# Java 学习

## 学习文件读取。

**1、实现目标**

　　读取文件，将文件中的数据一行行的取出。

**2、代码实现**

1）、方式1：

　　通过BufferedReader的readLine()方法。

[IMG_256](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

/\*\*

\* 功能：Java读取txt文件的内容 步骤：1：先获得文件句柄 2：获得文件句柄当做是输入一个字节码流，需要对这个输入流进行读取

\* 3：读取到输入流后，需要读取生成字节流 4：一行一行的输出。readline()。 备注：需要考虑的是异常情况

\*

\* @param filePath

\* 文件路径[到达文件:如： D:\aa.txt]

\* @return 将这个文件按照每一行切割成数组存放到list中。

\*/

public static List<String> readTxtFileIntoStringArrList(String filePath)

{

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

try

{

String encoding = "GBK";

File file = new File(filePath);

if (file.isFile() && file.exists())

{ // 判断文件是否存在

InputStreamReader read = new InputStreamReader(

new FileInputStream(file), encoding);// 考虑到编码格式

BufferedReader bufferedReader = new BufferedReader(read);

String lineTxt = null;

while ((lineTxt = bufferedReader.readLine()) != null)

{

list.add(lineTxt);

}

bufferedReader.close();

read.close();

}

else

{

System.out.println("找不到指定的文件");

}

}

catch (Exception e)

{

System.out.println("读取文件内容出错");

e.printStackTrace();

}

return list;

}

[IMG_257](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

2）、方式2

　　通过文件byte数组暂存文件中内容，将其转换为String数据，再按照 “回车换行” 进行分割。

[IMG_258](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

/\*\*

\* 读取filePath的文件，将文件中的数据按照行读取到String数组中

\* @param filePath 文件的路径

\* @return 文件中一行一行的数据

\*/

public static String[] readToString(String filePath)

{

File file = new File(filePath);

Long filelength = file.length(); // 获取文件长度

byte[] filecontent = new byte[filelength.intValue()];

try

{

FileInputStream in = new FileInputStream(file);

in.read(filecontent);

in.close();

} catch (FileNotFoundException e)

{

e.printStackTrace();

} catch (IOException e)

{

e.printStackTrace();

}

String[] fileContentArr = new String(filecontent).split("\r\n");

return fileContentArr;// 返回文件内容,默认编码

}

[IMG_259](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

3）、测试

[IMG_260](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

public static void main(String[] args)

{

List<String> stringList = readTxtFileIntoStringArrList("C:\\soft\\java\\tomcat\\apache-tomcat-7.0.40\\webapps\\appDataGenerate\\log4j\\lepai\_recognize\_cache.log");

System.out.println("-------使用BufferedReader读取-----------");

for(String str : stringList)

{

System.out.println(str);

}

System.out.println("\n---------使用byte直接缓存整个文件到内存----------------");

String[] stringArr = readToString("C:\\soft\\java\\tomcat\\apache-tomcat-7.0.40\\webapps\\appDataGenerate\\log4j\\lepai\_recognize\_cache.log");

for(int i = 0 ; i < stringArr.length ; i ++)

{

System.out.println(stringArr[i]);

}

}

[IMG_261](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

结果：

[IMG_262](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

-------使用BufferedReader读取-----------

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-12-01 14:52:04] [RecognizeCache] [INFO] : 读取文件：4209bad42de0f6e55c0daf0bd24b635a.txt

---------使用byte直接缓存整个文件到内存----------------

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-11-30 13:21:28] [RecognizeCache] [INFO] : RecogizeCache init

[2015-12-01 14:52:04] [RecognizeCache] [INFO] : 读取文件：4209bad42de0f6e55c0daf0bd24b635a.txt

[IMG_263](http://www.cnblogs.com/0201zcr/p/javascript:void(0);" \o "复制代码)

**3、比较**

　　方式1是将文件的**一部分或全部**数据读取出来用BufferReader缓存起来，需要再冲缓存中取数据，这样比要得时候去文件中读取要快一些。

　　方式2是一次把文本的原始内容直接读取到内存中再做处理（暂时不考虑内存大小），这样做效率也会提高。同时，可以处理当你使用第1方式用readLine()方法时，文件又有线程在不断的向文件中写数据【只处理现在已经在文件中的数据】。另外，用readline()之类的方法，可能需要反复访问文件，而且每次readline()都会调用编码转换，降低了速度，所以，在已知编码的情况下，按字节流方式先将文件都读入内存，再一次性编码转换是最快的方式。

　　有错误的希望大牛不吝赐教。 想了解一下，

　　1、通过ftp取一个文件到本地，我如何判断对方的文件是否已经写完了。

　　2、当我使用上面的BufferedReader的readLine()方法一行行读取文件的时候，我还向文件中添加数据，会不会出现文件读取结束不了的情况。

源码下载：

1. 概念

文件读取 1）按字节读取 2）按字符读取 3）俺行读取 4）随机读取

文件读取 Reader FileReader InputStreamReader(字节转化的桥梁) BufferedReader

字节流处理

InputStream

OutpuStream

对应的子类为 FileInputStream FileOutputStream

处理字符流的抽象类

InPutStreamReader

OutpuStreamWrite

字符流向字节流的转化

BufferedReader

BufferedWrite

BufferedReader 由Reader类扩展而来，提供通用的缓冲方式文本读取，readLine读取一个文本行，

从字符输入流中读取文本，缓冲各个字符，从而提供字符、数组和行的高效读取。

BufferedWriter  由Writer 类扩展而来，提供通用的缓冲方式文本写入， newLine使用平台自己的行分隔符，

将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

是所有字节输入所有类的超类 提供字节流的读取而非文本的读取 读取出来是byte

Reader 是读取字符流的抽象类 读取出来的是char【】 或者String

1. Api

FileInputStream 从文件系统中的某个文件中获得输入字节。  
构造方法摘要    
FileInputStream (File  file)   
          通过打开一个到实际文件的连接来创建一个 FileInputStream ，该文件通过文件系统中的 File 对象 file 指定。   
FileInputStream (FileDescriptor  fdObj)   
          通过使用文件描述符 fdObj 创建一个 FileInputStream ，该文件描述符表示到文件系统中某个实际文件的现有连接。   
FileInputStream (String  name)   
          通过打开一个到实际文件的连接来创建一个 FileInputStream ，该文件通过文件系统中的路径名 name 指定。   
   
   
Reader  
  
   |——BufferedReader   
   |\_\_\_InputStreamReader   
         |\_\_FileReader   
  
  
BufferedReader : 从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。  
   
构造方法摘要    
BufferedReader (Reader  in)   
          创建一个使用默认大小输入缓冲区的缓冲字符输入流。   
BufferedReader (Reader  in, int sz)   
          创建一个使用指定大小输入缓冲区的缓冲字符输入流。   
BufferedReader (Java Platform SE 6)   
BufferedReader的最大特点就是缓冲区的设置。通常Reader 所作的每个读取请求都会导致对底层字符或字节流进行相应的读取请求，如果没有缓冲，则每次调用 read() 或 readLine() 都会导致从文件中读取字节，并将其转换为字符后返回，而这是极其低效的。   
使用BufferedReader可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。   
因此，建议用 BufferedReader 包装所有其 read() 操作可能开销很高的 Reader（如 FileReader 和InputStreamReader）。例如，   
 BufferedReader in  
   = new BufferedReader(new FileReader("foo.in"));  
 将缓冲指定文件的输入。   
InputStreamReader (Java Platform SE 6)   
InputStreamReader 是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。   
   
构造方法摘要    
InputStreamReader (InputStream  in)   
          创建一个使用默认字符集的 InputStreamReader。   
InputStreamReader (InputStream  in, Charset  cs)   
          创建使用给定字符集的 InputStreamReader。   
InputStreamReader (InputStream  in, CharsetDecoder  dec)   
          创建使用给定字符集解码器的 InputStreamReader。   
InputStreamReader (InputStream  in, String  charsetName)   
          创建使用指定字符集的 InputStreamReader。

每次调用 InputStreamReader 中的一个 read() 方法都会导致从底层输入流读取一个或多个字节。要启用从字节到字符的有效转换，可以提前从底层流读取更多的字节，使其超过满足当前读取操作所需的字节。   
为了达到最高效率，可要考虑在 BufferedReader 内包装 InputStreamReader。例如：   
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));  
InputStreamReader最大的特点是可以指转换的定编码格式  
，这是其他类所不能的，从构造方法就可看出，  
这一点在读取中文字符时非常有用

FileReader  
1）FileReader类介绍：  
InputStreamReader类的子类，所有方法（read（）等）都从父类InputStreamReader中继承而来；  
2）与InputStreamReader类的区别：  
构造方法摘要    
FileReader (File  file)   
          在给定从中读取数据的 File 的情况下创建一个新 FileReader 。   
FileReader (FileDescriptor  fd)   
          在给定从中读取数据的 FileDescriptor 的情况下创建一个新 FileReader 。   
FileReader (String  fileName)   
          在给定从中读取数据的文件名的情况下创建一个新 FileReader    
  
该类与它的父类InputStreamReader的主要不同在于构造函数，主要区别也就在于构造函数！  
从InputStreamReader的构造函数中看到，参数为InputStream和编码方式，可以看出，  
当要指定编码方式时，必须使用InputStreamReader  
类；而FileReader构造函数的参数与FileInputStream同，为File对象或表示path的String，可以看出，当要根据File对象或者String读取一个文件时，用FileReader；  
我想FileReader子类的作用也就在于这个小分工吧。该类与它的父类InputStreamReader  
的主要不同在于构造函数，主要区别也就在于构造函数！  
从InputStreamReader  
的构造函数中看到，参数为InputStream和编码方式，可以看出，  
当要指定编码方式时，必须使用InputStreamReader  
类；而FileReader构造函数的参数与FileInputStream  
同，为File对象或表示path的String，可以看出，当要根据File对象或者String读取一个文件时，用FileReader；  
我想FileReader子类的作用也就在于这个小分工吧。

