## java.io 包下的类有哪些 + 面试题

### IO 介绍

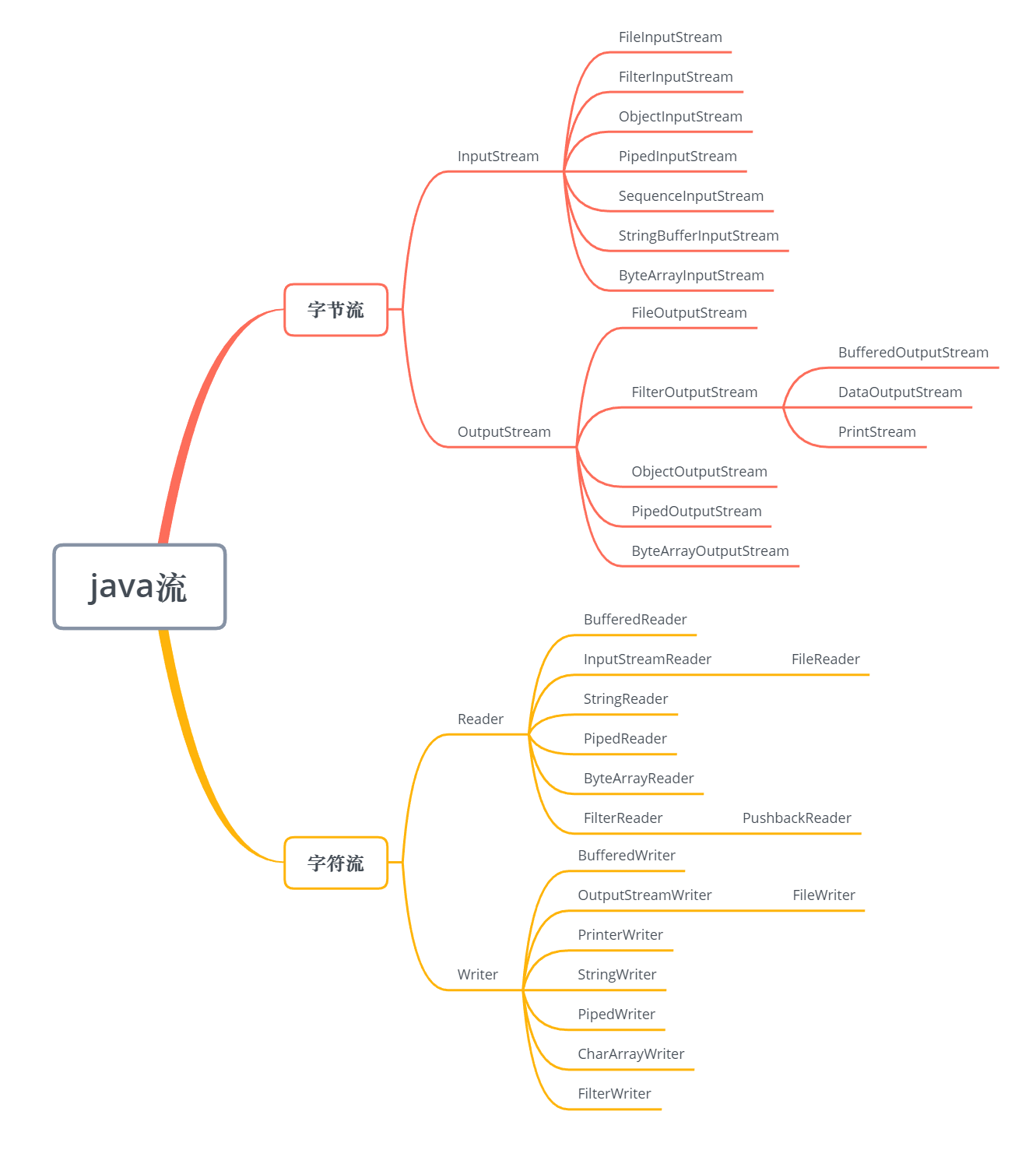
IO 是 Input/Output 的缩写，它是基于流模型实现的，比如操作文件时使用输入流和输出流来写入和读取文件等。

#### IO 分类

传统的 IO，按照流类型我们可以分为：

* 字符流
* 字节流

其中，字符流包括 Reader、Writer；字节流包括 InputStream、OutputStream。  
传统 IO 的类关系图，如下图所示：



avatar

#### IO 使用

了解了 IO 之间的关系，下面我们正式进入实战环节，分别来看字符流（Reader、Writer）和字节流（InputStream、OutputStream）的使用。

