# Day_8

## 泛型的使用

```
 *  1. jdk 5.0 新增的特性
 *
 *  2.在集合中使用泛型：
 *      总结：
 *      ① 集合接口或集合类在 jdk5.0 时都修改为带泛型的结构。
 *      ② 在实例化集合类时，可以指明具体的泛型类型。
 *      ③ 指明完以后，在集合类或接口中凡是定义类或接口时，内部结构使用到类的泛型的位置，都指定为实例化的泛型类型。
 *          比如：add(E e) --->  实例化以后：add(Integer e)
 *      ④ 注意点：泛型的类型必须是类，不能是基本数据类型。需要用到基本数据类型的位置，拿包装类替换。
 *      ⑤ 如果实例化时，没有指明泛型的类型。默认类型为 java.lang.Object 类型.
 *
 *  3.如何自定义泛型结构：泛型类、泛型接口; 泛型方法。见 GenericTest1.java

public class GenericTest {
    //在集合中使用泛型之前的情况
    @Test
    public void test(){
        ArrayList list = new ArrayList();
        //需求：存放学生的成绩
        list.add(78);
        list.add(76);
        list.add(89);
        list.add(88);
        //问题一：类型不安全
        list.add("Tom");
        for(Object score : list){
            //问题二：强转时，可能出现 ClassCastException
            int stuScore = (Integer) score;
            System.out.println(stuScore);
        }
    }
    //集合中使用泛型的情况:以 ArrayList 为例
    @Test
    public void test2(){
        ArrayList<Integer> list = new ArrayList<Integer>();
        list.add(78);
        list.add(87);
        //编译时就会类型检查，保证数据的安全
//        list.add("Tom");
```

```
        //方式一：
//      for(Integer score : list){
//          //避免了强转操作
//          int stuScore = score;
//          System.out.println(stuScore);//78 87
//      }
        //方式二：
        Iterator<Integer> iterator = list.iterator();
        while (iterator.hasNext()){
            int stuScore = iterator.next();
            System.out.println(stuScore);
        }
    }

    //集合中使用泛型的情况:以 HashMap 为例
    @Test
    public void test3(){
//      Map<String, Integer> map = new HashMap<String, Integer>();
        //jdk7 新特性：类型推断
        Map<String, Integer> map = new HashMap();
        map.put("Tom",87);
        map.put("Jerry",87);
        map.put("Jack",67);
//      map.put(123,"abc");
        //泛型的嵌套
        Set<Entry<String, Integer>> entry = map.entrySet();
        Iterator<Entry<String, Integer>> iterator = entry.iterator();
        while(iterator.hasNext()){
            Entry<String, Integer> e = iterator.next();
            String key = e.getKey();
            Integer value = e.getValue();
            System.out.println(key + "--->" + value);
            //Tom--->87    Jerry--->87    Jack--->67
        }
    }
}
```

**如何自定义泛型结构：泛型类、泛型接口；泛型方法。**

```
 *  1. 关于自定义泛型类、泛型接口：
 *      静态方法不能使用类的泛型
 *      异常类声明为泛型类
 *
 *      class Father<T1, T2> {
 *      }
```

```
 *          // 子类不保留父类的泛型
 *          // 1)没有类型 擦除
 *          class Son1 extends Father {// 等价于 class Son extends Father<Object,Object>{
 *          }
 *          // 2)具体类型
 *          class Son2 extends Father<Integer, String> {
 *          }
 *          // 子类保留父类的泛型
 *          // 1)全部保留
 *          class Son3<T1, T2> extends Father<T1, T2> {
 *          }
 *          // 2)部分保留
 *          class Son4<T2> extends Father<Integer, T2> {
 *          }
 *          // 子类不保留父类的泛型
 *          // 1)没有类型 擦除
 *          class Son<A, B> extends Father{//等价于 class Son extends Father<Object,Object>{
 *          }
 *          // 2)具体类型
 *          class Son2<A, B> extends Father<Integer, String> {
 *          }
 *          // 子类保留父类的泛型
 *          // 1)全部保留
 *          class Son3<T1, T2, A, B> extends Father<T1, T2> {
 *          }
 *          // 2)部分保留
 *          class Son4<T2, A, B> extends Father<Integer, T2> {
 *          }
public class GenericTest1 {
    @Test
    public void test(){
        //如果定义了泛型类，实例化没有指明类的泛型，则认为此泛型类型为 Object 类型
        //要求：如果定义了类是带泛型的，建议在实例化时要指明类的泛型
        Order order = new Order();
        order.setOrderT(123);
        order.setOrderT("ABC");
        //建议：实例化时指明类的泛型
        Order<String> order1 = new Order<>("orderAA",1001,"order:AA");
        order1.setOrderT("AA:hello");
    }
    @Test
    public void test2(){
        SubOrder sub1 = new SubOrder();
        //由于子类在继承带泛型的父类时，指明了泛型类型。则实例化子类对象时，不需
```

要再指明泛型。

```
            sub1.setOrderT(12);
            SubOrder1<String> sub2 = new SubOrder1<>();
            sub2.setOrderT("s");
        }
        @Test
        public void test3(){
            ArrayList<String> list1 = null;
            ArrayList<Integer> list2 = null;
            //泛型不同的引用不能相互赋值。
//          list1 = list2;
        }
        //测试泛型方法
        @Test
        public void test4(){
            Order<String> order = new Order<>();
            Integer[] arr = new Integer[]{1,2,3,4};
            //泛型方法在调用时，指明泛型参数的类型。
            List<Integer> list = order.copyFromArrayToList(arr);
            System.out.println(list);//[1, 2, 3, 4]
        }
}
```

**自定义的泛型类**/

```
public class Order<T>{
    String orderName;
    int orderId;
    //类的内部结构就可以使用类的泛型
    T orderT;
    public Order(){
        //编译不通过
//          T[] arr = new T[10];
        //编译通过
        T[] arr = (T[]) new Object[10];
    };
    public Order(String orderName,int orderId, T orderT){
        this.orderName = orderName;
        this.orderId = orderId;
        this.orderT = orderT;
    }

    //如下的三个方法都不是泛型方法
    public T getOrderT(){
        return orderT;
```

```
    }
    public void setOrderT(T orderT){
        this.orderT = orderT;
    }

    @Override
    public String toString() {
        return "Order{" +
                "orderName='" + orderName + '\'' +
                ", orderId=" + orderId +
                ", orderT=" + orderT +
                '}';
    }
    //泛型方法：在方法中出现了泛型的结构，泛型参数与类的泛型参数没有任何关系
    //换句话说，泛型方法所属的类是不是泛型类都没有关系
    //泛型方法，可以声明为静态的。原因：泛型参数是在调用方法时确定的。并非在实例化类时确定。
    public static <E> List<E> copyFromArrayToList(E[] arr){
        ArrayList<E> list = new ArrayList<>();
        for(E e : arr){
            list.add(e);
        }
        return list;
    }

}

public class SubOrder extends Order<Integer> {     //SubOrder:不是泛型类

    public static   <E> List<E> copyFromArrayToList(E[] arr){
        ArrayList<E> list = new ArrayList<>();
        for(E e : arr){
            list.add(e);
        }
        return list;
    }
}

public class SubOrder1<T> extends Order<T> {     //SubOrder1:仍然是泛型类

}
```

**泛型的应用**

```
* DAO：data(base) access object ： 数据访问对象*/
```

```
public class DAO <T>{ //表的共性操作的 DAO
    //添加一条记录
    public void add(T t){

    }
    //删除一条记录
    public boolean remove(int index){
        return false;
    }
    //修改一条记录
    public void update(int index,T t){

    }
    //查询一条记录
    public T getIndex(int index){
        return null;
    }
    //查询多条记录
    public List<T> getForList(int index){
        return null;
    }
    //泛型方法
    //举例：获取表中一共有多少条记录？ E->long 获取最大的员工入职时间？ E—>Date
类
    public <E> E getValue(){
        return null;
    }
}

public class Customer {   //此类对应数据库中的 customers 表
}

public class CustomerDAO extends DAO<Customer>{ //只能操作某一个表的 DAO
}

public class Student {
}

public class StudentDAO extends DAO<Student> { //只能操作某一个表的 DAO
}

public class DAOTest {
    @Test
    public void test(){
```

```
        CustomerDAO dao1 = new CustomerDAO();
        dao1.add(new Customer());
        List<Customer> list = dao1.getForList(10);

        StudentDAO dao = new StudentDAO();
        Student student = dao.getIndex(1);
    }
}
```

## 泛型在继承方面的体现通配符的使用

```
public class GenericTest {
    /*
    1.泛型在继承方面的体现
        虽然类 A 是类 B 的父类，G<A> 和 G<B>二者不具备子父类关系，二者是并列关系。
        补充：类 A 是类 B 的父类。A<G>是 B<G>的父类

     */
    @Test
    public void test(){
        Object obj = null;
        String str = null;
        obj = str;

        Object[] arr1 = null;
        String[] arr2 = null;
        arr1 = arr2;

        List<Object> list1 = null;
        List<String> list2 = new ArrayList<String>();
        //此时的 list1 和 list2 不具有子父类关系
        //编译不通过
//      list1 = list2;
        /*
        反证法：
            假设 list1 = list2;
            list1.add(123);导致混入非 String 的数据。出错。
         */
        show(list1);
        show1(list2);
    }
    public void show1(List<String> list){
    }
    public void show(List<Object> list){
```

```
    }
    @Test
    public void test2(){
        AbstractList<String> list1 = null;
        List<String> list2 = null;
        ArrayList<String> list3 = null;
        list1 = list3;
        list2 = list3;
        List<String> list4 = new ArrayList<>();
    }

    /*
     2.通配符的使用
        通配符：?
        类 A 是类 B 的父类，G<A>和 G<B>是没有关系的。二者共同的父类是：G<?>
     */
    @Test
    public void test3(){
        List<Object> list1 = null;
        List<String> list2 = null;
        List<?> list = null;
        list = list1;
        list = list2;
        //编译通过
//      print(list1);
//      print(list2);

        //
        List<String> list3 = new ArrayList<>();
        list3.add("AA");
        list3.add("BB");
        list3.add("CC");
        list = list3;
        //添加：对于 list<?>就不能向其内部添加数据
        //除了添加 null 之外
//      list.add("DD");
        list.add(null);
        //获取(读取)：允许读取数据，读取的数据类型是 Object.
        System.out.println(list.get(0));//AA
    }
    public void print(List<?> list){
        Iterator<?> iterator = list.iterator();
        while (iterator.hasNext()){
            Object obj = iterator.next();
```

```
            System.out.println(obj);
        }
    }

    /*
        3.有限制条件的通配符的使用
        ? extends A：
            G<? extends A>可以作为 G<A>和 G<B>的父类，其中 B 是 A 的子类。
        ? super A：
            G<? super A>可以作为 G<A>和 G<B>的父类，其中 B 是 A 的父类。
     */
    @Test
    public void test4(){
        List<? extends Person> list1 = null;
        List<? super Person> list2 = null;
        List<Student> list3 = new ArrayList<Student>();
        List<Person> list4 = new ArrayList<Person>();
        List<Object> list5 = new ArrayList<Object>();

        list1 = list3;
        list1 = list4;
        //不可以
//      list1 = list5;
        //不可以
//      list2 = list3;
        list2 = list4;
        list2 = list5;

        //读取数据
        list1 = list3;
        Person p = list1.get(0);
        //编译不通过
//      Student s = list1.get(0);
        list2 = list4;
        Object obj = list2.get(0);
        //编译不通过
//      Person p = list2.get(0);

        //写入数据：
        //编译不通过
//      list1.add(new Student());
        //编译通过
        list2.add(new Person());
        list2.add(new Student());
```

```
    }

}
```

```
* File 类的使用
* 1. File 类的一个对象，代表一个文件或一个文件目录(俗称：文件夹)
 * 2. File 类声明在 java.io 包下
 * 3. File 类中涉及到关于文件或文件目录的创建、删除、重命名、修改时间、文件大小等方法。
 *      并未涉及到写入或读取文件内容的操作。如果需要读取或写入文件内容，必须使用 IO 流来完成。
 * 4.后续 File 类的对象常会作为参数传递到流的构造器中，指明读取或写入的”终点“。
 * @author skoud
 * @create 2020-11-16-15:03
 */
public class FileTest {
    /*
      1.如何创建 File 类的实例
        File(String filePath)
        File(String parentPath,String childPath)
        File(File parentFile,String childPath)
      2.
        相对路径：相较于某个路径下，指明的路径。
        绝对路径：包含盘符在内的文件或文件目录的路径
      3.路径分隔符
        windows:\\
        unix:/
     */
    @Test
    public void test(){
        //构造器 1
        File file1 = new File("hello.txt");//相对于当前 module
    File file2 = new File("E:\\Develop\\Java_work\\workspace_idea\\JavaSenior\\day_8\\he.txt");
        System.out.println(file1);//hello.txt

    System.out.println(file2);//E:\Develop\Java_work\workspace_idea\JavaSenior\day_8\he.txt
        //构造器 2
```

```
        File file3 = new File("E:\\Develop","Java_work");
        System.out.println(file3);//E:\Develop\Java_work
        //构造器 3
        File file4 = new File(file3,"hi.txt");
        System.out.println(file4);//E:\Develop\Java_work\hi.txt
    }
    /*
          public String getAbsolutePath()：获取绝对路径
          public String getPath() ： 获取路径
          public String getName() ： 获取名称
          public String getParent()：获取上层文件目录路径。若无，返回 null
          public long length() ： 获取文件长度（即：字节数）。不能获取目录的长度。
          public long lastModified() ： 获取最后一次的修改时间，毫秒值
        如下的两个方法适用于文件目录
          public String[] list() ： 获取指定目录下的所有文件或者文件目录的名称数组
          public File[] listFiles() ： 获取指定目录下的所有文件或者文件目录的 File 数组
     */
    @Test
    public void test1(){
        File file1 = new File("hello.txt");
        File file2 = new File("E:\\Develop\\Java_work\\io\\hi.txt");

    System.out.println(file1.getAbsoluteFile());//E:\Develop\Java_work\workspace_idea\JavaSen
ior\day_8\hello.txt
        System.out.println(file1.getPath());//hello.txt
        System.out.println(file1.getName());//hello.txt
        System.out.println(file1.getParent());//null
        System.out.println(file1.length());//0 ->10(helloworld)
        System.out.println(new Date(file1.lastModified()));//0 -> Mon Nov 16 15:45:17
GMT+08:00 2020
        System.out.println();
        System.out.println(file2.getAbsoluteFile());//E:\Develop\Java_work\io\hi.txt
        System.out.println(file2.getPath());//E:\Develop\Java_work\io\hi.txt
        System.out.println(file2.getName());//hi.txt
        System.out.println(file2.getParent());//E:\Develop\Java_work\io
        System.out.println(file2.length());//0
        System.out.println(file2.lastModified());//0
    }
    @Test
    public void test2(){
        File file = new File("E:\\Develop\\Java_work\\workspace_idea\\JavaSenior");
        String[] list = file.list();
        for(String s : list){
            System.out.println(s);
```

```
            //.idea、day_1、day_2、day_3、day_4、day_5、day_6、day_7、day_8、JavaSenior.iml、
jdbc.properties、myproject03、out、src
        }
        File[] files = file.listFiles();
        for(File f : files){
            System.out.println(f);
            //E:\Develop\Java_work\workspace_idea\JavaSenior\.idea
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_1
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_2
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_3
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_4
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_5
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_6
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_7
            //E:\Develop\Java_work\workspace_idea\JavaSenior\day_8
            //E:\Develop\Java_work\workspace_idea\JavaSenior\JavaSenior.iml
            //E:\Develop\Java_work\workspace_idea\JavaSenior\jdbc.properties
            //E:\Develop\Java_work\workspace_idea\JavaSenior\myproject03
            //E:\Develop\Java_work\workspace_idea\JavaSenior\out
            //E:\Develop\Java_work\workspace_idea\JavaSenior\src
        }
    }
    /*
        public boolean renameTo(File dest):把文件重命名为指定的文件路径
        比如：file1.renameTo(file2)为例：
            要想保证返回 true，需要 file1 在硬盘中是存在的，且 file2 不能在硬盘中存在。
     */
    @Test
    public void test3(){
        File file1 = new File("hello.txt");//在 day_8 下
        File file2 = new File("E:\\Develop\\Java_work\\io\\hi.txt");
        boolean renameTo = file1.renameTo(file2);
        System.out.println(renameTo);//把文件 file1 重命名剪切到 file2
    }
    /*
          public boolean isDirectory()：判断是否是文件目录
          public boolean isFile() ：判断是否是文件
          public boolean exists() ：判断是否存在
          public boolean canRead() ：判断是否可读
          public boolean canWrite() ：判断是否可写
          public boolean isHidden() ：判断是否隐藏
     */
    @Test
    public void test4(){
```

```
        File file1 = new File("hello1.txt"); //hello(有) -> hello1(无)
        System.out.println(file1.isDirectory());//false -> false
        System.out.println(file1.isFile());//true -> false
        System.out.println(file1.exists());//true -> false
        System.out.println(file1.canRead());//true -> false
        System.out.println(file1.canWrite());//true -> false
        System.out.println(file1.isHidden());//false -> false
        File file2 = new File("E:\\Develop\\Java_work\\io1");//io(有) -> io1(无)
        System.out.println(file2.isDirectory());//true -> false
        System.out.println(file2.isFile());//false -> false
        System.out.println(file2.exists());//true -> false
        System.out.println(file2.canRead());//true -> false
        System.out.println(file2.canWrite());//true -> false
        System.out.println(file2.isHidden());//false -> false
    }
```

