[线程和线程池 1](#_Toc501539738)

[数据库连接池原理 1](#_Toc501539739)

[四类线程池 2](#_Toc501539740)

[线程安全，预防脏读 3](#_Toc501539741)

[加密算法 5](#_Toc501539742)

[类的属性反射 7](#_Toc501539743)

[文件读取 8](#_Toc501539744)

[FastJson 13](#_Toc501539745)

[Json字符串转 map 对象 13](#_Toc501539746)

[Json字符串转 List 集合 13](#_Toc501539747)

[复杂JSON字符串 转 List集合 14](#_Toc501539748)

[解析JSON字符串到指定对象class 15](#_Toc501539749)

[Map对象转化成json字符串 16](#_Toc501539750)

[JackJson 16](#_Toc501539751)

[指定对象Class转成 json字符串 17](#_Toc501539752)

[List集合转化成json字符串 17](#_Toc501539753)

[Json字符串转化成指定Class类 18](#_Toc501539754)

[Json字符串转化成集合List 18](#_Toc501539755)

# 线程和线程池

C3PO数据库连接池

<http://blog.csdn.net/suzunshou/article/details/47014545>

Dbutils基础知识介绍

<https://www.cnblogs.com/CQY1183344265/p/5854418.html>

# 数据库连接池原理

连接池基本的思想是在系统初始化的时候，将数据库连接作为对象存储在内存中，当用户需要访问数据库时，并非建立一个新的连接，而是从连接池中取出一个已建立的空闲连接对象。

使用完毕后，用户也并非将连接关闭，而是将连接放回连接池中，以供下一个请求访问使用。而连接的建立、断开都由连接池自身来管理。

同时，还可以通过设置连接池的参数来控制连接池中的初始连接数、连接的上下限数以及每个连接的最大使用次数、最大空闲时间等等。也可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。

线程的三种实现方式：

1. Thread类，可extends
2. Runable接口，需要实现run方法，通过new Thread类调用
3. callable接口，需要实现call方法，通过 FutureTask 调用，并获取返回值，可结合线程池使用。

# 四类线程池

### newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

这种类型的线程池特点是：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

### newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

### newSingleThreadExecutor

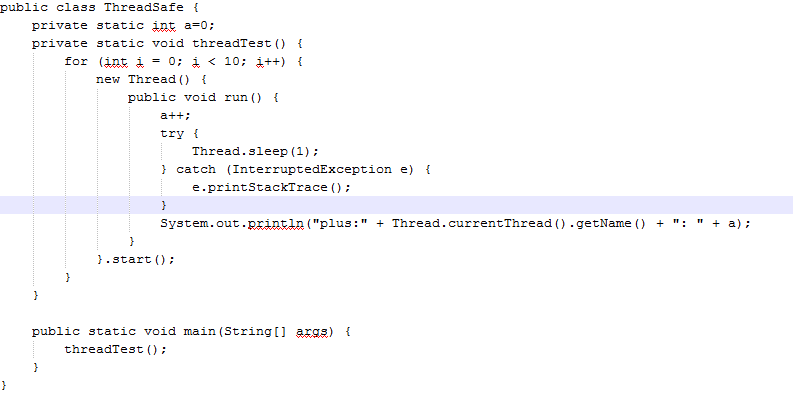
创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

### newScheduleThreadPool

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。

# 线程安全，预防脏读

线程脏读：



输出：

plus:Thread-5: 4

plus:Thread-3: 4

plus:Thread-0: 4

plus:Thread-4: 5

plus:Thread-8: 5

plus:Thread-9: 6

plus:Thread-7: 9

plus:Thread-2: 10

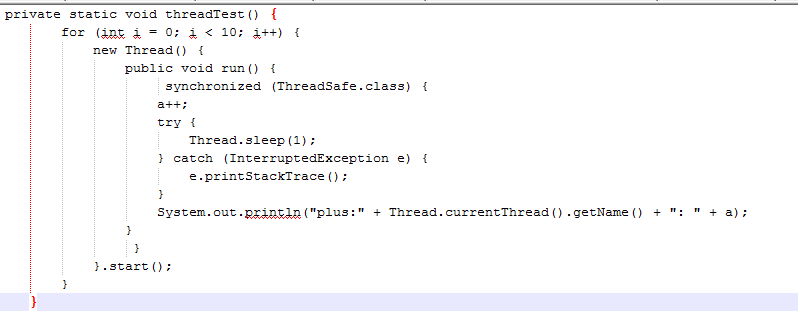
plus:Thread-6: 10

plus:Thread-1: 10

为了防止线程并发时的脏读情况，可采用以下方法避免

### 线程同步synchronized

synchronized是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。



输出：

plus:Thread-3: 1

plus:Thread-7: 2

plus:Thread-0: 3

plus:Thread-6: 4

plus:Thread-9: 5

plus:Thread-8: 6

plus:Thread-2: 7

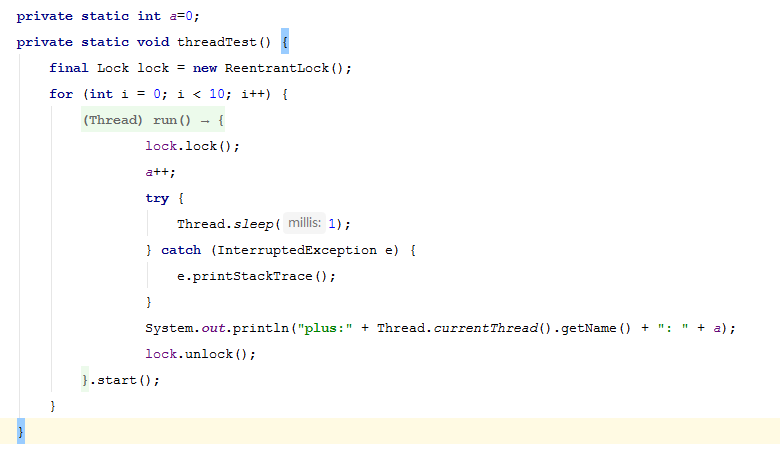
plus:Thread-5: 8

plus:Thread-4: 9

plus:Thread-1: 10

### 使用ReentrantLock类

Lock()--Unlock()



tryLock()—unlock()



输出：

plus:Thread-0: 5

plus:Thread-6: 8

plus:Thread-1: 13

plus:Thread-2: 14

plus:Thread-4: 15

plus:Thread-8: 16

plus:Thread-5: 17

plus:Thread-9: 18

plus:Thread-3: 19

plus:Thread-7: 20

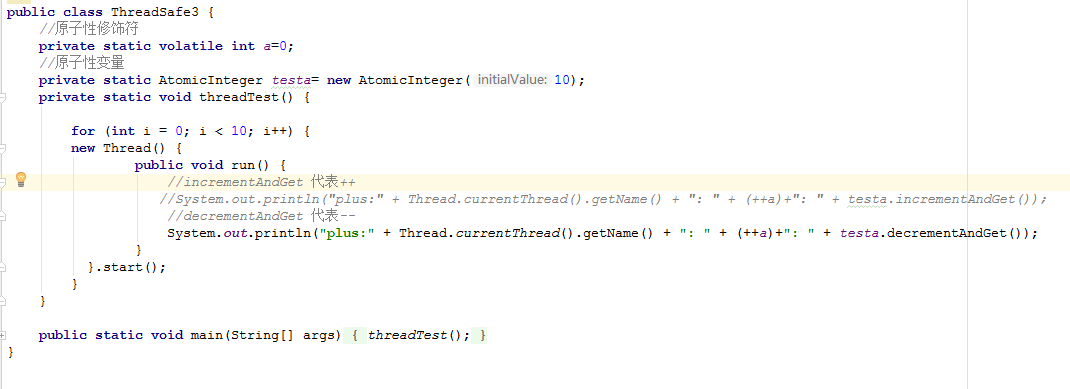
lockInterruptibly()

### 原子变量AtomicInteger内部锁

Java 5新增了AtomicInteger类，该类包含方法getAndIncrement()以及getAndDecrement()，这两个方法实现了原子加以及原子减操作，但是比较不同的是这两个操作没有使用任何加锁机制，属于无锁操作。

原子变量类相当于一种泛化得volatile变量，能够支持原子的和有条件的读-改-写操作。不需要挂起或重新调度线程。

AtomicInteger只能同步一个值，不能同步代码块



输出：

plus:Thread-1: 1: 9

plus:Thread-0: 2: 8

plus:Thread-3: 4: 6

plus:Thread-5: 3: 7

plus:Thread-2: 5: 5

plus:Thread-8: 6: 4

plus:Thread-9: 7: 3

plus:Thread-7: 8: 2

plus:Thread-4: 9: 1

plus:Thread-6: 10: 0

# 加密算法

### 原始加密算法

public final static String MD5(String pwd) {

//用于加密的字符

char md5String[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9','A', 'B', 'C', 'D', 'E', 'F' };

try {

//使用平台的默认字符集将此 String 编码为 byte序列，并将结果存储到一个新的 byte数组中

byte[] btInput = pwd.getBytes();

//信息摘要是安全的单向哈希函数，它接收任意大小的数据，并输出固定长度的哈希值。

MessageDigest mdInst = MessageDigest.getInstance("MD5");

//MessageDigest对象通过使用 update方法处理数据， 使用指定的byte数组更新摘要

mdInst.update(btInput);

// 摘要更新之后，通过调用digest（）执行哈希计算，获得密文

byte[] md = mdInst.digest();

// 把密文转换成十六进制的字符串形式

int j = md.length;

char str[] = new char[j \* 2];

int k = 0;

for (int i = 0; i < j; i++) { // i = 0

byte byte0 = md[i]; //95

str[k++] = md5String[byte0 >>> 4 & 0xf]; // 5

str[k++] = md5String[byte0 & 0xf]; // F

}

//返回经过加密后的字符串

return new String(str);

} catch (Exception e) {

return null;

}

}

在main方法中调用：

String str4 = "123456";

System.out.println(encodeByMd5(str4));

//输出：4QrcOUm6Wau+VuBX8g+IPg==

### commons-codec 包下的md5

//使用commons.codec包下DigestUtils的md5Hex方法，对abc进行加密

String str1 = DigestUtils.md5Hex(str);

System.out.println(str1.toUpperCase());

输出：abc转化成大写字母：900150983CD24FB0D6963F7D28E17F72

### 比MD5更安全的 SHA-1加密

String str11 = DigestUtils.sha1Hex(str);

System.out.println(str11);

输出： a9993e364706816aba3e25717850c26c9cd0d89d

### 使用Base64的decodeBase64、encodeBase64进行加解密

byte[] b = Base64.decodeBase64("YWJj".getBytes());

System.out.println(new String(b));//输出abc

byte[] code = Base64.encodeBase64("123456".getBytes());

System.out.println(new String(code)+"--");//输出：MTIzNDU2

### 使用安全性更高的Md5Crypt，

Md5Crypt基于算子的MD5，以$开头，安全性高，每次加密的结果都不一样。

String str2 = Md5Crypt.md5Crypt("abc".getBytes());

System.out.println(str2);//abc

# 类的属性反射

### 使用Field将类的属性进行底层封装

前提条件：

存在两个person类，有相同的属性

1. 第一种方法：

读取person类的各个属性，及值

public static void getFile(Object object) {

Class clz=object.getClass();

Field[] fields = clz.getDeclaredFields();

for (Field field : fields) {

try {

field.setAccessible(true);

System.out.println(field.getName()+ " value "+ field.get(object));

} catch (IllegalArgumentException | IllegalAccessException e) {

e.printStackTrace();

}

}

}

将反射类属性的方法封装成copy，使用时直接传参：

public static void copy(Object from, Object to) {

Class fromClass= from.getClass();

Class toClass= to.getClass();

Field[] fields = toClass.getDeclaredFields();

for (Field field : fields) {

try {

field.setAccessible(true);

System.out.println(field.getName());

Field fromfield = fromClass.getDeclaredField(field.getName());

fromfield.setAccessible(true);

field.set(to, fromfield.get(from));

} catch (Exception e) {

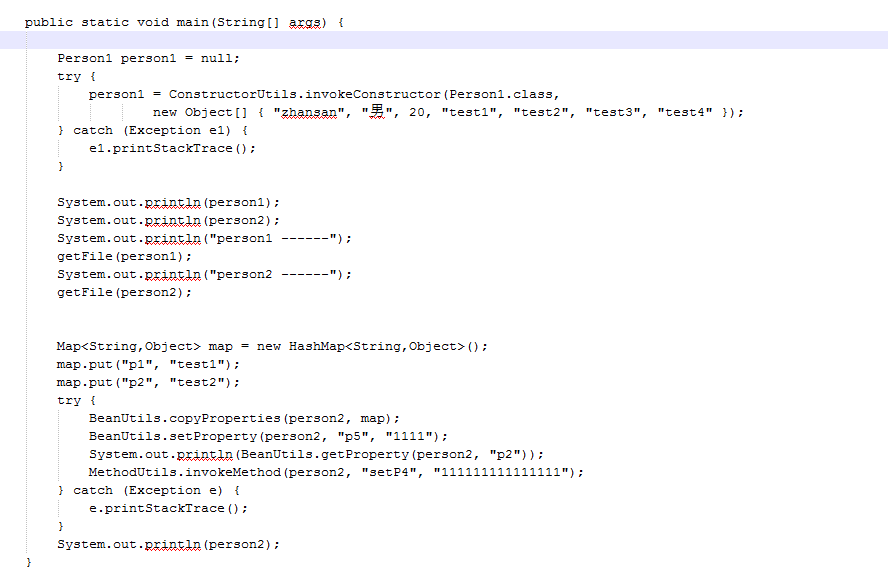
e.printStackTrace();

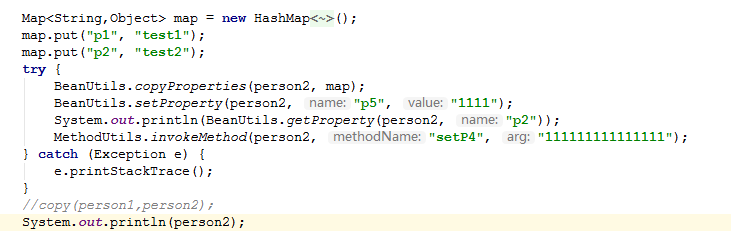
}

}

}

### 使用commons-beanutils包下的BeanUtils工具类





# 配置文件读取

### properties文件读取

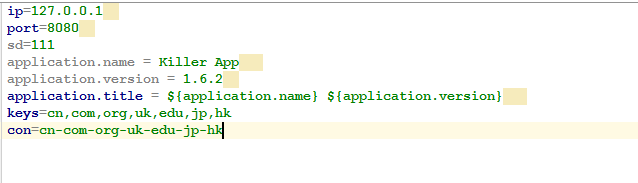
#### JDK自带的的IO包读取properties文件

import java.io.InputStreamReader;  
import java.util.Properties;  
  
public class PropertiesTestUtis {  
 private static Properties *properties* = null;  
  
 static {  
 if(null == *properties*){  
 *properties* = new Properties();  
 InputStreamReader reader = null;  
 try{  
 reader = new InputStreamReader(PropertiesTestUtis.class.getClassLoader().getResourceAsStream("config.properties"),"utf-8");  
 *properties*.load(reader);  
 reader.close();  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 if(reader != null){  
 reader = null;  
 }  
 }  
 }  
 }  
  
 private PropertiesTestUtis(){}  
  
 public static String getString(String key){  
 return *properties*.getProperty(key);  
 }  
 public static void main (String args[]){  
 String name = PropertiesTestUtis.*getString*("application.name");  
 System.*out*.println(" name="+name );  
 String title = PropertiesTestUtis.*getString*("application.title");  
 System.*out*.println(" title="+title );  
 }  
}

#### 使用commons.configuration包读取properties文件

使用commons.configuration包下的PropertiesConfiguration类读取properties文件

Properties文件内容：





*/\*\*   
 \*   
 \* 依次输出结果如下   
 \* 127.0.0.1   
 \* 8080   
 \* Killer App 1.6.2   
 \* defaultid   
 \* [cn, com, org, uk, edu, jp, hk]   
 \* [cn, com, org, uk, edu, jp, hk]   
 \* [cn, com, org, uk, edu, jp, hk]   
 \*/*

### commons-configuration 对XML文件读取

使用commons.configuration包下的XMLConfiguration类读取xml文件

public static void testReadXML() {  
 try{  
 XMLConfiguration config = new XMLConfiguration("config.xml");  
 //对于单独的元素，可以直接通过元素名称获取值  
 String str = config.getString("boy");  
 System.*out*.println("boy:\t"+str);  
   
 //对于循环出现的嵌套元素，可以通过父元素.子元素来获取集合值  
 List<Object> names = config.getList("student.name");  
 System.*out*.println("student.name:\t"+Arrays.*toString*(names.toArray()));  
   
 //对于单独的元素包含的值有多个的话如：a,b,c,d 可以通过获取集合  
 List<Object> titles = config.getList("title");  
 System.*out*.println("title:\t"+Arrays.*toString*(titles.toArray()));  
   
 //对于标签元素的属性，可以通过 标签[@属性名]这样的方式来获取  
 String size = config.getString("ball[@size]");  
 System.*out*.println("ball[@size]:\t"+size);  
   
 //对于嵌套标签的话，想获得某一项的话可以通过 标签名(索引名)这样的方式来获取  
 String id = config.getString("student(1)[@id]");  
 System.*out*.println("student(1)[@id]:\t"+id);  
   
 //对于标签里面的属性名称可以这么取  
 String go = config.getString("student(0).name[@go]");  
 System.*out*.println("student(0).name[@go]:\t"+go);  
   
 //对于标签里面的属性名称还可以这么取  
 go = config.getString("student.name(0)[@go]");  
 System.*out*.println("student.name(0)[@go]:\t"+go);  
   
 */\*\* 依次输出结果如下：  
 boy: tom  
 student.name: [lily, lucy]  
 title: [abc, cbc, bbc, bbs]  
 ball[@size]: 20  
 student(1)[@id]: 2  
 student(0).name[@go]: common1  
 student.name(0)[@go]: common1  
 \*/* } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

先创建对象person, @XmlRootElement

import java.util.Set;  
import javax.xml.bind.annotation.XmlElement;  
import javax.xml.bind.annotation.XmlElements;  
import javax.xml.bind.annotation.XmlRootElement;  
  
@XmlRootElement  
public class Person1 {  
 private String name;  
 private String sex;  
 private int age;  
 private String p1;  
 private String p2;  
 private String p3;  
 private String p4;  
 private Set<Book> books;  
   
 public Person1() {  
   
 }  
  
 @XmlElements(value={@XmlElement(name="books", type=Book.class)})  
 public Set<Book> getBooks() {  
 return books;  
 }  
 public void setBooks(Set<Book> books) {  
 this.books = books;  
 }  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public String getSex() {  
 return sex;  
 }  
 public void setSex(String sex) {  
 this.sex = sex;  
 }  
 public int getAge() {  
 return age;  
 }  
 public void setAge(int age) {  
 this.age = age;  
 }  
 public String getP1() {  
 return p1;  
 }  
 public void setP1(String p1) {  
 this.p1 = p1;  
 }  
 public String getP2() {  
 return p2;  
 }  
 public void setP2(String p2) {  
 this.p2 = p2;  
 }  
 public String getP3() {  
 return p3;  
 }  
 public void setP3(String p3) {  
 this.p3 = p3;  
 }  
   
 @XmlElement(name="hello2")  
 public String getP4() {  
 return p4;  
 }  
 public void setP4(String p4) {  
 this.p4 = p4;  
 }  
 public Person1(String name, String sex, int age, String p1, String p2, String p3, String p4) {  
 super();  
 this.name = name;  
 this.sex = sex;  
 this.age = age;  
 this.p1 = p1;  
 this.p2 = p2;  
 this.p3 = p3;  
 this.p4 = p4;  
 }  
 @Override  
 public String toString() {  
 return "Person1 [name=" + name + ", sex=" + sex + ", age=" + age + ", p1=" + p1 + ", p2=" + p2 + ", p3=" + p3  
 + ", p4=" + p4 + "]";  
 }  
}

#### 将person对象转化成xml后的内容输出到控制台

JAXBContext jc;  
try {  
 jc = JAXBContext.*newInstance*(Person1.class);  
 Marshaller ms = jc.createMarshaller();  
 Person1 person1 = new Person1("zhangsan", "11", 20, "test1", "test2", "test3", "test4");  
 Book book =new Book("book1", 110);  
 Book boo2 =new Book("book2", 100);  
 Set<Book> booklist = new HashSet<Book>();  
 booklist.add(book);  
 booklist.add(boo2);  
 person1.setBooks(booklist);  
  
 String path=System.*getProperty*("user.dir")+File.*separator*+"testdata"+File.*separator*;  
 File file = new File(path+"test.xml");  
 //将person对象转化成xml后的内容输出到控制台  
 ms.marshal(person1, System.*out*);

#### 将person对象转化成xml后的内容输入到xml文件中

ms.marshal(person1, file);

**public static void** convertObjectToXmlstr(Object obj){

**try** {

*jc* = JAXBContext.*newInstance*(obj.getClass());

Marshaller ms= *jc*.createMarshaller();

*//将person对象转化成xml后，直接在控制台输出*

ms.marshal(obj,System.***out***);

} **catch** (JAXBException e) {

e.printStackTrace();

}

}

**public static void** convertObjectToXmlfile(Object obj, String path){

**try** {

*jc* = JAXBContext.*newInstance*(obj.getClass());

Marshaller ms= *jc*.createMarshaller();

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

*//将person对象转化成xml后，存放到指定目录*

ms.marshal(obj,file);

} **catch** (JAXBException e) {

e.printStackTrace();

