File和IO重难点回顾

- File对象的含义?(代表计算机中的文件和目录)
- 基于File对象进行文件操作?(创建目录和文件,查找文件和目录,删除目录和文件,复制文件和目录,...)
- I/O流(读写内存,磁盘,网络中数据的对象,这里的流为数据流,这里的I/O为读写数据的对象)
- I/O流分类(按流向可分为输出流和输入流,按处理单元可划分为字节流和字符流,按功能单元可分为高级流和节点流)
- I/O的应用(创建流对象,读写数据,释放资源)
- 字节流超类(InputStream/OutputStream)
- 文件字节流(FileInputStream/FileOutputStream)
- 高级处理流(BufferedInputStream/BuffedOutputStream,ObjectOutputStream/ObjecInputStream,...)
- 字符流超类(Reader/Writer,本质上还是字节流)
- 文件字符流(FileReader/FileWriter)
- 转换流(InputStreamReader/OutputStreamWriter)
- 打印流(PrintWriter)

FAQ分析

- 如何理解项目中的文件上传?(从本地读,然后写到远端服务器上)
- 如何理解项目中文件的下载?(读远端服务器上的数据,然后写到本地电脑上)
- IO读写数据时常见的异常是什么?(FileNotFoundException,IOException)
- 高级流和低级流这里用到了什么设计模式？(装饰模式-例如高级流在低级流的基础上进行装饰)
- 序列化的应用场景？(内存对象的持久化，写入缓存，网络数据交互)
- serialVersionUID作用？(保证序列化和反序列化的正确性)
- 如何进行序列化的粒度控制？(默认所有属性都序列化，transient描述的属性可以不序列化)
- 如何理解对象的深拷贝和浅拷贝？(深拷贝拷贝的是内容，浅拷贝拷贝的是对象地址)

今日课程目标和安排

- 掌握异常的定义及处理方式
- 熟悉网络编程的基本概念以及入门实现

异常

异常的定义

程序中异常是对特定问题的一个描述，通过异常可以终止程序继续执行。在Java中异常类型为Throwable,它有两个子类分别Exception和Error,Exception是我们程序可以处理的异常，例如下标越界异常，算数异常，空指针异常，IO异常等，Error一般一种系统级的错误，例如内存溢出等。

程序中异常的作用?

- 终止程序执行。
- 实现与用户的交互。

入门案例分享

- 基于try{}catch(xx){}结构处理空指针异常

```
package exception;
public class NPEDemo{//NullPointerException->NPE

  public static void main(String[] args){
    doStringOper01(null);
  }
   public static void doStringOper01(String str){
     //如下语句可能会出现空指针异常
     String result=str.toUpperCase();
     //假如上面的语句出现了异常，下面输出语句是不会执行的。
     System.out.println(result);
    }
      public static void doStringOper02(String str){
        //假如希望如下语句出现了异常，后面的语句还要继续执行。
        //此时我们可以使用try{}catch(xxx){}...对可能出现异常的语句进行处理
        String result;
        try{
            result=str.toUpperCase();
        }catch(NullPointerException e){
            System.out.println("程序中的str的值不能null");
            result=null;
        }
        System.out.println(result);
    }

}
```

- 多catch(xx){}结构异常处理

```
package exception;

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

public class IOExcetionDemo {
    public static void main(String[] args){
        FileOutputStream fos=null;
        try {
            //1.构建输出流对象(编译阶段能够检测到的异常我们通常称之为检查异常)
            fos = new FileOutputStream("./abc/f1.txt", true);
            //2.写数据到文件
            fos.write("hello".getBytes(StandardCharsets.UTF_8));
        }catch (FileNotFoundException e){
           // System.out.println("文件没找到:"+e.getMessage());
            e.printStackTrace();//打印异常栈信息(包含的异常信息会更全面)
            //return;//遇到return语句时，是先finally，然后再返回
            //System.exit(-1);这条语句执行时，finally不在执行
        }catch (IOException e){
            //System.out.println("写数据或关闭流时出现了问题:"+e.getMessage());
            e.printStackTrace();//打印异常栈信息(包含的异常信息会更全面)
```

```
        }
    }
}
```

