* **基本IO操作**

RAF 提供了文件随机读写功能，但是没有提供复杂数据的读写功能。 Java 提供了另外一套可以灵活扩展的API： IO流。

IO流在节点流基础之上提供了丰富的的扩展功能，利用这些扩展流可以大大简化IO程序的开发。

**1. InputStream与OutputStream**

1.1. 输入与输出

流按照数据流向分为输入流和输入流：

**什么是输入**:输入是一个从外界进入到程序的方向，通常我们需要“读取”外界的数据时，使用输入。所以输入是用来读取数据的。

InputStream 代表输入流经常缩写为in/包含核心方法 read()/关闭方法 close()

**什么是输出**:输出是一个从程序发送到外界的方向，通常我们需要”写出”数据到外界时，使用输出。所以输出是用来写出数据的。

OutputStream 代表输出流经常缩写为out/包含核心方法 write()/关闭方法 close()

这两个流是抽象类，不能直接使用。在实际工作中使用其实现类：文件输出流 FileOutputStream

1.2. 节点流与处理流

按照流是否直接与特定的地方 (如磁盘、内存、设备等) 相连，分为节点流和处理流两类。

* **节点流**：可以从或向一个特定的地方（节点）读写数据。//节点流是流最原始的数据源，提供流最基本的功能。
* **处理流**：是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。//也称为高级流，过滤流，处理流是在其他流的基础之上扩展出更高级的功能。处理流的核心特点是必须依赖于另外一个流，自己不能独立功能，处理流是对另外的流进行扩展。处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。

**2. 文件流**

**2.1. 创建FOS对象(重写模式)**

FileOutputStream是文件的字节输出流，我们使用该流可以以字节为单位将数据写入文件。

构造方法:FileOutputStream(File file)

创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

构造方法：FileOutputStream(String filename):

创建一个向具有指定名称的文件中写入数据的输出文件流。

FileOutputStream fos = new FileOutputStream("demo.dat");

这里需要注意，若指定的文件已经包含内容，那么当使用FOS对其写入数据时，会将该文件中原有数据全部清除。

**2.2. 创建FOS对象(追加模式)**

通过上一节的构造方法创建的FOS对文件进行写操作时会覆盖文件中原有数据。若想在文件的原有数据之后追加新数据则需要以下构造方法创建FOS

构造方法:FileOutputStream(File file,boolean append)

创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

例如:

File file = new File("demo.dat"); FileOutputStream fos = new FileOutputStream(file,true);

构造方法：FileOutputStream(String filename,boolean append):

创建一个向具有指定名称的文件中写入数据的输出文件流。

FileOutputStream fos = new FileOutputStream("demo.dat",true);

以上两个构造方法中，第二个参数若为true,那么通过该FOS写出的数据都是在文件末尾追加的。

**2.3. 创建FIS对象**

FileInputStream是文件的字节输入流，我们使用该流可以以字节为单位读取文件内容。

FileInputStream有两个常用的构造方法:FileInputStream(File file)

创建用于读取给定的File对象所表示的文件FIS

另一个构造方法：FileInputStream(String name):

创建用于读取给定的文件系统中的路径名name所指定的文件的FIS

例如FileInputStream fis = new FileInputStream("demo");//创建一个用于读取demo.dat文件的输入流

**2.4. read()和write(int d)方法**

FileInputStream继承自InputStream,其提供了以字节为单位读取文件数据的方法read。

int read()

从此输入流中读取一个数据字节,若返回-1则表示EOF(End Of File)

FileOutputStream继承自OutputStream,其提供了以字节为单位向文件写数据的方法write。

void write(int d)

将指定字节写入此文件输出流。,这里只写给定的int值的”低八位”

例如

FileOutputStream fos = new FileOutputStream("demo.dat");

fos.write('A');//这里要注意，char占用2个字节，但这里只写入了1个字节。

**2.5. read(byte[] d)和write(byte[] d)方法**

FileInputStream也支持批量读取字节数据的方法:int read(byte[] b)

从此输入流中将最多 b.length 个字节的数据读入一个 byte 数组中 。

FileOutputStream也支持批量写出字节数据的方法:void write(byte[] d)

