

Java 应用与开发课程教学体系

很高兴同学们能够选修 Java 应用与开发课程。

希望我们一起通过这门课程的学习，建立 Java 语言编程的初步知识体系，掌握 Java 应用系统开发的方式、方法。更重要的，能够对编程这个事情、这项技能有更加深刻的认知，对未来的职业化发展有所促进。

Java 应用与开发课程的教学体系如图1所示，包括了 Java SE 和 Java EE 两个部分，每部分都涉及一些验证性实验，另外，会开展两次稍微大一点的集成开发项目。同时，在学习的过程中会穿插一些开发工具、设计模式、应用服务器和数据库的基本应用。

在课程学习的过程中，希望同学们要有足够的求知欲，养成良好的学习态度，具备不断探索的精神，多尝新、多实践、多总结。我想这是计算机专业人士应该具备的基本素养。

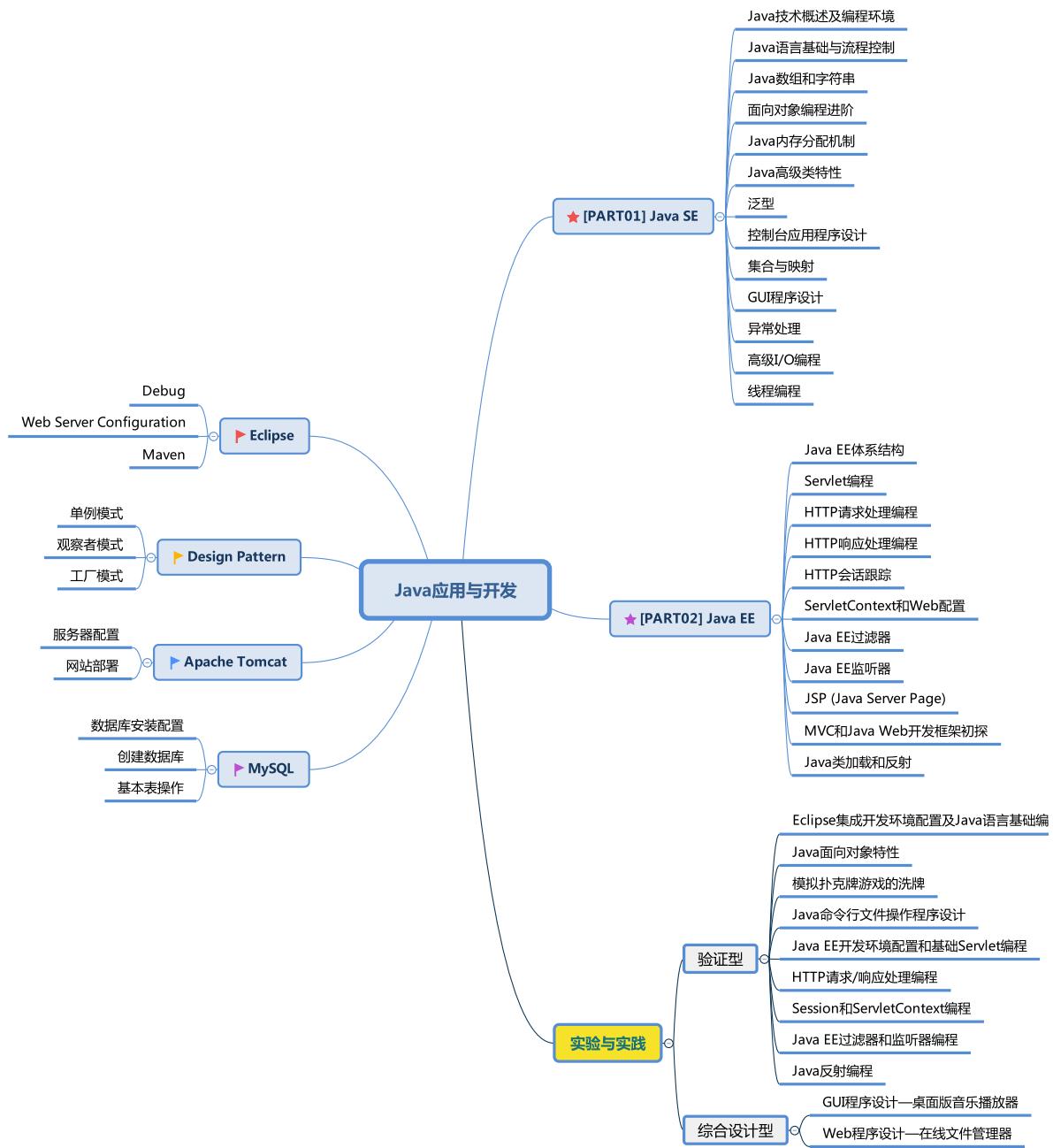


图 1 Java 应用与开发课程教学体系

☒ 1 ☒ Java 技术概述及开发环境

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第一周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 讲解 Java 的发展历程, 从 Java 的视角回顾 OOP;
2. 理解 Java 平台的相关概念和机制;
3. 掌握基本的 Java 开发环境配置方法。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

1.1 Java 技术概述

1.1.1 Java 发展简史

Java 的发展过程中伴随着多个伟大公司的起起落落。

- 1982** Sun 公司成立（安迪·贝托谢姆和麦克尼利）。
- 1986** Sun 公司上市。
- 1985** Sun 公司推出著名的 Java 语言。
- 2001** 9.11 事件前，Sun 市值超过 1000 亿美元；此后，由于互联网泡沫的破碎，其市值在一个月内跌幅超过 90%。
- 2004** Sun 公司和微软在旷日持久的 Java 官司中和解，后者支付前者高达 10 亿美元的赔偿费。
- 2006** 共同创始人麦克尼利辞去 CEO 一职，舒瓦茨担任 CEO 后尝试将 Sun 从设备公司向软件服务型公司转型，但不成功。
- 2010** Sun 公司被甲骨文公司收购。

Java 语言的版本迭代历程如图1.1所示。

1.1.2 Java 技术的特点

Java 具备以下技术特点：

面向对象 Java 是一种以对象为中心，以消息为驱动的面向对象的编程语言。

平台无关性 分为源代码级（需重新编译源代码，如 C/C++）和目标代码级（Java）平台无关。

分布式 可支持分布式技术及平台开发。

可靠性 不支持直接操作指针，避免了对内存的非法访问；自动单元回收功能防止内存丢失等动态内存分配导致的问题；解释器运行时实施检查，可发现数组和字符串访问的越界；提供了异常处理机制。

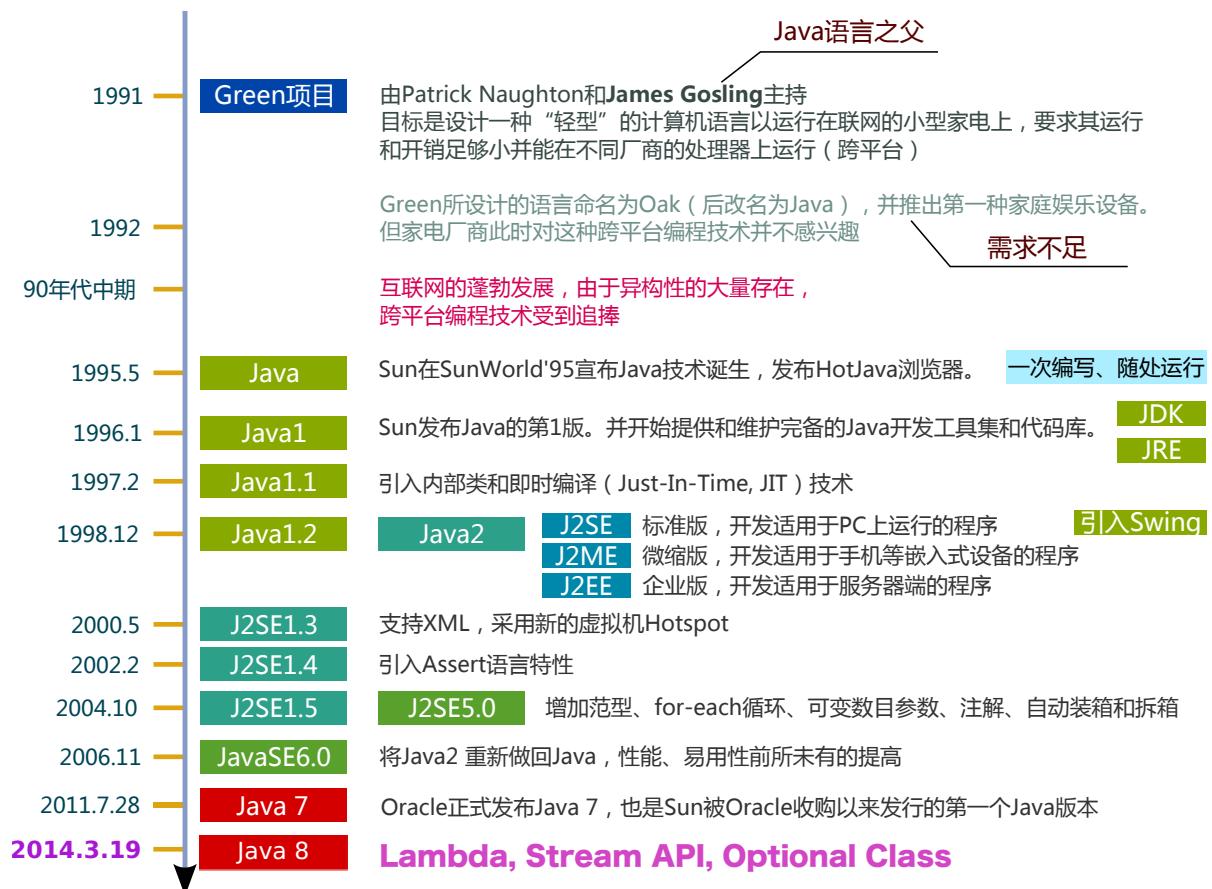


图 1.1 Java 版本迭代

多线程 C++ 没有内置的多线程机制，需调用操作系统的多线程功能来进行多线程程序设计；Java 提供了多线程支持。

网络编程 Java 具有丰富的网络编程库。

编译和解释并存 由编译器将 Java 源程序编译成字节码文件，再由运行系统解释执行字节码文件（解释器将字节码再翻译成二进制码运行）。

1.2 Java 平台核心机制

Java 技术栈如图1.2所示，程序的编译运行过程如图1.3所示。需要了解以下几个核心概念：

- Java 虚拟机
- 垃圾回收机制
- Java 运行时环境（Java Runtime Environment, JRE）

- JIT, Just-In-Time 传统解释器的解释执行是转换一条，运行完后就将其扔掉；JIT 会自动检测指令的运行情况，并将使用频率高（如循环运行）的指令解释后保存下来，下次调用时就无需再解释（相当于局部的编译执行），显著提高了 Java 的运行效率。



图 1.2 Java 技术栈

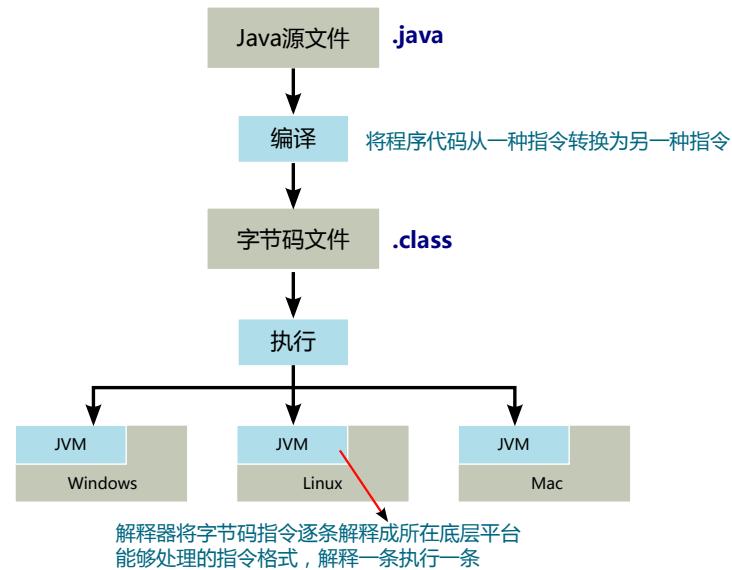


图 1.3 Java 程序编译运行过程

1.3 Java 开发环境

构建 Java 开发环境，需要首先获取和安装 Java 开发工具集，可以从 Oracle 官方网站链接 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 获取。下载完成后解压放入合适的磁盘目录下。

对于 Windows 操作系统，可以采用以下路径：

```
D:\Program Files\Java
```

对于 Linux 操作系统，可以采用以下路径：

```
/opt/jdk1.8.0_172
```

接下来进行环境变量配置，以 Windows 操作系统为例：

变量名	Path
------------	------

变量值 D:\Program Files\Java\jdk1.8.0_172\bin

配置完成后，可以看到 JDK 目录中包含以下子目录和文件：

```
1 bin COPYRIGHT db include javafx-src.zip  
2 jre lib LICENSE man README.html release src.zip  
3 THIRDPARTYLICENSEREADME-JAVAFX.txt THIRDPARTYLICENSEREADME.txt
```

对子目录的功能简要描述如下：

bin Java 开发工具，包括编译器、虚拟机、调试器、反编译器等；

jre Java 运行时，包括 Java 虚拟机、类库和其他资源文件；

lib 类库和所需支持性文件；

include 用于调试本地方法（底层平台）的 C++ 头文件；

src.zip 类库的源代码；

db Java DB 数据库，JDK6.0 新增项目，一种纯 Java 的关系型数据库。

1.4 Java 开发工具

业界普遍采用 Eclipse 或 IntelliJ IDEA 等集成开发环境进行 Java 大型工程开发，当然也可以采用文本编程工具 Vim 或 Emacs 等进行 Java 小型程序的开发。

本课程采用 Eclipse 作为首选集成开发环境。

1.5 Java 基本开发流程

本部分使用文本编程工具编写一个简单的 Java Hello World 程序，演示 Java 的基本开发和代码编译运行流程。首先，我们需要使用文本编程工具编写一个 Java 源文件 HelloWorld.java，文件命名必须与类名相同。

```
1 public class HelloWorld {  
2     public static void main(String[] args) {  
3         System.out.println("Hi, Java!");  
4     }  
5 }
```

然后，使用 `javac` 工具将源文件编译为字节码文件，编译完成后，我们可以看到生成了 `HelloWorld.class` 这一字节码文件。

```
1 > javac HelloWorld.java && ls  
2 HelloWorld.class  HelloWorld.java
```

接下来，我们使用 `java` 工具运行该程序，在终端正确打印了“Hi, Java”字符串。

```
1 > java HelloWorld  
2 Hi, Java!
```

说明

编写 Java 应用程序需掌握的几条规则如下：

1. Java 语言拼写是大小写敏感的（Case-Sensitive）；
2. 一个源文件中可以定义多个 Java 类，但其中最多只能有一个类被定义为 Public 类；
3. 如果源文件中包含了 `public` 类，则源文件必须和该 `public` 类同名；
4. 一个源文件包含多个 Java 类时，编译后会生成多个字节码文件，即每个类都会生成一个单独的“`.class`”文件，且文件名与类名相同。

☒ 2 ☒ Java 语言基础与流程控制

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第一周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. Java 语言基础包括: 数据类型、常量和变量、关键字与标识符、运算符与表达式、从键盘输入数据。
2. Java 流程控制包括: 语句和复合语句、分支结构 (选择结构)、循环结构、跳转语句。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

2.1 Java 语言基础

2.1.1 数据类型

Java 数据类型分为两大类：基本数据类型和引用数据类型。基本数据类型是由程序设计语言系统所定义、不可再划分的数据类型。所占内存大小固定，与软硬件环境无关，在内存中存放的是数据值本身。Java 的基本数据类型包括：整型（byte、short、int、long）、浮点型（float、double）、逻辑型（boolean）和字符型（char）。引用数据类型（复合数据类型）在内存中存放的是指向该数据的地址，不是数据值本身。引用数据类型包括类、数组、接口等。

数据类型的基本要素包括：

- 数据的性质（数据结构）
- 数据的取值范围（字节大小）
- 数据的存储方式
- 参与的运算

整型

Java 整型类型的数据位数及取值范围如表2.1所示。

表 2.1 整型数据类型

类型	数据位数	取值范围
byte（字节型）	8	$-2^7 \sim 2^7 - 1$
short（短整型）	16	$-2^{15} \sim 2^{15} - 1$
int（整型）（默认）	32	$-2^{31} \sim 2^{31} - 1$
long（长整型）（l 或 L）	64	$-2^{63} \sim 2^{63} - 1$

浮点型

Java 浮点型类型的数据位数及取值范围如表2.2所示。

表 2.2 整型数据类型

类型	数据位数	取值范围
float (单精度) (f 或 F)	32	$1.4E - 45 \sim 3.4E + 38$
double (双精度) (默认)	64	$4.9E - 324 \sim 1.8E + 308$

逻辑型

逻辑型又称为布尔型 (boolean)，布尔型数据类型的特性如下：

- 布尔型数据只有 true (真) 和 false (假) 两个取值。
- 布尔型数据存储占 1 个字节， 默认取值为 false。
- 布尔型数据 true 和 false 不能转换成数字表示形式。

字符型

- 字符型数据类型用来存储单个字符，采用的是 Unicode 字符集编码方案¹。
- 字符声明用单引号表示单个字符。
- 字符型数据可以转化为整型。

示例代码：字符数据类型示例

```
1 public class CharDemo {  
2     public static void main(String[] args) {  
3         char a = 'J';  
4         char b='Java'; //会报错  
5     }  
6 }
```

¹建议搜索理解什么是字符集和字符编码规则。

2.1.2 数据类型转换

数值型不同类型数据的转换

数值型不同类型数据之间的转换，**自动类型转换**需要符合以下条件：

1. 转换前的数据类型与转换后的类型兼容。
2. 转换后的数据类型的表示范围比转换前的类型大。
3. 条件 2 说明不同类型的数据进行运算时，需先转换为同一类型，然后进行运算。

转换从“短”到“长”的优先关系为：

byte → short → char → int → long → float → double

如果要将较长的数据转换成较短的数据时（不安全）就要进行**强制类型转换**，格式如下：

```
1 (预转换的数据类型) 变量名;
```

字符串型数据与数值型数据相互转换

示例代码：字符串数据转换为数值型数据示例

```
1 String myNumber = "1234.56";
2 float myFloat = Float.parseFloat(MyNumber);
```

字符串可用加号“+”来实现连接操作。若其中某个操作数不是字符串，该操作在连接之前会自动将其转换成字符串。所以可用加号来实现自动的转换。

示例代码：数值型数据转换成字符串数据示例

```
1 int myInt = 1234;           // 定义整形变量MyInt
2 String myString = "" + MyInt; // 将整形数据转换成了字符串
```

2.1.3 常量和变量

常量

整型常量 八进制、十六进制、十进制长整型后需要加 L 或 l。

浮点型常量 单精度后加 f 或 F，双精度后加 d 或 D 可省略。

逻辑型常量 true 或者 false。

字符型常量 单引号。

字符串常量 双引号。

示例代码：常量的声明

```
1 final int MAX = 10;  
2 final float PI = 3.14f;
```

变量

变量的属性包括变量名、类型、值和地址。Java 语言程序中可以随时定义变量，不必集中在执行语句之前。

示例代码：变量声明、初始化和赋值

```
1 int i, j = 0;  
2 i = 8;  
3 float k;  
4 k = 3.6f;
```

2.1.4 关键字与标识符

Java 的关键字（Java 保留字）如表2.3所示。

标识符是用来表示变量名、类名、方法名、数组名和文件名的有效字符序列。Java 语言对标识符的规定如下：

- 可以由字母、数字、下划线（_）、美元符号 (\$) 组合而成。
- 必须以字母、下划线或美元符号开头，不能以数字开头。
- 关键字不能当标识符使用。
- 区分大小写。

建议遵循驼峰命名，类名首字母大写，变量、方法及对象首字母小写的编码习惯。

表 2.3 Java 语言的关键字（保留字）

abstract	assert	boolean	break	byte	case
catch	char	class	continue	default	do
double	else	enum	extends	false	final
finally	float	for	if	implements	import
instanceof	int	interface	long	native	new
null	package	private	protected	public	return
short	static	super	switch	synchronized	this
volatile	throws	transient	true	try	void

2.1.5 运算符与表达式

按照运算符功能来分，Java 基本的运算符包括以下几类：

算术运算符	+、-、*、/、%、++、--
关系运算符	>、<、>=、<=、==、!=
逻辑运算符	!、&&、 、&、 、^
位运算符	>>、<<、>>>、&、 、^、~
赋值运算符	=、扩展赋值运算符，如+=、/=等
条件运算符	? :
其他运算符	包括分量运算符 .、下标运算符 []、实例运算符 instanceof、内存分配运算符 new、强制类型转换运算符 (类型)、方法调用运算符 () 等

图 2.1 Java 运算符

2.1.6 从键盘获得输入

由键盘输入的数据，不管是文字还是数字，Java 皆视为**字符串**，若是要由键盘输入获得数字则必须再经过类型转换。

示例代码：获得键盘输入字符串并转换为数字

```
1 import java.io.*;
```

```
1 public class MyClass {  
2     public static void main(String[] args) throws IOException {  
3         int num1, num2;  
4         String str1, str2;  
5         InputStreamReader in;  
6         in = new InputStreamReader(System.in);  
7         BufferedReader buf;  
8         buf = new BufferedReader(in);  
9         System.out.print("请输入第一个数: ");  
10        str1 = buf.readLine();           //将输入的内容赋值给字符串变量 str1  
11        num1 = Integer.parseInt(str1); //将 str1 转成 int 类型后赋给 num1  
12        System.out.print("请输入第二个数: ");  
13        str2 = buf.readLine();           //将输入的内容赋值给字符串变量 str2  
14        num2 = Integer.parseInt(str2); //将 str2 转成 int 类型后赋给 num2  
15        System.out.println(num1 + " * " + num2 + " = " + (num1 * num2));  
16    }  
17 }  
18 }
```

为了简化输入操作，从 JavaSE 5 版本开始在 `java.util` 类库中新增了一个类专门用于输入操作的类**Scanner**，可以使用该类输入一个对象。

示例代码：使用 Scanner 获得键盘输入并转换为特定数据类型

```
1 import java.util.*;  
2 public class MyClass {  
3     public static void main(String[] args)  
4     {  
5         Scanner reader = new Scanner(System.in);  
6         double num;  
7         num = reader.nextDouble(); //按照 double 类型读取键盘输入  
8         ...  
9     }  
10 }
```

`Scanner` 对象其他可用的数据读取方法包括：`nextByte()`、`nextDouble()`、`nextFloat()`、`nextInt()`、`nextLong()`、`nextShort()`、`next()`、`nextLine()`。

2.2 Java 流程控制

2.2.1 语句与复合语句

- Java 语言中语句可以是以分号 “;” 结尾的简单语句，也可以是用一对花括号 “{}” 括起来的复合语句。
- Java 中的注释形式：
 - 单行注释：//
 - 多行注释：/* */
 - 文件注释：/** */

2.2.2 分支结构

if 分支结构 1

if 双路条件结构
<pre>if(条件表达式){ 语句序列 1 } else { 语句序列 2 }</pre>
if 单路条件结构
<pre>if(条件表达式){ 语句序列 }</pre>

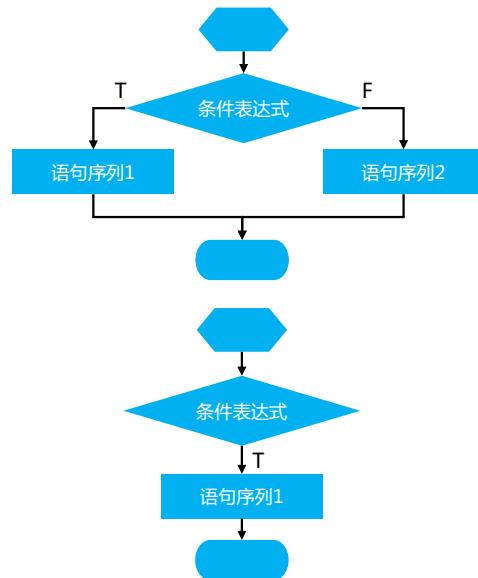


图 2.2 if 分支结构 1

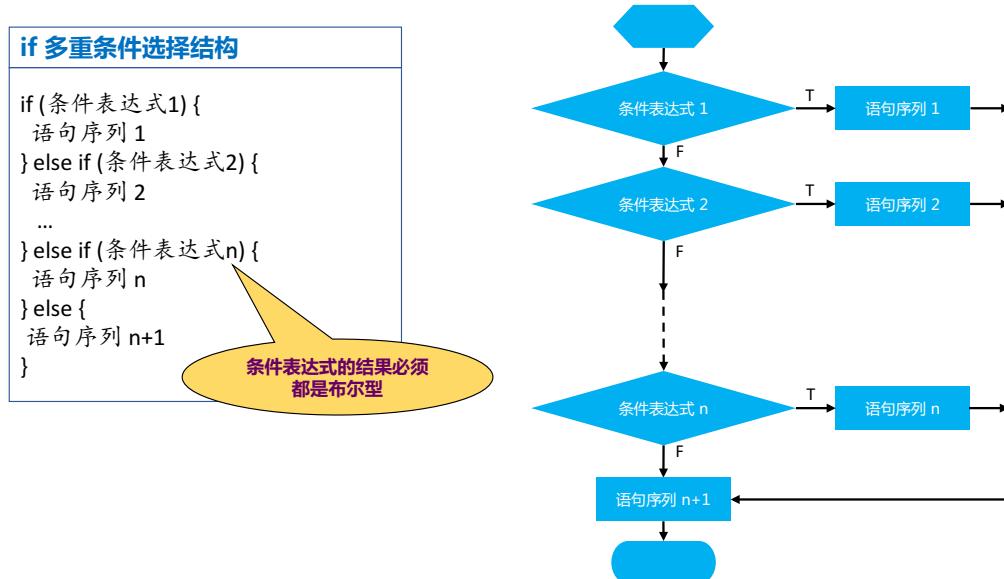


图 2.3 if 分支结构 2

if 分支结构 2

switch 分支结构

说明

在 Java 1.7 版本之后，switch 里表达式的类型可以为 String。

2.2.3 循环结构

while 循环

```

1 while(conditional expression) {
2     statements goes here ...
3 }
```

do-while 循环

```

1 do {
2     statements goes here ...
3 }
```

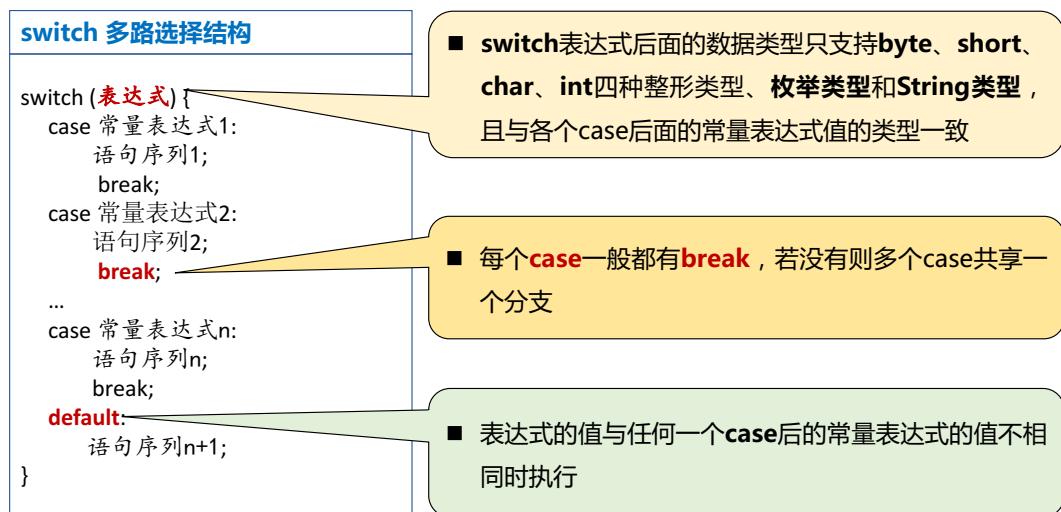


图 2.4 switch 分支结构

```

4   while (conditional expression);

```

for 循环 1

```

1   int [] integers = {1, 2, 3, 4};

3   for (int j = 0; j < integers.length; j++) {
4       int i = integers[j];
5       System.out.println(i);
6   }

```

for 循环 2

```

1   int [] integers = {1, 2, 3, 4};

3   for (int i : integers) {
4       System.out.println(i);
5   }

```

循环中的跳转

break 语句 使程序的流程从一个语句块（switch 或循环结构）内跳出。

continue 语句 终止当前这一轮（次）的循环，进入下一轮（次）循环。

return 语句 用来使程序从方法（函数）中返回，可返回一个值。

2.3 课后习题

2.3.1 简答题

1. Java 语言定义类哪些基本数据类型？其存储结构分别是什么样的？
2. 自动类型转换的前提是什么？转换时的优先级顺序如何？
3. 数字字符串转换为数值类型数据时，可以使用的方法有哪些？

2.3.2 小编程

1. 编写程序，从键盘输入一个浮点数，然后将该浮点数的整数部分输出。
2. 编写程序，从键盘输入 2 个整数，然后计算它们相除后得到的结果并输出，注意排除 0 除问题。

实验设计

实验名称: Eclipse 集成开发环境配置及 Java 语言基础编程练习

上机时间: 第一周

实验手册: 无（参照实验内容完成）

实验内容: 本次实验需要完成以下内容：

1. 使用文本编辑器完成 Java Hello World 程序编写，使用 javac 和 java 编译运行该程序；
2. 熟悉 Eclipse 集成开发环境，学习创建 Java 工程，使用 Maven 创建 Java 工程；
3. 根据授课幻灯片和讲义，尝试实现其中所有的示例代码。

实验要求: 本次实验不需要提交实验报告。

☒ 3 ☒ Java 数组和字符串

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第二周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握 Java 数组的概念
2. 学会一维数组和二维数组的使用; 认识 Arrays 类, 掌握操作数组相关方法
3. 掌握 Java 字符串的概念, 字符串与数组的关系; 学会 String 类常用字符串操作方法

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

3.1 数组的概念

数组是相同数据类型的元素按一定顺序排列的集合。在 Java 语言中，数组元素既可以为基本数据类型，也可以为对象。

Java 的内存分配（基础）

栈内存 存放定义的基本类型的变量和对象的引用变量，超出作用域将自动释放。

堆内存 存放由 new 运算符创建的对象和数组，由 Java 虚拟机的自动垃圾回收器来管理。

Java 数组的主要特点包括以下方面：

- 数组是相同数据类型的元素的集合；
- 数组中的各元素有先后顺序，它们在内存中按照这个先后顺序连续存放；
- 数组的元素用整个数组的名字和它自己在数组中的顺序位置来表示。

例如，`a[0]` 表示名字为 `a` 的数组中的第一个元素，`a[1]` 表示数组 `a` 的第二个元素，依次类推。

3.2 一维数组

3.2.1 创建数组

创建 Java 数组一般需经过三个步骤：

1. 声明数组；
2. 创建内存空间；
3. 创建数组元素并赋值。

示例代码：一维数组创建声明和内存分配

```
1 int [] x; //声明名称为x的int型数组，未分配内存给数组
```

```
2 x = new int[10]; //x中含有10个元素，并分配空间
```

```
1 int [] x = new int[10]; //声明数组并动态分配内存
```

说明

用 `new` 分配内存的同时，数组的每个元素都会自动赋默认值，整型为 0，实数为 0.0，布尔型为 `false`，引用型为 `null`。

3.2.2 一维数组的初始化

若在声明数组时进行赋值即初始化，称为静态内存分配。

```
1 数据类型[] 数组名 = {初值0, 初值1, …, 初值n};
```

示例代码：一维数组静态初始化

```
1 int [] a = {1,2,3,4,5};
```

注意

在 Java 程序中声明数组时，无论用何种方式定义数组，都不能指定其长度。

3.3 二维数组

Java 中无真正的多维数组，只是数组的数组。

3.3.1 二维数组的声明和内存分配

```
1 数据类型 [][] 数组名;
2 数组名 = new 数据类型 [行数][列数];
3 数据类型 [][] 数组名 = new 数据类型 [行数][列数];
```

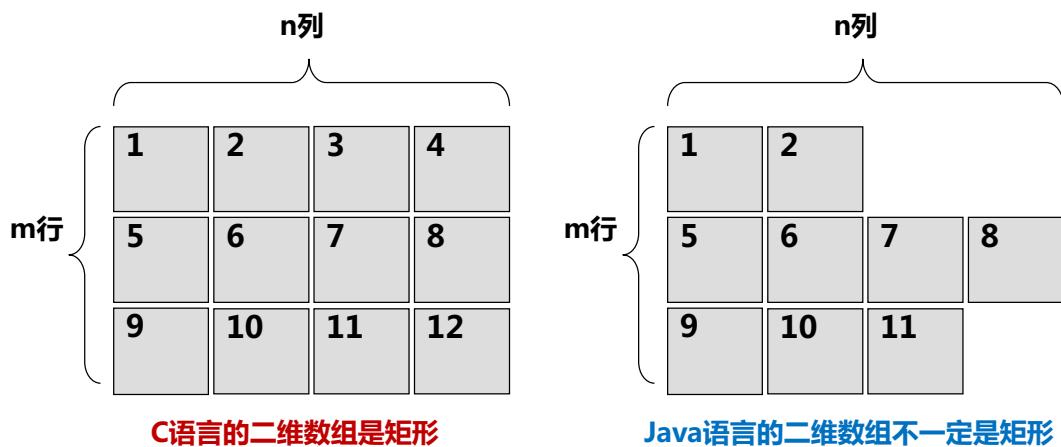


图 3.1 Java 和 C 语言数数组

3.3.2 二维数组定义的含义

- Java 中的二维数组看作是由多个一维数组构成
- 二维数组申请内存必须指定 **高层维数**

```
int[][] myArray1 = new int[10][];
int[][] myArray2 = new int[10][3];
```

- `int[][] x;`

表示定义了一个数组引用变量 `x`, 第一个元素为 `x[0]`, 最后一个为 `x[n-1]`, 其长度不确定

- `x = new int[3][];`

表示数组 `x` 有三个元素, 每个元素都是 `int[]` 类型的一维数组, 分别为 `int x[0][]`、`int[] x[1]`、`int[] x[2]`

```
x[0] = new int[3]; x[1] = new int[2];
```

给 `x[0]`、`x[1]`、`x[2]` 赋值 (长度可以不一样)

3.3.3 二维数组赋初值

```
int [][] a = {{11,22,33,44}, {66,77,88,99}};
```

注意

声明多维数组并初始化时不能指定其长度，否则出错。

3.4 Arrays 类

java.util.Arrays 工具类能方便地操作数组，它提供的所有方法都是静态的。该类具有以下功能：

给数组赋值 通过 fill 方法。

对数组排序 通过 sort 方法。

比较数组 通过 equals 方法比较数组中元素值是否相等。

查找数组元素 通过 binarySearch 方法能对排序好的数组进行二分查找法操作。

复制数组 把数组复制成一个长度为 length 的新数组。

示例代码：Array 操作示例

```
1  /*
2  * 数组比较 equals
3  */
4  String[] str1 = { "1", "2", "3" };
5  String[] str2 = { "1", "2", new String("3") };
6  System.out.println(Arrays.equals(str1, str2)); // 结果是true

8  /*
9  * 数组排序 sort
10 */
11 int[] score = { 79, 65, 93, 64, 88 };

13 // 将数组转换为字符串
14 String str = Arrays.toString(score);
15 System.out.println("原数组为：" + str);

17 Arrays.sort(score); // 作用是把一个数组按照有小到大进行排序，会改变原score
18 // 而不是创建新对象

19 // 将数组转换为字符串
20 System.out.println("排序后数组为：" + Arrays.toString(score));
```

```
22  /*
23  * 把数组中的所有元素替换成一个值 fill
24  */
25  int [] num = { 1, 2, 3 };
26  Arrays.fill (num, 6); // 参数1: 数组对象; 参数2: 替换的值
27  System.out.println(Arrays.toString(num)); // 打印结果: [6, 6, 6]

29  /*
30  * 通过二分法查询元素值在数组中的下标 binarySearch
31  */
32  char [] a = { 'a', 'b', 'c', 'd', 'e' };
33  int i = Arrays.binarySearch(a, 'd');
34  System.out.println(i); // 结果是: 3

36  char [] b = { 'e', 'a', 'c', 'b', 'd' };
37  Arrays.sort(b);
38  int j = Arrays.binarySearch(b, 'e');
39  System.out.println(j); // 结果是: 4

41  /*
42  * 把数组内容复制到一个新数组中 copyOf
43  */
44  int [] c = { 1, 2, 3 };
45  int [] d = Arrays.copyOf(c, c.length + 2); // 参数1: 原数组 参数2: 新数组的长度
46  System.out.println("原数组为: " + Arrays.toString(c));
47  System.out.println("复制后的新数组为: " + Arrays.toString(d));
```

3.5 字符串

字符串是用一对双引号括起来的字符序列。Java 语言中，字符串常量或变量均用类实现。

3.5.1 字符串变量的创建

示例代码：格式 1

```
1 String s; //声明字符串型引用变量s, 此时s的值为null  
2 s = new String("Hello"); //在堆内存中分配空间, 并将s指向该字符串首地址
```

示例代码：格式 2

```
1 String s = new String("Hello");
```

示例代码：格式 3

```
1 String s = "Hello";
```

3.5.2 String 类的常用方法

示例代码：求字符串长度

```
1 String str = new String("asdfzxc");  
2 int strlength = str.length(); //strlength = 7
```

示例代码：获取字符串某一位置字符

```
1 char ch = str.charAt(4); //ch = z
```

示例代码：提取子串

```
1 String str2 = str1.substring(2); //str2 = "dfzxc"  
2 String str3 = str1.substring(2,5); //str3 = "dfz"
```

示例代码：字符串连接

```
1 String str = "aa".concat("bb").concat("cc");
2 String str = "aa" + "bb" + "cc"; // 相当于上一行
```

示例代码：字符串比较

```
1 String str1 = new String("abc");
2 String str2 = new String("ABC");
3 int a = str1.compareTo(str2); //a>0
4 int b = str1.compareTo(str2); //b=0
5 boolean c = str1.equals(str2); //c=false
6 boolean d = str1.equalsIgnoreCase(str2); //d=true
```

示例代码：字符串中字符的大小写转换

```
1 String str = new String("asDF");
2 String str1 = str.toLowerCase(); //str1 = "asdf"
3 String str2 = str.toUpperCase(); //str2 = "ASDF"
```

示例代码：字符串中字符的替换

```
1 String str = "asdzxcasd";
2 String str1 = str.replace('a', 'g'); //str1 = "gsdzxcgsd"
3 String str2 = str.replace("asd", "fgh"); //str2 = "fghzxcfgh"
4 String str3 = str.replaceFirst ("asd", "fgh"); //str3 = "fghzxcasd"
5 String str4 = str.replaceAll ("asd", "fgh"); //str4 = "fghzxcfgh"
```