多catch结构合并

当多个异常，他们的处理方式相同时，可以对这些异常进行合并处理

```
package exception;

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

public class IOExcetionDemo {
    public static void main(String[] args){
        FileOutputStream fos=null;
        try {
            //1.构建输出流对象(编译阶段能够检测到的异常我们通常称之为检查异常)
            fos = new FileOutputStream("./abc/f1.txt", true);
            //2.写数据到文件
            String s="hello";//假设这个s的值会通过外部传入
            fos.write(s.getBytes(StandardCharsets.UTF_8));
        }catch (FileNotFoundException | NullPointerException e){
           // System.out.println("文件没找到:"+e.getMessage());
            e.printStackTrace();//打印异常栈信息(包含的异常信息会更全面)
            //return;//遇到return语句时，是先finally，然后再返回
            //System.exit(-1);这条语句执行时，finally不在执行
        }catch (IOException e){
            //System.out.println("写数据或关闭流时出现了问题:"+e.getMessage());
            e.printStackTrace();//打印异常栈信息(包含的异常信息会更全面)
        }
    }
}
```

finally结构

假如现在有一个语句，无论是否出现异常，它都要执行，此时这样的语句可以放在finally结构中。

```
package exception;

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

public class IOExcetionDemo {
    public static void main(String[] args){
        FileOutputStream fos=null;
        try {
            //1.构建输出流对象(编译阶段能够检测到的异常我们通常称之为检查异常)
```

```
            fos = new FileOutputStream("./abc/f1.txt", true);
            //2.写数据到文件
            String s="hello";
            fos.write(s.getBytes(StandardCharsets.UTF_8));
        }catch (FileNotFoundException | NullPointerException e){
           // System.out.println("文件没找到:"+e.getMessage());
            e.printStackTrace();//打印异常栈信息(包含的异常信息会更全面)
            //return;//遇到return语句时，是先finally，然后再返回
            //System.exit(-1);这条语句执行时，finally不在执行
        }catch (IOException e){
            System.out.println("写数据或关闭流时出现了问题:"+e.getMessage());
        }finally {//最终执行语句，无论是否出现异常，这个代码块都会执行
            //3.释放资源(一般都会把释放资源的过程放在此代码块中)
            if(fos!=null)try{fos.close();}catch (IOException e)
{e.printStackTrace();}
        }
    }
}
```

自关闭特性

我们创建的对象需要关闭时，一种方法是写在finally中，还有一种方式在try()接口中直接创建，这样对象无需手动再关闭。前提时try内部的对象需要实现 AutoCloseable 接口。

```
package io;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

public class AutoCloseableDemo {
    public static void main(String[] args) {
        try(
                FileOutputStream fos =//这里构建对象无需手动关闭
                        new FileOutputStream("./abc/f1.txt", true);
        ){
            //1.构建输出流对象(编译阶段能够检测到的异常我们通常称之为检查异常)
            //2.写数据到文件
            String s="hello";
            fos.write(s.getBytes(StandardCharsets.UTF_8));
        }catch (FileNotFoundException | NullPointerException e){
            // System.out.println("文件没找到:"+e.getMessage());
            e.printStackTrace();//打印异常栈信息(包含的异常信息会更全面)
            //return;//遇到return语句时，是先finally，然后再返回
            //System.exit(-1);这条语句执行时，finally不在执行
        } catch (IOException e){
            System.out.println("写数据或关闭流时出现了问题:"+e.getMessage());
        }
    }
}
```