}

}

#### 将xml字符串中的内容转化成 person对象

JAXBContext jc3 = JAXBContext.*newInstance*(Person1.class);  
Unmarshaller unmar3 = jc3.createUnmarshaller();  
String path3=System.*getProperty*("user.dir")+File.*separator*+"testdata"+File.*separator*;  
File file3 = new File(path3+"test.xml");  
//将xml转化成person对象 并输出  
Person1 person = (Person1) unmar3.unmarshal(file3);  
System.*out*.println(person);

**public static** Object convertXmlfileToObject(Class clz,String path){

Object obj=**null**;

**try** {

*jc* = JAXBContext.*newInstance*(clz);

Unmarshaller unmarshaller = *jc*.createUnmarshaller();

FileReader fileReader = **new** FileReader(path);

obj = unmarshaller.unmarshal(fileReader);

} **catch** (Exception e) {

e.printStackTrace();

}

**return** obj;

}

**public static** Object convertXmlstrToObject(Class clz,String xmlstr){

*//将xml字符串转化成* *person对象*

Object obj=**null**;

**try** {

*jc* = JAXBContext.*newInstance*(clz);

Unmarshaller unmarshaller = *jc*.createUnmarshaller();

StringReader sr = **new** StringReader(xmlstr);

obj = unmarshaller.unmarshal(sr);

} **catch** (JAXBException e) {

e.printStackTrace();

}

**return** obj;

}

### commons-io 的FileUtils工具类读取文件

需要包：

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.6</version>

</dependency>

#### 读文件readFileToString、readLines

//读文件

//static String:readFileToString(File file, String encoding)

System.out.println(FileUtils.readFileToString(new File("D:/a/b/cxyapi.txt"), "UTF-8"));

//static List<String>:readLines(File file, String encoding) 返回一个list

List<String> list = FileUtils.*readLines*(new File("d:/source/pushClaim.xml"),"utf-8");  
for(String l : list){  
 System.*out*.println(l);  
}

#### 写文件writeStringToFile、write

//四个参数分别为：目标文件，写入的字符串，字符集，是否追加  
FileUtils.*writeStringToFile*(new File("d:/source/pushClaim.xml"),"我是被写入的内容","",true);  
  
//write可以接受charsequence类型的数据，string,stringbuilder和stringbuffer都是实现了charsequence接口  
FileUtils.*write*(new File("d:/source/pushClaim.xml"),"target char sequence","utf-8",true);

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

lines.add("欢迎访问:");

lines.add("www.cxyapi.com");

FileUtils.writeLines(new File("D:/a/b/cxyapi.txt"),lines,true);

#### 删除文件/文件夹deleteDirectory、deleteQuietly

//删除目录

//static void:deleteDirectory(File directory)

FileUtils.deleteDirectory(new File("D:/not/cxyapi"));

//static boolean:deleteQuietly(File file) 文件夹不是空任然可以被删除，永远不会抛出异常

FileUtils.deleteQuietly(new File("D:/not/cxyapi"));

#### 移动、复制文件/文件夹

仅仅拷贝目录

File sourceDir= new File("D:\\source");  
File targetDir = new File("C:\\target");  
FileUtils.*copyDirectory*(sourceDir,targetDir,DirectoryFileFilter.*DIRECTORY*);

将sourcefile中的文件内容移到targetfile的pc.xml文件中，source中的文件被移除，pc.xml是不存在的，移动时自动创建。

File sourcefile = new File("D:\\source\\pushClaim.xml");  
File targetfile = new File("C:\\target\\pc.xml");  
FileUtils.*moveFile*(sourcefile,targetfile);

将sourcefile中的文件内容复制到targetfile的pc.xml文件中，pc.xml是不存在的，移动时自动创建。

File sourcefile1 = new File("D:\\source\\pushClaim.xml");  
File targetfile1 = new File("C:\\target\\pc1.xml");  
FileUtils.*copyFile*(sourcefile1,targetfile1,true);

拷贝文件到目录中

FileUtils.*copyFileToDirectory*(new File("d:/cxyapi.xml"), new File("d:/cxyapi"));

拷贝url到文件

FileUtils.*copyURLToFile*(new URL("http://www.cxyapi.com/rss/cxyapi.xml"), new File("d:/cxyapi.xml"));

# FastJson

需要包：

<**dependency**>  
 <**groupId**>com.alibaba</**groupId**>  
 <**artifactId**>fastjson</**artifactId**>  
 <**version**>1.2.43</**version**>  
</**dependency**>

### Json字符串转 map 对象

String jsonString1 = **"{\"param5\":\"value5\",\"param3\":\"value3\",\"param4\":\"value4\",\"param1\":\"value1\",\"param2\":\"value2\"}"**;  
System.***out***.println(jsonString1);  
Map<String,String> stringStringMap = (Map<String,String>) JSON.*parse*(jsonString1);  
**for** (String s : stringStringMap.**keySet**()) {  
 System.***out***.println(s + **"==>"** +stringStringMap.**get**(s));  
}

输出：

{"param5":"value5","param3":"value3","param4":"value4","param1":"value1","param2":"value2"}

param5==>value5

param3==>value3

param4==>value4

param1==>value1

param2==>value2

### Json字符串转 List 集合

String jsonString2 = **"[{\"param5\":\"value5\",\"param3\":\"value3\",\"param4\":\"value4\",\"param1\":\"value1\",\"param2\":\"value2\"},{\"p1\":\"v1\",\"p2\":\"v2\",\"p3\":\"v3\",\"p4\":\"v4\",\"p5\":\"v5\"}]"**;  
System.***out***.println(jsonString2);  
List<Map<String,String>> mapList = JSON.*parseObject*(jsonString2, **new** TypeReference<List<Map<String,String>>>(){});  
**for** (Map<String, String> stringObjectMap : mapList) {  
 **for** (String s : stringObjectMap.**keySet**()) {  
 System.***out***.println(s + **"==>"** + stringObjectMap.**get**(s));  
 }  
}

输出：

[{"param5":"value5","param3":"value3","param4":"value4","param1":"value1","param2":"value2"},{"p1":"v1","p2":"v2","p3":"v3","p4":"v4","p5":"v5"}]

param5==>value5

param3==>value3

param4==>value4

param1==>value1

param2==>value2

p1==>v1

p2==>v2

p3==>v3

p4==>v4

p5==>v5

### 复杂JSON字符串 转 List集合

String jsonString3 = **"{\"count\":2,\"list\":[{\"param5\":\"value5\",\"param3\":\"value3\",\"param4\":\"value4\",\"param1\":\"value1\",\"param2\":\"value2\"},{\"p1\":\"v1\",\"p2\":\"v2\",\"p3\":\"v3\",\"p4\":\"v4\",\"p5\":\"v5\"}]}"**;  
System.***out***.println(jsonString3);  
Map<String,Object> map = JSON.*parseObject*(jsonString3);  
System.***out***.println(map.**get**(**"count"**));  
String tempjsonString3 = map.**get**(**"list"**).toString();  
System.***out***.println(tempjsonString3);  
List<Map<String,String>> mapList2 = JSON.*parseObject*(tempjsonString3, **new** TypeReference<List<Map<String,String>>>(){});  
**for** (Map<String, String> stringObjectMap : mapList2) {  
 **for** (String s : stringObjectMap.**keySet**()) {  
 System.***out***.println(s + **"==>"** + stringObjectMap.**get**(s));  
 }  
}

输出：

{"count":2,"list":[{"param5":"value5","param3":"value3","param4":"value4","param1":"value1","param2":"value2"},{"p1":"v1","p2":"v2","p3":"v3","p4":"v4","p5":"v5"}]}

2

[{"param5":"value5","param3":"value3","param4":"value4","param1":"value1","param2":"value2"},{"p1":"v1","p2":"v2","p3":"v3","p4":"v4","p5":"v5"}]

param5==>value5

param3==>value3

param4==>value4

param1==>value1

param2==>value2

p1==>v1

p2==>v2

p3==>v3

p4==>v4

p5==>v5

### 解析JSON字符串到指定对象class

第一种方式，*parseArray*：

String jsonString4 = **"[{\"age\":12,\"date\":1465475917155,\"name\":\"s1\"},{\"age\":12,\"date\":1465475917175,\"name\":\"s2\"},{\"age\":12,\"date\":1465475917175,\"name\":\"s3\"},{\"age\":12,\"date\":1465475917175,\"name\":\"s4\"},{\"age\":12,\"date\":1465475917175,\"name\":\"s5\"},{\"age\":12,\"date\":1465475917175,\"name\":\"s6\"}]"**;  
System.***out***.println(jsonString4);  
List<Student> studentList = JSON.*parseArray*(jsonString4,Student.**class**);  
**for** (Student student : studentList) {  
 System.***out***.println(student);  
}

第二种方式，*parseObject*：

System.***out***.println(jsonString4);  
List<Student> studentList2 = JSON.*parseObject*(jsonString4,**new** TypeReference<List<Student>>(){});  
**for** (Student student : studentList2) {  
 System.***out***.println(student);  
}

输出：

[{"age":12,"date":1465475917155,"name":"s1"},{"age":12,"date":1465475917175,"name":"s2"},{"age":12,"date":1465475917175,"name":"s3"},{"age":12,"date":1465475917175,"name":"s4"},{"age":12,"date":1465475917175,"name":"s5"},{"age":12,"date":1465475917175,"name":"s6"}]

Student{name='s1', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s2', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s3', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s4', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s5', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s6', age=12, date=Thu Jun 09 20:38:37 CST 2016}

### Map对象转化成json字符串

Map<String, Object> maps = **new** HashMap<String,Object>();  
maps.**put**(**"grant\_code"**,**"XF9JKY0R"**);  
maps.**put**(**"is\_car\_config"**,0);  
maps.**put**(**"isCN"**,1);  
maps.**put**(**"vinCode"**,**"LGBG22E22AY081092"**);  
String jsonParam = JSON.*toJSONString*(maps);  
System.***out***.println(jsonParam);

输出：

{"grant\_code":"XF9JKY0R","is\_car\_config":0,"vinCode":"LGBG22E22AY081092","isCN":1}

# JackJson

需要包：

jackson-core-2.2.3.jar（核心jar包）

jackson-annotations-2.2.3.jar（该包提供Json注解支持）

jackson-databind-2.2.3.jar

<**dependency**>  
 <**groupId**>com.fasterxml.jackson.core</**groupId**>  
 <**artifactId**>jackson-core</**artifactId**>  
 <**version**>2.9.1</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>com.fasterxml.jackson.core</**groupId**>  
 <**artifactId**>jackson-annotations</**artifactId**>  
 <**version**>2.9.1</**version**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>com.fasterxml.jackson.core</**groupId**>  
 <**artifactId**>jackson-databind</**artifactId**>  
 <**version**>2.9.1</**version**>  
</**dependency**>

### 指定对象Class转成 json字符串

User user = **new** User();  
user.setName(**"小民"**);  
user.setEmail(**"xiaomin@sina.com"**);  
user.setAge(20);  
  
SimpleDateFormat dateformat = **new** SimpleDateFormat(**"yyyy-MM-dd"**);  
user.setBirthday(dateformat.parse(**"1996-10-01"**));  
  
*/\*\*  
 \* ObjectMapper是JSON操作的核心，Jackson的所有JSON操作都是在ObjectMapper中实现。  
 \* ObjectMapper有多个JSON序列化的方法，可以把JSON字符串保存File、OutputStream等不同的介质中。  
 \* writeValue(File arg0, Object arg1)把arg1转成json序列，并保存到arg0文件中。  
 \* writeValue(OutputStream arg0, Object arg1)把arg1转成json序列，并保存到arg0输出流中。  
 \* writeValueAsBytes(Object arg0)把arg0转成json序列，并把结果输出成字节数组。  
 \* writeValueAsString(Object arg0)把arg0转成json序列，并把结果输出成字符串。  
 \*/*ObjectMapper mapper = **new** ObjectMapper();  
  
*//User类转JSON  
//输出结果：{"name":"小民","age":20,"birthday":844099200000,"email":"xiaomin@sina.com"}*String json = mapper.writeValueAsString(user);  
System.***out***.println(json);

输出：

{"name":"小民","age":20,"birthday":844099200000,"email":"xiaomin@sina.com"}

### List集合转化成json字符串

List<User> users = **new** ArrayList<User>();  
users.**add**(user);  
String jsonlist = mapper.writeValueAsString(users);  
System.***out***.println(jsonlist);

输出：

[{"name":"小民","age":20,"birthday":844099200000,"email":"xiaomin@sina.com"}]

### Json字符串转化成指定Class类

String json = **"{\"name\":\"小民\",\"age\":20,\"birthday\":844099200000,\"email\":\"xiaomin@sina.com\"}"**;  
*/\*\*  
 \* ObjectMapper支持从byte[]、File、InputStream、字符串等数据的JSON反序列化。  
 \*/*ObjectMapper mapper = **new** ObjectMapper();  
User user = mapper.readValue(json, User.**class**);  
System.***out***.println(user);

输出：

User{name='小民aa', age=25, birthday=Tue Oct 01 00:00:00 CST 1996, email='xiaomin@sina.com'}

### Json字符串转化成集合List

方法一：

String jsonString=**"[{'id':'1'},{'id':'2'}]"**;  
ObjectMapper mapper = **new** ObjectMapper();  
JavaType javaType = mapper.getTypeFactory().constructParametricType(List.**class**, Bean.**class**);  
*//如果是Map类型 mapper.getTypeFactory().constructParametricType(HashMap.class,String.class, Bean.class);*List<Bean> lst = (List<Bean>)mapper.readValue(jsonString, javaType);

输出：

Student{name='s1', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s2', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s3', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s4', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s5', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s6', age=12, date=Thu Jun 09 20:38:37 CST 2016}

方法二：

String jsonString=**"[{'id':'1'},{'id':'2'}]"**;  
ObjectMapper mapper = **new** ObjectMapper();  
List<Bean> beanList = mapper.readValue(jsonString, **new** TypeReference<List<Bean>>() {});

输出：

Student{name='s1', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s2', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s3', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s4', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s5', age=12, date=Thu Jun 09 20:38:37 CST 2016}

Student{name='s6', age=12, date=Thu Jun 09 20:38:37 CST 2016}

# HttpClient

Httpclient 发送 get请求

Httpclient 发送 post请求

### Httpclient模拟get请求

需要包：

<dependency>  
 <groupId>org.apache.httpcomponents</groupId>  
 <artifactId>httpclient</artifactId>  
 <version>4.5.3</version>  
</dependency>

**private static** CloseableHttpClient *httpclient*;  
  
**static** {  
 PoolingHttpClientConnectionManager manager = **new** PoolingHttpClientConnectionManager();  
 manager.setMaxTotal(200); *//连接池最大并发连接数* manager.setDefaultMaxPerRoute(200);*//单路由最大并发数,路由是对maxTotal的细分  
 httpclient* = HttpClients.*custom*().setConnectionManager(manager).build();  
}  
  
*/\* ConnectionRequestTimeout httpclient使用连接池来管理连接，这个时间就是从连接池获取连接的超时时间，可以想象下数据库连接池  
 ConnectTimeout 建立连接最大时间  
 SocketTimeout 数据传输过程中数据包之间间隔的最大时间  
 HttpHost 代理  
 \*/***private static** RequestConfig *config* =RequestConfig.*copy*(RequestConfig.***DEFAULT***)  
 .setSocketTimeout(10000)  
 .setConnectTimeout(5000)  
 .setConnectionRequestTimeout(100).build();  
 *// .setProxy(new HttpHost("127.0.0.1",8888,"http")).build();***public static** String doGet(String url, Map<String, Object> header)  
 **throws** HttpClientException {  
 String ret = **""**;  
 HttpGet get = **new** HttpGet(url);  
 get.setConfig(*config*);  
 get.addHeader(HTTP.***CONTENT\_ENCODING***, **"UTF-8"**);  
 CloseableHttpResponse closeableHttpResponse = **null**;  
 **try** {  
 **if** (header != **null**) {  
 **for** (Map.Entry<String, Object> entry : header.**entrySet**()) {  
 get.setHeader(entry.**getKey**(), entry.**getValue**().toString());  
 }  
 }  
 closeableHttpResponse = *httpclient*.execute(get);  
 **if** (closeableHttpResponse.**getStatusLine**().**getStatusCode**() == 200) {  
 ret = EntityUtils.*toString*(closeableHttpResponse.**getEntity**(), **"UTF-8"**);  
 } **else** {  
 **throw new** HttpClientException(  
 **"System level error, Code=["** + closeableHttpResponse.**getStatusLine**().**getStatusCode**() + **"]."**);  
 }  
 } **catch** (ClientProtocolException e) {  
 **throw new** HttpClientException(**"HttpClient error,"** + e.getMessage());  
 } **catch** (IOException e) {  
 **throw new** HttpClientException(**"IO error,"** + e.getMessage());  
 } **finally** {  
 **if** (closeableHttpResponse != **null**) {  
 **try** {  
 closeableHttpResponse.**close**();  
 } **catch** (IOException e) {  
 }  
 }  
 }  
 **return** ret;  
}

**public static** String doGet(String url) **throws** HttpClientException {  
 **return** *doGet*(url,**null**);  
}

测试：

String result = **null**;  
String url =**"http://api.superepc.com/vtm/DataFunc?grant\_code=XF9JKY0R&is\_car\_config=0&isCN=1&vinCode=LGBG22E22AY081092"**;  
**try** {  
 result = *doGet*(**"http://123.58.251.183:8080/goods/UserServlet?method=loginMobile&loginname=test1&loginpass=test1"**);  
 String result1 = *doGet*(url);  
 System.***out***.println(result);  
} **catch** (HttpClientException e) {  
 e.printStackTrace();  
}

输出：

### Httpclient模拟post请求

**public static** String doPost(String url, Map<String, Object> params, Map<String, Object> header)  
 **throws** HttpClientException {  
 String ret = **""**;  
 HttpPost post = **new** HttpPost(url);  
 post.setConfig(*config*);  
 post.addHeader(HTTP.***CONTENT\_ENCODING***, **"UTF-8"**);  
 CloseableHttpResponse closeableHttpResponse = **null**;  
 HttpEntity postEntity = **null**;  
 **try** {  
 **if**(params!=**null**) {  
 List<NameValuePair> list = **new** ArrayList<NameValuePair>();  
 **for** (Map.Entry<String, Object> entry : params.**entrySet**()) {  
 list.**add**(**new** BasicNameValuePair(entry.**getKey**(), entry.**getValue**().toString()));  
 }  
 postEntity = **new** UrlEncodedFormEntity(list);  
 post.setEntity(postEntity);  
 }  
  
 **if** (header != **null**) {  
 **for** (Map.Entry<String, Object> entry : header.**entrySet**()) {  
 post.setHeader(entry.**getKey**(), entry.**getValue**().toString());  
 }  
 }  
 closeableHttpResponse = *httpclient*.execute(post);  
 **if** (closeableHttpResponse.**getStatusLine**().**getStatusCode**() == 200) {  
 ret = EntityUtils.*toString*(closeableHttpResponse.**getEntity**(), **"UTF-8"**);  
 } **else** {  
 **throw new** HttpClientException(  
 **"System level error, Code=["** + closeableHttpResponse.**getStatusLine**().**getStatusCode**() + **"]."**);  
 }  
 } **catch** (ClientProtocolException e) {  
 **throw new** HttpClientException(**"HttpClient error,"** + e.getMessage());  
 } **catch** (IOException e) {  
 **throw new** HttpClientException(**"IO error,"** + e.getMessage());  
 } **finally** {  
 **if**(postEntity!=**null**) {  
 **try** {  
 EntityUtils.*consume*(postEntity);  
 } **catch** (IOException e) {  
 }  
 }  
 **if** (closeableHttpResponse != **null**) {  
 **try** {  
 closeableHttpResponse.**close**();  
 } **catch** (IOException e) {  
 }  
 }  
 }  
 **return** ret;  
}

**public static** String doPost(String url, Map<String, Object> params) **throws** HttpClientException {  
 **return** *doPost*(url,params,**null**);  
}

### httpclient模拟post请求发送json数据

#### 传头部信息

**public static** String doPostByType(String contentType,String url,String jsonParam,Map<String,Object> header) **throws** HttpClientException {  
 String ret =**""**;  
 HttpPost post = **new** HttpPost(url);  
 post.setConfig(*config*);  
 post.addHeader(HTTP.***CONTENT\_ENCODING***,**"utf-8"**);  
 CloseableHttpResponse closeableHttpResponse =**null**;  
 StringEntity postEntity =**null**;  
 **try** {  
 **if** (jsonParam != **null**) {  
 postEntity = **new** StringEntity(jsonParam);  
 postEntity.setContentEncoding(**"UTF-8"**);  
 **if**(contentType.equals(**"json"**)){  
 postEntity.setContentType(**"application/json"**);  
 }**else if**(contentType.equals(**"soap+xml"**)){  
 postEntity.setContentType(**"application/soap+xml"**);  
 }  
  
 post.setEntity(postEntity);  
 }  
 **if** (header != **null**) {  
 **for** (Map.Entry<String, Object> entry : header.**entrySet**()) {  
 post.setHeader(entry.**getKey**(), entry.**getValue**().toString());  
 }  
 }  
 closeableHttpResponse = *httpclient*.execute(post);  
 **if** (closeableHttpResponse.**getStatusLine**().**getStatusCode**() == 200) {  
 ret = EntityUtils.*toString*(closeableHttpResponse.**getEntity**(), **"UTF-8"**);  
 } **else** {  
 **throw new** HttpClientException(**"System level error, Code=["** + closeableHttpResponse.**getStatusLine**().**getStatusCode**() + **"]."**);  
 }  
 } **catch** (ClientProtocolException e) {  
 **throw new** HttpClientException(**"HttpClient error,"** + e.getMessage());  
 }**catch** (IOException e) {  
 **throw new** HttpClientException(**"IO error,"** + e.getMessage());  
 } **finally** {  
 **try** {  
 **if**(postEntity!=**null**) {  
 EntityUtils.*consume*(postEntity);  
 }  
 **if**(closeableHttpResponse !=**null**){  
 closeableHttpResponse.**close**();  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 **return** ret;  
}

