# JAVA类介绍

## Java.lang

### Object

Object类是所有类的终极父类。 任何一个类都继承了Object类。

构造方法

无参构造方法：Object();

普通方法：11个普通方法：常用方法：toString、equals、hashCode

1）toString(); 返回该对象的字符串表示。

**Object 的toString() 有何作用？**

可以重写toString：打印输出一个对象的时候，就会输出符合我们所需求的格式数据。

**System.out.println()方法原理？**

为什么直接输出一个对象的时候和输出对象的toString方法返回的字符串结果是一样的呢？

通过查看println(o)源代码得知:直接输出一个对象的时候，实际上在println方法内部会调用toString方法，把toString方法返回的内容输出。

**toString()没有重写此方法返回的字符串表示： 完整类名+@+ 对象的哈希码值**

Object类源代码：

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

对象调用实现的效果：

System.out.println(o.toString()); //打印结果： java.lang.Object@18b3364

System.out.println(o); //打印结果： [java.lang.Object@18b3364](mailto:java.lang.Object@18b3364)

java内置的很多类 都重写了toString()方法：

如String类重写了此方法：

public String toString() {

return this;

}

自己创建类进行重写：

public String toString() {

//return super.toString();//进行重写eclipse会给你默认调用父类object操作：

//return getClass().getName() + "@" + Integer.toHexString(hashCode());

//A a=new A()，直接通过System.out.println(a);来打印成员变量id和那么的值

return “id:”+this.id+” name:”+this.name;

}

2）boolean equals(Object obj) 默认比较的是两个对象的内存地址

Equals比较的时候建议常量开头调用equals，可以避免空指针异常：”中国”.equals(star)

“**字符串**”重写了此方法:比较的是内容

重写equals方法：进行自己创建的Person对象成员变量id值对比

public boolean equals(Object obj) { //传入对象

Person p = (Person)obj; //强制类型转换

return this.id== p.id;

}

3）int hashCode() 返回该对象的哈希码值。 想象成对象的内存地址

重写hashCode方法：在自己创建的类Person中对比id即可

public int hashCode() {

return this.id;

}

4）protected Object clone() 创建并返回此对象的一个副本。

使用用途：当你需要使用此对象中的数据，但是又不希望修改此对象的数据时，

浅克隆：复制被克隆的对象，等于拷贝对象内容，两个内存地址不一样的对象，如果被克隆对象维护了另一个对象，是克隆不了维护中的对象的，

此方法的修饰符是protected（只有他自身或者继承他的类可以用）

第一步：在要被克隆的对象中重写Object中的clone的方法：把权限修饰符protected改为public即可：

public Object clone() throws CloneNotSupportedException {}

(重写父类的方法可以权限子类比父类大，但不能比父类小)

第二步：被克隆的对象要实现接口：标识符 Cloneable类：

public class User implements Cloneable{}

第三步：操作，(被克隆对象没有实现Cloneable接口异常：CloneNotSupportedException)

public static void main(String[] args) {

User u= new User();

u.setId(555);

u.setName("温文亮");

User p= new User();

try {

p=(User) u.clone();//返回的是Object

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

p.setName("你好");

System.out.println("u"+u.toString());

System.out.println("p"+p.toString());

5）protected void finalize()无用方法（了解即可）权限是protected

如果一个对象被垃圾回收器回收的时候，会先调用此方法

System.gc() 建议jvm虚拟机赶快启动垃圾回收器回收垃圾也会调用此方法

6）Class<?> getClass() 返回此 Object 的运行时类。

返回二进制Class对象

7）void notify() 唤醒 唤醒线程池等待线程其中的一个。

8）void notifyAll() 唤醒在此对象监视器上等待的所有线程。

9）void wait()等待 如果线程执行了wait方法，那么该线程会进入等待的状态，等待状态下的线程必须要被其他线程调用notify方法才能唤醒。

10）void wait(long timeout) 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线

程等待。

11）void wait(long timeout, int nanos)在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

7到11的方法都是需要在：同步锁中使用synchronized，

wait与notify方法要注意的事项：

1. wait方法与notify方法是属于Object对象 的。

2. wait方法与notify方法必须要在同步代码块或者是同步函数中才能 使用。

3. wait方法与notify方法必需要由锁对象调用。

synchronized(“这里放置的锁对象”)如对象是 a，那么在同步代码块中或同步方法中调用：a.notify() a.wait()

**例子:两个线程：一个打印，一个赋值，效果：打印生产水果，输出一个水果，不要出现线程安全问题**

第一步 定义一个简单的实体类 entity

有2个成员变量:name,sex

同时有2个线程,对资源中的变量操作

1个线程对name,age赋值，1个线程对name,age做变量的输出打印

public class Entity {

public String name;

public String sex;

public boolean flag = false;

}

第二步 输出线程，对资源对象Resource中成员变量,输出值

public class Output implements Runnable {

private Entity r ;

public Output(Entity r){ //通过构造方法传入相同的entity对象

this.r = r;

}

public void run() {

while(true){

synchronized(r){ //r的对象需要两个共同的对象：entity类

//判断标记,是false,等待

if(!r.flag){

try{r.wait();}catch(Exception ex){} //需要操作的共同对象调用此方法

}

System.out.println(r.name+".."+r.sex);

//标记改成false,唤醒对方线程

r.flag = false;

r.notify(); //需要操作的共同对象调用此方法

第三步 赋值线程类：对Entity中的成员变量赋值

public class Input implements Runnable {

private Entity r ;

public Input(Entity r){//在调用中通过构造方法传入相同的entity对象

this.r = r;

}

public void run() {

int i = 0 ;

while(true){

synchronized(r){

//标记是true,等待

if(r.flag){ //布尔值它定义在了实体类中

try{r.wait();}catch(Exception ex){}

}

if(i%2==0){ //通过取模的值，模拟生产水果

r.name = "苹果";

r.sex = "10元";

}else{

r.name = "雪梨";

r.sex = "20元";

}

//将对方线程唤醒,标记改为true

r.flag = true;//共同对象Entity

r.notify();

}

i++;

}

}

第四步 最后调用

public static void main(String[] args) {

Entity r = new Entity ();//创建两个线程要操作的对象

Input in = new Input(r);//两个线程有构造方法传入对象

Output out = new Output(r); //两个线程有构造方法传入对象

Thread tin = new Thread(in);//创建线程把继承了线程的传进入，这样才是在相同环境下操作的

Thread tout = new Thread(out);

tin.start();

tout.start();

}

### System

此类都是静态方法，静态字段（成员变量），无构造方法

静态字段（成员变量）

System.in,：标准的输入流，获得键盘输入的值：

System.out,：标准的输出流,在控制台打印信息

System.err标准的错误输出流

例子：获取键盘的值打印到txt文本中

第一步 获取键盘输入流

InputStream in=System.in;//获取键盘输入值

第二步 创建输出流

FileOutputStream out= new FileOutputStream("g:/a.txt");

while(true) {

第三步：接收数据：这里是把数据用int接收（在控制台进行输入内容，按回车键输出）

int a=in.read();

out.write(a);

out.flush();

//如果控制台输出单个'a'打印：c的值：97 ,c的值：13, c的值：10，打印3次，因为： //回车键\r（CR）的ASCII码值是13，\n换行键（LF）的ASCII码值是10.

//Windows 默认的回车代表的是 \r\n 也就是 ASCII 码中的 13 10

//如果你用的是 Linux，那就使用 \n ，即 10

System.out.println("a的值："+a); a：97

}

静态方法 7个方法

1）arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。

int[] i = {1,4,5,7,8};

int[] s=new int[4];

//源数组，索引起始位置，目标数组，索引起始位置，总共要复制几个索引

System.arraycopy(i, 1, s, 0, 3);

System.out.println(Arrays.toString(s));//打印：[4, 5, 7, 8]

2）static long currentTimeMillis() 返回以毫秒为单位的当前时间。

long i= System.currentTimeMillis();

System.out.println(i);//打印：1517839497791，这是毫秒值，配置simper

Date d=new Date(i);

String dd=new SimpleDateFormat("yyyy-MM-dd HH:hh:mm").format(d);

System.out.println(dd);打印：2018-02-05 22:10:15

3）Exit(int status) 退出jvm 如果参数是0表示正常退出，非0表示异常退出

说明：0和非0的数据都可以退出jvm，对于用户\程序员而言没有任何区别，对windows有意义，

System.exit(0);正常退出，后面代码不能执行

System.exit(1);正常退出，后面代码不能执行

4）gc()建议jvm虚拟机赶快启动垃圾回收器回收垃圾

System.gc();

5）getenv(String name) 获取指定的环境变量值

String a=System.getenv("JAVA\_HOME");

System.out.println(a);//打印 C:\Program Files\Java\jdk1.8.0\_152

6) getProperties() 确定当前的系统属性。全部windows系统属性，

Properties p=System.getProperties();

p.list(System.out);//打印到控制台，非常多信息，不写了

7）getProperty(String key)获取指定键指示的系统属性。

如果不知道传什么参数，可以通过6的方法打印出全部，再查看key

String s=System. getProperty(“os.name”)//打印：window 7

### Runtime

1、该类主要代表了应用程序运行的环境，没有构造方法

2、Runtime类封装了运行时的环境。每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接。

3、不能实例化Runtime对象，应用程序也不能创建自己的 Runtime 类实例，但可以通过 getRuntime 方法获取当前Runtime运行时对象的引用。一旦得到了一个当前的Runtime对象的引用，就可以调用Runtime对象的方法去控制Java虚拟机的状态和行为。

4、 当Applet和其他不被信任的代码调用任何Runtime方法时，常常会引起SecurityException异常。

普通方法

1）Runtime getRuntime()获得此类的获取当前Runtime运行时对象的引用。

Runtime runtime=Runtime.getRuntime();

2）Exec【执行】(String command) 运行windows的可执行文件： exe文件，返回当前的进程process对象，

6个重载方法

//获得Runtime对象

Runtime runtime=Runtime.getRuntime();

Process p=runtime.exec("c:/windows/notepad.exe");

Thread.sleep(3000);//让它延迟3秒就可以看到效果

p.destroy();//杀死此windows的进程

3）maxMemory()返回 Java 虚拟机试图使用的最大内存量。Jdk1.7以后不会一次性给你默认的64M内存空间，而是智能的增长

Java提供了无用单元自动收集机制。通过totalMemory()和freeMemory()方法可以知道对象的堆内存有多大，还剩多少。

Runtime runtime=Runtime.getRuntime();

long l1 =runtime.maxMemory();

System.out.println(l1);打印：1883242496我的是这值，根据情况而定

freeMemory()返回 Java 虚拟机中的空闲内存量。

Runtime runtime=Runtime.getRuntime();

long l=runtime.freeMemory();System.out.println(l); //打印：126930056 根据现实情况而定

totalMemory()返回 Java 虚拟机中的内存总量

Runtime runtime=Runtime.getRuntime();

System.out.println(runtime.totalMemory());打印：128974848

### Process

从1.5开始， ProcessBuilder.start()是创建Process的首选方式。

此类都是关系到执行windows可运行文件，此类都是要和Runtime配合着用，

它只有一个无参的构造方法，不能进行创建对象，是抽象类

方法只有6个都是抽象方法

普通方法

1）abstract int waitFor()等待程序直到子程序结束：打开了windows的记事本后关掉，就会继续往下执行代码

2）abstract int exitValue()返回子进程的出口值。

两个方法的例子

//获得Runtime

Runtime r = Runtime.getRuntime();

Process p = r.exec("notepad");

//等待程序直到子程序结束,意思就是你打开了windows的记事本后关掉，就会继续往下执行代码

p.waitFor();

//exitValue()方法返回子进程结束时的值：如果没有错误，将返回0，否则返回非0

System.out.println("Notepad returned " + p.exitValue());

//打印：Notepad returned： 0

3）abstract void destroy()杀掉子进程

//获得Runtime对象

Runtime runtime=Runtime.getRuntime();

Process p=runtime.exec("c:/windows/notepad.exe");

Thread.sleep(3000);//让它延迟3秒就可以看到效果

p.destroy();//杀死此windows的进程

4）abstract InputStream getErrorStream()获取子进程的错误流。

5）abstract InputStream getInputStream()获取子进程的输入流。

6）abstract OutputStream getOutputStream()获取子进程的输出流。

### Thread

Thread是继承了Runnable接口

进程可以说代表一个应用程序，（360有扫毒，清理垃圾）线程是执行程序代码，

一个程序至少有一个进程,一个进程至少有一个线程？

线程是一个动作,OS负责配发一个独立的CPU时间片(或者一个核,由OS来抽象)来执行这个动作.从这个意义来说,他是动态的.

二者虽然都有一个"程",但从OS的角度来看区别还是很大的.到了OS底层,没有进程和线程的概念.只有"指令"+"数据"的概念.但从上层角度来说.答案："是的".

面试题：一个java应用程序至少有几个线程？

有两个线程，一个主线程负责main方法执行。

一个垃圾回收器线程，负责了回收垃圾，

多线程好处：

1. 解决了一个进程能同时执行多个任务的问题

2. 提高了资源的利用率，不是效率，例子：迅雷下载东西

多线程弊端：

增加cpu负担。

降低一个进程中线程的执行概率

引发线程安全问题

出现死锁现象

实现线程 有两种方式：继承Thread类 和实现接口Runnable

第一种：实现接口Runnable和匿名线程

public static void main(String[] args) {

匿名线程写法一

Thread t= new Thread(new A());

t.start();

new Thread(new Runnable() {

public void run() {

Thread t2=Thread.currentThread();

t2.setName("匿名线程runnable");

for (int i = 0; i < 50; i++) {

System.out.println("我是:"+t2.getName()+" 运行了："+i);

}

}

}).start();

匿名线程写法二

Thread t3= new Thread(new Runnable() {

public void run() {

Thread t3=Thread.currentThread();

t3.setName("虚拟线程T3");

for (int i = 0; i < 50; i++) {

System.out.println("我是:"+t3.getName()+" 运行了："+i);

}

}

});

t3.start();

第二种继承Thread

new Thread() {// 匿名线程一

public void run() {

Thread t5=Thread.currentThread();

t5.setName("匿名线程Thread");

for (int i = 0; i < 50; i++) {

System.out.println("我是:"+t5.getName()+" 运行了："+i);

}

}

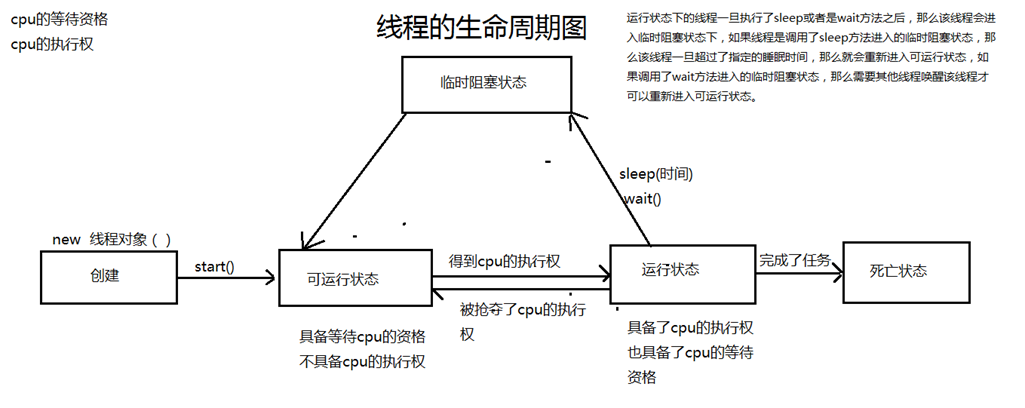
}.start();

注意:两个线程想要使用共同一个类中run线程需要：创建A类继承thread类

A a =new A();//此类继承了Thread类，实现了run方法

Thread t= new Thread(a) 把a对象放入Thread参数中，

Thread t1= new Tread(a)



创建：新创建了一个线程对象。

可运行：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取cpu的执行权。

运行：就绪状态的线程获取了CPU执行权，执行程序代码。

临时阻塞: 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

死亡：线程执行完它的任务时。

构造方法 还有ThreadGroup类的

Thread(Runnable target, String name) 分配新的 Thread 对象。Runnable参数，线程名字，注意可以是接口Runnable或继承了Thread的类因为它也是实现了Runnable接口的

A a = new A();//A类是必须继承了Thread类或者Runnable接口

Thread t = new Thread(a,"线程一");

普通方法

1）getName() 返回线程的名字

2）setName(String name) 设置线程对象名

3）static sleep() 线程睡眠指定的毫秒数。

注意Main使用此方法永远是主线程，

如要指定的线程使用此方法要在它的Run方法中使用，哪里才是另外线程执行的地方

**NObject b =new NObject("狗娃");**

**//b.sleep(3000);//请注意因为这个方法是静态的，谁调用它，谁睡眠，这里是主线程调用的，**

**b.setName("铁蛋");//设置线程名字**

**b.start();**

**注意 b.sleep(3000);//这里开启了线程，但是这里都是主线程执行地方，不能放这里，此方法要放到执行代码的地方，也就是Run方法中**

**放在不是主线程都要进行捕获操作不能抛出，因为子类异常要比父类相等或小：父类run方法没有异常、子类进行抛出就会出现异常**

4）static currentThread() 返回CPU正在执行的线程的对象：跟sleep一样，那个线程调用它，谁就是这个线程

**Thread a=Thread.currentThread();//获得当前线程，其实也是主线程获取对象，进行非静态方法调用**

**System.out.println(a.getName());//打印main，它是main中执行的**

**public void run() {**

**Thread a=Thread.currentThread();//获得当前线程**

**System.out.println("this:"+this);//打印：this:Thread[狗娃,5,main]**

**System.out.println("当前对象："+a);//打印：当前对象：Thread[狗娃,5,main]**

**}**

5）getPriority() 返回当前线程对象的优先级 默认线程的优先级是5

6）setPriority(int newPriority) 设置线程的优先级 虽然设置了线程的优先级，但是具体的实现取决于底层的操作系统的实现（最大的优先级是10 ，最小的1 ， 默认是5）。

**NObject b= new NObject("狗娃");**

**b.setPriority(10);//设置线程优先级为10**

**Thread.currentThread().setPriority(1);//设置主线程为1**

**System.out.println("自定义线程的优先级："+b.getPriority());**

**System.out.println("主线程优先级："+Thread.currentThread().getPriority());//priority()不是静态的方法，只能获得当前对象，再进行调用**

**b.start();**

7）void join(); 一个线程如果执行join语句，那么就有新的线程加入，执行该语句的线程必须要让步给新加入的线程先完成任务，然后才能继续执行。

join(long millis)另一个线程终止的时间最长为 millis 毫秒。

join(long millis, int nanos) 另一个线程该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。

**public void run() {**

**System.out.println("妈妈洗菜");**

**System.out.println("妈妈切菜");**

**System.out.println("妈妈准备炒菜，发现没有酱油了..");**

**//叫儿子去打酱油**

**Son s= new Son();**

**s.start();**

**try {**

**s.join(); //让给另一个线程执行完后再执行下面的代码。**

**} catch (InterruptedException e) {**

**e.printStackTrace();**

**}**

**System.out.println("妈妈继续炒菜");**

**System.out.println("全家一起吃饭..");**

**}**

#### 线程安全问题

同步锁是谁:

对于非static方法,同步锁就是this.

对于static方法,我们使用当前方法所在类的字节码对象(Apple2.class).

java线程同步机制的方式：

方式一：同步代码块

同步代码块的格式：

synchronized(锁对象){ //要共享的，比如静态使用的同一个，简单：“锁”OK

需要被同步的代码...

}

同步代码块要注意事项：

1. 任意的一个对象都可以做为锁对象。

2. 在同步代码块中调用了sleep方法并不是释放锁对象的。

3. 只有真正存在线程安全问题的时候才使用同步代码块，否则会降低效率的。

4. 多线程操作的锁 对象必须 是唯一共享 的。否则无效。

方式二：同步方法：就是在方法的返回值前面加：synchronized关键字

出现线程安全问题的根本原因：

1. 存在两个或者两个以上 的线程对象,而且线程之间共享着一个资源。

2. 有多个语句操作了共享资源。

死锁现象

死锁现象出现 的根本原因：

1. 存在两个或者两个以上的线程。

2. 存在两个或者两个以上的共享资源。

死锁现象的解决方案： 没有方案。只能尽量避免发生而已。

线程通讯 请看Object方法

一个线程完成了自己的任务时，要通知另外一个线程去完成另外一个任务.

**Object两个方法：**

**wait():** 等待 如果线程执行了wait方法，那么该线程会进入等待的状态，等待状态下的线程必须要被其他线程调用notify方法才能唤醒。

**notify()：** 唤醒 唤醒线程池等待线程其中的一个。

notifyAll() : 唤醒线程池所有等待 线程。

#### 守护线程

守护线程（后台线程）:在一个进程中如果只剩下了守护线程，那么守护线程也会死亡。

setDaemon(boolean on)将该线程标记为守护线程或用户线程。默认不是守护线程

isDaemon()测试该线程是否为守护线程。

作用：如希望一个线程结束另一个线程也跟着结束就可以设置成守护线程

Demo7 d = new Demo7("后台线程");

d.setDaemon(true); //设置线程是否为守护线程，true为守护线程， false为非守护线程。

System.out.println("是守护线程吗？"+ d.isDaemon()); //判断线程是否为守护线程。

d.start();

### String

new String(“abc”)创建了几个对象？

两个对象，一个对象是位于字符串常量池中，一个对象是位于堆内存中

一些字符串的内存对比情况

String str1=”hello”;

String str2=”hello”;

String str3=new String(“hello”);

String str4=new String(“hello”);

System.out.println(str1+str2);//true 在字符常量池中寻找hello内存地址一致

System.out.println(str3+str2);//false 内存不一样

System.out.println(str3+str4);//false进行内存对比在堆内存开辟空间，是不一致的

System.out.println(str3.equals(str4));//true 这是因为String重写了Object的equals方法，先转字符char[]数组进行一个一个字符进行对比，比较的是内容

Equals比较的时候建议常量开头调用equals，可以避免空指针异常：”中国”.equals(star)

构造方法：1个int[],6个byte[]，2个char[],2个stringBuffer

Int[]构造方法

1.String(int[] codePoints, int offset, int count)分配一个新的 String，它包含 Unicode 代码点数组参数一个子数组的字符。

int[] buf2 = {65,66,67};

//传入int类型数组buf2，从索引第0个开始，3个索引停止

str = new String(buf2,0,3);//打印：ABC

byte[]构造方法

1、String(byte[] bytes)通过使用平台的默认字符集解码指定的 byte 数组，构造一个新的 String。

//windows默认字符集是GBk，从字符集：97代表a，98代表b，99代表c

byte[] buf={97,98,99}

String str=new String(buf);

System.out.println(“字符串内容：”+str);//输出内容：abc

2.String(byte[] bytes, Charset charset) 通过使用指定的 charset 解码指定的 byte 数组，构造一个新的 String。

使用方法：把字节数组转为字符串用的，第一个参数是字节数组，第二个参数是字符编码。

byte[] bytes = new byte[1024];

new String(bytes,"UTF-8");

意思是把bytes数字按"UTF-8"的编码方式转成字符串。

3. String(byte[] bytes, int offset, int length)获取byte数组中指定的值

byte[] b = {97,98,99};

//传入byte类型数组b，从索引第0个开始，只要2个索引停止

String n =new String(b,0,2);

System.out.println(n); //打印：ab / 打印：ag

经常使用的字节流读取read的例子

FileInputStream f= new FileInputStream("D:/a.txt");

byte[] b= new byte[1024];

int i=0;

while((i=f.read(b)) !=-1) {

String e=new String(b,0,i);

System.out.println(e);

}

4. String(byte[] bytes, int offset, int length, Charset charset)

获取byte数组中指定的值，指定编码集

此方法很少用，跟第6个构造方法应该一样的，这里应该让你传入java.nio.charset 类

Charset传入，码表

5．String(byte[] bytes, int offset, int length, String charsetName)

通过使用指定的字符集解码指定的 byte 子数组，构造一个新的 String。

读取的文件如果是utf-8编码集，你也要指定它为utf-8进行读出，现工具大部分都是默认gbk的

例子

FileInputStream f= new FileInputStream("D:/a.txt");

byte[] b= new byte[1024];

int i=0;

while((i=f.read(b)) !=-1) {

//传入byte类型数组b，从索引第0个开始，读完全部，指定utf-8进行读出

String e=new String(b,0,I,”utf-8”);

System.out.println(e);

char[]构造方法

1. String(char[] value)

char[] arr = {'明','天','是','圣','诞'};

//使用字符数组构建一个字符串

str = new String(arr); //打印：明天是圣诞

2. String(char[] value, int offset, int count) 指定char数组开始和结束位置

char[] arr = {'明','天','是','圣','诞'};

str = new String(arr,3,2); //打印：圣诞

普通方法

获取方法

注意：length()：String是方法，数组的length是成员变量属性没有大括号的

1）Int length() 返回此字符串的长度。

String s= "abc，中国的人";

System.out.println(s.length());//打印：8

2）int indexOf(String str)查找字符串第一次出现的索引值。找不到返回 -1

用法：

String s= "abc,中国的人,cc中国";

int s1=s.indexOf("中国");打印：4

int s1=s.indexOf("国");打印：5

int s1=s.indexOf("国中");//打印：-1

System.out.println(s1);

3）int lastIndexOf(String str)查找字符串最后一次出现的索引值。找不到返回 -1

用法：和2的方法一样，只不过一个找第一次，一个最后一次

4）char charAt(int index) 返回指定索引处的 char 值。

String s= "abc,中国的人";

char s1=s.charAt(5);

System.out.println(s1);//打印：国

判断方法

1）boolean endsWith(String str) 是否以指定字符结束

用法：最常用来判断文件后缀名

String str = "Demo.java";

System.out.println(str.endsWith("va"));打印：true

System.out.println(str.endsWith("java"));打印：true

System.out.println(str.endsWith("ja"));打印：false

2）boolean isEmpty()是否长度为0 ，判断是否字符串是否为空，如：“”

用法：注意空指针异常

String s= "d.java";

System.out.println(s.isEmpty());打印：false

s=””;

System.out.println(s.isEmpty());打印：true

s=null;

System.out.println(s.isEmpty());打印：空指针异常

3）boolean contains(CharSequences) 是否包含有指定内容 应用：搜索

用法：

String s= "abc,中国的人,cc中国";

System.out.println(s.contains(“国人”));打印：false

System.out.println(s.contains(“中国”));打印：true

4）boolean equals(Object anObject) 判断内容是否相等

用法：注意：以常量为开头进行操作就不会出现空指针异常

String s= "abc,中国的人,cc中国";