##### ① Writer 使用

Writer 可用来写入文件，请参考以下代码：

// 给指定目录下的文件追加信息  
Writer writer = new FileWriter("d:\\io.txt",true);  
writer.append("老王");  
writer.close();

这几行简单的代码就可以实现把信息 老王 追加到 d:\\io.txt 的文件下，参数二表示的是覆盖文字还是追加文字。

##### ② Reader 使用

Reader 可用来读取文件，请参考以下代码：

Reader reader = new FileReader("d:\\io.txt");  
BufferedReader bufferedReader = new BufferedReader(reader);  
String str = null;  
// 逐行读取信息  
while (null != (str = bufferedReader.readLine())) {  
 System.out.println(str);  
}  
bufferedReader.close();  
reader.close();

##### ③ InputStream 使用

InputStream 可用来读取文件，请参考以下代码：

InputStream inputStream = new FileInputStream(new File("d:\\io.txt"));  
byte[] bytes = new byte[inputStream.available()];  
// 读取到 byte 数组  
inputStream.read(bytes);  
// 内容转换为字符串  
String content = new String(bytes, "UTF-8");  
inputStream.close();

##### ④ OutputStream 使用

OutputStream 可用来写入文件，请参考以下代码：

OutputStream outputStream = new FileOutputStream(new File("d:\\io.txt"),true);  
outputStream.write("老王".getBytes());  
outputStream.close();

### NIO 介绍

上面讲的内容都是 java.io 包下的知识点，但随着 Java 的不断发展，在 Java 1.4 时新的 IO 包出现了 java.nio，NIO（Non-Blocking IO）的出现解决了传统 IO，也就是我们经常说的 BIO（Blocking IO）同步阻塞的问题，NIO 提供了 Channel、Selector 和 Buffer 等概念，可以实现多路复用和同步非阻塞 IO 操作，从而大大提升了 IO 操作的性能。  
前面提到同步和阻塞的问题，那下面来看看同步和阻塞结合都有哪些含义。

|  |  |
| --- | --- |
| 组合方式 | 性能分析 |
| 同步阻塞 | 最常用的一种用法，使用也是最简单的，但是 I/O 性能一般很差，CPU 大部分在空闲状态 |
| 同步非阻塞 | 提升 I/O 性能的常用手段，就是将 I/O 的阻塞改成非阻塞方式，尤其在网络 I/O |

是长连接，同时传输数据也不是很多的情况下，提升性能非常有效。 这种方式通常能提升 I/O 性能，但是会增加 CPU 消耗，要考虑增加的 I/O 性能能不能补偿 CPU 的消耗，也就是系统的瓶颈是在 I/O 还是在 CPU 上  
异步阻塞 | 这种方式在分布式数据库中经常用到。例如，在往一个分布式数据库中写一条记录，通常会有一份是同步阻塞的记录，而还有两至三份是备份记录会写到其他机器上，这些备份记录通常都是采用异步阻塞的方式写 I/O；异步阻塞对网络 I/O 能够提升效率，尤其像上面这种同时写多份相同数据的情况  
异步非阻塞 | 这种组合方式用起来比较复杂，只有在一些非常复杂的分布式情况下使用，像集群之间的消息同步机制一般用这种 I/O 组合方式。例如，Cassandra 的 Gossip 通信机制就是采用异步非阻塞的方式。它适合同时要传多份相同的数据到集群中不同的机器，同时数据的传输量虽然不大，但是却非常频繁。这种网络 I/O 用这个方式性能能达到最高

了解了同步和阻塞的含义，下面来看 **NIO 的具体使用** ，请参考以下代码：

int port = 6666;  
new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try (Selector selector = Selector.open();  
 ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();) {  
 serverSocketChannel.bind(new InetSocketAddress(InetAddress.getLocalHost(), port));  
 serverSocketChannel.configureBlocking(false);  
 serverSocketChannel.register(selector, SelectionKey.OP\_ACCEPT);  
 while (true) {  
 selector.select(); // 阻塞等待就绪的 Channel  
 Set<SelectionKey> selectionKeys = selector.selectedKeys();  
 Iterator<SelectionKey> iterator = selectionKeys.iterator();  
 while (iterator.hasNext()) {  
 SelectionKey key = iterator.next();  
 try (SocketChannel channel = ((ServerSocketChannel) key.channel()).accept()) {  
 channel.write(Charset.defaultCharset().encode("老王，你好~"));  
 }  
 iterator.remove();  
 }  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}).start();  
  
