目录

[51. 内 部 类 （innerClass） 2](#_Toc510541131)

[1．非静态内部类 2](#_Toc510541132)

[2．静态内部类 4](#_Toc510541133)

[52. 数组 \_ 内存分析 6](#_Toc510541134)

[1. 数组概述 6](#_Toc510541135)

[53. 数组 \_ 数组的三种初始化方式 8](#_Toc510541136)

[1. 一维数组的声明 8](#_Toc510541137)

[2. 一维数组的初始化方式： 8](#_Toc510541138)

[54. 数组 \_ String \_ JDK源码分析 10](#_Toc510541139)

[55. String类的补充 \_ 常见面试题和内存分析 12](#_Toc510541140)

[56. StringBuilder和StringBuffer的使用（方法链的实现和JDK源码分析） 13](#_Toc510541141)

[57. StringBuilder和StringBuffer的使用（JDK源码分析内部机制） 15](#_Toc510541142)

[58. StringBuilder和StringBuffer的使用（常见方法补充和常见面试题答法） 16](#_Toc510541143)

[59. 模拟ArrayList容器的底层实现（JDK源码分析ArrayList） 18](#_Toc510541144)

[Plus：补充：关于ArrayCopy 21](#_Toc510541145)

[60. 多维数组基本语法和内存分析 22](#_Toc510541146)

[61. 多维数组练习矩阵运算 （Matrix） 24](#_Toc510541147)

[62. 数组的拷贝、排序、二分法、命令行参数以及增强for循环 25](#_Toc510541148)

[1. 数组的拷贝 25](#_Toc510541149)

[2. 数组排序 26](#_Toc510541150)

[3. 二分法查找 27](#_Toc510541151)

[4. Arrays替换填充 28](#_Toc510541152)

[5. 常见排序和二分法查找（重要！！面试前复习！！） 28](#_Toc510541153)

[6. 命令行参数的问题 String[] args 30](#_Toc510541154)

[63. Java常用类（JDK源码分析） 32](#_Toc510541155)

[64. 自动装箱和拆箱（Auto-boxing & Unboxing） 35](#_Toc510541156)

[1. 自动装箱： 35](#_Toc510541157)

[2. 自动拆箱： 35](#_Toc510541158)

[3. 缓存问题 35](#_Toc510541159)

[65. Date类的使用（JDK源码分析） 36](#_Toc510541160)

[1. 时间处理相关类： 36](#_Toc510541161)

[2. Date时间类(java.util.Date)： 36](#_Toc510541162)

## 51. 内 部 类 （innerClass）

### 1．非静态内部类

* **必须寄存在一个外部类对象中。**
* **非静态内部类的对象单独属于外部类的某个对象。（调用起来比较特殊）**

|  |
| --- |
| **---------------------------------非静态内部类---------------------------------------**  **public** **class** Outer {  **public** **static** **void** main(String[] args) {  //不能直接调用Nose nose = new Nose();  //必须先调用外部类对象，在调用内部类  Face f = **new** Face();  Nose n = f.**new** Nose();//必须导入Face.Nose的包  Face.Nose n2 = f.**new** Nose();//不用导入Face.Nose的包  n.breath();  n2.breath();  }  }  **class** Face{  **int** type;  **class** Nose{ //内部类Nose  String type;  **void** breath() {  System.***out***.println("呼吸！");  }  }  }  **-------------------------------------运行结果---------------------------------------**    **-------------------------------------生成的文件-------------------------------------** |

* **非静态内部类可以使用外部类的成员，但是外部类不能直接访问非静态内部类的成员。**

|  |
| --- |
| **-----------------------------非静态内部类调用外部成员-------------------------------**  **class** Face{  **int** type=20;  String shape = "大饼脸";  **class** Nose{  String type="鹰钩鼻";  **void** breath() {  //访问的是外部Face的type  System.***out***.println(Face.**this**.type);  //访问的是内部Nose的type  System.***out***.println(**this**.type);  //若不存在属性名冲突，则调用可以直接使用属性名  System.***out***.println(shape);  }  }  }  **-------------------------------------运行结果-------------------------------------** |

* 非静态内部类**不能**有静态方法、静态属性和静态初始化块。
* **静态成员不能访问非静态成员**：外部类的静态方法、静态代码块不能访问非静态内部类，包括不能使用非静态内部类定义变量、创建实例。
* **成员变量访问要点：**

1. 内部类里方法的局部变量：变量名

2. 内部类属性：this.变量名

3. 外部类属性：外部类.this.变量名

* **内部类的访问：**

1. 外部类中定义内部类：new innerClass()

2. 外部类以外的地方使用非静态内部类：

Outer.inner varname = OuterObject.new.inner();

### 2．静态内部类

* **当一个静态内部类对象存在，并不一定存在对应的外部类对象。**因此，静态内部类的实例方法并不能直接访问外部类的实例方法。**但可以访问外部类的静态成员。**
* **静态内部类看作外部类的一个静态成员（从属于类）。**因此，外部类的方法中可以通过：静态内部类.成员名 访问静态内部类的静态成员。通过new静态内部类() 访问静态内部类的实例。
* 在外部类的外面创建静态内部类：

OuterClass. InnerClass varname = new OuterClass.Innerclass();

|  |
| --- |
| **---------------------------------静态内部类调用外部成员-------------------------------**  **public** **class** Outer {  **public** **static** **void** main(String[] args) {  //外部类直接调用静态内部类  Face.Ear ear = **new** Face.Ear();  ear.listen();  }  }  **class** Face{  **static** String *color* = "yellow";  //静态内部类Ear  **static** **class** Ear{  **void** listen() {  System.***out***.println("我在听！");  //静态内部类可访问外部类静态成员  System.***out***.println(*color*);  }  }  } |

|  |
| --- |
| **总结：**   1. **成员变量、全局变量和局部变量**   **成员变量：定义在类中**  **局部变量：定义在方法中**   1. **非静态内部类：**   **依存于外部类的对象而存在，可以访问外部类的普通成员。**  **创建时必须先创建外部对象，再创建内部对象。**  **OuterClass outer = new OuterClass();**  **OuterClass.InnerClass varname = outer. New InnerClass();**   1. **静态内部类：**   **可以看作是外部类的一个静态成员。可以访问外部类的静态成员。其他类可以直接在没有创建外部成员的情况下直接访问。**  **创建时可以直接创建：**  **OuterClass.InnerClass varname = new OuterClass.Innerclass();** |

