|  |  |
| --- | --- |
| 积德累仁  福生于微 | 引言  人生在世匆匆几十年，生命是极其有限的。我们只有树立明确的奋斗目标，才会为行动指出正确的方向，才会在实现目标的过程中少走弯路；我们只有树立明确的目标，才能集中精力在某一职业领域，或某一方面取得较好较快的发展，才能突破自我，形成优势；我们只有树立明确的目标，才能有更积极向上的心态，求知探索的动力，才能不畏艰难险阻，永攀高峰，不畏困难挫折，奋勇前进，才能更好地激发潜能，让生命冲破时空的极限，释放出超强的能量，让自己的人生绚烂绽放。  与君共勉  凡事不要说“我不会”或“不可能”，因为你根本还没有做！ |

|  |  |  |
| --- | --- | --- |
| 更新记录 | | |
| **提交时间** | **提交设备** | **修改内容** |
| 2019/03/29 | 酷比魔方 | 合并内容 |
| 2019/03/31 | 联想G400 | 算法 |
| 2019/04/03 | 酷比魔方 | 开源框架 |
| 2019/04/17 | 酷比魔方 | 集合 |
| 2019/04/21 | 联想G400 | Java核心技术、CentOS |
| 2019/04/30 | 酷比魔方 | 数据库 |
| 2019/05/04 | 联想G400 | SQL试题 |
| 2019/05/10 | 酷比魔方 | 添加封面、算法导论 |
| 2019/05/17 | 联想G400 | 设计模式、多线程 |
| 2019/06/18 | 联想G400 | 开源框架 |
| 2019/06/21 | 联想G400 | 数据库 |
| 2019/06/30 | 联想G400 | 引入打开算法之门、Oracle+11g+从入门到精通 |
| 2019/07/11 | 联想G400 | 引入架构师 、数据结构 |
| 2019/08/22 | ThinkPad E450 | 引入人工智能、加入GitHub |
| 2019/08/26 | ThinkPad E450 | 引入redis开发与运维 |
| 2019/09/10 | 联想G400 | 引入Docker |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# 一、JAVA基础

## 一、Java核心技术Ⅰ

### 第一章、java程序设计概述

### 第二章、java程序设计环境

### 第三章、java的基本程序设计结构

#### 1、数据类型

**整型**：

int 4字节

short 2字节

long 8字节

byte 1字节

**浮点类型**：

double 8字节

float 4字节

**char类型**：

\b 退格

\t 制表

\n 换行

\r 回车

**boolean类型**：

#### 2、变量

(1)、变量必须初始化

(2)、常量：final关键字修饰只能赋值一次。

#### 3、字符串

(1)、子串s.substring(a,b) 截取子串

(2)、+号拼接字符串

(3)、空串：String s="" //是一个java对象，有自己长度和内容（空）

(4)、Null串：表示没有任何对象与该变量关联。

##### java.lang.string

**char** charAt(**int** index)//返回给定位置的代码单元

**int** codePointAt(**int** index) //返回给定位置的开始或结束的代码点（ASCLL码）

**int** offsetByCodePoints(**int** startIndex,**int** cpCount)//距离第startIndex个码点 cpCount 个码点（cpCount可正可负）是 相对于第0个码点第几个码点

**int** compareTo(String other)//按照字典顺序，如果字符串位于other之前返回一个负数，否则返回正数，相等返回0

**boolean** endWith(String suffix) //如果字符串以suffix结尾返回true

**boolean** equals(Object other) //

**boolean** equalsIgnoreCast(String other)

**int** indexOf(String str)

**int** indexOf(String str,**int** fromIndex)

**int** indexOf(**int** cp) //cp为字符编码

**int** indexOf(**int** cp,**int** fromIndex)//返回字符串或代码点cp匹配的第一个子串的开始位置，位置从0或fromIndex开始计算

**int** lastIndexOf(String str)

**int** lastIndexOf(String str,**int** fromIndex)

**int** lastIndexOf(**int** cp)

**int** lastIndexOf(**int** cp,**int** fromIndex)

**int** length()

**int** codePointCount(**int** startIndex,**int** endIndex)

//返回startIndex和endIndex-1之间的代码点数量，准确计算unicode字符的数量，而不是char的数量

String replace(CharSequence oldString,CharSequence newString)

**boolean** startsWith(String prefix) //如果字符串以prefix开头返回true

String subString(**int** beginIndex)

String subString(**int** beginIndex,**int** endIndex)

String toLowerCast()

String toUpperCast()

String trim()//删除开头结尾空格

##### 构建字符串

字符串连接效率低，每次连接字符串，都会构建一个新的String对象耗时耗空间

StringBuilder builder = **new** StringBuilder();

builder.append(ch);

String str=builder.toString();

##### java.lang.StringBuilder

StringBuilder()//构造一个空的字符串构建器

**int** length() //返回构建器或缓冲器中的代码单元数量

StringBuilder append(String str)//追加一个字符串并返回this

StringBuilder append(**char** c)//追加一个代码单元并返回this

StringBuilder appendCodePoint(**int** cp)//追加一个代码点。并将其转换为一个或两个代码单元并返回this

**void** setCharAt(**int** i,**char** c)//将第i个代码单元设置为c

StringBuilder insert(**int** offset,String str)

StringBuilder insert(**int** offset,**char** c)

StringBuilder delete(**int** startIndex,**int** endIndex)

String toString()

#### 4、输入输出

##### 读取控制台输入

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

String str=in.readLine();//读入一行

String str=in.next();//读取一个单词

**int** age=in.nextInt() //读取整数

doubloe d=in.nextDouble();

**boolean** hasNext()

**boolean** hasNextInt()

**boolean** hasNextDouble()

##### 文件输入与输出

Scanner in =**new** Scanner(Path.get("c:\\mydirectory\\myfile.txt");

PrintWriter out=**new** PrintWriter("myfile.txt");

**break**(带标签break）

#### 5、大数值

##### java.Math

###### .java.Math.BigInteger处理包含任意长度数字序列的数值

BigInteger add(BigInteger other)

BigInteger subtract(BigInteger other)

BigInteger multiply(BigInteger other)

BigInteger divide(BigInteger other)

BigInteger mod(BigInteger other)

**int** compareTo(BigInteger)

**static** BigInteger valueOf(**long** x) //返回 值等于x的大整数

BigInteger a=BigInteger.valueOf(100);

BigInteger c = a.add(b);

BigInteger d = c.multiply(b.add(BigInteger.valueOf(2)));

###### (2).java.Math.BigDecimal 处理任意精度的浮点数运算

**static** BigDecimal valueOf(**long** x)

**static** BigDecimal valueOf(**long** x,**int** scale)//返回值为x或x/10sacle次方的一个大实数

#### 数组

##### java.util.Arrays

**static** String toString(type[]a) //输出[a,b,c,d]

**static** type copyOf(type[]a,**int** length )

**static** type copyOf(type[]a,**int** start,**int** end)

**static** **void** sort(type[]a)

**static** **int** binarySearch(type[],type v)

**static** **int** binarySearch(type[],**int** start,**int** end ,type v)

**static** **void** fill(type[] a,type v) //将数组所有数据元素值设置为v

**static** **boolean** equals(type[]a,type[]b) //要求数组元素的个数必须相等，而且要求对应位置的元素也相等

**int** []a=**new** **int**[100];

Array.sort(a);

### 第四章、对象与类

### 第十章、部署应用程序和applet

#### 1、JAR文件

既可以包含类文件，也可以包含图像和声音等文件，采用zip技术压缩。

（1）清单文件MANIFEST.MF

（2）可运行JAR文件

（3）资源

#### 2、Java Web Start

（1）Java Web Start是一项在Internet上面发布应用程序的技术

（2）通过浏览器发布，下载到本地即可启动

（3）显示在浏览器之外的属于自己的框架中

#### 3、applet

#### 4、应用程序首选项存储

（1）属性映射Properties

### 第十一章、异常、断言、日志和调试

## Java核心技术Ⅱ

### 第一章、流与文件

#### 1、流

##### (1)读写字节

###### 1.java.io.InputStream

**abstract** **int** read() //从数据中读取一个字节结尾返回-1

**int** read(**byte**[] b) //读取一个字节数组，返回字节数，读到流结尾返回-1

**int** read(**byte**[] b, **int** off, **int** len) //读取一个字节数组

**long** skip(**long** n) //在输入流中跳过n个字节，返回实际跳过字节数（有可能小于n）

**int** available() //返回在不阻塞的情况下可获取的字节数

**void** close() //关闭输入流

**void** mark(**int** readlimit) //在输入流当前位置打一个标记

**void** reset()//返回最后一个标记，随后对read的调用将重新读入这些字节，如果当前没有任何标记，则这个流不被重置

**boolean** markSupported() //如果这个流支持打标记，则返回true

###### 2.java.io.OutputStream

**abstract** **void** write(**int** n) //写出一个字节的数据

**void** write(**byte** []b)

**void** write(**byte**[]b,**int** off,**int** len)//写出所有字节或某个范围的字节到数组b中

**void** close() //冲刷并关闭输出流

**void** flush() //冲刷输出流，即将所有缓冲的数据发送到目的地

##### (2)完整的流家族

###### 3.java.io.Closeable

**void** close() //关闭这个Closeable，可能会抛出IOException

###### java.io.Flushable

**void** flush() //冲刷这个Flushable

###### 5.java.lang.Readable

**int** read(CharBuffer cb)//尝试着向cb读入其可能持有数量的char值

###### 6.java.lang.Appendable

Appendable append(**char** c)

Appendable appenbd(CharSequence cs)//向这个Appendabled中追加给定的码元或给定序列中所有码元

###### 7.java.lang.CharSequence

**char** charAt(**int** index) //返回给定索引处的码元

**int** length()//返回在这个序列中的码元的数量

CharSequence subSequence(**int** startIndex,**int** endIndex)//返回由存储在startIndex到endIndex-1处的所有码元构成的CharSeqence

##### 组合流过滤器

###### java.io.FileInputStream

FileInputStream(String name)

FileInputStream(File file)//使用由name字符串或file对象指定路径名的文件创建一个新的文件输入流

###### java.io.FileOutputStream

FileOutputStream(String name)

FileOutputStream(String name,**boolean** append)

FileOutputStream(File file)

FileOutputStream(File file,**boolean** append)//使用由name字符串或file对象指定路径名的文件创建一个新的输出流

###### java.io.BufferedInputStream

BufferedInputStream(InputStream)//创建一个带缓冲区的流

###### java.io.BufferOutputStream

BufferOutputStream(OutputStream out)//创建一个带缓冲区的流

###### java.io.PushbackInputStream

PushbackInputStream(InputStream in)

PushbackInputStream(InputStream in,**int** size)//构建一个可以预览一个字节或者具有指定尺寸的回推缓冲区的流

**void** unread() //回推一个字节，它可以在下次调用read时再次被获取

#### 2、文本的输入与输出

##### (1)写出文本

###### java.io.PrintWrite

PrintWrite (Write out)

PrintWrite (Write out ,**boolean** autoFlush)//创建一个新的PrintWriter

PrintWrite(OutputStream out)

PrintWrite(OutputStream out,**boolean** autoFlush)//通过创建必须的中介OutputStreamWriter从已有的OutputStream中创建一个新的PrintWriter

PrintWrite(String filename)

PrintWrite(File file)//通过创建必须的中介FileWriter创建一个向给定文件写出的新的PrinteWriter

**void** print (Object obj) //通过打印toString产生的字符串来打印一个对象

**void** print(String s) //打印一个包含Unicode码元的字符串

**void** println(String s) //打印一个字符串，后面紧跟一个行终止符

**void** print(Char[]s) //打印在给定的字符串中的所有Unicode码元

**void** print (**char** c ) //打印一个Unicode码元

**void** print (**int** i)

**void** print (**long** l)

**void** print (**float** f)

**void** print (**double** b)

**void** print(**boolean** b)//以文本格式打印给定的值

**void** printf(String format,Object...args)//按照格式字符串指定的方式打印给定的值

**boolean** checkError()//如果产生格式化或输出错误，则返回true

##### (2)读入文本

##### (3)以文本格式存储对象

##### (4)字符集

#### 3、读写二进制数据

#### 4、ZIP文档

#### 5、对象流与序列化

#### 6、操作文件

#### 7、内存映射文件

#### 8、正则表达式

### 第二章、XML

#### 1、Xml概述

#### 2、解析XML文档

#### 3、验证XML文档

#### 4、使用XPath来定位信息

#### 5、使用命名空间

#### 6、流机制解析器

#### 7、生成XML文档

#### 8、XSL转换

### 第三章、网络

#### 1、连接到服务器

##### (1)套接字超时

套接字读取信息时，在有数据可供访问之前，读操作将会被阻塞，应设置合理的超时值

Socket s = **new** Socket(..);

s.setSoTimeout(10000); //超时无连接即给出SocketTimeOutException

**java.net.Socket**

Socket() //创建一个还未被连接的套接字

**void** connect(SocketAddress addresss) //将套接字连接到给定的地址

**void** connect(SocketAddress address,**int** timeoutInMilliseconds)//如果在给定的时间内无响应，则返回

**boolean** isConnected() //如果该套接字已被连接，则返回true

**boolean** isClosed() //如果套接字已经被关闭，则返回true

InputStream getInputStream()

OutputStream getOutputStream()//获取可以从套接字中读取数据的流，以及可以向套接字写出数据的流

##### (2)因特网地址

**java.net.InetAddress**

**static** InetAddress getByName(String host)

**static** InetAddress[]getAllByName(String host)//为给定的主机名创建一个InetAddress对象

**static** InetAddress getLocalHost()//为本地主机创建一个InetAddress对象

**byte**[]getAddress() //返回一个包含数字型地址的字节数组

String getHostAddress()//返回一个由十进制数组成的字符串，圆点隔开

String getHostName()//返回主机名

#### 2、实现服务器

**java.net.ServerSocket**

ServerSocket(**int** port) //创建一个监听端口的服务器套接字

Socket accept() //等待连接，阻塞线程直到建立连接为止

**void** close() //关闭服务器套接字

##### (1)为多个客户端服务

每建立一个连接开启一个线程

**public** **class** SimpleServerOfClients {

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

**int** i=1;

ServerSocket server=**new** ServerSocket(8189);

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

Socket client=server.accept();

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

Runnable r = **new** threadedEchoHandler(client);

Thread t=**new** Thread(r);

t.start();

i++;

}

}

}

**public** **class** threadedEchoHandler **implements** Runnable {

**private** Socket client;

**public** threadedEchoHandler(Socket i) {

client=i;

}

@Override

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

**try** {

InputStream ins=client.getInputStream();

OutputStream outs=client.getOutputStream();

Scanner in=**new** Scanner(ins);

PrintWriter out=**new** PrintWriter(outs, **true**);

System.***out***.println("Hello! Enter bye to exit!");

**boolean** done=**false**;

**while**(!done&&in.hasNextLine()) {

String line = in.nextLine();

System.***out***.println("回应：" + line);

**if** (line.trim().equalsIgnoreCase("bye")) {

done=**true**;

}

}

client.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

##### (2)半关闭

一端终止输入，但可接受另一端的数据，即停止请求，保持响应

**java.net.Socket**

**void** shutdownOutput() //将输出流设为流结束

**void** shutdownInput() //将输入流设为流结束

**boolean** isOutputShutdown() //如果输出已关闭，返回true

**boolean** isInputShutdown() //如果输入已关闭返回true

#### 3、可中断套接字

###### java.net.InetSocketAddress

InetSocketAddress（String hostname,**int** port)//通过主机和端口参数创建一个地址对象，并解析主机名，如果主机名不能被解析那么将该地址对象的unresolved的属性设为true

**boolean** isUnresolved() //如果不能解该地址对象，则返回true

###### java.nio.channels.SocketChannel

**static** SocketChannel open(SocketAddress address)//打开一个套接字通道，并将其连接到远程地址

###### java.nio.channels.Channels

**static** InputStream newInputStream(ReadableByteChannel channel)//创建一个输入，用以从指定通道读取数据

**static** OutputStream newOutputStream(WritableByteChannel channel)//创建一个输出流，用以向指定的通道写入数据

#### 4、获取Web数

###### java.net.URL

InputStream openStream() //打开一个用于读取资源数据的输入流

URLConnecction openConnection() //返回一个URLConnection对象

该对象负责管理与资源之间的连接

###### java.net.URLConnection

**void** setDoInput(**boolean** doInput)

**boolean** getDoInput()//如果doInput为true那么用户可以接收来自该URLConnection的输入

**void** setDoOuput(**boolean** doOutput)

**boolean** getDoOutput()//如果doOutput为true，用户可以将输出发送到该URLConnection

#### 5、发送E-mail

### 第四章、数据库编程

## javaSE初级教程视频学习笔记

### 第十三章

#### 13.1 内部类概述

在类中或方法中定义的类,包含内部类的类叫做外部类

Outer out=**new** Outer ();

Outer.Inter in=**new** out.**new** Inter();//内部类相当于外部类的成员变量或方法。

#### 13.2 匿名内部类

**public** **class** TestInter{

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

OuterI outer=**new** OuterI{ //用匿名类实现OuterI接口

@Override

**public** **void** show(){

}

}

}

}

**interface** OuterI{

**public** **void** show(){

}

}

#### 13.3 局部内部类

在局部定义的类叫做局部内部类（在方法中定义），在局部内部类可以访问外部类的变量；但想要访问方法的参数或方法内定义的变量必须声明为final类型，因为局部内部类的生命周期和变量的不一样（变量在栈中，对象在堆中）

#### 13.4 静态内部类

1.在内部类前添加static修饰符。

2.在静态内部类中可以声明静态成员变量，非静态内部类中不可以。

3.静态内部类不可以使用外部类的非静态变量。

4.创建静态内部类对象不需要其外部类的对象。

Outer out=**new** Outer(); //非静态内部类创建对象

Outer.Inter in=out.**new** Inter();

Outer.Inter in=**new** Outer.Inter();//静态内部类创建对象

#### 13.5 引用外部类对象

**public** **class** test{

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

}

**private** **int** count=1;

**class** Inter{

**private** **int** count=2;

**public** **void** print(count){

System.out.println(count); //参数count

System.out.println(**this**.count); //内部类成员变量

System.out.println(Outer.**this**.count); //内部类成员变量

}

}

}

**public** **class** Test **extends** A.B{

**public** Test(A a){

a.**super**(); //要调用构造方法Test（）先调用父类B的构造方法，要构造B，必须先构造A

}

}

**class** A{

**class** B{

}

}

### 第十四章

#### 14.1 异常概述

##### 什么是异常

程序运行过程中出现的不正常现象。

java中1/0.0是无穷大。而1/0是异常。

**try**: 把可能发生异常的代码包起来。

**catch**：捕获异常并处理， exception捕获所有异常，但没法分类。

**finally**：无论是否异常，都执行

**throw**：手动引发一个异常

**throws**: 定义任何被调用方法的异常。

##### 异常原因

1.用户输入错误。

2.代码本身错误。

3.环境因素（内存，磁盘）。

#### 14.2 异常的分类

Throwable->Exception

Throwable->error

error:内部错误或资源耗尽错误，很难恢复，不期望用户来处理。

exception：

ClassCastException:类型不能强转时抛出的异常。

NUllPointerException:空指针异常。

ArrayIndexOutOfBoundsException:数组越界异常。

ArithmeticException:算术异常例如除0；

#### 14.3 获取异常信息

一个异常只能被捕获一次。

#### 14.4 异常声明(异常向上传递)

指一个方法不处理它所产生的异常，而是调用层次向上传递，谁调用这个方法谁处理。

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

Test t=**new** Test();

**try**{

t.test();

}**catch**(ArithmeticException e){

System.***out***.print(e.getMessage());

}

}

**public** **void** test()**throws** ArithmeticException{

1/0;

}

#### 14.5 手动抛出异常

**throw** **new** Exception("这是我定义的异常");

#### 14.6 异常链

两个或多个异常出现在同一程序中，并且会出现嵌套抛出

#### 14.7 定义自己的异常

#### 14.8 更正异常的一个点知识点

### 第十五章

#### 15.1 多线程的基本知识

1.新建状态：线程创建之后

2.就绪状态：新建状态线程调用start（）或阻塞状态解除阻塞

3.运行状态：

4.阻塞状态：等待资源

5.终止状态 a.正常终止b.强制终止stop（）或destory（）c.异常终止

#### 15.2 继承Thread类创建线程

创建线程

**class** MyThread **extends** Thread{

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

}

}

直接手动调用run（）方法不会开辟新线程，要用.start();

#### 15.3 实现Runnable接口创建线程

实现Runnnable接口

**class** MyThread **implements** Runnable{

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

}

}

MyThread mt=**new** MyThread();

Thread t=**new** Thread(mt);

t.start();

extends Thread: java 是单继承，限制较大

implements Runnable:灵活，多实现，用到多

#### 15.4 多线程应用

##### 1.多线程并发执行

Thread t1=**new** Thread();

Thread t2=**new** Thread();

t1.start();

t2.start();

不一定t1先执行，先进入就绪状态由cpu选择，分配时间片；

##### 2.线程优先级

优先级高的线程有更大的可能先执行，并不是总先执行，低的也有机会执行。

Thread t1=**new** Thread();

Thread t2=**new** Thread();

t1.getPriority();//获取优先级，默认为5；

t1.setPriority(Thread.MAX\_PRIORITY);//设置为最高优先级10；

t2.setPriority(Thread.MIN\_PRIORITY);//设置为最低优先级1；

#### 15.5 线程调度的三个方法

1.休眠方法sleep(毫秒),sleep(毫秒，纳秒)；a.sleep();a线程休眠

2.暂停方法yield() a.yield(); a线程释放资源，大家（包括自己）抢资源,所以不会给比自己优先级低的线程机会。

3.挂起方法join() a.join(); a线程插队

#### 15.6 线程同步问题的由来

#### 15.7 java对于同步问题的解决方案

#### 15.8 线程的死锁

## 四、javaSE高级教程视频学习笔记

1.八大数据类型和封装

装箱和拆箱：即基本数据类型和封装类型的默认转换

2.finallize（）//清理垃圾函数，由虚拟机自动调用

System.gc() //建议虚拟机调用finallize()

3.流：输入输出流

字节流： 一字节为单位，主要处理字节或二进制数据的输入输出；

一般用于处理图像和声音文件

InputStream

OutputStream

字符流： 以字符为基本处理单位，主要用于处理字符或者文本类型。

一般用作处理文本读取存储以及网络的文本信息的交互.

Reader

Writer

文件管理：

File f=**new** File("C:\\User\\admin\\Desktop\\") //可以是文件或文件夹

f.getName()

f.getPath()

f.isFile()

f.exists()

f.mkdirs()

4.FileInputStream.read() 读取内容

InputStream 只能读取文件名不能读取内容。不能读取中文会乱码

5.FileOutputStream.write()

6.ByteArrayInputStream输入源是数组

7.ByteArrayoutputStream将数据写入内存而不是硬盘

8.BufferedInputStream

BufferedOutputStream

FileInputStream.read()：没有缓冲每操作一个字节就操作一次硬盘，浪费时间

BufferedInputStream.read()：构造对象时可以设置缓冲区大小。默认8192字节，一次读入内存缓冲区大小的内容大大减少 操作硬盘

9.字符流

GBK: 2字节

UTF-8:3字节

InputStreamReader.read()每次读取的字节是根据编码集决定的而不是一个字节

10.OutputStreamWriter.write()

11.FileReader

FileReader fr=**new** FileReader("C:\\ \\);

可以直接读取中文；但不能指定编码集

12.FileWriter

13.BufferReader

类似于FileInputStream和BufferInputStream

的关系

14.BufferWriter

15.序列化

class ss implement Serializable

对象持久化，保存到硬盘

网络中传输

Serializable接口

对象输入输出流

ObjectInputStream

ObjectOutputStream

### 第五章

#### 5.1 AWT简介

AWT第一代图形编程技术

Swing第二代图形编程技术

**Component**:组件，处于AWY顶层，是一个抽象类，封装了可视化组件的一般属性，所有显示在屏幕上的组件全都是靠它的子类

**Container**：容器，它是组件的一个子类。它增加了一个方法，允许其他组件加入到容器中。

**Panel**：面板它是容器的一个点子类，panel对象没有标题栏，也没有菜单栏和边框栏。

**Window**：窗口，它是一个顶级的窗口类，通常用他的子类Frame创建窗口。

**Frame**：框架，它是窗口的子类，包含标题栏菜单栏和控制按钮。

#### 5.2 Swing和AWT比较

Swing的开发是基于AWT的，AWT是Swing的基础。

Swing组件丰富，轻量型，不用针对不同平台修改。不要混用Swing和AWT会出现遮挡。

/\*\*

\*菜单实例

\*/

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** javax.swing.JFrame;

**import** javax.swing.JMenu;

**import** javax.swing.JMenuBar;

**import** javax.swing.JMenuItem;

**public** **class** Demo **extends** JFrame{

JMenuBar jmb=**new** JMenuBar(); //菜单栏

JMenu jm=**new** JMenu(); //菜单

JMenuItem jmi1=**new** JMenuItem();//菜单选项

JMenuItem jmi2=**new** JMenuItem();

**public** **void** Demo(){

**this**.setLayout(**null**);

jm.add(jmi1); //将菜单选项加入到菜单

jm.add(jmi2);

jmb.add(jm); //将菜单加入到菜单栏

**this**.setJMenuBar(jmb); //将菜单栏绑定到框架

**this**.setBounds(300,200,400,350);//设置窗体大小

**this**.addWindowListener(**new** WindowAdapter(){

**public** **void** windowClosing(WindowEvent e){

System.*exit*(0);

}

});

**this**.setVisible(**true**);

}

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

Demo de= **new** Demo();

}

}

#### 5.3创建窗体JFrame

SimpleFrame.java

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** javax.swing.JFrame;

**public** **class** SimpleFrame **extends** JFrame{

**public** SimpleFrame(){

**this**.setSize(500,400);

**this**.setVisible(**true**);

**this**.addWindowListener(**new** WindowAdapter(){

**public** **void** windowClosing(WindowEvent e){

System.exit(0);

}

});

}

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

**new** SimpleFrame();

}

}

CenterFrame.java

**import** java.awt.Dimension;

**import** java.awt.Toolkit;

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** javax.swing.JFrame;

**public** **class** CenterFrame **extends** JFrame{

**public** CenterFrame(){

**this**.setTitle("窗体居中显示");

Toolkit tk=Toolkit.*getDefaultToolkit*();

Dimension screenSize=tk.getScreenSize();

**int** screenWidth=screenSize.width;

**int** screenHeight=screenSize.height;

**this**.setSize(screenWidth/2,screenHeight/2);

**this**.setLocation(screenWidth/4,screenHeight/4);

**this**.repaint();

**this**.addWindowListener(**new** WindowAdapter(){

**public** **void** windowClosing(WindowEvent e){

System.*exit*(0);

}

});

**this**.setVisible(**true**);

}

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

**new** CenterFrame();

}

}

#### 5.4 Jpanel

1.面板相当于图书馆的书架，面板里放组件也可以嵌套面板

2.可以在面板上绘制图形

可以直接使用系统面板也可以自定义面板

自定义面板必须重写paintComponent(Graphics g)方法

**import** java.awt.Graphics;

**import** javax.swing.JPanel;

**public** **class** MyPanel **extends** JPanel{

**public** **void** paintComponent(Graphics g){

g.drawString("这事可面板上的内容",100,100);

}

}

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** javax.swing.JFrame;

**public** **class** MyFrame **extends** JFrame{

**public** MyFrame(){

MyPanel p=**new** MyPanel();

**this**.add(p);

**this**.setSize(500,300);

**this**.setVisible(**true**);

**this**.addWindowListener(**new** MyWindowListener());

}

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

**new** MyFrame();

}

**class** MyWindowListener **extends** WindowAdapter{

**public** **void** windowClosing(WindowEvent e){

System.*exit*(0);

}

}

}

#### 5.5事件处理模型

事件源：触发事件的组件。事件的信息被封装在事件对象中。

事件监听器：实现了监听接口的类的对象。

JFrame监听器：WindowListener

JButton监听器：ActionListener

#### 5.6 窗口事件

注册不同的监听器会用到不同的方法

如果自定义监听器实现多个接口要注册多次。

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** java.awt.event.WindowFocusListener;

**import** java.awt.event.WindowListener;

**import** javax.swing.JFrame;

**public** **class** MyFrame **extends** JFrame {

**public** MyFrame(){

**this**.setTitle("窗体事件的监听器");

**this**.setSize(500,400);

**this**.addWindowListener(**new** MyWindowListener());

**this**.addWindowFocusListener(**new** MyWindowListener());

**this**.setVisible(**true**);

}

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

**new** MyFrame();

}

**class** MyWindowListener **implements** WindowListener,WindowFocusListener{

/\*\*

\* WindowListener接口的方法

\*/

@Override

**public** **void** windowClosing(WindowEvent e){

System.***out***.println("窗口正在关闭");

System.*exit*(0);

}

@Override

**public** **void** windowClosed(WindowEvent e){

System.***out***.println("窗口已关闭");

}

@Override

**public** **void** windowOpened(WindowEvent e) {

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

System.***out***.println("窗口打开");

}

//监听最小化

@Override

**public** **void** windowIconified(WindowEvent e) {

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

}

//监听最大化

@Override

**public** **void** windowDeiconified(WindowEvent e) {

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

}

@Override

**public** **void** windowActivated(WindowEvent e) {

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

}

@Override

**public** **void** windowDeactivated(WindowEvent e) {

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

}

/\*\*

\* 接口WindowFocusListener的方法

\*

\*/

//窗口获得焦点

@Override

**public** **void** windowGainedFocus(WindowEvent e) {

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

System.***out***.println("窗口获得焦点");

}

//窗口失去焦点

@Override

**public** **void** windowLostFocus(WindowEvent e) {

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

System.***out***.println("窗口失去焦点");

}

}

}

#### 5.7事件适配器

监听器：实现接口implements

事件适配器：继承类extends

实现了一个 或多个监听接口的类。它为接口的事件处理

方法都提供了空实现。在使用时可以继承适配器泪并重写需要的时间处理方法。

**windowAdapter**: 实现了WindowFocusListener;WindowListener、WindowStateListener;EventListener;

**keyAdapter** :实现了keyListener;EventListener

**MouseAdapter**: 实现了MouseListener、MouseMotionListener

#### 5.8 JLabel

JLable:可以包含文字或图片

#### 5.9 JButton

一个组件注册多个监听， 先注册的监听器后执行

**package** com.java;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JPanel;

**public** **class** TestJButton{

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

**new** MyFrame();

}

}

**class** MyFrame **extends** JFrame{

JPanel jp=**new** JPanel();

JButton jb=**new** JButton("按钮");

JLabel jl=**new** JLabel("被按了0次");

**public** MyFrame(){

jp.add(jb);

jp.add(jl);

**this**.add(jp);

**this**.setTitle("JButton的使用");

**this**.setSize(500,300);

**this**.addWindowListener(**new** WindowAdapter(){

**public** **void** windowClosing(WindowEvent e){

System.exit(0);

}

});

jb.addActionListener(**new** ActionListener(){

**int** count=0;

@Override

**public** **void** actionPerformed(ActionEvent e) {

jl.setText("这是第"+(count++)+"次按下按钮");

}

});

**this**.setVisible(**true**);

}

}

#### 5.10 JTextField:文本输入框

实现ActionListener接口重写actionPerformed（ActionEvent e)

jtf.getText();

#### 5.11 JCheckBox

复选框

实现ItemListener

@Override

**public** **void** itemStateChanged(itemEvent e){

JCheckBox cb=(JCheckBox)e.getItem();

**if**(e.getStateChange()==ItemEvent.SELECTED){

}

}

#### 5.12 RadioButton

单选按钮

实现ActionListener

ButtonGroup bg=**new** ButtonGroup();//通过这样实现只能选择一个选项

bg.add(rb1);

bg.add(rb2);

bg.add(rb3);

#### 5.13 JComboBox

下拉列表

实现IteListener

JComboBox jcb=**new** JComboBox();

jcb.add("C++");

jcb.add("java");

jcb.add("php");

#### 5.14 JTabbedPanel

组件选项卡

不需要监听器

JTabbedPanel jtp=**new** JTabbedPanel();

JPanel jp1=**new** JPanel();

JPanel jp2=**new** JPanel();

JPanel jp3=**new** JPanel();

jtp.add(Jp1);

jtp.add(Jp2);

jtp.add(Jp3);

#### 5.15 FlowLayout流式布局

JPanel jp=**new** JPanel();

FlowLayout fl=**new** FlowLayout();//构造函数为空默认居中

// FlowLayout fl=new FlowLayout(FlowLayout.LEFT);

//FlowLayout fl=new FlowLayout(FlowLayout.RIGHT);

jp.setLayout(f1);

将组件一排排挨着放置。自动调用组件的getPreferredSize（）使组件以最佳尺寸显示，当容器大小重新设置后，布局也会随之改变。

#### 5.16 BorderLayout 边界布局

五区域：north,south;east; west;center;

每个区域只能添加一个组件或面板

如果区域没有放置东西；东西南北不会预留位置，而中间区域会预留空位置。

东西高变，南北宽变，中间宽高变

BorderLayout bl=**new** BorderLayout(10,5);//水平，竖直间距

**this**.setLayout(bl);

**this**.add(BorderLayout.NORTH,**new** JButton("north");

**this**.add(BorderLayout.SOUTH,**new** JButton("south");

**this**.add(BorderLayout.WEST,**new** JButton("west");

**this**.add(BorderLayout.EAST,**new** JButton("east");

**this**.add(BorderLayout.CENTER,**new** JButton("center");

#### 5.17 GridLayout 网格布局

从左到右，从上到下，一个网格只能放置一个组件或一个面板

GridLayout gl=**new** GridLayout(3,3,5,5);//行，列，宽，高

**this**.add(**new** JButton("1");

**this**.add(**new** JButton("2");

**this**.add(**new** JButton("3");

**this**.add(**new** JButton("4");

**this**.add(**new** JButton("5");

**this**.add(**new** JButton("6");

**this**.add(**new** JButton("7");

**this**.add(**new** JButton("8");

**this**.add(**new** JButton("9");

#### 5.18 CardLayout 卡片布局

多个组件共享同一空间；如蜘蛛纸牌的扑克牌

**public** **void** show(COntainer parent,String name)//根据名字显示

**public** **void** next(COntainer parent)//下一张

**public** **void** previous(COntainer parent)//前一张

**public** **void** first(COntainer parent)//第一张

**public** **void** last(COntainer parent)//最后一张

**package** com.java;

**import** java.awt.BorderLayout;

**import** java.awt.CardLayout;

**import** java.awt.FlowLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** javax.swing.ImageIcon;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JPanel;

**public** **class** TestCardLayout **extends** JFrame **implements** ActionListener{

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

**new** TestCardLayout();

}

JPanel jp1,jp2;

JButton jb1,jb2,jb3;

JLabel jl1,jl2,jl3;

**public** TestCardLayout(){

jp1=**new** JPanel();

jb1=**new** JButton("按钮1");

jb2=**new** JButton("按钮2");

jb3=**new** JButton("按钮3");

jb1.addActionListener(**this**);

jb2.addActionListener(**this**);

jb3.addActionListener(**this**);

jp1.setLayout(**new** FlowLayout());

jp1.add(jb1);

jp1.add(jb2);

jp1.add(jb3);

jp2=**new** JPanel();

jp2.setLayout(**new** CardLayout());

jl1=**new** JLabel(**new** ImageIcon("I:\\图片\\1001"));

jl2=**new** JLabel(**new** ImageIcon("I:\\图片\\1002"));

jl3=**new** JLabel(**new** ImageIcon("I:\\图片\\1003"));

jp2.add("label1",jl1);

jp2.add("label2",jl2);

jp2.add("label3",jl3);

**this**.setLayout(**new** BorderLayout());

**this**.add(BorderLayout.***NORTH***,jp1);

**this**.add(BorderLayout.***CENTER***,jp2);

**this**.setTitle("卡片布局");

**this**.setSize(500,300);

**this**.setVisible(**true**);

**this**.addWindowListener(**new** WindowAdapter(){

**public** **void** windowClosing(WindowEvent e) {

System.*exit*(0);

}

});

}

@Override

**public** **void** actionPerformed(ActionEvent e) {

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

**if**(e.getSource()==jb1){

((CardLayout)jp2.getLayout()).show(jp2,"label1");

}**else** **if**(e.getSource()==jb2){

((CardLayout)jp2.getLayout()).show(jp2,"label2");

}**else** **if**(e.getSource()==jb3){

((CardLayout)jp2.getLayout()).show(jp2,"label3");

}

}

}

#### 5.19实现自己的计算器

**package** com.java;

**import** java.awt.BorderLayout;

**import** java.awt.GridLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JPanel;

/\*\*

\* **@author** liuzhongming1

\*/

**public** **class** Caculator {

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

**new** CaculatorFrame();

}

}

**class** CaculatorFrame **extends** JFrame{

**public** CaculatorFrame(){

**this**.setTitle("计算器");

CaculatorPanel cp=**new** CaculatorPanel ();

**this**.add(cp);

**this**.pack();//自动调整窗体大小

**this**.addWindowListener(**new** WindowAdapter(){

**public** **void** Closing(WindowEvent e){

System.exit(0);

}

});

**this**.setVisible(**true**);

}

}

**class** CaculatorPanel **extends** JPanel{

JLabel display=**null**;

JPanel panel=**null**;

**double** result;

String command;

**boolean** start;

**public** CaculatorPanel(){

**this**.setLayout(**new** BorderLayout());

result=0;

command="=";

start=**true**;

display=**new** JLabel();

display.setText("0");

**this**.add(BorderLayout.NORTH,display);

panel=**new** JPanel();

panel.setLayout(**new** GridLayout(4,4));

CaculatorAction1 c1=**new** CaculatorAction1();

CaculatorAction2 c2=**new** CaculatorAction2();

addButton("7",c1);

addButton("8",c1);

addButton("9",c1);

addButton("+",c2);

addButton("4",c1);

addButton("5",c1);

addButton("6",c1);

addButton("-",c2);

addButton("1",c1);

addButton("2",c1);

addButton("3",c1);

addButton("\*",c2);

addButton("0",c1);

addButton(".",c1);

addButton("=",c2);

addButton("/",c2);

**this**.add(BorderLayout.CENTER,panel);

}

**private** **void** addButton(String s,ActionListener al){

JButton button=**new** JButton(s);

button.addActionListener(al);

panel.add(button);

}

//数字监听器

**class** CaculatorAction1 **implements** ActionListener{

@Override

**public** **void** actionPerformed(ActionEvent e) {

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

String input=e.getActionCommand();

**if**(start){

display.setText("");

start=**false**;

}

display.setText(display.getText()+input);

}

}

//命令监听器

**class** CaculatorAction2 **implements** ActionListener{

@Override

**public** **void** actionPerformed(ActionEvent e) {

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

String c=e.getActionCommand();

**if**(start){

command=c;

}**else**{

**if**(command.equals("+")){

result+=Double.parseDouble(display.getText());

}**else** **if**(command.equals("-")){

result-=Double.parseDouble(display.getText());

}**else** **if**(command.equals("\*")){

result\*=Double.parseDouble(display.getText());

}**else** **if**(command.equals("/")){

result/=Double.parseDouble(display.getText());

}**else** **if**(command.equals("=")){

result=Double.parseDouble(display.getText());

}

display.setText(""+result);

command=c;

start=**true**;

}

}

}

}

### 第六章

#### 6.1 网络编程InetAddress类

InetAddress：表示网络协议地址的类

**package** com.java;

**import** java.net.InetAddress;

**import** java.net.UnknownHostException;

**public** **class** TestInetAddress {

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

**try** {

InetAddress baidu=InetAddress.getByName("www.baidu.com");

InetAddress[] baidus=InetAddress.getAllByName("www.baidu.com");

System.out.println(baidu);

**for**(InetAddress ia:baidus){

System.out.println(ia);

}

InetAddress my=InetAddress.getLocalHost();//获取本机地址

System.out.println("本机地址："+my);

**byte**[] bs=**new** **byte**[]{(**byte**)220,(**byte**)181,111,(**byte**)188};

InetAddress.getByAddress(bs);

} **catch** (UnknownHostException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

#### 6.2 网络编程URl类

URl:同一资源定位符

URL url=**new** URL(http://bbs.java1995.com/forum.php?"+"mod=viewthread&tid=3615");

System.out.println("主机："+url.getHost());

System.out.println("端口："+url.getPort());

System.out.println("路径："+url.getPath());

System.out.println("协议："+url.getProtocol());

System.out.println("文件名："+url.getFile());

System.out.println("url内容："+url.getContent());

InputStreamReader isr=**new** InputStreamReader(url.openStream());

BufferReader br=**new** BufferReader(isr);

String line=**null**;

**while**(line=br.readLine())!=**null**){

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

}

#### 6.5-6.6 Socket实现聊天室

服务端

**package** com.java;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.OutputStreamWriter;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.util.ArrayList;

**import** java.util.List;

/\*\*

\*

\* 简易聊天室服务端

\* **@author** liuzhongming1

\*

\*/

**public** **class** Server {

**public** **static** List<Socket>sockets=**new** ArrayList<Socket>();

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

**try** {

System.out.println("服务器已启动");

ServerSocket server=**new** ServerSocket(1995);

**int** count=0;

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

System.out.println("等待客户端链接");

Socket s=server.accept();//阻塞方法；等待客户端连接

sockets.add(s);

System.out.println("客户端已连接，目前在线人数"+sockets.size());

//服务端为每一个分配线程

**new** ServerThread(s).start();

}

}

**catch** (IOException e) {

e.printStackTrace();

}

}

}

**class** ServerThread **extends** Thread{

**private** Socket s;

**public** ServerThread(Socket s){

**this**.s=s;

}

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

BufferedReader br;

**try**{

br = **new** BufferedReader(**new** InputStreamReader(s.getInputStream()));

//获得登陆的用户姓名，然后给所有用户发送系统消息XXX登录

String username=br.readLine();

**for**(Socket socket:Server.sockets){

OutputStreamWriter osw=**new** OutputStreamWriter(socket.getOutputStream());

osw.write("【系统消息】：欢迎"+username+"登录\n");

osw.flush();

}

//接受聊天信息，转发给所有客户端

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

String content=br.readLine();

**for**(Socket socket:Server.sockets){

OutputStreamWriter osw=**new** OutputStreamWriter(socket.getOutputStream());

osw.write(username+":"+content);

osw.flush();

}

System.out.println("日志记录："+username+":"+content);

}

}**catch** (IOException e) {

e.printStackTrace();

}

}

}

客户端

**package** com.java;

**import** java.awt.BorderLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.event.WindowAdapter;

**import** java.awt.event.WindowEvent;

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.OutputStreamWriter;

**import** java.net.Socket;

**import** javax.swing.JFrame;

**import** javax.swing.JPanel;

**import** javax.swing.JScrollPane;

**import** javax.swing.JTextArea;

**import** javax.swing.JTextField;

/\*\*

\* **@author** liuzhongming1

\*与服务器交互

\*

\*/

**public** **class** Client **extends** JFrame{

**private** Socket s;

**private** String username;

JPanel jp1,jp2; //两个面板

JTextArea jta; //显示聊天内容

JTextField jtf; //显示输入内容

JScrollPane jsp; //滚动条

**public** Client(Socket s,String username){

**this**.s=s;

**this**.username=username;

jp1=**new** JPanel();

jta=**new** JTextArea(20,30); //行，列

jsp=**new** JScrollPane(jta); //添加滚动条

jp1.add(jsp);

jp2=**new** JPanel();

jtf=**new** JTextField(20);

jp2.add(jtf);

**this**.setLayout(**new** BorderLayout());

**this**.add(BorderLayout.***CENTER***,jp1);

**this**.add(BorderLayout.***SOUTH***,jp2);

**this**.pack();

**this**.setTitle("当前用户："+username);

**this**.setVisible(**true**);

**this**.addWindowListener(**new** WindowAdapter(){

**public** **void** windowClosing(WindowEvent e){

System.*exit*(0);

}

});

jtf.addActionListener(**new** MyListener());

**new** GetThread().start();

}

**class** MyListener **implements** ActionListener{

@Override

**public** **void** actionPerformed(ActionEvent e) {

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

//jta.setText(jta.getText()+"\n"+jtf.getText());

//jtf.setText("");

//向服务器发送消息

String content=jtf.getText();

**try** {

OutputStreamWriter osw=**new** OutputStreamWriter(s.getOutputStream());

osw.write(content+"\n");

osw.flush();

jtf.setText("");

} **catch** (IOException e1) {

e1.printStackTrace();

}

}

}

//接受服务器发送的消息

**class** GetThread **extends** Thread{

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

**try** {

BufferedReader br=**new** BufferedReader(**new** InputStreamReader(s.getInputStream()));

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

String content=br.readLine();

jta.setText(jta.getText()+"\n"+content);

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

登录\n

**package** com.java;

**import** java.awt.FlowLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.io.IOException;

**import** java.io.OutputStreamWriter;

**import** java.net.Socket;

**import** java.net.UnknownHostException;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JTextField;

/\*\*

\*

\* **@author** liuzhongming1

\*

\*登录界面

\*

\*/

**public** **class** Login **extends** JFrame **implements** ActionListener{

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

**new** Login();

}

JLabel label=**new** JLabel("请输入用户名：");

JTextField jtf=**new** JTextField(20); //20个字;单行文本框

**public** Login(){

**this**.setLayout(**new** FlowLayout());

**this**.add(label);

**this**.add(jtf);

**this**.pack(); //自动调整大小

**this**.setVisible(**true**);

jtf.addActionListener(**this**);

}

//文本框监听器

@Override

**public** **void** actionPerformed(ActionEvent e) {

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

Socket s=**null**;

**try** {

s = **new** Socket("127.0.0.1",1995);

} **catch** (UnknownHostException e2) {

e2.printStackTrace();

} **catch** (IOException e2) {

e2.printStackTrace();

}

**try** {

OutputStreamWriter osw= **new** OutputStreamWriter(s.getOutputStream());

osw.write(jtf.getText()+"\n");

osw.flush();

} **catch** (IOException e1) {

e1.printStackTrace();

}

**new** Client(s,jtf.getText()); //打开客户端界面

**this**.dispose(); //关闭登录界面

}

}

## 五、集合

### 1.集合与数组

**数组：**（可以存储基本数据类型）是用来存现对象的一种容器，但是数组的长度固定，不适合在对象数量未知的情况下使用。

**集合**（只能存储对象，对象类型可以不一样）的长度可变，可在多数情况下使用。

### 2.集合中接口和类的关系

**Collection**接口是集合类的根接口，Java中没有提供这个接口的直接的实现类。但是却让其被继承产生了两个接口，就是**Set**和**List**。Set中不能包含重复的元素。List是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式。

**Map**是Java.util包中的另一个接口，它和Collection接口没有关系，是相互独立的，但是都属于集合类的一部分。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value。

**Iterator**所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：  
1.**hasNext()**是否还有下一个元素。  
2.**next()**返回下一个元素。  
3.**remove()**删除当前元素。



| **接口** | **子接口** | **是否有序** | **是否允许元素重复** |
| --- | --- | --- | --- |
| **Collection** |  | 否 |  |
| **List** | ArrayList | 否 | 是 |
|  | LinkedList | 否 | 是 |
|  | Vector | 否 | 是 |
| **Set** | AbstractSet | 否 | 否 |
|  | HashSet | 否 | 否 |
|  | TreeSet | 是（用二叉排序树） | 否 |
| **Map** | AbstractMap | 否 | 使用key-value来映射和存储数据，key必须唯一，value可以重复 |
|  | HashMap |  | 否 |
|  | TreeMap | 是（用二叉排序树） | 使用key-value来映射和存储数据，key必须唯一，value可以重复 |

### 3.list（有序、可重复）

List里存放的对象是有序的，同时也是可以重复的，List关注的是索引，拥有一系列和索引相关的方法，查询速度快。因为往list集合里插入或删除数据时，会伴随着后面数据的移动，所有插入删除数据速度慢。

#### ArrayList

ArrayList是基于数组的，在初始化ArrayList时，会构建空数组（Object[] elementData={}）。ArrayList是一个无序的，它是按照添加的先后顺序排列，当然，他也**提供了sort方法**，如果需要对ArrayList进行排序，只需要调用这个方法，提供Comparator比较器即可

**public** **class** Student **implements** Comparable {

**private** String name;

**private** **int** rollno;

**private** **int** age;

/\* 有参构造 \*/

@Override

**public** **int** compareTo(Student comparestu) {

**int** compareAge=((Student)comparestu).getAge();

/\* 正序排列 \*/

**return** **this**.age-compareAge;

}

}

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

ArrayList<Student> arraylist = **new** ArrayList<Student>();

arraylist.add(**new** Student(223, "Chaitanya", 26));

arraylist.add(**new** Student(245, "Rahul", 24));

arraylist.add(**new** Student(209, "Ajeet", 32));

Collections.sort(arraylist);

}

##### add操作：

　　1）如果是第一次添加元素，数组的长度被扩容到默认的capacity，也就是10.

　　2) 当发觉同时添加一个或者是多个元素，数组长度不够时，就扩容，这里有两种情况：

　　只添加一个元素，例如：原来数组的capacity为10，size已经为10，不能再添加了。需要扩容，新的capacity=old capacity+old capacity>>1=10+10/2=15.即新的容量为15。

　　当同时添加多个元素时，原来数组的capacity为10，size为10，当同时添加6个元素时。它需要的min capacity为16，而按照capacity=old capacity+old capacity>>1=10+10/2=15。new capacity小于min capacity，则取min capacity。

　　对于添加，如果不指定下标，就直接添加到数组后面，不涉及元素的移动，如果要添加到某个特定的位置，那需要将这个位置开始的元素往后挪一个位置，然后再对这个位置设置。

##### Remove操作：

Remove提供两种，按照下标和value。

　1）**remove(int index)**：首先需要检查Index是否在合理的范围内。其次再调用System.arraycopy将index之后的元素向前移动。

　2）**remove(Object o)**：首先遍历数组，获取第一个相同的元素，获取该元素的下标。其次再调用System.arraycopy将index之后的元素向前移动。

##### Get操作：

这个比较简单，直接对数组进行操作即可。

#### LinkedList

　　LinkedList是基于链表的，它是一个双向链表，每个节点维护了一个prev和next指针。同时对于这个链表，维护了first和last指针，first指向第一个元素，last指向最后一个元素。LinkedList是一个无序的链表，按照插入的先后顺序排序，**不提供sort方法对内部元素排序**。

##### Add元素：

　　LinkedList提供了几个添加元素的方法：addFirst、addLast、addAll、add等，时间复杂度为O(1)。

##### Remove元素：

LinkedList提供了几个移除元素的方法：removeFirst、removeLast、removeFirstOccurrence、remove等，时间复杂度为O(1)。

##### Get元素：

　　根据给定的下标index，判断它first节点、last直接距离，如果index<size（数组元素个数)/2,就从first开始。如果大于，就从last开始。这个和我们平常思维不太一样，也许按照我们的习惯，从first开始。这也算是一点小心的优化吧。

#### 遍历

在类集中提供了以下四种的常见输出方式：

1）Iterator：迭代输出，是使用最多的输出方式。

2）ListIterator：是Iterator的子接口，专门用于输出List中的内容。

3）foreach输出：JDK1.5之后提供的新功能，可以输出数组或集合。

4）for循环

代码示例如下：

**iterator的形式：**  
Iterator it = arr.iterator();

**while**(it.hasNext()){ object o =it.next(); ...}

**ListIterator的形式：**

ListIterator<String> iter = arr.listIterator();

**while**(iter.hasNext()) {

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

}

**foreach的形式：** **for**（**int**　i：arr）{...}

**for的形式：for**（**int** i=0;i<arr.size();i++）{...}

**ListIterator和Iterator**

ListIterator有add()方法，可以向List中添加对象，而Iterator不能。ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator就不可以。

ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。

ListIterator可以实现对象的修改，set()方法可以实现。Iierator仅能遍历，不能修改。

### 4.Set（无序、不能重复）

Set里存放的对象是**无序**，**不能重复**的，集合中的对象不按特定的方式排序，只是简单地把对象加入集合中。

##### HashSet

　　HashSet是基于HashMap来实现的，操作很简单，更像是对HashMap做了一次“封装”，而且只使用了HashMap的key来实现各种特性，而HashMap的value始终都是PRESENT。

　　HashSet不允许重复（HashMap的key不允许重复，如果出现重复就覆盖），允许null值，非线程安全。

###### 构造方法

**HashSet()**  
　　构造一个新的空 set，其底层 HashMap 实例的默认初始容量是 16，加载因子是 0.75，扩容为旧容量乘2。  
**HashSet(Collection<? extends E> c)**  
　　构造一个包含指定 collection 中的元素的新 set。  
**HashSet(int initialCapacity)**  
　　构造一个新的空 set，其底层 HashMap 实例具有指定的初始容量和默认的加载因子（0.75）。  
**HashSet(int initialCapacity, float loadFactor)**  
　　构造一个新的空 set，其底层 HashMap 实例具有指定的初始容量和指定的加载因子。

###### 方法

**boolean add(E e)**  
　如果此 set 中尚未包含指定元素，则添加指定元素。  
**void clear()**  
　从此 set 中移除所有元素。  
\*\* Object clone()  **返回此 HashSet 实例的浅表副本：并没有复制这些元素本身。**boolean contains(Object o)  **如果此 set 包含指定元素，则返回 true。**boolean isEmpty()\*\*  
　如果此 set 不包含任何元素，则返回 true。  
\*\* Iterator iterator()  **返回对此 set 中元素进行迭代的迭代器。**boolean remove(Object o)  **如果指定元素存在于此 set 中，则将其移除。**int size()\*\*  
　　返回此 set 中的元素的数量（set 的容量）。

##### TreeSet

　　基于 TreeMap 的 NavigableSet 实现。使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的 Comparator进行排序，具体取决于使用的构造方法。**构造方法和方法比较类似就不说了**

##### 对 set 的遍历

1.迭代遍历：

Set<String> set = **new** HashSet<String>(); Iterator<String> it = set.iterator();

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

String str = it.next();

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

}

2.for(foreach)循环遍历：

**for** (String str : set) {

System.out.println(str);

}

### 5.Map（键值对、键唯一、值不唯一）

　　Map集合中存储的是键值对，键不能重复，值可以重复。根据键得到值，对map集合遍历时先得到键的set集合，对set集合进行遍历，得到相应的值。

##### HashMap

　　数组方式存储key/value，**线程非安全**，**允许null作为key和value**，key不可以重复，value允许重复，不保证元素迭代顺序是按照插入时的顺序，key的hash值是先计算key的hashcode值，然后再进行计算，每次容量扩容会重新计算所以key的hash值，会消耗资源，要求key必须重写equals和hashcode方法

默认初始容量16，加载因子0.75，扩容为旧容量乘2，查找元素快，如果key一样则比较value，如果value不一样，则按照链表结构存储value，就是一个key后面有多个value；

**加载因子**是表示Hsah表中元素的填满的程度。若加载因子越大,填满的元素越多,好处是,空间利用率高了,但:冲突的机会加大了.反之,加载因子越小,填满的元素越少,好处是:冲突的机会减小了,但:空间浪费多了。冲突的机会越大,则查找的成本越高.反之,查找的成本越小.因而,查找时间就越小.

###### 添加：

**V put(K key, V value)** （可以相同的key值，但是添加的value值会覆盖前面的，返回值是前一个，如果没有就返回null）

**putAll(Map<? extends K,? extends V> m)** 从指定映射中将所有映射关系复制到此映射中（可选操作）。

###### 删除

**remove()** 删除关联对象，指定key对象

**clear()** 清空集合对象

###### 获取

**value get(key)** 可以用于判断键是否存在的情况。当指定的键不存在的时候，返回的是null。

###### 判断

**boolean isEmpty()** 长度为0返回true否则false

**boolean containsKey(Object key)** 判断集合中是否包含指定的key

**boolean containsValue(Object value)** 判断集合中是否包含指定的value

###### 长度

**Int size（）**map的主要的方法就这几个

##### Hashtable

　　Hashtable与HashMap类似，是HashMap的线程安全版，它支持线程的同步，即任一时刻只有一个线程能写Hashtable，因此也导致了Hashtale在写入时会比较慢，它继承自Dictionary类，不同的是它不允许记录的键或者值为null，同时效率较低。

##### LinkedHashMap

LinkedHashMap保存了记录的插入顺序，在用Iteraor遍历LinkedHashMap时，先得到的记录肯定是先插入的，在遍历的时候会比HashMap慢，有HashMap的全部特性。

##### TreeMap

　　基于**红黑二叉树**的NavigableMap的实现，线程非安全，不允许null，key不可以重复，value允许重复，存入TreeMap的元素应当实现Comparable接口或者实现Comparator接口，会按照排序后的顺序迭代元素，两个相比较的key不得抛出classCastException。主要用于存入元素的时候对元素进行自动排序，迭代输出的时候就按排序顺序输出。

##### 遍历

**第一种：KeySet()**  
　　将Map中所有的键存入到set集合中。因为set具备迭代器。所有可以迭代方式取出所有的键，再根据get方法。获取每一个键对应的值。 keySet():迭代后只能通过get()取key 。取到的结果会乱序，是因为取得数据行主键的时候，使用了HashMap.keySet()方法，而这个方法返回的Set结果，里面的数据是乱序排放的。

Map map = **new** HashMap();

map.put("key1","lisi1");

map.put("key2","lisi2");

map.put("key3","lisi3");

map.put("key4","lisi4");

//先获取map集合的所有键的set集合，keyset（）

Iterator it = map.keySet().iterator();

//获取迭代器

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

Object key = it.next();

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

}

**第二种： values() 获取所有的值.**  
Collection values()**不能获取到key对象**

Collection<String> vs = map.values();

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

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

String value = it.next();

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

}

**第三种：entrySet（）**  
Set<Map.Entry<K,V>> entrySet() //返回此映射中包含的映射关系的 Set 视图。（一个关系就是一个键-值对），就是把(key-value)作为一个整体一对一对地存放到Set集合当中的。Map.Entry表示映射关系。entrySet()：迭代后可以e.getKey()，e.getValue()两种方法来取key和value。返回的是Entry接口。

// 返回的Map.Entry对象的Set集合 Map.Entry包含了key和value对象

Set<Map.Entry<Integer, String>> es = map.entrySet();

Iterator<Map.Entry<Integer, String>> it = es.iterator();

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

// 返回的是封装了key和value对象的Map.Entry对象

Map.Entry<Integer, String> en = it.next();

// 获取Map.Entry对象中封装的key和value对象

Integer key = en.getKey();

String value = en.getValue();

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

}

　　推荐使用第三种方式，即**entrySet()方法，效率较高**。  
　　对于keySet其实是遍历了2次，一次是转为iterator，一次就是从HashMap中取出key所对于的value。而entryset只是遍历了第一次，它把key和value都放到了entry中，所以快了。两种遍历的遍历时间相差还是很明显的。

### 总结：

##### 1、Vector和ArrayList

　　1、vector是线程同步的，所以它也是线程安全的，而arraylist是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用arraylist效率比较高。

　　2、如果集合中的元素的数目大于目前集合数组的长度时，vector增长率为目前数组长度的100%，而arraylist增长率为目前数组长度的50%。如果在集合中使用数据量比较大的数据，用vector有一定的优势。

　　3、如果查找一个指定位置的数据，vector和arraylist使用的时间是相同的，如果频繁的访问数据，这个时候使用vector和arraylist都可以。而如果移动一个指定位置会导致后面的元素都发生移动，这个时候就应该考虑到使用linklist,因为它移动一个指定位置的数据时其它元素不移动。

**ArrayList 和Vector是采用数组方式存储数据**，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接序号索引元素，但是插入数据要涉及到数组元素移动等内存操作，所以索引数据快，插入数据慢，Vector由于使用了synchronized方法（线程安全）所以性能上比ArrayList要差，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快。

##### 2、arraylist和linkedlist

　　1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

　　2.对于随机访问get和set，ArrayList优于LinkedList，因为LinkedList要移动指针。

　　3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。 这一点要看实际情况的。若只对单条数据插入或删除，ArrayList的速度反而优于LinkedList。但若是批量随机的插入删除数据，LinkedList的速度大大优于ArrayList. 因为ArrayList每插入一条数据，要移动插入点及之后的所有数据。

##### 3、HashMap与TreeMap

　　1、 HashMap通过hashcode对其内容进行快速查找，而TreeMap中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用TreeMap（HashMap中元素的排列顺序是不固定的）。

　　2、在Map 中插入、删除和定位元素，HashMap是最好的选择。但如果您要按自然顺序或自定义顺序遍历键，那么TreeMap会更好。使用HashMap要求添加的键类明确定义了hashCode()和 equals()的实现。

　两个map中的元素一样，但顺序不一样，导致hashCode()不一样。

同样做测试：  
　在HashMap中，同样的值的map,顺序不同，equals时，false;而在treeMap中，同样的值的map,顺序不同,equals时，true，说明，treeMap在equals()时是整理了顺序了的。

##### 4、HashTable与HashMap

1、同步性:Hashtable是线程安全的，也就是说是同步的，而HashMap是线程序不安全的，不是同步的。

2、HashMap允许存在一个为null的key，多个为null的value 。

3、hashtable的key和value都不允许为null。

## 六、HttpClient

**Http协议**的重要性相信不用我多说了，HttpClient相比传统JDK自带的URLConnection，增加了易用性和灵活性（具体区别，日后我们再讨论），它不仅是客户端发送Http请求变得容易，而且也方便了开发人员测试接口（基于Http协议的），即提高了开发的效率，也方便提高代码的健壮性。因此熟练掌握HttpClient是很重要的必修内容，掌握HttpClient后，相信对于Http协议的了解会更加深入。

HttpClient是Apache Jakarta Common下的子项目，用来提供高效的、最新的、功能丰富的支持HTTP协议的客户端编程工具包，并且它支持HTTP协议最新的版本和建议.

1. 基于标准、纯净的Java语言。实现了Http1.0和Http1.1

2. 以可扩展的面向对象的结构实现了Http全部的方法（GET, POST, PUT, DELETE, HEAD, OPTIONS, and TRACE）。

3. 支持HTTPS协议。

4. 通过Http代理建立透明的连接。

5. 利用CONNECT方法通过Http代理建立隧道的https连接。

6. Basic, Digest, NTLMv1, NTLMv2, NTLM2 Session, SNPNEGO/Kerberos认证方案。

7. 插件式的自定义认证方案。

8. 便携可靠的套接字工厂使它更容易的使用第三方解决方案。

9. 连接管理器支持多线程应用。支持设置最大连接数，同时支持设置每个主机的最大连接数，发现并关闭过期的连接。

10. 自动处理Set-Cookie中的Cookie。

11. 插件式的自定义Cookie策略。

12. Request的输出流可以避免流中内容直接缓冲到socket服务器。

13. Response的输入流可以有效的从socket服务器直接读取相应内容。

14. 在http1.0和http1.1中利用KeepAlive保持持久连接。

15. 直接获取服务器发送的response code和 headers。

16. 设置连接超时的能力。

17. 实验性的支持http1.1 response caching。

18. 源代码基于Apache License 可免费获取。

使用HttpClient发送请求、接收响应很简单，一般需要如下几步即可。

1. 创建HttpClient对象。

2. 创建请求方法的实例，并指定请求URL。如果需要发送GET请求，创建HttpGet对象；如果需要发送POST请求，创建HttpPost对象。

3. 如果需要发送请求参数，可调用HttpGet、HttpPost共同的setParams(HetpParams params)方法来添加请求参数；对于HttpPost对象而言，也可调用setEntity(HttpEntity entity)方法来设置请求参数。

4. 调用HttpClient对象的execute(HttpUriRequest request)发送请求，该方法返回一个HttpResponse。

5. 调用HttpResponse的getAllHeaders()、getHeaders(String name)等方法可获取服务器的响应头；调用HttpResponse的getEntity()方法可获取HttpEntity对象，该对象包装了服务器的响应内容。程序可通过该对象获取服务器的响应内容。

6. 释放连接。无论执行方法是否成功，都必须释放连接

## 七、试题

### 1、JAVA中的几种基本数据类型是什么，各自占用多少字节。

|  |  |
| --- | --- |
| 类型 | 字节数(JAVA是采用Unicode编码。每一个字节占8位) |
| byte | 1字节 |
| short | 2字节 |
| int | 4字节 |
| float | 4字节 |
| double | 8字节 |
| long | 8字节 |
| boolean | (理论上占用1bit,1/8字节，实际处理按1byte处理) |
| char | 2字节（C语言中是1字节）可以存储一个汉字 |

### 2、String类能被继承吗，为什么。

被final修饰，所以不能被继承.

### 3、String，Stringbuffer，StringBuilder的区别。

String为字符串常量，而StringBuilder和StringBuffer均为字符串变量，即String对象一旦创建之后该对象是不可更改的，但后两者的对象是变量，是可以更改的。StringBuilder是线程不安全的，而StringBuffer是线程安全的。

### 4、ArrayList和LinkedList有什么区别。

(1).ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。 （LinkedList是双向链表，有next也有previous）

(2).对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

(3).对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

### 5、讲讲类的实例化顺序，比如父类静态数据，构造函数，字段，子类静态数据，构造函数，字段，当new的时候，他们的执行顺序。

(1).父类静态成员和静态初始化块 ，按在代码中出现的顺序依次执行

(2).子类静态成员和静态初始化块 ，按在代码中出现的顺序依次执行

(3).父类实例成员和实例初始化块 ，按在代码中出现的顺序依次执行

(4).父类构造方法

(5).子类实例成员和实例初始化块 ，按在代码中出现的顺序依次执行

(6).子类构造方法

### 6、用过哪些Map类，都有什么区别，HashMap是线程安全的吗,并发下使用的Map是什么，他们内部原理分别是什么，比如存储方式，hashcode，扩容，默认容量等。

#### (1) HashMap与HashTable的区别

a.Hashtable中的对象是线程安全的。而HashMap则是异步的，因此HashMap中的对象并不是线程安全的。因为同步的要求会影响执行的率，所以如果你不需要线程安全的集合那么使用。

b.HashMap可以让你将空值作为一个表的条目的key或value，但是Hashtable是不能放入空值的。HashMap最多只有一个key值为null，但可以有无数多个value值为null。

#### (2) LinkedHashMap

它的父类是HashMap，使用双向链表来维护键值对的次序，迭代顺序与键值对的插入顺序保持一致。LinkedHashMap需要维护元素的插入序，插入性能略低于HashMap，但在迭代访问元素时有很好的性能，因为它是以链表来维护内部顺序。

#### (3) TreeMap和SortedMap

Map接口派生了一个SortMap子接口，SortMap的实现类为TreeMap。TreeMap也是基于红黑树对所有的key进行排序，有两种排序方式，排序和定制排序。HashMap通常比TreeMap快一点(树和哈希表的数据结构使然)，建议多使用HashMap，在需要排序的Map时候才用TreeMap。可以自动排序数字和字母，不能自动排汉字

排汉字：

TreeMap map = **new** TreeMap( **new** MyCmp2() );

**public** **class** MyCmp2 **implements** Comparator{}

#### (4) WeakHashMap

WeakHashMap与HashMap的用法基本相同，区别在于：后者的key保留对象的强引用，即只要HashMap对象不被销毁，其对象所有key所引用的对象不会被垃圾回收;WeakHashMap适合短时间内就过期的缓存时最好使用weakHashMap，它包含了一个自动调用的方法expungeStaleEntries，这样就会在值被引用后直接执行这个隐含的方法，将不用的键清除掉。

#### (6) EnumMap类

1、EnumMap中所有key都必须是单个枚举类的枚举值，创建EnumMap时必须显示或隐式指定它对应的枚举类。

2、EnumMap根据key的自然顺序，即枚举值在枚举类中定义的顺序，来维护键值对的次序。

3、EnumMap不允许使用null作为key值，但value可以。

#### (7) ConcurrentHashMap

1.ConcurrentHashMap对整个桶数组进行了分段，而HashMap则没有

2.ConcurrentHashMap在每一个分段上都用锁进行保护，从而让锁的粒度更精细一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。

**背景介绍**：ConcurrentHashMap从JDK1.5开始随java.util.concurrent包一起引入JDK中，主要为了解决HashMap线程不安全和Hashtable效率不高的问题。众所周知，HashMap在多线程编程中是线程不安全的，而Hashtable由于使用了synchronized修饰方法而导致执行效率不高；因此，在concurrent包中，实现了ConcurrentHashMap以使在多线程编程中可以使用一个高性能的线程安全HashMap方案。

而JDK1.7之前的ConcurrentHashMap使用分段锁机制实现，JDK1.8则使用**数组**+**链表**+**红黑树**数据结构和**CAS原子操作**实现ConcurrentHashMap。

### JAVA8的ConcurrentHashMap为什么放弃了分段锁，有什么问题吗，如果你来设计，你如何设计。

### 8、有没有有顺序的Map实现类，如果有，他们是怎么保证有序的。

TreeMap和LinkedHashMap是有序的（TreeMap默认升序，LinkedHashMap则记录了插入顺序）

### 9、抽象类和接口的区别，类可以继承多个类么，接口可以继承多个接口么,类可以实现多个接口么。

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

2、抽象类要被子类继承，接口要被类实现。

3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

5、抽象类里的抽象方法必须全部被非抽象子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类。

6、抽象方法只能申明，不能实现。abstract void abc();不能写成abstract void abc(){}。

7、抽象类里可以没有抽象方法

8、如果一个类里有抽象方法，那么这个类只能是抽象类

9、抽象方法要被实现，所以不能是静态的，也不能是私有的。

10、接口可继承接口，并可多继承接口，但类只能单根继承。

### 10、继承和聚合的区别在哪。

**聚合**是关联关系的一种特例，他体现的是整体与部分、拥有的关系，即has-a的关系，此时整体与部分之间是可分离的，他们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享；比如计算机与CPU、公司与员工的关系等；表现在代码层面，和关联关系是一致的，只能从语义级别来区分。**继承**指的是一个类继承另外的一个类的功能，并可以增加它自己的新功能的能力。

### 11、IO模型有哪些，讲讲你理解的nio ，他和bio，aio的区别是啥，谈谈reactor模型。

NIO（Non-blocking I/O，在Java领域，也称为New I/O），是一种同步非阻塞的I/O模型，也是I/O多路复用的基础，已经被越来越多地应用到大型应用服务器，成为解决高并发与大量连接、I/O处理问题的有效方式。

同步阻塞I/O处理（也就是BIO，Blocking I/O）

### 12、反射的原理，反射创建类实例的三种方式是什么。

反射的原理：如果知道一个类的名称/或者它的一个实例对象， 就能把这个类的所有方法和变量的信息(方法名，变量名，方法，修饰符，类型，方法参数等等所有信息)找出来。

反射创建类实例的三种方式：

1.Class.forName("com.A");

2.**new** A().getClass();

3.A.**class**;

### 13、反射中，Class.forName和ClassLoader区别 。

class.forName()除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块。

classLoader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容,只有在newInstance才会去执行static块。

### 14、描述动态代理的几种实现方式，分别说出相应的优缺点。

动态代理有两种实现方式，分别是：jdk动态代理和cglib动态代理

jdk动态代理的前提是目标类必须实现一个接口，代理对象跟目标类实现一个接口，从而避过虚拟机的校验。

cglib动态代理是继承并重写目标类，所以目标类和方法不能被声明成final

### 15、动态代理与cglib实现的区别。

### 16、为什么CGlib方式可以对接口实现代理。

cglib动态代理是继承并重写目标类，所以目标类和方法不能被声明成final。而接口是可以被继承的。

### 17、final的用途。

1.final修饰的对象不能被修改；

2.final修饰的类不能被继承；

3.final修饰的方法不能被重写；

### 18、写出三种单例模式实现 。

### 19、如何在父类中为子类自动完成所有的hashcode和equals实现？这么做有何优劣。

父类的equals不一定满足子类的equals需求。比如所有的对象都继承Object，默认使用的是Object的equals方法，在比较两个对象的时候，是看他们是否指向同一个地址。但是我们的需求是对象的某个属性相同，就相等了，而默认的equals方法满足不了当前的需求，所以我们要重写equals方法。

### 20、请结合OO设计理念，谈谈访问修饰符public、private、protected、default在应用设计中的作用。

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。所以我们可以通过public、private、protected、default 来进行访问控制。

### 21、深拷贝和浅拷贝区别。

浅拷贝只拷贝指针，深拷贝就是拷贝他的值，重新生成的对像。

### 22、数组和链表数据结构描述，各自的时间复杂度。

数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。链表恰好相反，链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。访问数组中第 n 个数据的时间花费是 O(1) 但是要在数组中查找一个指定的数据则是 O(N) 。当向数组中插入或者删除数据的时候，最好的情况是在数组的末尾进行操作，时间复杂度是 O(1) ，但是最坏情况是插入或者删除第一个数据，时间复杂度是 O(N) 。在数组的任意位置插入或者删除数据的时候，后面的数据全部需要移动，移动的数据还是和数据个数有关所以总体的时间复杂度仍然是 O(N) 。

### 23、error和exception的区别，CheckedException，RuntimeException的区别。

**Error:（错误）**表示系统级的错误和程序不必处理的异常，是java运行环境中的内部错误或者硬件问题。比如：内存资源不足等。对于这种错误，程序基本无能为力，除了退出运行外别无选择，它是由Java虚拟机抛出的。

**Exception:（违例）**表示需要捕捉或者需要程序进行处理的异常，它处理的是因为程序设计的瑕疵而引起的问题或者在外的输入等引起的一般性问题，是程序必须处理的。Exception又分为运行时异常，受检查异常。

**RuntimeException:(运行时异常)**，表示无法让程序恢复的异常，导致的原因通常是因为执行了错误的操作，建议终止程序，因此，编译器不检查这些异常。

**CheckedException:(受检查异常)**，是表示程序可以处理的异常，也即表示程序可以修复（由程序自己接受异常并且做出处理）， 所以称之为受检查异常。

Throwable

Exception

error

ViualMachineError

IOException

RuntimeException

AWTEerror

stackOverFlowError

OutOfMemoryError

EOFException

FileNotFountException

ArithmeticException

MissingResourceException

ClassNotFoundException

NullPointerException

BufferOverflowException

将派生于Error或者RuntimeException的异常称为unchecked异常，所有其他的异常成为checked异常。RuntimeException是在编程过程中可以避免的错误，不需要声明或者try catch异常

### 24、请列出5个运行时异常。

NullPointerException

IndexOutOfBoundsException

ClassCastException

ArrayStoreException

BufferOverflowException

### 25、在自己的代码中，如果创建一个java.lang.String类，这个类是否可以被类加载器加载？为什么。

不可以，双亲委派模式会保证父类加载器先加载类，就是BootStrap（启动类）加载器加载jdk里面的java.lang.String类，而自定义的java.lang.String类永远不会被加载到。

### 26、说一说你对java.lang.Object对象中hashCode和equals方法的理解。在什么场景下需要重新实现这两个方法。

父类的equals不一定满足子类的equals需求。比如所有的对象都继承Object，默认使用的是Object的equals方法，在比较两个对象的时候，是看他们是否指向同一个地址。但是我们的需求是对象的某个属性相同，就相等了，而默认的equals方法满足不了当前的需求，所以我们要重写equals方法。如果重写了equals 方法就必须重写hashcode方法，否则就会降低map等集合的索引速度（hashcode是根据对象的内存地址经哈希算法得来的，如果你重载了equals，比如说是基于对象的内容实现的，而保留hashCode的实现不变，那么很可能某两个对象明明是“相等”，而hashCode却不一样。这样，当你用其中的一个作为键保存到hashMap、hasoTable或hashSet中，再以“相等的”找另一个作为键值去查找他们的时候，则根本找不到）。

### 27、在jdk1.5中，引入了泛型，泛型的存在是用来解决什么问题。

泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，提高代码的重用率。

### 28、这样的a.hashcode() 有什么用，与a.equals(b)有什么关系。

### 29、有没有可能2个不相等的对象有相同的hashcode。

/\*

\*hashCode的常规协定是：

\*1.在java应用程序执行期间,在对同一对象多次调用hashCode()方法时,必须一致地返回相同的整数,前提是将对象进行equals比较时所用

\*的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行,该整数无需保持一致。

\*2.如果根据equals(object)方法,两个对象是相等的,那么对这两个对象中的每个对象调用hashCode方法都必须生成相同的整数结果。

\*3.如果根据equals(java.lang.Object)方法,两个对象不相等,那么对这两个对象中的任一对象上调用hashCode()方法不要求一定生成

\*不同的整数结果。但是,程序员应该意识到,为不相等的对象生成不同整数结果可以提高哈希表的性能。

\*/

**public** **native** **int** hashCode();

/\*这里比较的是对象的内存地址,跟String.equals方法不同,它比较的只是对象的值\*/

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

**return** (**this** == obj);

}

有可能，最简单的方法，百分百实现的方式就是重写hascode();任何数n\*31都可以被jvm优化为(n<<5)-n，移位和减法的操作效率比乘法的操作效率高很多。

@Override

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

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + ((name == **null**) ? 0 : name.hashCode());

**return** result;

}

### 30、Java中的HashSet内部是如何工作的。

**public** HashSet() {

map = **new** HashMap<>();

}

默认使用的是HaseMap;

### 31、什么是序列化，怎么序列化，为什么序列化，反序列化会遇到什么问题，如何解决。

序列化是一种用来处理对象流的机制 ,所谓对象流就是将对象的内容进行流化。是为了解决在对对象流进行读写操作时所引发的问题。

**序列化的实现**：将需要被序列化的类实现Serializable接口，该接口没有需要实现的方法，**implements** Serializable只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个ObjectOutputStream(对象流)对象，接着，使用ObjectOutputStream对象的writeObject(Objectobj)方法就可以将参数为obj的对象写出(即保存其状态)，要恢复的话则用输入流。

**private** **static** **final** **long** ***serialVersionUID*** = -1672970955045193907L;

如果你修改了此类,要修改此值。否则以前用老版本的类序列化的类恢复时会出错。为了在反序列化时，确保类版本的兼容性，最好在每个要序列化的类中加入**private** **static** **final** **long** ***serialVersionUID***这个属性，具体数值自己定义。

SerialVersionUID，后面简称SUID，是当对象序列化的时候对象的一个标识，你可以使用工具serialver查看一个序列化对象的SUID。SUID用于控制对象的版本。你也可以在类文件中指定SUID。不指定SUID的结果就是当你添加或者更改类的域并已经序列化类的时候，类是不能再恢复了，因为新的SUID和之前的SUID不同了。Java的序列化过程依赖于正确的SUID来反序列化已经序列化的对象，如果SUID不匹配，那么就会抛 java.io.InvalidClassException 异常了。

SUID是源类的哈希值，如果类更新，例如域的改变，SUID会变化。

1.忽略SUID,相当于运行期间类的版本上的序列化和反序列上面没有差异。

2.写一个默认的SUID,这就好像线程头部。告诉JVM所有版本中有着同样SUID的都是同一个版本。

3.复制之前版本类的SUID。运行期间这个版本和之前版本是一样的版本。

4.使用类每个版本生成的SUID。如果SUID与新版本的类不同，那么运行期间两个版本是不同的，并且老版本类序列化后的实例并不可以反序列成新的类的实例。

其实序列化的作用是能转化成Byte流，然后又能反序列化成原始的类。能在网络进行传输，也可以保存在磁盘中，有了SUID之后，那么如果序列化的类已经保存了在本地中，中途你更改了类后，SUID变了，那么反序列化的时候就不会变成原始的类了，还会抛异常，主要就是用于版本控制

如果一个序列化类没有显式声明的serialVersionUID，则序列化运行时将计算默认的serialVersionUID值基于类的各个方面的类，如在Java（TM）中描述的对象序列化规范。然而，强烈建议所有可序列化类中明确声明的serialVersionUID值，因为默认的serialVersionUID的计算是类细节取决于编译器实现，可能会发生变化高度敏感，并因此可能导致反序列化过程中意外InvalidClassExceptions。因此，为了保证不同java编译器实现了一致的serialVersionUID值，序列化类必须声明一个明确的serialVersionUID值。它也强烈建议明确的serialVersionUID声明使用私有修饰符在可能的情况，因为这样的声明仅适用于立即声明类 - 的serialVersionUID字段不作为继承的成员是有用的。

### 32、面向对象的特征有哪些方面？

面向对象的特征主要有以下几个方面：

**抽象**：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

**继承：**继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段（如果不能理解请阅读阎宏博士的《Java与模式》或《设计模式精解》中关于桥梁模式的部分）。

**封装**：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口（可以想想普通洗衣机和全自动洗衣机的差别，明显全自动洗衣机封装更好因此操作起来更简单；我们现在使用的智能手机也是封装得足够好的，因为几个按键就搞定了所有的事情）。

**多态性**：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当A系统访问B系统提供的服务时，B系统有多种提供服务的方式，但一切对A系统来说都是透明的（就像电动剃须刀是A系统，它的供电系统是B系统，B系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A系统只会通过B类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：

1). 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；

2). 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

### 33、访问修饰符public,private,protected,以及不写（默认）时的区别？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 修饰符 | 当前类 | 同 包 | 子 类 | 其他包 |
| public | √ | √ | √ | √ |
| protected | √ | √ | √ | × |
| default | √ | √ | × | × |
| private | √ | × | × | × |

类的成员不写访问修饰时默认为default。默认对于同一个包中的其他类相当于公开（public），对于不是同一个包中的其他类相当于私有（private）。受保护（protected）对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。Java中，外部类的修饰符只能是public或默认，类的成员（包括内部类）的修饰符可以是以上四种。

### 34、String 是最基本的数据类型吗？

答：不是。Java中的基本数据类型只有8个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type），剩下的都是引用类型（reference type），Java 5以后引入的枚举类型也算是一种比较特殊的引用类型。

### 35、float f=3.4;是否正确？

答:不正确。3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;。

### 36、short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗？

答：对于short s1 = 1; s1 = s1 + 1;由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。而short s1 = 1; s1 += 1;可以正确编译，因为s1+= 1;相当于s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

### 37、Java有没有goto？

答：goto 是Java中的保留字，在目前版本的Java中没有使用。（根据James Gosling（Java之父）编写的《The Java Programming Language》一书的附录中给出了一个Java关键字列表，其中有goto和const，但是这两个是目前无法使用的关键字，因此有些地方将其称之为保留字，其实保留字这个词应该有更广泛的意义，因为熟悉C语言的程序员都知道，在系统类库中使用过的有特殊意义的单词或单词的组合都被视为保留字）

### 38、int和Integer有什么区别？

答：Java是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的包装类型（wrapper class），int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

原始类型: boolean，char，byte，short，int，long，float，double

包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

**class** AutoUnboxingTest {

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

Integer a = **new** Integer(3);

Integer b = 3; // 将3自动装箱成Integer类型

**int** c = 3;

System.***out***.println(a == b); // false 两个引用没有引用同一对象

System.***out***.println(a == c); // true a自动拆箱成int类型再和c比较

}

}

最近还遇到一个面试题，也是和自动装箱和拆箱有点关系的，代码如下所示：

**public** **class** Test03 {

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

Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;

System.***out***.println(f1 == f2);

System.***out***.println(f3 == f4);

}

}

如果不明就里很容易认为两个输出要么都是true要么都是false。首先需要注意的是f1、f2、f3、f4四个变量都是Integer对象引用，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf，如果看看valueOf的源代码就知道发生了什么。

**public** **static** Integer valueOf(**int** i) {

**if** (i >= IntegerCache.low && i <= IntegerCache.high)

**return** IntegerCache.cache[i + (-IntegerCache.low)];

**return** **new** Integer(i);

}

IntegerCache是Integer的内部类，其代码如下所示：

/\*\*

\* Cache to support the object identity semantics of autoboxing for values between

\* -128 and 127 (inclusive) as required by JLS.

