

# Java概述

---

面向对象

两个基本概念：类、对象

三大特性：封装、继承、多态

**JDK**(Java Development Kit Java开发工具包) JDK是提供给Java开发人员使用的，其中包含了java的开发工具，也包括了JRE。所以安装了JDK，就不用单独安装JRE了。其中的开发工具：编译工具(javac.exe) 打包工具(jar.exe)等

**JRE**(Java Runtime Environment Java运行环境) 包括Java虚拟机(JVM Java Virtual Machine)和Java程序所需的核心类库等，如果想要运行一个开发好的Java程序，计算机中只需要安装JRE即可。

**JVM**

# Java基本语法

---

基本数据类型

byte1 short2 int4 long8

float4 double8

char2 boolean

引用数据类型

类 接口 数组

# 数组

---

数组对象会在内存中开辟一整块连续的空间，而数组名中引用的是这块连续空间的首地址。

```
int[] arr = new int[3];
```

```
int[] arr = new int[]{1,2,3};
```

```
int [] [] arr = new int[] [];
```

```
int[][] arr = new int[3] [];
```

```
arr[0] = new int[3];
```

```
int [] [] arr = new int[] {{1,2},{1,2,3},{1,2,3,4}};
```

java.util.Arrays类

equals() fill() sort() binarySearch()

# 面向对象

---

万事万物皆对象

**堆 (Heap)**，此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。这一点在Java虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配。

**栈 (Stack)**，是指虚拟机栈。虚拟机栈用于存储局部变量等。局部变量表存放了编译期可知长度的各种基本数据类型 (boolean、byte、char、short、int、float、long、double)、对象引用 (reference类型，它不等同于对象本身，是对象在堆内存的首地址)。方法执行完，自动释放。

**方法区 (Method Area)**，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。常量池，静态域

private default protected public

子类中所有的构造器默认都会访问父类中空参数的构造器

当父类中没有空参数的构造器时，子类的构造器首行必须通过this(参数列表)或者super(参数列表)语句指定调用本类或者父类中相应的构造器。

方法具有多态特性，属性不具有

## 包装类

Integer Character

String.valueOf() Integer.parseInt()

## 单例设计模式：

一个类只能存在一个对象实例



```
class Singleton {  
    // 1.私有化构造器  
    private Singleton() {  
    }  
  
    // 2.内部提供一个当前类的实例  
    // 4.此实例也必须静态化  
    private static Singleton single = new Singleton();  
  
    // 3.提供公共的静态的方法，返回当前类的对象  
    public static Singleton getInstance() {  
        return single;  
    }  
}
```

让天下没有难学的技术



```
class Singleton {  
    // 1.私有化构造器  
    private Singleton() {  
    }  
  
    // 2.内部提供一个当前类的实例  
    // 4.此实例也必须静态化  
    private static Singleton single;  
    // 3.提供公共的静态的方法，返回当前类的对象  
    public static Singleton getInstance() {  
        if(single == null) {  
            single = new Singleton();  
        }  
        return single;  
    }  
}
```

懒汉式暂时还存在线程安全问题，讲到多线程时，可修复

让天下没有难学的技术

静态代码块 非静态代码块

**final:**

类不能被继承

方法不能被重写

标记的变量即为常量，不可修改

抽象类 接口

## 内部类

静态内部类

非静态内部类

局部内部类

只能在声明它的方法或代码块中使用

匿名内部类

# 异常

---

Error: Java虚拟机无法解决的严重问题，比如：StackOverflowError

Exception: 其它因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理。

## 运行时异常

是指编译器不要求强制处置的异常。一般是指编程时的逻辑错误，是程序员应该积极避免其出现的异常。  
java.lang.RuntimeException类及它的子类都是运行时异常。

## 编译时异常

是指编译器要求必须处置的异常。即程序在运行时由于外界因素造成的一般性异常。编译器要求Java程序必须捕获或声明所有编译时异常。

## 7.2 常见异常

---

### ● **java.lang.RuntimeException**

- ClassCastException
- ArrayIndexOutOfBoundsException
- NullPointerException
- ArithmeticException
- NumberFormatException
- InputMismatchException
- . . .

### ● **java.io.IOException**

- FileNotFoundException
- EOFException

### ● **java.lang.ClassNotFoundException**

### ● **java.lang.InterruptedIOException**

### ● **java.io.FileNotFoundException**

### ● **java.sql.SQLException**

Java程序的执行过程中如出现异常，会生成一个异常类对象

由虚拟机自动生成：程序运行过程中，虚拟机检测到程序发生了问题，如果在当前代码中没有找到相应的处理器，就会在后台自动创建一个对应异常类的实例对象并抛出——自动抛出

由开发人员手动创建：Exception exception = new ClassCastException();——创建好的异常对象不抛出对程序没有任何影响，和创建一个普通对象一样

# 异常的抛出机制



为保证程序正常执行，代码必须对可能出现的异常进行处理。

## 7.6 用户自定义异常类



- 一般地，用户自定义异常类都是**RuntimeException**的子类。
- 自定义异常类通常需要编写几个**重载的构造器**。
- 自定义异常需要提供**serialVersionUID**
- 自定义的异常通过**throw**抛出。
- 自定义异常最重要的是异常类的名字，当异常出现时，可以根据名字判断异常类型。

# 多线程

**进程(process)**是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域

**线程(thread)**, 进程可进一步细化为线程, 是一个程序内部的一条执行路径。线程作为调度和执行的单位, 每个线程拥有独立的运行栈和程序计数器(pc)

Java语言的JVM允许程序运行多个线程, 它通过java.lang.Thread 类来体现。

Thread类的特性

每个Thread对象相当于一个线程

每个线程都是通过某个特定Thread对象的run()方法来完成操作的, 经常把run()方法的主体称为线程体

通过该Thread对象的start()方法来启动这个线程, 而非直接调用run(), run()方法由JVM调用, 什么时候调用, 执行的过程控制都有操作系统的CPU 调度决定。想要启动多线程, 必须调用start方法。

# Thread类

## ● 构造器

- **Thread():** 创建新的Thread对象
- **Thread(String threadname):** 创建线程并指定线程实例名
- **Thread(Runnable target):** 指定创建线程的目标对象, 它实现了Runnable接口中的run方法
- **Thread(Runnable target, String name):** 创建新的Thread对象

创建线程:

## ● 方式一: 继承Thread类

- 1) 定义子类继承Thread类。
- 2) 子类中重写Thread类中的run方法。
- 3) 创建Thread子类对象, 即创建了线程对象。
- 4) 调用线程对象start方法: 启动线程, 调用run方法。

## ● 方式二: 实现Runnable接口

- 1) 定义子类, 实现Runnable接口。
- 2) 子类中重写Runnable接口中的run方法。
- 3) 通过Thread类含参构造器创建线程对象。
- 4) 将Runnable接口的子类对象作为实际参数传递给Thread类的构造器中。
- 5) 调用Thread类的start方法: 开启线程, 调用Runnable子类接口的run方法。

## 新增方式一：实现**Callable**接口

- 与使用**Runnable**相比， **Callable**功能更强大些
  - 相比**run()**方法， 可以有返回值
  - 方法可以抛出异常
  - 支持泛型的返回值
  - 需要借助**FutureTask**类， 比如获取返回结果

让天下没有难学

## 8.6 JDK5.0 新增线程创建方式



### 新增方式一：实现**Callable**接口

#### ● **Future**接口

- 可以对具体**Runnable**、**Callable**任务的执行结果进行取消、查询是否完成、获取结果等。
- **FutureTask**是**Future**接口的唯一的实现类
- **FutureTask** 同时实现了**Runnable**, **Future**接口。它既可以作为**Runnable**被线程执行，又可以作为**Future**得到**Callable**的返回值



## 新增方式二：使用线程池

- **背景：**经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
- **思路：**提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。
- **好处：**
  - 提高响应速度（减少了创建新线程的时间）
  - 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
  - 便于线程管理
    - ✓ corePoolSize: 核心池的大小
    - ✓ maximumPoolSize: 最大线程数
    - ✓ keepAliveTime: 线程没有任务时最多保持多长时间后会终止
    - ✓ ...

让天下没有难学的技术



## 线程池相关API

- JDK 5.0起提供了线程池相关API：[ExecutorService](#) 和 [Executors](#)
- **ExecutorService:** 真正的线程池接口。常见子类ThreadPoolExecutor
  - void execute(Runnable command)：执行任务/命令，没有返回值，一般用来执行 Runnable
  - <T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行 Callable
  - void shutdown()：关闭连接池
- **Executors:** 工具类、线程池的工厂类，用于创建并返回不同类型的线程池
  - Executors.newCachedThreadPool()：创建一个可根据需要创建新线程的线程池
  - Executors.newFixedThreadPool(n)：创建一个可重用固定线程数的线程池
  - Executors.newSingleThreadExecutor()：创建一个只有一个线程的线程池
  - Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

让天下没有难学的技术

## Thread类的有关方法(1)

- **void start():** 启动线程，并执行对象的run()方法
- **run():** 线程在被调度时执行的操作
- **String getName():** 返回线程的名称
- **void setName(String name):** 设置该线程名称
- **static Thread currentThread():** 返回当前线程。在Thread子类中就是this，通常用于主线程和Runnable实现类

让天下没有难学的

## 8.2 线程的创建和使用



### Thread类的有关方法(2)

- **static void yield():** 线程让步
  - 暂停当前正在执行的线程，把执行机会让给优先级相同或更高的线程
  - 若队列中没有同优先级的线程，忽略此方法
- **join() :** 当某个程序执行流中调用其他线程的 join() 方法时，调用线程将被阻塞，直到 join() 方法加入的 join 线程执行完为止
  - 低优先级的线程也可以获得执行
- **static void sleep(long millis):** (指定时间:毫秒)
  - 令当前活动线程在指定时间段内放弃对CPU控制，使其他线程有机会被执行，时间到后重排队。
  - 抛出InterruptedException异常
- **stop():** 强制线程生命期结束，不推荐使用
- **boolean isAlive():** 返回boolean，判断线程是否还活着

# 线程的调度

## ● 调度策略

➤ 时间片



➤ 抢占式：高优先级的线程抢占CPU

## ● Java 的调度方法

- 同优先级线程组成先进先出队列（先到先服务），使用时间片策略
- 对高优先级，使用优先调度的抢占式策略

让天下没有难学的

## 8.2 线程的创建和使用



### 线程的优先级

#### ● 线程的优先级等级

- MAX\_PRIORITY: 10
- MIN\_PRIORITY: 1
- NORM\_PRIORITY: 5

#### ● 涉及的方法

- **getPriority()** : 返回线程优先值
- **setPriority(int newPriority)** : 改变线程的优先级

#### ● 说明

- 线程创建时继承父线程的优先级
- 低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用

## 补充：线程的分类

Java中的线程分为两类：一种是**守护线程**，一种是**用户线程**。

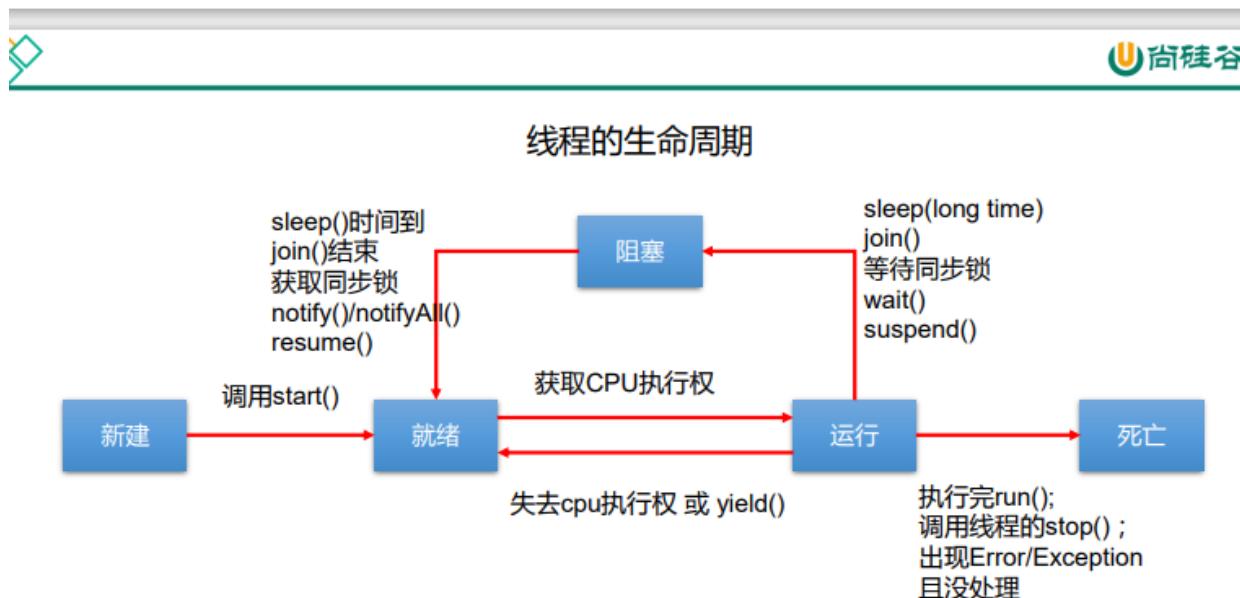
- 它们在几乎各个方面都是相同的，唯一的区别是判断JVM何时离开。
- 守护线程是用来服务用户线程的，通过在start()方法前调用**thread.setDaemon(true)**可以把一个用户线程变成一个守护线程。
- Java垃圾回收就是一个典型的守护线程。
- 若JVM中都是守护线程，当前JVM将退出。