1. 用法
2. **import** java.io.File;
3. **import** java.io.FileReader;
4. **import** java.io.IOException;
5. /\*\*
6. \* @description 本类实现了按行读取文件内容到内存，并打印输出的操作
7. \* @time 2010-3-2
8. \*/
9. **public** **class** readFile {
10. **public** **static** **void** main(String []arg0)
11. {
12. **try**{
13. **new** readFile("d:/test.txt");
14. }**catch** (IOException e)
15. {
16. e.printStackTrace();
17. }
18. }
19. /\*\*
20. \* @param file：文件的全路径名
21. \* @throws IOException
22. \*/
23. **public** readFile(String file) **throws** IOException
24. {
25. File f=**new** File(file);
26. BufferedReader br=**new** BufferedReader(**new** FileReader(f));
27. String line=**null**;
28. **while** ((line = br.readLine()) != **null**) {
29. {
30. System.out.println(line);
31. }
32. br.close();
33. }
34. }
35. }
36. //------------------参考资料---------------------------------
37. //
38. //1、按字节读取文件内容
39. //2、按字符读取文件内容
40. //3、按行读取文件内容
41. //4、随机读取文件内容
43. **import** java.io.BufferedReader;
44. **import** java.io.File;
45. **import** java.io.FileInputStream;
46. **import** java.io.FileReader;
47. **import** java.io.IOException;
48. **import** java.io.InputStream;
49. **import** java.io.InputStreamReader;
50. **import** java.io.RandomAccessFile;
51. **import** java.io.Reader;
53. **public** **class** ReadFromFile {
54. /\*\*
55. \* 以字节为单位读取文件，常用于读二进制文件，如图片、声音、影像等文件。
56. \*
57. \* @param fileName
58. \*            文件的名
59. \*/
60. **public** **static** **void** readFileByBytes(String fileName) {
61. File file = **new** File(fileName);
62. InputStream in = **null**;
63. **try** {
64. System.out.println("以字节为单位读取文件内容，一次读一个字节：");
65. // 一次读一个字节
66. in = **new** FileInputStream(file);
67. **int** tempbyte;
68. **while** ((tempbyte = in.read()) != -1) {
69. System.out.write(tempbyte);
70. }
71. in.close();
72. } **catch** (IOException e) {
73. e.printStackTrace();
74. **return**;
75. }
76. **try** {
77. System.out.println("以字节为单位读取文件内容，一次读多个字节：");
78. // 一次读多个字节
79. **byte**[] tempbytes = **new** **byte**[100];
80. **int** byteread = 0;
81. in = **new** FileInputStream(fileName);
82. ReadFromFile.showAvailableBytes(in);
83. // 读入多个字节到字节数组中，byteread为一次读入的字节数
84. **while** ((byteread = in.read(tempbytes)) != -1) {
85. System.out.write(tempbytes, 0, byteread);
86. }
87. } **catch** (Exception e1) {
88. e1.printStackTrace();
89. } **finally** {
90. **if** (in != **null**) {
91. **try** {
92. in.close();
93. } **catch** (IOException e1) {
94. }
95. }
96. }
97. }
99. /\*\*
100. \* 以字符为单位读取文件，常用于读文本，数字等类型的文件
101. \*
102. \* @param fileName
103. \*            文件名
104. \*/
105. **public** **static** **void** readFileByChars(String fileName) {
106. File file = **new** File(fileName);
107. Reader reader = **null**;
108. **try** {
109. System.out.println("以字符为单位读取文件内容，一次读一个字节：");
110. // 一次读一个字符
111. reader = **new** InputStreamReader(**new** FileInputStream(file));
112. **int** tempchar;
113. **while** ((tempchar = reader.read()) != -1) {
114. // 对于windows下，rn这两个字符在一起时，表示一个换行。
115. // 但如果这两个字符分开显示时，会换两次行。
116. // 因此，屏蔽掉r，或者屏蔽n。否则，将会多出很多空行。
117. **if** (((**char**) tempchar) != 'r') {
118. System.out.print((**char**) tempchar);
119. }
120. }
121. reader.close();
122. } **catch** (Exception e) {
123. e.printStackTrace();
124. }
125. **try** {
126. System.out.println("以字符为单位读取文件内容，一次读多个字节：");
127. // 一次读多个字符
128. **char**[] tempchars = **new** **char**[30];
129. **int** charread = 0;
130. reader = **new** InputStreamReader(**new** FileInputStream(fileName));
131. // 读入多个字符到字符数组中，charread为一次读取字符数
132. **while** ((charread = reader.read(tempchars)) != -1) {
133. // 同样屏蔽掉r不显示
134. **if** ((charread == tempchars.length)
135. && (tempchars[tempchars.length - 1] != 'r')) {
136. System.out.print(tempchars);
137. } **else** {
138. **for** (**int** i = 0; i < charread; i++) {
139. **if** (tempchars[i] == 'r') {
140. **continue**;
141. } **else** {
142. System.out.print(tempchars[i]);
143. }
144. }
145. }
146. }
147. } **catch** (Exception e1) {
148. e1.printStackTrace();
149. } **finally** {
150. **if** (reader != **null**) {
151. **try** {
152. reader.close();
153. } **catch** (IOException e1) {
154. }
155. }
156. }
157. }
159. /\*\*
160. \* 以行为单位读取文件，常用于读面向行的格式化文件
161. \*
162. \* @param fileName
163. \*            文件名
164. \*/
165. **public** **static** **void** readFileByLines(String fileName) {
166. File file = **new** File(fileName);
167. BufferedReader reader = **null**;
168. **try** {
169. System.out.println("以行为单位读取文件内容，一次读一整行：");
170. reader = **new** BufferedReader(**new** FileReader(file));
171. String tempString = **null**;
172. **int** line = 1;
173. // 一次读入一行，直到读入null为文件结束
174. **while** ((tempString = reader.readLine()) != **null**) {
175. // 显示行号
176. System.out.println("line " + line + ": " + tempString);
177. line++;
178. }
179. reader.close();
180. } **catch** (IOException e) {
181. e.printStackTrace();
182. } **finally** {
183. **if** (reader != **null**) {
184. **try** {
185. reader.close();
186. } **catch** (IOException e1) {
187. }
188. }
189. }
190. }
192. /\*\*
193. \* 随机读取文件内容
194. \*
195. \* @param fileName
196. \*            文件名
197. \*/
198. **public** **static** **void** readFileByRandomAccess(String fileName) {
199. RandomAccessFile randomFile = **null**;
200. **try** {
201. System.out.println("随机读取一段文件内容：");
202. // 打开一个随机访问文件流，按只读方式
203. randomFile = **new** RandomAccessFile(fileName, "r");
204. // 文件长度，字节数
205. **long** fileLength = randomFile.length();
206. // 读文件的起始位置
207. **int** beginIndex = (fileLength > 4) ? 4 : 0;
208. // 将读文件的开始位置移到beginIndex位置。
209. randomFile.seek(beginIndex);
210. **byte**[] bytes = **new** **byte**[10];
211. **int** byteread = 0;
212. // 一次读10个字节，如果文件内容不足10个字节，则读剩下的字节。
213. // 将一次读取的字节数赋给byteread
214. **while** ((byteread = randomFile.read(bytes)) != -1) {
215. System.out.write(bytes, 0, byteread);
216. }
217. } **catch** (IOException e) {
218. e.printStackTrace();
219. } **finally** {
220. **if** (randomFile != **null**) {
221. **try** {
222. randomFile.close();
223. } **catch** (IOException e1) {
224. }
225. }
226. }
227. }
229. /\*\*
230. \* 显示输入流中还剩的字节数
231. \*
232. \* @param in
233. \*/
234. **private** **static** **void** showAvailableBytes(InputStream in) {
235. **try** {
236. System.out.println("当前字节输入流中的字节数为:" + in.available());
237. } **catch** (IOException e) {
238. e.printStackTrace();
239. }
240. }
242. **public** **static** **void** main(String[] args) {
243. String fileName = "C:/temp/newTemp.txt";
244. ReadFromFile.readFileByBytes(fileName);
245. ReadFromFile.readFileByChars(fileName);
246. ReadFromFile.readFileByLines(fileName);
247. ReadFromFile.readFileByRandomAccess(fileName);
248. }
249. }
250. //二、将内容追加到文件尾部
251. **import** java.io.FileWriter;
252. **import** java.io.IOException;
253. **import** java.io.RandomAccessFile;
255. /\*\*
256. \* 将内容追加到文件尾部
257. \*/
258. **public** **class** AppendToFile {
259. /\*\*
260. \* A方法追加文件：使用RandomAccessFile
261. \*
262. \* @param fileName
263. \*            文件名
264. \* @param content
265. \*            追加的内容
266. \*/
267. **public** **static** **void** appendMethodA(String fileName,
269. String content) {
270. **try** {
271. // 打开一个随机访问文件流，按读写方式
272. RandomAccessFile randomFile = **new** RandomAccessFile(fileName, "rw");
273. // 文件长度，字节数
274. **long** fileLength = randomFile.length();
275. // 将写文件指针移到文件尾。
276. randomFile.seek(fileLength);
277. randomFile.writeBytes(content);
278. randomFile.close();
279. } **catch** (IOException e) {
280. e.printStackTrace();
281. }
282. }
284. /\*\*
285. \* B方法追加文件：使用FileWriter
286. \*
287. \* @param fileName
288. \* @param content
289. \*/
290. **public** **static** **void** appendMethodB(String fileName, String content) {
291. **try** {
292. // 打开一个写文件器，构造函数中的第二个参数true表示以追加形式写文件
293. FileWriter writer = **new** FileWriter(fileName, **true**);
294. writer.write(content);
295. writer.close();
296. } **catch** (IOException e) {
297. e.printStackTrace();
298. }
299. }
301. **public** **static** **void** main(String[] args) {
302. String fileName = "C:/temp/newTemp.txt";
303. String content = "new append!";
304. // 按方法A追加文件
305. AppendToFile.appendMethodA(fileName, content);
306. AppendToFile.appendMethodA(fileName, "append end. n");
307. // 显示文件内容
308. ReadFromFile.readFileByLines(fileName);
309. // 按方法B追加文件
310. AppendToFile.appendMethodB(fileName, content);
311. AppendToFile.appendMethodB(fileName, "append end. n");
312. // 显示文件内容
313. ReadFromFile.readFileByLines(fileName);
314. }
315. }

1. File file=**new** File(path+filename);
2. **if**(!file.exists())
3. {
4. **try** {
5. file.createNewFile();
6. } **catch** (IOException e) {
7. // TODO Auto-generated catch block
8. e.printStackTrace();
9. }
10. }

### 判断文件夹是否存在，不存在创建文件夹

**[java]** [view plain](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "view plain" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank) [copy](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "copy" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. File file =**new** File(path+filename);
2. //如果文件夹不存在则创建
3. **if**  (!file .exists())
4. {
5. file .mkdir();
6. }

### [java 写文件的三种方法比较](http://www.cnblogs.com/yezhenhan/archive/2012/09/10/2678690.html" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

**[java]** [view plain](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "view plain" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank) [copy](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "copy" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **import** java.io.File;
3. **import** java.io.FileOutputStream;
5. **import** java.io.\*;
7. **public** **class** FileTest {
9. **public** FileTest() {
11. }
13. **public** **static** **void** main(String[] args) {
15. FileOutputStream out = **null**;
17. FileOutputStream outSTr = **null**;
19. BufferedOutputStream Buff=**null**;
21. FileWriter fw = **null**;
23. **int** count=1000;//写文件行数
25. **try** {
27. out = **new** FileOutputStream(**new** File(“C:/add.txt”));
29. **long** begin = System.currentTimeMillis();
31. **for** (**int** i = 0; i < count; i++) {
33. out.write(“测试java 文件操作\r\n”.getBytes());
35. }
37. out.close();
39. **long** end = System.currentTimeMillis();
41. System.out.println(“FileOutputStream执行耗时:” + (end - begin) + ” 豪秒”);
43. outSTr = **new** FileOutputStream(**new** File(“C:/add0.txt”));
45. Buff=**new** BufferedOutputStream(outSTr);
47. **long** begin0 = System.currentTimeMillis();
49. **for** (**int** i = 0; i < count; i++) {
51. Buff.write(“测试java 文件操作\r\n”.getBytes());
53. }
55. Buff.flush();
57. Buff.close();
59. **long** end0 = System.currentTimeMillis();
61. System.out.println(“BufferedOutputStream执行耗时:” + (end0 - begin0) + ” 豪秒”);
63. fw = **new** FileWriter(“C:/add2.txt”);
65. **long** begin3 = System.currentTimeMillis();
67. **for** (**int** i = 0; i < count; i++) {
69. fw.write(“测试java 文件操作\r\n”);
71. }
73. fw.close();
75. **long** end3 = System.currentTimeMillis();
77. System.out.println(“FileWriter执行耗时:” + (end3 - begin3) + ” 豪秒”);
79. } **catch** (Exception e) {
81. e.printStackTrace();
83. }
85. **finally** {
87. **try** {
89. fw.close();
91. Buff.close();
93. outSTr.close();
95. out.close();
97. } **catch** (Exception e) {
99. e.printStackTrace();
101. }
103. }
105. }
107. }

### [java 写文件的三种方法比较](http://www.cnblogs.com/yezhenhan/archive/2012/09/10/2678690.html" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

**[java]** [view plain](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "view plain" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank) [copy](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "copy" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **import** java.io.File;
3. **import** java.io.FileOutputStream;
5. **import** java.io.\*;
7. **public** **class** FileTest {
9. **public** FileTest() {
11. }
13. **public** **static** **void** main(String[] args) {
15. FileOutputStream out = **null**;
17. FileOutputStream outSTr = **null**;
19. BufferedOutputStream Buff=**null**;
21. FileWriter fw = **null**;
23. **int** count=1000;//写文件行数
25. **try** {
27. out = **new** FileOutputStream(**new** File(“C:/add.txt”));
29. **long** begin = System.currentTimeMillis();
31. **for** (**int** i = 0; i < count; i++) {
33. out.write(“测试java 文件操作\r\n”.getBytes());
35. }
37. out.close();
39. **long** end = System.currentTimeMillis();
41. System.out.println(“FileOutputStream执行耗时:” + (end - begin) + ” 豪秒”);
43. outSTr = **new** FileOutputStream(**new** File(“C:/add0.txt”));
45. Buff=**new** BufferedOutputStream(outSTr);
47. **long** begin0 = System.currentTimeMillis();
49. **for** (**int** i = 0; i < count; i++) {
51. Buff.write(“测试java 文件操作\r\n”.getBytes());
53. }
55. Buff.flush();
57. Buff.close();
59. **long** end0 = System.currentTimeMillis();
61. System.out.println(“BufferedOutputStream执行耗时:” + (end0 - begin0) + ” 豪秒”);
63. fw = **new** FileWriter(“C:/add2.txt”);
65. **long** begin3 = System.currentTimeMillis();
67. **for** (**int** i = 0; i < count; i++) {
69. fw.write(“测试java 文件操作\r\n”);
71. }
73. fw.close();
75. **long** end3 = System.currentTimeMillis();
77. System.out.println(“FileWriter执行耗时:” + (end3 - begin3) + ” 豪秒”);
79. } **catch** (Exception e) {
81. e.printStackTrace();
83. }
85. **finally** {
87. **try** {
89. fw.close();
91. Buff.close();
93. outSTr.close();
95. out.close();
97. } **catch** (Exception e) {
99. e.printStackTrace();
101. }
103. }
105. }
107. }

[IMG_256](http://blog.csdn.net/jiangxinyu/article/details/7885518/javascript:;" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

### [java中的getParentFile](http://blog.csdn.net/cys1991/article/details/7593790" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

String name = "AAAA.txt";  
String lujing = "1"+"/"+"2";//定义路径  
File a = new File(lujing,name);

a.getParentFile().mkdirs();    //这里如果不加getParentFile(),创建的文件夹为"1/2/AAAA.txt/"

那么，a的意义就是“1/2/AAAA.txt”。  
  
这里a是File，但是File这个类在**[Java](http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)**里表示的不只是文件，虽然File在英语里是文件的意思。Java里，File至少可以表示文件或文件夹（大概还有可以表示系统设备什么的，这里不考虑，只考虑文件和文件夹）。  
  
也就是说，在“1/2/AAAA.txt”真正出现在磁盘结构里之前，它既可以表示这个文件，也可以表示这个路径的文件夹。那么，如果没有getParentFile(),直接执行a.mkdirs()，就是说，创建“1/2/AAAA.txt”代表的文件夹，也就是“1/2/AAAA.txt/”，在此之后，执行a.createNewFile()，试图创建a文件，然而以a为名的文件夹已经存在了，所以createNewFile()实际是执行失败的。你可以用System.out.println(a.createNewFile())这样来检查是不是真正创建文件成功。  
  
所以，这里，你想要创建的是“1/2/AAAA.txt”这个文件。在创建AAAA.txt之前，必须要1/2这个目录存在。所以，要得到1/2，就要用a.getParentFile()，然后要创建它，也就是a.getParentFile().mkdirs()。在这之后，a作为文件所需要的文件夹大概会存在了（有特殊情况会无法创建的，这里不考虑），就执行a.createNewFile()创建a文件。

### Java RandomAccessFile的使用

Java的RandomAccessFile提供对文件的读写功能，与普通的输入输出流不一样的是RamdomAccessFile可以任意的访问文件的任何地方。这就是“Random”的意义所在。

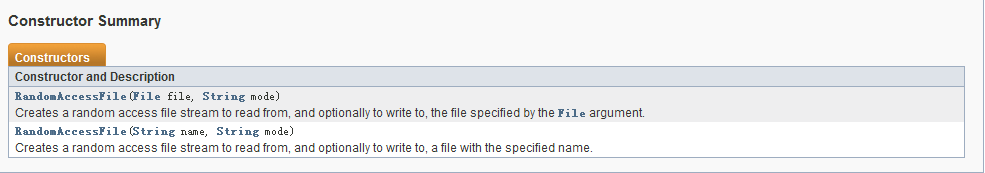
RandomAccessFile的对象包含一个记录指针，用于标识当前流的读写位置，这个位置可以向前移动，也可以向后移动。RandomAccessFile包含两个方法来操作文件记录指针。

long getFilePoint():记录文件指针的当前位置。

void seek(long pos):将文件记录指针定位到pos位置。

RandomAccessFile包含InputStream的三个read方法，也包含OutputStream的三个write方法。同时RandomAccessFile还包含一系列的readXxx和writeXxx方法完成输入输出。