## 52. 数组 \_ 内存分析

### 1. 数组概述

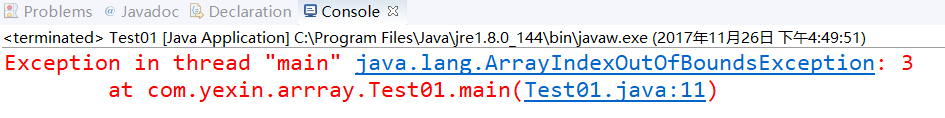
* **数组是相同类型数据的有序集合**

数组描述的是相同类型的若干个数据，按照一定的先后次序排列组合而成。其中，每一个数据称作一个数组元素，每个数组元素可以通过一个下标来访问它们。

* **数组也是对象**，数组元素相当于对象的成员变量。(内存图所示) 数组对象的初始化与成员变量一致（0, false, null, \u0000）。
* **数组长度是确定的**，不可变的。如果越界，则报异常：

ArrayIndexOutOfBoundsException：

**--------------------------------------数组越界异常-------------------------------------**



**--------------------------------------数组代码块--------------------------------------**

**public** **static** **void** main(String[] args) {

//a就是数组，数组属于引用类型，数组也是对象

//new:创建一个数组对象，数组大小为3

**int**[] a = **new** **int**[3];

a[0] = 23;

a[1] = 28;

a[2] = 32;

//数组可以是任意类型，下面是引用类型

Car[] cars = **new** Car[10];

cars[0] = **new** Car("奔驰");

cars[1] = **new** Car("宝马");

//数组长度显示

//array是一个属性，有final修饰，所以不能被赋值

System.***out***.println(a.length);

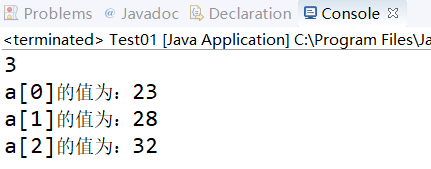
**for**(**int** i=0; i<a.length; i++) {

System.***out***.println("a["+i+"]的值为："+a[i]);

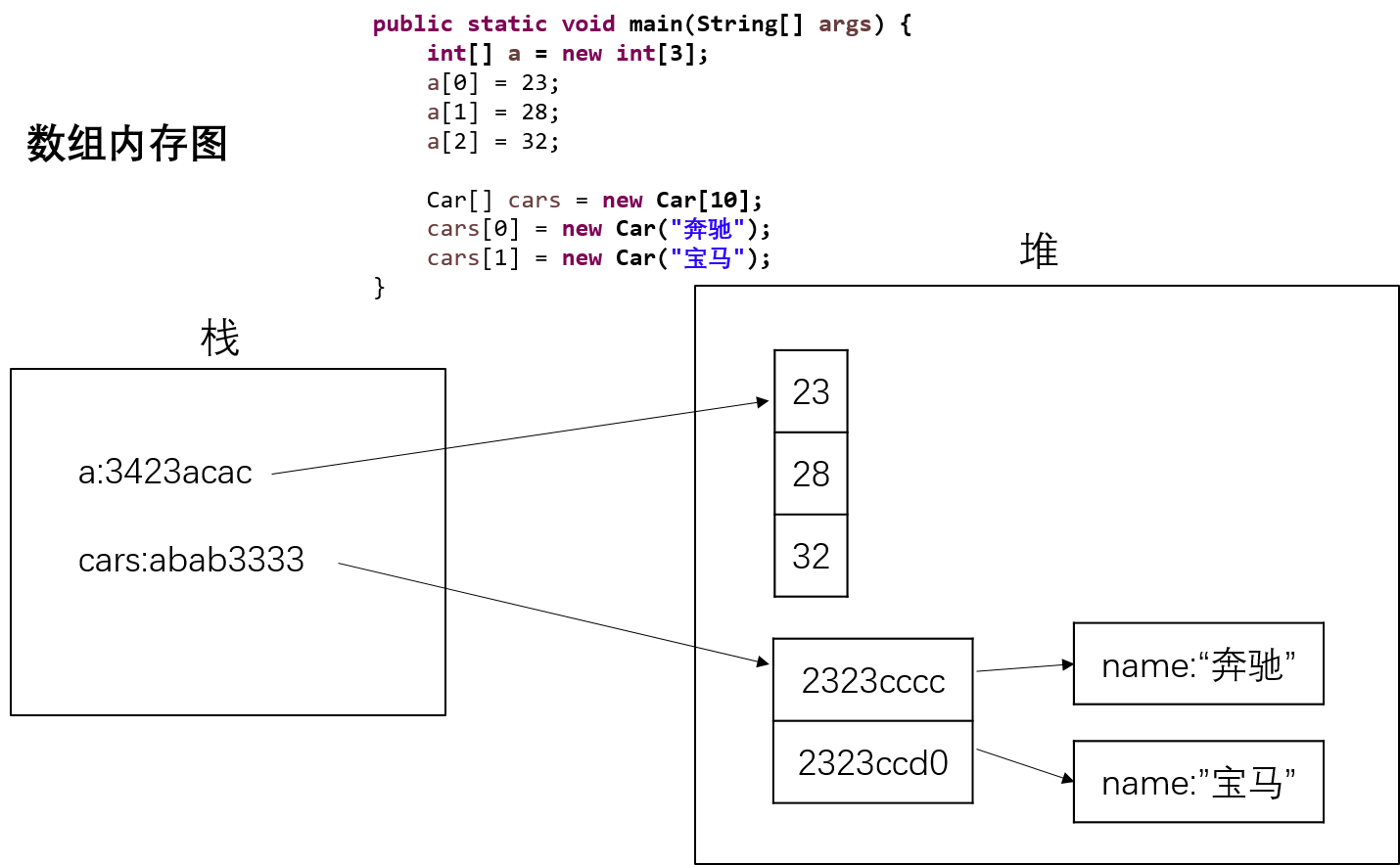
}

}

**----------------------------------数组代码块运行结果-----------------------------------**



**-------------------------------------数组内存分析--------------------------------------**



## 53. 数组 \_ 数组的三种初始化方式

### 1. 一维数组的声明

一维数组的声明有两种方式：

type[] arr\_name; 或 type arr\_name[];

### 2. 一维数组的初始化方式：

* **默认初始化：**与对象的成员变量一致（0，false，null，\u0000）；
* **动态初始化：**一个个单独赋值
* **静态初始化：**全部一起赋值

