

arena teaching system

Java 核心API（下）



Java企业应用及互联网
高级工程师培训课程

达内集团教学研发部 编著

目 录

Unit01	1
1. 文件操作—File	2
1.1. File 表示目录信息	2
1.1.1. listFiles()方法	2
1.1.2. FileFilter 接口	2
2. RandomAccessFile	3
2.1. 创建对象	3
2.1.1. 简介	3
2.1.2. 只读模式	3
2.1.3. 读写模式	3
2.2. 字节数据读写操作	4
2.2.1. write(int d)方法	4
2.2.2. read()方法	4
2.2.3. write(byte[] d)方法	4
2.2.4. read(byte[] b)方法	5
2.2.5. close()方法	5
2.3. 文件指针操作	5
2.3.1. getFilePointer 方法	5
2.3.2. seek 方法	6
2.3.3. skipBytes 方法	7
经典案例	8
1. 输出一个目录中的内容	8
2. 输出一个目录中所有扩展名为.txt 的文件	9
3. 测试 RandomAccessFile 的 read()和 write()方法	11
4. 测试 RandomAccessFile 的 read(byte[])和 write(byte[])方法	14
5. 测试文件指针的相关方法	17
课后作业	22
Unit02	23
1. 基本 IO 操作	25
1.1. IS 与 OS	25
1.1.1. 输入与输出	25
1.1.2. 节点流与处理流	25
1.1.3. IS 和 OS 常用方法	25
1.2. 文件流	26
1.2.1. 创建 FOS 对象(重写模式)	26
1.2.2. 创建 FOS 对象(追加模式)	26

1.2.3. 创建 FIS 对象	27
1.2.4. read()和 write(int d)方法	27
1.2.5. read(byte[] d)和 write(byte[] d)方法	27
1.3. 缓冲流	28
1.3.1. BOS 基本工作原理.....	28
1.3.2. BOS 实现写出缓冲.....	28
1.3.3. BOS 的 flush 方法	28
1.3.4. BIS 基本工作原理.....	29
1.3.5. BIS 实现输入缓冲.....	29
1.4. 对象流	29
1.4.1. 对象序列化概念	29
1.4.2. 使用 OOS 实现对象序列化.....	30
1.4.3. 使用 OIS 实现对象反序列化	30
1.4.4. Serializable 接口	30
1.4.5. transient 关键字	31
2. 文件数据 IO 操作	31
2.1. Reader 和 Writer	31
2.1.1. 字符流原理	31
2.1.2. 常用方法	32
2.2. 转换流	32
2.2.1. 字符转换流原理	32
2.2.2. 指定字符编码	33
2.2.3. 使用 OSW	33
2.2.4. 使用 ISR	34
经典案例	35
1. 测试构建 FOS 对象写文件（采用覆盖写和追加写两种模式）	35
2. 测试构建 FIS 对象并读取文件数据	38
3. 实现文件复制	40
4. 实现基于缓存区的文件复制	44
5. 实现存储 Emp 的序列化和反序列化	46
6. 按照指定编码将文本写入文件	50
7. 读出指定编码的文本文件	53
课后作业	56
Unit03	57
1. 文件数据 IO 操作.....	59
1.1. PrintWriter.....	59
1.1.1. 创建 PW 对象	59
1.1.2. print 与 println 方法	59
1.1.3. 使用 PW 输出字符数据.....	59

1.2. BufferedReader	60
1.2.1. 构建 BufferedReader	60
1.2.2. 使用 BR 读取字符串	60
2. 异常处理	60
2.1. 异常处理概述	60
2.1.1. 使用返回值状态标识异常	60
2.1.2. 异常处理机制	61
2.2. 异常的捕获和处理	61
2.2.1. Throwable, Error 和 Exception	61
2.2.2. try-catch	61
2.2.3. 多个 catch	62
2.2.4. finally 的作用	62
2.2.5. throw 关键字	62
2.2.6. throws 关键字	63
2.2.7. 重写方法时的 throws	63
2.3. Java 异常 API	63
2.3.1. RuntimeException	63
2.3.2. 常见 RuntimeException	64
2.4. Exception 常用 API	64
2.4.1. printStackTrace	64
2.4.2. getMessage	64
2.4.3. getCause	65
2.5. 自定义 Exception	65
2.5.1. 自定义异常的意义	65
2.5.2. 继承 Exception 自定义异常	65
2.5.3. 如何编写构造方法	66
经典案例	67
1. 将日志信息写入文本文件	67
2. 读取一个文本文件的所有行输出到控制台	69
3. FileUtils 实现文件复制功能（包含异常的处理）	71
4. 测试常见的 RuntimeException	75
5. FileUtils 实现文件复制功能（抛出自定义的文件复制异常）	81
课后作业	87
Unit04	89
1. 多线程基础	91
1.1. 进程与线程	91
1.1.1. 什么是进程	91
1.1.2. 什么是线程	91
1.1.3. 进程与线程的区别	91

1.1.4. 线程使用的场合	92
1.1.5. 并发原理	92
1.1.6. 线程状态	92
1.2. 创建线程	93
1.2.1. 使用 Thread 创建线并启动线程	93
1.2.2. 使用 Runnable 创建并启动线程	93
1.2.3. 使用内部类创建线程	93
1.3. 线程操作 API	94
1.3.1. Thread.currentThread 方法	94
1.3.2. 获取线程信息	94
1.3.3. 线程优先级	94
1.3.4. 守护线程	95
1.3.5. sleep 方法	95
1.3.6. yield 方法	95
1.3.7. join 方法	96
1.4. 线程同步	96
1.4.1. synchronized 关键字	96
1.4.2. 锁机制	96
1.4.3. 选择合适的锁对象	97
1.4.4. 选择合适的锁范围	97
1.4.5. 静态方法锁	97
1.4.6. wait 和 notify	98
经典案例	99
1. 创建两个线程，分别输出 1~100	99
2. 编写一个线程改变窗体的颜色 V1	101
3. 测试 currentThread 方法	104
4. 测试 getName 方法和 getId 方法	107
5. 测试守护线程	109
6. 编写一个线程改变窗体的颜色 V2	111
7. 测试 join 方法	113
课后作业	117
Unit05	120
1. 多线程基础	121
1.1. 线程同步	121
1.1.1. 线程安全 API 与非线程安全 API	121
1.1.2. 使用 ExecutorService 实现线程池	121
1.1.3. 使用 BlockingQueue	122
2. TCP 通信	123
2.1. Socket 原理	123

2.1.1. Socket 简介	123
2.1.2. 获取本地地址和端口号	123
2.1.3. 获取远端地址和端口号	124
2.1.4. 获取网络输入流和网络输出流	125
2.1.5. close 方法	125
2.2. Socket 通信模型	126
2.2.1. Server 端 ServerSocket 监听	126
2.2.2. Client 端 Socket 连接	126
2.2.3. C-S 端通信模型	126
经典案例	127
1. 测试使用 ExecutorService 实现线程池	127
2. 测试 BlockingQueue 的使用	131
3. 聊天室案例 V1	136
4. 聊天室案例 V2	144
课后作业	153
Unit06	154
1. TCP 通信	155
1.1. Socket 通信模型	155
1.1.1. Server 端多线程模型	155
2. UDP 通信	156
2.1. DatagramPacket	156
2.1.1. 构建接收包	156
2.1.2. 构建发送包	156
2.2. DatagramSocket	156
2.2.1. 服务端接收	156
2.2.2. 客户端发送	157
经典案例	158
1. 聊天室案例 V3	158
2. 聊天室案例 V4	167
3. 聊天室案例 V5	175
4. UDP 通信模型	184
课后作业	192
Unit07	195
1. XML 语法	197
1.1. XML 用途	197
1.1.1. XML 用途	197
1.2. 基本语法	197
1.2.1. XML 指令	197
1.2.2. 元素和属性	197

1.2.3. 大小写敏感	198
1.2.4. 元素必须有关闭标签	198
1.2.5. 必须有根元素	199
1.2.6. 元素必须正确嵌套	199
1.2.7. 实体引用	199
1.2.8. CDATA 段	200
2. XML 解析	200
2.1. XML 解析方式	200
2.1.1. SAX 解析方式	200
2.1.2. DOM 解析方式	200
2.2. 读取 XML	201
2.2.1. SAXReader 读取 XML 文档	201
2.2.2. Document 的 getRootElement 方法	201
2.3. Element	202
2.3.1. element 方法	202
2.3.2. elements 方法	202
2.3.3. getName 方法	203
2.3.4. getText 方法	203
2.3.5. attribute 方法	203
2.4. Attribute	204
2.4.1. getName,getValue	204
2.5. 写 XML	204
2.5.1. 构建 Document 对象	204
2.5.2. Document 的 addElement 方法	205
2.5.3. Element 的 addElement 方法	205
2.5.4. Element 的 addAttribute 方法	205
2.5.5. Element 的 addText 方法	206
2.5.6. XMLWriter 输出 XML 文档	206
2.6. XPath	206
2.6.1. 路径表达式	206
2.6.2. 谓词	208
2.6.3. 通配符	209
2.6.4. Dom4J 对 XPath 的支持	210
经典案例	211
1. 完成多条 Emp 信息的 XML 描述	211
2. 读取 XML 文档解析 Emp 信息	214
3. 将 Emp (存放在 List 中) 对象转换为 XML 文档	220
4. 在 XML 文档中查找指定特征的 Emp 信息	225
课后作业	230

Java 核心 API(下)

Unit01

知识体系.....Page 2

文件操作—File	File 表示目录信息	listFiles()方法
		FileFilter 接口
RandomAccessFile	创建对象	简介
		只读模式
		读写模式
	字节数据读写操作	write(int d)方法
		read()方法
		write(byte[] d)方法
		read(byte[] b)方法
		close()方法
	文件指针操作	getFilePointer 方法
		seek 方法
		skipBytes 方法

经典案例.....Page 8

输出一个目录中的内容	listFiles()方法
输出一个目录中所有扩展名为.txt 的文件	FileFilter 接口
测试 RandomAccessFile 的 read()和 write()方法	write(int b) 方法
	read() 方法
测试 RandomAccessFile 的 read(byte[])和 write(byte[])方法	write(byte[] b)方法
	read(byte[] b) 方法
测试文件指针的相关方法	getFilePointer 方法
	seek 方法
	skipBytes 方法

课后作业.....Page 22

1. 文件操作—File

1.1. File 表示目录信息

1.1.1. 【File 表示目录信息】listFiles()方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Tarena 达内科技</div> <h4>listFiles()方法</h4> <ul style="list-style-type: none"> File的listFiles方法用于返回一个抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的子项(文件或目录)。 File[] listFiles() 返回值：抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的文件和目录。如果目录为空，那么数组也将为空。如果抽象路径名不表示一个目录，或者发生 I/O 错误，则返回 null。 <div style="text-align: right;">+</div>
---	--

1.1.2. 【File 表示目录信息】FileFilter 接口


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Tarena 达内科技</div> <h4>FileFilter接口</h4> <ul style="list-style-type: none"> FileFilter用于抽象路径名的过滤器 此接口的实例可传递给 File 类的 listFiles(FileFilter) 方法。用于返回满足该过滤器要求的子项。 File[] listFiles(FileFilter filter) <pre>File[] list = dir.listFiles(new FileFilter() { @Override public boolean accept(File pathname) { return pathname.getName().startsWith("."); } });</pre> <div style="text-align: right;">+</div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Tarena 达内科技</div> <h4>FileFilter接口 (续1)</h4> <pre>public void testFileFilter(){ File dir = new File("."); File[] subs = dir.listFiles(new FileFilter(){ public boolean accept(File file) { return file.getName().endsWith(".txt"); } }); for(File sub : subs){ System.out.println(sub); } }</pre> <div style="text-align: right;">+</div>
---	--

2. RandomAccessFile

2.1. 创建对象


2.1.1. 【创建对象】简介

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>简介</h4> <ul style="list-style-type: none"> Java提供了一个可以对文件随机访问的操作，访问包括读和写操作。该类名为RandomAccessFile。该类的读写是基于指针的操作。 <div style="position: absolute; left: 455px; top: 250px; writing-mode: vertical-rl;">知识讲解</div> <div style="text-align: right;">+</div>
---	--

2.1.2. 【创建对象】只读模式



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>只读模式</h4> <ul style="list-style-type: none"> RandomAccessFile在对文件进行随机访问操作时有两个模式，分别为只读模式(只读取文件数据)，和读写模式(对文件数据进行读写)。 只读模式: 在创建RandomAccessFile时，其提供的构造方法要求我们传入访问模式: <ul style="list-style-type: none"> RandomAccessFile(File file,String mode) RandomAccessFile(String filename,String mode) 其中构造方法的第一个参数是需要访问的文件，而第二个参数则是访问模式: "r" :字符串" r" 表示对该文件的访问是只读的。 <div style="position: absolute; left: 455px; top: 490px; writing-mode: vertical-rl;">知识讲解</div> <div style="text-align: right;">+</div>
---	---

2.1.3. 【创建对象】读写模式



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>读写模式</h4> <ul style="list-style-type: none"> 创建一个基于文件访问的读写模式的RandomAccessFile我们只需要在第二个参数中传入" rw" 即可 <div style="position: absolute; left: 455px; top: 725px; writing-mode: vertical-rl;">知识讲解</div> <ul style="list-style-type: none"> RandomAccessFile raf = new RandomAccessFile(file," rw"); 那么这时在使用RandomAccessFile对该文件的访问就是又可读又可写的。 <div style="text-align: right;">+</div>
---	--

2.2. 字节数据读写操作



2.2.1. 【字节数据读写操作】write(int d)方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>write(int d)方法</h4> <ul style="list-style-type: none"> • RandomAccessFile提供了一个可以向文件中写出字节的方法: <ul style="list-style-type: none"> – void write(int d) • 该方法会根据当前指针所在位置处写入一个字节，是将参数int的“低8位”写出。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	--



2.2.2. 【字节数据读写操作】read()方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>read()方法</h4> <ul style="list-style-type: none"> • RandomAccessFile提供了一个可以从文件中读取字节的方法: <ul style="list-style-type: none"> – int read(); • 该方法会从文件中读取一个byte(8位) 填充到int的低8位, 高24位为0, 返回值范围正数: 0~255, 如果返回-1表示读取到了文件末尾! 每次读取后自动移动文件指针, 准备下次读取。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	---


2.2.3. 【字节数据读写操作】write(byte[] d)方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>write(byte[] d)方法</h4> <ul style="list-style-type: none"> • RandomAccessFile提供了一个可以向文件中写出一组字节的方法: <ul style="list-style-type: none"> – void write(byte[] d) • 该方法会根据当前指针所在位置处连续写出给定数组中的所有 字节。 • 与该方法相似的还有一个常用方法: <ul style="list-style-type: none"> – void write(byte[] d,int offset,int len) • 该方法会根据当前指针所在位置处连续写出给定数组中的部分字节，这个部分是从数组的offset处开始，连续len个字节。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	--

2.2.4. 【字节数据读写操作】read(byte[] b)方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>read(byte[] b)方法</h3> <ul style="list-style-type: none"> RandomAccessFile提供了一个可以从文件中批量读取字节的方法: <ul style="list-style-type: none"> int read(byte[] b) 该方法会从指针位置处尝试最多读取给定数组的总长度的字节量，并从给定的字节数组第一个位置开始，将读取到的字节顺序存放至数组中，返回值为实际读取到的字节量 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	--


2.2.5. 【字节数据读写操作】close()方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>close()方法</h3> <ul style="list-style-type: none"> RandomAccessFile在对文件访问的操作全部结束后，要调用close()方法来释放与其关联的所有系统资源。 <ul style="list-style-type: none"> void close() <p>RandomAccessFile raf = new RandomAccessFile(file, "rw");//读写操作 raf.close();//访问完后要关闭以释放系统资源。</p> <div style="text-align: right;">+</div>
---	--

2.3. 文件指针操作


2.3.1. 【文件指针操作】getFilePointer 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>getFilePointer方法</h3> <ul style="list-style-type: none"> RandomAccessFile的读写操作都是基于指针的，也就是说总是在指针当前所指向的位置进行读写操作。 方法: <ul style="list-style-type: none"> long getFilePointer() 该方法用于获取当前RandomAccessFile的指针位置。 <div style="text-align: right;">+</div>
---	---



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>getFilePointer方法(续1)</h3> <pre> RandomAccessFile raf = new RandomAccessFile(file," rw"); System.out.println(raf.getFilePointer());//0 raf.write('A'); System.out.println(raf.getFilePointer());//1 raf.writeInt(3); System.out.println(raf.getFilePointer());//5 raf.close(); </pre> <div style="display: flex; align-items: center;"> <div style="background-color: black; color: white; padding: 2px 5px; writing-mode: vertical-rl; margin-right: 5px;">代码清单</div> <div style="border: 1px solid black; padding: 5px; width: 100%;"> <div style="display: flex; justify-content: space-between;"> ++ </div> </div> </div>
---	--

2.3.2. 【文件指针操作】seek 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>seek方法</h3> <ul style="list-style-type: none"> RandomAccessFile提供了一个方法用于移动指针位置。 方法: <ul style="list-style-type: none"> – void seek(long pos) <p>该方法用于移动当前RandomAccessFile的指针位置。</p> <div style="display: flex; align-items: center;"> <div style="background-color: black; color: white; padding: 2px 5px; writing-mode: vertical-rl; margin-right: 5px;">代码清单</div> <div style="border: 1px solid black; padding: 5px; width: 100%;"> <div style="display: flex; justify-content: space-between;"> ++ </div> </div> </div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>seek方法(续1)</h3> <pre> RandomAccessFile raf = new RandomAccessFile(file," rw"); System.out.println(raf.getFilePointer());//0 raf.write('A');//指针位置1 raf.writeInt(3);//指针位置5 //将指针移动到文件开始处(第一个字节的位置) raf.seek(0); System.out.println(raf.getFilePointer());//0 raf.close(); </pre> <div style="display: flex; align-items: center;"> <div style="background-color: black; color: white; padding: 2px 5px; writing-mode: vertical-rl; margin-right: 5px;">代码清单</div> <div style="border: 1px solid black; padding: 5px; width: 100%;"> <div style="display: flex; justify-content: space-between;"> ++ </div> </div> </div>
---	---

2.3.3. 【文件指针操作】skipBytes 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>skipBytes方法</h4> <ul style="list-style-type: none"> • RandomAccessFile提供了一个方法可以尝试跳过输入的 n 个字节以丢弃跳过的字节。 • 方法: <ul style="list-style-type: none"> – int skipBytes(int n) • 该方法可能跳过一些较少数量的字节（可能包括零）。这可能由任意数量的条件引起；在跳过 n 个字节之前已到达文件的末尾只是其中的一种可能。 • 此方法不抛出 EOFException。返回跳过的实际字节数。如果 n 为负数，则不跳过任何字节。 <div style="text-align: right;">  </div>
---	---

经典案例

1. 输出一个目录中的内容

- 问题

输出当前目录下所有的内容。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：构建测试方法

首先，新建 `TestFile` 类，并在类中新建测试方法 `testListFiles`，代码如下所示：

```
import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {

    @Test
    public void testListFiles() {

    }

}
```

步骤二：获取当前目录下的内容

首先，使用 `File` 类构建表示当前目录的 `file` 对象；然后，使用 `File` 类的 `listFiles` 方法获取到当前目录下所有的内容，包括文件和目录；最后，遍历 `listFiles` 方法返回的数组查看当前目录下的内容，代码如下所示：

```
import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    @Test
    public void testListFiles() {

        File dir = new File(".");
        File[] subs = dir.listFiles();
        for (File sub : subs) {
            System.out.println(sub);
        }

    }
}
```

上述代码中，“.”表示的当前目录，本题案例的当前目录，就是当前工程下。

步骤三：运行

运行 testListFiles 方法，控制台输出结果如下所示：

```
.\classpath
.\project
.\a
.\bin
.\demo
.\doc
.\lib
.\myDir
.\src
```

- **完整代码**

本案例的完整代码如下所示：

```
import java.io.File;
import java.io.IOException;
import org.junit.Test;

public class TestFile {
    @Test
    public void testListFiles() {
        File dir = new File(".");
        File[] subs = dir.listFiles();
        for (File sub : subs) {
            System.out.println(sub);
        }
    }
}
```

2. 输出一个目录中所有扩展名为.txt 的文件

- **问题**

输出当前目录下，所有扩展名为.txt 的文件。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：构建测试方法

首先，在 TestFile 类中新建测试方法 testFileFilter，代码如下所示：

```
import java.io.File;
import java.io.FileFilter;
import java.io.IOException;
import org.junit.Test;

public class TestFile {

    @Test
```



```
public void testFileFilter() {  
  
}  
  
}
```

步骤二：获取当前目录下的扩展名为.txt 的文件

首先,使用 File 类构建表示当前目录的 file 对象;然后,使用 File 类的 listFiles 方法获取当前目录下的扩展名为.txt 的文件;最后,遍历 listFiles 方法返回的数组查看当前目录下的内容,代码如下所示:

```
import java.io.File;  
import java.io.FileFilter;  
import java.io.IOException;  
import org.junit.Test;  
  
public class TestFile {  
    @Test  
    public void testFileFilter() {  
  
        File dir = new File(".");  
        File[] subs = dir.listFiles(new FileFilter() {  
            public boolean accept(File file) {  
                return file.getName().endsWith(".txt");  
            }  
        });  
        for (File sub : subs) {  
            System.out.println(sub);  
        }  
  
    }  
}
```

此处使用了 listFiles 方法,该方法的参数为 FileFilter 类型,FileFilter 是抽象路径名的过滤器。此接口的实例可传递给 File 类的 listFiles(FileFilter) 方法,实现过滤功能。

• 完整代码

本案例的完整代码如下所示:

```
import java.io.File;  
import java.io.FileFilter;  
import java.io.IOException;  
import org.junit.Test;  
  
public class TestFile {  
    //... (之前案例的代码,略)  
  
    @Test  
    public void testFileFilter() {  
        File dir = new File(".");  
        File[] subs = dir.listFiles(new FileFilter() {  
            public boolean accept(File file) {  
                return file.getName().endsWith(".txt");  
            }  
        });  
    }  
}
```

```

    }
    });
    for (File sub : subs) {
        System.out.println(sub);
    }
}
}

```

3. 测试 RandomAccessFile 的 read()和 write()方法

- 问题

使用 RandomAccessFile 类进行文件的读写，详细要求如下：

- 1) 使用 RandomAccessFile 的 write 方法向文件 raf.dat 写入数字 “1”。
- 2) 使用 RandomAccessFile 的 read 方法从文件 raf.dat 中将数字 “1” 读取出来并打印到控制台。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 TestRandomAccessFile 类，添加测试方法 testWrite 方法

首先，新建类 TestRandomAccessFile；然后，在该类中添加测试方法 testWrite，代码如下所示：

```

import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试写出方法
     * @throws Exception
     */
    @Test
    public void testWrite()throws Exception{
    }
}

```

步骤二：使用 RandomAccessFile 类的 write 方法向文件写数据

首先，使用读写模式创建 RandomAccessFile 类的对象，然后，使用 RandomAccessFile 类的 write 方法向文件 raf.dat 写数字 “1”。该方法写出一个字节，写出的是 int 值的低 8 位。代码如下所示：

```

import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */

```

```
public class TestRandomAccessFile {  
    /**  
     * 测试写出方法  
     * @throws Exception  
     */  
    @Test  
    public void testWrite()throws Exception{  
  
        RandomAccessFile raf = new RandomAccessFile("raf.dat","rw");  
        //写出一个字节,写的是 int 值的低 8 位  
        raf.write(1);  
        raf.close();  
  
    }  
}
```

运行 testWrite 方法，在当前工程下生成文件 raf.dat，该文件的大小为 1 个字节。

步骤三：添加测试方法 testRead

在 TestRandomAccessFile 类中添加测试方法 testRead，代码如下所示：

```
import java.io.RandomAccessFile;  
import org.junit.Test;  
/**  
 * 测试 RandomAccessFile  
 */  
public class TestRandomAccessFile {  
    /**  
     * 测试写出方法  
     * @throws Exception  
     */  
    @Test  
    public void testWrite()throws Exception{  
        //... (代码略)  
    }  
    /**  
     * 测试读取方法  
     */  
  
    @Test  
    public void testRead()throws Exception{  
    }  
  
}
```

步骤四：使用 TestRandomAccessFile 类的 read 方法从文件读取数据

首先，使用只读模式创建 RandomAccessFile 类的对象；使用 RandomAccessFile 类的 read 方法从文件 raf.dat 读取数字“1”。该方法读取一个字节，读取的是 int 值的低 8 位。代码如下所示：

```
import java.io.RandomAccessFile;  
import org.junit.Test;  
  
/**
```

```

    * 测试 RandomAccessFile
    */
    public class TestRandomAccessFile {
        /**
         * 测试写出方法
         * @throws Exception
         */
        @Test
        public void testWrite()throws Exception{
            //... (代码略)
        }
        /**
         * 测试读取方法
         */
        @Test
        public void testRead()throws Exception{

            RandomAccessFile raf = new RandomAccessFile("raf.dat","r");
            //读取一个字节
            int d = raf.read();
            System.out.println(d);
            raf.close();

        }
    }
}

```

运行 testRead 方法，控制台输出结果如下：

```
1
```

从输出结果可以看出，已经将数字 “1” 读取出来。

• 完整代码

本案例的完整代码如下所示：

```

import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试写出方法
     * @throws Exception
     */
    @Test
    public void testWrite()throws Exception{
        RandomAccessFile raf = new RandomAccessFile("raf.dat","rw");
        //写出一个字节,写的是 int 值的低 8 位
        raf.write(1);
        raf.close();
    }
    /**
     * 测试读取方法
     */
}

```

```
@Test
public void testRead()throws Exception{
    RandomAccessFile raf = new RandomAccessFile("raf.dat","r");
    //读取一个字节
    int d = raf.read();
    System.out.println(d);
    raf.close();
}
}
```

4. 测试 RandomAccessFile 的 read(byte[])和 write(byte[])方法

- 问题

使用 RandomAccessFile 类进行文件的读写，详细要求如下：

- 1) 使用 RandomAccessFile 的 write(byte[])方法向文件 raf.dat 写入字符串 "HelloWorld"。
- 2) 使用 RandomAccessFile 的 read(byte[])方法从文件 raf.dat 中将字符串 "HelloWorld" 读取出来并打印到控制台。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：在 TestRandomAccessFile 类中，添加测试方法 testWriteByteArray 方法

在 TestRandomAccessFile 类中添加测试方法 testWriteByteArray 代码如下所示：

```
import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试批量写出
     */

    @Test
    public void testWriteByteArray() throws Exception {
    }

}
```

步骤二：使用 RandomAccessFile 类的 write(byte[])方法向文件写数据

首先，使用读写模式创建 RandomAccessFile 类的对象，然后，使用 RandomAccessFile 类的 write(byte[])方法向文件 raf.dat 中写字符串 "HelloWorld"。该方法将字节数组中所有字节一次性写出。代码如下所示：

```
import java.io.RandomAccessFile;
```

```
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试批量写出
     */
    @Test
    public void testWriteByteArray() throws Exception {

        RandomAccessFile raf = new RandomAccessFile("raf.dat", "rw");
        // 将字符串按照默认编码转换为字节数组
        byte[] buf = "HelloWorld".getBytes();
        // 将字节数组中所有字节一次性写出
        raf.write(buf);
        raf.close();

    }
}
```

运行 testWriteByteArray 方法，将当前工程下的 raf.dat 文件的大小变为 10 个字节。

步骤三：添加测试方法 testReadByteArray

在 TestRandomAccessFile 类中添加测试方法 testReadByteArray ,代码如下所示：

```
import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试批量写出
     */
    @Test
    public void testWriteByteArray() throws Exception {
        //... (代码略)
    }
    /**
     * 测试批量读取
     */

    @Test
    public void testReadByteArray() throws Exception {
    }

}
```

步骤四：使用 TestRandomAccessFile 类的 read (byte[]) 方法从文件读取数据

首先，使用只读模式创建 RandomAccessFile 类的对象；使用 RandomAccessFile 类

的 read(byte[])方法从文件 raf.dat 读取字符串 “HelloWorld”。代码如下所示：

```
import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试批量写出
     */
    @Test
    public void testWriteByteArray() throws Exception {
        //... (代码略)
    }
    /**
     * 测试批量读取
     */
    @Test
    public void testReadByteArray() throws Exception {

        RandomAccessFile raf = new RandomAccessFile("raf.dat", "r");
        // 创建长度为 10 的字节数组
        byte[] buf = new byte[10];
        // 尝试读取 10 个字节存入数组，返回值为读取的字节量
        int len = raf.read(buf);
        System.out.println("读取到了:" + len + "个字节");
        System.out.println(new String(buf));
        raf.close();

    }
}
```

运行 testReadByteArray 方法，控制台输出结果如下：

```
读取到了:10 个字节
HelloWorld
```

从输出结果可以看出，已经将字符串 “HelloWorld” 读取出来了。

• 完整代码

本案例中，类 TestRandomAccessFile 的完整代码如下所示：

```
import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    //... (之前案例的代码，略)

    /**
```

```

    * 测试批量写出
    *
    * @throws Exception
    */
    @Test
    public void testWriteByteArray() throws Exception {
        RandomAccessFile raf = new RandomAccessFile("raf.dat", "rw");
        // 将字符串按照默认编码转换为字节
        byte[] buf = "HelloWorld".getBytes();
        // 将字节数组中所有字节一次性写出
        raf.write(buf);
        raf.close();
    }

    /**
    * 测试批量读取
    *
    * @throws Exception
    */
    @Test
    public void testReadByteArray() throws Exception {
        RandomAccessFile raf = new RandomAccessFile("raf.dat", "r");
        // 创建长度为10的字节数组
        byte[] buf = new byte[10];
        // 尝试读取10个字节存入数组，返回值为读取的字节量
        int len = raf.read(buf);
        System.out.println("读取到了:" + len + "个字节");
        System.out.println(new String(buf));
        raf.close();
    }
}

```

5. 测试文件指针的相关方法

• 问题

使用 `RandomAccessFile` 类的对象操作文件指针，详细要求如下：

- 1) 获取文件 `raf.dat` 的指针。
- 2) 设置指针读取 `raf.dat` 中的字符串 "World"。
- 3) 将指针移动到 `raf.dat` 文件的开始。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：添加测试方法 `testPointer` 方法

在 `TestRandomAccessFile` 类中添加测试方法 `testPointer`，代码如下所示：

```

import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**

```



```
* 测试指针相关方法  
*/
```

```
@Test  
public void testPointer() throws Exception {  
  
}  
  
}
```

步骤二：使用 RandomAccessFile 类的 getFilePointer 方法获取文件指针

使用 RandomAccessFile 类的 getFilePointer 方法获取 raf.dat 文件的指针，代码如下所示：

```
import java.io.RandomAccessFile;  
import org.junit.Test;  
  
/**  
 * 测试 RandomAccessFile  
 */  
public class TestRandomAccessFile {  
    /**  
     * 测试指针相关方法  
     */  
    @Test  
    public void testPointer() throws Exception {  
  
        RandomAccessFile raf = new RandomAccessFile("raf.dat", "r");  
        // 输出指针位置,默认从 0 开始(文件的第一个字节位置)  
        System.out.println("指针位置:" + raf.getFilePointer());  
  
        raf.close();  
  
    }  
}
```

文件指针位置默认从 0 开始，即文件的第一个字节所在的位置。

运行 testPointer 方法，控制台输出结果如下：

```
指针位置:0
```

从输出结果可以看出，文件指针位置为 0。

步骤三：读取字符串 "World"

首先，使用 RandomAccessFile 类的 skipBytes 方法跳过 "Hello" 这 5 个字节并查看此时的文件指针位置；然后，使用 RandomAccessFile 类的 read(byte[])方法读取 "World" 并输出；最后，再次获取此时指针的位置，代码如下所示：

```
import java.io.RandomAccessFile;  
import org.junit.Test;
```

```
/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试指针相关方法
     */
    @Test
    public void testPointer() throws Exception {
        RandomAccessFile raf = new RandomAccessFile("raf.dat", "r");
        // 输出指针位置,默认从 0 开始 (文件的第一个字节位置)
        System.out.println("指针位置:" + raf.getFilePointer());

        // 读取 world,需要将 hello 这 5 个字节跳过
        raf.skipBytes(5);
        System.out.println("指针位置:" + raf.getFilePointer());
        // 读取 world 这 5 个字节
        byte[] buf = new byte[5];
        raf.read(buf);
        System.out.println(new String(buf));
        System.out.println("指针位置:" + raf.getFilePointer());

        raf.close();
    }
}
```

运行 testPointer 方法，控制台的输出结果如下：

指针位置:0

指针位置:5

World

指针位置:10

从输出结果可以看出，当使用 skipBytes 方法跳过 5 个字节时，文件指针位置为 5；从指针位置 5 开始读取，读取到了字符串“World”；读取完字符串“World”后，文件的指针位置变为 10。

步骤四：将指针移到文件开始

使用 RandomAccessFile 类的 seek 方法将文件指针移到 raf.dat 文件的开始，代码如下所示：

```
import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    /**
     * 测试指针相关方法
     */
}
```

```
@Test
public void testPointer() throws Exception {
    RandomAccessFile raf = new RandomAccessFile("raf.dat", "r");
    // 输出指针位置,默认从 0 开始(文件的第一个字节位置)
    System.out.println("指针位置:" + raf.getFilePointer());

    // 读取 World,需要将 Hello 这 5 个字节跳过
    raf.skipBytes(5);
    System.out.println("指针位置:" + raf.getFilePointer());
    // 读取 World 这 5 个字节
    byte[] buf = new byte[5];
    raf.read(buf);
    System.out.println(new String(buf));
    System.out.println("指针位置:" + raf.getFilePointer());

    // 将游标移动到文件开始

    raf.seek(0);
    System.out.println("指针位置:" + raf.getFilePointer());

    raf.close();
}
}
```

运行 testPointer 方法,控制台输出结果如下:

```
指针位置:0
指针位置:5
world
指针位置:10
```

指针位置:0

从输出结果可以看出,已经将指针移到了文件的开始。

- **完整代码**

本案例的完整代码如下所示:

```
import java.io.RandomAccessFile;
import org.junit.Test;

/**
 * 测试 RandomAccessFile
 */
public class TestRandomAccessFile {
    //... (之前案例的代码,略)

    /**
     * 测试指针相关方法
     */
    @Test
    public void testPointer() throws Exception {
```

```
RandomAccessFile raf = new RandomAccessFile("raf.dat", "r");
// 输出指针位置,默认从0开始(文件的第一个字节位置)
System.out.println("指针位置:" + raf.getFilePointer());

// 读取 world,需要将hello这5个字节跳过
raf.skipBytes(5);
System.out.println("指针位置:" + raf.getFilePointer());
// 读取 world这5个字节
byte[] buf = new byte[5];
raf.read(buf);
System.out.println(new String(buf));
System.out.println("指针位置:" + raf.getFilePointer());

// 将游标移动到文件开始
raf.seek(0);
System.out.println("指针位置:" + raf.getFilePointer());

raf.close();
}
}
```