RandomAccessFile的构造方法如下



mode的值有四个

"r":以只读文方式打开指定文件。如果你写的话会有IOException。

"rw":以读写方式打开指定文件，不存在就创建新文件。

"rws":不介绍了。

"rwd":也不介绍。

**[java]** [view plain](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "view plain" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank) [copy](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "copy" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. /\*\*
2. \* 往文件中依次写入3名员工的信息，
3. \* 每位员工有姓名和员工两个字段 然后按照
4. \* 第二名，第一名，第三名的先后顺序读取员工信息
5. \*/
6. **import** java.io.File;
7. **import** java.io.RandomAccessFile;
9. **public** **class** RandomAccessFileTest {
10. **public** **static** **void** main(String[] args) **throws** Exception {
11. Employee e1 = **new** Employee(23, "张三");
12. Employee e2 = **new** Employee(24, "lisi");
13. Employee e3 = **new** Employee(25, "王五");
14. File file = **new** File("employee.txt");
15. **if** (!file.exists()) {
16. file.createNewFile();
17. }
18. // 一个中文占两个字节 一个英文字母占一个字节
19. // 整形 占的字节数目 跟cpu位长有关 32位的占4个字节
20. RandomAccessFile randomAccessFile = **new** RandomAccessFile(file, "rw");
21. randomAccessFile.writeChars(e1.getName());
22. randomAccessFile.writeInt(e1.getAge());
23. randomAccessFile.writeChars(e2.getName());
24. randomAccessFile.writeInt(e2.getAge());
25. randomAccessFile.writeChars(e3.getName());
26. randomAccessFile.writeInt(e3.getAge());
27. randomAccessFile.close();
29. RandomAccessFile raf2 = **new** RandomAccessFile(file, "r");
30. raf2.skipBytes(Employee.LEN \* 2 + 4);
31. String strName2 = "";
32. **for** (**int** i = 0; i < Employee.LEN; i++) {
33. strName2 = strName2 + raf2.readChar();
34. }
35. **int** age2 = raf2.readInt();
36. System.out.println("strName2 = " + strName2.trim());
37. System.out.println("age2 = " + age2);
39. raf2.seek(0);
40. String strName1 = "";
41. **for** (**int** i = 0; i < Employee.LEN; i++) {
42. strName1 = strName1 + raf2.readChar();
43. }
44. **int** age1 = raf2.readInt();
45. System.out.println("strName1 = " + strName1.trim());
46. System.out.println("age1 = " + age1);
48. raf2.skipBytes(Employee.LEN \* 2 + 4);
49. String strName3 = "";
50. **for** (**int** i = 0; i < Employee.LEN; i++) {
51. strName3 = strName3 + raf2.readChar();
52. }
53. **int** age3 = raf2.readInt();
54. System.out.println("strName3 = " + strName3.trim());
55. System.out.println("age3 = " + age3);
56. }
57. }
59. **class** Employee {
60. // 年龄
61. **public** **int** age;
62. // 姓名
63. **public** String name;
64. // 姓名的长度
65. **public** **static** **final** **int** LEN = 8;
67. **public** Employee(**int** age, String name) {
68. **this**.age = age;
70. // 对name字符长度的一个处理
71. **if** (name.length() > LEN) {
72. name = name.substring(0, LEN);
73. } **else** {
74. **while** (name.length() < LEN) {
75. name = name + "/u0000";
76. }
77. }
78. **this**.name = name;
79. }
81. **public** **int** getAge() {
82. **return** age;
83. }
85. **public** String getName() {
86. **return** name;
87. }
89. }

### [高效的RandomAccessFile](http://zhang-xiujiao.iteye.com/blog/1150751" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

[http://zhang-xiujiao.iteye.com/blog/1150751](http://zhang-xiujiao.iteye.com/blog/1150751" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

**主体：**

RandomAccessFile类。其I/O性能较之其它常用开发语言的同类性能差距甚远，严重影响程序的运行效率。

开发人员迫切需要提高效率，下面分析RandomAccessFile等文件类的源代码，找出其中的症结所在，并加以改进优化，创建一个"性/价比"俱佳的随机文件访问类BufferedRandomAccessFile。

在改进之前先做一个基本**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)**：逐字节COPY一个12兆的文件（这里牵涉到读和写）。

|  |  |  |
| --- | --- | --- |
| 读 | 写 | 耗用时间（秒） |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |

我们可以看到两者差距约32倍，RandomAccessFile也太慢了。先看看两者关键部分的源代码，对比分析，找出原因。

**1．1．[RandomAccessFile]**

Java代码  [IMG_258](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **class** RandomAccessFile **implements** DataOutput, DataInput {
2. **public** **final** **byte** readByte() **throws** IOException {
3. **int** ch = **this**.read();
4. **if** (ch < 0)
5. **throw** **new** EOFException();
6. **return** (**byte**)(ch);
7. }
8. **public** **native** **int** read() **throws** IOException;
9. **public** **final** **void** writeByte(**int** v) **throws** IOException {
10. write(v);
11. }
12. **public** **native** **void** write(**int** b) **throws** IOException;
13. }

可见，RandomAccessFile每读/写一个字节就需对磁盘进行一次I/O操作。

**1．2．[BufferedInputStream]**

Java代码  [IMG_259](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **class** BufferedInputStream **extends** FilterInputStream {
2. **private** **static** **int** defaultBufferSize = 2048;
3. **protected** **byte** buf[]; // 建立读缓存区
4. **public** BufferedInputStream(InputStream in, **int** size) {
5. **super**(in);
6. **if** (size <= 0) {
7. **throw** **new** IllegalArgumentException("Buffer size <= 0");
8. }
9. buf = **new** **byte**[size];
10. }
11. **public** **synchronized** **int** read() **throws** IOException {
12. ensureOpen();
13. **if** (pos >= count) {
14. fill();
15. **if** (pos >= count)
16. **return** -1;
17. }
18. **return** buf[pos++] & 0xff; // 直接从BUF[]中读取
19. }
20. **private** **void** fill() **throws** IOException {
21. **if** (markpos < 0)
22. pos = 0;        /\* no mark: throw away the buffer \*/
23. **else** **if** (pos >= buf.length)  /\* no room left in buffer \*/
24. **if** (markpos > 0) {   /\* can throw away early part of the buffer \*/
25. **int** sz = pos - markpos;
26. System.arraycopy(buf, markpos, buf, 0, sz);
27. pos = sz;
28. markpos = 0;
29. } **else** **if** (buf.length >= marklimit) {
30. markpos = -1;   /\* buffer got too big, invalidate mark \*/
31. pos = 0;    /\* drop buffer contents \*/
32. } **else** {        /\* grow buffer \*/
33. **int** nsz = pos \* 2;
34. **if** (nsz > marklimit)
35. nsz = marklimit;
36. **byte** nbuf[] = **new** **byte**[nsz];
37. System.arraycopy(buf, 0, nbuf, 0, pos);
38. buf = nbuf;
39. }
40. count = pos;
41. **int** n = in.read(buf, pos, buf.length - pos);
42. **if** (n > 0)
43. count = n + pos;
44. }
45. }

**1．3．[BufferedOutputStream]**

Java代码  [IMG_260](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **class** BufferedOutputStream **extends** FilterOutputStream {
2. **protected** **byte** buf[]; // 建立写缓存区
3. **public** BufferedOutputStream(OutputStream out, **int** size) {
4. **super**(out);
5. **if** (size <= 0) {
6. **throw** **new** IllegalArgumentException("Buffer size <= 0");
7. }
8. buf = **new** **byte**[size];
9. }
10. **public** **synchronized** **void** write(**int** b) **throws** IOException {
11. **if** (count >= buf.length) {
12. flushBuffer();
13. }
14. buf[count++] = (**byte**)b; // 直接从BUF[]中读取
15. }
16. **private** **void** flushBuffer() **throws** IOException {
17. **if** (count > 0) {
18. out.write(buf, 0, count);
19. count = 0;
20. }
21. }
22. }

可见，Buffered I/O putStream每读/写一个字节，若要操作的数据在BUF中，就直接对内存的buf[]进行读/写操作；否则从磁盘相应位置填充buf[]，再直接对内存的buf[]进行读/写操作，绝大部分的读/写操作是对内存buf[]的操作。

**1．3．小结**

内存存取时间单位是纳秒级（10E-9），磁盘存取时间单位是毫秒级（10E-3），同样操作一次的开销，内存比磁盘快了百万倍。理论上可以预见，即使对内存操作上万次，花费的时间也远少对于磁盘一次I/O的开销。显然后者是通过增加位于内存的BUF存取，减少磁盘I/O的开销，提高存取效率的，当然这样也增加了BUF控制部分的开销。从实际应用来看，存取效率提高了32倍。

根据1.3得出的结论，现试着对RandomAccessFile类也加上缓冲读写机制。

随机访问类与顺序类不同，前者是通过实现DataInput/DataOutput接口创建的，而后者是扩展FilterInputStream/FilterOutputStream创建的，不能直接照搬。

**2．1．开辟缓冲区BUF[默认：1024字节]，用作读/写的共用缓冲区。**

**2．2．先实现读缓冲。**

读缓冲逻辑的基本原理：

* A 欲读文件POS位置的一个字节。
* B 查BUF中是否存在？若有，直接从BUF中读取，并返回该字符BYTE。
* C 若没有，则BUF重新定位到该POS所在的位置并把该位置附近的BUFSIZE的字节的文件内容填充BUFFER，返回B。

以下给出关键部分代码及其说明：

Java代码  [IMG_261](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **class** BufferedRandomAccessFile **extends** RandomAccessFile {
2. //  byte read(long pos)：读取当前文件POS位置所在的字节
3. //  bufstartpos、bufendpos代表BUF映射在当前文件的首/尾偏移地址。
4. //  curpos指当前类文件指针的偏移地址。
5. **public** **byte** read(**long** pos) **throws** IOException {
6. **if** (pos < **this**.bufstartpos || pos > **this**.bufendpos ) {
7. **this**.flushbuf();
8. **this**.seek(pos);
9. **if** ((pos < **this**.bufstartpos) || (pos > **this**.bufendpos))
10. **throw** **new** IOException();
11. }
12. **this**.curpos = pos;
13. **return** **this**.buf[(**int**)(pos - **this**.bufstartpos)];
14. }
15. // void flushbuf()：bufdirty为真，把buf[]中尚未写入磁盘的数据，写入磁盘。
16. **private** **void** flushbuf() **throws** IOException {
17. **if** (**this**.bufdirty == **true**) {
18. **if** (**super**.getFilePointer() != **this**.bufstartpos) {
19. **super**.seek(**this**.bufstartpos);
20. }
21. **super**.write(**this**.buf, 0, **this**.bufusedsize);
22. **this**.bufdirty = **false**;
23. }
24. }
25. // void seek(long pos)：移动文件指针到pos位置，并把buf[]映射填充至POS所在的文件块。
26. **public** **void** seek(**long** pos) **throws** IOException {
27. **if** ((pos < **this**.bufstartpos) || (pos > **this**.bufendpos)) { // seek pos not in buf
28. **this**.flushbuf();
29. **if** ((pos >= 0) && (pos <= **this**.fileendpos) && (**this**.fileendpos != 0)) {   // seek pos in file (file length > 0)
30. **this**.bufstartpos =  pos \* bufbitlen / bufbitlen;
31. **this**.bufusedsize = **this**.fillbuf();
32. } **else** **if** (((pos == 0) && (**this**.fileendpos == 0)) || (pos == **this**.fileendpos + 1)) {   // seek pos is append pos
33. **this**.bufstartpos = pos;
34. **this**.bufusedsize = 0;
35. }
36. **this**.bufendpos = **this**.bufstartpos + **this**.bufsize - 1;
37. }
38. **this**.curpos = pos;
39. }
40. // int fillbuf()：根据bufstartpos，填充buf[]。
41. **private** **int** fillbuf() **throws** IOException {
42. **super**.seek(**this**.bufstartpos);
43. **this**.bufdirty = **false**;
44. **return** **super**.read(**this**.buf);
45. }
46. }

至此缓冲读基本实现，逐字节COPY一个12兆的文件（这里牵涉到读和写，用BufferedRandomAccessFile试一下读的速度）：

|  |  |  |
| --- | --- | --- |
| 读 | 写 | 耗用时间（秒） |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |

可见速度显著提高，与BufferedInputStream+DataInputStream不相上下。

**2．3．实现写缓冲。**

写缓冲逻辑的基本原理：

* A欲写文件POS位置的一个字节。
* B 查BUF中是否有该映射？若有，直接向BUF中写入，并返回true。
* C若没有，则BUF重新定位到该POS所在的位置，并把该位置附近的 BUFSIZE字节的文件内容填充BUFFER，返回B。

下面给出关键部分代码及其说明：

Java代码  [IMG_262](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. // boolean write(byte bw, long pos)：向当前文件POS位置写入字节BW。
2. // 根据POS的不同及BUF的位置：存在修改、追加、BUF中、BUF外等情况。在逻辑判断时，把最可能出现的情况，最先判断，这样可提高速度。
3. // fileendpos：指示当前文件的尾偏移地址，主要考虑到追加因素
4. **public** **boolean** write(**byte** bw, **long** pos) **throws** IOException {
5. **if** ((pos >= **this**.bufstartpos) && (pos <= **this**.bufendpos)) { // write pos in buf
6. **this**.buf[(**int**)(pos - **this**.bufstartpos)] = bw;
7. **this**.bufdirty = **true**;
8. **if** (pos == **this**.fileendpos + 1) { // write pos is append pos
9. **this**.fileendpos++;
10. **this**.bufusedsize++;
11. }
12. } **else** { // write pos not in buf
13. **this**.seek(pos);
14. **if** ((pos >= 0) && (pos <= **this**.fileendpos) && (**this**.fileendpos != 0)) { // write pos is modify file
15. **this**.buf[(**int**)(pos - **this**.bufstartpos)] = bw;
16. } **else** **if** (((pos == 0) && (**this**.fileendpos == 0)) || (pos == **this**.fileendpos + 1)) { // write pos is append pos
17. **this**.buf[0] = bw;
18. **this**.fileendpos++;
19. **this**.bufusedsize = 1;
20. } **else** {
21. **throw** **new** IndexOutOfBoundsException();
22. }
23. **this**.bufdirty = **true**;
24. }
25. **this**.curpos = pos;
26. **return** **true**;
27. }

至此缓冲写基本实现，逐字节COPY一个12兆的文件，（这里牵涉到读和写，结合缓冲读，用BufferedRandomAccessFile试一下读/写的速度）：

|  |  |  |
| --- | --- | --- |
| 读 | 写 | 耗用时间（秒） |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |

可见综合读/写速度已超越BufferedInput/OutputStream+DataInput/OutputStream。

### [高效的RandomAccessFile【续】](http://zhang-xiujiao.iteye.com/blog/1150762" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

[http://zhang-xiujiao.iteye.com/blog/1150762](http://zhang-xiujiao.iteye.com/blog/1150762" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