3.5.3 理解 Java 字符串

示例代码：String.java 部分代码

```
1 public final class String
2 implements java.io.Serializable, Comparable<String>, CharSequence { //1
3
4     /** The value is used for character storage. */
```

```
5     private final char value[]; //2

7     /** The offset is the first index of the storage that is used. */
8     private final int offset;

10    /** The count is the number of characters in the String. */
11    private final int count;

13    /** Cache the hash code for the string */
14    private int hash; // Default to 0

16    /** use serialVersionUID from JDK 1.0.2 for interoperability */
17    private static final long serialVersionUID = -6849794470754667710L;
18    .....

20    public String substring(int beginIndex, int endIndex) { //3
21        if (beginIndex < 0) {
22            throw new StringIndexOutOfBoundsException(beginIndex);
23        }
24        if (endIndex > count) {
25            throw new StringIndexOutOfBoundsException(endIndex);
26        }
27        if (beginIndex > endIndex) {
28            throw new StringIndexOutOfBoundsException(endIndex - beginIndex);
29        }
30        return ((beginIndex == 0) && (endIndex == count)) ? this :
31            new String(offset + beginIndex, endIndex - beginIndex, value);
32    }
33 }
```

1. String 类是 final 类，即意味着 String 类不能被继承，并且它的成员方法都默认为 final 方法。
2. 从 String 类的成员属性可以看出 String 类其实是通过 char 数组来保存字符串的。
3. 无论是 substring 还是 concat 操作等都不是在原有的字符串上进行的，而是重新生成了一个新的字符串对象，最原始的字符串并没有被改变。

说明

String 对象一旦被创建就是固定不变的，对 String 对象的任何操作都不影响到原对象，而是会生成新的对象。

☒ 4 ☒ Java 面向对象编程进阶 A

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第二周（根据校历，本周有两次课）

参考教材: 本课程参考教材及资料如下：

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握 Java 包、继承、访问控制、方法重写的概念、机制和使用方法
2. 理解 Java 关键字 super 和关键字 this, 特别了解其指代的对象, 编程中的用法

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

4.1 包

为便于管理大型软件系统中数目众多的类，解决类的命名冲突问题以及进行访问控制，Java 引入包（package）机制，即将若干功能相关的类逻辑上分组打包到一起，提供类的多重类命名空间。

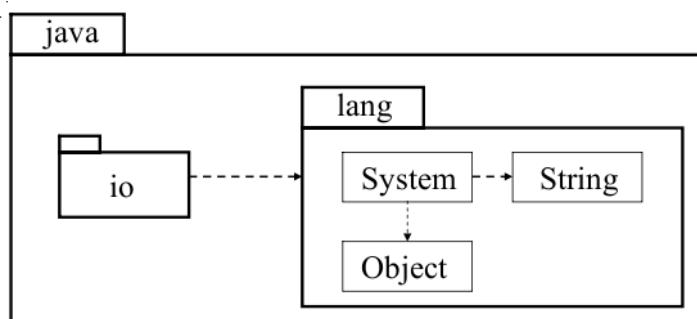


图 4.1 Java 包

4.1.1 JDK 常用包

JDK API 中的常用包如表所示。

表 4.1 JDK API 常用包

包名	功能说明	包的含义
java.lang	Java 语言程序设计的基础类	language 的简写
java.awt	创建图形用户界面和绘制图形图像的相关类	抽象窗口工具集
java.util	集合、日期、国际化、各种实用工具	utility 的简写
java.io	可提供数据输入/输出相关功能的类	input/output 的简写
java.net	Java 网络编程的相关功能类	网络
java.sql	提供数据库操作的相关功能类	结构化查询语言的简写

4.1.2 包的创建

package 语句作为 Java 源文件的第一条语句，指明该文件中定义的类所在的包（若缺省该语句，则指定为无名包）。语法格式如下：

```
1 package pkg1[.pkg2[.pkg3 ...]];
```

示例代码：创建包

```
1 package p1;
2 public class Test {
3     public void m1() {
4         System.out.println("In class Test, method m1 is running!");
5     }
6 }
```

package 语句对所在源文件中定义的所有类型（包括接口、枚举、注解）均起作用。Java 编译器把包对应于文件系统的目录管理，package 语句中，用“.”来指明包（目录）的层次。如果在程序 Test.java 中已定义了包 p1，编译时采用如下方式：

```
1 > javac Test.java
```

则编译器会在当前目录下生成 Test.class 文件。

若在命令行下使用如下命令：

```
1 > java -d /home/xiaodong/work01 Test.java
```

“-d /home/xiaodong/work01”是传给 Java 编译器的参数，用于指定此次编译生成的.class 文件保存到该指定路径下，并且如果源文件中有 package 语句，则编译时会自动在目标路径下创建与包同名的目录 p1，再将生成的 Test.class 文件保存到该目录下。

4.1.3 导入包中的类

为使用定义在不同包中的 Java 类，需用 import 语句来引入所需要的类。语法格式：

```
1 import pkg1[.pkg2 ...].( classname|*);
```

示例代码：导入和使用有名包中的类

```
1 import p1.Test; //or import p1.*;
2 public class TestPackage{
3     public static void main(String args[]){
4         Test t = new Test();
5         t.m1();
6     }
7 }
```

4.1.4 Java 包特性

一个类如果未声明为 `public` 的，则只能在其所在包中被使用，其他包中的类即使在源文件中使用 `import` 语句也无法引入它。可以不在源文件开头使用 `import` 语句导入要使用的有名包中的类，而是在程序代码中每次用到该类时都给出其完整的包层次，例如：

```
1 public class TestPackage{
2     public static void main(String args[]){
3         p1.Test t = new p1.Test();
4         t.m1();
5     }
6 }
```

4.2 继承

4.2.1 继承的概念

继承（Inheritance）是面向对象编程的核心机制之一，其本质是在已有类型基础上进行扩充或改造，得到新的数据类型，以满足新的需要。

根据需要定义 Java 类描述“人”和“学生”信息，示例代码如下：

示例代码：Class Person

```
1 public class Person {  
2     public String name;  
3     public int age;  
4     public Date birthDate;  
5     public String getInfo() {...}  
6 }
```

示例代码：Class Student

```
1 public class Student {  
2     public String name;  
3     public int age;  
4     public Date birthDate;  
5     public String school;  
6     public String getInfo() {...}  
7 }
```

我们可以通过继承简化 Student 类的定义：

示例代码：Class Student extends Person

```
1 public class Student extends Person {  
2     public String school;  
3 }
```

Java 类声明的语法格式如下：

```
1 [< 修饰符 >] class < 类名 > [extends < 父类名 >] {  
2     [< 属性声明 >]  
3     [< 构造方法声明 >]  
4     [< 方法声明 >]  
5 }
```

Object 类是所有 Java 类的最高层父类，如果在类的声明中未使用 extends 关键字指明其父类，则默认父类为 Object 类。

Java 只支持单继承，不允许多重继承。即：

- 一个子类只能有一个父类；
- 一个父类可以派生出多个子类。

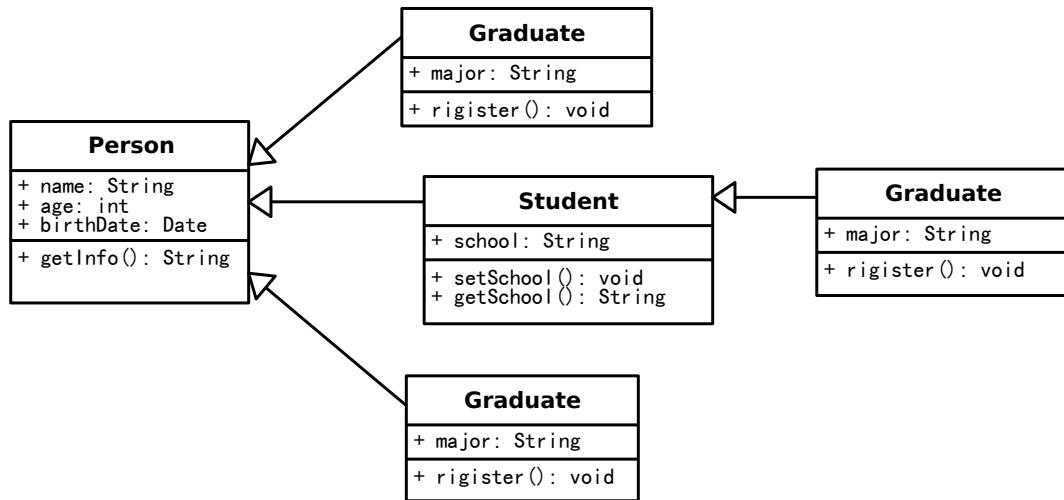


图 4.2 Java 包

4.2.2 类之间的关系

依赖关系 一个类的方法中使用到另一个类的对象（uses-a）¹。

聚合关系 一个类的对象包含（通过属性引用）了另一个类的对象（has-a）²。

泛化关系 一般化关系（is-a），表示类之间的继承关系、类和接口之间的实现关系以及接口之间的继承关系。

4.3 访问控制

访问控制是指对 Java 类或类中成员的操作进行限制，即规定其在多大的范围内可以被直接访问。

4.3.1 类的访问控制

在声明 Java 类时可以在 class 关键字前使用 public 来修饰，也可以不使用该修饰符。public 的类可在任意场合被引入和使用，而非 public 的类只能在其所在包中被使

¹ 车能够装载货物，车的装载功能（load() 方法）对货物（goods）有依赖。

² 车有发动机、车轮等，Car 对象是由 Engine 等对象构成的。

用。

4.3.2 类中成员的访问控制

表 4.2 类成员的访问控制

修饰符/作用范围	同一个类	同一个包	子类	任何地方
public	yes	yes	yes	yes
protected	yes	yes	yes	no
无修饰符	yes	yes	no	no
private	yes	no	no	no

4.3.3 访问控制注意的一些问题

- 一般不提倡将属性声明为 public 的，而构造方法和需要外界直接调用的普通方法则适合声明为 public 的。
- 在位于不同的包内，必须是子类的对象才可以直接访问其父类的 protected 成员，而父类自身的对象反而不能访问其所在类中声明的 protected 成员。
- 所谓“访问控制”只是控制对 Java 类或类中成员的直接访问，而间接访问是不做控制的，也不该进行控制。

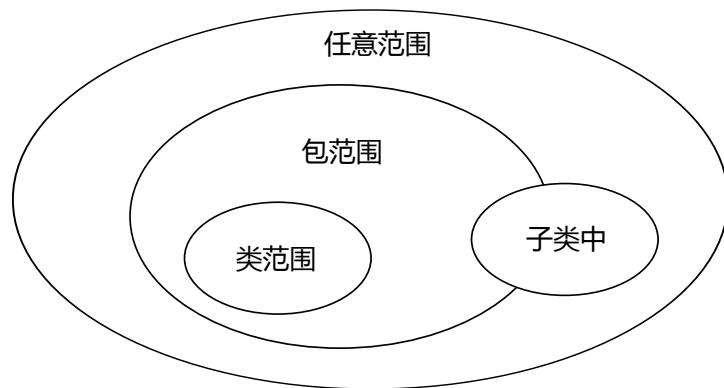


图 4.3 Java 访问控制

4.3.4 访问控制 protected

示例代码: A.java

```
1 package p1;
2 public class A {
3     public int m = 5;
4     protected int n = 6;
5 }
```

示例代码: B.java

```
1 package p2;
2 import p1.A;
3 public class B extends A {
4     public void mb() {
5         m = m + 1;
6         n = n * 2;
7     }
8
9     public static void main(String[] args) {
10        B b = new B();
11        b.m = 7; // 合法
12        b.n = 8; // 合法
13        A a = new A();
14        a.m = 9 // 合法
15        a.n = 10 // 非法
16    }
17 }
```

4.4 同名问题

4.4.1 方法重写

在子类中可以根据需要对从父类中继承来的方法进行重新定义，此称方法重写（Override）或覆盖。

- 重写方法必须和被重写方法具有相同的方法名称、参数列表和返回值类型；
- 重写方法不能使用比被重写方法更严格的访问权限；
- 重写方法不允许声明抛出比被重写方法范围更大的异常类型。

[示例代码：方法重写示例：Person.java](#)

```
1 public class Person {  
2     String name;  
3     int age;  
4     public String getInfo() {  
5         return "Name:" + name + "\t" + "age:" + age;  
6     }  
7 }
```

[示例代码：方法重写示例：Student.java](#)

```
1 public class Student extends Person {  
2     private String school;  
3     public void setSchool(String scholl) {  
4         this.school = school;  
5     }  
6     public String getSchool(){  
7         return school;  
8     }  
9     public String getInfo() {  
10        return "Name:" + name + "\tAge:" + age + "\tSchool:" + school;  
11    }  
12 }
```

示例代码：方法重写示例：Parent.java

```
1 public class Parent {  
2     public void method1() {...}  
3 }
```

示例代码：方法重写示例：Child.java

```
1 public class Child extends Parent {  
2     private void method1() {} //非法，权限更严格  
3 }
```

4.4.2 同名属性

```
1 public class Person {  
2     int age = 5;  
3     public int getAge() {  
4         return age;  
5     }  
6     public int getInfo() {  
7         return age;  
8     }  
9 }
```

```
1 public class Student extends Person {  
2     int age = 6;  
3     public int getAge() {  
4         return age;  
5     }  
6 }
```

```
1 public class Test {  
2     public static void main(String args[]) {  
3         Person p = new Person();  
4         System.out.println(p.getAge());  
5         Student s = new Student();  
6         System.out.println(s.age);  
7         System.out.println(s.getAge());  
8         System.out.println(s.getInfo());  
9     }  
10 }
```

上述程序的输出结果为：

output

```
5  
6  
6  
5
```

对上述 Student 对象同名属性的几点说明

1. 以“对象名. 属性名”方式直接访问时，使用的是子类中添加的属性 age；
2. 调用子类添加或者重写的方法时，方法中使用的是子类定义的属性 age；
3. 调用父类中定义的方法时，方法中使用的是父类中的属性 age；
4. 可以理解为“**层次优先就近原则**”，在哪个层次中的代码，就优先使用该层次类中定义的属性。不提倡使用同名属性。

4.4.3 关键字 super

在存在命名冲突（子类中存在方法重写或添加同名属性）的情况下，子类中的代码将自动使用子类中的同名属性或重写后的属性。当然也可以在子类中**使用关键字 super 引用父类中的成分**：

访问父类中定义的属性

```
1 super.<属性名>
```

调用父类中定义的成员方法

```
1 super.<方法名>(<实参列表>)
```

子类构造方法中调用父类的构造方法

```
1 super(<实参列表>)
```

super 的追溯不仅限于直接父类，而是先从直接父类开始查找，如果找不到则逐层上溯，一旦在某个层次父类中找到匹配成员即停止追溯并使用该成员。

示例代码：super 用法示例 A

```
1 class Animal {  
2     protected int i = 1; //用于测试同名属性，无现实含义  
3 }  
  
5 class Person extends Animal {  
6     protected int i = 2; //用于测试同名属性，无现实含义  
7     private String name = "Tom";  
8     private int age = 9;  
9     public String getInfo() {  
10         return "Name:" + name + "\tAge:" + age;  
11     }  
12     public void testI() {  
13         System.out.println(super.i);  
14         System.out.println(i);  
15     }  
16 }
```

示例代码：super 用法示例 B

```
1 class Student extends Person {  
2     private int i = 3;  
3     private String school = "THU";  
4     public String getInfo() {      //重写方法  
5         return super.getInfo() + "\tSchool:" + school;  
6     }  
7     public void testI() {      //重写方法  
8         System.out.println(super.i);  
9         System.out.println(i);  
10    }  
11 }  
12 public class Test {  
13     public static void main(String args[]) {  
14         Person p = new Person();  
15         System.out.println(p.getInfo());  
16         p.testI();  
17         Student s = new Student();  
18         System.out.println(s.getInfo());  
19         s.testI();  
20     }  
21 }
```

上述代码的输出结果为：

output

Name:Tom Age:9

1

2

Name:Tom Age:9 School:THU

2

3

4.4.4 关键字 this

在 Java 方法中，不但可以直接使用方法的局部变量，也可以使用调用该方法的对象。为解决可能出现的命名冲突，Java 语言引入 this 关键字来标明方法的当前对象。分

为两种情况：

- 在普通方法中，关键字 this 代表方法的调用者，即本次调用了该方法的对象；
- 在构造方法中，关键字 this 代表该方法本次运行所创建的那个新对象。

this 作为一个特殊的引用类型变量，可以通过“**this. 成员**”的方式访问其引用的当前对象的属性和方法。

示例代码：this 用法示例

```
1 public class MyDate {  
2     private int day = 17;  
3     private int month = 2;  
4  
5     public MyDate(int day, int month) {  
6         this.day = day; // A  
7         this.month = month;  
8     }  
9     ... // Some methods  
10  
11    public void setAll(int day, int month) {  
12        this.setMonth(month); // B  
13        this.setDay(day);  
14    }  
15 }
```

关于 this 的归纳说明

1. 在 Java 方法中直接给出变量名而不是“对象名. 变量名”的方式访问一个变量，系统首先尝试作为局部变量来处理；如果方法中不存在该名字的局部变量，才会到方法当前对象的成员变量中查找。
2. 在 Java 方法中直接调用一个方法而不指定其调用者时，则默认调用者为当前对象 this。

实验设计

☒ 5 ☒ Java 面向对象编程进阶 B

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第二周（根据校历，本周有两次课）

参考教材: 本课程参考教材及资料如下：

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解多态和虚方法调用的概念，掌握其用法
2. 掌握方法重载的方法
3. 掌握 static 属性、方法和初始化块的用法
4. 了解设计模式，掌握单例设计模式
5. 掌握 final 关键字的概念和使用方法

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

5.1 多态性

5.1.1 多态的概念

在 Java 中，子类的对象可以替代父类的对象使用称为**多态**。Java 引用变量与所引用对象间的类型匹配关系如下：

- 一个对象只能属于一种确定的数据类型，该类型自对象创建直至销毁不能改变。
- 一个引用类型变量可能引用（指向）多种不同类型的对象——既可以引用其声明类型的对象，也可以引用其声明类型的子类的对象。

```
1 Person p = new Student(); //Student 是 Person 的子类
```

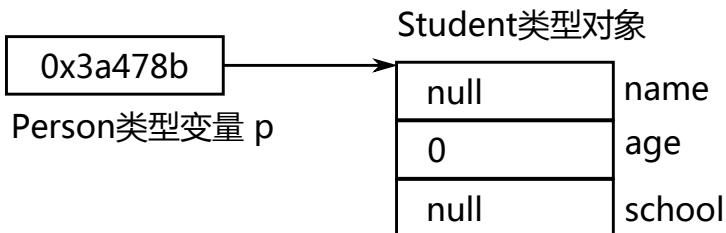


图 5.1 Java 多态

多态性同样适用与引用类型数组元素。

```
1 Person[] p = new Person[3];
2   p[0] = new Student(); // 假设 Student 类继承了 Person 类
3   p[1] = new Person();
4   p[2] = new Graduate(); //假设 Graduate 类继承了 Student 类
```

一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，该变量则不能再访问子类中添加的属性和方法，这体现了父类引用对子类对象的能力屏蔽性。

```
1 Student m = new Student();
2   m.setSchool("ouc"); // 合法
```

```
3 Person e = new Student();  
4 e.setSchool("ouc"); // 非法
```

5.1.2 多态用法示例

示例代码：Person.java

```
1 public class Person {...}
```

示例代码：Student.java

```
1 public class Student extends Person {  
2     private String school;  
  
4     public void setSchool(String school) {  
5         this.school = school;  
6     }  
  
8     public String getSchool() {  
9         return school;  
10    }  
  
12    @Override  
13    public String getInfo() {  
14        return super.getInfo() + "\tSchool: " + school;  
15    }  
16 }
```

示例代码：PolymorphismSample.java

```
1 public class PolymorphismSample {  
2     public void show(Person p) {  
3         System.out.println(p.getInfo());  
4     }  
.....
```

```
6 public static void main(String[] args) {  
7     PolymorphismSample ps = new PolymorphismSample();  
8     Person p = new Person();  
9     ps.show(p);  
10    Student s = new Student();  
11    ps.show(s);  
12}  
13}
```

多态提升方法通用性

以上代码中，`show()` 方法既可以处理 `Person` 类型的数据，又可以处理 `Student` 类型的数据，乃至未来定义的任何 `Person` 子类类型的数据，即不必为相关的每种类型单独声明一个处理方法，提高了代码的通用性。

5.1.3 虚方法调用

思考：一个引用类型的变量如果声明为父类的类型，但实际引用的是子类对象，则该变量就不能再访问子类中添加的属性和方法。但如果此时调用的是父类中声明过、且在子类中重写过的方法，情况如何？

补充代码。

5.1.4 对象造型

引用类型数据值之间的强制类型转换称为**造型**（Casting）。造型以下几种情况需要注意：

- 从子类到父类的类型转换可以自动进行。

```
1 Person p = new Student();
```

- 在多态的情况下，有时我们可能需要恢复一个对象的本来面目，以发挥其全部潜力。从父类到子类的类型转换必须通过造型实现。

```
1 Person p1 = new Student();  
2 Student s1 = (Student)p1; // 合法  
3 Person p2 = new Person();
```

```
4     Student s2 = (Student)p2; // 非法
```

3. 无继承关系的引用类型间的转换是非法的。

```
1     String s = "Hello World!";
2     Person p = (Person)s; // 非法
```

5.1.5 instanceof 运算符

如果运算符 instanceof 左侧的变量当前时刻所引用的对象的**真正类型**是其右侧给出的类型**或者是其子类**，则整个表达式的结果为 true。

```
1 class Person { --- }
2 class Student extends Person { --- }

4 public class Tool {
5     public void distribute(Person p) {
6         if (p instanceof Student) {
7             System.out.println("处理 Student 类型及其子类类型对象");
8         } else {
9             System.out.println("处理 Person 类型及其子类类型对象");
10        }
11    }
12 }
```

```
1 public class Test() {
2     public static void main(String[] args) {
3         Tool t = new Tool();
4         Student s = new Student();
5         t.distribute(t);
6     }
7 }
```

5.1.6 虚方法调用和造型

课程配套代码 ➔ package sample.oop.poly

- VirtualMethodSample.java
- Person.java
- Student.java

强调以下与虚方法调用和造型相关的重点：

- 系统根据运行时对象的真正类型来确定具体调用哪一个方法，这一机制被称为**虚方法调用**。
- 造型是引用类型数据值之间的强制类型转换。
- instanceof 运算符判断的是当前所引用对象的真正类型是什么，而不是声明的引用类型。

5.2 方法重载

5.2.1 方法重载的概念

在一个类中存在多个同名方法的情况称为**方法重载** (Overload)。Java 对方法重载有以下要求：

- 重载方法参数列表必须不同。
- 重载既可以用于普通方法，也可以用于构造方法。

课程配套代码 ➔ sample.oop.MethodOverloadSample.java

5.2.2 调用重载的构造方法

使用 this 调用当前类中重载构造方法

可以在构造方法的第一行使用关键字 this 调用其它（重载的）构造方法。

```
1 public class Person {  
2     ...  
3     public Person(String name,int age) {
```

```
4     this.name = name;
5     this.age = age;
6 }
7 public Person(String name) {
8     this(name, 18);
9 }
10 ...
11 }
```

注意

关键字 this 的此种用法只能用在构造方法中，且 this() 语句如果出现必须位于方法体中代码的第一行。

使用 super 调用父类构造方法

示例代码：Person.java

```
1 public class Person {
2     ... (此处没有无参构造方法)
3     public Person(String name, int age) {
4         this.name = name;
5         this.age = age;
6     }
7     ...
8 }
```

示例代码：Student.java

```
1 public class Student extends Person {
2     private String school;
3     public Student(String name, int age, String school) {
4         super(name, age); // 显式调用父类有参构造方法
5         this.school = school;
6     }
7     public Student(String school) { // 编译出错
8 }
```

```
8     // super(); // 隐式调用父类有参构造方法，则自动调用父类无参构造方法
9     this.school = school;
10    }
11 }
```

上述代码为什么会出现编译错误？

在 Java 类的构造方法中一定直接或间接地调用了其父类的构造方法（Object 类除外）。

1. 在子类的构造方法中可使用 super 语句调用父类的构造方法，其格式为 super(<实参列表>)。
2. 如果子类的构造方法中既没有显式地调用父类构造方法，也没有使用 this 关键字调用同一个类的其他重载构造方法，则系统会默认调用父类无参数的构造方法，其格式为 super()。
3. 如果子类构造方法中既未显式调用父类构造方法，而父类中又没有无参的构造方法，则编译出错。

课程配套代码 ➔ sample.oop.ConstructorOverloadSample.java

5.2.3 对象构造/初始化细节

第一阶段 为新建对象的实例变量分配存储空间并进行默认初始化。

第二阶段 按下述步骤继续初始化实例变量：

1. 绑定构造方法参数；
2. 如有 this() 调用，则调用相应的重载构造方法然后跳转到步骤 5；
3. 显式或隐式追溯调用父类的构造方法（Object 类除外）；
4. 进行实例变量的显式初始化操作；
5. 执行当前构造方法的方法体中其余的语句。

5.3 关键字 static

在 Java 类中声明**属性、方法和内部类**时，可使用关键字 static 作为修饰符。

- static 标记的属性或方法由整个类（所有实例）共享，如访问控制权限允许，可不必创建该类对象而直接用类名加“.”调用。
- static 成员也称“**类成员**”或“**静态成员**”，如“类属性”、“类变量”、“类方法”和“静态方法”等。

5.3.1 static 属性和方法

static 属性

- static 属性由其所在类（包括该类所有的实例）共享。
- 非 static 属性则必须依赖具体/特定的对象（实例）而存在。

static 方法

要在 static 方法中调用其所在类的非 static 成员，应首先创建一个该类的对象，通过该对象来访问其非 static 成员。

课程配套代码 ➔ sample.oop.StaticMemberAndMethodSample.java

5.3.2 初始化块

static 初始化块

在类的定义体中，方法的外部可包含 static 语句块，**static 块仅在其所属的类被载入时执行一次**，通常用于初始化 static（类）属性。

非 static 初始化块

非 static 的初始化块在创建对象时被自动调用。

课程配套代码 ➔ sample.oop.StaticInitBlockSample.java

5.3.3 静态导入

静态导入用于在一个类中导入其他类或接口中的 static 成员，语法格式如下：

```
1 import static <包路径>.<类名>.*  
3 import static <包路径>.<类名>.<静态成员名>
```

示例代码：静态导入应用示例

```
1 import static java.lang.Math.*;
2 public class Test {
3     public static void main(String[] args) {
4         double d = sin(PI * 0.45);
5         System.out.println(d);
6     }
7 }
```

5.3.4 Singleton 设计模式

所谓“模式”就是被验证为有效的常规问题的典型解决方案。设计模式（Design Pattern）在面向对象分析设计和软件开发中占有重要地位。好的设计模式可以使我们更加方便的重用已有的成功设计和体系结构，极大的提高代码的重用性和可维护性。

经典设计模式分类主要分为以下三大类：

创建型模式 涉及对象的实例化，特点是不让用户代码依赖于对象的创建或排列方式，避免用户直接使用 new 创建对象。

工厂方法模式、抽象工厂方法模式、生成器模式、原型模式和单例模式

行为型模式 涉及怎样合理的设计对象之间的交互通信，以及合理为对象分配职责，让设计富有弹性、易维护、易复用。

责任链模式、命令模式、解释器模式、迭代器模式、中介者模式、备忘录模式、观察者模式、状态模式、策略模式、模板方法模式和访问者模式

结构型模式 涉及如何组合类和对象以形成更大的结构，和类有关的结构型模式涉及如何合理使用继承机制，和对象有关的结构型模式涉及如何合理的使用对象组合机制。

适配器模式、组合模式、代理模式、享元模式、外观模式、桥接模式和装饰模式

Singleton 设计模式也称“单子模式”或“单例模式”。

采用调试方式讲解示例代码

Singleton 代码的特点包括以下几个方面：

1. 使用静态属性 `onlyone` 来引用一个“全局性”的 `Single` 实例。
2. 将构造方法设置为 `private` 的，这样在外界将不能再使用 `new` 关键字来创建该类的新实例。
3. 提供 `public static` 的方法 `getSingle()` 以使外界能够获取该类的实例，达到全局可见的效果。

在任何使用到 `Single` 类的 Java 程序中（这里指的是一次运行中），需要确保只有一个 `Single` 类的实例存在（如 Web 应用 `ServletContext` 全局上下文对象），则使用该模式。

5.4 关键字 final

在声明 Java 类、变量和方法时可以使用关键字 `final` 来修饰，以使其具有“终态”的特性：

1. `final` 标记的类不能被继承；
2. `final` 标记的方法不能被子类重写；
3. `final` 标记的变量（成员变量或局部变量）即成为常量，只能赋值一次；
4. `final` 标记的成员变量必须在声明的同时或在每个构造方法中显式赋值，然后才能使用；
5. `final` 不允许用于修饰构造方法、抽象类以及抽象方法。

关键字 `final` 应用举例如下：

```
1 public final class Test {  
2     public static int totalNumber = 5;  
3     public final int id;  
4     public Test() {  
5         id = ++totalNumber; // 赋值一次  
6     }  
7     public static void main(String[] args) {  
8         Test t = new Test();  
9         System.out.println(t.id);  
10        final int i = 10;  
11        final int j;
```

```
12     j = 20;  
13     j = 30; // 非法  
14 }  
15 }
```

实验设计

☒ 6 ☒ Java 内存模型与分配机制

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第二周（根据校历，本周有两次课）

参考教材: 本课程参考教材及资料如下：

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解 JVM 内存模型，掌握 JVM 内存构成
2. 理解 Java 程序的运行过程，学会通过调试模式观察内存的变化
3. 了解 Java 内存管理，认识垃圾回收
4. 建立编程时高效利用内存、避免内存溢出的理念

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

6.1 Java 内存模型

6.1.1 Java 虚拟机 (Java Virtual Machine, JVM)

Java 虚拟机的架构如图6.1所示。

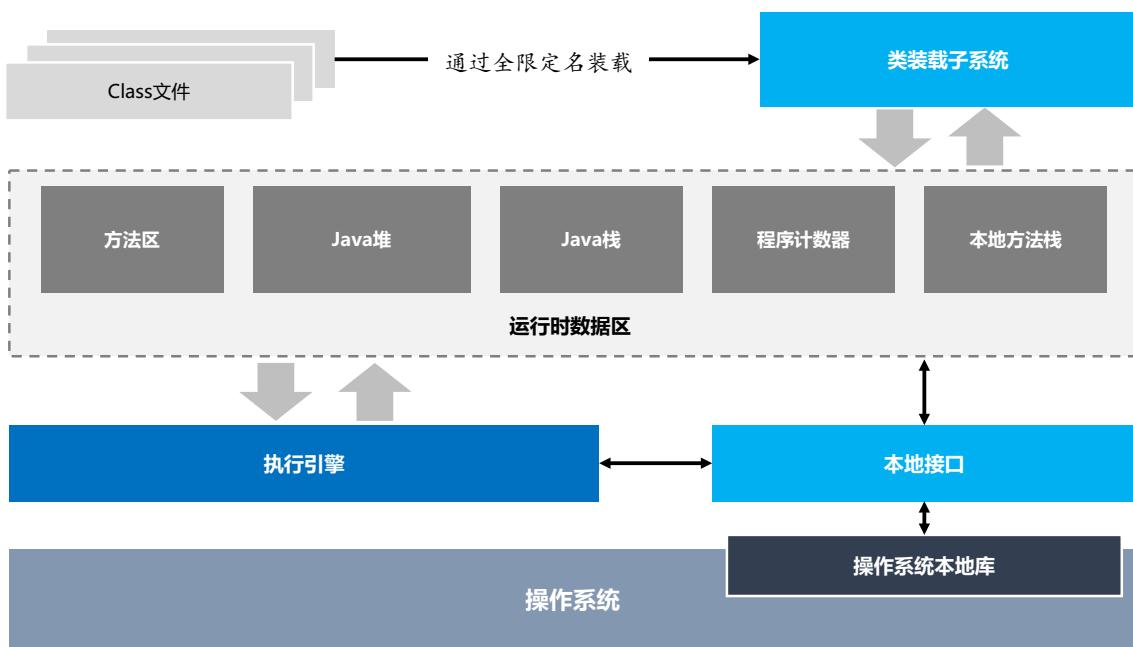


图 6.1 Java 虚拟机架构

- Java 程序运行在 JVM 上，JVM 是程序与操作系统之间的桥梁。
- JVM 实现了 Java 的平台无关性。
- JVM 是内存分配的前提。

6.1.2 JVM 内存模型

Java 程序运行过程会涉及的内存区域包括：

程序计数器 当前线程执行的字节码的行号指示器。

JVM内存模型

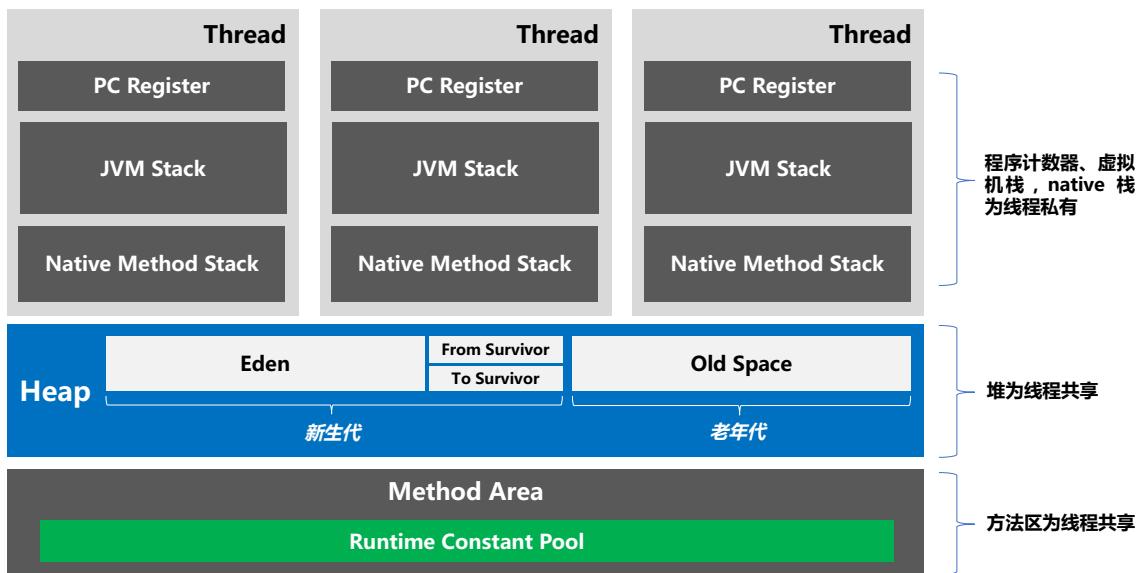


图 6.2 JVM 内存模型

栈 保存局部变量的值，包括：用来保存基本数据类型的值；保存类的实例，即堆区对象的引用（指针），也可以用来保存加载方法时的帧。（Stack）

堆 用来存放动态产生的数据，如 new 出来的对象和数组¹。（Heap）

常量池 JVM 为每个已加载的类型维护一个常量池，常量池就是这个类型用到的常量的一个有序集合。包括直接常量（基本类型、String）和对其他类型、方法、字段的符号引用。池中的数据和数组一样通过索引访问，常量池在 Java 程序的动态链接中起了核心作用。（Perm）

代码段 存放从硬盘上读取的源程序代码。（Perm）

数据段 存放 static 定义的静态成员。（Perm）

¹注意创建出来的对象只包含属于各自的成员变量，并不包括成员方法。因为同一个类的对象拥有各自的成员变量，存储在各自的堆内存中，但是他们共享该类的方法，并不是每创建一个对象就把成员方法复制一次。

6.2 Java 程序内存运行分析

6.2.1 预备知识和所用讲解程序

1. 一个 Java 文件，只要有 main 入口方法，即可认为这是一个 Java 程序，可以单独编译运行。
2. 无论是普通类型的变量还是引用类型的变量（俗称实例），都可以作为局部变量，他们都可以出现在栈中。
3. 普通类型的变量在栈中直接保存它所对应的值，而引用类型的变量保存的是一个指向堆区的指针。通过这个指针，就可以找到这个实例在堆区对应的对象。因此，**普通类型变量只在栈区占用一块内存，而引用类型变量要在栈区和堆区各占一块内存**。

示例代码：Test.java

```
1 public class Test {  
2     public static void main(String[] args) {  
3         Test test = new Test(); //1  
4         int data = 9; //2  
5         BirthDate d1 = new BirthDate(22, 12, 1982); //3  
6         BirthDate d2 = new BirthDate(10, 10, 1958); //4  
7         test.m1(data); //5  
8         test.m2(d1); //7  
9         test.m3(d2);  
10    }  
  
12    public void m1(int i) {  
13        i = 1234; //6  
14    }  
15    public void m2(BirthDate b) {  
16        b = new BirthDate(15, 6, 2010); //8  
17    }  
18    public void m3(BirthDate b) {  
19        b.setDay(18);  
20    }  
.....
```

21 }

6.2.2 程序调用过程

程序调用过程（一）

- JVM 自动寻找 main 方法，执行第一句代码，创建一个 Test 类的实例，在栈中分配一块内存，存放一个指向堆区对象的指针 110925。
- 创建一个 int 型的变量 data，由于是基本类型，直接在栈中存放 data 对应的值 9。
- 创建两个 BirthDate 类的实例 d1、d2，在栈中分别存放了对应的指针指向各自的对象。它们在实例化时调用了有参数的构造方法，因此对象中有自定义初始值。

程序调用过程（二）

- 调用 test 对象的 m1 方法，以 data 为参数。JVM 读取这段代码时，检测到 i 是局部变量，则会把 i 放在栈中，并且把 data 的值赋给 i。

程序调用过程（三）

- 把 1234 赋值给 i。

程序调用过程（四）

- m1 方法执行完毕，立即释放局部变量 i 所占用的栈空间。

程序调用过程（五）

- 调用 test 对象的 m2 方法，以实例 d1 为参数。JVM 检测到 m2 方法中的 b 参数为局部变量，立即加入到栈中，由于是引用类型的变量，所以 b 中保存的是 d1 中的指针，此时 b 和 d1 指向同一个堆中的对象。在 b 和 d1 之间传递是指针。