课后作业

1. 使用 commons-io API 输出一个目录中所有扩展名为.jar 的文件

使用 commons-io API 输出当前目录及其子目录下，所有扩展名为.jar 的文件。

2. 简述 RAF 的两种创建模式

3. 简述 RandomAccessFile 类的 read 方法和 write 方法使用 int 类型存储 byte 数据的方式和原因

4. 实现文件的复制功能

使用 RandomAccessFile 类的 read(byte[])方法和 write (byte[]) 方法实现文件复制。

5. 文件信息如下所示，下面代码的输出结果是？

请看 raf.dat 文件的内容如下：

Welcome,NEWYORK

以上字符都为英文字符，请看下列代码的输出结果是：()。

```
RandomAccessFile raf = new RandomAccessFile("raf.dat", "r");
raf.skipBytes(8);
byte[] buf = new byte[7];
raf.read(buf);
System.out.println(new String(buf));
raf.close();
```

Java 核心 API(下)

Unit02

知识体系.....Page 25

基本 IO 操作	IS 与 OS	输入与输出
		节点流与处理流
		IS 和 OS 常用方法
	文件流	创建 FOS 对象(重写模式)
		创建 FOS 对象(追加模式)
		创建 FIS 对象
		read()和 write(int d)方法
		read(byte[] d)和 write(byte[] d)方法
	缓冲流	BOS 基本工作原理
		BOS 实现写出缓冲
		BOS 的 flush 方法
		BIS 基本工作原理
		BIS 实现输入缓冲
	对象流	对象序列化概念
		使用 OOS 实现对象序列化
		使用 OIS 实现对象反序列化
		Serializable 接口
		transient 关键字
文件数据 IO 操作	Reader 和 Writer	字符流原理
		常用方法
	转换流	字符转换流原理
		指定字符编码
		使用 OSW
		使用 ISR

经典案例.....Page 35

测试构建 FOS 对象写文件(采用覆盖写和追加写两种模式)	构建 FOS 对象 (重写模式)
	构建 FOS 对象 (追加模式)
测试构建 FIS 对象并读取文件数据	构建 FIS 对象



实现文件复制	read()和 write(int b)方法
	read(byte[] b)和 write(byte[] b)
实现基于缓存区的文件复制	BOS 基本工作原理
	BOS 实现写出缓冲
	BOS 的 flush 方法
	BIS 基本工作原理
	BIS 实现输入缓冲
实现存储 Emp 的序列化和反序列化	使用 OOS 实现对象序列化
	使用 OIS 实现对象反序列化
按照指定编码将文本写入文件	使用 OSW
读出指定编码的文本文件	使用 ISR

课后作业.....Page 56



1. 基本 IO 操作

1.1. IS 与 OS



1.1.1. 【IS 与 OS】输入与输出



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">输入与输出</h4> <ul style="list-style-type: none"> 我们编写的程序除了自身会定义一些数据信息外，经常还会引用外界的数据，或是将自身的数据发送到外界。比如，我们编写的程序想读取一个文本文件，又或者我们想将程序中的某些数据写入到一个文件中。这时我们就要使用输入与输出。 什么是输入： <ul style="list-style-type: none"> 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。 什么是输出： <ul style="list-style-type: none"> 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。 <div style="text-align: right;">  </div>
---	---

1.1.2. 【IS 与 OS】节点流与处理流

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">节点流与处理流</h4> <ul style="list-style-type: none"> 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。 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。 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。 通常节点流也称为低级流。 通常处理流也称为高级流或过滤流。 <div style="text-align: right;">  </div>
---	--


1.1.3. 【IS 与 OS】IS 和 OS 常用方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">IS和OS常用方法</h4> <ul style="list-style-type: none"> InputStream是所有字节输入流的父类，其定义了基础的读取方法，常用的方法如下： <ul style="list-style-type: none"> int read() <p>读取一个字节，以int形式返回，该int值的“低八位”有效，若返回值为-1则表示EOF</p> int read(byte[] d) <p>尝试最多读取给定数组的length个字节并存入该数组，返回值为实际读取到的字节量。</p> <div style="text-align: right;">  </div>
---	--


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>IS和OS常用方法(续1)</h3> <ul style="list-style-type: none"> • OutputStream是所有字节输出流的父类，其定义了基础的写出方法，常用的方法如下： <ul style="list-style-type: none"> – void write(int d) 写出一个字节,写的是给定的int的“低八位” – void write(byte[] d) 将给定的字节数组中的所有字节全部写出 <div style="text-align: right;">  </div>
---	--

1.2. 文件流

1.2.1. 【文件流】创建 FOS 对象(重写模式)

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>创建FOS对象(重写模式)</h3> <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> – FileOutputStream(File file): 创建一个向指定 File 对象表示的文件中写出数据的文件输出流。 – FileOutputStream(String filename): 创建一个向具有指定名称的文件中写出数据的文件输出流。 <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>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。</p> </div>
---	---



1.2.2. 【文件流】创建 FOS 对象(追加模式)

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>创建FOS对象(追加模式)</h3> <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> – FileOutputStream(File file,boolean append): 创建一个向指定 File 对象表示的文件中写出数据的文件输出流。 – FileOutputStream(String filename,boolean append): 创建一个向具有指定名称的文件中写出数据的文件输出流。 <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <p>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。</p> </div>
---	---


1.2.3. 【文件流】创建 FIS 对象

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>创建FIS对象</h4> <ul style="list-style-type: none"> • FileInputStream是文件的字节输入流，我们使用该流可以以字节为单位从文件中读取数据。 • FileInputStream有两个常用的构造方法： <ul style="list-style-type: none"> – FileInputStream(File file) 创建一个从指定 File 对象表示的文件中读取数据的文件输入流。 – FileInputStream(String name) 创建用于读取给定的文件系统的路径名name所指定的文件的文件输入流 <div style="text-align: right;">  </div>
---	--

1.2.4. 【文件流】read()和 write(int d)方法



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>read()和write(int d)方法</h4> <ul style="list-style-type: none"> • 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。 <ul style="list-style-type: none"> – 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。 <ul style="list-style-type: none"> – 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” <div style="text-align: right;">  </div>
---	---

1.2.5. 【文件流】read(byte[] d)和 write(byte[] d)方法



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>read(byte[] d)和write(byte[] d)方法</h4> <ul style="list-style-type: none"> • FileInputStream也支持批量读取字节数据的方法： <ul style="list-style-type: none"> – int read(byte[] b) 从此输入流中将最多 b.length 个字节的数据读入到字节数组b中 • FileOutputStream也支持批量写出字节数据的方法： <ul style="list-style-type: none"> – 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。 – void write(byte[] d,int offset,int len) 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。 <div style="text-align: right;">  </div>
---	---

1.3. 缓冲流



1.3.1. 【缓冲流】BOS 基本工作原理

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">BOS基本工作原理</h4> <ul style="list-style-type: none"> 在向硬件设备做写出操作时，增大写出次数无疑会降低写出效率，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。 <div style="text-align: right;">  </div>
---	--


1.3.2. 【缓冲流】BOS 实现写出缓冲

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">BOS实现写出缓冲</h4> <pre> public void testBos()throws Exception { FileOutputStream fos = new FileOutputStream("demo.dat"); //创建缓冲字节输出流 BufferedOutputStream bos = new BufferedOutputStream(fos); //所有字节被存入缓冲区，等待一次性写出 bos.write("helloworld".getBytes()); //关闭流之前，缓冲输出流会将缓冲区内容一次性写出 bos.close(); } </pre> <div style="text-align: right;">  </div>
---	--


1.3.3. 【缓冲流】BOS 的 flush 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">BOS的flush方法</h4> <ul style="list-style-type: none"> 使用缓冲输出流可以提高写出效率，但是这也存在一个问题，就是写出数据缺乏即时性。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。 <div style="text-align: right;">  </div>
---	---

1.3.4. 【缓冲流】BIS 基本工作原理


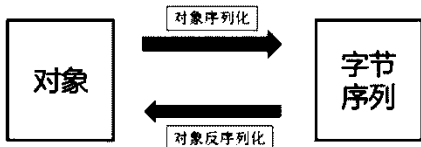
<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">BIS基本工作原理</h3> <ul style="list-style-type: none"> 在读取数据时若以字节为单位读取数据，会导致读取次数过于频繁从而大大的降低读取效率。为此我们可以通过提高一次读取的字节数量减少读写次数来提高读取的效率。 BufferedInputStream是缓冲字节输入流。其内部维护着一个缓冲区(字节数组)，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。 <div style="text-align: right;">++</div>
---	---

1.3.5. 【缓冲流】BIS 实现输入缓冲

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">BIS实现输入缓冲</h3> <pre> public void testBis()throws Exception { //创建缓冲字节输入流 FileInputStream fis = new FileInputStream("demo.dat"); BufferedInputStream bis = new BufferedInputStream(fis); int d = -1; //缓冲读入，实际上并非是一个字节一个字节从文件读取的。 while((d = bis.read())!=-1){ System.out.print(d + " "); } bis.close(); } </pre> <div style="text-align: right;">++</div>
---	---

1.4. 对象流

1.4.1. 【对象流】对象序列化概念

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">对象序列化概念</h3> <ul style="list-style-type: none"> 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。 <div style="text-align: center;">  </div> <div style="text-align: right;">++</div>
---	---

1.4.2. 【对象流】使用 OOS 实现对象序列化

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div><div>知识讲解</div><div><h3>使用OOS实现对象序列化</h3><ul style="list-style-type: none">ObjectOutputStream是用来对对象进行序列化的输出流。其实现对象序列化的方法为:<ul style="list-style-type: none">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。</div><div>++</div></div>
---	---

1.4.3. 【对象流】使用 OIS 实现对象反序列化




<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div><div>知识讲解</div><div><h3>使用OIS实现对象反序列化</h3><ul style="list-style-type: none">ObjectInputStream是用来对对象进行反序列化的输入流。其实现对象反序列化的方法为:<ul style="list-style-type: none">Object readObject()该方法可以从流中读取字节并转换为对应的对象。</div><div>++</div></div>
---	--

1.4.4. 【对象流】Serializable 接口

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div><div>知识讲解</div><div><h3>Serializable接口</h3><ul style="list-style-type: none">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。通常实现该接口的类需要提供一个常量serialVersionUID，表明该类的版本。若不显示的声明，在对象序列化时也会根据当前类的各个方面计算该类的默认serialVersionUID，但不同平台编译器实现有所不同，所以若想跨平台，都应显示的声明版本号。</div><div>++</div></div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>Serializable接口(续1)</h3> <ul style="list-style-type: none"> 如果声明的类的对象序列化存到硬盘上面，之后随着需求的变化更改了类的属性(增加或减少或改名)，那么当反序列化时，就会出现InvalidClassException，这样就会造成不兼容性的问题。 但当serialVersionUID相同时，它就会将不一样的field以type的预设值反序列化，可避开不兼容性问题。 <div style="text-align: right;">  </div> <div style="position: absolute; left: 455px; top: 175px; background-color: black; color: white; padding: 2px;">知识讲解</div> <div style="position: absolute; bottom: 10px; left: 455px;">  </div>
---	--




1.4.5. 【对象流】transient 关键字

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>transient关键字</h3> <ul style="list-style-type: none"> 对象在序列化后得到的字节序列往往比较大，有时我们在对一个对象进行序列化时可以忽略某些不必要的属性，从而对序列化后得到的字节序列“瘦身”。 关键字 transient 被该关键字修饰的属性在序列化时其值将被忽略。 <div style="text-align: right;">  </div> <div style="position: absolute; left: 455px; top: 410px; background-color: black; color: white; padding: 2px;">知识讲解</div> <div style="position: absolute; bottom: 10px; left: 455px;">  </div>
---	---

2. 文件数据 IO 操作

2.1. Reader 和 Writer

2.1.1. 【Reader 和 Writer】字符流原理

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>字符流原理</h3> <ul style="list-style-type: none"> Reader是字符输入流的父类 Writer是字符输出流的父类。 字符流是以字符(char)为单位读写数据的。一次处理一个unicode。 字符流的底层仍然是基本的字节流。 <div style="text-align: right;">  </div> <div style="position: absolute; left: 455px; top: 720px; background-color: black; color: white; padding: 2px;">知识讲解</div> <div style="position: absolute; bottom: 10px; left: 455px;">  </div>
---	--

2.1.2. 【Reader 和 Writer】常用方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h4>常用方法</h4> <ul style="list-style-type: none"> • Reader的常用方法: <ul style="list-style-type: none"> – int read():读取一个字符, 返回的int值“低16”位有效。 – int read(char[] chs):从该流中读取一个字符数组的length个字符并存入该数组, 返回值为实际读取到的字符数。 <div style="text-align: right;">+</div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h4>常用方法(续1)</h4> <ul style="list-style-type: none"> • Writer的常用方法: <ul style="list-style-type: none"> – void write(int c):写出一个字符,写出给定int值“低16”位表示的字符。 – void write(char[] chs):将给定字符数组中所有字符写出。 – void write(String str):将给定的字符串写出 – 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 <div style="text-align: right;">+</div>
---	---

2.2. 转换流

2.2.1. 【转换流】字符转换流原理

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h4>字符转换流原理</h4> <ul style="list-style-type: none"> • InputStreamReader : 字符输入流。 <ul style="list-style-type: none"> – 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 <ul style="list-style-type: none"> – 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。 <div style="text-align: right;">+</div>
---	---

2.2.2. 【转换流】指定字符编码

Technology
Tarena
达内科技

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

+

Technology
Tarena
达内科技

指定字符编码(续1)

- 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

+

2.2.3. 【转换流】使用 OSW

Technology
Tarena
达内科技

使用OSW

```

public void testOutput() throws IOException{
    FileOutputStream fos
        = new FileOutputStream("demo.txt");
    OutputStreamWriter writer
        = new OutputStreamWriter(fos,"UTF-8");
    String str = "大家好!";
    writer.write(str);
    writer.close();
}
            
```

在构造方法中设置字符集，这里我们使用UTF-8编码将字符串写入到demo.txt文件中

+

2.2.4. 【转换流】使用 ISR

使用ISR

public void testInput() throws IOException{
 FileInputStream fis
 = new FileInputStream("demo.txt");
 InputStreamReader reader
 = new InputStreamReader(fis, "UTF-8");
 int c = -1;
 while((c = reader.read()) != -1){
 System.out.print((char)c);
 }
 reader.close();
}

在构造方法中设置字符集，这里我们读取demo.txt时使用了UTF-8编码

经典案例

1. 测试构建 FOS 对象写文件（采用覆盖写和追加写两种模式）

• 问题

使用 `FileOutputStream` 类的对象向文件写入数据，详细要求如下：

- 1) 使用覆盖写的方式，向文件 `fos.dat` 写入字符串 "helloworld"。
- 2) 在步骤一的基础上，使用追加写的方式，向文件 `fos.dat` 写入字符串 "helloworld"。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 `TestFileOutputStream` 类，添加测试方法 `testFos` 方法

首先新建类 `TestFileOutputStream`，然后在该类中添加测试方法 `testFos`，代码如下所示：

```
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件字节输出流
 */
public class TestFileOutputStream {
    /**
     * 测试覆盖写方式
     */
    @Test
    public void testFos() throws Exception{
    }
}
```

步骤二：实现覆盖写

首先，创建文件字节输出流 `FileOutputStream` 的对象；然后，使用该类的 `write(byte[])` 方法将字符串 "helloworld" 写出；最后，将 `fos` 流关闭，以释放资源，代码如下所示：

```
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件字节输出流
 */
public class TestFileOutputStream {
    /**
     * 测试覆盖写方式
     */
    @Test
    public void testFos() throws Exception{
```

```
//创建文件字节输出流
//FileOutputStream fos = new FileOutputStream(new File("fos.dat"));
FileOutputStream fos = new FileOutputStream("fos.dat");

//写出一组字节
fos.write("helloworld".getBytes());
fos.close();

}
}
```

上述代码中，使用下列两种方式来构造实现覆盖写的 `FileOutputStream` 对象，其中一种方式是以 `File` 对象的方式构造，代码如下：

```
FileOutputStream fos= new FileOutputStream(new File("fos.dat"));
```

另一种方式是使用字符串的方式构造，代码如下：

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

运行 `testFos` 方法，在当前工程下生成了文件 `fos.dat`，该文件的内容为“helloworld”；再次运行 `testFos` 方法，并查看文件内容，该文件的内容仍然为“helloworld”。

步骤三：添加测试方法 `testFosByAppend`

在 `TestFileOutputStream` 类中添加测试方法 `testFosByAppend`，代码如下所示：

```
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件字节输出流
 */
public class TestFileOutputStream {
    /**
     * 测试覆盖写方式
     */
    @Test
    public void testFos()throws Exception{
        //...（代码略）
    }

    /**
     * 测试追加写的方式
     */

    @Test
    public void testFosByAppend()throws Exception{
    }

}
```

步骤四：实现追加写

追加写和覆盖写类似，唯一的区别是在构造 `FileOutputStream` 类的对象时，构造参数发生变化，使用如下方式构造：

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

可以看出在构造具备追加写的 `FileOutputStream` 对象时，多了一个参数，该参数值为 `true`，表示以追加的方式向文件 `fos.dat` 写入数据。

代码如下所示：

```
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件字节输出流
 */
public class TestFileOutputStream {
    /**
     * 测试覆盖写方式
     */
    @Test
    public void testFos()throws Exception{
        //... (代码略)
    }

    /**
     * 测试追加写的方式
     */
    @Test
    public void testFosByAppend()throws Exception{

        //创建文件字节输出流
        //FileOutputStream fos = new FileOutputStream(new File("fos.dat"),true );
        FileOutputStream fos = new FileOutputStream("fos.dat",true);

        //写出一组字节
        fos.write("helloworld".getBytes());
        fos.close();

    }
}
```

运行 `testFosByAppend` 方法 在当前工程下 `fos.dat` 文件的内容更新为“helloworld helloworld ”，说明实现了向文件 `fos.dat` 追加写入数据。

• 完整代码

本案例的完整代码如下所示：

```
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件字节输出流
```

```
*/
public class TestFileOutputStream {
    /**
     * 测试覆盖写方式
     */
    @Test
    public void testFos() throws Exception{
        //创建文件字节输出流
        //FileOutputStream fos = new FileOutputStream(new File("fos.dat"));
        FileOutputStream fos = new FileOutputStream("fos.dat");

        //写出一组字节
        fos.write("helloworld".getBytes());
        fos.close();
    }

    /**
     * 测试追加写的方式
     */
    @Test
    public void testFosByAppend() throws Exception{
        //创建文件字节输出流
        //FileOutputStream fos =
        //        new FileOutputStream(new File("fos.dat"),true );
        FileOutputStream fos = new FileOutputStream("fos.dat",true);

        //写出一组字节
        fos.write("helloworld".getBytes());
        fos.close();
    }
}
```

2. 测试构建 FIS 对象并读取文件数据

- 问题

使用 `FileInputStream` 类的对象从文件 `fos.dat` 中读取字符串 “helloworld”。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 `TestFileInputStream` 类，添加测试方法 `testFis` 方法

首先新建类 `TestFileInputStream`，然后在该类中添加测试方法 `testFis`，代码如下所示：

```
import java.io.FileInputStream;
import org.junit.Test;

/**
 * 测试文件输入流
 */
public class TestFileInputStream {
    /**
     * 测试文件输入流的创建以及读取数据
     */
    @Test
    public void testFis() throws Exception{
```

```
}  
}
```

步骤二：使用 TestFileInputStream 实现取出字符串 “helloworld”

首先，创建文件字节输出流 FileInputStream 的对象；然后，使用该类的 read 方法将字符串 “helloworld” 读取出来，该方法返回 -1 时表示读取到了文件末尾；最后，将 fis 流关闭，以释放资源，代码如下所示：

```
import java.io.FileInputStream;
import org.junit.Test;

/**
 * 测试文件输入流
 */
public class TestFileInputStream {
    /**
     * 测试文件输入流的创建以及读取数据
     */
    @Test
    public void testFis() throws Exception{

        //根据给定的 File 对象创建文件输入流
        //FileInputStream fis = new FileInputStream(new File("fos.dat"));
        //根据给定的文件路径创建文件输入流
        FileInputStream fis = new FileInputStream("fos.dat");

        int d = -1;
        while( (d = fis.read()) != -1){
            System.out.print((char)d + " ");
        }
        fis.close();

    }
}
```

上述代码中，使用下列两种方式来构造 FileInputStream 对象，其中一种方式是以 File 对象的方式构造，代码如下：

```
FileInputStream fos= new FileInputStream (new File("fos.dat"));
```

另一种方式是使用字符串的方式构造，代码如下：

```
FileInputStream fos= new FileInputStream ("fos.dat");
```

运行 testFis 方法，控制台输出结果如下所示：

```
h e l l o w o r l d
```

从输出结果可以看出，已经将字符串 “helloworld” 读取出来。

- **完整代码**

本案例的完整代码如下所示：

```
import java.io.FileInputStream;
import org.junit.Test;

/**
 * 测试文件输入流
 */
public class TestFileInputStream {
    /**
     * 测试文件输入流的创建以及读取数据
     */
    @Test
    public void testFis() throws Exception {
        //根据给定的 File 对象创建文件输入流
        //FileInputStream fis = new FileInputStream(new File("fos.dat"));
        //根据给定的文件路径创建文件输入流
        FileInputStream fis = new FileInputStream("fos.dat");

        int d = -1;
        while( (d = fis.read()) != -1){
            System.out.print((char)d + " ");
        }
        fis.close();
    }
}
```

3. 实现文件复制

- **问题**

使用 `FileOutputStream` 类和 `FileInputStream` 类实现文件复制，详细要求如下：

- 1) 使用 `FileOutputStream` 类的 `write(int)` 方法和 `FileInputStream` 类的 `read` 方法复制 `fos.dat` 文件为 `fos_copy1.dat` 文件。
- 2) 使用 `FileOutputStream` 类的 `write(byte[])` 和 `FileInputStream` 类的 `read (byte[])` 方法复制 `fos.dat` 文件为 `fos_copy2.dat` 文件。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：新建 `TestCopy` 类，添加测试方法 `testCopyFile1` 方法

首先新建类 `TestCopy`，然后在该类中添加测试方法 `testCopyFile1`，代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件复制
 */
public class TestCopy {
```

```
/**
 * 测试使用字节形式复制文件
 */
@Test
public void testCopyFile1() throws Exception{
}
}
```

步骤二：实现文件复制

首先，创建文件字节输入流 `FileInputStream` 类的对象和文件字节输出流 `FileOutputStream` 类的对象；然后，使用循环。在循环中，使用 `FileInputStream` 类的 `read` 方法从文件 `fos.dat` 中读取数据；使用 `FileOutputStream` 类的 `write` 方法将读取到的数据写入 `fos_copy1.dat` 文件中，直到 `read` 方法返回 -1，则退出循环；最后，将流关闭，以释放资源，代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件复制
 */
public class TestCopy {
    /**
     * 测试使用字节形式复制文件
     */
    @Test
    public void testCopyFile1() throws Exception{

        FileInputStream fis = new FileInputStream("fos.dat");
        FileOutputStream fos = new FileOutputStream("fos_copy1.dat");

        int d = -1;
        while((d = fis.read()) != -1){
            fos.write(d);
        }
        System.out.println("复制完毕");
        fis.close();
        fos.close();

    }
}
```

运行 `testCopyFile1` 方法，在当前工程下生成了文件 `fos_copy1.dat`，该文件的内容与文件 `fos.dat` 的内容相同。

步骤三：添加测试方法 `testCopyFile2`

在 `TestCopy` 类中添加测试方法 `testCopyFile2`，代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件复制
```



```
*/
public class TestCopy {
/**
 * 测试使用字节形式复制文件
 */
@Test
public void testCopyFile1()throws Exception{
    //... (代码略)
}

/**
 * 测试使用字节数组形式复制文件
 */

@Test
public void testCopyFile2()throws Exception{
}

}
```

步骤四：再次实现文件复制

首先，创建文件字节输入流 `FileInputStream` 类的对象和文件字节输出流 `FileOutputStream` 类的对象；然后，使用循环。在循环中，使用 `FileInputStream` 类的 `read (byte[])` 方法从文件 `fos.dat` 中读取数据；使用 `FileOutputStream` 类的 `write(byte[])` 方法将读取到的数据写入 `fos_copy2.dat` 文件中，直到 `read(byte[])` 方法返回-1，则退出循环；最后，将流关闭，以释放资源，代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件复制
 */
public class TestCopy {
/**
 * 测试使用字节形式复制文件
 */
@Test
public void testCopyFile1()throws Exception{
    //... (代码略)
}

/**
 * 测试使用字节数组形式复制文件
 */
@Test
public void testCopyFile2()throws Exception{

    FileInputStream fis = new FileInputStream("fos.dat");
    FileOutputStream fos = new FileOutputStream("fos_copy2.dat");

    int len = -1;
    byte[] buf = new byte[32];
    while((len = fis.read(buf)) != -1){
        fos.write(buf,0,len);
    }
}
```

```

    }
    System.out.println("复制完毕");
    fis.close();
    fos.close();

}
}

```

运行 testCopyFile2 方法,在当前工程下生成了文件 fos_copy2.dat,该文件的内容与文件 fos.dat 的内容相同。

• 完整代码

本案例中,类 TestCopy 的完整代码如下所示:

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试文件复制
 */
public class TestCopy {
    /**
     * 测试使用字节形式复制文件
     */
    @Test
    public void testCopyFile1() throws Exception{
        FileInputStream fis = new FileInputStream("fos.dat");
        FileOutputStream fos = new FileOutputStream("fos_copy1.dat");

        int d = -1;
        while((d = fis.read()) != -1){
            fos.write(d);
        }
        System.out.println("复制完毕");
        fis.close();
        fos.close();
    }

    /**
     * 测试使用字节数组形式复制文件
     */
    @Test
    public void testCopyFile2() throws Exception{
        FileInputStream fis = new FileInputStream("fos.dat");
        FileOutputStream fos = new FileOutputStream("fos_copy2.dat");

        int len = -1;
        byte[] buf = new byte[32];
        while((len = fis.read(buf)) != -1){
            fos.write(buf,0,len);
        }
        System.out.println("复制完毕");
        fis.close();
        fos.close();
    }
}

```

4. 实现基于缓存区的文件复制

- 问题

使用 `BufferedOutputStream` 类和 `BufferedInputStream` 类实现文件复制，即复制 `fos.dat` 文件为 `fos_copy3.dat` 文件。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 `TestBisAndBos` 类，添加测试方法 `testCopy` 方法

首先新建类 `TestBisAndBos`，然后在该类中添加测试方法 `testCopy`，代码如下所示：

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试基于缓冲流的复制文件
 */
public class TestBisAndBos {
    /**
     * 测试基于缓冲流的复制文件
     */
    @Test
    public void testCopy() throws Exception{
    }
}
```

步骤二：实现文件复制

使用 `BufferedInputStream` 和 `BufferedOutputStream` 实现文件复制的详细过程如下：

1) 首先，创建文件字节输入流 `FileInputStream` 类的对象，接着使用该文件输入流对象作为参数构造缓冲字节输入流 `BufferedInputStream` 类的对象；

2) 然后，创建文件字节输出流 `FileOutputStream` 类的对象，接着使用该文件输出流对象作为参数构造缓冲字节输出流 `BufferedOutputStream` 类的对象；

3) 构建循环。在循环中，使用 `BufferedInputStream` 类的 `read` 方法从文件 `fos.dat` 中读取数据；使用 `BufferedOutputStream` 类的 `write` 方法将读取到的数据写入 `fos_copy3.dat` 文件中，直到 `read` 方法返回-1，则退出循环。

4) 将流关闭，以释放资源。

代码如下所示：

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import org.junit.Test;

/**
 * 测试基于缓冲流的复制文件
 */
public class TestBisAndBos {
    /**
     * 测试基于缓冲流的复制文件
     */
    @Test
    public void testCopy()throws Exception{

        //创建缓冲字节输入流
        FileInputStream fis = new FileInputStream("fos.dat");
        BufferedInputStream bis = new BufferedInputStream(fis);

        //创建缓冲字节输出流
        FileOutputStream fos = new FileOutputStream("fos_copy3.dat");
        BufferedOutputStream bos = new BufferedOutputStream(fos);

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

        System.out.println("复制完毕");
        bis.close();
        bos.close();

    }
}
```

运行 testCopy 方法，在当前工程下生成了文件 fos_copy3.dat，该文件的内容与文件 fos.dat 的内容相同。

• 完整代码

本案例中，类 TestBisAndBos 的完整代码如下所示：

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import org.junit.Test;

/**
 * 测试基于缓冲流的复制文件
 */
public class TestBisAndBos {
    /**
     * 测试基于缓冲流的复制文件
     */
    @Test
    public void testCopy()throws Exception{
        //创建缓冲字节输入流
        FileInputStream fis = new FileInputStream("fos.dat");
        BufferedInputStream bis = new BufferedInputStream(fis);
```

```
//创建缓冲字节输出流
FileOutputStream fos = new FileOutputStream("fos copy3.dat");
BufferedOutputStream bos = new BufferedOutputStream(fos);

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

System.out.println("复制完毕");
bis.close();
bos.close();
}
}
```

5. 实现存储 Emp 的序列化和反序列化

- 问题

实现存储 Emp 的序列化和反序列化，详细要求如下：

- 1) 使用属性 name、age、gender、salary 为["张三",15,"男",4000]构造 Emp 类的对象。
- 2) 将第一步创建的 Emp 对象，序列化到文件 emp.obj 中。
- 3) 将第二步序列化到 emp.obj 中的 Emp 对象，反序列化出来并输出到控制台。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建 Emp 类

创建 Emp 类，代码如下所示：

```
import java.io.Serializable;
/**
 * 实现序列化接口后该类可以被序列化
 */
public class Emp implements Serializable{
    /**
     * 版本号
     */
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    private String gender;
    private double salary;

    public Emp(String name, int age, String gender, double salary) {
        this.name = name;
        this.age = age;
        this.gender = gender;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
```

```

        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    @Override
    public String toString() {
        return "Emp [name=" + name + ", age=" + age + ", gender=" + gender
            + ", salary=" + salary + "]\n";
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
}

```

特别需要注意的是，该类实现了 `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。

步骤二：新建测试方法

创建 `TestOisAndOos` 类，并在类中新建测试方法 `testOOS`，代码如下所示：

```

import org.junit.Test;

/**
 * 实现对象的序列化与反序列化
 */
public class TestOisAndOos {
    /**
     * 使用 oos 实现对象的序列化
     */
    @Test
    public void testOOS() throws Exception{
    }
}

```

步骤三：序列化 `Emp` 对象到文件中

将 `Emp` 对象序列化到文件 `emp.obj` 中的详细过程如下：

1) 使用属性 `name`、`age`、`gender`、`salary` 为["张三",15,"男",4000]构造 `Emp` 类的对象；

2) 创建文件字节输出流 `FileOutputStream` 类的对象,接着使用该文件字节输出流对象作为参数构造对象字节输出流 `ObjectOutputStream` 类的对象;

3) 使用 `ObjectOutputStream` 类的 `writeObject` 方法将 `Emp` 对象写入到文件 `emp.obj` 中;

4) 关闭 `oos` 对象流,以释放资源。

代码如下所示:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import org.junit.Test;

/**
 * 实现对象的序列化与反序列化
 */
public class TestOisAndOos {
    /**
     * 使用 oos 实现对象的序列化
     */
    @Test
    public void testOOS() throws Exception{

        FileOutputStream fos = new FileOutputStream("emp.obj");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        Emp emp = new Emp("张三",15,"男",4000);
        oos.writeObject(emp);
        System.out.println("序列化完毕");
        oos.close();

    }
}
```

运行 `testOOS` 方法,在当前工程下生成了文件 `emp.obj`,该文件的内容是二进制输入,无法读懂其中的内容。

步骤四：新建测试方法 `testOIS`

在 `TestOisAndOos` 类中,新建测试方法 `TestOIS`,代码如下所示:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import org.junit.Test;

/**
 * 实现对象的序列化与反序列化
 */
public class TestOisAndOos {
    /**
     * 使用 oos 实现对象的序列化
     */
    @Test
```

```
public void testOOS()throws Exception{
    //... (代码略)
}
/**
 * 使用 OIS 实现对象的反序列化
 */
@Test
public void testOIS()throws Exception{

}
}
```

步骤五：实现对 Emp 对象的反序列化

将 Emp 对象从文件 emp.obj 反序列化读取出来的详细过程如下：

1) 创建文件字节输入流 FileInputStream 类的对象，接着使用该文件字节输入流对象作为参数构造对象字节输入流 ObjectInputStream 类的对象；

2) 使用 ObjectInputStream 类的 readObject 方法将 Emp 对象从 emp.obj 文件中读取出来；

3) 关闭 ois 对象流，以释放资源。

代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import org.junit.Test;

/**
 * 实现对象的序列化与反序列化
 */
public class TestOisAndOos {
    /**
     * 使用 OOS 实现对象的序列化
     */
    @Test
    public void testOOS()throws Exception{
        //... (代码略)
    }
    /**
     * 使用 OIS 实现对象的反序列化
     */
    @Test
    public void testOIS()throws Exception{

        FileInputStream fis = new FileInputStream("emp.obj");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Emp emp = (Emp)ois.readObject();
        System.out.println("反序列化完毕");
        System.out.println(emp);
        ois.close();

    }
}
```


运行 testOIS 方法，控制台输出结果如下所示：

反序列化完毕

Emp [name=张三, age=15, gender=男, salary=4000.0]

从输出结果可以看出，已经将 Emp 对象反序列化出来了。

- **完整代码**

本案例中，类 TestOisAndOos 的完整代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import org.junit.Test;

/**
 * 实现对象的序列化与反序列化
 */
public class TestOisAndOos {
    /**
     * 使用 oos 实现对象的序列化
     */
    @Test
    public void testOOS() throws Exception {
        FileOutputStream fos = new FileOutputStream("emp.obj");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        Emp emp = new Emp("张三", 15, "男", 4000);
        oos.writeObject(emp);
        System.out.println("序列化完毕");
        oos.close();
    }
    /**
     * 使用 ois 实现对象的反序列化
     */
    @Test
    public void testOIS() throws Exception {
        FileInputStream fis = new FileInputStream("emp.obj");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Emp emp = (Emp) ois.readObject();
        System.out.println("反序列化完毕");
        System.out.println(emp);
        ois.close();
    }
}
```

6. 按照指定编码将文本写入文件

- **问题**

使用特定的编码，输出字符，并生成文本文件。文本文件的内容如图 - 1 所示：

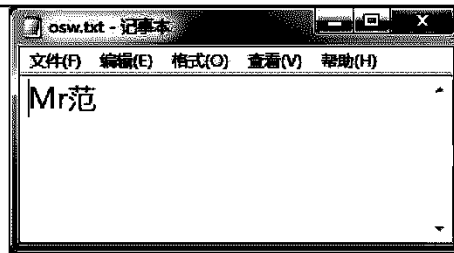


图 - 1

• 方案

首先需要创建文件字节输出流 `FileOutputStream` 类的对象，用于输出文件，代码如下所示：

```
FileOutputStream fos = new FileOutputStream("osw.txt");
```