throw关键字应用

throw关键字的作用主要是用于抛出异常，这个关键字后面跟的是异常对象。
这个异常会抛给调用着，谁调用了这个方法，这个异常就抛给谁。

```
package io;
public class ThrowDemo {
    static void doCompute01(int a,int b){
        if(b==0) {
            //抛出参数无效的异常,这里用户传什么参数我们决定不了
            //只能告诉用户你传递的参数是非法的，是不合理的。
            throw new IllegalArgumentException("除数不能为0");
        }
        int result=a/b;
        System.out.println(result);
    }

    static void doCompute02(int a,int b){
        try {
            int result = a / b;
            System.out.println(result);
        }catch (ArithmeticException e){
            //这里以后是日志记录
            System.out.println("除数不能为0");
            throw e;//处理以后，可以继续抛给调用着
        }
        System.out.println("==finish==");
    }
    public static void main(String[] args) {
      doCompute01(10,0);
      doCompute02(10,0);
    }
}
```

throws关键字应用

throws关键字用于方法声明处，声明要抛出的异常，这个关键字后跟的是异常类型。
它的目的是告诉调用方，这个方法可能会出现此类型的异常，你要引起注意。当然
方法内的异常你不想处理，可通过此关键字声明抛出。

```
package io;

import java.io.*;
public class ThrowsDemo {

    public static void doCopy() throws IOException {
        FileInputStream fis=new FileInputStream("a.png");
        FileOutputStream fos=new FileOutputStream("c.png");
        byte[] buf=new byte[fis.available()];
        fis.read(buf);
        fos.write(buf);
        fis.close();
        fos.close();
        System.out.println("copy ok");
    }
```

```
    public static void main(String[] args) throws IOException{
        doCopy();
    }
}
```

- throws 关键字后可以抛出多个异常类型，这里抛出异常类型可以是方法内部抛出异常类型的父类类型。
- 方法重写时，子类方法抛出的异常可以和父类方法相同或者时父类方法异常的子类类型。
- 方法重写时，子类方法抛出的异常个数不能多于父类方法抛出异常的异常个数。

Java中的检查和非检查异常

- 检查异常：编译阶段，编译器能够检测到异常，这种异常必须处理，处理的方式，要么捕获，要么继续抛出。
- 非检查异常：编译阶段无法检测到的异常，程序员对此类异常可以有选择性的处理。
  典型代表为RuntimeException或它的子类。

自定义异常?

当系统提供的异常类型，不足以满足我们对异常的需求时，我们可以选择自定义异常，尤其是和业务相关性比较的异常。 例如PhoneNotFoundException;
我们自定义异常一般都是继承RuntimeException或Exception类型，然后重写相关方法。

```
package exception;
/**
 * 自定义异常
 */
public class PhoneNotFoundException extends RuntimeException{
    private static final long serialVersionUID = -6274120866460982066L;

    public PhoneNotFoundException() {
    }
    public PhoneNotFoundException(String message) {
        super(message);
    }
    public PhoneNotFoundException(Throwable e) {
        super(e);
    }
}
```

总结(Summary)

重难点分析

- 异常的定义?(什么是异常,异常类型,异常的处理,自定义异常)
- 异常的类型?(Throwable,Exception,Error)
- 异常的处理?(异常捕获,异常抛出)
- 异常的捕获?(try{可能有异常异常语句}catch(异常类型){异常处理},
  try{}catch(){}...,try{}catch(){}....finally{},try{}finally{},异常合并,自动关闭)

- 异常的抛出?(throw-方法内部抛出异常对象,throws-方法声明处抛出异常)
- 检查异常和非检查异常(编译器能够检测到的异常称之为检查异常)
- 自定义异常?(直接或间接的继承Exception或RuntimeException,然后添加构造方法)

常见FAQ

- 为什么要有异常的定义?(控制程序的执行流,实现与调用方交互)
- Java中异常的超类类型是什么?(Throwable)
- Java中非检查异常的类型是什么?(RuntimeException-运行时异常)
- throws关键字的做用是什么?(声明方法执行时可能出现的异常)
- throw关键字的作用是什么?(在方法内声明抛出异常,控制程序的执行流)
- try{}代码块越大越好吗?(不是,太大会对性能有一定的影响)
- finally代码块的作用?(通常在这里会释放资源)
- finally代码块什么情况下不执行?(System.exist(-1))
- 我们为什么要自己定义异常类型?(业务需要,官方定义的异常不能满足我们需求)