目录

[1.重载不能通过返回值来区分方法的不同，只能通过参数的不同来区分 2](#_Toc385352750)

[2．没必要用this的地方就没必要用 3](#_Toc385352751)

[3. XMPP用TCP传的是XML流 3](#_Toc385352752)

[4.java事件监听分析 3](#_Toc385352753)

[5.多态是多种形态，一种行为 5](#_Toc385352754)

[6．静态方法是与类,而并非与单个的对象相关联的 5](#_Toc385352755)

[7．.实例化导出类的过程 5](#_Toc385352756)

[8.对创建的对象进行计数，成员变量定义如下 5](#_Toc385352757)

[9．匿名内部类 6](#_Toc385352758)

[10.使用LinkedList是实现栈最好的方式，还可以将LinkedList向上转型为Queue 6](#_Toc385352759)

[11.永远不必为清理前一个异常对象而担心,或者说为异常对象的清理而担心。他们都是用new在堆上创建的对象,所以垃圾回收器会自动把他们清理掉 9](#_Toc385352760)

[12．使用线程池，要执行任务的线程数目不会减少，但线程池规定了每次同时执行线程的数目 9](#_Toc385352761)

[13.try块中执行了return语句，照样会执行finally语句 10](#_Toc385352762)

[14．当使用字符串拼接操作的操作(+),编译器自动引入了java.lang.StringBuilder,虽然我们在源代码中并没有StringBuilder类，但是编译器却自作主张地使用它，因为它更高效 10](#_Toc385352763)

[15．toString()的递归调用错误 10](#_Toc385352764)

[16．FancyToy.class这样做不仅更简单，而且更安全，因为它在编译器时就会受到检查(因此不需要置于try语句块中)，并且它根除了对forName()方法的调用，所以更高效 11](#_Toc385352765)

[17．类装载过程 11](#_Toc385352766)

[18．记录一个一个类产生对象的个数 11](#_Toc385352767)

[19．rg.id < id ? -1 : (arg.id == id ? 0 : 1) 将大于小于等于都涵盖了 12](#_Toc385352768)

[20．动态的 instanceof 12](#_Toc385352769)

[21．内部类实例的创建和内部静态类的创建 15](#_Toc385352770)

[22.动态代理(处理一些事务，日志) 15](#_Toc385352771)

[23．泛型化的链式栈设计 18](#_Toc385352772)

[24．获取网络状态 21](#_Toc385352773)

[25.上传文件,请求服务器数据流 21](#_Toc385352774)

[26.边下载边解压 （与网络建立建立流通道，通过这个流通道来解压） 21](#_Toc385352775)

[27.先下载到本地，再对本地文件进行解压 21](#_Toc385352776)

[28.在固定时间启动定时器, 21](#_Toc385352777)

[29．Helpers方法整理 21](#_Toc385352778)

[30 LauncherApplication 注册广播接收器，发送开机广播 22](#_Toc385352779)

[31．ChooseServer 22](#_Toc385352780)

[32．数组转化为列表 22](#_Toc385352781)

[33．在java中用 .\* 表示任意字符 22](#_Toc385352782)

[34．在Windows中 D:\workspace\projects\javatest\.\ 表示javatest中的当前目录 22](#_Toc385352783)

[35．目录使用工具(Directory) 22](#_Toc385352784)

[36．策略模式处理目录文件 24](#_Toc385352785)

[37．字节流和字符流区别 26](#_Toc385352786)

[38．Java中的字节数组(原码，补码) 26](#_Toc385352787)

[39.java中采用补码存取数据，位运算 26](#_Toc385352788)

[40．Java中字节数组和16进制字符串转换 27](#_Toc385352789)

[41.BufferedReader 31](#_Toc385352790)

[42.DataInputStream可以按照可移植的方式从流中读取基本数据类型(int,char,long等); 31](#_Toc385352791)

[43.文本写入工具 TextFile 31](#_Toc385352792)

[44.文件复制程序可以通过transferTo()和transferFrom()将一个通道和另一个通道相连 34](#_Toc385352793)

[45.压缩和解压缩 34](#_Toc385352794)

[46．通过相对目录来查找这个目录下的文件 36](#_Toc385352795)

[47.异常不能夸线程传播给主线程，只能自己内部消化 36](#_Toc385352796)

[45．线程 join 36](#_Toc385352797)

[46. Volatile关键字涵义 36](#_Toc385352798)

[47．原子性的操作明确的几个方面，哪些线程，操作的资源，操作(读取和改变) 37](#_Toc385352799)

[48．在i/o中线程操作中，可以通过关闭流的方式将线程中断，处理相应的异常或者通过检查 Thread.interrupted()来中断阻塞，不过这个标志位会被清除，所以想要再次检查是否被中断，则可以在调用Thread.interrupted()时将结果存储起来 37](#_Toc385352800)

[49.wait(),notify()以及notifyAll()要在同步方法中调用,而且调用这些方法的对象的对象锁 37](#_Toc385352801)

[50．线程错误信号（对于多个任务对应同一对象锁，当被同步的方法或者代码块一旦执行，就当它执行到底，没有间断） 37](#_Toc385352802)

[51．阻塞队列(生产者-消费者) 38](#_Toc385352803)

[52．CountDownLatch的使用 38](#_Toc385352804)

[53．CountDownLatch和CyclicBarrier的区别 40](#_Toc385352805)

[54. Semaphore(资源“许可证”) 44](#_Toc385352806)

[55.CopyOnWriteArrayList允许在列表遍历时调用remove()方法，但是获得的Iterator是不能进行set和remove操作的 44](#_Toc385352807)

[56．银行出纳员仿真(线程池的概念) 44](#_Toc385352808)

[57.SynchronousQueue(同步队列) 45](#_Toc385352809)

[58.饭店仿真 46](#_Toc385352810)

[59.Looper的注意事项 47](#_Toc385352811)

[60.条件表达式的重构 47](#_Toc385352812)

[61.委托和继承的区别 47](#_Toc385352813)

