# 更新

2018年8月24日 增加模块：性能的优化

2018年8月14日 在其它模块添加Eclipse查找方法引用、类引用，快捷键

2018年7月21日 去除json放到javaweb中

2018年7月7日 去除文件操作中的文件上传至java常见web中

2018年7月2日 增加了properties、增加了文件操作中io的清空文件注意问题，更新装饰模式、正则表达式、去除类加载器放到基础上、增加Object两个方法重写运用

2018年6月29日增加了模板设计模式 单例模式

2018年6月27日增加在其它增加了properties中文问题

2018年6月25日增加在其它增加了int超出范围问题

2018年6月25日更新了获取今天或者之后多少天的日期

2018年6月23日增加了json格式

2018年6月21日增加了date类:计算两个日期相隔天数

2018年6月20日增加了Calendar类：判断闰年、之后和今天的日期

# Object

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：被克隆的对象中需重写Object中的clone的方法：把权限修饰符protected改为public即可(因为是自己要被别人克隆，权限要公共)：

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：调用：如有异常，被克隆对象没实现Cloneable：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) {

// TODO Auto-generated catch block

e.printStackTrace();

}

p.setName("你好");

cachCode和equalse的运用 快捷键生成对比你指定的变量如都一致是重复的

没有重写hashCode方法的类，返回32位对象在JVM中的地址；Long类重写了hashCode方法，返回计算出的hashCode数值

ComHashcode a = new ComHashcode(); a.hashCode()//打印870919696

Long num = new Long(8); num1.hashCode()//打印 8

集合的SetHash、MapHash(键)存储机制都需要调用这两个方法：

第一步：先调用hashCode方法查询到哈希值的位置:没有值直接放入，有值才会调用equals方法

第二步：调用equals方法：拿参数进行对比如：返回true不让存储，返回false在同一个哈希值存储两个数据

**简单例子**：实体类User：根据id进行存储至set集合，id一致的话不让存储进hashSet集合中

**实体类**

public class User {

private String s;

private int id;

public User(int id,String s) {this.id=id; this.s=s; } //构造方法

public String getS() {return s; }

public void setS(String s) {this.s = s; }

public int getId() {return id; }

public void setId(int id) {this.id = id; } 两个属性getset封装方法

第一步 重写hashCode方法，固定让id的值进行查找内存中的哈希表位置

public int hashCode() {

return this.id;

}

第二步 重写equals方法，传进来的值和本类的id进行对比

public boolean equals(Object obj) {//此参数传进来的值是你添加的对象set.add(new User(1,"福福"));

User u = (User) obj;

return this.id==u.getId(); //id对比

}

**Main方法调用**

Set<User> set=new HashSet<User>();

//哈希表特点**桶式结构（1个坑多个萝卜）**：算出来的哈希码值一样，进行equals对比不一致就两个元素放在同一个地方

set.add(new User(1,"福福"));

set.add(new User(1,"阿福"));

System.out.println(set.size()); 打印：1，（没有重写则打印2： 没有重写hashCode方法都是拿内存地址进行对比的）

# 集合|数组|properties

数组

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]

Properties 继承Hashtable(和HashMap区别：线程是安全的)

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.

第一步 判断配置文件是否存在

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

if(!file.exists()){

file.createNewFile();//如果配置文件不存在，则创建该配置文件

}

第二步 创建Properties对象 读取文件。

Properties properties = new Properties();

properties.load(new FileInputStream(file)); //把配置文件的信息加载到properties中

第三步 创建FileOutputStream对象重新生成配置文件对象：一定要先把目标文件加载到内存中，创建对象代码会把文件清空

注意：此FileOutputStream对象会先进行清空目标文件，构造方法中加true也不行，会内容追加多次

FileOutputStream fileOutputStream = new FileOutputStream(file);

int count = 0;

第四步 读取配置文件的运行次数

//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

String value = properties.getProperty("count");

if(value!=null){

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

}

第五步 判断使用的次数是否已经达到了三次 并推出jvm：System.exit(0);

if(count==3){

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

properties.store(fileOutputStream,"runtime"); //保存参数：流，描述文件

System.exit(0); //退出java虚拟机

}

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

properties.setProperty("count",count+""); //修改key中的值

第六步 重新保存文件

properties.store(fileOutputStream,"runtime");

}

# 异常

Throwable 是一个正常类 可以创建对象，

此类3个方法的效果如下:

public static void main(String[] args) {

//创建了一个Throwable对象。

Throwable t = new Throwable("头晕，感冒");

System.out.println("message: "+ t.getMessage());打印： message: 头晕，感冒

System.out.println("toString: "+ t.toString();); //打印：toString: java.lang.Throwable: 头晕，感冒

test();

}

public static void test(){

Throwable t = new Throwable();

t.printStackTrace();//打印：java.lang.Throwable

at com.wwl.reflect.Test.test(Test.java:16)

at com.wwl.reflect.Test.main(Test.java:12)

}

自定义异常操作

Sun公司没有内置负数异常：可以自己定义：

第一步：继承Exception或者RuntimeException

public class FuShuException extends RuntimeException{ //一般都是继承运行时异常

public FuShuException(String s){

super(s); //如果需要消息：需要调父类的带String构造方法

}

}

public static void main(String[] args) {

int avg = getAvg(50,60,-70,80);

System.out.println(avg);

}

\* 成绩没有负数,需要抛出异常,停止运算

public static int getAvg(int...source){ //运行异常不用抛出

int sum = 0 ;

for(int s : source){

if( s < 0){

第二步 手动抛出异常自己定义的异常对象（因为是此类是继承了运行时异常类所以不用抛出）

throw new FuShuException("成绩错误 "+s);

}

sum = sum + s;

}

return sum/source.length;

