# JAVA

## java

### JDK环境变量

JAVA\_HOME

C:\Program Files\Java\jdk...

path

%JAVA\_HOME%\bin

java5以上不需要配置CLASSPATH

运行环境：

JRE\_HOME jre的bin目录上级

### 结构

public class A{

public static void main(String Args[]){

int x=10;

System.out.println("int"+x);

}

}

编辑文本保存为 A.java。

使用命令 javac A.java （编译命令时可以不区分文件名大小写）

生成文件 A.class

运行命令 java A

保存时，一定要用类名作为文件名。单个java文件可以包含多个类，但最多只能有一个public类，如果某个类为public，则该类名要作为文件名。

导出jar时，需要选择主类为应用程序主入口。

运行时java -jar \*.jar

label: ........;

continue label;

### Applet

<applet code="A.class" width="100" height="50">

import java.applet.Applet;

import java.awt.Graphics;

public class A extends Applet{

public void paint(Graphics g){

g.drawString("xxxx",20,30);

}

}

### 数据类型

包括基本数据类型和引用数据类型，引用数据类型包括class类型，枚举，数组。

对象在函数中传递时，基本类型都是值传递，引用类型传递的是引用，可以在函数中改变该对象的值，但不能改变该对象的指向。

### 基本数据类型

| 数据类型 | 名称 | 位长 | 默认值 |
| --- | --- | --- | --- |
| 布尔型 | boolean | 1 | False |
| 字节型 | byte | 8 | 0 |
| 字符型 | char | 16 | ‘\u0000’ |
| 短整型 | short | 16 | 0 |
| 整型 | int | 32 | 0 |
| 长整型 | long | 64 | 0 |
| 浮点型 | float | 32 | 0.0f |
| 双精度型 | double | 64 | 0.0 |

========整型常量

不含小数的整数值，十进制以非0开头，八进制以0开头，十六进制以0x开头。如果在数值后面加L或者l则表示64位长整型。

========浮点型常量

1. 十进制形式：.123，0.123,123.0
2. 科学计数法：123e3或123E-3。E后面必须是整数。如果后面加f或F，代表单精度，加上d或者D，表示双精度。不加后缀默认为双精度，占64位。

========字符常量

可以用字符编码表示，如A的八进制表示’\101’，十六进制’\u0041’

无法通过键盘输入的用转译符号：

‘\b’ 退格 ‘\r’ 回车

‘\n’ 换行 ‘\t’ 水平制表符

‘\’’ ‘\”’ ‘\\’

### 类型转换

a=(int)b

String str = "12.2";

double d = Double.parseDouble(str);

int i = Integer.parseInt(str);//str不能是小数

### Class

class类型必须使用new来创建对象：B b=new B();

也可以创建临时对象调用函数：new B().fun();

this可以指代当前对象

super指代父类：super();super.fun();

super(xxx) 只能在构造函数第一行调用基类的构造

#### 修饰符

[public][abstract][final] class 类名

[extends父类][implement 接口]{}

抽象类abstract不能实例化一个对象，只能被继承。如Number只能产生一个数的子类，比如Integer或者Float。

final类不能有子类。为了提高系统安全性和定义一个完全类。

接口是消息传递的通道，通过接口，消息才能传递到处理方法中进行处理。

#### 成员变量

成员变量描述了类和对象的状态，有时也称为属性、数据、域。

[public][private][protected][package]

[static][final][transient][volatile] 类型 名称

final可以定义常量

protected变量可以被声明它的类和子类以及同一个包中的类访问。如果子类在其他包，子类的对象可以访问，但子类中由父类产生的对象就不能访问。

package在声明时常省略，即没有修饰的变量为package变量。

final变量在程序运行过程中不能被改变。

transient一般在对象序列化上使用。

volatile用来防止编译器对变量进行优化。

#### 成员方法

[public][private][protected][package]

[static][final][abstract][native][synchronized]

返回值类型 方法名(参数表)[throws 异常类型]

#### 语句块

static

{ //main函数所在类，需要用static语句块，其他不需要

System.out.println("xxxxxx");

}

在类中直接写的语句块，类似于类的构造函数中执行初始化语句，程序运行初始化时执行。

#### 继承和多态

继承使用关键字extends，子类如果和父类有同名函数时，函数访问权限不能低于父类。此时将子类对象赋值给父类对象，父类调用该函数时为子类的函数。

class Base{public void fun(){System.out.println("Base");}}

class Der extends Base{public void fun(){System.out.println("Der");}}

使用时Base b1 = new Dir();

或者Dir d = new Dir();Base b2 = d;

此时调用fun函数，输出为Dir。

将基类改为接口，可以使用接口实现多态

interface Base{public void fun();}

class Der implements Base{public void fun(){System.out.println("Der");}}

instanceof 用于测试一个对象是否是一个指定类的实例

#### 抽象类

不能被实例化的类，可以有静态方法，通过类名直接调用。

abstract class B//包含抽象方法的类必须声明为abstract

{

abstract void fun();//抽象方法不能有实现，且声明为abstract

}

class C extends B

{

void fun(){}

}

#### 匿名对象

创建对象时就调用其中的函数，属于临时对象

**new** A().show();

#### 创建对象时加入代码

创建A对象时，为A类加入成员变量y，重写show方法，在其中可以直接访问外部变量z

**int** z = 1;

A a = **new** A(){**int** y =1; **void** show(){System.***out***.println(x+z+y);}};