创建硬盘对应的文件或文件目录

public boolean createNewFile() ：创建文件。若文件存在，则不创建，返回 false

public boolean mkdir() ：创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。

public boolean mkdirs() ：创建文件目录。如果上层文件目录不存在，一并创建删除磁盘中的文件或文件目录

public boolean delete()：删除文件或者文件夹

删除注意事项

Java 中的删除不走回收站。

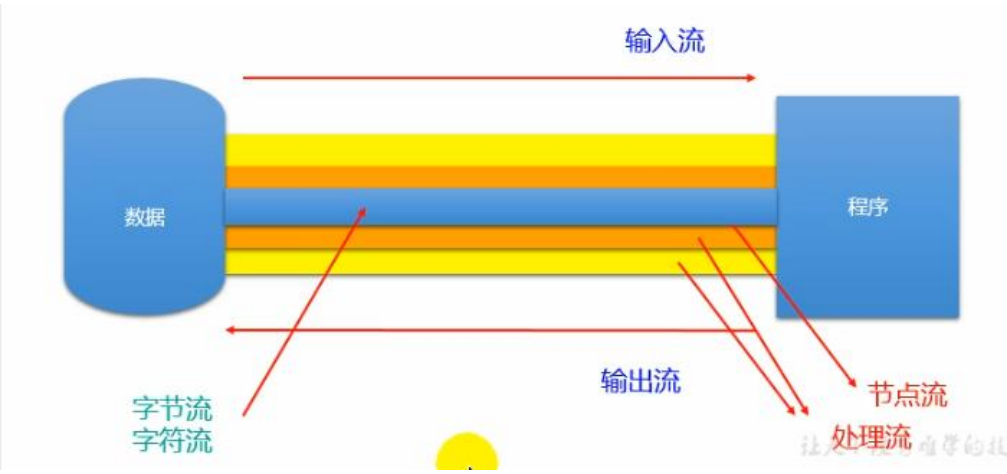
```
    @Test
    public void test5() throws IOException {
        //文件的创建
        File file = new File("hi.txt");
        if(!file.exists()){
            file.createNewFile();
            System.out.println("创建成功");
        }else{
            file.delete();
            System.out.println("删除成功");
        }
    }
    @Test
    public void test6(){
        //文件目录的创建
        File file1 = new File("E:\\Develop\\Java_work\\io\\io1\\io2");
        boolean mkdir = file1.mkdir();
        if(mkdir){
            System.out.println("创建成功 1！ ");
```

```
            }else{
                System.out.println("创建失败 1！");
            }
            File file2 = new File("E:\\Develop\\Java_work\\io\\io1\\io3");
            boolean mkdir1 = file2.mkdirs();
            if(mkdir1){
                System.out.println("创建成功 2！");
            }else{
                System.out.println("创建失败 2！");
            }
            //要想删除成功，io4 文件目录下不能有子目录或文件
            File file3 = new File("E:\\Develop\\Java_work\\io\\io1\\io4");
            System.out.println(file3.delete());
        }
}
```

## Day_9

# IO 流体系

| 分类 | 字节输入流 | 字节输出流 | 字符输入流 | 字符输出流 |
|---|---|---|---|---|
| 抽象基类 | InputStream | OutputStream | Reader | Writer |
| 访问文件 | FileInputStream | FileOutputStream | FileReader | FileWriter |
| 访问数组 | ByteArrayInputStream | ByteArrayOutputStream | CharArrayReader | CharArrayWriter |
| 访问管道 | PipedInputStream | PipedOutputStream | PipedReader | PipedWriter |
| 访问字符串 | | | StringReader | StringWriter |
| 缓冲流 | BufferedInputStream | BufferedOuputStream | BufferedReader | BufferedWriter |
| 转换流 | | | InputStreamReader | OutputStreamWriter |
| 对象流 | ObjectInputStream | ObjectOutputStream | | |
| | FilterInputStream | FilterOutputStream | FilterReader | FilterWriter |
| 打印流 | | PrintStream | | PrintWriter |
| 推回输入流 | PushbackInputStream | | PushbackReader | |
| 特殊流 | DataInputStream | DataOutputStream | | |



| (抽象基类) | 字节流 | 字符流 |
| --- | --- | --- |
| 输入流 | InputStream | Reader |
| 输出流 | OutputStream | Writer |

* 一、流的分类：
*  1.操作数据单位：字节流、字符流
*  2.数据的流向：输入流、输出流。
*  3.流的角色：节点流、处理流
*
*  二、流的体系结构

| 抽象基类 | 节点流(或文件流) | 缓冲流（处理流的一种） |
| --- | --- | --- |
| InputStream | FileInputStream<br>(read(byte[] buffer)) | BufferedInputStream<br>(read(byte[] buffer)) |
| OutputStream | FileOutputStream<br>(write(byte[] buffer,0,len)) | BufferedOutputStream<br>(write(byte[] buffer,0,len)) |
| Reader | FileReader<br>(read(char[] cbuf)) | BufferedReader<br>(read(char[] cbuf) /<br>readLine()) |
| Writer | FileWriter<br>(write(char[] cbuf,0,len)) | BufferedWriter<br>(write(char[] cbuf,0,len)) /<br>flush() |

```
public class FileReaderWriterTest {
    public static void main(String[] args) {
        File file = new File("hello.txt");//相较于当前工程
    System.out.println(file.getAbsolutePath());//E:\Develop\Java_work\workspace_idea\JavaSenior\hello.txt
        File file1 = new File("day_9\\hello.txt");
    System.out.println(file1.getAbsolutePath());//E:\Develop\Java_work\workspace_idea\JavaSenior\day_9\hello.txt
    }
    /*
        将 day_9 下的 hello.txt 文件内容读入程序中，并输出到控制台
        说明点：
        1.read():返回读入的一个字符。如果达到文件末尾，返回-1
        2.异常的处理：为了保证流资源一定可以执行关闭操作，需要使用 try-catch-finally 处理
        3.读入的文件一定要存在，否则就会报 FileNotFoundException。
     */
    @Test
    public void testFileReader(){
        FileReader fr = null;
```

```
        try {
            //1.实例化 File 类的对象，指明要操作的文件
            File file = new File("hello.txt");//相较于当前 Module
            //2.提供具体的流
            fr = new FileReader(file);
            //3.数据的读入
            //read():返回读入的一个字符。如果达到文件末尾，返回-1
            //方式一：
//          int data = fr.read();
//          while(data != -1){
//              System.out.print((char)data);
//              data = fr.read();//(自动读下一个)h e l l o w o r l d
//          }
            //方式二：
            int data;
            while((data = fr.read()) != -1){
                System.out.print((char)data); //h e l l o w o r l d
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //4.流的关闭操作
            try {
                if(fr != null)//防止空指针异常
                {
                    fr.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    //对 read()操作升级，使用 read 的重载方法
    @Test
    public void testFileReader1(){
        FileReader fr = null;
        try {
            //1.File 类的实例化
            File file = new File("hello.txt");
            //2.FileReader 流的实例化
            fr = new FileReader(file);
            //3.读入的操作
            //read(char[] cbub)：返回每次读入 cbuf 数组中的字符的个数。如果达到文件末
尾，返回-1
```

```
            char[] cbuf = new char[5];
            int len;
            while((len =fr.read(cbuf)) != -1){
                //方式一
                //错误的写法
//                for (int i = 0; i < cbuf.length; i++) {
//                    System.out.println(cbuf[i]);
//                }
                //正确的写法
//                for (int i = 0; i < len; i++) {
//                    System.out.print(cbuf[i]);//h e l l o w o r l d
//                }
                //方式二
                //错误的写法，对应着方式一的错误写法
//                String str = new String(cbuf);
//                System.out.println(str);
                //正确的写法
                String str = new String(cbuf,0,len);
                System.out.print(str);//h e l l o w o r l d
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //4.资源的关闭
            try {
                if(fr != null) {
                    fr.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
    /*
        从内存中写出数据到硬盘的文件里。
        说明：
         1.输出操作，对应的 File 可以不存在。并不会报异常。
         2.File 对应的硬盘的文件如果不存在，在输出的过程中，会自动创建此文件。
           File 对应的硬盘的文件如果存在：
            如果流使用的构造器是：FileWriter(file,false) / FileWriter(file):对原有文件的覆盖
            如果流使用的构造器是：FileWriter(file,true):不会对原有文件覆盖，而是在原有文件基础上追加内容
     */
    @Test
```

```
public void testFileWrite(){
    FileWriter fw = null;//true:追加  false:覆盖
    try {
        //1.提供 File 类的对象，指明写出到的文件
        File file = new File("hello1.txt");
        //2.提供 FileWriter 的对象，用于数据的写出
        fw = new FileWriter(file,true);
        //3.写出的操作
        fw.write("I have a dream!\n");
        fw.write("you need to have a dream!");
        //4.流资源的关闭
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(fw != null) {
            try {
                fw.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
@Test
public void testFileReaderFileWriter(){
    FileReader fr = null;
    FileWriter fw = null;
    try {
        //1.创建 File 类的对象，指明读入喝写出的文件
        File srcFile = new File("hello.txt");
        File destFile = new File("hello2.txt");
        //不能用字符流来处理图片等字节数据
        //2.创建输入流和输出流的对象
        fr = new FileReader(srcFile);
        fw = new FileWriter(destFile);
        //3.数据的读入和写出操作
        char[] cbuf = new char[5];
        int len;//记录每次读入到 cbuf 数组中的字符的个数
        while ((len = fr.read(cbuf)) != -1){
            fw.write(cbuf,0,len);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
```

```
                //4.关闭流资源
                //方式一
//              try {
//                  if(fw != null)
//                      fw.close();
//              }catch (IOException e){
//                  e.printStackTrace();
//              }finally {
//                  try{
//                      if (fr != null)
//                          fr.close();
//                  }catch (IOException e){
//                      e.printStackTrace();
//                  }
//              }
                //方式二
                try {
                    if(fw != null)
                        fw.close();
                }catch (IOException e) {
                    e.printStackTrace();
                }
                try{
                    if (fr != null)
                        fr.close();
                }catch (IOException e){
                    e.printStackTrace();
                }
            }
        }
}
```