System.out.println("你好".equals(s));打印：false，常量"你好"开头

5）boolean equalsIgnoreCase(String anotherString) 忽略大小写是否相等

转换方法

记住：字节数组与字符数组、字符串他们三者之间是可以互相转换的

1）char[] toCharArray()将字符串转换为字符数组

String s= "abc";

char[] a= s.toCharArray();

System.out.println(Arrays.toString(a));打印：[a, b, c]

2）byte[] getBytes(); 将字符串转换为字节数组

String s= "abc";

byte[] a= s.getBytes();

System.out.println("字节数组："+Arrays.toString(a));打印：字符数组：[97, 98, 99]

3）String(char[] value) 将字符数组转换为字符串

char[] s2={'a','b','c'};

String i=new String(s2);

System.out.println(i);

4）String toUpperCase() 转大写，String toLowerCase() 转小写

String s= "abc中国";

String a=s.toUpperCase();//打印：ABC中国

切割|替换|截取|去除空格

替换内容：String replace(char oldChar, char newChar)

他有1个重载方法：CharSequence target此类是接口String实现了它，可以说是String

String s= "今天晚上不考试";

String s1=s.replace("不", "要");//打印：今天晚上要考试

String s2=s.replace("不", "要好好");//打印：今天晚上要好好考试

System.out.println(s2);

切割：String[] split(String regex)

String s= "大家-下-午-好";

String[] a=s.split("-");

System.out.println(Arrays.toString(a));打印：[大家, 下, 午, 好]

String[] b=s.split(",");打印：全部内容，因为它是没有逗号的

截取：String substring(int beginIndex)

String substring(int beginIndex, int endIndex) 从第几个索引（包含）开始截取内容，到第几索引结束（不包含）：包头不包尾

String s= "广州传智播客";

String a=s.substring(2);//打印：传智播客

String a=s.substring(2,4);//打印：传智

System.out.println(a);

去除String trim() 去除字符串首尾的空格

String s= " 大家 都非常努力 ";

String a=s.trim();

System.out.println(a);//打印：大家 都非常努力

#### 例子

去除字符串两边空格的函数。模拟trim的方法。

说明：先进行前和后的空字符串判断确定索引值，使用截取内容方法substring

public static void main(String[] args) {

String str =" 传智 播客 ";

public static String myTrim(String str){

//先转换成字符 数组

char[] arr = str.toCharArray();

//定义两个 变量记录开始与结束 的索引值

int startIndex = 0 ;

int endIndex = arr.length -1;

//确定开始 的索引值

while(true){

if(arr[startIndex]==' '){

startIndex++;

}else{

break;

}

}

//确定结束 的索引值：

while(true){

if(arr[endIndex]==' '){

endIndex--;

}else{

break;

}

}

//截取子串返回

return str.substring(startIndex,endIndex+1);

获取上传文件名

String str = "D:\\20120512\\day12\\Demo1.java";

public static void getFileName(String path){

第一种

int index = path.lastIndexOf("\\") //先获得最后一个\\索引

String fileName = path.substring(index+1);//再进行开始第几个截取内容

System.out.println("文件名："+ fileName); //打印：Demo1.java

第二种切割String成数组

String[]arr= path.split (“\\\\”);//此方法有正则表达重载所以要这样写不让抱异常

System.out.println("文件名："+arr[arr.length-1]);//打印：Demo1.java

将字符串对象中存储的字符反序。

新中国好 -----> 好国中新

public static String reverse(String str){

char[] arr = str.toCharArray();

for(int a=0,b=arr.length-1;a<b; a++,b--){

char temp=arr[a];

arr[a] = arr[b];

arr[b] = temp;

}

//使用字符数组构建一个字符串。

return new String(arr);

}

内容java在字符串中出现的次数

public static void getCount(String str,String target){

int count = 0 ; //用于记录出现的次数

int fromIndex = 0; // 记录从那个索引值开始寻找目标子串

while((fromIndex = str.indexOf(target, fromIndex))!=-1){

//如果indexof方法返回 的不是-1，那么就是已经找到了目标 元素。

count++;

fromIndex = fromIndex+target.length();

}

System.out.println("出现的次数："+ count);

}

### Stringbuffer

线程安全的缓冲流字符串

为什么要用Stringbuffer？

字符串特点：

1、字符串是常量；它们的值在创建之后不能更改.

2、字符串的内容一旦发生了变化，那么马上会创建一个新的对象。

注意： 字符串的内容不适宜频繁修改，因为一旦修改马上就会创建一个新的对象。

如果需要频繁修改字符串的内容，建议使用字符串缓冲类（StringBuffer）。

StringBuffer 其实就是一个存储字符的容器。

4个构造方法，大体和String差不多

1） StringBuffer() 构造一个其中不带字符的字符串缓冲区，其初始容量为 16 个字符。

2） StringBuffer(CharSequence seq) 构造一个字符串缓冲区，它包含与指定的 CharSequence 相同的字符。CharSequence是一个接口，实现类String

3） StringBuffer(int capacity) 构造一个不带字符，但具有指定初始容量的字符串缓冲区。

4） StringBuffer(String str)构造一个字符串缓冲区，并将其内容初始化为指定的字符串内容。

普通方法

插入

1)append(Object b) 可以添加任意类型的数据到容器中

StringBuffer b= new StringBuffer("nnn");

b.append("abc");//打印：nnnabc

b.append(true);//打印：nnnabctrue

b.append(3.14f);//打印：nnnabctrue3.14

System.out.println(b);

2)insert(offset, str) 在字符串中插入数据

StringBuffer b= new StringBuffer("abc");

b.insert(2, "小麦");// 插入打印：ab小麦c

删除

1)delete(start,atr);指定开始和结束位置

StringBuffer b= new StringBuffer("ab小麦c");

b.delete(2, 4);

System.out.println(b);//打印abc

2) deleteCharAt(index);//根据指定的索引值删除一个字符

b.deleteCharAt(1);

System.out.println(b);//打印ac

修改

1）replace(int start, int end, String str) 根据指定的开始与结束索引值替代成指定的内容。

StringBuffer b= new StringBuffer("ab小麦c");

b.replace(2, 4, "明明");//打印ab明明c

2）reverse() 翻转字符串缓冲类的内容。 abc--->cba

StringBuffer b= new StringBuffer("ab小麦c");

System.out.println(b);// 打印c麦小ba

3）setCharAt(int index, char ch) 把指定索引值的字符替换指定的字符。

StringBuffer b= new StringBuffer("ab小麦c");

b.setCharAt(1, '小');

System.out.println(b);

4）substring(int start, int end) 根据指定的索引值截取子串。

//创建无参的构造方法字符串缓冲类

StringBuffer b= new StringBuffer("ab小麦c");

//它有返回值的要接收一下

String pp=b.substring(2, 4);//打印：小麦

System.out.println(pp);

5）ensureCapacity(int minimumCapacity) 指定StringBuffer内部的字符数组容量

一般都会在构造方法写上容量，构造方法有，一般不用

查看

1）indexOf(String str, int fromIndex) 查找指定的字符串第一次出现的索引值,并且指定开始查找的位置。

StringBuffer b= new StringBuffer("ab小麦c");

//指定字符串查找索引值

int i=b.indexOf("小麦");//打印 2

//从指定的开始地方找索引值

int i2=b.indexOf("c",2);//打印 2

System.out.println(i2);

2）lastIndexOf(String str)查找最后一个指定的内容返回索引值

StringBuffer b= new StringBuffer();//默认就16字符

b.append("abab");

int c=b.indexOf("ab");

int c1=b.lastIndexOf("ab");

System.out.println(c+":"+c1);//打印：0:2

3）capacity() 查看当前字符数组的容量（长度），不是内容中的长度默认有16字符容量。你在构造方法中添加了会把16也添加进去

StringBuffer b= new StringBuffer("ab");//默认就16，字符此处又增加了两个字符

int i=b.capacity();

System.out.println(i);//打印 18

还有一种情况

StringBuffer b= new StringBuffer();//默认就16字符，超过了16个字符它会再创建16个字符

b.append("ab");

int i=b.capacity();

System.out.println(i);//打印 16,没有超过16个字符长度，

4）length() 存储内容的长度（字符的个数）

StringBuffer b= new StringBuffer();

b.append("ab");

int i=b.length();charAt(int index)

5）toString() 把字符串缓冲类的内容转成字符串返回。把字符串缓冲类的内容转成字符串

作用：有些参数是要String的，不能传入StringBuffer，这样进行转换快

### StringBuilder

StringBuilder和Stringbuffer全部一样的：内容请看Stringbuffer，

区别：

StringBuilder处理数据速度快，但是线程是不安全的，操作效率高，jdk1.5的时候出现的

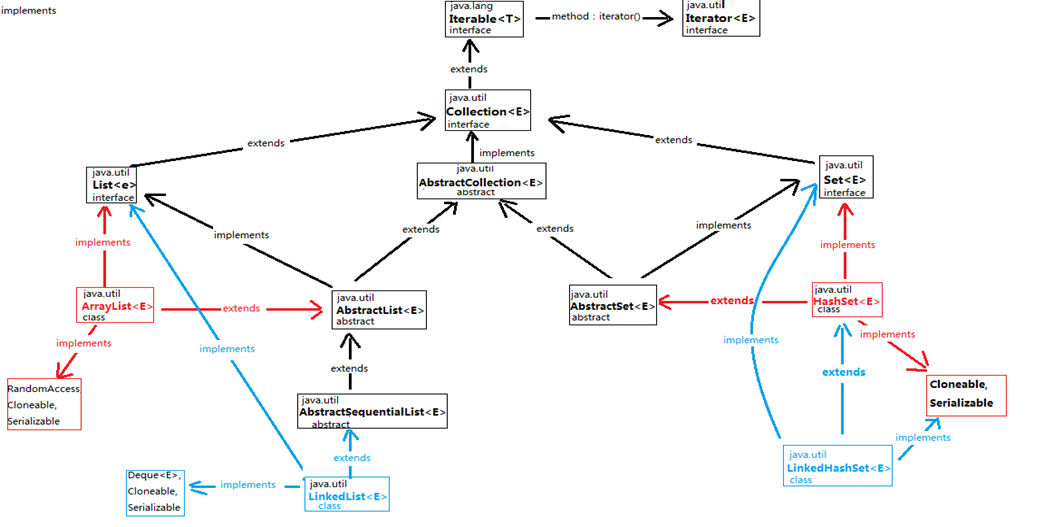
Stringbuffer 线程是安全的的；操作效率低，jdk1.0出现的

老师推荐使用线程不安全的，因为操作效率高，还有字符串现实中很少用到两个线程的，所有sun公司才在1.5才有的

## java.util

### 集合

Collection 它是一个接口类 Set 和list都继承了此接口



#### List

List：①ArrayList②LinkedList③Vector（了解）

如果是实现了List接口的集合类，该集合类具备的特点：有序，可重复。

有序： 集合的有序不是指自然顺序，而是指添加进去的顺序与元素出来的顺序是一致的。

添加

List接口中特有的方法具备的特点： 操作的方法都存在索引值。

只有List接口下面的集合类才具备索引值。其他接口下面的集合类都没有索引值。

1）add(int index, E element) 指定索引值，添加元素

list.add(1, "赵本山"); // 把元素添加到集合中的指定索引值位置上。

2）addAll(int index, Collection<? extends E> c)

list.addAll(2,list2); //把list2的元素添加到list集合指定索引值的位置上

获取

List独有方法，当然还有它父类collection方法

3）get(int index)

System.out.println("get方法获取元素："+list.get(1)); //根据索引值获取集合中的元素

使用get方法遍历集合的元素：

for (int i = 0; i < list.size() ; i++) {

System.out.print(list.get(i)+",");

}

4）int indexOf(Object o) 找出指定元素第一次出现在集合中 的索引值

System.out.println("找出指定元素第一次出现在集合中 的索引值："+ list.indexOf("本山"));

5）lastIndexOf(Object o) 找指定的元素最后一次出现在集合中的索引值

System.out.println("找指定的元素最后一次出现在集合中的索引值"+list.lastIndexOf("狗娃"));

6）subList(int fromIndex, int toIndex) 指定开始与结束的索引值截取集合中的元素。

List list= new ArrayList();

list.add("狗娃");

list.add("狗剩");

list.add("铁蛋"); //把元素添加到集合的末尾处。

list.add("狗娃");

List subList = list.subList(1, 3); //指定开始与结束的索引值截取集合中的元素。

System.out.println("子集合的元素是："+ subList);打印：子集合的元素是：[狗剩, 铁蛋]

修改

List独有方法，当然还有它父类collection方法

7）set(int index, E element) 使用指定的元素替换指定索引值位置的元素。

List list= new ArrayList();

list.add("狗娃");

list.set(0, "赵本山"); //使用指定的元素替换指定索引值位置的元素。

System.out.println("集合的元素："+list);打印：集合的元素：[赵本山]

迭代

listIterator(),它的父类是Iterator，Iterator有3个方法next()\hasNext()\remove()

8）listIterator()获得listIterator接口类

ListIterator it = list.listIterator();//返回的是一个List接口中特有的迭代器

listIterator 独有方法

1、hasPrevious() 判断是否存在上一个元素。

System.out.println("有上一个元素吗？"+ it.hasPrevious());

2、previous() 当前指针先向上移动一个单位，然后再取出当前指针指向的元素。

和next()区别：

next(); 先取出当前指针指向的元素，然后指针向下移动一个单位。

Previous（）：是先向上移动，再取出

Next：是先取出，再向下移动

死循环：

while(it.hasNext()){

it.next();

System.out.println("元素："+ it.previous());

倒着取出数据：

List list = new ArrayList();

list.add("狗娃");

list.add("狗剩");

list.add("铁蛋");

while(it.hasNext()){

it.next();//指针已经到最后了

}

while(it.hasPrevious()){//判断有没有上一个元素

System.out.println("元素："+ it.previous());//这里输出的就是倒着输出的

}

listIterator 独有方法3、add(E e) 把当前元素插入到当前指针指向的位置上。

List list = new ArrayList();

list.add("狗娃");

list.add("狗剩");

ListIterator it = list.listIterator();

it.next();

it.add("张三");

System.out.println("集合的元素："+ list);打印： 集合的元素：[狗娃, 张三, 狗剩]

listIterator 独有方法4、set(E e) 替换迭代器最后一次返回的元素。

List list= new ArrayList();

list.add("狗娃");

list.add("狗剩");

ListIterator g=list.listIterator();

g.next();

g.set("张三");

System.out.println("集合的元素："+list);打印：集合的元素：[张三, 狗剩]

ArrayList类：特点: 查询速度快，增删慢。

使用ArrayList无参的构造函数创建一个对象时， 默认的容量是多少? 如果长度不够使用时又自增增长多少？

答： ArrayList底层是维护了一个Object数组实现 的，使用无参构造函数时，Object数组默认的容量是10，当长度不够时，自动增长0.5倍。怪不得可以存所有数据

原理：

ArrayList 底层是维护了一个Object数组实现 的，查询快是因为：元素以元素之间的内存地址是连续的：0\*91—0\*92…..

什么时候使用ArrayList: 如果目前的数据是查询比较多，增删比较少的时候，那么就使用ArrayList存储这批数据。

为什么增删操作慢？

此类因为维护的是Object数组，增加的时候：第一：都会先检查容量够不够用，第二：如果不够用会把旧的数组进行重新拷贝新的数组，并增加1.5倍容量，试问如果大数据这样操作会快？ :10万数组就需要创建新的10万容量再进行拷贝数据

源码解析

如：本身数组容量是7，你要增加第8个容量，源码中它就需要使用Arrays.copyOf（old，new）方法拷贝旧的数组创建相同的容量数据，并增加1.5倍的容量，每次增加1个都要进行检查容量值，删除remove方法也有使用System.arraycopy方法，删除后要把后面数据一个一个向删除的索引移动一格。

构造方法

1）ArrayList()

构造一个初始容量为 10 的空列表。

2）ArrayList(Collection<? extends E> c)

构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。

3）ArrayList(int initialCapacity)

构造一个具有指定初始容量的空列表。

此类特有方法都不用学，只有两个不常用的和克隆一般不用：

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

此方法意思是默认是10只存了3个，其它7个通过此方法删除节省

ensureCapacity(int minCapacity)

构造方法3一样效果，一般都在构造方法中使用

Vector类

Vector(了解即可) 底层也是维护了一个Object的数组实现的，实现与ArrayList是一样的，但是Vector是线程安全的，操作效率低。

笔试题: 说出ArrayLsit与Vector的区别?

相同点： ArrayList与Vector底层都是使用了Object数组实现的。

不同点：

1. ArrayList是线程不同步的，操作效率高。

Vector是线程同步的，操作效率低。

2. ArrayList是JDK1.2出现，Vector是jdk1.0的时候出现的。

LinkedList类：特点: 查询慢，增删效率高。刚好和arrayList相反

此类底层是使用了链表数据结构实现的。数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合，老师说是大学本科有专门的课， 自己再去学对java也没什么好处因为都是封装起来的，对学c语言就一定要学；

一个元素分两块：一个本身，另一个记住下一个元素的内存地址，一环扣着一环，

元素和元素内存地址不连续，它就只能挨个挨个遍历完，所以查询速度慢

为什么增删快？

增加的时候：因为维护着下一个元素内存地址，只要让这个内存地址指向到新的内存地址，让新增加元素维护的内存地址指向另一个即可，

删除的时候：只需要将内存地址不指向要删除的元素即可，被删除的元素一旦没有变量指向就成垃圾对象，就没有人使用它了，这时候也不用把后面的元素移到它的位置，所以删除快

构造方法

LinkedList()构造一个空列表。

LinkedList(Collection<? extends E> c) 构造一个包含指定 collection 中的元素的列表，这些元素按其 collection 的迭代器返回的顺序排列。

特有常用方法

主要学习list方法，这个了解即可因为很多功能是重复的，

1）addFirst(E e) 把元素添加到集合的首位置上。

list.addFirst("狗娃");

2）addLast(E e) 把元素添加到集合的末尾处。

3）getFirst()

System.out.println("获取集合中首位置的元素:"+list.getFirst());

4）getLast()

System.out.println("获取集合中末尾的元素："+ list.getLast());

5）removeFirst()删除集合中的首位置元素，还会给你返回删除的数值

System.out.println("删除集合中的首位置元素并返回："+ list.removeFirst());

6）removeLast()删除集合中的末尾位置元素，还会给你返回删除的数值

System.out.println("删除集合中的末尾素并返回："+ list.removeLast());

7) descendingIterator() 返回逆序的迭代器对象

LinkedList l = new LinkedList();

l.add("张三");

l.add("李四");

l.add("王五");

Iterator r=l.descendingIterator();//获得逆序的迭代器

while(r.hasNext()) {

System.out.print(r.next());//打印：王五李四张三

下面4个方法：sun公司好心是让你模拟数据结构的

list.push("狗娃"); //将该元素插入此集合的开头处。

pop()移除并返回集合中的第一个元素

System.out.println("删除集合的首元素："+list.pop());

1：堆栈 （1.6） : 主要是用于实现堆栈数据结构的存储方式。

先进后出

push()

pop()

特点：先进后出，后进先出

offer（”狗剩”） 把元素添加到集合的末尾处

poll() System.out.println("删除集合的首元素: "+list.poll());

2：队列（双端队列1.5）： 主要是为了让你们可以使用LinkedList模拟队列数据结构的存储方式。

先进先出

offer()

poll()

特点：先进先出，后进后出

使用LinkedList实现堆栈数据结构的存储方式与队列的数据结构存储方式。

#### Set

Set具备的特点： 无序，不可重复。①HashSet②ThreeSet

此接口没有独有方法，全部都是它的父类collection的方法15个

Set是没有get方法的，得用①迭代器②增强型for进行遍历

无序： 添加元素的顺序与元素出来的顺序是不一致的。

不可重复：指的是内存地址不一样（元素的哈希值）

HashSet 底层是使用了哈希表来支持的，特点：存取速度快

HashSet的实现原理：

第一：往HashSet添加元素的时候，HashSet会先调用元素的hashCode方法得到元素的哈希值

第二：然后通过元素的哈希值经过移位等运算，就可以算出该元素在哈希表中的存储位置。

如何添加同一个对象，但内存地址不一样的？重写hashCode和equals！

因为存储过程是先调用该类的HashCode算出位置，如有值：才调用该类equalse方法

情况1： 如果hashCode算出该类存储的位置没有任何元素存储，那么该元素可以直接存储到该位置上。

情况2： 如果算出该元素的存储位置目前已经存在有其他的元素了，那么又会调用该元素的equals方法与该位置的元素再比较一次（默认进行内存对比，你要重写），如果equals返回的是true，那么该元素与这个位置上的元素就视为重复元素，不允许添加，如果equals方法返回的是false，那么该元素运行添加。

哈希表：其中一个特点：桶式结构,比喻：一个萝卜一个坑，它是可以多个萝卜一个坑

HashCode默认情况下表示的是内存地址，String 类已经重写了Object的hashCode方法了。equals也是重写了的，比较的是内容

注意： 如果两个字符串的内容一致，那么返回的hashCode 码肯定也会一致的。

调用equals比较内容，所以也加不进去

String str1 = "hello";

String str2 = new String("hello");

System.out.println("两个是同一个对象吗？"+(str1==str2));打印false

System.out.println("str1的hashCode："+ str1.hashCode());str1的hashCode:99162322

//String重写了Object的hashCode的方法：把字符串转成字符数组，挨个挨个乘以31算成数字，返回int数字

System.out.println("str2的hashCode:"+ str2.hashCode());str2的hashCode:99162322

例子需求： 接受键盘录入用户名与密码，如果用户名与密码已经存在集合中，那么就是视为重复元素，不允许添加到HashSet中。

public class NObject {

String name;

String password;

public NObject(String name,String password) {

this.name=name;

this.password=password;

}

//重写hashCode，这个是Object的，set集合添加的时候每次都会进行调用

public int hashCode() {

//这里调用了String重写了OBjec的方法

return this.name.hashCode()+this.password.hashCode();

}

//当对比了hashCode是一致的时候会再来这个equals进行对比确认

public boolean equals(Object obj) {

NObject b=(NObject)obj;

//现在调用的是String重写了Object的方法，是对比内容

return this.name.equals(b.name)&&this.password.equals(b.password);

}

public static void main(String[] args) throws Exception {

Scanner a =new Scanner(System.in);

HashSet set = new HashSet();

while(true) {

System.out.println(" 请输入账号：");

String c=a.next();

System.out.println(" 请输入密码：");

String v=a.next();

NObject no=new NObject(c,v);

if(set.add(no)) {

System.out.println("注册成功");

System.out.println("当前用户："+set.size()+"个");

}else {

System.out.println("注册失败");

}

ThreeSet：底层是使用红黑树（二叉树）数据结构实现的：左小右大：特点：会对元素进行排序存储

threeSet：和HashSet存储对象区别：

HsahSet存储是通过HashCode、eqauls方法进行判断存储的

ThreeSet：是通过让对象实现Comparable接口或者实现Comparator接口，在这里写方法进行判断，返回值是int：比0小放左边，比0大放右边，等于0（相同值）是加不进去的

原理：

1. 往TreeSet添加元素的时候，如果元素本身具备了自然顺序的特性，那么就按照元素自然顺序的特性进行排序存储。

2. 往TreeSet添加元素的时候，如果元素本身不具备自然顺序的特性，那么该元素所属的类必须要实现Comparable接口，把元素的比较规则定义在compareTo(T o)方法上。没有实现会报异常

3. 如果比较元素的时候，compareTo方法返回 的是0，那么该元素就被视为重复元素，不允许添加.(注意：TreeSet与HashCode、equals方法是没有任何关系。)

4. 往TreeSet添加元素的时候, 如果元素本身没有具备自然顺序 的特性，而元素所属的类也没有实现Comparable接口，那么必须要在创建TreeSet的时候传入一个比较器。

5. 往TreeSet添加元素的时候，如果元素本身不具备自然顺序的特性，而元素所属的类已经实现了Comparable接口， 在创建TreeSet对象的时候也传入了比较器那么是以比较器的比较规则优先使用。

如何自定义定义比较器： 自定义一个类实现Comparator接口即可，把元素与元素之间的比较规则定义在compare方法内即可。

自定义比较器的格式 ：

class 类名 implements Comparator{

}

//自定义一个比较器

class MyComparator implements Comparator<Emp>{

public int compare(Emp o1, Emp o2) {

return o1.id-o2.id;//要么返回比0小，要么等于0，要么大于0

}

//创建一个比较器对象

MyComparator comparator = new MyComparator();

//创建TreeSet的时候传入比较器

TreeSet tree = new TreeSet(comparator);//构造方法中传入比较器

推荐使用：使用比较器(Comparator)。它有构造方法传此实现类

字符串的比较规则：

情况一： 对应位置有不同的字符出现， 就比较的就是对应位置不同的字符。

情况二：对应位置上的字符都一样，比较的就是字符串的长度。

此类用的不多，写的不多，还有一个叫自动调整节点，它会在大于3比较后都在后边，就会重新调整，把3个最小的值放到形成交叉

#### Map

Map ①HashMap②TreeMap

如果是实现了Map接口的集合类，具备的特点： 存储的数据都是以键值对的形式存在的，键不可重复，值可以重复。

嵌套类：Map.Entry<K,V>映射项（键-值对）遍历用：可以同时获得键值对的值

方法 Map学这些方法即可

添加

Map<String,String> s=new HashMap<String, String>();

s.put("汪峰", "章子怡");

s.put("文章", "马伊琍");

s.put("谢霆锋", "张柏芝");

s.put(null, null);//也能作为数据，打印{null=null}

1）put(K key, V value)

System.out.println(s.put("谢霆锋", "王菲"));

2）putAll(Map<? extends K,? extends V> m)

s.putAll(s1);//把s1的map元素添加到s的map集合中

删除

3） remove(Object key)

System.out.println(s.remove("谢霆锋"));//根据键删除指定的值，返回该键的值,没有该键就不会删除数据，也不会报错，返回null;

4） clear()清空集合中的所有数据，打印：集合的元素

s.clear();

获取

5）get(Object key) 根据指定的key获得值，返回获得值

System.out.println(s.get("文章"));//打印：马伊琍

6）size()获得存储的个数

System.out.println(s.size());

判断

1） containsKey(Object key) 判断map集合是否包含指定的键,返回boolean

System.out.println(s.containsKey("汪峰"));

2） containsValue(Object value) 判断map集合是否包含指定的值,返回boolean

System.out.println(s.containsValue("章子怡"));

3） isEmpty()判断map集合是否为空元素,返回boolean

System.out.println(s.isEmpty());

