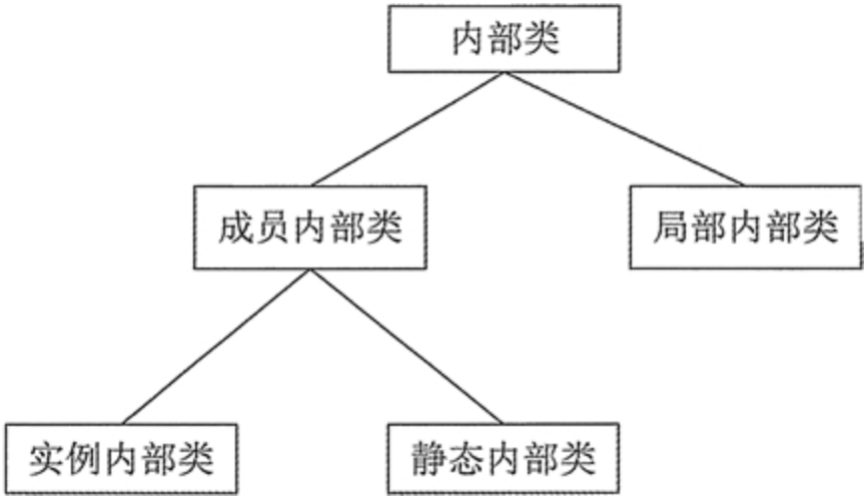
Java 高级特性

1 内部类

在类内部可定义成员变量和方法，且在类内部也可以定义另一个类。如果在类 Outer 的内部再定义一个类 Inner，此时类 Inner 就称为内部类（或称为嵌套类），而类 Outer 则称为外部类（或称为宿主类）。

内部类可以很好地实现隐藏，一般的非内部类是不允许有 private 与 protected 权限的，但内部类可以。内部类拥有外部类的所有元素的访问权限。

内部类可以分为：实例内部类、静态内部类和成员内部类。




```
public class Test {
    public class InnerClass {
        public int getSum(int x,int y) {
            return x + y;
        }
    }
    public static void main(String[] args) {
        Test.InnerClass ti = new Test().new InnerClass();
        int i = ti.getSum(2,3);
        System.out.println(i);    // 输出5
    }
}
```

- 内部类有以下特点：
  - **内部类可以访问外层类的所有（包括 private）成员。**
  - 内部类仍然是一个独立的类，在编译之后内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和$符号。
  - 内部类不能用普通的方式访问。内部类是外部类的一个成员。
  - **内部类声明成静态的，就不能随便访问外部类的成员变量，仍然是只能访问外部类的静态成员变量。**
  - 在外部类中可以直接通过内部类的类名访问内部类。
    - `InnerClass ic = new InnerClass(); // InnerClass为内部类的类名`
  - 在外部类以外的其他类中则需要通过内部类的完整类名访问内部类。
    - `Test.InnerClass ti = newTest().new InnerClass(); // Test.innerClass是内部类的完整类名`
  - 内部类与外部类不能重名。

1.1 静态嵌套类

可以在不创建外层类实例的情况下创建静态嵌套类。

```
class Outer1 {
    private static int value = 9;
    static class Nested1 {
        int calculate() {
            return value;
         }
    }
}

public class StaticNestedDemo1 {
    public static void main(String[] args) {
        Outer1.Nested1 nested = new Outer1.Nested1();
        System.out.println(nested.calculate());
    }
}
```

注意以下几点：

- 不需要创建外层类的实例就可以实例化静态嵌套类。
- 使用以下格式引用嵌套类：`OuterClassName.InnerClassName`

```
class Outer2 {
    private static int value = 9;
    static class Nested2 {
        int value = 10;
        int calculate() {
            return value;
        }
        int getOuterValue() {
            return Outer2.value;
        }
```

```
    }
}

public class StaticNestedDemo2 {
    public static void main(String[] args) {
        Outer2.Nested2 nested = new Outer2.Nested2();
        System.out.println(nested.calculate());     // 返回 10
        System.out.println(nested.getOuterValue()); // 返回 9
    }
}
```

1.2 成员内部类

成员内部类是一个类，该类的定义直接由另一个类或接口声明包围。仅在有一个外部类的实例引用时才能创建成员内部类的实例。**要在外层类中创建内部类的实例，可以像调用其他普通类一样调用内部类的构造方法。但是，要在外层类的层外部创建内部类的实例，可以使用以下语法：**

```
<外部类名>.<内部类名> inner = <外部类实例名>.new InnerClassName();
```

在内部类中，可以使用关键字 `this` 引用当前实例（内部类的实例）。要引用外层类的实例，可以使用 `<外部类名>.this` 。

成员内部类可用于完全隐藏一种实现，如果不使用内部类，则无法完成此操作。

```
interface Printer {
    void print(String message);
}
class PrinterImpl implements Printer {
    public void print(String message) {
        System.out.println(message);
    }
}
class SecretPrinterImpl {
    private class Inner implements Printer {
        public void print(String message) {
            System.out.println(「Inner:」 + message);
        }
    }
    public Printer getPrinter() {
        return new Inner();
    }
}
public class MemberInnerDemo2 {
    public static void main(String[] args) {
        Printer printer = new PrinterImpl();
        printer.print(”abc”);
        // 向下转换为 PrinterImpl
        PrinterImpl impl = (PrinterImpl) printer；

        Printer hiddenPrinter = (new SecretPrinterImpl()).getPrinter();
        hiddenPrinter.print(”abc”);
        // 无法将 hiddenPrinter 向下转换成 Outer.Inner
        // 因为 Inner 是 private 的
    }
}
```

1.3 局部类

局部内部类简称局部类（local class），是一个内部类，根据定义，**它不是任何其他类的成员类**（因为它的声明并不直接在外部类的声明中）。局部类有一个名称，而匿名类没有名称。

局部类可以在任何代码块中声明，它的作用域在这个块中，例如，可以在方法、 `if` 块、 `while` 块中声明一个局部类。如果某个类的实例仅在这个作用域内使用，则可以将这个类定义成局部类。

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.FormatStyle;

interface Logger {
    public void log(String message);
}

public class LocalClassDemo1 {
    // 一个非静态的成员
    String appStartTime = LocalDateTime.now().format(
            DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM));

    public Logger getLogger() {
        //在方法中声明一个局部类，这个局部类可以访问外部类的非静态成员
        class LoggerImpl implements Logger {
            public void log(String message) {
                System.out.println(appStartTime + " : " + message);
            }
        }
        return new LoggerImpl();
    }

    public static void main(String[] args) {
        LocalClassDemo1 test = new LocalClassDemo1();
        Logger logger = test.getLogger();
        logger.log("Local class example");
    }
}
```

局部类不仅可以访问其外层类的成员，还可以访问局部变量。但是，局部类只能访问 `final` 局部变量。

```
    public PrefixLogger getLogger(final String prefix) {
        class LoggerImpl implements PrefixLogger {
            public void log(String message) {
                System.out.println(prefix + " : " + message);
            }
        }
        return new LoggerImpl();
    }
```

1.4 匿名内部类

```
interface Printable {
    void print(String message);
}

public class AnonymousInnerClassDemo1 {
    public static void main(String[] args) {
```

```
        Printable printer = new Printable() {
            public void print(String message) {
                System.out.println(message);
            }
        };      // 注意，这里是一个分号
        printer.print(「Beach Music」);
    }
}
```

2 Lambda 和 Stream

Lambda 表达式也称为闭包（closures），它可以使某些 Java 代码结构更简短，更易于阅读，尤其是在使用内部类时。

2.1 函数式接口

函数式接口是只有一个抽象方法的接口，该抽象方法不能是覆盖 `java.lang.Object` 类的方法。函数式接口也称为单抽象方法接口。例如， `java.lang.Runnable` 是一个函数式接口，它只有一个 `run` 抽象方法。函数式接口可以有任意数量的默认方法和静态方法，也可以有覆盖 `java.lang.Object` 的公共方法。例如，如下的 `Calculator` 接口是一个函数式接口：

```
public interface Calculator {
    double calculate(int a, int b);
    public default int subtract(int a, int b) {
        return a – b;
    }
    public default int add(int a, int b) {
        return a * b;
    }
    @Override
    public String toString();
}
```

`Calculator` 接口有一个 `calculate` 方法，可以作为 Lambda 表达式的基础。该方法允许定义接收两个整数并返回一个 `double` 值的任何数学运算。下面是两个基于 `Calculator` 的 `Lambda` 表达式：

```
Calculator addition = （int a, int b） –> （a + b）;
System.out.println（addition.calculate(5, 20)）; // 打印 25.0
Calculator division = （int a, int b） –> （double） a / b;
System.out.println（division.calculate(5, 2); // 打印 2.5
```

2.2 语法

这里的语句组实际上就是实现接口的函数内容。当只有一条语句时，直接写在后面即可不需要表达式。

```
(parameter–list) –> expression
    (parameter–list) –> {
    语句组
}
```

其中，parameter–list是与函数式接口抽象方法的参数相同的参数列表。但是，每个参数的类型是可选的。换句话说，下面两个表达式是等价的：

```
Calculator addition = (int a, int b) –> (a + b);
Calculator addition = (a, b) –> (a + b);
```

常见的Lambda 表达式比如：`() -> {return;}` ，它可以被任何一个没有参数的函数式接口接受。

2.3 预定义函数式接口

- **Predicate**

`Predicate` 是一个函数，用于接收一个参数并根据该参数的值返回 `true` 或 `false` ，它有一个名为 `test` 的抽象方法。如下的 `PredicateDemo1` 类定义了一个 `Predicate` ，用于计算输入字符串，如果字符串中的每个字符都是数字，则返回 `true` 。

