1.Java操作时遇到的并发修改异常

注：java.util.ConcurrentModificationException

|  |
| --- |
| 原因：  1.调用list.remove()方法导致modCount和expectedModCount的值不一致（modCount为AbstractList的成员变量，expectedModCount为Itr类的成员变量，初始值为modCount）  2.for-each进行迭代底层也是用的迭代器，所以也会出现并发修改异常 |
| 单线程下的解决方法： |
| 1.使用普通for循环，记得遍历增加删除时，手动修改当前索引值 |
| 2.private class Itr implements Iterator<E> 使用Itr类提供的remove()方法 |

|  |
| --- |
| 多线程下的解决方法： |
| 1.在使用Iterator迭代的时候使用Synchronized或者lock进行同步 |
| 2.使用并发容器CopyOnwriteArrayList代替ArrayList和vector |





2.Mybatis常用标签与SQL编写注意事项

①常用标签

|  |
| --- |
| 定义SQL语句：select,insert,update,delete |
| 对象属性与结果集列名对应关系：resultMap |
| 动态SQL拼接：if,foreach,choose |
| 格式化输出：where,set,trim |
| 配置关联关系：collection,association |
| 定义常量及引用：sql,include |

②concat()与||使用的区别

Concat():只能连接两个字符串

LIKE CONCAT(CONCAT('%', #{student}),'%')  
||:可以连接多个字符串

LIKE '%'||#{STUDENT}||'%'

③xml文件中特殊符号的处理

=>And t.create\_time &lt;= #{endtime}

|  |
| --- |
| < &lt; |
| > &gt; |
| & &amp; |
| ' &apos; |
| " &quot; |

=><![CDATA[and t.create\_time <= #{endtime}]]>

注：CDATA区全称为character data,

解析器不对CDATA区中的内容进行解析

④Mybatis入参为基本数据类型或者String时

|  |
| --- |
| 一个参数时 #{\_parameter} |
| 多个参数时 #{param1} #{param2} … |
| 标准做法：mapper.java文件中形参添加注解 |
| 如 xxx(@param("name")String username,@param("password")String password){} |
| org.apache.ibatis.annotations.Param |

⑤test="…"中的正确写法（Mybatis使用OGNL表达式解析）

|  |
| --- |
| Test=" name!=null and name!='' " |
| Test =' name!=null and name=="xxx" ' |
| 可以调用java的java.lang.String中定义的方法 |
| Test=' name.indexOf(",")!=-1 ' |

3.操作Oracle数据库已知问题答疑

|  |
| --- |
| 关于in和exists的效率问题 |
| in是把内表和外表做hash连接，exists是对外表做循环处理。 |
| Ⅰselect \* from A where id in (select id from B)--使用in |
| Ⅱselect \* from A where exists(select B.id from B where B.id=A.id) --使用exists |
| Ⅲselect A.\* from A,B where A.id=B.id--不使用in和exists |
| Ⅰ使用的是A表的索引 Ⅱ使用的是B表的索引 Ⅲ同时使用了A，B的索引 |
| A表相对大则使用in，B表相对大则使用exists,Ⅲ虽然同时用了两个表的索引，但是单次扫描次数为笛卡尔积，效率最差 |
| |  | | --- | | in与=的区别 | | where name in('zhang','wang','zhao')<=> where name='zhang' or name='wang' or name='zhao' | | |  | | --- | | not in 与 not exists 的区别 | | 使用not in,内外表都进行全表扫描，没有用到索引; 而not extsts 的子查询依然能用到表上的索引。 所以无论那个表大，用not exists都比not in要快。(not in 的条件中包含null，会导致查询结果为空) | | 注意： *Column <> null*是一个永假式，导致查询结果为空，null正确用法：column is not null. | | |  | | --- | | exists与distinct的用法 | | 当查询中包含的表之间有一对多的关系时，避免在SELECT子句中使用DISTICT，可以使用EXISTS替换(distinct低效，exists高效) | | Exists()一大特点 -> 只要括号中的表达式有一个值存在，就立刻返回true，而不用遍历表中所有的数据 | | Distinct出现的位置为select关键字后面，distinct后面跟多个字段则去重 这几个字段完全相同的行数据。 | | | |