[62. 低于3.0版本使用Fragment的注意事项 47](#_Toc385352814)

[63.定义集合元素排列顺序 48](#_Toc385352815)

[64.获取小于等于这个值的一个2次幂的值 48](#_Toc385352816)

[65.获取String的boolean值 48](#_Toc385352817)

# 1.重载不能通过返回值来区分方法的不同，只能通过参数的不同来区分

# 2．没必要用this的地方就没必要用

# 3. XMPP用TCP传的是XML流

# 4.java事件监听分析

Java中的事件监听是整个Java消息传递的基础和关键。牵涉到两类对象：事件发生者和事件监听者。事件发生者是事件的起源，它可以是一个按钮，编辑框等。事件监听者就是事件的接受者，如果要想接收某个事件，它必须对该事件的发生者说一声：嗨，哥们，有事吱一声。这个过程相当与在事件发生者那儿注册了一下。当事件发生者真的有事件发生时，就会对所有对该事件注册的所有监听者送一份参考消息说：有内幕，有内幕！！事件监听者听到后，拿过来看一下，根据这份报纸各干各的事。参考消息这份报纸就是java中的事件对象。   
    具体实现呢,可以看看Button的源码。可能不好看得懂。那好我们仿照侯捷先生的作法，来模拟一个这样的事件传递：   
定义一个自己的事件:   
public   MyEvnet{   
    private   int   value;   
    public   int   getValue(){return   value;}   
    public   void   setValue(int   value){this.value=value;}   
}   
做一个接口   Listenable (怎么通知监听者，发生着和监听者协商怎么通知，发生的事件是什么，所以要定义接口)  
interface   Listenable{   
    public   void   enventChanged(MyEvent   e);   
}   
做一个事件发生者，此类中保存有所有在这里报到的类的引用   
public   MySource{   
    int   value;   
    Vector   listeners=new   Vector();   
    public   addListener(Listenable   l){listeners.add(l);}   
    public   void   setValue(int   value){   
      this.value=value;   
      fireChanged();   
    }   
    private   void   fireChanged(){   
      MyEvent   e=new   MyEvent(); //事件发生了  
      e.setValue(value);   
      for(int   i=0;i<listeners.size();i++){   
        Listenable   l=(Listenable)listeners.elementAt(i);   
        l.eventChanged(e);   
      }   
    }   
}   
好了，如果谁想监听MySource的value值改变了，就在MySource哪儿注册一下，然后写消息处理代码，就可以了,我们定义这样一个监听者：   
MyListener   implements   Listenable{   
    public   void   eventChanged(MyEvent   e){   
      System.out.println("value   changed   to:   "+e.getValue());   
    }   
}   
这样，当MySource的value真的改变时，就会触发此方法。   
    
然后在其他代码中把MyListener注册到MySource:   
MySource   ms=new   MySource();   
MyListener   ml=new   MyListener();   
ms.addListener(ml);   
ms.setValue(10);//事件发生了，通知各个监听器

# 5.多态是多种形态，一种行为

# 6．静态方法是与类,而并非与单个的对象相关联的

# 7．.实例化导出类的过程

a.调用基类构造器。这个步骤会不断地反复递归下去.

b.按声明顺序调用成员的初始化方法

c.调用导出类构造器的主体.

# 8.对创建的对象进行计数，成员变量定义如下

private static long counter = 0;

private final long id = counter++;(对该对象进行编号)

id是final的，因为我们不希望它的值在对象生命周期中被改变

# 9．匿名内部类

a.匿名内部类是一种隐藏的继承结构,所以类的初始化跟普通的导出类的初始化没有什么两样，即使是在导出类中有一个静态区域，也要先进行基类的初始化

b.用匿名内部类创建工厂方法，可以用来解决多继承的问题，内部类可以自己一个类，也可以使用外部类的成员变量

# 10.使用LinkedList是实现栈最好的方式，还可以将LinkedList向上转型为Queue

**public** **class** Stack<T> {

**private** LinkedList<T> storage = **new** LinkedList<T>();

**public** **void** push(T v)

{

storage.addFirst(v);

}

**public** T peek()

{

**return** storage.getFirst();

}

**public** T pop()

{

**return** storage.removeFirst();

}

**public** **boolean** empty()

{

**return** storage.isEmpty();

}

**public** String toString()

{

**return** storage.toString();

}

}

**public** **class** QueueDemo {

**public** **static** **void** printQ(Queue queue)

{

**while**(queue.peek() != **null**)

{

System.*out*.print(queue.remove() + " ");

}

System.*out*.println();

}

/\*\*

\* **@param** args

\*/

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

Queue<Integer> queue = **new** LinkedList<Integer>();

Random rand = **new** Random(45);

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

{

queue.offer(rand.nextInt(i + 10));

}

*printQ*(queue);

Queue<Character> qc = **new** LinkedList<Character>();

**for**(**char** c : "zhujianwen".toCharArray())

{

qc.offer(c);

}

*printQ*(qc);

}

}

Offer()方法是与Queue相关的方法之一,它在允许的情况下，将一个元素插入到队尾，或者返回false。Peek()与element()都将在不移除的情况下返回队头，但是peek()方法在队列为空时返回null，而element()会抛出NoSuchElementException()异常。Poll()和remove()方法将移除并返回队头，但是poll()在队列为空时返回null，而remove ()会抛出NoSuchElementException()异常

# 11.永远不必为清理前一个异常对象而担心,或者说为异常对象的清理而担心。他们都是用new在堆上创建的对象,所以垃圾回收器会自动把他们清理掉

# 12．使用线程池，要执行任务的线程数目不会减少，但线程池规定了每次同时执行线程的数目

package BackStage;

import java.util.concurrent.Executors;

import java.util.concurrent.ExecutorService;

public class JavaThreadPool {

public static void main(String[] args) {

// 创建一个可重用固定线程数的线程池

ExecutorService pool = Executors.newFixedThreadPool(2);

// 创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口

Thread t1 = new MyThread();

Thread t2 = new MyThread();

Thread t3 = new MyThread();