(**new** A(){...}).show(); 可以加入代码后创建临时变量使用

如果是直接创建接口的对象，则需要该语法实现接口的函数

#### 内部类

在类A内定义的类B，依赖于A的存在，所以B中的代码可以直接访问到A中的成员。

### 数组

int[] arr； 或者int arr[];

Java在数组定义时并不为数组元素分配内存，因此[]中不用指出数组的长度。

数组作为Array类的实例，可以引用属性和方法。

arr.length直接获取长度属性

for(int x : arr){}遍历数组中每个元素

Arrays.sort(arr);//对数组进行排序，需要实现 Comparable接口

========一维数组

1. int[] arr=new int[3];
2. int[] a={1,2,3,4,5};

========多维数组

int[][] arr=new int[3][4];

int arr[][]={{0,1,2,3,4},{3,4,5},{6,7,8}};

二维数组的第二维的长度可以不相等

### 包装类型

将基本数据类型包装到类类型中，实现toString和compareTo等函数

String str = Integer.toHexString(17); //将整数转换为十六进制字符串

short等基本类型包装类为Short这类将首字母大写，char对应Character，int对应Integer。

short st = 1;

Short s = new Short(st);//赋值装箱

s = 1;//自动装箱

### 枚举

枚举是一种包含自身全局对象的特殊的类，且构造函数必须是private。

enum E

{

e1(1),

e2(2)； //两个E的全局对象，类似单例

public int x = 0; //成员变量

E(int a){x = a;}//构造函数，只能是private

}

======使用

E e = E.e1;//获取枚举对象

if(E.e1 == e)

{

String str = e.toString();

//将枚举的对象名e1转换为字符串“e1”。与之对应的是E.valueOf("e1")，通过字符串“e1”获取对象E.e1

}

========遍历

for(E temp : E.values())

{

int value = temp.x; //获取对象内部的成员变量

}

### 接口

interface I

{

void fun();//不能有权限修饰符

}

class B implements I

{

public void fun(){}//实现接口时必须是public

}

### 异常

class Ex extends Exception

{//自定义异常类

String s = "Exx";

}

class B

{

void fun(int x) throws Ex

{//可以抛出异常的函数，不能直接被调用，需要在try、catch中调用

if(10 == x)

{

throw new Ex();

}

}

public void ProcessEx()

{

try{

fun(10);

}catch(Ex e)

{

System.out.println(e.s);

}

}

}

### 包

javac A.java -d D:\

在D：下生成class，如果A.java有声明包路径，则会生成包目录

一般新建项目为一个目录P，目录下有abc目录，则为一个abc的包。

假设main函数所在类为P目录下的ABC目录，则运行时，需要在P目录同级下运行：java P.ABC.A

package P.ABC;//包声明必须是第一句，目录间用.，表示P目录下的abc目录

import P.ABC.\*; //导入ABC包中的全部内容，也可以将\*改为类名，只导入某个类。导入的内容访问权限如果是默认，则只能在同一个包下才能访问

### 导入jar

将jar放入工程目录，右键jar文件，选择Build Path-》add to build path，则可以看到reference libraries中出现新的包和类。（不出现build path菜单说明工程配置损坏）

### javaDoc

/\*\*

\*说明

\*

\* @param 参数1说明

\* @param 参数2 说明

\* @return

\*/

参数说明中不能包含括号冒号等特殊字符

## function

### Scanner

Scanner scanner = **new** Scanner(System.***in***);

String str = scanner.nextLine();//获取字符串

**char** c = scanner.nextLine().charAt(0);//获取字符

**int** a = scanner.nextInt();//获取数字

scanner.close();

### String

================编码转换

byte[] send = str.getBytes("gbk");

String不能改变自身长度和内容

赋值修改时是重新分配内存，指向了新的地址

================构造

String str = "abcdef";

String str = new String();

String str = new String(char[]);

String str = new String(String);

================操作

比较操作一般不使用==

if(str.indexOf("bc") >= 0) {//字符串或字符查找}

lastIndexOf(str) 最后一次出现的索引位置

str.startsWith(" ab")//是否以某个字符序列开始

str += "xx";//添加字符

String substr = str.substring(1,4);//从下标1截取到下标3

str = str.toUpperCase();//转换为大写

str = str.replace("ab", "xx");//字符串替换

str = str.trim();//去掉首尾的空格

char[] s = str.toCharArray();//转换为字符数组

String[] s = str.split(" ");//字符串拆分

boolean b = str.contains("xx");//是否包含字符串

### StringBuffer

StringBuffer可以改变自身内容，属于线程安全

StringBuilder适合单线程使用

StringBuffer strb = **new** StringBuffer("abc");

strb.append("def");

strb.delete(0, 1);//删除第一个字符

strb.insert(3, "xx");//在下标为3的位置插入

strb.setCharAt(1, 'z');//替换下标为1的字符

strb.reverse();//逆序

strb.replace(1, 3, "mmm");//将下标1到2的字符替换

String str = strb.toString();

strb.delete(0, strb.length());//清空数据

### System

System.exit(0);//终止当前java虚拟机

long start = System.currentTimeMillis();//返回以毫秒为单位的当前时间

System.gc(); //清理垃圾

获取当前系统的属性

**import** java.util.\*;

Properties p = System.*getProperties*();