\*

\* The cache is initialized on first usage. The size of the cache

\* may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.

\* During VM initialization, java.lang.Integer.IntegerCache.high property

\* may be set and saved in the private system properties in the

\* sun.misc.VM class.

\*/

**private** **static** **class** IntegerCache {

**static** **final** **int** ***low*** = -128;

**static** **final** **int** ***high***;

**static** **final** Integer ***cache***[];

**static** {

// high value may be configured by property

**int** h = 127;

String integerCacheHighPropValue =

sun.misc.VM.*getSavedProperty*("java.lang.Integer.IntegerCache.high");

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

**try** {

**int** i = parseInt(integerCacheHighPropValue);

i = Math.*max*(i, 127);

// Maximum array size is Integer.MAX\_VALUE

h = Math.*min*(i, Integer.***MAX\_VALUE*** - (-***low***) -1);

} **catch**( NumberFormatException nfe) {

// If the property cannot be parsed into an int, ignore it.

}

}

***high*** = h;

***cache*** = **new** Integer[(***high*** - ***low***) + 1];

**int** j = ***low***;

**for**(**int** k = 0; k < ***cache***.length; k++)

***cache***[k] = **new** Integer(j++);

// range [-128, 127] must be interned (JLS7 5.1.7)

**assert** IntegerCache.***high*** >= 127;

}

**private** IntegerCache() {}

}

简单的说，如果整型字面量的值在-128到127之间，那么不会new新的Integer对象，而是直接引用常量池中的Integer对象，所以上面的面试题中f1==f2的结果是true，而f3==f4的结果是false。

提醒：越是貌似简单的面试题其中的玄机就越多，需要面试者有相当深厚的功力。

### 39、&和&&的区别？

答：&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是true整个表达式的值才是true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是null而且不是空字符串，应当写为：username != null &&!username.equals("")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的equals比较，否则会产生NullPointerException异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

### 40、解释内存中的栈(stack)、堆(heap)和方法区(method area)的用法。

答：通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用JVM中的栈空间；而通过new关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为Eden、Survivor（又可分为From Survivor和To Survivor）、Tenured；方法区和堆都是各个线程共享的内存区域，用于存储已经被JVM加载的类信息、常量、静态变量、JIT编译器编译后的代码等数据；程序中的字面量（literal）如直接书写的100、"hello"和常量都是放在常量池中，常量池是方法区的一部分，。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过JVM的启动参数来进行调整，栈空间用光了会引发StackOverflowError，而堆和常量池空间不足则会引发OutOfMemoryError。

String str = **new** String("hello");

上面的语句中变量str放在栈上，用new创建出来的字符串对象放在堆上，而"hello"这个字面量是放在方法区的。

补充1：较新版本的Java（从Java 6的某个更新开始）中，由于JIT编译器的发展和"逃逸分析"技术的逐渐成熟，栈上分配、标量替换等优化技术使得对象一定分配在堆上这件事情已经变得不那么绝对了。

补充2：运行时常量池相当于Class文件常量池具有动态性，Java语言并不要求常量一定只有编译期间才能产生，运行期间也可以将新的常量放入池中，String类的intern()方法就是这样的。

看看下面代码的执行结果是什么并且比较一下Java 7以前和以后的运行结果是否一致。

String s1 = **new** StringBuilder("go").append("od").toString();

System.out.println(s1.intern() == s1);

String s2 = **new** StringBuilder("ja").append("va").toString();

System.out.println(s2.intern() == s2);

调用这个方法之后把字符串对象加入常量池中，常量池我们都知道他是存在于方法区的，他是方法区的一部分，而方法区是线程共享的，所以常量池也就是线程共享的，但是他并不是线程不安全的，他其实是线程安全的，他仅仅是让有相同值的引用指向同一个位置而已，如果引用值变化了，但是常量池中没有新的值，那么就会新开辟一个常量结果来交给新的引用，而并非像线程不同步那样，针对同一个对象，new出来的字符串和直接赋值给变量的字符串存放的位置是不一样的，前者是在堆里面，而后者在常量池里面，另外，在做字符串拼接操作，也就是字符串相"+"的时候，得出的结果是存在在常量池或者堆里面.

### 41、Math.round(11.5) 等于多少？Math.round(-11.5)等于多少？

答：Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加0.5然后进行下取整。

### 42、switch 是否能作用在byte 上，是否能作用在long 上，是否能作用在String上？

答：在Java 5以前，switch(expr)中，expr只能是byte、short、char、int。从Java 5开始，Java中引入了枚举类型， expr也可以是enum类型，从Java 7开始，expr还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

### 43、用最有效率的方法计算2乘以8？

答： 2 << 3（左移3位相当于乘以2的3次方，右移3位相当于除以2的3次方）。

补充：我们为编写的类重写hashCode方法时，可能会看到如下所示的代码，其实我们不太理解为什么要使用这样的乘法运算来产生哈希码（散列码），而且为什么这个数是个素数，为什么通常选择31这个数？前两个问题的答案你可以自己百度一下，选择31是因为可以用移位和减法运算来代替乘法，从而得到更好的性能。说到这里你可能已经想到了：31 \* num 等价于(num << 5) - num，左移5位相当于乘以2的5次方再减去自身就相当于乘以31，现在的VM都能自动完成这个优化。

**public** **class** PhoneNumber {

**private** **int** areaCode;

**private** String prefix;

**private** String lineNumber;

@Override

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

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + areaCode;

result = prime \* result

+ ((lineNumber == **null**) ? 0 : lineNumber.hashCode());

result = prime \* result + ((prefix == **null**) ? 0 : prefix.hashCode());

**return** result;

}

@Override

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

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

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

PhoneNumber other = (PhoneNumber) obj;

**if** (areaCode != other.areaCode)

**return** **false**;

**if** (lineNumber == **null**) {

**if** (other.lineNumber != **null**)

**return** **false**;

} **else** **if** (!lineNumber.equals(other.lineNumber))

**return** **false**;

**if** (prefix == **null**) {

**if** (other.prefix != **null**)

**return** **false**;

} **else** **if** (!prefix.equals(other.prefix))

**return** **false**;

**return** **true**;

}

}

### 44、数组有没有length()方法？String有没有length()方法？

答：数组没有length()方法，有length 的属性。String 有length()方法。JavaScript中，获得字符串的长度是通过length属性得到的，这一点容易和Java混淆。

### 45、在Java中，如何跳出当前的多重嵌套循环？

答：在最外层循环前加一个标记如A，然后用break A;可以跳出多重循环。（Java中支持带标签的break和continue语句，作用有点类似于C和C++中的goto语句，但是就像要避免使用goto一样，应该避免使用带标签的break和continue，因为它不会让你的程序变得更优雅，很多时候甚至有相反的作用，所以这种语法其实不知道更好）

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

**int** i,j;

itcast:**for**(i=1;i<9;i++){

**for**(j=1;j<=i;j++){

**if**(i>4){

**break** itcast;//break跳出itcast标记的循环

}

System.***out***.print("\*");

}

System.***out***.print("\n");

}

}

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

**int** i,j;

itcast:**for**(i=1;i<9;i++){

**for**(j=1;j<=i;j++){

**if**(i>4){

**continue** itcast;//continue跳出itcast标记的本次循环，继续下一次循环

}

System.***out***.print("\*");

}

System.***out***.print("\n");

}

}

### 46、构造器（constructor）是否可被重写（override）？

答：构造器不能被继承，因此不能被重写，但可以被重载。

### 47、两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？

答：不对，如果两个对象x和y满足x.equals(y) == true，它们的哈希码（hash code）应当相同。Java对于eqauls方法和hashCode方法是这样规定的：

(1)如果两个对象相同（equals方法返回true），那么它们的hashCode值一定要相同；

(2)如果两个对象的hashCode相同，它们并不一定相同。

当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在Set集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降）。

补充：关于equals和hashCode方法，很多Java程序都知道，但很多人也就是仅仅知道而已，在Joshua Bloch的大作《Effective Java》（很多软件公司，《Effective Java》、《Java编程思想》以及《重构：改善既有代码质量》是Java程序员必看书籍，如果你还没看过，那就赶紧去亚马逊买一本吧）中是这样介绍equals方法的：首先equals方法

必须满足**自反性**（x.equals(x)必须返回true）、**对称性**（x.equals(y)返回true时，y.equals(x)也必须返回true）、**传递性**（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true）**和一致性**（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。实现高质量的equals方法的诀窍包括：

1. 使用==操作符检查"参数是否为这个对象的引用"；

2. 使用instanceof操作符检查"参数是否为正确的类型"；

3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；

4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；

5. 重写equals时总是要重写hashCode；

6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

### 48、是否可以继承String类？

答：String 类是final类，不可以被继承。

补充：继承String本身就是一个错误的行为，对String类型最好的重用方式是关联关系（Has-A）和依赖关系（Use-A）而不是继承关系（Is-A）。

### 49、当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？

答：是值传递。Java语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。在C#中可以编写如下所示的代码，但是在Java中却做不到。

using System;

namespace CS01 {

**class** Program {

**public** **static** **void** swap(ref **int** x, ref **int** y) {

**int** temp = x;

x = y;

y = temp;

}

**public** **static** **void** Main (string[] args) {

**int** a = 5, b = 10;

swap (ref a, ref b);

// a = 10, b = 5;

Console.WriteLine ("a = {0}, b = {1}", a, b);

}

}

}

说明：Java中没有传引用实在是非常的不方便，这一点在Java 8中仍然没有得到改进，正是如此在Java编写的代码中才会出现大量的Wrapper类（将需要通过方法调用修改的引用置于一个Wrapper类中，再将Wrapper对象传入方法），这样的做法只会让代码变得臃肿，尤其是让从C和C++转型为Java程序员的开发者无法容忍。Java wrapper类旨在将原始类型存储为引用类型，并将其视为实例化内存对象的引用。 这样做的主要好处是，特定原始类型的相应wrapper类可以容纳许多类型的信息，以便提供诸如从一种类型到另一种类型的转换以及大量其他方便的功能。 因为Java中有八个基本类型。

### 50、String和StringBuilder、StringBuffer的区别？

答：Java平台提供了两种类型的字符串：String和StringBuffer/StringBuilder，它们可以储存和操作字符串。其中**String是只读字符串**，也就意味着String引用的字符串内容是不能被改变的。而StringBuffer/StringBuilder类表示的字符串对象可以直接进行修改。StringBuilder是Java 5中引入的，它和StringBuffer的方法完全相同，

区别在于它是在单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer要高。

**面试题1** : 什么情况下用+运算符进行字符串连接比调用StringBuffer/StringBuilder对象的append方法连接字符串性能更好？

如果能保证在一条语句中把字符串或者字符串变量全部拼接完而不断开，那+拼接根本没有任何缺点，编译器还会有一些优化，如果拼接的全是字符串字面量，那效果更好，它们直接就变成了一个完整的字符串。

<https://blog.csdn.net/m0_37589327/article/details/78605268>

**面试题2**: 请说出下面程序的输出。

**class** StringEqualTest {

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

String s1 = "Programming";

String s2 = **new** String("Programming");

String s3 = "Program";

String s4 = "ming";

String s5 = "Program" + "ming";

String s6 = s3 + s4;

System.***out***.println(s1 == s2);

System.***out***.println(s1 == s5);

System.***out***.println(s1 == s6);

System.***out***.println(s1 == s6.intern());

System.***out***.println(s2 == s2.intern());

}

}

补充：解答上面的面试题需要清楚两点：

1. String对象的intern方法会得到字符串对象在常量池中对应的版本的引用（如果常量池中有一个字符串与String对象的equals结果是true），如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用；

2. 字符串的+操作其本质是创建了StringBuilder对象进行

append操作，然后将拼接后的StringBuilder对象用toString方法处理成String对象，这一点可以用 javap -c StringEqualTest.class命令获得class文件对应的JVM字节码指令就可以看出来。

### 51、重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？

答：方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求。

面试题：华为的面试题中曾经问过这样一个问题 - "为什么不能根据返回类型来区分重载"，快说出你的答案吧！

### 52、描述一下JVM加载class文件的原理机制？

答：JVM中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java中的类加载器是一个重要的Java运行时系统组件，它负责在运行时查找和装入类文件中的类。 由于Java的跨平台性，经过编译的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后JVM对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM更好的保证了Java平台的安全性，在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。下面是关于几个类加载器的说明：

**Bootstrap**：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；

**Extension**：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；

**System**：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。

### 53、char 型变量中能不能存贮一个中文汉字，为什么？

答：char类型可以存储一个中文汉字，因为Java中使用的编码是Unicode（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），一个char类型占2个字节（16比特），所以放一个中文是没问题的。

补充：使用Unicode意味着字符在JVM内部和外部有不同的表现形式，在JVM内部都是Unicode，当这个字符被从JVM内部转移到外部时（例如存入文件系统中），需要进行编码转换。所以Java中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如InputStreamReader和OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；对于C程序员来说，要完成这样的编码转换恐怕要依赖于union（联合体/共用体）共享内存的特征来实现了。

### 54、抽象类（abstract class）和接口（interface）有什么异同？

答：抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。**接口比抽象类更加抽象**，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是private、默认、protected、public的，而接口中的成员全都是public的。抽象类中可以定义变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

### 55、静态嵌套类(Static Nested Class)和内部类（Inner Class）的不同？

答：Static Nested Class是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化，其语法看起来挺诡异的，如下所示。

/\*\*

\* 扑克类（一副扑克）

\*/

**public** **class** Poker {

**private** **static** String[] *suites* = {"黑桃", "红桃", "草花", "方块"};

**private** **static** **int**[] *faces* = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

**private** Card[] cards;

/\*\*

\* 构造器

\*/

**public** Poker() {

cards = **new** Card[52];

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

**for**(**int** j = 0; j < *faces*.length; j++) {

cards[i \* 13 + j] = **new** Card(*suites*[i], *faces*[j]);

}

}

}

/\*\*

\* 洗牌 （随机乱序）

\*/

**public** **void** shuffle() {

**for**(**int** i = 0, len = cards.length; i < len; i++) {

**int** index = (**int**) (Math.*random*() \* len);

Card temp = cards[index];

cards[index] = cards[i];

cards[i] = temp;

}

}

/\*\*

\* 发牌

\* **@param** index 发牌的位置

\*/

**public** Card deal(**int** index) {

**return** cards[index];

}

/\*\*

\* 卡片类（一张扑克）

\* [内部类]

\*

\*/

**public** **class** Card {

**private** String suite; // 花色

**private** **int** face; // 点数

**public** Card(String suite, **int** face) {

**this**.suite = suite;

**this**.face = face;

}

@Override

**public** String toString() {

String faceStr = "";

**switch**(face) {

**case** 1: faceStr = "A"; **break**;

**case** 11: faceStr = "J"; **break**;

**case** 12: faceStr = "Q"; **break**;

**case** 13: faceStr = "K"; **break**;

**default**: faceStr = String.*valueOf*(face);

}

**return** suite + faceStr;

}

}

}

测试代码：

**class** PokerTest {

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

Poker poker = **new** Poker();

poker.shuffle(); // 洗牌

Poker.Card c1 = poker.deal(0); // 发第一张牌

// 对于非静态内部类Card

// 只有通过其外部类Poker对象才能创建Card对象

Poker.Card c2 = poker.**new** Card("红心", 1); // 自己创建一张牌

System.***out***.println(c1); // 洗牌后的第一张

System.***out***.println(c2); // 打印: 红心A

}

}

面试题 - 下面的代码哪些地方会产生编译错误？

**class** Outer {

**class** Inner {}

**public** **static** **void** foo() { **new** Inner(); }

**public** **void** bar() { **new** Inner(); }

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

**new** Inner();

}

}

注意：Java中非静态内部类对象的创建要依赖其外部类对象，上面的面试题中foo和main方法都是静态方法，静态方法中没有this，也就是说没有所谓的外部类对象，因此无法创建内部类对象，如果要在静态方法中创建内部类对象，可以这样做：

**new** Outer().**new** Inner();

### 56、Java 中会存在内存泄漏吗，请简单描述。

答：理论上Java因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是Java被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被GC回收，因此也会导致内存泄露的发生。

例如Hibernate的Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。

**import** java.util.Arrays;

**import** java.util.EmptyStackException;

**public** **class** MyStack<T> {

**private** T[] elements;

**private** **int** size = 0;

**private** **static** **final** **int** ***INIT\_CAPACITY*** = 16;

**public** MyStack() {

elements = (T[]) **new** Object[***INIT\_CAPACITY***];

}

**public** **void** push(T elem) {

ensureCapacity();

elements[size++] = elem;

}

**public** T pop() {

**if**(size == 0)

**throw** **new** EmptyStackException();

**return** elements[--size];

}

**private** **void** ensureCapacity() {

**if**(elements.length == size) {

elements = Arrays.*copyOf*(elements, 2 \* size + 1);

}

}

}

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的pop方法却存在内存泄露的问题，当我们用pop方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成OutOfMemoryError。

### 57、抽象的（abstract）方法是否可同时是静态的（static）,是否可同时是本地方法（native），是否可同时被synchronized修饰？

答：都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如C代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

### 58、阐述静态变量和实例变量的区别。

答：静态变量是被static修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；实例变量必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

补充：在Java开发中，上下文类和工具类中通常会有大量的静态成员。

### 59、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？

答：不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

### 60、如何实现对象克隆？

答：有两种方式：

(1). 实现Cloneable接口并重写Object类中的clone()方法；

(2). 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下。

**import** java.io.ByteArrayInputStream;

**import** java.io.ByteArrayOutputStream;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**public** **class** MyUtil {

**private** MyUtil() {

**throw** **new** AssertionError();

}

@SuppressWarnings("unchecked")

**public** **static** <T **extends** Serializable> T clone(T obj) **throws** Exception {

ByteArrayOutputStream bout = **new** ByteArrayOutputStream();

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

oos.writeObject(obj);

ByteArrayInputStream bin = **new** ByteArrayInputStream(bout.toByteArray());

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

**return** (T) ois.readObject();

/\*

\* 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义

\*ByteArrayOutputStream或ByteArrayInputStream是内存读写流，不同于指向硬盘的流，它内部是使用字节数组读内存的，

\*这个字节数组是它的成员变量，当这个数组不再使用变成垃圾的时候，Java的垃圾回收机制会将它回收。所以不需要关流

\*/

}

}

下面是测试代码：

**import** java.io.Serializable;

/\*\*

\* 人类

\*/

**class** Person **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = -9102017020286042305L;

**private** String name; // 姓名

**private** **int** age; // 年龄

**private** Car car; // 座驾

**public** Person(String name, **int** age, Car car) {

**this**.name = name;

**this**.age = age;

**this**.car = car;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** Car getCar() {

**return** car;

}

**public** **void** setCar(Car car) {

**this**.car = car;

}

@Override

**public** String toString() {

**return** "Person [name=" + name + ", age=" + age + ", car=" + car + "]";

}

}

/\*\*

\* 小汽车类

\*/

**class** Car **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = -5713945027627603702L;

**private** String brand; // 品牌

**private** **int** maxSpeed; // 最高时速

**public** Car(String brand, **int** maxSpeed) {

**this**.brand = brand;

**this**.maxSpeed = maxSpeed;

}

**public** String getBrand() {

**return** brand;

}

**public** **void** setBrand(String brand) {

**this**.brand = brand;

}

**public** **int** getMaxSpeed() {

**return** maxSpeed;

}

**public** **void** setMaxSpeed(**int** maxSpeed) {

**this**.maxSpeed = maxSpeed;

}

@Override

**public** String toString() {

**return** "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";

}

}

**class** CloneTest {

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

**try** {

Person p1 = **new** Person("Hao LUO", 33, **new** Car("Benz", 300));

Person p2 = MyUtil.*clone*(p1); // 深度克隆

p2.getCar().setBrand("BYD");

// 修改克隆的Person对象p2关联的汽车对象的品牌属性

// 原来的Person对象p1关联的汽车不会受到任何影响

// 因为在克隆Person对象时其关联的汽车对象也被克隆了

System.***out***.println(p1);

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

注意：基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

### 61、GC是什么？为什么要有GC？

答：GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显式操作方法。Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但JVM可以屏蔽掉显示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。

补充：垃圾回收机制有很多种，包括：**分代复制垃圾回收**、**标记垃圾回收**、**增量垃圾回收**等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

**伊甸园（Eden）**：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。

**幸存者乐园（Survivor）**：从伊甸园幸存下来的对象会被挪到这里。

**终身颐养园（Tenured）**：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。

当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

垃圾回收相关的JVM参数：

-Xms /-Xmx ：堆的初始大小/堆的最大大小

-Xmn ：堆中年轻代的大小

-XX:-DisableExplicitGC ：让System.gc()不产生任何作用

-XX:+PrintGCDetails ：打印GC的细节

-XX:+PrintGCDateStamps ：打印GC操作的时间戳

-XX:NewSize / XX:MaxNewSize ：设置新生代大小/新生代最大大小

-XX:NewRatio ：可以设置老生代和新生代的比例

-XX:PrintTenuringDistribution ：设置每次新生代GC后输出幸存者乐园中对象年龄的分布

-XX:InitialTenuringThreshold/-XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值

-XX:TargetSurvivorRatio：设置幸存区的目标使用率

### 62、String s = new String("xyz");创建了几个字符串对象？

两个对象，一个是静态区的"xyz"，一个是用new创建在堆上的对象。

### 63、接口是否可继承（extends）接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concrete class）？

接口可以继承接口，而且支持多重继承。抽象类可以实现(implements)接口，抽象类可继承具体类也可以继承抽象类。

### 64、一个".java"源文件中是否可以包含多个类（不是内部类）？有什么限制？

答：可以，但一个源文件中最多只能有一个公开类（public class）而且文件名必须和公开类的类名完全保持一致。

### 65、Anonymous Inner Class(匿名内部类)是否可以继承其它类？是否可以实现接口？

答：可以继承其他类或实现其他接口，在Swing编程和Android开发中常用此方式来实现事件监听和回调。

### 66、内部类可以引用它的包含类（外部类）的成员吗？有没有什么限制？

答：一个内部类对象可以访问创建它的外部类对象的成员，包括私有成员。

### 67、指出下面程序的运行结果。

**class** A {

**static** {

System.***out***.print("1");

}

**public** A() {

System.***out***.print("2");

}

}

**class** B **extends** A{

**static** {

System.***out***.print("a");

}

**public** B() {

System.***out***.print("b");

}

}

**public** **class** Hello {

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

A ab = **new** B();

ab = **new** B();

}

}

答：执行结果：1a2b2b。创建对象时构造器的调用顺序是：先初始化静态成员，然后调用父类构造器，再初始化非静态成员，最后调用自身构造器。

### 68、数据类型之间的转换：

如何将字符串转换为基本数据类型？如何将基本数据类型转换为字符串？

调用基本数据类型对应的包装类中的方法parseXXX(String)或valueOf(String)即可返回相应基本类型；

一种方法是将基本数据类型与空字符串（""）连接（+）即可获得其所对应的字符串；另一种方法是调用String 类中的valueOf()方法返回相应字符串。

### 69、如何实现字符串的反转及替换？

答：方法很多，可以自己写实现也可以使用String或StringBuffer/StringBuilder中的方法。有一道很常见的面试题是用递归实现字符串反转，代码如下所示：

**public** **static** String reverse(String originStr) {

**if**(originStr == **null** || originStr.length() <= 1)

**return** originStr;

**return** *reverse*(originStr.substring(1)) + originStr.charAt(0);

}

**new** StringBuilder(str).reverse().toString();

### 怎样将GB2312编码的字符串转换为ISO-8859-1编码的字符串？

String s1 = "你好";

String s2 = **new** String(s1.getBytes("GB2312"), "ISO-8859-1");

### 日期和时间

-如何取得年月日、小时分钟秒？

创建java.util.Calendar 实例，调用其get()方法传入不同的参数即可获得参数所对应的值。Java 8中可以使用java.time.LocalDateTimel来获取，代码如下所示。

**public** **class** DateTimeTest {

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

Calendar cal = Calendar.getInstance();

System.***out***.println(cal.get(Calendar.YEAR));

System.***out***.println(cal.get(Calendar.MONTH)); // 0 - 11

System.***out***.println(cal.get(Calendar.DATE));

System.***out***.println(cal.get(Calendar.HOUR\_OF\_DAY));

System.***out***.println(cal.get(Calendar.MINUTE));

System.***out***.println(cal.get(Calendar.SECOND));

// Java 8

LocalDateTime dt = LocalDateTime.now();

System.***out***.println(dt.getYear());

System.***out***.println(dt.getMonthValue()); // 1 - 12

System.***out***.println(dt.getDayOfMonth());

System.***out***.println(dt.getHour());

System.***out***.println(dt.getMinute());

System.***out***.println(dt.getSecond());

}

}

- 如何取得从1970年1月1日0时0分0秒到现在的毫秒数？

Calendar.getInstance().getTimeInMillis();

System.currentTimeMillis();

Clock.systemDefaultZone().millis(); // Java 8

- 如何取得某月的最后一天？

Calendar time = Calendar.getInstance();

time.getActualMaximum(Calendar.DAY\_OF\_MONTH);

- 如何格式化日期？

**import** java.text.SimpleDateFormat;

**import** java.time.LocalDate;

**import** java.time.format.DateTimeFormatter;

**import** java.util.Date;

**class** DateFormatTest {

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

SimpleDateFormat oldFormatter = **new** SimpleDateFormat("yyyy/MM/dd");

Date date1 = **new** Date();

System.***out***.println(oldFormatter.format(date1));

// Java 8

DateTimeFormatter newFormatter = DateTimeFormatter.*ofPattern*("yyyy/MM/dd");

LocalDate date2 = LocalDate.*now*();

System.***out***.println(date2.format(newFormatter));

}

}

补充：Java的时间日期API一直以来都是被诟病的东西，为了解决这一问题，Java 8中引入了新的时间日期API，其中包括LocalDate、LocalTime、LocalDateTime、Clock、Instant等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。如果不理解这些内容，可以参考我的另一篇文章《关于Java并发编程的总结和思考》。

### 72、打印昨天的当前时刻。

**import** java.util.Calendar;

**class** YesterdayCurrent {

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

Calendar cal = Calendar.*getInstance*();

cal.add(Calendar.***DATE***, -1);

System.***out***.println(cal.getTime());

}

}

在Java 8中，可以用下面的代码实现相同的功能。

**import** java.time.LocalDateTime;

**class** YesterdayCurrent {

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

LocalDateTime today = LocalDateTime.*now*();

LocalDateTime yesterday = today.minusDays(1);

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

}

}

### 73、比较一下Java和JavaSciprt。

JavaScript 与Java是两个公司开发的不同的两个产品。Java 是原Sun Microsystems公司推出的面向对象的程序设计语言，特别适合于互联网应用程序开发；而JavaScript是Netscape公司的产品，为了扩展Netscape浏览器的功能而开发的一种可以嵌入Web页面中运行的基于对象和事件驱动的解释性语言。JavaScript的前身是LiveScript；而Java的前身是Oak语言。 下面对两种语言间的异同作如下比较：

**基于对象和面向对象**：Java是一种真正的面向对象的语言，即使是开发简单的程序，必须设计对象； JavaScript是种脚本语言，它可以用来制作与网络无关的，与用户交互作用的复杂软件。它是一种基于对象（Object-Based）和事件驱动（Event-Driven）的编程语言，因而它本身提供了非常丰富的内部对象供设计人员使用。

**解释和编译**：Java的源代码在执行之前，必须经过编译。JavaScript是一种解释性编程语言，其源代码不需经过编译，由浏览器解释执行。（目前的浏览器几乎都使用了JIT（即时编译）技术来提升JavaScript的运行效率）

**强类型变量和类型弱变量**：Java采用强类型变量检查，即所有变量在编译之前必须作声明；JavaScript中变量是弱类型的，甚至在使用变量前可以不作声明，JavaScript的解释器在运行时检查推断其数据类型。

代码格式不一样。

补充：上面列出的四点是网上流传的所谓的标准答案。其实Java和JavaScript最重要的区别是一个是静态语言，一个是动态语言。目前的编程语言的发展趋势是函数式语言和动态语言。在Java中类（class）是一等公民，而JavaScript中函数（function）是一等公民，因此JavaScript支持函数式编程，可以使用Lambda函数和闭包（closure），当然Java 8也开始支持函数式编程，提供了对Lambda表达式以及函数式接口的支持。对于这类问题，在面试的时候最好还是用自己的语言回答会更加靠谱，不要背网上所谓的标准答案。

### 74、什么时候用断言（assert）

断言在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。一般来说，断言用于保证程序最基本、关键的正确性。断言检查通常在开发和测试时开启。为了保证程序的执行效率，在软件发布后断言检查通常是关闭的。

断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；如果表达式的值为false，那么系统会报告一个AssertionError。断言的使用如下面的代码所示：

**assert**(a > 0); // throws an AssertionError if a <= 0

断言可以有两种形式：

**assert** Expression1;

**assert** Expression1 : Expression2 ;

Expression1 应该总是产生一个布尔值。

Expression2 可以是得出一个值的任意表达式；这个值用于生成显示更多调试信息的字符串消息。

要在运行时启用断言，可以在启动JVM时使用-enableassertions或者-ea标记。要在运行时选择禁用断言，可以在启动JVM时使用-da或者-disableassertions标记。要在系统类中启用或禁用断言，可使用-esa或-dsa标记。还可以在包的基础上启用或者禁用断言。

注意：断言不应该以任何方式改变程序的状态。简单的说，如果希望在不满足某些条件时阻止代码的执行，就可以考虑用断言来阻止它。

### 75、Error和Exception有什么区别？

Error表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；Exception表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

**面试题：**2005年摩托罗拉的面试中曾经问过这么一个问题“If a process reports a stack overflow run-time error, what’s the most possible cause?”，给了四个选项

a. lack of memory;

b. write on an invalid memory space;

c. recursive function calling;

d. array index out of boundary.

Java程序在运行时也可能会遭遇StackOverflowError，这是一个无法恢复的错误，只能重新修改代码了，这个面试题的答案是c。如果写了不能迅速收敛的递归，则很有可能引发栈溢出的错误，如下所示：

**class** StackOverflowErrorTest {

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

*main*(**null**);

}

}

**提示**：用递归编写程序时一定要牢记两点：1. 递归公式；2. 收敛条件（什么时候就不再继续递归）。

### 76、try{}里有一个return语句，那么紧跟在这个try后的finally{}里的代码会不会被执行，什么时候被执行，在return前还是后?

答：会执行，在方法返回调用者前执行。

注意：在finally中改变返回值的做法是不好的，因为如果存在finally代码块，try中的return语句不会立马返回调用者，而是记录下返回值待finally代码块执行完毕之后再向调用者返回其值，然后如果在finally中修改了返回值，就会返回修改后的值。显然，在finally中返回或者修改返回值会对程序造成很大的困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java中也可以通过提升编译器的语法检查级别来产生警告或错误，Eclipse中可以设置，强烈建议将此项设置为编译错误。

### 77、Java语言如何进行异常处理，关键字：throws、throw、try、catch、finally分别如何使用？

Java通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供了良好的接口。在Java中，每个异常都是一个对象，它是Throwable类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。Java的异常处理是通过5个关键词来实现的：try、catch、throw、throws和finally。一般情况下是用try来执行一段程序，如果系统会抛出（throw）一个异常对象，可以通过它的类型来捕获（catch）它，或通过总是执行代码块（finally）来处理；try用来指定一块预防所有异常的程序；catch子句紧跟在try块后面，用来指定你想要捕获的异常的类型； throw语句用来明确地抛出一个异常；throws用来声明一个方法可能抛出的各种异常（当然声明异常时允许无病呻吟）；finally为确保一段代码不管发生什么异常状况都要被执行；try语句可以嵌套，每当遇到一个try语句，异常的结构就会被放入异常栈中，直到所有的try语句都完成。如果下一级的try语句没有对某种异常进行处理，异常栈就会执行出栈操作,直到遇到有处理这种异常的try语句或者最终将异常抛给JVM。

### 78、运行时异常与受检异常有何异同？

异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误，只要程序设计得没有问题通常就不会发生。受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发。Java编译器要求方法必须声明抛出可能发生的受检异常，但是并不要求必须声明抛出未被捕获的运行时异常。异常和继承一样，是面向对象程序设计中经常被滥用的东西，在Effective Java中对异常的使用给出了以下指导原则：

不要将异常处理用于正常的控制流（设计良好的API不应该强迫它的调用者为了正常的控制流而使用异常）

对可以恢复的情况使用受检异常，对编程错误使用运行时异常

避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）

优先使用标准的异常

每个方法抛出的异常都要有文档

保持异常的原子性

不要在catch中忽略掉捕获到的异常

### 79、列出一些你常见的运行时异常？

ArithmeticException（算术异常）

ClassCastException （类转换异常）

IllegalArgumentException （非法参数异常）

IndexOutOfBoundsException （下标越界异常）

NullPointerException （空指针异常）

SecurityException （安全异常）

### 80、阐述final、finally、finalize的区别。

**final**：修饰符（关键字）有三种用法：如果一个类被声明为final，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。将变量声明为final，可以保证它们在使用中不被改变，被声明为final的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。被声明为final的方法也同样只能使用，不能在子类中被重写。

**finally**：通常放在try…catch…的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。

**finalize**：Object类中定义的方法，Java中允许使用finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize()方法可以整理系统资源或者执行其他清理工作。

### 81、类ExampleA继承Exception，类ExampleB继承ExampleA。

有如下代码片断, 请问执行此段代码的输出是什么？

**try** {

**throw** **new** ExampleB("b")

} **catch**（ExampleA e）{

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

} **catch**（Exception e）{

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

}

答：输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类型]，抓取ExampleA类型异常的catch块能够抓住try块中抛出的ExampleB类型的异常）

面试题 - 说出下面代码的运行结果。（此题的出处是《Java编程思想》一书）

**class** Annoyance **extends** Exception {}

**class** Sneeze **extends** Annoyance {}

**class** Human {

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

**try** {

**try** {

**throw** **new** Sneeze();

} **catch** ( Annoyance a ) {

System.***out***.println("Caught Annoyance");

**throw** a;

}

} **catch** ( Sneeze s ) {

System.***out***.println("Caught Sneeze");

**return** ;

}**finally** {

System.***out***.println("Hello World!");

}

}

}

### 82、List、Set、Map是否继承自Collection接口？

List、Set 是，Map 不是。Map是键值对映射容器，与List和Set有明显的区别，而Set存储的零散的元素且不允许有重复元素（数学中的集合也是如此），List是线性结构的容器，适用于按数值索引访问元素的情形。

### 83、阐述ArrayList、Vector、LinkedList的存储性能和特性。

答：ArrayList 和Vector都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector中的方法由于添加了synchronized修饰，因此Vector是线程安全的容器，但性能上较ArrayList差，因此已经是Java中的遗留容器。LinkedList使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，内存的利用率更高），按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

Vector属于遗留容器（Java早期的版本中提供的容器，除此之外，Hashtable、Dictionary、BitSet、Stack、Properties都是遗留容器），已经不推荐使用，但是由于ArrayList和LinkedListed都是非线程安全的，如果遇到多个线程操作同一个容器的场景，则可以通过工具类Collections中的synchronizedList方法将其转换成线程安全的容器后再使用（这是对装潢模式的应用，将已有对象传入另一个类的构造器中创建新的对象来增强实现）。

补充：遗留容器中的Properties类和Stack类在设计上有严重的问题，Properties是一个键和值都是字符串的特殊的键值对映射，在设计上应该是关联一个Hashtable并将其两个泛型参数设置为String类型，但是Java API中的Properties直接继承了Hashtable，这很明显是对继承的滥用。这里复用代码的方式应该是Has-A关系而不是Is-A关系，另一方面容器都属于工具类，继承工具类本身就是一个错误的做法，使用工具类最好的方式是Has-A关系（关联）或Use-A关系（依赖）。同理，Stack类继承Vector也是不正确的。Sun公司的工程师们也会犯这种低级错误，让人唏嘘不已。

### 84、Collection和Collections的区别？

Collection是一个接口，它是Set、List等容器的父接口；Collections是个一个工具类，提供了一系列的静态方法来辅助容器操作，这些方法包括对容器的搜索、排序、线程安全化等等。

### 85、List、Map、Set三个接口存取元素时，各有什么特点？

List以特定索引来存取元素，可以有重复元素。Set不能存放重复元素（用对象的equals()方法来区分元素是否重复）。

Map保存键值对（key-value pair）映射，映射关系可以是一对一或多对一。Set和Map容器都有基于哈希存储和排序树的两种实现版本，基于哈希存储的版本理论存取时间复杂度为O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

### 86、TreeMap和TreeSet在排序时如何比较元素？Collections工具类中的sort()方法如何比较元素？

答：TreeSet要求存放的对象所属的类必须实现Comparable接口，该接口提供了比较元素的compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap要求存放的键值对映射的键必须实现Comparable接口从而根据键对元素进行排序。Collections工具类的sort方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象比较实现Comparable接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是Comparator接口的子类型（需要重写compare方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用

（Java中对函数式编程的支持）。

例子1：

**public** **class** Student **implements** Comparable<Student> {

**private** String name; // 姓名

**private** **int** age; // 年龄

**public** Student(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

@Override

**public** String toString() {

**return** "Student [name=" + name + ", age=" + age + "]";

}

@Override

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

**return** **this**.age - o.age; // 比较年龄(年龄的升序)

}

}

**import** java.util.Set;

**import** java.util.TreeSet;

**class** Test01 {

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

Set<Student> set = **new** TreeSet<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)

set.add(**new** Student("Hao LUO", 33));

set.add(**new** Student("XJ WANG", 32));

set.add(**new** Student("Bruce LEE", 60));

set.add(**new** Student("Bob YANG", 22));

**for**(Student stu : set) {

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

}

// 输出结果:

// Student [name=Bob YANG, age=22]

// Student [name=XJ WANG, age=32]

// Student [name=Hao LUO, age=33]

// Student [name=Bruce LEE, age=60]

}

}

例子2：

**public** **class** Student {

**private** String name; // 姓名

**private** **int** age; // 年龄

**public** Student(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

/\*\*

\* 获取学生姓名

\*/

**public** String getName() {

**return** name;

}

/\*\*

\* 获取学生年龄

\*/

**public** **int** getAge() {

**return** age;

}

@Override

**public** String toString() {

**return** "Student [name=" + name + ", age=" + age + "]";

}

}

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.List;

**class** Test02 {

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

List<Student> list = **new** ArrayList<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)

list.add(**new** Student("Hao LUO", 33));

list.add(**new** Student("XJ WANG", 32));

list.add(**new** Student("Bruce LEE", 60));

list.add(**new** Student("Bob YANG", 22));

// 通过sort方法的第二个参数传入一个Comparator接口对象

// 相当于是传入一个比较对象大小的算法到sort方法中

// 由于Java中没有函数指针、仿函数、委托这样的概念

// 因此要将一个算法传入一个方法中唯一的选择就是通过接口回调

Collections.*sort*(list, **new** Comparator<Student> () {

@Override

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

**return** o1.getName().compareTo(o2.getName()); // 比较学生姓名

}

});

**for**(Student stu : list) {

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

}

// 输出结果:

// Student [name=Bob YANG, age=22]

// Student [name=Bruce LEE, age=60]

// Student [name=Hao LUO, age=33]

// Student [name=XJ WANG, age=32]

}

}

### 87、Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别?

答：sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，请参考第66题中的线程状态转换图）。wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

补充：可能不少人对什么是进程，什么是线程还比较模糊，对于为什么需要多线程编程也不是特别理解。简单的说：进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位；线程是进程的一个实体，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。使用多线程的编程通常能够带来更好的性能和用户体验，但是多线程的程序对于其他程序是不友好的，因为它可能占用了更多的CPU资源。当然，也不是线程越多，程序的性能就越好，因为线程之间的调度和切换也会浪费CPU时间。时下很时髦的Node.js就采用了单线程异步I/O的工作模式。

### 88、线程的sleep()方法和yield()方法有什么区别？

① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会； yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；

③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；

④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

### 89、当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？

答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

### 90、请说出与线程同步以及线程调度相关的方法。

wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；

notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；

notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

补充：Java 5通过Lock接口提供了显式的锁机制（explicit lock），增强了灵活性以及对线程的协调。Lock接口中定义了加锁（lock()）和解锁（unlock()）的方法，同时还提供了newCondition()方法来产生用于线程之间通信的Condition对象；此外，Java 5还提供了信号量机制（semaphore），信号量可以用来限制对某个共享资源进行访问的线程的数量。在对资源进行访问之前，线程必须得到信号量的许可（调用Semaphore对象的acquire()方法）；在完成对资源的访问后，线程必须向信号量归还许可（调用Semaphore对象的release()方法）。

下面的例子演示了100个线程同时向一个银行账户中存入1元钱，在没有使用同步机制和使用同步机制情况下的执行情况。

银行账户类：

/\*\*

\* 银行账户

\*/

**public** **class** Account {

**private** **double** balance; // 账户余额

/\*\*

\* 存款

\* **@param** money 存入金额

\*/

**public** **void** deposit(**double** money) {

**double** newBalance = balance + money;

**try** {

Thread.*sleep*(10); // 模拟此业务需要一段处理时间

}

**catch**(InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

/\*\*

\* 获得账户余额

\*/

**public** **double** getBalance() {

**return** balance;

}

}

存钱线程类

/\*\*

\* 存钱线程

\*/

**public** **class** AddMoneyThread **implements** Runnable {

**private** Account account; // 存入账户

**private** **double** money; // 存入金额

**public** AddMoneyThread(Account account, **double** money) {

**this**.account = account;

**this**.money = money;

}

@Override

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

account.deposit(money);

}

}

测试类

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** Test01 {

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

Account account = **new** Account();

ExecutorService service = Executors.*newFixedThreadPool*(100);

**for**(**int** i = 1; i <= 100; i++) {

service.execute(**new** AddMoneyThread(account, 1));

}

service.shutdown();

**while**(!service.isTerminated()) {}

System.***out***.println("账户余额: " + account.getBalance());

}

}

在没有同步的情况下，执行结果通常是显示账户余额在10元以下，出现这种状况的原因是，当一个线程A试图存入1元的时候，另外一个线程B也能够进入存款的方法中，线程B读取到的账户余额仍然是线程A存入1元钱之前的账户余额，因此也是在原来的余额0上面做了加1元的操作，同理线程C也会做类似的事情，所以最后100个线程执行结束时，本来期望账户余额为100元，但实际得到的通常在10元以下（很可能是1元哦）。解决这个问题的办法就是同步，当一个线程对银行账户存钱时，需要将此账户锁定，待其操作完成后才允许其他的线程进行操作，代码有如下几种调整方案：

在银行账户的存款（deposit）方法上同步（synchronized）关键字

/\*\*

\* 银行账户

\*/

**public** **class** Account {

**private** **double** balance; // 账户余额

/\*\*

\* 存款

\* **@param** money 存入金额

\*/

**public** **synchronized** **void** deposit(**double** money) {

**double** newBalance = balance + money;

**try** {

Thread.*sleep*(10); // 模拟此业务需要一段处理时间

}

**catch**(InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

/\*\*

\* 获得账户余额

\*/

**public** **double** getBalance() {

**return** balance;

}

}

在线程调用存款方法时对银行账户进行同步

/\*\*

\* 存钱线程

\*/

**public** **class** AddMoneyThread **implements** Runnable {

**private** Account account; // 存入账户

**private** **double** money; // 存入金额

**public** AddMoneyThread(Account account, **double** money) {

**this**.account = account;

**this**.money = money;

}

@Override

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

**synchronized** (account) {

account.deposit(money);

}

}

}

通过Java 5显示的锁机制，为每个银行账户创建一个锁对象，在存款操作进行加锁和解锁的操作

**import** java.util.concurrent.locks.Lock;

**import** java.util.concurrent.locks.ReentrantLock;

/\*\*

\* 银行账户

\*/

**public** **class** Account {

**private** Lock accountLock = **new** ReentrantLock();

**private** **double** balance; // 账户余额

/\*\*

\* 存款

\* **@param** money

\* 存入金额

\*/

**public** **void** deposit(**double** money) {

accountLock.lock();

**try** {

**double** newBalance = balance + money;

**try** {

Thread.*sleep*(10); // 模拟此业务需要一段处理时间

}

**catch** (InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

**finally** {

accountLock.unlock();

}

}

/\*\*

\* 获得账户余额

\*/

**public** **double** getBalance() {

**return** balance;

}

}

按照上述三种方式对代码进行修改后，重写执行测试代码Test01，将看到最终的账户余额为100元。当然也可以使用Semaphore或CountdownLatch来实现同步。

### 91、编写多线程程序有几种实现方式？

Java 5以前实现多线程有两种实现方法：一种是继承Thread类；另一种是实现Runnable接口。两种方式都要通过重写run()方法来定义线程的行为，推荐使用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

补充：Java 5以后创建线程还有第三种方式：实现Callable接口，该接口中的call方法可以在线程执行结束时产生一个返回值，代码如下所示：

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**class** MyTask **implements** Callable<Integer> {

**private** **int** upperBounds;

**public** MyTask(**int** upperBounds) {

**this**.upperBounds = upperBounds;

}

@Override

**public** Integer call() **throws** Exception {

**int** sum = 0;

**for**(**int** i = 1; i <= upperBounds; i++) {

sum += i;

}

**return** sum;

}

}

**class** Test {

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

List<Future<Integer>> list = **new** ArrayList<>();

ExecutorService service = Executors.*newFixedThreadPool*(10);

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

list.add(service.submit(**new** MyTask((**int**) (Math.*random*() \* 100))));

}

**int** sum = 0;

**for**(Future<Integer> future : list) {

// while(!future.isDone()) ;

sum += future.get();

}

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

}

}

### 92、synchronized关键字的用法？

答：synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { … }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。

### 93、举例说明同步和异步。

如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

### 94、启动一个线程是调用run()还是start()方法？

答：启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

### 95、什么是线程池（thread pool）

答：在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

**newSingleThreadExecutor**：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

**newFixedThreadPool**：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

**newCachedThreadPool**：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

**newScheduledThreadPool**：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

**newSingleThreadExecutor**：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

如果希望在服务器上使用线程池，强烈建议使用newFixedThreadPool方法来创建线程池，这样能获得更好的性能。

### 96、线程的基本状态以及状态之间的关系？

答：说明：其中Running表示运行状态，Runnable表示就绪状态（万事俱备，只欠CPU），Blocked表示阻塞状态，阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

### 97、简述synchronized 和java.util.concurrent.locks.Lock的异同？

答：Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

### 98、Java中如何实现序列化，有什么意义？

答：序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。 要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆。

### 99、Java中有几种类型的流？

答：字节流和字符流。字节流继承于InputStream、OutputStream，字符流继承于Reader、Writer。在java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。关于Java的I/O需要注意的有两点：一是**两种对称性**（输入和输出的对称性，字节和字符的对称性）；二是**两种设计模式**（适配器模式和装潢模式）。另外Java中的流不同于C#的是它只有一个维度一个方向。

面试题 - 编程实现文件拷贝。（这个题目在笔试的时候经常出现，下面的代码给出了两种实现方案）

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.OutputStream;

**import** java.nio.ByteBuffer;

**import** java.nio.channels.FileChannel;

**public** **final** **class** MyUtil {

**private** MyUtil() {

**throw** **new** AssertionError();

}

**public** **static** **void** fileCopy(String source, String target) **throws** IOException {

**try** (InputStream in = **new** FileInputStream(source)) { //try()释放资源 java7新特性

**try** (OutputStream out = **new** FileOutputStream(target)) {

**byte**[] buffer = **new** **byte**[4096];

**int** bytesToRead;

**while**((bytesToRead = in.read(buffer)) != -1) {

out.write(buffer, 0, bytesToRead);

}

}

}

}

**public** **static** **void** fileCopyNIO(String source, String target) **throws** IOException {

**try** (FileInputStream in = **new** FileInputStream(source)) {

**try** (FileOutputStream out = **new** FileOutputStream(target)) {

FileChannel inChannel = in.getChannel();

FileChannel outChannel = out.getChannel();

ByteBuffer buffer = ByteBuffer.*allocate*(4096);

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

buffer.flip();

outChannel.write(buffer);

buffer.clear();

}

}

}

}

}

注意：上面用到Java 7的TWR，使用TWR后可以不用在finally中释放外部资源 ，从而让代码更加优雅。

### 写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数。

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**public** **final** **class** MyUtil {

// 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)

**private** MyUtil() {

**throw** **new** AssertionError(); \*\*\*\*\*\*\*\*

}

/\*\*

\* 统计给定文件中给定字符串的出现次数

\* **@param** filename 文件名

\* **@param** word 字符串

\* **@return** 字符串在文件中出现的次数

\*/

**public** **static** **int** countWordInFile(String filename, String word) {

**int** counter = 0;

**try** (FileReader fr = **new** FileReader(filename)) {

**try** (BufferedReader br = **new** BufferedReader(fr)) {

String line = **null**;

**while** ((line = br.readLine()) != **null**) {

**int** index = -1;

**while** (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {

counter++;

line = line.substring(index + word.length());

}

}

}

} **catch** (Exception ex) {

ex.printStackTrace();

}

**return** counter;

}

}

### 101、如何用Java代码列出一个目录下所有的文件？

如果只要求列出当前文件夹下的文件，代码如下所示：

**import** java.io.File;

**class** Test12 {

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

File f = **new** File("/Users/Hao/Downloads");

**for**(File temp : f.listFiles()) {

**if**(temp.isFile()) {

System.***out***.println(temp.getName());

}

}

}

}

如果需要对文件夹继续展开，代码如下所示：

**import** java.io.File;

**class** Test12 {

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

*showDirectory*(**new** File("/Users/Hao/Downloads"));

}

**public** **static** **void** showDirectory(File f) {

*\_walkDirectory*(f, 0);

}

**private** **static** **void** \_walkDirectory(File f, **int** level) {

**if**(f.isDirectory()) {

**for**(File temp : f.listFiles()) {

*\_walkDirectory*(temp, level + 1);

}

}**else** {

**for**(**int** i = 0; i < level - 1; i++) {

System.***out***.print("\t");

}

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

}

}

}

在Java 7中可以使用NIO.2的API来做同样的事情，代码如下所示：

**class** ShowFileTest {

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

Path initPath = Paths.get("/Users/Hao/Downloads");

Files.walkFileTree(initPath, **new** SimpleFileVisitor<Path>() {

@Override

**public** FileVisitResult visitFile(Path file, BasicFileAttributes attrs)

**throws** IOException {

System.***out***.println(file.getFileName().toString());

**return** FileVisitResult.CONTINUE;

}

});

}

}

### 用Java的套接字编程实现一个多线程的回显（echo）服务器。

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**public** **class** EchoServer {

**private** **static** **final** **int** ***ECHO\_SERVER\_PORT*** = 6789;

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

**try**(ServerSocket server = **new** ServerSocket(***ECHO\_SERVER\_PORT***)) {

System.***out***.println("服务器已经启动...");

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

Socket client = server.accept();

**new** Thread(**new** ClientHandler(client)).start();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

**private** **static** **class** ClientHandler **implements** Runnable {

**private** Socket client;

**public** ClientHandler(Socket client) {

**this**.client = client;

}

@Override

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

**try**(BufferedReader br = **new** BufferedReader(**new** InputStreamReader(client.getInputStream()));

PrintWriter pw = **new** PrintWriter(client.getOutputStream())) {

String msg = br.readLine();

System.***out***.println("收到" + client.getInetAddress() + "发送的: " + msg);

pw.println(msg);

pw.flush();

} **catch**(Exception ex) {

ex.printStackTrace();

} **finally** {

**try** {

client.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

}

}

注意：上面的代码使用了Java 7的TWR语法，由于很多外部资源类都间接的实现了AutoCloseable接口（单方法回调接口），因此可以利用TWR语法在try结束的时候通过回调的方式自动调用外部资源类的close()方法，避免书写冗长的finally代码块。此外，上面的代码用一个静态内部类实现线程的功能，使用多线程可以避免一个用户I/O操作所产生的中断影响其他用户对服务器的访问，简单的说就是一个用户的输入操作不会造成其他用户的阻塞。当然，上面的代码使用线程池可以获得更好的性能，因为频繁的创建和销毁线程所造成的开销也是不可忽视的。

下面是一段回显客户端测试代码：

**import** java.io.BufferedReader;

**import** java.io.InputStreamReader;

**import** java.io.PrintWriter;

**import** java.net.Socket;

**import** java.util.Scanner;

**public** **class** EchoClient {

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

Socket client = **new** Socket("localhost", 6789);

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

System.***out***.print("请输入内容: ");

String msg = sc.nextLine();

sc.close();

PrintWriter pw = **new** PrintWriter(client.getOutputStream());

pw.println(msg);

pw.flush();

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(client.getInputStream()));

System.***out***.println(br.readLine());

client.close();

}

}

如果希望用NIO的多路复用套接字实现服务器，代码如下所示。NIO的操作虽然带来了更好的性能，但是有些操作是比较底层的，对于初学者来说还是有些难于理解。

**import** java.io.IOException;

**import** java.net.InetSocketAddress;

**import** java.nio.ByteBuffer;

**import** java.nio.CharBuffer;

**import** java.nio.channels.SelectionKey;

**import** java.nio.channels.Selector;

**import** java.nio.channels.ServerSocketChannel;

**import** java.nio.channels.SocketChannel;

**import** java.util.Iterator;

**public** **class** EchoServerNIO {

**private** **static** **final** **int** ECHO\_SERVER\_PORT = 6789;

**private** **static** **final** **int** ECHO\_SERVER\_TIMEOUT = 5000;

**private** **static** **final** **int** BUFFER\_SIZE = 1024;

**private** **static** ServerSocketChannel serverChannel = **null**;

**private** **static** Selector selector = **null**; // 多路复用选择器

**private** **static** ByteBuffer buffer = **null**; // 缓冲区

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

init();

listen();

}

**private** **static** **void** init() {

**try** {

serverChannel = ServerSocketChannel.open();

buffer = ByteBuffer.allocate(BUFFER\_SIZE);

serverChannel.socket().bind(**new** InetSocketAddress(ECHO\_SERVER\_PORT));

serverChannel.configureBlocking(**false**);

selector = Selector.open();

serverChannel.register(selector, SelectionKey.OP\_ACCEPT);

} **catch** (Exception e) {

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

}

}

**private** **static** **void** listen() {

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

**try** {

**if** (selector.select(ECHO\_SERVER\_TIMEOUT) != 0) {

Iterator<SelectionKey> it = selector.selectedKeys().iterator();

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

SelectionKey key = it.next();

it.remove();

handleKey(key);

}

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

**private** **static** **void** handleKey(SelectionKey key) **throws** IOException {

SocketChannel channel = **null**;

**try** {

**if** (key.isAcceptable()) {

ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();

channel = serverChannel.accept();

channel.configureBlocking(**false**);

channel.register(selector, SelectionKey.OP\_READ);

} **else** **if** (key.isReadable()) {

channel = (SocketChannel) key.channel();

buffer.clear();

**if** (channel.read(buffer) > 0) {

buffer.flip();

CharBuffer charBuffer = CharsetHelper.decode(buffer);

String msg = charBuffer.toString();

System.out.println("收到" + channel.getRemoteAddress() + "的消息：" + msg);

channel.write(CharsetHelper.encode(CharBuffer.wrap(msg)));

} **else** {

channel.close();

}

}

} **catch** (Exception e) {

e.printStackTrace();

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

channel.close();

}

}

}

}

**import** java.nio.ByteBuffer;

**import** java.nio.CharBuffer;

**import** java.nio.charset.CharacterCodingException;

**import** java.nio.charset.Charset;

**import** java.nio.charset.CharsetDecoder;

**import** java.nio.charset.CharsetEncoder;

**public** **final** **class** CharsetHelper {

**private** **static** **final** String ***UTF\_8*** = "UTF-8";

**private** **static** CharsetEncoder *encoder* = Charset.*forName*(***UTF\_8***).newEncoder();

**private** **static** CharsetDecoder *decoder* = Charset.*forName*(***UTF\_8***).newDecoder();

**private** CharsetHelper() {

}

**public** **static** ByteBuffer encode(CharBuffer in) **throws** CharacterCodingException{

**return** *encoder*.encode(in);

}

**public** **static** CharBuffer decode(ByteBuffer in) **throws** CharacterCodingException{

**return** *decoder*.decode(in);

}

}

### 103、XML文档定义有几种形式？它们之间有何本质区别？解析XML文档有哪几种方式？

答：XML文档定义分为**DTD**和**Schema**两种形式，二者都是对XML语法的约束，其本质区别在于Schema本身也是一个XML文件，可以被XML解析器解析，而且可以为XML承载的数据定义类型，约束能力较之DTD更强大。对XML的解析主要有DOM（文档对象模型，Document Object Model）、SAX（Simple API for XML）和StAX（Java 6中引入的新的解析XML的方式， Streaming API for XML），其中DOM处理大型文件时其性能下降的非常厉害，这个问题是由DOM树结构占用的内存较多造成的，而且DOM解析方式必须在解析文件之前把整个文档装入内存，适合对XML的随机访问（典型的用空间换取时间的策略）；SAX是事件驱动型的XML解析方式，它顺序读取XML文件，不需要一次全部装载整个文件。当遇到像文件开头，文档结束，或者标签开头与标签结束时，它会触发一个事件，用户通过事件回调代码来处理XML文件，适合对XML的顺序访问；顾名思义，StAX把重点放在流上，实际上StAX与其他解析方式的本质区别就在于应用程序能够把XML作为一个事件流来处理。将XML作为一组事件来处理的想法并不新颖（SAX就是这样做的），但不同之处在于StAX允许应用程序代码把这些事件逐个拉出来，而不用提供在解析器方便时从解析器中接收事件的处理程序。

### 104、你在项目中哪些地方用到了XML？

答：XML的主要作用有两个方面：**数据交换**和**信息配置**。在做数据交换时，XML将数据用标签组装成起来，然后压缩打包加密后通过网络传送给接收者，接收解密与解压缩后再从XML文件中还原相关信息进行处理，XML曾经是异构系统间交换数据的事实标准，但此项功能几乎已经被JSON（JavaScript Object Notation）取而代之。当然，目前很多软件仍然使用XML来存储配置信息，我们在很多项目中通常也会将作为配置信息的硬代码写在XML文件中，Java的很多框架也是这么做的，而且这些框架都选择了dom4j作为处理XML的工具，因为Sun公司的官方API实在不怎么好用。

补充：现在有很多时髦的软件（如Sublime）已经开始将配置文件书写成JSON格式，我们已经强烈的感受到XML的另一项功能也将逐渐被业界抛弃。

### 105、阐述JDBC操作数据库的步骤。

下面的代码以连接本机的Oracle数据库为例，演示JDBC操作数据库的步骤。

加载驱动

Class.forName("oracle.jdbc.driver.OracleDriver");

创建连接

Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");

创建语句

PreparedStatement ps = con.prepareStatement("select \* from emp where sal between ? and ?");

ps.setInt(1, 1000);

ps.setInt(2, 3000);

执行语句

ResultSet rs = ps.executeQuery();

处理结果

**while**(rs.next()) {

System.out.println(rs.getInt("empno") + " - " + rs.getString("ename"));

}

关闭资源

**finally** {

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

**try** {

con.close();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

提示：关闭外部资源的顺序应该和打开的顺序相反，也就是说先关闭ResultSet、再关闭Statement、在关闭Connection。上面的代码只关闭了Connection（连接），虽然通常情况下在关闭连接时，连接上创建的语句和打开的游标也会关闭，但不能保证总是如此，因此应该按照刚才说的顺序分别关闭。此外，第一步加载驱动在JDBC 4.0中是可以省略的自动从类路径中加载驱动），但是我们建议保留。

### 106、Statement和PreparedStatement有什么区别？哪个性能更好？

与Statement相比，①PreparedStatement接口代表**预编译**的语句，它主要的优势在于可以减少SQL的编译错误并增加SQL的安全性（减少SQL注射攻击的可能性）；②PreparedStatement中的SQL语句是可以带参数的，避免了用字符串连接拼接SQL语句的麻烦和不安全；③当批量处理SQL或频繁执行相同的查询时，PreparedStatement有明显的性能上的优势，由于数据库可以将编译优化后的SQL语句缓存起来，下次执行相同结构的语句时就会很快（不用再次编译和生成执行计划）。

补充：为了提供对存储过程的调用，JDBC API中还提供了CallableStatement接口。存储过程（Stored Procedure）是数据库中一组为了完成特定功能的SQL语句的集合，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。虽然调用存储过程会在网络开销、安全性、性能上获得很多好处，但是存在如果底层数据库发生迁移时就会有很多麻烦，因为每种数据库的存储过程在书写上存在不少的差别。

### 107、使用JDBC操作数据库时，如何提升读取数据的性能？如何提升更新数据的性能？

要提升读取数据的性能，可以指定通过结果集（ResultSet）对象的setFetchSize()方法指定每次抓取的记录数（典型的空间换时间策略）；要提升更新数据的性能可以使用PreparedStatement语句构建批处理，将若干SQL语句置于一个批处理中执行。

### 108、在进行数据库编程时，连接池有什么作用？

由于创建连接和释放连接都有很大的开销（尤其是数据库服务器不在本地时，每次建立连接都需要进行TCP的三次握手，释放连接需要进行TCP四次握手，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销，这是典型的用空间换取时间的策略（浪费了空间存储连接，但节省了创建和释放连接的时间）。池化技术在Java开发中是很常见的，在使用线程时创建线程池的道理与此相同。基于Java的开源数据库连接池主要有：C3P0、Proxool、DBCP、BoneCP、Druid等。

补充：在计算机系统中时间和空间是不可调和的矛盾，理解这一点对设计满足性能要求的算法是至关重要的。大型网站性能优化的一个关键就是使用缓存，而缓存跟上面讲的连接池道理非常类似，也是使用空间换时间的策略。可以将热点数据置于缓存中，当用户查询这些数据时可以直接从缓存中得到，这无论如何也快过去数据库中查询。

当然，缓存的置换策略等也会对系统性能产生重要影响，对于这个问题的讨论已经超出了这里要阐述的范围。

### 109、什么是DAO模式？

DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共API中。用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。DAO模式实际上包含了两个模式，一是Data Accessor（数据访问器），二是Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。

### 110、事务的ACID是指什么？

**原子性(Atomic)**：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失败；

**一致性(Consistent)**：事务结束后系统状态是一致的；事务的一致性定义基本可以理解为是事务对数据完整性约束的遵循。这些约束可能包括主键约束、外键约束或是一些用户自定义约束。事务执行的前后都是合法的数据状态，不会违背任何的数据完整性，这就是“一致”的意思。

**隔离性(Isolated)**：并发执行的事务彼此无法看到对方的中间状态；

**持久性(Durable)**：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

补充：关于事务，在面试中被问到的概率是很高的，可以问的问题也是很多的。首先需要知道的是，只有存在并发数据访问时才需要事务。当多个事务访问同一数据时，可能会存在5类问题，包括3类数据读取问题（脏读、不可重复读和幻读）和2类数据更新问题（第1类丢失更新和第2类丢失更新）。

**脏读（Dirty Read）**：A事务读取B事务尚未提交的数据并在此基础上操作，而B事务执行回滚，那么A读取到的数据就是脏数据。

时间 转账事务A 取款事务B

T1   开始事务

T2 开始事务

T3   查询账户余额为1000元

T4   取出500元余额修改为500元

T5 查询账户余额为500元（脏读）

T6   撤销事务余额恢复为1000元

T7 汇入100元把余额修改为600元

T8 提交事务

**不可重复读（Unrepeatable Read）**：事务A重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务B修改过了。

时间 转账事务A 取款事务B

T1   开始事务

T2 开始事务

T3   查询账户余额为1000元

T4 查询账户余额为1000元

T5   取出100元修改余额为900元

T6   提交事务

T7 查询账户余额为900元（不可重复读）

**幻读（Phantom Read）**：事务A重新执行一个查询，返回一系列符合查询条件的行，发现其中插入了被事务B提交的行。

时间 统计金额事务A 转账事务B

T1   开始事务

T2 开始事务

T3 统计总存款为10000元

T4   新增一个存款账户存入100元

T5   提交事务

T6 再次统计总存款为10100元（幻读）

**第1类丢失更新**：事务A撤销时，把已经提交的事务B的更新数据覆盖了。

时间 取款事务A 转账事务B

T1 开始事务

T2   开始事务

T3 查询账户余额为1000元

T4   查询账户余额为1000元

T5   汇入100元修改余额为1100元

T6   提交事务

T7 取出100元将余额修改为900元

T8 撤销事务

T9 余额恢复为1000元（丢失更新）

**第2类丢失更新**：事务A覆盖事务B已经提交的数据，造成事务B所做的操作丢失。

时间 转账事务A 取款事务B

T1   开始事务

T2 开始事务

T3   查询账户余额为1000元

T4 查询账户余额为1000元

T5   取出100元将余额修改为900元

T6   提交事务

T7 汇入100元将余额修改为1100元

T8 提交事务

T9 查询账户余额为1100元（丢失更新）

数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为**表级锁**和**行级锁**；按并发事务锁定关系可以分为**共享锁**和**独占锁**。

直接使用锁是非常麻烦的，为此数据库为用户提供了**自动锁机制**，只要用户指定会话的事务隔离级别，数据库就会通过分析SQL语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，如下表所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重复读 | 幻读 | 第一类丢失更新 | 第二类丢失更新 |
| READ UNCOMMITED | 允许 | 允许 | 允许 | 不允许 | 允许 |
| READ COMMITTED | 不允许 | 允许 | 允许 | 不允许 | 允许 |
| REPEATABLE READ | 不允许 | 不允许 | 允许 | 不允许 | 不允许 |
| SERIALIZABLE | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

事务隔离级别和数据访问的并发性是对立的，事务隔离级别越高并发性就越差。所以要根据具体的应用来确定合适的事务隔离级别，这个地方没有万能的原则。

### 111、JDBC中如何进行事务处理？

Connection提供了事务处理的方法，通过调用setAutoCommit(false)可以设置手动提交事务；当事务完成后用commit()显式提交事务；如果在事务处理过程中发生异常则通过rollback()进行事务回滚。除此之外，从JDBC 3.0中还引入了Savepoint（保存点）的概念，允许通过代码设置保存点并让事务回滚到指定的保存点。

### 112、JDBC能否处理Blob和Clob？

Blob是指**二进制大对象**（Binary Large Object），而Clob是指**大字符对象**（Character Large Objec），因此其中Blob是为存储大的二进制数据而设计的，而Clob是为存储大的文本数据而设计的。JDBC的PreparedStatement和ResultSet都提供了相应的方法来支持Blob和Clob操作。下面的代码展示了如何使用JDBC操作LOB：

下面以MySQL数据库为例，创建一个张有三个字段的用户表，包括编号（id）、姓名（name）和照片（photo），建表语句如下：

create table tb\_user

(

id int primary key auto\_increment,

name varchar(20) unique not null,

photo longblob

);

下面的Java代码向数据库中插入一条记录：

**import** java.io.FileInputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**class** JdbcLobTest {

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

Connection con = **null**;

**try** {

// 1. 加载驱动（Java6以上版本可以省略）

Class.*forName*("com.mysql.jdbc.Driver");

// 2. 建立连接

con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test", "root", "123456");

// 3. 创建语句对象

PreparedStatement ps = con.prepareStatement("insert into tb\_user values (default, ?, ?)");

ps.setString(1, "骆昊"); // 将SQL语句中第一个占位符换成字符串

**try** (InputStream in = **new** FileInputStream("test.jpg")) { // Java 7的TWR

ps.setBinaryStream(2, in); // 将SQL语句中第二个占位符换成二进制流

// 4. 发出SQL语句获得受影响行数

System.***out***.println(ps.executeUpdate() == 1 ? "插入成功" : "插入失败");

} **catch**(IOException e) {

System.***out***.println("读取照片失败!");

}

} **catch** (ClassNotFoundException | SQLException e) { // Java 7的多异常捕获

e.printStackTrace();

} **finally** { // 释放外部资源的代码都应当放在finally中保证其能够得到执行

**try** {

**if**(con != **null** && !con.isClosed()) {

con.close(); // 5. 释放数据库连接

con = **null**; // 指示垃圾回收器可以回收该对象

}

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

}

### 113、简述正则表达式及其用途。

在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。

说明：计算机诞生初期处理的信息几乎都是数值，但是时过境迁，今天我们使用计算机处理的信息更多的时候不是数值而是字符串，正则表达式就是在进行字符串匹配和处理的时候最为强大的工具，绝大多数语言都提供了对正则表达式的支持。

