[保留两位小数（四舍五入） 2](#_Toc472683403)

[转义字符 3](#_Toc472683404)

[压缩文件 4](#_Toc472683405)

[SFTP使用 5](#_Toc472683406)

[方法： 7](#_Toc472683407)

[文件操作 7](#_Toc472683408)

[文件导出 7](#_Toc472683409)

[Xml导出 7](#_Toc472683410)

[word文档导出 8](#_Toc472683411)

[word文档词法解释 8](#_Toc472683412)

[File操作 8](#_Toc472683413)

[File类 8](#_Toc472683414)

[获得类路径 8](#_Toc472683415)

[Java流的分类 8](#_Toc472683416)

[按流向分: 8](#_Toc472683417)

[按数据传输单位分: 9](#_Toc472683418)

[概念 9](#_Toc472683419)

[输入输出流 9](#_Toc472683420)

[字节流的读写 10](#_Toc472683421)

[字符流的读写 10](#_Toc472683422)

[文件流的读写 11](#_Toc472683423)

[线程 11](#_Toc472683424)

[线程方法 11](#_Toc472683425)

[线程详解 11](#_Toc472683426)

[线程池 12](#_Toc472683427)

[java.util.concurrent类 Executors 12](#_Toc472683428)

[线程生命周期 14](#_Toc472683429)

[任务调度 14](#_Toc472683430)

[spring配置 14](#_Toc472683431)

[cron表达式 15](#_Toc472683432)

[", - \* /? L W C #"字符解释 15](#_Toc472683433)

[语法： 16](#_Toc472683434)

[例子： 16](#_Toc472683435)

[Calendar的使用 16](#_Toc472683436)

[时间比较 17](#_Toc472683437)

[Json与List互转 18](#_Toc472683438)

[WritableWorkbook导出Excle文件 18](#_Toc472683439)

[获取HttpServletRequest、HttpServletResponse对象 20](#_Toc472683440)

[配置文件的读取与PropertyPlaceholderConfigurer的用法 21](#_Toc472683441)

[方法一 21](#_Toc472683442)

[方法二 21](#_Toc472683443)

[方法三，分散配置 21](#_Toc472683444)

[方法四，自定义配置读取配置文件 22](#_Toc472683445)

[ConfigFactory.java文件 22](#_Toc472683446)

[配置xml文件，添加以下相应的属性文件代码 23](#_Toc472683447)

[Dao层利用模板（springMVC框架用） 23](#_Toc472683448)

[日期时间转换 24](#_Toc472683449)

[list里的map对象排序 24](#_Toc472683450)

[Spring使用 25](#_Toc472683451)

[spring获取bean 25](#_Toc472683452)

[Spring JdbcTemplate方法详解 25](#_Toc472683453)

[存储过程： 26](#_Toc472683454)

[Dom4J解析xml 26](#_Toc472683455)

# 保留两位小数（四舍五入）

float a = 0.121f;

float b = 0.127f;

方法一：String result = String.format("%.2f", b);

方法二：Math.round(b\*100) \* 0.01d

# 转义字符

\n 回车(\ )  
\t 水平制表符(\ )  
\b 空格(\)  
\r 换行(\ )  
\f 换页(\ )  
\' 单引号(\')  
\" 双引号(\")  
\\ 反斜杠(\\)  
\ddd 三位八进制

# 压缩文件

//压缩文件

String sourceDir="D:\\kkkk";//资源目录

String zipFilePath="D:\\hhh\\compressFile.zip";

File file = new File(sourceDir);

File zipFile = new File(zipFilePath);

// 创建写出流操作

OutputStream zipOutputStream = new FileOutputStream(zipFile);

BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(zipOutputStream);//缓冲区

ZipOutputStream zip = new ZipOutputStream(bufferedOutputStream);//包装zip输出流

String basePath = null;

File[] files = null;//文件列表

InputStream inputStream=null;

// 获取目录

if(file.isDirectory()) {

files = file.listFiles();//把目录下的文件放到文件列表里

for(File f:files){

inputStream = new FileInputStream(f);//输入流

String pathName;//路径名

byte[] buf = new byte[1024];//字节数组

int len = 0;

byte[] buffer = new byte[1024];

zip.putNextEntry(new ZipEntry(f.getName()));

while ((len = inputStream.read(buffer)) > 0) {

zip.write(buffer, 0, len);

}

}

}else {

files = new File[1];//只有一个文件

files[0] = file;

}

zip.closeEntry();

if(inputStream != null)inputStream.close();

zip.close();

# SFTP使用

要使用JSch，需要下载它的jar包，请从官网下载它：<http://www.jcraft.com/jsch/>

public static final String SFTP\_REQ\_HOST = "10.18.13.168";

public static final int SFTP\_REQ\_PORT = 22;

public static final String SFTP\_REQ\_USERNAME = "yhzh";

public static final String SFTP\_REQ\_PASSWORD = "123456";// SFTP会话

public static Session sftpSession;

public static ChannelSftp sftpChannel;

public void openConnection() {

JSch jsch = new JSch();// 创建JSch对象

try {

// 创建会话连接

sftpSession = jsch.getSession(SFTP\_REQ\_USERNAME, SFTP\_REQ\_HOST, SFTP\_REQ\_PORT);