将 b.length 个字节从指定 byte 数组写入此文件输出流中。

FileOutputStream fos = new FileOutputStream("demo.txt");

byte[] data = "HelloWorld".getBytes();

fos.write(data);//会将HelloWorld的所有字节写入文件。

将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此文件输出流的方法：

void write(byte[] d,int offset,int len)

FileOutputStream fos = new FileOutputStream("demo.txt"); byte[] data = "HelloWorld".getBytes(); fos.write(data,5,5);//只会将world这5个字节写入文件。

* **使用文件输出节点流写文件案例：**

文件输出流节点流，是以文件为目标数据源的节点流，是基本的流，只提供了基本的输出方法write()

public class Demo{

public static void main(String[] args) throws Exception {

String file="abc/fos.dat";

//利用文件节点流打开一个文件，当文件不存在时候，会自动创建文件

//文件存在时候将文件替换为新文件，当文件不能写时候，出现异常

FileOutputStream out = new FileOutputStream(file);

//测试基本的 byte 数据写出方法，将byte写到文件中有效范围（0~255）

out.write(65);

out.write(66);

out.close();

}

}

* **文件输入节点流的读取文件案例：**

文件输入节点流，是文件作为数据来源的节点流，也是基础节点流，提供了基本的文件读取功能。

public class Demo{

public static void main(String[] args) throws Exception {

String file="abc/fos.dat";

//用文件节点输入流打开文件

//如果文件不能打开或者文件不存在就抛出异常！

FileInputStream in = new FileInputStream(file);

//测试基本的节点流读取方法

//每次从文件中读取一个byte(0~255)

int b1 = in.read();

int b2 = in.read();

System.out.println(b1);

System.out.println(b2);

in.close();

}

}

* **利用文件流实现文件的复制功能案例：**

\* 在不使用缓冲流的情况下，读写性能很差！

public class Demo{

public static void main(String[] args) throws Exception{

//打开原始输入文件

FileInputStream in =new FileInputStream("d:/TETRIS.zip");

//打开目标的输出文件

FileOutputStream out = new FileOutputStream("d:/TETRIS\_new.zip");

//从in里读取每个byte写到out流中

int b;

while((b=in.read())!=-1){

//b代表原始文件中的每个byte

out.write(b);

}

in.close();

out.close();

System.out.println("成功！");

}

}

这个程序性能有瑕疵。利用缓存读写方法可以提供文件复制性能：

* **自定义缓存读写案例 ：**

public class Demo{

public static void main(String[] args) throws Exception{

FileInputStream in =new FileInputStream("d:/TETRIS.zip");

FileOutputStream out = new FileOutputStream("d:/TETRIS\_new.zip");

byte[] buf=new byte[1024\*8];//1K byte

//从输入流in中读取尽可能多的byte填充到缓存 buf 中，返回读取个数 1024

//int n = in.read(buf);//1024

//int n = in.read(buf);//1024

//...

//n = in.read(buf); // 1 ~ 1024

//n = in.read(buf); // -1

int n;

while((n = in.read(buf))!=-1){

//将buf中从0开始的连续n个byte写到 文件流out中

out.write(buf, 0, n);

}

in.close();

out.close();

System.out.println("OK!");

}

}

为了避免最后一次复制多余的byte必须使用 out.write(buf, 0, n) 方法。

**3. 缓冲流**BufferedOutputStream

**3.1. BOS基本工作原理**

与缓冲输入流相似，在向硬件设备做写出操作时，增大写出次数无疑也会降低写出效率，为此我们可以使用缓冲输出流来一次性批量写出若干数据减少写出次数来提高写出效率。

BufferedOutputStream缓冲输出流内部也维护着一个缓冲区，每当我们向该流写数据时，都会先将数据存入缓冲区，当缓冲区已满时，缓冲流会将数据一次性全部写出。

**3.2. BOS实现写出缓冲**

**3.3. BOS的flush方法**

有时我们需要在执行完某些写出操作后，就希望将这些数据确实写出，而非在缓冲区中保存直到缓冲区满后才写出。这时我们可以使用缓冲流的一个方法flush。

void flush()