}

# 时间类

### Date

计算两个日期的相隔天数(自己活了多少岁)

Public long getInterval(Date begin, Date end) throws Exception{

long day = 0;

SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd"); //第一步 创建格式化时间对象

if(begin\_date != null){

String b= sdf.format(begin); //第二步 先将Date格式化为只有年、月、日的形式，返回String。

begin = sdf.parse(b); //第三步然后将此String再转化为Date(此时时、分、秒就成为了00:00:00）

}

if(end!= null){

String e= sdf.format(end);

end = sdf.parse(e);

}

//第四步 调用getTime()方法返回Date的1970的毫秒表示:将两个日期的毫秒相减，然后除以一天有（24\*60\*60\*1000）毫秒，得到日期差。

day = (e.getTime()-b.getTime())/(24\*60\*60\*1000);

return day;

}

### Calendar

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");

}

}

第二种方式

if(i % 4 == 0 && i % 100 != 0 || i % 400 == 0){ //是闰年}else{ //不是闰年}

获取今天或者之后多少天的日期

第一步创建对象它已经是当前时间

Calendar calendar=Calendar.getInstance();

第二步 获取今天的日期，如6月25，那就是25

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

System.out.println("今年的日期是："+calendar.get(Calendar.DAY\_OF\_YEAR));是获取本年的日期

calendar.clear();//此语句是把日期更改为本月第一天,网上的是不正确的不应该用此方法

第三步 参数：今天的日期和未来的日期

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

第四步 再进行调用第二步的方法

//注意：使用clear方法是打印：十天之后的日期是：11，是不正确的，它把日期变为了本月1号

没有使用方法是正确的显示：下个月5号

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

# 文件操作

1、每次运行到代码FileOutputStream、FileWrite对象都会将目标文件进行清空

创建目录：mkdirs为子目录也进行创建，mkdir只创建一个文件目录

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

FileWriter f=new FileWriter(file); //运行完此代码：目标文件就已经清空了，加了true：不清空，追加内容

f.write("你好");

f.close();

实例 模拟加了true的操作

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

int count=0;

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

if(count ==0){

f=new FileWriter(file); 关键只运行一次创建对象

}

f.write("第"+(++count)+"次");打印：第1次第2次

}

f.close();

2、通过HashCode配合UUID获得8位字符创建8个子文件目录

String name=Integer.toHexString(UUID.randomUUID().hashCode()); //’ a’的hachCode是61;使用toHexString方法

String mkd="e:/update";

for(char c : name.toCharArray()) { //转成char数组

mkd +="/"+c;

}

new File(mkd).mkdirs();

## 执行exe文本

通过notepad.exe 来打开后缀txt文件需要用到Runtime类

File[] root= File.listRoots(); //获得所有系统盘符

Runtime runtime= Runtime.getRuntime(); //用来启动windos系统exe文件类

for (File f : root) {

if("E:\\".equals(f.getPath())) { //进入E盘

String[] tr=f.list(); //获得e盘根目录所有的文件和文件夹

for(String f1 : tr) {

if("自己".equals(f1)) {//进入”自己”文件夹

File f2=new File(f.getPath()+File.separator+f1);//创建File路径：E:\自己

String [] s=f2.list();

for (String string : s) {

if("oracle.txt".equals(string)) { //判断此文件是oracle.txt

//注意 这里要打开后缀txt文件的都是通过notepad.exe 打开的，exe后面需要一个空格

runtime.exec("notepad.exe "+f.getPath()+File.separator+f1+File.separator+"oracle.txt");

# 动态代理

静态代理书写步骤:

1.要求被装饰者和装饰者实现同一个接口或者继承同一个类

2.在装饰者中要有被装饰者的引用

3.对需要加强的方法进行加强

4.对不需要加强的方法调用原来的方法

## Proxy

作用：

1、不改源码的情况下，执行此方法前增加代码，或执行之后增加代码等

2、不改源码的情况下，改造方法的实现效果，如之前改造colse效果

匿名InvocationHandler接口操作

步骤一：被代理类必须是实现接口的类:A类实现类与AInterfaces接口类

public class A implements AInterfaces {

public void method() {

System.*out*.println("A方法");

}

}

public interface AInterfaces {

public abstract void method();

}

步骤二：创建代理类：P类，创建方法：dd();内部使用Proxy.newProxyInstance()方法得到被改造后的对象：Object

**public** **class** P {

**private** AInterfaces a=**new** A();

创建方法

**public** Object dd() {

参数：类装载器，目标对象所有接口,匿名代理对象InvocationHandler，

**return** Proxy.*newProxyInstance*(a.getClass().getClassLoader(),a.getClass().getInterfaces(),

**new** InvocationHandler() {

代理对象，当前执行的方法，当前方法的参数

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

**if**("method".equals(method.getName())) {

//在调用本类method方法前加一句话，（如果是全改此方法返回值不能是：method.invoke(ai, args);）

System.***err***.println("加你一句话进去");

**return** method.invoke(ai, args); ai这里必须是声明的对象，不可以A.class

}

//不改造的方法直接通过

**return** method.invoke(ai, args);

步骤三：得到改造后的对象：但还是Object，必须用接口类进行强转，再调用方法

ProxyTest pt=**new** ProxyTest();

Object o=pt.dd();//得到改造后的object对象

AInterfaces a=(AInterfaces) o;//必须用接口类进行强转和接收

a.method();

注意事项：

1、Proxy.newProxyInstance()：创建方式

要先进行创建对象AInterfaces a=new A();或A a=new A();用声明的名称

Proxy.newProxyInstance(类的装载,类的接口 , InvocationHandler(代理类接口))

2、method.invoke(ai, args)：不想改造的方法直接这样通过。ai 参数不能A.class

参数：1、声明的对象。2、方法中自带的Object[] args（返回值参数）

如要全新改造：被改造的方法就不要return method.invoke(ai, args)： 需return新方法的值，没有返回值就null代替.

