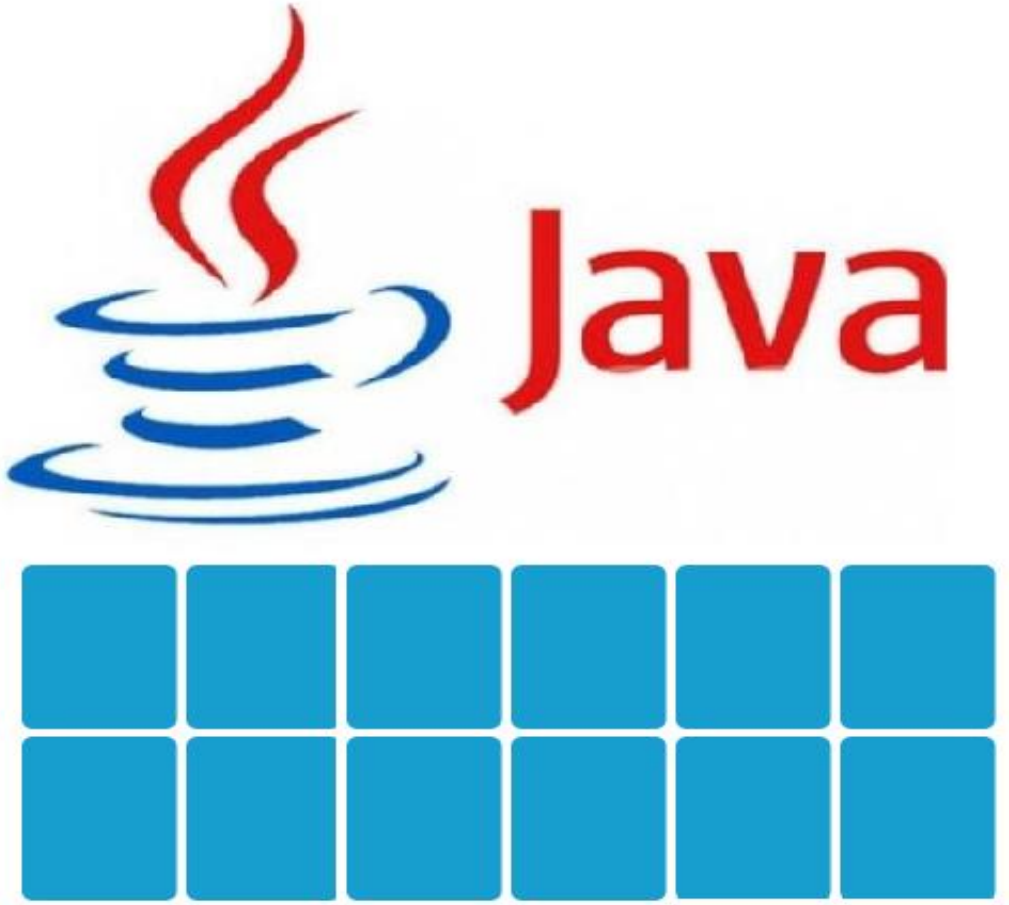


第九章

I/O流与文件操作

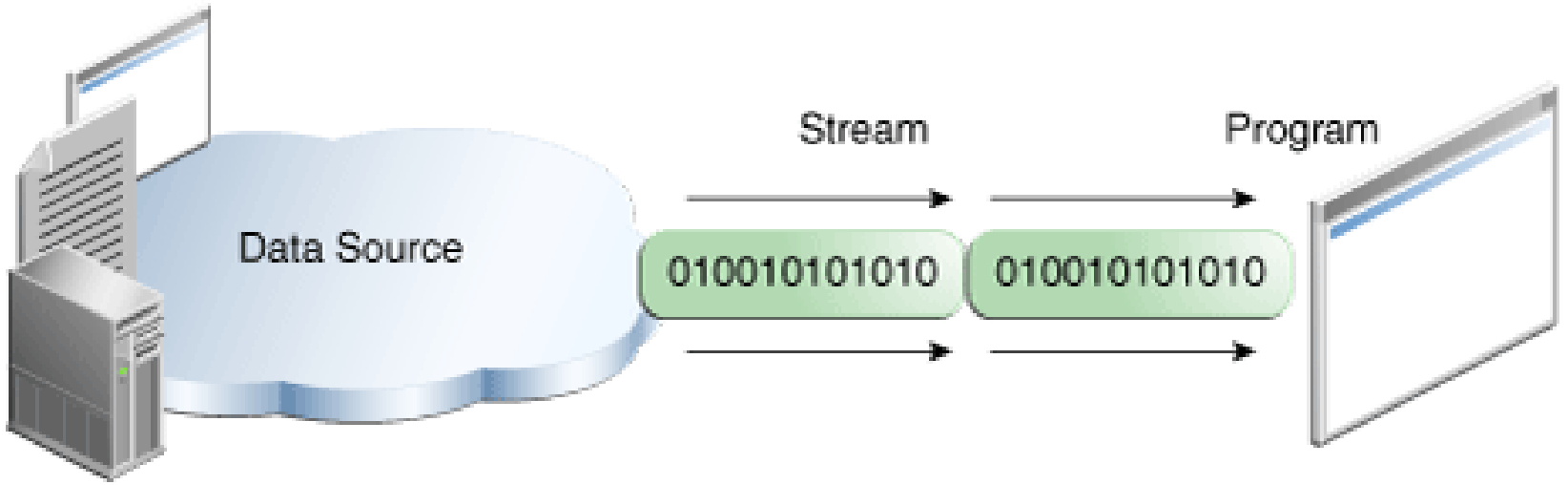
東北林業大學
NORTHEAST FORESTRY UNIVERSITY

I/O Streams

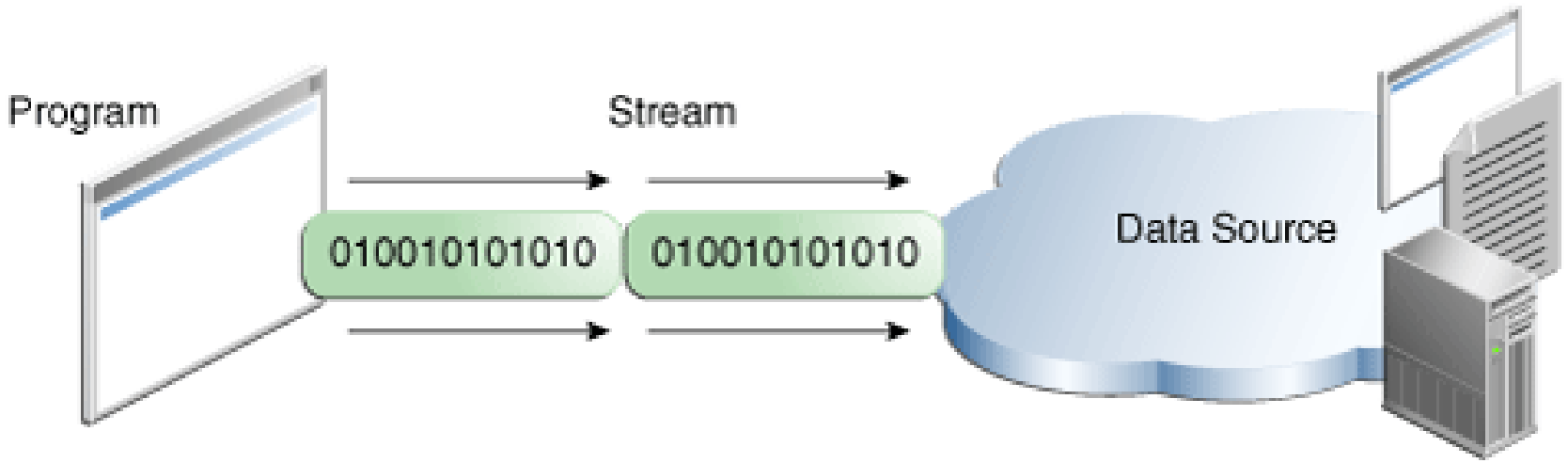
- An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

- IO流，可以表示不同类型的输入源与输出目标
- 输入源和输出目标，可以是保存，生成或使用数据的：磁盘文件，外围设备，远程网络等等

程序基于输入流(Input Stream)从源读取数据



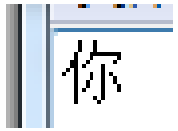
程序基于输出流(Output Stream)将数据写入目标

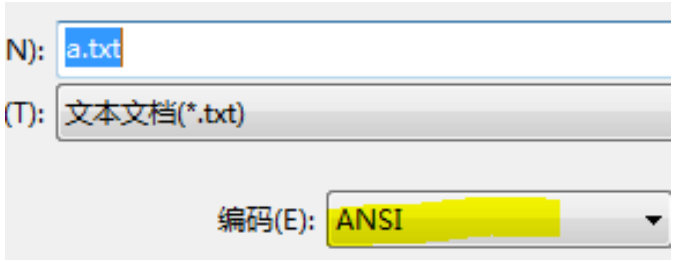


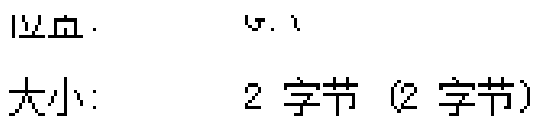
- Input Stream输入流，用于从数据源读取数据
- Output Stream输出流，用于将数据写入输出目标

- IO流，支持不同类型的数据，简单字节/原始数据类型/本地化字符/对象等等
- IO流，将不同的输入/输出，已相同的方式操作；创建的不同类型的流，有不同的实现方式，不同类型的流又有各自特有的操作方式
- 无论内部如何工作，所有IO流呈现出的都是相同的，简单的模式：程序中流入或流出的一系列数据
- 即，IO流，是数据源/数据目标，输入/输出的抽象表示

- 数据文件是基于byte而非bit，保存以及传输
- 1个字节(byte) == 8个二进制位(bit)
- 8个二进制位(bit)，可以用 十进制整数表示