然后，使用所创建的文件字节输出流对象 `fos`，作为参数，创建字符输出流 `OutputStreamWriter` 对象。需要注意的是，使用 `OutputStreamWriter` 时，最好指定字符集编码。如果不指定编码，则将使用平台默认的字符编码。代码如下所示：

```
OutputStreamWriter osw = new OutputStreamWriter(fos, "GBK");
```

随后，调用 `OutputStreamWriter` 的 `write` 方法，将字符串以指定的编码写入文件。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 `TestOSW` 类，添加测试方法 `testOSW` 方法

首先，新建名为 `TestOSW` 的类，并在类中添加单元测试方法 `testOSW`。代码如下所示：

```
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import org.junit.Test;

/**
 * 测试字符输出流
 */
public class TestOSW {
    /**
     * 按照指定编码将字符串写出
     * @throws Exception
     */
    @Test
    public void testOSW() throws Exception {
    }
}
```

步骤二：按照指定编码将文本写入文件

为了按照指定编码将文本写入文件，首先需要创建 `FileOutputStream` 类的对象用于输出文件；然后，创建 `OutputStreamWriter` 类的对象，并指定文件和编码，最后，调用该对象的 `write` 方法向文件中写入文本。注意：使用完毕后，关闭 `OutputStreamWriter` 对象。

代码如下所示：

```
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import org.junit.Test;

/**
 * 测试字符输出流
 */
public class TestOSW {
    /**
     * 按照指定编码将字符串写出
     * @throws Exception
     */
    @Test
    public void testOSW() throws Exception{

        FileOutputStream fos = new FileOutputStream("osw.txt");
        //根据指定编码写出字符串，编码名称忽略大小写
        OutputStreamWriter osw = new OutputStreamWriter(fos,"GBK");

        osw.write("Mr 范");
        osw.close();

    }
}
```

步骤三：测试

运行 `testOSW` 方法，将在当前工程下生成一个名为 `osw.txt` 的文件，打开该文件，文件中的内容为“Mr 范”。此文件为 4 字节，其中，一个中文字符为 2 个字节。

• 完整代码

本案例的完整代码如下所示：

```
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import org.junit.Test;

/**
 * 测试字符输出流
 */
public class TestOSW {
    /**
     * 按照指定编码将字符串写出
     * @throws Exception
     */
}
```

```
*/
@Test
public void testOSW() throws Exception{
    FileOutputStream fos = new FileOutputStream("osw.txt");
    //根据指定编码写出字符串，编码名称忽略大小写
    OutputStreamWriter osw = new OutputStreamWriter(fos, "GBK");

    osw.write("Mr 范");
    osw.close();
}
}
```

7. 读出指定编码的文本文件

• 问题

使用特定的编码，读取上一个案例中所创建的文本文件中的内容，并打印显示，效果如图 - 2 所示：

```
<terminated> TestISR [JUnit]
Mr 范
```

图 - 2

• 方案

首先需要创建文件字节输入流 `FileInputStream` 类的对象，用于读入文件，代码如下所示：

```
FileInputStream fis = new FileInputStream("osw.txt");
```

然后，使用所创建的文件字节输入流对象 `fis`，作为参数，创建字符输入流 `InputStreamReader` 类的对象，以读取字符。需要注意的是，使用 `InputStreamReader` 时，最好指定字符集编码。如果不指定编码，则将使用平台默认的字符编码。代码如下所示：

```
InputStreamReader isr = new InputStreamReader(fis, "GBK");
```

随后，调用 `InputStreamReader` 的 `read` 方法读取字符。需要注意的是，`read` 方法可以读取输入流中的下一个字符，并返回 `int` 值。如果因已到达流末尾而没有可读取的内容，则返回值 -1。因此，需要构建 `while` 循环来逐一读取文件中的每个字符，代码如下所示：

```
int chs = -1;
while((chs = isr.read()) != -1){
}
```

正因为 `InputStreamReader` 的 `read` 方法返回的是 `int` 数值,需要将其转换为字符后再输出,代码如下所示:

```
System.out.println((char)chs);
```

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：新建 `TestISR` 类，添加测试方法 `testISR` 方法

首先,新建名为 `TestISR` 的类,并在类中添加单元测试方法 `testISR`。代码如下所示:

```
import java.io.FileInputStream;
import java.io.InputStreamReader;
import org.junit.Test;

/**
 * 测试字符输入流
 */
public class TestISR {
    /**
     * 按照指定编码读取字符串
     * @throws Exception
     */
    @Test
    public void testISR()throws Exception{

    }
}
```

步骤二：按照指定编码读取文件

为了按照指定编码读取文件,首先创建 `FileInputStream` 类的对象用于读取文件;然后,创建 `InputStreamReader` 类的对象,并指定文件和编码,最后,循环调用该对象的 `read` 方法逐一读取文件中的字符并打印显示。注意:使用完毕后,需要关闭 `InputStreamReader` 对象。

代码如下所示:

```
import java.io.FileInputStream;
import java.io.InputStreamReader;
import org.junit.Test;

/**
 * 测试字符输入流
 */
public class TestISR {
    /**
     * 按照指定编码读取字符串
     * @throws Exception
     */
    @Test
    public void testISR()throws Exception{
```

```

        FileInputStream fis = new FileInputStream("osw.txt");
        //根据指定编码读取字符串，编码名称忽略大小写
        InputStreamReader isr = new InputStreamReader(fis,"GBK");
        int chs = -1;
        while((chs = isr.read()) != -1){
            System.out.println((char)chs);
        }
        isr.close();
    }
}

```

步骤三：测试

运行 testISR 方法，将打印显示"Mr 范"。

• 完整代码

本案例中，类 testISR 的完整代码如下所示：

```

import java.io.FileInputStream;
import java.io.InputStreamReader;
import org.junit.Test;

/**
 * 测试字符输入流
 */
public class TestISR {
    /**
     * 按照指定编码读取字符串
     * @throws Exception
     */
    @Test
    public void testISR()throws Exception{
        FileInputStream fis = new FileInputStream("osw.txt");
        //根据指定编码读取字符串，编码名称忽略大小写
        InputStreamReader isr = new InputStreamReader(fis,"GBK");
        int chs = -1;
        while((chs = isr.read()) != -1){
            System.out.println((char)chs);
        }
        isr.close();
    }
}

```

课后作业

1. 简述节点流和处理流的区别，以及 Java 流式输入输出的架构特点
2. 简述 RandomAccessFile 和 FileInputStream 及 FileOutputStream 在使用中的区别
3. 用自定义缓存区的方式实现文件的移动

使用 FileInputStream 类的 read(byte[])方法和 FileOutputStream 类的 write(byte[]) 方法实现文件移动。

4. 用缓冲流的方式实现文件的移动

使用 BufferedInputStream 类的 read 方法和 BufferedOutputStream 类的 write 方法实现文件移动。

5. 实现 empList 的序列化和反序列化

实现 empList 的序列化和反序列化，详细要求如下：

- 1) 在 List 集合中存放 Emp 类型的对象；
- 2) 将 List 集合序列化到文件 emplist.obj 中；
- 3) 从文件 emplist.obj 中将 List 集合反序列化取出并打印到控制台。

6. 简述 Serializable 接口和 transient 关键字的意义
7. 名称解释：ISO8859-1，GBK，UTF-8
8. 分别简述 ISR 和 OSW 的工作原理

Java 核心 API(下)

Unit03

知识体系.....Page 59

文件数据 IO 操作	PrintWriter	创建 PW 对象
		print 与 println 方法
		使用 PW 输出字符数据
	BufferedReader	构建 BufferedReader
		使用 BR 读取字符串
异常处理	异常处理概述	使用返回值状态标识异常
		异常处理机制
	异常的捕获和处理	Throwable,Error 和 Exception
		try-catch
		多个 catch
		finally 的作用
		throw 关键字
		throws 关键字
		重写方法时的 throws
	Java 异常 API	RuntimeException
		常见 RuntimeException
	Exception 常用 API	printStackTrace
		getMessage
		getCause
	自定义 Exception	自定义异常的意义
		继承 Exception 自定义异常
		如何编写构造方法

经典案例.....Page 67

将日志信息写入文本文件	创建 PW 对象
	print 与 println 方法
	使用 PW 输出字符数据
读取一个文本文件的所有行输出到控制台	构建 BufferedReader
	使用 BR 读取字符串


FileUtils 实现文件复制功能（包含异常的处理）	try-catch
	多个 catch
	finally 的作用
	throw 关键字
	throws 关键字
测试常见的 RuntimeException	常见 RuntimeException
FileUtils 实现文件复制功能（抛出自定义的文件复制异常）	自定义异常的意义
	继承 Exception 自定义异常
	如何编写构造方法

课后作业.....Page 87


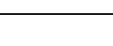
1. 文件数据 IO 操作

1.1. PrintWriter



1.1.1. 【PrintWriter】创建 PW 对象

<div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div>	<div style="text-align: right;">  </div> <h4 style="margin-top: 0;">创建PW对象</h4> <ul style="list-style-type: none"> PrintWriter是具有自动行刷新的缓冲字符输出流。其提供了比较丰富的构造方法。 <ul style="list-style-type: none"> 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。 <div style="text-align: right;">  </div>
---	--

1.1.2. 【PrintWriter】print 与 println 方法


<div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div>	<div style="text-align: right;">  </div> <h4 style="margin-top: 0;">print与println方法</h4> <ul style="list-style-type: none"> PrintWriter提供了丰富的重载print与println方法。其中println方法在于输出目标数据后自动输出一个系统支持的换行符。若该流是具有自动行刷新的，那么每当通过println方法写出的内容都会被实际写出，而不是进行缓存。 常用方法: <ul style="list-style-type: none"> 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 <div style="text-align: right;">  </div>
---	--

1.1.3. 【PrintWriter】使用 PW 输出字符数据


<div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div>	<div style="text-align: right;">  </div> <h4 style="margin-top: 0;">使用PW输出字符数据</h4> <pre> public void testPrintWriter()throws IOException{ 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!"); //立即写出该字符串 pw.close(); //关闭时，pw也会先进行flush } </pre> <div style="text-align: right;">  </div>
---	--

1.2. BufferedReader

1.2.1. 【BufferedReader】构建 BufferedReader

<div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div>	<div style="text-align: right;"></div> <h3>构建BufferedReader</h3> <ul style="list-style-type: none">• 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：<ul style="list-style-type: none">– BufferedReader(Reader reader) <div style="text-align: right;">+</div>
---	--


1.2.2. 【BufferedReader】使用 BR 读取字符串

<div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div>	<div style="text-align: right;"></div> <h3>使用BR读取字符串</h3> <ul style="list-style-type: none">• BufferedReader提供了一个可以便于读取一行字符串的方法：<ul style="list-style-type: none">– String readLine() <p>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。</p> <div style="text-align: right;">+</div>
---	--



2. 异常处理

2.1. 异常处理概述

2.1.1. 【异常处理概述】使用返回值状态标识异常



<div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div> <div style="border-bottom: 1px solid black; height: 15px; margin-bottom: 5px;"></div>	<div style="text-align: right;"></div> <h3>使用返回值状态标识异常</h3> <ul style="list-style-type: none">• 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。 <div style="text-align: right;">+</div>
---	---

2.1.2. 【异常处理概述】异常处理机制



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">异常处理机制</h4> <ul style="list-style-type: none"> 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。 <div style="text-align: right;">  </div>
---	--

2.2. 异常的捕获和处理

2.2.1. 【异常的捕获和处理】Throwable,Error 和 Exception

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">Throwable,Error和Exception</h4> <ul style="list-style-type: none"> Java异常结构中定义有Throwable类，Exception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；而Error表示Java运行时环境出现的错误，例如：JVM内存资源耗尽等。 <div style="text-align: right;">  </div>
---	---

2.2.2. 【异常的捕获和处理】try-catch

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">try-catch</h4> <ul style="list-style-type: none"> 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 如果没有异常产生，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 在catch中声明的异常对象（ 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 <div style="text-align: right;">  </div>
---	---

2.2.3. 【异常的捕获和处理】多个 catch

Tarena
达内科技

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

知识讲解

```
try{
    ...
} catch (NullPointerException e){
    ...
} catch (RuntimeException e){
    ...
} catch (Exception e){
    ...
}
```

子类型异常在前，父类型异常在后，这样的顺序依次捕获。否则编译不通过。

++

2.2.4. 【异常的捕获和处理】finally 的作用

Tarena
达内科技

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异常,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释放工作，如关闭打开的文件、删除临时文件等。

知识讲解

++

2.2.5. 【异常的捕获和处理】throw 关键字

Tarena
达内科技



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后抛出。
- 例如：
 - throw new ArithmeticException();



知识讲解

++

2.2.6. 【异常的捕获和处理】throws 关键字



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>throws关键字</h3> <ul style="list-style-type: none"> • 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，而希望调用这个方法的方法来统一处理，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： <pre>public static void stringToDate(String str) throws ParseException{ }</pre> <div style="text-align: right;">  </div>
---	---

2.2.7. 【异常的捕获和处理】重写方法时的 throws



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>重写方法时的throws</h3> <ul style="list-style-type: none"> • 如果使用继承时，在父类别的某个方法上宣告了throws某些异常，而在子类别中重新定义该方法时，您可以： <ul style="list-style-type: none"> – 不处理异常（重新定义时不设定throws） – 可仅throws父类别中声明的部分异常 – 可throws父类方法中抛出异常的子类异常 • 但是不可以： <ul style="list-style-type: none"> – throws出额外的异常 – throws父类方法中抛出异常的父类异常 <div style="text-align: right;">  </div>
---	--

2.3. Java 异常 API

2.3.1. 【Java 异常 API】RuntimeException



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>RuntimeException</h3> <p>Java异常可以分为可检测异常，非检测异常：</p> <ul style="list-style-type: none"> – 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 <ul style="list-style-type: none"> • 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。 <div style="text-align: right;">  </div>
---	---

2.3.2. 【Java 异常 API】常见 RuntimeException



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>常见 RuntimeException</h4> <ul style="list-style-type: none"> • 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。 <div style="text-align: right;">  </div>
---	--

2.4. Exception 常用 API



2.4.1. 【Exception 常用 API】printStackTrace

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>printStackTrace</h4> <ul style="list-style-type: none"> • 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() <pre> try{ ... }catch(Exception e){ e.printStackTrace();//输出执行堆栈信息 } </pre> <div style="text-align: right;">  </div>
---	--

2.4.2. 【Exception 常用 API】getMessage



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>getMessage</h4> <ul style="list-style-type: none"> • 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() <pre> try{ ... }catch(Exception e){ System.out.println(e.getMessage()); } </pre> <div style="text-align: right;">  </div>
---	---

2.4.3. 【Exception 常用 API】getCause



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>getCause</h4> <ul style="list-style-type: none"> 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 <code>getCause()</code> 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原异常。 Throwable <code>getCause()</code> <ul style="list-style-type: none"> 获取该异常出现的原因 <div style="text-align: right;">  </div>
---	---

2.5. 自定义 Exception

2.5.1. 【自定义 Exception】自定义异常的意义

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>自定义异常的意义</h4> <ul style="list-style-type: none"> 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。 <div style="text-align: right;">  </div>
---	--

2.5.2. 【自定义 Exception】继承 Exception 自定义异常

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>继承Exception自定义异常</h4> <ul style="list-style-type: none"> 创建自定义异常类, 语法格式: <pre>class [自定义异常类名] extends Exception{ ... }</pre> <div style="text-align: right;">  </div>
---	--

2.5.3. 【自定义 Exception】如何编写构造方法

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。

知识讲解

+

Technology
Tarena
达内科技

经典案例

1. 将日志信息写入文本文件

- 问题

生成一个文本文件 pw.txt，用于记录多行日志信息。pw.txt 文件的内容如图 - 1 所示：

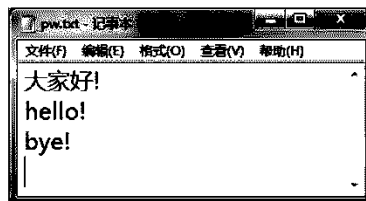


图 - 1

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 TestPrintWriter 类，添加测试方法 testPrintWriter 方法

首先，新建名为 TestPrintWriter 的类，并在类中添加单元测试方法 testPrintWriter。代码如下所示：

```
import java.io.PrintWriter;
import org.junit.Test;

/**
 * 测试缓冲字符输出流
 */
public class TestPrintWriter {
    /**
     * 测试缓冲字符输出流
     * @throws Exception
     */
    @Test
    public void testPrintWriter() throws Exception {

    }
}
```

步骤二：将多行文本写入文件

PrintWriter 是带有行刷新的缓冲字符输出流，它提供了丰富的重载 print 与 println 方法。其中，println 方法在于输出目标数据后自动输出一个系统支持的换行符。

为向文本文件中写入多行日志信息，首先需要创建一个 PrintWriter 类的对象，用于写入文件，然后调用 println 方法写入文本，最后关闭 PrintWriter 对象。代码如下所

示：

```
import java.io.PrintWriter;
import org.junit.Test;

/**
 * 测试缓冲字符输出流
 */
public class TestPrintWriter {
    /**
     * 测试缓冲字符输出流
     * @throws Exception
     */
    @Test
    public void testPrintWriter()throws Exception{

        PrintWriter pw = new PrintWriter("pw.txt");
        pw.println("大家好!");
        pw.println("hello!");
        pw.println("bye!");
        pw.close();

    }
}
```

步骤三：测试

运行 testPrintWriter 方法，将在当前工程下生成一个名为 pw.txt 的文件，打开该文件，文件中的内容和图 - 1 相同。

• 完整代码

本案例的完整代码如下所示：

```
import java.io.PrintWriter;
import org.junit.Test;

/**
 * 测试缓冲字符输出流
 */
public class TestPrintWriter {
    /**
     * 测试缓冲字符输出流
     * @throws Exception
     */
    @Test
    public void testPrintWriter()throws Exception{
        PrintWriter pw = new PrintWriter("pw.txt");

        pw.println("大家好!");
        pw.println("hello!");
        pw.println("bye!");
        pw.close();

    }
}
```

2. 读取一个文本文件的所有行输出到控制台

• 问题

读取上一个案例中所创建的文本文件中的内容，并逐行打印显示，效果如图 - 2 所示：

```
<terminated> TestBufferedReader [JU
大家好!
hello!
bye!
```

图 - 2

• 方案

首先需要创建文件字节输入流 `FileInputStream`，用于读入文件，代码如下：

```
FileInputStream fis = new FileInputStream("pw.txt");
```

然后使用所创建的文件字节输入流对象作为参数，构建 `InputStreamReader` 类的对象，用于将字节流变为字符流，代码如下：

```
InputStreamReader isr = new InputStreamReader(fis);
```

再使用所创建的字符输入流对象作为参数，创建 `BufferedReader` 类的对象，用于逐行读取文本文件的所有行，代码如下：

```
BufferedReader br = new BufferedReader(isr);
```

随后，调用 `BufferedReader` 的 `readLine` 方法读取一行文本。需要注意的是，如果因已到达流末尾而没有可读的内容，`readLine` 则返回 `null`。因此，需要构建 `while` 循环来逐一读取文件中的每行文本，代码如下：

```
String line = null;
while((line = br.readLine()) != null){
    ...
}
```

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 `TestBufferedReader` 类，添加测试方法 `testBufferedReader` 方法

首先，新建名为 `TestBufferedReader` 的类，并在类中添加单元测试方法 `testBufferedReader`。代码如下所示：

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import org.junit.Test;

/**
 * 测试缓冲字符输入流
 */
public class TestBufferedReader {
    /**
     * 使用缓冲字符输入流读取文本文件
     * @throws Exception
     */
    @Test
    public void testBufferedReader()throws Exception{

    }
}
```

步骤二：读取文本文件的所有行

创建 `BufferedReader` 类的对象，并调用该对象的 `readLine` 方法逐一读取文件中的每行文本并打印显示。注意：使用完毕后，需要关闭 `BufferedReader` 对象。

代码如下所示：

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import org.junit.Test;

/**
 * 测试缓冲字符输入流
 */
public class TestBufferedReader {
    /**
     * 使用缓冲字符输入流读取文本文件
     * @throws Exception
     */
    @Test
    public void testBufferedReader()throws Exception{

        FileInputStream fis = new FileInputStream("pw.txt");
        InputStreamReader isr = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(isr);

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

        br.close();

    }
}
```

步骤三：测试

运行 testBufferedReader 方法，将逐行打印显示文件中的内容。

- **完整代码**

本案例的完整代码如下所示：

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import org.junit.Test;

/**
 * 测试缓冲字符输入流
 */
public class TestBufferedReader {
    /**
     * 使用缓冲字符输入流读取文本文件
     * @throws Exception
     */
    @Test
    public void testBufferedReader()throws Exception{
        FileInputStream fis = new FileInputStream("pw.txt");
        InputStreamReader isr = new InputStreamReader(fis);
        BufferedReader br = new BufferedReader(isr);

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

        br.close();
    }
}
```

3. FileUtils 实现文件复制功能（包含异常的处理）

- **问题**

创建工具类 FileUtils，为其定义文件复制方法，实现文件复制功能，并添加异常处理机制。

- **方案**

前面的案例中，已经描述过文件复制功能的实现，此案例需要为其添加异常处理机制，用来处理复制过程中的各种异常情况，如需要复制的文件不存在等。

需要使用 try 语句捕获代码可能会产生的异常对象，并使用 catch 语句对异常做相应的处理。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，代码如下所示：

```
FileInputStream fis = null;
FileOutputStream fos = null;
try {
    //... 代码
```

```
    } catch (FileNotFoundException e) {
        System.err.println("文件没有找到");
        e.printStackTrace();
    } catch (IOException e) {
        System.err.println("读写异常");
        e.printStackTrace();
    }
}
```

上述代码中，可以捕获处理 `FileNotFoundException` 异常，以及 `IOException`。

`finally` 语句为异常处理提供一个统一的出口，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。代码如下所示：

```
FileInputStream fis = null;
FileOutputStream fos = null;
try {
    //... 代码
} catch (FileNotFoundException e) {
    //... 处理异常的代码
} catch (IOException e) {
    //...处理异常的代码
} finally {
    if (fis != null) {
        try {
            fis.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    if (fos != null) {
        try {
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 `FileUtils` 类，添加测试方法 `testCopy` 方法

首先，新建名为 `FileUtils` 的类，并在类中添加单元测试方法 `testCopy`。代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.junit.Test;

public class FileUtils {
    /**
     * 对复制文件的代码添加异常捕获机制
     */
    @Test
    public void testCopy() {
```

```
}  
}
```

步骤二：实现文件复制功能

为方法 testCopy 添加代码，实现文件复制功能，代码如下所示：

```
public void testCopy() {  
  
    FileInputStream fis = null;  
    FileOutputStream fos = null;  
    fis = new FileInputStream("fos.dat");  
    fos = new FileOutputStream("fos_copy4.dat");  
    int d = -1;  
    while ((d = fis.read()) != -1) {  
        fos.write(d);  
    }  
    System.out.println("复制完毕");  
  
}
```

步骤三：捕获并处理异常

为上一步骤中的代码添加 try/catch 语句，以实现异常的处理。

先处理 FileNotFoundException 异常，然后处理 IOException 异常，代码如下所示：

```
public void testCopy() {  
    FileInputStream fis = null;  
    FileOutputStream fos = null;  
  
    try {  
  
        fis = new FileInputStream("fos.dat");  
        fos = new FileOutputStream("fos_copy4.dat");  
        int d = -1;  
        while ((d = fis.read()) != -1) {  
            fos.write(d);  
        }  
        System.out.println("复制完毕");  
  
    } catch (FileNotFoundException e) {  
        System.err.println("文件没有找到");  
        e.printStackTrace();  
    } catch (IOException e) {  
        System.err.println("读写异常");  
        e.printStackTrace();  
    }  
  
}
```

步骤四：资源清理

使用 finally 语句释放相关资源，代码如下所示：


```
public void testCopy() {
    FileInputStream fis = null;
    FileOutputStream fos = null;
    try {
        fis = new FileInputStream("fos.dat");
        fos = new FileOutputStream("fos copy4.dat");
        int d = -1;
        while ((d = fis.read()) != -1) {
            fos.write(d);
        }
        System.out.println("复制完毕");
    } catch (FileNotFoundException e) {
        System.err.println("文件没有找到");
        e.printStackTrace();
    } catch (IOException e) {
        System.err.println("读写异常");
        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();
            }
        }
    }
}
```

步骤五：测试

运行 testCopy 方法，如果当前工程目录下没有文件 fos.dat，则将产生 FileNotFoundException 异常。

• 完整代码

本案例中，类 FileUtils 的完整代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.junit.Test;

public class FileUtils {
    /**
     * 对复制文件的代码添加异常捕获机制
     */
    @Test
    public void testCopy() {
        FileInputStream fis = null;
        FileOutputStream fos = null;
```

```

    try {
        fis = new FileInputStream("fos.dat");
        fos = new FileOutputStream("fos_copy4.dat");
        int d = -1;
        while ((d = fis.read()) != -1) {
            fos.write(d);
        }
        System.out.println("复制完毕");
    } catch (FileNotFoundException e) {
        System.err.println("文件没有找到");
        e.printStackTrace();
    } catch (IOException e) {
        System.err.println("读写异常");
        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();
            }
        }
    }
}
}

```

4. 测试常见的 RuntimeException

• 问题

写代码，制造常见的 RuntimeException 异常，详细要求如下：

- 1) 制造空指针异常 NullPointerException；
- 2) 制造数组越界异常 ArrayIndexOutOfBoundsException；
- 3) 制造数学异常 ArithmeticException；
- 4) 制造强制类型转换异常 ClassCastException；
- 5) 制造数值格式化异常 NumberFormatException

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建类

新建类 TestRuntimeException，代码如下所示：

```

/**
 * 测试常见的运行时异常
 */
public class TestRuntimeException {
}

```

步骤二：制造空指针异常

当一个引用的值为 null 的时候 如果通过引用访问对象成员变量或者调用方法 此时，会产生 NullPointerException。在此，首先创建测试方法 testNullPointerException；然后，将 String 类的引用 str 赋值为 null；最后，调用 String 类的 length 方法即会发生空指针异常，代码如下所示：

```
import org.junit.Test;

/**
 * 测试常见的运行时异常
 */
public class TestRuntimeException {

    /**
     * 测试空指针异常
     */
    @Test
    public void testNullPointerException(){
        String str = null;
        //会引发空指针异常
        System.out.println(str.length());
    }
}
```

运行 testNullPointerException 方法，在 JUnit 视图下会显示抛出了空指针异常，如图-3 所示。

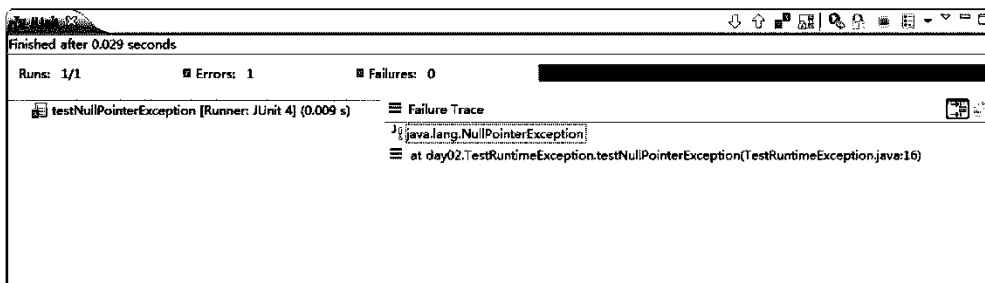


图- 3

步骤三：制造数组下标越界异常

如果使用为负或大于等于数组大小的索引来访问数组时抛出的异常。在此，首先创建测试方法 testArrayIndexOutOfBoundsException；然后，使用 String 类的 getBytes 方法返回字符串 “hello” 对应的字符数组；最后，输出该字符数组中索引为字符数组长度的字符，此操作即会发生数组下标越界异常，代码如下所示：

```
import org.junit.Test;

/**
 * 测试常见的运行时异常
 */
public class TestRuntimeException {
```

```
/**
 * 测试空指针异常
 */
@Test
public void testNullPointerException(){
    //... (代码略)
}

/**
 * 测试数组下标越界异常
 */
@Test
public void testArrayIndexOutOfBoundsException(){
    byte[] bytes = "hello".getBytes();
    //会引发数组下标越界异常
    System.out.println(bytes[bytes.length]);
}
}
```

运行 testArrayIndexOutOfBoundsException 方法，在 JUnit 视图下会显示数组下标越界异常，如图-4 所示。

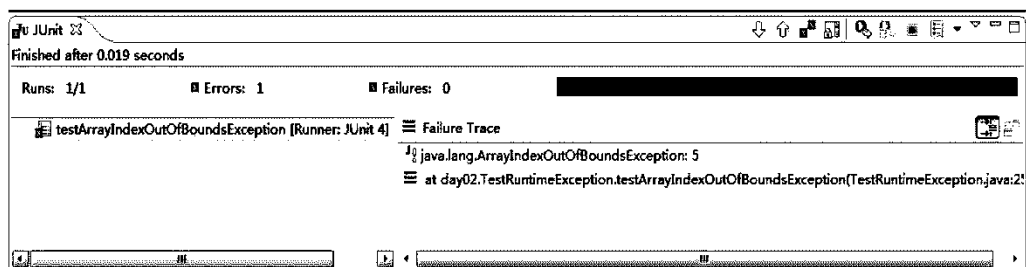


图- 4

步骤四：制造强制类型转换异常

当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此异常。在此，首先创建测试方法 testArithmeticException；然后，输出“5 除以 0”即会发生数学异常，代码如下所示：

```
import org.junit.Test;
/**
 * 测试常见的运行时异常
 */
public class TestRuntimeException {
    /**
     * 测试空指针异常
     */
    @Test
    public void testNullPointerException(){
        //... (代码略)
    }
    /**
     * 测试数组下标越界异常
     */
}
```

```
@Test
public void testArrayIndexOutOfBoundsException(){
    //... (代码略)
}

/**
 * 测试数学异常
 */
@Test
public void testArithmeticException(){
    //会引发数学异常
    System.out.println(5/0);
}
}
```

运行 testArithmeticException 方法，在 JUnit 视图下会显示抛出了数学异常，如图-5 所示。

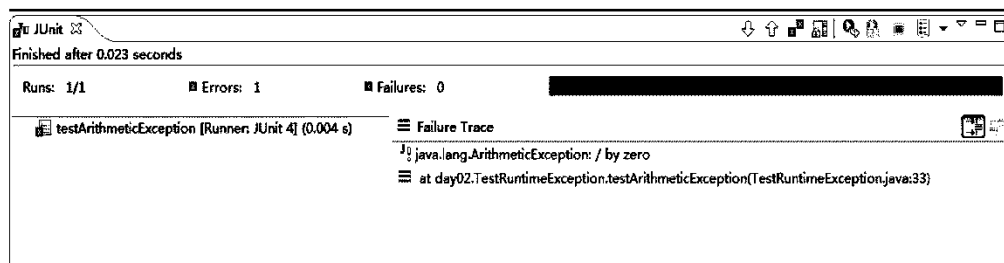


图- 5

步骤五：制造强制类型转换异常

当试图将对象强制转换为不是实例的子类时，抛出该异常。在此，首先创建测试方法 testClassCastException 然后，创建一个 Object 类的引用 obj 赋值为字符串“hello”；最后将对象 obj 强制转换为 Integer 类的对象即会发生强制类型转换异常，代码如下所示：

```
import org.junit.Test;
/**
 * 测试常见的运行时异常
 */
public class TestRuntimeException {
    /**
     * 测试空指针异常
     */
    @Test
    public void testNullPointerException(){
        //... (代码略)
    }
    /**
     * 测试数组下标越界异常
     */
    @Test
    public void testArrayIndexOutOfBoundsException(){
        //... (代码略)
    }
}
```

```

    * 测试数学异常
    */
    @Test
    public void testArithmeticException(){
        //... (代码略)
    }

    /**
     * 测试强制类型转换异常
     */
    @Test
    public void testClassCastException(){
        Object obj = "hello";
        //会引发强制类型转换异常
        Integer i = (Integer)obj;
    }
}

```

运行 testClassCastException 方法，在 JUnit 视图下会显示抛出了类型转换异常，如图 - 6 所示。

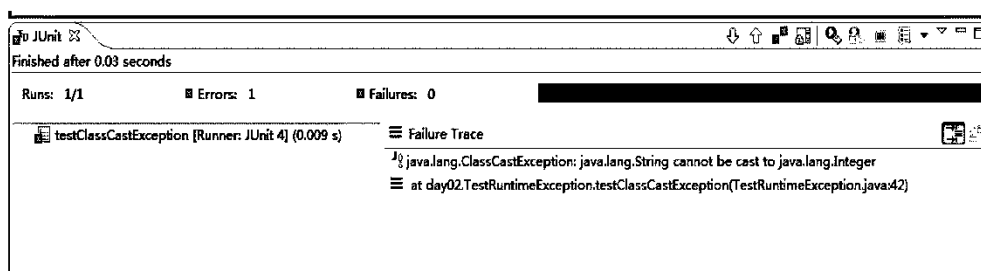


图 - 6

步骤六：制造 NumberFormat 异常

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。在此，首先创建测试方法 testNumberFormatException；然后，使用 Integer 类的 parseInt 方法将字符串“a”转换为数值即会发生 NumberFormat 异常，代码如下所示：

```

import org.junit.Test;
/**
 * 测试常见的运行时异常
 */
public class TestRuntimeException {
    /**
     * 测试空指针异常
     */
    @Test
    public void testNullPointerException(){
        //... (代码略)
    }

    /**
     * 测试数组下标越界异常
     */
}

```

```

@Test
public void testArrayIndexOutOfBoundsException(){
    //... (代码略)
}
/**
 * 测试数学异常
 */
@Test
public void testArithmeticException(){
    //... (代码略)
}
/**
 * 测试强制类型转换异常
 */
@Test
public void testClassCastException(){
    //... (代码略)
}

/**
 * 测试 NumberFormat 异常
 */
@Test
public void testNumberFormatException(){
    //会引发 NumberFormat 异常
    int num = Integer.parseInt("a");
}
}

