时，这些 wrapper 会很有用：

```
List<Integer> primes = new ArrayList<Integer>();
List<Integer> readonly = Collections.unmodifiableList(primes);
// 我们可以通过 primes 修改 list
primes.addAll(Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19));
// 但我们无法通过只读的 wrapper 进行修改
readonly.add(23);  // UnsupportedOperationException
```

最后一组 wrapper method 提供了被加到 collection 的值的运行时类型检查。这是 Java 5.0 中增加的，以补充由 generics 所提供的编译期类型安全性。在处理还没被转换成使用 generics 的原有程序代码时，这些 wrapper 会很有用。例如，如果你有个 SortedSet<String>，而且必须把其传递给预期有 Set 的 method 时，就可以使用检验 wrapper 来保证那个 method 不能增加任何东西到不是 String 的 set 上：

```
SortedSet<String> words = new TreeSet<String>();   // 一个 set
SortedSet<String> checkedWords =                   // 一个检验 set
    Collections.checkedSortedSet(words, String.class);
addWordsFromFile(checkedWords, filename);          // 传递给原有 method
```

特殊的 Collections

除了 wrapper method 之外，java.util.Collections 类也定义了实用程序 method，用于创建不可变的 collection instance，其中包含了单一元素和其他用于创建空 collection

Download at http://www.pin5i.com/

的 method。singleton()、singletonList()以及 singletonMap()会返回不可变的 Set、List 和Map 对象，其中包含单一赋值对象或单一键/值对。这些 method 很有用，例如，当你必须传递单一对象到预期要有 collection 的 method 时。

Collections类也包含了会返回空collection的method。如果你编写了会返回collection的method，通过返回空collection而不是返回像null这样的特殊值来处理无值返回（no-values-to-return）情况通常是最好方法：

```
Set<Integer> si = Collections.emptySet();
List<String> ss = Collections.emptyList();
Map<String,Integer> m = Collections.emptyMap();
```

最后，nCopies()会返回不可变的 List，其中包含单一特定对象的特定数量的副本：

```
List<Integer> tenzeros = Collections.nCopies(10, 0);
```

与数组互相转换

由对象和collection所组成的数组是用于类似的用途。list 与数组两者之间互相转换是有可能的：

```
String[] a ={ "this", "is", "a", "test" };   // 一个数组
List<String> l = Arrays.asList(a);           // 将数组视为不可成长的 list
List<String> m = new ArrayList<String>(l);   // 创建可成长的查看副本

// 在Java 5.0 中，asList()是个 varargs method，所以我们也可以这么做：
Set<Character> abc = new HashSet<Character>(Arrays.asList('a', 'b', 'c'));

// 定义了 toArray() method。此无自变量的版本创建了一个 Object[]数组，将
// 元素复制过来并将其返回
Object[] members = set.toArray();            // 让 set 元素成为数组

Object[] items = list.toArray();             // 让 list 元素成为数组
Object[] keys = map.keySet().toArray();      // 让 map 键对象成为数组
Object[] values = map.values().toArray();    // 让 map 值对象成为数组

// 如果你想让返回值是 Object[]以外的东西，可以传入由适当类型组成的
// 数组。如果数组太大，那么被复制到数组的 collection 元素就会以 null
// 结束
String[] c = l.toArray(new String[0]);
```

Collections 实用程序 method

java.util.Arrays 类定义了 method 来操作数组，java.util.Collections 类也定义了 method 来操作 collection。最值得注意就是排序和搜索 collection 元素的 method：

```
Collections.sort(list);
int pos = Collections.binarySearch(list, "key"); // list 必须已被排序
```

Download at http://www.pin5i.com/

这有一些令人感兴趣的其他 Collection method：

```
Collections.copy(list1, list2);  // 将list2复制到list1，改写list1
Collections.fill(list, o);       // 以Object o填充list
Collections.max(c);              // 找出Collection c中的最大元素
Collections.min(c);              // 找出Collection c中的最小元素

Collections.reverse(list);       // 反转list
Collections.shuffle(list);       // 混合list
```

实现 Collection

Java Collections Framework提供了抽象类，使得实现常用的collection的类型变得简单。以下的程序代码扩展AbstractList来定义QuadraticSequence。这是个列表实现，它会依需求计算列表值，而不是把它们实际存放在内存中的某处。请参阅AbstractSet、AbstractMap、AbstractQueue以及AbstractSequentialList。

```
import java.util.*;

/** 不可变的List<Double>表示序列ax^2 + bx + c */
public class QuadraticSequence extends AbstractList<Double> {
    final int size;
    final double a, b, c;

    QuadraticSequence(double a, double b, double c, int size) {
        this.a = a; this.b = b; this.c = c; this.size = size;
    }

    @Override public int size() { return size; }

    @Override public Double get(int index) {
        if (index<0 || index>=size) throw new ArrayIndexOutOfBoundsException();
        return a*index*index + b*index + c;
    }
}
```


线程与并行

Java平台从Java 1.0开始就以Thread类和Runnable接口来支持多线程或并行程序设计。Java 5.0以针对并行程序设计的新的实用程序的广泛组来加强支持。

创建、运行以及操作线程

Java使得在程序中定义与运作多个线程变得容易。java.lang.Thread是Java API中的基础线程类。定义线程的方法有两种：一种是制作Thread的子类，覆盖run() method，然后实例化你的Thread子类；另一种是定义实现了Runnable method的类

Download at http://www.pin5i.com/

(也就是定义 run() method)，然后传递此 Runnable 对象的 instance 给 Thread() 构造函数。无论是哪种方法，结果都是 Thread 对象，其中 run() method 是线程的主体。当你调用Thread对象的start() method时，解释器会创建一个新的线程来执行run() method。此新线程会继续运行，直到 run() method 结束。此时，原来的线程会从接在 start() method 后面的语句开始继续运行其本身。以下程序代码说明这点：

```
final List list; // 某些长的未排序的对象列表；已在其他地方初始化

/** 用来在后台对List排序的Thread类 */
class BackgroundSorter extends Thread {
  List l;
  public BackgroundSorter(List l) { this.l = l; }     // 构造函数
  public void run() { Collections.sort(l); }          // 线程主体
}

// 创建BackgroundSorter线程
Thread sorter = new BackgroundSorter(list);
// 开始运行；在原来的线程继续执行接下来
// 的语句时，新的线程会执行run() method
sorter.start();

// 这是另一个定义类似线程的方法
Thread t = new Thread(new Runnable() {            // 创建一个新的线程
  public void run() { Collections.sort(list); } // 为对象列表排序
});
t.start();                                        // 开始运行
```

线程的生命周期

线程可以处于六种状态中的一种。在 Java 5.0 中，这些状态是由 Thread.State 枚举类型表示，而线程的状态可以用 getState() method 来查询。Thread.State 常量列表很好地提供了线程生命周期的一览：

NEW

Thread 已被创建，但其 start() method 尚未被调用。所有线程都会从这个状态开始。

RUNNABLE

线程正在运行或在操作系统调度它时就可运行。

BLOCKED

因为线程在等待取得锁定以便进入同步 method 或程序块，所以线程并未运行。我们会在本节的稍后看到更多关于同步 method 和程序块的信息。

WAITING

线程因为调用了 Object.wait() 或 Thread.join() 而未运行。

Download at http://www.pin5i.com/

TIMED_WAITING

线程因为调用了 Thread.sleep() 或加上逾时值来调用 Object.wait() 或 Thread.join() 而未运行。

TERMINATED

线程已运行完毕。它的 run() method 已正常结束或通过抛出异常而结束。

线程优先级

线程可以以不同的优先级运行。指定优先级的线程通常会在已无具较高优先级的线程等待时才会运行。在运作线程优先级时，你可以使用这里的一些程序代码：

```
// 将线程优先级设定为低于一般标准
t.setPriority(Thread.NORM_PRIORITY-1);

// 将线程的优先级设定为低于当前线程
t.setPriority(Thread.currentThread().getPriority() - 1);

// 不需等待 I/O 的线程应该要明确地让出 CPU，以让其他具有相同优先
// 级的线程有机会运行
Thread t = new Thread(new Runnable() {
  public void run() {
    for(int i = 0; i < data.length; i++) {   // 逐一处理一组数据
      process(data[i]);                      // 加以处理
      if ((i % 10) == 0)                     // 但在每处理 10 个后，就暂
        Thread.yield();                      // 停以让其他线程运行
    }
  }
});
```

处理未被捕获的异常

线程通常会在到达 run() method 终点或执行到那个 method 中的 return 语句时终止。但是，线程也可以通过抛出异常来终止。当线程以这种方式结束时，默认的行为就是列出线程的名称、异常类型、异常信息以及堆栈追踪。在 Java 5.0 中，你可以对线程中未被捕获的异常安装自定义的处理程序。例如：

```
// 此线程正好抛出一个异常
Thread t = new Thread() {
        public void run() {throw new UnsupportedOperationException();}
    };

// 给线程一个名称以帮助调试
t.setName("My Broken Thread");

// 这是针对错误的处理程序
t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
        public void uncaughtException(Thread t, Throwable e) {
```

Download at http://www.pin5i.com/


```
            System.err.printf("Exception in thread %d '%s':" +
                              "%s at line %d of %s%n",
                              t.getId(),     // 线程id
                              t.getName(),   // 线程名称
                              e.toString(), // 异常名称与信息
                              e.getStackTrace()[0].getLineNumber(), // 行号
                              e.getStackTrace()[0].getFileName());  // 文件名
        }
    });
```

使线程休眠

通常，线程被用来执行一些固定间隔时间的重复工作。在进行牵涉到动画或类似效果的图形程序设计时更是如此。做到这个操作的关键就是让线程在特定的一段时间休眠或停止运行。这可以用静态的 `Thread.sleep()` method 做到，或者在 Java 5.0 中可以用 `TimeUnit` 类的列举常量的实用程序 method：

```
import static java.util.concurrent.TimeUnit.SECONDS;  // 实用程序类

public class Clock extends Thread {
    // 此字段是易变的，因为两个不同的线程会访问它
    volatile boolean keepRunning = true;

    public Clock() {       // 构造函数
        setDaemon(true); //daemon 线程：解释器可以在它运行时跳出
    }

    public void run() {          // 线程主体
        while(keepRunning) {     // 此线程会运行直到被要求停止为止
            long now = System.currentTimeMillis();      // 取得当前时间
            System.out.printf("%tr%n", now);            // 输出当前时间
            try { Thread.sleep(1000); }                 // 等待1000毫秒
            catch (InterruptedException e) { return; } // 在中断时结束
        }
    }

    // 要求线程停止运行。这是interrupt()的替代方案
    public void pleaseStop() { keepRunning = false; }

    // 此method示范如何使用Clock类
    public static void main(String[] args) {
        Clock c = new Clock();                      // 创建Clock线程
        c.start();                                  // 激活它
        try {  SECONDS.sleep(10); }                 // 等待10秒
        catch(InterruptedException ignore) {}       // 忽略中断
        // 现在停止 clock thread。我们也可以使用 c.interrupt()
        c.pleaseStop();
    }
}
```

请注意此范例中的 `pleaseStop()` method：它被设计来以可控制的方式来停止时钟线

Download at http://www.pin5i.com/

程。此范例被编写为可以通过调用从 Thread 继承来的 interrupt() method 来停止。Thread 类定义了 stop() method，但不建议使用。

运行与安排任务

Java 提供了一些方法，可以在无需显式地创建 Thread 对象的情况下，异步地执行任务或安排它们以供将来执行。接下来的章节介绍 Java 1.3 中增加的 Timer 类和 Java 5.0 的 java.util.concurrenct 包的执行者（executor）框架。

使用 Timer 安排工作

在 Java 1.3 中加入的 java.util.Timer 和 java.util.TimerTask 类使得重复任务的执行变得容易。这有一些程序代码的行为与先前显示的 Clock 类非常像：

```
import java.util.*;

// 定义显示时间的任务
TimerTask displayTime = new TimerTask() {
  public void run() { System.out.printf("%tr%n",
System.currentTimeMillis()); }
};
// 创建 timer 对象来执行此任务（或其他任务）
Timer timer = new Timer();
// 现在将那个任务定为每 1000 毫秒执行一次，就从现在开始
timer.schedule(displayTime, 0, 1000);

// 停止显示时间的任务
displayTime.cancel();
```

Executor 接口

在 Java 5.0 中，java.util.concurrent 包包含了 Executor interface。Executor 是个可以执行 Runnable 对象的对象。Executor 的使用者通常不需要知道 Executor 完成工作的方法：它只需要知道 Runnable 会在某个时间点执行。创建 Executor 实现可以使用一些不同的线程策略，从以下程序代码中可较清楚地得知（请注意，此范例也示范了 BolckingQueue 的用法）。

```
import java.util.concurrent.*;

/** 在当前线程中执行Runnable。 */
class CurrentThreadExecutor implements Executor {
    public void execute(Runnable r) { r.run(); }
}

/** 使用新创建的线程来执行各个 Runnable */
class NewThreadExecutor implements Executor {
```

Download at http://www.pin5i.com/


```
    public void execute(Runnable r) { new Thread(r).start(); }
}

/**
 * 创建一个线程来排列Runnable并依序加以执行
 */
class SingleThreadExecutor extends Thread implements Executor {
    BlockingQueue<Runnable> q = new LinkedBlockingQueue<Runnable>();
    public void execute(Runnable r) {
        // 这里不执行Runnable，只是把它放入队列
        // 我们的队列是不受限的，所以应该不会阻塞
        // 由于它不会阻塞，所以绝不会抛出InterruptedException
        try { q.put(r); }
        catch(InterruptedException never) { throw new AssertionError(never); }
    }

    // 这是实际执行了Runnable的线程主体
    public void run() {
        for(;;) {                       // 无限循环
            try {
                Runnable r = q.take();  // 取得下一个Runnable或等待
                r.run();                // 运行!
            }
            catch(InterruptedException e) {
                // 如果被中断，就停止执行已排入队列的Runnable
                return;
            }
        }
    }
}
```

这些示例实现帮助说明Executor运作的方法以及它分隔由安排策略执行任务及线程实现细节的方法。但是，很少有必要真的实现你自己的Executor，因为java.util.concurrent提供了有灵活性且强大的ThreadPoolExecutor类。此类通常会通过Executor类中的静态factory method来使用：

```
Executor oneThread = Executors.newSingleThreadExecutor(); // 1的池的大小
Executor fixedPool = Executors.newFixedThreadPool(10);  // 池中有10个线程
Executor unboundedPool = Executors.newCachedThreadPool(); // 要多少有多少
```

除了这些方便的factory method之外，如果你想指定线程池的最小与最大的大小或想针对不会被线程立即执行的任务来指定队列类型（例如有限制的、无限制的、依优先级排序或同步化）以供使用，也可以显式地创建ThreadPoolExecutor。

ExecutorService

如果你查看了ThreadPoolExecutor或以上Executor factory method所引用的签名，就会知道它是个ExecutorService。ExecutorService interface扩展了Executor，并加上了执行Callable对象的能力。Callable与Runnable很像，但是Callable不

Download at http://www.pin5i.com/

是随意把程序代码封装在 run() method 中，而是把程序代码放在 call() method 中。call()与 run()有两个地方非常不同：它会返回一个结果，而且允许抛出异常。

因为 call()会返回一个结果，所以 Callable interface 会接收结果的类型作为参数。例如，计算大质数的耗时程序块可以被封装在 Callable<BigInteger>中：

```
import java.util.concurrent.*;
import java.math.BigInteger;
import java.util.Random;
import java.security.SecureRandom;

/** 这是针对计算大质数的Callable实现 */
public class RandomPrimeSearch implements Callable<BigInteger> {
    static Random prng = new SecureRandom();  // self-seeding
    int n;
    public RandomPrimeSearch(int bitsize) { n = bitsize ; }
    public BigInteger call() { return BigInteger.probablePrime(n, prng); }
}
```

当然，你可以直接调用任一个 Callable 对象的 call() method，但如果要执行，则要用 ExecutorService 把它传递给 submit() method。因为 ExecutorService 实现通常会以异步方式执行任务，所以 submit() method 不可以只返回 call() method 的结果。submit() method 会返回 Future 对象。Future 只是对未来某个时间的结果承诺，它是对结果的类型来参数化，如以下程序代码片段所示：

```
// 尝试同时计算两个质数
ExecutorService threadpool = Executors.newFixedThreadPool(2);
Future<BigInteger> p = threadpool.submit(new RandomPrimeSearch(512));
Future<BigInteger> q = threadpool.submit(new RandomPrimeSearch(512));
```

一旦你有了 Future 对象，能用来做什么事呢？你可以调用 isDone()来查看 Callable 是否已完成执行；你可以调用 cancel()来取消 Callable 的执行，而且可以调用 isCancelled()来得知 Callable 是否在完成前被取消。但大部分时候，你只会调用 get()来取得 call() method 的结果。如果有必要，get()程序块会等待 call() method 完成。以下程序代码可以用来配合上面显示的 Future 对象使用：

```
BigInteger product = p.get().multiply(q.get());
```

请注意，get() method 可能会抛出 ExecutionExecption。应记住 Callable.call()可以抛出任何种类的异常。如果这个状况发生了，Future 就会把那个异常封装在 ExecutionExcetion 中并从 get()将它抛出。请注意，Future.isDone() method 会把 Callable 想成是“已完成（done）”，即使 call() method 是因为有异常而不正常终止。

Download at http://www.pin5i.com/

ScheduledExecutorService

ScheduledExecutorService是ExecutorService的扩展，它增加了类似Timer的进度安排能力。它能让你将Runnable或Callable安排为延迟一段指定时间之后执行或把Runnable安排为重复执行。在各种情况下，针对未来执行的进度安排工作的结果就是ScheduledFuture对象。Future也实现了Delay interface并提供了getDelay() method，可以用来查询在任务开始之前的剩余时间。

取得ScheduledExecutorService的最简单方法就是使用Executor类的factory method。以下程序代码使用ScheduledExecutorService来重复执行一个动作，也在固定时间间隔之后取消重复的动作。

```
/**
 * 以每秒cps个字符的速度输出随机ASCII字符，总
 * 共持续totalSeconds秒
 */
public static void spew(int cps, int totalSeconds) {
    final Random rng = new Random(System.currentTimeMillis());
    final ScheduledExecutorService executor =
        Executors.newSingleThreadScheduledExecutor();
    final ScheduledFuture<?> spewer =
        executor.scheduleAtFixedRate(new Runnable() {
                public void run() {
                    System.out.print((char)(rng.nextInt('~' - ' ') + ' '));
                    System.out.flush();
                }
            },
                                0, 1000000/cps, TimeUnit.MICROSECONDS);
    executor.schedule(new Runnable() {
            public void run() {
                spewer.cancel(false);
                executor.shutdown();
                System.out.println();
            }
        },
                    totalSeconds, TimeUnit.SECONDS);
}
```

互斥与锁

在使用多线程时，如果你允许多个线程访问同一个数据结构，就必须非常小心。考虑一下，当一个线程正试着逐一处理List中的元素时，而另一个线程正在排序那些元素，这样会发生什么事。预防这类有害的并行操作是多线程计算的主要问题之一。避免两个线程同时访问同一个对象的基本技巧，就是要求线程必须先取得对象的锁，才能加以修改。当任一个线程占有锁时，另一个请求得到锁的线程就必须等待，直到第一个线程完成任务并释放锁。每个Java对象都有基本功能来提供这样的锁定能力。

Download at http://www.pin5i.com/

要让对象具有线程安全性的最简单方式就是把所有具有敏感性的 method 声明为 synchronized。线程必须取得对象的锁才可以执行它的 synchronized method，这代表其他的线程都不可以同时执行任何其他的 synchronized method（如果 static method 被声明为 synchronized，线程就必须取得类的锁且以相同的方式运作）。如果要做到较精细的锁定，你可以指定在短时间内占有指定对象的锁的synchronized程序代码块：

```
// 此method在synchronized块中交换两个数组元素
public static void swap(Object[] array, int index1, int index2) {
  synchronized(array) {
    Object tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
  }
}

// java.util 中的Collection、Set、List 和Map的实现不具
// 有 synchronized method（除了原有的Vector 和Hashtable
// 实现）。在处理多线程时，可以取得同步的封装程序
// 的对象

List synclist = Collections.synchronizedList(list);
Map syncmap = Collections.synchronizedMap(map);
```

java.util.concurrent.locks 包

请注意，当你使用synchronized修饰符或语句时，你所要求的锁是作用在块内，当线程离开method或块时，它就会被自动释放。Java 5.0中的java.util.concurrent.locks包提供了替代方案：你可以用 Lock 对象显式地锁定与开锁。Lock 对象不会自动以块为作用范围，你必须小心使用 try/finally 结构来确保锁一定会被释放。另一方面，Lock 使得只运用以块为作用范围的锁无法达到的算法成为可能，例如以下的“交叉前进（hand-over-hand）”链接列表追踪：

```
import java.util.concurrent.locks.*;  // New in Java 5.0

/**
 * 由类型 E 的值所组成的链接列表的部分实现
 * 它示范了以 Lock 做交叉前进锁定
 */
public class LinkList<E> {
    E value;                    // 列表中 this 节点的值
    LinkList<E> rest;           // 列表的其他部分
    Lock lock;                  // this 节点的锁

    public LinkList(E value) {          // 列表的构造函数
        this.value = value;             // 节点值
        rest = null;                    // 这是列表中的唯一节点
        lock = new ReentrantLock();     // 我们可以锁定 this 节点
    }
```

Download at http://www.pin5i.com/


```
    /**
     * 附加一个节点到列表末端，使用交叉前进锁定来追踪列表。
     * 此method是具有线程安全性的：多个线程可以同时追踪
     * 列表的不同部分
     **/
    public void append(E value) {
        LinkList<E> node = this;   // 从this节点开始
        node.lock.lock();          // 锁定它

        // 循环处理，直到发现列表中的最后一个节点
        while(node.rest != null) {
            LinkList<E> next = node.rest;

            // 这是交叉前进的部分。锁定下一个节点，然后释放
            // 当前节点的锁。我们使用try/finally结构，以
            // 让当前节点即使在对下一个节点的锁定因为异常
            // 而失败时仍能释放锁

            try { next.lock.lock(); }        // 锁定下一个节点
            finally { node.lock.unlock(); } // 释放当前节点的锁
            node = next;
        }

        // 在这时，此节点是列表中的最后一个节点，我们
        // 对它做锁定。使用try/finally来确保释放锁
        try {
            node.rest = new LinkList<E>(value); // 添加新节点
        }
        finally { node.lock.unlock(); }
    }
}
```

死锁

当你使用锁定来预防线程同时访问同一个数据时，就必须小心避免死锁，这会发生在当两个线程互相等待对方释放它们所需的锁时。由于两者都无法继续，也没有任何一方会释放所占有的锁，所以它们都会停止运行。以下程序代码有发生死锁的倾向。死锁是否发生会随系统而异，也会随每次执行而异。

```
// 当两个线程试图锁定两个对象时，死锁就能发生，除
// 非它们都是以相同顺序要求锁
final Object resource1 = new Object();  // 这有两个要锁定的对象
final Object resource2 = new Object();
Thread t1 = new Thread(new Runnable() { // 先锁定resource1，然后锁定resource2
  public void run() {
    synchronized(resource1) {
      synchronized(resource2) { compute(); }
    }
  }
});
```

Download at http://www.pin5i.com/


```
Thread t2 = new Thread(new Runnable() {  // 先锁定resource2，然后锁定resource1
  public void run() {
    synchronized(resource2) {
      synchronized(resource1) { compute(); }
    }
  }
});

t1.start();  // 锁定resource1
t2.start();  // 锁定resource2，现在两个线程都无法进行!
```

协调线程

在多线程程序设计中，要求一个线程等待另一个线程来采取某些操作是很常见的。Java平台提供了一些方法来协调线程，其中包括了内置于Object和Thread类的method以及Java 5.0中引入的“synchronizer”实用程序类。

wait()与notify()

有时线程必须停止运行并等待，直到某个事件发生，在那之后它会被告知继续运行。这可以用wait()和notify() method来实现。但是，这些不是Thread类的method，它们是Object的method。就和每个Java对象都有与其相关联的锁一样，每个对象都可以维持等待线程列表。当线程调用对象的wait() method时，线程所占有的每一个锁都会被暂时释放，而线程会被加到那个对象的等待线程列表并停止运行。当另一个线程调用同一个对象的notifyAll() method时，对象就会唤醒等待的线程并允许它们继续运行：

```
import java.util.*;

/**
 * 一个队列。有个线程会调用push()来把一个对象插入队列。
 * 另一个线程会调用pop()将对象从队列取出。如果没有数据，pop()就
 * 会等待，直到有数据为止。这里所使用的是wait()/notify()。
 * wait()和notify()必须被使用在synchronized method或程序块中。在Java 5.0
 * 中，要用java.util.concurrent.BlockingQueue来代替
 */
public class WaitingQueue<E> {
    LinkedList<E> q = new LinkedList<E>();  // 存储对象的地方
    public synchronized void push(E o) {
        q.add(o);           // 将对象添加到列表的末尾
        this.notifyAll(); // 告诉等待中的线程，数据已准备好
    }
    public synchronized E pop() {
        while(q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException ignore) {}
        }
```

Download at http://www.pin5i.com/


```
            return q.remove(0);
        }
    }
```

请注意，这样的类在 Java 5.0 中是不必要的，因为 java.util.concurrent 定义了 BlockingQueue interface 和例如 ArrayBlockingQueue 这样的一般用途的实现。

等待指定条件

Java 5.0 为对象的 wait() 和 notifyAll() method 提供了替代方案。java.util.concurrent.locks 定义了具有 await() 和 signalAll() method 的 Condition 对象。Condition 对象一定会与 Lock 对象相结合，而且在用法上与置于每个 Java 对象内的锁定和等待能力大都相同。它的主要用途就是让每个 Lock 具有多个 Condition 成为可能，这在使用基于对象的锁定和等待时是不可能的。

等待线程完成

有时一个线程必须停止并等待另一个线程完成。你可以用 join() method 来完成：

```
List list;  // 要被排序的详细列表；已在其他地方初始化

// 定义线程来排序列表：降低其优先级，以让它只
// 有在当前线程在等待I/O时才开始运行
Thread sorter = new BackgroundSorter(list);                    // 先前已定义
sorter.setPriority(Thread.currentThread.getPriority()-1); // 降低优先级
sorter.start();                                                // 开始排序

// 同时，在原先的线程中从文件中读取数据
byte[] data = readData();  // 在其他地方定义的method

// 在能继续处理之前，列表需已被彻底排序，所以如果sorter线程
// 还没完成，则我们必须等它结束
try { sorter.join(); } catch(InterruptedException e) {}
```

同步化实用程序

java.util.concurrent 包含了四个“同步化程序（synchronizer）”类，能通过让线程等待直到出现指定条件为止来同步化并行程序的状态：

Semaphore

Semaphore 类模拟了信号量（semaphore），是传统的并行程序设计结构。概念上，semaphore 代表了一个或多个“许可证”（permit）。需要许可证的线程会调用 acqure()，接着在使用完时调用 release()。如果没有可用的许可证，acquire() 就会停止以使线程暂停，直到另一个线程释放许可证为止。

Download at http://www.pin5i.com/

CountDownLatch

闩(latch)在概念上是任一个具有两种可能状态而且从初始状态到最终状态只会变换一次的变量或并行结构。一旦变换发生，就会永久保持在最终状态。CountDownLatch是个并行实用程序，它可以存在两种状态：关闭与开启。在初始的关闭状态中，调用 await() method 的线程会暂停而且不能继续处理，直到变换为闩开启状态为止。一旦此变换发生，所有等待中的线程就会继续处理，所有将来调用await()的线程不会被暂停。从关闭到开启的变换发生于对countDown()调用指定次数时。

Exchanger

Exchanger 是个实用程序，能让两个线程会合并交换一些值。第一个调用 exchange() method的线程会暂停，直到有第二个线程调用了相同的 method。当这个情况发生时，由第一个线程传递给 exchange() method 的自变量，就会变成第二个线程的 method的返回值，反之亦然。当两个 exchange()调用返回时，这两个线程都能随意继续并行运行。Exchanger是generic类型且使用其类型参数来指定要被交换的值的类型。

CyclicBarrier

CyclicBarrier是个实用程序，能让N个线程的组互相等待以达到同步化的时刻。线程的数量是在CyclicBarrier第一次被创建时所指定的。线程会调用 await() method来暂停，直到最后一个线程调用await()，所有的线程在那个时刻都会再次继续。与CountDownLatch不同的是，CyclicBarrier会重设其计数值，并且可以立即被再次使用。CyclicBarrier在并行算法中很有用，并行算法中的计算会被分为许多部分，每个部分都会由独立的线程处理。在这样的算法中，线程通常必须会合，以让它们的部分解答能合并为完整的解答。为了使这样的操作较为容易，CyclicBarrier构造函数允许你指定最后一个线程所要执行的Runnable对象，它会在其他线程被唤醒并重新运行之前先调用 await()。此Runnable可以提供从各线程的计算组合出解答所需的协调，或指定新的计算给各个线程。

线程中断

在说明 sleep()、join()和wait() method 的范例中，你或许注意到，对这些 method 的调用都是被封装在捕获 InterruptedException的 try 语句中。这是必要的，因为 interrupt() method能让一个线程中断另一个线程的运行。中断的结果取决于你处理 InterruptedException的方法，一般对于被中断的线程较好的响应是停止运行。另一方面，如果你只是捕获并忽略InterruptedException，中断就仅只是停止线程，而不是暂停。

Download at http://www.pin5i.com/

如果interrput() method是在未被暂停的线程上被调用，线程就会继续运行，但它的“中断状态”会被设定，以指出已有请求被中断。线程可以通过调用静态的Thread.interrupted() method来测试它自己的中断状态，如果线程已被中断就会返回true，但副作用就是会清除中断状态。线程可以用isInterrupted()这个instance method来测试另一个线程上的中断状态，此method会查询状态，但不会加以清除。

如果线程在中断状态被设定时调用sleep()、join()或wait()，那么它不会被冻结，而是立即抛出InterruptedException(中断状态会因为抛出异常所造成的副作用而被清除)。同样地，如果调用interrupt() method的线程已在对sleep()、join()或wait()的调用中被冻结，那个线程就会因为抛出InterruptedException而不再被冻结。

线程冻结的最常见情况之一，就是在进行输入/输出时，线程通常必须暂停并等待来自文件系统或网络的数据(用来执行I/O操作的java.io、java.net以及java.nio API会在本章稍后讨论)。很不幸的是，interrupt() method不会唤醒冻结在java.io包的I/O method中的线程。这是java.io的缺点之一，已通过java.nio的New I/O API修正。如果线程在执行任一实现了java.nio.channels.InterruptibleChannel信道上的I/O操作时被中断，信道就会关闭，线程的中断状态会被设定，而且线程会通过抛出java.nio.channels.ClosedByInterruptException被唤醒。如果线程在中断状态已被设定时试图调用阻塞式I/O method，就会发生相同的事情。同样地，如果线程在被冻结在java.nio.channels.Selector的select() method中时被中断(或是在其中断状态已被设定时调用select())，select()就会停止冻结(或永不启动)并立即返回。在这样的情况下，不会有异常被抛出；被中断的线程会醒来，而select()调用会返回。

阻塞式队列

就如本章前面的“Queue与BlockingQueue接口”一节中所提到的，队列是个collection，其中的元素会在“尾端”被插入并在“前端”被移除。Queue interface和各种实现被加入java.util作为Java 5.0的一部分。java.util.concurrent扩展了Queue interface：BlockingQueue定义了put()和take() method，能让你增加或移除队列的元素，它在有必要时会被冻结，直到队列有空间或有元素可被移除为止。阻塞式队列在多线程程序设计中常被使用：一个线程产生了一些对象并把它们放在队列中以供另一个队列消耗——将那些对象从队列移除。

java.util.concurrent提供了五个BlockingQueue的实现：

Download at http://www.pin5i.com/

ArrayBlockingQueue

此实现是以数组为基础，而且就和所有的数组一样，具有在创建时就已建立的固定能力。以降低总处理能力为代价，此队列可在“尚可”模式中操作，线程会因put()或take()而被冻结，元素的取用是依它们到达的顺序。

LinkedBlockingQueue

此实现是以链接列表数据结构为基础。它可以有指定的最大大小，但按照默认，它实际上是无限的。

PriorityBlockingQueue

此无限队列没有实现FIFO（先进先出）顺序，而是以指定Comparator对象为基础来排序其元素，或者如果它们是Comparable对象而且没有被指定Comparaotr时，就是以它们的自然顺序为基础。由take()返回的元素是依据Comparator或Comparable排序的最小元素。关于非阻塞的版本，请参阅java.util.PriorityQueue。

DelayQueue

DelayQueue就像PriorityBlockingQueue，但元素实现了Delayed interface。Delayed是Comparable而且依元素被延迟的时间来排序，但DelayQueue只是个对元素排序的无限队列。它也限制了take()和相关的method，以让元素要到延迟时间已过时才能从队列中被移除。

SynchronousQueue

此类实现了具有能力为零的BlockingQueue的退化状况。对put()的调用会冻结，直到有其他的线程调用了take()，并且此时对take()的调用会冻结，直到有其他线程调用put()。

atomic 变量

java.util.concurrent.atomic包包含了实用程序类，能在不锁定的情况下在字段上进行原子操作（atomic operation）。原子操作是不可分割的操作：其他的线程无法看到在原子操作中的atomic变量。这些实用程序类定义了get()和set()访问method，它们具有volatile字段的特性，但也定义了如“比较与设定（compare-and-set）”和“取得与递增（get-and-increment）”这样行为上的一个原子的复合操作。以下的程序代码示范了AtomicInteger的用法，并与传统synchronized method的用法作了对比：

```
// count1()、count2()和count3()都是具有线程安全性的。两个
// 线程可以同时调用这些method，而且它们绝不会看到相同的返回值
public class Counters {
    // 这个计数器使用了synchronized method和锁定
    int count1 = 0;
    public synchronized int count1() { return count1++; }
```

Download at http://www.pin5i.com/


```
    // 这个计数器使用了在AtomicInteger上的单元递增
    AtomicInteger count2 = new AtomicInteger(0);
    public int count2() { return count2.getAndIncrement(); }

    // 这个乐观的计数器使用了compareAndSet()
    AtomicInteger count3 = new AtomicInteger(0);
    public int count3() {
        // 使用get()来取得计数器值，并使用compareAndSet()加以设定
        // 如果compareAndSet()返回false，就再试一下，直到我们在没有
        // 阻塞的情况下处理完循环
        int result;
        do {
            result = count3.get();
        } while(!count3.compareAndSet(result, result+1));
        return result;
    }
}
```

文件与目录

java.io.File类表示了文件或目录，并定义了一些重要的method来操作文件和目录。但请注意，这些method没有允许你读取文件的内容，那是java.io.FileInputStream的工作 —— 它只是Java中用到的许多种I/O流中的一种，会在接下来的章节中讨论。这里有一些你可以用File做的事：

```
import java.io.*;
import java.util.*;

// 取得用户根目录的名称并以File来表示
File homedir = new File(System.getProperty("user.home"));
// 创建一个File对象来表示那个目录中的文件
File f = new File(homedir, ".configfile");

// 判断文件的大小以及最后修改时间
long filelength = f.length();
Date lastModified = new java.util.Date(f.lastModified());

// 如果文件存在又不是个目录，而且具有可读性，
// 就把它移到新建的目录中
if (f.exists() && f.isFile() && f.canRead()) {          // 检查配置文件
  File configdir = new File(homedir, ".configdir");     // 新的配置目录
  configdir.mkdir();                                    // 创建那个目录
  f.renameTo(new File(configdir, ".config"));           // 将文件移入
}

// 列出根目录中的所有文件
String[] allfiles = homedir.list();

// 列出具有".java"后缀的所有文件
String[] sourcecode = homedir.list(new FilenameFilter() {
  public boolean accept(File d, String name) { return name.endsWith(".java"); }
});
```

Download at http://www.pin5i.com/


```
The File class gained some important additional functionality as of Java 1.2:
// 列出所有文件系统根目录，在Windows上，这会给我们针对所有驱动器
// 盘符的File对象（适用于Java 1.2及之后的版本）
File[] rootdirs = File.listRoots();

// 创建一个锁定文件，然后删除（适用于Java 1.2及之后的版本）
File lock = new File(configdir, ".lock");
if (lock.createNewFile()) {
  // 我们成功地创建了这个文件，现在安排在退出时删除它
  lock.deleteOnExit();

  // 现在执行应用程序安全性，让其他人不能
  // 同时执行它
  ...
}
else {
  // 我们没有创建文件；其他人占有了锁
  System.err.println("Can't create lock file; exiting.");
  System.exit(1);
}

// 创建临时文件以供处理过程中使用（适用于Java 1.2及之后的版本）
File temp = File.createTempFile("app", ".tmp");  // 文件名的前缀和后缀
// 使用temp文件做一些事
  ...
// 在完成时删除它
temp.delete();
```