迭代

有3种方式：1获得key，再通过get获得值，2获得values，3两个都获得

1）keySet() 获得key值的数据，以set集合返回

使用keySet方法进行遍历 ，缺点：只是返回了所有的键，没有值，

Set<String> d=s.keySet();//获得key值的数据，以set集合返回

Iterator<String> c=d.iterator();

while(c.hasNext()) {

System.out.println("键："+c.next()+"值："+s.get(c.next()));//这样是不行的，取值两次，要变量接收

String key =c.next();//变量保存键值

System.out.println("键："+key+"值："+s.get(key));//通过键获得值

2）values():获得Collection集合，进行获得所有值

使用使用values()方法进行遍历,缺点：此方法只能返回所有的值，没有键

Collection<String> d2=s.values();

Iterator<String> c1=d2.iterator();

while(c1.hasNext()) {

System.out.println("值："+c1.next());

}

3）entrySet(),获得Entry类，它有key、value两个成员变量属性

原理：此方法是使用自定义类JavaBean：Entry类的成员变量key，value，再把对象存到Set中：Set<Entry> entrys=map.entrSet()，它是静态内部类：开发都建议Set<Map.Entry<String, String>>d3=s.entrySet();

Set<Entry<String, String>>d3=s.entrySet();//这样写也可以的

Iterator<Entry<String, String>> c2=d3.iterator();

while(c2.hasNext()) {

Entry<String, String> cc=c2.next();//获得对象类Entry

System.out.println("值3："+cc);

cc.getKey();//获得Entry成员变量属性 key

cc.getValue(); //获得Entry成员变量属性 value

}

}

HashMap

HashMap 底层也是基于哈希表实现的。

原理和HashSet几乎一样：先调用hashCode，进行对比，不一样再调用equals方法比较

HashMap的存储原理：

往HashMap添加元素的时候，首先会调用键的hashCode方法得到元素的哈希码值，然后经过运算就可以算出该

元素在哈希表中的存储位置。

情况1： 如果算出的位置目前没有任何元素存储，那么该元素可以直接添加到哈希表中。

情况2：如果算出 的位置目前已经存在其他的元素，那么还会调用该元素的equals方法与这个位置上的元素进行比较，如果equals方法返回 的是false，那么该元素允许被存储，如果equals方法返回的是true，那么该元素被视为重复元素，不允许存储。

存储对象类希望成员变量id是同一类，也要重写此对象类的hashCode方法和equals方法

class Person{//创建键key类

int id;

String name;

public Person(int id, String name) {

super();

this.id = id;

this.name = name;

}

public String toString() {//重写此类继承Object 的方法

return "[编号："+this.id+" 姓名："+ this.name+"]";

}

public int hashCode() {//重写此类继承Object 的方法

return this.id;

}

public boolean equals(Object obj) {//重写此类继承Object 的方法

Person p = (Person) obj;

return this.id== p.id;

}

}

public class Demo5 {

public static void main(String[] args) {

HashMap<Person, String> map = new HashMap<Person, String>();

map.put(new Person(110,"狗娃"), "001");

map.put(new Person(220,"狗剩"), "002");

map.put(new Person(330,"铁蛋"), "003");

map.put(new Person(110,"狗娃"), "007"); //如果出现了相同键，那么后添加的数据的值会取代之前的值。

System.out.println("集合的元素："+ map);

}

TreeMap

此类和treeSet几乎一样

TreeMap也是基于红黑树（二叉树）数据结构实现 的， 特点：会对元素的键进行排序存储。

TreeMap 要注意的事项：

1. 往TreeMap添加元素的时候，如果元素的键具备自然顺序，那么就会按照键的自然顺序特性进行排序存储。

2. 往TreeMap添加元素的时候，如果元素的键不具备自然顺序特性， 那么键所属的类必须要实现Comparable接口，把键的比较规则定义在CompareTo方法上。

3. 往TreeMap添加元素的时候，如果元素的键不具备自然顺序特性，而且键所属的类也没有实现Comparable接口，那么就必须在创建TreeMap对象的时候传入比较器。

class Emp {//implements Comparable<Emp>{ //第2种实现Comparable接口

String name;

int salary;

public Emp(String name, int salary) {

super();

this.name = name;

this.salary = salary;

}

public String toString() {

return "[姓名："+this.name+" 薪水："+ this.salary+"]";

}

public int compareTo(Emp o) {//第2种实现Comparable接口的方法

return this.salary - o.salary;//大于0右边，小于0左边，等于0不添加

}

}

//自定义一个比较器

class MyComparator implements Comparator<Emp>{//第3种方式创建比较器类

public int compare(Emp o1, Emp o2) {//实现此方法

return o1.salary - o2.salary;

}

}

public class Demo6 {

public static void main(String[] args) {

第2种方式，键类实现了接口，进行调用

/\* TreeMap<Character, Integer> tree = new TreeMap<Character, Integer>();

tree.put('c',10);

tree.put('b',2);

tree.put('a',5);

tree.put('h',12);

System.out.println(tree);\*/

//第3种方式传入比较器

//创建一个自定义比较器

MyComparator comparator = new MyComparator();

TreeMap<Emp, String> tree = new TreeMap<Emp, String>(comparator);

tree.put(new Emp("冰冰", 2000),"001");

tree.put(new Emp("家宝", 1000),"002");

tree.put(new Emp("习总", 3000),"003");

tree.put(new Emp("克强", 5000),"005");

tree.put(new Emp("财厚", 5000),"008");

System.out.println(tree);

Hashtable

Hashtable的原理是和hashMap一致的，但是它是线程安全的，效率低。

它是1jdk1.0版本出来的，hashMap是jdk1.2特性

#### 工具类

Collections集合工具类

对list集合进行排序

1）Sort（list）：传入自然元素集合

ArrayList<Integer> list =new ArrayList<Integer>();

list.add(1);

list.add(16);

list.add(3);

list.add(8);

//排序

Collections.sort(list);

2）Sort（list，comaprator）：传入list集合，和比较器（通过创建类实现comaprator接口，实现方法：0以下最小，等于0,0以上最大）

class Person{ //对象类

int age;

String name;

public Person(int age, String name) {

this.age = age;

this.name = name;

}

public String toString() {

return "姓名："+this.name+" 年龄："+ this.age;

}

}

class Ab implements Comparator<Person>{//实现比较器接口

public int compare(Person o1, Person o2) {//实现比较器的方法

return o1.age-o2.age;//按年龄进行比较升序

}

}

public class NObject {

public static void main(String[] args) throws Exception {

Ab ab= new Ab();

ArrayList<Person> list =new ArrayList<Person>();

list.add(new Person(11, "狗娃"));

list.add(new Person(13, "李三"));

list.add(new Person(16, "李四"));

list.add(new Person(12, "王五"));

list.add(new Person(12, "张六"));

Collections.sort(list, ab);

System.out.println(list);//输出按年龄最小开始打印

}

3）Int binarySearch(list,key);此方法是指自然元素，不是对象，查找所在的索引值，要先排序了，再使用此方法

对list进行二分查找

二分：先排序再查找，二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构

ArrayList<Integer> list1 =new ArrayList<Integer>();

list1.add(1);

list1.add(16);

list1.add(3);

list1.add(8);

//排序

Collections.sort(list1);

int v=Collections.binarySearch(list1,16);

System.out.println(v);//排序后是第4位，所以索引值是：3

对集合取最大值或最小值

max(Collection)参数：集合

max(Collection, comaprator) 参数：集合，比较器

System.out.println(Collections.max(list,ab));//打印：姓名：李四 年龄：16

min(Collection) 参数：集合

min(Collection, comaprator) 参数：集合，比较器此方法不用先进行排序

reverse(Collection)//要先进行了排后使用，不然你使用了也看不出什么效果

将不同步的集合变成同步集合

把集合变称安全

Set synchronizedSet(Set<T> s);

List li=(ArrayList<Person>) Collections.synchronizedList(list);

Map synchronizedMap(Map<K,V> m)

List synchronizedList(list<T> s);

### Properties

Properties -----extends--Hashtable<Object,Object>

Properties（配置文件类）: 主要用于生产配置文件与读取配置文件的信息。

Properties要注意的细节：

1. 如果配置文件的信息一旦使用了中文，那么在使用store方法生成配置文件的时候只能使用字符流解决，如果使用字节流生成配置文件的话，默认使用的是iso8859-1码表进行编码存储，这时候会出现乱码。

2. 如果Properties中的内容发生了变化，一定要重新使用Properties生成配置文件，否则配置文件信息不会发生变化。

//读取配置文件爱你的信息

public static void readProperties() throws IOException{

第一步 创建Properties对象

Properties properties = new Properties();

第二步加载配置文件信息到Properties中

properties.load(new FileReader("F:\\persons.properties"));

第三步 遍历他也有sun公司的Entry操作，map也是此方法

Set<Entry<Object, Object>> entrys = properties.entrySet();

for(Entry<Object, Object> entry :entrys){

System.out.println("键："+ entry.getKey() +" 值："+ entry.getValue());

}

第四步 把修改后的Properties再生成一个配置文件

properties.setProperty("狗娃", "007");

properties.store(new FileWriter("F:\\persons.properties"), "hehe");

}

//使用Properties生产配置文件。

//第一个参数是一个输出流对象，第二参数是使用一个字符串描述这个配置文件的信息。

properties.store(new FileOutputStream("F:\\persons.properties"), "haha");

需求： 使用properties实现本软件只能 运行三次，超过了三次之后就提示购买正版，退jvm.

public class Demo5 {

public static void main(String[] args) throws IOException {

File file = new File("F:\\count.properties");

if(!file.exists()){

//如果配置文件不存在，则创建该配置文件

file.createNewFile();

}

//创建Properties对象。

Properties properties = new Properties();

//把配置文件的信息加载到properties中

properties.load(new FileInputStream(file));

FileOutputStream fileOutputStream = new FileOutputStream(file);//次创建对象一定要在loda后创建，不然会导致文件清空，你使用了参数true进行追加了，就会导致配置文件，追加很多次，

int count = 0; //定义该变量是用于保存软件的运行次数的。

//读取配置文件的运行次数

String value = properties.getProperty("count");//在第一次进行取值的时候文件是空的，取到的值是null，没有也不会报错

if(value!=null){

count = Integer.parseInt(value);//转换成int类型

}

//判断使用的次数是否已经达到了三次，

if(count==3){

System.out.println("你已经超出了试用次数，请购买正版软件！！");

System.exit(0);

}

count++;

System.out.println("你已经使用了本软件第"+count+"次");

properties.setProperty("count",count+"");

//使用Properties生成一个配置文件

properties.store(fileOutputStream,"runtime");

}

### Date

它很多方法被Calendar类代替，但是它还是很重要要配合SimpleDateFormat类使用

最常见方法转换我们想要日期格式：new SimpleDateFormat(“yyyy-MM-dd”).format(Date)

构造方法

Date() 无参构造方法，是获取当前系统时间

Date(long date) 分配 Date 对象并初始化此对象， 1970 年 1 月 1 日 00:00:00开始计算，如毫秒值 long 5856554，把这些毫秒转换为时间

普通方法

没有过时的方法：

equals(Object obj) 比较两个日期的相等性。

before(Date when) 测试此日期是否在指定日期之前。

after(Date when) 测试此日期是否在指定日期之后。

hashCode()返回此对象的哈希码值。

setTime(long time) 参数放入long的数字，进行转换成毫秒值

toString()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)。

//按指定格式时间转字符串

Date d=new Date();

SimpleDateFormat s= new SimpleDateFormat();//默认的格式

s.applyPattern("yyyy-MM-dd HH:mm:ss");

String s1=s.format(d);

System.out.println(s1);打印：2018-02-06 09:23:15

//字符串转换时间

String b="2015-03-05 11:29:02";//注意全部要一致包括空格

SimpleDateFormat v= new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date date=v.parse(b);

System.out.println(date);// 打印:Thu Mar 05 11:29:02 CST 2015

### Calendar

Calendar【日历】

它（字段）成员变量属性非常多，几乎都是静态的

获取此对象实例对象：getInstance()

因太多不一一列举，只给常见的：

Calendar calendar = Calendar.getInstance(); //获取当前的系统时间。

System.out.println("年："+ calendar.get(Calendar.YEAR));

System.out.println("月："+ (calendar.get(Calendar.MONTH)+1));

System.out.println("日："+ calendar.get(Calendar.DATE));

System.out.println("时："+ calendar.get(Calendar.HOUR\_OF\_DAY));

System.out.println("分："+ calendar.get(Calendar.MINUTE));

System.out.println("秒："+ calendar.get(Calendar.SECOND));

判断是否闰年

public void isLeapYear(int years){

Calendar cal=Calendar.getInstance();

cal.set(years,Calendar.DECEMBER,31);//12月是固定有31天的

if(cal.get(Calendar.DAY\_OF\_YEAR)==366){

System.out.println(years+"年是闰年");

}else{

System.out.println(years+"年平年");

}

}

获取今天或者之后多少天的日期

Calendar calendar=Calendar.getInstance();

public void getTest(){

calendar.setTime(new Date());

/\*获取今天的日期\*/

System.out.println("今天的日期是："+calendar.get(Calendar.DAY\_OF\_MONTH));

/\*获取十天之后的日期\*/

calendar.clear();//避免继承当前系统的时间

calendar.set(Calendar.DAY\_OF\_MONTH, calendar.get(Calendar.DAY\_OF\_MONTH)+10);

System.out.println("十天之后的日期是："+calendar.get(Calendar.DAY\_OF\_MONTH));

}

### Math

Math 数学类， 主要是提供了很多的数学公式。

abs(double a) 获取绝对值

ceil(double a) 向上取整

floor(double a) 向下取整

round(float a) 四舍五入

random() 产生一个随机数. 大于等于 0.0 且小于 1.0 的伪随机 double 值

l例子

public static void main(String[] args) {

System.out.println("绝对值:"+Math.abs(-3));//打印：3

System.out.println("向上取整："+Math.ceil(3.14)); //打印：4.0

System.out.println("向下取整："+Math.floor(-3.14)); //打印：-4.0

System.out.println("四舍五入:"+Math.round(3.54)); //打印：4

System.out.println("随机数："+Math.random());//

### Random

两个构造方法

Random()创建一个新的随机数生成器。

Random(long seed) 使用单个 long 种子创建一个新的随机数生成器。

Random random = new Random();

int randomNum = random.nextInt(9); //产生 的 随机数就是0-8之间

System.out.println("随机数："+ randomNum+1);就变成1-9

\*/

char[] arr = {'中','国','传','a','Q','f','B'};

StringBuilder sb = new StringBuilder();//使用缓冲字符串容器

Random random = new Random();

//需要四个随机数，通过随机数获取字符数组中的字符，

for(int i = 0 ; i< 4 ; i++){

int index = random.nextInt(arr.length); //产生的 随机数必须是数组的索引值范围之内的。

sb.append(arr[index]);

}

System.out.println("验证码："+ sb);

## java.io

IO技术主要的作用是解决设备与设备之间的数据传输问题。

比如：“硬盘--->内存” 内存的数据---->硬盘上把键盘的数据------->内存中

IO技术的应用场景：

导出报表 ， 上传大头照 、 下载 、 解释xml文件 ...

数据保存到硬盘上，该数据就可以做到永久性的保存。 数据一般是以文件的形式保存到硬盘上

### File

File类: 用于描述一个文件或者文件夹的。

通过File对象我们可以读取文件或者文件夹的属性数据

构造方法

1)File(String pathname) 通过给定路径名字符串转换为抽象路径名来创建一个新 File 实例。

File file= new File("d:/b/a.txt");

System.out.println("文件存在吗？"+file.exists());//判断文件方法

2)File(File parent, String child) 父亲抽象路径名和 child孩子 路径名创建一个新 File 实例。

File f = new File("d://a"); //没有文件的路径

if(!f.isDirectory()) { //判断文件夹是否存在：!f.isDirectory()

f.mkdirs();

}

File file = new File(f,"t.text");

if(!file.exists()) {

file.createNewFile();

}

InputStream a= System.in; //获取键盘上的输入流

OutputStream o = new FileOutputStream(file);

while(true) {

int input = a.read(); //回车键把信息发出去

o.write(input);

}

File(String parent, String child)父亲路径名字符串和孩子路径名字符串创建一个新 File 实例。

此构造好处：可以传两个参数字符串进行拼接，不用那么死板，

File parent= new File("d//a","a.txt");//进行构建一个File

File(URI uri) 通过将给定的 file: URI 转换为一个抽象路径名来创建一个新的 File 实例。

静态字段 1和2返回值是一样的，但类型一个String一个char。3和4也是一样

1、separator与系统有关的默认名称分隔符，返回String打印：”\”

System.out.println("目录分隔符："+File.separator);

2、separatorChar 与系统有关的默认名称分隔符。返回char，打印：”\”

1和2方法：linux:只有正斜杠：/ windows：正斜杠：/ 和反斜杠 \反斜杠在字符串有特殊含义所以：\

3、pathSeparatorChar 与系统有关的路径分隔符打印：”;”

4、pathSeparator 与系统有关的路径分隔符打印：”;”

3和4的方法是路径分隔符，在Window上是";"，在Unix上是":"

方法使用

创建文件

1）boolean createNewFile() 在指定位置创建一个空文件，成功就返回true，如果已存在就不创建然后返回false有异常需要抛出

File file7 = new File("d:\\dd.txt");

System.out.println("创建成功了吗？"+file7.createNewFile());

2）boolean mkdir() 指定位置创建目录，只会创建最后一级目录，如上级目录不存在就抛异常。

File dir = new File("d:\\我是文件夹");

System.out.println("创建文件夹成功吗？"+dir.mkdir()); // mkdir 创建一个单级文件夹，

3）mkdirs() 在指定的路径创建所有没有的目录

System.out.println("创建多级文件夹,当然一个也可以："+ dir2.mkdirs());

4）renameTo(File dest) 如果目标文件与源文件是在同一个路径下，那么renameTo的作用是重命名， 如果目标文件与源文件不是在同一个路径下，那么renameTo的作用就是剪切，但是不能操作文件夹进行剪切。

File f = new File("d:\\a.txt");第一步创建被操作的文件File

File destFile = new File("d:\\haha.txt"); 第二步 创建需要剪切还是重命名的目标文件

System.out.println("重命名成功吗？"+f.renameTo(destFile)) ; 打印：重命名成功吗？true

删除

1）boolean delete() 删除文件或一个空文件夹，是文件夹且不为空，则不能删除。

System.out.println("删除成功了吗？"+dir.delete());//此方法可以删除文件，和空文件夹，不可以删除非空文件夹

2）deleteOnExit() jvm退出的时候删除：createTempFile("文件名", "文件后缀")在系统中创建的临时文件。

File f=File.createTempFile("nihao", "txt");//临时文件夹在：C:\Users\Administrator\AppData\Local\Temp\ nihao4829296196798920txt

Thread.sleep(10000);//让他延迟10秒关闭，看效果

f.deleteOnExit();//调用此方法会删除上面创建的临时文件

判断

1）exists() 文件或文件夹是否存在。

2）isFile() 是否是一个文件，如果不存在，则始终为false。

3）isDirectory() 是否是一个目录，如果不存在，则始终为false。

4）isHidden() 是否是一个隐藏的文件或隐藏的目录。

5）isAbsolute() 测试此抽象路径名是否为绝对路径名。

获取

1）String getName() 获取文件或文件夹的名称，不包含上级路径。

2）getPath() 返回绝对路径，可以是相对路径，但是目录要指定

3）getAbsolutePath() 获取文件的绝对路径，与文件是否存在没关系

4）length() 获取文件的大小（字节数），如果文件不存在则返回0，如果是文件夹也返回0。

5）getParent()抽象路径名父目录的路径名；如果此路径名没有指定父目录，则返回null。

6）Long lastModified() 获取最后一次被修改的时间。

File f= new File("d:\\a\\haha.txt");

System.out.println("文件名："+f.getName());文件名：haha.txt

System.out.println("获得全路径："+f.getPath());获得全路径：d:\a\haha.txt

System.out.println("获得全路径"+f.getAbsolutePath());获得全路径d:\a\haha.txt

System.out.println("抽象路径名父目录："+f.getParent());抽象路径名父目录：d:\a

System.out.println("最后一次被修改的时间:"+f.lastModified());最后一次被修改的时间:1528897875921

System.out.println(f.length());

查看

1)staic File[] listRoots() 列出所有的根目录（Window中就是所有系统的盘符）

File[] roots = File.listRoots(); //列出所有的根目录 C:\D:\E:\,就是你的盘符

for(File cc : roots){

System.out.println(cc);打印：C:\ D:\ E:\

}

2)list() 返回目录下的文件或者目录名，包含隐藏文件。对于文件这样操作会返回null。

list(FilenameFilter filter)返回指定当前目录中符合过滤条件的子文件或子目录。

FilenameFilter:现此接口的类的实例用于过滤文件名。是一个标识类，什么方法都没有的

File file = new File("d:\\");

String[] fileNames = file.list(); //把当前文件夹下面的所有文件名与文件夹名 存储到一个String类型的数组中返回。

for(String fileName : fileNames){

System.out.println(fileName);

}

3)listFiles() 返回目录下的文件或者目录对象（File类实例），包含隐藏文件。

listFiles(FilenameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录。

File[] files = file.listFiles(); // 把当前文件夹下面的所有文件与文件夹都使用了一个FIle对象类型描述，然后把这些File对象存储到一个FIle数组中返回

for(File fileItem : files){

System.out.println("文件名："+ fileItem.getName());

}

##### 例子

需求1：列出一个文件夹的子孙文件与目录。

public static void listFiles1(File dir){

File[] files = dir.listFiles(); //列出所有 的子文件

for(File file : files){

if(file.isFile()){

System.out.println("文件名："+file.getName());

}else if(file.isDirectory()){

System.out.println("文件夹："+file.getName());

listFiles1(file);

}

2，列出指定目录中所有的子孙文件与子孙目录名，要求名称前面要有相应数量的空格：

第一级前面有0个，第二级前面有1个，第三级前面有2个...，以此类推。

public static void listFiles2(File dir,String space){ //space 存储的是空格

File[] files = dir.listFiles(); //列出所有 的子文件

for(File file : files){

if(file.isFile()){

System.out.println(space+file.getName());

}else if(file.isDirectory()){

System.out.println(space+file.getName());

listFiles2(file," "+space);

}

3，列出指定目录中所有的子孙文件与子孙目录名，要求要是树状结构，效果如下所示：

|--src

| |--cn

| | |--itcast

| | | |--a\_helloworld

| | | | |--HelloWorld.java

| | | |--b\_for

| | | | |--ForTest.java

| | | |--c\_api

| | | | |--Student.java

|--bin

| |--cn

| | |--itcast

| | | |--i\_exception

| | | | |--ExceptionTest.class

| | | |--h\_linecount

| | | | |--LineCounter3.class

| | | | |--LineCounter2.class

| | | | |--LineCounter.class

|--lib

| |--commons-io.jar

public static void listFiles3(File dir,String space){ //space 存储的是空格

File[] files = dir.listFiles(); //列出所有 的子文件

for(File file : files){

if(file.isFile()){

System.out.println(space+file.getName());

}else if(file.isDirectory()){

System.out.println(space+file.getName());

listFiles3(file,"| "+space);

}

}

需求4: 删除一个非空的文件夹。

public static void deleteDir(File dir){ // bb

File[] files = dir.listFiles(); //列出了所有的子文件

for(File file : files){

if(file.isFile()){

file.delete();

}else if(file.isDirectory()){

deleteDir(file);

}

}

dir.delete();

作业： 剪切一个非空目录。

public static void main(String[] args) {

/\* File dir = new File("F:\\1208project\\day22");

listFiles3(dir,"|--");\*/

File dir = new File("F:\\aa");

deleteDir(dir);

}

### 输入流

#### InputStream

字节流: 字节流读取得都是文件中二进制数据，读取到二进制数据不会经过任何的处理。

public abstract class InputStream extends Object implements Closeable

FileInputStream 是InputStream的子类: public class FileInputStream extends InputStream

使用FileInputStream读取文件数据的步骤：

1. 找到目标文件,2.建立数据的输入通道,3. 读取文件中的数据,4. 关闭资源.