清空缓冲区，将缓冲区中的数据强制写出。

BufferedOutputStream bos= new BufferedOutputStream(new FileOutputStream("demo.dat"));

bos.write('a');//并没有向磁盘写出，而是写入到了BOS的缓存中

bos.flush();//强制将缓存中的数据一次性写出，这时‘a’才会被写入磁盘

bos.close();//实际上，close()方法在变比缓冲流前也调用了flush()

**3.4. BIS基本工作原理**

在读取数据时若以字节为单位读取数据，会导致读取次数过于频繁从而大大的降低读取效率。为此我们可以通过提高一次读取的字节数量减少读写次数来提高读取的效率。

BufferedInputStream是缓冲字节输入流。其内部维护着一个缓冲区(字节数组)，使用该流在读取一个字节时，该流会尽可能多的一次性读取若干字节并存入缓冲区，然后逐一的将字节返回，直到缓冲区中的数据被全部读取完毕，会再次读取若干字节从而反复。这样就减少了读取的次数，从而提高了读取效率。BIS是一个处理流，该流为我们提供了缓冲功能。

**3.5. BIS实现输入缓冲**

使用缓冲流来实现文件复制:

FileInputStream fis = new FileInputStream("java.zip");

BufferedInputStream bis = new BufferedInputStream(fis);

FileOutputStream fos = new FileOutputStream("copy\_java.zip");

BufferedOutputStream bos = new BufferedOutputStream(fos);

int d = -1; while((d = bis.read())!=-1){

bos.write(d); } bis.close();//读写完毕后要关闭流，只需要关闭最外层的流即可 bos.close();

**4. 对象流**

**4.1. 对象序列化概念**

对象是存在于内存中的。有时候我们需要将对象保存到硬盘上，又有时我们需要将对象传输到另一台计算机上等等这样的操作。这时我们需要将对象转换为一个字节序列，而这个过程就称为对象序列化。相反，我们有这样一个字节序列需要将其转换为对应的对象，这个过程就称为对象的反序列化。

**4.2. 使用OOS实现对象序列化**

ObjectOutputStream是用来对对象进行序列化的输出流。（将对象变成byte）

其实现对象序列化的方法为:void writeObject(Object o)

该方法可以将给定的对象转换为一个字节序列后写出。

**4.3. 使用OIS实现对象反序列化**

ObjectInputStream是用来对对象进行反序列化的输入流。（将byte拼接为对象）

其实现对象反序列化的方法为:Object readObject()

该方法可以从流中读取字节并转换为对应的对象。

**4.4. Serializable接口**

ObjectOutputStream在对对象进行序列化时有一个要求，就是需要序列化的对象所属的类必须实现Serializable接口。

实现该接口不需要重写任何方法。其只是作为可序列化的标志。

建议:(必须遵守) 实现序列化接口是添加序列化序列化 版本号，可以保证对象序列化、反序列化的稳定。减少更改类对序列化的影响。

* **对象序列化案例：**

public class Demo0 {

public static void main(String[] args) throws Exception{

Person p1 = new Person("范传奇", 10);

Person p2 = new Person("李洪鹤", 10);

p1.friend = p2;

String file="abc/obj.dat";

//将对象写到文件中

FileOutputStream fos=new FileOutputStream(file);

BufferedOutputStream bos=new BufferedOutputStream(fos);

//对象输出流

ObjectOutputStream oos=new ObjectOutputStream(bos);

//将对象写到文件中

oos.writeObject(p1);

oos.writeObject(p2);

oos.close();

}

}

* **对象反序列化案例：**

public class Demo {

public static void main(String[] args)throws Exception{

String file = "abc/obj.dat";

FileInputStream fis=new FileInputStream(file);

BufferedInputStream bis=new BufferedInputStream(fis);

ObjectInputStream ois=new ObjectInputStream(bis);

//读取对象从文件读取一系列byte数据拼接为对象

Person p1 = (Person)ois.readObject();

Person p2 = (Person)ois.readObject();

System.out.println(p1);

System.out.println(p2);

ois.close();