**优化BufferedRandomAccessFile。**

优化原则：

* 调用频繁的语句最需要优化，且优化的效果最明显。
* 多重嵌套逻辑判断时，最可能出现的判断，应放在最外层。
* 减少不必要的NEW。

这里举一典型的例子：

Java代码  [IMG_263](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **void** seek(**long** pos) **throws** IOException {
2. ...
3. **this**.bufstartpos =  pos \* bufbitlen / bufbitlen; // bufbitlen指buf[]的位长，例：若bufsize=1024，则bufbitlen=10。
4. ...
5. }

seek函数使用在各函数中，调用非常频繁，上面加重的这行语句根据pos和bufsize确定buf[]对应当前文件的映射位置，用"\*"、"/"确定，显然不是一个好方法。

* 优化一：this.bufstartpos = (pos << bufbitlen) >> bufbitlen;
* 优化二：this.bufstartpos = pos & bufmask; // this.bufmask = ~((long)this.bufsize - 1);

两者效率都比原来好，但后者显然更好，因为前者需要两次移位运算、后者只需一次逻辑与运算（bufmask可以预先得出）。

至此优化基本实现，逐字节COPY一个12兆的文件，（这里牵涉到读和写，结合缓冲读，用优化后BufferedRandomAccessFile试一下读/写的速度）：

|  |  |  |
| --- | --- | --- |
| 读 | 写 | 耗用时间（秒） |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |
| BufferedRandomAccessFile优 | BufferedRandomAccessFile优 | 2.197 |

可见优化尽管不明显，还是比未优化前快了一些，也许这种效果在老式机上会更明显。

以上比较的是顺序存取，即使是随机存取，在绝大多数情况下也不止一个BYTE，所以缓冲机制依然有效。而一般的顺序存取类要实现随机存取就不怎么容易了。

**需要完善的地方**

提供文件追加功能：

Java代码  [IMG_264](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **boolean** append(**byte** bw) **throws** IOException {
2. **return** **this**.write(bw, **this**.fileendpos + 1);
3. }

提供文件当前位置修改功能：

Java代码  [IMG_265](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **boolean** write(**byte** bw) **throws** IOException {
2. **return** **this**.write(bw, **this**.curpos);
3. }

返回文件长度（由于BUF读写的原因，与原来的RandomAccessFile类有所不同）：

Java代码  [IMG_266](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **long** length() **throws** IOException {
2. **return** **this**.max(**this**.fileendpos + 1, **this**.initfilelen);
3. }

返回文件当前指针（由于是通过BUF读写的原因，与原来的RandomAccessFile类有所不同）：

Java代码  [IMG_267](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **long** getFilePointer() **throws** IOException {
2. **return** **this**.curpos;
3. }

提供对当前位置的多个字节的缓冲写功能：

Java代码  [IMG_268](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **void** write(**byte** b[], **int** off, **int** len) **throws** IOException {
2. **long** writeendpos = **this**.curpos + len - 1;
3. **if** (writeendpos <= **this**.bufendpos) { // b[] in cur buf
4. System.arraycopy(b, off, **this**.buf, (**int**)(**this**.curpos - **this**.bufstartpos), len);
5. **this**.bufdirty = **true**;
6. **this**.bufusedsize = (**int**)(writeendpos - **this**.bufstartpos + 1);
7. } **else** { // b[] not in cur buf
8. **super**.seek(**this**.curpos);
9. **super**.write(b, off, len);
10. }
11. **if** (writeendpos > **this**.fileendpos)
12. **this**.fileendpos = writeendpos;
13. **this**.seek(writeendpos+1);
14. }
15. **public** **void** write(**byte** b[]) **throws** IOException {
16. **this**.write(b, 0, b.length);
17. }

提供对当前位置的多个字节的缓冲读功能：

Java代码  [IMG_269](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. **public** **int** read(**byte** b[], **int** off, **int** len) **throws** IOException {
2. **long** readendpos = **this**.curpos + len - 1;
3. **if** (readendpos <= **this**.bufendpos && readendpos <= **this**.fileendpos ) { // read in buf
4. System.arraycopy(**this**.buf, (**int**)(**this**.curpos - **this**.bufstartpos), b, off, len);
5. } **else** { // read b[] size > buf[]
6. **if** (readendpos > **this**.fileendpos) { // read b[] part in file
7. len = (**int**)(**this**.length() - **this**.curpos + 1);
8. }
9. **super**.seek(**this**.curpos);
10. len = **super**.read(b, off, len);
11. readendpos = **this**.curpos + len - 1;
12. }
13. **this**.seek(readendpos + 1);
14. **return** len;
15. }
16. **public** **int** read(**byte** b[]) **throws** IOException {
17. **return** **this**.read(b, 0, b.length);
18. }
19. **public** **void** setLength(**long** newLength) **throws** IOException {
20. **if** (newLength > 0) {
21. **this**.fileendpos = newLength - 1;
22. } **else** {
23. **this**.fileendpos = 0;
24. }
25. **super**.setLength(newLength);
26. }
28. **public** **void** close() **throws** IOException {
29. **this**.flushbuf();
30. **super**.close();
31. }

至此完善工作基本完成，试一下新增的多字节读/写功能，通过同时读/写1024个字节，来COPY一个12兆的文件，（这里牵涉到读和写，用完善后BufferedRandomAccessFile试一下读/写的速度）：

|  |  |  |
| --- | --- | --- |
| 读 | 写 | 耗用时间（秒） |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |
| BufferedRandomAccessFile优 | BufferedRandomAccessFile优 | 2.197 |
| BufferedRandomAccessFile完 | BufferedRandomAccessFile完 | 0.401 |

**与MappedByteBuffer+RandomAccessFile的对比？**

JDK1.4+提供了NIO类 ，其中MappedByteBuffer类用于映射缓冲，也可以映射随机文件访问，可见JAVA设计者也看到了RandomAccessFile的问题，并加以改进。怎么通过MappedByteBuffer+RandomAccessFile拷贝文件呢？下面就是测试程序的主要部分：

Java代码  [IMG_270](http://blog.csdn.net/jiangxinyu/article/details/7885518/" \o "收藏这段代码" \t "http://blog.csdn.net/jiangxinyu/article/details/7885518/_blank)

1. RandomAccessFile rafi = **new** RandomAccessFile(SrcFile, "r");
2. RandomAccessFile rafo = **new** RandomAccessFile(DesFile, "rw");
3. FileChannel fci = rafi.getChannel();
4. FileChannel fco = rafo.getChannel();
5. **long** size = fci.size();
6. MappedByteBuffer mbbi = fci.map(FileChannel.MapMode.READ\_ONLY, 0, size);
7. MappedByteBuffer mbbo = fco.map(FileChannel.MapMode.READ\_WRITE, 0, size);
8. **long** start = System.currentTimeMillis();
9. **for** (**int** i = 0; i < size; i++) {
10. **byte** b = mbbi.get(i);
11. mbbo.put(i, b);
12. }
13. fcin.close();
14. fcout.close();
15. rafi.close();
16. rafo.close();
17. System.out.println("Spend: "+(**double**)(System.currentTimeMillis()-start) / 1000 + "s");

试一下JDK1.4的映射缓冲读/写功能，逐字节COPY一个12兆的文件，（这里牵涉到读和写）：

|  |  |  |
| --- | --- | --- |
| 读 | 写 | 耗用时间（秒） |
| RandomAccessFile | RandomAccessFile | 95.848 |
| BufferedInputStream + DataInputStream | BufferedOutputStream + DataOutputStream | 2.935 |
| BufferedRandomAccessFile | BufferedOutputStream + DataOutputStream | 2.813 |
| BufferedRandomAccessFile | BufferedRandomAccessFile | 2.453 |
| BufferedRandomAccessFile优 | BufferedRandomAccessFile优 | 2.197 |
| BufferedRandomAccessFile完 | BufferedRandomAccessFile完 | 0.401 |
| MappedByteBuffer+ RandomAccessFile | MappedByteBuffer+ RandomAccessFile | 1.209 |

确实不错，看来NIO有了极大的进步。建议采用 MappedByteBuffer+RandomAccessFile的方式。

## 二 String 类

1）学习string

### 概念构造函数

1. 构造函数  
        String(**byte[ ]** **bytes**)：**通过byte数组构造字符串对象**。  
        String(**char[ ]** **value**)：**通过char数组构造字符串对象**。  
        String(**Sting** **original**)：构造一个**original**的**副本**。即：**拷贝一个original**。  
        String(**StringBuffer** **buffer**)：通过StringBuffer数组构造字符串对象。    
   例如：  
         byte[] b = {'a','b','c','d','e','f','g','h','i','j'};  
         char[] c = {'0','1','2','3','4','5','6','7','8','9'};  
         String sb = new String(b);                 //**abcdefghij**  
         String sb\_sub = new String(b,3,2);     //**de**  
         String sc = new String(c);                  //**0123456789**  
         String sc\_sub = new String(c,3,2);    //**34**  
         String sb\_copy = new String(sb);       //**abcdefghij**    
         System.out.println("sb:"+sb);  
         System.out.println("sb\_sub:"+sb\_sub);  
         System.out.println("sc:"+sc);  
         System.out.println("sc\_sub:"+sc\_sub);  
         System.out.println("sb\_copy:"+sb\_copy);

用法

### Api

1. **char** **charAt(int index)** ：**取字符串中的某一个字符**，其中的参数index指的是字符串中序数。字符串的序数从0开始到length()-1 。  
       例如：String s = new String("abcdefghijklmnopqrstuvwxyz");  
             System.out.println("s.charAt(5): " + s.charAt(5) );  
             结果为： s.charAt(5): f  
   2. **int compareTo(String anotherString)** ：**当前String对象与anotherString比较**。**相等**关系**返回０**；**不相等**时，从两个字符串第0个字符开始比较，**返回第一个不相等的字符差**，另一种情况，**较长字符串的前面部分恰巧是较短的字符串，返回它们的长度差。**

**int compareTo(Object o)** ：如果o是String对象，和2的功能一样；否则抛出**ClassCastException**异常。  
    例如:String s1 = new String("abcdefghijklmn");  
            String s2 = new String("abcdefghij");  
           String s3 = new String("abcdefghijalmn");  
           System.out.println("s1.compareTo(s2): " + s1.compareTo(s2) ); //返回长度差  
           System.out.println("s1.compareTo(s3): " + s1.compareTo(s3) ); //返回'k'-'a'的差  
           结果为：s1.compareTo(s2): 4  
                       s1.compareTo(s3): 10

1. **String concat(String str)** ：**将该String对象与str连接在一起。**  
   5. **boolean contentEquals(StringBuffer sb)** **：将该String对象与StringBuffer对象sb进行比较。**  
   6. **static String copyValueOf(char[] data)** ：  
   7. **static String copyValueOf(char[] data, int offset, int count)** ：这两个方法**将char数组转换成String**，与其中一个构造函数类似。  
   8. **boolean endsWith(String suffix)** ：**该String对象是否以suffix结尾**。  
       例如：String s1 = new String("abcdefghij");  
              String s2 = new String("ghij");  
              System.out.println("s1.endsWith(s2): " + s1.endsWith(s2) );  
              结果为：s1.endsWith(s2): true

**int hashCode()** ：**返回当前字符的哈希表码**。

**int indexOf(String str)** ：**只找第一个匹配字符串位置**。  
16. **int indexOf(String str, int fromIndex)** ：**从fromIndex开始找第一个匹配字符串位置**。  
      例如：String s = new String("write once, run anywhere!");  
              String ss = new String("run");  
              System.out.println("s.indexOf('r'): " + s.indexOf('r') );  
              System.out.println("s.indexOf('r',2): " + s.indexOf('r',2) );  
              System.out.println("s.indexOf(ss): " + s.indexOf(ss) );  
              结果为：s.indexOf('r'): 1  
                      s.indexOf('r',2): 12  
                      s.indexOf(ss): 12

19. **int lastIndexOf(String str)**20. **int lastIndexOf(String str, int fromIndex)** 以上四个方法与13、14、15、16类似，不同的是：找**最后一个匹配的内容**。  
public class CompareToDemo {  
      public static void main (String[] args) {  
           String s1 = new String("ac**b**de**b**fg");  
        