程序调用过程（六）

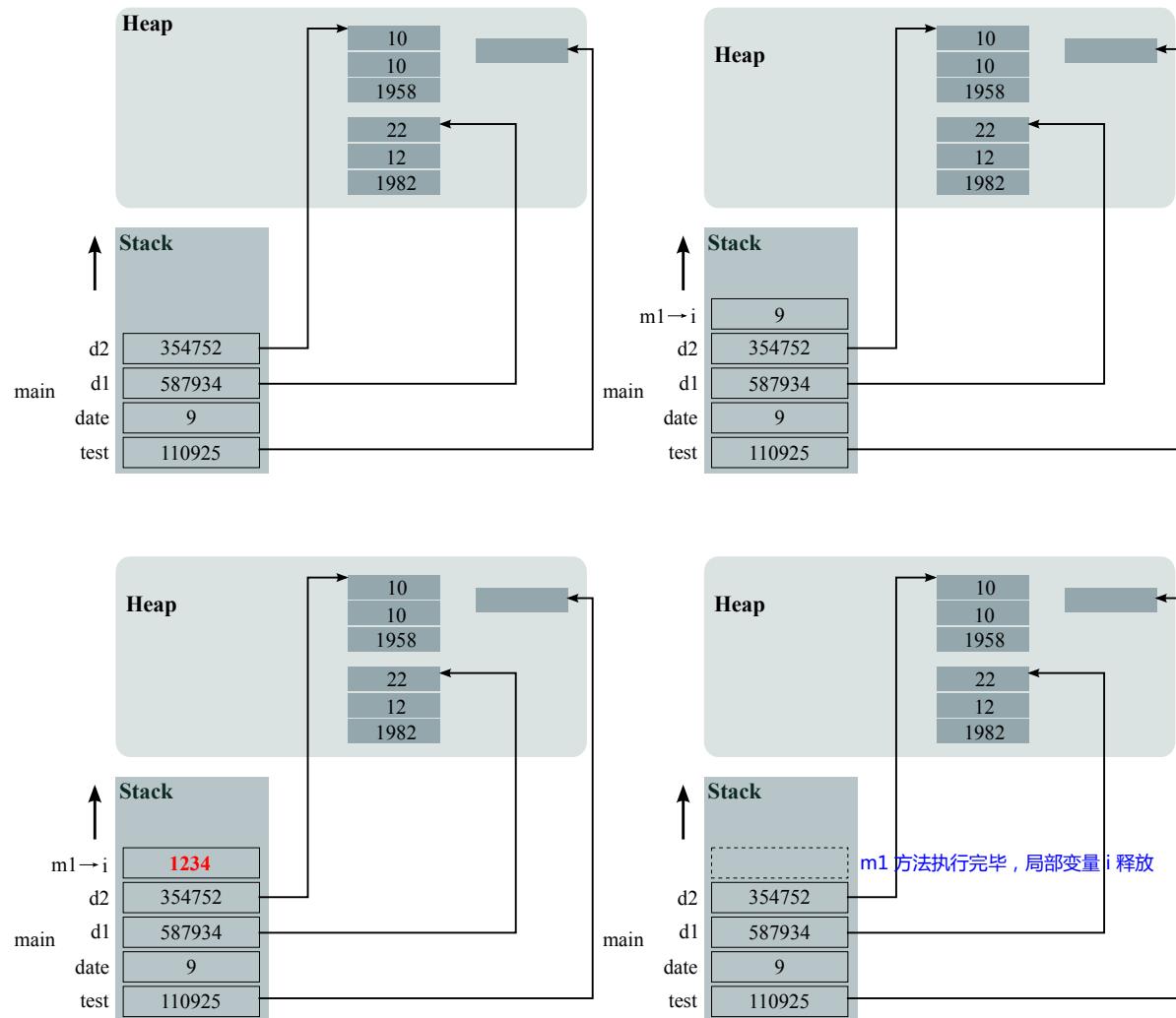
- m2 方法中又实例化了一个 BirthDate 对象，并且赋给 b。在内部执行过程是：在堆区 new 了一个对象，并且把该对象的指针保存在栈中 b 对应空间，此时实例 b 不再指向实例 d1 所指向的对象，但是实例 d1 所指向的对象并无变化，未对 d1 造成任何影响。

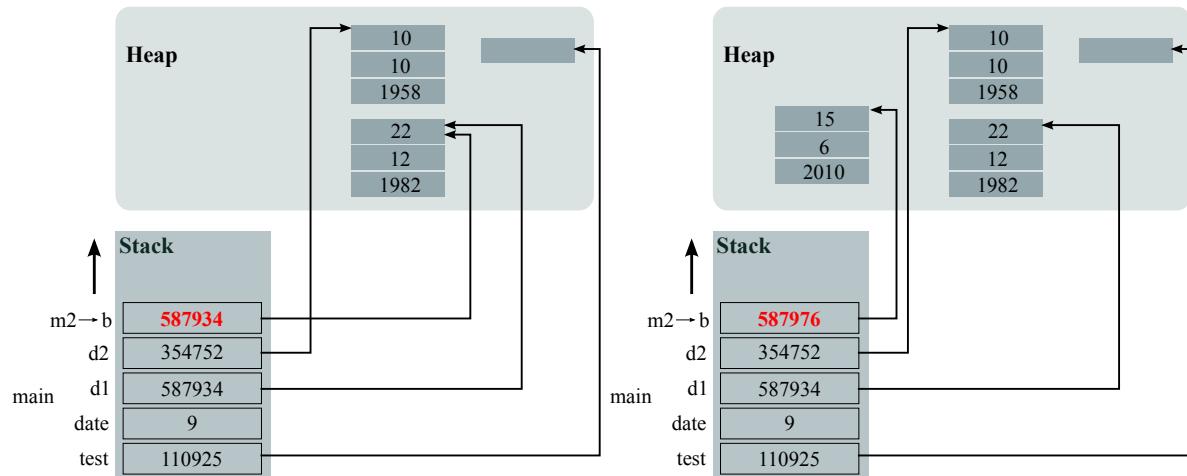
程序调用过程（七）

- m2 方法执行完毕，立即释放局部引用变量 b 所占的栈空间，注意只是释放了栈空间，堆空间要等待自动回收。

程序调用过程（八）

- 调用 test 实例的 m3 方法，以实例 d2 为参数。JVM 会在栈中为局部引用变量 b 分配空间，并且把 d2 中的指针存放在 b 中，此时 d2 和 b 指向同一个对象。再调用实例 b 的 setDay 方法，其实就是调用 d2 指向的对象的 setDay 方法。
- 调用实例 b 的 setDay 方法会影响 d2，因为二者指向的是同一个对象。
- m3 方法执行完毕，立即释放局部引用变量 b。





6.2.3 Java 程序运行内存分析小结

- 基本类型和引用类型，二者作为局部变量时都存放在栈中。
- 基本类型直接在栈中保存值，引用类型在栈中保存一个指向堆区的指针，真正的对象存放在堆中。
- 作为参数时基本类型就直接传值，引用类型传指针。

注意什么是对象

```
MyClass a = new MyClass();
```

此时 a 是指向对象的指针，而不能说 a 是对象。指针存储在栈中，对象存储在堆中，操作实例实际上是通过指针间接操作对象。多个指针可以指向同一个对象。

- **栈中的数据和堆中的数据销毁并不是同步的。** 方法一旦执行结束，栈中的局部变量立即销毁，但是堆中对象不一定销毁。因为可能有其他变量也指向了这个对象，直到栈中没有变量指向堆中的对象时，它才销毁；而且还不是马上销毁，要等垃圾回收扫描时才可以被销毁。
- **栈、堆、代码段、数据段等都是相对于应用程序而言的。** 每一个应用程序都对应唯一的一个 JVM 实例，每一个 JVM 实例都有自己的内存区域，互不影响，并且这些内存区域是该 JVM 实例所有线程共享的。

6.3 Java 内存管理建议

6.3.1 Java 垃圾回收机制

JVM 的垃圾回收机制（GC）决定对象是否是垃圾对象，并进行回收。垃圾回收机制的特点包括：

- 垃圾内存并不是用完了马上就被释放，所以会产生内存释放不及时的现象，从而降低内存的使用效率。而当程序庞大的时候，这种现象更为明显。
- 垃圾回收工作本身需要消耗资源，同样会产生内存浪费。

JVM 中的对象生命周期一般分为 7 个阶段：①创建阶段、②应用阶段、③不可视阶段、④不可到达阶段、⑤可收集阶段、⑥终结阶段、⑦释放阶段。

Java 需要内存管理，在 JVM 中运行的对象在整个生命周期中，进行人为的内存管理是必要的，主要原因体现在：

- 虽然 JVM 已经代替开发者完成了对内存的管理，但是硬件本身的资源是有限的。
- 如果 Java 的开发人员不注意内存的使用依然会造成较高的内存消耗，导致性能的降低。

6.3.2 JVM 内存溢出和参数调优

当遇到 OutOfMemoryError 时该如何做？

- 常见的 OOM（Out Of Memory）内存溢出异常，就是堆内存空间不足以存放新对象实例时导致。
- 永久区内存溢出相对少见，一般是由于需要加载海量的 Class 数据，超过了非堆内存的容量导致。通常出现在 Web 应用刚刚启动时。因此 Web 应用推荐使用预加载机制，方便在部署时就发现并解决该问题。
- 栈内存也会溢出，但是更加少见。

对内存溢出的处理方法不外乎这两种：① 调整 JVM 内存配置；② 优化代码。

创建阶段的 JVM 内存配置优化需要关注以下项：

堆内存优化 调整 JVM 启动参数 -Xms -Xmx -XX:newSize -XX:MaxNewSize，如调整初

始堆内存和最大对内存 -Xms256M -Xmx512M。或者调整初始 New Generation 的初始内存和最大内存 -XX:newSize=128M -XX:MaxNewSize=128M。

永久区内存优化 调整 PermSize 参数, 如 -XX:PermSize=256M -XX:MaxPermSize=512M。

栈内存优化 调整每个线程的栈内存容量, 如 -Xss2048K。

6.3.3 内存优化的小示例

减少无谓的对象引用创建

示例代码：Test 1

```
1 for( int i=0; i<10000; i++) {  
2     Object obj = new Object();  
3 }
```

示例代码：Test 2

```
1 Object obj = null;  
2 for( int i=0; i<10000; i++) {  
3     obj = new Object();  
4 }
```

内存性能分析

Test 2 比 Test 1 的性能要好。两段程序每次执行 for 循环都要创建一个 Object 的临时对象, JVM 的垃圾回收不会马上销毁但这些临时对象。相对于 Test 1, Test 2 则只在栈内存中保存一份对象的引用, 而不必创建大量新临时变量, 从而降低了内存的使用。

不要对同一对象初始化多次

```
1 public class A {  
2     private Hashtable table = new Hashtable();  
3     public A() {  
4         table = new Hashtable();  
5     }
```

```
6 }  
}
```

内存性能分析

上述代码 new 了两个 Hashtable，但是却只使用了一个，另外一个则没有被引用而被忽略掉，浪费了内存。并且由于进行了两次 new 操作，也影响了代码的执行速度。另外，不要提前创建对象，尽量在需要的时候创建对象。

6.3.4 对象其他生命周期阶段内存管理

应用 即该对象至少有一个引用在维护它。

不可视 即超出该变量的作用域。

因为 JVM GC 并不是马上进行回收，而是要判断对象是否被其他引用维护。所以，如果我们在使用完一个对象以后对其进行 obj = null 或者 obj.doSomething() 操作，将其标记为空，则帮助 JVM 及时发现这个垃圾对象。

不可到达 即在 JVM 中找不到对该对象的直接或者间接的引用。

可收集，终结，释放 垃圾回收器发现该对象不可到达， finalize 方法已经被执行，或者对象空间已被重用的时候。

Java 的 finalize() 方法

Java 所有类都继承自 Object 类，而 finalize() 是 Object 类的一个函数，该函数在 Java 中类似于 C++ 的析构函数（仅仅是类似）。一般来说可以通过重载 finalize() 的形式来释放类中对象。

```
1 public class A {  
2     public Object a;  
  
4     public A() {  
5         a = new Object();  
6     }  
  
8     protected void finalize() throws java.lang.Throwable {  
9         a = null; // 标记为空，释放对象  
10        super.finalize(); // 递归调用超类中的 finalize 方法  
}
```

```
11    }
12 }
```

什么时候 finalize() 被调用由 JVM 来决定。**尽量少用 finalize() 函数， finalize() 函数是 Java 提供给程序员一个释放对象或资源的机会。但它会加大 GC 的工作量，因此尽量少采用 finalize 方式回收资源。**

- 一般的，纯 Java 编写的 Class 不需要重写 finalize() 方法，因为 Object 已经实现了一个默认的，除非我们要实现特殊的功能。
- 用 Java 以外的代码编写的 Class(比如 JNI、C++ 的 new 方法分配的内存)，垃圾回收器并不能对这些部分进行正确的回收，这就需要我们覆盖默认的方法来实现对这部分内存的正确释放和回收。

实验设计

☒ 7 ☒ 高级类特性

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第四周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握抽象类和接口的概念、特性及定义方法
2. 理解抽象类和接口的异同和作用
3. 了解嵌套类的分类, 掌握嵌套类中静态嵌套类和匿名嵌套类的概念
4. 掌握匿名内部类的特征、继承和接口实现的用法
5. 掌握枚举类型的使用方法

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

7.1 抽象类

7.1.1 抽象类的概念

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的。如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类往往用来表征对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同但是本质上相同的具体概念的抽象。

7.1.2 定义抽象类

- 在定义 Java 方法时可以只给出方法头，而不必给出方法的实现细节，这样的方法被称为**抽象方法**。
- 抽象方法必须用关键字**abstract**修饰。
- 包含抽象方法的类必须声明为抽象类，用关键字**abstract**修饰。

示例代码：抽象类示例

```
1 public abstract class Animal { //定义为抽象类
2     private int age;
3
4     public void setAge(int age) {
5         this.age = age;
6     }
7
8     public int getAge(){
9         return age;
10    }
11
12    public abstract void eat(); //抽象方法
```

13 }

示例代码：抽象类继承

```
1 public class Person extends Animal {  
2     private String name;  
3     public void setName(String name) {  
4         this.name = name;  
5     }  
6     public String getName() {  
7         return name;  
8     }  
9     public void eat() { //重写方法  
10        System.out.println("洗手→烹饪→摆餐具→吃喝→收摊儿");  
11    }  
12 }
```

```
1 public class Bird extends Animal {  
2     public void fly(){  
3         System.out.println("我心飞翔!");  
4     }  
5     public void eat(){ //重写方法  
6         System.out.println("直接吞食!");  
7     }  
8 }
```

7.1.3 抽象类的特性与作用

抽象类的特性

- 子类必须实现其父类中的所有抽象方法，否则该子类也只能声明为抽象类。
- 抽象类不能被实例化。**问题** 抽象类能否有构造方法？

抽象类的作用

抽象类主要是通过继承由其子类发挥作用，包括两方面：

代码重用 子类可以重用抽象类中的属性和非抽象方法。

规划 子类中通过抽象方法的重写来实现父类规划的功能。

抽象类的其他特性

- 抽象类中可以不包含抽象方法。主要用于当一个类已经定义了多个更适用的子类时，为避免误用功能相对较弱的父类对象，干脆限制其实例化。
- 子类中可以不全部实现抽象父类中的抽象方法，但此时子类也只能声明为抽象类。
- 父类不是抽象类，但在子类中可以添加抽象方法，此情况下子类必须声明为抽象类。
- 多态性对于抽象类仍然适用，可以将引用类型变量（或方法的形参）声明为抽象类的类型。
- 抽象类中可以声明 static 属性和方法，只要访问控制权限允许，这些属性和方法可以通过 <类名>.<类成员> 的方法进行访问。

7.2 接口

7.2.1 接口 (interface) 的概念

在科技辞典中，“接口”被解释为“两个不同系统（或子程序）交接并通过它彼此作用的部分。在 Java 语言中，通过接口可以了解对象的交互界面，即明确对象提供的功能及其调用格式，而不需要了解其实现细节。

接口是抽象方法和常量值的定义的集合。从本质上讲，接口是一种特殊的**抽象类**，这种抽象类中**只包含常量定义和方法声明，而没有变量和方法的实现**。

7.2.2 定义接口

接口中定义的属性必须是 public static final 的，而接口中定义的方法则必须是 public abstract 的，因此这些关键字可以部分或全部省略。

示例代码：接口示例（未简化）

```
1 public interface Runner {  
2     public static final int id = 1;  
3     public abstract void start();  
4     public abstract void run();  
5     public abstract void stop();  
6 }
```

示例代码：与上述代码等价的标准定义

```
1 public interface Runner {  
2     int id = 1;  
3     void start();  
4     void run();  
5     void stop();  
6 }
```

7.2.3 接口的实现

和继承关系类似，类可以**实现**接口，且接口和实现类之间也存在多态性。

类继承和接口实现的语法格式如下：

```
1 [<modifier>] class <name> [extends <superclass>] [implements <interface> [,<interface>]*  
2     ] {  
3         <declarations>*  
4     }
```

示例代码：接口实现示例

```
1 public class Person implements Runner {  
2     public void start() {  
3         System.out.println("弯腰、蹬腿、咬牙、瞪眼、开跑");  
4     }  
5     public void run(){  
.....
```

```

6     System.out.println("摆动手臂、维持直线方向");
7 }
8 public void stop(){
9     System.out.println("减速直至停止、喝水");
10 }
11 }
```

通过接口可以指明多个类需要实现的方法，而这些类还可以根据需要继承各自的父类。或者说，**通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。**

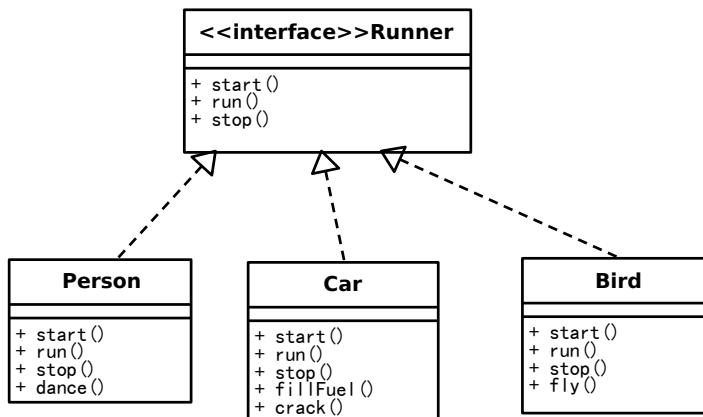


图 7.1 接口实现示例

类允许实现多重接口

课程配套代码 ➔ package sample.advance.interfacesample

7.2.4 接口间的继承

与接口的多重实现情况类似，由于不担心方法追溯调用上的不确定性，接口之间的继承允许“多重继承”的情况。

```

1 interface A {
2     public void ma();
3 }
4 interface B {
5     public int mb(int i);
6 }
```

```
6    }
7    interface C extends A,B { //接口的多重继承
8        public String mc();
9    }
10   class D implements C {
11       public void ma() {
12           System.out.println("Implements method ma()!");
13       }
14       public int mb(int i) {
15           return 2000 + i;
16       }
17       public String mc() {
18           return "Hello!";
19       }
20   }
```

上述代码中的 D 类缺省继承了 Object 类，直接实现了接口 C，间接实现了接口 A 和 B，由于多态性的机制，将来 D 类的对象可以当作 Object、C、A 或 B 等类型使用。

7.2.5 接口特性总结

- 通过接口可以实现不相关类的相同行为，而不需要考虑这些类之间的层次关系。
- 接口可以被多重实现。
- 接口可以继承其它的接口，并添加新的属性和抽象方法，接口间支持多重继承。

7.3 抽象类和接口剖析

7.3.1 语法层面的区别

概念差异——语法差异——用法差异——设计哲学

- 抽象类可以提供成员方法的实现细节，而接口中只能存在 public abstract 方法
- 抽象类中的成员变量可以为各种类型，而接口中的成员变量只能是 public static final 类型
- 抽象类可以有静态代码块和静态方法，接口中不能含有静态代码块以及静态方法

- 一个类只能继承一个抽象类，而一个类却可以实现多个接口

7.3.2 设计层面的区别

- 抽象类是对类的抽象（可以抽象但不宜实例化），而接口是对行为的抽象。
- 抽象类是对类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。
- 抽象类作为很多子类的父类，它是一种模板式设计。模板式设计：模板代表公共部分，公共部分需要改的则改动模板即可。
- 而接口是一种行为规范，它是一种辐射式设计。辐射式设计：接口进行了变更，则所有实现类都必须进行相应的改动。

7.3.3 怎样才是合理的设计？（门和警报的示例）

以门和警报设计作为示例，一般来说，门都有 `open()` 和 `close()` 这两个动作。通过抽象类和接口来定义这个抽象概念：

```
1 abstract class Door {  
2     public abstract void open();  
3     public abstract void close();  
4 }
```

```
1 interface Door {  
2     public abstract void open();  
3     public abstract void close();  
4 }
```

问题

如果现在我们需要门具有报警 `alarm()` 的功能该如何实现？

思路一

将这三个功能都放在抽象类里面，这样一来所有继承这个抽象类的子类都具备了报警功能，但是有的门并不一定需要具备报警功能。**不合理抽象**

思路二

将这三个功能都放在接口里面，但需要用到报警功能的类就需要实现这个接口中的 open() 和 close()，也许这个类根本就不具备 open() 和 close() 这两个功能，比如火灾报警器。**不合理规划**

Door 的 open()、close() 和 alarm() 根本就属于两个不同范畴内的行为：

- open() 和 close() 属于门本身固有的行为特性。
- alarm() 属于延伸的附加行为。

更为合理的思路

① 单独将报警设计为一个接口，包含 alarm() 行为；② Door 设计为单独的抽象类，包含 open() 和 close() 两种行为；③ 设计一个报警门继承 Door 类和实现 Alarm 接口。

课程配套代码 ➔ package sample.advance.door

7.4 嵌套类

7.4.1 什么是嵌套类

Java 语言支持类的嵌套定义，即允许将一个类定义在其他类的内部，其中内层的类被称为嵌套类。

嵌套类的分类

静态嵌套类 (Static Nested Class) 使用 static 修饰的嵌套类

内部类 (Inner Class) 非 static 的嵌套类

普通内部类 在类中的方法或语句块外部定义的非 static 类。

局部内部类 定义在方法或语句块中的类，也称局部类。

匿名内部类 定义在方法或语句块中，该类没有名字、只能在其所在之处使用一次。

7.4.2 静态嵌套类

静态嵌套类的特征

- 静态嵌套类不再依赖/引用外层类的特定对象，只是隐藏在另一个类中而已。
- 由于静态嵌套类的对象不依赖外层类的对象而独立存在，因而可以直接创建，进而也就无法在静态嵌套类中直接使用其外层类的非 static 成员。

课程配套代码 [▶ sample.advance.nestedclass.StaticNestedClassSample.java](#)

7.4.3 匿名内部类

匿名内部类是局部类的一种简化。

当我们只在一处使用到某个类型时，可以将之定义为局部类，进而如果我们只是创建并使用该类的一个实例的话，那么连类的名字都可以省略。

7.4.4 使用匿名内部类

示例代码：[Person.java](#)

```
1 public abstract class Person {  
2     private String name;  
3     private int age;  
4  
5     public Person() {}  
6  
7     public Person(String name, int age) {  
8         this.name = name;  
9         this.age = age;  
10    }  
11  
12    public String getInfo() {  
13        return "Name: " + name + "\t Age: " + age;  
14    }  
15  
16    public abstract void work();
```

17 }

示例代码：TestAnonymous.java

```
1 public class TestAnonymous {  
2     public static void main(String[] args) {  
3         Person sp = new Person() { // 匿名内部类  
4             public void work() {  
5                 System.out.println("个人信息: " + this.getInfo());  
6                 System.out.println("I am sailing.");  
7             }  
8         };  
9         sp.work();  
10    }  
11 }  
12 }
```

对上述代码的解释如下：

定义一个新的 Java 内部类，该类本身没有名字，但继承了指定的父类 Person，并在此匿名子类中重写了父类的 work() 方法，然后立即创建了一个该匿名子类的对象，再将其地址保存到引用变量 sp 中待用。

由于匿名类没有类名，而构造方法必须与类同名，所以匿名类不能显式的定义构造方法，但系统允许在创建匿名类对象时将参数传给父类构造方法（使用父类的构造方法）。

```
1 Person sp = new Person("Kevin", 30) {  
2     public void work() {  
3         System.out.println("个人信息: " + this.getInfo());  
4         System.out.println("I am sailing.");  
5     }  
6 };
```

匿名类除了可以继承现有父类之外，还可以实现接口，但不允许实现多个接口，且实现接口时就不能再继承父类了，反之亦然。

示例代码：Swimmer.java

```
1 public interface Swimmer {  
2     public abstract void swim();  
3 }
```

示例代码：[TestAnonymous2.java](#)

```
1 public class TestAnonymous2 {  
2     public static void main(String[] args) {  
3         TestAnonymous2 ta = new TestAnonymous2();  
4         ta.test(new Swimmer() { // 匿名类实现接口  
5             public void swim() {  
6                 System.out.println("I am swimming.");  
7             }  
8         });  
9  
10    public void test(Swimmer swimmer) {  
11        swimmer.swim();  
12    }  
13 }  
14 }
```

7.5 枚举类型

7.5.1 枚举类型的概念

Java SE 5.0 开始，引入了一种新的引用数据结构**枚举类型**。枚举类型均自动继承 `java.lang.Enum` 类，使用一组常量值来表示特定的数据集合，该集合中数据的数目确定（通常较少），且这些数据只能取预先定义的值。

```
1 public enum Week {  
2     MON, TUE, WED, THU, FRI, SAT, SUN  
3 }
```

无枚举类型前如何解决上述需求？

一般采用声明多个整型常量的做法实现枚举类的功能。

```
1 public class Week {  
2     public static final int MON = 1;  
3     public static final int TUE = 2;  
4     ...  
5 }
```

7.5.2 遍历枚举类型常量值

可以使用静态方法 values() 遍历枚举类型常量值。

示例代码：[ListEnum.java](#)

```
1 public class ListEnum {  
2     public static void main(String[] args) {  
3         Week[] days = Week.values();  
4         for(Week d: days) {  
5             System.out.println(d);  
6         }  
7     }  
8 }
```

7.5.3 组合使用枚举类型与 switch

课程配套代码 ➔ package sample.advance.enumclass

注意

1. case 字句必须省略其枚举类型的前缀，即只需要写成 case SUN:，而不允许写成 case Week.SUN:，否则编译出错。
2. 不必担心系统无法搞清这些常量名称的出处，因为 switch 后的小括号中的表达式已经指明本次要区分处理的是 Week 类型常量。

实验设计

☒ 8 ☒ 泛型

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第四周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 理解泛型的概念, 掌握其基本应用
 - 集合框架中的泛型
 - 泛型的向后兼容性
2. 掌握自定义泛型类和泛型方法
 - 理解类型参数
 - 理解差异性并能够定义自己的泛型类和泛型方法
 - 受限制的类型参数
3. 学会处理泛型类型, 包括使用通配符实现泛型容器遍历和操作

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

8.1 泛型概念

8.1.1 泛型的概念

泛型机制自 JDK 5.0 开始引入，其实质是将原本确定不变的数据类型参数化。作为对原有 Java 类型体系的扩充，使用泛型可以提高 Java 应用程序的类型安全、可维护性和可靠性。

传统的集合容器为了提供广泛的适用性，会将所有加入其中的元素当作 Object 类型来处理。基于此原因，在实际使用时，我们必须将从集合中取出的元素值再强制转换（造型）为所期望的类型。

无泛型机制的集合容器

```
1 Vector v = new Vector();
2 v.addElement(new Person("Tom", 18));
3 Person p = (Person) v.elementAt(0);
4 p.showInfo();
```

8.1.2 集合框架中的泛型

- 泛型允许编译器实施由开发者设定的附加类型约束，将类型检查从运行时挪到编译时进行，这样类型错误就可以在编译时暴露出来，而不是在运行时才发作（抛出 ClassCastException 运行异常）。
- 创建集合容器时规定其允许保存的元素类型，然后由编译器负责添加元素的类型合法性检查，在取用集合元素时则不必再进行造型处理。

在 Vector 中使用泛型

课程配套代码 ➔ sample.generics.VectorGenericsSample.java

在 Hashtable 中使用泛型

课程配套代码 ➔ sample.generics.HashtableGenericsSample.java

泛型的向后兼容性

- Java 语言中的泛型是维护向后兼容的，完全可以不采用泛型、而继续沿用过去的做法。
- 这些未加改造的旧式代码将无法使用泛型带来的便利和安全性。

未启用泛型机制的代码在高版本编译器中会输出如下形式的编译提示信息：

output

注: VectorGenericsSample.java 使用了未经检查或不安全的操作。

注: 有关详细信息, 请使用 -Xlint:unchecked 重新编译。

可以使用 SuppressWarnings 注解关闭编译提示或警告信息：

```
1 @SuppressWarnings({"unchecked"})
2 public class VectorGenericsSample {
3     ...
4 }
```

8.2 泛型类与泛型方法

类型参数

形如 `Vector<String>`, 其中, 尖括号括起来的部分被称为**类型参数**, 而这种由类型参数修饰的类型则被称为**泛型类**。

注意

应在声明泛型类变量和创建对象时均给出类型参数, 且两者应保持一致。

使用类型参数 *E* 进行泛型化处理的 `java.util.Vector` 类的定义代码摘要如下:

```
1 public class Vector<E> --- {
2     public void addElement(E obj) { --- }
3     public E elementAt(int index) { --- }
```

符号	意义
K	键，比如映射的键
V	值，比如 List 和 Set 的内容，或者 Map 中的值
E	元素，比如 Vector<E>
T	泛型

4 }

这里的 *E* 也称为“形式类型”参数。在实际使用该泛型类时，我们需要指定相应具体类型，即实际类型参数。

```
1 Vector<String> v = new Vector<String>();
```

编译器遇到 Vector<String> 类型变量时，即知道此 Vector 变量/对象的类型参数 *E* 已经被绑定为 String 类型，进而也就确定其 addElement() 方法的参数和 elementAt() 方法的返回值均为 String 类型。

8.2.1 定义泛型类

课程配套代码 ➔ package sample.generics.userdefined

代码示例中的泛型类 PersonG 可以在使用时通过类型参数 T 指定其属性 secrecy 的具体类型（以及该属性相应存/取方法的参数和返回值类型），进而提供了通用的信息存储能力。

形式类型参数的编程惯例

使用受限制的类型参数

泛型机制允许开发者对类型参数进行附加约束。

```
1 import java.util.Number;  
  
3 public class Point<T extends Number> {  
4     private T x;  
5     private T y;  
6     public Point() {}
```

```
7  public Point(T x, T y) {  
8      this.x = x;  
9      this.y = y;  
10     }  
11     ...  
12 }
```

类型参数不能为基本数据类型，而 `java.lang.Number` 是所有数值型封装类（如 `Integer`、`Float`、`Double` 等）的父类型，于是限制泛型类 `Point` 的类型参数必须为 `Number` 或其子类类型，并使用 `extends` 关键字来标明这种继承层次上限制。

8.2.2 定义泛型方法

与泛型类的情况类似，**方法也可以被泛型化，且无论其所属的类是否为泛型类。**

示例代码：泛型方法示例

```
1  public class Tool {  
2      public <T>T evaluate(T a, T b) {  
3          if (a.equals(b))  
4              return a;  
5          else  
6              return null;  
7      }  
8  }
```

- 上述代码中方法 `evaluate()` 声明中的“`<T>`”用于标明这是一个**泛型方法**。
- 类型 `T` 是可变的，不必显示的告知编译器 `T` 具体取何值，但出现多处（两个形参、一个返回值类型）的这些值必须都相同。

使用泛型方法，而不是定义泛型类的原则

- 不涉及到类中的其他方法时，则应将之定义为泛型方法，因为泛型方法的类型参数是局部性的，这样可以简化其所在类型的声明和处理开销。
- 要施加类型约束的方法为静态方法时，只能将之定义为泛型方法，因为静态方法不能使用其所在类的类型参数。

对泛型的理解

- 泛型类可以理解为具有广泛适用性、尚未最终定型的类型。
- Person<String> 和 Person<Double> 属于同一个类，但确是不同的类型。
- 同一个泛型类与不同的类型参数复合而成的类型间并不存在继承关系，即使是类型参数间存在继承关系时也是如此。
如：Vector<String> 不是 Vector<Object> 的子类。

8.3 处理泛型类型

8.3.1 遍历泛型 Vector 集合

遍历 Vector<String> 类型集合

```
1 public void overview(Vector<String> v) {  
2     for (String o: v) {  
3         String.out.println(o);  
4     }  
5 }
```

遍历其他类型参数的 Vector 集合元素该怎么办？

难道要定义 overview(Vector<Person> v)、overview(Vector<Integer> v) ... 显然过于繁琐，引用泛型机制后代码的通用性似乎不如从前？

8.3.2 泛型类型的处理方法

可能的处理方法（不要使用）

将遍历方法的形参定义为不带任何类型参数的原型类型 Vector，但这样会破坏已有的类型安全性。

```
1 public void overview(Vector v) {  
2     for (Object o: v) {  
3         String.out.println(o);  
4     }
```

```
5 }
```

使用通配符

为了解决类似泛型遍历的问题，Java 泛型机制中引入了通配符“?”。

```
1 public void overview(Vector<?> v) {  
2     for (Object o : v) {  
3         System.out.println(o);  
4     }  
5 }
```

类型通配符

使用类型通配符的好处包括：

1. `Vector<?>` 是任何泛型 `Vector` 的父类型，因此可以将 `Vector<String>`、`Vector<Integer>`、`Vector<Object>` 等作为实参传给 `overview(Vector<?> v)` 方法处理；
2. `Vector<?>` 类型的变量在调用方法时是受到限制的——凡是必须知道具体类型参数才能进行的操作均被禁止。

```
1 Vector<String> vs = new Vector<String>();  
2 vs.add("Tom");  
3 Vector<?> v = vs;  
4 v.add("Billy"); // 非法，编译器不知道具体类型参数  
5 Object e = v.elementAt(0); // 合法，允许检索元素，此时读取的元素均当作 Object 类型处理  
6 System.out.println(e);
```

上述限制不等同于将 `Vector<?>` 变为“只读”，在不需要编译器确定类型参数的情况下也是可以修改集合内容的，例如：

```
1 Vector<String> vs = new Vector<String>();  
2 vs.add("Tom");  
3 vs.add("Billy");  
4 Vector<?> v = vs;  
5 v.remove(new Integer(200)); // 形参为 Object 类型，运行不受影响  
6 v.clear(); // 不需要参数，运行不受影响
```


实验设计

☒ 9 ☒ 控制台应用程序设计

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第五周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 了解计算机人机交互发展
2. 掌握控制台程序设计开发中命令行参数、系统属性、标准输入输出的概念和相关 Java 操作
3. 掌握 Java 文件操作的常用方法
4. 了解注解类型
5. 学会 Jar 归档工具, 包括通过命令行或 IDE 进行 Java 程序归档的方法

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

9.1 从古老的计算机谈起

9.1.1 冯诺依曼机

我们的计算机是台遵守存储程序原理的冯诺依曼机器，基本组成包括运算器、控制器（合起来是 CPU）、存储器、输入设备、输出设备。你所面对的一切 SOC 也好，单板电脑也好，都是高度集成在一起的冯诺依曼机。

1950 年代的 IBM 1401

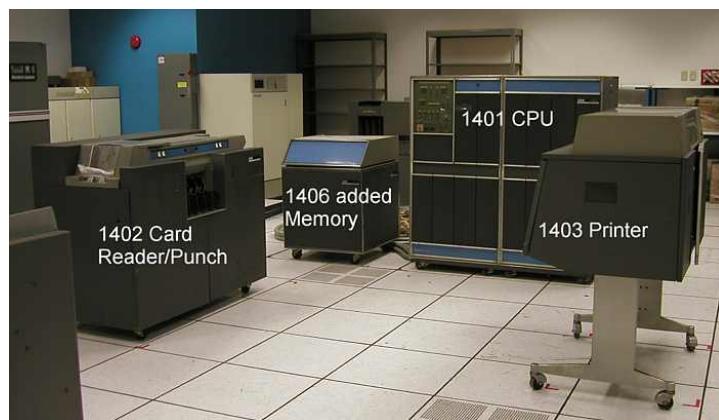


图 9.1 IBM 1401

2010 年代的树莓派开发板

9.1.2 人机交互

使用打孔卡片作为输入源，使用打印机作为输出设备

一摞打孔卡片，就是一个“文件”。它可以是一段程序，也可以是一段程序需要使用的数据。

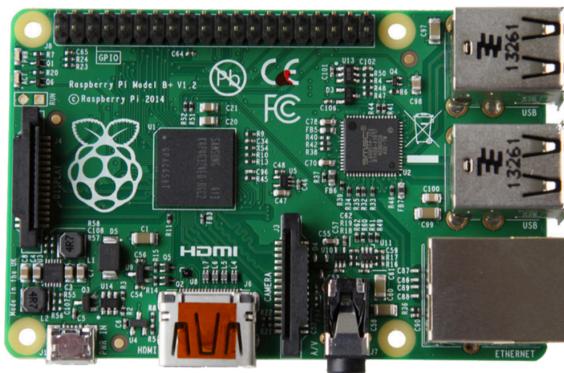


图 9.2 树莓派开发板

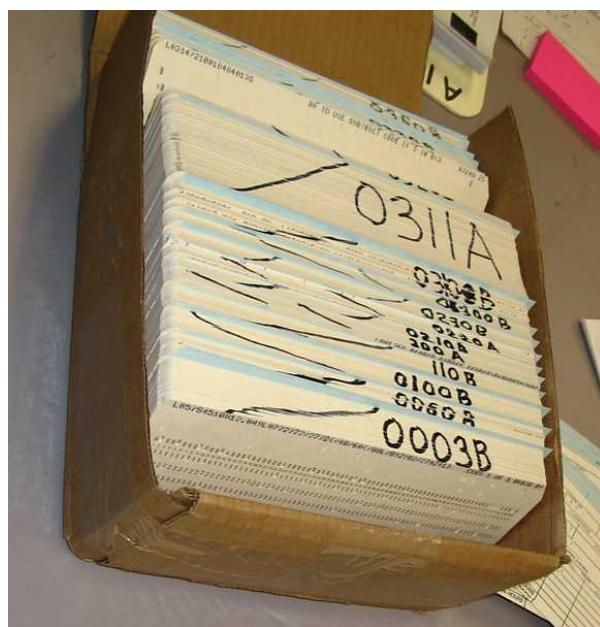


图 9.3 打孔卡片

BASIC 语言解释器

纸带在 70 年代还很流行，当年比尔盖茨的 BASIC 语言解释器，就是存在纸带上的，现在已经成文物了。

使用键盘作为输入设备，使用显示器作为输出设备

再厉害的科幻片导演，在飞船的人机交互界面表达上也未能超越同时代计算机的发展。

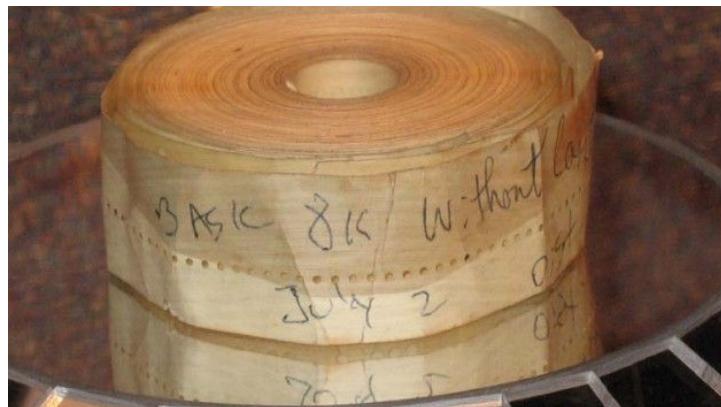


图 9.4 打孔卡片



图 9.5 分立的 Apple I



图 9.6 Apple I

9.2 命令行参数

9.2.1 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：

```
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()。



图 9.7 科幻电影中的计算机

- 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

Linux 下运行程序方法如下：

```
> java CommandLineArgsSample Lisa "Billy" "Mr Brown"
```

Windows 下运行程序方法如下：

```
C:\> java.exe CommandLineArgsSample Lisa "Billy" "Mr Brown" "a""b"
```

输出结果为：

output

```
Lisa  
Billy  
Mr Brown
```

9.2.2 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 }
```