**---------------------------------数组的声明、创建和初始化-------------------------------**

**public** **static** **void** main(String[] args) {

//数组的声明： "[]"放变量前后都可以

**int**[] a;

**int** b[];

//创建数组

a = **new** **int**[4];

b = **new** **int**[5];

//初始化：对数组元素的初始化

//1.默认初始化：与对象的成员变量一致（0，false，null，\u0000）

//2.动态初始化：就是一个个赋值

**for**(**int** i=0; i<a.length; i++) {

a[i] = i\*12;

}

//3.静态初始化:内存开辟的时候直接赋值

**int** c[] = {23,23,54,56};

Car[] cars = {

**new** Car("奔驰"),

**new** Car("比亚迪"),

**new** Car("宝马")

};

Car c2 = **new** Car("奔驰");

//下面代码输出为false，c2和cars[0]不是一个对象

System.***out***.println(c2 == cars[0]); }

## 54. 数组 \_ String \_ JDK源码分析

**Crtl+String单击可以打开String的JDK源码。打开新世界大门~~**

/\*\*

\* String类的各种方法的测试

\* **@author** YE XIN

\*

\*/

**public** **class** TestString {

**public** **static** **void** main(String[] args) {

//根据String的构造器，也可以向String类中传入char

/\*public String(char value[]) {

this.value = Arrays.copyOf(value, value.length);

}\*/

**char**[] c = {'a','b','c','d'};

String str = **new** String(c);

System.***out***.println(str);//结果：abcd

//charAt:传入i，返回数组第i+1位的值

System.***out***.println(str.charAt(2));//结果：c

//equals:比较对象必须是String

System.***out***.println(str.equals("abcd"));//结果：true

String str3 = "def";

String str4 = "def";

System.***out***.println(str3 == str4);//结果：true 两者指向同一个常量池中的常量

//indexOf:传一个char进来，返回所在位置

System.***out***.println(str.indexOf('c'));//结果：2

System.***out***.println(str.indexOf('y'));//结果：-1 找不到结果

//substring:返回一个子字符串

String trySubString = "unhappy sad";

System.***out***.println(trySubString.substring(2));//结果：happy sad

System.***out***.println(trySubString.substring(2, 7));//结果：happy

//replace：将新char串取代所有的老char

String tryReplace = "happy";

System.***out***.println(tryReplace.replace('p', 'd'));//结果：haddy

System.***out***.println(tryReplace.replace('d', 'p'));//结果：happy,老字符不存在，返回原字符串

//replace方法中，它使用的是先找到第一个oldchar的位置，这样节省时间，算法优化，值得学习

//split：切割字符串，并返回数组

String trySplit = "apple,pineapple,pear,strawberry";

String[] strArray = trySplit.split(",");

**for**(**int** i=0; i<strArray.length; i++) {

System.***out***.println(strArray[i]);

}//结果：apple pineapple pear strawberry

//trim:去除首尾空格

String tryTrim = " aa bb ";

System.***out***.println(tryTrim.trim());//结果：aa bb

System.***out***.println(tryTrim.trim().length());//结果：6

//valueOf:将各类其他数据类型转换为String类型

**int** i = 12345;

System.***out***.println(String.*valueOf*(i));//输出：12345

System.***out***.println("12345".equals(String.*valueOf*(i)));//输出：true 已经转化为String

//equalsIgnoreCase:判断两个String是否相等，忽略大小写

System.***out***.println("ABc".equalsIgnoreCase("abc"));//输出：true

//lastIndexOf:从右边往左边找

System.***out***.println("abcba".indexOf('b'));//输出：1

System.***out***.println("abcba".lastIndexOf('b'));//输出：3

//startsWith/endsWith:判断是否以传入的字符串开头/结尾

System.***out***.println("abcba".startsWith("ab"));//输出:true

System.***out***.println("abcba".endsWith("ba"));//输出：true

//toLowerCase/toUpperCase:全部转为大写/全部转为小写，String是不可变字符序列，所以是新生成的一个字符串

System.***out***.println("Abcba".toLowerCase());//输出：abcba

System.***out***.println("Abcba".toUpperCase());//输出：ABCBA }

}

## 55. String类的补充 \_ 常见面试题和内存分析

* **String是不可变字符串序列，所以每次操作都会生成一个全新的String对象**
* **下面的代码进行的是字符串的累加，这样的累加会创立非常多的String对象，非常浪费空间。这是面试中非常可能出现的题目，也是代码优化的一部分。**

**---------------------------------String里面的一个小问题-------------------------------**

String gh = "a";

**for**(**int** j=0; j<10;j++) {

gh += j;

}

System.***out***.println(gh);//输出：a0123456789 一共创立了11个String对象 非常浪费空间

String gh2 = **new** String("a");

**for**(**int** j=0; j<10;j++) {

gh2 += j;

}

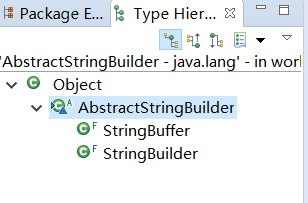
System.***out***.println(gh2);//一共创立了12个String对象

## 56. StringBuilder和StringBuffer的使用（方法链的实现和JDK源码分析）

1. **String是不可变字符串序列，StringBuilder是可变字符序列**
2. **StringBuffer：线程安全的，效率低**

**StringBuilder：线程不安全，效率高（一般用它）**

1. **我们一般针对局部变量，一般使用StringBuilder。**
2. **打开AbstractStringBuilder的Type Hierarchy可以看到继承它的有两个：StringBuilder和StringBuffer，所以可以将StringBuilder和StringBuffer看成是兄弟。**



1. **即使什么都没有传入，new StringBuilder()的时候，就会生成一个容量为16的char型数组：**

/\*\*

\* Constructs a string builder with no characters in it and an

\* initial capacity of 16 characters.

