

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

大纲      数组的概念      一维数组      二维数组      字符串

## 一维数组

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

**CODE ↪ 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

```

**动态内存分配说明**

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。

**CODE ↪ 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。

### 3.5.1 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"
```

截取子串：前闭后开

```
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"
```

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**。

为什么最好要在类中定义无参构造方法？

根据Java规范，当没有给一个类定义任何构造函数时，编译器会自动补上一个无参构造函数，若有的话就不会，这时就需要将此无参的构造函数写出来。否则，当尝试通过一个无参构造函数来创建对象（使用new关键字）时，编译器就会因为找不到这个无参构造函数而报错。

### 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。

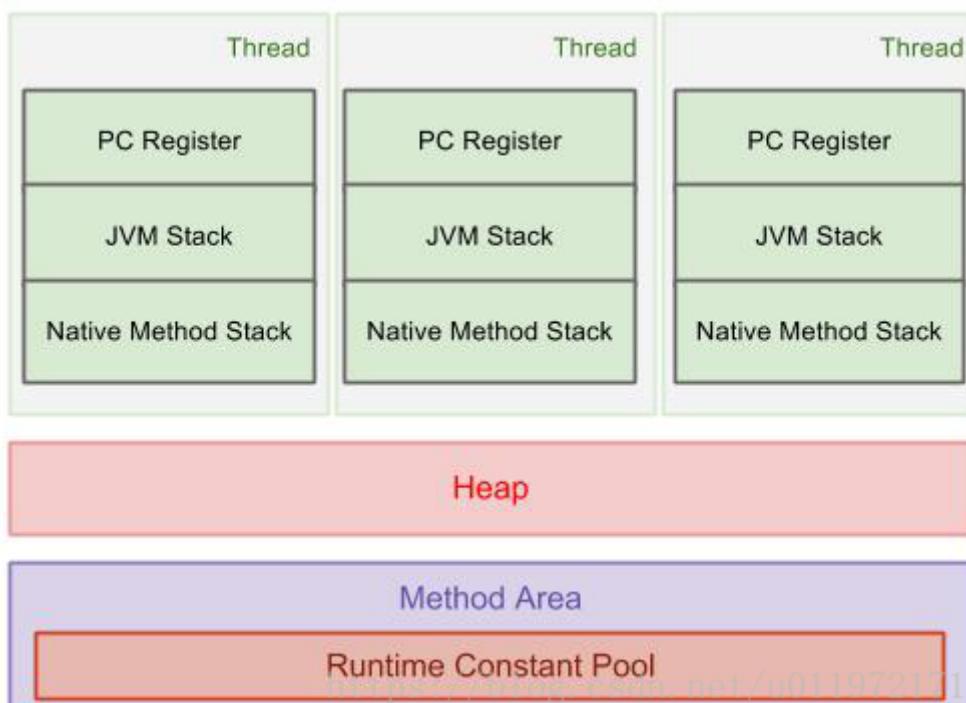
#### 非 static 初始化块

非 static 的初始化块在创建对象时被自动调用。

static 标记的属性或方法由整个类（所有实例）共享

static方法是为了**方便在没有创建对象的情况下进行调用（方法/变量）**。

## Java (JVM) 的内存模型



从这张图中很直观的看到，程序计数器、虚拟机栈、native栈（本地方法栈）是线程私有的，堆和方法区是线程共享的，现在详细介绍JVM各个区块。

**堆：**存放所有new出来的东西。

(1) 堆是java虚拟机所管理的内存区域中最大的一块，java堆是被所有线程共享的内存区域，在java虚拟机启动时创建，堆内存的唯一目的就是存放对象实例，几乎所有的对象实例都在堆内存分配空间。

(2) 堆是垃圾回收机制（GC）管理的主要区域，从垃圾回收的角度看，由于现在的垃圾收集器都是采用的分代收集算法，因此Java堆还可以初步细分为新生代和老年代。

(3) Java虚拟机规定，堆可以处于物理上不连续的内存空间中，只要逻辑上连续的即可。在实现上既可以是固定的，也可以是可动态扩展的。如果在堆内存没有完成实例分配，并且堆大小也无法扩展，就会抛出OutOfMemoryError异常。

**方法区：**存储被虚拟机加载的类信息，常量，静态常量，静态方法，运行时常量池等。

(1) 用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

(2) Sun HotSpot虚拟机把方法区叫做永久代（Permanent Generation），方法区中最重要的部分是运行时常量池（运行时常量池是方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时就会抛出OutOfMemoryError异常。）。

**虚拟机栈：**为虚拟机执行到的Java方法服务。每个方法被调用（执行）的时候都会创建一个栈帧，用于存储局部变量表（包括参数）、操作栈、动态链接、方法出口等信息。

局部变量表存放的是：编译期可知的基本数据类型、对象引用类型。

每个方法被调用直到执行完成的过程，就对应着一个栈帧在虚拟机中从入栈到出栈的过程。声明周期与线程相同，是线程私有的。

在Java虚拟机规范中，对这个区域规定了两种异常情况：

(1) 如果线程请求的栈深度太深，超出了虚拟机所允许的深度，就会出现StackOverFlowError（比如无限递归。因为每一层栈帧都占用一定空间，而Xss规定了栈的最大空间，超出这个值就会报错）。

