day05

异常处理

异常处理机制中的finally

- finally块定义在异常处理机制中的最后一块。它可以直接跟在try之后，或者最后一个catch之后。
- finally可以保证只要程序执行到了try语句块中，无论try语句块中的代码是否出现异常，最终finally都必定执行。
- finally通常用来做释放资源这类操作。

```
package exception;

/**
 * finally块
 * finally块是异常处理机制中的最后一块，它可以直接跟在try语句块之后或者最后一个catch块
 * 之后。
 * finally可以保证只要程序执行到try语句块中，无论try语句块中的代码是否出现异常，finally
 * 都【必定执行】！
 * 通常finally块用于做释放资源这类操作，比如IO操作后的关闭流动作就非常适合在finally中进行
 */
public class FinallyDemo {
    public static void main(String[] args) {
        System.out.println("程序开始了...");
        try{
            String str = "abc";
            System.out.println(str.length());
            return;
        }catch(Exception e){
            System.out.println("出错了，并处理了");
        }finally{
            System.out.println("finally中的代码执行了!");
        }
        System.out.println("程序结束了");
    }
}
```

IO操作时的异常处理机制应用

```
package exception;

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

/**
 *  异常处理机制在IO中的实际应用
 */
public class FinallyDemo2 {
    public static void main(String[] args) {
        FileOutputStream fos = null;
```

```
        try {
            fos = new FileOutputStream("fos.dat");
            fos.write(1);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if(fos!=null) {
                    fos.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

自动关闭特性

JDK7之后，java提供了一个新的特性:自动关闭。旨在IO操作中可以更简洁的使用异常处理机制完成最后的close操作。

```
语法:
try(
    定义需要在finally中调用close()方法关闭的对象.
){
    IO操作
}catch(XXXException e){
    ...
}
```

上述语法中可在try的"()"中定义的并初始化的对象必须实现了java.io.AutoCloseable接口,否则编译不通过.

```
package exception;

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

/**
 * JDK7之后java推出了一个特性:自动关闭特性
 * 旨在让我们用更简洁的语法完成IO操作的异常处理机制(主要就是简化了finally关闭流的操作)
 */
public class AutoCloseableDemo {
    public static void main(String[] args) {
        /*
            该特性是编译器认可的，并非虚拟机。实际上编译器编译完毕后的样子可参考FinallyDemo2
         */
        try(
                //只有实现了AutoCloseable接口的类才可以在这里定义!编译器最终会补充代码在
finally中调用其close关闭
                FileOutputStream fos = new FileOutputStream("fos.dat");
        ){
```

```
            fos.write(1);
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

上述代码是编译器认可的，而不是虚拟机。编译器在编译上述代码后会在编译后的class文件中改回成FinallyDemo2案例的代码样子(上次课最后的案例)。

throw关键字

throw用来对外主动抛出一个异常，通常下面两种情况我们主动对外抛出异常:

- 1:当程序遇到一个满足语法，但是不满足业务要求时，可以抛出一个异常告知调用者。
- 2:程序执行遇到一个异常，但是该异常不应当在当前代码片段被解决时可以抛出给调用者。

```
package exception;

/**
 * 测试异常的抛出
 */
public class Person {
    private int age;

    public int getAge() {
        return age;
    }

    public void setAge(int age) throws Exception {
        if(age<0||age>100){
            //使用throw对外抛出一个异常
            throw new RuntimeException("年龄不合法!");
        }
        this.age = age;
    }
}
package exception;

/**
 * throw关键字，用来对外主动抛出一个异常。
 * 通常下面两种情况我们主动对外抛出异常:
 * 1:当程序遇到一个满足语法，但是不满足业务要求时，可以抛出一个异常告知调用者。
 * 2:程序执行遇到一个异常，但是该异常不应当在当前代码片段被解决时可以抛出给调用者。
 */
public class ThrowDemo {
    public static void main(String[] args) {
        Person p = new Person();
        p.setAge(10000);//符合语法，但是不符合业务逻辑要求。
        System.out.println("此人年龄:"+p.getAge());
    }
}
```

throws关键字

当一个方法中使用throw抛出一个非RuntimeException的异常时，就要在该方法上使用throws声明这个异常的抛出。此时调用该方法的代码就必须处理这个异常，否则编译不通过。

```
package exception;

/**
 * 测试异常的抛出
 */
public class Person {
    private int age;

    public int getAge() {
        return age;
    }

    /**
     * 当一个方法使用throws声明异常抛出时,调用此方法的代码片段就必须处理这个异常
     */
    public void setAge(int age) throws Exception {
        if(age<0||age>100){
            //使用throw对外抛出一个异常
//            throw new RuntimeException("年龄不合法!");
            //除了RuntimeException之外,抛出什么异常就要在方法上声明throws什么异常
            throw new Exception("年龄不合法!");
        }
        this.age = age;
    }
}
```

