Java中的File类和递归及接口FileFilter、FilenameFilter

1. **File类**是存在于**java.io包**中的，直接继承于**java.lang,Object类**，实现了**Comparable<File>接口**，显然就会有**compareTo方法**。
2. 对一个**文件**或**文件夹**进行封装成**File对象**时，无论这个文件或者文件夹是否存在，都会把其**封装成对象**，在封装成对象的时候**并不会检测其是否存在**，当然如果不存在，就不会自动创建，故一般接下来需要进行**健壮性的判断**。与**IO流操作**不同，如果是文件输入流即文件读取流，指定的文件不存在，必会抛出异常**java.io.FileNotFoundException**，而文件输出流即文件写入流指定的文件不存在，就会**自动创建此文件**。
3. 如果一个文件对象不存在，对于文件和文件夹的创建方式需要调用不同的方法：
4. 首先利用**exists()方法**，判断是否存在。
5. 然后，如果是具体的文件，创建需要调用**createNewFile()方法**。
6. 如果是文件夹，但是如果是一层，则利用**mkdir()方法或mkdirs()方法均可**；如果是多层目录文件夹，则只能利用。

注意：**creatNewFile()、mkdir()、mkdirs()方法**都是返回Boolean值，true表示成功，false表示失败。

1. **File对象**可以作为IO流的构造方法的**参数**。**IO流**只能操作**文件**的内容，不能操作文件的**属性信息**，同时**IO流**也不能操作**文件夹**，这就是**File类**出现的意义。
2. **File类主要就是用来操作操作文件夹和文件的属性。**
3. **而IO流主要是操作具体文件中的内容。**
4. 对于封装File对象时的健壮性的判断：
5. 对**某个文件**进行封装成File对象：

**File file = new File("e:\\abc\\def\\222.txt");**

**if(!file.exists()) {**

**file.createNewFile();//创建成功返回true**

**}**

1. 对**某个文件夹**进行封装成File对象：

**File dir = new File("e:\\abc\\def\\ggg");**

**if(!dir.exists()) {**

**dir.mkdirs();//创建成功返回true**

**}**

**注意：mkdir方法只可以创建一层文件夹，而mkdirs可以创建多层文件夹，如果利用mkdir方法去创建封装了多层文件夹的对象，则返回false。但是可以利用mkdirs方法创建单层文件夹。**

1. **File类的字段Fields介绍：File类有4个静态字段，主要是分隔符。**
2. **separator和separatorChar：**

**两者都代表的是路径中的层次分隔符，在Unix系统上代表/，在Windows系统上代表\。两者的区别在于，separator相当于字符串String形式，而separatorChar只是代表字符char。**

**File.separator等同于public static final String FILE\_SEPARATOR = System.getProperty("file.separator");**



1. **pathSeparator和pathSeparatorChar：**

**同理，一个字符，一个字符串，功能都一样，都是代表分号 ;** 。

**相当于：public static final String PATH\_SEPARATOR = System.getProperty("path.separator");**





1. **File类的构造方法Constructors介绍**：
2. **最简单的参数filename：可以把一个文件夹或者一个文件封装成对象。**

**理解：注意封装的是最后一层文件夹或者直接就是文件本身。如删除这个对象，只能是删除文件本身，或者删除的是最后一层文件夹。**

**绝对路径创建，当然如果只是个文件名，则默认是在当前文件夹下创建此文件。**



1. **File（String parent , String child**）:把一个完成路径**分成两部分**。

**相对路径创建。**



1. 父目录可以是**File对象**。

把父目录封装成对象，更加便于操作父目录。



1. 把一个URL对象封装成File对象。

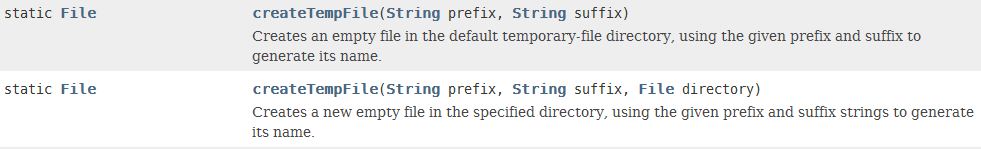
****

**注意**：File没有空参数的构造方法，所以必须指定路径。但是File对象的创建只是建立一种联系，文件是否存在不进行检测。

1. **File类的成员方法Methods介绍**：
2. **创建方法**：创建此文件对象，如果已经存在，则不创建，返回false，如果文件不存在，则便创建此文件对象，返回true。
3. **创建文件：①creatNewFile（）方法：只可以创建此文件，对应的文件夹无法创建，返回false；如果文件已经存在，则返回false；如果文件命名不规范，则创建失败。**