3、调用方法：因为得到了被改造后的Object对象，强转要必须注意：

强转操作：必须用接口类进行强转和接收

4、然后进行调用要调用的方法，实现改造

传入对象操作

步骤一：被代理类必须是实现接口的类:A类实现类与AInterfaces接口类

public class A implements AInterfaces {

public void method() {

System.*out*.println("A方法");

}

}

public interface AInterfaces {

public abstract void method();

}

步骤二：创建代理类：P类，实现接口：InvocationHandler，创建instance方法：内部使用方法Proxy.newProxyInstance方法：有返回值Object（得到改造后的对象）

public class B implements InvocationHandler{

private Object obj;

public Object instance(Object 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

this. obj =o;

//参数：目标对象A的类装载、获取被代理的所有接口，代理对象变成this

return Proxy.newProxyInstance(obj.getClass().getClassLoader(), obj.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");

//必须要有一个返回值，没有就null代替

return "真的啊";

}else {

result = method.invoke(obj, args);//可以说获得返回值

return result;

}

步骤三：调用操作：创建代理对象，调用代理对象的方法，得到改造后被代理Object对象的返回值，进行强转，调用方法

public static void main(String[] args) {

B b=new B();

//获得被改造后的被代理对象

Object obj=b.instance(new A());

// 使用Ainterfaces接口类进行强制转换和接收，

AInterfaces a=(AInterfaces) obj;

String c=a.MyName();

System.out.println(c);

a.MyName2();

## **Cglib代理**

此框架不要求目标类实现接口（被代理对象）Spring框架jar包已经有包含在内

第一步：导入jir包

经测试导入此cglib-nodep-3.1jar包就可以了

cglib-nodep-3.1.jar使用nodep包不需要关联asm的jar包,jar包内部包含asm的类.

cglib -3.1.jar 使用此jar包需要关联asm的jar包,否则运行时报错. asm-4.2.jar

第二步：创建被代理类：A

public class A {

public void add() {

System.out.println("此方法是A类add方法");

}

}

第三步：创建实现接口MethodInterceptor的代理类：B类，注意：invokeSuper

public class B implements MethodInterceptor{

public Object intercept(Object obj, Method method, Object[] params, MethodProxy proxy) throws Throwable {

if("add".equals(method.getName())) {

System.out.println("执行被增强的方法前执行");

//只想在改造方法的前后加代码的操作

Object o=null;

o=proxy.invokeSuper(obj, params);//注意此地方的方法和proxy不同哦

System.out.println("执行完被增强的方法后执行");

return o;

}

//不想改造的方法通过

return proxy.invokeSuper(obj, params);

}

第四步：调用操作：创建对象Enhancer，对象中的方法setSuperclass传入被代理对象，对象中的方法setCallback【回收】代理对象，对象中的方法create()得到改造后的对象

public static void main(String[] args) {

//生成对象Enhancer【增强子；强化剂；增加者】

Enhancer en = new Enhancer();

//传入被增强方法的类

en.setSuperclass(A.class);

//传入实现接口MethodInterceptor的类Callback

en.setCallback(new B());

A a=(A) en.create();

a.add();

}

# Jdk5新特性

## 注解操作

模仿junit的注解

**第一步 定义注解类**

注解本质就是一个接口,接口中可以有常量和抽象方法，抽象方法在注解中就称之为注解属性

注解属性类型:①基本类型②String③Class④Annotation⑤Enum:枚举⑥最后是1到5的数组

@Retention 规定注解保留到什么阶段 值为RetentionPolicy的三个枚举值

SOURCE:只在代码中保留,在字节码文件中就删除了

CLASS:在代码和字节码文件中保留

RUNTIME:所有阶段都保留

@Target 规定注解作用在什么上面 值为ElementType的枚举值

TYPE:作用在类 接口 等上面

METHOD:作用方法上面

FIELD:作用字段上面

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface MyTest {

//有默认值就可以在要注解的类上不用定义值了

public String value() default "";

int id();

String name() default "";

}

**第二步 定义处理注解的类 Processor（处理器）**

判断有无注解**：**method.isAnnotationPresent(注解类.class)

获得注解对象：MyTest mt=method.getAnnotation(注解类.class);

执行此方法：m.invoke(clazz.newInstance());还重载方法：（正常类实例化，可变参数）

public class Processor{

private static int id; //成员变量 id

private static String name; //成员变量 name

private static String value; //成员变量 value

public static int getId() {return id; }

public static String getName() {return name; }

public static String getValue() {return value; }

public static <T>T Common(Class<T> clazz) {

Method[] method=clazz.getMethods();

for(Method m:method) {

//判断这些方法有无使用MyTest注解

if(m.isAnnotationPresent(MyTest.class)) {

//得到注解的对象MyTest

MyTest mt=m.getAnnotation(MyTest.class);

//得到属性id值、name值、value值,可通过成员变量赋值

id=mt.id();

name = mt.name();

value = mt.value();

//执行此类中带有注解MyTest方法

m.invoke(clazz.newInstance());

}

}

return clazz.newInstance();

第三步 实体类和调用

实体类

public class Entity{

//①MyTest注解类定义了@Target(ElementType.METHOD)只能在方法中用此注解

@MyTest(id=1,name="f1的方法name",value="f1的方法value")

public void f1() {

System.out.println("我是f1方法");

}

//②id在注解中没有默认值所以必须赋值

@MyTest(id=2)

public void f2() {

System.out.println("我是f2方法");

}

调用main：Junit原理就是这样的，只是eclipse集成了：所以可以右键进行执行

public static void main(String[] args) {

//调用处理注解的类方法就会判断传入的类有没有注解

Entity e=Processor.Common(Entity.class);

}

## 泛型

1、方法泛型:定义在方法上的泛型就叫做方法泛型,作用的范围的当前方法内部，可以认为,当方法在被调用到时,虚拟机自动判断出泛型的具体类型.