|  |
| --- |
| 延申：数据库在进行多表连接时的常用处理算法 |
| ->hash join(哈希连接) ：将两表中的小表（称S）作为hash表，然后去扫描另一个表(称M)的每一行数据，用得出来的行数据根据连接条件去映射建立的hash表，hash表是放在内存中的，这样可以很快的得到对应的S表与M表相匹配的行 |
| ->nested loop join(嵌套循环连接)：通用的连接方式，分为内外表，每扫描外表的一行数据都要在内表中查找与之相匹配的行 |
| ->sort merge join(排序合并连接)：首先对两个表按照关联的字段进行排序，分别从两个表中取出一行数据进行匹配，如果合适放入结果集；不匹配将较小的那行丢掉继续匹配另一个表的下一行，依次处理直到将两表的数据取完。merge join的很大一部分开销花在排序上，也是同等条件下差于hash join的一个主要原因 |



4.easyui中自定义遮罩层

|  |
| --- |
| 遮罩背景： |
| $("<div class=\"datagrid-mask\"></div>").css({ display: "block", width: "100%", height:$(window).height() }).appendTo("body"); |

|  |
| --- |
| 遮罩居中文字： |
| $("<div class=\"datagrid-mask-msg\"></div>").html("数据加载中,请稍候…").appendTo("body").css({ display: "block", left: ($(document.body).outerWidth(true) - 190) / 2, top: ($(window).height() - 45) / 2 }); |

|  |
| --- |
| 设置遮罩层的堆叠顺序 |
| .css({"z-index":"9999"}); |

5.如何在JS代码中应用EL表达式

|  |
| --- |
| 首先搞明白：EL表达式是用来代替JSP页面中复杂的JAVA代码的，是在服务端解析完成的，而JS是在客户端浏览器解析完成的。服务端代码先于客户端代码执行，所以不能在JS代码中改变已经执行的EL表达式的值，但是EL可以给JS初始化传值。还有重要的一点，如果js脚本不是写在jsp页面中，而是单独的js文件，这时EL表达式初始化传值也是不起作用的。 |
| Ⅰ**js代码写在jsp页面中，此时可以直接使用EL表达式** |
| <script type="text/javascript">   $(function () {   new BacklogOverview("${param.alert}");   });  </script> |
| Ⅱ**js代码写在单独的文件中，JSP通过<script src="xxx"></script>引入js文件** |
| 通过JSP中提前定义JS变量的形式解决（将EL的值赋予本JSP中js变量） |
| <c:set var="contextPath" value="${pageContext.request.contextPath}" scope="application"/>   <script>   <%--JS gloable varilible--%>   var contextPath = "${contextPath}";   </script> |
| 在引入的js文件中可以使用contextPath 间接使用了EL表达式初始化值 |
| ?注意：在js代码中使用EL表达式，一定要用引号将表达式包住(单引号，双引号都可) |
| !解释：EL表达式语句执行时，会调用pageContext.findAttribute方法，用标识符作为关键字，分别从page、request、session、application四个域中查找相应的对象，找到则返回相应的对象，找不到则返回“” |

6.JS中对json的一些容易混淆操作

|  |
| --- |
| Ⅰ将json字符串转为json对象： |
| $.parseJSON( jsonstr ); （jQuery支持的转换方式） |
| JSON.parse(jsonstr)；(浏览器支持的转换方式) |
| eval('(' + jsonstr + ')'); （JavaScript支持的转换方式） |

|  |
| --- |
| Ⅱ 将json对象转为json字符串 |
| JSON.stringify(jsonObj) (浏览器支持的转换方式) |

|  |
| --- |
| Ⅲ java中将json字符串转为java对象 |
| JSONObject jsonObj= JSONObject.fromObject(jsonStr); |
| User user= (User)JSONObject.toBean(jsonObj,User.class); |