```
import java.util.function.Predicate;
public class PredicateDemo1 {
    public static void main(String[] args) {
        Predicate<String> numbersOnly = (input) -> {
            for (int i = 0; i < input.length(); i++) {
                char c = input.charAt(i);
                if ("0123456789".indexOf(c) == -1) {
                    return false;
                }
            }
            return true;
        };

        System.out.println(numbersOnly.test("12345"));  // true
        System.out.println(numbersOnly.test("100a"));   // false
    }
}
```

2.3 流式计算

- **创建流**

使用 `Stream` 的 `of` 静态方法可以创建一个顺序流，还可以将数组传递给 `of` 方法。 `Arrays` 类也有一个 `stream` 方法，该方法可将数组转换为顺序流。

```
Stream<Integer> stream = Stream.of(100, 200, 300);

String[] names = {"Bart", "Lisa", "Maggie"};
Stream<String> stream = Stream.of(names);

String[] names = {"Bart", "Lisa", "Maggie"};
Stream<String> stream = Arrays.stream(names);
```

`Collection` 接口提供了名为**`stream`**和 `parallelStream` 的默认方法，这两个方法分别返回顺序流和并行流，它们都将集合作为数据源。

- **过滤**

过滤一个流，是根据某些条件选择流中的元素，并针对所选元素返回一个新的 `Stream` 。其中生成新的流这一步是在 `count` 的过程中才进行的。 `filter` 可以认为仅仅绑定了一个过滤器到 `stream` 上。

```
int count = allArtlists.stream()
    .filter(artlist -> artlist.isFrom("London")).count();
```

- **映射**

`Stream` 接口的 `map` 方法对流的每个元素进行映射，并将元素传递给函数。

```
List<String> collected = Stream.of(”a”, ”b”, ”hello”)
    .map(string –> string.toUppercase()).collect(toList());
```

- **规约**

stream api的 `reduce` 方法用于对stream中元素进行聚合求值，最常见的用法就是将stream中一连串的值合成为单个值，比如为一个包含一系列数值的数组求和。

```
List<Integer> numList = Arrays.asList(1,2,3,4,5);
int result = numList.stream().reduce((a,b) –> a + b ).get();
System.out.println(result);
```

代码实现了对numList中的元素累加。lambada表达式的 `a` 参数是表达式的执行结果的缓存，也就是表达式这一次的执行结果会被作为下一次执行的参数，而第二个参数 `b` 则是依次为stream中每个元素。如果表达式是第一次被执行，`a` 则是stream中的第一个元素。

```
int result = numList.stream().reduce((a,b) –> {
  System.out.println(”a=” + a + ”,b=” + b);
  return a + b;
} ).get();

a=1,b=2
a=3,b=3
a=6,b=4
a=10,b=5
```

改进：它首次执行时表达式第一次参数并不是stream的第一个元素，而是通过签名的第一个参数identity来指定。

```
List<Integer> numList = Arrays.asList(1,2,3,4,5);
int result = numList.stream().reduce(0,(a,b) –>  a + b );
System.out.println(result);
```

- IntStream.range()产生指定区间的有序IntStream，这里需要传入一个区间（左闭右开），产生的元素不包含最后一个。

3 Enum枚举

枚举类型可以单独定义，也可以在类中定义。如果需要在应用程序中的多个位置引用它，那么可以单独定义枚举；如果只在某个类中使用，则最好在类中定义。

`CustomerType` 枚举有两个枚举值： `INDIVIDUAL` 和 `ORGANIZATION` 。枚举值区分大小写，按惯例均为大写。两个枚举值用逗号分隔，值可以写在一行或多行上。枚举值写在多行上是为了提高可读性。 在内部，每个枚举常量被赋予一个整数序号，第一个常量是 0。对于 `CustomerType` 枚举， `INDIVIDUAL` 的序号是 0， `ORGANIZATION` 的序号是 1。枚举序号很少用到。

```
public enum CustomerType {
    INDIVIDUAL,
    ORGANIZATION
}
public class Customer {
```

```
    public String customerName;
    public CustomerType customerType;
    public String address;
}
customer.customerType = CustomerType.INDIVIDUAL;
```

如果枚举只在一个类内部使用，可以将枚举定义为类的成员。例如， `Shape` 类中定义了一个 `ShapeType` 枚举。

```
public class Shape {
    private enum ShapeType {
        RECTANGLE, TRIANGLE, OVAL
    };
    private ShapeType type = ShapeType.RECTANGLE;
    public String toString() {
        if (this.type == ShapeType.RECTANGLE) {
            return ”Shape is rectangle”;
        }
        if (this.type == ShapeType.TRIANGLE) {
            return ”Shape is triangle”;
        }
        return ”Shape is oval”;
    }
}
```

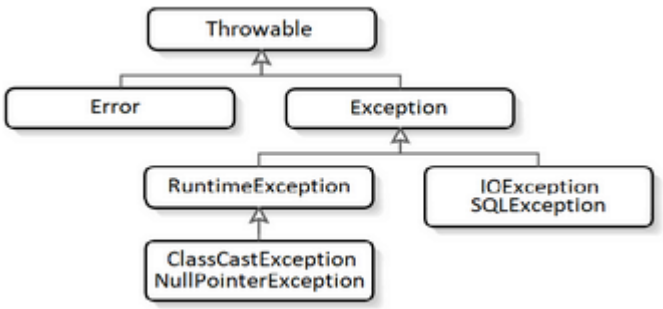
3.1 迭代

可以使用for循环迭代枚举中的值。

```
for (CustomerType customerType : CustomerType.values() ) {
    System.out.println(customerType);
}
```

4 错误处理

Java 中的错误通常称为异常（exception），这里的异常是异常事件（exceptional event）的缩写。在 Java 中有 3 种类型的异常，异常类层次结构如图所示：



- **错误。** `Error` 表示一种严重的问题，我们不应该尝试处理这种问题，因为这超出了我们的控制范围，而且这不是我们的错误造成的。例如，如果程序试图读取由硬件故障引起的文件，会发生输入输出错误。另一个例子是 `OutOfMemoryError` ，当 Java 虚拟机因内存不足而无法分配对象时，程序就会抛出该错误。在大多数情况下，在错误发生时，应允许程序突然退出。
- **运行时异常或未检查异常。**运行时异常或未检查异常指可能在运行时发生的异常，**不需要在方法中指定它。程序中可以选择捕获处理，也可以不处理。**例如，如果尝试对 `null` 对象调用方法，系统将抛出 `NullPointerException`

异常。又如，如果某个数组包含 5 个元素，但却试图访问它的第 6 个元素，系统将抛出 `IndexOutOfBoundsException` 异常。

- **检查异常**。检查异常是**在编译过程中应该预料到并从中恢复的异常**。它基本上是所有派生自 `Exception` 的子类，但不是 `RuntimeException` 的子类，**如`IOException`异常和`SQLException`异常。编译器可以通过拒绝编译任何包含可能引发已检查异常的程序来防止发生已检查异常。**

4.1 Exception类

在 Java 中所有异常类型都是内置类 `java.lang.Throwable` 类的子类，即 `Throwable` 位于异常类层次结构的顶层。`Throwable` 类下有两个异常分支 `Exception` 和 `Error` 。

运行时异常都是 `RuntimeException` 类及其子类异常，如 `NullPointerException` 、 `IndexOutOfBoundsException` 等。
非运行时异常是指 `RuntimeException` 以外的异常，类型上都属于 `Exception` 类及其子类。

`Exception` 类覆盖 `toString` 方法并添加 `printStackTrace` 方法， `toString` 方法返回异常的描述。

`printStackTrace` 方法具有以下签名： `public void printStackTrace()` 。该方法将打印异常的描述，后跟 `Exception` 对象的堆栈跟踪。通过分析堆栈跟踪，我们可以找出导致问题的那一行。下面是 `printStackTrace` 可能在控制台上打印的示例：

```
java.lang.NullPointerException
    at MathUtil.doubleNumber(MathUtil.java:45)
    at MyClass.performMath(MyClass.java: 18)
    at MyClass.main(MyClass.java: 90)
```

通过继承 `java.lang.Exception` 类，我们可以创建用户自定义异常。

```
public class AlreadyCapitalizedException extends Exception {
   @Override
   public String toString() {
      return ”Input has already been capitalized”;
    }
}
```

4.2 捕获异常

用 `try` 语句隔离可能抛出已检查异常的代码。 `try` 语句通常与 `catch` 语句和 `finally` 语句一起使用。这种隔离通常出现在方法体中。如果遇到错误，Java 将停止 `try` 块的处理并跳转到 `catch` 块。在这里，可以优雅地处理错误，或者通过抛出相同的或另一个 `Exception` 对象来通知用户。如果选择后者，则由客户处理错误。如果抛出的异常没有被捕获，应用程序将崩溃。

`finally` 中的代码无论是否发生错误，将始终被执行。

```
try {
   [可能抛出异常的代码]
} [catch (ExceptionType-1 e) {
   [抛出ExceptionType-1异常执行的代码]
}] [catch (ExceptionType-2 e) {
   [抛出ExceptionType-2异常执行的代码]
}]
  ...
} [catch (ExceptionType-n e) {
   [抛出ExceptionType-n异常执行的代码]
}]
[finally {
```

```
    [不论是否抛出异常，都将执行的代码]]
}]
```

与C++类似，当抛出一个异常后，**程序暂停当前函数的执行过程并立即开始寻找与异常匹配的 catch 子句**。如下的代码当不存在 `Hello.txt` 时，输出 `No such file found, Doing finally, Exception in thread "main" java.io.FileNotFoundException` 。

