Day03

新单词

- **transient**-稍纵即逝的
- **unsupported**-不支持的
- **encoding**-编码
- **try**-试试
- **catch**-抓住，捕获
- **throw**-抛出，扔出
- **error**-错误
- **AutoClose**-自动关闭

JAVA IO

对象流

对象输入流

java.io.ObjectInputStream使用对象流可以进行对象反序列化

构造器

```
ObjectInputStream(InputStream in)
将当前创建的对象输入流链接在指定的输入流上
```

方法

```
Object readObject()
进行对象反序列化并返回。该方法会从当前对象输入流链接的流中读取若干字节并将其还原为对象。这里要注意读取的字节必须是由ObjectOutputStream序列化一个对象所得到的字节。
```

例

```
package io;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

/**
 * 对象输入流，用来进行对象反序列化
 */
public class OISDemo {
```

```
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        //读取person.obj文件并将其中保存的数据进行反序列化
        FileInputStream fis = new FileInputStream("person.obj");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Person person = (Person)ois.readObject();
        System.out.println(person);

        ois.close();
    }
}
```

transient关键字

当一个属性被transient关键字修饰后，该对象在进行序列化时，转换出来的字节中是不包含该属性的。忽略不必要的属性可以达到对象"瘦身"的操作。

对象瘦身可以在对象持久化时减少磁盘开销。在进行传输时可以缩短传输速度。

如果该对象不需要序列化，那么该关键字不发挥其他任何效果

```
public class Person implements Serializable {
    private String name;                //姓名
    private int age;                    //年龄
    private String gender;              //性别
    private transient String[] otherInfo;      //其他信息
```

序列化时不包含otherInfo属性，并且反序列化时该属性值为null

```
Person{name='王克晶', age=18, gender='女', otherInfo=null}
```

字符流

- java将流按照读写单位划分为**字节流与字符流.**
- java.io.InputStream和OutputStream是所有字节流的超类
- **而java.io.Reader和Writer则是所有字符流的超类,它们和字节流的超类是平级关系**.
- Reader和Writer是两个抽象类,里面规定了所有字符流都必须具备的读写字符的相关方法.
- **字符流最小读写单位为字符(char)**,但是底层实际还是读写字节,只是字符与字节的转换工作由字符流完成.
- **字符流都是高级流**

超类

- java.io.Writer 所有字符输入流的超类

  常用方法

```
void write(int c):写出一个字符,写出给定int值”低16”位表示的字符。
void write(char[] chs):将给定字符数组中所有字符写出。
void write(String str):将给定的字符串写出
void write(char[] chs,int offset,int len):将给定的字符数组中从offset处开始连续的len个字符写出
```

- java.io.Reader 所有字符输出流的超类

  常用方法

```
int read():读取一个字符，返回的int值“低16”位有效。当返回值为-1时表达流读取到了末尾。
int read(char[] chs):从该流中读取一个字符数组的length个字符并存入该数组，返回值为实际读取到的字符量。当返回值为-1时表达流读取到了末尾。
```