new Thread(new Runnable() {  
 @Override  
 public void run() {  
 // Socket 客户端 1（接收信息并打印）  
 try (Socket cSocket = new Socket(InetAddress.getLocalHost(), port)) {  
 BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(cSocket.getInputStream()));  
 bufferedReader.lines().forEach(s -> System.out.println("客户端 1 打印：" + s));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}).start();  
  
new Thread(new Runnable() {  
 @Override  
 public void run() {  
 // Socket 客户端 2（接收信息并打印）  
 try (Socket cSocket = new Socket(InetAddress.getLocalHost(), port)) {  
 BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(cSocket.getInputStream()));  
 bufferedReader.lines().forEach(s -> System.out.println("客户端 2 打印：" + s));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}).start();

以上代码创建了两个 Socket 客户端，用于收取和打印服务器端的消息。  
其中，服务器端通过 SelectionKey（选择键）获取到 SocketChannel（通道），而通道都注册到 Selector（选择器）上，所有的客户端都可以获得对应的通道，而不是所有客户端都排队堵塞等待一个服务器连接，这样就实现多路复用的效果了。多路指的是多个通道（SocketChannel），而复用指的是一个服务器端连接重复被不同的客户端使用。

### AIO 介绍

AIO（Asynchronous IO）是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。  
AIO 实现简单的 Socket 服务器，代码如下：

int port = 8888;  
new Thread(new Runnable() {  
 @Override  
 public void run() {  
 AsynchronousChannelGroup group = null;  
 try {  
 group = AsynchronousChannelGroup.withThreadPool(Executors.newFixedThreadPool(4));  
 AsynchronousServerSocketChannel server = AsynchronousServerSocketChannel.open(group).bind(new InetSocketAddress(InetAddress.getLocalHost(), port));  
 server.accept(null, new CompletionHandler<AsynchronousSocketChannel, AsynchronousServerSocketChannel>() {  
 @Override  
 public void completed(AsynchronousSocketChannel result, AsynchronousServerSocketChannel attachment) {  
 server.accept(null, this); // 接收下一个请求  
 try {  
 Future<Integer> f = result.write(Charset.defaultCharset().encode("Hi, 老王"));  
 f.get();  
 System.out.println("服务端发送时间：" + DateFormat.getDateTimeInstance().format(new Date()));  
 result.close();  
 } catch (InterruptedException | ExecutionException | IOException e) {  
 e.printStackTrace();  
 }  
 }  
 @Override  
 public void failed(Throwable exc, AsynchronousServerSocketChannel attachment) {  
 }  
 });  
 group.awaitTermination(Long.MAX\_VALUE, TimeUnit.SECONDS);  
 } catch (IOException | InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}).start();  
  
// Socket 客户端  
AsynchronousSocketChannel client = AsynchronousSocketChannel.open();  
Future<Void> future = client.connect(new InetSocketAddress(InetAddress.getLocalHost(), port));  
future.get();  
ByteBuffer buffer = ByteBuffer.allocate(100);  
client.read(buffer, null, new CompletionHandler<Integer, Void>() {  
 @Override  
 public void completed(Integer result, Void attachment) {  
 System.out.println("客户端打印：" + new String(buffer.array()));  
 }  
  
 @Override  
 public void failed(Throwable exc, Void attachment) {  
 exc.printStackTrace();  
 try {  
 client.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
});  
Thread.sleep(10 \* 1000);

### 相关面试题

#### 1.使用以下哪个方法来判断一个文件是否存在？

A：createFile  
B：exists  
C：read  
D：exist

答：B

#### 2.以下说法错误的是？

A：同步操作不一定会阻塞  
B：异步操作不一定会阻塞  
C：阻塞一定是同步操作  
D：同步或异步都可能会阻塞

答：C

题目解析：异步操作也可能会阻塞，比如分布式集群消息同步，采用的就是异步阻塞的方式。

#### 3.BIO、NIO、AIO 的区别是什么？

答：它们三者的区别如下。

* BIO 就是传统的 java.io 包，它是基于流模型实现的，交互的方式是同步、阻塞方式，也就是说在读入输入流或者输出流时，在读写动作完成之前，线程会一直阻塞在那里，它们之间的调用是可靠的线性顺序。它的优点就是代码比较简单、直观；缺点就是 IO 的效率和扩展性很低，容易成为应用性能瓶颈。
* NIO 是 Java 1.4 引入的 java.nio 包，提供了 Channel、Selector、Buffer 等新的抽象，可以构建多路复用的、同步非阻塞 IO 程序，同时提供了更接近操作系统底层高性能的数据操作方式。
* AIO 是 Java 1.7 之后引入的包，是 NIO 的升级版本，提供了异步非堵塞的 IO 操作方式，因此人们叫它 AIO（Asynchronous IO），异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。