// 设置密码

sftpSession.setPassword(SFTP\_REQ\_PASSWORD);

sftpSession.setConfig("StrictHostKeyChecking", "no");

sftpSession.setTimeout(SFTP\_REQ\_TIMEOUT);

//sftpSession.connect(22);端口号默认是22

// 创建会话

sftpSession.connect();

// 打开SFTP通道

sftpChannel = (ChannelSftp) sftpSession.openChannel("sftp");

sftpChannel.connect();

sftpChannel.setFilenameEncoding("UTF-8");

} catch (Exception e) {

// TODO: handle exception

}

}

openConnection();

sftpChannel.put("D:\\kkkk\\a.txt","/home/yhzh/test",ChannelSftp.APPEND);

System.out.println("链接成功");

sftpChannel.quit();

sftpSession.disconnect();

## 方法：

**public void put(String src, String dst, int mode)**

指定文件传输模式为mode（mode可选值为：ChannelSftp.OVERWRITE，ChannelSftp.RESUME，

ChannelSftp.APPEND）

**public void put(InputStream src, String dst)**

将本地的input stream对象src上传到目标服务器，目标文件名为dst，dst不能为目录。

**InputStream inputStream=new FileInputStream("D:\\kkkk\\a.txt");**

**sftpChannel.put(inputStream,"/home/yhzh/test/b.txt",ChannelSftp.APPEND);**

## **文件操作**

/ sftpChannel.mkdir("kkk");//创建目录

// sftpChannel.rmdir("kkk");//删除文件夹

// sftpChannel.rm("a.txt");//删除指定文件

# 文件导出

## Xml导出

|  |
| --- |
| HttpServletResponse response；  OutputStream output = response.getOutputStream();  response.reset();// 清空输出流  response.setCharacterEncoding("utf-8");  // response.setHeader("Content-disposition", "attachment;filename="+new String("中文".getBytes("utf-8"),"iso8859-1")+new //SimpleDateFormat("yyyyMMddHHmmss").format(new Date())+".xls");// 设定输出文件头    response.setHeader("Content-disposition", "attachment; filename="+new Date().getTime()+".xls");// 设定输出文件头  response.setContentType("application/msexcel");// 定义输出类型  HSSFWorkbook wb = new HSSFWorkbook();  ExportExcelExample.getExcel(wb,defautRow);  wb.write(output);  output.flush();  output.close(); |

## word文档导出

### word文档词法解释

# File操作

## File类

|  |
| --- |
| File file=new File("d:/temp/sectem");  System.out.println(file.isFile());  System.out.println(file.isDirectory());  if(!file.exists()){  System.out.println("文件不存在");  file.mkdirs();  System.out.println("文件已经创建");  }else {  System.out.println("文件已经存在");  }  **//获取文件上级路径**  **file.getParentFile()** |

## 获得类路径

|  |
| --- |
| 获得类路径this.getClass().getClassLoader().getResource("/").getPath()  即-------> /WEB-INF/classes/ |

# Java流的分类

## 按流向分:

输入流: 程序可以从中读取数据的流。  
输出流: 程序能向其中写入数据的流。

## 按数据传输单位分:

字节流: 以字节为单位传输数据的流  
字符流: 以字符为单位传输数据的流

## 概念

Java中的流分为两种，一种是字节流，另一种是字符流，分别由四个抽象类来表示（每种流包括输入和输出两种所以一共四个）:InputStream，OutputStream，Reader，Writer。Java中其他多种多样变化的流均是由它们派生出来的:

JDK所提供的所有流类位于java.io包中，都分别继承自以下四种抽象流类：

InputStream：继承自InputStream的流都是用于向程序中输入数据的，且数据单位都是字节（8位）。  
OutputSteam：继承自OutputStream的流都是程序用于向外输出数据的，且数据单位都是字节（8位）。  
Reader：继承自Reader的流都是用于向程序中输入数据的，且数据单位都是字符（16位）。  
Writer：继承自Writer的流都是程序用于向外输出数据的，且数据单位都是字符（16位）。

stream代表的是任何有能力产出数据的数据源，或是任何有能力接收数据的接收源。在Java的IO中，所有的stream（包括Inputstream和Out stream）都包括两种类型：

　　(1)字节流

　　表示以字节为单位从stream中读取或往stream中写入信息，即io包中的inputstream类和outputstream类的派生类。通常用来读取二进制数据，如图象和声音。

