## 1.Jsp中request传递参数汉字乱码

<%@ page language="java" pageEncoding="UTF-8"%>

< !-- 告诉浏览器该jsp格式为utf-8，此时注意网页本身(右击->properties)的编码是否为utf-8 -->  
  
在head内加入：  
<!-- 告诉服务器,这里采用的是utf-8 -->  
< meta http-equiv="Content-Type" content="text/html; charset=utf-8">  
  
3.1对于get请求：  
String name2 = new String(name.getBytes("iso-8859-1"),"UTF-8");  
  
3.2 对于post请求  
 request.setCharacterEncoding("utf-8");//对于post方法有效，对get无效.一定要放在解析的第一行，否则不行。  
 String name = request.getParameter("name");

<%@ **page** language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>  
<%@ **page** import="java.io.\*,java.util.\*" %>  
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  
<html>  
<head>  
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">  
 <title>菜鸟教程(runoob.com)</title>  
</head>  
<body>  
<h1>使用 GET 方法读取数据</h1>  
<ul>  
 <li><p><b>站点名:</b>  
 **<%** String name =request.getParameter("name");  
 String name2 = new String(name.getBytes("iso-8859-1"),"UTF-8");  
 **%>  
 <%=** name2**%>** </p></li>  
 <li><p><b>网址:</b>  
 **<%=** request.getParameter("url")**%>** </p></li>  
</ul>  
</body>  
</html>

在浏览器中输入：

<http://localhost:8080/mavenWebDemo/test.jsp?name=%E8%8F%9C%E9%B8%9F%E6%95%99%E7%A8%8B&url=http://ww.runoob.com>

输出：



## 2. java统计字符串中每个字符出现的次数

例如String str = “abcaaaefdabbhg”;   
统计该字符串中每个字符出现的次数，输出：   
a====5   
b====3   
c====1   
d====1   
e====1   
f====1   
g====1   
h====1

public static void count(String str){  
//将字符串转化为字符数组  
char[] chars = str.toCharArray();  
//创建一个HashMap名为hm  
HashMap<Character,Integer> hm = new HashMap();  
  
//定义一个字符串c，循环遍历遍历chars数组  
for(char c : chars){  
//containsKey(c),当c不存在于hm中  
if(!hm.containsKey(c)){  
hm.put(c,1);  
}else{  
//否则获得c的值并且加1  
hm.put(c, hm.get(c)+1);  
}  
  
//或者上面的if和else替换成下面这一行  
/\* hm.put(c,hm.containsKey(c) ? hm.get(c)+1:1);\*/  
}  
  
  
for(Character key: hm.keySet()){  
//hm.keySet()代表所有键的集合,进行格式化输出  
System.out.println(key + "====" + hm.get(key));  
}  
}  
  
public static void main(String[] args) {  
// TODO 自动生成的方法存根  
String str = "abcaaaefdabbhg";  
count(str);  
}  
}

## 3.Java中字符串indexof() 的使用方法

Java中字符串中子串的查找共有四种方法(indexof())

indexOf 方法返回一个整数值，指出 String 对象内子字符串的开始位置。如果没有找到子字符串，则返回-1。

如果 startindex 是负数，则 startindex 被当作零。如果它比最大的字符位置索引还大，则它被当作最大的可能索引。

Java中字符串中子串的查找共有四种方法，如下：

1、int indexOf(String str) ：返回第一次出现的指定子字符串在此字符串中的索引。

2、int indexOf(String str, int startIndex)：从指定的索引处开始，返回第一次出现的指定子字符串在此字符串中的索引。

3、int lastIndexOf(String str) ：返回在此字符串中最右边出现的指定子字符串的索引。

4、int lastIndexOf(String str, int startIndex) ：从指定的索引处开始向后搜索，返回在此字符串中最后一次出现的指定子字符串的索引。

private static void testIndexOf() {

String string = "aaa456ac";

//查找指定字符是在字符串中的下标。在则返回所在字符串下标；不在则返回-1.