|  |
| --- |
| Ⅳ java中将json字符串转为java对象集合 |
| JSONArray jsonArray = JSONArray.fromObject(jsonStr); |
| List<User> users= (List<User>)JSONArray.toCollection(jsonArray, User.class); |

|  |
| --- |
| Ⅴ java中将对象转为json对象，再转为json字符串 |
| JSONObject jsonObject = JSONObject.fromObject(user); |
| String jsonStr = jsonObject.toString(); |

|  |
| --- |
| Ⅵ java中将对象集合转为json对象数组，再转为json字符串 |
| JSONArray array = JSONArray.fromObject(list);  String jsonstr = array.toString(); |

7.JSP的执行原理？

|  |
| --- |
| Ⅰ.Tomcat加载jsp文件时，因为其内置了一个jsp翻译引擎，专门用来翻译jsp文件，产物为java源文件 |
| Ⅱ.index.jsp会被翻译成index\_jsp.java源文件，经编译器编译生成index\_jsp.class类文件 |
| Ⅲ.客户端浏览器访问jsp文件时，tomcat底层执行的时jsp对应的java程序，将所有out.print()和out.write()中的内容写入response响应正文中 |

|  |
| --- |
| 注意：index\_jsp.java这个类继承了HttpJspBase,而HttpJspBase继承了HttpServlet。所以JSP就是Servlet,只不过职责不同，JSP的强项是做页面展示。 |

8.JSP中EL表达式的应用？

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 11个隐式对象： | requestScope | sessionScope | pageScope | applicationScope | pageContext | header | headerValues | param | paramValues | cookie | initParam |

|  |
| --- |
| requestScope <=> request.getAttribute("xxx") |
| sessionScope <=> session.getAttribute("xxx") |
| pageScope <=> pageContext.getAttribute("xxx") |
| applicationScope <=> application.getAttribute("xxx") |
| pageContext <=> 可以获取JSP九大内置对象 eg: pageContext.getRequest() <=> ${pageContext.request) |
| Header <=> request.getHeader("xxx") |
| headerValues <=> request.getHeaders("xxx") |
| Param <=> request.getParameter("xxx") |
| paramValues <=> request.getParameterValues("xxx") |
| Cookie <=> 类型为Map<String,Cookie>，其中key是cookie的name，value是cookie对象，  eg:${cookie.JSESSIONID.value }就是获取sessionId |
| initParam <=> 获取web.xml中<context-param>内的参数,key为<param-name>标签内的值 |
| 注意：11个隐式对象中，除了pageContext类型不是map(PageContext对象)，其它10个对象都是Map类型的 |
| 注意：EL表达式在获取Map的值或Bean的属性值是，可以使用“点”的方法，也可以使用“下标”的方法。但是  Map的键或Bean的属性名中包含下划线或横杠时，那么就必须使用“下标”方法，eg：${initParam['user\_name']} |

9.ES6(ECMAScript6)新特性(http://www.cnblogs.com/Wayou/p/es6\_new\_features.html)

|  |
| --- |
| 箭头操作符 （<https://www.cnblogs.com/fundebug/p/6904753.html）> |
| 类的支持(class关键字) |
| 增强的对象字面量（编写面向对象的js更方便了） |
| 字符串模板(超级有用！拼接字符串时不用一堆+号分割变量了)  eg:var num = Math.random(); console.log(`your number is ${num}`); |
| 解构 |
| 参数默认值，不定参数，拓展参数 |
| let与const关键字 |
| For…of值遍历 |
| Map,Set,WeakMap,WeakSet 4中集合类型 |
| Promises (处理异步操作的一种模式) |

常用：

|  |
| --- |
| a=a?b:c a为true则取b的值，a为false则取c的值 |
| a=a||b a如果不为null或者undefined则取a的值，否则取b的值 等价于a?a:b |
| a=a&&b||c a、b都为真，则取b的值；a、b有一个为假，则取c的值 等价于a?b?b:c:c |

高级：

|  |
| --- |
| call(),apply(),reduce()高阶函数的应用 |