#### 不传头部信息

**public static** String doPostByType(String contentType,String url,String Param) **throws** HttpClientException {  
 **return** *doPostByType*(contentType,url,Param,**null**);  
}

测试类：

**public static void** doPostJsonTest(){  
 String url =**"http://123.58.251.183:8080/goods/json2"**;  
 Map<String, Object> map = **new** HashMap<String,Object>();  
 map.**put**(**"count"**, 2);  
 String jsonParam = JSON.*toJSONString*(map);  
 Map<String, Object> header = **new** HashMap<String,Object>();  
 header.**put**(**"token"**, **"61b3590090982a0185dda9d3bd793b46"**);  
 **try** {  
 String reString = HttpUtils.*doPostJson*(url, jsonParam, header);  
 System.***out***.println(reString);  
 } **catch** (HttpClientException e) {  
 e.printStackTrace();  
 }  
}

### httpclient模拟上传文件操作

#### 传头部信息

**public static** String doUpload(String url, File file,Map<String,Object> params,Map<String,Object> header){  
 String ret =**""**;  
 HttpPost post = **new** HttpPost(url);  
 post.setConfig(*config*);  
 post.addHeader(HTTP.***CONTENT\_ENCODING***,**"UTF-8"**);  
 CloseableHttpResponse response = **null**;  
 **try** {  
 MultipartEntityBuilder entityBuilder = MultipartEntityBuilder.*create*();  
 entityBuilder.addBinaryBody(**"file"**,file);  
 **if**(params!=**null**){  
 **for**(Map.Entry<String,Object> entry:params.**entrySet**()){  
 entityBuilder.addTextBody(entry.**getKey**(),entry.**getValue**().toString());  
 }  
 }  
 post.setEntity(entityBuilder.build());  
 **if**(header!=**null**){  
 **for**(Map.Entry<String,Object> entry:header.**entrySet**()){  
 post.addHeader(entry.**getKey**(),entry.**getValue**().toString());  
 }  
 }  
  
 response = *httpclient*.execute(post);  
 **if**(response.**getStatusLine**().**getStatusCode**()==200){  
 ret = EntityUtils.*toString*(response.**getEntity**(),**"utf-8"**);  
 }**else**{  
 **throw new** HttpClientException(**"System level error, Code=["** + response.**getStatusLine**().**getStatusCode**() + **"]."**);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **catch** (HttpClientException e) {  
 e.printStackTrace();  
 }**finally** {  
 **if**(response!=**null**){  
 **try** {  
 response.**close**();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 **return** ret;  
}

#### 不传头部信息

**public static** String doUpload(String url,File file){  
 **return** *doUpload*(url,file,**null**,**null**);  
}

测试类：

String url=**"http://123.58.251.183:8080/FileSever/upload.do"**;  
String filePath = System.*getProperty*(**"user.dir"**)+ File.***separator***+**"testdata"**+File.***separator***;  
File file = **new** File(filePath+**"test1.xml"**);  
String result = HttpUtils.*doUpload*(url,file);  
System.***out***.println(result);

### httpclient模拟下载文件操作

#### 传头部信息

**public static void** doDownload(String url, File descfile,Map<String,Object> header){  
 HttpPost post = **new** HttpPost(url);  
 post.setConfig(*config*);  
 post.addHeader(HTTP.***CONTENT\_ENCODING***,**"UTF-8"**);  
 CloseableHttpResponse response = **null**;  
 **try** {  
 **if**(header!=**null**){  
 **for**(Map.Entry<String,Object> entry:header.**entrySet**()){  
 post.addHeader(entry.**getKey**(),entry.**getValue**().toString());  
 }  
 }  
 response = *httpclient*.execute(post);  
 **if**(response.**getStatusLine**().**getStatusCode**()==200){  
 FileUtils.*copyToFile*(response.**getEntity**().**getContent**(),descfile);  
 }**else**{  
 **throw new** HttpClientException(**"System level error, Code=["** + response.**getStatusLine**().**getStatusCode**() + **"]."**);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **catch** (HttpClientException e) {  
 e.printStackTrace();  
 }**finally** {  
 **if**(response!=**null**){  
 **try** {  
 response.**close**();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

#### 不传头部信息

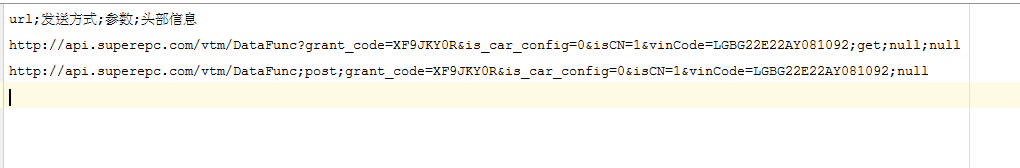
**public static void** doDownload(String url,File descfile){  
 *doDownload*(url,descfile,**null**);  
}

测试类：

String url=**"http:// 59.110.139.20:8080/FileSever/upload.do"**;  
String filePath = System.*getProperty*(**"user.dir"**)+ File.***separator***+**"testdata"**+File.***separator***;  
File file = **new** File(filePath+**"test1.xml"**);  
String result = HttpUtils.*doUpload*(url,file);  
System.***out***.println(result);  
  
JSONObject object =JSON.*parseObject*(result);  
String fileId = object.getString(**"fileId"**);  
**try** {  
 String result2 = HttpUtils.*doGet*(**"http://59.110.139.20:8080/FileSever/url.do?fileId="**+fileId);  
 JSONObject object1 = JSON.*parseObject*(result2);  
 String downloadUrl = object1.getString(**"url"**);  
 File file1 = **new** File(filePath+**"test2.xml"**);  
 HttpUtils.*doDownload*(downloadUrl,file1);  
} **catch** (HttpClientException e) {  
 e.printStackTrace();  
}

### Httpclient模拟从文件中获取的请求

Data.txt内容如下：



将字符串转化成Map,重载doPost方法：

**public static** String doPost(String url, String params,String regex) **throws** HttpClientException {  
 String[] param =params.split(regex);  
 Map<String,Object> map = **new** HashMap<String,Object>();  
 **for**(**int** i=0;i<param.**length**;i++){  
 String[] pp = param[i].split(**"="**);  
 **for**(String p:pp){  
 map.**put**(pp[0],pp[1]);  
 }  
 }  
 **return** *doPost*(url,map,**null**);  
}

测试类：

**public static void** main(String[] args) **throws** HttpClientException, IOException {  
 String filePath = System.*getProperty*(**"user.dir"**) + File.***separator*** + **"testdata"** + File.***separator***;  
 File file = **new** File(filePath + **"data.txt"**);  
 File file1 = **new** File(filePath + **"result.txt"**);  
 String ret = **""**;  
 List<String> lines = FileUtils.*readLines*(file, **"UTF-8"**);  
 **for** (**int** m = 0; m < lines.**size**(); m++) {  
 **if** (m > 0) {  
 String[] ss = lines.**get**(m).split(**";"**);  
 String url = ss[0];  
 String method = ss[1];  
 **if** (**"get"**.equals(method)) {  
 ret = HttpUtils.*doGet*(url);  
 *writeTofile*(file1, **"\nget ------>"** + ret);  
 System.***out***.println(**"get------>"** + ret);  
 } **else if** (**"post"**.equals(method)) {  
 ret = HttpUtils.*doPost*(url, ss[2], **"&"**);  
 *writeTofile*(file1, **"\npost ------>"** + ret);  
 System.***out***.println(**"post------>"** + ret);  
 }  
 }  
 }  
}

**public static void** writeTofile(File file,String str) **throws** IOException {  
 FileUtils.*writeStringToFile*(file,str,**"utf-8"**,**true**);  
  
}

输出：

get------>DataFunc([{"TID":"8519","retCode":"1"}])

post------>DataFunc([{"TID":"8519","retCode":"1"}])

# excel读取

### XSSFWorkbook、HSSFWorkbook读取Excel

#### 1、需要包poi、poi-ooxml

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi</artifactId>

<version>3.17</version>

</dependency>

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi-ooxml</artifactId>

<version>3.17</version>

</dependency>

#### 2、读取各版本excel

Student.class:

public class Student {

/\*\*

\* id

\*/

private Integer id;

/\*\*

\* 学号

\*/

private String no;

/\*\*

\* 姓名

\*/

private String name;

/\*\*

\* 学院

\*/

private String age;

/\*\*

\* 成绩

\*/

private float score;

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getNo() {

return no;

}

public void setNo(String no) {

this.no = no;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getAge() {

return age;

}

public void setAge(String age) {

this.age = age;

}

public float getScore() {

return score;

}

public void setScore(float score) {

this.score = score;

}

@Override

public String toString() {

return "Student [id=" + id + ", no=" + no + ", name=" + name + ", age=" + age + ", score=" + score + "]";

}

}

package com.test.excel;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.math.BigInteger;

import java.util.ArrayList;

import java.util.List;

import org.apache.commons.lang3.StringUtils;

import org.apache.poi.hssf.usermodel.HSSFCell;

import org.apache.poi.hssf.usermodel.HSSFRow;

import org.apache.poi.hssf.usermodel.HSSFSheet;

import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.apache.poi.xssf.usermodel.XSSFCell;

import org.apache.poi.xssf.usermodel.XSSFRow;

import org.apache.poi.xssf.usermodel.XSSFSheet;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;

public class ExcelUtils {

/\*\*

\* suffix of excel 2003

\*/

public static final String OFFICE\_EXCEL\_V2003\_SUFFIX = "xls";

/\*\*

\* suffix of excel 2007

\*/

public static final String OFFICE\_EXCEL\_V2007\_SUFFIX = "xlsx";

/\*\*

\* suffix of excel 2010

\*/

public static final String OFFICE\_EXCEL\_V2010\_SUFFIX = "xlsx";

public static final String EMPTY = "";

public static final String DOT = ".";

public static final String LIB\_PATH = "lib";

public static final String STUDENT\_INFO\_XLS\_PATH = LIB\_PATH + "/student\_info" + DOT + OFFICE\_EXCEL\_V2003\_SUFFIX;

public static final String STUDENT\_INFO\_XLSX\_PATH = LIB\_PATH + "/student\_info" + DOT + OFFICE\_EXCEL\_V2007\_SUFFIX;

public static final String NOT\_EXCEL\_FILE = " is Not a Excel file!";

public static final String PROCESSING = "Processing...";

/\*\*

\* Check which version of The Excel file is. Throw exception if Excel file path is illegal.

\*

\* @param path the Excel file

\* @return a list that contains Students from Excel.

\* @throws IOException

\*/

public static List<Student> readExcel(String path) throws IOException, IllegalArgumentException {

if (StringUtils.isBlank(path)) {

throw new IllegalArgumentException(path + " excel file path is either null or empty");

} else {

String suffiex = getSuffiex(path);

if(StringUtils.isBlank(suffiex)){

throw new IllegalArgumentException(path + " suffiex is either null or empty");

}

if (OFFICE\_EXCEL\_V2003\_SUFFIX.equals(suffiex)) {

return readXls(path);

} else if (OFFICE\_EXCEL\_V2007\_SUFFIX.equals(suffiex)||OFFICE\_EXCEL\_V2010\_SUFFIX.equals(suffiex)) {

return readXlsx(path);

} else {

throw new IllegalArgumentException(path + NOT\_EXCEL\_FILE);

}

}

}

/\*\*

\* Read the Excel 2017 or 2010

\* @param path the path of the excel file

\* @return

\* @throws IOException

\*/

public static List<Student> readXlsx(String path) throws IOException {

System.out.println(PROCESSING + path);

InputStream is = new FileInputStream(path);

XSSFWorkbook xssfWorkbook = new XSSFWorkbook(is);

Student student = null;

List<Student> list = new ArrayList<Student>();

// Read the Sheet

for (int numSheet = 0; numSheet < xssfWorkbook.getNumberOfSheets(); numSheet++) {

XSSFSheet xssfSheet = xssfWorkbook.getSheetAt(numSheet);

if (xssfSheet == null) {

continue;

}

// Read the Row

for (int rowNum = 1; rowNum <= xssfSheet.getLastRowNum(); rowNum++) {

XSSFRow xssfRow = xssfSheet.getRow(rowNum);

if (xssfRow != null) {

student = new Student();

XSSFCell id = xssfRow.getCell(0);

XSSFCell no = xssfRow.getCell(1);

XSSFCell name = xssfRow.getCell(2);

XSSFCell age = xssfRow.getCell(3);

XSSFCell score = xssfRow.getCell(4);

student.setId(Float.valueOf(getValue(id)).intValue());

student.setNo(getValue(no));

student.setName(getValue(name));

student.setAge(getValue(age));

student.setScore(Float.valueOf(getValue(score)));

list.add(student);

}

}

}

return list;

}

/\*\*

\* Read the Excel 2003

\* @param path the path of the Excel

\* @return

\* @throws IOException

\*/

public static List<Student> readXls(String path) throws IOException {

System.out.println(PROCESSING + path);

InputStream is = new FileInputStream(path);

HSSFWorkbook hssfWorkbook = new HSSFWorkbook(is);

Student student = null;

List<Student> list = new ArrayList<Student>();

// Read the Sheet

for (int numSheet = 0; numSheet < hssfWorkbook.getNumberOfSheets(); numSheet++) {

HSSFSheet hssfSheet = hssfWorkbook.getSheetAt(numSheet);

if (hssfSheet == null) {

continue;

}

// Read the Row

for (int rowNum = 1; rowNum <= hssfSheet.getLastRowNum(); rowNum++) {

HSSFRow hssfRow = hssfSheet.getRow(rowNum);

if (hssfRow != null) {

student = new Student();

HSSFCell no = hssfRow.getCell(0);

HSSFCell name = hssfRow.getCell(1);

HSSFCell age = hssfRow.getCell(2);

HSSFCell score = hssfRow.getCell(3);

student.setNo(getValue(no));

student.setName(getValue(name));

student.setAge(getValue(age));

student.setScore(Float.valueOf(getValue(score)));

list.add(student);

}

}

}

return list;

}

@SuppressWarnings("static-access")

private static String getValue(XSSFCell xssfRow) {

if (xssfRow.getCellType() == xssfRow.CELL\_TYPE\_BOOLEAN) {

return String.valueOf(xssfRow.getBooleanCellValue());

} else if (xssfRow.getCellType() == xssfRow.CELL\_TYPE\_NUMERIC) {

return String.valueOf(xssfRow.getNumericCellValue());

} else {

return String.valueOf(xssfRow.getStringCellValue());

}

}

@SuppressWarnings("static-access")

private static String getValue(HSSFCell hssfCell) {

if (hssfCell.getCellType() == hssfCell.CELL\_TYPE\_BOOLEAN) {

return String.valueOf(hssfCell.getBooleanCellValue());

} else if (hssfCell.getCellType() == hssfCell.CELL\_TYPE\_NUMERIC) {

return String.valueOf(hssfCell.getNumericCellValue());

} else {

return String.valueOf(hssfCell.getStringCellValue());

}

}

public static String getSuffiex(String path) {

if(StringUtils.isBlank(path)){

return EMPTY;

}

int index = path.lastIndexOf(DOT);

if (index == -1) {

return EMPTY;

}

return path.substring(index + 1, path.length());

}

}

测试：

public static void main(String[] args) throws IOException {  
 try {  
 String path =System.*getProperty*("user.dir")+File.*separator*+"testdata"+File.*separator*+"test.xlsx";  
 List<Student> list = *readExcel*(path);  
 System.*out*.println(list);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

### Excel4J读取Excel

#### 1、需要包com.github.crab2died.Excel4J

<dependency>

<groupId>com.github.crab2died</groupId>

<artifactId>Excel4J</artifactId>

<version>2.1.2</version>

<exclusions>

<exclusion>

<groupId>org.apache.poi</groupId>

<artifactId>poi</artifactId>

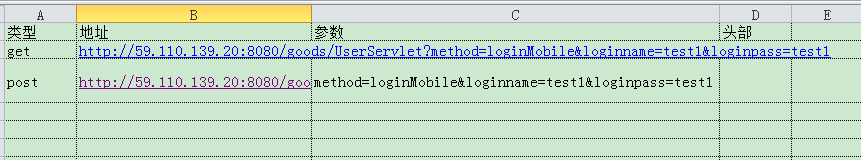
</exclusion>

</exclusions>

</dependency>

#### 2、读取excel

Excel内容如下：



import com.github.crab2died.annotation.ExcelField;  
  
  
public class TestCase {  
   
 @ExcelField(title = "类型")  
 private String type;  
   
 @ExcelField(title = "地址")  
 private String url;  
   
 @ExcelField(title = "参数")  
 private String params;  
   
 @ExcelField(title = "头部",order=2)  
 private String header;  
   
 @ExcelField(title = "测试结果")  
 //@ExcelField(title = "测试结果", order=1, writeConverter = ResultWirteConverible.class)  
 private String result;  
   
  
 public String getResult() {  
 return result;  
 }  
  
 public void setResult(String result) {  
 this.result = result;  
 }  
  
 public String getType() {  
 return type;  
 }  
  
 public void setType(String type) {  
 this.type = type;  
 }  
  
 public String getUrl() {  
 return url;  
 }  
  
 public void setUrl(String url) {  
 this.url = url;  
 }  
  
 public String getParams() {  
 return params;  
 }  
  
 public void setParams(String params) {  
 this.params = params;  
 }  
  
 public String getHeader() {  
 return header;  
 }  
  
 public void setHeader(String header) {  
 this.header = header;  
 }  
  
 @Override  
 public String toString() {  
 return "TestCase [type=" + type + ", url=" + url + ", params=" + params + ", header=" + header + ", result="  
 + result + "]";  
 }  
  
}

public static void main(String[] args) {  
 String path =System.*getProperty*("user.dir")+File.*separator*+"testdata"+File.*separator*+"apitest.xlsx";  
 String path2 =System.*getProperty*("user.dir")+File.*separator*+"testdata"+File.*separator*+"apitest2.xlsx";  
 String filepath =System.*getProperty*("user.dir")+File.*separator*+"testdata"+File.*separator*;  
 try {  
 List<TestCase> list = ExcelUtils.*getInstance*().readExcel2Objects(path,TestCase.class);  
 for (TestCase testCase : list) {  
 System.*out*.println(testCase);  
 testCase.setResult("ok");  
 }  
 //输出到excel   
 ExcelUtils.*getInstance*().exportObjects2Excel(list, TestCase.class, filepath+"TestCase.xlsx");

}

# Email 发送

需要包

<dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-email</artifactId>  
 <version>1.5</version>  
</dependency>

### HtmlEmail 简单邮件发送

emailConfig.properties文件内容：

mail.host=smtp.163.com  
mail.username=15021771575@163.com  
mail.password=suncui0806  
mail.title=mail test   
mail.touser=929194573@qq.com,Nouyang@cccis.com  
mail.ccaddress=sundy0806@126.com

EmailUtils类中，邮件设置封装：

public static HtmlEmail emailSet(String title,String emailMsg,String fromEmail,String ps,String[] toEmail) throws EmailException {  
 HtmlEmail email = new HtmlEmail();  
 email.setCharset("UTF-8");  
 email.setHostName(EmailParams.*host*);  
 email.setAuthentication(fromEmail,ps);  
 email.setFrom(fromEmail);//发件人  
 for(String touser:toEmail){  
 email.addTo(touser);//收件人  
 }  
 email.setSubject(title); //邮件标题  
 email.setHtmlMsg(emailMsg);//邮件内容  
 return email;  
}

EmailUtils类中，邮件发送类：

//发送一个简单的邮件，只有内容。  
public static void sendEmail(String title,String emailMsg,String fromEmail,String ps,String[] toEmail) throws EmailException {  
 HtmlEmail email =*emailSet*(title,emailMsg,fromEmail,ps,toEmail);  
 email.send();  
}

测试发送类：

public static void main(String[] args) throws EmailException, MalformedURLException {  
 String filePath = System.*getProperty*("user.dir") + File.*separator* + "testdata" + File.*separator* + "apitest.xlsx";  
 EmailUtils.*sendEmail*(EmailParams.*title*, "这是一个测试邮件", EmailParams.*username*, EmailParams.*password*, EmailParams.*tousers*);

}

### HtmlEmail网络上的文件发送

public static void sendNetworkPicEmail(String title,String emailMsg,String fromEmail,String ps,String[] toEmail) throws EmailException, MalformedURLException {  
 HtmlEmail email =*emailSet*(title,emailMsg,fromEmail,ps,toEmail);  
 EmailAttachment emailAttachment = new EmailAttachment();  
 emailAttachment.setURL(new URL("http://www.apache.org/images/asf\_logo\_wide.gif"));  
 emailAttachment.setDisposition(EmailAttachment.*ATTACHMENT*);  
 emailAttachment.setDescription("Apache logo 发送");  
 emailAttachment.setName("Apache logo");  
 email.attach(emailAttachment);  
 email.send();  
}

测试发送类：

EmailUtils.sendNetworkPicEmail(EmailParams.title, "这是一个测试邮件", EmailParams.username, EmailParams.password, EmailParams.tousers);

### HtmlEmail发送HTML格式的邮件

public static void sendHTMLEmail(String title,String fromEmail,String ps,String[] toEmail) throws EmailException, MalformedURLException {  
 HtmlEmail email =*emailSet*(title,fromEmail,ps,toEmail);  
 URL url = new URL("http://www.apache.org/images/asf\_logo\_wide.gif");  
 String cid =email.embed(url, "Apache logo");  
 email.setHtmlMsg("<html>The apache logo - <img src=\"cid:"+cid+"\"></html>");  
  
 email.send();  
}

测试类：

EmailUtils.sendHTMLEmail(EmailParams.title,EmailParams.username, EmailParams.password, EmailParams.tousers);