　　（2）字符流

　　以Unicode字符为导向的stream，表示以Unicode字符为单位从stream中读取或往stream中写入信息。

　　区别：

　　Reader和Writer要解决的，最主要的问题就是国际化。原先的I/O类库只支持8位的字节流，因此不可能很好地处理16位的Unicode字符流。Unicode是国际化的字符集(更何况[Java](http://java.chinaitlab.com/)内置的char就是16位的Unicode字符)，这样加了Reader和Writer之后，所有的I/O就都支持Unicode了。此外新类库的性能也比旧的好。

　　但是，Read和Write并不是取代InputStream和OutputStream，有时，你还必须同时使用"基于byte的类"和"基于字符的类"。为此，它还提供了两个"适配器(adapter)"类。InputStreamReader负责将InputStream转化成Reader，而OutputStreamWriter则将OutputStream转化成Writer。

## 输入输出流

|  |
| --- |
| File file=**new** File("D:/文件/aaaaaaaaaaaa");  **try** {  OutputStream out = **new** FileOutputStream(file+"/1789.mp4");  FileInputStream inputStream = **new** FileInputStream("d:文件路径");  BufferedInputStream bInputStream = **new** BufferedInputStream(inputStream,102400) ;  BufferedOutputStream buOutputStream = **new** BufferedOutputStream(out,102400);  **byte**[] data = **new** **byte**[102400];  **while**(bInputStream.read(data)!=-1){  buOutputStream.write(data);  }  buOutputStream.flush();  buOutputStream.close();  bInputStream.close();    } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  } |

## 字节流的读写

|  |
| --- |
| **try** {  FileInputStream inputStream=**new** FileInputStream("H:/KuGou/安东阳 - 分手的理由.mp3");  BufferedInputStream br=**new** BufferedInputStream(inputStream, 100000);    OutputStream outputStream=**new** FileOutputStream("MP3/mp/分手的理由.mp3");      BufferedOutputStream bOutputStream=**new** BufferedOutputStream(outputStream,100000);    **byte**[] by=**new** **byte**[100000];  **int** n;  Long time=System.*currentTimeMillis*();  System.*out*.println("当前时间"+time);  **int** count=0;  **while** ((n=br.read(by))!=-1) {  bOutputStream.write(by, 0, n);  count++;  }  bOutputStream.flush();  bOutputStream.close();  outputStream.close();  br.close();  System.*out*.println("循环次数"+count);  System.*out*.println("最终时间"+(System.*currentTimeMillis*()-time));  } **catch** (FileNotFoundException e) {  System.*err*.println("输入流异常");  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  } |

## 字符流的读写

|  |
| --- |
| try {  BufferedReader bf = new BufferedReader(new FileReader("te.txt"));  //BufferedWriter bWriter=new BufferedWriter(new FileWriter("MP3/t2.txt"));  // PrintWriter pWriter=new PrintWriter(new File("MP3/t3.txt"));  PrintWriter pWriter=new PrintWriter(new FileWriter("MP3/t4.txt"));    String string;  int count2=0;  long tim2=System.*currentTimeMillis*();  while((string=bf.readLine())!=null){  pWriter.println(string);;  count2++;  }  pWriter.flush();  pWriter.close();  bf.close();  System.*out*.println("消耗时间"+(System.*currentTimeMillis*()-tim2));    } catch (FileNotFoundException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } catch (IOException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } |

## 文件流的读写

|  |
| --- |
| **try** {  FileReader fileReader = **new** FileReader("te.txt");  FileWriter fileWriter=**new** FileWriter("MP3/newt.txt");    **char**[] ch=**new** **char**[100];  **int** n;  **int** count=0;  Long tim=System.*currentTimeMillis*();  **while**((n=fileReader.read(ch))!=-1)  {  fileWriter.write(ch,0,n);  count++;  }  fileWriter.flush();  fileWriter.close();  fileReader.close();  System.*out*.println("循环了"+count);  System.*err*.println("消耗时间"+(System.*currentTimeMillis*()-tim));  } **catch** (FileNotFoundException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } |

# 线程

## 线程方法

1. wait()方法会让调用线程休眠，直到Thread.interrupt()中断它，过了指定的时间，或者一个线程调用notify()或者notifyAll()方法唤醒它。
2. 当某个对象调用notify()时，如果有任何线程正在通过wait()等待对象，那么就会唤醒其中一个线程，当某个对象调用notifyAll()方法时，会唤醒所有正在等待对象的线程。

## 线程详解

注：一个Thread的实例线程，一旦调用start()方法，这个实例的starrted标记就标记为true，现实中不管这个线程后来有没有执行到底，只要调用一次start()方法，就再也没有机会运行了，这就意味着，通过Thread实例的start()方法，一个实例中只能产生有一个线程。

如果要在一个实例上产生多个线程，也就是我们说的线程池，就得实现Runnable接口

线程对象和线程是两个不同的概念。启动线程是说在某个线程对象上启动了该实例对应的线程，当该线程结束后，并不会就立即消失。

public class A implements Runnable {

private int x = 0;

public void run() {

System.out.println("A执行===="+(++x));