RandomAccessFile

java.io包也定义了RandomAccessFile类，能让你从文件中的任意位置读取二进制数据。这在指定的情况中很有用，但大部分的应用程序会使用下一章节所描述的流类来循序读取文件。这有个使用RandomAccessFile的简短范例：

```
// 打开文件以供读取/写入（"rw"）访问
File datafile = new File(configdir, "datafile");
RandomAccessFile f = new RandomAccessFile(datafile, "rw");
f.seek(100);                    // 移至文件的第100字节处
byte[] data = new byte[100];    // 创建缓冲区来保存数据
f.read(data);                   // 从文件读取100个字节
int i = f.readInt();            // 从文件读取4字节的整数
f.seek(100);                    // 退回第100字节的地方
f.writeInt(i);                  // 先写入整数
f.write(data);                  // 然后写入100个字节
f.close();                      // 在完成时关闭文件
```

使用 java.io 输入/输出

java.io包定义了大量的类供读取与写入流（或循序）数据。InputStream和OutputStream类是用于读取与写入字节流，而Reader和Writer类是用于读取与写

Download at http://www.pin5i.com/

入字符串流。流可以被嵌套，这代表你可以从读取并处理来自底层Reader流的FilterReader对象读取字符。这个底层的Reader流可以从InputStream读取字节并把它们转换为字符。

读取控制台输入

你可以用流执行一些常见的操作，其中一项是读取用户在控制台的输入：

```
import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.print("What is your name: ");
String name = null;
try {
  name = console.readLine();
}
catch (IOException e) { name = "<" + e + ">"; }  // 这绝不该发生
System.out.println("Hello " + name);
```

从文本文件读取多行

从文件读取多行文本是个类似的操作。以下程序代码会读取整个文本文件并在读到结尾时结束：

```
String filename = System.getProperty("user.home") + File.separator + ".cshrc";
try {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  String line;
  while((line = in.readLine()) != null) {  // 读取一行，检查是否为文件结尾
    System.out.println(line);              // 列出此行
  }
  in.close();    // 当处理完成后一定要关闭流
}
catch (IOException e) {
  // 在这处理FileNotFoundException等异常
}
```

将文本写入文件

在本书各处，你都会看到System.out.purintln() method被用来将文字显示在控制台上。System.out仅只是引用输出流，你可以用类似的技术将文字输出在任何的输出流上。以下程序代码显示如何将文字输出至文件：

```
try {
  File f = new File(homedir, ".config");
  PrintWriter out = new PrintWriter(new FileWriter(f));
```

Download at http://www.pin5i.com/


```
    out.println("## Automatically generated config file. DO NOT EDIT!");
    out.close();  // 已完成写入
  }
  catch (IOException e) { /* 处理异常 */ }
```

读取二进制文件

但是，不是所有的文件都包含文本。以下程序代码将文件视为字节流，并将字节读入大数组中：

```
try {
  File f;                                  // 要读取的文件；已在其他地方初始化
  int filesize = (int) f.length();         // 计算文件大小
  byte[] data = new byte[filesize];        // 创建足够大的数组
  // 创建一个流来读取文件
  DataInputStream in = new DataInputStream(new FileInputStream(f));
  in.readFully(data);  // 将文件内容读入数组
  in.close();
}
catch (IOException e) { /* 处理异常 */ }
```

压缩数据

其他各种Java平台的包定义了专门的流类，来以一些有效的方式操作流数据。以下程序代码说明如何使用来自java.util.zip的流类来计算数据的校验码（checksum），然后在将数据写入文件时加以压缩：

```
import java.io.*;
import java.util.zip.*;

try {
  File f;                                  // 要写入的文件；已在其他地方初始化
  byte[] data;                             // 要写入的数据；已在其他地方初始化
  Checksum check = new Adler32();          // 用来计算简单校验码的对象

  // 创建一个流，将字节写入文件 f
  FileOutputStream fos = new FileOutputStream(f);
  // 创建一个流，将字节压缩并写入 fos
  GZIPOutputStream gzos = new GZIPOutputStream(fos);
  // 创建一个流，计算写入 gzos 的字节的校验码
  CheckedOutputStream cos = new CheckedOutputStream(gzos, check);

  cos.write(data);                 // 现在将数据写入嵌套流
  cos.close();                     // 关闭流的嵌套链
  long sum = check.getValue();     // 取得计算好的校验码
}
catch (IOException e) { /* 处理异常 */ }
```

Download at http://www.pin5i.com/

读取 ZIP 文件

java.util.zip包也包含了ZipFile类，能让你随机访问ZIP文件的入口并让你通过流读取那些入口：

```
import java.io.*;
import java.util.zip.*;
String filename;    // 要读取的文件；已在其他地方初始化
String entryname;   // 从 ZIP 文件读出的入口；已在其他地方初始化
ZipFile zipfile = new ZipFile(filename);                    // 打开 ZIP 文件
ZipEntry entry = zipfile.getEntry(entryname);               // 取得一个入口
InputStream in = zipfile.getInputStream(entry);             // 用来读取入口的流
BufferedInputStream bis = new BufferedInputStream(in);      // 改善效率
// 现在从 bis 读取字节……
// 输出 ZIP 文件的内容
for(java.util.Enumeration e = zipfile.entries(); e.hasMoreElements();) {
  ZipEntry zipentry = (ZipEntry) e.nextElement();
  System.out.println(zipentry.getName());
}
```

计算消息摘要

如果你必须计算具有密码强度的校验码（也就是所谓的消息摘要（message digest）），可以使用 java.security 包中的一个流类。例如：

```
import java.io.*;
import java.security.*;
import java.util.*;


File f;             // 用来读取及计算摘要的文件；已在其他地方初始化
List text = new ArrayList();   // 我们在这存储成行的文本

// 取得可以计算 SHA 消息摘要的对象
MessageDigest digester = MessageDigest.getInstance("SHA");
// 用来从文件 f 读取字节的流
FileInputStream fis = new FileInputStream(f);
// 此流会从 fis 读取字节，并计算 SHA 消息摘要
DigestInputStream dis = new DigestInputStream(fis, digester);
// 此流会从 dis 读取字节，并将它们转换为字符
InputStreamReader isr = new InputStreamReader(dis);
// 此流一次可以读取一行
BufferedReader br = new BufferedReader(isr);
// 现在从流读取文本行
for(String line; (line = br.readLine()) != null; text.add(line)) ;
// 关闭流
br.close();
// 取得消息摘要
byte[] digest = digester.digest();
```

Download at http://www.pin5i.com/

将数据存入数组与从数组取出

到目前为止，我们使用了各种流类来操作流数据，但数据本身最终是来自文件或会被写入控制台。java.io包定义了其他的流类，可以从字节数组或文本字符串读取与写入数据：

```
import java.io.*;

// 设定一个流，它使用字节数组作为目的地
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(baos);
out.writeUTF("hello");              // 将一些字符串数据当作字节写出
out.writeDouble(Math.PI);           // 将浮点值当作字节写出
byte[] data = baos.toByteArray();   // 取得我们写出的字节数组
out.close();                        // 关闭流

// 设定一个流从字符串读取字符
Reader in = new StringReader("Now is the time!");
// 读取字符直到达到终点
int c;
while((c = in.read()) != -1) System.out.print((char) c);
```

以此方式操作的其他类包括了ByteArrayInputStream、StringWriter、CharArrayReader以及CharArrayWriter。

使用管道进行线程通信

PipedInputStream和PipedOutputStream以及它们的基于字符的另两个类——PipedReader和PipedWriter，是由java.io所定义的另一组有趣的流。这些流会被两个想要进行通信的线程成对使用。一个线程会写入字节到PipedOutputStream或写入字符到PipedWriter，而另一个线程会从相应的PipedInputStream或PipedReader读取字节或字符：

```
// 一对连通的管道 I/O流形成了一个管道。一个线程会写入字节到
// PipedOutputStream，而另一个线程会从相应的PipedInputStream
// 读取它们，或针对字符使用PipedWriter/PipedReader
final PipedOutputStream writeEndOfPipe = new PipedOutputStream();
final PipedInputStream readEndOfPipe = new
PipedInputStream(writeEndOfPipe);

// 此线程会从管道读取字节并抛弃它们
Thread devnull = new Thread(new Runnable() {
  public void run() {
    try { while(readEndOfPipe.read() != -1); }
    catch (IOException e) {}  // 忽略它
  }
});
devnull.start();
```

Download at http://www.pin5i.com/

使用 java.net 进行网络连接

java.net 包定义了一些类，它们让网络应用程序的编写变得超乎想象的容易。各种范例如下：

使用 URL 类进行网络连接

最易于使用的网络连接类就是URL，它表示全球资源定位器(uniform resource locator)。不同的Java实现可能支持不同的URL协议，但至少你可以利用其对http://、ftp://以及file://的支持。在Java 1.4中，安全HTTP（secure HTTP）也用https://协议来支持。这有一些使用URL类的方法：

```
import java.net.*;
import java.io.*;

// 创建一些URL对象
URL url=null, url2=null, url3=null;
try {
  url = new URL("http://www.oreilly.com");          // 绝对URL
  url2 = new URL(url, "catalog/books/javanut4/"); // 相对URL
  url3 = new URL("http:", "www.oreilly.com", "index.html");
} catch (MalformedURLException e) { /* 忽略此异常 */ }

// 从输入流读取URL的内容
InputStream in = url.openStream();

// 为能对读取过程做更多的控制，取得URLConnection对象
URLConnection conn = url.openConnection();

// 现在取得一些关于URL的信息
String type = conn.getContentType();

String encoding = conn.getContentEncoding();
java.util.Date lastModified = new java.util.Date(conn.getLastModified());
int len = conn.getContentLength();

// 如果有需要，就使用流读取URL的内容
InputStream in = conn.getInputStream();
```

运用 Socket

有时你需要对网络应用程序做比URL类所能做到的程度还高的控制。在这样的情况下，你可以使用Socket直接与服务器通信。例如：

```
import java.net.*;
import java.io.*;

// 这是个简单的客户端程序，它会联机至web server，请求一个
// 文件并从服务器读取文件
```

Download at http://www.pin5i.com/


```
String hostname = "java.oreilly.com";   // 要联机的服务器
int port = 80;                          // HTTP 的标准端口
String filename = "/index.html";        // 要从服务器读取的文件
Socket s = new Socket(hostname, port);  // 联机至服务器

// 取得可用来与服务器通信的 I/O 流
InputStream sin = s.getInputStream();
BufferedReader fromServer = new BufferedReader(new InputStreamReader(sin));
OutputStream sout = s.getOutputStream();
PrintWriter toServer = new PrintWriter(new OutputStreamWriter(sout));

// 使用 HTTP 协议请求来自服务器的文件
toServer.print("GET " + filename + " HTTP/1.0\r\n\r\n");
toServer.flush();

// 现在读取服务器的响应，假设它是个文本文件并把它输出
for(String l = null; (l = fromServer.readLine()) != null; )
  System.out.println(l);

// 在完成时把一切关闭
toServer.close();
fromServer.close();
s.close();
```

使用 SSL 处理 Secure Socket

在 Java 1.4 中，Java Secure Socket Extension（或 JSSE）被加到包 javax.net 和 javax.net.ssl 中的核心 Java 平台。此 API 加密了在 socket 上使用 SSL（Secure Sockets Layer，也就是所谓的 TLS）协议的网络通信。SSL 被广泛用于 Internet 上：它是使用了 https://协议的安全 web 通信的基础。在 Java 1.4 和之后的版本中，你可以用 https://配合如前所示的 URL 类，来安全地下载来自支持 SSL 的 web 服务器的文件。

就和所有的 Java 安全 API 一样，JSSE 具高度可配置性，并对所有在 SSL socket 上设定与通信的细节给予低级的控制。javax.net 和 javax.net.ssl 包相当复杂，但实际上，你只需要一些类来安全地与服务器通信。以下程序是先前的程序代码的变体，它使用了 HTTPS 来代替 HTTP，以安全地传输请求 URL 的内容：

```
import java.io.*;
import java.net.*;
import javax.net.ssl.*;
import java.security.cert.*;

/**
 * 使用 HTTPS 从 web 服务器取得文件。用法：
 *   java HttpsDownload <hostname> <filename>
 **/
public class HttpsDownload {
    public static void main(String[] args) throws IOException {
        // 取得用来建立 SSL socket 的 SocketFactory 对象
```

Download at http://www.pin5i.com/


```
        SSLSocketFactory factory =
            (SSLSocketFactory) SSLSocketFactory.getDefault();

        // 使用factory来建立secure socket，联机至
        // 指定web服务器的HTTPS端口
        SSLSocket sslsock=(SSLSocket)factory.createSocket(args[0], // 主机名称
                                                          443);    // HTTPS端口
        // 取得由web服务器所提供的证书
        SSLSession session = sslsock.getSession();
        X509Certificate cert;
        try { cert = (X509Certificate)session.getPeerCertificates()[0]; }
        catch(SSLPeerUnverifiedException e) { // 如果没有证书或证书无效
            System.err.println(session.getPeerHost() +
                               " did not present a valid certificate.");
            return;
        }

        // 显示关于认证的细节
        System.out.println(session.getPeerHost() +
                           " has presented a certificate belonging to:");
        System.out.println("\t[" + cert.getSubjectDN().getName() + "]");
        System.out.println("The certificate bears the valid signature of:");
        System.out.println("\t[" + cert.getIssuerDN().getName() + "]");

        // 如果用户不信任此认证就放弃
        System.out.print("Do you trust this certificate (y/n)? ");
        System.out.flush();
        BufferedReader console =
            new BufferedReader(new InputStreamReader(System.in));
        if (Character.toLowerCase(console.readLine().charAt(0)) != 'y') return;

        // 现在使用secure socket就像使用正常的socket一样
        // 首先，在SSL socket上送出正常的HTTP请求
        PrintWriter out = new PrintWriter(sslsock.getOutputStream());
        out.print("GET " + args[1] + " HTTP/1.0\r\n\r\n");
        out.flush();

        // 接着，读取服务器的响应并将它输送至控制台
        BufferedReader in =
         new BufferedReader(new InputStreamReader(sslsock.getInputStream()));
        String line;
        while((line = in.readLine()) != null) System.out.println(line);

        // 最后，关闭socket
        sslsock.close();
    }
}
```

服务器

客户端应用程序使用Socket与服务器通信，服务器也做了相同的事：它使用Socket对象与各个客户端通信。但是，服务器有额外的工作，它必须能识别并接收客户端的联

Download at http://www.pin5i.com/

机请求。这是用ServerSocket类实现的。以下程序代码展示了使用ServerSocket的方法。此程序代码实现了简单的HTTP服务器，它会通过回送（或镜像）HTTP请求的全部内容来响应所有的请求。在对HTTP客户端调试时，像这样的虚拟服务器是很有用的：

```
import java.io.*;
import java.net.*;

public class HttpMirror {
  public static void main(String[] args) {
    try {
      int port = Integer.parseInt(args[0]);          // 要监听的端口
      ServerSocket ss = new ServerSocket(port);      // 建立socket来监听
      for(;;) {                                      // 无限循环
        Socket client = ss.accept();                 // 等待联机
        ClientThread t = new ClientThread(client);   // 用一个线程来处理
        t.start();                                   // 让线程开始运行
      }                                              // 再次循环
    }
    catch (Exception e) {
      System.err.println(e.getMessage());
      System.err.println("Usage: java HttpMirror <port>;");
    }
  }

  static class ClientThread extends Thread {
    Socket client;
    ClientThread(Socket client) { this.client = client; }
    public void run() {
      try {
        // 让流与客户端通信
        BufferedReader in =
          new BufferedReader(new InputStreamReader(client.getInputStream()));
        PrintWriter out =
          new PrintWriter(new OutputStreamWriter(client.getOutputStream()));

        // 发送HTTP响应报头至客户端
        out.print("HTTP/1.0 200\r\nContent-Type: text/plain\r\n\r\n");

        // 读取来自客户端的HTTP请求并立刻响应
        // 当我们从客户端读到用来发送请求及其发送结束
        // 的空白行时就停止

        String line;
        while((line = in.readLine()) != null) {
          if (line.length() == 0) break;
          out.println(line);
        }

        out.close();
        in.close();
        client.close();
      }
```

Download at http://www.pin5i.com/


```
        catch (IOException e) { /* 忽略异常 */ }
      }
    }
  }
```

此服务器程序代码可以用JSSE来修改以支持SSL联机。但是，让服务器具有安全性是比让客户端具有安全性更复杂的，因为服务器必须具有可以给客户端的证书。因此，服务器端的JSSE不在此处示范。

数据报（datagram）

URL和Socket都是在基于流的网络联机上执行网络连接的。但是，要跨网络设定与维护流，则需要在网络层上做一些事。有时你需要以低层次的方法来加速跨网络的数据包传输速度，但你不必在意流的维护。此外，如果你不需要保证数据包的到达或数据包以和发送相同的顺序到达，就会对DatagramSocket和DatagramPacket类感兴趣：

```
import java.net.*;

// 使用 datagram 送出信息到另一台计算机
try {
  String hostname = "host.example.com";        // 数据所要送达的计算机
  InetAddress address =                        // 将DNS主机名称转换为
    InetAddress.getByName(hostname);           // 低层次的IP地址
  int port = 1234;                             // 所要联机的端口
  String message = "The eagle has landed.";    // 要送出的信息
  byte[] data = message.getBytes();            // 将字符串转换为字节
  DatagramSocket s = new DatagramSocket();     // 要送出信息的socket
  DatagramPacket p =                           // 创建包以传送
    new DatagramPacket(data, data.length, address, port);
  s.send(p);                                   // 现在进行传送!
  s.close();                                   // 在完成时一定要关闭socket
}
catch (UnknownHostException e) {}  // 由InetAddress.getByName()抛出
catch (SocketException e) {}       // 由new DatagramSocket()抛出
catch (java.io.IOException e) {}   // 由DatagramSocket.send()抛出

// 这里是其他计算机收到 datagram 的方式
try {
  byte[] buffer = new byte[4096];                // 用来保存数据的缓冲区

  DatagramSocket s = new DatagramSocket(1234);   // 用来接收的socket
  DatagramPacket p =
    new DatagramPacket(buffer, buffer.length);   // 要被接收的包
  s.receive(p);                                  // 等待包到达
  String msg =                                   // 将包从字节转
    new String(buffer, 0, p.getLength());        // 换为字符串
  s.close();                                     // 一定要关闭socket
}
catch (SocketException e) {}       // 由new DatagramSocket()抛出
catch (java.io.IOException e) {}   // 由DatagramSocket.receive()抛出
```

Download at http://www.pin5i.com/

测试主机的可达性（reachability）

在 Java 5.0 中，InetAddress 类有个 isReachable() method，它会尝试判定主机是否可达。以下程序代码在 Unix *ping* 实用程序的原生 Java 实现中使用：

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Ping {
    public static void main(String[] args) throws IOException {
        try {
            String hostname = args[0];
            int timeout = (args.length > 1)?Integer.parseInt(args[1]):2000;
            InetAddress[] addresses = InetAddress.getAllByName(hostname);
            for(InetAddress address : addresses) {
                if (address.isReachable(timeout))
                    System.out.printf("%s is reachable%n", address);
                else
                    System.out.printf("%s could not be contacted%n", address);
            }
        }
        catch (UnknownHostException e) {
            System.out.printf("Unknown host: %s%n", args[0]);
        }
        catch(IOException e) { System.out.printf("Network error: %s%n", e); }
        catch (Exception e) {
            // ArrayIndexOutOfBoundsException 或 NumberFormatException
            System.out.println("Usage: java Ping <hostname> [timeout in ms]");
        }
    }
}
```


使用 java.nio 进行 I/O 与网络连接

Java 1.4 针对高性能、非阻塞式（nonblocking）I/O 和网络连接引入了全新的 API。此 API 主要是由三个新包组成：java.nio 定义了 Buffer 类，用来存储一连串的字节或其他基本值；java.nio.channels 定义了信道，数据可通过它在缓冲区和数据来源地或目的端（data source or sink，例如文件或网络 socket）之间传输，此包也包含了用于非阻塞式 I/O 的重要类；最后，java.nio.charset 包包含了有效地将字节缓冲区转换为字符缓冲区的类。以下章节包含了使用这三种包的范例以及使用 New I/O API 进行特定 I/O 工作的范例。

基本 Buffer 操作

java.nio 包包含了抽象的 Buffer 类，它定义了在缓冲区上的一般操作。此包也定义了类型特有的子类，例如 ByteBuffer、CharBuffer 和 IntBuffer。以下程序代码说

Download at http://www.pin5i.com/

明在ByteBuffer上的典型的序列缓冲区操作。其他类型特有的缓冲区类也有类似的method。

```
import java.nio.*;

// Buffer不具有公用构造函数，它们是被分配的
ByteBuffer b = ByteBuffer.allocate(4096); // 创建4096字节的缓冲区
// 或是这样做来试着从低层次的OS取得有效的缓冲区
ByteBuffer buf2 = ByteBuffer.allocateDirect(65536);
// 这是另一个取得缓冲区的方法：通过“封装”一个数组
byte[] data;  // 假设数组已创建且在其他地方初始化
ByteBuffer buf3 = ByteBuffer.wrap(data);  // 创建使用数组的缓冲区
// 创建“查看缓冲区”以把字节当成其他类型来查看也是有可能的
buf3.order(ByteOrder.BIG_ENDIAN);          // 指定缓冲区的字节顺序
IntBuffer ib = buf3.asIntBuffer();         // 把那些字节当作整数来查看

// 现在存储一些数据到缓冲区
b.put(data);        // 把数组的字节复制到缓冲区的当前位置
b.put((byte)42);    // 把另一个字节存储在新的当前位置
b.put(0, (byte)9); // 改写缓冲区中的第一个字节，不改变位置
b.order(ByteOrder.BIG_ENDIAN);  // 设定缓冲区的字节顺序
b.putChar('x');        // 把2个字节的Unicode字符存储在缓冲区
b.putInt(0xcafebabe); // 把4个字节的int存储到缓冲区

// 这些是查询关于缓冲区基本数值的method
int capacity = b.capacity();  // 此缓冲区可以保存多少字节？ (4,096)
int position = b.position();  // 下个要被写入或读取的字节在哪里？
// 缓冲区的界限指定了缓冲区有多少字节可以被使用
// 在写入到缓冲区时，这应该就是容量；从缓冲区读取数据
// 时，这应该就是先前写入的字节数量
int limit = b.limit();            // 应该要使用多少？
int remaining = b.remaining();  // 还剩下多少？返回界限位置
boolean more=b.hasRemaining();  // 测试缓冲区中是否还有空间

// 位置（position）和界限（limit）也可以用具有相同名称的method设定
// 假设你要读取已写入缓冲区的字节
b.limit(b.position());            // 将界限设定为当前位置
b.position(0);                    // 将界限设定为0，从开头开始读取

// 你通常不会用前面的两个调用，而是会用方便的method
b.flip();    // 将界限设定为位置，位置设定为0；准备读取
b.rewind(); // 将位置设定为0；不改变界限；准备读取
b.clear();  // 将位置设定为0，界限设定为容量；准备写入

// 假设你已调用flip()，就可以开始从缓冲区读取字节
buf2.put(b);          // 从b读取所有字节并把它们放入buf2
b.rewind();           // 将b倒回以便从开头再次读取
byte b0 = b.get();   // 读取第一个字节；递增缓冲区位置
byte b1 = b.get();   // 读取第二个字节；递增缓冲区位置
byte[] fourbytes = new byte[4];
b.get(fourbytes);    // 读取接下来的4个字节并将缓冲区位置加4
byte b9 = b.get(9); // 读取第10个字节，但不改变当前位置
int i = b.getInt(); // 把接下来的4个字节当成整数来读取；位置加4
```

Download at http://www.pin5i.com/


```
// 丢弃已读取的字节，将剩余的移转至缓冲区的开
// 头；将位置设定为界限，将界限设定为容量，准备
// 好缓冲区以供写入更多字节
b.compact();
```

你或许注意到，许多缓冲区 method 会返回它们所操作的对象。这让 method 调用在程序代码中可以被“链接”，如下：

```
ByteBuffer bb=ByteBuffer.allocate(32).order(ByteOrder.BIG_ENDIAN).putInt(1234);
```

在 java.nio 及其子包中的许多 method 会返回当前对象，使得这种 method 链接成为可能。请注意，使用这种链接是个风格上的选择（我在本章中避开了），在效率上并没有重大的影响。

ByteBuffer 是最重要的缓冲区类。然而，另一个常用的类是 CharBuffer。CharBuffer 对象可以通过封装字符串来创建，也可以被转换为字符串。CharBuffer 实现了新的 java.lang.CharSequence interface，这代表在特定的应用程序中，它可以像 String 或 StringBuffer 一样被使用（亦即用于正则表达式模式匹配）。

```
// 从字符串创建只读CharBuffer
CharBuffer cb = CharBuffer.wrap("This string is the data for the
CharBuffer");
String s = cb.toString();  // 用toString() method 或依据 toString()的
System.out.println(cb);    // 隐式调用来转换为字符串
char c = cb.charAt(0);     // 使用CharSequence method 来取得字符
char d = cb.get(1);        // 或使用CharBuffer 绝对位置来读取
// 相对位置读取字符并递增当前位置
// 请注意，当CharBuffer 被转换为 String 或被当成CharSequence 使用
// 时，只有当前位置到界限的字符会被使用
char e = cb.get();
```

ByteBuffer 中的字节通常会被转换为 CharBuffer 中的字符，反之亦然。当我们考虑到 java.nio.charset 包时，就会看到要如何操作。

基本信道操作

缓冲区本身并没有什么用，把字节存入缓冲区只是要把它们再次读出，其他没什么内容。但是，缓冲区通常会配合信道一起使用：你的程序把字节存入缓冲区，然后把缓冲区传递给信道，信道把字节从缓冲区读出并将它们写入文件、网络 socket 或其他目的地。或者反过来看，你的程序把缓冲区传给信道，信道从文件、socket 或其他来源地读取字节并把那些字节存储在缓冲区中，然后这些字节就能被你的程序获取。java.nio.channels 包定义了代表文件、socket、datagram 以及管道等信道类（我们会在本章的稍后看到这些具体类的范例）。但是，以下程序代码是以由 java.nio.channels 所定义的各种信道 interface 的功能为基础，应该能配合任一个 Channel 对象运行：

Download at http://www.pin5i.com/


```
Channel c;  // 实现了Channel interface的对象，已在其他地方初始化
if (c.isOpen()) c.close();  // 这是唯一由Channel 定义的method

// read()和write() method是由ReadableByteChannel
// 和WritableByteChannel interface所定义
ReadableByteChannel source;       // 已在其他地方初始化
WritableByteChannel destination; // 已在其他地方初始化
ByteBuffer buffer = ByteBuffer.allocateDirect(16384); // 低层次的16 KB缓冲区

// 这是用于从源信道读取字节并将它们写入目的信道
// 的基本循环，它会执行到不再有字节可从来源地读取以
// 及不再有已在缓冲区的字节要写入目的地为止
while(source.read(buffer) != -1 || buffer.position() > 0) {
  // 滚动缓冲区：将界限设定为当前位置，当目前位置设定为0。这会
  // 让缓冲区准备好供读取（这是由信道 * 写入 * 操作完成）
  buffer.flip();
  // 将缓冲区中的一些或全部字节写入目的地
  destination.write(buffer);
  // 丢弃已被写入的字节，将剩余的复制到缓冲区的开头。将当前
  // 位置设定为界限，将界限设定为容量，将缓冲区准备好以供写入
  // （这是由信道 * 读取 * 操作完成）
  buffer.compact();
}

// 不要忘记关闭信道
source.close();
destination.close();
```

除了先前的程序代码中介绍的ReadableByteChannel和WritableByteChannel interface之外，java.nio.channels还定义了几个其他的信道interface。ByteChannel只是扩展具有可读性和具有可写性的interface，但没有增加任何的新method。它是有用的信道简略表达方式，同时支持读取与写入。GatheringByteChannel是WritableByteChannel的扩展，它定义了write() method，会从多个缓冲区收集数据并将数据写至多个缓冲区。同样地，ScatteringByteChannel是ReadableByteChannel的扩展，它定义了read() method，会从信道读取字节并把它们传播或分布到多个缓冲区。当处理使用固定大小报头的网络协议，而且你想把报头与其他要被传输的数据分开存储在缓冲区时，收集和传播的write()与read() method会很有用。

有个会令人混淆、必须意识到的一点，就是信道读取操作牵涉到将字节写入（或放入）缓冲区，而信道写入操作牵涉到从缓冲区读取（或取得）字节。因此，当我说flip() method准备好缓冲区以供读取时，我是指它准备了缓冲区以供信道write()操作使用！对于缓冲区的compact() method，相反的操作也成立。

使用 Charset 编码与译码文字

java.nio.charset.Charset对象代表一个字符集以及那个字符集的编码。Charset

Download at http://www.pin5i.com/

以及与其相结合的类，CharsetEncoder和CharsetDecoder，定义了一些method用来将字符组成的字符串编码为字节序列以及将字节序列译码为由字符组成的字符串。由于这些类是New I/O API的一部分，它们使用了ByteBuffer和CharBuffer类：

```
// 最简单的情况。使用Charset的便捷例程来转换
Charset charset = Charset.forName("ISO-8859-1"); // 取得Latin-1 Charset
CharBuffer cb = CharBuffer.wrap("Hello World");  // 要编码的字符
// 将字符编码并将字节存储在新分配的ByteBuffer中
ByteBuffer bb = charset.encode(cb);
// 将这些字节译码为新分配的CharBuffer并把它们输出
System.out.println(charset.decode(bb));
```

请注意此范例中使用的ISO-8859-1（又名Latin-1）字符集（charset）。此8位字符集适用于大部分的西欧语言，其中包括英文。只使用英文的程序员也可以使用7位的US-ASCII字符集。Charset类本身并未进行编码和译码，而先前的便捷例程在内部创建了CharsetEncoder和CharsetDecoder类。如果你计划要编码或译码多次，那么自行创建这些对象会更有效率：

```
Charset charset = Charset.forName("US-ASCII");    // 取得字符集
CharsetEncoder encoder = charset.newEncoder();    // 从中创建编码器
CharBuffer cb = CharBuffer.wrap("Hello World!"); // 取得CharBuffer
WritableByteChannel destination;                  // 已在其他地方初始化
destination.write(encoder.encode(cb));            // 编码字符并写入
```

前面的CharsetEncoder.encode() method在每次被调用时，都必须分配新的ByteBuffer。如果要取得最大的效率，你可以调用较低层次的method将编码和译码处理到现有的缓冲区中：

```
ReadableByteChannel source;                       // 已在其他地方初始化
Charset charset = Charset.forName("ISO-8859-1"); // 取得字符集
CharsetDecoder decoder = charset.newDecoder();    // 从中创建译码器
ByteBuffer bb = ByteBuffer.allocateDirect(2048); // 用来保存字节的缓冲区
CharBuffer cb = CharBuffer.allocate(2048);        // 用来保存字符的缓冲区

while(source.read(bb) != -1) {   // 从信道读取字节，直到EOF为止
  bb.flip();                     // 滚动字节缓冲区以准备解码
  decoder.decode(bb, cb, true);  // 将字节译码至字符
  cb.flip();                     // 滚动字符缓冲区以准备输出
  System.out.print(cb);          // 输出字符
  cb.clear();                    // 清除字符缓冲区
  bb.clear();                    // 准备字节缓冲区以供下一个信道读取
}
source.close();                  // 已使用完信道，所以将它关闭
System.out.flush();              // 确定所有的输出字符都已出现
```

前面的程序代码依据的事实是ISO-8859-1是8位编码字符集以及在字符和字节之间有一对一的映射。但是，对于较复杂的字符集，例如Unicode的UTF-8编码或日文所使用的EUC-JP字符集，一个字节不够用，有些（或全部）字符必须要有多个字节。在遇到这

Download at http://www.pin5i.com/

种情况时，无法保证缓冲区中所有的字节可以一次性被编码（缓冲区的结尾可能包含了不完整的字符）。此外，由于一个字符可能被译码为多个字节，所以要知道给定的字符串会被译码为多少字节是有困难的。以下程序代码显示一个循环，你可以用它来以较一般的方式译码字节：

```
ReadableByteChannel source;                          // 已在其他地方初始化
Charset charset = Charset.forName("UTF-8");          // Unicode 编码
CharsetDecoder decoder = charset.newDecoder();       // 从中创建译码器
ByteBuffer bb = ByteBuffer.allocateDirect(2048);     // 用来保存字节的缓冲区
CharBuffer cb = CharBuffer.allocate(2048);           // 用来保存字符的缓冲区

// 告诉译码器忽略坏字节可能产生的错误
decoder.onMalformedInput(CodingErrorAction.IGNORE);
decoder.onUnmappableCharacter(CodingErrorAction.IGNORE);

decoder.reset();                          // 如果译码器在之前已被使用过，则将它重设
while(source.read(bb) != -1) {            // 从信道读取字节，直到 EOF
  bb.flip();                              // 滚动字节缓冲区以准备译码
  decoder.decode(bb, cb, false);          // 将字节译码为字符
  cb.flip();                              // 滚动字符缓冲区以准备输出
  System.out.print(cb);                   // 输出字符
  cb.clear();                             // 清除字符缓冲区
  bb.compact();                           // 丢弃已译码的字节
}
source.close();                           // 已使用完信道，将它关闭
// 在这时，缓冲区中可能还有一些字节要译码
bb.flip();                                // 准备解码
decoder.decode(bb, cb, true);             // 返回 true 以指出这是最后一个调用
decoder.flush(cb);                        // 输出最后一个字符
cb.flip();                                // 滚动字符缓冲区
System.out.print(cb);                     // 输出最后的字符
```

文件处理

FileChannel 是个具体的Channel 类，它会执行文件 I/O 并实现了ReadableByteChannel和WritableByteChannel（虽然它的read() method只有在底层的文件已打开来供读取时才能运作，而它的write() method只有在文件已开启来供写入时才能运作）。通过使用java.io 包取得FileChannel 对象来创建FileInputStream、FileOutputStream或RandomAccessFile，然后调用那个对象的getChannel() method（Java 1.4 中增加的）。举例来看，你可以用两个FileChannel对象来复制文件：

```
String filename = "test";    // 要复制的文件名称
// 创建流来读取源文件并写至副本
FileInputStream fin = new FileInputStream(filename);
FileOutputStream fout = new FileOutputStream(filename + ".copy");
// 使用流来创建相对应的信道对象
FileChannel in = fin.getChannel();
FileChannel out = fout.getChannel();
```

Download at http://www.pin5i.com/


```
// 针对副本配置低层次的 8kB 缓冲区
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
while(in.read(buffer) != -1 || buffer.position() > 0) {
  buffer.flip();        // 准备从缓冲区读取并写入至文件
  out.write(buffer);    // 写入一些或全部缓冲区内容
  buffer.compact();     // 丢弃所有已写入的字节并准备从文件读取
}                       // 更多字节，将它们存入缓冲区
in.close();             // 当使用完时，一定要关闭信道和流
out.close();
fin.close();            // 请注意，关闭 FileChannel 并不会
fout.close();           // 自动地关闭底层的流
```

FileChannel 具有特殊的 transferTo() 和 transferFrom() method，让从 FileChannel 传送指定数量的字节到其他指定的信道或从其他的信道到 FileChannel 变得特别容易（而且在许多操作系统上特别有效率）。这些 method 能让我们简化前面的文件复制程序代码如下：

```
FileChannel in, out;                // 假设这些就像先前的范例中一样
                                    // 被初始化
long numbytes = in.size();          // 源文件中的字节数量
in.transferTo(0, numbytes, out);    // 传送那个数量到输出信道
```

可以用 transferFrom() 代替 transferTo() 写出有相同功能的程序代码（请注意这两个 method，可知的是它们的自变量会有不同的顺序）：

```
long numbytes = in.size();
out.transferFrom(in, 0, numbytes);
```

FileChannel 具有其他信道类所没有的功能，其中最重要的一个，就是以“内存映射”(memory map)文件或一部分文件的功能，亦即取得代表文件内容的 MappedByteBuffer (ByteBuffer 的子类)，并且允许你只要通过对缓冲区做读出和写入，就能读取及选择性地写入文件内容。以内存映射文件是消耗甚大的操作，所以这个技巧通常只在你需要重复访问大型文件时才会有效率。内存映射还提供了另一个方法来执行先前所展示的同一文件复制操作：