9.3 系统属性

9.3.1 系统属性概述

- 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。

系统属性的用途主要包括：

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
```

9.3.2 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`

`Properties` 类的可用方法包括：

Enumeration propertyNames() 返回以 Enumeration 类型表示的所有可用系统属性的名称。

String getProperty(String key) 获得特定系统属性的属性值。

Object setProperty(string key, String value) 设置/添加单个系统属性信息。

void load(InputStream inStream)

void store(OutputStream out, String header) 实现属性信息的导入/导出操作。

9.4 标准输入/输出

9.4.1 标准输入/输出概述

控制台程序的交互方式中：

- 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）。

9.4.2 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）。

9.4.3 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;
4
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。
- 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。
- `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。
- 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。

9.5 文件操作

9.5.1 文件操作对象

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。我们可以使用以下构造方法创建 File 类对象：

- 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。

9.5.2 使用 File 类

课程配套代码 ➔ sample.commandline.FileOperationSample.java

9.5.3 File 类的主要方法

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()

9.5.4 文件 I/O 有关读写类

常见的文本文件 I/O 操作的类包括：

- **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.FileWriter** 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。

9.5.5 读取文件内容

示例代码：[ReadFileSample.java](#)

```
1 import java.io.*;  
  
3 public class ReadFileSample {  
4     public static void main (String[] args) {  
5         String fname = "test.txt";  
6         File f = new File(fname);  
  
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}
```

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回车时返回空字符串的情况不同。

9.5.6 输出内容到文件

示例代码：[WriteFileSample.java](#)

```
1 import java.io.*;  
2  
3 public class WriteFileSample {  
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如下：

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。

9.5.7 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。

9.5.8 使用 FileFilter 实现文件过滤

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

9.6 注解 (Annotation)

9.6.1 注解概述

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注解类的一个实例。注解类型采用 `@interface` 标记来声明。

JDK5.0 及后续版本定义的几种有用的注解类型包括：

- `public @interface Deprecated`

- public @interface Override
- public @interface SuppressWarnings

9.6.2 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     ...  
3     @Override  
4     public String toString() { // 重写方法  
5         return "Name: " + name;  
6     }  
7 }
```

`toString` 的原始定义如下：

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

9.6.3 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 }
```

9.6.4 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。该注解语法格式比较自由，下述均可。

```
1 @SuppressWarnings(value={"deprecation"})
2 @SuppressWarnings(value={"deprecation","unchecked"})
3 @SuppressWarnings("deprecation")
4 @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 }
```

代码编译时，则不会再输出先前的提示 API 过时信息。

9.7 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 文件的主要作用包括：

- 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 工具基本语法格式如下：

```
1 >jar {-ctxui} [vfm0Me] [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。

9.7.1 制作并使用自己的 jar 文件

示例代码：A.java

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

示例代码：TestJar.java

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

```
5     a.ma();  
6 }  
7 }
```

① 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：

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

output

② 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”
```

③ 编译 TestJar.java 源程序，并运行。

9.7.2 发布 Java 应用程序

我们一般使用 `java <应用程序名字>` 的方式运行 Java 程序。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。

示例代码：TestApp01.java

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

示例代码：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("Test App 02");
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 }
```

1. 编译程序
2. 程序归档发布

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

3. 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
```

9.7.3 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。清单文件格式如下：

<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。

☒ 10 ☒ 集合与映射

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第五周; 第六周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握列表 (List)、集 (Set)、映射 (Map) 的概念、层次关系及应用
2. 掌握迭代器 (iterator)、Enumeration 接口等容器操作常用 API

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

10.1 集合概念及分类

10.1.1 集合和数组

在编程中，常常需要集中存放多个数据。从传统意义上讲，数组是我们的一个很好的选择，前提是事先已经明确知道我们将要保存的对象的数量。一旦在数组初始化时指定了这个数组长度，这个数组长度就是不可变的。如果我们需要保存一个可以动态增长的数据（在编译时无法确定具体的数量），java 的集合类就是一个很好的设计方案了。

面向存放多个数据的需求，数组和集合类型具有以下用法差异：

- 数组用于存放指定长度的数据。
- 需要保存可以动态增长的数据（在编译时无法确定具体的数量），则需要用到 Java 的集合类。

10.1.2 集合类型

集合就是将若干用途、性质相同或相近的“数据”组合而成一个整体。集合类型分类如下：

集 Set 集合中不区分元素的顺序，不允许出现重复元素。例如应用于记录所有用户名的场合。

列表 List 集合区分元素的顺序，且允许包含重复元素。相当于数据结构中的线性表，具体表现为数组和向量、链表、栈、队列等。

映射 Map 中保存成对的“键→值”（Key-Value）信息，映射中不能包含重复的键，每个键最多只能映射一个值。

注意

Java 集合中只能保存引用类型的数据，实际上存放的是对象的引用而非对象本身。Java API 中的集合类型均定义在 `java.util` 包中。

10.1.3 对 Java 集合中只能保存引用类型的数据的说明

Java 集合只能存放引用类型数据，它们都是存放引用类型数据的容器，不能存放如 int、long、float、double 等基本类型的数据。

集合存储对象

Java 集合中实际存放的只是对象的引用，每个集合元素都是一个引用变量，实际内容都放在堆内存或者方法区里面，但是基本数据类型是在栈内存上分配空间的，栈上的数据随时就会被收回的。

基本类型数据如何解决呢？

可以通过包装类把基本类型转为对象类型，存放引用就可以解决这个问题。更方便的，由于有了自动拆箱和装箱功能，基本数据类型和其对应对象（包装类）之间的转换变得很方便，想把基本数据类型存入集合中，直接存就可以了，系统会自动将其装箱成封装类，然后加入到集合当中。

10.1.4 集合相关 API 的关系

10.2 Collection 和 Map 接口

10.2.1 Collection 接口

java.util.Collection 接口是描述 Set 和 List 集合类型（不包含 Map）的根接口，其中定义了有关集合操作的普遍性方法：

- boolean add(Object o) 向集合中添加一个元素，在子接口中此方法发生了分化，如 Set 接口中添加重复元素时会被拒绝（返回 false，而不是出错）；List 接口则会接受重复元素且返回 true。
- boolean remove(Object o) 从集合中移除指定的元素。
- int size() 返回集合中元素的数目。
- boolean isEmpty() 判断集合是否为空（即是否包含任何元素）。
- boolean contains(Object o) 判断集合中是否包含指定的元素。
- void clear() 移除当前集合中的所有元素。

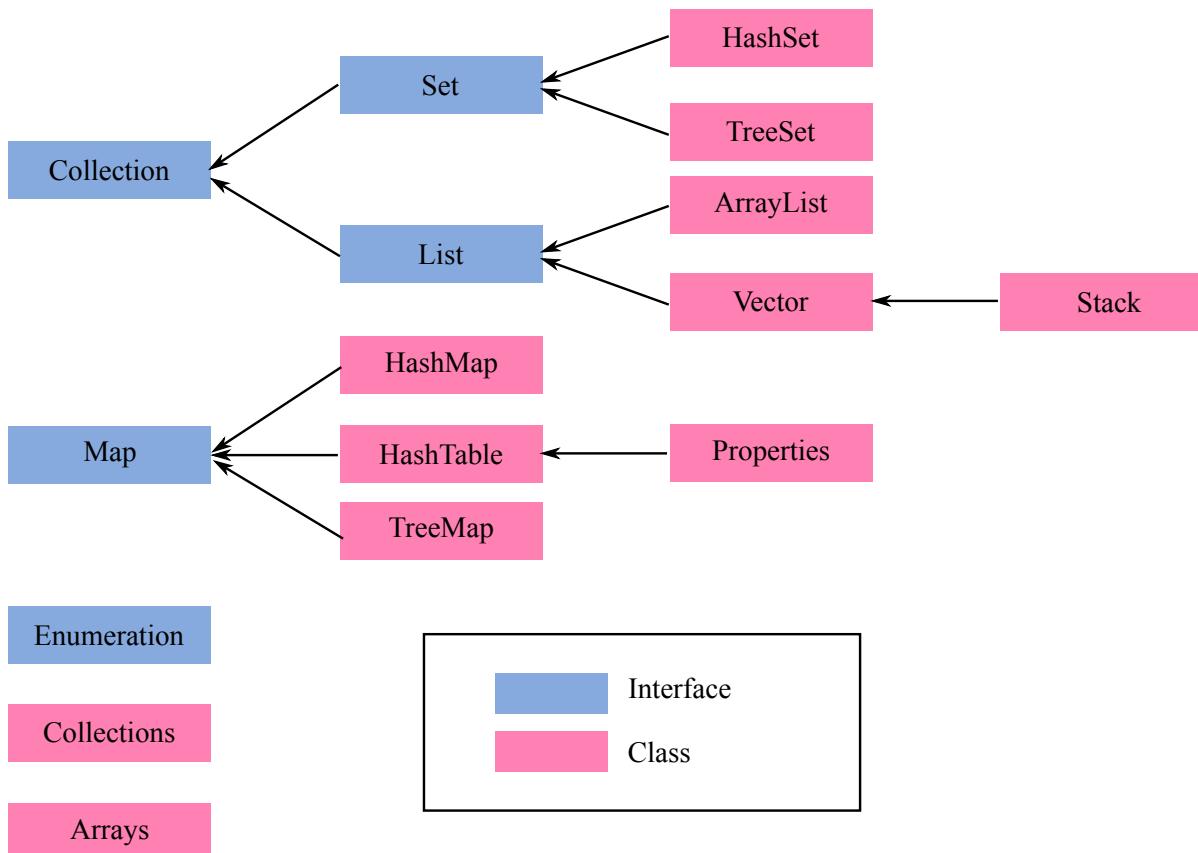


图 10.1 集合相关 API 的关系

- `Iterator iterator()` 返回在此集合的元素上进行迭代的迭代器。
- `Object[] toArray()` 返回包含当前集合中所有元素的数组。

10.2.2 Set 和 List 接口

`java.util.Set` 和 `java.util.List` 分别描述前述的集和列表结构，二者均为 `Collection` 的子接口。

`Set` 接口模拟了数学意义的集合。`List` 接口规定使用者可以对列表元素的插入位置进行精确控制，并添加了根据元素索引来访问元素等功能，接口中新添加了相应方法：

- `void add(int index, Object element)`
- `Object get(int index)`
- `Object set(int index, Object element)`
- `int indexOf(Object o)` 返回列表中首次出现指定元素的索引，如果列表不包含指定元素，则返回 -1。

- Object remove(int index)

10.2.3 Map 接口

java.util.Map 接口描述了映射结构，Map 结构允许以键集、值集合或键—值映射关系集的形式查看某个映射的内容。主要方法：

- Object put(Object key, Object value) 向当前映射中加入一组新的键—值对，并返回所加入元素的“值”，如果此映射中以前包含一个该键的映射关系，则用新值替换旧值。
- Object get(Object key) 返回此映射中映射到指定键的值，没有则返回 null。
- boolean isEmpty()
- void clear()
- int size()
- boolean containsKey(Object key) 如果映射中包含指定键的映射关系，则返回 true，否则返回 false。
- boolean containsValue(Object value)
- Set keySet() 返回此映射中包含的键的 set 视图，此 Set 受映射支持，所以对映射的改变可以在此 Set 中反映出来，反之亦然。
- Collection values() 返回此映射包含值的 Collection 视图，此 Collection 受映射支持，所以对映射的改变可以在此 Collection 中反映出来，反之亦然。

10.3 列表

10.3.1 ArrayList 类

java.util.ArrayList 类实现了 List 接口，用于表述长度可变的数组列表。ArrayList 列表允许元素取值为 null。除实现了 List 接口定义的所有功能外，还提供了一些方法来操作列表容量的大小，相关方法包括：

- public ArrayList() 构造方法：创建一个初始容量为 10 的空列表。
- public ArrayList(int initialCapacity)
- public void ensureCapacity(int minCapacity) 对容器进行扩容。

- public void trimToSize() 将此 ArrayList 实例的容量调整为列表的当前大小。

10.3.2 代码的局部性能优化 ensureCapacity

合理的使用 ArrayList ensureCapacity(int n) 方法可以对代码性能进行优化：

- 该方法可以对 ArrayList 底层的数组进行扩容。
- 显示的调用这个函数，如果参数大于低层数组长度的 1.5 倍，那么这个数组的容量就会被扩容到这个参数值，如果参数小于低层数组长度的 1.5 倍，那么这个容量就会被扩容到低层数组长度的 1.5 倍。
- 在适当的时机，好好利用这个函数，将会使我们写出来的程序性能得到很大的提升。

课程配套代码 ➔ sample.setlistmap.ArrayListEnSureCapacitySample.java

10.3.3 Vector 类

java.util.Vector 也实现了 List 接口，其描述的也是可变长度的对象数组。Vector 与 ArrayList 的差别主要包括：

Vector 是同步（线程安全）的，运行效率要低一些，主要用在多线程环境中，而 ArrayList 是不同步的，适合在单线程环境中使用。

常用方法（除实现 List 接口中定义的方法外）：

- public Vector()
- public Object elementAt(int index)
- public void addElement(Object obj)
- public void removeElementAt(int index)
- public void insertElementAt(E obj, int index)
- public boolean removeElement(Object obj)
- public void removeAllElements()
- public Object[] toArray()

10.3.4 什么是线程安全

线程安全 在多线程访问时采用加锁机制，当一个线程访问该类的某个数据时进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用，不会出现数据不一致或者数据污染。（Vector、HashTable 等）

线程不安全 不提供数据访问保护，有可能出现多个线程先后更改数据导致出现“脏数据”。（ArrayList、LinkedList、HashMap 等）

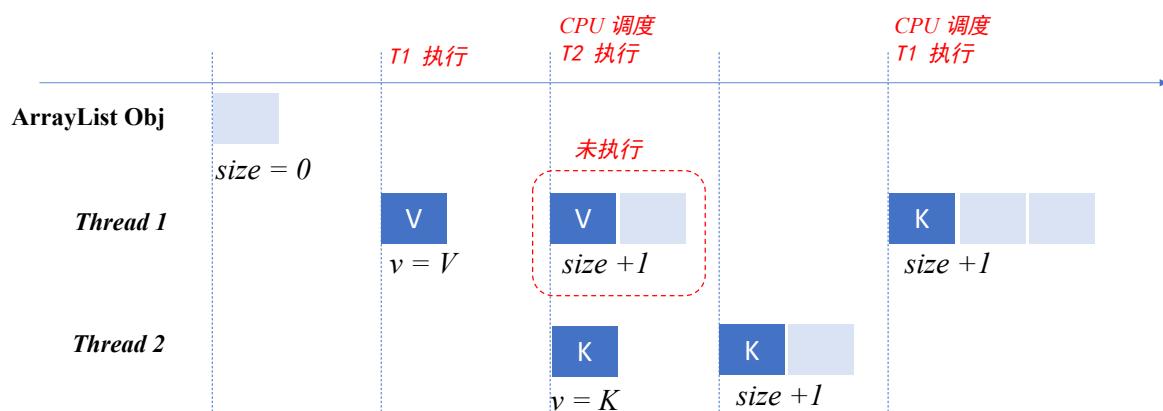


图 10.2 线程安全

比如一个 ArrayList 类，在添加一个元素的时候，它可能会有两步来完成：

1. 在 Items[Size] 的位置存放此元素；
2. 增大 Size 的值。

在单线程运行的情况下，如果 Size=0，添加一个元素后，此元素在位置 0，而且 Size=1；而如果是在多线程情况下，比如有两个线程，线程 A 先将元素存放在位置 0。但是此时 CPU 调度线程 A 暂停，线程 B 得到运行的机会。线程 B 也向此 ArrayList 添加元素，因为此时 Size 仍然等于 0，所以线程 B 也将元素存放在位置 0。接下来线程 A 和线程 B 都继续运行，都增加 Size 的值。当前，ArrayList 中元素实际上只有一个，存放在位置 0，而 Size 却等于 2，这就是**线程不安全**。

10.3.5 Stack

`java.util.Stack` 类继承了 `Vector` 类，对应数据结构中以“**后进先出**”（Last in First out, LIFO）方式存储和操作数据的**对象栈**。`Stack` 类提供了常规的栈操作方法：

- public Stack() 构造方法，创建一个空栈。
- public Object push(E item) 向栈中压入数据。
- public Object pop() 移除栈顶对象并作为此方法的返回值。
- public Object peek() 查看/返回栈顶对象，但不从栈中移除它。
- public boolean empty() 测试栈是否为空。
- public void clear() 清空栈。
- public int search(Object o) 返回对象在栈中的位置，以 1 为基数。

10.4 Iterator 接口

10.4.1 Iterator 接口概述

- 对于 ArrayList 可以使用 get() 方法访问其元素；
- 对于 Vector，还可以使用 elementAt() 方法访问其元素；
- 后续 Set 和 Map 集合也有各自不同的元素访问方式。

是否有一种统一的方式来遍历各种不同类型集合中的元素呢？

Java.util.Iterator 接口描述的是以统一方式对各种集合元素进行遍历/迭代的工具，也称为“**迭代器**”。迭代器允许在遍历过程中移除集合中的（当前遍历到的那个）元素。主要方法包括：

- boolean hasNext() 如果仍有元素可以迭代，则返回 true，否则返回 false。
- Object next() 返回迭代的下一个元素，重复调用此方法直到 hasNext() 方法返回 false。
- void remove() 将当前迭代到的元素从迭代器指向的集合中移除。

10.4.2 使用迭代器

我们一般不直接创建迭代器对象，而是通过调用集合对象的 iterator() 方法（该方法在 Collection 接口中定义）来获取。

示例代码：[TestIterator.java](#)

```
1 import java.util.ArrayList;
```

```
2 import java.util.Iterator;  
  
4 public class TestIterator {  
5     public static void main(String[] args) {  
6         ArrayList a = new ArrayList();  
7         a.add("China");  
8         a.add("USA");  
9         a.add("Korea");  
10        Iterator it = a.iterator();  
  
12        while(it.hasNext()) {  
13            String country = (String) it.next();  
14            System.out.println(country);  
15        }  
16    }  
17}
```

注意

迭代器相当于原始集合的一个“视图”，即一种表现形式，而不是复制其中所有元素得到的拷贝，因此在迭代器上的操作将影响到原来的集合。

10.5 集

10.5.1 HashSet 类

`java.util.HashSet` 类实现了 `java.util.Set` 接口，描述典型的 Set 集合结构。

- `HashSet` 中不允许出现重复元素，不保证集合中元素的顺序。
- `HashSet` 中允许包含值为 `null` 的元素，但最多只能有一个 `null` 元素。

10.5.2 TreeSet 类

`java.util.TreeSet` 类也实现了 `java.util.Set`，它描述的是 Set 的一种变体——可以实现排序功能的集合。

- 在将对象元素添加到 `TreeSet` 集中时会自动按照某种比较规则将其插入到有序的对象序列中，以保证 `TreeSet` 集合元素组成对象序列时刻按照“升序”排列（例

如按照字典顺序排列);

- 对于用户自定义的类型的数据可以自行定义其所需的排序规则(使用 Comparable 接口)。

10.5.3 Comparable 接口

java.lang.Comparable 接口中定义的 compareTo() 方法用于提供对其实现类的对象进行整体排序所需的比较逻辑, 所为的排序可以理解为按照某种标准来比较对象的大小以确定其次序。

- 实现类基于 compareTo() 方法的排序被称为**自然排序**。
- compareTo() 方法被称为它的**自然比较方法**, 具体的排序原则可由实现类根据需要而定。

方法格式如下:

```
1 int compareTo(Object o) {  
2 }
```

使用 Comparable 接口实现自然排序

[示例代码: Person.java](#)

```
1 public class Person implements java.lang.Comparable {  
2     private final int id;  
3     ...  
4  
5     public Person(int id, String name, int age) {  
6         this.id = id;  
7         ...  
8     }  
9     ...  
10    @Override  
11    public int compareTo(Object o) {  
12        Person p = (Person) o;  
13        return this.id - p.id;
```

```
14    }
15
16    @Override
17    public boolean equals(Object o) {
18        boolean flag = false;
19        if (o instanceof Person) {
20            if (this.id == ((Person)o).id) {
21                flag = true;
22            }
23        }
24        return flag;
25    }
}
```

示例代码： TestComparable.java

```
1 import java.util.TreeSet;
2 import java.util.Iterator;
3
4 public class TestComparable {
5     public static void main(String[] args) {
6         TreeSet ts = new TreeSet();
7         ts.add(new Person(1003, "Bob", 15));
8         ts.add(new Person(1008, "Alice", 25));
9         ts.add(new Person(1001, "Kevin", 30));
10    }
11    Iterator it = ts.iterator();
12    while (it.hasNext()) {
13        Person employee = (Person) it.next();
14        System.out.println(employee);
15    }
16}
```

output

Id: 1001 Name: Kevin Age:30

Id: 1003 Name: Bob Age:15

Id: 1008 Name: Alice Age:25

对上述程序的几点说明：

1. 用户在重写 `compareTo()` 方法以定制比较逻辑时，需要确保其与等价性判断方法 `equals()` 保持一致，即确保条件 “`(x.compareTo(y) == 0) == (x.equals(y))`” 永远成立，否则逻辑上不合理。所以上例同时重写了 `equals()` 方法。
2. 为保证能够实现元素的排序功能，`TreeSet` 集合要求向其加入的对象元素必须是 `Comparable` 接口的实现类的实例，否者程序运行时会抛出**造型异常** (`java.lang.ClassCastException`)。
3. `Comparable` 接口并不专用于集合框架。

10.6 映射

10.6.1 HashMap 类

`java.util.HashMap` 类实现了 `java.util.Map` 接口，该类基于**哈希表**实现了前述的映射集合结构。

- `HashMap` 结构不保证其中元素（映射信息）的先后顺序，且允许使用 `null` “值” 和 `null` “键”。
- 当集合中不存在当前检索的“键”所对应的映射值时，`HashMap` 的 `get()` 方法会返回空值 `null`，而不会运行出错。
- 影响 `HashMap` 性能的两个参数：初始容量（Initial Capacity）和加载因子（Load Factor）。

10.6.2 HashTable 类

`java.util.Hashtable` 与 `HashMap` 作用基本相同，也实现了 `Map` 接口，采用哈希表的方式将“键”映射到相应的“值”。

`Hashtable` 与 `HashMap` 的差别主要包含以下方面：

- `Hashtable` 中元素的“键”和“值”均不允许为 `null`，而 `HashMap` 则允许。
- `Hashtable` 是同步的，即线程安全的，效率相对要低一些，适合在多线程环境下使用；而 `HashMap` 是不同步的，效率相对高一些，提倡在单线程环境中使用。
- 除此之外，`Hashtable` 与 `HashMap` 的用法格式完全相同。

10.6.3 TreeMap 类

java.util.TreeMap 类实现了将 Map 映射中的元素按照“键”进行升序排列的功能，其排序规则可以是默认的按照“键”的自然顺序排列，也可以使用指定的其他排序规则。

向 TreeMap 映射中添加的元素“键”所属的类必须实现 Comparable 接口。

```
1 public MyKey implements Comparable {
2     private final int id;
3     ...
4     public MyKey(int id) {
5         this.id = id;
6     }
7     ...
8     @Override
9     public int compareTo(Object o) {
10        return this.id - ((MyKey) o).id;
11    }
12    @Override
13    public boolean equals(Object o) {
14        return (o instanceof MyKey) && (this.id == ((MyKey) o).id);
15    }
16    @Override
17    public int hashCode() {
18        return new Integer(id).hashCode();
19    }
20 }
```

对上述程序的说明

MyKey 类重写 equals() 方法的同时也重写了 hashCode() 方法，这是一种规范的做法，目的是为了维护 hashCode() 方法的常规协定，该协定要求相等对象必须具有相等的哈希码，即当两个对象使用 equals() 方法比较结果为等价时，它们各自调用 hashCode() 方法也应该返回相同的结果。

10.7 其他相关 API

10.7.1 Enumeration 接口

java.util Enumeration 接口作用与 Iterator 接口类似，但只提供了遍历 Vector 和 Hashtable（及子类 Properties）类型集合元素的功能，且不支持集合元素的移除操作。

```
1 import java.util.*;  
  
3 public class TestEnumeration {  
4     public static void main(String[] args) {  
5         Vector v = new Vector();  
6         v.addElement("Lisa");  
7         v.addElement("Billy");  
8         v.addElement("Brown");  
  
10     Enumeration e = v.elements();  
  
12     while(e.hasMoreElements()) {  
13         String value = (String) e.nextElement();  
14         System.out.println(value);  
15     }  
16 }  
17 }
```

10.7.2 Collections 类

java.util.Collections 类定义了多种集合操作方法，能够实现了对集合元素的**排序、取极值、批量拷贝、集合结构转换、循环移位以及匹配性检查**等功能。Collections 类的主要方法包括：

- public static void sort(List list)
- public static void reverse(List list)
- public static void shuffle(List list)
- public static void copy(List dest, List src)
- public static ArrayList list(Enumeration e)

- public static int frequency(Collection c, Object o)
- public static T max(Collection coll)
- public static T min(Collection coll)
- public static void rotate(List list, int distance)

10.7.3 Arrays 类

java.util.Arrays 类定义了多种数组操作方法，实现了对数组元素的排序、填充、转换为列表或字符串形式、增强的检索和深度比较等功能。Arrays 类的主要方法包括：

- public static List asList(Object... a)
- public static void sort(<类型>[] a)
- public static int binarySearch(int[] a, int key)
- public static String toString(int[] a)

说明

Arrays 类在前面数组部分有阐述。

☒ 11 ☒ GUI 编程

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第六周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 了解用 Java 开发桌面软件图形用户界面的常用工具集
2. 掌握 AWT 的常用组件和视觉控制
3. 深入理解 GUI 事件处理机制
4. 了解 Applet, 特别是其历史渊源, 了解与 Applet 类似的技术
5. 理解 Swing 和 AWT 的关系, 学习使用 Swing 的典型组件构建较复杂的图形界面程序

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

使用 Java 语言，我们可以开发平台无关的图形界面程序。Java 语言常用的 GUI 库包括 AWT、Swing 和 JavaFX。另外，我们也可以采用多语言混合开发的模式。本节主要介绍使用 AWT 及 Swing 的进行 GUI 开发的相关技术细节。

11.1 GUI 组件及布局

GUI (Graphical User Interface) 即图形用户界面，Java 主要分为 AWT 和 Swing 两大系列 GUI API。其中，AWT (Abstract Window Toolkit) 即为抽象窗口工具集，是 Java 包含的最基础的图形界面库，AWT 相关软件包主要包括：

java.awt 包 提供基本 GUI 组件、视觉控制和绘图工具 API。

java.awt.event 包 提供 Java GUI 事件处理 API。

11.1.1 组件和容器

组件 (Component) 是图形用户界面的基本组成元素，凡是能够以图形化方式显示在屏幕上并能够与用户进行交互的对象均为组件，如菜单、按钮、标签、文本框、滚动条等。组件包含以下特征：

- 组件不能独立地显示出来，必须将组件放在一定的容器中才可以显示出来。
- JDK 的 java.awt 包中定义了多种 GUI 组件类，如 Menu、Button、Label、TextField 等。
- 抽象类 java.awt.Component 是除菜单相关组件之外所有 Java AWT 组件类的根父类，该类规定了 GUI 组件的基本特性，如尺寸、位置和颜色效果等，并实现了作为一个 GUI 部件所应具备的基本功能。
- java.awt.MenuComponent 是所有与菜单相关的组件的父类。

容器 (Container) 实际上是 Component 的子类，容器类对象本身也是一个组件，具有组件的所有性质，另外还具有容纳其它组件和容器的功能。容器类对象可使用方法 add() 添加组件。

AWT 包含的两种主要的容器类型如下：

java.awt.Window 可自由停泊的顶级窗口。

java.awt.Panel 可作为容器容纳其他组件，但不能独立存在，必须被添加到其他容器（如 Frame）中。

图11.1展示了 AWT 主要组件和容器的接口与实现类之间的层次关系，需要深入理解并掌握。

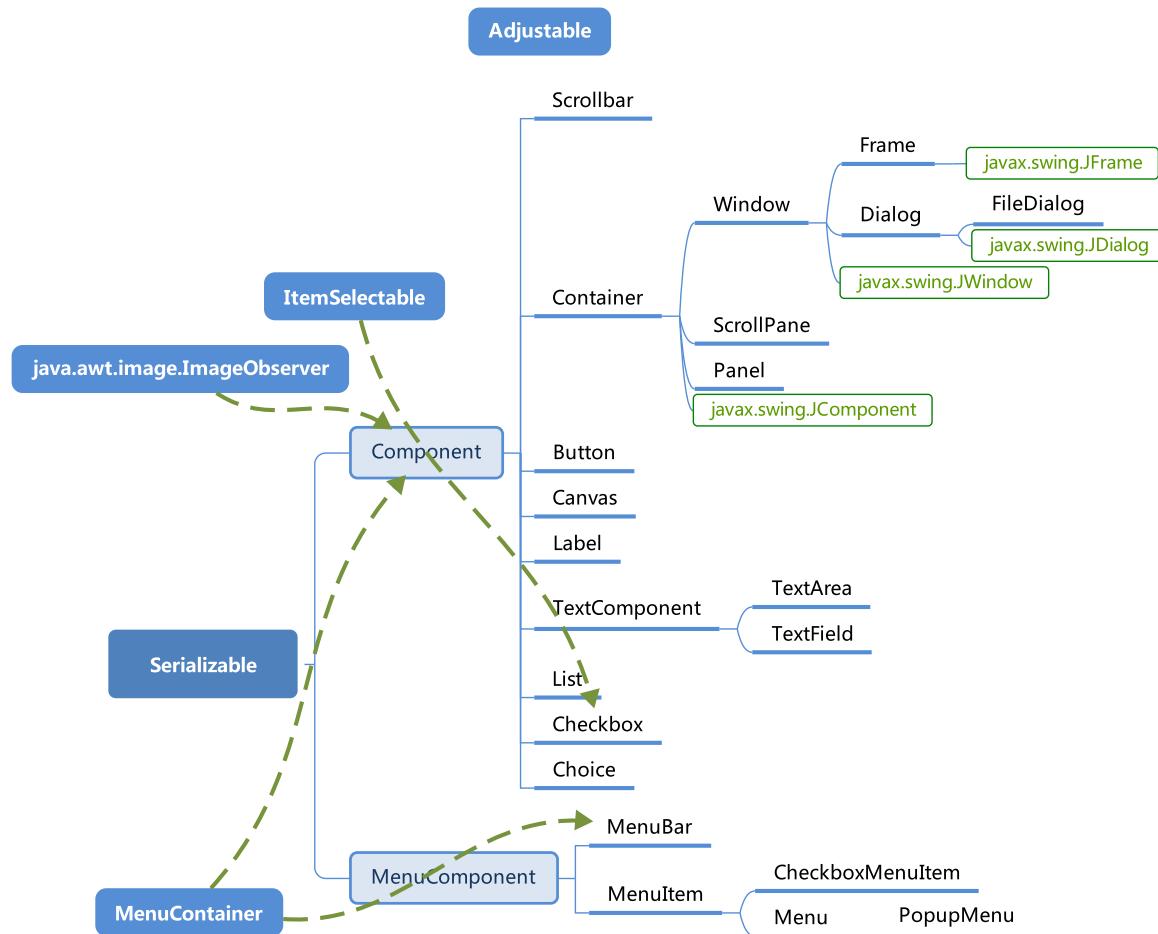


图 11.1 AWT 组件和容器层次架构

11.1.2 常用的组件和容器

AWT 常用的组件和容器如下表所示。

表 11.1 AWT 常用的组件和容器

组件类型	父类	说明
Button	Component	可接收点击操作的矩形 GUI 组件
Canvas	Component	用于绘图的面板
Checkbox	Component	复选框组件
CheckboxMenuItem	MenuItem	复选框菜单项组件
Choice	Component	下拉式列表框，内容不可改变
Component	Object	抽象的组件类
Container	Component	抽象的容器类
Dialog	Window	对话框组件，顶级窗口、带标题栏
FileDialog	Dialog	用于选择文件的平台相关对话框
Frame	Window	基本的 Java GUI 窗口组件
Label	Component	标签类
List	Component	包含内容可变的条目的列表框组件
MenuBar	MenuComponent	菜单条组件
Menu	MenuItem	菜单组件
MenuItem	MenuComponent	菜单项组件
Panel	Container	基本容器类，不能单独停泊
PopupMenu	Menu	弹出式菜单组件
Scrollbar	Component	滚动条组件
ScrollPane	Container	带水平及垂直滚动条的容器组件
TextComponent	Component	TextField 和 TextArea 的基本功能
TextField	TextComponent	单行文本框
TextArea	TextComponent	多行文本域
Window	Container	抽象的 GUI 窗口类，无布局管理器

11.1.3 Frame 类

Frame 类的显示效果是一个标准的图形窗口，它封装了 GUI 组件的各种属性信息，如尺寸、可见性等。

1. Frame 对象的显示效果是一个可自由停泊的顶级“窗口”，带有标题和尺寸重置角标。
2. Frame 默认初始化为不可见的，可以调用 Frame 对象的 setVisible(true) 方法使之变为可见。
3. 作为容器 Frame 还可使用 add() 方法包含其他组件。

课程配套代码 ▶ sample.awt.FrameSample.java

11.1.4 组件定位

Java 组件在容器中的定位由**布局管理器**决定。如要人工控制组件在容器中的定位，可取消布局管理器，然后使用 Component 类的成员方法 setLocation()、setSize()、setBounds() 设置。组件布局在桌面窗口的定位参考图11.2。

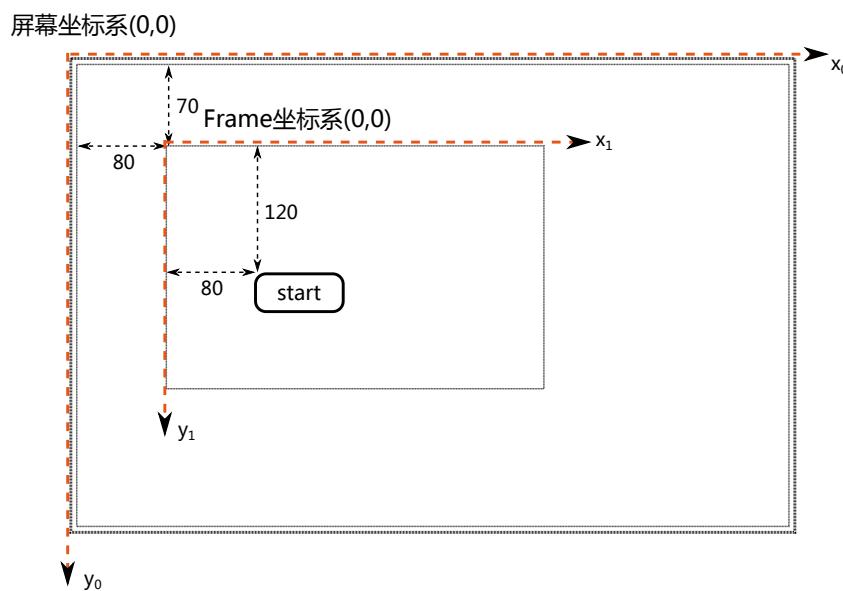


图 11.2 组件定位参照系

11.1.5 Panel 类

- Panel 提供容纳组件的空间。
- Panel 不能独立存在，必须被添加到其他容器中。
- 可以采用和所在容器不同的布局管理器。

课程配套代码 ➔ sample.awt.FrameWithPanelSample.java

11.1.6 布局管理器

容器对其中所包含组件的排列方式，包括组件的位置和大小设定，被称为容器的布局（Layout）。

为了使图形用户界面具有良好的平台无关性，Java 语言提供了布局管理器来管理容器的布局，而不建议直接设置组件在容器中的位置和尺寸。布局管理器类层次如下：

LayoutManager---FlowLayout



LayoutManager2---BorderLayout



说明

LayoutManager2 是 LayoutManager 的子接口。

每个容器都有一个布局管理器，当容器需要对某个组件进行定位或判断其大小尺寸时，就会调用其对应的布局管理器。可以在容器创建后调用其 `setLayout()` 方法设置其布局管理器类型。Container 类型容器没有默认的布局管理器，即其 `layoutMgr` 属性为 `null`，在其子类中才进行分化。

常用容器的默认的布局管理器如图所示。

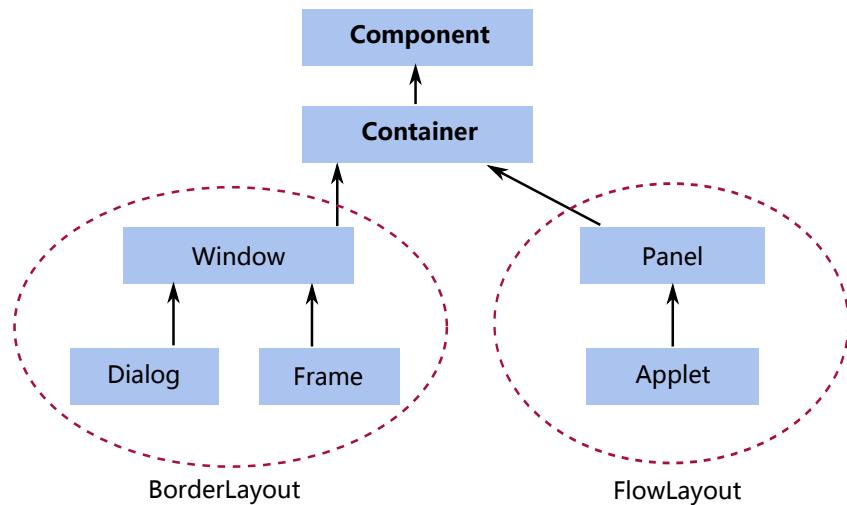


图 11.3 容器默认布局管理器

FlowLayout

流式布局是 Panel (及其子类) 类型容器的默认布局管理器类型。其布局效果如下：

- 组件在容器中按照加入次序逐行定位，行内从左到右，一行排满后换行。
- 不改变组件尺寸，即按照组件原始大小进行显示。
- 组件间的对齐方式默认为居中对齐，也可在构造方法中设置不同的组件间距、行距及对齐方式。

流式布局管理器提供以下构造方法：

- **public FlowLayout()**

组件对齐方式默认为居中对齐，组件的水平和垂直间距默认为 5 个像素。

- **public FlowLayout(int align)**

显式设定组件的对其方式，组件的水平和垂直间距默认为 5 个像素。

FlowLayout.LEFT

FlowLayout.RIGHT

FlowLayout.CENTER

- **public FlowLayout(int align, int hgap, int vgap)**

显式设定组件的对其方式、组件的水平和垂直间距。

课程配套代码 ➔ sample.awt.layout.FlowLayoutSample.java

BorderLayout

边界布局是 Window 及其子类（包括 Frame、Dialog）容器的默认布局管理器。其布局效果如下：

- BorderLayout 将整个容器的布局划分成东、西、南、北、中五个区域，组件只能被添加到指定的区域。如不指定组件的加入部位，则默认加入到 Center 区域。
- 每个区域只能加入一个组件，如加入多个，则先前加入的组件会被遗弃。
- 组件尺寸被强行控制，即与其所在区域的尺寸相同。

边界布局管理器提供如下构造方法：

- **public BorderLayout()**

构造一个 BorderLayout 布局管理器，其所包含的组件/区域间距为 0。

- **public BorderLayout(int hgap, int vgap)**

构造一个 BorderLayout 布局管理器，根据参数的组件/区域间距。

◆ BorderLayout 型布局容器尺寸缩放原则

- 北、南两个区域只能在水平方向缩放（宽度可调整）。
- 东、西两个区域只能在垂直方向缩放（高度可调整）。
- 中部可在两个方向上缩放。

GridLayout

网格布局的效果如下：

- 将容器区域划分成规则的矩形网格，每个单元格区域大小相等，组件被添加到每个单元格中，按组件加入顺序先从左到右填满一行后换行，行间从上到下。
- GridLayout 型布局的组件大小也被布局管理器强行控制，与单元格同等大小，当容器尺寸发生改变时，组件的相对位置保持不变，但大小自动调整。

网格布局管理器提供如下构造方法：

- **public GridLayout()** 所有组件于一行中，各占一列。

- **public GridLayout(int rows, int cols)**

通过参数指定布局的行数和列数。

- **public public GridLayout(int rows, int cols, int hgap, int vgap)**

通过参数指定布局的行数、列数，以及组件间水平间距和垂直间距。

课程配套代码 ➔ sample.awt.layout.GridLayoutSample.java

CardLayout

卡片布局的效果如下：

- 将多个组件在同一容器区域内交替显示，相当于多张卡片摞在一起，只有最上面的卡片是可见的。
- 可以按名称显示某一张卡片，或按先后顺序依次显示，也可以直接定位到第一张或最后一张卡片。

卡片布局管理器提供以下主要方法：

- **public void first(Container parent)**
- **public void last(Container parent)**
- **public void previous(Container parent)**
- **public void next(Container parent)**
- **public void show(Container parent, String name)**

课程配套代码 ➔ sample.awt.layout.CardLayoutSample.java

11.1.7 容器的嵌套使用

利用容器嵌套可以在某个原本只能包含一个组件的区域中显示多个组件。

课程配套代码 ➔ sample.awt.layout.FlowLayoutSample.java

11.2 GUI 事件处理

11.2.1 Java 事件和事件处理机制

从 JDK 1.1 开始，Java 采用了一种名为“事件代理模型”(Event Delegation Model)的事件处理机制。基本原理如下：

1. 事先定义多种事件类型

2. 约定各种 GUI 组件在与用户交互时，遇到特定操作则会触发相应的事件，即自动创建事件类对象并提交给 Java 运行时系统
3. 系统接收到事件类对象后，立即将其发送给专门的事件处理对象，该对象调用其事件处理方法，处理先前的事件类型对象，实现预期的处理逻辑

若需要关注某个组件产生的事件，则可以在该组件上注册适当的事件处理方法，实际上注册的事件处理器方法所属类型的一个对象——事件监听器。如图11.5所示。

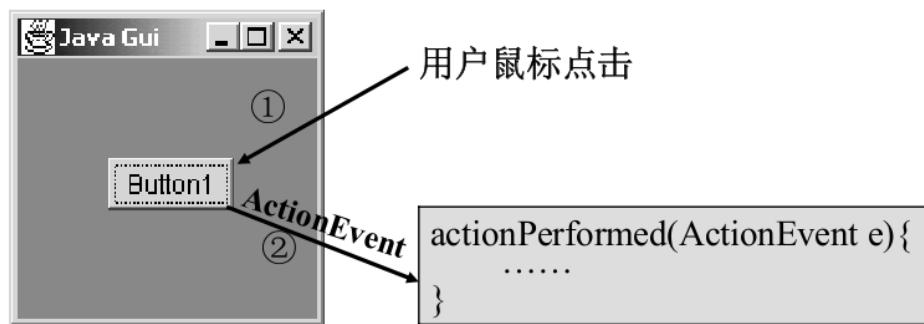


图 11.4 事件处理机制示例

11.2.2 事件处理相关概念

事件 (Event) 一个事件类型的对象，用于描述了发生什么事情，当用户在组件上进行操作时会触发相应的事件。

事件源 (Event Source) 能够产生事件的 GUI 组件对象，如按钮、文本框等。

事件处理方法 (Event Handler) 能够接收、解析和处理事件类对象，实现与用户交互功能的方法。

事件监听器 (Event Listener) 调用事件处理方法的对象。

课程配套代码 ▶ sample.awt.event.ActionEventSample.java

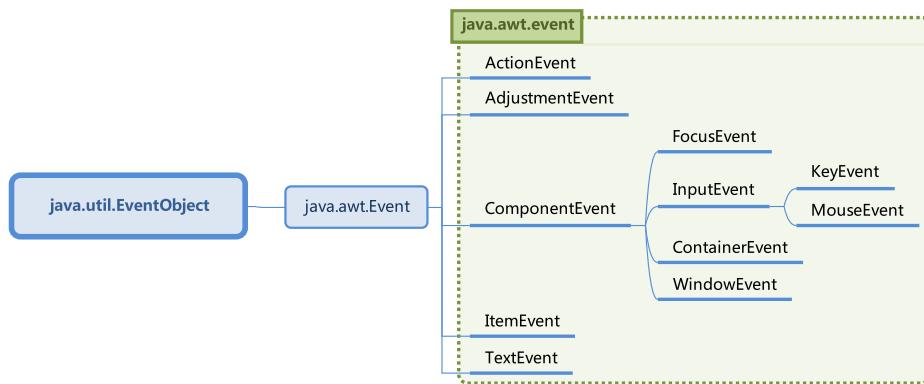


图 11.5 事件处理机制示例

表 11.2 GUI 事件及相应监听器接口

事件类型	相应监听器接口	监听器接口中的方法
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
Mouse	MouseListener	mousePressed(MouseEvent) ...
MouseMotion	MouseMotionListener	mouseDragged(MouseEvent) ...
Key	KeyListener	keyPressed(KeyEvent)
Focus	FocusListener	focusGained(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
Component	ComponentListener	componentMoved(ComponentEvent) ...
Window	WindowListener	windowClosing(WindowEvent) ...
Container	ContainerListener	componentAdded(ContainerEvent) ...
Text	TextListener	textValueChanged(TextEvent) ...