- 即，1个字节可以用255以内的十进制整数表示
- 1个字符(char)，占的字节与编码有关。英文字符占1个字节；中文字符，在UTF8中占3个字节，在GBK中占2个字节



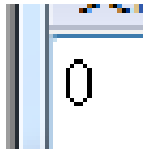




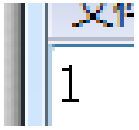
中文Win7默认使用GBK字符集

中文Win10系统默认使用UTF8字符集

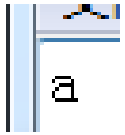
数字0，由整数48表示



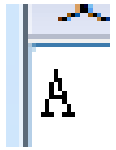
读取字节的十进制整数：48



卖取字节的十进制整数：49



读取字节的十进制整数：97



读取字节的十进制整数：65

读取字节的十进制整数：196

读取字节的十进制整数：227

GBK中文占用2个字节

由2个255内的十进制整数表示

- Java.io.InputStream抽象类，输入流操作的超类，支持子类以基本字节的方式操作二进制数据
  - int read() throws IOException，抽象方法，由具体子类实现。返回流中下一字节(必然是0-255之间的整数表示)，如果到达流末没有可读字节，返回-1
  - 基本子类，ByteArrayInputStream，FileInputStream等
- Java.io.OutputStream抽象类
  - void write(int b) throws IOException，抽象方法。将十进制数按字节，写入输出流
  - 基本子类，ByteArrayOutputStream，FileOutputStream，等
- 基于单字节的基本read()/write()方法，仅用于理解IO流执行过程，实际开发不会使用