### ● JDK中用**Thread.State**类定义了线程的几种状态

要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用**Thread**类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的五种状态：

- **新建**：当一个**Thread**类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
- **就绪**：处于新建状态的线程被**start()**后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源
- **运行**：当就绪的线程被调度并获得CPU资源时，便进入运行状态，**run()**方法定义了线程的操作和功能
- **阻塞**：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出CPU并临时中止自己的执行，进入阻塞状态
- **死亡**：线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束

让天下没有难学的技术



线程的同步：

对于并发工作，你需要某种方式来防止两个任务访问相同的资源（其实就是共享资源竞争）。防止这种冲突的方法就是当资源被一个任务使用时，在其上加锁。第一个访问某项资源的任务必须锁定这项资源，使其他任务在其被解锁之前，就无法访问它了，而在其被解锁之时，另一个任务就可以锁定并使用它了。

**同步代码块** synchronized(对象) { 需要被同步的代码; }

**同步方法** public synchronized void show (String name){ }

任何对象都可作为同步锁

同步方法的锁：静态方法类名.class 非静态方法this

同步代码块的锁：自己指定，一般也为类名.class或this

必须确保使用同一个资源的多个线程共用一把锁

### 释放锁的操作：

当前线程的同步方法、同步代码块执行结束。

当前线程在同步代码块、同步方法中遇到break、return终止了该代码块、该方法的继续执行。

当前线程在同步代码块、同步方法中出现了未处理的Error或Exception，导致异常结束。

当前线程在同步代码块、同步方法中执行了线程对象的wait()方法，当前线程暂停，并释放锁。

### Lock锁

显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。

java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。

ReentrantLock 类实现了 Lock，它拥有与 synchronized 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁。

```
class A{
    private final ReentrantLock lock = new ReentrantLock();
    public void m(){
        lock.lock();
        try{
            //保证线程安全的代码;
        }
        finally{
            lock.unlock();
        }
    }
}
```

注意：如果同步代码有异常，要将unlock()写入finally语句块

使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性

Lock > 同步代码块 > 同步方法

线程的通信

### ● **wait()** 与 **notify()** 和 **notifyAll()**

➤ **wait()**: 令当前线程挂起并放弃CPU、同步资源并等待，使别的线程可访问并修改共享资源，而当前线程排队等候其他线程调用**notify()**或**notifyAll()**方法唤醒，唤醒后等待重新获得对监视器的所有权后才能继续执行。

➤ **notify()**: 唤醒正在排队等待同步资源的线程中优先级最高者结束等待

➤ **notifyAll ()**: 唤醒正在排队等待资源的所有线程结束等待.

- 这三个方法只有在**synchronized**方法或**synchronized**代码块中才能使用，否则会报 **java.lang.IllegalMonitorStateException** 异常。
- 因为这三个方法必须有锁对象调用，而任意对象都可以作为**synchronized**的同步锁，因此这三个方法只能在**Object**类中声明。

让天下没有难学的技术

## › 8.5 线程的通信



### **wait() 方法**

- 在当前线程中调用方法： 对象名.**wait()**
- 使当前线程进入等待（某对象）状态，直到另一线程对该对象发出 **notify**（或**notifyAll**）为止。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
- 调用此方法后，**当前线程将释放对象监控权**，然后进入等待
- 在当前线程被**notify**后，要重新获得监控权，然后从断点处继续代码的执行。

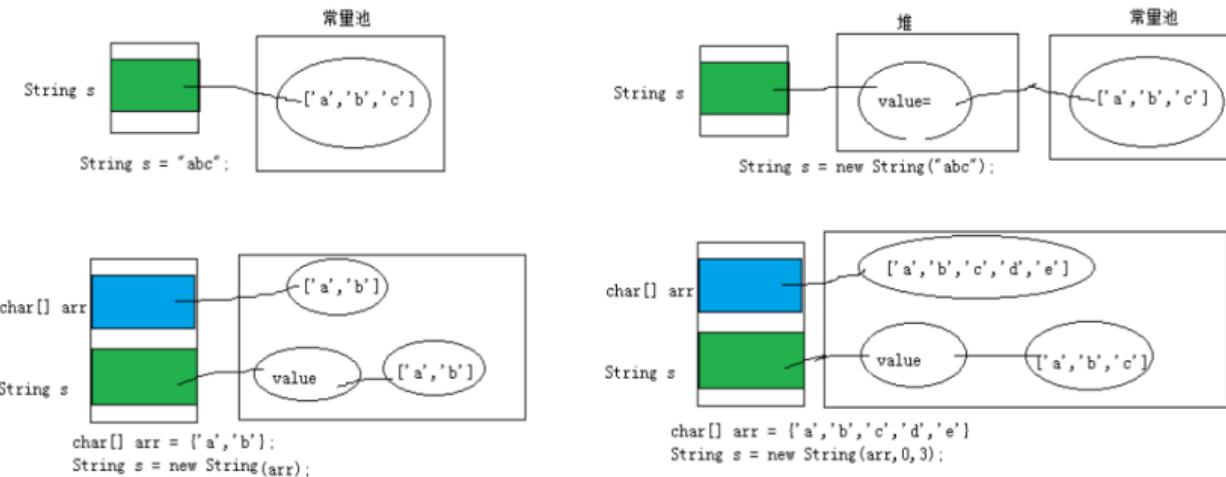
### **notify()/notifyAll()**

- 在当前线程中调用方法： 对象名.**notify()**
- 功能：唤醒等待该对象监控权的一个/所有线程。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

## Java常用类

### String

不可变的字符序列（常量），final修饰的byte数组



常量与常量的拼接结果在常量池。且常量池中不会存在相同内容的常量。

只要其中有一个是变量，结果就在堆中

如果拼接的结果调用intern()方法，返回值就在常量池中

- **int length():** 返回字符串的长度: `return value.length`
- **char charAt(int index):** 返回某索引处的字符 `return value[index]`
- **boolean isEmpty():** 判断是否是空字符串: `return value.length == 0`
- **String toLowerCase():** 使用默认语言环境，将 String 中的所有字符转换为小写
- **String toUpperCase():** 使用默认语言环境，将 String 中的所有字符转换为大写
- **String trim():** 返回字符串的副本，忽略前导空白和尾部空白
- **boolean equals(Object obj):** 比较字符串的内容是否相同
- **boolean equalsIgnoreCase(String anotherString):** 与equals方法类似，忽略大小写
- **String concat(String str):** 将指定字符串连接到此字符串的结尾。等价于用“+”
- **int compareTo(String anotherString):** 比较两个字符串的大小
- **String substring(int beginIndex):** 返回一个新的字符串，它是此字符串的从beginIndex开始截取到最后的一个子字符串。
- **String substring(int beginIndex, int endIndex) :** 返回一个新字符串，它是此字符串从beginIndex开始截取到endIndex(不包含)的一个子字符串。



## 9.1 字符串相关的类：String常用方法2



- **boolean endsWith(String suffix):** 测试此字符串是否以指定的后缀结束
- **boolean startsWith(String prefix):** 测试此字符串是否以指定的前缀开始
- **boolean startsWith(String prefix, int toffset):** 测试此字符串从指定索引开始的子字符串是否以指定前缀开始

- **boolean contains(CharSequence s):** 当且仅当此字符串包含指定的 char 值序列时, 返回 true
  - **int indexOf(String str):** 返回指定子字符串在此字符串中第一次出现处的索引
  - **int indexOf(String str, int fromIndex):** 返回指定子字符串在此字符串中第一次出现处的索引, 从指定的索引开始
  - **int lastIndexOf(String str):** 返回指定子字符串在此字符串中最右边出现处的索引
  - **int lastIndexOf(String str, int fromIndex):** 返回指定子字符串在此字符串中最后一次出现处的索引, 从指定的索引开始反向搜索
- 注: indexOf和lastIndexOf方法如果未找到都是返回-1

让天下没有难学的技术



## 9.1 字符串相关的类: String常用方法3



- **String replace(char oldChar, char newChar):** 返回一个新的字符串, 它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。
  - **String replace(CharSequence target, CharSequence replacement):** 使用指定的字面值替换序列替换此字符串所有匹配字面值目标序列的子字符串。
  - **String replaceAll(String regex, String replacement):** 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。
  - **String replaceFirst(String regex, String replacement):** 使用给定的 replacement 替换此字符串匹配给定的正则表达式的一个子字符串。
- 
- **boolean matches(String regex):** 告知此字符串是否匹配给定的正则表达式。
- 
- **String[] split(String regex):** 根据给定正则表达式的匹配拆分此字符串。
  - **String[] split(String regex, int limit):** 根据匹配给定的正则表达式来拆分此字符串, 最多不超过limit个, 如果超过了, 剩下的全部都放到最后一个元素中。

## StringBuffer

可变的字符序列, 可以对字符串内容进行增删, 此时不会产生新的对象。

非final修饰的数组

StringBuffer reverse() : 把当前字符序列逆转

效率低、线程安全

## StringBuilder

StringBuilder 和 StringBuffer 非常类似, 均代表可变的字符序列, 而且提供相关功能的方法也一样

效率高、线程不安全

## 日期时间API

## System

System.currentTimeMillis()

## Date

### 2. java.util.Date类

表示特定的瞬间，精确到毫秒

- 构造器：

- **Date():** 使用无参构造器创建的对象可以获取本地当前时间。
- **Date(long date)**

- 常用方法

- **getTime():** 返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。
- **toString():** 把此 Date 对象转换为以下形式的 String: dow mon dd hh:mm:ss zzz yyyy 其中：dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)，zzz是时间标准。
- 其它很多方法都过时了。

### 3. java.text.SimpleDateFormat类

- Date类的API不易于国际化，大部分被废弃了，**java.text.SimpleDateFormat**类是一个不与语言环境有关的方式来格式化和解析日期的具体类。
- 它允许进行**格式化：日期→文本、解析：文本→日期**
- **格式化：**
  - **SimpleDateFormat()**：默认的模式和语言环境创建对象
  - **public SimpleDateFormat(String pattern)**：该构造方法可以用参数pattern指定的格式创建一个对象，该对象调用：
  - **public String format(Date date)**：方法格式化时间对象date
- **解析：**
  - **public Date parse(String source)**：从给定字符串的开始解析文本，以生成一个日期。

让天下没有难学的技

## 9.2 JDK8之前日期时间API



字母	日期或时间元素	表示	示例
G	Era 标志符	<a href="#">Text</a>	AD
y	年	<a href="#">Year</a>	1996; 96
M	年中的月份	<a href="#">Month</a>	July; Jul; 07
w	年中的周数	<a href="#">Number</a>	27
W	月份中的周数	<a href="#">Number</a>	2
D	年中的天数	<a href="#">Number</a>	189
d	月份中的天数	<a href="#">Number</a>	10
F	月份中的星期	<a href="#">Number</a>	2
E	星期中的天数	<a href="#">Text</a>	Tuesday; Tue
a	Am/pm 标记	<a href="#">Text</a>	PM
H	一天中的小时数 (0-23)	<a href="#">Number</a>	0
k	一天中的小时数 (1-24)	<a href="#">Number</a>	24
K	am/pm 中的小时数 (0-11)	<a href="#">Number</a>	0
h	am/pm 中的小时数 (1-12)	<a href="#">Number</a>	12
n	小时中的分钟数	<a href="#">Number</a>	30
s	分钟中的秒数	<a href="#">Number</a>	55
S	毫秒数	<a href="#">Number</a>	978
z	时区	<a href="#">General time zone</a>	Pacific Standard Time; PST; GMT-08:00
Z	时区	<a href="#">RFC 822 time zone</a>	-0800