Enumeration pNames = p.propertyNames();

**while**(pNames.hasMoreElements())

{

String key = (String)pNames.nextElement();

String value = System.*getProperty*(key);

System.*out*.println(key+" "+value);

}

#### 获取当前目录

System.getProperty("user.dir")

### Runtime

打开和关闭其他程序

Runtime tr = Runtime.getRuntime();

Process p = tr.exec("notepad.exe");

p.destroy();

### Thread

Thread()

Thread(实现runnable的对象)

Thread(实现runnable的对象，窗口名)

### 创建线程

#### 后台线程

创建的线程默认是前台线程，即主线程结束后，线程未运行完也不会结束。在start前调用函数setDaemon可以让线程变成后台线程，当主线程运行结束后也停止。

Thread th = **new** Thread(**new** T());

th.setDaemon(**true**);

th.start();

#### 继承 Thread

**class** T **extends** Thread

{

**public** **void** run(){}

}

调用：

**new** T().start();

**new** Thread(){  
 @Override  
 **public void** run() {  
   
 }  
}.start();

#### 实现Runnable接口

**class** T **implements** Runnable {

**public** **void** run(){ }

}

调用：

**new** Thread(**new** T()).start();

可以在Thread类中传入同一个对象，让多个线程共享同一个对象的内容。

**class** T **implements** Runnable {

**int** count = 10;

**public** **void** run() {

**while** (count > 0) {

Thread t = Thread.*currentThread*();//获取运行该语句的线程

System.*out*.println(t.getName() + " count:" + count--);

}

}

}

调用：

T t = **new** T();

**new** Thread(t,"window1").start();

**new** Thread(t,"window2").start();

#### 线程休眠

**try**

{

Thread.*sleep*(2000);//当前线程休眠2S

}**catch**(InterruptedException e){}

#### 线程优先级

可以设置某个线程的优先级，作为提高程序效率的一种手段。

th.setPriority(Thread.MAX\_PRIORITY);

#### 线程让步

当某个线程执行到yield函数时，转换成就绪状态，只有与当前线程优先级相同或者更高的才能获得执行机会。

Thread.yield();

#### 线程插队

当某个线程中调用其他线程的join方法后，调用的线程被阻塞，直到被调用的线程结束后它才会执行。

th.join();

#### 线程同步

多个线程操作同一对象，在该对象类的同步代码块或同步方法中操作相同资源，可以进行互斥。

第一个线程进入synchronized(lock)后占用了锁，第二个线程到synchronized(lock)时阻塞，直到第一个线程结束synchronized块内语句或者其他激活条件时第二个线程才执行synchronized(lock)后一句

同步代码块：

static Object lock = **new** Object();

**public** **void** run()

{

。。。。。

**synchronized**(lock){ 同步块 }

。。。。。

}

同步方法：

同步方法的锁就是this对象，相当于进入函数后进入synchronized(this)语句块

synchronized void fun(){}

=============多线程通信

lock.wait() 使当前线程放弃同步锁lock，并进入等待，直到其他线程进入此同步锁，并调用notify或者notifyAll方法唤醒该线程为止。

notify唤醒此同步锁上等待的第一个调用wait方法的线程。

notify和wait需要使用锁对象调用，且调用位置在该锁对象由Syncoronized触发的代码块内。

线程1，2公用Stack对象，1线程调用push()，2线程调用pop()。两个线程依靠互斥对象和notiy、wait方法交替执行

Stack stack = **new** Stack();

**new** Thread(**new** Th1(stack)).start();

**new** Thread(**new** Th2(stack)).start();

**class** Stack {

//可以不使用lock对象，直接把push和pop定义成synchronized方法，则两个方法使用进入时使用的是this作为锁

**static** Object *lock* = **new** Object();

**void** push() **throws** InterruptedException {

**synchronized**(*lock*)

{

**int** i = 0;

**while** (**true**) {

System.***out***.println("push" + i++);

Thread.*sleep*(1000);

**if** (i == 5) {

i = 0;

*lock*.notify();

*lock*.wait();

}

}

}

}

**void** pop() **throws** InterruptedException {

**synchronized** (*lock*) {

**int** i = 0;

**while** (**true**) {

System.***out***.println("pop...." + i++);

Thread.*sleep*(1000);

**if** (i == 5) {

i = 0;

*lock*.notify();

*lock*.wait();

}

}

}

}

}

### Math

Math.PI

Math.ceil(1.1)//向上取整 值为2

Math.floor(1.9)//向下取整 值为1

Math.abs(-1)//绝对值

Math.round(1.5)//四舍五入 值为2

Math.random()//生成大于等于0.1小于1的随机数

产生10个[0-100]的随机数

**import** java.util.Random;

Random r = **new** Random();