第一步：在方法内返回值左边进行声明泛型<T>（任意字母，通常大写）;T 是任意类型：①基本数据类型②引用类型

public static **<T>**void a() {}

第二步：在形参中声明：①集合②数组③只是任意类型类型:如果单是任意类型:(T t) ，数组(T[] array)，集合（List<T> list）

public static <T>void change**(T[] arrays)** {}

第三步：如果返回值还是这个传进来的类型 T，可在返回值类型中使用 ：单独任意类型：T ，数组就是T[]，集合就是List<T>

public static <T> **T** change(T[] arrays) {}

编写一个泛型方法，实现指定位置数组元素的交换:①需要最少两个索引值，②泛型数组参数

public static <T>void change(T[] arrays,int i,int j) {

T t =arrays[i];

arrays[i]=arrays[j];

arrays[j]=t;

}

调用

public static void main(String[] args) {

String[] str= {"a","b","c"};

change(str, 0, 1);

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

}

2、类上的泛型:定义在类上的放行叫做类泛型,作用范围是整个类中都可以使用

第一步：在B类中定义一个泛型<T>:

class B<T>{} //当然可以定义多个泛型：class B<T,K>{}, 这个写法就和Map键值对一样的调用了

第二步：创建对象

创建对象：B<String> a= new B<String>();

数组创建对象：B<String[]> a= new B<String[]>();

集合创建对象：B<String[]> a= new B<String[]>();

注意

类中定义的泛型：不能用静态方法上，要使用静态方法上自己进行制定

3、通配符：<?> 前提是使用了泛型：泛型没有继承关系，需要用一个泛型引用来引用不同的泛型实现时,泛型中写他们共同的父类是不行的

List<?> list = null; 两个泛型不一样的基本类型：接收类型需要通用这两个

list = new ArrayList<String>();

list = new ArrayList<Integer>();

4、泛型的边界 限制任意接收类型：如果没有指定泛型默认可以接收任意的类型

Extends（上边界） ：它或者它的子孙类型

class B {} //B类

class C extends B{} //C类继承B类

public class A{

public static void main(String[] args) {

//此时ArrayList<C>中的泛型可以用C类或者B类

List<? extends B> list =new ArrayList<C>();

Super（下边界）：它或它的父类、祖宗类

和上边相反 List<? super B> list =new ArrayList<C>();

注意：extends：

坏处是：在传入对象时,只能传入null

好处是：获取到泛型的对象时,可以调用上边界的方法.

super ：

好处是：可以传入对象时,可以传入下边界的子孙类对象

坏处是：获取到泛型对象时,只能调用Object身上的方法

## 枚举

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");

}

};//枚举值必须在第一位

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()

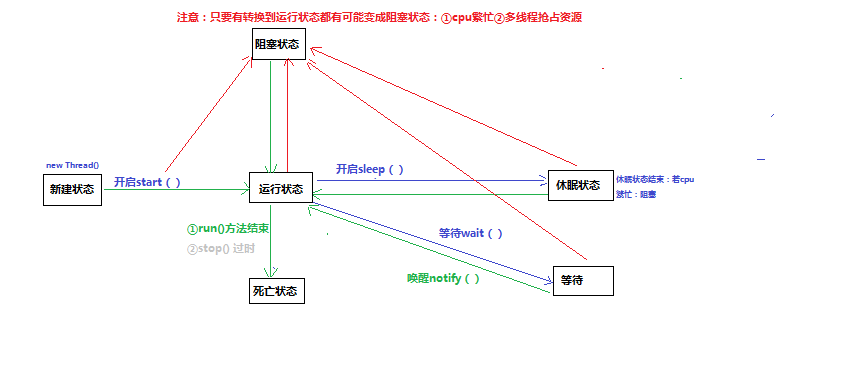
public abstract void a();

}

Mian方法中：调用方法 ： EnumTest.nan.a();

## 线程池

状态 enum类：Thread.State: ①NEW(新建)②RUNNABLE(运行)③BLOCKED(阻塞)④WAITING(等待)⑤TIMED\_WAITING(休眠)⑥TERMINATED(结束)



线程池：java.util.concurrent

**Callable<V>**接口类：代替**Runnable**接口类（Runnable弊端：①没有返回值②不能抛异常）

第一步 Excutors线程工厂类中全是静态方法：Executors.newFixedThreadPool(2);获得接口实现类对象：

//返回线程池对象，接口类接收，参数是int（线程个数）

ExecutorService es=Executors.newFixedThreadPool(2);

第二步 调用submit()方法就会运行：**参数是接口Runnable类或者Callable<T>**

Runnable写法

es.submit(new Runnable() {

public void run() {

System.out.println(Thread.currentThread().getName()+" 线程提交任务");

//打印：原始的线程名字：pool-1-thread-1 线程提交任务 这里打印第一个线程名字

}

});

es.submit(new Runnable() {

public void run() {

System.out.println(Thread.currentThread().getName()+"线程提交任务");

//打印：原始的线程名字：pool-1-thread-2 线程提交任务 这里打印第二个线程名字

}

});

Callable<T>写法：①得到 Future 接口泛型类去接收 ②获得返回值：get（）

Future<String> v=es.submit(new Callable<String>() { **//获得Future<String>接口对象**

public String call(){

return "笑死我了";

}

});

String a=v.get(); **//获得返回值**

System.out.println(a);

第三步: 销毁线程池 （一般不用，建立线程池就是为了频繁使用线程）

es.shutdown();