**测试 FileInputOutputStream 喝 FileOutputStream 的使用**

```
 * 结论：
 * 1. 对于文本文件(.txt,.java,.c,.cpp)，使用字符流处理
        纯复制也可以用字节流处理
 * 2. 对于非文本文件(.jpg,.mp3,.mp4,.avi,.doc,.ppt,...)，使用字节流处理

public class FileInputOutputStreamTest {
    //使用字节流 FileInputStream 处理文本文件，可能出现乱码。
    @Test
    public void testFileInputStream(){
        FileInputStream fis = null;
        try {
```

```
            //1.造文件
            File file = new File("hello.txt");
            //2.造流
            fis = new FileInputStream(file);
            //3.读数据
            byte[] buffer = new byte[5];
            int len;//记录每次读取的字节的个数
            while((len = fis.read(buffer)) != -1){
                String str = new String(buffer,0,len);
                System.out.print(str);//helloworld,一个字节可以显示英文字母和数字，当有
中文时出现乱码
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //4.关闭资源
            if(fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    //实现对图片的复制操作
    @Test
    public void testFileInputOutputStream(){
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            File srcFile = new File("Love.jpg");
            File destFile = new File("Love2.jpg");
            fis = new FileInputStream(srcFile);
            fos = new FileOutputStream(destFile);
            byte[] buffer = new byte[5];
            int len;
            while((len = fis.read(buffer)) != -1){
                fos.write(buffer,0,len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(fis != null) {
```

```
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    //指定路径下文件的复制
    public void copyFile(String srcPath,String destPath){
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            File srcFile = new File(srcPath);
            File destFile = new File(destPath);
            fis = new FileInputStream(srcFile);
            fos = new FileOutputStream(destFile);
            byte[] buffer = new byte[1024];
            int len;
            while((len = fis.read(buffer)) != -1){
                fos.write(buffer,0,len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
```

```
                }
            }
        }
    }
    @Test
    public void testCopyFile(){
        long start = System.currentTimeMillis();
        //内存层面读会有乱码，文件里读无乱码
        String srcPath = "hello.txt";
        String destPath = "hello3.txt";
//      String srcPath = "假装待复制文件";
//      String destPath = "假装已复制文件";
        copyFile(srcPath,destPath);
        long end = System.currentTimeMillis();
        System.out.println("复制花费时间为：" +(end - start));
    }
}
```

**处理流之一：缓冲流的使用**

```
 * 1.缓冲流
 *   BufferedInputStream
 *   BufferedOutputStream
 *   BufferedReader
 *   BufferedWriter
 *
 * 2.作用：提供流的读取、写入的速度
 *       提高读写速度的原因：内部提供了一个缓冲区
 * 3。处理流，就是“套接”在已有的流的基础上。

public class BufferTest {
    /*
     实现非文本文件的复制
     */
    @Test
    public void BufferedStreamTest(){
        FileInputStream fis = null;
        FileOutputStream fos = null;
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
        try {
            //1.造文件
            File srcFile = new File("Love");
            File destFile = new File("Love1");
```

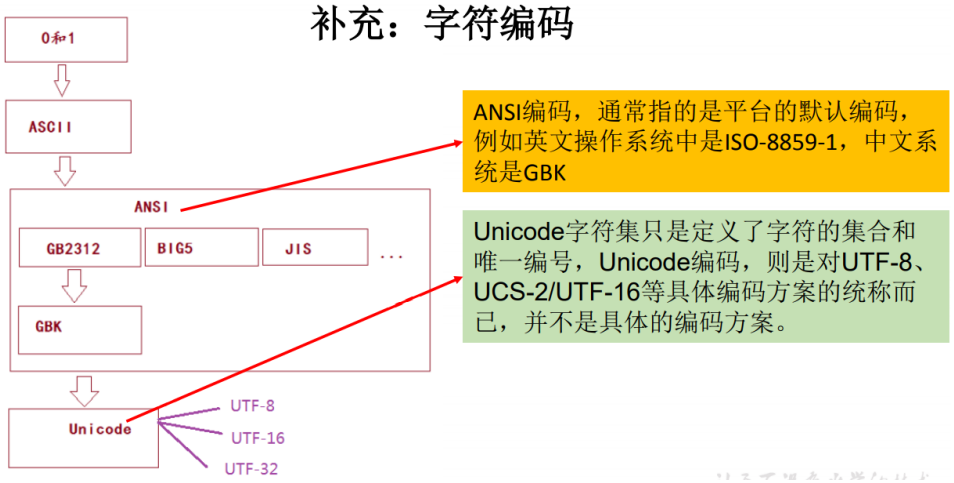
```
            //2.造流
            //2.1 造节点流
            fis = new FileInputStream(srcFile);
            fos = new FileOutputStream(destFile);
            //2.2 造缓冲流
            bis = new BufferedInputStream(fis);
            bos = new BufferedOutputStream(fos);

            //3.具体的复制细节：读取、写入的过程
            byte[] buffer = new byte[10];
            int len;
            while((len = bis.read(buffer)) != -1){
                bos.write(buffer,0,len);
//                  bos.flush();//刷新缓冲区
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //4.资源关闭
            //要求：先关闭外层的流，再关闭内层的流
            if(bis != null) {
                try {
                    bis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(bos != null) {
                try {
                    bos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            //说明：在关闭外层流的同时，内层流也会自动的关闭。关于内层流的关闭，
我们可以省略。
//              fis.close();
//              fos.close();
        }
    }
    //实现文件复制的方法
    public void copyFileWithBuffered(String srcPath,String destPath){
        BufferedInputStream bis = null;
        BufferedOutputStream bos = null;
```

```
        try {
            //1.造文件
            FileInputStream fis = null;
            FileOutputStream fos = null;
            File srcFile = new File(srcPath);
            File destFile = new File(destPath);

            //2.造流
            //2.1 造节点流
            fis = new FileInputStream(srcFile);
            fos = new FileOutputStream(destFile);
            //2.2 造缓冲流
            bis = new BufferedInputStream(fis);
            bos = new BufferedOutputStream(fos);

            //3.具体的复制细节：读取、写入的过程
            byte[] buffer = new byte[1024];
            int len;
            while((len = bis.read(buffer)) != -1){
                bos.write(buffer,0,len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //4.资源关闭
            //要求：先关闭外层的流，再关闭内层的流
            if(bis != null) {
                try {
                    bis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(bos != null) {
                try {
                    bos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            //说明：在关闭外层流的同时，内层流也会自动的关闭。关于内层流的关闭，
我们可以省略。
//          fis.close();
//          fos.close();
```

```
        }
    }
    @Test
    public void testCopyFileWithBuffered(){
        long start = System.currentTimeMillis();
        //内存层面读会有乱码，文件里读无乱码
        String srcPath = "假装待复制文件";
        String destPath = "假装已复制文件";
        copyFileWithBuffered(srcPath,destPath);
        long end = System.currentTimeMillis();
        System.out.println("复制花费时间为：" +(end - start));//618 -> 176
    }
    /*
        使用 BufferedReader 和 BufferedWriter 实现文本文件的复制
     */
    @Test
    public void testBufferedReaderBufferedWriter(){
        BufferedReader br = null;
        BufferedWriter bw = null;
        try {
            //创建文件和相应的流
            br = new BufferedReader(new FileReader(new File("dbcp.txt")));
            bw = new BufferedWriter(new FileWriter(new File("dbcp1.txt")));
            //读写操作
            //方式一：使用 char[]数组
//            char[] cbuf = new char[1024];
//            int len;
//            while((len = br.read(cbuf)) != -1){
//                bw.write(cbuf,0,len);
//  //              bw.flush();
//            }
            //方式二：使用 String
            String data;
            while ((data = br.readLine()) != null){
                //方法一：
//                bw.write(data + "\n");//data 中不包含换行符
                //方法二：
                bw.write(data);//data 中不包含换行符
                bw.newLine();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //关闭资源
```

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


## 补充：字符编码




**处理流之二：转换流的使用**

* 1.转换流：属于字符流
*  InputStreamReader ：将一个字节的输入流转换为字符的输入流

```
 *          OutputStreamWriter ：将一个字符的输出流转换为字节的输出流
 *
 * 2.作用：提供字节流与字符流之间的转换
 *
 * 3.解码：字节、字节数组  ---> 字符数组、字符串
 *    编码：字符数组、字符串 ---> 字节、字节数组
 *
 * 4.字符集
 *   ASCII：美国标准信息交换码。用一个字节的 7 位可以表示。
 *   ISO8859-1：拉丁码表。欧洲码表，用一个字节的 8 位表示。
 *   GB2312：中国的中文编码表。最多两个字节编码所有字符
 *   GBK：中国的中文编码表升级，融合了更多的中文文字符号。最多两个字节编码
 *   Unicode：国际标准码，融合了目前人类使用的所有字符。为每个字符分配唯一的字符码。所有的文字都用两个字节来表示。
 *   UTF-8：变长的编码方式，可用 1-4 个字节来表示一个字符

public class InputStreamReaderTest {
    /*
        此时处理异常的话，仍然应该使用 try-catch-finally
        InputStreamReader 的使用：实现字节的输入流到字符的输入流的转换
     */
    @Test
    public void test1() throws IOException {
        FileInputStream fis = new FileInputStream("dbcp.txt");
//        InputStreamReader isr = new InputStreamReader(fis);//使用系统默认字符集
        //参数 2 指明了字符集，具体使用哪个字符集，取决于文件的 dbcp.txt 保存时使用的字符集
        InputStreamReader isr = new InputStreamReader(fis,"UTF-8");//gbk 时乱码
        char[] cbuf =    new char[20];
        int len;
        while((len = isr.read(cbuf)) != -1){
            String str = new String(cbuf,0,len);
            System.out.print(str);//无乱码输出
        }
        isr.close();
    }
    /*
        此时处理异常的话，仍然应该使用 try-catch-finally
        综合使用 InputStreamReader 和 OutputStreanWriter
     */
    @Test
    public void test2() throws IOException {
        //造文件、造流
        File file1 = new File("dbcp.txt");
```

```
        File file2 = new File("dbcp_gbk.txt");
        FileInputStream fis = new FileInputStream(file1);
        FileOutputStream fos = new FileOutputStream(file2);
        InputStreamReader isr = new InputStreamReader(fis,"utf-8");
        OutputStreamWriter osw = new OutputStreamWriter(fos,"gbk");
        //读写过程
        char[] cbuf = new char[20];
        int len;
        while((len = isr.read(cbuf)) != -1){
            osw.write(cbuf,0,len);
        }
        //关闭资源
        isr.close();
        osw.close();
    }
```

**其他流的使用**

```
 *  1.标准的输入、输出流
 *  2.打印流
 *  3.数据流

public class OtherStreamTest {
    /*
      1.标准的输入、输出流
       1.1
        System.in:标准的输入流，默认从键盘输入
        System.out:标准的输出流，默认从控制台输出
       1.2
        System 类的 setIn(InputStream is)/setOut(PrintStream ps)方式重新指定输入和输出的
流
       1.3 练习：
            从键盘输入字符串，要求将读取到的整行字符串转成大写输出。然后继续
            进行输入操作，直至当输入“e”或者“exit”时，退出程序。
         方法一：使用 Scanner 实现，调用 next()返回一个字符串
         方法二：使用 System.in 实现。System.in(字节流)---> 转换流 --->（字符流）
BufferedReader 的 readLine().

    public static void main (String[] args) {
        BufferedReader br = null;
        try {
            InputStreamReader isr = new InputStreamReader(System.in);//转成字符流
            br = new BufferedReader(isr);
            while (true) {
                System.out.println("请输入字符串：");
                String data = br.readLine();
```

```
                if ("e".equalsIgnoreCase(data) || "exit".equalsIgnoreCase(data)) {
                    System.out.println("程序结束！");
                    break;
                }
                String upperCase = data.toUpperCase();
                System.out.println(upperCase);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    /*
        2.打印流：PrintStream 和 PrintWriter
          2.1 提供了一系列重载的 print()和 println()
     */
    @Test
    public void test2(){
        PrintStream ps = null;
        try {
            FileOutputStream fos = new FileOutputStream(new File("D:\\IO\\text.txt"));
            // 创建打印输出流,设置为自动刷新模式(写入换行符或字节 '\n' 时都会刷新
输出缓冲区)
            ps = new PrintStream(fos, true);
            if (ps != null) {       // 把标准输出流(控制台输出)改成文件
                System.setOut(ps);
            }
            for (int i = 0; i <= 255; i++) { // 输出 ASCII 字符
                System.out.print((char) i);
                if (i % 50 == 0) { // 每 50 个数据一行
                    System.out.println(); // 换行
                }
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            if (ps != null) {
```

```
                    ps.close();
                }
            }
        }
        /*
            3.数据流
            3.1 DataInputStream 和 DataOutputStream
            3.2 作用：用于读取或写出基本数据类型的变量或字符串
            练习：将内存中的字符串、基本数据类型的变量写到文件中。
            注意：处理异常的话，仍然应该使用 try-catch-finally.
         */
        @Test
        public void test3() throws IOException {
            //1.
            DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"));
            //2.
            dos.writeUTF("宋万达");
            dos.flush();//刷新操作，将内存中的数据写入文件。
            dos.writeInt(23);
            dos.flush();
            dos.writeBoolean(true);
            dos.flush();
            //3/
            dos.close();
        }
        /*
            将文件中存储的基本数据类型变量和字符串读取到内存中，保存在变量中。
            注意点：读取不同类型的数据的顺序要与当初写入文件时，保存的数据顺序一致。
         */
        @Test
        public void test4() throws IOException {
            //1.
            DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"));
            //2.
            String name = dis.readUTF();
            int age = dis.readInt();
            boolean isMale = dis.readBoolean();
            System.out.println("name = " + name + ",age = " + age + ",isMale = " + isMale);
            //3.
            dis.close();
        }
    }
```