\*/

**public** StringBuilder() {

**super**(16);

}

**6. Append方法结尾return this，因此append可以方法链。**

**---------------------------------StringBuilder相关代码--------------------------------**

**public** **static** **void** main(String[] args) {

StringBuilder sb = **new** StringBuilder();//字符数组长度初始为16

StringBuilder sb1 = **new** StringBuilder(32);//字符数组长度初始为32

StringBuilder sb2 = **new** StringBuilder("abcd");//字符数组长度初始为4+16,value[]={'a','b','c','d',\u0000,\u0000...}

//append累加，可以累加int，char.....大概有十几个append的重写方法。

sb2.append("efg");//将efg累加到数组中

System.***out***.println(sb2.length());//输出：7

sb2.append(**true**);//append对象是一个boolean

System.***out***.println(sb2);//输出：abcdefgtrue

System.***out***.println(sb2.length());//输出：11

//因为StringBuilder中append方法最后都是return this，所以可以实现方法链

sb2.append(**false**).append(321).append("随便 ");

System.***out***.println(sb2); //输出：abcdefgtruefalse321随便

StringBuilder gh2 = **new** StringBuilder("a");

**for**(**int** j=0; j<10;j++) {

gh2 = gh2.append(j);

}

System.***out***.println(gh2);

//输出：a0123456789 整个过程只有两个对象。a字符串一个对象，StringBuilder一个对象

}

## 57. StringBuilder和StringBuffer的使用（JDK源码分析内部机制）

**1. 创建StringBuilder的时候数组的长度设置的是17，但是循环时加了1000个字符，那是如何分配的？**

**----------------------------StringBuilder数组扩展的相关源码----------------------------**

**public** AbstractStringBuilder append(**int** i) {

//...

ensureCapacityInternal(spaceNeeded);

//....

**return** **this**;

}

**private** **void** ensureCapacityInternal(**int** minimumCapacity) {

//如果所需容量大于现在的数组长度，newCapacity扩容

**if** (minimumCapacity - value.length > 0) {

value = Arrays.copyOf(value, newCapacity(minimumCapacity));

//将新的的newCapacity copy到新的value中

}

}

**private** **int** newCapacity(**int** minCapacity) {

//先定义一个新的容量为现有容量\*2+2

**int** newCapacity = (value.length << 1) + 2;

**if** (newCapacity - minCapacity < 0) {

newCapacity = minCapacity;

}

//返回的是newCapcacity

**return** (newCapacity <= 0 || MAX\_ARRAY\_SIZE - newCapacity < 0)

?hugeCapacity(minCapacity)

: newCapacity;

}

* **根据上面源码，如果一开始的StringBuilder数组长度为17，那么进行一次扩展之后，它的长度为17\*2+2（36）。用新数组替代老数组，老数组的对象会被回收掉。Value原来指向老数组，新数组被创建后，value指向新数组。**

**——————通过替换老数组实现数组扩容**

## 58. StringBuilder和StringBuffer的使用（常见方法补充和常见面试题答法）

**--------------------------------StringBuilder方法补充---------------------------------**

//delete方法:传入start和end，删除从start到end的内容

StringBuilder testDelete = **new** StringBuilder("this is a good test");

System.***out***.println(testDelete.delete(10, 15)); //输出：this is a test

//测试ArrayCopy

/\* src the source array.

startPos starting position in the source array.

dest the destination array.

destPos starting position in the destination data.

length the number of array elements to be copied.

\*/

//新数组 = des到destPos为止 + src从startPos开始

Byte[] src = {1,2,3,4,5,6,7};

System.*arraycopy*(src, 1, src, 2, 4);

**for**(**int** i = 0;i< src.length;i++){

System.***out***.print("-> " + src[i]);//输出：-> 1-> 2-> 2-> 3-> 4-> 5-> 7

}

//reverse方法

StringBuilder testReverse = **new** StringBuilder("test reverse");

System.***out***.println(testReverse.reverse());//输出：esrever tset

//StringBuffer:public synchronized int length() ;都有一个synchronized，线程安全

StringBuffer sg = **new** StringBuffer();

//StringBuffer其他和StringBuilder一模一样

**1. String和StringBuilder的区别是什么？**

* **String是不可变字符序列，StringBuilder是可变字符序列**
* **为什么String不能变？因为value前面加了private，没有暴露给外界。加了final，不能更改。**
* **StringBuilder的内存分析，先建立一个16位的char型数组**
* **扩展到动态扩容，如果容量不够，现有容量\*2+2，指向新的数组，旧的数组将被回收**

**2. 自己写的reverse方法**

//自己写的reverse方法，缺点：会创建src2.length()个Byte数组

Byte[] src2 = {1,2,3,4,5,6,7};

Byte[] dest = **new** Byte[7];

**for**(**int** i = 6;i>-1;i--) {

System.*arraycopy*(src2, i, dest, src.length-i-1, 1);

}

**for**(**int** i = 0;i<dest.length;i++) {

System.***out***.print("--"+dest[i]);

}

//自己写的reverse方法2：只需要两个数组即可

**int**[] src3 = {1,2,3,4,5};

**int**[] dest3 = **new** **int**[src3.length];

**for**(**int** i=0;i<src3.length;i++) {

dest3[src3.length-i-1]=src3[i];

}

**for**(**int** i = 0;i<dest3.length;i++) {

System.***out***.print("--"+dest3[i]);

}

## 59. 模拟ArrayList容器的底层实现（JDK源码分析ArrayList）

**1. 类比StringBuilder，其实是实现了一个char[] value；char的数组value**

**那么能不能定义一个类，实现Object[] elements。这个类就是ArrrayList。**

**2. ArrayList属于容器，很多容器的底层实现就是数组。**

**3. ArrayList位于Java的Util包**

**4. 以下是自己实现的ArrayList**

**--------------------------------自己实现的ArrayList---------------------------------**

**package** com.yexin.myCollection;

/\*\*

\* 模拟实现JDK中提供的ArrayList类

\* **@author** IPLAB

\*

\*/

**public** **class** MyArrayList {

/\*\*

\* The value is used for object storage.

\*/

**private** Object[] value;

/\*\*

\* The size is the number of object used.

\*/

**private** **int** size;

**public** **int** size() {

**return** size;

}

/\*\*

\* 空构造器，16位的Object

\*/

**public** MyArrayList(){

//value = new Object[16];

**this**(10);//调用另一个构造器

}

/\*\*

\* 构造器，构造指定size的ArrayList

\* **@param** size

\*/

**public** MyArrayList(**int** size) {

**if**(size<0) {

**try** {

**throw** **new** Exception(); //手动抛出一个异常。讲到异常章节再说，先混个眼熟

}**catch** (Exception e){

e.printStackTrace();

}

}

value = **new** Object[size];