```

运行 testNumberFormatException 方法，在 JUnit 视图下会显示抛出了 NumberFormat 异常，如图 - 7 所示。

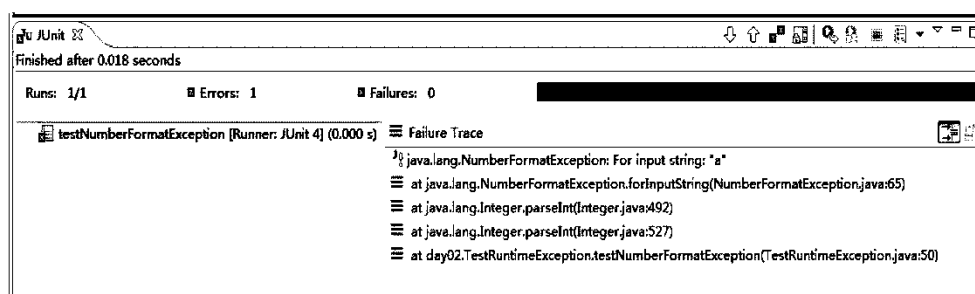


图 - 7

• 完整代码

本案例中，类 TestRuntimeException 的完整代码如下所示：

```

import org.junit.Test;

/**
 * 测试常见的运行时异常
 */
public class TestRuntimeException {
    /**

```

```

    * 测试空指针异常
    */
    @Test
    public void testNullPointerException(){
        String str = null;
        //会引发空指针异常
        System.out.println(str.length());
    }
    /**
    * 测试数组下标越界异常
    */
    @Test
    public void testArrayIndexOutOfBoundsException(){
        byte[] bytes = "".getBytes();
        //会引发数组下标越界异常
        System.out.println(bytes[bytes.length]);
    }
    /**
    * 测试数学异常
    */
    @Test
    public void testArithmeticException(){
        //会引发数学异常
        System.out.println(5/0);
    }
    /**
    * 测试强制类型转换异常
    */
    @Test
    public void testClassCastException(){
        Object obj = "hello";
        //会引发强制类型转换异常
        Integer i = (Integer)obj;
    }
    /**
    * 测试 NumberFormat 异常
    */
    @Test
    public void testNumberFormatException(){
        //会引发 NumberFormat 异常
        int num = Integer.parseInt("a");
    }
}

```

5. FileUtils 实现文件复制功能（抛出自定义的文件复制异常）

• 问题

在上一案例的基础上，在工具类 FileUtils 中再次定义文件复制方法，实现文件复制功能，并添加自定义异常处理。

• 方案

实现和使用自定义异常的过程如下：

1) 首先，新建自定义异常类 CopyException，该类继承自 Exception；然后在该类中添加以下四个构造方法，这四个构造方法的实现，都为使用 super 调用父类对应参数的构造方法即可。以下是自定义异常一般的实现方式，代码如下：


```
public class CopyException extends Exception{
    public CopyException() {
        super();
    }
    public CopyException(String message, Throwable cause) {
        super(message, cause);
    }
    public CopyException(String message) {
        super(message);
    }
    public CopyException(Throwable cause) {
        super(cause);
    }
}
```

2) 为了屏蔽底层的 IOException, 在实现文件复制的异常处理时, 抛出自定义异常 CopyException, 即在 catch 块内使用 throw 关键字抛出自定义异常 CopyException 的实例, 代码如下:

```
try {
    //实现文件复制
} catch (FileNotFoundException e) {
    //抛出自定义异常
    throw new CopyException("文件没有找到",e);
} catch (IOException e) {
    //抛出自定义异常
    throw new CopyException("读写异常",e);
}
```

3) 在实现复制的方法声明处使用 throws 关键字声明该方法会抛出自定义异常 CopyException, 代码如下:

```
public void testCopy1()throws CopyException{... ...}
```

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建自定义异常 CopyException

首先, 新建自定义异常类 CopyException, 该类继承自 Exception; 然后在该类中添加以下四个构造方法, 这四个构造方法的实现都为使用 super 调用父类对应参数的构造方法, 代码如下所示:

```
/**
 * 自定义异常
 */
public class CopyException extends Exception{
    public CopyException() {
        super();
    }
    public CopyException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
public CopyException(String message) {
    super(message);
}
public CopyException(Throwable cause) {
    super(cause);
}
}
```

步骤二：添加测试方法

在类 FileUtils 中添加测试方法 testCopy1，在该方法中实现文件复制功能，代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.junit.Test;

public class FileUtils {
    /**
     * 对复制文件的代码添加异常捕获机制
     */

    @Test
    public void testCopy1()throws CopyException{
        FileInputStream fis = null;
        FileOutputStream fos = null;
        fis = new FileInputStream("fos.dat");
        fos = new FileOutputStream("fos_copy4.dat");
        int d = -1;
        while ((d = fis.read()) != -1) {
            fos.write(d);
        }
        System.out.println("复制完毕");
    }
}
```

步骤三：捕获并处理异常

为上一步骤中的代码添加自定义异常处理。首先，在 catch 块内使用 throw 关键字抛出自定义异常 CopyException 的实例，然后，在 testCopy1 方法声明处使用 throws 关键字声明该方法会抛出自定义异常 CopyException，代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.junit.Test;

public class FileUtils {
    /**
     * 对复制文件的代码添加异常捕获机制
     */

    @Test

    public void testCopy1()throws CopyException{
```

```
FileInputStream fis = null;
FileOutputStream fos = null;

try {

    fis = new FileInputStream("fos.dat");
    fos = new FileOutputStream("fos copy4.dat");
    int d = -1;
    while ((d = fis.read()) != -1) {
        fos.write(d);
    }
    System.out.println("复制完毕");

} catch (FileNotFoundException e) {
    //抛出自定义异常
    throw new CopyException("文件没有找到",e);
} catch (IOException e) {
    //抛出自定义异常
    throw new CopyException("读写异常",e);
}

}
```

步骤四：资源清理

使用 finally 块释放相关资源，及时将流 fis、fos 关闭，代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.junit.Test;

public class FileUtils {
    /**
     * 对复制文件的代码添加异常捕获机制
     */
    @Test
    public void testCopy1()throws CopyException{
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream("fos.dat");
            fos = new FileOutputStream("fos copy4.dat");
            int d = -1;
            while ((d = fis.read()) != -1) {
                fos.write(d);
            }

            System.out.println("复制完毕");
        } catch (FileNotFoundException e) {
            //抛出自定义异常
            throw new CopyException("文件没有找到",e);
        } catch (IOException e) {
            //抛出自定义异常
            throw new CopyException("读写异常",e);
        }
    }
}
```

```

    } finally {
        if (fis != null) {
            try {
                fis.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (fos != null) {
            try {
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

FileUtils.java

步骤五：测试

运行 testCopy1 方法，如果当前工程目录下没有文件 fos.dat，则将抛出 CopyException 异常，如图 - 8 所示。

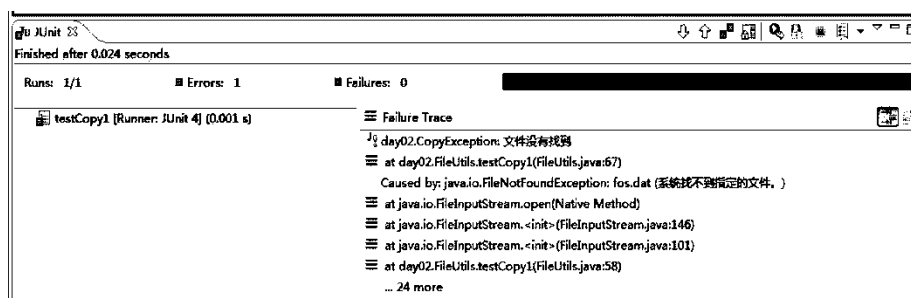


图 - 8

完整代码

本案例中，类 CopyException 的完整代码如下所示：

```

/**
 * 自定义异常
 */
public class CopyException extends Exception {
    public CopyException() {
        super();
    }
    public CopyException(String message, Throwable cause) {
        super(message, cause);
    }
    public CopyException(String message) {
        super(message);
    }
    public CopyException(Throwable cause) {
        super(cause);
    }
}

```

类 FileUtils 的完整代码如下所示：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import org.junit.Test;

public class FileUtils {
    //... (之前案例的代码, 略)

    /**
     * 对复制文件的代码添加异常捕获机制
     */
    @Test
    public void testCopy1() throws CopyException {
        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream("fos.dat");
            fos = new FileOutputStream("fos_copy4.dat");
            int d = -1;
            while ((d = fis.read()) != -1) {
                fos.write(d);
            }
            System.out.println("复制完毕");
        } catch (FileNotFoundException e) {
            //抛出自定义异常
            throw new CopyException("文件没有找到", e);
        } catch (IOException e) {
            //抛出自定义异常
            throw new CopyException("读写异常", e);
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            if (fos != null) {
                try {
                    fos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

课后作业

1. 读取文本文件中的每一行数据信息，求其和再存入最后一行

文本文件 pw.txt 的内容如下所示：

大家好!
hello!
bye!

操作完毕之后的文本文件 pw.txt 的内容如下所示：

大家好!
hello!
bye!
大家好!hello!bye!

2. 简述 Error 和 Exception 的区别

3. 下面关于异常处理的说法正确的是

- A.try ... catch ... catch，当多个 catch 时,后一个 catch 捕获类型一定是前一个的父类。
- B.try...finally 可以组合使用。
- C.throw 抛出的一些异常，程序不进行处理，程序编译也无错误。
- Dthrows 一定是写在方法后面。

4. 改正下面代码的编译错误

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;

public class Foo {
    public void writeData(){
        try {
            FileOutputStream fos= new FileOutputStream("out.dat");
            fos.write(1);
            fos.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

5. 简述 RuntimeException 和非 RuntimeException 的区别
6. 写出 5 个常见的 RuntimeException
7. printStackTrace 方法的作用在于？

Java 核心 API(下)

Unit04

知识体系.....Page 91

多线程基础	进程与线程	什么是进程
		什么是线程
		进程与线程的区别
		线程使用的场合
		并发原理
		线程状态
	创建线程	使用 Thread 创建线并启动线程
		使用 Runnable 创建并启动线程
		使用内部类创建线程
	线程操作 API	Thread.currentThread 方法
		获取线程信息
		线程优先级
		守护线程
		sleep 方法
		yield 方法
		join 方法
	线程同步	synchronized 关键字
		锁机制
		选择合适的锁对象
		选择合适的锁范围
		静态方法锁
		wait 和 notify

经典案例.....Page 99

创建两个线程，分别输出 1~100	使用 Thread 创建并启动线程
编写一个线程改变窗体的颜色 V1	使用 Runnable 创建并启动线程
	使用内部类创建线程
测试 currentThread 方法	Thread.currentThread 方法
测试 getName 方法和 getId 方法	获取线程信息



测试守护线程	守护线程
编写一个线程改变窗体的颜色 V2	sleep 方法
测试join 方法	join 方法

课后作业.....Page 117



1. 多线程基础

1.1. 进程与线程



1.1.1. 【进程与线程】什么是进程

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">什么是进程</h4> <ul style="list-style-type: none"> • 进程是操作系统中运行的一个任务(一个应用程序运行在一个进程中)。 • 进程 (process) 是一块包含了某些资源的内存区域。操作系统利用进程把它的工作划分为一些功能单元。 • 进程中所包含的一个或多个执行单元称为线程 (thread)。进程还拥有私有的虚拟地址空间, 该空间仅能被它所包含的线程访问。 • 线程只能归属于一个进程并且它只能访问该进程所拥有的资源。当操作系统创建一个进程后, 该进程会自动申请一个名为主线程或首要线程的线程。 <div style="text-align: right;">  </div>
---	---


1.1.2. 【进程与线程】什么是线程

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">什么是线程</h4> <ul style="list-style-type: none"> • 一个线程是进程的一个顺序执行流。 • 同类的多个线程共享一块内存空间和一组系统资源, 线程本身有一个供程序执行时的堆栈。线程在切换时负荷小, 因此, 线程也被称为轻负荷进程。一个进程中可以包含多个线程。 <div style="text-align: right;">  </div>
---	---

1.1.3. 【进程与线程】进程与线程的区别

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">进程与线程的区别</h4> <ul style="list-style-type: none"> • 一个进程至少有一个线程。 线程的划分尺度小于进程, 使得多线程程序的并发性高。另外, 进程在执行过程中拥有独立的内存单元, 而多个线程共享内存, 从而极大地提高了程序的运行效率。线程在执行过程中与进程的区别在于每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行, 必须依存在应用程序中, 由应用程序提供多个线程执行控制。 从逻辑角度来看, 多线程的意义在于一个应用程序中, 有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用来实现进程的调度和管理以及资源分配。 <div style="text-align: right;">  </div>
---	--

1.1.4. 【进程与线程】线程使用的场合




线程使用的场合

- 线程通常用于在一个程序中需要同时完成多个任务的情况。我们可以将每个任务定义为一个线程，使他们得以一同工作。
- 也可以用于在单一线程中可以完成，但是使用多线程可以更快的情况。比如下载文件。

+

1.1.5. 【进程与线程】并发原理




并发原理

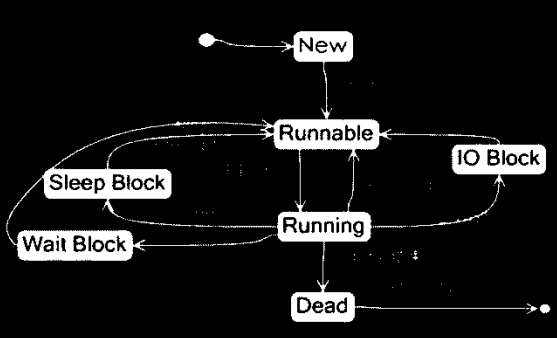
- 多个线程“同时”运行只是我们感官上的一种表现。事实上线程是并发的，OS将时间划分为很多时间片段（时间片），尽可能均匀分配给每一个线程，获取时间片段的线程被CPU运行，而其他线程全部等待。所以微观上是走走停停的，宏观上都在运行。这种现象叫并发，但不是绝对意义上的“同时发生”。

+

1.1.6. 【进程与线程】线程状态





线程状态





+

1.2. 创建线程



1.2.1. 【创建线程】使用 Thread 创建并启动线程

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">使用Thread创建并启动线程</h4> <ul style="list-style-type: none"> Thread类是线程类，其每一个实例表示一个可以并发运行的线程。我们可以通过继承该类并重写run方法来定义一个具体的线程。其中重写run方法的目的是定义该线程要执行的逻辑。启动线程时调用线程的start()方法而非直接调用run()方法。start()方法会将当前线程纳入线程调度，使当前线程可以开始并发运行。当线程获取时间片段后会自动开始执行run方法中的逻辑。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	---

1.2.2. 【创建线程】使用 Runnable 创建并启动线程



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">使用Runnable创建并启动线程</h4> <ul style="list-style-type: none"> 实现Runnable接口并重写run方法来定义线程体，然后在创建线程的时候将Runnable的实例传入并启动线程。 这样做的好处在于可以将线程与线程要执行的任务分离减少耦合，同时java是单继承的，定义一个类实现Runnable接口这样的做法可以更好的去实现其他父类或接口。因为接口是多继承关系。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	--

1.2.3. 【创建线程】使用内部类创建线程



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">使用内部类创建线程</h4> <ul style="list-style-type: none"> 通常我们可以通过匿名内部类的方式创建线程，使用该方式可以简化编写代码的复杂度，当一个线程仅需要一个实例时我们通常使用这种方式来创建。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	---

1.3. 线程操作 API



1.3.1. 【线程操作 API】Thread.currentThread 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">Thread.currentThread方法</h4> <ul style="list-style-type: none"> • Thread的静态方法currentThread方法可以用于获取运行当前代码片段的线程。 • Thread current = Thread.currentThread(); <div style="text-align: right;">  </div>
---	--



1.3.2. 【线程操作 API】获取线程信息

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">获取线程信息</h4> <ul style="list-style-type: none"> • Thread提供了 获取线程信息的相关方法: <ul style="list-style-type: none"> - long getId():返回该线程的标识符 - String getName():返回该线程的名称 - int getPriority():返回线程的优先级 - Thread.state getState():获取线程的状态 - boolean isAlive():测试线程是否处于活动状态 - boolean isDaemon():测试线程是否为守护线程 - boolean isInterrupted():测试线程是否已经中断 <div style="text-align: right;">  </div>
---	--



1.3.3. 【线程操作 API】线程优先级

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">线程优先级</h4> <ul style="list-style-type: none"> • 线程的切换是由线程调度控制的，我们无法通过代码来干涉，但是我们可以通过提高线程的优先级来最大程度的改善线程获取时间片的几率。 • 线程的优先级被划分为10级，值分别为1-10，其中1最低，10最高。线程提供了3个常量来表示最低，最高，以及默认优先级: <ul style="list-style-type: none"> - Thread.MIN_PRIORITY, - Thread.MAX_PRIORITY, - Thread.NORM_PRIORITY <p>void setPriority(int priority):设置线程的优先级。</p> <div style="text-align: right;">  </div>
---	---

1.3.4. 【线程操作 API】守护线程

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">守护线程</h3> <ul style="list-style-type: none"> 守护线程与普通线程在表现上没有什么区别，我们只需要通过Thread提供的方法来设定即可： <ul style="list-style-type: none"> void setDaemon(boolean) 当参数为true时该线程为守护线程。 守护线程的特点是，当进程中只剩下守护线程时，所有守护线程强制终止。 GC就是运行在一个守护线程上的。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	---

1.3.5. 【线程操作 API】sleep 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">sleep方法</h3> <ul style="list-style-type: none"> Thread的静态方法sleep用于使当前线程进入阻塞状态： <ul style="list-style-type: none"> static void sleep(long ms) 该方法会使当前线程进入阻塞状态指定毫秒，当阻塞指定毫秒后，当前线程会重新进入Runnable状态，等待分配时间片。 该方法声明抛出一个InterruptedException。所以在使用该方法时需要捕获这个异常。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	--

1.3.6. 【线程操作 API】yield 方法


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">yield方法</h3> <ul style="list-style-type: none"> Thread的静态方法yield: <ul style="list-style-type: none"> static void yield() 该方法用于使当前线程主动让出当前CPU时间片回到Runnable状态，等待分配时间片。 <div style="text-align: right;">  </div> <div style="text-align: right;">+</div>
---	--

1.3.7. 【线程操作 API】join 方法


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>join方法</h4> <ul style="list-style-type: none"> Thread的方法join: <ul style="list-style-type: none"> void join() 该方法用于等待当前线程结束。 该方法声明抛出InterruptedException。 <div style="text-align: right;">++</div>
---	--

1.4. 线程同步



1.4.1. 【线程同步】synchronized 关键字

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>synchronized关键字</h4> <ul style="list-style-type: none"> 多个线程并发读写同一个临界资源时会发生“线程并发安全问题” 常见的临界资源: <ul style="list-style-type: none"> 多线程共享实例变量 多线程共享静态公共变量 若想解决线程安全问题，需要将异步的操作变为同步操作。 <ul style="list-style-type: none"> 异步操作:多线程并发的操作，相当于各干各的。 同步操作:有先后顺序的操作，相当于你干完我再干。 <p>synchronized关键字是java中的同步锁</p> <div style="text-align: right;">++</div>
---	--



1.4.2. 【线程同步】锁机制

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>锁机制</h4> <ul style="list-style-type: none"> Java提供了一种内置的锁机制来支持原子性： 同步代码块(synchronized 关键字)，同步代码块包含两部分：一个作为锁的对象的引用，一个作为由这个锁保护的代码块。 synchronized (同步监视器—锁对象引用){ //代码块 } 若方法所有代码都需要同步也可以给方法直接加锁。 每个Java对象都可以用做一个实现同步的锁，线程进入同步代码块之前会自动获得锁，并且在退出同步代码块时自动释放锁，而且无论是通过正常途径退出还是通过抛异常退出都一样，获得内置锁的唯一途径就是进入由这个锁保护的同步代码块或方法。 <div style="text-align: right;">++</div>
---	--



1.4.3. 【线程同步】选择合适的锁对象

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">选择合适的锁对象</h4> <ul style="list-style-type: none"> 使用synchronized需要对一个对象上锁以保证线程同步。那么这个所对象应当注意: <ul style="list-style-type: none"> 多个需要同步的线程在访问该同步块时，看到的应该是同一个所对象引用。否则达不到同步效果。 通常我们会使用this来作为锁对象。 <div style="text-align: right;">  </div> <div style="text-align: center;">+</div>
---	---

1.4.4. 【线程同步】选择合适的锁范围

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">选择合适的锁范围</h4> <ul style="list-style-type: none"> 在使用同步块时，应当尽量在允许的情况下减少同步范围，以提高并发的执行效率。 <div style="text-align: right;">  </div> <div style="text-align: center;">+</div>
---	--

1.4.5. 【线程同步】静态方法锁

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">静态方法锁</h4> <ul style="list-style-type: none"> 当我们对一个静态方法加锁，如: <ul style="list-style-type: none"> public synchronized static void xxx(){....} 那么该方法锁的对象是类对象。每个类都有唯一的一个类对象。获取类对象的方式:类名.class。 静态方法与非静态方法同时声明了synchronized，他们之间是非互斥关系的。原因在于，静态方法锁的是类对象而非静态方法锁的是当前方法所属对象。 <div style="text-align: right;">  </div> <div style="text-align: center;">+</div>
---	---

1.4.6. 【线程同步】wait 和 notify

wait和notify



知识讲解

- 多线程之间需要协调工作。
- 例如，浏览器的一个显示图片的 `displayThread` 想要执行显示图片的任务，必须等待下载线程 `downloadThread` 将该图片下载完毕。如果图片还没有下载完，`displayThread` 可以暂停，当 `downloadThread` 完成了任务后，再通知 `displayThread` “图片准备完毕，可以显示了”，这时，`displayThread` 继续执行。
- 以上逻辑简单的说就是：如果条件不满足，则等待。当条件满足时，等待该条件的线程将被唤醒。在 Java 中，这个机制的实现依赖于 `wait/notify`。等待机制与锁机制是密切关联的。



经典案例

1. 创建两个线程，分别输出 1~100

- 问题

使用 Thread 创建两个线程，分别输出 1~100。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 TestThread 类及 MyThread 类

首先，新建类 TestThread；然后，在该类所在的文件中新建类 MyThread，MyThread 类继承自 Thread 类，代码如下所示：

```
/**
 * 测试线程
 */
public class TestThread {
    /**
     * 测试多线程并发运行
     */
    public static void main(String[] args) {
    }
}
/**
 * 线程
 */
class MyThread extends Thread{
}
```

步骤二：实现输出 1 到 100

在 MyThread 类中覆盖 run 方法，在该方法中实现输出 1 到 100，代码如下所示：

```
/**
 * 测试线程
 */
public class TestThread {
    /**
     * 测试多线程并发运行
     */
    public static void main(String[] args) {
    }
}
/**
 * 线程
 */
class MyThread extends Thread{

    public void run(){
```

```
        for(int i=1;i<=100;i++){
            System.out.println(i);
        }
    }
}
```

步骤三：创建并启动线程

在 TestThread 类的 main 方法中，创建两个线程 t1 和 t2，然后使用线程 t1、t2 的 start 方法启动线程，代码如下所示：

```
/**
 * 测试线程
 */
public class TestThread {
    /**
     * 测试多线程并发运行
     */

    public static void main(String[] args) {
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        t1.start();
        t2.start();
    }

}

/**
 * 线程
 */
class MyThread extends Thread{
    public void run(){
        for(int i=1;i<=100;i++){
            System.out.println(i);
        }
    }
}
```

运行 TestThread 类，控制台会输出两次 1 到 100。

• 完整代码

本案例的完整代码如下所示：

```
/**
 * 测试线程
 */
public class TestThread {
    /**
     * 测试多线程并发运行
     */

    public static void main(String[] args) {
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();

        t1.start();
```

```

        t2.start();
    }
}
/**
 * 线程
 */
class MyThread extends Thread{
    public void run(){
        for(int i=1;i<=100;i++){
            System.out.println(i);
        }
    }
}

```

2. 编写一个线程改变窗体的颜色 V1

• 问题

编写一个线程改变窗体的颜色，详细要求如下：

- 1) 使用 Runnable 创建线程，该线程实现窗口的颜色在黑色和白色之间不断的切换。
- 2) 使用内部类创建线程的方式，实现窗口的颜色在黑色和白色之间不断的切换。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建类 TestRunnable

创建 TestRunnable 类，该类继承自 JFrame 类，并实现 Runnable 接口，代码如下所示：

```

import javax.swing.JFrame;
/**
 * 测试 Runnable
 */
public class TestRunnable extends JFrame implements Runnable{
    public static void main(String[] args) {
    }
}

```

步骤二：覆盖 Runnable 接口的 run 方法，实现窗体颜色切换

在 TestRunnable 类中，覆盖 Runnable 接口的 run 方法。在该方法中，首先创建 JPanel 类的对象 panel，并将其放在窗体上；然后，使用 while(true) 循环，在循环中，切换 panel 的颜色从而达到窗体颜色变化，代码如下所示：

```

import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JPanel;
/**
 * 测试 Runnable
 */
public class TestRunnable extends JFrame implements Runnable{

```

```
@Override
public void run() {
    int i = 0;
    JPanel panel = new JPanel();
    panel.setSize(300, 300);
    this.setContentPane(panel);
    while(true){
        i = i==0?1:0;
        if(i==0){
            panel.setBackground(Color.BLACK);
        }else{
            panel.setBackground(Color.WHITE);
        }
    }
}

public static void main(String[] args) {
}
}
```

步骤三：显示窗体、启动线程

在 TestRunnable 类的 main 方法中，首先设置窗体显示，然后启动线程，代码如下所示：

```
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JPanel;
/**
 * 测试 Runnable
 */
public class TestRunnable extends JFrame implements Runnable{
    @Override
    public void run() {
        int i = 0;
        JPanel panel = new JPanel();
        panel.setSize(300, 300);
        this.setContentPane(panel);
        while(true){
            i = i==0?1:0;
            if(i==0){
                panel.setBackground(Color.BLACK);
            }else{
                panel.setBackground(Color.WHITE);
            }
        }
    }
}

public static void main(String[] args) {

    TestRunnable r = new TestRunnable();
    r.setSize(300, 300);
    r.setVisible(true);
    r.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Thread t = new Thread(r);
    t.start();

}
}
```

运行 TestRunnable 类会显示窗体，窗体在黑色和白色之间不断切换。

步骤四：使用内部类的方式创建线程

使用内部类创建线程的方式，实现窗口的颜色在黑色和白色之间不断的切换，代码如下所示：

```
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JPanel;

/**
 * 使用匿名内部类形式创建线程
 * @author Xiloer
 */
public class TestInnerThread {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300,300);
        final JPanel panel = new JPanel();
        panel.setSize(300,300);
        frame.setContentPane(panel);
        frame.setVisible(true);
        Thread t = new Thread(){
            public void run(){
                int i =0;
                while(true){
                    i = i==0?1:0;
                    if(i==0){
                        panel.setBackground(Color.BLACK);
                    }else{
                        panel.setBackground(Color.WHITE);
                    }
                }
            }
        };
        t.start();
    }
}
```

• 完整代码

本案例中，类 TestRunnable 的完整代码如下所示：

```
import java.awt.Color;
import javax.swing.JFrame;
import javax.swing.JPanel;
/**
 * 测试 Runnable
 */
public class TestRunnable extends JFrame implements Runnable{
    @Override
    public void run() {
        int i = 0;
        JPanel panel = new JPanel();
        panel.setSize(300, 300);
        this.setContentPane(panel);
        while(true){
            i = i==0?1:0;
            if(i==0){
                panel.setBackground(Color.BLACK);
            }else{
                panel.setBackground(Color.WHITE);
            }
        }
    }
}
```

```
    }  
    }  
}  
public static void main(String[] args) {  
    TestRunnable r = new TestRunnable();  
    r.setSize(300, 300);  
    r.setVisible(true);  
    r.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    Thread t = new Thread(r);  
    t.start();  
}  
}
```

类 TestInnerThread 的完整代码如下所示：

```
import java.awt.Color;  
import javax.swing.JFrame;  
import javax.swing.JPanel;  
  
/**  
 * 使用匿名内部类形式创建线程  
 * @author Xiloer  
 */  
public class TestInnerThread {  
    public static void main(String[] args) {  
        JFrame frame = new JFrame();  
        frame.setSize(300,300);  
        final JPanel panel = new JPanel();  
        panel.setSize(300,300);  
        frame.setContentPane(panel);  
        frame.setVisible(true);  
        Thread t = new Thread(){  
            public void run(){  
                int i =0;  
                while(true){  
                    i = i==0?1:0;  
                    if(i==0){  
                        panel.setBackground(Color.BLACK);  
                    }else{  
                        panel.setBackground(Color.WHITE);  
                    }  
                }  
            }  
        };  
        t.start();  
    }  
}
```

3. 测试 currentThread 方法

- 问题

测试 currentThread 方法，详细要求如下：

- 1) 新建类 TestCurrentThread 在该类中创建方法 testCurrent 实现输出当前线程。
- 2) 在 TestCurrentThread 类的 main 方法中，输出当前线程并调用 testCurrent 查看当前线程。
- 3) 在 main 方法中，使用内部类创建线程 t，该线程中实现输出当前线程并调用

testCurrent 查看当前线程。

4) 启动线程 t。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 TestCurrentThread 类，添加 testCurrent 方法

新建类 TestCurrentThread，在该类中创建方法 testCurrent，在该方法中使用 Thread 类的静态方法 currentThread 获取当前线程并输出当前线程，代码如下所示：

```
/**
 * 测试当前线程
 */
public class TestCurrentThread {
    public static void main(String[] args) {
    }
}
/**
 * 测试不同线程调用该方法时，获取这个线程
 */
public static void testCurrent(){
    System.out.println("    运    行    testCurrent    方    法    的    线    程
是："+Thread.currentThread());
}
```

步骤二：在 main 方法中获取当前线程

在 TestCurrentThread 类的 main 方法中，输出当前线程并调用 testCurrent 查看当前线程，代码如下所示：

```
/**
 * 测试当前线程
 */
public class TestCurrentThread {
    public static void main(String[] args) {

        System.out.println("运行 main 方法的线程："+Thread.currentThread());
        testCurrent();

    }
}
/**
 * 测试不同线程调用该方法时，获取这个线程
 */
public static void testCurrent(){
    System.out.println("    运    行    testCurrent    方    法    的    线    程
是："+Thread.currentThread());
}
```

运行 TestCurrentThread 方法，控制台输出结果如下：

运行 main 方法的线程:Thread[main,5,main]

运行 testCurrent 方法的线程是:Thread[main,5,main]

输出结果中的“Thread[main,5,main]”表示当前线程的名字为 main、优先级为 5、当前线程的组线程为 main。

步骤三：在内部类中查看当前线程

在 main 方法中，首先，使用内部类创建线程 t，该线程中实现输出当前线程并调用 testCurrent 查看当前线程；然后启动线程 t，代码如下所示：

```
/**
 * 测试当前线程
 */
public class TestCurrentThread {
    public static void main(String[] args) {
        System.out.println("运行main 方法的线程:"+Thread.currentThread());
        testCurrent();

        Thread t = new Thread(){
            @Override
            public void run() {
                System.out.println("线程 t:"+Thread.currentThread());
                testCurrent();
            }
        };
        t.start();
    }
}

/**
 * 测试不同线程调用该方法时，获取这个线程
 */
public static void testCurrent(){
    System.out.println(" 运 行      testCurrent      方 法 的 线 程
是:"+Thread.currentThread());
}
```

运行 TestCurrentThread 类，控制台的输出结果如下：

运行 main 方法的线程:Thread[main,5,main]

运行 testCurrent 方法的线程是:Thread[main,5,main]

线程 t:Thread[Thread-0,5,main]

运行 testCurrent 方法的线程是:Thread[Thread-0,5,main]

输出结果中的“Thread[Thread-0,5,main]”表示当前线程的名字为 Thread-0、优先级为 5、当前线程的组线程为 main。

• 完整代码

本案例的完整代码如下所示：

```
/**
 * 测试当前线程
 */
public class TestCurrentThread {
    public static void main(String[] args) {
        System.out.println("运行 main 方法的线程："+Thread.currentThread());
        testCurrent();
        Thread t = new Thread(){
            @Override
            public void run() {
                System.out.println("线程 t："+Thread.currentThread());
                testCurrent();
            }
        };
        t.start();
    }
}
/**
 * 测试不同线程调用该方法时，获取这个线程
 */
public static void testCurrent(){
    System.out.println("运行 testCurrent 方法的线程是："
        +Thread.currentThread());
}
}
```

4. 测试 getName 方法和 getId 方法

• 问题

测试 Thread 类的 getName 方法和 getId 方法，详细要求如下：

- 1) 创建两个线程，输出默认的线程名字和默认的 ID。
- 2) 创建一个线程，设置线程的名字并输出线程名字和默认的 ID。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：获取默认的线程名字和 ID

首先，新建类 TestThreadNameAndId，在该类的 main 方法中，创建两个线程 t0、t1；接着分别使用 Thread 类的 getName 方法和 getId 方法获取线程的名字和 ID，代码如下所示：

```
/**
 * 获取线程名字及 ID
 */
public class TestThreadNameAndId {
    /**
     * 测试线程的 getName 方法以及 getId 方法
     */
    public static void main(String[] args) {
        Thread t = new Thread();
        System.out.println(t.getName());
    }
}
```

```
System.out.println(t.getId());

Thread t1 = new Thread();
System.out.println(t1.getName());
System.out.println(t1.getId());
}
}
```

运行 TestThreadNameAndId 类，控制台的输出结果如下：

```
Thread-0
9
Thread-1
10
```

从输出结果可以看出，默认的线程名字为“Thread-+数字”的形式；ID 为从数字 9 开始的，这是因为，9 之前的数字被虚拟机的线程占用掉了。

步骤二：为线程添加自定义的名字

在构造 Thread 类的对象时，可以通过 Thread(String)这个构造方法给线程自定义名字，代码如下所示：

```
/**
 * 获取线程名字及 ID
 */
public class TestThreadNameAndId {
    /**
     * 测试线程的 getName 方法以及 getId 方法
     */
    public static void main(String[] args) {
        Thread t = new Thread();
        System.out.println(t.getName());
        System.out.println(t.getId());

        Thread t1 = new Thread();
        System.out.println(t1.getName());
        System.out.println(t1.getId());

        Thread t2 = new Thread("自定义名字的 Thread");
        System.out.println(t2.getName());
        System.out.println(t2.getId());
    }
}
```

运行 TestThreadNameAndId 类，控制台的输出结果如下：

```
Thread-0
9
Thread-1
10
```

```
自定义名字的 Thread
11
```

从输出结果可以看出,当通过构造方法设置自定义名字后,使用 Thread 类的 getName 方法获取名字时得到的则是自定义的名字。

- **完整代码**

本案例的完整代码如下所示：

```
/**
 * 获取线程名字及 ID
 */
public class TestThreadNameAndId {
    /**
     * 测试线程的 getName 方法以及 getId 方法
     */
    public static void main(String[] args) {
        Thread t = new Thread();
        System.out.println(t.getName());
        System.out.println(t.getId());

        Thread t1 = new Thread();
        System.out.println(t1.getName());
        System.out.println(t1.getId());

        Thread t2 = new Thread("自定义名字的 Thread");
        System.out.println(t2.getName());
        System.out.println(t2.getId());
    }
}
```

