



面向对象的软件构造导论

第八章 流与输入输出



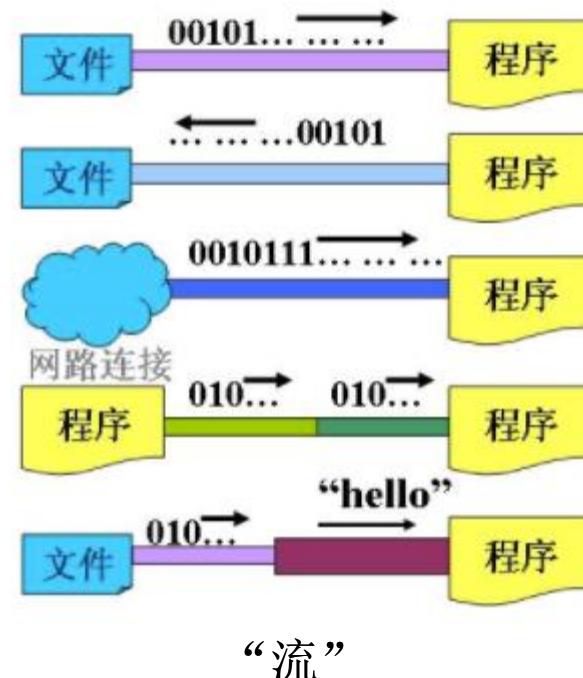
课程导航

- 流
- 输入输出流
- Java流继承框架
- 操作文件
- 对象输入/输出流与序列化
- 数据访问对象模式



流

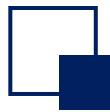
- 流是个抽象的概念：流是一组有序的数据序列，将数据从一个地方带到另一个地方
- 输入输出设备的抽象
 - Java程序中，对于数据的输入/输出操作都是以“流”的方式进行。
 - 设备可以是文件，网络，内存等。





□ 流的分类

- 按照流的方向主要分为输入流和输出流两大类。
- 数据流按照数据单位的不同分为字节流和字符流。
 - 字节流：基本单位为字节
 - 字符流：基本单位为字符
- 按照功能可以划分为节点流和处理流。
 - 节点流：可以从或向一个特定的地方（节点）读写数据，与数据源直接相连
 - 处理流：通过一个间接流类去调用节点流类（用来包装节点流），以达到更加灵活方便地读写各种类型的数据

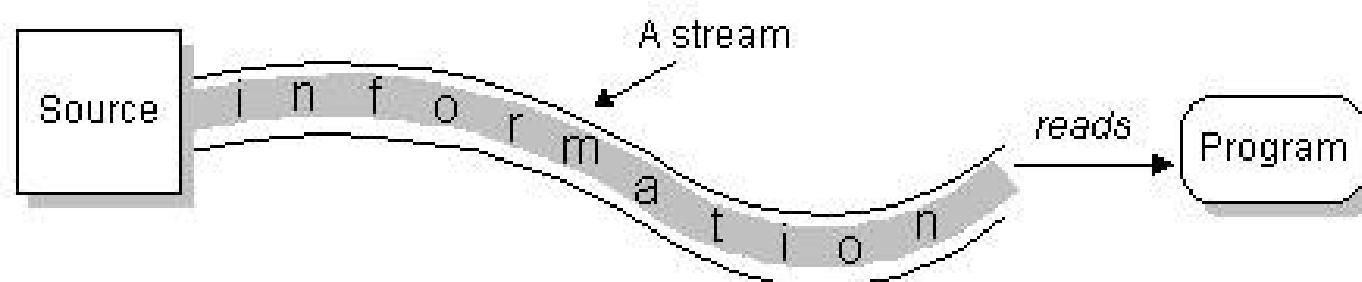


流

□ 流具有方向性，至于是输入流还是输出流则是一个相对的概念，一般以程序为参考：

- 如果数据的流向是程序至设备，我们称之为输出流，
- 反之，我们称之为输入流。

□ 可以将流想象成一个“水流管道”，水流就在这管道中形成了，自然就出现了方向的概念。





课程导航

- 流
- 输入输出流
- Java流继承框架
- 操作文件
- 对象输入/输出流与序列化
- 数据访问对象模式



Java系统流

- 每个Java程序都带有一个系统流，系统流对应的类为 `java.lang.System`
- 包含三个子类：
 - **System.in**: 标准输入流，默认设备是键盘。
 - **System.out**: 标准输出流，默认设备是控制台。
 - **System.err**: 标准错误流，默认设备是控制台。



读取控制台输入

- Java的控制台输入由System.in完成。
- 为了获得一个绑定到控制台的字符流，你可以把System.in包装在一个BufferedReader对象中来创建一个字符流。
- 下面是创建BufferedReader的基本语法：

```
BufferedReader br = new BufferedReader(new  
    InputStreamReader(System.in));
```

- BufferedReader对象创建后，我们便可以使用read()方法从控制台读取一个字符，或者用readLine()方法读取一个字符串。



从控制台读取多字符输入

- 下面的程序示范了用**read()**方法从控制台不断读取字符直到用户输入q。

```
//使用 BufferedReader 在控制台读取字符
import java.io.*;
public class BRRead {
    public static void main(String[] args) throws IOException {
        char c;
        // 使用 System.in 创建 BufferedReader
        BufferedReader br = new BufferedReader(new
                                         InputStreamReader(System.in));
        System.out.println("输入字符，按下 'q' 键退出。");
        // 读取字符
        do {
            c = (char) br.read();
            System.out.println(c);
        } while (c != 'q');
    }
}
```

System.out包含方法：
• **print()**: 不换行
• **println()**: 换行
• **write()**: 很少用



从控制台读取多字符输入

□ 以上实例编译运行结果如下：

1. 输入字符，按下 'q' 键退出。
2. runoob+回车+q+回车
- 3.>> r
- 4.>> u
- 5.>> n
- 6.>> o
- 7.>> o
- 8.>> b
- 9.>>
- 10.>>
- 11.>> q