           System.out.println(s1.lastIndexOf((int)'**b**',**7**));  
     }  
}

|  |
| --- |
| String : 字符串类型  一、**构造函数**      String(**byte[ ]** **bytes**)：**通过byte数组构造字符串对象**。      String(**char[ ]** **value**)：**通过char数组构造字符串对象**。      String(**Sting** **original**)：构造一个**original**的**副本**。即：**拷贝一个original**。      String(**StringBuffer** **buffer**)：通过StringBuffer数组构造字符串对象。   例如：       byte[] b = {'a','b','c','d','e','f','g','h','i','j'};       char[] c = {'0','1','2','3','4','5','6','7','8','9'};       String sb = new String(b);                 //**abcdefghij**       String sb\_sub = new String(b,3,2);     //**de**       String sc = new String(c);                  //**0123456789**       String sc\_sub = new String(c,3,2);    //**34**       String sb\_copy = new String(sb);       //**abcdefghij**         System.out.println("sb:"+sb);       System.out.println("sb\_sub:"+sb\_sub);       System.out.println("sc:"+sc);       System.out.println("sc\_sub:"+sc\_sub);       System.out.println("sb\_copy:"+sb\_copy);       **输出结果**：**sb:abcdefghij                       sb\_sub:de                        sc:0123456789                         sc\_sub:34                         sb\_copy:abcdefghij**  二、方法：       说明：①、所有方法均为public。            ②、书写格式： [修饰符] <返回类型><方法名([参数列表])>        例如：static int parseInt(String s)       表示此方法(parseInt)为类方法(static),返回类型为(int),方法所需要为String类型。  1. **char** **charAt(int index)** ：**取字符串中的某一个字符**，其中的参数index指的是字符串中序数。字符串的序数从0开始到length()-1 。     例如：String s = new String("abcdefghijklmnopqrstuvwxyz");           System.out.println("s.charAt(5): " + s.charAt(5) );           结果为： s.charAt(5): f 2. **int compareTo(String anotherString)** ：**当前String对象与anotherString比较**。**相等**关系**返回０**；**不相等**时，从两个字符串第0个字符开始比较，**返回第一个不相等的字符差**，另一种情况，**较长字符串的前面部分恰巧是较短的字符串，返回它们的长度差。** 3. **int compareTo(Object o)** ：如果o是String对象，和2的功能一样；否则抛出**ClassCastException**异常。     例如:String s1 = new String("abcdefghijklmn");             String s2 = new String("abcdefghij");            String s3 = new String("abcdefghijalmn");            System.out.println("s1.compareTo(s2): " + s1.compareTo(s2) ); //返回长度差            System.out.println("s1.compareTo(s3): " + s1.compareTo(s3) ); //返回'k'-'a'的差            结果为：s1.compareTo(s2): 4                        s1.compareTo(s3): 10 4. **String concat(String str)** ：**将该String对象与str连接在一起。** 5. **boolean contentEquals(StringBuffer sb)** **：将该String对象与StringBuffer对象sb进行比较。** 6. **static String copyValueOf(char[] data)** ： 7. **static String copyValueOf(char[] data, int offset, int count)** ：这两个方法**将char数组转换成String**，与其中一个构造函数类似。 8. **boolean endsWith(String suffix)** ：**该String对象是否以suffix结尾**。     例如：String s1 = new String("abcdefghij");            String s2 = new String("ghij");            System.out.println("s1.endsWith(s2): " + s1.endsWith(s2) );            结果为：s1.endsWith(s2): true 9. **boolean equals(Object anObject)** ：**当anObject不为空并且与当前String对象一样，返回true；否则，返回false**。 10. **byte[] getBytes()** ：**将该String对象转换成byte数组**。 11. **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)** ：**该方法将字符串拷贝到字符数组中**。其中，srcBegin为拷贝的起始位置、srcEnd为拷贝的结束位置、字符串数值dst为目标字符数组、dstBegin为目标字符数组的拷贝起始位置。      例如：char[] s1 = {'I',' ','l','o','v','e',' ','h','e','r','!'};//s1=I love her!            String s2 = new String("you!"); s2.getChars(0,3,s1,7); //s1=I love you!            System.out.println( s1 );            结果为：I love you! 12. **int hashCode()** ：**返回当前字符的哈希表码**。 13. **int indexOf(int ch)** ：**只找第一个匹配字符位置**。 14. **int indexOf(int ch, int** **fromIndex)** ：**从fromIndex开始找第一个匹配字符位置**。 15. **int indexOf(String str)** ：**只找第一个匹配字符串位置**。 16. **int indexOf(String str, int fromIndex)** ：**从fromIndex开始找第一个匹配字符串位置**。       例如：String s = new String("write once, run anywhere!");               String ss = new String("run");               System.out.println("s.indexOf('r'): " + s.indexOf('r') );               System.out.println("s.indexOf('r',2): " + s.indexOf('r',2) );               System.out.println("s.indexOf(ss): " + s.indexOf(ss) );               结果为：s.indexOf('r'): 1                       s.indexOf('r',2): 12                       s.indexOf(ss): 12 17. **int lastIndexOf(int ch)** 18. **int lastIndexOf(int ch, int fromIndex)** 19. **int lastIndexOf(String str)** 20. **int lastIndexOf(String str, int fromIndex)** 以上四个方法与13、14、15、16类似，不同的是：找**最后一个匹配的内容**。 public class CompareToDemo {       public static void main (String[] args) {            String s1 = new String("ac**b**de**b**fg");                   System.out.println(s1.lastIndexOf((int)'**b**',**7**));      } } **运行结果**：**5**        （其中**fromIndex**的参数为 **7**，是从字符串**acbdebfg**的最后一个字符**g**开始**往前数**的位数。既是从字符**c**开始匹配，寻找最后一个匹配**b**的位置。所以结果为 **5**）   21. **int length()** ：**返回当前字符串长度**。 22. **String replace(char** **oldChar, char** **newChar)** ：**将字符号串中第一个oldChar替换成newChar**。 23. **boolean startsWith(String prefix)** ：**该String对象是否以prefix开始**。 24. **boolean startsWith(String** **prefix, int** **toffset)** ：**该String对象从toffset位置算起，是否以prefix开始**。      例如：String s = new String("write once, run anywhere!");              String ss = new String("write");              String sss = new String("once");              System.out.println("s.startsWith(ss): " + s.startsWith(ss) );              System.out.println("s.startsWith(sss,6): " + s.startsWith(sss,6) );              结果为：s.startsWith(ss): true                      s.startsWith(sss,6): true |

1. **String substring(int beginIndex)** ：**取从beginIndex位置开始到结束的子字符串**。  
   26.**String substring(int beginIndex, int endIndex)** ：**取从beginIndex位置开始到endIndex位置的子字符串**。  
   27. **char[ ] toCharArray()** ：**将该String对象转换成char数组**。  
   28. **String toLowerCase()** ：**将字符串转换成小写**。  
   29. **String toUpperCase()** ：将字符串转换成大写。  
        例如：String s = new String("java.lang.Class String");  
                System.out.println("s.toUpperCase(): " + s.toUpperCase() );  
                System.out.println("s.toLowerCase(): " + s.toLowerCase() );  
                结果为：s.toUpperCase(): JAVA.LANG.CLASS STRING  
                     s.toLowerCase(): java.lang.class string

30. **static String valueOf(boolean b)**31. **static String valueOf(char c)**32. **static String valueOf(char[] data)**33. **static String valueOf(char[] data, int offset, int count)**34. **static String valueOf(double d)**35. **static String valueOf(float f)**36. **static String valueOf(int i)**37. **static String valueOf(long l)**38. **static String valueOf(Object obj)**     以上方法用于将各种不同类型转换成Java字符型。这些都是类方法。

注意点

### Java中String类的常用方法: ****public char charAt(int index)**** 返回字符串中第index个字符； ****public int length()**** 返回字符串的长度； ****public int indexOf(String str)**** 返回字符串中第一次出现str的位置； ****public int indexOf(String str,int fromIndex)**** 返回字符串从fromIndex开始第一次出现str的位置； ****public boolean equalsIgnoreCase(String another)**** 比较字符串与another是否一样（忽略大小写）； ****public String replace(char oldchar,char newChar)**** 在字符串中用newChar字符替换oldChar字符 ****public boolean startsWith(String prefix)**** 判断字符串是否以prefix字符串开头； ****public boolean endsWith(String suffix)**** 判断一个字符串是否以suffix字符串结尾； ****public String toUpperCase()**** 返回一个字符串为该字符串的大写形式； ****public String toLowerCase()**** 返回一个字符串为该字符串的小写形式 ****public String substring(int beginIndex)**** 返回该字符串从beginIndex开始到结尾的子字符串； ****public String substring(int beginIndex,int endIndex)**** 返回该字符串从beginIndex开始到endsIndex结尾的子字符串 ****public String trim()**** 返回该字符串去掉开头和结尾空格后的字符串 ****public String[] split(String regex)**** 将一个字符串按照指定的分隔符分隔，返回分隔后的字符串数组 实例：   public class ****SplitDemo****{      public static void main (String[] args) {              String ****date**** = "2008/09/10";             ****String[ ]**** ****dateAfterSplit****= new String[3];             ****dateAfterSplit****=****date.split("/")****;         ****//以“/”作为分隔符来分割date字符串，并把结果放入3个字符串中。****             for(int i=0;i<****dateAfterSplit****.length;i++)                        System.out.print(****dateAfterSplit****[i]+" ");       } } ****运行结果****：****2008 09 10          //结果为分割后的3个字符串****

实例：  
TestString1.java:  
程序代码  
public class TestString1  
{  
    public static void main(String args[]) {  
        String s1 = "Hello World" ;  
        String s2 = "hello world" ;  
        System.out.println(s1.charAt(1)) ;  
        System.out.println(s2.length()) ;  
        System.out.println(s1.indexOf("World")) ;  
        System.out.println(s2.indexOf("World")) ;  
        System.out.println(s1.**equals**(s2)) ;  
        System.out.println(s1**.equalsIgnoreCase**(s2)) ;

        String s = "我是J2EE程序员" ;  
        String sr = s.replace('我','你') ;  
        System.out.println(sr) ;  
    }  
}

TestString2.java:  
程序代码

public class TestString2  
{  
    public static void main(String args[]) {  
        String s = "Welcome to Java World!" ;  
        String s2 = "   magci   " ;  
        System.out.println(s.startsWith("Welcome")) ;  
        System.out.println(s.endsWith("World")) ;  
        String sL = s.toLowerCase() ;  
        String sU = s.toUpperCase() ;  
        System.out.println(sL) ;  
        System.out.println(sU) ;  
        String subS = s.substring(11) ;  
        System.out.println(subS) ;  
        String s1NoSp = s2.trim() ;  
        System.out.println(s1NoSp) ;  
    }

## 三 正则表达式

### 常用正则表达式

|  |  |
| --- | --- |
| 规则 | 正则表达式语法 |
| 一个或多个汉字 | ^[\u0391-\uFFE5]+$ |
| 邮政编码 | ^[1-9]\d{5}$ |
| QQ号码 | ^[1-9]\d{4,10}$ |
| 邮箱 | ^[a-zA-Z\_]{1,}[0-9]{0,}@(([a-zA-z0-9]-\*){1,}\.){1,3}[a-zA-z\-]{1,}$ |
| 用户名（字母开头 + 数字/字母/下划线） | ^[A-Za-z][A-Za-z1-9\_-]+$ |
| 手机号码 | ^1[3|4|5|8][0-9]\d{8}$ |
| URL | ^((http|https)://)?([\w-]+\.)+[\w-]+(/[\w-./?%&=]\*)?$ |
| 18位身份证号 | ^(\d{6})(18|19|20)?(\d{2})([01]\d)([0123]\d)(\d{3})(\d|X|x)?$ |

### 正则表达式语法

|  |  |
| --- | --- |
| 元字符 | 描述 |
| \ | 将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，“\\n”匹配\n。“\n”匹配换行符。序列“\\”匹配“\”而“\(”则匹配“(”。即相当于多种编程语言中都有的“转义字符”的概念。 |
| ^ | 匹配输入字符串的开始位置。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。 |
| $ | 匹配输入字符串的结束位置。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。 |
| \* | 匹配前面的子表达式任意次。例如，zo\*能匹配“z”，“zo”以及“zoo”。\*等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次(大于等于1次）。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“do”或“does”中的“do”。?等价于{0,1}。 |
| {n} | n是一个非负整数。匹配确定的n次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。 |
| {n,} | n是一个非负整数。至少匹配n次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o\*”。 |
| {n,m} | m和n均为非负整数，其中n<=m。最少匹配n次且最多匹配m次。例如，“o{1,3}”将匹配“fooooood”中的前三个o。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。 |
| ? | 当该字符紧跟在任何一个其他限制符（\*,+,?，{n}，{n,}，{n,m}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串“oooo”，“o+?”将匹配单个“o”，而“o+”将匹配所有“o”。 |
| .点 | 匹配除“\r\n”之外的任何单个字符。要匹配包括“\r\n”在内的任何字符，请使用像“[\s\S]”的模式。 |
| (pattern) | 匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JScript中则使用$0…$9属性。要匹配圆括号字符，请使用“\(”或“\)”。 |
| (?:pattern) | 匹配pattern但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分是很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。 |
| (?=pattern) | 正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。 |
| (?!pattern) | 正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。 |
| (?<=pattern) | 反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。 |
| (?<!pattern) | 反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。 |
| x|y | 匹配x或y。例如，“z|food”能匹配“z”或“food”或"zood"(此处请谨慎)。“(z|f)ood”则匹配“zood”或“food”。 |
| [xyz] | 字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。 |
| [^xyz] | 负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。  注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身. |
| [^a-z] | 负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置（即正则表达式的“匹配”有两种概念，一种是匹配字符，一种是匹配位置，这里的\b就是匹配位置的）。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”。 |
| \B | 匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。 |
| \cx | 匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的“c”字符。 |
| \d | 匹配一个数字字符。等价于[0-9]。 |
| \D | 匹配一个非数字字符。等价于[^0-9]。 |
| \f | 匹配一个换页符。等价于\x0c和\cL。 |
| \n | 匹配一个换行符。等价于\x0a和\cJ。 |
| \r | 匹配一个回车符。等价于\x0d和\cM。 |
| \s | 匹配任何不可见字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。 |
| \S | 匹配任何可见字符。等价于[^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于\x09和\cI。 |
| \v | 匹配一个垂直制表符。等价于\x0b和\cK。 |
| \w | 匹配包括下划线的任何单词字符。类似但不等价于“[A-Za-z0-9\_]”，这里的"单词"字符使用Unicode字符集。 |
| \W | 匹配任何非单词字符。等价于“[^A-Za-z0-9\_]”。 |
| \xn | 匹配n，其中n为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“\x41”匹配“A”。“\x041”则等价于“\x04&1”。正则表达式中可以使用ASCII编码。 |
| \num | 匹配num，其中num是一个正整数。对所获取的匹配的引用。例如，“(.)\1”匹配两个连续的相同字符。 |
| \n | 标识一个八进制转义值或一个向后引用。如果\n之前至少n个获取的子表达式，则n为向后引用。否则，如果n为八进制数字（0-7），则n为一个八进制转义值。 |
| \nm | 标识一个八进制转义值或一个向后引用。如果\nm之前至少有nm个获得子表达式，则nm为向后引用。如果\nm之前至少有n个获取，则n为一个后跟文字m的向后引用。如果前面的条件都不满足，若n和m均为八进制数字（0-7），则\nm将匹配八进制转义值nm。 |
| \nml | 如果n为八进制数字（0-7），且m和l均为八进制数字（0-7），则匹配八进制转义值nml。 |
| \un | 匹配n，其中n是一个用四个十六进制数字表示的Unicode字符。例如，\u00A9匹配版权符号（&copy;）。 |
| \< \> | 匹配词（word）的开始（\<）和结束（\>）。例如正则表达式\<the\>能够匹配字符串"for the wise"中的"the"，但是不能匹配字符串"otherwise"中的"the"。注意：这个元字符不是所有的软件都支持的。 |
| \( \) | 将 \( 和 \) 之间的表达式定义为“组”（group），并且将匹配这个表达式的字符保存到一个临时区域（一个正则表达式中最多可以保存9个），它们可以用 \1 到\9 的符号来引用。 |
| | | 将两个匹配条件进行逻辑“或”（Or）运算。例如正则表达式(him|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意：这个元字符不是所有的软件都支持的。 |
| + | 匹配1或多个正好在它之前的那个字符。例如正则表达式9+匹配9、99、999等。注意：这个元字符不是所有的软件都支持的。 |
| ? | 匹配0或1个正好在它之前的那个字符。注意：这个元字符不是所有的软件都支持的。 |
| {i} {i,j} | 匹配指定数目的字符，这些字符是在它之前的表达式定义的。例如正则表达式A[0-9]{3} 能够匹配字符"A"后面跟着正好3个数字字符的串，例如A123、A348等，但是不匹配A1234。而正则表达式[0-9]{4,6} 匹配连续的任意4个、5个或者6个数字 |