②利用**静态方法**创建对象：



示例：

File tempFile = File.*createTempFile*("aaa", ".txt");

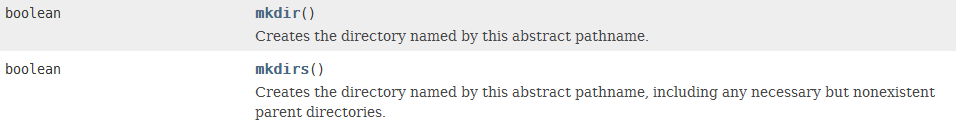
File tempFile2 = File.*createTempFile*("ccc", ".mp3", **new** File("e:\\aaaa"));

System.***out***.println(tempFile.getAbsolutePath());

System.***out***.println(tempFile2.getAbsolutePath());



1. **创建文件夹**：创建**单层文件夹和多层文件夹**。



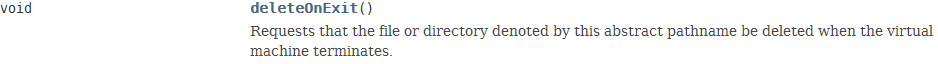
1. **删除方法**：

**①delete方法**：返回的仍然是boolean值。

**windows系统输出文件是从里往外删除的，只有文件夹中没有文件时，此文件夹才可以删除掉。当文件正在被流操作时，delete方法不能删掉。删除文件或文件夹，都是delete方法。**



②**deleteOnExit**：指定此文件对象**当虚拟机退出时**自动删除。



1. **获取方法：**
2. **获取名字**：**getName（）方法：获取的是封装的文件名字或者（最低级）的文件夹名字。**



1. **获取路径及根据路径创建新的File对象：**
   1. **获取绝对路径:getAbsoluPath（），返回字符串。**



紧接着，**getAbsoluteFile**：按照**此对象的绝对路径创建一个新File对象**，等价于。

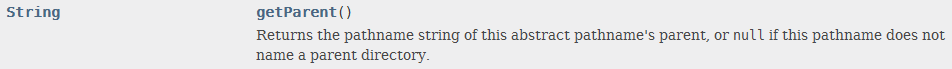


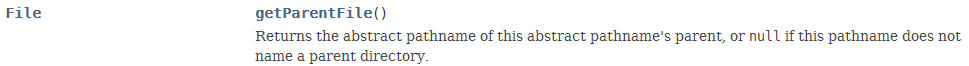
* 1. 获取**路径**：如果是绝对路径就返回绝对路径，否则返回相对路径。



* 1. **获取文件夹和文件夹对象**：（一个返回String路径一个返回File对象）如果没有相对路径，则返回null。

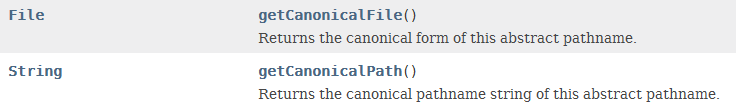
**getParent和getParentFile方法：**





* 1. 获取CanonicFile和CanonicalPath：

**canonic 英[kə'nɒnɪk]美 [kə'nɑnɪk] adj. 轮唱的；依教规的**



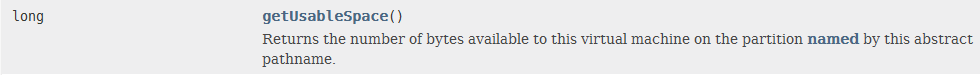
1. **获取文件大小**：单位是字节，如果是**文件夹**，则返回结果是0。



1. **获取分区（c、d、e等）的空间大小**:
   1. 获取**该盘符**的总的空间大小：



* 1. **获取虚拟机可以使用此盘符的空间**：某些情况下，与**getFreeSpace方法**等价。



* 1. 获取盘符**未分配的**空间：



1. **获取上次修改时间**:返回的是毫秒值。



1. **判断方法**：
2. **存在性判断:**



1. **文件或文件夹判断：注意:当真正不存在时，isFile和isDirectory都会返回false，只有在File对象所指的文件或文件夹真实存在时，才会正确返回到底是文件还是文件夹。所以判断之前最好是利用exists方法进行健壮性判断。**
   1. 文件判断方法：