构造方法

1）FileInputStream(String name) 通过指定的路径进行构造输入流

FileInputStream io=new FileInputStream("d://a.txt");

2）FileInputStream(File file) 通过File类文件进行构造输入流对象

File file =new File("G:\\a.txt");

if(!file.exists()) {

file.createNewFile();

}

FileInputStream io=new FileInputStream(file);

3)FileInputStream(FileDescriptor fdObj) 通过使用文件描述符 fdObj 创建一个 FileInputStream，该文件描述符表示到文件系统中某个实际文件的现有连接。

常用方法

1） int read() 从此输入流中读取一个数据字节。

FileInputStream f= new FileInputStream("g:/a.txt");

char a=(char) f.read();//int类型强转成cha字符

System.out.println(a);//打印

2） int read(byte[] b) 把读到的全部写入byte数组中

File file =new File("G:\\a.txt");

if(!file.exists()) {

file.createNewFile();

}

FileInputStream io=new FileInputStream(file);

byte[] b= new byte[1024\*1024];

int i=0;//设置变量，接收a.text中内容长度

while((i=io.read(b))!= -1) {//变量i接收长度

//接收byte数组b，从索引0开始，如用b.length因为你构建byte[10]，但你内容只有4个，导致后面就是6个空字符串

System.out.println(new String(b,0,i));

3）int read(byte[] b, int off, int len) 从此输入流中将最多len个字节的数据读入一个 byte 数组中。

4）void close()关闭此文件输入流并释放与此流有关的所有系统资源。

BufferedInputStream 缓冲流

凡是缓冲流都不具备读写功能，构造方法都需要传入字节流或字符流对象

BufferedInputStream---extends--->FilterInputStream它是过滤器流，不是fileInputStream

原理

其实该类内部只不过是维护了一个8kb的字节数组而已。

private static int defaultBufferSize = 8192;

public BufferedInputStream(InputStream in) { //构造方法中放入此数据8192

this(in, defaultBufferSize);

}

构造方法

BufferedInputStream(InputStream in)

BufferedInputStream(InputStream in, int size)

使用BufferedInputStream的步骤:

1. 找到目标文件。

2. 建立数据 的输入通道

3. 建立缓冲 输入字节流流

4. 关闭资源

FileInputStream fis=new FileInputStream("g:/f.docx");

BufferedInputStream bis=new BufferedInputStream(fis);

int i=0;

long c= System.currentTimeMillis();//获取初始时间

while((i=bis.read()) !=-1) {

//进行强制类型转换为char

//System.out.print((char)i);//获取内容，一般字节都用来传图片二进制

}

long d= System.currentTimeMillis();//获取结束时间

System.out.println("读取时间："+""+(d-c));

//关闭的时候只需要关闭最外层的流就行了

bis.close();

经测试读取2m文件使用缓冲耗费56，没有使用6000多

#### Reader

字符流： 字符流会把读取到的二进制的数据进行对应的编码与解码工作。

字符流 = 字节流 + 编码(解码)

关系图

Reader(抽象类)-- InputStreamReader（转换流，主要编码作用）----- FileReader(字符流读写)

输入字符流：

----------| Reader 输入字符流的基类 抽象类

----------------| InputStreamReader编码与解码工作

---------------------| FileReader 读取文件的输入字符流。

字符流： 字符流读取的数据是以字符为单位的 。 字符流也是读取文件中的二进制数据，不过会把这些二进制数据转换成我们能识别的字符。

字符流 = 字节流 + 解码

FileReader

FileReader的用法：

1. 找到目标文件

2. 建立数据的输入通道

3. 读取数据

4. 关闭资源

构造方法

FileReader(String fileName) 在给定从中读取数据的文件名的情况下创建一个新FileReader。

FileReader(File file) 在给定从中读取数据的 File 的情况下创建一个新 FileReader。

FileReader(FileDescriptor fd) 在给定从中读取数据的 FileDescriptor 的情况下创建一个新 FileReader。

Sytem的in,out,err是通过FileDescriptor中的in,out,err来实现的

这个类我们好像不怎么能使用到，因为我们无法去定义其中的fd值，所以我们能使用的基本只有其中为我们定义好的in,out,err了，但是众所周知，java已将帮我们封装好了更方便的System.in/out/err了，所以平常我们基本见不到其使用，只有在看源码时，会经常碰到了。

普通方法

**它没有方法，有继承的方法：**从类 java.io.InputStreamReader 继承的方法

从类 java.io.Reader 继承的方法

从类 java.lang.Object 继承的方法

使用配合字符数组读取文件。

public static void readTest2() throws IOException{

//找到目标文件

File file = new File("F:\\1208project\\day21\\src\\day21\\Demo1.java");

// 建立数据的输入通道

FileReader fileReader = new FileReader(file);

//建立缓冲字符数组读取文件数据

char[] buf = new char[1024];

int length = 0 ;

while((length = fileReader.read(buf))!=-1){

System.out.print(new String(buf,0,length));

}

}

BufferedReader BufferedReader类-------extends----Reader(抽象类)

缓冲输入字符流出现的目的是为了提高读取文件 的效率和拓展了FileReader的功能。

其实该类内部也是维护了一个字符数组

记住：缓冲流都不具备读写文件的能力。需要传入字符输入流

构造方法

BufferedReader(Reader in) 创建一个使用默认大小输入缓冲区的缓冲字符输入流。

BufferedReader(Reader in, int sz) 创建一个使用指定大小输入缓冲区的缓冲字符输入流。

普通方法

1 void close() 关闭该流并释放与之关联的所有资源。

2 void mark(int readAheadLimit) 标记流中的当前位置。

3 boolean markSupported() 判断此流是否支持 mark() 操作（它一定支持）。

4 int read() 读取单个字符。

5 int read(char[] cbuf, int off, int len) 将字符读入数组的某一部分。

6 String readLine() 读取一个文本行。

自己去实现readLine方法：

public static String myReadLine(FileReader fileReader) throws IOException{

//创建一个字符串缓冲类对象

StringBuilder sb = new StringBuilder(); //StringBuilder主要是用于存储读取到的数据

int content = 0 ;

while((content = fileReader.read())!=-1){

if(content=='\r'){

continue;

}else if(content=='\n'){

break;

}else{

//普通字符

sb.append((char)content);

}

}

//代表已经读取完毕了。

if(content ==-1){

return null;

}

return sb.toString();

}

7 boolean ready() 判断此流是否已准备好被读取。

8 void reset() 将流重置到最新的标记。

9 long skip(long n) 跳过字符。

BufferedReader的使用步骤：

1.找到目标文件

2 .建立数据的输入通道。

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输入通道。

FileReader fileReader = new FileReader(file);

//建立缓冲输入字符流

BufferedReader bufferedReader = new BufferedReader(fileReader);

//读取数据

/\*int content = bufferedReader.read(); //读到了一个字符。 读取到的字符肯定也是从Bufferedreader内部的字符数组中获取的到。所以效率高。

System.out.println((char)content);\*/

//使用BUfferedReader拓展的功能，readLine() 一次读取一行文本，如果读到了文件的末尾返回null表示。

String line = null;

while((line = bufferedReader.readLine())!=null){ // readLine每次读取一行数据

System.out.println(Arrays.toString("aaa".getBytes()));

}

//关闭资源

bufferedReader.close();

}

### 输出流

#### OutputStream

此类是字节输出流的所有父类，它是抽象类

普通方法

void close()关闭此输出流并释放与此流有关的所有系统资源。

void flush()刷新此输出流并强制写出所有缓冲的输出字节。

void write(byte[] b) 将 b.length 个字节从指定的 byte 数组写入此输出流。

void write(byte[] b, int off, int len)参数：b：byte数组，off、len指定索引开始结束

abstract void write(int b) 将指定的字节写入此输出流。

FileOutputStream

FileOutputStream步骤

1. 找到目标文件

2. 建立数据的输出通道。

3. 把数据转换成字节数组写出。

4. 关闭资源

FileOutputStream要注意的细节：

1. 使用FileOutputStream 的时候，如果目标文件不存在，那么会自动创建目标文件对象。

2. 使用FileOutputStream写数据的时候，如果目标文件已经存在，那么会先清空目标文件中的数据，然后再写入数据。

3.使用FileOutputStream写数据的时候, 如果目标文件已经存在，需要在原来数据基础上追加数据的时候应该使用new FileOutputStream(file,true)构造函数，第二参数为true。

4.使用FileOutputStream的write方法写数据的时候，虽然接收的是一个int类型的数据，但是真正写出的只是一个字节的数据，只是把低八位的二进制数据写出，其他二十四位数据全部丢弃。00000000-000000000-00000001-11111111 511 ，11111111---> -1

构造方法

FileOutputStream(String name) 创建一个向具有指定名称的文件中写入数据的输出文件流。

FileOutputStream(String name, boolean append) 创建具有指定 name 的文件中写入数据的输出文件流。

OutputStream out = new FileOutputStream("g:\\dd.txt",true);//true不删除数据，在末尾处进行追加数据

String st = "hello nimei";

byte[] n=st.getBytes();

out.write(n,0,n.length);

out.close();

FileOutputStream(File file) 创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

FileOutputStream(File file, boolean append)创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

FileOutputStream(FileDescriptor fdObj) 创建一个向指定文件描述符处写入数据的输出文件流，该文件描述符表示一个到文件系统中的某个实际文件的现有连接。

普通方法

void close()关闭此文件输出流并释放与此流有关的所有系统资源。

protected void finalize()清理到文件的连接，并确保在不再引用此文件输出流时调用此流的 close 方法。

FileChannel getChannel()返回与此文件输出流有关的唯一 FileChannel 对象。

FileDescriptor getFD() 返回与此流有关的文件描述符。

void write(byte[] b) 将 b.length 个字节从指定 byte 数组写入此文件输出流中。

void write(byte[] b, int off, int len)将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此文件输出流。

void write(int b) 将指定字节写入此文件输出流。

通过输入流和输出流进行拷贝图片

public static void test22() throws Exception{

File file= new File("C:\\Users\\WWL\\Documents\\5.jpg");//输入流，读取

FileInputStream fi =new FileInputStream(file);

FileOutputStream out = new FileOutputStream(new File("E:\\5.jpg"));

byte[] b = new byte[1024];

int count=0;//默认先设置读了0

while((count=fi.read(b))!=-1){

out.write(b,0,count);//读了多少写多少

}

//关闭原则：先开后关，后开先关

out.close();

fi.close();

}

BufferedOutputStream 缓冲字节输出流

BufferedOutputStream --extends--> FilterOutputStream---extends-->OutputStream

出现的目的是为了提高写数据的效率。 内部也是维护了一个8kb的字节数组而已。

BufferedOutputStream 要注意的细节

使用BufferedOutStream写数据的时候，它的write方法是是先把数据写到它内部维护的字节数组中。

如果需要把数据真正的写到硬盘上面，需要调用flush方法或者是close方法、 或者是内部维护的字节数组已经填满数据的时候。

构造方法

BufferedOutputStream(OutputStream out)

BufferedOutputStream(OutputStream out, int size)

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输出通道

FileOutputStream Out= new FileOutputStream(file);

//建立缓冲输出字节流对象

BufferedOutputStream buffered= new BufferedOutputStream(Out);

//把数据写出

buffered.write(\r\n我去你妹啊，为什么可以中文啊.因为getBytes方法是默认按系统字符集编码给你存入数组的".getBytes());

//把缓冲数组中内部的数据写到硬盘上面。

//buffered.flush();

buffered.close();//此方法本身就有先执行flush()再close（）；

#### Writer

FileWriter-------extends------🡪 OutputStreamWriter（编码解码用）-------extends----🡪 Writer(抽象类)

输出字符流:

------| Writer 输出字符流的基类。 抽象类

-----------| OutputStreamWriter字符流通向字节流的桥梁：字符编码

-----------------| FileWriter 向文件数据输出字符流

它是所有的字符输出流的父类，抽象类，

FileWriter此类继承OutputStreamWriter

构造方法

1 FileWriter(File file) 根据给定的 File 对象构造一个 FileWriter 对象。

2 FileWriter(File file, boolean append) 根据给定的 File 对象构造一个 FileWriter 对象。

3 FileWriter(String fileName) 根据给定的文件名构造一个 FileWriter 对象。

4 FileWriter(String fileName, boolean append)根据给定的文件名以及指示是否附加写入数据的 boolean 值来构造 FileWriter 对象。

5 FileWriter(FileDescriptor fd) 构造与某个文件描述符相关联的 FileWriter 对象。

普通方法

它和输入流一样没有方法：继承方法：

从类 java.io.OutputStreamWriter 继承的方法

close, flush, getEncoding, write, write, write

从类 java.io.Writer 继承的方法

append, append, append, write, write

从类 java.lang.Object 继承的方法

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

FileWriter的使用步骤：

1. 找到目标文件。

2. 建立数据输出通道

3. 写出数据。

4. 关闭资源

FileWriter要注意的事项：

1. 使用FileWriter写数据的时候，FileWriter内部是维护了一个1024个字符数组的，写数据的时候会先写入到它内部维护的字符数组中，如果需要把数据真正写到硬盘上，需要调用flush或者是close方法或者是填满了内部的字符数组。

2. 使用FileWriter的时候，如果目标文件不存在，那么会自动创建目标文件。

3.使用FileWriter的时候， 如果目标文件已经存在了，那么默认情况会先清空文件中的数据，然后再写入数据 ， 如果需要在原来的基础上追加数据，需要使用“new FileWriter(File , boolean)”的构造方法，第二参数为true。

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据输出通道

FileWriter fileWriter = new FileWriter(file,true);//可以追加

//准备数据，把数据写出

String data = "今天天气非常好！！";

fileWriter.write(“\r\n”+data); //字符流具备解码的功能。\r\n换行

//刷新字符流

// fileWriter.flush();

//关闭资源

fileWriter.close();

缓冲字符输出流 BufferedWriter类-------extends----Writer(抽象类)

BufferedWriter内部只不过是提供了一个8192长度的字符数组作为缓冲区而已，拓展了FileWriter的功能。

记住：缓冲流都不具备读写文件的能力。需要传入字符输入流

构造方法

BufferedWriter(Writer out) 创建一个使用默认大小输出缓冲区的缓冲字符输出流。

BufferedWriter(Writer out, int sz)创建一个使用给定大小输出缓冲区的新缓冲字符输出流。

普通方法

void close()关闭此流，但要先刷新它。

void flush() 刷新该流的缓冲。

void newLine()写入一个行分隔符。

bufferedWriter.newLine(); //newLine() 换行。 实际上就是想文件输出\r\n.

void write(char[] cbuf, int off, int len) 写入字符数组的某一部分。

void write(int c) 写入单个字符。

void write(String s, int off, int len) 写入字符串的某一部分。

BufferedReader的使用步骤：

1.找到目标文件

2 .建立数据的输入通道。

//找到目标文件

File file = new File("F:\\a.txt");

//位置在工程目录编译后的文件夹中：bin，它不是web项目 System.out.println(FileTest.class.getClassLoader().getResource("a.txt").getPath());

//建立数据的输出通道

FileWriter fileWriter = new FileWriter(file,true);

//建立缓冲输出流对象

BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

//写出数据

// bufferedWriter.newLine(); //newLine() 换行。 实际上就是向文件输出\r\n.

bufferedWriter.write("\r\n");

bufferedWriter.write("前两天李克强来萝岗！！");

//关闭资源

bufferedWriter.flush();

// bufferedWriter.close();

### 转换流

注意：没有字符流转换为字节流的，：你见过有高富帅想变为穷小子的？

输入字节流的转换流：InputStreamReader 是字节流通向字符流的桥

输出字节流的转换流： OutputStreamWriter 可以把输出字节流转换成输出字符流 。

转换流的作用：聊天使用到它

1. 如果目前所获取到的是字节流需要转换字符流使用，就可以使用转换流。字节流----> 字符流

2. 使用转换流可以指定编码表进行读写文件。

public class InputStreamReader extends Reader

它的子类就是字符输入流FileReader

InputStreamReader 是字节流通向字符流的桥

InputStreamReader 输入转换流

InputStreamReader 是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

每次调用 InputStreamReader 中的一个 read() 方法都会导致从底层输入流读取一个或多个字节。要启用从字节到字符的有效转换，可以提前从底层流读取更多的字节，使其超过满足当前读取操作所需的字节。

为了达到最高效率，可要考虑在 BufferedReader 内包装 InputStreamReader。例如：

BufferedReader in= new BufferedReader(new InputStreamReader(System.in));

构造方法

InputStreamReader(InputStream in, String charsetName) 创建使用指定字符集的 InputStreamReader。

//使用输入字节流的转换流指定码表进行读取文件数据

public static void readTest2() throws IOException{

File file = new File("F:\\a.txt");

FileInputStream fileInput = new FileInputStream(file);

//创建字节流的转换流并且指定码表进行读取

InputStreamReader input = new InputStreamReader(fileInput,"utf-8");//在文件a编码是utf-8，这里也要设置成utf-8才不会乱码

char[] buf = new char[1024];

int length = 0;

while((length = input.read(buf))!=-1){

System.out.println(new String(buf,0,length));

}

InputStreamReader(InputStream in) 创建一个使用默认字符集的 InputStreamReader。

InputStreamReader(InputStream in, Charset cs)创建使用给定字符集的 InputStreamReader。

InputStreamReader(InputStream in, CharsetDecoder dec) 创建使用给定字符集解码器的 InputStreamReader。

普通方法

void close()关闭该流并释放与之关联的所有资源。

String getEncoding()返回此流使用的字符编码的名称。

int read()读取单个字符。

int read(char[] cbuf, int offset, int length)将字符读入数组中的某一部分。

boolean ready()判断此流是否已经准备好用于读取。

OutputStreamWriter 输出转换流

OutputStreamWriter 是字符流通向字节流的桥梁：可使用指定的 charset 将要写入流中的字符编码成字节。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

每次调用 write() 方法都会导致在给定字符（或字符集）上调用编码转换器。在写入底层输出流之前，得到的这些字节将在缓冲区中累积。可以指定此缓冲区的大小，不过，默认的缓冲区对多数用途来说已足够大。注意，传递给 write() 方法的字符没有缓冲。

为了获得最高效率，可考虑将 OutputStreamWriter 包装到 BufferedWriter 中，以避免频繁调用转换器。例如：

Writer out= new BufferedWriter(new OutputStreamWriter(System.out));

构造方法

OutputStreamWriter(OutputStream out, String charsetName)创建使用指定字符集的 OutputStreamWriter。

//使用输出字节流的转换流指定码表写出数据

public static void writeTest2() throws IOException{

File file = new File("F:\\a.txt");

//建立数据的输出通道

FileOutputStream fileOutput = new FileOutputStream(file);

//把输出字节流转换成字符流并且指定编码表。

OutputStreamWriter out = new OutputStreamWriter(fileOutput, "utf-8");

outputStreamWriter.write("新中国好啊");

//关闭资源

out.close();

OutputStreamWriter(OutputStream out) 创建使用默认字符编码的 OutputStreamWriter。

OutputStreamWriter(OutputStream out, Charset cs) 创建使用给定字符集的 OutputStreamWriter。

OutputStreamWriter(OutputStream out, CharsetEncoder enc) 创建使用给定字符集编码器的 OutputStreamWriter。

普通方法

void close()关闭此流，但要先刷新它。

void flush()刷新该流的缓冲。

String getEncoding()返回此流使用的字符编码的名称。

void write(char[] cbuf, int off, int len) 写入字符数组的某一部分。

void write(int c) 写入单个字符。

void write(String str, int off, int len) 写入字符串的某一部分。

### 打印流

PrintWriter------extends---- Writer 字符打印流

PrintStream -----extends --- FilterOutputStream 字节打印流

打印流（printStream） 打印流可以打印任意类型的数据，而且打印数据之前都会先把数据转换成字符串再进行打印。

打印到别的地方

构造方法

1、PrintWriter(File file) 使用指定文件创建不具有自动行刷新的新 PrintWriter。

//收集异常的日志信息。

File logFile = new File("F:\\2015年1月8日.log");

PrintStream dd = new PrintStream( new FileOutputStream(logFile,true) );

try{

int c = 4/0; //被除数为0异常

}catch(Exception e){

e.printStackTrace(dd);//异常信息通过流参数进行传到文件上去

}

2、PrintWriter(String fileName) 创建具有指定文件名称且不带自动行刷新的新 PrintWriter。

//创建一个打印流

PrintStream printStream = new PrintStream("g:\\a.txt");

printStream.println(97);// System.out.println("");一样效果

3、PrintWriter(File file, String csn) 创建具有指定文件和字符集且不带自动刷行新的新 PrintWriter。

4、PrintWriter(OutputStream out) 根据现有的 OutputStream 创建不带自动行刷新的新 PrintWriter。

5、PrintWriter(OutputStream out, boolean autoFlush) 通过现有的 OutputStream 创建新的 PrintWriter。

6、PrintWriter(String fileName, String csn) 创建具有指定文件名称和字符集且不带自动行刷新的新 PrintWriter。

7、PrintWriter(Writer out) 创建不带自动行刷新的新 PrintWriter。

8、PrintWriter(Writer out, boolean autoFlush) 创建新 PrintWriter。

普通方法

Append方法3个重载，format2个重载，print9个重载，printf2个重载，write5个重载，println10个重载

File file = new File("g:\\a.txt");

//创建一个打印流

PrintStream printStream = new PrintStream("g:\\a.txt");

printStream.println(97);//原数字97打印出去

System.setOut(printStream);//设置在哪里输出

System.out.println("我在哪？");

//最常用的方法：System.out获得对象PrintStream再调用println方法打印到控制台

System.out.println("你用的的");

打印到控制台操作：

PrintStream a = new PrintStream(System.out);

a.print("你好");

a.write("ni你好".getBytes());

打印字符流控制台操作：需要关流或者flush()才有输出

PrintWriter p = new PrintWriter(System.out);

p.println("你好");

p.write("谢谢"+"\r\n"+"谢你妹");

//p.flush();

p.close();

### 对象流

对象的输入输出流 : 对象的输入输出流 主要的作用是用于写对象的信息与读取对象的信息。 对象信息一旦写到文件上那么对象的信息就可以做到持久化了

对象输入输出流要注意的细节：

1. 如果对象需要被写出到文件上，那么对象所属的类必须要实现Serializable接口。 Serializable接口没有任何的方法，是一个标识接口而已。

2. 对象的反序列化创建对象的时候并不会调用到构造方法的、

3. serialVersionUID 是用于记录class文件的版本信息的，serialVersionUID这个数字是通过一个类的类名、成员、包名、工程名算出的一个数字。

4. 使用ObjectInputStream反序列化的时候，ObjeectInputStream会先读取文件中的serialVersionUID，然后与本地的class文件的serialVersionUID进行对比，如果这两个id不一致，那么反序列化就失败了。一旦改动属性添加删除等，id就会不一样

5. 如果序列化与反序列化的时候可能会修改类的成员，那么最好一开始就给这个类指定一个serialVersionUID，如果一类已经指定的serialVersionUID，然后在序列化与反序列化的时候，jvm都不会再自己算这个 class的serialVersionUID了。

6. 如果一个对象某个数据不想被序列化到硬盘上，可以使用关键字transient修饰。

7. 如果一个类维护了另外一个类的引用，那么另外一个类也需要实现Serializable接口。

对象输入流 ObjectInputStream

ObjectInputStream ----extends--- InputStream

对象输出流 ObjectOutputStream

整体例子

//把文件中的对象信息读取出来-------->对象的反序列化

public static void readObj() throws IOException, ClassNotFoundException{

//找到目标文件

File file = new File("F:\\obj.txt");

//建立数据的输入通道

FileInputStream fileInput = new FileInputStream(file);

//建立对象的输入流对象

ObjectInputStream object = new ObjectInputStream(fileInput);

//读取对象信息

User user = (User) object.readObject(); //创建对象肯定要依赖对象所属 的class文件。

System.out.println("对象的信息："+ user);

}

//定义方法把对象的信息写到硬盘上------>对象的序列化。

public static void writeObj() throws IOException{

//把user对象的信息持久化存储。

Address add = new Address("中国","广州");javabean类要记得实现Serializable接口

User user = new User("admin","123",15,add);

//找到目标文件

File file = new File("F:\\obj.txt");

//建立数据输出流对象

FileOutputStream Output = new FileOutputStream(file);

//建立对象的输出流对象

ObjectOutputStream objectOutput = new ObjectOutputStream(Output);

//把对象写出

objectOutput.writeObject(user);

//关闭资源

objectOutput.close();

class User implements Serializable{ 实现接口

private static final long serialVersionUID = 1L;I uid

String userName ;

String password;

transient int age; //此属性不想被系列化到硬盘，使用transient 透明，

Address address ;

循环读取对象

java.io.StreamCorruptedException: invalid type code: AC问题解决，问题描述：

每次向一个文件中序列化对象时 ，每次只想向文件末尾追加对象，而不是覆盖，可以使用FileInputStream（文件名，true）；在读取数据的时候第一次会正常读取，不会报错，当读取第二次的时候，就会报出java.io.StreamCorruptedException: invalid type code: AC的错误。

问题分析：

由于用FileOutputStream（文件名，true）向同一个文件中序列化对象，每次都会向文件中序列化一个header。在反序列化的时候每个 ObjectInputStream 对象只会读取一个header，那么当遇到第二个的时候就会报错，导致出现异常。

解决方案：

一共三种方法，推荐使用第二种；

一、运用集合：

在第一次序列化对象之前，把要序列化的对象添加到集合中，把这个集合序列化到文件中。然后每次序列化之前，除了把准备序列化的对象添加到集合中，再把已经序列化的集合反序列化回来添加到集合中，然后再把集合序列化到文件中。

二、重写ObjectOutputSream的writeStreamHeader（）方法：

判断是不是第一次写入，若是则写入头部，若不是则不写入头部

重写writeStreamHeader（）方法

class MyObjectOutputStream extends ObjectOutputStream{

public MyObjectOutputStream(OutputStream out) throws IOException {

super(out);

}

public void writeStreamHeader() throws IOException{

return;

}

}

//序列化

public static void set(File file,Person p) throws Exception{

FileOutputStream fos = new FileOutputStream(file,true);

/\*\*

判断是否是第一次写入

\*/

if(file.length()<1){

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(p);

oos.close();

}else{

MyObjectOutputStream mos = new MyObjectOutputStream(fos);

mos.writeObject(p);

mos.close();

}

}

//反序列化

public static List<Person> get(File file) throws Exception{

List<Person> list = new ArrayList<Person>();

FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis);

while(fis.available()>0){

Person p = (Person) ois.readObject();

list.add(p);

}

ois.close();

return list;

}

三：不重写ObjectOutputSream的writeStreamHeader（）方法。在反序列化的while循环中，每次都创建一个新的ObjectInputStream用来读取header

ObjectInputStream obj=null;

try {

File file= new File("bin/User.txt");

if(!file.exists()) {

file.createNewFile();

}

InputStream in =UserDao.class.getClassLoader().getResourceAsStream(file.getName());

List<User> list= new ArrayList<User>();

while(true) {

try {

//每次都new一个新的ObjectInputStream：由于用FileOutputStream（文件名，true）向同一个文件中序列化对象，每次都会向文件中序列化一个header。反序列化的时候每个 ObjectInputStream 对象只会读取一个header，所以。。

obj=new ObjectInputStream(in);//把它放入循环中

User user=(User) obj.readObject();

list.add(user);

}catch(EOFException e) {

break; //读完末尾会抛异常，进行跳出循环

}catch(NullPointerException e){

continue; //发现是空的对象进行跳出本次循环

}

## Java.net

### InetAddress(IP类)

Class InetAddress extends Object implements Serializable

网络编程： 网络编程主要用于解决计算机与计算机（手机、平板..）之间的数据传输问题。

不需要基于html页面就可以达到数据之间的传输。 比如： feiQ , QQ , 微信....

网页编程： 就是要基于html页面的基础上进行数据的交互的。 比如： 珍爱网、 oa(办公自动化)、 高考的报告系统...

计算机网络： 分布在不同地域 的计算机通过外部设备链接起来达到了消息互通、资源共享的效果就称作为一个计算机网络。

网络通讯的三要素： 1. IP 2. 端口号。3. 协议.

192.168.10.1

IP地址：IP地址的本质就是一个由32位的二进制数据组成的数据。 后来别人为了方便我们记忆IP地址，就把IP地址切成了4份，每份8bit. 2^8 = 0~255

00000000-00000000-00000000-00000000

IP地址 = 网络号+ 主机号。

IP地址的分类：

A类地址 = 一个网络号 + 三个主机号 2^24 政府单位

B类地址 = 两个网络号+ 两个主机号 2^16 事业单位（学校、银行..）

C类地址= 三个网络号+ 一个主机号 2^8 私人使用..

端口号的范围： 0~65535

从0到1023，系统紧密绑定于一些服务。

1024~65535 我们可以使用....