### 用法

ava | 复制

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void main(String[] args) {      // 要验证的字符串      String str = "service@xsoftlab.net";      // 邮箱验证规则      String regEx = "[a-zA-Z\_]{1,}[0-9]{0,}@(([a-zA-z0-9]-\*){1,}\\.){1,3}[a-zA-z\\-]{1,}";      // 编译正则表达式      Pattern pattern = Pattern.compile(regEx);      // 忽略大小写的写法      // Pattern pat = Pattern.compile(regEx, Pattern.CASE\_INSENSITIVE);      Matcher matcher = pattern.matcher(str);      // 字符串是否与正则表达式相匹配      boolean rs = matcher.matches();      System.out.println(rs);  } |

### 在字符串中查询字符或者字符串

Java | 复制

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public static void main(String[] args) {      // 要验证的字符串      String str = "baike.xsoftlab.net";      // 正则表达式规则      String regEx = "baike.\*";      // 编译正则表达式      Pattern pattern = Pattern.compile(regEx);      // 忽略大小写的写法      // Pattern pat = Pattern.compile(regEx, Pattern.CASE\_INSENSITIVE);      Matcher matcher = pattern.matcher(str);      // 查找字符串中是否有匹配正则表达式的字符/字符串      boolean rs = matcher.find();      System.out.println(rs);  } |

## 四Java中 map用法

### **概念**

Map 接口提供三种collection 视图，允许以键集、值集或键-值映射关系集的形式查看某个映射的内容。映射顺序 定义为迭代器在映射的 collection 视图上返回其元素的顺序。某些映射实现可明确保证其顺序，如 TreeMap 类；另一些映射实现则不保证顺序，如HashMap 类。

### Api

　　void clear()

　　从此映射中移除所有映射关系（可选操作）。

　　boolean containsKey(Object key)

　　如果此映射包含指定键的映射关系，则返回 true。

　　boolean containsValue(Object value)

　　如果此映射将一个或多个键映射到指定值，则返回 true。

Set<Map.Entry<K,V>> entrySet()

　　返回此映射中包含的映射关系的 Set 视图。

　　boolean equals(Object o)

　　比较指定的对象与此映射是否相等。

　　V get(Object key)

　　返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。

　　int hashCode()

　　返回此映射的哈希码值。

　　boolean isEmpty()

　　如果此映射未包含键-值映射关系，则返回 true。

　　Set<K> keySet()

　　返回此映射中包含的键的 Set 视图。

　　V put(K key, V value)

　　将指定的值与此映射中的指定键关联（可选操作）。

　　void putAll(Map<? extends K,? extends V> m)

　　从指定映射中将所有映射关系复制到此映射中（可选操作）。

　　V remove(Object key)

　　如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。

　　int size()

　　返回此映射中的键-值映射关系数。

　　Collection<V> values()

　　返回此映射中包含的值的 Collection 视图。

### Map的一般用法

1.声明一个Map :

Map map = new HashMap();

2 .向map中放值 ，注意： map是key-value的形式存放的，如：

map.put("sa","dd");

3 .从map中取值 :

String str = map.get("sa").toString,

结果是： str = "dd'

4 .遍历一个map,从中取得key和value :

Map m= new HashMap();

for(Object obj : map.keySet()){

Object value = map.get(obj );

}

## 五list用法示例详解

### API

List 元素 元素可以重复 一般是有序的怎么存的怎么取

ArrayList 底层数据结构使用的是数组结构。线程不同步

linkedList 底层数据结构是 链表

Vector 底层是数据结构数组 线程是同步

boolean add(int index, E element)  
boolean addAll(index,Collection)

 public static void List\_add(){  
     ArrayList a1 = new ArrayList();  
     a1.add("java");  
     a1.add("php");//List集合中的元素可以重复  
    a1.add(".net");  
     System.out.println("原集合："+a1);  
     a1.add(1, "Flash");  
     a1.add(0, "ps");      
     System.out.println(a1);  
  
    ArrayList a2 = new ArrayList();  
    a2.add("javascript");  
    a2.add("3dMax");  
    a2.add("IBM");  
  
    a1.addAll(0, a2);  
    System.out.println(a1);  
}

 boolean remove(int index)

复制代码 代码如下:

 public static void List\_remove(){  
     ArrayList a1 = new ArrayList();  
     a1.add("javascript");  
     a1.add("php");  
     a1.add("flash");  
     System.out.println("原集合："+a1);  
  
     a1.remove(0);  
     System.out.println(a1);  
}

修改指定角标的元素  set(int index, E element)  返回的是修改的那个元素

复制代码 代码如下:

 public static void List\_set() {  
     ArrayList a1 = new ArrayList();  
     a1.add("javascript");  
     a1.add("php");  
     a1.add(".net");  
     System.out.println("原集合："+a1);  
  
     a1.set(1, "falsh");  
     System.out.println(a1);  
}

查

复制代码 代码如下:

get(int index)   返回列表中指定位置的元素  
subList(int fromIndex, int toIndex)    返回列表中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的部分元素。

复制代码 代码如下:

 public static void List\_get() {  
     ArrayList a1 = new ArrayList();  
     a1.add("java");  
     a1.add("php");  
     a1.add("flash");  
  
     System.out.println(a1.get(0));//获取指定角标的元素，有了该方法就可以遍历该集合中的所有元素  
  
     System.out.println(a1.subList(1, 3));//获取集合中某一部分的元素,包含头不包含尾  
}

List集合特有的迭代器:ListIterator(是Iterator的子接口)

注意：  
在迭代时，是不可以通过集合对象的方法操作集合中的元素  
因为会发生ConcurrentModificationException异常（并发异常）  
所以，在迭代器时，只能用迭代器的方法造作元素  
因为Iterator方法是有限的所以只能对元素进行判断，取出，删除的操作  
如果想要其他的操作如添加，修改等，就需要使用其子接口，ListIterator  
该接口只能通过List集合的listIterator方法获取

复制代码 代码如下:

 public class ListIteratorDemo {  
     public static void main(String[] args) {  
         ArrayList a1 = new ArrayList();  
         a1.add("java01");  
         a1.add("java02");  
         a1.add("java03");  
         a1.add("java04");  
  
         System.out.println("原集合是："+a1);  
  
      /\*在迭代过程中准备添加或者删除元素  
      Iterator it = al.iterator();  
      while (it.hasNext()){  
      　　Object obj = it.next();  
  
      　　if (obj.equals("java02"))  
      　　//al.add("java008");//会出现并发异常，因为迭代器正在操作集合，不能再用集合的方法操作集合了  
      　　it.remove();//将java02的引用从集合中删除了  
      　　System.out.println("obj:"+obj);  
        }  
        \*/

    //只有List的listIterator有增，删，改，查这些功能，因为只有List有索引  
       ListIterator li = a1.listIterator();  
          while (li.hasNext()){  
          if(li.next().equals("java02"))  
          //li.add("java009");  
          li.set("java006");  
        }     
    }  
}

### 代码 用法

**[java]** [view plain](http://blog.csdn.net/tianmo2010/article/details/7046602" \o "view plain" \t "http://blog.csdn.net/lskyne/article/details/_blank)[copy](http://blog.csdn.net/tianmo2010/article/details/7046602" \o "copy" \t "http://blog.csdn.net/lskyne/article/details/_blank)

1. **package** code;
2. **import** java.util.ArrayList;
3. **import** java.util.Iterator;
4. **public** **class** SimpleTest {

7. **public** **static** **void** main(String []args){
9. ArrayList list1 = **new** ArrayList();
10. list1.add("one");
11. list1.add("two");
12. list1.add("three");
13. list1.add("four");
14. list1.add("five");
15. list1.add(0,"zero");
16. System.out.println("<--list1中共有>" + list1.size()+ "个元素");
17. System.out.println("<--list1中的内容:" + list1 + "-->");
19. ArrayList list2 = **new** ArrayList();
20. list2.add("Begin");
21. list2.addAll(list1);
22. list2.add("End");
23. System.out.println("<--list2中共有>" + list2.size()+ "个元素");
24. System.out.println("<--list2中的内容:" + list2 + "-->");
26. ArrayList list3 =  **new** ArrayList();
27. list3.removeAll(list1);
28. System.out.println("<--list3中是否存在one: "+ (list3.contains("one")? "是":"否")+ "-->");
30. list3.add(0,"same element");
31. list3.add(1,"same element");
32. System.out.println("<--list3中共有>" + list3.size()+ "个元素");
33. System.out.println("<--list3中的内容:" + list3 + "-->");
34. System.out.println("<--list3中第一次出现same element的索引是" + list3.indexOf("same element") + "-->");
35. System.out.println("<--list3中最后一次出现same element的索引是" + list3.lastIndexOf("same element") + "-->");

38. System.out.println("<--使用Iterator接口访问list3->");
39. Iterator it = list3.iterator();
40. **while**(it.hasNext()){
41. String str = (String)it.next();
42. System.out.println("<--list3中的元素:" + list3 + "-->");
43. }
45. System.out.println("<--将list3中的same element修改为another element-->");
46. list3.set(0,"another element");
47. list3.set(1,"another element");
48. System.out.println("<--将list3转为数组-->");
49. // Object []  array =(Object[]) list3.toArray(new   Object[list3.size()] );
50. Object [] array = list3.toArray();
51. **for**(**int** i = 0; i < array.length ; i ++){
52. String str = (String)array[i];
53. System.out.println("array[" + i + "] = "+ str);
54. }
56. System.out.println("<---清空list3->");
57. list3.clear();
58. System.out.println("<--list3中是否为空: " + (list3.isEmpty()?"是":"否") + "-->");
59. System.out.println("<--list3中共有>" + list3.size()+ "个元素");
61. //System.out.println("hello world!");
62. }
63. }

## 六 数组（Array）：

概念：

相同类型数据的集合。

### 定义数组

**方式1（推荐，更能表明数组类型）**

　　type[] 变量名 = new type[数组中元素的个数];

　　比如：

　　int[] a = new int[10];

　　数组名，也即引用a，指向数组元素的首地址。

**方式2（同C语言）**

　　type变量名[] = new type[数组中元素的个数];

　　如：

　　int a[] = new int[10];

**方式3** **定义时直接初始化**

　　type[] 变量名 = new type[]{逗号分隔的初始化值};

　　其中红色部分可省略，所以又有两种：

　　int[] a = {1,2,3,4};

　　int[] a = new int[]{1,2,3,4};

　　其中int[] a = new int[]{1,2,3,4};的第二个方括号中不能加上数组长度，因为元素个数是由后面花括号的内容决定的。

用法：

### 数组运用基础

**数组长度**

　　Java中的每个数组都有一个名为length的属性，表示数组的长度。

　　length属性是public final int的，即length是只读的。数组长度一旦确定，就不能改变大小。

**equals()**

　　数组内容的比较可以使用equals()方法吗？

　　如下程序：

[IMG_256](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

public class ArrayTest

{

public static void main(String[] args)

{

int[] a = {1, 2, 3};

int[] b = {1, 2, 3};

System.out.println(a.equals(b));

}

}

[IMG_257](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

　　输出结果是false。

　　所以证明**不能直接用equals()方法比较数组内容，因为没有override Object中的实现，所以仍采用其实现，即采用==实现equals()方法，比较是否为同一个对象。**

　　怎么比较呢？一种解决方案是自己写代码，另一种方法是利用**java.util.Arrays**。

　　java.util.Arrays中的方法全是static的。其中包括了equals()方法的各种重载版本。

　　代码如下：

IMG_258

[IMG_259](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

import java.util.Arrays;public class ArrayEqualsTest

{

//Compare the contents of two int arrays

public static boolean isEquals(int[] a, int[] b)

{

if( a == null || b == null )

{

return false;

}

if(a.length != b.length)

{

return false;

}

for(int i = 0; i < a.length; ++i )

{

if(a[i] != b[i])

{

return false;

}

}

return true;

}

public static void main(String[] args)

{

int[] a = {1, 2, 3};

int[] b = {1, 2, 3};

System.out.println(isEquals(a,b));

System.out.println(Arrays.equals(a,b));

}

}

[IMG_260](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

**数组元素不为基本数据类型时**

　　数组元素不为基本原生数据类型时，存放的是引用类型，而不是对象本身。当生成对象之后，引用才指向对象，否则引用为null。

　　如下列程序：

IMG_261

[IMG_262](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

public class ArrayTest2

{

public static void main(String[] args)

{

Person[] p = new Person[3];

//未生成对象时，引用类型均为空

System.out.println(p[0]);

//生成对象之后，引用指向对象

p[0] = new Person(10);

p[1] = new Person(20);

p[2] = new Person(30);

for(int i = 0; i < p.length; i++)

{

System.out.println(p[i].age);

}

}

}class Person

{

int age;

public Person(int age)

{

this.age = age;

}

}

[IMG_263](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

　　输出：

　　null

　　10

　　20

　　30

**也可以在初始化列表里面直接写：**

　　Person[] p = new Person[]{new Person(10), new Person(20), new Person(30)};

### 二维数组

　　二维数组是数组的数组。

**二维数组基础**

　　基本的定义方式有两种形式，如：

　　type[][] i = new type[2][3];（推荐）

　　type i[][] = new type[2][3];

　　如下程序：

[IMG_264](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

public class ArrayTest3

{

public static void main(String[] args)

{

int[][] i = new int[2][3];

System.out.println("Is i an Object? "

+ (i instanceof Object));

System.out.println("Is i[0] an int[]? "

+ (i[0] instanceof int[]));

}

}

[IMG_265](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

　　输出结果是两个true。

**变长的二维数组**

　　二维数组的每个元素都是一个一维数组，这些数组不一定都是等长的。

　　声明二维数组的时候可以只指定第一维大小，空缺出第二维大小，之后再指定不同长度的数组。但是注意，第一维大小不能空缺（不能只指定列数不指定行数）。

　　如下程序：

[IMG_266](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

public class ArrayTest4

{

public static void main(String[] args)

{

//二维变长数组

int[][] a = new int[3][];

a[0] = new int[2];

a[1] = new int[3];

a[2] = new int[1];

//Error: 不能空缺第一维大小

//int[][] b = new int[][3]; }

}

[IMG_267](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

　　二维数组也可以在定义的时候初始化，使用花括号的嵌套完成，这时候不指定两个维数的大小，并且根据初始化值的个数不同，可以生成不同长度的数组元素。

　　如下程序：

[IMG_268](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

public class ArrayTest5

{

public static void main(String[] args)

{

int[][] c = new int[][]{{1, 2, 3},{4},{5, 6, 7, 8}};

for(int i = 0; i < c.length; ++i)

{

for(int j = 0; j < c[i].length; ++j)

{

System.out.print(c[i][j]+" ");

}

System.out.println();

}

}

}

[IMG_269](http://www.cnblogs.com/mengdd/archive/2013/01/04/javascript:void(0);" \o "复制代码)

　　输出：

　　1 2 3

　　4

　　5 6 7 8

分类: [Java](http://www.cnblogs.com/mengdd/category/397114.html" \t "http://www.cnblogs.com/mengdd/archive/2013/01/04/_blank)

注意点：

## 七Java中的[重载与重写的区别](http://www.cnblogs.com/bluestorm/archive/2012/03/01/2376236.html)

### 概念：

    （1） 方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。

重载Overloading是一个类中多态性的一种表现。

    （2） [Java](http://java.chinaitlab.com/" \t "http://www.cnblogs.com/bluestorm/archive/2012/03/01/_blank)的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。

调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法, 这就是多态性。

    （3） 重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

（1） 父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。在[Java](http://java.chinaitlab.com/" \t "http://www.cnblogs.com/bluestorm/archive/2012/03/01/_blank)中，子类可继承父类中的方法，而不需要重新编写相同的方法。

但有时子类并不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。

方法重写又称方法覆盖。

    （2）若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。

如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。

    （3）子类函数的访问修饰权限不能少于父类的；

用法：

注意点

**重写与重载的区别在于：**

重写多态性起作用，对调用被重载过的方法可以大大减少代码的输入量，同一个方法名只要往里面传递不同的参数就可以拥有不同的功能或返回值。

用好重写和重载可以设计一个结构清晰而简洁的类，可以说重写和重载在编写代码过程中的作用非同一般.

