

JAVA 应用与开发 控制台应用程序设计

让我们愉快的 *Coding* 起来吧...

王晓东

中国海洋大学信息学院计算机系

OCTOBER 12, 2018



■ 88

大纲

- 1 命令行参数
- 2 系统属性
- 3 标准输入/输出
- 4 文件操作
- 5 过时 API
- 6 注解（Annotation）
- 7 归档工具

命令行参数

命令行参数

命令行参数

在启动时 Java 控制台应用程序，可以一次性地向程序中传递（零至多个）字符串参数，这些参数被称为命令行参数。语法格式如下：

```
1 java <应用程序类名> [<命令行参数>]*
```

命令行参数

在启动时 Java 控制台应用程序，可以一次性地向程序中传递（零至多个）字符串参数，这些参数被称为命令行参数。语法格式如下：

```
1 java <应用程序类名> [<命令行参数>]*
```

❖ 说明

- 命令行参数将被系统接收并静态初始化为一个一维的 String 数组对象，然后将之作为实参传给应用程序入口方法 main()。
- 命令行参数须使用空格符分隔，如果参数中包含空格符则必须使用双引号括起来。

课程配套代码 ▶ `sample.commandline.CommandLineArgsSample.java`

命令行参数

课程配套代码 ▶ `sample.commandline.CommandLineArgsSample.java`

Linux 下运行程序

```
1 > java CommandLineArgsSample Lisa "Billy" "Mr_Brown"
```


命令行参数

课程配套代码 ▶ `sample.commandline.CommandLineArgsSample.java`

Linux 下运行程序

```
1 > java CommandLineArgsSample Lisa "Billy" "Mr_Brown"
```

Windows 下运行程序

```
1 C:\> java.exe CommandLineArgsSample Lisa "Billy" "Mr_Brown" "a""b"
```

命令行参数

课程配套代码 ▶ `sample.commandline.CommandLineArgsSample.java`

Linux 下运行程序

```
1 > java CommandLineArgsSample Lisa "Billy" "Mr_Brown"
```

Windows 下运行程序

```
1 C:\> java.exe CommandLineArgsSample Lisa "Billy" "Mr_Brown" "a""b"
```

输出结果:

output

```
Lisa  
Billy  
Mr Brown
```

可变参数方法

- Java 语言允许在定义方法时指定使用任意数量的参数，其格式是在参数类型后加 “...”。
- 可变长度参数必须放在参数列表的最后，一个方法最多只能包含一个可变长度参数。
- 编译时，可变参数被当作**一维数组处理**。

```
1 public void myprint(String s, int i, Object... objs) { // 可变参数方法
2     System.out.println(s.toUpperCase());
3     System.out.println(100 * i);
4     for(Object o: objs) { // 作为一维数组处理
5         System.out.println(o);
6     }
7 }
```

系统属性

Java 系统属性

- 记录当前操作系统和 JVM 等相关的环境信息。
- 以**键值对**的形式存在，由**属性名称**、**属性值**两部分组成。
- 均为字符串形式。

系统属性

Java 系统属性

- 记录当前操作系统和 JVM 等相关的环境信息。
- 以**键值对**的形式存在，由**属性名称**、**属性值**两部分组成。
- 均为字符串形式。

❖ 系统属性的用途

系统属性在 URL 网络编程、数据库编程和 Java Mail 邮件收发等编程中经常使用，一般被用来设置代理服务器、指定数据库的驱动程序类等。

除了使用代码方法外，也可使用命令在运行程序时添加新的系统属性：

```
1 >java -Dmmmm=vvvv SystemPropertiesSample
```

遍历、操作系统属性

可以使用 `System.getProperties()` 获得一个封装了当前运行环境下所有系统属性信息的 `Properties` 类 (`java.util.Properties`) 的实例。

课程配套代码 ▶ `sample.commandline.SystemPropertiesSample.java`

❖ 可用方法

Enumeration propertyNames()

String getProperty(String key)

Object setProperty(string key, String value)

void load(InputStream inStream)

void store(OutputStream out, String header)

标准输入/输出

标准输入/输出

❖ 控制台程序的交互方式

- 用户使用键盘作为**标准输入设备**向程序输入数据
- 程序利用计算机终端窗口作为**程序标准输出设备**显示输出数据

这种操作被称为**标准输入/输出**（Standard Input/Output）。

标准输入/输出的分类

java.lang.System 类的三个静态类成员提供了有关标准输入/输出的 IO 操作功能。

System.in 从“标准输入”读入数据 (java.io.InputStream 类型)