```
public int ff() throws FileNotFoundException {
    try {
        FileInputStream dis=new FileInputStream("Hello.txt");
    } catch (FileNotFoundException fne) {
        System.out.print("No such file found, ");
        throw fne;
        System.out.println("QAQ"); // 永远不会被执行
    } finally {
        System.out.print("Doing finally, ");
    }
    return 0;
}
```

在 Java 7 和更高的版本中，如果捕获的多个异常由相同的代码处理，则允许在一个catch块中捕获多个异常。catch块的语法如下，多个异常由管道字符 `|` 分隔：

```
catch(exception-1 | exception-2 ... e) {
    // 处理异常
}
```

> `catch` 块和 `finally` 块是可选的，二者必须至少有一个，但在 `try-with-resources` 结构中除外。因此，一个 `try` 块可以带一个或多个 `catch` 块，可以带一个 `finally` 块，也可以同时带 `catch` 块和 `finally` 块。

注意到 `parseDouble` 方法的签名如下，其中的 `throws` 语句表明它可能抛出 `NumberFormatException` 异常，而捕获该异常是方法调用者的责任。

`public static double parseDouble(String s) throws NumberFormatExcpetion`

```
public class NumberDoubler {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String input = scanner.next();
        try {
            double number = 2 * Double.parseDouble(input);
            System.out.printf("Result: %s", number);
        } catch (NumberFormatException e) {
            System.out.println("Invalid input.");
        }
        scanner.close();
    }
}
```

- **从方法中抛出异常**

从方法中抛出异常时，调用代码必须捕获由方法抛出的异常。 `throws` 语句表明方法可能抛出指定类型的异常。给出一个 `capitalize` 方法，它将参数 `String` 的首字母改成大写的形式：

```
public String capitalize(String s) throws NullPointerException {
    if (s == null) {
        throw new NullPointerException(
                "You passed a null argument");
```

```
    }
    Character firstChar = s.charAt(0);
    String theRest = s.substring(1);
    return firstChar.toString().toUpperCase() + theRest;
}
```

> 当对一个方法进行重写时，只能抛出基类版本中指定的异常。（no more）
>
> 但当重写的是父类的构造方法时，可以增加抛出新的异常。（no less）

4.3 关闭资源

许多 Java 操作都涉及使用某种资源，使用完这些资源后必须关闭这些资源。在 JDK 7 之前，通常用 `finally` 确保调用了 `close` 方法：

```
try {
    // 打开资源
} catch （Exception e） {

} finally {
    // 关闭资源
}
```

即使只处理一个资源，也需要在 `finally` 块中写多行代码，何况经常需要在一个 `try` 块中打开多个资源。JDK 7 增加了 `try-with-resources` 语句，它可使资源自动关闭。其语法如下：

```
try ( resources ) {
    // 使用这些资源执行某些操作
} catch (Exception e) {
    // 用 e 执行某些操作
}
```

例如，在 Java 7 或更高的版本中打开数据库连接的代码如下：

```
Connection connection = null;
try (Connection connection = openConnection(); ) { //  如果还有其他资源，也打开
    // 使用 connection 执行操作
} catch (SQLException e) {

}
```

并不是所有的资源都可自动关闭。只有实现 `java.lang.AutoCloseable` 接口的资源类才可自动关闭。幸运的是，JDK 7 中修改了许多输入/输出和数据库资源来支持该特性。如下的例子中输入流将会被自动关闭：

```
try (FileInputStream inputStream = newFileInputStream(new File(”test”))) {
        System.out.println(inputStream.read());
} catch (IOException e) {
        throw new RuntimeException(e.getMessage(), e);
}
```

Java 9 再次增强了这种 `try` 语句。Java 9 不要求在 `try` 后的圆括号内声明并创建资源，只需要自动关闭的资源有 `final` 修饰或者是有效的 `final (effectively final)`，Java 9 允许将资源变量放在 `try` 后的圆括号内。

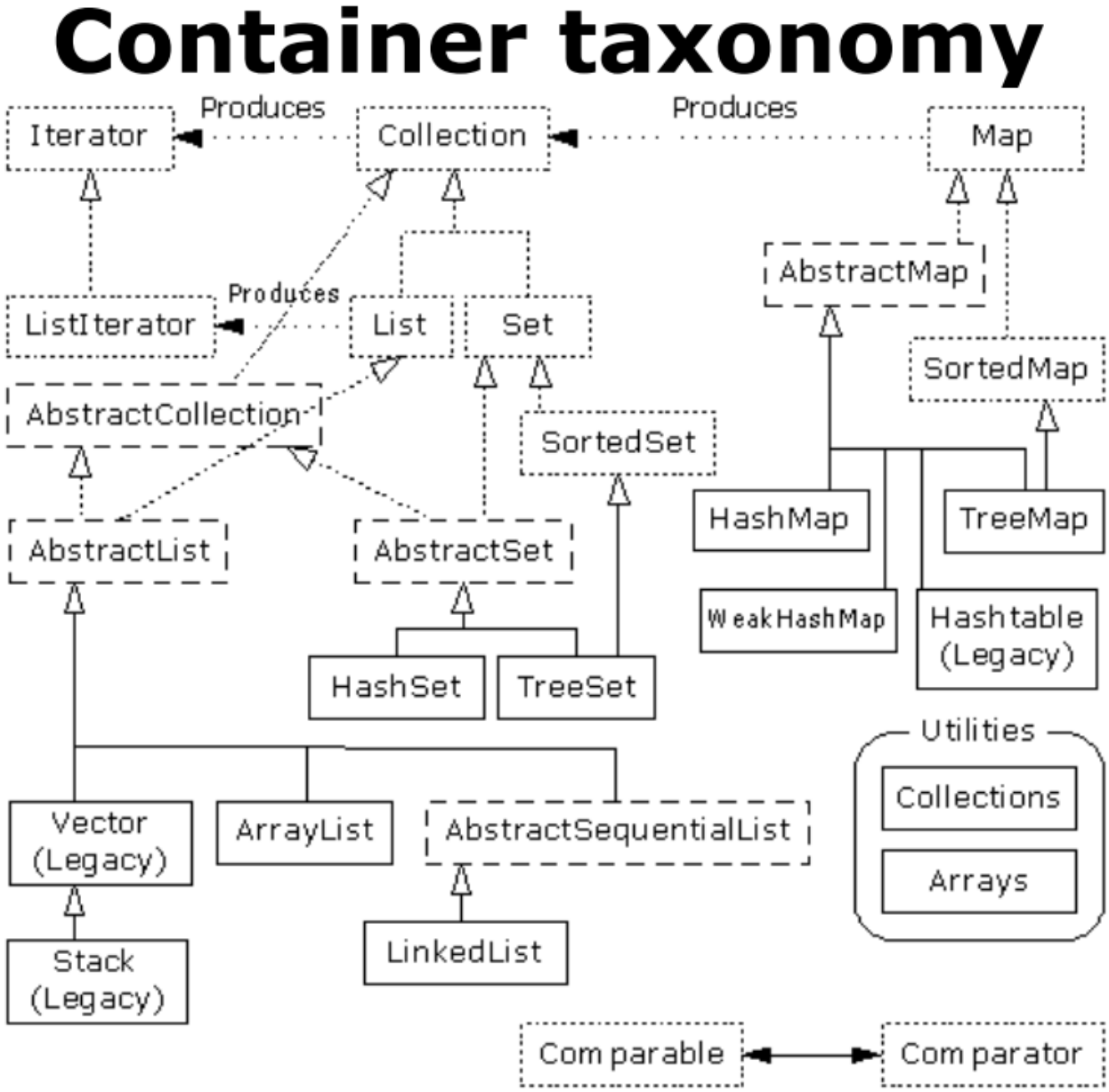
```
public class AutoCloseTest {
    public static void main(String[] args) throws IOException {
        // 有final修饰的资源
```

```
        final BufferedReader br = new BufferedReader(new FileReader("AutoCloseTest.java"));
        // 没有显式使用final修饰，但只要不对该变量重新赋值，该变量就是有效的
        final PrintStream ps = new PrintStream(new FileOutputStream("a. txt"));
        // 只要将两个资源放在try后的圆括号内即可
        try (br; ps) {
            // 使用两个资源
            System.out.println(br.readLine());
            ps.println("C语言中文网");
        }
    }
}
```

5 集合框架和泛型

5.1 集合框架

毋庸置疑，集合框架中的主要类型是 `Collection` 接口。 `List` 、 `Set` 和 `Queue` 是 `Collection` 的 3 个子接口。此外，还有一个 `Map` 接口可用于存储键/值对。Map的子接口 `SortedMap` 保证键按升序排列。Map的其他实现还有 `AbstractMap` 及其具体实现 `HashMap` 。其他接口包括 `Iterator` 和 `Comparator` ，后者用于使对象可排序和可比较。



`Collection` 提供了许多易于使用的方法。要添加元素，使用 `add` 方法；要添加另一个 `Collection` 的成员，使用 `addAll` 方法；要删除所有元素，使用 `clear` ；要获取集合中的元素数量，使用 `size` 方法；要测试集合是否包含元素，使用 `isEmpty` ；要将它的元素移动到数组中，使用 `toArray` 方法。 需要注意的重点是， `Collection` 扩展了 `Iterable` 接口，从该接口继承了 `iterator` 方法，此方法将返回一个 `Iterator` 对象，可以用该对象迭代集合的元素。

List ArrayList

`List` 是有序元素的集合，也称为序列（sequence）。可以使用索引访问它的元素，也可以将元素插入到指定的位置。

`List` 是一个接口，而 `ArrayList` 是 `List` 接口的一个实现类。 `ArrayList` 类继承并实现了 `List` 接口。

因此， `List` 接口不能被构造，也就是我们说的不能创建实例对象，但是可以通过继承自本接口的实现类的对象实例化 `List` 对象，而 `ArrayList` 实现类的实例对象就在这充当了这个指向 `List` 接口的对象引用。 `List list=new ArrayList();`

`ArrayList` 的无参数构造方法创建一个 `ArrayList` 对象，列表的初始容量为 10 个元素。当添加的元素超过其容量时，列表的大小将自动增加。如果知道 `ArrayList` 中的元素数量将超过默认的容量，可以使用另一个构造方法： `public ArrayList(int initialCapacity)`

下面展示一些方法：

- `public boolean add(java.lang.Object element)`
- `public void add(int `index`, java.lang.Object element)`
- `public java.lang.Object set(int index, java.lang.Object element)`
- `public java.lang.Object remove(int index)`

