@Retention(RetentionPolicy.*RUNTIME*)

编译器将Annotation储存于class档中，可由VM读入

Activity提供了getCacheDir()和getFilesDir()方法:

getCacheDir()方法用于获取/data/data/pkg/cache目录

getFilesDir()方法用于获取/data/data/pkg/files目录

 public File getDir（String name, int mode）：name目录名称、mode权限，如果传入的目录不存在，系统会创建此目录，路径为"/data/data/程序Package Name/app\_name"，name就是传入的name。  
  
       public File getFileStreamPath（String name）：name为文件名称，所取得的文件必须是放在/data/data/程序  
Package Name/files的文件。  
  
       public File getFilesDir()：取得"/data/data/程序Package Name/files"这个目录的File。  
  
       public FileInputStream openFileInput（String name）：读取文件，name为文件名不能包含路径，文件的路径为"/data/data/程序Package Name/files"，如果文件不存在，将会出现FileNotFoundException。  
  
        public FileOutputStream openFileOutput（String name, int mode）：写文件，name为文件名不能包含路径，mode为权限，文件的路径为"/data/data/程序Package Name/files"。

Json的转化：json-lib-2.2.3-jdk13.jar

1. **import** net.sf.json.JSON;
2. **import** net.sf.json.JSONArray;
3. **import** net.sf.json.JSONObject;
4. **import** net.sf.json.xml.XMLSerializer;
6. /\*\*
7. \* Json串处理工具类
8. \* json,list,map等之间的各种转化
9. \* @author
10. \* @version 2.0 create at,2009-2-12
11. \*
12. \*/
13. **public** **class** JsonUtil {
14. /\*\*
15. \* 将json格式的字符串转化为jsonobject
16. \* @param str
17. \* @return
18. \* @throws ApplicationException
19. \*/
20. **public** **static** JSONObject stringToJsonObject(String str) **throws** ApplicationException {
21. JSONObject jsonObject = **null**;
22. **if** (Validator.isNotNull(str)) {
23. **try**{
24. jsonObject = JSONObject.fromObject(str);
25. }**catch**(Exception ex){
26. **throw** **new** ApplicationException("要转化的字符串格式不对，必须为json格式！");
27. }
28. }
29. **return** jsonObject;
30. }
32. /\*\*
33. \* 从指定的List<map>对象中获取要设置的值后组装成返回给前台的JSON对象字符串.
34. \* 主要用于给前台分页grid组装数据
35. \* @param list    list<map>从数据库中查询出来的数据
36. \* @param count   总数
37. \* @return
38. \*/
39. @SuppressWarnings("unchecked")
40. **public** **static** String fromMap(List list, **int** count) {
41. //将list<map>转化为jsonarray
42. JSONArray options = formListArray(list);
43. JSONObject result = **new** JSONObject();
44. result.put("result", options);
45. result.put("totalCount", count);
46. **return** result.toString();
47. }
49. /\*\*
50. \* 将Map<String,String>转化为JSONObject格式的字符串
51. \* @param map
52. \* @return {"success":true,"result":{}}
53. \*/
54. **public** **static** String fromMapToJson(Map<String, String> map) {
55. Iterator<String> it = map.keySet().iterator();
56. JSONObject jsonObject = **new** JSONObject();
57. JSONObject json = **new** JSONObject();
58. **while** (it.hasNext()) {
59. String key = (String) it.next();
60. jsonObject.put(key.toLowerCase(), map.get(key));
61. }
62. json.put("success", **true**);
63. json.put("result", jsonObject.toString());
64. **return** json.toString();
65. }
67. /\*\*
68. \*  将list<map>转化为jsonobject格式的字符串
69. \* @param list
70. \* @return {"result":{...}}
71. \*/
72. @SuppressWarnings("unchecked")
73. **public** **static** String fromListMap(List list) {
74. JSONArray options = **new** JSONArray();
75. **for** (Object obj : list) {
76. Iterator it = ((Map) obj).keySet().iterator();
77. Object value;
78. JSONObject option = **new** JSONObject();