System.out 向“标准输出”写出数据 (java.io.PrintStream 类型)

System.err 向“标准错误”写出数据 (java.io.PrintStream 类型)

标准输入/输出的分类

java.lang.System 类的三个静态类成员提供了有关标准输入/输出的 IO 操作功能。

System.in 从“标准输入”读入数据 (java.io.InputStream 类型)

System.out 向“标准输出”写出数据 (java.io.PrintStream 类型)

System.err 向“标准错误”写出数据 (java.io.PrintStream 类型)

❖ PrintStream 类的主要方法

- print()/println() 方法被进行了多次重载 (boolean、char、int、long、float、double 以及 char[], Object 和 String)。

读取控制台输入的传统方法

```
1  import java.io.InputStreamReader;
2  import java.io.BufferedReader;
3  import java.io.IOException;

5  public class TestStandardInput {
6      public static void main (String args[]) {
7          String s;
8          InputStreamReader isr = new InputStreamReader(System.in);
9          BufferedReader br = new BufferedReader(isr);
10         try {
11             s = br.readLine();
12             while (!s.equals("")) {
13                 System.out.println("Read:␣" + s);
14                 s = br.readLine();
15             }
16             br.close();
17         } catch (IOException e) {
18             e.printStackTrace();
19         }
20     }
21 }
```

对上述程序的几点解释

- `System.in` 为 `InputStream` 类型对象，功能较弱，只能以字节为单位从预定义的标准输入（键盘）读取信息。

对上述程序的几点解释

- `System.in` 为 `InputStream` 类型对象，功能较弱，只能以字节为单位从预定义的标准输入（键盘）读取信息。
- 程序并没有直接操作 `System.in` 对象进行读取操作，而是将其封装为一个功能稍强的 `InputStreamReader` 对象，以字符为单位读取信息。实际的过程为：`InputStreamReader` 对象并没有直接读取键盘输入，而是多次调用 `System.in` 对象的读字节功能，再将所得字节转换为字符。

对上述程序的几点解释

- `System.in` 为 `InputStream` 类型对象，功能较弱，只能以字节为单位从预定义的标准输入（键盘）读取信息。
- 程序并没有直接操作 `System.in` 对象进行读取操作，而是将其封装为一个功能稍强的 `InputStreamReader` 对象，以字符为单位读取信息。
- `InputStreamReader` 仍不能令人满意，再次封装，得到 `BufferedReader` 对象。后者提供了缓冲读取的功能，即多次调用 `InputStreamReader` 读字符操作，然后将所读取的多个字符积累起来组成字符串，其间以换行符为分隔，最终实现以行为单位读取字符串功能。

对上述程序的几点解释

- `System.in` 为 `InputStream` 类型对象，功能较弱，只能以字节为单位从预定义的标准输入（键盘）读取信息。
- 程序并没有直接操作 `System.in` 对象进行读取操作，而是将其封装为一个功能稍强的 `InputStreamReader` 对象，以字符为单位读取信息。
- `InputStreamReader` 仍不能令人满意，再次封装，得到 `BufferedReader` 对象。
- 当在键盘上空回车时，`BufferedReader` 的 `readLine()` 方法接收到的不是空值 `null`，而是一个长度为零的字符串`""`，其中包含 0 个字符但仍然是一个 Java 对象。

文件操作

❖ 创建 File 类对象

java.io 包中定义与数据输入、输出功能有关的类，包括提供文件操作功能的 File 类。

```
public File(String pathname)
```

通过给定的路径/文件名字符串创建一个新 File 实例。

```
public File(String parent, String child)
```

通过分别给定的 parent 路径名和 child 文件名（也可以是子路径名）或字符串来创建一个新 File 实例。

课程配套代码 ▶ `sample.commandline.FileOperationSample.java`

❖ 文件/目录名操作

- `String getName()`
- `String getPath()`
- `String getAbsolutePath()`
- `String getParent()`

❖ 设置和修改操作

- `boolean delete()`
- `void deleteOnExit()`
- `boolean createNewFile()`
- `setReadOnly()`
- `boolean renameTo(File dest)`