```
import java.util.ArrayList;
import java.util.List;

public class ListDemo1 {
    public static void main(String[] args) {
        List myList = new ArrayList();
        String s1 = ”Hello”;
        String s2 = ”Hello”;
        myList.add(100);
        myList.add(s1);
        myList.add(s2);
        myList.add(s1);
        myList.add(1);
        myList.add(2, ”World”);
        myList.set(3, ”Yes”);
        myList.add(null);
        System.out.println(”Size: ” + myList.size());
        for (Object object : myList) {
            System.out.println(object);
        }
    }
}
```

输出：Size: 7 100 Hello World Yes Hello 1 null

- 我们为什么不直接使用 `ArrayList list = newArrayList();` 的方式来实例化 `ArrayList` 对象，并调用其中的方法，而有时候大部分会通过 `List list = new ArrayList();` 来实例化对象呢?

    1. 两种实例化的方式有区别，第一种方式创建的对象继承了 `ArrayList` 的所有属性和方法。 而第二种方式创建的对象只能调用 `List` 接口中的方法，不能调用 `ArrayList` 中的方法。
    2. 使用第二种方式创建的对象更为方便。 `List` 接口有多个实现类，现在用的是 `ArrayList` ，如果要将其更换成其它的实现类，如 `LinkedList` 或者 `Vector` 等等，这时只需要改变这一行就行了： `List list = new LinkedList();`

- `java.util.Vector` 类和 `ArrayList` 类都是 `List` 接口的实现。 `Vector` 和 `ArrayList` 都提供了类似的功能，但 `Vector` 是同步的，而 `ArrayList` 是非同步的。非同步版本应该优先于同步版本，因为前者更快。如果需要在多线程环境中使用非同步实现，仍然可以自己将它同步。

Iterator和迭代

`Collection` 接口扩展了 `Iterable` 接口，它只有一个 `iterator` 方法，这个方法返回可用于在 `Collection` 上迭代的 `java.util.Iterator` 对象。

Iterator 的方法如下：

- hasNext
  - **Iterator的内部指针最初指向第一个元素之前的位置**。如果指针后面有更多元素，则 hasNext 返回 true 。第一次对 Iterator 调用 next 会导致其指针指向第一个元素。
- next
  - 该方法**将内部指针移动到下一个元素并返回该元素**。在返回最后一个元素后调用 next 将抛出 java.util.NoSuchElementException 异常。因此，在调用 next 之前，应调用 hasNext 测试一下是否有下一个元素。
- remove
  - 该方法将删除内部指针指向的元素。
- 可以使用 iterator 或 for 循环进行迭代。

```
Iterator iterator = myList.iterator();
while (iterator.hasNext()) {
    String element = (String) iterator.next();
    System.out.println(element);
}

for (Iterator iterator = myList.iterator(); iterator.hasNext(); ) {
    String element = (String) iterator.next();
    System.out.println(element);
}

for (Type identifier : expression) {
    statement(s)
}
```

Set HashSet

Set 类似于数学中的集合。 Set 不允许包含重复元素，如果试图添加重复元素， Set 的 add 方法将返回 false 。

Set 的某些实现最多允许一个 null 元素，有些不允许有 null 。例如， Set 最流行的实现 HashSet 最多允许一个 null 元素。注意，在使用 HashSet 时，不能保证元素的顺序一直不变。 HashSet 应该是首选，因为它比 Set 的其他实现（如 TreeSet 和 LinkedHashSet ）更快一些。

```
Set set = new HashSet();
set.add(”Hello”);
if (set.add(”Hello”)) {
    System.out.println(”addition successful”);
} else {
    System.out.println(”addition failed”);
}
```

Map 和 HashMap

```
public void put(java.lang.Object key, java.lang.Object value)
    // 在已经存在这个键时，会将value替换为新值
public void remove(java.lang.Object key)
public java.lang.Object get(java.lang.Object key)
```

还有 3 个无参数方法。

- `keySet`，返回 `Map` 中包含所有键的 `Set`。
- `values`，返回 `Map` 中包含的所有值的 `Collection`。
- `entrySet`，返回一个包含 `Map.Entry` 对象的 `Set`。每个 `Entry` 对象表示一个键/值对。`Map.Entry` 接口提供返回键的 `getKey` 方法和返回值的 `getValue` 方法。
- `containsKey(Object key)` 如果 Map 包含指定键的映射，则返回 true
- `containsValue(Object Value)` 如果此 Map 将一个或多个键映射到指定值，则返回 true
- `isEmpty()`
- `size()`

`Map` 的实现最常用的是 `HashMap` 和 `Hashtable`。`HashMap` 是非同步的，而 `Hashtable` 是同步的，因此 `HashMap` 是两者之间更快的一个。

> - 如果要将自定义类型作为 `Hashmap` 的 `key`，需要重写 `hashCode( )` 和 `equals()` 方法。

遍历 `map` 的方法有很多，这里提供遍历最常用的 `entry` 的方法：

```
public static void main(String[] args) {
    Map<String, String> map = new HashMap<String, String>();
    // map.put("Java入门教程", "http://c.biancheng.net/java/");
    // map.put("C语言入门教程", "http://c.biancheng.net/c/");
    for (Map.Entry<String, String> entry : map.entrySet()) {
        String mapKey = entry.getKey();
        String mapValue = entry.getValue();
        System.out.println(mapKey + "： " + mapValue);
    }
}
```

Collections类

Collections 类是 Java 提供的一个操作 Set、List 和 Map 等集合的工具类。Collections 类提供了许多操作集合的静态方法，借助这些静态方法可以实现集合元素的排序、查找替换和复制等操作。下面介绍 Collections 类中操作集合的常用方法。

- 排序相关
    - `void reverse(List list)`：对指定 List 集合元素进行逆向排序。
    - `void shuffle(List list)`：对 List 集合元素进行随机排序（shuffle 方法模拟了“洗牌”动作）。
    - `void sort(List list)`：根据元素的自然顺序对指定 List 集合的元素按升序进行排序。
    - `void sort(List list, Comparator c)`：根据指定 Comparator 产生的顺序对 List 集合元素进行排序。
    - `void swap(List list, int i, int j)`：将指定 List 集合中的 i 处元素和 j 处元素进行交换。
    - `void rotate(List list, int distance)`：当 distance 为正数时，将 list 集合的后 distance 个元素“整体”移到前面；当 distance 为负数时，将 list 集合的前 distance 个元素“整体”移到后面。该方法不会改变集合的长度。

```
public static void main(String[] args) {
    ArrayList<Integer> arrayList = new ArrayList<>();
```

```
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(5);
    Comparator comparator = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
                    return o2 – o1;
        }
    };
    Collections.sort(arrayList, comparator);
    for (Integer a:arrayList ) {
        System.out.println(a);
    }
}
```

5.2 泛型

在 Java 5 或更高的版本中仍可以使用不带类型参数的泛型类型，如下所示。 `get` 方法返回的类型是 `Object` ，因此需进行向下转换。

```
List stringList1 = new ArrayList();
stringList1.add(1);
stringList1.add(”without generics”);
String s1 = (String) stringList1.get(0);
```

新的Java版本希望使用泛型类型。要实例化泛型类型，需要传递与声明泛型类型时相同的参数列表，例如，要创建一个使用 `String` 的 `ArrayList` ，需要在尖括号中传递一个 `String` 参数：

```
List<String> myList = new ArrayList<String>();
List<String> myList = new ArrayList<>(); // 在Java 7以后，可以在构造方法中省略泛型类的类型参数
```

通配符

泛型类型本身也是 Java 类型，如下面的 `list1` 和 `list2` 引用了不同类型的对象：

```
List<Object> list1 = new ArrayList<>();
List<String> list2 = new ArrayList<>();
```

尽管 `String` 是 `Object` 的子类，但 `List<String>` 与 `List<Object>` 并没有任何关系。因此，把 `List<String>` 传递给希望得到 `List<Object>` 参数的方法会引发编译错误。解决这个问题的办法是使用通配符 `?` 。可以用 `List<?>` 表示任意类型的List对象。

```
public class WildCardDemo1 {
    public static void printList(List<?> list) {
        for (Object element : list) {
            System.out.println(element);
       }
    }
    public static void main(String[] args) {
        List<String> list1 = new ArrayList<>();
        list1.add(”Hello”);
        list1.add(”World”);
        printList(list1);

        List<Integer> list2 = new ArrayList<>();
        list2.add(100);
        list2.add(200);
```

```
        printList(list2);
    }
}
```

- **有界通配符**

使用通配符 `?` 时，会使得无法在编译时进行类型安全性检查，因为这意味着可以传递任何类型的 `List` 。为了解决这个问题，我们可以允许定义类型变量的上界（upper bound）。通过这种方式，就可以传递一个类型或它的子类型。对于 `getAverage` 方法，就可能传递一个 `List<Number>` 或 `Number` 子类的 `List` 实例，例如 `List<Integer>` 或 `List<Float>` 。