}

/\*\*

\* 判断ArrayList是否为空

\* **@return**

\*/

**public** **boolean** isEmpty() {

**return** size == 0;

}

/\*\*

\* 获得指定Object的index值

\* **@param** obj

\* **@return** 如果存在，返回index

\* 如果不存在，返回-1

\*/

**public** **int** indexOf(Object obj) {

**if**(obj == **null**) {

**return** -1;

}**else** {

**for**(**int** i=0;i<value.length;i++) {

**if**(obj==value[i]) {

**return** i;

}

}

**return** -1;

}

}

/\*\*

\* 倒着遍历判断最后一个存在的值

\* **@param** obj

\* **@return** index 如果存在

\* -1 所查Object为空 / 不存在

\*/

**public** **int** lastIndexOf(Object obj) {

**if**(obj == **null**) {

**return** -1;

}**else** {

**for**(**int** i=value.length-1;i>=0;i--) {

**if**(obj==value[i]) {

**return** i; //return方法：1.返回值 2.停止，所以不会继续执行

}

}

**return** -1;

}

}

/\*\*

\* 索引值是否在数组长度之内

\* **@param** index

\*/

**public** **void** rangeCheck(**int** index) {

**if**(index<0||index>size-1) {

**try** {

**throw** **new** Exception(); //手动抛出一个异常。讲到异常章节再说，先混个眼熟

}**catch** (Exception e){

e.printStackTrace();

}

}

}

/\*\*

\* 是否包含传入的Object

\* **@param** obj

\* **@return** true 包含

\* false 不包含

\*/

**public** **boolean** isContained(Object obj) {

**return** indexOf(obj) >= 0;

}

/\*\*

\* 通过size来提供索引，向ArrayList放东西

\* **@param** obj

\*/

**public** **void** add(Object obj) {

value[size]=obj;

size++;

**if**(size>=value.length) {

//装不下，要扩容

**int** newCapacity = value.length\*2;

Object[] newList = **new** Object[newCapacity];

**for**(**int** i=0;i<value.length;i++) {

newList[i]=value[i];

}

value = newList;

}

}

/\*\*

\* 取出数据

\* **@param** index

\* **@return**

\*/

**public** Object get(**int** index) {

rangeCheck(index);

**return** value[index];

}

/\*\*

\* 测试方法

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

MyArrayList list = **new** MyArrayList(2);

list.add("aaa");

list.add(**new** Human("Vera"));

list.add("bbbb");

Human h = (Human) list.get(1);

System.***out***.println(h.getName());//输出：Vera

System.***out***.println(list.get(2));//输出：bbbb

//数组越界丢异常

//System.out.println(list.get(-1));

System.***out***.println(list.size);//输出：3

}

}

## Plus：补充：关于ArrayCopy

**1. /\*\* src the source array. startPos starting position in the source array. dest**

**\* the destination array. destPos starting position in the destination data.**

**\* length the number of array elements to be copied.**

**\*/**

**2. 新数组 = des到destPos为止 + src从startPos开始（插入length个）+des剩下的。**

**3.** Byte[] src = { 1, 2, 3, 4, 5, 6, 7 };

System.*arraycopy*(src, 1, src, 2, 4);

**for** (**int** i = 0; i < src.length; i++) {

System.***out***.print("-> " + src[i]);// 输出-> 1-> 2-> 2-> 3-> 4-> 5-> 7

}

**4. 对于常用的插入方法中：插入后变为1 2 3 3 4 5 6 7，然后将第二个3替换为element。**

**public** **void** add(**int** index, E element) {

rangeCheckForAdd(index);

ensureCapacityInternal(size + 1); // Increments modCount!!

System.*arraycopy*(elementData, index, elementData, index + 1,

size - index);

elementData[index] = element;

size++;

}

## 60. 多维数组基本语法和内存分析

**1. 二维数组举例：**

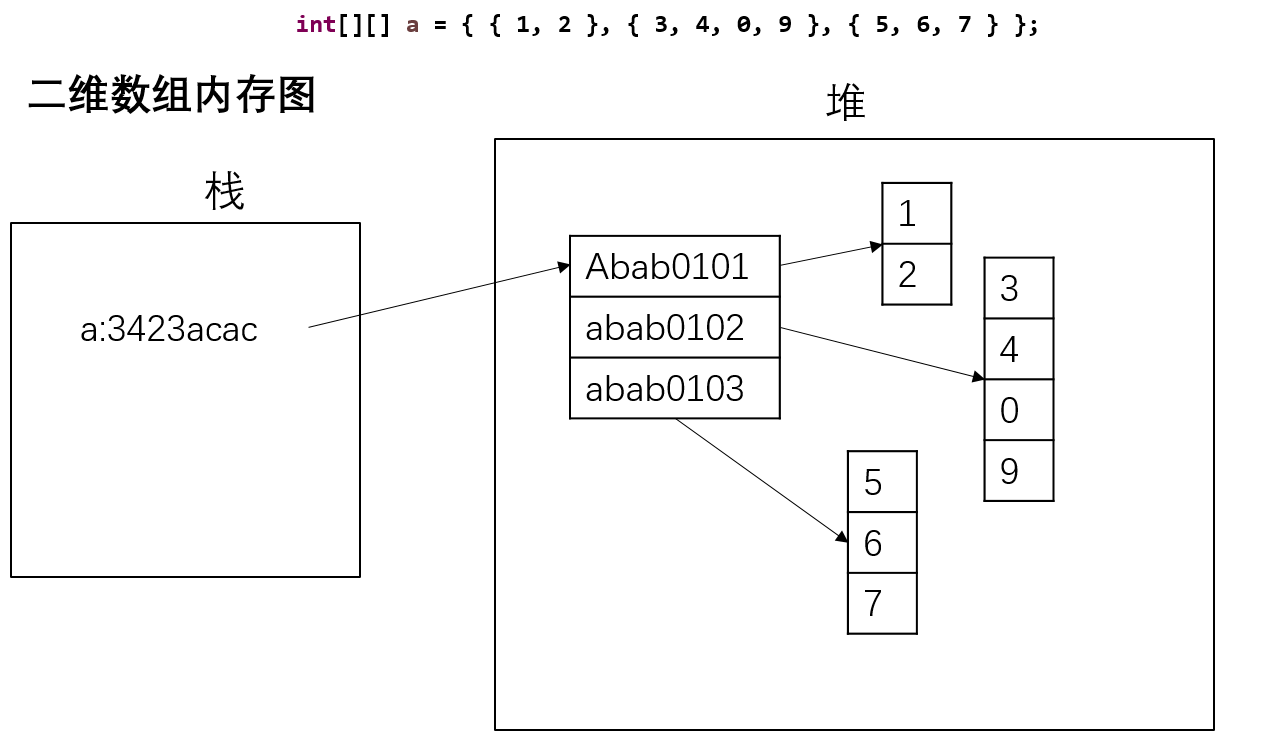
**int [][] a = {{1,2},{3,4,0,9},{5,6,7}}; （数组元素还是数组）**