❖ 测试操作

- boolean exists()
- boolean canWrite()
- boolean canRead()
- boolean isFile()
- boolean isDirectory()
- boolean isAbsolute()

❖ 目录操作

- boolean mkdir()
- String[] list()
- File[] listFiles()

❖ 获取常规文件信息操作

- long lastModified()
- long length()

文件 I/O 有关读写类

❖ 常见的文本文件 I/O 操作的类

`java.io.FileReader` 类

提供 `read()` 方法以字符为单位从文件中读入数据。

`java.io.FileWrite` 类

提供 `write()` 方法以字符为单位向文件写出数据。

`java.io.BufferedReader` 类

提供 `readLine()` 方法以行为单位读入一行字符。

`java.io.PrintWriter` 类

提供 `print()` 和 `println()` 方法以行为单位写出数据。

读取文件内容

CODE ReadFile.java

```
1      import java.io.*;
2
3      public class ReadFileSample {
4          public static void main (String[] args) {
5              String fname = "test.txt";
6              File f = new File(fname);
7
8              try {
9                  FileReader fr = new FileReader(f); // 1
10                 BufferedReader br = new BufferedReader(fr);
11                 String s = br.readLine();
12                 while (s != null) { // 2
13                     System.out.println("读入: " + s);
14                     s = br.readLine(); }
15                 br.close();
16             } catch (FileNotFoundException e1) {
17                 System.err.println("File_not_found: " + fname);
18             } catch (IOException e2) {
19                 e2.printStackTrace();
20             }
21         }
22     }
```

读取文件内容

👉 上述代码几点说明

1. `FileReader` 的构造方法被重载过，接受以字符串形式给出的文件名，上述代码等价于：

```
1  FileReader fr = new FileReader("test.txt");
```

2. 使用 `BufferedReader` 的 `readLine()` 方法读文件，遇到文件结尾则返回 `null`，而不是 `""`，与读取键盘输入遇到空回车时返回空字符串的情况不同。

输出内容到文件

CODE WriteFileSample

```
1      import java.io.*;
3
3      public class TestWriteFile {
4          public static void main (String[] args) {
5              File file = new File("tt.txt");
6              try {
7                  InputStreamReader is = new InputStreamReader(System.in);
8                  BufferedReader in=new BufferedReader(is);
9                  FileWriter fw = new FileWriter(file);
10                 PrintWriter out = new PrintWriter(fw);
11                 String s = in.readLine();
12                 while(!s.equals("")) { // 从键盘逐行读入数据输出到文件
13                     out.println(s);
14                     s = in.readLine();
15                 }
16                 in.close(); // 关闭 BufferedReader 输入流
17                 out.close(); // 关闭连接文件的 PrintWriter 输出流
18             } catch (IOException e) {
19                 e.printStackTrace();
20             }
21         }
22     }
```

对上述代码的几点说明

1. 写文件时如果目标文件不存在，程序运行不会出错，而是自动创建该文件，但如果目标路径不存在，则会出错。
2. 写文件操作结束后一定要关闭输出流，即关闭文件，否则被操作文件仍处于打开状态，不安全。

文件过滤

文件过滤，即只检索和处理符合特定条件的文件。最常见的为按照文件类型（后缀）进行划分，如查找.class 或.xml 文件。文件过滤可以使用 `java.io.FileFilter` 接口，该接口只定义了一个抽象方法 `accept`。

```
boolean accept(File pathname)
```

测试参数指定的 `File` 对象对应的文件（目录）是否应该保留在文件列表中，即不被过滤。

在实际应用中，可以定义该接口的一个实现类，重写其中的 `accept()` 方法，在方法中添加文件过滤逻辑，然后创建一个该实现类的对象作为参数传递给 `File` 对象的文件列表方法 `list()`，在 `list()` 方法执行过程中会自动调用前者的 `accept()` 方法来过滤文件。

使用 FILEFILTER 实现文件过滤

课程配套代码 ▶ `sample.commandline.filefilter`

过时 API

过时 API

过时 API 是指那些过去定义的，现已不提倡使用的 API，包括类、属性和方法等。过时 API 均存在相应的替代物，这些替代者可能采用了更标准化的命名惯例，或者功能更适用。在将来的 JDK 版本中，过时 API 可能不再被支持，所以开发中应尽量避免使用。

```
1  import java.util.*;
2
3  public class TestDeprecation {
4      public static void main(String[] args) {
5          Date now = new Date();
6          int hour = now.getHours(); // 过时API
7          System.out.println(hour);
8      }
9  }
```