转换流

java.io.InputStreamReader和OutputStreamWriter是常用的字符流的实现类。

实际开发中我们不会直接操作他们，但是他们在流连接中是必不可少的一环。

流连接中的作用

- 衔接字节流与其他字符流
- 将字符与字节相互转换

意义

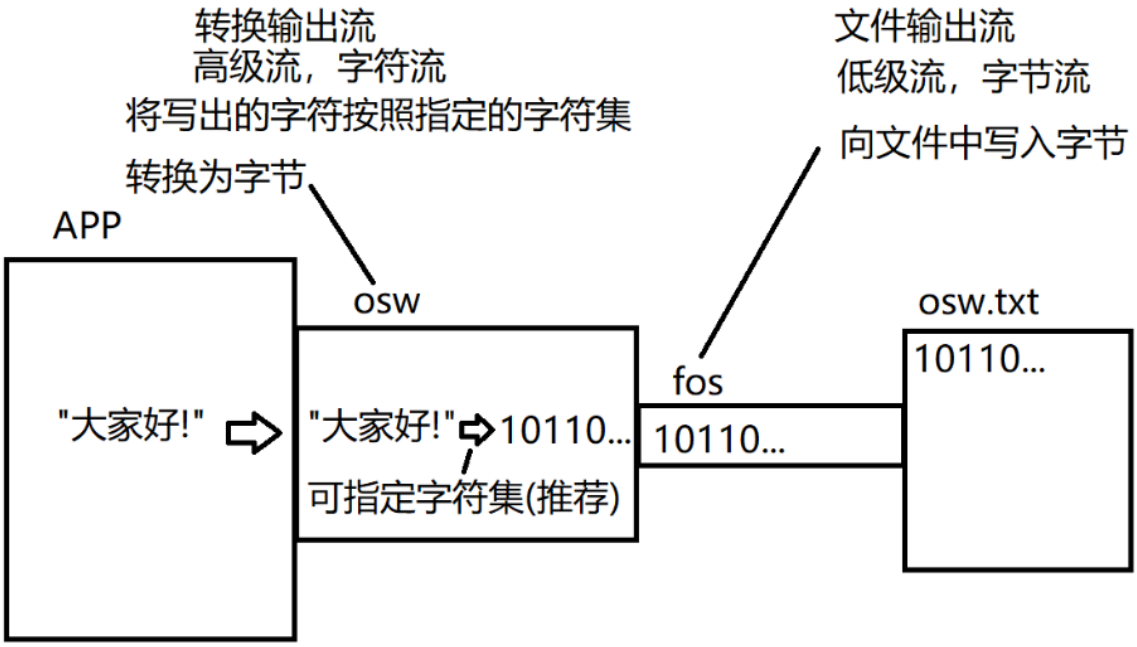
实际开发中我们还有功能更好用的字符高级流.但是其他的字符高级流都有一个共通点:不能直接连接在字节流上.而实际操作设备的流都是低级流同时也都是字节流.因此不能直接在流连接中串联起来.转换流是一对可以连接在字节流上的字符流,其他的高级字符流可以连接在转换流上.在流连接中起到"转换器"的作用(负责字符与字节的实际转换)

输出流

构造器

```
OutputStreamWriter(OutputStream out,Charset cs)
基于给定的字节输出流以及字符编码创建OSW

OutputStreamWriter(OutputStream out)
该构造方法会根据系统默认字符集创建OSW
```

示意



例

```
package io;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.nio.charset.StandardCharsets;

/**
 * 转换流写出文本数据
 */
public class OSWDemo {
    public static void main(String[] args) throws IOException {
        //向文件osw.txt中写出文本数据
        FileOutputStream fos = new FileOutputStream("osw.txt");
        OutputStreamWriter osw = new OutputStreamWriter(fos,
StandardCharsets.UTF_8);

        osw.write("夜空中最亮的星，能否听清，");
        osw.write("那仰望的人心底的孤独和叹息。");
        System.out.println("写出完毕");
        osw.close();

    }
}
```

输入流

构造器

```
InputStreamWriter(InputStream in,Charset cs)
基于给定的字节输入流以及字符编码创建当前转换流

InputStreamWriter(InputStream in)
该构造方法会根据系统默认字符集创建当前转换流
```

例

```
package io;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.nio.charset.StandardCharsets;

/**
 * 使用转换流读取文本数据
 */
public class ISRDemo {
    public static void main(String[] args) throws IOException {
        //将osw.txt文件中的文本信息读取回来
        FileInputStream fis = new FileInputStream("osw.txt");
        InputStreamReader isr = new InputStreamReader(fis,
StandardCharsets.UTF_8);
        //00000000 00000000 10011101 01110010
        int d;
        while(( d = isr.read()) != -1) {
            System.out.print((char) d);
        }
        isr.close();
    }
}
```