(2) 虚拟机栈可以动态扩展，如果扩展到无法申请足够的内存空间，会出现OOM(OutOfMemory)。

**本地方法栈：**

为虚拟机执行到的Native方法服务。

(1) 本地方法栈与Java虚拟机栈作用非常类似，其区别是：Java虚拟机栈是为虚拟机执行Java方法服务的，而本地方法栈则为虚拟机执使用到的Native方法服务。

(2) Java虚拟机没有对本地方法栈的使用和数据结构做强制规定，Sun HotSpot虚拟机就把Java虚拟机栈和本地方法栈合二为一。

(3) 本地方法栈也会抛出StackOverFlowError和OutOfMemoryError。

**程序计数器：**是最小的一块内存区域，它的作用是当前线程所执行的字节码的行号指示器，在虚拟机的模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、异常处理、线程恢复等基础功能都需要依赖计数器完成。说的通俗一点，我们知道，Java是支持多线程的，程序先去执行A线程，执行到一半，然后就去执行B线程，然后又跑回来接着执行A线程，那程序是怎么记住A线程已经执

行到哪里了呢？这就需要程序计数器了。因此，为了线程切换后能够恢复到正确的执行位置，每条线程都有一个独立的程序计数器，这块儿属于线程私有的内存。

## Java 中比较操作 “==” 和 “equals()”

操作符 “==” 用来比较两个操作元是否相等，这两个操作元既可以是基本类型，也可以是引用类型。当操作符 “==” 两边都是引用类型变量时，这两个引用变量必须都引用同一个对象，结果才为true。

而equals()方法是在java.lang.Object类中定义的，用来比较两个对象（的值）是否相等。

下面举例说明：

```
String str1 = new String( "Hello" );
```

```
String str2 = new String( "Hello" );
```

str1和str2变量引用不同的String对象，但是它们的内容都是“Hello”，因此用“==”比较的结果为false，而用equals()方法比较的结果为true。

## 抽象类特征

### 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。**

### 抽象类和接口的异同？

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。

一个类只能继承一个抽象类（因为你不可能同时是生物又是非生物）。但是一个类可以同时实现多个接口，比如开车接口，滑冰接口。

1、抽象类和接口都不能被直接实例化，如果二者要实例化，就涉及到多态。如果抽象类要实例化，那么抽象类定义的变量必须指向一个子类对象，这个子类继承了这个抽象类并实现了这个抽象类的所有抽象方法。如果接口要实例化，那么这个接口定义的变量要指向一个子类对象，这个子类必须实现了这个接口所有的方法。

2、抽象类要被子类继承，接口要被子类实现。

3、接口里面只能对方法进行声明，抽象类既可以对方法进行声明也可以对方法进行实现。

4、抽象类里面的抽象方法必须全部被子类实现，如果子类不能全部实现，那么子类必须也是抽象类。接口里面的方法也必须全部被子类实现，如果子类不能实现那么子类必须是抽象类。

5、接口里面的方法只能声明，不能有具体的实现。这说明接口是设计的结果，抽象类是重构的结果。

6、抽象类里面可以没有抽象方法。

7、如果一个类里面有抽象方法，那么这个类一定是抽象类。

8、抽象类中的方法都要被实现，所以抽象方法不能是静态的static，也不能是私有的private。

9、接口（类）可以继承接口，甚至可以继承多个接口。但是类只能继承一个类。

10、抽象级别（从高到低）：接口>抽象类>实现类。

11、抽象类主要是用来抽象类别，接口主要是用来抽象方法功能。当你关注事物的本质的时候，请用抽象类；当你关注一种操作的时候，用接口。

12、抽象类的功能应该要远多于接口，但是定义抽象类的代价较高。因为高级语言一个类只能继承一个父类，即你在设计这个类的时候必须要抽象出所有这个类的子类所具有的共同属性和方法；但是类（接口）却可以继承多个接口，因此每个接口你只需要将特定的动作方法抽象到这个接口即可。也就是说，接口的设计具有更大的可扩展性，而抽象类的设计必须十分谨慎。

类型	abstract class	Interface
定义	abstract class关键字	Interface关键字
继承	抽象类可以继承一个类和实现多个接口；子类只可以继承一个抽象类	接口只可以继承接口（一个或多个）；子类可以实现多个接口
访问修饰符	抽象方法可以有public、protected和default这些修饰符	接口方法默认修饰符是public。你不可以使用其它修饰符
方法实现	可定义构造方法，可以有抽象方法和具体方法	接口完全是抽象的，没构造方法，且方法都是抽象的，不存在方法的实现
实现方式	子类使用extends关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现	子类使用关键字implements来实现接口。它需要提供接口中所有声明的方法的实现
作用	了把相同的东西提取出来，即重用	为了把程序模块进行固化的契约，是为了降低偶合