从控制台读取字符串

- 下面的程序用**readline()**方法读取和显示字符行直到输入单词“end”。

```
//使用 BufferedReader 在控制台读取字符
import java.io.*;
public class BRReadLines {
    public static void main(String[] args) throws IOException {
        // 使用 System.in 创建 BufferedReader
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'end' to quit.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while (!str.equals("end"));
    }
}
```



从控制台读取字符串

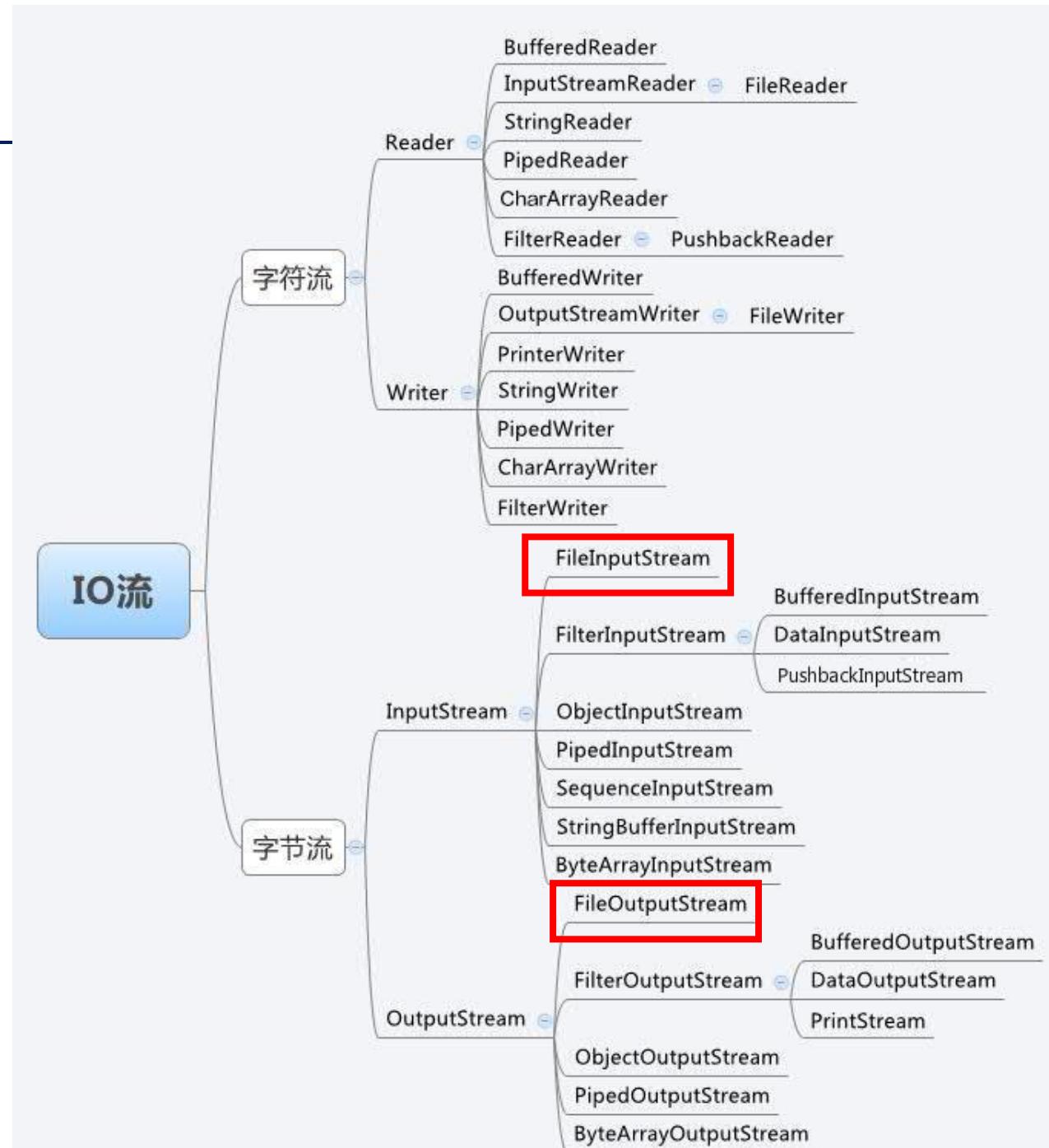
□ 以上示例编译运行结果如下：

```
1.Enter lines of text.  
2.Enter 'end' to quit.  
3.This is line one  
4.This is line one  
5.This is line two  
6.This is line two  
7.end
```



文件的输入输出

下面将要讨论的两个重要的流是 **FileInputStream** 和 **FileOutputStream**。





FileInputStream

- 该流用于从文件读取数据，它的对象可以用关键字 **new** 来创建。
- 有多种构造方法可用来创建对象。
 - 可以使用字符串类型的文件名来创建一个输入流对象来读取文件：

```
InputStream f = new FileInputStream("C:/java/hello");
```

- 也可以使用一个文件对象来创建一个输入流对象来读取文件。我们首先得使用 **File()** 方法来创建一个文件对象：

```
File f = new File("C:/java/hello");
```

```
InputStream in = new FileInputStream(f);
```



FileOutputStream

- 该类用来创建一个文件并向文件中写数据。
- 如果该流在打开文件进行输出前，目标文件不存在，那么该流会创建该文件。
- 有两个构造方法可以用来创建 **FileOutputStream** 对象。
 - 使用字符串类型的文件名来创建一个输出流对象：

```
OutputStream f = new FileOutputStream("C:/java/hello");
```

- 使用一个文件对象来创建一个输出流来写文件。我们首先得使用 **File()** 方法来创建一个文件对象：

```
File f = new File("C:/java/hello");
OutputStream fOut = new FileOutputStream(f);
```