### HtmlEmail 发送、抄送 有附件的邮件

邮件bean:

public class MailInfo {  
 //发件人  
 private String fromAddress;  
 //收件人  
 private List<String> toAdress;  
 //抄送人  
 private List<String> ccAdress;  
 // 附件信息  
 private List<EmailAttachment> attachments;  
 //主题  
 private String subject;  
 //邮件内容  
 private String content;  
  
 public String getFromAddress() {  
 return fromAddress;  
 }  
  
 public void setFromAddress(String fromAddress) {  
 this.fromAddress = fromAddress;  
 }  
  
 public List<String> getToAdress() {  
 return toAdress;  
 }  
  
 public void setToAdress(List<String> toAdress) {  
 this.toAdress = toAdress;  
 }  
  
 public List<String> getCcAdress() {  
 return ccAdress;  
 }  
  
 public void setCcAdress(List<String> ccAdress) {  
 this.ccAdress = ccAdress;  
 }  
  
 public List<EmailAttachment> getAttachments() {  
 return attachments;  
 }  
  
 public void setAttachments(List<EmailAttachment> attachments) {  
 this.attachments = attachments;  
 }  
  
 public String getSubject() {  
 return subject;  
 }  
  
 public void setSubject(String subject) {  
 this.subject = subject;  
 }  
  
 public String getContent() {  
 return content;  
 }  
  
 public void setContent(String content) {  
 this.content = content;  
 }  
}

MailUtil类：

public class MailUtil {  
 public static void sendMail(MailInfo mailInfo){  
 HtmlEmail email = new HtmlEmail();  
 try {  
 // 配置信息  
 email.setHostName(EmailParams.*host*);  
 email.setFrom(EmailParams.*username*);  
 email.setAuthentication(EmailParams.*username*,EmailParams.*password*);  
 email.setCharset("UTF-8");  
 email.setSubject(mailInfo.getSubject());  
 email.setHtmlMsg(mailInfo.getContent());  
  
 //添加附件  
 List<EmailAttachment> attachmentList = mailInfo.getAttachments();  
 if(null != attachmentList && attachmentList.size()>0){  
 for(EmailAttachment attachment:attachmentList){  
 email.attach(attachment);  
 }  
 }  
 //添加收件人  
 List<String> toAddressList = mailInfo.getToAdress();  
 if(null != toAddressList && toAddressList.size()>0){  
 for(String toAddress:toAddressList){  
 email.addTo(toAddress);  
 }  
 }  
 //添加抄送人  
 List<String> ccAddressList = mailInfo.getCcAdress();  
 if(null != ccAddressList && ccAddressList.size()>0){  
 for(String ccAddress:ccAddressList){  
 email.addCc(ccAddress);  
 }  
 }  
 email.send();  
 System.*out*.println("邮件发送成功！");  
 } catch (EmailException e) {  
 e.printStackTrace();  
 }  
 }  
}

测试类：

public class MailSendTest {  
 public static void main(String[] args) {  
 MailInfo mailInfo = new MailInfo();  
 mailInfo.setSubject("测试主题");  
 mailInfo.setFromAddress(EmailParams.*username*);  
 mailInfo.setContent("内容：<h1>test,测试</h1>");  
  
 //收件人  
 List<String> toList = new ArrayList<String>();  
 String[] tos = EmailParams.*tousers*;  
 for(String to:tos){  
 toList.add(to);  
 }  
 mailInfo.setToAdress(toList);  
  
 //抄送人  
 List<String> ccList = new ArrayList<String>();  
 String[] ccs = EmailParams.*ccaddress*;  
 for(String cc: ccs){  
 ccList.add(cc);  
 }  
 mailInfo.setCcAdress(ccList);  
  
 //添加附件  
 List<EmailAttachment> attachmentList = new ArrayList<EmailAttachment>();  
 String filePath ="./testdata/";  
 String attachmentName="apitest.xlsx,test.xml";  
 String[] attachs = attachmentName.split(",");  
 for(int i=0;i<attachs.length;i++){  
 EmailAttachment att = new EmailAttachment();  
 att.setPath(filePath+attachs[i]);  
 att.setName(attachs[i]);  
 attachmentList.add(att);  
 }  
 mailInfo.setAttachments(attachmentList);  
 MailUtil.*sendMail*(mailInfo);  
 }  
}

# Aviator 表达式

需要包：

<dependency>  
 <groupId>com.googlecode.aviator</groupId>  
 <artifactId>aviator</artifactId>  
 <version>3.3.0</version>  
</dependency>

### 执行表达式

Aviator的数值类型仅支持Long和Double, 任何整数都将转换成Long, 任何浮点数都将转换为Double, 包括用户传入的变量数值

Long result = (Long) AviatorEvaluator.*execute*("22+1+100");  
System.*out*.println(result);//123

### 变量使用

Aviator的String是任何用单引号或者双引号括起来的字符序列

String myname = "Lily";  
Map<String,Object> map = new HashMap<String,Object>();  
map.put("myname",myname);  
  
//方法一：  
String result1 = (String) AviatorEvaluator.*execute*(" 'hello '+ myname",map);  
System.*out*.println(result1);  
//方法二：  
Object result11 = AviatorEvaluator.*exec*("'hello '+ myname",myname);  
System.*out*.println(result11.toString());  
  
String result2 = (String) AviatorEvaluator.*execute*("\"a\'b\'c\"");//输出：a'b'c  
String result3 = (String) AviatorEvaluator.*execute*("'a\"b\"c'");//输出：a"b"c  
String result4 = (String) AviatorEvaluator.*execute*("'a\"b\"c' + 3");//输出：a"b"c3  
String result5 = (String) AviatorEvaluator.*execute*("'a\"b\"c' + hello");//输出：a"b"cnull

### 函数使用

通过string.substring('habcello', 1, 3)获取字符串'ab', 然后通过函数string.contains判断e是否在'abc'中。

System.*out*.println(AviatorEvaluator.*execute*("string.substring('habcello',1,3)"));//ab

System.*out*.println(AviatorEvaluator.*execute*("string.contains(\"abc\",\"ab\")"));//true

System.*out*.println(AviatorEvaluator.*execute*("string.contains(\"abc\",string.substring('habcello',1,3))"));

System.*out*.println(AviatorEvaluator.*execute*("string\_to\_date('2018-09-11',\"yyyymmdd\")"));//Tue Jan 09 00:00:00 CST 2018

System.*out*.println(AviatorEvaluator.*exec*("date\_to\_string(dd,\"yyyy-mm-dd\")",new Date()));//2017-18-30

### 自定义函数

public class AviatorUtil extends AbstractFunction {  
  
 @Override  
 public AviatorObject call(Map<String, Object> env, AviatorObject arg1, AviatorObject arg2) {  
 Number num1 = FunctionUtils.*getNumberValue*(arg1,env);  
 Number num2 = FunctionUtils.*getNumberValue*(arg2,env);  
 //乘法  
 AviatorObject o = new AviatorDouble(num1.doubleValue() \* num2.doubleValue());  
 return o;  
 }  
 public String getName() {  
 return "multiplication";  
 }

}

测试类：

public static void main(String[] args) {  
 AviatorEvaluator.*addFunction*(new AviatorUtil());  
 System.*out*.println(AviatorEvaluator.*execute*("multiplication(5,4)"));//输出：20  
 System.*out*.println(AviatorEvaluator.*execute*("multiplication(10,multiplication(5,4))"));//输出：200  
}

### 编译表达式

通过compile方法可以将表达式编译成Expression的中间对象,

当要执行表达式的时候传入map1并调用Expression的execute方法即可

String expression = "a/(b+c)>10";  
Expression exp = AviatorEvaluator.*compile*(expression);  
Map<String,Object> map1 = new HashMap<String,Object>();  
map1.put("a",100);  
map1.put("b",0);  
map1.put("c",5);  
Boolean bool = (Boolean) exp.execute(map1);  
System.*out*.println(bool);//true

### 访问数组和集合

List<String> list = new ArrayList<String>();  
 list.add("hello");  
 list.add("world");  
  
 int[] nums = new int[5];  
 for(int i=0;i<nums.length;i++){  
 nums[i]=i;  
 }  
  
 Map<String,Object> map2 = new HashMap<String,Object>();  
 // map2.put("date",DateUtils.date2Str(new Date(),DateUtils.DATE\_FORMAT\_SEC));  
 map2.put("date",AviatorEvaluator.*exec*("date\_to\_string(dd,\"yyyy-mm-dd\")",new Date()));  
 System.*out*.println(AviatorEvaluator.*execute*("string\_to\_date('2018-09-11',\"yyyymmdd\")"));//Tue Jan 09 00:00:00 CST 2018  
 System.*out*.println(AviatorEvaluator.*exec*("date\_to\_string(dd,\"yyyy-mm-dd\")",new Date()));//2017-18-30  
  
 // map2.put("date1", "dd");  
 Map<String,Object> map3 = new HashMap<String,Object>();  
 map3.put("list",list);  
 map3.put("nums",nums);  
 map3.put("map2",map2);  
  
 System.*out*.println(AviatorEvaluator.*execute*("list[0]+' '+list[1]",map3));//hello world  
 System.*out*.println(AviatorEvaluator.*execute*("'nums[0]+nums[1]+nums[2]='+(nums[0]+nums[1]+nums[2])",map3));//nums[0]+nums[1]+nums[2]=3  
System.*out*.println(AviatorEvaluator.*execute*("'当前时间为：'+map2.date",map3));//当前时间为：2017-12-30 19:56:42  
}

# JSONPath解析json

需要包：

<dependency>  
 <groupId>com.jayway.jsonpath</groupId>  
 <artifactId>json-path</artifactId>  
 <version>2.4.0</version>  
</dependency>

### 使用说明

1、JSONPath是xpath在json的应用

2、JSONPath 是参照xpath表达式来解析xml文档的方式，json数据结构通常是匿名的并且不一定需要有根元素。

3、JSONPath 用一个抽象的名字$来表示最外层对象

4、JSONPath 允许使用通配符 \* 表示所以的子元素名和数组索引

### JSONPath表达式语法

JSONPath 表达式可以使用.符号解析json：

$.store.book[0].title

或者使用[]符号

$['store']['book'][0]['title']

Json文件内容如下：

{ "store": {  
 "book": [   
 { "category": "reference",  
 "author": "Nigel Rees",  
 "title": "Sayings of the Century",  
 "price": 8.95  
 },  
 { "category": "fiction",  
 "author": "Evelyn Waugh",  
 "title": "Sword of Honour",  
 "price": 12.99,  
 "isbn": "0-553-21311-3"  
 }  
 ],  
 "bicycle": {  
 "color": "red",  
 "price": 19.95  
 }  
 }  
}

首先，读取json文件，使用commons.io的 FileUtils的readFileToString方法：

String path =System.*getProperty*("user.dir")+File.*separator*+"testdata"+File.*separator*+"test.json";

String jsonString = FileUtils.*readFileToString*(new File(path),"utf-8");

ReadContext context = JsonPath.*parse*(json);

其次，输出book[1]的author值。有两种方法：

方法一：

JsonPath.*read*(json,"$.store.book[1].author");

或

context.read("$.store.book[1].author");

输出：Evelyn Waugh

方法二：

JsonPath.*read*(json,"$['store']['book'][1]['author']");

context.read("$['store']['book'][1]['author']");

输出：Evelyn Waugh

### XPATH 和 JSONPath获取元素的方法比较

[]在xpath表达式总是从前面的路径来操作数组，索引是从1开始。

使用JOSNPath的[]操作符操作一个对象或者数组，索引是从0开始。

|  |  |  |
| --- | --- | --- |
| **XPath** | **JSONPath** | **结果** |
| /store/book/author | $.store.book[\*].author | 书点所有书的作者 |
| //author | $..author | 所有的作者 |
| /store/\* | $.store.\* | store的所有元素。所有的bookst和bicycle |
| /store//price | $.store..price | store里面所有东西的price |
| //book[3] | $..book[2] | 第三个书 |
| //book[last()] | $..book[(@.length-1)] | 最后一本书 |
| //book[position()<3] | $..book[0,1]  $..book[:2] | 前面的两本书。 |
| //book[isbn] | $..book[?(@.isbn)] | 过滤出所有的包含isbn的书。 |
| //book[price<10] | $..book[?(@.price<10)] | 过滤出价格低于10的书。 |
| //\* | $..\* | 所有元素。 |

//输出book[\*]中category == 'reference'的book  
List<Object> categorys = context.read("$.store.book[?(@.category == 'reference')]");  
for(Object st: categorys){  
 System.*out*.println(st.toString());  
 //{category=reference, author=Nigel Rees, title=Sayings of the Century, price=8.95}  
}  
  
//输出book[\*]中price>10的book  
List<Object> prices = context.read("$.store.book[?(@.price>10)]");  
for(Object p:prices){  
 System.*out*.println(p.toString());  
 //{category=fiction, author=Evelyn Waugh, title=Sword of Honour, price=12.99, isbn=0-553-21311-3}  
}  
//bicycle[\*]中含有color元素的bicycle  
List<Object> color = context.read("$.store.bicycle[?(@.color)]");  
for(Object is :color){  
 System.*out*.println(is.toString());//{color=red, price=19.95}  
}  
  
//输出该json中所有price的值  
List<Object> pp = context.read("$..price");  
for(Object p :pp){  
 System.*out*.println(p.toString()); //8.95 12.99 19.95  
}  
  
List<String> authors = context.read("$.store.book[\*].author");  
for (String str : authors) {  
 System.*out*.println(str);//Nigel Rees Evelyn Waugh  
}

# C3p0数据库操作

需要导入对应数据库应用的包：

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>6.0.6</version>  
</dependency>  
<dependency>  
 <groupId>c3p0</groupId>  
 <artifactId>c3p0</artifactId>  
 <version>0.9.1.2</version>  
</dependency>  
<dependency>  
 <groupId>commons-dbutils</groupId>  
 <artifactId>commons-dbutils</artifactId>  
 <version>1.7</version>  
</dependency>

使用c3p0框架，对数据库进行配置，c3p0-config.xml

<?xml version="1.0" encoding="UTF-8"?>   
<c3p0-config>   
 <!-- 默认配置，只可以出现一次 -->   
 <default-config>   
 <!-- 配置JDBC 四个基本属性 -->  
 <property name ="driverClass">com.mysql.jdbc.Driver</property >  
 <property name ="jdbcUrl">jdbc:mysql://192.168.100.63:3308/SweetCustomSystem</property >  
 <property name ="user">root</property>  
 <property name ="password">123123</property>  
 <!--当连接池中的连接耗尽的时候c3p0一次同时获取的连接数。Default: 3 -->  
 <property name="acquireIncrement">3</property>  
　　 <!-- 初始化数据库连接池时连接的数量 -->  
 <property name="initialPoolSize">5</property>  
 <!-- 数据库连接池中的最小的数据库连接数 -->  
 <property name="minPoolSize">2</property>  
 <!-- 数据库连接池中的最大的数据库连接数 -->  
 <property name="maxPoolSize">10</property>  
 </default-config>   
</c3p0-config>

创建JDBCUtils类

import com.mchange.v2.c3p0.ComboPooledDataSource;  
  
import javax.sql.DataSource;  
import java.sql.Connection;  
import java.sql.PreparedStatement;  
import java.sql.ResultSet;  
import java.sql.SQLException;  
  
public class JDBCUtils {  
  
 // 获得c3p0连接池对象  
 private static ComboPooledDataSource *ds* = new ComboPooledDataSource();  
  
 */\*\*  
 \* 获得数据库连接对象  
 \*  
 \** ***@return*** *\** ***@throws*** *SQLException  
 \*/* public static Connection getConnection() throws SQLException {  
 return *ds*.getConnection();  
 }  
  
 */\*\*  
 \* 获得c3p0连接池对象  
 \*  
 \** ***@return*** *\*/* public static DataSource getDataSource() {  
 return *ds*;  
 }  
  
 public static void closeDB(Connection conn, PreparedStatement ps, ResultSet rs) {  
 try {  
 if (rs != null) {  
 rs.close();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (ps != null) {  
 ps.close();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (conn != null) {  
 conn.close();  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 // 释放资源，将数据库连接还给数据库连接池  
 public static void closeDB(PreparedStatement ps, Connection conn) {  
 try {  
 if (ps != null) {  
 ps.close();  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if (conn != null) {  
 conn.close();  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("----------" + JDBCUtils.*getDataSource*());  
  
 }  
}

[使用QueryRunner类对数据表的增、删、查](http://blog.csdn.net/a911711054/article/details/77719685)

import bean.Users;  
import org.apache.commons.dbutils.QueryRunner;  
import org.apache.commons.dbutils.handlers.\*;  
import utils.JDBCUtils;  
import java.sql.SQLException;  
import java.util.List;  
import java.util.Map;  
  
public class UsersDao {  
 public static List<Users> getAllUsers() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select \* from users;";  
 List<Users> list = (List<Users>) queryRunner.query(sql, new BeanListHandler(Users.class));  
 return list;  
 }  
  
  
 public static int insert(Object[] params) throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "insert into users (userId,address,email,passWd,phone,type,userName) values (?,?,?,?,?,?,?)";  
 int result = queryRunner.update(sql,params);  
 return result;  
 }  
  
 public static int update(Object[] params) throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "update users set address=?,email=? where userId=?";  
 int result = queryRunner.update(sql,params);  
 return result;  
 }  
  
 public static int delete(Object obj) throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "delete from users where userId=?";  
  
 int result = queryRunner.update(sql,129);  
 return result;  
 }

*/\*\*  
 \* 结果集第一种处理方法，ArrayHandler  
 \* 将结果第一行存储到对象数组中Object[]  
 \** ***@throws*** *SQLException  
 \* \*/*public static void searchToArray() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select \* from users";  
 Object [] objects = queryRunner.query(sql, new ArrayHandler());  
 for(Object obj:objects){  
 System.*out*.println(obj.toString());  
 }  
 }  
*/\*\*  
 \* 结果集的第二种处理方法，ArrayListHandler  
 \* 将结果集的每一行，封装到对象数组中，出现很多对象数组  
 \* 对象数组存储到List集合  
 \** ***@throws*** *SQLException  
 \* \*/*public static void searchToArrayList() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select \* from users";  
 List<Object[]> list = queryRunner.query(sql,new ArrayListHandler());  
 for(Object[] objects:list){  
 for(Object obj:objects) {  
 System.*out*.print(obj.toString());  
 }  
 System.*out*.println();  
 }  
 }  
*/\*\*  
 \* 结果集的第三种处理方法BeanHandler  
 \* 将结果集的第一行数据，封装为JavaBean对象  
 \* 注意：被封装成数据到JavaBean对象，Sort类必须有空参构造  
 \** ***@throws*** *SQLException  
 \* \*/*public static void searchToBean() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select \* from users";  
 Users users = queryRunner.query(sql,new BeanHandler<Users>(Users.class));  
 System.*out*.println(users);  
}  
  
  
*/\*\*  
 \* 结果集第四种处理方法，BeanListHandler  
 \* 将数据结果集的每一行数据，封装为JavaBean对象  
 \* 多个JavaBean对象封装到List集合中  
 \** ***@throws*** *SQLException  
 \* \*/*public static void searchToBeanList() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select \* from users";  
 List<Users> usersList = queryRunner.query(sql,new BeanListHandler<Users>(Users.class));  
 for(Users user:usersList){  
 System.*out*.println(user.toString());  
 }  
 }  
  
*/\*\*  
 \* 结果集第五种处理方法，ColumnListHandler  
 \* 结果集，指定列的数据，存储到List集合中  
 \* List<Object> 每个列数据类型不同  
 \** ***@throws*** *SQLException  
 \* \*/*public static void searchToColumnList() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select \* from users";  
 //调用方法query，传递结果集实现类ColumnListHandler  
 //实现类构造方法，使用字符串的列名  
 List<Object> list = queryRunner.query(sql,new ColumnListHandler<Object>("userName"));  
 for(Object user:list){  
 System.*out*.println(user.toString());  
 }  
}  
  
*/\*\*  
 \* 结果集第六种处理方法、  
 \* 对于查询后，只有一个结果  
 \** ***@throws*** *SQLException  
 \* \*/*public static void searchToSingleRusult() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select count(\*) from users";  
 long obj = queryRunner.query(sql,new ScalarHandler<Long>());  
 System.*out*.println(obj);  
}  
  
*/\*\*  
 \* 结果集的第七种处理方式，MapHandler  
 \* 将结果集第一行数据，封装到Map集合中  
 \* Map<键，值> 键：列名 值：这列数据  
 \** ***@throws*** *SQLException  
 \* \*/* public static void searchToMap() throws SQLException {  
 QueryRunner queryRunner = new QueryRunner(JDBCUtils.*getDataSource*());  
 String sql = "select \* from users";  
 List<Map<String, Object>> mapList = queryRunner.query(sql,new MapListHandler());  
 for(Map<String,Object> map:mapList){  
 for(Map.Entry<String,Object> entry : map.entrySet()){  
 System.*out*.print(entry.getKey()+":"+entry.getValue());  
 }  
 System.*out*.println();  
 }  
  
 }

测试类：

public static void main(String[] args) throws SQLException {  
 Object[] params = {129,"上海市徐汇区龙田路","aaa@167.com","123123","15012221111","2","sundy"};  
 Object[] params1 = {"上海市徐汇区龙田路11111","aaa@167.com",129};  
  
 List<Users> list = UsersDao.*getAllUsers*();  
 for(Users user:list){  
 // System.out.println(user.toString());  
 }  
  