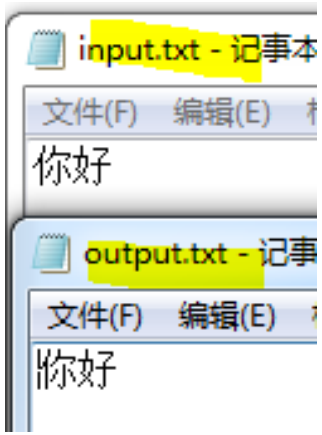
```java
private static void getByteStreams() throws IOException {
    FileInputStream in = new FileInputStream( name: "G:/input.txt");
    FileOutputStream out = new FileOutputStream( name: "G:/output.txt");
    int c;
    while ((c = in.read()) != -1) {
        System.out.println("读取字节的十进制整数: " + c);
        out.write(c);
    }
    in.close();
    out.close();
}
```

从输入流中循环读取字节
直到返回-1

基于指定文件创建输入流
基于指定文件创建输出流
输入/输出流的构造函数，均抛出IOException

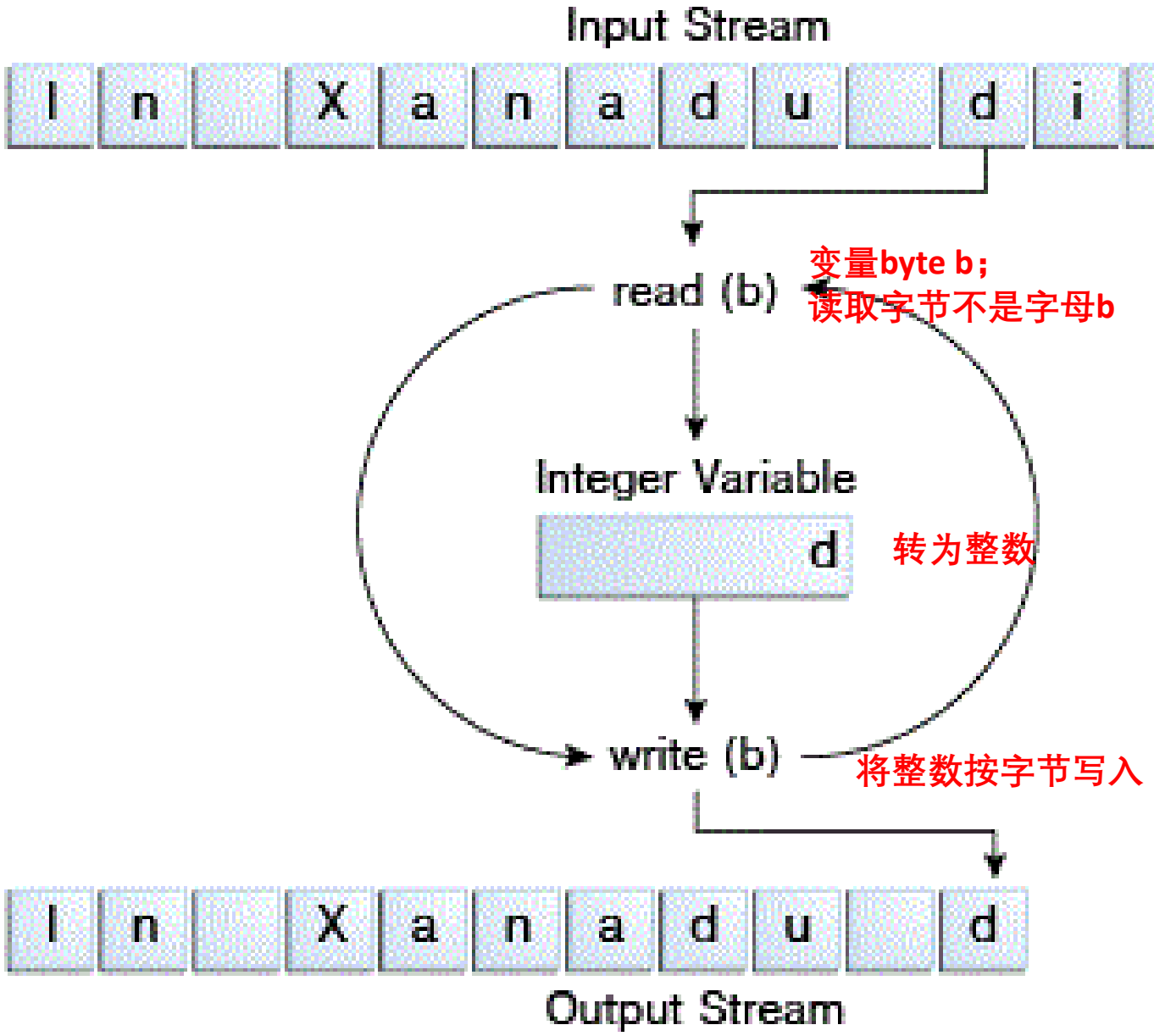
将十进制整数按字节
循环追加到输出流中保存

输入输出流
！必须关闭资源！



FileInputStream类的构造函数

```java
public FileInputStream( @NotNull String name) throws FileNotFoundException {
    this(name != null ? new File(name) : null);
}
```


Input Stream
I n X a n a d u d i d
read (b)
变量byte b；
读取字节不是字母b
Integer Variable
d
转为整数
write (b)
将整数按字节写入
I n X a n a d u d
Output Stream

Always Close Streams

- Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

- IO流，不会像其他对象，因失去引用而自动释放占用的内存资源。因此，所有IO流必须被正确关闭，否则会导致内存溢出
- 为确保无论是否出现异常，资源均被关闭，应在finally块中关闭资源

```java
private static void getByteStreams2() {
    FileInputStream in = null;
    FileOutputStream out = null;
    try {
        in = new FileInputStream( name: "G:/input.txt");
        out = new FileOutputStream( name: "G:/output.txt");
        int c;
        while ((c = in.read()) != -1) {
            out.write(c);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (in != null) {
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (out != null) {
            try {
                out.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
```