# Day_10

## 对象流的使用

* 1。ObjectInputStream 和 ObjectOutputStream
* 2.作用：用于存储和读取基本数据类型数据或对象的处理流。它的强大之处就是可以把 Java 中的对象写入到数据源中，也能把对象从数据源中还原回来。
* 3。要想一个 java 对象是可序列化的，需要满足相应的要求。见 Person.java
* 4.序列化机制：
*  对象序列化机制允许把内存中的 Java 对象转换成平台无关的二进制流，从而允许把这种二进制流持久地保存在磁盘上，或通过网络将这种二进制流传输到另一个网络节点。当其它程序获取了这种二进制流，就可以恢复成原来的 Java 对象
*

```
public class ObjectInputOutputStreamTest {
    /*
      序列化过程：将内存中的 java 对象保存到磁盘中或通过网络传输出去
        使用 ObjectOutputStream 实现
     */
    @Test
    public void testObjectOutputStream(){
        ObjectOutputStream oos = null;
        try {
            //1.
            oos = new ObjectOutputStream(new FileOutputStream("object.dat"));//目前 IDEA 版本 txt 不可以
            //2.
            oos.writeObject(new String("我爱学习"));//生成.dat 格式文件
            oos.flush();
            //自定义类的序列化
            oos.writeObject(new Person("宋万达",23));
            oos.flush();

            oos.writeObject(new Person("宋千达",23,1001,new Account(1)));
            //Account 没有序列化 -> 改成 Serializable 的
            oos.flush();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //3.
            if(oos != null) {
                try {
                    oos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
```

```
            }
        }
    }
    /*
        反序列化：将磁盘文件中的对象还原为内存中的一个 java 对象
         使用 ObjectInputStream 实现
     */
    @Test
    public void testObjectInputStream(){
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream("object.dat"));
            Object obj = ois.readObject();//读取对象
            String str = (String) obj;
            System.out.println(str);//我爱学习
            Object obj2 = ois.readObject();
            Person person = (Person) obj2;          //static 修饰后
            System.out.println(person);//Person{name='宋万达' -> 'null', age=23 -> 0, id=0,
account=null}
            Object obj3 = ois.readObject();
            Person person1 = (Person) obj3;          //transient 修饰后
            System.out.println(person1);//Person{name='宋千达' -> 'null', age=23 -> 0, id=1001,
account=Account{balance=1.0}}
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            if(ois != null) {
                try {
                    ois.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

**Person 需要满足如下的要求，方可序列化**

```
 * 1.需要实现接口：Serializable
 * 2.当前类提供一个全局常量：serialVersionUID
 * 3.除了当前 Person 类需要实现 Serializable 接口之外，还必须保证其内部所有属性也必须
是可序列化的。
 *         （默认情况下，基本数据类型是可序列化的）
```

```
 *  补充：ObjectOutputStream 和 ObjectInputStream 不能序列化 static 和 transient 修饰的成员变量
 *
public class Person implements Serializable {
    public static final long serialVersionUID = 12345L;
    private static String name;//不让序列化
    private transient int age;//不让序列化
    private int id;
    private Account account;

    public Person(String name, int age, int id,Account account) {
        this.name = name;
        this.age = age;
        this.id = id;
        this.account = account;
    }
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", id=" + id +
                ", account=" + account +
                '}';
    }
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public Person() {
    }
}
```

```
class Account implements Serializable{
    public static final long serialVersionUID = 123456L;
    private double balance;
    public void setBalance(double balance) {
        this.balance = balance;
    }
    public double getBalance() {
        return balance;
    }
    public Account(double balance) {
        this.balance = balance;
    }
    @Override
    public String toString() {
        return "Account{" +
                "balance=" + balance +
                '}';
    }
}
```

**RandomAccessFile 的使用**

```
 * 1.RandomAccessFile 直接继承于 java.lang.Object 类，实现了 DataInput 和 DataOutput 接口
 *   2.RandomAccessFile 既可以作为一个输入流，又可以作为一个输出流
 *   3.如果 RandomAccessFile 作为输出流时，写出到的文件如果不存在，则在执行过程中自动创建。如果写出到的文件存在，则会对原有文件内容进行覆盖。（默认从头覆盖）
 *   4.可以通过相关的操作，实现 RandomAccessFile“插入”数据的效果
 *
public class RandomAccessFileTest {
    @Test
    public void test1() throws FileNotFoundException {
        RandomAccessFile raf1 = null;
        RandomAccessFile raf2 = null;
        try {
            //1.
            raf1 = new RandomAccessFile(new File("Love.jpg"),"r");
            raf2 = new RandomAccessFile(new File("Love.jpg"),"rw");
            //2.
            byte[] buffer = new byte[1024];
            int len;
            while((len = raf1.read(buffer)) != -1){
                raf2.write(buffer,0,len);
            }
        } catch (IOException e) {
            e.printStackTrace();
```

```
        } finally {
            //3.
            if(raf1 != null) {
                try {
                    raf1.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(raf2 != null) {
                try {
                    raf2.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    @Test
    public void test2(){
        RandomAccessFile raf1 = null;
        try {
            raf1 = new RandomAccessFile("hello.txt","rw");//abcdefghijklmn -> xyzdefghijklmn
            raf1.seek(3);//将指针调到角标为 3 的位置
            raf1.write("xyz".getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(raf1 != null) {
                try {
                    raf1.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    /*
        使用 RandomAccessFile 实现数据的插入效果
        abcdefghijklmn ——> abcxyzdefghijklmn
     */
    @Test
    public void test3(){
```

```
        RandomAccessFile raf = null;
        try {
            raf = new RandomAccessFile("hello.txt", "rw");
            raf.seek(3);
            //保存指针 3 后面的所有数据到 StringBuilder 中
            StringBuilder builder = new StringBuilder((int)new File("hello.txt").length());
            byte[] buffer = new byte[20];
            int len;
            while((len = raf.read(buffer)) != -1){
                builder.append(new String(buffer, 0,len));
            }
            //调回指针，写入"xyz"
            raf.seek(3);
            raf.write("xyz".getBytes());
            //将 StringBuilder 中的数据写入到文件里
            raf.write(builder.toString().getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(raf != null) {
                try {
                    raf.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //思考：将 StringBuilder 替换为 ByteArrayOutputStream
    }
}
```

| OSI参考模型 | TCP/IP参考模型 | TCP/IP参考模型各层对应协议 |
|---|---|---|
| 应用层 | 应用层 | HTTP、FTP、Telnet、DNS… |
| 表示层 | | |
| 会话层 | | |
| 传输层 | 传输层 | TCP、UDP、… |
| 网络层 | 网络层 | IP、ICMP、ARP… |
| 数据链路层 | 物理+数据链路层 | Link |
| 物理层 | | |

一、网络编程中有两个主要的问题：
 *  1.如何准确地定位网络上一台或多台主机；定位主机上的特定的应用
 *  2.找到主机后如何可靠高效地进行数据传输
 *
 * 二、网络编程中的两个要素：
 *  1.对应问题一：IP 和端口号
 *  2.对应问题二：提供网络通信协议：TCP/IP 参考模型（应用层、传输层、网络层、物理+数据链路层）
 *
 * 三、通信要素一：IP 和端口号
 * 1.IP：唯一的标识 Internet 上的计算机（通信实体）
 * 2.在 Java 中使用 InetAddress 类代表 IP
 * 3.IP 分类：IPv4 和 IPv6；万维网 和 局域网
 * 4.域名： www.baidu.com www.mi.com www.sina.com
 * 5.本地回路地址:127.0.0.1 对应着：localhost
 * 6.如何实例化 InetAddress：两个方法：getByName(String host) 、 getLocalHost()
 *  两个常用方法：getHostName() / getHostAddress()
 * 7.端口号：正在计算机上运行的进程（程序）
 *  要求：不同的进程有不同的端口号
 *  范围：被规定为一个 16 位的整数 0~65535
 * 8. 端口号与 IP 地址的组合得出一个网络套接字：Socket。
 *

```
public class InetAddressTest {
    public static void main(String[] args) {
        try {
            //File file = new File("hello.txt");
            InetAddress inet1 = InetAddress.getByName("192.168.10.14");
            System.out.println(inet1);// /192.168.10.14
            InetAddress inet2 = InetAddress.getByName("www.3.cn");
            System.out.println(inet2);//www.3.cn/115.231.142.3(DNS 解析完发回)
            InetAddress inet3 = InetAddress.getByName("127.0.0.1");
            System.out.println(inet3);// /127.0.0.1
            //获取本地 ip
            InetAddress inet4 = InetAddress.getLocalHost();
            System.out.println(inet4);// MSI/10.136.157.169
            //getHostName
            System.out.println(inet2.getHostName());//www.3.cn
            //getHostAddress
            System.out.println(inet2.getHostAddress());//115.231.142.3
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

- **TCP协议：**
  - 使用TCP协议前，须先建立TCP连接，形成传输数据通道
  - 传输前，采用“**三次握手**”方式，点对点通信，是可靠的
  - TCP协议进行通信的两个应用进程：客户端、服务端。
  - 在连接中可进行大数据量的传输
  - 传输完毕，需释放已建立的连接，效率低
- **UDP协议：**
  - 将数据、源、目的封装成数据包，不需要建立连接
  - 每个数据报的大小限制在64K内
  - 发送不管对方是否准备好，接收方收到也不确认，故是不可靠的
  - 可以广播发送
  - 发送数据结束时无需释放资源，开销小，速度快

**实现 TCP 的网络编程**

* 例子 1：客户端发送信息给服务端，服务端将数据显示在控制台

```
public class TCPTest1 {
    //客户端
    @Test
    public void client(){
        Socket socket = null;
        OutputStream os = null;
        try {
            //1.创建 Socket 对象，指明服务器端的 ip 和端口号
            InetAddress inet = InetAddress.getByName("192.168.0.120");
            socket = new Socket(inet,8899);
            //2.获取一个输出流，用于输出数据
            os = socket.getOutputStream();
            //3.写出数据的操作
            os.write("宋万达".getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //4.资源的关闭
            if(os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(socket != null) {
                try {
                    socket.close();
```

```
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    //服务端
    @Test
    public void server(){
        ServerSocket ss = null;
        Socket socket = null;
        InputStream is = null;
        ByteArrayOutputStream baos = null;
        try {
            //1.创建服务器端的 ServerSocket，指明自己的端口号
            ss = new ServerSocket(8899);
            //2.调用 accept()表示接收来自客户端的 socket
            socket = ss.accept();
            //3.获取输入流
            is = socket.getInputStream();
            //不建议这样写，可能会有乱码
//          byte[] buffer = new byte[20];
//          int len;
//          while ((len = is.read(buffer)) != -1){
//              String str = new String(buffer,0,len);
//              System.out.println(str);
//          }
            // 4.读取输入流中的数据
            baos = new ByteArrayOutputStream();
            byte[] buffer = new byte[5];
            int len;
            while ((len = is.read(buffer)) != -1){
                baos.write(buffer,0,len);
            }
            System.out.println(baos.toString());
            System.out.println("收到来自："+socket.getInetAddress().getHostAddress()+"的
数据");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //5.关闭资源
            if(baos != null) {
                try {
                    baos.close();
```

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

**实现 TCP 的网络编程**

```
 *     例子 2：客户端发送文件给服务端，服务端将文件保存在本地

public class TCPTest2 {
    @Test
    public void client(){
        Socket socket = null;
        OutputStream os = null;
        FileInputStream fis = null;
        try {
            //1.
            socket = new Socket(InetAddress.getByName("127.0.0.1"),9090);
            //2.
            os = socket.getOutputStream();
            //3.
```

```
            fis = new FileInputStream(new File("beauty.jpg"));
            //4.
            byte[] buffer = new byte[1024];
            int len;
            while((len = fis.read(buffer)) != -1){
                os.write(buffer,0,len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

}
@Test
public void server(){
        ServerSocket ss = null;
        Socket socket = null;
        InputStream is = null;
        FileOutputStream fos = null;
        try {
            //1.
            ss = new ServerSocket(9090);
            //2.
```

```
                socket = ss.accept();
                //3.
                is = socket.getInputStream();
                fos = new FileOutputStream(new File("beauty1.jpg"));
                byte[] buffer = new byte[1024];
                int len;
                while((len = is.read(buffer)) != -1){
                    fos.write(buffer,0,len);
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                if(fos != null) {
                    try {
                        fos.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
                if(is != null) {
                    try {
                        is.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
                if(socket != null) {
                    try {
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
                if(ss != null) {
                    try {
                        ss.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
```