}

}

public class B extends Thread {

private int x = 0;

public void run() {

System.out.println("B执行===="+(++x));

}

}

// B线程对象只能启动一次

B b = new B();

b.start();

## 线程池

**为什么要用线程池:**

1.减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2.可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

### java.util.concurrent类 Executors

// ExecutorService pool = Executors. newSingleThreadExecutor();

ExecutorService pool = Executors.newFixedThreadPool(3);

**1. 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。

**2.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。

**3. 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。

**4.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。

线程

//同一个实例，生成多个线程，线程池

//Runnable的实例是可运行的，但它自己并不能直接运行，它需要被Thread对象来包//装才能被运行

A a = new A();

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

{

new Thread(a).start();

}

## 线程生命周期

新建 new Thread()

死亡 dead

运行 running

就绪 start()

**线程终止运行：**

1. sleep(long l) 释放cpu 的执行权，不释放锁。
2. wait()方法 释放cpu执行权，释放锁。调用 notify()方法回到就绪状态。
3. yield()方法 线程挂起，临时暂停，让线程资源释放出来。

# 任务调度

## spring配置

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:task="http://www.springframework.org/schema/task"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans-3.2.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.2.xsd  http://www.springframework.org/schema/task  http://www.springframework.org/schema/task/spring-task-3.2.xsd">    <bean id="doExportExcel" class="com.sanss.excelreport.DoExportExcel"></bean>  <task:scheduled-tasks>  <!-- 时报-->  <task:scheduled ref="doExportExcel" method="doExportHourReport"  cron="0 0 1-2 \* \* ?" />  <!-- 日报 -->  <task:scheduled ref="doExportExcel" method="doExportDayReport"  cron="0 0 0 \* \* ?" />  <!-- 周报 -->  <task:scheduled ref="doExportExcel" method="doExportWeekReport"  cron="0 0 0/168 \* \* ?" />  <!-- 月报 -->  <task:scheduled ref="doExportExcel" method="doExportMonthReport"  cron="0 0 0 1 \* ?" />  </task:scheduled-tasks>  </beans> |

## cron表达式

Seconds:可出现", - \* /"四个字符，有效范围为0-59的整数   
Minutes:可出现", - \* /"四个字符，有效范围为0-59的整数   
Hours:可出现", - \* /"四个字符，有效范围为0-23的整数   
DayofMonth:可出现", - \* / ? L W C"八个字符，有效范围为0-31的整数   
Month:可出现", - \* /"四个字符，有效范围为1-12的整数或JAN-DEc   
DayofWeek:可出现", - \* / ? L C #"四个字符，有效范围为1-7的整数或SUN-SAT两个范围。1表示星期天，2表示星期一， 依次类推 {周一：2，周二：3，周三：4，周四：5，周五：6，周六：7，周天：1}  
Year:可出现", - \* /"四个字符，有效范围为1970-2099年

## ", - \* /? L W C #"字符解释

,:表示列出枚举值值。例如：在Minutes域使用5,20，则意味着在5和20分每分钟触发一次。

-:表示范围，例如在Minutes域使用5-20，表示从5分到20分钟每分钟触发一次

\*：表示匹配该域的任意值

/：表示起始时间开始触发，然后每隔固定时间触发一次，例如在Minutes域使用5/20,则意味着5分钟触发一次，而25，45等分别触发一次

?:只能用在DayofMonth和DayofWeek两个域。它也匹配域的任意值，但实际不会。

L:表示最后，只能出现在DayofWeek和DayofMonth域，如果在DayofWeek域使用5L,意味着在最后的一个星期四触发。

W:表示有效工作日(周一到周五),只能出现在DayofMonth域，另外一点，W的最近寻找不会跨过月份

LW:这两个字符可以连用，表示在某个月最后一个工作日，即最后一个星期五。

#:用于确定每个月第几个星期几，只能出现在DayofMonth域。例如在4#2，表示某月的第二个星期三。

## 语法：

字符串以5或6个空格隔开，分为6或7个域，每一个域代表一个含义

Seconds Minutes Hours DayofMonth Month DayofWeek Year或Seconds Minutes Hours DayofMonth Month DayofWeek

秒 分 时 天 月 星期几 年 （7个域）

秒 分 时 天 月 星期几 （6个域）

秒（0~59）   
分钟（0~59）   
小时（0~23）   
天（月）（0~31，但是你需要考虑你月的天数）   
月（0~11）   
天（星期）（1~7 1=SUN 或 SUN，MON，TUE，WED，THU，FRI，SAT）   
年份（1970－2099）

**注：由于"月份中的日期"和"星期中的日期"这两个元素互斥的,必须要对其中一个设置” ?”**

**“？”字符仅被用于天（月）和天（星期）两个子表达式，表示不指定值**

## 例子：

"0 15 10 \* \* ?" 每天上午10:15触发   
"0 15 10 \* \* ? \*" 每天上午10:15触发

"0 0/5 14 \* \* ?" 在每天下午2点到下午2:55期间的每5分钟触发   
"0 0/5 14,18 \* \* ?" 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发