文件输入输出实例

```
1.import java.io.*;
2.public class fileStreamTest {
3.    public static void main(String[] args) throws IOException {
4.        File f = new File("a.txt");
5.        FileOutputStream fop = new FileOutputStream(f);
6.        // 构建FileOutputStream对象,文件不存在会自动新建
7.
8.        OutputStreamWriter writer = new OutputStreamWriter(fop, "UTF-8");
9.        // 构建OutputStreamWriter对象,参数可以指定编码,默认为操作系统默认编码
10.
11.       writer.append("中文输入");
12.       // 写入到缓冲区
13.
14.       writer.append("\r\n");
15.       // 换行
16.
17.       writer.append("English");
18.
```



文件输入输出实例

```
19.         writer.close();
20.         // 关闭写入流,同时会把缓冲区内容写入文件,所以上面的注释掉
21.         fop.close();
22.         // 关闭输出流,释放系统资源
23.
24.         FileInputStream fip = new FileInputStream(f);
25.         // 构建FileInputStream对象
26.
27.         InputStreamReader reader = new InputStreamReader(fip, "UTF-8");
28.         // 构建InputStreamReader对象,编码与写入相同
29.
30.         StringBuffer strbuf = new StringBuffer();
31.         while (reader.ready()) {
32.             strbuf.append((char) reader.read());
33.             // 转成char加到StringBuffer对象中
34.         }
```



文件输入输出实例

```
35.     System.out.println(strbuf.toString());  
36.     reader.close();  
37.     // 关闭读取流  
38.  
39.     fip.close();  
40.     // 关闭输入流,释放系统资源  
41.  
42. }  
43. }
```



课程导航

- 流
- 输入输出流
- Java流继承框架
- 操作文件
- 对象输入/输出流与序列化
- 数据访问对象模式



完整的流家族

□ 与C语言只有单一类型**FIFE***包打天下不同，Java拥有一个流家族，包含各种输入/输出流类型，其数量超过60个！

□ IO流的分类

- 方向： `input/reader, output/writer`
- 数据：
 - 字节(`Byte, 8bit`): 类型多样，包括文本、图片、声音、视频等
 - 字符(`Character, 多数为16bit`): 仅限纯文本

读入word文档里的内容是？

- A 字符流
- B 字节流
- C 不确定

提交

读入word文档里的内容是？



A 字符流



B 字节流



C 不确定

提交



完整的流家族

Java所有的流类位于java.io包中，都分别继承自以下四种抽象流类型(四大家族)

	字节流	字符流
输入流	InputStream	Reader
输出流	OutputStream	Writer



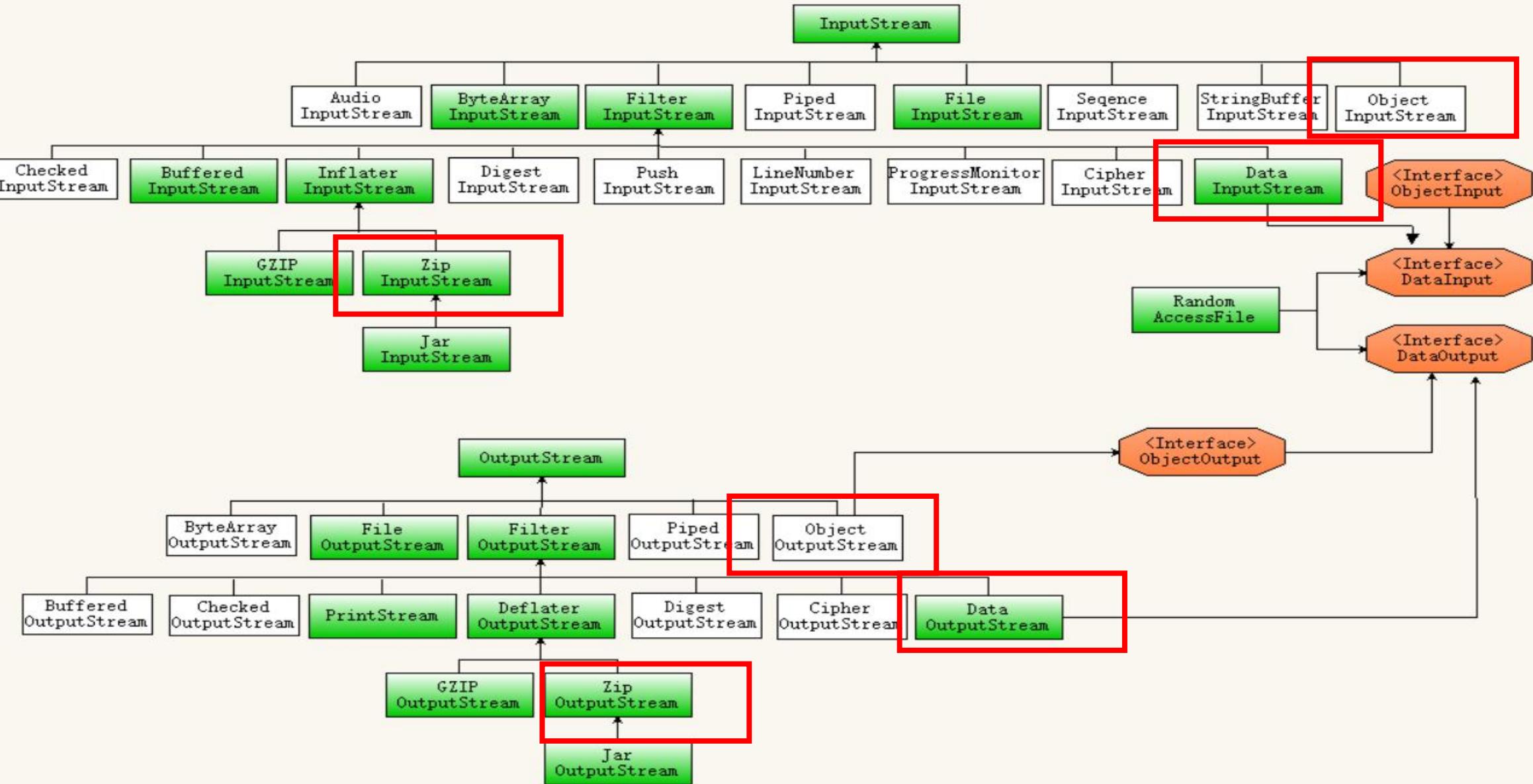
完整的流家族

- 在java中只要“类名”以Stream结尾的都是字节流，以“Reader/Writer”结尾的都是字符流
- 所有的流都实现了java.io.Closeable接口
 - 都是可关闭的，都有close()方法
 - 流毕竟是一个管道，用完之后要关闭，不然会耗费很多资源
 - 输出流有flush()
- 字节流与字符流是可转换的（例如本课件中第16-17页）

```
FileOutputStream fop = new FileOutputStream(f);
// 构建FileOutputStream对象，文件不存在会自动新建
OutputStreamWriter writer = new OutputStreamWriter(fop, "UTF-8");
// 构建OutputStreamWriter对象，参数可以指定编码，默认为操作系统默认编码
```



