二

Java IO 体系、线程模型大总结

Java 中的 I/O 按照其发展历程，可以划分为传统 IO（阻塞式 I/O）和新 IO（非阻塞式 I/O）。

传统 I/O

传统 IO 也称为 BIO（Blocking IO），是面向字节流或字符流编程的 I/O 方式。

一个典型的基于 BIO 的文件复制程序，字节流方式：

```
public class FileCopy01 {
    public static void main(String[] args) {
        //使用 jdk7 引入的自动关闭资源的 try 语句（该资源类要实现 AutoCloseable 或 Clos
        try (FileInputStream fis = new FileInputStream("D:\\file01.txt");
             FileOutputStream fos = new FileOutputStream("D:\\file01_copy.txt")) {
            byte[] buf = new byte[126];
            int hasRead = 0;
            while ((hasRead = fis.read(buf)) > 0) {
                //每次读取多少就写多少
                fos.write(buf, 0, hasRead);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

字符流方式：

```
public class FileCopy02 {
    public static void main(String[] args) {
        //使用 jdk7 引入的自动关闭资源的 try 语句
        try (FileReader fr = new FileReader("D:\\file01.txt");
             FileWriter fw = new FileWriter("D:\\file01_copy2.txt")) {
            char[] buf = new char[2];
            int hasRead = 0;
            while ((hasRead = fr.read(buf)) > 0) {
                //每次读取多少就写多少
                fw.write(buf, 0, hasRead);
```

```
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

字符缓冲，按行读取：

```
public class FileCopy02_2 {
    public static void main(String[] args) {
        //使用普通的 Reader 不方便整行读取,可以使用 BufferReader 包装,资源变量要定义在 t
        try (FileReader fr = new FileReader("D:\\file01.txt");
             FileWriter fw = new FileWriter("D:\\file01_copy2_2.txt");
             BufferedReader bufferedReader = new BufferedReader(fr);
             BufferedWriter bufferedWriter = new BufferedWriter(fw)) {
            String line;
            while ((line = bufferedReader.readLine()) != null) {
                //每次读取一行、写入一行
                bufferedWriter.write(line);
                bufferedWriter.newLine();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

随机读写（RandomAccessFile）：

```
public class FileCopy03 {
    public static void main(String[] args) {
        try (RandomAccessFile in = new RandomAccessFile("D:\\file01.txt","rw");
             RandomAccessFile out = new RandomAccessFile("D:\\file01_copy3.txt","rw
            byte[] buf = new byte[2];
            int hasRead = 0;
            while ((hasRead = in.read(buf)) > 0) {
                //每次读取多少就写多少
                out.write(buf, 0, hasRead);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

通常进行输入输出的内容是文本内容，应该考虑使用字符流；如果是二进制内容，则应考虑使用字节流；RandomAccessFile 支持自由访问文件的任意位置。如果需要访问文件的部分内容，而不是从头读到尾，可以优先考虑 RandomAccessFile，比如文件断点续传。

Java NIO

NIO 也称新 IO 或者非阻塞 IO（Non-Blocking IO）。传统 IO 是面向输入/输出流编程的，而 NIO 是面向通道编程的。

NIO 的 3 个核心概念：Channel、Buffer、Selector。我们先来谈谈其中的两个。

Channel（通道）

Channel 是对 IO 输入/输出系统的抽象，是 IO 源与目标之间的连接通道，NIO 的通道类似于传统 IO 中的各种“流”。与 InputStream 和 OutputStream 不同的是，Channel 是双向的，既可以读，也可以写，且支持异步操作。这契合了操作系统的特性，比如 linux 底层通道就是双向的。此外 Channel 还提供了 map() 方法，通过该方法可以将“一块”数据直接映射到内存中。因此也有人说，NIO 是面向块处理的，而传统 I/O 是面向流处理的。

程序不能直接访问 Channel 中的数据，必须通过 Buffer（缓冲区）作为中介。Channel 可以直接将文件的部分或者全部映射成 Buffer。Channel 是一个接口，有多种实现类，比较常用的是 FileChannel、SocketChannel、ServerSocketChannel、DatagramChannel，分别用于文件读写，TCP 客户端、服务端网络通信、UDP 通信。

Channel 通常都不是通过构造器来创建的，而是通过传统的输入/输出流的 getChannel() 来返回。通过不同类型的 Stream 获得的 Channel 也不同。比如常见的几个 Channel 的获取方式如下：