让天下没有难学的技

## Calendar

## 4. java.util.Calendar(日历)类

- Calendar是一个抽象基类，主用用于完成日期字段之间相互操作的功能。
- 获取Calendar实例的方法
  - 使用`Calendar.getInstance()`方法
  - 调用它的子类`GregorianCalendar`的构造器。
- 一个Calendar的实例是系统时间的抽象表示，通过`get(int field)`方法来取得想要的时间信息。比如YEAR、MONTH、DAY\_OF\_WEEK、HOUR\_OF\_DAY、MINUTE、SECOND
  - `public void set(int field,int value)`
  - `public void add(int field,int amount)`
  - `public final Date getTime()`
  - `public final void setTime(Date date)`
- 注意：
  - 获取月份时：一月是0，二月是1，以此类推，12月是11
  - 获取星期时：周日是1，周二2，。。。周六是7



**LocalDate、LocalTime、LocalDateTime**

- **LocalDate**、**LocalTime**、**LocalDateTime** 类是其中较重要的几个类，它们的实例是不可变的对象，分别表示使用 ISO-8601 日历系统的日期、时间、日期和时间。它们提供了简单的本地日期或时间，并不包含当前的时间信息，也不包含与时区相关的信息。

- **LocalDate** 代表 ISO 格式 (yyyy-MM-dd) 的日期，可以存储生日、纪念日等日期。
- **LocalTime** 表示一个时间，而不是日期。
- **LocalDateTime** 是用来表示日期和时间的，这是一个最常用的类之一。

注：ISO-8601 日历系统是国际标准化组织制定的现代公民的日期和时间的表示法，也就是公历。

让天下没有难学的技术

	方法	描述
	<b>now() / * now(ZonedDateTime zone)</b>	静态方法，根据当前时间创建对象/指定时区的对象
	<b>of()</b>	静态方法，根据指定日期/时间创建对象
	<b>getDayOfMonth() / getDayOfYear()</b>	获得月份天数(1-31) / 获得年份天数(1-366)
	<b>getDayOfWeek()</b>	获得星期几(返回一个 DayOfWeek 枚举值)
	<b>getMonth()</b>	获得月份，返回一个 Month 枚举值
	<b>getMonthValue() / getYear()</b>	获得月份(1-12) / 获得年份
	<b>getHour() / getMinute() / getSecond()</b>	获得当前对象对应的小时、分钟、秒
	<b>withDayOfMonth() / withDayOfYear() / withMonth() / withYear()</b>	将月份天数、年份天数、月份、年份修改为指定的值并返回新的对象
	<b>plusDays(), plusWeeks(), plusMonths(), plusYears(), plusHours()</b>	向当前对象添加几天、几周、几个月、几年、几小时
	<b>minusMonths() / minusWeeks() / minusDays() / minusYears() / minusHours()</b>	从当前对象减去几个月、几周、几天、几年、几小时

## Instant

- **Instant:** 时间线上的一个瞬时点。这可能被用来记录应用程序中的事件时间戳。
- 在处理时间和日期的时候，我们通常会想到年,月,日,时,分,秒。然而，这只是时间的一个模型，是面向人类的。第二种通用模型是面向机器的，或者说是连续的。在此模型中，时间线中的一个点表示为一个很大的数，这有利于计算机处理。[在UNIX中，这个数从1970年开始，以秒为单位；同样的，在Java中，也是从1970年开始，但以毫秒为单位。](#)
- [java.time包通过值类型Instant提供机器视图，不提供处理人类意义上的时间单位。Instant表示时间线上的一点，而不需要任何上下文信息，例如，时区。概念上讲，它只是简单的表示自1970年1月1日0时0分0秒（UTC）开始的秒数。因为java.time包是基于纳秒计算的，所以Instant的精度可以达到纳秒级。](#)
- $(1 \text{ ns} = 10^{-9} \text{ s})$  1秒 = 1000毫秒 =  $10^6$ 微秒= $10^9$ 纳秒

让天下没有难学的IT

## 9.3 JDK8中新日期时间API



方法	描述
<code>now()</code>	静态方法，返回默认UTC时区的Instant类的对象
<code>ofEpochMilli(long epochMilli)</code>	静态方法，返回在1970-01-01 00:00:00基础上加上指定毫秒数之后的Instant类的对象
<code>atOffset(ZoneOffset offset)</code>	结合即时的偏移来创建一个 OffsetDateTime
<code>toEpochMilli()</code>	返回1970-01-01 00:00:00到当前时间的毫秒数，即为时间戳

时间戳是指格林威治时间1970年01月01日00时00分00秒(北京时间1970年01月01日08时00分00秒)起至现在的总秒数。

## DateTimeFormatter

### 9.3.3 格式化与解析日期或时间

`java.time.format.DateTimeFormatter` 类：该类提供了三种格式化方法：

- 预定义的标准格式。如：  
`ISO_LOCAL_DATE_TIME;ISO_LOCAL_DATE;ISO_LOCAL_TIME`
- 本地化相关的格式。如：`ofLocalizedDateTime(FormatStyle.LONG)`
- 自定义的格式。如：`ofPattern("yyyy-MM-dd hh:mm:ss")`

方法	描述
<code>ofPattern(String pattern)</code>	静态方法，返回一个指定字符串格式的 <code>DateTimeFormatter</code>
<code>format(TemporalAccessor t)</code>	格式化一个日期、时间，返回字符串
<code>parse(CharSequence text)</code>	将指定格式的字符序列解析为一个日期、时间

让天下没有难学的技术

## Java比较器

Java实现对象排序的方式有两种：

自然排序：`java.lang.Comparable`

- ### 方式一：自然排序：`java.lang.Comparable`
- `Comparable`接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序。
  - 实现 `Comparable` 的类必须实现 `compareTo(Object obj)` 方法，两个对象即通过 `compareTo(Object obj)` 方法的返回值来比较大小。**如果当前对象this大于形参对象obj，则返回正整数，如果当前对象this小于形参对象obj，则返回负整数，如果当前对象this等于形参对象obj，则返回零。**
  - 实现`Comparable`接口的对象列表（和数组）可以通过 `Collections.sort` 或 `Arrays.sort` 进行自动排序。实现此接口的对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器。
  - 对于类 C 的每一个 e1 和 e2 来说，当且仅当 `e1.compareTo(e2) == 0` 与 `e1.equals(e2)` 具有相同的 boolean 值时，类 C 的自然排序才叫做与 `equals` 一致。建议（虽然不是必需的）**最好使自然排序与 equals 一致**。

让天下没有难学的技术

定制排序：`java.util.Comparator`（匿名内部类）

## 方式二：定制排序：java.util.Comparator

- 当元素的类型没有实现java.lang.Comparable接口而又不方便修改代码，或者实现了java.lang.Comparable接口的排序规则不适合当前的操作，那么可以考虑使用 Comparator 的对象来排序，强行对多个对象进行整体排序的比较。
- 重写 compare(Object o1, Object o2)方法，比较o1和o2的大小：如果方法返回正整数，则表示o1大于o2；如果返回0，表示相等；返回负整数，表示o1小于o2。
- 可以将 Comparator 传递给 sort 方法（如 Collections.sort 或 Arrays.sort），从而允许在排序顺序上实现精确控制。
- 还可以使用 Comparator 来控制某些数据结构（如有序 set或有序映射）的顺序，或者为那些没有自然顺序的对象 collection 提供排序。

## System

---

该类的构造器是private的，所以无法创建该类的对象

其内部的成员变量和成员方法都是static的

- 成员变量
  - **System**类内部包含**in**、**out**和**err**三个成员变量，分别代表标准输入流(键盘输入)，标准输出流(显示器)和标准错误输出流(显示器)。
- 成员方法
  - **native long currentTimeMillis():**  
该方法的作用是返回当前的计算机时间，时间的表达格式为当前计算机时间和GMT时间(格林威治时间)1970年1月1号0时0分0秒所差的毫秒数。
  - **void exit(int status):**  
该方法的作用是退出程序。其中**status**的值为0代表正常退出，非零代表异常退出。**使用该方法可以在图形界面编程中实现程序的退出功能等。**

让天下没有难学的

## 9.5 System类



### ➤ **void gc():**

该方法的作用是请求系统进行垃圾回收。至于系统是否立刻回收，则取决于系统中垃圾回收算法的实现以及系统执行时的情况。

### ➤ **String getProperty(String key):**

该方法的作用是获得系统中属性名为**key**的属性对应的值。系统中常见的属性名以及属性的作用如下表所示：

属性名	属性说明
<code>java.version</code>	Java 运行时环境版本
<code>java.home</code>	Java 安装目录
<code>os.name</code>	操作系统的名称
<code>os.version</code>	操作系统的版本
<code>user.name</code>	用户的账户名称
<code>user.home</code>	用户的主目录
<code>user.dir</code>	用户的当前工作目录

## Math

提供了一系列静态方法用于科学计算。其方法的参数和返回值类型一般为double型。

## BigInteger

BigInteger可以表示不可变的任意精度的整数

构造器： `BigInteger(String val)`：根据字符串构建BigInteger对象

## ● 常用方法

- public BigInteger **abs()**: 返回此 BigInteger 的绝对值的 BigInteger。
- BigInteger **add(BigInteger val)** : 返回其值为 (this + val) 的 BigInteger
- BigInteger **subtract(BigInteger val)** : 返回其值为 (this - val) 的 BigInteger
- BigInteger **multiply(BigInteger val)** : 返回其值为 (this \* val) 的 BigInteger
- BigInteger **divide(BigInteger val)** : 返回其值为 (this / val) 的 BigInteger。整数相除只保留整数部分。
- BigInteger **remainder(BigInteger val)** : 返回其值为 (this % val) 的 BigInteger。
- BigInteger[] **divideAndRemainder(BigInteger val)**: 返回包含 (this / val) 后跟 (this % val) 的两个 BigInteger 的数组。
- BigInteger **pow(int exponent)** : 返回其值为 (this<sup>exponent</sup>) 的 BigInteger。

## BigDecimal

---

- 一般的Float类和Double类可以用来做科学计算或工程计算，但在**商业计算中，要求数字精度比较高，故用到java.math.BigDecimal类。**
- BigDecimal类支持不可变的、任意精度的有符号十进制定点数。
- 构造器
  - public BigDecimal(double val)
  - public BigDecimal(String val)
- 常用方法
  - public BigDecimal **add(BigDecimal augend)**
  - public BigDecimal **subtract(BigDecimal subtrahend)**
  - public BigDecimal **multiply(BigDecimal multiplicand)**
  - public BigDecimal **divide(BigDecimal divisor, int scale, int roundingMode)**

## 枚举类

---

1. 私有化类的构造器，保证不能在类的外部创建其对象
2. 在类的内部创建枚举类的实例。声明为： `public static final`
3. 对象如果有实例变量，应该声明为`private final`，并在构造器中初始化

```
class Season{  
    private final String SEASONNAME;//季节的名称  
    private final String SEASONDESC;//季节的描述  
    private Season(String seasonName, String seasonDesc){  
        this.SEASONNAME = seasonName;  
        this.SEASONDESC = seasonDesc;  
    }  
    public static final Season SPRING = new Season("春天", "春暖花开");  
    public static final Season SUMMER = new Season("夏天", "夏日炎炎");  
    public static final Season AUTUMN = new Season("秋天", "秋高气爽");  
    public static final Season WINTER = new Season("冬天", "白雪皑皑");  
}
```

- 使用 `enum` 定义的枚举类默认继承了 `java.lang.Enum` 类，因此不能再继承其他类
- 枚举类的构造器只能使用 `private` 权限修饰符
- 枚举类的所有实例必须在枚举类中显式列出(, 分隔 ; 结尾)。列出的实例系统会自动添加 `public static final` 修饰
- 必须在枚举类的第一行声明枚举类对象

●JDK 1.5 中可以在 `switch` 表达式中使用 `Enum` 定义的枚举类的对象作为表达式，`case` 子句可以直接使用枚举值的名字，无需添加枚举类作为限定。

让天下没有难学

### > 10.1.3 使用 `enum` 定义枚举类



```
public enum SeasonEnum {
    SPRING("春天", "春风又绿江南岸"),
    SUMMER("夏天", "映日荷花别样红"),
    AUTUMN("秋天", "秋水共长天一色"),
    WINTER("冬天", "窗含西岭千秋雪");

    private final String seasonName;
    private final String seasonDesc;
    private SeasonEnum(String seasonName, String seasonDesc) {
        this.seasonName = seasonName;
        this.seasonDesc = seasonDesc;
    }
    public String getSeasonName() {
        return seasonName;
    }
    public String getSeasonDesc() {
        return seasonDesc;
    }
}
```