System.out.println(string.indexOf("b"));//indexOf(String str)；返回结果：-1，"b"不存在

// 从第四个字符位置开始往后继续查找，包含当前位置

System.out.println(string.indexOf("a",3));//indexOf(String str, int fromIndex)；返回结果：6

//（与之前的差别：上面的参数是 String 类型，下面的参数是 int 类型）参考数据：a-97,b-98,c-99

// 从头开始查找是否存在指定的字符

System.out.println(string.indexOf(99));//indexOf(int ch)；返回结果：7

System.out.println(string.indexOf('c'));//indexOf(int ch)；返回结果：7

//从fromIndex查找ch，这个是字符型变量，不是字符串。字符a对应的数字就是97。

System.out.println(string.indexOf(97,3));//indexOf(int ch, int fromIndex)；返回结果：6

System.out.println(string.indexOf('a',3));//indexOf(int ch, int fromIndex)；返回结果：6

//这个就是灵活运用String类提供的方法，拆分提供的字符串。

//String s = "D:\\Android\\sdk\\add-ons";

//System.out.println(s);

//while (s.lastIndexOf("\\") > 0) {

// s = s.substring(0, s.lastIndexOf("\\"));

// System.out.println(s);

//}

}

## 4.java Random.nextInt()方法的使用

public int nextInt(int n)

该方法的作用是生成一个随机的int值，该值介于[0,n)的区间，也就是0到n之间的随机int值，包含0而不包含n。

import java.util.Random;

import org.junit.Test;

public class RandomDemo {

@Test

public void Demo(){

Random rnd = new Random();

int code = rnd.nextInt(8999) + 1000;

System.out.println("code:"+code);

}

@Test

public void Demo1(){

Random r = new Random();

int nextInt = r.nextInt();

Random r1 = new Random(10);

int nextInt2 = r1.nextInt();

System.out.println("nextInt:"+nextInt);

System.out.println("nextInt2:"+nextInt2);

}

/\*\*

\* 生成[0,1.0)区间的小数

\*

\*/

@Test

public void Demo2(){

Random r = new Random();

double d1 = r.nextDouble();

System.out.println("d1:"+d1);

}

/\*\*

\* 生成[0,5.0)区间的小数

\*

\*/

@Test

public void Demo3(){

Random r = new Random();

double d2 = r.nextDouble()\* 5;

System.out.println("d1:"+d2);

}

/\*\*

\* 生成[1,2.5)区间的小数

\*

\*/

@Test

public void Demo4(){

Random r = new Random();

double d3 = r.nextDouble() \* 1.5 + 1;

System.out.println("d1:"+d3);

}

/\*\*

\* 生成任意整数

\*

\*/

@Test

public void Demo5(){

Random r = new Random();

int n1 = r.nextInt();

System.out.println("d1:"+n1);

}

/\*\*

\* 生成[0,10)区间的整数

\*

\*/

@Test

public void Demo6(){

Random r = new Random();

int n2 = r.nextInt(10);

int n3 = Math.abs(r.nextInt() % 10);

System.out.println("n2:"+n2);

System.out.println("n3:"+n3);

}

/\*\*

\* 生成[0,10]区间的整数

\*

\*/

@Test

public void Demo7(){

Random r = new Random();

int n3 = r.nextInt(11);

int n4 = Math.abs(r.nextInt() % 11);

System.out.println("n3:"+n3);

System.out.println("n4:"+n4);

}

/\*\*

\* 生成[-3,15)区间的整数

\*

\*/

@Test

public void Demo8(){

Random r = new Random();

int n4 = r.nextInt(18) - 3;

int n5 = Math.abs(r.nextInt() % 18) - 3;

System.out.println("n4:"+n4);

System.out.println("n5:"+n5);

}

}

## 5.java中charAt()方法的使用

public charcharAt(int index)

charAt(int index)方法是一个能够用来检索特定索引下的字符的String实例的方法.

charAt()方法返回指定索引位置的char值。索引范围为0~length()-1.