Thread t4 = new MyThread();

Thread t5 = new MyThread();

// 将线程放入池中进行执行

pool.execute(t1);

pool.execute(t2);

pool.execute(t3);

pool.execute(t4);

pool.execute(t5);

// 关闭线程池

pool.shutdown();

}

}

class MyThread extends Thread {

@Override

public void run() {

System.out.println(Thread.currentThread().getName() + "正在执行。。。");

}

}

# 13.try块中执行了return语句，照样会执行finally语句

# 14．当使用字符串拼接操作的操作(+),编译器自动引入了java.lang.StringBuilder,虽然我们在源代码中并没有StringBuilder类，但是编译器却自作主张地使用它，因为它更高效

# 15．toString()的递归调用错误

如果你希望toString()方法打印出对象的内存地址，也许你会考虑使用this关键字，当你创建了一个对象,并将其打印出来的时候，你会得到一串非常长的异常。其实，当如下代码运行时:

“InfiniteRecursion address:” + this,这里发生了自动类型转换，由InfiniteRecursion类型转换成String类型。因为编译器看到一个String对象后面跟着一个”+”，而再后面的对象不是String,于是试着将this转换成一个String。它怎么转换呢,正是通过this上的toString()，于是就发生了递归调用

# 16．FancyToy.class这样做不仅更简单，而且更安全，因为它在编译器时就会受到检查(因此不需要置于try语句块中)，并且它根除了对forName()方法的调用，所以更高效

# 17．类装载过程

a.加载：这是由类加载器执行的.该步骤将查找字节码（通常在在classpath所指定的路径中查找，但这并非是必须的），并从这些字节码中创建一个Class对象

b.链接：在链接阶段将验证类中的字节码，为静态域分配存储空间，并且如果是必须的话，将解析这个类创建的对其他类的所有引用

c.初始化：如果该类具有超类，则对其初始化，执行静态初始化和静态初始化块

# 18．记录一个一个类产生对象的个数

**public** **class** ClassInstanceCount {

**static** **class** Counter **extends** HashMap<String,Integer>

{

**public** **void** count(String type)

{

Integer quantity = get(type);

**if**(quantity == **null**)

put(type,1);

**else**

put(type,quantity + 1);

}

}

}

# 19．rg.id < id ? -1 : (arg.id == id ? 0 : 1) 将大于小于等于都涵盖了

# 20．动态的 instanceof

**public** **class** PetCount {

**static** **class** PetCounter **extends** LinkedHashMap<Class<? **extends** Pet>, Integer>

{

**public** PetCounter()

{

/\*public static final List<Class<? extends Pet>> allTypes =

Collections.unmodifiableList(Arrays.asList(

Pet.class,Dog.class,Cat.class,Rodent.class,

Mutt.class,Pug.class,Egyptianmau.class,Manx.class,

Cymric.class,Rat.class,Mouse.class,Hamster.class));

\*/

**super**(MapData.*map*(LiteralPetCreator.*allTypes*, 0));

}

/\*\*

\* 统计基类和导出类的个数

\* **@param** pet

\*/

**public** **void** count(Pet pet)

{

**for**(Map.Entry<Class<? **extends** Pet>, Integer> pair:

entrySet())

{

**if**(pair.getKey().isInstance(pet))

{

put(pair.getKey(),pair.getValue() + 1);

}

}

}

**public** String toString()

{

StringBuilder result = **new** StringBuilder("{");

**for**(Map.Entry<Class<? **extends** Pet>, Integer> pair:

entrySet())

{

result.append(pair.getKey().getSimpleName());

result.append("=");

result.append(pair.getValue());

result.append(", ");

}

result.delete(result.length() - 2, result.length());

result.append("}");

**return** result.toString();

}

}

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

PetCounter petCount = **new** PetCounter();

**for**(Pet pet : Pets.*createArray*(20))

{

System.*out*.print(pet.getClass().getSimpleName() + " ");

//传入具体实例

petCount.count(pet);

}

System.*out*.println();

System.*out*.print(petCount);

}

}

输出结果：Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat Egyptianmau Hamster Egyptianmau Mutt Mutt Cymric Mouse Pug Mouse Cymric

{Pet=20, Dog=6, Cat=9, Rodent=5, Mutt=3, Pug=3, Egyptianmau=2, Manx=7, Cymric=5, Rat=2, Mouse=2, Hamster=1}

# 21．内部类实例的创建和内部静态类的创建

前者：Out out = new Out();

Inner in = out.new Inner();

后者：Inner in = new Out.Inner();

# 22.动态代理(处理一些事务，日志)

/\*\*

\* 动态处理工具

\* **@author** Administrator

\*

\*/

**class** DynamicProxyHandler **implements** InvocationHandler

{

//被代理的类(即委托类用于执行自己的代码)

**private** Object proxied;

**public** DynamicProxyHandler(Object proxied) {

**this**.proxied = proxied;

}

@Override

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

**throws** Throwable {

//委托类执行之前，插入要做的事，比如事务

System.*out*.println("\*\*\*\* proxy:" + proxy.getClass() +

", method : " + method + ", args:" + args);

**if**(args != **null**)

{

**for**(Object arg: args)

{

System.*out*.println(" " + arg);

}

}

//委托类执行自己的方法

method.invoke(proxied, args);

//委托类执行之后

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

**return** **null**;

}

}

/\*\*

\* 消费代理

\* **@author** Administrator

\*

\*/

**class** SimpleDynamicProxy

{

**public** **static** **void** consumer(Interface iface)

{

iface.doSomething();

iface.somethingElse("bonobo");

}

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

RealObject real = **new** RealObject();

// consumer(real);

//在运行时创建的动态对象，在编译期间并没有它的字节码信息

Interface proxy = (Interface) Proxy.*newProxyInstance*(

Interface.**class**.getClassLoader(),

**new** Class[]{Interface.**class**},

**new** DynamicProxyHandler(real));

*consumer*(proxy);