**for**(**int** x = 0; x < 10 ; ++x)

{

System.*out*.println(r.nextInt(100));

}

### 日期

//获取日期差

Calendar c = Calendar.*getInstance*();

c.set(2018, 11, 3,21,39,12);

**long** time1 = c.getTimeInMillis();

c.set(2018, 11, 3,22,49,12);

**long** time2 = c.getTimeInMillis();

**long** between\_minutes=(time2-time1)/(1000\*60);

Date dateCur = **new** Date();//获取当前时间

System.currentTimeMillis();//获取当前时间

//格式化日期

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

String strCurTime = sdf.format(dateCur);

String setTime = sdf.format(c.getTime());

Date dateParse = sdf.parse("2018-11-3 22:49:12");

### 文件操作

#### 文件

File ff = **new** File("D:\\1.txt");

ff.exists();//判断文件或者目录是否存在

ff.isDirectory();//判断是否是文件夹

ff.isFile();//判断是否是文件

ff.createNewFile();//不存在则创建新文件

ff.delete();//删除文件

ff.renameTo(**new** File("D:\\xx.txt"));//移动文件，也可以修改文件夹名,文件夹必须存在

ff.mkdirs();//创建目录，目录可以多层级创建

ff.mkdir();//创建目录，目录只能是一层新目录

ff.delete();//删除目录，不能删除有子目录的目录

#### 遍历文件夹和子文件夹

**void** ShowFile(File ff)

{

**if**(!ff.isDirectory()) **return**;

File[] ffList = ff.listFiles();

**for**(File f : ffList)

{

**if** (f.isFile()){

String fileName = f.getName();

**if**(fileName.endsWith(".txt"))

System.***out***.println(f.getName() + " " + f.getPath());

}

**else** *ShowFile*(f);

}

｝

#### 读文件

FileInputStream fis = **new** FileInputStream("D:\\1.txt");

InputStreamReader isr = **new** InputStreamReader(fis,"UTF-8");

BufferedReader bfr = **new** BufferedReader(isr);

String line;

**while**((line = bfr.readLine()) != **null**)

{

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

}

bfr.close();

isr.close();

fis.close();

#### 写文件

FileOutputStream fos = **new** FileOutputStream("D:\\2.txt");

OutputStreamWriter osw = **new** OutputStreamWriter(fos, "utf-8");

BufferedWriter bfw = **new** BufferedWriter(osw);

bfw.write("aaaa\r\n");

bfw.close();

osw.close();

fos.close();

#### 追加内容

FileWriter和FileReader只能是本地的utf-8编码，不能更改

**boolean** append = **true**;

FileWriter fw = **new** FileWriter("D:\\1111.txt",append);

fw.write("aaaa\r\n");

fw.flush();

fw.close();

### 序列化

//实现序列化接口Serializable（包含serialVersionUID）

//序列化类中的成员必须要支持序列化 static,transient后的变量不能被序列化

//当一个父类实现序列化，子类自动实现序列化，不需要显式实现Serializable接口

**class** People **implements** Serializable

{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**int** id;

String name;

People(**int** id,String name)

{

**this**.id = id;

**this**.name = name;

}

}

**class** PeopleList **implements** Serializable

{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;//确保序列化和反序列化使用相同id

**private** ArrayList<People> list = **new** ArrayList<People>();

**void** Add(**int** id , String Name)

{

list.add(**new** People(id, Name));

}

**void** Show()

{

**for**(People p:list) System.***out***.println(p.id + " " + p.name);

}

}

============序列化

**void** Save() **throws** FileNotFoundException, IOException

{

PeopleList pList = **new** PeopleList();

pList.Add(1, "aa");

pList.Add(2, "bb");

pList.Add(3, "cc");

ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("D:\\temp.bin"));

oos.writeObject(pList);

oos.flush();

oos.close();

}

==========反序列化

**void** Read() **throws** FileNotFoundException, IOException, ClassNotFoundException

{

ObjectInputStream oin = **new** ObjectInputStream(**new** FileInputStream("D:\\temp.bin"));

PeopleList list = (PeopleList ) oin.readObject();

list.Show();

}

## Generics

### 泛型

import java.util.\*;

for(object x : arr) 不能改变其中内容，只能访问元素

不能在<>中写基本类型，只能是基本类型的包装类型

//迭代容遍历

Iterator<String> it = l.iterator();

**while**(it.hasNext()){

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

}

**class** Data<T>

{

T temp;

**public** **void** save(T temp)

{

**this**.temp = temp;

}

**public** Object get()

{

**return** temp;

}

}

使用时

Data<String> d = **new** Data<String>();

d.save("abc");

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

### 去重/排序

#### Hash

需要重写方法hashCode和equals

**class** Data

{

**public** **int** i = 0;

//重写方法

**public** String toString()

{

Integer x = i;

**return** x.toString();

}

**public** **int** hashCode()

{

**return** i;

}

**public** **boolean** equals(Object obj)

{

**if**(obj == **this**)

**return** **true**;

**if**(! (obj **instanceof** Data))

**return** **false**;

Data d = (Data)obj;