11.2.3 GUI 事件类型层次

11.2.4 GUI 事件及相应监听器接口

11.2.5 多重事件监听器

- 一般情况下，事件源可以产生多种不同类型的事件，因而可以注册（触发）多种不同类型的监听器。

- 一个事件源组件上可以注册多个监听器，针对同一个事件源的同一种事件也可以注册多个监听器，一个监听器可以被注册到多个不同的事件源上。

课程配套代码 ➔ sample.awt.event.MultActionsEventSample.java

11.2.6 事件适配器

- 当创建事件监听器类时，需要实现相应的监听器接口，而实现类中又必须重写/实现接口中的每一个抽象方法，这在 GUI 事件处理过程中经常会成为一种负担。
- 事件适配器使用了设计模式中的缺省适配器（Default Adapter）。事件适配器类（Adapter）是针对大多数事件监听器接口定义的相应抽象类，适配器类实现了相应监听器接口中所有的方法，但不做任何事。

课程配套代码 ➔ sample.awt.event.EventAdapterSample.java

11.2.7 事件适配器

表 11.3 常用的 GUI 事件适配器

监听器接口	对应适配器类	说明
MouseListener	MouseAdapter	鼠标事件适配器
MouseMotionListener	MouseMotionAdapter	鼠标运动事件适配器
WindowListener	WindowAdapter	窗口事件适配器
FocusListener	FocusAdapter	焦点事件适配器
KeyListener	KeyAdapter	键盘事件适配器
ComponentListener	ComponentAdapter	组件事件适配器
ContainerListener	ContainerAdapter	容器事件适配器

11.2.8 内部类和匿名类在 GUI 事件处理中的应用

监听器类中封装的业务逻辑具有非常强的针对性，一般没有重用价值，因此经常采用内部类或匿名类的形式来实现。

将示例代码改为匿名类模式

课程配套代码 [sample.awt.layout.CardLayoutSample.java](#)

11.3 Applet

Applet 也称 Java 小程序，在支持 Java 的浏览器环境中运行，通常用于在网页中实现嵌入图片、播放声音等多媒体功能，或添加其他的客户端处理逻辑（如网络计算器）。

严格的说，Applet 是能够嵌入到 HTML 页面中，且可以通过 Web 浏览器下载并执行的一种 Java 程序。

目前，该项技术在新项目中已经很少使用。

11.3.1 Applet 生命周期

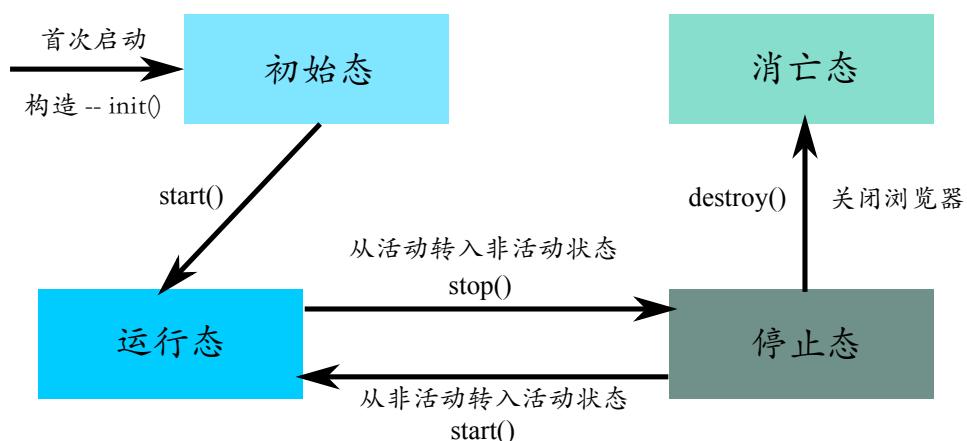


图 11.6 Applet 生命周期

11.3.2 Applet 程序示例

示例代码：[GoodbyeApplet.java](#)

```
1 import java.applet.Applet;
2 import java.awt.Color;
3 import java.awt.Graphics;
4
5 public class GoodbyeApplet extends Applet {
6     String text;
```

```
7  public void init() {  
8      text = "Hello Applet, goodbye!";  
9      this.setBackground(new Color(120, 180, 140));  
10 }  
11 public void paint(Graphics g) {  
12     g.drawString(text, 25, 25);  
13 }  
14 }
```

示例代码： test.html

```
1 <html>  
2 <applet code="HelloApplet.class" width=200 height=150> </applet>  
3 </html>
```

可以使用 Applet viewer 小工具运行示例：

```
1 >appletviewer test.html
```

11.4 Swing 概述

Swing 是建立在 AWT 基础上的一种增强型 Java GUI 组件和工具集，Swing 与 AWT 的关系如下：

- 使用轻量组件以替代 AWT 中的绝大部分重量组件。
- 提供 AWT 所缺少的一些附件组件和观感控制机制。
- 提供更好的平台无关性。

重量组件 (Heavy-Weight Components)

- 重量组件通过委托对等组件（对等组件指底层平台，如 Windows 操作系统的用户界面组件）来完成具体工作，包括组件的绘制和事件响应。AWT 中的组件均为重量组件，或者说，AWT 组件只是对本地对等组件的封装。
- 开销大、效率低、无法实现组件的“透明”效果。

轻量组件 (Light-Weight Components)

- 轻量组件不存在本地对等组件，通过 Java 绘图技术在其所在的容器窗口中绘图得到。
- 能够实现组件的透明效果，能够做到不同平台上的一致表现。
- 组件绘制和事件处理机制的开销小。
- 轻量组件最终需要包含在一个重量容器中。因此，Swing 组件中的几个顶层容器（如 JFrame、JDialog 和 JApplet）采用了重量组件，其余的均为轻量组件。
- 不建议轻重组件混用。

可视化组件 (Visual Component)

- 和 AWT 组件类似，Swing 组件也分为可视化和非可视化组件。
- 可视化组件为能够显示特定形状、颜色和尺寸的组件；非可视化组件也称支持类，如布局管理器等。
- Swing 可视化组件类名均以 J 开头。

11.5 Swing 典型组件

11.5.1 JFrame

1. JFrame 继承并扩充了 java.awt.Frame 类。

JFrame 不再是一个单一容器，而是由相互间存在包含关系的多个不同容器面板 (JRootPane, GlassPane, LayeredPane, ContentPane) 组成，我们实际上只使用其中的内容面板 (ContentPane)。

2. JFrame 实现了 javax.swing.WindowConstants 接口，该接口中定义了用于控制窗口关闭操作的整型常量，包括：

- DO NOTHING ON CLOSE
- HIDE ON CLOSE
- DISPOSE ON CLOSE
- EXIT ON CLOSE

以下给出 JFrame 示例代码。

示例代码：TestJFrame.java

```
1 import java.awt.Color;
2 import java.awt.Container;
3 import java.awt.FlowLayout;
4 import javax.swing.JFrame;
5 import javax.swing.JLabel;

7 public class TestJFrame {
8     public static void main(String[] args){
9         JFrame jf = new JFrame("My Test");
10        Container c = jf.getContentPane();
11        c.setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));
12        JLabel greet = new JLabel("Hello, World!");
13        JLabel bye = new JLabel("Bye, World!");
14        bye.setBackground(Color.BLUE);
15        bye.setOpaque(true); // 设置为不透明
16        c.add(greet);
17        c.add(bye);
18        c.setBackground(Color.GREEN);
19        jf.setSize(400, 300);
20        jf.setLocation(400, 200);
21        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // 等同于显示注册
22        jf.setVisible(true);
23    }
24 }
```

为了方便开发，从 JDK5.0 开始 JFrame 类重写了其 add()、remove() 和 setLayout() 方法，这些重写后的方法将针对 JFrame 的添加组件、移除组件和设置布局管理器等操作自动转发给其内容面板 contentPane，以实现对 contentPane 的直接控制。

对上述代码的改写：

```
1 JFrame jf = new JFrame("My Test");
2 jf.setLayout(new FlowLayout(FlowLayout.LEFT, 20, 20));
3 JLabel greet = new JLabel("Hello, World!");
```

```
4 jf.add(greet);
5 jf.getContentPane().setBackground(Color.GREEN); // 注意
```

注意：`setBackground()` 方法在 `JFrame` 类中并没有进行重写，因此不会将针对 `JFrame` 的颜色设置操作自动转发到其内容面板。

11.5.2 Swing 按钮、菜单和工具条

JButton 能够实现更复杂的显式效果，例如可以使用图片作为按钮标签、设置快捷键和添加工具提示信息。

菜单 同样分为菜单条、菜单和菜单项（`JMenuBar`、`JMenu`、`JMenuItem`），用法与 AWT 完全相同。

工具条 工具条是用于显示常见组件的条形容器，一般用法是向工具条中添加一系列图标形式的按钮，并将之置于窗口上方边缘。例如，`BorderLayout` 布局的北部区域，对于大多数的外观，用户可以用鼠标直接将工具栏拖到 `BorderLayout` 布局的其他未添加组件的边缘区域，如西部或东部，也可以将之拖出到单独的窗口中显示。

示例代码：[TestSwing.java](#)

```
1 import java.awt.event.*;
2 import javax.swing.*;

4 public class TestSwing implements ActionListener {
5     public static void main(String[] args) {
6         new TestSwing().createUI();
7     }
8
9     public void createUI() {
10        JFrame jf = new JFrame("Test Swing");
11        JMenuBar jmb = new JMenuBar();
12        JMenu menu_file = new JMenu("File");
13        JMenu menu_help = new JMenu("Help");
14        JMenuItem mi_new = new JMenuItem("New");
15        JMenuItem mi_open = new JMenuItem("Open");
16        JMenuItem mi_save = new JMenuItem("Save");
```

```
17     mi_new.addActionListener(this);
18     mi_open.addActionListener(this);
19     mi_save.addActionListener(this);

21     mi_new.setMnemonic('N');
22     mi_open.setMnemonic('O');
23     mi_save.setMnemonic('S');
24     menu_file.setMnemonic('F');
25     menu_help.setMnemonic('h');

27     menu_file.add(mi_new);
28     menu_file.add(mi_open);
29     menu_file.add(mi_save);

31     jmb.add(menu_file);
32     jmb.add(menu_help);

34     JToolBar jtb = new JToolBar();
35     JButton button_new = new JButton("NEW");
36     JButton button_open = new JButton("OPEN");
37     JButton button_save = new JButton("SAVE");

39     button_new.setActionCommand("New");
40     button_open.setActionCommand("Open");
41     button_save.setActionCommand("Save");

44     button_new.setToolTipText("Create a new file");
45     button_open.setToolTipText("Open the file");
46     button_save.setToolTipText("Save the file");

48     button_new.addActionListener(this);
49     button_open.addActionListener(this);
50     button_save.addActionListener(this);

52     jtb.add(button_new);
```

```
53     jtb.add(button_open);
54     jtb.add(button_save);

56     JPanel jp = new JPanel();
57     JButton button_start = new JButton("Start");
58     JButton button_stop = new JButton("Stop");
59     button_start.addActionListener(this);
60     button_stop.addActionListener(this);
61     jp.add(button_start);
62     jp.add(button_stop);

64     jf.setJMenuBar(jmb); // Menu 不需要使用 add() 方法添加
65     jf.add(jtb, "North");
66     jf.add(jp, "South");

68     jf.setSize(600, 400);
69     jf.setLocation(400, 200);
70     jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
71     jf.setVisible(true);
72 }

74 @Override
75 public void actionPerformed(ActionEvent e) {
76     System.out.println(e.getActionCommand());
77 }
78 }
```

11.5.3 标准对话框

使用标准对话框（ JOptionPane）可以实现程序与用户的便捷交互，如向用户发送错误通知、警告/确认用户操作、接收用户输入或选择的简单信息等。

示例代码：标准对话框示例

```
1 @Override
2 public void actionPerformed(ActionEvent e) {
```

```
3  String s = e.getActionCommand();
4  if (s.equals("Error")) {
5      JOptionPane.showMessageDialog(null, "这是一个错误提示对话框", "错误提示",
6          JOptionPane.ERROR_MESSAGE);
7  } else if (s.equals("Confirm Quit")) {
8      int result = JOptionPane.showConfirmDialog(null, "真的要退出程序么？",
9          "请确认退出", JOptionPane.YES_NO_OPTION);
10     if (result == JOptionPane.OK_OPTION) {
11         System.exit(0);
12     }
13 } else if (s.equals("Warning")) {
14     Object[] options = { "继续", "撤销" };
15     int result = JOptionPane.showOptionDialog(null, "本操作可能导致数据丢失",
16         "警告", JOptionPane.DEFAULT_OPTION,
17         JOptionPane.WARNING_MESSAGE, null, options, options[0]);
18     if (result == 0)
19         System.out.println("继续操作");
20 } else if (s.equals("Input")) {
21     String name = JOptionPane.showInputDialog("请输入姓名：");
22     if (name != null)
23         System.out.println("输入的姓名为" + name);
24 } else if (s.equals("Choice")) {
25     Object[] possibleValues = { "体育", "政治", "经济", "文化" };
26     Object selectedValue = JOptionPane.showInputDialog(null,
27         "Choice one", "Input", JOptionPane.INFORMATION_MESSAGE,
28         null, possibleValues, possibleValues[0]);
29     String result = (String) selectedValue;
30     if (result != null)
31         System.out.println("你的选择是：" + result);
32 }
33 }
```

11.5.4 表格和树

- **javax.swing.JTable**

用于以传统的表格形式来显示数据，通过注册监听器的方式关联响应的处理逻辑

辑。

- 表头：标题行，给出每一列（字段）的名称。
- 表体：由多行多列、规则矩阵形式的单元格组成，真正的数据信息则显示在每个单元格中。

- **javax.swing.JTree**

以树状结构分层次显示数据信息，例如操作系统提供的资源管理器。

表格示例

```
1 import java.awt.event.WindowAdapter;
2 import java.awt.event.WindowEvent;
3 import javax.swing.JFrame;
4 import javax.swing.JScrollPane;
5 import javax.swing.JTable;
6 public class TableExample {
7     public static void main(String[] args) {
8         JFrame myFrame = new JFrame("Table Example");
9         Object data [][] = {
10             {1, "张三", "男", "18", "010.82607080"},
11             {2, "李四", "女", "24", "010.82607080},
12             {3, "王五", "男", "30", "010.82607080},
13             // ...
14         };
15         String columnNames[] = {
16             "编号", "姓名", "性别", "年龄", "电话"};
17         JTable table = new JTable(data, columnNames);
18         table.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
19         JScrollPane pane = new JScrollPane(table);
20         myFrame.add("Center", pane);
21         myFrame.setSize(450, 250);
22         myFrame.addWindowListener(new WindowAdapter(){
23             public void windowClosing(WindowEvent e) {
24                 System.exit(0);
25             }
26         });
27 }
```

```
27     myFrame.setVisible(true);
28 }
29 }
```

表格其他参考例程参考如下链接：

1. A Simple Interactive JTable Tutorial

<http://www.javalobby.org/articles/jtable/>

树示例

```
1 public class TreeExample {
2     public static DefaultMutableTreeNode createNodes() {
3         DefaultMutableTreeNode rootNode = new DefaultMutableTreeNode("Java Stuff");
4         DefaultMutableTreeNode resources = new DefaultMutableTreeNode("Resources");
5         DefaultMutableTreeNode tools = new DefaultMutableTreeNode("Tools");
6         rootNode.add(resources);
7         rootNode.add(tools);
8         DefaultMutableTreeNode webSites = new DefaultMutableTreeNode("Web Sites");
9         DefaultMutableTreeNode books = new DefaultMutableTreeNode("Books");
10        resources.add(webSites);
11        resources.add(books);
12        tools.add(new DefaultMutableTreeNode("JBuilder"));
13        tools.add(new DefaultMutableTreeNode("Visual J++"));
14        return rootNode;
15    }
16    public static void main(String[] args) {
17        JFrame myFrame = new JFrame("Tree Example");
18        DefaultMutableTreeNode rootNode = createNodes();
19        JTree tree = new JTree(rootNode);
20        tree.setRootVisible(true);
21        JScrollPane pane = new JScrollPane();
22        pane.setViewportView(tree);
23        myFrame.add(pane, "Center");
24        myFrame.setSize(400, 250);
25        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26        myFrame.setVisible(true);
```

```
27 }  
28 }
```

11.5.5 JTable 和 JTree 的 MVC 模式

JTable 和 JTree 采用了相对独立的方式向组件提供要显示的数据，即当显示/处理的数据结构较复杂时，将 GUI 组件结构分为相对独立的模型、视图、控制器三个模块，模块间存在专门的分工和协作关系。

1. 模型（Model） 维护数据并提供数据访问方法，即数据和数据的处理逻辑。
2. 视图（View） 绘制模型的视觉表现，即显示数据。视图就是用户能够看到并与之进行交互的用户界面。
3. 控制器（Controller） 负责处理事件或者说程序的流程控制，接受用户输入，并调用/操控模型和视图以实现用户需求。

MVC 作用原理如图11.7所示。

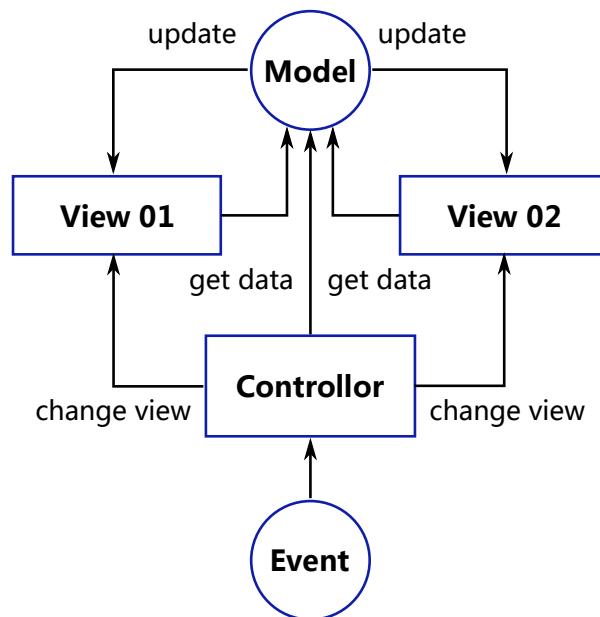


图 11.7 MVC 作用原理

11.5.6 定时器

`javax.swing.Timer` 提供了定时器功能，用于在指定的时间延迟之后触发 `ActionEvent` 事件，以执行所需的处理逻辑。具体的做法是：首先创建 `Timer` 对象，并在其上注册一个或多个 `ActionListener` 类型的监听器，在监听器事件处理方法 `actionPerformed()` 中应以实现给出要延时执行的任务代码，然后调用 `Timer` 对象的 `start()` 方法启动定时器即可。

`Timer` 包含的常用方法如下：

setRepeats() 设置计时器的动作。

setInitialDelay() 设置首次延迟时间。

start() 开始计时器。

stop() 停止计时器。

restart() 恢复计时器。

☒ 12 ☒ 异常处理

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第七周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 掌握 Java 异常的概念和分类
2. 深入理解 Java 异常处理机制

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

12.1 异常的概念及分类

12.1.1 什么是异常

在 Java 语言中，程序运行出错被称为出现异常。异常（Exception）是程序运行过程中发生的事件，该事件可以中断程序指令的正常执行流程。Java 异常分为两大类：

错误（Error） 是指 JVM 系统内部错误、资源耗尽等严重情况。

违例（Exception） 则是指其他因编程错误或偶然的外在因素导致的一般性问题，例如对负数开平方根、空指针访问、试图读取不存在的文件以及网络连接中断等。

给出一个 Java 运行时异常的示例：

示例代码：[TestException.java](#)

```
1 public class TestException {  
2     public static void main(String[] args) {  
3         String friends [] = {"Lisa", "Bily", "Kessy"};  
4         for(int i = 0; i < 5; i++) {  
5             System.out.println(friends [i]);  
6         }  
7         System.out.println("\n this is the end");  
8     }  
9 }
```

标准输出如下，程序编译通过，但运行时出错。

[output](#)

```
Lisa  
Bily  
Kessy  
Exception in thread "main" java.lang.  
ArrayIndexOutOfBoundsException: 3  
at TestException.main(TestException.java:6)
```

12.1.2 Java 异常分类

Throwable 类是 Java 语言中所有异常类的父类。Java 异常分类如图12.1所示。

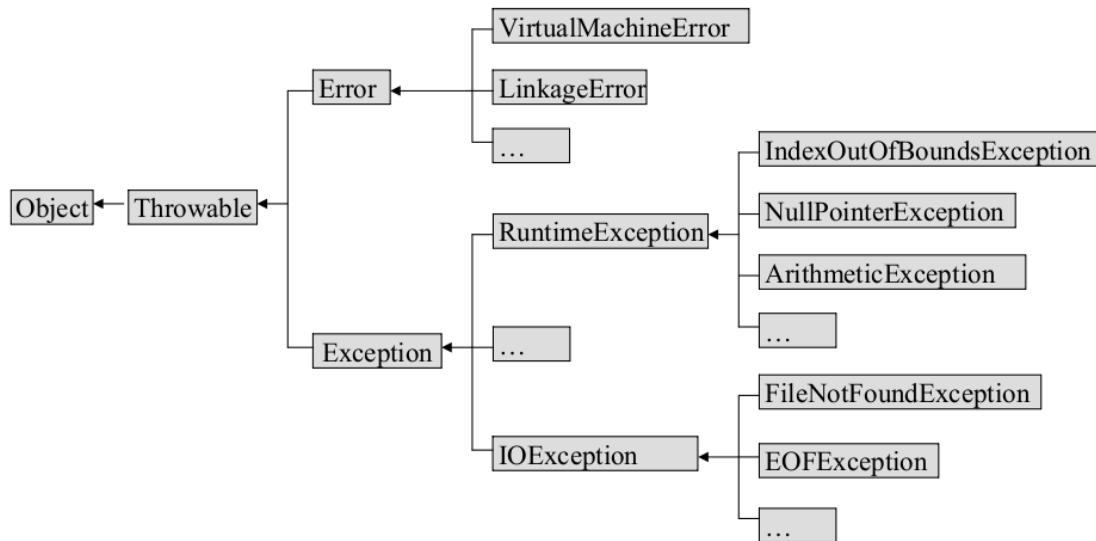


图 12.1 Java 异常分类

12.1.3 常见错误

链接错误 (LinkageError)

LinkageError 是指程序链接错误。例如，一个类中用到另外一个类，在编译前一个类之后，后一个类发生了不相容的改变时，再使用前一个类则会出现链接错误。最常见的就是后一个类的.class 文件被误删除。

虚拟机错误 (VirtualMachineError)

当 Java 虚拟机崩溃或用尽了它继续操作所需的资源时，会抛出该错误。其中比较有代表性的是 StackOverflowError，当应用程序递归太深而导致栈内存溢出时会出现该异常。

示例代码：[TestVMError.java](#)

```
1 public class TestVMError {  
2     public static void main(String[] args) {  
.....
```

```
3     TestVModelError t = new TestVModelError();
4     t.f(100000);
5 }
6 public int f(int n) {
7     if (n <= 0) {
8         return 0;
9     }
10    int k = n * this.f(n-1);
11    return k;
12 }
13 }
```

标准输出如下：

output

```
Exception in thread "main" java.lang.StackOverflowError
at TestException.f(TestException.java:7)
at TestException.f(TestException.java:10)
```

12.1.4 常见异常

运行时异常（RuntimeException）

运行时异常主要包括：

- 错误的类型转换；
- 数组下标越界；
- 空指针访问。

其中，空指针异常（NullPointerException）是如果试图访问不指向任何对象的引用变量的成员，将会产生空指针异常。例如：

```
1 Person p = null;
2 System.out.println(p.age);
```

IO 异常（IOException）

- 从一个不存在的文件中读取数据；

- 越过文件结尾继续读取；
- 连接一个不存在的 URL。

以下给出 IOException 的一个示例，注意下述代码无法通过编译！

示例代码：TestIOException.java

```
1 import java.io.*;
2 public class TestIOException {
3     public static void main(String[] args) {
4         FileInputStream in = new FileInputStream("myfile.txt");
5         int b;
6         b = in.read();
7         while (b != -1) {
8             System.out.print((char) b);
9             b = in.read();
10        }
11        in.close();
12    }
13 }
```

对上述代码无法编译的解释

只要是有可能出现 IOException 的 Java 代码，在编译时就会出错，而不会等到运行时才发生。编译出错信息大致如下：

output

TestIOException.java:4: 未报告的异常 java.io.FileNotFoundException;

必须对其进行捕捉或声明以便抛出

 FileInputStream in = new FileInputStream("myfile.txt");

... ...

12.2 Java 异常处理机制

Java 异常处理的宗旨包括：

- 返回到一个安全和已知的状态；

- 能够使用户执行其他的命令；
- 如果可能，则保存所有的工作；
- 如果有必要，可以退出以避免造成进一步的危害。

Java 异常处理的基本机制是：

- Java 程序执行过程中如出现异常，系统会监测到并自动生成一个相应的异常类对象，然后再将它交给运行时系统；
- 运行时系统再寻找相应的代码来处理这一异常。如果 Java 运行时系统找不到可以处理异常的代码，则运行时系统将终止，相应的 Java 程序也将退出。
- 程序本身通常对错误（Error）无能为力，因而一般只处理违例（Exception）。

12.2.1 捕获异常

Java 语言捕获异常的基本程序逻辑结构如下：

```
1 try {  
2     ... //可能产生异常的代码  
3 } catch (ExceptionName1 e) {  
4     ... //当产生 ExceptionName1 型异常时的处置措施  
5 } catch (ExceptionName2 e) {  
6     ... //当产生 ExceptionName2 型异常时的处置措施  
7 } finally {  
8     ... //无条件执行的语句  
9 }
```

以下给出一段捕获处理数组访问越界异常的示例：

示例代码：[ArrayIndexOutOfBoundsExceptionSample.java](#)

```
1 public class ArrayIndexOutOfBoundsExceptionSample {  
2     public static void main(String[] args) {  
3         String friends []={”Lisa”, ”Billy”, ”Kessy”};  
4         try {  
5             for( int i = 0; i < 5; i++) {  
6                 System.out.println( friends [i]);  
7             }  
8         }  
9     }  
10 }
```

```
8 } catch(ArrayIndexOutOfBoundsException e) {  
9     System.out.println("index err");  
10 }  
11 System.out.println("\nthis is the end");  
12 }  
13 }
```

12.2.2 使用 finally 语句

Java 异常处理中，finally 语句是可选的，其作用是为异常处理提供一个统一的出口，使得在控制流转到程序的其他部分以前，能够对程序的状态作统一的管理。

示例代码：FinallySample.java

```
1 public class FinallySample {  
2     public static void main(String[] args) {  
3         String friends[]={"Lisa","Billy","Kessy"};  
4         try {  
5             for (int i = 0; i < 5; i++) {  
6                 System.out.println(friends[i]);  
7             }  
8         } catch(ArrayIndexOutOfBoundsException e) {  
9             System.out.println("index err");  
10            return;  
11        } finally {  
12            System.out.println("in finally block!");  
13        }  
14        System.out.println("this is the end");  
15    }  
16 }
```

注意，不论 try 代码块中是否发生了异常事件，finally 块中的语句都会被执行。当 catch 语句块中出现 return 语句时，finally 语句块同样会执行。上述代码的输出：

output

Lisa

Billy

Kessy
index err
in finally block!

12.2.3 操纵异常对象

发生异常时，系统将自动创建异常类对象，并将作为实参传递给匹配的 catch 语句块的形参，这样我们就可以在语句块中操纵该异常对象了。主要使用异常类的父类 Throwable 中定义的两个成员方法：

- public String getMessage() 返回描述当前异常的详细消息字符串；
- public void printStackTrace() 用来跟踪异常事件发生时运行栈的内容，并将相关信息输出到标准错误输出设备。本方法比较常用，在没有找到适合的异常处理代码时，系统也会自动调用该方法输出错误信息。

可以参考以下代码追踪运行栈信息：

示例代码：A.java

```
1 public class A {  
2     public void work(int[] a) {  
3         String s = this.contain(a, 3);  
4         System.out.println("Result: " + s);  
5     }  
  
7     public String contain(int[] source, int dest) {  
8         String result = "no!";  
9         try {  
10             for (int i = 0; i < source.length; i++) {  
11                 if (source[i] == dest)  
12                     result = "yes!";  
13             }  
14         } catch(Exception e) {  
15             System.out.println("异常信息: " + e.getMessage());  
16             System.out.println("运行栈信息: ");  
.....
```

```
17     e.printStackTrace();
18     result = "error!";
19 }
20 return result;
21 }
22 }
```

示例代码： MyTest.java

```
1 public class MyTest {
2     public static void main(String[] args) {
3         A tst = new A();
4         tst.work(null);
5     }
6 }
```

程序输出结果如下：

output

```
Exception Message: null
Stack Trace:
java.lang.NullPointerException
    at A.contain(A.java:9)
    at A.work(A.java:3)
    at MyTest.main(MyTest.java:4)
Result: error!
```

12.2.4 捕获和处理 IOException

以下给出捕获和处理 IOException 的示例：

示例代码： TestIOException.java

```
1 import java.io.*;
2 public class TestIOException {
3     public static void main(String[] args) {
```

```
4   try {
5       FileInputStream in = new FileInputStream("myfile.txt");
6       int b;
7       b = in.read();
8       while(b != -1) {
9           System.out.print((char) b);
10      b = in.read();
11  }
12  in.close();
13 } catch (FileNotFoundException e) {
14     System.out.println("File is missing!");
15 } catch (IOException e) {
16     e.printStackTrace();
17 }
18 System.out.println("It's ok!");
19 }
20 }
```

FileNotFoundException 是 IOException 的子类，基于多态性机制，后一个 catch 语句也可以处理 FileNotFoundException，因此前一个 catch 语句块可以取消，但这样就无法区分是“文件不存在”引发异常或其他 I/O 异常了。

异常处理知识点

- 对于只可能产生 RuntimeException 的代码可以不使用 try-catch 语句进行处理，如果对于这些相对安全的代码仍然采用了 try 语句块的形式，则 try 后可以省略 catch 语句块或 finally 语句块，但不能同时省略。
- 如果试图捕获和处理代码中根本不可能出现的异常，编译器也会指出这种不当行为。

12.2.5 声明抛出异常

声明抛弃异常是 Java 中处理违例的第二种方式。如果一个方法中的代码在运行时可能生成某种异常，但在本方法中不必、或者不能确定如何处理此类异常时，则可以声明抛弃该异常；此时方法中将不对此类异常进行处理，而是由该方法的调用者负责处理。语法格式如下：

```
1 [< 修饰符 >] < 返回值类型 > < 方法名 > (< 参数列表 >) [throws < 异常类型 > [,<
2   异常类型 >]*] {
3   [< Java语句 >]*
4 }
```

示例代码： TestThrowsException.java

```
1 import java.io.*;
2 public class TestThrowsException {
3   public static void main(String[] args) {
4     TestThrowsException t = new TestThrowsException();
5     try {
6       t.readFile();
7     } catch (IOException e) {
8       System.out.println(e);
9     }
10   }
11   public void readFile() throws IOException {
12     FileInputStream in = new FileInputStream("myfile.txt");
13     int b;
14     b = in.read();
15     while (b != -1) {
16       System.out.print((char) b);
17       b = in.read();
18     }
19     in.close();
20   }
21 }
```

声明抛出异常的注意事项

- 除非事先约定，否则在开发过程中不要在自己编写的方法中采用抛出异常的方式。
- 重写方法不允许抛出比被重写方法范围更大的异常类型。例如 IOException 重写后抛出 FileNotFoundException 和 EOFException 被允许，而抛出 Exception 则不被

允许。

12.2.6 人工抛出异常

Java 异常类对象除了在程序运行出错时由系统自动生成并抛出之外，也可根据需要人工创建并抛出：

```
1 IOException e = new IOException(); // 创建异常类对象  
2 throw e; // 抛出操作，即将该异常对象提交给Java运行环境
```

被抛出的必须是 `Throwable` 或其子类类型的对象。例如，下述语句因为人工抛出的并非 `Throwable` 或其子类的对象，在编译时会产生语法错误：

```
1 throw new String("want to throw");
```

以下给出一个人工抛出异常的示例：

示例代码： `TestThrowException.java`

```
1 import java.util.Scanner;  
  
3 public class TestThrowException {  
4     public static void main(String[] args) {  
5         TestThrowException t = new TestThrowException();  
6         System.out.print("Please input your age: ");  
7         System.out.print("Your age: " + t.inputAge());  
8     }  
9     public int inputAge() {  
10        int result = -1;  
11        Scanner scan = new Scanner(System.in);  
12        while (true) {  
13            try {  
14                result = scan.nextInt();  
15                if (result < 0 || result > 130) {  
16                    Exception me = new Exception("You come from Mars? ");  
17                    throw me;  
18                }  
19            break;  
.....
```

```
20 } catch (Exception e1) {  
21     System.out.println(e1.getMessage() + "Please input your age again: ");  
22     continue;  
23 }  
24 }  
25 return result;  
26 }  
27 }
```

上述代码所述的情况利用异常处理机制实现数据取值范围的检查并不太合适。应用异常处理机制的原则如下：

- 当明确知道可能出错的地方或能够通过简单的检查而有效防止错误发生，就应该使用 if-else 语句来预防错误发生；
- 只有当我们无法明确知道错误发生之处或无法完全避免异常，才不得不通过异常处理的方式来捕获和处理异常。

12.3 用户自定义异常

Java 语言及许多类库针对常见异常状况已事先定义了相应的异常类型，并在程序运行出错时由系统自动创建相应对象并进行抛出、捕获和处理，因此一般不需要用户人工抛出异常对象或定义新的异常类型，但针对特殊的需要也可以这样做。

我们一般通过继承 Exception 类来实现异常类型自定义，由于用户自定义的异常不会由系统自动检测并抛出，所以只能靠人工触发并抛出。

以下给出用户自定义异常的示例：

示例代码：[CustomizingException.java](#)

```
1 public class CustomizingException extends Exception {  
2     private int idnumber;  
3  
4     public MyException(String message, int id) {  
5         super(message);  
6         this.idnumber = id;  
7     }  
8 }
```

```
9  public int getId() {  
10     return idnumber;  
11 }  
12 }
```

上述自定义异常类代码的构造方法中使用 super 调用其父类 Exception 的有参构造方法，以便在创建异常对象时将用户的定制的报错信息传递给父类中定义的 private 属性 Message（该属性由 Throwable 类定义），将来在捕获和处理异常时就可以通过调用该对象的 getMessage() 方法访问到该信息。

示例代码：TestCustomizingException.java

```
1 public class TestCustomizingException {  
2     public void regist (int num) throws MyException {  
3         if (num < 0) {  
4             throw new MyException("人数为负值, 不合理", 3);  
5         }  
6         System.out.println("登记人数: " + num);  
7     }  
8  
9     public void manager() {  
10        try {  
11            regist (-100);  
12        } catch (MyException e) {  
13            System.out.println("登记失败, 出错种类" + e.getId());  
14            e.printStackTrace();  
15        }  
16        System.out.print("本次登记操作结束");  
17    }  
18  
19    public static void main(String args[]) {  
20        new TestCustomizingException().manager();  
21    }  
22 }
```

上述测试程序输出结果如下：

output

登记失败，出错种类 3

MyException: 人数为负值，不合理 ...