简单来说 BIO 就是传统 IO 包，产生的最早；NIO 是对 BIO 的改进提供了多路复用的同步非阻塞 IO，而 AIO 是 NIO 的升级，提供了异步非阻塞 IO。

#### 4.读取和写入文件最简洁的方式是什么？

答：使用 Java 7 提供的 Files 读取和写入文件是最简洁，请参考以下代码：

// 读取文件  
byte[] bytes = Files.readAllBytes(Paths.get("d:\\io.txt"));  
// 写入文件  
Files.write(Paths.get("d:\\io.txt"), "追加内容".getBytes(), StandardOpenOption.APPEND);

读取和写入都是一行代码搞定，可以说很简洁了。

#### 5.Files 常用方法都有哪些？

答：Files 是 Java 1.7 提供的，使得文件和文件夹的操作更加方便，它的常用方法有以下几个：

* Files. exists()：检测文件路径是否存在
* Files. createFile()：创建文件
* Files. createDirectory()：创建文件夹
* Files. delete()：删除一个文件或目录
* Files. copy()：复制文件
* Files. move()：移动文件
* Files. size()：查看文件个数
* Files. read()：读取文件
* Files. write()：写入文件

#### 6.FileInputStream 可以实现什么功能？

答：FileInputStream 可以实现文件的读取。

题目解析：因为 FileInputStream 和 FileOutputStream 很容易被记反，FileOutputStream 才是用来写入文件的，所以也经常被面试官问到。

#### 7.不定项选择：为了提高读写性能，可以采用什么流？

A：InputStream  
B：DataInputStream  
C：BufferedReader  
D：BufferedInputStream  
E：OutputStream  
F：BufferedOutputStream

答：D、F

题目解析：BufferedInputStream 是一种带缓存区的输入流，在读取字节数据时可以从底层流中一次性读取多个字节到缓存区，而不必每次都调用系统底层；同理，BufferedOutputStream 也是一种带缓冲区的输出流，通过缓冲区输出流，应用程序先把字节写入缓冲区，缓存区满后再调用操作系统底层，从而提高系统性能，而不必每次都去调用系统底层方法。

#### 8.FileInputStream 和 BufferedInputStream 的区别是什么？

答：FileInputStream 在小文件读写时性能较好，而在大文件操作时使用 BufferedInputStream 更有优势。

#### 9.以下这段代码运行在 Windwos 平台，执行的结果是？

Files.createFile(Paths.get("c:\\pf.txt"), PosixFilePermissions.asFileAttribute(  
 EnumSet.of(PosixFilePermission.OWNER\_READ)));

A：在指定的盘符产生了对应的文件，文件只读  
B：在指定的盘符产生了对应的文件，文件只写  
C：在指定的盘符产生了对应的文件，文件可读写  
D：程序报错

答：D

题目解析：本题目考察的是 Files.createFile 参数传递的问题，PosixFilePermissions 不支持 Windows，因此在 Windows 执行会报错 java.lang.UnsupportedOperationException: ‘posix:permissions’ not supported as initial attribute。

### 总结

在 Java 1.4 之前只有 BIO（Blocking IO）可供使用，也就是 java.io 包下的那些类，它的缺点是同步阻塞式运行的。随后在 Java 1.4 时，提供了 NIO（Non-Blocking IO）属于 BIO 的升级，提供了同步非阻塞的 IO 操作方式，它的重要组件是 Selector（选择器）、Channel（通道）、Buffer（高效数据容器）实现了多路复用的高效 IO 操作。而 AIO（Asynchronous IO）也叫 NIO 2.0，属于 NIO 的补充和升级，提供了异步非阻塞的 IO 操作。

还有另一个重要的知识点，是 Java 7.0 时新增的 Files 类，极大地提升了文件操作的便利性，比如读、写文件 Files.write()、Files.readAllBytes() 等，都是非常简便和实用的方法。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/interview-code/src/main/java/com/interview)

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持