```
long filesize = in.size();
ByteBuffer bb = in.map(FileChannel.MapMode.READ_ONLY, 0, filesize);
while(bb.hasRemaining()) out.write(bb);
```

由 java.nio.channels 所定义的信道 interface 包含了 ByteChannel 而没有 CharChannel。信道 API 是低层次的而且只提供了读取字节的 method。先前的范例都把文件当作二进制文件。使用前面介绍过的 CharsetEncoder 和 CharsetDecoder 类来在字节和字符之间转换是有可能的，但当你想处理文本文件时，使用 java.io 包的 Reader 和 Writer 类通常会比 CharBuffer 更容易。幸好 Channels 类定义了简单的 method，在原有的和新的 API 之间搭起了桥梁。以下程序代码把 Reader 和 Writer 对

Download at http://www.pin5i.com/

象封入输入和输出信道内，从输入信道读取成行的Latin-1文本，并通过将编码改为UTF-8来把它们写至输出信道。

```
ReadableByteChannel in;        // 假设这些已在其他地方初始化
WritableByteChannel out;
// 从FileChannel和字符集名称创建Reader和Writer
BufferedReader reader=new BufferedReader(Channels.newReader(in, "ISO-8859-1"));
PrintWriter writer = new PrintWriter(Channels.newWriter(out, "UTF-8"));
String line;
while((line = reader.readLine()) != null) writer.println(line);
reader.close();
writer.close();
```

与FileInputSterm及FileOutput类不同的是，FileChannel类允许对文件内容进行随机访问。零自变量的position() method会返回文件指针（下一个要被读取的字节的位置），而一个自变量的position() method能让你把这个指针设定为你所想要的任何值，这让你能以java.io.RandomAccessFile所用的方法在文件中随机访问。这里有个范例：

```
// 假设你有个分布了数据记录的文件，而文件最后的1024个
// 字节是提供那些记录的位置的索引。这里的程序代码会
// 读取文件的索引，查找文件中第一次记录的位置，然后读
// 取记录

FileChannel in = new FileInputStream("test.data").getChannel(); // 信道
ByteBuffer index = ByteBuffer.allocate(1024);    // 用来保存索引的缓冲区
long size = in.size();                           // 文件的大小
in.position(size - 1024);                        // 索引的开始位置
in.read(index);                                  // 读取索引
int record0Position = index.getInt(0);           // 取得第一个索引入口
in.position(record0Position);                    // 移到文件中的那个位置
ByteBufer record0 = ByteBuffer.allocate(128);    // 取得缓冲区以保存数据
in.read(record0);                                // 最后，读取记录
```

我们在这里所考虑到的FileChannel决定性特性，就是它对所有并行访问（互斥锁）或并行写入（共享锁）锁定整个文件或一部分文件的功能（请注意，有些操作系统严格地强制执行所有的锁，而另一些则只提供了咨询性的锁定能力，它会要求程序互相协调并尝试在读取与写入共享文件的一部分之前先取得锁）。在前面的随机访问范例中，假设我们想确定在我们读取记录数据时没有其他程序正在修改它。我们可以用以下程序代码取得在那部分文件上的共享锁：

```
FileLock lock = in.lock(record0Position, // 锁定区域的开始处
                        128,     // 锁定区域的长度
                        true);   // 共享锁：预防并行更新
                                 // 但允许并行读取
in.position(record0Position);    // 移至索引开始处
in.read(record0);                // 读取索引数据
lock.release();                  // 已使用完锁，所以将它释放
```

Download at http://www.pin5i.com/

客户端网络连接

New I/O API 包含了网络连接功能和文件访问功能。如果要通过网络来通信，可以使用 SocketChannel 类。使用静态的 open() method 创建 SocketChannel，然后与其他的信道对象之间进行字节的读取与写入。以下程序代码使用 SocketChannel 来传送 HTTP 请求到 web 服务器，并将服务器的响应（包括所有的 HTTP 报头）存储在文件中。请注意 java.net.InetSocketAddress 的用法，它是 java.net.SocketAddress 的子类，用来告诉 SocketChannel 联机的对象。这些类也被引入作为 New I/O API 的一部分。

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

// 创建一个 SocketChannel，联机至 www.oreilly.com 的 web 服务器
SocketChannel socket =
  SocketChannel.open(new InetSocketAddress("www.oreilly.com",80));
// 用于编码 HTTP 请求的字符集
Charset charset = Charset.forName("ISO-8859-1");
// 发送 HTTP 请求至服务器。以字符串开始，将它封装为 CharBuffer，编码
// 为 ByteBuffer，然后写入 socket
socket.write(charset.encode(CharBuffer.wrap("GET / HTTP/1.0\r\n\r\n")));
// 创建 FileChannel 来存储服务器的响应
FileOutputStream out = new FileOutputStream("oreilly.html");
FileChannel file = out.getChannel();
// 取得缓冲区以在从 socket 转移到文件时保存字节
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
// 现在循环处理，直到所有字节都已从 socket 读取并写入文件
while(socket.read(buffer) != -1 || buffer.position() > 0) {  // 做完了吗?
  buffer.flip();          // 准备从缓冲区读取字节并写入文件
  file.write(buffer);     // 写入一些或全部字节到文件
  buffer.compact();       // 丢弃已被写入的字节
}
socket.close();           // 关闭 socket 信道
file.close();             // 关闭文件信道
out.close();              // 关闭底层文件
```

另一个创建 SocketChannel 的方法，就是使用无自变量版本的 open()，它会创建未联机的信道。这让你能调用 socket() method 来取得底层的 socket，依所需配置 socket 并以 connect method 联机至所要的主机。例如：

```
SocketChannel sc = SocketChannel.open();  // 开启一个未联机的 socket 信道
Socket s = sc.socket();                   // 取得底层的 java.net.Socket
s.setSoTimeout(3000);                     // 在 3 秒之后超时
// 现在将 socket 信道联机至所要的主机和端口
sc.connect(new InetSocketAddress("www.davidflanagan.com", 80));

ByteBuffer buffer = ByteBuffer.allocate(8192);  // 创建缓冲区
try { sc.read(buffer); }                        // 试着从 socket 读取
```

Download at http://www.pin5i.com/


```
catch(SocketTimeoutException e) {                    // 在这里捕捉超时
  System.out.println("The remote computer is not responding.");
  sc.close();
}
```

除了SocketChannel类之外，java.nio.channels包还定义了用datagram代替socket来进行网络连接的DatagramChannel。

New I/O API的最强大特性之一，就是信道（例如SocketChannel和DatagramChannel）可以被用在非阻塞模式中。我们会在稍后的章节中看到范例。

服务器端网络连接

java.net包针对客户端和服务器端之间的通信定义了Socket，并定义了由服务器使用的ServerSocket来监听并接受来自客户端的连接。java.nio.channels包也类似于此：它对数据转移定义了SocketChannel类，对接受联机定义了ServerSocketChannel类。ServerSocketChannel是个特殊的信道，因为它没有实现ReadableByteChannel或WritableByteChannel。它没有read()和write() method，但具有用于接受客户端连接和取得SocketChannel的accept() method，并通过SockeCthannel与客户端通信。这有段程序代码，是针对简单、单一线程的服务器，它会监听8000端口的联机并回复当前时间给所有联机的客户端：

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

public class DateServer {
    public static void main(String[] args) throws java.io.IOException {
        // 取得CharsetEncoder以编码要送至客户端的文字
        CharsetEncoder encoder = Charset.forName("US-ASCII").newEncoder();

        // 创建一个新的ServerSocketChannel，并将它绑定至8000端口
        // 请注意，这必须使用底层的ServerSocket来完成
        ServerSocketChannel server = ServerSocketChannel.open();
        server.socket().bind(new java.net.InetSocketAddress(8000));

        for(;;) {  // 此服务器会永远运行
            // 等待客户端连接
            SocketChannel client = server.accept();
            // 以字符串形式取得当前日期与时间
            String response = new java.util.Date().toString() + "\r\n";
            // 封装、编码并传送字符串至客户端
            client.write(encoder.encode(CharBuffer.wrap(response)));
            // 与客户端中断连接
            client.close();
        }
    }
}
```

Download at http://www.pin5i.com/

非阻塞式 I/O

前面的 DateServer 类是个简单的网络服务器。因为它没有维护与任何客户端的连接，所以绝对不会需要同时与各个客户端通信，而且使用中的 SocketChannel 绝不会超过一个。然而，实际上服务器必须要能同时与多个客户端通信。java.io和java.net API 只允许阻塞式I/O，所以用这些API编写的服务器都必须对各个客户端使用独立的线程。对于具有许多客户端的大型服务器来说，这个方法并不恰当。为了解决这个问题，New I/O API 允许大部分的信道（但不包含 FileChannel）使用非阻塞式模式，并且允许单一线程管理所有悬置的连接。这是用Selector对象完成的，它会记录一组已注册的信道，并且可以将其冻结直到一个或多个信道已准备好做I/O，如以下程序代码所示。这个范例比本章大部分的范例都长，但它是个完整的运作中的服务器类，它管理了一个 ServerSocketChannel以及任意数量通过单一Selector对象与客户端的SocketChannel 连接。

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;            // 针对 Set 和 Iterator

public class NonBlockingServer {
    public static void main(String[] args) throws IOException {

        //  取得你所需的字符编码器和译码器
        Charset charset = Charset.forName("ISO-8859-1");
        CharsetEncoder encoder = charset.newEncoder();
        CharsetDecoder decoder = charset.newDecoder();

        // 配置缓冲区以供与客户端通信
        ByteBuffer buffer = ByteBuffer.allocate(512);

        // 此程序代码中的所有信道都会是使用非阻塞模式
        // 因此创建一个 Selector 对象，它会在监控所有的
        // 信道时冻结，并且只有在一个或多个信道准备好做
        // I/O 之类的事情时停止冻结
        Selector selector = Selector.open();

        // 创建一个新的 ServerSocketChannel，并将它连接至 8000 端口
        // 请注意，这必须使用底层 ServerSocket 来完成
        ServerSocketChannel server = ServerSocketChannel.open();
        server.socket().bind(new java.net.InetSocketAddress(8000));
        // 让 ServerSocketChannel 成为非阻塞模式
        server.configureBlocking(false);
        // 现在使用 Selector 来注册（但请注意，register()会在信道上被
        // 调用，而不是在 selector 对象上）
        // SelectionKey 代表了此信道与此 Selector 的注册
        SelectionKey serverkey = server.register(selector,
                                                 SelectionKey.OP_ACCEPT);
```

Download at http://www.pin5i.com/


```
for(;;) {   // 主要的服务器循环。服务器会永远运行
    // 此调用会冻结，直到在其中一个已注册的信道
    // 上有活动。这是非阻塞式 I/O 中的关键 method
    selector.select();

    // 取得一个 java.util.Set，它包含了所有已准备好做 I/O 的
    // 信道的 SelectionKey 对象
    Set keys = selector.selectedKeys();

    // 使用 java.util.Iterator 来逐一处理被选定的键
    for(Iterator i = keys.iterator(); i.hasNext(); ) {
        // 取得 set 中的下一个 SelectionKey，并将它从 set 中
        // 移除。它必须被明确地移除，否则就会被下一个对
        // select()的调用返回
        SelectionKey key = (SelectionKey) i.next();
        i.remove();

        // 检查这个键是否是在你注册 ServerSocketChannel 时所
        // 获得的 SelectionKey
        if (key == serverkey) {
            // ServerSocketChannel 上的活动代表有客户端正
            // 尝试要连接至服务器
            if (key.isAcceptable()) {
                // 接受客户端连接并取得一个 SocketChannel 来
                // 与客户端通信
                SocketChannel client = server.accept();
                // 让客户端信道成为非阻塞模式
                client.configureBlocking(false);
                // 现在用 Selector 对象来注册，告诉
                // 它你想知道什么时候会有数据从这个通
                // 道被读出
                SelectionKey clientkey =
                    client.register(selector, SelectionKey.OP_READ);
                // 添加一些客户端状态到键上。当你与客户端交谈时
                // 就会使用到这个状态
                clientkey.attach(new Integer(0));
            }
        }
        else {
            // 如果从键所组成的 Set 中取得的键不是
            // ServerSocketChannel 键，那么它就一定是代
            // 表其中一个客户端连接的键
            // 从键取得信道
            SocketChannel client = (SocketChannel) key.channel();

            // 如果已执行到这，信道中就应该会有数据可读
            // 取，但要再做检查
            if (!key.isReadable()) continue;

            // 现在从客户端读取字节。假设客户端的所有字节
            // 都在一个读取操作中
            int bytesread = client.read(buffer);

            // 如果 read()返回 -1，就是指示流的结尾，这
            // 代表客户端已中断连接，所以要解除已选取键的
```

Download at http://www.pin5i.com/


```
                    // 注册并关闭信道
                    if (bytesread == -1) {
                        key.cancel();
                        client.close();
                        continue;
                    }
                    // 否则，将字节译码为请求字符串
                    buffer.flip();
                    String request = decoder.decode(buffer).toString();
                    buffer.clear();
                    // 现在以请求字符串为基础来回复客户端
                    if (request.trim().equals("quit")) {
                        // 如果请求是“quit”，就送出最后的信息
                        // 关闭信道并解除对 SelectionKey 的注册
                        client.write(encoder.encode(CharBuffer.wrap("Bye.")));
                        key.cancel();
                        client.close();
                    }
                    else {
                        // 否则，发送由此请求的序号加上大写版
                        // 本的请求字符串所组成的响应字符串。请注
                        // 意，记录序号的方法是通过“添加”
                        // 对象到 SelectionKey 并每次递增

                        // 从 SelectionKey 取得序号
                        int num = ((Integer)key.attachment()).intValue();
                        // 针对响应字符串
                        String response = num + ": " +
                            request.toUpperCase();
                        // 封装、编码并写入响应字符串
                        client.write(encoder.encode(CharBuffer.wrap(response)));
                        // 添加递增的序号至键
                        key.attach(new Integer(num+1));
                    }
                }
            }
        }
    }
}
```

非阻塞式I/O对于网络服务器写入最有用，它对于同时有多个网络连接悬置的客户端也很有用。例如，考虑web浏览器下载网页以及那个页面同时引用到的图像。非阻塞式I/O的另一个有趣用法就是执行非阻塞式socket连接操作。这个概念就是你可以要求SocketChannel建立连接至远程主机，然后在底层OS设定跨网络的连接时做其他事(例如建立GUI)。之后，如果连接还没建立好，你就进行select()调用以冻结，直到连接已被建立。针对非阻塞式连接的程序代码看起来就像这样：

```
// 创建新的、未联机的 SocketChannel。把它设定为非阻
// 塞模式，使用新的 Selector 加以注册，然后告诉它进行
// 连接。connect 调用会返回，而不是等待网络连接被完
// 全建立
```

Download at http://www.pin5i.com/


```
Selector selector = Selector.open();
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
channel.register(selector, SelectionKey.OP_CONNECT);
channel.connect(new InetSocketAddress(hostname, port));

// 现在在连接建立期间做其他的事
// 例如，在这建立GUI

// 如果现在有需要就冻结，直到SocketChannel已准备好联机
// 由于你已用这个selector注册了唯一的信道，所以不必查看键set，
// 你知道哪个信道已准备好
while(selector.select() == 0) /* 空循环 */;

// 此调用对于完成非阻塞式连接是必需的
channel.finishConnect();

// 最后，关闭selector，这会解除信道的注册
selector.close();
```

XML

Java 1.4 和 Java 5.0 已为 Java 平台增加了强大的 XML 处理特性：

org.xml.sax

此包以及它的两个子包定义了实际标准的 SAX API（SAX 代表 Simple API for XML）。SAX 是个事件驱动、XML 解析的 API：在解析 XML 文件时，SAX 解析器会调用指定 ContentHandler 对象（以及其他相关的处理程序对象）的 method。文件的结构和内容会由 method 调用来完整描述。要存储任何的状态或执行任何适当的动作是取决于 ContentHandler 实现。此包包括了针对 SAX 2 API 的类以及针对 SAX 1 的过时类。

org.w3c.dom

此包定义了一些 interface，以树形结构来表现 XML 文件。Document Object Model（DOM）是 World Wide Web Consortium（W3C）的建议（实质上是个标准）。DOM 解析器会读取 XML 文件并将它转换成由节点组成的树，那代表了文件的完整内容。一旦文件的树形结构被建立，程序就可以随意查看并操作它。Java 1.4 包含了 Level 2 DOM 的核心模块，而 Java 5.0 包含了 Level 3 DOM 的核心、事件以及加载 / 存储模块。

javax.xml.parsers

此包提供了用于实例化 SAX 和 DOM 解析器以解析 XML 文件的高层次 interface。

Download at http://www.pin5i.com/

javax.xml.transform

此包以及它的子包定义了使用 XSLT 标准转换 XML 文件内容和表达方法的 Java API。

javax.xml.validation

这个 Java 5.0 包对验证模式的 XML 文件提供了支持。实现必须支持 W3C XML Schema 标准，而且也要支持其他的模式类型。

javax.xml.xpath

此包也是 Java 5.0 中新出现的，支持 XPath 的判定以选择 XML 文件中的节点。

使用这些包的范例会出现在下面的章节中。

使用 SAX 解析 XML

使用SAX解析XML文件的第一步就是取得SAX解析器。如果你有自己的SAX解析器实现，就只要实例化适当的解析器类即可。但是，通常较简单的方式是使用javax.xml.parsers包来实例化由Java实现所提供的SAX解析器。程序代码看起来就像这样：

```
import javax.xml.parsers.*;

// 取得用来创建 SAX 解析器的 factory 对象
SAXParserFactory parserFactory = SAXParserFactory.newInstance();

// 设定 factory 对象来指定它所创建的解析器的属性
parserFactory.setValidating(true);
parserFactory.setNamespaceAware(true);

// 现在创建一个 SAXParser 对象
SAXParser parser = parserFactory.newSAXParser();    // 可能会抛出异常
```

SAXParser是个围绕于org.xml.sax.XMLReader类的简单封装程序。一旦你取得它，就如前面的程序代码，只要调用各种parse() method就可以解析文件了。其中有一些method使用了过时的SAX 1 HandlerBase类，而其他的使用了当前的SAX 2 org.xml.sax.helpers.DefaultHandler类。DefaultHandler类提供了所有ContentHandler、ErrorHandler、DTDHandler以及EntityResolver interface的空实现，这些都是SAX解析器在解析XML文件时可以调用的method。通过使用DefaultHandler的子类并定义你自己关心的method，就可以执行任何必需的动作来响应由解析器产生的method调用。以下的程序代码显示了使用SAX来解析XML文件的method，并判定出现在文件中的XML元素数量以及出现在那些元素中的一般文字的字符数量（也许排除了“可忽略的空白”）：

Download at http://www.pin5i.com/


```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SAXCount {
    public static void main(String[] args)
        throws SAXException,IOException, ParserConfigurationException
    {
        // 创建一个解析器 factory 并用它来产生解析器
        SAXParserFactory parserFactory = SAXParserFactory.newInstance();
        SAXParser parser = parserFactory.newSAXParser();
        // 这是你所要解析的文件的名称
        String filename = args[0];
        // 实例化 DefaultHandler 子类来进行计算
        CountHandler handler = new CountHandler();
        // 启动解析器。它会读取文件并调用处理程序的 method
        parser.parse(new File(filename), handler);
        // 当完成时，报告由处理程序对象所存储的结果
        System.out.println(filename + " contains " + handler.numElements +
                           " elements and " + handler.numChars +
                           " other characters ");
    }

    // 这个内部类扩展了 DefaultHandler 来计算 XML 文件
    // 中的元素和文字数量，并把结果存储在公共字段中。
    // 你还可以覆盖很多其他的 DefaultHandler method，但你
    // 只需要这些
    public static class CountHandler extends DefaultHandler {
        public int numElements = 0, numChars = 0;  // 在这储存计数
        // 此 method 会在解析器遇到任何 XML 元素的开始标记时
        // 被调用。忽略这些自变量但计算元素
        public void startElement(String uri, String localname, String qname,
                                 Attributes attributes) {
            numElements++;
        }

        // 此 method 会在遇到元素中的任意一个一般文字时被调用
        // 仅只计算那个文字中的字符数量
        public void characters(char[] text, int start, int length) {
            numChars += length;
        }
    }
}
```

使用 DOM 解析 XML

DOM API 与 SAX API 差异极大。虽然 SAX 在扫描 XML 文件时是很有效率的方式，但它不太适合想要修改文件的程序。DOM解析器不是把XML文件转换为一连串的method调用，而是把文件转换为 org.w3c.dom.Document 对象，那是由 org.w3c.dom.Node

Download at http://www.pin5i.com/

对象组成的树。将全部XML文件转换为树形结构后能提供对整个文件的的随机访问，但会消耗大量的内存。

在DOM API中，文件树中的每个节点实现了Node interface和类型特有的subinterface（DOM文件中最常见的节点类型是Element和Text节点）。当解析器对文件的解析完成时，你的程序就可以使用Node及其subinterface的各种method来查看及操作那个树。以下程序代码使用JAXP来取得DOM解析器（在JAXP的定义中，它被称为DocumentBuilder）。它接下来会解析XML文件并从中建立一个文件树。然后，它查看文件树来搜索<sect1>元素并输出每个<title>的内容。

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class GetSectionTitles {
    public static void main(String[] args)
        throws IOException, ParserConfigurationException,
               org.xml.sax.SAXException
    {
        // 创建一个 factory 对象以供创建 DOM 解析器并加以设定
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setIgnoringComments(true);  // 我们想忽略注释
        factory.setCoalescing(true);        // 将 CDATA 转换为 Text 节点
        factory.setNamespaceAware(false);   // 无命名空间：这是默认的
        factory.setValidating(false);       // 不验证 DTD：这也是默认的

        // 现在使用 factory 来创建 DOM 解析器，也就是 DocumentBuilder
        DocumentBuilder parser = factory.newDocumentBuilder();

        // 解析文件并建立文件树来代表其内容
        Document document = parser.parse(new File(args[0]));

        // 查询文件所包含的所有 <sect1> 元素的列表
        NodeList sections = document.getElementsByTagName("sect1");
        // 逐一处理那些 <sect1> 元素，一次处理一个
        int numSections = sections.getLength();
        for(int i = 0; i < numSections; i++) {
            Element section = (Element)sections.item(i);  // 一个 <sect1>
            // 每个 <sect1> 的第一个子元素应该是 <title> 元
            // 素，但可能会先有一些空白 Text 节点，所
            // 以逐一处理直到找到第一个子元素
            Node title = section.getFirstChild();
            while(title != null && title.getNodeType() != Node.ELEMENT_NODE)
                title = title.getNextSibling();
            // 列出此元素的 Text 节点所包含的文字
            if (title != null)
                System.out.println(title.getFirstChild().getNodeValue());
        }
    }
}
```

Download at http://www.pin5i.com/

转换 XML 文件

javax.xml.transform包定义了用来创建Transformer对象的TransformerFactory类。Transformer可以将文件从Source表达方式转换为Result表达方式，并在处理过程中选择性地对文件内容应用XSLT转换。有三个子包定义了Source和Result interface的具体实现，让文件能在这三种表达方式之间转换：

javax.xml.transform.stream

将文件当成由 XML 文本所组成的流来表达。

javax.xml.transform.dom

将文件当成 DOM Document 树来表达。

javax.xml.transform.sax

将文件当作一连串的 SAX method 调用来表达。

以下程序代码说明这些包将文件的表达方式从DOM Document树转换为XML文本所组成的流的方法。这段程序代码有个有趣的特性就是，它没有通过解析文件来建立Document 树，而是从头开始建立。

```
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DOMToStream {
    public static void main(String[] args)
        throws ParserConfigurationException,
               TransformerConfigurationException,
               TransformerException
    {
        // 创建 DocumentBuilderFactory 和 DocumentBuilder
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        // 然而不解析 XML 文件，只创建可以由你自己增加内容的空文件
        Document document = db.newDocument();

        // 现在使用 DOM method 来建立文件树
        Element book = document.createElement("book"); // 创建新元素
        book.setAttribute("id", "javanut4");           // 给它一个属性
        document.appendChild(book);                    // 放入文件
        for(int i = 1; i <= 3; i++) {                  // 增加更多元素
            Element chapter = document.createElement("chapter");
            Element title = document.createElement("title");
            title.appendChild(document.createTextNode("Chapter " + i));
            chapter.appendChild(title);
```

Download at http://www.pin5i.com/


```
        chapter.appendChild(document.createElement("para"));
        book.appendChild(chapter);
    }

    // 现在创建一个 TransformerFactory 并用它来创建 Transformer 对象，把
    // 我们的 DOM 文件转换为由 XML 文本组成的流
    //  newTransformer()没有自变量意味着没有 XSLT 样式表
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // 创建用于转换的 Source 和 Result 对象
    DOMSource source = new DOMSource(document);          // DOM 文件
    StreamResult result = new StreamResult(System.out);  // 成为 XML 文本

    // 最后，进行转换
    transformer.transform(source, result);
  }
}
```

javax.xml.transform最有趣的用法牵涉到XSLT样式表。XSLT是个复杂但强大的XML文法，它描述了XML文件内容被转换为另一种形式（例如，XML、HTML或一般文字）的方法。XSLT样式表的教学超出了本书的范围，但以下程序代码（只包含6个关键行）说明了将这样的样式表（它本身是个XML文件）应用至另一个XML文件，并将最后产生的文件写入流的方法：

```
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Transform {
    public static void main(String[] args)
        throws TransformerConfigurationException,
               TransformerException
    {
        // 取得针对输入、样式表以及输出的 Source 和 Result 对象
        StreamSource input = new StreamSource(new File(args[0]));
        StreamSource stylesheet = new StreamSource(new File(args[1]));
        StreamResult output = new StreamResult(new File(args[2]));

        // 创建一个 transformer 并执行转换
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer(stylesheet);
        transformer.transform(input, output);
    }
}
```

Download at http://www.pin5i.com/

验证 XML 文件

javax.xml.validation 包能让你针对模式验证 XML 文件。从 javax.xml.parsers 包取得的 SAX 和 DOM 解析器可以在解析过程中对 DTD 执行验证，但此包将验证与解析分开，并且也对任意的模式类型提供了支持。所有的实现都必须支持 W3C XML Schema 并且允许支持其他的模式类型，例如 RELAX NG。

如果使用包，则要以 SchemaFactory instance 开始 —— 那是针对指定模式类型的解析器，使用此解析器将模式文件解析为 Schema 对象。从 Schema 取得 Validator，然后使用 Validator 来验证你的 XML 文件。文件被指定为 SAXSource 或 DOMSource 对象。你可以从 javax.xml.transform 取回这些类。

如果文件是有效的，Validator 对象的 validate() method 就会正常返回；如果文件不是有效的，validate() 就会抛出 SAXException。你可以安装针对 Validator 的 org.xml.sax.ErrorHandler 对象来对会造成异常的各种验证错误提供一些控制。

```
import javax.xml.XMLConstants;
import javax.xml.validation.*;
import javax.xml.transform.sax.SAXSource;
import org.xml.sax.*;
import java.io.*;

public class Validate {
    public static void main(String[] args) throws IOException {
        File documentFile = new File(args[0]); // 第一个自变量是文件
        File schemaFile = new File(args[1]);    // 第二个自变量是模式

        // 取得解析器来解析 W3C 模式。请注意 javax.xml 包的用法
        // 此包只包含了一个由常量组成的类
        SchemaFactory factory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);

        // 现在解析模式文件来创建 Schema 对象
        Schema schema = null;
        try { schema = factory.newSchema(schemaFile); }
        catch(SAXException e) { fail(e); }

        // 从 Schema 取得 Validator 对象
        Validator validator = schema.newValidator();

        // 取得针对文件的 SAXSource 对象
        // 我们也可以在这儿使用 DOMSource
        SAXSource source =
            new SAXSource(new InputSource(new FileReader(documentFile)));

        // 现在验证文件
        try { validator.validate(source); }
        catch(SAXException e) { fail(e); }
```

Download at http://www.pin5i.com/


```
        System.err.println("Document is valid");
    }

    static void fail(SAXException e) {
        if (e instanceof SAXParseException) {
            SAXParseException spe = (SAXParseException) e;
            System.err.printf("%s:%d:%d: %s%n",
                              spe.getSystemId(), spe.getLineNumber(),
                              spe.getColumnNumber(), spe.getMessage());
        }
        else {
            System.err.println(e.getMessage());
        }
        System.exit(1);
    }
}
```

求 XPath 表达式的值

XPath 是用来引用 XML 文件中的特定节点的语言。例如，XPath 表达式“//section/title/text()”会引用文件中的任意深度的 <section> 元素中的 <title> 元素内的文字。关于 XPath 语言的完整说明超出了本书的范围。Java 5.0 中新出现的 javax.xml.xpath 包提供了一个方法来找出文件中与 XPath 表达式相匹配的所有节点。

```
import javax.xml.xpath.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class XPathEvaluator {
    public static void main(String[] args)
        throws ParserConfigurationException, XPathExpressionException,
               org.xml.sax.SAXException, java.io.IOException
    {
        String documentName = args[0];
        String expression = args[1];

        // 将文件解析为 DOM 树
        // XPATH 也可以与 SAXInputSource 一起使用
        DocumentBuilder parser =
            DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc = parser.parse(new java.io.File(documentName));

        // 取得 XPath 对象来求表达式的值
        XPath xpath = XPathFactory.newInstance().newXPath();

        System.out.println(xpath.evaluate(expression, doc));

        // 或求表达式的值以取得所有和节点相匹配的 DOM NodeList，然后
        // 逐一处理各个节点直至最后
        NodeList nodes = (NodeList)xpath.evaluate(expression, doc,
                                                  XPathConstants.NODESET);
```

Download at http://www.pin5i.com/


```
            for(int i = 0, n = nodes.getLength(); i < n; i++) {
                Node node = nodes.item(i);
                System.out.println(node);
            }
        }
    }
```

类型、反射与动态加载

java.lang.Class类表达了Java中的数据类型还有java.lang.reflect包中的类，它给Java程序提供了自我评测（或自反射）的能力。Java类可以考虑它自己或任何其他的类，并判定它的超类及它定义了哪些method等。

Class对象

有好几种方法可以在Java中取得Class对象：

```
// 取得任意对象o的Class
Class c = o.getClass();

// 以各种预先定义的常量取得基本类型的Class对象
c = Void.TYPE;          // 特殊的“无返回值”类型
c = Byte.TYPE;          // 代表字节的Class对象
c = Integer.TYPE;       // 代表整数的Class对象
c = Double.TYPE;        // 诸如此类。类似的还有Short、Character、Long、Float

// 将类直接量表达为类型名称后面接着“.class”
c = int.class;          // 与Integer.TYPE相同
c = String.class;       // 与"dummystring".getClass()相同
c = byte[].class;       // 字节数组的类型
c = Class[][].class;    // 由Class对象组成的数组所组成的数组类型
```

Class上的反射（Reflection）

一旦有了Class对象，你就可以用它执行一些有趣的反射操作：

```
import java.lang.reflect.*;

Object o;                       // 某个要研究的未知对象
Class c = o.getClass();         // 取得其类型

// 如果它是数组，就找出它的基类型
while (c.isArray()) c = c.getComponentType();

// 如果c不是基本类型，就输出它的类层次
if (!c.isPrimitive()) {
  for(Class s = c; s != null; s = s.getSuperclass())
    System.out.println(s.getName() + " extends");
}
```

Download at http://www.pin5i.com/


```
// 尝试创建属于 c 的新实例，这会需要无自变量的构造函数
Object newobj = null;
try { newobj = c.newInstance(); }
catch (Exception e) {
  // 处理 InstantiationException、IllegalAccessException
}

// 查看类是否有个名称为 setText 的 method，它会接收一个 String
// 如果有，就用字符串自变量来调用它
try {
    Method m = c.getMethod("setText", new Class[] { String.class });
    m.invoke(newobj, new Object[] { "My Label" });
} catch(Exception e) { /* 在这处理异常 */ }

// 这些是 Java 5.0 中的 varargs method，所以语法会更为清楚
// 寻找并调用名称为“put”的 method，它会接收两个 Object 自变量
try {
    Method m = c.getMethod("add", Object.class, Object.class);
    m.invoke(newobj, "key", "value");
} catch(Exception e) { System.out.println(e); }

// 在 Java 5.0 中，我们可以在枚举类型和常量上使用反射 (reflection)
Class<Thread.State> ts = Thread.State.class;  // Thread.State 类型
if (ts.isEnum()) {  // 如果它是枚举类型
    Thread.State[] constants = ts.getEnumConstants(); // 取得它的常量
}
try {
    Field f = ts.getField("RUNNABLE");          // 取得名称为“RUNNABLE”的字段
    System.out.println(f.isEnumConstant());  // 它是个列举常量吗?
}
catch(Exception e) { System.out.println(e); }

// JVM 会在运行时丢弃泛型信息，但它会由于编译器而被存储在类文件
// 中，并且可以通过反射来访问
try {
    Class map = Class.forName("java.util.Map");

    TypeVariable<?>[] typevars = map.getTypeParameters();
    for(TypeVariable<?> typevar : typevars) {
        System.out.print(typevar.getName());
        Type[] bounds = typevar.getBounds();
        if (bounds.length > 0) System.out.print(" extends ");
        for(int i = 0; i < bounds.length; i++) {
            if (i > 0) System.out.print(" & ");
            System.out.print(bounds[i]);
        }
        System.out.println();
    }
}
catch(Exception e) { System.out.println(e); }

// 在 Java 5.0 中，反射也可被用在 annotation 类型上，并决定
// 运行时可见注释的值
```

Download at http://www.pin5i.com/


```
Class<?> a = Override.class;  // annotation 类
if (a.isAnnotation()) {       // 这是个 annotation 类型吗?
    // 寻找一些 meta-annotation
    java.lang.annotation.Retention retention =
        a.getAnnotation(java.lang.annotation.Retention.class);
    if (retention != null)
        System.out.printf("Retention: %s%n", retention.value());
}
```

动态类加载

Java 中的 Class 也为动态类加载提供了简单的机制。但是，对于在动态类加载上的更加完全的控制，你应该使用 java.lang.ClassLoader 对象，通常是 java.net.URLClassLoader。这个技术很有用，例如，当你想加载的类是被命名在配置文件中而不是被编写在程序中的时候。

```
// 动态地加载名称指定于配置文件中的类
String classname =                          // 查询类的名称
    config.getProperty("filterclass",       // 属性名称
                       "com.davidflanagan.filters.Default"); // 默认值

try {
  Class c = Class.forName(classname);   // 动态地加载类
  Object o = c.newInstance();           // 动态地将它实例化
} catch (Exception e) { /* 处理异常 */ }
```

前面的程序代码只有在被加载的类在类路径中才行得通。如果情况并非如此，你可以创建自定义的 ClassLoader 对象从你自己指定的路径（或 URL）加载类：

```
import java.net.*;
String classdir = config.getProperty("filterDirectory"); // 查询类路径
try {
  ClassLoader loader = new URLClassLoader(new URL[] { new URL(classdir) });
  Class c = loader.loadClass(classname);
}
catch (Exception e) { /* 处理异常 */ }
```

动态代理服务器

java.lang.reflect 包的 Porxy 类和 InvocationHandler interface 是在 Java 1.3 中增加的。Proxy 是个强大但不常使用的类，它能让你动态地创建实现一个特定 interface 或一组 interface 的新类或实例。它也分派了对 InvocationHandler 对象的 interface method 调用。

Download at http://www.pin5i.com/

对象持久性保存

Java 平台为对象持久保存（persistence）提供了两个机制：存储对象状态的能力，以便对象可在稍后被重建。这两个机制都牵涉到序列化，其中第二个是特别针对 JavaBean 的。

序列化

java.io 包的最重要特性之一就是将对象序列化：把对象转换成字节流，以在稍后反序列化为原始对象的副本。以下程序代码显示如何使用序列化把对象存储至文件并在稍后将它读回：

```
Object o;   // 我们要序列化的对象；它必须实现Serializable
File f;     // 用来存储数据的文件

try {
  // 将对象序列化
  ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(f));
  oos.writeObject(o);
  oos.close();

  // 将对象读回
  ObjectInputStream ois = new ObjectInputStream(new FileInputStream(f));
  Object copy = ois.readObject();
  ois.close();
}
catch (IOException e) { /* 处理输入/输出异常 */ }
catch (ClassNotFoundException cnfe) { /* readObject() 可能会抛出这个 */ }
```

先前的范例序列化至文件，但请记住，你可以将已序列化对象写入任一种流类型。因此，你可以把对象写入字节数组，然后把它从字节数组读回，创建对象的副本。你可以把对象的字节写入压缩流甚至将字节跨网络连接发送至另一个程序的流！

JavaBeans 持久保存

Java 1.4 引入了序列化机制是为了要配合 JavaBeans 组件使用。java.io 序列化是通过存储对象的内部字段的状态来完成。另一方面，java.beans 持久保存是通过将 bean 的状态存储为对定义于类内的公共 method 的一连串调用。由于它是以公共 API 为基础，而不是内部状态，所以 JavaBeans 持久保存机制允许相同 API 不同实现之间的互通，以更完善的方式来处理版本差异，并且适合已序列化对象的长期存储。

bean 以及任一个 descendant bean 或其他被 java.beans.XMLEncoder 序列化的对象，可以用 java.beans.XMLDecoder 来反序列化。这些类会对指定的流做写入和读取，但它们本身并不是流类。这里是可用来对 bean 编码的方式：

Download at http://www.pin5i.com/