过时 API

编译程序时输出提示信息:

output

注意: `TestDeprecation.java` 使用或覆盖了已过时的 API。
注意: 要了解详细信息, 请使用 `-Xlint:deprecation` 重新编译。

使用下述命令重新编译程序:

```
1 >javac -Xlint:deprecation TestDeprecation.java
```

输出更详细说明信息:

output

```
TestDeprecation.java:5: 警告:[deprecation] java.util.Date 中的  
getHours() 已过时  
int hour = now.getHours(); ^  
1 警告
```

对上述代码的改造

在 Java API 文档中，java.util.Date 类的 getHour() 部分已作如下说明：

“从 JDK 1.1 开始，由 Calendar.get(Calendar.HOUR_OF_DAY) 取代”

CODE ♦ TestDeprecation.java

```
1  import java.util.*;
2
3  public class TestDeprecation {
4      public static void main(String[] args) {
5          Calendar c = Calendar.getInstance();
6          int hour = c.get(Calendar.HOUR_OF_DAY);
7          System.out.println(hour);
8      }
9  }
```


注解 (Annotation)

注解

是从 JDK5.0 开始新添加的一种语言特性，区别于代码注释（Comment）。

- 注解不直接影响程序的语义，开发和部署工具可以对其读取并以某种形式处理这些注解，可能生成其他 Java 源文件、XML 文档或要与包含注解的程序一起使用的其他构件。

注解

是从 JDK5.0 开始新添加的一种语言特性，区别于代码注释（Comment）。

- 注解不直接影响程序的语义，开发和部署工具可以对其读取并以某种形式处理这些注解，可能生成其他 Java 源文件、XML 文档或要与包含注解的程序一起使用的其他构件。
- 本质上，注解就是可以添加到代码中的一种类似于修饰符的成分，可以用于声明包、类、构造方法、方法、属性、参数和变量等场合。

Java 语言采用了一类新的数据类型来描述注解——注解类型相当于类或接口，每一条注解相当于该类的一个实例。

注解类型采用 `@interface` 标记来声明，例如，JDK5.0 及后续版本的 Java API (`java.lang`) 中定义了几种有用的注解类型：

- `public @interface Deprecated`
- `public @interface Override`
- `public @interface SuppressWarnings`

Override 注解

java.lang.Override 类型注解用于指明被注解的方法重写了父类中的方法，如果不是合法的方法重写，则编译报错。

```
1 public class Person {
2     private String name;
3     public Person(String name) {
4         this.name = name;
5     }
6     public void setName(String name) {
7         this.name = name;
8     }
9     public String getName() {
10        return name;
11    }
12    @Override
13    public String toString() { // 重写方法
14        return "Name:␣" + name;
15    }
16 }
```

toString 的原始定义

```
1 public String toString() {
2     return getClass().getName() + "@" + Integer.toHexString(hashCode());
3 }
```

DEPRECATED 注解

Deprecated 注解的作用是标记过时的 API。如果通过方法重写或调用的方式来使用已被注解为过时的方法时，编译器将会根据注解信息发现不应该使用此方法，并作提醒。

```
1 public class A {  
2     @deprecated  
3     public void ma() {  
4         System.out.println("In class A, just for test!");  
5     }  
6 }
```

```
1 public class TestMyDeprecation {  
2     public static void main(String[] args) {  
3         A a = new A();  
4         a.ma();  
5     }  
6 }
```

SUPPRESSWARNINGS 注解

使用 SuppressWarnings 注解可以关闭编译器对指定的一种或多种问题的提示/警告功能。

❖ 语法格式，比较自由，下述均可

```
@SuppressWarnings(value={"deprecation"})
```

```
@SuppressWarnings(value={"deprecation","unchecked"})
```

```
@SuppressWarnings("deprecation")
```

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

```
1  import java.util.*;
2  //import java.lang.SuppressWarnings;
3
4  @SuppressWarnings(value={"deprecation"})
5  public class TestSuppressWarnings {
6      public static void main(String[] args) {
7          Date now = new Date();
8          int hour = now.getHours();
9          System.out.println(hour);
10     }
11 }
```