如果你想让一些方法有默认实现，那么使用抽象类吧。

如果你想实现多重继承，那么你必须使用接口。

匿名内部类是局部类的一种简化，省略了类的名字。

# 匿名类的语法

这里举一个简单的例子：

```
Runnable hello = new Runnable() {
    public void run() {
        System.out.println("hello");
    }
};
```

一个匿名类由以下几个部分组成：

1. new操作符
2. Runnable：接口名称。这里还可以填写抽象类、普通类的名称。
3. ()：这个括号表示构造函数的参数列表。由于Runnable是一个接口，没有构造函数，所以这里填一个空的括号表示没有参数。
4. {...}：大括号中间的代码表示这个类内部的一些结构。在这里可以定义变量名称、方法。跟普通的类一样。

## 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。

常见异常分类：

错误的类型转换；

数组下标越界；

空指针访问。

## ❖ 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

### ☛ IOException 示例 1

课程配套代码 ➔ sample.exception.IOExceptionSample.java

<sup>1</sup> 上述代码无法编译：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。



## Java 异常处理的原则

- ▶ 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

所谓的「异常处理机制」就是能够在你出现逻辑错误的时候，尽可能的为你返回出错信息以及出错的代码大致位置，方便你排查错误。

大纲      异常的概念及分类      Java 异常处理机制

## Java 异常处理机制

- ▶ 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。
- ▶ 程序员通常对错误（Error）无能为力，因而一般只处理违例（Exception）。

异常处理中 `finally` 语句的作用：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。

无论是否捕获或处理异常，`finally` 块里的语句都会被执行。当在 `try` 块或 `catch` 块中遇到 `return` 语句时，`finally` 语句块将在方法返回之前被执行。

### 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：

```
>java -Dmffff=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。

#### 遍历、操作系统属性

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

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

##### ❖ 可用方法

`Enumeration propertyNames()`

`String getProperty(String key)`

`Object setProperty(string key, String value)`

`void load(InputStream inStream)`

`void store(OutputStream out, String header)`

Java	功能说明
<code>list()</code>	列出目录下文件。
<code>renameTo()</code>	修改文件名。
<code>exists()</code>	用于判断文件或目录是否存在。
<code>createNewFile()</code>	用于创建新的文件。
<code>mkdir()</code>	用于创建新的目录。
<code>delete()</code>	用于删除当前文件或目录。
无	复制文件。
无	转移文件。
无	比较文件内容。

Java的编译器是 `javac(.exe)`、Java的执行器/解释器是 `java(.exe)`

JVM负责运行字节码：JVM把每一条要执行的字节码交给解释器，翻译成对应的机器码，然后由解释器执行。JVM解释执行字节码文件就是JVM操作Java解释器进行解释执行字节码文件的过程。

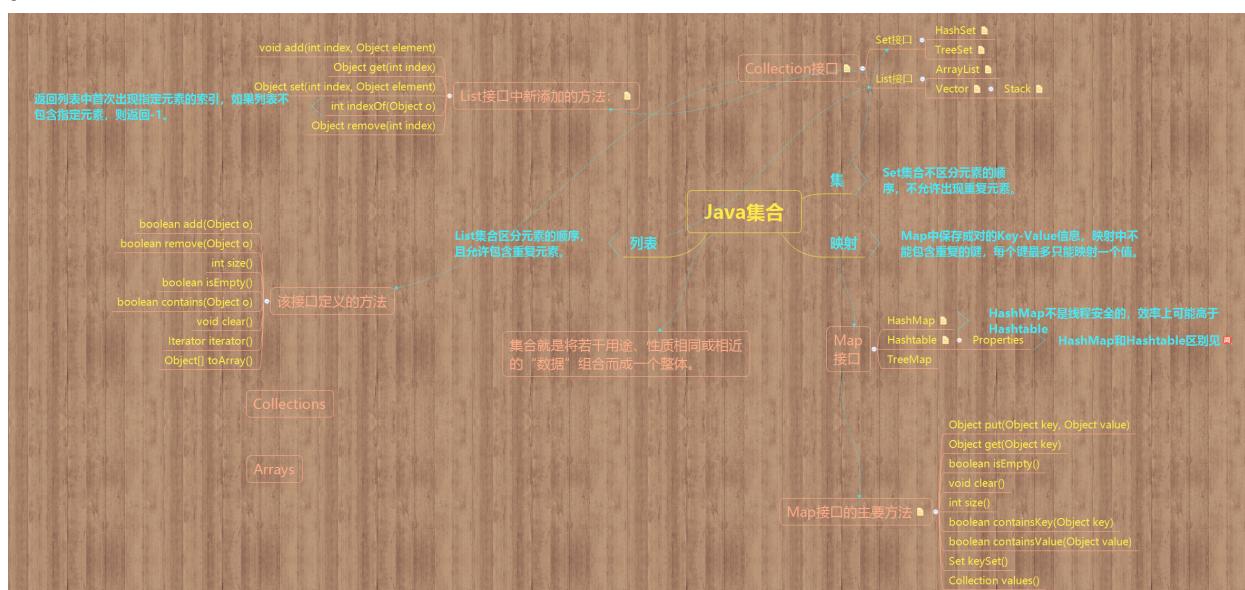
Java编译器：将Java源文件（.java文件）编译成字节码文件（.class文件，是特殊的二进制文件，二进制字节码文件），这种字节码就是JVM的“机器语言”。javac.exe可以简单看成是Java编译器。

Java解释器：是JVM的一部分。Java解释器用来解释执行Java编译器编译后的程序。

java.exe可以简单看成是Java解释器。

注意：通常情况下，一个平台上的二进制可执行文件不能在其他平台上工作，因为此可执行文件包含了对目标处理器的机器语言。而Class文件这种特殊的二进制文件，是可以运行在任何支持Java虚拟机的硬件平台和操作系统上的！

## Java集合容器



## ITERATOR 接口

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。

15

28

## 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。

CODE ↗ TestIterator.java