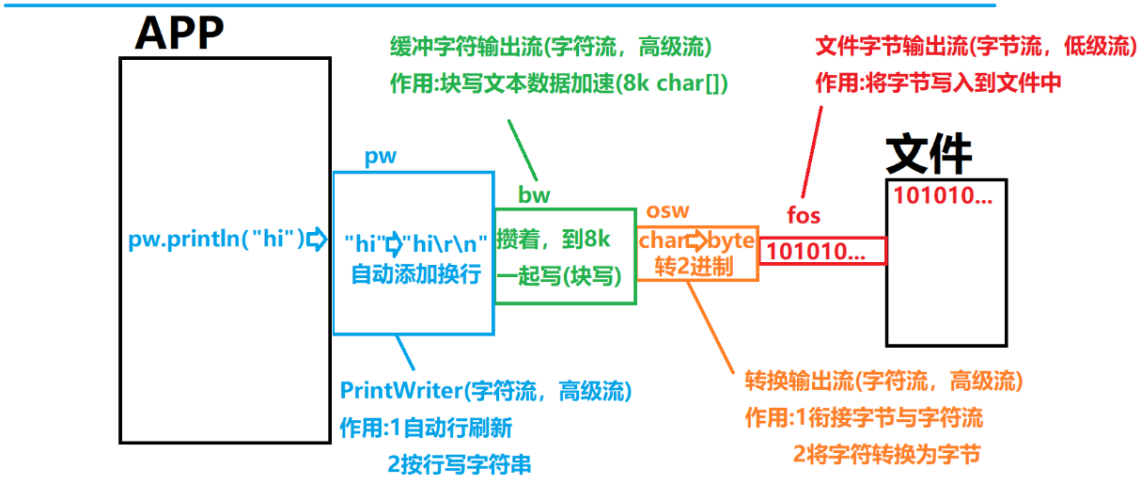
缓冲字符流

缓冲字符输出流-java.io.PrintWriter

- **java.io.BufferedWriter和BufferedReader**
- 缓冲字符流内部也有一个缓冲区,读写文本数据以块读写形式加快效率.并且缓冲流有一个特别的功能:**可以按行读写文本数据.缓冲流内部维护一个char数组，默认长度8192.以块读写方式读写字符数据保证效率**
- java.io.PrintWriter具有自动行刷新的缓冲字符输出流,实际开发中更常用.**它内部总是会自动连BufferedWriter作为块写加速使用**.

工作原理

```
this(new BufferedWriter(new OutputStreamWriter(new FileOutputStream(fileName))))
```



特点

- 可以按行写出字符串
- 具有自动行刷新功能

对文件写操作的构造器

```
PrintWriter(File file)
PrintWriter(String path)

还支持指定字符集
PrintWriter(File file,String csn)
PrintWriter(String path,String csn)

上述构造器看似PW可以直接对文件进行操作，但是它是一个高级流，实际内部会进行流连接：
this(new BufferedWriter(new OutputStreamWriter(new
FileOutputStream(fileName))),false);
如上面工作原理图
```

例

```
package io;

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.UnsupportedEncodingException;
import java.nio.charset.StandardCharsets;

/**
 * 缓冲字符输出流:java.io.PrintWriter
 * 特点:
 * 1:按行写出字符串
 * 2:具有自动的行刷新功能
 *
 */
```

```
public class PWDemo {
    public static void main(String[] args) throws FileNotFoundException,
UnsupportedEncodingException {
        //向文件中写入文本数据
        /*
            PrintWriter(File file)
            PrintWriter(String path)
         */
        PrintWriter pw = new PrintWriter("pw.txt");
//        PrintWriter pw = new PrintWriter("pw.txt", "UTF-8");
        pw.println("我祈祷拥有一颗透明的心灵，和会流泪的眼睛。");
        pw.println("给我再去相信的勇气，oh越过黄昏去拥抱你。");
        System.out.println("写出完毕");
        pw.close();
    }
}
```

其他构造器

PritWriter(Writer writer)
将当前实例化的PrintWriter链接在指定的字符输出流上

PrintWriter(OutputStream out)
将当前实例化的PrintWriter链接在指定的字节输出流上
由于除了转换流外的其他字符流都不能直接连在字节流上，因此这个构造器内部会自动链接在BufferedWriter上
并且让BufferedWriter链接在转换流OutputStream上，最后再让转换流链接再指定的字节输出流上

