## JavaSE文档

1. **异常**

1.1、什么是异常？

在程序中，错误可能产生于程序员没有预料到的各种情况，或者超出程序员可控范围的环境。

为了能够及时有效地处理程序中的运行错误，Java 专门引入了异常类。

例如:

开车上班,走啊走啊！突然前面出现石头。那么车就不能正常行驶了....

又比如我们程序中的文件拷贝，如果文件路径不存在！那么我们应该怎么处理等等....

1.2、异常中的几个关键字:

**Java异常机制用到的几个关键字：try、catch、finally、throw、throws。**

try: 用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块

内发生异常时，异常就被抛出。

catch: 用于捕获异常。catch用来捕获try语句块中发生的异常。

finally: finally语句块总是会被执行。它主要用于回收在try块里打开的物力资源(如数据库

连接、网络连 接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch

块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，

则就不会跳回执行，直接停止。

 throw : 用于抛出异常。

throws: 用在方法签名中，用于声明该方法可能抛出的异常。

1.3、throw和throws区别

throw语句用在方法体内，表示抛出异常，由方法体内的语句处理。  
throws语句用在方法声明后面，表示再抛出异常，由该方法的调用者来处理。  
  
throws主要是声明这个方法会抛出这种类型的异常，使它的调用者知道要捕获这个异常。  
throw是具体向外抛异常的动作，所以它是抛出一个异常实例。  
  
throws说明你有那个可能，倾向。  
throw的话，那就是你把那个倾向变成真实的了。

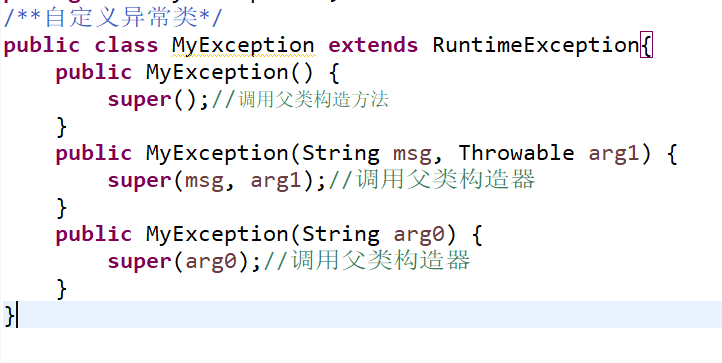
同时：  
1、throws出现在方法头;而throw出现在方法体。  
2、throws表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执

行throw则一定抛出了某种异常。  
3、两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛

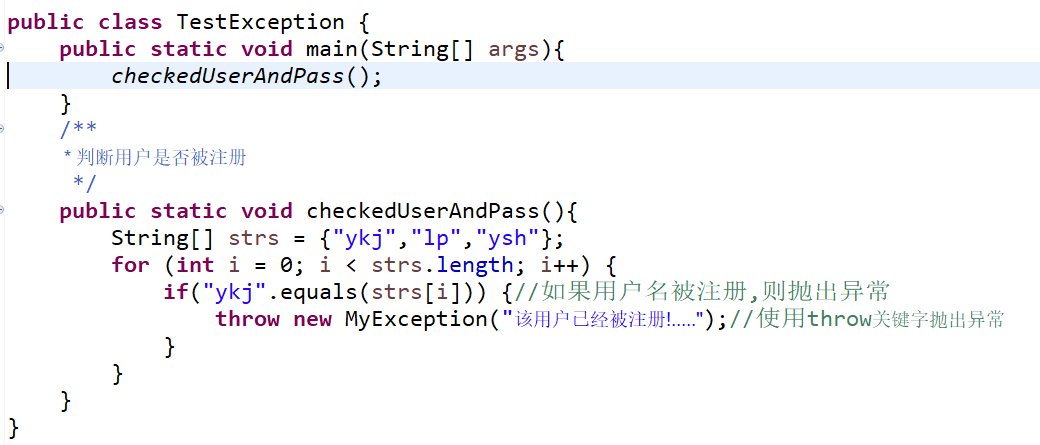
出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

1.4、自定义异常

自定义异常类:



测试类:



1. **IO流**

2.1、IO流的概述

IO流即输入流(input)和输出流(output)。

2.2、IO流的分类

2.1按照处理数据单位不同分为: 字节流(byte) 和 字符流(char)

字节流指的是在读写文件内容时是一个字节一个字节(byte)进行读写的

而字符流则是在读写文件的内容时是以一个字符一个字符(char)进行续写的

2.2按照流向分为: 输入流(Input) 和 输出流(output)

输入流指定是把文件中的内容读取到java程序中的过程

而输出流则是指把程序中的内容写入到指定的文件中

2.3按照功能的不同分为 节点流(低级流) 和 处理流(高级流)

通过程序直接向程序中读取文件或者向文件中写入数据的方式叫做节点流

在程序读取文件或者向文件中写入数据的流进行包装的流叫做处理流

|  |  |  |
| --- | --- | --- |
|  | 字节流 | 字符流 |
| 输入流 | InputStream | Reader |
| 输出流 | OutputStream | Writer |

2.3、File类

1. **构造方法**

|  |
| --- |
| **[File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "File(java.io.File, java.lang.String))**([File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \o "java.io 中的类) parent, [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) child)            根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。 |
| **[File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "File(java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) pathname)            通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。 |
| **[File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "File(java.lang.String, java.lang.String))**([String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) parent, [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) child)            根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。 |

1. **方法**

**2.1、创建文件和目录**

|  |  |
| --- | --- |
| boolean | **[mkdir](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "mkdir())**()            创建单级目录 |
| boolean | **[mkdirs](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "mkdirs())**()            创建多级目录 |
| boolean | **[createNewFile](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "createNewFile())()**在指定的目录下创建文件 |

**2.2、删除文件或者目录**

|  |  |
| --- | --- |
| boolean | **[delete](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "delete())**()            删除此抽象路径名表示的文件或目录。 |

**2.3、获取文件路径的方法**

|  |  |
| --- | --- |
| [File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \o "java.io 中的类) | **[getAbsoluteFile](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "getAbsoluteFile())**()            获取绝对路径返回File类型 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[getAbsolutePath](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "getAbsolutePath())**()            获取绝对路径返回String类型 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[getParent](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "getParent())()**获取当前文件的父路径 |
| [File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \o "java.io 中的类) | **[getParentFile](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "getParentFile())()**获取当前文件的父路径，返回一个File类型 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[getPath](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "getPath())()**获取当前路径，返回一个String类型 |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类) | **[getName](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "getName())()** 获取文件的名称和后缀 |
| long | **[length](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "length())()**获取指定文件的总字节长度 |

**2.4、关于文件和目录的判断**

|  |  |
| --- | --- |
| boolean | **[exists](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "exists())**()           判断当前目录或者文件是否存在 |
| boolean | **[isDirectory](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "isDirectory())()**      判断当前抽象路径是否为目录 |
| boolean | **[isFile](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "isFile())()**判断当前抽象路径是否为文件 |
| boolean | **[isHidden](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "isHidden())()**判断当前抽象路径下的文件是否为隐藏文件 |
| boolean | **[isAbsolute](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "isAbsolute())()**判断当前抽象路径是否为绝对路径 |
| boolean | **[canRead](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "canRead())()**判断当前抽象路径下的文件是否为只读文件 |
| boolean | **[canWrite](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "canWrite())()**判断当前抽象路径下的文件是否为只写文件 |

**2.5、获取子目录或者子文件**

|  |  |
| --- | --- |
| [String](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/String.html" \o "java.lang 中的类)[] | **[list](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "list())**()          获取指定目录下的所有子文件或者子目录的名称，返回字符串数组 |
| [File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \o "java.io 中的类)[] | **[listFiles](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "listFiles())**()           获取指定目录下的所有子文件或者子目录绝对路径的File类型数组 |
| static [File](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \o "java.io 中的类)[] | **[listRoots](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/io/File.html" \l "listRoots())**()           返回当前操作系统下所有盘符 |

练习如下:

1.删除指定目录下的所有文件，包括其子目录中的文件

2.拷贝指定目录下的所有文件(不包含其子目录中的文件)到另一个目标目录中

1、输入一个目录

2、判断目录是否存在，是否是一个真实的目录

3、获取当前目录下的所有子文件

4、指定一个目录，用来存放赋值的文件

5、判断哪些文件哪些文件哪些是目录

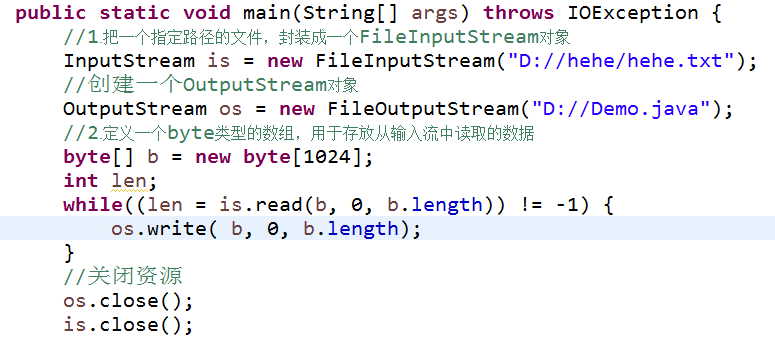
6、如果是一个文件，对象文件进行读写(文件的创建)

2.4.字节输入和输出流

InputStream(字节输入流)

OutputStream(字节输出流)

案例:

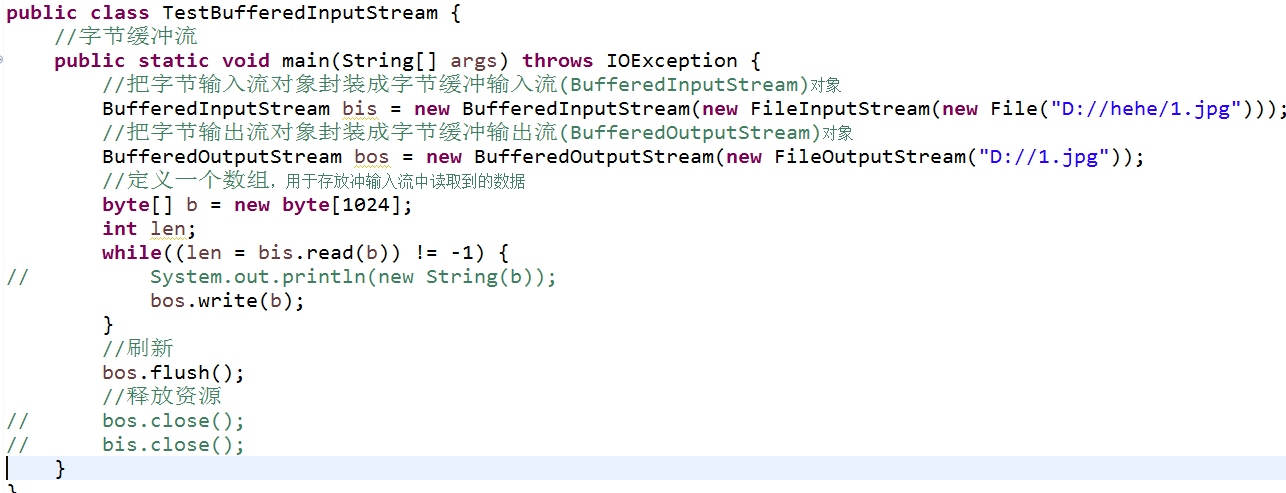


2.5、字节缓冲输入流和输出流

BufferedInputStream(字节缓冲输入流)

BufferedOutputStream(字节缓冲输出流)

案例:



2.6、字符输入和输出流

Reader(字符输入流)