如: str.charAt(0)检索str中的第一个字符,str.charAt(str.length()-1)检索最后一个字符.

public static void main(String[] args) {

String s = "bejing welcome you";

System.out.println(s.charAt(1)); //e

System.out.println(s.charAt(5)); //g

System.out.println(s.charAt(15)); //y

}

## 6.BigDecimal加减乘除运算

使用BigDecimal要用String来够造，要做一个加法运算，需要先将两个浮点数转为String，然后够造成BigDecimal。

在其中一个上调用add方法，传入另一个作为参数，然后把运算的结果（BigDecimal）再转换为浮点数。

public static double add(double v1,double v2)

public static double sub(double v1,double v2)

public static double mul(double v1,double v2)

public static double div(double v1,double v2)

public static double div(double v1,double v2,int scale)

public static double round(double v,int scale)

import java.math.BigDecimal;

/\*\*

\* 由于Java的简单类型不能够精确的对浮点数进行运算，这个工具类提供精

\* 确的浮点数运算，包括加减乘除和四舍五入。

\*/

public class Arith{ //默认除法运算精度

private static final int DEF\_DIV\_SCALE = 10; //这个类不能实例化

private Arith(){

}

/\*\*

\* 提供精确的加法运算。

\* @param v1 被加数

\* @param v2 加数

\* @return 两个参数的和

\*/

public static double add(double v1,double v2){

BigDecimal b1 = new BigDecimal(Double.toString(v1));

BigDecimal b2 = new BigDecimal(Double.toString(v2));

return b1.add(b2).doubleValue();

}

/\*\*

\* 提供精确的减法运算。

\* @param v1 被减数

\* @param v2 减数

\* @return 两个参数的差

\*/

public static double sub(double v1,double v2){

BigDecimal b1 = new BigDecimal(Double.toString(v1));

BigDecimal b2 = new BigDecimal(Double.toString(v2));

return b1.subtract(b2).doubleValue();

}

/\*\*

\* 提供精确的乘法运算。

\* @param v1 被乘数

\* @param v2 乘数

\* @return 两个参数的积

\*/

public static double mul(double v1,double v2){

BigDecimal b1 = new BigDecimal(Double.toString(v1));

BigDecimal b2 = new BigDecimal(Double.toString(v2));

return b1.multiply(b2).doubleValue();

}

/\*\*

\* 提供（相对）精确的除法运算，当发生除不尽的情况时，精确到

\* 小数点以后10位，以后的数字四舍五入。

\* @param v1 被除数

\* @param v2 除数

\* @return 两个参数的商

\*/

public static double div(double v1,double v2){

return div(v1,v2,DEF\_DIV\_SCALE);

}

/\*\*

\* 提供（相对）精确的除法运算。当发生除不尽的情况时，由scale参数指

\* 定精度，以后的数字四舍五入。

\* @param v1 被除数

\* @param v2 除数

\* @param scale 表示表示需要精确到小数点以后几位。

\* @return 两个参数的商

\*/

public static double div(double v1,double v2,int scale){

if(scale<0){

throw new IllegalArgumentException(

"The scale must be a positive integer or zero");

}

BigDecimal b1 = new BigDecimal(Double.toString(v1));

BigDecimal b2 = new BigDecimal(Double.toString(v2));

return b1.divide(b2,scale,BigDecimal.ROUND\_HALF\_UP).doubleValue();

}

/\*\*

\* 提供精确的小数位四舍五入处理。

\* @param v 需要四舍五入的数字

\* @param scale 小数点后保留几位

\* @return 四舍五入后的结果

\*/

public static double round(double v,int scale){

if(scale<0){

throw new IllegalArgumentException("The scale must be a positive integer or zero");

}

BigDecimal b = new BigDecimal(Double.toString(v));

BigDecimal one = new BigDecimal("1");

return b.divide(one,scale,BigDecimal.ROUND\_HALF\_UP).doubleValue();

}

};

## 7.数组转换成List集合

方法一 ：通过add把数组中的数据循环添加到List集合中

方法二 ：采用java中集合自带的asList()方法完成转换

public static void ArrayToList(Object[] obj){  
 System.*out*.println("方法一-----------------------------------------");  
 List<Object> list = new ArrayList<Object>();  
 //String数组转List集合  
 for(Object str : obj){  
 list.add(str);  
 }  
 // 输出List集合  
 for(Object li : list){  
 System.*out*.println(li.toString());  
 }  
  