让天下没有难学

## 注解

- 定义新的 Annotation 类型使用 **@interface** 关键字
- 自定义注解自动继承了 **java.lang.annotation.Annotation** 接口
- Annotation 的成员变量在 Annotation 定义中以无参数方法的形式来声明。其方法名和返回值定义了该成员的名字和类型。我们称为配置参数。类型只能是八种基本数据类型、**String** 类型、**Class** 类型、**enum** 类型、**Annotation** 类型、**以上所有类型的数组**。
- 可以在定义 Annotation 的成员变量时为其指定初始值，指定成员变量的初始值可使用 **default** 关键字
- 如果只有一个参数成员，建议使用 **参数名为 value**
- 如果定义的注解含有配置参数，那么使用时必须指定参数值，除非它有默认值。格式是“参数名 = 参数值”，如果只有一个参数成员，且名称为 **value**，可以省略“**value =**”
- 没有成员定义的 Annotation 称为 **标记**；包含成员变量的 Annotation 称为元数据 Annotation

注意：自定义注解必须配上注解的信息处理流程才有意义。

让天下没有难学的技术

### 10.2.3 自定义 Annotation



```

@MyAnnotation(value="尚硅谷")
public class MyAnnotationTest {
    public static void main(String[] args) {
        Class clazz = MyAnnotationTest.class;
        Annotation a = clazz.getAnnotation(MyAnnotation.class);
        MyAnnotation m = (MyAnnotation) a;
        String info = m.value();
        System.out.println(info);
    }
}

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{
    String value() default "auguigu";
}

```

元注解

● **@Retention**: 只能用于修饰一个 Annotation 定义, 用于指定该 Annotation 的生命周期, @Retention 包含一个 **RetentionPolicy** 类型的成员变量, 使用 @Retention 时必须为该 value 成员变量指定值:

- **RetentionPolicy.SOURCE**: 在源文件中有效 (即源文件保留), 编译器直接丢弃这种策略的注释
- **RetentionPolicy.CLASS**: 在 class 文件中有效 (即 class 保留), 当运行 Java 程序时, JVM 不会保留注解。这是默认值
- **RetentionPolicy.RUNTIME**: 在运行时有效 (即运行时保留), 当运行 Java 程序时, JVM 会保留注释。程序可以通过反射获取该注释。



● **@Target**: 用于修饰 Annotation 定义, 用于指定被修饰的 Annotation 能用于修饰哪些程序元素。 @Target 也包含一个名为 value 的成员变量。

取值 (ElementType)		取值 (ElementType)	
CONSTRUCTOR	用于描述构造器	PACKAGE	用于描述包
FIELD	用于描述域	PARAMETER	用于描述参数
LOCAL_VARIABLE	用于描述局部变量	TYPE	用于描述类、接口(包括注解类型)或enum声明
METHOD	用于描述方法		

● **@Documented**: 用于指定被该元 Annotation 修饰的 Annotation 类将被 javadoc 工具提取成文档。默认情况下, javadoc 是不包括注解的。

- 定义为 Documented 的注解必须设置 Retention 值为 RUNTIME。

● **@Inherited**: 被它修饰的 Annotation 将具有 **继承性**。如果某个类使用了被 @Inherited 修饰的 Annotation, 则其子类将自动具有该注解。

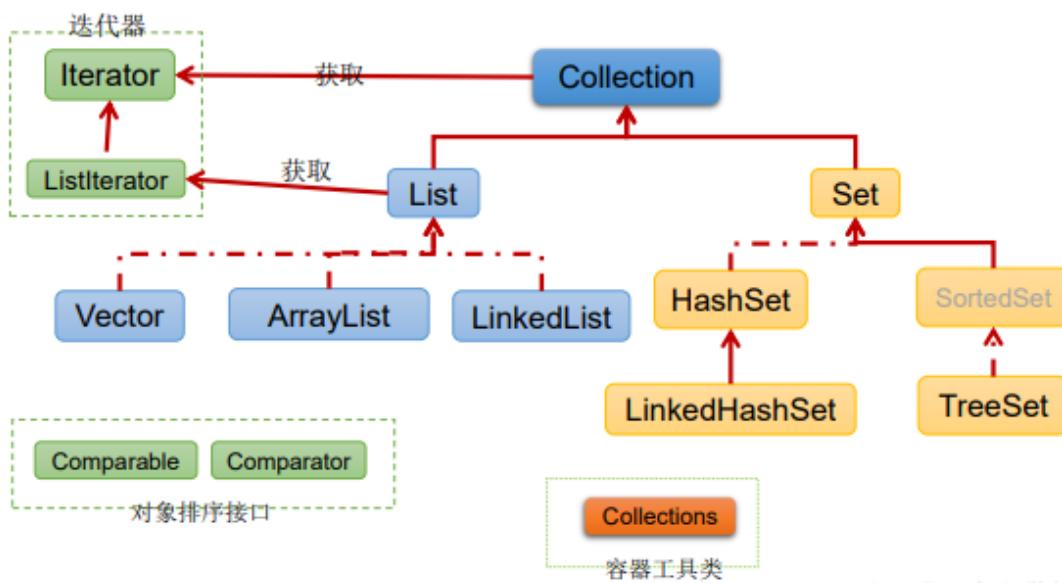
- 比如: 如果把标有 @Inherited 注解的自定义的注解标注在类级别上, 子类则可以继承父类类级别的注解
- 实际应用中, 使用较少

# Java集合



## 11.1 Java 集合框架概述：Collection接口继承树

尚硅谷



JDK提供的集合API位于java.util包内

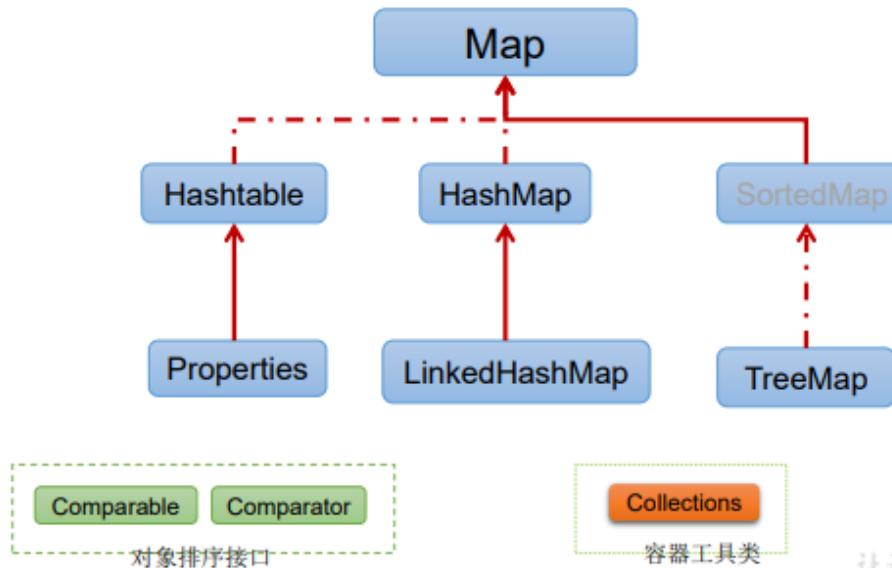
让天下没有难学的技术



## 11.1 Java 集合框架概述：Map接口继承树

尚硅谷

$$y = f(x); \\ y = x^2 + 3;$$



让天下没有难学的技术

## Collection接口

- 1、添加
  - `add(Object obj)`
  - `addAll(Collection coll)`
- 2、获取有效元素的个数
  - `int size()`
- 3、清空集合
  - `void clear()`
- 4、是否是空集合
  - `boolean isEmpty()`
- 5、是否包含某个元素
  - `boolean contains(Object obj)`: 是通过元素的`equals`方法来判断是否是同一个对象
  - `boolean containsAll(Collection c)`: 也是调用元素的`equals`方法来比较的。拿两个集合的元素挨个比较。

让天下没有难学的

## 11.2 Collection 接口方法



- 6、删除
  - `boolean remove(Object obj)` : 通过元素的`equals`方法判断是否是要删除的那个元素。只会删除找到的第一个元素
  - `boolean removeAll(Collection coll)`: 取当前集合的差集
- 7、取两个集合的交集
  - `boolean retainAll(Collection c)`: 把交集的结果存在当前集合中，不影响c
- 8、集合是否相等
  - `boolean equals(Object obj)`
- 9、转成对象数组
  - `Object[] toArray()`
- 10、获取集合对象的哈希值
  - `hashCode()`
- 11、遍历
  - `iterator()`: 返回迭代器对象，用于集合遍历

让天下没有难学的

## List接口

元素有序、且可重复

- List除了从Collection集合继承的方法外，List集合里添加了一些根据索引来操作集合元素的方法。

- void add(int index, Object ele):在index位置插入ele元素
- boolean addAll(int index, Collection eles):从index位置开始将eles中的所有元素添加进来
- Object get(int index):获取指定index位置的元素
- int indexOf(Object obj):返回obj在集合中首次出现的位置
- int lastIndexOf(Object obj):返回obj在当前集合中末次出现的位置
- Object remove(int index):移除指定index位置的元素，并返回此元素
- Object set(int index, Object ele):设置指定index位置的元素为ele
- List subList(int fromIndex, int toIndex):返回从fromIndex到toIndex位置的子集合

## ArrayList

本质上，ArrayList是对象引用的一个“变长”数组，基于**动态数组**

JDK1.8：ArrayList像懒汉式，一开始创建一个长度为0的数组，当添加第一个元素时再创建一个始容量为10的数组

Arrays.asList(...)方法返回的List集合，既不是ArrayList实例，也不是Vector实例。Arrays.asList(...)返回值是一个固定长度的List集合

ArrayList每次请求扩容其大小1.5倍的空间

## LinkedList

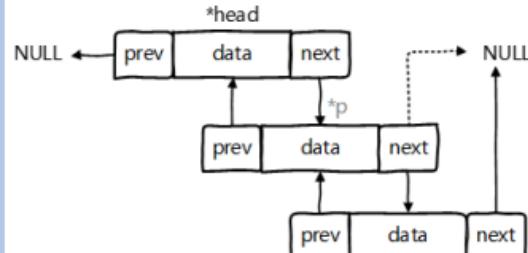
- 对于**频繁的插入或删除元素**的操作，建议使用LinkedList类，效率较高
- 新增方法：
  - void addFirst(Object obj)
  - void addLast(Object obj)
  - Object getFirst()
  - Object getLast()
  - Object removeFirst()
  - Object removeLast()



- **LinkedList: 双向链表**, 内部没有声明数组, 而是定义了Node类型的first和last, 用于记录首末元素。同时, 定义内部类Node, 作为LinkedList中保存数据的基本结构。Node除了保存数据, 还定义了两个变量:
  - prev变量记录前一个元素的位置
  - next变量记录下一个元素的位置

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```



让天下没有难学的技术

## Vector

- **Vector** 是一个古老的集合, JDK1.0就有了。大多数操作与ArrayList相同, 区别之处在于**Vector**是线程安全的。
- 在各种list中, 最好把ArrayList作为缺省选择。当插入、删除频繁时, 使用LinkedList; Vector总是比ArrayList慢, 所以尽量避免使用。
- 新增方法:
  - **void addElement(Object obj)**
  - **void insertElementAt(Object obj,int index)**
  - **void setElementAt(Object obj,int index)**
  - **void removeElement(Object obj)**
  - **void removeAllElements()**

Vector和ArrayList几乎是完全相同的, 唯一的区别在于Vector是同步类(synchronized), 属于强同步类。

Vector每次扩容请求其大小的2倍空间

## Set接口

无序不可重复

### HashSet

集合元素可以是 null

实质上就是放到HashMap的key中

HashSet 集合判断两个元素相等的标准: 两个对象通过 hashCode() 方法比较相等, 并且两个对象的 equals() 方法返回值也相等。

对于存放在Set容器中的对象, 对应的类一定要重写equals()和hashCode(Object obj)方法, 以实现对象相等规则。即: “相等的对象必须具有相等的散列码”。