5. 测试守护线程

- **问题**

测试守护线程,详细要求如下：

- 1) 使用内部类创建线程的方式创建线程 d,该线程实现每隔 0.1 秒输出字符串“后台线程”。
- 2) 设置线程 d 为守护线程并启动该线程。
- 3) 使 main 线程阻塞 5 秒,然后输出字符串“main 线程结束了”。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：创建线程,实现每隔 0.1 秒输出字符串“后台线程”

首先新建类 TestDaemonThread;然后在该类的 main 方法中,使用内部类创建线程的方式创建线程 d;最后,线程 d 实现每隔 0.1 秒输出字符串“后台线程”,代码如下所示：

```
public class TestDaemonThread {
    public static void main(String[] args) {
        Thread d = new Thread(){
            public void run() {
                while(true){
```

```

        System.out.println("后台线程");
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
    }
}
};
}
}
}

```

步骤二：设置 d 线程为后台线程

在 main 方法中，首先，设置 d 线程为后台线程并启动该线程；然后，使用 Thread 类的 sleep 方法使 main 线程阻塞 5 秒；最后，输出字符串 “main 线程结束了”，代码如下所示：

```

public class TestDaemonThread {
    public static void main(String[] args) {
        Thread d = new Thread(){
            public void run() {
                while(true){
                    System.out.println("后台线程");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                    }
                }
            }
        };

        d.setDaemon(true);
        d.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
        }
        //进程中所有前台线程结束后，后台线程强制结束
        System.out.println("main 线程结束了");

    }
}

```

运行 TestDaemonThread 类，控制台会不断输出字符串 “后台线程”，直到输出字符串 “main 线程结束了” 为止。这是因为，d 线程被设置为守护线程，守护线程的特点是，当进程中只剩下守护线程时，所有守护线程强制终止。

• 完整代码

本案例的完整代码如下所示：

```

public class TestDaemonThread {
    public static void main(String[] args) {
        Thread d = new Thread(){
            public void run() {

```

```

        while(true){
            System.out.println("后台线程");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
    }
};
d.setDaemon(true);
d.start();
try {
    Thread.sleep(5000);
} catch (InterruptedException e) {
}
//进程中所有前台线程结束后，后台线程强制结束
System.out.println("main 线程结束了");
}
}

```

6. 编写一个线程改变窗体的颜色 V2

• 问题

在案例“编写一个线程改变窗体的颜色 v1”基础上实现当前案例，即，在使用内部类创建线程的方式，实现窗口的颜色在黑色和白色之间不断的切换的基础上，实现每隔一秒窗体的颜色在黑色和白色之间切换。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建 TestSleep 类

新建 TestSleep 类，使该类中的代码与 TestInnerThread 类中的代码保持一致，代码如下所示：

```

/**
 * 测试 sleep 方法
 */
public class TestSleep {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(300,300);
        final JPanel panel = new JPanel();
        panel.setSize(300,300);
        frame.setContentPane(panel);
        frame.setVisible(true);
        Thread t = new Thread(){
            public void run(){
                int i =0;
                while(true){
                    i = i==0?1:0;
                    if(i==0){
                        panel.setBackground(Color.BLACK);
                    }else{
                        panel.setBackground(Color.WHITE);
                    }
                }
            }
        };
    }
}

```

```
    }  
    }  
};  
t.start();  
}  
}
```

步骤二：实现颜色切换

使用 Thread 类的 sleep 方法实现每隔一秒窗体的颜色在黑色和白色之间切换,代码如下所示：

```
/**  
 * 测试 sleep 方法  
 */  
public class TestSleep {  
    public static void main(String[] args) {  
        JFrame frame = new JFrame();  
        frame.setSize(300,300);  
        final JPanel panel = new JPanel();  
        panel.setSize(300,300);  
        frame.setContentPane(panel);  
        frame.setVisible(true);  
        Thread t = new Thread(){  
            public void run(){  
                int i =0;  
                while(true){  
                    i = i==0?1:0;  
                    if(i==0){  
                        panel.setBackground(Color.BLACK);  
                    }else{  
                        panel.setBackground(Color.WHITE);  
                    }  
  
                    try {  
                        Thread.sleep(1000);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        };  
        t.start();  
    }  
}
```

运行 TestSleep 类，会显示窗体，窗体上的颜色每隔一秒在黑色和白色之间切换。

• 完整代码

本案例中，类 TestSleep 的完整代码如下所示：

```
/**  
 * 测试 sleep 方法  
 */  
public class TestSleep {  
    public static void main(String[] args) {  
        JFrame frame = new JFrame();
```

```

frame.setSize(300,300);
final JPanel panel = new JPanel();
panel.setSize(300,300);
frame.setContentPane(panel);
frame.setVisible(true);
Thread t = new Thread(){
    public void run(){
        int i =0;
        while(true){
            i = i==0?1:0;
            if(i==0){
                panel.setBackground(Color.BLACK);
            }else{
                panel.setBackground(Color.WHITE);
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
};
t.start();
}
}

```

7. 测试 join 方法

• 问题

使用两个线程模拟图片下载的过程，详细要求如下：

- 1) 创建线程 t1 ,该线程模拟实现图片下载的过程 ,即在该线程中实现输出字符串 “t1:正在下载图片:” +下载的百分数 ,例如 :“t1:正在下载图片:30%” ;到 100%之后 ,显示 “t1:图片下载完成”。
- 2) 创建线程 t2 ,在该线程中 ,首先输出 “t2:等待图片下载完毕” ;然后将 t1 线程作为 t2 线程的子线程 ;最后 ,输出 “t2:显示图片”。
- 3) 启动线程 t1 , t2。
- 4) 要求 ,一定是线程 t1 执行完毕之后 ,才会执行线程 t2 中的显示图片。即显示了 “t1:图片下载完成” 之后 ,才会显示 “t2:显示图片”。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建线程 t1

新建类 TestJoin ,该类的 main 方法中创建线程 t1 ,该线程模拟实现图片下载的过程 ,即在该线程中实现输出字符串 “t1:正在下载图片:” +下载的百分数 ,例如 :“t1:已下载图片:30%” ,代码如下所示：

```

public class TestJoin {
    public static void main(String[] args) {
        final Thread t1 = new Thread(){

```



```

        public void run(){
            for(int i=0;i<=10;i++){
                System.out.println("t1:正在下载图片:"+i*10+"%");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("t1:图片下载完成");
        }
    };
}
}

```

步骤二：创建线程 t2

在 main 方法中，创建线程 t2，在该线程中，首先输出“t2:等待图片下载完毕”；然后将 t1 线程作为 t2 线程的子线程；最后，输出“t2:显示图片”，代码如下所示：

```

public class TestJoin {
    public static void main(String[] args) {
        final Thread t1 = new Thread(){
            public void run(){
                for(int i=0;i<=10;i++){
                    System.out.println("t1:正在下载图片:"+i*10+"%");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("t1:图片下载完成");
            }
        };

        Thread t2 = new Thread(){
            public void run(){
                System.out.println("t2:等待图片下载完毕");
                try {
                    t1.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("t2:显示图片");
            }
        };

    }
}

```

步骤三：启动线程

启动线程 t1，t2，代码如下所示：

```

public class TestJoin {
    public static void main(String[] args) {
        final Thread t1 = new Thread(){

```

```

        public void run(){
            for(int i=0;i<=10;i++){
                System.out.println("t1:正在下载图片:"+i*10+"%");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            System.out.println("t1:图片下载完成");
        }
    };
    Thread t2 = new Thread(){
        public void run(){
            System.out.println("t2:等待图片下载完毕");
            try {
                t1.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("t2:显示图片");
        }
    };

    t1.start();
    t2.start();

}
}

```

步骤四：测试

运行 TestJoin 类，控制台会输出 “t2:等待图片下载完毕”，也会输出 “t1:正在下载图片:0%”，以及 “t1:正在下载图片:10%” 等信息。最后，直到输出 “t1:图片下载完成” 后，才会输出 “t2:显示图片”。

这是因为使用了 join 方法，该方法在此用于等待 t1 线程执行结束，再执行 t2 线程。

• 完整代码

本案例中，类 TestJoin 的完整代码如下所示：

```

public class TestJoin {
    public static void main(String[] args) {
        final Thread t1 = new Thread(){
            public void run(){
                for(int i=0;i<=10;i++){
                    System.out.println("t1:正在下载图片:"+i*10+"%");
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("t1:图片下载完成");
            }
        };
        Thread t2 = new Thread(){
            public void run(){

```

```
        System.out.println("t2:等待图片下载完毕");
        try {
            t1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("t2:显示图片");
    }
};
t1.start();
t2.start();
}
}
```

课后作业

1. 线程和进程的区别
2. 简述线程的状态及其转换
3. 简述线程的两种创建方式以及它们的区别
4. 下列关于线程 API 说法正确的是

A .Thread 类的静态方法 `currentThread` 方法可以用于获取运行当前代码片段的线程。
B . Thread 类的 `getName` 方法返回该线程的名称。
C . Thread 类的 `getId` 方法返回该线程的标识符。
D . Thread 类的 `getPriority` 方法获取线程的状态
5. 编写计时线程，每隔 5 秒钟输出当前的日期-时间，主线程结束后计时完毕
6. 下列关于 `sleep` 和 `yield` 说法正确的是

A.sleep 方法是用于阻塞线程，直到被中断为止，`yield` 方法是用于获取 CPU 时间片段。
B.sleep 方法用于让出 CPU 时间片段，`yield` 方法用于使线程阻塞指定毫秒数。
C.sleep 方法用于使线程阻塞指定毫秒数，`yield` 方法用于让出 CPU 时间片段。
D.sleep 方法用于获取 CPU 时间片段，`yield` 用于阻塞线程，直到被中断为止。
7. 已知类 `Foo` 代码如下，下列锁机制不合适的是

`Foo` 类代码如下所示：

```
class Foo {  
    private List list1 = new ArrayList();  
    private List list2 = new ArrayList();  
    public void add 1(Object obj) {  
        list1.add(obj);  
    }  
    public void remove(Object obj) {  
        list1.remove(obj);  
    }  
    public void add 2(Object obj){  
        list2.add(obj);  
    }  
}
```

下列锁机制不合适的是 ()。

A .

```
class Foo {
    private List list1 = new ArrayList();
    private List list2 = new ArrayList();
    public void add_1(Object obj) {
        synchronized(this) {
            list1.add(obj);
        }
    }
    public void remove(Object obj) {
        synchronized(this) {
            list1.remove(obj);
        }
    }
    public void add_2(Object obj){
        synchronized(this) {
            list2.add(obj);
        }
    }
}
```

B.

```
class Foo {
    private List list1 = new ArrayList();
    private List list2 = new ArrayList();
    public void add_1(Object obj) {
        synchronized(list1) {
            list1.add(obj);
        }
    }
    public void remove(Object obj) {
        synchronized(list1) {
            list1.remove(obj);
        }
    }
    public void add_2(Object obj){
        synchronized(list2) {
            list2.add(obj);
        }
    }
}
```

C .

```
class Foo {
    private List list1 = new ArrayList();
    private List list2 = new ArrayList();
    public synchronized void add_1(Object obj) {
        list1.add(obj);
    }
    public synchronized void remove(Object obj) {
        list1.remove(obj);
    }
    public synchronized void add_2(Object obj){
        list2.add(obj);
    }
}
```

D .

```
class Foo {
    private List list1 = new ArrayList();
    private List list2 = new ArrayList();
    public void add_1(Object obj) {
        synchronized(this) {
            list1.add(obj);
        }
    }
    public void remove(Object obj) {
        synchronized(this) {
            list1.remove(obj);
        }
    }
    public void add_2(Object obj){
        synchronized(obj) {
            list2.add(obj);
        }
    }
}
```

Java 核心 API(下)

Unit05

知识体系.....Page 121

多线程基础	线程同步	线程安全 API 与非线程安全 API
		使用 ExecutorService 实现线程池
		使用 BlockingQueue
TCP 通信	Socket 原理	Socket 简介
		获取本地地址和端口号
		获取远端地址和端口号
		获取网络输入流和网络输出流
		close 方法
	Socket 通信模型	Server 端 ServerSocket 监听
		Client 端 Socket 连接
		C-S 端通信模型

经典案例.....Page 127

测试使用 ExecutorService 实现线程池	使用 ExecutorService 实现线程池
测试 BlockingQueue 的使用	使用 BlockingQueue
聊天室案例 V1	C-S 端通信模型
聊天室案例 V2	

课后作业.....Page 153

1. 多线程基础

1.1. 线程同步

1.1.1. 【线程同步】线程安全 API 与非线程安全 API

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h4>线程安全API与非线程安全API</h4> <ul style="list-style-type: none"> • StringBuffer 是同步的 synchronized append(); • StringBuilder 不是同步的 append(); <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 线程安全 </div> <ul style="list-style-type: none"> • Vector 和 Hashtable 是同步的 • ArrayList 和 HashMap 不是同步的 <ul style="list-style-type: none"> • 获取线程安全的集合方式: <ul style="list-style-type: none"> – Collections.synchronizedList() 获取线程安全的List集合 – Collections.synchronizedMap() 获取线程安全的Map <div style="text-align: right;">++</div>
---	---

1.1.2. 【线程同步】使用 ExecutorService 实现线程池

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h4>使用ExecutorService实现线程池</h4> <ul style="list-style-type: none"> • ExecutorService是java提供的用于管理线程池的类。 • 线程池有两个主要作用: <ul style="list-style-type: none"> – 控制线程数量 – 重用线程 • 当一个程序中若创建大量线程，并在任务结束后销毁，会给系统带来过度消耗资源，以及过度切换线程的危险，从而可能导致系统崩溃。为此我们应使用线程池来解决这个问题。 <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 线程安全 </div> <div style="text-align: right;">++</div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h4>使用ExecutorService实现线程池（续1）</h4> <ul style="list-style-type: none"> • 线程池的概念：首先创建一些线程，它们的集合称为线程池，当服务器接受到一个客户请求后，就从线程池中取出一个空闲的线程为之服务，服务完后不关闭该线程，而是将该线程还回到线程池中。 • 在线程池的编程模式下，任务是提交给整个线程池，而不是直接交给某个线程，线程池在拿到任务后，它就在内部找有无空闲的线程，再把任务交给内部某个空闲的线程， • 一个线程同时只能执行一个任务，但可以同时向一个线程池提交多个任务 <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 线程安全 </div> <div style="text-align: right;">++</div>
---	--

代码清单

使用ExecutorService实现线程池(续2)

Tarena 达内科技

- 线程池有以下几种实现策略:
 - Executors.newCachedThreadPool()
创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。
 - Executors.newFixedThreadPool(int nThreads)
创建一个可重用固定线程集合的线程池，以共享的无界队列方式来运行这些线程。
 - Executors.newScheduledThreadPool(int corePoolSize)
创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。
 - Executors.newSingleThreadExecutor()
创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

+

1.1.3. 【线程同步】使用 BlockingQueue

代码清单

使用BlockingQueue

Tarena 达内科技

- BlockingQueue是双缓冲队列。
- 在多线程并发时，若需要使用队列，我们可以使用 Queue，但是要解决一个问题就是同步，但同步操作会降低并发对Queue操作的效率。
- BlockingQueue内部使用两条队列，可允许两个线程同时向队列一个做存储，一个做取出操作。在保证并发的同时提高了队列的存取效率。

+

代码清单



使用BlockingQueue(续1)

Tarena 达内科技

第一个元素（头部）				
	抛出异常	特殊值	阻塞	超时期
插入	addFirst(e)	offerFirst(e)	putFirst(e)	offerFirst(e,time,unit)
移值	removeFirst()	pollFirst()	takeFirst()	pollFirst(time,unit)
检查	getFirst()	peekFirst()	不适用	不适用

最后一个元素（尾部）				
	抛出异常	特殊值	阻塞	超时期
插入	addLast(e)	offerLast(e)	putLast(e)	offerLast(e,time,unit)
移值	removeLast()	pollLast()	takeLast()	pollLast(time,unit)
检查	getLast()	peekLast()	不适用	不适用


+

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>使用BlockingQueue(续2)</h3> <ul style="list-style-type: none"> • ArrayBlockingQueue:规定大小的BlockingDeque,其构造函数必须带一个int参数来指明其大小,其所含的对象是以FIFO(先入先出)顺序排序的. • LinkedBlockingQueue:大小不定的BlockingDeque,若其构造函数带一个规定大小的参数,生成的BlockingDeque有大小限制,若不带大小参数,所生成的BlockingDeque的大小由Integer.MAX_VALUE来决定,其所含的对象是以FIFO(先入先出)顺序排序的 • PriorityBlockingQueue:类似于LinkedBlockDeque,但其所含对象的排序不是FIFO,而是依据对象的自然排序顺序或者是构造函数的Comparator决定的顺序. • SynchronousQueue:特殊的BlockingQueue,对其的操作必须是放和取交替完成的. <div style="text-align: right;">  </div>
---	--



2. TCP 通信

2.1. Socket 原理

2.1.1. 【Socket 原理】Socket 简介

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>Socket简介</h3> <ul style="list-style-type: none"> • socket通常称作“套接字”，用于描述IP地址和端口，是一个通信链的句柄。在Internet上的主机一般运行了多个服务软件，同时提供几种服务。每种服务都打开一个Socket，并绑定到一个端口上，不同的端口对应于不同的服务。 • 应用程序通常通过“套接字”向网络发出请求或者应答网络请求。Socket和ServerSocket类位于java.net包中。ServerSocket用于服务端，Socket是建立网络连接时使用的。在连接成功时，应用程序两端都会产生一个Socket实例，操作这个实例，完成所需的会话。 <div style="text-align: right;">  </div>
---	---

2.1.2. 【Socket 原理】获取本地地址和端口号

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>获取本地地址和端口号</h3> <ul style="list-style-type: none"> • java.net.Socket为套接字类，其提供了很多方法，其中我们可以通过Socket获取本地的地址以及端口号。 <ul style="list-style-type: none"> – int getLocalPort() 该方法用于获取本地使用的端口号 – InetAddress getLocalAddress() 该方法用于获取套接字绑定的本地地址 • 使用InetAddress获取本地的地址方法: <ul style="list-style-type: none"> – String getCanonicalHostName() 获取此 IP 地址的完全限定域名。 – String getHostAddress() 返回 IP 地址字符串（以文本表现形式）。 <div style="text-align: right;">  </div>
---	---

代码封装

获取本地地址和端口号(续1)

Tarena
达内科技

```

public void testSocket()throws Exception {
    Socket socket = new Socket("localhost",8088);
    InetAddress add = socket.getLocalAddress();
    System.out.println(add.getCanonicalHostName());
    System.out.println(add.getHostAddress());
    System.out.println(socket.getLocalPort());
}
                    
```

+

2.1.3. 【Socket 原理】获取远端地址和端口号

代码封装

获取远端地址和端口号

Tarena
达内科技

- 通过Socket获取远端的地址以及端口号。
 - int getPort()
该方法用于获取远端使用的端口号
 - InetAddress .getInetAddress()
该方法用于获取套接字绑定的远端地址

代码封装

获取远端地址和端口号(续1)

Tarena
达内科技


```


public void testSocket()throws Exception {
    Socket socket = new Socket("localhost",8088);
    InetAddress inetAdd = socket.getInetAddress();
    System.out.println(
        inetAdd.getCanonicalHostName());
    System.out.println(inetAdd.getHostAddress());
    System.out.println(socket.getPort());
}
                    
```

+


124

2.1.4. 【Socket 原理】获取网络输入流和网络输出流

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>获取网络输入流和网络输出流</h3> <ul style="list-style-type: none"> 通过Socket获取输入流与输出流,这两个方法是使用Socket通讯的关键方法。 <ul style="list-style-type: none"> InputStream getInputStream() 该方法用于返回此套接字的输入流。 OutputStream.getOutputStream() 该方法用于返回此套接字的输出流。 <div style="text-align: right;">+</div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>获取网络输入流和网络输出流(续1)</h3> <pre> public void testSocket()throws Exception { Socket socket = new Socket("localhost",8088); InputStream in = socket.getInputStream(); OutputStream out = socket.getOutputStream(); } </pre> <div style="text-align: right;">+</div>
---	---

2.1.5. 【Socket 原理】close 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>close方法</h3> <ul style="list-style-type: none"> 当使用Socket进行通讯完毕后,要关闭Socket以释放系统资源。 <ul style="list-style-type: none"> void close() 关闭此套接字。当关闭了该套接字后也会同时关闭由此获取的输入流与输出流。 <div style="text-align: right;">+</div>
---	---

2.2. Socket 通信模型

2.2.1. 【Socket 通信模型】Server 端 ServerSocket 监听

代码清单

Server端ServerSocket监听

Tarena
达内科技

- java.net.ServerSocket是运行于服务端应用程序中。通常创建ServerSocket需要指定服务端口号，之后监听Socket的连接:

```
...
//创建ServerSocket并申请服务端口号8088
ServerSocket server = new ServerSocket(8088);
/*方法会产生阻塞，直到某个Socket连接,并返回请求连接的Socket*/
Socket socket = server.accept();
...
```

Tarena
达内科技

2.2.2. 【Socket 通信模型】Client 端 Socket 连接

代码清单

Client端Socket连接

Tarena
达内科技

- 当服务端创建ServerSocket并通过accept()方法侦听后，我们就可以通过在客户端应用程序中创建Socket来向服务端发起连接。
- 需要注意的是，创建Socket的同时就发起连接，若连接异常会抛出异常。

```
//参数1：服务端的IP地址，参数2:服务端的服务端口号
Socket socket = new Socket( "localhost" ,8088);
...
```

Tarena
达内科技

2.2.3. 【Socket 通信模型】C-S 端通信模型

代码清单

C-S端通信模型

Tarena
达内科技

服务器端
客户端

```

graph TD
    subgraph Server [服务器端]
        A[创建监听服务] --> B[等待连接]
        B --> C[进行通讯]
        C --> D[关闭连接]
    end
    subgraph Client [客户端]
        E[连接服务器] --> F[进行通讯]
        F --> G[关闭连接]
    end
    B -- "建立连接" --> E
    C -- "进行通讯" --> F
            
```

Tarena
达内科技

经典案例

1. 测试使用 ExecutorService 实现线程池

- 问题

使用 ExecutorService 实现线程池，详细要求如下：

- 1) 线程池要执行的任务为每隔一秒输出一次当前线程的名字，总计输出 10 次。
- 2) 创建一个线程池，该线程池中只有两个空线程。
- 3) 使线程池执行 5 次步骤一的任务。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建类

首先，创建名为 TestExecutorService 的类；然后，在该类相同的文件中创建名为 Handler 的类，该类实现了 Runnable 接口，是线程池要执行的任务，代码如下所示：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 测试线程池
 */
public class TestExecutorService {
    public static void main(String[] args) {

    }
}
class Handler implements Runnable{
    public void run(){

    }
}
```

步骤二：覆盖 run 方法

在 Handler 类中覆盖 run 方法，在该方法中实现每隔一秒输出一次当前线程的名字，总计输出 10 次，代码如下所示：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 测试线程池
 */
public class TestExecutorService {
    public static void main(String[] args) {
    }
}
class Handler implements Runnable{
```

```
public void run(){

    String name = Thread.currentThread().getName();
    System.out.println("执行当前任务的线程为:"+name);
    for(int i=0;i<10;i++){
        System.out.println(name+": "+i);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.println(name+":任务完毕");

}
}
```

步骤三：创建线程池

使用 Executors 类的静态方法 newFixedThreadPool，创建一个包含两个空线程的线程池，代码如下所示：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 测试线程池
 */
public class TestExecutorService {
    public static void main(String[] args) {

        ExecutorService threadPool = Executors.newFixedThreadPool(2);

    }
}

class Handler implements Runnable{
    public void run(){
        String name = Thread.currentThread().getName();
        System.out.println("执行当前任务的线程为:"+name);
        for(int i=0;i<10;i++){
            System.out.println(name+": "+i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println(name+":任务完毕");
    }
}
```

步骤四：设置线程池要执行的任务

使用 ExecutorService 类的 execute 方法设置线程池要执行的任务，在此线程池要执行的任务即为 Handler 类中创建的任务；另外，要执行 5 次该任务，那么循环 5 次即可，代码如下所示：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 测试线程池
 */
public class TestExecutorService {
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        for(int i=0;i<5;i++){
            Handler handler = new Handler();
            threadPool.execute(handler);
        }

        class Handler implements Runnable{
            public void run(){
                String name = Thread.currentThread().getName();
                System.out.println("执行当前任务的线程为:"+name);
                for(int i=0;i<10;i++){
                    System.out.println(name+": "+i);
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println(name+":任务完毕");
            }
        }
    }
}
```

运行上述代码，控制台输出的结果如下：

```
执行当前任务的线程为:pool-1-thread-1
执行当前任务的线程为:pool-1-thread-2
pool-1-thread-2:0
pool-1-thread-1:0
pool-1-thread-1:1
pool-1-thread-2:1
pool-1-thread-1:2
pool-1-thread-2:2
pool-1-thread-2:3
pool-1-thread-1:3
pool-1-thread-2:4
pool-1-thread-1:4
pool-1-thread-1:5
pool-1-thread-2:5
pool-1-thread-1:6
pool-1-thread-2:6
pool-1-thread-1:7
pool-1-thread-2:7
pool-1-thread-2:8
pool-1-thread-1:8
pool-1-thread-2:9
pool-1-thread-1:9
pool-1-thread-1:任务完毕
执行当前任务的线程为:pool-1-thread-1
pool-1-thread-1:0
```



```
pool-1-thread-2:任务完毕
执行当前任务的线程为:pool-1-thread-2
pool-1-thread-2:0
pool-1-thread-1:1
pool-1-thread-2:1
pool-1-thread-1:2
pool-1-thread-2:2
pool-1-thread-1:3
pool-1-thread-2:3
pool-1-thread-1:4
pool-1-thread-2:4
pool-1-thread-1:5
pool-1-thread-2:5
pool-1-thread-1:6
pool-1-thread-2:6
pool-1-thread-1:7
pool-1-thread-2:7
pool-1-thread-1:8
pool-1-thread-2:8
pool-1-thread-1:9
pool-1-thread-2:9
pool-1-thread-1:任务完毕
执行当前任务的线程为:pool-1-thread-1
pool-1-thread-1:0
pool-1-thread-2:任务完毕
pool-1-thread-1:1
pool-1-thread-1:2
pool-1-thread-1:3
pool-1-thread-1:4
pool-1-thread-1:5
pool-1-thread-1:6
pool-1-thread-1:7
pool-1-thread-1:8
pool-1-thread-1:9
pool-1-thread-1:任务完毕
```

从输出结果可以看出，线程池每次启动两个线程来执行任务。由于要求执行 5 次任务，所以线程池分三次执行，前两次各执行 2 个任务，最后一次执行一个任务。

• 完整代码

本案例中，类 TestExecutorService 的完整代码如下所示：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 测试线程池
 */
public class TestExecutorService {
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(2);
        for(int i=0;i<5;i++){
            Handler handler = new Handler();
            threadPool.execute(handler);
        }
    }
}
class Handler implements Runnable{
    public void run(){
```

```
String name = Thread.currentThread().getName();
System.out.println("执行当前任务的线程为:"+name);
for(int i=0;i<10;i++){
    System.out.println(name+": "+i);
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
System.out.println(name+":任务完毕");
}
}
```

2. 测试 BlockingQueue 的使用

• 问题

测试 BlockingQueue 的使用，详细要求如下：

1) 首先，使用 ArrayBlockingQueue 类创建一个大小为 10 的双缓冲队列 queue；然后，循环 20 次向队列 queue 中添加元素，如果 5 秒内元素仍没有入队到队列中，则返回 false。

2) 首先，使用 ArrayBlockingQueue 类创建一个大小为 10 的双缓冲队列 queue；然后，将 0 到 9，10 个数字加入到队列 queue 中；最后，循环 20 次从队列 queue 中取元素，如果 5 秒内还没有元素可取出则返回 null。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建双缓冲队列

首先，创建 TestBlockingQueue 类；然后在该类中添加测试方法 testOffer；最后，使用 ArrayBlockingQueue 类创建一个大小为 10 的双缓冲队列 queue，代码如下所示：

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import org.junit.Test;

/**
 * 测试双缓冲队列
 */
public class TestBlockingQueue {
    /**
     * 测试入队方法
     */
    @Test
    public void testOffer() {
        BlockingQueue <Integer> queue
            = new ArrayBlockingQueue<Integer>(10);
    }
}
```

步骤二：测试 offer 方法

使用双缓冲队列 BlockingQueue 的 offer 方法向队列 queue 中添加元素，代码如下所示：

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import org.junit.Test;

/**
 * 测试双缓冲队列
 */
public class TestBlockingQueue {
    /**
     * 测试入队方法
     */
    @Test
    public void testOffer() {
        BlockingQueue<Integer> queue
            = new ArrayBlockingQueue<Integer>(10);

        for(int i=0;i<20;i++){
            try {
                //设置 5 秒超时，5 秒内元素仍没有入队到队列中，则返回 false
                boolean b = queue.offer(i,5,TimeUnit.SECONDS);
                System.out.println("存入是否成功:"+b);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

上述代码中的 offer 方法的三个参数分别表示入队元素、超时时长以及超时的时间单位。

运行上述代码，控制台输出结果如下所示：

```
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:true
存入是否成功:false
存入是否成功:false
存入是否成功:false
```

```
存入是否成功:false
存入是否成功:false
存入是否成功:false
存入是否成功:false
存入是否成功:false
存入是否成功:false
存入是否成功:false
```

从输出结果可以看出，输出了 10 次“存入是否成功:true”、10 次“存入是否成功:false”，这是因为双缓冲队列的大小为 10，入队 10 个元素后，则不能再有元素入队了。

步骤三：创建测试方法 testPull

首先在 TestBlockingQueue 类中添加测试方法 testPull；然后，使用 ArrayBlockingQueue 类创建一个大小为 10 的双缓冲队列 queue，代码如下所示：

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import org.junit.Test;

/**
 * 测试双缓冲队列
 */
public class TestBlockingQueue {
    /**
     * 测试入队方法
     */
    @Test
    public void testOffer() {
        BlockingQueue<Integer> queue
            = new ArrayBlockingQueue<Integer>(10);
        for(int i=0;i<20;i++){
            try {
                //设置 5 秒超时，5 秒内元素仍没有入队到队列中，则返回 false
                boolean b = queue.offer(i,5,TimeUnit.SECONDS);
                System.out.println("存入是否成功:"+b);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    /**
     * 测试出队方法
     */

    @Test
    public void testPull() {
        BlockingQueue<Integer> queue
            = new ArrayBlockingQueue<Integer>(10);
    }
}
```

步骤四：测试 poll 方法

首先,使 0 到 9,10 个数字加入到队列 queue 中,然后,使用双缓冲队列 BlockingQueue 的 poll 方法使 queue 中的元素出队,如果 5 秒内还没有元素可取出则返回 null,代码如下所示:

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import org.junit.Test;

/**
 * 测试双缓冲队列
 */
public class TestBlockingQueue {
    /**
     * 测试入队方法
     */
    @Test
    public void testOffer() {
        BlockingQueue<Integer> queue
            = new ArrayBlockingQueue<Integer>(10);
        for(int i=0;i<20;i++){
            try {
                //设置 5 秒超时, 5 秒内元素仍没有入队到队列中, 则返回 false
                boolean b = queue.offer(i,5,TimeUnit.SECONDS);
                System.out.println("存入是否成功:"+b);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    /**
     * 测试出队方法
     */
    @Test
    public void testPull() {
        BlockingQueue<Integer> queue
            = new ArrayBlockingQueue<Integer>(10);

        for(int i=0;i<10;i++){
            queue.offer(i);
        }
        for(int i =0;i<20;i++){
            //获取元素, 设置 5 秒超时, 5 秒内还没有元素可取出则返回 null
            try {
                Integer num = queue.poll(5, TimeUnit.SECONDS);
                System.out.println("元素:"+num);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

运行上述代码, 控制台输出的结果如下所示:

元素:0

```
元素:1
元素:2
元素:3
元素:4
元素:5
元素:6
元素:7
元素:8
元素:9
元素:null
元素:null
元素:null
元素:null
元素:null
元素:null
元素:null
元素:null
元素:null
元素:null
元素:null
```

从输出结果可以看出，输出了 10 次 “元素:+数字”、10 次 “元素:null”，这是因为双缓冲队列中只有 10 个元素，全部取出后，再取则返回 null。

• 完整代码

本案例中，类 TestBlockingQueue 的完整代码如下所示：

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import org.junit.Test;

/**
 * 测试双缓冲队列
 */
public class TestBlockingQueue {
    /**
     * 测试入队方法
     */
    @Test
    public void testOffer() {
        BlockingQueue<Integer> queue
            = new ArrayBlockingQueue<Integer>(10);
        for(int i=0;i<20;i++){
            try {
                //设置 5 秒超时，5 秒内元素仍没有入队，则返回 false
                boolean b = queue.offer(i,5,TimeUnit.SECONDS);
                System.out.println("存入是否成功:"+b);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
/**
 * 测试出队方法
 */
@Test
public void testPull() {
    BlockingQueue<Integer> queue
        = new ArrayBlockingQueue<Integer>(10);
    for(int i=0;i<10;i++){
        queue.offer(i);
    }
    for(int i =0;i<20;i++){
        //获取元素, 设置5 秒超时, 5 秒内还没有元素可取出则返回 null
        try {
            Integer num = queue.poll(5, TimeUnit.SECONDS);
            System.out.println("元素:"+num);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

3. 聊天室案例 V1

• 问题

使用 Java 的 Socket 实现客户端和服务端之间的连接, 并使客户端向服务端发送一条消息。

通信过程如表-1 所示:

表-1 客户端与服务端通信过程

客户端	服务器
1.客户端发送“你好, 服务器!”	2.接收客户端信息, 并显示

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建客户端类

新建名为 com.tarena.part1 的包, 并在包下新建名为 Client 的类, 用于表示客户端, 代码如下所示:

```
package com.tarena.part1;

/**
 * 客户端应用程序
 * 第一步: 实现向服务器发送一条信息
 */
public class Client {
}
```

步骤二：创建 Socket 类的对象

在 Client 类中声明全局变量 socket 表示一个客户端 Socket 对象，并在实例化 Client 类时使用构造方法“Socket (String ip,int port)”来创建 Socket 类的对象。此时，需要进行异常处理。代码如下所示：

```
package com.tarena.part1;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

/**
 * 客户端应用程序
 * 第一步:实现向服务器发送一条信息
 */
public class Client {
    //客户端 Socket

    private Socket socket;

    /**
     * 构造方法，用于初始化
     */

    public Client(){
        try {
            socket = new Socket("localhost",8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

步骤三：创建客户端工作方法 start()

创建客户端工作方法 start()，并添加代码实现连接服务器端并发送信息。

首先使用 Socket 类的 getOutputStream 方法获取对应 Socket 对象的网络字节输出流对象；然后，为了写出数据，构造缓冲字符输出流 PrintWriter 类的对象，使用该对象的 println 方法向服务器发送数据。