**实现 TCP 的网络编程**

* 例题 3：从客户端发送文件给服务端，服务端保存到本地。并返回“发送成功”给客户端。并关闭相应的连接

```
/
public class TCPTest3 {
    @Test
    public void client(){
        Socket socket = null;
        OutputStream os = null;
        FileInputStream fis = null;
        InputStream is = null;
        ByteArrayOutputStream baos = null;
        try {
            //1.
            socket = new Socket(InetAddress.getByName("127.0.0.1"),9090);
            //2.
            os = socket.getOutputStream();
            //3.
            fis = new FileInputStream(new File("beauty.jpg"));
            //4.
            byte[] buffer = new byte[1024];
            int len;
            while((len = fis.read(buffer)) != -1){
                os.write(buffer,0,len);
            }
            //关闭数据的输出
            socket.shutdownOutput();
            //5.接收来自服务器端的数据，并显示到控制台上
            is = socket.getInputStream();
            baos = new ByteArrayOutputStream();
            byte[] buffer1 = new byte[20];
            int len1;
            while((len1 = is.read(buffer)) != -1){
                baos.write(buffer1,0,len1);
            }
            System.out.println(baos.toString());

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //6.
            if(fis != null) {
                try {
```

```
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(os != null) {
                try {
                    os.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(baos != null) {
                try {
                    baos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
    @Test
    public void server(){
        ServerSocket ss = null;
        Socket socket = null;
        InputStream is = null;
        FileOutputStream fos = null;
        OutputStream os = null;
        try {
            //1.
            ss = new ServerSocket(9090);
            //2.
            socket = ss.accept();
            //3.
            is = socket.getInputStream();
            //4.
            fos = new FileOutputStream(new File("beauty1.jpg"));
```

```
        //5.
        byte[] buffer = new byte[1024];
        int len;
        while((len = is.read(buffer)) != -1){
            fos.write(buffer,0,len);
        }
        //6.服务器端给予客户端反馈
        os = socket.getOutputStream();
        os.write("照片已查收！".getBytes());
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        //7.
        if(fos != null) {
            try {
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(is != null) {
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(socket != null) {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(ss != null) {
            try {
                ss.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(os != null)
        try {
            os.close();
```

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

*** UDP 协议的网络编程**

```
public class UDPTest {
    @Test
    public void sender() throws IOException {
        DatagramSocket socket = new DatagramSocket();
        String str = "我叫万达我最帅！";
        byte[] data = str.getBytes();
        InetAddress inet = InetAddress.getLocalHost();
        DatagramPacket packet = new DatagramPacket(data,0,data.length,inet,9090);
        socket.send(packet);
        socket.close();

    }

    @Test
    public void receiver() throws IOException {
        DatagramSocket socket = new DatagramSocket(9090);
        byte[] buffer = new byte[100];
        DatagramPacket packet = new DatagramPacket(buffer,0,buffer.length);
        socket.receive(packet);
        System.out.println(new String(packet.getData(),0,packet.getLength()));
        socket.close();
    }
}
```

**URL 网络编程**

```
 * 1。URL：统一资源定位符，对应着互联网的某一资源地址
 * 2.格式：
 *       http://localhost:8080/examples/beauty.jpg?usrname=Tom
 *       协议      主机名   端口号   资源地址      参数列表
public class URLTest {
    public static void main(String[] args) {
        try {
            URL url = new URL("http://localhost:8080/examples/beauty.jpg?usrname=Tom");
            //public String getProtocol( ) 获取该 URL 的协议名
            System.out.println(url.getProtocol());//http
            //public String getHost( ) 获取该 URL 的主机名
```

```
            System.out.println(url.getHost());//localhost
            //public String getPort( ) 获取该 URL 的端口号
            System.out.println(url.getPort());//8080
            //public String getPath( ) 获取该 URL 的文件路径
            System.out.println(url.getPath());// /examples/beauty.jpg
            //public String getFile( ) 获取该 URL 的文件名
            System.out.println(url.getFile());// /examples/beauty.jpg?usrname=Tom
            //public String getQuery( ) 获取该 URL 的查询名
            System.out.println(url.getQuery());// usrname=Tom
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

**从 URL 中下载资源**

```
public class URLTest1 {
    public static void main(String[] args){
        HttpURLConnection urlConnection = null;
        InputStream is = null;
        FileOutputStream fos = null;
        try {
            URL url = new URL("http://localhost:8080/examples/beauty.jpg?usrname=Tom");
            urlConnection = (HttpURLConnection) url.openConnection();
            urlConnection.connect();
            is = urlConnection.getInputStream();
            fos = new FileOutputStream("day_10\\ beauty3.jpg");
            byte[] buffer = new byte[1024];
            int len;
            while((len = is.read(buffer)) != -1){
                fos.write(buffer,0,len);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if(fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(is != null) {
                try {
```

```
                    is.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if(urlConnection != null)
            urlConnection.disconnect();
        }
    }
}
```

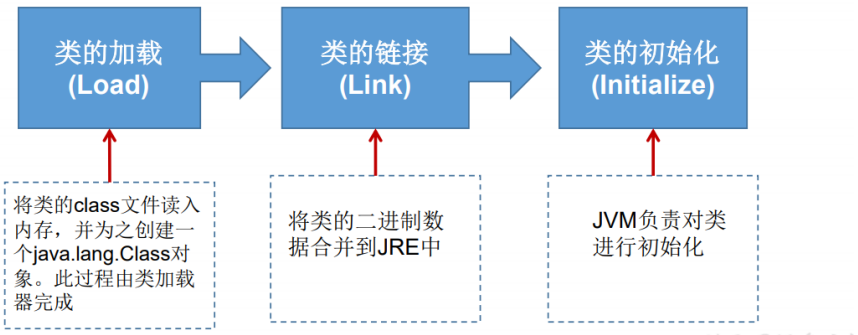
## Day_11

### 哪些类型可以有Class对象？

（1）class：
外部类，成员(成员内部类，静态内部类)，局部内部类，匿名内部类
（2）interface：接口
（3）[]：数组
（4）enum：枚举
（5）annotation：注解@interface
（6）primitive type：基本数据类型
（7）void

### 了解：类的加载过程

当程序主动使用某个类时，如果该类还未被加载到内存中，则系统会通过如下三个步骤来对该类进行初始化。




```
public class ReflectionTest {
    //反射之前，对于 Person 的操作
    @Test
    public void test1(){
        //1.创建 Person 类的对象
        Person p1 = new Person("Tom",12);
        //2.通过对象，调用其内部的属性、方法
```

```
        p1.age = 10;
        System.out.println(p1.toString());//Person{name='Tom', age=10}
        p1.show();//i am a human!
        //在 Person 类外部，不可以通过 Person 类的对象调用其内部私有结构
        //比如：name、showNation()以及私有的构造器
    }
    //反射之后，对于 Person 的操作
    @Test
    public void test2() throws Exception{
        Class clazz = Person.class;
        //1.通过反射，创建 Person 类的对象
        Constructor cons = clazz.getConstructor(String.class, int.class);
        Object obj = cons.newInstance("Tom", 12);
        Person p = (Person) obj;
        System.out.println(p.toString());//Person{name='Tom', age=12}
        //2.通过反射，调用对象指定的属性、方法
        //调用属性
        Field age = clazz.getDeclaredField("age");
        age.set(p,10);
        System.out.println(p.toString());//Person{name='Tom', age=10}
        //调用方法
        Method show = clazz.getDeclaredMethod("show");
        show.invoke(p);//i am a human!
        //通过反射，可以调用 Person 类的私有结构的。比如：私有的构造器、方法、属性。
        //调用私有的构造器
        Constructor cons1 = clazz.getDeclaredConstructor(String.class);
        cons1.setAccessible(true);
        Person p1 = (Person)cons1.newInstance("Jerry");
        System.out.println(p1);//Person{name='Jerry', age=0}
        //调用私有的属性
        Field name = clazz.getDeclaredField("name");
        name.setAccessible(true);
        name.set(p1,"songwanda");
        System.out.println(p1);//Person{name='songwanda', age=0}
        //调用私有的方法
        Method showNation = clazz.getDeclaredMethod("showNation", String.class);
        showNation.setAccessible(true);
    String nation = (String) showNation.invoke(p1,"china");//相当于 p1.showNation("中国")
                                                           //my nation is china
        System.out.println(nation);//china
    }
    /* 疑问？
       通过直接 new 的方式或反射的方式都可以调用公共的结构，开发中到底用哪个？
          建议：直接 new 的方式。
```

什么时候会使用：反射的方式。反射的特征：动态性

反射机制与面向对象中的封装性是不是矛盾的？如何看待这两个技术

不矛盾。不建议调私有结构，如果非要调用，可以使用反射。

**关于 java.lang.Class 类的理解**

1.类的加载过程；

程序在经过 javac.exe 命令以后，会生成一个或多个字节码文件(.class 结尾)，接着我们使用 java.exe 命令对某个字节码文件进行解释运行，相当于将某个字节码文件加载到内存中。此过程就称为类的加载。加载到内存中的类，我们就称为运行时类，此运行时类，就作为 Class 的一个实例。

2.换句话说，Class 的实例就对应着一个运行时类。

3.加载到内存中的运行时类，会缓存一定时间，在此时间之内，我们可以通过不同的方式来获取此运行时类。

```
 */
//获取 Class 的实例的方式（前三种方式需要掌握）
 @Test
public void test3() throws ClassNotFoundException {
    //方式一:调用运行时类的属性：.class
    Class clazz1 = Person.class;
    System.out.println(clazz1);//class com.skoud.java.Person
    //方式二：通过运行时类的对象
    Person p1 = new Person();
    Class clazz2 = p1.getClass();
    System.out.println(clazz2);//class com.skoud.java.Person
    //方式三：调用 Class 的静态方法：forName(String classPath)
    Class clazz3 = Class.forName("com.skoud.java.Person");
    System.out.println(clazz3);//class com.skoud.java.Person

    System.out.println((clazz1 == clazz2));//true
    System.out.println((clazz1 == clazz3));//true
    //方式四：使用类的加载器：ClassLoader
    ClassLoader classLoader = ReflectionTest.class.getClassLoader();
    Class clazz4 = classLoader.loadClass("com.skoud.java.Person");
    System.out.println(clazz4);//class com.skoud.java.Person
    System.out.println((clazz1 == clazz4));//true
 }
//万事万物皆对象？对象.xxx,File,URL,反射,前端、数据库操作

//Class 实例可以是哪些结构的说明：
@Test
public void test4(){
    Class c1 = Object.class;
    Class c2 = Comparable.class;
    Class c3 = String[].class;
```

```
        Class c4 = int[][].class;
        Class c5 = ElementType.class;
        Class c6 = Override.class;
        Class c7 = int.class;
        Class c8 = void.class;
        Class c9 = Class.class;
        int[] a = new int[10];
        int[] b = new int[100];
        Class c10 = a.getClass();
        Class c11 = b.getClass();
    // 只要元素类型与维度一样，就是同一个 Class
        System.out.println(c10 == c11);
    }
}
```

**了解类的加载器**

```
public class ClassLoaderTest {
    @Test
    public void test1(){
        //对于自定义类，使用系统类加载器进行加载
        ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();

    System.out.println(classLoader);//jdk.internal.loader.ClassLoaders$AppClassLoader@2f0e14
0b
        //调用系统类加载器的 getParent():获取扩展类加载器
        ClassLoader classLoader1 = classLoader.getParent();

    System.out.println(classLoader1);//jdk.internal.loader.ClassLoaders$PlatformClassLoader@7
25bef66
        //调用扩展类加载器的 getParent():无法获取引导类加载器
        //引导类加载器主要负责加载 java 的核心类库，无法加载自定义类的
        ClassLoader classLoader2 = classLoader1.getParent();
        System.out.println(classLoader2);//null

        ClassLoader classLoader3 = String.class.getClassLoader();
        System.out.println(classLoader3);//null
    }
    /*
        Properties:用来读取配置文件。

     */
    @Test
    public void test2() throws IOException {
        Properties pros = new Properties();
```

```
            //此时的文件默认在当前的 module 下
            //读取配置文件的方式一：
//          FileInputStream fis = new FileInputStream("jdbc.properties");
            FileInputStream fis = new FileInputStream("src\\jdbc1.properties");
            pros.load(fis);
            //读取配置文件的方式二：使用 ClassLoader
            //配置文件默认识别为：当前 module 的 src 下
//          ClassLoader classLoader = ClassLoaderTest.class.getClassLoader();
//          InputStream is = classLoader.getResourceAsStream("jdbc1.properties");
//          pros.load(is);
            String user = pros.getProperty("user");
            String password = pros.getProperty("password");
            System.out.println("user = " +user + ",password = "+ password);
            //user =  宋万达,password = abc123 -> user =  宋千达,password = abc123
        }
}
```

**通过反射创建对应的运行时类的对象**

```
public class NewInstanceTest {
    @Test
    public void test1() throws IllegalAccessException, InstantiationException {
        Class<Person> clazz = Person.class;
        /*
 newInstance():调用此方法，创建对应的运行时类的对象。内部调用了运行时类的空参构造器。要想此方法正常的创建运行时类的对象，要求：
          1.运行时类必须提供空参的构造器
          2.空参的构造器的访问权限得够。通常，设置为 public。
  通常在 javabean 中要求提供一个 public 的空参构造器。原因：
          1.便于通过反射，创建运行时类的对象。
          2.便于子类继承此运行时类时，默认调用 super()时，保证父类有此构造器
        Person obj = clazz.newInstance();
        System.out.println(obj);// Person{name='null', age=0}
    }
    //体会反射的动态性
    @Test
    public void test2(){

        int num = new Random().nextInt(3);//0,1,2
        String classPath = "";
        switch (num){
            case 0:
                classPath = "java.util.Date";
                break;
            case 1:
                classPath = "java.lang.Object";
```

```
                break;
            case 2:
                classPath = "com.skoud.java.Person";
                break;
        }
        try {
            Object obj = getInstance(classPath);
            System.out.println(obj);//运行时才知道构造了哪个对象
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /*
      创建一个指定类的对象
      classPath：指定类的全类名
     */
    public Object getInstance(String classPath) throws Exception{
        Class clazz = Class.forName(classPath);
        return clazz.newInstance();
    }
}
```

**获取当前运行时类的属性结构（了解）**