**return** i == d.i;

}

}

#### Tree

需要实现Comparable

**class** Data **implements** Comparable

{

**public** **int** i = 0;

**public** **int** compareTo(Object obj)

{//如果是包装类型，可以直接用其中的compareTo函数

Data d = (Data)obj;

**return** **this**.i - d.i; //等于返回0，小于返回负数，大于返回正数

}

}

可以自定义排序规则

TreeMap<String,Integer> m = **new** TreeMap<String,Integer>(**new** MyComparator());

**class** MyComparator **implements** Comparator

{

**public** **int** compare(Object obj1,Object obj2)

{

String s1 = (String)obj1;

String s2 = (String)obj2;

**return** s2.compareTo(s1);

}

}

### Vector

Vector<String> v = **new** Vector<String>();

v.setSize(10);

v.trimToSize();//使容量和使用长度相等，充分利用空间

排序

vecStu.sort(**new** Comparator<Student>() {

**public** **int** compare(Student o1, Student o2) {

**return** o1.id - o2.id;}

});

### 单列集合

#### ArrayList

封装了一个长度可变的数组，随机访问快

ArrayList<String> l = **new** ArrayList<String>();

l.add("abc"); //添加元素

l.size()//获取元素个数

l.get(0)//获取下标为0的元素

l.set(0, "xx");//设置下标为0元素

l.remove(0);//移除第0号元素

List<String> sub = l.subList(1, 2);//截取部分

l.addAll(1,sub);//添加list中从下标1开始的元素

l.toString();//转换为字符串包含所有元素

#### LinkedList

内部为双向循环链表，删除节点快

特有方法：getFirst(),removeFirst(),addFirst()

#### HashSet

存储的元素不保证有序且不重复

HashSet<Integer> s = new HashSet<Integer>();

#### TreeSet

存储的元素不重复，且保证有序

### 双列集合

#### HashMap

##### 数据定义

**class** MyKey

{

**public** MyKey(**int** \_id) {

id = \_id;

}

**int** id;

@Override

**public** **int** hashCode() {

//hashMap是数组加链表的结构

//id%3表示hashMap是3个位置的数组，每个位置中再根据equals判断是否相等

//遍历输出时可以看到id顺序是根据%结果放置

**return** id%3;

}

@Override

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

**return** id == ((MyKey)obj).id;

}

@Override

**public** String toString() {

**return** String.*format*("%d", id);

}

}

HashMap<MyKey, Double> map= **new** HashMap<MyKey, Double>();

##### 放入元素

map.put(**new** MyKey(1), 5.1);

map.put(**new** MyKey(2), 7.3);

map.put(**new** MyKey(2), 2.1);//相同则替换

##### 合并

//返回值 参数...

BiFunction<Double, Double, Double> fun = (p1,p2)->{**return** p1 +p2;};

// BiFunction<Integer, Integer, Integer> fun = new BiFunction<Integer, Integer, Integer>()

// {

//

// @Override

// public Integer apply(Integer t, Integer u) {

// // **TODO** Auto-generated method stub

// return t + u;

// }

//

// };

map.merge(**new** MyKey(2), 1.1, fun);//相同则合并

##### 遍历

//遍历key

**for**(MyKey key : map.keySet())

{

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

}

//遍历value

**for**(Double d : map.values())

{

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

}

//遍历key和value

**for** (Map.Entry<MyKey,Double> o : map.entrySet()) {

System.***out***.println(o.getKey() + " " + o.getValue());

}

##### 查找

MyKey key = **new** MyKey(4);

**if**(map.containsKey(key))//查找

{

Double b = map.get(key);

map.put(key, 100.1);//替换

}

#### TreeMap

需要实现Comparable

**class** MyKey **implements** Comparable<MyKey>

{

**public** MyKey(**int** \_id) {

id = \_id;

}

**int** id;

@Override

**public** **int** compareTo(MyKey o) {

**return** id - o.id;

}

@Override

**public** String toString() {

**return** String.*format*("%d", id);

}

}

键值是按照顺序排列

基本的二叉树都需要满足一个基本性质，即树中的任何节点的值大于它的左子节点，且小于它的右子节点。按照这个基本性质使得树的检索效率大大提高。

TreeMap的实现是红黑树[算法](http://lib.csdn.net/base/31" \t "http://blog.csdn.net/zhangyuan19880606/article/details/_blank" \o "算法与数据结构知识库)的实现，是一颗自平衡的排序二叉树

### Properties

Properties p = **new** Properties();

p.setProperty("color", "red");

p.setProperty("size", "12");

System.*out*.println(p.getProperty("color"));

### Collections工具类

ArrayList<Double> l = new ArrayList<Double>();

Collections.reverse(l);//对list进行反转

shuffle(list)//进行随机排序

sort(list)

swap(list,int i, int j)将制定list中i处和j处元素进行交换

addAll(Collection c, T elements)将所有元素添加到制定的collection中

binarySearch(list,object)使用二分法查找有序的list中的某个元素

Object max(Collection col)查找最大元素

Object min(Collection col)

replaceAll(list,oldobj,newobj)用新的obj替换list中所有旧的obj

## socket

### URL

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

String s = "www.baidu.com";

InetAddress ts = **null**;