```
public class BoundedWildcardDemo1 {
    public static double getAverage(
            List<? extends Number> numberList) {
        double total = 0.0;
        for (Number number : numberList) {
            total += number.doubleValue();
        }
        return total/numberList.size();
    }
   public static void main(String[] args) {
        List<Integer> integerList = new ArrayList<>();
        integerList.add(3);
        integerList.add(30);
        integerList.add(300);
        System.out.println(getAverage(integerList));     // 111.0
        List<Double> doubleList = new ArrayList<>();
        doubleList.add(3.0);
        doubleList.add(33.0);
        System.out.println(getAverage(doubleList));     // 18.0
   }
}
```

泛型方法

6 输入 / 输出

> Java NIO API 是java.nio包及其子包的一部分。JDK 7 新引入了一系列被称为 NIO.2 的包对现有技术进行补充。虽然没有 java.nio2 包，但可以在 `java.nio.file` 包及其子包中找到新的类型。NIO.2 的一个特性是 `Path` 接口，它的设计目的是取代 `java.io.File` 类。

> **用`Files`类读取和写入文件只适用于小文件。对于较大的文件以及新增的功能，需要使用流**，它就像水管一样，用于实现数据传输。有 4 种类型的流： `InputStream` 、 `OutputStream` 、 `Reader` 和 `Writer` 。为了获得更好的性能，还有一些类可以封装这些流并缓冲正在读取或写入的数据。 从流中读取和写入流要求按顺序执行，这意味着要读取第二个数据单元，必须先读取第一个。要想随机访问文件，换句话说，想随机访问文件的任何部分，需要一个不同的 Java 类型。 `java.io.RandomAccessFile` 类过去是非顺序操作的良好选择，但是现在更好的方法是使用 `java.nio.channel.SeekableByteChannel` 。

6.1 File类

File 类中的 separator 静态字段提供了基于操作系统的文件路径分隔符。Linux下是 / ，windows下是 \ ，类似的， file.charSeparator 返回一个char。

在 Java 中，File 类是 java.io 包中唯一代表磁盘文件本身的对象，也就是说，如果希望在程序中操作文件和目录，则都可以通过 File 类来完成。File 类定义了一些方法来操作文件，如新建、删除、重命名文件和目录等。

File 类不能访问文件内容本身，如果需要访问文件内容本身，则需要使用输入/输出流。

File 类提供了几个构造方法，如：

```
public File(java.lang.String pathname) // 绝对/相对路径
File file1 = new File("C:\\temp\\myNote.txt"); // 在 Windows 系统上
File file2 = new File("/tmp/myNote.txt"); // 在 Linux/Unix 系统上
File file3 = new File("music"); // 相对路径
```

下面是File类的较重要的方法。

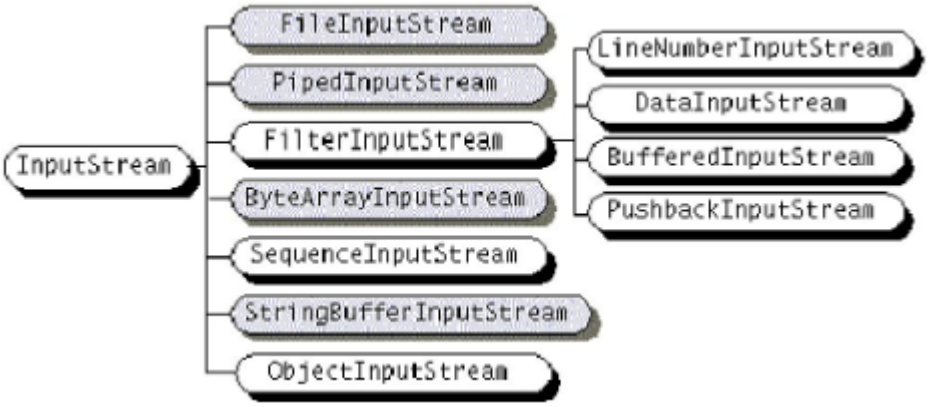
- public boolean canRead() 用于测试应用程序是否可以读取此File引用的文件。
- public boolean canWrite() 用于测试应用程序是否可以写入此File引用的文件。
- public boolean createNewFile() throws IOException 用此File指定的名称，在当前位置创建一个空的新文件。
- public boolean delete() 删除此File引用的文件或目录。
- public boolean makeDir() 创建此File命名的目录。
- public boolean isFile() 测试此File是否引用了文件。
- public boolean isDirectory() 测试此File是否引用了目录。
- public boolean exists() 测试此File所表示的文件或目录是否存在。
- public File[] listFiles() 如果此File表示一个目录，方法将返回一个File对象数组，该数组用于引用目录中的子目录和文件；否则，返回 null。

6.2 输入输出流

字节流

- InputStream

可以用 InputStream 从接收装置读取二进制数据。 InputStream 是一个抽象类，有许多具体的子类。作为字节输入流的直接或间接的父类，定义了许多有用的、所有子类必需的方法，包括读取、移动指针、标记、复位、关闭等方法。需要注意的是，这些方法大多可能抛出 IOException 异常。



InputStream 的核心是 3 个重载的 read 方法：

```
// 读取一个字节，返回值为所读的字节
public int read()
// 读取多个字节放置到字节数组中，读取的字节数量为 b 的长度，返回值为实际读取的字节的数量
public int read(byte[] data)
// 读取 len 个字节，放置到以下标 off 开始的字节数组 b 中，返回值为实际读取的字节的数量
public int read(byte[] data, int offset, int length)
```

`InputStream` 使用一个内部指针，该指针指向要读取的数据的起始位置。每个重载 `read` 方法都会返回读取的字节数，如果没有数据被读入 `InputStream` ，则返回 `-1` 。如果内部指针到达文件末尾，则返回 `-1` .

```
int i = inputStream.read();
while (i != -1) {
    byte b = (byte)i; // 使用 b 完成操作
}
```

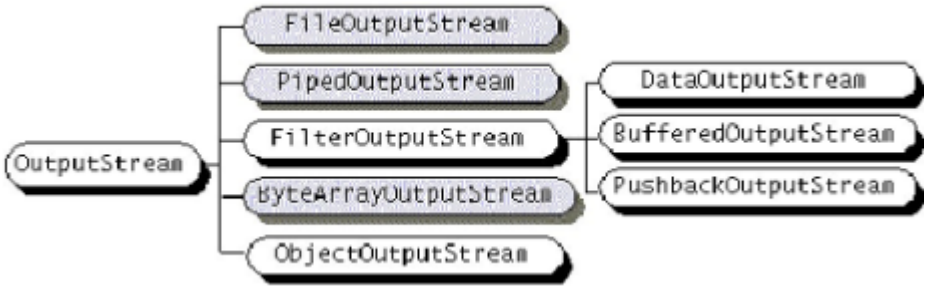
除了 `read` 方法， `InputStream` 还定义了以下方法：

- `public int available() throws IOException` 返回可无阻塞地读取（或跳过）的字节数。
- `public long skip(long n) throws IOException` 从这个 `InputStream` 中跳过指定的字节数，返回实际跳过的字节数，返回值可能小于指定的值。
- `public void mark(int readLimit)` 标记这个 `InputStream` 中内部指针的当前位置。之后调用 `reset` 将指针返回到标记的位置。 `readLimit` 参数指定标记位置失效之前要读取的字节数。
- `public void reset()` 将 `InputStream` 中的内部指针重新定位到标记的位置。
- `public void close()` 关闭这个 `InputStream` 。
- `boolean markSupported()` ： 当前的流是否支持读指针的记录功能。

```
byte[] buffer = new byte[256];
int bytes = System.in.read(buffer, 0, 256); // System.in 就是一个inputStream
String str = new String(buffer, offset:0, bytes, charsetName:""); // 字符编码参数可选
```

- **OutputStream**

`OutputStream` 类也是抽象类，作为字节输出流的直接或间接的父类，当程序需要向外部设备输出数据时，需要创建 `OutputStream` 的某一个子类的对象来完成。该类的常用方法也可能抛出 `IOException` 异常。



- `void write(int b)` ： 往流中写一个字节 b。
- `void write(byte b[])` ： 往流中写一个字节数组 b。
- `void write(byte b[], int off, int len)` ： 把字节数组 b 中从下标 off 开始，长度为 len 的字节写入流中。
- `flush()` ： 把缓存中所有内容强制输出到流中。
- `close()` ： 流操作完毕后必须关闭。

子类：字节数组输入输出流

- **`ByteArrayInputStream` 类**
    - 从内存的字节数组中读取数据，该类有如下两种构造方法重载形式。
    - `ByteArrayInputStream(byte[] buf)` ： 创建一个字节数组输入流，字节数组类型的数据源由参数 buf 指定。
    - `ByteArrayInputStream(byte[] buf,int offse,int length)` ： 创建一个字节数组输入流，其中，参数 buf 指定字节数组类型的数据源，offset 指定在数组中开始读取数据的起始下标位置，length 指定读取的元素个数。

```
byte[] b = new byte[] { 1, –1, 25, –22, –5, 23 }; // 创建数组
ByteArrayInputStream bais = new ByteArrayInputStream(b, 0, 6); // 创建字节数组输入流
int i = bais.read(); // 从输入流中读取下一个字节，并转换成int型数据
```

- **`ByteArrayOutputStream`类**
    - 向内存的字节数组中写入数据

```
public class Test09 {
    public static void main(String[] args) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] b = new byte[] { 1, –1, 25, –22, –5, 23 }; // 创建数组
        baos.write(b, 0, 6); // 将字节数组b中的前4个字节元素写到输出流中
        System.out.println(”数组中一共包含：” + baos.size() + ”字节”); // 输出缓冲区中的字节数
        byte[] newByteArray = baos.toByteArray(); // 将输出流中的当前内容转换成字节数组
        System.out.println(Arrays.toString(newByteArray)); // 输出数组中的内容
    }
}