}

}

* 文件数据IO操作

**1. Reader和Writer**

**1.1. 字符流原理**

Reader是所有字符输入流的父类，而Writer是所有字符输出流的父类。字符流是以字符(char)为单位读写数据的。一次处理一个unicode字符流都是高级流，其底层都是依靠字节流进行读写数据的，所以底层仍然是基于字节读写数据的。字符流只能处理文本文件！

**1.2. 常用方法**

**Reader的常用方法:**

1)int read()

读取一个字符，返回的int”值低16”位有效。

2)int read(char[] chs)

从该流中读取一个字符数组length个字符并存入该数组，返回值为实际读取到的字符量。

**Writer的常用方法:**

1)void write(int c)

写出一个字符,写出给定int值”低16”位表示的字符。

2)void write(char[] chs)

将给定字符数组中所有字符写出。

3)void write(String str)

将给定的字符串写出

4)void write(char[] chs,int offset,int len):

将给定的字符数组中从offset处开始连续的len个字符写出

**2．转换流**

**2.1. 字符转换流原理**

InputStreamReader：字符输入流， 使用该流可以设置字符集，并按照指定的字符集从流中按照该编码将字节数据转换为字符并读取。

OutputStreamWriter：字符输出流，使用该流可以设置字符集，并按照指定的字符集将字符转换为对应字节后通过该流写出。

**2.2. 指定字符编码**

InputStreamReader的构造方法允许我们设置字符集:

InputStreamReader(InputStream in,String charsetName)

基于给定的字节输入流以及字符编码创建ISR

InputStreamReader(InputStream in)

该构造方法会根据系统默认字符集创建ISR

OutputStreamWriter:的构造方法:

OutputStreamWriter(OutputStream out,String charsetName)

基于给定的字节输出流以及字符编码创建OSW

OutputStreamWriter(OutputStream out)

该构造方法会根据系统默认字符集创建OSW

**2.3. 使用OutputStreamWriter**

public void testOutput() throws IOException{

FileOutputStream fos = new FileOutputStream("demo.txt");

OutputStreamWriter writer

//这里使用的字符编码为UTF-8

= new OutputStreamWriter(fos,"UTF-8");

String str = "大家好!";//UTF-8中文为3个字节，英文符号占1个字节

writer.write(str);//写出后该文件大小应该为10字节

writer.close();

}

**2.4. 使用InputStreamReader**

public void testInput() throws IOException{

FileInputStream fis = new FileInputStream("demo.txt");

/\* 这里设置了字符编码为GBK之后再通过ISR读取demo.txt文件时就使用GBK编码读取字符了

InputStreamReader reader = new InputStreamReader(fis,"GBK");

int c = -1;

while((c = reader.read()) != -1){

System.out.print((char)c);

}

reader.close();

}

**3. PrintWriter**

**3.1. 创建PrintWriter对象**

PrintWriter是具有自动行刷新的缓冲该字符输出流。其提供了比较丰富的构造方法:

* PrintWriter(File file)
* PrintWriter(String fileName)
* PrintWriter(OutputStream out)
* PrintWriter(OutputStream out,boolean autoFlush)
* PrintWriter(Writer writer)
* PrintWriter(Writer writer,boolean autoFlush)

其中参数为OutputStream与Writer的构造方法提供了一个可以传入boolean值参数，

该参数用于表示PrintWriter是否具有自动行刷新。

**3.2. PrintWriter的重载print和println方法**

使用PrintWriter写出字符串时我们通常不使用Writer提供的write()相关方法，而是使用print和println等方法，PrintWriter提供了若干重载的print与println方法，其中println方法是在写出数据后自动追加一个系统支持的换行符。

重载方法有:

* void print(int i)//打印整数
* void print(char c)//打印字符
* void print(boolean b)//打印boolean值
* void print(char[] c)//打印字符数组
* void print(double d)//打印double值
* void print(float f)//打印float值
* void print(long l)//打印long值
* void print(String str)//打印字符串

注:这些方法还有类似的println方法，参数与上面相同。

**3.3. 使用PW输出字符数据**

FileOutputStream fos = new FileOutputStream("demo.txt");