例

```
package io;

import java.io.*;
import java.nio.charset.StandardCharsets;

/**
 *  自行完成流连接向文件写出字符串
 */
public class PWDemo2 {
    public static void main(String[] args) throws FileNotFoundException {
        //负责:将写出的字节写入到文件中
        FileOutputStream fos = new FileOutputStream("pw2.txt");
        //负责:将写出的字符全部转换为字节(可以按照指定的字符集转换)
        OutputStreamWriter osw = new OutputStreamWriter(fos,
StandardCharsets.UTF_8);
        //负责:块写文本数据(攒够8192个字符一次性写出)
        BufferedWriter bw = new BufferedWriter(osw);
        //负责:按行写出字符串
        PrintWriter pw = new PrintWriter(bw);

        pw.println("你停在了这条我们熟悉的街,");
        pw.println("把你准备好的台词全念一遍。");
```

```
        System.out.println("写出完毕");
        pw.close();
    }
}
```

自动行刷新

PrintWriter支持自动行刷新，每当我们调用println方法写出一行内容后自动flush一次。

```
对应的构造器
PritWriter(Writer writer,boolean autoflush)
如果第二个参数为true则开启自动行刷新
```

例

```
package io;

import java.io.*;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

/**
 * 实现一个简易的记事本工具
 * 利用流连接
 * 在文件输出流上最终链接到PrintWriter上。
 * 然后将用户在控制台上输入的每一行字符串都可以按行写入到对应的文件中。
 * 当用户在控制台上单独输入"exit"时程序退出。
 */
public class AutoFlushDemo {
    public static void main(String[] args) throws FileNotFoundException {
        //负责:将写出的字节写入到文件中
        FileOutputStream fos = new FileOutputStream("note.txt");
        //负责:将写出的字符全部转换为字节(可以按照指定的字符集转换)
        OutputStreamWriter osw = new OutputStreamWriter(fos,
StandardCharsets.UTF_8);
        //负责:块写文本数据(攒够8192个字符一次性写出)
        BufferedWriter bw = new BufferedWriter(osw);
        //负责:按行写出字符串
        PrintWriter pw = new PrintWriter(bw,true);//开启自动行刷新

        Scanner scanner = new Scanner(System.in);
        System.out.println("请开始输入内容，单独输入exit退出");
        while(true){
            String line = scanner.nextLine();
            //String可以忽略大小写比较字符串内容:equalsIgnoreCase
            if("exit".equalsIgnoreCase(line)){
                break;
            }
            pw.println(line);//每当println后自动flush。注意:print方法并不会自动flush
        }
        System.out.println("再见!");
        pw.close();
    }
```

```
}
```

缓冲字符输入流-java.io.BufferedReader

缓冲字符输入流内部维护一个默认8192长度的char数组，总是以块读取文本数据保证读取效率。

缓冲输入流提供了一个**按行读取文本**数据的方法

```
String readLine()
返回一行字符串。方法内部会连续扫描若干个字符，直到遇到换行符为止，将换行符之前的内容以一个字符串形式返回。
返回的字符串中不含有最后的换行符。
返回值有三种情况:
1:正常一行内容
2:空字符串。当读取了一个空行时(这一行只有一个换行符)。
3:null。当流读取到了末尾时。

当我们第一次调用readLine()时，流并不是只读取了一行字符串，而是先进行块读操作(一次性读取8192个字符并转入到内部的char数组中)，然后扫描内部的char数组，然后将第一行字符串返回。第二次调用后再继续扫描后去的内容以此类推。
```

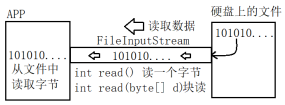
例

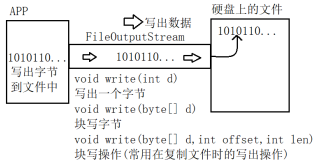
```
package io;

import java.io.*;

/**
 * 使用缓冲字符输入流读取文本数据
 */
public class BRDemo {
    public static void main(String[] args) throws IOException {
        //将当前源代码输出到控制台上
        /*
            1:创建文件输入流读取当前源代码文件
            2:进行流连接最终链接到BufferedReader上
            3:读取每一行字符串并输出到控制台上
         */
        FileInputStream fis = new FileInputStream(
                "./src/main/java/io/BRDemo.java"
        );
        InputStreamReader isr = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(isr);

        String line;
        while((line = br.readLine())!=null) {
            System.out.println(line);
        }
        br.close();
    }
}
```