数组中一共包含：6字节
[1, –1, 25, –22, –5, 23]
```

子类：文件输入输出流

- **`FileInputStream`**

`FileInputStream` 是 Java 流中比较常用的一种，它表示从文件系统的某个文件中获取输入字节。通过使用 `FileInputStream` 可以访问文件中的一个字节、一批字节或整个文件。
在创建 `FileInputStream` 类的对象时，如果找不到指定的文件将抛出 `FileNotFoundException` 异常，该异常必须捕获或声明抛出。
`FileInputStream` 常用的构造方法主要有如下两种重载形式。

- `FileInputStream(File file)` ： 通过打开一个到实际文件的连接来创建一个 `FileInputStream` ，该文件通过文件系统中的 `File` 对象 `file` 指定。
- `FileInputStream(String name)` ： 通过打开一个到实际文件的链接来创建一个 `FileInputStream` ，该文件通过文件系统中的路径名 `name` 指定。

```
public static void main(String[] args) {
        File f = new File(”D:/myJava/HelloJava.java”);
        FileInputStream fis = null;
        try {
            // 因为File没有读写的能力,所以需要有个InputStream
            fis = new FileInputStream(f);
            // 定义一个字节数组
            byte[] bytes = new byte[1024];
            int n = 0; // 得到实际读取到的字节数
            while ((n = fis.read(bytes)) != –1) {
                String s = new String(bytes, 0, n); // 将数组中从下标0到n的内容给s
```

```
                System.out.println(s);
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                fis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- **`FileOutputStream`**

继承自 OutputStream 类，重写和实现了父类中的所有方法。 `FileOutputStream` 类的对象表示一个文件字节输出流，可以向流中写入一个字节或一批字节。在创建 FileOutputStream 类的对象时，如果指定的文件不存在，则创建一个新文件；如果文件已存在，则清除原文件的内容重新写入。

`FileOutputStream` 类的构造方法主要有如下 4 种重载形式。

- `FileOutputStream(File file)` ： 创建一个文件输出流，参数 `file` 指定目标文件。
- `FileOutputStream(File file,boolean append)` ： 创建一个文件输出流，参数 `file` 指定目标文件， `append` 指定是否将数据添加到目标文件的内容末尾，如果为 `true` ，则在末尾添加；如果为 `false` ，则覆盖原有内容；其默认值为 `false` 。
- `FileOutputStream(String name)` ： 创建一个文件输出流，参数 `name` 指定目标文件的文件路径信息。
- `FileOutputStream(String name,boolean append)` ： 创建一个文件输出流，参数 `name` 和 `append` 的含义同上。

```
public static void main(String[] args) {
        FileInputStream fis = null; // 声明FileInputStream对象fis
        FileOutputStream fos = null; // 声明FileOutputStream对象fos
        try {
            File srcFile = new File("D:/myJava/HelloJava.java");
            fis = new FileInputStream(srcFile); // 实例化FileInputStream对象
            File targetFile = new File("D:/myJava/HelloJava.txt"); // 创建目标文件对象，该文件不存在
            fos = new FileOutputStream(targetFile); // 实例化FileOutputStream对象
            byte[] bytes = new byte[1024]; // 每次读取1024字节
            int i = fis.read(bytes);
            while (i != -1) {
                fos.write(bytes, 0, i); // 向D:\HelloJava.txt文件中写入内容
                i = fis.read(bytes);
            }
            System.out.println("写入结束！");
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                fis.close(); // 关闭FileInputStream对象
                fos.close(); // 关闭FileOutputStream对象
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

字符流

Reader类是所有字符流输入类的父类，该类定义了许多方法，这些方法对所有子类都是有效的。

- `int read()` 从输入流中读取一个字符，并把它转换为 0~65535 的整数。如果返回 -1， 则表示已经到了输入流的末尾。
- `int read(char[] cbuf)` 从输入流中读取若干个字符，并把它们保存到参数 cbuf 指定的字符数组中，返回读取的字符数，如果返回 -1，则表示已经到了输入流的末尾。
- `int read(char[] cbuf,int off,int len)` 从输入流中读取若干个字符，并把它们保存到参数 cbuf 指定的字符数组中。其中，off 指定在字符数组中开始保存数据的起始下标，len 指定读取的字符数。返回实际读取的字符数，如果返回 -1，则表示已经到了输入流的末尾。

- `InputStreamReader` 类是 `Reader` 类的子类，可以将字节输入流转换为字符输入流，可以指定字符编码。

对应的，**Writer类是所有的字符输出类的父类。**

- `void write(int c)` 向输出流中写入一个字符
- `void write(char[] cbuf)` 把参数 cbuf 指定的字符数组中的所有字符写到输出流中
- `void write(char[] cbuf,int off,int len)` 把参数 cbuf 指定的字符数组中的若干字符写到输出流中。其中，off 指定字符数组中的起始下标，len 表示元素个数
- `void write(String str)` 向输出流中写入一个字符串
- `void write(String str, int off,int len)` 向输出流中写入一个字符串中的部分字符。其中，off 指定字符串中的起始偏移量，len 表示字符个数
- `append(char c)` 将参数 c 指定的字符添加到输出流中
- `append(charSequence esq)` 将参数 esq 指定的字符序列添加到输出流中
- `append(charSequence esq,int start,int end)` 将参数 esq 指定的字符序列的子序列添加到输出流中。其中，start 指定子序列的第一个字符的索引，end 指定子序列中最后一个字符后面的字符的索引，也就是说子序列的内容包含 start 索引处的字符，但不包括 end索引处的字符。

子类：字符文件输入输出流

读取和输出字符文件的类是 `FileReader` 和 `FileWriter` 。

- 在用该类的构造方法创建 `FileReader` 读取对象时，默认的字符编码及字节缓冲区大小都是由系统设定的。要自己指定这些值，可以在 `FileInputStream` 上构造一个 `InputStreamReader` 。

```
public static void main(String[] args) {
    FileReader fr = null;
    try {
        fr = new FileReader("D:/myJava/HelloJava.java"); // 创建FileReader对象
        int i = 0;
        while ((i = fr.read()) != -1) { // 循环读取
            System.out.print((char) i); // 将读取的内容强制转换为char类型
        }
    } catch (Exception e) {
        System.out.print(e);
    } finally {
```

```
            try {
                fr.close(); // 关闭对象
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

子类：字符缓冲区输入输出流

`BufferedReader` 类主要用于辅助其他字符输入流，它带有缓冲区，可以先将一批数据读到内存缓冲区。接下来的读操作就可以直接从缓冲区中获取数据，而不需要每次都从数据源读取数据并进行字符编码转换，这样就可以提高数据的读取效率。
`BufferedReader`类的构造方法需要使用已有的`Reader`对象，有如下两种重载形式。

- `BufferedReader(Reader in)` ： 创建一个 `BufferedReader` 来修饰参数 `in` 指定的字符输入流。
- `BufferedReader(Reader in,int size)` ： 创建一个 `BufferedReader` 来修饰参数 `in` 指定的字符输入流，参数 `size` 则用于指定缓冲区的大小，单位为字符。

此外， `BufferedReader` 还提供了 `readLine()` 方法，该方法返回包含该行内容的字符串，但该字符串中不包含任何终止符，如果已到达流末尾，则返回 `null` 。

`BufferedWriter`类主要用于辅助其他字符输出流，它同样带有缓冲区，可以先将一批数据写入缓冲区，当缓冲区满了以后，再将缓冲区的数据一次性写到字符输出流，其目的是为了提高数据的写效率。

`BufferedWriter` 类的构造方法有如下两种重载形式。

- `BufferedWriter(Writer out)` ： 创建一个 `BufferedWriter` 来修饰参数 `out` 指定的字符输出流。
- `BufferedWriter(Writer out,int size)` ： 创建一个 `BufferedWriter` 来修饰参数 `out` 指定的字符输出流，参数 size 则用于指定缓冲区的大小，单位为字符。

转换流

正常情况下，字节流可以对所有的数据进行操作，但是有些时候在处理一些文本时我们要用到字符流，比如，查看文本的中文时就是需要采用字符流更为方便。所以 Java IO 流中提供了两种用于将字节流转换为字符流的转换流。
`InputStreamReader` 用于将字节输入流转换为字符输入流，其中 `OutputStreamWriter` 用于将字节输出流转换为字符输出流。使用转换流可以在一定程度上避免乱码，还可以指定输入输出所使用的字符集。

6.3 对象序列化

对象序列化(Serialize) 指将一个Java对象写入IO流中，与此对应的是，对象的反序列化(Deserialize) 则指从IO流中恢复该Java对象。如果想让某个Java对象能够序列化，则必须让它的类实现 `java.io.Serializable` 接口：

```
public interface Serializable {
}
```

`Serializable` 接口是一个空接口，实现该接口无须实现任何方法，它只是告诉JVM该类可以被序列化机制处理。

`ObjectOutputStream` 类继承了 `OutputStream` 类，同时实现了 `ObjectOutput` 接口，提供将对象序列化并写入流中的功能，该类的构造方法 `public ObjectOutputStream(OutputStream out)` 需要传入一个 `OutputStream` 对象，用来表示将对象二进制流写入到指定的 `OutputStream` 中。

```
public static void main(String[] args) throws FileNotFoundException, IOException {
    ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(”object.txt”));
```

```
    //创建一个ObjectOutputStream 输出流
    Person per = new Person(”JAVA_SE”, 7);
    //将Per对象写入输出流
    oos.writeObject(per);
}
```