### 114、Java中是如何支持正则表达式操作的

Java中的String类提供了支持正则表达式操作的方法，包括：matches()、replaceAll()、replaceFirst()、split()。此外，Java中可以用**Pattern**类表示正则表达式对象，它提供了丰富的API进行各种正则表达式操作。

**面试题**： - 如果要从字符串中截取第一个英文左括号之前的字符串，例如：北京市(朝阳区)(西城区)(海淀区)，截取结果为：北京市，那么正则表达式怎么写？

**import** java.util.regex.Matcher;

**import** java.util.regex.Pattern;

**class** RegExpTest {

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

String str = "北京市(朝阳区)(西城区)(海淀区)";

Pattern p = Pattern.*compile*(".\*?(?=\\()");

Matcher m = p.matcher(str);

**if**(m.find()) {

System.***out***.println(m.group());

}

}

}

说明：上面的正则表达式中使用了懒惰匹配和前瞻，如果不清楚这些内容，推荐读一下网上很有名的《正则表达式30分钟入门教程》。

### 115、获得一个类的类对象有哪些方式？

方法1：类型.class，例如：String.class

方法2：对象.getClass()，例如："hello".getClass()

方法3：Class.forName()，例如：Class.forName("java.lang.String")

### 116、如何通过反射创建对象？

方法1：通过类对象调用newInstance()方法，例如：String.class.newInstance()

方法2：通过类对象的getConstructor()或getDeclaredConstructor()方法获得构造器（Constructor）对象并调用其newInstance()方法创建对象，例如：String.class.getConstructor(String.class).newInstance("Hello");

### 117、如何通过反射获取和设置对象私有字段的值？

可以通过类对象的**getDeclaredField()**方法字段（Field）对象，然后再通过字段对象的**setAccessible(true)**将其设置为可以访问，接下来就可以通过get/set方法来获取/设置字段的值了。下面的代码实现了一个反射的工具类，其中的两个静态方法分别用于获取和设置私有字段的值，字段可以是基本类型也可以是对象类型且支持多级对象操作，例如ReflectionUtil.get(dog, "owner.car.engine.id");可以获得dog对象的主人的汽车的引擎的ID号。

**import** java.lang.reflect.Constructor;

**import** java.lang.reflect.Field;

**import** java.lang.reflect.Modifier;

**import** java.util.ArrayList;

**import** java.util.List;

/\*\*

\* 反射工具类

\*/

**public** **class** ReflectionUtil {

**private** ReflectionUtil() {

**throw** **new** AssertionError();

}

/\*\*

\* 通过反射取对象指定字段(属性)的值

\* **@param** target 目标对象

\* **@param** fieldName 字段的名字

\* **@throws** 如果取不到对象指定字段的值则抛出异常

\* **@return** 字段的值

\*/

**public** **static** Object getValue(Object target, String fieldName) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

**try** {

**for**(**int** i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(**true**);

target = f.get(target);

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(**true**);

**return** f.get(target);

}**catch** (Exception e) {

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

}

}

/\*\*

\* 通过反射给对象的指定字段赋值

\* **@param** target 目标对象

\* **@param** fieldName 字段的名称

\* **@param** value 值

\*/

**public** **static** **void** setValue(Object target, String fieldName, Object value) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

**try** {

**for**(**int** i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(**true**);

Object val = f.get(target);

**if**(val == **null**) {

Constructor<?> c = f.getType().getDeclaredConstructor();

c.setAccessible(**true**);

val = c.newInstance();

f.set(target, val);

}

target = val;

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(**true**);

f.set(target, value);

}

**catch** (Exception e) {

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

}

}

}

### 如何通过反射调用对象的方法？

**import** java.lang.reflect.Method;

**class** MethodInvokeTest {

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

String str = "hello";

Method m = str.getClass().getMethod("toUpperCase");

System.***out***.println(m.invoke(str)); // HELLO

}

}

### 119、简述一下面向对象的"六原则一法则"。

**单一职责原则**：一个类只做它该做的事情。（单一职责原则想表达的就是"高内聚"，写代码最终极的原则只有六个字"高内聚、低耦合"，就如同葵花宝典或辟邪剑谱的中心思想就八个字"欲练此功必先自宫"，所谓的高内聚就是一个代码模块只完成一项功能，在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。我们都知道一句话叫"因为专注，所以专业"，一个对象果承担太多的职责，那么注定它什么都做不好。这个世界上任何好的东西都有两个特征，一个是功能单一，好的相机绝对不是电视购物里面卖的那种一个机器有一百多种功能的，它基本上只能照相；另一个是模块化，好的自行车是组装车，从减震叉、刹车到变速器，所有的部件都是可以拆卸和重新组装的，好的乒乓球拍也不是成品拍，一定是底板和胶皮可以拆分和自行组装的，一个好的软件系统，它里面的每个功能模块也应该是可以轻易的拿到其他系统中使用的，这样才能实现软件复用的目标。）

**开闭原则**：软件实体应当对扩展开放，对修改关闭。（在理想的状态下，当我们需要为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。要做到开闭有*两个要点*：

①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；

②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱，如果不清楚如何封装可变性，可以参考《设计模式精解》一书中对桥梁模式的讲解的章节。）

**依赖倒转原则**：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代，请参考下面的里氏替换原则。）

**里氏替换原则**：任何时候都可以用子类型替换掉父类型。（关于里氏替换原则的描述，Barbara Liskov女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。）

**接口隔离原则**：接口要小而专，绝不能大而全。（臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。Java中的接口代表能力、代表约定、代表角色，能否正确的使用接口一定是编程水平高低的重要标识。）

**合成聚合复用原则**：优先使用聚合或合成关系复用代码。（通过继承来复用代码是面向对象程序设计中被滥用得最多的东西，因为所有的教科书都无一例外的对继承进行了鼓吹从而误导了初学者，类与类之间简单的说有三种关系，Is-A关系、Has-A关系、Use-A关系，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是Has-A关系，合成聚合复用原则想表达的是优先考虑Has-A关系而不是Is-A关系复用代码，原因嘛可以自己从百度上找到一万个理由，需要说明的是，即使在Java的API中也有不少滥用继承的例子，例如Properties类继承了Hashtable类，Stack类继承了Vector类，这些继承明显就是错误的，更好的做法是在Properties类中放置一个Hashtable类型的成员并且将其键和值都设置为字符串来存储数据，而Stack类的设计也应该是在Stack类中放一个Vector对象来存储数据。记住：任何时候都不要继承工具类，工具是可以拥有并可以使用的，而不是拿来继承的。）

**迪米特法则**：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（迪米特法则简单的说就是如何做到"低耦合"，门面模式和调停者模式就是对迪米特法则的践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web开发中作为前端控制器的Servlet或Filter不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度，如下图所示。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）

### 120、简述一下你了解的设计模式。

答：所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。

在GoF的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类（创建型[对类的实例化过程的抽象化]、结构型[描述如何将类或对象结合在一起形成更大的结构]、行为型[对在不同的对象之间划分责任和算法的抽象化]）共23种设计模式，包括：Abstract Factory（抽象工厂模式）

Builder（建造者模式）

Factory Method（工厂方法模式）

Prototype（原始模型模式）

Singleton（单例模式）

Adapter（适配器模式）

Decorator（装饰模式）

Proxy（代理模式）

Facade（门面模式，也叫外观模式）

Bridge（桥梁模式）

Composite（合成模式）

Flyweight（享元模式）

Strategy（策略模式）

Template Method（模板方法模式）

Observer（观察者模式）

Iterator（迭代子模式）

Chain Of Responsibility（责任链模式）

Command（命令模式）

Memento（备忘录模式）

State（状态模式）

Visitor（访问者模式）

Mediator（调停者模式，也叫中介者模式）

Interpreter（解释器模式）

面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：

**工厂模式**：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。

**代理模式**：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache代理、防火墙代理、同步化代理、智能引用代理。

**适配器模式**：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。

**模板方法模式**：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。

除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections工具类和I/O系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

### 121、用Java写一个单例类。

饿汉式单例

**public** **class** Singleton {

**private** Singleton(){}

**private** **static** Singleton *instance* = **new** Singleton();

**public** **static** Singleton getInstance(){

**return** *instance*;

}

}

懒汉式单例

**public** **class** Singleton {

**private** **static** Singleton *instance* = **null**;

**private** Singleton() {}

**public** **static** **synchronized** Singleton getInstance(){

**if** (instance == **null**) instance ＝ **new** Singleton();

**return** instance;

}

}

注意：实现一个单例有两点注意事项，①将构造器私有，不允许外界通过构造器创建对象；②通过公开的静态方法向外界返回类的唯一实例。这里有一个问题可以思考：Spring的IoC容器可以为普通的类创建单例，它是怎么做到的呢？

### 122、什么是UML？

UML是统一建模语言（Unified Modeling Language）的缩写，它发表于1997年，综合了当时已经存在的面向对象的建模语言、方法和过程，是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持。使用UML可以帮助沟通与交流，辅助应用设计和文档的生成，还能够阐释系统的结构和行为。

### 123、UML中有哪些常用的图？

UML定义了多种图形化的符号来描述软件系统部分或全部的静态结构和动态结构，包括：用例图（use case diagram）、类图（class diagram）、时序图（sequence diagram）、协作图（collaboration diagram）、状态图（statechart diagram）、活动图（activity diagram）、构件图（component diagram）、部署图（deployment diagram）等。在这些图形化符号中，有三种图最为重要，分别是：用例图（用来捕获需求，描述系统的功能，通过该图可以迅速的了解系统的功能模块及其关系）、类图（描述类以及类与类之间的关系，通过该图可以快速了解系统）、时序图（描述执行特定任务时对象之间的交互关系以及执行顺序，通过该图可以了解对象能接收的消息也就是说对象能够向外界提供的服务）。

### 124、用Java写一个冒泡排序。

冒泡排序几乎是个程序员都写得出来，但是面试的时候如何写一个逼格高的冒泡排序却不是每个人都能做到，下面提供一个参考代码：

**import** java.util.Comparator;

/\*\*

\* 排序器接口(策略模式: 将算法封装到具有共同接口的独立的类中使得它们可以相互替换)

\*/

**public** **interface** Sorter {

/\*\*

\* 排序

\* **@param** list 待排序的数组

\*/

**public** <T **extends** Comparable<T>> **void** sort(T[] list);

/\*\*

\* 排序

\* **@param** list 待排序的数组

\* **@param** comp 比较两个对象的比较器

\*/

**public** <T> **void** sort(T[] list, Comparator<T> comp);

}

**import** java.util.Comparator;

/\*\*

\* 冒泡排序

\*/

**public** **class** BubbleSorter **implements** Sorter {

@Override

**public** <T **extends** Comparable<T>> **void** sort(T[] list) {

**boolean** swapped = **true**;

**for** (**int** i = 1, len = list.length; i < len && swapped; ++i) {

swapped = **false**;

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

**if** (list[j].compareTo(list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = **true**;

}

}

}

}

@Override

**public** <T> **void** sort(T[] list, Comparator<T> comp) {

**boolean** swapped = **true**;

**for** (**int** i = 1, len = list.length; i < len && swapped; ++i) {

swapped = **false**;

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

**if** (comp.compare(list[j], list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = **true**;

}

}

}

}

}

### 125、用Java写一个折半查找。

折半查找，也称二分查找、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组已经为空，则表示找不到指定的元素。这种搜索算法每一次比较都使搜索范围缩小一半，其时间复杂度是O(logN)。

**import** java.util.Comparator;

**public** **class** MyUtil {

**public** **static** <T **extends** Comparable<T>> **int** binarySearch(T[] x, T key) {

**return** *binarySearch*(x, 0, x.length- 1, key);

}

// 使用循环实现的二分查找

**public** **static** <T> **int** binarySearch(T[] x, T key, Comparator<T> comp) {

**int** low = 0;

**int** high = x.length - 1;

**while** (low <= high) {

**int** mid = (low + high) >>> 1;

**int** cmp = comp.compare(x[mid], key);

**if** (cmp < 0) {

low= mid + 1;

}**else** **if** (cmp > 0) {

high= mid - 1;

}**else** {

**return** mid;

}

}

**return** -1;

}

// 使用递归实现的二分查找

**private** **static**<T **extends** Comparable<T>> **int** binarySearch(T[] x, **int** low, **int** high, T key) {

**if**(low <= high) {

**int** mid = low + ((high -low) >> 1);

**if**(key.compareTo(x[mid])== 0) {

**return** mid;

}**else** **if**(key.compareTo(x[mid])< 0) {

**return** *binarySearch*(x,low, mid - 1, key);

}**else** {

**return** *binarySearch*(x,mid + 1, high, key);

}

}

**return** -1;

}

}

### 基本类型的默认值

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 类型 | byte | short | Int | long | float | double | char | boolean |
| 默认值 | 0 | 0 | 0 | 0L | 0.0f | 0.0d | '\u0000' | false |

# 二、JVM知识

## 1、什么情况下会发生栈内存溢出。

如果线程请求分配的栈容量超过java虚拟机栈允许的最大容量的时候，java虚拟机将抛出一个StackOverFlowError异常。

如果java虚拟机栈可以动态拓展，并且扩展的动作已经尝试过，但是目前无法申请到足够的内存去完成拓展，或者在建立新线程的时候没有足够的内存去创建对应的虚拟机栈，那java虚拟机将会抛出一个OutOfMemoryError异常。

## 2、JVM的内存结构，Eden和Survivor比例。

#### JVM的堆内存

新生代：(1/3):新对象，小

Eden:(4/5)

Survivor:GC后幸存的对象

From Survivor:（1/10）

To Survivor:（1/10）

老生代：（2/3）老对象，大

#### 对象的一生

1、对象A被new出来之后，是被存放在Eden区的。注释：Eden即伊甸园，亚当和夏娃的故事。

2、当发生一次GC之后，Eden区存活下来的对象A会被复制到Survivor 1区（此时Survivor 1为To Survivor）；Survivor 0 （此时为From Survivor）中存活的对象也会被复制到Survivor 1中。

3、GC会清空Eden和Survivor 0 （即From Survivor）中存储的所有对象。因为Eden和Survivor 0 中存活的对象都被复制到 Survivor 1中了，所以清空是没问题的。

4、 交换Survivor 0和Survivor 1的角色：即此时有数据的Survivor 1作为From Survivor，被清空的Survivor 0作为To Survivor。要保证在GC发生之前，To Survivor永远是空的那个。

5、下次GC发生时，重复上述步骤。将Eden中存活的对象复制到To Survivor，将From Survivor中活的对象也复制到To Survivor。

在上述步骤3中，发生GC时，From Survivor中存活的对象并不是全部都会被复制到To Survivor中，而是根据这个对象在Survivor区中存活了多久而决定去向，当一个对象在Survivor中存活了很久（即经历了多次GC还没死），就会在发生GC时被复制到旧生代中。

我是一个普通的java对象，我出生在Eden区，在Eden区我还看到和我长的很像的小兄弟，我们在Eden区中玩了挺长时间。有一天Eden区中的人实在是太多了，我就被迫去了Survivor区的“To”区，自从去了Survivor区，我就开始漂泊了，因为Survivor的两个区总是交换名字，所以我总是搬家，搬到To Survivor居住，搬来搬去，居无定所。直到我18岁的时候，爸爸说我成人了，该去社会上闯闯了。于是我就去了年老代那边，年老代里，人很多，并且年龄都挺大的，我在这里也认识了很多人。在年老代里，我生活了20年(每次GC加一岁)，然后被回收。

## 3、JVM内存为什么要分成新生代，老年代，持久代。新生代中为什么要分为Eden和Survivor。

如果没有Survivor，Eden区每进行一次Minor GC，存活的对象就会被送到老年代。老年代很快被填满，触发Major GC（因为Major GC一般伴随着Minor GC，也可以看做触发了Full GC）。老年代的内存空间远大于新生代，进行一次Full GC消耗的时间比Minor GC长得多。你也许会问，执行时间长有什么坏处？频发的Full GC消耗的时间是非常可观的，这一点会影响大型程序的执行和响应速度，更不要说某些连接会因为超时发生连接错误了。

只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。

设置两个Survivor区最大的好处就是**解决了碎片化**，整个过程中，永远有一个survivor space是空的，另一个非空的survivor space无碎片

## 4、JVM中一次完整的GC流程是怎样的，对象如何晋升到老年代，说说你知道的几种主要的JVM参数。

对象晋升老生代一共有三个可能：

1.当对象达到成年，经历过15次GC（默认15次，可配置），对象就晋升为老生代

2.大的对象会直接在老生代创建

3.新生代跟幸存区内存不足时，对象可能晋升到老生代

-Xms：初始堆大小

-Xmx：堆最大内存

-Xss：栈内存

-XX:PermSize 初始永久带内存

-XX:MaxPermSize 最大永久带内存

## 5、你知道哪几种垃圾收集器，各自的优缺点，重点讲下cms和G1，包括原理，流程，优缺点。

**串行垃圾收集器**：收集时间长，停顿时间久

**并发垃圾收集器**：碎片空间多

**CMS**:并发标记清除。他的主要步骤有：初始收集，并发标记，重新标记，并发清除（删除），重置

**G1**：主要步骤：初始标记，并发标记，重新标记，复制清除（整理）

CMS的缺点是对cpu的要求比较高。G1是将内存化成了多块，所有对内段的大小有很大的要求。CMS是清除，所以会存在很多的内存碎片。G1是整理，所以碎片空间较小

## 6、垃圾回收算法的实现原理。

## 7、当出现了内存溢出，你怎么排错。

内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。整个JVM内存大小=**年轻代大小** + **年老代大小** + **持久代大小**,目前来说，常遇到的泄漏问题如下：

**年老代堆空间被占满**：

异常： java.lang.OutOfMemoryError: Java heap space

这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。这种情况一般来说是因为内存泄漏或者内存不足造成的。某些情况因为长期的无法释放对象，运行时间长了以后导致对象数量增多，从而导致的内存泄漏。另外一种就是因为系统的原因，大并发加上大对象，Survivor Space区域内存不够，大量的对象进入到了老年代,然而老年代的内存也不足时，从而产生了Full GC，但是这个时候Full GC也无发回收。这个时候就会产生java.lang.OutOfMemoryError: Java heap space

代码内的内存泄漏可以通过一些分析工具进行分析，然后找出泄漏点进行改善。第二种原因导致的OutOfMemoryError可以通过，优化代码和增加Survivor Space等方式去优化。

**持久代被占满**

异常：java.lang.OutOfMemoryError: PermGen space

Perm空间被占满。无法为新的class分配存储空间而引发的异常。这个异常以前是没有的，但是在Java反射大量使用的今天这个异常比较常见了。主要原因就是大量动态反射生成的类不断被加载，最终导致Perm区被占满。

解决方案：

增加持久代的空间 -XX:MaxPermSize=100M。

如果有自定义类加载的需要排查下自己的代码问题。

**堆栈溢出**

异常：java.lang.StackOverflowError

一般就是递归没返回，或者循环调用造成

**线程堆栈满**

异常：Fatal: Stack size too small

java中一个线程的空间大小是有限制的。JDK5.0以后这个值是1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。 解决：增加线程栈大小。**-Xss2m**。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

**系统内存被占满**

异常：java.lang.OutOfMemoryError: unable to create new native thread

这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。 分配给Java虚拟机的内存愈多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。同时，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总共内生产的线程数。

解决：

1. 重新设计系统减少线程数量。

2. 线程数量不能减少的情况下，通过-**Xss**减小单个线程大小。以便能生产更多的线程。

## 8、JVM内存模型的相关知识了解多少，比如重排序，内存屏障，happen-before，主内存，工作内存等。

内存屏障：为了保障执行顺序和可见性的一条cpu指令

重排序：为了提高性能，编译器和处理器会对执行进行重拍

happen-before：操作间执行的顺序关系。有些操作先发生。

主内存：共享变量存储的区域即是主内存

工作内存：每个线程copy的本地内存，存储了该线程以读/写共享变量的副本

## 9、简单说说你了解的类加载器，可以打破双亲委派么，怎么打破。

JVM将类加载过程分为三个步骤：装载（Load），链接（Link）和初始化(Initialize)

1、**装载**：查找并加载类的二进制数据；

2、**链接**：

验证：确保被加载类的正确性；

准备：为类的静态变量分配内存，并将其初始化为默认值；

解析：把类中的符号引用转换为直接引用；

1. **初始化**：为类的静态变量赋予正确的初始值；

那为什么我要有验证这一步骤呢？首先如果由编译器生成的class文件，它肯定是符合JVM字节码格式的，但是万一有高手自己写一个class文件，让JVM加载并运行，用于恶意用途，就不妙了，因此这个class文件要先过验证这一关，不符合的话不会让它继续执行的，也是为了安全考虑吧。

准备阶段和初始化阶段看似有点矛盾，其实是不矛盾的，如果类中有语句：**private** **static** **int** a = 10，它的执行过程是这样的，首先字节码文件被加载到内存后，先进行链接的验证这一步骤，验证通过后准备阶段，给a分配内存，因为变量a是static的，所以此时a等于int类型的默认初始值0，即a=0,然后到解析（后面在说），到初始化这一步骤时，才把a的真正的值10赋给a,此时a=10。

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时**数据区的方法区**内，然后在堆区创建一个这个类的java.lang.Class对象，用来封装类在方法区类的对象

加载类的方式有以下几种：

1）从本地系统直接加载

2）通过网络下载.class文件

3）从zip，jar等归档文件中加载.class文件

4）从专有数据库中提取.class文件

5）将Java源文件动态编译为.class文件（服务器）

**启动类加载器： Bootstrap ClassLoader**

负责加载$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类

**其他类加载器： Extension ClassLoader**

负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包

**App ClassLoader**：

负责加载classpath中指定的jar包及目录中class

**Custom ClassLoader：**

属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

加载过程中会先检查类是否被已加载，检查顺序是**自底向上**，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

**双亲委派模型**：如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。所以**检查顺序为从下到上，加载顺序为从上到下**。

打破双亲委派机制则不仅要继承ClassLoader类，还要重写loadClass和findClass方法

**public** **class** TestClassLoaderN **extends** ClassLoader {

**private** String name;

**public** TestClassLoaderN(ClassLoader parent, String name) {

**super**(parent);

**this**.name = name;

}

@Override

**public** String toString() {

**return** **this**.name;

}

@Override

**public** Class<?> loadClass(String name) **throws** ClassNotFoundException {

Class<?> clazz = **null**;

ClassLoader system = *getSystemClassLoader*();

**try** {

clazz = system.loadClass(name);

} **catch** (Exception e) {

// ignore

}

**if** (clazz != **null**)

**return** clazz;

clazz = findClass(name);

**return** clazz;

}

@Override

**public** Class<?> findClass(String name) {

InputStream is = **null**;

**byte**[] data = **null**;

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

**try** {

is = **new** FileInputStream(**new** File("d:/Test.class"));

**int** c = 0;

**while** (-1 != (c = is.read())) {

baos.write(c);

}

data = baos.toByteArray();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**try** {

is.close();

baos.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

**return** **this**.defineClass(name, data, 0, data.length);

}

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

TestClassLoaderN loader = **new** TestClassLoaderN(

TestClassLoaderN.**class**.getClassLoader(), "TestLoaderN");

Class clazz;

**try** {

clazz = loader.loadClass("test.classloader.Test");

Object object = clazz.newInstance();

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

## 10.讲讲JAVA的反射机制。

Java程序在运行状态可以动态的获取类的所有属性和方法，并实例化该类，调用方法的功能

在类加载的时候，jvm会创建一个class对象，class对象是可以说是反射中最常用的，获取class对象的方式的主要有三种

根据类名：类名.class

根据对象：对象.getClass()

根据全限定类名：Class.forName(全限定类名)

### 构造函数：

@Test

**public** **void** constructorTest() **throws** Exception {

Constructor<?>[] constructors = userClass.getConstructors(); // 获取全部的构造函数

// 取消安全性检查,设置后才可以使用private修饰的构造函数，也可以单独对某个构造函数进行设置

// Constructor.setAccessible(constructors, true);

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

Class<?> parameterTypesClass[] = constructors[i].getParameterTypes();

System.***out***.print("第" + i + "个构造函数:\t (");

**for** (**int** j = 0; j < parameterTypesClass.length; j++) {

System.***out***.print(parameterTypesClass[j].getName() + (j == parameterTypesClass.length - 1 ? "" : "\t"));

}

logger.info(")");

}

// 调用构造函数，实例化对象

logger.info("实例化，调用无参构造:\t" + constructors[0].newInstance());

logger.info("实例化，调用有参构造:\t" + constructors[1].newInstance("韦德", 35));

}

输出：

第0个构造函数: ()

第1个构造函数: (java.lang.String int)

实例化，调用无参构造: User [name=init, age=0]

实例化，调用有参构造: User [name=韦德, age=35]

### 属性：

**public** **void** fieldTest() **throws** Exception {

User user = userClass.newInstance();

Field fields[] = userClass.getDeclaredFields();// 获取当前类所有属性

// 获取公有属性(包括父类)

// Field fields[] = cl.getFields();

// 取消安全性检查,设置后才可以获取或者修改private修饰的属性，也可以单独对某个属性进行设置

Field.setAccessible(fields, **true**);

**for** (Field field : fields) {

// 获取属性名 属性值 属性类型

logger.info("属性名:" + field.getName() + "\t属性值:" + field.get(user) + " \t属性类型:" + field.getType());

}

Field fieldUserName = userClass.getDeclaredField("name");

// 取消安全性检查,设置后才可以获取或者修改private修饰的属性，也可以批量对所有属性进行设置

fieldUserName.setAccessible(**true**);

fieldUserName.set(user, "韦德");// 可以直接对 private 的属性赋值

logger.info("修改属性后对象:\t" + user);

}

### 方法：

**public** **void** methodTest() **throws** Exception {

User user = userClass.newInstance();

Method[] methods = userClass.getDeclaredMethods(); // 获取当前类的所有方法

// Method[] methods = userClass.getMethods(); // 获取公有方法(包括父类)

Method.setAccessible(methods, **true**);// 取消安全性检查,设置后才可以调用private修饰的方法，也可以单独对某个方法进行设置

**for** (Method method : methods) {// 获取方法名和返回类型 获取参数类型：getParameterTypes

logger.info("方法名:" + method.getName() + " \t返回类型:" + method.getReturnType().getName());

}

Method getMethod = userClass.getDeclaredMethod("getName");// 获取无参方法

getMethod.setAccessible(**true**);// 取消安全性检查,设置后才可以调用private修饰的方法，也可以批量对所有方法进行设置

logger.info("调用getName方法：" + getMethod.invoke(user));// 调用无参方法

Method setMethod = userClass.getDeclaredMethod("setName", String.**class**);// 获取有参方法

setMethod.setAccessible(**true**);// 取消安全性检查,设置后才可以调用private修饰的方法，也可以批量对所有方法进行设置

logger.info("调用setName方法：" + setMethod.invoke(user, "韦德"));// 调用有参方法

logger.info("通过set方法修改属性后对象:\t" + user);

}

## 10、你们线上应用的JVM参数有哪些。

-server

-Xms4096M //堆容量初始值

-Xmx4096M //堆容量最大值

-Xmn1024M //新生代容量，所以老年代容量 = 堆容量 - 新生代容量 = 3072M (1:4)

-Xss256K //线程堆栈空间大小

-XX:MaxDirectMemorySize //Direct Buffer Memory大小

-Djava.awt.headless=true //使用缺少外设的系统配置模式

-Dfile.encoding=UTF-8 //设置编码规范

-XX:PermSize=500M

-XX:MaxPermSize=500M

-XX:SurvivorRatio=65536

-XX:MaxTenuringThreshold=0

-Xnoclassgc

-XX:+DisableExplicitGC

-XX:+UseParNewGC

-XX:+UseConcMarkSweepGC

-XX:+UseCMSCompactAtFullCollection

-XX:CMSFullGCsBeforeCompaction=0

-XX:+CMSClassUnloadingEnabled

-XX:-CMSParallelRemarkEnabled

-XX:CMSInitiatingOccupancyFraction=90

-XX:SoftRefLRUPolicyMSPerMB=0

-XX:+PrintClassHistogram

-XX:+PrintGCDetails //打印gc日志详细信息

-XX:+PrintGCTimeStamps //打印gc发生时相对jvm启动的时间戳，后来加入了PrintGCDateStamps，打印gc发生的日期

-XX:+PrintHeapAtGC

-Xloggc:log/gc.log //设置gc日志的生成位置

jmx配置用于远程管理

-XX:+HeapDumpOutOfMemoryError //当出现OOM时，打印堆快照

-XX:HeapDumpPath //堆快照打印路径，建议文件后缀设为hprof，可被MAT识别

-XX:+DisableExplicitGC //关闭System.gc()

-XX:SurvivorRatio=1 //Eden区与Survivor区的大小比值

-XX:+UserConcMarkSweepGC //使用CMS收集器

-XX:+UserParNewGC //新生代使用ParNew收集器

-XX:+CMSParallelRemarkEnabled //降低标记停顿

-XX+UseCMSCompactAtFullCollection //在full gc的时候，对年老代的压缩

-XX:CMSFullGCsBeforeCompaction=0 //full gc后不压缩老年代内存空间

-XX:LargePageSizeInBytes //内存页的大小

-XX:+UseFastAccessorMethods //原始类型的快速优化

-XX:+UseCMSInitiatingOccupancyOnly //使用手动定义初始化定义开始CMS收集，禁止hostspot自行触发CMS GC

-XX:CMSInitiatingOccupancyFraction=80 //老年代使用80％后开始CMS收集

-XX:SoftRefLRUPolicyMSPerMB=0 //每兆堆空闲空间中SoftReference的存活时间为0秒

## 11、g1和cms区别,吞吐量优先和响应优先的垃圾收集器选择。

Cms是以获取最短回收停顿时间为目标的收集器。基于标记-清除算法实现。比较占用cpu资源，且易造成碎片。

G1是面向服务端的垃圾收集器，是jdk9默认的收集器，基于标记-整理算法实现。可利用多核、多cpu，保留分代，实现可预测停顿，可控。

## 12、怎么打出线程栈信息。

第一步：在终端运行Java程序

第二步：通过命令 pidof java 找到已经启动的java进程的ID，选择需要查看的java程序的进程ID

第三步：使用命令 kill -3 <java进行的 pid> 打印出java程序的线程堆栈信息

第四步：通常情况下运行的项目可能会比较大，那么这个时候打印的堆栈信息可能会有几千到几万行，为了方便查看，我们往往需要将输出内容进行重定向

使用linux下的重定向命令方式即可：例如： demo.sh > run.log 2>&1 将输出信息重定向到 run.log中。

注：在操作系统中，0 1 2分别对应着不同的含义， 如下：

0 ： 标准输入，即：C中的stdin ， java中的System.in

1 ： 标准输出， 即：C中的stdout ，java中的System.out

2 ： 错误输出， 即：C中的stderr ， java中的System.err

## 13、请解释如下jvm参数的含义：

-server Server模式启动

-Xms512m 最小堆内存512m

-Xmx512m 最大512m

-Xss1024K 每个线程栈空间1m

-XX:PermSize=256m 永久代256

-XX:MaxPermSize=512m -最大永久代256

XX:MaxTenuringThreshold=20XX: -最大转为老年代检查次数20

CMSInitiatingOccupancyFraction=80 Cms回收开启时机：内存占用80%

XX:+UseCMSInitiatingOccupancyOnly。命令JVM不基于运行时收集的数据来启动CMS垃圾收集周期

# 三、开源框架知识

## 一、Spring框架

### 1、Spring概述

#### 1.Spring的核心是控制反转

通过配置文件来完成业务对象之间的依赖注入，鼓励注入对接口编程而不是对类编程

#### 2.提供了事务处理功能

#### 3.Spring有7个模块组成

##### (1)核心容器

提供了Spring框架的核心功能。BeanFactory是核心容器的主要组件。它通过控制反转将应用程序的配置和依赖性规范和实际的应用程序代码分开。

##### (2)Spring Context

通过配置文件向Spring框架提供上下文信息，它构建在BeanFactory之上，另外增加了国际化，资源访问等功能。

##### (3)Spring AOP

Spring的依赖注入为AOP支持

##### (4)Spring DAO

提供了一个简单有效的JDBC应用

##### (5)Spring ORM

Spring 除了有自己的JDBC应用之外，还提供了对其他框架的支持，例如JDO，Hibernate和iBatis（Mybatis前身）

##### (6)Spring Web

提供了简化的处理多部分请求机将请求参数绑定到域对象的任务。

##### (7)Spring MVC

Spring 提供了MVC2模式的实现

### 2、开始Spring之旅

#### 1.搭建Spring环境

（1）下载安装JDK，设置环境变量

（2）配置eclipse

（3）下载tomcat

（4）设定TOMCAT\_HOME

（5）下载eclipse的Tomcat插件

（6）配重块Tomcat插件

（7）下载Spring

Log4j是Apache的一个开源项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件 ，甚至是套接口服务器、NT的事件记录器、UNIX Syslog守护进程等；我们也可以控制每一条日志的输出格式；通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程。最令人感兴趣的就是，这些可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

### 3、Spring基础概念

#### 1.反向控制/依赖注入

**IOC：Inversion of Control 反向控制**

控制反转是一种将组件依赖关系的创建和管理置于程序外部的技术。由容器控制程序之间的关系，而不是由代码直接控制，由于控制权由代码转向了容器，所以称为反转

**DI：Dependency Injection 依赖注入**

#### 2.依赖注入的3种实现方式

##### （1）接口注入

##### set注入

<bean id=*"userService"* class=*"com.lyu.spring.service.impl.UserService"*>

<property name=*"userDao"* ref=*"userDaoMyBatis"*></property>

</bean>

**public** **class** UserService **implements** IUserService {

**private** IUserDao userDao1;

**public** **void** setUserDao(IUserDao userDao1) {

**this**.userDao1 = userDao1;

}

}

##### （3）构造器注入

<bean id=*"userService"* class=*"com.lyu.spring.service.impl.UserService"*>

<constructor-arg name=*"userDao"* ref=*"userDaoJdbc"*></constructor-arg>

<constructor-arg name=*"user"* ref=*"user"*></constructor-arg>

</bean>

**public** **class** UserService **implements** IUserService {

**private** IUserDao userDao;

**private** User user;

**public** UserService(IUserDao userDao, User user) {

**this**.userDao = userDao;

**this**.user = user;

}

}

### 4、Spring的核心容器

#### 1.什么是Bean

#### 2.Bean的基础知识

Bean的表示（id和name）

Bean的类（class）

Singleton：

singleton="true"默认模式，只有一个实例存在

singleton="false"每次创建新的实例

depends-on=""初始化一个Bean之前强制初始化另一个Bean

#### 3.Bean的生命周期

定义:不是所有Java对象都称之为Bean，只有被Spring容器管理的Java对象称之为Bean。

配置文件

初始化：

（1）配置文档汇总通过制定init-method属性来完成

在.java中增加一个init()方法，在配置文档中指定要初始化 的方法

（2）实现InitializingBean接口

.java实现InitializingBean接口，增加一个afterPropertiesSet（）

使用：

(1)使用BeanWrapper

HelloWold helloWorld =**new** HelloWorld();

BeanWrapper bw=**new** BeanWrapperImpl(helloWorld);

bw.setPropertyValue("msg","HelloWorld");

System.out.println(bw.getPropertyValue("msg"));

(2)使用BeanFactory

InpputStream is =**new** FileInputStream("config.xml");

XmlBeanFactory factory =**new** XmlBeanFactory(is);

Helloworld helloWorld=(HelloWold)factory.getBean("HelloWold");

System.out.println(helloWorld.getMsg());

(3)使用ApplicationContext

ApplicationContext atx=**new** FielSystemXmlApplicationContext("HelloWold");

HelloWold helloWorld =**new** (HelloWold) atx.getBean("HelloWold")

System.out.println(HelloWold.getMsg());

销毁

(1)在配置文档通过指定destory-menthod属性来完成

(2)实现disposableBean接口

#### 4.用 ref的属性指定依赖的3种模式

（1）用local的属性指定

（2）用Bean的属性指定

（3）用parent属性指定

#### 5.Bean制动装配的5种模式

（1）使用byName模式

（2）使用byType模式

（3）使用constructor模式

（4）使用autodetect模式

（5）使用no模式

#### 6.依赖检查

（1）使用simple模式

（2）使用object模式

（3）使用all模式

（4）使用none模式

#### 7.集合的注入方式

（1）List

（2）Set

（3）Map

（4）Properties

### 5、Spring的AOP

**AOP**：Aspect Oriented Programming 面向方面编程

**OOP**：面向对象编程

#### 1、java代理机制

在java的动态代理机制中，有两个重要的类或接口，一个是 InvocationHandler(Interface)、另一个则是 Proxy(Class)，这一个类和接口是实现我们动态代理所必须用到的。

每一个动态代理类都必须要实现InvocationHandler这个接口，并且每个代理类的实例都关联到了一个handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由InvocationHandler这个接口的 invoke 方法来进行调用。

##### invoke 方法

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

**proxy**:　 指代我们所代理的那个真实对象

**method:**　指代的是我们所要调用真实对象的某个方法的Method对象

**args**: 指代的是调用真实对象某个方法时接受的参数

##### Proxy

这个类的作用就是用来动态创建一个代理对象的类，它提供了许多的方法，但是我们用的最多的就是 newProxyInstance 这个方法：

**public** **static** Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) **throws** IllegalArgumentException

**loader**:一个ClassLoader对象，定义了由哪个ClassLoader对象来对生成的代理对象进行加载。

**interfaces**:　一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了。

**h**:一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上。

##### 代理简介

代理模式的适用，总结为：代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等

##### 代理模式的用途

###### (1)远程代理(Remote Proxy)

为远程对象提供一个本地的代理对象， 典型的例子如RMI， EJB，local bean 为remote 接口对象提供一个stub

###### (2)虚拟代理(Virtual Proxy)

允许内存开销较大的对象在需要的时候创建。只有我们真正需要这个对象的时候才创建。

虚拟代理模式(VirtualProxy)是一种节省内存的技术，它建议创建那些占用大量内存或处理复杂的对象时，把创建这类对象推迟到使用它的时候。在特定的应用中，不同部分的功能由不同的对象组成，应用启动的时候，不会立即使用所有的对象。在这种情况下，虚拟代理模式建议推迟对象的创建直到应用程序需要它为止。对象被应用第一次引用时创建并且同一个实例可以被重用。这种方法优缺点并存。

**优点：**

种方法的优点是，在应用程序启动时，由于不需要创建和装载所有的对象，因此加速了应用程序的启动。

**缺点:**

因为不能保证特定的应用程序对象被创建，在访问这个对象的任何地方，都需要检测确认它不是空(null)。也就是，这种检测的时间消耗是最大的缺点。

应用虚拟代理模式，**需要设计一个与真实对象具有相同接口的单独对象**（指虚拟代理）。不同的客户对象可以在创建和使用真实对象地方用相应的虚拟对象来代替。虚拟对象把真实对象的引用作为它的实例变量维护。代理对象不要自动创建真实对象，当客户需要真实对象的服务时，调用虚拟代理对象上的方法，并且检测真实对象是否被创建。如果真实对象已经创建，代理把调用转发给真实对象，如果真实对象没有被创建：

1）代理对象创建真实对象

2）代理对象把这个对象分配给引用变量。

3）代理把调用转发给真实对象

按照这种安排，验证对象存在和转发方法调用这些细节对于客户是不可见的。客户对象就像和真实对象一样与代理对象进行交互。因此客户从检测真实对象是否为null中解脱出来，另外，由于创建代理对象在时间和处理复杂度上要少于创建真实对象。因此，在应用程序启动的时候，用代理对象代替真实对象初始化。

###### (3)写入时复制代理(Copy-On-WriteProxy)

用来控制对象的复制，方法是延迟对象的复制，直到客户真的需要为止。是虚拟代理的一个变体。

###### (4)保护代理(Protection (Access)Proxy)

为不同的客户提供不同级别的目标对象访问权限

###### (5)缓存代理(Cache Proxy)

为开销大的运算结果提供暂时存储，它允许多个客户共享结果，以减少计算或网络延迟。

###### (6)防火墙代理(Firewall Proxy)

控制网络资源的访问，保护主题免于恶意客户的侵害。

###### (7)同步代理(SynchronizationProxy)

在多线程的情况下为主题提供安全的访问。

###### (8)智能引用代理(Smart ReferenceProxy)

当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等。

###### (9)复杂隐藏代理

用来隐藏一个类的复杂集合的复杂度，并进行访问控制。有时候也称为外观代理(Fa?ade Proxy)，这不难理解。复杂隐藏代理和外观模式是不一样的，因为代理控制访问，而外观模式是不一样的，因为代理控制访问，而外观模式只提供另一组接口。

##### 静态代理与动态代理

**静态代理：**由程序员或者自动生成工具生成代理类，然后进行代理类的编译和运行。在代理类、委托类运行之前，代理类已经以.class的格式存在。

**动态代理：**在程序运行时，由反射机制动态创建而成

##### 静态代理实例

**public** **class** CountProxy **implements** Count {

**private** CountImpl countImpl;

**public** CountProxy(CountImpl countImpl) {

**this**.countImpl = countImpl;

}

@Override

**public** **void** queryCount() {

System.***out***.println("事务处理之前");

countImpl.queryCount();// 调用委托类的方法;

System.***out***.println("事务处理之后");

}

@Override

**public** **void** updateCount() {

System.***out***.println("事务处理之前");

countImpl.updateCount();// 调用委托类的方法;

System.***out***.println("事务处理之后");

}

}

##### 动态代理实现

**public** **class** DynamicSubject **implements** InvocationHandler {

**private** Object sub;

**public** DynamicSubject() {}

**public** DynamicSubject(Object obj) {

sub = obj;

}

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

System.***out***.println( " before calling " + method);

method.invoke(sub,args);

System.***out***.println( " after calling " + method);

**return** **null** ;

}

}

##### jdk代理和CJLIB代理

代理的目的是为了统一为代码进行功能增强（例如spring aop切面编程）

java动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP。

2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP。

3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换。

4、jdk代理性能优于cglib代理。

如何强制使用CGLIB实现AOP

(1)添加CGLIB库，SPRING\_HOME/cglib/\*.jar

(2)在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class=*"true"*/>

JDK动态代理和CGLIB字节码生成的区别

（1）JDK动态代理只能对实现了接口的类生成代理，而不能针对类

（2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法

因为是继承，所以该类或方法最好不要声明成final

**public** **interface** UserManager { //接口

**public** **void** addUser(String id, String password);

**public** **void** delUser(String id);

}

**public** **class** UserManagerImpl **implements** UserManager {

**public** **void** addUser(String id, String password) {

System.***out***.println(".: 调用了UserManagerImpl.addUser()方法！ ");

}

**public** **void** delUser(String id) {

System.***out***.println(".: 调用了UserManagerImpl.delUser()方法！ ");

}

}

/\*\*

\* JDK动态代理类

\*/

**public** **class** JDKProxy **implements** InvocationHandler {

**private** Object targetObject;//需要代理的目标对象

**public** Object newProxy(Object targetObject) {//将目标对象传入进行代理

**this**.targetObject = targetObject;

**return** Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),

targetObject.getClass().getInterfaces(), **this**);//返回代理对象 ，必须实现接口

}

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

**throws** Throwable {

checkPopedom();//一般我们进行逻辑处理的函数比如这个地方是模拟检查权限

Object ret = **null**; // 设置方法的返回值

ret = method.invoke(targetObject, args); //调用invoke方法，ret存储该方法的返回值

**return** ret;

}

**private** **void** checkPopedom() {//模拟检查权限的例子

System.***out***.println(".:检查权限 checkPopedom()!");

}

}

/\*\*

\* CGLibProxy动态代理类的实例

\*/

**public** **class** CGLibProxy **implements** MethodInterceptor {

**private** Object targetObject;// CGLib需要代理的目标对象

**public** Object createProxyObject(Object obj) {

**this**.targetObject = obj;

Enhancer enhancer = **new** Enhancer();

enhancer.setSuperclass(obj.getClass());

enhancer.setCallback(**this**);

Object proxyObj = enhancer.create();

**return** proxyObj;// 返回代理对象

}

**public** Object intercept(Object proxy, Method method, Object[] args,

MethodProxy methodProxy) **throws** Throwable {

Object obj = **null**;

**if** ("addUser".equals(method.getName())) {// 过滤方法

checkPopedom();// 检查权限

}

obj = method.invoke(targetObject, args);

**return** obj;

}

**private** **void** checkPopedom() {

System.***out***.println(".:检查权限 checkPopedom()!");

}

}

**public** **class** Client {

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

UserManager userManager = (UserManager) **new** CGLibProxy()

.createProxyObject(**new** UserManagerImpl());

System.***out***.println("-----------CGLibProxy-------------");

userManager.addUser("tom", "root");

System.***out***.println("-----------JDKProxy-------------");

JDKProxy jdkPrpxy = **new** JDKProxy();

UserManager userManagerJDK = (UserManager) jdkPrpxy

.newProxy(**new** UserManagerImpl());

userManagerJDK.addUser("tom", "root");

}

}

JDK代理是不需要以来第三方的库，只要要JDK环境就可以进行代理，它有几个要求

1、 实现InvocationHandler?

2、 使用Proxy.newProxyInstance产生代理对象

3、 被代理的对象必须要实现接口

CGLib 必须依赖于CGLib的类库，但是它需要类来实现任何接口代理的是指定的类生成一个子类，覆盖其中的方法，是一种继承但是针对接口编程的环境下推荐使用JDK的代理

在Hibernate中的拦截器其实现考虑到不需要其他接口的条件Hibernate中的相关代理采用的是CGLib来执行。

#### 2、AOP的三个关键概念

**切入点（Pointcut）**：是连接点的集合，是程序中需要注入Advice的位置的集合。

接口ClassFilter：用来将切入点限定在给定的目标类中

接口MethodMatcher：用来判断切入点是否匹配目标类给定的方法

**通知（Advice）**

Advice是某个连接点所采用的处理逻辑，也就是向连接点处注入的代码

**Advisor**：是Pointcut和Advice的配置器，它包含Pointcut、和Advice，是将Advice注入程序中Pointcut位置的代码

#### 3、Spring的3种切入点的实现

##### 静态切入点

只限定于给定的方法和目标类，不考虑方法的参数，只在第一次调用切入点时计算位置，然后缓存起来。

<bean id=*"settersAndAbsquatulatePointcut"*

class=*"org.springframework.aop.support.RegexMethodPointcut"*>

<proterty name=*"patterns"*>

<!--设定切入点-->

<List>

<value>.\*save.\*</value>

<value>\*.do.\*</value>

</List>

</property>

</bean>

\*save.\*：表示所有以save开头的方法都是切入点

\*.do.\*：表示所有以do开头的方法都是切入点

##### 动态切入点

动态切入点与静态切入点的区别，它不仅限定于给点的方法和类，还可以指定方法的参数，动态切入点不能缓存，每次调用时都进行计算。

##### 自定义切入点

#### 4、Spring的通知（Advice）

a、Interception Around通知

最基本的通知，在切入点前后执行，实现MethodInterceptor接口

b、Before通知

只在切入点之前执行，实现MethodBeforeAdvice接口

c、After Returning通知

只在切入点之后执行，实现AfterReturningAdvice接口

d、Throw通知

只在切入点抛出异常时执行，实现ThrowsAdvice接口

#### 5、Spring的Advisor

org.springframework.aop.support.DefaultPointcutAdvisor是最通用的类。在Spring中主要通过XML的方式配置Pointcut和Advice。

#### 6、用ProxyFactoryBean创建AOP代理

a、使用ProxyFactoryBean代理目标类的所有方法

b、使用ProxyFactoryBean代理目标类的指定方法

#### 7、Spring中AOP的两种代理方式

a、java动态代理

b、CGLIB代理

#### 8、Spring中的自动代理

### 6、Spring的事物

#### 1、简述事物处理

（1）事务处理的基本概念。

（2）事务处理的特性：原子性，一致性，隔离性，持久性

#### 2、事务处理的三种方式

（1）关系型数据库事务处理

（2）传统的JDBC事务处理

（3）分布式事务处理

#### 3、Spring的事务处理

##### (1)Spring事务处理概述

a.Spring事务中心接口

**interface** PlatformtransactionManager

TransactionStatus getTransaction()//目前的事务

**void** commit() //提交事务

**void** rollback() //事务回滚

b.TransactionDefinition 事务处理时的一些属性定义

**int** getPropagationBehavior() //获取事务的传播行为

**int** getlsolationLevel() //获取事务的隔离层次

**int** getTimeOut() //判断事务是否超时

**boolean** isReadOnly() //判断是否为只读事务

String getName() //返回一个事务的名字

TransactionStatus代表了目前的事务

**boolean** isNewTranaction() //判断是否为一个新事务

**void** setRollbackOnly() //设定为只读事务

**boolean** isCompleted(); //判断一个事务是否完成

##### (2)编程式事务处理

##### (3)声明式事务处理

### 7、Spring的持久层封装

#### 1.传统的JDBC数据访问技术

**try**{

//获取数据连接

Class.forName(com.micsoft.jdbc.sqlserver.SQLServerDriver);

conn = DriverManager.getConnection(jdbc:micsoft:sqlserver://location:1443/stdb,admin,admin);

coon.setAutoCommit(**false**); //开始启动事务

stmt =coon.createStatement();

//执行相应操作

stmt.executeUpdate("insert into hello values (1,1,1)"):

//执行成功则提交事务

coon.commit();

}**catch**(SQLException e ){

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

**try**{

coon.rollback();//执行不成功，则回滚

}**catch**{

}

}

}

}

#### 2.通过XML实现DataSource（数据源）注入

(1)使用Spring自带的DriverManagerDataSource

(2)使用DBCP连接池

(3)使用tomcat提供的JNDI

#### 3.使用JdbcTemplate访问数据

### 8、Spring的Web框架

#### 1.Web框架介绍

#### 2.SpringMVC实例

(1).编写web.xml

主要作用装载DispatcherServlet类，读取Spring配置文件

1. .编写输出的JSP页面index.jsp

(3).编写控制器HelloWorldAction.java

实现Controller接口

(4).配置Spring文档dispatcherServlet-servlet.xml

### 9、Spring的定时器

定时器的两种实现方式

Timer

Quartz

### 10、Spring与Struts

（1）下载Struts

（2）配置Struts

### 11、Spring与Hibernate的整合

1.下载Hibernate和Middlegen-Hibernate和hibernate-extensions并解压缩。

2.下载相应数据库和数据库驱动程序

### 12、Spring中使用Ant

#### 1、什么是ant

ant是构建工具

#### 2、什么是构建

概念到处可查到，形象来说，你要把代码从某个地方拿来，编译，再拷贝到某个地方去等等操作，当然不仅与此，但是主要用来干这个。

#### 3、ant的好处

跨平台 --因为ant是使用java实现的，所以它跨平台

使用简单--与ant的兄弟make比起来

语法清晰--同样是和make相比

功能强大--ant能做的事情很多，可能你用了很久，你仍然不知道它能有多少功能。当你自己开发一些ant插件的时候，你会发现它更多的功能。

#### 4、ant的兄弟make

ant做的很多事情，大部分是曾经有一个叫make的所做的，不过对象不同，**make更多应用于c/c++** ,**ant更多应用于Java**。当然这不是一定的，但大部分人如此。

#### 5、构建ant环境

要使用ant首先要构建一个ant环境，步骤很简单：

1),安装jdk，设置JAVA\_HOME ,PATH ,CLASS\_PATH(这些应该是看这篇文章的人应该知道的)

2),下载ant 地址http://www.apache.org/找一个你喜欢的版本，或者干脆最新的版本

3),解压ant 你得到的是一个压缩包，解压缩它，并把它放在一个尽量简单的目录，例如D:/ant-1.6虽然你不一 定要这么做，但这么做是有好处的。

4),设置ANT\_HOME PATH中添加ANT\_HOME目录下的bin目录

5),测试一下你的设置,开始-->运行-->cmd进入命令行-->键入 ant 回车,如果看到

Buildfile: build.xml does not exist!

Build failed

那么恭喜你你已经完成ant的设置

#### 6、体验ant

##### 首先你要知道你要干什么，

就像每个语言都有HelloWorld一样，一个最简单的应用能让人感受一下Ant，我现在想做的事情是：

编写一些程序

编译它们

把它打包成jar包

把他们放在应该放置的地方

运行它们

这里为了简单起见只写一个程序，就是HelloWorld.java程序代码如下：

**package** test.ant;

**public** **class** HelloWorld{

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

System.out.println("Hello world1");

}

}

##### 2、运用ant

为了达到上边的目的，你可以手动的用javac 、copy 、jar、java来完成，但是考虑一下如果你有成百上千个类，在多次调试，部署的时候，一次次的javac 、copy、jar、java那将是一份辛苦的工作。现在看看ant怎么优雅的完成它们。要运行ant需要有一个build.xml虽然不一定要叫这个名字，但是建议你这么做,下边就是一个完整的build.xml，然后我们来详细的解释每一句

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<project name=*"HelloWorld"* default=*"run"* basedir=*"."*>

<property name=*"src"* value=*"src"*/>

<property name=*"dest"* value=*"classes"*/>

<property name=*"hello\_jar"* value=*"hello1.jar"*/>

<target name=*"init"*>

<mkdir dir=*"${dest}"*/>

</target>

<target name=*"compile"* depends=*"init"*>

<javac srcdir=*"${src}"* destdir=*"${dest}"*/>

</target>

<target name=*"build"* depends=*"compile"*>

<jar jarfile=*"${hello\_jar}"* basedir=*"${dest}"*/>

</target>

<target name=*"run"* depends=*"build"*>

<java classname=*"test.ant.HelloWorld"* classpath=*"${hello\_jar}"*/>

</target>

<target name=*"clean"*>

<delete dir=*"${dest}"* />

<delete file=*"${hello\_jar}"* />

</target>

<target name=*"rerun"* depends=*"clean,run"*>

<ant target=*"clean"* />

<ant target=*"run"* />

</target>

</project>

###### 解析build.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

build.xml中的第一句话，没有实际的意义

<project name=*"HelloWorld"* default=*"run"* basedir=*"."*> </project>

ant的所有内容必须包含在这个里边，name是你给它取的名字，basedir故名思意就是工作的根目录 .代表当前目录。default代表默认要做的事情。

<property name=*"src"* value=*"src"*/>

类似程序中的变量，为什么这么做，想一下变量的作用

<target name=*"compile"* depends=*"init"*>

<javac srcdir=*"${src}"* destdir=*"${dest}"*/>

</target>

把你想做的每一件事情写成一个target ，它有一个名字，depends是它所依赖的target，在执行这个target例如这里的compile之前，ant会先检查init是否曾经被执行过，如果执行过则直接直接执行compile，如果没有则会先执行它依赖的target例如这里的init，然后在执行这个target 。

编译：

<target name=*"compile"* depends=*"init"*>

<javac srcdir=*"${src}"* destdir=*"${dest}"*/>

</target>

做jar包:

<target name=*"build"* depends=*"compile"*>

<jar jarfile=*"${hello\_jar}"* basedir=*"${dest}"*/>

</target>

运行：

<target name=*"run"* depends=*"build"*>

<java classname=*"test.ant.HelloWorld"* classpath=*"${hello\_jar}"*/>

</target>

为了不用拷贝，我们可以在最开始定义好目标文件夹，这样ant直接把结果就放在目标文件夹中了。

新建文件夹:

<target name=*"init"*>

<mkdir dir=*"${dest}"*/>

</target>

为了更多一点的功能体现，又加入了两个target ，删除生成的文件

<target name=*"clean"*>

<delete dir=*"${dest}"* />

<delete file=*"${hello\_jar}"* />

</target>

再次运行，这里显示了如何在一个target里边调用其他的target

<target name=*"rerun"* depends=*"clean,run"*>

<ant target=*"clean"* />

<ant target=*"run"* />

</target>

###### 检验一下你的ant

新建一个src的文件夹，然后把HelloWorld.java按照包目录放进去

做好build.xml文件 ，在命令行下键入ant ,你会发现一个个任务都完成了。每次更改完代码只需要再次键入ant，有的时候我们可能并不想运行程序，只想执行这些步骤中的某一两个步骤，例如我只想重新部署而不想运行，键入 ant build ant中的每一个任务都可以这样调用**ant + target name** 好了，这样一个简单的ant任务完成了。

#### 7、时候使用ant

也许你听到别人说起ant，一时冲动准备学习一下ant，当你看完了上边的第一个实例，也许你感觉ant真好，也许你感觉ant不过如此，得出这些结论都不能说错，虽然ant很好用，但并不是在任何情况下都是最好的选择，例如windows上有更多更简单，更容易使用的工具，比如eclipse+myeclipse eclipse+wtp等等，无论是编译，部署，运行使用起来比ant更容易，方便但有些情况则是ant发挥的好地方：

1、服务器上部署的时候

当你的程序开发完成，部署人员要部署在服务器上的时候，总不能因为因为安装一个程序就配置一个eclipse+myeclipse吧， ant在这个时候是个很好的选择，因为它小巧，容易配置，你带着你写好的build.xml到任何一台服务器上，只需要做简单的修改（一些设定，例如目录），然后一两个命令完成，这难道不是一件美好的事情吗。

2、linux上，很多时候是这样的，程序开发是在windows下，但是程序要在linux或者unix上运行，在linux或者在unix(特别是unix上)部署是个麻烦的事情，这个时候ant的特点又出来了，因为ant是跨平台的，你在build.xml可以在大多数操作系统上使用，基本不需要修改。

3、当服务器维护者不懂编程的时候

很多人都有过这样的经历，使用你们程序的人，并不懂得写程序。你得程序因为版本更新，因为修正bug需要一次又一次得重新部署。

这个时候你会发现教一个人是如此得困难。但是有ant后，你只需要告诉他，输入ant xxx等一两个命令，一切ok.

在实际的工作过程中可能会出现以下一些情况，一个项目分成很多个模块，每个小组或者部门负责一个模块，为了测试，他们自己写了一个build.xml,而你负责把这些模块组合到一起使用，写一个build.xml，这个时候你有两种选择：

1,自己重新写一个build.xml ，这将是一个麻烦的事情

2,尽量利用他们已经写好的build.xml，减少自己的工作

举个例子：

假设你下边有三个小组，每个小组负责一个部分，他们分别有一个src 和一个写好的build.xml这个时候你拿到他们的src，你需要做的是建立三个文件夹src1 ,src2, src3分别把他们的src和build.xml放进去，然后写一个build.xml

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<project name=*"main"* default=*"build"* basedir=*"."*>

<property name=*"bin"* value=*"${basedir}/bin"* />

<property name=*"src1"* value=*"${basedir}/src1"* />

<property name=*"src2"* value=*"${basedir}/src2"* />

<property name=*"src3"* value=*"${basedir}/src3"* />

<target name=*"init"*>

<mkdir dir=*"${bin}"* />

</target>

<target name=*"run"*>

<ant dir=*"${src1}"* target=*"run"* />

<ant dir=*"${src2}"* target=*"run"* />

<ant dir=*"${src3}"* target=*"run"* />

</target>

<target name=*"clean"*>

<ant dir=*"${src1}"* target=*"clean"* />

<ant dir=*"${src2}"* target=*"clean"* />

<ant dir=*"${src3}"* target=*"clean"* />

</target>

<target name=*"build"* depends=*"init,call"*>

<copy todir=*"${bin}"*>

<fileset dir=*"${src1}"*>

<include name=*"\*.jar"* />

</fileset>

<fileset dir=*"${src2}"*>

<include name=*"\*.jar"* />

</fileset>

<fileset dir=*"${src3}"*>

<include name=*"\*.jar"* />

</fileset>

</copy>

</target>

<target name=*"rebuild"* depends=*"build,clean"*>

<ant target=*"clean"* />

<ant target=*"build"* />

</target>

</project>

ok你的任务完成了。

ok,上边你完成了任务，但是你是否有些感触呢，在那些build.xml中，大多数是重复的，而且更改一次目录需要更改不少东西。是否能让工作做的更好一点呢，答案是肯定的，引入两个东西：

1,propery

2,xml include

这两个东西都有一个功能，就是能把build.xml中<propery />中的内容分离出来，共同使用，除此之外它们各有特点：

propery的特点是维护简单，只需要简单的键值对，因为并不是所有人都喜欢xml的格式；

xml include的特点是不单可以提取出属性来，连target也可以。

还是以前的例子：

例如我们想把src1 src2 src3这三个属性从xml中提出来，可以新建一个文件叫all.properties，里边的内容：

src1=D://study//ant//src1

src2=D://study//ant//src2

src3=D://study//ant//src3

然后你的build.xml文件可以这样写，别人只需要更改配置文件，而不许要更改你的build.xml文件了

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<project name=*"main"* default=*"build"* basedir=*"."*>

<property file=*"all.properties"* />

<property name=*"bin"* value=*"${basedir}/bin"* />

<target name=*"init"*>

<mkdir dir=*"${bin}"* />

</target>

<target name=*"run"*>

<ant dir=*"${src1}"* target=*"run"* />

<ant dir=*"${src2}"* target=*"run"* />

<ant dir=*"${src3}"* target=*"run"* />

</target>

<target name=*"clean"*>

<ant dir=*"${src1}"* target=*"clean"* />

<ant dir=*"${src2}"* target=*"clean"* />

<ant dir=*"${src3}"* target=*"clean"* />

</target>

<target name=*"build"* depends=*"init,call"*>

<copy todir=*"${bin}"*>

<fileset dir=*"${src1}"*>

<include name=*"\*.jar"* />

</fileset>

<fileset dir=*"${src2}"*>

<include name=*"\*.jar"* />

</fileset>

<fileset dir=*"${src3}"*>

<include name=*"\*.jar"* />

</fileset>

</copy>

</target>

<target name=*"rebuild"* depends=*"build,clean"*>

<ant target=*"clean"* />

<ant target=*"build"* />

</target>

<target name=*"test"*>

<ant dir=*"${src1}"* target=*"test"* />

<ant dir=*"${src2}"* target=*"test"* />

<ant dir=*"${src3}"* target=*"test"* />

</target>

</project>

如果你自己看的话你会看到这样一个target

<target name=*"test"*>

<ant dir=*"${src1}"* target=*"test"* />

<ant dir=*"${src2}"* target=*"test"* />

<ant dir=*"${src3}"* target=*"test"* />

</target>

有的时候你想给每个小组的build.xml加入几个target，一种做法是每个里边写，然后在这里调用，但是有一种更好的方法。 你可以写一个include.xml文件，内容如下

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<property name=*"src"* value=*"src"*/>

<property name=*"dest"* value=*"classes"*/>

<target name=*"test"* >

<ant target=*"run"* />

</target>

然后更改你三个小组的build.xml文件,每个里边加入如下内容

<!--include a xml file ,it can be common propery ,can be also a target -->

<!DOCTYPE project [

<!ENTITY share-variable SYSTEM *"file:../include.xml"*>

]>

&share-variable;

变成如下的样子，这个时候，你只要在include.xml添加propery , 添加target，三个build.xml会同时添加这些propery和target，而且不会让三个组的build.xml变得更复杂。

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!--include a xml file ,it can be common propery ,can be also a target -->

<!DOCTYPE project [

<!ENTITY share-variable SYSTEM *"file:../include.xml"*>

]>

<project name=*"HelloWorld"* default=*"run"* basedir=*"."*>

<!--use the include -->

&share-variable;

<!--defined the property-->

<!--via include

<property name="src" value="src"/>

<property name="dest" value="classes"/>

-->

<property name=*"hello\_jar"* value=*"hello1.jar"*/>

<!--define the op-->

<target name=*"init"*>

<mkdir dir=*"${dest}"*/>

</target>

<target name=*"compile"* depends=*"init"*>

<javac srcdir=*"${src}"* destdir=*"${dest}"*/>

</target>

<target name=*"build"* depends=*"compile"*>

<jar jarfile=*"${hello\_jar}"* basedir=*"${dest}"*/>

</target>

<target name=*"run"* depends=*"build"*>

<java classname=*"test.ant.HelloWorld"* classpath=*"${hello\_jar}"*/>

</target>

<target name=*"clean"*>

<delete dir=*"${dest}"* />

<delete file=*"${hello\_jar}"* />

</target>

<target name=*"rerun"* depends=*"clean,run"*>

<ant target=*"clean"* />

<ant target=*"run"* />

</target>

</project>

掌握了上边的那些内容之后，你就知道如何去写一个好的ant，但是你会发现当你真的想去做的时候，你不能马上作出好的build.xml，因为你知道太少的ant的默认提供的命令.这个时候如果你想完成任务，并提高自己，有很多办法：

1、很多开源的程序都带有build.xml，看看它们如何写的

2、ant的document，里边详细列写了ant的各种默认命令，及其丰富

3、google，永远不要忘记它

ok,在这之后随着你写的ant build越来越多，你知道的命令就越多，ant在你的手里也就越来越强大了。这个是一个慢慢积累的过程。

ant的例子很好找，各种开源框架都会带有一个build.xml仔细看看，会有很大收获

另外一个经常会用到的，但是在开源框架的build.xml一般没有的是cvs，如果使用的是远程的cvs，可以这样使用

<xml version=*"1.0"* encoding=*"utf-8"*?>

<project>

<property name=*"cvsroot"* value=*":pserver:wang:@192.168.1.2:/cvsroot"*/>

<property name=*"basedir"* value=*"/tmp/testant/"*/>

<property name=*"cvs.password"* value=*"wang"*/>

<property name=*"cvs.passfile"* value=*"${basedir}/ant.cvspass"*/>

<target name=*"initpass"*>

<cvspass cvsroot=*"${cvsroot}"* password=*"${cvs.password}"* passfile=*"${cvs.passfile}"*/>

</target>

<target name=*"checkout"* depends=*"initpass"*>

<cvs cvsroot=*"${cvsroot}"* command=*"checkout"* cvsrsh=*"ssh"* package=*"myproject"* dest=*"${basedir}"* passfile=*"${cvs.passfile}"*/>

</target>

</project>

在eclipse里边先天支持ant，**所以你可以在eclipse里边直接写build.xml**,因为eclipse提供了提示功能，自动补充功能，它能让你事半功倍。

使用方法，只需要建立一个工程，然后建立一个叫build.xml的文件。然后就可以在里边写你的ant build了。

### 13、Spring中使用Junit

下载安装Junit

## 二、Netty

### 1.Netty简介

Netty是一个java开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。

Netty本身是用于快速构建服务端与客户端之间通信协议的框架。Netty在消息处理上使用责任链模式，用户可以轻松方便的对它进行扩展。官方也提供了大量的优秀的扩展。

Netty是一个NIO客户端服务器框架，可以快速，轻松地开发网络应用程序，如协议服务器和客户端。 它大大简化和简化了网络编程，如TCP和UDP套接字服务器 。

### Netty使用场景

1.Dubbo 协议默认使用 Netty 作为基础通信组件，用于实现各进程节点之间的内部通信。

2.淘宝的消息中间件 RocketMQ 的消息生产者和消息消费者之间，也采用 Netty 进行高性能、异步通信。

### 3.Netty下载

下载jar包或者配置pom.xml

<dependencies>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty</artifactId>

<!-- Use 'netty-all' for 4.0 or above --> <version>X.Y.Z.Q</version>

<scope>compile</scope>

</dependency>

</dependencies>

### 4.入门编程“hello”

**public** **class** Server {

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

EventLoopGroup pGroup = **new** NioEventLoopGroup();//处理服务器端接受客户端连接

EventLoopGroup cGroup = **new** NioEventLoopGroup();//进行网络通信

ServerBootstrap b = **new** ServerBootstrap();//创建辅助工具类，用于服务器通道的一系列配置

b.group(pGroup,cGroup)//绑定两个线程组

.channel(NioServerSocketChannel.**class**)//指定NIO模式

.option(ChannelOption.***SO\_BACKLOG***, 1024)//设定值tcp缓冲区大小

.option(ChannelOption.***SO\_SNDBUF***, 32\*1024)//设置发送缓冲区大小

.option(ChannelOption.***SO\_RCVBUF***, 32\*1024)//设置接受缓冲区大小

.option(ChannelOption.***SO\_KEEPALIVE***, **true**)//保持连接

.childHandler(**new** ChannelInitializer<SocketChannel>(){

@Override

**protected** **void** initChannel(SocketChannel sc) **throws** Exception {

sc.pipeline().addLast(**new** ServerHandler());//在这里配置具体数据接受方法的处理

}

});

ChannelFuture cf1 = b.bind(8765).sync();//进行绑定

cf1.channel().closeFuture().sync();//等待关闭

pGroup.shutdownGracefully();

cGroup.shutdownGracefully();

}

}

**public** **class** Client {

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

EventLoopGroup group = **new** NioEventLoopGroup();//

Bootstrap b =**new** Bootstrap();

b.group(group)

.channel(NioSocketChannel.**class**)

.handler(**new** ChannelInitializer<SocketChannel>(){

@Override

**protected** **void** initChannel(SocketChannel sc) **throws** Exception {

sc.pipeline().addLast(**new** ClientHandler());//

}

});

ChannelFuture cf1 = b.connect("127.0.0.1",8765).sync();

cf1.channel().closeFuture().sync();//等待关闭

group.shutdownGracefully();

}

}

**public** **class** ServerHandler **extends** ChannelHandlerAdapter{

@Override

**public** **void** channelRead(ChannelHandlerContext ctx,Object msg) **throws** UnsupportedEncodingException{

ByteBuf bf =(ByteBuf)msg;

**byte** [] req = **new** **byte**[bf.readableBytes()];

bf.readBytes(req);

String body =**new** String(req,"utf-8");

System.***out***.println("server："+body);

String response = "返回给客户端的响应："+body;

ctx.writeAndFlush(Unpooled.*copiedBuffer*(response.getBytes()));

}

}

**public** **class** ClientHandler **extends** ChannelHandlerAdapter{

@Override

**public** **void** channelRead(ChannelHandlerContext ctx,Object msg) **throws** UnsupportedEncodingException{

ByteBuf bf =(ByteBuf)msg;

**byte** [] req = **new** **byte**[bf.readableBytes()];

bf.readBytes(req);

String body =**new** String(req,"utf-8");

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

ReferenceCountUtil.*release*(msg);

}

}

### 5.Netty解决tcp拆包、粘包

#### 粘包、拆包

熟悉tcp编程的可能都知道，无论十服务器还是客户端，当我们读取或者发送数据的时候，都需要考虑TCP底层的粘包/拆包机制。

Tcp是一个“流”协议，所谓流就是**没有界限地传递数据**。Tcp底层并不了解上层业务数据具体含义，它会根据TCP缓冲区的实际情况进行包的划分，也就是说一个在业务上完整的可能会被tcp分成多个包进行发送，也可能把多个小宝封装成大的数据包发送出去，这就是所谓的TCP粘包、拆包问题。

#### TCP粘包、拆包产生的原因

1. 应用程序write写入的字节大小大于套接字发送缓冲区的大小。
2. 进行MSS大小的TCP分段
3. 以太网帧的payload大于MTU进行IP分片。

#### TCP粘包、拆包解决方案

1. 消息定长，例如每个报文的大小固定为200字节，如果不够，空位补空格；
2. 在包尾部增加特殊字符进行分割，例如加回车等。
3. 将消息分为消息头和消息体，在消息头中包含表示消息总长度的字段，然后进行业务逻辑的处理。

#### 自定义分隔符

.childHandler(**new** ChannelInitializer<SocketChannel>(){

@Override

**protected** **void** initChannel(SocketChannel sc) **throws** Exception {

ByteBuf buf = Unpooled.*copiedBuffer*("$\_".getBytes());//设置特定分隔符

sc.pipeline().addLast(**new** DelimiterBasedFrameDecoder(1024,buf));

sc.pipeline().addLast(**new** StringDecoder());//设置字符串形式的解码

sc.pipeline().addLast(**new** ServerHandler());//在这里配置具体数据接受方法的处理

}

});

.handler(**new** ChannelInitializer<SocketChannel>(){

@Override

**protected** **void** initChannel(SocketChannel sc) **throws** Exception {

ByteBuf buf = Unpooled.*copiedBuffer*("$\_".getBytes());//设置特定分隔符

sc.pipeline().addLast(**new** DelimiterBasedFrameDecoder(1024,buf));

sc.pipeline().addLast(**new** StringDecoder());//设置字符串形式的解码

sc.pipeline().addLast(**new** ClientHandler());//

}

});

#### 定长报文

.childHandler(**new** ChannelInitializer<SocketChannel>(){

@Override

**protected** **void** initChannel(SocketChannel sc) **throws** Exception {

sc.pipeline().addLast(**new** FixedLengthFrameDecoder(10));

sc.pipeline().addLast(**new** StringDecoder());//设置字符串形式的解码

sc.pipeline().addLast(**new** ServerHandler());//在这里配置具体数据接受方法的处 }

});

.handler(**new** ChannelInitializer<SocketChannel>(){

@Override

**protected** **void** initChannel(SocketChannel sc) **throws** Exception {

sc.pipeline().addLast(**new** FixedLengthFrameDecoder(10));

sc.pipeline().addLast(**new** StringDecoder());//设置字符串形式的解码

sc.pipeline().addLast(**new** ClientHandler());//

}

});

### 6.Netty编解码技术

#### 编解码技术

说白了就是java序列化技术，序列化目的就两个，第一进行网络传输，第二对象持久化。虽然我们可以使用java进行对象序列化，netty去传输，但是java序列化的硬伤太多，比如Java序列化不可以跨语言、序列化后码流太大、序列化性能太低等等。

#### 主流的编解码框架

JBoss的Marshalling包

Google的Protobuf

基于Protobuf的Kyro

MessagePack框架

#### JBoss Marshalling

JBoss Marshalling是一个java对象序列化包，对JDK默认的序列化框架进行了优化，但又保持跟java.io.Serializable接口的兼容，同时增加了一些可调的参数和附加特性；

类库：jboss-marshalling-1.3.0、jboss-marshalling-serial-1.3.0

下载地址：<https://www.jboss.org/jbossmarshalling/downloads>

#### JBossMarshalling结合Netty

.handler(**new** ChannelInitializer<SocketChannel>(){

@Override

**protected** **void** initChannel(SocketChannel sc) **throws** Exception {

sc.pipeline().addLast(MarshallingCodeCFactory.buildMarshallingDecoder()); sc.pipeline().addLast((MarshallingCodeCFactory.buildMarshallingEncoder());

sc.pipeline().addLast(**new** ClientHandler());//

}

});

## 三、Spring MVC

### 1.首先在web.xml中配置入口servlet

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springConfig/dispatcher-servlet.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

### DispatcherServlet

类图



## 四、Struts2

### Sturts2原理

**public** **void** doFilter(ServletRequest req, ServletResponse res, FilterChain chain) **throws** IOException, ServletException ...{

HttpServletRequest request = (HttpServletRequest) req;

HttpServletResponse response = (HttpServletResponse) res;

ServletContext servletContext = filterConfig.getServletContext();

// 在这里处理了HttpServletRequest和HttpServletResponse。

DispatcherUtils du = DispatcherUtils.getInstance();

du.prepare(request, response);//正如这个方法名字一样进行locale、encoding以及特殊request parameters设置

**try** ...{

request = du.wrapRequest(request, servletContext);//对request进行包装

} **catch** (IOException e) ...{

String message = "Could not wrap servlet request with MultipartRequestWrapper!";

LOG.error(message, e);

**throw** **new** ServletException(message, e);

}

ActionMapperIF mapper = ActionMapperFactory.getMapper();//得到action的mapper

ActionMapping mapping = mapper.getMapping(request);// 得到action 的 mapping

**if** (mapping == **null**) ...{

// there is no action in this request, should we look for a static resource?