## ●向HashSet中添加元素的过程：

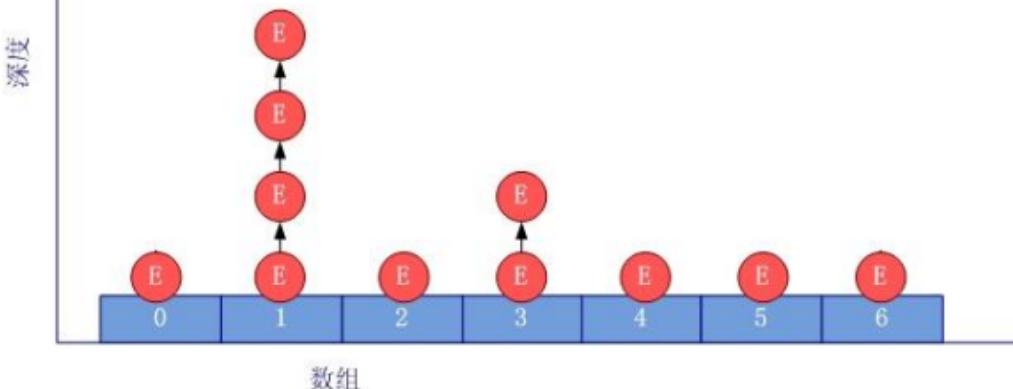
- 当向 HashSet 集合中存入一个元素时， HashSet 会调用该对象的 hashCode() 方法来得到该对象的 hashCode 值，然后根据 hashCode 值，通过某种散列函数决定该对象在 HashSet 底层数组中的存储位置。（这个散列函数会与底层数组的长度相计算得到在数组中的下标，并且这种散列函数计算还尽可能保证能均匀存储元素，越是散列分布，该散列函数设计的越好）
- 如果两个元素的 hashCode() 值相等，会再继续调用 equals 方法，如果 equals 方法结果为 true，添加失败；如果为 false，那么会保存该元素，但是该数组的位置已经有元素了，那么会通过链表的方式继续链接。
- 如果两个元素的 equals() 方法返回 true，但它们的 hashCode() 返回值不相等， HashSet 将会把它们存储在不同的位置，但依然可以添加成功。

让天下没有难学的技

## 11.5 Collection子接口之二： Set接口



### Set实现类之一： HashSet



底层也是数组，初始容量为16，当如果使用率超过0.75，（ $16 \times 0.75 = 12$ ）就会扩大容量为原来的2倍。（16扩容为32，依次为64,128....等）

### LinkedHashSet

- **LinkedHashSet** 是 **HashSet** 的子类
- **LinkedHashSet** 根据元素的 **hashCode** 值来决定元素的存储位置，但它同时使用双向链表维护元素的次序，这使得元素看起来是以**插入顺序保存的**。
- **LinkedHashSet****插入性能略低于 HashSet**，但在迭代访问 **Set** 里的全部元素时有很好的性能。
- **LinkedHashSet** 不允许集合元素重复。

让天下没有难学的技术

## 11.5 Collection子接口之二： Set接口



```
Set set = new LinkedHashSet();
set.add(new String("AA"));
set.add(456);
set.add(456);
set.add(new Customer("刘德华", 1001));
```

LinkedHashSet底层结构

让天下没有难学的技术

### TreeSet

- **TreeSet** 是 **SortedSet** 接口的实现类，**TreeSet** 可以确保集合元素处于排序状态。
- **TreeSet**底层使用**红黑树**结构存储数据
- 新增的方法如下： (了解)
  - **Comparator comparator()**
  - **Object first()**
  - **Object last()**
  - **Object lower(Object e)**
  - **Object higher(Object e)**
  - **SortedSet subSet(fromElement, toElement)**
  - **SortedSet headSet(toElement)**
  - **SortedSet tailSet(fromElement)**
- **TreeSet** 两种排序方法：**自然排序**和**定制排序**。默认情况下，**TreeSet** 采用自然排序。

# 排 序—自然排序

●**自然排序：**TreeSet 会调用集合元素的 `compareTo(Object obj)` 方法来比较元素之间的大小关系，然后将集合元素按升序(默认情况)排列

●如果试图把一个对象添加到 TreeSet 时，则该对象的类必须实现 Comparable 接口。

➤ 实现 Comparable 的类必须实现 `compareTo(Object obj)` 方法，两个对象即通过 `compareTo(Object obj)` 方法的返回值来比较大小。

● Comparable 的典型实现：

➤ BigDecimal、BigInteger 以及所有的数值型对应的包装类：按它们对应的数值大小进行比较

➤ Character：按字符的 unicode 值来进行比较

➤ Boolean：true 对应的包装类实例大于 false 对应的包装类实例

➤ String：按字符串中字符的 unicode 值进行比较

➤ Date、Time：后边的时间、日期比前面的时间、日期大

让天下没有难学的技术



## 11.5 Collection子接口之二：Set接口



# 排 序—自然排序

●向 TreeSet 中添加元素时，只有第一个元素无须比较 `compareTo()` 方法，后面添加的所有元素都会调用 `compareTo()` 方法进行比较。

●**因为只有相同类的两个实例才会比较大小，所以向 TreeSet 中添加的应该是同一个类的对象。**

●对于 TreeSet 集合而言，它**判断两个对象是否相等的唯一标准**是：两个对象通过 `compareTo(Object obj)` 方法比较返回值。

●当需要把一个对象放入 TreeSet 中，重写该对象对应的 `equals()` 方法时，应保证该方法与 `compareTo(Object obj)` 方法有一致的结果：如果两个对象通过 `equals()` 方法比较返回 true，则通过 `compareTo(Object obj)` 方法比较应返回 0。否则，让人难以理解。

# 排 序—定制排序

●TreeSet的自然排序要求元素所属的类实现Comparable接口，如果元素所属的类没有实现Comparable接口，或不希望按照升序(默认情况)的方式排列元素或希望按照其它属性大小进行排序，则考虑使用定制排序。定制排序，通过Comparator接口来实现。需要重写 `compare(T o1, T o2)` 方法。

●利用 `int compare(T o1, T o2)` 方法，比较 o1 和 o2 的大小：如果方法返回正整数，则表示 o1 大于 o2；如果返回 0，表示相等；返回负整数，表示 o1 小于 o2。

●要实现定制排序，需要将实现 Comparator 接口的实例作为形参传递给 TreeSet 的构造器。

●此时，仍然只能向 TreeSet 中添加类型相同的对象。否则发生 ClassCastException 异常。

●使用定制排序判断两个元素相等的标准是：通过 Comparator 比较两个元素返回了 0。

让天下没有难学的技术

# Map接口

- Map与Collection并列存在。用于保存具有**映射关系**的数据:key-value
- Map 中的 key 和 value 都可以是任何引用类型的数据
- Map 中的 key 用**Set**来存放, **不允许重复**, 即同一个 Map 对象所对应的类, 须重写hashCode()和equals()方法
- 常用String类作为Map的“键”
- key 和 value 之间存在单向一对一关系, 即通过指定的 key 总能找到唯一的、确定的 value
- Map接口的常用实现类: HashMap、TreeMap、LinkedHashMap和Properties。其中, **HashMap**是 Map 接口使用频率最高的实现类

## ● 添加、删除、修改操作:

- Object put(Object key, Object value): 将指定key-value添加到(或修改)当前map对象中
- void putAll(Map m): 将m中的所有key-value对存放到当前map中
- Object remove(Object key): 移除指定key的key-value对, 并返回value
- void clear(): 清空当前map中的所有数据

## ● 元素查询的操作:

- Object get(Object key): 获取指定key对应的value
- boolean containsKey(Object key): 是否包含指定的key
- boolean containsValue(Object value): 是否包含指定的value
- int size(): 返回map中key-value对的个数
- boolean isEmpty(): 判断当前map是否为空
- boolean equals(Object obj): 判断当前map和参数对象obj是否相等

## ● 元视图操作的方法:

- Set keySet(): 返回所有key构成的Set集合
- Collection values(): 返回所有value构成的Collection集合
- Set entrySet(): 返回所有key-value对构成的Set集合

# HashMap

- **HashMap**是 Map 接口使用频率最高的实现类。
- 允许使用null键和null值, 与HashSet一样, 不保证映射的顺序。
- 所有的key构成的集合是Set:无序的、不可重复的。所以, key所在的类要重写: equals()和hashCode()
- 所有的value构成的集合是Collection:无序的、可以重复的。所以, value所在的类要重写: equals()
- 一个key-value构成一个entry
- 所有的entry构成的集合是Set:无序的、不可重复的
- **HashMap 判断两个 key 相等的标准**是: 两个 key 通过 equals() 方法返回 true, hashCode 值也相等。
- **HashMap 判断两个 value相等的标准**是: 两个 value 通过 equals() 方法返回 true。

## HashMap源码中的重要常量

**DEFAULT\_INITIAL\_CAPACITY** : HashMap的默认容量, 16

**MAXIMUM\_CAPACITY** : HashMap的最大支持容量,  $2^{30}$

**DEFAULT\_LOAD\_FACTOR**: HashMap的默认加载因子

**TREEIFY\_THRESHOLD**: Bucket中链表长度大于该默认值, 转化为红黑树

**UNTREEIFY\_THRESHOLD**: Bucket中红黑树存储的Node小于该默认值, 转化为链表

**MIN\_TREEIFY\_CAPACITY**: 桶中的Node被树化时最小的hash表容量。(当桶中Node的数量大到需要变红黑树时, 若hash表容量小于**MIN\_TREEIFY\_CAPACITY**时, 此时应执行resize扩容操作这个**MIN\_TREEIFY\_CAPACITY**的值至少是**TREEIFY\_THRESHOLD**的4倍。)

**table**: 存储元素的数组, 总是2的n次幂

**entrySet**: 存储具体元素的集

**size**: HashMap中存储的键值对的数量

**modCount**: HashMap扩容和结构改变的次数。

**threshold**: 扩容的临界值, =容量\*填充因子

**loadFactor**: 填充因子

让天下没有难学的技术



- HashMap 的内部存储结构其实是**数组+链表+树的结合**。当实例化一个HashMap时，会初始化initialCapacity和loadFactor，在put第一对映射关系时，系统会创建一个长度为initialCapacity的Node数组，这个长度在哈希表中被称为容量(Capacity)，在这个数组中可以存放元素的位置我们称之为“桶”(bucket)，每个bucket都有自己的索引，系统可以根据索引快速的查找bucket中的元素。
- 每个bucket中存储一个元素，即一个Node对象，但每一个Node对象可以带一个引用变量next，用于指向下一个元素，因此，在一个桶中，就有可能生成一个Node链。也可能是一个一个TreeNode对象，每一个TreeNode对象可以有两个叶子结点left和right，因此，在一个桶中，就有可能生成一个TreeNode树。而新添加的元素作为链表的last，或树的叶子结点。

让天下没有难学的技术



### 那么HashMap什么时候进行扩容和树形化呢？

当HashMap中的元素个数超过数组大小(数组总大小length,不是数组中个数size)\*loadFactor时，就会进行数组扩容，loadFactor的默认值(DEFAULT\_LOAD\_FACTOR)为0.75，这是一个折中的取值。也就是说，默认情况下，数组大小(DEFAULT\_INITIAL\_CAPACITY)为16，那么当HashMap中元素个数超过 $16 * 0.75 = 12$ （这个值就是代码中的threshold值，也叫做临界值）的时候，就把数组的大小扩展为 $2 * 16 = 32$ ，即扩大一倍，然后重新计算每个元素在数组中的位置，而这是一个非常消耗性能的操作，所以如果我们已经预知HashMap中元素的个数，那么预设元素的个数能够有效的提高HashMap的性能。

当HashMap中的其中一个链的对象个数如果达到了8个，此时如果capacity没有达到64，那么HashMap会先扩容解决，如果已经达到了64，那么这个链会变成树，结点类型由Node变成TreeNode类型。当然，如果当映射关系被移除后，下次resize方法时判断树的结点个数低于6个，也会把树再转为链表。

## 关于映射关系的key是否可以修改? answer: 不要修改

映射关系存储到HashMap中会存储key的hash值，这样就不用在每次查找时重新计算每一个Entry或Node（TreeNode）的hash值了，因此如果已经put到Map中的映射关系，再修改key的属性，而这个属性又参与hashcode值的计算，那么会导致匹配不上。

### 总结: JDK1.8相较于之前的变化:

- 1.HashMap map = new HashMap(); //默认情况下，先不创建长度为16的数组
- 2.当首次调用map.put()时，再创建长度为16的数组
- 3.数组为Node类型，在jdk7中称为Entry类型
- 4.形成链表结构时，新添加的key-value对在链表的尾部（七上八下）
- 5.当数组指定索引位置的链表长度>8时，且map中的数组的长度> 64时，此索引位置上的所有key-value对使用红黑树进行存储。

让天下没有难学的技术

## 面试题：负载因子值的大小，对HashMap有什么影响

- 负载因子的大小决定了HashMap的数据密度。
- 负载因子越大密度越大，发生碰撞的几率越高，数组中的链表越容易长，造成查询或插入时的比较次数增多，性能会下降。
- 负载因子越小，就越容易触发扩容，数据密度也越小，意味着发生碰撞的几率越小，数组中的链表也就越短，查询和插入时比较的次数也越小，性能会更高。但是会浪费一定的内存空间。而且经常扩容也会影响性能，建议初始化预设大一点的空间。
- 按照其他语言的参考及研究经验，会考虑将负载因子设置为0.7~0.75，此时平均检索长度接近于常数。