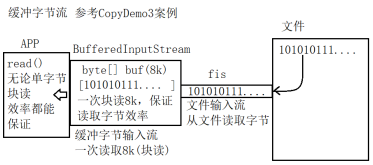
IO总结

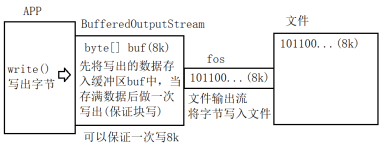
	输入流		输出流	
	字节输入流 java.io.InputStream	字符输入流 java.io.Reader	字节输出流 java.io.OutputStream	字符输出流 java.io.Writer
节点流 低级流	FileInputStream 文件输入流 作用: 实际连接程序与文件的管道 负责从文件中读取字节		FileOutputStream 文件输出流 作用: 实际连接程序与文件的管道 负责将字节写入文件中	
处理流 高级流	BufferedInputStream 缓冲字节输入流 作用: 块读字节数据加速 ObjectInputStream 对象输入流 作用: 进行对象反序列化	InputStreamReader 转换输入流 作用 1:衔接字节与字符流 2:将读取的字节转字符 BufferedReader 缓冲字符输入流 作用 1:块读文本数据加速 2:按行读取字符串	BufferedOutputStream 缓冲字节输出流 作用: 块写字节数据加速的 ObjectOutputStream 对象输出流 作用: 进行对象序列化	OutputStreamWriter 转换输出流 作用: 1:衔接字节与字符流 2:将写出的字符转字节 PrintWriter 缓冲字符输出流 带自动行刷新功能 作用 1:块写文本数据加速 2:按行写出字符串 3:自动行刷新