获取此类对象三个：

1）static InetAddress getLocalHost() 返回本地主机。获取本机的IP地址对象

InetAddress address = InetAddress.getLocalHost();

2）static InetAddress getByName(String host)在给定主机名的情况下确定主机的 IP 地址。

//获取别人机器的iP对象

InetAddress address=InetAddress.getByName("192.168.1.1");

3）static InetAddress[] getAllByName(String host)根据系统上名称服务返回其 IP 地址组成的数组

//通过数组获得百度的ip

for(InetAddress a:address1){

System.out.println(a.getHostAddress());//返回一个IP地址的字符串表示形式

System.out.println(a.getHostName());//返回主机名

}

其它方法

1）String getHostAddress()返回 IP 地址字符串（以文本表现形式）。

System.out.println("IP地址："+address.getHostAddress());打印：ip地址：192.168.0.5

2）String getHostName()获取此 IP 地址的主机名。

System.out.println("主机名："+address.getHostName());打印：主机名：WWL-PC

整体例子

InetAddress assress= InetAddress.getLocalHost();//返回异常，主要怕你机器没有装网卡，连ip都获取不到

System.out.println(assress.getHostAddress());//返回一个IP地址的字符串表示形式

System.out.println(assress.getHostName());//返回主机名

//可以根据一个IP地址的字符串形式或者是一个主机名生成一个IP地址对象。

InetAddress address = InetAddress.getByName("Jolly-pc140116");

System.out.println("IP地址："+address.getHostAddress());

System.out.println("主机名："+address.getHostName());

网络通讯的协议： udp通讯协议 tcp通讯协议。

### TCP协议

TCP通讯协议特点：

1. tcp是基于IO流进行数据 的传输 的，面向连接。

2. tcp进行数据传输的时候是没有大小限制的。

3. tcp是面向连接，通过三次握手的机制保证数据的完整性。 可靠协议。

4. tcp是面向连接的，所以速度慢。

5. tcp是区分客户端与服务端的。比如： 打电话、 QQ\feiQ的文件传输、 迅雷下载....

tcp协议下的Socket：需要用到的类和方法

Socket(客户端) , tcp的客户端一旦启动马上要与服务端进行连接。

ServerSocket(服务端类)

Socket（客户端）java.net. Socket implements java.io.Closeable

tcp的客户端使用步骤：

1. 建立tcp的客户端服务。

2. 获取到对应的流对象。

3.写出或读取数据

4. 关闭资源。

//建立tcp的服务(ip和端口号)

Socket socket = new Socket(InetAddress.getLocalHost(),9090);

//获取到Socket的输出流对象

OutputStream outputStream = socket.getOutputStream();

//利用输出流对象把数据写出即可。

outputStream.write("服务端你好".getBytes());

//获取到输入流对象，读取服务端回送的数据。

InputStream inputStream = socket.getInputStream();

byte[] buf = new byte[1024];

int length = inputStream.read(buf);

System.out.println("客户端接收到的数据："+ new String(buf,0,length));

//关闭资源

socket.close();

构造方法

Socket()通过系统默认类型的 SocketImpl 创建未连接套接字

Socket(InetAddress address, int port)创建一个流套接字并将其连接到指定 IP 地址的指定端口号。

Socket(InetAddress address, int port, InetAddress localAddr, int localPort) 创建一个套接字并将其连接到指定远程地址上的指定远程端口。

Socket(Proxy proxy) 创建一个未连接的套接字并指定代理类型（如果有），该代理不管其他设置如何都应被使用。

protected Socket(SocketImpl impl) 使用用户指定的 SocketImpl 创建一个未连接 Socket。

Socket(String host, int port) 创建一个流套接字并将其连接到指定主机上的指定端口号。

ServerSocket（服务端）java.net ServerSocket implements java.io.Closeable

常见异常：java.net.BindException: 端口被占用。

ServerSocket的使用 步骤

1. 建立tcp服务端 的服务。

2. 接受客户端的连接产生一个Socket. serverSocket.accept()

3. 获取对应的流对象读取或者写出数据。

4. 关闭资源。

ServerSocket例子

//建立Tcp的服务端,并且监听一个端口。

ServerSocket serverSocket = new ServerSocket(9090);

//接受客户端的连接

Socket socket = serverSocket.accept(); //accept() 接受客户端的连接 该方法也是一个阻塞型的方法，没有客户端与其连接时，会一直等待下去。

//获取输入流对象，读取客户端发送的内容。

InputStream inputStream = socket.getInputStream();

byte[] buf = new byte[1024];

int length = 0;

length = inputStream.read(buf);

System.out.println("服务端接收："+ new String(buf,0,length));

//获取socket输出流对象，想客户端发送数据

OutputStream outputStream = socket.getOutputStream();

outputStream.write("客户端你好啊！".getBytes());

//关闭资源

serverSocket.close();

构造方法

ServerSocket()创建非绑定服务器套接字。

**ServerSocket(int port) 创建绑定到特定端口的服务器套接字**。

ServerSocket(int port, int backlog) 利用指定的 backlog 创建服务器套接字并将其绑定到指定的本地端口号。

ServerSocket(int port, int backlog, InetAddress bindAddr) 使用指定的端口、侦听 backlog 和要绑定到的本地 IP 地址创建服务器。

模拟Tomcat例子

public class TomcatDemo extends Thread {

Socket socket;

public TomcatDemo(Socket socket){

this.socket = socket;

}

public void run() {

try {

//获取socket的输出流对象

OutputStream outputStream = socket.getOutputStream();

//把数据写到浏览器上

outputStream.write("<html><head><title>aaa</title></head><body>你好啊浏览器</body></html>".getBytes());

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

public static void main(String[] args) throws IOException {

//建立tcp的服务端

ServerSocket serverSocket = new ServerSocket(9090);

//不断的接受客户端的连接

while(true){

Socket socket = serverSocket.accept();

new TomcatDemo(socket).start();

}

}

}

聊天例子

需求： 客户端与服务端一问一答聊天。

1.如果使用BuffrerdReader的readline方法一定要加上\r\n才把数据写出。

2.使用字符流一定要调用flush方法数据才会写出。

//聊天的客户端

public class ChatClient {

public static void main(String[] args) throws IOException {

//建立tcp的客户端服务

Socket ket = new Socket(InetAddress.getLocalHost(),9090);

//获取socket的输出流对象。

OutputStreamWriter Out = new OutputStreamWriter(socket.getOutputStream());

//获取socket的输入流对象

BufferedReader s = new BufferedReader(new InputStreamReader(ket.getInputStream()));

//获取键盘的输入流对象，读取数据

BufferedReader keyReader = new BufferedReader(new InputStreamReader(System.in));

String line = null;

//不断的读取键盘录入的数据，然后把数据写出

while((line = keyReader.readLine())!=null){

out.write(line+"\r\n");

//刷新

out.flush();

//读取服务端回送的数据

line = s.readLine();

System.out.println("服务端回送的数据是："+line);

}

//关闭资源

ket.close();

}

//聊天的服务端

public class ChatServer {

public static void main(String[] args) throws IOException {

//建立tcp的服务端

ServerSocket serverSocket = new ServerSocket(9090);

//接受客户端的连接，产生一个SOcket

Socket socket = serverSocket.accept();

//获取到Socket的输入流对象

BufferedReader socketReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

//获取到Socket输出流对象

OutputStreamWriter socketOut = new OutputStreamWriter(socket.getOutputStream());

//获取键盘的输入流对象

BufferedReader keyReader = new BufferedReader(new InputStreamReader(System.in));

//读取客户端的数据

String line = null;

while((line = socketReader.readLine())!=null){

System.out.println("服务端接收到的数据："+ line);

System.out.println("请输入回送给客户端的数据：");

line = keyReader.readLine();

socketOut.write(line+"\r\n");

socketOut.flush();

}

//关闭资源

serverSocket.close();

}

### UDP协议

使用到两个类：服务端和客户端都是一样的操作

public class DatagramSocket extends Object (udp插座服务)

public final class DatagramPacket extends Object (数据包类)

在java中网络通讯业称作为Socket(插座)通讯，要求通讯 的两台器都必须要安装Socket。

不同的协议就有不同的插座（Socket）

UDP通讯协议的特点：

1. 将数据极封装为数据包，面向无连接。

2. 每个数据包大小限制在64K中

3.因为无连接，所以不可靠

4. 因为不需要建立连接，所以速度快

5.udp 通讯是不分服务端与客户端的，只分发送端与接收端。比如： 物管的对讲机, 游戏...

主要区别：send()发送数据和receive(datagramPacket)接收数据

udp协议下的Socket:

DatagramSocket(udp插座服务)

DatagramPacket(数据包类)

DatagramPacket(buf, length, address, port)

buf: 发送的数据内容

length : 发送数据内容的大小。

address : 发送的目的IP地址对象

port : 端口号。

UDP基本操作

发送端的使用步骤：

1. 建立udp的服务。

2. 准备数据，把数据封装到数据包中发送。 发送端的数据包要带上ip地址与端口号。

3. 调用udp的服务，发送数据。

4. 关闭资源。

public class Demo1Sender {

public static void main(String[] args) throws IOException {

//建立udp的服务

DatagramSocket datagramSocket = new DatagramSocket();

//准备数据，把数据封装到数据包中。

String data = "这个是我第一个udp的例子..";

//创建了一个数据包

DatagramPacket packet = new DatagramPacket(data.getBytes(), data.getBytes().length,InetAddress.getLocalHost() , 9090);

//调用udp的服务发送数据包

datagramSocket.send(packet);

//关闭资源 ---实际上就是释放占用的端口号

datagramSocket.close();

}

接收端

接收端的使用步骤

1. 建立udp的服务

2. 准备空 的数据 包接收数据。

3. 调用udp的服务接收数据。

4. 关闭资源

public class Demo1Receive {

public static void main(String[] args) throws IOException {

//建立udp的服务 ，并且要监听一个端口。

DatagramSocket socket = new DatagramSocket(9090);

//准备空的数据包用于存放数据。

byte[] buf = new byte[1024];

DatagramPacket datagramPacket = new DatagramPacket(buf, buf.length); // 1024

//调用udp的服务接收数据

socket.receive(datagramPacket); //receive是一个阻塞型的方法，没有接收到数据包之前会一直等待。 数据实际上就是存储到了byte的自己数组中了。

System.out.println("接收端接收到的数据："+ new String(buf,0,datagramPacket.getLength())); // getLength() 获取数据包存储了几个字节。

//关闭资源

socket.close();

模拟数据丢失

udp是一个不可靠（数据包可能会丢失）的协议

什么情况下数据包会出现丢失呢？

1.带宽不足 。

2.cpu的处理能力不足。

发送端

//建立udp的服务

DatagramSocket socket = new DatagramSocket();

//准备数据，数据封装到数据中发送

DatagramPacket packet = null;

for(int i = 0 ; i< 10; i++){ //连续发送10个数据包

String data =i +"这里省略非常多数据字符串，测试自己写很多内容上去 ";

packet = new DatagramPacket(data.getBytes(), data.getBytes().length, InetAddress.getLocalHost(), 9090);

//发送数据包

socket.send(packet);

}

//关闭资源

socket.close();

接收端

//建立udp的服务

DatagramSocket socket = new DatagramSocket(9090);

//建立空的数据包存储数据

byte[] buf = new byte[1024];

DatagramPacket packet = new DatagramPacket(buf, buf.length);

//不断接收数据包

while(true){

socket.receive(packet);

System.out.println(new String(buf,0,packet.getLength()));

Thread.sleep(10);//模仿cpu不足造成数据丢失

}

FQ、QQ原理广播地址255

每个网络程序都有自己所处理的特定格式数据,如果接收到的数据不符合指定的格式，那么就会被当成垃圾数据丢弃。(加密..)

飞Q接收的数据格式：

version:time :sender : ip: flag:content ;

版本号 时间 发送人 :IP： 发送的标识符(32): 真正的内容;

在udp协议中，有一个IP地址称作为广播地址，广播地址就是主机号为255地址。

给广播IP地址发送消息的时候，在同一个网络段的机器都可以接收 到信息。

192.168.15.255

//使用udp协议给飞Q发送消息。

// 建立udp的服务

DatagramSocket socket = new DatagramSocket();

// 准备数据，把数据封装到数据包中

String data = getData("feiQ你好！");

DatagramPacket packet = new DatagramPacket(data.getBytes(),

data.getBytes().length,

InetAddress.getByName("192.168.15.255"), 2425);

// 发送数据

socket.send(packet);

// 关闭资源

socket.close();

}

// 把数据拼接成指定格式的数据

public static String getData(String content) {

StringBuilder sb = new StringBuilder();

sb.append("1.0:");

sb.append(System.currentTimeMillis() + ":");

sb.append("习大大:");

sb.append("192.168.10.1:");

sb.append("32:");

sb.append(content);

return sb.toString();

}

# XML

介绍：

三种操作方式：

DOM

基于XML文档树结构的解析

适用于多次访问的XML文档

特点：比较消耗资源

SAX

基于事件的解析

适用于大数据量的XML文档

特点：占用资源少，内存消耗小

DOM4J

非常优秀的Java XML API

性能优异、功能强大

开放源代码

DOM4J解析XML文件

DOM4J是第三方的，基于Dom进行优化的：需要导入包：dom4j-1.6.1.jar

hibernate包中的jaxen-1.1-beta-7.jar

<书架>

<书 id="1">

<书名>倚天屠龙记</书名>

<作者>金庸</作者>

<价格>100</价格>

</书>

<书 id="2">

<书名>陆小凤</书名>

<作者>古龙</作者>

<价格>99</价格>

</书>

</书架>

整体简单流程

//第一步 创建对象 通过流read读取全部xml内容

SAXReader sr = new SAXReader();

Document d=sr.read(Demo1.class.getClassLoader().getResourceAsStream("a.xml"));

//第二步 获得根节点对象Element对象

Element root=d.getRootElement();

//第三步 获得根节点中的下一个标签<书>：用根节点的对象root获取

有3种方式 ①获取root下指定的标签：root.element("书");

②获取全部：list获取root.elements();迭代：root.elementIterator()

③获取重名的名字全部：list获取root.elements(“书”);迭代：root.elementIterator(“书”)

List<Element> list=root.elements();

Element shu=list.get(1);

Element shu2=root.element("书");

//第四步 获取属性得到对象Attribute

Attribute id=shu.attribute("id");

String idName=id.getName();//打印：id 就是属性名字不是值

String idValue2=id.getValue();//打印：值 ：1

String idValue=id.getStringValue();//也是打印：1

System.out.println(shu.element("作者").getStringValue());//只有StringValue这个属性，打印：金庸

Element shuming=shu.element("书名");

shuming.getText(); //获取到书名：倚天屠龙记

System.out.println();

获取Document对象，得到整个文档信息

public static void loadDocument(){

try{

SAXReader saxReader = new SAXReader();

doc = saxReader.read(new File("a.xml"));

}catch (Exception e) {

// TODO: handle exception

e.printStackTrace();

}

}

进行查询

public static void showPhoneInfo() {

//第一步 获取XML的根节点

Element root = doc.getRootElement();

//第二步 遍历所有的Brand标签

for (Iterator itBrand = root.elementIterator(); itBrand.hasNext();) {

Element brand = (Element) itBrand.next();

// 输出标签的name属性

System.out.println("品牌：" + brand.attributeValue("name"));

// 遍历Type标签

for (Iterator itType = brand.elementIterator(); itType.hasNext();) {

Element type = (Element) itType.next();

// 输出标签的name属性

System.out.println("\t型号：" + type.attributeValue("name"));

}

}

}

添加操作

public static void addNewPhoneInfo(){

// 获取XML的根节点

Element root = doc.getRootElement();

// 创建Brand标签

Element el = root.addElement("Brand");

// 给Brand标签设置属性

el.addAttribute("name", "三星");

// 创建Type标签

Element typeEl = el.addElement("Type");

// 给Type标签设置属性

typeEl.addAttribute("name", "Note4");

saveXML("src/收藏信息.xml");

}

修改操作

public static void updatePhoneInfo(){

// 获取XML的根节点

Element root = doc.getRootElement();

int id = 0;

for (Iterator itBrand = root.elementIterator(); itBrand.hasNext();) {

Element brand = (Element) itBrand.next();

id++;

brand.addAttribute("id", id + "");

}

saveXML("src/收藏信息.xml");

}

删除操作

public static void deleteItem(){

// 获取XML的根节点

Element root = doc.getRootElement();

int id = 0;

for (Iterator itBrand = root.elementIterator(); itBrand.hasNext();) {

Element brand = (Element) itBrand.next();

if (brand.attributeValue("name").equals("华为")) {

brand.getParent().remove(brand);

}

}

// saveXML("src/收藏信息.xml");

}

进行修改、增加、删除最后需要进行重新保存操作

public static void saveXML(String path){

try {

OutputFormat format = OutputFormat.createPrettyPrint();

format.setEncoding("GBK"); // 指定XML编码

XMLWriter writer;

writer = new XMLWriter(new FileWriter(path), format);

writer.write(doc);

writer.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

DOM 解析XMl文件

DOM解析xml，java专门有提供包org.w3c.dom,不用第三方提供的包

DOM介绍

文档对象模型（Doucument Object Model）

DOM包xml文档映射成一个倒挂的树

常用接口类介绍

常用接口：Document：表示整个XML 文档 Element【元素】

常用两个方法：

getElementsByTagName(String Tag)按文档顺序返回文档中指定标记名称的所有元素集合

createElement(String tagName) 创建指定标记名称的元素

Node:该文档树中的单个节点

NodeListgetChildNodes() 获取该元素的所有子节点，返回节点集合

Element：XML 文档中的一个元素

getTagName() 获取元素名称

DOM对XMl整体操作

DOM解析XMl文件步骤：

创建解析器工厂对象

解析器工厂对象创建解析器对象

解析器对象指定xml文件创建Document对象

以Document对象为起点操作DOM树

<?xml version="1.0" encoding="GB2312"?>

<PhoneInfo>

<Brand name="华为">

<Type name="U8650"/>

<Type name="HW123"/>

<Type name="HW321"/>

</Brand>

<Brand name="苹果">

<Type name="iPhone4"/>

</Brand>

</PhoneInfo>

第一步 获取Doucument方法 获得 整个XML 文档

获取DOM树

public void DocumentTest(){

//第一步：创建解析器工厂对象,打开源码它是一个抽象类，所以不能new，Builder【建立者，建筑者】

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

try {

//第二步：解析器工厂对象创建解析器对象

DocumentBuilder documentBuilder= factory.newDocumentBuilder();

//第三步：解析器对象指定xml文件创建Document对象,注意此包是：org.w3c.dom.Document;

document=documentBuilder.parse("收藏信息.xml");

} catch (SAXException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} catch (ParserConfigurationException e) {

e.printStackTrace();

}

}

第二步 进行查询操作 获取节点、元素方法

public void query(){

//第一步：Document对象为起点操作DOM树,

NodeList st= document.getElementsByTagName("Brand");

for(int i=0;i<st.getLength();i++){

//第二步：获取每一个标签为Brand的节点

Node node=st.item(i);

//把节点Node强制转换成元素Element,元素才有获得属性操作

Element ele=(Element)node;

//使用获取属性方法attribute

String name=ele.getAttribute("name");

System.out.println(name);

//获取子节点操作；Child【儿童，小孩】;node.getChildNodes();这里可以是强制类型转换后的ele，也可以是没有强制类型转换的节点：node。效果一样的

NodeList dd=node.getChildNodes();

for(int j=0;j<dd.getLength();j++){

Node node1=dd.item(j);

//这里要注意，空格会认为是文本，需要判断，类型是元素才进行强制类型转换

if(node1.getNodeType()==Node.ELEMENT\_NODE){

Element ele1=(Element)node1;

String name2=ele1.getAttribute("name");

System.out.println(name2);

System.out.println("-------------");

第三步 添加操作

public void add(){

// (1)创建Brand元素

Element brand=document.createElement("Brand");

// (2)设置Brand元素的name=“三星”

brand.setAttribute("name", "三星");

// (3)创建Type元素

Element ele1=document.createElement("Type");

// (4)设置Brand元素的name=“Note3”

ele1.setAttribute("name", "Note3");

// (5)type加到Brand里面去

brand.appendChild(ele1);

// (6)通过document取到最终元素PhoneInfo，再把Brand加到<PhoneInfo>元素中

document.getElementsByTagName("PhoneInfo").item(0).appendChild(brand);

this.saveXMl("new.xml");

}

第四步 修改方法

//修改元素，为每一个Brand添加ID属性，取值分别为0、1、2

public void update(){

//获取每一个Brand元素

NodeList brands=document.getElementsByTagName("Brand");

for(int i=0;i<brands.getLength();i++){

Node brand=brands.item(i);

Element eleBrand=(Element)brand;

eleBrand.setAttribute("id", i+"");

}

this.saveXMl("new.xml");

}

第五步 删除方法

//删除手机品牌为“华为”的元素

public void delete(){

NodeList brands=document.getElementsByTagName("Brand");

for(int i=0;i<brands.getLength();i++){

Node brand=brands.item(i);

Element eleBrand=(Element)brand;

if(eleBrand.getAttribute("name").equals("华为")){

eleBrand.getParentNode().removeChild(eleBrand);

}

}

this.saveXMl("new.xml");

}

第六步 修改或添加了最后还需要重新保存

步骤

获得TransformerFactory对象

创建Transformer对象

创建DOMSource对象

包含XML信息

设置输出属性

编码格式

创建StreamResult对象

包含保存文件的信息

将XML保存到指定文件中

//保存xml文件操作，如是需要一个文件保存到另一个文件，需要先进行解析获得Document再使用此方法才会有全部内容，不然只有头部信息<?xml version="1.0" encoding="GB2312" standalone="no"?><PhoneInfo><!-- 不依赖外部的DTD文档的意思 -->

//当然使用流的方式复制粘贴是另一回事了

public void saveXMl(String path){

//也是一个抽象类工厂

TransformerFactory factory=TransformerFactory.newInstance();

//设置缩进的字符数，设置4个缩进

factory.setAttribute("indent-number", 4);

try {

Transformer transformer=factory.newTransformer();

//指定特定的转换格式

transformer.setOutputProperty(OutputKeys.ENCODING, "gbk");

//这句话意思是要不要上面的设置进行缩进啊，默认是no

transformer.setOutputProperty(OutputKeys.INDENT, "yes");

//源文件

DOMSource source= new DOMSource(document);

//要保存的目标文件

//StreamResult result= new StreamResult(new FileOutputStream(path));

//使用了factory.setAttribute("indent-number", 4);要转换成字符流

StreamResult result= new StreamResult(new OutputStreamWriter(new FileOutputStream(path),"gbk"));

//进行两个参数源文件 和目标文件，有异常：TransformerException需要try catch

transformer.transform(source, result);

} catch (TransformerConfigurationException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (TransformerException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (UnsupportedEncodingException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

# 数组

数组没有进行规划类的

数组的好处： 对分配到数组对象中每一个数据都分配一个编号（索引值、角标、下标）,索引值的范围是从0开始，最大是： 长度-1.

数组中最常见的问题：

1. NullPointerException 空指针异常

原因： 引用类型变量没有指向任何对象，而访问了对象的属性或者是调用了对象的方法。

2. ArrayIndexOutOfBoundsException 索引值越界。

原因：访问了不存在的索引值。

数组中的元素与元素之间的内存地址是连续的。

例子

冒泡排序

总结：第一轮循环下来a[0]都已经拿到最大值或最小值。

int[] a= {44,60,-10,70};

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

for(int j=i+1;j<a.length;j++) {//i+1意思是不和自己进行对比，因为每次对比都已经得到此数组的最大值了，就没有必要了，

if(a[i]<a[j]) {//先从44进行对比60，位置的值就会发生变化：a[0] 已经变成了60，不是44了，后续是拿60来对比后面的，所以第一轮下来是变成了{70,44,-10,60]。

int temp=a[i];

a[i]=temp;

a[i]=a[j];

a[j]=temp;

}

System.out.println("按降序排列："+Arrays.toString(a));

反转排序：

　　　　int[] arr = {23,12,48,56,45};

　　　　for(int i=0;i<arr.length / 2;i++) {

int temp = arr[i];

arr[i] = arr[arr.length-i-1];

arr[arr.length-i-1] = temp;

}

System.out.println(Arrays.toString(arr))

二分查找法（折半）: 使用前提必需是有序的数组。因为你不排序的话就有可能找不到

不重头开始找，而是在数组中间开始找，适合数量庞大的数组查找，效率高

//定义三个变量分别记录最大、最小、中间的查找范围索引值

public int a(int[] arr, int target){

int max = arr.length-1;

int min = 0;

int mid = (max+min)/2;

while(true){

if(target>arr[mid]){ //判断此值大于中间值

min = mid+1;//把最小值=中间值加1

}else if(target<arr[mid]){ //判断此值小于于中间值

max = mid -1;//把最大值=中间值减1

}else{

//找到了元素，直接reunrn中间值结束方法

return mid;

}

//没有找到的情况是最大值小于最小值

if (max<min){

return -1;

}

//重新计算中间索引值，每次都要重新计算中间值

mid = (min+max)/2;

}

Attrays工具类

升序排序

1）sort(int[] a) 此方法很多重载

int[] a= {44,60,-10,70};

Arrays.sort(a);进行升序排序

2）binarySearch(int[] a, int key)二分搜索法来搜索指定in型数组，使用此方法必须先排序 因为：2分法是折中开始查找，此方法很多重载

int[] a= {44,60,-10,70};

Arrays.sort(a);进行升序排序

System.out.println(Arrays.binarySearch(a, 5));

3)toString()返回指定数组内容的字符串表示形式。

System.out.println(Arrays.toString(a));//[-10, 44, 60, 70]

复制数组

4)copyOf(int[] original, int newLength) ：original源数组，newLength：新数组长度，此方法很多重载

Integer[] arr= {10,8,6,8};

Integer[] arr2= Arrays.copyOf(arr, 8);

//备注：新数组指定的长度大于旧的，其它没有的值就是默认的值：Integer是一个包装类，引用类型的变量默认初始值是null

System.out.println(Arrays.toString(arr2)); //打印：[10, 8, 6, 8, null, null, null, null]

复制部分数组

5)copyOfRange(boolean[] original, int from, int to) original：源数组，

from开始复制的索引值，to：结束的索引值：包头不包尾

Integer[] arr3= Arrays.copyOfRange(arr, 1, 4);

比较两个数组元素是否一致

6)equals(int[],int[])

System.out.println(Arrays.equals(arr, arr2));//比较元素是否一致

将数组变成集合

7)List asList(T[]);

List<Integer> a=Arrays.asList(arr);

二维数组二维数组是数组的数组

二维数组 的定义格式：

数据类型[][] 变量名 = new 数据类型[长度1][长度2];

长度1：一条烟有多少盒。

长度2： 一盒烟有多少根。

二维数组 的初始化方式：

动态初始化:

数据类型[][] 变量名 = new 数据类型[长度1][长度2];

静态初始化：

数据类型[][] 变量名 = {{元素1,元素2...},{元素1,元素2...},{元素1,元素2...} ..}

int[][] a = {{10,11,9},{67,12},{33,35,39,40}};

遍历二维数组

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

for(int j = 0 ; j<arr[i].length ; j++){

System.out.print(arr[i][j]+",");

}

//换行

System.out.println();

}

int[][] arr = new int[3][4];它是数组索引1由记录了另一个数组内存地址

System.out.println("二维数组的长度："+ arr.length); // 3

System.out.println("二维数组的长度："+ arr[1].length); //这里就是长度4

# 基础知识

## 修饰符

class类修饰符

public（访问控制符），将一个类声明为公共类，他可以被任何对象访问，一个程序的主类必须是公共类。使用对象：类、接口、变量、方法

final：将一个类生命为最终（即非继承类），表示他不能被其他类继承。

Abstract：将一个类声明为抽象类，没有实现的方法，需要子类提供方法实现。

default (即缺省，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。

成员变量修饰符

public（公共访问控制符），指定该变量为公共的，他可以被任何对象的方法访问。

protected（保护访问控制符）可以别被自己的类和子类访问。在子类中可以覆盖此变量。

default默认什么也不写，在同一个包中的类可以访问，其他包中的类不能访问。

private（私有访问控制符）指定该变量只允许自己的类的方法访问，其他任何类（包括子类）中的方法均不能访问。

final，最终修饰符，指定此变量的值不能变。

static（静态修饰符）指定变量被所有对象共享，即所有实例都可以使用该变量。变量属于这个类。

transient（过度修饰符）指定该变量是系统保留，暂无特别作用的临时性变量。

volatile（易失修饰符）指定该变量可以同时被几个线程控制和修改。

方法修饰符

public（公共控制符）

private（私有控制符）只能有自己类等方法访问，其他的类不能访问（包括子类）

protected（保护访问控制符）指定该方法可以被它的类和子类进行访问。

final，指定该方法不能被重载。

static，指定不需要实例化就可以激活的一个方法。

synchronized，同步修饰符，在多个线程中，该修饰符用于在运行前，对他所属的方法加锁，以防止其他线程的访问，运行结束后解锁。

四大修饰符

Java中，可以使用访问控制符来保护对类、变量、方法和构造方法的访问。Javav支持 4 种不同的访问权限。

private : 在同一类内可见。使用对象：变量、方法。 注意：不能修饰类（外部类）

default (即缺省，什么也不写）: 在同一包内可见，不使用任何修饰符。使用对象：类、接口、变量、方法。

protected : 对同一包内的类和所有子类可见。使用对象：变量、方法。 注意：不能修饰类（外部类）:意思是不能再class上修饰。

public : 对所有类可见。使用对象：类、接口、变量、方法

公有访问修饰符-public

被声明为 public 的类、方法、构造方法和接口能够被任何其他类访问。

如果几个相互访问的 public 类分布在不同的包中，则需要导入相应 public 类所在的包。由于类的继承性，类所有的公有方法和变量都能被其子类继承。

以下函数使用了公有访问控制：

public static void main(String[] arguments) {

// ...