79. **while** (it.hasNext()) {
80. String key = (String) it.next();
81. value = ((Map) obj).get(key);
82. value = value != **null** ? value : "";
83. option.put(key.toLowerCase(), value);
84. }
85. options.add(option);
86. }
87. JSONObject result = **new** JSONObject();
88. result.put("result", options.toString());
89. **return** result.toString();
90. }
92. /\*\*
93. \* 从list<map>转化为treenode的json
94. \* @param list
95. \* @return
96. \*/
97. @SuppressWarnings("unchecked")
98. **public** **static** String toAsynTreeJson(List list) {
99. JSONArray ja = **new** JSONArray();
100. **for** (Object obj : list) {
101. Iterator it = ((Map) obj).keySet().iterator();
102. JSONObject option = **new** JSONObject();
103. **while** (it.hasNext()) {
104. String key = (String) it.next();
105. Object value = ((Map) obj).get(key);
106. key = key.toLowerCase();
107. value = value == **null** ? "" : value.toString();
108. //是否含有带checkbox的treenode
109. **if** (key.equals("checked")) {
110. **boolean** check = value.equals("1") ? **true** : **false**;
111. option.put("checked", check);
112. }
113. //是为叶子节点还是为非叶子节点
114. **else** **if**("leaf".equals(key)){
115. **boolean** leaf = value.toString().trim().equals("1")?**true**:**false**;
116. option.put("leaf", leaf);
117. }**else** **if**("draggable".equals(key)){
118. **boolean** draggable = value.toString().trim().equals("1")?**true**:**false**;
119. option.put("draggable", draggable);
120. }
121. **else**
122. option.put(key, value);
123. }
124. ja.add(option);
125. }
126. **return** ja.toString();
127. }
129. /\*\*
130. \* 将list<map>转化为jsonarray
131. \* 此JSONArray的格式将会是这样[{},{}]
132. \* @param list
133. \* @return
134. \*/
135. @SuppressWarnings("unchecked")
136. **public** **static** JSONArray formListArray(List list) {
137. JSONArray options = **new** JSONArray();
138. **for** (Object obj : list) {
139. Iterator it = ((Map) obj).keySet().iterator();
140. JSONObject option = **new** JSONObject();
141. **while** (it.hasNext()) {
142. String key = (String) it.next();
143. Object value = ((Map) obj).get(key);
144. value = value != **null** ? value : "";
145. option.put(key.toLowerCase(), value);
146. }
147. options.add(option);
148. }
149. **return** options;
150. }
151. /\*\*
152. \* 从List<Model>转化成jsonArray
153. \* list中装的对象为具体的po
154. \* @param list
155. \* @return
156. \*/
157. @SuppressWarnings("unchecked")
158. **public** **static** JSONArray formListModelArray(List list){
159. JSONArray options = **new** JSONArray();
160. **for** (Object obj : list) {
161. options.add(obj);
162. }
163. **return** options;
164. }
165. /\*\*
166. \* 将JSONArray逆向转化为List  list中可以为任何元素，
167. \* @param json  符合JSONArray格式的字符串
168. \*              可以是很复杂的格式，如[{},[],{a:[],b:{},c:{cc:[]...}},'test']
169. \* @return
170. \*/
171. @SuppressWarnings("unchecked")
172. **public** **static** List JSONArrayConvertList(String json) {
173. ArrayList list = **new** ArrayList();
174. JSONArray ja = JSONArray.fromObject(json);
175. populateArray(ja, list);
176. **return** list;
177. }
179. @SuppressWarnings("unchecked")
180. **private** **static** **void** populateArray(JSONArray jsonArray, List list) {
181. //循环遍历jsonarray
182. **for** (**int** i = 0; i < jsonArray.size(); i++) {