**Java中多维数组不必是规则矩阵形式。**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **i j** | **j=0** | **j=1** | **j=2** | **j=3** |
| **i=0** | **1** | **2** |  |  |
| **i=1** | **3** | **4** | **0** | **9** |
| **i=2** | **5** | **6** | **7** |  |

**2. 二维数组内存分析图**

* **一级数组的默认初始化值为null**
* **二级数组的默认初始化值为int**

****

**3. Java中多维数组的声明和初始化应按从高维到低维的顺序进行，例如：**

|  |
| --- |
| int [][] a = new int [3][]; //定义一级数组长度为3  a[0] = new int[2]; //二级数组长度为2  a[1] = new int[4]; //二级数组长度为4  a[2] = new int[3]; //二级数组长度为3  int t1[][] = new int[][4]; //非法 |

## 61. 多维数组练习矩阵运算 （Matrix）

**1. 矩阵的加法：对应位置的元素相加。**

**1 3 与 3 4 相加 等于 4 7**

**2 4 5 6 7 10**

* **c.length()表示的是第一维的长度，若c[3][4]，则c.length()为3，i列的长度为c[i].length()；**

**--------------------------------矩阵加法---------------------------------**

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

**int**[][] b = {{3,4},{5,6}};

**int**[][] c = **new** **int**[2][2];

**for**(**int** i=0;i<2;i++) {

**for**(**int** j=0;j<2;j++) {

c[i][j]=a[i][j]+b[j][j];

System.***out***.print(c[i][j]+"\t");

}

System.***out***.println();

}

## 62. 数组的拷贝、排序、二分法、命令行参数以及增强for循环

### 1. 数组的拷贝

|  |
| --- |
| **System类里也包含了一个static void arraycopy( Object src, int srcpos, Object dest, int destPos, int length)方法，该方法可以将src数组里的元素赋值给dest数组的元素，其中srcpos指定从src数组的第几个元素开始赋值，length参数指定将src数组的多少个元素赋给dest数组的元素。**  **String[] s = {“Microsoft”,”IBM”,”SUM”,”Oracle”,”Apple”};**  **String[] sBak = new String[6];**  **System.arraycopy(s,0,sBak,0,s.length); //将s拷贝给sBak** |

* **System类的定义和相关内容**

|  |
| --- |
| **System类包含一些有用的类字段和方法。它不能被实例化。**  **在System类提供的设施中，有标准输入、标准输出和错误输出流；对外部定义的属性和环境变量的访问；加载文件和库的方法；还有快速复制数组的一部分的实用方法。** |

* **java.util.Arrays**

|  |
| --- |
| **此类包含用来操作数组（比如排序和搜索）的各种方法。此类还包含一个允许将数组作为列表来查看的静态工厂。**  **除非特别注明，否则如果指定数组引用为null，则此类中的方法都会抛出NullPointerException(空指针异常)** |

* + **打印数组**

|  |
| --- |
| **Int[] a = {1,2};**  **System.out.println(a); //[I@35ce36**  **System.out.println(Arrays.toString(a)); //[1, 2]** |

### 2. 数组排序

|  |
| --- |
| **int[] a = {1, 2, 323, 23, 543, 12, 68};**  **Arrays.sort(a);**  **System.out.println(Arrays.toString(a)); //**[1, 2, 12, 23, 68, 323, 543] |

* + **如果数组元素是引用类型，那么需要怎么做呢？**

|  |
| --- |
| **public** **class** Test02 {  **public** **static** **void** main(String[] args) {  Man[] msMans = { **new** Man(3, "a"), **new** Man(60, "b"), **new** Man(2, "c") };  Arrays.*sort*(msMans);  System.***out***.println(Arrays.*toString*(msMans));  // 输出：[c, a, b]  }  }  //Comparable是一个接口，必須自己重新实现。返回情況如下：  //负整数 小于 0 相等 正整数 大于  **class** Man **implements** Comparable {  **int** age;  **int** id;  String name;  **public** Man(**int** age, String name) {  **super**();  **this**.age = age;  **this**.name = name;  }  **public** String toString() {  **return** **this**.name;  }  @Override  **public** **int** compareTo(Object o) {  // **TODO** Auto-generated method stub  Man man = (Man) o;  **if** (**this**.age < man.age) {  **return** -1;  }  **if** (**this**.age > man.age) {  **return** 1;  }  **return** 0;  }  } |

### 3. 二分法查找

|  |
| --- |
| **int[] a = {1,2,323,23,543,12,68};**  **Arrays.sort(a); //使用二分法查找，必须现对数组进行排序**  **System.out.println(“该元素的索引：”+Arrays.binarySearch(a, 12));**  //输出： 该元素的索引：2 (已经经过排序，所以是2) |

### 4. Arrays替换填充

|  |
| --- |
| **Arrays.fill(a, 2, 4, 100); //将2到4索引的元素替换为100** |

### 5. 常见排序和二分法查找（重要！！面试前复习！！）

* + **冒牌排序**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[] value = {3,1,6,2,3,2,7,5,2};  *bubbleSort*(value);  System.***out***.println(Arrays.*toString*(value));  }  //冒泡排序原理：一共遍历value.length遍。第一次找出最大的值，放到最后；第二次找到第二大的值，放到倒数第二的位置，以此类推。时间复杂度为O(n^2)  **public** **static** **void** bubbleSort(**int**[] value) {  **int** temp;  **for**(**int** i=0;i<value.length;i++) {  **for**(**int** j=0;j<value.length-1-i;j++) {  **if**(value[j]>value[j+1]) {  temp = value[j];  value[j]=value[j+1];  value[j+1]=temp;  }  }  }  } |