}

}

# 23．泛型化的链式栈设计

**public** **class** LinkedStack<T> {

**private** **static** **class** Node<U>

{

U item;

Node<U> next;

Node()

{

item = **null**;

next = **null**;

}

Node(U item,Node<U> next)

{

**this**.item = item;

**this**.next = next;

}

**boolean** end()

{

**return** item == **null** && next == **null**;

}

}

**private** Node<T> top = **new** Node<T>();

**public** **void** push(T item)

{

top = **new** Node<T>(item,top);

}

**public** T pop()

{

T result = top.item;

**if**(!top.end())

top = top.next;

**return** result;

}

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

LinkedStack<String> lss = **new** LinkedStack<String>();

**for**(String s : "a b c".split(" "))

lss.push(s);

String s;

**while**((s = lss.pop()) != **null**)

System.*out*.println(s);

}

}

# 24．获取网络状态

DeviceNetStatusReceiver(getCurrentState)

# 25.上传文件,请求服务器数据流

ConnectionHandlerImpl

# 26.边下载边解压 （与网络建立建立流通道，通过这个流通道来解压）

Helpers 🡪 unZipPlateFile

# 27.先下载到本地，再对本地文件进行解压

Helpers 🡪 unZipRegionFile

# 28.在固定时间启动定时器,

DefaultApplicationServiceImpl

# 29．Helpers方法整理

a.获取进度框

b. 从assets中读取图片 (getImageFromAssetFile)

c.将日期对象转换成String对象 (dateToString)

d.从文件中读取字符串，以及ByteArrayOutputStream的用法 (readFileGetStr)

e.将图片流存入本地(writeImage)

f.创建桌面快捷方式 (installShortCut)

g.控制view的动画效果 (startScrollLeftAnimation)

h.显示进度框(showProgress)

i.将字符串转换成MD5字符(MD5)

j.递归删除目录(deleteDir)

k. 控制AlertDialog点击不消失(controlDialogShow)

l.获取时间的毫秒表示单位(getTimeMillions)

# 30 LauncherApplication 注册广播接收器，发送开机广播

a. ApplicationBootReceiver.APPLICATION\_BOOT 开机广播

b. "android.net.conn.CONNECTIVITY\_CHANGE" 网络变化发送的广播

31．ChooseServer

向两个服务器发送请求，获得相应时间，判断时间差，选择网络

通过全局变量得到线程执行的结果，通过while(ture)实现

# 32．数组转化为列表

Arrays.*asList*(c) (Object[] c)

# 33．在java中用 .\* 表示任意字符

# 34．在Windows中 D:\workspace\projects\javatest\.\ 表示javatest中的当前目录

# 35．目录使用工具(Directory)

**public** **final** **class** Directory {

**public** **static** File[] local(File dir,**final** String regex)

{

**return** dir.listFiles(**new** FilenameFilter() {

**private** Pattern pattern = Pattern.*compile*(regex);

@Override

**public** **boolean** accept(File dir, String name) {

**return** pattern.matcher(

**new** File(name).getName()).matches();

}

});

}

**public** **static** File[] local(String path,**final** String regex)

{

**return** *local*(**new** File(path), regex);

}

**public** **static** **class** TreeInfo **implements** Iterable<File>

{

**public** List<File> files = **new** ArrayList<File>();

**public** List<File> dirs = **new** ArrayList<File>();

@Override

**public** Iterator<File> iterator() {

**return** files.iterator();

}

**void** addAll(TreeInfo other)

{

files.addAll(other.files);

dirs.addAll(other.dirs);

}

@Override

**public** String toString() {

**return** "dirs: " + PPrint.*pformat*(dirs) +

"\n\nfiles: " + PPrint.*pformat*(files);

}

}

**public** **static** TreeInfo walk(String start,String regex)

{

**return** *recurseDirs*(**new** File(start), regex);

}

**public** **static** TreeInfo walk(File start)

{

**return** *recurseDirs*(start, ".\*");

}

/\*\*

\* 匹配任意文件

\* **@param** start

\* **@return**

\*/

**public** **static** TreeInfo walk(String start)

{

**return** *recurseDirs*(**new** File(start), ".\*");

}

/\*\*

\* 递归统计该目录下的所有文件和目录

\* **@param** startDir

\* **@param** regex

\* **@return**

\*/

**static** TreeInfo recurseDirs(File startDir,String regex)

{

TreeInfo result = **new** TreeInfo();

**for**(File item : startDir.listFiles())

{

**if**(item.isDirectory())

{

result.dirs.add(item);

result.addAll(*recurseDirs*(item,regex));

}**else**

{

**if**(item.getName().matches(regex))

result.files.add(item);

}

}

**return** result;

}

}

# 36．策略模式处理目录文件

**public** **class** ProcessFiles {

**public** **interface** Strategy

{

**void** process(File file);

}

**private** Strategy strategy;

**private** String ext;

**public** ProcessFiles(Strategy strategy,String ext)

{

**this**.strategy = strategy;

**this**.ext = ext;

}

**public** **void** start(String[] args)

{

**try** {

**if**(args.length == 0)

{

processDirectoryTree(**new** File("."));

}**else**

{

**for**(String arg : args)

{

File fileArg = **new** File(arg);

**if**(fileArg.isDirectory())

{

processDirectoryTree(fileArg);

}**else**

{

**if**(!arg.endsWith("." + ext))

arg += "." + ext;

strategy.process(

**new** File(arg).getCanonicalFile());

}

}

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

**public** **void** processDirectoryTree(File root) **throws** IOException

{

**for**(File file : Directory.*walk*(root.getAbsolutePath(),".\*\\." + ext))

strategy.process(file.getCanonicalFile());

}

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

**new** ProcessFiles(**new** ProcessFiles.Strategy() {

@Override

**public** **void** process(File file) {

System.*out*.println(file);

}

}, "class").start(args);

}

}

# 37．字节流和字符流区别

把来自于“字节”层次结构中的类和“字符”层次结构中的类结合起来使用，要用到“适配器”类：InputStreamReader可以把InputStream转换为Reader,而OutputStreamWriter可以把OutputStream转换为Writer