完整的，内部处理IO异常的代码

FileInputStream类的构造函数
抛出IO异常

read()/write()方法
抛出IO异常

IO流必须被关闭
因此置于finally块

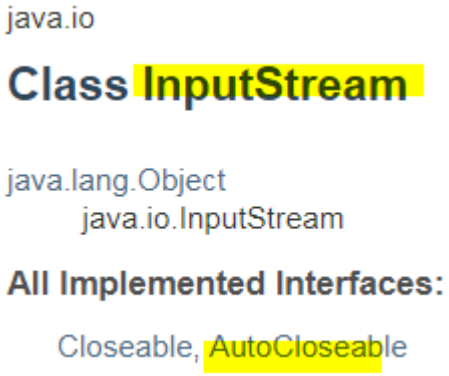
大量啰嗦冗余
与业务逻辑无关的
模板代码

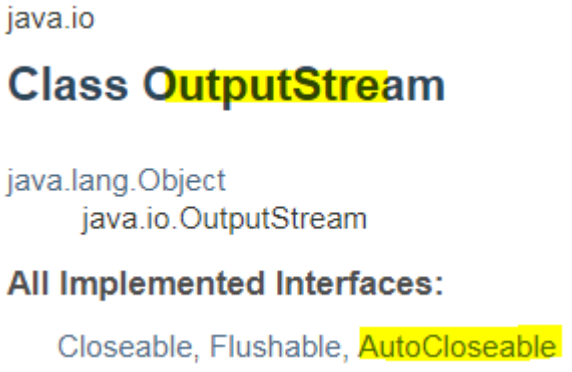
close()方法
抛出IO异常
因此必须继续处理

The try-with-resources Statement

- The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable, can be used as a resource.
- try-with-resources，在try语句中，声明需要关闭的资源，从而可以保证，无论try块是否引发异常，资源在try块结束后自动关闭(Java7)
- 任何实现了java.lang.AutoCloseable接口的类型，均是支持自动关闭的资源类型
- java.io.Closeable接口继承AutoCloseable接口
- 资源的自动关闭，与异常无关

- 原全部需要手动调用close()方法关闭的资源，全部支持try-with-resources自动关闭





在try语句()内，声明try中使用的资源
而非在try{}块内，声明
资源将在try语句结束后，自动关闭

```
try (FileInputStream in = new FileInputStream( name: "G:/input.txt");
     FileOutputStream out = new FileOutputStream( name: "G:/output.txt")) {
```

执行顺序测试

```
public class MyResource implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("Closed");
        ++++++++++++++++++++++++++
    try(MyResource r = new MyResource()) {
        System.out.println("Try");
        throw new Exception();
    } catch (Exception e) {
        System.out.println("Catch");
    } finally {
        System.out.println("Finally");
```

自定义支持自动关闭的资源

在try语句中
创建自动关闭资源对象

```
Try
Closed
Catch
Finally
```

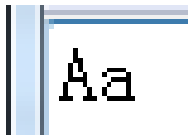
即使引发异常
依然先关闭资源
后执行异常处理
最后执行finally

- 在try语句中声明使用资源后，执行顺序
  - 无异常，在try块执行后，自动关闭资源，finally块
  - 引发异常，自动关闭资源，catch块，finally块

- try-with-resources语句，极大的简化了资源处理代码，使开发者无需关心资源状态，无需关心资源对象的创建顺序，无需关心资源对象的正确关闭方式

基于资源自动关闭简化的代码
无需finally块关闭资源

```
private static void getByteToChar() throws IOException {
    try (FileInputStream in = new FileInputStream( name: "G:/input.txt");
         FileOutputStream out = new FileOutputStream( name: "G:/output.txt")) {
        int c;
        while ((c = in.read()) != -1) {
            System.out.println("读取字节的十进制整数：" + c);
            System.out.println("将十进制整数转为字符：" + (char) c);
            out.write(c);
```




```
读取字节的十进制整数：65
将十进制整数转为字符：A
读取字节的十进制整数：97
将十进制整数转为字符：a
```

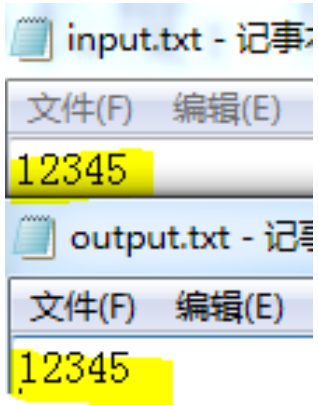
- Java9，进一步简化，在try语句中声明需关闭的资源即可。因此，IO流允许在try语句外创建

```
private static void getTryWithResources2() throws IOException {
    FileInputStream in = new FileInputStream( name: "G:/input.txt");
    FileOutputStream out = new FileOutputStream( name: "G:/output.txt");
    try (in; out) {
        int c;
        while ((c = in.read()) != -1) {
            System.out.println("读取字节的十进制整数：" + c);
            out.write(c);
```

但资源的自动关闭，与受检异常的显式处理无关
构造流受检异常的处理，依然是必须的

- Java.io.InputStream
  - int read(byte[] b)，将流中字节读取到**字节数组**b中，第1个字节置入数组0位置…，直到读取到数组b长度的字节位置为止；返回读取的字节长度；如果没有可读字节，返回-1
- Java.io.OutputStream
  - void write(byte[] b, int off, int len)，从**字节数组**b，off位置开始读取，至长度len结束；