OutputStreamWriter osw = new OutputStreamWriter(fos,"UTF-8");

//创建带有自动行刷新的PW

PrintWriter pw = new PrintWriter(osw,true);

pw.println("大家好!");

pw.close();

**4. BufferedReader**

**4.1. 构建BufferedReader对象**

BufferedReader是缓冲字符输入流，其内部提供了缓冲区，可以提高读取效率。

BufferedReader的常用构造方法:BufferedReader(Reader reader)

例如:

FileInputStream fis = new FileInputStream("demo.txt");

InputStreamReader isr = new InputStreamReader(fis);

BufferedReader br = new BufferedReader(isr);

注:因为BufferedReader在构造实例时需要传入一个字符流，所以当我们想基于一个字节流进行读取时，要先将字节流转换为字符流后方可创建缓冲字符输入流BufferedReader。

**4.2. 使用BR读取字符串**

BufferedReader提供了一个可以便于读取一行字符串:

String readLine()

该方法连续读取一行字符串，直到读取到换行符为止，返回的字符串中不包含该换行符。若EOF则该方法返回null。

例如:

FileInputStream fis = new FileInputStream("demo.txt");

InputStreamReader isr = new InputStreamReader(fis);

BufferedReader br = new BufferedReader(isr);

String str = null;

while((str = br.readLine()) != null){ System.out.println(str); }

br.close();

* **字符输出流案例：**

public class Demo{

public static void main(String[] args) throws Exception {

String file = "abc/osw.txt";

FileOutputStream fos=new FileOutputStream(file);

BufferedOutputStream bos=new BufferedOutputStream(fos);

//OutputStreamWriter是高级流，必须依赖其他的字节流，扩展的字符编码功能。

OutputStreamWriter writer=new OutputStreamWriter(bos, "utf-8");

//将字符输出到文件

//将30000（田）编码为UTF-8的byte数据

//并且将byte写到文件中。

writer.write(30000);

writer.write("田地");

char[] chs = {'你','好','J','A','V','A'};

writer.write(chs);

writer.write(chs,2,4);

writer.close();

}

}

* **字符输入流案例：**

\* 读取文本信息

public class Demo{

public static void main(String[] args) throws Exception{

String file="abc/osw.txt";

FileInputStream fis=new FileInputStream(file);

BufferedInputStream bis=new BufferedInputStream(fis);

//InputStreamReader 是高级流，必须依赖基本的字节流，扩展了字符的解码功能

//将字节流中的byte数据读取后进行字符解码得到解码以后的字符数据

InputStreamReader reader=new InputStreamReader(bis, "utf-8");//GBK

//读取字符,返回0~65535之间的字符

int c = reader.read();

char ch = (char)c;

System.out.println(ch);

c = reader.read();

ch = (char)c;

System.out.println(ch);

int c;

while((c = reader.read())!=-1){

char ch = (char)c;

System.out.println(ch);

}

reader.close();

}

}

* **利用PrintWriter 写文本文件案例：**

public class Demo{

public static void main(String[] args) throws Exception{

String file="abc/pw.txt";

FileOutputStream fos=new FileOutputStream(file);

BufferedOutputStream bos=new BufferedOutputStream(fos);

OutputStreamWriter osw =new OutputStreamWriter(bos, "utf-8");

//PrintWriter：是高级流，扩展了println方法和print

// true 自动清理缓存功能，每个println方法之后会执行一个flush方法

PrintWriter out=new PrintWriter(osw, true);

out.println("Hello World!");

out.close();

}

}

建议：写文本文件时候要明确指定文本的编码。

* **利用BufferedReader读取文本文件案例：**

public class Demo{

public static void main(String[] args) throws Exception{

String file = "abc/day07.md";

FileInputStream fis = new FileInputStream(file);

BufferedInputStream bis=new BufferedInputStream(fis);

InputStreamReader isr = new InputStreamReader(bis,"UTF-8");

BufferedReader in=new BufferedReader(isr);

\* readLine 返回文本文件中的一行数据

\* 不包含回车字符，当读取到文件末尾时候返回null

String str;

while((str=in.readLine())!=null){

System.out.println(str);