字符层次的类中没有方法 read(byte[] bytes)方法

Java.util.zip类库就是面向字节的而不是面向字符的

# 38．Java中的字节数组(原码，补码)

read（）方法 读出的是int型的0-255

byte是-128-127，这里用了int到byte的转化规则。214-256 = -42

# 39.java中采用补码存取数据，位运算

-1 的原码 1 000 0001 反码 1 1111110 补码 1 111 1111 hex : ff

1 的原码0 000 0001 反码 0 000 0001 补码 0 000 0001 hex: 1

补码的补码得到源码

java中都是以源码和补码来存储数据，位运算，然后展现出来的结果是将反码转换成源码用10进制表示的数据

**byte**[] b = **new** **byte**[1];

b[0] = -1;

String hex = Integer.*toHexString*(b[0] & 0xFF);

System.*out*.println(hex);

# 40．Java中字节数组和16进制字符串转换

2进制 1000 1001 1010 1011 1100 1101 1110 1111   
16进制 8 9 a(10) b(11) c(12) d(13) e(14) f(15)

一个byte用二进制表示需要8位，而用16进制需要2位字符

一个字符char占2个字节byte

一个汉字占2个字节byte

一个字母占1个字节byte

其他情况

对于汉字来说，采用gbk编码占两字节，采用utf8编码占三个字节

Java中byte用二进制表示占用8位，而我们知道16进制的每个字符需要用4位二进制位来表示。

所以我们就可以把每个byte转换成两个相应的16进制字符，即把byte的高4位和低4位分别转换成相应的16进制字符H和L，并组合起来得到byte转换到16进制字符串的结果new String(H) + new String(L)。

同理，相反的转换也是将两个16进制字符转换成一个byte，原理同上。

根据以上原理，我们就可以将byte[] 数组转换为16进制字符串了，当然也可以将16进制字符串转换为byte[]数组了

**public** **class** Hex {

**private** **static** **final** **char**[] *DIGITS\_LOWER* = {

'0','1','2','3','4','5','6','7',

'8','9','a','b','c','d','e','f',

};

**private** **static** **final** **char**[] *DIGITS\_UPPDER* = {

'0','1','2','3','4','5','6','7',

'8','9','A','B','C','D','E','F',

};

**public** **static** **char**[] encodeHex(**byte**[] data)

{

**return** *encodeHex*(data,**true**);

}

**public** **static** **char**[] encodeHex(**byte**[] data,**boolean** toLowerCase)

{

**return** *encodeHex*(data,

toLowerCase ? *DIGITS\_LOWER* : *DIGITS\_UPPDER*);

}

/\*\*

\* 0x0F:00001111

\* data[i] >>> 4 将高4位转成字符

\* **@param** data

\* **@param** toDigits

\* **@return**

\*/

**protected** **static** **char**[] encodeHex(**byte**[] data,**char**[] toDigits)

{

**int** l = data.length;

**char**[] out = **new** **char**[l << 1];

**for**(**int** i = 0,j = 0;i < l;i++)

{

out[j++] = toDigits[0x0F & data[i] >>> 4];

out[j++] = toDigits[0x0F & data[i]];

}

**return** out;

}

**public** **static** String encodeHexStr(**byte**[] data)

{

**return** *encodeHexStr*(data,**true**);

}

**public** **static** String encodeHexStr(**byte**[] data,**boolean** toLowerCase)

{

**return** *encodeHexStr*(data,

toLowerCase ? *DIGITS\_LOWER* : *DIGITS\_UPPDER*);

}

**protected** **static** String encodeHexStr(**byte**[] data,**char**[] toDigits)

{

**return** **new** String(*encodeHex*(data, toDigits));

}

/\*\*

\* f & 0xFF = 229

\* (byte) (f & 0xFF) = -27 (-27 = 229 - 256)

\* **@param** data

\* **@return**

\*/

**public** **static** **byte**[] decodeHex(**char**[] data)

{

**int** len = data.length;

**if**((len & 0x01) != 0)

{

**throw** **new** RuntimeException("Odd number of characters");

}

**byte**[] out = **new** **byte**[len >> 1];

**for** (**int** i = 0, j = 0; j < len; i++) {

**int** f = *toDigit*(data[j], j) << 4;

j++;

f = f | *toDigit*(data[j], j);

j++;

out[i] = (**byte**) (f & 0xFF);

}

**return** out;

}

**protected** **static** **int** toDigit(**char** ch, **int** index) {

**int** digit = Character.*digit*(ch, 16);

**if** (digit == -1) {

**throw** **new** RuntimeException("Illegal hexadecimal character "

+ ch+ " at index " + index);

}

**return** digit;

}

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

String srcStr = "待转换字符串";

String encodeStr = *encodeHexStr*(srcStr.getBytes());

String decodeStr = **new** String(*decodeHex*(encodeStr.toCharArray()));

System.*out*.println("转换前：" + srcStr);

System.*out*.println("转换后：" + encodeStr);

System.*out*.println("还原后：" + decodeStr);

}

}

# 41.BufferedReader

**public** **class** BufferedInputFile {

**public** **static** String read(String filename) **throws** IOException

{

BufferedReader in = **new** BufferedReader(

**new** FileReader(filename));

String s;

StringBuilder sb = **new** StringBuilder();

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

{

sb.append(s + "\n");

}

in.close();

**return** sb.toString();

}

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

File f = **new** File(".");

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

**try** {

System.*out*.println(*read*(

"./src/com/home/io/BufferedInputFile.java"));

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

# 42.DataInputStream可以按照可移植的方式从流中读取基本数据类型(int,char,long等);

# 43.文本写入工具 TextFile

/\*\*

\* 流一定要记得关，否则产生意想不到的后果

\* **@author** think in java

\*

\*/

**public** **class** TextFile **extends** ArrayList<String> {

**private** **static** **final** **long** *serialVersionUID* = -8854023807342264041L;

**public** **static** String read(String fileName)

{

StringBuilder sb = **new** StringBuilder();