* + **二分法查找**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[] arr = { 234, 546, 853, 56, 12, 76, 345, 7, 53, 1, 3, 260 };  **int** searchWord = 345; // 所要查找的数  **int** generalSearchCount = *generalSearch*(arr, searchWord); // 普通查找循环的次数  **int** binarySearchCount = *BinarySearch*(arr, searchWord); // 二分法查找循环的次数  System.***out***.println("普通循环查找" + searchWord + "的次数是" + generalSearchCount);  System.***out***.println("二分法查找" + searchWord + "的次数是" + binarySearchCount);  //输出： 普通循环查找345的次数是7  //输出： 二分法查找345的次数是3  }  **static** **int** generalSearch(**int**[] arr, **int** searchWord) {  // 普通的循环法，最少需要比较一次，比如查找234，最多需要比较11次，比如260  **int** searchCount = 0;  **for** (**int** i = 0; i < arr.length; i++) {  searchCount++;  **if** (arr[i] == searchWord) {  **break**;  }  }  **return** searchCount;  }  **static** **int** BinarySearch(**int**[] arr, **int** searchWord) {  // 二分法查找必须先排序,从小到大排序  Arrays.*sort*(arr);  // 二分法查找  **int** iIndex = 0; // 类似指针的东西  **int** iStart = 0;  **int** iEnd = arr.length - 1;  **int** searchCount = 0;  **for** (**int** i = 0; i < arr.length / 2; i++) {  searchCount++;  iIndex = (iStart + iEnd) / 2;  **if** (arr[iIndex] < searchWord) {  iStart = iIndex;  }  **if** (arr[iIndex] == searchWord) {  **break**;  }  **if** (arr[iIndex] > searchWord) {  iEnd = iIndex;  }  }  **return** searchCount;  } |

### 6. 命令行参数的问题 String[] args

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **for**(**int** i=0;i<args.length;i++) {  System.***out***.println("args["+i+"] = "+args[i]);  }  }  **Eclipse中如何使用？**  **Run** 🡪 **Run Configurations** 🡪 **Arguments 🡪 在图示部分输入arguments**    **运行结果：** |

**6. 增强for循环**

|  |
| --- |
| **int**[] a = {23,44,76,12,87,34,29,12};  //表示遍历a: m代表a数组里的一个元素  **for** (**int** m:a) {  System.***out***.print(m+"\t");  //输出：23 44 76 12 87 34 39 12  } |

## 63. Java常用类（JDK源码分析）

**1. Java常用类包括：**

* + **基本数据类型的包装类**
  + **字符串相关类**
    - **不可变字符序列：String**
    - **可变字符序列：StringBuffer, StringBuilder(线程不安全，效率高)**
  + **时间处理相关类**
    - **Date**
    - **DateFormat、SimpleDateFormat**
    - **Calendar**
  + **Math类**
  + **File类**
  + **枚举类**

**2. 基本数据类型的包装类 （Wrapper Class）**

* **为什么需要包装类？**

**-- JAVA并不是纯面向对象的语言（比如8种数据类型就不是对象，增加运行效率）。但是我们在实际使用中经常需要将基本数据转化成对象，便于操作。比如：集合的操作中。这时，我们就需要将基本类型数据转化成对象。（比如ArrayList定义的就是放入一个对象，如果想要放入一个数，就会自动变成包装类再放入）**

* **包装类均位于java.lang包，包装类和基本数据类型的对应关系：**

|  |  |
| --- | --- |
| **基本数据类型** | **包装类** |
| **byte** | **Byte** |
| **boolean** | **Boolean** |
| **short** | **Short** |
| **char** | **Character** |
| **int** | **Integer** |
| **long** | **Long** |
| **float** | **Float** |
| **double** | **Double** |

* **Byte、Double、Float、Integer、Long、Short都继承了抽象Number类**
* **所谓包装类：就是用一个类把值给包起来了，值作为类中的一个属性出现。**
* **包装类的作用**
  + **提供：字符串、基本数据类型、对象之间的互相转化的方式**
  + **包含每种基本数据类型的相关属性如最大值、最小值等**
* **所有包装类（Wrapper Class）都有类似的方法，掌握一个其他的都类似，下面的例子以Integer为例：**

**---------------------------------Integer包装类的使用举例-------------------------------**

**public** **static** **void** main(String[] args) {

Integer i = **new** Integer(1000); //把1000包装成一个对象

System.***out***.println(Integer.***MAX\_VALUE***); //输出：2147483647

System.***out***.println(Integer.*toHexString*(i)); //把i转成16进制的，输出：3e8

//parseInt:把一个字符串按照十进制转化成一个int

Integer i2 = Integer.*parseInt*("234");

Integer i3 = **new** Integer("333"); //也是可以的

System.***out***.println(i2+10); //输出：244

System.***out***.println(i3+10); //输出：343

//intValue：将Integer的内容转化成一个真正的数字（int）

System.***out***.println(i2.intValue()); //输出：234

//把数字转成字符串

String str = 234+"";

}

## 64. 自动装箱和拆箱（Auto-boxing & Unboxing）

### 1. 自动装箱：

* **基本数据类型自动地封装到与它相同类型的包装中。**

|  |
| --- |
| //jdk5.0之后实现自动装箱（在编译器中），不用写new;右边是数字，左边是引用类型。  //编译器改进：Integer a = new Integer(1000);  Integer a = 1000;  Integer b = 1000;  System.***out***.println(a==b);//false:两个对象  System.***out***.println(a.equals(b)); //true,equals比较值的大小 |

### 2. 自动拆箱：

* **包装类对象自动地转换成基本数据类型：**

|  |
| --- |
| //自动拆箱，左边是基本数据类型，右边是引用类型  //编译器改进：new Integer(1500).intValue();  **int** c = **new** Integer(1500);  **int** d = b; //b.intValue(); |