- FileChannel：由文件流 FileInputStream、FileOutputStream 的 getChannel() 方法返回。
- ServerSocketChannel：由 ServerSocketChannel 的静态方法 open() 返回。
- SocketChannel：由 SocketChannel 的静态方法 open() 返回。

Channel 中读写数据对应的方法分别是 read(ByteBuffer) 和 write(ByteBuffer) 方法。一些 Channel 还提供了 map() 方法将 Channel 对应的部分或全部数据映射为 ByteBuffer（实际的实现类为 MappedByteBuffer）。如果 Channel 对应的数据过大，使用 map() 方法一次性映射到内存会引起性能下降，此时还得用"多次重复取水"的方式处理。

Buffer（缓冲）

Buffer 本质上就是一个容器，其底层持有了一个具体类型的数组来存放具体数据。从 Channel 中取数据或者向 Channel 中写数据都需要通过 Buffer。在 Java 中 Buffer 是一个抽象类，除 boolean 之外的基本数据类型都提供了对应的 Buffer 实现类。比较常用的是 ByteBuffer 和 CharBuffer。

通常 Buffer 的实现类中都没有提供 public 的构造方法，而是提供了静态方法 allocate(int capacity) 用来创建自身对应的 Buffer 对象。使用 get、put 方法来读取、写入数据到 Buffer 中。ByteBuffer 还支持直接缓冲区，即 ByteBuffer 还提供了 allocateDirect(int capacity) 方法来创建直接缓冲区，能与当前操作系统更好地耦合，进一步提高 I/O 的性能。但是分配直接缓冲区的系统开销较大，只适合缓冲区较大且需要长期驻留的情况。

Buffer 中还有两个经常调用的重要方法，即 flip() 和 clear()。flip 方法为从 Buffer 中取出数据做好准备，而 clear 方法为再次向 Buffer 中写入数据做好准备。

NIO 案例

下面通过几个例子演示一下 NIO 的日常操作。

文件复制：

```
public class FileCopy04 {
    public static void main(String[] args) {
        try (FileInputStream fis = new FileInputStream("D:\\file01.txt");
             FileOutputStream fos = new FileOutputStream("D:\\file01_copy4.txt");
             FileChannel inc = fis.getChannel();
             FileChannel outc = fos.getChannel()
        ) {
            ByteBuffer buffer = ByteBuffer.allocate(4);
            //多次重复"取水"的方式
            while (inc.read(buffer) != -1) {
                buffer.flip();
                outc.write(buffer);
                buffer.clear();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

文件复制—映射方式：

```
public class FileCopy05 {
    public static void main(String[] args) {
        File f = new File("D:\\file01.txt");
        try (FileInputStream fis = new FileInputStream(f);
             FileOutputStream fos = new FileOutputStream("D:\\file01_copy5.txt");
             FileChannel inc = fis.getChannel();
             FileChannel outc = fos.getChannel()
        ) {
            //将 FileChannel 里的全部数据映射到 ByteBuffer 中
            MappedByteBuffer mappedByteBuffer = inc.map(FileChannel.MapMode.READ_ON
            outc.write(mappedByteBuffer);
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

文件复制—零拷贝方式

transferFrom 方式：

```
public class FileCopy06 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("D:\\file01.txt");
        FileOutputStream fos = new FileOutputStream("D:\\file01_copy06.txt");
        FileChannel srcChannel = fis.getChannel();
        FileChannel destChannel = fos.getChannel();
        destChannel.transferFrom(srcChannel, 0, srcChannel.size());
        destChannel.close();
        srcChannel.close();
        fis.close();
        fos.close();
    }
}
```

transferTo 方式：

```
public class FileCopy07 {
    public static void main(String[] args) throws IOException {
        FileInputStream fis = new FileInputStream("D:\\file01.txt");
        FileOutputStream fos = new FileOutputStream("D:\\file01_copy07.txt");
        FileChannel srcChannel = fis.getChannel();
        FileChannel destChannel = fos.getChannel();
        long size = srcChannel.size();
        long position = 0;
        while (size > 0) {
            long count = srcChannel.transferTo(position, srcChannel.size(), destCha
            position += count;
            size -= count;
        }

        destChannel.close();
        srcChannel.close();
        fis.close();
        fos.close();
    }
}
```