InputStream与OutputStream的层次结构



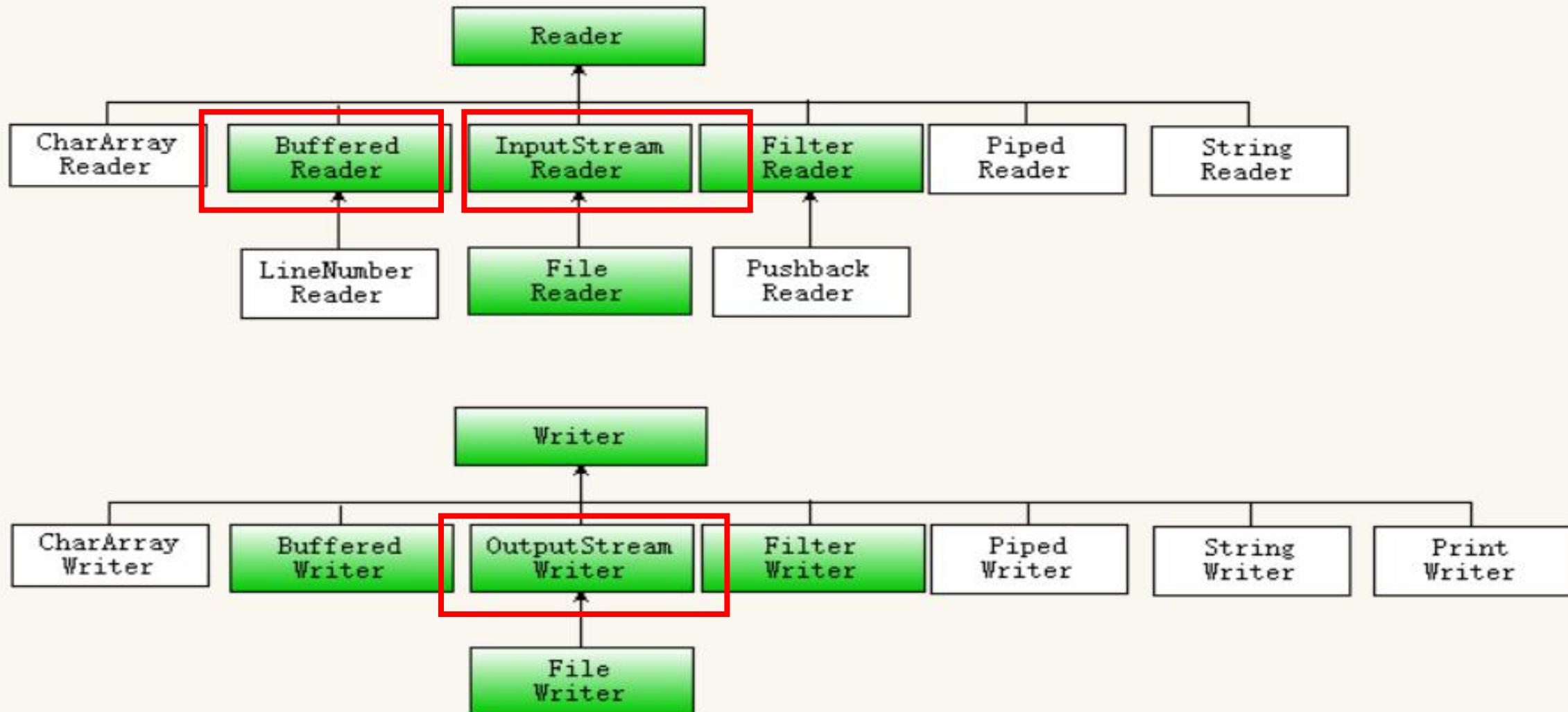


InputStream与OutputStream的层次结构

- **InputStream**和**OutputStream**类可以读写单个字节或字节数组，这些类构成了输入流与输出流的层次结构的基础。
- 要想读写字符串和数字，就需要功能更强大的子类，
 - 例如，**DataInputStream**和**DataOutputStream**可以以二进制格式读写所有 的基本Java类型。
- 还包含了多个很有用的输入/输出流，
 - 例如，**ZipInputStream**和**ZipOutputStream**可以用我们常见的ZIP压缩格式读写文件。



Reader和Writer的层次结构





Reader和Writer的层次结构

□ 对于**Unicode**文本，可以使用抽象类Reader和Writer的子类

- Reader/Writer类的基本方法与InputStream/OutputStream中的方法类似。
- Reader/Writer都是基于字符的

□ 常用的子类

- BufferedReader: 用来提高效率，例如包装InputStreamReader

```
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

- InputStreamReader/OutputStreamReader: 可包装 InputStream，从而将基于字节的输入流转换为基于字符的 Reader, 用于从文件中读取字符