注意：1）没有调用es.shutdown();它还是在运行状态不会关闭

2）第一步中设置的线程个数2个,运行的线程有3个不会报异常而是会等待哪个线程返回池中了再运行第3个线程。

3）要获取当前线程：Thread.currentThread()

## 同步锁Lock

Lock：java.util.concurrent.Lock :JDk1.5 代替synchronized（同步锁）

private int i=50;

第一步 创建接口Lock类 接收它的实现类 ReentrantLock

private Lock l = new ReentrantLock();

public void run() {

while(true) {

第二步 获取锁 lock.lock();

l.lock();

try {

if(i>0) {

Thread.sleep(10);

System.out.println("还剩票："+--i);

}

} catch (InterruptedException e) {

e.printStackTrace();

} }finally {

第三步 释放锁 调用Lock接口方法unlocklock.unlock();（一般放在try/catch快的finally下）

l.unlock();

}

}

}

注意：同步锁弊端：当线程任务中出现了多个同步（多个锁）时，如果同步中嵌套了其它的同步，这时候容易引发一种现象:程序出现无限等待：死锁，没有办法解决，能避免就避免

## 静态导入

作用：简化书写：导包的时候引入静态static：只对静态成员变量和静态成员方法有效

import static java.lang.System.out;

public class Test {

public static void main(String[] args) {

out.println("我使用了静态导入，就不用前缀System了");

//System.out.println("sfd");

}

}

# java8新特性

## Lambda表达式

使用前提是：必须有”函数式接口”

概念：一个接口类中只有一个抽象方法（不能有第二个），叫函数式接口

如何检测此类是不是函数式接口呢？

直接在接口类上方只用注解@FunctionalInterface，没报错就是

基本格式

Lambda的格式就是为了将抽象方法，翻译为以下三点

1. 一些参数（方法参数）
2. 一个箭头
3. 一些代码（方法体，大括号括起来）

例如抽象方法：

Public abstract int sum(int a , int b);

翻译称为lambda的标准格式：

(int a,int b) -> {return a + b}; method( (int a,int b) -> {return a + b;} );

入门列子：B类实现了A接口：sum加法运算方法，

之前写法

public class C {

public static void main(String[] args) {

//首先使用接口的格式来创建了一个计算机对象，

A a= new B();

//调用method方法

method(a);

}

//参数是接口类型：计算机接口

public static void method(A a){

int result = a.sum(10, 20);

System.out.println("结果是："+result);

}

}

Lambda写法

public static void main(String[] args) {

method((a,b) -> a +b);//lambda表达式简写

method( (int a,int b) -> {return a + b;} );//正常写法

}

//参数是接口类型：计算机接口

public static void method(A a){

int result = a.sum(10, 20);

System.out.println("结果是："+result);

}

区别：直接方法里写表达式，不用new对象，不用创建实现类

总结：Lambda表达式就是充当了A接口类型的参数

初步理解

Lambda表达式前面的小括号，其实就是接口抽象方法的小括号

箭头代表拿着小括号的数据做什么事情，是一个指向的动作

箭头后面就代表拿到了参数之后做什么事情

Lambda表示是的语义本身就代表了怎么做这件事情，没有对象概念，（更加简单直观）

简便格式

1、Lambda表达式当中的参数类型可以省略不写

2、如果参数有且只有一个，那么小括号可以省略。

3、如果语句只有一个，那么大括号和return也可以省略

例子：接口类A，是函数式接口：int sum(int a);

public static void main(String[] args) {

method( (int a) -> {return ++a;} );//正式写法

method((a) -> {return ++a;});//省略参数类型

method(a -> {return ++a;});//省略小括号

method(a -> ++a);//省略大括号和return

}

//参数是接口类型：计算机接口

public static void method(A a){

int result = a.sum(10);

System.out.println("结果是："+result);

}

## 方法引用

在某些场景下，lambda表达式要做的事情，其实在另一类中已经写过了

那么此时如果通过lambda表达式重复编写相同代码，就是浪费、

可以复用已经存在的方法逻辑：

如果Lambda表达式需要做的事情，在另外一个类中已经有了，那么可以直接拿过来替换Lambda方法引用的写法：

通过类名称引用静态方法，格式：类名称 ：：静态方法名

public static void main(String[] args) {

method(a -> System.out.println("将"+a+"做成可口的事务"));

method(B::zuo);//方法调用：B类中的zuo必须要静态的方法

}

public static void method(A a){

a.sum("白菜");

}

通过类名称引用成员方法，格式：对象名 ：：方法名

public static void main(String[] args) {

method(a -> System.out.println("将"+a+"做成可口的事务"));

B b= new B();//先创建对象

method(b::zuo);//方法调用：

}

public static void method(A a){

a.sum("白菜");

}

## Stream流式

入门：如集合要进行截取、转换、过滤、打印之前要大量代码，新特性简便了此写法

public static void main(String[] args) {

List<String> lst = new ArrayList<>();

lst.add("赵丽颖,91");

lst.add("邓超,98");

lst.add("刘备,93");

lst.add("陈赫,33");

lst.add("郑凯,22");

lst.add("神算子,94");

//这句话意思就是：s（自定义命名）得到集合每一个String对象，进行截取每一个分数，再进行转换成int类型，过滤掉90分数以下的，最后遍历每一个进行打印

lst.stream().map(s ->s.split(",")[1]).map(Integer::parseInt).filter(a -> a > 90)

.forEach(System.out::println);

//获取名字，进行遍历打印

lst.stream().map(s ->s.split(",")[0])

.forEach(System.out::println);

}

常用方式获取Stream类方式

Java8当中的”流（不是io流）”其实就是Stream接口对象：你可以认为是高级集合

JDK提供了一个流接口：java.util.stream.Stream<T>

根据集合获取流对象：集合名称.stream();