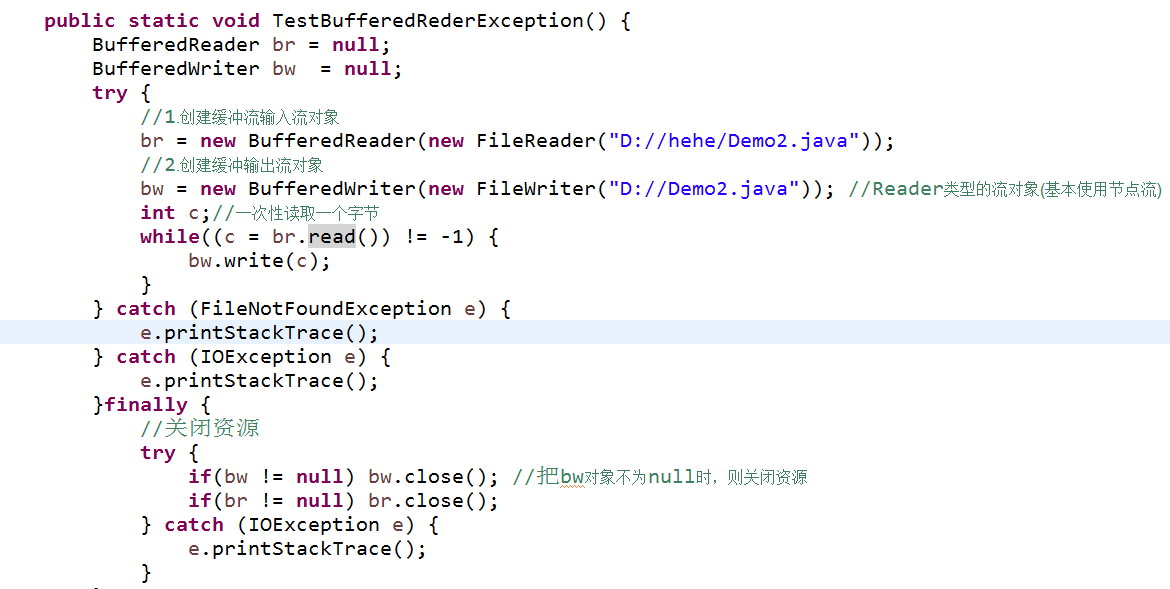
Writer(字符输出流)

案例:



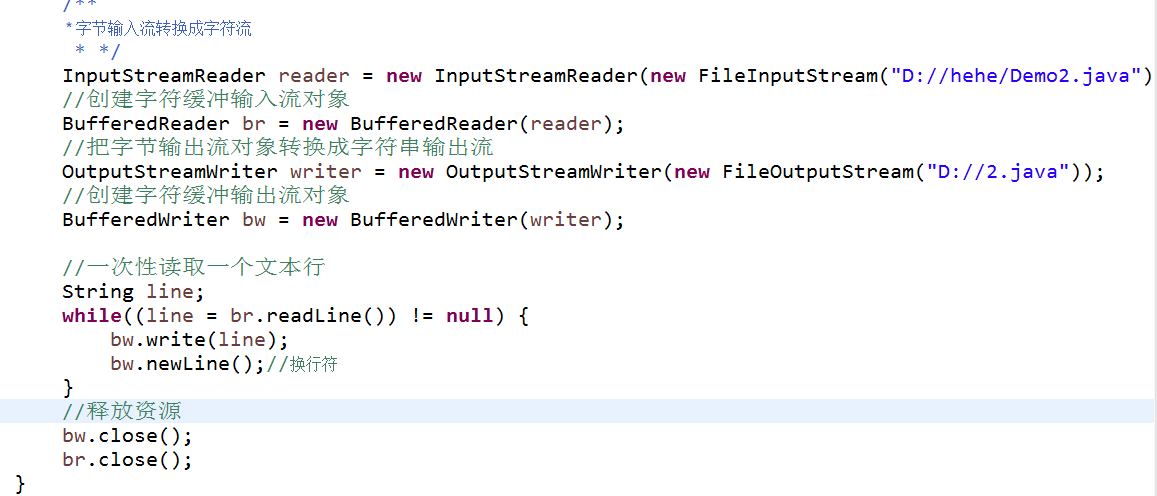
2.7、字符缓冲输入和输出流

案例:



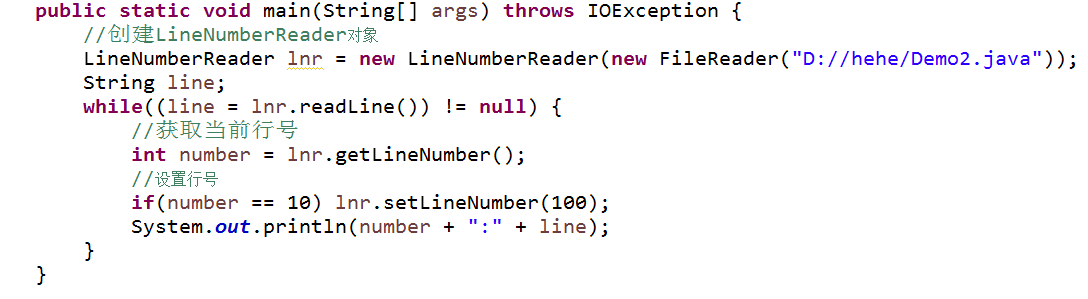
2.8、转换流(InputStreamReader和OutputStreamWriter)

可以把字节输入和输出流转换成字符流进行读和写



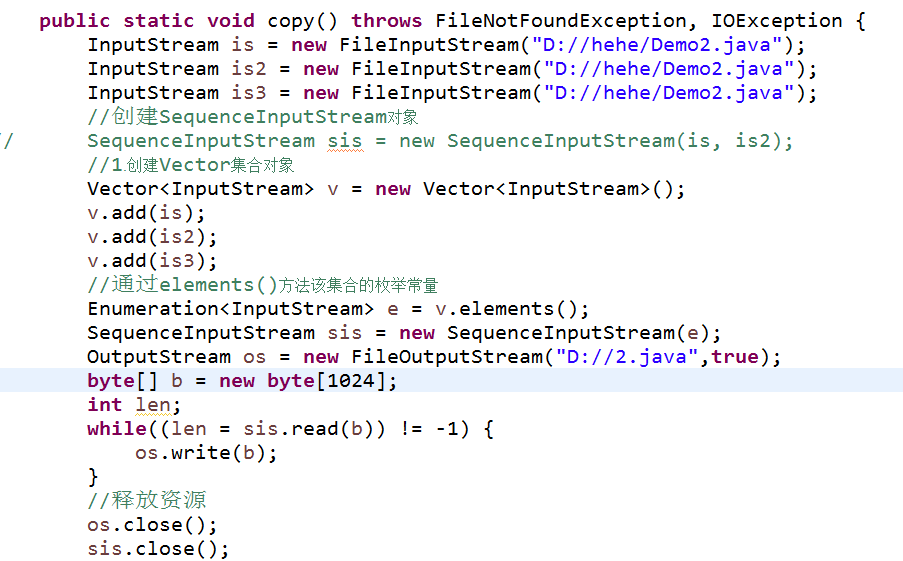
2.9、行号流(LineNumberReader)

可以一次性读取一行数据的通过获取并且设置行号



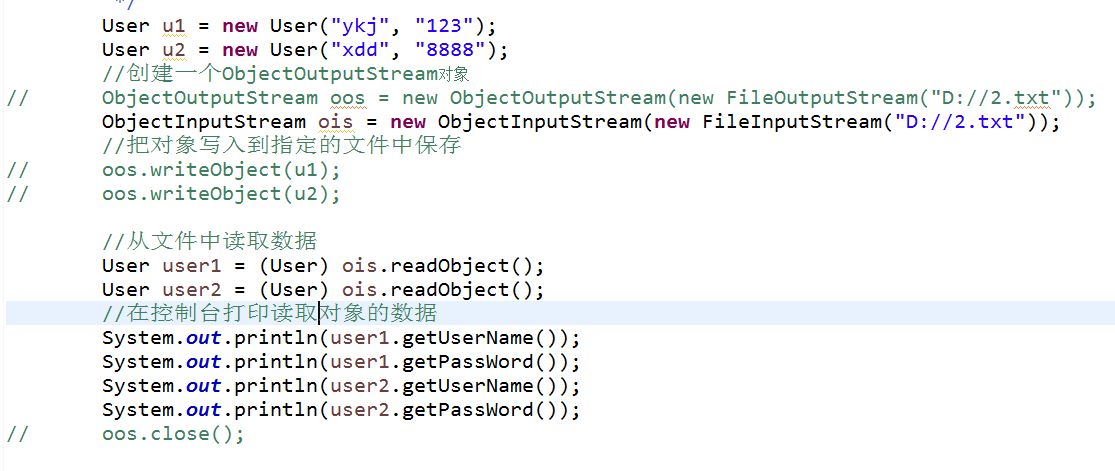
2.10、合并流(SequenceInputStream)

可以把多个输入流进行合并，然后有顺序的写入到文件中



2.11、对象流(ObjectInputStream和ObjectOutputStream）

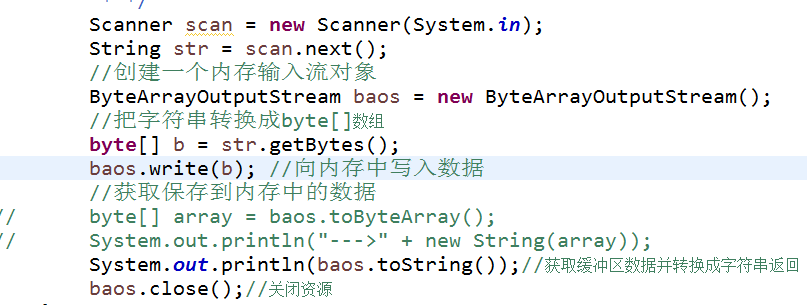
可以向文件中写入或者读取一个对象



2.12、内存流(ByteArrayOutputStream和ByteArrayInputStream）

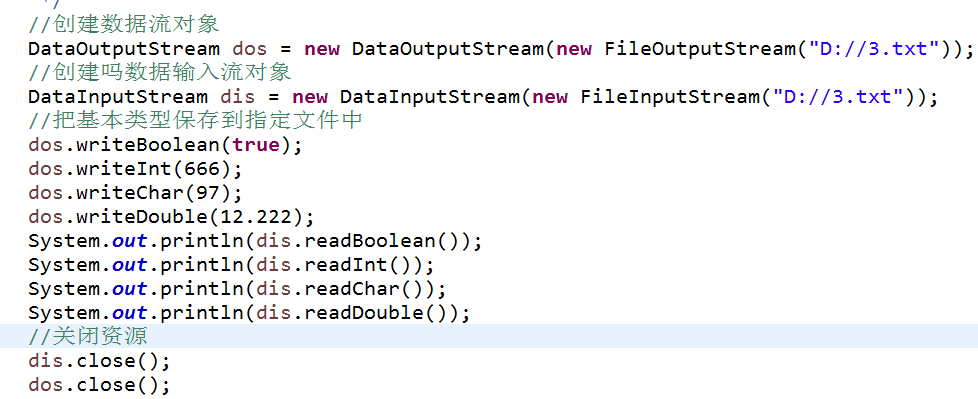
“ByteArrayOutputStream”类实现了一个输出流，其中的数据被写入一个 byte 数组。缓冲区

会随着数据的不断写入而自动增长。可使用 toByteArray() 和 toString() 获取数



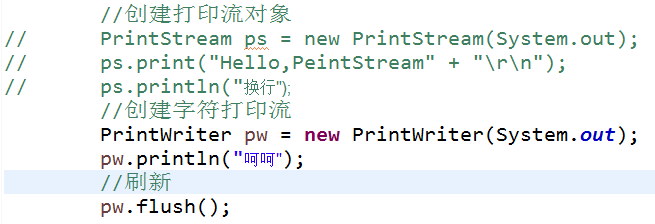
2.13、数据流(DataInputStream和DataOutputStream)