```
InputStream inputStream = new FileInputStream("D:\\test\\1.txt");
Reader inputStreamReader = new InputStreamReader(inputStream);
```



课程导航

- 流
- 输入输出流
- Java流继承框架
- 操作文件
- 对象输入/输出流与序列化
- 数据访问对象模式



操作文件

- 前面小节已经介绍了如何从文件中读写数据，然而文件管理的内涵远比读写要广。
- **Path**和**Files**类封装了在用户机器上处理文件系统所需的所有功能。
 - 例如，**Files**类可以用来移除或重命名文件，或者在查询文件最后被修改的时间。
- 本小节主要讲述以下内容：
 - **Path**
 - 读写文件
 - 创建文件和目录
 - 复制、移动和删除文件
 - 获取文件信息



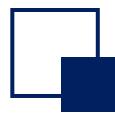
Path

- Path(路径) 表示的是一个目录名序列，其后还可以跟着一个文件名。
 - 以根部件（例如C:\）开始的路径是绝对路径；
 - 否则，就是相对路径。
- 下面我们分别创建一个绝对路径和一个相对路径：

```
Path absolute = Paths.get("c:\\data\\myfile.txt");

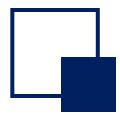
// Paths.get(basePath, relativePath)
// 假设当前基础路径“c:\\data”
Path relative = Paths.get("myfile.txt");
```

- 静态的Paths.get方法接受一个或多个字符串，并将它们用默认文件系统的路径分隔符（类UNIX文件系统是/，Windows是\\）连接起来。



读写文件

- **Files**是操作文件的工具类，包含了大量的方法
- **Files**类可以使得普通文件操作变得快捷。
- **Files**提供的读写方法，受内存限制，只能读写小文件。对于大型文件，还是需要文件流，每次只读写一部分文件内容。



读写文件（读）

□ 可以用下面的方式很容易地读取文件的所有内容：

- `byte[] bytes = Files.readAllBytes(path);`

□ 我们还可以以如下的方式从文本文件中读取内容：

- `var content = Files.readString(path, charset)`, 例如：

```
String content = Files.readString(Paths.get("/path/to/file.txt"), StandardCharsets.ISO_8859_1);
```

□ 但是如果希望将文件当作行序列读入，那么可以调用：

- `List<String> lines = Files.readAllLines(path, charset);`



读写文件（写）

□ 如果希望写出一个字符串到文件中，可以调用：

- `Files.writeString(path, content.charset);`

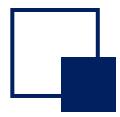
□ 向指定文件追加内容，可以调用：

- `Files.write(path, content.getBytes(charset), StandardOpenOption.APPEND);`

□ 还可以用下面的语句将一个行的集合写出到文件中：

- `Files.write(path, lines, charset);`