```
1 import java.util.ArrayList;
2 import java.util.Iterator;
3
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();
11
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。

16

28

## 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

- 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。

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。

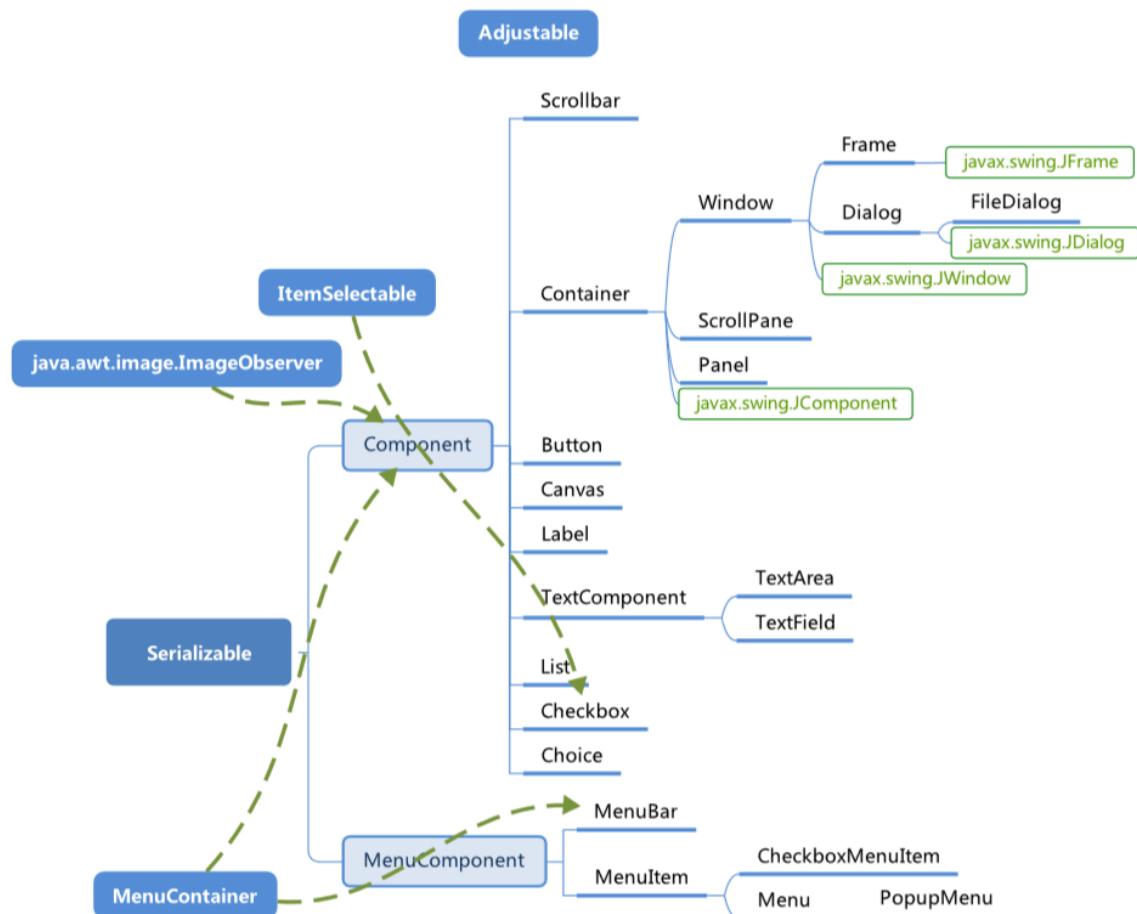


图 11.1 AWT 组件和容器层次架构

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

泛型：

泛型，即“参数化类型”。

作用：一、提高了Java程序的类型安全性（让问题尽可能在编译期就能抓到，而不会等到执行期才冒出来）；

二、消除了强制类型转换；

线程是程序内部的顺序控制流。

让CPU在同一个时间之内执行一个程序中的好几个程序段来完成工作，这就是多线程的概念。

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。[PS:对照P206—208的示例代码]