**try** {

BufferedReader in = **new** BufferedReader(

**new** FileReader(

**new** File(fileName).getAbsoluteFile()));

**try** {

String s;

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

{

sb.append(s);

sb.append("\n");

}

} **finally**{

in.close();

}

} **catch** (IOException e) {

**throw** **new** RuntimeException();

}

**return** sb.toString();

}

**public** **static** **void** write(String fileName,String text)

{

**try** {

PrintWriter out =

**new** PrintWriter(

**new** File(fileName).getAbsoluteFile());

**try** {

out.print(text);

} **finally**{

out.close();

}

} **catch** (IOException e) {

**throw** **new** RuntimeException();

}

}

**public** TextFile(String fileName,String splitter)

{

**super**(Arrays.*asList*(*read*(fileName).split(splitter)));

**if**(get(0).equals(""))

remove(0);

}

**public** TextFile(String fileName)

{

**this**(fileName,"\n");

}

**public** **void** write(String fileName)

{

**try** {

PrintWriter out =

**new** PrintWriter(

**new** File(fileName).getAbsoluteFile());

**try** {

**for**(String item : **this**)

{

out.println(item);

}

} **finally**{

out.close();

}

} **catch** (IOException e) {

**throw** **new** RuntimeException();

}

}

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

String file = *read*("./src/com/home/io/HexStringTest.java");

*write*("./test.txt",file);

TextFile text = **new** TextFile("./test.txt");

text.write("./test2.txt");

}

}

# 44.文件复制程序可以通过transferTo()和transferFrom()将一个通道和另一个通道相连

File

file1 = **new** File("./test.txt"),

file2 = **new** File("./test3.txt");

FileChannel

in = **new** FileInputStream(file1).getChannel(),

out = **new** FileOutputStream(file2).getChannel();

in.transferTo(0, in.size(), out);

in.close();

out.close();

# 45.压缩和解压缩

/\*\*

\* 压缩

\* **@param** args

\* **@throws** IOException

\*/

**public** **static** **void** zip(String ...args) **throws** IOException

{

FileOutputStream f = **new** FileOutputStream("./test.zip");

ZipOutputStream zos = **new** ZipOutputStream(f);

BufferedOutputStream out = **new** BufferedOutputStream(zos);

zos.setComment("A test of Java Zipping");

**for**(String arg : args)

{

File file = **new** File(arg);

// 使用字符级别的会产生乱码内容,使用字节api虽然可以避免内容的乱码 但是避免不了名字的乱码，（ant.jar）可以解决问题

// BufferedReader in =

// new BufferedReader(

// new FileReader(

// file.getAbsoluteFile()));

FileInputStream fos = **new** FileInputStream(file);

BufferedInputStream in = **new** BufferedInputStream(fos);

zos.putNextEntry(**new** ZipEntry(file.getName()));

**int** c;

**byte**[] bytes = **new** **byte**[1024];

**while**((c = in.read(bytes)) != -1)

{

out.write(bytes,0,c);

}

in.close();

fos.close();

out.flush();

}

out.close();

}

**public** **static** **void** unzip() **throws** IOException

{

FileInputStream fi = **new** FileInputStream("./test.zip");

ZipInputStream zis = **new** ZipInputStream(fi);

BufferedInputStream bis = **new** BufferedInputStream(zis);

ZipEntry ze;

**while**((ze = zis.getNextEntry()) != **null**)

{

String filename = ze.getName();

File destFile = **new** File("./" + filename);

FileOutputStream out = **new** FileOutputStream(destFile);

**byte**[] buffer = **new** **byte**[1024];

**int** count;

**while** ((count = zis.read(buffer)) != -1) {

out.write(buffer, 0, count);

}

**if** (out != **null**) {

out.close();

}

}

bis.close();

zis.close();

fi.close();

}

# 46．通过相对目录来查找这个目录下的文件

File file = **new** File("../..", "java");

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

# 47.异常不能夸线程传播给主线程，只能自己内部消化

# 45．线程 join

主线程和线程t

回到Join，这里所说的调用方就是主线程，主线程调用线程t的Join方法，导致主线程阻塞，直到t线程执行完毕，才返回到主线程中。

简单理解，在主线程中调用t.Join()，也就是在主线程中加入了t线程的代码，必须让t线程执行完毕之后，主线程（调用方）才能正常执行。

# 46. Volatile关键字涵义

Volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

Java语言规范中指出：为了获得最佳速度，允许线程保存共享成员变量的私有拷贝，而且只当线程进入或者离开同步代码块时才与共享成员变量的原始值对比。

这样当多个线程同时与某个对象交互时，就必须要注意到要让线程及时的得到共享成员变量的变化。

而volatile关键字就是提示VM：对于这个成员变量不能保存它的私有拷贝，而应直接与共享成员变量交互。

使用建议：在两个或者更多的线程访问的成员变量上使用volatile。当要访问的变量已在synchronized代码块中，或者为常量时，不必使用。

由于使用volatile屏蔽掉了VM中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

# 47．原子性的操作明确的几个方面，哪些线程，操作的资源，操作(读取和改变)

# 48．在i/o中线程操作中，可以通过关闭流的方式将线程中断，处理相应的异常或者通过检查 Thread.interrupted()来中断阻塞，不过这个标志位会被清除，所以想要再次检查是否被中断，则可以在调用Thread.interrupted()时将结果存储起来

# 49.wait(),notify()以及notifyAll()要在同步方法中调用,而且调用这些方法的对象的对象锁

# 50．线程错误信号（对于多个任务对应同一对象锁，当被同步的方法或者代码块一旦执行，就当它执行到底，没有间断）

**synchronized**(sharedMonitor) -🡪1

{

<setup condition for T2>🡪2

sharedMonitor.notify();🡪3

}

**while**(someCondition)🡪4

{

**synchronized** (sharedMonitor) {🡪5

sharedMonitor.wait();🡪6

}

}

由于第4行没有放入同步代码块，所以执行权交给其他线程执行上面方法，恰巧第2行满足条件发出唤醒信号(由于只唤醒一次)，执行权重新交给第2个方法，将无限等待，产生死锁