 //int result = UsersDao.insert(params);  
 // int result = UsersDao.update(params1);  
// int result = UsersDao.delete(129);  
//  
// if(result>0){  
// System.out.println(result+"-----------操作成功");  
// }else{  
// System.out.println("操作失败");  
// }  
  
 // UsersDao.searchToMap();  
 // UsersDao.searchToArray();  
 // UsersDao.searchToArrayList();  
 // UsersDao.searchToBean();  
 // UsersDao.searchToBeanList();  
 UsersDao.*searchToColumnList*();  
 UsersDao.*searchToSingleRusult*();  
  
 }

# Moco

### Moco原理简介

Moco会根据一些配置，启动一个真正的HTTP服务（会监听本地的某个端口）。当发起请求满足一个条件时，它就给回复一个应答。Moco的底层没有依赖于像Servlet这样的重型框架，而是基于一个叫Netty网络应用框架直接编写的，这样一来，绕过了复杂的应用服务器，所以，它的速度是极快的

### Moco独立运行所需环境

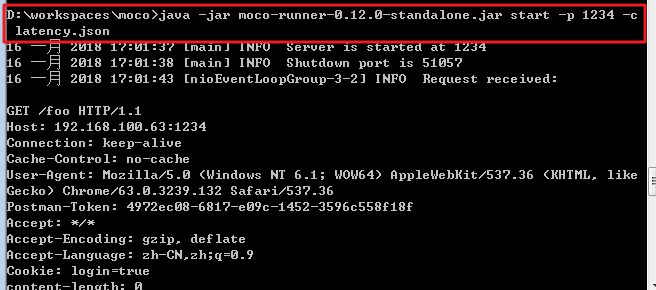
* 配置java环境,安装好JDK
* moco-runner-0.12.0-standalone.jar

### Moco启动http服务

电脑中运行cmd,进入到Moco的jar包所在路径下，执行命令：

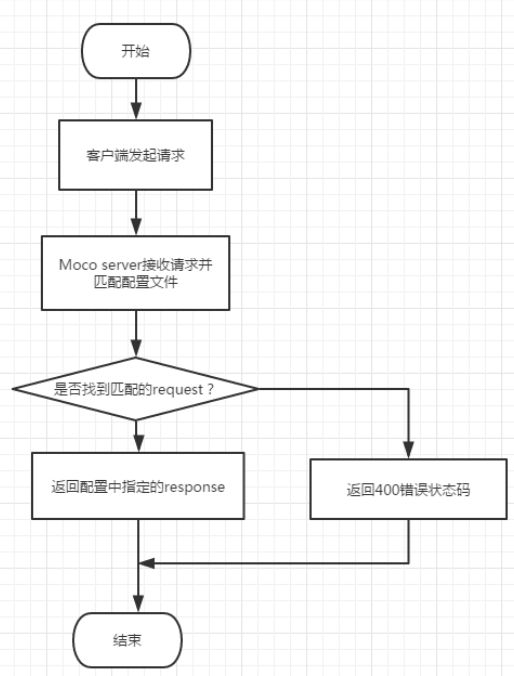
java -jar <path-to-moco-runner> http -p <monitor-port> -c < configuration -file>

* <path-to-moco-runner>：moco-runner-0.11.0-standalone.jar包的路径
* <monitor-port>：http服务监听的端口
* <configuration -file>：配置文件路径



上面是在本地启动了一个http server，其中监听端口是1234，配置文件是latency.json

### Moco HTTP(s) 配置文件的工作原理



### Moco 的约定

#### 约定接口的uri

Json文件内容：

[

{

"request" : {

"uri" : "/foo"

},

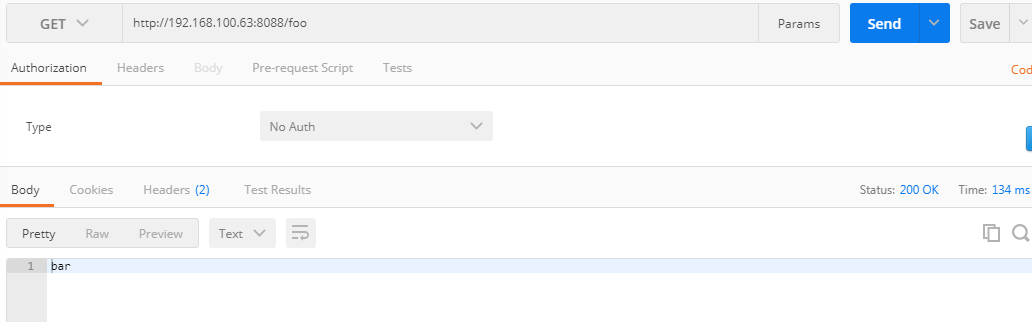
"response" : {

"text" : "bar"

}

}

]



#### 约定请求forms、方法、参数

Json文件内容：

[

{

"request": {

"method": "get",

"uri": "/foo",

"queries": {

"param": "blash"

}

},

"response": {

"text": "get--------"

}

},

{

"request": {

"method": "post",

"uri": "/foo",

"forms":

{

"param":"blash"

}

},

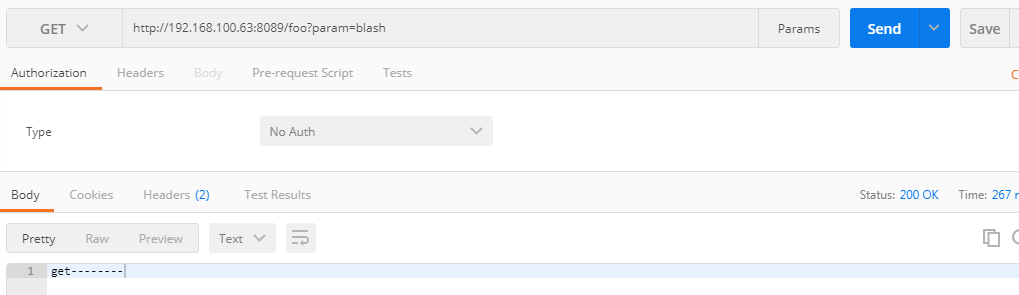
"response": {

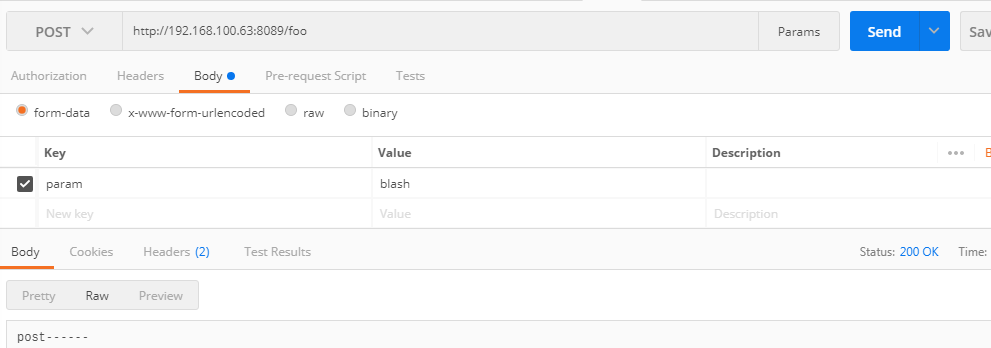
"text": "post------"

}

}

]





#### 约定请求头部、状态码、cookies

Json文件内容：

[

{

"request" :{

"method" : "post",

"headers" : {

"content-type" : "application/json"

}

},

"response" :{

"status" : 200,

"text" : "bar",

"cookies" :{

"login" : "true"

},

"headers" :{

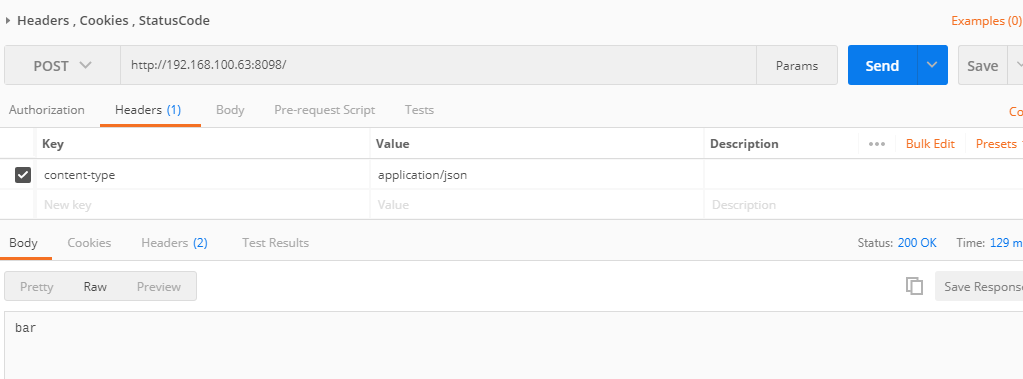
"content-type" : "application/json"

}

}

}

]



#### 约定以指定json作为响应body

Json文件内容：

[

{

"request": {

"method" : "get",

"uri": "/carTaskForRule"

},

"response": {

"file" : "1219\_sample\_v0.7.txt"

}

},

{

"request" :{

"uri": "/getJson",

"method" : "post",

"headers" : {

"content-type" : "application/json"

}

},

"response" :{

"status" : 200,

"headers" :{

"content-type" : "application/json"

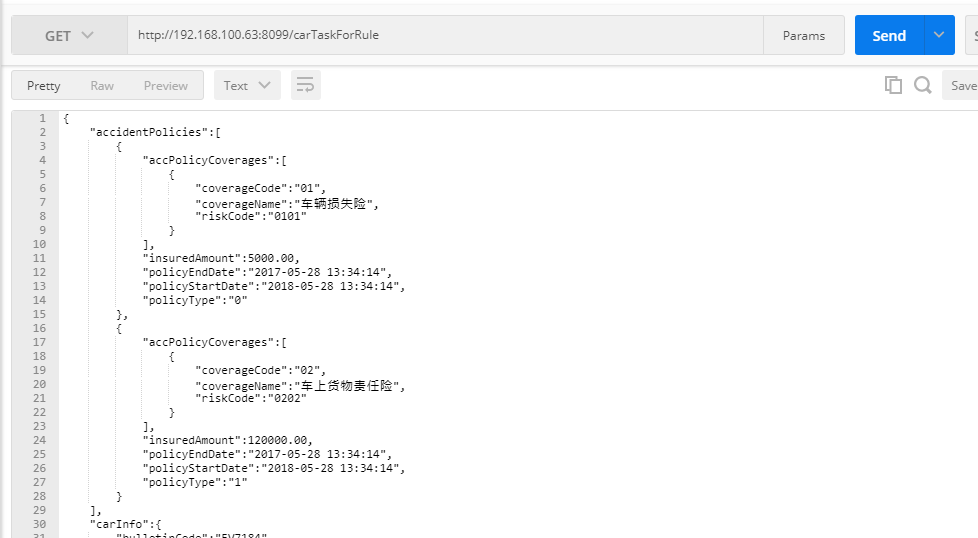
},

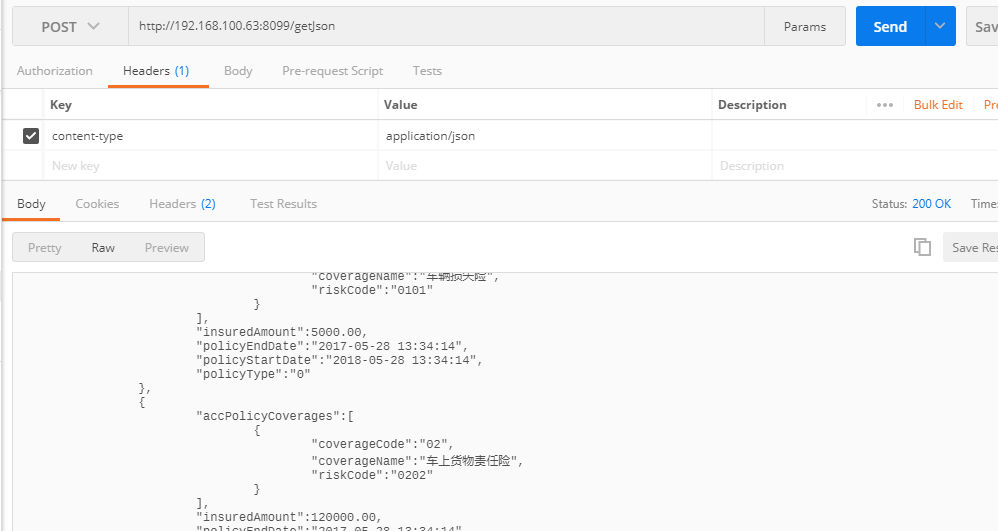
"file" : "1219\_sample\_v0.7.txt"

}

}

]





#### 约定指定时间后响应

Json文件内容：

[

{

"description":"request注释行，可以在添加对接口的描述",

"request" :{

"uri": "/foo"

},

"response" :{

"latency": {

"duration": 10,

"unit": "second"

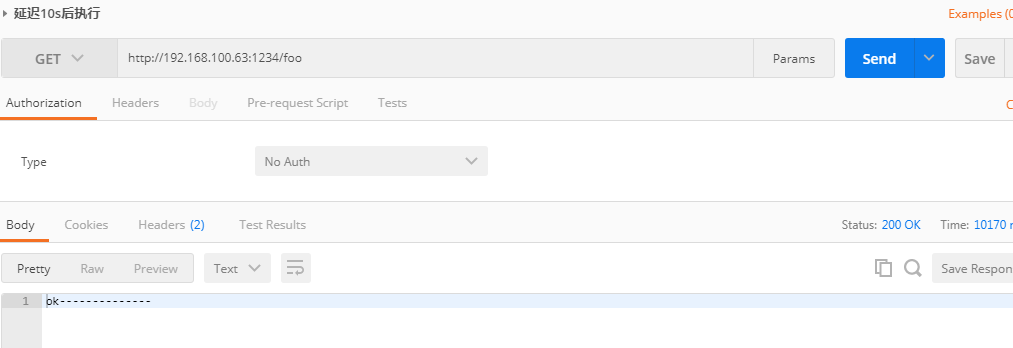
},

"text" : "ok--------------"

}

}

]



#### 约定重定向

Json文件内容：

[

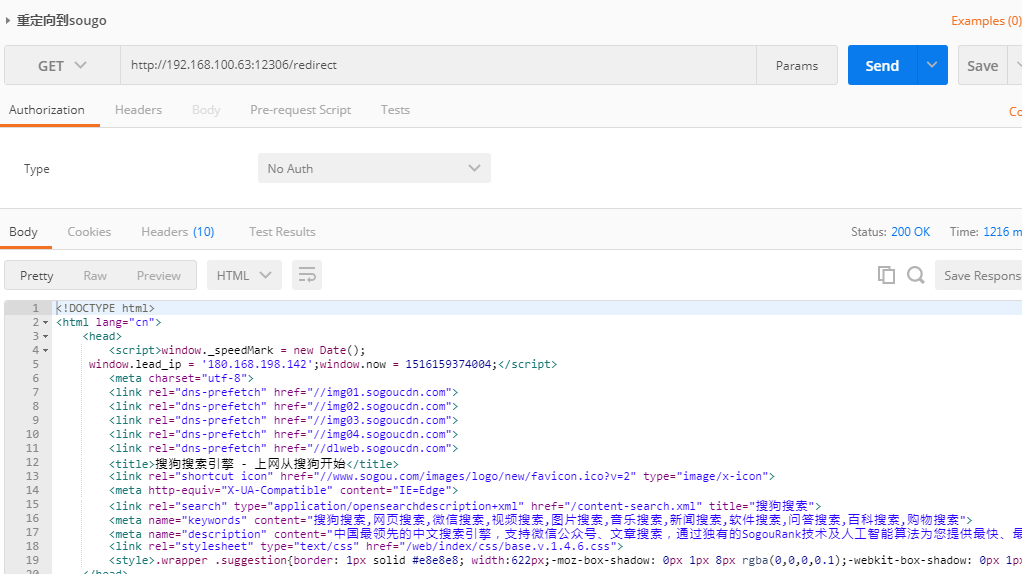
{

"request" : { "uri" : "/redirect" },

"redirectTo" : "http://www.sogou.com"

}

]



# Mockito的使用

# Maven常用插件

## 1、全局属性配置

Pom.xml中配置：

<properties>

   <project.build.name>tools</project.build.name>

<!-- 文件拷贝时的编码 -->

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<!-- 编译时的编码 -->

<maven.compiler.encoding>UTF-8</maven.compiler.encoding>

<package.environment>product</package.environment>

</properties>

## 2、编译插件

Pom.xml中配置：

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 <encoding>${project.build.sourceEncoding}</encoding>  
 </configuration>  
</plugin>

source： 源代码编译版本；

target： 目标平台编译版本；

encoding： 字符集编码；

## ****2、设置资源文件的编码方式****

Pom.xml文件中配置：

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-resources-plugin</artifactId>  
 <version>2.4.3</version>  
 <executions>  
 <execution>  
 <phase>compile</phase>  
 </execution>  
 </executions>  
 <configuration>  
 <encoding>${project.build.sourceEncoding}</encoding>  
 </configuration>  
</plugin>

xml、properties文件都是资源文件，编码的时候遇到中文要进行转码。<encoding>${project.build.sourceEncoding}</encoding>

## 3、将项目打成jar包

Pom.xml文件中配置：

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-jar-plugin</artifactId>  
 <version>3.0.2</version>  
 <!-- The configuration of the plugin -->  
 <configuration>  
 <!-- Configuration of the archiver -->  
 <archive>  
 <!-- 生成的jar中，不要包含pom.xml和pom.properties这两个文件 -->  
 <addMavenDescriptor>false</addMavenDescriptor>  
 <!-- Manifest specific configuration -->  
 <manifest>  
 <!-- 是否要把第三方jar放到manifest的classpath中 -->  
 <addClasspath>true</addClasspath>  
 <!-- 生成的manifest中classpath的前缀，因为要把第三方jar放到lib目录下，所以classpath的前缀是lib/ -->  
 <classpathPrefix>lib/</classpathPrefix>  
 <!-- 应用的main class -->  
 <mainClass>com.sc.test.ApiTest</mainClass>  
 </manifest>  
 </archive>  
 </configuration>  
</plugin>

在将项目打成jar包时，有时会需要将项目打成可以直接运行的jar包，因此就需要将项目依赖的jar包也打入jar包中。就需要用到maven-assembly-plugin插件。

## 4、打压缩包

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <version>3.1.0</version>  
 <!-- The configuration of the plugin -->  
 <configuration>  
 <!-- Specifies the configuration file of the assembly plugin -->  
 <descriptors>  
 <descriptor>src/main/assembly/assembly.xml</descriptor>  
 </descriptors>  
 </configuration>  
 <executions>  
 <execution>  
 <id>make-assembly</id>  
 <phase>package</phase>  
 <goals>  
 <goal>single</goal>  
 </goals>  
 </execution>  
 </executions>  
</plugin>

此外，还需要在src/main下新建assembly/assembly.xml文件。内容如下：

<assembly>  
 <id>bin</id>  
 <includeBaseDirectory>false</includeBaseDirectory>  
 <!-- 最终打包成一个用于发布的zip文件 -->  
 <formats>  
 <format>zip</format>  
 </formats>  
  
 <!-- Adds dependencies to zip package under lib directory -->  
 <dependencySets>  
 <dependencySet>  
 <!--  
 不使用项目的artifact，第三方jar不要解压，打包进zip文件的lib目录  
 -->  
 <useProjectArtifact>false</useProjectArtifact>  
 <!--<outputDirectory>lib</outputDirectory>-->  
 <unpack>false</unpack>  
 </dependencySet>  
 </dependencySets>  
  
 <fileSets>  
 <!-- 把项目相关的说明文件，打包进zip文件的根目录 -->  
 <!--<fileSet>-->  
 <!--<directory>${project.basedir}</directory>-->  
 <!--<outputDirectory>/</outputDirectory>-->  
 <!--</fileSet>-->  
  
 <!-- 把项目的配置文件，打包进zip文件的config目录 -->  
 <fileSet>  
 <directory>${deploy.dir}/classes/</directory>  
 <outputDirectory>/conf</outputDirectory>  
 <includes>  
 <include>\*.xml</include>  
 <include>\*.properties</include>  
 </includes>  
 </fileSet>  
 <!-- 把项目自己编译出来的jar文件，打包进zip文件的根目录 -->  
 <fileSet>  
 <directory>${project.build.directory}</directory>  
 <outputDirectory></outputDirectory>  
 <includes>  
 <include>\*.jar</include>  
 </includes>  
 </fileSet>  
 </fileSets>  
</assembly>

## 5、将项目打成war包

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.1.1</version>

<configuration>

<warName>ROOT</warName>

<webResources>

<resource>

<directory>src/main/resources/config/${package.environment}</directory>

<targetPath>WEB-INF/classes/config</targetPath>

<filtering>true</filtering>

</resource>

</webResources>

</configuration>

</plugin>

warName ：指定了打好包的名字；

webResources 下 resource 下 directory ：配置真正使用的资源文件存放的位置，通常这个位置不是一个标准 Web 项目应该存放资源文件的位置；

webResources 下 resource 下 targetPath：将上一步 directory 中配置的文件目录下的文件都拷贝到这个目录下；‘

webResources 下 resource 下 filtering：开启打包之前将 directory 下的文件拷贝（覆盖）到 targetPath 下这种配置。

## 6、copy依赖jar包

<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-dependency-plugin</artifactId>  
 <executions>  
 <execution>  
 <id>copy</id>  
 <phase>package</phase>  
 <goals>  
 <goal>copy-dependencies</goal>  
 </goals>  
 <configuration>  
 <includeScope>runtime</includeScope>  
 <outputDirectory>target/lib/</outputDirectory>  
 </configuration>  
 </execution>  
 </executions>  
</plugin>