根据数组获取流对象：Stream.of(数组名称)

注意：当中的元素必须是引用类型才行，

属性

Filter：过滤(能产生boolean结果的lambda)：如果参数lambda产生了true，则要元素

Map：使用lambda表达式操作转换每条数据等操作，可使用方法引用：：有返回值

ForEach：进行遍历每条数据并打印，没有返回值

正式写法：

public static void main(String[] args) {

List<String> lst = new ArrayList<>();

lst.add("91");

lst.add("98");

lst.add("93");

lst.stream().map((String str) -> {

Integer num= Integer.parseInt(str);

return num; //转换变成了Integer集合

}).filter((Integer c) ->{

boolean bool=c >=93;

return bool; //返回布尔值

}).forEach((Integer result) ->{

System.out.println(result);

});

}

简单一点：只有一个参数：去掉括号和属性，只有一句代码：去掉return，和大括号

lst.stream().map(str -> Integer.parseInt(str)).filter(c ->c >=93)

.forEach( result ->System.out.println(result));

再简单一点：使用方法引用

lst.stream().map(Integer::parseInt) .filter(c ->c >=93).forEach( System.out::println);

并发流

如果对流当中的元素，使用多个人同时处理，这就是”并发”

集合名称.parallelStream()和集合名称.Stream():前者速度更快，调用更更多的线程进行操作，具体多少，不清楚，jdk自己处理，只要正确使用，没有线程安全问题

lst.parallelStream().forEach(System.out:: println);//直接获取并发流

lst.stream().parallel();//普通流变并发流

# java9新特性

模块化

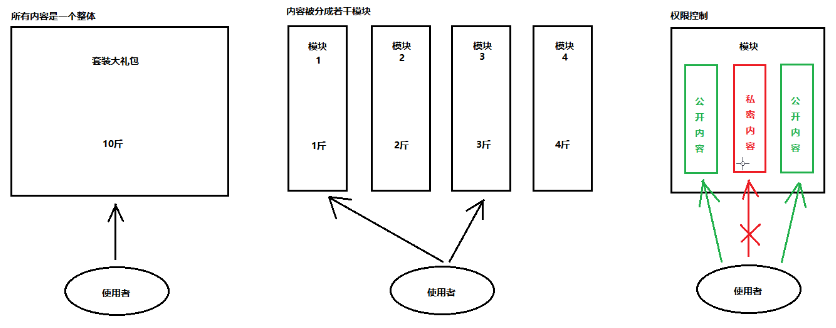
比喻：如捆绑式卖东西，不能只买一部分，要么全买，要么不买，这样不好

零卖了：变成了几个部分（几个模块），就可以自己挑选了，要的留下，不要的丢掉，这就是模块化

一个工程分模块，

感觉就是jar包模块化把包分解几个模块后缀：java.base.jmod

1、Jar体积变小，2、权限控制使用你的jar包更精确



老师讲解了：自带的rt.jar,只用到一丢丢

# 反射

步骤：先获取Class对象—>获得①构造方法对象②普通方法对象③成员变量对象

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();

通过class获得java.lang.reflect.Constructor构造创建对象

第一步 获取Class对象

Class clazz=Class.forName("com.wwl.User");

第二步 获取构造方法 获得java.lang.reflectConstructor构造对象

①暴力获得所有的构造方法包括私有化的

Constructor[] clazz =clazz.getDeclaredConstructors();

for(Constructor con:cons2) {

System.out.println(con);

}

②获取单个构造方法,跟上面方法名就是少了一个S，参数是可变参数

Constructor cons3=clazz.getDeclaredConstructor(int.class,String.class);

第三步 创建对象 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）操作

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);

通过class对象获得java.lang.reflect.Method执行方法

第一步 获得class对象

Class clazz=Class.forName("com.wwl.User");

第二步 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); //打印方法名：public void nih.Person.eat(java.lang.String)

第三步 执行方法操作 参数中需要对象，所以还需要构造方法获得对象

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

Constructor con=clazz.getDeclaredConstructor(int.class,String.class);

User u=( User) con.newInstance();

m.invoke(u,"曾丽方");//执行方法内容没有参数不写或者写null

}

暴力执行私有化的方法：

m2.setAccessible(true);

m2.invoke(p, 666);

通过class对象获得反射类中的成员变量

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);

# 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 单词字符和[a-zA-Z\_0-9] 一样：大小写a到z还有下划线：\_和0到9数字，

\W 非单词字符：[^\w] 除了\w的其它都是true

Greedy 数量词 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

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：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

[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）

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

# 设计模式

单例模式 ：饿汉模式、懒汉模式

一个类Class只有一个实例(对象)存在。 使用Singleton【单例】的好处还在于可以节省内存，因为它限制了实例的个数，有利于Java垃圾回收（garbage collection）

单例类确保自己只有一个实例、必须自己创建自己的实例、必须为其他对象提供唯一的实例

饿汉模式 对于初始化慢，占用资源多的重量级对象使用，首选饿汉模式：程序运行时快

因为单例的实例被声明成 static和 final变量了，在第一次加载类到内存中时就会初始化，所以创建实例本身是线程安全的。

public class A{

private static final A a= new A(); //类加载时就初始化

private A(){} //私有化构造方法

public static A getA(){ return a; }

懒汉模式

public class A {

private volatile static A a; //声明成 volatile 保证编译器不进行优化

private A (){} //私有化构造方法

public static A getA() {

if (a == null) {

synchronized (A.class) {

if (a == null) {

a = new A();}}}

return instance;

}

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();

}

第一步 创建目标类：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");

}

}

第二步 自己创建一个类继承或实现A类同样的接口类： B类

public class B implements AInterfaces{

private AInterfaces aInterfaces; //第一步 在本类维护实现接口的类

//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) { //第二步 在调用中：创建对象传入接口的实现类对象

this.aInterfaces=aInterfaces;