# 51．阻塞队列(生产者-消费者)

BlockingQueue（控制线程的挂起和唤醒

）

Wait()和notify()必须要放在同步方法里面，并不依赖线程，而是锁对象

# 52．CountDownLatch的使用

concurrent包里面的CountDownLatch其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。   
CountDownLatch的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个CountDownLatch对象的await()方法，其他的任务执行完自己的任务后调用同一个CountDownLatch对象上的countDown()方法，这个调用await()方法的任务将一直阻塞等待，直到这个CountDownLatch对象的计数值减到0为止。   
  
      举个例子，有三个工人在为老板干活，这个老板有一个习惯，就是当三个工人把一天的活都干完了的时候，他就来检查所有工人所干的活。记住这个条件：三个工人先全部干完活，老板才检查。所以在这里用Java代码设计两个类，Worker代表工人，Boss代表老板，具体的代码实现如下：   
  
public class Worker implements Runnable{     
         
    private CountDownLatch downLatch;     
    private String name;     
         
    public Worker(CountDownLatch downLatch, String name){     
        this.downLatch = downLatch;     
        this.name = name;     
    }     
        
    public void run() {     
        this.doWork();     
        try   
{     
            TimeUnit.SECONDS.sleep(new Random().nextInt(10));     
        }catch(InterruptedException ie){     
        }     
       System.out.println(this.name + "活干完了！");     
        this.downLatch.countDown();     
             
    }     
         
    private void doWork()   
    {     
       System.out.println(this.name + "正在干活!");     
    }     
         
}   
  
==================   
public class Boss implements Runnable {     
    
    private CountDownLatch downLatch;     
         
    public Boss(CountDownLatch downLatch){     
        this.downLatch = downLatch;     
    }     
         
    public void run() {     
        System.out.println("老板正在等所有的工人干完活......");     
        try {     
            this.downLatch.await();     
        } catch (InterruptedException e) {     
        }     
        System.out.println("工人活都干完了，老板开始检查了！");     
    }     
    
}   
  
  
============================   
    
public class CountDownLatchDemo {     
    
    public static void main(String[] args) {     
       ExecutorService executor = Executors.newCachedThreadPool();     
             
        CountDownLatch latch = new CountDownLatch(3);     
             
        Worker w1 = new Worker(latch,"张三");     
        Worker w2 = new Worker(latch,"李四");     
        Worker w3 = new Worker(latch,"王二");     
             
       Boss boss = new Boss(latch);     
             
        executor.execute(w3);     
        executor.execute(w2);     
        executor.execute(w1);     
        executor.execute(boss);     
             
        executor.shutdown();     
    }     
}

# 53．CountDownLatch和CyclicBarrier的区别

1.CountDownLatch减计数，CyclicBarrier加计数。   
2.CountDownLatch是一次性的，CyclicBarrier可以重用。   
  
然后我们用被大家说烂了的跑步的例子继续说事儿：   
  
1. 有五个人，一个裁判。这五个人同时跑，裁判开始计时，五个人都到终点了，裁判喊停，然后统计这五个人从开始跑到最后一个撞线用了多长时间。

**public** **class** Race {

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

**final** **int** num = 5;

**final** CountDownLatch begin = **new** CountDownLatch(1);

**final** CountDownLatch end = **new** CountDownLatch(num);

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

**new** Thread(**new** AWorker(i, begin, end)).start();

}

// judge prepare...

**try** {

Thread.*sleep*((**long**) (Math.*random*() \* 5000));

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

System.*out*.println("judge say : run !");

begin.countDown();

**long** startTime = System.*currentTimeMillis*();

**try** {

end.await();

} **catch** (InterruptedException e) {

e.printStackTrace();

} **finally** {

**long** endTime = System.*currentTimeMillis*();

System.*out*.println("judge say : all arrived !");

System.*out*.println("spend time: " + (endTime - startTime));

}

}

}

**class** AWorker **implements** Runnable {

**final** CountDownLatch begin;

**final** CountDownLatch end;

**final** **int** id;

**public** AWorker(**final** **int** id, **final** CountDownLatch begin,

**final** CountDownLatch end) {

**this**.id = id;

**this**.begin = begin;

**this**.end = end;

}

@Override

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

**try** {

System.*out*.println(**this**.id + " ready !");

begin.await();

// run...

Thread.*sleep*((**long**) (Math.*random*() \* 10000));

} **catch** (Throwable e) {

e.printStackTrace();

} **finally** {

System.*out*.println(**this**.id + " arrived !");

end.countDown();

}

}

}  
CountDownLatch强调的是一个线程（或多个）需要等待另外的n个线程干完某件事情之后才能继续执行。 上述例子，main线程是裁判，5个AWorker是跑步的。运动员先准备，裁判喊跑，运动员才开始跑（这是第一次同步，对应begin）。5个人谁跑到终点了，countdown一下，直到5个人全部到达，裁判喊停（这是第二次同步，对应end），然后算时间。   
  
2. 继续，还是这五个人（这五个人真无聊..），这次没裁判。规定五个人只要都跑到终点了，大家可以喝啤酒。但是，只要有一个人没到终点，就不能喝。 这里也没有要求大家要同时起跑(当然也可以，加latch)。

**public** **class** Beer {

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

**final** **int** count = 5;

**final** CyclicBarrier barrier = **new** CyclicBarrier(count, **new** Runnable() {

@Override

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

System.*out*.println("drink beer!");

}

});

// they do not have to start at the same time...