```
// 创建 JavaBean 并设定它的一些属性
javax.swing.JFrame bean = new javax.swing.JFrame("PersistBean");
bean.setSize(300, 300);
// 现在将它已编码的格式存储在文件 bean.xml 中
BufferedOutputStream out =                    // 创建输出流
    new BufferedOutputStream(new FileOutputStream("bean.xml"));
XMLEncoder encoder = new XMLEncoder(out);     // 创建流的编码器
encoder.writeObject(bean);                    // 对 bean 编码
encoder.close();                              // 关闭编码器和流
```

这里是相对应的程序代码，用来将 bean 从其已序列化的形式中解码：

```
BufferedInputStream in =                      // 创建输出流
    new BufferedInputStream(new FileInputStream("bean.xml"));
XMLDecoder decoder = new XMLDecoder(in);      // 创建流的译码器
Object b = decoder.readObject();              // 对 bean 解码
decoder.close();                              // 关闭译码器和流
bean = (javax.swing.JFrame) b;                // 将 bean 转换为适当的类型
bean.setVisible(true);                        // 开始使用它
```

安全性

java.security 包定义了一些与 Java 访问控制结构有关的类，这会在第六章做更详细的讨论。这些类能让Java程序在受限制的环境中运行未被信任的程序代码，在那个环境中，程序代码不会造成什么损害。虽然这些是重要的类，但你很少需要使用它们。较有趣的类是用于消息摘要和数字签名的那些类，在以下章节会对它们作示范。

消息摘要

消息摘要(message digest)是个值，也就是所谓的加密校验和(cryptographic checksum)或安全散列（secure hash），那是根据一连串的字节计算得到的。摘要的长度通常会远小于计算时所依据的数据的长度。当传送数据（消息）时，你可以把消息摘要一起传送出去。消息的接收者接着就可以重新计算已收到数据上的消息摘要，并通过比较计算得到的摘要与收到的摘要来判定消息或摘要是否在传输过程中损毁或被篡改。我们在本章的前面讨论流时，已看到了一个计算消息摘要的方法。类似的技术也可以用来计算非流形式的二进制数据的消息摘要：

```
import java.security.*;

// 取得一个对象并使用“安全散列算法 (Secure Hash Algorithm)”来计算消息摘要
// 这个 method 可能会抛出 NoSuchAlgorithmException
MessageDigest md = MessageDigest.getInstance("SHA");

byte[] data, data1, data2, secret;  // 一些已在其他地方被初始化的字节数组
```

Download at http://www.pin5i.com/


```
// 创建针对单一字节数组的摘要
byte[] digest = md.digest(data);

// 创建针对几大块数据的摘要
md.reset();                 // 选择性的：会由digest()自动调用
md.update(data1);           // 处理第一块数据
md.update(data2);           // 处理第二块数据
digest = md.digest();       // 计算摘要

// 创建加密摘要（keyed digest），如果你知道加密字节（secret byte）就能做验证
md.update(data);                // 要与摘要一起传送的数据
digest = md.digest(secret);  // 加入加密字节并计算摘要

// 用像这样的方式验证摘要
byte[] receivedData, receivedDigest;  // 我们所收到的数据和摘要
byte[] verifyDigest = md.digest(receivedData);  // 对已收到的数据做摘要
// 比较计算得来的摘要和收到的摘要
boolean verified =  java.util.Arrays.equals(receivedDigest, verifyDigest);
```

数字签名

数字签名（digital signature）结合了消息摘要算法和公共密钥密码学（public-key cryptography）。消息的传送者Alice可以计算消息的摘要，然后用她的私用密钥（private key）来加密摘要。接着她把消息和已加密的摘要传送给接收者Bob。Bob知道Alice的公共密钥（毕竟那是公开的），所以他可以用它来解密摘要并验证那个消息没有被篡改。在执行验证时，Bob也知道摘要是用Alice的私用密钥加密的，因为他能成功地用Alice的公共密钥来对摘要解密。由于Alice是唯一知道自己的私用密钥的人，所以信息一定是来自Alice。数字签名就是因此得名，就像笔和纸的签名一样，它可以验证文件或信息的来源。但是，与笔和纸的签名所不同的是，要伪造数字签名虽不是不可能，但难度非常高，这不仅只是剪贴到另一份文件而已。

Java使得创建数字签名变得很容易。但是，为了要创建数字签名，你需要有java.security.PrivateKey对象。假设你的系统中已有密钥库（keystore）（请参阅第八章的keytool说明文件），就可以用如下的程序代码来取得一个密钥：

```
// 这里是我们所需要的一些基本数据
File homedir = new File(System.getProperty("user.home"));
File keyfile = new File(homedir, ".keystore");  // 或从配置文件读取
String filepass = "KeyStore password"           // 用于整个文件的密码
String signer = "david";                        // 从配置文件读取
String password = "No one can guess this!";     // 最好对此做提示
PrivateKey key;  // 这是我们想在密钥库中寻找的密钥

try {
  // 取得KeyStore对象，然后将数据放入
  KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
  keystore.load(new BufferedInputStream(new FileInputStream(keyfile)),
                filepass.toCharArray());
```

Download at http://www.pin5i.com/


```
    // 现在得到了所需的密钥
    key = (PrivateKey) keystore.getKey(signer, password.toCharArray());
  }
  catch (Exception e) { /* 在这处理各种异常 */ }
```

一旦有了PrivateKey对象，就可以用java.security.Signature对象来创建数字签名：

```
PrivateKey key;                  // 就如前面所显示的，已初始化
byte[] data;                     // 要被签名的数据
Signature s =                    // 取得对象来创建并验证签名
  Signature.getInstance("SHA1withDSA");  // 可能会抛出
                                         // NoSuchAlgorithmException
s.initSign(key);                 // 将它初始化，可能会抛出InvalidKeyException
s.update(data);                  // 要签名的数据，可能会抛出SignatureException
/* s.update(data2); */           // 调用多次以声明所有的数据
byte[] signature = s.sign();     // 计算签名
```

Signature对象可以验证数字签名：

```
byte[] data;          // 加上签名的数据，已在其他地方初始化
byte[] signature;     // 要被验证的签名，已在其他地方初始化
String signername;    // 创建签名的人，已在其他地方初始化
KeyStore keystore;    // 存储认证的地方，就如前面所显示的，已初始化

// 寻找此签名者的公共密钥认证
java.security.cert.Certificate cert = keystore.getCertificate(signername);
PublicKey publickey = cert.getPublicKey();  // 从这取得公共密钥

Signature s = Signature.getInstance("SHA1withDSA"); // 或者某种其他的算法
s.initVerify(publickey);                            // 用于验证的设定
s.update(data);                                     // 说明已签名的数据
boolean verified = s.verify(signature);             // 验证签名数据
```

已签名对象

java.security.SignedObject类是个有用的公共程序，用于将数字签名与对象封装在一起。SignedObject接着就可以被序列化并传送至接收者，接收者可以将它反序列化并使用verify() method来验证签名：

```
Serializable o;  // 要被加上签名的对象，必须是Serialable
PrivateKey k;    // 用来签名的密钥，已在其他地方初始化
Signature s = Signature.getInstance("SHA1withDSA"); // 签名“引擎”
SignedObject so = new SignedObject(o, k, s);        // 创建SignedObject

// SignedObject封装了对象o。现在它可以被序列化
// 并传送至接收者

// 这里是接收者验证SignedObject的方法
SignedObject so;            // 要被反序列化的SignedObject
Object o;                   // 要从so中提取出来的原始对象
PublicKey pk;               // 用来验证的密钥
```

Download at http://www.pin5i.com/


```
Signature s = Signature.getInstance("SHA1withDSA"); // 验证“引擎”
if (so.verify(pk,s))          // 如果签名是有效的，
  o = so.getObject();         // 就提取被封装的对象
```

密码术

java.security包包括了基于密码术的类，但它不包含用于实际加密和解密的类，那是javax.crypto包的工作。此包支持对称密钥密码术，在这种密码术中，加密和解密都是使用相同的密钥，而且加密数据的传送者和接收者都必须知道这个密钥。

加密密钥（secret key）

SecretKey interface 代表了加密密钥，任何加密操作的第一步都是取得适当的SecretKey。很不幸的是，JDK 所提供的 *keytool* 程序无法产生并存储加密密钥，所以程序必须自己处理这些工作。这里有一些程序代码，显示了各种使用SecretKey对象的方法：

```
import javax.crypto.*;
import javax.crypto.spec.*;
// 使用 KeyGenerator 对象产生加密密钥

KeyGenerator desGen = KeyGenerator.getInstance("DES");          // DES 算法
SecretKey desKey = desGen.generateKey();                        // 产生密钥
KeyGenerator desEdeGen = KeyGenerator.getInstance("DESede");    // Triple DES
SecretKey desEdeKey = desEdeGen.generateKey();                  // 产生密钥

// SecretKey 是不透明的密钥表示方式。使用 SecretKeyFactory 来转
// 换成具有透明性、操作性的表达方式：存储在文件中，安全地
// 传送至接收端等
SecretKeyFactory desFactory = SecretKeyFactory.getInstance("DES");
DESKeySpec desSpec = (DESKeySpec)
  desFactory.getKeySpec(desKey, javax.crypto.spec.DESKeySpec.class);
byte[] rawDesKey = desSpec.getKey();
// 对 DESede 密钥做相同的事
SecretKeyFactory desEdeFactory = SecretKeyFactory.getInstance("DESede");
DESedeKeySpec desEdeSpec = (DESedeKeySpec)
  desEdeFactory.getKeySpec(desEdeKey, javax.crypto.spec.DESedeKeySpec.class);
byte[] rawDesEdeKey = desEdeSpec.getKey();

// 将密钥的原始字节转换成 SecretKey 对象
DESedeKeySpec keyspec = new DESedeKeySpec(rawDesEdeKey);
SecretKey k = desEdeFactory.generateSecret(keyspec);

// 对于 DES 和 DESede 密钥，有更简单的密钥创建方法
// SecretKeySpec 实现了 SecretKey，所以用它来代表这些密钥
byte[] desKeyData = new byte[8];          // 从文件读取 8 字节的数据
byte[] tripleDesKeyData = new byte[24]; // 从文件读取 24 字节的数据
SecretKey myDesKey = new SecretKeySpec(desKeyData, "DES");
SecretKey myTripleDesKey = new SecretKeySpec(tripleDesKeyData, "DESede");
```

Download at http://www.pin5i.com/

使用密码进行加密与解密

一旦你取得适当的SecretKey对象，用于加密和解密的主要类就是Cipher。用法如下：

```
SecretKey key;      // 用先前所示的方法取得 SecretKey
byte[] plaintext;   // 要加密的数据，已在其他地方初始化

// 得到对象来执行加密或解密
Cipher cipher = Cipher.getInstance("DESede");  // Triple-DES 加密
// 初始化cipher 对象以供加密
cipher.init(Cipher.ENCRYPT_MODE, key);
// 现在加密数据
byte[] ciphertext = cipher.doFinal(plaintext);

// 如果我们有多块数据要加密，可以这么做
cipher.update(message1);
cipher.update(message2);
byte[] ciphertext = cipher.doFinal();

// 只要倒过来解密
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] decryptedMessage = cipher.doFinal(ciphertext);

// 解密多块数据
byte[] decrypted1 = cipher.update(ciphertext1);
byte[] decrypted2 = cipher.update(ciphertext2);
byte[] decrypted3 = cipher.doFinal(ciphertext3);
```

对流加密与解密

Cipher 类也可以配合 CipherInputStream 或 CipherOutputStrea 使用，在读取与写入流数据时进行加密或解密：

```
byte[] data;                                  // 要加密的数据
SecretKey key;                                // 以先前所示的方式初始化
Cipher c = Cipher.getInstance("DESede");      // 要执行加密的对象
c.init(Cipher.ENCRYPT_MODE, key);             // 把它初始化

// 创建一个流来把字节写到文件
FileOutputStream fos = new FileOutputStream("encrypted.data");

// 创建一个流，在把字节送到那个流之前先加密
// 如果要在读取字节期间加密或解密，请参阅 CipherInputStream
CipherOutputStream cos = new CipherOutputStream(fos, c);

cos.write(data);                              // 将数据加密并写入文件
cos.close();                                  // 一定要记得关闭流
java.util.Arrays.fill(data, (byte)0);         // 清除未被加密的数据
```

Download at http://www.pin5i.com/

已加密对象

最后，javax.crypto.SealedObject 类提供了特别简单的方式来执行加密。此类会序列化指定的对象，并加密最后产生的字节流。SealedObject 本身接着可以被序列化并传送给接收者。接收者只有在知道必要的 SecretKey 时才能获取原始对象：

```
Serializable o;              // 要被加密的对象，必须是Serializable
SecretKey key;                                  // 要用来加密的密钥
Cipher c = Cipher.getInstance("Blowfish");   // 用来执行加密的对象
c.init(Cipher.ENCRYPT_MODE, key);            // 使用密钥来初始化它
SealedObject so = new SealedObject(o, c);    // 创建已被封装的对象

// 原始对象 o 经过加密后封装在对象 so 中，
// 现在它可以被序列化并传送
// 这里是接收者解密原始对象的方法
Object original = so.getObject(key);         // 必须使用相同的SecretKey
```

各式各样的平台特色

以下的章节详述了 Java 平台重要但繁杂的特色，包括了特性（property）、参数选择（preferences）、进程（process）以及管理（management）和安装（instrumentation）。

特性

java.util.Properties 是 java.util.Hashtable 的子类，是原有的 collection 类，比 Java 1.2 中引入的 Collections API 还早。Properties 对象会管理字符串键和字符串值之间的映射并定义了一些method，能把这些映射写入纯文本文件或（在Java 5.0中）XML 文件并读取。这让 Properties 类非常适合用于配置以及用户参数选择文件。Properties 类也被用于由 System.getProperty 返回的系统特性：

```
import java.util.*;
import java.io.*;

// 注意：如果从未被信任的程序代码（例如 applets）中进行调用，则这些系统特
// 性调用中有很多会抛出安全性异常
String homedir = System.getProperty("user.home"); // 取得一个系统特性
Properties sysprops = System.getProperties();     // 取得所有系统特性

// 列出所有已定义的系统特性的名称
for(Enumeration e = sysprops.propertyNames(); e.hasMoreElements();)
  System.out.println(e.nextElement());

sysprops.list(System.out); // 这有个更简单的输出特性方法
// 从配置文件读取特性
Properties options = new Properties();          // 清空特性列表
File configfile = new File(homedir, ".config");  // 配置文件
```

Download at http://www.pin5i.com/


```
try {
  options.load(new FileInputStream(configfile)); // 从文件载入特性
} catch (IOException e) { /* 在这处理异常 */ }

// 查询特性 ("color"), 如果还没定义, 就指定默认值 ("gray")
String color = options.getProperty("color", "gray");

// 将名称为"color"的特性设定为值"green"
options.setProperty("color", "green");

// 将 Properties 对象的内容存储回文件
try {
  options.store(new FileOutputStream(configfile),    // 输出流
                "MyApp Config File");                // 文件头注释文字
} catch (IOException e) { /* 处理异常 */ }

// 在Java 5.0中, 特性可以被写入XML文件或从XML文件读取
try {
  options.storeToXML(new FileOutputStream(configfile),  // 输出流
                     "MyApp Config File");              // 注释文字
  options.loadFromXML(new FileInputStream(configfile)); // 将它读回
}
catch(IOException e) { /* 处理异常 */ }
catch(InvalidPropertiesFormatException e) { /* 变形的输出 */ }
```

参数选择

Java 1.4引入了Preferences API，这是特别用来处理用户和整个系统的参数选择，它在此用途上比Properties更有用。Preferences API是由java.util.prefs包所定义的，那个包中的主要类是Preferences。你可以用静态method Preferences.userNodeForPackage()取得含有用户特有的参数选择的Preferences对象，并且用Preferences.systemNodeForPackage()取得含有整个系统参数选择的Preferences对象。这两个method都会接收java.lang.Class对象作为唯一的自变量，并返回由那个包中所有类所共享的Preferences对象（这代表你所使用的参数选择名称在包中必须是唯一的）。一旦你拥有Preferences对象，就可以使用get() method来查询具名参数选择的字符串值，或使用该类型所特有的其他method，例如getInt()、getBoolean()以及getByteArray()。请注意，如果要查询参数选择值，就必须传递默认值给method。如果具有此指定名称的参数选择还没被注册或保存这些参数选择数据的数据库无法被访问，就会被返回默认值。Prefereneces的典型用法如下：

```
package com.davidflanagan.editor;
import java.util.prefs.Preferences;

public class TextEditor {
  // 要由参数选择值初始化的字段
  public int width;              // 以栏数计算的屏幕宽度
  public String dictionary;      // 用于拼写检查的字典名称
```

Download at http://www.pin5i.com/


```
public void initPrefs() {
  // 取得针对此包的用户和系统参数选择的 Preferences 对象
  Preferences userprefs = Preferences.userNodeForPackage(TextEditor.class);
  Preferences sysprefs = Preferences.systemNodeForPackage(TextEditor.class);

  // 查询参数选择值。请注意，一定要传递默认值
  width = userprefs.getInt("width", 80);
  // 使用系统参数选择作为默认值来查询用户参数选择
  dictionary = userprefs.get("dictionary",
                             sysprefs.get("dictionary",
                                          "default_dictionary"));
  }
}
```

除了用来查询参数选择值的get() method之外，还有用来设定具名参数选择值的对应put() method：

```
// 用户已指示新的参数选择，所以把它储存储起来
userprefs.putBoolean("autosave", false);
```

如果你的应用程序想要在程序运行期间被通知到用户或系统参数选择的改变，它可以用addPreferenceChangeListener()来注册PreferenceChangeListener。Preferences对象可以将它的参数选择的名称与值导出成为XML文件，而且可以从这样的XML文件读取参数选择。Preference对象是以层次形式存在的，那通常对应到包名称的层次。用于引导这种层次的Preferences对象的确存在，但普通的应用程序通常不会用到。

进程

在本章的前面，我们看过在同一个Java解释器中创建并操作多个运行中的线程有多容易。Java也有个java.lang.Process类，它代表了执行于解释器外的操作系统进程。Java程序可以用流与外部进程通信，就像与运行于网络上的其他计算机中的服务器通信一样。使用Process一定会有平台依赖性，而且也不便于移植，但它的确很有用。

```
// 通过查询配置文件中所要执行的命令名称来最大化可移植性
java.util.Properties config;
String cmd = config.getProperty("sysloadcmd");
if (cmd != null) {
  // 执行此命令。进程p代表执行中的命令
  Process p = Runtime.getRuntime().exec(cmd);            // 启动此命令
  InputStream pin = p.getInputStream();                  // 从中读取字节
  InputStreamReader cin = new InputStreamReader(pin);    // 将它们转换为字符
  BufferedReader in = new BufferedReader(cin);           // 读取字符行
  String load = in.readLine();                           // 取得命令输出
  in.close();                                            // 关闭流
}
```

Download at http://www.pin5i.com/

在 Java 5.0 中，java.lang.ProcessBuilder 类提供了更具灵活性的方式来启动新进程，而不是用 Runtime.exec() method。ProcessBuilder 使得通过 Map 来控制环境变量成为可能，并让工作目录的设定变得简单。它也有个选项，可将它所启动的进程的标准错误流自动转到标准输出流，这让读取 Process 的所有输出变得容易。

```
import java.util.Map;
import java.io.*

public class JavaShell {
    public static void main(String[] args) {
        // 我们使用这来个启动命令
        ProcessBuilder launcher = new ProcessBuilder();
        // 我们继承来的环境变量。我们可以修改以下这些
        Map<String,String> environment = launcher.environment();
        // 我们的进程会合并错误流与标准输出流
        launcher.redirectErrorStream(true);
        // 读取用户的输入
        BufferedReader console =
            new BufferedReader(new InputStreamReader(System.in));

        while(true) {
            try {
                System.out.print("> ");                    // 显示提示
                System.out.flush();                        // 强迫显示
                String command = console.readLine();       // 读取输入

                if (command.equals("exit")) return;    // exit 命令

                else if (command.startsWith("cd ")) { // 改变目录
                    launcher.directory(new File(command.substring(3)));
                }
                else if (command.startsWith("set ")) {// 设定环境变量
                    command = command.substring(4);
                    int pos = command.indexOf('=');
                    String name = command.substring(0,pos).trim();
                    String var = command.substring(pos+1).trim();
                    environment.put(name, var);
                }

                else { // 否则它是个要启动的进程
                    // 将命令拆散成为个别标记
                    String[] words = command.split(" ");
                    launcher.command(words);       // 设定命令
                    Process p = launcher.start(); // 并启动新的进程

                    // 现在从进程读取并显示输出，
                    // 直到不再有输出可读取
                    BufferedReader output = new BufferedReader(
                            new InputStreamReader(p.getInputStream()));
                    String line;
                    while((line = output.readLine()) != null)
                        System.out.println(line);
```

Download at http://www.pin5i.com/


```
                    // 此进程现在应该完成了，但要等待以确定
                    p.waitFor();
                }
            }
            catch(Exception e) {
                System.out.println(e);
            }
        }
    }
```

管理与安装

Java 5.0 包含了运行中应用程序的远程监控和管理的强大JMX API。完整的 javax.management API超出了本书的范围。其包含的 java.lang.management 包是针对Java虚拟机本身的监控和管理的JMX应用；java.lang.instrument 是另一个Java 5.0包，它允许定义可被用来处理运行中的JVM的“代理程序”。在提供支持的VM中，java.lang.instrument可以被用来在类文件被加载时复定义，例如，增加采样或涵盖测试程序代码。类复定义超出了本章的范围，但以下程序代码使用了Java 5.0中新出现的安装和管理特性，来判断Java程序的资源使用状况。此范例也说明了 Runtime.addShutdownHook() method，它注册了在VM开始关机时所要运行的程序代码。

```
#import java.lang.instrument.*;
import java.lang.management.*;
import java.util.List;
import java.io.*;

public class ResourceUsageAgent {
    // Java agent 类定义了一个premain() method以在main()之前运行
    public static void premain(final String args, final Instrumentation
inst) {
        // 此代理程序只是注册关机hook以在VM退出时运行
        Runtime.getRuntime().addShutdownHook(new Thread() {
                public void run() {
                    // 这段程码会在VM退出时运行
                    try {
                        // 决定要把输出送到哪里
                        PrintWriter out;
                        if (args != null && args.length() > 0)
                            out = new PrintWriter(new FileWriter(args));
                        else
                            out = new PrintWriter(System.err);

                        // 使用java.lang.management来查询尖峰线程使用状况
                        ThreadMXBean tb = ManagementFactory.getThreadMXBean();
                        out.printf("Current thread count: %d%n",
                                   tb.getThreadCount());
```

Download at http://www.pin5i.com/


```
                    out.printf("Peak thread count: %d%n",
                               tb.getPeakThreadCount());

                    // 使用java.lang.management来查询尖峰内存使用状况
                    List<MemoryPoolMXBean> pools =
                        ManagementFactory.getMemoryPoolMXBeans();
                    for(MemoryPoolMXBean pool: pools) {
                        MemoryUsage peak = pool.getPeakUsage();
                        out.printf("Peak %s memory used: %,d%n",
                                   pool.getName(), peak.getUsed());
                        out.printf("Peak %s memory reserved: %,d%n",
                                   pool.getName(), peak.getCommitted());
                    }

                    // 使用传递给premain()的Instrumentation对象来取得
                    // 所有已被载入的类的列表
                    Class[] loaded = inst.getAllLoadedClasses();
                    out.println("Loaded classes:");
                    for(Class c : loaded) out.println(c.getName());

                    out.close();  // 关闭并清空输出流
                    }

                catch(Throwable t) {
                    // shotdown hook中的异常会被忽略，所以我们
                    // 不会显式地将它列出
                    System.err.println("Exception in agent: " + t);
                    }
                }
            });
        }
    }
```

如果要使用这个代理程序监控Java程序的资源使用状况，你必须先正常地编译此类。接着，把产生出来的类文件和指定了包含 premain() method的清单存储在JAR文件中。创建包含这行的清单文件：

```
Premain-Class: ResourceUsageAgent
```

使用像这样的命令来创建JAR文件：

```
% jar cmf manifest agent.jar ResourceUsageAgent*.class
```

最后，如果要使用代理程序，就在Java解释器中加上 -javaagent标记来赋值给JAR文件和代理程序自变量：

```
% java -javaagent:agent.jar=/tmp/usage.info my.java.Program
```

Download at http://www.pin5i.com/

第六章

Java 的安全性

Java程序可以从各种资源中动态地装载Java类，其中包括不可信的资源，例如通过不安全网络可以访问到的网站。创建并处理这种自由程序代码的能力是Java极大的长处和特性之一。但是，为了让程序能顺利运行，Java非常重视能让不可信的程序代码安全地运行的安全结构，而不必担心对主机系统造成损害。

在Java中对安全系统的需求可由applet看出——applet是个小型的应用程序，被设计来内嵌在网页中（注1）。当用户访问含有applet的网页时（使用支持Java的web浏览器），web浏览器会下载定义了applet的Java类文件并加以运行。如果没有安全系统，applet可能会通过删除文件、安装病毒、窃取机密信息等来在用户的系统上造成混乱。如果applet再巧妙些，就可以利用用户的系统来伪造E-mail、产生垃圾邮件或尝试入侵其他系统。

Java对这种危险程序代码的主要防范方法就是访问控制（access control）：不安全的程序代码就不具有对特定的敏感核心Java API的访问权限。例如，不安全的applet通常不会被允许读取、写入或删除主机系统上的文件，或连接到网络上下载它的web服务器之外的任何计算机。本章会说明Java的访问控制结构以及一些其他方面的Java安全系统。

注1：applet在《Java Foundation Classes in a Nutshell》（O'Reilly）中有说明，并未涵盖于本书中，但它们在这仍能作为良好的范例。

Download at http://www.pin5i.com/

安全风险

Java从基础上的设计就考虑到安全性，这让它比许多其他已有的系统平台有更大的优势。不过，没有系统能保证100%的安全性，Java也不例外。

Java安全结构是由安全专家设计的，并已被许多其他安全专家研究调查过。其一致的结论就是此结构本身相当坚固而强壮，理论上没有任何安全漏洞（至少还没有被发现）。但是，此安全结构的实现是另一回事，在特定的Java实现中，安全漏洞被找到并修补已有很长的时间了。例如，在1999年4月，Sun在Java 1.1中的类验证器的实现就被发现存在缺陷。针对Java 1.1.6和1.1.7的补丁程序被发布，问题在Java 1.1.8中已解决。在1999年8月，Microsoft的Java Virtual Machine中出现了严重的缺陷。Microsoft修正了问题，并且不再随着它们最新版本的web浏览器发布它们的JVM。

Java VM实现中的安全缺陷很可能会继续被发现（并修补）。尽管如此，Java仍或许是目前最安全的平台。对于恶意Java程序代码利用安全漏洞的个案报道并不多。实际上，Java平台显露出足够的安全性，尤其是与一些不安全且受病毒困扰的平台作比较时。

JAVA VM安全性与类文件验证

最底层的Java安全结构牵涉到Java Virtual Machine的设计以及它所执行的字节码。Java VM不允许任何形式的对底层系统个别内存地址的直接访问，这避免了Java程序代码干扰原有的硬件和操作系统。这些在VM上的故意限制也反映在Java语言本身，它不支持指针和指针运算。此语言不允许整数被转化为对象引用，也不允许对象引用被转化为整数，而且没有任何方法可以取得对象在内存中的地址。在没有这些能力的情况下，恶意程序代码毫无立足之地。

除了Virtual Machine指令集的安全设计之外，每当加载不可信的类时，VM就会进行所谓字节码验证（byte-code verification）的过程。此过程确保类的字节码（以及它们的操作数）都是有效的；程序代码绝不会对VM堆栈造成下溢（underflow）或上溢（overflow）；局部变量在被初始化之前不会被使用；字段、method以及类访问控制修饰符都会被考虑等。验证步骤是被设计来避免VM运行有问题的程序代码，那些程序代码可能会使VM崩溃或让VM进入未被定义或未被测试的状态，使VM易于被恶意程序代码攻击。字节码验证是对精心编写带有恶意的Java字节码和未被信任的Java编译器的防范，未被信任的Java编译器可能会输出无效的字节码。

Download at http://www.pin5i.com/

验证与加密

java.security包（及其子包）提供了用于认证的类和接口。就如第五章所描述的，这个安全结构能让Java程序代码创建并验证消息摘要和数字签名。这些技术可以确保所有数据（例如Java类文件）都是可信任的：它来自于真正发出数据的人，而且数据在传输过程中没有被不小心或恶意地修改。

Java Cryptography Extension（或 JCE）包含了 javax.crypto包及其子包，这些包定义了用于数据加密和解密的类。这是对许多应用程序都很重要的安全相关特性，但与避免不可信程序破坏主机系统的基本问题没有直接相关，所以不在本章中讨论。

访问控制

就如我们在本章开头所提到的，Java安全结构的中心是访问控制：不可信的程序代码绝不可以被授予对敏感Java API部分的访问权限，若这样会让它能做有恶意的事。就如我们要在下面的章节讨论的，Java访问控制模型在Java 1.0和Java 1.2之间有重大的发展。从那时开始，访问控制模型就已变得相当稳定，从 Java 1.2 之后就很少有重大的改变。下个章节提供了Java安全性从Java 1.0到Java 1.2的简短发展历史，它展示了安全模型在最近的主要改变。

Java 1.0：沙箱

在第一版的Java中，所有安装在本地系统上的Java程序代码都是默认为可信的。但是，所有通过网络下载的程序代码都是不可信的，并且是在被开玩笑称为“沙箱（sandbox）”的受限环境中运行。沙箱的访问控制策略是由当前安装的java.lang.SecurityManager对象所定义。当系统程序代码要执行受限制的操作时（例如读取本地文件系统的文件），它会先调用当前安装的SecurityManager对象的适当method（例如checkRead()）。如果不可信的程序代码正在运行，SecurityManager就会抛出SecurityException，以避免受限制的操作发生。

最常见的SecurityManager类的用户是支持Java的web浏览器，它安装了SecurityManager对象来允许applet在不损害主机系统的情况下运行。当然，安全策略的细节是web浏览器的实现细节，但applet通常会被以下几种方式限制：

- applet不可以读取、写入、重命名或删除文件。它不能查询文件的长度、修改日期甚至检查指定文件是否存在。同样地，applet不能创建、列出或删除目录。

Download at http://www.pin5i.com/

- 除了下载applet的计算机之外，applet不能连接到其他计算机或接受来自其他计算机的连接。它不能使用任何特权端口（亦即端口1024以下（包含1024）的端口）。
- applet不能执行系统层次的功能，例如加载原生类库、产生新的进程或结束Java解释器。除了它本身所创建的之外，applet不能操作任何线程或线程组。在Java 1.1和之后的版本中，除了随着applet一起下载的之外，applet不能使用Java Reflection API来取得关于类的非公共成员的信息。
- applet不能访问与图形和GUI相关的功能。它不能初始化打印工作、访问系统的剪贴板或事件队列。此外，所有由applet创建的窗口通常都会显示明显的可见指示，标示它们是“不安全的”，以避免applet使用某些其他应用程序的外观来进行欺骗。
- applet不能读取某些系统特性，尤其是`user.home`和`user.dir`特性，它们定义了用户的根目录和当前工作目录。
- applet不能通过注册新的`SceurityManager`对象来绕过这些安全限制。

沙箱的运作方式

假设applet（或运行于沙箱中的一些其他未被信任的程序代码）试图通过传递*/etc/passwd*这个文件名给`FileInputStream()`来读取文件*/etc/passwd*的内容。编写`FileInputStream`类的程序员察觉到此类提供了对系统资源（文件）的访问，所以此类的使用因而受访问控制的支配。基于此原因，他们编写了`FileInputSteam()`构造函数来使用`SecurityManager`类。

每当`FileInputStream()`被调用时，它会检查`SecrutiyManager`对象是否已安装。如果安装了，构造函数就会调用`SecurityManager`对象的`checkRead()` method将文件名（在这个例子中是*/etc/passwd*）当成唯一的自变量来传递。`checkRead()` method没有返回值，它不是正常返回就是抛出`SecurityException`。如果此method返回，那么`FileInputStream()`构造函数就只执行初始化所需的操作并返回，否则，它会允许`SecurityException`传播至调用者。当这个情况发生时，`FileInputStream`对象就不会被创建，而applet无法取得对*/etc/passwd*文件的访问权。

Java 1.1：数字签名类

Java 1.1保留了Java 1.0的沙箱模型，但增加了`java.security`包及其数字签名功能。通过使用这些功能，Java类可以用数字签名并验证。因此，web浏览器和其他的Java安装可以被设定为包括可信实体的有效数字签名的可信的下载程序代码。这样的程序代码被当成是安装在本地的，所以它被给予对Java API的完全访问权限。在这个版本中，*javakey*程序管理了以数字方式签名的Java程序代码的JAR文件。虽然Java 1.1增加了

Download at http://www.pin5i.com/

重要的功能来信任以数字方式签名的程序代码（否则程序代码就不会被信任），但它仍坚持基本的沙箱模型：被信任的程序代码能取得所有访问权，而未被信任的程序代码只取得非常受限制的访问权。

Java 1.2：权限与策略

Java 1.2 把新的访问控制特性引入了 Java 安全结构中，这些特性是由 java.security 包中的类所实现。Policy 类是最重要的之一：它定义了 Java 安全策略。Policy 对象将 CodeSource 对象映射至相关联的 Permission 对象的集合。CodeSource 对象代表一段 Java 程序代码的源代码，它包括了类文件的 URL（可以是在本地的文件）和将数字签名应用至类文件的实体列表。Permission 对象与 Policy 中的 CodeSource 相结合，定义了会被授予来自指定源代码的程序代码的权限。各种 Java API 包括了代表不同类型权限的 Permission 子类，包括了 java.lang.RuntimePermission、java.io.FilePermission 以及 java.net.SocketPermission 等。

在此访问控制模型下，SecurityManager 类仍然是主要类，访问控制请求仍会通过调用 SecurityManager 的 method 来完成。但是，默认的 SecurityManager 实现将大部分的请求委派至 AccessController 类，它会以 Permission 和 Policy 结构为基础来作出访问决定。

Java 1.2 访问控制结构有几个重要特性：

- 来自不同出处的程序代码可以被给予不同的权限。换言之，此结构支持各种信任层次，甚至于在本地安装的程序代码也可以被当成未被信任或部分未被信任。在此结构下，只有系统类和标准扩展包能被当成完全信任来执行。
- 不再需要定义 SecurityManager 的自定义子类来定义安全策略。策略可以由系统管理员通过编辑文本文件或使用 *policytool* 程序来配置，这会在第八章中说明。
- 此结构未被限制于 SecurityManager 类中固定的访问控制 method。Permission 子类可以轻易地被定义来管理对系统资源的访问（例如，通过包含原生程序代码的标准扩展包来接触系统资源）。

权限与策略的运作方式

我们回到 applet 的范例，此 applet 尝试创建 FileInputSteram 来读取文件 */etc/passwd*。在 Java 1.2 及之后的版本中，FileInputStream() 构造函数的行为与 Java 1.0 和 Java 1.1 中的完全一样：它会查看 SecurityManager 是否已被安装，如果安装了，就调用它的 checkRead() method 来传递要被读取的文件名。

Download at http://www.pin5i.com/

在Java 1.2中有改变的是checkRead() method的默认行为。除非程序已用它自己的安全管理员来取代默认的安全管理员，否则默认的实现会创建FilePermission对象来代表所需的访问。此FilePermission对象有个“/etc/passwd”目标（target）并且有个“读取”动作（action）。checkRead() method会把这个FilePermission对象传递给java.security.AccessController类的静态checkPermission() method。

在Java 1.2中进行真正的访问控制工作的是AccessController和它的checkPermission() method。此method会判定每个进行调用method的CodeSource，并使用当前的Policy对象来决定与其相关联的Permission对象。有了这些信息，AccessController就可以决定对*/etc/passwd*文件的访问是否应该被允许。

Permission类同时代表了由Policy所授予的权限以及由像FileInputStream()这种构造函数所请求的权限。当请求权限时，Java通常会用到非常明确的目标（像是“/etc/passwd”）的FilePermission（或其他的Permission子类）。但是，当授予权限时，Policy通常会以通配符目标代表许多文件（例如“/etc/*”）来使用FilePermission对象。Permission子类（例如FilePermission）的关键特性之一，就是它定义了implies() method，可以判断拥有读取“/etc/*”的权限是否意味着拥有读取“/etc/passwd”的权限。

针对所有人的安全性

程序员、系统管理员以及终端用户都有不同的安全性考虑，因此，在Java安全结构中会扮演不同的角色。

针对系统程序员的安全性

系统程序员是定义新Java API的人，这些API允许对敏感系统资源进行访问。这些程序员通常会运用对系统具有未保护的访问的原生method。它们必须使用Java访问控制结构来避免未被信任的程序代码执行那些原生method。为了要实现这一点，系统程序员必须在他们的程序代码中小心地在适当地方插入SecurityManager调用。系统程序员可能会选择使用已存在的Permission子类来管理对API所能接触的系统资源的访问，或是可能会决定要定义Permission的特定子类。

系统程序员担负了极大的安全性重任：如果在程序代码中没有执行适当的访问控制检查，就可能会危及整个Java平台的安全性。相关的细节很复杂，已超越了本书的范围。但是，幸好在Java中与原生method有关的系统程序设计相当少。我们几乎全都是应用程序员，可以只依赖已存在的API。

Download at http://www.pin5i.com/

针对应用程序员的安全性

使用了核心 Java API 和标准扩展包但没有定义新扩展包或编写原生 method 的程序员，可以只依赖创建那些 API 的系统程序员在安全性上的努力。换言之，我们大部分的 Java 程序员可以只是使用 Java API，而无需担心将安全性漏洞引进 Java 平台。

事实上，应用程序员很少必须使用访问控制结构。如果你正在编写可能会被当成未被信任的程序代码的Java程序代码，就应该会察觉到由典型的安全策略加在未被信任的程序代码上的限制。请记住，有些 method（例如读取与写入文件）可能会抛出 SecurityException 对象，但不要觉得你必须自己编写程序代码来捕获这些异常。通常，对 SecurityException 的适当响应是允许它不被捕获而被传播出去，以让它终止应用程序。

有时，身为一个应用程序员，你会想编写个应用程序，它可以加载未被信任的类并通过访问控制检查来加以运行。如果要在 Java 1.2 及之后的版本中完成，就必须先安装安全管理器：