注意，这里需要进行异常处理，并在 finally 语句中，关闭 Socket 对象。

Client 类的代码如下所示：

```
package com.tarena.part1;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

/**
 * 客户端应用程序
```



```
* 第一步:实现向服务器发送一条信息
*/
public class Client {
    //客户端 Socket
    private Socket socket;
    /**
     * 构造方法,用于初始化
     */
    public Client(){
        try {
            socket = new Socket("localhost",8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 客户端工作方法
     */

    public void start(){
        try {
            OutputStream out = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");
            PrintWriter pw = new PrintWriter(osw,true);
            pw.println("你好!服务器");
        } catch (Exception e) {
            e.printStackTrace();
        } finally{
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

步骤四：为客户端类定义 main() 方法

为类 Client 定义 main() 方法,并在 main() 方法中,创建 Client 对象,调用上一步中所创建的 start() 方法。代码如下所示：

```
package com.tarena.part1;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

/**
 * 客户端应用程序
 * 第一步:实现向服务器发送一条信息
 */
public class Client {
    //客户端 Socket
```

```
private Socket socket;
/**
 * 构造方法，用于初始化
 */
public Client(){
    try {
        socket = new Socket("localhost",8088);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
/**
 * 客户端工作方法
 */
public void start(){
    try {
        OutputStream out = socket.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");
        PrintWriter pw = new PrintWriter(osw,true);
        pw.println("你好！服务器");
    } catch (Exception e) {
        e.printStackTrace();
    } finally{
        if(socket != null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

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

步骤五：创建服务器端类

新建名为 Server 的类，用于表示服务器端，代码如下所示：

```
package com.tarena.part1;

/**
 * 服务端应用程序
 */
public class Server {
}
```

步骤六：创建 ServerSocket 类的对象

在 Server 类中声明全局变量 serverSocket 表示一个服务器端 ServerSocket 对象，并在实例化 Server 类时使用构造方法 "ServerSocket(int port)" 来创建对象。此时，需要进行异常处理。代码如下所示：

```
package com.tarena.part1;

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

/**
 * 服务端应用程序
 */
public class Server {
    //服务端 Socket

    private ServerSocket serverSocket;

    /**
     * 构造方法，用于初始化
     */

    public Server(){
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

步骤七：创建服务器端工作方法 start()

创建服务器端工作方法 start()，用于读取客户端发来的信息。

首先监听客户端的连接，得到 Socket 对象，并使用 Socket 类的 getInputStream 方法获取对应 Socket 对象的网络字节输入流对象；然后，为了读取数据，构造缓冲字符输入流 BufferedReader 类的对象，并调用该对象的 readLine 方法获取客户端发来的数据。

注意，这里需要进行异常处理，并在 finally 语句中，关闭 Socket 对象。

Server 类的代码如下所示：

```
package com.tarena.part1;

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

/**
 * 服务端应用程序
 */
public class Server {
    //服务端 Socket
    private ServerSocket serverSocket;
    /**
     * 构造方法，用于初始化
     */
}
```

```
public Server(){
    try {
        serverSocket = new ServerSocket(8088);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
/**
 * 服务端开启方法
 */

public void start(){
    try {
        System.out.println("等待客户端连接...");
        //监听客户端的连接
        Socket socket = serverSocket.accept();
        System.out.println("客户端已连接!");
        InputStream in = socket.getInputStream();
        InputStreamReader isr = new InputStreamReader(in,"UTF-8");
        BufferedReader br = new BufferedReader(isr);
        System.out.println("客户端说:"+br.readLine());
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

步骤八：为服务器端类定义 main 方法

为类 Server 定义 main 方法，并在 main 方法中，创建 Server 对象，调用上一步中所创建的 start 方法。代码如下所示：

```
package com.tarena.part1;

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

/**
 * 服务端应用程序
 */
public class Server {
    //服务端 Socket
    private ServerSocket serverSocket;
    /**
     * 构造方法，用于初始化
     */
    public Server(){
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 服务端开启方法
     */
}
```

```
*/
public void start(){
    try {
        System.out.println("等待客户端连接...");
        //监听客户端的连接
        Socket socket = serverSocket.accept();
        System.out.println("客户端已连接!");
        InputStream in = socket.getInputStream();
        InputStreamReader isr = new InputStreamReader(in, "UTF-8");
        BufferedReader br = new BufferedReader(isr);
        System.out.println("客户端说:"+br.readLine());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    Server server = new Server();
    server.start();
}
}
```

步骤九：测试

首先，启动服务器端；然后再启动客户端。

服务器端控制台运行输出如下：

```
等待客户端连接...
客户端已连接!
客户端说:你好! 服务器
```

• 完整代码

本案例中，Client 类的完整代码如下所示：

```
package com.tarena.part1;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;

/**
 * 客户端应用程序
 * 第一步:实现向服务器发送一条信息
 */
public class Client {
    //客户端 Socket
    private Socket socket;
    /**
     * 构造方法，用于初始化
     */
    public Client(){
        try {
```

```

        socket = new Socket("localhost",8088);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
/**
 * 客户端工作方法
 */
public void start(){
    try {
        OutputStream out = socket.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");
        PrintWriter pw = new PrintWriter(osw,true);
        pw.println("你好！服务器");
    } catch (Exception e) {
        e.printStackTrace();
    } finally{
        if(socket != null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

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

```

Server 类的完整代码如下所示：

```

package com.tarena.part1;

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

/**
 * 服务端应用程序
 */
public class Server {
    //服务端 Socket
    private ServerSocket serverSocket;
    /**
     * 构造方法，用于初始化
     */
    public Server(){
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 服务端开启方法
     */
    public void start(){
        try {
            System.out.println("等待客户端连接...");

```

```
//监听客户端的连接
Socket socket = serverSocket.accept();
System.out.println("客户端已连接!");
InputStream in = socket.getInputStream();
InputStreamReader isr = new InputStreamReader(in, "UTF-8");
BufferedReader br = new BufferedReader(isr);
System.out.println("客户端说:"+br.readLine());
} catch (Exception e) {
    e.printStackTrace();
}
}

public static void main(String[] args) {
    Server server = new Server();
    server.start();
}
}
```

4. 聊天室案例 V2

- 问题

改善聊天室案例 V1，实现客户端重复发送数据到服务器端的功能。即，用户可以在控制台不断输入内容，并将内容逐一发送给服务端。

- 方案

此案例在上一个案例的基础上修改部分功能即可。

首先，对于客户端而言，为了能够重复发送信息，需要构建循环，并在循环中，不断读入控制台录入的数据并发送。代码如下所示：

```
//创建 Scanner 读取用户输入内容
Scanner scanner = new Scanner(System.in);
while(true){
    //pw 为 PrintWriter 对象
    pw.println(scanner.nextLine());
}
```

其次，对于服务器端，也需要构建循环，并在循环中不断读取客户端发来的数据并打印显示。代码如下所示：

```
//循环读取客户端发送的信息
while(true){
    //br 为 BufferedReader 对象
    System.out.println("客户端说:"+br.readLine());
}
```

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建客户端类

新建名为 `com.tarena.part2` 的包，并在包下新建名为 `Client` 的类，用于表示客户端。在 `Client` 类中声明全局变量 `socket` 表示一个客户端 `Socket` 对象，并在实例化 `Client` 类时创建 `Socket` 类的对象。代码如下所示：

```
package com.tarena.part2;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
 * 客户端应用程序
 * 第二步:实现向服务器循环发送信息
 */
public class Client {
    //客户端 Socket
    private Socket socket;
    /**
     * 构造方法，用于初始化
     */
    public Client(){
        try {
            socket = new Socket("localhost",8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

步骤二：创建客户端工作方法 `start()`

创建客户端工作方法 `start()`，并添加代码实现连接服务器端并发送信息。为了能够重复发送信息，需要构建循环，并在循环中，不断读入控制台录入的数据并发送。

`Client` 类的代码如下所示：

```
package com.tarena.part2;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
 * 客户端应用程序
 * 第二步:实现向服务器循环发送信息
 */
public class Client {
    //客户端 Socket
    private Socket socket;
    /**
     * 构造方法，用于初始化
     */
}
```



```

public Client(){
    try {
        socket = new Socket("localhost",8088);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
/**
 * 客户端工作方法
 */

public void start(){
    try {
        OutputStream out = socket.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");
        PrintWriter pw = new PrintWriter(osw,true);

        //创建 Scanner 读取用户输入内容
        Scanner scanner = new Scanner(System.in);
        while(true){
            pw.println(scanner.nextLine());
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally{
        if(socket != null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}
}

```

步骤三：为客户端类定义 main() 方法

为类 Client 定义 main() 方法，并在 main() 方法中，创建 Client 对象，调用上一步中所创建的 start() 方法。代码如下所示：

```

package com.tarena.part2;

import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
 * 客户端应用程序
 * 第二步:实现向服务器循环发送信息
 */
public class Client {
    //客户端 Socket
    private Socket socket;
    /**
     * 构造方法，用于初始化

```

```

    */
    public Client(){
        try {
            socket = new Socket("localhost",8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 客户端工作方法
     */
    public void start(){
        try {
            OutputStream out = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");
            PrintWriter pw = new PrintWriter(osw,true);

            //创建 Scanner 读取用户输入内容
            Scanner scanner = new Scanner(System.in);
            while(true){
                pw.println(scanner.nextLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally{
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

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

}

```

步骤四：创建服务器端类

新建名为 `Server` 的类，用于表示服务器端，并在 `Server` 类中声明全局变量 `serverSocket` 表示一个服务器端 `ServerSocket` 对象，并在实例化 `Server` 类时创建该对象。此时，需要进行异常处理。代码如下所示：

```

package com.tarena.part2;

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

/**
 * 服务端应用程序
 */
public class Server {

```

```
//服务端 Socket
private ServerSocket serverSocket;
/**
 * 构造方法，用于初始化
 */
public Server(){
    try {
        serverSocket = new ServerSocket(8088);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

步骤五：创建服务器端工作方法 start()

创建服务器端工作方法 start()，用于读取客户端发来的信息。需要构建循环，并在循环中不断读取客户端发来的数据并打印显示。

Server 类的代码如下所示：

```
package com.tarena.part2;

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

/**
 * 服务端应用程序
 */
public class Server {
    //服务端 Socket
    private ServerSocket serverSocket;
    /**
     * 构造方法，用于初始化
     */
    public Server(){
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 服务端开启方法
     */
    public void start(){

        try {
            System.out.println("等待客户端连接...");
            //监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in,"UTF-8");
            BufferedReader br = new BufferedReader(isr);
```

```
//循环读取客户端发送的信息
while(true){
    System.out.println("客户端说:"+br.readLine());
}
} catch (Exception e) {
    e.printStackTrace();
}
}
}
```

步骤六：为服务器端类定义 main 方法

为类 Server 定义 main 方法，并在 main 方法中，创建 Server 对象，调用上一步中所创建的 start 方法。代码如下所示：

```
package com.tarena.part2;

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

/**
 * 服务端应用程序
 */
public class Server {
    //服务端 Socket
    private ServerSocket serverSocket;
    /**
     * 构造方法，用于初始化
     */
    public Server(){
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 服务端开启方法
     */
    public void start(){
        try {
            System.out.println("等待客户端连接...");
            //监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);

            //循环读取客户端发送的信息
            while(true){
                System.out.println("客户端说:"+br.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
}  
  
public static void main(String[] args) {  
    Server server = new Server();  
    server.start();  
}
```

步骤七：测试

首先，启动服务器端。此时，服务器端将等待客户端的连接和数据发送。服务器端控制台输出如图 - 1 所示：

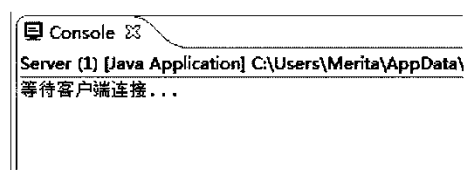


图 - 1

然后再启动客户端，并在客户端重复输入文本后回车。界面效果如图 - 2 所示：

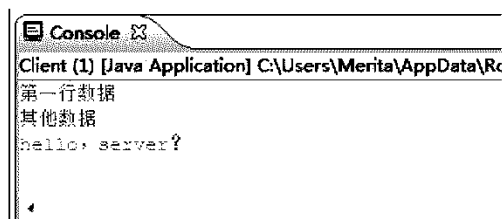


图 - 2

查看服务器端的控制台输出，界面效果如图 - 3 所示：



图 - 3

• 完整代码

本案例中，Client 类的完整代码如下所示：

```
package com.tarena.part2;
```

```
import java.io.IOException;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

/**
 * 客户端应用程序
 * 第二步:实现向服务器循环发送信息
 */
public class Client {
    //客户端 Socket
    private Socket socket;
    /**
     * 构造方法,用于初始化
     */
    public Client(){
        try {
            socket = new Socket("localhost",8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 客户端工作方法
     */
    public void start(){
        try {
            OutputStream out = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");
            PrintWriter pw = new PrintWriter(osw,true);

            //创建 Scanner 读取用户输入内容
            Scanner scanner = new Scanner(System.in);
            while(true){
                pw.println(scanner.nextLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally{
            if(socket != null){
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

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

Server 类的完整代码如下所示：

```
package com.tarena.part2;

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

```
import java.net.Socket;

/**
 * 服务端应用程序
 */
public class Server {
    //服务端 Socket
    private ServerSocket serverSocket;
    /**
     * 构造方法，用于初始化
     */
    public Server(){
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    /**
     * 服务端开启方法
     */
    public void start(){
        try {
            System.out.println("等待客户端连接...");
            //监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);

            //循环读取客户端发送的信息
            while(true){
                System.out.println("客户端说:"+br.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }
}
```

课后作业

1. 下列属于线程安全类的是：
 - A.StringBuffer
 - B.StringBuilder
 - C.ArrayList
 - D.HashMap

2. 使用 BlockingQueue 实现生产者和消费者模型

Java 核心 API(下)

Unit06

知识体系.....Page 155

TCP 通信	Socket 通信模型	Server 端多线程模型
UDP 通信	DatagramPacket	构建接收包
		构建发送包
	DatagramSocket	服务端接收
		客户端发送

经典案例.....Page 158


聊天室案例 V3	Server 端多线程模型
聊天室案例 V4	
聊天室案例 V5	
UDP 通信模型	构建接收包
	构建发送包
	服务端接收
	客户端发送

课后作业.....Page 192

1. TCP 通信


1.1. Socket 通信模型


1.1.1. 【Socket 通信模型】Server 端多线程模型



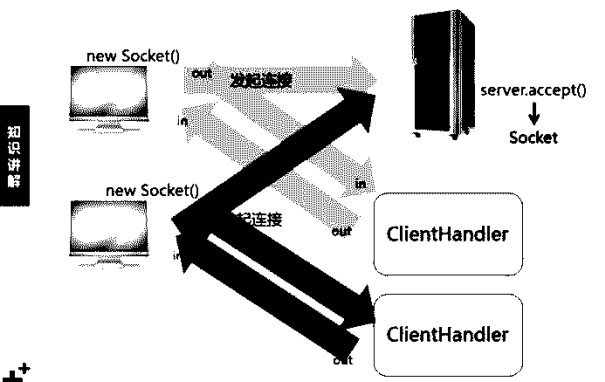
Server端多线程模型


- 若想使一个服务端可以支持多客户端连接，我们需要解决以下问题：
 - 循环调用accept方法侦听客户端的连接
 - 使用线程来处理单一客户端的数据交互
- 因为需要处理多客户端，所以服务端要周期性循环调用accept方法，但该方法会产生阻塞，所以与某个客户端的交互就需要使用线程来并发处理。





Server端多线程模型（续1）






2. UDP 通信

2.1. DatagramPacket

2.1.1. 【DatagramPacket】构建接收包

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">构建接收包</h4> <ul style="list-style-type: none"> • DatagramPacket : UDP数据报基于IP建立的,每台主机有65536个端口号可以使用。数据报中字节数限制为65536-8 • 构造接收包: <ul style="list-style-type: none"> - DatagramPacket(byte[] buf, int length) 将数据包中length长的数据装进buf数组。 - DatagramPacket(byte[] buf, int offset, int length) 将数据包中从offset开始、length长的数据装进buf数组。 <div style="text-align: right;">+</div>
---	---

2.1.2. 【DatagramPacket】构建发送包

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">构建发送包</h4> <ul style="list-style-type: none"> • 构造发送包: <ul style="list-style-type: none"> - DatagramPacket(byte[] buf, int length, InetAddress clientAddress, int clientPort) 从buf数组中,取出length长的数据创建数据包对象,目标是clientAddress地址,clientPort端口,通常用来发送数据给客户端。 - DatagramPacket(byte[] buf, int offset, int length, InetAddress clientAddress, int clientPort) 从buf数组中,取出offset开始的、length长的数据创建数据包对象,目标是clientAddress地址,clientPort端口,通常用来发送数据给客户端。 <div style="text-align: right;">+</div>
---	---

2.2. DatagramSocket

2.2.1. 【DatagramSocket】服务端接收

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">服务端接收</h4> <ul style="list-style-type: none"> • DatagramSocket 用于接收和发送UDP的Socket实例 <ul style="list-style-type: none"> - DatagramSocket(int port) 创建实例,并固定监听port端口的报文。通常用于服务端 • receive(DatagramPacket d) 接收数据报文到d中。receive方法产生“阻塞”。 <div style="text-align: right;">+</div>
---	---

2.2.2. 【DatagramSocket】客户端发送

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div data-bbox="751 275 900 313">客户端发送</div> <div data-bbox="1214 271 1326 313">Tarena 达内科技</div> <div data-bbox="751 327 1289 622"><ul style="list-style-type: none">• DatagramSocket 用于接收和发送UDP的Socket实例<ul style="list-style-type: none">– 无参的构造方法DatagramSocket() 通常用于客户端编程，它并没有特定监听的端口，仅仅使用一个临时的。程序会让操作系统分配一个可用的端口。• send(DatagramPacket dp) 该方法用于发送报文dp到目的地。</div> <div data-bbox="724 427 756 524">知识讲解</div> <div data-bbox="724 680 756 714">+</div>
---	--

经典案例

1. 聊天室案例 V3

• 问题

重构聊天室案例，使用线程来实现一个服务器端可以同时接收多个客户端的信息。通信过程如表 - 1 所示：

表-1 客户端与服务器端通信过程

客户端	服务器
	启动服务器等待客户端连接
客户端 A 连接服务器，并发送信息给服务器。	服务器接收客户端发送过来的信息，并输出信息显示
客户端 B 连接服务器，并发送信息给服务器	
客户端 C 连接服务器，并发送信息给服务器	

客户端 A，B，C . . . 可以同时去连接服务器，和服务器进行通信。

• 方案

之前的聊天室案例中，已经实现了客户端和服务端一对一的通信。现在需要实现多个客户端连接同一个服务器，则需要服务器端循环等待多个客户端发送的请求。

为实现此功能，首先需要在服务器端定义内部类，并在该内部类中设置线程要执行的任务。此案例中，线程要执行的任务即为接收对应的客户端的消息并打印显示。该内部类的代码如下：

```
/**
 * 线程体，用于并发处理不同客户端的交互
 */
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;
    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);

            // 循环读取客户端发送的信息
            while (true) {
                System.out.println("客户端说:" + br.readLine());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (socket != null) {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}
}
}

```

其次，在服务器端 `Server` 类中的 `start` 方法中，需要使用 `while (true)` 循环，在循环中阻塞等待多个客户端的连接。当有客户端连接时，则使用上文定义的任务作为线程的任务，并启动一个该任务对应的线程，来处理服务器和该客户端之间的通信，代码如下：

```

public void start() {
    try {
        //循环监听客户端的连接
        while(true){
            System.out.println("等待客户端连接...");
            // 监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");

            //启动一个线程来完成针对该客户端的交互
            ClientHandler handler = new ClientHandler(socket);
            new Thread(handler).start();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

而客户端的实现与 V2 版本完全相同。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：新建类 `Server`

创建 `Server` 类，在该类的构造方法中，创建 `ServerSocket` 类的对象。在实例化对象时使用构造方法 “`ServerSocket (int port)`” 来构造 `ServerSocket` 类的对象，以申请服务的端口号，代码如下所示：

```

package com.tarena.part3;

import java.net.ServerSocket;
import java.net.Socket;

/**
 * 服务端应用程序

```

```
*/
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;

    /**
     * 构造方法，用于初始化
     */
    public Server() {
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

步骤二：定义服务器线程要执行的任务

在 Server 类中定义成员内部类 ClientHandler。该内部类需要实现 Runnable 接口并实现该接口的 run 方法。在该方法中实现线程要执行的任务，在此，线程要执行的任务即为接收对应的客户端的消息和向对应的客户端发送消息，代码如下所示：

```
package com.tarena.part3;

import java.net.ServerSocket;
import java.net.Socket;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

/**
 * 服务端应用程序
 */
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;

    /**
     * 构造方法，用于初始化
     */
    public Server() {
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 线程体，用于并发处理不同客户端的交互
     */

    private class ClientHandler implements Runnable {
        // 该线程用于处理的客户端
        private Socket socket;
        public ClientHandler(Socket socket) {
```

}

步骤三：启动线程

一个该任务对应的线程，来处理服务器和该客户端之间的通信，代码如下所示：


```
/**
 * 服务端开启方法
 */

public void start() {
    try {
        //循环监听客户端的连接
        while(true){
            System.out.println("等待客户端连接...");
            // 监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");

            //启动一个线程来完成针对该客户端的交互
            ClientHandler handler = new ClientHandler(socket);
            new Thread(handler).start();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * 线程体，用于并发处理不同客户端的交互
 */
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;
    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);

            // 循环读取客户端发送的信息
            while (true) {
                System.out.println("客户端说:" + br.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

步骤四：编写启动服务器的代码

在 Server 类的 main 方法中，编写启动服务器端的代码。代码如下所示：

```
package com.tarena.part3;

import java.net.ServerSocket;
import java.net.Socket;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
/**
 * 服务端应用程序
 */
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;

    /**
     * 构造方法，用于初始化
     */
    public Server() {
        try {
            serverSocket = new ServerSocket(8088);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开启方法
     */
    public void start() {
        try {
            //循环监听客户端的连接
            while(true){
                System.out.println("等待客户端连接...");
                // 监听客户端的连接
                Socket socket = serverSocket.accept();
                System.out.println("客户端已连接!");

                //启动一个线程来完成针对该客户端的交互
                ClientHandler handler = new ClientHandler(socket);
                new Thread(handler).start();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }

    /**
     * 线程体，用于并发处理不同客户端的交互
     */
}
```

```

    */
    private class ClientHandler implements Runnable {
        // 该线程用于处理的客户端
        private Socket socket;
        public ClientHandler(Socket socket) {
            this.socket = socket;
        }

        @Override
        public void run() {
            try {
                InputStream in = socket.getInputStream();
                InputStreamReader isr = new InputStreamReader(in, "UTF-8");
                BufferedReader br = new BufferedReader(isr);

                // 循环读取客户端发送的信息
                while (true) {
                    System.out.println("客户端说:" + br.readLine());
                }
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (socket != null) {
                    try {
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}

```

步骤五：编写客户端的代码

新建 Client 类，并在该类中编写实现客户端需要的代码。Client 类的代码和上一个案例中的代码相同，详见完整代码。

• 完整代码

本案例中，类 Server 的完整代码如下所示：

```

package com.tarena.part3;

import java.net.ServerSocket;
import java.net.Socket;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
/**
 * 服务端应用程序
 */
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;

    /**
     * 构造方法，用于初始化
     */
}

```

```

public Server() {
    try {
        serverSocket = new ServerSocket(8088);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * 服务端开启方法
 */
public void start() {
    try {
        //循环监听客户端的连接
        while(true){
            System.out.println("等待客户端连接...");
            // 监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");

            //启动一个线程来完成针对该客户端的交互
            ClientHandler handler = new ClientHandler(socket);
            new Thread(handler).start();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    Server server = new Server();
    server.start();
}

/**
 * 线程体，用于并发处理不同客户端的交互
 */
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);
            // 循环读取客户端发送的信息
            while (true) {
                System.out.println("客户端说:" + br.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

```
    }  
}  
}
```

类 Client 的完整代码如下所示：

```
package com.tarena.part3;  
  
import java.io.IOException;  
import java.io.OutputStream;  
import java.io.OutputStreamWriter;  
import java.io.PrintWriter;  
import java.net.Socket;  
import java.util.Scanner;  
  
/**  
 * 客户端应用程序  
 */  
public class Client {  
    //客户端 Socket  
    private Socket socket;  
  
    /**  
     * 构造方法，用于初始化  
     */  
    public Client(){  
        try {  
            socket = new Socket("localhost",8088);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    /**  
     * 客户端工作方法  
     */  
    public void start(){  
        try {  
            OutputStream out = socket.getOutputStream();  
            OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");  
            PrintWriter pw = new PrintWriter(osw,true);  
  
            //创建 Scanner 读取用户输入内容  
            Scanner scanner = new Scanner(System.in);  
            while(true){  
                pw.println(scanner.nextLine());  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally{  
            if(socket != null){  
                try {  
                    socket.close();  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        Client client = new Client();  
        client.start();  
    }  
}
```

2. 聊天室案例 V4

• 问题

重构聊天室案例,使服务端可以将用户的信息转发给所有客户端,并在每个客户端控制台上显示。通信过程如表 - 2 所示:

表- 2 客户端与服务端通信过程

客户端	服务端
	启动服务器等待客户端连接
客户端 A 连接服务器,并发送信息给服务器。	服务器接收客户端发送过来的信息,并转发给所有客户端
客户端 B 连接服务器,并发送信息给服务器。	
客户端 C 连接服务器,并发送信息给服务器。	
客户端 A 收到服务器发送的信息并输出到控制台	
客户端 B 收到服务器发送的信息并输出到控制台	
客户端 C 收到服务器发送的信息并输出到控制台	

• 方案

之前的聊天室案例中,已经实现了多个客户端可以连接同一个服务器端的通信。现在需要实现服务端对某个客户端发送的信息进行广播(转发给所有客户端)的工作,并且使客户端在接收到服务端转发的信息后输出到控制台。

为实现此功能,首先需要在服务器端定义一个集合类型的属性,用于存储所有客户端的输出流。代码如下:

```
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;
    // 所有客户端输出流
    private List<PrintWriter> allOut;
}
```

然后在 Server 的内部类中 run 方法的最开始处将客户端的输出流存入该集合。之后,每当客户端发送信息后就遍历集合,将信息写入集合中所有的输出流中(相当于将信息转发给所有的客户端)。代码如下:

```
public void run() {
    PrintWriter pw = null;
    try {
        //将客户端的输出流存入共享集合,以便广播消息
        OutputStream out = socket.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");
        pw = new PrintWriter(osw,true);
        /*
         * 将用户信息存入共享集合
         */
        allOut.add(pw);
    }
```

随后将客户端发送过来的信息转发给所有的客户端，代码如下：

```
String message = null;
// 循环读取客户端发送的信息
while ((message = br.readLine())!=null) {
    /*
     * 遍历所有输出流，将该客户端发送的信息转发给所有客户端
     */
    for(PrintWriter o : allOut){
        o.println(message);
    }
}
```

之后在客户端中定义一个内部类，并定义线程要执行的任务，这里循环读取服务端发送过来的信息，并打印到控制台。代码如下：

```
private class ServerHandler implements Runnable{
    @Override
    public void run() {
        try {
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);
            while(true){
                System.out.println(br.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

最后，在 Client 的 start 方法中启动一个该任务对应的线程，来处理服务器和该客户端之间的通信。代码如下：

```
//将接收服务端信息的线程启动
ServerHandler handler = new ServerHandler();
Thread t = new Thread(handler);
t.setDaemon(true);
t.start();
```

• 步骤

步骤一：定义 Server 类

定义 Server 类，并在类中定义属性 allOut，该属性使用 PrintWriter 作为集合的泛型，用于存储输出流。在 Server 类的构造方法中，创建 java.util.ArrayList 类的实例来初始化该属性。代码如下所示：

```
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;
```

```
// 所有客户端输出流
```

```
private List<PrintWriter> allOut;
```

```
/**
```

```
 * 构造方法，用于初始化
```

```
 */
```

```
public Server() {
```

```
    try {
```

```
        serverSocket = new ServerSocket(8088);
```

```
        allOut = new ArrayList<PrintWriter>();
```

```
    } catch (Exception e) {
```

```
        e.printStackTrace();
```

```
    }
```

```
}
```

```
}
```

步骤二：定义 Server 的内部类 ClientHandler

为类 Server 定义内部类 ClientHandler，并向集合添加客户端的输出流。

在内部类的 run 方法中，通过该线程处理的客户端的 Socket 获取向该客户端发送信息的输出流，并将该输出流包装为 PrintWriter 后存入集合 allOut 中；当客户端断线，需要将输出流从共享集合中删除。代码如下所示：

```
private class ClientHandler implements Runnable {
```

```
    // 该线程用于处理的客户端
```

```
    private Socket socket;
```

```
    public ClientHandler(Socket socket) {
```

```
        this.socket = socket;
```

```
    }
```

```
    @Override
```

```
    public void run() {
```

```
        PrintWriter pw = null;
```

```
        try {
```

```
            //将客户端的输出流存入共享集合，以便广播消息
```

```
            OutputStream out = socket.getOutputStream();
```

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

```
            pw = new PrintWriter(osw, true);
```

```
            /*
```

```
             * 将用户信息存入共享集合
```

```
            */
```

```
            allOut.add(pw);
```

```
        } catch (Exception e) {
```

```
            e.printStackTrace();
```

```
        } finally {
```

```
            /*
```

```
             * 当客户端断线，要将输出流从共享集合中删除
```

```
            */
```



```
allOut.remove(pw);

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

步骤三：修改 run 方法，将客户端发送的信息转发给所有客户端

继续为 run 方法添加代码：在获取到客户端发送过来的信息后，遍历集合 allOut，将获取到的信息写入该集合中每一个输出流中，从而将该信息发送给所有客户端。代码如下所示：

```
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        PrintWriter pw = null;
        try {
            //将客户端的输出流存入共享集合，以便广播消息
            OutputStream out = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");
            pw = new PrintWriter(osw, true);
            /*
             * 将用户信息存入共享集合
             */
            allOut.add(pw);

            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);

            String message = null;
            // 循环读取客户端发送的信息
            while ((message = br.readLine()) != null) {
                /*
                 * 遍历所有输出流，将该客户端发送的信息转发给所有客户端
                 */
                for (PrintWriter o : allOut) {
                    o.println(message);
                }
            }

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            /*
```

```

        * 当客户端断线，要将输出流从共享集合中删除
        */
        allOut.remove(pw);
        if (socket != null) {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}
}

```

步骤四：定义 Server 类的 start 和 main 方法

为 Server 类定义 start 方法和 main 方法，这两个方法的代码和之前的案例相同，不再赘述。

步骤五：定义客户端线程要执行的任务

在 Client 类中定义成员内部类 ServerHandler。该内部类需要实现 Runnable 接口并实现该接口的 run 方法。在该方法中实现线程要执行的任务，在此，线程要执行的任务为循环接收服务端的消息并打印到控制台。代码如下所示：

```

private class ServerHandler implements Runnable{
    @Override
    public void run() {
        try {
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);
            while(true){
                System.out.println(br.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}

```

步骤六：修改 Client 类的 start 方法，创建并启动线程

修改 Client 类中的 start 方法，在该方法中，创建并启动线程。代码如下所示：

```

public void start(){
    try {
        //将接收服务端信息的线程启动
        ServerHandler handler = new ServerHandler();
        Thread t = new Thread(handler);
        t.setDaemon(true);
        t.start();

        OutputStream out = socket.getOutputStream();
        OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");
        PrintWriter pw = new PrintWriter(osw, true);

        //创建 Scanner 读取用户输入内容
        Scanner scanner = new Scanner(System.in);
        while(true){
            pw.println(scanner.nextLine());
        }
    }
}

```

```

    }
    } catch (Exception e) {
        e.printStackTrace();
    } finally{
        if(socket != null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
}

```

• 完整代码

本案例中，类 Server 的完整代码如下所示：

```

package com.tarena.part4;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;

/**
 * 服务端应用程序
 */
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;
    // 所有客户端输出流
    private List<PrintWriter> allOut;
    /**
     * 构造方法，用于初始化
     */
    public Server() {
        try {
            serverSocket = new ServerSocket(8088);
            allOut = new ArrayList<PrintWriter>();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 服务端开启方法
     */
    public void start() {
        try {
            //循环监听客户端的连接
            while(true){
                System.out.println("等待客户端连接...");
                // 监听客户端的连接
            }
        }
    }
}

```

```

        Socket socket = serverSocket.accept();
        System.out.println("客户端已连接!");

        //启动一个线程来完成针对该客户端的交互
        ClientHandler handler = new ClientHandler(socket);
        new Thread(handler).start();
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

public static void main(String[] args) {
    Server server = new Server();
    server.start();
}

/**
 * 线程体，用于并发处理不同客户端的交互
 */
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        PrintWriter pw = null;
        try {
            //将客户端的输出流存入共享集合，以便广播消息
            OutputStream out = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");
            pw = new PrintWriter(osw, true);
            /*
             * 将用户信息存入共享集合
             */
            allOut.add(pw);

            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);

            String message = null;
            // 循环读取客户端发送的信息
            while ((message = br.readLine()) != null) {
                /*
                 * 遍历所有输出流，将该客户端发送的信息转发给所有客户端
                 */
                for(PrintWriter o : allOut){
                    o.println(message);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            /*
             * 当客户端断线，要将输出流从共享集合中删除
             */
            allOut.remove(pw);
            if (socket != null) {

```



```

    }
    } catch (Exception e) {
        e.printStackTrace();
    } finally{
        if(socket != null){
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[] args) {
    Client client = new Client();
    client.start();
}
/**
 * 该线程用于接收服务端发送过来的信息
 */
private class ServerHandler implements Runnable{
    @Override
    public void run() {
        try {
            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);
            while(true){
                System.out.println(br.readLine());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
}

```

3. 聊天室案例 V5

• 问题

现有的聊天室功能虽然已经完成,但是由于客户端的频繁连接与断开,会使得服务端频繁的创建及销毁线程。随着客户端的增加,服务端的线程也在增加,这无疑会对服务端的资源造成浪费,并且由于过多的线程导致的过度切换也会为服务端带来崩溃的风险。与此同时,多个线程会共享服务端的集合属性 `allOut`, 这里还存在着多线程并发的安全问题。

为此,需要重构聊天室案例,使用线程池技术来解决服务端多线程问题,并解决多线程并发的安全问题。

• 方案

之前的聊天室案例中,已经实现了聊天室的基本功能,现在需要对程序进行优化,使程序更加健壮。因此,需要使用线程池来控制客户端连接后启动和管理线程,并解决由于多线程共享 `Server` 属性 `allOut` 所引起的并发安全问题。

为实现此功能,首先需要在服务器端定义一个线程池类型的属性,用于管理服务端的线

程创建及管理。代码如下所示：

```
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;
    // 所有客户端输出流
    private List<PrintWriter> allOut;
    // 线程池
    private ExecutorService threadPool;
    /**
     * 构造方法，用于初始化
     */
    public Server() {
        try {
            serverSocket = new ServerSocket(8088);

            allOut = new ArrayList<PrintWriter>();

            threadPool = Executors.newFixedThreadPool(40);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

其次我们修改 Server 的 start 方法 将原来创建并启动线程的代码替换为使用线程池管理的方式。代码如下所示：

```
/**
 * 服务端开启方法
 */
public void start() {
    try {
        //循环监听客户端的连接
        while(true){
            System.out.println("等待客户端连接...");
            // 监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");

            //启动一个线程来完成针对该客户端的交互
            ClientHandler handler = new ClientHandler(socket);
            threadPool.execute(handler);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

然后在 Server 中添加三个方法，用于操作属性 allOut，并使用同步锁，使三个方法变为同步的。代码如下所示：

```
/**
 * 将输出流存入共享集合，与下面两个方法互斥，保证同步安全
 * @param out
```

```

    */
    private synchronized void addOut(PrintWriter out){
        allOut.add(out);
    }
    /**
     * 将给定输出流从共享集合删除
     * @param out
     */
    private synchronized void removeOut(PrintWriter out){
        allOut.remove(out);
    }
    /**
     * 将消息转发给所有客户端
     * @param message
     */
    private synchronized void sendMessage(String message){
        for(PrintWriter o : allOut){
            o.println(message);
        }
    }
}

```

最后，将原来操作向集合中添加，删除元素。遍历集合并将信息写入每一个输出流的操作改造为调用三个方法，以确保同步安全。代码如下所示：

```

/*
 * 将用户信息存入共享集合
 * 需要同步
 */
addOut(pw);
...

// 循环读取客户端发送的信息
while ((message = br.readLine())!=null) {
    /*
     * 遍历所有输出流，将该客户端发送的信息转发给所有客户端
     * 需要同步
     */
    sendMessage(message);
}
...
/*
 * 当客户端断线，要将输出流从共享集合中删除
 * 需要同步
 */
removeOut(pw);
...