String resourcePath = RequestUtils.getServletPath(request);

**if** ("".equals(resourcePath) && **null** != request.getPathInfo()) ...{

resourcePath = request.getPathInfo();

}

**if** ("true".equals(Configuration.get(WebWorkConstants.WEBWORK\_SERVE\_STATIC\_CONTENT))

&& resourcePath.startsWith("/webwork")) ...{

String name = resourcePath.substring("/webwork".length());

findStaticResource(name, response);

} **else** ...{

// this is a normal request, let it pass through

chain.doFilter(request, response);

}

// WW did its job here

**return**;

}

Object o = **null**;

**try** ...{

//setupContainer(request);

o = beforeActionInvocation(request, servletContext);

//整个框架最最核心的方法，下面分析

du.serviceAction(request, response, servletContext, mapping);

} **finally** ...{

afterActionInvocation(request, servletContext, o);

ActionContext.setContext(**null**);

}

}

du.serviceAction(request, response, servletContext, mapping);

//这个方法询问ActionMapper是否需要调用某个Action来处理这个（request）请求，如果ActionMapper决定需要调用某个Action，FilterDispatcher把请求的处理交给ActionProxy

}

**public** **void** serviceAction(HttpServletRequest request, HttpServletResponse response, String namespace, String actionName, Map requestMap, Map parameterMap, Map sessionMap, Map applicationMap) ...{

HashMap extraContext = createContextMap(requestMap, parameterMap, sessionMap, applicationMap, request, response, getServletConfig()); //实例化Map请求 ，询问ActionMapper是否需要调用某个Action来处理这个（request）请求

extraContext.put(SERVLET\_DISPATCHER, **this**);

OgnlValueStack stack = (OgnlValueStack) request.getAttribute(ServletActionContext.WEBWORK\_VALUESTACK\_KEY);

**if** (stack != **null**) ...{

extraContext.put(ActionContext.VALUE\_STACK,**new** OgnlValueStack(stack));

}

**try** ...{

ActionProxy proxy = ActionProxyFactory.getFactory().createActionProxy(namespace, actionName, extraContext);

//这里actionName是通过两道getActionName解析出来的, FilterDispatcher把请求的处理交给ActionProxy，下面是ServletDispatcher的 **TODO**:

request.setAttribute(ServletActionContext.WEBWORK\_VALUESTACK\_KEY, proxy.getInvocation().getStack());

proxy.execute();

//通过代理模式执行ActionProxy

**if** (stack != **null**)...{

request.setAttribute(ServletActionContext.WEBWORK\_VALUESTACK\_KEY,stack);

}

} **catch** (ConfigurationException e) ...{

log.error("Could not find action", e);

sendError(request, response, HttpServletResponse.SC\_NOT\_FOUND, e);

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

log.error("Could not execute action", e);

sendError(request, response, HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR, e);

} }

## 五、SpringCloud

### 一、微服务架构概述

#### 单体架构存在的缺点

复杂性逐渐提高、技术债务逐渐上升、部署速度逐渐变慢、阻碍技术创新、无法按需伸缩。

#### 什么是微服务

每个无服务都可以独立运行在自己的进程里

一系列的独立运行的微服务共同构建起了整个系统。

每个服务为独立业务开发，一个微服务一般完成某个特定功能。

微服务之间通过轻量级的通信机制通信，例如REST API或RPC。

#### 微服务特点

易于开发和维护、启动较快、局部修改易于部署、技术栈不受限、按需伸缩；运维成本较高、分布式的复杂性、接口调整成本较高、重复劳动。

#### 微服务设计原则

单一职责原则、服务自治原则、轻量级通信原则、接口明确原则。

### 二、开始使用spring cloud

#### Spring cloud特点

约定优于配置、开箱即用快速启动、适用于各种环境、轻量级组件、组件支持丰富功能齐全、选型中立。

### 三、服务提供者和服务消费者

### 四、服务发现和服务注册

## 六、试题

### 1、简单讲讲tomcat结构，以及其类加载器流程，线程模型等。

Server- –多个service

Container级别的：–>engine–》host–>context

Listenter

Connector

Logging、Naming、Session、JMX等等

### 2、tomcat如何调优，涉及哪些参数 。

硬件上选择，

操作系统选择:linux

版本选择:64位

jdk选择

配置jvm参数：

**-server**

我不管你什么理由，只要你的tomcat是运行在生产环境中的，这个参数必须给我加上因为tomcat默认是以一种叫java –client的模式来运行的，server即意味着你的tomcat是以真实的production的模式在运行的，这也就意味着你的tomcat以server模式运行时将拥有：更大、更高的并发处理能力，更快更强捷的JVM垃圾回收机制，可以获得更多的负载与吞吐量。。。更。。。还有更。。。Y给我记住啊，要不然这个-server都不加，那是要打屁股了。

**-Xms–Xmx**

即JVM内存设置了，把Xms与Xmx两个值设成一样是最优的做法，有人说Xms为最小值，Xmx为最大值不是挺好的，这样设置还比较人性化，科学化。人性？科学？你个头啊。大家想一下这样的场景：

一个系统随着并发数越来越高，它的内存使用情况逐步上升，上升到最高点不能上升了，开始回落，你们不要认为这个回落就是好事情，由其是大起大落，在内存回落时它付出的代价是CPU高速开始运转进行垃圾回收，此时严重的甚至会造成你的系统出现“卡壳”就是你在好好的操作，突然网页像死在那边一样几秒甚至十几秒时间，因为JVM正在进行垃圾回收。因此一开始我们就把这两个设成一样，使得Tomcat在启动时就为最大化参数充分利用系统的效率，这个道理和jdbcconnection pool里的minpool size与maxpool size的需要设成一个数量是一样的原理。

**–Xmn**

设置年轻代大小为512m。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。**持久代一般固定大小为64m**，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个**堆的3/8**。

**-Xss**

是指设定每个线程的堆栈大小。这个就要依据你的程序，看一个线程 大约需要占用多少内存，可能会有多少线程同时运行等。一般不易设置超过1M，要不然容易出现out ofmemory。

配置connector的线程数量:

<Connector port=*"8080"* protocol=*"HTTP/1.1"*

URIEncoding=*"UTF-8"* minSpareThreads=*"25"* maxSpareThreads=*"75"*

enableLookups=*"false"* disableUploadTimeout=*"true"* connectionTimeout=*"20000"*

acceptCount=*"300"* maxThreads=*"300"* maxProcessors=*"1000"* minProcessors=*"5"*

useURIValidationHack=*"false"* compression=*"on"* compressionMinSize=*"2048"*

compressableMimeType=*"text/html,text/xml,text/javascript,text/css,text/plain"* redirectPort=*"8443"*

**URIEncoding=”UTF-8”**:

使得tomcat可以解析含有中文名的文件的url，真方便，不像apache里还有搞个mod\_encoding，还要手工编译

**maxSpareThreads:**

maxSpareThreads 的意思就是如果空闲状态的线程数多于设置的数目，则将这些线程中止，减少这个池中的线程总数。

**minSpareThreads**

最小备用线程数，tomcat启动时的初始化的线程数。

**enableLookups**

这个功效和Apache中的HostnameLookups一样，设为关闭。如果为true，则可以通过调用request.getRemoteHost()进行DNS查询来得到远程客户端的实际主机名，若为false则不进行DNS查询，而是返回其ip地址。

**connectionTimeout**

connectionTimeout为网络连接超时时间毫秒数

**maxThreads**

maxThreads Tomcat使用线程来处理接收的每个请求。这个值表示Tomcat可创建的最大的线程数，即最大并发数。

**acceptCount**

acceptCount是当线程数达到maxThreads后，后续请求会被放入一个等待队列，这个acceptCount是这个队列的大小，如果这个队列也满了，就直接refuse connection

**maxProcessors与minProcessors**

在Java中线程是程序运行时的路径，是在一个程序中与其它控制线程无关的、能够独立运行的代码段。它们共享相同的地址空间。多线程帮助程序员写出CPU最大利用率的高效程序，使空闲时间保持最低，从而接受更多的请求。通常Windows是1000个左右，**Linux是2000个左右**。

**useURIValidationHack**

我们来看一下tomcat中的一段源码：

**if** (connector.getUseURIValidationHack()) {

String uri = validate(request.getRequestURI());

**if** (uri == **null**) {

res.setStatus(400);

res.setMessage("Invalid URI");

**throw** **new** IOException("Invalid URI");

} **else** {

req.requestURI().setString(uri);

// Redoing the URI decoding

req.decodedURI().duplicate(req.requestURI());

req.getURLDecoder().convert(req.decodedURI(), **true**);

}

}

可以看到如果把useURIValidationHack设成"false"，可以减少它对一些url的不必要的检查从而减省开销

开启gzip压缩:给Tomcat配置gzip压缩(HTTP压缩)功能:

compression="on" compressionMinSize="2048"

compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"

HTTP压缩可以大大提高浏览网站的速度，它的原理是，在客户端请求网页后，从服务器端将网页文件压缩，再下载到客户端，由客户端的浏览器负责解压缩并浏览。相对于普通的浏览过程HTML,CSS,Javascript , Text ，它可以**节省40%左右的流量**。更为重要的是，它可以对动态生成的，包括CGI、PHP , JSP , ASP , Servlet,SHTML等输出的网页也能进行压缩，压缩效率惊人。

1)compression="on" 打开压缩功能

2)compressionMinSize="2048" 启用压缩的输出内容大小，这里面默认为2KB

3)noCompressionUserAgents="gozilla, traviata" 对于以下的浏览器，不启用压缩

4)compressableMimeType="text/html,text/xml"　压缩类型

最后不要忘了把8443端口的地方也加上同样的配置，因为如果我们走https协议的话，我们将会用到8443端口这个段的配置，对吧？

<!--enable tomcat ssl-->

<Connector port=*"8443"* protocol=*"HTTP/1.1"*

URIEncoding=*"UTF-8"* minSpareThreads=*"25"* maxSpareThreads=*"75"*

enableLookups=*"false"* disableUploadTimeout=*"true"* connectionTimeout=*"20000"*

acceptCount=*"300"* maxThreads=*"300"* maxProcessors=*"1000"* minProcessors=*"5"*

useURIValidationHack=*"false"*

compression=*"on"* compressionMinSize=*"2048"*

compressableMimeType=*"text/html,text/xml,text/javascript,text/css,text/plain"*

SSLEnabled=*"true"*

scheme=*"https"* secure=*"true"*

clientAuth=*"false"* sslProtocol=*"TLS"*

keystoreFile=*"d:/tomcat2/conf/shnlap93.jks"* keystorePass=*"aaaaaa"* /> trimSpaces:去掉jsp中的空白行

### 3、讲讲Spring加载流程。

(1).首先，对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是**ServletContext**，其为后面的spring IoC容器提供宿主环境；

(2).其 次，在web.xml中会提供有contextLoaderListener。在web容器启动时，会触发容器初始化事件，此时 contextLoaderListener会监听到这个事件，其contextInitialized方法会被调用，在这个方法中，spring会初始 化一个启动上下文，这个上下文被称为根上下文，即WebApplicationContext，这是一个接口类，确切的说，其实际的实现类是 **XmlWebApplicationContext**。这个就是spring的IoC容器，其对应的Bean定义的配置由web.xml中的 context-param标签指定。在这个IoC容器初始化完毕后，spring以WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE为属性Key，将其存储到ServletContext中，便于获取；

(3).再 次，contextLoaderListener监听器初始化完毕后，开始初始化web.xml中配置的Servlet，这里是**DispatcherServlet**，这个servlet实际上是一个标准的前端控制器，用以转发、匹配、处理每个servlet请 求。DispatcherServlet上下文在初始化的时候会建立自己的IoC上下文，用以持有springmvc相关的bean。在建立DispatcherServlet自己的IoC上下文时，会利用WebApplicationContext.ROOTWEBAPPLICATIONCONTEXTATTRIBUTE 先从ServletContext中获取之前的根上下文(即WebApplicationContext)作为自己上下文的parent上下文。有了这个 parent上下文之后，再初始化自己持有的上下文。这个DispatcherServlet初始化自己上下文的工作在其initStrategies方 法中可以看到，大概的工作就是初始化处理器映射、视图解析等。这个servlet自己持有的上下文默认实现类也是 xmlWebApplicationContext。初始化完毕后，spring以与servlet的名字相关(此处不是简单的以servlet名为 Key，而是通过一些转换，具体可自行查看源码)的属性为属性Key，也将其存到ServletContext中，以便后续使用。这样每个servlet 就持有自己的上下文，即拥有自己独立的bean空间，同时各个servlet共享相同的bean，即根上下文 (第2步中初始化的上下文)定义的那些 bean。

### 4、Spring AOP的实现原理。

#### (1).什么是AOP

**AOP（Aspect-Oriented Programming，面向方面编程）**，可以说是**OOP（Object-Oriented Programing，面向对象编程）**的补充和完善。

OOP引入**封装**、**继承**和**多态性**等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，**OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系**。

AOP技术它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于**减少系统的重复代码**，**降低模块间的耦合度**，并有利于未来的**可操作性**和**可维护性**。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天功的妙手将这些剖开的切面复原，不留痕迹。

AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。

#### (2).AOP实现原理

一是采用**动态代理技术**，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；

二是采用**静态织入**的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

### 5、讲讲Spring事务的传播属性。

#### 七个事物传播属性：

**PROPAGATION\_REQUIRED** -- 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

**PROPAGATION\_SUPPORTS** -- 支持当前事务，如果当前没有事务，就以非事务方式执行。

**PROPAGATION\_MANDATORY** -- 支持当前事务，如果当前没有事务，就抛出异常。

**PROPAGATION\_REQUIRES\_NEW** -- 新建事务，如果当前存在事务，把当前事务挂起。

**PROPAGATION\_NOT\_SUPPORTED** -- 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

**PROPAGATION\_NEVER** -- 以非事务方式执行，如果当前存在事务，则抛出异常。

**PROPAGATION\_NESTED** -- 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION\_REQUIRED类似的操作。

#### 五个隔离级别

**ISOLATION\_DEFAULT：**这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.另外四个与JDBC的隔离级别相对应；

**ISOLATION\_READ\_UNCOMMITTED：** 这是事务最低的隔离级别，它充许别外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读，不可重复读和幻像读。

**ISOLATION\_READ\_COMMITTED：**保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现，但是可能会出现不可重复读和幻像读。

**ISOLATION\_REPEATABLE\_READ：** 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)。

**ISOLATION\_SERIALIZABLE ：**这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读，不可重复读外，还避免了幻像读。

### 6、Spring如何管理事务的。

Spring并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。Spring事务管理器的接口是org.springframework.transaction.**PlatformTransactionManager**，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了

1）、PlatformTransactionManager：事务管理器，主要用于平台相关事务的管理。主要包括三个方法：

①、commit：事务提交。

②、rollback：事务回滚。

③、getTransaction：获取事务状态。

2）、TransacitonDefinition：事务定义信息，用来定义事务相关属性，给事务管理器PlatformTransactionManager使用这个接口有下面四个主要方法：

①、getIsolationLevel：获取隔离级别。

②、getPropagationBehavior：获取传播行为。

③、getTimeout获取超时时间。

④、isReadOnly：是否只读（保存、更新、删除时属性变为false--可读写，查询时为true--只读）事务管理器能够根据这个返回值进行优化，这些事务的配置信息，都可以通过配置文件进行配置。

  3）、TransationStatus：事务具体运行状态，事务管理过程中，每个时间点事务的状态信息。例如：

①、hasSavepoint()：返回这个事务内部是否包含一个保存点。

②、isCompleted()：返回该事务是否已完成，也就是说，是否已经提交或回滚。

③、isNewTransaction()：判断当前事务是否是一个新事务

### 7、Spring怎么配置事务（具体说出一些关键的xml元素）。

#### 织入式事务

<!-- 1.将连接池 -->

<bean name=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"jdbcUrl"* value=*"${jdbc.jdbcUrl}"*></property>

<property name=*"driverClass"* value=*"${jdbc.driverClass}"*></property>

<property name=*"user"* value=*"${jdbc.user}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<!-- 事务核心管理器,封装了所有事务操作. 依赖于连接池 -->

<bean name=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 配置事务通知 -->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!-- 以方法为单位,指定方法应用什么事务属性 isolation:隔离级别 propagation:传播行为 read-only:是否只读 -->

<tx:method name=*"save\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"false"* />

<tx:method name=*"persist\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"false"* />

<tx:method name=*"update\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"false"* />

<tx:method name=*"modify\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"false"* />

<tx:method name=*"delete\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"false"* />

<tx:method name=*"remove\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"false"* />

<tx:method name=*"get\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"true"* />

<tx:method name=*"find\*"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"true"* />

<tx:method name=*"transfer"* isolation=*"REPEATABLE\_READ"*

propagation=*"REQUIRED"* read-only=*"false"* />

</tx:attributes>

</tx:advice>

<!-- 配置织入 -->

<aop:config>

<!-- 配置切点表达式 -->

<aop:pointcut expression=*"execution(\* cn.itcast.service.\*ServiceImpl.\*(..))"* id=*"txPc"* />

<!-- 配置切面 : 通知+切点 advice-ref:通知的名称 pointcut-ref:切点的名称 -->

<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"txPc"* />

</aop:config>

#### 注解式事务

<!-- 事务核心管理器,封装了所有事务操作. 依赖于连接池 -->

<bean name=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 开启使用注解管理aop事务 -->

<tx:annotation-driven transaction-manager=*"transactionManager"* />

### 8、说说你对Spring的理解，非单例注入的原理？它的生命周期？循环注入的原理，aop的实现原理，说说aop中的几个术语

它们是怎么相互工作的。

1）、基于XML的事务配置。

2）、基于注解方式的事务配置。

铺垫：

1）、spring的事务管理是通过Aop的方式来实现。

2）、声明式事务是spring对事务管理的最常用的方式，因为这种方式对代码的影响最小，因此也就符合非侵入式的轻量级的容器的概念；

### 9、Springmvc 中DispatcherServlet初始化过程。

首先，我们从web.xml中开始，在web.xml中我们首先配置的是contextLoaderListener，它的作用就是启动web容器时，自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener这个接口，在web.xml配置这个监听器，启动容器时，就会自动执行它实现的contextInitialized()方法。这样就能够在客户端请求之前向ServletContext中添加任意的对象；在ServletContextListener中的核心逻辑便是初始化WebApplicationContext实例并存放至ServletContext中。

ContextLoaderListener监听器的作用就是启动Web容器时，自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener这个接口，在web.xml配置了这个监听器，启动容器时，就会默认执行它实现的contextInitialized()方法初始化WebApplicationContext实例，并放入到ServletContext中。由于在ContextLoaderListener中关联了ContextLoader这个类，所以整个加载配置过程由ContextLoader来完成。

### netty的线程模型，netty如何基于reactor模型上实现的。

Netty 的线程模型是基于NIO的Selector 构建的，使用了异步驱动的Reactor 模式来构建的线程模型，可以很好的支持成百上千的 SocketChannel 连接。由于 READ/WRITE 都是非阻塞的，可以充分提升I/O线程的运行效率 ，避免了IO阻塞导致线程挂起， 同时可以让一个线程支持对多个客户端的连接SocketChannel的 READ/WRITE 操作， 从根本上解决了传统阻塞IO的一线程处理一连接的弊端。

#### Reactor 模式

是一种为处理服务请求并发,提交到一个或者多个服务处理程序的事件设计模式。当请求抵达后，服务处理程序使用解多路分配策略，然后同步地派发这些请求至相关的请求处理程序。

##### 1、单线程reactor

ractor 单线程模式是指所有的I/O操作都在一个NIO线程完成，该线程的职责：

1.作为NIO服务端，接收客户端TCP连接

2.作为NIO客户端，向客户端发送TCP连接

3.READ/WRITE 客户端的请求

### 11、为什么选择netty。

### 12、什么是TCP粘包，拆包。解决方式是什么。

如果客户端连续不断的向服务端发送数据包时，服务端接收的数据会出现两个数据包粘在一起的情况，这就是TCP协议中经常会遇到的粘包以及拆包的问题；传输层除了有TCP协议外还有UDP协议。那么UDP是否会发生粘包或拆包的现象呢？

答案是不会。**UDP是基于报文发送**的，从UDP的帧结构可以看出，在UDP首部采用了16bit来指示UDP数据报文的长度，因此在应用层能很好的将不同的数据报文区分开，从而避免粘包和拆包的问题。而**TCP是基于字节流**的，虽然应用层和TCP传输层之间的数据交互是大小不等的数据块，但是TCP把这些数据块仅仅看成一连串无结构的字节流，没有边界；

另外从TCP的帧结构也可以看出，在TCP的首部没有表示数据长度的字段，基于上面两点，在使用TCP传输数据时，才有粘包或者拆包现象发生的可能。

发生TCP粘包或拆包有很多原因，现列出常见的几点，可能不全面，欢迎补充，

1、要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包。

2、待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。

3、要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。

4、接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。

**粘包、拆包解决办法**：

通过以上分析，我们清楚了粘包或拆包发生的原因，那么如何解决这个问题呢？解决问题的关键在于如何给每个数据包添加边界信息，常用的方法有如下几个：

1、发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。

2、发送端将每个数据包封装为固定长度（不够的可以通过补0填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。

3、可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

### 13、netty的fashwheeltimer的用法，实现原理，是否出现过调用不够准时，怎么解决。

### 14、netty的心跳处理在弱网下怎么办。

### 15、netty的通讯协议是什么样的。

### 16、springmvc用到的注解，作用是什么，原理。

### 17、springboot启动机制。

### 18、[SpringMVC和Struts2区别比较](https://www.cnblogs.com/huajiezh/p/6415444.html)

1**、**Struts2是类级别的拦截， 一个类对应一个request上下文，SpringMVC是方法级别的拦截，一个方法对应一个request上下文，而方法同时又跟一个url对应,所以说从[架构](http://lib.csdn.net/base/16)本身上SpringMVC就容易实现restful url,而struts2的架构实现起来要费劲，因为Struts2中Action的一个方法可以对应一个url，而其类属性却被所有方法共享，这也就无法用注解或其他方式标识其所属方法了。springmvc是基于方法开发的，struts2是基于类开发的。

2、由上边原因，SpringMVC的方法之间基本上独立的，独享request response数据，请求数据通过参数获取，处理结果通过ModelMap交回给框架，方法之间不共享变量，而Struts2搞的就比较乱，虽然方法之间也是独立的，但其所有Action变量是共享的，这不会影响程序运行，却给我们编码 读程序时带来麻烦，每次来了请求就创建一个Action，一个Action对象对应一个request上下文。

因为二者基于的出发点不同，所以在接收请求参数时区别比较大。springmvc是通过参数绑定接收浏览器请求参数（方法上各种形参(request、response、session、Model、ModelMap、简单类型、简单pojo、包装类pojo)浏览器的请求参数），方法执行结束，形参数据就会销毁。 struts2是通过类中的成员变量接收浏览器请求参数的。

**3、**由于Struts2需要针对每个request进行封装，把request，session等servlet生命周期的变量封装成一个一个Map，供给每个Action使用，并保证线程安全，所以在原则上，是比较耗费内存的。  
springmvc：因为方法之间是线程隔离的，所以springmvc支持单例和多例的开发方式。  
struts2：成员变量在开发中多线程会存在线程安全问题，所以struts2只能支持多例的开发方式。

4、 拦截器实现机制上，Struts2有以自己的interceptor机制，SpringMVC用的是独立的AOP方式，这样导致Struts2的配置文件量还是比SpringMVC大。

5、SpringMVC的入口是**DispatchServlet**，而Struts2是**StrutsPrepareAndExecuteFilter**（这里要指出，filter和servlet是不同的。以前认为filter是servlet的一种特殊），这就导致了二者的机制不同，这里就牵涉到servlet和filter的区别了。

6、SpringMVC集成了Ajax，使用非常方便，只需一个注解@ResponseBody就可以实现，然后直接返回响应文本即可，而Struts2拦截器集成了Ajax，在Action中处理时一般必须安装插件或者自己写代码集成进去，使用起来也相对不方便。

7、SpringMVC验证支持JSR303，处理起来相对更加灵活方便，而Struts2验证比较繁琐，感觉太烦乱。

8、Spring MVC和Spring是无缝的。从这个项目的管理和安全上也比Struts2高（当然Struts2也可以通过不同的目录结构和相关配置做到SpringMVC一样的效果，但是需要xml配置的地方不少）。

9、设计思想上，Struts2更加符合OOP的编程思想， SpringMVC就比较谨慎，在servlet上扩展。

10、SpringMVC开发效率和性能高于Struts2。  
11、SpringMVC可以认为已经100%零配置。

12、springmvc是通过存储数据ModelAndView，在前台页面通过jstl(默认)展示数据的、struts2是通过值栈存储数据，在页面通过OGNL展示数据的。

### 19、拦截器（Interceptor）和过滤器（Filter）的区别

| **Filter** | **Interceptor** | **总结** |
| --- | --- | --- |
| Filter 接口定义在 javax.servlet 包中 | 接口 HandlerInterceptor 定义在org.springframework.web.servlet 包中 |  |
| Filter 定义在 web.xml 中 | 定义struts.xml或者spring-mvc.xml中 |  |
| Filter在只在 Servlet 前后起作用。Filters 通常将 请求和响应（request/response） 当做黑盒子，Filter 通常不考虑servlet 的实现。 | 拦截器能够深入到方法前后、异常抛出前后等，因此拦截器的使用具有更大的弹性。允许用户介入（hook into）请求的生命周期，在请求过程中获取信息，Interceptor 通常和请求更加耦合。 | 在Spring构架的程序中，要优先使用拦截器。几乎所有 Filter 能够做的事情， interceptor 都能够轻松的实现 |
| Filter 是 Servlet 规范规定的。 | 而拦截器既可以用于Web程序，也可以用于Application、Swing程序中。 | 使用范围不同 |
| Filter 是在 Servlet 规范中定义的，是 Servlet 容器支持的。 | 而拦截器是在 Spring容器内的，是Spring框架支持的。 | 规范不同 |
| Filter 不能够使用 Spring 容器资源 | 拦截器是一个Spring的组件，归Spring管理，配置在Spring文件中，因此能使用Spring里的任何资源、对象，例如 Service对象、数据源、事务管理等，通过IoC注入到拦截器即可 | Spring 中使用 interceptor 更容易 |
| Filter 是被 Server(like Tomcat) 调用 | Interceptor 是被 Spring 调用 | 因此 Filter 总是优先于 Interceptor 执行 |

### 20、谈谈你对Struts2的理解。

struts2是一个按照MVC模式设计的一个web层的框架,其实它就相当于一个大的servlet.但是它和servlet是有区别的.servlet默认在第一次访问的时候创建,只创建一次,是单实例对象.struts2的action是多实例对象,每次访问的时候都会创建action对象

struts.xml配置文件中的package标签的namespace属性值和action标签里面的name属性值构成访问路径,action标签里的class值表示所对应的Action类的全路径.如果不配置method属性的时候,Action类里面默认执行的方法是execute方法.action标签里的method属性值对应类中Action的方法名.在result标签里面配置方法的返回值到页面.但是,如果一个类的方法很多,就需要写多个action标签,这时可以使用通配符来简化配置。

在继承ActionSupport类后,可以使用ActionContext和ServletActionContext类来获取表单提交的数据。

struts2封装表单数据的方式,就是将表单数据封装到实体类对象中..可以使用表达式封装(主要是通过生成get,set方法来完成页面和action的双向交互),模型驱动封装(通过实现ModelDriven接口类)

在Action类里面可以把数据存到值栈里面,在页面中获取到值栈的数据;在页面中可以通过struts2标签和ognl表达式来获取值栈中的数据。

### 21、谈谈你对Hibernate的理解

1. 面向对象设计的软件内部运行过程可以理解成就是在不断创建各种新对象、建立对象之间的关系，调用对象的方法来改变各个对象的状态和对象消亡的过程，不管程序运行的过程和操作怎么样，本质上都是要得到一个结果，程序上一个时刻和下一个时刻的运行结果的差异就表现在内存中的对象状态发生了变化。

2.为了在关机和内存空间不够的状况下，保持程序的运行状态，需要将内存中的对象状态保存到持久化设备和从持久化设备中恢复出对象的状态，通常都是保存到关系数据库来保存大量对象信息。从Java程序的运行功能上来讲，保存对象状态的功能相比系统运行的其他功能来说，应该是一个很不起眼的附属功能，java采用jdbc来实现这个功能，这个不起眼的功能却要编写大量的代码，而做的事情仅仅是保存对象和恢复对象，并且那些大量的jdbc代码并没有什么技术含量，基本上是采用一套例行公事的标准代码模板来编写，是一种苦活和重复性的工作。

3.通过数据库保存java程序运行时产生的对象和恢复对象，其实就是实现了java对象与关系数据库记录的映射关系，称为**ORM（即Object Relation Mapping）**，人们可以通过封装JDBC代码来实现了这种功能，封装出来的产品称之为ORM框架，Hibernate就是其中的一种流行ORM框架。使用Hibernate框架，不用写JDBC代码，仅仅是调用一个save方法，就可以将对象保存到关系数据库中，仅仅是调用一个get方法，就可以从数据库中加载出一个对象。

4.使用Hibernate的基本流程是：配置Configuration对象、产生SessionFactory、创建session对象，启动事务，完成CRUD操作，提交事务，关闭session。

5.使用Hibernate时，先要配置hibernate.cfg.xml文件，其中配置数据库连接信息和方言等，还要为每个实体配置相应的hbm.xml文件，hibernate.cfg.xml文件中需要登记每个hbm.xml文件。

6.在应用Hibernate时，重点要了解Session的缓存原理，级联，延迟加载和hql查询。

### 22、对Spring的理解。

1.Spring实现了工厂模式的工厂类（在这里有必要解释清楚什么是工厂模式），这个类名为BeanFactory（实际上是一个接口），在程序中通常BeanFactory的子类ApplicationContext。Spring相当于一个大的工厂类，在其配置文件中通过<bean>元素配置用于创建实例对象的类名和实例对象的属性。

2. Spring提供了对IOC良好支持，IOC是一种编程思想，是一种架构艺术，利用这种思想可以很好地实现模块之间的解耦。IOC也称为DI（Depency Injection），什么叫依赖注入呢？譬如：

Class Programmer{

Computer computer = **null**;

**public** **void** code(){

Computer computer = **new** IBMComputer();//1

Computer computer = beanfacotry.getComputer();//2

computer.write();

}

**public** **void** setComputer(Computer computer){

**this**.computer = computer;

}

}

另外两种方式都由依赖，第一个直接依赖于目标类，第二个把依赖转移到工厂上，第三个彻底与目标和工厂解耦了。在spring的配置文件中配置片段如下：

<bean id=*”computer”* class=*”cn.itcast.interview.Computer”*></bean>

<bean id=*”programmer”* class=*”cn.itcast.interview.Programmer”*>

<property name=*”computer”* ref=*”computer”*></property>

</bean>

3. Spring提供了对AOP技术的良好封装， AOP称为面向切面编程，就是系统中有很多各不相干的类的方法，在这些众多方法中要加入某种系统功能的代码，例如，加入**日志**，加入**权限判断**，加入**异常处理**，加入**事务控制**。这种应用称为AOP。实现AOP功能采用的是代理技术，客户端程序不再调用目标，而调用代理类，代理类与目标类对外具有相同的方法声明，有两种方式可以实现相同的方法声明，一是实现相同的接口，二是作为目标的子类在，JDK中采用Proxy类产生动态代理的方式为某个接口生成实现类，如果要为某个类生成子类，则可以用CGLI B。在生成的代理类的方法中加入系统功能和调用目标类的相应方法，系统功能的代理以Advice对象进行提供，显然要创建出代理对象，至少需要目标类和Advice类。spring提供了这种支持，只需要在spring配置文件中配置这两个元素即可实现代理和aop功能，例如：

<bean id=*”proxy”* type=*”org.spring.framework.aop.ProxyBeanFactory”*>

<property name=*”target”* ref=*””*></property>

<property name=*”advisor”* ref=*””*></property>

</bean>

### 23、Struts2优缺点

**优点：**

1. 实现MVC模式，结构清晰,使开发者只关注业务逻辑的实现。

2．有丰富的tag可以用 ,Struts的标记库(Taglib)，如能灵活动用，则能大大提高开发效率。

3. 页面导航，使系统的脉络更加清晰。通过一个配置文件，即可把握整个系统各部分之间的联系，这对于后期的维护有着莫大的好处。尤其是当另一批开发者接手这个项目时，这种优势体现得更加明显。

4. 提供Exception处理机制 。

5. 数据库链接池管理 。

6. 支持I18N 。

**缺点：**

1.转到展示层时，需要配置forward，如果有十个展示层的jsp，需要配置十次struts，而且还不包括有时候目录、文件变更，需要重新修改forward，注意，每次修改配置之后，要求重新部署整个项目，而tomcat这样的服务器，还必须重新启动服务器。

2.Struts2 的Action必需是thread－safe方式，它仅仅允许一个实例去处理所有的请求。所以action用到的所有的资源都必需统一同步，这个就引起了线程安全的问题。

3.测试不方便. Struts的每个Action都同Web层耦合在一起，这样它的测试依赖于Web容器，单元测试也很难实现。不过有一个Junit的扩展工具Struts TestCase可以实现它的单元测试。

4. 类型的转换. Struts的FormBean把所有的数据都作为String类型，它可以使用工具Commons-Beanutils进行类型转化。但它的转化都是在Class级别，而且转化的类型是不可配置的。类型转化时的错误信息返回给用户也是非常困难的。

5.对Servlet的依赖性过强. Struts处理Action时必需要依赖ServletRequest 和ServletResponse，所有它摆脱不了Servlet容器。

6.前端表达式语言方面.Struts集成了JSTL，所以它主要使用JSTL的表达式语言来获取数据。可是JSTL的表达式语言在Collection和索引属性方面处理显得很弱。

7.对Action执行的控制困难. Struts创建一个Action，如果想控制它的执行顺序将会非常困难。甚至你要重新去写Servlet来实现你的这个功能需求。

8.对Action 执行前和后的处理. Struts处理Action的时候是基于class的hierarchies，很难在action处理前和后进行操作。

9. 对事件支持不够，在struts中，实际是一个表单Form对应一个Action类(或DispatchAction)，换一句话说：在Struts中实际是一个表单只能 对应一个事件，struts这种事件方式称为application event，application event和component event相比是一种粗粒度的事件。

### 24、说说struts1与struts2的区别。

1.都是MVC的WEB框架,

2 struts1的老牌框架，应用很广泛，有很好的群众基础，使用它开发风险很小，成本更低！struts2虽然基于这个框架，但是应用群众并多，相对不成熟，未知的风险和变化很多，开发人员相对不好招，使用它开发项目的风险系数更大，用人成本更高！

3.struts2毕竟是站在前辈的基础设计出来，它会改善和完善struts1中的一些缺陷，struts1中一些悬而未决问题在struts2得到了解决。

4.struts1的前端控制器是一个Servlet，名称为ActionServlet，struts2的前端控制器是一个filter，在struts2.0中叫FilterDispatcher，在struts2.1中叫StrutsPrepareAndExecuteFilter。

5.struts1的action需要继承Action类，struts2的action可以不继承任何类；struts1对同一个路径的所有请求共享一个Action实例，struts2对同一个路径的每个请求分别使用一个独立Action实例对象，所有对于struts2的Action不用考虑线程安全问题。

6.在struts1中使用formbean封装请求参数，在struts2中直接使用action的属性来封装请求参数。

7.struts1中的多个业务方法放在一个Action中时（即继承DispatchAction时），要么都校验，要么都不校验；对于struts2，可以指定只对某个方法进行校验，当一个Action继承了ActionSupport且在这个类中只编写了validateXxx()方法，那么则只对Xxx()方法进行校验。

（一个请求来了的执行流程进行分析，struts2是自动支持分模块开发，并可以不同模块设置不同的url前缀，这是通过package的namespace来实现的；struts2是支持多种类型的视图；struts2的视图地址可以是动态的，即视图的名称是支持变量方式的，举例，论坛发帖失败后回来还要传递boardid。视图内容显示方面：它的标签用ognl，要el强大很多，在国际化方面支持分模块管理，两个模块用到同样的key，对应不同的消息；）

与Struts1不同，Struts2对用户的每一次请求都会创建一个Action，所以Struts2中的Action是线程安全的。

给我印象最深刻的是：struts配置文件中的redirect视图的url不能接受参数，而struts2配置文件中的redirect视图可以接受参数。

### 25、struts的核心组件有哪些？

* 1. FilterDispatcher,struts2的核心组件，整个struts2的调度中心，它对请求进行过滤并决定struts2是否出来该请求。
  2. Strtus.xml：struts2的应用配置文件，它负责配置系统中用到的action
  3. Action:strtus2的动作执行单元实际处理用户的请求，封装业务所需的数据

d)Result：action运行后要转向下一个资源，可以是视图也可以说其他的action

### 26、struts2获取request的方式？

实现servletRequest接口，然后写一个request属性，写上set get方法

通过ServletContext上下文来获取

### 27、Strus2的执行过程

1.客户端浏览器发出http请求

2.根据web.xml配置，该请求被filterDispatcher接收。

3.根据struts.xml配置，找到需要调用的action类和方法，并通过IOC控制反转方式，将值注入给Aciton

4.Action调用业务逻辑组件处理业务逻辑，这一步包含表单验证。

5.Action执行完毕，根据struts.xml中的配置找到对应的翻译结果Result，并跳转到相应的界面。

6.返回http响应到客户端浏览器。

### 28、为什么要使用struts2？

开源、mvc框架、纯pojo的action、更好的标签特性、易测性、易扩展性

### 29、struts.xml中result的type有哪些类型？

Dispatcher：struts2默认的结果类型，把控制权转发给应用程序里的某个资源，不能把控制权转发给一个外部资源，若需要把控制权重定向到一个外部资源，应该使用redirect结果类型。

Redirect 把响应重定向到另一个资源

RedirectAction 把响应重定向到另一个Action

Freemarcker、velocity、chain、httpherder、xslt、plainText、stream、json.

### 30、Struts2返回json数据

1.type配置为json

<package name=*"mystruts"* extends=*"struts-default,json-default"*>

<action name=*"api\_\*"* class=*"com.xx.xx.Action"* method=*"{1}"*>

<result name=*"success"* type=*"json"*></result>

</action>

</package>

1. 借助输出流

PrintWriter writer = **null**;

HttpServletResponse response = ServletActionContext.getResponse();

response.setContentType("application/json;charset=utf-8");

writer = response.getWriter();

response.setDateHeader("Expires", 0);

JSONObject object = JSONObject.fromObject(out);

writer.write(object.toString());

writer.flush();

### 31、一般情况下，关系数据模型与对象模型之间有哪些匹配关系?

表对应类、记录对应对象、表的字段对应类的属性

### 32、hibernate 数据的三个状态

我们把数据库比喻成一个登记簿吧，当我们把信息登记到登记簿上的时候，这个过程就是**持久化的过程**。当登记完成之后，登记簿上的数据就是**持久态数据**，所谓持久态就是一直存在的状态。当我们需要用数据的时候，会从登记簿上查信息，我们查到的信息记录在临时的纸张或者脑海里，然后登记簿会放回原位，这时候临时纸张上或者脑海里的数据就是**游离态**，随时可能被遗忘，在hibernate中就是随时会被销毁的数据。**瞬态数据**和**游离态数据**有点儿类似，但是一点，就是他是存在持久化之前的数据，就是说我需要登记个人信息，**我的个人信息还没有登记到登记簿上的时候，信息就是瞬态，一旦登记到登记簿上就变成持久态了，然后再查询到的就是游离态。**

**a. 瞬时状态（临时状态）**

当**new** 对象时候，处于瞬时状态（如果程序运行完了，该对象会被垃圾回收）。

**b. 持久状态**

跟session有关，就是持久状态，持久状态的对象，任何的修改，都会影响到数据库中与之对应的数据。

**c. 托管状态（游离状态）**

当session不在管理对象的时候，脱离了session 的管理，处于托管状态的对象，修改属性，对数据库数据没有任何影响。 脱管对象拥有数据库的识别值，可通过update()、saveOrUpdate()等方法，转变成持久对象。

企业开发中,使用saveOrUpdate(obj): 来替代save(obj)或update(obj)方法 ，避免因为状态的改变,导致方法出错,saveOrUpdate(obj)可以根据obj的状态,来选择是save()还是update()。

**脱管对象具有如下特点**：

  1.本质上与瞬时对象相同，在没有任何变量引用它时，JVM会在适当的时候将它回收；

   2.比瞬时对象多了一个数据库记录标识值。

**状态转变过程：**

User user = **new** User(); //创建一个瞬态对象

user.setName(userName);

user.setPassword("test");

Session session = sessionFactory.openSession();

Tansaction tx = session.beginTansaction();

session.save(user);

//此时user已变为持久态

tx.commit();

Tansaction tx2 = session.beginTansaction();

user.setPassword("pdw");

tx2.commit();

//虽然在这个事物中并没有调用session的save()方法来保存user对象，但是usr处于持久态，

//所以对user对象所做的任何修改都持久化到数据库中 ---------持久态自动更新

//那么数据库中的密码也变成了pdw

session.close();

//session close完后，对象就变成游离态。

另一种方法直接得到持久态的对象：

Session session = sessionFactory.openSession();

Tansaction tx = session.beginTansaction();

User user = (User) session.load(User.**class**,"key")；//相当于hql 从DB读出来的 当然是持久态

tx.commit();

session.close();

### 33、Hibernate中load和get的区别？

1.如果数据库中，没有userId的对象，如果通过get方法加载，则返回的是一个Null；如果通过Load则返回一个代理对象，如果后面代码调用user对象的某个属性,会抛出objectNotFoundException

2.Load支持延迟加载，get不支持。

### 34、getCurrentSession与openSession的区别

1.getCurrentSession创建的session会绑定到当前线程,而openSession不会。

2.getCurrentSession创建的线程会在事务回滚或事物提交后自动关闭,而**openSession必须手动关闭。**

### 35、Hibernate的工作原理

1.读取并解析配置文件

2.读取解析映射信息，创建sessionFactory

3.打开session

4.创建事务Transation

5.持久化操作

6.提交事务

7.关闭session

8.关闭sessionFactory

### 36、为什么要用hibernate？

1.对jdbc访问数据库的代码做了封装，大大简化了数据访问层繁琐的重复性代码。

2.Hibernate是一个基于JDBC的主流持久性框架，是一个优秀的ORM实现，他很大程度的简化DAO的编码工作。

3.Hibernate使用JAVA反射机制，而不是字节码增强程序来实现透明性。

### 37、Hibernate是如何延迟加载的？

Lazy策略为延迟加载策略，Hibernate通过JDK代理对其进行实现，即使用延迟加载的对象，在获取对象的时候返回的是对象的代理，并不是对象的真正引用，只有在对象真正被调用的时候，Hibernate才会对其进行查询，返回真正的对象。

这在某种程度上对性能起到一定的优化。Hibernate的延迟加载还可以减少程序与数据库的连接次数，因为使用了延迟加载，Hibernate将延缓执行SQL语句，减少对数据库的访问次数，从而提高执行效率。

### 38、hibernate的缓存

Hibernate框架也应用了缓存技术，并实现了两级缓存，一级缓存即Session的缓存，二级缓存即SessionFactory的缓存。

**一级缓存 session级缓存**

一级缓存是Session级的缓存，其生命周期很短，与Session相互对应。一级缓存由Hibernate进行管理，属于事务范围的缓存。

当程序调用Session的load()方法、get()方法、save()方法、saveOrUpdate()方法、update()方法或查询接口方法时，Hibernate会对实体对象进行缓存；当通过load()方法或get()方法查询实体对象时，Hibernate会首先到缓存中查找，在找不到实体对象的情况下，Hibernate才会发出SQL语句到数据库中查询，从而提高了Hibernate的使用效率。

**二级缓存 SessionFactory级缓存**

二级缓存是SessionFactory级的缓存，其生命周期与SessionFactory一致。二级缓存可以在多个Session之间共享，属于进程范围或群集范围的缓存。

二级缓存是一个可插拔的缓存插件，它的使用需要第三方缓存产品的支持。在Hibernate框架中，通过Hibernate配置文件配置二级缓存的使用策略。

注：对于二级缓存，可以使用一些不经常更新的数据或参考的数据，此时其性能会得到明显的提升。例如一个新闻网站，当发布一条热点新闻时，会有成千上万的访问量，而此条新闻并没有发生任何的变化，如果每一个用户访问都要查询数据库，势必对系统性能造成一定的影响，此时可以考虑应用二级缓存。如果经常变换的数据则不应应用二级缓存。

### 39、如何优化Hibernate？

1.使用双向一对多关联，不使用单向一对多

2.灵活使用单向一对多

3.不使用一对一，用多对一取代

4.配置对象缓存，不适用集合缓存

5.一对多集合使用bag，多对多使用set

6.继承类使用显式多态

7.表字段要少，表关联不要怕多，有二级缓存。

### 40、Hibernate的主键生成策略？

1.sequence,通知Hibernate框架，主键的值采用指定序列生成，然后插入数据库，主要用于Oracle,DB2,不用程序员参与。

<generator class=*"sequence"*>

<param name=*"sequence"*>foo\_seq</param><!-- 必须加上 -->

</generator>

2.identity,通知hibernate框架，主键值采用数据库自动增长机制，每次进行save()操作，hibernate都会根据（数据库）自增的方式，生成一个id值，不用程序员参与，主要用于mySQL , SQLServer

<generator class=*"identity"*></generator>

3.uuid(西方常用),hibernate 每次进行save()操作，都会随机生成一个32的不重复的字符串，不用程序员去参与维护，PO类的Id属性必须为String

4.native 根据dialect(方言)不同,来自动的选择identity或sequence智能选择。是企业中常用的

5.assigned 不推荐使用，程序言要自己维护主键的Id值，当数据量很大时候很难维护

### 41、Hibernate的级联操作

1.cascade操作

all:所有情况下都进行级练操作 ，save-update和delete

save-update: 在进行save()/update()/saveOrUpdate时候进行级练操作

delete:在进行delete时候进行级练操作

all-delete-orphan :适合集合中删除，在返回的集合中执行remove()操作

none:在任何情况下都不进行级练操作

2.inverse属性的作用

是否放弃维护关联关系 true放弃 false不放弃

### 42、Hibernate有哪5个核心接口？

**Configuration**接口：配置Hibernate，根据其启动hibernate，创建SessionFactory对象；

**SessionFactory**接口：初始化Hibernate，充当数据存储源的代理，创建session对象，sessionFactory是线程安全的，意味着它的同一个实例可以被应用的多个线程共享，是重量级、二级缓存；

**Session**接口：负责保存、更新、删除、加载和查询对象，是线程不安全的，避免多个线程共享同一个session，是轻量级、一级缓存；

**Transaction**接口：管理事务；

**Query**和**Criteria**接口：执行数据库的查询。

### 43、什么是重量级？什么是轻量级？

**轻量级**是指它的创建和销毁不需要消耗太多的资源，意味着可以在程序中经常创建和销毁session的对象；**重量级**意味不能随意的创建和销毁它的实例，会占用很多的资源。

### 44、谈谈Spring的IOC和DI

首先想说说IoC（Inversion of Control，控制倒转）。这是spring的核心，贯穿始终。所谓IoC，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系。这是什么意思呢，举个简单的例子，我们是如何找女朋友的？常见的情况是，我们到处去看哪里有长得漂亮身材又好的mm，然后打听她们的兴趣爱好、qq号、电话号、ip号、iq号………，想办法认识她们，投其所好送其所要，然后嘿嘿……这个过程是复杂深奥的，我们必须自己设计和面对每个环节。传统的程序开发也是如此，在一个对象中，如果要使用另外的对象，就必须得到它（自己new一个，或者从JNDI中查询一个），使用完之后还要将对象销毁（比如Connection等），对象始终会和其他的接口或类藕合起来。

那么IoC是如何做的呢？有点像通过婚介找女朋友，在我和女朋友之间引入了一个第三者：婚姻介绍所。婚介管理了很多男男女女的资料，我可以向婚介提出一个列表，告诉它我想找个什么样的女朋友，比如长得像李嘉欣，身材像林熙雷，唱歌像周杰伦，速度像卡洛斯，技术像齐达内之类的，然后婚介就会按照我们的要求，提供一个mm，我们只需要去和她谈恋爱、结婚就行了。简单明了，如果婚介给我们的人选不符合要求，我们就会抛出异常。整个过程不再由我自己控制，而是有婚介这样一个类似容器的机构来控制。Spring所倡导的开发方式就是如此，所有的类都会在spring容器中登记，告诉spring你是个什么东西，你需要什么东西，然后spring会在系统运行到适当的时候，把你要的东西主动给你，同时也把你交给其他需要你的东西。所有的**类的创建、销毁都由 spring来控制**，也就是说**控制对象生存周期的不再是引用它的对象，而是spring**。对于某个具体的对象而言，以前是它控制其他对象，现在是所有对象都被spring控制，所以这叫控制反转。如果你还不明白的话，我决定放弃。

IoC的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI（Dependency Injection，依赖注入）来实现的。比如对象A需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了 spring我们就只需要告诉spring，A中需要一个Connection，至于这个Connection怎么构造，何时构造，A不需要知道。在系统运行时，spring会在适当的时候制造一个Connection，然后像打针一样，注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个Connection是由spring注入到A中的，依赖注入的名字就这么来的。那么DI是如何实现的呢？ Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，**spring就是通过反射来实现注入**的。

### 45、解释Spring的IoC? 几种注入依赖的方式？Spring的优点？

IOC你就认为他是一个生产和管理bean的容器就行了，原来需要在调用类中new的东西，现在都是有这个IOC容器进行产生，同时，要是产生的是单利的bean，他还可以给管理bean的生命周期！

**依赖注入的三种实现类型：接口注入、 Setter 注入和构造器注入。**

Spring的优点：主要是根据它的IOC和AOP体现的。我感觉他就是把我们以前用到的工厂模式和代理模式进行了一个封装。IOC主要是解决了代码的耦合性问题，而AOP是面向切面编程的最好解释！

### 46、什么是AOP？

Aspect Oriented Programming (面向方面编程)

OOP是面向对象编程,AOP是在OOP基础之上一种更高级的设计思想.

OOP和AOP之间也存在一些区别,OOP侧重于对象的提取和封装.

AOP侧重于方面组件,方面组件可以理解成封装了通用功能的组件,

方面组件可以通过配置方式灵活的切入到某一批目标对象方法上.

### 47、Spring的通知类型有哪些？

通知决定了方面组件功能在目标对象方法上执行的时机.Spring框架提供了以下5中类型通知.

**a.前置通知<aop:before>**

方面功能在目标方法之前调用.

**b.后置通知<aop:afterReturning>**

方面功能在目标方法之后调用.(如果目标方法抛出异常则不会执行方面功能)

**c.最终通知<aop:after>**

方面功能在目标方法之后调用.(目标方法有无异常都会执行方面功能)

**d.环绕通知<aop:around>**

方面功能在目标方法之前和之后调用.

**e.异常通知<aop:afterThrowing>**

方面功能在目标方法抛出异常之后调用.

### 48、springMVC的运行原理。

**SpringMVC运行原理**

1. 客户端请求提交到DispatcherServlet

2. 由DispatcherServlet控制器查询一个或多个HandlerMapping，找到处理请求的Controller

3. DispatcherServlet将请求提交到Controller

4. Controller调用业务逻辑处理后，返回ModelAndView

5. DispatcherServlet查询一个或多个ViewResoler视图解析器，找到ModelAndView指定的视图

6. 视图负责将结果显示到客户端

**DispatcherServlet是整个Spring MVC的核心。**它负责接收HTTP请求组织协调Spring MVC的各个组成部分。其主要工作有以下三项：

1. 截获符合特定格式的URL请求。

2. 初始化DispatcherServlet上下文对应的WebApplicationContext，并将其与业务层、持久化层的WebApplicationContext建立关联。

3. 初始化Spring MVC的各个组成组件，并装配到DispatcherServlet中。

### 49、 SpringMVC常用的注解有哪些？

**@RequestMapping**：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

**@RequestBody**：注解实现接收http请求的json数据，将json转换为java对象。

**@ResponseBody**：注解实现将conreoller方法返回对象转化为json对象响应给客户。

# 操作系统

## CentOS

### 1.yum

yum（全称为 Yellow dog Updater, Modified）是一个在Fedora和RedHat以及SUSE中的Shell前端软件包管理器。

基於RPM包管理，能够从指定的服务器自动下载RPM包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软体包，

无须繁琐地一次次下载、安装。yum提供了查找、安装、删除某一个、一组甚至全部软件包的命令，而且命令简洁而又好记。

yum的命令形式一般是如下：yum [options] [command] [package ...]

其中的[options]是可选的，选项包括-h（帮助），-y（当安装过程提示选择全部为"yes"），-q（不显示安装的过程）等等。[command]为所要进行的操作，[package ...]是操作的对象。

Redhat和Fedora的软件安装命令是rpm，但是用rpm安装软件最大的麻烦就是需要手动寻找安装该软件所需要的一系列依赖关系，超级麻烦不说，要是软件不用了需要卸载的话由于卸载掉了某个依赖关系而导致其他的软件不能用是非常恼人的。令人高兴的是，Fedora终于推出了类似于 ubuntu中的apt的命令yum，令Fedora的软件安装变得简单容易。

yum的宗旨是自动化地升级，安装/移除rpm包，收集rpm包的相关信息，检查依赖性并自动提示用户解决。yum的关键之处是要有可靠的repository，顾名思义，这是软件的仓库，它可以是http或ftp站点， 也可以是本地软件池，但必须包含rpm的header，header包括了rpm包的各种信息，包括描述，功能，提供的文件，依赖性等.正是收集了这些 header并加以分析，才能自动化地完成余下的任务。yum的一切配置信息都储存在一个叫**yum.conf**的配置文件中，通常位于**/etc目录下**，这是整个yum系统的重中之重。

### 2.rpm

RPM[1] 是RPM Package Manager（RPM软件包管理器（原Red Hat Package Manager，现在是一个递归缩写））的缩写，这一文件格式名称虽然打上了RedHat的标志，但是其原始设计理念是开放式的，现在包括OpenLinux、S.u.S.E.以及Turbo Linux等Linux的分发版本都有采用，可以算是公认的行业标准了。

### 3.yum与rpm

rpm是由红帽公司开发的软件包管理方式，使用rpm我们可以方便的进行软件的安装、查询、卸载、升级等工作。但是rpm软件包之间的依赖性问题往往会很繁琐,尤其是软件由多个rpm包组成时。

Yum（全称为 Yellow dog Updater, Modified）是一个在Fedora和RedHat以及SUSE中的Shell前端软件包管理器。基於RPM包管理，能够从指定的服务器自动下载RPM包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软体包，无须繁琐地一次次下载、安装。

### 4.挂载点及各目录

**挂载点：**实际上就是linux中的磁盘文件系统的入口目录，类似于windows中的用来访问不同分区的C:、D:、E:等盘符。

其实winxp也支持将一个磁盘分区挂在一个文件夹下面，只是我们C:、D:这样的盘符操作用惯了，一般没有将分区挂到文件夹。

**/ 根目录：**唯一必须挂载的目录。不要有任何的犹豫，选一个分区，挂载它！（在绝大多数情况下，有10G的容量应该是够用了。当然了，很多东西都是多多益善的)

**/boot：**它包含了操作系统的内核和在启动系统过程中所要用到的文件，建这个分区是有必要的，启动分区，即使主要的根分区出现了问题，计算机依然能够启动。这个分区的大小约在60MB—120MB之间。

**/home：**是用户的home目录所在地，这个分区的大小取决于有多少用户。如果是多用户共同使用一台电脑的话，这个分区是完全有必要的，况且根用户也可以很好地控制普通用户使用计算机，如对用户或者用户组实行硬盘限量使用，限制普通用户访问哪些文件等。

**/tmp：**用来存放临时文件。这对于多用户系统或者网络服务器来说是有必要的。这样即使程序运行时生成大量的临时文件，或者用户对系统进行了错误的操作，文件系统的其它部分仍然是安全的。因为文件系统的这一部分仍然还承受着读写操作，所以它通常会比其它的部分更快地发生问题。

**/usr：**应用程序目录。大部分的软件都安装在这里。就像是Windows里面的Program Files。

**/var：**日志文件，经常会变动，硬盘读写率高的文件放在此中。

**/srv：**一些服务启动之后，这些服务所需要取用的资料目录。在文件系统这一环节中，建议您选择： ReiserFS和Ext3

**/opt：**存放可选的安装文件，个人一般把自己下载的软件存在里面，比如永中Office、LumaQQ等等。

**Swap：**交换分区，可能不是必须的，但是有这个交换分区可以当内存不够用时防止内存溢出(memory out).不过按照传统，并且照顾到您的安全感，还是挂载它吧。它的容量只要大于您的物理内存就可以了，如果超过了您物理内存两倍的容量，那绝对是一种浪费。

**/var/log：**系统日志记录分区。一般多用户系统或者网络服务器要建立这个分区，因为设立了这个分区，即使系统的日志文件出现了问题，他们也不会影响到操作系统的主分区。

**/bin：**存放程序，里面的程序可以直接通过命令调用，而不需要进入程序所在的文件夹。

**/dev：**存放所有设备文件，如fd0为软盘设备，cdrom为光盘设备。

**/etc：**是Linux下的配置文件的老家， 很多管理和使用的修改都在这里。

**/sbin：**和bin目录相似，存放标准系统管理文件，一般只有超级用户才能使用。

### 5.磁盘管理

步骤是 分区(parted) ---> 格式化 (mkfs) ----->挂载(mount)

分区有很多工具。比如说fdisk，parted等等吧，这里需要知道两个分区类型。一种是传统老旧的类型叫MBR，一种是现在常用的类型GPT。

简单说下区别：MBR最多只有4个主分区，在逻辑分区上也比较局限（主分区个数有限，如果再想分区，就要抽出来一个扩展分区，在扩展分区里添加逻辑分区）。而GPT基本上分区数足够了，分区的存储大小也足够了。

fdisk这个分区工具只支持传统的MBR分区类型，parted分区工具这两种类型都支持。

分区以后，要创建文件系统，mkfs（make file system）就是格式化创建文件系统，这个都是针对某一块磁盘的分区进行的格式化。

ext4表示一种linux支持的文件系统。

比如： mkfs.ext4 /dev/sdf2 /dev/sdf2 这个就是设备名称。

你比如sda、sdb、sdc 等等，这就是第几块磁盘的意思，a就是第一块硬盘，b就是第二块。比如sda1 sda2 sda3 这个就表示是第一块硬盘里，sda1第一块分区，sda2第二块分区。

格式化以后，就要挂载文件系统，就是把一个分区设备，挂到一个挂载点上。（挂载点说白了就是一个存在的目录）

命令格式就是 # mount /dev/sdb1 /home/bala

（这里再说个知识点，挂载命令里/dev/sdb1这个设备名，现在推荐的做法都是用uuid唯一标识符了，因为设备名称会改变，和内核发现设备的顺序有关）

在给一个分区创建文件系统时，会分配它一个唯一的uuid。

那么怎么样知道这个设备名，对应那个uuid呢？这里有个命令

# blkid #就是block id 可以看到哪个设备名对应哪个uuid 。然后挂载的时候，使用uuid，而不是设备名了哈哈

文件系统这块，必须要提及文件系统表/etc/fstab，系统启动的时候会读取这个文件，自动挂载里边的记录的设备。

比如查看下：# cat /etc/fstab

UUID=80c0be4e-fed2-4542-9259-d667b6ae34eb / ext4 defaults 1 1

UUID=9ae2f239-db3e-4aae-8cef-85d05cf9d61d /boot ext4 defaults 1 2

UUID=4bc97876-6add-4768-96ca-13876451a570 /home ext4 defaults 1 2

UUID=afddbec2-c7b2-4bde-97d3-fc4a4deb380e swap swap defaults 0 0

tmpfs /dev/shm tmpfs defaults 0 0

devpts /dev/pts devpts gid=5,mode=620 0 0

sysfs /sys sysfs defaults 0 0

proc /proc proc defaults 0 0

表里各个项意思是：设备名 挂载点 文件系统类型 选项 balala

发现以上文件系统类型项中，出现比如swap tmpfs 等等。这些是特殊的文件系统，并不仅仅是物理媒介，他们是作为系统接口来使用的文件系统。

主要有proc（挂载在/proc）、sysfs（挂载在/sys）、tmpfs

还要提及swap。说白了就是内存不咋够用了，利用一些磁盘上的空间，进行数据交换的一块空间。可以使用free命令查看swap使用情况。

可以使用磁盘分区当交换空间，也可以使用一个文件当交换空间。具体命令就不列出了。

还有个比较重要的知识点就是inode节点。这个有兴趣的话，可以自己去查下资料。

文件系统这块总结几个常用的命令吧，很多命令里有很多选项参数，这里就不一一列举了。

# parted

# fdisk -l

# mkfs

# blkid

# mount

# umount

# sync # 同步，强制缓冲区里的东西，写到磁盘里。

# df # disk free磁盘使用情况

# du # 目录下，各个文件占用存储大小

# free # 查看内存和交换分区swap使用情况

# ls -i # inode节点

# vi /etc/fstab # 查看文件系统表

## 二、鸟哥的私房菜基础版

## 三、试题

### 1、Linux系统下你关注过哪些内核参数，说说你知道的。

### 2、Linux下IO模型有几种，各自的含义是什么。

（1）、BIO（blocking IO）：同步阻塞 I/O

能够及时返回数据，无延迟

性能下降

（2）、NIO（nonblocking IO）：同步非阻塞 I/O

拷贝数据的整个过程，进程仍然是阻塞的

需要不断询问数据是否准备好了

能够在等待任务完成的过程中处理其他事件

由于需要轮询，所以延迟会增加

（3）、多路复用IO（ IO multiplexing）

select/poll调用后会阻塞进程，但可以同时阻塞多个IO事件操作（文件描述符），有数据可读或可写（就绪事件），就通知用户进程。

select 需要每次注册事件（轮询），而epoll不需要每次注册事件（没有轮询，回调函数）

IO多路复用阻塞在select/epoll的系统调用之上的，而真正的IO系统调用如recvfrom是非阻塞的。

（4）、信号驱动I/O（ signal driven IO）

（5）、异步 I/O（asynchronous IO）

### 3、select、epoll和poll有什么区别。

### 4、平时用到哪些Linux命令。

### 5、用一行命令查看文件的最后五行。

tail -n 5 filename

### 6、用一行命令输出正在运行的java进程。

ps -ef |grep java

### 7、介绍下你理解的操作系统中线程切换过程。System.err

### 8、进程和线程的区别。

### 9、top 命令之后有哪些内容，有什么作用。

### 10、线上CPU爆高，请问你如何找到问题所在。

### 11、软连接（符号连接）

linux下的软链接类似于windows下的快捷方式

建立软连接：ln -s /home/gamestat    /gamestat

删除软连接：rm -rf  /gamestat

### 硬连接

硬链接只能引用同一文件系统中的文件。它引用的是文件在文件系统中的物理索引(也称为 inode)。当您移动或删除原始文件时，硬链接不会被破坏，因为它所引用的是文件的物理数据而不是文件在文件结构中的位置。硬链接的文件不需要用户有访问原始文件的权限，也不会显示原始文件的位置，这样有助于文件的安全。如果您删除的文件有相应的硬链接，那么这个文件依然会保留，直到所有对它的引用都被删除。

1、修改源文件或者目标文件, 对应另外一个文件也会发生相应修改.

2、删除源文件或者目标文件, 对另外一个文件没有影响.

3、硬链接文件不占用存储空间.

4、不能对目录文件进行创建硬链接操作.

5、硬链接文件不能跨文件系统.

创建硬链接：ln test2.sh test\_l.sh。

### vi命令

输入gg跳到文件开头，输入/ 内容后回车，n键向下搜索。

输入G跳到文件结尾，输入？内容后回车，n键向上搜索。

# 五、网络

## 一、网络分层

### 1、几种分层方式

**OSI分层（7层）**

物理层、数据链路层、网络层、运输层、会话层、表示层、应用层

**TCP/IP分层（4层）**

网络接口层、网络层、运输层、应用层

**五层协议（5层）**

物理层、数据链路层、网络层、运输层、应用层

### 2、五层结构的概述

**应用层**：

描述：通过应用进程间的交互来完成特定网络应用

数据：报文

协议：HTTP, SMTP(邮件), FTP(文件传送)

**运输层**：

描述：向两个主机进程之间的通信提供通用的数据传输服务。

数据：TCP:报文段，UDP:用户数据报

协议：TCP, UDP

**网络层**：

描述：为分组交换网上的不同主机提供通信服务

数据：包或IP数据报

协议：IP

**数据链路层**：

数据：帧

**物理层**：

数据：比特

### 3、七层结构概述：

**应用层**

功能：用户接口

协议：FTP、DNS、Telnet、SNMP、SMTP、HTTP、WWW、NFS

**表示层**

功能：数据表示；加密与解密、数据的压缩与解压缩、图像编码与解码等特殊处理过程

常见：URL加密、口令加密、图片编解码等 JPEG、MPEG、ASCII

**会话层**

功能：保证不同应用程序的数据独立；建立、管理和终止会话。

常见：服务器验证用户登录、断点续传

协议：NFS、SQL、NetBIOS、RPC

**运输层**

功能：可靠或不可靠数据传输；数据重传前的错误纠正。

常见：进程、端口（socket）

协议：TCP、UDP、SPX

**网络层**

数据：(数据包Packet)

功能：负责数据包从源到宿的传递和网际互连

设备：路由器、多层交换机、防火墙等

协议：IP、ICMP、ARP、PARP、OSPF、IPX、RIP、IGRP等

**数据链路层**

数据：(帧Frame)

功能：将比特组合成字节，进而组合成帧；用MAC地址访问介质；错误可以被发现但不能被纠正。

设备：网卡、网桥、二层交换机等

协议：PPP、FR、HDLC、VLAN、MAC等

**物理层**：

数据：(比特Bit)

功能：设备间接收或发送比特流；

设备：中继器、网线、集线器、HUB等

协议：RJ45、CLOCK、IEEE802.3等

### 4、应用层

#### 域名系统DNS

例：某用户通过主机A浏览西安交大的主页 www.xjtu.edu.cn

1. A向本地域名服务器DNS查询

2. 如果DNS上有www.xjtu.edu.cn的记录，就立即返回IP地址给主机A

3. 如果DNS上没有该域名记录，则DNS向根域名服务器发出查询请求

4. 根域名服务器把负责cn域的顶级域名服务器B的IP地址告诉DNS

5. DNS向B查询获得二级域名服务器C的IP地址，最终迭代查询到www.xjtu.edu.cn的ip直接返回DNS。

#### http请求解析：

GET /search?hl=zh-CN&source=hp&q=domety&aq=f&oq= HTTP/1.1

//请求方法 空格 url 协议 回车符 换行符

Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-powerpoint,

application/msword, application/x-silverlight, application/x-shockwave-flash, \*/\*

Referer: <a href="http://www.google.cn/">http://www.google.cn/</a>

Accept-Language: zh-cn

Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; TheWorld)

Host: <a href="http://www.google.cn">www.google.cn</a>

Connection: Keep-Alive

Cookie: PREF=ID=80a06da87be9ae3c:U=f7167333e2c3b714:NW=1:TM=1261551909:LM=1261551917:S=ybYcq2wpfefs4V9g;

NID=31=ojj8d-IygaEtSxLgaJmqSjVhCspkviJrB6omjamNrSm8lZhKy\_yMfO2M4QMRKcH1g0iQv9u-2hfBW7bUFwVh7pGaRUb0RnHcJU37y-

FxlRugatx63JLv7CWMD6UB\_O\_r

**User-Agent**：产生请求的浏览器类型;

**Accept**：客户端可识别的响应内容类型列表;星号 “ \* ” 用于按范围将类型分组，用 “ / ” 指示可接受全部类型，用“ type/\* ”指示可接受 type 类型的所有子类型;

**Accept-Language**：客户端可接受的自然语言;

**Accept-Encoding**：客户端可接受的编码压缩格式;

**Accept-Charset**：可接受的应答的字符集;

**Host**：请求的主机名，允许多个域名同处一个IP 地址，即虚拟主机;

**connection**：连接方式(close 或 keepalive);

**Cookie**：存储于客户端扩展字段，向同一域名的服务端发送属于该域的cookie;

#### Http1.1与Http1.0的区别

http1.0使用非持久连接（短连接），而http1.1默认是持久连接（长连接），当然也可以配置成非持久连接

### 5、运输层

使用UDP和TCP协议的各种应用和应用层协议

应用 应用层协议 运输层协议

名字转换 DNS(域名系统) UDP

文件传送 TFTP(简单文件传送协议) UDP-------------TFTP

路由器选择协议 RIP(路由信息协议) UDP

IP地址配置 DHCP(动态主机配置协议) UDP

网络管理 SNMP(简单网络管理协议) UDP

远程服务器 NFS(网络文件系统) UDP

多播 IGMP(网际组管理协议) UDP

电子邮件 SMTP(简单邮件传送协议) TCP

远程终端 TELNET(远程终端协议) TCP

万维网 HTTP(超文本传送协议) TCP

文件传送 FTP(文件传送协议) TCP ------------FTP

UDP

特点:

无连接的（发送数据之前不需要建立连接，因此减少了开销和发送数据之前的时延）

尽最大努力交付（不保证可靠支付，因此主机不需要维持复杂的连接状态表）

面向报文的（UDP对应用层交下来的报文，添加完首部后就直接交付IP层。如果太长就会分片）

UDP没有拥塞控制

UDP支持一对一、一对多、多对一和多对多的交互通信

UDP的首部开销小（只有8个字节，TCP有20个字节）

报文：

源端口：2字节 = 16bit = 0 ~ 65535

目的端口：2字节

长度：2字节

检验和：2字节

TCP

特点：

面向连接的运输层协议。

点对点（一对一）通信。

可靠交付。

全双工通信（TCP连接的两端都设有发送缓存和接收缓存，用来临时存放双向通信的数据）。

面向字节流。

TCP与UDP在发送报文时所采用的方式完全不同。TCP具体发送的报文由接收方给出的窗口值和当前网络拥塞的程度来决定一个报文段包含多少字节。而UDP发送的报文长度由应用进程给出

TCP可靠传输工作原理：

TCP连接的端点叫做套接字(socket)或插口。套接字socket = (IP地址：端口号)

停止等待协议：

1.每发送完一个分组就设置一个超时计时器。

2. 必须暂时保存已发送的分组的副本

3. 分组和确认分组都必须编号

4. 超市计时器设置的重传时间比数据在分组传输的平均时间更长一些

5.如果接收方接收到数据发送确认没有被发送方接收到，那么发送方超时后会重新发送分组，并且接收方收到重复的分组会丢弃并重传确认。 如果接收方收到的确认是已经接受过的，那么会无视这个确认。

6.停止等待协议（自动重传ARQ）虽然简单，但是信道利用率低。

连续ARQ协议和滑动窗口协议:

1.先不计是否收到确认发送窗口个分组

2.每收到一个确认窗口向前滑动一格。

TCP报文格式：

1.源端口和目的端口 各占2字节

2.序号 4字节

3.确认号 4字节期望收到对方下一个报文的第一个数据字节的序号

4.数据偏移 4位

5.保留 6字节

6.紧急URG 当URG=1表示紧急指针有效

7.确认ACK

8.推送PSH

9.复位RST 当RST = 1时，释放连接并重新建立连接

10.同步SYN 当SYN = 1 ACK = 0时，表明这是一个连接请求报文段。(第一次握手)

11.终止FIN FIN = 1，请求释放连接。 （第一次挥手）

12.窗口

13.检验和

14.紧急指针

15.选项

TCP三次握手：

1.客户端TCP向服务端TCP发送一个特殊的TCP报文段，不包含应用层数据，报文中SYN=1，设置一个初始号client\_isn,记录在报文段的序列号seq中。

2.SYN报文段到达服务器后，为该TCP链接分配缓存和变量，并向客户端发送允许链接的报文段。其中，SYN = 1， ACK = client\_isn+1，seq = server\_isn;

3.客户端收到允许连接的报文后，客户端也给连接分配缓存和变量，客户端向服务端发送一个报文段，其中ACK = server\_isn+1，SYN = 0，并且由于连接已经建立所以现在可以携带应用层数据。

TCP四次挥手：

1. 客户端发送连接释放报文段，报文中FIN = 1, seq = u;

2. 服务端接收到连接释放报文后发出确认报文，其中ACK = 1; seq = v; ack = u + 1;

3. 服务端在发送完数据后，发送连接释放报文FIN = 1, seq = w, ack = u + 1;并停止向客户端发送数据。

4. 客户端收到连接释放报文后，发送确认报文， ACK = 1; seq = u + 1; ack = w + 1;并且进入等待2MSL，防止服务端没有接收到确认报文，重传报文。并且使连接产生的报文都消失。

TCP协议的连接是全双工连接，一个TCP连接存在双向的读写通道。

简单说来是 “先关读，后关写”，一共需要四个阶段。以客户机发起关闭连接为例：

1. 服务器读通道关闭

2. 客户机写通道关闭

3. 客户机读通道关闭

4. 服务器写通道关闭

TCP拥塞控制

拥塞控制和流量控制的区别

流量控制针对的是点对点之间的（发送方和接收方）之间的速度匹配服务，因为接收方的应用程序读取的速度不一定很迅速，而接收方的缓存是有限的，就需要避免发送的速度过快而导致的问题。拥塞控制是由于网络中的路由和链路传输速度限制，要避免网络的过载和进行的控制。

拥塞控制算法

拥塞控制算法主要包含了三个部分：慢启动、拥塞避免和快速回复

**慢启动**

慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。一般一开始为1个MSS，之后翻倍这样来增加，呈指数增长。其中

1、慢启动过程有一个阈值ssthresh，一旦到达阈值就进入拥塞避免模式。这是第一种离开结束慢启动的方式

2、如果收到了一个丢包提示，就将cwnd设为1并且重新开始慢启动过程，这时要把阈值ssthresh设为当前cwnd值的一半。

3、如果收到了三次冗余的ACK，就执行一次快速重传并且进入快速恢复状态，这是最后一种结束慢启动的过程。

**拥塞避免**

进入拥塞避免说明cwnd值大约是上一次遇到拥塞是的一半，这时候不能翻倍，而是将cwnd的值每次增加一个MSS。

结束的过程有两种可能：

1、当出现超时时，将cwnd值设为1个MSS，并且将ssthresh阈值设为当前cwnd值的一半。

2、当收到三个冗余ACK时，将ssthresh阈值设为当前cwnd值的一半，并且将cwnd值设为当前cwnd值的一半加3， 即ssthresh阈值加3，并且进入快速恢复状态。

**快速恢复**

快速恢复就是指进入快速恢复前的一系列操作，即将ssthresh阈值设为当前cwnd值的一半，并且将cwnd值设为当前cwnd值的一半加3，即ssthresh阈值加3，之后进入拥塞避免状态，即每次cwnd的值加1个MSS。

### 6、网络层

协议：

地址解析协议 ARP

网际控制报文协议 ICMP

网际组管理协议 IGMP

IP：

IP地址分类：

A类:1.0.0.0~126.255.255.255,默认子网掩码/8,即255.0.0.0(其中127.0.0.0~127.255.255.255为环回地址,用于本地环回测试等用途)；

B类:128.0.0.0~191.255.255.255,默认子网掩码/16,即255.255.0.0；

C类:192.0.0.0~223.255.255.255,默认子网掩码/24,即255.255.255.0；

D类:224.0.0.0~239.255.255.255,一般于用组播

E类:240.0.0.0~255.255.255.255(其中255.255.255.255为全网广播地址),E类地址一般用于研究用途

## 二、试题

### 1、http1.0和http1.1有什么区别。

HTTP 1.0主要有以下几点变化：

请求和相应可以由于多行首部字段构成

响应对象前面添加了一个响应状态行

响应对象不局限于超文本

服务器与客户端之间的连接在每次请求之后都会关闭

实现了Expires等传输内容的缓存控制

内容编码Accept-Encoding、字符集Accept-Charset等协商内容的支持

这时候开始有了请求及返回首部的概念，开始传输不限于文本（其他二进制内容）

HTTP 1.1加入了很多重要的性能优化：持久连接、分块编码传输、字节范围请求、增强的缓存机制、传输编码及请求管道。

### 2、TCP三次握手和四次挥手的流程，为什么断开连接要4次,如果握手只有两次，会出现什么。

**第一次握手(SYN=1, seq=x)**:

客户端发送一个 TCP 的 SYN 标志位置1的包，指明客户端打算连接的服务器的端口，以及初始序号 X,保存在包头的序列号(Sequence Number)字段里。发送完毕后，客户端进入 SYN\_SEND 状态。

**第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1)**:

服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为1。服务器端选择自己 ISN 序列号，放到 Seq 域里，同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加1，即X+1。发送完毕后，服务器端进入 SYN\_RCVD 状态。