特点:该流实现了对象java中的八种基本数据类型进行读和写，在读和写是必须使用对应的方法才可以



2.14、打印流(PrintStream和PrintWriter）

特点:可以把任意数据类型的值进行打印输出



1. **线程**

3.1、什么是进程

进程是指在系统中正在运行的一个应用程序

每个进程之间是独立的，每个进程均运行在其专用且受保护的内存空间内

3.2、什么是线程

1个进程要想执行任务，必须得有线程（每1个进程至少要有1条线程)

线程是进程的基本执行单元，一个进程（程序）的所有任务都在线程中执行

3.3、多线程的概念

**1、什么是多线程**

1个进程中可以开启多条线程，每条线程可以并行（同时）执行不同的任务

例如:进程 ->车间，线程->车间工人

多线程技术可以提高程序的执行效率

**2、多线程原理**

同一时间，CPU只能处理1条线程，只有1条线程在工作（执行）多线程并发执行，其实是CPU

快速地在多条线程之间调度（切换）如果CPU调度线程的时间足够快，就造成了多线程并发执

行的假象思考：如果线程非常非常多，会发生什么情况？CPU会在N多线程之间调度，CPU会累

死，消耗大量的CPU资源每条线程被调度执行的频次会降低（线程的执行效率降低）

**3、多线程的优缺点**

3.1、多线程的优点：

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

3.2、多线程的缺点：

开启线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），如果

开启大量的线程，会占用大量的内存空间，降低程序的性能

线程越多，CPU在调度线程上的开销就越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

3.4、时间片的概念

即CPU分配给各个程序的时间，每个线程被分配一个时间段，称作它的[时间片](https://www.baidu.com/s?wd=%E6%97%B6%E9%97%B4%E7%89%87&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)。即该进程允许

运行的时间，使各个程序从表面上看是同时进行的

3.5、如何创建线程

5.1、继承Thread类

Class 类名 extends Thread{重写Thread中的run方法}

5.2、实现Runnable接口

5.3、实现Callable接口

3.6、线程的生命周期

**6.1、新建状态**

在Java语言中使用new 操作符创建一个线程后，该线程仅仅是一个空对象，它具备类线程的

一些特征，但此时系统没有为其分配资源，这时的线程处于创建状态。线程处于创建状态时，

可通过Thread类的方法来设置各种属性，如线程的优先级(setPriority)、线程名(setName)和

线程的类型(setDaemon)等

**6.2、就绪状态**

使用start()方法启动一个线程后，系统为该线程分配了除CPU外的所需资源，使该线程处于

就绪状态。此外，如果某个线程执行了yield()方法，那么该线程会被暂时剥夺CPU资源，重

新进入就绪状态。

**6.3、运行状态**

如果处于就绪状态的线程获得了CPU，开始执行run()方法的线程执行体，则该线程处于运行状态

**6.4、阻塞状态**

当处于运行状态的线程失去所占用CPU资源之后，便进入阻塞状态

阻塞状态(Blocked)：一个正在运行的线程因某些原因不能继续运行时，就进入阻

塞状态。这些原因包括：

1.等待阻塞：运行状态中的线程执行wait()方法，使本线程进入到等待阻塞状态；

2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会

进入同步阻塞状态；

3.其他阻塞 -- 通过调用线程的sleep()或join()或发出了I/O请求时，线程会进入到阻

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。

**6.5、死亡状态**

线程在run()方法执行结束后进入死亡状态。此外，如果线程执行了interrupt()或stop()方

法，那么它也会以异常退出的方式进入死亡状态。

3.7、线程常用方法

[start](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "start())() 使该线程开始执行；Java 虚拟机调用该线程的 run 方法。

[sleep](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "sleep(long))(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），此操作受到

系统计时器和调度程序精度和准确性的影响。

[currentThread](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "currentThread())() 返回对当前正在执行的线程对象的引用。

[getName](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "getName())()返回该线程的名称。

[join](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "join(long))(long millis) 等待该线程终止的时间最长为 millis 毫秒。

[setDaemon](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "setDaemon(boolean))(boolean on)将该线程标记为守护线程或用户线程。

[setPriority](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "setPriority(int))(int newPriority) 更改线程的优先级。

[getPriority](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "getPriority())() 返回线程的优先级。

[interrupt](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "interrupt())() 中断线程。

[yield](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "yield())() 暂停当前正在执行的线程对象，并执行其他线程。

[isAlive](mk:@MSITStore:C:\\Users\\58460\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/../../java/lang/Thread.html" \l "isAlive())() 测试线程是否处于活动状态。

3.8、同步代码块

synchronized(对象名){同步代码块....} //注意所对象必须是同一个对象

静态同步代码块

静态你同步代码块所对象为当前类

Synchronized(类名.class){静态同步代码块....}

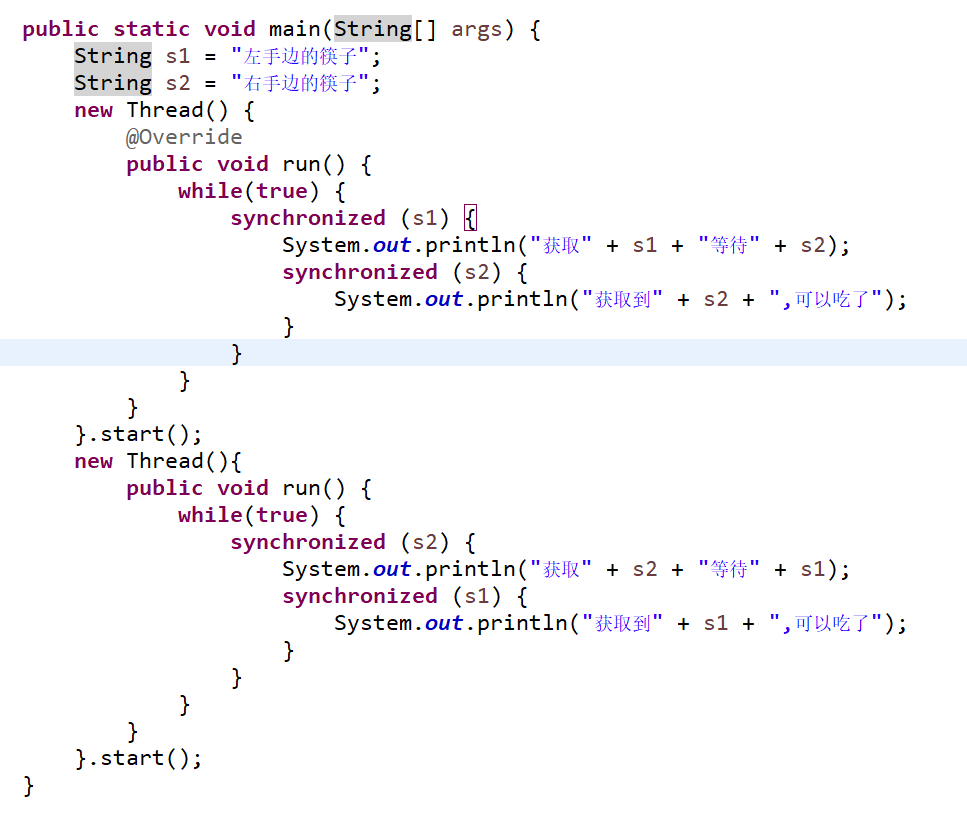
3.9、同步方法

访问修饰符 + synchronized + 返回值类型 + 方法名(){}

访问修饰符 + static + synchronized + 返回值类型 + 方法名(){}

3.10、死锁

即同步代码块嵌套同步代码块



练习:抢火车票

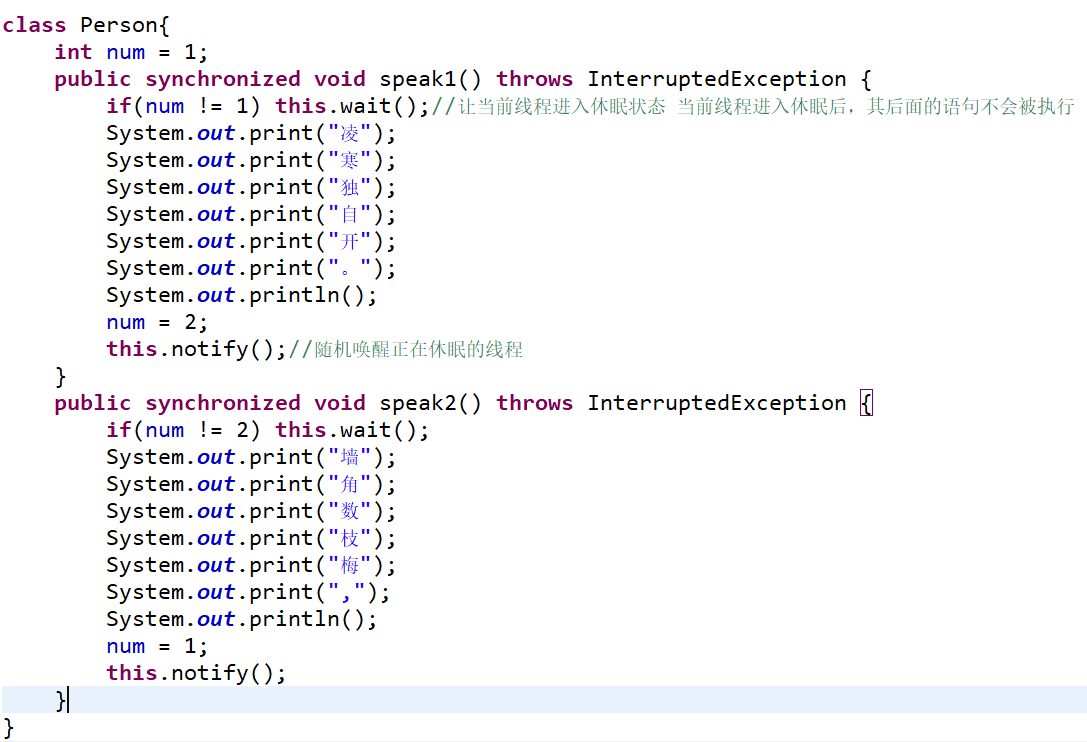
3.11、两条线程之间的通信(是两条线程有规律的执行)

wait():让当前线程进入休眠状态，只用使用notify()或者notifyAll()方法来唤醒

notify()：随机唤醒正在休眠状态下的线程。

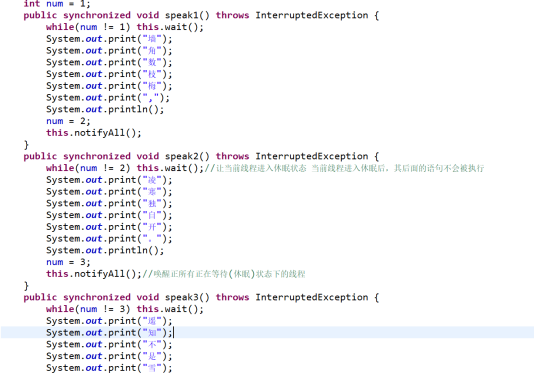
notifyAll():唤醒所有正在休眠状态下的线程。

如下代码显示:





3.12、三条线程之间的通信





3.13、lock锁

1. Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock

是一个类，通过这个类可以实现同步访问。

2、Lock和synchronized有一点非常大的不同，采用synchronized不需要用户去手动释

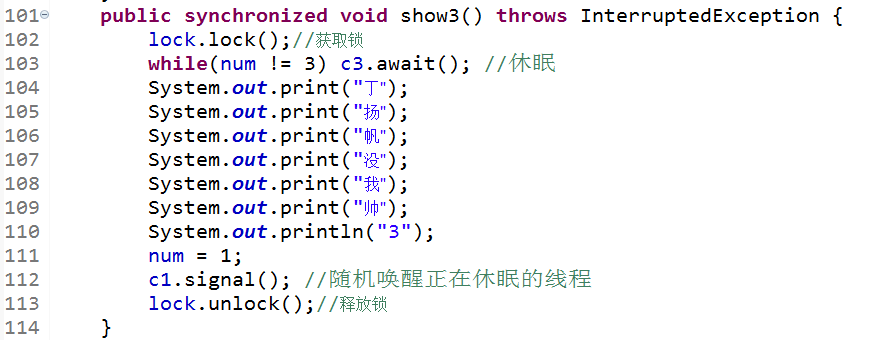
放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释

放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导

致出现死锁现象。

案例:





测试类:



3.14、计时器

在我们日常生活中，我们常常会遇到有关计时器的事情。例如京东商城会在某年某月某日某时

某分某秒进行特价活动，那么当时间到达这个时间点上的时候该事件就会触发。

在java中Timer是一种定时器工具，用来在一个后台线程计划执行指定任务。它可以计划执行

一个任务一次或反复多次。TimerTask一个抽象类，它的子类代表一个可以被Timer计划的任务。

案例:



3.15、线程池

系统需要处理非常多的请求、每个请求都需要开一个线程来操作、这时候系统需要无限制的去

创建和销毁、因为线程无法复用、线程的数量也是无限的大（创建线程和关闭线程上有一定的耗

时和耗能,系统性能下降）,从java5开始，java提供了自己的线程池。线程池中的线程在执行

完任务后，不会直接死亡，而是又返回到线程池中等待下一次被调用，当关闭线程池后线程池

中的线程才会死亡。java中使用Executors线程工厂类创建线程池对象，其方法如下:

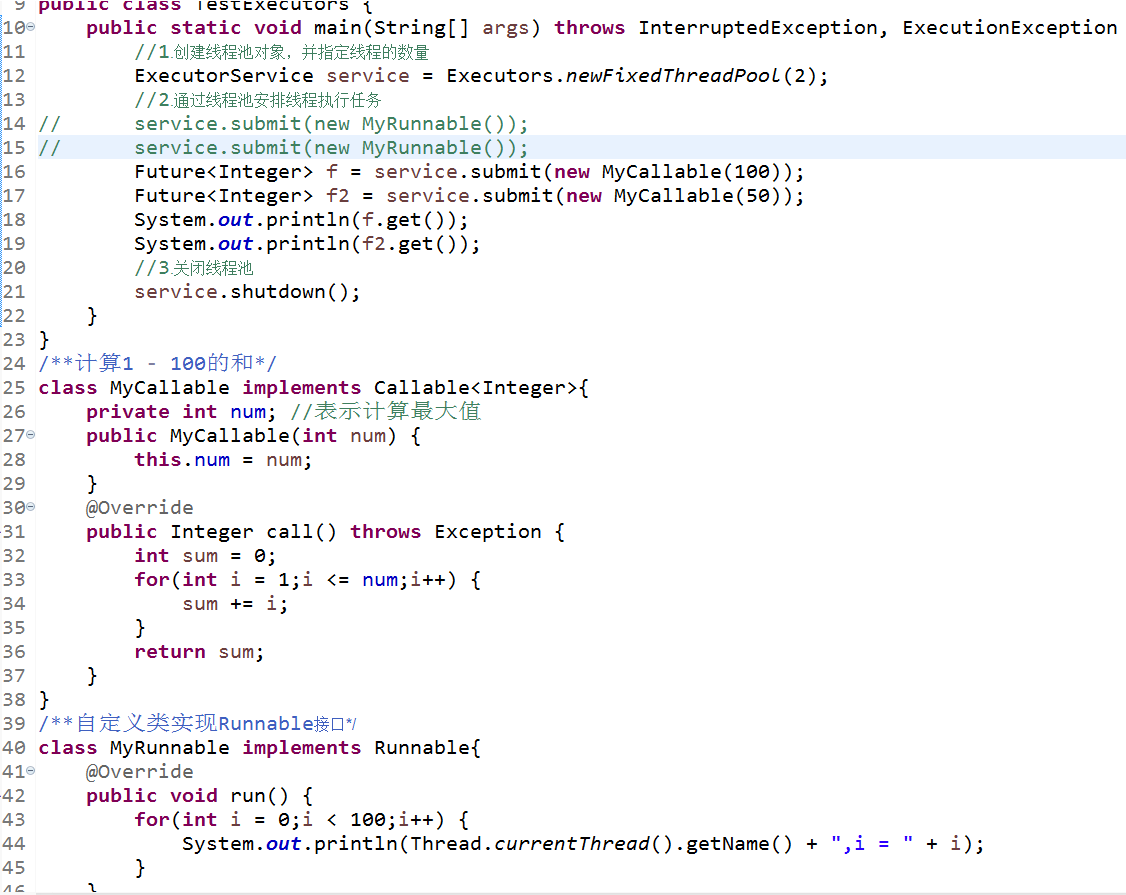
|  |  |
| --- | --- |
| static [ExecutorService](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ExecutorService.html" \o "java.util.concurrent 中的接口) | **[newSingleThreadExecutor](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Executors.html" \l "newSingleThreadExecutor())**()            创建只有一条线程的线程池对象 |
| static [ExecutorService](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ExecutorService.html" \o "java.util.concurrent 中的接口) | **[newSingleThreadExecutor](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Executors.html" \l "newSingleThreadExecutor(java.util.concurrent.ThreadFactory))**([ThreadFactory](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ThreadFactory.html" \o "java.util.concurrent 中的接口) threadFactory)        创建具有指定线程条数的线程对象 |

在java中ExecutorService类表示线程池,在线程池类中的submit方法可以直接执行Callable

和Runnable类型的任务，方法如下:

|  |  |
| --- | --- |
| <T> [Future](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Future.html" \o "java.util.concurrent 中的接口)<T> | **[submit](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ExecutorService.html" \l "submit(java.util.concurrent.Callable))**([Callable](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Callable.html" \o "java.util.concurrent 中的接口)<T> task)            执行一个Callable类型的任务 |
| [Future](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/Future.html" \o "java.util.concurrent 中的接口)<?> | **[submit](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ExecutorService.html" \l "submit(java.lang.Runnable))**([Runnable](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/lang/Runnable.html" \o "java.lang 中的接口) task)            执行一个Runnable类型的任务 |
| void | **[shutdown](mk:@MSITStore:C:\\Users\\Administrator\\Desktop\\JDK_API_1_6_zh_CN.CHM::/java/util/concurrent/ExecutorService.html" \l "shutdown())()** 关闭线程池 |

案列:



3.16、简单工厂设计模式

1、简单工厂设计模式:

简单工厂设计模式，又叫静态工厂设计模式。通过定义一个工厂类，这个类主要负责创建一些

指定类的实例

优点:客户端不需要再负责创建对象，明确了各个类的职责。

缺点:静态工厂类负责所有对象的创建，如果有新的对象增加，或者某些对象的创建方式不一样

的时候，需要不断的更改工厂类，这样不利用后期的维护

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

2、工厂方法模式:

把工厂类定义为接口，接口中只负责定义方法。而这个方法都是由其子类来实现这些方法

优点:客户端不需要再负责对象的创建，从而明确了各个类之间的职责。如果由新的类增加，只

需要增加一个具体类、和一个对应的工厂类就可以了。这样即不影响已有代码，后期维护容易，

从而增加了系统扩展性

缺点:需要额外编写代码(这一步是有必要的)

|  |
| --- |
|  |
|  |
|  |
|  |

四、网络编程

4.1、什么是计算机网络？

[计算机网络](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%BD%91%E7%BB%9C/18763" \t "_blank)是指将[地理](https://baike.baidu.com/item/%E5%9C%B0%E7%90%86" \t "_blank)位置不同的具有独立功能的多台[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA/140338" \t "_blank)及其外部设备，通过通信线路连

接起来，在[网络操作系统](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/3997" \t "_blank)，[网络管理软件](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E7%AE%A1%E7%90%86%E8%BD%AF%E4%BB%B6/6579078" \t "_blank)及[网络通信协议](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E9%80%9A%E4%BF%A1%E5%8D%8F%E8%AE%AE/4438611" \t "_blank)的管理和协调下，实现资源共享和信息传递的计算机系统.

网络又分：局域网、城域网、广域网

4.2、什么是网络编程?

网络编程就是使用网络套接字来达到通信目的的编程就叫网络编程

套接字:指源ip地址和目标ip地址以及源端口号和目标端口号组合

4.3、网络模型

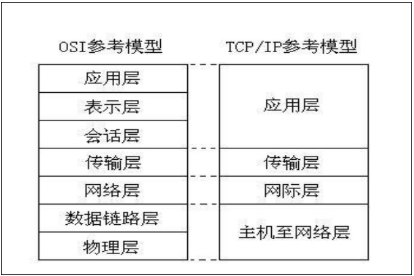
3.1计算机网络之间以何种规则进行通信，就是网络模型研究问题

3.2 网络模型一般是指

OSI（Open System Interconnection开放系统互连）参考模型

TCP/IP参考模型

3.3网络参考模型图



4.4、网络编程三要素

网络三要素分别是:ip地址、端口号、协议

**1.ip地址**

计算机的唯一标识

查看IP地址方式:在DOS命令窗口中输入 ipconfig

查看网络是否通畅: ping IP地址

ip地址分类：

A、B、C、D、E

A类: 从1.0.0.0 到127.0.0.0

B类: 从128.0.0.0到191.255.255.255

C类：从192.0.0.0到223.255.255.255

D类: 从224.0.0.0到239.255.255.255

E类: 从240.0.0.0到255.255.255.254

**2.端口号**

计算机程序的唯一标识

端口号取值范围(0 - 65535)

**3.协议:数据传输/交互规则**

UDP协议:

类似于发短信，只需要定义发送端和接收端

特点:效率高,数据传输不安全

TCP协议:

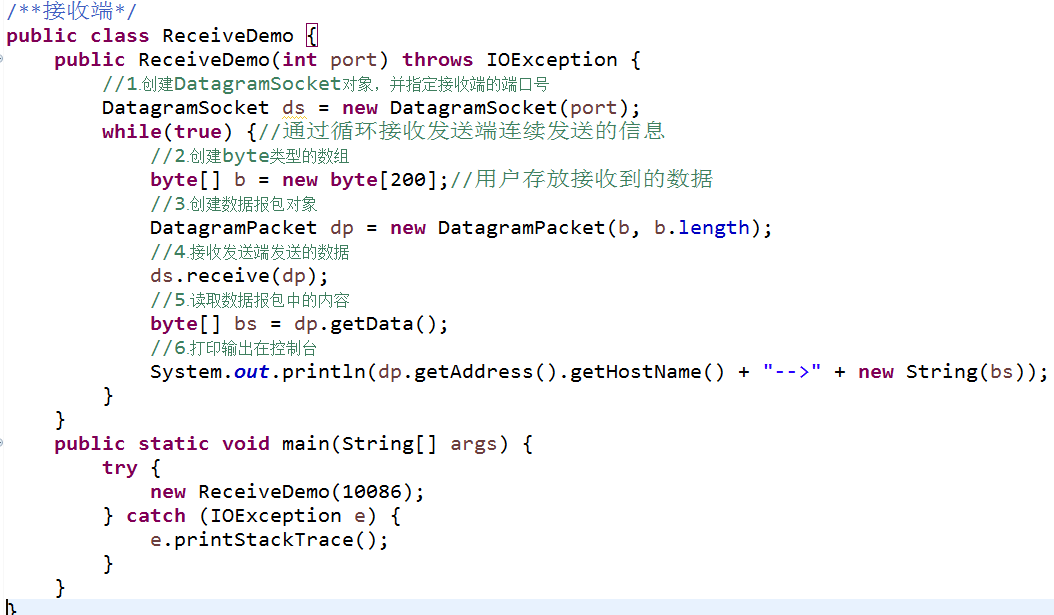
类似于QQ聊天,需要定义客户端和服务器端

特点:效率低，安全性高

4.5、使用不同协议进行编程

**1、UDP协议下编程:**

接收端:



发送端：

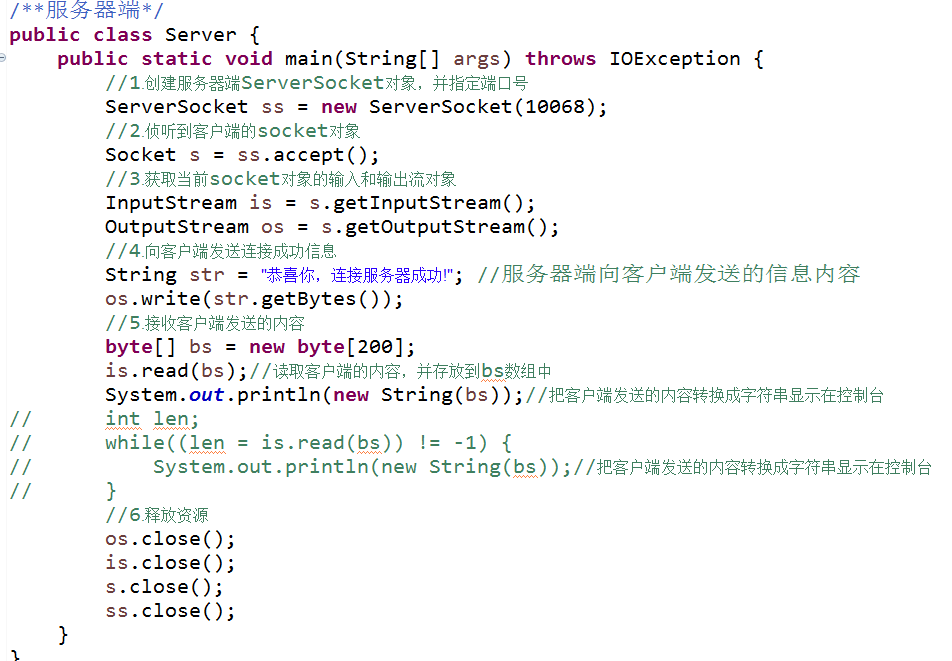


**2、TCP协议下编程:**

客户端:



服务器端:



**3、文件上传案例:**

|  |
| --- |
|  |
|  |
|  |

**4、群聊案例:**

|  |
| --- |
|  |
|  |
|  |

五、可变参数

Java1.5增加了新特性：可变参数：适用于参数个数不确定，类型确定的情况，java把可变参数当做

数组处理。

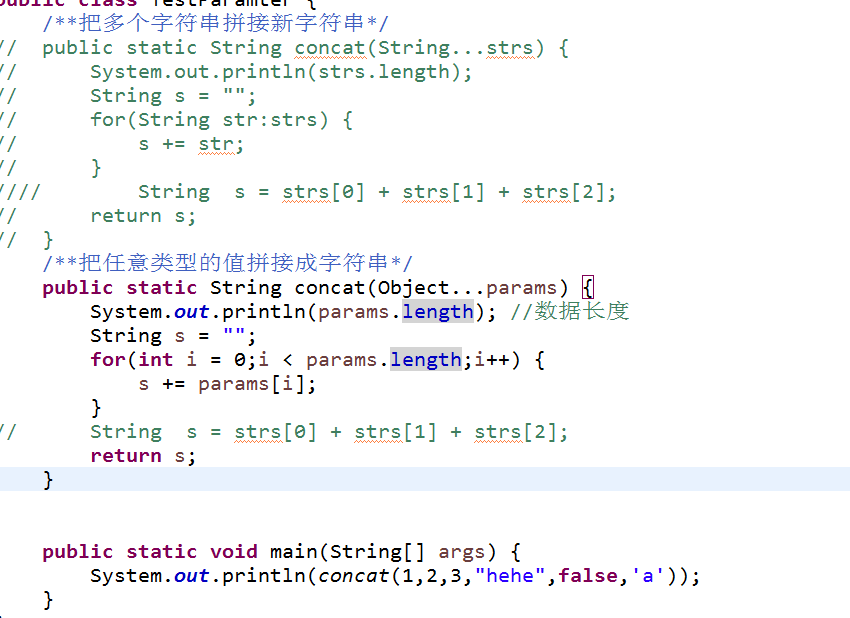
注意：可变参数必须位于最后一项。

因：当可变参数个数多余一个时，必将有一个不是最后一项，所以只支持有一个可变参数。因为参数

个数不定，所以当其后边还有相同类型参数时，java无法区分传入的参数属于前

一个可变参数还是后边的参数，所以只能让可变参数位于最后一项。

案例:



六、反射

6.1、什么是反射？

反射说的是在运行状态中，对于任何一个类，我们都能够知道这个类有哪些方法和属性。

对于任何一个对象，我们都能够对它的方法和属性进行调用。我们把这种动态获取对象信

息和调用对象方法的功能称之为反射机制

6.2、反射的作用(这里使用Person类来演示)

获取Person类的Class对象方式有三种,分别为一下方式：

1、通过类的class属性类获取

语法:类名.class

2、通过对象方法getClass()获取

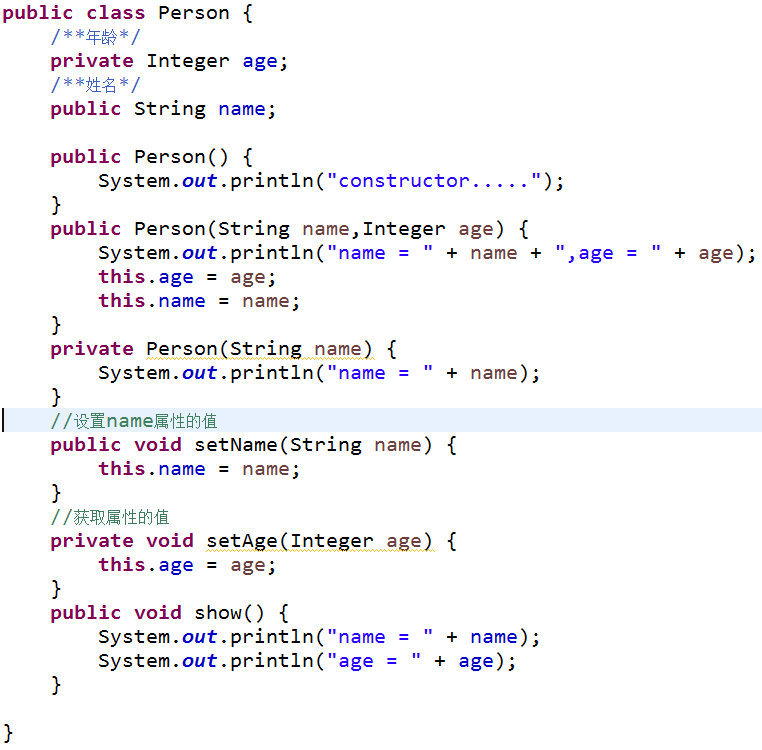
语法:对象名.getClass();

3、通过Class类中的方法forName()获取；

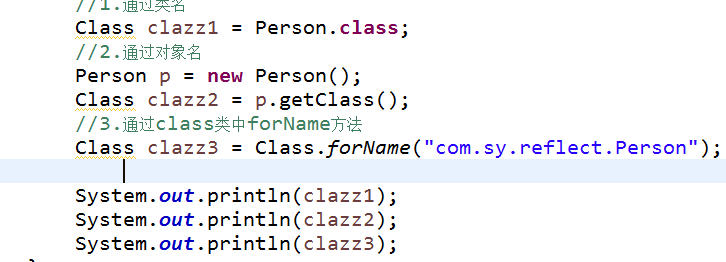
语法:Class.forName("类路径")