12.4 断言

12.4.1 什么是断言

从 JDK1.4 版本开始，Java 语言中引入了断言（Assert）机制，允许 Java 开发者在代码中加入一些检查语句，主要目的是用于程序调试。

- 当我们需要在约定的条件不成立时中断当前程序执行操作的话可以使用断言；
- 使用断言是为了在测试阶段确定程序内部出错位置和出错信息，而不是控制程序流程；
- 断言机制在用户定义的 boolean 表达式（判定条件）结果为 false 时抛出一个 Error 对象，其类型为 AssertionError；
- 作为 Error 的一种，断言失败也不需捕获处理或者声明抛出，一旦出现了则终止程序、不必进行补救和恢复。

12.4.2 启用和禁用断言

命令行开启断言功能

Java 运行时环境默认设置为关闭断言功能，因此在使用断言以前，需要在运行 Java 程序时先开启断言功能。方法如下：

```
>java -ea MyClass
```

或者：

```
>java -enableassertions MyClass
```

命令行关闭断言功能

```
>java -da MyClass
```

或者：

```
java - disableassertions MyClass
```

Eclipse IDE 开启断言

在项目上点击右键 ➔ Run As ➔ Run Configurations ➔ Arguments，在 VM arguments 中，加入 -enableassertions 或 -ea 即可。

12.4.3 使用断言

第一种断言的语法格式如下：

```
assert <boolean 表达式>;
```

以下给出使用第一种断言语法的示例代码：

示例代码：TestAssertion.java

```
1 public class TestAssertion {  
2     public static void main(String[] args) {  
3         new TestAssertion().process(-12);  
4     }  
5     public void process(int age) {  
6         assert age >= 0;  
7         System.out.println("您的年龄：" + age);  
8         // ---  
9     }  
10 }
```

以上程序执行的输出如下：

output

```
Exception in thread "main" java.lang.AssertionError  
at TestAssertion.process(TestAssertion.java:8)  
at TestAssertion.main(TestAssertion.java:4)
```

第二种断言的语法格式如下：

```
1 assert < boolean 表达式 >:< 表达式 2 >;
```

以下给出第二种断言格式的示例代码：

示例代码：[TestAssertion2.java](#)

```
1 public class TestAssertion2 {  
2     public static void main(String[] args) {  
3         new TestAssertion2().process(-12);  
4     }  
5     public void process(int age) {  
6         assert age >= 0: "年龄值不合理";  
7         System.out.println("您的年龄：" + age);  
8         //---  
9     }  
10 }
```

输出结果如下：

output

```
Exception in thread "main" java.lang.AssertionError: 年龄值不合理  
at TestAssertion.process(TestAssertion.java:8)  
at TestAssertion.main(TestAssertion.java:4)
```

说明

断言失败时，系统会自动将表达式 2 的值传递给新创建的 `AssertionError` 对象，进而将其转换为一个消息字符串保存起来，这样就可以在获得更多/更有针对性的检查失败细节信息。因此，其中的表达式 2 可以是任何基本数据类型或引用数据类型，但必须提供一个值，即不能为空值。

☒ 13 ☒ 高级 I/O 编程

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第七周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 深入理解 Java 的 I/O 原理
2. 掌握 Java 基本 I/O 流类型
3. 掌握 I/O 的几种应用编程方式

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

13.1 Java I/O 原理

13.1.1 Java I/O 原理

I/O (Input/Output) 基本概念

- 数据源 (Data Source)
- 数据宿 (Data Sink)
- 流 (Stream)

Java 中把不同的数据源与程序间的数据传输都抽象表述为流，java.io 包中定义了多种 I/O 流类型实现数据 I/O 功能。

13.1.2 Java I/O 流的分类

按照数据流动的方向

Java 流可分为输入流 (Input Stream) 和输出流 (Output Stream)。

- 输入流只能从中读取数据，而不能向其写出数据；
- 输出流则只能向其写出数据，而不能从中读取数据。
- (特例 `java.io.RandomAccessFile` 类)

根据数据流所关联的是数据源还是其他数据流

可分为节点流 (Node Stream) 和处理流 (Processing Stream)。

- 节点流直接连接到数据源；
- 处理流是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现增强的数据读/写功能，处理流并不直接连到数据源。

按传输数据的“颗粒大小”

可分为字符流 (Character Stream) 和字节流 (Byte Stream)。

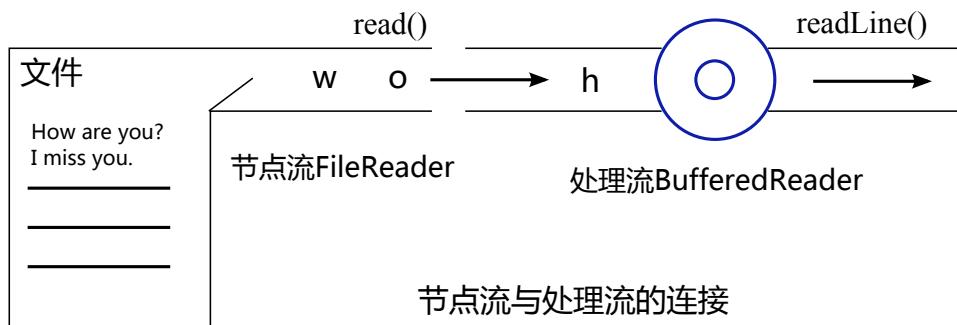


图 13.1 InputStream 和 OutputStream

- 字节流以字节为单位传输数据，每次传送一个或多个字节。
- 字符流以字符为单位传输数据，每次传送一个或多个字符。¹

Java 命名惯例

凡是以 InputStream 或 OutputStream 结尾的类型均为字节流，凡是以 Reader 或 Writer 结尾的均为字符流。

13.2 基础 I/O 流

13.2.1 InputStream and OutputStream

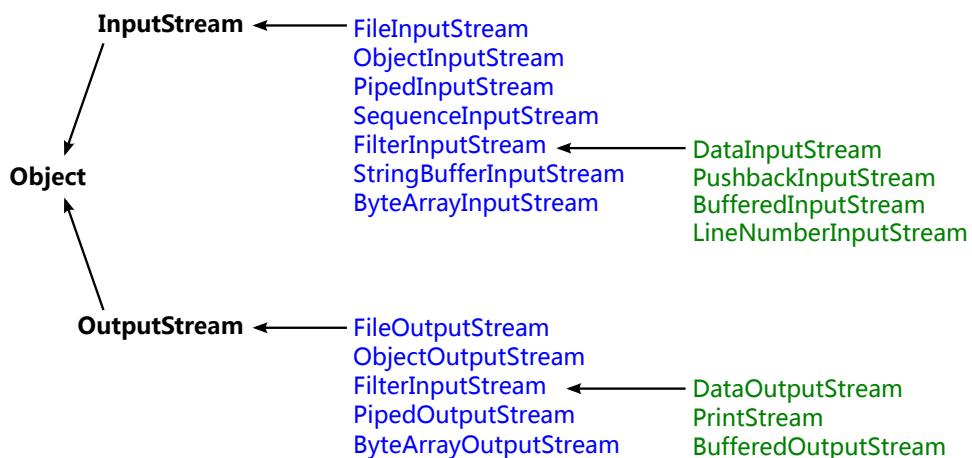


图 13.2 InputStream 和 OutputStream

¹从 JDK1.4 版本开始，Sun 公司引入了新的 Java I/O API (NIO, New Input/Output)，提供面向数据块、异步 I/O 操作。

13.2.2 InputStream

抽象类 `java.io.InputStream` 是所有字节输入流类型的父类，该类中定义了以字节为单位读取数据的基本方法，并在其子类中进行了分化和实现。

三个基本的 read 方法

- `int read()`
- `int read(byte[] buffer)`
- `int read(byte[] buffer, int offset, int length)`

其它方法

- `void close()`
- `int available()`
- `skip(long n)`
- `boolean markSupported()`

13.2.3 OutputStream

`java.io.OutputStream` 与 `java.io.InputStream` 对应，是所有字节输出流类型的抽象父类。

三个基本的 write 方法

- `void write(int c)`
- `void write(byte[] buffer)`
- `void write(byte[] buffer, int offset, int length)`

其它方法

- `void close()`
- `void flush()`

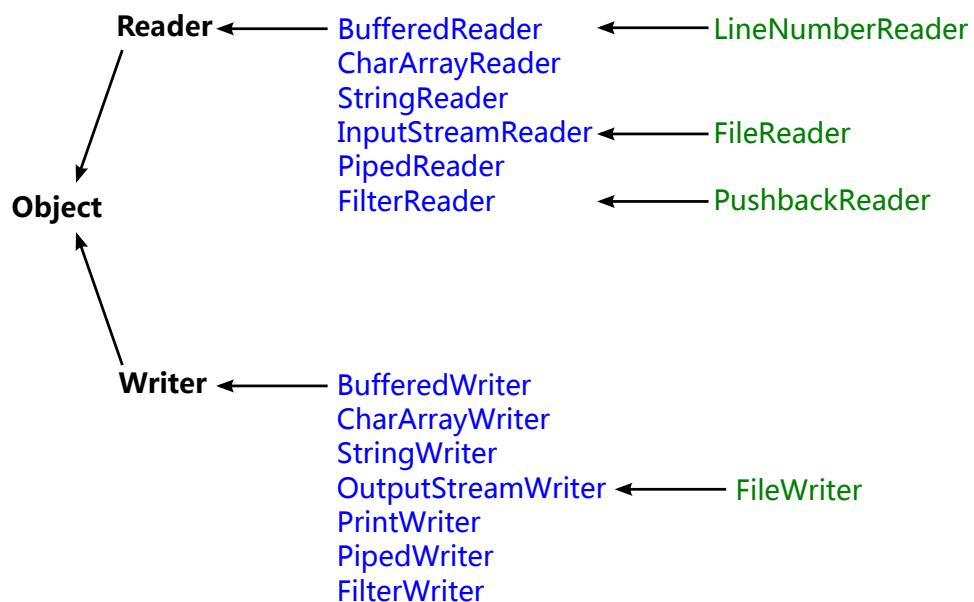


图 13.3 InputStream 和 OutputStream

13.2.4 Reader and Writer

13.2.5 Reader

抽象类 `java.io.Reader` 是所有字符输入流类型的父类，其中声明了用于读取字符流的有关方法。

三个基本的 read 方法

- `int read()`
- `int read(char[] cbuf)`
- `int read(char[] cbuf, int offset, int length)`

其它方法

- `void close()`
- `boolean ready()`
- `skip(long n)`
- `boolean markSupported()`
- `void mark(int readAheadLimit)`

13.2.6 Writer

java.io.Writer 与 java.io.Reader 类对应，是所有字符输出流类型的共同父类。

五个基本的 write 方法

- void write(int c)
- void write(char[] cbuf)
- void write(char[] cbuf, int offset, int length)
- void write(String string)
- void write(String string, int offset, int length)

其它方法

- void close()

13.3 常用 I/O 流类型

13.3.1 FileInputStream/FileOutputStream

- FileInputStream 用于读取本地文件中字节数据，FileOutputStream 用于将字节数据写出到文件。
- FileInputStream 不适合获取文本文件中的字符信息，要读取并显示的文件中如果含有双字节字符（如中文），则会显示乱码，此时应该采用字符流类型。
- 可以用于复制任何格式的文件，如文本、音视频以及可执行文件等二进制文件，因为以字节为单位进行数据复制时并不对文件内容进行解析。

示例代码：Fragment: 使用字节流实现文件复制

```
1 FileInputStream fis = new FileInputStream("in.txt");
2 FileOutputStream fos = new FileOutputStream("out.txt");
3 int read = fis.read();
4 while (read != -1) {
5     fos.write(read);
6     read = fis.read();
```

```
7    }
8    fis.close();
9    fos.close();
```

13.3.2 FileReader/FileWriter

- FileReader 用于以字符为单位读取文本文件， FileWriter 类用于将字符数据写出到文本文件。
- 字符 I/O 流类型只能处理文本文件，因为二进制文件中保存的字节信息不能正常解析为字符。

示例代码：Fragment: 使用字符流实现文件复制

```
1  FileReader fis = new FileReader("in.txt");
2  // The second arg is boolean append, true for appending, false for covering.
3  FileWriter fos = new FileWriter("out.txt", true);
4  int read = fis.read();
5  while (read != -1) {
6      fos.write(read);
7      read = fis.read();
8  }
9  fis.close();
10 fos.close();
```

13.3.3 BufferedReader/BufferedWriter

- BufferedReader 用于缓冲读取字符、字符数组或行，采用缓冲处理能够提高效率，该类所封装的字节输入流对象需要在构造方法中指定。
 - public BufferedReader(Reader in)
 - public BufferedReader(Reader in, int size) // size of buffer
- BufferedWriter 提供字符的缓冲写出功能，该类的 newLine() 方法可以写出平台相关的行分隔符来标记一行的终止，此分割符由系统属性 line.separator 确定。

示例代码：Fragment: 使用字符处理流实现文件复制

```
1 BufferedReader br = new BufferedReader(new FileReader("in.txt"));
2 BufferedWriter bw = new BufferedWriter(new FileWriter("out.txt"));
3 String s = br.readLine();
4 while (s != null) {
5     bw.write(s);
6     bw.newLine(); // notice.
7     s = br.readLine();
8 }
```

13.3.4 Other I/O Classes

- InputStreamReader/OutputStreamWriter
- PrintStream/PrintWriter
- DataInputStream/DataOutputStream
- CharArrayReader/CharArrayWriter

13.4 I/O 应用

13.4.1 属性信息的导入/导出

如果要永久记录用户自定义的属性，可以采用 Properties 类的 load()/store() 方法进行属性的导入/导出操作，即将属性信息写出到文件中和从文件中读取属性信息到程序。

示例代码：SaveProperties.java

```
1 import java.io.FileWriter;
2 import java.util.Properties;
3 public class SaveProperties {
4     public static void main(String[] args) {
5         try {
6             Properties ps = new Properties();
7             ps.setProperty("name", "Kevin");
8             ps.setProperty("password", "12345");
.....
```

```
9     FileWriter fw = new FileWriter("props.txt");
10    ps.store(fw, "loginfo");
11    fw.close();
12 } catch (Exception e) {
13     e.printStackTrace();
14 }
15 }
16 }
```

13.4.2 属性信息的导入/导出

示例代码：[LoadProperties.java](#)

```
1 import java.io .FileWriter;
2 import java. util .Properties;
3 public class LoadProperties {
4     public static void main(String[] args) {
5         try {
6             Properties ps = new Properties();
7             FileReader fr = new FileReader("props.txt");
8             ps.load(fr);
9             fr .close();
10            ps. list (System.out);
11        } catch (Exception e) {
12            e.printStackTrace();
13        }
14    }
15 }
```

文件 props.txt 内容如下：

```
1 #loginfo
2 #Sun Nov 04 21:20:17 CST 2012
3 password=12345
4 name=Kevin
```

标准输出如下：

```
-- listing properties --
password=12345
name=Kevin
```

output

13.4.3 对象序列化

概念

对象序列化（Object Serialization）是指将对象的状态数据以字节流的形式进行处理，一般用于实现对象的持久性，即长久保存一个对象的状态并在需要时获取该对象的信息以重新构造一个状态完全相同的对象。对象序列化可以理解为使用 I/O “对象流”类型实现对象读/写操作。

- **对象的持久性**（Object Persistence） 长久保存一个对象的状态并在需要时获取该对象的信息以重新构造一个状态完全相同的对象。
- **对象序列化**（Object Serialization） 通过写出对象的状态数据来记录一个对象。
- **对象序列化的主要任务** 写出对象的状态信息，并遍历该对象对其他对象的引用，递归的序列化所有被引用到的其他对象，从而建立一个完整的序列化流。

13.4.4 实现对象序列化

要序列化一个对象，其所属的类必须实现以下两种接口之一：

- java.io.Serializable
- java.io.Externalizable

java.io.ObjectOutputStream 和 ObjectInputStream 类分别提供了对象的序列化和反序列化功能。

注意

- 在对象序列化过程中，其所属类的 static 属性和方法代码不会被序列化；

- 对于个别不希望被序列化的非 static 属性，也可以在属性声明的时候使用 transient 关键字进行标明。

课程配套代码 ➜ sample.io.serialization

☒ 14 ☒ 线程编程

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第八周

参考教材: 本课程参考教材及资料如下:

- 陈国君主编, Java 程序设计基础 (第 5 版), 清华大学出版社, 2015.5
- Bruce Eckel, Thinking in Java (3rd)

教学目标

1. 线程基础: 理解任务调度、进程和线程, 掌握其联系和区别; 掌握 Java 的线程模型, 以及如何创建线程; 理解后台线程。
2. 线程控制: 理解线程的生命周期, 明白各阶段的含义; 掌握线程控制方法, 理解各线程控制方法对线程状态切换的作用。
3. 线程的同步: 理解临界资源问题, 进一步明白线程安全的意义; 了解关键字 synchronized 的用法; 了解死锁的概念; 通过生产者—消费者问题分析理解线程同步。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

14.1 线程基础

14.1.1 进程

进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用程序运行的载体。进程是一种抽象的概念，从来没有统一的标准定义。进程一般由程序、数据集合和进程控制块三部分组成。程序用于描述进程要完成的功能，是控制进程执行的指令集；数据集合是程序在执行时所需要的数据和工作区；进程控制块 (Program Control Block, 简称 PCB)，包含进程的描述信息和控制信息，是进程存在的唯一标志。

进程具有的特征：

动态性 进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的；

并发性 任何进程都可以同其他进程一起并发执行；

独立性 进程是系统进行资源分配和调度的一个独立单位；

结构性 进程由程序、数据和进程控制块三部分组成。

14.1.2 什么是线程

根据多任务原理，在一个程序内部也可以实现多个任务（顺序控制流）的并发执行，其中每个任务被称为线程 (Thread)。更专业的表述为：**线程是程序内部的顺序控制流**。

在早期的操作系统中并没有线程的概念，进程是能拥有资源和独立运行的最小单位，也是程序执行的最小单位。任务调度采用的是时间片轮转的抢占式调度方式，而进程是任务调度的最小单位，每个进程有各自独立的一块内存，使得各个进程之间内存地址相互隔离。

后来，随着计算机的发展，对 CPU 的要求越来越高，进程之间的切换开销较大，已经无法满足越来越复杂的程序的要求，于是就发明了线程。线程是程序执行中一个单一的顺序控制流程，是程序执行流的最小单元，是处理器调度和分派的基本单位。

一个进程可以有一个或多个线程，各个线程之间共享程序的内存空间（也就是所在进程的内存空间）。一个标准的线程由线程 ID、当前指令指针（PC）、寄存器和堆栈组成。而进程由内存空间（代码、数据、进程空间、打开的文件）和一个或多个线程组成。

线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。线程自己只拥有一点在运行中必不可少的资源（如程序计数器，一组寄存器和栈），但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

14.1.3 线程和进程的区别

进程和线程都是由操作系统所体现的程序运行的基本单元，系统利用该基本单元实现系统对应用的并发性。从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。进程和线程的区别有：

1. 每个进程都有独立的代码和数据空间（进程上下文），进程切换的开销大。
2. 线程作为轻量的“进程”，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换的开销小。
3. 多进程——在操作系统中能同时运行多个任务（程序）。
4. 多线程——在同一应用程序中有多个顺序流同时执行。

14.1.4 线程的概念模型

在 Java 语言中，多线程机制通过虚拟 CPU 来实现。

1. 虚拟的 CPU，由 `java.lang.Thread` 类封装和虚拟；
2. 虚拟 CPU 所执行的代码，传递给 `Thread` 类对象；
3. 虚拟 CPU 所处理的数据，传递给 `Thread` 类代码对象。

14.1.5 创建线程

Java 的线程是通过 `java.lang.Thread` 类来实现的。每个线程都是通过某个特定 `Thread` 对象所对应的方法 `run()` 来完成其操作的，方法 `run()` 称为线程体。

以下给出一个创建线程的示例：

示例代码：[TestThread1.java](#)

```
1 public class TestThread1 {  
2     public static void main(String args[]) {  
3         Runner1 r = new Runner1();  
4         Thread t = new Thread(r);  
5         t.start();  
6     }  
7 }  
  
9 class Runner1 implements Runnable {  
10    public void run() {  
11        for(int i=0; i<30; i++) {  
12            System.out.println("No. " + i);  
13        }  
14    }  
15 }
```

参考以上代码，给出创建和启动线程的一般步骤：

1. 定义一个类实现 Runnable 接口，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建 Runnable 接口实现类的对象；
3. 创建 Thread 类的对象（封装 Runnable 接口实现类型对象）；
4. 调用 Thread 对象的 start() 方法，启动线程。

14.1.6 多线程的目标

Java 中引入线程机制的目的在于实现**多线程**（Multi-Thread）并发的任务执行。以下代码基于同一个线程体创建并运行两个线程，基于线程体共享和主线程初始对象共享，多线程之间可以共享代码和数据。

线程	虚拟 CPU	代码	数据
t1	Thread 类对象	Runner2 类中的 run() 方法	Runnable 类型对象 r
t2	Thread 类对象	Runner2 类中的 run() 方法	Runnable 类型对象 r

示例代码：[TestThread2.java](#)

```
1 public class TestThread2 {  
2     public static void main(String args[]) {  
3         Runner2 r = new Runner2();  
4         Thread t1 = new Thread(r);  
5         Thread t2 = new Thread(r);  
6         t1.start();  
7         t2.start();  
8     }  
9 }  
  
11 class Runner2 implements Runnable {  
12     public void run() {  
13         for (int i=0; i<20; i++) {  
14             String s = Thread.currentThread().getName(); //获取当前运行中的线程对象  
15             System.out.println(s + ":" + i);  
16         }  
17     }  
18 }
```

14.1.7 创建线程的第二种方式

可以直接继承 Thread 类创建线程。

1. 定义一个类继承 Thread 类，重写其中的 run() 方法，加入所需的处理逻辑；
2. 创建该 Thread 类的对象；
3. 调用该对象的 start() 方法。

以下给出示例代码：

示例代码：[TestThread3.java](#)

```
1 public class TestThread3 {  
2     public static void main(String args[]) {  
3         Thread t = new Runner3();  
4         t.start();  
5     }  
6 }
```

```
6  }
7
8 class Runner3 extends Thread {
9     public void run() {
10        for (int i=0; i<30; i++) {
11            System.out.println("No. " + i);
12        }
13    }
14 }
```

14.1.8 两种创建线程的方式比较

使用 Runnable 接口创建线程

- 可以将虚拟 CPU、代码和数据分开，形成清晰的模型；
- 线程体 run() 方法所在的类还可以从其他类继承一些有用的属性或方法；
- 有利于保持程序风格的一致性。

直接继承 Thread 类创建线程

- Thread 子类无法再从其他类继承；
- 编写简单，run() 方法的当前对象就是线程对象，可直接操作。

14.1.9 后台线程

相关概念

后台处理 也称为后台运行，是指在分时处理或多任务系统中，当实时、会话式、高优先级或需迅速响应的计算机程序不再使用系统资源时，计算机去执行较低优先级程序的过程。批量处理、文件打印通常采取后台处理的形式。

后台线程 是指那些在后台运行的，为其他线程提供服务的功能，如 JVM 的垃圾回收线程等，后台线程也称为守护线程（Daemon Thread）。

用户线程 和后台线程相对应，其他完成用户任务的线程可称为“用户线程”。

Thread 类提供的与后台线程相关的方法包括：

1. 测试当前线程是否为守护线程，如果是则返回 true，否则返回 false。

```
1 public final boolean isDaemon();
```

2. 将当前线程标记为守护线程或用户线程，本方法必须在启动线程前调用。

```
1 public final void setDaemon(Boolean on);
```

以下给出使用后台线程的示例：

[示例代码：TestDaemonThread.java](#)

```
1 public class TestDaemonThread {  
2     public static void main(String[] args) {  
3         Thread t1 = new MyRunner(10);  
4         t1.setName("用户线程t1");  
5         t1.start();  
6         Thread t2 = new MyRunner(10000);  
7         t2.setDaemon(true);  
8         t2.setName("后台线程t2");  
9         t2.start();  
10        for (int i = 0; i < 10; i++) {  
11            System.out.println(Thread.currentThread().getName() + ":" + i);  
12        }  
13        System.out.println("主线程结束");  
14    }  
15}  
  
16 class MyRunner extends Thread {  
17     private int n;  
18     public MyRunner(int n) {  
19         this.n = n;  
20     }  
21     public void run() {  
22         for (int i = 0; i < n; i++) {  
23             System.out.println(this.getName() + ":" + i);  
24         }  
25         System.out.println(this.getName() + "结束");  
26     }  
27}
```

```
27 }  
28 }
```

对上述代码的说明

后台线程线程 t2 并没有如预期的输出数字 0-9999，而是提前终止。这是因为，待用户线程（这里包括主线程和线程 t1）全部运行结束后，JVM 检测到只剩下后台线程在运行的时候，就退出了当前应用程序的运行。

请将上述代码中的“t2.setDaemon(true);”注释后，编译运行程序进行比较运行结果。

14.1.10 GUI 线程

GUI 程序运行过程中，系统会自动创建若干个 GUI 线程以提供所需的功能，主要包括**窗体显示和重绘、GUI 事件处理、关闭抽象窗口工具集等**。

AWT-Windows 线程 负责从操作系统获取底层事件通知，并将之发送到系统事件队列（EventQueue）等待处理。在其他平台上运行时，此线程的名字也会作相应变化，例如在 Unix 系统则为“AWT-Unix”。

AWT-EventQueue-n 线程 也称事件分派线程，该线程负责从事件队列中获取事件，将之分派到相应的 GUI 组件（事件源）上，进而触发各种 GUI 事件处理对象，并将之传递给相应的事件监听器进行处理。

AWT-Shutdown 线程 负责关闭已启用的抽象窗口工具，释放其所占用的资源，该线程将等到其他 GUI 线程均退出后才开始其清理工作。

DestroyJavaVM 线程 在所有其他用户线程退出后，负责释放任意线程所占用系统资源并卸载 Java 虚拟机。该线程在主线程运行结束时由系统自动启动，但要等到所有其他用户线程均退出后才开始其卸载工作。

以下给出测试 GUI 线程的示例：

示例代码：TestGUIThread.java

```
1 import java.awt.*;  
2 import java.awt.event.*;
```

```
4 public class TestGUIThread {  
5     public static void main(String[] args) throws Exception {  
6         Frame f = new Frame();  
7         Button b = new Button("Press Me");  
8         MyMonitor mm = new MyMonitor();  
9         b.addActionListener(mm);  
10        f.addWindowListener(mm);  
11        f.add(b, "Center");  
12        f.setSize(100, 60);  
13        f.setVisible(true);  
14        MyThreadViewer.view();  
15    }  
16 }  
  
18 class MyMonitor extends WindowAdapter implements ActionListener {  
19     public void actionPerformed(ActionEvent e) {  
20         MyThreadViewer.view();  
21     }  
22 }  
  
24 class MyThreadViewer {  
25     public static void view() {  
26         Thread current = Thread.currentThread();  
27         System.out.println("当前线程名称: " + current.getName());  
28         int total = Thread.activeCount();  
29         System.out.println("活动线程总数: " + total + "个");  
30         Thread[] threads = new Thread[total];  
31         current.enumerate(threads);  
32         for (Thread t: threads) {  
33             String role = t.isDaemon() ? "后台线程" : "用户线程";  
34             System.out.println(" -" + role + t.getName());  
35         }  
36         System.out.println(" ----- ");  
37     }  
38 }
```

14.2 线程控制

14.2.1 线程的生命周期

线程的生命周期包含以下 5 个状态（状态间关系如图14.1所示）：

新建状态 调用 Thread 构造方法，未显式调用 start() 方法前；

就绪状态 调用 start() 方法后，线程在就绪队列里等候；

运行状态 开始执行线程体代码；

阻塞状态 因某事件发生，例如线程进行 I/O 操作，等待用户输入数据；

终止状态 线程 run() 方法执行完毕。

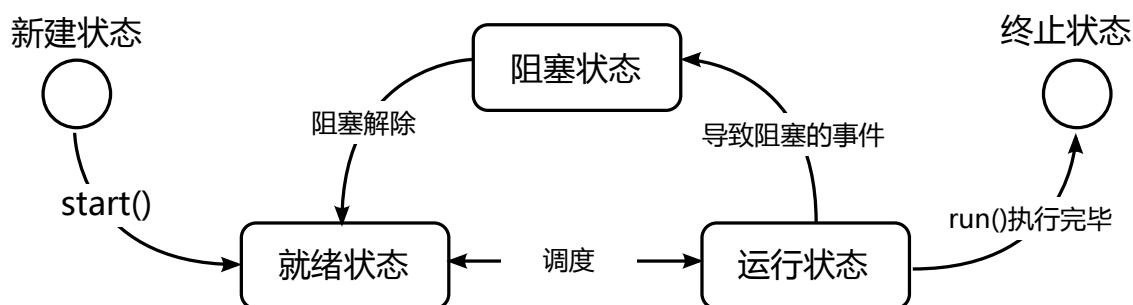


图 14.1 Java 线程生命周期状态

14.2.2 线程优先级

线程的优先级用数字来表示，范围从 1 到 10。主线程的缺省优先级是 5，子线程的优先级默认与其父线程相同。可以使用 Thread 类的下述方法获得和设置线程的优先级。

- 获取当前线程优先级

```
1 public final int getPriority();
```

- 设定当前线程优先级

```
1 public final void setPriority (int newPriority);
```

相关静态整型常量如下：

```
1 Thread.MIN_PRIORITY = 1  
2 Thread.MAX_PRIORITY = 10  
3 Thread.NORM_PRIORITY = 5
```

14.2.3 线程串行化

在多线程程序中，如果在一个线程运行的过程中要用到另一个线程的运行结果，则可进行线程的串型化处理。

Thread 类提供的线程串行化相关方法包括：

```
1 public final void join();  
2 public final void join(long millis);  
3 public final void join(long millis, int nanos);
```

以下给出实现线程串行化的代码示例：

示例代码：[TestJoin.java](#)

```
1 public class TestJoin {  
2     public static void main(String[] args) {  
3         MyRunner r = new MyRunner();  
4         Thread t = new Thread(r);  
5         t.start();  
6         try {  
7             t.join();  
8         } catch(InterruptedException e) {  
9             e.printStackTrace();  
10        }  
11        for(int i = 0; i < 50; i++) {  
12            System.out.println("主线程: " + i);  
13        }  
14    }  
15 }  
16  
17 class MyRunner implements Runnable {  
18     public void run() {  
.....
```

```
19     for (int i = 0; i < 50; i++) {  
20         System.out.println("子线程: " + i);  
21     }  
22 }
```

说明

上述程序中，主线程在执行过程中调用了线程 t 的 join() 方法，该方法导致当前线程（主线程）阻塞，直到线程 t 运行终止后，主线程才会获得继续执行的机会。这相当于将线程 t 串行加入到主线程中。

14.2.4 线程休眠

线程休眠，即暂停执行当前运行中的线程，使之进入阻塞状态，待经过指定的“延迟时间”后再醒来并转入到就绪状态。

Thread 类提供的线程休眠相关方法包括：

```
1 public static void sleep(long millis);  
2 public static void sleep(long millis, int nanos);
```

以下给出使用线程休眠实现的数字计数器示例：

示例代码：Digitaltimer.java

```
1 import java.util.Calendar;  
2 import java.util.GregorianCalendar;  
3 import javax.swing.*;  
4  
5 public class DigitalClock {  
6     public static void main(String[] args) {  
7         JFrame jf = new JFrame("Clock");  
8         JLabel clock = new JLabel("clock");  
9         clock.setHorizontalAlignment(JLabel.CENTER);  
10        jf.add(clock, "Center");  
11        jf.setSize(140, 80);  
12        jf.setLocation(500, 300);  
13        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
.....
```

```
14    jf . setVisible ( true );
15    Thread t = new MyThread(clock);
16    t . start ();
17 }
18 }

20 class MyThread extends Thread {
21     private JLabel clock;
22     private int i;
23     public MyThread(JLabel clock) {
24         this . clock = clock;
25         this . i = 1;
26     }
27     public void run() {
28         while(true) {
29             clock . setText(String . valueOf(i++));
30             try {
31                 Thread.sleep(1000);
32             } catch(InterruptedException e) {
33                 e . printStackTrace();
34             }
35         }
36     }
37 }
```

14.2.5 线程让步

线程让步，让运行中的线程主动放弃当前获得的 CPU 处理机会，但不是使该线程阻塞，而是使之转入就绪状态。Thread 类提供的线程让步相关方法：

```
public static void yield();
```

以下给出一个线程让步的示例：

示例代码：[TestYield.java](#)

```
import java. util . Date;
```

```
3 public class TestYield {  
4     public static void main(String[] args) {  
5         Thread t1 = new MyThread(false);  
6         Thread t2 = new MyThread(true);  
7         Thread t3 = new MyThread(false);  
8         t1.start();  
9         t2.start();  
10        t3.start();  
11    }  
12 }  
  
14 class MyThread extends Thread {  
15     private boolean flag;  
16     public MyThread(boolean flag) {  
17         this.flag = flag;  
18     }  
19     public void setFlag(boolean flag) {  
20         this.flag = flag;  
21     }  
22     public void run() {  
23         long start = new Date().getTime();  
24         for(int i = 0; i < 200; i++) {  
25             if(flag) {  
26                 Thread.yield();  
27             }  
28             System.out.println(this.getName() + ":" + i + "\t");  
29         }  
30         long end = new Date().getTime();  
31         System.out.println("\n" + this.getName() + "执行时间: " + (end - start) + "ms");  
32     }  
33 }
```

从执行结果来看，由于设置了线程让步，第二个线程明显执行时间长。调用 `yield()` 方法只是令当前线程主动在时间片到期前使其他线程获得运行机会。

14.2.6 线程挂起与恢复

线程挂起 暂时停止当前运行中的线程，使之转入阻塞状态，并且不会自动恢复运行。

线程恢复 使得一个已挂起的线程恢复运行。

Thread 类提供的线程挂起与恢复的相关方法：

```
1 public final void suspend();
2 public final void resume();
```

suspend() 和 resume() 方法已经不提倡使用，原因是 suspend() 方法挂起线程时并不释放其锁定的资源，这可能会影响到其他线程的执行，且容易导致线程死锁。

14.2.7 线程等待与通知

将运行中的线程转为阻塞状态的另外一种途径是：调用该线程中被锁定资源（Java 对象）的 wait() 方法，该方法在 Object 类中定义，其功能是让当前线程等待，直到有其他线程调用了同一个对象的 notify() 或 notifyAll() 方法通知其结束等待，或是经历了约定的等待时间后，等待线程才会醒来，重新进入可执行状态。

线程等待与线程挂起比较：

- 线程挂起时不会释放所占用的资源；
- 线程等待时则会释放资源，以使其他线程获得运行机会。

14.3 线程的同步

14.3.1 临界资源问题

两个线程 A 和 B 在同时操纵 Stack 类的同一个实例（栈），A 向栈里 push 一个数据，B 要从堆栈中 pop 一个数据。

◆ 代码

```
1 public class Stack {
2     int idx = 0;
3     char[ ] data = new char[6];
```

```
5  public void push(char c) {  
6      data[idx] = c;  
7      idx++;  
8  }  
9  public char pop() {  
10     idx--;  
11     return data[idx];  
12  }  
13 }
```

❖ 问题分析

- 操作之前，假设 $\text{data} = |a|b| |||$, $\text{idx} = 2$;
- 线程 A 执行 push 中的第一个语句，将 c 推入堆栈; $\text{data} = |a|b|c| |||$, $\text{idx} = 2$;
- 线程 A 还未执行 $\text{idx}++$ 语句，A 的执行被线程 B 中断，B 执行 pop 方法， $\text{data} = |a|b|c| |||$ $\text{idx} = 1$;
- 线程 A 继续执行 push 的第二个语句: $\text{data} = |a|b|c| |||$, $\text{idx} = 2$;
- 最后的结果相当于 c 没有入栈，产生这种问题的原因在于对共享数据访问的操作的不完整性。

14.3.2 什么是临界资源

在并发程序设计中，对多线程共享的资源或数据称为**临界资源**（或同步资源），而把每个线程中访问临界资源的那一段代码称为**临界代码**（或临界区）。

- 在一个时刻只能被一个线程访问的资源就是临界资源。
- 访问临界资源的那段代码就是临界区，**临界区必须互斥地使用**。

14.3.3 互斥锁

- Java 引入对象**互斥锁**机制来实现线程的互斥操作，保证共享数据操作的完整性。
- Java 中每个对象都有一个互斥锁与之相连，用来保证在任一时刻，只能有一个线程访问该对象。
- 多线程对临界资源的并发访问是通过竞争互斥锁实现的。

14.3.4 synchronized 的用法

为了保证互斥，Java 语言使用 **synchronized** 关键字标识同步的 **资源**，包括：

- 对象

```
1 synchronized(对象) {  
2     临界代码段  
3 }
```

- 方法

```
1 public synchronized 返回值类型 方法名 {  
2     方法体  
3 }
```

等效方式：

```
1 public 返回值类型 方法名 {  
2     synchronized(this) {  
3         方法体  
4     }  
5 }
```

- 语句块（一段代码），用法同方法的等效方式

14.3.5 synchronized 的用法示例

❖ 用于方法声明中，标明整个方法为同步方法

```
1 public synchronized void push(char c) {  
2     data[idx] = c;  
3     idx++;  
4 }
```

❖ 用于修饰语句块，标明整个语句块为同步块

```
1 // Other code
```

```
2 public char pop() {  
3     synchronized(this) {  
4         idx--;  
5         return data[idx];  
6     }  
7     // Other code  
8 }
```

14.3.6 synchronized 的功能

- 首先判断对象或者方法的**互斥锁**是否存在，若存在就获得互斥锁，然后执行紧随其后的临界代码段或方法体；
- 如果对象或方法的互斥锁不在（已经被其他线程拿走），就进入线程**等待状态**，直到获得互斥锁。

课程配套代码 ➔ sample.thread.syn.WithdrawMoneyFromBankSample.java

14.3.7 线程死锁

并发运行的多个线程间彼此等待、都无法运行的状态称为**线程死锁**。

为避免死锁，在线程进入阻塞状态时应尽量释放其锁定的资源，以为其他的线程提供运行的机会。

❖ 相关方法

- public final void wait()
- public final void notify()
- public final void notifyAll()