**第三次握手(ACK=1，ACKnum=y+1)**

客户端再次发送确认包(ACK)，SYN 标志位为0，ACK 标志位为1，并且把服务器发来 ACK 的序号字段+1，放在确定字段中发送给对方，并且在数据段放写ISN的+1发送完毕后，客户端进入 ESTABLISHED 状态，当服务器端接收到这个包时，也进入 ESTABLISHED 状态，TCP 握手结束。

在 socket 编程中，客户端执行 connect() 时。将触发三次握手

TCP 提供一种面向连接的、可靠的字节流服务在一个 TCP 连接中，仅有两方进行彼此通信。广播和多播不能用于 TCP

TCP 使用校验和，确认和重传机制来保证可靠传输

TCP 给数据分节进行排序，并使用累积确认保证数据的顺序不变和非重复

TCP 使用滑动窗口机制来实现流量控制，通过动态改变窗口的大小进行拥塞控制

**第一次挥手(FIN=1，seq=x)**

假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为1的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。发送完毕后，客户端进入 FIN\_WAIT\_1 状态。

**第二次挥手(ACK=1，ACKnum=x+1)**

服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。发送完毕后，服务器端进入 CLOSE\_WAIT 状态，客户端接收到这个确认包之后，进入 FIN\_WAIT\_2 状态，等待服务器端关闭连接。

**第三次挥手(FIN=1，seq=y)**

服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为1。发送完毕后，服务器端进入 LAST\_ACK 状态，等待来自客户端的最后一个ACK。

**第四次挥手(ACK=1，ACKnum=y+1)**

客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 TIME\_WAIT状态，等待可能出现的要求重传的 ACK 包。服务器端接收到这个确认包之后，关闭连接，进入 CLOSED 状态。客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK ，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 CLOSED 状态。两次后会重传直到超时。如果多了会有大量半链接阻塞队列。

### 3、TIME\_WAIT和CLOSE\_WAIT的区别。

TIME\_WAIT状态就是用来重发可能丢失的ACK报文。

TIME\_WAIT 表示主动关闭，CLOSE\_WAIT 表示被动关闭。

### 4、说说你知道的几种HTTP响应码，比如200, 302, 404。

**1xx**：信息，请求收到，继续处理

**2xx**：成功，行为被成功地接受、理解和采纳

**3xx**：重定向，为了完成请求，必须进一步执行的动作

**4xx**：客户端错误，请求包含语法错误或者请求无法实现

**5xx**：服务器错误，服务器不能实现一种明显无效的请求

200 ok 一切正常

302 Moved Temporatily 文件临时移出

404 not found

### 5、当你用浏览器打开一个链接（如：http://www.javastack.cn）的时候，计算机做了哪些工作步骤。

Dns解析–>端口分析–>tcp请求–>服务器处理请求–>服务器响应–>浏览器解析—>链接关闭

### 6、TCP/IP如何保证可靠性，说说TCP头的结构。

typedef struct \_tcp\_hdr

{

unsigned short src\_port; //源端口号

unsigned short dst\_port; //目的端口号

unsigned int seq\_no; //序列号

unsigned int ack\_no; //确认号

#if LITTLE\_ENDIAN

unsigned char reserved\_1:4; //保留6位中的4位首部长度

unsigned char thl:4; //tcp头部长度

unsigned char flag:6; //6位标志

unsigned char reseverd\_2:2; //保留6位中的2位

#else

unsigned char thl:4; //tcp头部长度

unsigned char reserved\_1:4; //保留6位中的4位首部长度

unsigned char reseverd\_2:2; //保留6位中的2位

unsigned char flag:6; //6位标志

#endif

unsigned short wnd\_size; //16位窗口大小

unsigned short chk\_sum; //16位TCP检验和

unsigned short urgt\_p; //16为紧急指针

}tcp\_hdr;

### 7、如何避免浏览器缓存。

无法被浏览器缓存的请求：

HTTP信息头中包含Cache-Control:no-cache，pragma:no-cache，或Cache-Control:max-age=0等告诉浏览器不用缓存的请求需要根据Cookie，认证信息等决定输入内容的动态请求是不能被缓存的经过HTTPS安全加密的请求（有人也经过测试发现，ie其实在头部加入Cache-Control：max-age信息，firefox在头部加入Cache-Control:Public之后，能够对HTTPS的资源进行缓存，参考《HTTPS的七个误解》）

POST请求无法被缓存

HTTP响应头中不包含Last-Modified/Etag，也不包含Cache-Control/Expires的请求无法被缓存

### 8、如何理解HTTP协议的无状态性。

HTTP 是一个属于应用层的面向对象的协议，HTTP协议一共有五大特点：

1、支持客户/服务器模式；

2、简单快速；

3、灵活；

4、无连接；

无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。请求时建连接、请求完释放连接，以尽快将资源释放出来服务其他客户端。随着时间的推移，网页变得越来越复杂，里面可能嵌入了很多图片，这时候每次访问图片都需要建立一次 TCP 连接就显得很低效。后来，Keep-Alive被提出用来解决这效率低的问题。

Keep-Alive 功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive 功能避免了建立或者重新建立连接

5、无状态。

无状态是指协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。即我们给服务器发送 HTTP 请求之后，服务器根据请求，会给我们发送数据过来，但是，发送完，不会记录任何信息。HTTP 是一个无状态协议，这意味着每个请求都是独立的，Keep-Alive 没能改变这个结果。

HTTP协议这种特性有优点也有缺点，优点在于解放了服务器，每一次请求“点到为止”不会造成不必要连接占用，缺点在于每次请求会传输大量重复的内容信息。客户端与服务器进行动态交互的 Web 应用程序出现之后，HTTP 无状态的特性严重阻碍了这些应用程序的实现，毕竟交互是需要承前启后的，简单的购物车程序也要知道用户到底在之前选择了什么商品。于是，两种用于保持 HTTP 连接状态的技术就应运而生了，一个是 Cookie，而另一个则是 Session。

Cookie可以保持登录信息到用户下次与服务器的会话，换句话说，下次访问同一网站时，用户会发现不必输入用户名和密码就已经登录了（当然，不排除用户手工删除Cookie）。而还有一些Cookie在用户退出会话的时候就被删除了，这样可以有效保护个人隐私。

Cookies最典型的应用是判定注册用户是否已经登录网站，用户可能会得到提示，是否在下一次进入此网站时保留用户信息以便简化登录手续，这些都是Cookies的功用。另一个重要应用场合是“购物车”之类处理。用户可能会在一段时间内在同一家网站的不同页面中选择不同的商品，这些信息都会写入 Cookies，以便在最后付款时提取信息。

与 Cookie 相对的一个解决方案是 Session，它是通过服务器来保持状态的。当客户端访问服务器时，服务器根据需求设置 Session，将会话信息保存在服务器上，同时将标示 Session 的 SessionId 传递给客户端浏览器，**浏览器将这个 SessionId 保存在内存中**，我们称之为无过期时间的 Cookie。浏览器关闭后，这个 Cookie 就会被清掉，它不会存在于用户的 Cookie 临时文件。以后浏览器每次请求都会额外加上这个参数值，服务器会根据这个 SessionId，就能取得客户端的数据信息。

如果客户端浏览器意外关闭，服务器保存的 Session 数据不是立即释放，此时数据还会存在，只要我们知道那个 SessionId，就可以继续通过请求获得此 Session 的信息，因为此时后台的 Session 还存在，当然我们可以设置一个 Session 超时时间，一旦超过规定时间没有客户端请求时，服务器就会清除对应 SessionId 的 Session 信息。

### 9、简述Http请求get和post的区别以及数据包格式。

### 10、HTTP有哪些method

HTTP 0.9 这个版本只有GET方法

HTTP 1.0 这个版本有GET HEAD POST这三个方法

HTTP 1.1 这个版本是当前版本，包含GET HEAD POST OPTIONS PUT DELETE TRACE CONNECT这8个方法

### 11、简述HTTP请求的报文格式。

一个http请求报文由四部分组成

**请求行（request line)**：

请求行由请求方法字段、URL字段和HTTP协议版本字段，组成，它们用空格分隔，

**消息头部（header)：**

请求头部由key/value键值对组成，每行一对，key和value用冒号":"分隔，请求头部通知服务器有关于client端的请求信息，典型的请求头：

User-Agent：产生请求的浏览器类型

Accept：client端可识别的内容类型列表

Host：请求的主机名，允许多个域名同处一个ip地址，即虚拟主机

空行：最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器请求头结束。对于一个完整的http请求来说空行是必须的，否则服务器会任务本次请求的数据尚未完全发送到server，处于等待状态

**请求正文：**

请求数据不在GET方法中使用，而是在POST中使用。POST方法适用于需要client填写表单的场合，与请求数据相关的最常用的请求头是Content-Type 和Content-Length

### 12、HTTP的长连接是什么意思。

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码： **Connection:keep-alive**

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。

### 13、HTTPS的加密方式是什么，讲讲整个加密解密流程。

HTTPS其实是有两部分组成：HTTP + SSL / TLS，也就是在HTTP上又加了一层处理加密信息的模块。服务端和客户端的信息传输都会通过TLS进行加密，所以传输的数据都是加密后的数据。

1. 客户端发起HTTPS请求

这个没什么好说的，就是用户在浏览器里输入一个https网址，然后连接到server的443端口。

2. 服务端的配置

采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面(startssl就是个不错的选择，有1年的免费服务)。这套证书其实就是一对公钥和私钥。如果对公钥和私钥不太理解，可以想象成一把钥匙和一个锁头，只是全世界只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

3. 传送证书

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

4. 客户端解析证书

这部分工作是有客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随即值。然后用证书对该随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

5. 传送加密信息

这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

6. 服务端解密信息

服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

7. 传输加密后的信息

这部分信息是服务段用私钥加密后的信息，可以在客户端被还原

8. 客户端解密信息

客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

自制证书：

安装JDK，使用bin下keytool.exe

cmd运行：keytool -genkey -alias tomcat -storetype PKCS12 -keyalg RSA -keysize 2048 -keystore d:\mykeystore\keystore.p12 -validity 3650 -ext san=ip:192.168.100.132 -dname "CN=garyyan, OU=mycompany, O=mycompany, L=gd, ST=gd, C=china"

此命令中间只需要输入密码，就能生成keystore，假设密码是：123456

其中：

1）keystore可理解为一个数据库，可以存很多个组数据。

每组数据主要包含下面两种数据：

a:密钥实体（Key entity）——密钥（secret key）又或者是私钥和配对公钥（采用非对称加密）

b:可信任的证书实体（trusted certificate entries）——只包含公钥

2）-keystore d:\mykeystore\keystore.p12，指定在d:\mykeystore（先要手动创建此文件夹），生成keystore:keystore.p12

3）-alias tomcat，为其指明在keystore中的唯一的别名：tomcat ，因为keystore中可能还存有其它的别名，如：tomcat 2

4）-storetype PKCS12指明密钥仓库类型是PKCS12

5）-keyalg RSA，指定加密算法,本例中的采用通用的RAS加密算法

6）-keysize 2048指定密钥的长度为2048

7）-validity 3650 指定证书的有效期为3650天

8）-ext san=ip:192.168.100.132请根据你的服务器的IP地址设置，如果不进行设置，客户端在访问的时候可能会报错

9）-dname “CN=garyyan, OU=mycompany,O=mycompany,L=gd, ST=gd, C=china

(单位的两字母国家代码)”，我在测试的过程中发现随便填就行

我们用的操作系统（windows, linux, unix ,android ios等）都预置了很多信任的根证书，比如我的windows中就包含VeriSign的根证书，

国内许多公司的做法：自己做根证书CA（自己充当类似于VeriSign的角色），然后让用户下载安装根CA（当然了，其中只含有公钥）到机器中，12306就是这样干的（SRCA就是12306的根证书），然后再自己给自己颁发服务器证书，这样用户机器上也有他的CA，服务器发来的服务器证书也是这本CA颁发的，当然也顺利通过了。

### 14、Http和https的三次握手有什么区别。

TPC/IP协议是传输层协议，主要解决数据如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据

### 15、什么是分块传送。

### 16、Session和cookie的区别

# 六、架构设计与分布式

## 一、架构师之路

### 一、架构师审美观

#### 1、设计模式

##### (1).Proxy代理模式

##### (2).Factory工厂模式

##### (3).Singleton单例模式

##### (4).Delegate委派模式

##### (5).Strategy策略模式

##### (6).Prototype原型模式

##### (7).Template模板模式

#### 2、Spring5

##### (1).IOC容器设计原理及高级特性

##### (2).AOP设计原理

##### (3).FactoryBean与BeanFactory

##### (4).Spring事务处理机制

##### (5).基于SpringJdbc手写ORM框架

##### (6).SpringMvc9大组件

##### (7).手写实现SpringMVC框架

##### (8).SpringMVC与Struts2对比分析

##### (9).Spring5新特性

#### 3、MyBatis

##### (1).代码自动生成器

##### (2).MyBatis关联查询、嵌套查询

##### (3).缓存使用场景及选择策略

##### (4).Spring集成SqlSession与Mapper MyBatis事务

##### (5).分析MyBatis的动态代理的真正实现

##### (6).手写Mini版的MyBatis

### 二、DevOps（架构师工具箱）

#### 1、Maven

##### (1).生成可执行jar，理解scope生成最精确的jar

##### (2).类冲突、包依赖NoClassDefFoundError问题的定位与解决

##### (3).全面理解Maven的Lifecycle、Phase、Goal

##### (4).架构师必备之Maven生成Archetype

##### (5).Maven流行插件实战、手写自己的插件

##### (6).Nexus的使用、上传、配置

##### (7).对比Gradle

#### 2、Jenkins

##### (1).搭建Jenkins自动部署环境

##### (2).Jenkins集成Maven、Git实现自动部署

##### (3).test/pre/production多环境部署

##### (4).Jenkins多环境配置、权限管理及插件使用

#### 3、Sonar

##### (1).使用Sonar进行代码质量管理

##### (2).关于代码检查工具findBugs/PMD的运用

##### (3).SonarQube代码质量管理平台的安装及使用

##### (4).使用Jenkins与Sonar集成对代码质量进行持续检测

##### (5).idea与sonar集合的使用

#### 4、Git

##### (1).什么时Git及Git的工作原理

##### (2).Git常用命令Best practise（避坑教学）

##### (3).Git冲突时怎么引起的，怎么解决

##### (4).架构师职责：Git Flow规范团队Git使用教程

##### (5).团队案例分享（买不到，才是最贵的）

#### 5、敏捷开发

##### (1).传统模式的问题

##### (2).当前行业面临的问题

##### (3).微服务与敏捷

##### (4).敏捷文化

##### (5).敏捷组织架构

##### (6).TDD、结对编程、CI

### 三、架构师必备技术栈（分布式架构）

#### 1、分布式架构原理

##### (1).漫谈分布式架构

##### (2).初识分布式架构及意义

##### (3).如何把应用从单机拓展到分布式

##### (4).大型分布式架构演进过程

##### (5).构建分布式架构最重要的元素

##### (6).CDN加速静态文件访问

#### 2、分布式架构策略

##### (1).分布式机构网络通信原理刨析

##### (2).通信协议中的序列化和反序列化

##### (3).基于框架的RPC技术WebService/RMI/Hession

##### (4).深入分析Zookeeper在discon配置中心的应用

##### (5).基于Zookeeper实现分布式服务器动态上下线感知

##### (6).深入分析Zookeeper Zab协议及选举机制源码解读

##### (7).Dubbo管理中心及监控平台安装部署

##### (8).基于Dubbo分布式系统架构实战

##### (9).Dubbo容错机制及高拓展性分析

#### 3、分布式架构中间件

##### (1).分布式消息通信ActiveMQ/Kafka/RabbitMQ

##### (2).Reids主从复制原理及无磁盘复制分析

##### (3).图解Redis中AOF和RDB持久化策略原理

##### (4).MongoDB企业级集群解决方案

##### (5).MongoDB数据分片、转存、及恢复策略

##### (6).基于OpenResty部署应用层Nginx及Nginx+lua实践

##### (7).Nginx反向代理服务器及负载均衡服务配置实战

##### (8).基于Netty实现高性能IM聊天

##### (9).基于Netty实现Dubbo多协议通信支持

##### (10).Netty无锁化串行及高并发处理机制

#### 4、分布式架构实践

##### (1).分布式全局ID生成方案

##### (2).Session跨域共享及企业级单点登陆解决方案实战

##### (3).分布式事务解决方案实战

##### (4).高并发下的服务降级、限流实战

##### (5).基于分布式架构下分布式锁的解决方案实战

##### (6).分布式架构下实现分布式定时调度

### 四、微服务架构

#### 1、微服务框架

##### (1).与微服务之间的关系

##### (2).热部署实战

##### (3).核心组件Stater、Actuator、AutoConfiguation、Cli集成MyBatis实现多数据源路由实战

##### (4).集成Dubbo实战

##### (5).集成Redis缓存实战

##### (6).集成Swagger2构建API管理及测试体系

##### (7).实现多环境配置动态解析

#### 2、Spring Cloud

##### (1).Eureka注册中心

##### (2).Ribbon集成REST实现负载均衡

##### (3).Fegion声明式服务调用

##### (4).Hystrix服务熔断降级方式

##### (5).Zuul实现微服务网关

##### (6).Config分布式统一配置中心

##### (7).Sleuth调用链路跟踪

##### (8).BUS消息总线

##### (9).基于Hystrix实现接口降级实战

##### (10).集成Spring Cloud实现统一整合方案

#### 3、Docker与虚拟化

##### (1).Docker的镜像、仓库、容器

##### (2).Docker File构建LNMP环境部署个人博客Wordpress

##### (3).Docker compose构建LNMP环境部署个人博客Wordpress

##### (4).Docker网络组成、路由互联、Openvswitch

##### (5).基于Swarn构建Docker集群实战

##### (6).Kubernetes简介

#### 4、漫谈微服务架构

##### (1).SOA架构和微服务架构之间的区别与联系

##### (2).如何设计微服务及其设计原则

##### (3).解惑Spring Boot流行因素及能解决什么问题

##### (4).什么是Spring Cloud,为何要选择Spring Cloud

##### (5).基于全局分析Spring Cloud各组件所解决的问题

### 五、性能优化

#### 1、理解性能优化

##### (1).性能基准

##### (2).性能优化到底是什么

##### (3).衡量维度

#### 2、JVM调优

##### (1).知其然，知其所以然

##### (2).详解什么是JVM运行时数据区

##### (3).详解什么是JVM内存模型JMM

##### (4).详解GC可达

##### (5).详解各垃圾回收器使用场景Throughput/CMS

##### (6).详解GC日志，从日志看端倪

##### (7).实战MAT分析dump文件（推理、验证）

#### 3、Tomcat调优

##### (1).how it works？探寻Tomcat运行机制及框架

##### (2).分析Tomcat线程模型

##### (3).Tomcat系统参数认识及调优

##### (4).基准测试

#### 4、MySql调优

##### (1).理解MySql底层B+Tree机制

##### (2).SQL执行计划详解

##### (3).索引优化详解

##### (4).SQL语句优化

### 六、架构必备（底层知识）

#### 1、java内存模型JMM

##### (1).线程通信

##### (2).消息传递

#### 2、内存模型

##### (1).重排序

##### (2).顺序一致性

##### (3).Happens-Before

##### (4).As-if-Serial

#### 3、Synchronized

##### (1).同步重量级锁

##### (2).Synchronized原理

##### (3).锁优化（自旋锁、轻量级锁、重量级锁、偏向锁）

#### 4、Volatile

##### (1).Volatile实现机制

##### (2).内存语义

##### (3).内存模型

#### 5、DCL

##### (1).单例模式

##### (2).DCL

##### (3).解决方案

#### 6、并发基础

##### (1).Aqs(Abstractqueuedsynchronized同步器、Cih同步队列、同步状态的获取和释放、线程阻塞和唤醒)

##### (2).Cas（Compare And Swap、缺陷）

#### 7、锁

##### (1).ReentrantLock

##### (2).ReentrantreadwriteLock

##### (3).Condition

#### 8、开发工具类

##### (1).Cyclicbarrier

##### (2).Countdownlatch

##### (3).Semphore

#### 9、并发集合

##### (1).Concurrenthashmap

##### (2).Concurrentlinkedqueue

#### 10、原子操作

##### (1).Atmicboolean

##### (2).AtmicInteger

##### (3).Atmiclong

##### (4).AtmicIntegeraray

##### (5).Atmicreferencearray

#### 11、引用类型

##### (1).Atmicreference

##### (2).Atmicreferencefieldupdater

#### 12、其他

##### (1).TheradLocal

##### (2).Fork/Join

#### 13、线程池

##### (1).Executor

##### (2).Threadpoolexecutor

##### (3).Callable和Future

##### (4).Scheduledexecutorservice

##### (5).Scheduledececutorservice

## 二、百万年薪架构师

### 一、智能互联网之总体架构设计篇

#### 1、智能互联网之互联网架构演进之路

#### 2、智能互联网之单体架构[Monolith]设计与实战

#### 3、智能互联网之水平分层架构[Horizontal hierarchical]设计与实践

#### 4、智能互联网之面向服务架构[Service Oriented Architecture]设计与实践

#### 5、智能互联网之微服务架构[Microservices]设计与实践

#### 6、智能互联网之服务网格架构[Service Mesh]设计与实践

#### 7、智能互联网千亿级真实案例实战

#### 8、进一步迭代微服务拆分方案，以适应容器云环境

### 二、智能互联网之核心技术实践

#### 1、互联网高可用设计手段

#### 2、互联网高并发设计手段

#### 3、互联网服务无状态化设计与实践

#### 4、互联网服务负载均衡设计与实践

#### 5、互联网服务幂等设计与实践

##### 什么是幂等性

HTTP/1.1中对幂等性的定义是：一次和多次请求某一个资源**对于资源本身**应该具有同样的结果（网络超时等问题除外）。也就是说，**其任意多次执行对资源本身所产生的影响均与一次执行的影响相同**。幂等性是系统服务对外一种承诺，声明为幂等的服务会认为外部调用失败是常态，并且失败之后必然会有重试。这里需要关注几个重点：

1.幂等不仅仅只是一次（或多次）请求对资源没有副作用（比如查询数据库操作，没有增删改，因此没有对数据库有任何影响）。

2.幂等还包括第一次请求的时候对资源产生了副作用，但是以后的多次请求都不会再对资源产生副作用。

3.幂等关注的是以后的多次请求是否对资源产生的副作用，而不关注结果。

4.网络超时等问题，不是幂等的讨论范围。

##### 什么时候需要幂等

用户在APP上连续点击了多次提交订单，后台应该只产生一个订单。

SELECT col1 FROM tab1 WHER col2=2，无论执行多少次都不会改变状态，是天然的幂等。

UPDATE tab1 SET col1=1 WHERE col2=2，无论执行成功多少次状态都是一致的，因此也是幂等操作。

UPDATE tab1 SET col1=col1+1 WHERE col2=2，每次执行的结果都会发生变化，这种不是幂等的。

#### 6、互联网分布式锁设计与实践

#### 7、互联网分布式事务设计与实践

#### 8、互联网服务降级设计与实践

#### 9、互联网服务限流设计与实践

#### 10、互联网服务熔断设计与实践

#### 11、互联网服务之灰度发布设计与实践

#### 12、互联网服务之全链路压测设计与实践

#### 13、互联网服务之千亿级真实案例实战

#### 14、分布式锁进一步细化场景，AP场景和CP场景的锁实现方案区别

#### 15、服务熔断新增基于容器云的技术考虑

#### 16、分布式事务柔性方案进一步考虑更多场景的支持

### 三、智能互联网之关键系统实战篇

#### 1、互联网注册中心系统设计与实践

#### 2、互联网配置中心系统设计与实践

#### 3、互联网消息队列系统设计与实践

#### 4、互联网分布式请求跟踪系统设计与实践

#### 5、互联网服务管理系统设计与实践

#### 6、互联网分布式服务调度系统设计与实践

#### 7、互联网智能个性化推荐系统设计与实践

#### 8、互联网智能搜索引擎设计与实践

#### 9、互联网之千亿级真实案例实践

### 四、智能互联网之弹性容器云与Service Mesh篇

#### 1、互联网云分类及使用场合

#### 2、互联网为什么需要弹性容器云

#### 3、互联网弹性容器云设计与实践

#### 4、互联网微服务架构痛点

#### 5、互联网服务网格架构方向

#### 6、互联网服务网格架构设计与实践

#### 7、互联网之千亿级真实案例实践

### 五、智能互联网之数据存储篇

#### 1、互联网数据库演进之路[RDBMS-NoSQL-NewSQL]

#### 2、互联网传统关系型数据库分库分表设计与实践

#### 3、互联网传统关系型数据库性能优化全攻略

#### 4、互联网传统关系型数据库高可用设计与实践

#### 5、互联网NoSQL分布式非关系型数据库设计与应用实践

#### 6、互联网NewSQL分布式关系型数据库设计与应用实践

#### 7、互联网数据库数据无缝迁移设计与实践

#### 8、互联网缓存数据设计与应用实践

#### 9、互联网数据一致性设计与实践

#### 10、互联网之千亿级真实案例实战

#### 11、新增数据库索引机制深度剖析

### 六、智能互联网之服务治理篇

#### 1、互联网服务治理地位

#### 2、互联网服务治理数据采集方法

#### 3、互联网服务治理之传统监控设计手段

#### 4、互联网服务治理之传统进程监控

#### 5、互联网服务治理之传统语义监控

#### 6、互联网服务治理之传统错误日志监控

#### 7、互联网服务治理之传统机器资源监控

#### 8、互联网服务治理之传统数据波动监控

#### 9、互联网服务治理之传统优秀开源监控框架

#### 10、互联网服务治理之新型立体监控背景

#### 11、互联网服务治理之新型立体监控解决问题

#### 12、互联网服务治理之新型立体监控设计与实践

#### 13、互联网服务治理之线上问题定位与分析

#### 14、互联网之千亿级真实案例实战

#### 15、服务治理方案进一步升级细化迭代

### 七、智能互联网之案例实践篇

#### 1、百度空间Feed流[类似于微信朋友圈/新浪微博]百亿级案例实践

#### 2、百度空间用户体系国际化千亿级案例实践

#### 3、IM[即时通信]百亿级案例实践

#### 4、Push移动推送千亿级案例实践

#### 5、二手交易平台千亿级案例实践

#### 6、其他公司高可用高并发案例

#### 7、具体案例进一步升级、加入交易等业务的分析

#### 8、增加国际化多IDC间服务/数据同步服务深度剖析

#### 9、新增安全方面架构设计案例剖析

### 八、智能互联网之开源框架篇

#### 1、智能互联网之RPC框架如何选择

#### 2、智能互联网之千亿级真实案例实战

### 九、智能互联网之未来架构演进篇

#### 1、互联网未来架构之两平台一中心一趋势

#### 2、互联网未来架构之弹性容器云平台

#### 3、互联网未来架构之服务网格平台

#### 4、互联网未来架构之服务治理中心

#### 5、互联网未来架构之人工智能趋势

### 十、智能互联网之架构师成长之路

#### 1、技术人员发展之道

#### 2、架构师的高度

#### 3、价值十评价架构师的核心准则

#### 4、技术第一生产力意义

#### 5、技术管理者视野

#### 6、专业决定高度

#### 7、团队如何知人

#### 8、以己为尺

#### 9、架构师的自我修养

#### 10、格局重要性

#### 11、如何培养架构师大格局以及大局观

#### 12、软技能方面进一步增加架构师成长、格局认知等方面的实战经验分享

#### 13、新增面试技巧的深度讲解

## 三、试题

### 1、用java自己实现一个LRU。

### 2、分布式集群下如何做到唯一序列号。

### 3、设计一个秒杀系统，30分钟没付款就自动关闭交易。

### 4、如何使用redis和zookeeper实现分布式锁？有什么区别优缺点，会有什么问题，分别适用什么场景。（延伸：如果知道redlock，讲讲他的算法实现，争议在哪里）

### 5、如果有人恶意创建非法连接，怎么解决。

### 6、分布式事务的原理，优缺点，如何使用分布式事务，2pc 3pc 的区别，解决了哪些问题，还有

### 7、哪些问题没解决，如何解决，你自己项目里涉及到分布式事务是怎么处理的。

### 8、什么是一致性hash。

### 9、什么是restful，讲讲你理解的restful。

### 10、如何设计一个良好的API。

### 11、如何设计建立和保持100w的长连接。

### 12、解释什么是MESI协议(缓存一致性)。

### 13、说说你知道的几种HASH算法，简单的也可以。

### 14、什么是paxos算法， 什么是zab协议。

### 15、一个在线文档系统，文档可以被编辑，如何防止多人同时对同一份文档进行编辑更新。

### 16、线上系统突然变得异常缓慢，你如何查找问题。

### 17、说说你平时用到的设计模式。

### 18、Dubbo的原理，有看过源码么，数据怎么流转的，怎么实现集群，负载均衡，服务注册

### 19、和发现，重试转发，快速失败的策略是怎样的 。

### 20、一次RPC请求的流程是什么。

### 21、自己实现过rpc么，原理可以简单讲讲。Rpc要解决什么问题。

### 22、异步模式的用途和意义。

### 23、编程中自己都怎么考虑一些设计原则的，比如开闭原则，以及在工作中的应用。

### 24、设计一个社交网站中的“私信”功能，要求高并发、可扩展等等。 画一下架构图。

### 25、MVC模式，即常见的MVC框架。

### 26、聊下曾经参与设计的服务器架构并画图，谈谈遇到的问题，怎么解决的。

### 27、应用服务器怎么监控性能，各种方式的区别。

### 28、如何设计一套高并发支付方案，架构如何设计。

### 29、如何实现负载均衡，有哪些算法可以实现。

### 30、Zookeeper的用途，选举的原理是什么。

### 31、Zookeeper watch机制原理。

### 32、Mybatis的底层实现原理。

### 33、请思考一个方案，实现分布式环境下的countDownLatch。

### 34、后台系统怎么防止请求重复提交。

### 35、描述一个服务从发布到被消费的详细过程。

### 36、讲讲你理解的服务治理。

### 37、如何做到接口的幂等性。

### 38如何做限流策略，令牌桶和漏斗算法的使用场景。

### 39、什么叫数据一致性，你怎么理解数据一致性。

### 40、分布式服务调用方，不依赖服务提供方的话，怎么处理服务方挂掉后，大量无效资源请求的浪费，如果只是服务提供方吞吐不高的时候该怎么做，如果服务挂了，那么一会重启，该怎么做到最小的资源浪费，流量半开的实现机制是什么。

### 41、dubbo的泛化调用怎么实现的，如果是你，你会怎么做。

### 42、远程调用会有超时现象，如果做到优雅的控制，JDK自带的超时机制有哪些，怎么实现的。

### 43、RPC

**简介：**PC（Remote Procedure Call）—[远程过程调用](https://baike.baidu.com/item/%E8%BF%9C%E7%A8%8B%E8%BF%87%E7%A8%8B%E8%B0%83%E7%94%A8/7854346)，它是一种通过[网络](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C/143243)从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。[RPC协议](https://baike.baidu.com/item/RPC%E5%8D%8F%E8%AE%AE)假定某些[传输协议](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE/8048821)的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI[网络通信](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1/9636548)模型中，RPC跨越了[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82/4329536)和[应用层](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E5%B1%82/4329788)。RPC使得开发包括网络[分布式](https://baike.baidu.com/item/%E5%88%86%E5%B8%83%E5%BC%8F)多程序在内的应用程序更加容易。

RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复[信息](https://baike.baidu.com/item/%E4%BF%A1%E6%81%AF/111163)，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

有多种 RPC模式和执行。最初由 Sun 公司提出。IETF ONC 宪章重新修订了 Sun 版本，使得 ONC RPC 协议成为 IETF 标准协议。现在使用最普遍的模式和执行是开放式软件基础的分布式计算[环境](https://baike.baidu.com/item/%E7%8E%AF%E5%A2%83)（DCE）。

**RPC对比http:**

RPC主要是基于TCP/IP协议的，而HTTP服务主要是基于HTTP协议的，我们都知道HTTP协议是在传输层协议TCP之上的，所以效率来看的话，HTTP是应用层协议，而TCP是传输层协议RPC当然是要更胜一筹啦。对于大型企业来说，内部子系统较多、接口非常多的情况下，RPC框架的好处就显示出来了，首先就是长链接，不必每次通信都要像http一样去3次握手什么的，减少了网络开销；其次就是RPC框架一般都有注册中心，有丰富的监控管理；发布、下线接口、动态扩展等，对调用方来说是无感知、统一化的操作。RPC效率更高，而HTTP服务开发迭代会更快。

**为什么要用RPC？**

其实这是应用开发到一定的阶段的强烈需求驱动的。

1. 如果我们开发简单的单一应用，逻辑简单、用户不多、流量不大，那我们用不着；

2. 当我们的系统访问量增大、业务增多时，我们会发现一台单机运行此系统已经无法承受。此时，我们可以将业务拆分成几个**互不关联的应用**，分别部署在各自机器上，以划清逻辑并减小压力。此时，我们也可以不需要RPC，因为应用之间是互不关联的。  
 3. 当我们的业务越来越多、应用也越来越多时，自然的，我们会发现有些功能已经**不能简单划分开来或者划分不出来**。此时，可以将公共业务逻辑抽离出来，将之组成独立的服务Service应用 。而原有的、新增的应用都可以与那些独立的Service应用 交互，以此来完成完整的业务功能。所以此时，我们急需**一种高效的应用程序之间的通讯手段**来完成这种需求，所以你看，RPC大显身手的时候来了！  
其实3描述的场景也是**服务化 、微服务** 和**分布式系统架构** 的基础场景。即RPC框架就是实现以上结构的有力方式。

**Java中常用的RPC框架**

目前常用的RPC框架如下：

1. **Thrift**：thrift是一个软件框架，用来进行可扩展且跨语言的服务的开发。它结合了功能强大的软件堆栈和代码生成引擎，以构建在 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml 这些编程语言间无缝结合的、高效的服务。

2. **Dubbo**：Dubbo是一个分布式服务框架，以及SOA治理方案。其功能主要包括：高性能NIO通讯及多协议集成，服务动态寻址与路由，软负载均衡与容错，依赖分析与降级等。 Dubbo是阿里巴巴内部的SOA服务化治理方案的核心框架，Dubbo自2011年开源后，已被许多非阿里系公司使用。

3. **Spring Cloud**：Spring Cloud由众多子项目组成，如Spring Cloud Config、Spring Cloud Netflix、Spring Cloud Consul 等，提供了搭建分布式系统及微服务常用的工具，如配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性token、全局锁、选主、分布式会话和集群状态等，满足了构建微服务所需的所有解决方案。Spring Cloud基于Spring Boot, 使得开发部署极其简单。

# 七、数据结构

## 一、数据结构——c++实现

### 三、线性表

#### 3.1线性表的定义

**线性表（linear-list）**是一种最常用最简单的数据结构，是N个相同类型元素的有限序列，记为L=(a1,a2,..an)。

#### 3.2线性表的顺序表示

用顺序存储的线性表成为**顺序表（sequencetial list）**，插入、删除和定位一个元素的时间复杂度为O（n）。

#### 3.3线性表的链表表示

线性表长度固定且要求物理存储空间连续，插入和删除时间复杂度为O(n),为了克服这些缺点，采用链表式存储线性表。链表式使用**节点（node）**构造链，节点一般有一个数据域和一个或多个指针域，根据指针域的个数分为**单链**、**单循环链**、**双向链**，其中每种链又分为**带头节点**和**不带头节点**。

##### 单链表

不带头节点



带头节点



（空链表）



（非空链表）

##### 单循环链表

不带头节点



带头节点



（非空链表）



（空链表）

##### 双向循环链表

不带头节点



带头节点



（非空链表）



（空链表）

##### 静态链表

### 四、栈、队列和递归

栈和队列是线性表的特殊情况，和线性表的数据元素和逻辑结构均相同，只是线性表可以在任意位置插入和删除，栈只能在表头位置插入和删除，队列只能在表头删除表尾插入。

#### 4.1栈

##### 顺序栈

顺序栈就是采用顺序存储方式存储的栈

##### 链式栈

链式栈就是用链式存储方式存储的栈



（不带头节点的单链式栈）

#### 4.2队列

### 五、串、数组、广义表

### 六、树和森林

#### 6.1树的概念

##### 6.1.1定义

**树(tree)T**是一个包含n(n≥0) 个数据元素（在树中每个数据元素用一个 结点表示）的有限集合。

##### 6.1.2术语

**节点（node）**：在树中每一个数据元素及指向其子树根的分支称为一个结点

**节点的度（degree of node）**：一个结点的子树数目称之为该结点的度

**终端结点(nonterminal node)**：在树中，度为0的结点称为终端结点或叶子(leaf)

**非终端结点(terminal node)**：在树中，度不为0的结点称为非终端结点或分支节点。

**树的度（degree of tree）**：树的节点中最大的度称为树的度。

**森林（forest）**：森林是m(m≥0)棵互不相交的树的集合

##### 6.1.3树的表示形式

树形表示法、嵌套集合表示法、凹入目录表示法、广义表形式表示法

#### 6.2二叉树

##### 定义

**二叉树BT**是有限个节点的集合。

高度为h的二叉树具有最大数目(2h-1)个节点，则为**满二叉树(full binary tree)**

高度为h的二叉树，除第h层外，其余层均有最大节点数，且h层节点从左向右连续分布，则称其为**完全二叉树(complete binary tree)**

##### 性质

1. 有n(n>0)个节点的二叉树分支树为n-1.
2. 若二叉树的高为h(h≥0),则最少有h个节点，最多有2h-1个节点.
3. 含有n个节点的二叉树，高度最大为n，最小为log2(n+1).
4. 一棵有n个节点的完全二叉树，从上到下，从左到右依次编号为0,1,2,.. ,n-1则有以下关系：
5. .若i=0，则i无双亲，若i>0,i的**双亲**为(i/2)-1.
6. .若2\*i+1≤n，则**左孩子**为2\*i+1。
7. .若2\*i+2≤n，则**右孩子**为2\*i+2。
8. .若n为偶数且i≥1，则i是双亲的右孩子，且有i-1的左兄弟。
9. .若n为奇数且i<n-1,则i是双亲的左孩子，且有i+1的右兄弟。

#### 6.3二叉树的存储结构

##### 数组存储

将根结点中的数据元素存储在数组的0号位置，对于二叉树中任一个结点，如果它的数据元素存储在数组的第i个位置，就把它的左、右孩子结点中的数据元素分别存放在数组的第2\*i+1,2\*i+2位置。这种表示法对于非完全二叉树尤其是单支树，空间利用率低，且空节点必须设置特殊标识。

##### 链表表示法

一般情况下，用链表表示二叉树是比较合适的。在二叉树的链表表示中，树中的每一个元素用一个结点表示，结点一般包括三个域，date域存储数据元素的信息，leftchild用于存放左孩子节点的指针，rightChild用于存放右孩子的指针。

在二叉链表中要想找出一个结点的双亲是比较困难的，必须通过二叉树的遍历才能实现。如果在应用中需要方便地找到任何一个结点的双亲，可以在结点中增加一个**Parent**域来指向该结点的**双亲**，这种表示方法称为**三叉链表。**

** （二叉树三叉链表表示法）**

#### 6.4遍历二叉树

##### 6.4.1前序遍历

前序遍历（preorder travelsal）的递归定义为：若二叉树为空，遍历结束。否则

1.访问根结点

2.前序遍历根结点的左子树；

3.前序遍历根结点的右子树。

##### 6.4.2中序遍历

中序遍历（inorder travelsal）的递归定义为：若二叉树为空，遍历结束。否则，

1.中序遍历根结点的左子树；

2.访问根结点；

3.中序遍历根结点的右子树。

##### 6.4.3后序遍历

后序遍历（postorder travelsal）的递归定义为：若二叉树为空，遍历结束。否则，

1.后序遍历根结点的左子树；

2.后序遍历根结点的右子树；

3.访问根结点。

前缀表达式和后缀表达式分别称为**波兰式**和**逆波兰式**，它们在编译程序中起着非常重要的作用。中缀表达式和人们习惯使用的表达式类似，但是由于中缀表达式中没有括号使其不能正确反映运算的实际顺序，所以用处不大。

在遍历二叉树时，无论采用前面所讲的哪一种方式进行遍历，其基本操作都是访问结点，遍历操作的**时间复杂度均为O(n)**,。在前序、中序和后序遍历的过程中，递归时栈所需要的空间最多等于树的深度h乘以每个栈元素所需空间，在最坏的情况下，二叉树的深度等于结点数目，所以**空间复杂度为O(n)。**

#### 6.5线索二叉树

#### 6.6二叉树的应用

二叉树结构在实际问题中有着广泛的应用。二叉排序树、堆排序、哈夫曼树等是最典型的二叉树应用。

##### 6.6.1堆

设有n个数据元素的关键字为(k0,k1,...,kn-1)，如果它们满足以下的关系ki≤k2i+1且ki≤k2i+2（或ki≥k2i+1且ki≥k2i+2）则称之为堆(heap)。在**完全二叉树**中任何非终端结点的关键字均不大于（或不小于）其左、右孩子结点的关键字。

##### 6.6.2哈夫曼树

一棵二叉树中由根结点到某个结点所经过的分支序列叫做由根结点到这个结点的**路径**，由根结点到某个结点所经过的分支数称为由根结点到该结点的**路径长度**。由根结点到所有叶结点的路径长度之和称为该**二叉树的路径长度。**如果二叉树中每一个叶结点都带有某一确定权从根结点到各个叶结点的路径长度与对应叶结点权值的乘积之和叫做**二叉树的带权路径长度，**记作WPL=∑Wk\*Lk 。对于一组确定权值的叶结点，所构造出的不同形态二叉树的带权路径长度并不相同。在此把其中具有最小带权路径长度的二叉树称为**最优二叉树**，最优二叉树也称为**哈夫曼树。**

#### 6.7树和森林

##### 6.7.1树的存储结构

###### 双亲表示法

用一组连续的存储空间（数组）存储树中的每一个结点，数组中的一个元素表示树中的一个结点，在数组元素中除包括结点本身的数据信息外，还保存该结点的双亲结点在数组中的序号。树的这种存储方法也称为**双亲表示法**。



(树)



(树的双亲表示数组)

###### 孩子表示法

用一个多重链表表示树，链表中的每个结点包括一个数据域和多个指针域。数据域存储树中结点的自身信息，每个指针指向该结点的一个孩子结点，通过各个指针反映树中各结点之间的关系。在这种表示法中，树中每个结点有多个指针域，形成了多重链。树的这种存储方法又常称作**多重链表法**。

###### 双亲-孩子表示法



###### 孩子-兄弟表示法



##### 6.7.2树、森林和二叉树的转换

###### 树转换为二叉树

树向二叉树的转化就是按照树的孩子-兄弟表示法进行的，将一棵树转化为二叉树的方法是：

1.连线：树中所有相邻兄弟结点之间加一条线

2.删线：对树中的每个结点，只保留它与第一个孩子结点之间的连线，删去它与其他孩子结点之间的连线

3.美化：以树的根结点为轴心，将这棵树顺时针转动 使其层次分明



###### 森林转换为二叉树

森林转化为二叉树的方法为：

1.依次将森林中的每棵树转化成相应的二叉树

2.从第二棵二叉树开始，依次把当前的二叉树作为前一棵二叉树根结点的右子树，此时所得到的二叉树就是由森林转化得到的二叉树。

 

###### 二叉树转换为森林

将一棵二叉树转化为树或森林，具体方法如下：

1.连线：若结点P是其双亲结点F的左孩子，则把从结点P沿右分支所找到的所有节点和节点F用线连起来。

2.删线：删除二叉树中所有结点和其右孩子结点之间的连线。

3.美化：整理由步骤1和2所得到的树或森林，使之结构层次分明





##### 6.7.3树的遍历

###### 树的先根遍历

若树为空，遍历结束。否则，

1.访问根结点；

2.按照从左到右的顺序先根遍历根结点的每一棵子树。

###### 树的后根遍历

若树为空，遍历结束。否则，

1.按照从左到右的顺序先根遍历根结点的每一棵子树；

2.访问根结点。

###### 树的层次遍历

就是从树的第一层（根结点）开始，自上至下逐层遍历，在同一层中，则按从左到右的顺序对结点逐个访问。

##### 6.7.4森林的遍历

###### 森林的先根遍历

若森林为空，返回；否则，

1.访问森林中第一棵树的根结点；

2.先根遍历第一棵树的根结点的子树森林；

3.先根遍历除第一棵外其他树组成的森林

###### 森林的中根遍历

若森林为空，返回；否则，

1.中根遍历第一棵树的根结点的子树森林；

2.访问森林中第一棵树的根结点；

3.中根遍历除第一棵外其他树组成的森林

###### 森林的后根遍历

若森林为空，返回；否则，

1.后根遍历第一棵树的根结点的子树森林；

2.后根遍历除第一棵外其他树组成的森林；

3.访问森林中第一棵树的根结点

### 七、图

图(Graph)是一种比树更为复杂的非线性数据结构,Graph=(V,E),其中V={x|x∈某个数据对象}是数据元素的集合，在图中，数据元素一般被称为**顶点(vertex)**,E={(v,w)|v,w∈V}是数据元素之间关系的集合，也叫做**边（edge）**集合。Path(v,w)表示从顶点v到w的一条单向通路，它是有方向的。如果顶点对(v,w)是无序的，则此图称为**无向图**；如果顶点对<v,w>是有序的，则此图称为**有向图。**

# 八、算法

## 一、打开算法之门

### 什么是算法及为什么要关注算法

#### 正确性

#### 资源利用

我们关注影响算法运行时间的主项，而不关心系数，也就是说我们关注运行时间的增长量级。假设我们搜索一个具有n项的表的算法需要花费50n+125个机器周期，其中n≥3时，50n相对于125有更大的影响，n越大就越明显，因此我们描述这个算法的运行时间时，我们忽略低阶项125，我们还将舍弃系数50.从抽象的角度比较算法时，我们只关注算法的增长顺序，而会忽视主项的系数或者低阶项。

#### 针对非计算机专业人士的计算机算法

即使你并非一个计算机内行，算法也与你密切相关，你使用的GPS、搜索引擎都使用到了算法。

#### 针对计算机专业人士的计算机算法

即使计算机硬件技术不断改进和发展，但是整个系统的性能不仅仅依靠选择运行较快的硬件或者高效的操作系统，选择高效的算法对提升系统性能也同样重要。

### 如何描述和评估计算机算法

#### 如何描述计算机算法

### 数据压缩

压缩分为**有损压缩**和**无损压缩**，MP3和JPEG是有损压缩，zip是无损压缩。

### 难？问题

#### 棕卡车问题

即**旅行商问题，**还没有发现是否存在能在O(nC)(C为常数)时间内完成的算法。许多问题拥有同样的特点，也没有人证明这样的算法不存在。我们称这些问题为**NP-完全问题（NP-complete）**。

#### 2、P、NP、NP-完全类

一般将能在**多项式**时间内解决的问题视为易处理的，称此类问题为**P类**问题。

## 二、算法导论

### 1.算法在计算机中的作用

### 2.算法基础

### 3.函数的增长

### 4.分治策略

#### (1).基本思想及策略：

       首先，说说分治的思想。分治， “分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础，如排序算法(快速排序，归并排序)，傅立叶变换等。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。

       分治通过减小问题规模，对问题各个击破，其策略是：对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决，否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。

#### (2).分治法适用的情况

    分治法所能解决的问题一般具有以下几个特征：

    1 该问题的规模缩小到一定的程度就可以容易地解决

    2 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

    3 利用该问题分解出的子问题的解可以合并为该问题的解；

    4 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

#### (3).分治法的基本步骤

分治法在每一层递归上都有三个步骤：

    step1 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

    step2 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题

step3 合并：将各个子问题的解合并为原问题的解。

#### (4).实例

1).平面最近点对问题

       最近点对问题：给定平面上的N个点，一共可以组成n(n-1)/2对点对，找出距离最近的两个点。蛮力法就是对这n(n-1)/2对点对逐对进行距离计算，通过循环求得点集中的最近点对。

**分治法代码参考**

**import** java.util.Arrays;

// 提供一个静态方法返回平面内最近点对：public static Point[] nearest(Point[] array)

// 其他的方法皆为私有的助手方法

**public** **class** NearestPoint {

// 返回给定点集中的最近点对，如果只有一个点，则返回只包含一个点的数组

**public** **static** Point[] nearest(Point[] array) {

Point[] cpArray = Arrays.*copyOfRange*(array, 0, array.length);

SortComparable.sort(cpArray);

**return** *mergeNearest*(cpArray, 0, cpArray.length - 1);

}

// 分治求最近点对

**private** **static** Point[] mergeNearest(Point[] points, **int** begin, **int** end) {

**if** (begin == end)

**return** **new** Point[] { points[begin] };

**else** {

**int** mid = (begin + end) / 2;

Point[] left = *mergeNearest*(points, begin, mid);

Point[] right = *mergeNearest*(points, mid + 1, end);

**return** *merge*(left, right, points, begin, mid, end);

}

}

// 合并左右两边，求最近点对

**private** **static** Point[] merge(Point[] left, Point[] right, Point[] points,

**int** begin, **int** mid, **int** end) {

**if** (left.length == 1 && right.length == 1)

**return** **new** Point[] { left[0], right[0] };

Point[] partNearest = *partNearest*(left, right);

**return** *mergeNearest*(points, partNearest, begin, mid, end);

}

// 比较左右两边，返回局部最近点对

**private** **static** Point[] partNearest(Point[] left, Point[] right) {

// 两个数组长度不可能同时为0

**if** (left.length == 1) {

**return** right;

} **else** **if** (right.length == 1) {

**return** left;

} **else** {

**if** (left[0].distance(left[1]) < right[0].distance(right[1]))

**return** left;

**else**

**return** right;

}

}

**private** **static** Point[] mergeNearest(Point[] points, Point[] partNearest,

**int** begin, **int** mid, **int** end) {

// 初始化两边在d范围内的点， 并按y坐标排序

Point[] left = **null**;

Point[] right = **null**;

Point[] nearest = partNearest;

**double** middle = (points[mid].getX() + points[mid + 1].getX()) / 2;

**double** d = partNearest[0].distance(partNearest[1]);

**for** (**int** i = mid; i >= begin - 1; i--) {

**if** (i != begin - 1) {

**if** (middle - points[i].getX() <= d)

**continue**;

} **else** {

left = Arrays.*copyOfRange*(points, i + 1, mid + 1);

SortComparable.sort(left, **new** PointYComparaotr());

**break**;

}

}

**for** (**int** i = mid + 1; i <= end + 1; i++) {

**if** (i != end + 1) {

**if** (points[i].getX() - middle <= d)

**continue**;

} **else** {

right = Arrays.*copyOfRange*(points, mid + 1, i);

SortComparable.sort(right, **new** PointYComparaotr());

**break**;

}

}

// 遍历left数组，在right中寻找符合条件的点

**for** (Point inLeft : left) {

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

**if** (inLeft.getY() - right[i].getY() > d)

**continue**;

**if** (inLeft.getY() - right[i].getY() < -d)

**break**;

**if** (inLeft.distance(right[i]) < d) {

nearest = **new** Point[] { inLeft, right[i] };

d = inLeft.distance(right[i]);

}

}

}

**return** nearest;

}

}

**public** **class** TestNearestPoints {

**static** **final** **int** ***SIZE*** = 100;

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

PointGenerator pg = **new** PointGenerator(10000);

Point[] array = **new** Point[***SIZE***];

**for**(**int** i = 0; i < array.length; i++)

array[i] = pg.next();

Point[] nearP = NearestPoint.*nearest*(array);

**double** d = nearP[0].distance(nearP[1]);

System.***out***.println(*nearest*(array) == d);

}

**private** **static** **double** nearest(Point[] array) {

**double** d = 214748364; // the guard

**for**(**int** i = 0; i < array.length - 1; i++)

**for**(**int** j = i + 1; j < array.length; j++)

**if**(d > array[i].distance(array[j])) {

d = array[i].distance(array[j]);

}

**return** d;

}

}

### 5.概率分析与随机算法

### 6.堆排序

### 7.快速排序

### 8.线性时间排序

### 9.中位数与顺序统计量

### 10.基本数据结构

### 11.散列表

### 12.二叉搜索树

### 13.红黑树

### 14数据结构的扩张

### 15.动态规划

### 16.贪心算法

### 17.摊还分析

### 18.B树

### 19.斐波那契堆

### 20.Van Emde Boas树

### 21.用于不相交集合的数据结构

### 22.基本的图算法

### 23.最小生成树

### 24.单源最短路径

### 25.所有结点对的最短路径问题

### 26.最大流

### 27.多线程算法

### 28.矩阵运算

### 29.线性规划

### 30.多项式与快速傅里叶变换

### 31.数论算法

### 32.字符串匹配

### 33.计算几何学

### 34.NP完全性

### 35.近似算法

## 三、算法趣题

### 1.二、八、十进制回文数

/\*\*

\* 求用十进制、二进制、八进制表示都是回

\* 文数的所有数字中，大于十进制数10 的最小值。

\* 2019/06/18

\*/

**private** **static** **void** PalindromeNumber(){

Integer num = 11;

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

**if**(Integer.*toBinaryString*(num).equals(**new** StringBuilder(Integer.*toBinaryString*(num)).reverse().toString())//二进制回文

&& Integer.*toOctalString*(num).equals(**new** StringBuilder(Integer.*toOctalString*(num)).reverse().toString())//八进制回文

&& String.*valueOf*(num).equals(**new** StringBuilder(String.*valueOf*(num)).reverse().toString())){//十进制回文

System.***out***.println("最小数为:"+num);//结果585

**break**;

}

num+=2;//二进制回文最高位为1是最低位也为1，所以结果必为奇数

}

}

### 2.数列四则运算

/\*\*

\* 组合的方法是在各个数字之间插入四则运算的运算符组成算式，然后计算算式的结果（某些数位之间可以没有运算符，但最少要插入1 个运算符）

\* 求位于1000~9999，满足上述条件的数 例：351　→　3×51＝ 153

\* **@throws** ScriptException

\*

\*/

**private** **static** **void** theFourFundamental() **throws** ScriptException{

String [] op = {"+","-","\*","/",""};

**for**(**int** i = 1000;i < 9999;i++){

String numStr = String.*valueOf*(i);

**for**(**int** j = 0;j<op.length;j++){

**for**(**int** k = 0;k<op.length;k++){

**for**(**int** l = 0;l<op.length;l++){

String val = String.*valueOf*(numStr.charAt(3))+op[j]

+String.*valueOf*(numStr.charAt(2))+op[k]

+String.*valueOf*(numStr.charAt(1))+op[l]

+String.*valueOf*(numStr.charAt(0));

ScriptEngineManager manager = **new** ScriptEngineManager();

ScriptEngine se = manager.getEngineByName("js");

**if**(val.length() > 4 && String.*valueOf*(i).equals(String.*valueOf*(se.eval(val)))){

System.***out***.println(i+"表达式："+val);//1395表达式：5\*9\*31

}

}

}

}

}

System.***out***.println("----结束");

}

### 3.翻牌

/\*\*

\* 这里有100 张写着数字1~100 的牌，并按顺序排列着。最开始所有牌都是背面朝上放置。某人从第2 张牌开始，隔1 张牌翻牌。然后第2,

\*4, 6, …, 100 张牌就会变成正面朝上。

\*接下来，另一个人从第3 张牌开始，隔2 张牌翻牌（原本背面朝上的，翻转成正面朝上；原本正面朝上的，翻转成背面朝上）。再接下来，

\*又有一个人从第4 张牌开始，隔3 张牌翻牌（ 图1 ）。

\*像这样，从第n 张牌开始，每隔n－1 张牌翻牌，直到没有可翻动的牌为止。

\*求当所有牌不再变动时，所有背面朝上的牌的数字。

\*/

**private** **static** **void** turnOver(){

**int** [] arr = **new** **int**[100];

**int** size = 2;

**for**(**int** j=1;j<arr.length;j++){

**for**(**int** i = j;i<arr.length;i += size){

**if**(arr[i]==0){

arr[i] = 1;//1为正面

}**else**{

arr[i] = 0;

}

}

size++;

}

**for**(**int** k = 0;k<arr.length;k++){

**if**(arr[k]==0){

System.***out***.println(k+1);

}

}

}

### 4.切分木棒

/\*\*

\* 切割木棒

\* 假设要把长度为n 厘米的木棒切分为1 厘米长的小段，但是1 根木棒只能由1 人切分

\* 求最多有m 个人时，最少要切分几次

\*/

**private** **static** **void** cuttingSticks(**int** n,**int** m,**int** current,**int** times){

**if**(current >= n){//切割完成

System.***out***.println("切割次数："+times);

}**else** **if**(current <= m){

*cuttingSticks*(n,m,current\*2,++times);

}**else**{

*cuttingSticks*(n,m,current+m,++times);

}

}

### 还用现金支付吗

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

list.add(0, 10);

list.add(1, 50);

list.add(2, 100);

list.add(3, 500);

quTi.cashPay(1000,list,15);

/\*\*

\*

\* 有 10，50，100，500几种面值

\* 最多15张现金有多少种方式组成1000

\* target:目标值：1000

\* list {10,50,100,500} 币种

\* num 15 最多钱币

\*/

**int** count =0;//组成方式数量

**public** **void** cashPay(**int** target,List list,**int** num) {

List<Integer> copyList = **new** ArrayList<Integer>();

copyList.addAll(list);

**if**(copyList != **null** && copyList.size() >= 1) {

**int** current = (**int**) copyList.get(0);

copyList.remove(0);

**if**(copyList == **null** || copyList.size() == 0) {

**if**((target/current <= num && target%current ==0)) {//

System.***out***.println(++count);

}

}**else** {

**for** (**int** i = 0;i<= (target/current<num ?target/current :num);i++) {

cashPay(target-current \* i,copyList,num-i);

}

}

}

}

### 考拉兹猜想

## 四、算法竞赛入门经典训练指南

### 第四章 算法设计方法

## 五、排序算法

### (1)、冒泡排序

**简介：**冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

**时间复杂度：**冒泡排序最好的[时间复杂度](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6)为 IMG_256 ，冒泡排序的最坏时间复杂度为 IMG_256 ，因此冒泡排序总的平均时间复杂度为 IMG_257。

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(n²) | O(n) | O(n²) | O(1) | 不稳定 |

**稳定性**：假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些**记录的相对次序保持不变**，即在原序列中，r[i]=r[j]，且r[i]在r[j]之前，而在排序后的序列中，r[i]仍在r[j]之前，则称这种排序算法是稳定的；否则称为不稳定的。冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，是不会再交换的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

**代码示例：**

/\*\*

\* 冒泡排序

\* 比较相邻的元素。如果第一个比第二个大，就交换他们两个。

\* 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

\* 针对所有的元素重复以上的步骤，除了最后一个。

\* 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

\* **@param** numbers 需要排序的整型数组

\*/

**public** **static** **void** bubbleSort(**int**[] numbers){

**int** temp = 0;

**int** size = numbers.length;

**for**(**int** i = 0 ; i < size-1; i ++){

**for**(**int** j = 0 ;j < size-1-i ; j++){

**if**(numbers[j] > numbers[j+1]){//交换两数位置

temp = numbers[j];

numbers[j] = numbers[j+1];

numbers[j+1] = temp;

}

}

}

### (2)、快速排序

**简介**：快速排序（Quicksort）,基于分治的思想，是冒泡排序的改进型。首先在数组中选择一个基准点（该基准点的选取可能影响快速排序的效率，后面讲解选取的方法），然后分别从数组的两端扫描数组，设两个指示标志（lo指向起始位置，hi指向末尾)，首先从后半部分开始，如果发现有元素比该基准点的值小，就交换lo和hi位置的值，然后从前半部分开始扫秒，发现有元素大于基准点的值，就交换lo和hi位置的值，如此往复循环，直到lo>=hi,然后把基准点的值放到hi这个位置。一次排序就完成了。以后采用递归的方式分别对前半部分和后半部分排序，当前半部分和后半部分均有序时该数组就自然有序了。

**空间效率**：由于快速排序是递归的，需要借助一个递归工作栈来保存每一层递归调用的必要信息，其容量应与递归调用的最大深度一致。最好情况下为⌈log2(n+1)⌉;最坏情况下，因为要进行n-1次递归调用，所以栈的深度为O(n)；平均情况下栈的深度为O（log2n)。因而空间复杂度在最坏情况下为O(n),平均情况下为O(log2n)。

**时间效率**：快速排序的运行时间与划分是否对称有关，而后者又与具体使用的划分算法有关。快速排序最坏的情况发生在两个区域分别包含n-1个元素和0个元素时，这种程度的不对称性若发生在每一层递归上，即对应于初始排序表基本有序货基本逆序时，就得到最坏情况下的时间复杂度为O(n2)。

　　　有很多方法可以提高算法的效率。一种方法是当递归过程中划分得到的子序列的规模较小时不要再继续调用快速排序，可以直接采用直接插入排序算法进行后续的排序工作。另一种方法就是尽量选取一个可以将数据中分的枢轴元素。如从序列的头尾以及中间选取三个元素，再取这三个元素的中间值作为最终的枢轴元素（数据结构与算法分析）；或者随机从当前列表中选取枢轴元素（算法导论），这样做使得最坏情况在实际安排中几乎不会发生。

　　在最理想状态下，也即Partition()可能做到最平衡的划分中，得到的两个子问题的大小都不可能大于n/2，这种情况下，快速排序的运行速度将大大提升，此时，时间复杂度为O(nlog2n)。好在快速排序平均情况下运行时间与其最佳情况下的运行时间很接近，而不是接近最坏情况下的运行时间。

**快速排序是所有内部排序算法中平均性能最优的排序算法。**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(nlog₂n) | O(nlog₂n) | O(n²) | O(1)（原地分区递归版） | 不稳定 |

**代码示例：**

**public** **static** **int** partition(**int** []array,**int** lo,**int** hi){

**int** key=array[lo]; //固定的切分方式

**while**(lo<hi){

**while**(array[hi]>=key && hi>lo){//从后半部分向前扫描

hi--;

}

array[lo]=array[hi];

**while**(array[lo]<=key && hi>lo){//从前半部分向后扫描

lo++;

}

array[hi]=array[lo];

}

array[hi]=key;

**return** hi;

}

**public** **static** **void** sort(**int**[] array,**int** lo ,**int** hi){

**if**(lo>=hi){

**return** ;

}

**int** index=*partition*(array,lo,hi);

*sort*(array,lo,index-1);

*sort*(array,index+1,hi);

}

### 、选择排序

**选择排序（Selection sort）**：一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对 n个元素的表进行排序总共进行至多 n-1 次交换。在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。

**复杂度分析**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(n²) | O(n²) | O(n²) | O(1) | 不稳定 |

**代码示例：**

**public** **static** **void** selectionSort(**int**[] a) {

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

**int** min = i; //选出之后待排序中值最小的位置

**for** (**int** j = i + 1; j < a.length; j++) {

**if** (a[j] < a[min]) {

min = j;

}

} //最小值不等于当前值时进行交换

**if** (min != i) {

**int** temp = a[i];

a[i] = a[min];

a[min] = temp;

}

}

System.***out***.println(Arrays.toString(a));

}

### 、插入排序

**插入排序（Insertion sort）**：思想是循环数组中每一个元素，然后将这些元素依次插入到一个有序的数列中去，实现数组的排序，也就是说循环到的当前元素索引的左侧都是已经排好序的，针对当前元素，将它依次与相邻的左侧元素比较，小于左侧就两个交换，循环比较和交换的过程，直至元素大于左侧相邻元素，或者是该元素已被交换至最左侧。

**复杂度分析：**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(n²) | O(n²) | O(n²) | O(1) | 稳定 |

**代码实现：**

**public** **static** **void** insertionSort(**int**[] arr) {

**int** i, j;

**int** n = arr.length;

**int** target;

//假定第一个元素被放到了正确的位置上

//这样，仅需遍历1 - n-1

**for** (i = 1; i < n; i++){

j = i;

target = arr[i];

**while** (j > 0 && target < arr[j - 1]){

arr[j] = arr[j - 1];

j--;

}

arr[j] = target;

}

System.***out***.println(Arrays.toString(arr));

}

### 、希尔排序

**希尔排序(Shell's Sort)**，也称 递减增量排序算法，是插入排序的一种更高效的改进版本。希尔排序是 非稳定排序算法。希尔排序是基于插入排序的以下两点性质而提出改进方法的：

插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率，但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位。

希尔排序是先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

**基本思想**：将待排序数组按照步长gap进行分组，然后将每组的元素利用直接插入排序的方法进行排序；每次再将gap折半减小，循环上述操作；当gap=1时，利用直接插入，完成排序。

可以看到步长的选择是希尔排序的重要部分。只要最终步长为1任何步长序列都可以工作。一般来说最简单的步长取值是初次取数组长度的一半为增量，之后每次再减半，直到增量为1。

**复杂度分析：**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(nlog2 n) | O(nlog2 n) | O(nlog2 n) | O(1) | 不稳定 |

**代码实现：**

**public** **static** **void** sort(**int**[] a) {

**int** length = a.length;

**int** h = 1;

**while** (h < length / 3)

h = 3 \* h + 1;

**for** (; h >= 1; h /= 3) {

**for** (**int** i = 0; i < a.length - h; i += h) {

**for** (**int** j = i + h; j > 0; j -= h) {

**if** (a[j] < a[j - h]) {

**int** temp = a[j];

a[j] = a[j - h];

a[j - h] = temp;

}

}

}

}

}

### 、归并排序

**归并排序**是建立在归并操作上的一种有效的排序算法，1945年由约翰·冯·诺伊曼首次提出。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用，且各层分治递归可以同时进行。

归并排序算法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

**复杂度分析：**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(nlog₂n) | O(nlog₂n) | O(nlog₂n) | O(n) | 稳定 |

**代码实现：**

**public** **class** Merge {

**private** **static** **int**[] *aux*; //归并所需的辅助数组

**public** **static** **void** sort(**int**[] a) {

*aux* = **new** **int**[a.length]; //一次性分配空间

*sort*(a, 0, a.length - 1);

}

**public** **static** **void** sort(**int**[] a, **int** low, **int** high) {

**if** (low >= high) {

**return**;

}

**int** mid = (low + high) / 2;

*sort*(a, low, mid);//将左半边排序

*sort*(a, mid + 1, high);//将右半边排序

*merge*(a, low, mid, high);

}

/\*\*

\* 该方法先将所有元素复制到aux[]中，然后在归并会a[]中。方法咋归并时(第二个for循环)

\* 进行了4个条件判断：

\* - 左半边用尽(取右半边的元素)

\* - 右半边用尽(取左半边的元素)

\* - 右半边的当前元素小于左半边的当前元素(取右半边的元素)

\* - 右半边的当前元素大于等于左半边的当前元素(取左半边的元素)

\* \*/

**public** **static** **void** merge(**int**[] a, **int** low, **int** mid, **int** high) {

//将a[low..mid]和a[mid+1..high]归并

**int** i = low, j = mid + 1;

**for** (**int** k = low; k <= high; k++) {

*aux*[k] = a[k];

}

**for** (**int** k = low; k <= high; k++) {

**if** (i > mid) {

a[k] = *aux*[j++];

} **else** **if** (j > high) {

a[k] = *aux*[i++];

} **else** **if** (*aux*[j] < *aux*[i]) {

a[k] = *aux*[j++];

} **else** {

a[k] = *aux*[i++];

}

}

}

}

### (7)、堆排序

**堆排序**是一种树形选择排序，是对直接选择排序的有效改进。

**堆定义**：具有n个元素的序列 （h1,h2,...,hn),当且仅当满足（hi>=h2i,hi>=2i+1）或（hi<=h2i,hi<=2i+1） (i=1,2,...,n/2)时称之为堆。在这里只讨论满足前者条件的堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项（大顶堆）。完全二 叉树可以很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。

**思想**:初始时把要排序的数的序列看作是一棵顺序存储的二叉树，调整它们的存储顺序，使之成为一个 堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点交换。然后对前面(n-1)个数重新调整使之成为堆。依此类推，直到只有两个节点的堆，并对 它们作交换，最后得到有n个节点的有序序列。从算法描述来看，堆排序需要两个过程，**一是建立堆**，**二是堆顶与堆的最后一个元素交换位置**。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

**复杂度分析**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(nlog₂n) | O(nlog₂n) | O(nlog₂n) | **O(1)** | 不稳定 |

**代码示例：**

**public** **class** HeapSort {

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

**int**[] a={49,38,65,97,76,13,27,49,78,34,12,64};

**int** arrayLength=a.length;

**for**(**int** i=0;i<arrayLength-1;i++){ //循环建堆

*buildMaxHeap*(a,arrayLength-1-i); //建堆

*swap*(a,0,arrayLength-1-i); //交换堆顶和最后一个元素

System.***out***.println(Arrays.toString(a));

}

}

//对data数组从0到lastIndex建大顶堆

**public** **static** **void** buildMaxHeap(**int**[] data, **int** lastIndex){

//从lastIndex处节点（最后一个节点）的父节点开始

**for**(**int** i=(lastIndex-1)/2;i>=0;i--){

**int** k=i; //k保存正在判断的节点

**while**(k\*2+1<=lastIndex){ //如果当前k节点的子节点存在

**int** biggerIndex=2\*k+1; //k节点的左子节点的索引

//如果biggerIndex小于lastIndex，即biggerIndex+1代表的k节点的右子节点存在

**if**(biggerIndex<lastIndex){

//若果右子节点的值较大

**if**(data[biggerIndex]<data[biggerIndex+1]){

//biggerIndex总是记录较大子节点的索引

biggerIndex++;

}

}

//如果k节点的值小于其较大的子节点的值

**if**(data[k]<data[biggerIndex]){

//交换他们

*swap*(data,k,biggerIndex);

//将biggerIndex赋予k，开始while循环的下一次循环，重新保证k节点的值大于其左右子节点的值

k=biggerIndex;

}**else**{

**break**;

}

}

}

}

//交换

**private** **static** **void** swap(**int**[] data, **int** i, **int** j) {

**int** tmp=data[i];

data[i]=data[j];

data[j]=tmp;

}

}

### (8)、基数排序

**基数排序(radix sort)**又称桶排序（bucket sort），相对于常见的比较排序，基数排序是一种分配式排序，即通过将所有数字分配到应在的位置最后再覆盖到原数组完成排序的过程。

**复杂度分析**

| **平均时间复杂度** | **最好情况** | **最坏情况** | **空间复杂度** | **稳定性** |
| --- | --- | --- | --- | --- |
| O(d(n+r)) | O(d(n+r)) | O(d(n+r)) | O(n+r) | 稳定 |

**代码示例**

**package** sort;

**public** **class** RadixSort {

**private** **static** **void** radixSort(**int**[] array,**int** d)

{

**int** n=1;//代表位数对应的数：1,10,100...

**int** k=0;//保存每一位排序后的结果用于下一位的排序输入

**int** length=array.length;

**int**[][] bucket=**new** **int**[10][length];//排序桶用于保存每次排序后的结果，这一位上排序结果相同的数字放在同一个桶里

**int**[] order=**new** **int**[length];//用于保存每个桶里有多少个数字

**while**(n<d) {

**for**(**int** num:array){ //将数组array里的每个数字放在相应的桶里

**int** digit=(num/n)%10;

bucket[digit][order[digit]]=num;

order[digit]++;

}

**for**(**int** i=0;i<length;i++){//将前一个循环生成的桶里的数据覆盖到原数组中用于保存这一位的排序结果

**if**(order[i]!=0){//这个桶里有数据，从上到下遍历这个桶并将数据保存到原数组中

**for**(**int** j=0;j<order[i];j++){

array[k]=bucket[i][j];

k++;

}

}

order[i]=0;//将桶里计数器置0，用于下一次位排序

}

n\*=10;

k=0;//将k置0，用于下一轮保存位排序结果

}

}

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

**int**[] A=**new** **int**[]{73,22, 93, 43, 55, 14, 28, 65, 39, 81};

*radixSort*(A, 100);

**for**(**int** num:A){

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

}

}

}

## 六、算法试题

### (1)、10亿个数字里里面找最小的10个。

/\*\*

\* 首先建立节点个数为10的最大堆，然后考虑每一个新的值，让他和堆顶比较，

\* 比堆顶大的元素直接抛弃，如果比堆顶小的数字，让他替换堆顶，然后调整堆。

\*/

**public** **class** MaxTenNumber {

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

/\*\*第一个元素0不参与，只是用于占位置,这样的话，只要array[k]>array[2k]

\* && array[k]>array[2k+1]那就是最大堆了，

\* 此外，这里暂时用20个数代替1亿个

\*/

**int**[] array={0,1,2,3,4,7,8,9,10,11,12,13,14,15,16,17,18,19,20,6,5};

//建立建立节点个数为10的最大堆

**for**(**int** i=10/2;i>=1;i--){

*adjustHeap*(array, i, 10);

}

**for**(**int** i=11;i<array.length;i++){

//如果这个元素小于堆顶，和堆顶交换，然后重新调整堆

**if**(array[i]<array[1]){

*swap*(array, i, 1);

*adjustHeap*(array, 1, 10);

}

}

System.***out***.println("最小的10个数字为：");

**for**(**int** i=1;i<=10;i++){

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

}

}

/\*\*

\* 交换

\* **@param** array

\* **@param** i

\* **@param** j

\*/

**private** **static** **void** swap(**int**[] array, **int** i, **int** j) {

**int** tem=array[i];

array[i]=array[j];

array[j]=tem;

}

/\*\*

\* 在以array[head]为根的左右子树是最大堆的前提下把以array[head]为根的树调整为最大堆

\* **@param** array

\* **@param** head

\* **@param** tail

\*/

**static** **void** adjustHeap(**int**[] array,**int** head,**int** tail){

**int** root=array[head];

**int** i=2\*head;

**while**(i<=tail){

**int** max=array[i];

**if**(i+1<=tail)

**if**(array[i+1]>array[i]){

max=array[i+1];

i++;

}

**if**(root>max)

**break**;//别手抖写成了return;

**else**{

array[i/2]=array[i];

}

i\*=2;

}

array[i/2]=root;

}

}

### (2)、有1亿个数字，其中有2个是重复的，快速找到它，时间和空间要最优。

### (3)、2亿个随机生成的无序整数,找出中间大小的值。

### (4)、给一个不知道长度的（可能很大）输入字符串，设计一种方案，将重复的字符排重。

### (5)、遍历二叉树。

### (6)、有3n+1个数字，其中3n个中是重复的，只有1个是不重复的，怎么找出来。

### (7)、写一个字符串（如：www.javastack.cn）反转函数。

### (8)、常用的排序算法，快排，归并、冒泡。 快排的最优时间复杂度，最差复杂度。冒泡排序的优化方案。

### (9)、二分查找的时间复杂度，优势。

### (10)、一个已经构建好的TreeSet，怎么完成倒排序。

### (11)、什么是B+树，B-树，列出实际的使用场景。

### (12)、一个单向链表，删除倒数第N个数据。

### (13)、200个有序的数组，每个数组里面100个元素，找出top20的元素。

### (14)、单向链表，查找中间的那个元素。

### (15)、已知有个rand7（）的函数，返回1到7的随机自然数，怎样利用这个rand7（）构造rand10（）。

要保证rand10()在整数1-10的均匀分布，可以构造一个1-10\*n的**均匀分布**的随机整数区间（n为任何正整数）。假设x是这个1-10\*n区间上的一个随机整数，那么x%10+1就是**均匀分布**在1-10区间上的整数。由于(rand7()-1)\*7+rand7()可以构造出均匀分布在1-49的随机数（原因见下面的说明），可以将41～49这样的随机数剔除掉，得到的数1-40仍然是均匀分布在1-40的，这是因为每个数都可以看成一个独立事件。

下面说明为什么(rand7()-1)\*7+rand7()可以构造出均匀分布在1-49的随机数:首先rand7()-1得到一个离散整数集合{0，1，2，3，4，5，6}，其中每个整数的出现概率都是1/7。那么(rand7()-1)\*7得到一个离散整数集合A={0，7，14，21，28，35，42}，其中每个整数的出现概率也都是1/7。而rand7()得到的集合B={1，2，3，4，5，6，7}中每个整数出现的概率也是1/7。显然集合A和B中任何两个元素组合可以与1-49之间的一个整数一一对应，也就是说1-49之间的任何一个数，可以唯一确定A和B中两个元素的一种组合方式，反过来也成立。由于A和B中元素可以看成是独立事件，根据独立事件的概率公式P(AB)=P(A)P(B)，得到每个组合的概率是1/7\*1/7=1/49。因此(rand7()-1)\*7+rand7()生成的整数均匀分布在1-49之间，每个数的概率都是1/49。