"0 0-5 14 \* \* ?" 在每天下午2点到下午2:05期间的每1分钟触发

"0 15 10 ? \* MON-FRI" 周一至周五的上午10:15触发   
"0 15 10 15 \* ?" 每月15日上午10:15触发

"0 15 10 L \* ?" 每月最后一日的上午10:15触发

"0 15 10 ? \* 6L" 每月的最后一个星期五上午10:15触发

"0 15 10 ? \* 6#3" 每月的第三个星期五上午10:15触发

"0 10,44 14 ? 3 WED" 每年三月的星期三的下午2:10和2:44触发

# Calendar的使用

|  |
| --- |
| /\*\*  \*  \* 获取7天前 周  \*/  @SuppressWarnings("static-access")  **public** **static** String[] getWeek(**int** month){  //month不能小于1，此次应该写个排错处理，我就不写了  SimpleDateFormat dateFormat=**new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  Calendar calendar =Calendar.*getInstance*();  calendar.set(Calendar.*YEAR*,Integer.*parseInt*(**new** SimpleDateFormat("yyyy").format(**new** Date())));  calendar.set(Calendar.*MONTH*,month);  calendar.add(calendar.*DAY\_OF\_MONTH*, -7);  calendar.set(Calendar.*HOUR\_OF\_DAY*, 0);  calendar.set(Calendar.*MINUTE*, 0);  calendar.set(Calendar.*SECOND*, 0);    **int** minDay= calendar.getActualMinimum(Calendar.*DATE*) ;  **int** maxDay = calendar.getActualMaximum(Calendar.*DATE*) ;  String minDayOfMonth=dateFormat.format(calendar.getTime());  calendar.add(calendar.*DAY\_OF\_MONTH*, +6);  calendar.set(Calendar.*HOUR\_OF\_DAY*, 23);  calendar.set(Calendar.*MINUTE*, 59);  calendar.set(Calendar.*SECOND*, 59);  String maxDayOfMonth=dateFormat.format(calendar.getTime());  String[] dataArray=**new** String[]{minDayOfMonth,maxDayOfMonth};//把该月的第一天跟最后一天扔到数组里  **return** dataArray;  } |
|  |

# 时间比较

时间截取

**int** changRececieveHout=Integer.*parseInt*(changReceiveTime.substring(11,13));

**int** changRececieveMinutes=Integer.*parseInt*(changReceiveTime.substring(14,16));

Date nowTime = **new** Date();

Calendar calendar =Calendar.*getInstance*();

Calendar calendar2 =Calendar.*getInstance*();

//---------------往后推要个一个小时---------------

calendar.set(Calendar.*YEAR*,Integer.*parseInt*(**new** SimpleDateFormat("yyyy").format(**new** Date())));

calendar.set(calendar.*DATE*, offwork\_time.~~getDate~~());

calendar.set(Calendar.*HOUR\_OF\_DAY*, offwork\_time.~~getHours~~()-1);

calendar.set(Calendar.*MINUTE*, offwork\_time.~~getMinutes~~());

calendar.set(Calendar.*SECOND*, -1);

//-------------往前推30分钟-----------------

calendar2.set(Calendar.*YEAR*,Integer.*parseInt*(**new** SimpleDateFormat("yyyy").format(**new** Date())));

calendar2.set(calendar.*DATE*, offwork\_time.~~getDate~~());

calendar2.set(Calendar.*HOUR\_OF\_DAY*, offwork\_time.~~getHours~~());

calendar2.set(Calendar.*MINUTE*, offwork\_time.~~getMinutes~~()+30);

calendar2.set(Calendar.*SECOND*, offwork\_time.~~getSeconds~~());

**long** t1 = calendar.getTime().getTime(),t2= nowTime.getTime(), t3= calendar2.getTime().getTime();

flag\_1=(t1<=t2&&t2<=t3)?**true**:**false**;

# Json与List互转

|  |
| --- |
| //使用 fastjson-1.1.41.jar 架包  import com.alibaba.fastjson.JSON;  //List 转 Json  String str = JSON.toJSONString(List);  //字符串转 json 数组  JSONArray obj=JSONArray.fromObject(aString)  //json 数组转 List  List list = JSONArray.toList(obj);  //js 传过来的json params数据 var params = Ext.encode({startTime:startTime,dimenIp:dimen\_ip,dimen\_type:dimen\_type});，java取出  JSONObject object = JSONObject.fromObject(params);  String startTime = object.getString("startTime");  String dimenIp = object.getString("dimenIp");  String dimen\_type = object.getString("dimen\_type"); |