14.3.8 线程间通信

通过线程间的**对话**来解决线程间的同步问题。

❖ 线程间通信的有效手段

wait() 如果一个正在执行同步代码（synchronized）的线程 A 执行了 wait() 调用（在对象 x

上), 该线程暂停执行而进入对象 x 的等待队列, 并释放已获得的对象 x 的互斥锁。线程 A 要一直等到其他线程在对象 x 上调用 notify() 或 notifyAll() 方法, 才能重新获得对象 x 的互斥锁后继续执行 (从 wait() 语句后继续执行)。

notify() 唤醒正在等待该对象互斥锁的第一个线程。

notifyAll() 唤醒正在等待该对象互斥锁的所有线程, 具有最高优先级的线程首先被唤醒并执行。

14.3.9 Object.wait() 和 notify()

- wait() 和 notify() 只能在同步代码块中调用。
- wait() 在放弃 CPU 资源的同时交出了对资源的控制权。

14.3.10 Thread.sleep() 与 Object.wait()、notify() 的区别

- 所属对象不同。sleep() 是 Thread 类的方法, 而 wait(), notify(), notifyAll() 是 Object 类中定义的方法, 都会影响线程的执行行为。
- 锁行为不同。Thread.sleep() 不会导致锁行为的改变, 如果当前线程是拥有锁的, 那么 Thread.sleep() 不会让线程释放锁。可以简单认为和锁相关的方法都定义在 Object 类中, 因此调用 Thread.sleep() 不会影响锁的相关行为。
- 线程恢复方式不同。Thread.sleep 和 Object.wait 都会暂停当前的线程, 即表示它暂时不再需要 CPU 的执行时间。区别是调用 wait 后, 需要别的线程执行 notify/notifyAll 才能够重新获得 CPU 执行时间。

14.3.11 生产者—消费者问题

生产者消费者模型 (如图14.2) 就是在一个系统中存在**生产者**和**消费者**两种角色, 他们之间通过**内存缓冲区**进行通信, 生产者生产消费者需要的资料, 消费者把资料做成产品。

❖ 生产者消费者问题是线程模型中的经典问题

- 生产者和消费者在同一时间段内共用同一存储空间, 生产者向空间里生产数据, 而消费者取走数据。

- 阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者解耦的。

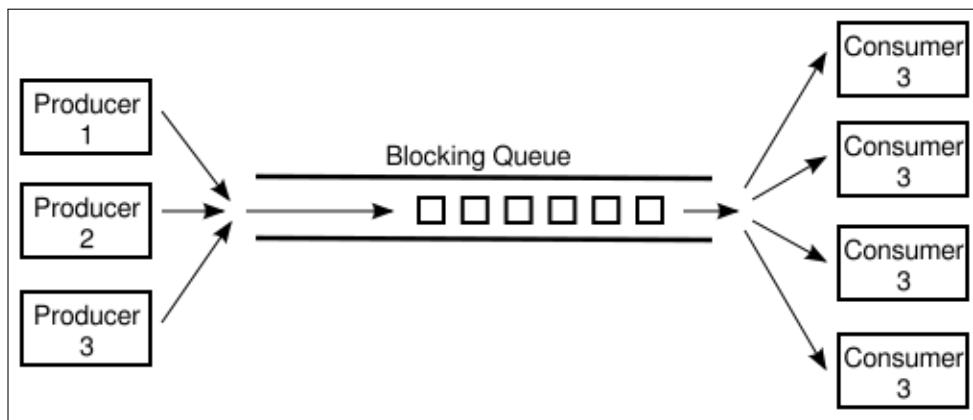


图 14.2 生产者消费者问题

课程配套代码 [sample.thread.syn.ProducerAndConsumer.java](#)

☒ 15 ☒ Java EE 体系结构

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第九周

参考教材: 本课程参考教材及资料如下:

- 吕海东, 张坤编著, Java EE 企业级应用开发实例教程, 清华大学出版社,
2010 年 8 月

教学目标

1. 了解软件开发的现状与发展趋势, 了解企业级应用的特点
2. 掌握 Java EE 的概念和规范, 掌握 Java EE 容器、组件和通信协议的类型和功能

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

15.1 软件开发的现状

15.1.1 软件开发现状

面向 Internet 开发企业级 Web 应用

面向对象 OOA/OOD/OOP, Java、C#

面向组件 软件系统是由许多小的组件构建和装配起来的

采用标准规范开发 J2EE, MS.NET

全面采用框架技术 Struts、Spring、Hibernate、AJAX、WebWork

软件系统采用分层结构和设计模式 MVC

工厂化流水线开发模式 CVS

可视化软件建模 UML、RUP、ROSE

15.1.2 企业级应用的特点

分布式 通过局域网或 Internet 连接分布在一个组织内部或世界各地的部门及用户。

高速反应性 企业组织需要不断地改变业务规则来适应业务需求或商业模式的不断变化。

高安全性 企业应用系统必须保证运行的高度安全性和可靠性。

可扩展性 要求软件架构具备灵活的可扩展能力和伸缩性，满足信息资源及用户群体的不断发展。

集成化 必须尽可能的集成已有的遗留系统，最大限度的利用信息资源。

15.2 Java EE 概述

15.2.1 什么是 Java EE

- Java EE 是基于 Java SE 标准版基础上的一组开发**以服务器为中心的企业级应用**的技术和规范。

- 用于规范化、标准化以 Java 为开发语言的企业级软件的开发、部署和管理。
- 达到减少开发费用、降低软件复杂性和快速交付的目的。

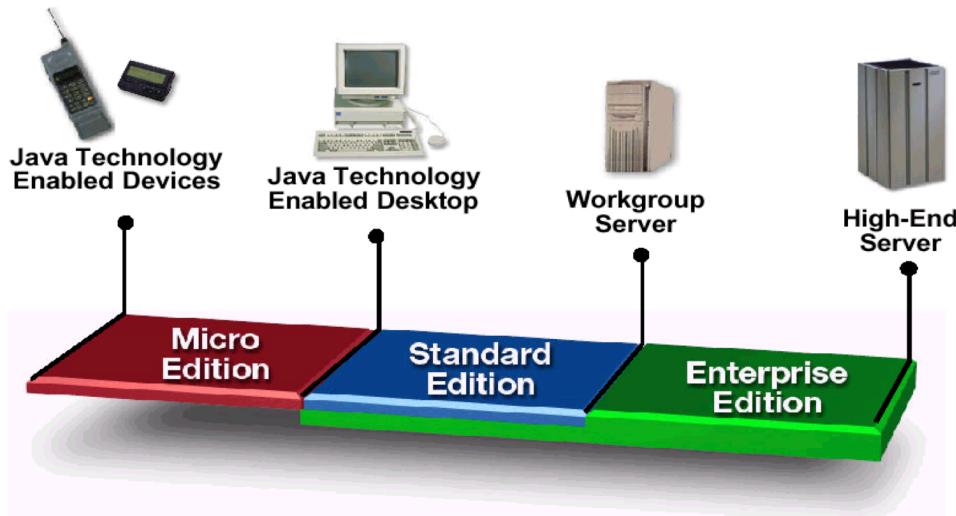


图 15.1 Java 的三个平台版本

15.2.2 Java EE 规范

Java EE 规范定义了面向 Internet 的企业级软件应用的组成部分和各组成部分之间的交互协议。

容器规范 容器（Container）是组件的运行环境，负责组件的生命周期管理和调用。

组件规范 组件（Component）是 Java EE 应用的标准化部件，完成系统的业务和逻辑功能，在 Java EE 应用中组件运行在容器内，由容器管理组件的创建、调用和销毁整个生命周期。在 Java EE 应用中组件之间是不能直接调用的，必须通过容器完成。

服务规范 Java EE 规定了连接各种外部资源的标准接口 API，简化了连接各种不同类型外部资源的设计和编程。如 JDBC API 提供了连接数据库的标准接口；JMS API 可以连接各种外部的消息服务系统。

通信协议规范 Java EE 规范使用目前市场上主流的通信协议 HTTP、HTTPS 等，改进了与其他平台的互操作性。

开发角色规范 Java EE 分别定义了 7 种不同的角色合作进行应用系统的开发，确保系统开发高效而有序，提高软件的成功率。

15.3 Java EE 容器

◆ 容器的功能

- 容器是运行**组件**的环境对象，提供了组件运行所需要的服务，并管理组件的生成、调用和销毁整个生命周期。
- 在 Java EE 规范下，所有 Java EE 组件都由容器来创建和销毁。

容器的优势

- 简化了企业级软件开发中复杂的对象管理事务；
- 克服了 C++ 语言等内存泄漏缺陷；
- 减轻软件开发人员的负担。

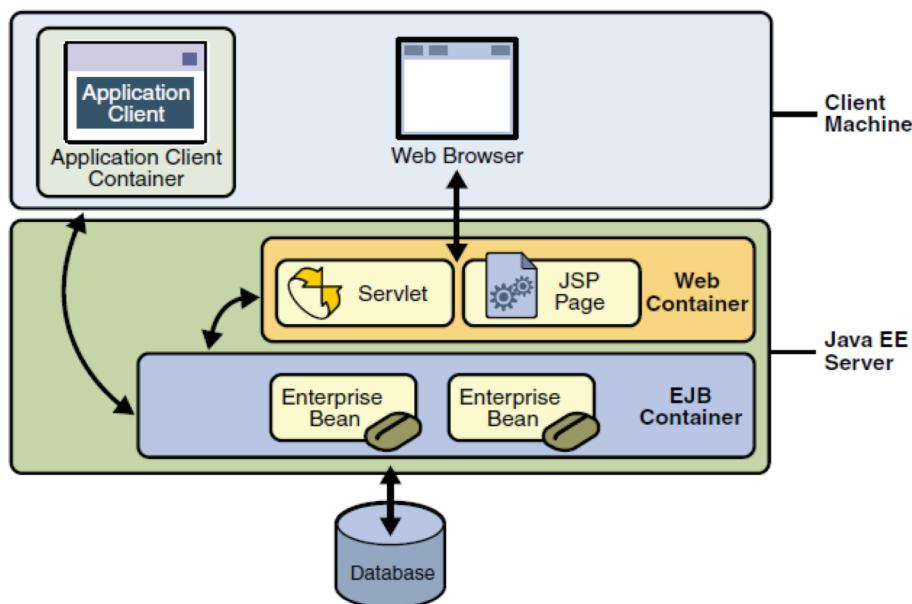


图 15.2 Java EE 容器

15.3.1 客户端应用容器

- 客户端应用容器（Application Client Container）即是普通 Java SE 的 JVM，管理并运行客户 JavaBean 组件，与一般的 Java 类没有区别。
- Java EE 规范将客户端应用容器纳入自己的管理范围之内，进行统一的约定。

15.3.2 Applet 容器

- Applet 容器（Applet Container）是具有 Java SE Plugin 插件的 Web 浏览器，驻留在客户端，管理和运行 Java Applet 组件。
- Applet 容器使得 Web 具有丰富的图形界面和事件响应机制，进而开发出具有极高交互性的 Web 应用软件。

15.3.3 Web 容器

- Web 容器（Web Container）运行在符合 Java EE 规范的应用服务器上，驻留在服务器端，外部应用可以通过**HTTP 和 HTTPS**协议与 Web 容器通信，进而访问 Web 容器管理的 Web 组件。
- Web 容器管理 Web 组件的运行和调用。Java EE 定义了两种 Web 组件：**Servlet 和 JSP**，可以产生动态 Web 内容，结合**数据库技术**，用于动态 Web 应用的开发。

15.3.4 企业 JavaBean 容器

- EJB 容器（EJB Container）用于管理企业级 JavaBean 对象的生命周期和方法调用。Java EE 规范定义了 3 种运行在 EJB 容器内的组件：**会话 EJB、消息驱动 EJB 和实体 EJB**，分别完成不同领域的业务处理。
- EJB 容器运行在符合 Java EE 的应用服务器内，驻留在服务器端。
- 其他组件通过 RMI/IOP 协议与 EJB 容器通信，通过 EJB 容器来访问 EJB 组件的业务方法。

EJB 主要应用于重量级企业应用系统开发，在以 Web 服务为主的企业业务系统中，可以选择轻量级组件替代 EJB。

15.4 Java EE 组件

15.4.1 Java EE 组件（Component）

- Java EE 规范约定组成企业级软件系统的组成单元是**组件**。
- 组件使用特定的配置信息部署在符合 Java EE 规范的服务器容器中运行，并与其他组件组装在一起，组成整个 Java EE 应用系统。

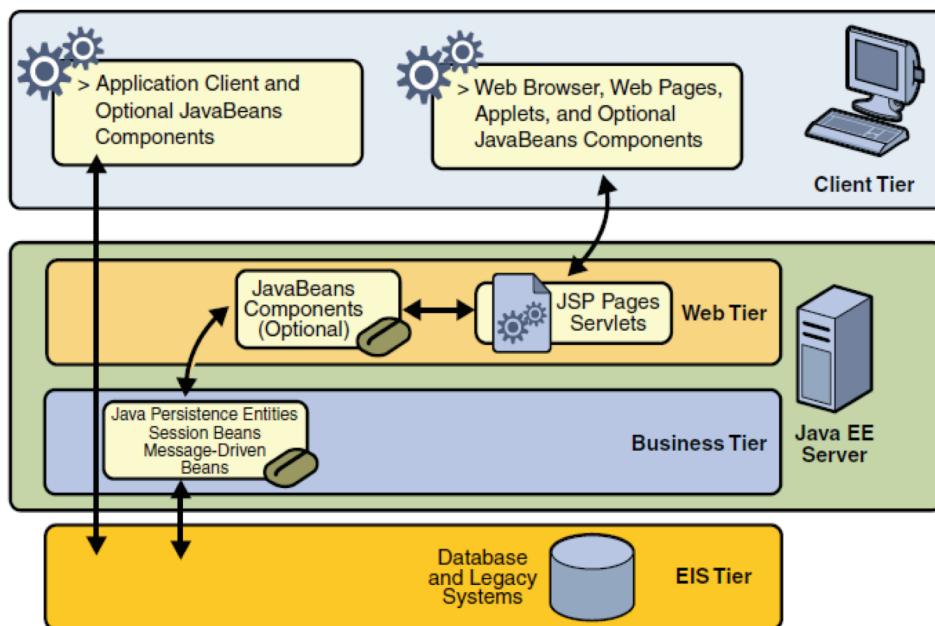


图 15.3 Java EE 组件

15.4.2 Java EE 组件列表

1. Application Client Component
2. Applet Component
3. **Web Component ***
 - Servlet
 - JSP
4. **EJB Component ***
 - Session Bean
 - Entity Bean
 - Message Driven Bean

15.4.3 客户端组件

- 客户端组件即 JavaBean 类，基于 Java SE 平台，运行在客户端容器内，有独立的 JVM 空间。
- 客户端组件一般用富客户端图形界面显示，图形框架用 Swing 开发，可以远程调

用 Web 组件和 EJB 组件。

15.4.4 Applet 组件

- Applet 组件采用 Java Applet 框架技术开发，运行在 Applet 容器，即客户端 Web 浏览器，需要有 Java SE 的插件支持。
- 目前客户端 JavaBean 组件和 Applet 组件已经逐渐被 RIA（富互联网应用）技术取代，不推荐在企业级应用中使用客户端组件和 Applet 组件。

15.4.5 Web 组件 *

Web 组件在近十几年的互联网应用中得到广泛应用，一度成为 Java EE 的核心。

- Web 组件运行在服务器端的 Web 容器内，能接收 HTTP 请求并进行处理，产生动态 Web 响应。
- 近年来，随着开发人员发现 Web 组件开发过于繁琐和细化，在 Web 组件基础上发布了各种用于简化 Web 组件开发的框架和技术，其中最著名的就是[Struts](#)、[Spring Web MVC](#)、[JSF](#)等，都是对标准 Web 组件的扩展和更新。

15.4.6 EJB 组件 *

- EJB 组件运行在符合 Java EE 的应用服务器内，驻留在服务器端。Java EE 的其他组件，包括 EJB 组件通过 RMI/IIOP 协议与 EJB 容器通信，远程调用 EJB 的功能方法。
- Java EE 5.0 之前，EJB 性能差，饱受诟病。Rod Johnson¹针对 EJB 的缺点，开发了轻量级的企业组件管理技术 Spring，[可以使用普通的 JavaBean 组件完全取代 EJB 组件](#)。
- Java EE 5.0 之后，Sun 公司全面引入 Spring 框架思想和 Java SE 5.0 的[注解编程技术](#)，推出了 EJB 3.0 组件规范。从而确立了 EJB 在大型企业软件项目开发中的地位。

¹Spring Framework 创始人，著名作者。

15.5 Java EE 服务 API

15.5.1 Java EE 服务 API

Java EE 提供了标准化的服务接口 API 来统一各种外部资源的访问和控制。Java EE 提供的标准化服务接口主要包含以下方面：

- 数据库连接服务 API-JDBC
- 消息服务连接服务 API-JMS
- 数据持久化服务 API-JPA
- 命名和目录服务 API-JNDI
- 安全性和授权服务 API-JAAS
- 电子邮件服务 API-JavaMail
- 事务服务 API-JTA
- XML 处理服务 API-JAXP
- XML Web 服务 API-JAX-WS
- XML 绑定服务 API-JAXB
- 带附件的 SOAP 服务 API-SAAJ
- XML Web 服务注册 API-JAXR
- 与其他遗留系统交互服务 API-J2EE Connector Architecture

15.6 组件间通信协议

15.6.1 组件间通信协议

Java EE 组件运行在 Java EE 容器内，组件之间不允许直接取得对象引用和直接调用（**隔离性**），只能使用**规定的通信协议**与组件所在的容器进行通信并请求目标组件。

15.6.2 HTTP 和 HTTPS

❖ HTTP

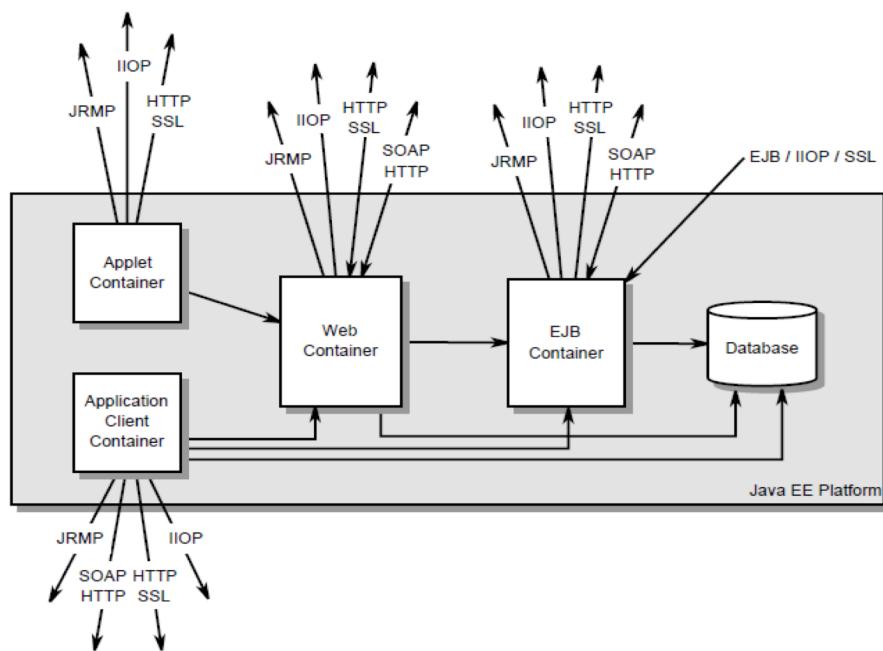


图 15.4 Java EE 组件间通信协议

Java EE 规范继续使用 **HTTP** 作为与 Web 容器通信的标准协议，延续 Web 应用的标准化，使访问以下资源都使用相同的 HTTP 协议：

- 静态 HTML 页面
- 访问 Java EE 的 Web 组件 Servlet 和 JSP

❖ HTTPS

HTTP 的加密（SSL）版本。

15.6.3 RMI 和 RMI-IIOP

❖ RMI

RMI (Remote Method Invocation)，远程方法调用大大增强了 Java 开发分布式应用的能力，可以被认为是远程过程调用的 Java 版本。Java EE 的 EJB 容器使用 RMI 协议进行通信。

❖ RMI-IIOP

RMI-IIOP (Java Romote Method Invocation Over the Internet Inter-ORB Portocol) 是 RMI 功能扩展版本，增加了如分布式垃圾收集和可下载类文件等，目前 Java EE 应用

中与 EJB 容器和组件通信都是用 RIM-IIOP。

15.6.4 SOAP

SOAP (Simple Object Access Protocol) 是一种标准化的通信规范，主要用于与 Web Services 交互调用。SOAP 以 XML 格式交换数据，与编程语言、平台和硬件无关。²

²SOAP 1.2 是业界共同的标准，属于第二代的 XML 协议（第一代主要为 XML-RPC 以及 WDDX 技术）。

☒ 16 ☒ Servlet 编程

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第九周

参考教材: 本课程参考教材及资料如下:

- 吕海东, 张坤编著, Java EE 企业级应用开发实例教程, 清华大学出版社,
2010 年 8 月

教学目标

1. 理解 Web 的概念及工作模式, 掌握 Java Web 应用的构成。
2. 掌握 Servlet 的概念、体系结构及生命周期管理基本原理。
3. 掌握 Servlet 的编程及配置方法, 了解 Servlet 的在 Tomcat 服务器上的部署方式
(war)。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

JSP 和 JSF 都是建立在 Servlet 基础之上的，其他 Web 框架如 Struts、WebWork 和 Spring MVC 都是基于 Servlet。所有很有必要对 Servlet 编程技术进行更为深入的研究，这是我们未来学习框架技术和形成自己框架软件的基础。

16.1 Web 基础

16.1.1 什么是 Web

- Web 本质上就是 Internet 上所有文档（资源）的集合，如 HTML 网页、CSS、JS、图片、动态网页、声音、视频等。
- Web 文档保存在 Web 站点上，Web 站点驻留在 Web 服务器上。
- 常见 Web 服务器有 Apache、IIS、WebLogic、GlassFish、JBoss 和 Tomcat 等。

Web 文档都有唯一的地址，通过 URL 来进行定位：

协议://IP 地址:端口/站点名/目录/文件名

```
1 http://210.30.108.30:8080/jyerm/admin/login.jsp  
2 ftp://210.30.108.30/software/jdk.zip
```

16.1.2 Web 工作模式

Web 使用请求/响应模式进行工作，Web 服务器不会主动将 Web 文档发送到客户端。

1. 由客户（一般是浏览器）使用 URL 对 Web 文档进行请求；
 2. Web 服务器接收并处理请求；
 3. 处理结束后将响应内容发送到客户。
- Web 请求方式主要有**GET、POST、PUT、DELETE 和 HEAD**。
 - Web 响应一般情况下是 HTML 文档，也可以是其他类型资源。
 - Web 使用 MIME (Multipurpose Internet mail Extensions) 标准来确定具体的响应类型。HTTP 响应总体上分为两类：文本类型（纯文本字符、HTML、XML）和二

进制原始类型（图片、声音、视频）。

16.1.3 Java Web 应用的构成

- HTML 文档
- CSS
- JavaScript
- 图片文件
- Servlet
- JSP
- JavaBean 类
- Java Lib
- Web 配置文件：/WEB-INF/web.xml

演示

在 Eclipse 中创建一个 Java Dynamic Project。

16.2 Servlet 概述

16.2.1 Servlet 概述

什么是 Servlet

- Servlet 是一种 Java Class，它运行在 Java EE 的 Web 容器内，由 Web 容器负责它的对象的创建和销毁，不能直接由其它类对象来调用。
- 当 Web 容器接收到对它的 HTTP 请求时，自动创建 Servlet 对象，并自动调用它的 doPost 或 doGet 方法。

Servlet 的主要功能

- 接收用户 HTTP 请求。
- 取得 HTTP 请求提交的数据。
- 调用 JavaBean 对象的方法。
- 生成 HTML 类型或非 HTML 类型的 HTTP 动态响应。
- 实现其他 Web 组件的跳转，包括重定向和转发。

与 Servlet 相近的技术

- CGI (Common Gateway Interface)。
- MS 的 HTTP DLL 技术。
- Perl 语言编写的处理代码。

Servlet 的特点

- 使用 Java 语言编写。
- 可以运行在符合 J2EE 规范的所有应用服务器上，实现跨平台运行。
- 单进程、多线程技术，运行速度快，节省服务器资源。

16.2.2 Servlet 体系结构

- javax.servlet 包含支持所有协议的通用的 Web 组件接口和类；
- javax.servlet.http 包含了支持 HTTP 协议的接口和类。

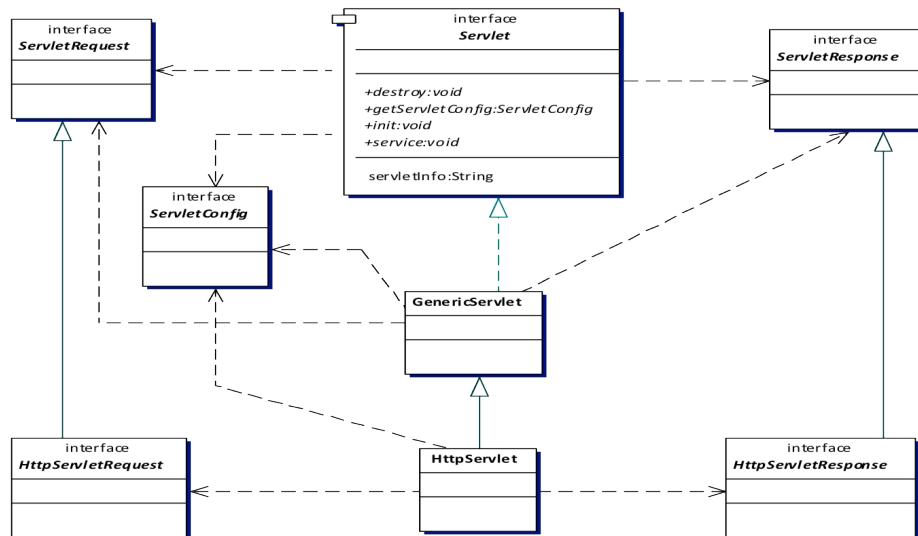


图 16.1 Servlet 类结构

16.3 Servlet 编程

16.3.1 引入包

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
```

16.3.2 类定义

编写接收 HTTP 请求并进行 HTTP 响应的 Servlet 需要继承 `javax.servlet.http.HttpServlet`。

```
1 public class LoginAction extends HttpServlet {
2     // Code goes on.
3 }
```

16.3.3 重写 doGet 方法

父类 `HttpServlet` 的 `doGet` 方法是空的，没有实现任何代码，子类需要重写此方法。

```
2 public void doGet(HttpServletRequest request, HttpServletResponse response)
3 throws ServletException, IOException {
4     // Rewrite the method.
5 }
```

当 HTTP 请求为 GET 时自动运行，每次请求都运行一次。

16.3.4 重写 doPost 方法

编写 Servlet 需要重写父类的 `doPost` 方法。

```
1 public void doPost(HttpServletRequest request, HttpServletResponse response)
2 throws ServletException, IOException {
3     // Rewrite the method.
4 }
```

当请求方式为 POST 时自动运行，每次请求都运行一次。

doGet 和 doPost 方法都接收 Web 容器自动创建的请求对象和响应对象，使得 Servlet 能够解析请求数据和发送响应给客户端。

16.3.5 重写 init 方法

当 Web 服务器创建 Servlet 对象后，会自动调用 init 方法完成初始化功能，一般将耗时的连接数据库和打开外部资源文件的操作放在 init 方法中。

init 方法在 Web 容器创建 Servlet 对象后立即执行，且只执行一次。

```
1 public void init (ServletConfig config) throws ServletException {  
2     super.init (config);  
3     // 这里放置初始化工作代码.  
4 }
```

在 init 方法中使用 Web 容器传递的 config 对象取得 Servlet 的各种配置初始参数，进而使用这些参数完成读取数据库或其他外部资源。

16.3.6 重写 destroy 方法

当 Web 容器需要销毁 Servlet 对象时，一般是 Web 容器停止运行或 Servlet 源代码修改而重新部署时，Web 容器自动运行 destroy 方法完成清理工作，如关闭数据库连接和 I/O 流。

```
1 public void destroy() {  
2     try {  
3         cn.close ();  
4     } catch (Exception e) {  
5         application.Log("登录处理关闭数据库错误" + e.getMessage());  
6     }  
7 }
```

代码中 application 为 Web 应用的上下文环境对象。

16.4 Servlet 生命周期

16.4.1 Servlet 的运行过程

1. 用户在浏览器请求 ServletURL 地址。
2. Web 容器接收到请求，检查是 Servlet 请求，将处理交给 Servlet 引擎。

3. Servlet 引擎根据 URL 地址检查是否有 Servlet 映射，如果没有则返回错误信息给浏览器。
4. 有 servlet 映射时，先检查是否有实例在运行。
5. 如果没有实例运行，则创建 Servlet 类的对象，调用其构造方法，然后调用 init() 方法。
6. 如果有实例在运行，则根据请求的方法是 GET 或 POST，自动调 doGet() 或 doPost() 方法。将请求对象和响应对象传给 doGet() 或 doPost() 方法。
7. 在 doGet() 或 doPost() 方法内通过 HttpServletRequest 的请求对象分析出用户发送的请求信息。
8. 按用户的要求进行业务处理。
9. 通过 HttpServletResponse 响应对象向浏览器发送响应信息。

16.4.2 Servlet 处理流程

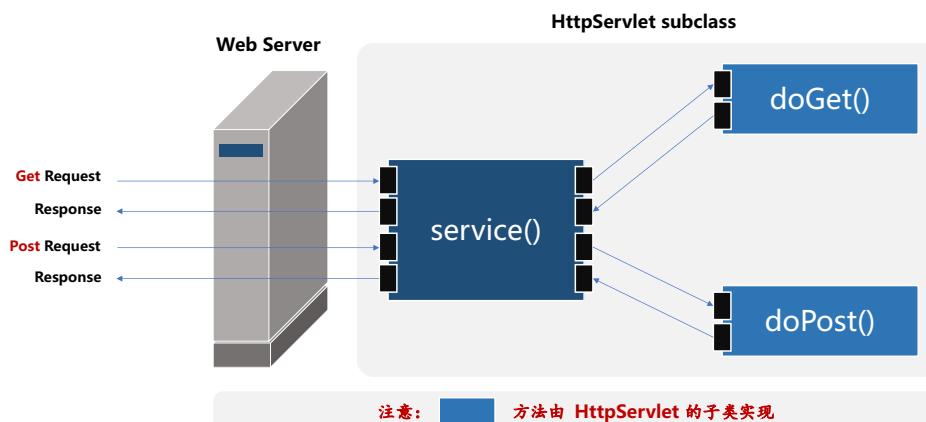


图 16.2 Servlet 处理流程

16.5 Servlet 配置

- Servlet 作为 Web 组件可以处理 HTTP 请求/响应，因此对外要求一个唯一的 URL 地址。
- Servlet 是一个 Java 类文件，不像 JSP 那样直接存放在 Web 目录下就能获得 URL 请求访问地址。

- Servlet 必须在 Web 的配置文件/**WEB-INF/web.xml**中进行配置和映射才能响应 HTTP 请求。
- Servlet 的配置分为**声明**和**映射**两个步骤。

16.5.1 Servlet 声明

通知 Web 容器 Servlet 的存在。

```
1 <servlet>
2   <servlet-name>loginaction</servlet-name>
3   <servlet-class>ouc.java.servlet .LoginAction</servlet-class>
4 </servlet>
```

<servlet-name> 声明 Servlet 的名字，要求在一个 web.xml 文件内名字唯一。

<servlet-class> 指定 Servlet 的全名，即包名. 类名。

16.5.2 Servlet 初始参数

在 Servlet 的声明中可以配置 Servlet 初始参数，如数据库的 Driver、URL、账号和密码等信息。在 Servlet 中可以读取这些信息，避免在 Servlet 代码中定义这些信息，修改时无需重新编译 Servlet。

```
1 <servlet>
2   <init-param>
3     <param-name>driver</param-name>
4     <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
5   </init-param>
6 </servlet>
```

在 Servlet 中取得以上定义的参数的方法：

```
1 String driver = config.getInitParameter("driver");
```

16.5.3 Servlet 启动时机

在配置 Servlet 时，可以指示 Servlet 跟随 Web 容器一起自动启动。这时，Servlet 就可以在没有请求的情形下，进行实例化和初始化，完成特定任务。自启动 Servlet 的配置语法：

```
1 <load-on-startup>2</load-on-startup>
```

数字越小越先启动，0 表示紧跟 Web 容器启动后第一个启动。

16.5.4 Servlet 映射

- 任何 Web 文档在 Internet 上都要有一个 URL 地址才能被请求访问。
- Servlet 不能像 JSP 一样直接放在 Web 的发布目录上，需要单独映射 URL 地址。
- 在 **/WEB-INF/web.xml** 中进行 Servlet 的 URL 映射。

❖ 映射语法

```
1 <servlet-mapping>
2   <servlet-name>servlet name</servlet-name>
3   <url-pattern>URL</url-pattern>
4 </servlet-mapping>
```

其中，servlet name 与 Servlet 声明中的名称要一致。

❖ 映射地址方式 ① 绝对地址方式映射

```
1 <servlet-mapping>
2   <servlet-name>LoginAction</servlet-name>
3   <url-pattern>/login.action</url-pattern>
4 </servlet-mapping>
```

❖ 映射地址方式 ② 匹配目录模式映射方式

```
1 <servlet-mapping>
2   <servlet-name>MainAction</servlet-name>
3   <url-pattern>/main/*</url-pattern>
```

```
4 </servlet-mapping>
```

在这个配置中，只要以/main 开头的任何 URL 都能请求此 Servlet。

◆ 映射地址方式 ③ 匹配扩展名模式映射方式

```
1 <servlet-mapping>
2   <servlet-name>MainAction</servlet-name>
3   <url-pattern>*.action</url-pattern>
4 </servlet-mapping>
```

以上配置中扩展名为 action 的任何请求均被此 Servlet 响应。

注意：不能混合使用以上两种配置模式，否则会在 Web 项目部署并运行时产生运行时错误。

如以下配置是错误的：

```
1 <servlet-mapping>
2   <servlet-name>MainAction</servlet-name>
3   <url-pattern>/main/*.action</url-pattern>
4 </servlet-mapping>
```

16.6 Servlet 部署

编译好的 Servlet class 文件应该放到指定的 Web 应用目录下，才能被 Web 容器找到，这个路径为：[/WEB-INF/classes/package/FileName.class](#)

例如 Servlet 类 LoginAction：

```
1 package ouc.java.servlet ;
2 public class LoginAction extends HttpServlet {
3   //
4 }
```

存放路径为：[/WEB-INF/classes/ouc/java/servlet/LoginAction.class](#)

16.7 Servlet 例

16.7.1 Eclipse

New Project ➔ Web ➔ Dynamic Web Project ➔ Next >

- Project name: sample.servlet
- Target runtime: Apache Tomcat v8.0
- Dynamic web module version: 3.0
- Configuration: Default Configuration for Apache Tomcat v8.0

➔ Next >

Source folder on build path: src

➔ Next >

Generate web.xml deployment descriptor

➔ Finish

16.7.2 WebContent/WEB-INF/web.xml

Add following statements between <web-app> and </web-app>.

```
1 <servlet>
2   <servlet-name>HelloServlet</servlet-name>
3   <servlet-class>ouc.javaweb.HelloServlet</servlet-class>
4   <load-on-startup>0</load-on-startup>
5 </servlet>

7 <servlet-mapping>
8   <servlet-name>HelloServlet</servlet-name>
9   <url-pattern>/hello</url-pattern>
10 </servlet-mapping>
```

16.7.3 Java Resources/src

Create java class file named ``HelloServlet.java''.