**重写方法的规则**：

1、参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。

2、返回的类型必须一直与被重写的方法的返回类型相同，否则不能称其为重写而是重载。

3、访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）

4、重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。例如：

父类的一个方法申明了一个检查异常IOException，在重写这个方法是就不能抛出Exception,只能抛出IOException的子类异常，可以抛出非检查异常。

**而重载的规则：**

1、必须具有不同的参数列表；

2、可以有不责骂的返回类型，只要参数列表不同就可以了；

3、可以有不同的访问修饰符；

4、可以抛出不同的异常；

## 八Java 文件的遍历

**在java中遍历一个文件夹里边的所有文件，可以有两种方式：**

**一、递归遍历，通常也是开发者第一时间能想到的方法，递归遍历的优点是：实现起来相对简单，代码量相对较少，执行效率较高，缺点是：比较吃内存，对硬件要求较高；具体算法如下：**

[IMG_256](http://www.cnblogs.com/fenglie/articles/javascript:void(0);" \o "复制代码)

### 递归遍历

private void getDirectory(File file) {

File flist[] = file.listFiles();

if (flist == null || flist.length == 0) {

return 0;

}

for (File f : flist) {

if (f.isDirectory()) {

//这里将列出所有的文件夹

System.out.println("Dir==>" + f.getAbsolutePath());

getDirectory(f);

} else {

//这里将列出所有的文件

System.out.println("file==>" + f.getAbsolutePath());

}

}

}

[IMG_257](http://www.cnblogs.com/fenglie/articles/javascript:void(0);" \o "复制代码)

**二、非递归遍历，递归遍历的话是很容易理解的，但是非递归遍历的话估计思路不太容易上来，我当时也苦思良久，发现java有个一个LinkedList这玩意，字面上也大概知道，这个是用来保存一个链接文件的列表，有这个的话就好办了，具体思路是这样：遍历一个文件夹的时候，如果是文件夹，就添加到linkedlist里去，如果是文件则列出；这样该目录的文件和文件夹就遍历完毕了，文件夹全部被保存到linkedlist里；所以剩下的就是遍历linkedlist里边的文件夹的文件，遍历方式同上边的操作一致，如果是文件夹就添加到linkedlist里边（Ps：遍历过程中的linkedlist始终是同一个list），当然每次从列表取出一个文件夹之后需要从列表中删除该文件夹，这里采用linkedlist.removeFirst()来读取，这样是每次读取列表的第一个元素并将其从列表剔除。这样只要遍历到linkedlist的状态是isEmty的时候就说明遍历完毕了。算法如下：**

[IMG_258](http://www.cnblogs.com/fenglie/articles/javascript:void(0);" \o "复制代码)

### 非递归遍历

private void GetDirectorySize(File file) {

LinkedList list = new LinkedList(); //保存待遍历文件夹的列表

GetOneDir(file, list); //调用遍历文件夹根目录文件的方法

File tmp;

while (!list.isEmpty()) {

tmp = (File) list.removeFirst();

//这个地方的判断有点多余，但是为了保险还是给个判断了，正常情况列表中是只有文件夹的

//但是不排除特殊情况，例如：本身是文件夹的目标在压入堆栈之后变成了文件

if (tmp.isDirectory()) {

GetOneDirSize(tmp, list);

} else {

System.out.println("file==>" + tmp.getAbsolutePath());

}

}

}

// 遍历指定文件夹根目录下的文件

private void GetOneDir(File file , LinkedList list){

//每个文件夹遍历都会调用该方法

System.out.println("Dir==>" + f.getAbsolutePath());

File[] files = file.listFiles();

sumdir += 1;

if (files == null || files.length == 0) {

return ;

}

for (File f : files) {

if (f.isDirectory()) {

list.add(f);

} else {

//这里列出当前文件夹根目录下的所有文件

System.out.println("file==>" + f.getAbsolutePath());

}

}

}

[IMG_259](http://www.cnblogs.com/fenglie/articles/javascript:void(0);" \o "复制代码)

推荐使用算法一，效率略高。

[IMG_260](http://www.cnblogs.com/fenglie/articles/javascript:void(0);" \o "复制代码)

import java.io.File;import java.util.ArrayList;

public class FileTest {

private static ArrayList<String> filelist = new ArrayList<String>();

public static void main(String[] args) throws Exception {

String filePath = "E://Struts2";

getFiles(filePath);

}

/\*

\* 通过递归得到某一路径下所有的目录及其文件

\*/

static void getFiles(String filePath){

File root = new File(filePath);

File[] files = root.listFiles();

for(File file:files){

if(file.isDirectory()){

/\*

\* 递归调用

\*/

getFiles(file.getAbsolutePath());

filelist.add(file.getAbsolutePath());

System.out.println("显示"+filePath+"下所有子目录及其文件"+file.getAbsolutePath());

}else{

System.out.println("显示"+filePath+"下所有子目录"+file.getAbsolutePath());

}

}

}

}

[IMG_261](http://www.cnblogs.com/fenglie/articles/javascript:void(0);" \o "复制代码)

## 九Java类的继承

### 概念

继承的语法结构：  
    [修饰符列表] class 子类名 extends 父类名{  
        类体;  
    }  
子类就是当前这个类，父类就是我们要复用的那个类  
java中只支持单继承;c++支持多继承  
一个类如果没有显式继承其他类，则该类默认继承Object  
任何一个类都有一个始祖类Object，所有类的超类

**为什么要继承？**  
    代码的复用，方法的重写、覆盖。  
  
**在有继承关系的两个类中：**  
    private(私有)可以被继承，但无法访问和覆盖，即使在子类中覆盖了父类的方法，在多态里面仍然会调用父类的方法。  
    静态方法有继承无覆盖，覆盖发生在对象的成员方法中，也就是发生在堆内存中，但是静态数据都是在静态内存块中。  
  
发生覆盖的条件：  
    1 发生在有继承关系的两个类之间  
    2 方法必须具有相同的方法名，返回值类型，参数列表(如果参数列表不同，系统会默认这两个重载后的不同的两个方法)  
    3 重写的方法不能比被重写的方法有更低的访问权限  
    4 重写的方法不能比被重写的方法抛出更宽泛的异常  
    5 私有方法不能被覆盖，不调用不报错，调用就报错  
    6 构造方法无法被覆盖，因为构造方法不能被继承  
    7 静态方法不存在覆盖，继承是对象的继承，而不是类的继承。静态方法进入内存以后，不会再有新的内存变化，所以没有覆盖这一说。  
    8 覆盖指的是成员方法，和成员变量无关

## 十Java 中接口

### 概念

接口（Interface），在**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/wei_zhi/article/details/_blank)**编程语言中是一个抽象类型，是抽象方法的集合。接口通常以interface来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

如果一个类只由抽象方法和全局常量组成，那么这种情况下不会将其定义为一个抽象类。只会定义为一个接口，所以接口严格的来讲属于一个特殊的类，而这个类里面只有抽象方法和全局常量，就连构造方法也没有。

范例：定义一个接口

interface A{//定义一个接口

public static final String MSG = "hello";//全局常量

public abstract void print();//抽象方法

}

### 接口的使用

1、由于接口里面存在抽象方法，所以接口对象不能直接使用关键字new进行实例化。接口的使用原则如下：   
（1）接口必须要有子类，但此时一个子类可以使用implements关键字实现多个接口；   
（2）接口的子类（如果不是抽象类），那么必须要覆写接口中的全部抽象方法；   
（3）接口的对象可以利用子类对象的向上转型进行实例化。

对于子类而言，除了实现接口外，还可以继承抽象类。若既要继承抽象类，同时还要实现接口的话，使用一下语法格式：

class 子类 [extends 父类] [implemetns 接口1,接口2,...] {}

## 十一 java多线程

### 概念

1.       **[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank)**和他的API都可以使用并发。可以指定程序包含不同的执行线程，每个线程都具有自己的方法调用堆栈和程序计数器，使得线程在与其他线程并发地执行能够共享程序范围内的资源，比如共享内存，这种能力被称为多线程编程（multithreading）,在核心的C和C++语言中并不具备这种能力，尽管他们影响了JAVA的设计。

2.       线程的生命周期

新线程的生命周期从“新生”状态开始。程序启动线程前，线程一直是“新生”状态；程序启动线程后，线程进入“可运行”状态。“可运行”状态的线程，被认为是正在执行他的任务。

在程序启动线程之前，线程一直处于“等待”状态，只有当另一个线程通知正在等待的线程继续执行时，这个线程才会从“等待”状态恢复到“可运行”状态。

“可运行”状态的线程可以进入“定时等待”状态，等待一个指定的时间段。当时间到达或线程正在等待的某个事件发生时，该线程就会返回“可运行”状态。即使处理器可以使用，处于“定时等待”状态和“等待”状态的线程也不能用它。当处于“可运行”状态的线程正在等待另一个线程执行任务时，如果它提供了可选的等待时间段，则这个线程会进入“定时等待”状态。当另一个线程通知了这个线程，或者当定时的时间段到达时（以先满足的为准），这个线程就会返回到“可运行”状态.。使线程进入“定时等待”状态的另一方法是是处于“可运行”状态的线程睡眠。睡眠线程会在“定时等待”状态维持一个指定的时间段（称为睡眠时间段），过了这段时间，它会返回到“可运行”状态。当线程没有工作要执行时，它会立即睡眠。；例

当线程试图执行某个任务，而任务又不能立即完成，线程就从“可运行”状态转到“阻塞”状态。；例。即使有处理器可供使用，“阻塞”状态的线程也不能使用它。

线程成功完成任务，或者（由于出错）终止了时，“可运行”线程就会进入“终止”状态（有时称“停滞”状态）。

在**[操作系统](http://lib.csdn.net/base/operatingsystem" \o "操作系统知识库" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank)**级别，JAVA的“可运行”状态通常包含两个独立的状态。当线程首先从“新生”状态转到“可运行”状态，线程处于“就绪”状态。当操作系统将线程给处理器时，线程就从“就绪”状态进入“运行”状态（即开始执行），这也被称为“调度线程”。大多数操作系统中，每个线程被赋予一小段处理器时间（时间片）来执行任务。当时间片到达时，线程就会返回到“就绪”状态，而操作系统将另一个线程给予处理器。

3.       线程优先级与线程调度

JAVA的线程优先级范围为MIN\_PRIORITY(常量1)到MAX\_PRIORITY(常量10)，默认是NORM\_PRIORITY(常量5)

4.       创建并执行线程

创建线程推介实现Runnable接口

1. Runnable与Thread类
2. **import** java.util.Random;


6. **public** **class** PrintTask **implements** Runnable
8. {
10. **private** **final** **int** sleepTime; // random sleep time for thread
12. **private** **final** String taskName; // name of task
14. **private** **final** **static** Random generator = **new** Random();


18. **public** PrintTask( String name )
20. {
22. taskName = name; // set task name


26. // pick random sleep time between 0 and 5 seconds
28. sleepTime = generator.nextInt( 5000 ); // milliseconds
30. } // end PrintTask constructor


34. // method run contains the code that a thread will execute
36. **public** **void** run()
38. {
40. **try** // put thread to sleep for sleepTime amount of time
42. {
44. System.out.printf( "%s going to sleep for %d milliseconds.\n",
46. taskName, sleepTime );
48. Thread.sleep( sleepTime ); // put thread to sleep
50. } // end try
52. **catch** ( InterruptedException exception )
54. {
56. System.out.printf( "%s %s\n", taskName,
58. "terminated prematurely due to interruption" );
60. } // end catch


64. // print task name
66. System.out.printf( "%s done sleeping\n", taskName );
68. } // end method run
70. } // end class PrintTask
71. **import** java.lang.Thread;


75. **public** **class** ThreadCreator
77. {
79. **public** **static** **void** main( String[] args )
81. {
83. System.out.println( "Creating threads" );


87. // create each thread with a new targeted runnable
89. Thread thread1 = **new** Thread( **new** PrintTask( "task1" ) );
91. Thread thread2 = **new** Thread( **new** PrintTask( "task2" ) );
93. Thread thread3 = **new** Thread( **new** PrintTask( "task3" ) );


97. System.out.println( "Threads created, starting tasks." );


101. // start threads and place in runnable state
103. thread1.start(); // invokes task1抯 run method
105. thread2.start(); // invokes task2抯 run method
107. thread3.start(); // invokes task3抯 run method