`ObjectInputStream` 类继承了 `InputStream` 类，同时实现了 `ObjectInput` 接口，提供了将对象序列化并从流中读取出来的功能。该类方法的构造方法 `public ObjectInputStream(InputStream in)` 需要传入一个 `InputStream` 对象，用来创建从指定 `InputStream` 读取的 `ObjectInputStream` 。

- `transient` 关键字修饰的成员变量，在类的实例对象的序列化处理过程中会被忽略。 因此，transient变量不会贯穿对象的序列化和反序列化，生命周期仅存于调用者的内存中而不会写到磁盘里进行持久化。
- 如果序列化的对象包含其他对象，则所包含对象的类也必须实现 `Serializable` 接口，以便所包含的对象也可以序列化。
- 此外，类的静态成员不能被序列化。序列化只是序列化了对象而已。

7 线程

Java 有两种方法可以用来创建线程：扩展 `java.lang.Thread` 类；创建一个实现 `java.lang.Runnable` 接口的类，并将它的一个实例传递给 `Thread` 。 若选择第一种方法，需要覆盖它的 `run` 方法，并在其中编写希望由线程执行的代码。一旦有一个Thread对象，就可调用它的 `start` 方法来启动线程。线程启动时，将执行它的 `run` 方法。一旦 `run` 方法返回或抛出异常，线程就会死亡并被作为垃圾回收。

每个线程都有一个状态，可以是以下 6 种状态之一。

（1）new，线程还没有启动的状态。

（2）runnable，线程正在执行的状态。

（3）blocked，线程正在等待一个锁（lock），以便访问某个对象的状态。

（4）waiting，线程无限期等待另一个线程执行某个动作的状态。

（5）timed_waiting，线程在指定时间内等待另一个线程执行某个动作的状态。

（6）terminated，线程退出的状态。

表示线程的这些状态的值封装在 `java.lang.Thread.State` 枚举中。这个枚举的成员是 `NEW` 、 `RUNNABLE` 、 `BLOCKED` 、 `WAITING` 、 `TIMED_WAITING` 和 `TERMINATED` 。

`Thread` 类的公共构造方法：

```
public Thread()
public Thread(String name)
public Thread(Runnable target)
public Thread(Runnable target, String name)
```

`Thread` 类的其他一些有用的方法：

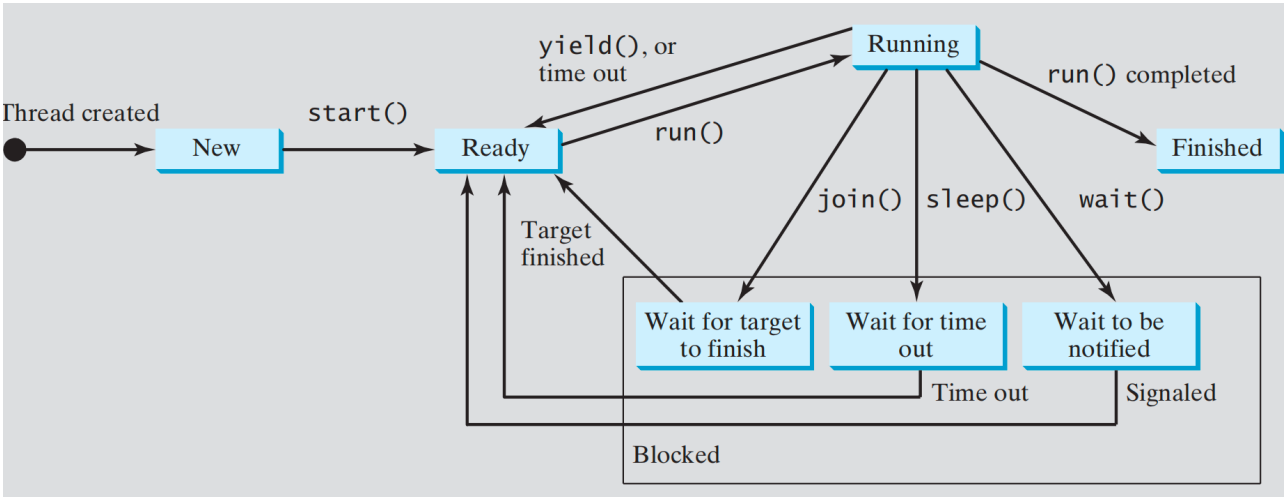
```
public String getName() // 返回这个线程的名称。
public Thread.State getState() //返回这个线程当前的状态。
public void interrupt()/// 中断这个线程。
public void start()//启动这个线程。
// 设置进程优先级，1~10，10 优先级最高，main 优先级为 5
public final void setPriority(int priority)
```

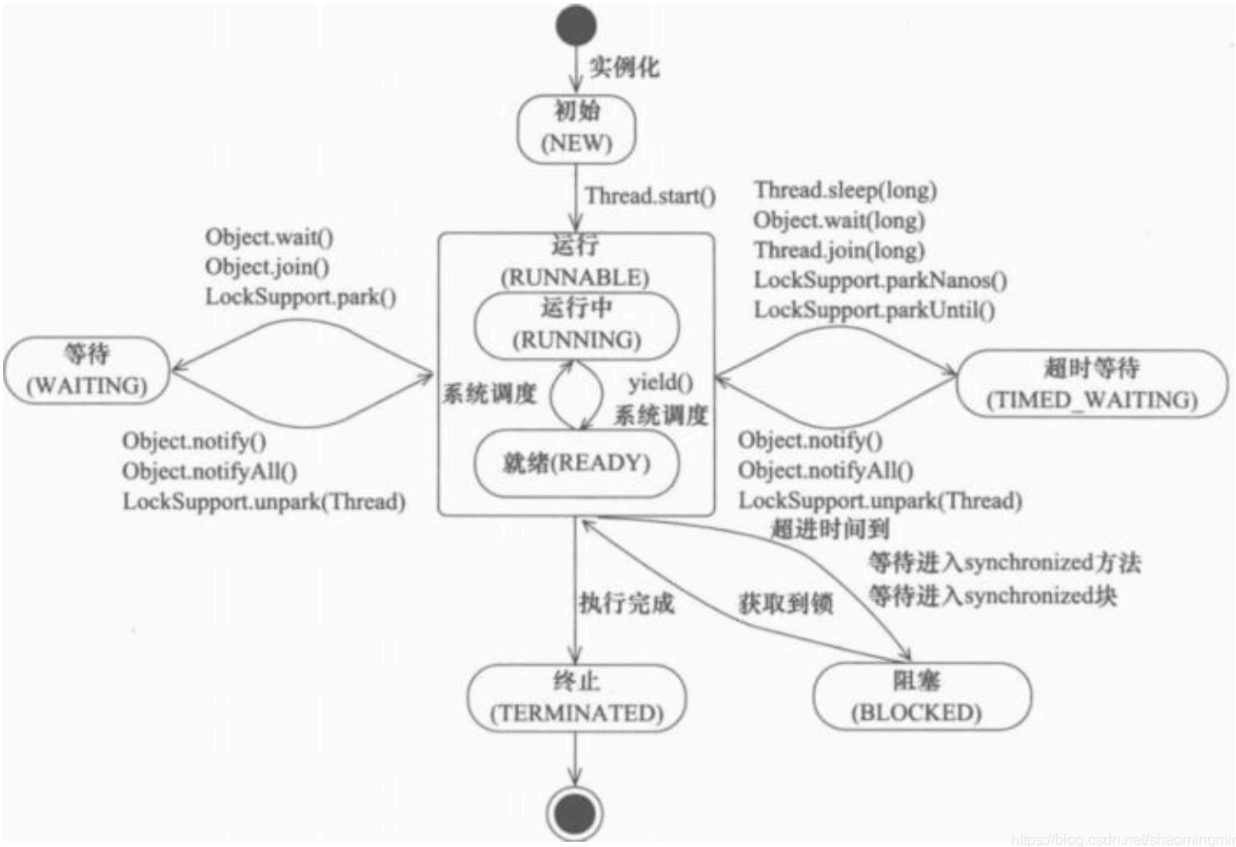
// 这两个函数是静态的。这意味着不管是哪个对象执行的，实际上都是正在 CPU 执行的那个 CPU 去 sleep。
public static void sleep(long millis) //在指定的毫秒时间内停止当前线程。
public static Thread currentThread() // 静态方法，返回当前的工作线程。

7.1 进程生命周期

进程生命周期：值得注意的是

- start() 会使线程进入就绪态，等待调度机调度它开始运行，此时会自动调用它的 run() 来进入运行态。
- wait() 函数内对信号量机制做了包装，会阻塞当前进程。
- 进程进入 finish() 后实际上已经死了，不能起死回生。
- sleep() 时间到后并不会立即运行，而是进入就绪队列等待下一次调度。





- `join()` 方法

如果希望在执行其他操作之前等待另一个线程完成它的任务，可以使用 `join()` 方法。当前线程转为阻塞状态。等到另一个线程结束，当前线程再由阻塞状态变为就绪状态，等待 cpu 资源。

如下的代码启动线程并调用它的join方法，导致当前线程被阻塞，直到调用join的线程完成它的工作。该程序运行时将打印「After 3 seconds」后面跟着「Exit」。

```
public static void main(String[] args) {
        Thread thread = new Thread(() -> {
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }
        System.out.println(”After 3 seconds”);
    });
    thread.start();

    try {
        thread.join();
    } catch (InterruptedException e) {
    }
    System.out.println(”Exit”);
}
```

- `sleep(long)` 方法

`sleep` 导致当前线程休眠，与 `wait` 方法不同的是 `sleep` 不会释放当前占有的锁，`sleep(long)` 会导致线程进入 `TIMED-WATING` 状态，而 `wait()` 方法会导致当前线程进入 `WATING` 状态。