 System.*out*.println("方法二-----------------------------------------");  
  
 // String数组转List集合  
 List<Object> list1 = Arrays.*asList*(obj);  
 // 输出List集合  
 for(Object li : list1){  
 System.*out*.println(li.toString());  
 }  
}

## 8.List集合转换成数组

方法一 ：把List中的数据循环添加到数组中

方法二 ：采用集合的toArray()方法直接把List集合转换成数组

public static void ListToArray(List<Object> list){  
 System.*out*.println("方法一-----------------------------------------");  
 Object[] strArray = new Object[list.size()];  
 // List转换成数组  
 for(int i=0;i<strArray.length;i++){  
 strArray[i]=list.get(i);  
 }  
  
 // 输出List集合  
 for(Object s : strArray){  
 System.*out*.println(s.toString());  
 }  
  
  
 System.*out*.println("方法二-----------------------------------------");  
 // List转换成数组  
 Object[] str = list.toArray(new Object[list.size()]);  
 // 输出List集合  
 for(Object s:str){  
 System.*out*.println(s.toString());  
 }  
}

不管是数组转换成集合，还是集合转换成数组，都要注意转换类型的一致性，String[]数组转String类型的集合，当需要使用int，double等集合的时候，需要使用对应的对象   
如：数组int[]用Integer[]，double[]用Double[]   
因为List集合是对象的集合，而int、double等不是对象，所以需要用字段的对应对象类

## 9.Map的遍历

### 通过获取所有的key按照key来遍历

public static void getMapvalue1(Map<Object,Object> map){  
 System.*out*.println("Map 遍历方法一:");  
 for(Object o : map.keySet()){  
 System.*out*.println("key:"+o.toString()+" value:"+map.get(o).toString());  
 }  
}

### 通过Map.entrySet使用iterator遍历key和value

public static void getMapValue4(Map<Object,Object> map){  
 System.*out*.println("Map 遍历方法四:");  
 Iterator<Map.Entry<Object, Object>> iterator =map.entrySet().iterator();  
 while (iterator.hasNext()){  
 Map.Entry<Object,Object> entry = iterator.next();  
 System.*out*.println("key:"+entry.getKey().toString()+" value:"+entry.getValue().toString());  
 }  
}

### 通过Map.entrySet遍历key和value

public static void getMapValue2(Map<Object,Object> map){  
 System.*out*.println("Map 遍历方法二:");  
 for(Map.Entry<Object,Object> entry : map.entrySet()){  
 System.*out*.println("key:"+entry.getKey().toString()+" value:"+entry.getValue().toString());  
 }  
}

### 通过Map.values()遍历所有的value，但不能遍历key

public static void getMapValue3(Map<Object,Object> map){  
 System.*out*.println("Map 遍历方法三:");  
 for(Object o : map.values()){  
 System.*out*.println("value:"+o.toString());  
 }  
}

测试方法：

public static void main(String[] args) {  
 Map<Object,Object> map = new HashMap<Object,Object>();  
 for(int i=1;i<=20;i++){  
 map.put(i,"mapValue"+i);  
 }  
 *getMapvalue1*(map);  
 *getMapValue2*(map);  
 *getMapValue3*(map);  
 *getMapValue4*(map);  
}

## 10.生成唯一Token

*/\*\*  
 \* 生成Token  
 \* Token：gBu2bAZ7oC4eKxpVrWtI6A==  
 \** ***@return*** *\*/*public static String makeToken(){ //checkException  
 // 7346734837483 834u938493493849384 43434384  
 String token = (System.*currentTimeMillis*() + new Random().nextInt(999999999)) + "";  
 //数据指纹 128位长 16个字节 md5  
 try {  
 MessageDigest md = MessageDigest.*getInstance*("md5");  
 byte md5[] = md.digest(token.getBytes());  
 //base64编码--任意二进制编码明文字符 adfsdfsdfsf  
 BASE64Encoder encoder = new BASE64Encoder();  
 return encoder.encode(md5);  
 } catch (NoSuchAlgorithmException e) {  
 throw new RuntimeException(e);  
 }  
}  
  
public static void main(String[] args) {  
 String s = *makeToken*();  
 System.*out*.println(s);//wkra+s2rBtLDRAhTgBuP4w==  
}