}

Java 程序的 main() 方法必须设置成公有的，否则，Java 解释器将不能运行该类。

受保护的访问修饰符-protected

被声明为 protected 的变量、方法和构造器能被同一个包中的任何其他类访问，也能够被不同包中的子类访问。

protected 访问修饰符不能修饰类和接口，方法和成员变量能够声明为 protected，但是接口的成员变量和成员方法不能声明为 protected。

子类能访问 protected 修饰符声明的方法和变量，这样就能保护不相关的类使用这些方法和变量。

默认访问修饰符-不使用任何关键字

使用默认访问修饰符声明的变量和方法，对同一个包内的类是可见的。接口里的变量都隐式声明为 public static final,而接口里的方法默认情况下访问权限为 public。

如下例所示，变量和方法的声明可以不使用任何修饰符。

实例

String version = "1.5.1";

boolean processOrder() {

return true;

}

私有访问修饰符-private

私有访问修饰符是最严格的访问级别，所以被声明为 private 的方法、变量和构造方法只能被所属类访问，并且类和接口不能声明为 private。

声明为私有访问类型的变量只能通过类中公共的 getter 方法被外部类访问。

Private 访问修饰符的使用主要用来隐藏类的实现细节和保护类的数据。

final(最终、修饰符)

final关键字的用法：

1. final关键字修饰一个基本类型的变量时，该变量不能重新赋值，第一次的值为最终的。

2. fianl关键字修饰一个引用类型变量时，该变量不能重新指向新的对象。

3. final关键字修饰一个方法的时候，该方法不能被重写。

4. final关键字修饰一个类的时候，该类不能被继承。

常量的修饰符一般为： public static final

## 类

### Class

**package** com.wwl.test;//此类的路径位置

//需要用到的包

**import** java.io.File;

**public** **class** B {

//私有成员变量

**private** Entity e;

//无参构造方法

**public** B() {}

**public** B(Entity e) { //有参构造方法

**this**.e=e;

}

**public** **void** b() {

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

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

}

//主方法

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

File f = **new** File("");

}

}

类的组成：

1. package包 package com.wwl.util 此类在com/wwl/util文件夹下

2. import 导包 import java.util.List; 误区：java.util.×：是不会占用内存的

3. 成员变量 随着对象的创建而存在，随着对象的消失而消失。是有默认的初始值的。

4. 构造方法 在创建对象的时候由jvm调用的。用来创建对象的

5. 普通方法 由我们使用对象调用的

6. 局部变量 是没有默认的初始值的，必须要先初始化才能使用。

构造方法（constructor）

构造方法的作用： 它是用来创建对象的

构造方法要注意的细节：

1. 构造函数 是没有返回值类型的。

2. 构造函数的函数名必须要与类名一致。

3. 构造方法是在创建对应的对象：A a=new A()时，jvm就会主动调用到对应的构造方法。

4. 如果类没有显式的写上构造方法时，那么java编译器会自动添加无参的构造方法

5. 如果已经显式的写上一个构造方法时,那么java编译器则不会添加无参的构造方法。

6. 构造方法是可以在一个类中以方法重载的形式存在多个的。

7. 参数个数相同，位置type不同，都可以视为一个不同的构造方法

方法重载跟返回值类型和修饰符无关Java的重载是发生在本类中的，重载的条件是在本类中有多个方法名相同，但参数列表不同（可能是，参数个数不同，参数类型不同）

### interface

接口的定义格式：

interface 接口名{

}

接口要注意的事项 ：

1. 接口是一个特殊的类。

2. 接口的成员变量默认的修饰符为： public static final 。那么也就是说接口中的成员变量都是常量。

3. 接口中 的方法都是抽象的方法，默认的修饰符为： public abstract。

4. 接口不能创建对象。

5. 接口是没有构造方法的。

6. 接口是给类去实现使用的，非抽象类实现一个接口的时候，必须要把接口中所有方法全部实现。

7.抽象类实现接口类后：可以实现接口的方法也可以不去实现接口的方法，那么其子类就不用实现接口的方法了

8.一个类可以实现多个接口 。

实现接口的格式：

class 类名 implements 接口名{

}

接口的作用：

1. 程序的解耦。 （低耦合）

2. 定义约束规范。

3. 拓展功能。

接口与接口之间关系： 继承关系。

接口与接口之间要注意事项：

1. 一个接口是可以继承多个接口的。

### abstract

抽象类(abstract)

格式：public abstract class B {}

抽象类的应用场景：抽象类是要有继承关系的

我们在描述一类事物的时候，发现该种事物确实存在着某种行为，

但是这种行为目前是不具体的，那么我们可以抽取这种行为的声明，但是

不去实现该种行为，这时候这种行为我们称作为抽象的行为，我们就需要使用抽象类。

抽象类的好处: 强制要求子类一定要实现指定的方法。

抽象类要注意的细节：

1. 如果方法没有方法体，那么该方法必须要使用abstract修饰,修饰成抽象的方法

2. 如果一个类出现了抽象的函数，那么该类也必须 使用abstract修饰。

3. 如果一个class类继承了抽象类，那么必须要把抽象类的所有抽象方法全部实现。

4. 抽象类可以存在非抽象方法，也可以存在抽象的方法.

5. 抽象类可以不存在抽象方法的。

5. 抽象类是不能创建对象的。

疑问：为什么抽象类不能创建对象呢？

因为抽象类是存在抽象方法的，如果能让抽象类创建对象的话，那么使用抽象的对象

调用抽象方法是没有任何意义的。

6. 抽象类是存在构造方法的，构造方法是提供给子类创建对象初始化父类的属性的。

abstract不能与以下关键字共同修饰一个方法：

1. abstract不能与private共同修饰一个方法。

2. abstract 不能与static共同修饰一个方法。

3. abstract 不能与final共同修饰一个方法。

7.抽象类实现接口可以实现接口的方法也可以不去实现接口的方法

### enum

作用：防止别人乱传数据，而必须是一定范围内的值，如：性别、方向键：上中下，季节，星期。

枚举值：通过反编译明白：枚举值其实就是一个当前枚举类的对象名称

Enum D{

男,女

}

这两个值可以看做：D 男=new D(); D 女=new D();

Mian方法中：调用

public static void main(String[] args) {

EnumTest[] nan= EnumTest.values();

nan[1].name();//打印 女

for(EnumTest n:nan) {

System.out.println(n);//打印： 男，女

}

}

构造方法：默认是private的无参构造方法，可以自定义带参私有构造方法（了解）

public enum EnumTest {

UP("上"),DOWN("下"),LEFT("左"),RIGHT("右"); //枚举值必须在第一位

public String value; //可以定义成员变量：它没有隐藏属性final、static、private

private EnumTest(String value) { //定义带参构造方法，枚举值（4个对象）就变成带括号的UP("上")

this.value=value;

}

public void p() {//可以定义成员方法：它没有隐藏属性final、static、private

System.out.println("value:"+value);

}

}

Mian方法中：调用方法 ： EnumTest.DOWN.p(); 调用不是私有的变量：String a=EnumTest.DOWN.value;

可以定义抽象方法：public abstract void a();（了解，几乎遇不到）

public enum EnumTest {

nan("男") {

@Override

public void a() {

System.out.println("我是男的");

}

},

nv("女") {

@Override

public void a() {

System.out.println("我是女的");

}

};//枚举值

public String value;//可以定义成员变量

private EnumTest(String value) {//定义成员构造函数

this.value=value;

}

//可以定义抽象方法 a()

public abstract void a();

}

Mian方法中：调用方法 ： EnumTest.nan.a();

## 面向对象

继承(extends)

继承的格式：

class 类名1 extends 类名2{

}

继承要注意的事项：

1. 父类私有的成员不能被继承。

2. 父类的构造方法不能被继承。可以调用

3. 创建子类对象时默认会先创建父类对象再调用父类无参的构造函数。

通过super(参数)指定调用父类带参构造方法

疑问： 为什么要调用父类的构造方法啊？这样子做的意义在那？

调用父类的构造方法是可以初始化从父类继承下去的属性的。

自己写的类默认都是继承Object的，构造方法第一句都是隐式有super()语句

Super关键字代表了父类空间的引用。this代表本类对象的引用

super关键字的作用：

1. 子父类存在着同名的成员时，在子类中默认是访问子类的成员，可以通过super关键字指定访问父类的成员。super.name;

2. 创建子类对象时，默认会先调用父类无参的构造方法，可以通过super关键字指定调用父类的构造方法。super(参数)

super关键字调用父类构造方法要注意的事项：

1. 如果在子类的构造方法上没有指定调用父类的构造方法，那么java编译器会在子类的构造方法上面加上super()语句。

2. super关键字调用父类的构造函数时，该语句必须要是子类构造函数中的第一个语句。

3. super与this关键字不能同时出现在同一个构造函数中调用其他的构造函数。因为两个语句都需要第一个语句。

super关键字与this关键字的区别：

1. 代表的事物不一致。

1. super关键字代表的是父类空间的引用。

2. this关键字代表的是所属函数的调用者对象。

2. 使用前提不一致。

1. super关键字必须要有继承关系才能使用。

2. this关键字不需要存在继承关系也可使用。

3. 调用构造方法的区别：

1. super关键字是调用父类的构造函数。

2. this关键字是调用本类的构造函数。

this 关键字代表了所属方法的调用者对象。

this():调用本类构造方法，有参数：调用带参的对应构造方法

this.参数名/方法名();调用成员变量/方法

this关键字作用：

1. 如果存在同名成员变量与局部变量时，在方法内部默认是访问局部变量的数据，可以通过this关键字指定访问成员变量的数据。

2. 在一个构造函数中可以调用另外一个构造函数初始化对象。

this关键字调用其他的构造函数要注意的事项：

1. this关键字调用其他的构造函数时，this关键字必须要位于构造函数中 的第一个语句。

2. this关键字在构造函数中不能出现相互调用的情况，因为是一个死循环。

this关键字要注意事项：

1. 存在同名的成员变量与局部变量时，在方法的内部访问的是局部变量(java采取的是“就近原则”的机制访问的。)

2. 如果在一个方法中访问了一个变量，该变量只存在成员变量的情况下，那么java编译器会在该变量的前面添加this关键字。

class Animal{

String name ; //成员变量

String color;

public Animal(String n , String c){

name = n;

color = c;

}

//this关键字代表了所属函数的调用者对象

public void eat(){

//System.out.println("this:"+ this);

String name = "老鼠"; //局部变量

System.out.println(name+"在吃..."); //需求： 就要目前的name是成员变量的name.

}

class Demo6

{

public static void main(String[] args)

{

Animal dog = new Animal("狗","白色"); //现在在内存中存在两份name数据。

Animal cat = new Animal("猫","黑色");

cat.eat();

}

}

方法的重写/重载

目前的问题：父类的功能无法满足子类的需求。

方法重写的前提： 必须要存在继承的关系。

方法的重写: 子父类出了同名的方法，这个我们就称作为方法的重写。

什么是时候要使用方法的重写：父类的功能无法满足子类的需求时。

方法重写要注意的事项：

1.方法重写时， 方法名与形参列表必须一致。

2.方法重写时，子类的权限修饰符必须要大于或者等于父类的权限修饰符。

3.方法重写时，子类的返回值类型必须要小于或者 等于父类的返回值类型。

4.方法重写时，子类抛出的异常类型要小于或者等于父类抛出的异常类型。

Exception(最坏)

RuntimeException(小坏)

方法的重载：在一个类中 存在两个或者两个 以上的同名函数,称作为方法重载。

方法重载的要求

1. 方法名要一致。

2. 形参列表不一致（形参的个数或形参 的类型不一致）

3. 与返回值类型无关。

toString的重写：

public String toString() {

//return super.toString();//进行重写eclipse会给你默认调用父类object操作

//return getClass().getName() + "@" + Integer.toHexString(hashCode());//源码

//A a=new A()，直接通过System.out.println(a);来打印成员变量id和那么的值

return “id:”+this.id+” name:”+this.name;

}

多态

多态：一个对象具备多种形态。

(父类的引用类型变量指向了子类的对象)

或者是接口 的引用类型变量指向了接口实现类的对象)

多态的前提：必须存在继承或者实现 关系。

动物 a = new 狗();

多态要注意的细节：

1. 多态情况下，子父类存在同名的成员变量时，访问的是父类的成员变量。

2. 多态情况下，子父类存在同名的非静态的成员方法时，访问的是子类的成员方法。

3. 多态情况下，子父类存在同名的静态的成员方法时，访问的是父类的成员方法。

4. 多态情况下，不能访问子类特有的成员。

如果需要访问子类特有的成员，那么需要进行类型强制转换.

总结：多态情况下，子父类存在同名的成员时，访问的都是父类的成员，除了在同名、非静态方法时才是访问子类的。

编译看左边，运行不一定看右边。

编译看左边：java编译器在编译的时候，会检查引用类型变量所属的类是否具备指定的成员，如果不具备马上编译报错。

多态的应用：

1. 多态用于形参类型的时候，可以接收更多类型的数据 。

2. 多态用于返回值类型的时候，可以返回更多类型的数据。

多态的好处： 提高了代码的拓展性。

## instanceof

instanceof关键字的作用：判断一个对象是否属于指定的类别。

instanceof关键字的使用前提：判断的对象与指定的类别必须要存在继承或者实现的关系。

instanceof关键字的使用格式：

对象 instanceof 类别

instanceof关键字的作用：

做强制类型转换之前都会使用该关键字先判断一把，然后在进行转换的。

## static

静态成员变量 如果有数据需要被共享给所有对象使用时，那么就可以使用static修饰。

静态成员变量的访问方式：

方式1： 可以使用对象进行访问。

格式： 对象.变量名。

方式二： 可以使用类名进行访问。

格式： 类名.变量名;

注意：

1. 非静态的成员变量只能使用对象进行访问，不能使用类名进行访问。

2. 千万不要为了方便访问数据而使用static修饰成员变量，只有成员变量的数据是真正需要被共享的时候才使用static修饰。

static修饰成员变量的应用场景：

如果一个数据需要被所有对象共享使用的时候，这时候即可好实用static修饰。

静态方法

访问方式：

方式一：可以使用对象进行访问。

对象.静态的函数名();

方式二：可以使用类名进行访问。

类名.静态函数名字。

推荐使用是类名直接访问静态的成员。

静态的成员变量与非静态的成员变量的区别：

1. 作用上的区别：

1. 静态的成员变量的作用共享一个数据给所有的对象使用。

2. 非静态的成员变量的作用是描述一类事物的公共属性。

2. 数量与存储位置上的区别：

1. 静态成员变量是存储方法区内存中，而且只会存在一份数据。

2. 非静态的成员变量是存储在堆内存中，有n个对象就有n份数据。

3. 生命周期的区别：

1. 静态的成员变量数据是随着类的加载而存在，随着类文件的消失而消失。

2.非静态的成员数据是随着对象的创建而存在，随着对象被垃圾回收器回收而消失。

静态函数要注意的事项：

1. 静态函数是可以调用类名或者对象进行调用的，而非静态函数只能使用对象进行调用。

2. 静态方法可以直接访问静态的成员，但是不能直接访问非静态的成员。

原因：静态函数是可以使用类名直接调用的，这时候可能还没有存在对象，

而非静态的成员数据是随着对象的存在而存在的。

3. 非静态的方法是可以直接访问静态与非静态的成员。

原因：非静态函数只能由对象调用，当对象存在的时候，静态数据老早就已经存在了，而非静态数据也随着对象的创建而存在了。

4. 静态函数不能出现this或者super关键字。

原因：因为静态的函数是可以使用类名调用的，一旦使用类名调用这时候不存在对象，而this关键字是代表了一个函数 的调用者对象，这时候产生了冲突。

静态的数据的生命周期：静态的成员变量数据是优先于对象存在的。

static什么时候修饰一个函数？

如果一个函数没有直接访问到非静态的成员时，那么就可以使用static修饰了。 一般用于工具类型的方法

静态方法不能访问非静态的成员？

静态方法只要存在有对象，那么也可以访问非静态的数据。只是不能直接访问而已。

静态代码块

静态代码块是在类名.class文件加载到内存的时候就马上执行的。

就是在创建对象前执行，一般用来只创建一次对象，随着jvm启动就希望执行的代码：比如占内存大的datasource（连接池）创建对象

static{

System.out.println("静态代码块执行了...");

}

## 自动装箱与拆箱

jdk1.5新特性之-----自动装箱与自动拆箱。

java是面向对象的语言，任何事物都可以使用类进行描述，sun就使用了

一些类描述java中八种基本数据类型数据

基本数据类型 包装类型

byte Byte

short Short

int Integer

long Long

float Float

double Double

boolean Boolean

char Character

基本数据类型数据有了对应的包装类型的好处：就可以使用方法

public class Demo5 {

public static void main(String[] args) {

String str = "12";

//字符串转换成int类型数据。 可以把字符串转换成对应的数字

int i = Integer.parseInt(str);

System.out.println(i+1);

//把数字转换成字符串

System.out.println("把整数转换成对应 的字符串："+Integer.toString(i));

//把整数转换成对应的进制形式

System.out.println("10的二进制："+ Integer.toBinaryString(10));//1010

System.out.println("10的二进制："+ Integer.toOctalString(10));//12

System.out.println("10的十六进制："+ Integer.toHexString(10));//a

//可以把字符串当成对应的进行数据帮你转换

String a = "10";

int a = Integer.parseInt(a, 2);//把字符串a用2进制进行转换

System.out.println("a="+a);

//集合： 集合是可以存储任意对象类型数据的容器。1.5新特性自动装箱，所以可以存储基本类型

ArrayList list = new ArrayList();

list.add(1);

list.add(2);

list.add(3);

//自动装箱： 自动把java的基本数据类型数据转换成对象类型数据。

int temp = 10; //基本数据类型

Integer b =new Integer（temp）;//如是JDK1.4之前做法需要这样赋值

Integer b =temp; //1.5新特性自动装箱：把a存储的值赋予给b变量。

//自动拆箱： 把引用类型的数据转换成基本类型的数据

Integer c = new Integer(13);

int d = c.intValue();//JDK1.4之前的做法

int d = c; //1.5JDk新特性自动拆箱直接赋值，帮你简化

System.out.println(d);

//引用的数据类型

Integer e = 128; Integer e = 127;

Integer f = 128; Integer f = 127;

System.out.println("同一个对象吗？"+(e==f)); 如果是127：true，128：false

为什么呢：Integer类内部维护了缓冲数组，该缓冲数组存储的-128~127 这些数据在一个数组中。如果你获取的数据是在这个范围之内的，那么就直接从该缓冲区中获取对应的数据。

为什么不把它缓冲大一点呢：其实在编程里面，这里的数据使用频率是最高的，扩大浪费内存空间

## 可变参数

需求： 定义一个函数做加法功能（函数做几个数据 的加法功能是不确定）。

可变参数的格式：

数据类型... 变量名

可变参数要注意的细节：

1. 如果一个函数的形参使用上了可变参数之后，那么调用该方法的时候可以传递参数也可以不传递参数。

2. 可变参数实际上是一个数组对象。

3. 可变参数必须位于形参中的最后一个参数。

4. 一个方法最多只能有一个可变参数，因为可变参数要位于形参中最后一个位置上。

public class Demo4 {

public static void main(String[] args) {

int[] arr = {1,2,45,6,7};

/\*System.out.println(arr);

add(arr);\*/

add();//传入的是空，长度是0

}

public static void add(String name,int... arr){ //可变参数要最后一个参数

int result = 0;

for(int item : arr){

result+=item;

}

System.out.println("总和："+ result);

}

## 内部类

匿名内部类

匿名内部类：没有类名的类就称作为匿名内部类。

匿名内部类的好处：简化书写。

匿名内部类的使用前提：必须存在继承或者实现关系才能使用。

匿名内部类一般是用于实参。

匿名内部类继承关系的使用和实现接口类方式使用

匿名内部类继承关系的使用

class Animal{ //也可以抽象类abstract class Animal{

//public abstract Animal run();

//public abstract void sleep();

public Animal run(){

System.out.println("狗在跑..");

return this;

}

public void sleep(){

System.out.println("狗趴在睁开眼睛睡..");

}

}

public class C{

public void print(){

//Animal a = new Animal(){ 匿名实现父类Animal操作1，多态父类引用

new Animal(){ //匿名实现父类Animal操作2

//匿名内部的成员

public Animal run(){

System.out.println("狗在跑..");

return this; //通过this关键字返回调用者父类对象，可以在进行调用本类别的方法

}

public void sleep(){

System.out.println("狗趴在睁开眼睛睡..");

}

/\*//特有的方法 父类引用调用不了，除非直接

public void bite(){

System.out.println("狗在咬人..");

}.run().sleep();//run()方法返回父类所以可以再继续调用方法

//}.bite();//调用子类独有方法，前提不能父类引用

//a.bite();

/\*a.run();

a.sleep();\*/

}

public static void main(String[] args) {

//System.out.println("Hello World!");

C outer = new C();

outer.print();

}

}

匿名内部类实现接口类方式使用

interface Dao{ //接口类

public void add();

}

public class C{

public void print(){

//创建一个匿名内部类的对象，匿名实现Dao接口类

new Dao(){

public void add(){ //实现接口方法

System.out.println("添加成功");

}

}.add();//它已经是一个匿名对象，进行调用方法

}

public static void main(String[] args) {

C a= new C();//创建对象

a.print();//调用方法

}

}

成员内部类

内部类：一个类定义在另外一个类的内部，那么该类就称作为内部类。

内部类的class文件名： 外部类$内部类. 好处：便于区分该class文件属于哪个外部

成员内部类的访问方式：

方式一：在外部类提供一个方法创建内部类的对象进行访问。

方式二：在其他类直接创建内部类的对象。

格式：外部类.内部类 变量名 = new 外部类().new 内部类();

注意： 如果是一个静态内部类，那么在其他类创建的格式：

外部类.内部类 变量名 = new 外部类.内部类();

内部类的应用场景：

我们在描述A事物的时候，发现描述的A事物内部还存在另外一个比较复杂的事物B时候，而且这个比较复杂事物B还需要访问A事物的属性等数据，那么这时候我们就可以使用内部类描述B事物。

比如： 人--->心脏

class 人{

血，氧气等....

class 心脏{

}

}

内部类的好处：内部类可以直接访问外部类的所有成员。

内部类要注意的细节：

1. 如果外部类与内部类存在同名的成员变量时，在内部类中默认情况下是访问内部类的成员变量。

可以通过"外部类.this.成员变量名" 指定访问外部类的 成员。

2. 私有的成员内部类只能在外部类提供一个方法创建内部类的对象进行访问，不能在其他类创建对象了。

3. 成员内部类一旦出现了静态的成员，那么该类也必须使用static修饰。

局部内部类

局部内部类： 在一个类 的方法内部定义另外一个类，那么另外一个类就称作为局部内部类。

局部内部类要注意的细节：

1. 如果局部 内部类访问了一个局部变量，那么该局部变量必须使用final修饰、

## 异常

前提例子

数组越界问题：抛出异常jvm处理机制

main主方法收到这个异常，main中也没有对数组异常进行处理，继续抛出：抛给了main的调用者JVM，JVM收到异常后，做了两件事情：将异常信息，以红色字体输出控制台，2将程序停止，结束

public static void main(String[] args){

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

int i =getArray(arr); //java.lang.ArrayIndexOutOfBoundsException: 3

System.out.println("出现越界问题不会进入此语句");

System.out.println(i);

}

由于没找到3索引，导致运行时发生了异常。这个异常JVM认识（java内置了很多常用的异常类）：ArrayIndexOutOfBoundsException（本身有描述：异常的名称、异常的内容、异常的产生位置）java将这些信息直接封装到异常对象中抛给调用者main

public static int getArray(int[] arr) {

int i= arr[3]; //arr[3]没有这个索引，此时JVM会创建对象内置的异常类会主动抛出越界异常，将异常的对象进行抛出给调用者

return i+1; //此句不会执行

}

数组NULL问题：try|catch的处理机制 null异常一帮不用处理的

public static int getArray(int[] arr)throws NullPointerException,ArrayIndexOutOfBoundsException{