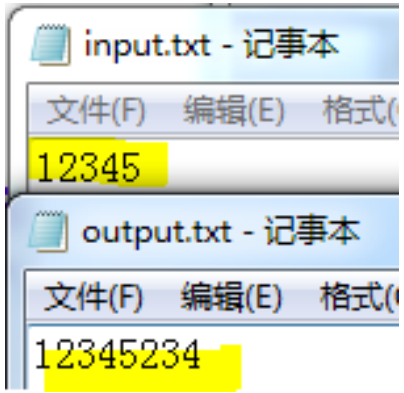
```java
try(FileInputStream in = new FileInputStream( name: "G:/input.txt");
    FileOutputStream out = new FileOutputStream( name: "G:/output.txt")) {
    byte[] buffer = new byte[4];
    int len = 0;
    while ((len = in.read(buffer)) != -1) {
        out.write(buffer,  off: 0,  len);
```



定义4字节长度的字符数组作为缓冲区
从流中读取字节数组长度的字节(4)
即，1234
写入字节数组缓冲区
从字节数组缓冲区的0位置读取
至read()本次读取的字节长度(4)
即，1234
写入输出流
从流中继续读取
读取到1个字节，写入缓冲区，len=1
从缓冲区的0位置读取1长度，即第0个元素，5
写入输出流
从流中继续读取，无字节返回-1，结束循环

使用没有指定字节数组起始位置的write()方法

```
in = new FileInputStream("G:/input.txt");
out = new FileOutputStream("G:/output.t
byte[] buffer = new byte[4];
int len = 0;
while ((len = in.read(buffer)) != -1) {
    out.write(buffer);
}
finally {
```



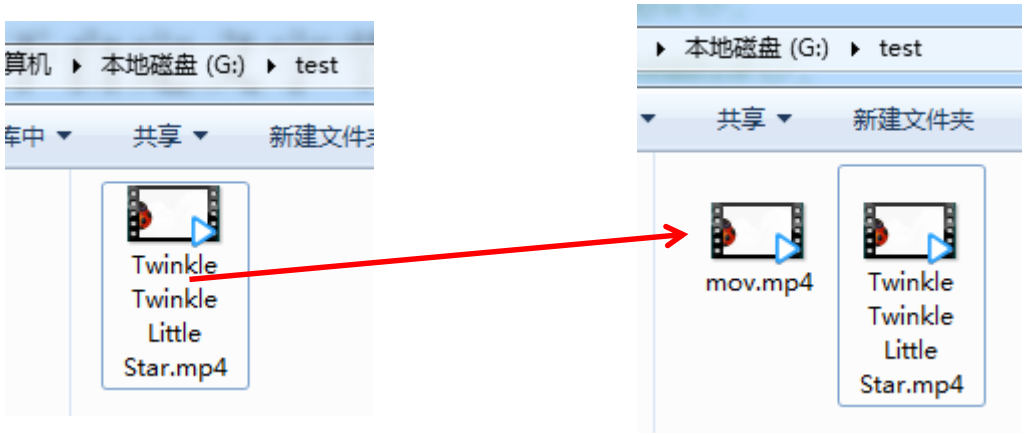
缓冲区4个字节，因此
第一次读取：1234
写入
最后一次读取5
由于没有自动清空数组的操作
因此仅覆盖数组中的数据
并将未覆盖位置数据一并写入

缓冲区字节数组设置过低
是为了演示循环读写操作
实际开发中应调整到合适的大小

InputStream to OutputStream

- 即使基于字节数组作为缓冲区，仍需考虑长度/读取/写入/位移等操作
- long transferTo(out) throws IOExecption，InputStream类中方法，支持直接将输入流“转移至”一个输出流，返回总字节长度(Java9)

```
private static void getTransferTo() throws IOException {
    try (FileInputStream is =
                 new FileInputStream( name: "G:/test/Twinkle Twinkle Little Star.mp4");
    FileOutputStream os = new FileOutputStream( name: "G:/test/mov.mp4")) {
        is.transferTo(os);
    }
```



transferTo()方法源码
默认使用8192长度的字节数组

```
 * @since 9
 */
ublic long transferTo(OutputStream out) throws IOException {
    Objects.requireNonNull(out, message: "out");
    long transferred = 0;
    byte[] buffer = new byte[DEFAULT_BUFFER_SIZE];
    int read;
    while ((read = this.read(buffer, off: 0, DEFAULT_BUFFER_SIZE)) >= 0) {
        out.write(buffer, off: 0, read);
        transferred += read;
    }
    return transferred;
```

与集合removeIf()方法相似
Java封装了复杂的实现操作
进一步简化开发

Character Streams

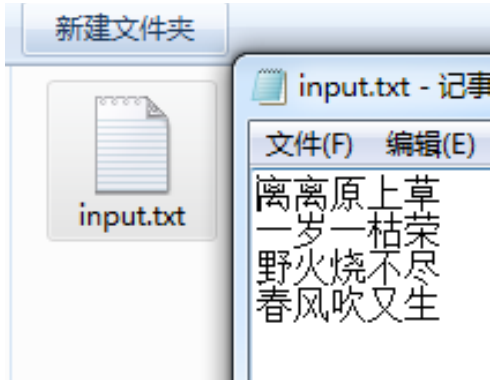
- Java提供了字符流操作(Character Streams)，支持直接读写文本文件内容，而无需手动完成字节到字符的转换
- BufferedReader/InputStreamReader等，基于缓冲区从字符输入流中读取文本(不再学习讨论)

- byte[] readAllBytes() throws IOException， InputStream类中方法，支持直接将流中所有字节读出到字节数组(Java9)
  - 将所有输入流中字节，一次读出到字节数组，没有缓冲区。因此，不应用于读取包含大量数据的输入流
- Files.readString(path)，处理文件操作时讨论

- 需求：按字符串，读取指定文本文件中的内容

```
try (FileInputStream is = new FileInputStream( name: "G:/test/input.txt")) {
    byte[] bytes = is.readAllBytes();
    System.out.println(new String(bytes));
```

String类支持
基于字节数组的构造函数
因此
可将文本文件读出为字节数组
基于字节数组创建字符串对象



中文win7下创建的txt文件
默认使用GBK编码
而，Java默认使用UTF-8编码字符集
乱码：当编码与解码不符时

```
.encoding=UTF-8
����Я�c�
h��h����
¥���η���
���綍����
```

```
try (FileInputStream is = new FileInputStream( name: "G:/test/input.txt
    byte[] bytes = is.readAllBytes();
    System.out.println(new String(bytes,  charsetName: "GBK"));
```

String支持声明字符集的构造函数

中文win10，默认使用UTF-8
因此无需声明

```
.encoding=(
离离原上草
一岁一枯荣
野火烧不尽
春风吹又生
```

- 当读写普通文件(任何二进制文件)时，使用基于字节数组缓冲区的基本输入输出流处理；或基于java9中方法

- 当读写文本文件时，使用BufferedReader等基于文本优化的字符输入输出流处理；或基于Java9中方法

What Is a Path?