## 11.Selenium等待页面元素出现

import org.openqa.selenium.support.ui.WebDriverWait;

WebDriverWait wait;

*/\*\*  
 \*  
 \* 等待输入框enable  
 \*  
 \** ***@param*** *webElement  
 \*/*public void waitElementClickable(WebElement webElement) {  
 long startTime = System.*currentTimeMillis*();  
 wait.until(ExpectedConditions.*elementToBeClickable*(webElement));  
 long endTime = System.*currentTimeMillis*();  
 float seconds = (endTime - startTime) / 1000F;  
  
 log.info("等待时间："+ Float.*toString*(seconds) + " seconds.");  
}  
  
*/\*\*  
 \* 等待页面元素出现  
 \*  
 \** ***@param*** *xpath  
 \*/*public void waitWebElementDisplay(String xpath) {  
  
 long startTime = System.*currentTimeMillis*();  
 wait.until(ExpectedConditions.*presenceOfElementLocated*(By.*xpath*(xpath)));  
 long endTime = System.*currentTimeMillis*();  
 float seconds = (endTime - startTime) / 1000F;  
  
 log.info("等待时间："+ Float.*toString*(seconds) + " seconds.");}  
  
*/\*\*  
 \* 等待搜索结果出现  
 \*  
 \** ***@param*** *searchCondition  
 \*/*public void waitSearchResult(String searchCondition) {  
 String xpath = "//\*[text() = '" + searchCondition + "']";  
 wait.until(ExpectedConditions.*presenceOfElementLocated*(By.*xpath*(xpath)));  
 sleep();  
}  
  
*/\*\*  
 \* 等待元素不可见  
 \*  
 \** ***@param*** *xpath  
 \*/*public void waitElementInvisible(String xpath){  
  
 long startTime = System.*currentTimeMillis*();  
 wait.until(ExpectedConditions.*invisibilityOfElementLocated*(By.*xpath*(xpath)));  
 long endTime = System.*currentTimeMillis*();  
 float seconds = (endTime - startTime) / 1000F;  
  
 log.info("等待消失时间："+ Float.*toString*(seconds) + " seconds.");  
}

## 12.javaWeb工程，使用Jsp+servlet 实现删除表中指定行数据

1、在JSP页面的<a>标签中将href指向对删除业务进行操作的doDeleteServlet，并传递需要删除的行的id

<a href="doDeleteServlet?id=当前行数据的id">删除</a>

<form>  
<table width="95%" border="1" cellpadding="2" cellspacing="1" style="table-layout:fixed;">  
 <caption>用户信息</caption>  
 <tr>  
 <td width="80px" align="center" nowrap>用&nbsp;户&nbsp;id&nbsp;</td>  
 <td width="150px" nowrap>地&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;址</td>  
 <td width="180px" nowrap>&nbsp;email&nbsp;</td>  
 <td width="100px" nowrap>用&nbsp;户&nbsp;名</td>  
 <td width="100px" nowrap>用&nbsp;户&nbsp;密&nbsp;码</td>  
 <td width="100px" nowrap>手&nbsp;机&nbsp;号&nbsp;码</td>  
 <td width="100px" align="center" nowrap>用&nbsp;户&nbsp;类&nbsp;型</td>  
 <td width="80px" align="center" nowrap>操&nbsp;&nbsp;&nbsp;作</td>  
 </tr>  
 <**c:forEach** items="**${**userList**}**" var="users" step="1" varStatus="xh">  
 <tr>  
 <td >**${**users.userId**}**</td>  
 <td>**${**users.address**}**</td>  
 <td>**${**users.email**}**</td>  
 <td>**${**users.userName**}**</td>  
 <td>**${**users.passWd**}**</td>  
 <td>**${**users.phone**}**</td>  
 <td>**${**users.type**}**</td>  
 <td><a href="**<%=**request.getContextPath()**%>**/UserDeleteServlet?id=**${**users.userId**}**">删 除</a></td>  
 </tr>  
 </**c:forEach**>  
  