## LinkedHashMap

- LinkedHashMap 是 HashMap 的子类
- 在HashMap存储结构的基础上，使用了一对双向链表来记录添加元素的顺序
- 与LinkedHashSet类似， LinkedHashMap 可以维护 Map 的迭代顺序：迭代顺序与 Key-Value 对的插入顺序一致

## TreeMap

### Map实现类之三： TreeMap

- TreeMap存储 Key-Value 对时，需要根据 key-value 对进行排序。 TreeMap 可以保证所有的 Key-Value 对处于**有序**状态。
- TreeSet底层使用**红黑树**结构存储数据
- TreeMap 的 Key 的排序：
  - **自然排序**： TreeMap 的所有的 Key 必须实现 Comparable 接口，而且所有的 Key 应该是同一个类的对象，否则将会抛出 ClassCastException
  - **定制排序**： 创建 TreeMap 时，传入一个 Comparator 对象，该对象负责对 TreeMap 中的所有 key 进行排序。此时不需要 Map 的 Key 实现 Comparable 接口
- TreeMap判断**两个key相等的标准**： 两个key通过compareTo()方法或者compare()方法返回0。

对于不了解者推荐的书

## Hashtable

- Hashtable是个古老的 Map 实现类，JDK1.0就提供了。不同于HashMap， Hashtable是线程安全的。
- Hashtable实现原理和HashMap相同，功能相同。底层都使用哈希表结构，查询速度快，很多情况下可以互用。
- 与HashMap不同， Hashtable 不允许使用 null 作为 key 和 value
- 与HashMap一样， Hashtable 也不能保证其中 Key-Value 对的顺序
- Hashtable判断两个key相等、两个value相等的标准，与HashMap一致。

## Properties

## Map实现类之五：Properties

- Properties 类是 Hashtable 的子类，该对象用于处理属性文件
- 由于属性文件里的 key、value 都是字符串类型，所以 Properties 里的 key 和 value 都是字符串类型
- 存取数据时，建议使用setProperty(String key, String value)方法和 getProperty(String key)方法

```
Properties pros = new Properties();
pros.load(new FileInputStream("jdbc.properties"));
String user = pros.getProperty("user");
System.out.println(user);
```

## Collections工具类

---

- Collections 是一个操作 Set、List 和 Map 等集合的工具类
- Collections 中提供了一系列静态的方法对集合元素进行排序、查询和修改等操作，还提供了对集合对象设置不可变、对集合对象实现同步控制等方法
- **排序操作：（均为static方法）**
  - reverse(List): 反转 List 中元素的顺序
  - shuffle(List): 对 List 集合元素进行随机排序
  - sort(List): 根据元素的自然顺序对指定 List 集合元素按升序排序
  - sort(List, Comparator): 根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
  - swap(List, int, int): 将指定 list 集合中的 i 处元素和 j 处元素进行交换

让天下没有难学的技术

## 11.7 Collections工具类



### Collections常用方法

#### 查找、替换

- Object max(Collection): 根据元素的自然顺序，返回给定集合中的最大元素
- Object max(Collection, Comparator): 根据 Comparator 指定的顺序，返回给定集合中的最大元素
- Object min(Collection)
- Object min(Collection, Comparator)
- int frequency(Collection, Object): 返回指定集合中指定元素的出现次数
- void copy(List dest, List src): 将src中的内容复制到dest中
- boolean replaceAll(List list, Object oldVal, Object newVal): 使用新值替换 List 对象的所有旧值

Collections 类中还提供了多个 synchronizedXxx() 方法，该方法可使将指定集合包装成线程同步的集合，从而可以解决多线程并发访问集合时的线程安全问题

## 泛型

声明在类、接口

用在变量类型、返回值类型、参数类型

#### 泛型方法

[访问权限] <泛型> 返回类型 方法名([泛型标识 参数名称]) 抛出的异常

- <? extends Number> (无穷小 , Number)  
只允许泛型为Number及Number子类的引用调用

- <? super Number> [Number , 无穷大)  
只允许泛型为Number及Number父类的引用调用

- <? extends Comparable>  
只允许泛型为实现Comparable接口的实现类的引用调用

## IO流

### File类

- **public File(String pathname)**

以pathname为路径创建File对象，可以是**绝对路径或者相对路径**，如果 pathname是相对路径，则默认的当前路径在系统属性user.dir中存储。

- 绝对路径：是一个固定的路径，从盘符开始
- 相对路径：是相对于某个位置开始

- **public File(String parent, String child)**

以parent为父路径， child为子路径创建File对象。

- **public File(File parent, String child)**

根据一个父File对象和子文件路径创建File对象

- 路径中的每级目录之间用一个**路径分隔符**隔开。

- 路径分隔符和系统有关：

- windows和DOS系统默认使用“\”来表示
- UNIX和URL使用“/”来表示

- Java程序支持跨平台运行，因此路径分隔符要慎用。

- 为了解决这个隐患，File类提供了一个常量：

**public static final String separator**。根据操作系统，动态的提供分隔符。

- 举例：

```
File file1 = new File("d:\\atguigu\\info.txt");
File file2 = new File("d:" + File.separator + "atguigu" + File.separator + "info.txt");
File file3 = new File("d:/atguigu");
```

## ● File类的获取功能

- `public String getAbsolutePath()`: 获取绝对路径
- `public String getPath()` : 获取路径
- `public String getName()` : 获取名称
- `public String getParent()`: 获取上层文件目录路径。若无，返回`null`
- `public long length()` : 获取文件长度（即：字节数）。不能获取目录的长度。
- `public long lastModified()` : 获取最后一次的修改时间，毫秒值
  
- `public String[] list()` : 获取指定目录下的所有文件或者文件目录的名称数组
- `public File[] listFiles()` : 获取指定目录下的所有文件或者文件目录的File数组

## ● File类的重命名功能

- `public boolean renameTo(File dest)`: 把文件重命名为指定的文件路径

让天下没有难学的课



## 13.1 File 类的使用：常用方法



## ● File类的判断功能

- `public boolean isDirectory()`: 判断是否是文件目录
- `public boolean isFile()` : 判断是否是文件
- `public boolean exists()` : 判断是否存在
- `public boolean canRead()` : 判断是否可读
- `public boolean canWrite()` : 判断是否可写
- `public boolean isHidden()` : 判断是否隐藏

## ● File类的创建功能

- `public boolean createNewFile()` : 创建文件。若文件存在，则不创建，返回`false`
- `public boolean mkdir()` : 创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。
- `public boolean mkdirs()` : 创建文件目录。如果上层文件目录不存在，一并创建

**注意事项：如果你创建文件或者文件目录没有写盘符路径，那么，默认在项目路径下。**

## ● File类的删除功能

- `public boolean delete()`: 删除文件或者文件夹  
删除注意事项：

Java中的删除不走[回收站](#)。

要删除一个文件目录，请注意该文件目录内不能包含文件或者文件目录

# 流的分类

# 流的分类

- 按操作**数据单位**不同分为：字节流(8 bit), 字符流(16 bit)
- 按数据流的**流向**不同分为：输入流，输出流
- 按流的**角色**的不同分为：节点流，处理流

(抽象基类)	字节流	字符流
输入流	<b>InputStream</b>	<b>Reader</b>
输出流	<b>OutputStream</b>	<b>Writer</b>

1. Java的IO流共涉及40多个类，实际上非常规则，都是从如下4个抽象基类派生的。
2. 由这四个类派生出来的子类名称都是以其父类名作为子类名后缀。

节点流：直接从数据源或目的地读写数据

处理流：不直接连接到数据源或目的地，而是“连接”在已存在的流（节点流或处理流）之上，通过对数据的处理为程序提供更为强大的读写功能。

## IO 流体系

分类	字节输入流	字节输出流	字符输入流	字符输出流
抽象基类	InputStream	OutputStream	Reader	Writer
访问文件	FileInputStream	FileOutputStream	FileReader	FileWriter
访问数组	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
访问管道	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
访问字符串			StringReader	StringWriter
缓冲流	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
转换流			InputStreamReader	OutputStreamWriter
对象流	ObjectInputStream	ObjectOutputStream		
	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
打印流		PrintStream		PrintWriter
推回输入流	PushbackInputStream		PushbackReader	
特殊流	DataInputStream	DataOutputStream		

## InputStream和Reader

- `InputStream` 和 `Reader` 是所有输入流的基类。
- `InputStream` (典型实现: `FileInputStream`)
  - `int read()`
  - `int read(byte[] b)`
  - `int read(byte[] b, int off, int len)`
- `Reader` (典型实现: `FileReader`)
  - `int read()`
  - `int read(char[] c)`
  - `int read(char[] c, int off, int len)`
- 程序中打开的文件 IO 资源不属于内存里的资源，垃圾回收机制无法回收该资源，所以应该显式关闭文件 IO 资源。
- `FileInputStream` 从文件系统中的某个文件中获得输入字节。`FileInputStream` 用于读取非文本数据之类的原始字节流。要读取字符流，需要使用 `FileReader`

## InputStream

### ● `int read()`

从输入流中读取数据的下一个字节。返回 0 到 255 范围内的 int 字节值。如果因为已经到达流末尾而没有可用的字节，则返回值 -1。

### ● `int read(byte[] b)`

从此输入流中将最多 `b.length` 个字节的数据读入一个 `byte` 数组中。如果因为已经到达流末尾而没有可用的字节，则返回值 -1。否则以整数形式返回实际读取的字节数。

### ● `int read(byte[] b, int off,int len)`

将输入流中最多 `len` 个数据字节读入 `byte` 数组。尝试读取 `len` 个字节，但读取的字节也可能小于该值。以整数形式返回实际读取的字节数。如果因为流位于文件末尾而没有可用的字节，则返回值 -1。

### ● `public void close() throws IOException`

关闭此输入流并释放与该流关联的所有系统资源。

让天下没有难学的技术



## 13.2 IO流原理及流的分类

尚硅谷

## Reader

### ● `int read()`

读取单个字符。作为整数读取的字符，范围在 0 到 65535 之间 (0x00-0xffff) (2 个字节的 Unicode 码)，如果已到达流的末尾，则返回 -1

### ● `int read(char[] cbuf)`

将字符读入数组。如果已到达流的末尾，则返回 -1。否则返回本次读取的字符数。

### ● `int read(char[] cbuf,int off,int len)`

将字符读入数组的某一部分。存到数组 `cbuf` 中，从 `off` 处开始存储，最多读 `len` 个字符。如果已到达流的末尾，则返回 -1。否则返回本次读取的字符数。

### ● `public void close() throws IOException`

关闭此输入流并释放与该流关联的所有系统资源。

## OutputStream和Writer

- OutputStream 和 Writer 也非常相似:

- `void write(int b/int c);`
- `void write(byte[] b/char[] cbuf);`
- `void write(byte[] b/char[] buff, int off, int len);`
- `void flush();`
- `void close();` 需要先刷新，再关闭此流

- 因为字符流直接以字符作为操作单位，所以 Writer 可以用字符串来替换字符数组，即以 String 对象作为参数

- `void write(String str);`
- `void write(String str, int off, int len);`

- FileOutputStream 从文件系统中的某个文件中获得输出字节。FileOutputStream 用于写出非文本数据之类的原始字节流。要写出字符流，需要使用 FileWriter

## OutputStream

- `void write(int b)`

将指定的字节写入此输出流。write 的常规协定是：向输出流写入一个字节。要写入的字节是参数 b 的八个低位。b 的 24 个高位将被忽略。即写入0~255范围的。

- `void write(byte[] b)`

将 b.length 个字节从指定的 byte 数组写入此输出流。write(b) 的常规协定是：应该与调用 write(b, 0, b.length) 的效果完全相同。

