1. if()语句的使用（这个问题我发现在代码中还是比较普遍的）

    在代码里，有时经常这样写条件语句：

    if(a.methodA()&&b.methodB()){

        ......

    }else if(a.methodA()){

        ......

    }else if(b.methodB()){

        ......

    }

    如果a.methodA()(或者b.methodB())结果为真，方法methodA()(或者methodB())将执行两次，如果methodA()(或者methodB())方法里面执行了大量代码的情况下，将严重影响代码的执行效率。当然这里只以一种情况说明，类似还有判断条件返回值为空时的情况。

2. for()语句的使用

   在代码里，有时经常这样写循环语句：

   String[] strs = new String[100];

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

         strs[i] = i+"";

   }

   这样在每次执行for循环的时候，都需要计算strs.length的值，这样就会降低循环语句的执行性能，可以首先将strs.length的值作为局部变量保存，然后使用循环语句。当然这还不是最好的执行方式。对于数组或者集合遍历，我们可以使用增强的for循环。如果一定需要使用到变量i，可以使用0值比较的方式。代码如下：

   for(int i = strs.length - 1; i >= 0;i++){

         strs[i] = i+"";

   }

3. 有关流的读写操作(以字节流为例)

   经常我们会这样直接对流进行读写：

   FileInputStream fis = new FileInputStream(new File("aaa.txt"));

   FileOutputStream fos = new FileOutputStream(new File("bbb.txt"));

   int bytedata = -1;

   while((bytedata = fis.read())!=-1){

        fos.write(bytedata);

   }

   如果我们使用自己定义的字节数组作为缓冲区，具体代码如下：

   FileInputStream fis = new FileInputStream(new File("aaa.txt"));

   FileOutputStream fos = new FileOutputStream(new File("bbb.txt"));

   byte[] buffer = new byte[1024]; //自定义缓冲区，通常定义为512的倍数

   int length = -1;

   while((length = fis.read(buffer))!=-1){

fos.write(buffer,0,length);

   }

   对于采用自定义缓冲区的方式读取与写入数据，性能还是有很大的提升，特别是在文件数据比较大的情况下。

4. 有关流关闭问题

   经常我们在流进行包装或者转化的时候写下面的代码：

   BufferedReader  reader = new BufferedReader(new InputStreamReader(new FileInputStream(new File("aaa.txt"))));

   reader.close();

   很多资料或者书上都这么写，开发中也有很多时候我们这样用，当我们执行reader.close()时，BufferedReader()里面new出来的流都会不会自动关闭，这个我自己也没有完全弄清楚，不过在开发中我们最好还是单步使用流操作，以便流能及时关闭。

5. 有关HashMap

   有关HashMap的使用存在下面几个问题：

   （1）当需要放入到HashMap里面的元素比较多时，最好定义HashMap的初始值。

    这里从HashMap的底层实现来看的，HashMap的底层实现是一个数组，这个数组里面的每一个元素是一个链表，HashMap默认的数组容量为16，装载因子为0.75，当放入HashMap中的元素的数量超过16\*0.75=12时，系统就会对HashMap的容量进行扩充（扩充为原来的两倍），在扩充的同时，还会对里面的HashMap元素进行重新的hash运算，如果你放入HashMap中的元素数量很大，这样HashMap将不断扩充容量，不断进行哈希运算，这样带来的比较大的内存消耗。严重影响系统的性能。

   （2）有关HashMap的遍历

    对HashMap的遍历有很多种方式，经常我们都是先通过调用HashMap的keySet()方法获取键的集合，然后对键值进行迭代，获取对应值。而更加高效的使用方式是：使用HashMap的entrySet()获取键值对的集合，该集合里面包含的元素是Map.Entry，然后通过调用Map.Entry方法里面的getKey方法和getValue方法分别获取键和值。关于其中的原因，主要是HashMap的底层实现就是使用Entry对象存放每一个元素。

   （3） HashMap是线程不安全的，不能被多个线程共享使用，如果想在线程安全条件下使用，可以使用ConcurrentHashMap代替。

    注：HashMap和ConcurrentHashMap类的底层实现方式以及代码都是比较经典的，尤其是ConcurrentHashMap使用锁分离的方式解决了HashMap的线程不安全的问题，更是经典中的经典，有时间大家可以自己研究，如果他们的底层实现都弄清楚了，开发的时候就可以避免很多并发与性能上面的问题。