111. System.out.println( "Tasks started, main ends.\n" );
113. } // end main
115. } // end class RunnableTester

### 线程管理与Executor框架 就是线程池

1. **import** java.util.concurrent.Executors;
2. **import** java.util.concurrent.ExecutorService;
4. **public** **class** TaskExecutor
5. {
6. **public** **static** **void** main( String[] args )
7. {
8. // create and name each runnable
9. PrintTask task1 = **new** PrintTask( "task1" );
10. PrintTask task2 = **new** PrintTask( "task2" );
11. PrintTask task3 = **new** PrintTask( "task3" );
13. System.out.println( "Starting Executor" );
15. // create ExecutorService to manage threads
16. ExecutorService threadExecutor = Executors.newCachedThreadPool();
18. // start threads and place in runnable state
19. threadExecutor.execute( task1 ); // start task1
20. threadExecutor.execute( task2 ); // start task2
21. threadExecutor.execute( task3 ); // start task3
23. // shut down worker threads when their tasks complete
24. threadExecutor.shutdown();
26. System.out.println( "Tasks started, main ends.\n" );
27. } // end main
28. } // end class TaskExecutor

### 线程同步

（thread synchronization），协调多个并发线程对共享数据的访问。这种方式同步多个线程，就可以保证访问共享对象的每个线程都能同步地将其他所有线程排除在外，这被称为“互斥”。

另一个方法，使用JAVA内置的监控器（monitor）。每个对象都有一个监控器和监控锁（或内置锁）。监控器保证任何时候监控锁由具有最大可能的唯一一个线程持有。

（2）同步的数据共享：执行原子操作。

**[java]** [view plain](http://blog.csdn.net/kkkloveyou/article/details/6724083" \o "view plain" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank) [copy](http://blog.csdn.net/kkkloveyou/article/details/6724083" \o "copy" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank)

1. // Adds integers to an array shared with other Runnables
3. **import** java.lang.Runnable;


7. **public** **class** ArrayWriter **implements** Runnable
9. {
11. **private** **final** SimpleArray sharedSimpleArray;
13. **private** **final** **int** startValue;


17. **public** ArrayWriter( **int** value, SimpleArray array )
19. {
21. startValue = value;
23. sharedSimpleArray= array;
25. } // end constructor


29. **public** **void** run()
31. {
33. **for** ( **int** i = startValue; i < startValue + 3; i++ )
35. {
37. sharedSimpleArray.add( i ); // add an element to the shared array
39. } // end for
41. } // end method run
43. } // end class ArrayWrite

**[java]** [view plain](http://blog.csdn.net/kkkloveyou/article/details/6724083" \o "view plain" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank) [copy](http://blog.csdn.net/kkkloveyou/article/details/6724083" \o "copy" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank)

1. // Fig 5.2: SharedArrayTest.java
3. // Executes two Runnables to add elements to a shared SimpleArray.
5. **import** java.util.concurrent.Executors;
7. **import** java.util.concurrent.ExecutorService;
9. **import** java.util.concurrent.TimeUnit;


13. **public** **class** SharedArrayTest
15. {
17. **public** **static** **void** main( String[] arg )
19. {
21. // construct the shared object
23. SimpleArray sharedSimpleArray = **new** SimpleArray( 6 );


27. // create two tasks to write to the shared SimpleArray
29. ArrayWriter writer1 = **new** ArrayWriter( 1, sharedSimpleArray );
31. ArrayWriter writer2 = **new** ArrayWriter( 11, sharedSimpleArray );


35. // execute the tasks with an ExecutorService
37. ExecutorService executor = Executors.newCachedThreadPool();
39. executor.execute( writer1 );
41. executor.execute( writer2 );


45. executor.shutdown();


49. **try**
51. {
53. // wait 1 minute for both writers to finish executing
55. **boolean** tasksEnded = executor.awaitTermination(
57. 1, TimeUnit.MINUTES );


61. **if** ( tasksEnded )
63. System.out.println( sharedSimpleArray ); // print contents
65. **else**
67. System.out.println(
69. "Timed out while waiting for tasks to finish." );
71. } // end try
73. **catch** ( InterruptedException ex )
75. {
77. System.out.println(
79. "Interrupted while wait for tasks to finish." );
81. } // end catch
83. } // end main
85. } // end class SharedArrayTest

**[java]** [view plain](http://blog.csdn.net/kkkloveyou/article/details/6724083" \o "view plain" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank) [copy](http://blog.csdn.net/kkkloveyou/article/details/6724083" \o "copy" \t "http://blog.csdn.net/kkkloveyou/article/details/_blank)

1. // Fig.5.3 : SimpleArray.java
3. // Class that manages an integer array to be shared by multiple
5. // threads with synchronization.
7. **import** java.util.Random;


11. **public** **class** SimpleArray
13. {
15. **private** **final** **int** array[]; // the shared integer array
17. **private** **int** writeIndex = 0; // index of next element to be written
19. **private** **final** **static** Random generator = **new** Random();


23. // construct a SimpleArray of a given size
25. **public** SimpleArray( **int** size )
27. {
29. array = **new** **int**[ size ];
31. } // end constructor


35. // add a value to the shared array
37. **public** **synchronized** **void** add( **int** value )
39. {
41. **int** position = writeIndex; // store the write index


45. **try**
47. {
49. // put thread to sleep for 0-499 milliseconds
51. Thread.sleep( generator.nextInt( 500 ) );
53. } // end try
55. **catch** ( InterruptedException ex )
57. {
59. ex.printStackTrace();
61. } // end catch


65. // put value in the appropriate element
67. array[ position ] = value;
69. System.out.printf( "%s wrote %2d to element %d.\n",
71. Thread.currentThread().getName(), value, position );


75. ++writeIndex; // increment index of element to be written next
77. System.out.printf( "Next write index: %d\n", writeIndex );
79. } // end method add


83. // used for outputting the contents of the shared integer array
85. **public** String toString()
87. {
89. String arrayString = "\nContents of SimpleArray:\n";


93. **for** ( **int** i = 0; i < array.length; i++ )
95. arrayString += array[ i ] + " ";


99. **return** arrayString;
101. } // end method toString
103. } // end class SimpleArray

### 读解Thread类API

static int MAX\_PRIORITY

          线程可以具有的最高优先级。

static int MIN\_PRIORITY

          线程可以具有的最低优先级。

static int NORM\_PRIORITY

          分配给线程的默认优先级。

构造方法摘要

Thread(Runnable target)

          分配新的 Thread 对象。

Thread(String name)

          分配新的 Thread 对象。

方法摘要

static Thread currentThread()

          返回对当前正在执行的线程对象的引用。

 ClassLoader getContextClassLoader()

          返回该线程的上下文 ClassLoader。

 long getId()

          返回该线程的标识符。

 String getName()

          返回该线程的名称。

 int getPriority()

          返回线程的优先级。

 Thread.State getState()

          返回该线程的状态。

 ThreadGroup getThreadGroup()

          返回该线程所属的线程组。

static boolean holdsLock(Object obj)

          当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。

 void interrupt()

          中断线程。

static boolean interrupted()

          测试当前线程是否已经中断。

 boolean isAlive()

          测试线程是否处于活动状态。

 boolean isDaemon()

          测试该线程是否为守护线程。

 boolean isInterrupted()

          测试线程是否已经中断。

 void join()

          等待该线程终止。

 void join(long millis)

          等待该线程终止的时间最长为 millis 毫秒。

 void join(long millis, int nanos)

          等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。

 void resume()

          已过时。 该方法只与 suspend() 一起使用，但 suspend() 已经遭到反对，因为它具有死锁倾向。有关更多信息，请参阅为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？。

 void run()

          如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。

 void setContextClassLoader(ClassLoader cl)

          设置该线程的上下文 ClassLoader。

 void setDaemon(boolean on)

          将该线程标记为守护线程或用户线程。

static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)

          设置当线程由于未捕获到异常而突然终止，并且没有为该线程定义其他处理程序时所调用的默认处理程序。

 void setName(String name)

          改变线程名称，使之与参数 name 相同。

 void setPriority(int newPriority)

          更改线程的优先级。

 void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)

          设置该线程由于未捕获到异常而突然终止时调用的处理程序。

static void sleep(long millis)

          在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）。

static void sleep(long millis, int nanos)

          在指定的毫秒数加指定的纳秒数内让当前正在执行的线程休眠（暂停执行）。

 void start()

          使该线程开始执行；Java 虚拟机调用该线程的 run 方法。

 void stop()

          已过时。 该方法具有固有的不安全性。用 Thread.stop 来终止线程将释放它已经锁定的所有监视器（作为沿堆栈向上传播的未检查 ThreadDeath 异常的一个自然后果）。如果以前受这些监视器保护的任何对象都处于一种不一致的状态，则损坏的对象将对其他线程可见，这有可能导致任意的行为。stop 的许多使用都应由只修改某些变量以指示目标线程应该停止运行的代码来取代。目标线程应定期检查该变量，并且如果该变量指示它要停止运行，则从其运行方法依次返回。如果目标线程等待很长时间（例如基于一个条件变量），则应使用 interrupt 方法来中断该等待。有关更多信息，请参阅《为何不赞成使用 Thread.stop、Thread.suspend 和 Thread.resume？》。

 void stop(Throwable obj)

          已过时。 该方法具有固有的不安全性。请参阅 stop() 以获得详细信息。该方法的附加危险是它可用于生成目标线程未准备处理的异常（包括若没有该方法该线程不太可能抛出的已检查的异常）。有关更多信息，请参阅为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？。

 void suspend()

          已过时。 该方法已经遭到反对，因为它具有固有的死锁倾向。如果目标线程挂起时在保护关键系统资源的监视器上保持有锁，则在目标线程重新开始以前任何线程都不能访问该资源。如果重新开始目标线程的线程想在调用 resume 之前锁定该监视器，则会发生死锁。这类死锁通常会证明自己是“冻结”的进程。有关更多信息，请参阅为何 Thread.stop、Thread.suspend 和 Thread.resume 遭到反对？。

 String toString()

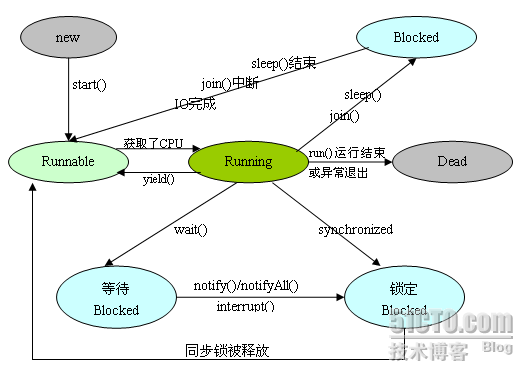
          返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

static void yield()

          暂停当前正在执行的线程对象，并执行其他线程。

### 线程的状态转换图

线程在一定条件下，状态会发生变化。线程变化的状态转换图如下：



1、新建状态（New）：新创建了一个线程对象。

2、就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3、运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

4、阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待池中。

（二）、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（三）、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5、死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

七、线程的调度

1、调整线程优先级：Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

static int MAX\_PRIORITY

          线程可以具有的最高优先级，取值为10。

static int MIN\_PRIORITY

          线程可以具有的最低优先级，取值为1。

static int NORM\_PRIORITY

          分配给线程的默认优先级，取值为5。

Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级。

每个线程都有默认的优先级。主线程的默认优先级为Thread.NORM\_PRIORITY。

线程的优先级有继承关系，比如A线程中创建了B线程，那么B将和A具有相同的优先级。

JVM提供了10个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用Thread类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。

2、线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。

3、线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。

4、线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

5、线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

6、线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。

注意：Thread中suspend()和resume()两个方法在JDK1.5中已经废除，不再介绍。因为有死锁倾向。

### 常见线程名词解释

主线程：JVM调用程序mian()所产生的线程。

当前线程：这个是容易混淆的概念。一般指通过Thread.currentThread()来获取的进程。

后台线程：指为其他线程提供服务的线程，也称为守护线程。JVM的垃圾回收线程就是一个后台线程。

前台线程：是指接受后台线程服务的线程，其实前台后台线程是联系在一起，就像傀儡和幕后操纵者一样的关系。傀儡是前台线程、幕后操纵者是后台线程。由前台线程创建的线程默认也是前台线程。可以通过isDaemon()和setDaemon()方法来判断和设置一个线程是否为后台线程。

### Java中的多线程有三种实现方式： 1.继承Thread类，重写run方法。Thread本质上也是一个实现了Runnable的实例，他代表一个线程的实例，并且启动线程的唯一方法就是通过Thread类的start方法。 2.实现Runnable接口，并实现该接口的run()方法.创建一个Thread对象，用实现的Runnable接口的对象作为参数实例化Thread对象，调用此对象的start方法。 3.实现Callable接口，重写call方法。Callable接口与Runnable接口的功能类似，但提供了比Runnable更强大的功能。有以下三点 1）.Callable可以在人物结束后提供一个返回值，Runnable没有提供这个功能。 2）.Callable中的call方法可以抛出异常，而Runnable的run方法不能抛出异常。 3）.运行Callable可以拿到一个Future对象，表示异步计算的结果，提供了检查计算是否完成的方法。 需要注意的是，无论用那种方式实现了多线程，调用start方法并不意味着立即执行多线程代码，而是使得线程变为可运行状态。

**run start的区别**  
start方法是启动一个线程，而线程中的run方法来完成实际的操作。  
如果开发人员直接调用run方法，那么就会将这个方法当作一个普通函数来调用，并没有多开辟线程，开发人员如果希望多线程异步执行，则需要调用start方法。  
  
**sleep wait的区别**  
1.两者处理的机制不同，sleep方法主要是，让线程暂停执行一段时间，时间一到自动恢复，并不会释放所占用的锁，当调用wait方法以后，他会释放所占用的对象锁，等待其他线程调用notify方法才会再次醒来。  
2.sleep是Threa的静态方法，是用来控制线程自身流程的，而wait是object的方法，用于进行线程通信。  
3.两者使用的区域不同。sleep可以在任何地方使用，wait必须放在同步控制方法，或者语句块中执行。

**synchronized notify wait的运用**  
synchronized关键字有两种用法，synchronized方法和synchronized语句块。  
public synchronized void function(){}  
synchronized(object){}  
当某个资源被synchronized所修饰，线程1线程2等多个线程在共同请求这个资源，线程1先请求到，调用了对象的wait方法释放了对象的锁，此时线程2可以对这个对象进行访问，在工作结束时可以调用对象的notify方法，唤醒等待队列中正在等待的线程，此时被唤醒的线程将会再一次拿到对象锁，对对象进行操作。可以调用notifyAll方法，唤醒等待队列中的所有线程。