```
System.setSecurityManager(new SecurityManager());
```

接着使用 java.net.URLClassLoader 来载入未被信任的类。URLClassLoader 会指定一组默认的安全权限给它加载的类，但在某些情况下，你可能会想要修改由 Policy 和 PermissionCollection 类为被加载程序代码授予的权限。

针对系统管理员的安全性

在Java 1.2和之后的版本中，系统管理员负责定义他们所维护之计算机的默认安全策略。默认的策略被存储在 Java 安装过程中的 *lib/security/java.policy* 文件中。系统管理员可以手动编辑这个文本文件，或使用来自 Sun 的 *policytool* 程序以可视化方式来编辑文件。*policytool* 是较好的定义策略的方式，所以底层策略文件的语法不在本书中说明。

默认的 *java.policy* 文件定义了与 Java 1.0 和 Java 1.1 非常类似的策略：系统类和已安装的扩展包会被完全信任，而其他的程序代码都是未被信任的，只能有一些简单的权限。例如，在一些组织中，将额外的权限授予从安全的内部网络下载的程序代码是适当的。

为了定义有效的安全策略，系统管理员必须了解Java平台的各种 Permission 子类、它们所支持的目标和行为名称以及授予任一个特定权限的安全实现。这些主题在标题为“Permissions in the Java 2 Standard Edition Development Kit (JDK)”的文件中说明得很详细，这份文件可以在 *http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html* 在线取得。

Download at http://www.pin5i.com/

针对终端用户的安全性

大部分的终端用户都完全没有考虑到安全性：他们的Java程序应该只能在他们不介入的安全方式下运行。但是，有些具有经验的终端用户可能会想要定义他们自己的安全策略。终端用户可以通过自己运行*policytool*来定义个人策略文件，此文件扩大了系统策略。默认的个人策略是存储于用户根目录中名为*.java.policy*的文件。根据默认，Java会加载此策略文件并使用它来扩大系统策略文件。

在Java 1.2及之后的版本中，当启动Java解释器时，用户可以指定额外的策略文件来使用。如果要这么做，需使用-D选项来定义java.security.policy属性。例如：

```
C:\> java -Djava.security.policy=policyfile UntrustedApp
```

该行在使用由文件或URL策略文件所指定的策略来扩大默认的系统和用户策略后运行UntrustedApp类。如果是要取代系统和用户策略而不是加以扩大，可以在属性说明中使用双等号：

```
C:\> java -Djava.security.policy==policyfile UntrustedApp
```

但请注意，只有在SecurityManager已安装的情况下，指定策略文件才会有用。如果用户不信任应用程序，那么他可能不会相信那个应用程序会自动安装它自己的安全管理员。在这样的情况下，他可以定义java.security.manager系统属性：

```
C:\> ava -Djava.security.manager -Djava.security.policy=policyfile \ UntrustedApp
```

此属性的值并不重要，只要把它定义得足以告诉Java解释器要自动安装默认的SecurityManager对象，以规范应用程序遵守系统、用户和java.security.policy策略文件中所描述的访问控制策略即可。

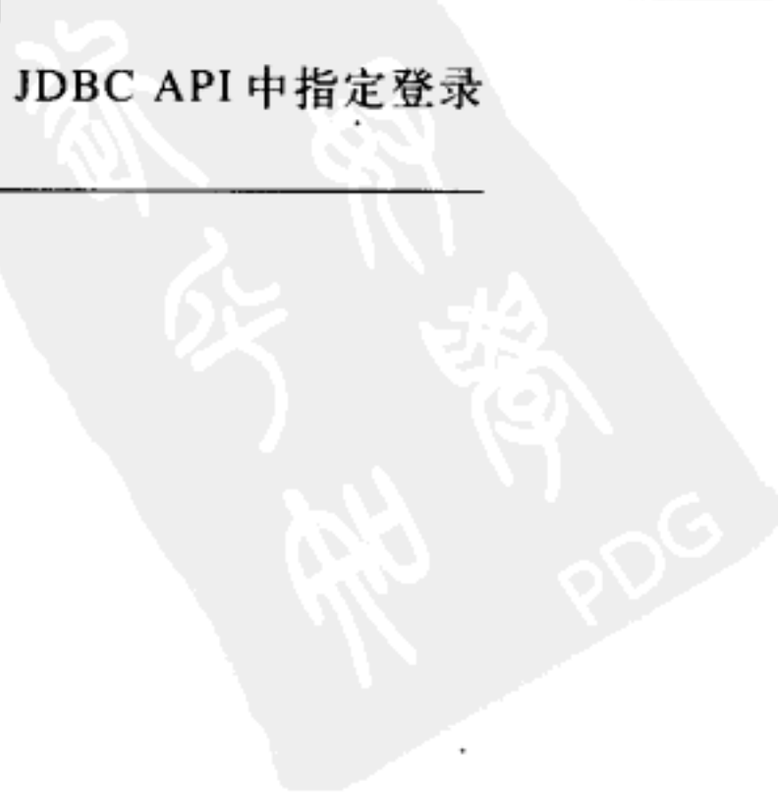
Permission 类

表6-1列出了一些由核心Java平台定义的重要Permission子类，并归纳了它们所代表的权限。关于这些权限类的完整列表/详细说明、它们的目标和行为目录、method列表以及它们所需的权限，请参阅*http://java.sun.com/j2se/1.5.0/docs/guide/security/permissions.html*（此文件是标准说明文件集的一部分，可以与JDK一起下载）。

Download at http://www.pin5i.com/

表 6-1：Java Permission 类

Permission 类	说明
java.security.AllPermission	此特殊权限类的 instance 包含了所有其他的权限
javax.sound.sampled.AudioPermission	控制播放与记录声音的能力
javax.security.auth.AuthPermission	控制对 javax.security.auth 及其子包中认证 method 进行访问的能力
java.awt.AWTPermission	控制对 java.awt 及其子包中敏感 method 的访问
java.io.FilePermission	管理对文件系统的访问
java.util.logging.LoggingPermission	控制程序修改登录配置（logging configuration）的能力
java.net.NetPermission	管理对与网络连接相关数据（例如流处理程序与 HTTP 认证）的访问。请参阅 java.net.SocketPermission
java.util.PropertyPermission	管理对系统属性的访问
java.lang.reflect.ReflectPermission	管理通过 java.lang.reflect 包对通常无法被访问的类和类成员的访问
java.lang.RuntimePermission	管理对一些method和数据的访问。有许多受控制的 method 是由 java.lang.System 和 java.lang.Runtime 所定义
java.security.SecurityPermission	管理对各种与安全性相关的 method 的访问
java.io.SerializablePermission	管理对与序列化相关的 method 的访问
java.net.SocketPermission	管理对网络的访问
java.sql.SQLPermission	管理在 java.sql JDBC API 中指定登录流的能力



Download at http://www.pin5i.com/

第七章

程序设计与文档规范

本章说明了一些重要且有用的Java程序设计与文档规范。其中包含：

- 一般命名与大小写惯例
- 可移植性提示与惯例
- Javadoc 说明文件注释语法与惯例
- JavaBeans 惯例

在此所说明的惯例都不具强制性。但是，遵循这些惯例能让你的程序代码更易阅读和维护、具有可移植性并能自我注释（self-documenting）。

命名与大小写惯例

以下广为采用的命名惯例适用于Java中的包、类型、method、字段以及常量。因为这些惯例几乎全世界都遵循，而且因为它们会影响到你定义的类的公共API，所以应该要小心谨慎地遵守：

包

为了确保你的公共可见的包名是唯一的，可以在它们前面加上你经过颠倒的Internet域名（例如，`com.davidflanagan.utils`）。所有的包名都应该是小写。分布于一切完备的JAR文件中、由应用程序在内部使用的程序代码包不是公共可见的，所以不必遵循此惯例。在这样的情况下，使用应用程序名称作为包名或包前缀是很常见的。

Download at http://www.pin5i.com/

类型

类型名称应该以大写字母开头，而且可以用大小写混合的方式（例如String）。如果类名是由多个单词所组成，那么其中的每个单词都应该以大写字母开头（例如，StringBuffer）。如果类型名称或类型名称中的其中一个单词是字母缩写，则此字母缩写就应该全部用大写字母编写（例如URL、HTMLParser）。

因为类和枚举类型是被设计来代表对象，所以你应该选择名词形式的类名（例如Thread、Teapot、FormatConverter）。

当interface被用来提供关于实现此interface类的额外信息时，选择具有形容词形式的interface名称是很常见的（例如Runnable、Colneable、Serializable）。annotation类型也常是用这种方式命名。当interface的运行较像是抽象父类时，就使用名词形式的名称（例如Document、FileNameMap、Collection）。

method

method名称一定应以小写字母开头。如果名称包含了多个单词，那么在第一个单词之后的每个单词都应以大写字母开头（例如insert()、insertObject()、insertObjectAt()）。method名称通常都是经过精心设计的，所以一般它们的第一个单词是动词。method名称可以用需要的长度让它们的用途清楚，但要尽可能选择简洁的名称。

字段与常量

非常量字段的名称遵循着与method名称相同的大小写惯例。如果字段是static final常量，就应该使用大写来编写。如果常量的名称包含了多个单词，那么这些单词就应该以下划线隔开（例如MAX_VALUE）。字段名称应该选择最能说明字段或其取值的含义的。

由枚举类型定义的常量通常也会都用大写字母表示。但是，因为其他的程序设计语言对枚举值使用小写或大小写混合，所以此惯例并没有像在类或interface的static final字段中使用大写字母的惯例一样具有权威。

参数

method参数遵循与非常量字段相同的大小写惯例。method参数的名称出现在method的说明文件中，所以你应该选择能让参数的用途尽可能清楚的名称。要试着让参数名称为一个单词并以一致的方式来使用它们。例如，如果Widget-Processor类定义了许多会接受Widget对象作为第一参数的method，就在每个method中把这个参数命名为widget甚至是w。

局部变量

局部变量名称是实现细节，在类的外部绝不会看到。不过，选择好的名称能让你的

Download at http://www.pin5i.com/

程序代码较易于阅读、理解以及维护。变量的命名通常会遵循与method和字段相同的惯例。

除了针对特殊名称类型的惯例之外，还有一些与名称中所使用的字符有关的惯例，应该在你的名称中使用。Java允许标识符中有$字符，但依惯例，此用法是留给由源代码处理程序所产生的合成名称（例如，Java编译器用它来让内部类运行）。此外，Java允许对名称使用来自整个Unicode字符集的字母和数字字符（alphanumeric character）。虽然这对于非英语系（non-English-speaking）的程序员很方便，但Unicode字符的使用应该要被限制在局部变量、私有method和字段以及不属于类的公共API中的其他名称。

可移植性惯例和纯Java规则

Sun的格言，或者说是它对于Java的核心价值论点就是“Write once, run anywhere”。Java使得具有可移植性程序的编写变得容易，但Java程序不能自动在所有的Java平台上顺利运行。以下提示能帮助避免出现可移植性方面的问题。像此处所列出的可移植性规则，是Sun现在已废止的“100% Pure Java”认证项目与品牌活动的焦点。

native method

具有可移植性的Java程序代码可以使用核心Java API中的所有method，其中包括被实现为原生（native）method的那些method。但是，具有可移植性的程序代码绝不可以定义它自己的native method。就其本质来看，native method必须被移植到各个新平台，所以它们直接破坏了Java的“Write once, run anywhere”承诺。

Runtime.exec() method

调用`Runtime.exec()` method来产生一个进程并执行原生系统上的外部命令，在具有可移植性的程序代码中是很少被允许的。这是因为要被执行的原生OS命令无法保证在所有的平台上都存在或都以同样的方式执行。唯一能合法使用`Runtime.exec()`的时机，就是在用户被允许指定所要执行的命令时。这可以通过在运行时输入命令或通过在配置文件或参数选择对话框中指定。

System.getenv() method

使用`System.getenv()`是不具有可移植性的。此method之前已废止，但在Java 5.0中被再次引入。

无说明文件的类

具有可移植性的Java程序代码只能使用Java平台的有说明文件部分的类和接口。大部分的Java实现都会有额外的无说明文件的公共类，那些是实现的一部分，但不是Java平台的一部分。没有任何方式能避免程序使用与依赖这些无说明文件的

Download at http://www.pin5i.com/

类，但这么做是不具有可移植性的，因为无法保证这些类在所有的 Java 实现或平台上都存在。

java.awt.peer 包

java.awt.peer 包中的接口是 Java 平台的一部分，但只是对由 AWT 实现的用法做使用说明。直接使用这些接口的应用程序是不具可移植性的。

针对专门实现（Implementation-specific）的特性

具有可移植性的程序代码绝不可以依赖单独实现所特有的特性。例如，Microsoft 发布了一个 Java 运行时系统的版本，其中包含了一些 method，但那些 method 并非由 Sun 定义为 Java 平台的一部分。很明显地，任何依赖这种扩展功能的程序会无法被移植到其他的平台。Microsoft 的 Java 平台私有扩展功能造成了 Sun 和 Microsoft 之间的法律纠纷，最终使得 Microsoft 不再支持 Java。

针对专门实现的缺陷

就和具有可移植性的程序代码绝不可以依赖针对专门实现的特性一样，它也绝不可以依赖针对专门实现的缺陷。如果类或 method 的行为与规范所定义的不同，具有可移植性的程序就不可以依赖这个行为，因为那可能在不同的平台上会有差异且最终会被修正。

针对专门实现的行为

有时不同的平台和不同的实现会呈现出不同的行为，依据 Java 的规范来看，那些都是合法的。具有可移植性的程序代码绝不可以依赖任何一个特定的行为。例如，Java 规范没有规定具有相等优先级的线程可共享 CPU，或是长时间运行的线程可以激活另一个具有相同优先级的线程。如果应用程序假设了其中一个特定的行为，那么它就可能不会在所有的平台上都正确地运行。

标准扩展

具有可移植性的程序代码可以依赖针对 Java 平台的标准扩展，但如果这么做，就应该明确地指出它使用的是哪个扩展，并当它在未安装那个扩展的系统上运行时，要有明确地给出的适当错误信息。

完整的程序

所有具有可移植性的 Java 程序都必须完整且独立：除了核心平台和标准扩展类之外，它必须提供所有它要使用的类。

定义系统类

具有可移植性的 Java 程序代码绝不会在任何的系统或标准扩展包中定义类。这样做破坏了那些包的保护界限，并且暴露了在包可见的实现细节。

Download at http://www.pin5i.com/

硬编码文件名

具有可移植性的程序不会包含硬编码的（hardcoded）文件或目录名称。这是因为不同的平台会有差异极大的文件系统组织方式，并使用了不同的目录分隔字符。如果你需要处理文件或目录，就要让用户指定文件名或至少要指定可找到此文件的基目录。此声明可以在运行时在配置文件中或作为程序的命令行自变量来完成。当把文件或目录名称与目录名称合并时，要使用`File()`构造函数或`File.separator`常量。

换行字符

不同的系统使用了不同的字符或字符序列作为换行字符（line separator）。不要把`\n`、`\r`或`\r\n`硬编码为程序的换行字符，而是要用`PrintStream`或`printWriter`的`println()` method，它会自动以适合于此平台的换行字符来结束一行或是使用`line.separator`系统属性值。在Java 5.0及之后的版本中，你也可以对`java.util.Formatter`及相关类的`printf()`和`format()` method使用`"%n"`。

Java说明文件的注释

大部分Java程序代码中的普通注释是在说明程序代码的实现细节。相较之下，Java语言规范定义了所谓*doc*注释的特殊注释类型，用来说明你的程序代码的API。doc注释是以`/**`开始（而不是一般的`/*`）并以`*/`结束的普通多行注释，它会出现在类型或成员定义的前面，并包含针对那个类型或成员的说明文件。此说明文件可以包含简单的HTML格式化标记和其他提供额外信息的特殊关键字。doc注释会被编译器忽略，但它们可以由*javadoc*程序被提取出来并转换成在线的HTML说明文件（关于*javadoc*的更多信息，请参阅第八章）。这有个包含了适当doc注释的范例类：

```
/**
 * 这个具有不变性的类代表了<i>复数</i>
 *
 * @作者 David Flanagan
 * @版本 1.0
 */
public class Complex {
  /**
   * 存放此复数的实数部分
   * @参阅 #y
   */
  protected double x;

  /**
   * 存放此复数的虚数部分
   * @参阅 #x
```

Download at http://www.pin5i.com/


```
     */
    protected double y;

    /**
     * 创建新的Complex对象来代表复数x+yi
     * @参数x复数的实数部分
     * @参数y复数的虚数部分
     */
    public Complex(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * 将两个Complex对象相加并产生代表它们总和的第三个对象
     * @参数c1 Complex对象
     * @参数c2 另一个Complex对象
     * @返回用来代表<code>c1</code>和<code>c2</code>总和
     * 的新Complex对象
     * @异常java.lang.NullPointerException
     *  如果其中有个自变量是<code>null</code>
     */
    public static Complex add(Complex c1, Complex c2) {
        return new Complex(c1.x + c2.x, c1.y + c2.y);
    }
}
```

doc 注释的结构

doc注释的主体应该要以关于要被说明的类型或成员的一句话摘要来开始。这句话本身可以显示作为归纳的说明文件，所以它应该被独立编写。在起始句子的后面可以接着任意数量的其他句子和段落，用来更详细地描述类、接口、method或字段。

在具描述性的段落后面，doc注释可以包含任意数量的其他段落，其中每个段落都是以特殊的doc注释标记开始，例如@author、@param或@returns。这些加上标记的段落提供了关于类、接口、method或字段的专门信息，可让*javadoc*程序以标准的方式来显示。完整的doc注释标记列于接下来的段落中。

在doc注释中，具有描述性的信息可以包含简单的HTML标注标记，例如用于强调的<I>、用于类、method以及字段名称的<code>以及用于多行程序代码范例的<pre>。它也可以包含<p>标记来将语句拆为分隔的段落以及用<ul>、<li>和相关的标记来显示加上标号的列表和类似的结构。但请记住，你所编写的信息是被内嵌在更大、更复杂的HTML文件中。基于此原因，doc注释不应该包含HTML的主要结构标记，例如<h2>或<hr>，因为那可能会干扰到较大文档的结构。

要避免在doc注释中使用<a>标记来引入超级链接或交叉引用，而是使用特殊的

Download at http://www.pin5i.com/

{@link} doc 注释标记，它与其他的 doc 注释标记不同，可以出现在 doc 注释中的任何地方。就如在接下来的章节中所描述的，{@link}标记能让你在不知道 *javadoc* 所使用的 HTML 结构惯例和文件名的情况下，指定对其他类、接口、method 以及字段的超级链接。

如果你想在doc注释中加上图像，可以把图像文件放在源代码目录的*doc-files*子目录中。让图像与类具有相同的名称并加上整数后缀。例如，出现于名称为Circle的类的 doc 文件中的第二张图片，可以用这个 HTML 标记来定义：

```
<img src="doc-files/Circle-2.gif">
```

因为这些doc注释是被内嵌在Java注释中，所以在处理之前，每一行注释的前置空白和星号都会被剔除。因此，你不必担心星号出现在最后产生的说明文件中，或担心注释的缩排会影响到以 <pre> 标记被包含于注释中的程序代码范例的缩排。

doc 注释标记

javadoc 能识别一些专门的标记，都是以 @ 字符开始。这些 doc 注释标记能让你以标准的方式将专门的信息编码到你的注释中，而且它们能让 *javadoc* 选择针对那份信息的适当输出格式。例如，@param 标记让你指定 method 的单一参数名称和意义。*javadoc* 可以提取出这份信息，并使用 HTML 的 <dl> 列表、HTML 的 <table> 或任何适合的方式来加以显示。

以下的doc注释标记能被*javadoc*识别。doc注释通常应该会以此处所列的顺序来使用这些标记：

@author *name*

加入包含指定名称的“Author:”项。此标记应该针对每个类或接口定义来使用，但绝不可以用于单独的 method 和字段。如果一个类有多个作者，那么就在相邻的行上使用多个 @author 标记。例如：

```
@author David Flanagan
@author Paula Ferguson
```

依时间顺序列出作者，原始作者放在第一个。如果没人知道作者，可以使用“unascribed”。*javadoc* 不会输出作者信息，除非定义了 -author 命令行自变量。

@version *text*

插入含有指定文本的“Version:”项。例如：

```
@version 1.32, 08/26/04
```

此标记应该被包含在每个类和接口的 doc 注释中，但不可以用于单独的 method 和

Download at http://www.pin5i.com/

字段。它常与版本控制系统（例如SCCS、RCS或CVS）的版本自动编号功能一起使用。*javadoc* 不会在产生的说明文件中输出版本信息，除非定义了 -version 命令行自变量。

@param *parameter-name description*

对当前method的“Parameters:”段落增加指定的参数及其描述。method或构造函数的doc注释必须为method的每个参数包含一个 @param。这些标记出现的顺序应该与method指定的参数的顺序相同。此标记只能用于method和构造函数的doc注释中。你应该尽可能使用词组和语句段以让描述保持简短。但是，如果参数需要有详细的说明文件，则此描述可以覆盖数行并包含任意数量的文字。针对源代码形式上的可读性，要考虑使用空格来让这些描述彼此对齐。例如：

```
@param o        要被插入的对象
@param index    所要插入的位置
```

@return *description*

插入含有特定描述的“Returns:”段落。此标记应该出现在method的每个doc注释中，除非method返回 void 或它是个构造函数。描述的长度可以依需要决定，但请考虑使用语句段来保持简短。例如：

```
@return <code>true</code> if the insertion is successful, or
        <code>false</code> if the list already contains the specified object.
```

@exception *full-classname description*

加入包含了指定异常名称和描述的“Throws:”项。针对出现在 throws 子句中的每个已校验异常，method或构造函数的doc注释应该包含一个@exception标记。例如：

```
@exception java.io.FileNotFoundException
           If the specified file could not be found
```

当有些异常是method的用户想要捕获的异常时，@exception标记可以被选择性地用来说明method可能抛出的未校验异常（亦即RuntimeException的子类）。如果method可以抛出多个异常，那么就在相邻行上使用多个 @exception标记并依字母顺序列出这些异常。此描述可以依所需尽可能短或长，以说明异常的重要性。该标记只能用于method和构造函数注释。@throws标记是@exception的同义词。

@throws *full-classname description*

此标记是 @exception 的同义词。

@see *reference*

加入含有指定引用的“See Also:”项。此标记可以出现在任何类型的doc注释中。针对 *reference* 的语法会在本章稍后的“doc注释中的交叉引用”中说明。

Download at http://www.pin5i.com/

@deprecated *explanation*

此标记说明了接下来的类型或成员已过时，应避免使用。*javadoc*会对说明文件加上显著的“Deprecated”项并包含指定的*explanation*文字。此段文字应说明类或成员何时会过时，如果有可能，还要提供替代的类或成员并包含对其的链接。例如：

```
@deprecated As of Version 3.0, this method is replaced
            by {@link #setColor}.
```

虽然Java编译器会忽略所有的注释，但它会注意到doc注释中的@deprecated标记。当此标记出现时，编译器会注意到它所产生的类文件中的过时问题。这让它能对其他依赖已过时特性的类提出警告。

@since *version*

说明类型或成员何时会被加入API。此标记后面应该接着版本号或其他版本规范。例如：

```
@since JNUT 3.0
```

每个针对类型的doc注释都应该包含一个@since标记，而对于在类型的最初版本后加入的任一个成员，在它们的doc注释中都应该要有@since标记。

@serial *description*

严格说来，类被序列化的方式是它的公共API的一部分。如果你编写了预期会被序列化的类，就应该用@serial和以下所列出的相关标记来说明它的序列化格式。对于任一个属于Serializable类的序列化状态的一部分的字段，@serial应该要出现在doc注释中。对于使用默认序列化机制的类，这表示了所有未被声明为transient的字段，其中包括了被声明为private的字段。*description*应该是字段及它在序列化对象中用途的简短描述。

在Java 1.4中，你也可以在类和包层次上使用@serial标记来指定“序列化格式页”是否应该针对类或包而产生。语法为：

```
@serial include
@serial exclude
```

@serialField *name type description*

Serializable类可以通过在名称为serialPersistentFields的字段中声明由ObjectStreamField对象所组成的对象，来定义它的序列化格式。对于这样的类，针对serialPersistentFields的doc注释应该要包含针对数组的每个元素的@serialField标记。每个标记指定了类的序列化状态中特定字段的名称、类型和描述。

@serialData *description*

Serializable类可以定义一个writeObject() method来写入由默认的序列化

Download at http://www.pin5i.com/

机制所写入的之外的数据。Externalizable类定义了一个writeExternal() method，来负责将对象的完整状态写至序列化流。@serialData标记应该被使用在针对这些writeObject()和writeExternal() method的doc注释中，而description应该要说明method所使用的序列化格式。

在线doc注释标记

除了前面的标记之外，*javadoc*也支持好几种在线标记（inline tag），它们可以出现在HTML文本在doc注释中所能出现的任何地方。因为这些标记会直接出现在HTML文本流中，所以必须要用大括号作为分隔字符来隔开标记文本与HTML文本。Java所支持的在线标记包括了以下几种：

{@link *reference*}

在Java 1.2及之后的版本中，{@link}标记就像是@see标记，除了它不是在特殊的“See Also:”段落中放置对指定引用的链接，而是在线插入链接。{@link}标记可以出现在HTML文本在doc注释中所能出现的任何地方。换言之，它可以出现在类、接口、method或字段的初始描述中以及在与@param、@returns、@exception和@deprecated标记相关联的描述中。{@link}的*reference*使用了接下来在“doc注释中的交叉引用”中所描述的语法。例如：

```
@param regexp The regular expression to search for. This string
              argument must follow the syntax rules described for
              {@link java.util.regex.Pattern}.
```

{@linkplain *reference*}

在Java 1.4及之后的版本中，{@linkplain}标记就像是{@link}，除了链接的文本是用一般的字型来格式化，而不是{@link}标记所使用的程序代码字型。这在*reference*同时包含链接*feature*以及说明显示于链接中的替代文字的*label*时最有用。关于*reference*自变量的*feature*和*label*部分的讨论，请参阅“doc注释中的交叉引用”。

{@inheritDoc}

当一个method覆盖父类中的method或实现接口中的method时，你可以省略doc注释，*javadoc*会自动继承来自于被覆盖或被实现method的说明文件。但在Java 1.4中，{@inheritDoc}标记能让你继承单独标记中的文字。此标记也允许你继承及扩大具有描述性的注释文字。如果要继承单独标记，可以这样：

```
@param index @{inheritDoc}
@return @{inheritDoc}
```

如果要继承整个doc注释，把你自己的文字加在它之前或之后，可以这样使用标记：

Download at http://www.pin5i.com/


```
This method overrides {@link java.langObject#toString}, documented as follows:
<P>{@inheritDoc}
<P>This overridden version of the method returns a string of the form...
```

{@docRoot}

此在线标记没有接收参数，而且会以对产生的说明文件所在根目录的引用来取代。这对引用外部文件（例如图像或版权声明）的超级链接很有用：

```
<img src="{@docroot}/images/logo.gif">
This is <a href="{@docRoot}/legal.html">Copyrighted</a> material.
```

{@docRoot} 是在 Java 1.3 中引入的。

{@literal text}

此在线标记会照字面显示文字，它会略过其中所有的 HTML 并忽略可能包含的 *javadoc* 标记。它不会保持留白格式，但在 <pre> 标记中留白格式很有用。{@literal}于 Java 5.0 及之后的版本中被提供。

{@code text}

此标记就像{@literal}标记，但会以程序代码字型显示直接量文字。等于是：

```
<code>{@literal text}</code>
```

{@code} 于 Java 5.0 及之后的版本中被提供。

{@value}

{@value}标记在不具有自变量的情况下，被在线用于static final字段的doc注释中并会被那个字段的常量值所取代。此标记是在Java 1.4中引入的，而且只能用于常量字段。

{@value *reference*}

这是{@value}的变体，包含了对static final 字段的 *reference*，它会被那个字段的常量值所取代。虽然无自变量版本的{@value}标记是在 Java 1.4 中引入的，但这个版本只有在Java 5.0及之后的版本中才被提供。关于 *reference* 的语法，请参阅“doc 注释中的交叉引用”。

doc 注释中的交叉引用

@see 标记以及在线标记{@link}、{@linkplain}和{@value}都把交叉引用编码成一些其他的说明文件源代码，通常是针对其他类型或成员的说明文件注释。

*reference*可以采用三种不同的形式。如果它以引号字符开始，它就会被当成书籍名称或某种可打印资源并以此被显示。如果 *reference*是以 < 字符开始，它就会被当成使用了<a>标记的HTML超级链接，而超级链接会被插入输出的说明文件中。此形式的@see 标记可以插入链接到其他的在线文件，例如程序员指南或用户手册。

Download at http://www.pin5i.com/

如果 `reference` 不是被引号括起的字符串或超级链接，那么它就会被预期为具有以下形式：

```
feature label
```

在这种情况下，*javadoc* 输出由 `label` 所指定的文字并将它编码为对指定 `feature` 的超级链接。如果 `label` 被省略（通常会如此），*javadoc* 就使用所指定的 `feature` 的名称来代替。

`feature` 可以引用包、类型或类型成员，使用的是以下的其中一种形式：

pkgname

对具名包的调用。例如：

```
@see java.lang.reflect
```

pkgname.typename

对由完整包名所指定的类、接口、枚举类型或 annotation 类型的引用。例如：

```
@see java.util.List
```

typename

在没有加上包名的情况下，对所指定的类型的引用。例如：

```
@see List
```

javadoc 会通过搜索当前包及已导入类的列表来解析具有这个名称的引用。

typename#methodname

对指定类型中具名 method 或构造函数的引用。例如：

```
@see java.io.InputStream#reset
@see InputStream#close
```

如果类型是在没有包名的情况下被指定，它就会以对 *typename* 所述的方式那样来解析。如果 method 被重载或类以相同名称定义了字段，那么此语法就会含糊不清。

typename#methodname(paramtypes)

以参数显式指定的类型对 method 或构造函数的引用。这在交叉引用重载的 method 时很有用。例如：

```
@see InputStream#read(byte[], int, int)
```

#methodname

在当前类、接口或包含类、超类、当前类或接口的超接口的其中之一中，对非重载的 method 或构造函数的引用，使用这种简洁的形式来引用同一个类中的其他 method。例如：

```
@see #setBackgroundColor
```

Download at http://www.pin5i.com/

#methodname(paramtypes)

在当前类、接口或包含类、超类的其中之一中对 method 或构造函数的引用。此形式之所以能处理重载的method，是因为它显式地列出了method参数的类型。例如：

```
@see #setPosition(int, int)
```

typename#fieldname

对指定类中具名字段的引用。例如：

```
@see java.io.BufferedInputStream#buf
```

如果类型是在没有包名的情况下被指定，就会以如针对 *typename* 所描述的方式那样来解析。

#fieldname

对当前类型或包含类、超类或当前类型的超接口其中之一中的字段的调用。例如：

```
@see #x
```

包的 doc 注释

针对类、接口、method、构造函数以及字段的说明文件注释，会出现在Java源代码中它们所要说明的特性的定义前面。*javadoc* 也可以读取并显示包的概括性说明文件。由于包被定义在目录中而不是在单一的源代码文件中，所以*javadoc* 会在包含此包的类的源代码的目录中，寻找名为*package.html* 的文件，其中有包的说明文件。

package.html 文件应该包含针对包的简单 HTML 说明文件。它也可以包含 `@see`、`@link`、`@deprecated`以及 `@since`标记。由于*package.html* 不是 Java 源代码的文件，所以它所包含的说明文件应该是 HTML 且不会是 Java 注释（亦即它不应该被包括在 `/**` 和 `*/` 字符中）。最后，所有出现在*package.html* 中的 `@see` 和 `@link` 都必须使用完全限定的类名。

除了定义针对每个包的*package.html*文件之外，你也可以通过在那些包的源码树（source tree）中定义一个 *overview.html* 文件来对一组包提供高层次的说明文件。当*javadoc* 在该源码树上运行时，就会使用 *overview.html* 作为它所显示的最高层次概观。

JavaBeans 惯例

JavaBeans 是用来定义可重用模块化软件组件的结构。JavaBeans 声明包含了以下 bean 的定义："一个具有可重用性的软件组件，它可以在程序建立工具（builder tool）中以可视化的方式来操作"。就如你所看到的，这是个相当宽松的定义，bean 可以有多种形式。bean 最常见的用处就是用于图形用户界面组件，例如 `java.awt` 和 `javax.swing` 包

Download at http://www.pin5i.com/

的组件。这些在《Java Foundation Classes in a Nutshell》和《Java Swing》中说明，这两本书都是由O'Reilly出版。虽然所有的bean都可以用可视化的方式来操作，但这并不代表每个bean都有自己的可视化表示方式。例如，javax.sql.RowSet类（说明于O'Reilly的《Java Enterprise in a Nutshell》中）是JavaBeans组件，它代表由数据库查询所产生的数据。其实对JavaBeans组件的简单性和复杂性上并没有任何限制。最简单的bean通常是基本的图形界面组件，例如java.awt.Button对象。但即使复杂的系统，例如具可嵌入性的电子数据表程序包，也可以作为一个单独的bean运行。

JavaBeans组件模型是由java.beans、java.beans.beancontext包以及一些重要的命名和API惯例所组成，那些是bean和bean操作工具必须遵守的规范。这些惯例不是JavaBeans API本身的一部分，但对bean开发者来说，它们在许多方面都比API本身更为重要。这些惯例有时被称为设计模式（design pattern），它们指定了像method名称和由bean定义的特性访问（property accessor）method的签名这样的内容。如果你所编写的类不是要成为bean，以适合在程序建立工具中做可视化的操作，那么就不必遵循这些惯例。但是，JavaBeans惯例已被广泛使用并理解，你可以通过遵循相关的惯例来改善你的程序代码的可用性和重用性，尤其在property accessor method的命名惯例上更是如此。

我们会在本节的稍后说明这些惯例本身。但是，先浏览JavaBeans模型是十分必要的。

Bean基础

任何符合特定基本规则的对象都可以是bean；所有的bean都不必是Bean类的子类。有许多的bean是GUI组件，编写不会出现在屏幕上的“不可见”bean也是相当有可能的而且通常很有用（但是，在完成的应用程序中不会出现在屏幕上的bean并不代表就不能用beanbox工具来做可视化的操作）。

bean可以由它导出的属性（property）、事件（event）以及method来描绘。property是bean的一个内部状态，可用程序来设定并查询，这通常会通过标准的get和set accessor method来进行。

bean会通过产生事件来与它所嵌入的应用程序以及其他bean沟通。JavaBeans API使用与AWT和Swing组件所使用的相同的事件模型。此模型是以java.util.EventObject类和java.util.EventListener接口为基础，这部分在《Java Foundation Classes in a Nutshell》（O'Reilly）中有详细的说明。简言之，事件模型的运作就像这样：

- bean在定义事件时，会提供add和remove method以针对那个事件的监听对象（listener object）进行注册和注销监听。

Download at http://www.pin5i.com/

- 应用程序如果要在那个类型的事件发生时被通知，可以使用这些method来注册相应类型的事件监听对象。
- 当事件发生时，bean会通过传递描述此事件的事件对象给由事件监听接口所定义的method，来通知所有已注册的监听程序。

单点传播（unicast）事件是相当罕见的事件种类，在这样的情况下只有单一已注册的监听对象。如果尝试注册多于一个的监听程序，单点传播事件的add注册method就会抛出TooManyListenersException。

由bean导出的method只是由bean所定义的公共method，其中不包括取得并设定属性值和注册与注销事件监听程序的那些method。

除了先前所描述的常规属性之外，JavaBeans API也支持好几种专门的属性子类型。变址属性（indexed property）就是具有数组值的属性，还包括能访问数组元素和整个数组的读取函数（getter）与设定函数（setter）的method。约束属性（bound property）会在每当属性的值改变时，就发送PropertyChangeEvent给任何有兴趣的PropertyChangeListener对象。限定属性（constrained property）的任何改变都可以被任何感兴趣的监听程序所否决。当bean的限定属性的值改变时，bean必须发送PropertyChangeEvent给所有感兴趣的VetoableChangeListener对象。如果有任一个对象抛出PropertyVetoException，则此属性值就不会被更改，而PropertyVetoException会被传播回属性的setter method。

bean类

bean类本身必须遵循以下惯例：

类名

在bean的类名上没有任何限制。

超类

bean可以扩展任何其他的类。bean常是AWT或Swing组件，但这里并没有什么限制。

实例化

bean应该提供不带参数的构造函数，以让bean操作工具可以轻易地实例化bean。

属性（property）

假设bean具有遵循这些模式的accessor method，它定义了类型T的属性p（如果T是boolean类型的，那么专门形式的getter method是被允许的）：

Download at http://www.pin5i.com/

Getter