6.有关单例模式的使用

    在编写代码的时候，如果要使用单例模式，最好使用饿汉式的单例方式，代码如下：

    public class Singleton {

private static final Singleton instance = new Singleton();

private Singleton(){

}

public static  Singleton getInstance(){

return instance;

}

     如果使用懒汉式的单例方式，代码如下：

     public class Singleton {

    private static  Singleton instance = null;

    private Singleton(){

    }

    public static  Singleton getInstance(){

              if(instance == null){

                  return new Singleton();

              }

    }

    对于懒汉式的单例，是线程不安全的，在并发的时候会产生一些问题，虽然可以使用双重检查枷锁等机制可以实现懒汉式的线程安全，但是运行效率并不是很高。具体细节可以在网上查阅相关资料。

7.有关网页的抓取

  在我们的项目中，经常我们抓取网页内容是通过HttpClient的方式，当然走我们的责任链时，对抓取出来的网页各种因素都需要考虑，还是比较恰当的，但是，如果我们只需要获取网页的内容或者网页中某些特殊节点，可以直接借助JDK为我们提供的类URLConnection，这样做更能提高效率，因为屏蔽掉了HttpClient需要做的一些额外的工作。在RSS和阅读模式的实现中我就是采用JDK提供的URLConnection来抓取网页内容的。

**8.Java堆和栈的区别**

Java中把内存空间分为两种：**栈空间**和**堆空间**。

|  |  |  |
| --- | --- | --- |
|  | **栈（stack）** | **堆（heap）** |
| 特点 | 栈中主要存放一些基本数据类型的变量(int,short,long,byte,float,double,Boolean,char)和对象的引用（对象的句柄）。  String s1 = “abc”;  String s2 = “abc”;  S1 == s2 | 堆中主要存放的是由new创建的对象和数组。  String s1 = new String(“abc”);  String s2 = new String(“abc”);  S1 != s2 |
| 优点 | 存取速度比堆快，仅次于寄存器，栈数据可以共享。  Int a = 3; int b = 3;会出现a和b共同指向3的情况。 | 可以动态分配内存的大小，生存周期也不必事先告诉编译器，因为他是在运行时动态分配内存的，Java垃圾回收器会自动收走这些不用的数据。 |
| 缺点 | 存在栈中的数据大小和生存周期必须是确定的，缺乏灵活性。 | 由于要在运行时动态分配内存，所以存取的速度比较慢。 |
| 共同点 | 堆和栈都是Java用来在Ram中存放数据的地方。与C++不同，Java自动管理堆和栈，程序员不能直接的设置堆和栈。 | |

**9.设计模式之单例模式**

（1）线程不安全的懒汉模式

Public class Singleton{

Private static Singleton instance;

Public static Singleton getInstance()

{

If(instance == null)

{

Return instance;

}

}

}

（2）线程安全的懒汉模式

Public class Singleton{

Private static Single instance;

Public static synchronized Singleton getInstance()

{

If(instance == null)

{

Instance = new Singleton();

}

Return instance;

}

}

（3）饿汉模式（线程安全）：类加载的时候实例化对象，避免了线程不安全

Public class Singleton{

Private static Singleton instance = new Singleton();

Public static Singleton getInstance()

{

Return instance;

}

}

（4）变种饿汉模式（线程安全）

Public class Singleton{

Private static Singleton instance = null;

Static

{

Instance = new Singleton();

}

Public Singleton getInstance()

{

Return this.instance;

}

}

（5）静态内部类模式

Public class Singleton{

Private static class SingletonHolder{

Private static final Singleton INSTANCE = new Singleton();

}

Public static final Singleton getInstance()

{

Return SingletonHolder.INSTANCE;

}

}

（6）枚举

Public enum Singleton{

INSTANCE;

Public void whateverMethod(){}

}

**10.关键字this的注意点**

如果要调用成员变量或者方法，需要在前面加上this关键字，不加也能跑，但是不够清晰，加上this这个关键字，就是要调用本类中的成员变量或者方法，而不是本方法中的局部变量，还有super方法也一样，是调用父类的成员变量或者方法。