```

• 步骤

步骤一：定义 Server 类

定义 Server 类，并在 Server 类中添加 ExecutorService 类型的属性 threadPool，并在构造方法中将其初始化。初始化时，使用固定大小的线程池，线程数量为 40。这里使用 Executors 类的 newFixedThreadPool(int threads)方法来创建固定大小的线程池。代码如下所示：


```
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;
    // 所有客户端输出流
    private List<PrintWriter> allOut;
    // 线程池
    private ExecutorService threadPool;
    /**
     * 构造方法，用于初始化
     */
    public Server() {
        try {
            serverSocket = new ServerSocket(8088);
            allOut = new ArrayList<PrintWriter>();

            threadPool = Executors.newFixedThreadPool(40);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

步骤二：为 Server 类创建 start 方法

为 Server 类创建 start 方法。在该方法中，将原代码中创建并启动线程的方式，改为使用线程池来完成。创建内部类实例后，使用 ExecutorService 的 execute(Runnable runn)方法来启动线程。代码如下所示：

```
/**
 * 服务端开启方法
 */
public void start() {
    try {
        //循环监听客户端的连接
        while(true){
            System.out.println("等待客户端连接...");
            // 监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");

            //启动一个线程来完成针对该客户端的交互
            ClientHandler handler = new ClientHandler(socket);
            threadPool.execute(handler);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

步骤三：为 Server 类定义 addOut 方法

定义 addOut 方法，该方法向 Server 的属性 allOut 集合中添加输出流，并使用 synchronized 关键字修饰，使该方法变为同步方法。代码如下所示：

```
/**
 * 将输出流存入共享集合，与下面两个方法互斥，保证同步安全
```

```
* @param out
*/
private synchronized void addOut(PrintWriter out){
    allOut.add(out);
}
```

步骤四：为 Server 类定义 removeOut 方法

定义 removeOut 方法，该方法从 Server 的属性 allOut 集合中删除输出流，并使用 synchronized 关键字修饰，使该方法变为同步方法。代码如下所示：

```
/**
 * 将给定输出流从共享集合删除
 * @param out
 */
private synchronized void removeOut(PrintWriter out){
    allOut.remove(out);
}
```

步骤五：为 Server 类定义 sendMessage 方法

定义 sendMessage 方法，该方法用于遍历 Server 的属性 allOut 集合元素，将信息写入每一个输出流，并使用 synchronized 关键字修饰，使该方法变为同步方法。代码如下所示：

```
/**
 * 将消息转发给所有客户端
 * @param message
 */
private synchronized void sendMessage(String message){
    for(PrintWriter o : allOut){
        o.println(message);
    }
}
```

步骤六：创建内部类

创建 Server 的内部类 ClientHandler，在内部类中定义 run 方法。在 run 方法中，调用前面步骤中所创建的 addOut 方法。代码如下所示：

```
/**
 * 线程体，用于并发处理不同客户端的交互
 */
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        PrintWriter pw = null;
        try {
            //将客户端的输出流存入共享集合，以便广播消息
            OutputStream out = socket.getOutputStream();
```

```

        OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");
        pw = new PrintWriter(osw, true);
        /*
         * 将用户信息存入共享集合
         * 需要同步
         */
        addOut(pw);
    }
}

```

步骤七：继续为 run 方法添加代码

继续为 run 方法添加代码，调用 sendMessage 方法，实现将信息转发给所有客户端。

代码如下所示：

```

/**
 * 线程体，用于并发处理不同客户端的交互
 */
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        PrintWriter pw = null;
        try {
            // 将客户端的输出流存入共享集合，以便广播消息
            OutputStream out = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");
            pw = new PrintWriter(osw, true);
            /*
             * 将用户信息存入共享集合
             * 需要同步
             */
            addOut(pw);

            InputStream in = socket.getInputStream();
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");
            BufferedReader br = new BufferedReader(isr);

            String message = null;
            // 循环读取客户端发送的信息
            while ((message = br.readLine()) != null) {
                /*
                 * 遍历所有输出流，将该客户端发送的信息转发给所有客户端
                 * 需要同步
                 */
                sendMessage(message);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```
}  
}
```

步骤八：删除输出流

继续为 run 方法添加代码，调用 removeOut 方法从集合 allOut 中删除输出流。代码如下所示：

```
/**  
 * 线程体，用于并发处理不同客户端的交互  
 */  
private class ClientHandler implements Runnable {  
    // 该线程用于处理的客户端  
    private Socket socket;  
  
    public ClientHandler(Socket socket) {  
        this.socket = socket;  
    }  
  
    @Override  
    public void run() {  
        PrintWriter pw = null;  
        try {  
            //将客户端的输出流存入共享集合，以便广播消息  
            OutputStream out = socket.getOutputStream();  
            OutputStreamWriter osw = new OutputStreamWriter(out, "UTF-8");  
            pw = new PrintWriter(osw, true);  
            /*  
             * 将用户信息存入共享集合  
             * 需要同步  
             */  
            addOut(pw);  
  
            InputStream in = socket.getInputStream();  
            InputStreamReader isr = new InputStreamReader(in, "UTF-8");  
            BufferedReader br = new BufferedReader(isr);  
  
            String message = null;  
            // 循环读取客户端发送的信息  
            while ((message = br.readLine()) != null) {  
                /*  
                 * 遍历所有输出流，将该客户端发送的信息转发给所有客户端  
                 * 需要同步  
                 */  
                sendMessage(message);  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    } finally {  
        /*  
         * 当客户端断线，要将输出流从共享集合中删除  
         * 需要同步  
         */  
        removeOut(pw);  
  
        if (socket != null) {  
            try {  
                socket.close();  
            }  
        }  
    }  
}
```

```

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

```

步骤九：定义 Server 类的 main 方法

为 Server 类定义 main 方法，该方法的代码和之前的案例相同，不再赘述。

步骤十：定义 Client 类

Client 类的代码和之前的案例相同，参照之前代码即可

• 完整代码

本案例中，类 Server 的完整代码如下所示：

```

package com.tarena.part5;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

/**
 * 服务端应用程序
 */
public class Server {
    // 服务端 Socket
    private ServerSocket serverSocket;
    // 所有客户端输出流
    private List<PrintWriter> allOut;
    // 线程池
    private ExecutorService threadPool;
    /**
     * 构造方法，用于初始化
     */
    public Server() {
        try {
            serverSocket = new ServerSocket(8088);

            allOut = new ArrayList<PrintWriter>();

            threadPool = Executors.newFixedThreadPool(40);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

* 服务端开启方法
*/
public void start() {
    try {
        //循环监听客户端的连接
        while(true){
            System.out.println("等待客户端连接...");
            // 监听客户端的连接
            Socket socket = serverSocket.accept();
            System.out.println("客户端已连接!");

            //启动一个线程来完成针对该客户端的交互
            ClientHandler handler = new ClientHandler(socket);
            threadPool.execute(handler);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
/**
 * 将输出流存入共享集合，与下面两个方法互斥，保证同步安全
 * @param out
 */
private synchronized void addOut(PrintWriter out){
    allOut.add(out);
}
/**
 * 将给定输出流从共享集合删除
 * @param out
 */
private synchronized void removeOut(PrintWriter out){
    allOut.remove(out);
}
/**
 * 将消息转发给所有客户端
 * @param message
 */
private synchronized void sendMessage(String message){
    for(PrintWriter o : allOut){
        o.println(message);
    }
}

public static void main(String[] args) {
    Server server = new Server();
    server.start();
}

/**
 * 线程体，用于并发处理不同客户端的交互
 */
private class ClientHandler implements Runnable {
    // 该线程用于处理的客户端
    private Socket socket;

    public ClientHandler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        PrintWriter pw = null;
        try {
            //将客户端的输出流存入共享集合，以便广播消息
            OutputStream out = socket.getOutputStream();
            OutputStreamWriter osw = new OutputStreamWriter(out,"UTF-8");

```

```

        pw = new PrintWriter(osp,true);
        /*
         * 将用户信息存入共享集合
         * 需要同步
         */
        addOut(pw);

        InputStream in = socket.getInputStream();
        InputStreamReader isr = new InputStreamReader(in, "UTF-8");
        BufferedReader br = new BufferedReader(isr);

        String message = null;
        // 循环读取客户端发送的信息
        while ((message = br.readLine())!=null) {
            /*
             * 遍历所有输出流，将该客户端发送的信息转发给所有客户端
             * 需要同步
             */
            sendMessage(message);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        /*
         * 当客户端断线，要将输出流从共享集合中删除
         * 需要同步
         */
        removeOut(pw);

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

```

本案例中，类 Client 没有修改，参照之前代码即可。

4. UDP 通信模型

• 问题

使用 Java 的 DatagramSocket 实现客户端和服务端通信。通信过程如表-3 所示：

表-3 客户端与服务端通信过程

客户端	服务器
1.客户端发送 "Hello! I'm Client"	2.接收客户端信息
4.客户端接收信息	3.服务器发送 "Hello ! I'm Server"

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建客户端类和服务器端类

首先，创建类 Client 表示客户端；然后，创建类 Server 表示服务器端，Client 类代码如下所示：

```
public class Client {
    public static void main(String[] args) {
    }
}
```

Server 类代码如下所示：

```
public class Server{
    public static void main(String[] args) {
    }
}
```

步骤二：客户端向服务器发送数据，服务器端接收该数据

要实现客户端向服务器发送数据 “Hello! I'm Client”，详细过程如下：

1) 创建 Socket 实例。在 Client 类中新建 start 方法，在该方法中，创建用于接收和发送 UDP 的 DatagramSocket 类的实例。客户端使用无参数的构造方法构造 DatagramSocket 类的实例即可。

2) 构建发送包。使用如下构造方法来构造数据包 DatagramPacket 类的对象：

```
DatagramPacket(byte[] buf, int length, InetAddress clientAddress, int clientPort)
```

上述构造方法表示从 buf 数组中，取出 length 长度的数据创建数据包对象，该数据包对象的目标地址是 clientAddress、端口是 clientPort。本案例的目标 IP 地址是 127.0.0.1，即本机、目标端口是 8088。

3) 发送数据。使用 DatagramSocket 类提供的 send 方法，向目标地址和端口发送报文。

Client 类代码如下所示：

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Client {
    private void start() {
        try {

            DatagramSocket client = new DatagramSocket();
            String sendStr = "Hello! I'm Client";
            byte[] sendBuf;
            sendBuf = sendStr.getBytes();
            InetAddress addr = InetAddress.getByName("127.0.0.1");
            int port = 8088;
            DatagramPacket sendPacket = new DatagramPacket(sendBuf,
                sendBuf.length, addr, port);
```



```

        client.send(sendPacket);

    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {

}
}

```

要实现服务器接收客户端发送的数据，详细过程如下：

1) 创建 Socket 实例。在 Server 类中新建 start 方法，在该方法中，首先创建用于接收和发送 UDP 的 DatagramSocket 类的实例。服务器端使用构造方法 “DatagramSocket(int port)” 来构造 DatagramSocket 类的实例。该构造方法可以固定监听 port 端口的报文。

2) 构建接收包。使用如下构造方法来构造数据包 DatagramPacket 类的对象：

```
DatagramPacket(byte[] buf, int length)
```

上述构造方法表示将数据包中 length 长度的数据装进 buf 数组。

3) 接收数据。使用 DatagramSocket 类的 receive 方法接收数据报文。

Server 类代码如下所示：

```

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Server {

    public void start() {
        try {

            DatagramSocket server = new DatagramSocket(8088);
            byte[] recvBuf = new byte[100];
            DatagramPacket recvPacket = new DatagramPacket(recvBuf, recvBuf.length);
            server.receive(recvPacket);
            String recvStr = new String(recvPacket.getData(), 0,
                recvPacket.getLength());
            System.out.println("客户端说:" + recvStr);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {

    }
}

```

步骤三：服务器向客户端发送数据，客户端接收数据

要实现服务器向客户端发送数据 “Hello ! I'm Server”，详细过程如下：

1) 构建发送包。从上一步客户端发送的数据包中获取客户端的地址和端口号，使用如下构造方法来构造数据包 DatagramPacket 类的对象：

```
DatagramPacket(byte[] buf, int length, InetAddress clientAddress, int clientPort)
```

上述构造方法表示从 buf 数组中，取出 length 长度的数据创建数据包对象，该数据包对象的目标地址是 clientAddress、端口是 clientPort。

2) 发送数据。使用 DatagramSocket 类提供的 send 方法，向目标地址和端口发送报文。

Server 类代码如下所示：

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Server {

    public void start() {
        try {
            DatagramSocket server = new DatagramSocket(8088);
            byte[] recvBuf = new byte[100];
            DatagramPacket recvPacket = new DatagramPacket(recvBuf,
            recvBuf.length);
            server.receive(recvPacket);
            String recvStr = new String(recvPacket.getData(), 0,
            recvPacket.getLength());
            System.out.println("客户端说:" + recvStr);

            int port = recvPacket.getPort();
            InetAddress addr = recvPacket.getAddress();
            String sendStr = "Hello ! I'm Server";
            byte[] sendBuf;
            sendBuf = sendStr.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendBuf,
            sendBuf.length, addr, port);
            server.send(sendPacket);
            server.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }
}
```

客户端接收数据的过程和服务端接收数据的过程类似，在这里不再赘述。

Client 类代码如下所示：

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
```

```
public class Client {
    private void start() {
        try {
            DatagramSocket client = new DatagramSocket();
            String sendStr = "Hello! I'm Client";
            byte[] sendBuf;
            sendBuf = sendStr.getBytes();
            InetAddress addr = InetAddress.getByName("127.0.0.1");
            int port = 8088;
            DatagramPacket sendPacket = new DatagramPacket(sendBuf,
                sendBuf.length, addr, port);

            client.send(sendPacket);

            byte[] recvBuf = new byte[100];
            DatagramPacket recvPacket = new DatagramPacket(recvBuf, recvBuf.length);
            client.receive(recvPacket);
            String recvStr = new String(recvPacket.getData(), 0,
                recvPacket.getLength());
            System.out.println("服务端说:" + recvStr);

            client.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

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

步骤四：添加客户端和服务端启动的代码

在 Client 类的 main 方法中调用该类的 start 方法、在 Server 类的 main 方法中调用该类 start 方法。

Client 类代码如下所示：

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Client {
    private void start() {
        try {
            DatagramSocket client = new DatagramSocket();
            String sendStr = "Hello! I'm Client";
            byte[] sendBuf;
            sendBuf = sendStr.getBytes();
            InetAddress addr = InetAddress.getByName("127.0.0.1");
            int port = 8088;
            DatagramPacket sendPacket = new DatagramPacket(sendBuf,
                sendBuf.length, addr, port);

            client.send(sendPacket);

            byte[] recvBuf = new byte[100];
            DatagramPacket recvPacket = new DatagramPacket(recvBuf,
```

```

recvBuf.length);
    client.receive(recvPacket);
    String recvStr = new String(recvPacket.getData(), 0,
        recvPacket.getLength());
    System.out.println("服务端说:" + recvStr);

    client.close();
} catch (Exception e) {
    e.printStackTrace();
}
}

public static void main(String[] args) {

    Client client = new Client();
    client.start();

}
}

```

Server 类代码如下所示：

```

import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Server {

    public void start() {
        try {
            DatagramSocket server = new DatagramSocket(8088);
            byte[] recvBuf = new byte[100];
            DatagramPacket recvPacket = new DatagramPacket(recvBuf,
recvBuf.length);
            server.receive(recvPacket);
            String recvStr = new String(recvPacket.getData(), 0,
                recvPacket.getLength());
            System.out.println("客户端说:" + recvStr);

            int port = recvPacket.getPort();
            InetAddress addr = recvPacket.getAddress();
            String sendStr = "Hello ! I'm Server";
            byte[] sendBuf;
            sendBuf = sendStr.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendBuf,
                sendBuf.length, addr, port);
            server.send(sendPacket);
            server.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {

        Server server = new Server();
        server.start();

    }
}

```

步骤五：测试

测试客户端和服务端是否通信成功。首先，启动服务器端；然后再启动客户端。

客户端控制台运行输出如下：

服务端说:Hello ! I'm Server

服务器端控制台运行输出如下：

客户端说:Hello! I'm Client

- **完整代码**

本案例中，Client 类的完整代码如下所示：

```
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Client {
    private void start() {
        try {
            DatagramSocket client = new DatagramSocket();
            String sendStr = "Hello! I'm Client";
            byte[] sendBuf;
            sendBuf = sendStr.getBytes();
            InetAddress addr = InetAddress.getByName("127.0.0.1");
            int port = 8088;
            DatagramPacket sendPacket = new DatagramPacket(sendBuf,
                sendBuf.length, addr, port);

            client.send(sendPacket);

            byte[] recvBuf = new byte[100];
            DatagramPacket recvPacket = new DatagramPacket(recvBuf,
                recvBuf.length);
            client.receive(recvPacket);
            String recvStr = new String(recvPacket.getData(), 0,
                recvPacket.getLength());
            System.out.println("服务端说:" + recvStr);

            client.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {

        Client client = new Client();
        client.start();

    }
}
```

Server 类的完整代码如下所示：

```
import java.net.DatagramPacket;
```

```
import java.net.DatagramSocket;
import java.net.InetAddress;

public class Server {

    public void start() {
        try {
            DatagramSocket server = new DatagramSocket(8088);
            byte[] recvBuf = new byte[100];
            DatagramPacket recvPacket = new DatagramPacket(recvBuf,
                recvBuf.length);
            server.receive(recvPacket);
            String recvStr = new String(recvPacket.getData(), 0,
                recvPacket.getLength());
            System.out.println("客户端说:" + recvStr);

            int port = recvPacket.getPort();
            InetAddress addr = recvPacket.getAddress();
            String sendStr = "Hello ! I'm Server";
            byte[] sendBuf;
            sendBuf = sendStr.getBytes();
            DatagramPacket sendPacket = new DatagramPacket(sendBuf,
                sendBuf.length, addr, port);
            server.send(sendPacket);
            server.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        Server server = new Server();
        server.start();
    }
}
```

课后作业

1. 完成聊天室的私聊功能

完成聊天室私聊功能。私聊功能是指，客户端之间可以实现一对一的聊天。
服务器端程序启动后，将等待客户端连接，界面效果如图 - 1 所示：

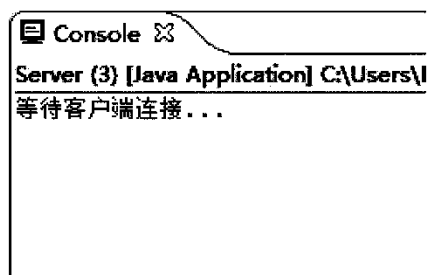


图 - 1

客户端程序运行时，需要用户先输入昵称。用户输入昵称之后，提示用户可以开始聊天。
界面效果如图 - 2 所示：



图 - 2

另一个客户端运行起来后，也需要输入昵称，界面效果如图 - 3 所示：

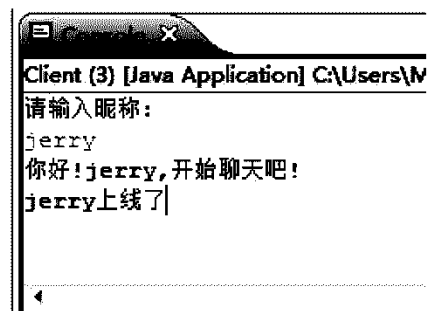


图 - 3

此时，其他运行中的客户端会收到昵称为“jerry”的客户端上线的消息。比如，之前运行起来的客户端“mary”的界面效果如图 - 4 所示：

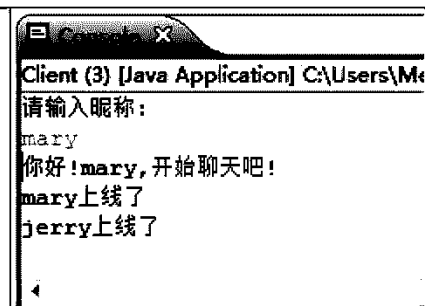


图 - 4

其他客户端可以通过输入类似“\jerry:你好”这样的字样和昵称为“jerry”的客户端私聊。比如，昵称为“mary”的客户端可以输入如图 - 5 所示的信息：

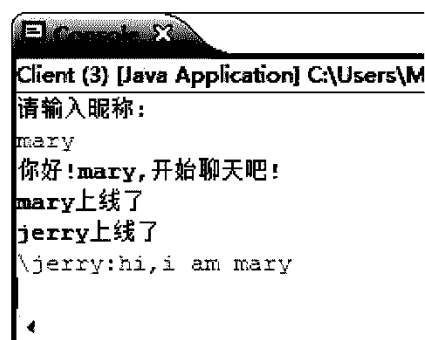


图 - 5

注意：如果需要进行私聊，必需使用“\昵称:信息”的格式发送消息。其中，“\昵称:”为固定格式，“昵称”表示要私聊的客户端的昵称；“信息”表示需要发送的消息。例如：“\jerry:你好”，表示发送消息“你好”给昵称为“jerry”的客户端。

昵称为“jerry”的客户端将接收到客户端“mary”发来的信息，界面效果如图 - 6 所示：

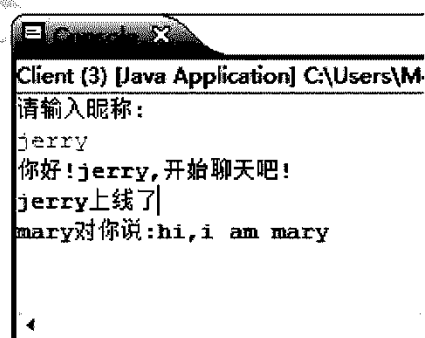


图 - 6

如果某客户端程序停止运行，其他客户端程序可以接收到消息并显示。例如，昵称为“jerry”的客户端停止运行，昵称为“mary”的客户端的界面效果如图 - 7 所示：

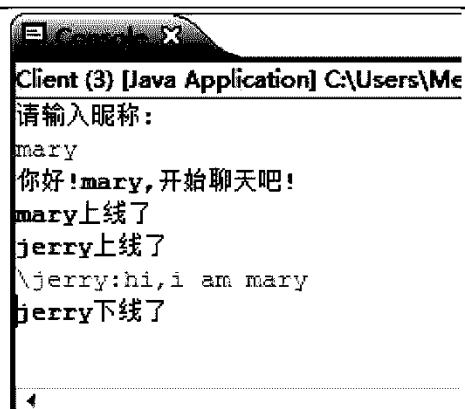


图 - 7

对于服务器端而言，只要有客户端连接，就会在界面输出提示信息。界面效果如图 - 8 所示：

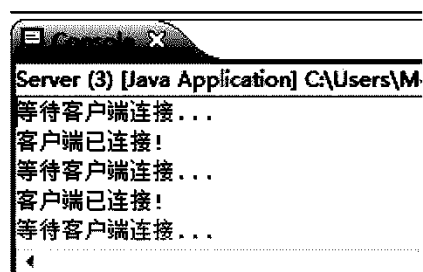


图 - 8

Java 核心 API(下)

Unit07

知识体系.....Page 197



XML 语法	XML 用途	XML 用途
	基本语法	XML 指令
		元素和属性
		大小写敏感
		元素必须有关闭标签
		必须有根元素
		元素必须正确嵌套
		实体引用
		CDATA 段
XML 解析	XML 解析方式	SAX 解析方式
		DOM 解析方式
	读取 XML	SAXReader 读取 XML 文档
		Document 的 getRootElement 方法
	Element	element 方法
		elements 方法
		getName 方法
		getText 方法
		attribute 方法
	Attribute	getName,getValue
	写 XML	构建 Document 对象
		Document 的 addElement 方法
		Element 的 addElement 方法
		Element 的 addAttribute 方法
		Element 的 addText 方法
		XMLWriter 输出 XML 文档
	XPath	路径表达式
		谓词
		通配符
		Dom4J 对 XPath 的支持

完成多条 Emp 信息的 XML 描述	XML 处理指令
	元素和属性
	大小写敏感
	元素必须有关闭标签
	必须有根元素
	元素必须正确嵌套
	实体引用
	CDATA 段
读取 XML 文档解析 Emp 信息	SAXReader 读取 XML 文档
	Document 的 getRootElement 方法
	element 方法
	elements 方法
	getName 方法
	getText 方法
	attribute 方法
	getName,getValue
将 Emp (存放在 List 中) 对象转换为 XML 文档	SAXReader 读取 XML 文档
	Document 的 getRootElement 方法
	element 方法
	elements 方法
	getName 方法
	getText 方法
	attribute 方法
	getName,getValue
在 XML 文档中查找指定特征的 Emp 信息	构建 Document 对象
	Document 的 addElement 方法
	Element 的 addElement 方法
	Element 的 addAttribute 方法
	Element 的 addText 方法
	XMLWriter 输出 XML 文档

1. XML 语法



1.1. XML 用途

1.1.1. 【XML 用途】XML 用途



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>XML用途</h4> <ul style="list-style-type: none"> • XML 指可扩展标记语言（EXtensible Markup Language），是独立于软件和硬件的信息传输工具 • XML 应用于 web 开发的许多方面，常用于简化数据的存储和共享。 • XML 简化数据共享 • XML 简化数据传输 • XML 简化平台的变更 <div style="text-align: right;">  </div>
---	---



1.2. 基本语法

1.2.1. 【基本语法】XML 指令



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>XML处理指令</h4> <ul style="list-style-type: none"> • XML处理指令，简称PI（processing instruction）。处理指令用来指挥解析引擎如何解析XML文档内容。 <ul style="list-style-type: none"> – <code><?xml version="1.0" encoding="utf-8" ?></code> – 在XML中，所有的处理指令都以<code><?</code>开始，<code>?></code>结束。<code><?</code>后面紧跟的是处理指令的名称。XML处理指令要求指定一个version属性。并允许指定可选的standalone和encoding，其中standalone是指是否允许使用外部声明，可设置为yes或no。yes是指定不使用外部声明。no为使用。encoding是指作者使用的字符编码格式。有UTF-8,GBK,gb2312等等。 <div style="text-align: right;">  </div>
---	--

1.2.2. 【基本语法】元素和属性



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>元素和属性</h4> <ul style="list-style-type: none"> • XML 文档包含 XML 元素。 • XML 元素指的是从（且包括）开始标签直到（且包括）结束标签的部分。元素可包含其他元素、文本或者两者的混合物。元素也可以拥有属性。 <pre> <datasource id="db_oracle"> <property name="url"> jdbc:thin@192.168.0.26:1521:tarena </property> <property name="dbUser">openlab</property> <property name="dbPwd">open123</property> </datasource> </pre> <div style="text-align: right;">  </div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>元素和属性(续1)</h3> <ul style="list-style-type: none"> XML 元素可以在开始标签中包含属性，属性 (Attribute) 提供关于元素的额外 (附加) 信息。属性通常提供不属于数据组成部分的信息，但是对需要处理这个元素的应用程序来说却很重要。 XML 属性必须加引号，属性值必须被引号包围，不过单引号和双引号均可使用。 如果属性值本身包含双引号，那么有必要使用单引号包围它，或者可以使用实体引用。 <pre><datasource id="db_oracle"> ... </datasource></pre> <div style="text-align: right;">  </div>
---	---

1.2.3. 【基本语法】大小写敏感

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>大小写敏感</h3> <ul style="list-style-type: none"> XML对大小写是敏感的，这一点不象HTML。在XML中，标记 < Letter> 和标记 < letter> 是不一样的。 因此，起始和结束标记的大小写应该写成相同的： <pre><letter> ... </letter> <Letter> ... </Letter></pre> <div style="text-align: right;">  </div>
---	---

1.2.4. 【基本语法】元素必须有关闭标签

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>元素必须有关闭标签</h3> <ul style="list-style-type: none"> XML要求每个元素必须由起始标签和关闭标签组成。 关闭标签与起始标签的名字相同，写法上多一个 "/" <Letter> 只有起始标记是不行的。 <Letter> </Letter> 必须要有关闭标签 <div style="text-align: right;">  </div>
---	---

1.2.5. 【基本语法】必须有根元素

Tarena
达内科技

必须有根元素

- XML要求必须有根元素，所谓根元素就是不被其它元素包围。并且根元素只能有一个。

```
<datasource id= "db_oracle" > 根元素
  <property name="url">
    jdbc:thin@192.168.0.26:1521:tarena
  </property>
  <property name="dbUser">openlab</property>
  <property name="dbPwd">open123</property>
</datasource>
```

这里不能再定义与datasource平级的元素！

Tarena
达内科技

1.2.6. 【基本语法】元素必须正确嵌套

Tarena
达内科技

元素必须正确嵌套

- XML要求所有元素必须正确的嵌套。