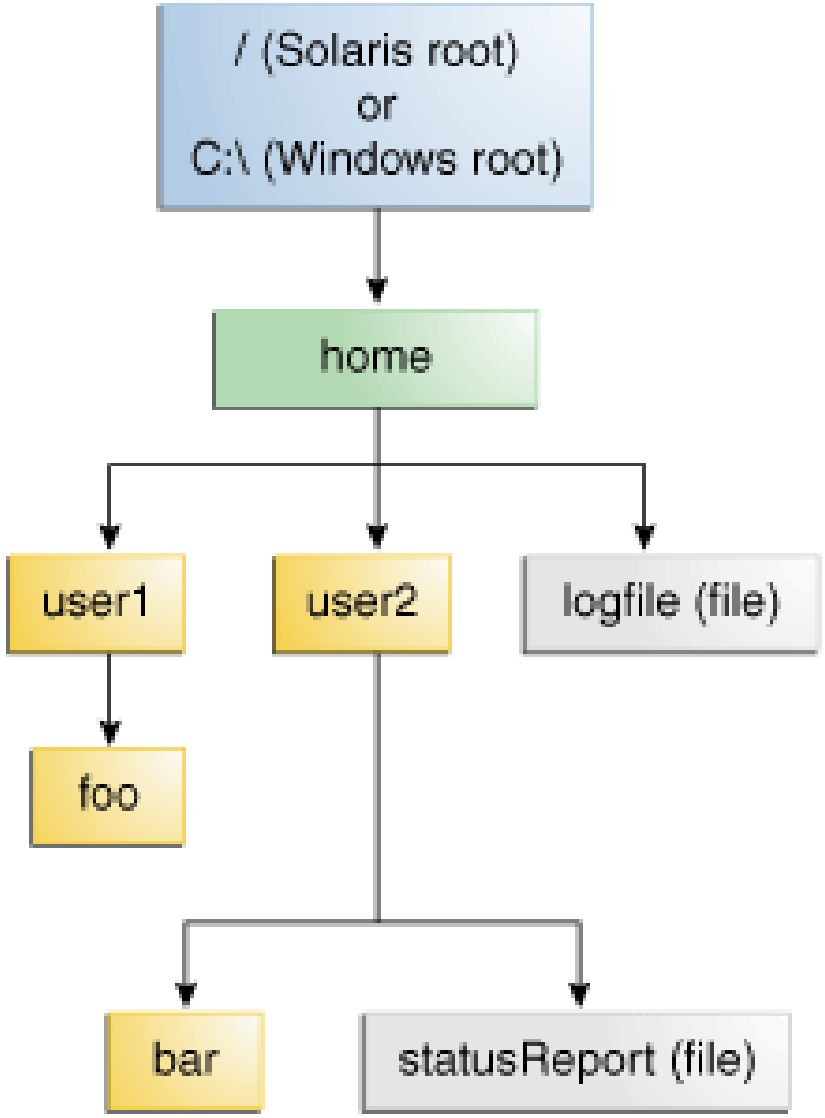
- A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most file systems in use today store the files in a tree (or hierarchical) structure.

- 文件系统，在某种形式的介质上存储和组织文件(例如，一个或多个硬盘驱动器)，以便于检索
- 目前，文件系统均以树型(或分层)结构存储文件
- 树顶部是一个或多个根节点，在根节点下，有文件和目录
- 每个目录可以包含文件和子目录，这些文件和子目录又可以包含文件和子目录等等，可能达到几乎无限的深度(windows下文件全名有长度限制)

- Linux
- /home/sally/statusReport

- Windows
- C:\home\sally\statusReport

- CC：路径建议使用“/”反斜杠表示
- 即windows下的路径也应描述为
- C:/home/sally/statusReport



- Absolute & Relative
- 绝对路径，始终包含根元素和查找文件所需的完整目录列表。 例如，D:/test/a.txt。找到文件所需的所有信息都包含在路径声明中

- 相对路径，例如，a.txt。没有更多信息，程序将无法访问。即，相对路径，最终也必须基于绝对路径描述

File I/O(Featuring NIO.2)

- Java.io.File类，包含耦合了文件路径声明，以及文件操作方法的类；且是同步阻塞的(不再学习讨论)

- NIO2 (java8)，将文件路径与文件操作，分离；且支持异步非阻塞
  - java.nio.file.Path接口，系统文件/目录的路径
  - java.nio.file.Files工具类，包含处理文件操作的方法，包括文件的，创建，删除，复制，移动等

Interface Path

- An object that may be used to locate a file in a file system. It will typically represent a system dependent **file path**.
- java.nio.file.Path

- Path接口，可以表示一个绝对的/相对的文件/目录的，路径
- Path代表一个不依赖于系统的文件路径。即运行在不同操作系统下，Path的具体实现不同(windows/linux)，但开发者仅需面向Path描述路径，不同系统，而无需关心操作系统差异*
- Path仅用于描述路径，不包含对指定路径的操作方法
- 相对路径不能以“/”开始。例如，
  - /example/a.txt，描述的是绝对路径
  - example/a.txt，是相对路径