}

public void MyName() { //第三步 要改造的方法进行重写即可

System.out.println("我要改造你");

}

//第四步 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();

}

# 性能优化

1、尽量指定类、方法的final修饰符

带有final修饰符的类是不可派生的。在Java核心API中，有许多应用final的例子，例如java.lang.String，整个类都是final的。为类指定final修饰符可以让类不可以被继承，为方法指定final修饰符可以让方法不可以被重写。如果指定了一个类为final，则该类所有的方法都是final的。Java编译器会寻找机会内联所有的final方法，内联对于提升Java运行效率作用重大，具体参见Java运行期优化。此举能够使性能平均提高50%。

2、尽量重用对象

特别是String对象的使用，出现字符串连接时应该使用StringBuilder/StringBuffer代替。由于Java虚拟机不仅要花时间生成对象，以后可能还需要花时间对这些对象进行垃圾回收和处理，因此，生成过多的对象将会给程序的性能带来很大的影响。

3、尽可能使用局部变量

调用方法时传递的参数以及在调用中创建的临时变量都保存在栈中速度较快，其他变量，如静态变量、实例变量等，都在堆中创建，速度较慢。另外，栈中创建的变量，随着方法的运行结束，这些内容就没了，不需要额外的垃圾回收。

4、及时关闭流

Java编程过程中，进行数据库连接、I/O流操作时务必小心，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销，稍有不慎，将会导致严重的后果。

5、尽量减少对变量的重复计算

明确一个概念，对方法的调用，即使方法中只有一句语句，也是有消耗的，包括创建栈帧、调用方法时保护现场、调用方法完毕时恢复现场等。所以例如下面的操作：

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

建议替换为： 意思就是先算出共几条数据，再判断

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

这样，在list.size很大的时候，就减少了很多的消耗

6、尽量采用懒加载的策略，即在需要的时候才创建

String str = "aaa";if (i == 1){list.add(str);}

建议替换为：先判断再 进行后续的创建对象，定义变量操作

if (i == 1){String str = "aaa";list.add(str);}

7、慎用异常 异常对性能不利，抛出异常首先要创建一个新的对象

Throwable接口的构造函数调用名为fillInStackTrace的本地同步方法，fillInStackTrace方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，Java虚拟机就必须调整调用堆栈，因为在处理过程中创建了一个新的对象。异常只能用于错误处理，不应该用来控制程序流程。

8、循环内不要不断创建对象引用

for (int i = 1; i <= count; i++){Object obj = new Object;}

这种做法会导致内存中有count份Object对象引用存在，count很大的话，就耗费内存了，建议为改为：

Object obj = null;for (int i = 0; i <= count; i++) {obj = new Object; }

这样的话，内存中只有一份Object对象引用，每次new Object的时候，Object对象引用指向不同的Object罢了，但是内存中只有一份，这样就大大节省了内存空间了。

9、关于数组、集合的使用

①基于效率和类型检查的考虑，尽可能使数组array，无法确定数组大小才使用ArrayList

②优先使用HashMap、ArrayList、StringBuilder，

除非线程安全需要，否则不推荐使用Hashtable、Vector、StringBuffer，后三者由于使用同步机制而导致了性能开销

③不要将数组声明为public static final

因为这毫无意义，这样只是定义了引用为static final，数组的内容还是可以随意改变的，将数组声明为public更是一个安全漏洞，这意味着这个数组可以被外部类所改变。

④实现RandomAccess接口的集合比如ArrayList，应当使用最普通的for循环而不是foreach循环来遍历

⑤公用的集合类中不使用的数据一定要及时remove掉

如果一个集合类是公用的（也就是说不是方法里面的属性），那么这个集合里面的元素是不会自动释放的，因为始终有引用指向它们。所以，如果公用集合里面的某些数据不使用而不去remove掉它们，那么将会造成这个公用集合不断增大，使得系统有内存泄露的隐患。

⑥使用'system.arraycopy ()'代替通过来循环复制数组

10、尽量在合适的场合使用单例

使用单例可以减轻加载的负担、缩短加载的时间、提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

（1）控制资源的使用，通过线程同步来控制资源的并发访问

（2）控制实例的产生，以达到节约资源的目的

（3）控制数据的共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信

11 不要创建一些不使用的对象，不要导入一些不使用的类

这毫无意义，如果代码中出现”The value of the local variable i is not used”、”The import java.util is never used”，那么请删除这些无用的内容

12 静态的变量、常量

①常量声明为static final，并以大写命名

这样在编译期间就可以把这些内容放入常量池中，避免运行期间计算生成常量的值。另外，将常量的名字以大写命名也可以方便区分出常量与变量

②尽量避免随意使用静态变量

当某个对象被定义为static的变量所引用，那么gc通常是不会回收这个对象所占有的堆内存的，如：

public class A{ private static B b = new B;}

此时静态变量b的生命周期与A类相同，如果A类不被卸载，那么引用B指向的B对象会常驻内存，直到程序终止

13、使用同步代码块替代同步方法

这点在多线程模块中的synchronized锁方法块一文中已经讲得很清楚了，除非能确定一整个方法都是需要进行同步的，否则尽量使用同步代码块，避免对那些不需要进行同步的代码也进行了同步，影响了代码执行效率。

# 路径专题

1、获取工程目录的物理路径：需要有文件，通常用于查找配置文件

1）、System.getProperty("user.dir")： user.dir是固定的,key：System.getPropertys()可以获得很多key系统信息

应用：可以后面拼接字符串， 如：找到工程目录下的a.txt(不是src下的，编译会在bin下)

System.out.println(System.getProperty("user.dir"));