# WritableWorkbook导出Excle文件

|  |
| --- |
| **//使用 jxl.jar 架包**  **package** com.util;  **import** java.io.File;  **import** java.io.IOException;  **import** java.util.List;  **import** java.util.Map;  **import** com.esri.arcgis.display.NewTextBezierCurveFeedback;  **import** jxl.Workbook;  **import** jxl.format.Alignment;  **import** jxl.format.Border;  **import** jxl.format.BorderLineStyle;  **import** jxl.format.Colour;  **import** jxl.format.Pattern;  **import** jxl.write.Label;  **import** jxl.write.~~VerticalAlignment~~;  **import** jxl.write.WritableCellFormat;  **import** jxl.write.WritableFont;  **import** jxl.write.WritableSheet;  **import** jxl.write.WritableWorkbook;  **import** jxl.write.biff.RowsExceededException;  @SuppressWarnings({ "unused", "deprecation" })  **public** **class** WorkBookFactory {  /\*\*  \* 获取一个工作薄实例  \*/  @SuppressWarnings({ "unused", "rawtypes" })  **public** **static** WritableWorkbook getWritableWorkbookInstance(File file,String sheetTile,**int** columnCount,List list){  WritableWorkbook workBook = **null**;  **try** {    workBook = Workbook.*createWorkbook*(file);  WritableSheet sheet = workBook.createSheet(sheetTile, 0);  //设置列宽  **for**(**int** i= 0;i<columnCount;i++)  {  sheet.setRowView(0, (**short**)(1.7\*256)); // 设置行的高度  sheet.setColumnView(i, 20); // 默认列的宽度  sheet.setColumnView(0, 30); // 设置列的宽度  }  WritableFont titlefontStyle = **new** WritableFont(WritableFont.*createFont*("微软雅黑"), 14);  titlefontStyle.setBoldStyle(WritableFont.*BOLD*);//设置字体加粗  WritableCellFormat titleFormate = **new** WritableCellFormat(titlefontStyle);//生成一个单元格样式控制对象  // titleFormate.setBackground(Colour.BLUE\_GREY, Pattern.SOLID);  // titleFormate.setBackground(Colour.DARK\_BLUE2, Pattern.SOLID);//这个深蓝底  // titleFormate.setBackground(Colour.OCEAN\_BLUE, Pattern.SOLID);//海蓝底背景  titleFormate.setBackground(Colour.*DARK\_TEAL*, Pattern.*SOLID*);//这个黑蓝底  titleFormate.setAlignment(Alignment.*CENTRE*); // 设置为右对齐  titleFormate.setVerticalAlignment(~~VerticalAlignment~~.*CENTRE*);//设置垂直居中  titleFormate.setBorder(Border.*ALL*, BorderLineStyle.*HAIR*); // 添加边框  titlefontStyle.setColour(Colour.*WHITE*);  String[] title = {"时间","时间","时间","时间","时间","时间","时间"};  **for**(**int** i= 0 ;i<columnCount;i++)  {  Label label = **new** Label(i,0,title[i],titleFormate);//参数含义 Label(column，row，content,style)  sheet.addCell(label);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*以下为内容样式\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  WritableFont fontStyle = **new** WritableFont(WritableFont.*createFont*("宋体"), 12);  fontStyle.setColour(Colour.*BLACK*);//字体颜色  WritableCellFormat cellFormat = **new** WritableCellFormat(fontStyle);  cellFormat.setBackground(Colour.*WHITE*, Pattern.*SOLID*);// 设置单元格的背景颜色  // cellFormat.setAlignment(Alignment.JUSTIFY); //自动调整 单元格内容，设置这个跟设置左右对齐会冲突，只能应用其中一个样式  cellFormat.setAlignment(Alignment.*CENTRE*); // 设置为右对齐  cellFormat.setVerticalAlignment(~~VerticalAlignment~~.*CENTRE*);//设置垂直居中  //设置边框xian  // cellFormat.setBorder(Border.ALL, BorderLineStyle.MEDIUM); // 中等细黑边框  cellFormat.setBorder(Border.*ALL*, BorderLineStyle.*getStyle*(1)); // 这个方法可以设置边框线条粗细  //填充数据进入 Excel  **for**(**int** i = 0 ;i <list.size();i++){  sheet.setRowView(i+1, (**short**)(1.6\*256)); // 设置行的高度  Map map = (Map)list.get(i);  Object[] objects={map.get("a"),map.get("b"),map.get("c"),map.get("d")};  **for**(**int** j= 0;j<columnCount;j++)  {  Label label = **new** Label(j,(i+1),objects[j].toString(),cellFormat);//Label(column，row，content,style)  sheet.addCell(label);  }  }  } **catch** (Exception e) {  e.printStackTrace();  }    **return** workBook;  }    }  //执行写入操作  workbook.write();  workbook.close(); |