183. **if** (jsonArray.get(i).getClass().equals(JSONArray.**class**)) {  //如果元素是JSONArray类型
184. ArrayList \_list = **new** ArrayList();
185. list.add(\_list);
186. //递归遍历，此为深度遍历，先把最内层的jsonobject给遍历了
187. populateArray(jsonArray.getJSONArray(i), \_list);
188. } **else** **if** (jsonArray.get(i).getClass().equals(JSONObject.**class**)) {  //如果是JSONObject类型
189. HashMap \_map = **new** HashMap();
190. list.add(\_map);
191. //遍历JSONObject
192. populate(jsonArray.getJSONObject(i), \_map);
193. } **else** {  //如果都不是的话就直接加入到list中
194. list.add(jsonArray.get(i));
195. }
196. }
197. }
199. @SuppressWarnings("unchecked")
200. **private** **static** Map populate(JSONObject jsonObject, Map map) {
201. **for** (Iterator iterator = jsonObject.entrySet().iterator(); iterator
202. .hasNext();) {
203. String entryStr = String.valueOf(iterator.next());
204. String key = entryStr.substring(0, entryStr.indexOf("="));
205. **if** (jsonObject.get(key).getClass().equals(JSONObject.**class**)) {
206. HashMap \_map = **new** HashMap();
207. map.put(key, \_map);
208. populate(jsonObject.getJSONObject(key), \_map);
209. } **else** **if** (jsonObject.get(key).getClass().equals(JSONArray.**class**)) {
210. ArrayList list = **new** ArrayList();
211. map.put(key, list);
212. populateArray(jsonObject.getJSONArray(key), list);
213. } **else** {
214. map.put(key, jsonObject.get(key));
215. }
216. }
217. **return** map;
218. }
220. /\*\*
221. \* Map转换成JSONObject
222. \* @param map
223. \* @return
224. \*/
225. @SuppressWarnings("unchecked")
226. **public** JSONObject populateToJsonObject(Map map) {
227. JSONObject temp = **new** JSONObject();
228. Iterator iter = map.entrySet().iterator();
229. **while** (iter.hasNext()) {
230. Entry entry = (Entry) iter.next();
231. String key = entry.getKey().toString();
232. String value = entry.getValue() == **null** ? "" : entry.getValue().toString();
233. temp.put(key, value);
234. }
235. **return** temp;
236. }
237. /\*\*
238. \*将json转化为xml
239. \*\*/
240. **public** **static** String jsonToXml(JSON json,Boolean appendHint){
241. XMLSerializer xmlSerializer = **new** XMLSerializer();
242. xmlSerializer.setTypeHintsEnabled(appendHint);
243. xmlSerializer.setRootName("root");
244. **return** xmlSerializer.write(json);
245. }
246. /\*\*
247. \*将xml转化为json
248. \*\*/
249. **public** **static** JSON xmlToJson(String xml){
250. XMLSerializer xmlSerializer = **new** XMLSerializer();
251. JSON json = xmlSerializer.read(xml);
252. **return** json;
253. }
255. }

正则表达式：

\d 数字：[0-9]   
\D 非数字： [^0-9]   
\s 空白字符：[ \t\n\x0B\f\r]   
\S 非空白字符：[^\s]   
\w 单词字符：[a-zA-Z\_0-9]   
\W 非单词字符：[^\w]

Pattern pattern=Pattern.compile（regEx）;

Martch m = Pattern.martch(string)

**队列：**

**Queue**   
------------   
1.ArrayDeque, （数组双端队列）   
2.PriorityQueue, （优先级队列）   
3.ConcurrentLinkedQueue, （基于链表的并发队列）   
4.DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）   
5.ArrayBlockingQueue, （基于数组的并发阻塞队列）   
6.LinkedBlockingQueue, （基于链表的FIFO阻塞队列）   
7.LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）   
8.PriorityBlockingQueue, （带优先级的无界阻塞队列）   
9.SynchronousQueue （并发同步阻塞队列）   
-----------------------------------------------------