**11.ArrayList和HashMap注意啊点**

如果定义成类的成员变量，在方法调用时一定要先clear，防止数据混乱。

**12.if else和三元运算比较**

三元运算更有效率。

**13.Java序列化的机制和原理**

Serialization（序列化）是将一种对象以一连串的字节描述的过程。

Deserialzation（反序列化）是将这些字节重建成一个对象的过程。

序列化的必要性：

Java中，一切都是对象，在分布式环境下经常需要将Object从这一段的网络或者设备传送到另一端，这就需要有一种可以在两端传输的协议。

如何序列化一个对象：

实现Serializatable接口，Serializable接口没有方法，更像是一个标记。

Java的序列化算法：

1. 将对象实例相关的类元数据输出
2. 递归的输出类的超类描述直到不再有超类
3. 类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值
4. 从上至下递归输出实例的数据

Code:

**public** **class** SerialTest **implements** Serializable{

**public** **byte** version = 100;

**public** **byte** count = 0;

@Test

**public** **void** createTest() **throws** Exception

{

FileOutputStream fos = **new** FileOutputStream("D://temp.out");

ObjectOutputStream oos = **new** ObjectOutputStream(fos);

SerialTest serialTest = **new** SerialTest();

oos.writeObject(serialTest);

oos.flush();

oos.close();

}

@Test

**public** **void** createTest2() **throws** Exception

{

FileInputStream fis = **new** FileInputStream("D://temp.out");

ObjectInputStream ois = **new** ObjectInputStream(fis);

SerialTest serialTest = (SerialTest) ois.readObject();

System.*out*.println("version=" + serialTest.version);

}

}

**14.时间服务器搭建**

一台服务机，N台客户机

Exam:以192.168.193.12作为服务机，192.168.193.13～44作为客户机

（1）在服务器上安装ntp server

下载：ntp-4.2.6.tar.gz

解压：tar zxvf ntp-4.2.6.tar.gz

安装：cd ntp-4.2.6

./configure --prefix=/usr/local/ntp

--enable-all-clocks

--enable-parse-clocks

Make

Male install

配置：修改配置文件ntp.conf vi /etc/ntp.conf

添加restrict default nomodify

restrict 192.168.193.0 mask 255.255.255.0 nomodify

启动：/usr/sbin/ntpd –c/etc/ntp.conf –p /tmp/ntpd.pid

修改时间：date –s 14:26:29

（2）配置客户机

下载：ntpdate.tar.gz到/usr/sbin

解压：tar zxvf ntpdate.tar.gz

配置：crontab –u root –e

\* \* \* \* \* /usr/sbin/ntpdate 192.168.193.12

启动：/etc/init.d/cron start

**15.Linux命令集合**

（1）清缓存

杀进程：kill -9

启缓存：memcached –d –m 2048 –u root –p 11211

（2）查看nohuo日志

Tail –f nohup.out 实时查看

Tail –100 nohup.out 查看最后100行

（3）26服务器重启agent

./stop.sh ./start.sh

（4）编辑模式查看nohuo.out

Vim nohup.out

/Exception

N键 代表下一个

（5）连接备机

119.145.9.208 root hwhst1!

Ssh 10.1.3.217 root hwhst1!