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。

#### 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：

# 线程的生命周期

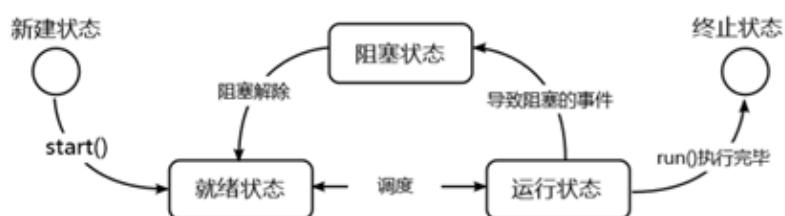
**新建状态** 调用 Thread 构造方法，未显式调用 start() 方法前；

**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。



**新建状态** 的线程对象已被分配了内存空间和其他资源，并已被初始化，但尚未被调度。一旦被调度，即变成就绪状态。

**就绪状态**（又称可运行状态），已具备运行的条件。原来处于阻塞状态的线程被解除阻塞后也将进入就绪状态。

就绪状态的线程被调度并获得 CPU 资源时，便进入**运行状态**。运行状态表示线程正在运行并且已经拥有了对 CPU 的控制权。当线程对象被调度执行时，它将自动调用本对象的 run() 方法，从该方法的第一条语句开始执行，一直到运行完毕，除非该线程主动让出 CPU 的控制权。

处于运行状态的线程在下列情况下将让出 CPU 的控制权 (\*):

- 一、线程运行完毕。
- 二、有比当前线程优先级更高的线程处于就绪状态。
- 三、线程主动睡眠一段时间。
- 四、线程在等待某一资源。

一个正在执行的线程如果让出 CPU 并暂时中止自己的执行，线程处于这种不可运行的状态被称为**阻塞状态**。该状态下，即使 CPU 空闲也不能执行线程。

下面几种情况可使一个线程进入阻塞状态：

- 一、调用 sleep()或 yield()方法；
- 二、为等待一个条件变量，线程调用 wait()方法；
- 三、该线程与另一个线程 join()在一起。

只有当引起阻塞的原因被消除时，阻塞状态的线程才可以转入就绪状态，重新进到线程队列中排队等待 CPU 资源，以便从原来的暂停处继续运行。

处于**消亡状态**的线程不具有继续运行的能力。

导致线程消亡的原因有两个：

- 一、正常运行的线程完成了它的全部工作，即执行完了 run()方法的最后一条语句并退出；
- 二、当进程因故停止运行时，该进程中的所有线程将被强行终止。

#### 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）

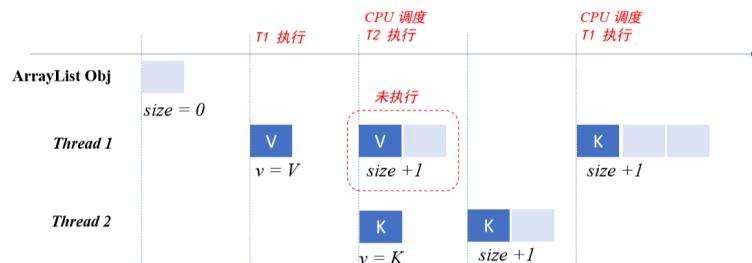


图 10.2 线程安全

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。

---

## Java I/O原理：

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）。

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

## 13.2 基础 I/O 流

### 13.2.1 InputStream and OutputStream

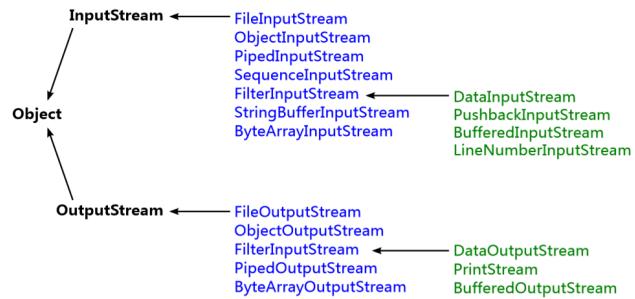


图 13.2 InputStream 和 OutputStream

Java的类加载机制及其特点：

JVM类加载机制分为五个部分：加载，验证，准备，解析，初始化

大纲

反射

类的加载、连接和初始化

类加载器

使用反射生成并操作对象

## 类加载机制

### ◆ 关于类加载机制的几点说明

1. 全盘负责：当一个类加载器负责加载某个 Class 时，该 Class 所依赖和引用的其它 Class 也将由该类加载器负责载入，除非显式使用另一个类加载器载入。
2. 父类委托：先让父类加载器试图加载该 Class，只有父类加载器无法加载该类时才尝试从自己的类路径中加载该类。
3. 缓存机制：类加载器先从缓存中搜索 Class，只有当缓存中不存在该 Class 对象时，系统才会重新读取该类对应的二进制数据。