//对数组判空

if( arr == null){

//手动抛出异常,抛出空指针异常当然你不手动抛出虚拟机也会主动抛出的，

throw new NullPointerException("数组不存在");//第一步：手动抛出异常传入了有参的构造函数

}

return arr[3]+1;

}

public static void main(String[] args) {

int[] arr = null;

try{

int i = getArray(arr);

System.out.println(i);//此语句不会输出

}catch(NullPointerException ex){ 异常对象 对象变量名

System.out.println("###"+ex);//第二步：捕捉异常并接收手动抛出信息‘数组不存在’

}

System.out.println("Game Over"); //进行了处理后续的代码会进行执行

}

异常体系：

--------| Throwable 所有错误或者异常的父类

--------------| Error（错误）

--------------| Exception(异常) 异常一般都通过代码处理

------------------| 运行时异常: 如果一个方法内部抛出了一个运行时异常，那么方法上 可以声明也可以不 声明，调用者可以以处理也可以不处理。

------------------| 编译时异常(非运行时异常、受检异常): 如果一个方法内部抛出了一个编译时异常对象，那么方法上就必须要声明，而且调用者也必须要处理。

运行时异常： RuntimeException以及RuntimeException子类 都是属于运行时异常。

编译时异常： 除了运行时异常就是编译异常。

疑问： 为什么java编译器会如此严格要求编译时异常，对运行时异常如此宽松？

运行时异常都是可以通过程序员良好的编程习惯去避免，所以java编译器就没有严格要求处理运行时异常。

疑问： 下面的信息是通过printStackTrace方法打印出来，那么异常对象从何而来呢？

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Demo10.div(Demo10.java:10)

at Demo10.main(Demo10.java:5)

jvm运行到a/b这个语句的时候，发现b为0，除数为0在我们现实生活中是属于

不正常的情况，jvm一旦发现了这种不正常的情况时候，那么jvm就会马上创建

一个对应的异常对象，并且会调用这个异常对象 的printStackTrace的方法来处理。

Throwable 是一个正常类 可以new

Throwable常用的方法：

getMessage() 返回有参String构造方法中的信息

toString() 返回当前异常对象的完整类名+如有getMessage()返回来的信息。

printStackTrace() 打印异常的栈信息+当前异常对象的完整类名+如有getMessage()返回来的信息。

java.lang.Throwable //全路径名

at Demo8.text<Demo8.java:36> //方法名和行数

at Demo8.main<Demo8.java:30>

例子 模拟

class Demo8 {

public static void main(String[] args) {

/\*

//创建了一个Throwable对象。

Throwable t = new Throwable("头晕，感冒..");

String info = t.toString();

String message = t.getMessage();

System.out.println("toString: "+ info); // java.lang.Throwable 包名+类名 = 完整类名

System.out.println("message: "+ message);

\*/

test();

}

public static void test(){

//

Throwable t = new Throwable();

t.printStackTrace();

}

}

错误(error)

----------| Throwable 所以异常或者错误类的超类

--------------|Error 错误 错误一般是用于jvm或者是硬件引发的问题，所以我们一般不会通过代码去处理错误的。

--------------|Exception 异常 是需要通过代码去处理的。

何为错误:

1、在dos进行javac Demo.java后运行java demo，名字搞错的时候出现错误：

java.lang.NoClassDefFoundError:demo:意思是找不到此文件错误

2、内存溢出错误:java.lang.OutOfMemoryError:

class Demo9{

public static void main(String[] args) {

//java虚拟机在默认的情况下只能管理64m内存。

byte[] buf = new byte[1024\*1024\*1024];//现在是1G内存

System.out.println("Hello World!");

}

}

进行运行的实收出现错误异常：

如何区分错误与异常呢：

1、如果程序出现了不正常的信息，如果不正常的信息的类名是以Error结尾的，那么肯定是一个错误。还有一个特殊的ThreadDeath：调用 Thread 类中带有零参数的 stop 方法时，受害线程将抛出一个 ThreadDeath 实例。

2、如果是以Exception结尾的，那么肯定就是一个异常。

try|catch|finally

捕获处理的格式：

try{

可能发生异常的代码；

c = a/b; //jvm在这句话的时候发现了不正常的情况，那么就会创建一个对应的异常对象。然后跳转到catch捕获的对应异常下面的代码就不会执行了

int b=6;

}catch(捕获的异常类型 变量名){

处理异常的代码....

}

捕获处理要注意的细节：

try块中：

如果try块中出了异常的代码，c=a/b是异常的代码，就不会往下执行，会跳到catch创建对应异常对象中，执行一个try块后面是可以跟有多个catch块的，也就是一个try块可以捕获多种异常的类型。

一个try块可以捕获多种异常的类型,不能先Exception父类先捕获，要以小到大

try块外：

如果try块中代码出了异常经过了处理之后，那么try-catch块外面的代码可以正常执行。

以后捕获处理 的时候是否就是捕获Exception即可？

错的，因为我们在现实开发中遇到不同的异常类型的时候，我往往会有不同 的处理方式。所以要分开不同的异常类型处理。

finally块:

finally块的 使用前提是必须要存在try块才能使用。

finally块的代码在任何情况下都会执行的，除了jvm退出的情况。

finally非常适合做资源释放的工作，这样子可以保证资源文件在任何情况下都 会被释放。

try块的三种组合方式：

第一种： 比较适用于有异常要处理，但是没有资源要释放的。

try{

可能发生异常的代码

}catch(捕获的异常类型 变量名){

处理异常的代码

}

第二种：比较适用于既有异常要处理又要释放资源的代码。

try{

可能发生异常的代码

}catch(捕获的异常类型 变量名){

处理异常的代码

}finally{

释放资源的代码;

}

第三种： 比较适用于内部抛出的是运行时异常，并且有资源要被释放。

try{

可能发生异常的代码

}finally{

释放资源的代码;

}

除法运算异常：它属于运行时异常，没有进行处理会创建异常对象，

系统自行使用方法：printStackTrace进行再控制台打印

class Demo10 {

public static void main(String[] args) {

int[] arr = null;

div(4,0,arr);

}

public static void div(int a , int b,int[] arr){

int c = 0;

try{

c = a/b; //jvm在这句话的时候发现了不正常的情况，那么就会创建一个对应的异常对象。

System.out.println("数组的长度："+ arr.length);

}catch(ArithmeticException e){

//处理异常的代码....

System.out.println("异常处理了....");

System.out.println("toString:"+ e.toString());

}catch(NullPointerException e){

System.out.println("出现了空指针异常....");

}catch(Exception e){

System.out.println("我是急诊室，包治百病！");

}

System.out.println("c="+c);

}

}

throw|throws

抛出处理（throw throws）

throw简单理解就是自己主动抛出一个异常对象（不管它有没有错误）：因为代码出现异常了jvm会主动给你抛出这个异常对象再调用方法printStackTrace()打印到控制台

抛出处理要注意的细节：这里指的都是编译时异常

1. 如果一个方法的内部抛出了一个编译时异常(非运行时异常、受检异常）对象，那么必须要在方法上声明抛出,但是运行时异常RuntimeException可以不用也可以用

2. 如果调用了一个声明抛出编译时异常(非运行时异常、受检异常）的方法，那么调用者必须要处理异常。

3. 如果一个方法内部抛出了一个异常对象，那么throw语句后面的代码都不会再执行了（一个方法遇到了throw关键字，该方法也会马上停止执行的）。

4. 在一种情况下，只能抛出一种类型异常对象。方法块中不能使用两个throw，使用if判断可以

throw 与throws两个关键字：

1. throw关键字是用于方法内部的，throws是用于方法声声明上的。

2. throw关键字是用于方法内部抛出一个异常对象的，throws关键字是用于在方法声明上声明抛出异常类型的。

3. throw关键字后面只能有一个异常对象，throws后面一次可以声明抛出多种类型的 异常。

疑问：何时使用抛出处理？何时捕获处理？原则是如何？

如果你需要通知到调用者，你代码出了问题，那么这时候就使用抛出处理.

如果代码是直接与用户打交道遇到了异常千万不要再抛，再抛的话，就给了用户了。

这时候就应该使用捕获处理。

class Demo11 {

public static void main(String[] args) {

try{

int[] arr = null;

div(4,0,arr); //调用了一个 声明抛出异常类型 的方法

}catch(Exception e){

System.out.println("出现异常了...");

e.printStackTrace();

}

}

public static void div(int a, int b,int[] arr) throws Exception,NullPointerException {

if(b==0){

throw new Exception(); //抛出一个异常对象...

}else if(arr==null){

throw new NullPointerException();

}

int c = a/b;

System.out.println("c="+c);

}

}

## ASCII编码

为什么会乱码？因为每个码表设置的字节都不一样，好比gpk中文进行Iso8859-1进行转换（它的字节都是一位，又不支持中文），所以会造成乱码

ASCII： 美国标准信息交换码。用一个字节的7位可以表示，都是一个字节

ISO8859-1： 拉丁码表。欧洲码表，用一个字节的8位表示。跟ascii基础上增加了，从而支持了欧洲一些国家

GB2312： 英文占一个字节， 中文占两个字节。

GBK： 英文占一个字节， 中文占两个字节。

Unicode： 国际标准码，。所有文字都用两个字节来表示，

UTF-8： 英文占一个字节，中文占三个字节。 最多用三个字节来表示一个字符。

（我们以后接触最多的是iso8859-1、gbk、utf-8）

UTF-16: 不管英文中文都是占两个字节。

编码与解码

编码： 把看得懂的字符变成看不懂码值这个过程我们称作为编码。字符---数字

解码： 把码值查找对应的字符，我们把这个过程称作为解码。数字----字符

注意： 以后编码与解码一般我们都使用统一的码表。否则非常容易出乱码。

unicode实际上就是用了utf-16.它不管字母还是中文统一都是2个字节，它有一个恶心之处，就是打印出来的都有-1和-2进行开头

public class Demo7 {

public static void main(String[] args) throws Exception {

/\*

String str = "中国";

byte[] buf = str.getBytes("utf-8");// 平台默认的编码表是gbk编码表。 编码

System.out.println("数组的元素："+Arrays.toString(buf)); //

str = new String(buf,"utf-8"); //默认使用了gbk码表去解码。

System.out.println("解码后的字符串："+ str);

\*/

/\*String str = "a中国"; // ,0, 97, 78, 45, 86, -3

byte[] buf = str.getBytes("unicode"); //编码与解码的时候指定 的码表是unicode实际上就是用了utf-16.

System.out.println("数组的内容："+ Arrays.toString(buf));

\*/

String str = "大家好";

byte[] buf = str.getBytes(); //使用gbk进行编码

System.out.println("字节数组："+ Arrays.toString(buf)); // -76, -13, -68, -46, -70, -61

str = new String(buf,"iso8859-1");

// 出现乱码之后都可以被还原吗？ 出现乱码不一定能还原

byte[] buf2 = str.getBytes("iso8859-1");

str = new String(buf2,"gbk");

System.out.println(str);

## 反射

有栈内存 有堆内存，有方法区，硬盘有Demo.class 和Person.class

进行：Person p = new Person(110,"老猪头")的时候加载到方法区里面，方法区会对字节码Person.class进行解剖

看一下里面有那些成员：id\name啊，一些方法啊，一大堆很零散的数据，这些就会创建new Class()对象出来，这个对象已经包含了字节码的所有内容了

所谓的反射获取到这个对象Class()，它有哪些内容啊，设置它的成员，方法等：

反射：当一个字节码文件加载到内存的时候，jvm会对该字节码进行解剖，然后会创建一个对象的Class对象，

把字节码文件的信息全部都存储到该Class对象中，我们只要获取到Class对象，我们就可以使用字节码对象

设置对象的属性或者调用对象的方法等操作。。。。

步骤：先获取Class对象—>就可以获得构造方法、普通方法、成员变量的三个对象

1）Class clazz=Class.forName("nih.Person");

构造方法

2）Constructor[] cons=clazz.getDeclaredConstructors();

普通方法

3）Method m=clazz.getDeclaredMethod("ni",int.class);

成员变量

4）Field[] fields=clazz.getFields();

注意：在反射技术中一个类的任何成员都有对应的类进行描述。比如：成员变量（Field） 方法---->Method类

-------------------------------------------- 对于反射来说没有什么不可能的

获取Class方式

public static void main(String[] args) throws ClassNotFoundException {

//推荐使用：获取Class对象的方式一

Class clazz=Class.forName("nih.Person");

System.out.println("能获取成功吗:"+clazz);

//获取Class对象方式二

Class clazz2=Person.class;

System.out.println("是同一个吗："+(clazz==clazz2));

//获取Class对象方式三

Class clazz3=new Person().getClass();

System.out.println("是同一个吗："+(clazz3==clazz2));

//经过测试都是同一个，内存地址都一样

1. getName()类的名称（全名，全限定名）

2 getSimpleName()类的的简单名称（不带包名）

3. getModifiers(); 类的的修饰符

获取构造方法对象

创建对象操作，私有化构造方法暴力创建对象操作

通过Class对象获取对应类的所有公共构造方法,Constructor对象它是在java.lang.reflect包中，

获得公共public的构造方法

Constructor[] cons=clazz.getConstructors();//无参构造方法，可以写上null，也可以什么也不写

for(Constructor con:cons) {

System.out.println(con);

//打印：public nih.Person(),public nih.Person(int,java.lang.String)

}

暴力获得所有的构造方法包括私有化的

Constructor[] cons2=clazz.getDeclaredConstructors();

for(Constructor con:cons2) {

System.out.println(con);

}

获取单个构造方法,跟上面方法名就是少了一个S，它们参数是三个点的（可有可无可多个），参数要一一对应，进行获取

Constructor cons3=clazz.getDeclaredConstructor(int.class,String.class);

有了构造方法当然就可以创建一个对象了，返回的是Object进行强转

Person s=(Person)cons3.newInstance(1,"sdf");

s.eat();

获取私有化的构造方法,暴力反射创建对象（黑客）

Constructor cons4=clazz.getDeclaredConstructor();

//暴力反射创建对象（黑客）

cons4.setAccessible(true);//设置成权限可以访问的

Person s1=(Person)cons4.newInstance();

System.out.println(s1);

获取方法对象

获得公共public的所有方法包括父类的：（method对象）

Class clazz=Class.forName("nih.Person");

Method[] ms1=clazz.getMethods();

for(Method m:ms1) {

System.out.println(m);

}

获得所有自己的方法包括私有化的方法

Method[] ms=clazz.getDeclaredMethods();

for(Method m:ms) {

System.out.println(m);

}

Person p=(Person)constructor.newInstance();

------------------------------------------------------------------------------------------------------------------------------

参数说明：第一个是问你方法名，第2个是传入参数：因为有重载的方法，所以要传参数，没有null代替或者不写

Method m=clazz.getMethod("eat",String.class);

System.out.println(m);//打印方法名：public void nih.Person.eat(java.lang.String)

执行方法：参数说明：第一个参数要你传入对象a：A a=new A()，如果是静态方法写null也可以，;第二个根据上面获得获得Method方法的类型实体参数

Person p=(Person)cons3.newInstance(1,"sdf");

m.invoke(p,"曾丽方");//执行方法内容

Method m2=clazz.getDeclaredMethod("ni",int.class);//获得私有化等的方法，上面只能获得公共的public方法

暴力执行私有化的方法：

m2.setAccessible(true);

m2.invoke(p, 666);

获得成员变量 对象

静态变量也可以说明是类变量，对象是field

//Field[] fields=clazz.getFields();//此方法是获得public修饰符的，其它的权限修饰符包括默认不写的都是获取不了的

Field[] fields=clazz.getDeclaredFields();

for(Field field:fields) {

System.out.println(field+" ");//打印：int nih.Person.id ，java.lang.String nih.Person.name ，对象： nih.Person2 nih.Person.cc

}

//获取一个

Field f=clazz.getDeclaredField("id");

//要访问就要设置暴力放射，成员变量几乎都是private的

f.setAccessible(true);

//参数说明：第一个参数要你传入对象a：A a=new A()，第二个设置参数

//f.setInt(p, 888);

f.set(p, 666);//老师说推荐就使用这个

System.out.println(p.id);

## 增强型for循环

jdk1.5出现的新特性---->增强for循环

增强for循环的作用： 简化迭代器的书写格式。(注意：增强for循环的底层还是使用了迭代器遍历。)

增强for循环的适用范围： 如果是实现了Iterable接口的对象或者是数组对象都可以使用增强for循环。

增强for循环的格式：

for(数据类型 变量名 :遍历的目标){

}

增强for循环要注意的事项：

1. 增强for循环底层也是使用了迭代器获取的，只不过获取迭代器由jvm完成，不需要我们获取迭代器而已，所以在使用增强for循环变量元素的过程中不准使用集合对象对集合的元素个数进行修改。

2. 迭代器遍历元素与增强for循环变量元素的区别：使用迭代器遍历集合的元素时可以删除集合的元素，而增强for循环变量集合的元素时，不能调用迭代器的remove方法删除元素。

3. 普通for循环与增强for循环的区别：普通for循环可以没有变量的目标，而增强for循环一定要有变量的目标。

## 泛型

泛型是jdk1.5使用的新特性。

泛型的好处：

1. 将运行时的异常提前到编译时。

2. 避免了无谓的强制类型转换 。

泛型在集合中的常见应用：

ArrayList<String> list = new ArrayList<String>(); true 推荐使用。

//以下两种写法主要是为了兼顾新老系统的兼用性问题。

ArrayList<String> list = new ArrayList(); true

ArrayList list = new ArrayList<String>(); true

注意:泛型没有多态的概念，左右两边的数据类型必须要一致，或者只是写一边的泛型类型。

自定义泛型

自定义泛型： 自定义泛型就是一个数据类型的占位符或者是一个数据类型的变量。

在泛型中不能使用基本数据类型，如果需要使用基本数据类型，那么就使用基本数据类型对应的包装类型。

如：Int[]要更改为包装类Integer[] arry= new Integer[1024];

Int[] arry= new Int[1024];如这个在传参arry的时候会报错，接收参数不会

byte----> Byte, short---> Short , long----> Long,

double ----> Double, float -----> Float , boolean-----Boolean

int----> Integer, char----- Character

方法泛型的使用

需求： 定义一个方法可以接收任意类型的参数，而且返回值类型必须要与实参的类型一致。

第一种

修饰符 <声明自定义的泛型>返回值类型 函数名(使用自定义泛型 ...){

}

Public <T>void method(T a){} //自定义泛型必须在返回值写上<T>

Public <T>String method(T[] a ){ return “”;}

第二种

修饰符 <声明自定义的泛型>使用自定义泛型 函数名(使用自定义泛型 ...){

}

Public <T>T method(T a){ return a;}

方法泛型注意的事项：

1. 在方法上自定义泛型，这个自定义泛型的具体数据类型是在调用该方法的时候传入实参时确定具体的数据类型的。

2. 自定义泛型只要符合标识符的命名规则即可, 但是自定义泛型我们一般都习惯使用一个大写字母表示。T和E，因为： T Type E Element

public class Demo2 {

public static void main(String[] args) {

String str = getData("abc");

Integer i = getData(123);}

public static <abc>abc getData(abc o){//可以这样写，通常都是使用 T 或者E

return o;

}

类泛型的使用

使用了泛型类就不用在方法的返回值上声明泛型了,但是静态方法还要在方法上声明

泛型类的定义格式：

class 类名<声明自定义泛型>{

}

泛型类要注意的事项：

1. 在类上自定义泛型的具体数据类型是在使用该类的时候创建对象时候确定的。

2. 如果一个类在类上已经声明了自定义泛型，如果使用该类创建对象的时候没有指定 泛型的具体数据类型，那么方法参数默认为Object类型

3.在类上自定义泛型不能作用于静态的方法，如果静态的方法需要使用自定义泛型，那么需要在方法上自己声明使用。这是由于类是在创建对象才确定类型的，静态不用创建对象

class MyArrays<T>{ //类中声明泛型

public void reverse(T[] arr){ //方法中就不用申明泛型了，在创建对象没有使用泛型，参数会变成Object[] arr

for(int startIndex = 0, endIndex = arr.length-1 ; startIndex<endIndex ; startIndex++,endIndex--){

T temp = arr[startIndex]; //泛型的使用

arr[startIndex] = arr[endIndex];

arr[endIndex] = temp;

}

}

// toString的重写

public String toString(T[] arr){ //方法中就不用申明泛型了,创建对象没有声明泛型类型，参数会变成Object[] arr

StringBuilder sb = new StringBuilder();

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

if(i==0){

sb.append("["+arr[i]+",");

}else if(i==arr.length-1){

sb.append(arr[i]+"]");

}else{

sb.append(arr[i]+",");

}

}

return sb.toString();

}

public static <T>void print(T[] t){ //静态方法必须要声明泛型，不会冲突，好比局部变量和成员变量

public class Demo3 {

public static void main(String[] args) {

Integer[] arr = {10,12,14,19};

MyArrays<Integer> tool = new MyArrays<Integer>(); //创建对象使用泛型

tool.reverse(arr);

System.out.println("数组的元素："+tool.toString(arr));

MyArrays<String> tool2 = new MyArrays<String>();

String[] arr2 = {"aaa","bbb","ccc"};

tool2.reverse(arr2);

MyArrays tool2 = new MyArrays (); 创建对象没有使用泛型，类中的方法都变成Object参数

}

接口泛型的使用 是和类一样的

泛型接口的定义格式:

interface 接口名<声明自定义泛型>{

}

泛型接口要注意的事项：

1. 接口上自定义的泛型的具体数据类型是在实现一个接口的时候指定的。

2. 在接口上自定义的泛型如果在实现接口的时候没有指定具体的数据类型，那么默认为Object类型。

需求： 目前我实现一个接口的时候，我还不明确我目前要操作的数据类型，我要等待创建接口实现类 对象的时候我才能指定泛型的具体数据类型。

如果要延长接口自定义泛型 的具体数据类型，那么格式如下：

格式：

public class Demo4<T> implements Dao<T>{

}

interface Dao<T>{ //接口类泛型使用

public void add(T t);

}

public class Demo4<T> implements Dao<T> {//延长泛型接口类型指定操作

public static void main(String[] args) {

Demo4<String> d = new Demo4<String>();//创建对象进行指定类型

}

public void add(T t){

}

泛型的上下限

有时候泛型定义太广泛了，什么都可以传入类型，我要进行限制类型的范围就使用

泛型中通配符： ？

<? super Integer >:存储本类和父类 泛型的下限

<? extends Number> ： 存储本类和子类，泛型上限

我只要Number和integer的类型，其它不要

public class Demo5 {

public static void main(String[] args) {

ArrayList<Integer> list1 = new ArrayList<Integer>();

ArrayList<Number> list2 = new ArrayList<Number>();

HashSet<String> set = new HashSet<String>();

getData(set);

}

//泛型的上限

public static void getData(Collection<? extends Number> c){

}

//泛型的下限

public static void print(Collection<? super Integer> c){

}

## 递归

递归：自己调用自己

递归的使用前提： 必须要有条件的情况下调用。没条件死循环；

需求： 算出5的阶乘。 5! = 5\*4 \*3 \* 2\*1

public class Demo9 {

public static void main(String[] args) {

int result = print(5);

System.out.println("结果："+ result);

}

//递归方式进行操作

public static int print(int num){

if(num==1){

return 1;

}else{

return num\*print(num-1);

}

}

//不用递归方式进行操作

public static int test(int num){

int result = 1;

while(num>0){

result = result\*num;

num--;

}

return result;

}

}

# 设计模式

代理模式

原理

代理模式就是多一个代理类出来，

\*替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？

\*因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，

\*我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法

\*代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

interface AInterfaces { 接口类

public void method();

}

--------------------------- A类实现了 AInterfaces接口------------------------------------

class A implements AInterfaces {

public void method() {

System.out.println("the original method!");

}

}

-------------------------- B类也实现AInterfaces接口 ------------------------------------------

class B implements AInterfaces {

private A a; 也可以接口类AInterfaces作为变量：通用

public B(){

this.a = new A();这是写死的，一般都是设置变量，在调用中传参

}

public void method() {

proxy2();自身方法进行先使用 proxy2()

A.method(); 调用A中的方法

proxy1();

}

private void proxy1() {

System.out.println("after proxy!");

}

private void proxy2() {

System.out.println("before proxy!");

}

------------------------调用-----------------------------------------

public class Main {

public static void main(String[] args) {

B b = new Proxy();

b.method();

}

}

prox动态代理

不用匿名内部类的操作

interface AInterfaces {

public String MyName();

public void MyName2()；

}

-----------------------前提被代理A类是一个接口实现类------------------------------------

public class A implements AInterfaces {

public String MyName() {

return "打印第1句话";

}

public void MyName2() {

System.out.println("打印第2句话");

}

--------------------------B类实现InvocationHandler-此接口有-- invoke 方法--------------------------------

public class B implements InvocationHandler{

private Object o;

public Object instance(Object o) {

this.o=o;//重要这步是把传入的对象传到成员变量，让下面方法要用到，因为调用操作是调用此方法的方法

//参数：目标对象A的类装载、获取被代理的所有接口，代理对象

return Proxy.newProxyInstance(o.getClass().getClassLoader(), o.getClass().getInterfaces(), this);

}

//代理对象、当前调用的方法，当前传入的参数，

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

Object result=null;

System.out.println("执行方法前");

if("MyName".equals(method.getName())) {

System.out.println("MyName被我改造了");

return "真的啊";//必须要有一个返回值，没有就null代替

}else {

result = method.invoke(o, args);//可以说获得返回值

return result;

}

}

-----------------------------调用操作------------------------------------------

public static void main(String[] args) {

B b=new B();

Object obj=b.instance(new A());//获得被改造后的被代理对象

//你都知道传入的b就是一个Ainterfaces接口类，

AInterfaces a=(AInterfaces) obj;

String c=a.MyName();

System.out.println(c);

a.MyName2();

打印：执行方法前

MyName被我改造了

真的啊

执行方法前

打印第2句话

}

匿名内部类操作

public interface AInterfaces {

public void MyName();

public void MyName2();

}

-----------------------前提被代理A类要是接口实现类------------------------------------

public class A implements AInterfaces {

public void MyName() {

System.out.println("打印第1句话");

}

public void MyName2() {

System.out.println("打印第2句话");

}

-----------------------------调用操作------------------------------------------

public static void main(String[] args) {

final A a=new A();

Object ob=Proxy.newProxyInstance(A.class.getClassLoader(), A.class.getInterfaces(), new InvocationHandler() {

//代理对象、当前调用的方法，当前传入的方法，

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

System.out.println("我先执行了");

//改造A类中的MyName方法

if("MyName".equals(method.getName())) {

System.out.println("我要改造MyName方法");

return null;//必须要有返回值：没有返回值返回一个null；底层会判断

}else {

//不想改造的方法调被代理的方法，你有什么返回值我给你return什么返回值

return method.invoke(b, args);