}

in.close();

}

}

* **综合案例**

\* 将控制台输入的信息保存文本文件中

public class Demo{

public static void main(String[] args)throws Exception {

Scanner in = new Scanner(System.in);

String file="abc/mail.txt";

File mail = new File(file);

if(mail.exists()){

System.out.print("旧文件存在，还要吗(y/n)？");

String s = in.nextLine();

if(s.equals("n")){

if(mail.delete()){

System.out.println("已经删除");

}

}

}

//true: 表示追加方式写文件

FileOutputStream fos=new FileOutputStream(file, true);

BufferedOutputStream bos =new BufferedOutputStream(fos);

OutputStreamWriter osw = new OutputStreamWriter(bos, "UTF-8");

PrintWriter out=new PrintWriter(osw, true);

//从控制台读取一行，向文件中写一行

while(true){

String str = in.nextLine();

if(str.equals(":exit")){

break;

}

out.println(str);

}

out.close();

}

}

* **异常处理**

**1. 异常处理概述**

**1.1. 使用返回值状态标识异常**

在JAVA语言出现以前，传统的异常处理方式多采用返回值来标识程序出现的异常情况，这种方式虽然为程序员所熟悉，但却有多个坏处。

首先，一个API可以返回任意的返回值，而这些返回值本身并不能解释该返回值是否代表一个异常情况发生了和该异常的具体情况，需要调用API的程序自己判断并解释返回值的含义。

其次，并没有一种机制来保证异常情况一定会得到处理，调用程序可以简单的忽略该返回值，需要调用API的程序员记住去检测返回值并处理异常情况。这种方式还让程序代码变得冗长，尤其是当进行IO操作等容易出现异常情况的处理时，代码的很大部分用于处理异常情况的switch分支，程序代码的可读性变得很差。

**1.2. 异常处理机制**

当程序中抛出一个异常后，程序从程序中导致异常的代码处跳出，java虚拟机检测寻找和try关键字匹配的处理该异常的catch块，如果找到，将控制权交到catch块中的代码，然后继续往下执行程序，try块中发生异常的代码不会被重新执行。如果没有找到处理该异常的catch块，在所有的finally块代码被执行和当前线程的所属的ThreadGroup的uncaughtException方法被调用后，遇到异常的当前线程被中止。

**2. 异常的捕获和处理**

**2.1. Throwable,Error和Exception**

Java异常结构中定义有Throwable类，Exceotion和Error是其派生的两个子类。其中Exception表示由于网络故障、文件损坏、设备错误、用户输入非法等情况导致的异常，这类异常是可以通过Java异常捕获机制处理的。而Error表示Java运行时环境出现的错误，例如:JVM内存溢出等。

**2.2. try-catch**

try{...}语句指定了一段代码，该段代码就是一次捕获并处理例外的范围。在执行过程中，该段代码可能会产生并抛出一种或几种类型的异常对象，它后面的catch语句分别对这些异常做相应的处理。

如果没有列外产生，所有的catch代码段都被略过不执行

在catch语句块中是对异常进行处理的代码。catch中声明的异常对（ catch(SomeException e) ）封装了异常事件发生的信息，在catch语句块中可以使用这个对象的一些方法获取这些信息

常见格式:

try{

//可能出现异常的代码片段

}catch(Exception e){

//处理该异常的代码片段

}

**2.3. 多个catch**

每个try语句块可以伴随一个或多个catch语句，用于处理可能产生的不同类型的异常 。catch捕获的异常类型由上至下的捕获异常类型的顺序应是子类到父类的

例如

try{

…

}catch(NullPointerException e){ //子类异常应在上面捕获

…

}catch(RuntimeException e){ //父类异常在下面捕获

…

}catch(Exception e){ //应养成最终捕获Exception的习惯

…

}

通常在书写代码的时候，我们应当在最后一个catch中捕获Exception，这样可以保证代码不会因为出现一个未在catch中声明的异常而导致捕获失败使得程序终止。