* 1. 文件夹判断方法：



1. **隐藏文件**判断方法：



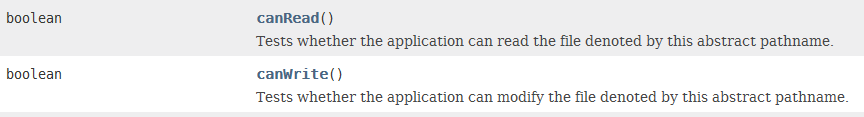
1. **判断路径名是否是绝对路径**：



1. **可执行性判断**：canExecute（）。



1. **可读写判断**：canRead或canWrite方法。



1. **重命名方法**：**renameTo（File dest）**,注意参数是另外一个对象。



1. **比较方法**：
   1. **compareTo 方法**，依照**路径**进行比较，**返回0，负数，正数**。（**即cscii码的差值**）



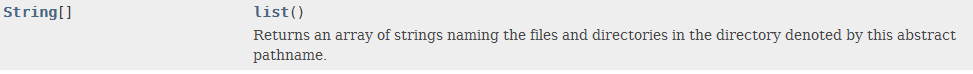
**②相等性判断**：当且仅当两个**File对象的路径完全相同**时，返回true。



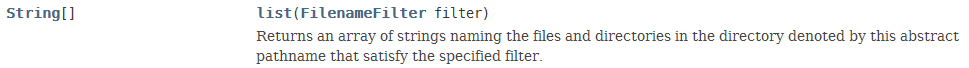
1. **列举方法：list和listFiles方法**：调用此方法的**File对象必须是文件夹**，且不能是隐藏的。如果是文件或是隐藏的文件夹，则抛出java.lang.NullPointerException异常。如果此文件夹内部为空，则会返回一个**空的字符串数组或File数组**。

**list返回字符串数组，而listFiles返回的是File数组。-**

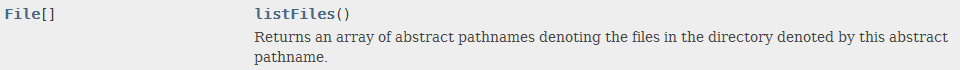
1. **list方法**：此**文件夹对象**下的**文件和子文件夹名**列举出来，返回一个**String数组**。



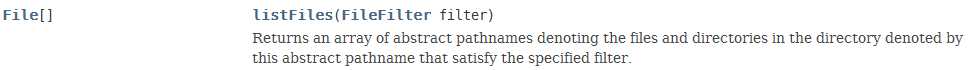
带有**文件名过滤器**的list方法：



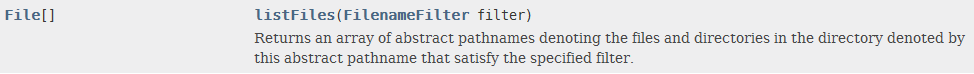
1. **listFiles（）: 把该文件夹下的所有文件和子文件夹封装成File对象，返回一个File数组。**



带有**文件过滤器**的listFiles（FileFilter filter）方法;



带有**文件名过滤器**的**listFiles（FilenameFilter filter）方法**：



1. **静态方法listRoots**：利用**类名**直接调用，获取**盘符**，**不依赖于任何一个对象**。



如  **File[] files = File.listRoots();**

**for(File file1 : files) {**

**System.out.println(file1);**

**}**

结果为: 