## 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。



## 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。

doGet 和 doPost 方法都接收 Web 容器自动创建的请求对象和响应回对象，使得 Servlet 能够解析请求数据和发送响应给客户端。

## 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，自动调用 doGet() 或 doPost() 方法。将请求对象和响应回对象传给 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 响应回对象向浏览器发送响应信息。



- ▶ Servlet 必须在 Web 的配置文件/**WEB-INF/web.xml** 中进行配置（声明和映射）才能响应 HTTP 请求。

## ❖ 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。

```
<servlet>
```

```
  <servlet-name> servlet的名字 </servlet-name>          //servlet的名字与
  Servlet声明中的名称要一致。
  <servlet-class> servlet的路径 </servlet-class>
  <init-param>
    <param-name> 初始化参数名字 </param-name>
    <param-value> 初始化参数值 </param-value>
  </init-param>
  <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。
```

```
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name> servlet的名字 </servlet-name>
    <url-pattern> /发布到服务器的名字 </url-pattern>
  </servlet-mapping>
```

采用注解 @WebServlet("/hello") 实现 Servlet 与 URL 的映射

@WebServlet(name = "", urlPatterns = {"/my"}) —— 通过 WebServlet 注解类型来声明一个 Servlet。

name 属性是可选的，如果有一般是用来提供 Servlet 类的名称，关键是 urlPatterns 属性，也是可选的，但是几乎都会用到它，通过 /my 调用这个 Servlet。

如果同时使用注解和 web.xml 部署 Servlet 时，注解会不起作用。

Web容器在启动时，它会为每个Web应用程序都创建一个表达Web应用环境的对象（即 ServletContext），其生命周期与Web应用相同。

所有客户端和Web组件都能共享ServletContext对象，进而取得Web应用的基本信息。ServletContext对象可以通过this.getServletContext()方法获得其对象的引用。

### Servlet编程实例：

```
16  public class HelloServlet extends HttpServlet {
17      private static final long serialVersionUID = 1L;
18
19      public HelloServlet() {
20          super();
21      }
22
23      /**
24      * doGet方法，处理HTTP GET请求
25      */
26      protected void doGet(HttpServletRequest request, HttpServletResponse response)
27          throws ServletException, IOException {
28          response.getWriter().append("Served at: ").append(request.getContextPath());
29      }
30
31      /**
32      * doPost方法，处理HTTP POST请求
33      */
34      protected void doPost(HttpServletRequest request, HttpServletResponse response)
35          throws ServletException, IOException {
36          doGet(request, response);
37      }
38
39  }
```

Github LoginServlet

### 会话跟踪的方法：

## 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。

一般 3 和 4 组合使用。



### 19.2.1 URL 重写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

```
1 <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。

```
1 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。

```
1 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

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。

### 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();  
  
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）

# 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。

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。

## 过滤器编程：

```
20  @WebFilter(filterName = "LoginVerificationFilter", urlPatterns = { "/index/*", "/admin/*" })
21  public class LoginVerificationFilter implements Filter {
22
23      public LoginVerificationFilter() {
24
25
26          public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
27              throws IOException, ServletException {
28              HttpServletRequest req = (HttpServletRequest) request;
29              HttpServletResponse resp = (HttpServletResponse) response;
30
31              System.out
32                  .println(req.getServletPath().substring(1) + " have passed filter named " + this.getClass().getName());
33
34              // 如果Session对象中保存了用户名，则认为用户已经登录；否则重定向到登录页面
35              if (req.getSession().getAttribute("username") != null) {
36                  chain.doFilter(request, response);
37              } else {
38                  resp.sendRedirect("../user/login.html");
39              }
40          }
41
42          public void init(FilterConfig fConfig) throws ServletException {
43
44      }
45
46          public void destroy() {
```