**程序**：

|  |
| --- |
| **int** rand7(){  **int** x=0;  **do** {          x=(rand7()-1)\*7+rand7();      }  **while**(x>40);  **return** x%10+1;  } |

**自我总结：**

核心思想在于构建大于0-10范围的均匀随机数，然后从中取0-10范围的数据。

### 、最长连续字串最大和

**public** **class** MAxSumSubSeq {

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

**int**[] a = {-12, -11, -4, -13, -5, -32};

*getMaxSumSeq*(a);

}

**private** **static** **void** getMaxSumSeq(**int**[] a){

**int** rmax = Integer.***MIN\_VALUE***; // -2147483648

**int** sum = Integer.***MIN\_VALUE***;

**int** start = -1;

**int** end = -1;

**int** temp = -1;

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

**if**(sum>0){

sum+=a[i];

}**else**{

sum = a[i];

temp = i;

}

**if**(sum>rmax){

start = temp;

rmax= sum;

end = i;

}

}

**for**(**int** j = start;j<=end;j++){

System.***out***.print(a[j]+" ");

}

System.***out***.println("\nsum:"+rmax);

}

}

### (17)、双向链表排序时间复杂度平均要是O（nlgn），要避免最坏的O（n^2）的情况。空间复杂度是O（1）.

### (18)、有字符串s和p，判断p是否为s的字串，如果是找出p在s中第一次出现的下标。

# 九、数据库知识

## 一、Oracle+11g+从入门到精通

### 一、Oracle数据库概述

#### 1.oracle11g有四个版本

**企业版**：

优点：高可靠性、高安全性、更好的数据管理、领先一步的商务智能

**标准版**：适用于多大四个插槽的服务器。

优点：多平台自动管理、丰富的开发功能、灵活的定制服务

**标准版1**：适用于两个插槽的服务器

优点：应用服务支持、多平台自动管理、丰富的开发功能、灵活的定制服务

**个人版**：只提供oracle作为DBMS的基本数据库管理服务，适用于单用户开发环境

2.数据库的名称、数据表的定义、数据库账户、权限等信息，通常放在一个“数据字典”中，数据字典是数据库管理系统工作的依据。

3.DBMS可以分为网状型、层次型、关系型、面向对象型，关系型是目前最流行的。

4.用户要对数据库进行操作，是由DBMS把操作从应用程序带到外部级，概念级，再导向内部级进而操作存储器中的数据。

5.数据库系统：一般由数据库、数据库管理系统（及其开发工具）、应用系统和数据库管理员（DBA）构成。

6.在不引起混淆的情况下，将数据库系统简称为数据库。

#### 7.数据库的三级模式为外模式、概念模式和内模式

##### 概念模式

概念模式也称为模式，是对数据库中全局数据逻辑结构的描述，是全体用户公用的数据视图这种描述是一种抽象描述，不涉及具体硬件环境与平台，也与软件环境无关。

##### 外模式

外模式也称为子模式或者用户模式，它是数据库用户能够看见和使用的局部数据的逻辑结构和特征的描述。是数据库用户的数据视图。外模式通常是模式的子集，一个模式可以包含多个外模式，一个应用程序只能使用一个外模式（根据程序端口号判断）

##### 内模式

内模式也称为存储模式，一个模式只有一个内模式，它是数据物理结构和存储方式的描述，定义所有内部记录类型、索引和文件的组织形式，以及数据控制方面的细节

#### 8.数据库的二级映像

##### 外模式/模式映像

每一个外模式都有一个此映像，定义了外模式与模式之间的对应关系。如果模式需要修改，那么只需对此映像修改，而外模式不变，那么应用程序就不必改变。保证数据的逻辑独立性

##### 模式/内模式映像

此映像是唯一的，因为数据库只有一个模式和外模式。当数据库存储结构改变时，可改变此映像，使得模式保持不变。保证数据的物理独立性。

#### 9.数据模型分为：概念数据模型、逻辑数据模型、物理数据模型

##### 概念数据模型

又称概念模型，是一种面向客观世界、面向用户的模型，与具体数据库管理系统无关。

##### 逻辑数据模型

又称为逻辑模型，是一种面向数据库系统的模型：分为层次模型、关系模型、网状模型、面向对象模型。

层次模型用“树结构”表示数据之间的联系

关系模型用“二维表”来表示数据之间的联系。

网状模型用“图结构”来表示数据之间的联系。

面向对象模型用“对象”来表示数据之间的联系。

##### 物理数据模型

又称为物理模型，是数据模型在计算机物理结构上的表示。

#### 10.数据模型由数据结构、数据操纵和完整性约束三部分组成

##### （1）数据完整性约束

**实体完整性**：

若属性A是基本关系R的主属性，则属性A不能取空值。

**参照完整性**：

若属性F是基本关系R的外码，它与基本关系S的主码Ks相对应，则对于R中每个元组在F上的值必须：取空值或者等于S中的某个元组的主码值。

**用户定义完整性**：

#### 11.Oracle基本术语

##### （1）数据字典

存储系统空间，表视图索引、簇、同义词，性能和统计信息，用户名，角色权限等等。

##### （2）数据文件

##### （3）控制文件

是一个很小的二进制文件

##### （4）日志文件

日志模式分为归档模式和非归档模式

##### （5）表空间

一个数据库通常包括SYSTEM、SYSAUX、TEMP三个默认表空间

###### 表空间类型：

系统表空间

永久表空间

临时表空间

撤销表空间

大文件表空间和小文件表空间

###### 表空间的状态：

读写状态

只读状态

脱机状态：不被用户将访问，要访问先设为联机状态

###### 表空间的作用：

控制用户所占用的控件配额

控制数据库所占用的磁盘空间

提高数据库I/O性能

有助于数据库分类管理

##### （6）段

数据段：用于存储表中的所有数据

索引段：用于存储索引的所有数据

临时段：用于存储排序所产生的临时数据

回滚段：用于存储数据被修改之前的值，为事务回滚提供支持。

##### （7）区

##### （8）数据块

是最小的数据管理单位

### 二.Oracle在Windows平台上的安装与配置

### 三.SQL语言基础

#### 1.数据定义

##### create:

创建表：

create table 表名 （

列名 数据类型 [列完整性约束条件]

）;

创建视图：

crate view <视图名>[<列名>,<列名>..]

as<子查询>

[WITH CHECK OPTION]

创建索引：

create [unique][cluster] index<索引名> on <表名>（<列名>[<次序>]...）

unique表示此索引的每一个索引值不能重复，对应唯一的数据记录。

cluster表示要建立的索引是聚簇索引，索引可以建在一个列上，也可以建在多个列上

##### drop：

删除表：

drop table <表名>;

删除视图：

drop view <视图名>;

删除索引：

drop inde <索引名>

##### alter:

alter table<表名>

[add<新列名><数据类型>[完整性约束]]

[drop<完整性约束名>]

[modify<列名><数据类型>]

#### 2.数据查询

##### （1）简单查询

算数表达式

select salary\*(1+0.1)from employees;

distinct关键字

删除重复行

##### （2）where子句

a.条件表达式

A=B

A>B

A<B

A!=B 或者A<>B

A like B

通配符 %代表0个、1个或多个字符\_代表一个字符。

NOT<条件表达式>

b.连接运算符

AND、OR

c.NULL值

##### （3）order by

##### （4）order by 子句

### 四、Oracle PL/SQL语言及编程

#### 1.PL/SQL简介

##### （1）结构:

declare

--声明一些变量、常量、用户定义的数据类型以及游标等

--可选，如不需要可以不写

begin

--主程序体

exception

--异常处理程序

end

--主程序体结束

##### （2）注释：

单行注释：--

多行注释：/\* \*/

##### （3）PL/SQL字符集

合法字符集：

大小写字母

数字

非显示的字符、制表符、空格、回车

数学符号：+-\*/<>=

间隔符：（）[]{}?!:;'"@#$%^&

分界符：

##### （4）PL/SQL数据类型：

###### 数字类型

number:存储整数或浮点数

pls\_integer 只存整数

binary\_interger 只存整数

###### 字符类型

varchar2

char

long

nchar

nvarchar2

###### 日期类型

只有DATE一种类型，世纪、年、月、天、小时、分、秒各占一个字节

###### 布尔类型

type定义的数据类型

type<数据类型名>is<数据类型>

##### （5）PL/SQL变量和常量

定义常量

<常量名>constant<数据类型>:=<值>;

定义变量

<变量名><数据类型>[(宽度):=<初始值>];

变量初始化

未初始化的变量被赋值为NULL

##### （6）PL/SQL语句控制结构

###### 选择结构：

1.if语句

if...then

if...then...else

if...then...elsif

2.case语句

case 检测表达式

when 表达式1 then 语句序列1

when 表达式2 then 语句序列2

[else 其他语句序列]

end；

###### NULL结构：

IS NULL判断字段是否为NULL

###### 循环结构：

loop...exit...end

loop...exit when...end

while...loop...end

for...in...loop...end

goto

##### （7）PL/SQL表达式

字符表达式

并置运算符“||”

‘Hello’||‘World'的值为’Hello World‘

布尔表达式

#### 2.PL/SQL的游标

##### 显示游标：用户声明和操作的一种游标

声明游标：

cursor<游标名>is select<语句>；

打开游标：

open<游标名>；

提取游标：

fetch<游标名>into<变量列表>;

关闭游标：

close<游标名>;

##### 隐式游标：Oracle为所有数据操纵语句自动声明和操纵的一种游标

##### 游标属性：

%found：是否找到游标

%notfound：是否没有找到游标

%rowcount游标行数

%isopen游标是否打开

##### 游标变量：

声明游标变量

打开游标变量

open<游标变量>for<select语句>;

关闭游标变量

#### 3.过程

#### （1）创建过程

create[or replace]procedure<过程名>

（<参数1>,[方式1]<数据类型1>,

<参数2>,[方式2]<数据类型2>

...）

is|as

pl/sql过程体

#### （2）调用过程

execute 过程名；

#### （3）删除过程

drop procedure 过程名；

#### （4）过程参数类型及传递

in参数类型

out参数类型

in out参数类型

#### 4.函数

创建函数

create[or replace]function<>

(<参数1>，[方式1]<数据类型1>)

return<表达式>

is|aspl/sql 程序体

调用函数

删除函数

drop function 函数名

#### 5.程序包

程序包：包含说明部分和包体部分

创建包：

包说明部分

create package<包名>

is

变量、常量及数据类型定义；

游标定义的头部；

函数、过程的定义和参数列表以及返回类型

end<包名>;

包体部分

create package body<包名>

as

游标、函数、过程的具体定义；

end<包名>;

调用包：

variable man\_num NUMBER

execute man\_num:=my\_package.F\_count\_num('M')

删除包：

drop package my\_package

#### 6.触发器

#### （1）触发器的基本原理

触发器类型：

DML触发器：对表或视图执行DML操作时触发

INSTEAD OF触发器：只定义在视图上，用来替换实际的操作语句

系统触发器：对数据库系统进行操作

相关概念：

触发事件

触发条件

触发对象

触发操作

触发时机

条件谓词

触发子类型

#### （2）创建触发器

#### （3）执行触发器

#### （4）删除触发器

### 五、熟悉SQL\*Plus——Oracle数据库环境

#### 1.进入和退出SQL\*Plus环境

（1）启动SQL\*Plus

开始-》程序-》Oracle-OraDb11g\_homel->应用程序开发-》SQL Plus->输入用户名密码

win+R->输入sqlplus “用户名\密码”-》确定

（2）创建SQL\*Plus快捷方式

找到sqlplus.exe创建快捷方式，在快捷方式后面+用户名/密码

（3）退出SQL\*Plus环境

exit或者quit

#### 2.SQL\*Plus编辑器的编辑命令

（1）编辑命令

（2）保存命令

save命令

input命令

edit命令

#### 3.加入注释

（1）remark XXX.sql

（2）/\* \*/

（3）--

#### 4.运行命令

（1）命令行方式

（2）SQL缓冲区方式

（3）命令文件方式

#### 5.编写交互命令

### 十二、数据库控制

#### 1.用事务控制操作

##### （1）什么是事务：

事务是用户定义的一个数据库操作序列，这些操作要么全做，要么不做。可以是一组SQL语句、一条SQL语句或整个程序

事务的特性：

原子性

一致性

隔离性

持久性

事务的状态：

活动状态

部分提交状态

失败状态

提交状态

终止状态

##### （2）设置事务

设置只读事务

set transaction read only;

设置读写事务

set transaction read write；

为事务分配回滚段

set transaction use rollback segment system;

##### （3）事务提交

显示提交：commit

自动提交：set autocommit on

隐式提交：

##### （4）事务回滚

撤销未提交事务中SQL命令对数据所作的修改，已经提交的事务不能进行回滚

\*利用回滚段中存储的数据来撤销未提交事务中SQL命令所作的修改

\*解除对所有数据的事务封锁

\*结束事务

##### （5）设置回退点

回退点又称为保存点，将事务划分成若干小部分，这样就不用回滚整个事务，可以回滚到指定保存点，有更大的灵活性。

\*回滚保存点之后的部分事务

\*删除在该保存点之后建立的全部保存点，该保存点保留，以便多次回滚。

\*解除保存点之后表的封锁或行的封锁。

savepoint sp01;

rollback to sp01;

#### 2.用锁控制并发存取

##### （1）为何加锁

###### 什么是锁

锁是对数据进行并发控制的机制，Oracle使用锁来保证事务的隔离性，锁就是事务可以对数据进行操作的权限。

###### 锁的类型

**按权限分**：

排他锁：

又称为X锁或写锁

共享锁：

又称为S锁或读锁

**按分配的资源分**：

数据锁：

在表中获得并保护数据

字典锁：

防止两个用户同时修改同一个表是结构

内部锁：

分布锁:

并行缓冲管理锁:

查询锁信息

##### （2）加锁的方法

行共享锁RS(Row Share)

Lock table xxx in row share mode;

行排他锁RX(Row Exclusive)

Lock table xxx in row exclusive mode;

共享锁

Lock table xxx in share mode;

共享排他锁

Lock table xxx in share row exclusive mode;

排他锁X

Lock table xxx in exclusive mode;

## 二、SQL试题

### 1.创建数据

#### 学生表Student

create table Student(SId varchar(10),Sname varchar(10),Sage date,Ssex varchar(10));

insert into Student values('01' , '赵雷' , to\_date ('1990-01-01','yyyy-MM-dd') , '男');

insert into Student values('02' , '钱电' , to\_date ('1990-12-21','yyyy-MM-dd') , '男');

insert into Student values('03' , '孙风' , to\_date ('1990-12-20','yyyy-MM-dd') , '男');

insert into Student values('04' , '李云' , to\_date ('1990-12-06','yyyy-MM-dd') , '男');

insert into Student values('05' , '周梅' , to\_date ('1991-12-01','yyyy-MM-dd') , '女');

insert into Student values('06' , '吴兰' , to\_date ('1992-01-01','yyyy-MM-dd') , '女');

insert into Student values('07' , '郑竹' , to\_date ('1989-01-01','yyyy-MM-dd') , '女');

insert into Student values('09' , '张三' , to\_date ('2017-12-20','yyyy-MM-dd') , '女');

insert into Student values('10' , '李四' , to\_date ('2017-12-25','yyyy-MM-dd') , '女');

insert into Student values('11' , '李四' , to\_date ('2012-06-06','yyyy-MM-dd') , '女');

insert into Student values('12' , '赵六' , to\_date ('2013-06-13','yyyy-MM-dd') , '女');

insert into Student values('13' , '孙七' , to\_date ('2014-06-01','yyyy-MM-dd') , '女');

#### 课程表Course

create table Course(CId varchar(10),Cname varchar(10),TId varchar(10));

insert into Course values('01' , '语文' , '02');

insert into Course values('02' , '数学' , '01');

insert into Course values('03' , '英语' , '03');

#### 教师表Teacher

create table Teacher(TId varchar(10),Tname varchar(10));

insert into Teacher values('01' , '张三');

insert into Teacher values('02' , '李四');

insert into Teacher values('03' , '王五');

#### 成绩表SC

create table SC(SId varchar(10),CId varchar(10),score decimal(18,1));

insert into SC values('01' , '01' , 80);

insert into SC values('01' , '02' , 90);

insert into SC values('01' , '03' , 99);

insert into SC values('02' , '01' , 70);

insert into SC values('02' , '02' , 60);

insert into SC values('02' , '03' , 80);

insert into SC values('03' , '01' , 80);

insert into SC values('03' , '02' , 80);

insert into SC values('03' , '03' , 80);

insert into SC values('04' , '01' , 50);

insert into SC values('04' , '02' , 30);

insert into SC values('04' , '03' , 20);

insert into SC values('05' , '01' , 76);

insert into SC values('05' , '02' , 87);

insert into SC values('06' , '01' , 31);

insert into SC values('06' , '03' , 34);

insert into SC values('07' , '02' , 89);

insert into SC values('07' , '03' , 98);

### Sql试题

#### 1.查询" 01 "课程比" 02 "课程成绩高的学生的信息及课程分数

select \* from Student RIGHT JOIN (

select t1.SId, class1, class2 from

(select SId, score as class1 from sc where sc.CId = '01') t1,

(select SId, score as class2 from sc where sc.CId = '02') t2

where t1.SId = t2.SId AND t1.class1 > t2.class2

) r

on Student.SId = r.SId;

#### 2.查询同时选修" 01 "课程和" 02 "课程的情况

select \* from

(select \* from sc where sc.CId = '01') t1,

(select \* from sc where sc.CId = '02') t2

where t1.SId = t2.SId;

#### 查询选修" 01 "课程但可能没修" 02 "课程的情况(没修时显示为 null )

select \* from

(select \* from sc where sc.CId = '01') t1

left join

(select \* from sc where sc.CId = '02') t2

on t1.SId = t2.SId;

#### 4.查询没修" 01 "课程但修" 02 "课程的情况

select \* from sc

where sc.SId not in (

select SId from sc

where sc.CId = '01'

)

AND sc.CId= '02';

#### 5.查询平均成绩大于等于 60 分的同学的学生编号和学生姓名和平均成绩

select student.SId,sname,ss from student,(

select SId, AVG(score) as ss from sc

GROUP BY SId

HAVING AVG(score)> 60

)r

where student.sid = r.sid;

#### 6.查询所有同学的学生编号、学生姓名、选课总数、所有课程的成绩总和，不显示没选课的学生

select student.sid, student.sname,r.coursenumber,r.scoresum

from student,

(select sc.sid, sum(sc.score) as scoresum, count(sc.cid) as coursenumber from sc

group by sc.sid)r

where student.sid = r.sid;

#### 7.查询所有同学的学生编号、学生姓名、选课总数、所有课程的成绩总和，显示没选课的学生

select s.sid,s.sname from student s

left join

(select

sc.sid, sum(sc.score) as scoresum, count(sc.cid) as coursenumber

from sc

group by sc.sid

)r

on s.sid = r.sid

#### 查有成绩的学生信息

select \* from student where exists (select sc.sid from sc where student.sid = sc.sid);

#### 9.查询学过「张三」老师授课的同学的信息

select student.\* from student,teacher,course,sc

where

student.sid = sc.sid

and course.cid=sc.cid

and course.tid = teacher.tid

and tname = '张三';

#### 10.查询没有学全所有课程的同学的信息

select \* from student

where student.sid not in (

select sc.sid from sc

group by sc.sid

having count(sc.cid)= (select count(cid) from course)

);

#### 11.查询至少有一门课与学号为" 01 "的同学所学相同的同学的信息

select \* from student

where student.sid in (

select sc.sid from sc

where sc.cid in(

select sc.cid from sc

where sc.sid = '01'

)

);

#### 12.查询和" 01 "号的同学学习的课程完全相同的其他同学的信息

select \* from student where sid in

(select sid from SC where sid not in

(select sid from SC where cid not in (select cid from SC where sid='01'))

group by sid

having count(\*)=(select count(\*) from SC where sid='01') and sid != '01');

#### 13.查询没学过"张三"老师讲授的任一门课程的学生姓名

select \* from student

where student.sid not in(

select sc.sid from sc where sc.cid in(

select course.cid from course where course.tid in(

select teacher.tid from teacher where tname = '张三'

)

));

#### 14.查询两门及其以上不及格课程的同学的学号，姓名及其平均成绩

select s.sid, s.sname, b.avgScore from (select sc.sid,AVG(sc.score) as avgScore from sc

where sc.score<60

group by sc.sid

having count(\*)>1) b left join student s on b.sid= s.sid;

#### 15.按平均成绩从高到低显示所有学生的所有课程的成绩以及平均成绩

select \* from sc

left join (

select sid,avg(score) as avscore from sc

group by sid

)r

on sc.sid = r.sid

order by avscore desc;

#### 16.查询各科成绩最高分、最低分和平均分,及格率，中等率，优良率，优秀率

select

sc.CId ,

max(sc.score)as 最高分,

min(sc.score)as 最低分,

AVG(sc.score)as 平均分,

count(\*)as 选修人数,

sum(case when sc.score>=60 then 1 else 0 end )/count(\*)as 及格率,

sum(case when sc.score>=70 and sc.score<80 then 1 else 0 end )/count(\*)as 中等率,

sum(case when sc.score>=80 and sc.score<90 then 1 else 0 end )/count(\*)as 优良率,

sum(case when sc.score>=90 then 1 else 0 end )/count(\*)as 优秀率

from sc

GROUP BY sc.CId

ORDER BY count(\*)DESC, sc.CId ASC

#### 17.按各科成绩进行排序,并显示排名,Score 重复时保留名次空缺

select a.cid, a.sid, a.score, count(b.score)+1 as rank

from sc a

left join sc b

on a.score<b.score and a.cid = b.cid

group by a.cid, a.sid,a.score

order by a.cid, rank ASC;

#### 查询学生的总成绩，并进行排名，总分重复时不保留名次空缺

## 三、数据库试题

### 1、数据库隔离级别有哪些，各自的含义是什么，MYSQL默认的隔离级别是是什么。

① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

③ Read committed (读已提交)：可避免脏读的发生。

④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

　在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

### 什么是脏读、不可重复读、幻读。

**脏读**:是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

**不可重复读**：是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

**幻读**:是事务非独立执行时发生的一种现象。例如事务T1对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作，这时事务T2又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。而操作事务T1的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务T2中添加的，就好像产生幻觉一样，这就是发生了幻读。幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而**幻读针对的是一批数据整体**（比如数据的个数）。

### MYSQL有哪些存储引擎，各自优缺点。

MYSQL存储引擎种类（了解）：MyISAM、InnoDB、BDB、MEMORY、MERGE、EXAMPLE、NDBCluster、ARCHIVE、CSV、BLACKHOLE、FEDERATED等。在**Oracle 和SQL Server等数据库中只有一种存储引擎**，所有数据存储管理机制都是一样的。而MySql数据库提供了多种存储引擎。用户可以根据不同的需求为数据表选择不同的存储引擎，用户也可以根据自己的需要编写自己的存储引擎。简单的说下什么是存储引擎，存储引擎说白了就是如何存储数据、如何为存储的数据建立索引和如何更新、查询数据等技术的实现方法。因为在关系数据库中数据的存储是以表的形式存储的，所以存储引擎也可以称为**表类型**（即存储和操作此表的类型）。

【1】MyISAM： ①、拥有较高的插入，查询速度。②、不支持事务，行级锁和外键约束的功能。③、使用表级锁，并发性能差。④、主机宕机后，MyISAM表易损坏，灾难恢复性不佳。⑤、可以配合锁，实现操作系统下数据的复制备份、迁移。⑥、只缓存索引，数据的缓存是通过操作系统缓存区来实现的，可能引发过多的系统调用且效率不佳。⑦、数据紧凑存储，因此可获得更小的索引和更快的全表扫描性能。

【2】InnoDB：5.5版本后Mysql的默认数据库，事**务型数据库的首选引擎**，①、支持ACID事务。②、支持行级锁定。③、灾难恢复性好。④、支持外键关联。⑤、支持热备份。⑥、对于InnoDB引擎中的表，其数据的物理组织是簇表（Cluster Table），主键索引和数据是在一起的，数据按主键的顺序物理分布。⑦、实现了缓冲管理，不仅能缓冲索引也能缓冲数据，并且能够自动创建散列索引以加快数据的获取。

【3】MEMORY：①、所有数据置于内存的存储引擎，拥有极高的插入，更新和查询效率。但是会占用和数据量成正比的内存空间。②、其内容会在Mysql重新启动时丢失，复制维护时需要小心。③、使用表级锁，虽然内存访问速度快，但是频繁的读写，表级锁会成为瓶颈。④、只支持固定大小的行，varchar类型的字段会存储为固定长度的Char类型，浪费空间。⑤、不支持TEXT、BLOB字段，当有些查询需要使用临时表时（因为是存在于内存中，所以这种类型常应用于临时表中），如果表中有TEXT、BLOB字段，那么会转换为基于磁盘的MyISAM表，严重降低性能。⑥、由于内存资源成本比较昂贵，一般不建议设置过大的内存表，如果内存表满了，可通过清除数据或调整内存表参数来避免报错。⑦、MEMORY表在所有客户端之间共享。

### 4、高并发下，如何做到安全的修改同一行数据。

使用排他锁：

set autocommit =0;*--关闭自动提交*

begin;*--开始事务*

select \* from order where id=989879 for update;*--查询信息*

update order set name='names';*--修改信息*

commit;*--提交事务*

select \* from t for update;*-- 会等待行锁释放之后，返回查询结果。*

select \* from t for update nowait;*-- 不等待行锁释放，提示锁冲突，不返回结果*

select \* from t for update wait 5 ;*-- 等待5秒，若行锁仍未释放，则提示锁冲突，不返回结果*

select \* from t for update skip locked;*-- 查询返回查询结果，但忽略有行锁的记录*

使用共享锁：

set autocommit =0;*--关闭自动提交*

begin;*--开始事务*

select \* from order where id=989879 lock in share mode;*--锁定查询的字段*

update order set name='names';*--修改信息*

commit;*--提交事务*

一般情况下推荐使用select…for update。使用lock in share mode时需要注意死锁的问题。一般的select是不需要考虑数据是否被锁定，最多根据多版本一致读的特性读取之前的版本。加入for update之后，Oracle就要求启动一个新事务，尝试对数据进行加锁。如果当前已经被加锁，默认的行为必然是block等待。使用nowait子句的作用就是避免进行等待，当发现请求加锁资源被锁定未释放的时候，直接报错返回。  
在日常中，我们对for update的使用还是比较普遍的，特别是在如pl/sql developer中手工修改数据。此时只是觉得方便，而对for update真正的含义缺乏理解。  
 for update是Oracle提供的**手工提高锁级别和范围**的特例语句。Oracle的锁机制是目前各类型数据库锁机制中比较优秀的。所以，Oracle认为一般不需要用户和应用直接进行锁的控制和提升。甚至认为死锁这类锁相关问题的出现场景，大都与手工提升锁有关。所以，Oracle并不推荐使用for update作为日常开发使用。而且，在平时开发和运维中，使用了for update却忘记提交，会引起很多锁表故障。  
 那么，什么时候需要使用for update？就是那些需要业务层面数据独占时，可以考虑使用for update。场景上，比如火车票订票，在屏幕上显示邮票，而真正进行出票时，需要重新确定一下这个数据没有被其他客户端修改。所以，在这个确认过程中，可以使用for update。这是统一的解决方案方案问题，需要前期有所准备。

### 乐观锁和悲观锁是什么，INNODB的标准行级锁有哪2种，解释其含义。

**乐观锁**（Optimistic Concurrency Control，缩写”OCC”）：是一种并发控制的方法。乐观的认为多用户并发的事务在处理时不会彼此互相影响，各事务能够在使用锁的情况下处理各自的数据。在提交更新数据之前，每个事务会先检查该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，正在提交的事务会进行回滚。不过，当需求多为更新数据时，就会增大数据之间的冲突，也就增大CPU的计算开销，此时不建议使用。

**数据是否修改的标准是**：在表中的数据进行操作时，先给数据表最新的数据加一个版本(version)字段，每操作一次，将那条记录的版本号加1。也就是先查询出那条记录，获取出version字段,修改完数据后准备提交之前,先判断此刻version的值是否与刚刚查询出来时的version的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，将version字段的值加1；如果更新时发现此刻的version值与刚刚获取出来的version的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。

**悲观锁**（Pessimistic Concurrency Control，缩写”PCC”）：与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。所以悲观并发控制主要用于数据争用激烈的环境，以及发生并发冲突时使用锁保护数据的成本要低于回滚事务的成本的环境中。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。要使用悲观锁，我们必须关闭mysql数据库的自动提交属性。

**InnoDB的标准行级锁有哪2种**：

**共享锁**：共享锁指的就是对于多个不同的事务，对同一个资源共享同一个锁，在执行语句后面加上lock in share mode就代表对某些资源加上共享锁。

**排它锁**：排它锁与共享锁相对应，就是指对于多个不同的事务，对同一个资源只能有一把锁。在需要执行的语句后面加上for update就可以了。对于update,insert,delete语句会自动加排它锁。

### SQL优化的一般步骤是什么，怎么看执行计划，如何理解其中各个字段的含义。

#### Mysql

##### 查看执行计划

explain+sql

##### 各字段含义

###### ID

id列中的数据为一组数字，表示执行select语句顺序

id值相同时，执行顺序由上至下

id值越大优先级越高，越先被执行

###### SELECT\_TYPE

查询的类型，主要是用于区分普通查询、联合查询、子查询等复杂的查询。

**SIMPLE**：简单的select查询，查询中不包含子查询或者union

**PRIMARY**：查询中包含任何复杂的子部分，最外层查询则被标记为primary

**SUBQUERY**：在select 或 where列表中包含了子查询

**DERIVED**：在from列表中包含的子查询被标记为derived（衍生），mysql会递归执行这些子查询，把结果放在临时表里

**UNION**：若第二个select出现在union之后，则被标记为union；若union包含在from子句的子查询中，外层select将被标记为derived

**UNION RESULT**：从union表获取结果的select

**DEPENDENT SUBQUERY**: 依赖外部结果的子查询

**DEPENDENT UNION:** 当union作为子查询时，第二或是第二个后的查询的 select\_type值

###### TABLE

输出数据行所在的表的名称

< unionM,N>由ID为M,N查询union产生的结果集

< derivedN>/< subqueryN> 由Id为n的查询产生的结果

###### PARTITIONS

对于分区表，显示查询的分区ID

对于非分区表，显示的为 NULL

###### TYPE

访问类型，sql查询优化中一个很重要的指标，结果值从好到坏依次是：

system > const > eq\_ref > ref > fulltext > ref\_or\_null > index\_merge > unique\_subquery > index\_subquery > range > index > ALL

**system** 这是const联接类型的一个特例，当查询的表只有一行时使用

**const**：表示通过索引一次就找到了，const用于比较primary key 或者 unique索引。因为只需匹配一行数据，所有很快。如果将主键置于where列表中，mysql就能将该查询转换为一个const

**eq\_ref**：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键 或 唯一索引扫描。

**ref**：非唯一性索引扫描，返回匹配某个单独值的所有行。本质是也是一种索引访问，它返回所有匹配某个单独值的行，然而他可能会找到多个符合条件的行，所以它应该属于查找和扫描的混合体

**ref\_or\_null：**类似于 ref类型的查询，但是附加了对 null 值列的查询

**range：**只检索给定范围的行，使用一个索引来选择行。key列显示使用了那个索引。一般就是在where语句中出现了between、<、>、in等的查询。**这种索引列上的范围扫描比全索引扫描要好**。只需要开始于某个点，结束于另一个点，不用扫描全部索引

**index**：Full Index Scan，index与ALL区别为index类型只遍历索引树。这通常为ALL块，应为索引文件通常比数据文件小。（Index与ALL虽然都是读全表，但index是从索引中读取，而ALL是从硬盘读取）

ALL：Full Table Scan，遍历全表以找到匹配的行

###### POSSIBLE\_KEYS

指出mysql能使用哪些索引来优化查询

查询列所涉及的列上的索引都会被列出来，但不一定会被使用

###### KEY

实际使用的索引，如果为NULL，则没有使用索引。

查询中如果使用了覆盖索引，则该索引仅出现在key列表中。

###### KEY\_LEN

表示索引字段的最大的可能长度

key\_len的长度由字段定义计算而来，并非数据的实际长度

###### REF

表示那些列或常量被用于查找索引列上的值

###### ROWS

表示msql通过索引的统计信息，估算的所需读取的行数

rows值的大小时个统计抽样的结果，并不十分的准确

###### FILTERED

表示返回的结果的行数占需要读取行数的百分比

Filtered列的值越大越好

Filtered列的值依赖的说统计信息

###### EXTRA

Distinct 优化distinct操作，在找到第一匹配的元祖后即停止找同样的动作

Not exists 使用 not exists 来优化查询

Using filesort 使用额外操作进行排序，通常会出现在order by 或 group by 查询

Using index 使用了覆盖索引进行查询

Using temporary mysql需要使用临时表来处理查询，常见于排序，子查询，和分组查询

Using where 需要在mysql服务器层使用 where条件来过滤数据

slect tables /optimized away 直接通过索引来获得数据，不用访问表

### 8、MYsql的索引原理，索引的类型有哪些，如何创建合理的索引，索引如何优化。

### 9、聚集索引和非聚集索引的区别。

### 10、select for update 是什么含义，会锁表还是锁行或是其他。

### 11、为什么要用Btree实现，它是怎么分裂的，什么时候分裂，为什么是平衡的。

### 12、数据库的ACID是什么。

### 13、某个表有近千万数据，CRUD比较慢，如何优化。

### 14、Mysql怎么优化table scan的。

### 15、如何写sql能够有效的使用到复合索引。

### 16、mysql中in 和exists 区别。

### 17、数据库自增主键可能的问题。

### 18、MVCC的含义，如何实现的。

### 19、你做过的项目里遇到分库分表了吗，怎么做的，有用到中间件么，比如sharding jdbc等,他们的原理知道么。

### 20、MYSQL的主从延迟怎么解决。

# 十、消息队列

## 1、消息队列的使用场景。

## 2、消息的重发，补充策略。

## 3、如何保证消息的有序性。

## 4、用过哪些MQ，和其他mq比较有什么优缺点，MQ的连接是线程安全的吗，你们公司的MQ服务架构怎样的。

## 5、MQ系统的数据如何保证不丢失。

## 6、rabbitmq如何实现集群高可用。

## 7、kafka吞吐量高的原因。

## 8、kafka 和其他消息队列的区别，kafka 主从同步怎么实现。

## 9、利用mq怎么实现最终一致性。

## 10、使用kafka有没有遇到什么问题，怎么解决的。

## 11、MQ有可能发生重复消费，如何避免，如何做到幂等。

## 12、MQ的消息延迟了怎么处理，消息可以设置过期时间么，过期了你们一般怎么处理。

# 十一、缓存

## Redis开发与运维

### 初始redis

#### Redis八个特性

速度快、基于键值对的数据结构服务器、丰富的功能、简单稳定、客户端语言多、持久化、主从复制、高可用和分布式。

#### Redis可以做什么

缓存、排行榜系统、计数器应用、社交网络、消息队列系统。

#### 安装只需6步

$ wget http://download.redis.io/releases/redis-3.0.7.tar.gz  
$ tar xzf redis-3.0.7.tar.gz  
$ ln -s redis-3.0.7 redis   
$ cd redis  
$ make  
$ make install

第三步建立软连接是为了不把redis指定到固定版本上，有利于将来升级，这是安装软件的好习惯。

#### Redis关闭

Redis关闭的过程： 断开与客户端的连接、 持久化文件生成， 是一种相对优雅的关闭方式。

除了可以通过shutdown命令关闭Redis服务以外， 还可以通过kill进程号的方式关闭掉Redis， 但是不要粗暴地使用kill-9强制杀死Redis服务， 不但不会做持久化操作， 还会造成缓冲区等资源不能被优雅关闭， 极端情况会造成AOF和复制丢失数据的情况。

shutdown还有一个参数， 代表是否在关闭Redis前， 生成持久化文件 ：redis-cli shutdown nosave|save

#### redis版本

借鉴了linux的命名规则，版本第二位如果时奇数则为非稳定版本，偶数为稳定版本，当前奇数版本就是下一偶数版本的开发版本，所以线上常选择偶数版本，奇数版本可以用来提前体验新特性。

### 第二章、API的理解与使用

#### 全局命令

**keys \*** :输出所有的键

**dbsize**: 输出键总数

**exists key**： 检查键是否存在，存在返回1，不存在返回0.

**del kely**： 删除键

**expire key seconds**：设置键过期时间

**ttl key**：大于0为剩余过期时间，-1为没设置过期时间，-2键不存在。

**type key:**返回键类型。

#### 数据结构和内部编码



#### 字符串

##### 命令

set hello world #设置键为hello值为world的键值对

extists hello #键hello不存在时返回0

setnx hello redis #hello不存在成功，已存在则返回0

setxx hello redis #hello已存在则设置成功，不存在则返回0

get hello #获取hello值，获取不存在的键则返回nil(空)

mset a 1 b 2 c 3 d 4 #批量设置键值对

mget a b c d #批量获取值

incr hello #值自增，用于计数；值不是整数返回错误，键不存在当0处理，返回1

decr hell #值自减

incrby hello 3 #自增指定数字

decrby hello 3 #自减指定数字

incrbyfloat hello 3.0 #自增浮点数

append hello redis #字符串尾部追加值

strlen hello #字符串长度

getset hello world #设置值为world并返回原值

setrange hello 0 a #设置值的0位置为a

getrange hello 0 2 #返回值0到2的位置

##### 内部编码

字符串类型的内部编码有3种：

int：8个字节的长类型

embstr：小于等于39字节的字符串

raw：大于39字节的字符串

##### 典型应用

###### 缓存

UserInfo getUserInfo(**long** id){

userRedisKey = "user:info:" + id

value = redis.get(userRedisKey);

UserInfo userInfo;

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

userInfo = deserialize(value);

} **else** {

userInfo = mysql.get(id);

**if** (userInfo != **null**)

redis.setex(userRedisKey, 3600, serialize(userInfo));

}

**return** userInfo;

}

###### 计数

**long** incrVideoCounter(**long** id) {

key = "video:playCount:" + id;

**return** redis.incr(key);

}

#### 哈希

##### 命令

hset user:1 name tom #设置user 1 name为tom

hget user:1 name #获取user 1的name属性

hdel user:1 name #删除user 1的name属性

hlen user:1 #获取user 1的属性个数

hmset user:1 name tom age 20 city Beijing #批量设置属性值

hmget user:1 name age city #批量获取属性值

hexists user:1 name #判断属性是否存在

hkeys user:1 #获取所有属性名

hvals user:1 #获取所有属性值

hgetall user:1 #获取所有属性名属性值

hincrby user:1 name #自增

hincrbyfloat user:1 name #浮点自增

hstrlen user:1 name #获取字符串长度

##### 内部编码

当哈希类型元素个数小于hash-max-ziplist-entries配置（默认512个） 、同时所有值都小于hash-max-ziplist-value配置（默认64字节） 时， Redis会使用**ziplist**作为哈希的内部实现，不满足此二条件时，会使  
用**hashtable**作为哈希的内部实现。

#### 列表

##### 命令

lpush/rpush listkey c b a #从左向右/从右向左插入元素

lrange listkey 0-1 #从左向右获取所有元素

linsert listkey before/after b c #在b元素前/后插入c

lindex listkey -1 #获取指定位置的元素

llen listkey #获取当前列表长度

lpop/rpop listkey #删除最左/右边的元素

lrem listkey count a #count=0：删除所有；>0从左到右删除count个元素；<0从右向左删除|count|个元素

ltrim listkey 1 3 #删除第2-4个元素

lset listkey 2 python #修改第三个元素为python

##### 内部编码

当列表的元素个数小于list-max-ziplist-entries配置（默认512个） ， 同时列表中每个元素的值都小于list-max-ziplist-value配置时（默认64字节） ， Redis会选用**ziplist**来作为列表的内部实现来减少内存的使  
用。当列表类型无法满足ziplist的条件时， Redis会使用**linkedlist**作为列表的内部实现

#### 集合

集合中**不允许有重复元素**， 并且集合中的元素是**无序**的， 不能通过索引下标获取元素。

##### 命令

sadd myset a b c #添加元素

srem myset a b #删除元素

scard myset #计算元素个数

sismember myset c #元素是否存在，存在返回1，否则返回0

srandmember myset 2 #随机返回2个元素

spop myset #随机弹出一个元素

smembers myset #所有元素

sinter user:1:follow uer:2:follow #求两个集合交集

sunion user:1:follow uer:2:follow #求两个集合并集

sdiff user:1:follow uer:2:follow #求两个集合差集

sinterstore user:1\_2:inter user:1:follow uer:2:follow #求两个集合交集并保存

sunionstore user:1\_2:union user:1:follow uer:2:follow #求两个集合并集

sdiffstore user:1\_2:idiff user:1:follow uer:2:follow #求两个集合差集

##### 内部编码

当集合中的元素都是整数且元素个数小于set-maxintset-entries配置（默认512个）时，Redis会选用intset来作为集合的内部实现，从而减少内存的使用。当集合类型无法满足intset的条件时， Redis会使  
用hashtable作为集合的内部实现。

#### 有序集合

##### 命令

zadd user:ranking 251 tom #添加成员

zcard user:ranking #计算成员个数

zscore user:ranking tom #计某成员的分数

zrank user:ranking tom #计算某成员排名

zrem user:ranking tom #删除某成员

zincrby user:ranking 9 tom #给tom增加9分

### 第五章、持久化

Redis支持RDB和AOF两种持久化机制。

#### RDB

##### 触发机制

RDB持久化是把当前进程数据生成快照保存到硬盘的过程，触发RDB持久化过程分为**手动触发**和**自动触发**。

**手动触发：**分别对应**save**和**bgsave**命令，**save命令阻塞**当前Redis服务器， 直到RDB过程完成为止， 对于内存比较大的实例会造成长时间阻塞， 线上环境不建议使用。**bgsave命令**Redis进程执行fork操作创建子进程， RDB持久化过程由子进程负责，完成后自动结束。阻塞只发生在fork阶段， 一般时间很短。

**自动触发：**使用save相关配置、从节点执行全量复制、执行debug reload、执行shutdown且没开启AOF持久化。

##### RDB文件的处理

RDB文件保存在dir配置指定的目录下，文件名通过dbfilename配置指定。可以通过执行**config set dir**{newDir}和c**onfig set dbfilename**{newFileName}运行期动态执行。Redis默认采用LZF算法对生成的RDB文件做压缩处理，压缩后的文件远远小于内存大小， 默认开启，可以通过参数**config set rdbcompression**{yes|no}动态修改。

##### RDB优缺点

RDB是一个紧凑压缩的二进制文件，代表Redis在某个时间点上的数据快照。非常适用于备份，全量复制等场景。RDB方式数据没办法做到实时持久化/秒级持久化。因为bgsave每次运行都要执行fork操作创建子进程，属于重量级操作，频繁执行成本过高。

#### AOF

AOF（ append only file） 持久化AOF的主要作用是解决了数据持久化的实时性， 目前已经是Redis持久化的主流方式。

### 第八章、理解内存

执行info memory命令获取内存相关指标。

used\_memory:存储所有数据内存占用量

used\_memory\_rss：redis进程占用操作系统内存

mem\_fragmentation\_ratio：碎片率，等于used\_memory\_rss/ used\_memory，大于1时此值越大说明碎片化越严重；小于1时一般出现在操作系统把Redis内存交换（Swap） 到硬盘导致，此时Redis性能会变得很差， 甚至僵死。

reidis内存消耗主要包括：自身内存+对象内存+缓冲内存+内存碎片。

**对象内存**是Redis内存占用最大的一块， 存储着用户所有的数据。**缓冲内存**主要包括：客户端缓冲、复制积压缓冲区、AOF缓冲区。**内存碎片**：Redis默认的内存分配器采用jemalloc， 可选的分配器还有： glibc、tcmalloc。分配内存策略一般采用固定范围的内存块进行分配。方便管理和重复利用，也带来了内存碎片。当存储数据参差不齐时频繁更新数据和键过期删除会造成大量内存碎片。解决办法：尽量数据对齐、安全重启。

排查当前系统是否支持并开启THP， 如果开启建议关闭， 防止copy-on write期间内存过度消耗。

## 二、试题

### 1、常见的缓存策略有哪些，如何做到缓存(比如redis)与DB里的数据一致性。

缓存策略的分类：

    (1)、**基于访问的时间**：此类算法按各缓存项被访问时间来组织缓存队列，决定替换对象。如LRU

(2)、 **基于访问频率**：此类算法用缓存项的被访问频率来组织缓存。如LFU、LRU2、2Q、LIRS。

(3)、**访问时间与频率兼顾**：通过兼顾访问时间和频率。使得数据模式在变化时缓存策略仍有较好性能。如FBR、LRUF、ALRFU。多数此类算法具有一个可调或自适应参数，通过该参数的调节使缓存策略在基于访问时间与频率间取得一个平衡。

(4)、**基于访问模式**：某些应用有较明确的数据访问特点，进而产生与其相适应的缓存策略。如专用的VoD系统设计的A&L缓存策略，同时适应随机、顺序两种访问模式的SARC策略。

### 2、你们项目中用到了什么缓存系统，如何设计的。

### 3、如何防止缓存击穿和雪崩。

**缓存穿透**：缓存穿透是说收到一个请求，但是该请求缓存中不存在，只能去数据库中查询，然后放进缓存。但当有好多请求同时访问同一个数据时，业务系统把这些请求全发到了数据库；或者恶意构造一个逻辑上不存在的数据，然后大量发送这个请求,这样每次都会被发送到数据库，最总导致数据库挂掉。

     解决办法：对于恶意访问，一种思路是先做校验，对恶意数据直接过滤掉，不要发送至数据库层；第二种思路是缓存空结果，就是对查询不存在的数据也记录在缓存中，这样就可以有效的减少查询数据库的次数。非恶意访问，结合缓存击穿说明。

**缓存击穿**：上面提到的某个数据没有，然后好多请求查询数据库，可以归为缓存击穿的范畴：对于热点数据，当缓存失效的一瞬间，所有的请求都被下放到数据库去请求更新缓存，数据库被压垮。

     解决办法：防范此类问题，一种思路是加全局锁，就是所有访问某个数据的请求都共享一个锁，获得锁的那个才有资格去访问数据库，其他线程必须等待。但现在大部分系统都是分布式的，本地锁无法控制其他服务器也等待，所以要用到全局锁，比如Redis的setnx实现全局锁。另一种思想是对即将过期的数据进行主动刷新，比如新起一个线程轮询数据，或者比如把所有的数据划分为不同的缓存区间，定期分区间刷新数据。第二个思路与缓存雪崩有点关系。

**缓存雪崩**：缓存雪崩是指当我们给所有的缓存设置了同样的过期时间，当某一时刻，整个缓存的数据全部过期了，然后瞬间所有的请求都被抛向了数据库，数据库就崩掉了。

     解决办法：解决思路要么是分治，划分更小的缓存区间，按区间过期；要么给每个key的过期时间加一个随机值，避免同时过期，达到错峰刷新缓存的目的。

**缓存刷新**：既清空缓存，一般在insert、update、delete操作后就需要刷新缓存，如果不执行就会出现脏数据。但当缓存请求的系统崩掉后，返回给缓存的值为null。

### 4、缓存数据过期后的更新如何设计。

### 5、redis的list结构相关的操作。

### 6、Redis的数据结构都有哪些。

### 7、Redis的使用要注意什么，讲讲持久化方式，内存设置，集群的应用和优劣势，淘汰策略等。

### 8、redis2和redis3的区别，redis3内部通讯机制。

### 9、当前redis集群有哪些玩法，各自优缺点，场景。

### 10、Memcache的原理，哪些数据适合放在缓存中。

### 11、redis和memcached 的内存管理的区别。

### 12、Redis的并发竞争问题如何解决，了解Redis事务的CAS操作吗。

### 13、Redis的选举算法和流程是怎样的。

### 14、redis的持久化的机制，aof和rdb的区别。

### 15、redis的集群怎么同步的数据的。

### 16、知道哪些redis的优化操作。

### 17、Reids的主从复制机制原理。

### 18、Redis的线程模型是什么。

### 19、请思考一个方案，设计一个可以控制缓存总体大小的自动适应的本地缓存。

### 20、如何看待缓存的使用（本地缓存，集中式缓存），简述本地缓存和集中式缓存和优缺点。

### 21、本地缓存在并发使用时的注意事项。

# 十二、全文检索

## 一.Solr

### 1.什么是全文检索

磁盘文件检索（非结构化数据）

从非结构化的数据中提供一部分数据再重新组织数据叫索引

先创建索引，再对索引进行搜索的过程叫全文检索

lucene是apache提供的开源的全文检索工具包

**创建索引**：原始文档-》获取文档-》创建文档对象-》分析文档-》创建索引（写入索引库）

**查询索引**：创建索引接口，创建查询索引对象，执行查询，渲染结果

中文分词器

标准分析器。中日韩分析器，smart分析器，ik分析器

索引的维护，增删改查

### 2.solor的安装及其配置

solor是一个war包，放到tomcat下运行

jetty solor内置服务器，不如tomcat稳定

lucene工作量较大，主要表现在，索引维护、索引性能优化、搜索性能优化，不建议使用。

solor是apache下顶级开源项目，java开发的。

solor实现索引方法：用post方法向服务器发送一个描述Filed及其内容的xml文档，solor根据xml文档添加、删除、更新索引，solor只接受http get请求，返回xml，json格式的数据

lucene是开源代码的全文检索工具包，不是一个完整的搜索引擎，solor是一个搜索引擎服务

#### 数据库

关系型数据库

oracle

mySql

非关系型数据库

noSql:redis

索引库

#### solor下载

官方下载

#### 目录结构

bin：solor.cmd双击启动solor ，走jetty服务器

contrib：插件目录，第三方插件

dist: jar包，war包

docs：文档

example：例子

licenses：许可说明书

#### 搭建solor服务器

tomact服务有两个放置jar包地方，应用的lib下和tomcat自身的lib

web.xml是tomcat的入口，weblogic.xml是weblogic的入口

### 3.solor后台管理界面

Dashboard:仪表盘，展示各种参数、版本

Logging：日志

core Admin:数据库管理；一个数据库一个core，添加core失败可以手动复制核文件夹，并修改里面core.property文件中的名字

java properties

Thread Dump:曾经走过哪些进程

下拉框选核

overView:

analysis:分词器，默认标准分词器

dataImport:将关系型数据库数据导入索引库

documents: post:增删改，get：查 支持json、xml、csv

Files:配置文件

ping：测试solor服务器是否存活

query:

schema Browser:

### 4.schema.xml中域及动态域

filed：域 name,type，indexd，stored，required,multiValued(多值，分割),omitNorms，

dynamicField:动态域（没有配置过的域不能添加使用，动态域使用通配符\*\_i,\*\_s.....）

copyField:复制域 ：合并多个域，用来一次搜索多个域

fieldType:域类型

name:

class:

solr.strField

solr.BoolField

solor.TrieIntField

solr.TrieFloatField

solr.TrieLongField

solor.TrieDoubleField

solor.TextField

precisionStep:

positionIncrementGap:

### 5.配置中文分词器

1、把ikAnalyzer.jar添加到solor/WEB\_INF/lib目录下

2、复制ikAnalyzer的配置文件和自定义词典和停用词典到solor的classpath下

3、在schema.xml添加一个自定义fieldType,使用中文分析器

<fieldType name=*"text\_Ik"* class=*"solr.TextField"*>

<analyzer class=*"ora.wlter.analyzer.luncene,IKAnalyzer"*>

<fieldType>

4、定义 field，制定 type为text\_IK

<field name=*"title\_ik"* type=*"text\_ik"* index=*"true"* stored=*"true"*>

<field name=*"content\_ik"* type=*"text\_ik"* index=*"true"* stored=*"false"* multiValued=*"true"*>

### 6.solor后台管理界面增删改查

XML

<delete>

<query>id:change:me</query>

</delete>

<commit/>

### 7.管理界面导入数据

1、把dataimport插件依赖的jar包添加到solorcore（collection1\lib）中

2、数据库驱动包

3、配置solorConfig.xml 创建

data-config.xml配置数据库连接，用户密码，映射字段到域

4、管理界面 dataimport execute

### 8.管理界面query查询

两关键词： product\_name:幸福 AND product\_catalog\_name:环保餐具

一个关键词，加过滤条件

product\_price:[\* TO 10] //无穷小到10元

sort排序 product\_price asc

start,rows 开始，条数

fl:指定搜索字段

df:默认搜索字段（域）

wt:输出格式

hl:高亮

### Java代码添加数据

String baseUrl="http://localhost:8080/solor/connection1";

SolorServer solorServer = HttpSolorServer(baseUrl);

sorlorInputDocument doc= **new** sorlorInputDocument ();

doc.setField("id","111")

doc.setField("name","fabingbing")

solorServer .add(doc);

### 10.java查询

String baseUrl="http://localhost:8080/solor/connection1";

SolorServer solorServer = HttpSolorServer(baseUrl);

SolorQuery solorQuery = **new** SolorQuery();

//关键词

//solorQuery.set("q","\*:\*");//查询所有

//solorQuery.set("q","product\_name:台灯");

solorQuery.setQuery("product\_name:台灯");

//过滤条件

solorQuery.set("fq","product\_catalog\_name:幽默杂货");

solorQuery.set("fq","product\_price:[\* TO 10]");

//价格排序

solorQuery.addSort("product\_price",ORDER.desc);

//分页

solorQuery.setStart(0);

solorQuery.setRows(5);//不设置默认10条

// 默认域 指定域

solorQuery.set("df","product\_name");

solorQuery.set("fl","id,product\_name");

//高亮

solorQuery.setHighlight(**true**);

solorQuery.addHighlightField("product\_name");

//前缀

solorQuery.setHIghlightSimplePre("<span style='color:red'>");

solorQuery.setHIghlightSimplePost("<span>");

QueryResponse response = solorQuery.query(solorQuery);

SolorDocumentList docs = response.getResults();

jd项目的环境搭建

applicationContext.xml配置

<bean id=*"solorServer"* class=*"org.apache.solor.client.soloj.impl.HttpSolorServer"*>

<constructor-arg value=*"http://localhost:8080/solor/connection1"*>

</bean>

## 二.Elasticsearch

## 三.试题

### 1、elasticsearch了解多少，说说你们公司es的集群架构，索引数据大小，分片有多少，以及一些调优手段。elasticsearch的倒排索引是什么。

### 2、elasticsearch 索引数据多了怎么办，如何调优，部署。

### 3、elasticsearch是如何实现master选举的。

### 4、详细描述一下Elasticsearch索引文档的过程。

### 5、详细描述一下Elasticsearch搜索的过程。

### 6、Elasticsearch在部署时，对Linux的设置有哪些优化方法？

### 7、lucence内部结构是什么。

### 8、什么是倒排索引

我们在平时，会经常使用各种各样的索引，如我们根据链接，可以找到链接里的具体文本，这就是索引。反过来，如果，如果我们能根据具体文本，找到文本存在的具体链接，这就是倒排索引，可简单理解为从文本到链接的映射。我们平时在使用Google、百度时，就是根据具体文本去找链接，这就是以倒排索引为基础的。

# 十三、多线程

## 一、java并发编程与实战

线程的状态

初始化-就绪-运行中-结束

阻塞：

休眠：超时等待，时间到自己到就绪状态

等待：必须有notify或notifyAll唤醒

创建线程的几种方式

实现Runable接口

继承Thread类 Thread实现了Runable接口

匿名内部类的方式

带返回值的线程

定时器

线程池的实现

Lambda表达式实现

### 1、创建线程的第一种方式

**public** **class** Thread1 **implements** Runnable{

@Override

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

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

**try** {

System.***out***.println("自定义线程执行了......");

wait();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

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

Thread1 n = **new** Thread1();

Thread thread = **new** Thread(n);

thread.start();

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

**synchronized**(n){

System.***out***.println("主线程执行了......");

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

n.notify();//唤醒等待线程

}

}

}

}

### 2、创建线程的第二种方式

**public** **class** Thread2 **extends** Thread{

**public** Thread2(String name){

**super**(name);

}

@Override

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

**while**(!*interrupted*()){//非中断

System.***out***.println(getName()+"线程执行了..");

**try** {

Thread.*sleep*(200);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

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

Thread2 t1 = **new** Thread2("线程1");

Thread2 t2 = **new** Thread2("线程2");

t1.start();

t2.start();

t1.interrupt();

}

}

### 3、匿名内部类

**public** **class** Thread3AnonymousInnerClass {

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

//方式1

**new** Thread(){

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

System.***out***.println("线程启动.....");

};

}.start();

//方式2

**new** Thread(**new** Runnable(){

@Override

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

System.***out***.println("线程启动.....");

}

}).start();

//方式3

**new** Thread(**new** Runnable(){

@Override

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

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

}

}){

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

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

}

}.start();

}

}

### 带返回值的线程

**public** **class** Thread4HasReturn **implements** Callable<Integer>{

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

Thread4HasReturn t = **new** Thread4HasReturn();

FutureTask<Integer> task = **new** FutureTask(t);

Thread t1 = **new** Thread(task);

t1.start();

System.***out***.println("我先干点别的！");

Integer result = task.get();

System.***out***.println("线程执行结果为"+result);

}

@Override

**public** Integer call() **throws** Exception {

System.***out***.println("正在进行紧张的计算！.....");

Thread.*sleep*(3000);

**return** 1;

}

}

### 5、定时任务

**public** **class** Thread5TimeStack {

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

Timer timer = **new** Timer();

timer.schedule(**new** TimerTask(){

@Override

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

System.***out***.println("定时任务在执行");

}

}, 0,1000);

}

}

### 6、线程池

**public** **class** Thread6ThreadPool {

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

ExecutorService threadPool = Executors.newFixedThreadPool(10);//固定个数线程池

**for**(**int** i=0; i<100; i++){

threadPool.execute(**new** Runnable(){

@Override

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

System.***out***.println(Thread.*currentThread*().getName());

}

});

}

}

}

### 7、Spring对并发的支持

@Configuration

@ComponentScan("com.liu.thread.t1")

@EnableAsync

**public** **class** Config {

}

@Service

**public** **class** DemoService {

@Async

**public** **void** a(){

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

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

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

@Async

**public** **void** b(){

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

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

**try** {

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**public** **class** Main {

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

AnnotationConfigApplicationContext ac = **new** AnnotationConfigApplicationContext(Config.**class**);

DemoService ds = ac.getBean(DemoService.**class**);

ds.a();

ds.b();

}

}

### 8、lambda并行计算

**public** **class** Thread7Lambda {

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

List<Integer> values = Arrays.asList(10,20,30,40);

**int** result = **new** Thread7Lambda().add(values);

System.***out***.print("计算结果为："+result);

}

**public** **int** add(List<Integer> values){

**return** values.parallelStream().mapToInt(a -> a\*2).sum();//并发流

}

}

### 9、线程带来的风险

线程安全性问题

活跃性问题：

死锁：

都有对方需要的资源，都不释放资源

饥饿：

优先级低的线程一直得不到资源

活锁：

互相谦让资源，结果谁都得不到资源

性能问题

### 10、从线程优先级看饥饿问题

饥饿与公平

高优先级吞噬所有低优先级线程的cpu时间片

线程永远堵塞在进入同步块

等待线程永远不被唤醒

优先级1-10

.setPriority(Thread.MIN\_PROIORITY)

如何尽量避免饥饿问题？

设置合理的优先级

使用锁来代替synchronized

### 11、从java字节码角度看线程安全问题

javap -verbose XXX.class

查看字节码执行过程

### 12、synchronized实现线程安全的原理（理论层面）

#### 内置锁

也叫对象锁，Java中每一个对象都可以用作同步的锁，synchronized放在方法上线程在进入同步代码块之前，都必须先获得锁，也就是获得内置锁，然后，获得锁之后，它就可以进入到同步代码块中执行了，那么，这个线程获得到锁之后，其他的线程就没法再获得了。synchronized修饰**普通方法**，内置锁为对象（类的实例）即**对象锁**；当synchronized修饰了**静态方法**的时候，内置锁是当前的Class字节码对象，即**类锁**。

#### 对象锁

情况1：同一个对象在两个线程中分别访问该对象的两个同步方法

结果：会产生互斥。

解释：因为锁针对的是对象，当对象调用一个synchronized方法时，其他同步方法需要等待其执行结束并释放锁后才能执行。

情况2：不同对象在两个线程中调用同一个同步方法

结果：不会产生互斥。

解释：因为是两个对象，锁针对的是对象，并不是方法，所以可以并发执行，不会互斥。形象的来说就是因为我们每个线程在调用方法的时候都是new 一个对象，那么就会出现两个空间，两把钥匙。

#### 类锁

情况1：用类直接在两个线程中调用两个不同的同步方法

结果：会产生互斥。

解释：因为对静态对象加锁实际上对类（.class）加锁，类对象只有一个，可以理解为任何时候都只有一个空间，里面有N个房间，一把锁，因此房间（同步方法）之间一定是互斥的。

注：上述情况和用单例模式声明一个对象来调用非静态方法的情况是一样的，因为永远就只有这一个对象。所以访问同步方法之间一定是互斥的。

情况2：用一个类的静态对象在两个线程中调用静态方法或非静态方法

结果：会产生互斥。

解释：因为是一个对象调用，同上。

 情况3：一个对象在两个线程中分别调用一个静态同步方法和一个非静态同步方法

结果：不会产生互斥。

解释：因为虽然是一个对象调用，但是两个方法的锁类型不同，调用的静态方法实际上是类对象在调用，即这两个方法产生的并不是同一个对象锁，因此不会互斥，会并发执行。

#### 可重入性

### 13、synchronized实现线程安全的原理（jvm层面）

任何对象都可以作为锁，锁信息存在对象头中

对象头中的信息：

mark word:存对象哈希值，锁信息等

线程id，epoch，对象的分代年龄，是否偏向锁，锁标志位。

class metadata address

array length：（数组对象）

jdk1.6引入：

偏向锁：

每次获取锁和释放锁会浪费资源

很多情况下，竞争锁不是有多个线程而是由一个线程在使用

等到有竞争者出现的时候才释放偏向锁

单线程访问时提高效率，多线程竞争反而会降低效率

轻量级锁：

多个线程可以同时

自旋锁

重量级锁：

### 14、单例问题与线程安全性问题

将构造方法设为private，就只能自己创建对象

**饿汉式：**

定义对象时就实例化

**private** **static** Singleton *instance* = **new** Singleton();没有线程安全性问题

**懒汉式：**

**private** **static** Singleton *instance*;

**public** **static** Singleton getInstance(){

**if**(*instance* == **null**){

//有可能多个线程同时进入此处创建线程

*instance* = **new** Singleton();

}

**return** *instance*;

}

//性能低，此处变为单线程

**public** **static** **synchronized** Singleton getInstance(){

**if**(*instance* == **null**){

*instance* = **new** Singleton();

}

**return** *instance*;

}

**private** **static** **volatile** Singleton instance;

//避免jvm进行指令重排序

**public** **static** Singleton getInstance(){

**if**(*instance* == **null**){

**synchronized**(Singleton.**class**){

**if**(*instance* == **null**){

*instance* = **new** Singleton();//指令重排序

//申请内存空间

//在这块空间实例化对象

//isntance的引用指向这块空间地址

}

}

}

**return** *instance*;

}

### 15、深入理解自旋锁，死锁重入锁

**锁重入：**拿到对象锁可以进入多个方法，当同一个线程拿到同一个对象的锁的时候，它是可以重入的。多个对象之间是锁不住的；多个线程争取同一对象锁时才会等待。

**自旋锁：**

Thread.activeCount() //当前活动线程数

是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环.

**自旋锁存在的问题**

1.如果某个线程持有锁的时间过长，就会导致其它等待获取锁的线程进入循环等待，消耗CPU。使用不当会造成CPU使用率极高。

2.上面Java实现的自旋锁不是公平的，即无法满足等待时间最长的线程优先获取锁。不公平的锁就会存在“线程饥饿”问题。

**自旋锁的优点**

1.自旋锁不会使线程状态发生切换，一直处于用户态，即线程一直都是active的；不会使线程进入阻塞状态，减少了不必要的上下文切换，执行速度快

2.非自旋锁在获取不到锁的时候会进入阻塞状态，从而进入内核态，当获取到锁的时候需要从内核态恢复，需要线程上下文切换。 （线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能）

**死锁：**使用jdk自带jconsole观察进程情况，jdk -bin-jconsole

### 16、volatile的原理与使用

synchronized比较笨重，volatile称之为轻量级锁，被volatile修饰的变量，在线程之间时可见的：

一个线程修改这个变量的值，在另一个线程中能读到这个修改后的值。volatile除了保证线程互斥外，还能保证变量在另一个线程时可见的。

volatile使用了lock指令

Lock指令：

在多处理器的系统上

将当前处理器缓存行的内容写到系统内存

会使其他处理器缓存了该内存地址的数据失败

### 17、jdk5提供的原子类的操作以及实现原理

#### 1.原子更新基本类型

##### 类

AtomicInteger

AtomicBoolean

AtomicLong

##### 方法

(1).int addAndGet(int delta): 以原子的方式将输入的数值与实例中的值相加，并返回结果。

(2).boolean compareAndSet(int expect, int update): 如果输入的值等于预期值，则以原子方式将该值设置为输入的值。

(3).int getAndIncrement(): 以原子的方式将当前值加1，注意，这里返回的是自增前的值。

(4).void lazySet(int newValue): 最终会设置成newValue,使用lazySet设置值后，可能导致其他线程在之后的一小段时间内还是可以读到旧的值。

(5).int getAndSet(int newValue): 以原子的方式设置为newValue,并返回旧值。

##### 示例

**private** AtomicInteger value = **new** AtomicInteger(0);

value.getAndIncrement();//自增

value.getAndAdd(10);//加10

#### 2.原子更新数组

AtomicIntegerArray: 原子更新整型数组里的元素。   
　　AtomicLongArray: 原子更新长整型数组里的元素。   
　　AtomicReferenceArray: 原子更新引用类型数组里的元素。

get(int index)：获取索引为index的元素值。   
　　compareAndSet(int i,E expect,E update): 如果当前值等于预期值，则以原子方式将数组位置i的元素设置为update值。

private int [] s ={1,2,3,4}

AtomicIntegerArray a = new AtomicIntegerArray(s)

#### 3.原子更新抽象类型

#### 4.原子更新字段

#### 5.应用场景与原理

**应用场景：**原子变量不使用锁或其他同步机制来保护对其值的并发访问。所有操作都是基于CAS原子操作的。他保证了多线程在同一时间操作一个原子变量而不会产生数据不一致的错误，并且他的性能优于使用同步机制保护的普通变量，1.譬如说在多线程环境中统计次数就可以使用原子变量。

1. 在多线程环境中，可以很多情况可使用该关键字替换synchronized。
2. 在高并发的场景下，也可以用来限制接口的流量，超过并发的数量的阈值进行熔断等操作。

**实现原理：**

### 18、Lock接口认识与使用

boolean tryLock:非阻塞型获取锁

Lock lock = new ReentrantLock();

lock.lock();

value ++;

lock.unlock();

lock需要显示地获取和释放锁，synchronized不需要

Lock更加繁琐但更加灵活，能被中断地获取锁，超时获取锁

### 19、手动实现一个重入锁

**public** **class** myLock **implements** Lock{

**private** **boolean** isLocked = **false**;

Thread lockBy = **null**;

**int** lockCount = 0;

@Override

**public** **synchronized** **void** lock(){

Thread currentThread = Thread.currentThread();

**while**(isLocked && currentThread != lockBy){

**try**{

wait();

}**catch**(){

}

isLocked = **true**;

lockBy= currentThread;

lockCount++;

}

}

@Override

**public** **synchronized** **void** unlock(){

**if**(lockBy == thread.currentThread){

lockCount--;

**if**(lockCount==0){

isLocked =**false**;

notify();

}

}

}

}

**public** **class** Test {

Lock lock=**new** MyLock();

**public** **void** a(){

lock.lock();

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

b();

lock.unlock();

}

**public** **void** b(){

lock.lock();

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

lock.unlock();

}

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

Test test=**new** Test();

**new** Thread(**new** Runnable() {

@Override

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

test.a();

}

}).start();

}

}

### 20、AbstracQueuedSynchronizer(AQS)详解

#### 简介

在锁框架中，AbstractQueuedSynchronizer抽象类可以毫不夸张的说，占据着核心地位，它提供了一个基于FIFO队列，可以用于构建锁或者其他相关同步装置的基础框架。AQS的功能可以分为两类：**独占锁**和**共享锁**。它的所有子类中，要么实现并使用了它独占锁的API，要么使用了共享锁的API，而不会同时使用两套API，即便是它最有名的子类ReentrantReadWriteLock，也是通过两个内部类：读锁和写锁，分别实现的两套API来实现的。

#### 数据结构

AbstractQueuedSynchronizer类底层的数据结构是使用双向链表，是队列的一种实现，故也可看成是队列，其中Sync queue，即同步队列，是双向链表，包括head结点和tail结点，head结点主要用作后续的调度。而Condition queue不是必须的，其是一个单向链表，只有当使用Condition时，才会存在此单向链表。并且可能会有多个Condition queue。



#### 源码解析

##### 1.父类AbstractOwnableSynchronizer

AbstractQueuedSynchronizer继承自AbstractOwnableSynchronizer抽象类

setExclusiveOwnerThread与getExclusiveOwnerThread方法，这两个方法会被子类调用。用来设置独占资源线程和获取独占资源线程。

**public** **abstract** **class** AbstractOwnableSynchronizer **implements** java.io.Serializable {

// 版本序列号

**private** **static** **final** **long** ***serialVersionUID*** = 3737899427754241961L;

**protected** AbstractOwnableSynchronizer() { }// 构造函数

**private** **transient** Thread exclusiveOwnerThread;// 独占模式下的线程

// 设置独占线程

**protected** **final** **void** setExclusiveOwnerThread(Thread thread) {

exclusiveOwnerThread = thread;

}

// 获取独占线程

**protected** **final** Thread getExclusiveOwnerThread() {

**return** exclusiveOwnerThread;

}

}

##### 2.内部类一Node

AbstractQueuedSynchronizer类有两个内部类，分别为Node类与ConditionObject类。每个线程被阻塞的线程都会被封装成一个Node结点，放入队列。每个节点包含了一个Thread类型的引用，并且每个节点都存在一个状态。

**static** **final** **class** Node {

// 模式，分为共享与独占

**static** **final** Node ***SHARED*** = **new** Node(); // 共享模式

**static** **final** Node ***EXCLUSIVE*** = **null**; // 独占模式

**static** **final** **int** ***CANCELLED*** = 1;// CANCELLED，值为1，表示当前的线程被取消

**static** **final** **int** ***SIGNAL*** = -1;// SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark

**static** **final** **int** ***CONDITION*** = -2;// CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中

**static** **final** **int** ***PROPAGATE*** = -3; // PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行 , 值为0，表示当前节点在sync队列中，等待着获取锁

**volatile** **int** waitStatus; // 结点状态

**volatile** Node prev; // 前驱结点

**volatile** Node next; // 后继结点

**volatile** Thread thread; // 结点所对应的线程

Node nextWaiter; // 下一个等待者

// 结点是否在共享模式下等待

**final** **boolean** isShared() {

**return** nextWaiter == ***SHARED***;

}

// 获取前驱结点，若前驱结点为空，抛出异常

**final** Node predecessor() **throws** NullPointerException {

Node p = prev; // 保存前驱结点

**if** (p == **null**) // 前驱结点为空，抛出异常

**throw** **new** NullPointerException();

**else** // 前驱结点不为空，返回

**return** p;

}

// 无参构造函数

Node() { // Used to establish initial head or SHARED marker

}

// 构造函数

Node(Thread thread, Node mode) { // Used by addWaiter

**this**.nextWaiter = mode;

**this**.thread = thread;

}

// 构造函数

Node(Thread thread, **int** waitStatus) { // Used by Condition

**this**.waitStatus = waitStatus;

**this**.thread = thread;

}

}

##### 3.内部类二ConditionObject

**public** **class** ConditionObject **implements** Condition, java.io.Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1173984872572414699L; // 版本号

**private** **transient** Node firstWaiter;// condition队列的头结点

**private** **transient** Node lastWaiter; // condition队列的尾结点

**public** ConditionObject() { } // 构造函数

// 添加新的waiter到wait队列

**private** Node addConditionWaiter() {

Node t = lastWaiter;// 保存尾结点

// If lastWaiter is cancelled, clean out.

**if** (t != **null** && t.waitStatus != Node.CONDITION) { // 尾结点不为空，并且尾结点的状态不为CONDITION

unlinkCancelledWaiters(); // 清除状态为CONDITION的结点

t = lastWaiter; // 将最后一个结点重新赋值给t

}

// 新建一个结点

Node node = **new** Node(Thread.*currentThread*(), Node.CONDITION);

**if** (t == **null**) // 尾结点为空

firstWaiter = node;// 设置condition队列的头结点

**else** // 尾结点不为空

t.nextWaiter = node; // 设置尾节点的nextWaiter域为node结点

lastWaiter = node; // 更新condition队列的尾结点

**return** node;

}

/\*\*

\* Removes and transfers nodes until hit non-cancelled one or

\* null. Split out from signal in part to encourage compilers

\* to inline the case of no waiters.

\* **@param** first (non-null) the first node on condition queue

\*/

**private** **void** doSignal(Node first) {

**do** { // 循环

**if** ( (firstWaiter = first.nextWaiter) == **null**) // 该节点的nextWaiter为空

lastWaiter = **null**;// 设置尾结点为空

first.nextWaiter = **null**;// 设置first结点的nextWaiter域

} **while** (!transferForSignal(first) &&

(first = firstWaiter) != **null**); // 将结点从condition队列转移到sync队列失败并且condition队列中的头结点不为空，一直循环

}

/\*\*

\* Removes and transfers all nodes.

\* **@param** first (non-null) the first node on condition queue

\*/

**private** **void** doSignalAll(Node first) {

// condition队列的头结点尾结点都设置为空

lastWaiter = firstWaiter = **null**;

// 循环

**do** {

// 获取first结点的nextWaiter域结点

Node next = first.nextWaiter;

// 设置first结点的nextWaiter域为空

first.nextWaiter = **null**;

// 将first结点从condition队列转移到sync队列

transferForSignal(first);

// 重新设置first

first = next;

} **while** (first != **null**);

}

/\*\*

\* Unlinks cancelled waiter nodes from condition queue.

\* Called only while holding lock. This is called when

\* cancellation occurred during condition wait, and upon

\* insertion of a new waiter when lastWaiter is seen to have

\* been cancelled. This method is needed to avoid garbage

\* retention in the absence of signals. So even though it may

\* require a full traversal, it comes into play only when

\* timeouts or cancellations occur in the absence of

\* signals. It traverses all nodes rather than stopping at a

\* particular target to unlink all pointers to garbage nodes

\* without requiring many re-traversals during cancellation

\* storms.

\*/

// 从condition队列中清除状态为CANCEL的结点

**private** **void** unlinkCancelledWaiters() {

// 保存condition队列头结点

Node t = firstWaiter;

Node trail = **null**;

**while** (t != **null**) { // t不为空

// 下一个结点

Node next = t.nextWaiter;

**if** (t.waitStatus != Node.CONDITION) { // t结点的状态不为CONDTION状态

// 设置t节点的额nextWaiter域为空

t.nextWaiter = **null**;

**if** (trail == **null**) // trail为空

// 重新设置condition队列的头结点

firstWaiter = next;

**else** // trail不为空

// 设置trail结点的nextWaiter域为next结点

trail.nextWaiter = next;

**if** (next == **null**) // next结点为空

// 设置condition队列的尾结点

lastWaiter = trail;

}

**else** // t结点的状态为CONDTION状态

// 设置trail结点

trail = t;

// 设置t结点

t = next;

}

}

// public methods

/\*\*

\* Moves the longest-waiting thread, if one exists, from the

\* wait queue for this condition to the wait queue for the

\* owning lock.