</table>  
</form>

2、在doDeleteServlet的doGet()方法中，调用dao中的删除方法

3、删除后，调用“取列表的”servlet，该servlet调用dao中的查询方法， 最终转到就最开始的jsp页面

@WebServlet("/UserDeleteServlet")  
public class UserDeleteServlet extends HttpServlet {  
 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 try {  
 System.*out*.println(request.getParameter("id")+"-----------------");  
 int result = UsersDao.*delete*(request.getParameter("id"));  
 if(result>0){  
 System.*out*.println(result+"-----------操作成功");  
 }else{  
 System.*out*.println("操作失败");  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 doPost(request,response);  
 }  
  
 protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 String page = null;  
 try {  
 request.setAttribute("userList", UsersDao.*getAllUsers*());  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 page = "/user/usersForeach.jsp";  
 getServletContext().getRequestDispatcher(page).forward(request,response);  
 }  
}

## 13.多线程并发解决方案（原子变量的使用）

下面是一个没有控制并发的计数器：

1. **public** **class** Counter **implements** Runnable {
2. **private** **static** **int** count;
4. **public** **void** run() {
5. System.out.println(Thread.currentThread().getName()
6. + ":" + (++count));
7. }
9. **public** **static** **void** main(String[] args){
10. Counter counter = **new** Counter();
11. Thread t1 = **new** Thread(counter);
12. Thread t2 = **new** Thread(counter);
13. Thread t3 = **new** Thread(counter);
14. Thread t4 = **new** Thread(counter);
15. t1.start();
16. t2.start();
17. t3.start();
18. t4.start();
19. }
20. }

每次执行输出的结果不一样，如：

1. Thread-1:2
2. Thread-0:1
3. Thread-2:3
4. Thread-3:3

从Java内存模型的角度来看，简单的counter++的执行过程其实分为如下三步：

第一步，从主内存中加载counter的值到线程工作内存

第二步，执行加1运算

第三步，把第二步的执行结果从工作内存写入到主内存

为了预防并发情况的产生，可以使用原子变量java.util.concurrent.atomic包下的类

1. **public** **class** Counter **implements** Runnable {
2. **private** **final** AtomicInteger count = **new** AtomicInteger(0);
4. **public** **void** run() {
5. System.out.println(Thread.currentThread().getName()
6. + ":" + count.incrementAndGet());
7. }
9. **public** **static** **void** main(String[] args){
10. Counter counter = **new** Counter();
11. Thread t1 = **new** Thread(counter);
12. Thread t2 = **new** Thread(counter);
13. Thread t3 = **new** Thread(counter);
14. Thread t4 = **new** Thread(counter);
15. t1.start();
16. t2.start();
17. t3.start();
18. t4.start();
19. }
20. }

