JAVASE-DAY07/DAY08 晚课

赵秀佳: 18500690315

1. 节点流和处理流的区别

- 节点流:又称为"低级流"
  - 特点:直接链接程序与另一端的"管道"，是真实读写数据的流
  - IO 一定是建立在节点流的基础上进行的。
  - 文件流就是典型的节点流(低级流)
- 处理流:又称为"高级流"
  - 特点:不能独立存在，必须链接在其他流上
  - 目的:当数据经过当前高级流时可以对数据进行某种加工操作，来简化我们的同等操作
  - 实际开发中我们经常"串联"一组高级流最终到某个低级流上，使读写数据以流水线式的加工处理完成。这一操作也被称为使"流的链接"。流链接也是 JAVA IO 的精髓所在。

2. 对象的序列化和反序列化(面试)

3.1 序列化

- 序列化定义
  - 将一个对象转换为一组**可被传输或保存**的字节。这组字节中除了包含对象本身的数据外，还会包含结构信息。
- 序列化的意义
  - 实际开发中，我们通常会将对象

        - 写入磁盘，进行长久保存
        - 在网络间两台计算机中的 java 间进行传输
    - 无论是保存在磁盘中还是传输，都需要将对象转换为字节后才可以进行。
- 序列化要求
    - 对象输出流要求写出的对象必须实现接口:java.io.Serializable
- ObjectOutputStream 对象输出流的序列化操作

```
Java
void writeObject(Object obj)
1.将给定的对象转换为一组可保存或传输的字节
2.通过其链接的流将字节写出
```

3.2 反序列化

- 反序列化定义
    - 将若干字节还原为对象
- 使用 ObjectInputStream 对象输入流

```
Java
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

```
Java
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
```

3. IO 总结

3.1 Java IO 必会概念

- java io 可以让我们用标准的读写操作来完成对不同设备的读写数据工作.
- java 将 IO 按照方向划分为输入与输出,参照点是我们写的程序.
- **输入:**用来读取数据的,是从外界到程序的方向,用于获取数据.
- **输出:**用来写出数据的,是从程序到外界的方向,用于发送数据.

Java 将 IO 比喻为"流",即:stream. 就像生活中的"电流","水流"一样,它是以同一个方向顺序移动的过程.只不过这里流动的是字节(2 进制数据).所以在 IO 中有输入流和输出流之分,我们理解他们是连接程序与另一端的"管道",用于获取或发送数据到另一端.

流的读写是顺序读写的，只能顺序向后写或向后读，不能回退。

3.2 Java 定义了字节流父类(抽象类)

- java.io.InputStream:所有字节输入流的超类,其中定义了读取数据的方法.因此将来不管读取的是什么设备(连接该设备的流)都有这些读取的方法,因此我们可以用相同的方法读取不同设备中的数据

```
Java
int read()：//读取一个字节，返回的 int 值低 8 位为读取的数据。如果返回值为整数-1 则表示读取到了流的末尾
int read(byte[] data)：//块读取，最多读取 data 数组总长度的数据并从数组第一个位置开始存入到数组中，返回值表示实际读取到的字节量，如果返回值为-1 表示本次没有读取到任何数据，是流的末尾。
```

- java.io.OutputStream:所有字节输出流的超类,其中定义了写出数据的方法.

```
Java
void write(int d)//写出一个字节，写出的是给定的 int 值对应 2 进制的低八位。
```

```
void write(byte[] data) //块写，将给定字节数组中所有字节一次性写出。
void write(byte[]data,int off,int len)//块写，将给定字节数组从下标
off 处开始的连续 len 个字节一次性写出。
```

4.3 文件流

文件流是一对低级流，用于读写文件的流。

- java.io.**FileOutputStream 文件输出流**，继承自 java.io.OutputStream
  - 常用构造器：
    - 覆盖模式：覆盖模式是指若指定的文件存在，文件流在创建时会先将该文件原内容清除。

```
Java
FileOutputStream(String pathname)//创建文件输出流用于
向指定路径表示的文件做写操作
FileOutputStream(File file)//创建文件输出流用于向
File 表示的文件做写操作。
//注:如果写出的文件不存在文件流自动创建这个文件，但是如
果该文件所在的目录不存在会抛出异常
:java.io.FileNotFoundException
```

    - 追加模式：追加模式是指若指定的文件存在，文件流会将写出的数据陆续追加到文件中。

```
Java
FileOutputStream(String pathname,boolean append) //
如果第二个参数为 true 则为追加模式，false 则为覆盖模式
FileOutputStream(File file,boolean append)：//同上
```

  - 常用方法：

```
Java
void write(int d) //向文件中写入一个字节，写入的是 int 值 2 进制
```

```
的低八位。
void write(byte[] data) //向文件中块写数据。将数组 data 中所有字节一次性写入文件。
void write(byte[] data,int off,int len)//向文件中快写数据。将数组 data 中从下标 off 开始的连续 len 个字节一次性写入文件。
```

- java.io.**FileInputStream 文件输入流**，继承自 java.io.InputStream
  - 常用构造器：