打印：I:\eclipse\eclipsev4.7\eclipse\_v4.7-64\project\ceshi

Web工程的打印：E:\eclipse\eclipse-jee-4.7-win64\eclipse

FileReader b=new FileReader(System.getProperty("user.dir")+"\\a.txt"));

System.out.println(b.readLine());

2）、类.class.getResource("").getFile()：注意：字符串中有无”/”是有区别的

一、可以空字符串，

二、可以只有”/”,

三、web运行环境是”/”的话路径是在tomcat的类库Lib下

四、不可以单独拼接文件目录

**--------------------------------------------------字符串中开头有”/”是在类根路径中找--------------------------------------------------------**

Demo.class.getResource("/d.txt").getFile();

Web运行环境下下是在Tomcat中进行查找：Demo类和文件d.text都在类根目录下

/E: /apache-tomcat-7.0.78/webapps/test/WEB-INF/classes/d.txt

JAVA运行环境下是在eclipase中找：Demo类和文件d.text都在类根目录下

/E:/eclipse/eclipse-jee-4.7-win64/project/test/build/classes/d.txt

**--------------------------------------------------字符串中无”/”是在本类中下进行查找-------------------------------------------------------------------**

Demo.class.getResource("d.txt").getFile();

Web运行环境下是在Tomcat中进行查找：Demo类和文件d.text都在：com/wwl/servlet/

打印：/E:/ apache-tomcat-7.0.78/webapps/test/WEB-INF/classes/com/wwl/servlet/d.txt

JAVA运行环境下是在eclipase中找：Demo类和文件d.text都在：com/wwl/servlet/

/E:/eclipse/eclipse-jee-4.7-win64/project/text/build/classes/com/wwl/servlet/d.txt

3）、类.class.getClassLoader().getResource("").getPath();这个字符串中有无”/”都一样的

和2几乎一样的

可以空字符串，但不可以单独只有”/”，不可以单独拼接文件目录

2、获取部署在Tomcat下真实路径：this.getServletContext().getRealPath("")；Real【真实】

一、可以单独拼接文件目录名称

二、通常用于文件上传，

三、可以空字符串：找到部署在tomcat工程目录

String diskpath=this.getServletContext().getRealPath("");//空字符串：获得工程名称目录下

//打印：E:\apache-tomcat-7.0.78\webapps\test

String diskpath=this.getServletContext().getRealPath("/WEB-INF");//到WEB-inf下

# 其它

1、Integer类内部维护了一个静态的数组，该数组内部存储了-128~127这些数据，当我们创建一个Interger对象的时候，如果数据是-128~127范围之内，那么会直接从数组中获取，而不会创建新的Integer对象，目的是为了节省内存：因为这个数据是常用的：享元模式

Integer i = 127;

Integer i1=127;

System.out.println(i==i1);//打印：false 如果为127的话就打印为true

2、JDk工具类：数组Arrays， 集合Collections

获取最大：int i=Collections.max(list); 获取最小：int c=Collections.min(list); 排序Collections.sort(list);

3、This就是本类对象的引用，A a=new A() a就是对象也就是this 静态不能有this

4、int超出范围会变成负数：时间毫秒都建议是long型

int i =1000\*60\*60\*30\*50\*10000\*1000\*100000;

System.out.println(i);//打印：-446693376

5、properties文件 value存中文问题 需要转义成Unicode（万国码）

p.properties文件内容：y=\u4F60\u597D\u554A\uFF0C\u66FE\u4E3D\u65B9

\u: Unicode编码的意思

中文转换Unicode：JDk有自带的命令 :cmd输入”native2ascii” 回车 然后输入中文名后再回车就有了

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

Properties p = new Properties();

p.load(Test.class.getResourceAsStream("/p.properties"));

System.out.println(p.getProperty("y"));//打印：你好啊，曾丽方

}

6增强for循环底层还是迭代器：不可以修改数据，普通for：可以修改数据

Map<String, String[]> v=request.getParameterMap();

Collection<String[]> l=v.values();

for(String[] e:l) {

//增强for循环是不能修改的，只能进行赋值后进行重新保存；

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

e[i]= new String(e[i].getBytes("iso8859-1"),"utf-8");//直接这样就可以保存数据了，不用再进行set

}

}要明白引用问题：栈中的是固定值无法修改，再次修改是拷贝，而堆:需要内存地址(有一个或多个指向同一个内存地址）

7 StringWriter—字符串字符流是writer装饰类

作用：把流中的字符串写进此字符串输出流中

场景：现只知道自定义标签获取标签体内的内容需要使用到

/\*PageContext pc= (PageContext) getJspContext();

//FileWriter是指向文件的输出流，StringWriter是指向字符串的输出流

StringWriter sw = new StringWriter();

//把它写进此(字符串输出流)中来获取标签体的内容

getJspBody().invoke(sw);

String str=sw.toString();//获取到内容

str=str.toUpperCase();//字符串转大写方法

pc.getOut().write(str);//输出到页面

8 Eclipse的使用

快速查找类的引用

不使用全局搜索ctrl+h的方法；

1、选中该类或者接口，右键选中References

2、选中Projects,即可；

该方法还可以快速查找该类或者接口 作为另一个类的属性时被引用的情况，

用于查找Dao接口对应的Service接口 很快捷有效；

查询一个方法在哪里引用

不使用全局搜索ctrl+h的方法；

1、选中声明的方法名；

2、右键，选择Open Call Hierarchy；

快捷键

shift+ctrl+T跳出窗口 输入方法名：查询一个方法在哪个包

ctrl+alt+方向键下：快速复制本行

alt+方向键上下：快速移动本行

shift+ctrl+o：快速导包和删除没用的包

Ctrl+Shift+/  ：  注释/\* \*/

Ctrl+Shift+\  ：   消除/\* \*/注释