6.3、示例

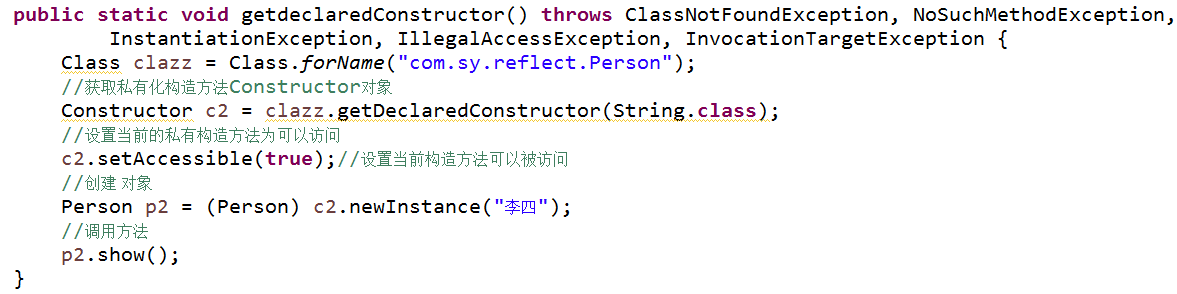
**先定义一个Person类**

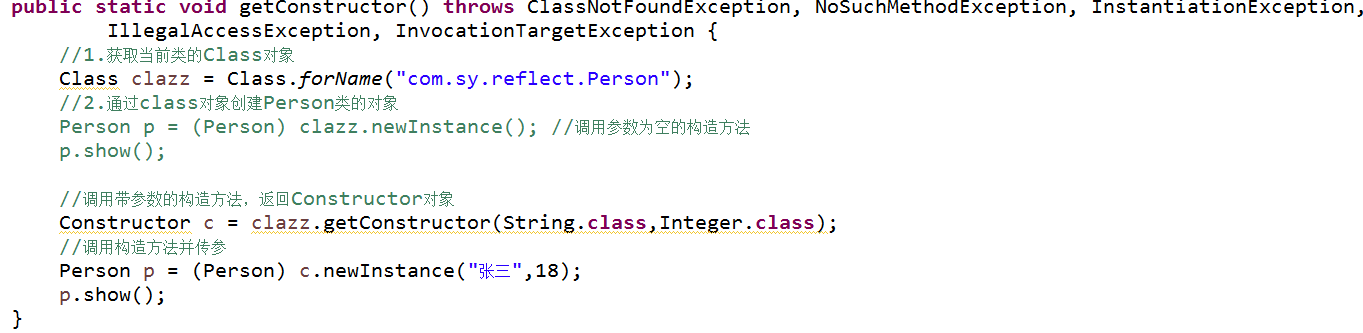


**测试类:**

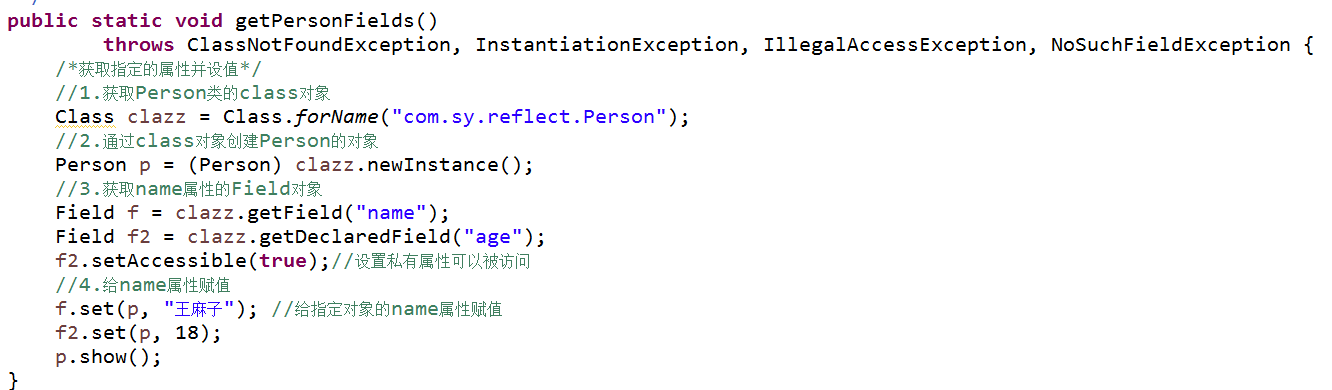


**通过Person类的Class对象创建Person类对象**

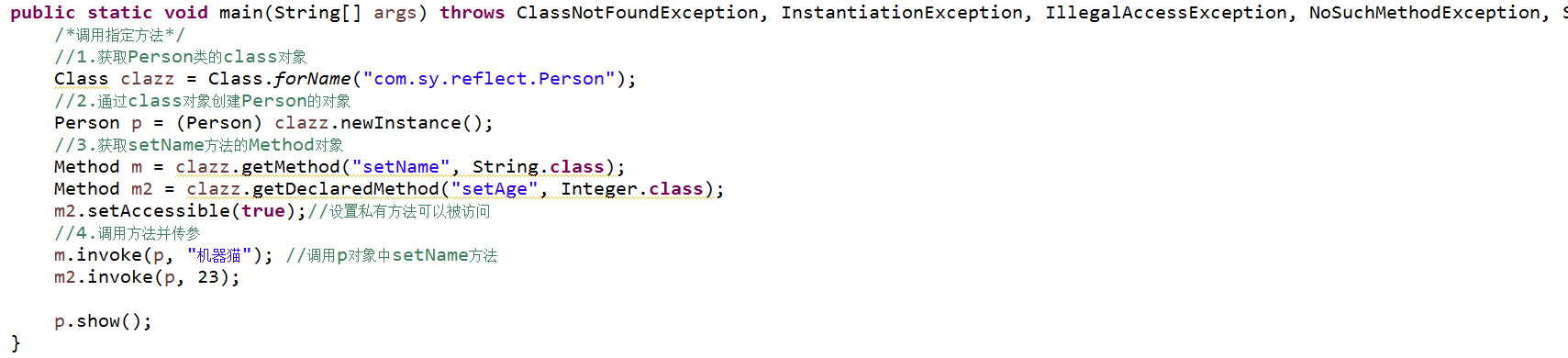




**通过反射给Person类中的属性设置**

、

**通过反射调用Person类中的方法**



七、动态代理

关于Java中的动态代理，我们首先需要了解的是一种常用的设计模式--代理模式，而对于

代理，根据创建代理类的时间点，又可以分为静态代理和动态代理。

7.1、代理模式

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负

责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与

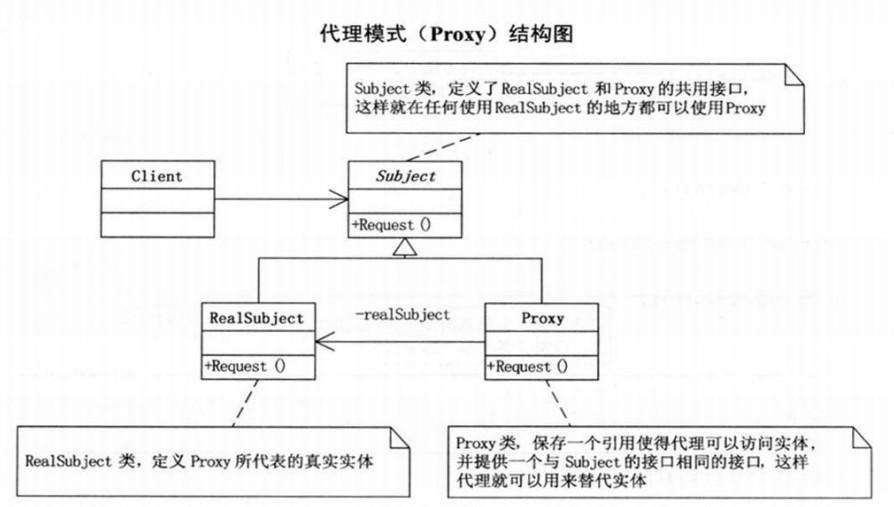
委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对

象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。简

单的说就是，我们在访问实际对象时，是通过代理对象来访问的，代理模式就是在访问实际对

象时引入一定程度的间接性，因为这种间接性，可以附加多种用途。在后面我会解释这种间接

性带来的好处。代理模式结构图:



7.2、静态代理

静态代理：由程序员创建或特定工具自动生成源代码，也就是在编译时就已经将接口，被代

理类，代理类等确定下来。在程序运行之前，代理类的.class文件就已经生成。

1、静态代理实现

 根据上面代理模式的类图，来写一个简单的静态代理的例子。假如一个班的同学要向老师交班

费，但是都是通过班长把自己的钱转交给老师。这里，班长就是代理学生上交班费，班长就是

学生的代理。

* 1. 首先，我们创建一个Person接口。这个接口就是学生（被代理类），和班长（代理类）的

公共接口，他们都有上交班费的行为。这样，学生上交班费就可以让班长来代理执行。

|  |
| --- |
| */\*\*  \* 静态代理模式  \** ***@author*** *ykj  \*/* public interface Person {  */\*\*上交班费\*/* void giveMoney(); } |

1.2、定义Student类并实现Person接口，在Student类中实现giveMoney()方法

|  |
| --- |
| */\*\*  \* 学生类  \** ***@author*** *ykj  \*/* public class Student implements Person{  */\*\*学生姓名\*/* private String name;   public Student(String name) {  this.name = name;  }  */\*\*实现上交班费方法\*/* @Override  public void giveMoney() {  System.*out*.println(name + "交了100元RMB");  } } |

1.3、StudentsProxy类，这个类也实现了Person接口，但是还另外持有一个学生类对象，由于实

现了Peson接口，同时持有一个学生对象，那么他可以代理学生类对象执行上交班费（执行giveMoney()

方法）行为。

|  |
| --- |
| */\*\*  \* Student代理类,实现了Person接口  \** ***@author*** *ykj  \*/* public class StudentProxy implements Person{  */\*\*持有被代理类对象的引用\*/* private Student student;   public StudentProxy(Student student) {  this.student = student;  }   */\*\*实现和被代理类相同的方法\*/* @Override  public void giveMoney() {  student.giveMoney();  } } |

1.4、创建测试类，测试代理类的使用

|  |
| --- |
| public class Test {  public static void main(String[] args) {  */\*\*被代理的学生小杨，他的班费上交有代理对象班长完成\*/* Student s = new Student("小杨");  */\*\*生成代理对象，并将小杨传给代理对象\*/* Person p = new StudentProxy(s);  */\*\*班长代理上交班费\*/* p.giveMoney();  } |

总结:这里并没有直接通过张三（被代理对象）来执行上交班费的行为，而是通过班长（代理对象）

来代理执行了。这就是代理模式。

2、动态代理

java的动态代理是运行时动态的根据需要被代理的接口列表interfaces生成一个代理类，该代理类

实现了接口列表interfaces中的所有方法，然后在方法的内部实际是讲该方法的调用转发给了实现了

InvocationHandler接口的对象，顾名思义，该对象让包含代理时被代理方法的代理逻辑。

2.1、动态代理实现

在java的java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口，通过这个类

和这个接口可以生成JDK动态代理类和动态代理对象。

其中，Proxy类中提供一个newProxyInstance()静态方法，通过该方法可以快速创建动态代理对象。

修改测试类，通过JDK动态代理创建代理对象:

|  |
| --- |
| public class Test {  public static void main(String[] args) {  */\*\*被代理的学生小杨，他的班费上交有代理对象班长完成\*/* Student s = new Student("小杨"); */\*\*  \* 通过Proxy类中的newProxyInstance方法动态创建指定一组接口的代理对象  \* 参数1:用来指明生成代理对象使用哪个类装载器  \* 参数2:用来指明生成哪个对象的代理对象，通过接口指定  \* 参数3:表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上  \* \*/* Person p = (Person) Proxy.*newProxyInstance*(Person.class.getClassLoader(),new Class[]{Person.class},new InvocationHandler(){  */\*\*  \*  \** ***@param*** *o 表示代理对象  \** ***@param*** *method 表示被调用的方法对象  \** ***@param*** *objects 表示调用方法的实际参数  \** ***@return*** *执行方法后返回值  \*/* @Override  public Object invoke(final Object o, final Method method, final Object[] objects) throws Throwable {  Object obj = method.invoke(s);  System.*out*.println(method.getName());  return obj;  }  });  p.giveMoney();  } } |

八、Java注解

**8.1、注解概述**

在Java中，注解(Annotation)引入始于Java5，用来描述Java代码的元信息，通常

情况下注解不会直接影响代码的执行，-有些注解可以用来做到影响代码执行

****@Value 作用于成员变量 @Autowired @Resource等等****

****作用类上注解 @Repository @Service @Controller @Configuration等等****

****作用方法上的 @Override @Bean @RequestMapping等等****

****作用于参数的注解: @Parma @RequestParam等等****

****注意:注解本身没有实际意义，需要和java反射连用才会有直接效果****

**8.2、注解作用**

1. 生成文档。这是最常见的，也是java 最早提供的注解。常用的有@see @param

@return 等

1. 跟踪代码依赖性，实现替代配置文件功能。比较常见的是spring 2.5 开始的基于

注解配置。作用就是减少配置。现在的框架基本都 使用了这种配置来减少配置文

件的数量

3、在编译时进行格式检查。如@override 放在方法前，如果你这个方法并不是覆盖

了超类方法，则编译时就能检查出

**8.3、Java内置注解**

@Deprecated、@Override、@SuppressWarnings

**1、@Deprecated**

这个注释是一个标记注释。所谓标记注释，就是在源程序中加入这个标记后，并不影

响程序的编译，但有时编译器会显示一些警告信息； 那么Deprecated注释是什么意

思呢？如果你经常使用eclipse等IDE编写java程序时，可能会经常在属性或方法

提示中看到这个词。如果某个类成员的提示中出现了个词，就表示这个并不建议使用

这个类成员。因为这个类成员在未来的JDK版本中可能被删除。之所以在现在还保留，

是因为给那些已经使用了这些类成员的程序一个缓冲期。如果现在就去了，那么这些

程序就无法在新的编译器中编译了

可以用来标记类，方法，属性。如果上述三种元素不再使用，使用@Deprecated注解

如果代码使用了@Deprecated注解的类，方法或属性，编译器会进行警告

**2、@Override**

这个注释的作用是标识某一个方法是否覆盖了它的父类的方法。

### **3、@SuppressWarnings**

当我们的一个方法调用了弃用的方法或者进行不安全的类型转换，编译器会生成

警告。我们可以为这个方法增加@SuppressWarnings注解，来抑制编译器生成警告

@SuppressWarnings用来抑制编译器生成警告信息。可以修饰的元素为类，方法，

方法参数，属性，局部变量

注意：使用@SuppressWarnings注解，采用就近原则，比如一个方法出现警告，

我们尽量使用@SuppressWarnings注解这个方法，而不是注解方法所在的类。虽

然两个都能抑制编译器生成警告，但是范围越小越好，因为范围大了，不利于我

们发现该类下其他方法的警告信息。

**8.4、Java元注解**

|  |  |
| --- | --- |
| **注解类型** | **含义** |
| Documented | 表示含有该注解类型的元素(带有注释的)会通过javadoc或类似工具进行文档化 |
| Inherited | 表示注解类型能被自动继承 |
| Retention | 表示注解类型的存活时长 |
| Target | 表示注解类型所适用的程序元素的种类 |