1. **toXxxx方法：**
2. **toString方法**：文件路径名。



1. toURI方法：返回一个URI对象。



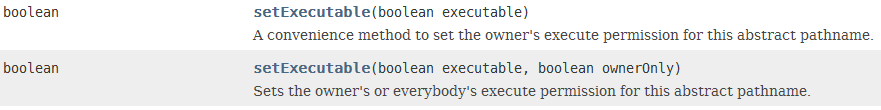
1. toPath方法：返回一个Path对象。



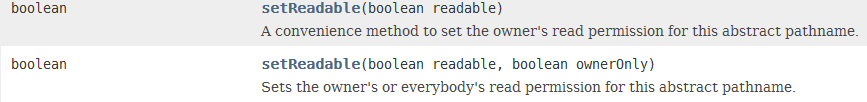
1. **setXxxx方法：**
2. **setModified**: **设置上次修改时间**。



1. **setExecutable方法**：

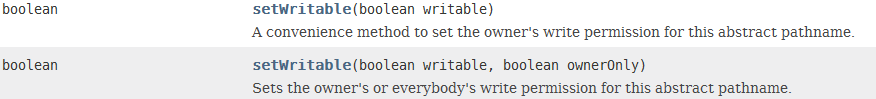


1. **setReadable方法**：





1. **setWritable方法**：



1. **过滤器**：**实现接口创建类**。

如果需要列出**所需要的文件**，就会使用到过滤器，先创建个**类**（过滤器）实现接口**FilenameFilter接口或者FileFilter**，并根据需求实现其中的方法**accep方法**。然后创建此**类**的一个对象传进File类的**list方法或者listFiles**里面去即可根据过滤器而找出需要的**文件对象**或者**名称**。

1. **FileFilter接口**：

**只有一个抽象方法：accept（File）方法，检测**



参数：**File对象（路径名字）**。返回：当且仅当文件对象列表中包含这个路径名，才返回true。

1. **FilenameFilter接口**:

只有一个**抽象方法**:接口中的方法都是抽象方法：**accpt（）方法**。



1. **文件过滤器**练习:
2. **文件名过滤器:**

创建一个**文件名字过滤器**：过滤出txt文件。

package january;

import java.io.File;

import java.io.FilenameFilter;

public class FileFilterByName implements FilenameFilter {

@Override

public boolean accept(File arg0, String arg1) {

return arg1.endsWith(".txt");

}

}

主方法中：list方法中传入**一个过滤器对象**。

File file = new File("e:\\");

**String[] names = file.list(new FileFilterByName());**

for(String name : names) {

System.out.println(name);

}

1. **文件过滤器**：

创建一个**文件过滤器**:过滤出**文件夹**。

package january;

import java.io.File;

import java.io.FileFilter;

public class FileFilterDemo111 implements FileFilter {

@Override

public boolean accept(File arg0) {

return arg0.isDirectory();

}

}

主方法中:  
File file = new File("e:\\");

File[] names2 = file.listFiles(new FileFilterDemo111());

for(File name : names2) {

System.out.println(name);

}

1. 利用递归的方法，把一个目录文件中的所有的子文件夹及其以“.java”结尾的文件的路径都输出来。关键是文件过滤器。

**private** **static** **void** printName(File dir) {

**if** (**null** == dir || !dir.exists()) {

**return**;

}

System.***out***.println(dir.getAbsolutePath());

**if** (dir.isDirectory()) {

File[] files = dir.listFiles(new FileFilter() {

@Override

public boolean accept(File file) {

Boolean flag = file.isDirectory()

|| file.getName().endsWith(".java");

return flag;

} });

**for** (File file : files) {

*printName*(file);

}}}

1. **递归:**

**递归：在调用一个方法的过程中，又出现直接或者间接地调用该方法本身，称为方法的递归调用。**

1. 在调用方法f的过程中，又调用f方法，这是**直接调用本方法**；
2. 在调用方法f的过程中，又调用了f2 方法，而在调用f2 方法中过程中，又要调用f方法，这就是**间接调用**。

**最常见的就是在一个方法体内又调用了本身（直接）或者是调用了此方法的另外一个方法（间接）。**

**注意： 递归一定要明确条件，容易栈溢出StackOverflowException。**

**注意递归的次数，因为调用的方法都会进栈，调用多少次就会有多少个方法进入栈，如果递归次数过多，会导致栈溢出异常。**

**如果一个方法重复调用，且每次调用时，参与运算的结果与上一次调用有关，此时就可用递归来解决问题。**

1. 递归练习：
2. 深度遍历：

private static void DieDaiMethodDemo(File dir) {

System.out.println(dir.getAbsolutePath());