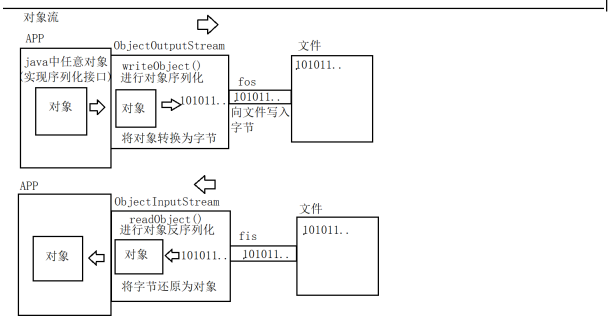
字节流相关图示





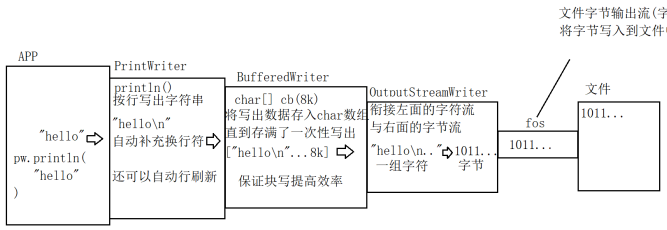




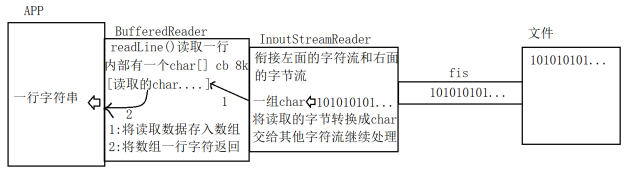


对象流作用:方便我们读写java对象，不再考虑对象与字节的转换工作

字符流相关图示



使用PrintWriter完成上述一系列流连接目的就是可以方便的按行写出字符串还能保证写出效率



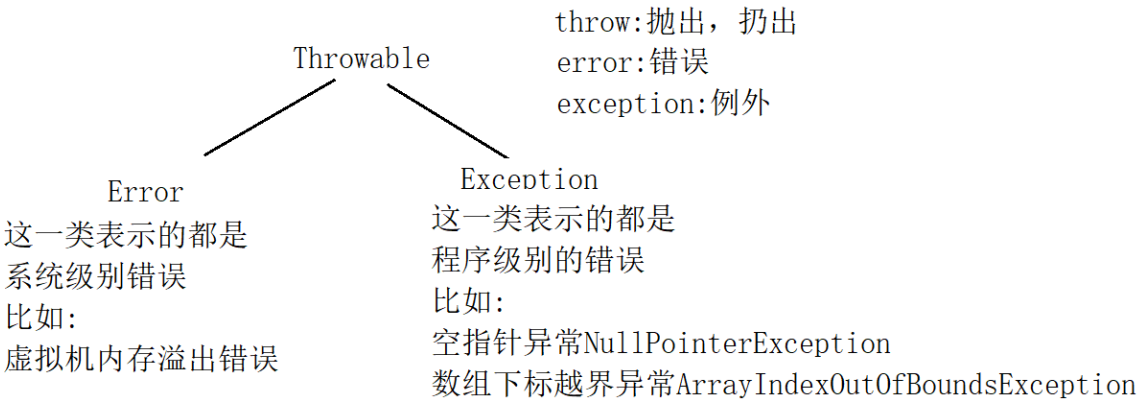
第一次调用readLine就一次性读取8192个字符存入
缓冲区(char数组)然后将其中第一行内容返回。
当再次调用readLine时，再将第二行内容直接返回。
直到数组所有内容均返回后会再次读取8192个字符，
如此往复以块读取保证效率。

使用BufferedReader完成上述流连接目的是可以按行读取字符串并且还可以保证读取效率

JAVA 异常处理

java异常处理机制

- java中所有错误的超类为:Throwable。其下有两个子类:Error和Exception
- Error的子类描述的都是系统错误，比如虚拟机内存溢出等。
- Exception的子类描述的都是程序错误，比如空指针，下表越界等。
- 通常我们程序中处理的异常都是Exception。



我们使用异常处理机制通常只关心Exception这类问题，很少关注Error
异常处理机制的目的在程序运行期间出现了错误后可以提供补救措施(B计划)

try-catch

```
package exception;

/**
 * 异常处理机制中的try-catch
 * 语法:
 * try{
 *     代码片段...
 * }catch(XXXException e){
 *     出现错误后的补救措施(B计划)
 * }
 */
public class TryCatchDemo {
    public static void main(String[] args) {
        System.out.println("程序开始了...");
        /*
            try{}语句块不能单独写，后面要么跟catch语句块要么跟finally语句块

            异常处理机制关注的是:明知道程序可能出现某种错误，但是该错误无法通过修改逻辑
            完全规避掉时，我们会使用异常处理机制，在出现该错误是提供一种补救办法。
            凡是能通过逻辑避免的错误都属于bug!就应当通过逻辑去避免!
         */
        try {
//            String str = null;
//            String str = "";
            String str = "a";
        /*
            若str=null的情况
            当JVM执行到下面代码时:str.length()会出现空指针，此时虚拟机就会根据该情况
            实例化一个对应的异常实例出来，即:空指针异常实例 NullPointerException实例
            然后将程序从一开始执行到报错这句话的过程设置到该异常实例中，此时该异常通过
            类型名字可以表达出现了什么错误，并将来可以通过输出错误信息来得知错误出现在那里
            虚拟机会将该异常抛出
            当某句代码抛出了一个异常时，JVM会做如下操作:
```

```
            1:检查报错这句话是否有被异常处理机制控制(有没有try-catch)
              如果有，则执行对应的catch操作，如果没有catch可以捕获该异常则视为没有
              异常处理动作
            2:如果没有异常处理，则异常会被抛出当当前代码所在的方法之外由调用当前方法的
              代码片段处理该异常
         */
            System.out.println(str.length());//抛出空指针异常
            System.out.println(str.charAt(0));
            System.out.println(Integer.parseInt(str));
            /*
                当try中某句代码报错后，就会跳出try执行下面对应的catch块，执行后就会
                退出catch继续向后执行。因此try语句块中报错代码以下的内容都不会被执行
             */
            System.out.println("!!!!!!!!!!!!!!!");
//        }catch(NullPointerException e){
//            //这里实际开发中是写补救措施的，通常也会将异常信息输出便于debug
//            System.out.println("出现了空指针，并解决了!");
//        }catch(StringIndexOutOfBoundsException e){
//            System.out.println("处理字符串下标越界问题!");
//        }
        /*
            当try语句块中可能出现的几种不同异常对应的处理办法相同时，可以采取合并
            catch的做法，用同一个catch来捕获这几种可能出现的异常，而执行措施使用
            同一个。
         */
        }catch(NullPointerException|StringIndexOutOfBoundsException e){
            System.out.println("处理空指针或下标越界!");
        /*
            当catch捕获某个超类型异常时，那么try语句块中出现它类型异常时都可以被这个
            catch块捕获并处理。

            如果多个catch捕获的异常之间存在继承关系时，一定是子类异常在上超类异常在下
         */
        }catch(Exception e){
            System.out.println("反正就是出了个错!");
        }
        System.out.println("程序结束了...");
    }
}
```