```
public class FieldTest {
    @Test
    public void test1(){
        Class clazz = Person.class;
        //获取属性结构
        //getFields():获取当前运行时类及其父类中声明为 public 访问权限的属性
        Field[] fields = clazz.getFields();
        for(Field f : fields){
        System.out.println(f);//public int com.skoud.java1.Person.id 、 public double
com.skoud.java1.Creature.weight
        }
    //getDeclaredFields():获取当前运行时类中声明的所有属性，不包含父类中声明的属性
        Field[] declaredFields = clazz.getDeclaredFields();
        for(Field f : declaredFields){
            System.out.println(f);//private java.lang.String com.skoud.java1.Person.name
            //int com.skoud.java1.Person.age、 public int com.skoud.java1.Person.id
        }
    }
    //权限修饰符 数据类型 变量名
    @Test
    public void test2(){
```

```
        Class clazz = Person.class;
        Field[] declaredFields = clazz.getDeclaredFields();
        for(Field f : declaredFields){
            //1.权限修饰符
            int modifier = f.getModifiers();
            System.out.println(Modifier.toString(modifier));//private  缺省  public
            //2.数据类型
            Class type = f.getType();
            System.out.println(type.getName());//java.lang.String int int
            //3.变量名
            String fName = f.getName();
            System.out.println(fName);//name age id
        }
    }
}
```

**获取运行时类的方法结构(了解)**

```
public class MethodTest {
    @Test
    public void test1(){
        Class clazz = Person.class;
        //getMethods():获取当前运行时类及其所有父类中声明为 public 权限的方法
        Method[] methods = clazz.getMethods();
        for(Method m : methods){
            System.out.println(m);
        }
        //getDeclaredMethods():获取当前运行时类中声明的所有方法。（不包含父类中声明
的方法）
        Method[] declaredMethods = clazz.getDeclaredMethods();
        for(Method m : declaredMethods){
            System.out.println(m);
        }
    }
    /*
    @Xxxx
    权限修饰符  返回值类型  方法名(参数类型 1  形参名 1,...) throws XxxException{}
     */
    @Test
    public void test2(){
        Class clazz = Person.class;
        Method[] declaredMethods = clazz.getDeclaredMethods();
        for(Method m : declaredMethods){
            //1.获取方法声明的注解
            Annotation[] annotations = m.getAnnotations();
```

```
                for(Annotation a : annotations){
                    System.out.println(a);//@com.skoud.java1.MyAnnotation("hello")
                }
                //2.权限修饰符
                System.out.print(Modifier.toString(m.getModifiers()) + "\t");
                //3.返回值类型
                System.out.print(m.getReturnType().getName() + "\t");
                //4.方法名
                System.out.print(m.getName());
                System.out.print("(");
                //5.形参列表
                Class[] parameterTypes = m.getParameterTypes();
                if(!(parameterTypes == null && parameterTypes.length == 0)){
                    for (int i = 0; i < parameterTypes.length; i++) {
                        if(i == parameterTypes.length -1){
                            System.out.print(parameterTypes[i].getName() + "args_" + i);
                            break;
                        }
                        System.out.print(parameterTypes[i].getName() + "args_" + i + ",");
                    }
                }
                System.out.print(")");
                //6.抛出的异常
                Class[] exceptionTypes = m.getExceptionTypes();
                if(exceptionTypes.length > 0){
                    System.out.print("throws ");
                    for (int i = 0; i < exceptionTypes.length; i++) {
                        if(i == exceptionTypes.length -1){
                            System.out.print(exceptionTypes[i].getName() );
                            break;
                        }
                        System.out.print(exceptionTypes[i].getName() + ",");
                    }
                }
                System.out.println();
            }
        }
    }
```

**获取当前运行时类其他特征**

```
public class OtherTest {
    /*
      获取构造器结构
     */
```

```
@Test
public void test1(){
    Class clazz = Person.class;
    //getConstructors():获取当前运行时类中声明为 public 的构造器
    Constructor[] constructors = clazz.getConstructors();
    for(Constructor c : constructors){
        System.out.println(c);//public com.skoud.java1.Person()
    }
    //getDeclaredConstructors():获取当前运行时类中声明的所有构造器
    Constructor[] declaredConstructors = clazz.getDeclaredConstructors();
    for(Constructor c : declaredConstructors){
        System.out.println(c);//com.skoud.java1.Person(java.lang.String,int)
                              //private com.skoud.java1.Person(java.lang.String)
                              //public com.skoud.java1.Person()
    }
}
/*
 获取运行时类的父类
 */
@Test
public void test2(){
    Class clazz = Person.class;
    Class superclass = clazz.getSuperclass();
    System.out.println(superclass);//class com.skoud.java1.Creature

}
/*
获取运行时类的带泛型的父类
*/
@Test
public void test3(){
    Class clazz = Person.class;
    Type genericSuperclass = clazz.getGenericSuperclass();
    System.out.println(genericSuperclass);//com.skoud.java1.Creature<java.lang.String>
}
/*
获取运行时类的带泛型的父类的泛型

代码：逻辑性代码 vs 功能性代码
*/
@Test
public void test4(){
    Class clazz = Person.class;
    Type genericSuperclass = clazz.getGenericSuperclass();
```

```
        ParameterizedType paramType = (ParameterizedType) genericSuperclass;
        //获取泛型类型
        Type[] actualTypeArguments = paramType.getActualTypeArguments();
        System.out.println(((Class)actualTypeArguments[0]).getName());//java.lang.String
    }
    /*
    获取运行时类实现的接口
     */
    @Test
    public void test5(){
        Class clazz = Person.class;
        Class[] interfaces = clazz.getInterfaces();
        for(Class c: interfaces){
            System.out.println(c);//interface java.lang.Comparable 、 interface
com.skoud.java1.MyInterface
        }
        System.out.println();
        //获取运行时类的父类实现的接口
        Class[] interfaces1 = clazz.getSuperclass().getInterfaces();
        for(Class c : interfaces1){
            System.out.println(c);//interface java.io.Serializable
        }
    }
    /*
        获取运行时类所在的包
     */
    @Test
    public void test6(){
        Class clazz = Person.class;
        Package pack = clazz.getPackage();
        System.out.println(pack);//package com.skoud.java1
    }
    /*
      获取运行时类声明的注解
     */
    @Test
    public void test7(){
        Class clazz = Person.class;
        Annotation[] annotations = clazz.getAnnotations();
        for(Annotation a : annotations){
            System.out.println(a);//@com.skoud.java1.MyAnnotation("hi")
        }
    }
}
```

**调用运行时类中指定的结构：属性、方法、构造器**

```
public class ReflectionTest {
    /*
        不需要掌握
     */
    @Test
    public void testField() throws Exception{
        Class clazz = Person.class;
        //创建运行时类的对象
        Person p = (Person) clazz.newInstance();
        //获取指定的属性:要求运行时类中属性声明为 public
        //通常不采用此方法
        Field id = clazz.getField("id");
        /*
         设置当前属性的值
          set():参数 1：指明设置哪个对象的属性  参数 2：将此属性值设置为多少
         */
        id.set(p,1001);
        /*
         获取当前属性的值
         get()：参数 1：获取哪个对象的当前属性值
         */
        int pId = (int) id.get(p);
        System.out.println(pId);//1001
    }
    /*
        如何操作运行时类中的指定属性  --需要掌握
     */
    @Test
    public void testField1() throws Exception{
        Class clazz = Person.class;
        //创建运行时类的对象
        Person p = (Person) clazz.newInstance();
        //1.getDeclaredField(String fieldName):获取运行时类中指定变量名的属性
        Field name = clazz.getDeclaredField("name");
        //2.保证当前属性是可访问的
        name.setAccessible(true);
        //3.获取、设置指定对象的此属性值
        name.set(p,"Tom");
        System.out.println(name.get(p));
    }
    /*
        如何操作运行时类中的指定的方法  --  需要掌握
     */
```

```
    @Test
    public void testMethod() throws Exception{
        Class clazz = Person.class;
        //创建运行时类的对象
        Person p = (Person) clazz.newInstance();
        /*    1.获取指定的某个方法
              getDeclaredMethod():参数 1：指明获取的方法的名称  参数 2：指明获取的方法的形参列表

         */
        Method show = clazz.getDeclaredMethod("show", String.class);
        //2.保证当前属性是可访问的
        show.setAccessible(true);
        /*
              3.调用方法的 invoke():参数 1：方法的调用者  参数 2：给方法形参赋值的实参
              invoke()的返回值即为对应类中调用的方法的返回值。
         */
        Object returnValue = show.invoke(p, "china");//我的国籍是：china
        System.out.println(returnValue);//china
        //如何调用静态方法
        //private static void showDesc()
        Method showDesc = clazz.getDeclaredMethod("showDesc");
        showDesc.setAccessible(true);
        //如果调用的运行时类中的方法没有返回值，则此 invoke()返回 null
//        Object returnVal = showDesc.invoke(Person.class);//i am a good man
        Object returnVal = showDesc.invoke(null);//i am a good man
        System.out.println(returnVal);//null
    }
    /*
        如何调用运行时类中的指定的构造器
     */
    @Test
    public void testConstructor() throws Exception{
        Class clazz = Person.class;
        /*
              1.获取指定的构造器
              getDeclaredConstructors():参数：指明构造器的参数列表
         */
        Constructor constructor = clazz.getDeclaredConstructor(String.class);
        //2.保证此构造器是可访问的
        constructor.setAccessible(true);
        //3.调用此构造器创建运行时类的对象
        Person person = (Person) constructor.newInstance("Tom");
        System.out.println(person);//Person{name='Tom', age=0, id=0}
```

```
    }
}
```

## Day_12

**静态代理举例**

```
 * 特点：代理类和被代理类在编译期间，就确定下来了。
 *
interface ClothFactory{
    void produceCloth();
}
//代理类
class ProxyClothFactory implements ClothFactory{
    private ClothFactory factory;//用被代理类对象进行实例化

    public ProxyClothFactory(ClothFactory factory) {
        this.factory = factory;
    }

    @Override
    public void produceCloth() {
        System.out.println("代理工厂做一些准备工作！");
        factory.produceCloth();
        System.out.println("代理工厂做一些后续的收尾工作");
    }
}
//被代理类
class NikeClothFactory implements ClothFactory{

    @Override
    public void produceCloth() {
        System.out.println("Nike 生产一批运动服");
    }
}
public class StaticProxyTest {
    public static void main(String[] args) {
        //创建被代理类对象
        NikeClothFactory nike = new NikeClothFactory();
        //创建代理类的对象
        ProxyClothFactory proxyClothFactory = new ProxyClothFactory(nike);
        proxyClothFactory.produceCloth();
        //代理工厂做一些准备工作！
        //Nike 生产一批运动服
        //代理工厂做一些后续的收尾工作
```

```
    }
}
```

**动态代理举例**

```
interface Human{
    String getBelief();
    void eat(String food);
}
//被代理类
class SuperMan implements Human{

    @Override
    public String getBelief() {
        return "I believe i can fly";
    }
    @Override
    public void eat(String food) {
        System.out.println("I like eating " + food);
    }
}
//AOP 与动态代理
class HumanUtil{
    public void method1(){
        System.out.println("==============通用方法一==============");
    }
    public void method2(){
        System.out.println("==============通用方法二==============");
    }
}
/*
    要想实现动态代理，需要解决的问题？
        问题一：如何根据加载到内存中的被代理类，动态的创建一个代理类及其对象
        问题二：当通过代理类的对象调用方法时，如何动态的调用被代理类中的同名方法。
 */
class ProxyFactory{
    //调用此方法，返回一个代理类的对象。解决问题一
    public static Object getProxyInstance(Object obj){//obj:被代理类的对象
        MyInvocationHandler handler = new MyInvocationHandler();
        handler.bind(obj);
        return
Proxy.newProxyInstance(obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),handler);
    }
}
class MyInvocationHandler implements InvocationHandler{
```

```
    private Object obj;//需要使用被代理的对象进行赋值
    public void bind(Object obj){
        this.obj = obj;
    }
    //当我们通过代理类的对象，调用方法 a 时，就会自动的调用如下的方法:invoke()
    //将被代理类要执行的方法 a 的功能就声明为 invoke()中
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        HumanUtil util = new HumanUtil();
        util.method1();
        //method：即为代理类对象调用的方法，此方法也就作为了被代理类对象要调用的
方法
        //obj:被代理类的对象
        Object returnValue = method.invoke(obj, args);
        util.method2();
        //上述方法的返回值就作为当前类中的 invoke()的返回值
        return returnValue;

    }
}
public class ProxyTest {
    public static void main(String[] args) {
        //只需要知道代理类和接口
        SuperMan superman = new SuperMan();
        //proxyInstance:代理类的对象
        Human proxyInstance =(Human) ProxyFactory.getProxyInstance(superman);
        //当通过代理类对象调用方法时，会自动地调用被代理类中同名的方法
        System.out.println(proxyInstance.getBelief());//I believe i can fly
        proxyInstance.eat("HotPot");//I like eating HotPot
        NikeClothFactory nikeClothFactory = new NikeClothFactory();
ClothFactory proxyClothFactory =
 (ClothFactory) ProxyFactory.getProxyInstance(nikeClothFactory);
        proxyClothFactory.produceCloth();//Nike 生产一批运动服
    }
}
```

**Lambda 表达式的使用**