```
// 写入二进制文件:  
byte[] data = ...  
Files.write(Paths.get("/path/to/file.txt"), data);  
// 写入文本并指定编码:  
Files.writeString(Paths.get("/path/to/file.txt"), "文本内容...", StandardCharsets.ISO_8859_1);  
// 按行写入文本:  
List<String> lines = ...  
Files.write(Paths.get("/path/to/file.txt"), lines);
```



创建文件和目录

□ 创建新目录可以调用：

- `Files.createDirectory(path);`
- 其中，路径中除了最后一个部件外，其他部分都必须是已存在的。

□ 可以使用下面的语句创建一个空文件：

- `Files.createFile(path);`
- 如果文件已经存在了，那么这个调用就会抛出异常。



复制、移动和删除文件

- 将文件从一个位置复制到另一个位置可以直接调用：
 - `Files.copy(fromPath, toPath);`
- 移动文件(即复制并删除原文件)可以调用：
 - `Files.move(fromPath, toPath);`
- 如果目标路径已经存在，那么复制或移动将失败。如果想要覆盖已有的目标路径，可以使用**REPLACE_EXISTING**选项。如果想要复制所有的文件属性，可以使用**COPY_ATTRIBUTES**选项。也可以像下面这样同时选择这两个属性：
 - `Files.copy(fromPath, toPath, standardCopyOption.REPLACE_EXISTING, standardCopyOption.COPY_ATTRIBUTES);`



复制、移动和删除文件

- 可以将移动操作定义为原子性的，这样就可以保证要么移动操作成功完成，要么源文件继续保持在原来位置。具体可以使用**ATOMIC_MOVE**选项来实现：
 - `Files.move(fromPath, toPath, standardCopyOption.ATOMIC_MOVE);`
- 最后，删除文件可以调用：
 - `Files.delete(path);`
- 如果要删除的文件不存在，这个方法就会抛出异常。因此，可转而使用下面的方法：
 - `boolean deleted = Files.deleteIfExists(path);`
 - 该删除方法还可以用来移除空目录。



获取文件信息

□ 下面的静态方法都将返回一个**boolean**值，表示检查路径的某个属性的结果：

- exists()
- isHidden()
- isReadable(), isWritable(), isExecutate()
- isRegularFile(), isDirectory(), isSymbolicLink()
- Size()



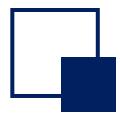
课程导航

- 流
- 输入输出流
- Java流继承框架
- 操作文件
- 对象输入输出流与序列化
- 数据访问对象模式



对象输入输出与序列化

- 当你需要存储相同类型的数据时，使用固定长度的记录格式是一个不错的选择。但是，在面向对象程序中创建的对象很少全部都具有相同的类型。例如，你可能有一个**Person**记录数组，但是实际上却包含诸如**Student**这样的子类实例。
- 我们当然可以自己设计出一种数据格式来存储这种多态集合，但幸运的是，我们并不需要这么做。

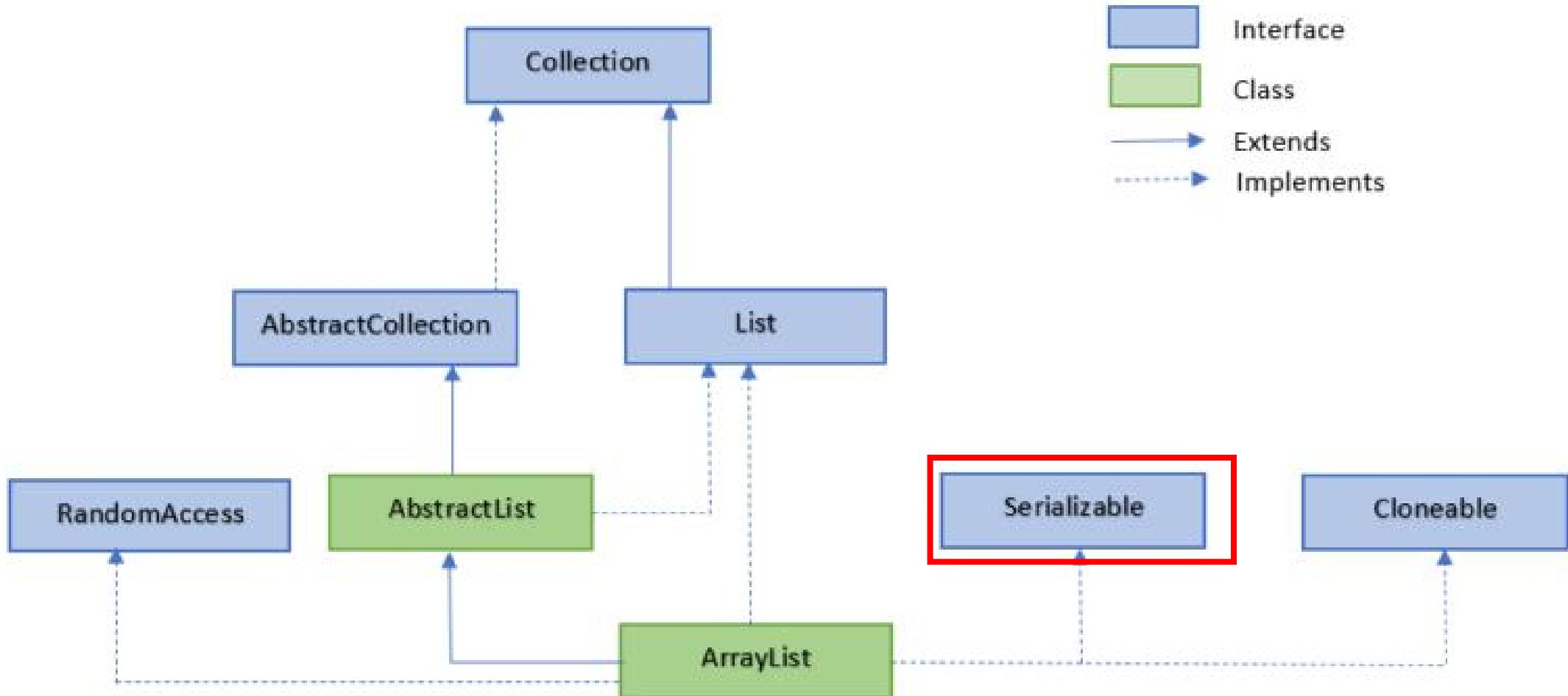


对象输入输出与序列化

- Java语言支持一种称为**对象序列化(object serialization)**的通用机制，它可以将任何对象写出到输出流中，并在之后将其读回。
- 序列化与反序列化
 - 把对象转换为字节序列的过程称为对象的序列化(**serialize**)。
 - 把字节序列恢复为对象的过程称为对象的反序列化(**deserialize**)。



ArrayList类





对象序列化用途

- 把对象的字节序列永久地保存到硬盘上，通常存放在文件中。
 - 在很多应用中，需要对某些对象进行序列化，让它们离开内存空间，进入物理硬盘，以便长期保存。比如最常见的**Web**服务器中的**Session**对象，当有**10**万用户并发访问，就有可能出现**10**万个**Session**对象，内存可能吃不消，于是**Web**容器就会把一些**Session**先序列化到硬盘中，等要用了，再把保存在硬盘中的对象还原到内存中。
- 在网络上传送对象的字节序列。
 - 当两个进程在进行远程通信时，彼此可以发送各种类型的数据。
 - 无论是何种类型的数据，都会以二进制序列的形式在网络上传送。
 - 发送方需要把这个**Java**对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为**Java**对象。



JDK类库中的序列化API

- **Java.io.ObjectOutputStream**代表对象输出流，它的**writeObject(Object obj)**方法可对参数指定的**obj**对象进行序列化，把得到的字节序列写到一个目标输出流中。
- **ObjectOutputStream(OutputStream out)**
 - 创建一个**ObjectOutputStream**使得你可以将对象写出到指定的**OutputStream**。
- **Void writeObject(Object obj)**
 - 写出指定的对象到**ObjectOutputStream**，这个方法将存储指定对象的类、类的签名以及这个类及其超类中所有非静态和非瞬时的域的值。
- 对象序列化步骤如下：
 - 创建一个对象输出流，它可以包装一个其他类型的目标输出流，如文件输出流；
 - 通过对象输出流的**writeObject()**方法写对象；



JDK类库中的序列化API

- **Java.io.ObjectInputStream**代表对象输入流，它的**readObject()**方法可从一个源输入流中读取字节序列，再把它们反序列化为一个对象，并将其返回。
- **ObjectInputStream(InputStream in)**
 - 创建一个**ObjectInputStream**用于从指定的**InputStream**中读回对象信息。
- **Object readObject()**
 - 从**ObjectInputStream**中读入一个对象。特别是，这个方法会读回对象的类、类的签名以及这个类及其超类中所有非静态和非瞬时的域的值。它指定的反序列化允许恢复多个对象引用。
- 对象反序列化步骤如下：
 - 创建一个对象输入流，它可以包装一个其他类型的源输入流，如文件输入流；
 - 通过对象输入流的**readObject()**方法读取对象；



序列化案例

□ 定义了如下的Person类，该类实现了Serializable 接口

```
public class Person implements Serializable {  
    private String name;  
    private int age;  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
    //为了方便查看对象，重写默认的toString()  
    @Override  
    public String toString(){  
        return "Person{" + "name=\"" + name + "\"" + ", age=" + age + '}';  
    }  
}
```



序列化案例

□ 调用ObjectOutputStream对象的writeObject输出可序列化对象