# Get和Post的区别

## 相同点

都是向服务器发送请求：Get是向服务器发送索取数据的一种请求，而Post是向服务器提交数据的一种请求

## 不同点

### 1、请求的数据

GET请求，请求的数据会附加在URL之后，以?分割URL和传输数据，多个参数用&连接。URL的编码格式采用的是ASCII编码，而不是uniclde，即是说所有的非ASCII字符都要编码之后再传输。

POST请求：POST请求会把请求的数据放置在HTTP请求包的包体中。上面的item=bandsaw就是实际的传输数据。

因此，GET请求的数据会暴露在地址栏中，而POST请求则不会。

### 2、用途

GET用于从服务器端获取数据，包括静态资源(HTML|JS|CSS|Image等等)、动态数据展示(列表数据、详情数据等等)。

POST用于向服务器提交数据，比如增删改数据，提交一个表单新建一个用户、或修改一个用户等。

### 3、缓存

GET时默认可以复用前面的请求数据作为缓存结果返回，此时以完整的URL作为缓存数据的KEY。所以有时候为了强制每次请求都是新数据，可以在URL后面加上一个随机参数Math.random或时间戳new Date().getTime()、或版本号，比如abc.com?a=1&rnd=0.123987之类的。这也是目前一些静态资源后面加一个很长的版本号的原因，jquery-min.js?v=13877770表示一个版本，当页面引用jquery-min.js?v=13877771时浏览器必然会重新去服务器请求这个资源。jQuery.ajax方法，如果cache=false，则会在GET请求参数中附加”\_={timestamp}”来禁用缓存。

POST一般则不会被这些缓存因素影响。

### 4、安全性

默认对于nginx的access log，会自动记录get或post的完整URL，包括其中带的参数。

对于POST来说，请求的报文却不会被记录，这些对于敏感数据来说，POST更安全一些。

### 5、Url长度限制

GET是通过URL提交数据，因此GET可提交的数据量就跟URL所能达到的最大长度有直接关系。

HTTP协议本身对GET和POST都没有对长度的限制，而对于URL长度上的限制是浏览器受服务器的配置限制或者内存的大小。

### 6、幂等

GET幂等，POST不幂等

幂等是指同一个请求方法执行多次和仅执行一次的效果完全相同。

引入幂等主要是为了处理同一个请求重复发送的情况，比如在请求响应前失去连接，如果方法是幂等的，就可以放心地重发一次请求。这也是浏览器在后退/刷新时遇到POST会给用户提示的原因：POST语义不是幂等的，重复请求可能会带来意想不到的后果。

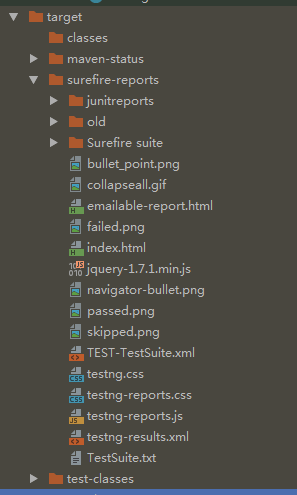
比如在微博这个场景里，GET的语义会被用在「看看我的Timeline上最新的20条微博」这样的场景，而POST的语义会被用在「发微博、评论、点赞」这样的场景中

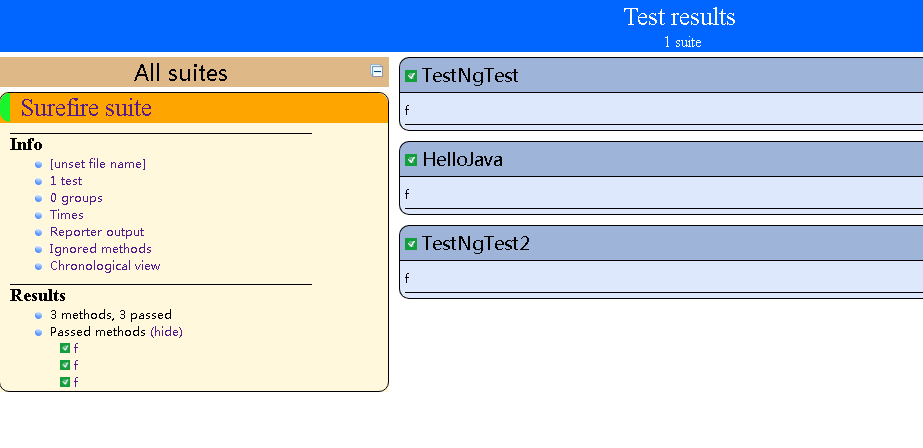
# testNg

## 使用testng自带插件生成测试报告

pom.xml文件中设置如下时，测试报告默认生成在target\surefire-reports目录下。

<dependencies>  
 <dependency>  
 <groupId>org.testng</groupId>  
 <artifactId>testng</artifactId>  
 <version>6.13.1</version>  
 <scope>test</scope>  
 </dependency>  
</dependencies>  
<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-surefire-plugin</artifactId>  
 <version>2.20.1</version>  
 <configuration>  
 <includes>  
 <include>HelloJava.java</include>  
 <include>%regex[.\*Test.\*]</include>  
 </includes>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>





## 引用reportng包生成测试报告

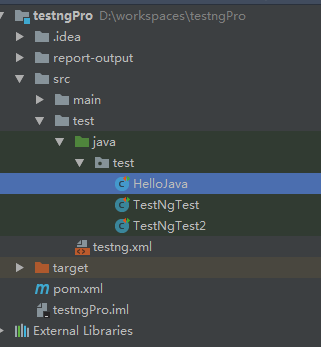
Pom.xml文件中设置如下时，生成测试报告在项目根目录的report-output下：

<!-- 添加插件 关联testNg.xml -->  
<plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-surefire-plugin</artifactId>  
 <version>${maven-surefire-plugin.version}</version>  
 <configuration>  
 <suiteXmlFiles>  
 <suiteXmlFile>${suiteXmlFile}</suiteXmlFile>  
 </suiteXmlFiles>  
 <testFailureIgnore>true</testFailureIgnore>  
 <properties>  
 <property>  
 <name>usedefaultlisteners</name>  
 <value>false</value>  
 </property>  
 <property>  
 <name>listener</name>  
 <value>org.uncommons.reportng.HTMLReporter, org.uncommons.reportng.JUnitXMLReporter</value>  
 </property>  
 </properties>  
 <reportsDirectory>report-output/</reportsDirectory>  
 <workingDirectory>target/</workingDirectory>  
 </configuration>  
</plugin>

新增testng.xml在src/test/testng.xml下

<?xml version="1.0" encoding="utf-8" ?>  
<suite name="testproj" parallel="false">  
 <test name="testDemo1">  
 <packages>  
 <package name="test"/>  
 </packages>  
 <!-- <classes> -->  
 <!-- <class name="com.testproj.Demo.TestDemo1"></class> -->  
 <!-- <class name="com.testproj.Demo.TestDemo1"></class> -->  
 <!-- <class name="com.testproj.Demo.TestDemo1"></class> -->  
 <!-- </classes> -->  
 </test>  
</suite>

三个测试类：





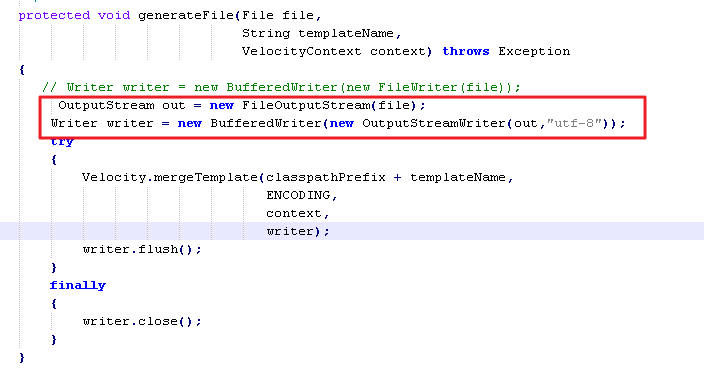
## Reportng包生成测试报告乱码解决方案

1.下载reportng源码 https://github.com/dwdyer/reportng.git

2.修改AbstractReporter.java并保存，记得一定要导入OutputStream类。

OutputStream out = new FileOutputStream(file);

Writer writer = new BufferedWriter(new OutputStreamWriter(out,"utf-8"));



3.因为该项目打包用的是ant，所以需要配置ant环境

1).下载ant源码包 http://ant.apache.org/bindownload.cgi

2).解压到安装位置,如D:\apache-ant-1.9.6

3).进入计算机--属性--高级系统配置--高级--环境变量--修改环境变量

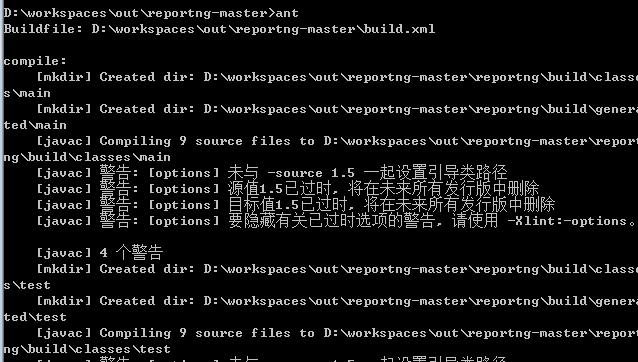
系统环境变量中新增ANT\_HOME，值为：D:\apache-ant-1.9.6，修改path变量，

追加值：%ANT\_HOME%\bin

4).验证ant， cmd：ant –version



5).进入到项目的根目录，并执行ant



6).进入到项目的dist目录下，查看新打的包为：reportng-1.1.5.jar，依赖包为：velocity-dep-1.4.jar

7).找到当前使用reportng要生成报告的项目，依赖包所在位置，

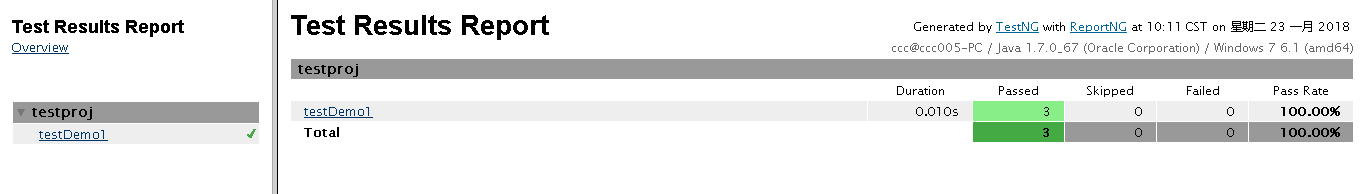
一般在：C:\Users\用户名\.m2\repository\org\uncommons\reportng下

在该目录下新增1.1.5的文件夹，并将上述生成的包reportng-1.1.5.jar复制到该文件夹下。

8).在当前使用reportng要生成报告的项目的pom.xml文件中修改reportng的版本为1.1.5，并引用其依赖包：velocity

<dependency>  
 <groupId>org.uncommons</groupId>  
 <artifactId>reportng</artifactId>  
 <version>1.1.5</version>  
 <scope>test</scope>  
</dependency>  
<!--reportng1.1.5依赖包-->  
<dependency>  
 <groupId>org.apache.velocity</groupId>  
 <artifactId>velocity</artifactId>  
 <version>1.7</version>  
</dependency>

9).重新执行maven test命令。查看report-output下生成的测试报告显示



## 将Reportng生成的测试报告压缩成zip包

需要导入依赖包：

<!--用于zip文件压缩-->  
<dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-compress</artifactId>  
 <version>1.15</version>  
</dependency>

以下类为zip压缩包工具类且通用。可用于多层级目录文件压缩。

package com.sc.util;  
  
import java.io.BufferedInputStream;   
import java.io.BufferedOutputStream;   
import java.io.File;   
import java.io.FileInputStream;   
import java.io.FileOutputStream;   
import java.io.IOException;   
import java.io.InputStream;   
import java.io.OutputStream;   
import java.util.ArrayList;   
import java.util.List;  
import org.apache.commons.compress.archivers.ArchiveEntry;   
import org.apache.commons.compress.archivers.zip.Zip64Mode;   
import org.apache.commons.compress.archivers.zip.ZipArchiveEntry;   
import org.apache.commons.compress.archivers.zip.ZipArchiveInputStream;   
import org.apache.commons.compress.archivers.zip.ZipArchiveOutputStream;   
   
public class ZipUtil {   
   
 public static void main(String[] args) {   
   
// String dir = "D:\\data\\datanew2\\apitest\\report-output";  
// String zippath = "D:\\data\\datanew2\\apitest\\report-output\\test2.zip";  
// ZipUtil.zip(dir, zippath);  
 ZipUtil.*unzip*("d:\\API测试报告\_2018-01-26\_10\_22\_40.zip","d:\\API");  
   
 System.*out*.println("success!");   
 }   
 */\*\*   
 \* zip压缩文件   
 \** ***@param*** *dir   
 \** ***@param*** *zippath   
 \*/* public static void zip(String dir ,String zippath){   
 List<String> paths = *getFiles*(dir);   
 *compressFilesZip*(paths.toArray(new String[paths.size()]),zippath,dir);   
 }   
 */\*\*   
 \* 递归取到当前目录所有文件   
 \** ***@param*** *dir   
 \** ***@return*** *\*/* public static List<String> getFiles(String dir){   
 List<String> lstFiles = null;   
 if(lstFiles == null){   
 lstFiles = new ArrayList<String>();   
 }   
 File file = new File(dir);   
 File [] files = file.listFiles();   
 for(File f : files){   
 if(f.isDirectory()){   
 lstFiles.add(f.getAbsolutePath());   
 lstFiles.addAll(*getFiles*(f.getAbsolutePath()));   
 }else{   
 String str =f.getAbsolutePath();   
 lstFiles.add(str);   
 }   
 }   
 return lstFiles;   
 }   
   
 */\*\*   
 \* 文件名处理   
 \** ***@param*** *dir   
 \** ***@param*** *path   
 \** ***@return*** *\*/* public static String getFilePathName(String dir,String path){   
 String p = path.replace(dir+File.*separator*, "");   
 p = p.replace("\\", "/");   
 return p;   
 }   
 */\*\*   
 \* 把文件压缩成zip格式   
 \** ***@param*** *files 需要压缩的文件   
 \** ***@param*** *zipFilePath 压缩后的zip文件路径 ,如"D:/test/aa.zip";   
 \*/* public static void compressFilesZip(String[] files,String zipFilePath,String dir) {   
 if(files == null || files.length <= 0) {   
 return ;   
 }   
 ZipArchiveOutputStream zaos = null;   
 try {   
 File zipFile = new File(zipFilePath);   
 zaos = new ZipArchiveOutputStream(zipFile);   
 zaos.setUseZip64(Zip64Mode.*AsNeeded*);   
 //将每个文件用ZipArchiveEntry封装   
 //再用ZipArchiveOutputStream写到压缩文件中   
 for(String strfile : files) {   
 File file = new File(strfile);   
 if(file != null) {   
 String name = *getFilePathName*(dir,strfile);   
 ZipArchiveEntry zipArchiveEntry = new ZipArchiveEntry(file,name);   
 zaos.putArchiveEntry(zipArchiveEntry);   
 if(file.isDirectory()){   
 continue;   
 }   
 InputStream is = null;   
 try {   
 is = new BufferedInputStream(new FileInputStream(file));   
 byte[] buffer = new byte[1024 ];   
 int len = -1;   
 while((len = is.read(buffer)) != -1) {   
 //把缓冲区的字节写入到ZipArchiveEntry   
 zaos.write(buffer, 0, len);   
 }   
 zaos.closeArchiveEntry();   
 }catch(Exception e) {   
 throw new RuntimeException(e);   
 }finally {   
 if(is != null)   
 is.close();   
 }   
   
 }   
 }   
 zaos.finish();   
 }catch(Exception e){   
 throw new RuntimeException(e);   
 }finally {   
 try {   
 if(zaos != null) {   
 zaos.close();   
 }   
 } catch (IOException e) {   
 e.printStackTrace();  
 throw new RuntimeException(e);   
 }   
 }   
   
 }   
   
   
 */\*\*   
 \* 把zip文件解压到指定的文件夹   
 \** ***@param*** *zipFilePath zip文件路径, 如 "D:/test/aa.zip"   
 \** ***@param*** *saveFileDir 解压后的文件存放路径, 如"D:/test/" ()   
 \*/* public static void unzip(String zipFilePath, String saveFileDir) {   
 if(!saveFileDir.endsWith("\\") && !saveFileDir.endsWith("/") ){   
 saveFileDir += File.*separator*;   
 }   
 File dir = new File(saveFileDir);   
 if(!dir.exists()){   
 dir.mkdirs();   
 }   
 File file = new File(zipFilePath);   
 if (file.exists()) {   
 InputStream is = null;   
 ZipArchiveInputStream zais = null;   
 try {   
 is = new FileInputStream(file);   
 zais = new ZipArchiveInputStream(is);   
 ArchiveEntry archiveEntry = null;   
 while ((archiveEntry = zais.getNextEntry()) != null) {   
 // 获取文件名   
 String entryFileName = archiveEntry.getName();   
 // 构造解压出来的文件存放路径   
 String entryFilePath = saveFileDir + entryFileName;   
 OutputStream os = null;   
 try {   
 // 把解压出来的文件写到指定路径   
 File entryFile = new File(entryFilePath);   
 if(entryFileName.endsWith("/")){   
 entryFile.mkdirs();   
 }else{   
 os = new BufferedOutputStream(new FileOutputStream(   
 entryFile));   
 byte[] buffer = new byte[1024 ];   
 int len = -1;   
 while((len = zais.read(buffer)) != -1) {   
 os.write(buffer, 0, len);   
 }   
 }   
 } catch (IOException e) {   
 throw new IOException(e);   
 } finally {   
 if (os != null) {   
 os.flush();   
 os.close();   
 }   
 }   
   
 }   
 } catch (Exception e) {   
 throw new RuntimeException(e);   
 } finally {   
 try {   
 if (zais != null) {   
 zais.close();   
 }   
 if (is != null) {   
 is.close();   
 }   
 } catch (IOException e) {   
 throw new RuntimeException(e);   
 }   
 }   
 }   
 }   
}

## 监听Reportng生成的测试报告结果并发送邮件

1. 先获取测试报告所在路径。
2. 将上述路径下的文件压缩为zip包
3. 将zip包作为附件，发送邮件
4. 为了避免每次打zip包时将上一次生成的zip包重复打入当前包中，可手动清空测试报告目录下的文件。

import com.sc.email.EmailUtils;  
import com.sc.util.DateUtil;  
import com.sc.util.ZipUtil;  
import org.apache.commons.io.FileUtils;  
import org.testng.IReporter;  
import org.testng.ISuite;  
import org.testng.xml.XmlSuite;  
import java.io.File;  
import java.util.List;  
  
public class TestNGEmailReport implements IReporter {  
 @Override  
 public void generateReport(List<XmlSuite> list, List<ISuite> list1, String s) {  
 System.*out*.println("report--------------");  
 String path =System.*getProperty*("user.dir")+File.*separator*+"report-output";  
 //文件名过长，可能会导致邮件中附件的名称显示为“未命名.bin”  
 String zip\_path = path + File.*separator* +"API测试结果\_"+ DateUtil.*getCurrentDate*()+".zip";  
 ZipUtil.*zip*(path,zip\_path);  
  
 try {  
 EmailUtils.*sendEmailsWithAttachments*("api测试结果","具体测试结果，请查看附件，谢谢！",zip\_path);  
 FileUtils.*cleanDirectory*(new File(path));  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
}

如果直接执行testng.xml文件，则只需要在testng.xml文件中设置如下

<?xml version="1.0" encoding="utf-8" ?>  
<suite name="testproj" parallel="methods">  
 <test name="testDemo1">  
 <listeners>  
 <listener class-name="org.uncommons.reportng.HTMLReporter" />  
 <listener class-name="org.uncommons.reportng.JUnitXMLReporter" />  
 <listener class-name="com.sc.retry.OverrideIAnnotationTransformer"></listener>  
 <listener class-name="com.sc.test.TestNGEmailReport"></listener>  
 </listeners>  
 <classes>  
 <class name="com.sc.test.ApitestNgTest"></class>  
 </classes>  
 </test>  
</suite>

如果执行pom.xml文件，则需要在pom文件中对测试报告目录进行资源copy设置

<!--拷贝report-output文件夹到target目录下-->  
<resource>  
 <directory>report-output</directory>  
 <targetPath>${project.build.directory}/report-output</targetPath>  
</resource>

## TestNg 注解

### 依赖性测试dependsOnMethods

"test1()"只有在"test2()"运行成功的情况下才会执行，否则"test1()"将跳过测试

@Test(dependsOnMethods = "test2")  
public void test1(){  
 System.*out*.println("this is test1 ---------------");  
}  
@Test  
public void test2(){  
 System.*out*.println("this is test2 ---------------");  
}

### 异常测试expectedExceptions

“异常测试”是指从单元测试中抛出的异常

public class TestNgException {  
 @Test(expectedExceptions = ArithmeticException.class)  
 public void testException(){  
 int i = 1/0;  
 System.*out*.println(i);  
 }  
}

### 忽略测试enabled

“忽略”表示是否忽略单元测试

public class TestNgIgnore {  
 @Test(enabled = true)  
 public void test1(){  
 System.*out*.println("this is test1 ---------------");  
 }  
 @Test(enabled = false)  
 public void test2(){  
 System.*out*.println("this is test2 ---------------");  
 }

### 时间测试timeOut

“时间测试”表示如果单元测试所花费的时间超过指定的毫秒数，则测试将会终止，并将其标记为失败

public class TestNgTimeout {  
 @Test(timeOut = 3000)  
 public void test1(){  
 System.*out*.println("this is test1 ---------------");  
 }  
}