# 获取HttpServletRequest、HttpServletResponse对象

在CXF里不能直接方法HttpServletRequest和HttpServletResponse，

但可以从两个地方获取当前请求相关的Http对象，

1. 从服务器类实现中获取
2. 从拦截器中获取

|  |
| --- |
| //从拦截器中获取Message对象  Message message = PhaseInterceptorChain.*getCurrentMessage*();  HttpServletRequest request = (HttpServletRequest) message.get(AbstractHTTPDestination.*HTTP\_REQUEST*);  HttpServletResponse response = (HttpServletResponse) message.get(AbstractHTTPDestination.*HTTP\_RESPONSE*);  ServletOutputStream outputStream = response.getOutputStream();  response.setCharacterEncoding("utf-8");  response.setHeader("Content-disposition", "attachment;filename=1234.xls");  WritableWorkbook workbook=WorkBookFactory.*getWritableWorkbookInstance*(outputStream, "卧槽", 7, list2);  workbook.write();  workbook.close(); |

# 配置文件的读取与PropertyPlaceholderConfigurer的用法

## 方法一

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans           xmlns="http://www.springframework.org/schema/beans"           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"           xmlns:p="http://www.springframework.org/schema/p"           xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/**spring-beans-2.5.xsd**">  <!-- 使用spring提供的**PropertyPlaceholderConfigurer**读取数据库配置信息**.properties** -->           <bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">                     <property name="**locations**">                              <list>  <!--  这里的classpath可以认为是项目中的src属性名是 **locations，使用子标签<list></list>可以指定多个数据库的配置文件，这里指定了一个**  -->                                       <value>classpath:resource/config/jdbc.properties</value>                              </list>                     </property>           </bean>  </beans> |

## 方法二

|  |
| --- |
| <bean id="propertyConfigurer" class="org.springframework.beans.factory.config.**PropertyPlaceholderConfigurer**">      <property name="**locations**">        <list>           <value>**/WEB-INF/ jdbc.properties**</value>        </list>      </property>    </bean |

## 方法三，分散配置

|  |
| --- |
| <bean id="propertyConfigurerForProject1" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <!--order属性代表其加载顺序-->  <property name="order" value="1" />  <!--ignoreUnresolvablePlaceholders为是否忽略不可解析的 Placeholder-->  <property name="ignoreUnresolvablePlaceholders" value="true" />  <property name="location">  <value>classpath:/spring/include/dbQuery.properties</value>  </property>  </bean> |

|  |
| --- |
| <bean id="propertyConfigurerForProject2" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  <!--order属性代表其加载顺序-->  <property name="order" value="2" />  <!--ignoreUnresolvablePlaceholders为是否忽略不可解析的 Placeholder-->  <property name="ignoreUnresolvablePlaceholders" value="true" />  <property name="locations">  <list>  <value>classpath:/spring/include/jdbc-parms.properties</value>  <value>classpath:/spring/include/base-config.properties</value>  </list>  </property>  </bean> |

注意：如果配置了多个PropertyPlaceholderConfigurer，则ignoreUnresolvablePlaceholders**属性必须设置且为true**,否则propertyConfigurerForProject2的properties文件不会被加载.

## 方法四，自定义配置读取配置文件

### ConfigFactory.java文件

|  |
| --- |
| **package** com.util;  **import** java.io.IOException;  **import** java.util.Properties;  **import** org.springframework.beans.factory.config.PropertyPlaceholderConfigurer;  //自定义配置文件要继承 PropertyPlaceholderConfigurer  **public** **class** ConfigFactory **extends** PropertyPlaceholderConfigurer {  **private** **static** Properties *props*;  //主要思想是通过PropertyPlaceholderConfigurer的mergeProperties方法获取spring加载完成的键值对;  **public** Properties mergeProperties(){  **try**  {  *props*=**super**.mergeProperties();  } **catch** (IOException e) {  e.printStackTrace();  }    **return** *props*;  }  **public** **static** String getProperty(String key) {  **return** *props*.getProperty(key);  }  } |

### 配置xml文件，添加以下相应的属性文件代码

|  |
| --- |
| <bean id="propertyConfigurer2" class="com.util.ConfigFactory">  <property name="ignoreResourceNotFound" value="true" />  <property name="locations">  <list>  <value>classpath:config.properties</value>  <value>classpath:jdbc.properties</value>  <value>classpath:sqlload.properties</value>  </list>  </property>  </bean> |

# Dao层利用模板（springMVC框架用）

|  |
| --- |
| **package** com.sanss.alarm.dao;  **import** javax.annotation.Resource;  **import** javax.sql.DataSource;  **import** org.springframework.jdbc.core.support.JdbcDaoSupport;  **public** **class** SuperDao **extends** JdbcDaoSupport {  @Resource(name = "sessionFactory") //数据源，跟xml文件数据源bean id名字一样  **public** **void** setSessionFactoryOverride(DataSource dataSource) {  **super**.setDataSource(dataSource);  }  } |

# 日期时间转换

SimpleDateFormat format2 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

SimpleDateFormat format = **new** SimpleDateFormat("yyyyMMddHHmmss");

Calendar calendar=Calendar.*getInstance*();

calendar.set(2016, (5-1), 6, 9, 0, 20);