```
1 package ouc.javaweb;  
2  
3 import java.io.IOException;  
4 import java.io.PrintWriter;  
5 import javax.servlet.ServletException;  
6 import javax.servlet.http.HttpServlet;  
7 import javax.servlet.http.HttpServletRequest;  
8 import javax.servlet.http.HttpServletResponse;  
9  
10 public class HelloServlet extends HttpServlet {  
11     public void doGet(HttpServletRequest request, HttpServletResponse response)  
12         throws ServletException, IOException {  
13         response.setContentType("text/html");  
14         response.setCharacterEncoding("UTF-8");  
15         PrintWriter out = response.getWriter();  
16         out.println("<html>");  
17         out.println("<head><title>A Servlet Sample</title></head>");  
18         out.println("<body>");  
19         out.println("<h1>Hello, Servlet!</h1>");  
20         out.println("A Servlet is a Java-based server-side web technology. ");  
21         out.println("</body></html>");  
22         out.flush();  
23         out.close();  
24     }  
25 }
```

sample.servlet ➔ 鼠标右键 ➔ Run as ➔ Run on Server

➔ Choose an existing server ➔ Tomcat v8.0 Server at localhost ➔ Finish

在浏览器中请求页面<http://localhost:8080/sample.servlet/hello>。

☒ 17 ☒ HTTP 请求处理编程

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第十周

参考教材: 本课程参考教材及资料如下:

- 吕海东, 张坤编著, Java EE 企业级应用开发实例教程, 清华大学出版社,
2010 年 8 月

教学目标

1. 掌握 HTTP 协议的特点以及 HTTP 请求中包含哪些信息。
2. 理解 Java HTTP 请求对象的类型及其生命周期, 掌握请求对象的功能。
3. 学习并实践掌握部分请求对象方法的用法。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

17.1 HTTP 请求内容

17.1.1 Web 工作模式

Web 通常使用**请求—响应**模式。

- 客户端（浏览器）向服务器发出 HTTP 请求，在 HTTP 请求中包含传递到服务器的数据；
- Web 服务器接收到请求，对请求进行处理。
- Web 服务器使用 HTTP 向客户端发送响应。
- 客户端接收到响应后，进行显示或页面跳转。

17.1.2 HTTP 请求中包含的信息

HTTP 请求中包含的信息包括两部分：**请求头和请求体**。

请求头

```
1 GET /articles/news/today.jsp HTTP/1.1
2 Accept: */*
3 Accept-Language: en-us
4 Connection: Keep-Alive
5 Host: localhost
6 Referer: http://localhost/links.asp
7 User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
8 Accept-Encoding: gzip, deflate
9 ...
```

❖ HTTP 请求头标记和说明

User-Agent 浏览器的机器环境

Accept 浏览器支持哪些 MIME 数据类型

Accept-Charset 浏览器支持的字符编码

Accept-Encoding 浏览器支持哪种数据压缩格式

Accept-Language 浏览器指定的语言环境

Host 浏览器访问的主机名

Referer 浏览器是从哪个页面来的

Cookie 浏览器保存的 cookie 对象

Java EE Web 组件 Servlet 和 JSP 中可以使用请求对象的方法读取这些请求内容，进而进行相应的处理。

请求体

每次 HTTP 请求时，在请求头之后会有一个**空行**，接下来是请求中包含的提交数据，即**请求体**。

① GET 请求

无请求体，请求数据直接在请求的 URL 地址中，作为 URL 的一部分发送给 Web 服务器。

```
1 http://localhost:8080/webapp/login.do?id=9001&pass=9001
```

- 请求体为空，提交数据直接在 URL 上，作为请求头部分传输到 Web 服务器，通过 URL 的QueryString 部分能得到提交的参数数据。
- 此种方式对提交数据的大小有限制，不同浏览器会有所不同，如 IE 为 2083 字节。GET 请求时数据会出现在 URL 中，保密性差，实际编程中要尽量避免。

② POST 请求

- 请求体数据单独打包为数据块，通过 Socket 直接传递到 Web 服务器端，数据不会在地址栏出现。
- 可以提交大的数据，包括二进制文件，实现文件上传功能。
- 原则上 POST 请求对提交的数据没有大小限制。

17.2 Java EE 请求对象

17.2.1 请求对象类型与生命周期

◆ 请求对象接口类型

- Java EE 规范中的通用请求对象要实现接口 javax.servlet.ServletRequest
- HTTP 请求对象要实现接口

javax.servlet.http.HttpServletRequest

◆ 请求对象生命周期

在 Java Web 组件开发中，不需要 Servlet 或 JSP 自己创建请求对象，它们由**Web 容器自动创建**，并传递给 Servlet 和 JSP 的服务方法 doGet 和 doPost，在服务处理方法中直接使用请求对象即可。

17.2.2 请求对象类型与生命周期

◆ 请求对象创建

每次 Web 服务器接收到 HTTP 请求时，会自动创建实现 HttpServletRequest 接口的对象。在创建该对象之后，Web 服务器将请求头和请求体信息写入请求对象，Servlet 和 JSP 可以通过请求对象的方法取得这些请求信息，继而可以取得用户提交的数据。

◆ 请求对象销毁

当 Web 服务器处理 HTTP 请求，向客户端发送 HTTP 响应结束后，会自动销毁请求对象，保存在请求对象中的数据随即丢失。当下次请求时新的请求对象又会被创建。

17.2.3 请求对象功能与方法

◆ 请求对象方法一般分类

- 取得请求头信息；
- 取得请求体中包含的提交参数数据，包含表单元素或地址栏 URL 的参数；
- 取得客户端的有关信息，如请求协议、IP 地址和端口等；

- 取得服务器端的相关信息，如服务器的 IP 等；
- 取得请求对象的属性信息，用于在一个请求的转发对象之间传递数据。

17.2.4 取得请求头

- `String getHeader(String name)`

```
1 String browser = request.getHeader("User-Agent");
```

- `int getIntHeader(String name)`

```
1 int size = request.getIntHeader("Content-Length");
```

- `long getDateHeader(String name)`

```
1 long datetime = request.getDateHeader("If-Modified-Since");
```

- `Enumeration getHeaderNames()`

```
1 for (Enumeration e = request.getHeaderNames(); e.hasMoreElements(); ) {  
2     String headerName = (String) e.nextElement();  
3     System.out.println("Name = " + headerName);  
4 }
```

17.2.5 取得请求中包含的提交参数数据

- 在 Web 开发中，用户通过表单或 URL 参数将客户端数据提交到服务器端，这些数据被 Web 服务器自动封装到请求对象中。
- Web 组件 Servlet 和 JSP 可以通过请求对象获得用户提交的数据。

`String getParameter(String name)`

取得表单或 URL 参数中指定名称的数据值。

表单

```
1 Product Name: <input type="text" name="productName" />
```

URL 参数

```
1 productSearch.do?productName=Acer
```

以下代码可以取得以上的参数名为 productName 的数据：

```
1 String productName = request.getParameter("productName");
```

`String[] getParameterValues(String name)`

取得指定名称的参数数据数组，如复选框和复选列表。

表单复选数据

```
1 爱好：  
2 <input type="checkbox" name="behave" value="旅游">旅游  
3 <input type="checkbox" name="behave" value="读书">读书  
4 <input type="checkbox" name="behave" value="体育">体育
```

如下代码取得选定的爱好：

```
1 String [] behaves = request.getParameterValues("behave");  
2 for (int i = 0; i < behaves.length; i++) {  
3     out.println(beaves[i]);  
4 }
```

`Enumeration getParameterNames()`

取得所有请求的参数名称。

```
1 for (Enumeration enums = request.getParameterNames(); enums.hasMoreElements();) {  
2     String paramName = (String) enums.nextElement();  
3     System.out.println(paramName);  
4 }
```

Map getParameterMap()

取得所有请求的参数名和值，包装在一个 Map 对象中，可以使用这个对象同时取得所有的参数名和参数值。

```
1 Map params = request.getParameterMap();
2 Set names = params.keySet();
3 for (Object o: names) {
4     String paramName = (String) o;
5     out.print(paramName + " = " + params.get(paramName) + "<br/>");
6 }
```

ServletInputStream getInputStream() throws IOException

取得客户端的输入流。

- 当用户提交的数据中包含文件上传时，提交的数据可以以二进制编码方式提交到服务器。
- 当表单既有文本字段还有文件上传时，就需要对此二进制流进行解析，从而分离出文本和上传文件。
- 可以使用第三方框架/Jar 包实现上传文件的处理，如[Apache 的 Common upload 组件](#)。

注意

当使用 getParameter() 方法后，就无法使用 getInputStream() 方法，反之亦然。

```
1 <form action="" method="" enctype="multipart/form-data">
2   姓名 : <input type="text" name="name"/><br/>
3   照片 : <input type="file" name="photo"/><br/>
4   <input type="submit" value="提交"/>
5 </form>
```

17.2.6 取得其他客户端信息

`String getRemoteHost()` 取得请求客户的主机名。

`String getRemoteAddr()` 取得请求客户端的 IP 地址。

`int getRemotePort()` 取得请求客户的端口号。

`String getProtocol()` 取得请求的协议。

`String getContentType()` 取得请求体的 MIME 内容类型。

`int getContentLength()` 取得请求体为二进制流时请求体的长度。

`String getProtocol()` 取得请求的协议，一般为 HTTP。

17.2.7 取得服务器端信息

`String getServerName()` 取得服务器的 HOST，一般为 IP 地址。

`int getServerPort()` 取得服务器接收端口。

☒ 18 ☒ HTTP 响应处理编程

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第十周

参考教材: 本课程参考教材及资料如下:

- 吕海东, 张坤编著, Java EE 企业级应用开发实例教程, 清华大学出版社,
2010 年 8 月

教学目标

1. 掌握 HTTP 响应的内容, 包括响应状态行、响应头、响应体。
2. 理解 Java HTTP 响应对象的类型及其生命周期, 掌握响应对象的功能。
3. 学习并实践掌握部分响应对象方法的用法。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

18.1 HTTP 响应的内容

18.1.1 HTTP 响应的内容

在 Web 服务器接收请求处理后，向客户端发送 HTTP 响应（Response）。响应的内容包括：

- 响应状态（Status Code）
- 响应头（Response Header）
- 响应体（Response Body）

HTTP 响应状态行

表明响应的状态信息，如成功、失败、错误。状态行的构成包括：版本 / 状态代码 / 状态消息。

❖ 状态行例子

HTTP/1.1 200 ok

1. 版本：使用的 HTTP 协议版本，如 HTTP/1.1；
2. 状态代码：3 位数字；
 - 1xx：收到请求，没有处理完。
 - 2xx：成功，响应完毕。
 - 3xx：重定向，到另一个请求中去。
 - 4xx：失败，没有请求的文档等。
 - 5xx：内部错误，代码出现异常。
3. 状态描述。

响应头

Web 服务器在向客户端发送 HTTP 响应时也可以包含响应头，来指示客户端浏览器如何处理响应体，主要包括响应的类型、字符编码和字节大小等信息。

❖ 常见响应头内容

1. 指示 HTTP 响应可以接收到的文档类型集：Accept
2. 告知客户可以接收的字符集：Accept-Charset
3. 响应的字符编码集：Accept-Encoding
4. 响应体的 MIME 类型：Content-Type
5. 响应体的语言类型：Content-Language
6. 响应体的长度和字节数：Content-Length
7. 通知客户端到期时间：Expires
8. 缓存情况：Cache-Control
9. 重定向到另一个 URL 地址：Redirect

响应体

响应体类型由响应头确定，可以是任何类型。浏览器在处理响应体之前，会收到响应头，根据响应头的信息，确定如何处理响应体。**如响应头的 Content-Type 为 PDF，则浏览器会启动 PDF Reader 来处理此响应体以显示 PDF 文档。**

❖ 常用响应类型

1. 纯文本：text/plain
2. HTML：text/html
3. 图片：image/gif, image/jpeg
4. PDF：application/pdf

注意

- 文本类型响应要求响应头中包含 MIME 类型和字符编码集，使用**字符输出流**向客户端发送响应体数据；
- 二进制数据类型响应需要在响应头中包含 MIME 类型，不要设置字符编码集，使用**字节输出流**向客户端发送响应体数据。

18.2 HTTP 响应对象

18.2.1 响应对象类型

HTTP 响应对象类型为：javax.servlet.http.HttpServletResponse。响应对象职责主要包括：

- 设置状态行；
- 发送响应头；
- 向 Web 浏览器发送 HTTP 响应体；
- 控制页面的重定向，即将告知浏览器再发送一次请求。

18.2.2 响应对象生命周期

1. Web 容器自动为每次 Web 组件的请求生成一个响应对象。
2. Web 容器创建响应对象后，传入到 doGet 或 doPost 方法。
3. 通过响应对象向浏览器发响应。
4. 响应结束后，Web 容器销毁响应对象，释放所占用的内存。

18.3 响应对象功能和方法

18.3.1 设置响应状态码

一般情况下，Web 开发人员不需要通过编程来改变响应状态码，Web 服务器会根据请求处理的情况自动设置状态码，并发送到客户端浏览器。例如，当客户请求不存在的 URL 地址时，Web 服务器会自动设置状态码为 404，状态消息为 not found。

```
public void setStatus(int code)
```

直接发送指定的响应状态码，没有设置状态消息，只有默认的状态消息，如果无对应状态消息则显示为空。

```
public void setStatus(int code, String message)
```

设置指定的状态码，同时设定自定义的状态消息，可以修改默认的状态消息。该方法在 Servlet 2.5 后被舍弃，一般不要使用。

```
public void sendError(int sc) throws IOException
```

向客户端发送指定的错误信息码，可以是任意定义的整数。

```
1 response.setCharacterEncoding("GBK");  
2 response.sendError(580);
```

```
public void sendError(int sc, String msg) throws IOException
```

向客户端发送指定的错误信息码和自定义状态消息。

```
1 response.setCharacterEncoding("GBK");  
2 response.sendError(580, "自定义错误");
```

18.3.2 设置响应头

当客户端接收到响应状态为 200 时，浏览器会继续接收响应头信息，来确定响应体的类型和大小。

```
public void setHeader(String name, String value)
```

将指定名称和值的响应头发送到客户端。

```
1 response.setHeader("Content-Type", "text/html");
```

```
public void setIntHeader(String name, int value)
```

设置整数类型的响应头的名和值。

```
1 response.setHeader("Content-Length", 20);
```

注意

实际项目中无需设定该响应头，Web 服务器会自动计算并发送给浏览器。

```
public void setDateHeader(String name, long date)
```

设定日期类型的响应头，参数 date 为 GMT 格式的日期。

18.3.3 设置响应头的便捷方法

```
public void setContentType(String type)
```

直接设置响应内容类型 MIME 响应头。

```
public void setContentLength(int len)
```

设置响应体长度，以字节为单位。

```
void setCharacterEncoding(String charset)
```

设置响应字符集，包括响应状态，响应头和响应体。

```
public void setBufferSize(int size)
```

设定响应体的缓存字节数。

如设定响应体缓存为 4k：

```
response.setBufferSize(4096);
```

注意

Servlet 在发送响应时，一般按照发送状态码、响应头和响应体的顺序进行，大的响应体缓存，可以允许 Servlet 有更多的时间发送状态码和响应头，这种情况发生在响应头和响应体同时写的情况。编程的时候最好先把响应头全部设定后，再发送响应体。

18.3.4 响应对象方法——向客户端传送 Cookie

◆ `public void addCookie(Cookie cookie)`

此方法功能将 Cookie 对象放置在响应头中，随响应内容到浏览器客户端，并保存到客户端的 PC 的本地目录中。

```
1 Cookie cookie01=new Cookie("userid", "9001");
2 response.addCookie(cookie01);
```

18.3.5 响应对象方法——请求重定向

❖ public void sendRedirect(String url)

将对客户的响应重定向到新的 URL 上，让客户端浏览器对此 URL 进行请求。

重定向到登录页面，相当于在浏览器地址栏上再输入一次 URL 地址，进行一次 HTTP 请求：

```
1 String url="../admin/login.jsp";
2 response.sendRedirect(url);
```

18.3.6 设置响应体发送功能

响应体即浏览器实际显示的具体内容，可以时 HTML 网页，也可以是其他文件格式，由响应头的 Content-Type 决定。

响应体的类型主要分为两大类，即文本类型和二进制类型。文本类型使用字符输出流 PrintWriter 的对象来实现；二进制类型由 OutputStream 的对象来实现。

❖ public PrintWriter getWriter()

取得字符输出流。

❖ public ServletOutputStream getOutputStream()

取得二进制输出流。

18.3.7 设置响应体——文本类型响应体发送编程

1. 设置响应类型 ContentType

```
1 response.setContentType("text/html"); //响应类型为 HTML 文档
```

2. 设置响应字符编码

```
1 response.setCharacterEncoding("GBK"); //字符编码使用 GBK
```

3. 取得字符输出流对象

```
1 PrintWriter out = response.getWriter();
```

4. 向流对象中发送文本数据

```
1 out.println("<html><body></body></html>"); //输出文本字符
```

5. 清空流中缓存的字符

```
1 out.flush();
```

6. 关闭流

```
1 out.close();
```

18.3.8 设置响应体——文本类型响应体发送编程

示例代码：[示例代码](#)

```
1 response.setContentType("text/html; charset=gb2312");
2 PrintWriter out = response.getWriter();

4 out.println("<html><head>");
5 out.println("</head><body>");
6 out.println("hello! ");
7 out.println("</body></html>");
8 out.flush();
9 out.close();
```

18.3.9 设置响应体——二进制类型响应体发送编程

1. 设置响应类型 ContentType

```
1 response.setContentType("image/jpeg"); //响应类型为 JPEG 图片
```

2. 取得字节输出流对象

```
1 OutputStream out = response.getOutputStream(); //取得字节输出流
```

3. 向流对象中发送字节数据

```
1 out.println(200); //输出字节数据
```

4. 清空流中缓存的字节

```
1 out.flush();
```

5. 关闭流

```
1 out.close();
```

注意：二进制响应编程不需要设置字符编码。

☒ 19 ☒ HTTP 会话跟踪编程

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第十一周

参考教材: 本课程参考教材及资料如下:

- 吕海东, 张坤编著, Java EE 企业级应用开发实例教程, 清华大学出版社,
2010 年 8 月

教学目标

1. 掌握会话的基本概念, 理解会话不是仅仅使用 HTTP 协议就能够保证的, 而是客户端浏览器和服务器端在 HTTP 协议之上采用额外的技术协同的结果。
2. 掌握常用的会话跟踪技术, 了解采用 URL 重写维持会话跟踪的方法; 理解 Cookie 和 Session 的协同机制, 掌握使用 Cookie 和 Session 实现会话跟踪的技术。
3. 能够使用 Cookie 和 Session 编写会话跟踪代码。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

19.1 会话基本概念

19.1.1 什么是会话

在 Web 应用中把客户端浏览器开始请求 Web 服务器，访问不同 Web 文档进行请求/响应，到结束访问的一系列过程称为会话，即一次会话（Session）。当用户访问第一个 Java Web 组件时，Java EE Web 服务器自动为用户创建一个会话对象。

例如，当当网进行图书浏览、购买、完成结算的全过程可能是一次会话；登录 126 邮箱，完成浏览收件箱、编写邮件、发送邮件、登出邮箱可以是一次会话。

19.1.2 会话跟踪

❖ HTTP 的固有设计“缺陷”

由于 Web 应用采用 HTTP 协议，而 HTTP 协议是无状态、不持续的协议，所以需要独立于 HTTP 协议的会话跟踪技术，用于记录会话的状态信息。

❖ 什么是会话跟踪

- 在一个会话内，当用户在次访问时，服务器需要能够定位是先前访问的同一个用户。
- Web 应用需要在用户访问的一个会话内，让 Web 服务器保存客户的信息（如客户的账号或客户的购物车），称为**会话跟踪**，即 Web 服务器必须使用某种技术保存客户的信息。

19.1.3 Java EE Web 会话跟踪方法

1. **重写 URL** 将客户端的信息附加在请求 URL 地址的参数中，Web 服务器取得参数信息，完成客户端信息的保存。
2. **隐藏表单字段** 将要保存的客户信息，如用户登录账号使用隐藏表单字段发送到服务器端，完成 Web 服务器保持客户状态信息。

3. **Cookie** 使用 Java EE API 提供的 Cookie 对象，可以将客户信息保存在 Cookie 中，完成会话跟踪功能。
4. **HttpSession 对象** Java EE API 专门提供了 HttpSession 会话对象保存客户的信息来实现会话跟踪。

以下分别介绍相关会话跟踪技术。

19.2 URL 重写

19.2.1 URL 重写实现实现会话跟踪的方法

浏览器端构造 URL 请求

- 在进行 HTTP 请求时，可以在 URL 地址后直接附加请求参数，把客户端的数据传输到 Web 服务器端。
- Web 服务器通过 HttpServletRequest 请求对象取得这些 URL 地址后面附加的请求参数。
- 这种 URL 地址后附加参数的方式称为 URL 重写。

URL 重写示例

```
<a href="..//product/main.do?userid=9001&category=11">产品管理</a>
```

此例中，将客户 ID 附加在地址栏上，以?name=value 形式附加在 URL 后，多个参数使用 & 符号进行间隔。

服务器端解析 URL 获取用户会话标识

Web 服务器端使用请求对象取得 URL 后附加的客户端参数数据。

```
String userid = request.getParameter("userid"); // 取得用户 ID 参数数据
```

浏览器和服务器两端持续带会话标识通信

为保证 Web 应用能在以后持续的请求/响应中实现会话跟踪，必须保证每次请求都要在 URI 地址中加入 userid=9001 参数，进而实现会话跟踪。

如下为 Servlet 重定向请求的附加参数：

```
response.sendRedirect("../product/view.do?productid=1201&userid=" + userid);
```

19.2.2 URL 重写的缺点

- URL 传递参数的限制
- 安全性缺陷
- 编程繁杂

19.3 Cookie

19.3.1 什么是 Cookie

- Cookie 在 Java EE 之前就已经存在，是由 Netscape 浏览器引入的，用于在客户端保存服务器端数据，实现一种简单有效的客户/服务器的信息交换模式。
- Cookie 是 Web 服务器保存在客户端的小的文本文件，存储许多 name/value 对，可以保存如登录帐号、用户喜好等会话数据。
- Cookie 由 Web 服务器创建，由 Web 服务器在进行 HTTP 响应时，将 Cookie 保存在 HTTP 响应头中并发送给浏览器，浏览器收到 HTTP 响应头，解析出 Cookie，将它保存在客户的本地隐藏文件中。
- 客户浏览器每次向 Web 服务器发出 HTTP 请求时，自动将 Cookie 保存在请求头中，随请求体一起发送到服务器，这个过程不需要人工参与。
- Web 服务器可以从请求对象中取出 Cookie，进而得到 Cookie 中保存的名称/值对，从而实现会话跟踪。

19.3.2 Java EE 规范 Cookie API

Java EE API 提供**javax.servlet.http.Cookie**类来表达一个 Cookie 对象。

- HttpServletResponse 接口中定义了保存 Cookie 到浏览器的方法
- HttpServletRequest 接口中定义了取得客户端保存的 Cookie 对象的方法

❖ Cookie 对象的创建

使用 Cookie 类的构造方法 public Cookie(String name, String value)

```
1 Cookie cookie01 = new Cookie("userid", "9001");
```

19.3.3 Cookie 的主要方法

get 方法

1. public String getName()
2. public String getValue()
3. public int getMaxAge()
4. public String getPath()
5. public int getVersion()
6. public String getDomain()

set 方法

1. public void setValue(String newValue)
2. **public void setMaxAge(int expiry)**
3. public void setDomain(String pattern)
4. public void setPath(String uri)

19.3.4 将 Cookie 保存到客户端

1. 创建 Cookie 对象

```
1 String userid = request.getParameter("userid"); // 取得登录 ID  
2 Cookie cookie01 = new Cookie("userid", useid); // 保存到Cookie中
```

2. 设置 Cookie 属性

```
1 cookie01.setMaxAge(7 * 24 * 60 * 60); // 设置cookie的有效期
```

3. 发送 Cookie 到客户端

```
1 response.addCookie(cookie01);
```

19.3.5 Web 服务器读取客户端保存的 Cookie

❖ public Cookie[] getCookies()

```
1 Cookie[] cookies = request.getCookies();  
2  
3 for (int i = 0; i < cookies.length; i++) {  
4     if (cookies[i].getName().equals("userid")) {  
5         String name = cookies[i].getValue();  
6     }  
7 }
```

19.3.6 Cookie 的缺点

- 存储方式单一
- 存储位置限制
- 大小受浏览器限制
- 可用性限制
- 安全性限制（可以采用手动 Cookie 加解密）

19.4 Java EE 会话对象

19.4.1 什么是会话对象

- Java EE 规范提出了一种服务器实现**会话跟踪**的机制，即 HttpSession 接口，实现该接口的对象称为 Session 对象。
- Session 对象保存在 Web 服务器上，每次会话过程创建一个，为用户保存各自的会话信息提供全面支持。
- 注意不要将过多的数据存放在会话对象内，如只在一个请求期间内需要传递的数据，就不要存储在会话对象中，而应该保存在**请求对象**中。

19.4.2 会话对象的类型和取得

会话对象的类型接口为**javax.servlet.http.HttpSession**, 在请求对象类型 HttpServletRequest 中定义了取得会话对象的方法。

1. public HttpSession getSession()

- 如果 Web 服务器内没有此客户的会话对象，则 Web 容器创建新的会话对象并返回；
- 如果已经存在会话对象，则直接返回此对象的引用。

2. public HttpSession getSession(boolean create)

- boolean 参数为 true 时，同无参数的 getSession 方法；
- boolean 参数为 false 时，如存在会话对象则返回对象引用，如无会话对象则返回 null，Web 容器不会创建会话对象。

19.4.3 会话对象的功能和方法

◆ public void setAttribute(String name, Object value)

将数据存入会话对象，以 name/value 模式进存储，value 的类型为通用的 **Object** 类型，可以将任何 Java 对象保存到会话对象中。

```
1 HttpSession session = request.getSession();
2 Collection shopcart = new ArrayList();
3 Session.setAttribute("shopcart", shopcart);
```

例如，使用容器类型 Collection、List、Set 或 Map 可以非常容易的实现电子商务网站购物车的存取功能。

◆ public Object getAttribute(String name)

取出保存在会话中指定名称属性的值对象，需要根据保存时使用的类型进行**造型/强制类型转换**。

```
1 Collection shopcart = (Collection) session.getAttribute("shortcart");
```

❖ public void removeValue(String name)

清除会话对象中某个属性对象。

```
1 session.removeAttribute("shopcart");
```

❖ public Enumeration getAttributeNames()

取得会话对象中保存的所有属性名称列表，返回一个枚举器类型对象。

```
1 Enumeration enume = session.getAttributeNames();
2 while (enume.hasMoreElements()) {
3     String name = (String) enume.nextElement();
4     out.println(name);
5 }
```

❖ public void setMaxInactiveInterval(int interval)

设置会话对象的失效期限，即 2 次请求直接的时间间隔，以秒为单位。

```
1 session.setMaxInactiveInterval(15 * 60);
```

❖ public int getMaxInactiveInterval()

取得会话的有效间隔时间，返回整数，表示间隔的秒数。

```
1 int maxtimes = session.getMaxInactiveInterval();
```

❖ public void invalidate()

立即迫使会话对象失效，将当前的会话对象销毁，同时清除会话对象内的所有属性，该方法一般用在注销处理的 Servlet 中。

```
1 session.invalidate();
```

❖ public boolean isNew()

测试取得的会话对象是否是刚刚创建的，true 表示刚刚创建，false 表示会话对象已经存在。

❖ public long getCreateTime()

取得会话对象的创建时间，返回 long 类型数据，表示从 1970.1.1.0 时开始到创建时间所间隔的毫秒数。

❖ public String getId()

取得会话对象的 ID，是唯一性的字符串代码。

由于 HTTP 协议的无状态性，为了使用 Web 容器能识别不同客户而确定各自的会话对象，Web 容器需要将会话 ID 保存到客户端。当客户进行 HTTP 请求时，需要发送此 ID 给服务器，Web 服务器根据此 ID 定位服务器内部的会话对象，实现指定客户的会话跟踪。

19.4.4 会话对象的生命周期

会话对象的生命周期比请求对象和响应对象的生命周期要长久，可以跨越多次不同的 Web 组件 JSP 和 Servlet 的请求和响应，因此会话对象可以作为不同 JSP 和 Servlet 之间的数据共享区，保存不同页面需要访问的数据。

创建状态 新的用户请求到来。

活动状态 一个会话有效期内的所有请求，将共享一个会话对象。

销毁状态 • 客户端浏览器所保存的 SessionID 被销毁时；

- 服务器端执行会话对象的 invalidate() 方法时；
- 客户端请求间隔时间超时。

19.4.5 会话 ID 的保存方式

默认情况下，会话的 ID 值会**自动发送到客户端的 Cookie 中进行保存**。

每次 HTTP 请求时，所有 Cookie 会随请求一起发送到 Web 服务器，保存在请求对象中。Web 服务器从 Cookie 中得到 SessionID，进而定位服务器内的 HttpSession 会话对象，实现 Web 的会话跟踪功能。

19.4.6 会话对象的应用示例

❖ 保存用户的登录 ID

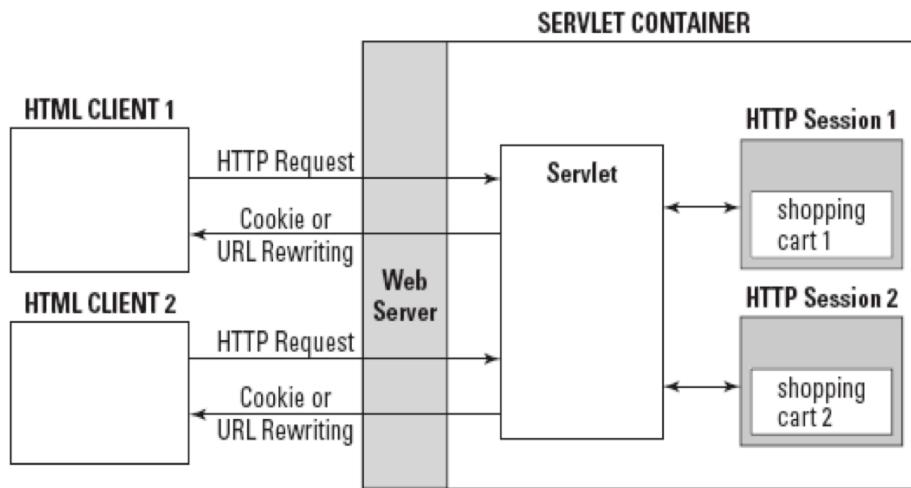


图 19.1 使用 HttpSession 实现会话跟踪

```

1 void doPost(HttpServletRequest request, HttpServletResponse response) {
2     String userId = request.getParameter("userId");
3     String password = request.getParameter("password");
4
5     User user = BeanFactory.get("USER"); // 从Bean工厂获取一个User的对象实例
6
7     if (user.validate(userId, password) { // 验证用户的有效性
8         HttpSession session = request.getSession(); // 取得会话对象
9         session.setAttribute("userId", userId); // 将UserId保存到会话对象
10    } else { // 用户无效
11        response.sendRedirect("../login.jsp"); // 重定向到登录页面
12    }
13 }
```

◆ 会话对象销毁（编程一般在注销功能中）

```

1 public class LogoutAction extends HttpServlet {
2     public void doGet(HttpServletRequest request, HttpServletResponse response) {
3         HttpSession session = request.getSession();
4         session.invalidate(); // 清除此会话对象
5         response.sendRedirect("login.jsp"); // 跳转到登录页面
6     }
}
```

7 } |

☒ 20 ☒ ServletContext 和 Web 配置

基本信息

课程名称: Java 应用与开发

授课教师: 王晓东

授课时间: 第十周

参考教材: 本课程参考教材及资料如下:

- 吕海东, 张坤编著, Java EE 企业级应用开发实例教程, 清华大学出版社,
2010 年 8 月

教学目标

Java EE Web 应用需要部署在符合 Java EE 规范的 Web 容器中运行, 如何取得 Web 应用本身的信息在编程中非常重要。

1. 掌握 Web 应用对象 ServletContext。
2. 了解 Web 应用的配置方法。
3. 掌握 MVC 模式 Web 开发中发挥核心作用的转发, 区别转发与重定向。

授课方式

理论课: 多媒体教学、程序演示

实验课: 上机编程

教学内容

20.1 Web 应用环境对象

20.1.1 Web 应用环境对象

将 Web 应用部署到服务器上，启动 Web 服务器后，Web 容器为每个 Web 应用创建一个表达 Web 应用环境的对象（即 ServletContext 对象），并将 Web 应用的基本信息存储在这个 ServletContext 对象中。

❖ Web 应用环境对象的用途

- 所有 Web 组件都可以访问此 ServletContext 对象，进而取得 Web 应用的基本信息。
- ServletContext 还可以作为整个 Web 应用的共享容器对象，能够被所有会话请求共用，保存 Web 应用的共享信息。

20.1.2 Web 应用环境对象的生命周期

ServletContext 对象的生命周期与 Web 应用相同。

创建 Web 容器启动后，自动创建 ServletContext 对象；

销毁 Web 容器停止时，自动销毁 ServletContext 对象。

注意

如果在 ServletContext 对象中保存的对象信息需要长久保存，一般编写 ServletContext 对象的监听器，在此对象销毁之前将其中保存的对象数据进行持久化处理，如保存到数据库或者文件中。

20.1.3 Web 应用环境对象的类型和取得

Web 应用环境对象是接口**javax.servlet.ServletContext**的实现。

❖ 在 Servlet 内直接取得 ServletContext 接口对象

```
1 ServletContext ctx = this.getServletContext();
```

20.1.4 Web 应用环境对象的功能和方法

❖ Web 级数据共享容器

```
public void setAttribute(String name, Object object)
```

对象保存到 ServletContext。

```
1 ServletContext ctx = this.getServletContext();
2 ctx.setAttribute("userId", "Kevin");
3 ctx.setAttribute("age", 20); // 自动完成 int 类型转换为 Integer 对象类型
```

```
public Object getAttribute(String name)
```

读取保存在 ServletContext 对象中指定名称的属性对象，不存在则返回 null。

```
1 String userId = (String) ctx.getAttribute("userId");
2 int age = (Integer) ctx.getAttribute("age"); // 自动拆箱，将 Integer 转为 int
```

```
public void removeAttribute(String name)
```

将指定的属性从 ServletContext 对象中删除。

```
Enumeration getAttributeNames()
```

取得所有属性的名称列表，返回一个枚举器对象，可以用于遍历所有属性名称。

```
1 Enumeration nums = ctx.getAttributeNames();
2 while (nums.hasMoreElements()) {
3     System.out.println(nums.nextElement());
4 }
```

注意

ServletContext 大量的方法请自行学习掌握。

20.2 Java EE Web 的配置

20.2.1 配置文件 web.xml

Web 的配置文件为 **/WEB-INF/web.xml**, /WEB-INF 目录是**被 Web 服务器保护的目录**, 客户端浏览器无法直接访问该目录下的任何文件, Struts、Spring 等框架都将配置文件保存在该目录下。

20.2.2 web.xml 的主要配置项

- Servlet 声明 (Servlet)
- Servlet 映射 (Servlet-mapping)
- Web 级初始参数 (context-param)
- 过滤器 (filter)
- 过滤器映射 (filter-mapping)
- 监听器 (listener)
- 异常跳转页面 (error-page)
- MIME 类型映射 (mime-mapping)
- 会话对象超时 (session-config)
- 外部资源声明 (resource-ref)
- 外部标记库描述符文件 (taglib)

20.2.3 Web 初始参数配置

❖ Web 初始参数配置

```
1 <context-param>
2   <description>数据库驱动</description>
3   <param-name>driverName</param-name>
4   <param-value>sun.jdbc.odbc.jdbcOdbcDriver</param-value>
5 </context-param>
```

❖ Web 组件取得 Web 初始参数

在 Servlet 中可以通过 ServletContext 对象取得 Web 初始参数。

`public String getInitParameter(String name)` 取得指定名称的 Web 初始参数。

```
1 ServletContext ctx = this.getServletContext();  
2 String driverName = ctx.getInitParameter("driverName");
```

20.2.4 会话超时配置

❖ 在 Java 代码中配置 HttpSession 对象的超时时间

```
1 HttpSession session = request.getSession();  
2 session.setMaxInactiveInterval(15 * 60); // 设置会话超时为15分钟
```

❖ 在 Web 配置文件中进行会话超时配置¹

```
1 <session-config>  
2   <session-timeout>900</session-timeout>  
3 </session-config>
```

20.3 Servlet 配置对象

20.3.1 Servlet 配置对象 ServletConfig

- Java EE 为取得 Servlet 的配置信息，提供一个 Servlet 配置对象 API。
- 该对象在 Servlet 初始化阶段由 Web 容器实例化，并将当前 Servlet 的配置数据写入到此对象，供 Servlet 读取使用。

20.3.2 配置对象的类型和取得

配置对象类型 javax.servlet.ServletConfig 是一个接口，具体实现类由容器厂商实现。

¹推荐在 web.xml 中配置会话超时。

ServletConfig 对象在 Servlet 的 init 方法中取得，由 Web 容器以参数方式注入到 Servlet：

```
1 private ServletConfig config = null;  
  
3 public void init(ServletConfig config) throws ServletException {  
4     super.init(config);  
5     this.config = config;  
6 }
```

1. 要取得 ServletConfig 对象需要重写 init 方法，并传递 ServletConfig 参数。然后在 doGet 和 doPost 方法中即可以使用 config 对象。
2. 与 ServletContext 和 HttpSession 对象不同，Web 容器为每个 Servlet 实例创建一个 ServletConfig 对象，不同 Servlet 之间无法共享此对象。

20.3.3 ServletConfig 功能和方法

public String getInitParameter(String name)

取得指定的 Servlet 配置参数。与 Web 初始参数不同，Servlet 初始参数在 Servlet 声明中定义。

```
1 <servlet>  
2   <servlet-name>ServletConfigSample</servlet-name>  
3   <servlet-class>ouc.javaee.ServletConfigSample</servlet-class>  
4   <init-param>  
5     <param-name>url</param-name>  
6     <param-value>jdbc:oracle:thin:@210.30.108.5:1521:oracle</param-value>  
7   </init-param>  
8 </servlet>
```

注意 <init-param> 标签要放置在 <servlet-name> 和 <servlet-class> 后，否则编译错误。

❖ 取得配置的 Servlet 初始参数

```
1 String url = config.getInitParameter("url");
```

public Enumeration getInitParameterNames()

取得所有 Servlet 初始化参数。

```
public String getServletName()
```

取得 Servlet 的名称。

👉 public ServletContext getServletContext()

ServletConfig 对象提供了取得 ServletContext 对象的方法，与在 Servlet 内使用 this.getServletContext() 一样，返回 ServletContext 实例的对象引用。

```
1 ServletContext ctx = config.getServletConfig();
```

20.4 转发和重定向

20.4.1 Web 跳转方式

重定向 (redirect)

典型的重定向跳转方式如下：

- 地址栏手工输入新的 URL 地址；
- 单击超链接；
- 提交 FORM 表单；
- 使用响应对象 response 的 sendRedirect() 方法。

重定向跳转方法都是由客户端浏览器来执行的，由此可见重定向增加了网络的访问流量。

转发 (forward)

- 转发是在服务器端进行页面直接跳转的方法。
- 转发是指 Web 组件在服务器端直接请求到另外 Web 组件的方式。

- 转发在 Web 容器内部完成，不需要通过客户端浏览器，因此客户端浏览器的地址还停留在初次请求的地址上。

Web 开发中应该尽量使用转发实现 Web 组件之间的导航。

20.4.2 实现转发

◆ 取得转发对象

转发对象类型为 javax.servlet.RequestDispatcher。

通过请求对象 HttpServletRequest 取得

```
1 RequestDispatcher rd = request.getRequestDispatcher("main.jsp");
```

使用 ServletContext 对象的方法取得

```
1 RequestDispatcher rd = this.getServletContext().getRequestDispatcher("/main.jsp");
```

取得转发对象后，调用转发对象的方法 forward 完成转发。

```
1 RequestDispatcher rd = request.getRequestDispatcher("main.jsp");
2 rd.forward(request, response);
```

目标页面 main.jsp 的目录说明

- 如果上述代码所在的 Servlet 被映射到 /employee/main.action (一个虚拟请求地址)，则转发的目标页面 main.jsp 也需要放到 /employee 目录下；
- 如果 main.jsp 和 Servlet 映射地址不在同一目录，就需要使用相对路径定位，如把 main.jsp 放在 /department/main.jsp， 则取得转发对象需要按照如下示例代码进行修改：

```
1 RequestDispatcher rd =
2 request.getRequestDispatcher("../department/main.jsp");
```

20.4.3 Servlet 之间共享数据的方法总结

- 使用 ServletContext 对象 对象生命周期长，会长时间占用内存。

2. 使用会话对象 对象生命周期较长，会长时间占用内存。
3. 使用请求对象，基于转发传递数据 对象生命周期短，内存会及时释放。

20.4.4 转发与重定向的区别

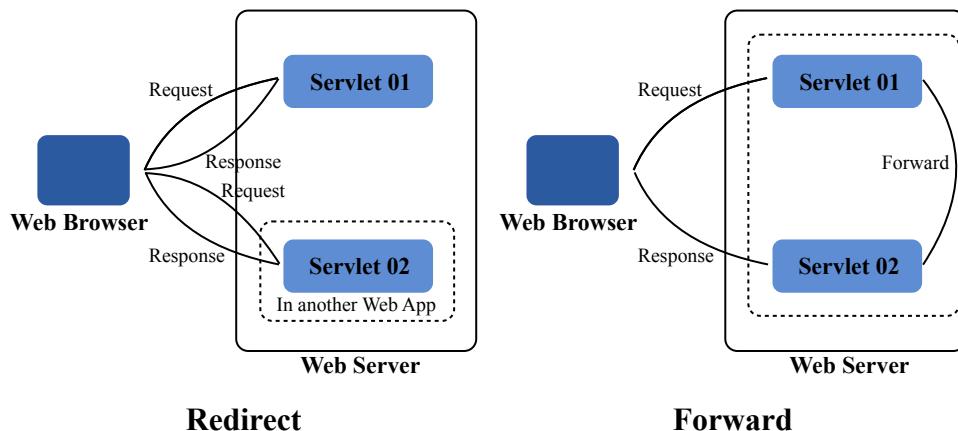


图 20.1 重定向和转发的区别

1. **发生的地点不同** 重定向由客户端完成，而转发由服务器完成。
2. **请求/响应的次数不同** 重定向两次请求，创建两个请求对象和响应对象，而转发是一次请求，只创建一个请求对象和响应对象。重定向无法共享请求/响应对象，而转发可以。
3. **目标位置不同** 重定向可以跳转到 Web 应用以外的文档，而转发只能在一个 Web 内部文件中间进行。

注意

转发之前不应有响应发送，否则导致异常 `javax.servlet.IllegalStateException` 抛出。