```

15  /*
16  @WebFilter(filterName = "UserValidateFilter", urlPatterns = "/*", initParams = {
17      @WebInitParam(name = "EXCLUDED_PAGES", value = "login.html;login.jsp;bootstrap/4.1.3/css/bootstrap.min.css") })
18  */
19  public class UserValidateFilter implements Filter {
20
21      private String excludedPages;
22      private String[] excludedPageArray;
23
24      public UserValidateFilter() {
25
26      }
27
28      public void init(FilterConfig config) throws ServletException {
29          excludedPages = config.getInitParameter("EXCLUDED_PAGES");
30
31          if (null != excludedPages && excludedPages.length() != 0) {
32              excludedPageArray = excludedPages.split(String.valueOf(';'));
33          }
34      }
35
36      public void destroy() {
37          this.excludedPages = null;
38          this.excludedPageArray = null;
39      }
40
41      public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
42          throws IOException, ServletException {
43
44          HttpServletRequest req = (HttpServletRequest) request;
45          HttpServletResponse resp = (HttpServletResponse) response;
46
47          boolean isExcludedPage = false;
48
49          System.out.println(req.getServletPath().substring(1) + " have passed filter named " + this.getClass());
50
51          // 遍历例外URL数组，判断当前URL是否与例外页面相同
52          for (String page : excludedPageArray) {
53              if (req.getServletPath().substring(1).equals(page)) {
54                  System.out.println(page + ", you're excluded.");
55                  isExcludedPage = true;
56                  break;
57              }
58          }
59
60          if (isExcludedPage || req.getSession().getAttribute("username") != null) {
61              chain.doFilter(request, response);
62          } else {
63              resp.sendRedirect("login.html");
64          }
65      }
66  }
67 
```

## 配置过滤器：

[大纲](#) [过滤器概述](#) [Java EE 过滤器 API](#) [Java EE 过滤器编程和配置](#) [过滤器的主要任务](#) [过滤器应用实例：用户登录验证和权限](#)

### 过滤器编程示例

配置过滤器，在 Web 应用的配置文件/WEB-INF/web.xml 中配置声明和过滤 URL 地址映射。

#### ❖ 过滤器声明

```

1  <filter>
2      <description>此过滤器完成对请求数据编写及进行修改</description>
3      <display-name>字符集编码过滤器</display-name>
4      <filter-name>EncodingFilter</filter-name><!-- * -->
5      <filter-class>javaee.ch08.CharEncodingFilter</filter-class><!-- * -->
6      <init-param>
7          <description>Content Type</description>
8          <param-name>contentType</param-name>
9          <param-value>text/html</param-value>
10         </init-param>
11         <init-param>
12             <description>Encoding</description>
13             <param-name>encoding</param-name>
14             <param-value>utf-8</param-value>
15         </init-param>
16     </filter>

```



## 过滤器映射语法:

```
1 <filter-mapping>
2   <filter-name>EncodingFilter</filter-name>
3   <servlet-name>SaveCookie</servlet-name>
4   <servlet-name>GetCookie</servlet-name>
5   <url-pattern>/employee/add.do</url-pattern>
6   <url-pattern>/admin/*</url-pattern>
7   <dispatcher>FORWARD</dispatcher>
8   <dispatcher>INCLUDE</dispatcher>
9   <dispatcher>REQUEST</dispatcher>
10  <dispatcher>ERROR</dispatcher>
11 </filter-mapping>
```

对过滤器映射标记的说明:

<filter-mapping> 与 <filter> 标记平级，且在 <filter> 之后，即先声明后映射的原则。

<filter-name> 应该与过滤器声明中的 <filter-name> 一致。

<url-pattern> 过滤器映射地址声明，每个过滤器映射可以定义多个。

<servlet-name> 指示过滤器对指定的 Servlet 进行过滤，每个过滤器映射可以定义多个。

<dispatcher> 从 Servlet API 2.4 开始，过滤器映射增加了根据请求类型有选择的对映射地址进行过滤，提供标记 <dispatcher> 实现请求类型的选择。

- ▶ REQUEST 当请求直接来自客户时，过滤器才工作。
- ▶ FORWARD 当请求是来自 Web 组件转发到另一个组件时，过滤器工作。
- ▶ INCLUDE 当请求来自 include 操作时，过滤器生效。
- ▶ ERROR 当转发到错误页面时，过滤器起作用。

监听器接口	引入版本	监听器事件
ServletContextListener	2.3	ServletContextEvent
ServletContextAttributeListener	2.3	ServletContextAttributeEvent
HttpSessionListener	2.3	HttpSessionEvent
HttpSessionActivationListener	2.3	HttpSessionEvent
HttpSessionBindingListener	2.3	HttpSessionBindingEvent
ServletRequestListener	2.4	ServletRequestEvent
ServletRequestAttributeListener	2.4	ServletRequestAttributeEvent

1. ServletContext 对象监听器
2. ServletContext 对象属性监听器
3. HttpSession 对象监听器
4. HttpSession 对象属性监听器
5. HttpServletRequest 对象监听器
6. HttpServletRequest 属性监听器

MVC设计模式：

## JSP 方式

### ❖ 仅有的一点优势

1. 无需额外的配置文件，无需框架的帮助，即可完成逻辑。
2. 简单易上手。

### ❖ 劣势

1. Java 代码由于混杂在一个 HTML 环境中而显得混乱不堪，可读性非常差。一个 JSP 文件有时候会变成几十 K，甚至上百 K，经常难以定位逻辑代码的所在。
2. 编写代码时非常困惑，不知道代码到底应该写在哪里，也不知道别人是不是已经曾经实现过类似的功能，到哪里去引用。
3. 突然之间，某个需求发生了变化。于是，每个人蒙头开始全程替换，还要小心翼翼的，生怕把别人的逻辑改了。
4. 逻辑处理程序需要自己来维护生命周期，对于类似数据库事务、日志等众多模块无法统一支持。



## 框架方式

- ▶ 时代进一步发展，人们发现简单的 JSP 和 Servlet 已经很难满足人们懒惰的要求。于是，人们开始试图总结一些公用的 Java 类，来解决 Web 开发过程中碰到的问题。这时，横空出世了一个框架，叫做**Struts**。它非常先进地实现了**MVC 模式**，成为了广大程序员的福音。
- ▶ 在一定程度上，Struts 能够解决 Web 开发中的职责分配问题，使得显示与逻辑分开。
- ▶ 不过开始的在很长一段时间里，学习使用 Struts 的程序员往往无法清晰的明白我们到底需要 Web 框架帮我们做什么，我们到底需要它完成点什么功能。



#### 24.1.4 那么我们需要什么？

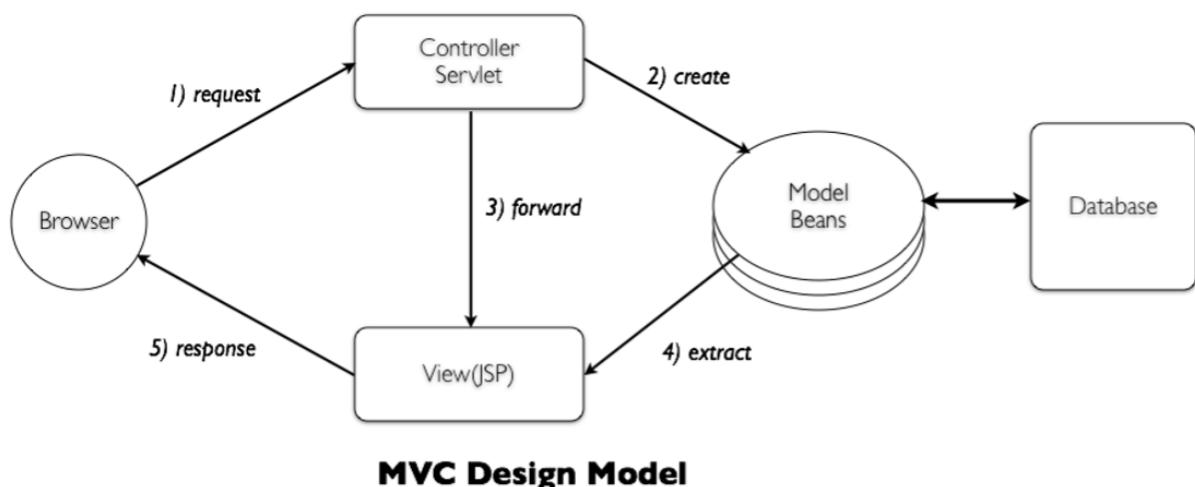
在回顾写代码的历史之后，回头来看看，我们到底需要什么？

无论是使用 JSP，还是使用 Struts1，或是 Struts2，我们至少都需要一些必须的元素（如果没有这些元素，或许我还真不知道这个程序会写成什么样子）：

1. **数据** 在用户登录实例中就是 name 和 password。他们共同构成了程序数据的核心载体。事实上，我们往往会有个 User 类来封装 name 和 password，这样会使得我们的程序更加 OO。无论怎么说，数据会穿插在这个程序的各处，成为程序运行的核心。
2. **页面展示** 例如用户登录页面 login.jsp。没有这个页面，一切的请求、验证和错误展示也无从谈起。在页面上，我们需要利用 HTML，把我们需要展现的数据都呈现出来。同时我们也需要完成一定的页面逻辑，例如，错误展示，分支判断等。
3. **处理具体业务的场所** 不同阶段，处理具体业务的场所就不太一样。原来用 JSP 和 Servlet，后来用 Struts1 或者 Struts2 的 Action。

- ▶ **数据 ↡ Model**
- ▶ **页面展示 ↡ View**
- ▶ **处理具体业务的场所 ↡ Control**

- 1、**数据** 数据会穿插在这个程序的各处，成为程序运行的核心。
- 2、**页面展示** 在页面上把需要展现的数据都呈现出来，同时需要完成一定的页面逻辑。
- 3、**处理具体业务的场所**



作用：MVC分层有助于管理复杂的应用程序，不同的开发人员可同时开发视图、控制器逻辑和业务逻辑，从而简化了分组开发。

优点：代码重用性高；方便维护；耦合性低；部署快；不同的层各司其职，有利于软件工程化管理

缺点：没有明确定义；不适合小型规模的应用程序；增加系统结构和实现的复杂性

## 单例模式

大纲

多态性

方法重载

关键字 static

关键字 final

## Singleton 设计模式

### ❖ Singleton 代码的特点

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。

### ❖ Singleton 的使用场景

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。