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

**new** Thread(**new** Worker(i, barrier)).start();

}

}

}

**class** Worker **implements** Runnable {

**final** **int** id;

**final** CyclicBarrier barrier;

**public** Worker(**final** **int** id, **final** CyclicBarrier barrier) {

**this**.id = id;

**this**.barrier = barrier;

}

@Override

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

**try** {

System.*out*.println(**this**.id + "starts to run !");

Thread.*sleep*((**long**) (Math.*random*() \* 10000));

System.*out*.println(**this**.id + "arrived !");

**this**.barrier.await();

} **catch** (InterruptedException e) {

e.printStackTrace();

} **catch** (BrokenBarrierException e) {

e.printStackTrace();

}

}

}import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class Beer {

public static void main(String[] args) {

final int count = 5;

final CyclicBarrier barrier = new CyclicBarrier(count, new Runnable() {

@Override

public void run() {

System.out.println("drink beer!");

}

});

// they do not have to start at the same time...

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

new Thread(new Worker(i, barrier)).start();

}

}

}

class Worker implements Runnable {

final int id;

final CyclicBarrier barrier;

public Worker(final int id, final CyclicBarrier barrier) {

this.id = id;

this.barrier = barrier;

}

@Override

public void run() {

try {

System.out.println(this.id + "starts to run !");

Thread.sleep((long) (Math.random() \* 10000));

System.out.println(this.id + "arrived !");

this.barrier.await();

} catch (InterruptedException e) {

e.printStackTrace();

} catch (BrokenBarrierException e) {

e.printStackTrace();

}

}

}

* CyclicBarrier强调的是n个线程，大家相互等待，只要有一个没完成，所有人都得等着。正如上例，只有5个人全部跑到终点，大家才能开喝，否则只能全等着。   
    
  再强调下，CountDownLatch强调一个线程等多个线程完成某件事情。CyclicBarrier是多个线程互等，等大家都完成。

# 54. Semaphore(资源“许可证”)

操作系统的信号量是个很重要的概念，在进程控制方面都有应用。Java 并发库 的Semaphore 可以很轻松完成信号量控制，Semaphore可以控制某个资源可被同时访问的个数，acquire()获取一个许可，如果没有就等待，而release()释放一个许可。比如在Windows下可以设置共享文件的最大客户端访问个数。   
Semaphore维护了当前访问的个数，提供同步机制，控制同时访问的个数。在数据结构中链表可以保存“无限”的节点，用Semaphore可以实现有限大小的链表。另外重入锁ReentrantLock也可以实现该功能，但实现上要负责些，代码也要复杂些。   
下面的Demo中申明了一个只有5个许可的Semaphore，而有20个线程要访问这个资源，通过acquire()和release()获取和释放访问许可。

# 55.CopyOnWriteArrayList允许在列表遍历时调用remove()方法，但是获得的Iterator是不能进行set和remove操作的

# 56．银行出纳员仿真(线程池的概念)

PriorityQueue

(优先级队列)

CustomerLine(阻塞队列)

LinkedList(等待队列)🡪线程池

CustomerGenerator（生产者）

Teller

Teller

Teller

Customer

Customer

Customer

take

put

# 57.SynchronousQueue(同步队列)

没有任何阻塞内容的阻塞队列，每个put()都必须等待一个take(),反之亦然

 一种无缓冲的等待队列，类似于无中介的直接交易，有点像原始社会中的生产者和消费者，生产者拿着产品去集市销售给产品的最终消费者，而消费者必须亲自去集市找到所要商品的直接生产者，如果一方没有找到合适的目标，那么对不起，大家都在集市等待。相对于有缓冲的BlockingQueue来说，少了一个中间经销商的环节（缓冲区），如果有经销商，生产者直接把产品批发给经销商，而无需在意经销商最终会将这些产品卖给那些消费者，由于经销商可以库存一部分商品，因此相对于直接交易模式，总体来说采用中间经销商的模式会吞吐量高一些（可以批量买卖）；但另一方面，又因为经销商的引入，使得产品从生产者到消费者中间增加了额外的交易环节，单个产品的及时响应性能可能会降低。

SynchronousQueue(像是市场但没有代理商(集合)

消费者(当调用take()的时候到市场登记一下)

生产者(当调用put()的时候到市场登记一下)

# 58.饭店仿真





# 59.Looper的注意事项

**new** Thread() {

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

Looper.*prepare*();

Toast.*makeText*(context, "《" + paperName + "》," + downloading.getVolumelName() + " 下载完成。", 200).show();

Looper.*loop*();

}

}.start();

Looper中间的代码要另起线程，不然会堵塞当前线程

# 60.条件表达式的重构

a.将条件反转

b.以多态取代条件表达式

# 61.委托和继承的区别

继承:一旦对象创建完成，无法再改变与类型相关的行为

# 62. 低于3.0版本使用Fragment的注意事项

a.使用FragmentActivity代替Activity

b.加入Fragment

FragmentManager fragmentManager = [**getFragmentManager()**](http://developer.android.com/reference/android/app/Activity.html#getFragmentManager())**(3.0以下版本getSupportFragmentManager())**

FragmentTransaction fragmentTransaction = fragmentManager.[**beginTransaction()**](http://developer.android.com/reference/android/app/FragmentManager.html#beginTransaction());

ExampleFragment fragment = new ExampleFragment();

fragmentTransaction.add(R.id.fragment\_container, fragment);

fragmentTransaction.commit();

# 63.定义集合元素排列顺序

此列子集中的元素是map

**private** **final** **static** Comparator<Map<String, Object>> *sDisplayNameComparator* =

**new** Comparator<Map<String, Object>>() {

**private** **final** Collator collator = Collator.*getInstance*();

**public** **int** compare(Map<String, Object> map1, Map<String, Object> map2) {

**return** collator.compare(map1.get("title"), map2.get("title"));

}

};

List<Map<String, Object>> myData = new ArrayList<Map<String, Object>>();

Collections.sort(myData, sDisplayNameComparator);

# 64.获取小于等于这个值的一个2次幂的值

17：0000,0000,0000,0000,0000,0000,0001,0001 🡪 Integer.highestOneBit(n) 得到结果

16：0000,0000,0000,0000,0000,0000,0001,0000

返回的是最高位的一个1，其他全是0

# 65.获取String的boolean值

String shuf(shuf可以为空)

Boolean.*valueOf*(shuf).booleanValue()