```
public T getP()
```

Boolean getter

```
public boolean isP()
```

Setter

```
public void setP(T)
```

Exception

属性 accessor method 可以抛出任何类型的已校验或未校验的异常。

变址（indexed）属性

变址属性是属于数组类型的属性，它提供了能读取与设置整个数组的accessor method以及能读取和设置每个单独数组元素的method。如果bean定义了以下的accessor method，就是定义了属于类型 `T[]`的变址属性 `p`：

Array getter

```
public T[] getP()
```

Element getter

```
public T getP(int)
```

Array setter

```
public void setP(T[])
```

Element setter

```
public void setP(int,T)
```

Exception

变址属性 accessor method 可以抛出任何类型的已校验或未校验的异常。如果所提供的地址超出界限，它们就应该抛出 `ArrayIndexOutOfBoundsException`。

约束（bound）属性

约束属性就是在它的值改变时，它会产生 `PropertyChangeEvent` 的属性。以下是约束属性的惯例：

Accessor method

约束属性的 getter 和 setter method 遵循与一般属性相同的惯例。

Download at http://www.pin5i.com/

监听程序的注册

定义了一或多个约束属特性的bean必须定义一对method以供监听程序的注册，这些监听程序会在任意一个约束属性值改变时被通知。那些method必须具有这些签名：

```
public void addPropertyChangeListener(PropertyChangeListener)
public void removePropertyChangeListener(PropertyChangeListener)
```

具名的属性监听程序的注册

bean 可以选择性地提供额外的method，让事件监听程序能针对单独约束属性值的改变来注册。属性的名称会被传递给这些 method 且 method 具有以下签名：

```
public void addPropertyChangeListener(String, PropertyChangeListener)
public void removePropertyChangeListener(String, PropertyChangeListener)
```

单属性（per-property）监听程序的注册

bean 可以选择性地提供额外的专属于单独属性的事件监听程序注册method。对于属性 *p*，这些 method 具有以下签名：

```
public void addPListener(PropertyChangeListener)
public void removePListener(PropertyChangeListener)
```

这种类型的 method 允许用 beanbox 来区分约束属性与非约束属性。

通知（notification）

当约束属性的值改变时，bean 就应该更新它的内部状态以反映此变化，然后传递 `PropertyChangeEvent` 给每个注册这个 bean 或专门的约束属性的 `PropertyChangeListener` 对象的 `propertyChange()` method。

支持

`java.beans.PropertyChangeSupport` 对于实现约束属性是很有帮助的。

限定属性

限定属性的任何改变都可以由已注册的监听程序否决。大部分的限定属性也是约束属性。以下是限定属性的惯例：

Getter

限定属性的 getter method 与一般属性的 getter method 相同。

Setter

如果属性的改变被否决，限定属性的 setter method 就会抛出 `PropertyVetoException`。对于属于类型 *T* 的属性 *p*，签名看起来就像这样：

```
public void setP(T) throws PropertyVetoException
```

Download at http://www.pin5i.com/

监听程序的注册

定义了一个或多个限定属性的bean必须定义一对method以供监听程序注册，这些监听程序会在任一个限定属性值改变时被通知。那些method必须具有这些签名：

```
public void addVetoableChangeListener(VetoableChangeListener)
public void removeVetoableChangeListener(VetoableChangeListener)
```

具名的属性监听程序的注册

bean可以选择性地提供额外的method，让事件监听程序能针对单独限定属性值的改变来注册。属性的名称会被传递给这些method且method具有以下签名：

```
public void addVetoableChangeListener(String, VetoableChangeListener)
public void removeVetoableChangeListener(String, VetoableChangeListener)
```

单属性监听程序的注册

bean可以选择性地提供额外的专属于单独限定属性的监听程序注册method。对于属性 *p*，这些method具有以下签名：

```
public void addPListener(VetoableChangeListener)
public void removePListener(VetoableChangeListener)
```

通知

当限定属性的setter method被调用时，bean必须产生描述了所请求的改变的PropertyChangeEvent，并把那个事件传递给每个对bean或专门限定属性注册的VetoableChangeListener对象的vetoableChange() method。如果任一个监听程序通过抛出PropertyVetoException来否决改变，bean就必须发出另一个PropertyChangeEvent将属性恢复至原先的值，然后它自己应该再抛出PropertyVetoException。另一方面，如果属性的改变未被否决，bean就应该更新它的内部状态以反映此变化。如果此限定属性也是约束属性，bean就应该在这时通知PropertyChangeListener对象。

支持

java.beans.VetoableChangeSupport对于实现限定属性很有帮助。

事件

当约束和限定属性被改变时，除了会产生PropertyChangeEvent事件之外，bean还可以产生其他类型的事件。名为 *E* 的事件应该遵循下面这些惯例：

事件类

事件类应该直接或间接扩展java.util.EventObject，而且应该被命名为EEvent。

Download at http://www.pin5i.com/

监听程序接口

事件必须与扩展了java.util.EventListener的事件监听程序接口相关联，并且被命名为ElistenerEvent。

监听程序method

事件监听程序接口可以定义任意数量的method，这些method会接收属于EEvent类型的单一自变量并返回void。

监听程序的注册

bean必须定义一对method以供想在E事件发生时被通知的事件监听程序注册。那些method应具有以下签名：

```
public void addEListener(EListener)
public void removeEListener(EListener)
```

单点传播事件

单点传播事件一次只允许一个监听程序对象被注册。如果E是单点传播事件，那么监听程序的注册method就应该具有这个签名：

```
public void addEListener(EListener) throws TooManyListenersException
```

Download at http://www.pin5i.com/

第八章

Java 开发工具

Sun 的 Java 实现包含了一些针对 Java 开发者的工具。当然，其中主要的是 Java 解释器和 Java 编译器，也还有一些其他的工具。本章说明与 JDK 一起的大部分工具。值得注意的是，本章省略了专门针对企业程序设计的 RMI 和 IDL 工具，这两个工具在《Java Enterprise in a Nutshell》（O'Reilly）中有说明。

此处所说明的工具是Sun开发工具的一部分，它们是实现的细节，而不是Java规范本身的一部分。如果你使用 Sun JDK 之外的 Java 开发环境，就应该查阅厂商提供的工具说明文件。

本章中的一些范例对文件和路径的分隔字符使用了 Unix 惯例。如果你的开发平台是 Windows，那么就把文件名中的斜线改为反斜线，并把路径指定语句中的冒号改为分号。

---

apt　　**注释处理工具**（Annotation Processing Tool）

提纲（synopsis）

```
apt [options] sourcefiles
```

说明

apt 会读取并解析指定的 *sourcefiles*。它所发现的所有注释都会被传递给适当的注释处理对象（processor factory object），此对象使用注释来产生以注释内容为基础的辅助源代码或数据文件，然后 *apt* 会编译 *sourcefiles* 并产生文件。

annotation processor 类和 factory 类是用 `com.sun.mirror.apt` API 以及其他 `com.sun.mirror` 的子包来定义的。

Download at http://www.pin5i.com/

选项

*apt*与*javac*共享了几个选项。如果命令行自变量是以@开始，*apt*就会把它当成文件，并从那个被指定的文件中读取选项和源代码文件。相关信息请参阅*javac*。

`-Aname=value`

将`name=value`对当作自变量传递给注释处理器。

`-cp path`
`-classpath path`

设定classpath。请参阅*javac*。

`-d dir`

用来放置类文件的目录。请参阅*javac*。

`-factory classname`

明确地指定所要使用的注释处理factory的类名。

`-factorypath path`

指定注释处理factory所要搜索的路径，而不搜索classpath。

`-help`

输出辅助说明并退出。

`-nocompile`

告诉*apt*不要编译`sourcefiles`或任何已产生的文件。

`-print`

只解析指定的`sourcefiles`并输出它们所定义的类型的提纲。不处理注释或编译任何的文件。

`-s dir`

指定用来存放产生的源文件的根目录。

`-source version`

指定所接受的语言版本。请参阅*javac*。

`-version`

输出*apt*版本信息。

`-X`

显示有关非标准选项的信息。

另可参阅 *javac*（第四章）

Download at http://www.pin5i.com/

extcheck　　JAR 版本冲突检查公用程序

提纲

```
extcheck [-verbose] jarfile
```

说明

extcheck 会检查被包含于指定 *jarfile* 中的扩展包（或那个扩展包的较新版本）是否已安装在系统上。它通过从被指定的 *jarfile* 和在系统扩展包目录中找到的所有 JAR 文件读取 Specification-Title 和 Specification-Version 清单属性（manifest attribute）来实现。

extcheck 是被设计用于自动化安装命令脚本中。如果没有加上 -verbose 选项，它就不会输出检查的结果，而是会在指定的扩展包没有与任何已安装的扩展包冲突而且可以安全地被安装时，就将退出代码（exit code）设定为 0。如果具有相同名称的扩展包已被安装而且版本号等于或大于指定文件的版本号，它就会把退出代码设定为非零值。

选项

-verbose

输出检查到的已安装扩展包并显示检查的结果。

另可参阅　*jar*

jarsigner　　JAR 签名与验证工具

提纲

```
jarsigner [options] jarfile signer
jarsigner -verify jarfile
```

说明

jarsigner 会将数字签名加到指定的 *jarfile*，而且如果指定了 -verify 选项，它就会验证已添加至 JAR 文件的数字签名。所指定的 *signer* 是不区分大小写的昵名或别名，那是签名被拿来使用的项目的。所指定的 *signer* 名称会被用来查询能产生签名的私用密钥。

当你将你的数字签名用在 JAR 文件上时，实际上就是为文件的内容作了担保，你承诺 JAR 文件中只包含无恶意的程序代码、没有违反著作权的文件等。当你验证以数字方式签名的 JAR 文件时，可以判定谁是文件的签名者，并（如果验证成功）判定 JAR 文件的

Download at http://www.pin5i.com/

内容自从加上签名后是否有更改、毁损或篡改。验证数字签名与决定是否要相信你所验证的签名所属的人或组织是完全不同的。

*jarsigner*以及相关的*keytool*程序取代了 Java 1.1 中的*javakey*程序。

选项

*jarsigner*定义了一些选项，其中有许多选项指定了找到特定签名者的密钥的方法。当使用 `-verify` 选项来验证有签名的 JAR 文件时，其实这些选项大部分都是不必要的：

`-certs`

如果此选项有与 `-verify` 或 `-verbose` 选项一起被指定，就会使*jarsigner*显示和签名的 JAR 文件有关的公钥证书的细节。

`-Jjavaoption`

将指定的 *javaoption* 直接传递给 Java 解释器。

`-keypass` *password*

指定用来加密特定 *signer* 的密钥的密码。如果没有指定此选项，*jarsigner*就会提示你要输入密码。

`-keystore` *url*

keystore 是个包含了密钥和证书的文件。此选项指定了 *keystore* 的文件名或 URL，并被指定的 *signer* 的私用和公用密钥都会在那里被查询。默认值是用户根目录（系统属性*user.home*的值）中名称为*.keystore*的文件。这也是由*keytool*所管理的 *keystore* 的默认位置。

`-sigfile` *basename*

指定了 *.SF* 和 *.DSA* 文件的基本名称（base name），它们会被添加到 JAR 文件的 *META-INF/*目录。如果没有指定这个选项，基本文件名就会依 *signer* 名称来决定。

`-signedjar` *outputfile*

指定由*jarsigner*所创建的已签名 JAR 文件的名称。如果未指定此选项，*jarsigner* 就会改写命令行上指定的 *jarfile*。

`-storepass` *password*

指定用来验证*keystore*完整性的密码（但不对私有密钥做加密）。如果省略此选项，*jarsigner* 就会提示你输入密码。

`-storetype` *type*

指定由 *-keystore* 选项所指定的 *keystore* 的类型。默认值为系统默认的 *keystore* 类型，在大部分的系统上是 Java Keystore 类型，也就是所谓的 `JKS`。如果你已安装了 Java Cryptography Extension，或许会想要用 `JCEKS` *keystore* 来表示。

Download at http://www.pin5i.com/

-verbose

显示关于签名或验证过程的额外信息。

-verify

指定 *jarsigner* 应该验证加上签名的 JAR 文件，而不是对它签名。

另可参阅 *jar*、*keytool*、*javakey*

jar　　Java 归档工具

提纲

```
jar c|t|u|x[f][m][M][0][v] [jar-file] [manifest] [-C directory] [input-files]
jar i [jar-file]
```

说明

jar 是个工具，它可以创建并操作 JavaArchive（JAR）文件。JAR 文件是个含有 Java 类文件、那些类所需要的辅助资源文件以及选择性的超元信息（meta-information）的 ZIP 文件。此 meta-information 包含列出 JAR 归档文件内容的清单文件，并提供关于每个文件的辅助信息。

jar 命令可以创建 JAR 文件、列出 JAR 文件的内容以及提取 JAR 归档文件中的文件。在 Java 1.2 及之后的版本中，它也可以增加文件到已有的归档文件，或更新归档文件的清单文件。在 Java 1.3 及之后的版本中，*jar* 也可以给 JAR 文件增加索引项。

jar 命令的语法会让人联想到 Unix *tar*（tape archive）命令。*jar* 的大部分选项都是被指定为一块串接的字母，它们被当成单一自变量而不是单独的命令行自变量来传递。第一个自变量的第一个字母指定了 *jar* 所要执行的动作，这项是必要的，其他的字母是选择性的。各种文件自变量是取决于被指定的那些字母。

就和在 *javac* 中一样，任何以 @ 开头的命令行自变量都会被当成是文件的名称，其中包含一些操作或文件名。

命令选项

jar 的第一个选项的第一个字母指定了 *jar* 所要执行的基本操作。可用的选项有：

c

创建新的 JAR 归档文件。有个输入文件和/或目录列表必须要被指定为 *jar* 的最后一个自变量。新创建的 JAR 文件会将 *META-INF/MANIFEST.MF* 文件作为它的第一项。此自动创建的清单会列出 JAR 文件的内容，并包含每个文件的消息摘要。

Download at http://www.pin5i.com/

i

将此JAR文件的内容和它在Class-Path清单属性中所引用的所有JAR文件的内容加上索引。最后产生的索引会被存储为JAR文件中的*META-INF/INDEX.LIST*，而且可以被Java解释器或applet viewer用来优化其类和资源的查找算法，并避免下载不必要的JAR文件。此i选项后面必须跟着要被索引的JAR文件名称，其他的选项都不被允许。Java 1.3及之后的版本提供。

t

列出JAR归档文件的内容。

u

更新JAR归档文件的内容。所有列在命令行上的文件都会被加到归档文件中。当与m选项一起使用时，这会向JAR文件增加指定的清单信息。Java 1.2及之后的版本提供。

x

提取JAR归档文件的内容。命令行上所指定的文件和目录会被提取并被创建在当前的工作目录中。如果没有指定文件或目录名称，那么所有JAR文件中的文件和目录都会被提取。

修饰符选项

这四个命令中的每一个指定了字母后面可以接着额外的字母，提供关于所要执行的操作的进一步细节：

f

指出*jar*要操作名称被指定在命令行上的JAR文件。如果此选项不存在，*jar*就会从标准输入中读取JAR文件，并/或将JAR文件写入标准输出。如果f选项存在，命令行就必须包含所要操作的JAR文件的名称。

m

当*jar*创建或更新JAR文件时，它会自动创建（或更新）JAR归档文件中名为*META-INF/MANIFEST.MF*的清单文件。此默认清单只是列出JAR文件的内容。有许多的JAR文件会要求清单中要指定额外的信息，m选项告诉*jar*命令在命令行中行指定了清单模板。*jar*会读取此清单文件，并把它所包含的所有信息存储在它所创建的*META-INF/MANIFEST.MF*文件中。

M

配合c和u命令一起使用，用来告诉*jar*不要创建默认的清单文件。

Download at http://www.pin5i.com/

v

告诉*jar*产生详细的输出。

0

配合c和u命令一起使用，用来告诉*jar*把文件存储在JAR归档文件中而不加以压缩。请注意，此选项是数字0，而不是字母O。

文件

*jar*的第一个选项是由初始的命令字母和各种选项字母所组成。第一个选项后面接着一个文件列表：

jar-file

如果第一个选项包含字母f，那个选项就必须接着所要创建或操作的JAR文件的名称。

manifest-file

如果第一个选项包含字母m，那个选项就必须接着文件名称，其中含有清单信息。如果第一个选项同时包含字母f和m，则JAR和清单文件就应该以与f和m选项相同的出现顺序列出。*jar*会自动创建JAR文件的清单，除非指定了M选项。当*manifest-file*与m选项一起被指定时，除了自动产生的项之外，还应该包含要被放在清单中的额外清单项。

files

要被插入JAR归档文件或从JAR归档文件中提取出来的一个或多个文件和/或目录的列表。

额外选项

除了前面列出的所有选项之外，*jar*也支持以下选项：

-C *dir*

用在要被处理的文件列表中，它告诉*jar*在处理后续的文件和目录时要更改至指定的*dir*。后续的文件和目录名称都是相对于*dir*来解释的，并且会在不加上*dir*作为前缀的情况下被插入JAR归档文件中。-C选项的数量不受限制，各个-C选项的功效会持续到遇到下一个-C选项。由-C选项指定的目录会以相对于当前工作目录的方式来解释，而不是由前一个-C选项所指定的目录来解释。Java 1.2及之后的版本提供。

-J*javaopt*

将选项*javaopt*传递至Java解释器。

Download at http://www.pin5i.com/

范例

*jar*命令具有一组容易令人混乱的选项，但在大部分情况下，它的用法相当简单。如果要创建一个简单的JAR文件，它含有当前目录中所有的类文件和名称为*images*的子目录中的所有文件，你可以输入：

```
% jar cf my.jar *.class images
```

如果要详细地列出JAR归档文件的内容：

```
% jar tvf your.jar
```

如果要提取JAR文件的清单文件以供查看或编辑：

```
% jar xf the.jar META-INF/MANIFEST.MF
```

如果要更新JAR文件的清单：

```
% jar ufm my.jar manifest.template
```

另可参阅 *jarsigner*

---

java Java解释器

提纲

```
java [ interpreter-options ] classname [ program-arguments ]
java [ interpreter-options ] -jar jarfile [ program-arguments ]
```

说明

*java*是Java字节码解释器，它会运行Java程序。要被运行的程序是由`classname`所指定的类。这必须是完全限定名称：它必须包含类的包名，但不能有*.class*扩展名。例如：

```
% java david.games.Checkers
% java Test
```

被指定的类必须定义与以下签名完全相同的`main()` method：

```
public static void main(String[] args)
```

此method是用来作为程序进入点：解释器会在此处开始执行。

在Java 1.2及之后的版本中，程序可以被封装在可执行的JAR文件中。要运行以这种方法封装的程序，可以使用`-jar`选项来指定JAR文件。可执行的JAR文件的清单必须包含`Main-Class`属性，它指定了Jar文件中哪个类包含了`main()` method，解释器会在那里开始执行。

Download at http://www.pin5i.com/

所有出现在要被执行的类或JAR文件前面的命令行选项都是针对Java解释器本身的选项，所有接在类名或JAR文件名后面的选项都是针对程序的选项。它们会被Java解释器忽略并被当成字符串数组传递给此程序的main() method。

Java解释器会运行到main() method后结束，所有由此程序所创建的线程（除了被标识为后台线程（daemon thread）的那些线程）也都会结束。

解释器版本

*java*程序是基本版本的Java解释器。但是，除了这个程序之外，还有几种其他版本的Java解释器。这些版本都与Java类似，但具有专门的功能。此列表包含了所有的解释器版本，其中包括那些不再被使用的。

java

这是基本版本的Java解释器。通常使用这个就行了。

javaw

这个版本的解释器只有Windows平台有。当你想运行Java程序（例如脚本）但又不要创建控制台窗口时，就使用*javaw*。

客户端或服务器端VM

Sun的“热点（HotSpot）”虚拟机有两种版本：一种是供暂时存在的客户端应用程序使用，而另一种是供长时间运行的服务器程序代码使用。在Java 1.4中，你可以用-server选项来选择服务器版本的VM，可以用-client选项来指定客户端VM（这是默认值）。在Java 5.0中，如果解释器检测到它是运行于“服务器等级”的硬件上（通常是具有多个CPU的计算机），就会自动进入服务器模式。

原有的解释器版本

oldjava

这个版本的解释器被包含在Java 1.2和Java 1.3中，这是为了保证对Java 1.1解释器的兼容性。它用Java 1.1的类载入模式（class-loading scheme）来载入类。很少有Java应用程序会必须使用这个版本的解释器，而它在Java 1.4中已被撤除。

oldjavaw

在Java 1.2和1.3中，这个版本的解释器只有Windows平台才有，它结合了*oldjava*和*javaw*的特性。

java_g

在Java 1.0和Java 1.1中，*java_g*是调试版本的Java解释器，它包含了一些专门的命令行选项。Windows平台也有个*javaw_g*程序。*java_g*没有被包含在Java 1.2及之后的版本中。

Download at http://www.pin5i.com/

标准 VM

在 Java 1.3 中，你可以用 -classic 选项来指定你要使用“标准 VM”（在本质上与 Java 1.2 VM 相同）来代替 HotSpot VM（它使用了渐增式编译（incremental compilation））。这个选项在 Java 1.4 中已被移除。

实时编译器（Just-in-time compiler）

在 Java 1.2 和 Java 1.3 中，当你指定 -classic 选项时，Java 解释器会使用实时编译器（如果你的平台提供）。JIT 会在运行时把 Java 字节码转换为原生机器指令，并大幅加速典型 Java 程序的运行。如果你不想使用 JIT，可以通过将 JAVA_COMPILER 环境变量设定为“NONE”，或使用 -D 选项将 java.compiler 系统属性设定为“NONE”来停用 JIT。

```
% setenv JAVA_COMPILER NONE                          // Unix csh语法
% java -Djava.compiler=NONE MyProgram
```

如果你想使用不同的 JIT 编译器实现，可以把环境变量或系统属性设定为所要实现的名称。此环境变量和属性在 Java 1.4 中已不再被使用，Java 1.4 使用了含有高效率的 JIT 技术的 HotSpot VM。

线程系统

在 Solaris 和相关的 Unix 平台上，你可以选择由 Java 1.2 解释器和 Java 1.3 的“标准 VM”所使用的线程类型。如果要使用原生 OS 线程，可以指定 -native；如果要使用非原生线程（或 green thread，此为默认值），可以指定 -green。在 Java 1.3 中，默认的“客户端 VM”使用了原生线程。在 Java 1.3 中指定了 -green 或 -native 也就暗含指定了 -classic。这些选项在 Java 1.4 中已不再（或不需要）被支持。

常用选项

以下是最常被使用的选项。

-classpath *path*

指定在尝试加载类时 Java 所要搜索的目录和 JAR 文件。在 Java 1.2 及之后的版本中，此选项只会指定应用程序类的位置。在 Java 1.0 和 1.1 中，如果使用 *oldjava* 解释器，此选项会指定系统类、扩展类以及应用程序类的位置。

-cp

-classpath 的同义词。Java 1.2 及之后的版本提供。

-*Dpropertyname=value*

将系统属性列表中的 *propertyname* 定义为等于 *value*。你的 Java 程序可以通过属性名称来查询被指定的值。你可以指定任意数量的 -D 选项。例如：

Download at http://www.pin5i.com/


```
% java -Dawt.button.color=gray -Dmy.class.pointsize=14 my.class
```

-fullversion

列出完整的Java版本字符串并退出，其中包括编译编号（build number）。可与 *-version* 作比较。

-help, -?

输出使用信息并退出。请参阅 -X。

-jar *jarfile*

运行指定的可执行文件 *jarfile*。被指定的 *jarfile* 的清单必须包含一个 Main-Class 属性，以识别出具有 main() method的类，程序会在那里开始运行。Java 1.2 以及之后的版本提供。

-showversion

功能类似 -version 选项，除了解释器会在输出版本信息之后继续运行。Java 1.3 及之后的版本提供。

-version

输出 Java 解释器的版本并退出。

-X

显示非标准解释器选项（那些会以 -X 开始）的使用信息并退出。Java 1.2 及之后的版本提供。

-Xbootclasspath:*path*

指定由目录、ZIP文件和JAR文件所组成的搜索路径，*java* 解释器应该要用它来查询系统类。此选项的使用非常罕见。Java 1.2 及之后的版本提供。

-Xbootclasspath/a:*path*

将指定的 *path* 添加到系统 classpath。Java 1.3 及之后的版本提供。

-Xbootclasspath/p:*path*

将指定的 *path* 默认为系统启动 classpath。Java 1.3 及之后的版本提供。

Assertion 选项

以下选项指定assertion是否被测试以及在何处被测试。这些选项是在Java 1.4中加入的。

-disableassertions[:*where*]

停用 assertion。它是 Java 1.4 中新出现的，而且可以缩写为 -da。在单独使用时，它会停用所有的 assertion（除了在系统类中的那些），这是默认值。如果要停用单一类中的assertion，可以在选项后面加上冒号及完全限定类名。如果要停用整个包

Download at http://www.pin5i.com/

（以及它所有的子包）中的assertion，可以在此选项后面加上冒号、包的名称以及三个点号。请参阅 -enableassertions 和 -disablesystemassertions。

-da[:*where*]

停用 assertion。请参阅 -disableassertions。

-disablesystemassertions

停用所有系统类中的 assertion（这是默认值）。它可以被缩写为 -dsa 而且没有任何选项。

-dsa

-disablesystemassertions 的缩写。

-enableassertions[:*where*]

启用 assertion。此选项可被缩写为 -ea。在单独使用时，它会启用所有的assertion（除了在系统类中的）。如果要启用单一类中的 assertion，可以在选项后面加上冒号和完整类名。如果要启用整个包（及其子包）中的 assertion，可以在选项后面加上冒号、包名以及三个点号。请参阅 -disableassertions 和 -enablesystemassertions。

-ea[:*where*]

启用 assertion。这是 -enableassertions 的缩写。

-enablesystemassertions

启用所有系统类中的 assertion。可被缩写为 -esa。

-esa

-enablesystemassertions 的缩写。

性能调整选项

以下选项会选择要运行哪个版本的VM，并微调其内存配置、内存回收以及渐增式编译。以 -X 开始的选项是非标准的，可能会因版本不同而异。

-classic

运行“标准 VM”来代替默认的高性能“客户端 VM”。只有 Java 1.3 才提供。

-client

针对典型的客户端应用程序来优化HotSpot VM的渐增式编译。此选项通常会延迟一些编译，以支持较快速的应用程序启动时间。Java 1.4 及之后的版本提供。请参阅 -server 选项。

Download at http://www.pin5i.com/

-d32

以 32 位模式运行。此选项在 Java 1.4 及之后的版本中是有效的，但现在只有针对 Solaris 平台实现。

-d64

以 64 位模式运行。此选项在 Java 1.4 及之后的版本中是有效的，但现在只有针对 Solaris 平台实现。

-green

在像 Solaris 和 Linux 这种支持多种线程风格的操作系统上选择非原生（或 green）线程。这是 Java 1.2 的默认值。在 Java 1.3 中，可使用这个选项也可使用 -classic 选项。请参阅 -native。只有 Java 1.2 和 1.3 提供。

-native

在像 Solaris 和 Linux 这种支持多种线程风格的操作系统上选择原生线程，而不是默认的 green 线程。在某些环境中（例如在多 CPU 的计算机上运行时）使用原生线程很有好处。在 Java 1.3 中，默认的 HotSpot 虚拟机使用了原生线程。只有 Java 1.2 和 1.3 提供。

-server

针对服务器等级的应用程序来优化 VM 的渐增式编译。通常，此选项会造成较长的启动时间，但会有较好的后续性能。Java 1.4 及之后的版本提供。在 Java 5.0 及之后的版本中，如果是在“服务器等级”的硬件上运行（例如双 CPU 的机器），则有许多 VM 会自动选择此选项。请参阅 -client。

-Xbatch

告诉 HotSpot VM 要在前台执行所有的实时编译，而不考虑编译所需的时间。如果没有加上这个选项，VM 就会在前台作解释时在后台编译这些 method。Java 1.3 及之后的版本提供。

-Xincgc

使用渐增式的内存回收。在此模式中，内存回收程序会在后台连续运行，而运行中的程序很少会在内存回收发生时出现明显的停顿。但是，使用此选项通常会造成整体性能降低 10%。Java 1.3 及之后的版本提供。

-Xint

告诉 HotSpot VM 只以解释模式运行，而不执行任何实时的编译。Java 1.3 及之后的版本提供。

-Xmixed

告诉 HotSpot VM 在常被使用的 method（“hotspots”）上执行实时编译，而以解释

Download at http://www.pin5i.com/

模式执行其他的 method。这是默认的行为。可与 -Xbatch 和 Xint 作比较。Java 1.3 及之后的版本提供。

-Xms *initmem*[k|m]

指定当解释器启动时，要分配多少内存给堆。依默认，*initmem*会以字节为单位来指定。你可以通过添加字母k来以千字节指定，或是添加字母M来以兆字节指定。默认值是 2 MB。对于大型或对内存需要较多的应用程序（例如Java编译器），你可以通过启动具有较大数量内存的解释器来改善运行时性能。初始堆的大小至少必须指定为 1 MB。Java 1.2 及之后的版本提供。在 Java 1.2 之前，要使用 -ms。

-Xmx*maxmem*[k|m]

指定针对动态分配的对象和数组，解释器所能使用的最大堆的大小。*maxmem*根据默认是以字节为单位来指定。你可以通过添加字母k来以千字节指定*maxmem*，或是通过添加字母M来以兆字节指定*maxmem*。默认值为64 MB。堆的大小无法被指定为小于 2 MB。Java 1.2 及之后的版本提供。在 Java 1.2 之前，要使用 -mx。

-Xnoclassgc

不要对类进行内存回收。Java 1.2 及之后的版本提供。在 Java 1.1 中要使用 -noclassgc。

-Xss*size*[k|m]

以字节、千字节或兆字节来设定线程堆的大小。Java 1.3 及之后的版本提供。

Instrumentation 选项

以下选项支持调试、剖析（profiling）以及其他的 VM。以 X 开头的选项是非标准的，而且可能会随版本而异。

-agentlib:*agent*[=*options*]

这是 Java 5.0 中新出现的，此选项指定了要与解释器一起启动的 JVMTI 代理程序以及针对代理程序的选项。JVMTI就是Java虚拟机工具接口（Java Virtual Machine Tool Interface），并为未来版本中的JVMDI 和JVMPI（调试与剖析接口）打基础。这代表一般的 -agentlib 选项会取代工具所特有的选项（例如 -Xdebug 和 -Xrunhprof）。范例：

```
% java -agentlib:hprof=help
% java -agentlib:jdwp=help
```

-agentpath:*path-to-agent*[=*options*]

就和 -agentlib 一样，但具有针对代理程序函数库的明确指定路径。Java 5.0 及之后的版本提供。

Download at http://www.pin5i.com/

-debug

使*java*能以允许*jdb*调试程序将自己添加到解释器部分的方式来启动。在Java 1.2及之后的版本中，这个选项已被 -Xdebug取代。

-javaagent:*jarfile*[=*options*]

当解释器启动时加载Java语言instrumentation代理程序。被指定的*jarfile*必须要有包含了Agent-Class属性的清单。此属性必须命名一个类，其中包含了代理程序的premain() method。所有的选项都会与java.lang.instrument.Instrumentation对象一起被传递给这个premain() method。进一步的细节请参阅java.lang.instrument。

-verbose, -verbose:class

每当*java*加载类时就输出消息。在Java 1.2及之后的版本中，你可以用-verbose:class作为同义词。

-verbose:gc

当内存回收发生时就输出消息。Java 1.2及之后的版本提供。在Java 1.2之前，可以使用 -verbosegc。

-verbose:jni

当原生method被调用时就输出消息。Java 1.2及之后的版本提供。

-Xcheck:jni

当使用Java Native Interface功能时就执行额外的验证检查。Java 1.2及之后的版本提供。

-Xdebug

以允许调试程序与解释器通信的方式来启动解释器。Java 1.2及之后的版本提供。在Java 1.2之前，可以使用-debug。这在Java 5.0中已过时，要改为使用-agentlib选项。

-Xfuture

严格地检查所有被载入的类文件的格式。如果没有加上这个选项，*java*就会执行与Java 1.1所执行的相同检查。Java 1.2及之后的版本提供。

-Xloggc:*filename*

以加上时间戳的方式将内存回收事件记录到具名文件。

-Xprof

将概要输出以标准输出形式输出。Java 1.3及之后的版本提供。在Java 1.2中或当使用 -classic选项时，就使用 -Xrunhprof。在Java 1.2之前，要使用 -prof。

Download at http://www.pin5i.com/

-Xrunhprof:*suboptions*

开启CPU、堆或监控程序。*suboptions*是以逗号分隔、由*name=value*对组成的列表。可用-Xrunhprof:help来得知被支持的选项和值的列表。Java 1.2及之后的版本提供。这在Java 5.0中已过时，要改用-agentlib选项。

高级选项

Java解释器也支持少数以-XX开始的高级配置选项。这些选项依赖于版本和平台，而Sun的说明文件把它们描述成“不建议随便使用”。但是，如果你想微调实际环境的线程、内存配置、内存回收、信号处理或实时编译性能，或许就会对它们有兴趣。请访问*http://java.sun.com/docs/hotspot/*。

载入类

Java解释器知道要到哪里找出组成Java平台的系统类。在Java 1.2及之后的版本中，它也知道要到哪里找出安装在系统扩展包目录中的所有扩展包的类。但是，解释器必须被告知要到何处找出组成所要运行的应用系统的非系统类。

类文件被存储在对应它们的包名的目录中。例如，类com.davidflanagan.utils.Util是被存储在*com/davidflanagan/utils/Util.class*文件中。默认情况下，解释器会使用当前工作目录作为根目录并搜索在此目录中及目录下的所有类。

解释器也可以搜索ZIP和JAR文件中的类。如果要告诉解释器在哪里搜索类，可以指定*classpath*：这是由目录、ZIP和JAR归档文件所组成的列表。在寻找类时，解释器依各位置的指定顺序来逐一搜索。

指定classpath的最简单方法就是设定CLASSPATH环境变量，这非常像由Unix shell或Windows命令解释路径所使用的PATH变量。如果在Unix中要指定classpath，可以输入像这样的命令：

```
% setenv CLASSPATH .:~/myclasses:/usr/lib/javautils.jar:/usr/lib/javaapps
```

在Windows系统上，可以使用如下的命令：

```
C:\> set CLASSPATH=.;c:\myclasses;c:\javatools\classes.zip;d:\javaapps
```

请注意，Unix和Windows使用了不同的字符来分隔目录和路径元素。

你也可以用-classpath或-cp选项来对Java解释器指定classpath。以这些方式所指定的路径会覆盖由CLASSPATH环境变量所指定的路径。在Java 1.2及之后的版本中或

Download at http://www.pin5i.com/

在使用 *oldjava* 解释器时，这个选项指定了针对所有类的搜索路径，其中包括了系统类和扩展类。

另可参阅 *javac*、*jdb*

javac　　Java 编译器

提纲