**try**

{

ts = InetAddress.*getByName*(s);

}**catch**(UnknownHostException e){}

**if**(**null** != ts)

{

System.*out*.println(s + " IP地址是 " + ts.getHostAddress());

}

//构造和解析URL

URL u = **new** URL("https://www.baidu.com/s?ie=utf-8&wd=baidu");

u.getProtocol() //获取协议https

u.getHost()//获取主机名www.baidu.com

u.getFile()//获取文件名s?ie=utf-8&wd=baidu

**import** java.io.\*;

//读取当前页面的内容，用字符串读出html文件中写的内容

BufferedReader r = **new** BufferedReader(**new** InputStreamReader(u.openStream()));

String s;

**while**(**null** != (s = r.readLine()))

{

System.*out*.println(s);

}

r.close();

### TCP

**import** java.io.\*;

**import** java.net.\*;

#### 服务端

**try** {

ServerSocket server = **new** ServerSocket(1234);

**while** (**true**) {

Socket s = server.accept();

DataInputStream data = **new** DataInputStream(s.getInputStream());

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

data.close();

s.close();

}

} **catch** (Exception e) {}

#### 客户端

**try** {

Socket s = **new** Socket("localhost", 1234);

DataOutputStream data = **new** DataOutputStream(s.getOutputStream());

data.writeUTF("xx");

data.close();

s.close();

} **catch** (Exception e) {}

### UDP

**import** java.net.\*;

DatagramPacket用于发送和接收时，构造函数不同

#### 服务端

**try**

{

DatagramSocket dSocket = **new** DatagramSocket(1234);

**while**(**true**){

**byte**[] inBuffer = **new** **byte**[100];

DatagramPacket inPacket = **new** DatagramPacket(inBuffer, inBuffer.length);

dSocket.receive(inPacket);//接收信息

InetAddress cAddr = inPacket.getAddress();

**int** cPort = inPacket.getPort();

String s = **new** String(inPacket.getData(),0,inPacket.getLength());

System.*out*.println("recive: " + s);

System.*out*.println("client name: " + cAddr.getHostName());

System.*out*.println("client prot: " + cPort);

}

}**catch**(Exception e){}

#### 客户端

**try** {

DatagramSocket dSocket = **new** DatagramSocket();

InetAddress sAddr = InetAddress.*getByName*("127.0.0.1");

String s = "xxxx";

**byte**[] outBuffer = s.getBytes();

DatagramPacket outPacket = **new** DatagramPacket(outBuffer,outBuffer.length,sAddr,1234);

dSocket.send(outPacket);

dSocket.close();

} **catch** (Exception e) {}

## c++Mix

### C++调用java

JNI(Java Native Interface)意为JAVA本地调用，它允许Java代码和其他语言写的代码进行交互

需要将jre\bin\server\jvm.dll加入环境变量path，移动dll放入工程运行时出错

加载包时，需要包的全路径，如env->FindClass("java/lang/String");

#### Java文件

编译成class文件TestFun.class

**public** **class** TestFun {

**public** **static** **int** intMethod(**int** x,**int** y) {

**return** x\*y;

}

**public** **void** sayHelloFromJava(String str,**int** listLen,**int**[] listVal)

{

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

**for**(**int** i = 0 ; i <listLen;++i)

{

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

}

}

}

#### C++调用

#include <jni.h>

#pragma comment(lib,"jvm.lib")

头文件目录

C:\Java\jdk1.8.0\_60\include;C:\Java\jdk1.8.0\_60\include\win32

Lib目录

C:\Java\jdk1.8.0\_60\lib

运行dll目录

C:\Java\jre1.8.0\_60\bin\server

//加载jvm

JavaVMInitArgs vm\_args;

memset(&vm\_args, 0, sizeof(vm\_args));

vm\_args.version = JNI\_VERSION\_1\_8;//jdk版本

vm\_args.nOptions = 1;

JavaVMOption options[1];

options[0].optionString = "-Djava.class.path=.";

vm\_args.options = options;

JNIEnv \*env;

JavaVM \*jvm;

long status = JNI\_CreateJavaVM(&jvm, (void\*\*)&env, &vm\_args);

if (status == JNI\_ERR) throw(string("JNI\_CreateJavaVM erro"));

//加载类TestFun

jclass cls = env->FindClass("TestFun");

if (0 == cls ) throw(string("FindClass erro"));

//调用类中的静态方法intMethod

jmethodID mid = env->GetStaticMethodID(cls, "intMethod", "(II)I");//返回值int，两个参数为int类型

if (0 == mid) throw(string("intMethod erro"));

jint square = env->CallStaticIntMethod(cls, mid, 5,5);

printf("Result of intMethod: %d\n", square);

//创建类并调用函数sayHelloFromJava

jmethodID ctor = env->GetMethodID(cls, "<init>", "()V");

jobject obj = env->NewObject(cls, ctor);

mid = env->GetMethodID(cls, "sayHelloFromJava", "(Ljava/lang/String;I[I)V");//无返回值，第一个参数int，第二个String

if (0 == mid) throw(string("sayHelloFromJava erro"));

jstring str1 = env->NewStringUTF("I am class Instance");

const jint len = 3;

jintArray testIntArray = env->NewIntArray(len);

jint test[len] = {1,2,3};

env->SetIntArrayRegion(testIntArray, 0, len, test);

env->CallVoidMethod(obj, mid, str1, len, testIntArray);

jvm->DestroyJavaVM();