其中自增自减方法说明：

incrementAndGet(),返回新值（即加1后的值）

getAndIncrement(),返回旧值（即加1前的原始值）

decrementAndGet(),返回新值（即减1后的值）

getAndDecrement(),返回旧值（即减1前的原始值）

## 14.Java 中的悲观锁和乐观锁的实现

### 锁（locking）

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算

处理中，我们希望针对某个cut-off时间点的数据进行处理，而不希望在结算进行过程中

（可能是几秒种，也可能是几个小时），数据再发生变化。此时，我们就需要通过一些机

制来保证这些数据在某个操作过程中不会被外界修改，这样的机制，在这里，也就是所谓

的“锁”，即给我们选定的目标数据上锁，使其无法被其他程序修改。

### 悲观锁（Pessimistic Locking）

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自

外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定

状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能

真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系

统不会修改数据）。

一个典型的倚赖数据库的悲观锁调用：

select \* from account where name=”Erica” for update

这条sql 语句锁定了account 表中所有符合检索条件（name=”Erica”）的记录。

本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。

### 乐观锁（Optimistic Locking）

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依

靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库

性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是基于数据版本

（Version）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于

数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来

实现。

读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提

交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据

版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

对于上面修改用户帐户信息的例子而言，假设数据库中帐户信息表中有一个

version字段，当前值为1；而当前帐户余额字段（balance）为$100。

1 操作员A 此时将其读出（version=1），并从其帐户余额中扣除$50

（$100-$50）。

2 在操作员A操作的过程中，操作员B也读入此用户信息（version=1），并

从其帐户余额中扣除$20（$100-$20）。

3 操作员A完成了修改工作，将数据版本号加一（version=2），连同帐户扣

除后余额（balance=$50），提交至数据库更新，此时由于提交数据版本大

于数据库记录当前版本，数据被更新，数据库记录version更新为2。

4 操作员B完成了操作，也将版本号加一（version=2）试图向数据库提交数

据（balance=$80），但此时比对数据库记录版本时发现，操作员B提交的

数据版本号为2，数据库记录当前版本也为2，不满足“提交版本必须大于记

录当前版本才能执行更新“的乐观锁策略，因此，操作员B 的提交被驳回。

这样，就避免了操作员B 用基于version=1 的旧数据修改的结果覆盖操作

员A的操作结果的可能。

**悲观锁:**一段执行逻辑加上悲观锁,不同线程同时执行时,只能有一个线程执行,其他的线程在入口处等待,直到锁被释放.

**乐观锁:**一段执行逻辑加上乐观锁,不同线程同时执行时,可以同时进入执行,在最后更新数据的时候要检查这些数据是否被其他线程修改了(版本和执行初是否相同),没有修改则进行更新,否则放弃本次操作.

从解释上可以看出,悲观锁具有很强的独占性,也是最安全的.而乐观锁很开放,效率高,安全性比悲观锁低,因为在乐观锁检查数据版本一致性时也可能被其他线程修改数据.从下面的例子中可以看出来这里说的安全差别.

上面提到的乐观锁的概念中其实已经阐述了它的具体实现细节：主要就是两个步骤：**冲突检测和数据更新。**其实现方式有一种比较典型的就是 Compare and Swap ( CAS )。

CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

　　　　CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置V的值与预期原值A相匹配，那么处理器会自动将该位置值更新为新值B。否则处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“ 我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。 ”这其实和乐观锁的冲突检查+数据更新的原理是一样的。

　JAVA对CAS的支持：

　　　　在JDK1.5 中新增 java.util.concurrent (J.U.C)就是建立在CAS之上的。相对于对于 synchronized 这种阻塞算法，CAS是非阻塞算法的一种常见实现。所以J.U.C在性能上有了很大的提升。

　　　　以 java.util.concurrent 中的 AtomicInteger 为例，看一下在不使用锁的情况下是如何保证线程安全的。主要理解 getAndIncrement 方法，该方法的作用相当于 ++i 操作。

复制代码

1 public class AtomicInteger extends Number implements java.io.Serializable {

2 private volatile int value;

3

4 public final int get() {

5 return value;

6 }

7

8 public final int getAndIncrement() {

9 for (;;) {

10 int current = get();

11 int next = current + 1;

12 if (compareAndSet(current, next))

13 return current;

14 }

15 }

16

17 public final boolean compareAndSet(int expect, int update) {

18 return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

19 }

20 }

复制代码

　　　　在没有锁的机制下,字段value要借助volatile原语，保证线程间的数据是可见性。这样在获取变量的值的时候才能直接读取。然后来看看 ++i 是怎么做到的。

　　　  getAndIncrement 采用了CAS操作，每次从内存中读取数据然后将此数据和 +1 后的结果进行CAS操作，如果成功就返回结果，否则重试直到成功为止。

　　　  而 compareAndSet 利用JNI（Java Native Interface）来完成CPU指令的操作：

1 public final boolean compareAndSet(int expect, int update) {

2 return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

3 }

　　　　其中unsafe.compareAndSwapInt(this, valueOffset, expect, update);类似如下逻辑：

1 if (this == expect) {

2 this = update

3 return true;

4 } else {

5 return false;

6 }

　　　　那么比较this == expect，替换this = update，compareAndSwapInt实现这两个步骤的原子性呢？ 参考CAS的原理

　　CAS原理：

　　　　CAS通过调用JNI的代码实现的。而compareAndSwapInt就是借助C来调用CPU底层指令实现的。