- `void write(byte[] b,int off,int len)`

将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此输出流。

- `public void flush()throws IOException`

刷新此输出流并强制写出所有缓冲的输出字节，调用此方法指示应将这些字节立即写入它们预期的目标。

- `public void close() throws IOException`

关闭此输出流并释放与该流关联的所有系统资源。

让天下没有难学的技术



## 13.2 IO流原理及流的分类



## Writer

- `void write(int c)`

写入单个字符。要写入的字符包含在给定整数值的 16 个低位中，16 高位被忽略。即写入0 到 65535 之间的Unicode码。

- `void write(char[] cbuf)`

写入字符数组。

- `void write(char[] cbuf,int off,int len)`

写入字符数组的某一部分。从off开始，写入len个字符

- `void write(String str)`

写入字符串。

- `void write(String str,int off,int len)`

写入字符串的某一部分。

- `void flush()`

刷新该流的缓冲，则立即将它们写入预期目标。

- `public void close() throws IOException`

关闭此输出流并释放与该流关联的所有系统资源。

让天下没有难学的技术

# 节点流(或文件流)

## 读取文件

1. 建立一个流对象，将已存在的一一个文件加载进流。

➤ `FileReader fr = new FileReader(new File("Test.txt"));`

2. 创建一个临时存放数据的数组。

➤ `char[] ch = new char[1024];`

3. 调用流对象的读取方法将流中的数据读入到数组中。

➤ `fr.read(ch);`

4. 关闭资源。

➤ `fr.close();`

让天下没有难学的技

## 13.3 节点流(或文件流)



```
FileReader fr = null;
try {
    fr = new FileReader(new File("c:\\test.txt"));
    char[] buf = new char[1024];
    int len;
    while ((len = fr.read(buf)) != -1) {
        System.out.print(new String(buf, 0, len));
    }
} catch (IOException e) {
    System.out.println("read-Exception :" + e.getMessage());
} finally {
    if (fr != null) {
        try {
            fr.close();
        } catch (IOException e) {
            System.out.println("close-Exception :" + e.getMessage());
        }
    }
}
```

## 写入文件

1. 创建流对象，建立数据存放文件

➤ `FileWriter fw = new FileWriter(new File("Test.txt"));`

2. 调用流对象的写入方法，将数据写入流

➤ `fw.write("atguigu-songhongkang");`

3. 关闭流资源，并将流中的数据清空到文件中。

➤ `fw.close();`

让天下没有难学

## 13.3 节点流(或文件流)



```
FileWriter fw = null;
try {
    fw = new FileWriter(new File("Test.txt"));
    fw.write("atguigu-songhongkang");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (fw != null)
        try {
            fw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
}
```

- › 定义文件路径时，注意：可以用“/”或者“\”。
- › 在写入一个文件时，如果使用构造器FileOutputStream(file)，则目录下有同名文件将被覆盖。
- › 如果使用构造器FileOutputStream(file,true)，则目录下的同名文件不会被覆盖，在文件内容末尾追加内容。
- › 在读取文件时，必须保证该文件已存在，否则报异常。
- › 字节流操作字节，比如：.mp3, .avi, .rmvb, mp4, jpg, .doc, .ppt
- › 字符流操作字符，只能操作普通文本文件。最常见的文本文件：.txt, .java, .c, .cpp 等语言的源代码。尤其注意.doc,excel,ppt这些不是文本文件。

## 缓冲流

- › 为了提高数据读写的速度，Java API提供了带缓冲功能的流类，在使用这些流类时，会创建一个内部缓冲区数组，缺省使用8192个字节(8Kb)的缓冲区。

```
public  
class BufferedInputStream extends FilterInputStream {  
  
    private static int DEFAULT_BUFFER_SIZE = 8192;
```

- › 缓冲流要“套接”在相应的节点流之上，根据数据操作单位可以把缓冲流分为：
  - **BufferedInputStream 和 BufferedOutputStream**
  - **BufferedReader 和 BufferedWriter**

- 当读取数据时，数据按块读入缓冲区，其后的读操作则直接访问缓冲区
- 当使用BufferedInputStream读取字节文件时，BufferedInputStream会一次性从文件中读取8192个(8Kb)，存在缓冲区中，直到缓冲区装满了，才重新从文件中读取下一个8192个字节数组。
- 向流中写入字节时，不会直接写到文件，先写到缓冲区中直到缓冲区写满，BufferedOutputStream才会把缓冲区中的数据一次性写到文件里。使用方法**flush()**可以强制将缓冲区的内容全部写入输出流
- 关闭流的顺序和打开流的顺序相反。只要关闭最外层流即可，关闭最外层流也会相应关闭内层节点流
- **flush()**方法的使用：手动将buffer中内容写入文件
- 如果是带缓冲区的流对象的**close()**方法，不但会关闭流，还会在关闭流之前刷新缓冲区，关闭后不能再写出

## 转换流

- 转换流提供了在字节流和字符流之间的转换
- Java API提供了两个转换流：
  - **InputStreamReader**: 将InputStream转换为Reader
  - **OutputStreamWriter**: 将Writer转换为OutputStream
- 字节流中的数据都是字符时，转成字符流操作更高效。
- 很多时候我们使用转换流来处理文件乱码问题。实现编码和解码的功能。

### InputStreamReader

- 实现将字节的输入流按指定字符集转换为字符的输入流。
  - 需要和InputStream “套接” 。
  - 构造器
    - **public InputStreamReader(InputStream in)**
    - **public InputStreamReader(InputStream in, String charsetName)**
- 如： Reader isr = new InputStreamReader(System.in,"gbk");
- 

让天下没有难学的课

## 13.5 处理流之二：转换流



### OutputStreamWriter

- 实现将字符的输出流按指定字符集转换为字节的输出流。
- 需要和OutputStream “套接” 。
- 构造器
  - **public OutputStreamWriter(OutputStream out)**
  - **public OutputStreamWriter(OutputStream out, String charsetName)**

## 标准输入、输出流

- `System.in`和`System.out`分别代表了系统标准的输入和输出设备
- 默认输入设备是：键盘，输出设备是：显示器
- `System.in`的类型是`InputStream`
- `System.out`的类型是`PrintStream`，其是`OutputStream`的子类  
`FilterOutputStream` 的子类
- 重定向：通过`System`类的`setIn`, `setOut`方法对默认设备进行改变。
  - `public static void setIn(InputStream in)`
  - `public static void setOut(PrintStream out)`

## 打印流

---

- 实现将**基本数据类型**的数据格式转化为**字符串**输出
- 打印流：**PrintStream**和**PrintWriter**
  - 提供了一系列重载的`print()`和`println()`方法，用于多种数据类型的输出
  - `PrintStream`和`PrintWriter`的输出不会抛出`IOException`异常
  - `PrintStream`和`PrintWriter`有自动`flush`功能
  - `PrintStream` 打印的所有字符都使用平台的默认字符编码转换为字节。  
在需要写入字符而不是写入字节的情况下，应该使用 `PrintWriter` 类。
  - `System.out`返回的是`PrintStream`的实例

## 数据流

---

- 为了方便地操作Java语言的基本数据类型和String的数据，可以使用数据流。
- 数据流有两个类：(用于读取和写出基本数据类型、String类的数据)
  - **DataInputStream** 和 **DataOutputStream**
  - 分别“套接”在 **InputStream** 和 **OutputStream** 子类的流上

### ● **DataInputStream**中的方法

boolean readBoolean()	byte readByte()
char readChar()	float readFloat()
double readDouble()	short readShort()
long readLong()	int readInt()
String readUTF()	void readFully(byte[] b)

### ● **DataOutputStream**中的方法

- 将上述的方法的read改为相应的write即可。

让天下没有难学的技术

## 对象流

### ● **ObjectInputStream**和**ObjectOutputStream**

- 用于存储和读取**基本数据类型**数据或**对象**的处理流。它的强大之处就是可以把Java中的对象写入到数据源中，也能把对象从数据源中还原回来。
- **序列化：**用**ObjectOutputStream**类**保存**基本类型数据或对象的机制
- **反序列化：**用**ObjectInputStream**类**读取**基本类型数据或对象的机制
- **ObjectOutputStream**和**ObjectInputStream**不能序列化**static**和**transient**修饰的成员变量

# 对象的序列化

- 对象序列化机制 允许把内存中的Java对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。//当其它程序获取了这种二进制流，就可以恢复成原来的Java对象
- 序列化的好处在于可将任何实现了Serializable接口的对象转化为字节数据，使其在保存和传输时可被还原
- 序列化是 RMI (Remote Method Invoke – 远程方法调用) 过程的参数和返回值都必须实现的机制，而 RMI 是 JavaEE 的基础。因此序列化机制是 JavaEE 平台的基础
- 如果需要让某个对象支持序列化机制，则必须让对象所属的类及其属性是可序列化的，为了让某个类是可序列化的，该类必须实现如下两个接口之一。否则，会抛出NotSerializableException异常
  - Serializable
  - Externalizable

让天下没有难学的技

## 13.9 处理流之六：对象流



### 对象的序列化

- 凡是实现Serializable接口的类都有一个表示序列化版本标识符的静态变量：
  - private static final long serialVersionUID;
  - serialVersionUID用来表明类的不同版本间的兼容性。简言之，其目的是以序列化对象进行版本控制，有关各版本反序列化时是否兼容。
  - 如果类没有显示定义这个静态常量，它的值是Java运行时环境根据类的内部细节自动生成的。若类的实例变量做了修改，serialVersionUID 可能发生变化。故建议，显式声明。
- 简单来说，Java的序列化机制是通过在运行时判断类的serialVersionUID来验证版本一致性的。在进行反序列化时，JVM会把传来的字节流中的serialVersionUID与本地相应实体类的serialVersionUID进行比较，如果相同就认为是一致的，可以进行反序列化，否则就会出现序列化版本不一致的异常。(InvalidCastException)

# 使用对象流序列化对象

- 若某个类实现了 `Serializable` 接口，该类的对象就是可序列化的：

- 创建一个 `ObjectOutputStream`
  - 调用 `ObjectOutputStream` 对象的 `writeObject(对象)` 方法输出可序列化对象
  - 注意写出一次，操作 `flush()` 一次

- 反序列化

- 创建一个 `ObjectInputStream`
  - 调用 `readObject()` 方法读取流中的对象

- 强调：如果某个类的属性不是基本数据类型或 `String` 类型，而是另一个引用类型，那么这个引用类型必须是可序列化的，否则拥有该类型的 `Field` 的类也不能序列化

让天下没有难学的技术



## 13.9 处理流之六：对象流



//序列化：将对象写入到磁盘或者进行网络传输。  
//要求对象必须实现序列化

```
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("data.txt"));
Person p = new Person("韩梅梅", 18, "中华大街", new Pet());
oos.writeObject(p);
oos.flush();
oos.close();
```

//反序列化：将磁盘中的对象数据源读出。

```
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("data.txt"));
Person p1 = (Person)ois.readObject();
System.out.println(p1.toString());
ois.close();
```

## 随机存取文件流

# RandomAccessFile 类

- RandomAccessFile 声明在java.io包下，但直接继承于java.lang.Object类。并且它实现了DataInput、DataOutput这两个接口，也就意味着这个类既可以读也可以写。
- RandomAccessFile 类支持“随机访问”的方式，程序可以直接跳到文件的任意地方来读、写文件
  - 支持只访问文件的部分内容
  - 可以向已存在的文件后追加内容
- RandomAccessFile 对象包含一个记录指针，用以标示当前读写处的位置。  
RandomAccessFile 类对象可以自由移动记录指针：
  - long getFilePointer(): 获取文件记录指针的当前位置
  - void seek(long pos): 将文件记录指针定位到 pos 位置

让天下没有难学的技术



## 13.10 随机存取文件流



# RandomAccessFile 类

## ● 构造器

- public RandomAccessFile(File file, String mode)
- public RandomAccessFile(String name, String mode)

## ● 创建 RandomAccessFile 类实例需要指定一个 mode 参数，该参数指定 RandomAccessFile 的访问模式：

- r: 以只读方式打开
- rw: 打开以便读取和写入
- rwd: 打开以便读取和写入；同步文件内容的更新
- rws: 打开以便读取和写入；同步文件内容和元数据的更新

- 如果模式为只读r，则不会创建文件，而是会去读取一个已经存在的文件，如果读取的文件不存在则会出现异常。如果模式为rw读写。如果文件不存在则会去创建文件，如果存在则不会创建。

我们可以用RandomAccessFile这个类，来实现一个多线程断点下载的功能，用过下载工具的朋友们都知道，下载前都会建立两个临时文件，一个是与被下载文件大小相同的空文件，另一个是记录文件指针的位置文件，每次暂停的时候，都会保存上一次的指针，然后断点下载的时候，会继续从上一次的地方下载，从而实现断点下载或上传的功能，有兴趣的朋友们可以自己实现下

# NIO

Java NIO (New IO, Non-Blocking IO)是从Java 1.4版本开始引入的一套新的IO API，可以替代标准的Java IO API。NIO与原来的IO有同样的作用和目的，但是使用的方式完全不同，NIO支持面向缓冲区的（IO是面向流的）、基于通道的IO操作。NIO将以更加高效的方式进行文件的读写操作。

Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。



## Path、Paths和Files核心API

- 早期的Java只提供了一个File类来访问文件系统，但File类的功能比较有限，所提供的方法性能也不高。而且，大多数方法在出错时仅返回失败，并不会提供异常信息。
- NIO.2为了弥补这种不足，引入了Path接口，代表一个平台无关的平台路径，描述了目录结构中文件的位置。Path可以看成是File类的升级版本，实际引用的资源也可以不存在。
- 在以前IO操作都是这样写的：

```
import java.io.File;  
File file = new File("index.html");
```
- 但在Java7中，我们可以这样写：

```
import java.nio.file.Path;  
import java.nio.file.Paths;  
Path path = Paths.get("index.html");
```

让天下没有难学的技术



## Path、Paths和Files核心API

- 同时，NIO.2在java.nio.file包下还提供了Files、Paths工具类，Files包含了大量静态的工具方法来操作文件；Paths则包含了两个返回Path的静态工厂方法。
- Paths类提供的静态get()方法用来获取Path对象：
  - static Path get(String first, String ... more)：用于将多个字符串串连成路径
  - static Path get(URI uri)：返回指定uri对应的Path路径

# Path接口

## ● Path 常用方法:

- String `toString()` : 返回调用 Path 对象的字符串表示形式
- boolean `startsWith(String path)` : 判断是否以 path 路径开始
- boolean `endsWith(String path)` : 判断是否以 path 路径结束
- boolean `isAbsolute()` : 判断是否是绝对路径
- Path `getParent()` : 返回Path对象包含整个路径, 不包含 Path 对象指定的文件路径
- Path `getRoot()` : 返回调用 Path 对象的根路径
- Path `getFileName()` : 返回与调用 Path 对象关联的文件名
- int `getNameCount()` : 返回Path 根目录后面元素的数量
- Path `getName(int idx)` : 返回指定索引位置 idx 的路径名称
- Path `toAbsolutePath()` : 作为绝对路径返回调用 Path 对象
- Path `resolve(Path p)` : 合并两个路径, 返回合并后的路径对应的Path对象
- File `toFile()`: 将Path转化为File类的对象

让天下没有难学的技术

## 13-11 NIO.2中Path、Paths、Files类的使用



### Files 类

#### ● java.nio.file.Files 用于操作文件或目录的工具类。

#### ● Files常用方法:

- Path `copy(Path src, Path dest, CopyOption ... how)` : 文件的复制
- Path `createDirectory(Path path, FileAttribute<?> ... attr)` : 创建一个目录
- Path `createFile(Path path, FileAttribute<?> ... arr)` : 创建一个文件
- void `delete(Path path)` : 删除一个文件/目录, 如果不存在, 执行报错
- void `deleteIfExists(Path path)` : Path对应的文件/目录如果存在, 执行删除
- Path `move(Path src, Path dest, CopyOption...how)` : 将 src 移动到 dest 位置
- long `size(Path path)` : 返回 path 指定文件的大小

## Files 类

- Files常用方法: 用于判断
  - boolean exists(Path path, LinkOption ... opts) : 判断文件是否存在
  - boolean isDirectory(Path path, LinkOption ... opts) : 判断是否是目录
  - boolean isRegularFile(Path path, LinkOption ... opts) : 判断是否是文件
  - boolean isHidden(Path path) : 判断是否是隐藏文件
  - boolean isReadable(Path path) : 判断文件是否可读
  - boolean isWritable(Path path) : 判断文件是否可写
  - boolean notExists(Path path, LinkOption ... opts) : 判断文件是否不存在
- Files常用方法: 用于操作内容
  - SeekableByteChannel newByteChannel(Path path, OpenOption...how) : 获取与指定文件的连接, how 指定打开方式。
  - DirectoryStream<Path> newDirectoryStream(Path path) : 打开 path 指定的目录
  - InputStream newInputStream(Path path, OpenOption...how):获取 InputStream 对象
  - OutputStream newOutputStream(Path path, OpenOption...how) : 获取 OutputStream 对象

## 网络编程

### 如何实现网络中的主机互相通信

#### ● 通信双方地址

- IP
- 端口号

#### ● 一定的规则 (即: 网络通信协议。有两套参考模型)

- OSI参考模型: 模型过于理想化, 未能在因特网上进行广泛推广
- TCP/IP参考模型(或TCP/IP协议): 事实上的国际标准。

端口号与IP地址的组合得出一个网络套接字: Socket

## 反射

反射机制允许程序在执行期 借助于Reflection API取得任何类的内部信息, 并能直接操作任意对象的内部属性及方法。

加载完类之后，在堆内存的方法区中就产生了一个Class类型的对象（一个类只有一个Class对象），这个对象就包含了完整的类的结构信息。我们可以通过这个对象看到类的结构。**这个对象就像一面镜子，透过这个镜子看到类的结构，所以，我们形象的称之为：反射。**

正常方式：**引入需要的“包类”名称** → **通过new实例化** → **取得实例化对象**

反射方式：**实例化对象** → **getClass()方法** → **得到完整的“包类”名称**

## Class 类

- 对象照镜子后可以得到的信息：某个类的属性、方法和构造器、某个类到底实现了哪些接口。对于每个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个结构(class/interface/enum/annotation/primitive type/void/[])的有关信息。

- Class本身也是一个类
- Class 对象只能由系统建立对象
- 一个加载的类在 JVM 中只会有一个Class实例
- 一个Class对象对应的是一个加载到JVM中的一个.class文件
- 每个类的实例都会记得自己是由哪个 Class 实例所生成
- 通过Class可以完整地得到一个类中的所有被加载的结构
- Class类是Reflection的根源，针对任何你想动态加载、运行的类，唯有先获得相应的 Class对象

让天下没有难学的技术



### 15.2 理解Class类并获取Class的实例



#### Class类的常用方法

方法名	功能说明
static Class forName(String name)	返回指定类名 name 的 Class 对象
Object newInstance()	调用缺省构造函数，返回该Class对象的一个实例
getName()	返回此Class对象所表示的实体（类、接口、数组类、基本类型或void）名称
Class getSuperClass()	返回当前Class对象的父类的Class对象
Class [] getInterfaces()	获取当前Class对象的接口
ClassLoader getClassLoader()	返回该类的类加载器
Class getSuperclass()	返回表示此Class所表示的实体的超类的Class
Constructor[] getConstructors()	返回一个包含某些Constructor对象的数组
Field[] getDeclaredFields()	返回Field对象的一个数组
Method getMethod(String name,Class ... paramTypes)	返回一个Method对象，此对象的形参类型为paramType

#### 获取Class实例

## 获取Class类的实例(四种方法)

1) 前提: 若已知具体的类, 通过类的class属性获取, 该方法最为安全可靠, 程序性能最高

实例: `Class clazz = String.class;`

2) 前提: 已知某个类的实例, 调用该实例的getClass()方法获取Class对象

实例: `Class clazz = "www.atguigu.com".getClass();`

3) 前提: 已知一个类的全类名, 且该类在类路径下, 可通过Class类的静态方法forName()获取, 可能抛出ClassNotFoundException

实例: `Class clazz = Class.forName("java.lang.String");`

4) 其他方式(不做要求)

```
ClassLoader cl = this.getClass().getClassLoader();
```

```
Class clazz4 = cl.loadClass("类的全类名");
```

Navigation bar: back forward search history refresh

## 创建运行时类的对象

### 有了Class对象, 能做什么?

创建类的对象: 调用Class对象的newInstance()方法

要求: 1) 类必须有一个无参数的构造器。  
2) 类的构造器的访问权限需要足够。

难道没有无参的构造器就不能创建对象了吗?

不是! 只要在操作的时候明确的调用类中的构造器, 并将参数传递进去之后, 才可以实例化操作。

步骤如下:

- 1) 通过Class类的`getDeclaredConstructor(Class ... parameterTypes)`取得本类的指定形参类型的构造器
- 2) 向构造器的形参中传递一个对象数组进去, 里面包含了构造器中所需的各个参数。
- 3) 通过Constructor实例化对象。

在 Constructor 类中存在一个方法: +

```
public T newInstance(Object... initargs) {
```

以上是反射机制应用最多的地方。

北京大学出版社

## 获取运行时类的完整结构

使用反射可以取得：

### 1. 实现的全部接口

➤ `public Class<?>[] getInterfaces()`

确定此对象所表示的类或接口实现的接口。

### 2. 所继承的父类

➤ `public Class<? Super T> getSuperclass()`

返回表示此 Class 所表示的实体（类、接口、基本类型）的父类的 Class。

让天下没有难学的技术

## 15.5 获取运行时类的完整结构



### 3. 全部的构造器

➤ `public Constructor<T>[] getConstructors()`

返回此 Class 对象所表示的类的所有 public 构造方法。

➤ `public Constructor<T>[] getDeclaredConstructors()`

返回此 Class 对象表示的类声明的所有构造方法。

● Constructor 类中：

➤ 取得修饰符： `public int getModifiers();`

➤ 取得方法名称： `public String getName();`

➤ 取得参数的类型： `public Class<?>[] getParameterTypes();`

## 4.全部的方法

➤ `public Method[] getDeclaredMethods()`

返回此Class对象所表示的类或接口的全部方法

➤ `public Method[] getMethods()`

返回此Class对象所表示的类或接口的public的方法

● Method类中：

➤ `public Class<?> getReturnType()`取得全部的返回值

➤ `public Class<?>[] getParameterTypes()`取得全部的参数

➤ `public int getModifiers()`取得修饰符

➤ `public Class<?>[] getExceptionTypes()`取得异常信息

注释1

## ➤ 15.5 获取运行时类的完整结构

## 5.全部的Field

➤ `public Field[] getFields()`

返回此Class对象所表示的类或接口的public的Field。

➤ `public Field[] getDeclaredFields()`

返回此Class对象所表示的类或接口的全部Field。

● Field方法中：

➤ `public int getModifiers()` 以整数形式返回此Field的修饰符

➤ `public Class<?> getType()` 得到Field的属性类型

➤ `public String getName()` 返回Field的名称。

## 6. Annotation相关

- `get Annotation(Class<T> annotationClass)`
- `getDeclaredAnnotations()`

## 7. 泛型相关

获取父类泛型类型: `Type getGenericSuperclass()`

泛型类型: `ParameterizedType`

获取实际的泛型类型参数数组: `getActualTypeArguments()`

## 8. 类所在的包 `Package getPackage()`

### 调用运行时类的指定结构

#### 1. 调用指定方法

通过反射，调用类中的方法，通过`Method`类完成。步骤：

1. 通过`Class`类的`getMethod(String name, Class...parameterTypes)`方法取得一个`Method`对象，并设置此方法操作时所需要的参数类型。
2. 之后使用`Object invoke(Object obj, Object[] args)`进行调用，并向方法中传递要设置的`obj`对象的参数信息。



让天下没有难学的技术

### 15.6 调用运行时类的指定结构



#### `Object invoke(Object obj, Object ... args)`

**说明：**

1. `Object` 对应原方法的返回值，若原方法无返回值，此时返回`null`
2. 若原方法若为静态方法，此时形参`Object obj`可为`null`
3. 若原方法形参列表为空，则`Object[] args`为`null`
4. 若原方法声明为`private`, 则需要在调用此`invoke()`方法前，显式调用方法对象的`setAccessible(true)`方法，将可访问`private`的方法。

## 2. 调用指定属性

在反射机制中，可以直接通过Field类操作类中的属性，通过Field类提供的set()和get()方法就可以完成设置和取得属性内容的操作。

- `public Field getField(String name)` 返回此Class对象表示的类或接口的指定的public的Field。
- `public Field getDeclaredField(String name)` 返回此Class对象表示的类或接口的指定的Field。

### ● 在Field中：

- `public Object get(Object obj)` 取得指定对象obj上此Field的属性内容
- `public void set(Object obj, Object value)` 设置指定对象obj上此Field的属性内容

让天下没有难学的技术

## 15.6 调用运行时类的指定结构



### 关于`setAccessible`方法的使用

- Method和Field、Constructor对象都有`setAccessible()`方法。
- `setAccessible`启动和禁用访问安全检查的开关。
- 参数值为true则指示反射的对象在使用时应该取消Java语言访问检查。
  - 提高反射的效率。如果代码中必须用反射，而该句代码需要频繁的被调用，那么请设置为true。
  - 使得原本无法访问的私有成员也可以访问
- 参数值为false则指示反射的对象应该实施Java语言访问检查。

## 应用：动态代理

动态代理是指客户通过代理类来调用其它对象的方法，并且是在程序运行时根据需要动态创建目标类的代理对象。

## Java8新特性

### Lambda表达式

Lambda是一个匿名函数，我们可以把Lambda表达式理解为是一段可以传递的代码（将代码像数据一样进行传递）。使用它可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使Java的语言表达能力得到了提升。