```
 * 1.举例  (o1,o2) -> Integer.compare(o1,o2);
 * 2. 格式：
 *        -> : lambda 操作符  或  箭头操作符
 *        ->左边：lambda 的形参列表  (其实就是接口中的抽象方法的形参列表)
 *        ->右边：lambda 体(其实就是重写的抽象方法的方法体)
 *
 * 3.Lambda 表达式的使用：(分为 6 种情况介绍)
```

```
 *       总结：
 *       ->左边：lambda 形参列表的参数类型可以省略（类型推断）：如果 lambda 形参列表只有一个参数，其一对()也可以省略
 *       ->右边：lambda 体应该使用一对{}包裹；如果 lambda 体只有一条执行语句(可能是 return 语句)，省略这一对{}和 return。
 *
 * 4.Lambda 表达式的本质：作为接口的实例
 * 5.如果一个接口中，只声明了一个抽象方法，则此接口就称为函数式接口。我们可以在一个接口上使用 @FunctionalInterface 注解，这样做可以检查它是否是一个函数式接口。
 *
 * 6.所以以前用匿名实现类表示的现在都可以用 Lambda 表达式来写

public class LambdaTest1 {
    //语法格式一：无参，无返回值
    @Test
    public void test1() {
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("i love nanjing");
            }
        };
        r1.run();//i love nanjing
        System.out.println("***************");
        Runnable r2 = () -> {
            System.out.println("i love njupt");
        };
        r2.run();//i love njupt
    }
    //Lambda 需要一个参数，但是没有返回值。
    @Test
    public void test2(){
        Consumer<String> con = new Consumer<>() {
            @Override
            public void accept(String s) {
                System.out.println(s);
            }
        };
        con.accept("孤舟荡尽波澜");//孤舟荡尽波澜
        System.out.println("***************");
        Consumer<String> con1 = (String s) ->{
            System.out.println(s);
        };
        con1.accept("再日出，又扬帆！");//再日出，又扬帆！
```

```
}
//数据类型可以省略，因为可由编译器推断得出，称为“类型推断”
@Test
public void test3(){
    Consumer<String> con1 = (String s) ->{
        System.out.println(s);
    };
    con1.accept("再日出，又扬帆！");//再日出，又扬帆！
    System.out.println("**********");
    Consumer<String> con2 = (s) -> {
        System.out.println(s);
    };
    con2.accept("再日出，又扬帆！");//再日出，又扬帆！
}
@Test
public void test4(){
    ArrayList<String> list = new ArrayList<>();//类型推断
    int[] arr = {1,2,3};//类型推断
}
//语法格式四：Lambda 若只需要一个参数时，参数的小括号可以省略
@Test
public void test5(){
    Consumer<String> con1 = (s) -> {
        System.out.println(s);
    };
    con1.accept("再日出，又扬帆！");//再日出，又扬帆！
    System.out.println("**********");
    Consumer<String> con2 = s -> {
        System.out.println(s);
    };
    con2.accept("再日出，又扬帆！");//再日出，又扬帆！
}
//语法格式五：Lambda 需要两个或以上的参数，多条执行语句，并且可以有返回值
@Test
public void test6(){
    Comparator<Integer> com1 = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            System.out.println(o1);
            System.out.println(o2);
            return o1.compareTo(o2);
        }
    };
    com1.compare(12,21);// 12 21 -1
```

```
        System.out.println("********************");
        Comparator<Integer> com2 = (o1,o2) -> {
            System.out.println(o1);
            System.out.println(o2);
            return o1.compareTo(o2);
        };
        System.out.println(com2.compare(12, 6));//12    6    1
    }
    //语法格式六：当 Lambda 体只有一条语句时，return 与大括号若有，都可以省略
    public void test7(){
        Comparator<Integer> com1 = (o1,o2) -> {
            return o1.compareTo(o2);
        };
        System.out.println(com1.compare(12, 6));//12 6 1
        System.out.println("**********");
        Comparator<Integer> com2 =(o1,o2) -> o1.compareTo(o2);
        System.out.println(com2.compare(12, 21));//12 21 -1
    }
    @Test
    public void test8(){
        Consumer<String> con1 = (String s) ->{
            System.out.println(s);
        };
        con1.accept("再日出，又扬帆！");//再日出，又扬帆！
        System.out.println("**********");
        Consumer<String> con2 = s -> System.out.println(s);
        con2.accept("再日出，又扬帆！");//再日出，又扬帆！
    }
}
```

# Java 内置四大核心函数式接口

| 函数式接口 | 参数类型 | 返回类型 | 用途 |
| --- | --- | --- | --- |
| **Consumer<T>**<br>**消费型接口** | T | void | 对类型为T的对象应用操作，包含方法：<br>**void accept(T t)** |
| **Supplier<T>**<br>**供给型接口** | 无 | T | 返回类型为T的对象，包含方法：**T get()** |
| **Function<T, R>**<br>**函数型接口** | T | R | 对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法：**R apply(T t)** |
| **Predicate<T>**<br>**断定型接口** | T | boolean | 确定类型为T的对象是否满足某约束，并返回 boolean 值。包含方法：**boolean test(T t)** |

| | | | |
|---|---|---|---|
| BiFunction<T, U, R> | T, U | R | 对类型为 T, U 参数应用操作，返回 R 类型的结果。包含方法为： R apply(T t, U u); |
| UnaryOperator<T> (Function子接口) | T | T | 对类型为T的对象进行一元运算，并返回T类型的结果。包含方法为：T apply(T t); |
| BinaryOperator<T> (BiFunction 子接口) | T, T | T | 对类型为T的对象进行二元运算，并返回T类型的结果。包含方法为： T apply(T t1, T t2); |
| BiConsumer<T, U> | T, U | void | 对类型为T, U 参数应用操作。<br>包含方法为： void accept(T t, U u) |
| BiPredicate<T,U> | T,U | boolean | 包含方法为： boolean test(T t,U u) |
| ToIntFunction<T><br>ToLongFunction<T><br>ToDoubleFunction<T> | T | int<br>long<br>double | 分别计算int、long、double值的函数 |
| IntFunction<R><br>LongFunction<R><br>DoubleFunction<R> | int<br>long<br>double | R | 参数分别为int、long、double 类型的函数 |

*** java 内置的 4 大核心函数式接口**

```
 * 消费型接口 Consumer<T>      void accept(T t)
 * 供给型接口 Supplier<T>      T get()
 * 函数型接口 Function<T,R>    R apply(T t)
 * 断定型接口 Predicate<T>     boolean test(T t)
 *
public class LambdaTest2 {
    @Test
    public void test1(){
        happyTime(500, new Consumer<Double>() {
            @Override
            public void accept(Double aDouble) {
                System.out.println("开心消费了： " + aDouble);
            }
        });//开心消费了： 500.0
        System.out.println("************");
        happyTime(400,money -> System.out.println("开心消费了： " + money));
        //开心消费了： 400.0
    }
    public void happyTime(double money, Consumer<Double> con){
        con.accept(money);
    }
    @Test
    public void test2(){
        List<String> list = Arrays.asList("北京","南京","天津","东京","西京","普京");
        List<String> filterStrs = filterString(list, new Predicate<String>() {
            @Override
            public boolean test(String s) {
                return s.contains("京");
            }
        });
```

```
        System.out.println(filterStrs);//[北京，南京，东京，西京，普京]
        List<String> filterStrs1 = filterString(list,s -> s.contains("京"));
        System.out.println(filterStrs1);//[北京，南京，东京，西京，普京]

    }
    //根据给定的规则，过滤集合中的字符串。此规则由 Predicate 的方法决定
    public List<String> filterString(List<String> list, Predicate<String> pre){
        ArrayList<String> filterList = new ArrayList<>();
        for(String s : list){
            if(pre.test(s)){
                filterList.add(s);
            }
        }
        return filterList;
    }
}
```

**方法引用的使用**

```
 * 1.使用情景：当要传递给 Lambda 体的操作，已经有实现的方法了，可以使用方法引用！
 * 2.方法引用，本质上就是 lambda 表达式，而 lambda 表达式作为函数式接口的实例。所以方法引用，也是函数式接口的实例。
 * 3.使用格式：    类(或对象) :: 方法名
 * 4.具体分为如下的三种情况：
 *    情况 1：对象 :: 非静态方法
 *    情况 2：类 :: 静态方法
 *    情况 3：类 :: 非静态方法
 * 5.方法引用使用的要求：要求接口中的抽象方法的”形参列表和返回值类型“与方法引用的方法的”形参列表和返回值类型相同“。
 *
public class MethodRefTest {
    // 情况一：对象 :: 实例方法
    //Consumer 中的 void accept(T t)
    //PrintStream 中的 void println(T t)
    @Test
    public void test1() {
        Consumer<String> con1 = str -> System.out.println(str);
        con1.accept("beijing");//beijing
        System.out.println("*************");
        PrintStream ps = System.out;
        Consumer<String> con2 = ps::println;
        con2.accept("beijing");//beijing
    }

    //Supplier 中的 T get()
```

```
//Employee 中的 String getName()
@Test
public void test2() {
	Employee emp = new Employee(1001, "Tom", 23, 5600);
	Supplier<String> sup1 = () -> emp.getName();
	System.out.println(sup1.get());//Tom
	System.out.println("**********");
	Supplier<String> sup2 = emp::getName;
	System.out.println(sup2.get());//Tom
}

// 情况二：类 :: 静态方法
//Comparator 中的 int compare(T t1,T t2)
//Integer 中的 int compare(T t1,T t2)
@Test
public void test3() {
	Comparator<Integer> com = (t1,t2) ->    Integer.compare(t1,t2);
	System.out.println(com.compare(12, 21));//-1
	System.out.println("**********");
	Comparator<Integer> com2 = Integer::compareTo;
	System.out.println(com2.compare(12, 21));//-1
}

//Function 中的 R apply(T t)
//Math 中的 Long round(Double d)
@Test
public void test4() {
	Function<Double,Long> func1 = d -> Math.round(d);
	System.out.println(func1.apply(12.3));//12
	System.out.println("**********");
	Function<Double,Long> func2 = Math::round;
	System.out.println(func2.apply(12.6));//13
}

// 情况三：类 :: 实例方法  (有难度)
// Comparator 中的 int comapre(T t1,T t2)
// String 中的 int t1.compareTo(t2)
@Test
public void test5() {
	Comparator<String> com1 = (s1,s2) -> s1.compareTo(s2);
	System.out.println(com1.compare("abc", "abd"));//-1
	System.out.println("**********");
	Comparator<String> com2 = String::compareTo;
	System.out.println(com2.compare("abd", "abc"));//1
```

```
    }

    //BiPredicate 中的 boolean test(T t1, T t2);
    //String 中的 boolean t1.equals(t2)
    @Test
    public void test6() {
        BiPredicate<String,String> pre1 = (s1,s2) -> s1.equals(s2);
        System.out.println(pre1.test("abc", "abc"));//true
        BiPredicate<String,String> pre2 = String::equals;
        System.out.println(pre2.test("abc", "abd"));//false

    }

    // Function 中的 R apply(T t)
    // Employee 中的 String getName();
    @Test
    public void test7() {
        Employee emp = new Employee(1001, "Jerry", 23, 500);
        Function<Employee,String> func1 = e -> e.getName();
        System.out.println(func1.apply(emp));//Jerry
        System.out.println("*************");
        Function<Employee,String> func2 = Employee::getName;
        System.out.println(func2.apply(emp));//Jerry1
    }
}

一、构造器引用
 *  和方法引用类似，函数式接口的抽象方法的形参列表和构造器的形参列表一致。
 *  抽象方法的返回值类型即为构造器所属的类的类型。
 * 二、数组引用
 *  可把数组看做是一个特殊的类，则写法与构造器引用一致。
*/
public class ConstructorRefTest {
    //构造器引用
    //Supplier 中的 T get()
    //Employee 的空参构造器:Employee()
    @Test
    public void test1(){
        Supplier<Employee> sup = new Supplier<Employee>() {
            @Override
            public Employee get() {
                return new Employee();
            }
        };
```

```
        sup.get();
        System.out.println("********************");
        Supplier<Employee> sup1 = () ->    new Employee();
        sup1.get();
        System.out.println("********************");
        Supplier<Employee> sup2 = Employee::new;
        sup2.get();
    }

    //Function 中的 R apply(T t)
    @Test
    public void test2(){
        Function<Integer,Employee> func1 = id -> new Employee(id);
        Employee employee = func1.apply(1001);
        System.out.println(employee);//Employee{id=1001, name='null', age=0, salary=0.0}
        System.out.println("**********************");
        Function<Integer,Employee> func2 = Employee::new;
        Employee employee1 = func2.apply(1002);
        System.out.println(employee1);//Employee{id=1002, name='null', age=0, salary=0.0}
    }

    //BiFunction 中的 R apply(T t,U u)
    @Test
    public void test3(){
        BiFunction<Integer,String,Employee> func1 = (id,name) -> new Employee(id,name);
        System.out.println(func1.apply(1001,"Tom"));//Employee{id=1001, name='Tom', age=0,
salary=0.0}
        System.out.println("********************");
        BiFunction<Integer,String,Employee> func2 = Employee::new;
        System.out.println(func2.apply(1002,    "Jerry"));//Employee{id=1002,    name='Jerry',
age=0, salary=0.0}
    }

    //数组引用
    //Function 中的 R apply(T t)
    @Test
    public void test4(){
        Function<Integer,String[]> fun1 = length -> new String[length];
        String[] arr1 = fun1.apply(5);
        System.out.println(Arrays.toString(arr1));//[null, null, null, null, null]
        System.out.println("********************");
        Function<Integer,String[]> fun2 = String[]::new;
        String[] arr2 = fun2.apply(5);
        System.out.println(Arrays.toString(arr2));//[null, null, null, null, null]
```