编译时，则不会再输出先前的提示 API 过时信息。

归档工具

Java 归档工具是 JDK 中提供的一种多用途的存档及压缩工具，可以将多个文件或目录合并/ 压缩为单个的 Java 归档文件（jar, java archive）。

❖ jar 文件的主要作用

- 发布和使用类库；
- 作为程序组件或者插件程序的基本部署单位；
- 用于打包与组件相关联的资源文件。

❖ 使用 jar 工具基本语法格式

```
1 >jar {ctxui} [vfmOMe] [jar-file] [manifest-file] \  
2 [entry-point] [-C dir] files ...
```

参数说明

- c 创建新的归档文件。
- t 列出归档目录。
- x 解压缩已归档的指定（或者所有）文件。
- u 更新现有的归档文件。
- v 在标准输出中生成详细输出。
- f 指定归档文件名。
- m 包含指定清单文件中的清单信息。
- e 为捆绑到可执行 jar 文件的独立应用程序指定应用程序入口点。
- 0 仅存储，不使用任何 ZIP 压缩。
- M 不创建条目的清单文件。
 - i 为指定的 jar 文件生成索引信息。
- C 更改为指定的目录并包含其中的文件。

制作并使用自己的 JAR 文件

CODE ♦ A.java

```
1 public class A {  
2     public void ma() {  
3         System.out.println("In class A!");  
4     }  
5 }
```

CODE ♦ TestJar.java

```
1 public class TestJar {  
  
3     public static void main(String[] args) {  
4         A a = new A();  
5         a.ma();  
6     }  
7 }
```

制作并使用自己的 JAR 文件

❶ 编译源文件 A.java 得到字节码文件 A.class，在 A.class 所在路径下，运行如下命令进行归档处理：

```
1 >jar -cvf mylib.jar *.class
```

输出如下：

output

```
jar -cvf mylib.jar *.class
added manifest
adding: A.class(in = 380) (out= 275)(deflated 27%)
```

❷ 要使用 mylib.jar 文件中的字节码文件，必须先将其加入到编译和运行环境的 CLASSPATH 中（注意必须指定到.jar 文件的文件名）。

```
1 >export CLASSPATH="../Users/xiaodong/temp/mylib.jar"
```

❸ 编译 TestJar.java 源程序，并运行。

一直以来，我们总是使用 `java < 应用程序名字 >` 的方式运行 Java。
学习了归档工具后，有了一个新的选择：

以归档文件的形式发布 Java 程序并直接从归档文件中运行。

发布 JAVA 应用程序

CODE ♦ TestApp01.java

```
1 public class TestApp01 {  
2     public static void main(String[] args) {  
3         System.out.println("App01 is running...");  
4     }  
5 }
```

CODE ♦ TestApp02.java

```
1 import java.awt.*;  
2 import java.awt.event.*;  
  
4 public class TestApp02 {  
5     public static void main(String[] args) {  
6         Frame f = new Frame("TestApp02");  
7         f.setSize(200, 200);  
8         f.addWindowListener(new WindowAdapter() {  
9             public void windowClosing(WindowEvent e) {  
10                 System.exit(0);  
11             }  
12         });  
13         f.setVisible(true);  
14     }  
15 }
```

发布 JAVA 应用程序过程

❖ 编译程序

❖ 程序归档发布

```
1 >jar -cfe mylib01.jar TestApp01 *.class  
2 >jar -cfe mylib02.jar TestApp02 *.class
```

通过使用 `-e` 参数指定当前归档文件的应用程序入口点 (Entry-Point)。我们查看 `jar` 包中的清单文件可以发现多了一条 `Main-Class` 属性。

❖ 运行程序

```
1 >java -jar mylib01.jar  
2 >java -jar mylib02.jar
```

清单文件

清单文件提供了归档文件的有关说明信息。jar 包中使用一个特定的目录（META-INF）存放 MANIFEST.MF 清单文件。

❖ 清单文件格式

<属性名>:<属性值>

❖ MANIFEST.MF 示例

```
1 Manifest-Version: 1.0
2 Created-By: 1.6.0_33 (Apple Inc.)
3 Main-Class: TestApp01
```

每行最多 72 字符，写不下可以续行，续行必须以空格开头，且以空格开头的行都会被视为前一行的续行。可以自定义清单文件。

THE END

WANGXIAODONG@OUC.EDU.CN