### 分组测试groups

捆绑几个单元测试并一起运行. 下面是一个有四个方法的类，三个组(A1，A2和A3)

public class TestNgGroups {  
 @Test(groups = "A1")  
 public void test1(){  
 System.*out*.println("this is test1 ---------------");  
 }  
 @Test(groups = "A2")  
 public void test2(){  
 System.*out*.println("this is test2 ---------------");  
 }  
 @Test(groups = "A1")  
 public void test3(){  
 System.*out*.println("this is test3 ---------------");  
 }  
 @Test(groups = "A3")  
 public void test4(){  
 System.*out*.println("this is test4 ---------------");  
 }  
}

testing.xml设置如下：

<?xml version="1.0" encoding="utf-8" ?>  
<suite name="testproj" parallel="false">  
 <test name="testDemo1">  
 <groups>  
 <run>  
 <include name="A1"/>  
 </run>  
 </groups>  
 <classes>  
 <class name="testng.TestNgGroups"></class>  
 </classes>  
 </test>  
</suite>

### 参数化测试

#### @DataProvider用于参数化测试

public class TestNgDataprovider {  
 @Test(dataProvider = "userData")  
 public void test(Class clazz, String[] str) {  
 System.*out*.println(clazz + "-------------" + str[0]);  
 System.*out*.println(clazz + "-------------" + str[1]);  
 }  
  
 @DataProvider(name = "userData")  
 public Object[][] data() {  
 Object[][] objects = new Object[][]{  
 {Vector.class, new String[]{"java.util.Arrays", "java.util.List"}},  
 {String.class, new String[]{"this is my str", "this is my pp"}},  
 {Integer.class, new String[]{"123", "345"}},  
 {Float.class, new String[]{"12.45f", "33.11f"}}};  
 return objects;  
 }  
}

输出：

class java.util.Vector-------------java.util.Arrays

class java.util.Vector-------------java.util.List

class java.lang.String-------------this is my str

class java.lang.String-------------this is my pp

class java.lang.Integer-------------123

class java.lang.Integer-------------345

class java.lang.Float-------------12.45f

class java.lang.Float-------------33.11f

#### @Parameters用于参数化测试

“@Parameters”在需要参数测试的方法中声明，参数化数据将在TestNG的XML配置文件中提供

@Parameters({"param1"})  
@Test  
public void paramterTest(String param1){  
 System.*out*.println("\n---------------"+param1);  
}

Testng.xml中设置如下：

<?xml version="1.0" encoding="utf-8" ?>  
<suite name="testngpro" parallel="tests" thread-count="1">  
 <parameter name="param1" value="http://127.0.0.1:4723/wd/hub" />  
 <test name="testDemo1">  
 <classes>  
 <class name="testng.TestNgDataprovider"></class>  
 </classes>  
 </test>  
</suite>

注意：直接运行TestNgDataprovider类，会报错。需要运行testng.xml才可以。

Parameter &apos;param1&apos; is required by @Test on method paramterTest but has not been marked @Optional or defined

in C:\Users\ccc\.IntelliJIdea2017.3\system\temp-testng-customsuite.xml

at org.testng.internal.Parameters.createParams(Parameters.java:290)

at org.testng.internal.Parameters.createParametersForMethod(Parameters.java:359)

at org.testng.internal.Parameters.createParameters(Parameters.java:620)

at org.testng.internal.Parameters.handleParameters(Parameters.java:769)

at org.testng.internal.ParameterHandler.handleParameters(ParameterHandler.java:49)

at org.testng.internal.ParameterHandler.createParameters(ParameterHandler.java:37)

at org.testng.internal.Invoker.invokeTestMethods(Invoker.java:914)

at org.testng.internal.TestMethodWorker.invokeTestMethods(TestMethodWorker.java:125)

at org.testng.internal.TestMethodWorker.run(TestMethodWorker.java:109)

at org.testng.TestRunner.privateRun(TestRunner.java:648)

at org.testng.TestRunner.run(TestRunner.java:505)

at org.testng.SuiteRunner.runTest(SuiteRunner.java:455)

at org.testng.SuiteRunner.runSequentially(SuiteRunner.java:450)

at org.testng.SuiteRunner.privateRun(SuiteRunner.java:415)

at org.testng.SuiteRunner.run(SuiteRunner.java:364)

at org.testng.SuiteRunnerWorker.runSuite(SuiteRunnerWorker.java:52)

at org.testng.SuiteRunnerWorker.run(SuiteRunnerWorker.java:84)

at org.testng.TestNG.runSuitesSequentially(TestNG.java:1187)

at org.testng.TestNG.runSuitesLocally(TestNG.java:1116)

at org.testng.TestNG.runSuites(TestNG.java:1028)

at org.testng.TestNG.run(TestNG.java:996)

at org.testng.IDEARemoteTestNG.run(IDEARemoteTestNG.java:72)

at org.testng.RemoteTestNGStarter.main(RemoteTestNGStarter.java:123)

### 并发测试

#### 测试类使用注解threadPoolSize

threadPoolSize用来指明线程池的大小。

以下例子，并发数为5，可用线程数3

public class TestNgThreadPoolSize {  
 @Test(threadPoolSize = 3,invocationCount = 5)  
 public void threadPool(){  
 System.*out*.println("Thread ----------"+Thread.*currentThread*().getName());  
 }  
}

输出：

Thread ----------TestNG-methods-3

Thread ----------TestNG-methods-2

Thread ----------TestNG-methods-2

Thread ----------TestNG-methods-3

Thread ----------TestNG-methods-1

#### Testng配置文件并发测试

1）、Parallel=”methods”的意思是指TestNG会将method作为并发的元子单位，即每个method运行在自己的thread中

public class TestNgThreadPoolSize1 {  
 @Test  
 public void threadPool\_A(){  
 System.*out*.println();  
 System.*out*.println("Thread A----------"+Thread.*currentThread*().getId());  
 }  
 @Test  
 public void threadPool\_B(){  
 System.*out*.println("Thread B----------"+Thread.*currentThread*().getId());  
 }  
 @Test  
 public void threadPool\_C(){  
 System.*out*.println("Thread C----------"+Thread.*currentThread*().getId());  
 }  
 @Test  
 public void threadPool\_D(){  
 System.*out*.println("Thread D----------"+Thread.*currentThread*().getId());  
 }  
 @Test  
 public void threadPool\_E(){  
 System.*out*.println("Thread E----------"+Thread.*currentThread*().getId());  
 }  
}

testing.xml配置：

<?xml version="1.0" encoding="utf-8" ?>  
<suite name="testngpro" parallel="methods" thread-count="2">  
 <parameter name="param1" value="http://127.0.0.1:4723/wd/hub" />  
 <test name="testDemo1">  
 <classes>  
 <class name="testng.TestNgThreadPoolSize1"></class>  
 </classes>  
 </test>  
</suite>

因为parallel=”methods”,所以每个method都有自己的thread,故输出：

Thread A----------1

Thread B----------1

Thread C----------1

Thread D----------1

Thread E----------1

2）、parallel=”tests”,则指会将test 作为并发的元子单位。

# Jvisualvm分析dump文件

## Jvisualvm简介

jvisualvm是JDK自带的Java性能分析工具，在JDK的bin目录下，文件名就叫jvisualvm.exe。

jvisualvm可以监控本地、远程的java进程，实时查看进程的cpu、堆、线程等参数，对java进程生成dump文件，并对dump文件进行分析。

## 监控java服务的堆内存溢出

运行JAVA类，Jvisualvm后台监控

需要注意：

1. 运行的类所属工程为web工程
2. 想要控制程序可用内存，需要部署到tomcat
3. 控制类可用内存大小，可在右键Run Configurations🡪Arguments的VM arguments中添加如：-Xms256m -Xmx256m的参数
4. 打开JDK的安装目录的bin目录下的 jvisualvm.exe

监控的类：

**import** java.util.ArrayList;

**public** **class** JavaHeapTest {

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

{

ArrayList list=**new** ArrayList();

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

{

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

list.add(**new** JavaHeapTest());

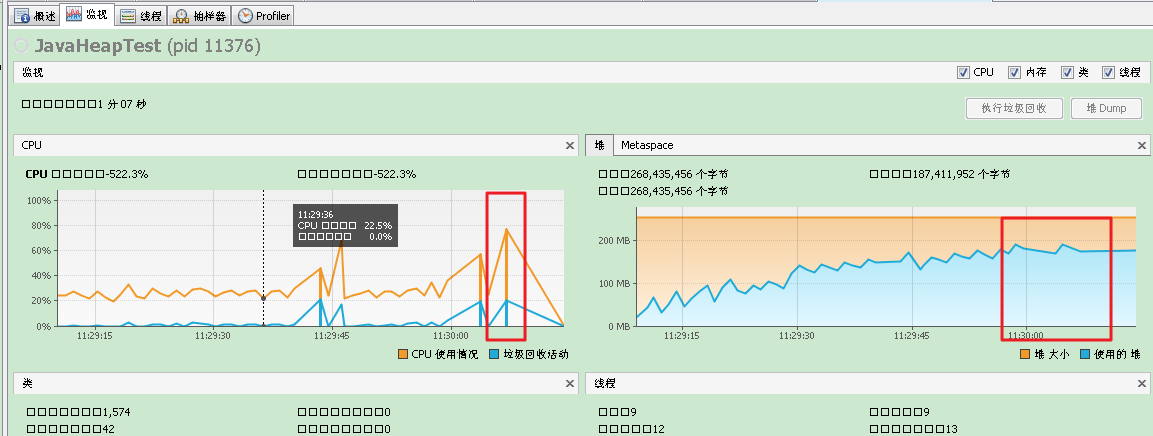
}

}

}

Jvisualvm监视截图：

场景：本地设置的最大堆内存是256M左右，代码里面是往一个List里面疯狂加数据，直到撑爆堆内存，得到的监控内容如下：



分析：红框框出的部分是发生堆内存溢出时的情形，

右侧已使用的堆大小（蓝色部分）快接近200MB时，左侧蓝色部分的垃圾回收因为内存过小大量使用CPU进行垃圾回收，使 CPU使用率过高，导致内存溢出。

所以，dump文件中的实例列表其实是反映了使用的堆的情况，而使用的堆内存并没有达到预先设置的最大堆内存，只是在申请堆内存的过程中超出了预先设置的最大堆内存，然后内存溢出。

也可以cmd，执行命令行：jmap -dump:format=b,file=20170307.dump 16048 生成dump文件

20170307.dump 为指定dump文件名称

16048 为线程Pid,可通过jps命令查看

运行时，可以添加参数 -XX:+HeapDumpOnOutOfMemoryError 在当前工程下会自动生成一个dump文件，然后装入Jvisualvm分析。

## 监控java服务的死锁

1. 运行java类DeadLock，程序锁定
2. Cmd, 输入命令 jps,查看当前程序pid
3. 根据pid，执行：jstack 3324 > 3324.dump 命令 （3324为pid）
4. 到执行命令的目录，查看3324.dump文件，发现死锁情况

**public** **class** DeadLock {

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

Resource r1 = **new** Resource();

Resource r0 = **new** Resource();

Thread myTh1 = **new** LockThread1(r1, r0);

Thread myTh0 = **new** LockThread0(r1, r0);

myTh1.setName("DeadLock-1 ");

myTh0.setName("DeadLock-0 ");

myTh1.start();

myTh0.start();

}

}

**class** Resource {

**private** **int** i;

**public** **int** getI() {

**return** i;

}

**public** **void** setI(**int** i) {

**this**.i = i;

}

}

**class** LockThread1 **extends** Thread {

**private** Resource r1, r2;

**public** LockThread1(Resource r1, Resource r2) {

**this**.r1 = r1;

**this**.r2 = r2;

}

@Override

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

**int** j = 0;

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

**synchronized** (r1) {

System.*out*.println("The first thread got r1's lock " + j);

**synchronized** (r2) {

System.*out*.println("The first thread got r2's lock " + j);

}

}

j++;

}

}

}

**class** LockThread0 **extends** Thread {

**private** Resource r1, r2;

**public** LockThread0(Resource r1, Resource r2) {

**this**.r1 = r1;

**this**.r2 = r2;

}

@Override

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

**int** j = 0;

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

**synchronized** (r2) {

System.*out*.println("The second thread got r2's lock " + j);

**synchronized** (r1) {

System.*out*.println("The second thread got r1's lock" + j);

}

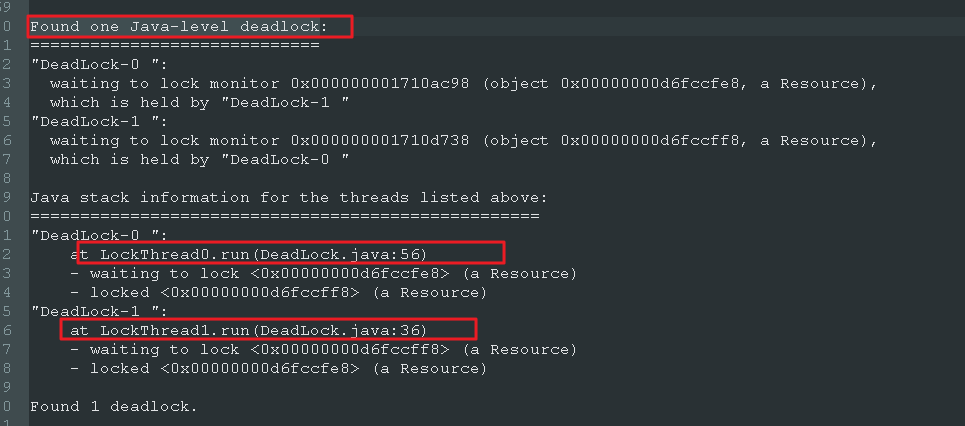
}

j++;

}

}

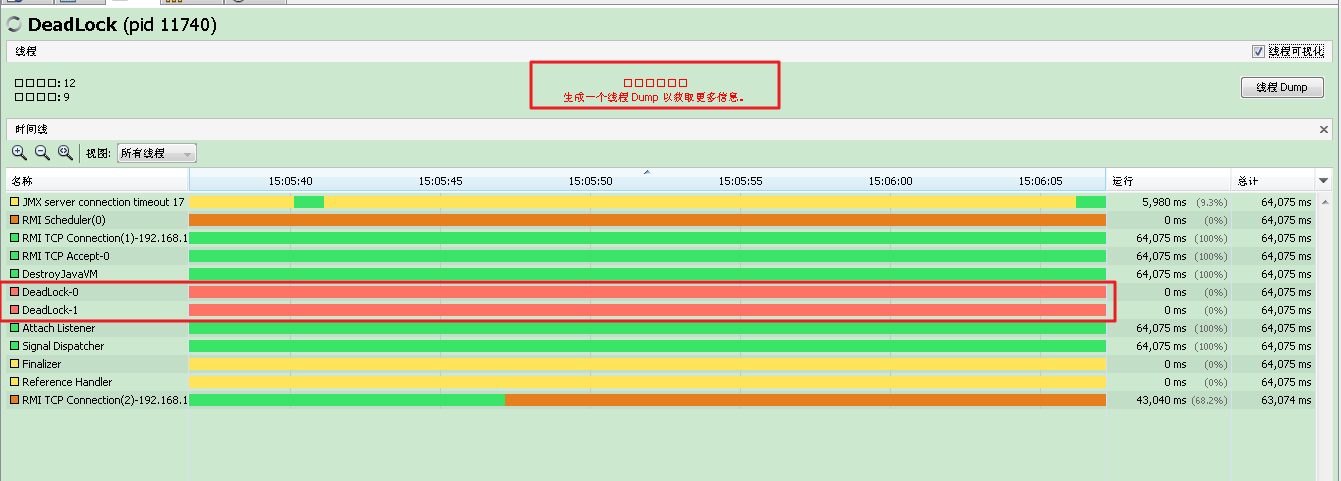
}

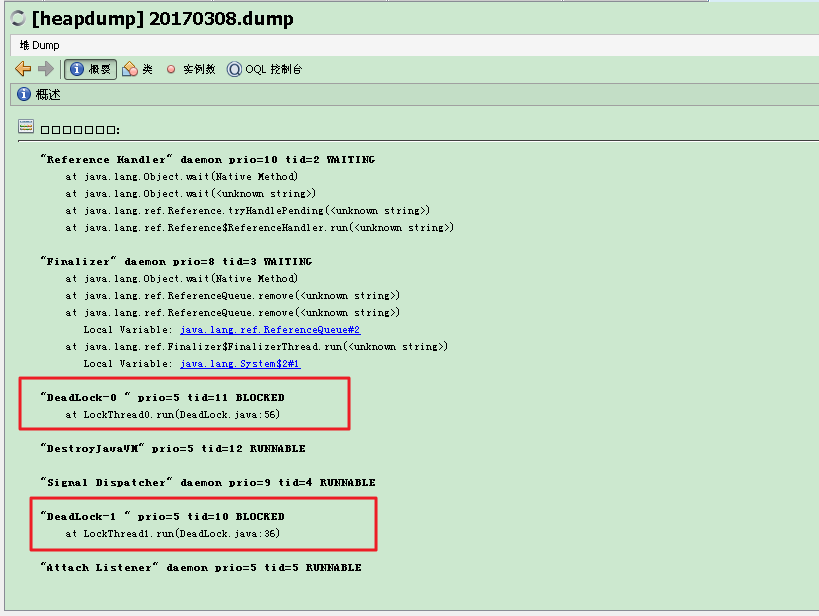


上述dump文件的红色框中，可定位发生死锁的类的具体哪一行代码有异常。

可以将上述dump文件导入到Jvisualvm中查看。

亦可以直接在Jvisualvm中监控当前线程。红色字体内容为发现死锁。





# Web开发Session超时设置

在Java Web开发中，Session为我们提供了很多方便，Session是由浏览器和服务器之间维护的。Session超时理解为：浏览器和服务器之间创建了一个Session，由于客户端长时间（休眠时间）没有与服务器交互，服务器将此Session销毁，客户端再一次与服务器交互时之前的Session就不存在了.

## 设置Session超时时间方式

### 方式一：web.xml中设置session-config

在web.xml中设置session-config如下：

<session-config>

<session-timeout>2</session-timeout>

</session-config>

即客户端连续两次与服务器交互间隔时间最长为2分钟，2分钟后session.getAttribute()获取的值为空.

### 方式二：在Tomcat的/conf/web.xml中设置session-config

在Tomcat的/conf/web.xml中session-config,默认值为：30分钟

<session-config>

<session-timeout>30</session-timeout>

</session-config>

### 方式三：在Servlet中设置setMaxInactiveInterval

HttpSession session = request.getSession();

session.setMaxInactiveInterval(60);//单位为秒

## 重点说明

1.优先级：Servlet中API设置 > 程序/web.xml设置 > Tomcat/conf/web.xml设置

2.若访问服务器session超时（本次访问与上次访问时间间隔大于session最大的不活动的间隔时间）了，即上次会话结束，但服务器与客户端会产生一个新的会话，之前的session里的属性值全部丢失，产生新的sesssionId

3.客户端与服务器一次有效会话（session没有超时），每次访问sessionId相同，若代码中设置了session.setMaxInactiveInterval()值，那么这个session的最大不活动间隔时间将被修改，并被应用为新值。

4.Session的销毁（代表会话周期的结束）：在某个请求周期内调用了Session.invalidate()方法，此请求周期结束后，session被销毁；或者是session超时后自动销毁；或者客户端关掉浏览器

5.对于JSP，如果指定了<%@ page session="false"%>，则在JSP中无法直接访问内置的session变量，同时也不会主动创建session，因为此时JSP未自动执行request.getSession()操作获取session。

# Java Listener 使用

监听器用于监听web应用中某些对象、信息的创建、销毁、增加，修改，删除等动作的发生，然后作出相应的响应处理。当范围对象的状态发生变化的时候，服务器自动调用监听器对象中的方法。常用于统计在线人数和在线用户，系统加载时进行信息初始化，统计网站的访问量等等。

## 使用步骤

1.通过实现具体接口创建实现类（可实现多个监听器接口）

2.配置实现类成为监听器，有两种配置方式：

直接用@WebListener注解修饰实现类

通过web.xml方式配置，代码如下：

<listener>  
 <listener-class>  
 session.SessionCounter  
 </listener-class>  
</listener>

## 常用Web事件监听接口

### ServletContextListener用于监听Web应用的启动与关闭

该接口的两个方法：

contextInitialized(ServletContextEvent event); // 启动web应用时调用

contextDestroyed(ServletContextEvent event); // 关闭web应用时调用

如何获得application对象：

ServletContext application = event.getServletContext();

示例：

@WebListener

public class MyServetContextListener implements ServletContextListener{

//web应用关闭时调用该方法

@Override

public void contextDestroyed(ServletContextEvent event) {

ServletContext application = event.getServletContext();

String userName = application.getInitParameter("userName");

System.out.println("关闭web应用的用户名字为："+userName);

}

//web应用启动时调用该方法

@Override

public void contextInitialized(ServletContextEvent event) {

ServletContext application = event.getServletContext();

String userName = application.getInitParameter("userName");

System.out.println("启动web应用的用户名字为："+userName);

}

}

### ServletContextAttributeListener用于监听ServletContext范围（application）内属性的改变

该接口的两个方法：

attributeAdded(ServletContextAttributeEvent event); // 当把一个属性存进application时触发

attributeRemoved(ServletContextAttributeEvent event); // 当把一个属性从application删除时触发

attributeReplaced(ServletContextAttributeEvent event); // 当替换application内的某个属性值时触发

如何获得application对象：

ServletContext application = event.getServletContext();

示例：

@WebListener

public class MyServletContextAttributeListener implements ServletContextAttributeListener{

//向application范围内添加一个属性时触发

@Override

public void attributeAdded(ServletContextAttributeEvent event) {

String name = event.getName();//向application范围添加的属性名

Object val = event.getValue(); //向application添加的属性对应的属性值

System.out.println("向application范围内添加了属性名为："+name+"，属性值为："+val+"的属性");

}

//删除属性时触发

@Override

public void attributeRemoved(ServletContextAttributeEvent event) {

// ...