当我们调用一个含有throws声明异常抛出的方法时，编译器要求我们必须处理这个异常，否则编译不通过。 处理手段有两种:

- 使用try-catch捕获并处理这个异常
- 在当前方法(本案例就是main方法)上继续使用throws声明该异常的抛出给调用者解决。 具体选取那种取决于异常处理的责任问题。

```
package exception;

/**
 * throw关键字，用于主动对外抛出一个异常
 */
public class ThrowDemo {
    public static void main(String[] args){
        System.out.println("程序开始了...");
        try {
            Person p = new Person();
            /*
                当我们调用一个含有throws声明异常抛出的方法时,编译器要求
                我们必须添加处理异常的手段,否则编译不通过.而处理手段有两种
                1:使用try-catch捕获并处理异常
                2:在当前方法上继续使用throws声明该异常的抛出
                具体用哪种取决于异常处理的责任问题
```

```
             */
            p.setAge(100000);//典型的符合语法，但是不符合业务逻辑要求
            System.out.println("此人年龄:"+p.getAge()+"岁");
        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("程序结束了...");
    }
}
```

注意，永远不应当在main方法上使用throws!!

####

含有throws的方法被子类重写时的规则

```
package exception;

import java.awt.*;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.SQLException;

/**
 * 子类重写超类含有throws声明异常抛出的方法时对throws的几种特殊的重写规则
 */
public class ThrowsDemo {
    public void dosome()throws IOException, AWTException {}
}
class SubClass extends ThrowsDemo{
//    public void dosome()throws IOException, AWTException {}

    //可以不再抛出任何异常
//    public void dosome(){}

    //可以仅抛出部分异常
//    public void dosome()throws IOException {}

    //可以抛出超类方法抛出异常的子类型异常
//    public void dosome()throws FileNotFoundException {}

    //不允许抛出额外异常(超类方法中没有的,并且没有继承关系的异常)
//    public void dosome()throws SQLException {}

    //不可以抛出超类方法抛出异常的超类型异常
//    public void dosome()throws Exception {}
}
```

java网络编程

java.net.Socket

Socket(套接字)封装了TCP协议的通讯细节，是的我们使用它可以与服务端建立网络链接，并通过 它获取两个流(一个输入一个输出)，然后使用这两个流的读写操作完成与服务端的数据交互

java.net.ServerSocket

ServerSocket运行在服务端，作用有两个:

1:向系统申请服务端口，客户端的Socket就是通过这个端口与服务端建立连接的。

2:监听服务端口，一旦一个客户端通过该端口建立连接则会自动创建一个Socket，并通过该Socket与客户端进行数据交互。

如果我们把Socket比喻为电话，那么ServerSocket相当于是某客服中心的总机。

与服务端建立连接案例:

```
package socket;

import java.io.IOException;
import java.net.Socket;

/**
 * 聊天室客户端
 */
public class Client {
    /*
        java.net.Socket 套接字
        Socket封装了TCP协议的通讯细节，我们通过它可以与远端计算机建立链接，
        并通过它获取两个流(一个输入，一个输出)，然后对两个流的数据读写完成
        与远端计算机的数据交互工作。
        我们可以把Socket想象成是一个电话，电话有一个听筒(输入流)，一个麦克
        风(输出流)，通过它们就可以与对方交流了。
     */
    private Socket socket;

    /**
     * 构造方法，用来初始化客户端
     */
    public Client(){
        try {
            System.out.println("正在链接服务端...");
            /*
                实例化Socket时要传入两个参数
                参数1:服务端的地址信息
                      可以是IP地址，如果链接本机可以写"localhost"
                参数2:服务端开启的服务端口
                我们通过IP找到网络上的服务端计算机，通过端口链接运行在该机器上
                的服务端应用程序。
                实例化的过程就是链接的过程，如果链接失败会抛出异常:
                java.net.ConnectException: Connection refused: connect
             */
            socket = new Socket("localhost",8088);
```

```
            System.out.println("与服务端建立链接!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 客户端开始工作的方法
     */
    public void start(){

    }

    public static void main(String[] args) {
        Client client = new Client();
        client.start();
    }
}
package socket;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 聊天室服务端
 */
public class Server {
    /**
     * 运行在服务端的ServerSocket主要完成两个工作:
     * 1:向服务端操作系统申请服务端口，客户端就是通过这个端口与ServerSocket建立链接
     * 2:监听端口，一旦一个客户端建立链接，会立即返回一个Socket。通过这个Socket
     *   就可以和该客户端交互了
     *
     * 我们可以把ServerSocket想象成某客服的"总机"。用户打电话到总机，总机分配一个
     * 电话使得服务端与你沟通。
     */
    private ServerSocket serverSocket;

    /**
     * 服务端构造方法，用来初始化
     */
    public Server(){
        try {
            System.out.println("正在启动服务端...");
            /*
                实例化ServerSocket时要指定服务端口，该端口不能与操作系统其他
                应用程序占用的端口相同，否则会抛出异常:
                java.net.BindException:address already in use

                端口是一个数字，取值范围:0-65535之间。
                6000之前的的端口不要使用，密集绑定系统应用和流行应用程序。
             */
            serverSocket = new ServerSocket(8088);
            System.out.println("服务端启动完毕!");
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开始工作的方法
     */
    public void start(){
        try {
            System.out.println("等待客户端链接...");
            /*
                ServerSocket提供了接受客户端链接的方法:
                Socket accept()
                这个方法是一个阻塞方法，调用后方法"卡住"，此时开始等待客户端
                的链接，直到一个客户端链接，此时该方法会立即返回一个Socket实例
                通过这个Socket就可以与客户端进行交互了。

                可以理解为此操作是接电话，电话没响时就一直等。
             */
            Socket socket = serverSocket.accept();
            System.out.println("一个客户端链接了！");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }
}
```

客户端与服务端完成第一次通讯(发送一行字符串)

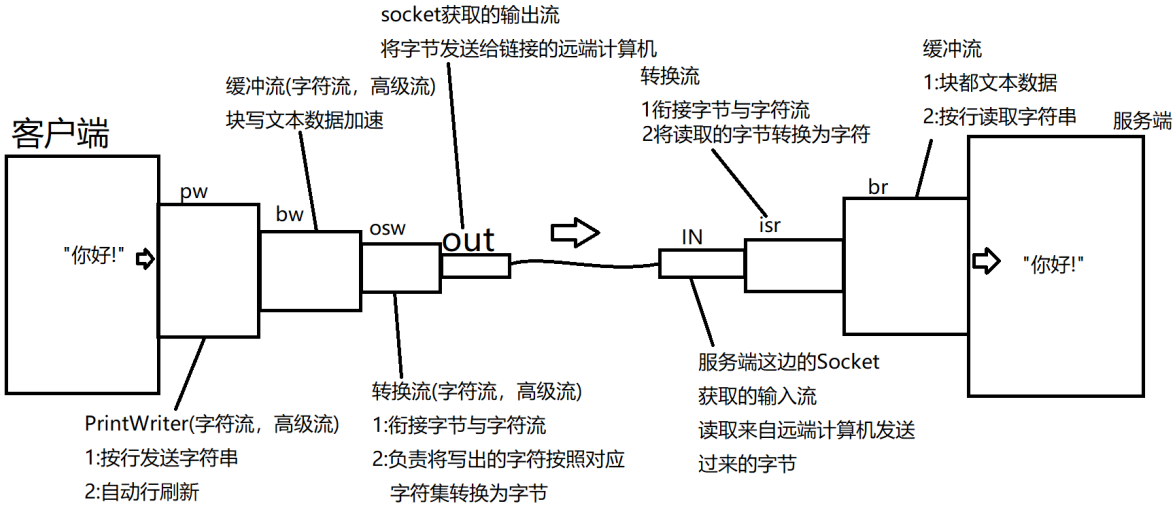
Socket提供了两个重要的方法:

OutputStream getOutputStream()

该方法会获取一个字节输出流，通过这个输出流写出的字节数据会通过网络发送给对方。

InputStream getInputStream()