```
public class SeriDemo {  
    public static void main(String[] args) {  
        // ObjectOutputStream 流  
        try {  
            Person p1 = new Person("zhangsan", 30);  
            System.out.println(p1);  
            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.dat"));  
            oos.writeObject(p1);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```



序列化案例

口调用ObjectInputStream对象的readObject()得到序列化的对象

```
//创建一个ObjectInputStream输入流
try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.dat"))){
    System.out.println("readObject(): ");
    Person zhangsan = (Person) ois.readObject();
    System.out.println(zhangsan);
} catch (Exception e) {
    e.printStackTrace();
}
```



课程导航

- 流
- 输入输出流
- Java流继承框架
- 操作文件
- 对象输入/输出流与序列化
- 数据访问对象模式



数据访问对象模式

- 数据访问对象模式(Data Access Object Pattern)或DAO模式用于把低级的数据访问操作从高级的业务服务中分离出来。
- 数据访问对象模式的参与者：
 - 数据访问对象接口：该接口定义了在一个模型对象上要执行的标准操作。
 - 数据访问对象实体类：该类实现了上述的接口，负责从数据源获取数据，数据源可以是数据库，也可以是 xml，或者是其他的存储机制。
 - 模型对象/数值对象：该对象是简单的普通对象，包含了 get/set 方法来存储通过使用 DAO 类检索到的数据。