**16.正则表达式研习**

**1．正则表达式的作用**

正则表达式是记录文本规则的代码

**2.元字符**

（1）\b 代表着单词的开头过结尾，它只匹配一个位置

\bhi\b

（2）. 除了换行符之外的任意字符

\bhi\b.\*\bLucy\b

（3）\* 代表数量，\*前面的内容可以连续使用任意次

（4）\d 匹配一个数字

0\d{2}-\d{8} {2}{8}的意思是前面的\d必须连续的重复2次和8次

（5）\s 匹配任意空白符（包括空格、Tab、换行符、中文全角空格）

（6）\w 匹配字母或下划线或汉字

\ba\w\*\b

（7）^ 匹配字符的开始

（8）$ 匹配字符的结束

^\d{5,12}$ {5,12}重复次数不能少于5大于12

**3.重复限定符**

（1）\* 重复0次或更多次

（2）+ 重复1次或更多次

（3）? 重复0次或1次

（4）{n} 重复n次

（5）{n,}重复n次或更多次

（6）{n,m} 重复n到m次

**4.字符类**

[aeiou] 匹配任意一个英文元音字符

[.?!] 匹配标点符号

[0-9] 匹配数字

[a-z0-9A-Z] 匹配字母或数字

**5.分枝条件**

0\d{2}-\d{8}|0\d{3}-\d{7} 匹配两种以连字符分隔的号码

**6.分组**

(\d{1,3}\.){3}\d{1,3} (\d{1,3}\.){3}匹配三位数字加上一个英文.号，这个分组重复3次

**7.反义**

\W 匹配任意不是字母、数字、下划线、汉字的字符

\S 匹配任意不是空白符的字符

\D 匹配任意非数字的字符

\B 匹配不是开头或结束的位置

[^x] 匹配除了x意外的任意字符

[^aeiou] 匹配除了aeiou这几个字母以外的任意字符

## 17.Java线程池研习

1.为什么要用线程池？

诸如Web服务器、数据库服务器、文件服务器或者邮件服务器之类的许多服务器应用程序都面向处理来自某些远程来源的大量短小的任务。

请求以某种方式到达服务器，这种方式可能是通过网络协议（例如：Http、Ftp、POP）、JMS队列或者可能通过轮询数据库。

不管请求如何到达，服务器应用程序中经常出现的情况是：单个任务处理的时间很短而请求的数据却是巨大的。

2.线程池解决了什么问题？

构建服务器应用程序的一个简单的模型是：

每当一个请求到达的时候就创建一个新的线程，然后在线程中为请求服务。

每个请求对应一个线程的方法的不足之处在于：

1. 为每个请求创建一个线程的开销过大。
2. 为每个请求创建新线程的服务器在创建和销毁线程上花费的时间和消耗的资源要比花在处理实际的请求的时间和资源更多。
3. 活动的线程也消耗系统的资源。在一个JVM里创建太多的线程可能会导致由于过度消耗内存而用完内存或“切换过度”。

3.使用线程池有什么风险

（1）死锁

任何的多线程应用程序都有死锁的风险。

（2）资源不足

在恰当的调整了线程池的大小时，线程池执行的很好。

线程消耗包括内存和其他系统资源在内的大量资源。除了Thread对象所需的内存之外，每个线程都需要两个可能很大的执行调用堆栈。

如果线程池太大，那么被那些线程消耗的资源可能严重的影响系统的性能。在线程之间进行切换将会浪费时间，而且使用超出比你实际需要的线程可能会引起资源匮乏问题，因为此线程正在消耗一些资源，而这些资源可能会被其它任务更有效的利用。除了线程自身所使用的资源之外，服务请求时所作的工作可能需要其他的资源，例如：JDBC，套接字或连接。

（3）并发错误

（4）线程泄漏

当从池中除去一个线程以执行一项任务，而在任务完成后该线程却没有返回池中，会发生这种情况。

发生线程泄漏的一种情形出现在一个RuntimeException或一个Error时。如果池没有捕捉到它们，那么线程只会退出而线程池的大小将会永远减少一个。当这种情况发生的次数足够多时，线程池最终就为空，而且系统将停止，因为没有可能的线程来处理任务。

（5）请求过载

4．怎样有效的使用线程池

（1）不要对那些同步等待其他任务结果的任务排队。这可能会导致死锁，在这种死锁中，所有线程都被一些任务所占用，这些任务一次等待排队任务的结果，而这些任务又无法执行，因为所有的线程都很忙。

（2）在为时间可能很长的操作使用合用的线程时要小心。如果程序必须等待诸如IO完成这样的某个资源，那么请指定最长的等待时间，以及随后是失效还是将任务重新排队以便稍后执行。

（3）理解任务。要有效的调整线程池的大小，需要理解正在排队的任务以及它们在做什么。

5. 线程池总结

线程池的作用：

限制系统中执行线程的数量。

根据系统的环境情况，可以自动或者手动设置线程数量，达到最佳的运行效果；少了浪费系统资源，多了造成系统拥挤效率不高。

用线程池控制线程数量，其他线程排队等候，一个任务执行完毕，再从队列中取最前面的任务执行，若队列中没有等待线程，线程池的这一资源处于等待。当一个新的任务需要运行时，如果线程池中有等待的工作线程，就可以开始运行了，否则进入等待队列。

为什么要用线程池：

减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

可以根据系统的承受力，调整线程中工作线程的数目，防止因为消耗过多的内存，而把服务器累的趴下(每个线程需要大约1MB内存，线程开的越多，消耗的资源越多)。