}

//替换属性值时触发

@Override

public void attributeReplaced(ServletContextAttributeEvent event) {

// ...

}

}

### ServletRequestListener与ServletRequestAttributeListener

ServletRequestListener用于监听用户请求

ServletRequestAttributeListener用于监听request范围内属性的变化。

ServletRequestListener两个需要实现的方法

requestInitialized(ServletRequestEvent event); //用户请求到达、被初始化时触发

requestDestroyed(ServletRequestEvent event); // 用户请求结束、被销毁时触发

ServletRequestAttributeListener两个需要实现的方法

attributeAdded(ServletRequestAttributeEvent event); // 向request范围内添加属性时触发

attributeRemoved(ServletRequestAttributeEvent event); // 从request范围内删除某个属性时触发

attributeReplaced(ServletRequestAttributeEvent event); // 替换request范围内某个属性值时触发

获取reqeust对象

HttpServletRequest req = (HttpServletRequest)event.getServletRequest();

示例：

@WebListener

public class MyRequestListener implements ServletRequestListener,ServletRequestAttributeListener{

//用户请求结束、被销毁时触发

@Override

public void requestDestroyed(ServletRequestEvent event) {

HttpServletRequest req = (HttpServletRequest) event.getServletRequest();

String ip = req.getRemoteAddr();

System.out.println("IP为:"+ip+"的用户发送到"+req.getRequestURI()+"的请求结束");

}

//用户请求到达、被初始化时触发

@Override

public void requestInitialized(ServletRequestEvent event) {

HttpServletRequest req = (HttpServletRequest) event.getServletRequest();

String ip = req.getRemoteAddr();

System.out.println("IP为:"+ip+"的用户发送到"+req.getRequestURI()+"的请求被初始化");

}

//向request范围内添加属性时触发

@Override

public void attributeAdded(ServletRequestAttributeEvent event) {

String name = event.getName();

Object val = event.getValue();

System.out.println("向request范围内添加了名为："+name+"，值为："+val+"的属性");

}

//删除request范围内某个属性时触发

@Override

public void attributeRemoved(ServletRequestAttributeEvent event) {

//...

}

//替换request范围内某个属性值时触发

@Override

public void attributeReplaced(ServletRequestAttributeEvent event) {

// ...

}

}

### HttpSessionListener与HttpSessionAttributeListener

HttpSessionListener监听用户session的开始与结束

HttpSessionAttributeListener监听HttpSession范围（session）内的属性的改变。

HttpSessionListener要实现的方法：

sessionCreated(HttpSessionEvent event); // 用户与服务器的会话开始、创建时触发

sessionDestroyed(HttpSessionEvent event); // 用户与服务器的会话结束时触发

HttpSessionAttributeListener要实现的方法：

attributeAdded(HttpSessionBindingEvent event) ; // 向session范围内添加属性时触发

attributeRemoved(HttpSessionBindingEvent event); // 删除session范围内某个属性时触发

attributeReplaced(HttpSessionBindingEvent event); // 替换session范围内某个属性值时触发

如何得到session对象

HttpSession session = event.getSession();

示例：**用监听器统计网站在线人数**

原理：每当有一个访问连接到服务器时，服务器就会创建一个session来管理会话。那么我们就可以通过统计session的数量来获得当前在线人数。

所以这里用到的是HttpSessionListener。

1：创建监听器类，实现HttpSessionListener接口。

2：重写监听器类中的方法

import javax.servlet.http.HttpSessionEvent;  
import javax.servlet.http.HttpSessionListener;  
import java.util.concurrent.atomic.AtomicInteger;  
  
public class SessionCounter1 implements HttpSessionListener {  
 AtomicInteger count = new AtomicInteger(0);  
 @Override  
 public void sessionCreated(HttpSessionEvent httpSessionEvent) {  
 count.incrementAndGet();//返回的是新值（即加1后的值）  
 count.getAndIncrement();//方法是返回旧值（即加1前的原始值）  
 }  
  
 @Override  
 public void sessionDestroyed(HttpSessionEvent httpSessionEvent) {  
 count.decrementAndGet();//返回的是新值（即减1后的值）  
 count.getAndIncrement();//方法是返回旧值（即减1前的值）  
 }  
 //获取活动的session个数(在线人数)  
 public AtomicInteger getActiveSessions() {  
 return count;  
 }  
}

3：在web.xml中配置监听器。注意：**监听器>过滤器>serlvet，配置的时候要注意先后顺序**

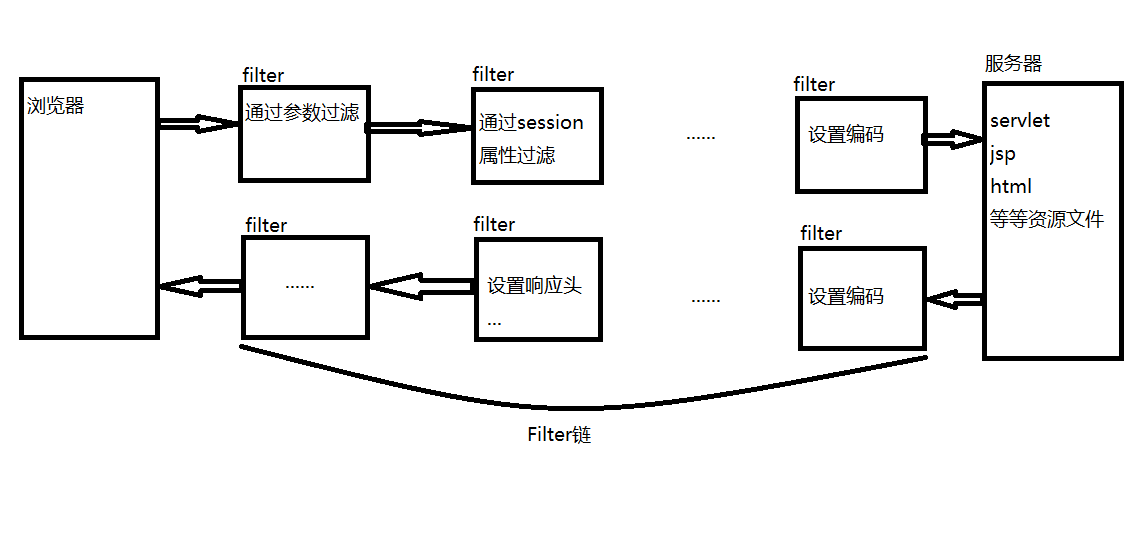
<listener>  
 <listener-class>  
 session.SessionCounter1  
 </listener-class>  
</listener>

 4：在显示在线人数处通过session.getAttribute("Count")即可获取在线人数值。

# Java Filter使用

## Filter 原理

过滤器是处于客户端与服务器资源文件之间的一道过滤网，在访问资源文件之前，通过一系列的过滤器对请求进行修改、判断等，把不符合规则的请求在中途拦截或修改。也可以对响应进行过滤，拦截或修改响应。



如图，浏览器发出的请求先递交给第一个filter进行过滤，符合规则则放行，递交给filter链中的下一个过滤器进行过滤。过滤器在链中的顺序与它在web.xml中配置的顺序有关，配置在前的则位于链的前端。当请求通过了链中所有过滤器后就可以访问资源文件了，如果不能通过，则可能在中间某个过滤器中被处理掉。

在doFilter（）方法中，chain.doFilter()前的一般是对request执行的过滤操作，chain.doFilter后面的代码一般是对response执行的操作。

## Filter的常用操作

过滤器一般用于登录权限验证、资源访问权限控制、敏感词汇过滤、字符编码转换等等操作，便于代码重用，不必每个servlet中还要进行相应的操作

## Filter的使用

1、新建一个class，实现接口Filter（注意：是javax.servlet中的Filter）

2、重写过滤器的doFilter(request，response，chain)方法。另外两个init()、destroy()方法一般不需要重写。在doFilter方法中进行过滤操作。常用逻辑有：获取请求、获取响应、获取session、放行

HttpServletRequest request=(HttpServletRequest) arg0;//获取request对象

HttpServletResponse response=(HttpServletResponse) arg1;//获取response对象

HttpSession session=request.getSession();//获取session对象

过滤操作代码......

chain.doFilter(request, response);//放行，通过了当前过滤器，递交给下一个filter进行过滤

3、在web.xml中配置过滤器。这里要谨记一条原则：在web.xml中，监听器>过滤器>servlet。也就是说web.xml中监听器配置在过滤器之前，过滤器配置在servlet之前，否则会出错

<filter>

<filter-name>loginFilter<／filter-name>／／过滤器名称

<filter-class>com.ygj.control.loginFilter<／filter-class>／／过滤器类的包路径

<init—param> //可选

<param—name>参数名<／param-name>／／过滤器初始化参数

<param-value>参数值<／param-value>

<／init—pamm>

<／filter>

<filter-mapping>／／过滤器映射

<filter-name>loginFilter<／filter-name>

<url—pattern>指定过滤器作用的对象</url-pattern>

在配置中需要注意的有两处：一是<filter-class>指明过滤器类所在的包路径。二是<url-pattren>处定义过滤器作用的对象。一般有以下规则：

1：作用与所有web资源：<url—pattern>/\*</url-pattern>。则客户端请求访问任意资源文件时都要经过过滤器过滤，通过则访问文件，否则拦截。

2：作用于某一文件夹下所有文件：<url—pattern>/dir/\*</url-pattern>

3：作用于某一种类型的文件：<url—pattern>\*.扩展名</url-pattern>。比如<url—pattern>\*.jsp</url-pattern>过滤所有对jsp文件的访问请求。

4：作用于某一文件夹下某一类型文件：<url—pattern>/dir/\*.扩展名</url-pattern>

如果一个过滤器需要过滤多种文件，则可以配置多个<filter-mapping>，一个mapping定义一个url-pattern来定义过滤规则。

### 示例1：用过滤器实现登录验证，没登录则驳回访问请求并重定向到登录页面。

public class LoginFilter implements Filter {  
 @Override  
 public void init(FilterConfig filterConfig) throws ServletException {  
 }  
  
 @Override  
 public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {  
 HttpServletRequest request = (HttpServletRequest) servletRequest;  
 HttpServletResponse response = (HttpServletResponse) servletResponse;  
  
 String requestURI = request.getRequestURI().substring(request.getRequestURI().indexOf("/",1),request.getRequestURI().length());  
 if("/user/adduser.jsp".equals(requestURI)||"/UserServlet".equals(requestURI)){  
 filterChain.doFilter(request,response);  
 }else{  
 HttpSession session = request.getSession(false);  
 if(session == null || session.getAttribute("name")==null){  
 response.sendRedirect(request.getContextPath()+"/user/adduser.jsp");  
 return;  
 }else{  
 filterChain.doFilter(request,response);  
 }  
 }  
 }  
  
 @Override  
 public void destroy() {  
 }  
}

web.xml文件配置：

<filter>  
 <filter-name>loginFilter</filter-name>  
 <filter-class>filter.LoginFilter</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>loginFilter</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

### 示例2：设置字符编码

public class EncodingFilter implements Filter {  
  
 private String encode;  
  
 @Override  
 public void init(FilterConfig filterConfig) throws ServletException {  
 this.encode = filterConfig.getInitParameter("encode");  
 }  
  
 @Override  
 public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {  
 // 1.强制转换  
 HttpServletRequest request = (HttpServletRequest)servletRequest;  
 HttpServletResponse response =(HttpServletResponse)servletResponse;  
  
 // 2.操作  
 request.setCharacterEncoding(encode);  
 response.setContentType("text/html;charset=utf-8");  
  
 // 3.放行  
 filterChain.doFilter(request,response);  
 }  
  
 @Override  
 public void destroy() {  
 }  
}

web.xml文件配置：

<filter>  
 <filter-name>encodingFilter</filter-name>  
 <filter-class>filter.EncodingFilter</filter-class>  
  
 <init-param>  
 <param-name>encode</param-name>  
 <param-value>utf-8</param-value>  
 </init-param>  
</filter>  
<filter-mapping>  
 <filter-name>encodingFilter</filter-name>  
 <url-pattern>/\*</url-pattern>  
</filter-mapping>

# Spring简介

IoC（Inversion of Control，控制反转）。

是spring的核心，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。

Spring所倡导的开发方式是所有的类都会在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。

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

## Spring 常用注解

**@Component[不分层的情况下]**

**@Repository() [Dao层]**

对应数据访问层Bean

1. @Repository(value="userDao")
2. **public** **class** UserDaoImpl **extends** BaseDaoImpl<User> {
3. ………
4. }

@Repository(value="userDao")注解是告诉Spring，让Spring创建一个名字叫“userDao”的UserDaoImpl实例。

当Service需要使用Spring创建的名字叫“userDao”的UserDaoImpl实例时，就可以使用@Resource(name = "userDao")注解告诉Spring，Spring把创建好的userDao注入给Service即可。

1. // 注入userDao，从数据库中根据用户Id取出指定用户时需要用到
2. @Resource(name = "userDao")
3. **private** BaseDao<User> userDao;

**@Service() [service层]**

对应的是业务层Bean

1. @Service("userService")
2. **public** **class** UserServiceImpl **implements** UserService {
3. ………
4. }

@Service("userService")注解是告诉Spring，当Spring要创建UserServiceImpl的的实例时，bean的名字必须叫做"userService"，这样当Action需要使用UserServiceImpl的的实例时,就可以由Spring创建好的"userService"，然后注入给Action：在Action只需要声明一个名字叫“userService”的变量来接收由Spring注入的"userService"即可

1. // 注入userService
2. @Resource(name = "userService")
3. **private** UserService userService;

**@Controller() [Action类]**

对应表现层的Bean，也就是Action

1. @Controller
2. @Scope("prototype")
3. **public** **class** UserAction **extends** BaseAction<User>{
4. ……
5. }

使用@Controller注解标识UserAction之后，就表示要把UserAction交给Spring容器管理，在Spring容器中会存在一个名字为"userAction"的action，这个名字是根据UserAction类名来取的。

如果@Controller不指定其value【@Controller】，则默认的bean名字为这个类的类名首字母小写，如果指定value【@Controller(value="UserAction")】或者【@Controller("UserAction")】，则使用value作为bean的名字。

可以利用容器的scope="prototype"来保证每一个请求有一个单独的Action来处理，避免struts中Action的线程安全问题

# Spring 创建对象

## 直接new

如：User usero = new User();

## 反射获取对象

Class clz= Class.*forName*("com.sc.test.User");   
User user = (User) clz.newInstance();  
user.setAge(12);  
System.*out*.println(user);

## 利用spring框架,设值注入xml

spring容器创建的时候，会将所有配置的bean对象创建出来，默认bean都是单例的。代码通过getBean()方法从容器获取指定的bean实例，容器首先会调用Bean类的无参构造器，创建实例对象。

<?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"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  
 <bean id="user" class="com.sc.test.User">  
 <property name="name" value="sundy"></property>  
 <property name="age" value="99"></property>  
 </bean>  
</beans>

测试类：

import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
public class XmlBeansTest {  
 public static void main(String[] args) {  
 ApplicationContext context = new ClassPathXmlApplicationContext("springmvc.xml");  
 User user1 = (User)context.getBean("user");  
 User user2 = context.getBean(User.class);  
 System.*out*.println(user1==user2);  
 }  
}

上述类中，构建出两个学生对象，执行测试后发现两个对象的内存地址相同，所以打印出user1==user2的结果为 true。

因为spring默认创建的对象是单例的，配置文件中默认配置为：scope="singleton"（单例模式：容器初始化时需要使用name建，每次获取的都是同一个对象，默认值），同上。

如果需要创建多例对象，需要在配置文件中添加scope="prototype"（原型模式（N个对象）：真正使用时才会创建，每获取一次，都会创建不同对象）即可

## 利用spring框架,构造注入xml

首先，需要在实体bean中添加对应的构造函数：

public People(String name, String age, Car car) {  
 this.name = name;  
 this.age = age;  
 this.car = car;  
}

然后，需要在xml中注入该实体bean,并为各属性设值。

<bean id="people" class="com.sc.bean.People">  
 <constructor-arg index="0" value="ssss"/>  
 <constructor-arg index="1" value="23"/>  
 <constructor-arg index="2" ref="car"/>  
</bean>

验证构造注入并输出：

ApplicationContext context = new ClassPathXmlApplicationContext("springmvc.xml");

People people =(People) context.getBean("people");  
System.*out*.println(people);

输出结果：People{name='ssss', age='23', car=Car{name='北京现代', price=550000.0}}

## 利用spring框架,命名空间P注入xml

首先，需要在xml中引入命名空间p

xmlns:p="http://www.springframework.org/schema/p"

其次，实例bean中需要有无参的构造函数

public People() {  
}

然后，使用命名空间p注入

<bean id="people1" class="com.sc.bean.People" p:name="aaaa" p:age="15" p:car-ref="car"/>

验证，并输出测试结果：

People people1 =(People) context.getBean("people1");  
System.*out*.println(people1);

输出结果：

People{name='aaaa', age='15', car=Car{name='北京现代', price=550000.0}}

## 在xml中直接配置需要扫描的包

注：在项目中添加Spring AOP相关的JAR文件以及xsd约束文件。

由于是基于注解的DI，所以无需再Spring配置文件中进行节点配置，只需配置包扫描器即可！

配置包扫描器后，该包下以及子包中的类才可以被Spring扫描，去寻找被注解的类和属性，让Spring容器管理赋值。

<?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"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd">  
 <!-- 扫描指定包带@Controller注解的类-->  
 <context:component-scan base-package="com.sc.test">  
 <context:include-filter type="annotation" expression="org.springframework.stereotype.Component"/>  
 </context:component-scan>  
</beans>

指定@Component中的value即可在测试类中的getBean（）中植入即可。

package com.sc.test;  
import org.springframework.stereotype.Component;  
  
@Component("cartest")  
public class Car {  
 private String name;  
 private double price;  
 public String getName() {  
 return name;  
 }  
 public void setName(String name) {  
 this.name = name;  
 }  
 public double getPrice() {  
 return price;  
 }  
 public void setPrice(double price) {  
 this.price = price;  
 }  
 @Override  
 public String toString() {  
 return "Car [name=" + name + ", price=" + price + "]";  
 }  
}

# Spring 关联对象注入

## 1.单个对象依赖

Xml中需要配置要关联的对象 用ref引入：

<bean id="car" class="com.sc.bean.Car">  
 <property name="name" value="北京现代"></property>  
 <property name="price" value="550000"></property>  
</bean>  
<bean id="users" class="com.sc.bean.User">  
 <property name="name" value="sc"></property>  
 <property name="age" value="19"></property>  
 <property name="car" ref="car"></property>  
</bean>

测试：

ApplicationContext context = new ClassPathXmlApplicationContext("springmvc.xml");  
User user = context.getBean(User.class);  
System.*out*.println(user.getCar());

## 2.对象List依赖

Xml中需要配置要关联的对象，用List引入：

<bean id="users" class="com.sc.bean.User">  
 <property name="name" value="sc"></property>  
 <property name="age" value="19"></property>  
 <property name="carlist">  
 <list>  
 <bean class="com.sc.bean.Car">  
 <property name="name" value="上海大众"></property>  
 <property name="price" value="250000"></property>  
 </bean>  
 <ref bean="car"></ref> <!-- 这里可以直接引入外部对象-->  
 </list>  
 </property>  
</bean>

测试：

ApplicationContext context = new ClassPathXmlApplicationContext("springmvc.xml");  
User user = context.getBean(User.class);  
System.*out*.println(user.getCarlist());

输出：

[Car{name='上海大众', price=250000.0}, Car{name='北京现代', price=550000.0}]

## 3.重复对象Set依赖

当xml中引用的集合里有重复对象时，使用set注入，可去重。

<bean id="users" class="com.sc.bean.User">  
 <property name="name" value="sc"></property>  
 <property name="age" value="19"></property>  
 <property name="carset">  
 <set>  
 <bean class="com.sc.bean.Car">  
 <property name="name" value="上海大众"></property>  
 <property name="price" value="250000"></property>  
 </bean>  
 <ref bean="car"></ref> <!-- 这里可以直接引入外部对象-->  
 <ref bean="car"></ref>  
 </set>  
 </property>  
</bean>

测试输出：

System.*out*.println(user.getCarset());

[Car{name='上海大众', price=250000.0}, Car{name='北京现代', price=550000.0}]

## 4.对象Map依赖

<bean id="users" class="com.sc.bean.User">  
 <property name="name" value="sc"></property>  
 <property name="age" value="19"></property>  
 <property name="carmap">  
 <map>  
 <entry key="name" value="一汽大众"></entry><!--分别录入对象属性-->  
 <entry key="price" value="500000"></entry>  
 <entry key="car2" value-ref="car"></entry> <!--直接引入已有对象-->  
 <entry key="car3"> <!--将属性封装到指定对象-->  
 <bean class="com.sc.bean.Car">  
 <property name="name" value="宝马"></property>  
 <property name="price" value="1000000"></property>  
 </bean>  
 </entry>  
 </map>  
 </property>  
</bean>

其中 User类中，需要加入

private Map<Object,Object> carmap;  
  
public Map<Object, Object> getCarmap() {  
 return carmap;  
}  
  
public void setCarmap(Map<Object, Object> carmap) {  
 this.carmap = carmap;  
}

测试结果如下：

System.*out*.println(user.getCarmap());

{name=一汽大众, price=500000, car2=Car{name='北京现代', price=550000.0}, car3=Car{name='宝马', price=1000000.0}}