```
javac [ options ] files
```

说明

javac 是 Java 编译器，它会将 Java 源代码（在 *.java* 文件中）编译为 Java 字节码（在 *.class* 文件中）。Java 编译器本身是用 Java 编写的。

javac 可以被传递任意数量的 Java 源文件（source file），这些文件的名称都必须以 *.java* 扩展名结束。*javac* 会为每个定义于源文件的类产生单独的 *.class* 类文件。每个源文件可以包含任意数量的类，虽然只有一个可以是公共的顶层类。源文件的名称（不含 *.java* 扩展名）必须与它所包含的公共类的名称相匹配。

在 Java 1.2 及之后的版本中，如果在命令行上指定的文件名是以字符 @ 开始，那个文件就不会被当成 Java 源文件，而是会被当成由编译器选项和 Java 源文件组成的列表。因此，如果你把针对特殊项目的 Java 源文件列表保留在名为 *project.list* 的文件中，就可以用以下命令一次编译所有文件：

```
% javac @project.list
```

如果要编译源文件，*javac* 就必须能找到所有用于源文件中的类的定义。它会在源文件和类文件这两种形式中寻找定义，自动编译所有不具有对应类文件或在最近编译过后又被修改的源文件。

常用选项

最常被使用的编译选项如下：

`-classpath` *path*

指定路径，供 *javac* 用来查询特定源代码中所引用的类。这个选项会覆盖所有由 CLASSPATH 环境变量所指定的路径。所指定的 *path* 是由目录、ZIP 文件以及 JAR 归档文件所组成的有序列表，在 Unix 系统上会以冒号分隔，在 Windows 系统上会

Download at http://www.pin5i.com/

以分号分隔。如果 -sourcepath 选项没有被设定，那么这个选项也会指定针对源文件的搜索路径。

-d *directory*

指定用来存储类文件的目录（或在哪个目录下）。默认情况下，*javac* 会把它产生的 *.class* 文件存储在与定义有那些类的 *.java* 文件相同的目录中。但是，如果 -d 选项被指定了，所指定的 *directory* 就会被当成类层次的根，而 *.class* 文件会被放在这个目录中或在其下的适当子目录中，这取决于类的包名。因此，有以下命令：

```
% javac -d /java/classes Checkers.java
```

如果 *Checkers.java* 文件没有 *package* 语句，文件 *Checkers.class* 就会被放在 */java/classes* 目录中。另一方面，如果源文件是在包中：

```
package com.davidflanagan.games;
```

.class 文件就会被存储在 */java/classes/com/davidflanagan/games* 中。当 -d 选项被指定时，*javac* 就会自动创建它所需要的目录来把它的类文件存储在适当的地方。

-encoding *encoding-name*

当字符编码名称与默认的平台编码不同时，可用此选项来指定源文件所使用的字符编码名称。

-g

告诉 *javac* 要对输出类文件加上行号、源文件以及局部变量信息，以供调试程序使用。默认情况下，*javac* 只会产生行号。

-g:none

告诉 *javac* 把调试信息加在输出类文件中。Java 1.2 及之后的版本提供。

-g:*keyword-list*

告诉 *javac* 输出调试信息的类型，这些是由以逗号分隔的 *keyword-list* 所指定的。有效的关键字是：source，它指定了源文件信息；lines，指定了行号信息；vars，指定了局部变量调试信息。Java 1.2 及之后的版本提供。

-help

输出选项列表。请参阅 -X。

-J*javaoption*

将自变量 *javaopotion* 直接传给 Java 解释器。例如：-J-Xmx32m。*javaoption* 不能包含空格；如果有多个自变量要传给解释器，可以使用多个 -J 选项。Java 1.1 及之后的版本提供。

Download at http://www.pin5i.com/

-source *release-number*

指定编写程序代码所用的Java版本。合法的*release-number*值是5、1.5、1.4和1.3。选项5和1.5是同义的而且是默认值：编译器接受所有Java 5.0语言特性。使用-source 1.4可以让编译器忽略Java 5.0语言特性，例如enum关键字。使用-source 1.3可以让编译器忽略Java 1.4引入的assert关键字。此选项在Java 1.4及之后的版本中提供。

-sourcepath *path*

指定目录、ZIP文件以及JAR归档文件列表，以供*javac*在寻找源文件时搜索。在这个源路径中找到的文件如果没有相对的类文件被找到或是源文件比类文件新，就会被编译。默认情况下，源文件会与类文件在同一个地方被搜索到。Java 1.2及之后的版本提供。

-verbose

告诉编译器显示关于进行中事务的信息。尤其是它会使*javac*列出所有它编译的源文件，其中包括没有出现在命令行上的文件。

-X

告诉*javac*编译器显示非标准选项（那些都会以-X开头）的使用信息。Java 1.2及之后的版本提供。

警告选项

以下选项控制了*javac*所产生的警告信息：

-deprecation

告诉*javac*每当使用到过时的API时就产生相应的警告信息。默认情况下，*javac*只会对每个使用了过时API的文件提出一条警告信息。Java 1.1及之后的版本提供。在Java 5.0中，这是-Xlint:deprecation的同义词。

-nowarn

告诉*javac*不要输出警告信息。但错误仍要报告。

-Xlint

启用所有与“lint”程序有关的建议警告。在使用这个选项时，以下所有详细说明的警告都是被建议的。

-Xlint:*warnings*

启用或停用以逗号分隔的具名警告类型列表。在使用这个选项时，可用的警告类型如下。具名警告可以通过在它前面加上减号来暂停：

Download at http://www.pin5i.com/

all

启用所有的 lint 警告。

deprecation

对使用过时的 API 提出警告。请参阅 -deprecation。

fallthrough

当 switch 语句中的一个 case "落到（falls through）" 下一个 case 时提出警告。请参阅 -Xswitchcheck。

finally

当 finally 子句无法正常结束时提出警告。

path

当命令行上某处所指定的路径目录不存在时提出警告。

serial

如果 Serializable 类不具有 serialVersionUID 字段就提出警告。

unchecked

对每个未校验的泛型的使用提供详细的警告信息。

-Xmaxerrors *num*

列出的错误不要多于 *num* 个。

-Xmaxwarns *num*

列出的警告不要多于 *num* 个。

-Xstdout filename

告诉*javac*将警告和错误信息传送到指定的文件，而不是将它们写入平台。Java 1.4 及之后的版本提供。

-Xswitchcheck

对 switch 语句中 "fall through" 的 case 子句提出警告。在 Java 5.0 中，要使用 -Xlint:fallthrough。

交叉编译选项

以下选项在使用*javac*来编译倾向于在不同 Java 版本下运行的类文件时很有用：

-bootclasspath *path*

指定*javac*用来查找系统类的路径。此选项没有指定用来运行编译器本身的系统类，只有编译器所读取的系统类。Java 1.2 及之后的版本提供。

Download at http://www.pin5i.com/

-endorseddirs *path*

覆盖目录以搜索已签署的标准 JAR 文件。

-extdirs path

指定一个目录列表以搜索标准扩展功能的 JAR 文件。Java 1.2 及之后的版本提供。

-target *version*

指定产生的类文件所要使用的类文件格式。*version* 可以是 1.1、1.2、1.3、1.4、1.5 或 5。选项 1.5 和 5 是同义的而且在 Java 5.0 中是默认值，除非指定了 -source 1.4，在这样的情况下，-target 1.4 就是默认值。使用这个标记可以设定类文件的版本编号，以让最后产生的类文件不被先前版本的 VM 执行。

-Xbootclasspath:*path*

-bootclasspath 的替代方案。

-Xbootclasspath/a:*path*

将指定的路径附加至 bootclasspath。Java 1.3 及之后的版本提供。

-Xbootclasspath/p:*path*

将指定的路径加在 bootclasspath 前面。

环境

CLASSPATH

指定由目录、ZIP 文件以及 JAR 归档文件所组成的有序列表，*javac* 应该要在其中寻找用户类文件和源文件。此变量可以由 -classpath 选项覆盖。

另可参阅

java、*jdb*

---

javadoc

Java 说明文件管理器

提纲

javadoc [*options*] *@list package... sourcefiles...*

说明

javadoc 可以为任意数量所指定的包和类产生 API 说明文件。*javadoc* 命令行可以列出任意数量的包名和任意数量的 Java 源文件。为了方便起见，在处理大量的命令行选项或大量的包和类名时，你可以把它们全部放在辅助文件中，并在命令行指定的那个文件名称的前面加上 @ 字符。

Download at http://www.pin5i.com/

javadoc 使用 *javac* 编译器来处理所有被指定的 Java 源文件和指定包中的所有 Java 源文件。它使用从这个过程中收集到的信息来产生详细的 API 说明文件。最重要的是，产生的说明文件包含了所有定义于源文件中的说明注释内容。关于在你自己的Java程序代码中编写注释的信息，请参阅第七章。

当你为 *javadoc* 指定要处理的 Java 源文件时，必须要指定包含了完整的文件路径的文件名称。但是，更常见的是使用 *javadoc* 来建立针对整个包的类别的说明文件。当你指定 *javadoc* 所要处理的包时，要指定包名，而不是指定包含有那个包的源代码的目录。在这样的情况下，你可能必须指定 `-sourcepath` 选项，以让源代码不是存储在已列出的默认 classpath 的位置时，*javadoc* 可以正确地找到你的包的源代码。

javadoc 默认情况下会创建 HTML 说明文件，但你可以通过定义能以你所要的格式来产生说明文件的 `doclet` 类来自定义它的行为。你可以使用由 `com.sun.javadoc` 包定义的 doclet API 来编写你自己的 doclet。此包的说明文件被包含在 Java 1.2 及之后版本中的标准说明文件中。

javadoc 在 Java 1.2 中得到了重要的新功能。我们在这说明此程序的 Java 1.2 及之后的版本，但不区分各版本之间的特性差异。

选项

javadoc 定义了大量的选项。有些是 *javadoc* 一定能识别的标准选项，其他选项则是由产生说明文件的 doclet 所定义。针对标准 HTML doclet 的选项包含于以下列表中：

`-1.1`

模拟Java 1.1版的 `javadoc` 的输出风格和目录结构。此选项存在于Java 1.2和1.3，但在 Java 1.4 中已移除。

`-author`

在产生的说明文件中加上由 `@author` 所指定的作者信息。只有默认的doclet提供。

`-bootclasspath`

指定系统类的位置。这在交叉编译时很有用。关于此选项的更多信息，请参阅 *javac*。

`-bottom` *`text`*

在每个产生的HTML文件底端显示 *`text`*，可以包含HTML标记。请参阅 `-footer`。只有默认的 doclet 提供。

Download at http://www.pin5i.com/

-breakiterator

使用java.text.BreakIterator算法来决定doc注释中总结句子的结尾。只有默认的 doclet 提供。

-charset *encoding*

指定针对输出的字符编码。当然，这取决于你的源代码的说明注释中所使用的编码。*encoding*值会被用在 HTML 输出中的 <meta> 标记中。只有默认的 doclet 提供。

-classpath *path*

指定一个路径让*javadoc*查询类文件和源文件（如果你没有指定 -sourcepath 选项）。因为*javadoc*使用*javac*编译器，所以它必须要能找到要被说明的包所引用的所有类的类文件。关于此选项以及由 CLASSPATH 环境变量所提供的默认值的更多信息，请参阅*java*和*javac*。

-d *directory*

指定*javadoc*用来存储所产生的HTML文件的目录。如果忽略此选项，就会使用当前目录。只有默认的 doclet 提供。

-docencoding *encoding*

指定要被用于输出 HTML 文档的编码。此处所指定的编码名称可能不会与以 -charset 选项指定的字符集名称相匹配。只有默认的 doclet 有提供。

-docfilessubdirs

以递归方式复制*doc-files*目录的所有子目录，而不是只直接复制包含于*doc-files*中的文件。只有默认的 doclet 提供。

-doclet *classname*

指定用来产生说明文件的 doclet 类的名称。如果没有指定此选项，*javadoc*就会使用默认的 HTML doclet 来产生说明文件。

-docletpath *classpath*

如果由 -doclet 标记所指定的类无法从默认的 *classpath*获得时，系统可以从这个选项指定的路径进行加载。

-doctitle *text*

提供一个标题来显示在说明文件概述文件的开头。此文件通常是读者在浏览产生的说明文件时第一个看到的东西。标题可以包含HTML标记。只有默认的doclet提供。

-encoding *encoding-name*

指定输入源文件和它们所包含的说明注释的字符编码。它可以与-docencoding所指定的输出编码不同。默认值为平台的默认编码方式。

Download at http://www.pin5i.com/

-exclude *packages*

从由 -subpackages 选项定义的包集合中排除特定名称的包。*packages*是个以冒号分隔的包名列表。只有默认的 doclet 提供。

-excludedocfilessubdir *dirs*

在指定了 -docfilessubdirs时，排除 *doc-files* 目录的指定子目录。这在排除版本控制目录时很有用。*dirs*是以冒号分隔、相对于*doc-files*目录的目录名称列表。只有默认的 doclet 提供。

-extdirs *dirlist*

指定用来搜索标准扩展包的目录列表。只有在以 -bootclasspath来进行交叉编译时才需要。相关细节请参阅 *javac*。

-footer *text*

指定要显示在每个文件底部导航条右边的文字。*text* 可以包含HTML标记。请参阅 -bottom 和 -header。只有默认的 doclet 提供。

-group *title packagelist*

javadoc 会产生顶层的概述页面，列出生成说明文件中的所有包，默认情况下这些包会依字母顺序列在一个表格中。但是，你可以用这个选项把它们拆散为由相关包组成的组。*title* 指定了包组的标题，例如“Core Packages”。*packagelist* 是以冒号分隔的由包名组成的列表，其中每个名称都可以在结尾加上*字符。*javadoc* 命令行可以包含任意数量的 -group选项。例如：

```
% javadoc -group "AWT Packages" java.awt* \
-group "Swing Packages" javax.accessibility:javax.swing*
```

-header *text*

指定在接近文件顶部、上方导航条的右边所要显示的文字。*text* 可以包含 HTML 标记。请参阅 -footer、-doctitle以及 -windowtitle。只有默认的doclet提供。

-help

显示 *javadoc* 的使用说明。

-helpfile *file*

指定HTML文件的名称，其中包含关于生成的说明文件的使用辅助说明。*javadoc* 在所有生成文件中都包含了对此辅助说明文件的链接。如果没有指定这个选项，*javadoc* 就会创建默认的辅助说明文件。只有默认的 doclet 提供。

-J*javaoption*

将自变量 *javaoption* 直接传递给Java解释器。在处理大量的包时，你可能会需要使用这个选项来增加 *javadoc* 允许使用的内存量。例如：

Download at http://www.pin5i.com/


```
% javadoc -J-Xmx64m
```

请注意，因为 -J 选项会在*javadoc*启动之前被直接传给 Java 解释器，所以它们不可以被包括在命令行上以 @list 语法指定的外部文件中。

-keywords

告诉*javadoc*要把类型和成员名称包含在 <Meta> 标记关键字列表中。只有默认的 doclet 提供。

-link *url*

指定另一个由*javadoc*生成的说明文件的顶层目录的绝对或相对URL。*javadoc*使用这个 URL 作为基准 URL，以从当前文件链接至当前文件中未说明的包、类、method和字段。例如，在使用*javadoc*来产生你自己的包的说明文件时，你可以用这个选项把你的说明文件链接至核心 Java API 的*javadoc*说明文件。只有默认的 doclet 提供。

由*url*所指定的目录必须包含一个名为*package-list*的文件，而且*javadoc*必须要能在运行时读取这个文件。这个文件会由先前运行的*javadoc*自动产生，它包含了所有说明于 *url* 下的包的列表。

-link选项可以被指定多次，虽然这在较早的Java 1.2版本中并不能正常运行。如果没有指定 -link，那么在产生的说明文件中，对在外部的类和成员的引用就不会被超级链接。

-linkoffline *url packagelist*

与 -link 选项类似，除了*pakagelist*文件会被明确地在命令行上指定之外。当*url*所指定的目录没有*package-list*文件或那个文件在*javadoc*运行时无法取用时，这个选项很有用。只有默认的 doclet 提供。

-linksource

创建每个源文件的 HTML 版本并在说明文件页面加上针对这些 HTML 的链接。只有默认的 doclet 提供。

-locale *language_country_variant*

指定生成说明文件的位置（locale）。它被用来查询资源文件（resource file），其中包含针对输出文件的本地化信息和文字。

-nocomment

忽略所有的 doc 注释并产生只包含原始 API 信息但没有任何相关信息的说明文件。只有默认的 doclet 提供。

Download at http://www.pin5i.com/

-nodeprecated

告诉*javadoc*忽略掉已过时特性的说明文件。此选项暗示着-nodeprecatedlist。只有默认的doclet提供。

-nodeprecatedlist

告诉*javadoc*不要产生*deprecated-list.html*文件而且不要输出链接到导航条上。只有默认的doclet提供。

-nohelp

告诉*javadoc*不要在导航条上产生针对它的辅助说明文件或链接。只有默认的doclet提供。

-noindex

告诉*javadoc*不要产生索引文件。只有默认的doclet提供。

-nonavbar

告诉*javadoc*省略每个文件顶端和底端的导航条，也省略由-header和-footer指定的文字。这在产生要被打印的说明文件时很有用。只有默认的doclet提供。

-noqualifier *packages* | all

*javadoc*会针对同一个要说明的包中的类，在它产生的说明文件中忽略包名。此选项告诉它要额外忽略特定包中的类的包名，如果使用了关键字all，就是忽略所有的包名。*packages*是以冒号分隔的包名列表，它可以包含*通配符来指出子包。例如，-noqualifier java.io:java.nio.*会排除在java.io包和java.nio及其子包中所有类的包名。只有默认的doclet提供。

-nosince

忽略doc注释中的@since标记。只有默认的doclet提供。

-notimestamp

不要输出时间戳到HTML注释中。只有默认的doclet提供。

-notree

告诉*javadoc*不要产生*tree.html*类层次结构图或在导航条中提供链接。只有默认的doclet提供。

-overview *filename*

从*filename*读取概要doc注释并在概要页中使用那个注释。此文件没有包含Java源代码，所以doc注释不应该真的出现在/**和*/定界符之间。

-package

在输出中加上包层次可见的类和成员以及public与protected类和成员。

Download at http://www.pin5i.com/

-private

在产生的说明文件中加上所有的类和成员，其中包括 private 和包层次可见的类和成员。

-protected

在产生的输出中加上 public 和 protected 类与成员。这是默认值。

-public

在产生的输出中只加上 public 类和成员。省略了 protected、private 以及包层次可见的类和成员。

-quiet

抑制除了警告和错误消息之外的输出。

-serialwarn

对没有适当地以 @serial 和相关 doc 注释标记来说明序列化格式的可序列化类提出警告。只有默认的 doclet 提供。

-source *release*

指定编写源文件所使用的 Java 版本。请参阅 *javac* 的 -source 选项。合法的值有 5、1.5、1.4 和 1.3。选项 1.5 和 5 是同义的，而且是默认值。

-sourcepath *path*

指定源文件的搜索路径，通常会被设定为单一根目录。*javadoc* 会在寻找实现了指定包的 Java 源文件时使用这个路径。

-splitindex

产生多个索引文件，每个字母对应一个索引文件。在说明大量程序代码时会用到这个选项。否则，由 *javadoc* 产生的单一索引文件会因太庞大而难以使用。只有默认的 doclet 提供。

-stylesheetfile *file*

指定一个文件以用来为生成的 HTML 做 CSS 样式表。*javadoc* 会在产生的说明文件中插入对此文件的适当链接。只有默认的 doclet 提供。

-subpackages *packages*

指定 *javadoc* 应该处理特定的包及其全部子包。*packages* 是以冒号分隔、由包名或包名前缀所组成的列表。使用此选项通常会比明确地列出所有需要的包名来得简单。例如：

```
-subpackages java:javax
```

请参阅 -exclude。只有默认的 doclet 提供。

Download at http://www.pin5i.com/

-tag *tagname:where:header-text*

指定*javadoc*应该通过输出文字*header-text*、后面接着标记的文字，来处理名为*tagname*的doc注释标记。这使得在doc注释中使用简单的自定义标记（使用与@return和@author相同的语法）成为可能。*where*是个字符字符串，指定了此自定标记所被允许的doc注释类型。这些字符及其意义为a(all：所有地方都有效)、p（package：包）、t（type：类和接口）、c（constructor：构造函数）、m（method）以及f（field：字段）。

-tag选项的第二个用途是指定标记被处理以及输出时的出现顺序。你可以在-tag选项后面加上标准标记的名称来指定这个顺序。自定义标记和taglet可以被包含在此标准-tag选项列表中。只有默认的doclet提供。

-taglet *classname*

指定"taglet"类的类名称来处理自定义标记。taglet的编写在此不作说明。-taglet标记可以用-tag标记来散布，以指定标记被处理和输出的顺序。只有默认的doclet提供。

-tagletpath *classpath*

指定以冒号分隔、由JAR文件或目录组成的列表，它们构成了用来搜索taglet类的classpath。只有默认的doclet提供。

-use

针对列出类或包的用法的每个类和包，产生并插入链接到额外的文件中。

-verbose

在处理来源文件时显示额外的信息。

-version

从被产生的输出中的@version标记加上信息。此选项并不会告诉*javadoc*要输出它自己的版本编号。只有默认的doclet提供。

-windowtitle *text*

它定义了要输出在每个被产生文件的<Title>标记中的文本。此标题通常会作为web浏览器窗口的标题出现，并且会出现在历史记录与书签列表中。*text*不应该包含HTML标记。请参阅-doctitle和-header。只有默认的doclet提供。

环境

CLASSPATH

此环境变量指定了*javadoc*用来寻找类文件和源文件的默认classpath。它会被

Download at http://www.pin5i.com/

-classpath和-sourcepath选项覆盖。关于classpath的进一步讨论，请参阅*java*和*javac*。

另可参阅 *java*、*javac*

---

javah C 语言原生 method 的 stub 产生器

提纲

```
javah [ options ] classnames
```

说明

*javah*会产生C语言头文件和源文件（*.h*和*.c*文件），当以C实现Java的原生method时，就会用到它们。原生 method 接口在 Java 1.0 和Java 1.1 之间有所改变。在 Java 1.1 及之前的版本中，*javah* 会产生针对老式原生 method 的文件；在 Java 1.1 中，-jni 选项指出*javah* 应该产生新式的文件。在 Java 1.2 及之后的版本中，此选项是默认值。

本节只会说明使用*javah* 的方式。关于如何使用 C 来实现 Java 的原生 method 的完整说明，则超出了本书的范围。

选项

-bootclasspath

指定用来搜索系统类的路径。关于进一步的讨论请参阅*javac*。Java 1.2 及之后的版本提供。

-classpath *path*

指定*javah*用来查询命令行上的类名的路径。此选项会覆盖所有由CLASSPATH环境变量指定的路径。在Java 1.2之前，此选项可以指定系统类和扩展类的位置；在1.2及之后的版本中，它只能指定应用程序类的位置。请参阅-bootclasspath。关于classpath 的更进一步的讨论，请参阅*java*。

-d *directory*

指定*javah* 用来存储所产生的文件的目录。默认情况下，*javah* 会把它们存储在当前目录中。此选项不可以与 -o 一起使用。

-force

使*javah* 一定会写入输出文件，即使那没有包含有用的内容。

-help

使*javah* 显示简单的使用说明并退出。

Download at http://www.pin5i.com/

-Jjavaopt

传递选项 *javaopt* 至 Java 解释器。

-jni

指出 *javah* 应该输出头文件以配合 Java Native Interface（JNI）使用，而不是配合旧的 JDK 1.0 原生接口。此选项在 Java 1.2 及之后的版本中是默认值。另请参阅 -old。Java 1.1 及之后的版本提供。

-o *outputfile*

将所有输出合并至单一文件，而不是对每个指定的类创建独立的文件。

-old

输出 Java 1.0 风格的原生 method 的文件。在 Java 1.2 之前，这是默认值。另请参阅 -jni。Java 1.2 及之后的版本提供。

-stubs

针对类文件而不是针对头文件来产生 *.c* 的 stub 文件。此选项只针对 Java 1.0 原生 method 接口。请参阅 -old。

-trace

指定 *javah* 应该把追踪输出命令包含于它所产生的 stub 文件中。在 Java 1.2 及之后的版本中，此选项已过时且已被删除。你可以用 Java 解释器中的 -verbose:jni 选项来代替。

-v、-verbose

详细（verbose）模式。使 *javah* 输出关于进行中事项的消息。在 Java 1.2 及之后的版本中，-verbose 是同义词。

-version

使 *javah* 显示它的版本号。

环境

CLASSPATH

指定 *javah* 用来搜索指定类的默认 classpath。关于 classpath 的更进一步的讨论，请参阅 *java*。

另可参阅 *java*、*javac*

Download at http://www.pin5i.com/

javap　　Java 类反汇编器

提纲

```
javap [ options ] classnames
```

说明

javap 会读取由命令行上的类名所指定的类文件，并列出由那些类所定义的人能看懂的 API。*javap* 也可以反汇编指定的类，显示它们所包含的 method 的 Java VM 字节码。

选项

`-b`

启用与 Java 1.1 版的 *javap* 输出的向后兼容性。此选项的存在是针对依赖于 *javap* 精确输出格式的程序。Java 1.2 及之后的版本提供。

`-bootclasspath` *path*

指定针对系统类的搜索路径。关于此极少使用的选项的信息请参阅 *javac*。Java 1.2 及之后的版本提供。

`-c`

显示每个被指定类的每个 method 的程序代码（即 Java VM 字节码）。此选项一定会反汇编所有的 method，而不管它们的可见度。

`-classpath` *path*

指定路径以供 *javap* 查询命令行上所命名的类。此选项会覆盖由 CLASSPATH 环境变量所指定的路径。在 Java 1.2 之前，这个自变量会指定针对所有系统类、扩展类以及应用程序类的路径。在 Java 1.2 及之后的版本中，它只指定应用程序的 classpath。也请参阅 `-bootclasspath` 和 `-extdirs`。关于 classpath 的更多信息，请参阅 *java* 和 *javac*。

`-extdirs dirs`

指定一个或多个针对扩展类进行搜索的目录。关于此极少使用的选项的信息，请参阅 *javac*。

`-J`*javaopt*

将选项 *javaopt* 传递给 Java 解释器。

`-l`

如果能在类文件中取得，就显示行号和局部变量列表。此选项通常只有在配合 `-c` 使用时才有用。默认情况下，*javac* 编译器不会在它的类文件中加上局部变量信息。请参阅针对 *javac* 的 `-g` 及相关选项。

Download at http://www.pin5i.com/

-help

输出使用说明并退出。

-Jjavaoption

将指定的*javaoption*直接传递给Java解释器。

-package

显示在包层次可见的（package-visible）、protected以及public类成员，但不显示private成员。这是默认值。

-private

显示所有的类成员，其中包括private成员。

-protected

只显示protected和public成员。

-public

只显示指定类的public成员。

-s

使用内部的VM类型和method签名格式来输出类成员声明，而不使用更具可读性的源代码格式。

-verbose

详细模式。输出关于各个指定类的每个成员的额外信息（使用Java注释的形式）。

环境

CLASSPATH

指定针对应用程序类的默认搜索路径。-classpath选项会覆盖这个环境变量。关于classpath的内容请参阅*java*。

另可参阅

java、*javac*

---

javaws　　Java Web Start启动器

提纲

```
javaws
javaws [ options ] url
```

说明

*javaws*是针对Java Web Start网络应用程序启动器（launcher）的命令行接口。当在没

Download at http://www.pin5i.com/

有添加*url*的情况下启用时，*javaws*会显示图形化的高速缓存监视程序，它能让高速缓存应用程序被启动以及让 Java Web Start 被配置。

如果 JNLP（Java Network Launching Protocol）的 URL 被指定在命令行上，*javaws*就会启动指定的应用程序。

选项

-association

允许在使用 -silent -import 的期间创建文件关联。

-codebase *url*

以指定的 *url* 覆盖 JNLP 文件中的 codebase。

-import

将指定的应用程序导入用户高速缓存中但不加以运行。

-offline

以离线模式运行。

-online

以联机模式启动。这是默认的行为。

-shortcut

允许在 -silent -import 的期间创建桌面快捷方式。

-silent

在与 -import 一起使用时，此选项能避免 GUI 窗口出现。

-system

使用系统高速缓存。

-uninstall

从用户的高速缓存删除由 *url* 所标识的应用程序并退出。

-updateVersions

更新*javaws*配置文件（例如在升级到新版的 Java 之后）。

-userConfig *name* [*value*]

设定布局特性 *name*；或者，如果 *value* 被指定，就将 *name* 设定为指定的值。

-viewer

启动高速缓存监视应用程序。如果在不加上自变量的情况下调用*javaws*，这就是默认的行为。

Download at http://www.pin5i.com/

-wait

直到被启动的应用程序退出时才退出。

-Xclearcache

清除用户高速缓存并退出。

-Xnosplash

不显示 Java Web Start 快闪画面。

jconsole 图形化 Java 进程监控程序

提纲

```
jconsole [ options ]
jconsole [ options ] pid
jconsole [ options ] host:port
```

说明

jconsole 是针对内存、线程、类加载及其他由 `java.lang.management` 所提供的监控工具的图形化界面。它可以监控一个或多个本地或远程 Java 进程。只有以特殊的系统属性集启动的进程才能被监控。如果要允许 Java VM 能被本地监控，就应这样启动：

```
% jconsole -Dcom.sun.management.jmxremote=true
```

如果要允许 Java VM 能被远程监控，就应这样启动：

```
% jconsole -Dcom.sun.management.jmxremote.port=port
```

其中，`port` 是 *jconsole* 所要连接的远程端口。

你可以在不指定本地或远程进程的情况下启动 *jconsole*，并使用它的 Connection 菜单来建立连接。这是将 *jconsole* 连接至多个 Java 进程的唯一方法。

如果要在 *jconsole* 启动时连接至本地进程，只要把进程 id 列在命令行上就可以了。请查看 *jps* 来判定进程 id。

如果要在 *jconsole* 启动时连接至远程进程，就要在命令行上指定主机名称和通信端口号。此通信端口号应该与由目标进程的 `com.sun.management.jmxremote.port` 系统属性所指定的相同。

Download at http://www.pin5i.com/

选项

-help

显示使用说明。

-interval=*n*

将更新期设定为 *n* 秒。默认值为 4。

-version

显示 *jconsole* 版本并退出。

另可参阅 *jps*、*jstat*

---

jdb

Java 调试器

概要

```
jdb [ options ] class [ program options ]
jdb connect options
```

说明

jdb 是 Java 类的调试器。它是基于文本、面向命令行的，而且具有像 C 和 C++ 程序所使用的 Unix *dbx* 或 *gdb* 调试器的命令行语法。

jdb 是用 Java 编写的，所以它会在 Java 解释器中运行。当 *jdb* 与 Java 类名一起被调用时，它会启动另一个 *java* 解释器副本，此副本会使用命令行上指定的所有解释器选项。新的解释器会以能让它与 *jdb* 通信的特殊选项来启动。新的解释器会加载指定的类文件，然后在执行第一个字节码之前停止并等待调试命令。

jdb 也可以对已在另一个 Java 解释器中运行的程序进行调试。要这么做需要把特殊的选项传递给 *java* 解释器和 *jdb*。请参阅以下的 -attach 选项。

jdb 表达式语法

例如 print、dump 以及 supend 这样的 *jdb* 调试命令，能让你以名称来引用类、对象、method、字段以及要被调试的程序中的线程。你可以用名称来引用类，是否加上它们的包名都可以。你也可以用名称来引用静态类成员。你可以用对象 ID 来引用个别的对象，而对象 ID 是八位的十六进制整数。或者，当你正在调试的类包含局部变量信息时，你通常可以使用局部变量名称来引用对象。你可以使用一般的 Java 语法来引用对象的字段和数组的元素；你也可以使用这个语法来编写相当复杂的表达式。在 Java 1.3 中，*jdb* 甚至支持使用标准 Java 语法的 method 调用。

Download at http://www.pin5i.com/

选项

当加上指定的类文件来调用*jdb*时，可以指定所有的*java*解释器选项。关于这些选项的说明，请参阅*java*的参考页面。此外，*jdb*支持以下选项：

`-attach [host:]port`

指出*jdb*应该连接到已在指定主机（如果没有指定，就是本地主机）上运行的Java VM，并正在监听指定端口上的调试连接。Java 1.3及之后的版本提供。

为了要以这种方式使用*jdb*来连接至运行中的VM，VM必须已用特殊的命令行选项启动。在Java 1.3和1.4中，使用这些选项：

```
% java -Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

在Java 5.0中，使用这些选项来代替：

```
% java -agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n
```

Java调试结构允许复杂的解释器到调试器的连接选项，而*java*和*jdb*提供了选项和子选项使其成为可能。对于这些选项的详细说明超出了本书的范围。

`-connect connector:args`

此选项对于将*jdb*连接至要被调试的进程，提供了最普通和具有灵活性的方法。指定`connector`（一个Java类）的名称，后面接着具有name=vaule形式、以逗号分隔的列表。Java 1.4及之后的版本提供。关于可用的connector及它们的自变量，请参阅`-listconnectors`。

`-help`

显示的使用信息列出了所支持的选项。

`-launch`

当*jdb*启动时就启动指定的应用程序。这避免了明确地以`run`命令来加以启动的必要。Java 1.3及之后的版本提供。

`-listconnectors`

列出可用的连接方法。每个connector都是一个Java类和自变量列表。Java 5.0及之后的版本提供。请参阅`-connect`选项。

`-listen port`

监听Java VM的特定端口以连接至调试器。如果要让它运作，VM就必须加上像这样的选项：

```
% java -agentlib:jdwp=transport=dt_socket,address=8000,server=n,suspend=y
```

Java 1.4及之后的版本提供。

Download at http://www.pin5i.com/

-listenany

和-listen选项一样，但*jdb*会挑选一个端口来监听，并列出端口号以在启动Java进程来调试时使用。Java 1.4 及之后的版本提供。

-sourcepath *path*

指定当尝试找出对应至要被调试的类文件的源文件时所要搜索的位置。如果没有指定，*jdb*就会根据默认使用 classpath。Java 1.3 及之后的版本提供。

-tclient

告诉*jdb*调用客户端的Java解释器版本。

-tserver

告诉*jdb*调用服务器端的Java解释器版本。

-version

显示*jdb*版本号并退出。

命令

*jdb*能理解以下的调试命令。关于更多的信息请使用help命令。

?或help

列出所有已支持的命令，并对每条命令加上简短的说明。

!!

这条简短的命令会取代上一条输入的命令。后面可以加上额外的文字来跟在那条命令后面。

catch *[exception-class]*

每当有指定异常被抛出时就产生一个断点。如果没有指定异常，此命令就会列出当前捕获的异常。使用 ignore 可以避免这些断点出现。

classes

列出所有已被载入的类。

clear

列出所有当前已设定的断点。

clear *class.method [(param-type...)]*

清除设定于指定类的指定 method 里的断点。

clear *[class:line]*

移除设定于指定类的指特定行的断点。

Download at http://www.pin5i.com/

cont

继续执行。当当前线程停在断点时，就应该使用此命令。

down [*n*]

在当前线程的调用堆栈中向下移 *n* 帧（frame）。如果没有指定 *n*，就向下移一帧。

dump id...