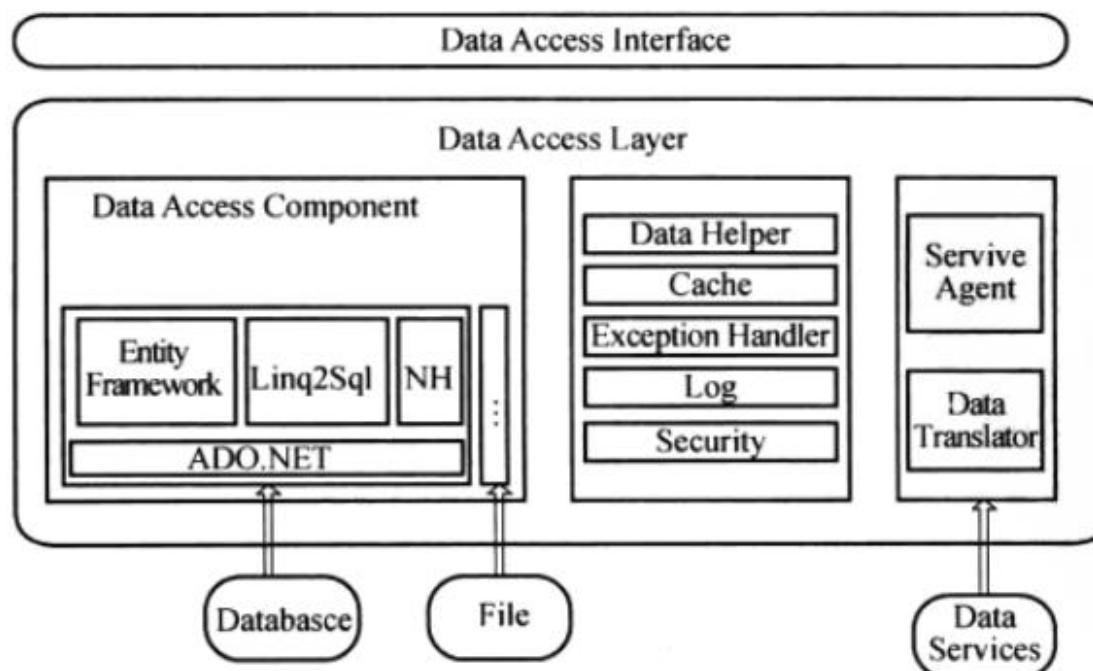
数据访问对象模式

□ 数据访问对象模式优点：

- 隔离数据层：由于新增了**DAO**层，不会影响到服务或者实体对象与数据库交互，发生错误会在该层进行异常抛出。

□ 缺点：

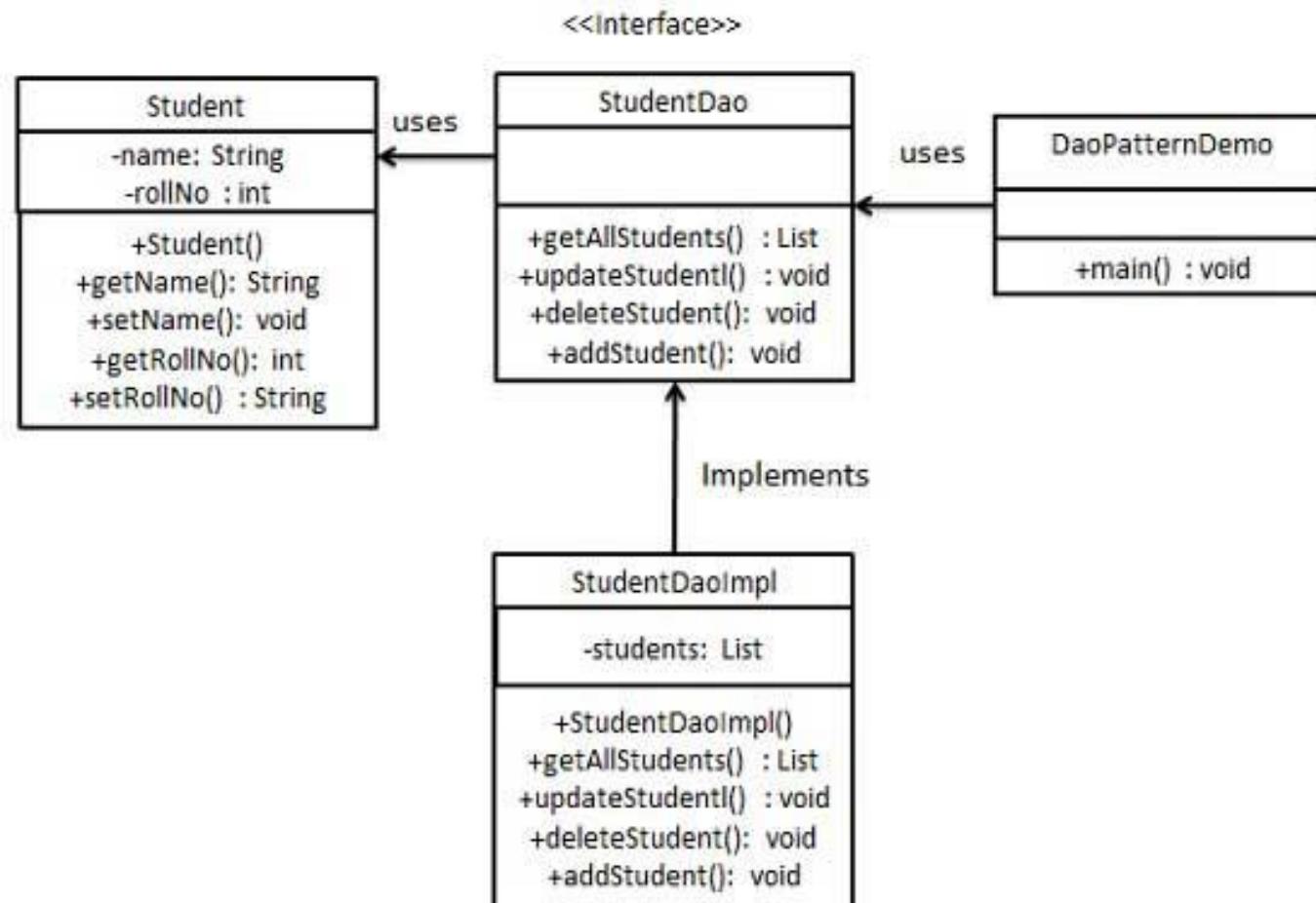
- 代码量增加：增加一层，代码量增加（不过该缺点实际中可忽略）





数据访问对象模式实现

- 我们将创建一个作为模型对象或数值对象的 **Student** 对象。 **StudentDao** 是数据访问对象接口。**StudentDaoImpl** 是实现了数据访问对象接口的实体类。
- DaoPatternDemo**, 我们的演示类使用 **StudentDao** 来演示数据访问对象模式的用法。





数据访问对象模式实现

步骤一：创建数值对象

```
1. public class Student {  
2.     private String name;  
3.     private int rollNo;  
4.  
5.     Student(String name, int rollNo){  
6.         this.name = name;  
7.         this.rollNo = rollNo;  
8.     }  
9.  
10.    public String getName() {  
11.        return name;  
12.    }
```

```
13.    public void setName(String name) {  
14.        this.name = name;  
15.    }  
16.  
17.    public int getRollNo() {  
18.        return rollNo;  
19.    }  
20.  
21.    public void setRollNo(int rollNo) {  
22.        this.rollNo = rollNo;  
23.    }  
24.}
```



数据访问对象模式实现

步骤二：创建数据访问对象接口

```
1. public interface StudentDao {  
2.     ...  
3.     List<Student> getAllStudents();  
4.     ...  
5.     Student getStudent(int rollNo);  
6.     ...  
7.     void updateStudent(Student student);  
8.     ...  
9.     void deleteStudent(Student student);  
10.}
```



数据访问对象模式实现

步骤三：创建实现了上述接口的实体类

```
1. public class StudentDaoImpl implements StudentDao {  
2.     //列表是当作一个数据库  
3.     List<Student> students;  
4.  
5.     public StudentDaoImpl() {  
6.         students = new ArrayList<Student>();  
7.         Student student1 = new Student("Robert", 0);  
8.         Student student2 = new Student("John", 1);  
9.         students.add(student1);  
10.        students.add(student2);  
11.    }  
12.    @Override  
13.    public void deleteStudent(Student student) {  
14.        students.remove(student.getRollNo());  
15.        System.out.println("Student: Roll No " + student.getRollNo() + ",  
16. deleted from database");  
17.    }  
18.}
```



数据访问对象模式实现

步骤三：创建实现了上述接口的实体类

```
17.//从数据库中检索学生名单
18.    @Override
19.    public List<Student> getAllStudents() {
20.        return students;
21.    }
22.
23.    @Override
24.    public Student getStudent(int rollNo) {
25.        return students.get(rollNo);
26.    }
27.
28.    @Override
29.    public void updateStudent(Student student) {
30.        students.get(student.getRollNo()).setName(student.getName());
31.        System.out.println("Student: Roll No " + student.getRollNo()
32.                           + ", updated in the database");
33.    }
34.}
```



数据访问对象模式实现

步骤四：创建StudentDao来演示数据访问对象模式的用法

```
1. public class DataAccessObjectPatternDemo {  
2.     public static void main(String[] args) {  
3.         StudentDao studentDao = new StudentDaoImpl();  
4.         //输出所有的学生  
5.         for (Student student : studentDao.getAllStudents()) {  
6.             System.out.println("Student: [RollNo : "  
7.                     + student.getRollNo() + ", Name : " + student.getName() + "]");  
8.         }  
9.         System.out.println();  
10.        //更新学生  
11.        Student student = studentDao.getAllStudents().get(0);  
12.        student.setName("Michael");  
13.        studentDao.updateStudent(student);  
14.        System.out.println();  
15.        //获取学生  
16.        studentDao.getStudent(0);  
17.        System.out.println("Student: [RollNo : " + student.getRollNo() + ",  
18.                           Name : " + student.getName() + "]");  
19.    }
```



数据访问对象模式实现

□ 步骤五：验证输出

```
1.Student: [RollNo : 0, Name : Robert ]  
2.Student: [RollNo : 1, Name : John ]  
3.Student: Roll No 0, updated in the database  
4.Student: [RollNo : 0, Name : Michael ]
```



课程导航

- 流
- 输入输出流
- Java流继承框架
- 操作文件
- 对象输入/输出流与序列化
- 数据访问对象模式