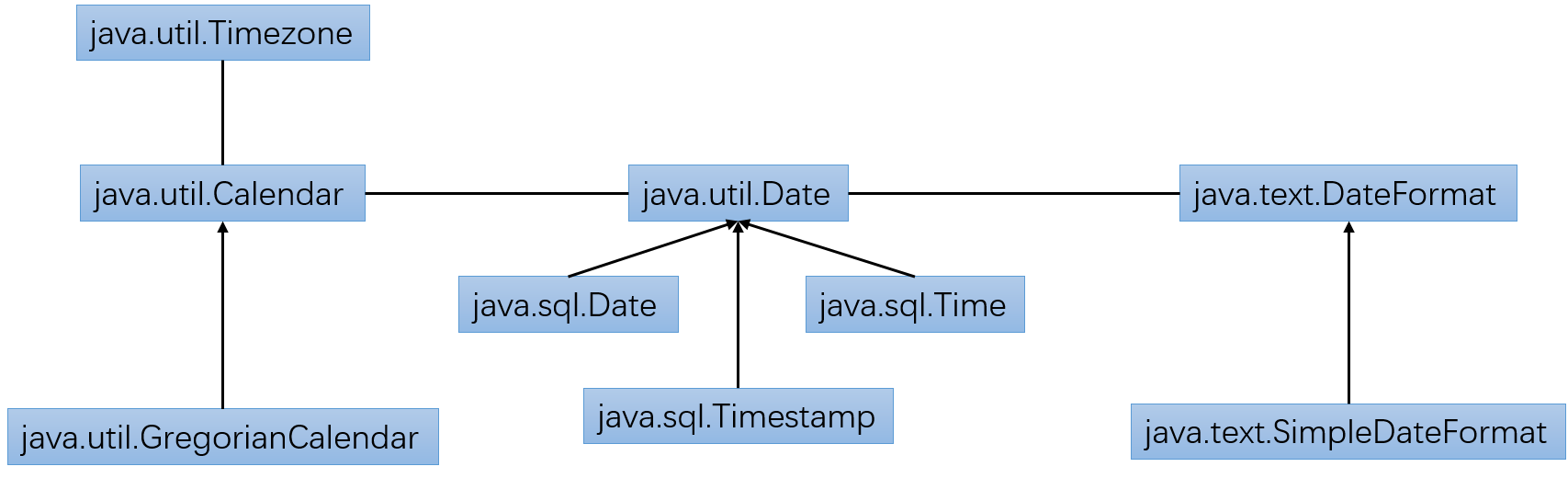
### 3. 缓存问题

* **[-128,127]之间的数，仍然当作基本数据类型来处理,为了提高效率,有缓存。**

|  |
| --- |
| //Cache to support the object identity semantics of autoboxing for values between  //\* -128 and 127 (inclusive) as required by JLS.  Integer d1 = 123;  Integer d2 = 123;  System.***out***.println(d1==d2); //true  System.***out***.println(d1.equals(d2));//true |

## 65. Date类的使用（JDK源码分析）

### 1. 时间处理相关类：

****

* **java.util.Date：核心**
* **java.sql.Date/Time/Timestamp：数据库相关，继承核心**
* **java.text.DateFormat：时间格式化**

**java.text.SimpleDateFormat：时间和字符串转换**

* **java.util.Calendar：日期计算，比如100天之后是哪天**
* **java.util.GregorianCalendar：公历**

### 2. Date时间类(java.util.Date)：

* **在标准Java类库中包含一个Date类。它的对象表示一个特定的瞬间。**
* **Java中如何表现时间：选择1970.1.1 0点作为标准纪元。从该时刻开始到某个时刻的毫秒数，类型是long。在零点之前，就用负数表示。**

|  |
| --- |
| //The class Date represents a specific instant in time, with millisecond precision  //public Date() {  //this(System.currentTimeMillis());  //}  Date d = **new** Date();  **long** t = System.*currentTimeMillis*();  System.***out***.println(t); //1522742521607,标准纪元到现在的毫秒数    //距离标准纪元1000毫秒  Date d2 = **new** Date(1000);  System.***out***.println(d2); //Thu Jan 01 09:00:01 JST 1970  System.***out***.println(d2.~~toGMTString~~());//1 Jan 1970 00:00:01 GMT 不建议使用    //getTime:获得毫秒数  System.***out***.println(d2.getTime()); //1000    //setTime:设置毫秒数  d2.setTime(23432345);  System.***out***.println(d2.~~toGMTString~~());//1 Jan 1970 06:30:32 GMT    /\*\*  \* compare to:时间比较  \* 返回值：+:d3>d4； 0:d3=d4 -:d3<d4  \*/  Date d3 = **new** Date(434234);  Date d4 = **new** Date(1000);  System.***out***.println(d3.compareTo(d4)); |
|  |

## 66. DateFormat和SimpleDateFormat (时间和字符串的互相转换)

### 1. 简介

**完成字符串和时间对象的转化**

* **format：把时间转化成字符串**
* **parse：把字符串转化成时间**

|  |
| --- |
| //DateFormat是一个抽象类，不能自己建，只能建它的子类  //建立一个自定义的时间格式，自己随便定义，也叫格式化字符串  DateFormat df = **new** SimpleDateFormat("yyyy年MM月dd日 hh时mm秒");  //使用format将Date转化为字符串  Date d = **new** Date(123213123L);  String str = df.format(d);  System.***out***.println(str); //1970年01月02日 07时13秒    //使用parse将字符串转化成Date  //需要在main方法中throws parseException  String str2 = "1970年01月02日 07时13秒";  Date d2;  **try** {  d2 = df.parse(str2);  System.***out***.println(d2);//Fri Jan 02 07:13:00 JST 1970  System.***out***.println(d2.getTime());  } **catch** (ParseException e) {  e.printStackTrace();  } |

### 2. 日期格式化对应字母

|  |  |  |  |
| --- | --- | --- | --- |
| **Letter** | **Date or Time Component** | **Presentation** | **Examples** |
| **G** | **Era designator** | **Text** | **AD** |
| **y** | **Year** | **Year** | **1996; 96** |
| **Y** | **Week year** | **Year** | **2009; 09** |
| **M** | **Month in year (context sensitive)** | **Month** | **July; Jul; 07** |
| **L** | **Month in year (standalone form)** | **Month** | **July; Jul; 07** |
| **w** | **Week in year** | **Number** | **27** |
| **W** | **Week in month** | **Number** | **2** |
| **D** | **Day in year** | **Number** | **189** |
| **d** | **Day in month** | **Number** | **10** |
| **F** | **Day of week in month** | **Number** | **2** |
| **E** | **Day name in week** | **Text** | **Tuesday; Tue** |
| **u** | **Day number of week (1 = Monday, ..., 7 = Sunday)** | **Number** | **1** |
| **a** | **Am/pm marker** | **Text** | **PM** |
| **H** | **Hour in day (0-23)** | **Number** | **0** |
| **k** | **Hour in day (1-24)** | **Number** | **24** |
| **K** | **Hour in am/pm (0-11)** | **Number** | **0** |
| **h** | **Hour in am/pm (1-12)** | **Number** | **12** |
| **m** | **Minute in hour** | **Number** | **30** |
| **s** | **Second in minute** | **Number** | **55** |
| **S** | **Millisecond** | **Number** | **978** |
| **z** | **Time zone** | **General time zone** | **Pacific Standard Time; PST; GMT-08:00** |
| **Z** | **Time zone** | **RFC 822 time zone** | **-0800** |
| **X** | **Time zone** | **ISO 8601 time zone** | **-08; -0800; -08:00** |

## 67. Calendar和GregorianCalendar的使用（日期计算）

**1. 介绍**

**人们对于时间的认识是：*某年某月某日*这样的日期概念**