finally块

finally块是异常处理机制中的最后一块

- finally可以直接跟在try语句块之后
- finally可以跟在**最后一个**catch块之后
- fianlly下面不能再定义catch块

特点:

只要程序执行到异常处理机制中(执行到try语句块中)，无论try中的代码是否出现异常，finally最终都**必定执行**

作用:

通常用来执行释放资源这一类操作。例如IO操作完毕后的流关闭。

例

```
package exception;

/**
 * finally块
 */
public class FinallyDemo {
    public static void main(String[] args) {
        System.out.println("程序开始了");
        try {
            String str = "null";
            System.out.println(str.length());
            return;//结束方法，结束前finally还是要执行的
        } catch (Exception e) {
            System.out.println("出错了");
        }finally {
            System.out.println("finally中的代码执行了");
        }
        System.out.println("程序结束了");
    }
}
```

在IO中的应用

```
package exception;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 * 异常处理机制在IO中的应用
 */
public class FinallyDemo2 {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream("./fos.dat");
            fos.write(1);
        } catch (IOException e) {
            System.out.println("出错了");
        } finally {
```

```
            try {
                if(fos!=null) {
                    fos.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

自动关闭特性

JDK7之后java推出了一个新特性:自动关闭特性可以更优雅的在异常处理机制中关闭IO

语法

```
try(
    声明并初始化IO对象
){
    IO操作
}catch(IOException e){//catch各种IO异常
    ...
}
```

例

```
package exception;

import java.io.FileOutputStream;
import java.io.IOException;

/**
 * JDK7之后java推出了一个新特性:自动关闭特性
 * 可以更优雅的在异常处理机制中关闭IO
 */
public class AutoCloseableDemo {
    public static void main(String[] args) {
        //自动关闭特性是编译器认可的，编译后就变成FinallyDemo2的样子
        try(
                FileOutputStream fos = new FileOutputStream("fos.dat");
        ){
            fos.write(1);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

总结

java将流分为两类:节点流与处理流:

- **节点流**:也称为**低级流**.

  节点流的另一端是明确的,是实际读写数据的流,读写一定是建立在节点流基础上进行的.

- **处理流**:也称为**高级流**.

  处理流不能独立存在,必须连接在其他流上,目的是当数据流经当前流时对数据进行加工处理来简化我们对数据的该操作.

实际应用中,我们可以通过串联一组高级流到某个低级流上以流水线式的加工处理对某设备的数据进行读写,这个过程也成为流的连接,这也是IO的精髓所在.

缓冲流

缓冲流是一对高级流，在流链接中链接它的**目的是加快读写效率。**缓冲流内部**默认缓冲区为8kb**，缓冲流**总是块读写数据来提高读写效率。**

java.io.BufferedOutputStream缓冲字节输出流，继承自java.io.OutputStream

常用构造器

- BufferedOutputStream(OutputStream out)：创建一个默认8kb大小缓冲区的缓冲字节输出流，并连接到参数指定的字节输出流上。
- BufferedOutputStream(OutputStream out,int size)：创建一个size指定大小(单位是字节)缓冲区的缓冲字节输出流，并连接到参数指定的字节输出流上。