Java NIO 2.0

JDK 7 对原有的 NIO 进行了改进。第一个改进是提供了全面的文件 I/O 相关 API。第二个改进是增加了异步的基于 Channel 的 IO 机制。

我们说说第一个，第二个也就是通常所说的 AIO（Asynchronous IO），即异步 IO。由于实际工作中不常见，我们就不做介绍了。

原来的 I/O 框架中只有一个 File 类来操作文件，新的 NIO 引入了 Path 接口，代表一个平台无关的路径。并且提供了 Paths、Files 两个强大的工具类来方便文件操作。

使用新的 API 来完成文件复制代码大大简化：

```
public class FileCopy06 {
    public static void main(String[] args) {
        try (OutputStream fos = new FileOutputStream("D:\\file01_copy6.txt")) {
            Files.copy(Paths.get("D:\\file01.txt"), fos);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Java 网络编程

下文通过几个网络 IO 的例子，循序渐进地讲述 Socket 编程的流程。先讲解传统 Socket 编程（阻塞式），再讲解基于 NIO 的 Socket 编程。

传统 Socket 编程

需求描述：实现一个简单的 C/S 架构的客户端/服务端通信程序，分别包括客户端程序和服务端程序。

版本 1：实现客户端/服务端一次性简单通信

这个例子只是简单的“一问一答”模式，极其简单地演示一下 Socket 编程的逻辑。

服务端：

```
public class Server1 {
    public static void main(String[] args) throws IOException {
        //开启一个 TCP 服务端,占用一个本地端口
        ServerSocket serverSocket = new ServerSocket(6666);
        //服务端循环不断地接受客户端的连接
        while (true) {
            Socket socket = null;
```

```
            try {
                //与单个客户端通信的代码放在一个 try 代码块中，单个客户端发生异常（断开）
                System.out.println("server start...");
                //下面这行代码会阻塞,直到有客户端连接
                socket = serverSocket.accept();
                System.out.println("客户端" + socket.getRemoteSocketAddress() + "上线
                //从 Socket 中获得输入输出流,接收和发送数据
                InputStream inputStream = socket.getInputStream();
                OutputStream outputStream = socket.getOutputStream();
                byte[] buf = new byte[1024];
                int len;
                while ((len = inputStream.read(buf)) != -1) {
                    String msg = new String(buf, 0, len);
                    System.out.println("来自客户端的消息: " + msg);
                    String serverResponseMsg = "服务端收到了来自您的消息【" + msg + "
                    //向客户端回写消息
                    outputStream.write(serverResponseMsg.getBytes(StandardCharsets.
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                //当与一个客户端通信结束后，需要关闭对应的 socket
                if (socket != null) {
                    try {
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

客户端

```
public class Client1 {
    public static void main(String[] args) {
        Socket socket = new Socket();
        SocketAddress address = new InetSocketAddress("127.0.0.1", 6666);
        try {
            socket.connect(address, 2000);
            OutputStream outputStream = socket.getOutputStream();
            String clientMsg = "服务端你好! 我是客户端! 你的 IP 是: " + socket.getRemo
            outputStream.write(clientMsg.getBytes(StandardCharsets.UTF_8));

            InputStream inputStream = socket.getInputStream();
            byte[] buf = new byte[1024];
            int len;
            while ((len = inputStream.read(buf)) != -1) {
                String msgFromServer = new String(buf, 0, len);
                System.out.println("来自服务端的消息:" + msgFromServer);
            }
        } catch (IOException e) {
```

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

版本 2：实现客户端可以不断接收用户输入

版本 1 演示了最简单的 Socket 编程，只能实现一次性通信。现在要求客户端能够不断地接收用户输入，多次与服务端通信。服务端代码不变，客户端改造如下：

```
public class Client2 {
    public static void main(String[] args) {
        Socket socket = new Socket();
        SocketAddress address = new InetSocketAddress("127.0.0.1", 6666);
        try {
            socket.connect(address, 2000);
            OutputStream outputStream = socket.getOutputStream();
            InputStream inputStream = socket.getInputStream();

            BufferedReader bufferedReader = new BufferedReader(new InputStreamReade
            String clientMsg;
            System.out.println("请输入消息:");
            while ((clientMsg = bufferedReader.readLine()) != null) {
                outputStream.write(clientMsg.getBytes(StandardCharsets.UTF_8));
                byte[] buf = new byte[1024];
                int readLen = inputStream.read(buf);
                String msgFromServer = new String(buf, 0, readLen);
                System.out.println("来自服务端的消息:" + msgFromServer);
                System.out.println("请输入消息:");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

版本 3：使用字符流包装

上面的版本是按字节方式读取数据的，缓冲字节数组大小无法权衡，太小了不足以存放一行数据时，将会读取到不完整的数据，产生乱码。我们使用字符流包装字节流，读取整行数据，改进如下。

服务端：