String endTime ="20160506143058";

//时间字符串转日日期

Date date = format.parse(endTime);

System.*err*.println("时间=="+format.parse(endTime).toString());

//手动设置日历时间

calendar.setTime(format.parse(endTime));

System.*out*.println("打印日历时间=="+format2.format(calendar.getTime()));

# list里的map对象排序

Collections.*sort*(list, **new** Comparator<Map<String, Object>>(){

**public** **int** compare(Map<String, Object> o1, Map<String, Object> o2) {

SimpleDateFormat format = **new** SimpleDateFormat("yyyy/MM/dd HH:mm:ss");

**try** {

**return** format.parse(o1.get("SampleSecond").toString()).getTime()>format.parse(o2.get("SampleSecond").toString()).getTime()?1:( format.parse(o1.get("SampleSecond").toString()).getTime()==format.parse(o1.get("SampleSecond").toString()).getTime()?0:-1);

} **catch** (ParseException e) {

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

e.printStackTrace();

}

**return** -1;

}

});

# Spring使用

## spring获取bean

ServletContext servletContext = request.getSession().getServletContext();

ApplicationContext factory = WebApplicationContextUtils.*getWebApplicationContext*(servletContext);

RecordsQueryServiceImpl service = (RecordsQueryServiceImpl) factory.getBean("RecordsQueryService");

## Spring JdbcTemplate方法详解

* **execute方法：**可以用于执行任何SQL语句，一般用于执行DDL语句；
* **update方法及batchUpdate方法：**update方法用于执行新增、修改、删除等语句；batchUpdate方法用于执行批处理相关语句；
* **query方法及queryForXXX方法：**用于执行查询相关语句；
* **call方法：**用于执行存储过程、函数相关语句。

1. //1.查询一行数据并返回int型结果
2. jdbcTemplate.queryForInt("select count(\*) from test");
3. //2. 查询一行数据并将该行数据转换为Map返回
4. jdbcTemplate.queryForMap("select \* from test where name='name5'");
5. //3.查询一行任何类型的数据，最后一个参数指定返回结果类型
6. jdbcTemplate.queryForObject("select count(\*) from test", Integer.**class**);
7. //4.查询一批数据，默认将每行数据转换为Map
8. jdbcTemplate.queryForList("select \* from test");
9. //5.只查询一列数据列表，列类型是String类型，列名字是name
10. jdbcTemplate.queryForList("
11. select name from test where name=?", new Object[]{"name5"}, String.**class**);
12. //6.查询一批数据，返回为SqlRowSet，类似于ResultSet，但不再绑定到连接上
13. SqlRowSet rs = jdbcTemplate.queryForRowSet("select \* from test");

# 存储过程：

Connection conn = **this**.getJdbcTemplate().getDataSource().getConnection();

CallableStatement cs = **null**;

cs = conn.prepareCall("{? = call P\_Tkg\_AddOne(?,?,?,?,?,?,?,?,?,?,?)}");

cs.registerOutParameter(1, java.sql.Types.***INTEGER***);

cs.setInt(2,operatorId);

cs.setInt(3,systemId);

cs.setString(4,tkgMenm);

cs.setString(5,total);

cs.setString(6,destName);

cs.execute();

cs.close();

conn.close();

# Dom4J解析xml

**import** javax.xml.parsers.DocumentBuilder;

**import** javax.xml.parsers.DocumentBuilderFactory;

**import** org.dom4j.Attribute;

**import** org.dom4j.Document;

**import** org.dom4j.Element;

**import** org.dom4j.Node;

**import** org.dom4j.io.SAXReader;

# 数据表导入导出

## 导出

exp uu/111111@10.18.13.44/orcl file=d:/sys\_menu.dmp tables=sys\_menu full=y

报错：

[EXP-00026: 指定了冲突模式](http://blog.csdn.net/feier7501/article/details/25432249)

解决：去掉 ~~full =y~~ 有可能用户权限不一样导致的，如dba 跟普通用户

exp uu/111111@10.18.13.44/orcl file=d:/sys\_menu.dmp tables=(uu.sys\_menu)

## 导入

报错：

[oracle 导入DMP文件时IMP-00013: 只有 DBA 才能导入由其他 DBA 导出的文件 IMP-00000: 未成功终止导入](http://blog.csdn.net/breaker892902/article/details/11004495)

解决：

授予该用户 dba 权限 grant dba to uu;

C:\>imp uu/111111@10.18.13.44/orcl file=d:/sys\_menu.dmp full=y

## 导入已经存在的表

表要有唯一主键，不然会重复导入

imp uu/111111@10.18.13.60/orcl file=d:/book.dmp full=y ignore=y;

查看表约束 user\_constraints；

select constraint\_name from user\_constraints where table\_name='BOOK';

## 添加主键

alter table book add constraint pk\_id primary key (id);

## 删除主键；

alter table book drop constraint pk\_id;