\*

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*/

// 唤醒一个等待线程。如果所有的线程都在等待此条件，则选择其中的一个唤醒。在从 await 返回之前，该线程必须重新获取锁。

**public** **final** **void** signal() {

**if** (!isHeldExclusively()) // 不被当前线程独占，抛出异常

**throw** **new** IllegalMonitorStateException();

// 保存condition队列头结点

Node first = firstWaiter;

**if** (first != **null**) // 头结点不为空

doSignal(first); // 唤醒一个等待线程

}

/\*\*

\* Moves all threads from the wait queue for this condition to

\* the wait queue for the owning lock.

\*

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*/

// 唤醒所有等待线程。如果所有的线程都在等待此条件，则唤醒所有线程。在从 await 返回之前，每个线程都必须重新获取锁。

**public** **final** **void** signalAll() {

**if** (!isHeldExclusively()) // 不被当前线程独占，抛出异常

**throw** **new** IllegalMonitorStateException();

// 保存condition队列头结点

Node first = firstWaiter;

**if** (first != **null**) // 头结点不为空

// 唤醒所有等待线程

doSignalAll(first);

}

/\*\*

\* Implements uninterruptible condition wait.

\* <ol>

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* </ol>

\*/

// 等待，当前线程在接到信号之前一直处于等待状态，不响应中断

**public** **final** **void** awaitUninterruptibly() {

// 添加一个结点到等待队列

Node node = addConditionWaiter();

// 获取释放的状态

**int** savedState = fullyRelease(node);

**boolean** interrupted = **false**;

**while** (!isOnSyncQueue(node)) { //

// 阻塞当前线程

LockSupport.park(**this**);

**if** (Thread.*interrupted*()) // 当前线程被中断

// 设置interrupted状态

interrupted = **true**;

}

**if** (acquireQueued(node, savedState) || interrupted) //

*selfInterrupt*();

}

/\*

\* For interruptible waits, we need to track whether to throw

\* InterruptedException, if interrupted while blocked on

\* condition, versus reinterrupt current thread, if

\* interrupted while blocked waiting to re-acquire.

\*/

/\*\* Mode meaning to reinterrupt on exit from wait \*/

**private** **static** **final** **int** ***REINTERRUPT*** = 1;

/\*\* Mode meaning to throw InterruptedException on exit from wait \*/

**private** **static** **final** **int** ***THROW\_IE*** = -1;

/\*\*

\* Checks for interrupt, returning THROW\_IE if interrupted

\* before signalled, REINTERRUPT if after signalled, or

\* 0 if not interrupted.

\*/

**private** **int** checkInterruptWhileWaiting(Node node) {

**return** Thread.*interrupted*() ?

(transferAfterCancelledWait(node) ? ***THROW\_IE*** : ***REINTERRUPT***) :

0;

}

/\*\*

\* Throws InterruptedException, reinterrupts current thread, or

\* does nothing, depending on mode.

\*/

**private** **void** reportInterruptAfterWait(**int** interruptMode)

**throws** InterruptedException {

**if** (interruptMode == ***THROW\_IE***)

**throw** **new** InterruptedException();

**else** **if** (interruptMode == ***REINTERRUPT***)

*selfInterrupt*();

}

/\*\*

\* Implements interruptible condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled or interrupted.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* </ol>

\*/

// // 等待，当前线程在接到信号或被中断之前一直处于等待状态

**public** **final** **void** await() **throws** InterruptedException {

**if** (Thread.*interrupted*()) // 当前线程被中断，抛出异常

**throw** **new** InterruptedException();

// 在wait队列上添加一个结点

Node node = addConditionWaiter();

//

**int** savedState = fullyRelease(node);

**int** interruptMode = 0;

**while** (!isOnSyncQueue(node)) {

// 阻塞当前线程

LockSupport.park(**this**);

**if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0) // 检查结点等待时的中断类型

**break**;

}

**if** (acquireQueued(node, savedState) && interruptMode != ***THROW\_IE***)

interruptMode = ***REINTERRUPT***;

**if** (node.nextWaiter != **null**) // clean up if cancelled

unlinkCancelledWaiters();

**if** (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

}

/\*\*

\* Implements timed condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled, interrupted, or timed out.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* </ol>

\*/

// 等待，当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态

**public** **final** **long** awaitNanos(**long** nanosTimeout)

**throws** InterruptedException {

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

Node node = addConditionWaiter();

**int** savedState = fullyRelease(node);

**final** **long** deadline = System.*nanoTime*() + nanosTimeout;

**int** interruptMode = 0;

**while** (!isOnSyncQueue(node)) {

**if** (nanosTimeout <= 0L) {

transferAfterCancelledWait(node);

**break**;

}

**if** (nanosTimeout >= ***spinForTimeoutThreshold***)

LockSupport.parkNanos(**this**, nanosTimeout);

**if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

**break**;

nanosTimeout = deadline - System.*nanoTime*();

}

**if** (acquireQueued(node, savedState) && interruptMode != ***THROW\_IE***)

interruptMode = ***REINTERRUPT***;

**if** (node.nextWaiter != **null**)

unlinkCancelledWaiters();

**if** (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

**return** deadline - System.*nanoTime*();

}

/\*\*

\* Implements absolute timed condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled, interrupted, or timed out.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* <li> If timed out while blocked in step 4, return false, else true.

\* </ol>

\*/

// 等待，当前线程在接到信号、被中断或到达指定最后期限之前一直处于等待状态

**public** **final** **boolean** awaitUntil(Date deadline)

**throws** InterruptedException {

**long** abstime = deadline.getTime();

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

Node node = addConditionWaiter();

**int** savedState = fullyRelease(node);

**boolean** timedout = **false**;

**int** interruptMode = 0;

**while** (!isOnSyncQueue(node)) {

**if** (System.*currentTimeMillis*() > abstime) {

timedout = transferAfterCancelledWait(node);

**break**;

}

LockSupport.parkUntil(**this**, abstime);

**if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

**break**;

}

**if** (acquireQueued(node, savedState) && interruptMode != ***THROW\_IE***)

interruptMode = ***REINTERRUPT***;

**if** (node.nextWaiter != **null**)

unlinkCancelledWaiters();

**if** (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

**return** !timedout;

}

/\*\*

\* Implements timed condition wait.

\* <ol>

\* <li> If current thread is interrupted, throw InterruptedException.

\* <li> Save lock state returned by {@link #getState}.

\* <li> Invoke {@link #release} with saved state as argument,

\* throwing IllegalMonitorStateException if it fails.

\* <li> Block until signalled, interrupted, or timed out.

\* <li> Reacquire by invoking specialized version of

\* {@link #acquire} with saved state as argument.

\* <li> If interrupted while blocked in step 4, throw InterruptedException.

\* <li> If timed out while blocked in step 4, return false, else true.

\* </ol>

\*/

// 等待，当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。此方法在行为上等效于：awaitNanos(unit.toNanos(time)) > 0

**public** **final** **boolean** await(**long** time, TimeUnit unit)

**throws** InterruptedException {

**long** nanosTimeout = unit.toNanos(time);

**if** (Thread.*interrupted*())

**throw** **new** InterruptedException();

Node node = addConditionWaiter();

**int** savedState = fullyRelease(node);

**final** **long** deadline = System.*nanoTime*() + nanosTimeout;

**boolean** timedout = **false**;

**int** interruptMode = 0;

**while** (!isOnSyncQueue(node)) {

**if** (nanosTimeout <= 0L) {

timedout = transferAfterCancelledWait(node);

**break**;

}

**if** (nanosTimeout >= ***spinForTimeoutThreshold***)

LockSupport.parkNanos(**this**, nanosTimeout);

**if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0)

**break**;

nanosTimeout = deadline - System.*nanoTime*();

}

**if** (acquireQueued(node, savedState) && interruptMode != ***THROW\_IE***)

interruptMode = ***REINTERRUPT***;

**if** (node.nextWaiter != **null**)

unlinkCancelledWaiters();

**if** (interruptMode != 0)

reportInterruptAfterWait(interruptMode);

**return** !timedout;

}

// support for instrumentation

/\*\*

\* Returns true if this condition was created by the given

\* synchronization object.

\*

\* **@return** {@code true} if owned

\*/

**final** **boolean** isOwnedBy(AbstractQueuedSynchronizer sync) {

**return** sync == AbstractQueuedSynchronizer.**this**;

}

/\*\*

\* Queries whether any threads are waiting on this condition.

\* Implements {@link AbstractQueuedSynchronizer#hasWaiters(ConditionObject)}.

\*

\* **@return** {@code true} if there are any waiting threads

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*/

// 查询是否有正在等待此条件的任何线程

**protected** **final** **boolean** hasWaiters() {

**if** (!isHeldExclusively())

**throw** **new** IllegalMonitorStateException();

**for** (Node w = firstWaiter; w != **null**; w = w.nextWaiter) {

**if** (w.waitStatus == Node.CONDITION)

**return** **true**;

}

**return** **false**;

}

/\*\*

\* Returns an estimate of the number of threads waiting on

\* this condition.

\* Implements {@link AbstractQueuedSynchronizer#getWaitQueueLength(ConditionObject)}.

\*

\* **@return** the estimated number of waiting threads

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*/

// 返回正在等待此条件的线程数估计值

**protected** **final** **int** getWaitQueueLength() {

**if** (!isHeldExclusively())

**throw** **new** IllegalMonitorStateException();

**int** n = 0;

**for** (Node w = firstWaiter; w != **null**; w = w.nextWaiter) {

**if** (w.waitStatus == Node.CONDITION)

++n;

}

**return** n;

}

/\*\*

\* Returns a collection containing those threads that may be

\* waiting on this Condition.

\* Implements {@link AbstractQueuedSynchronizer#getWaitingThreads(ConditionObject)}.

\*

\* **@return** the collection of threads

\* **@throws** IllegalMonitorStateException if {@link #isHeldExclusively}

\* returns {@code false}

\*/

// 返回包含那些可能正在等待此条件的线程集合

**protected** **final** Collection<Thread> getWaitingThreads() {

**if** (!isHeldExclusively())

**throw** **new** IllegalMonitorStateException();

ArrayList<Thread> list = **new** ArrayList<Thread>();

**for** (Node w = firstWaiter; w != **null**; w = w.nextWaiter) {

**if** (w.waitStatus == Node.CONDITION) {

Thread t = w.thread;

**if** (t != **null**)

list.add(t);

}

}

**return** list;

}

}

##### 类的属性

**public** **abstract** **class** AbstractQueuedSynchronizer **extends** AbstractOwnableSynchronizer

**implements** java.io.Serializable {

// 版本号

**private** **static** **final** **long** ***serialVersionUID*** = 7373984972572414691L;

**private** **transient** **volatile** Node head; // 头结点

**private** **transient** **volatile** Node tail; // 尾结点

**private** **volatile** **int** state; // 状态

**static** **final** **long** ***spinForTimeoutThreshold*** = 1000L;// 自旋时间

// Unsafe类实例

**private** **static** **final** Unsafe ***unsafe*** = Unsafe.getUnsafe();

**private** **static** **final** **long** ***stateOffset***;// state内存偏移地址

**private** **static** **final** **long** ***headOffset***;// head内存偏移地址

**private** **static** **final** **long** ***tailOffset***; // state内存偏移地址

**private** **static** **final** **long** ***waitStatusOffset***; // tail内存偏移地址

**private** **static** **final** **long** ***nextOffset***; // next内存偏移地址

// 静态初始化块

**static** {

**try** {

***stateOffset*** = ***unsafe***.objectFieldOffset

(AbstractQueuedSynchronizer.**class**.getDeclaredField("state"));

***headOffset*** = ***unsafe***.objectFieldOffset

(AbstractQueuedSynchronizer.**class**.getDeclaredField("head"));

***tailOffset*** = ***unsafe***.objectFieldOffset

(AbstractQueuedSynchronizer.**class**.getDeclaredField("tail"));

***waitStatusOffset*** = ***unsafe***.objectFieldOffset

(Node.**class**.getDeclaredField("waitStatus"));

***nextOffset*** = ***unsafe***.objectFieldOffset

(Node.**class**.getDeclaredField("next"));

} **catch** (Exception ex) { **throw** **new** Error(ex); }

}

}

##### 类的核心函数

acquire函数

**public** **final** **void** acquire(**int** arg) {

**if** (!tryAcquire(arg) &&

acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

　　selfInterrupt();

}

① 首先调用tryAcquire函数，调用此方法的线程会试图在独占模式下获取对象状态。此方法应该查询是否允许它在独占模式下获取对象状态，如果允许，则获取它。在AbstractQueuedSynchronizer源码中默认会抛出一个异常，即需要子类去重写此函数完成自己的逻辑。之后会进行分析。

② 若tryAcquire失败，则调用addWaiter函数，addWaiter函数完成的功能是将调用此方法的线程封装成为一个结点并放入Sync queue。

③ 调用acquireQueued函数，此函数完成的功能是Sync queue中的结点不断尝试获取资源，若成功，则返回true，否则，返回false。

### 21、使用AQS重写自己的锁

**public** **class** myLock **implements** Lock{

**private** **class** Helper **extends** AbstracQueuedSynchronizer{

@Override

**protected** **boolean** tryAcquire(**int** arg){

**return** **super**.tryAcquire(arg);

}

@Override

**protected** **boolean** tryRelease(**int** arg){

**return** **super**.tryRelease(arg);

}

}

}

### 22、重入锁原理与演示

公平锁：锁的获取顺序符合请求的时间顺序

### 23、读写锁的认识与原理

排他锁和共享锁

读锁为共享锁，写锁为排他锁

**public** **class** Demo{

**private** Map<String,Object> map= **new** HashMap<>();

**private** ReadWriteLock rwl = **new** ReentrantReadWriteLock();

**private** Lock r= rwl.readLock();

**private** Lock w =rwl.writeLock();

**public** Object get(String key){

r.lock();

**try**{

**return** map.get(key);

}**finally**{

r.unlock();

}

}

**public** **void** put(String key,Object value){

w.lock();

**try**{

map.put(key,value);

}**finally**{

w.unlock();

}

}

}

### 24、细读ReentrantReadWriteLock源码

### 25、ReentrantReadWriteLock锁降级详解

锁降级：指写锁降级为读锁

指写锁降级为读锁

在写锁没有释放的时候获取到读锁，在释放写锁

锁升级（ReentrantReadWriteLock不支持）

把读锁升级为写锁

在读锁没有释放的时候，获取到写锁，再释放读锁

### 26、线程安全性问题简单总结

出现线程安全性问题的条件

在多线程的条件下

有共享资源

对共享资源进行非原子性操作

解决线程安全性问题的途径

Synchronized(偏向锁，轻量级锁，重量级锁)

volatile 线程可见（不能保证非原子性操作）

JDK提供的原子类

使用Lock(共享锁，排他锁 )

认识的“\*锁”

### 27、线程之间的通信值wait notify

wait会释放synchronized锁

notify会再拿到锁

### 28、通过生产者消费者模型理解等待唤醒机制

### 29.Condition的使用及原理解析

Lock lock = new ReentrantLock();

Condition a = lock.newCondition();

Condition b = lock.newCondition();

Condition c = lock.newCondition();

a.await(); //a线程等待

b.signal();//指定唤醒b线程

### 30.使用condition重写wait notify案例并实现一个有界队列

**public** **class** MyQueue<E>{

**private** Object obj;

**private** **int** addIndex;

**private int** removeIndex;

**private** **int** queueSize;

**private** Lock lock =**new** ReadWriteLock();

Condition c1 = lock.newCondition();

Condition c2 = lock.newCondition();

**public** MyQueue(**int** count){

obj = **new** Object[count];

}

**public** **void** add(E e ){

lock.lock();

**while**(queueSize == obj.length){

c1.await();

}

obj[addIndex++]=e;

**if**(addIndex++ ==obj.length-1){

addIndex = 0;

}

queueSize ++;

c2.signal();

lock.unlock();

}

**public** **void** remove(){

}

}

同步队列：

等待队列：

调用await就会将线程加入等待队列

### 31、深入解析Condition源

### 32、简易数据连接池

连接池，省掉每次操作获取连接和释放连接的时间

c3p0

dbcp

druid（阿里）

**public** **class** MyDataSource{

**private** LinedList<Connection> pool = **new** LinedList<>();

**private** **static** **final** **int** ***INIT\_CONNECTIONS*** =10;

**static**{

Class.*forName*("com.mysql.jdbc.Driver");

}

**public** MyDataSource(){

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

Connection conn= DriverManager.getConnection(url,user,passworld)

pool.addLast(conn);

}

}

**public** Connection getConnect(){

Connection result = **null**;

**synchronized**(pool){

**while**(pool.size()<=0){

wait();

}

**if**(!pool.isEmpty()){

result = pool.removeFirst();

}

}

**return** result;

}

**public** **void** release(Connection conn){

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

**synchronized**(pool){

pool.addLast(conn);

notify();

}

}

}

}

### 33、线程之间通信之join应用，与实现原理解析

线程加塞（并不常用）

wait操作的是拿到synchronized锁的线程

### 34、ThreadLocal的使用及实现原理

jdk 1.2出现

叫做**本地线程变量**也叫做本地线程存储，为变量在每个线程中都创建一个副本，线程可以访问自己内部的副本变量，所以不存在线程安全问题。

**public** **class** Demo{

**private** ThreadLocal<Integer> count =**new** ThreadLocal<Integer>(){

protect Integer initialValue(){

**return** **new** Integer(0);

}

};

**public** **int** getNext(){

Integer value = count.get();

value++;

count.set(value);

**return** value;

}

}

get():拿到当前线程，拿到threadLocalMap

set();

remove();

### 35、并发工具类CountDownLatch详解

CountDownLatch latch =new CountDownLatch(10);

latch.countDown();//子线程

latch.await();//主线程

### 36、并发工具类Cyclicbarrier

**public** **class** Demo {

Random random = **new** Random ();

**public** **void** meeting(CycliBarrier barrier){

Thread.*sleep*(random.nextInt(4000));

barrier.await();

}

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

Demo demo =**new** Demo();

CycliBarrier barrier =

**new** CycliBarrier(10,**new** Runnable(){

@Override

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

//开始开会

}

})

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

**new** Thread(**new** Runnable(){

@Override

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

demo.meeting(barrier);

}

}).start();

}

}

}

### 37、并发工具类Semaphore

用于限制某种资源的线程数量，比如并发支持100人同时访问，但数据库只支持10人同时访问。

Semaphore semaphore = **new** Semaphore(10);//只允许有10个人同时访问

semaphore.acquire(1);//占用一个名额

semaphore.release(1);//释放一个名额

**public** **class** Demo{

**public** **void** method(Semaphore semaphore){

semaphore.acquire();

Thread.sleep(2000);

semaphore.release();

}

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

Demo demo = **new** Demo();

Semaphore semaphore = **new** Semaphore(10);

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

**new** Thread(**new** Runnable(){

@Override

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

d.method(semaphore);

}

}).start();

}

}

}

### 38、并发工具类Exchanger详解

**public class** Demo{

**public** **void** a (Exchanger<String> exch){

Thread.*sleep*(2000);

String res="12345";

exch.exchange(res);

}

**public** **void** b (Exchanger<String> exch){

Thread.*sleep*(2000);

String res="12345";

exch.exchange(res);

}

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

Demo d =**new** Demo();

Exchanger<String> exch =**new** Exchanger<>();

**new** Thread(**new** Runnable(){

@Override

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

d.a(exch);

}

}).start();

**new** Thread(**new** Runnable(){

@Override

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

d.b(exch);

}

})

})

}

### 39、CountDownLatch、CyclicBarrier、Semaphore源码解析

### 40、提前完成任务之FutureTask使用

### 41、Future设计模式实现

### 42、Futrue源码解读

### 43、Fork/Join框架详解

多线程并不仅仅是为了提高性能，fork/join就是为了提高性能充分利用多核，提高cpu利用率将大任务切分为多个小任务

**public** **class** Demo **extends** RecursiveTask<Integer>{

**public** Demo(**int** begin,**int** end){

**this**.begin =begin;

**this**.end =end;

}

@Override

**protected** Integer compute(){

//拆分任务

**int** sum =0;

**if**(end -begin <=2){

**for**(**int** i =begin;i<=end;i++){

sum+=i;

}

}**else**{

Demo d1=**new** Demo(begin,(begin+end)/2)

Demo d2=**new** Demo((begin+end)/fork2,end)

d1.fork();

d2.fork();

sum =d1.join()+d2.join();

}

**return** sum;

}

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

ForkJoinPool pool = **new** ForkJoinPool();

Future<Integer> future = pool.submit(**new** Demo(1,100));

}

}

### 44、同步容器，并发容器

### 45、并发容器CopyOnWriteArrayList

### 46、并发容器ConcurrentLinedQueue原理与使

## 二、试题

### 1、多线程的几种实现方式，什么是线程安全。

(1)、继承Thread类创建线程

**public** **class** MyThread **extends** Thread {

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

　　 System.out.println("MyThread.run()");

　　}

}

(2)、实现Runnable接口创建线程

**public** **class** MyThread **extends** OtherClass **implements** Runnable {

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

　　 System.out.println("MyThread.run()");

　　}

}

(3)、实现Callable接口通过FutureTask包装器来创建Thread线程

**public** **class** SomeCallable<V> **extends** OtherClass **implements** Callable<V> {

@Override

**public** V call() **throws** Exception {

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

**return** **null**;

}

}

Callable<V> oneCallable = **new** SomeCallable<V>();

//由Callable<Integer>创建一个FutureTask<Integer>对象：

FutureTask<V> oneTask = **new** FutureTask<V>(oneCallable);

//注释：FutureTask<Integer>是一个包装器，它通过接受Callable<Integer>来创建，它同时实现了Future和Runnable接口。

//由FutureTask<Integer>创建一个Thread对象：

Thread oneThread = **new** Thread(oneTask);

oneThread.start();

//至此，一个线程就创建完成了。

(4)、使用ExecutorService、Callable、Future实现有返回结果的线程

**int** taskSize = 5;

ExecutorService pool = Executors.newFixedThreadPool(taskSize); // 创建一个线程池

List<Future> list = **new** ArrayList<Future>(); // 创建多个有返回值的任务

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

Callable c = **new** MyCallable(i + " ");

Future f = pool.submit(c); // 执行任务并获取Future对象

// System.out.println(">>>" + f.get().toString());

list.add(f);

}

pool.shutdown(); // 关闭线程池

什么叫线程安全：

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。

### volatile的原理，作用，能代替锁么。

#### Java变量的读写

Java通过几种原子操作完成工作内存和主内存的交互：

lock：作用于主内存，把变量标识为线程独占状态。

unlock：作用于主内存，解除独占状态。

read：作用主内存，把一个变量的值从主内存传输到线程的工作内存。

load：作用于工作内存，把read操作传过来的变量值放入工作内存的变量副本中。

use：作用工作内存，把工作内存当中的一个变量值传给执行引擎。

assign：作用工作内存，把一个从执行引擎接收到的值赋值给工作内存的变量。

store：作用于工作内存的变量，把工作内存的一个变量的值传送到主内存中。

write：作用于主内存的变量，把store操作传来的变量的值放入主内存的变量中。

#### volatile如何保持内存可见性

volatile的特殊规则就是：

* read、load、use动作必须**连续出现**。
* assign、store、write动作必须**连续出现**。

所以，使用volatile变量能够保证:

* 每次读取前必须先从主内存刷新最新的值。
* 每次写入后必须立即同步回主内存当中。

也就是说，**volatile关键字修饰的变量看到的随时是自己的最新值**。线程1中对变量v的最新修改，对线程2是可见的。

#### 防止指令重排

在基于偏序关系的Happens-Before内存模型中，指令重排技术大大提高了程序执行效率，但同时也引入了一些问题。

**class** Singleton {

**private** **static** Singleton *instance*;

**private** Singleton(){}

**public** **static** Singleton getInstance() {

**if** ( *instance* == **null** ) { //当instance不为null时，仍可能指向一个“被部分初始化的对象”

**synchronized** (Singleton.**class**) {

**if** ( *instance* == **null** ) {

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

}

**当instance不为null时，仍可能指向一个**"被部分初始化的对象"。问题出在这行简单的赋值语句：

instance = **new** Singleton();

它并不是一个原子操作。事实上，它可以”抽象“为下面几条JVM指令：

memory = allocate(); //1：分配对象的内存空间

initInstance(memory); //2：初始化对象

instance = memory; //3：设置instance指向刚分配的内存地址

上面操作2依赖于操作1，但是操作3并不依赖于操作2，所以JVM可以以“优化”为目的对它们进行重排序，经过重排序后如下：

memory = allocate(); //1：分配对象的内存空间

instance = memory; //3：设置instance指向刚分配的内存地址（此时对象还未初始化）

initInstance(memory); //2：初始化对象

可以看到指令重排之后，操作 3 排在了操作 2 之前，即**引用instance指向内存memory时，这段崭新的内存还没有初始化**——即，引用instance指向了一个"被部分初始化的对象"。此时，如果另一个线程调用getInstance方法，由于instance已经指向了一块内存空间，从而if条件判为false，方法返回instance引用(return给调用者半个对象)，用户得到了没有完成初始化的“半个”单例。  
解决这个该问题，只需要将instance声明为volatile变量：

private static volatile Singleton instance;

#### volatile如何防止指令重排

**内存屏障**（**Memory Barrie**r，或有时叫做内存栅栏，Memory Fence）是一种CPU指令，用于控制特定条件下的重排序和内存可见性问题。Java编译器也会根据内存屏障的规则禁止重排序。

内存屏障可以被分为以下几种类型：  
**LoadLoad**屏障：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。  
**StoreStore**屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。  
**LoadStore**屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。  
**StoreLoa**d屏障：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。        在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。

     有的处理器的重排序规则较严，无需内存屏障也能很好的工作，Java编译器会在这种情况下不放置内存屏障。

volatile关键字通过“内存屏障”来防止指令被重排序。

为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。然而，对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，Java内存模型采取保守策略。

下面是基于保守策略的JMM内存屏障插入策略：

* 在每个volatile写操作的前面插入一个StoreStore屏障。
* 在每个volatile写操作的后面插入一个StoreLoad屏障。
* 在每个volatile读操作的前面插入一个LoadLoad屏障。
* 在每个volatile读操作的后面插入一个LoadStore屏障。

### 画一个线程的生命周期状态图。



### sleep和wait的区别。

1. sleep()方法是属于Thread类中的。而wait()方法属于Object类中的。
2. sleep方法没有**释放锁**，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。sleep不出让系统资源；wait是进入线程等待池等待，出让系统资源，其他线程可以占用CPU。
3. wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用
4. sleep必须**捕获异常**，而wait，notify和notifyAll不需要捕获异常

### sleep和sleep(0)的区别。

Sleep 接口均带有表示睡眠时间长度的参数 timeout。调用以上提到的 Sleep 接口，会有条件地将调用线程从当前处理器上移除，并且有可能将它从线程调度器的可运行队列中移除。这个条件取决于调用 Sleep 时timeout 参数。

当 timeout = 0， 即 Sleep(0)，如果线程调度器的可运行队列中有大于或等于当前线程优先级的就绪线程存在，操作系统会将当前线程从处理器上移除，调度其他优先级高的就绪线程运行；如果可运行队列中的没有就绪线程或所有就绪线程的优先级均低于当前线程优先级，那么当前线程会继续执行，就像没有调用 Sleep(0)一样。

当 timeout > 0 时，如：Sleep(1)，会引发线程上下文切换：调用线程会从线程调度器的可运行队列中被移除一段时间，这个时间段约等于 timeout 所指定的时间长度。为什么说约等于呢？是因为睡眠时间单位为毫秒，这与系统的时间精度有关。通常情况下，系统的时间精度为 10 ms，那么指定任意少于 10 ms但大于 0 ms 的睡眠时间，均会向上求值为 10 ms。

而调用 SwitchToThread() 方法，如果当前有其他就绪线程在线程调度器的可运行队列中，始终会让出一个时间切片给这些就绪线程，而不管就绪线程的优先级的高低与否。

### Lock与Synchronized的区别 。

1. 如果使用 synchronized ，如果A不释放，B将一直等下去，不能被中断。使用ReentrantLock，如果A不释放，可以使B在等待了足够长的时间以后，中断等待，而干别的事情。
2. synchronized是在JVM层面上实现的，是java中的一个关键字，不但可以通过一些监控工具监控synchronized的锁定，而且在代码执行时出现异常，JVM会自动释放锁定，但是使用Lock则不行，lock是通过代码实现的，要保证锁定一定会被释放，就必须将unLock()放到finally{}中。
3. 在资源竞争不是很激烈的情况下，Synchronized的性能要优于ReetrantLock，但是在资源竞争很激烈的情况下，Synchronized的性能会下降几十倍，但是ReetrantLock的性能能维持常态。

### 7、synchronized的原理是什么，一般用在什么地方(比如加在静态方法和非静态方法的区别，静态方法和非静态方法同时执行的时候会有影响吗)，解释以下名词：重排序，自旋锁，偏向锁，轻量级锁，可重入锁，公平锁，非公平锁，乐观锁，悲观锁。

### 8、用过哪些原子类，他们的原理是什么。

### 9、JUC下研究过哪些并发工具，讲讲原理。

### 10、用过线程池吗，如果用过，请说明原理，并说说newCache和newFixed有什么区别，构造函数的各个参数的含义是什么，比如coreSize，maxsize等。

### 11、线程池的关闭方式有几种，各自的区别是什么。

### 12、假如有一个第三方接口，有很多个线程去调用获取数据，现在规定每秒钟最多有10个线程同时调用它，如何做到。

### 13、spring的controller是单例还是多例，怎么保证并发的安全。

### 14、用三个线程按顺序循环打印abc三个字母，比如abcabcabc。

### 15、ThreadLocal用过么，用途是什么，原理是什么，用的时候要注意什么。

### 16、如果让你实现一个并发安全的链表，你会怎么做。

### 17、有哪些无锁数据结构，他们实现的原理是什么。

### 18、讲讲java同步机制的wait和notify。

### 19、CAS机制是什么，如何解决ABA问题。

### 20、多线程如果线程挂住了怎么办。

### 21、countdowlatch和cyclicbarrier的内部原理和用法，以及相互之间的差别(比如countdownlatch的await方法和是怎么实现的)。

### 22、对AbstractQueuedSynchronizer了解多少，讲讲加锁和解锁的流程，独占锁和公平所加锁有什么不同。

### 23、使用synchronized修饰静态方法和非静态方法有什么区别。

### 24、简述ConcurrentLinkedQueue和LinkedBlockingQueue的用处和不同之处。

### 25、导致线程死锁的原因？怎么解除线程死锁。

### 26、非常多个线程（可能是不同机器），相互之间需要等待协调，才能完成某种工作，问怎么设计这种协调方案。

### 27、用过读写锁吗，原理是什么，一般在什么场景下用。

### 28、开启多个线程，如果保证顺序执行，有哪几种实现方式，或者如何保证多个线程都执行完再拿到结果。

1.第一种使用join()方法

2.使用单线程池，这样就能根据传入的顺序执行线程

3.信号量

4.使用ReentrantLock

### 29、延迟队列的实现方式，delayQueue和时间轮算法的异同。

#### 简介

延时任务有别于定时任务，定时任务往往是固定周期的，有明确的触发时间。而延时任务一般没有固定的开始时间，它常常是由一个事件触发的，而在这个事件触发之后的一段时间内触发另一个事件。也就是说，任务事件生成时并不想让消费者立即拿到，而是延迟一定时间后才接收到该事件进行消费。

#### 场景

在订单系统中，一个用户某个时刻下单之后通常有30分钟的时间进行支付，如果30分钟之内没有支付成功，那么这个订单将自动进行过期处理。

#### 实现方案

##### 定时器轮询遍历数据库记录

###### 简介

起一个线程定时去扫数据库或者一个数据库定时Job，找到那些超时的数据，直接更新状态，或者拿出来执行一些操作。

###### 优点

实现简单，不会引入其他的技术，开发周期短。

###### 缺点

数据量大和分库分表后性能瓶颈和实现变得复杂。

##### JDK的DelayQueue

###### 简介

DelayQueue位于java.util.concurrent包下，由于是线程安全的它可以有多个消费者和多个生产者，DelayQueue本质是封装了一个PriorityQueue，使之线程安全，之所以要用到PriorityQueue，主要是需要排序。也许后插入的消息需要比队列中的其他消息提前触发，那么这个后插入的消息就需要最先被消费者获取，这就需要排序功能。PriorityQueue内部使用最小堆来实现排序队列，使用最小堆来实现优先级队列主要是因为最小堆在插入和获取时，时间复杂度相对都比较好，都是O(logN)。

###### 代码实现

/\*

\* 定义放在延迟队列中的对象，需要实现Delayed接口

\*/

**public** **class** DelayedTask **implements** Delayed {

**private** **int** \_expireInSecond = 0;

**public** DelayedTask(**int** delaySecond) {

Calendar cal = Calendar.getInstance();

cal.add(Calendar.SECOND, delaySecond);

\_expireInSecond = (**int**) (cal.getTimeInMillis() / 1000);

}

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

**long** d = (getDelay(TimeUnit.NANOSECONDS) - o.getDelay(TimeUnit.NANOSECONDS));

**return** (d == 0) ? 0 : ((d < 0) ? -1 : 1);

}

**public** **long** getDelay(TimeUnit unit) {

Calendar cal = Calendar.getInstance();

**return** \_expireInSecond - (cal.getTimeInMillis() / 1000);

}

}

下面定义了三个延迟任务，分别是10秒，5秒和15秒。依次入队列，期望5秒钟后，5秒的消息先被获取到，然后每个5秒钟，依次获取到10秒数据和15秒的那个数据。

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

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

//定义延迟队列

DelayQueue<DelayedTask> delayQueue = **new** DelayQueue<DelayedTask>();

//定义三个延迟任务

DelayedTask task1 = **new** DelayedTask(10);

DelayedTask task2 = **new** DelayedTask(5);

DelayedTask task3 = **new** DelayedTask(15);

delayQueue.add(task1);

delayQueue.add(task2);

delayQueue.add(task3);

System.***out***.println(sdf.format(**new** Date()) + " start");

**while** (delayQueue.size() != 0) {

//如果没到时间，该方法会返回

DelayedTask task = delayQueue.poll();

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

Date now = **new** Date();

System.***out***.println(sdf.format(now));

}

Thread.*sleep*(1000);

}

}

##### JDK ScheduledExecutorService

ScheduledExecutorService比上面一种DelayQueue更加实用。因为，一般来说，使用DelayQueue获取消息后触发事件都会实用多线程的方式执行，以保证其他事件能准时进行。而ScheduledThreadPoolExecutor就是对这个过程进行了封装，让大家更加方便的使用。同时在加强了部分功能，比如定时触发命令。

##### 时间轮（netty）

###### 简介

时间轮是一种非常惊艳的数据结构。其在Linux内核中使用广泛，是Linux内核定时器的实现方法和基础之一。按使用场景，大致可以分为两种时间轮：**原始时间轮**和**分层时间轮**。分层时间轮是原始时间轮的升级版本，来应对时间“槽”数量比较大的情况，对内存和精度都有很高要求的情况。我们延迟任务的场景一般只需要用到原始时间轮就可以了。

原始时间轮：如下图一个轮子，有8个“槽”，可以代表未来的一个时间。如果以秒为单位，中间的指针每隔一秒钟转动到新的“槽”上面，就好像手表一样。如果当前指针指在1上面，我有一个任务需要4秒以后执行，那么这个执行的线程回调或者消息将会被放在5上。那如果需要在20秒之后执行怎么办，由于这个环形结构槽数只到8，如果要20秒，指针需要多转2圈。位置是在2圈之后的5上面（20 % 8 + 1）。这个圈数需要记录在槽中的数据结构里面。这个数据结构最重要的是两个指针，一个是触发任务的函数指针，另外一个是触发的总第几圈数。时间轮可以用简单的数组或者是环形链表来实现。时间轮在插入和移除的复杂度都是O(1)。



##### 利用quartz等定时任务

quartz是一个企业级的开源的任务调度框架，quartz内部使用TreeSet来保存Trigger，如下图。Java中的TreeSet是使用TreeMap实现，TreeMap是一个红黑树实现。红黑树的插入和删除复杂度都是logN。和最小堆相比各有千秋。最小堆插入比红黑树快，删除顶层节点比红黑树慢。  
相比上述的三种轻量级的实现功能丰富很多。有专门的任务调度线程，和任务执行线程池。quartz功能强大，主要是用来执行周期性的任务，当然也可以用来实现延迟任务。但是如果只是实现一个简单的基于内存的延时任务的话，quartz就稍显庞大。

##### Redis的ZSet实现

##### 7.rabbitMq实现延时队列

# 十四、设计模式

## 前言：设计模式原则

总原则：**开闭原则（Open Close Principle）**

开闭原则就是说**对扩展开放**，**对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类等，后面的具体设计中我们会提到这点。

**1、单一职责原则**

不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，如若不然，就应该把类拆分。

**2、里氏替换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是**抽象化**。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

里氏替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

**5、迪米特法则（最少知道原则）（Demeter Principle）**

就是说：一个类对自己依赖的类知道的越少越好。也就是说无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

**6、合成复用原则（Composite Reuse Principle）**

原则是尽量首先使用合成/聚合的方式，而不是使用继承。

## 一、创建型模式：

### 1、简单工厂模式

**简介**：又叫做静态工厂方法（StaticFactory Method） 模式， 但**不属于** 23 种设计模式之一。列在此处仅作对比用。

**缺点：**不符合开闭原则，如果你要加一个产品类C，你就要修改工厂类里面的生成产品的代码

**public** **interface** Product {

//声明类所需继承的共同接口，也可以是抽象类

}

**public** **class** ProductA **implements** Product {

**public** ProductA() {

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

}

}

**public** **class** ProductB **implements** Product {

**public** ProductB() {

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

}

}

**public** **class** Factory {

//可以在工厂类中添加任何你所需要的逻辑

**public** **static** Product create(String str){

**if**(str.equalsIgnoreCase("ProductA")){//生成ProductA 硬编码

**return** **new** ProductA();

}**else** **if**(str.equalsIgnoreCase("ProductB")){ //生成ProductB

**return** **new** ProductB();

}

**return** **null**;

}

}

**public** **class** Client {

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

//调用Factory的静态方法生成所要的类

Factory.*create*("productA"); //硬编码

Factory.*create*("ProductB");

}

}

### 2、工厂方法模式

**简介：**工厂方法模式定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。

**意义：**工厂的目的是为了帮我们解决创建对象实例的问题，并且工厂能够隐藏类名及对象创建的细节，让我们无需关心对象的创建。提取其中个性化善变的部分为独立类，通过依赖注入以达到解耦、复用和方便后期维护拓展的目的。

**缺点：**如果产品类过多，要生成很多的工厂类。

**public** **interface** Provider { //接口

**public** Sender produce();

}

**public** **class** SendMailFactory **implements** Provider { //实现类1

@Override

**public** Sender produce(){

**return** **new** MailSender();

}

}

**public** **class** SendSmsFactory **implements** Provider{ //实现类2

@Override

**public** Sender produce() {

**return** **new** SmsSender();

}

}

**public** **class** Test { //调用工厂

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

Provider provider = **new** SendMailFactory();

Sender sender = provider.produce();

sender.Send();

}

}

### 3、抽象工厂模式：

**简介：**抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式

**特点：**同一产品簇下的产品用同一工厂，如：机枪和机枪子弹属于同一产品簇，手枪和手枪子弹属于同一产品簇。

**public** **interface** Gift {

//声明产品赠品的接口，当然也可以是抽象类，同样为了简单就不声明方法了

}

**public** **class** GiftA **implements** Gift {

**public** GiftA() {

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

}

}

**public** **class** GiftB **implements** Gift {

**public** GiftB(){

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

}

}

**public** **interface** Factory {

**public** Product createProduct();

**public** Gift createGift();

}

**public** **class** FactoryA **implements** Factory {

@Override

**public** Product createProduct() {

**return** **new** ProductA();

}

@Override

**public** Gift createGift() {

**return** **new** GiftA();

}

}

**public** **class** FactoryB **implements** Factory {

@Override

**public** Product createProduct() {

**return** **new** ProductB();

}

@Override

**public** Gift createGift() {

**return** **new** GiftB();

}

}

**public** **class** Client {

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

Factory factory;

factory = **new** FactoryA();

factory.createProduct();

factory.createGift();

factory = **new** FactoryB();

factory.createProduct();

factory.createGift();

}

}

**实际应用**：Spring中最为重要的组件就是IOC，而IOC中非常重要的部分就是应用了工厂模式的代码。而工厂模式依赖于Java的反射机制。工厂模式在spring 中的体现Spring Bean 的创建是典型的工厂模式， 这一系列的 **Bean** 工厂， 也即 IOC 容器为开发者管理对象间的依赖关系提供了很多便利和基础服务

### 4、单例模式

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

**public** **class** Singleton {

// 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载

**private** **static** Singleton *instance* = **null**;

**private** Singleton() { //私有构造方法，防止被实例化

}

// 静态工程方法，创建实例

**public** **static** Singleton getInstance() {

**if** (*instance* == **null**) {

**synchronized** (*instance*) {

**if** (*instance* == **null**) {

*instance* = **new** Singleton();

}

}

}

**return** *instance*;

}

// 如果该对象被用于序列化，可以保证对象在序列化前后保持一致

**public** Object readResolve() {

**return** *instance*;

}

}

### 建造者模式

**建造者模式（Builder Pattern）** 又名生成器模式，是一种对象构建模式。它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方法可以构造出不同表现（属性）的对象。

**建造者模式** 是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节

**public** **class** Meal {

**private** String food;

**private** String drink;

**public** String getFood() {

**return** food;

}

**public** **void** setFood(String food) {

**this**.food = food;

}

**public** String getDrink() {

**return** drink;

}

**public** **void** setDrink(String drink) {

**this**.drink = drink;

}

}

**public** **abstract** **class** MealBuilder {

Meal meal = **new** Meal();

**public** **abstract** **void** buildFood();

**public** **abstract** **void** buildDrink();

**public** Meal getMeal(){

**return** meal;

}

}

**public** **class** MealA **extends** MealBuilder{

**public** **void** buildDrink() {

meal.setDrink("可乐");

}

**public** **void** buildFood() {

meal.setFood("薯条");

}

}

**public** **class** MealB **extends** MealBuilder{

**public** **void** buildDrink() {

meal.setDrink("柠檬果汁");

}

**public** **void** buildFood() {

meal.setFood("鸡翅");

}

}

**public** **class** KFCWaiter {

**private** MealBuilder mealBuilder;

**public** KFCWaiter(MealBuilder mealBuilder) {

**this**.mealBuilder = mealBuilder;

}

**public** Meal construct(){

mealBuilder.buildFood();//准备食物

mealBuilder.buildDrink();//准备饮料

**return** mealBuilder.getMeal(); //准备完毕，返回一个完整的套餐给客户

}

}

**public** **class** Test {

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

MealA a = **new** MealA(); //套餐A

KFCWaiter waiter = **new** KFCWaiter(a);//准备套餐A的服务员

Meal mealA = waiter.construct(); //获得套餐

System.***out***.print("套餐A的组成部分:");

System.***out***.println("食物："+mealA.getFood()+"； "+"饮品："+mealA.getDrink());

}

}

**应用：**

1. StringBuilder 中的 append 方法使用了建造者模式

2.mybatis 中的建造者模式

org.apache.ibatis.session 包下的 SqlSessionFactoryBuilder 类

### 6、原型模式

**浅复制**：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

**深复制**：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

/\*\*

\*浅克隆

\*/

**public** **class** Pig **implements** Cloneable{

**private** String name;

**private** Date birthday;

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

Pig pig = (Pig)**super**.clone();

**return** pig;

}

}

/\*\*

\*深克隆一（手动对引用对象克隆）

\*/

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

Pig pig = (Pig)**super**.clone();

pig.birthday = (Date) pig.birthday.clone(); //深克隆

**return** pig;

}

/\*\*

\*深克隆 二（序列化）

\*/

**public** **class** Pig **implements** Serializable {

**private** String name;

**private** Date birthday;

**protected** Object deepClone() **throws** CloneNotSupportedException, IOException, ClassNotFoundException {

//将对象写入流中

ByteArrayOutputStream bao = **new** ByteArrayOutputStream();

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

oos.writeObject(**this**);

//将对象从流中取出

ByteArrayInputStream bis = **new** ByteArrayInputStream(bao.toByteArray());

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

**return** (ois.readObject());

}

}

## 二、结构型模式：

### 7、适配器模式：

(1).**类的适配器**：

/\*\*

\*拓展源类

\*/

**public** **class** Source {

**public** **void** method1() {

System.***out***.println("this is original method!");

}

}

/\*\*

\*拓展目标类

\*/

**public** **interface** Targetable {

**public** **void** method1();

**public** **void** method2();

}

/\*\*

\*适配器

\*/

**public** **class** Adapter **extends** Source **implements** Targetable {

@Override

**public** **void** method2() {

System.***out***.println("this is the targetable method!");

}

}

/\*\*

\*调用

\*/

**public** **class** AdapterTest {

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

Targetable target = **new** Adapter();

target.method1();

target.method2();

}

}

(2).**对象适配器模式**：

/\*\*

\*适配器

\*/

**public** **class** Wrapper **implements** Targetable {

**private** Source source;

**public** Wrapper(Source source){

**super**();

**this**.source = source;

}

@Override

**public** **void** method2() {

System.***out***.println("this is the targetable method!");

}

@Override

**public** **void** method1() {

source.method1();

}

}

(3).**接口适配器模式**：

第三种适配器模式是接口的适配器模式，接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行

**public** **interface** Sourceable {

**public** **void** method1();

**public** **void** method2();

}

**public** **abstract** **class** Wrapper2 **implements** Sourceable{

**public** **void** method1(){}

**public** **void** method2(){}

}

**public** **class** SourceSub1 **extends** Wrapper2 {

**public** **void** method1(){

System.***out***.println("the sourceable interface's first Sub1!");

}

}

**public** **class** SourceSub2 **extends** Wrapper2 {

**public** **void** method2(){

System.***out***.println("the sourceable interface's second Sub2!");

}

}

**public** **class** WrapperTest {

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

Sourceable source1 = **new** SourceSub1();

Sourceable source2 = **new** SourceSub2();

source1.method1();

source1.method2();

source2.method1();

source2.method2();

}

}

### 8、装饰器模式：（例：Spring aop代码增强）

**public** **interface** Sourceable {

**public** **void** method();

}

**public** **class** Source **implements** Sourceable {

@Override

**public** **void** method() {

System.out.println("the original method!");

}

}

**public** **class** Decorator **implements** Sourceable {

**private** Sourceable source;

**public** Decorator(Sourceable source){

**super**();

**this**.source = source;

}

@Override

**public** **void** method() {

System.out.println("before decorator!");

source.method();

System.out.println("after decorator!");

}

}

//测试类：

**public** **class** DecoratorTest {

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

Sourceable source = **new** Source();

Sourceable obj = **new** Decorator(source);

obj.method();

}

}

### 代理模式：

#### 1.静态代理：

**public** **interface** Sourceable {

**public** **void** method();

}

**public** **class** Source **implements** Sourceable {

@Override

**public** **void** method() {

System.***out***.println("the original method!");

}

}

**public** **class** Proxy **implements** Sourceable {

**private** Source source;

**public** Proxy(){

**super**();

**this**.source = **new** Source();

}

@Override

**public** **void** method() {

before();

source.method();

atfer();

}

**private** **void** atfer() {

System.***out***.println("after proxy!");

}

**private** **void** before() {

System.***out***.println("before proxy!");

}

}

//测试类：

**public** **class** ProxyTest {

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

Sourceable source = **new** Proxy();

source.method();

}

}

#### 动态代理

**public** **class** DynamicProxyHandler **implements** InvocationHandler {

**private** Object object;

**public** DynamicProxyHandler(**final** Object object) {

**this**.object = object;

}

@Override

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

System.***out***.println("前置方法");

Object result = method.invoke(object, args);

System.***out***.println("后置方法");

**return** result;

}

}

**public** **class** DynamicProxyTest {

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

Sourceable source= **new** Source();

Sourceable proxy = (Sourceable) Proxy.newProxyInstance(Source.**class**.getClassLoader(), **new**

Class[]{Source.**class**}, **new** DynamicProxyHandler(source));

proxy .buyHosue();

}

}

### 10、外观模式：

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度

**public** **class** CPU {

**public** **void** startup(){

System.***out***.println("cpu startup!");

}

**public** **void** shutdown(){

System.***out***.println("cpu shutdown!");

}

}

**public** **class** Memory {

**public** **void** startup(){

System.***out***.println("memory startup!");

}

**public** **void** shutdown(){

System.***out***.println("memory shutdown!");

}

}

**public** **class** Disk {

**public** **void** startup(){

System.***out***.println("disk startup!");

}

**public** **void** shutdown(){

System.***out***.println("disk shutdown!");

}

}

**public** **class** Computer {

**private** CPU cpu;

**private** Memory memory;

**private** Disk disk;

**public** Computer(){

cpu = **new** CPU();

memory = **new** Memory();

disk = **new** Disk();

}

**public** **void** startup(){

System.***out***.println("start the computer!");

cpu.startup();

memory.startup();

disk.startup();

System.***out***.println("start computer finished!");

}

**public** **void** shutdown(){

System.***out***.println("begin to close the computer!");

cpu.shutdown();

memory.shutdown();

disk.shutdown();

System.***out***.println("computer closed!");

}

}

//User类如下：

**public** **class** User {

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

Computer computer = **new** Computer();

computer.startup();

computer.shutdown();

}

}

### 桥接模式：

适配器：改变已有的两个接口，让他们相容。  
桥接模式：分离抽象化和实现，使两者的接口可以不同，目的是分离。  
所以说，如果你拿到两个已有模块，想让他们同时工作，那么你使用的适配器。如果你还什么都没有，但是想分开实现，那么桥接是一个选择

**public** **interface** Sourceable {

**public** **void** method();

}

**public** **class** SourceSub1 **implements** Sourceable {

@Override

**public** **void** method() {

System.***out***.println("this is the first sub!");

}

}

**public** **class** SourceSub2 **implements** Sourceable {

@Override

**public** **void** method() {

System.***out***.println("this is the second sub!");

}

}

**public** **abstract** **class** Bridge {

**private** Sourceable source;

**public** **void** method(){

source.method();

}

**public** Sourceable getSource() {

**return** source;

}

**public** **void** setSource(Sourceable source) {

**this**.source = source;

}

}

**public** **class** MyBridge **extends** Bridge {

**public** **void** method(){

getSource().method();

}

}

**public** **class** BridgeTest {

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

Bridge bridge = **new** MyBridge();

Sourceable source1 = **new** SourceSub1();//调用第一个对象

bridge.setSource(source1);

bridge.method();

Sourceable source2 = **new** SourceSub2(); //调用第二个对象

bridge.setSource(source2);

bridge.method();

}

}

### 组合模式

**经典案例**：系统目录结构；文件系统由目录和文件组成。每个目录都可以装内容。目录的内容可以是文件，也可以是目录。

**public** **class** TreeNode {

**private** String name;

**private** TreeNode parent;

**private** Vector<TreeNode> children = **new** Vector<TreeNode>();

**public** TreeNode(String name){

**this**.name = name;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** TreeNode getParent() {

**return** parent;

}

**public** **void** setParent(TreeNode parent) {

**this**.parent = parent;

}

//添加孩子节点

**public** **void** add(TreeNode node){

children.add(node);

}

//删除孩子节点

**public** **void** remove(TreeNode node){

children.remove(node);

}

//取得孩子节点

**public** Enumeration<TreeNode> getChildren(){

**return** children.elements();

}

}

**public** **class** Tree {

TreeNode root = **null**;

**public** Tree(String name) {

root = **new** TreeNode(name);

}

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

Tree tree = **new** Tree("A");

TreeNode nodeB = **new** TreeNode("B");

TreeNode nodeC = **new** TreeNode("C");

nodeB.add(nodeC);

tree.root.add(nodeB);

System.***out***.println("build the tree finished!");

}

}

### 享元模式：

**享元模式（Flyweight Pattern）**主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式

**public** **class** ConnectionPool {

**private** Vector<Connection> pool;

**private** String url = "jdbc:mysql://localhost:3306/test";

**private** String username = "root";

**private** String password = "root";

**private** String driverClassName = "com.mysql.jdbc.Driver";

**private** **int** poolSize = 100;

**private** **static** ConnectionPool *instance* = **null**;

Connection conn = **null**;

/\*构造方法，做一些初始化工作\*/

**private** ConnectionPool() {

pool = **new** Vector<Connection>(poolSize);

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

**try** {

Class.*forName*(driverClassName);

conn = DriverManager.getConnection(url, username, password);

pool.add(conn);

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

/\* 返回连接到连接池 \*/

**public** **synchronized** **void** release() {

pool.add(conn);

}

/\* 返回连接池中的一个数据库连接 \*/

**public** **synchronized** Connection getConnection() {

**if** (pool.size() > 0) {

Connection conn = pool.get(0);

pool.remove(conn);

**return** conn;

} **else** {

**return** **null**;

}

}

}

## 三、行为型模式

### 14、策略模式

策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

**主要思想：**使不同的算法可以被相互替换，而不影响客户端的使用

**优点：** 1、算法可以自由切换。 2、避免使用多重条件判断。 3、扩展性良好。

**缺点：** 1、策略类会增多。 2、所有策略类都需要对外暴露。

**public** **interface** ICalculator {

**public** **int** calculate(String exp);

}

**public** **abstract** **class** AbstractCalculator {

**public** **int**[] split(String exp,String opt){

String array[] = exp.split(opt);

**int** arrayInt[] = **new** **int**[2];

arrayInt[0] = Integer.*parseInt*(array[0]);

arrayInt[1] = Integer.*parseInt*(array[1]);

**return** arrayInt;

}

}

**public** **class** Plus **extends** AbstractCalculator **implements** ICalculator {

@Override

**public** **int** calculate(String exp) {

**int** arrayInt[] = split(exp,"\\+");

**return** arrayInt[0]+arrayInt[1];

}

}

**public** **class** Minus **extends** AbstractCalculator **implements** ICalculator {

@Override

**public** **int** calculate(String exp) {

**int** arrayInt[] = split(exp,"-");

**return** arrayInt[0]-arrayInt[1];

}

}

**public** **class** Multiply **extends** AbstractCalculator **implements** ICalculator {

@Override

**public** **int** calculate(String exp) {

**int** arrayInt[] = split(exp,"\\\*");

**return** arrayInt[0]\*arrayInt[1];

}

}

**public** **class** StrategyTest {

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

String exp = "2+8";

ICalculator cal = **new** Plus();

**int** result = cal.calculate(exp);

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

}

}

### 模板方法模式

**主要思想**：定义一个算法流程，将一些特定步骤的具体实现、延迟到子类。使得可以在不改变算法流程的情况下，通过不同的子类、来实现“定制”流程中的特定的步骤。

**public** **abstract** **class** AbstractCalculator {

**public** **final** **int** calculate(String exp,String opt){ //final 禁止重写 主方法，实现对本类其它方法的调用

**int** array[] = split(exp,opt);

**return** calculate(array[0],array[1]); //调用子类实现的方法

}

/\*被子类重写的方法\*/

**abstract** **public** **int** calculate(**int** num1,**int** num2);

**public** **int**[] split(String exp,String opt){

String array[] = exp.split(opt);

**int** arrayInt[] = **new** **int**[2];

arrayInt[0] = Integer.*parseInt*(array[0]);

arrayInt[1] = Integer.*parseInt*(array[1]);

**return** arrayInt;

}

}

**public** **class** Plus **extends** AbstractCalculator {

@Override

**public** **int** calculate(**int** num1,**int** num2) {

**return** num1 + num2;

}

}

**public** **class** StrategyTest {

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

String exp = "8+8";

AbstractCalculator cal = **new** Plus();

**int** result = cal.calculate(exp, "\\+");

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

}

}

### 观察者模式

**定义**

观察者模式：定义对象间一种**一对多**的依赖关系，使得每当一个对象改变状态，则所有**依赖它的对象**都会得到通知并**自动更新**。

**意义**

此设计模式最重要的作用就是 解耦！将观察者与被观察者解耦，使得他们之间的依赖性更小。

**public** **interface** Observer {

**public** **void** update();

}

**public** **class** Observer1 **implements** Observer {

@Override

**public** **void** update() {

System.out.println("observer1 has received!");

}

}

**public** **class** Observer2 **implements** Observer {

@Override

**public** **void** update() {

System.out.println("observer2 has received!");

}

}

**public** **interface** Subject {

**public** **void** add(Observer observer); //增加观察者

**public** **void** del(Observer observer); //删除观察者

**public** **void** notifyObservers(); //通知所有的观察者

**public** **void** operation(); //自身的操作

}

**public** **abstract** **class** AbstractSubject **implements** Subject {

**private** Vector<Observer> vector = **new** Vector<Observer>();

@Override

**public** **void** add(Observer observer) {

vector.add(observer);

}

@Override

**public** **void** del(Observer observer) {

vector.remove(observer);

}

@Override

**public** **void** notifyObservers() {

Enumeration<Observer> enumo = vector.elements();

**while**(enumo.hasMoreElements()){

enumo.nextElement().update();

}

}

}

**public** **class** MySubject **extends** AbstractSubject {

@Override

**public** **void** operation() {

System.out.println("update self!");

notifyObservers();

}

}

//测试类：

**public** **class** ObserverTest {

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

Subject sub = **new** MySubject();

sub.add(**new** Observer1());

sub.add(**new** Observer2());

sub.operation();

}

}

### 迭代子模式

**迭代子模式：**迭代子模式可以顺序访问一个聚集中的元素而不必暴露聚集的内部表象。

迭代器意义：https://blog.csdn.net/dxiaolai/article/details/76359332

**public** **interface** Collection {

**public** Iterator iterator();

**public** Object get(**int** i); //取得集合元素

**public** **int** size(); //取得集合大小

}

**public** **interface** Iterator {

**public** Object previous(); //前移

**public** Object next(); //后移

**public** **boolean** hasNext();

**public** Object first(); //取得第一个元素

}

**public** **class** MyCollection **implements** Collection {

**public** String string[] = {"A","B","C","D","E"};

@Override

**public** Iterator iterator() {

**return** **new** MyIterator(**this**);

}

@Override

**public** Object get(**int** i) {

**return** string[i];

}

@Override

**public** **int** size() {

**return** string.length;

}

}

**public** **class** MyIterator **implements** Iterator {

**private** Collection collection;

**private** **int** pos = -1;

**public** MyIterator(Collection collection){

**this**.collection = collection;

}

@Override

**public** Object previous() {

**if**(pos > 0){

pos--;

}

**return** collection.get(pos);

}

@Override

**public** Object next() {

**if**(pos<collection.size()-1){

pos++;

}

**return** collection.get(pos);

}

@Override

**public** **boolean** hasNext() {

**if**(pos<collection.size()-1){

**return** **true**;

}**else**{

**return** **false**;

}

}

@Override

**public** Object first() {

pos = 0;

**return** collection.get(pos);

}

}

**public** **class** Test {

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

Collection collection = **new** MyCollection();

Iterator it = collection.iterator();

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

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

}

}

}

### 18、责任链模式

责任链模式是一种对象的行为模式。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。Tomcat中的Filter就是使用了责任链模式

### 19、命令模式：

这个很好理解，命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉Struts的同学应该知道，Struts其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想！

**public** **interface** Command {

**public** **void** exe();

}

**public** **class** MyCommand **implements** Command {

**private** Receiver receiver;

**public** MyCommand(Receiver receiver) {

**this**.receiver = receiver;

}

@Override

**public** **void** exe() {

receiver.action();

}

}

**public** **class** Receiver {

**public** **void** action(){

System.***out***.println("command received!");

}

}

**public** **class** Invoker {

**private** Command command;

**public** Invoker(Command command) {

**this**.command = command;

}

**public** **void** action(){

command.exe();

}

}

**public** **class** Test {

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

Receiver receiver = **new** Receiver();

Command cmd = **new** MyCommand(receiver);

Invoker invoker = **new** Invoker(cmd);

invoker.action();

}

}

### 备忘录模式

**何时使用：**很多时候我们总是需要记录一个对象的内部状态，这样做的目的就是为了允许用户取消不确定或者错误的操作，能够恢复到他原先的状态。

**public** **class** Original {

**private** String value;

**public** String getValue() {

**return** value;

}

**public** **void** setValue(String value) {

**this**.value = value;

}

**public** Original(String value) {

**this**.value = value;

}

**public** Memento createMemento(){

**return** **new** Memento(value);

}

**public** **void** restoreMemento(Memento memento){

**this**.value = memento.getValue();

}

}

**public** **class** Memento {

**private** String value;

**public** Memento(String value) {

**this**.value = value;

}

**public** String getValue() {

**return** value;

}

**public** **void** setValue(String value) {

**this**.value = value;

}

}

**public** **class** Storage {

**private** Memento memento;

**public** Storage(Memento memento) {

**this**.memento = memento;

}

**public** Memento getMemento() {

**return** memento;

}

**public** **void** setMemento(Memento memento) {

**this**.memento = memento;

}

}

**public** **class** Test {

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

Original origi = **new** Original("egg");// 创建原始类

Storage storage = **new** Storage(origi.createMemento()); // 创建备忘录

System.***out***.println("初始化状态为：" + origi.getValue()); origi.setValue("niu"); // 修改原始类的状态

System.***out***.println("修改后的状态为：" + origi.getValue());

origi.restoreMemento(storage.getMemento()); // 回复原始类的状态

System.***out***.println("恢复后的状态为：" + origi.getValue());

}

}

### 21、状态模式

根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等。

**意图**：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。

**主要解决**：对象的行为依赖于它的状态（属性），代码中包含大量与对象状态有关的条件判断时，可以根据它的状态改变而改变它的相关行为。

**public** **class** State {

**private** String value;

**public** String getValue() {

**return** value;

}

**public** **void** setValue(String value) {

**this**.value = value;

}

**public** **void** method1(){

System.***out***.println("execute the first opt!");

}

**public** **void** method2(){

System.***out***.println("execute the second opt!");

}

}

**public** **class** Context {

**private** State state;

**public** Context(State state) {

**this**.state = state;

}

**public** State getState() {

**return** state;

}

**public** **void** setState(State state) {

**this**.state = state;

}

**public** **void** method() {

**if** (state.getValue().equals("state1")) {

state.method1();

} **else** **if** (state.getValue().equals("state2")) {

state.method2();

}

}

}

**public** **class** Test {

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

State state = **new** State();

Context context = **new** Context(state);

state.setValue("state1"); //设置第一种状态

context.method();

state.setValue("state2"); //设置第二种状态

context.method();

}

}

### 22、访问者模式

简单来说，访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。

**public** **interface** Visitor {

**public** **void** visit(Subject sub);

}

**public** **class** MyVisitor **implements** Visitor {

@Override

**public** **void** visit(Subject sub) {

System.***out***.println("visit the subject："+sub.getSubject());

}

}

**public** **interface** Subject {

**public** **void** accept(Visitor visitor);

**public** String getSubject();

}

**public** **class** MySubject **implements** Subject {

@Override

**public** **void** accept(Visitor visitor) {

visitor.visit(**this**);

}

@Override

**public** String getSubject() {

**return** "love";

}

}

**public** **class** Test {

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

Visitor visitor = **new** MyVisitor();

Subject sub = **new** MySubject();

sub.accept(visitor);

}

}

### 23、中介者模式

中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。如果使用中介者模式，只需关心和Mediator类的关系，具体类类之间的关系及调度交给Mediator就行，这有点像spring容器的作用

**public** **interface** Mediator {

**public** **void** createMediator();

**public** **void** workAll();

}

**public** **class** MyMediator **implements** Mediator {

**private** User user1;

**private** User user2;

**public** User getUser1() {

**return** user1;

}

**public** User getUser2() {

**return** user2;

}

@Override

**public** **void** createMediator() {

user1 = **new** User1(**this**);

user2 = **new** User2(**this**);

}

@Override

**public** **void** workAll() {

user1.work();

user2.work();

}

}

**public** **abstract** **class** User {

**private** Mediator mediator;

**public** Mediator getMediator(){

**return** mediator;

}

**public** User(Mediator mediator) {

**this**.mediator = mediator;

}

**public** **abstract** **void** work();

}

**public** **class** User1 **extends** User {

**public** User1(Mediator mediator){

**super**(mediator);

}

@Override

**public** **void** work() {

System.***out***.println("user1 exe!");

}

}

**public** **class** User2 **extends** User {

**public** User2(Mediator mediator){

**super**(mediator);

}

@Override

**public** **void** work() {

System.***out***.println("user2 exe!");

}

}

**public** **class** Test {

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

Mediator mediator = **new** MyMediator();

mediator.createMediator();

mediator.workAll();

}

}

### 解释器模式

**解释器（Interpreter）模式**：定义一个语言的文法，并且建立一个解释器来解释该语言中的句子，这里的“语言”是指使用规定格式和语法的代码。解释器模式是一种行为型模式。

**public** **interface** Expression {

**public** **int** interpret(Context context);

}

**public** **class** Plus **implements** Expression {

@Override

**public** **int** interpret(Context context) {

**return** context.getNum1()+context.getNum2();

}

}

**public** **class** Minus **implements** Expression {

@Override

**public** **int** interpret(Context context) {

**return** context.getNum1()-context.getNum2();

}

}

**public** **class** Context {

**private** **int** num1;

**private** **int** num2;

**public** Context(**int** num1, **int** num2) {

**this**.num1 = num1;

**this**.num2 = num2;

}

**public** **int** getNum1() {

**return** num1;

}

**public** **void** setNum1(**int** num1) {

**this**.num1 = num1;

}

**public** **int** getNum2() {

**return** num2;

}

**public** **void** setNum2(**int** num2) {

**this**.num2 = num2;

}

}

**public** **class** Test {

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

// 计算9+2-8的值

**int** result = **new** Minus().interpret((**new** Context(**new** Plus()

.interpret(**new** Context(9, 2)), 8)));

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

}

}

# 十五、分布式调度框架Elastic-job

## 一、搭建maven

下载解压

配置环境变量

## 二、安装zookeeper

下载上传tar包

tar -zxvf zookeeper-3.4.8.tar.gz

在Zookeeper根目录下添加两个文件夹data、logs

mkdir data

mkdir logs

进入到data文件夹下，添加一个文件myid，里面的内容先填写1

cd zookeeper-3.4.8/conf

cp zoo\_sample.cfg zoo.cfg

vi zoo.cfg

dataDir= /home/demo/zookeeper/zookeeper-3.4.8/data

dataLogDir= /home/demo/zookeeper/zookeeper-3.4.8/logs

server.1=10.0.2.15:2888:38888

export ZOOKEEPER\_HOME=/home/demo/zookeeper/zookeeper-3.4.8

export PATH=$ZOOKEEPER\_HOME/bin:$PATH

export PATH

cd ../bin

sh zkServer.sh start 启动

sh zkServer.sh restart 重启

sh zkServer.sh status 查看状态

测试

./zkCli.sh -server localhost:2181

连接失败

linux和window防火墙

360安全

## 三、elastic-job简介

Elastic-Job-Lite定位为轻量级无中心化解决方案，使用jar包的形式提供分布式任务的协调服务。外部依赖仅Zookeeper。

功能列表：

分布式调度协调

弹性扩容缩容

失效转移

错过执行作业重触发

作业分片一致性，保证同一分片在分布式环境中仅一个执行实例

自诊断并修复分布式不稳定造成的问题

支持并行调度

支持作业生命周期操作

丰富的作业类型

Spring整合以及命名空间提供

运维平台

基本概念：

分片

首先 节点之间建立关系 需要使用到 service iptables stop 命令

其次 java客户端连接zookeeper需要命令  vim /etc/sysconfig/iptables 编辑文件加入一句

-A INPUT -m state --state NEW -m tcp -p tcp --dport 2181 -j ACCEPT

elastic-job-console

在https://github.com/miguangying/elastic-job-lite-console下载zip包,解压并执行bin\start.sh,通过http://YOUR\_IP:8899/可访问

初始管理员账号root 密码root 可通过conf\auth.properties修改用户名及密码。

## 四、真实案例之批量发货流程

### 1.流程图



### 2.流程概述

1. .新增批量发货二级菜单，点击菜单进入批量发货界面。
2. 选择时间选择物流公司，导出发货模板。
3. 用户填写物流单号等相关信息，在同一页面导入发货文件。
4. 校验不通过给出相关提示，通过后保存文件，创建一类任务，给出成功提示。
5. 定时任务1，分解一类任务为10个一类子任务，创建二类任务，删除csv文件。
6. 定时任务2对定时任务1跑完的任务根据一类子任务编号进行分片并发发货，修改任 务、子任务‘订单状态、插入发货表。
7. 查看任务处理结果
8. 定时任务3定时清除任务两表7天前数据。

### 3.部署注册中心

部署三台zookeeper分别到两台商户中心服务器(168.1.14.214,168.1.14.215)和一台闲置运管服务器（168.1.14.234）组成定时任务注册中心集群。执行定时任务的3台机器注册到zookeeper集群，由zookeeper进行分片。

### 4.部署文件服务器

在现有文件服务器168.1.14.248，168.1.14.249，168.1.14.250，168.1.14.251（现有文件服务器）的/emall下安装fastDFS集群，存储excel文件。

### 5.定时任务服务

分布式定时任务采用elastic-job-lite,分为10片运行，代码部署到168.1.14.214,168.1.14.215，168.1.14.243三台服务器，每台服务器分别承担4，3，3片任务。开启失效转移，若有服务器不可用则，重新分片，由其他服务器承担当机服务器原先的分片。

### 6.定时任务服务部署

定时任务部署方案有两种，分为web项目war包部署和java工程jar包部署

**web项目war包部署**

1. .在搭建好的web项目中开发批处理代码，打成war包。
2. 部署到168.1.14.214,168.1.14.215 的/home/emall/Tomcat 和168.1.14.234 的/emall/Tomcat(重新安装tomcat，/home/emall/下tomcat部有运管和现有定时任务代码，
3. 此现有定时任务不可启动)
4. 日志打印到/home/emall/logs/elasticJob/elasticJob.log
5. 日志清理采用现有方案。

**java工程jar包部署**

（1）.在搭建好的java项目中开发批处理代码，打成可运行jar包.

(2).用提供的spring.handlers、spring.schemas、spring.tooling替换jar包中META-INF文件夹下的三个文件（eclipse导出可执行jar时，不同spring相关jar包中的此三文件会互相覆盖，提供的三文件为整合版）。

(3).将修改后的jar包分别上传至168.1.14.214,168.1.14.215,168.1.14.234三台服务器/home/emall/elasticJob下运行。

(4).日志打印到/home/emall/elasticJob/elasticJob.log

(5).编写脚本定时清理日志

# 十六、正则表达式

## 一、js正则

### 1、正则表达式作用

校验数据

替换文本

提取字符串

### 2、语法

**开始符**：^

**结束符**：$

+：号代表前面的字符必须至少出现一次

\*：号代表字符可以不出现，也可以出现一次或者多次

?：问号代表前面的字符最多只可以出现一次

**非打印字符**：

\cx 匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，

将c视为一个原义的 'c' 字符。

\f 匹配一个换页符。等价于 \x0c 和 \cL。

\n 匹配一个换行符。等价于 \x0a 和 \cJ。

\r 匹配一个回车符。等价于 \x0d 和 \cM。

\s 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。注意 Unicode 正则表达式会匹配全角空格符。

\S 匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。

\t 匹配一个制表符。等价于 \x09 和 \cI。

\v 匹配一个垂直制表符。等价于 \x0b 和 \cK。

**特殊字符**：

$ 匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。

要匹配 $ 字符本身，请使用 \$。

( ) 标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 \( 和 \)。

\* 匹配前面的子表达式零次或多次。要匹配 \* 字符，请使用 \\*。

+ 匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 \+。

. 匹配除换行符 \n 之外的任何单字符。要匹配 . ，请使用 \. 。

[ 标记一个中括号表达式的开始。要匹配 [，请使用 \[。

? 匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。

\ 将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如， 'n' 匹配字符 'n'。

'\n' 匹配换行符。序列 '\\' 匹配 "\"，而 '\(' 则匹配 "("。

^ 匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。要匹配 ^ 字符本身，请使用 \^。

{ 标记限定符表达式的开始。要匹配 {，请使用 \{。

| 指明两项之间的一个选择。要匹配 |，请使用 \|。

**限定符**：

\* 匹配前面的子表达式零次或多次。例如，zo\* 能匹配 "z" 以及 "zoo"。\* 等价于{0,}。

+ 匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。

? 匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 、 "does" 中的 "does" 、 "doxy" 中的 "do" 。

? 等价于 {0,1}。

{n} n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。

{n,}n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。

'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o\*'。

{n,m} m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的

前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

**定位符**：

^ 匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。

$ 匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。

\b 匹配一个单词边界，即字与空格间的位置。

\B 非单词边界匹配。

**元字符**：

\：将下一个字符标记为一个特殊字符、或一个原义字符、或一个 向后引用、或一个八进制转义符。

例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 "\(" 则匹配 "("。

^：匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。

$：匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。

\*：匹配前面的子表达式零次或多次。例如，zo\* 能匹配 "z" 以及 "zoo"。\* 等价于{0,}。

+：匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。

?：匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 。? 等价于 {0,1}。

{n}：n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。

{n,}：n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的

所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o\*'。

{n,m}：m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood"

中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

?：当该字符紧跟在任何一个其他限制符 (\*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少

的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串

"oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。

.：匹配除换行符（\n、\r）之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用像"(.|\n)"的模式。

(pattern)：匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在VBScript

中使用 SubMatches 集合，在JScript 中则使用 $0…$9 属性。要匹配圆括号字符，请使用 '\(' 或 '\)'。

(?:pattern) ：匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。

这在使用 "或" 字符 (|) 来组合一个模式的各个部分是很有用。例如， 'industr(?:y|ies) 就是一个比

'industry|industries' 更简略的表达式。

(?=pattern) ：正向肯定预查（look ahead positiveassert），在任何匹配pattern的字符串开始处匹配查找字符串。

这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，"Windows(?=95|98|NT|2000)"能匹配

"Windows2000"中的"Windows"，但不能匹配"Windows3.1"中的"Windows"。预查不消耗字符，也就是说，

在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。

(?!pattern) ：正向否定预查(negative assert)，在任何不匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，

也就是说，该匹配不需要获取供以后使用。例如"Windows(?!95|98|NT|2000)"能匹配"Windows3.1"中的"Windows"，

但不能匹配"Windows2000"中的"Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后

立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。

(?<=pattern)： 反向(look behind)肯定预查，与正向肯定预查类似，只是方向相反。例如，"(?<=95|98|NT|2000)Windows"能匹配

"2000Windows"中的"Windows"，但不能匹配"3.1Windows"中的"Windows"。

(?<!pattern)： 反向否定预查，与正向否定预查类似，只是方向相反。例如"(?<!95|98|NT|2000)Windows"能匹配"3.1Windows"

中的"Windows"，但不能匹配"2000Windows"中的"Windows"。

x|y ：匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。

[xyz]：字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。

[^xyz]：负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'、'l'、'i'、'n'。

[a-z]：字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。

[^a-z]：负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。

\b：匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，

但不能匹配 "verb" 中的 'er'。

\B：匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。

\cx ：匹配由 x 指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，

将c视为一个原义的 'c' 字符。

\d：匹配一个数字字符。等价于 [0-9]。

\D：匹配一个非数字字符。等价于 [^0-9]。

\f：匹配一个换页符。等价于 \x0c 和 \cL。

\n：匹配一个换行符。等价于 \x0a 和 \cJ。

\r：匹配一个回车符。等价于 \x0d 和 \cM。

\s：匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。

\S：匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。

\t：匹配一个制表符。等价于 \x09 和 \cI。

\v：匹配一个垂直制表符。等价于 \x0b 和 \cK。

\w：匹配字母、数字、下划线。等价于'[A-Za-z0-9\_]'。

\W：匹配非字母、数字、下划线。等价于 '[^A-Za-z0-9\_]'。

\xn：匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，'\x41' 匹配 "A"。

'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。

\num：匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。例如，'(.)\1' 匹配两个连续的相同字符。

\n：标识一个八进制转义值或一个向后引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，

如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。

\nm ：标识一个八进制转义值或一个向后引用。如果 \nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果 \nm

之前至少有 n 个获取，则 n 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字 (0-7)，则 \nm 将匹配八进制转义值 nm。

\nml：如果 n 为八进制数字 (0-3)，且 m 和 l 均为八进制数字 (0-7)，则匹配八进制转义值 nml。

\un：匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如， \u00A9 匹配版权符号 (?)。

### 3、预算符优先级

从上到下由高到低：

\ 转义符

(), (?:), (?=), [] 圆括号和方括号

\*, +, ?, {n}, {n,}, {n,m} 限定符

^, $, \任何元字符、任何字符 定位点和序列（即：位置和顺序）

| 替换，"或"操作字符具有高于替换运算符的优先级，使得"m|food"匹配"m"或"food"。若要匹配"mood"或"food"，请使用括号创建子表达式，从而产生"(m|f)ood"。

### 4、例：校验经度

var longrg = /^(\-|\+)?(((\d|[1-9]\d|1[0-7]\d|0{1,3})\.\d{0,6})|(\d|[1-9]\d|1[0-7]\d|0{1,3})|180\.0{0,6}|180)$/;

/^ //开始

(\-|\+)? //一个-号或者一个正号，或者没有符号

(

((\d | [1-9]\d | 1[0-7]\d | 0{1,3} )\. \d{0,6})

|(\d | [1-9]\d | 1[0-7]\d | 0{1,3} ) | 180\.0{0,6} | 180)

$/ //结束符

# 十七、反向代理Nginx

## 1.nginx简介

nginx:高性能的web服务器，在流量前1000名网站niginx使用量已超过apache，是一个高性能的http和反向代理服务器，也是一个IMAP/POP3/SMTP代理服务器，是由俄罗斯开发的。

## 2.nginx的安装

先安装pcre（依赖）：yum install pcre pcre-devel (正则表达式库及其依赖)

cd /usr/local/src

下载 : wget http://nginx.org/dowloading/nginx-1.4.2.tar.gz

解压：tar zxvf nginx-1.4.2.tar.gz

cd nginx-1.4.2

configure-prefix=/usr/local/nginx

make && make install

## 3.nginx启动

cd /usr/local/nginx

conf:配置文件

html：网页文件

logs:日志文件

sbin：主要二进制程序

cd sbin

./sbin/nginx (80端口被占用 )

netstat -antp(查看端口被哪个程序和进程占用) kill -9

## 4.nginx信号控制

master进程：管理worker进程

worker进程：真正干活的进程

kill -INT 进程号 快速关闭

kill -QUIT 进程号 优雅的关闭（请求结束后关闭）

hup:修改配置文件不需要重启（平滑的读取新配置文件）

kill -HUP 进程号

kill -USR1 进程号 (重读日志，在日志按月/日分割是有用 )

kill -USR2 进程号 平滑升级

kill -HUP `cat logs/nginx.pid` (自动获取进程号)

## 5.nginx命令控制

./sbin/nginx 启动

./sbin/nginx -s reload 重启

./sbin/nginx -s stop 关闭

./sbin/nginx -s reopen 重读配置文件

## 6.nginx配置

全局区

worker\_processes1；//有一个工作的子进程一般设置为cpu\*核数

Event{

worker\_connections 1024; //一个子进程最大允许连接数

}

http{

server{

listener 80;

server\_name z.com;

location / {

root z.com; 可写相对路径

index index.html

}

基于端口的虚拟主机配置

server{

listener 80;

server\_name 192.168.1.204;

location / {

root z.com; 可写相对路径

index index.html

}

server{

listener 2022;

server\_name z.com ;

location / {

root z.com; 可写相对路径

index index.html

}

}

}

## 7.nginx日志管理

针对不同的server做不同的Log

accress\_log logs/host.access.log main

main格式是定义好的一种日志格式，并起个名字，便于引用

log\_format main '$remote\_addr-$remote\_user[$time\_local] "$request"'

'$status $body\_bytes\_sent "$http\_referer"'

'"$http\_user\_agent" "$http\_x\_forwarded\_for"';

www.baidu.com/robots.txt //爬虫user-agent大全

----------------------------------------------------------

定时任务完成日志切割

date -s '2013 -09-21 19:00:38' 修改服务器时间

clock -w

date -d yesterday +%y%m%d 格式化时间

写脚本

vim runlog.sh

#!bin/bash

LOGPATH=/usr/local/nginx/logs/z.com.access.log

BASEPATH=/data/$(date -d yesterday +%Y%m)

mkdir -p $BASEPATH 按月创建备份文件夹

bak=$BASEPATH/$(date -d yesterday +%Y%m%d%H%M ).zcom.access.log

mv $LOGPATH $bak

touch $LOGPATH

kill -USR1 `cat /usr/local/nginx/logs/nginx.pid`

crontab -e

\*/1 \* \* \* \* sh /data/runlog.sh 定时执行脚本

## 8.location详解之精准匹配

location[=|~|~\*|^~] patt{

}

location= patt{} 精准匹配

location patt{} 一般匹配

location ~ patt{} 正则匹配

首先看有没有精准匹配，如果有则停止匹配

location = /index.html{

root /var/www/index.html

}

## 9.location详解之正则表达式

location ~ image{

root /var/www；

index index.html;

}

## 10.location总结

精准匹配：匹配上则停止匹配，如果没有匹配，则进行普通匹配（普通匹配多个匹配上，谁命中的最长就记忆谁）然后进行正则匹配正则匹配按照从上到下顺序匹配（只要命中立即返回）如果正则没有命中则返回普通匹配的记忆结果.