- Creating a Path
- java.nio.file.Paths工具类，通过转换路径字符串或URI来创建Path对象
  - Paths类自动按系统的文件路径格式处理路径
  - Path get(String path)
  - Path get(Uri uri)
- Path.of(String path)，基于Path接口中的静态方法创建Path对象(Java11)

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
```

等效

```
Path p1 = Path.of( first: "/tmp/foo");
```

- 底层，自动基于当前运行的文件系统操作，实现文件与系统的解耦。即，相同的文件操作代码，可运行在不同系统上

Path.of()源码

```
public static Path of(String first, String... more) {
    return FileSystems.getDefault().getPath(first, more);
```

- 原IO下的File类，无基于运行系统自动转换路径实现

- Path定义了许多获取文件数据信息的方法
  - Path getFileName()，返回文件名或名称元素序列的最后一个元素。即，最后一个路径描述
  - Path getParent()，返回父目录的路径
  - Path getRoot()，返回路径的根

```
Path p = Path.of( first: "D:/test/input.txt");
System.out.println(p.getClass().getName());

System.out.println(p);
System.out.println(p.getFileName());
System.out.println(p.getParent());
System.out.println(p.getRoot());
```

不同文件系统的
具体实现

```
sun.nio.fs.WindowsPath
D:\test\input.txt
input.txt
D:\test
D:\
```

路径自动转为
默认Win系统的正斜杠

Path重写了toString()方法
因此可直接控制台输出
但以上方法的返回值不是string对象

- Joining Paths
  - Path resolve(Path other)方法，将路径拼接为一个新路径
  - Path Path.of(String... more)方法，基于参数数组实现
- Path支持路径比较等操作

```
Path dir = Path.of( first: "D:/test");
Path file = Path.of( first: "input.txt");
Path p1 = dir.resolve(file);
System.out.println(p1);

Path p2 = Path.of( first: "D:/test/input.txt");
System.out.println(p1.equals(p2));

Path p3 = Path.of( first: "D:",  ...more: "test", "input.txt");
System.out.println(p1.equals(p3));
```

重写了equals()方法

```
D:\test\input.txt
true
true
```

方法的路径拼接
无需手动添加目录的/正斜杠

```
Path base = Path.of( first: "D:/test");
Path file = Path.of( first: "input.txt");
System.out.println( base.toString());
System.out.println(file.toString());
System.out.println(base.toString() + file.toString());
System.out.println(base.resolve(file));
```

```
D:\test
input.txt
D:\testinput.txt
D:\test\input.txt
```

！错误！

自动为目录+文件路径的拼接
添加斜杠

基于path字符串拼接
与基于resolve()方法路径拼接
是完全不同的

因此，应正确使用resolve()方法实现拼接文件路径

File Operations

- java.nio.file.Files工具类，提供了丰富的静态方法，读取/写入/操作，文件与目录
- Files方法基于Path操作

- Checking a File or Directory
- Creating a File or Directory
- Copying a File or Directory
- Moving a File or Directory
- Deleting a File or Directory

- Checking a File or Directory
  - boolean exists(Path path)/notExists(Path path)，Path路径是否存在
  - Boolean isDirectory(Path path)，path是否为目录

```
private static final Path BASE_PATH = Path.of( first: "D:/test");
```

```
Path p = BASE_PATH.resolve("aaaaa");
System.out.println(Files.exists(p));
System.out.println("目录: " + Files.isDirectory(p));

System.out.println(Files.exists(BASE_PATH));
System.out.println("目录: " + Files.isDirectory(BASE_PATH));

Path p2 = BASE_PATH.resolve("input.txt");
System.out.println(Files.exists(p2));
System.out.println("目录: " + Files.isDirectory(p2));
```

```
false
目录 : false
true
目录 : true
true
目录 : false
```

- Creating a Directory
  - Path createDirectory(Path dir) throws IOException。目录路径已存在则异常；目录路径为多级目录，异常
  - Path createDirectories(Path dir) throws IOException。自动创建多级不存在目录；目录已存在，无异常

```
Path dir = Path.of( first: "D:/test/a/b/c");
Files.createDirectories(dir);
```

createDirectories()方法
创建自动创建所有不存在目录

- Creating a File
  - Path createFile(path) throws IOException。基于指定路径，创建文件。文件存在，异常

目录可能不存在
因此，先创建目录

```
Path dir = Files.createDirectories(BASE_PATH.resolve("a"));
Files.createFile(dir.resolve("a.txt"));
```

在D:/test下，创建a目录
在a目录下，创建a.txt文件

- Copying a File or Directory
  - Path copy(Path source, Path target, CopyOption... options) throws IOException，将文件复制到目标文件。默认，如果文件已经存在，异常
  - 如果source为目录，不会复制里面的文件，仅相当于创建一个空目录
  - java.nio.file.StandardCopyOption枚举，实现了CopyOption接口，复制选项

ATOMIC_MOVE
Move the file as an atomic file system operation.

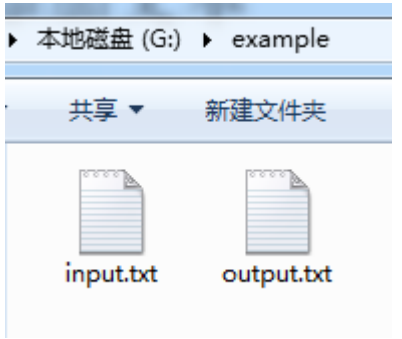
COPY_ATTRIBUTES
Copy attributes to the new file.

REPLACE_EXISTING
Replace an existing file if it exists.

将文件作为原子文件系统操作移动(多线程操作)
将属性复制到新文件
如果存在，替换现有文件