```
<datasource id= "db_oracle" >
  <property name="dbPwd"> open123
</datasource> 这里嵌套关系出现了错误!
</property>
```

Tarena
达内科技

1.2.7. 【基本语法】实体引用

Tarena
达内科技



实体引用

- 实体可以是常用的短语，键盘字符，文件，数据库记录或任何包含数据的项。在XML中，有时实体内包含一些字符，如 & < > " ' 等。这些均需要对其进行转义，否则会对XML解释器生成错误。

实体引用	字符	说明
<	<	小于
>	>	大于
&	&	与字符(和字符)
'	'	单引号
"	"	双引号

Tarena
达内科技


1.2.8. 【基本语法】CDATA 段

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>CDATA段</h4> <ul style="list-style-type: none"> 格式：<![CDATA [文本内容]]> 特殊标签中的实体引用都被忽略，所有内容被当成一整块文本数据对待 <div style="border: 1px solid black; padding: 5px;"> <pre><?xml version="1.0" encoding="utf-8"?> <root> <![CDATA[<hello> <world>]]> <subRoot> </subRoot> </root></pre> </div> <div style="border: 1px solid black; padding: 2px; margin-top: 5px; width: fit-content;"> 无论这里写什么，都会被当做一个文本 </div> <div style="text-align: right; margin-top: 10px;">  </div>
---	--


2. XML 解析

2.1. XML 解析方式

2.1.1. 【XML 解析方式】SAX 解析方式



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>SAX解析方式</h4> <ul style="list-style-type: none"> SAX (simple API for XML) 是一种XML解析的替代方法。相比于DOM，SAX是一种速度更快，更有效的方法。它逐行扫描文档，一边扫描一边解析。而且相比于DOM，SAX可以在解析文档的任意时刻停止解析。 优点： 解析可以立即开始，速度快，没有内存压力 缺点： 不能对节点做修改 <div style="text-align: right; margin-top: 10px;">  </div>
---	---



2.1.2. 【XML 解析方式】DOM 解析方式

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>DOM解析方式</h4> <ul style="list-style-type: none"> DOM : (Document Object Model, 即文档对象模型) 是 W3C 组织推荐的处理 XML 的一种方式。 DOM解析器在解析XML文档时，会把文档中的所有元素，按照其出现的层次关系，解析成一个个Node对象(节点)。 优点:把xml文件在内存中构造树形结构，可以遍历和修改节点 缺点： 如果文件比较大，内存有压力，解析的时间会比较长 <div style="text-align: right; margin-top: 10px;">  </div>
---	---



2.2. 读取 XML

2.2.1. 【读取 XML】SAXReader 读取 XML 文档

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>SAXReader 读取 XML 文档</h4> <ul style="list-style-type: none"> • 使用 SAXReader 需要导入 dom4j-full.jar 包。 • dom4j 是一个 Java 的 XML API，类似于 jdom，用来读写 XML 文件的。dom4j 是一个非常优秀的 Java XML API，具有性能优异、功能强大和易用特点，同时它也是一个开放源代码的软件。 • 创建 SAXReader 来读取 XML 文档。 <div style="text-align: right;">  </div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>SAXReader 读取 XML 文档(续1)</h4> <pre> public static Document readXML(String filename) throws DocumentException{ try { //创建SAXReader SAXReader reader = new SAXReader(); //读取指定文件 Document doc = reader.read(new File(filename)); return doc; } catch (DocumentException e) { e.printStackTrace(); throw e; } } </pre> <div style="text-align: right;">  </div>
---	--

2.2.2. 【读取 XML】Document 的 getRootElement 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>Document 的 getRootElement 方法</h4> <ul style="list-style-type: none"> • Document 对象是一棵文档树的根，可为我们提供对文档数据最初（或最顶层）的访问入口。 • Element 对象表示 XML 文档中的元素。元素可包含属性、其他元素或文本。如果元素含有文本，则在文本节点中表示该文本。 • Element getRootElement() 用于获取根元素。 <div style="text-align: right;">  </div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h3>Document的getRootElement方法(续1)</h3> <pre> try { Document doc = readXML("build.xml"); //获取根元素 Element root = doc.getRootElement(); } catch (Exception e) { e.printStackTrace(); } </pre> <div style="text-align: right;">++</div>
---	--

2.3. Element


2.3.1. 【Element】element 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h3>element方法</h3> <ul style="list-style-type: none"> • Element的方法: <ul style="list-style-type: none"> – Element element(String name) 获取当前元素下的指定名字的子元素 <pre> /** * 测试element方法 */ public static void testElement(Element element){ //获取当前元素下名为path的子元素 Element e = element.element("path"); System.out.println(e); } </pre> <div style="text-align: right;">++</div>
---	--


2.3.2. 【Element】elements 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h3>elements方法</h3> <ul style="list-style-type: none"> • Element的方法: <ul style="list-style-type: none"> – List elements() 获取当前元素下的所有子元素 <pre> /** * 测试elements方法 */ public static void testElements(Element element){ List<Element> elements = element.elements(); for(Element e : elements){ System.out.println(e); } } </pre> <div style="text-align: right;">++</div>
---	--


2.3.3. 【Element】getName 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>getName方法</h4> <ul style="list-style-type: none"> • getName的方法: <ul style="list-style-type: none"> – String getName() <ul style="list-style-type: none"> 获取当前元素的元素名 <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <div style="display: flex; align-items: center;"> <div style="background-color: black; color: white; padding: 2px 5px; writing-mode: vertical-rl; margin-right: 5px;">知识讲解</div> <div> <pre>/** * 测试getName方法 */ public static void testGetName(Element element){ String name = element.getName(); System.out.println(name); }</pre> </div> </div> <div style="text-align: right; margin-top: 10px;">++</div> </div>
---	--

2.3.4. 【Element】getText 方法


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>getText方法</h4> <ul style="list-style-type: none"> • getText的方法: <ul style="list-style-type: none"> – String getText() <ul style="list-style-type: none"> 获取当前元素的文本节点(起始标记与结束标记之间的文本) <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <div style="display: flex; align-items: center;"> <div style="background-color: black; color: white; padding: 2px 5px; writing-mode: vertical-rl; margin-right: 5px;">知识讲解</div> <div> <pre>/** * 测试getText方法 */ public static void testGetText(Element element){ String name = element.getText(); System.out.println(name); }</pre> </div> </div> <div style="text-align: right; margin-top: 10px;">++</div> </div>
---	---

2.3.5. 【Element】attribute 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>attribute方法</h4> <ul style="list-style-type: none"> • attribute的方法: <ul style="list-style-type: none"> – Attribute attribute(int index) <ul style="list-style-type: none"> 获取当前元素的指定属性, index为索引, 从0开始。 – Attribute attribute(String name) <ul style="list-style-type: none"> 获取当前元素的指定名字的属性。 <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <div style="display: flex; align-items: center;"> <div style="background-color: black; color: white; padding: 2px 5px; writing-mode: vertical-rl; margin-right: 5px;">知识讲解</div> <div> <pre>/** * 测试attribute方法 */ public static void testAttribute(Element element){ //获取当前元素的第一个属性 Attribute attr = element.attribute(0); //获取当前元素的name属性 //Attribute attr = element.attribute("name"); }</pre> </div> </div> <div style="text-align: right; margin-top: 10px;">++ }</div> </div>
---	--


2.4. Attribute


2.4.1. 【Attribute】getName,getValue

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>getName,getValue</h4> <ul style="list-style-type: none"> Attribute对象用于描述一个元素中的某个属性信息。 根据该对象可以获取属性名和属性值等信息 <ul style="list-style-type: none"> String getName() : 获取属性的名字 String getValue() : 获取属性的值 <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 代码清单 </div> <pre> /** * 测试Attribute方法 */ public static void testAttribute(Element element){ //获取当前元素的第一个属性 Attribute attr = element.attribute(0); System.out.println(attr.getName()); System.out.println(attr.getValue()); } </pre> <div style="text-align: right;">+</div>
---	--


2.5. 写 XML

2.5.1. 【写 XML】构建 Document 对象


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>构建Document对象</h4> <ul style="list-style-type: none"> 通过DOM4J写出XML的第一步是创建文档对象 Document。 创建Document对象的方式是通过: <ul style="list-style-type: none"> DocuemnetHelperder 的静态方法来获取 static Docuemnet createDocument() <ul style="list-style-type: none"> 创建一个Docuemnet对象并返回 <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 代码清单 </div> <div style="text-align: right;">+</div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>构建Document对象(续1)</h4> <pre> /** * 创建Document对象 */ public static Document createDocument(){ return DocumentHelper.createDocument(); } </pre> <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 代码清单 </div> <div style="text-align: right;">+</div>
---	--


2.5.2. 【写 XML】Document 的 addElement 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>Document的addElement方法</h4> <ul style="list-style-type: none"> 创建好Document对象后，通过其提供的方法addElement()方法添加根元素。 <ul style="list-style-type: none"> Element addElement() 向文档中添加根元素并返回该元素,该方法应只调用一次。 <div style="border: 1px solid black; padding: 5px;"> <pre>/** * 测试写出xml文件 */ public static void testWriteXml(){ Document doc = createDocument(); //添加根元素 Element root = doc.addElement("project"); }</pre> </div> <div style="text-align: right;">++</div>
---	---



2.5.3. 【写 XML】Element 的 addElement 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>Element的addElement方法</h4> <ul style="list-style-type: none"> 有了根元素后，我们可以向根元素中追加新的子元素，最后构成复杂的树结构。 <ul style="list-style-type: none"> Element addElement(String name) 向当前元素中添加指定名字的子元素 <div style="border: 1px solid black; padding: 5px;"> <pre>/** * 测试写出xml文件 */ public static void testWriteXml(){ ... Element root = doc.addElement("project"); //向根元素中添加名为path的子元素 Element ele = root.addElement("path"); }</pre> </div> <div style="text-align: right;">++</div>
---	---



2.5.4. 【写 XML】Element 的 addAttribute 方法

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>Element的addAttribute方法</h4> <ul style="list-style-type: none"> 我们也可以为元素添加属性 <ul style="list-style-type: none"> Element addAttribute(String name,String value) 向当前元素添加指定的属性及对应的值,返回值依然是当前元素，这样的好处在于连续追加元素时代码书写简洁。 <div style="border: 1px solid black; padding: 5px;"> <pre>/** * 测试写出xml文件 */ public static void testWriteXml(){ ... Element ele = root.addElement("path"); //为path元素添加属性id,其值为cp ele.addAttribute("id", "cp"); }</pre> </div> <div style="text-align: right;">++</div>
---	--

2.5.5. 【写 XML】Element 的 addText 方法



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>Element的addText方法</h4> <ul style="list-style-type: none"> 我们也可以为元素添加属性 <ul style="list-style-type: none"> Element addText(String text) 向当前元素添加指定内容的文本。 <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 代码讲解 </div> <pre> /** * 测试写出xml文件 */ public static void testWriteXml(){ ... //为path元素添加属性id,其值为cp ele.addAttribute("id", "cp"); //添加文本信息 ele.addText("content"); } </pre> <div style="text-align: right;">  </div>
---	--

2.5.6. 【写 XML】XMLWriter 输出 XML 文档

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>XMLWriter输出XML文档</h4> <ul style="list-style-type: none"> 当我们构建完XML文档后，可以通过XMLWriter将文档输出以生成xml文件。 <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 代码讲解 </div> <pre> /** * 测试写出xml文件 */ public static void testWriteXml()throw IOException{ ... //写出 XMLWriter writer = new XMLWriter(); FileOutputStream fos = new FileOutputStream("builder.xml"); writer.setOutputStream(fos); writer.write(doc); writer.close(); } </pre> <div style="text-align: right;">  </div>
---	--


2.6. XPath

2.6.1. 【XPath】路径表达式


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>路径表达式</h4> <ul style="list-style-type: none"> XPath 是一门在 XML 文档中查找信息的语言。XPath 可用在 XML 文档中对元素和属性进行遍历。 由于我们单纯使用dom定位节点时，大部分时间需要一层一层的处理，如果有了XPath，定位节点将变得很轻松。他可以根据路径，属性，甚至是条件进行节点的检索。 XPath 使用路径表达式在 XML 文档中进行导航 XPath 包含一个标准函数库 XPath 是 XSLT 中的主要元素 XPath 是一个 W3C 标准 <div style="border-left: 2px solid black; padding-left: 5px; margin-top: 10px;"> 代码讲解 </div> <div style="text-align: right;">  </div>
---	---


路径表达式


路径表达式(续1)



- 斜杠 (/) 作为路径内部的分隔符。
- 同一个节点有绝对路径和相对路径两种写法:
 - 路径 (absolute path) 必须用 "/" 起首, 后面紧跟根节点, 比如 /step/step/...
 - 相对路径 (relative path) 则是除了绝对路径以外的其他写法, 比如 step/step, 也就是不使用 "/" 起首。
- "." 表示当前节点。
- ".." 表示当前节点的父节点










路径表达式

路径表达式(续2)




- nodename (节点名称) : 表示选择该节点的所有子节点
- "/" : 表示选择根节点
- "//" : 表示选择任意位置的某个节点
- "@" : 表示选择某个属性






路径表达式


路径表达式(续3)





- 以此为例 :


```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
  <book>
    <title lang="eng">Harry Potter</title>
    <price>29.99</price>
  </book>
  <book>
    <title lang="eng">Learning XML</title>
    <price>39.95</price>
  </book>
</bookstore>
```










207

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>路径表达式(续4)</h3> <ul style="list-style-type: none"> • <code>/bookstore</code> : 选取根节点bookstore, 这是绝对路径写法。 • <code>bookstore/book</code> : 选取所有属于 bookstore 的子元素的book元素, 这是相对路径写法。 • <code>//book</code> : 选择所有 book 子元素, 而不管它们在文档中的位置。 • <code>bookstore//book</code> : 选择所有属于 bookstore 元素的后代的book元素, 而不管它们位于 bookstore 之下的什么位置。 • <code>//@lang</code> : 选取所有名为 lang 的属性。 <div style="text-align: right;">  </div>
---	--



2.6.2. 【XPath】谓语句



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>谓语句</h3> <ul style="list-style-type: none"> • "谓语句条件", 就是对路径表达式的附加条件。 • 所有的条件, 都写在方括号"[]"中, 表示对节点进行进一步的筛选。 <div style="text-align: right;">  </div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>谓语句(续1)</h3> <ul style="list-style-type: none"> • <code>/bookstore/book[1]</code> : 表示选择bookstore的第一个book子元素。 • <code>/bookstore/book[last()]</code> : 表示选择bookstore的最后一个book子元素。 • <code>/bookstore/book[last()-1]</code> : 表示选择bookstore的倒数第二个book子元素。 • <code>/bookstore/book[position()<3]</code> : 表示选择bookstore的前两个book子元素。 • <code>//title[@lang]</code> : 表示选择所有具有lang属性的title节点。 • <code>//title[@lang='eng']</code> : 表示选择所有lang属性的值等于"eng"的title节点。 <div style="text-align: right;">  </div>
---	---



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>谓语句(续2)</h3> <ul style="list-style-type: none"> • <code>/bookstore/book[price]</code> : 表示选择bookstore的book子元素, 且被选中的book元素必须带有price子元素。 • <code>/bookstore/book[price>35.00]</code> : 表示选择bookstore的book子元素, 且被选中的book元素的price子元素值必须大于35。 • <code>/bookstore/book[price>35.00]/title</code> : 表示在上例结果集中, 选择title子元素。 • <code>/bookstore/book/price[.>35.00]</code> : 表示选择值大于35的"/bookstore/book"的price子元素。 <div style="text-align: right;">  </div>
---	---



2.6.3. 【XPath】通配符

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>通配符</h3> <ul style="list-style-type: none"> • <code>"**"</code>表示匹配任何元素节点。 • <code>"@"</code>表示匹配任何属性值。 • <code>node()</code>表示匹配任何类型的节点。 <div style="text-align: right;">  </div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>通配符(续1)</h3> <ul style="list-style-type: none"> • <code>//*</code> : 选择文档中的所有元素节点。 • <code>/*/*</code> : 表示选择所有第二层的元素节点。 • <code>/bookstore/*</code> : 表示选择bookstore的所有元素子节点。 • <code>//title[@*]</code> : 表示选择所有带有属性的title元素。 <div style="text-align: right;">  </div>
---	--

2.6.4. 【XPath】Dom4J 对 XPath 的支持

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>Dom4J对XPath的支持</h3> <ul style="list-style-type: none"> • 采用xpath查找需要引入jaxen-xx-xx.jar。 <ul style="list-style-type: none"> – 否则会报： <code>java.lang.NoClassDefFoundError:org/jaxen/jaxenException</code>异常。 • Document提供了一个对XPATH检索的方法： • List <code>selectNodes(String xpath)</code> <ul style="list-style-type: none"> – 传入xpath路径，获取相应的信息 <div style="text-align: right;">  </div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>Dom4J对XPath的支持(续1)</h3> <pre> public static void testXPath(Document doc){ List list = doc.selectNodes("/project/path[@id= 'cp']"); System.out.println(list.size()); for(Object o : list){ System.out.println(o); } } </pre> <div style="text-align: right;">  </div>
---	--

经典案例

1. 完成多条 Emp 信息的 XML 描述

- 问题

现有多条 Emp 信息数据，如表-1 所示：

表-1 Emp 信息数据

id	name	age	gender	salary
1	张三	34	男	3000
2	李四	21	女	4000
3	王五	46	女	6500
4	赵六	28	男	4400
5	钱七	53	男	12000

表-1 中，每一行数据表示一条 Emp 信息。要求完成表-1 中 Emp 信息数据的 XML 描述。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建 XML 文件

首先，创建名为 EmpList.xml 的 XML 文件；然后，在该文件中使用处理指令设置属性 version 以及属性 encoding 的值，代码如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
```

步骤二：确定根元素

XML 要求必须有根元素，所谓根元素就是不被其它元素包围，并且根元素只能有一个。本案例使用<list>作为根元素，表示该元素内可以包含多条子元素作为 Emp 信息数据，代码如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<list>  
</list>
```

注意，在该文档中，不能再定义与 list 平级的 XML 元素。

步骤三：定义表示 Emp 信息数据的元素

首先，在根元素 <list> 下，定义一个子元素 <emp></emp>，用于表示一条 Emp 信息，代码如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<list>
```

```
  <emp>
</emp>
```

```
</list>
```

步骤四：为 <emp> 元素定义 id 属性

为元素 <emp> 定义属性 id，用于表示 Emp 信息数据中的 id，代码如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<list>
```

```
  <emp id="1">
</emp>
```

```
</list>
```

步骤五：为 <emp> 元素定义子元素

为元素 <emp> 定义子元素 <name>、<age>、<gender> 和 <salary>，分别表示 Emp 信息数据中的 name、age、gender 以及 salary。并为这四个子元素添加文本信息，以记载 Emp 的相关信息数据，代码如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<list>
  <emp id="1">
```

```
    <name>张三</name>
    <age>34</age>
    <gender>男</gender>
    <salary>3000</salary>
```

```
  </emp>
</list>
```

步骤六：实现多条 Emp 信息数据的 XML 描述

一个 <emp> 元素表示一条 Emp 数据，因此，可以用多个 <emp> 元素来描述剩余的多条 Emp 信息数据。代码如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<list>
  <emp id="1">
    <name>张三</name>
    <age>34</age>
    <gender>男</gender>
    <salary>3000</salary>
  </emp>
```

```
  <emp id="2">
```

```

    <name>李四</name>
    <age>21</age>
    <gender>女</gender>
    <salary>4000</salary>
</emp>
<emp id="3">
    <name>王五</name>
    <age>46</age>
    <gender>女</gender>
    <salary>6500</salary>
</emp>
<emp id="4">
    <name>赵六</name>
    <age>28</age>
    <gender>男</gender>
    <salary>4400</salary>
</emp>
<emp id="5">
    <name>钱七</name>
    <age>53</age>
    <gender>男</gender>
    <salary>12000</salary>
</emp>

</list>

```

• 完整代码

本案例中，EmpList.xml 文件的完整内容如下所示：

```

<?xml version="1.0" encoding="UTF-8"?>
<list>
  <emp id="1">
    <name>张三</name>
    <age>34</age>
    <gender>男</gender>
    <salary>3000</salary>
  </emp>
  <emp id="2">
    <name>李四</name>
    <age>21</age>
    <gender>女</gender>
    <salary>4000</salary>
  </emp>
  <emp id="3">
    <name>王五</name>
    <age>46</age>
    <gender>女</gender>
    <salary>6500</salary>
  </emp>
  <emp id="4">
    <name>赵六</name>
    <age>28</age>
    <gender>男</gender>
    <salary>4400</salary>
  </emp>
  <emp id="5">
    <name>钱七</name>
    <age>53</age>

```

```
<gender>男</gender>
<salary>12000</salary>
</emp>
</list>
```

2. 读取 XML 文档解析 Emp 信息

- 问题

解析上一案例中创建的 XML 文档 EmpList.xml。首先，将每一个<emp>节点中的属性和子元素封装为一个 Emp 对象；然后，将 Emp 对象存储到 List 集合中并输出到控制台。

- 方案

dom4j 是一个 Java 的 XML API，用来读写 XML 文件。dom4j 是一个非常优秀的 Java XML API，具有性能优异、功能强大和极端易用的特点，同时它也是一个开放源代码的软件。使用 dom4j 来实现对 XML 文档的解析，详细过程如下：

- 1) 使用 dom4j 需要导入 dom4j 对应的 jar 包。
- 2) 创建 SAXReader 类的对象来实现读取 XML 文档，代码如下：

```
SAXReader reader = new SAXReader();
```

3) 使用 SAXReader 类的 read 方法获取 Document 对象，Document 对象是一棵文档树的根，可为我们提供对文档数据的最初（或最顶层）的访问入口，代码如下：

```
Document doc = reader.read(new File("EmpList.xml"));
```

4) 使用 Document 对象的 getRootElement 方法获取要解析的 XML 文档的根元素，该方法返回值类型为 Element。Element 对象表示 XML 文档中的元素。元素可包含属性、其它元素或文本。如果元素含有文本，则在文本节点中表示该文本，代码如下：

```
Element root = doc.getRootElement();
```

5) 接下来，可以使用 Element 对象提供的方法继续解析 XML 文档，例如：其 elements 方法用来获取当前元素下的所有子元素，代码如下：

```
List<Element> elements = root.elements();
```

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：导入 dom4j 对应的 jar 包

在当前工程下导入 dom4j 对应的 jar 包。

步骤二：新建类及测试方法

首先，新建类 TestDom；然后在该类中新建测试方法 testReadXml，代码如下所示：

```
import org.junit.Test;

public class TestDom {
    /**
     * 使用 DOM 解析 XML 文件
     */
    @Test
    public void testReadXml() {
    }
```

步骤三：创建 SAXReader 类的对象，获取 Document 对象

创建 SAXReader 类的对象来实现读取 XML 文档；然后，使用 SAXReader 类的 read 方法获取 Document 对象，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 DOM 解析 XML 文件
     */
    @Test
    public void testReadXml() {

        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("EmpList.xml"));
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

步骤四：获取根元素

使用 Document 对象的 getRootElement 方法获取 EmpList.xml 文档的根元素，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;
```

```
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 DOM 解析 XML 文件
     */
    @Test
    public void testReadXml() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("EmpList.xml"));

            // 获取根节点 list
            Element root = doc.getRootElement();

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

步骤五：获取<list>节点下的所有子元素

使用 Element 对象的 elements 方法获取<list>节点下的所有子元素 即所有的<emp>节点，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 DOM 解析 XML 文件
     */
    @Test
    public void testReadXml() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("EmpList.xml"));
            // 获取根节点 list
            Element root = doc.getRootElement();
```

```
// 获取 list 下的所有子节点 emp
```

```
List<Element> elements = root.elements();
```

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

步骤六：封装 Emp 对象，存储到 List 集合中

1) 创建存储的数据类型为 Emp 类型的 List 集合 emps；

2) 循环上一步中的 elements 集合，每循环一次获取一个 emp 元素。在循环中使用 Element 对象的 attribute 方法获取 id 属性对应的 Attribute 对象，再使用 Attribute 对象的 getValue 方法就可以获取到属性 id 对应的文本信息，即 Emp 对象的属性 id 的信息；

3) 在循环中，使用 Element 对象的 elementText 方法获取节点<name>、<age>、<gender>以及<salary>对应的文本信息，即 Emp 对象的属性 name、age、gender 以及 salary。

4) 在循环中，将上述获取到的信息封装为 Emp 对象，存储到集合 emps 中。

代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 DOM 解析 XML 文件
     */
    @Test
    public void testReadXml() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("EmpList.xml"));
            // 获取根节点 list
            Element root = doc.getRootElement();
            // 获取 list 下的所有子节点 emp
            List<Element> elements = root.elements();
            // 保存所有员工对象的集合

            List<Emp> emps = new ArrayList<Emp>();
```



```

        for (Element element : elements) {
            int id = Integer.parseInt(element.attribute("id").getValue());
            String name = element.elementText("name");
            int age = Integer.parseInt(element.elementText("age"));
            String gender = element.elementText("gender");
            double salary = Double.parseDouble(element
                .elementText("salary"));
            Emp emp = new Emp(id, name, age, gender, salary);
            emps.add(emp);
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

步骤七：输出集合

将 emps 集合的信息输出到控制台，代码如下所示：

```

import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 DOM 解析 XML 文件
     */
    @Test
    public void testReadXml() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("EmpList.xml"));
            // 获取根节点 list
            Element root = doc.getRootElement();
            // 获取 list 下的所有子节点 emp
            List<Element> elements = root.elements();
            // 保存所有员工对象的集合
            List<Emp> emps = new ArrayList<Emp>();
            for (Element element : elements) {
                int id = Integer.parseInt(element.attribute("id").getValue());
                String name = element.elementText("name");
                int age = Integer.parseInt(element.elementText("age"));
                String gender = element.elementText("gender");
                double salary = Double.parseDouble(element
                    .elementText("salary"));
                Emp emp = new Emp(id, name, age, gender, salary);
                emps.add(emp);
            }
        }
    }
}

```

```

        System.out.println("解析完毕");
        System.out.println(emps);

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

步骤八：运行

运行 testReadXml 方法，控制台输出结果如下所示：

解析完毕

```

[Emp [id=1,name=张三, age=34, gender=男, salary=3000.0], Emp [id=2,name=李四,
age=21, gender=女, salary=4000.0], Emp [id=3,name=王五, age=46, gender=女,
salary=6500.0], Emp [id=4,name=赵六, age=28, gender=男, salary=4400.0], Emp
[id=5,name=钱七, age=53, gender=男, salary=12000.0]]

```

从输出结果可以看出，已经对 EmpList.xml 文档进行解析，将每一个<emp>节点中的属性和子元素封装为一个 Emp 对象并将 Emp 对象存储到 List 集合中。

• 完整代码

本案例中，类 TestDom 的完整代码如下所示：

```

import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 DOM 解析 XML 文件
     */
    public void testReadXml() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("EmpList.xml"));
            // 获取根节点 list
            Element root = doc.getRootElement();
            // 获取 list 下的所有子节点 emp
            List<Element> elements = root.elements();
            // 保存所有员工对象的集合
            List<Emp> emps = new ArrayList<Emp>();
            for (Element element : elements) {
                int id = Integer.parseInt(element.attribute("id").getValue());

```

```
String name = element.elementText("name");
int age = Integer.parseInt(element.elementText("age"));
String gender = element.elementText("gender");
double salary = Double.parseDouble(element
    .elementText("salary"));
Emp emp = new Emp(id, name, age, gender, salary);
emps.add(emp);
}
System.out.println("解析完毕");
System.out.println(emps);
} catch (Exception e) {
    e.printStackTrace();
}
}
}
```

3. 将 Emp (存放在 List 中) 对象转换为 XML 文档

- 问题

在 List 集合中存储了如下数据：

```
List<Emp> emps = new ArrayList<Emp>();
emps.add(new Emp(1, "张三", 33, "男", 9000));
emps.add(new Emp(2, "李四", 26, "男", 5000));
emps.add(new Emp(3, "王五", 48, "男", 34000));
```

请将集合 emps 中的所有 Emp 对象转换为 XML 文件的形式。

- 方案

使用 dom4j 建立 XML 文档的过程如下：

1) 创建文档对象，代码如下：

```
Document doc = DocumentHelper.createDocument();
```

2) 创建根节点，代码如下：

```
Element root = doc.addElement("list");
```

3) 在节点下添加注释、属性、子节点，Element 提供如下方法：

addComment：方法添加注释

addAttribute：添加属性

addElement：添加子元素

4) 通过 XMLWriter 生成物理文件。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：添加测试方法 testWriteXml

首先在 TestDom 类中新建测试方法 testWriteXml；然后在该测试方法中，添加 List 集合存储 Emp 对象的代码，代码如下所示：

```
import java.util.ArrayList;
import java.util.List;

import org.junit.Test;

public class TestDom {
    /**
     * 测试写 xml
     */

    @Test
    public void testWriteXml() {
        List<Emp> emps = new ArrayList<Emp>();
        emps.add(new Emp(1, "张三", 33, "男", 9000));
        emps.add(new Emp(2, "李四", 26, "男", 5000));
        emps.add(new Emp(3, "王五", 48, "男", 34000));
    }
}
```

步骤二：创建文档对象

使用 DocumentHelper 类的静态方法 createDocument 创建文档对象 Document，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 测试写 xml
     */
    @Test
    public void testWriteXml() {
        List<Emp> emps = new ArrayList<Emp>();
        emps.add(new Emp(1, "张三", 33, "男", 9000));
        emps.add(new Emp(2, "李四", 26, "男", 5000));
        emps.add(new Emp(3, "王五", 48, "男", 34000));

        try {
            Document doc = DocumentHelper.createDocument();
        } catch (Exception e) {
```

```
        e.printStackTrace();
    }

}
```

步骤三：创建根节点

使用 Document 类的 addElement 方法，创建根节点<list>，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 测试写 xml
     */
    @Test
    public void testWriteXml() {
        List<Emp> emps = new ArrayList<Emp>();
        emps.add(new Emp(1, "张三", 33, "男", 9000));
        emps.add(new Emp(2, "李四", 26, "男", 5000));
        emps.add(new Emp(3, "王五", 48, "男", 34000));

        try {
            Document doc = DocumentHelper.createDocument();
            // 添加根标记

            Element root = doc.addElement("list");

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

步骤四：添加子元素

首先，循环集合 emps，在循环中获取 Emp 对象的信息；然后，在循环中使用 Element 类的 addAttribute 方法在该元素下添加属性，属性的值为对应 Emp 对象中的成员变量的值；使用 addElement 方法在该元素下添加子元素；使用 addText 方法为该子元素添加文本，该文本也为对应 Emp 对象中成员变量的值，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
```

```
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 测试写 xml
     */
    @Test
    public void testWriteXml() {
        List<Emp> emps = new ArrayList<Emp>();
        emps.add(new Emp(1, "张三", 33, "男", 9000));
        emps.add(new Emp(2, "李四", 26, "男", 5000));
        emps.add(new Emp(3, "王五", 48, "男", 34000));

        try {
            Document doc = DocumentHelper.createDocument();
            // 添加根标记

            Element root = doc.addElement("list");
            for (Emp emp : emps) {
                // 向根元素中添加名为 emp 的子元素
                Element ele = root.addElement("emp");
                // 为 emp 元素添加属性 id, 其值为 cp
                ele.addAttribute("id", emp.getId() + "");
                ele.addElement("name").addText(emp.getName());
                ele.addElement("age").addText(emp.getAge() + "");
                ele.addElement("gender").addText(emp.getGender());
                ele.addElement("salary").addText(emp.getSalary() + "");
            }

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

步骤五：生成物理文件

通过 XMLWriter 生成物理文件，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 测试写 xml
     */
    @Test
    public void testWriteXml() {
        List<Emp> emps = new ArrayList<Emp>();
```

```
emps.add(new Emp(1, "张三", 33, "男", 9000));
emps.add(new Emp(2, "李四", 26, "男", 5000));
emps.add(new Emp(3, "王五", 48, "男", 34000));

try {
    Document doc = DocumentHelper.createDocument();
    // 添加根标记
    Element root = doc.addElement("list");
    for (Emp emp : emps) {
        // 向根元素中添加名为 emp 的子元素
        Element ele = root.addElement("emp");
        // 为 emp 元素添加属性 id
        ele.addAttribute("id", emp.getId() + "");
        ele.addElement("name").addText(emp.getName());
        ele.addElement("age").addText(emp.getAge() + "");
        ele.addElement("gender").addText(emp.getGender());
        ele.addElement("salary").addText(emp.getSalary() + "");
    }
    // 写出

    XMLWriter writer = new XMLWriter();
    FileOutputStream fos = new FileOutputStream("emps.xml");
    writer.setOutputStream(fos);
    writer.write(doc);
    writer.close();

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

步骤六：运行

运行 `testWriteXml` 方法，会在当前工程目录下生成 `emps.xml` 文件，该文件中的内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<list>
  <emp id="1">
    <name>张三</name>
    <age>33</age>
    <gender>男</gender>
    <salary>9000.0</salary>
  </emp>
  <emp id="2">
    <name>李四</name>
    <age>26</age>
    <gender>男</gender>
    <salary>5000.0</salary>
  </emp>
  <emp id="3">
    <name>王五</name>
    <age>48</age>
    <gender>男</gender>
    <salary>34000.0</salary>
  </emp>
</list>
```

- **完整代码**

本案例的完整代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    //... (之前案例的代码, 略)

    /**
     * 测试写 xml
     */
    @Test
    public void testWriteXml() {
        List<Emp> emps = new ArrayList<Emp>();
        emps.add(new Emp(1, "张三", 33, "男", 9000));
        emps.add(new Emp(2, "李四", 26, "男", 5000));
        emps.add(new Emp(3, "王五", 48, "男", 34000));

        try {
            Document doc = DocumentHelper.createDocument();
            // 添加根标记
            Element root = doc.addElement("list");
            for (Emp emp : emps) {
                // 向根元素中添加名为 emp 的子元素
                Element ele = root.addElement("emp");
                // 为 emp 元素添加属性 id
                ele.addAttribute("id", emp.getId() + "");
                ele.addElement("name").addText(emp.getName());
                ele.addElement("age").addText(emp.getAge() + "");
                ele.addElement("gender").addText(emp.getGender());
                ele.addElement("salary").addText(emp.getSalary() + "");
            }
            // 写出
            XMLWriter writer = new XMLWriter();
            FileOutputStream fos = new FileOutputStream("emps.xml");
            writer.setOutputStream(fos);
            writer.write(doc);
            writer.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4. 在 XML 文档中查找指定特征的 Emp 信息

- **问题**

在上一案例中，我们创建了 emps.xml 文件，本案例要求查找该文件中属性 id 的值为

2 的<emp>节点，并读取该节点下子节点<name>的文本信息。另外，要求使用 XPath 来实现。

- **方案**

XPath 是一门在 XML 文档中查找信息的语言。XPath 可用来在 XML 文档中对元素和属性进行遍历。我们使用 DOM 定位节点时，大部分时间需要一层一层的处理，如果有了 XPath，我们定位节点将变得很轻松。它可以根据路径，属性，甚至是条件进行节点的检索。例如本案例中，检索属性 id 的值为 2 的<emp>节点，则可以使用如下路径表达式：

```
/list/emp[@id='2']
```

上述表达式中还使用谓语句条件。所谓“谓语句条件”，就是对路径表达式的附加条件。所有的条件，都写在方括号“[]”中，表示对节点进行进一步的筛选。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：新建测试方法

在 TestDom 类中新建测试方法 findId，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 XPath 查找指定信息
     */

    @Test
    public void findId() {
    }
}
```

步骤二：创建 SAXReader 类的对象，获取 Document 对象

首先，创建 SAXReader 类的对象来实现读取 XML 文档；然后，使用 SAXReader 类的 read 方法获取 Document 对象，代码如下所示：

```
import java.io.File;
```

```
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 XPath 查找指定信息
     */
    @Test
    public void findId() {
        try {

            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("emps.xml"));

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

步骤三：使用 XPath 和谓词条件查找节点

使用 Element 对象的 selectNodes 方法，获取 XPath 和谓词条件为 “/list/emp[@id='2']” 的所有元素，即查找到所有 id 为 2 的 emp 元素，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 XPath 查找指定信息
     */
    @Test
    public void findId() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("emps.xml"));
            // 查找 id 为 2 的用户信息
```

```
        List list = doc.selectNodes("/list/emp[@id='2']");
        System.out.println("选取了:" + list.size() + "条数据");

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

步骤四：获取<emp>节点下<name>子节点的文本信息

循环遍历上一步得到的 list 集合，在循环中使用 Element 对象的 elementText 方法获取<emp>节点下<name>子节点的文本信息，代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    /**
     * 使用 XPath 查找指定信息
     */
    @Test
    public void findId() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("emps.xml"));
            // 查找 id 为 1 的用户信息
            List list = doc.selectNodes("/list/emp[@id='2']");
            System.out.println("选取了:" + list.size() + "条数据");

            for (Object o : list) {
                Element e = (Element) o;
                System.out.println("name:" + e.elementText("name"));
            }

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

步骤五：运行

运行 findId 方法，控制台输出结果如下：

选取了:1 条数据

name:李四

从输出结果可以看出,已经查找到了 emps.xml 文件中属性 id 的值为 2 的<emp>节点,并读取了该节点下子节点<name>的文本信息。

• 完整代码

本案例的完整代码如下所示：

```
import java.io.File;
import java.io.FileOutputStream;
import java.util.ArrayList;
import java.util.List;

import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;
import org.junit.Test;

public class TestDom {
    //... (之前案例的代码, 略)

    /**
     * 使用 XPath 查找指定信息
     */
    @Test
    public void findId() {
        try {
            // 创建 SAXReader
            SAXReader reader = new SAXReader();
            // 读取指定文件
            Document doc = reader.read(new File("emps.xml"));
            // 查找 id 为 1 的用户信息
            List list = doc.selectNodes("/list/emp[@id='2']");
            System.out.println("选取了:" + list.size() + "条数据");
            for (Object o : list) {
                Element e = (Element) o;
                System.out.println("name:" + e.elementText("name"));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

课后作业

1. 下列 XML 文档中，符合 XML 语法规则的是

A .

```
<customer>
  <address>123 MainStreet</Address>
</customer>
```

B .

```
<customer>
  <name>Joe's XML Works</name>
  <address>New York
</costomer>
```

C .

```
<customer type=extemal>
  <name>Partners Unlimited</name>
</customer>
```

D.

```
<customer name="JohnDoe">
  <address>123 Main Street</address>
  <zip code="01837"/>
</customer>
```

2. 简述 SAX 和 DOM 解析方式的不同

3. 客户端以 XML 格式向服务器端发送数据 V1

客户端以 XML 格式向服务器端发送数据，详细要求如下：

- 1) 客户端读取 EmpList.xml 数据，将其发送到服务器端。
- 2) 服务器接收到 XML 格式的数据后，进行解析，将解析到的数据输出到控制台。

4. 客户端以 XML 格式向服务器端发送数据 V2(提高题，选做)

客户端以 XML 格式向服务器端发送数据，详细要求如下：

- 1) 客户端读取 EmpList.xml 中的数据，将其发送到服务器端。
- 2) 服务器接收到 XML 格式的数据后，将其输出到 Emp_Server.xml 文件中。