## 11.nginx rewrite语法解析

url重写

if

= 判断是否相等

~匹配正则表达式

~\*不区分大小写的正则表达式

-f-d-e判断是否为文件，目录，是否存在

if（$remote\_addr=192.168.1.100）{ #禁止此ip访问

return 403;

}

if（$http\_user\_agent ~ MSIE）{#识别ie浏览器

rewrite ^.\*$ ie.html；

break;#不写break，会陷入循环重定向

}

if（!-e $document\_root$fastcgi\_script\_name）{

return ^.\*$ /404.html;

break;

}

set 设置变量

return 返回状态码

break

rewrite

rewrite上下文不变。fastcgi\_script\_name不变

## 12.nginx+php的编译

apache一般是把php当作自己的一个模块来启动，

而nginx是把http请求变量（如get,user\_agent等）转发给php进程

即php独立进程，与nginx进行通信。称为fastcgi运行方式。

nginx+php的配置比较简单，把请求的信息转发给9000端口的php进程

让php进程处理指定目录下的php文件

location ~\.php${

root html;

fastcgi\_pass 127.0.0.1:9000;

fastcgi\_index index,php;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name

include fastcgi\_params;

}

碰到php文件把根目录定位到html

把请求上下文转交给php进程，

--------------------------------------------------------------------------

重写

location /ecshop{

rewrite "goods-(\d{1,7})\.html" /ecshop/goods.php?id=$1;

}

# 十八、前端知识

## Html

### 1.canvas

创建标签：

<canvas id="myCanvas" width="200" height="100" style="border:1px solid #000000;">

</canvas>

使用js画图形：

var c=document.getElementById("myCanvas");

var ctx=c.getContext("2d");

画矩形：

ctx.fillStyle="#FF0000"; //填充色

ctx.fillRect(0,0,150,75); //1.x,1.y,2.x,2.y 对角线坐标

画线：

ctx.strokeStyle="blue"; //线条的颜色

ctx.moveTo(0,0); //定义线条开始坐标

ctx.lineTo(200,100); //lineTo(x,y) 定义线条结束坐标

ctx.stroke(); //绘制线条:

画圆：

ctx.beginPath();//方法开始一条新路径，如果没有此方法则影响前面绘制的图形

ctx.arc(95,50,40,0,2\*Math.PI); x,y,r,start,stop

ctx.stroke();

绘制文字：

ctx.font="30px Arial"; //定义字体和大小

ctx.fillText("Hello World",20,50); 文字，x，y 绘制实心文字

ctx.strokeText("Hello World",20,50) ; //绘制空心文字

渐变色：

线条渐变：

当我们使用渐变对象，必须使用两种或两种以上的停止颜色。

var grd=ctx.createLinearGradient(0,0,200,0);

grd.addColorStop(0,"red");

grd.addColorStop(1,"blue");

ctx.fillStyle=grd;

ctx.fillRect(10,10,150,80);

径向圆渐变：

var grd=ctx.createRadialGradient(75,50,5,90,60,100);

grd.addColorStop(0,"red");

grd.addColorStop(1,"white");

ctx.fillStyle=grd;

ctx.fillRect(10,10,150,80);

将图片加载到画布：

var c=document.getElementById("myCanvas");

var ctx=c.getContext("2d");

var img=document.getElementById("scream");

img.onload = function() {

ctx.drawImage(img,10,10);

}

## CSS

list-style-type: disc; //默认。标记是实心圆。

-webkit-margin-before:1em // 是元素上边距数值为1倍字体高度即字体高为10px的话，上边距就为10px -webkit是浏览器内核 -webkit-margin-before是用于内核为-webkit的浏览器的

1、box-sizing 设置为 "border-box"。这可令浏览器呈现出带有指定宽度和高度的框，并把边框和内边距放入框中。

2、overflow:

visible 默认值。内容不会被修剪，会呈现在元素框之外。

hidden 内容会被修剪，并且其余内容是不可见的。

scroll 内容会被修剪，但是浏览器会显示滚动条以便查看其余的内容。

auto 如果内容被修剪，则浏览器会显示滚动条以便查看其余的内容。

inherit 规定应该从父元素继承 overflow 属性的值。

3、border:1px solid rgba(0,0,0,0.1);

意思是 设定元素的边框为1像素宽，实线，颜色使用rgba来表达。其中，rgba是CSS3中的属性，rgba括号中前3个数字代表着 red green blue三种颜色的rgb值，0-255，最后一个是设定这个颜色的透明度即alpha值。范围从0到1，越接近1，代表透明度越低

4、-webkit-user-select 和-moz-user-select 的值设为none.禁用复制粘贴

### 1、css语法

CSS 规则由两个主要的部分构成：选择器，以及一条或多条声明。

CSS声明总是以分号(;)结束，声明组以大括号({})括起来。

### 2、CSS Id 和Class

CSS 中 id 选择器以 "#" 来定义，**ID属性不要以数字开头**，数字开头的ID在 Mozilla/Firefox 浏览器中不起作用。

类选择器以一个点"."号显示，类名的第一个字符不能使用数字！它无法在 Mozilla 或 Firefox 中起作用。

### 3、如何插入样式表

#### 外部样式表

<head>

<link rel="stylesheet" type="text/css" href="mystyle.css">

</head>

#### 内部样式表

<head>

<style>

hr {color:sienna;}

p {margin-left:20px;}

body {background-image:url("images/back40.gif");}

</style>

</head>

#### 内联样式

<p style="color:sienna;margin-left:20px">这是一个段落。</p>

#### 多重样式优先级

内联样式）Inline style > （内部样式）Internal style sheet >（外部样式）External style sheet > 浏览器默认样式

### 4、CSS 背景

#### 背景颜色

background-color:#b0c4de;

CSS中，颜色值通常以以下方式定义:

十六进制 - 如："#ff0000"

RGB - 如："rgb(255,0,0)"

颜色名称 - 如："red"

#### 背景图像

background-image:url('paper.gif');

background-repeat:repeat-x; //图像水平平铺

background-repeat:no-repeat; //图片显示一次不平铺

background-position:right top;//图片的位置

background-attachment:fixed; //图片是否随页面移动，默认为scroll

scroll 背景图片随页面的其余部分滚动。这是默认

fixed 背景图像是固定的

inherit 指定background-attachment的设置应该从父元素继承

#### 背景- 简写属性

{background:#ffffff url('img\_tree.png') no-repeat right top;

当使用简写属性时，属性值的顺序为：:

background-color

background-image

background-repeat

background-attachment

background-position

以上属性无需全部使用，你可以按照页面的实际需要使用.

### 5、CSS 文本格式

#### 文本颜色

color:red;

文本的对齐方式

text-align:center;

left 把文本排列到左边。默认值：由浏览器决定。

right 把文本排列到右边。

center 把文本排列到中间。

justify 实现两端对齐文本效果。

inherit 规定应该从父元素继承 text-align 属性的值。

#### 文本修饰

text-decoration:none;

none 默认。定义标准的文本。

underline 定义文本下的一条线。

overline 定义文本上的一条线。

line-through 定义穿过文本下的一条线。

blink 定义闪烁的文本。

inherit 规定应该从父元素继承 text-decoration 属性的值。

#### 文本转换

text-transform:uppercase; //

text-transform:lowercase;//

text-transform:capitalize;// 首字母大写

#### 文本缩进

text-indent:50px;

### 6、CSS 字体

#### 通用字体系列

Serif Serif字体中字符在行的末端拥有额外的装饰

Sans-serif "Sans"是指无 - 这些字体在末端没有额外的装饰

Monospace 所有的等宽字符具有相同的宽度

#### 特定字体系列

一个特定的字体系列（如 "Times" 或 "Courier"）

#### 字体系列

font-family:"Times New Roman", Times, serif;//如果浏览器不支持第一种字体，他将尝试下一种字体， 如果字体系列的名称超过一个字，它必须用引号。

#### 字体样式

font-style:normal;

font-style:italic;

font-style:oblique;//和斜体类似，支持不太好

#### 字体大小

font-size:40px;

font-size:2.5em;//为了避免Internet Explorer 中无法调整文本的问题，许多开发者使用 em 单位代替像素。1em和当前字体大小相等。在浏览器中默认的文字大小是16px。使用百分比和EM组合在所有浏览器的解决方案中，设置 <body>元素的默认字体大小的是百分比：

body {font-size:100%;}

h1 {font-size:2.5em;}

h2 {font-size:1.875em;}

p {font-size:0.875em;}

### 7、CSS 链接

#### 颜色修饰

a:link {color:#000000;} /\* 未访问链接\*/

a:visited {color:#00FF00;} /\* 已访问链接 \*/

a:hover {color:#FF00FF;} /\* 鼠标移动到链接上 \*/

a:active {color:#0000FF;} /\* 鼠标点击时 \*/

#### 文本修饰

a:link {text-decoration:none;}

a:visited {text-decoration:none;}

a:hover {text-decoration:underline;}

a:active {text-decoration:underline;}

### 8、CSS 列表

ul.a {list-style-type: circle;}//空心圆

ul.b {list-style-type: square;}//实心方

ol.c {list-style-type: upper-roman;}//罗马计数

ol.d {list-style-type: lower-alpha;}//字母计数

图像作为列标记

ul

{

list-style-image: url('sqpurple.gif');

}

### 9、CSS 表格

表格边框

border: 1px solid black;

折叠边框

border-collapse:collapse;

border-collapse 属性设置表格的边框是否被折叠成一个单一的边框或隔开：

表格文字对齐

text-align:right; //水平对齐

vertical-align:bottom;//垂直对齐

### 10、css盒子模型

Margin(外边距) - 清除边框外的区域，外边距是透明的。

Border(边框) - 围绕在内边距和内容外的边框。

Padding(内边距) - 清除内容周围的区域，内边距是透明的。

Content(内容) - 盒子的内容，显示文本和图像。

### 11、CSS 边框

border-style

border-top-style:dotted;

border-right-style:solid;

border-bottom-style:dotted;

border-left-style:solid;

none: 默认无边框

dotted::定义一个点线边框

dashed: 定义一个虚线边框

solid: 定义实线边框

double: 定义两个边框。 两个边框的宽度和 border-width 的值相同

groove: 定义3D沟槽边框。效果取决于边框的颜色值

ridge: 定义3D脊边框。效果取决于边框的颜色值

inset:定义一个3D的嵌入边框。效果取决于边框的颜色值

outset: 定义一个3D突出边框。 效果取决于边框的颜色值

### 12、CSS 轮廓（outline）

outline

outline-color

outline-style

outline-width

Inherit

outline-color

color-name

hex-number

rgb-number

invert

inherit

outline-style

none

dotted

dashed

solid

double

groove

ridge

inset

outset

inherit

outline-width

thin

medium

thick

length

inherit

### 13、CSS margin

margin:25px 50px 75px 100px;

上边距为25px

右边距为50px

下边距为75px

左边距为100px

margin:25px 50px 75px;

上边距为25px

左右边距为50px

下边距为75px

margin:25px 50px;

上下边距为25px

左右边距为50px

margin:25px;

所有的4个边距都是25px

### 14、CSS Padding（填充）

padding:25px 50px 75px 100px;

上填充为25px

右填充为50px

下填充为75px

左填充为100px

padding:25px 50px 75px;

上填充为25px

左右填充为50px

下填充为75px

padding:25px 50px;

上下填充为25px

左右填充为50px

padding:25px;

所有的填充都是25px

### 15、CSS 分组和嵌套选择器

#### 分组选择器

h1,h2,p

{

color:green;

}

#### 嵌套选择器

.marked p //为所有class="marked"元素内的p元素指定一个样式

{

color:white;

}

### 16、CSS尺寸

height 设置元素的高度。

line-height 设置行高。

max-height 设置元素的最大高度。

max-width 设置元素的最大宽度。

min-height 设置元素的最小高度。

min-width 设置元素的最小宽度。

width 设置元素的宽度。

### 17、CSS Display(显示) 与 Visibility（可见性）

visibility:hidden可以隐藏某个元素，但隐藏的元素仍需占用与未隐藏之前一样的空间。

display:none可以隐藏某个元素，且隐藏的元素不会占用任何空间

### 18、CSS Positioning(定位)

position 属性的四个值：

static // 默认值，即没有定位，元素出现在正常的流中

relative // 相对定位元素的定位是相对其正常位置。

fixed // 元素的位置相对于浏览器窗口是固定位置。

absolute //absolute 定位使元素的位置与文档流无关，因此不占据空间。absolute 定位的元素和其他元素重叠。

### 19、CSS Float

float:right; //一个浮动元素会尽量向左或向右移动，直到它的外边缘碰到包含框或另一个浮动框的边框为止。如果你把几个浮动的元素放到一起，如果有空间的话，它们将彼此相邻。

clear:both; //clear 属性指定元素两侧不能出现浮动元素

### 20、CSS组合选择符

#### 后代选取器(以空格分隔)

div p

{

background-color:yellow;

}

#### 子元素选择器(以大于号分隔）

div>p

{

background-color:yellow;

}

#### 相邻兄弟选择器（以加号分隔）

div+p

{

background-color:yellow;

}

#### 普通兄弟选择器（以破折号分隔）

div~p

{

background-color:yellow;

}

### 21、CSS 伪类(Pseudo-classes)

:checked input:checked 选择所有选中的表单元素

:disabled input:disabled 选择所有禁用的表单元素

:empty p:empty 选择所有没有子元素的p元素

:enabled input:enabled 选择所有启用的表单元素

:first-of-type p:first-of-type 选择每个父元素是p元素的第一个p子元素

:in-range input:in-range 选择元素指定范围内的值

:invalid input:invalid 选择所有无效的元素

:last-child p:last-child 选择所有p元素的最后一个子元素

:last-of-type p:last-of-type 选择每个p元素是其母元素的最后一个p元素

:not(selector) :not(p) 选择所有p以外的元素

:nth-child(n) p:nth-child(2) 选择所有p元素的第二个子元素

:nth-last-child(n) p:nth-last-child(2) 选择所有p元素倒数的第二个子元素

:nth-last-of-type(n) p:nth-last-of-type(2) 选择所有p元素倒数的第二个为p的子元素

:nth-of-type(n) p:nth-of-type(2) 选择所有p元素第二个为p的子元素

:only-of-type p:only-of-type 选择所有仅有一个子元素为p的元素

:only-child p:only-child 选择所有仅有一个子元素的p元素

:optional input:optional 选择没有"required"的元素属性

:out-of-range input:out-of-range 选择指定范围以外的值的元素属性

:read-only input:read-only 选择只读属性的元素属性

:read-write input:read-write 选择没有只读属性的元素属性

:required input:required 选择有"required"属性指定的元素属性

:root root 选择文档的根元素

:target #news:target 选择当前活动#news元素(点击URL包含锚的名字)

:valid input:valid 选择所有有效值的属性

:link a:link 选择所有未访问链接

:visited a:visited 选择所有访问过的链接

:active a:active 选择正在活动链接

:hover a:hover 把鼠标放在链接上的状态

:focus input:focus 选择元素输入后具有焦点

:first-letter p:first-letter 选择每个<p> 元素的第一个字母

:first-line p:first-line 选择每个<p> 元素的第一行

:first-child p:first-child 选择器匹配属于任意元素的第一个子元素的 <]p> 元素

:before p:before 在每个<p>元素之前插入内容

:after p:after 在每个<p>元素之后插入内容

:lang(language) p:lang(it) 为<p>元素的lang属性选择一个开始值

### 22、CSS 伪元素

:link a:link 选择所有未访问链接

:visited a:visited 选择所有访问过的链接

:active a:active 选择正在活动链接

:hover a:hover 把鼠标放在链接上的状态

:focus input:focus 选择元素输入后具有焦点

:first-letter p:first-letter 选择每个<p> 元素的第一个字母

:first-line p:first-line 选择每个<p> 元素的第一行

:first-child p:first-child 选择器匹配属于任意元素的第一个子元素的 <]p> 元素

:before p:before 在每个<p>元素之前插入内容

:after p:after 在每个<p>元素之后插入内容

:lang(language) p:lang(it) 为<p>元素的lang属性选择一个开始值

## 三、JS高级教程

### 1、JavaScript 对象

1、person=new Object();

person.firstname="John";

person.lastname="Doe";

person.age=50;

person.eyecolor="blue";

2、person={firstname:"John",lastname:"Doe",age:50,eyecolor:"blue"};

3、function person(firstname,lastname,age,eyecolor)

{

this.firstname=firstname;

this.lastname=lastname;

this.age=age;

this.eyecolor=eyecolor;

}

### 2、JavaScript Number 对象

var myNumber=128;

myNumber.toString(16); // 返回 80

myNumber.toString(8); // 返回 200

myNumber.toString(2); // 返回 10000000

var x = 1000 / "Apple";

isNaN(x); // 返回 true

var y = 100 / "1000";

isNaN(y); // 返回 false

### 3、JavaScript 字符串（String） 对象

var carname='Volvo XC60';

var character=carname[7];

属性:

length

prototype

constructor

方法:

charAt() //返回指定位置的字符

charCodeAt() //返回指定位置的字符的 Unicode 编码

concat() //用于连接两个或多个数组。该方法不会改变现有的数组，而仅仅会返回被连接数组的一个副本

fromCharCode() //接受一个指定的 Unicode 值，然后返回一个字符串。

indexOf()

lastIndexOf()

match() //方法可在字符串内检索指定的值，或找到一个或多个正则表达式的匹配。该方法类似 indexOf() 和 lastIndexOf()，但是它返回指定的值，而不是字符串的位置。

replace()

search()

slice() //从已有的数组中返回选定的元素

split() //用于把一个字符串分割成字符串数组

substr() //可在字符串中抽取从 start 下标开始的指定数目的字符

substring() //用于提取字符串中介于两个指定下标之间的字符。

toLowerCase()

toUpperCase()

valueOf() //可返回 Boolean 对象的原始值。

### 4、JavaScript Date（日期） 对象

getFullYear()

使用 getFullYear() 获取年份。

getTime()

getTime() 返回从 1970 年 1 月 1 日至今的毫秒数。

setFullYear()

如何使用 setFullYear() 设置具体的日期。

toUTCString()

如何使用 toUTCString() 将当日的日期（根据 UTC）转换为字符串。

getDay()

如何使用 getDay() 和数组来显示星期，而不仅仅是数字。

### 5、JavaScript Array（数组） 对象

创建数组

#### (1). 常规方式

var myCars=new Array();

myCars[0]="Saab";

myCars[1]="Volvo";

myCars[2]="BMW";

#### (2).简洁方式

var myCars=new Array("Saab","Volvo","BMW");

#### (3). 字面

var myCars=["Saab","Volvo","BMW"];

### 6、JavaScript Boolean（布尔） 对象

如果布尔对象无初始值或者其值为:

0

-0

null

""

false

undefined

NaN

那么对象的值为 false。否则，其值为 true（即使当变量值为字符串 "false" 时）！

### 7、JavaScript Math（算数） 对象

#### (1).Math 对象属性

E 返回算术常量 e，即自然对数的底数（约等于2.718）。

LN2 返回 2 的自然对数（约等于0.693）。

LN10 返回 10 的自然对数（约等于2.302）。

LOG2E 返回以 2 为底的 e 的对数（约等于 1.414）。

LOG10E 返回以 10 为底的 e 的对数（约等于0.434）。

PI 返回圆周率（约等于3.14159）。

SQRT1\_2 返回返回 2 的平方根的倒数（约等于 0.707）。

SQRT2 返回 2 的平方根（约等于 1.414）。

#### (2).Math 对象方法

abs(x) 返回 x 的绝对值。

acos(x) 返回 x 的反余弦值。

asin(x) 返回 x 的反正弦值。

atan(x) 以介于 -PI/2 与 PI/2 弧度之间的数值来返回 x 的反正切值。

atan2(y,x) 返回从 x 轴到点 (x,y) 的角度（介于 -PI/2 与 PI/2 弧度之间）。

ceil(x) 对数进行上舍入。

cos(x) 返回数的余弦。

exp(x) 返回 Ex 的指数。

floor(x) 对 x 进行下舍入。

log(x) 返回数的自然对数（底为e）。

max(x,y,z,...,n) 返回 x,y,z,...,n 中的最高值。

min(x,y,z,...,n) 返回 x,y,z,...,n中的最低值。

pow(x,y) 返回 x 的 y 次幂。

random() 返回 0 ~ 1 之间的随机数。

round(x) 把数四舍五入为最接近的整数。

sin(x) 返回数的正弦。

sqrt(x) 返回数的平方根。

tan(x) 返回角的正切。

### 8、JavaScript RegExp 对象

#### (1).语法

var patt=new RegExp(pattern,modifiers);

或者更简单的方式:

var patt=/pattern/modifiers;

pattern（模式） 描述了表达式的模式

modifiers(修饰符) 用于指定全局匹配、区分大小写的匹配和多行匹配

注意：当使用构造函数创造正则对象时，需要常规的字符转义规则（在前面加反斜杠 \）。比如，以下是等价的：

var re = new RegExp("\\w+");

var re = /\w+/;

#### (2).修饰符

修饰符用于执行区分大小写和全局匹配:

修饰符 描述

i 执行对大小写不敏感的匹配。

g 执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。

m 执行多行匹配。

#### (3).方括号

方括号用于查找某个范围内的字符：

表达式 描述

[abc] 查找方括号之间的任何字符。

[^abc] 查找任何不在方括号之间的字符。

[0-9] 查找任何从 0 至 9 的数字。

[a-z] 查找任何从小写 a 到小写 z 的字符。

[A-Z ] 查找任何从大写 A 到大写 Z 的字符。

[A-z] 查找任何从大写 A 到小写 z 的字符。

[adgk] 查找给定集合内的任何字符。

[^adgk] 查找给定集合外的任何字符。

(red|blue|green) 查找任何指定的选项。

#### (4).元字符

元字符（Metacharacter）是拥有特殊含义的字符：

元字符 描述

. 查找单个字符，除了换行和行结束符。

\w 查找单词字符。

\W 查找非单词字符。

\d 查找数字。

\D 查找非数字字符。

\s 查找空白字符。

\S 查找非空白字符。

\b 匹配单词边界。

\B 匹配非单词边界。

\0 查找 NULL 字符。

\n 查找换行符。

\f 查找换页符。

\r 查找回车符。

\t 查找制表符。

\v 查找垂直制表符。

\xxx 查找以八进制数 xxx 规定的字符。

\xdd 查找以十六进制数 dd 规定的字符。

\uxxxx 查找以十六进制数 xxxx 规定的 Unicode 字符。

#### (5).量词

量词 描述

n+ 匹配任何包含至少一个 n 的字符串。

例如，/a+/ 匹配 "candy" 中的 "a"，"caaaaaaandy" 中所有的 "a"。

n\* 匹配任何包含零个或多个 n 的字符串。

例如，/bo\*/ 匹配 "A ghost booooed" 中的 "boooo"，"A bird warbled" 中的 "b"，但是不匹配 "A goat grunted"。

n? 匹配任何包含零个或一个 n 的字符串。

例如，/e?le?/ 匹配 "angel" 中的 "el"，"angle" 中的 "le"。

n{X} 匹配包含 X 个 n 的序列的字符串。

例如，/a{2}/ 不匹配 "candy," 中的 "a"，但是匹配 "caandy," 中的两个 "a"，且匹配 "caaandy." 中的前两个 "a"。

n{X,} X 是一个正整数。前面的模式 n 连续出现至少 X 次时匹配。

例如，/a{2,}/ 不匹配 "candy" 中的 "a"，但是匹配 "caandy" 和 "caaaaaaandy." 中所有的 "a"。

n{X,Y} X 和 Y 为正整数。前面的模式 n 连续出现至少 X 次，至多 Y 次时匹配。

例如，/a{1,3}/ 不匹配 "cndy"，匹配 "candy," 中的 "a"，"caandy," 中的两个 "a"，匹配 "caaaaaaandy" 中的前面三个 "a"。注意，当匹配 "caaaaaaandy" 时，即使原始字符串拥有更多的 "a"，匹配项也是 "aaa"。

n{X,} 匹配包含至少 X 个 n 的序列的字符串。

n$ 匹配任何结尾为 n 的字符串。

^n 匹配任何开头为 n 的字符串。

?=n 匹配任何其后紧接指定字符串 n 的字符串。

?!n 匹配任何其后没有紧接指定字符串 n 的字符串。

#### (6).RegExp 对象方法

方法 描述

compile 编译正则表达式。

exec 检索字符串中指定的值。返回找到的值，并确定其位置。

test 检索字符串中指定的值。返回 true 或 false。

#### (7).支持正则表达式的 String 对象的方法

方法 描述

search 检索与正则表达式相匹配的值。

match 找到一个或多个正则表达式的匹配。

replace 替换与正则表达式匹配的子串。

split 把字符串分割为字符串数组。

### 9、JS浏览器BOM

#### (1).javascript window

有三种方法能够确定浏览器窗口的尺寸。

对于Internet Explorer、Chrome、Firefox、Opera 以及 Safari：

window.innerHeight - 浏览器窗口的内部高度(包括滚动条)

window.innerWidth - 浏览器窗口的内部宽度(包括滚动条)

对于 Internet Explorer 8、7、6、5：

document.documentElement.clientHeight

document.documentElement.clientWidth

或者

document.body.clientHeight

document.body.clientWidth

window.open() - 打开新窗口

window.close() - 关闭当前窗口

window.moveTo() - 移动当前窗口

window.resizeTo() - 调整当前窗口的尺寸

#### (2).JavaScript Window Screen

screen.availWidth - 可用的屏幕宽度

screen.availHeight - 可用的屏幕高度

#### (3).JavaScript Window Location

location.hostname 返回 web 主机的域名

location.pathname 返回当前页面的路径和文件名

location.port 返回 web 主机的端口 （80 或 443）

location.protocol 返回所使用的 web 协议（http:// 或 https://）

#### (4).Window History

history.back() - 与在浏览器点击后退按钮相同

history.forward() - 与在浏览器中点击按钮向前相同

#### (5).Window Navigator

txt = "<p>浏览器代号: " + navigator.appCodeName + "</p>";

txt+= "<p>浏览器名称: " + navigator.appName + "</p>";

txt+= "<p>浏览器版本: " + navigator.appVersion + "</p>";

txt+= "<p>启用Cookies: " + navigator.cookieEnabled + "</p>";

txt+= "<p>硬件平台: " + navigator.platform + "</p>";

txt+= "<p>用户代理: " + navigator.userAgent + "</p>";

txt+= "<p>用户代理语言: " + navigator.systemLanguage + "</p>";

#### (6).JavaScript 弹窗

window.alert("sometext");

window.alert() 方法可以不带上window对象，直接使用alert()方法。

window.confirm("sometext");

window.confirm() 方法可以不带上window对象，直接使用confirm()方法。

window.prompt("sometext","defaultvalue"); //提示输入框

window.prompt() 方法可以不带上window对象，直接使用prompt()方法。

#### (7).JavaScript 计时事件

setInterval() - 间隔指定的毫秒数不停地执行指定的代码。

setTimeout() - 暂停指定的毫秒数后执行指定的代码

<script>

var myVar=setInterval(function(){myTimer()},1000);

function myTimer(){

var d=new Date();

var t=d.toLocaleTimeString();

document.getElementById("demo").innerHTML=t;

}

function myStopFunction(){

clearInterval(myVar);

}

</script>

<script>

var myVar;

function myFunction(){

myVar=setTimeout(function(){alert("Hello")},3000);

}

function myStopFunction(){

clearTimeout(myVar);

}

</script>

#### (8).JavaScript Cookie

<script>

function setCookie(cname,cvalue,exdays){

var d = new Date();

d.setTime(d.getTime()+(exdays\*24\*60\*60\*1000));

var expires = "expires="+d.toGMTString();

document.cookie = cname+"="+cvalue+"; "+expires;

}

function getCookie(cname){

var name = cname + "=";

var ca = document.cookie.split(';');

for(var i=0; i<ca.length; i++) {

var c = ca[i].trim();

if (c.indexOf(name)==0) return c.substring(name.length,c.length);

}

return "";

}

function checkCookie(){

var user=getCookie("username");

if (user!=""){

alert("Welcome again " + user);

}

else {

user = prompt("Please enter your name:","");

if (user!="" && user!=null){

setCookie("username",user,30);

}

}

}

</script>

## 四、JQuery

### 1、jQuery 安装

添加jQuery

本地：从 jquery.com 下载 jQuery 库

<script src="jquery-1.10.2.min.js"></script>

远程：从 CDN 中载入 jQuery, 如从 Google 中加载 jQuery

百度 CDN:

<script src="https://apps.bdimg.com/libs/jquery/2.1.4/jquery.min.js">

又拍云 CDN:

<script src="http://upcdn.b0.upaiyun.com/libs/jquery/jquery2.0.2.min.js">

新浪 CDN:

<script src="http://lib.sinaapp.com/js/jquery/2.0.2/jquery-2.0.2.min.js">

Google CDN:

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">

Microsoft CDN:

<script src="http://ajax.htmlnetcdn.com/ajax/jQuery/jquery-1.10.2.min.js">

下载 jQuery

Production version - 用于实际的网站中，已被精简和压缩。

Development version - 用于测试和开发（未压缩，是可读的代码）

### jQuery 选择器

|  |  |  |
| --- | --- | --- |
| **选择器** | **实例** | **选取** |
| [\*](https://www.w3school.com.cn/jquery/selector_all.asp) | $("\*") | 所有元素 |
| [#*id*](https://www.w3school.com.cn/jquery/selector_id.asp) | $("#lastname") | id="lastname" 的元素 |
| [.*class*](https://www.w3school.com.cn/jquery/selector_class.asp) | $(".intro") | 所有 class="intro" 的元素 |
| [*element*](https://www.w3school.com.cn/jquery/selector_element.asp) | $("p") | 所有 <p> 元素 |
| .*class*.*class* | $(".intro.demo") | 所有 class="intro" 且 class="demo" 的元素 |
| [:first](https://www.w3school.com.cn/jquery/selector_first.asp) | $("p:first") | 第一个 <p> 元素 |
| [:last](https://www.w3school.com.cn/jquery/selector_last.asp) | $("p:last") | 最后一个 <p> 元素 |
| [:even](https://www.w3school.com.cn/jquery/selector_even.asp) | $("tr:even") | 所有偶数 <tr> 元素 |
| [:odd](https://www.w3school.com.cn/jquery/selector_odd.asp) | $("tr:odd") | 所有奇数 <tr> 元素 |
| [:eq(*index*)](https://www.w3school.com.cn/jquery/selector_eq.asp) | $("ul li:eq(3)") | 列表中的第四个元素（index 从 0 开始） |
| [:gt(*no*)](https://www.w3school.com.cn/jquery/selector_gt.asp) | $("ul li:gt(3)") | 列出 index 大于 3 的元素 |
| [:lt(*no*)](https://www.w3school.com.cn/jquery/selector_lt.asp) | $("ul li:lt(3)") | 列出 index 小于 3 的元素 |
| :not(*selector*) | $("input:not(:empty)") | 所有不为空的 input 元素 |
| [:header](https://www.w3school.com.cn/jquery/selector_header.asp) | $(":header") | 所有标题元素 <h1> - <h6> |
| [:animated](https://www.w3school.com.cn/jquery/selector_animated.asp) |  | 所有动画元素 |
| [:contains(*text*)](https://www.w3school.com.cn/jquery/selector_contains.asp) | $(":contains('W3School')") | 包含指定字符串的所有元素 |
| [:empty](https://www.w3school.com.cn/jquery/selector_empty.asp) | $(":empty") | 无子（元素）节点的所有元素 |
| :hidden | $("p:hidden") | 所有隐藏的 <p> 元素 |
| [:visible](https://www.w3school.com.cn/jquery/selector_visible.asp) | $("table:visible") | 所有可见的表格 |
| s1,s2,s3 | $("th,td,.intro") | 所有带有匹配选择的元素 |
| [[*attribute*]](https://www.w3school.com.cn/jquery/selector_attribute.asp) | $("[href]") | 所有带有 href 属性的元素 |
| [[*attribute*=*value*]](https://www.w3school.com.cn/jquery/selector_attribute_equal_value.asp) | $("[href='#']") | 所有 href 属性的值等于 "#" 的元素 |
| [[*attribute*!=*value*]](https://www.w3school.com.cn/jquery/selector_attribute_notequal_value.asp) | $("[href!='#']") | 所有 href 属性的值不等于 "#" 的元素 |
| [[*attribute*$=*value*]](https://www.w3school.com.cn/jquery/selector_attribute_end_value.asp) | $("[href$='.jpg']") | 所有 href 属性的值包含以 ".jpg" 结尾的元素 |
| [:input](https://www.w3school.com.cn/jquery/selector_input.asp) | $(":input") | 所有 <input> 元素 |
| [:text](https://www.w3school.com.cn/jquery/selector_input_text.asp) | $(":text") | 所有 type="text" 的 <input> 元素 |
| [:password](https://www.w3school.com.cn/jquery/selector_input_password.asp) | $(":password") | 所有 type="password" 的 <input> 元素 |
| [:radio](https://www.w3school.com.cn/jquery/selector_input_radio.asp) | $(":radio") | 所有 type="radio" 的 <input> 元素 |
| [:checkbox](https://www.w3school.com.cn/jquery/selector_input_checkbox.asp) | $(":checkbox") | 所有 type="checkbox" 的 <input> 元素 |
| [:submit](https://www.w3school.com.cn/jquery/selector_input_submit.asp) | $(":submit") | 所有 type="submit" 的 <input> 元素 |
| [:reset](https://www.w3school.com.cn/jquery/selector_input_reset.asp) | $(":reset") | 所有 type="reset" 的 <input> 元素 |
| [:button](https://www.w3school.com.cn/jquery/selector_input_button.asp) | $(":button") | 所有 type="button" 的 <input> 元素 |
| [:image](https://www.w3school.com.cn/jquery/selector_input_image.asp) | $(":image") | 所有 type="image" 的 <input> 元素 |
| [:file](https://www.w3school.com.cn/jquery/selector_input_file.asp) | $(":file") | 所有 type="file" 的 <input> 元素 |
| [:enabled](https://www.w3school.com.cn/jquery/selector_input_enabled.asp) | $(":enabled") | 所有激活的 input 元素 |
| [:disabled](https://www.w3school.com.cn/jquery/selector_input_disabled.asp) | $(":disabled") | 所有禁用的 input 元素 |
| [:selected](https://www.w3school.com.cn/jquery/selector_input_selected.asp) | $(":selected") | 所有被选取的 input 元素 |
| [:checked](https://www.w3school.com.cn/jquery/selector_input_checked.asp) | $(":checked") | 所有被选中的 input 元素 |
| $(this) |  | 选取当前 HTML 元素 |

### 3、jQuery 事件

click()

dblclick() //双击事件

mouseenter() //当鼠标指针穿过元素时，会发生 mouseenter 事件。

mouseleave() //当鼠标指针离开元素时，会发生 mouseleave 事件。

mousedown() //按下鼠标按键时，会发生mousedown 事件。

mouseup() //当在元素上松开鼠标按钮时，会发生 mouseup 事件。

hover() //hover()方法用于模拟光标悬停事件。

focus() //当元素获得焦点时，发生 focus 事件。

blur() //当元素失去焦点时，发生 blur 事件

### 4、jQuery 效果- 隐藏和显示

### 5、jQuery 效果 - 淡入淡出

jQuery fadeIn() 方法

$("#div1").fadeIn();

$("#div2").fadeIn("slow");

$("#div3").fadeIn(3000);

jQuery fadeOut() 方法

$("#div1").fadeOut();

$("#div2").fadeOut("slow");

$("#div3").fadeOut(3000);

jQuery fadeToggle() 方法

$("#div1").fadeToggle();

$("#div2").fadeToggle("slow");

$("#div3").fadeToggle(3000);

jQuery fadeTo() 方法

$("#div1").fadeTo("slow",0.15); //允许渐变为给定的不透明度（值介于 0 与 1 之间）

$("#div2").fadeTo("slow",0.4);

$("#div3").fadeTo("slow",0.7);

### 6、jQuery 效果 - 滑动

jQuery slideDown() 方法

$(selector).slideDown(speed,callback); //可选的 callback 参数是滑动完成后所执行的函数名称

jQuery slideUp() 方法

$(selector).slideUp(speed,callback); //可选的 callback 参数是滑动完成后所执行的函数名称

jQuery slideToggle() 方法

$(selector).slideToggle(speed,callback);

### 7、jQuery 效果- 动画

jQuery 动画 - animate() 方法

$(selector).animate({params},speed,callback);

$("div").animate({left:'250px'});//右移250像素

### 8、jQuery 停止动画

jQuery stop() 方法

$(selector).stop(stopAll,goToEnd);

可选的 stopAll 参数规定是否应该清除动画队列。默认是 false，即仅停止活动的动画，允许任何排入队列的动画向后执行。

可选的 goToEnd 参数规定是否立即完成当前动画。默认是 false。

因此，默认地，stop() 会清除在被选元素上指定的当前动画。

### 9、jQuery - 链(Chaining)

$("#p1").css("color","red").slideUp(2000).slideDown(2000);//上2秒下2秒

### 10、获取内容和属性

获得内容 - text()、html() 以及 val()

text() //设置或返回所选元素的文本内容

html() // 设置或返回所选元素的内容（包括 HTML 标记）

val() // 设置或返回表单字段的值(value)

获取属性 - attr()

alert($("#runoob").attr("href"));// attr() 方法用于获取属性值

### 11、设置内容和属性

### 12、添加元素

### 13、删除元素

### 14、获取并设置 CSS 类

### 15、方法

### 16、尺寸

### 17、jQuery 遍历

jQuery 提供了多种遍历 DOM 的方法。遍历方法中最大的种类是树遍历（tree-traversal）

### 18、祖先

jQuery parent() 方法

$("span").parent();//返回每个 <span> 元素的的直接父元素

jQuery parents() 方法

$("span").parents();//返回所有 <span> 元素的所有祖先

$("span").parents("ul"); //返回所有 <span> 元素的所有祖先，并且它是 <ul> 元素

jQuery parentsUntil() 方法

$("span").parentsUntil("div");//返回介于 <span> 与 <div> 元素之间的所有祖先元素

### 19、后代

jQuery children() 方法

$("div").children();//返回每个 <div> 元素的所有直接子元素

$("div").children("p.1");//返回类名为 "1" 的所有 <p> 元素，并且它们是 <div> 的直接子元素

jQuery find() 方法

$("div").find("span");//返回属于 <div> 后代的所有 <span> 元素

$("div").find("\*");//返回 <div> 的所有后代

### 20、同胞(siblings)

jQuery siblings() 方法

$("h2").siblings();//返回 <h2> 的所有同胞元素

$("h2").siblings("p");//返回属于 <h2> 的同胞元素的所有 <p> 元素：

jQuery next() 方法

$("h2").next();//返回 <h2> 的下一个同胞元素

jQuery nextAll() 方法

$("h2").nextAll();//返回 <h2> 的所有跟随的同胞元素

jQuery nextUntil() 方法

$("h2").nextUntil("h6");//返回介于 <h2> 与 <h6> 元素之间的所有同胞元素

prev(), prevAll() 以及 prevUntil() 方法的工作方式与上面的方法类似，只不过方向相反而已：

21、过滤

jQuery first() 方法

$("div p").first();//选取首个 <div> 元素内部的第一个 <p> 元素

jQuery last() 方法

$("div p").last();//选择最后一个 <div> 元素中的最后一个 <p> 元素

jQuery eq() 方法

$("p").eq(1);//索引号从 0 开始，因此首个元素的索引号是 0 而不是 1。下面的例子选取第二个 <p> 元素

jQuery filter() 方法

$("p").filter(".url");//返回带有类名 "url" 的所有 <p> 元素

jQuery not() 方法

$("p").not(".url");//返回不带有类名 "url" 的所有 <p> 元素

### 22、AJAX load() 方法

$(selector).load(URL,data,callback);

$("#div1").load("demo\_test.txt");

$("#div1").load("demo\_test.txt #p1");//下面的例子把 "demo\_test.txt" 文件中 id="p1" 的元素的内容，加载到指定的 <div> 元素中

$("button").click(function(){

$("#div1").load("demo\_test.txt",function(responseTxt,statusTxt,xhr){

if(statusTxt=="success")

alert("外部内容加载成功!");

if(statusTxt=="error")

alert("Error: "+xhr.status+": "+xhr.statusText);

});

});

responseTxt - 包含调用成功时的结果内容

statusTXT - 包含调用的状态

xhr - 包含 XMLHttpRequest 对象

### 23、AJAX get() 和 post() 方法

get

$("button").click(function(){

$.get("demo\_test.php",function(data,status){

alert("数据: " + data + "\n状态: " + status);

});

});

post

$("button").click(function(){

$.post("/try/ajax/demo\_test\_post.php",

{

name:"菜鸟教程",

url:"http://www.runoob.com"

},

function(data,status){

alert("数据: \n" + data + "\n状态: " + status);

});

});

## 五、JSON

### 1、JSON 对象

访问对象值

var myObj, x;

myObj = { "name":"runoob", "alexa":10000, "site":null };

x = myObj.name;

x = myObj["name"];

for (x in myObj) {

document.getElementById("demo").innerHTML += x + "<br>"; //获取键

document.getElementById("demo").innerHTML += myObj[x] + "<br>"; //获取值

}

嵌套 JSON 对象

myObj = {

"name":"runoob",

"alexa":10000,

"sites": {

"site1":"www.runoob.com",

"site2":"m.runoob.com",

"site3":"c.runoob.com"

}

}

x = myObj.sites.site1;

// 或者

x = myObj.sites["site1"];

修改值

myObj.sites.site1 = "www.google.com";

myObj.sites["site1"] = "www.google.com";

删除对象属性

delete myObj.sites.site1;

delete myObj.sites["site1"]

### 2、JSON 数组

JSON 数组在中括号中书写。

myObj = {

"name":"网站",

"num":3,

"sites":[ "Google", "Runoob", "Taobao" ]

}

x = myObj.sites[0];

嵌套 JSON 对象中的数组

var myObj, i, j, x = "";

myObj = {

"name":"网站",

"num":3,

"sites": [

{ "name":"Google", "info":[ "Android", "Google 搜索", "Google 翻译" ] },

{ "name":"Runoob", "info":[ "菜鸟教程", "菜鸟工具", "菜鸟微信" ] },

{ "name":"Taobao", "info":[ "淘宝", "网购" ] }

]

}

for (i in myObj.sites) {

x += "<h1>" + myObj.sites[i].name + "</h1>";

for (j in myObj.sites[i].info) {

x += myObj.sites[i].info[j] + "<br>";

}

}

### 3、JSON.parse()

JSON 通常用于与服务端交换数据。

在接收服务器数据时一般是字符串。

我们可以使用 JSON.parse() 方法将数据转换为 JavaScript 对象。

var obj = JSON.parse('{ "name":"runoob", "alexa":10000, "site":"www.runoob.com" }');

document.getElementById("demo").innerHTML = obj.name + "：" + obj.site;

### 4、JSON.stringify()

var obj = { "name":"runoob", "alexa":10000, "site":"www.runoob.com"};

var myJSON = JSON.stringify(obj);//我们可以使用 JSON.stringify() 方法将 JavaScript 对象转换为字符串。

### 5、JSONP 教程

Jsonp(JSON with Padding) 是 json 的一种"使用模式"，可以让网页从别的域名（网站）那获取资料，即跨域读取数据。

## 六、AJax

### 1、创建 XMLHttpRequest 对象

var xmlhttp;

if (window.XMLHttpRequest)

{

// IE7+, Firefox, Chrome, Opera, Safari 浏览器执行代码

xmlhttp=new XMLHttpRequest();

}

else

{

// IE6, IE5 浏览器执行代码

xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");

}

### 2、向服务器发送请求

xmlhttp.open("GET","ajax\_info.txt",true);

xmlhttp.send();

与 POST 相比，GET 更简单也更快，并且在大部分情况下都能用。

然而，在以下情况中，请使用 POST 请求：

无法使用缓存文件（更新服务器上的文件或数据库）

向服务器发送大量数据（POST 没有数据量限制）

发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠

GET 请求

为了避免获得的是缓存，请向 URL 添加一个唯一的 ID：

xmlhttp.open("GET","/try/ajax/demo\_get.php?t=" + Math.random(),true);

xmlhttp.send();

传递参数：

xmlhttp.open("GET","/try/ajax/demo\_get2.php?fname=Henry&lname=Ford",true);

xmlhttp.send();

POST 请求

如果需要像 HTML 表单那样 POST 数据，请使用 setRequestHeader() 来添加 HTTP 头。然后在 send() 方法中规定您希望发送

xmlhttp.open("POST","/try/ajax/demo\_post2.php",true);

xmlhttp.setRequestHeader("Content-type","application/x-www-form-urlencoded");

xmlhttp.send("fname=Henry&lname=Ford");

异步 - True 或 False

当使用 async=true 时

请规定在响应处于 onreadystatechange 事件中的就绪状态时执行的函数：

xmlhttp.onreadystatechange=function()

{

if (xmlhttp.readyState==4 && xmlhttp.status==200)

{

document.getElementById("myDiv").innerHTML=xmlhttp.responseText;

}

}

xmlhttp.open("GET","/try/ajax/ajax\_info.txt",true);

xmlhttp.send();

使用 async=false 时

请不要编写 onreadystatechange 函数 - 把代码放到 send() 语句后面即可：

xmlhttp.open("GET","/try/ajax/ajax\_info.txt",false);

xmlhttp.send();

document.getElementById("myDiv").innerHTML=xmlhttp.responseText;

### 3、AJAX - 服务器 响应

强求数据为xml

xmlhttp.onreadystatechange=function()

{

if (xmlhttp.readyState==4 && xmlhttp.status==200)

{

xmlDoc=xmlhttp.responseXML;

txt="";

x=xmlDoc.getElementsByTagName("ARTIST");

for (i=0;i<x.length;i++)

{

txt=txt + x[i].childNodes[0].nodeValue + "<br>";

}

document.getElementById("myDiv").innerHTML=txt;

}

}

xmlhttp.open("GET","cd\_catalog.xml",true);

xmlhttp.send();

### 4、AJAX - onreadystatechange 事件

onreadystatechange：

存储函数（或函数名），每当 readyState 属性改变时，就会调用该函数。

readyState：

0: 请求未初始化

1: 服务器连接已建立

2: 请求已接收

3: 请求处理中

4: 请求已完成，且响应已就绪

status：

200: "OK"

404: 未找到页面

# 十九、人工智能

## 三、python基础

### 一、第一课软件安装

#### Python简介

##### 起源

Python的作者是著名的“龟叔”Guido van Rossum，他希望有一种语言，这种语言能够像C语言那样，能够全面调用计算机的功能接口，又可以像shell那样，可以轻松的编程。龟叔从ABC语言看到希望，ABC语言是由荷兰的数学和计算机研究所开发的。龟叔也参与到ABC语言的开发。由于一系列原因ABC并没有快速传播使用。因此，龟叔开始写Python语言。

##### 优点

简单易学、免费开源、可扩展性可嵌入性、丰富的库。

##### 缺点

运行速度慢、中文资料匮乏、代码不能加密。

#### Python安装

<https://www.python.org/downloads/>下载安装。

#### 安装Anaconda

##### Anaconda是什么

Anaconda是一个用于**科学计算**的Python发行版，支持 Linux, Mac, Windows系统，提供了包管理与环境管理的功能，可以很方便地解决多版本python并存、切换以及各种第三方包安装问题。Anaconda利用工具/命令**conda**来进行package和environment的管理，并且已经包含了Python和相关的配套工具

##### Anaconda 的优点

省时省心、分析利器。

##### Anaconda下载安装

<https://www.anaconda.com/download/>下载安装

### 二、第二课

# 二十、第一本Docker书

## 简介

Docker是一个能把应用程序自动部署到容器的开源引擎。

# 九十八、Demo搭建

## 创建虚拟机

精髓：先装系统后装增强工具，否则进入系统弹出桌面的增强光盘重新安装增强工具。设备-》安装增强功能

xShell连接服务器

右键虚拟机->设置->网络，连接方式选择NAT

上图界面选择端口转发，添加一项，客户端也就是虚拟机填写ssh默认的22，填写一个你想转发给主机的端口号，比如9023。NAT网络地址转换，客户端虚拟机的所有网络包都会由主机也就是你正在用的机器转发，所以这里设定端口转发后，你连接本机的9023端口就相当于连接客户机的22端口。

ssh root@127.0.0.1 9023

xFtp连接服务器

开启window是的ftp服务

控制面板 - 程序 - 启用和关闭Windows功能-》Internet information Service->开启ftp服务

linux可能会禁用root用户登陆ftp，要确认

rpm -qa | grep vsftpd 查看服务器是否安装xFtpd服务

yum -y install vsftpd 安装xFtpd服务

chkconfig vsftpd on 设置开机启动xFtpd服务

127.0.0.1 9023ip 端口

本机ping虚拟机：管理-》全局设定-》网络-》host-only添加适配器

## 创建用户

### Root用户登录

用户名/密码：root/123456。

### 2.创建用户组

groupadd demo --创建用户组

**3.创建用户**

useradd -d /home/demo -m demo -g demo --创建用户及其主目录，及所属就用户组

passwd demo --为用户设置密码

## 安装jdk

### 1.卸载openJdk

1.查看java

rpm -qa | grep java

tzdata-java-2016j-1.el6.noarch

java-1.7.0-openjdk-1.7.0.131-2.6.9.0.el6\_8.x86\_64

java-1.6.0-openjdk-1.6.0.41-1.13.13.1.el6\_8.x86\_64

2.卸载jdk

rpm -e --nodeps java-1.7.0-openjdk-1.7.0.131-2.6.9.0.el6\_8.x86\_64

rpm -e --nodeps java-1.6.0-openjdk-1.6.0.41-1.13.13.1.el6\_8.x86\_64

### 2.安装JDK方式一

1.下载jdk

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html //jdk-8u161-linux-x64.rpm

2.新建/opt/tools文件夹，将文件拷入

3.安装jdk

cd /opt/tools/

rpm -ivh jdk-8u161-linux-x64.rpm

java -version

### 3.安装JDK方式二

mkdir JDK --创建目录

ftp上传.tar.gz包

tar -xzvf jdk-8u191-linux-x64.tar.gz

cd /

-----------------------------chmod 777 -R / --为所有用户赋权限（权限太大，有待优化）

vi .bash\_profile --创建文件

PATH=$PATH:$HOME/bin --需要root用户权限或者root用户给demo用户赋权

export JAVA\_HOME=/home/demo/JDK/jdk1.8.0\_191

PATH=$JAVA\_HOME/bin:$PATH:$HOME/bin

export PATH

source .bash\_profile --使环境变量生效



## 四.安装tomcat

1.下载tomcat

http://tomcat.apache.org/download-80.cgi // apache-tomcat-9.0.4.tar.gz

2.新建/opt/tools文件夹，将文件拷入

3.安装tomcat

cd /opt/tools/

tar xzvf apache-tomcat-9.0.4.tar.gz //解压

mv apache-tomcat-9.0.4 /opt/tomcat/

/opt/tomcat/bin/startup.sh

iptables -I INPUT -p tcp -m state --state NEW -m tcp --dport 8080 -jACCEPT

iptables -L -n

service iptables save

-------------------------------------------------------------------------------------------------------

mkdir tomcat --创建目录

ftp上传安装包

tar -xzvf apache-tomcat-8.5.35.gz --解压

cd apache-tomcat-8.5.35/bin

vi catalina.sh

export JAVA\_HOME='/home/demo/JDK/jdk1.8.0\_191'



chmod +x catalina.sh -赋执行权限

未写

配置tomcat参数

修改日志级别

隐藏版本信息（替换catalina.jar）

定时清理日志

## 五.安装eclipse

1.下载eclipse

wget http://mirrors.neusoft.edu.cn/eclipse/technology/epp/downloads/release/neon/1a/eclipse-java-neon-1a-linux-gtk-x86\_64.tar.gz

2.解压 tar 包到目录 /opt

sudo tar -xzvf eclipse-java-neon-1a-linux-gtk-x86\_64.tar.gz -C /opt

3.建立符号连接

ln -s /opt/eclipse/eclipse /usr/bin/eclipse

4.创建桌面启动器

右击桌面空白处-》创建启动器 /usr/bin/eclipse

## 六.安装oracle

创建oracle用户及用户组（root执行）

groupadd oinstall

groupadd dba

useradd -g oinstall -G dba oracle

命令核查oracle用户是否配置完善

id oracle

xftp上传两个zip安装包

解压安装包，解压完成后会生成一个database文件夹

unzip linux\_11gR2\_database\_1of2.zip

unzip linux\_11gR2\_database\_2of2.zip

查看内存

grep MemTotal /proc/meminfo

查看交换分区大小

grep SwapTotal /proc/meminfo

查看硬盘空间

df -h

依赖包要求

binutils-2.17.50.0.6

compat-libstdc++-33-3.2.3

elfutils-libelf-0.125

elfutils-libelf-devel-0.125

elfutils-libelf-devel-static-0.125

gcc-4.1.2

gcc-c++-4.1.2

glibc-2.5-24

glibc-common-2.5

glibc-devel-2.5

glibc-headers-2.5

kernel-headers-2.6.18

ksh-20060214

libaio-0.3.106

libaio-devel-0.3.106

libgcc-4.1.2

libgomp-4.1.2

libstdc++-4.1.2

libstdc++-devel-4.1.2

make-3.81

sysstat-7.0.2

unixODBC-2.2.11

unixODBC-devel-2.2.11

你可以使用以下命令查看上面这些软件包的版本是否大于等于上面的要求：

rpm -q binutils compat-libstdc++ elfutils-libelf elfutils-libelf-devel elfutils-libelf-devel-static gcc gcc-c++ glibc glibc-common glibc-devel glibc-headers kernel-headers ksh libaio libaio-devel libgcc libgomp libstdc++ libstdc++-devel make sysstat unixODBC unixODBC-devel

如果出现有未安装的软件包，使用yum把这些软件包都更新一遍：

yum install binutils compat-libstdc++ elfutils-libelf elfutils-libelf-devel elfutils-libelf-devel-static gcc gcc-c++ glibc glibc-common glibc-devel glibc-headers kernel-headers ksh libaio libaio-devel libgcc libgomp libstdc++ libstdc++-devel make sysstat unixODBC unixODBC-devel

配置内核参数，在/etc/sysctl.conf文件中，添加以下内核参数：

fs.aio-max-nr = 1048576

fs.file-max = 6815744

kernel.shmall = 2097152

kernel.shmmax = 536870912

kernel.shmmni = 4096

kernel.sem = 250 32000 100 128

net.ipv4.ip\_local\_port\_range = 9000 65500

net.core.rmem\_default = 262144

net.core.rmem\_max = 4194304

net.core.wmem\_default = 262144

net.core.wmem\_max = 1048586

为使上述配置生效而不重启系统，执行如下命令

/sbin/sysctl -p

为了提高Oracle软件性能，需要为Oracle用户添加以下shell配置：

Shell Limit 在limits.conf中的项 硬限制

打开文件描述符的最大数量 nofile 65536

单个用户可用的最大进程数 nproc 16384

进程堆栈段的最大大小 stack 10240

在/etc/security/limits.conf文件，添加以下参数：

oracle soft nproc 2047

oracle hard nproc 16384

oracle soft nofile 1024

oracle hard nofile 65536

在/etc/pam.d/login文件中添加一行：

session required pam\_limits.so

在/etc/profile文件添加以下脚本：

if [ $USER = "oracle" ]; then

if [ $SHELL = "/bin/ksh" ]; then

ulimit -p 16384

ulimit -n 65536

else

ulimit -u 16384 -n 65536

fi

fi

创建并配置环境变量

cd /home/oracle

mkdir oracleInstall

cd oracleInstall

mkdir app

cd ../

chown -R oracle:oinstall oracleInstall

chmod -R 775 oracleInstall

配置oracle环境变量

切换为oracle用户登录

su - oracle

vi .bash\_profile

export ORACLE\_BASE=/home/oracle/oracleInstall

export ORACLE\_HOME=/home/oracle/oracleInstall/app

export ORACLE\_SID=orcl

export PATH=$ORACLE\_HOME/bin:$PATH

开始正式安装

su - root

yum install -y ld-linux.so.2

yum install libXt\*

vncserver

yum install -y xdpyinfo

export DISPLAY=:1.0

xhost +

mount -t tmpfs -o size=10000M /tmp

su - oracle

export DISPLAY=:1.0

xhost +

su - oracle

cd /home/oracle/database

一定要使用oracle用户登录图形界面，否则运行runInstaller会报错显示错误(因为后面使用图形化界面安装的，需要权限去运行图形界面程序)。./runInstaller

## 七．安装solr集群

### 1.上传文件

root用户上传文件到/root/solrCloud

tomcat

solr

zookeeper

### 2.解压zookeeper

tar -zxvf zookeeper-3.4.8.tar.gz

复制zookeeper三份

cp -r zookeeper-3.4.8 zookeeper-3.4.8-01

cp -r zookeeper-3.4.8 zookeeper-3.4.8-02

cp -r zookeeper-3.4.8 zookeeper-3.4.8-03

rm -rf zookeeper-3.4.8

cd zookeeper-3.4.8-01

mkdir data

cd data

touch myid

echo 1 >> myid

cd zookeeper-3.4.8-02

mkdir data

cd data

touch myid

echo 2 >> myid

cd zookeeper-3.4.8-03

mkdir data

cd data

touch myid

echo 3 >> myid

把conf目录下的zoo\_sample.cfg文件改名为zoo.cfg,修改zoo.cfg配置文件

cd zookeeper-3.4.8-01/conf

vi zoo.cfg

mv zoo\_sample.cfg zoo.cfg

dataDir=/root/solrCloud/zookeeper-3.4.8-01/data

clientPort=2181

server.1=10.0.2.15:2881:3881 #内部通讯端口号：投票选举端口号

server.2=10.0.2.15:2882:3882

server.3=10.0.2.15:2883:3883

cd zookeeper-3.4.8-02/conf

mv zoo\_sample.cfg zoo.cfg

dataDir=/root/solrCloud/zookeeper-3.4.8-02/data

clientPort=2182

server.1=10.0.2.15:2881:3881 #内部通讯端口号：投票选举端口号

server.2=10.0.2.15:2882:3882

server.3=10.0.2.15:2883:3883

cd zookeeper-3.4.8-03/conf

mv zoo\_sample.cfg zoo.cfg

dataDir=/root/solrCloud/zookeeper-3.4.8-03/data

clientPort=2183

server.1=10.0.2.15:2881:3881 #内部通讯端口号：投票选举端口号

server.2=10.0.2.15:2882:3882

server.3=10.0.2.15:2883:3883

### 3.编写启动脚本

cd /root/solrCloud

vi startAll.sh

cd /root/solrCloud/zookeeper-3.4.8-01/bin/

./zkServer.sh start

cd /root/solrCloud/zookeeper-3.4.8-02/bin/

./zkServer.sh start

cd /root/solrCloud/zookeeper-3.4.8-03/bin/

./zkServer.sh start

赋权限

chmod u+x startAll.sh

### 4.启动zookeeper

./startAll.sh

### 5.编写查看状态脚本

vi queryAllStatus.sh

cd /root/solrCloud/zookeeper-3.4.8-01/bin/

./zkServer.sh status

cd /root/solrCloud/zookeeper-3.4.8-02/bin/

./zkServer.sh status

cd /root/solrCloud/zookeeper-3.4.8-03/bin/

./zkServer.sh status

赋权限

chmod u+x queryAllStatus.sh

### 7.查看zookeeper启动及选举状态

./queryAllStatus.sh

### 8.solr单机搭建

cd /root

mkdir solr

上传解压tomcat

tar -zxvf apache-tomcat-8.5.35.gz

cd /root/solr/apache-tomcat-8.5.35/conf

vi server.xml

修改端口号

8505

8580

8509

上传solr-7.1.0.zip

unzip solr-7.1.0.zip

cd /root/solr/apache-tomcat-8.5.35/webapps

cp -r /root/solr/solr-7.1.0/server/solr-webapp/webapp /root/solr/apache-tomcat-8.5.35/webapps/solr

cd /root/solr/apache-tomcat-8.5.35/webapps/solr/webapp

cp \* -r ../

cd ../

rm -rf webapp/

cd /root/solr/solr-7.1.0/server/lib/ext

cp \* /root/solr/apache-tomcat-8.5.35/webapps/solr/WEB-INF/lib

/root/solr/solr-7.1.0/server/lib

cp metrics\*.jar /root/solr/apache-tomcat-8.5.35/webapps/solr/WEB-INF/lib

cd /root/solr/apache-tomcat-8.5.35/webapps/solr/WEB-INF

mkdir classes

cp /root/solr/solr-7.1.0/server/resources/log4j.properties /root/solr/apache-tomcat-8.5.35/webapps/solr/WEB-INF/classes

cp -r /root/solr/solr-7.1.0/server/solr /root/solr/solrhome

/root/solr/apache-tomcat-8.5.35/webapps/solr/WEB-INF

vi web.xml

打开注释，修改

<env-entry>

<env-entry-name>solr/home</env-entry-name>

<env-entry-value>/put/your/solr/home/here</env-entry-value>

<env-entry-type>java.lang.String</env-entry-type>

</env-entry>

注释 <security-constraint>段

启动tomcat

浏览器打开 localhost:8580/solr/index.html

### 9.solr集群搭建

解压tomcat

tar -zxvf apache-tomcat-8.5.35.gz

mv apache-tomcat-8.5.35 apache-tomcat-8.5.35-01

cp -r apache-tomcat-8.5.35-01 apache-tomcat-8.5.35-02

cp -r apache-tomcat-8.5.35-01 apache-tomcat-8.5.35-03

cp -r apache-tomcat-8.5.35-01 apache-tomcat-8.5.35-04

cd /root/solrCloud/apache-tomcat-8.5.35-01/conf

vi server.xml

修改端口号

8105

8180

8109

cd /root/solrCloud/apache-tomcat-8.5.35-02/conf

vi server.xml

修改端口号

8205

8280

8209

cd /root/solrCloud/apache-tomcat-8.5.35-03/conf

vi server.xml

修改端口号

8305

8380

8309

cd /root/solrCloud/apache-tomcat-8.5.35-04/conf

vi server.xml

修改端口号

8405

8480

8409

把单机版的solr工程复制到集群中的tomcat中

cp -r /root/solr/apache-tomcat-8.5.35/webapps/solr/ /root/solrCloud/apache-tomcat-8.5.35-01/webapps/

cp -r /root/solr/apache-tomcat-8.5.35/webapps/solr/ /root/solrCloud/apache-tomcat-8.5.35-02/webapps/

cp -r /root/solr/apache-tomcat-8.5.35/webapps/solr/ /root/solrCloud/apache-tomcat-8.5.35-03/webapps/

cp -r /root/solr/apache-tomcat-8.5.35/webapps/solr/ /root/solrCloud/apache-tomcat-8.5.35-04/webapps/

cp -r /root/solr/solrhome/ /root/solrCloud/solrhome01

cp -r /root/solr/solrhome/ /root/solrCloud/solrhome02

cp -r /root/solr/solrhome/ /root/solrCloud/solrhome03

cp -r /root/solr/solrhome/ /root/solrCloud/solrhome04

vi solrhome01/solr.xml

10.0.2.15

8180

vi solrhome02/solr.xml

10.0.2.15

8280

vi solrhome03/solr.xml

10.0.2.15

8380

vi solrhome04/solr.xml

10.0.2.15

8480

要修改solr服务的web.xml文件。把solrhome关联起

vi /root/solrCloud/apache-tomcat-8.5.35-01/webapps/solr/WEB-INF/web.xml

/root/solrCloud/solrhome01

vi /root/solrCloud/apache-tomcat-8.5.35-02/webapps/solr/WEB-INF/web.xml

/root/solrCloud/solrhome02

vi /root/solrCloud/apache-tomcat-8.5.35-03/webapps/solr/WEB-INF/web.xml

/root/solrCloud/solrhome03

vi /root/solrCloud/apache-tomcat-8.5.35-04/webapps/solr/WEB-INF/web.xml

/root/solrCloud/solrhome04

让zookeeper统一管理配置文件。需要把/conf目录上传到zookeeper，保证zookeeper集群启动

/root/solr/solr-7.1.0/server/scripts/cloud-scripts

./zkcli.sh -zkhost 10.0.2.15:2181,10.0.2.15:2182,10.0.2.15:2183 -cmd upconfig -confdir /root/solrCloud/solrhome01/configsets/sample\_techproducts\_configs/conf -confname myconf

修改tomcat/bin目录下的catalina.sh 文件，关联solr和zookeeper

vi /root/solrCloud/apache-tomcat-8.5.35-01/bin/catalina.sh

在JAVA\_OPTS="$JAVA\_OPTS -Dorg.apache.catalina.security.SecurityListener.UMASK=`umask`"下加

JAVA\_OPTS="-DzkHost=10.0.2.15:2181,10.0.2.15:2182,10.0.2.15:2183"

vi /root/solrCloud/apache-tomcat-8.5.35-02/bin/catalina.sh

JAVA\_OPTS="-DzkHost=10.0.2.15:2181,10.0.2.15:2182,10.0.2.15:2183"

vi /root/solrCloud/apache-tomcat-8.5.35-03/bin/catalina.sh

JAVA\_OPTS="-DzkHost=10.0.2.15:2181,10.0.2.15:2182,10.0.2.15:2183"

vi /root/solrCloud/apache-tomcat-8.5.35-04/bin/catalina.sh

JAVA\_OPTS="-DzkHost=10.0.2.15:2181,10.0.2.15:2182,10.0.2.15:2183"

启动停止tomcat脚本

vi startTomcatAll.sh

/root/solrCloud/apache-tomcat-8.5.35-01/bin/startup.sh

/root/solrCloud/apache-tomcat-8.5.35-02/bin/startup.sh

/root/solrCloud/apache-tomcat-8.5.35-03/bin/startup.sh

/root/solrCloud/apache-tomcat-8.5.35-04/bin/startup.sh

vi shutTomcatAll.sh

/root/solrCloud/apache-tomcat-8.5.35-01/bin/shutdown.sh

/root/solrCloud/apache-tomcat-8.5.35-02/bin/shutdown.sh

/root/solrCloud/apache-tomcat-8.5.35-03/bin/shutdown.sh

/root/solrCloud/apache-tomcat-8.5.35-04/bin/shutdown.sh

赋权限

chmod u+x startTomcatAll.sh

chmod u+x shutTomcatAll.sh

vi /etc/hosts

在 127.0.0.1 和::1 后面添加 你的新主机名

## 八．Redis集群搭建

这里安装的是redis源码包，所以需要gcc环境支持

root用户登陆

yum -y install gcc automake autoconf libtool make 安装make命令

　　yum install gcc-c++ 安装gcc

tar -zxvf redis-3.0.0.tar.gz 解压

mkdir redisInstall 创建安装目录

cd redisInstall

mkdir etc

cp /home/demo/redis/redis-3.0.0/redis.conf ./etc

cd etc

vi redis.conf

将daemonize no 改成daemonize yes

./redis-server 启动redis

搭建集群

　yum install ruby 安装ruby  
　　　　yum install rubygems

升级ruby

sudo yum install curl

gpg2 --keyserver hkp://keys.gnupg.net --recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3

curl -L get.rvm.io | bash -s stable

下载上传ruby压缩包

tar -zxvf ruby-2.6.0.tar.gz

cd ruby-2-6.0

./configure --prefix=/home/demo/ruby/rubyInstall -prefix是将ruby安装到指定目录，

make && make install 编译安装

vi /etc/profile

source /etc/profile

ruby -v 查看验证安装

gem install redis 安装ruby和redis的接口程序

# 九十九.杂项

### MUI APP配置地图

1.申请秘钥

2.android清单文件配置

3.manifest.json 模块权限配置

SDK配置写入appkey

代码视图：

permissions下：

"Maps": {

"description": "管理地图插件"

},

plus下：

"maps": {

"baidu": {

"appkey\_ios": "Qe8Nu6zV1kiXBNtGaRi0mG4fdPyREFnb",

"appkey\_android": "jIBWFldRGoyeD1fuX5QyvVi9OUhxligI",

"description": "百度地图"

}

}

4.assets/data/dcloud\_properties.xml

<!--<feature name="Maps" value="io.dcloud.js.map.amap.JsMapPluginImpl"/>-->高德

<feature name="Maps" value="io.dcloud.js.map.JsMapPluginImpl"/> 百度（壳子中将百度注释掉）

<service name="Maps" value="io.dcloud.js.map.MapInitImpl"/>

5.libs:

BaiduLBS\_Android\_base.jar,BaiduLBS\_Android\_geolocation\_v7.1.jar,BaiduLBS\_Android\_v4.3.1.jar,map-baidu.jar

armeabi:

libBaiduMapSDK\_base\_v4\_3\_1.so,libBaiduMapSDK\_map\_v4\_3\_1.so, liblocSDK7a.so

删除没用的jar包，因为方法总数超出android上限（DEX）

### Jsp转为html

客户第一次请求jsp页面时，jsp引擎会将jsp文件中的脚本代码(HTML)和代码片段(java代码)全部转换为java代码，转换过程非常直观：

对于HTML文本只需要用简单的out.println方法包裹，对于java脚本只做保留或简单的处理。本机jsp编译的class目录 eclipse\workspace\.metadata \.plugins\org.eclipse.wst.server.core\tmp0\work\Catalina\localhost\linux jsp 编译的class目录 tomcat\work

本机java编译的class目录 eclipse\workspace\.metadata \.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps

linux java编译的class目录 tomcat\webapps

编辑tomcat/conf/server.xml

<Host appBase="webapps" autoDeploy="true" name="localhost" unpackWARs="true">

<Context path="/emall" docBase="emall" debug="0" source="org.eclipse.jst.jee.server:emall" reloadable="true"></Context>

</Host>

path:指定访问该Web应用的URL入口。

docBase:指定Web应用的文件路径，可以给定绝对路径，也可以给定相对于<Host>的appBase属性的相对路径。

reloadable:如果这个属性设为true，tomcat服务器在运行状态下会监视在WEB-INF/classes和WEB-INF/lib目录下class文件的改动，

如果监测到有class文件被更新的，服务器会自动重新加载Web应用。

.classpath文件保存build path信息，更新svn可能导致项目不编译，需要重新设置build path

### 3、中国经纬度

最东端 东经135度2分30秒 黑龙江和乌苏里江交汇处 135.041667

最西端 东经73度40分 帕米尔高原乌兹别里山口（乌恰县） 73.666667

最南端 北纬3度52分 南沙群岛曾母暗沙 3.866667

最北端 北纬53度33分 漠河以北黑龙江主航道（漠河) 53.55

### 4、解决eclipse强关后启动失败

E:\asiainfo\workspace\.metadata\.plugins\org.eclipse.core.resources 删除文件