域描述符

|  |  |
| --- | --- |
| V | void |
| Z | boolean |
| B | byte |
| C | char |
| S | short |
| I | int |
| J | long |
| F | float |
| D | double |
| L.. | 引用类型  如Ljava/lang/String; |
| [.. | 数组类型  如 [I |
| [[.. | 二维数组  如 [[I |

## extend

### swing

======安装

http://www.eclipse.org/windowbuilder/download.php

页面找到eclipse对应版本，点击link

复制下载地址粘贴在eclipse中help-》install new software中，下载插件

新建other-》windowBuilder-》Swing designer-》JFrame

======控件设置

MVC（Model View Controller）：模型、视图和控制器

点击design进入可视化编辑（view）

点击控件，再点击窗口位置即可加入代码（Model）

右键控件，Add event hander加入响应事件(Control)

### json

=======依赖

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

<version>20180130</version>

</dependency>

=======生成

//{"name":"AA","do":["eat","sleep"],"age":10}

JSONObject obj = new JSONObject() ;

obj.put("name", "AA") ;

obj.put("age",10) ;

JSONArray arr = new JSONArray() ;

arr.put("eat") ;

arr.put("sleep") ;

obj.put("do", arr) ;

String str = obj.toString() ;

System.out.println(str);

=======解析

String str = "{\"dat\":[{\"inf\":\"abc\",\"lon\":\"116\",\"lat\":\"116\"},{\"inf\":\"def\",\"lon\":\"116.1\",\"lat\":\"116.1\"},{\"inf\":\"gg\",\"lon\":\"116.2\",\"lat\":\"116.2\"}]}";

JSONObject jsonObj = new JSONObject(str);

JSONArray arr = jsonObj.getJSONArray("dat");

for (Iterator<Object> tor = arr.iterator(); tor.hasNext();) {

JSONObject obj = (JSONObject) tor.next();

System.out.println(obj.get("inf"));

System.out.println(obj.get("lon"));

System.out.println(obj.get("lat"));

}

String json = '[["abc","116","39"],["def","116.1","39.2"],["gg","116.2","39.1"]]'

JSONArray arr = new JSONArray(json);

int len = arr.length();

for (int pos = 0 ; pos< len;++pos)

{

JSONArray arrObj = (JSONArray)arr.get(pos);

if(arrObj.length() != 3) continue;

String infStr = (String) arrObj.get(0);

String lonStr = (String) arrObj.get(1);

String latStr = (String) arrObj.get(2);

}

### log4j

#### 配置说明

log4j.rootLogger = [ level ] , appenderName, appenderName, …

level 是日志记录的优先级

优先级从高到低：【字母不区分大小写】

OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL、自定义的级别。

通过在这里定义level，可以控制到应用程序中相应级别的日志信息的开关。比这里定义了INFO级别，则应用程序中所有INFO级别以下日志信息将不被输出。

appenderName指带后面要配置的日志控制，可以同时编写多个日志控制。其中的Threshold字段定义了控制级别，表示控制该级别以上的日志。

如：log4j.appender.\*\*\*\*.Threshold = DEBUG 表示控制DEBUG 级别以上的日志的输出位置（在原有底层优先级日志输出的基础上，将DEBUG 级别以上的日志再输出到另一个文件）

配置日志信息输出目的地Appender，其语法为：

log4j.appender.appenderName = fully.qualified.name.of.appender.class

log4j.appender.appenderName.option1 = value1

log4j.appender.appenderName.option = valueN

其中，Log4j提供的appender有以下几种：

org.apache.log4j.ConsoleAppender（控制台），

org.apache.log4j.FileAppender（文件），

org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件），

org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件），

org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

#### 依赖

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

#### 配置文件

log4j.properties文件放在工程生成的classpath下

java文件夹下放java代码，该目录中的文件就会放在classpath下

javaweb中放在WEB-INF/classes

### 设置###

log4j.rootLogger = debug,A,B,C

### 输出ALL级别以上日志到控制台###

log4j.appender.A = org.apache.log4j.ConsoleAppender

log4j.appender.A.Target = System.out

log4j.appender.A.layout = org.apache.log4j.PatternLayout

log4j.appender.A.layout.ConversionPattern = [%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n

### 输出DEBUG 级别以上的日志到=./logs/a.log javaweb只能绝对路径###

log4j.appender.B = org.apache.log4j.DailyRollingFileAppender

log4j.appender.B.File = ./logs/a.log

log4j.appender.B.Append = true

log4j.appender.B.Threshold = DEBUG

log4j.appender.B.layout = org.apache.log4j.PatternLayout

log4j.appender.B.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

### 输出ERROR 级别以上的日志到=./logs/b.log ###

log4j.appender.C = org.apache.log4j.DailyRollingFileAppender

log4j.appender.C.File =./logs/b.log

log4j.appender.C.Append = true

log4j.appender.C.Threshold = ERROR

log4j.appender.C.layout = org.apache.log4j.PatternLayout

log4j.appender.C.layout.ConversionPattern = %-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n