```
    }
}
```

**Stream 类**

```
1. Stream 关注的是对数据的运算，与 CPU 打交道
 *      集合关注的是数据的存储，与内存打交道
 * 2.
 *    Stream  自己不会存储元素。
 *    Stream  不会改变源对象。相反，他们会返回一个持有结果的新 Stream。
 *    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 执行流程
 * ① Stream 的实例化
 * ② 一系列的中间操作(过滤、映射、。。。)
 * ③ 终止操作
 *
 * 4.说明：
 *   4.1 一个中间操作链，对数据源的数据进行处理
 *   4.2 一旦执行终止操作，就执行中间操作链，并产生结果。之后，不会再被使用。
 *
 * 测试 Stream 的实例化
 *
public class StreamAPITest {
    //创建 Stream 方式一：通过集合
    @Test
    public void test1(){
        List<Employee> employees = EmployeeData.getEmployees();
        //default Stream<E> stream() : 返回一个顺序流
        Stream<Employee> stream = employees.stream();
        // efault Stream<E> parallelStream() : 返回一个并行流
        Stream<Employee> parallelStream = employees.parallelStream();
    }
    //创建 Stream 方式二：通过数组
    @Test
    public void test2(){
        int[] arr = new int[]{1,2,3,4,5,6};
        //调用 Arrays 类的 static <T> Stream<T> stream(T[] array): 返回一个流
        IntStream stream = Arrays.stream(arr);
        Employee e1 = new Employee(1001, "Tom");
        Employee e2 = new Employee(1002, "Jerry");
        Employee[] arr1 = new Employee[]{e1,e2};
        Stream<Employee> stream1 = Arrays.stream(arr1);
    }
    //创建 Stream 方式三：通过 Stream 的 of()
```

```
    @Test
    public void test3(){
        Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6);
    }
    //创建 Stream 方式四：创建无限流
    @Test
    public void test4(){
        //迭代
        //public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
        //遍历前 10 个偶数
        Stream.iterate(0,t -> t + 2).limit(10).forEach(System.out::println);
        //生成
        //public static<T> Stream<T> generate(Supplier<T> s)
        Stream.generate(Math::random).limit(10).forEach(System.out::println);
    }
}
```

*** 测试 Stream 的中间操作**

```
public class StreamAPITest1 {
    //1-筛选与切片
    @Test
    public void test1(){
        List<Employee> list = EmployeeData.getEmployees();
        //filter(Predicate p):接收 Lambda,从流中排除某些元素
        Stream<Employee> stream = list.stream();
        //练习：查询员工表中薪资大于 7000 的员工信息
        stream.filter(e -> e.getSalary() > 7000).forEach(System.out::println);//e 是 Employee 的
实例
            // filter(predicate).forEach(Consumer)   ---> 中间操作.终止操作
        //limit(long maxSize):截断流，使其元素不超过给定数量
        list.stream().limit(3).forEach(System.out::println);//重新 list.stream
        //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
        list.stream().skip(3).forEach(System.out::println);
        //distinct():筛选，通过流所生成元素的 hashCode()和 equals()去除重复元素
        list.add(new Employee(1010,"刘强东",40,8000));
        list.add(new Employee(1011,"宋万达",23,250));
        list.add(new Employee(1011,"宋万达",23,250));
        list.stream().distinct().forEach(System.out::println);
    }
    //2-映射
    @Test
    public void test2(){
```

```
        //map(Function f):接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
        List<String> list = Arrays.asList("aa", "bb", "cc", "dd");
        list.stream().map(str -> str.toUpperCase()).forEach(System.out::println);//AA BB CC DD
        //练习 1：获取员工姓名长度大于 3 的员工的姓名。
        List<Employee> employees = EmployeeData.getEmployees();
        Stream<String> nameStream = employees.stream().map(Employee::getName);
        nameStream.filter(name -> name.length() >3).forEach(System.out::println);

        //练习 2：
Stream<Stream<Character>> streamStream = list.stream().map(StreamAPITest1::fromStringToStream);
        streamStream.forEach(s ->{
            s.forEach(System.out::println);//a a b b c c d d
        });
        //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
Stream<Character> characterStream = list.stream().flatMap(StreamAPITest1::fromStringToStream);
        characterStream.forEach(System.out::println);//a a b b c c d d

        //mapToDouble(ToDoubleFunction f):接收一个函数作为参数，该函数会被应用到每个元素上，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个元素上，产生一个新的 LongStream。
    }
    //将字符串中的多个字符构成的集合转换为对应的 Stream 的实例·
    public static Stream<Character> fromStringToStream(String str){
        ArrayList<Character> list = new ArrayList<>();
        for(Character c: str.toCharArray()){
            list.add(c);
        }
        return list.stream();
    }
    @Test
    public void test3(){
        ArrayList<Integer> list1 = new ArrayList<>();
        list1.add(1);
        list1.add(2);
        list1.add(3);
        ArrayList list2 = new ArrayList();
        list2.add(4);
        list2.add(5);
        list2.add(6);
```

```
//          list1.add(list2);
            System.out.println(list1.size());//4
            list1.addAll(list2);
            System.out.println(list1.size());//6
        }
        //3-排序
        @Test
        public void test4(){
            //sorted() 产生一个新流，其中按自然顺序排序
            List<Integer> list = Arrays.asList(12, 43, 65, 34, 87, 0, -98, 7);
            list.stream().sorted().forEach(System.out::print);//-98 0 7 12 34 43 65 87

            List<Employee> employees = EmployeeData.getEmployees();
            employees.stream().sorted().forEach(System.out::println);//报错，原因：Employee 没有
实现 Comparable 接口
            //sorted(Comparator com) 产生一个新流，其中按比较器顺序排序
//employees.stream().sorted((e1,e2) -> Integer.compare(e1.getAge(),e2.getAge())).forEach(System.out::println);
            employees.stream().sorted((e1,e2) -> {
int ageValue = Integer.compare(e1.getAge(), e2.getAge());
                if(ageValue != 0){
                    return ageValue;
                }else {
                    return Double.compare(e1.getSalary(),e2.getSalary());
                }//二级排序
            }).forEach(System.out::println);

        }
}
```

*** 测试 Stream 的终止操作**

```
public class StreamAPITest2 {
    //1.匹配与查找
    @Test
    public void test() {
        List<Employee> employees = EmployeeData.getEmployees();
        //allMatch(Predicate p) 检查是否匹配所有元素
        //练习：是否所有员工的年龄都大于 18
        boolean allMatch = employees.stream().allMatch(e -> e.getAge() > 18);
        System.out.println(allMatch);//false
        //anyMatch(Predicate p) 检查是否至少匹配一个元素
        boolean anyMatch = employees.stream().anyMatch(e -> e.getSalary() > 10000);
        System.out.println(anyMatch);//false
        //noneMatch(Predicate p) 检查是否没有匹配所有元素
        //练习：是否存在员工姓"雷"
```

```
        boolean noneMatch = employees.stream().noneMatch(e -> e.getName().startsWith("雷"));
        System.out.println(noneMatch);//false
        //findFirst() 返回第一个元素
        Optional<Employee> employee = employees.stream().findFirst();
        System.out.println(employee);//Optional[Employee{id=1001, name='马化腾', age=34, salary=6000.38}]
        //findAny() 返回当前流中的任意元素
        Optional<Employee> any = employees.parallelStream().findAny();
        System.out.println(any);//Optional[Employee{id=1006, name='比尔盖茨', age=42, salary=9500.43}]
        //count() 返回流中元素总数
        long count = employees.stream().count();
        System.out.println(count);//8
        //max(Comparator c) 返回流中最大值
        //练习返回最高的工资
        Stream<Double> salaryStream = employees.stream().map(Employee::getSalary);
        Optional<Double> maxSalary = salaryStream.max((e1, e2) -> Double.compare(e1, e2));
//      Optional<Double> maxSalary1 = salaryStream.max(Double::compare);
        System.out.println(maxSalary);//Optional[9876.12]
        //min(Comparator c) 返回流中最小值
        //返回工资最低的员工
        Optional<Employee> employee1 = employees.stream().min((e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary()));
        System.out.println(employee1);//Optional[Employee{id=1008, name='扎克伯格', age=35, salary=2500.32}]
        //forEach(Consumer c)内部迭代 (使用 Collection 接口需要用户去做迭代，称为外部迭代。相反，Stream API 使用内部迭代——它帮你把迭代做了)
        employees.stream().forEach(System.out::println);
    }
    //2-归约
    @Test
    public void test2(){
        //reduce(T identity, BinaryOperator b) 可以将流中元素反复结合起来，得到一个值。返回 T
        //练习 1：计算 1-10 的自然数的和
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        Integer sum = list.stream().reduce(0, Integer::sum);
        System.out.println(sum);//55 (identity:初始值 10 结果变为 65)
        //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>
        //练习 2：计算公司所有员工工资的总和
        List<Employee> employees = EmployeeData.getEmployees();
        Stream<Double> salaryStream = employees.stream().map(Employee::getSalary);
//      Optional<Double> sumMoney = salaryStream.reduce(Double::sum);
```

```
        Optional<Double> sumMoney = salaryStream.reduce((d1, d2) -> d1 + d2);
        System.out.println(sumMoney);//Optional[48424.08]
    }
    //3-收集
    @Test
    public void test3(){
        //collect(Collector c)将流转换为其他形式。接收一个 Collector 接口的实现，用于给Stream 中元素做汇总的方法
        //练习：查找工资大于 6000 的员工，结果返回为一个 List 或 Set
        List<Employee> employees = EmployeeData.getEmployees();
        List<Employee> employeeList = employees.stream().filter(e -> e.getSalary() > 6000).collect(Collectors.toList());
        employeeList.forEach(System.out::println);
        Set<Employee> employeeSet = employees.stream().filter(e -> e.getSalary() > 6000).collect(Collectors.toSet());
        employeeSet.forEach(System.out::println);
//        Collection<Employee> employeeCollection = employees.stream().filter(e -> e.getSalary() > 6000).collect(Collectors.toCollection());
//        employeeCollection.forEach(System.out::println);
    }
}
```

**Optinal 类：**为了在程序中避免出现空指针异常而创建的

```
 * 常用的方法：ofNullable(T t)
 *                orElse(T t)

public class OptionalTest {
    /*
      Optional.of(T t) : 创建一个 Optional 实例，t 必须非空；
      Optional.empty() : 创建一个空的 Optional 实例
      Optional.ofNullable(T t)：t 可以为 null
     */
    @Test
    public void test() {
        Girl girl = new Girl();
        //of(T t)：保证 t 是非空的
        Optional<Girl> optionalGirl = Optional.of(girl);//girl 为 null 时报错
    }

    @Test
    public void test2() {
        Girl girl = new Girl();
        girl = null;
        //ofNullable(T t)：t 可以为 null
```

```
        Optional<Girl> optionalGirl = Optional.ofNullable(girl);
        //orElse(T t)：如果当前的 Optional 内部封装的 t 是非空的，则返回内部的 t
        //如果内部的 t 是空的，则返回 orElse()方法中的参数 t
        Girl girl1 = optionalGirl.orElse(new Girl("宋万达"));
        System.out.println(optionalGirl);//Optional[Girl{name='null'}]     girl 为 null 时 ：
Optional.empty
        System.out.println(girl1);//Girl{name='宋万达'}
    }

    public String getGirlName(Boy boy) {
        return boy.getGirl().getName();
    }

    @Test
    public void test3() {
        Boy boy = new Boy();
        String girlName = getGirlName(boy);
        System.out.println(girlName);//空指针异常
    }

    //优化以后的 getGirlName 方法
    public String getGirlName1(Boy boy) {
        if (boy != null) {
            Girl girl = boy.getGirl();
            if (girl != null) {
                return girl.getName();
            }
        }
        return null;
    }

    @Test
    public void test4() {
        Boy boy = new Boy();
        String girlName = getGirlName1(boy);
        System.out.println(girlName);//null
    }

    //使用 Optional 类的 getGirlName():
    public String getGirlName3(Boy boy) {
        Optional<Boy> boyOptional = Optional.ofNullable(boy);
        Boy boy1 = boyOptional.orElse(new Boy(new Girl("宋万达")));
        Girl girl = boy1.getGirl();
        Optional<Girl> girlOptional = Optional.ofNullable(girl);
```

```
        //girl1 一定非空
        Girl girl1 = girlOptional.orElse(new Girl("宋千达"));
        return girl1.getName();
    }
    @Test
    public void test5(){
        Boy boy = null;
        Boy boy1 = null;
        boy1 = new Boy();//*
        String girlName = getGirlName3(boy);
        String girlName1 = getGirlName(boy1);
        System.out.println(girlName);//宋万达
//        System.out.println(girlName1);//宋千达
    }
}

public class OptionalTest {
    @Test
    public void test1(){
        //empty():创建的 Optional 对象内部的 value = null
        Optional<Object> op1 = Optional.empty();
        if(!op1.isPresent()){ //Optional 封装的数据是否包含数据
            System.out.println("数据为空");
        }
        System.out.println(op1);//Optional.empty
        System.out.println(op1.isPresent());//false
        //如果 Optional 封装的数据 value 为空，则 get()报错。否则，value 不为空时，返回。
        System.out.println(op1.get());//报错
    }
    @Test
    public void test2(){
        String str = "hello";
        Optional<String> op1 = Optional.of(str);
        //get()通常与 of()方法搭配使用。y 用于获取内部的封装的数据 value
        String str1 = op1.get();//
        System.out.println(str1);//hello
    }
    @Test
    public void test3(){
        String str = "beijing";
        Optional<String> op1 = Optional.ofNullable(str);
        String str2 = op1.orElse("shanghai");
        System.out.println(str2);//beijing
    }
```