```
Java
    FileInputStream(String pathname) //创建读取指定路径下对应的文件的文件输入流，如果指定的文件不存在则会抛出异常 java.io.FileNotFoundException
    FileInputStream(File file) //创建读取 File 表示的文件的文件输入流，如果 File 表示的文件不存在则会抛出异常 java.io.IOException。
```

  - 常用方法

```
Java
int read() //从文件中读取一个字节，返回的 int 值低八位有效，如果返回的 int 值为整数-1 则表示读取到了文件末尾。
int read(byte[] data)//块读数据，从文件中一次性读取给定的 data 数组总长度的字节量并从数组第一个元素位置开始存入数组中。返回值为实际读取到的字节数。如果返回值为整数-1 则表示读取到了文件末尾。
```

4.4 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

```Java
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

```Java
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 缓冲字节输入流**，继承自 java.io.InputStream
  - 常用构造器

```Java
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

4.5 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

```
Java
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

```
Java
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)。
```

  - 注意：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

```
Java
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

```
Java
```

```
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
```

- 注意：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

4.6 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

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：

- 块写或块读文本数据加速
- 可以按行写或读字符串

PrintWriter

java.io.PrintWriter 具有自动行刷新的缓冲字符输出流

- 常用构造器

```
Java
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。

```
Java
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

```
Java
void println(String line) //按行写出一行字符串
```

- 特点

```
Java
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
```

4. 异常

4.1 自动关闭特性

```
Java
try{

}catch(IOException e){

}catch(Exception e){

}
```

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

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

例如:

```
Java
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();
       }
     }
   }
```

4.2 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:

- 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。

- 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。

```
Java
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;
    }
 }
```

调用

```
Java
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());
    }
}
```

4.3 异常分类（面试题）

Java 异常可以分为可检测异常，非检测异常：

- **可检测异常：**可检测异常经编译器验证，对于声明抛出异常的任何方法，编译器将强制执行处理或声明规则，不捕捉这个异常，编译器就通不过，不允许编译
- **非检测异常：**非检测异常不遵循处理或者声明规则。在产生此类异常时，不一定非要采取任何适当操作，编译器不会检查是否已经解决了这样一个异常
- RuntimeException 类属于非检测异常，因为普通 JVM 操作引起的运行时异常随时可能发生，此类异常一般是由特定操作引发。但这些操作在 java 应用程序中会频繁出现。因此它们不受编译器检查与处理或声明规则的限制。实际上 RuntimeException 及其子类型表达的都是因为程序漏洞(BUG),即:逻辑不严谨等原因导致的。这类异常都是通过修复代码可完全避免的异常，因此不应当由异常处理机制来处理

常见的 RuntimeException 子类

- IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数
- NullPointerException：当应用程序试图在需要对象的地方使用 null 时，抛出该异常
- ArrayIndexOutOfBoundsException：当使用的数组下标超出数组允许范围时，抛出该异常
- ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出该异常
- NumberFormatException：当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。

4.4 自定义异常（写项目常用）

自定义异常通常用来定义那些业务上的异常问题。

定义自定义异常需要注意以下问题:

- 异常的类名要做到见名知义
- 需要是 Exception 的子类
- 提供超类异常提供的所有种类构造器

5. 网络编程

5.1 网络应用模型

- **C/S：**这里的 C 是指 Client，即客户端。而 S 是指 Server，即服务端。网络上的的应用本质上就是两台计算机上的软件进行交互。而客户端和服务端就是对应的两个应用程序，即客户端应用程序和服务端应用程序
- **B/S：**这里的 B 是 Browser，即浏览器，而 S 是指 Server。浏览器是一个通用的客户端，可以与不同的服务端进行交互。但是本质上 B/S 还是 C/S 结构，只不过浏览器是一个通用的客户端而已。

5.2 可靠传输与不可靠传输

TCP 协议与 UDP 协议都是传输协议，客户端程序与服务端程序基于这些协议完成网络间的数据交互。

- TCP 是可靠传输协议，是面向连接的协议，保证数据传输中的可靠性和完整性。
  - TCP 保证可靠传输，但是传输效率低，占用带宽高。
- UDP 是不可靠传输协议，不保证数据传输的完整性。
  - UDP 不保证可靠传输，但是传输速度块，占用带宽小。

5.3 Socket 与 ServerSocket

- 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。
- **聊天室客户端**

```
Java
package cn.tedu.io;

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

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() throws IOException {
        System.out.println("开始连接");
          /*
                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("连接成功");
    }
    /**
     * 客户端开始工作的方法
     */
    public void start() {
        try {
            OutputStream os=socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(os);
            BufferedWriter bw = new  BufferedWriter(osw);
            PrintWriter pw = new PrintWriter(bw,true);
            pw.println("你瞅啥");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

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

}
```

- **聊天室服务端**

```
Java
package cn.tedu.io;
```

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

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("服务器正在启动");
                  /*
                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("服务器启动完毕");
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
    /**
     * 服务端开始工作的方法
     */
    public void start() {
        System.out.println("等待客户端连接");
        try {
            Socket socket=ServerSocket.accept();
            System.out.println("一个客户端连接完毕");
            InputStream is=socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader br = new BufferedReader(isr);
            String message=br.readLine();
            System.out.println(message);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }

}
```