**1、Documented**

@Documented：表示拥有该注解的元素可通过javadoc此类的工具进行文档化。

该类型应用于注解那些影响客户使用带注释(comment)的元素声明的类型。如

果类型声明是用Documented来注解的，这种类型的注解被作为被标注的程序

成员的公共API

**2、Inherited**

@Inherited：表示该注解类型被自动继承，如果用户在当前类中查询这个元注

解类型并且当前类的声明中不包含这个元注解类型，那么也将自动查询当前类

的父类是否存在Inherited元注解，这个动作将被重复执行知道这个标注类型

被找到，或者是查询到顶层的父类

**3、Retention**

@Retention：表示该注解类型的注解保留的时长。当注解类型声明中没有@Retention

元注解，则默认保留策略为RetentionPolicy.CLASS。关于保留策略(RetentionPolicy)

是枚举类型，共定义3种保留方式

|  |  |
| --- | --- |
| **name** | **exp** |
| SOURCE | 仅存在Java源文件，经过编译器后便丢弃相应的注解 |
| CLASS | 存在Java源文件，以及经编译器后生成的Class字节码文件，但在运行时VM不再保留注释 |
| RUNTIME | 存在源文件、编译生成的Class字节码文件，以及保留在运行时VM中，可通过反射性地读取注解 |

### **4、Target**

表示该注解类型的所使用的程序元素类型。当注解类型声明中没有@Target元注

解，则默认为可适用所有的程序元素。如果存在指定的@Target元注解，则编译

器强制实施相应的使用限制。关于程序元素(ElementType)是枚举类型，共定义

8种程序元素

|  |  |
| --- | --- |
| **ElementType** | **exp** |
| TYPE | 类、接口（包括注解类型）或枚举声明 |
| CONSTRUCTOR | 构造方法声明 |
| PACKAGE | 包声明 |
| LOCAL\_VARIABLE | 局部变量声明 |
| METHOD | 方法声明 |
| ANNOTATION\_TYPE | 注解类型声明 |
| PARAMETER | 参数声明 |
| FIELD | 字段声明（包括枚举常量） |

**8.5、自定义注解**

如@Ety，其中@的意思是告诉编译器这是一个注解。而Ety则是注解的名字。定义

一个注解我们需要@interface，使用通常在文件中，代码如下：

|  |
| --- |
| @Documented  @Inherited  @Retention(RetentionPolicy.***RUNTIME***)  @Target(ElementType.***FIELD***)  **public** **@interface** Ety {  //表示给实体对象设置值  String value();  //表示元素名称  String name();  } |

九、Java8新特性

9.1、接口中默认方法

**1、传统的方法(之前的知识)**

在java8之前java中接口里面的方法默认都是public abstract修饰的抽象方法并且没有

方法体。

**2、static方法**

1.使用static修饰接口中的方法并且必须有方法体(方法主体)。

2.接口中的static方法只能被接口本身调用。

语法:接口名.方法名();

3.接口中的static方法不能被子接口类继承

4.接口中的static方法不能被实现类覆盖及直接调用

**3、default方法**

1.在接口中被default修饰的方法必须要有方法体

2.接口中的default方法不能直接被自己调用，而是由实现类调用

语法:实现类对象名.方法名();

3.接口中的default方法可以被子接口继承

4.接口中的default方法可以被实现类重写和调用

9.2、函数式接口

**1、什么是函数式接口**

函数式接口(Functional Interface)就是一个有且仅有一个抽象方法,但是可以有多个非

抽象方法的接口,函数式接口可以被隐式转换成Lambda表达式。

**函数式接口API**

JDK1.8之前已有的函数式接口:

java.lang.Runnable

java.util.concurrent.Callable

java.util.Comparator

java.io.FileFilter

java.lang.reflect.InvocationHandler

java.awt.event.ActionListener

**2、JDK1.8新增加的函数式接口**

java.util.Function 此包中包含的很多类 ，用来支持java的函数式编程

**3、函数式接口注解**

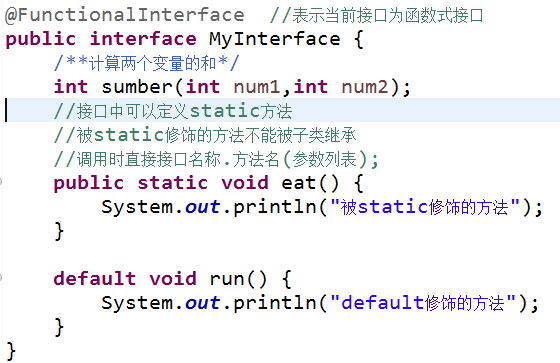
@FunctionalInterface我们在函数式接口上面加上此注解后，接口里面就只能够有一个抽

象方法了,当然不加此注解且只有一个抽象方法的接口也是函数式接口,只是没有限定提示

而已。

9.3、函数式接口示例

**1、定义函数式接口**



**2、Lambda表达式**

**2.1、什么是lambda表达式**

简答来说:可以看成是对匿名内部类的简写，使用Lambda表达式时，接口必须时函数式接口

2.2、Lambda表达式的语法

**基本语法:**

**函数式接口 变量名 = (参数1,参数2,....) -> {**

**//方法体**

**}**

特点说明:

(参数1，参数2,...)表示参数列表

->表示连接符

{}内部时方法体

1.右边的类型会根据左边的函数式接口类型自动推断;

2.如果形参列表为空，只需保留();

3.如果形参只有一个,()可以省略，只需要参数的名称即可;

4.如果执行的语句只有一句，且无返回值,花括号可以省略,若有返回值，则若想省去{},则

必须同时省略return,且执行语句也保证只有一句。

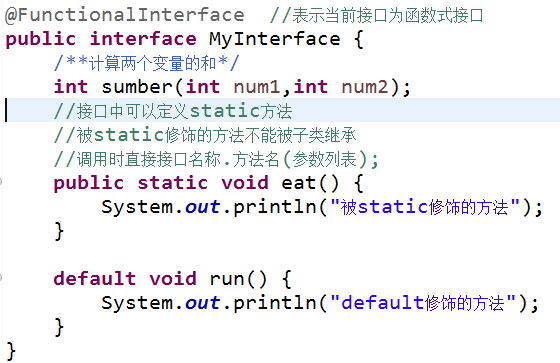
5.形参列表的数据类型会自动推断

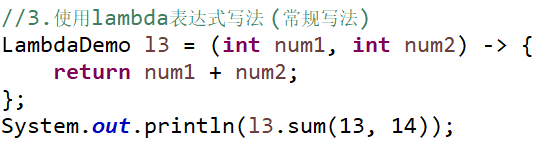
6.lambda不会生成一个单独的内部类文件。

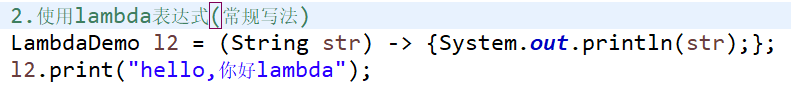
7.lambda表达式若访问了局部变量，则局部变量必须是final的，若是局部变量没有加final

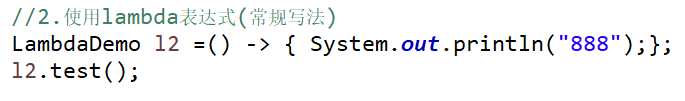
关键字，系统会自动添加，此后在修改该局部变量，会报错。

2.3、案例1:





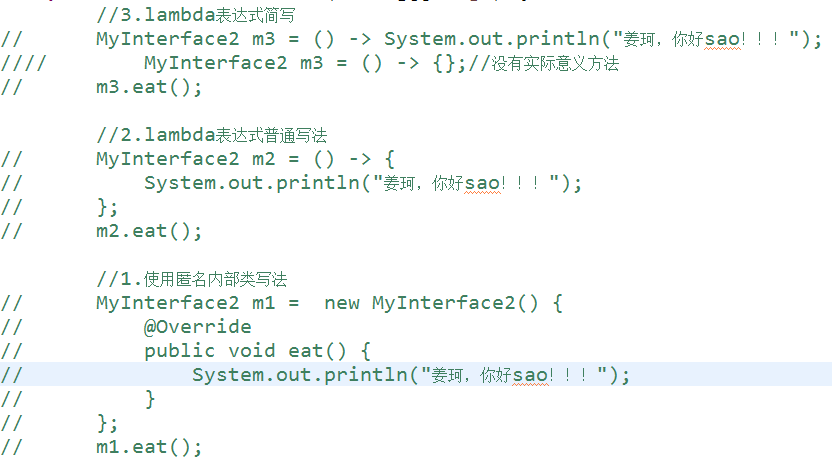




、

2.4、案例2



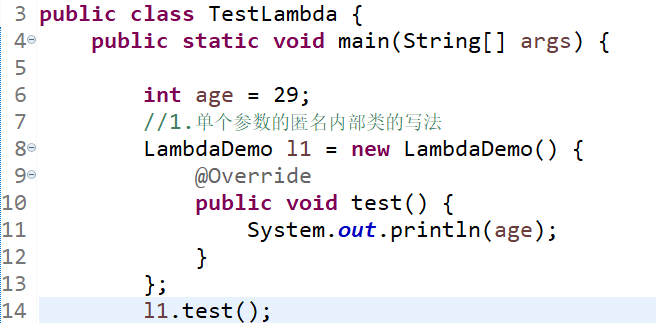


**3、Lambda的作用域**

在Lambda表达式中访问外层作用域和老版本的匿名对象中的方式很相似，你可以 直接 方法

标记了final的外层局部变量，或者实例的字段以及静态变量，但如果访问局部 变量，要

求局部变量必须是final修饰的



注意:上面代码中一旦在匿名的内部类中使用age,则第6行的age会自动被编辑为final的

如果在lambda表达式中也是一样的

**4、方法的引用**

**概念:**方法的引用是对lambda表达式的优化，当lambda表达式重写的抽象方法，已经有现成

的方法可以使用时，此时，lambda表达式就显得冗余了，方法引用的优势是省去了参数的传递。

**方法引用的格式为:**

引用类型 :: 方法名

**方法引用的分类:**

静态方法的引用：

格式: 类名::静态方法名

实例方法的引用：

格式: 类名::静态方法名

对象方法的引用：

格式: 对象名::静态方法名

构造方法的引用：

格式: 类名::new

**4.1、静态方法引用的案例**

|  |
| --- |
| */\*\*  \* 静态方法引用演示  \*/* public class MethodRef01 {  public static void main(String[] args) {  //把一个纯数字类型的字符串转换成int类型  //使用函数型接口Function  //1.匿名内部类实现方式  Function<String,Integer> f1 = new Function<String, Integer>() {  @Override  public Integer apply(String s) {  return Integer.*parseInt*(s);  }  };  //调用方法 // System.out.println(f1.apply("520") + 1);   //2.使用lambda表达式优化  Function<String,Integer> f2 = s -> Integer.*parseInt*(s);  //调用方法 // System.out.println(f2.apply("520") + 2);   //3.使用方法引用类优化lambda表达式  Function<String,Integer> f3 = Integer :: *parseInt*;  //调用方法  System.*out*.println(f2.apply("520") + 3);  } } |

**4.2、实例方法引用案例**

|  |
| --- |
| */\*\*  \* 实例方法的引用案例  \*/* public class MethodRef02 {  public static void main(String[] args) {  //把一个字符串中的所有小写字母转换成大写  //使用函数型接口实现  //1.匿名内部类实现方式  Function<String,String> f1 = new Function<String, String>() {  @Override  public String apply(String s) {  return s.toUpperCase();  }  };  //调用方法 // System.out.println(f1.apply("abcdef"));   //2.使用lambda表达式优化匿名内部类方式  Function<String,String> f2 = s -> s.toUpperCase();  //调用方法 // System.out.println(f2.apply("abcdef"));   //3.使用实例方法的引用优化lambda表达式  Function<String,String> f3 = String :: toUpperCase;  System.*out*.println(f3.apply("abcdef"));  } } |