列出指定对象的所有字段的值。如果你指定了类名称，dump就会显示出所有类（静态）method 和变量，并显示超类和已实现接口的列表。对象和类可以通过名称或它们的八位十六进制 ID 号来指定。线程也可以用简便的 *t@thread-number* 来指定。

exit 或 quit

结束 *jdb*。

gc

运行内存回收以强制回收未被使用的对象。

ignore *exception-class*

不要把特定的异常当成断点。此命令会关闭 catch 命令。此命令不会使 Java 解释器忽略异常，它只是告诉 *jdb* 要忽略异常。

list [*line-number*]

列出指定行及其前后几行的源代码。如果没有指定行号，就使用当前线程的当前堆栈帧的行号。被列出的行是来自当前线程的当前堆栈帧的源文件。使用use命令可以告诉 *jdb* 要到哪里找源文件。

list *method*

显示指定 method 的源代码。

load *classname*

将指定类载入 *jdb*。

locals

显示针对当前堆栈帧的局部变量列表。Java 程序代码必须使用 -g 选项来编译，以便于包含局部变量信息。

methods *class*

列出指定类的所有 method。使用dump列出对象的实例变量或类的类（静态）变量。

print *id...*

列出指定项的值。各个入口项是类、对象、字段或局部变量，而且可以用名称或八位十六进制 ID 号来指定。你也可以用特殊的 *t@thread-number* 语法来引用线程。print 命令会通过调用它的 toString() method 来显示对象的值。

Download at http://www.pin5i.com/

next

执行源代码的当前行，其中包括它做的所有 method 调用。另可参阅 step。

resume [*thread-id...*]

继续执行指定的线程。如果没有指定线程，所有挂起的线程就会恢复。另请参阅 suspend。

run [*class*] [*args*]

运行指定类的 main() method，把指定的自变量传递给它。如果没有指定类或自变量，就会使用 *jdb* 命令行上指定的类和自变量。

step

运行当前线程当前所在的行并再次停止。如果这行调用了 method，就会进入那个 method 然后停止。另请参阅 next。

stepi

执行一条单一 Java VM 指令。

step up

运行直到当前 method 返回至其调用者，并再次停止。

stop

列出当前的断点。

stop at *class:line*

在指定类的指定行设置断点。程序会在运行到这行时停止。可以使用 clear 来删除断点。

stop in *class.method [(param-type...)]*

在指定类的指定 method 的开始设置断点。程序会在进入此 method 时停止。可以使用 clear 来删除断点。

suspend [*thread-id...*]

挂起指定的线程。如果没有指定线程，就挂起所有运行中的线程。使用 resume 可以重新启动它们。

thread *thread-id*

将当前线程设定为指定的线程。此线程会由一些其他的 *jdb* 命令隐含地使用。

threadgroup *name*

设定当前线程组。

threadgroups

列出所有在正在调试的 Java 解释器会话上运行的线程组。

Download at http://www.pin5i.com/

threads [*threadgroup*]

列出具名线程组中的所有线程。如果没有指定线程组，就列出当前线程组（由 *threadgoup* 指定）中的所有线程。

up [*n*]

在当前线程的调用堆栈中向上移 *n* 帧。如果没有指定 *n*，就向上移一帧。

use [*source-file-path*]

设定 *jdb* 所使用的路径，以查询要被调试的类的源文件。如果没有指定路径，就显示当前源路径。

where [thread-id] [all]

显示指定线程的堆栈踪迹。如果没有指定线程，就显示当前线程的堆栈踪迹。如果指定了 all，就显示所有线程的堆栈踪迹。

wherei [*thread-idx*]

显示指定或当前线程的堆栈踪迹，其中包括详细的程序计数信息。

环境

CLASSPATH

指定了目录、ZIP 文件以及 JAR 归档文件的有序列表（在 Unix 系统中用冒号分隔，在 Windows 系统中用分号分隔），*jdb* 会以此列表来寻找类定义。当使用这个环境变量指定路径时，*jdb* 一定会隐式地将系统类的位置附加到此路径的结尾。如果没有指定这个环境变量，默认的路径就是当前目录和系统类。此变量会被 -classpath 选项覆盖。

另可参阅 *java*

---

jinfo 显示 Java 进程的配置

提纲

```
jinfo [ options ] pid                          // 在本地进程上的信息
jinfo [ options ] executable core              // 来自 core 文件的信息
jinfo [ options ] [process-name@]hostname      //  来自远程进程的信息
```

说明

jinfo 会列出运行中的 Java 进程或核心文件的系统特性和 JVM 命令行选项。*jinfo* 的启动方式有以下三种：

Download at http://www.pin5i.com/

- 指定运行于本地的Java进程的进程id来取得关于它的配置信息。请参阅*jps*来列出本地的进程。
- 如果要从核心文件取得事后检查（post-mortem）配置信息，就在命令行上指定产生核心文件的Java可执行文件和核心文件本身。
- 如果要取得关于远程运行的Java进程的配置信息，就指定远程主机的名称并选择性地在前面加上远程进程名称。在远程主机上必须运行*jsadebugd*。

在Java 5.0中，*jinfo*是实验性的、未被支持的，而且不是在所有平台都可用的。

选项

这些选项是互斥的，只能指定其中一个。

`-flags`

只列出JVM标记，而不列出系统特性。

`-help、-h`

列出辅助说明信息。

`-sysprops`

只列出系统特性，而不列出JVM标记。

另可参阅

jps、*jsadebugd*

---

jmap 显示内存使用状况

提纲

```
jmap [ options ] pid                          // 本地进程
jmap [ options ] executable core              // 核心文件
jmap [ options ] [process-name@]hostname      // 远程进程
```

说明

*jmap*会列出本地或远程Java进程或Java核心文件的内存使用情况的信息。依据配合调用的选项，*jamp*可以显示的内存使用状况报表有四种。相关细节请参阅选项章节。*jmap*可以用三种方式来启动：

- 指定运行于本地的Java进程的进程id来取得关于它的配置信息。请参阅*jps*来列出本地进程。

Download at http://www.pin5i.com/

- 如果要从核心文件取得事后检查（post-mortem）配置信息，就在命令行上指定产生核心文件的 Java 可执行文件和核心文件本身。
- 如果要取得关于远程运行的 Java 进程的配置信息，就指定远程主机的名称并选择性地在前面加上远程进程名称和 @ 符号。在远程主机上必须运行 *jsadebugd*。

在 Java 5.0 中，*jmap* 是实验性的、未被支持的，而且不是在所有平台都可用的。

选项

在不加上选项进行调用时，*jmap* 会列出 VM 所加载的共享对象或函数库的内存映射（map）。其他的报告可以使用以下选项来产生。这些选项是互斥的，只能指定其中一个。

`-heap`

显示堆内存使用情况的汇总。

`-help, -h`

列出辅助说明信息。

`-histo`

以类显示堆使用情况的柱状图。

`-permstat`

显示已加载的类所使用的内存，以类加载程序来分组。

另可参阅 *jps*、*jsadebugd*

---

jps 列出 Java 进程

提纲

```
jps [ options ] [ hostname[:port ] ]
```

说明

jps 会列在本地主机或在指定远程主机上运行的 Java 进程。如果指定了远程主机，那台主机就必须运行 *jstatd* 后台进程（daemon）。针对每个 Java 进程，它会显示进程正在执行的类或 JAR 文件的进程 id 和名称。进程 id 会被其他的一些 Java 工具使用，例如 *jconsole*、*jstat* 以及 *jmap*。

选项

以下的选项会更改默认的 *jps* 显示项目。除了 `-q` 之外，单一字母的选项可以结合成单一命令行自变量，例如 `-lmv`：

Download at http://www.pin5i.com/

-help

显示使用状况信息。

-l

列出在每个 Java 进程中运行的主要类的完整包名称或 JAR 文件的完整路径。

-m

列出传递给每个 Java 进程的 main() method 的自变量。

-q

只列出 Java 进程标识符，而不列出应用程序名称或任何额外的信息。

-v

列出通过每个 Java 进程传递给 Java 解释器的自变量。

-V

列出通过标记文件（例如 *.hotspotrc*）传递给解释器的自变量。

另可参阅 *jstatd*

---

jsadebugd 用于远程调试的 daemon 进程

提纲

```
jsadebugd pid [ process-name ]                // 运行中的进程
jsadebugd executable core [ process-name ]    // 核心文件
```

说明

*jsadebugd*是个服务器进程，它允许在本地Java进程或核心文件上的*jinfo*、*jmap*和*jstack*的远程调用。通过在命令行上指定运行中的Java进程或可执行文件的进程id和核心文件对，就可以调用*jsadebugd*。如果同一台主机上要同时运行多个*jsadebugd*服务器，就在这些自变量后面加上可识别的进程名称，让远程的客户端用来识别所需的进程。

jsadebugd 会启动 *rmiregistry* 服务器。

在 Java 5.0 中，*jsadebugd* 是实验性的、未被支持的，而且不是在所有平台都可用的。

另可参阅 *jinfo*、*jmap*、*jstack*

Download at http://www.pin5i.com/

jstack 显示 Java 进程的堆栈踪迹

提纲

```
jstack [ options ] pid                          // 本地进程
jstack [ options ] executable core              // 核心文件
jstack [ options ] [process-name@]hostname      // 远程进程
```

说明

jstack 会列出运行于指定 Java 进程中的各个 Java 线程的堆栈踪迹。*jstack* 的启动方式有三种：

- 指定运行于本地的Java进程的进程id来取得关于它的配置信息。请参阅*jps*来列出本地进程。
- 如果要从核心文件取得事后检查配置信息，就在命令行上指定产生核心文件的Java可执行文件和核心文件本身。
- 如果要取得有关远程运行的 Java 进程的配置信息，就指定远程主机的名称并选择性地在前面加上远程进程名称和 @ 符号。在远程主机上必须运行 *jsadebugd*。

在 Java 5.0 中，*jstack* 是实验性的、未被支持的，而且不是在所有平台都可用。

选项

`-help`、`-h`

列出辅助说明信息。

`-m`

以“混合模式”显示堆栈踪迹，也就是同时显示 Java 和原生 method 堆栈帧。如果没有加上这个选项，就是默认为只显示 Java 堆栈帧。

另可参阅 *jps*、*jsadebugd*

jstat Java VM 统计数据

提纲

```
jstat [options ] pid [ interval[s|ms] [ count ]]
jstat [options ] pid@hostname[:port] [ interval[s|ms] [ count ]]
```

Download at http://www.pin5i.com/

说明

*jstat*会探测运行中的JVM一次或重复多次，并显示关于它的类加载、实时编译、内存回收性能的统计数据。所显示的信息类型是由`option`来指定。要被探测的本地进程会通过它的进程id来指定，例如由*jps*返回的id。远程Java进程可以通过指定远程进程id、远程主机名称以及远程主机的*rmiregistry*服务器上所运行的通信端口编号（如果不是默认的1099）来探测。远程主机也必须运行*jstatd*服务器。

默认情况下，*jstat*会探测指定的Java VM一次。你也可以指定以毫秒或秒为单位的探测间隔来让它重复探测。如果你这么做，就可以额外指定要实施的探测总次数。

*jconsole*可以报告许多与*jstat*相同的统计数据，但是使用图形形式来加以显示而不是表格形式。在Java 5.0中，*jinfo*是实验性的、未被支持的，而且不是在所有平台都可用。

选项

`-help`

显示辅助说明信息。

`-options`

显示*jstat*所能显示的报告类型的列表。每次运行*jstat*，你都必须使用其中一个被列出的选项。

`-version`

显示*jstat*版本信息并退出。

`-h n`

当*jstat*重复探测Java进程时，此选项指定了多久就要在输出中重复表格的表头。此选项必须接在以下一种报告类型选项后面。

`-t`

对由*jstat*产生的报告增加Timestamp栏。此栏显示从目标Java进程启动后所经过的时间（以秒为单位）。

以下选项指定由*jstat*报告的统计数据类型。除非你加上`-help`、`-options`或`-version`来运行*jstat*，否则你应该精确地指定这些选项的其中一个，而且它必须是命令行上的第一个选项。大部分的选项会产生内存回收细节的详细报告。关于这些报告的解释，请参阅Sun的工具说明文件（这是JDK文件包的一部分）。

`-class`

报告已加载的类的数量和它们的大小（以kilobyte为单位）。

Download at http://www.pin5i.com/

-compiler

报告已执行的实时编译的数量以及所花的时间。

-gc

报告堆内存回收统计数据。

-gccapacity

报告内存回收程序的各个内存区的容量信息。

-gccause

就像 -gcutil 所报告的，但加上关于最近内存回收的成因的信息。

-gcnew

报告关于内存回收程序的“新一代（new generation）”内存池的信息。

-gcnewcapacity

报告内存回收程序的“新一代”内存池的容量信息。

-gcold

报告内存回收程序的“旧时代”和“永久代”（permanent）内存池的信息。

-gcoldcapacity

报告内存回收程序的“旧时代”内存池的信息。

-gcpermcapacity

报告内存回收程序的“永久代”的信息。

-gcutil

报告内存回收的汇总数据。

-printcompilation

报告关于实时编译的额外信息，其中包括已编译的 method 的名称。

另可参阅 *jconsole*、*jps*、*jstatd*

---

jstatd

jstat daemon

提纲

```
jstatd options
```

说明

jstatd 是服务器，它提供关于本地 Java 进程的信息给运行于远程主机上的 *jps* 和 *jstat* 程序。

Download at http://www.pin5i.com/

*jstatd*使用了RMI而且需要特殊的安全权限才能顺利运行。如果要启动*jstatd*，可以创建以下文件并把它命名为*jstatd.policy*：

```
grant codebase "file:${java.home}../lib/tools.jar {
    permission java.security.AllPermission
}
```

此策略会将所有的权限授予任何由JDK的名为*tools.jar*的JAR文件所载入的类。如果用这个策略来启动*jstatd*，可以使用这行命令：

```
% jstatd -J-Djava.security.policy=jstat.policy
```

如果已有*rmiregistry*服务器在运行，*jstatd*就会使用它；否则，它会创建它自己的RMI注册（registry）。

选项

`-n` *rminame*

将*jstatd*远程对象与RMI registry中的*rminame*名称绑定。默认的名称是"JstatRemoteHost"，这就是*jps*和*jstat*所寻找的。如果要使用这个选项，远程的*jps*和*jstat*调用中就必须使用*rminame*。

`-nr`

告诉*jstatd*如果没有已运行的内部RMI registry，不要启动内部RMI registry。

`-p` *port*

寻找*port*上已有的RMI registry，如果没有找到已有的registry，就在那个通信端口上启动一个。

另可参阅 *jps*、*jstat*

---

keytool 密钥与证书管理工具

提纲

```
keytool command options
```

说明

*keytool*管理并操作*keystore*，*keystore*是公钥、密钥和公钥证书的存储库。*keytool*定义了各种命令，用于产生密钥、将数据导入*keystore*以及导出并显示*keystore*数据。密钥和证书是使用不区分大小写的名称或别名来存储在*keystore*中。*keytool*使用这个别名来引用密钥或证书。

Download at http://www.pin5i.com/

*keytool*的第一个选项总是指定所要执行的基本命令。后续的选项提供了关于如何执行命令的细节。只有命令才一定要被指定。如果命令所需的选项不具有默认值，*keytool*就会以交互的方式来提示你输入值。

命令

-certreq

针对指定的别名产生PKCS#10格式的证书签署请求（certificate signing request）。此请求会被写入指定的文件或标准输出流。此请求应该被送到认证机构（certificate authority，CA），它会验证请求者并回送加上签名的证书来证明请求者的公钥是真的。对于这个已加上签名的证书接着可以用-import命令来导入*keystore*。此命令使用了以下选项：-alias、-file、-keypass、-keystore、-sigalg、-storepass、-storetype和-v。

-delete

从指定*keystore*删除指定的别名。此命令使用了以下选项：-alias、-keystore、-storepass、-storetype以及-v。

-export

将证书及相关的指定别名写入指定文件或标准输出。此命令使用了以下选项：-alias、-file、-keystore、-rfc、-storepass、-storetype和-v。

-genkey

产生一对公共/私有密钥以及针对公钥的自签（self-signed）的X.509证书。自签证书本身通常没什么用，所以这个命令后面通常会接着-certreq。此命令使用了以下选项：-alias、-dname、-keyalg、-keypass、-keysize、-keystore、-sigalg、-storepass、-storetype、-v和-validity。

-help

列出所有可用的*keytool*命令及它们的选项。此命令不会与任何其他命令一起使用。

-identitydb

从利用已废除的*javakey*程序所管理的原有身份数据库中读取密钥和证书并把它们存储到*keystore*中，以便让它们能被*keytool*操作。此身份数据库是从指定的文件中被读取的，如果没有指定文件，就会从标准输入来读取。密钥和证书会被写入指定的*keystore*文件中，如果它还不存在，就会自动创建。此命令使用了以下选项：-file、-keystore、-storepass、-storetype和-v。

-import

从指定的文件或从标准输入中读取证书或PKCS#7格式的证书链（certificate

Download at http://www.pin5i.com/

chain)，并把它当成可信的证书来以指定的别名存储在*keystore*中。此命令使用了以下选项：-alias、-file、-keypass、-keystore、-noprompt、-storepass、-storetype、-trustcacerts 和 -v。

-keyclone

复制指定别名的密钥存储入口，并把它存储在*keystore*中的一个新别名中。此命令使用了以下选项：-alias、-dest、-keypass、-keystore、-new、-storepass、-storetype 和 -v。

-keypasswd

变更用来加密与指定别名相关联的私钥的密码。此命令使用了以下选项：-alias、-keypass、-new、-storetype 和 -v。

-list

(在标准输出上)显示与指定别名相关联的证书的特征。如果加上 -v 选项，就以人能看懂的格式列出证书的细节；如果加上 -rfc，就以机器可理解的、可打印的编码格式列出证书的内容。此命令使用了以下选项：-alias、-keystore、-rfc、-storepass、-storetype 和 -v。

-printcert

显示从指定文件或标准输入中读取的证书的内容。与大部分的*keytool*命令不同，这个命令没有使用密钥存储。此命令使用了以下选项：-file 和 -v。

-selfcert

针对与指定别名称相关联的公钥创建自签证书，并用它取代已与那个别名相关联的任何证书或证书链。此命令使用了以下选项：-alias、-dname、-keypass、-keystore、-sigalg、-storepass、-storetype、-v 和 -validity。

-storepasswd

变更保护密钥存储整体完整性的密码。新的密码在长度上至少要有六个字符。此命令使用了以下选项：-keystore、-new、-storepass、-storetype 和 -v。

选项

以下列出的各种选项都可以传递给各种 *keytool* 命令。这些选项中有很多有合理的默认值。对于任何不具有默认值且未被指定的选项，*keytool* 会以交互的方式提示我们输入值。

-alias *name*

指定密钥存储中用到的别名。默认值为“mykey”。

Download at http://www.pin5i.com/

-dest *newalias*

为 -keyclone 命令指定新的别名（目的别名）。如果没有指定，*keytool* 就会提示我们输入值。

-dname *X.500-distinguished-name*

指定 X.500 专有名称（distinguished name）要出现在由 -selfcert 或 -genkey 所产生的证书上。专有名称是个高度限定的名称，在全球都是唯一的。例如：

```
CN=David Flanagan, OU=Editorial, O=OReilly, L=Cambridge, S=Massachusetts, C=US
```

如果没有指定识别名称，*keytool* 的 -genkey 命令就会提示输入。如果没有指定替代名称，-selfcert 命令就会使用当前证书的专有名称。

-file *file*

为 *keytool* 命令指定输入或输出文件。如果没有指定，*keytool* 就会从标准输入中读取或写入到标准输出。

-keyalg *algorithm-name*

配合 -gnekey 使用，以指定要产生哪种类型的加密密钥。在 Sun 所默认的 Java 实现中，唯一支持的算法是“DSA”，如果此选项被忽略，这就是默认值。

-keypass *password*

指定在加密密钥存储中的私钥所用的密码。如果没有指定这个选项，*keytool* 会先尝试 -storepass password。如果不成功，它就会提示我们要输入适当的密码。

-keysize *size*

配合 -genkey 命令一起使用，以指定生成的密钥以位为单位的长度。如果没有指定，默认值为 1024。

-keystore *filename*

指定密钥存储文件的位置。如果没有指定，就会使用位于用户根目录中名为 *.keystore* 的文件。

-new *new-password-or-alias*

配合 -keyclone 命令一起使用以指定新的别名，以及配合 -keypasswd 和 -storepasswd 一起使用以指定新的密码。如果没有指定，*keytool* 就会提示我们要输入这个选项的值。

-noprompt

配合 -import 命令一起使用，以在无法针对导入的证书创建信任链时禁止和用户之间的交互。如果没有指定这个选项，-import 命令就会提示用户。

Download at http://www.pin5i.com/

-rfc

配合 -list 和 -export 命令一起使用，以指定证书的输出应该要使用由 RFC 1421 所指定的可打印编码格式。如果没有指定这个选项，-export 就会以二进制格式输出证书，而 -list 会只列出证书的特征。此选项在 -list 命令中不可以与 -v 结合。

-sigalg *algorithm-name*

指定签署证书的数字签名算法。如果省略，则此选项的默认值会取决于底层公钥的类型。如果它是 DSA 密钥，则默认的算法就是“SHA1withDSA”；如果它是 RSA 密钥，则默认的签名算法就是“MD5withRSA”。

-storepass *password*

指定用来保护整个密钥存储文件的完整性的密码。这个密码也作为本身没有指定 -keypass 的所有私钥的默认密码。如果没有指定 -storepass，*keytool* 就会加以提示。此密码的长度至少为六个字符。

-storetype *type*

指定所要使用的密钥存储类型。如果没有指定这个选项，就会从系统安全属性文件中取得默认值。通常，默认值为“JKS”——Sun 的 Java Keystore 类型。

-trustcacerts

配合 -import 命令一起使用，以指示包含于 *jar/lib/security/cacerts* 文件的密钥存储中的自签证书应该被当成可信任的。如果忽略这个选项，*keytool* 就会忽略那个文件。

-v

详细模式，只要定义了它，就会让许多的 *keytool* 命令产生额外的输出。

-validity *time*

配合 -gnekey 和 -selfcert 命令一起使用，以指定所产生的证书的有效期（单位为天）。如果没有指定，默认值为 90 天。

另可参阅 *jarsigner*、*policytool*

---

native2ascii 将文字转换为具有 Unicode 转义字符的 ASCII

提纲

```
native2ascii [ options ] [ inputfile [ outputfile ]]
```

说明

native2ascii 是个简单的程序，它读取使用地区编码的文本文件（通常是 Java 源文件），

Download at http://www.pin5i.com/

并把它转换为Java语言规范所允许的Latin-1-plus-ASCII-encoded-Unicode形式。当你必须编辑Java程序代码文件，但又没有可以处理文件编码的编辑器时，这是很有帮助的。

*inputfile*和*outputfile*是选择性的。如果没有指定，就会使用标准输入和标准输出，这使得 *native2ascii* 很适合用在管道中。

选项

-encoding *encoding-name*

指定源文件所使用的编码方式。如果没有指定这个选项，就会从 file.encoding 系统属性中取得编码方式。

-reverse

指定转换应该要反向进行 —— 从已编码的 \uxxxx 字符转换为原生编码方式的字符。

另可参阅 java.io.InputStreamReader、java.io.OutputStreamWriter

---

pack200 压缩 JAR 文件

提纲

pack200 [*options*] *outputfile jarfile*

说明

*pack200*会使用由JSR 200和标准gzip压缩算法的定义的压缩算法来紧密压缩JAR文件。请注意，命令行上的输出文件是被指定在输入的JAR 文件之前。

基本选项

所有的*pack200*选项都同时以详细形式和单字母形式存在，详细形式是以双连字符开始，而单字母形式是以单连字符开始。当选项必须要有值时，这个值在详细形式中应该用等号与选项分隔，而且不可以有空格；而在单字母形式中应该要紧接在选项后面，中间不可以有空格或标点符号。

--config-file=*file*、-f*file*

从指定的配置文件读取选项。*file*应该是具有name=vaule格式的 java.util.Properties文件。所支持的特性名称与列于此处的详细形式的选项名称相同，但连字符转换为句号。

Download at http://www.pin5i.com/

--effort=*value*、-E*value*

指定尝试压缩JAR文件的程度。*value*必须是介于0和9之间的数字。0代表完全不压缩，只是产生输入的JAR文件的副本。默认值为5。

--help、-h

显示辅助说明信息并退出。

--log-file=*file*、-l*file*

将输出记录到文件。

--no-gzip、-g

告诉*pack200*不要对已压缩的JAR文件使用gzip压缩。如果你想要使用不同的压缩过滤程序，例如*bzip2*，就使用这个选项。默认值为--gzip。

--no-keep-file-order、-o

允许*pack200*重新排序JAR文件的元素。--keep-file-order为默认值。

--quiet、-q

抑制输出信息。

--pass-file=*file*、-P*file*

跳过指定的文件不进行压缩。如果*file*是以/结束，目录中所有的文件都会被跳过而不被压缩。这个选项可以被指定多次。

--repack、-r

压缩指定的JAR文件，然后立刻解压缩。在这样的情况下，在命令行上所指定的*outputfile*就应该是JAR文件的名称。在使用*jarsigner*来签名之前，在JAR文件上进行压缩/解压缩循环是很重要的，因为压缩/解压缩循环会重排类文件的一些内部元素，并使得JAR文件清单中的所有数字签名或校验码变为无效。

--strip-debug、-G

从Java类文件永久移去调试属性，而不是对它们进行压缩。这会使得要对最后产生的JAR文件进行调试会较为困难。

--verbose、-v

显示更多的输出信息。

--version、-V

显示版本编号并退出。

高级压缩选项

以下选项提供了在由*pack200*所执行的压缩上的精细控制。

Download at http://www.pin5i.com/

--deflate-hint=*value*、-H*value*

指定*pack200*是否应该保留输入的JAR文件中各个项目的紧缩状态。默认的*value*为keep，指将状态保留。如果*value*是true，就会在已压缩的归档文件中放入一个提示，指示解压缩程序在解压缩所有项目后应该再把它们紧缩。如果*value*是false，就会在已压缩的归档文件中放个提示，指示解压缩程序应该在不紧缩JAR文件中各个项目的情况下加以存储。*value*使用true或false可以稍微降低压缩文件的大小，因为紧缩提示不必针对每个项目来存储。

--modification-time=*value*、-m*value*

在使用默认的keep作为*value*时，*pack200*会留传JAR文件中每个项目的修改次数。如果你指定了latest，那就只会留传最近的修改时间，而且会在所有的项目解压缩时被应用。

--segment-limit=*n*、-S*n*

将目标片段（segment）大小设定为*n*。*pack200*文件可以被分割为独立的压缩片段，以便于降低解压缩程序所需的内存数量。此选项会设定各个片段的近似大小。默认值为1兆字节。值-1会产生单一型片段，而值0会对每个类文件产生一个片段。较大的片段大小会产生较佳的压缩率，但会需要额外的内存来解压缩。

--unknown-attribute=*action*、-U*action*

指示*pack200*应该如何处理未知的类文件属性。默认的*action*是pass，这指示整个类文件会在不压缩的情况下被留传。*action*若是error，则是指示*pack200*应该产生错误信息。*action*若是strip，则是说此属性应该要从类文件中移去。

--class-attribute=*name*=*action*、-C*name*=*action*

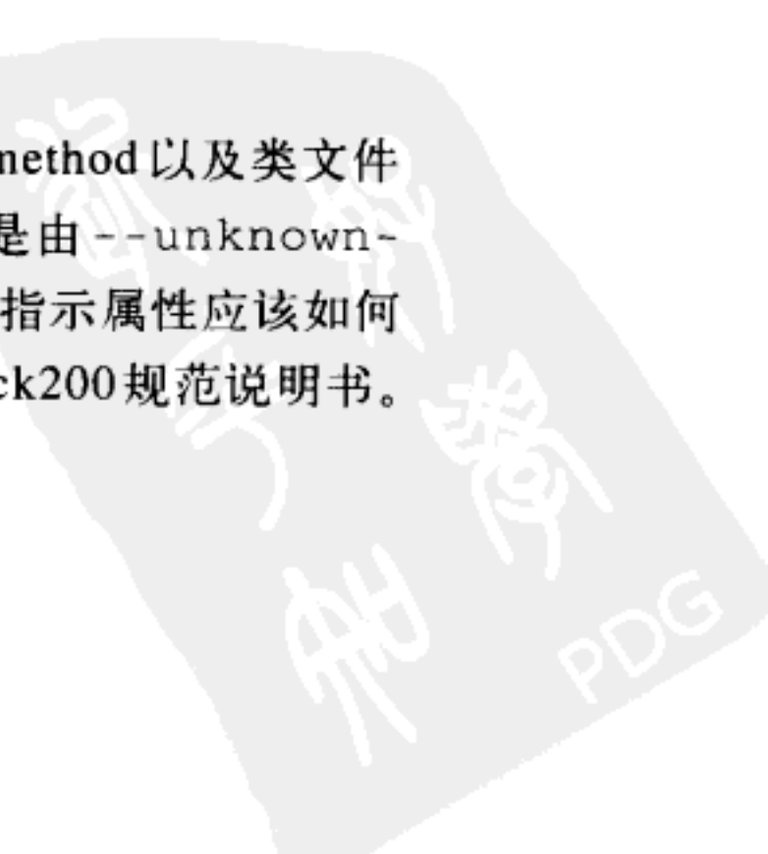
--code-attribute=*name*=*action*、-D*name*=*action*

--field-attribute=*name*=*action*、-F*name*=*action*

--method-attribute=*name*=*action*、-M*name*=*action*

这四个选项指定*pack200*应该如何处理特定的具名类、字段、method以及类文件中的程序代码属性。属性的名称是由*name*指定。*action*可以是由--unknown-attribute选项所支持的pass、strip和error值，也可以是指示属性应该如何被压缩的“布局字符串”。关于在布局语言上的细节，请参阅Pack200规范说明书。这些选项可以重复，以指定对多个属性的处理。

另可参阅 *unpack200*



Download at http://www.pin5i.com/

policytool 策略文件的创建与管理工具

提纲

```
policytool
```

说明

policytool 会显示 Swing 用户界面，使得安全策略配置文件的编辑变得容易。Java 的安全结构是以策略文件为基础，它指定了各种源代码允许的权限集。默认情况下，Java 的安全策略是由存储于*jre/lib/security/java.policy* 文件中的系统策略文件和存储于用户根目录的*.java.policy* 策略文件中的用户策略文件所定义的。系统管理员和用户可以用文本编辑器来编辑这些文件，但文件的语法有些复杂，所以使用 *policytool* 来定义和编辑安全策略通常会比较简单。

选择要编辑的策略文件

当*policytool* 启动时，默认情况下，它会开启用户根目录中的*.java.policy* 文件。使用 File 菜单中的 New、Open 和 Save 命令来分别创建新的策略文件、开启已存在的文件以及存储编辑过的文件。

编辑策略文件

*policytool*的主窗口会显示包含于政策文件中的项目列表。每个项目指定了一个程序代码来源和该来源的程序代码所拥有的权限。此窗口也包含了一些按钮，能让你增加新项目、编辑已存在的项目或删除策略文件中的项目。如果你新增或编辑一个项目，则*policytool* 会开启新的窗口来显示那个策略项目的细节。

随着 JAAS API 在 Java 1.4 中加入核心 Java 平台，*policytool* 允许 `Principal` 的规范，那是组要被授予的权限。

每个策略文件都有一个相关联的密钥存储，在验证Java源代码的数字签名时，策略文件就会从密钥存储取得所需的证书。你通常可以依赖默认的密钥存储，但如果你必须明确地为策略文件指定密钥存储，那么可以使用 *policytool* 主窗口的 Edit 菜单中的 Change Keystore 命令。

新增或编辑策略项目

策略项目编辑器窗口显示了策略项目的代码来源，以及与该代码来源相结合的权限列表。它也包含了一些按钮，能让你增加新的权限、删除权限或编辑已存在的权限。

Download at http://www.pin5i.com/

当定义新的策略项目时，第一步就是要指定程序代码来源。程序代码来源是由URL所定义的，就是下载程序代码和/或必须出现在程序代码中的数字签名列表的URL。通过输入URL和/或以逗号分隔的别名列表，就可以指定这些值。这些别名标识了在密钥存储中与策略文件相关联的可信任的证书。

在定义了策略项目的代码来源之后，你必须定义要被授权给来自那个来源的程序代码所拥有的权限。你可以使用Add Permission和Edit Permission按钮来增加和编辑权限。这些按钮会弹出另一个*policytool*窗口。

定义权限

如果要在权限编辑器窗口定义权限，首先要从Permission下拉菜单选择所要的权限类型，然后从Targer Name菜单选择适当的目标值，在此菜单中的选项是取决于你所选择的权限。有些类型的权限，例如FilePermission，没有固定的可能目标集合，那么你通常必须输入你要的目标。例如，你可以输入“/tmp/-”来指定*/tmp*目录，输入“/tmp/*”来指定那个目录以及子目录中的所有文件。关于它们所支持的目标的说明，请参阅各个`Permission`类的说明文件。

依据你所选择的权限类型，你也可以从Actions菜单选择一个或多个动作值。当你已选择权限和适当的目标与动作值时，就可以单击Okay按钮来关闭窗口了。

另可参阅 *jarsigner*、*keytool*

---

serialver 类版本编号产生器

提纲

```
serialver [ -classpath path ] classnames...
serialver [ -classpath path ] -show
```

说明

*serialver*会显示一个类或一些类的版本编号。这个版本编号是用于序列化：每当类的序列化格式改变时，版本编号就必须改变。

如果被指定的类声明了`long serialVersionUID`常量，那么该字段的值就会被显示。否则，具有唯一性的版本编号就会通过将Secure Hash Algorithm（SHA）应用至由类定义的API上来计算出来。这个程序对于计算类的初始唯一版本编号很有用，接着那个编号会被声明为类中的常量。*serialver*的输出是一行合法的Java程序代码，适合粘贴到类定义中。

Download at http://www.pin5i.com/

选项

-classpath *path*

指定类的搜索路径。

-show

当指定 -show 选项时，*serialver* 会显示简单的图形界面，能让用户一次输入一个类名并取得它的序列化 UID。当使用 -show 时，命令行上不可以指定类名。

环境

CLASSPATH

serialver 是用 Java 编写的，所以它对 CLASSPATH 环境变量的敏感度与 *java* 解释器相同。被指定的类会以相对于此 classpath 的方式来查询。

另可参阅

java.io.ObjectStreamClass

---

unpack200 解压缩 JAR 文件

提纲

unpack200 [*options*] *packedfile jarfile*

说明

unpack200 会解压缩被 *pack200* 工具压缩的 JAR 文件并选择性地以 *gzip* 来压缩。在命令行上指定压缩文件的名称和要对它解压缩的 JAR 文件的名称。

因为 *unpack200* 被作为 Java 安装程序的一部分，所以它是原生应用程序，可以在没有 Java 解释器的情况下在系统上运行。

选项

所有的 *unpack200* 选项都同时以详细形式和单字母形式存在，详细形式以双连字符开始，而单字母形式以单连字符开始。当选项必须要有值时，这个值在详细形式中应该要用等号与选项分隔而且不可以有空格；在单字母形式中应该要紧接在选项后面，中间不可以有空格或标点符号。

--deflate-hint=*value* -H*value*

指定 *unpack200* 是否应该压缩最后产生的 JAR 文件里的个别项目。*value* 必须是 true、false 或 keep。默认值为 keep，这指示每个 JAR 项目应该有与原来的 JAR 文件中相同的压缩形式。

Download at http://www.pin5i.com/

`--help、-h`

显示辅助说明信息并退出。

`--log-file=file、-lfile`

将输出记录至文件。

`--quiet、-q`

抑制输出信息。

`--remove-pack-file、-r`

在对压缩文件解压缩后，就将它删除。

`--verbose、-v`

显示更多的输出信息。

`--version、-V`

显示版本编号并退出。

另可参阅 *jar*、*pack200*

坚果系列

Download at http://www.pin5i.com/

JAVA技术手册

Java 5.0是自Java出现以来最重要的版本。随着对Java语言的主要修改和Java平台中重要的新API的出现，需要精通的新特性有很多。一如往常，《Java技术手册》（《Java in a Nutshell》）已经准备好了，而且有意愿、有能力来帮助你。

《Java技术手册》（《Java in a Nutshell》）以对Java程序设计语言的快速介绍开始，所以你可以立刻开始撰写程序代码。第五版已被彻底修订与扩充以涵盖Java 5.0的新语言特性：

- Generics
- Annotations
- For/in 循环
- Autoboxing
- Varargs method
- Static import

原书共一千两百余页，分为两大部分：第一部分为涵盖入门介绍、语法、面向对象程序设计、Java平台、安全性、程序设计与文档制作习惯、开发工具等章节的“Introducing Java”，秉承著作者David Flanagan一贯的言简意赅的风格，精要的文字能够让读者以更少的时间获得更多的收获；第二部分“API Quick Reference”占原书约三分之二的篇幅。考虑过一般开发者的习惯与查询便利性之后，为减轻读者书本重量上的负担以及购买的成本，中文译本已将与在线说明文档重叠的“API Quick Reference”部分删减。

《Java技术手册》（《Java in a Nutshell》）系列曾经荣获Jolt Award奖项，是一本有口皆碑的Java经典畅销书，历年来有许多程序员都是以阅读此书开始进入Java程序设计的殿堂的。

O'Reilly Media, Inc. 授权东南大学出版社出版

ISBN 7-5641-0560-7

9 787564 105600 >

O'REILLY®
www.oreilly.com.cn

ISBN 7-5641-0560-7

定价：48.00 元

A NUTSHELL® HANDBOOK

“坚果”（Nutshell）系列计算机技术手册是O'Reilly公司著名图书品牌。是由最权威的作者，以最精炼的语言，从最实用的角度撰写的内容全面、资料丰富的技术手册。多年来，坚果系列技术手册已成为全世界千千万万高级计算机用户、程序员和管理员日常工作中不可或缺的工具。