File[] files = dir.listFiles();

for(File file1 : files) {

if(file1.isDirectory()) {

**DieDaiMethodDemo(file1);**

}

else

System.out.println(file1.getAbsolutePath()+"$$$$$$$$$"+file1.isFile());

}

}

1. **求阶乘**。

private static int Multi(int num) throws Exception {

if(num < 0) {

throw new Exception();

}

if(num > 1) {

return num\*Multi(--num);

}

return 1;

}

1. **删除目录**: **删除一个文件夹（或者叫做删除目录）（里面包含文件和其他文件夹）。**删除目录： **windows系统输出文件是从里往外删除的，只有文件夹中没有文件时，此文件夹才可以删除掉。当文件正在被流操作时，delete方法不能删掉。删除文件或文件夹，都是delete方法。**

Windows系统中删除文件夹，都是从内往外删除，先把文件夹内部的删除后，再删除文件夹。这一点正符合了递归的思想，递归正是一致通过反复的调用本身，直达最里面，然后再一步一步的往外走。

private static void deleteDirtory(File dir) {

File[] files = dir.listFiles();

for (File file : files) {

if (file.isDirectory()) {

**deleteDirtory(file);**

}

else

System.out.println("file : "+file+" 删除 "+file.delete());

}

System.out.println("dir : "+dir+"删除"+dir.delete());

}

1. **Hanio（汉诺）塔问题：**

**注意：递归是倒着来的，最后的需求是把所有的盘子移到C处，至于第一次是把一个盘子移到B处还是C处就靠递归的规律了。发现：num为奇数时，先是A🡺C,num为偶数时，先是A🡺B.**

**/\*\***

**\* 汉诺塔问题分析：**

**\* 64个盘子，从大到小，从下到上排列着，大的在下，小的在上，要从A处移到C处。**

**\* 解决思路： 移动64个到c处，如果能够移动63个到B处，然后把第64个移动到c处，再把63个移动到C处。**

**\* 存在三步：①移动n-1个从源移动到借助地B，**

**\* ② 移动第n个盘子从源到目的地；**

**\* ③ 再移动n-1个从借助地移动到目的地即可。**

**注意在移动n个、n-1个、n-2个……等等的过程中，源始终不变都是A，但是目的地会不断发生变化，因为移动n个时，目的地是C，但是移动n-1个时，目的地就是B，所以移动相邻个数n、n-1个时，目的地和借助地不断交替。**

**\***

**\* 按照递归思想，移动64个从A处到C处，那么只需要完成移动63个从一处移动到另一处，。。。。。。。。**

**\* 从而移动3个从A处移到C处，先移动2个从A处到B处，再把第三个移动到C处，再把2个从Ｂ处移到Ｃ处，**

**\* 移动２个从Ａ处移动到Ｃ处，首先移动一个从Ａ处移到Ｂ处，然后移动第二个从Ａ处到Ｃ处，最后是从Ｂ处移动到Ｃ处。**

**\* 最终究归结到把一个从一个地方移到另外一个地方，即可。**

代码：

package january;

public class HanioDemo {

**static long count = 0;// 用于记录需要多少次移动次数**

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

int num = 3;

**hanioMoves(num,** "A"**,** "C" **,** "B"**);**

System.out.println("移动" + num + "个盘子，总共移动的次数 :" + count);

}

private static void hanioMoves(int num, String from, String to, String by)

throws Exception {

if (num <= 1) {

if (num == 1) {

move(from, to);

}

else {

throw new Exception("输入数据有误。");

}

}

else {

**hanioMoves(--num, from, by, to);// 把num-1 个从A处移到B处，利用C**

**move(from, to);// 把最下面的一个从A处移到C处**

**hanioMoves(num, by, to, from);// 再把num-1个从B处移动到C处利用A**

}

}

private static void move(String from, String to) {

count++;

**System.out.println(from + " ===> " + to); // 简单的一次移动动作**

}

}

**移动30个，就需要1073741823次移动**。如果num为x，次数为count1，则如果num变为x + 1，则num就是**num\*2 + 1**. **次数是上次的两倍再加一。**

**num为3时，结果为：**