常用方法

`flush()`：强制将缓冲区中已经缓存的数据一次性写出

缓冲流的写出方法功能与`OutputStream`上一致，需要知道的时`write`方法调用后并非实际写出，而是先将数据存入缓冲区(内部的字节数组中)，当缓冲区满了时会自动写出一次。

java.io.BufferedInputStream缓冲字节输出流，继承自java.io.InputStream

常用构造器

- BufferedInputStream(InputStream in)：创建一个默认8kb大小缓冲区的缓冲字节输入流，并连接到参数指定的字节输入流上。
- BufferedInputStream(InputStream in,int size)：创建一个size指定大小(单位是字节)缓冲区的缓冲字节输入流，并连接到参数指定的字节输入流上。

常用方法

缓冲流的读取方法功能与`InputStream`上一致，需要知道的时`read`方法调用后缓冲流会一次性读取缓冲区大小的字节数据并存入缓冲区，然后再根据我们调用`read`方法读取的字节数进行返回，直到缓冲区所有数据都已经通过`read`方法返回后会再次读取一组数据进缓冲区。即:块读取操作

对象流

对象流是一对高级流，在流链接中的作用是完成对象的**序列化**与**反序列化**

序列化：是对象输出流的工作，将一个对象按照其结构转换为一组字节的过程。

反序列化：是对象输入流的工作，将一组字节还原为对象的过程。

java.io.ObjectInputStream对象输入流，继承自java.io.InputStream

常用构造器

ObjectInputStream(InputStream in)：创建一个对象输入流并连接到参数in这个输入流上。

常用方法

Object readObject()：进行对象反序列化，将读取的字节转换为一个对象并以Object形式返回(多态)。

如果读取的字节表示的不是一个java对象会抛出异常:java.io.ClassNotFoundException

java.io.ObjectOutputStream对象输出流，继承自java.io.OutputStream

常用构造器

ObjectOutputStream(OutputStream out)：创建一个对象输出流并连接到参数out这个输出流上

常用方法

void writeObject(Object obj)：进行对象的序列化，将一个java对象序列化成一组字节后再通过连接的输出流将这组字节写出。

如果序列化的对象没有实现可序列化接口:java.io.Serializable就会抛出异常:java.io.NotSerializableException

序列化接口java.io.Serrializable

该接口没有任何抽象方法，但是只有实现了该接口的类的实例才能进行序列化与反序列化。

实现了序列化接口的类建议显示的定义常量:static final long serialVersionUID = 1L;

可以为属性添加关键字**transient**，被该关键字修饰的属性在序列化是会被忽略，达到对象**序列化瘦身**的目的。

字符流

java将流按照读写单位划分为字节与字符流。字节流以字节为单位读写，字符流以字符为单位读写。

转换流java.io.InputStreamReader和OutputStreamWriter

功能无需掌握，了解其核心意义:

1:衔接其它字节与字符流

2:将字符与字节进行转换

相当于是现实中的"转换器"

缓冲字符输出流

缓冲字符输出流需要记住的是PrintWriter和BufferedReader

作用:

1:块写或块读文本数据加速

2:可以按行写或读字符串

java.io.PrintWriter 具有自动行刷新的缓冲字符输出流

常用构造器

PrintWriter(String filename) :可以直接对给定路径的文件进行写操作

PrintWriter(File file):可以直接对File表示的文件进行写操作

上述两种构造器内部会自动完成流连接操作。

PrintWriter(OutputStream out):将PW链接在给定的字节流上(构造方法内部会自行完成转换流等流连接)

PrintWriter(Writer writer):将PW链接在其它字符流上

PrintWriter(OutputStream out,boolean autoflush)

PrintWriter(Writer writer,boolean autoflush)

上述两个构造器可以在链接到流上的同时传入第二个参数，如果该值为true则开启了自动行刷新功能。

常用方法

void println(String line)：按行写出一行字符串

特点

自动行刷新，当打开了该功能后，每当使用println方法写出一行字符串后就会自动flush一次

java异常处理机制:

- 异常处理机制是用来处理那些可能存在的异常，但是无法通过修改逻辑完全规避的场景。
- 而如果通过修改逻辑可以规避的异常是bug，不应当用异常处理机制在运行期间解决！应当在编码时及时修正

try语句块用来包含可能出错的代码片段

catch用来捕获并处理对应的异常，可以定义多个，也可以合并多个异常在一个catch中。