```
Path source = BASE_PATH.resolve("input.txt");
Path target = BASE_PATH.resolve("output.txt");
Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
```

覆盖文件



使用copy() 2个参数的默认方法
如果目标文件已经存在
异常无法复制

- Moving a File or Directory
  - Path move(Path source, Path target, CopyOption... options) throws IOException，将文件移动或重命名为目标文件。
  - 默认，如果目标文件存在，则异常，可通过options参数声明移动选项
  - 如果在本目录下移动，相当于文件改名

```
Path source = BASE_PATH.resolve("input.txt");
Path target = BASE_PATH.resolve("output.txt");
Files.move(source, target);
```

在相同目录下移动
则相当于将input.txt改名为output.txt

++++++++++++++++++++++++++

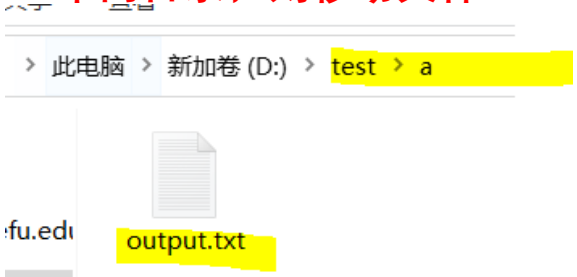
```
Path target = Path.of( first: "D:/test02");
Files.move(BASE_PATH, target);
```

将目录改名
与目录可以不为空

++++++++++++++++++++++++++

```
Path source = BASE_PATH.resolve("input.txt");
Path target = Path.of( first: "D:/test/a/output.txt");
// 目标目录可能不存在
Files.createDirectories(target.getParent());
Files.move(source, target);
```

不同目录，则移动文件



- Deleting a File or Directory
  - void delete(Path path) throws IOException。删除指定路径；路径不存在，异常
  - boolean deleteIfExists(Path path) throws IOException。路径不存在，不删除。返回是否删除成功
  - 如果路径为目录，目录中包含文件(即不为空)，2种删除均异常

```
Path path = BASE_PATH.resolve("input.txt");
Files.deleteIfExists(path);
```

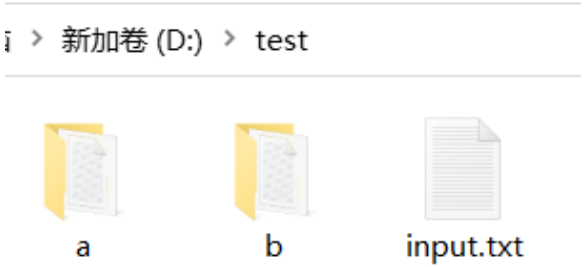
```
Path dir = Path.of( first: "D:/test");
Files.deleteIfExists(dir);
```

目录不为空，异常

```
javaagent:D:\idea\11b\idea_rt.jar=4934:D:\idea\bin -
java.nio.file.DirectoryNotEmptyException: D:\test
s.WindowsFileSystemProvider.implDelete(WindowsFileSy
```

需求删除目录，及
目录下的全部子目录及文件?

- Stream<Path> walk(Path start, int maxDepth) throws IOException
- Stream<Path> walk(Path start) throws IOException
- 遍历，基于指定深度遍历path路径中的文件
- 避免了基于IO File的递归调用




```
Path dir = Path.of( first: "D:/test");
Files.walk(dir).forEach(System.out::println);
System.out.println("---------");
Files.walk(dir,  maxDepth: 1).forEach(System.out::println);
```

```
D:\test
D:\test\a
D:\test\a\a.txt
D:\test\b
D:\test\b\b.txt
D:\test\input.txt
---------
D:\test
D:\test\a
D:\test\b
D:\test\input.txt
```

指定1级深度，则不会再进入内部的目录

- 需求：在指定目录下，检索指定文件，全部删除

```
Path file = Path.of( first: "a.txt");
Files.walk(BASE_PATH)
        .filter(p -> p.getFileName().equals(file))
        .forEach(p -> {
            try {
                Files.delete(p);
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
```

预删除文件为相对路径描述
因此，需要比较文件的名称

由于函数式接口没有声明抛出异常
因此Labmda表达式的结果也禁止抛出异常
所有受检异常必须在内部捕获处理

- 需求：删除指定的，包含文件/目录的整个文件目录

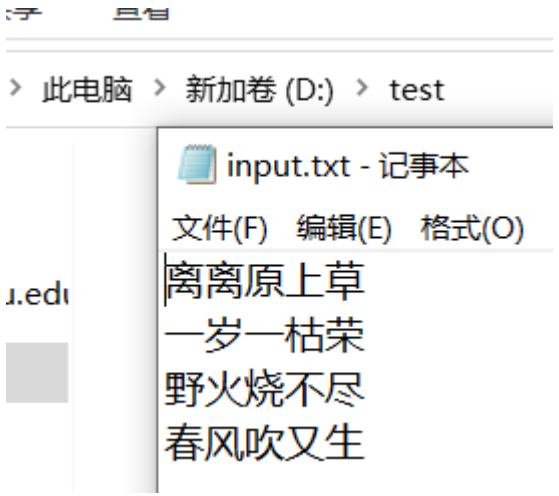
进入目录后逆向排序
先文件，后目录

```
Files.walk(BASE_PATH)
        .sorted(Comparator.reverseOrder())
        .forEach(p -> {
            System.out.println(p);
            try {
                Files.delete(p);
            } catch (IOException e) {
                e.printStackTrace();
            }
        });
```

```
D:\test\input.txt
D:\test\b\b.txt
D:\test\b
D:\test\a\a.txt
D:\test\a
D:\test
```

- 需求：按字符串，读取指定文本文件中的内容
- String Files.readString(path, charset) throws IOException，基于指定路径及字符集读取文本文件

```
String result = Files.readString(Path.of( first: "D:/test/input.txt"));
System.out.println(result);
```



离离原上草
一岁一枯荣
野火烧不尽
春风吹又生