#### 调用

**import** org.apache.log4j.Logger;

**private** **static** Logger *logger* = Logger.*getLogger*(类名.**class**);

*logger*.debug("debug");

*logger*.info("info");

*logger*.warn("warning");

*logger*.error("error");

*logger*.fatal("fatal");

### request

<dependency>

<groupId>org.apache.httpcomponents</groupId>

<artifactId>httpclient</artifactId>

<version>4.5.6</version>

</dependency>

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.io.IOException;

import java.net.URI;

import java.net.URISyntaxException;

import org.apache.http.HttpEntity;

import org.apache.http.HttpResponse;

import org.apache.http.NameValuePair;

import org.apache.http.StatusLine;

import org.apache.http.client.ClientProtocolException;

import org.apache.http.client.config.RequestConfig;

import org.apache.http.client.entity.UrlEncodedFormEntity;

import org.apache.http.client.methods.CloseableHttpResponse;

import org.apache.http.client.methods.HttpGet;

import org.apache.http.client.methods.HttpPost;

import org.apache.http.impl.client.CloseableHttpClient;

import org.apache.http.impl.client.HttpClientBuilder;

import org.apache.http.message.BasicNameValuePair;

import org.apache.http.util.EntityUtils;

public static byte[] GetRequest(String url) throws URISyntaxException, ClientProtocolException, IOException {

RequestConfig config = RequestConfig.custom().setConnectTimeout(5000).setSocketTimeout(3000).build();

CloseableHttpClient client = HttpClientBuilder.create().setDefaultRequestConfig(config).build();

HttpGet httpGet = new HttpGet();

httpGet.setURI(new URI(url));

HttpResponse httpResponse = client.execute(httpGet);

HttpEntity entity = httpResponse.getEntity();

byte[] body = EntityUtils.toByteArray(entity);

StatusLine sL = (StatusLine) httpResponse.getStatusLine();

int statusCode = ((org.apache.http.StatusLine) sL).getStatusCode();

if (statusCode != 200)

return null;

return body;

}

public static byte[] sendPost(String url, Map<String, Object> params) throws ClientProtocolException, IOException {

List<NameValuePair> pairs = null;

if (params != null && !params.isEmpty()) {

pairs = new ArrayList<NameValuePair>(params.size());

for (String key : params.keySet()) {

pairs.add(new BasicNameValuePair(key, params.get(key).toString()));

}

}

HttpPost httpPost = new HttpPost(url);

if (pairs != null && pairs.size() > 0) {

httpPost.setEntity(new UrlEncodedFormEntity(pairs));

}

RequestConfig config = RequestConfig.custom().setConnectTimeout(5000).setSocketTimeout(3000).build();

CloseableHttpClient client = HttpClientBuilder.create().setDefaultRequestConfig(config).build();

CloseableHttpResponse response = client.execute(httpPost);

int statusCode = response.getStatusLine().getStatusCode();

if (statusCode != 200) {

httpPost.abort();

throw new RuntimeException("HttpClient,error status code :" + statusCode);

}

HttpEntity entity = response.getEntity();

if (entity != null) {

return EntityUtils.toByteArray(entity);

} else {

return null;

}

}

String url = "http://127.0.0.1:8080/DevSer/GetDevDat";

//byte[] body = GetRequest(url);

Map<String, Object> params = new HashMap<String, Object>();

params.put("IntVal", 12);

params.put("StrVal", "ss");

byte[] body = sendPost(url,params);

String json = new String(body, "UTF-8");

### xml

<dependency>

<groupId>jdom</groupId>

<artifactId>jdom</artifactId>

<version>1.1</version>

</dependency>

import java.io.\*;

import org.jdom.\*;

import org.jdom.input.\*;

SAXBuilder builder = new SAXBuilder();

Document doc = builder.build(new File("cfg.xml"));

Element root = doc.getRootElement();

String Ip = root.getChild("Ip").getValue();

String port = root.getChild("port").getText();

### poi

import java.io.FileOutputStream;

import org.apache.poi.hssf.usermodel.HSSFCell;

import org.apache.poi.hssf.usermodel.HSSFCellStyle;

import org.apache.poi.hssf.usermodel.HSSFRow;

import org.apache.poi.hssf.usermodel.HSSFSheet;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

//创建一个webbook，对应一个Excel文件

HSSFWorkbook wb = new HSSFWorkbook();

//格式

HSSFCellStyle style = wb.createCellStyle();

//在webbook中添加一个sheet,对应Excel文件中的sheet

HSSFSheet sheet = wb.createSheet("学生表一");

//在sheet中第1行第一列加入数据xx

HSSFRow row = sheet.createRow(0);

HSSFCell cell = row.createCell(0);

cell.setCellValue("xxx");

cell.setCellStyle(style); //设置单元格格式

//保存文件

try

{

FileOutputStream fout = new FileOutputStream("D:/students.xls");

wb.write(fout);

fout.close();

}

catch (Exception e)

{

e.printStackTrace();

}