- `yield()` 方法–线程让步

`yield` 会使当前线程让出CPU 执行时间片，与其他线程一起重新竞争 CPU 时间片。

7.2 创建线程

- 扩展 `Thread` 类

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
            try {
                sleep(1000);
            } catch (InterruptedException e) {

            }
            }
        }
}
public class ThreadDemo2 {
        public static void main(String[] args) {
        MyThread thread = new MyThread();
```

```
        thread.start();
    }
}
```

- 实现 `java.lang.Runnable` 接口。这个接口有个需要实现的 `run` 方法。之后需要实例化 `Thread` 类并将 `Runnable` 实例传递给它的构造方法。

```
public class RunnableDemo1 implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
        }
    }

    public static void main(String[] args) {
        RunnableDemo1 demo = new RunnableDemo1();
        Thread thread = new Thread(demo);
        thread.start();
    }
}
```

7.3 线程停止

`Thread` 类有一个 `stop` 方法可用于停止线程，由于这个方法不安全，因此不推荐使用它。相反，应该使 `run` 方法在需要停止某个线程时能够自然退出。一种常用的技术是使用带条件的 `while` 循环。**当想要停止线程时，只需将条件值设为`false`即可。一般由其他的函数来调用修改`condition`的值的方法。**如果代码是长时间运行的，那么在每次迭代中还需要检查线程是否已经被中断。如果线程被中断，则需要退出 `while` 循环。下面是这个策略的语法：

```
boolean condition = true;
public void run {
    while （condition） {
        if （Thread.interrupted()） { // 当前线程被中断 break;
        } // 执行长时间运行任务
    }
}
```

在类中，还需要提供一个方法来修改 `condition` 的值：

```
public synchronized void stopThread() {
    condition = false;
}
```

7.4 线程池

使用线程池的原因：

- 1. 线程的创建和销毁的开销都比较大，降低资源消耗
- 2. 线程是可复用的，提高响应速度
- 3. 对多任务多线程进行管理，提高线程的可管理性

7.5 同步

方法同步

每个 Java 对象都有一个内在锁（intrinsic lock），有时称为监视器锁（monitor lock）。获得对象的内在锁是能够独占该对象访问权的一种方式。

`synchronized` 修饰符可用于锁定对象。当一个线程调用一个非静态 `synchronized` 方法时，它会在方法执行之前，自动尝试获得该方法对象的内在锁。在该方法返回前，线程一直持有锁。**一旦某个线程锁定了某个对象，其他线程就不能调用同一个对象的同一方法或其他同步方法。**其他线程只有等待，直到这个锁再次变成可用的为止。锁还可以重入（reentrant），这意味着持有锁的线程可以调用同一对象上的其他同步方法。当这个方法返回时，释放内在锁。

> 静态方法也可以同步，在这种情况下，会使用与该方法的类关联的Class对象的锁。

SafeUserStat是线程安全的：

```
public class SafeUserStat {
    int userCount;

    public synchronized int getUserCount() {
        return userCount;
    }

    public synchronized void increment() {
        userCount++;
    }

    public synchronized void decrement() {
        userCount--;
    }
}
```

块同步

当无法同步方法时（如无法修改共享对象的源代码），可以通过块同步（block synchronization）锁定**任何**对象。一个同步块可提供对象的内部锁。锁在块中的代码执行之后被释放。对于一个不安全的 `UserStat`，可以如下来进行同步调用：

```
UserStat userStat = new UserStat();
...
public void incrementCounter() {
    synchronized（userStat） {
        // 在 userStat 上调用
        // increment、decrement 和 getUserCount 方法
        // 都将被同步
        userStat.increment();
```

```
    }
}
```

7.6 线程中断

Java的中断是一种协作机制，也就是说**通过中断并不能直接中断另外一个线程，而需要被中断的线程自己处理中断。**

在Java的中断模型中，每个线程都有一个 `boolean` 标识，代表着是否有中断请求（该请求可以来自所有线程，包括被中断的线程本身）。例如，当线程t1想中断线程t2，只需要在线程t1中将线程t2对象的中断标识置为 `true` ，然后线程2可以选择在合适的时候处理该中断请求，甚至可以不理会该请求，就像这个线程没有被中断一样。

中断有关的方法有以下 3 个：

- `void interrupt()` 中断线程，设置线程的中断位true
- `boolean isInterrupted()` 检查线程的中断标记位，true-中断状态， false-非中断状态
- `static boolean interrupted()` 静态方法，返回当前线程的中断标记位，同时清除中断标记，改为false。比如当前线程已中断，调用 `interrupted()` ，返回 `true` , 同时将当前线程的中断标记位改为 `false` , 再次调用 `interrupted()` ，会发现返回 `false`

> 具体来说，当对一个线程，调用 `interrupt()` 时，
>
> - ① 如果线程处于被阻塞状态（例如处于 `sleep` ， `wait` ， `join` 等状态），那么线程将立即退出被阻塞状态，并抛出一个 `InterruptedException` 异常。仅此而已。
> - ② 如果线程处于正常活动状态，那么会将该线程的中断标志设置为 `true` ，仅此而已。被设置中断标志的线程将继续正常运行，不受影响。
>
> `interrupt()` 并不能真正的中断线程，需要被调用的线程自己进行配合才行。也就是说，一个线程如果有被中断的需求，那么就可以这样做。
>
> - ① 在正常运行任务时，经常检查本线程的中断标志位，如果被设置了中断标志就自行停止线程。
> - ② 在调用阻塞方法时正确处理 `InterruptedException` 异常。（例如， `catch` 异常后就结束线程。）

7.7 线程协调

`java.lang.Object`类提供了几个在线程协调中有用的方法。

- `public final void wait() throws InterruptedException`
  - 导致当前线程等待，直到另一个线程调用该对象的 `notify` 或 `notifyAll` 方法。 `wait` **方法通常在同步方法中使用**，其会导致正在访问同步方法的调用线程将自己置于等待状态并放弃对象锁。
- `public final void wait(long timeout) throws InterruptedException`
  - 导致当前线程等待，直到另一个线程调用该对象的 `notify` 或 `notifyAll` 方法，或者指定的时间已经过去。
- `public final void notify()`
  - 将通知正在等待该对象锁的单个线程。如果有多个线程在等待，则选择通知其中一个线程，这种选择是任意的。
- `pubic final void notifyAll()`
  - 将通知所有等待该对象锁的线程。

> - 为什么wait(), notify(), notifyAll() 必须在同步方法/代码块中调用?
>
> **只有在调用线程拥有某个对象的独占锁时，才能够调用该对象的`wait()`,`notify()`和`notifyAll()`方法。这一点通常不会被程序员注意，因为程序验证通常是在对象的同步方法或同步代码块中调用它们的。如果尝试在未获取对象锁时调用这三个方法，那么你将得到"java.lang.IllegalMonitorStateException:current thread not owner"。**
>
> 当一个线程正在某一个对象的同步方法中运行时调用了这个对象的 `wait()` 方法，那么这个线程将释放该对象的独占锁并被放入这个对象的等待队列。注意， `wait()` 方法强制当前线程释放对象锁。这意味着在调用某对象的 `wait()` 方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的 `wait()` 方法。
>
> 当某线程调用某对象的 `notify()` 或 `notifyAll()` 方法时，**任意一个(对于`notify()`)或者所有(对于`notifyAll()`)**在该对象的等待队列中的线程，将被转移到该对象的入口队列。接着这些队列(译者注:可能只有一个)将竞争该对象的锁，最终获得锁的线程继续执行。如果没有线程在该对象的等待队列中等待获得锁，那么 `notify()` 和 `notifyAll()` 将不起任何作用。在调用对象的 `notify()` 和 `notifyAll()` 方法之前，调用线程必须已经得到该对象的锁。因此，必须在某个对象的同步方法或同步代码块中才能调用该对象的 `notify( )`或 `notifyAll()` 方法。
>
> 对于处于某对象的等待队列中的线程，只有当其他线程调用此对象的 `notify()` 或 `notifyAll()` 方法时才有机会继续执行。

7.8 锁

> 我们已经知道可以使用 `synchronized` 修饰符来锁定共享资源。虽然 `synchronized` 很容易使用，但是这种锁定机制也有局限性，例如，试图获取这种内在锁的线程不能后退，而且如果无法获得锁，线程就会**无限期地阻塞。此外，锁定和解锁仅限于方法和代码块：无法在一个方法中锁定一个资源，在另一个方法中释放它。**

`Lock` 接口提供了一些方法可以克服 Java 内在锁的局限性。 `Lock` 接口定义了 `lock` 方法和 `unlock` 方法。这意味着，只要持有对锁的引用，就可以在程序的任何地方释放锁。在大多数情况下，最好是在调用 `lock` 之后，在 `finally` 子句中调用 `unlock` ，以确保锁总能被释放：

```
aLock.lock();
try {
    // 利用锁定的资源完成一些操作
} finally {
    aLock.unlock();
}
```

如果某个锁不可用， `lock` 方法将一直阻塞，直到锁可用为止。这种行为类似于使用 `synchronized` 产生的隐式锁的效果。除了锁定和解锁， `Lock` 接口还提供了 `tryLock` 方法：

```
boolean tryLock()
boolean tryLock(long time, TimeUnit timeUnit)
```

对第一个重载，当锁可用时，返回 `true` ；否则返回 `false` 。在后一种情况下，它不会发生阻塞。

对第二个阻塞，如果锁可用，返回 `true` ；否则等到指定的时间失效，而且如果没有获得锁，将返回 `false` 。 `time` 参数指定它将等待的最长时间， `timeUnit` 参数指定第一个参数的时间单位。