**2.4. finally的作用**

finally语句为异常处理提供一个统一的出口，使得在控制流程转到程序其它部分以前，能够对程序的状态作统一管理。

无论try所指定的程序块中是否抛出例外,finally所指定的代码都要被执行，通常在finally语句中可以进行资源的消除工作，如关闭打开的文件、删除临时文件等。

finally语句块只能定义在try语句块之后，或者最后一个catch语句块之后，且只能定义一次。

**2.5. throw关键字**

当程序发生错误而无法处理的时候，会抛出对应的异常对象，除此之外，在某些时刻，您可能会想要自行抛出异常，例如在异常处理结束后，再将异常抛出，让下一层异常处理区块来捕捉，若想要自行抛出异常，您可以使用“throw”关键词，并生成指定的异常对象。

例如:throw new ArithmeticException();

**2.6. throws关键字**

程序中会声明许多方法（Method），这些方法中可能会因某些错误而引发异常，但您不希望直接在这个方法中处理这些异常，而希望调用这个它的方法来统一处理，这时候您可以使用“throws”关键词来声明这个方法将会抛出异常

例如:

public static void stringToDate(String str) throws ParseException{ ……}

**2.7. 重写方法时的throws**

当使用继承时，在父类的某个方法上声明了throws抛出某些异常，而在子类中重写该方法时，我们可以做以下的操作:

不处理异常（重写方法时不声明throws）

可仅在throws中声明父类中声明的部分异常

可在throws中声明父类方法中抛出的异常的子类异常

但是不能做以下操作：

重写方法时在throws中声明抛出额外的异常

重写方法时在throws中声明父类方法中声明的抛出异常的父类异常

**3.Java异常API**

**3.1. RuntimeException**

Java异常可以分为可检测异常，非检测异常

可检测异常：可检测异常经编译器验证，对于声明抛出异常的任何方法，编译器将强制执行处理或声明规则，不捕捉这个异常，编译器就通不过，不允许编译

非检测异常：非检测异常不遵循处理或者声明规则。在产生此类异常时，不一定非要采取任何适当操作，编译器不会检查是否已经解决了这样一个异常

RuntimeException 类属于非检测异常，因为普通JVM操作引起的运行时异常随时可能发生，此类异常一般是由特定操作引发。但这些操作在java应用程序中会频繁出现。因此它们不受编译器检查与处理或声明规则的限制 。

**3.2. 常见RuntimeException**

IllegalArgumentException

抛出的异常表明向方法传递了一个不合法或不正确的参数

NullPointerException

当应用程序试图在需要对象的地方使用 null 时，抛出该异常

ArrayIndexOutOfBoundsException

当使用的数组下标超出数组允许范围时，抛出该异常

ClassCastException

当试图将对象强制转换为不是实例的子类时，抛出该异常

NumberFormatException

当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。

**4. Exception常用API**

**4.1. printStackTrace**

Throwable中定义了一个方法可以输出错误信息，用来跟踪异常事件发生时执行堆栈的内容。该方法定义为:

void printStackTrace()

例如:

try{

…

}catch(Exception e){

e.printStackTrace();//输出执行堆栈信息

}

**4.2. getMessage**

Throwable中定义了一个方法可以得到有关异常事件的信息。该方法定义为:String getMessage()

例如:

try{

…

}catch(Exception e){

System.out.println(e.getMessage());

}

**4.3. getCause**

很多时候,当一个异常由另一个异常导致异常而被抛出的时候,Java库和开放源代码会将一种异常包装成另一种异常。这时,日志记录和打印根异常就变得非常重要。Java异常类提供了 getCause()方法来检索导致异常的原因,这些可以对异常根层次的原因提供更多的信息。该Java实践对代码的调试或故障排除有很大的帮助。另外,如果要把一个异常包装成另一种异常,构造一个新异常就要传递源异常。

Throwable getCause()

获取该异常出现的原因。

**5. 自定义Exception**

**5.1. 自定义异常的意义**

Java异常机制可以保证程序更安全和更健壮。虽然Java类库已经提供很多可以直接处理异常的类，但是有时候为了更加精准地捕获和处理异常以呈现更好的用户体验，需要开发者自定义异常。