**4.3、对象方法引用案例**

PrintMessage类:

|  |
| --- |
| */\*\*  \* 打印信息类  \*/* public class PrintMessage {  */\*\*  \* 把方法中的参数直接输出在控制台  \*/* public void printMsg(Object value){  System.*out*.println(value);  } } |

测试类

|  |
| --- |
| */\*\*  \* 对象方法引用演示  \*/* public class MethodRef03 {  public static void main(String[] args) {  //演示控制台直接输出参数  //使用Consumer接口   //1.匿名内部类实现方式  //创建PrintMessage类对象  PrintMessage pm = new PrintMessage();  Consumer<String> c1 = new Consumer<String>() {  @Override  public void accept(String s) {  //调用PrintMessage中的对象方法  pm.printMsg(s);  }  };  //调用方法 // c1.accept("aaa");   //2.使用lambda表达式优化匿名内部类  Consumer<String> c2 = s -> pm.printMsg(s);  //调用方法 // c2.accept("bbb");   //3.使用对象方法引用优化lambda表达式  Consumer<String> c3 = pm :: printMsg;  //调用方法  c3.accept("ccc");  } } |

**4.4、构造方法的引用**

实现步骤:

1.创建一个实体类(比如:Student类)

2.编写测试类演示构造方法引用

|  |
| --- |
| */\*\*  \* 构造方法的引用演示  \*/* public class MethodRef04 {  public static void main(String[] args) {  //创建Student类对象  //使用供给型接口实现无参构造引用或者使用函数型接口实现带参的构造引用  //1.使用匿名内部类实现  Supplier<Student> s1 = new Supplier<Student>() {  @Override  public Student get() {  return new Student();  }  };  //调用方法 // System.out.println(s1.get());   //2.使用lambda优化匿名内部类  Supplier<Student> s2 = () -> new Student(); // System.out.println(s2);   //3.使用构造方法引用优化lambda表达式  Supplier<Student> s3 = Student :: new;  System.*out*.println(s3);  } } |

9.4、Stream API

Java8中有两大最为重要的改变。第一个是lambda表达式，另一个则是StreamAPI.

StreamAPI(java.util.stream)把真正的函数式编程风格引入到java中,这时目前为止对java类库的最

好补充，因为Stream API可以极大提供Java程序员的生产力，让程序员写出更高效、干净、简洁的代

码。

Stream是Java8中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂

的查询、过滤和映射数据等操作，使用StreamAPI 对集合数据进行操作，就类似于使用SQL执行的数据

库查询。也可以使用StreamAPI 来并行执行操作。简言之，StreamAPI 提供了一种高效且易于使用的处

理数据的方式。

**1、为什么要使用Stream API**

实际开发中，项目中多数数据都来自于Mysql、Oracle等，但现在数据源可以更多了，有MongDB、Redis

等，而这些NoSQL的数据就需要Java层面去处理。

Stream和Collection集合的区别：Collection是一种静态的内存数据结构，而Stream是有关计算的。

前者是主要面向内存，存储在内存中，后者主要是面向CPU,通过CPU实现计算。

**2、什么是Stream**

Stream到底是什么呢？是数据渠道，用于操作数据源(集合、数组等)所生成的元素序列。

集合讲的是数据，Stream讲的是计算!

注意:

1.Stream 自己不会存储元素

2.Stream不会改变源对象。相反，他们会返回一个持有结果的新Stream.

3.Stream操作是延迟执行的。这意味着他们会等到需要结果的 时候才执行。

**3、Stream的三个操作步骤**

**3.1、创建Stream**

一个数据源(如:集合、数组),获取一个流

**3.2、中间操作**

一个中间操作链，对数据源的数据进行处理（每次操作完毕返回值都是一个新Stream对象）

可以支持链式操作

**3.3、终止操作(终端操作)**

一旦执行终止操作，就执行中间操作链，并产生结果。之后，不会在被使用。

终止操作

......

Map

Filter

数据源

一系列“中间操作”形成的流水线

**4、Stream的实例化**

1、创建Stream方式一:通过集合

Java8中的Collection接口被扩展，提供了两个获取流的方法:

default Strean<E> stream():返回一个顺序流

default Stream<E> parallelStream():返回一个并行流

2、创建Stream方式二: 通过数组

Java8中的Arrays的静态方法stream()可以获取数组流:

static <T> Stream<T> stream(T[] array)：返回一个流

重载形式,能够处理对应基本类型的数组:

public static IntStream stream(int[] array)

public static LongStream stream(long[] array)

public static DoubleStream stream(double[] array)

3、创建Stream方式三: 通过Stream的of()

可以调用Stream类静态方法of(),通过显示值创建一个流，它可以接收任意数量的参数。

static <T> Stream<T> of(T...values):返回一个流

4、创建Stream方式三: 创建无限流

可以使用静态方法Stream.iterate()和Stream.generate(),创建无限流。

迭代

static <T> Stream<T> iterate(final T seed,final UnaryOperator<T> f)

生成

static <T> Stream<T> generate(Supplier<T> s)

**5、Stream的中间操作**

多个中间操作可以连接起来形成一个流水线，除非流水线上触发终止操作，否则中间操作不会执行

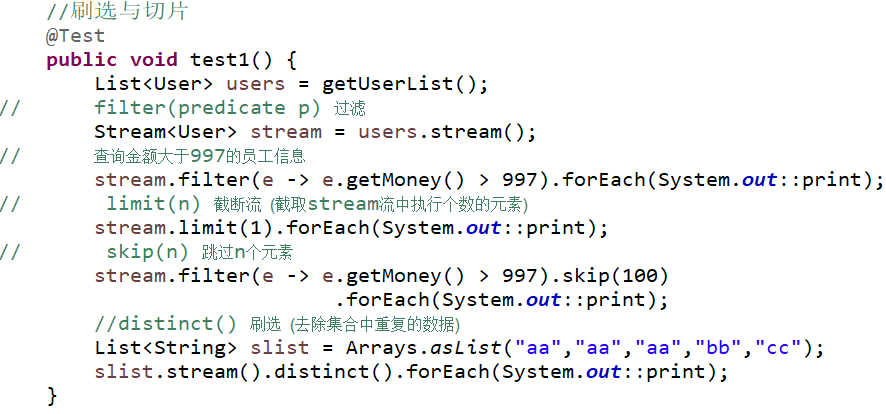
任何的处理！而在终止操作室一次性全部处理，称为“惰性求值”。

**5.1、筛选与切片**

|  |  |
| --- | --- |
| **方 法** | **描 述** |
| filter(Predicate<? super T> p) | 接收Lambda，从流中排出某些元素 |
| distinct() | 刷选，通过流所生成元素的hashCode()和equals()去除重复元素 |
| limit(long maxSize) | 截断流，使其元素不超过给定数量 |
| skip(long n) | 跳过元素，返回一个扔掉了前n个元素的流。若流中元素不足n个，则返回一个空流，与limit(n)互补 |



案例:



**5.2、映射**

|  |  |
| --- | --- |
| **方 法** | **描 述** |
| map(Function f) | 接收一个函数作为参数，该函数会被应用到每一个元素上，并将其映射成一个新的元素。 |
| mapToDouble(ToDoubleFunction f) | 接收一个函数作为参数，该函数会被应用到每一个元素上，产生一个新的DoubleStream. |
| mapToInt(ToIntFunction f) | 接收一个函数作为参数，该函数会被应用到每一个元素上，产生一个新的IntStream. |
| mapToLong(ToLongFunction f) | 接收一个函数作为参数，该函数会被应用到每一个元素上，产生一个新的LongStream. |
| flatMap(Function f) | 接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流 |

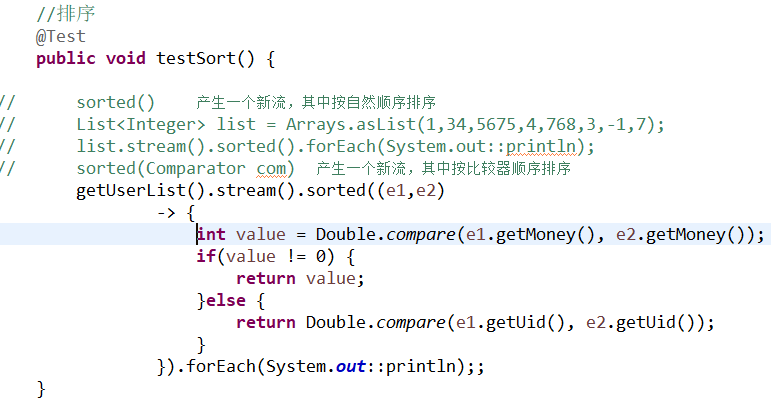
案例:



**5.3、排序**

|  |  |
| --- | --- |
| **方 法** | **描 述** |
| sorted() | 产生一个新流，其中按自然顺序排序 |
| sorted(Comparator com) | 产生一个新流，其中按比较器顺序排序 |

案例:



**6、Stream的终止操作**

终止操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List、Integer，甚至

是void

流进行了终止操作后，不能再次使用。

**6.1、匹配与查找**

|  |  |
| --- | --- |
| **方 法** | **描 述** |
| allMatch(Predicate p) | 检查是否匹配所有元素 |
| anyMatch(Predicate p) | 检查是否至少匹配一个元素 |
| noneMatch(Predicate p) | 检查是否没有匹配所有元素 |
| findFirst() | 返回第一个元素 |
| findAny() | 返回当前流中的任意元素 |
| count() | 返回流中元素的总个数 |
| max(Comparator c) | 返回流中最大值 练习:返回金额最低的用户 |
| min(Comparator c) | 返回流中最小值 练习:返回金额最低的用户 |
| forEach(Consumer c) | 内部迭代 |

案例:

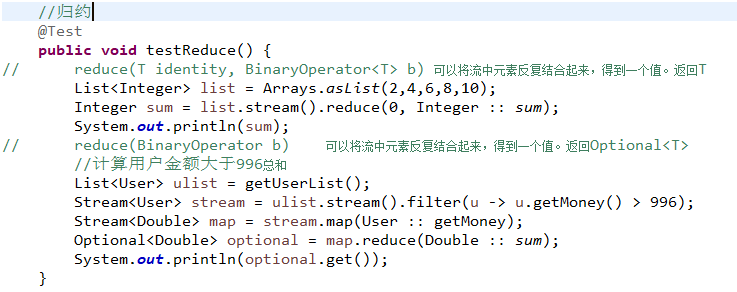


**6.2、归约**

|  |  |
| --- | --- |
| **方 法** | **描 述** |
| reduce(T identity, BinaryOperator<T> b) | 可以将流中元素反复结合起来，得到一个值。返回T |
| reduce(BinaryOperator b) | 可以将流中元素反复结合起来，得到一个值。返回Optional<T> |

备注:map和reduce的连接通常称为map-reduce模式，因Google用它来进行网络搜索而出名

案例:



**6.3、收集**

|  |  |
| --- | --- |
| **方 法** | **描 述** |
| collect(Collector c) | 将流转换为其它形式。接收一个Collector接口的实现，用于给Stream中元素做汇总的方法 |

Collector接口中方法的实现决定了如何对流执行收集的操作(如收集到List、Set、Map).另外，

Collectors实用类提供了很多静态方法，可以方便的创建集合常见收集器实例。

具体方法与实例如下表:

|  |  |  |
| --- | --- | --- |
| **方 法** | **返回值类型** | **作用** |
| toList() | List<T> | 把流中元素收集到List |
| toSet() | Set<T> | 把流中元素收集到Set |
| toCollection() | Collection<T> | 把流中元素收集到创建集合 |
| Counting() | Long | 计算流中元素个数 |

案例:



**参考博客**:https://blog.csdn.net/u010837355/article/details/124105657