}

}

});

AInterfaces a=(AInterfaces) ob;/

//每调用一次方法都会执行invoke方法

a.MyName();//打印：我先执行了，然后打印：我要改造MyName方法

//每调用一次方法都会执行invoke方法

a.MyName2();//打印：我先执行了，然后打印：第2句话

}

}

-------------------------------------------------经典的invoke方法 -------------------------

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

if (method.getName().matches("^(remove|persist|merge)\\w\*$")){

System.out.println("检测到事务方法,开启事务");

}

if (proxy instanceof String){

System.out.println("这是个SQL字符串而不是个结果集 动态执行它");

}

Object o = method.invoke(Class.forName(getName()).newInstance(), args);//这里做你原来要做的事

System.out.println("得到原来的返回值:"+o);

if (method.getReturnType() == ResultSet.class ){

System.out.println("检测到查询方法,将结果装箱");

}

return "改变返回的值";

}

-----------------------------调用操作------------------------------------------

public class Main {

public static void main(String[] args) {

B b=new B();

AInterfaces a=(AInterfaces) b.instance(new A());

a.MyName();

}

装饰模式

装饰模式和代理对象很像：也需要一个接口，创建类实现同样的接口，改造自己想要改造的方法，不想改造的调用回被装饰类的方法（当然前提要进行new被装饰的对象）

缺点：接口如果方法很多方法的话，每一个方法都需要进行调用操作累死你；

public interface AInterfaces {

public void MyName();

public void MyName2();

}

-----------------被装饰的类：A-----------------------------------

public class A implements AInterfaces {

public void MyName() {

System.out.println("打印第1句话");

}

public void MyName2() {

System.out.println("打印第2句话");

}

}

--------------------------创建B类实现A类同样的接口---------------------------------

public class B implements AInterfaces{

private AInterfaces aInterfaces;

//通过构造方法进行传入接口类，当然也可以传入被改造类，这为了通用才传接口类

public B(AInterfaces aInterfaces) {

this.aInterfaces=aInterfaces;

}

public void MyName() {

System.out.println("我要改造你");

}

//不想改造的方法：在方法中调A类的方法

public void MyName2() {

aInterfaces.MyName2(); //调A类的方法

}

--------------------------调用---------------------------------

public static void main(String[] args) {

AInterfaces a= new B(new A());

a.MyName();

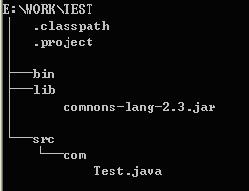
a.MyName2();

}

# DOS环境下导包操作

DOS环境下含包并引用第三方jar的java程序的编译及运行

1、程序目录机构

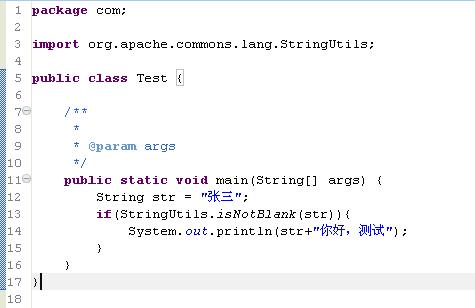


bin：class文件生成目录

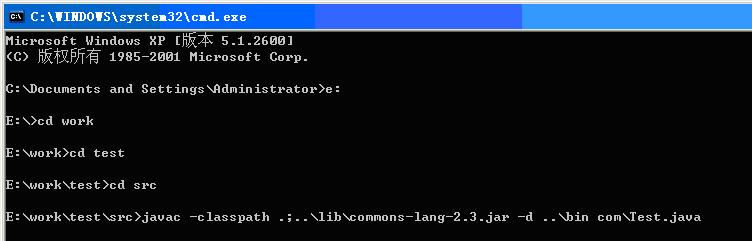
lib：第三方jar包目录

src：源程序文件目录

2、程序代码：

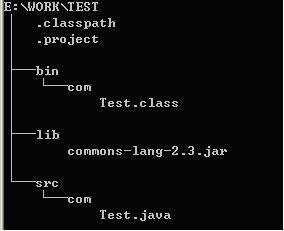


3、程序编译

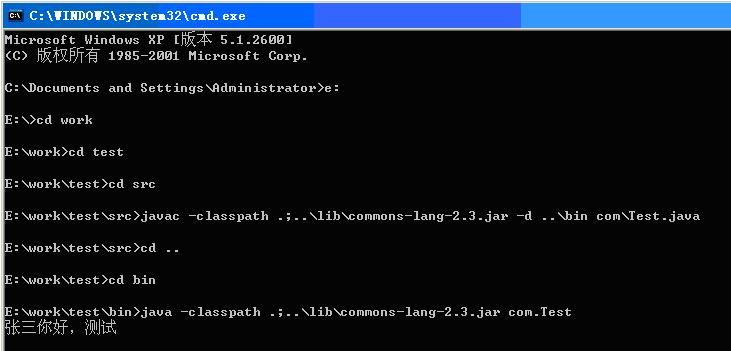


javac –classpath .;..\lib\commons-lang-2.3.jar -d ..\bin com\Test.java

4、编译后的程序目录：



5、运行程序及结果显示



java -classpath .;..\lib\commons-lang-2.3.jar com.Test

引用多个jar包

假设有个程序的启动方法在test.java里

运行Java程序是java test，但是如果要引用其他jar包，网上大多数的方法都是java -classpath xx.jar test

但是jar很多的话，就要一个一个写上去，非常麻烦。而且jar包可能经常变

现在用以下一句就能解决问题，假设jar包都放在lib文件夹里

java -Djava.ext.dirs=lib test

注意：要JDK1.6以上才可以，如果不怕麻烦也可以这样写：

javac -classpath c:\lib\log4j.jar;D:\lib\logging.jar src\org\myproject\\*.java

还可以指定输出目录，如输出到bin（bin目录要预告建立好）

javac -classpath c:\lib\log4j.jar;D:\lib\logging.jar -d bin src\org\myproject\\*.java

# 路径专题

网页

网页超链接

　　超链接有三种书写路径的方式

　　　　1）绝对路径

　　　　2）以"/"开头的相对路径

　　　　3）不以"/"开头的相对路径

1.1绝对地址

完整的URL

<a href="http://localhost:8080/javaee/jsp/b.jsp">这是绝对地址超链接</a><br/>

1.2以"/"开头的相对地址

/代表了整个web项目，即:http://localhost:8080/

<a href="/javaee/jsp/b.jsp">这是以"/"开头的相对地址超链接</a><br/>

1.3.不以"/"开头的相对地址

不以/开头，则相对于当前资源的路径

当前资源的路径为:http://localhost:8080/javaee/jsp/

而b.jsp也在此路径下

所以直接书写b.jsp

<a href="b.jsp">这是不以"/"开头的相对地址超链接</a><br/>

1.4分析：

　　1）绝对路径（以协议开头的路径）：最终请求路径为：http://localhost:8080/javaee/jsp/b.jsp

　　2)相对路径：

　　　　2.1) 相对于主机的路径（以“/”开头）：相对于当前主机（可以简单理解为ip地址，如果想深入了解请研究tomcat的Server.xml文件。这里是localhost）的路径，请求的最终路径为：http://localhost:8080/javaee/jsp/b.jsp

2.2)相对于请求的路径（不以“/”开头）：相对于当前请求（浏览器的请求）的路径。

2、表单

　　表单有三种书写路径的方式

　　　　1）绝对路径

　　　　2）以"/"开头的相对路径

　　　　3）不以"/"开头的相对路径

2.1绝对路径

<form action="http://localhost:8080/javaee/jsp/b.jsp" methoe="get">

username:<input type="text" name="username" value="">

<input type="submit" value="提交---绝对地址 ">

</form>

2.2以/开头的相对地址

此时的/代表整个web项目，即:http://localhost:8080/

<form action="/javaee/jsp/b.jsp" methoe="get">

username:<input type="text" name="username" value="">

<input type="submit" value="提交---以/开头的相对地址">

</form>

2.3不以"/"开头的相对路径

<form action="b.jsp" methoe="get">

username:<input type="text" name="username" value="">

<input type="submit" value="提交---不以/开头的相对地址 ">

</form>

3、表单提交到Servlet

表单提交到Servlet有三种书写方式

1，绝对路径

2，以"/"开头的相对地址

3，不以"/"开头的相对地址

3.1绝对地址

完整的URL

<form action="http://localhost:8080/javaee/PathServlet" methoe="get">

username:<input type="text" name="username" value="">

<input type="submit" value="表单提交到Servlet---绝对地址">

</form>

3.2以/开头的相对地址

此时的/代表整个web项目，即:http://localhost:8080/

<form action="/javaee/PathServlet" methoe="get">

username:<input type="text" name="username" value="">

<input type="submit" value="表单提交到Servlet---以/开头的相对地址">

</form>

3.3不以/开头的相对地址

不以/开头的相对路径是相对于当前资源的路径

此时form.jsp的地址为：http://localhost:8080/javaee/jsp/form.jsp

所以当前资源路径为:http://localhost:8080/javaee/jsp

而要提交的Servlet的路径为Http://localhost:8080/javaee/PathServlet

所以路径为当前路径的上一级路径下

即路径为:../PathServlet

注：.代表当前路径

..代表当前路径的上一级路径

<form action="../PathServlet" methoe="get">

username:<input type="text" name="username" value="">

<input type="submit" value="表单提交到Servlet---不以/开头的相对地址">

</form>

</body>

4、重定向

　　表单有三种书写路径的方式

　　　　1）绝对路径

　　　　2）以"/"开头的相对路径

　　　　3）不以"/"开头的相对路径

注意：在这里中我们一般只会用相对路径，不会去用绝对路径

response.sendRedirect("http://localhost:8080/javaee/jsp/b.jsp");

4.1.以"/"开头的相对路径

此时，/代表整个web工程的路径，即http://localhost:8080/

response.sendRedirect("/javaee/jsp/b.jsp");

4.2.不以"/"开头的相对路径

此时是相对于当前资源的相对路径

当前资源路径为:http://localhost:8080/javaee/RedirectServlet

即表示:RedirectServlet在路径http://localhost:8080/javaee之下

而b.jsp在http://localhost:8080/javaee/jsp/b.jsp

所以最终地址写为:jsp/b.jsp

response.sendRedirect("jsp/b.jsp");

5、服务端路径：请求转发、请求包含路径方式一样的

服务端路径是指在服务器上面运行的请求，比如请求转发（常用）、请求包含等。服务端的路径有两种：相对于当前应用的路径和相对于当前请求的路径。

　　请求转发有两种书写路径的方式

　　　　1）以"/"开头的相对路径

　　　　2）不以"/"开头的相对路径

　　注意：服务器端的路径不能是绝对路径，只能是相对路径，也分为以/开头和不以/开头两种

1.以"/"开头的相对路径

此时，/代表当前web项目，即:http://localhost:8080/javaee

request.getRequestDispatcher("/jsp/b.jsp").forward(request, response);

2.不以"/"开头的相对路径

相对于当前资源的相对路径

此时，当前资源的路径为:http://localhost:8080/javaee/DispatcherServlet

所以要转发去的资源的路径以:http://localhost:8080/javaee开头

request.getRequestDispatcher("jsp/b.jsp").forward(request, response);

注意：所有Web层框架的底层使用的都是Web的基础Filter（Struts）或Servlet（SpringMVC），请求都是request，响应都是response，所以各个Web层框架的转发或重定向底层都是利用request和response进行的。

资源获取路径

　　获取资源的路径主要有3种，分别是ServletContext、Class和ClassLoader。其中ServletContext是WEB阶段的，Tomcat提供的一种获取资源的方式；Class和ClassLoader获取资源主要是JavaAPI提供的一种获取流的方式，

　　由于这是JDK提供的，所以不仅局限于Web，在普通Java类中也可以使用，主要用于获取src目录及其子目录下的文件流。

ServletContext

　　这里ServletContext获取资源的路径是相对系统的绝对路径（在Windows中是带盘符的，可以用来获取上传或下载文件的具体路径）。

　　基本语法：servletContext.getRealPath("路径");参数中的路径必须是相对路径，可以“/”开头，也可以不使用“/”开头，但无论是否使用“/”开头都是相对当前应用路径，建议以"/"开头（这样可以尽量统一）。

另外获取ServletContext的方法如下：

　　　　1）使用request获取： request.getSession().getServletContext();

　　　　2)在Servlet中获取：this.getServletContext()；

　　　　3)使用FilterConfig对象获取（在Filter中使用）：config.getServletContext();

　　 ServletContext获取资源必须是相对路径，不能是绝对路径，但不管是以/开头，还是不以/开头， 都是相对于当前资源的相对路径 。

String path1 = this.getServletContext().getRealPath("/a.properties");

String path2 = this.getServletContext().getRealPath("a.properties");

System.out.println(path1);

System.out.println(path2);

//输出的地址一样

ClassLoader获取资源

　　和Class获取资源类似。只是不同的写法而已，也是用于获取文件流的。

　　用法：classLoader.getResourceAsStream("路径")。参数中的路径可以以“/”开头，也可以不以“/”开头（建议）。但带不带“/”的都表示相对于当前类的路径。

　　ClassLoader类加载器不能通过绝对地址来加载资源，只能通过相对地址来加载资源 但相对地址不管前面加不加/都是相当于类路径的相对地址

加了/,其地址是相对于类路径的相对地址

InputStream in =

this.getClass().getClassLoader().getResourceAsStream("/cn/ccnu/classloaderpath/c.properties");

Properties prop = new Properties();

prop.load(in);

System.out.println(prop.getProperty("url"));

不加/,其地址是相对于类路径的相对地址

InputStream in =

this.getClass().getClassLoader().getResourceAsStream("cn/ccnu/classloaderpath/c.properties");

Properties prop = new Properties();

prop.load(in);

System.out.println(prop.getProperty("url"));

总结:不能使用绝对地址，而只能只用相对地址

且不管加不加/的相对地址，都是相对于类路径的相对地址

Class获取资源

　　Class获取资源主要是用作自己写的配置文件，用来读取内容。

　　用法：clazz.getResourceAsStream("路径")。参数中的路径可以以“/”开头，也可以不以“/”开头。其中带“/”的表示相对于当前类的路径，不以“/”开头表示相对于当前class所在目录的路径。

　　Class读取资源不能是绝对路径，只能是相对路径，又分为以/开头或者是不以/开头 。

1.以/开头的相对路径

此时的/代表类路径，即:/javaee/WEB-INF/classes

InputStream in =

ClassServlet.class.getResourceAsStream("/cn/ccnu/classpath/b.properties");

Properties porp = new Properties();

porp.load(in);

System.out.println(porp.getProperty("url"));

2.不以/开头的相对路径

此时相对的是：类ClassServlet.class的路径，

即:\javaee\WEB-INF\classes\cn\ccnu\classpath

即:/javaee/WEB-INF/classes/cn/ccnu/classpath

InputStream in = ClassServlet.class.getResourceAsStream("b.properties");

Properties porp = new Properties();

porp.load(in);

System.out.println(porp.getProperty("url"));

<url-pattern>路径

这里的路径仅仅是指web.xml文件中的标签内容。这个路径是虚拟的路径，只有相对路径（相对于当前应用的路径），但相对路径的写法可以是精确查询和模糊查询。

　　要么以“\*”开关，要么为“/”开头,当通常情况看下，我们都会以"/"开头。

<!—精确查询，是相对于当前应用-->

<url-pattern>/servlet/testPath</url-pattern>

<!—模糊查询，表示匹配servlet目录下的所有文件或请求。/\*表示匹配所有-->

<url-pattern>/servlet/\*</url-pattern>

<!—模糊查询，表示匹配所有后缀名为do的文件或请求-->

<url-pattern>\*.do</url-pattern>

总结

1）客户端是带“/”相对当前主机。

2）服务端(包括上述的服务端、url-pattern路径和ServletContext路径)带不带“/”都是相对当前应用(建议带上)。

3）Class带“/”是相对当前类路径。不带“/”是相对于当前位置。

4）ClassLoader无论带不带“/”都是当前类路径(建议带上)。

# 打jar包

需要使用到jdk的开发工具（jar.exe）.

jar的用法：

使用格式： jar cvf jar文件的名字 class文件或者是文件夹

第一种：直接通过class文件操作 jar cvf 新名字.jar 目标文件.class

命令：Jar cvf ww.jar a.class b.class 然后生成了：ww.jar

第二种：把上面两个文件放到qq文件夹下

命令：Jar cvf qq.jar qq 然后生成了qq.jar

通过包打开qq.jar此文件清单：

qq文件夹下有：a.class b.class a.class是主方法入口

META-INF文件夹：MANIFEST.MF

打开MANIFEST.MF：

最后面添加内容：Main-Class: c.a

注意：

1、英文冒号和c之间有空格的，少一个空格和多一个空格都不行

2、A.class中的package c;要和文件夹路径一致

打jar包要注意的事项：

1. 一个程序打完jar之后必须要在清单文件上指定入口类：格式Main-Class: 包名.类名

2. jar包双击运行仅对于图形化界面的程序起作用，对控制台的程序不起作用。

jar文件的作用：

1. 方便用户快速运行一个项目。

2. 提供工具类以jar包的形式给别人使用。

如果使用jar包里面的类必须要先设置classpath路径。

jre = jvm+ 核心类库

package qq;

import javax.swing.\*;

class Demo9 {

public static void main(String[] args) {

System.out.println("QQ程序..");

JFrame frame = new JFrame("QQ程序");

frame.setSize(400,500);

frame.setVisible(true); //设置窗口可见。

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

}

Eclipse进行打jar包：选中工程包右键Export搜索jar选中JAR file第一个路径目标文件按两次Next进入选择Main class选择主线程操作然后完成

貌似不能进行jar中的文件进行修改，可以使用jar资源文件，路径操作：

InputStream in = UserDao.class.getClassLoader().getResourceAsStream("User.txt");进行加载

# 正则表达式

类在：java.util.regex. Pattern

matches(String regex) 告知此字符串是否匹配给定的正则表达式。

String aa="45679";

System.out.println(aa.matches("[1-9]\\d{4,10}")?"合法":"不合法");

[1-9]\\d{4,10}意思是：开头只能在1到9取值，长度在5到9之间才是true

\\：双斜杠是因为单\在字符串有特殊意义，要进行转义

[1-9]：先写这个是让它开头第一个只能填1-9

\d：是正则中的预定义字符类：取值0-9，

{4,10}：是多少位

常用预定义字符类：

. 任何字符（与行结束符可能匹配也可能不匹配）

\d 数字：[0-9] ,\D非数字 [^0-9]

\w 单词字符：[a-zA-Z\_0-9] 大小写a到z还有下划线：\_和0到9，

\W 非单词字符：[^\w] 除了\w的其它都是true

注意：任何预定义字符没有加上数量词之前都只能匹配一个字符

Greedy 数量词

System.out.println("+:一次或者多次："+"1".matches("\\d+"));//返回true

System.out.println("?:一次或者零："+"".matches("\\d?"));//返回true

System.out.println("\*:零或者多次："+"123".matches("\\d\*"));//返回true

System.out.println("{3} 指定3位："+"456".matches("\\d{3}"));//返回true

System.out.println("{3,} 最少3位："+"567".matches("\\d{3,}"));//返回true

System.out.println("{3,9}指定3到9："+"123".matches("\\d{3,9}"));//返回true

范围词：

[abc] a、b 或 c（简单类）

[^abc] 任何字符，除了 a、b 或 c（否定）

[a-zA-Z] a 到 z 或 A 到 Z，两头的字母包括在内（范围）

注意：范围词里面不管内容多长，也一定要配合数量词，不然只能匹配一个字符

正则表达式主要用于操作字符串的规则，正则表达式对字符串的操作主要有以下几种应用

匹配

matches()方法：手机号码：”[1][34578]\\d{9}”:11位数，前面占两位后面配合数量词9

没使用正则表达，就比较内容

切割

字符串的Split()也可以使用正则表达式的

把空格去掉：因为它有好几个空格，使用到正则数量词：+一次或者多次

String vv="你 好 啊 哈 哈";

String[] v=vv.split(" +");//空格再配合正则的数量词：+：至少一次或多次

System.out.println(Arrays.toString(v));//[你, 好, 啊, 哈, 哈]

切割掉重叠的字

String vv="大家明明天天天玩得得得得得开心点点";

()分组：分组是为了让正则的内容被复用起来，组号是从1开始的

((A)(B(C))):第一组：(A)(B(C))，第2组：(A)，第3组：(B(C)，第4组是(C)

String[] v=vv.split("(.)\\1+");//所匹配的内容：至少有相同或者多个：如大大成立

System.out.println(Arrays.toString(v));//[大家, , 玩, 开心]

替换

replaceAll(String regex, String replacement)：参数：regex：正则表达式，2要替换的内容

老师介绍了论坛打广告，输入电话替换成\*\*\*\*\*

replaceAll(“[1][34578]\\d{9}”,“让你打广告”)

替换重复的内容

String vv="大家明明天天天玩得得得得得开心点点";

//如果需要在正则的外部引用组的内容，那么是使用"$组号"

System.out.println(vv.replaceAll("(.)\\1+", "$1"));//大家明天玩得开心点

查找

这个放一放没有看，使用到类java.util.regex. Pattern对象，在18天就单一个视频

# int和type互转

这里简单记录下两种转换方式：

第一种：

1、int与byte[]之间的转换（类似的byte short,long型）

[java] view plain copy

/\*\*

\* 将int数值转换为占四个字节的byte数组，本方法适用于(低位在前，高位在后)的顺序。 和bytesToInt（）配套使用

\* @param value

\* 要转换的int值

\* @return byte数组

\*/

public static byte[] intToBytes( int value )

{

byte[] src = new byte[4];

src[3] = (byte) ((value>>24) & 0xFF);

src[2] = (byte) ((value>>16) & 0xFF);

src[1] = (byte) ((value>>8) & 0xFF);

src[0] = (byte) (value & 0xFF);

return src;

}

/\*\*

\* 将int数值转换为占四个字节的byte数组，本方法适用于(高位在前，低位在后)的顺序。 和bytesToInt2（）配套使用

\*/

public static byte[] intToBytes2(int value)

{

byte[] src = new byte[4];

src[0] = (byte) ((value>>24) & 0xFF);

src[1] = (byte) ((value>>16)& 0xFF);

src[2] = (byte) ((value>>8)&0xFF);

src[3] = (byte) (value & 0xFF);

return src;

}

byte[]转int

[java] view plain copy

/\*\*

\* byte数组中取int数值，本方法适用于(低位在前，高位在后)的顺序，和和intToBytes（）配套使用

\*

\* @param src

\* byte数组

\* @param offset

\* 从数组的第offset位开始

\* @return int数值

\*/

public static int bytesToInt(byte[] src, int offset) {

int value;

value = (int) ((src[offset] & 0xFF)

| ((src[offset+1] & 0xFF)<<8)

| ((src[offset+2] & 0xFF)<<16)

| ((src[offset+3] & 0xFF)<<24));

return value;

}

/\*\*

\* byte数组中取int数值，本方法适用于(低位在后，高位在前)的顺序。和intToBytes2（）配套使用

\*/

public static int bytesToInt2(byte[] src, int offset) {

int value;

value = (int) ( ((src[offset] & 0xFF)<<24)

|((src[offset+1] & 0xFF)<<16)

|((src[offset+2] & 0xFF)<<8)

|(src[offset+3] & 0xFF));

return value;

}

第二种：1、int与byte[]之间的转换（类似的byte，short,long型）

[java] view plain copy

/\*\*

\* 将int数值转换为占四个字节的byte数组，本方法适用于(低位在前，高位在后)的顺序。

\* @param value

\* 要转换的int值

\* @return byte数组

\*/

public static byte[] intToBytes(int value)

{

byte[] byte\_src = new byte[4];

byte\_src[3] = (byte) ((value & 0xFF000000)>>24);

byte\_src[2] = (byte) ((value & 0x00FF0000)>>16);

byte\_src[1] = (byte) ((value & 0x0000FF00)>>8);

byte\_src[0] = (byte) ((value & 0x000000FF));

return byte\_src;

}

byte[]转int

[java] view plain copy

/\*\*

\* byte数组中取int数值，本方法适用于(低位在前，高位在后)的顺序。

\*

\* @param ary

\* byte数组

\* @param offset

\* 从数组的第offset位开始

\* @return int数值

\*/

public static int bytesToInt(byte[] ary, int offset) {

int value;

value = (int) ((ary[offset]&0xFF)

| ((ary[offset+1]<<8) & 0xFF00)

| ((ary[offset+2]<<16)& 0xFF0000)

| ((ary[offset+3]<<24) & 0xFF000000));

return value;

}

# 类加载器

JDK中提供的类加载器：

\* 引导/系统类加载器 ：Java\jre7\lib\rt.jar

\* 扩展类加载器 ：Java\jre7\lib\ext\\*.jar

\* 应用类加载器 ：自定义的类,类路径下的所有class文件.

类加载

我们编写的.java文件，jvm会将java变成.class文件，该文件要想运行，必须加载到你内存中，然后会生成一个对象Class对象

类加载器层次结构：

引导类加载器：用来加载rt.jar,最核心的jia包：全部类都来源这里

扩展类加载器: 用来加载ext目录下的所有jar包：我们用不上



应用类加载器：自己编写的类

实验：

应用类加载器：自己编写的类

ClassLoader ap=Test.class.getClassLoader();

System.out.println(ap);//打印：sun.misc.Launcher$AppClassLoader@2d8eef25

引导类加载器：

ClassLoader st=String.class.getClassLoader();

System.out.println(st);//打印null，因为使用c语言编写的类，可以这样认为打印null的加载器类是引导加载器类

扩展类加载器:在exp的jar包找到类sun.net.spi.nameservice.dns.DNSNameService测试

ClassLoader ext=DNSNameService.class.getClassLoader();

System.out.println(ext);//打印：sun.misc.Launcher$ExtClassLoader@9cboF4

全盘负责委托机制：

class文件由应用类加载器获得到,没有加载,向上一层委托向扩展类加载器委托,向上一层进行委托委托给引导类加载器.引导类加载器查看class哪些它负责,将自己负责的这个class进行加载.不是其负责的就向下传递扩展类加载器.扩展类加载器查看是否是其管理的class,如果是加载,不是就再向下到应用类加载器.

当一个类运行的时候，有可能有其它的类:

应用类加载器询问扩展类加载器：你加载过这类吗？

扩展类加载器不搭理，直接问（引导类加载器）：你加载过这些类吗？

引导类加载器：我查查，有一个是我负责的，我加载

扩展类加载器：接下来我来查，有几个是我负责，我加载，还有几个类我已经加载完成了，你可以直接使用

应用类加载器：收到了 剩下的我来