通过该方法获取的字节输入流读取的是远端计算机发送过来的数据。



客户端代码：

```
package socket;

import java.io.*;
import java.net.Socket;

/**
 * 聊天室客户端
 */
public class Client {
    /*
        java.net.Socket 套接字
        Socket封装了TCP协议的通讯细节，我们通过它可以与远端计算机建立链接，
        并通过它获取两个流(一个输入，一个输出)，然后对两个流的数据读写完成
        与远端计算机的数据交互工作。
        我们可以把Socket想象成是一个电话，电话有一个听筒(输入流)，一个麦克
        风(输出流)，通过它们就可以与对方交流了。
     */
    private Socket socket;

    /**
     * 构造方法，用来初始化客户端
     */
    public Client(){
        try {
            System.out.println("正在链接服务端...");
            /*
                实例化Socket时要传入两个参数
                参数1:服务端的地址信息
                      可以是IP地址，如果链接本机可以写"localhost"
                参数2:服务端开启的服务端口
                我们通过IP找到网络上的服务端计算机，通过端口链接运行在该机器上
                的服务端应用程序。
                实例化的过程就是链接的过程，如果链接失败会抛出异常:
                java.net.ConnectException: Connection refused: connect
             */
            socket = new Socket("localhost",8088);
            System.out.println("与服务端建立链接!");
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }

    /**
     * 客户端开始工作的方法
     */
    public void start(){
        try {
            /*
                Socket提供了一个方法:
                OutputStream getOutputStream()
                该方法获取的字节输出流写出的字节会通过网络发送给对方计算机。
             */
            //低级流，将字节通过网络发送给对方
            OutputStream out = socket.getOutputStream();
            //高级流，负责衔接字节流与字符流，并将写出的字符按指定字符集转字节
            OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");
            //高级流，负责块写文本数据加速
            BufferedWriter bw = new BufferedWriter(osw);
            //高级流，负责按行写出字符串，自动行刷新
            PrintWriter pw = new PrintWriter(bw,true);

            pw.println("你好服务端!");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.start();
    }
}
```

服务端代码:

```
package socket;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * 聊天室服务端
 */
public class Server {
    /**
     * 运行在服务端的ServerSocket主要完成两个工作:
     * 1:向服务端操作系统申请服务端口，客户端就是通过这个端口与ServerSocket建立链接
     * 2:监听端口，一旦一个客户端建立链接，会立即返回一个Socket。通过这个Socket
```

```
     *    就可以和该客户端交互了
     *
     * 我们可以把ServerSocket想象成某客服的"总机"。用户打电话到总机，总机分配一个
     * 电话使得服务端与你沟通。
     */
    private ServerSocket serverSocket;

    /**
     * 服务端构造方法，用来初始化
     */
    public Server(){
        try {
            System.out.println("正在启动服务端...");
            /*
                实例化ServerSocket时要指定服务端口，该端口不能与操作系统其他
                应用程序占用的端口相同，否则会抛出异常:
                java.net.BindException:address already in use

                端口是一个数字，取值范围:0-65535之间。
                6000之前的的端口不要使用，密集绑定系统应用和流行应用程序。
             */
            serverSocket = new ServerSocket(8088);
            System.out.println("服务端启动完毕!");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开始工作的方法
     */
    public void start(){
        try {
            System.out.println("等待客户端链接...");
            /*
                ServerSocket提供了接受客户端链接的方法:
                Socket accept()
                这个方法是一个阻塞方法，调用后方法"卡住"，此时开始等待客户端
                的链接，直到一个客户端链接，此时该方法会立即返回一个Socket实例
                通过这个Socket就可以与客户端进行交互了。

                可以理解为此操作是接电话，电话没响时就一直等。
             */
            Socket socket = serverSocket.accept();
            System.out.println("一个客户端链接了！");

            /*
                Socket提供的方法:
                InputStream getInputStream()
                获取的字节输入流读取的是对方计算机发送过来的字节
             */
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in,"UTF-8");
            BufferedReader br = new BufferedReader(isr);
```

```
            String message = br.readLine();
            System.out.println("客户端说:"+message);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }
}
```

总结

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。

finally是异常的最后一块，只要程序执行到try中则必走。一般用于释放资源这类操作。

throw用于主动对外抛出异常。要么是满足语法不满足业务主动抛出异常，要么就是实际发生了异常但是不应当在当前代码片段被解决是抛出。具体情况要结合实际业务分析。

throws用于在方法声明时声明该异常的抛出，使得调用者必须处理该异常。