**5.2. 继承Exception自定义异常**

创建自定义异常类，语法格式：

public class [自定义异常类名] extends Exception{

…

}

**5.3. 如何编写构造方法**

当定义好自定义异常后，我们可以通过Eclipse来自动生成相应的构造方法。

具体步骤如下:

1.声明一个类并继承自Exception

2.右键点击Source

3.选择Generate Constructors from Superclass

4.选中父类中所有构造方法后确认生成

* **异常处理案例:**

public class Demo{

public static void main(String[] args)throws Exception {

SimpleDateFormat fmt =new SimpleDateFormat("yyyy-MM-dd");

String str = "A2017-03-01";//"一九九七年";

String s = "a55";

Date d=null;

int age=0;

try{

d = fmt.parse(str);

age = Integer.parseInt(s);

}catch(ParseException e){

System.out.println("替换系统时间");

d = new Date();

}catch(NumberFormatException e){

System.out.println("处理数字解析异常");

age = 18;

}

System.out.println(d);

System.out.println(age);

//...

}

}

* **finally案例：**

try catch 中可以写一个fianlly代码块，finally代码块无论是否出现异常都会执行。

\* try catch finally 的规则

public class Demo{

public static void main(String[] args) {

// y = println(test(x))

System.out.println(test("5A"));

System.out.println(test(null));

System.out.println(test(""));

}

public static int test(String s){

try{

int n = s.charAt(0)-'0';

return n;

}catch(NullPointerException e){

return -1;

}catch (RuntimeException e) {

return -2;

}finally{//最终的，最终执行代码块

//无论是否出现异常永远执行

//用于执行回收资源：关闭文件等

System.out.println("finally");

}

}

}

* **一个完整的异常处理案例:**

public class Demo{

public static void main(String[] args) {

String from = "abc";

String to = "abc/ok.new";

try{

//Java编译器会检查：如果方法有异常声明就必须进行处理

copy(from, to);

System.out.println("完成");

}catch(RuntimeException e){

e.printStackTrace();

System.out.println(e.getMessage());

}catch (IOException e) {

e.printStackTrace();

System.out.println(e.getMessage());

}

}

//方法中如果抛出了异常，就需要使用throws声明方法有意外情况发生，否则会有编译错误

public static void copy(String from,String to) throws IOException,

RuntimeException {

File file = new File(from);

if(! file.exists()){

throw new RuntimeException(

"源文件不存在！"+from);

}

if(! file.isFile()){

throw new RuntimeException(

"源不是文件！"+from);

}

//...可以增加其他异常处理逻辑必须在try

FileInputStream in = null;

FileOutputStream out = null;

try{

in=new FileInputStream(from);

out=new FileOutputStream(to);

byte[] buf = new byte[1024\*8];

int n;

while((n=in.read(buf))!=-1){

out.write(buf, 0, n);

}

}catch(FileNotFoundException e){

e.printStackTrace();

}catch (IOException e) {

//复制期间的读写失败！

e.printStackTrace();

//复制期间出现的意外情况，必须

//再次抛给调用者，告诉调用者复制失败了

throw e;

}finally{

//回收系统资源close()方法本身也会抛出异常，必须进行捕获处理

try{

//由于in可能为null，为了避免空指针异常，必须检查是否为空

if(in!=null) in.close();

if(out!=null) out.close();

}catch(IOException e){

e.printStackTrace();

}

}

}

}

异常的继承关系：

所有异常都继承于： Throwable

Error 的子类表示系统不可恢复错误，一旦出现Java虚拟机就崩溃了，程序就无法恢复执行了。

Exception 的子类是系统可恢复异常，一旦出现异常只要使用代码进行适当的处理程序就可以恢复继续执行。

RuntimeException：非检查异常，是系统不检查的异常，Java编译器不检查RuntimeException和其子类的异常，无论是抛出和捕获都不检查。这类异常需要程序员编程需要避免的异常。

Error案例：

\* 系统不可恢复的错误：堆内存溢出（内存泄漏）

public class Demo10 {

public static void main(String[] args) {

List<Foo> list=

new ArrayList<Foo>();

for(int i=0; i<50000000; i++){

list.add(new Foo());

//new Foo();

}

}

}