```
public class Server3 {
    public static void main(String[] args) throws IOException {
        //开启一个 TCP 服务端,占用一个本地端口
        ServerSocket serverSocket = new ServerSocket(6666);
        //服务端循环不断地接受客户端的连接
        while (true) {
            Socket socket = null;
            try {
                //与单个客户端通信的代码放在一个 try 代码块中，单个客户端发生异常（断开）
                System.out.println("server start...");
                //下面这行代码会阻塞,直到有客户端连接
                socket = serverSocket.accept();
                System.out.println("客户端" + socket.getRemoteSocketAddress() + "上
                //从 Socket 中获得输入输出流,接收和发送数据
                PrintWriter socketPrintWriter = new PrintWriter(socket.getOutputStr
                BufferedReader socketBufferedReader = new BufferedReader(new InputS
                String msg;
                while ((msg = socketBufferedReader.readLine()) != null) {
                    System.out.println("来自客户端的消息: " + msg);
                    String serverResponseMsg = "服务端收到了来自您的消息【" + msg + "
                    socketPrintWriter.println(serverResponseMsg);
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                //当与一个客户端通信结束后，需要关闭对应的 socket
                if (socket != null) {
                    try {
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

客户端：

```
public class Client3 {
    public static void main(String[] args) {
        Socket socket = new Socket();
```

```
        SocketAddress address = new InetSocketAddress("127.0.0.1", 6666);
        try {
            socket.connect(address, 2000);
            PrintWriter socketPrintWriter = new PrintWriter(socket.getOutputStream(
            BufferedReader socketBufferedReader = new BufferedReader(new InputStrea

            BufferedReader bufferedInputReader = new BufferedReader(new InputStream
            String clientMsg;
            System.out.println("请输入消息:");
            while ((clientMsg = bufferedInputReader.readLine()) != null) {
                socketPrintWriter.println(clientMsg);
                String msgFromServer = socketBufferedReader.readLine();
                System.out.println("来自服务端的消息:" + msgFromServer);
                System.out.println("请输入消息:");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

版本 4：实现多客户端与服务器通信

上面的例子中，只能实现一个客户端和服务端的通信。假如有多个客户端连接服务端，就只能等上一个客户端处理完毕，服务端重新通过 accept() 方法从队列中取出连接请求时才能处理。可以使用多线程的方式实现一个服务器同时响应多个客户端。

和上一个版本相比，客户端代码没有改动，服务端改进如下：

```
public class Server4 {
    public static void main(String[] args) throws IOException {
        //开启一个 TCP 服务端,占用一个本地端口
        ServerSocket serverSocket = new ServerSocket(6666);
        //服务端循环不断地接受客户端的连接
        System.out.println("server start...");
        while (true) {
            Socket socket;
            try {
                socket = serverSocket.accept();
                System.out.println("客户端" + socket.getRemoteSocketAddress() + "上线
                //为每一个客户端分配一个线程
                Thread workThread = new Thread(new Handler(socket));
                workThread.start();
            } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }
    }
}

class Handler implements Runnable {
    private Socket socket;

    public Handler(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            //从 Socket 中获得输入输出流,接收和发送数据
            PrintWriter socketPrintWriter = new PrintWriter(socket.getOutputStream(
            BufferedReader socketBufferedReader = new BufferedReader(new InputStrea
            String msg;
            while ((msg = socketBufferedReader.readLine()) != null) {
                System.out.println("来自客户端" + socket.getRemoteSocketAddress() +
                String serverResponseMsg = "服务端收到了来自您的消息【" + msg + "】,并
                socketPrintWriter.println(serverResponseMsg);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //当与一个客户端通信结束后，需要关闭对应的 socket
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

版本 5：实现一个简单的网络聊天室

一个服务端支持多个客户端同时连接，每个客户端都能不断读取用户键入的消息，发送给服务器并由服务器广播到所有连到服务器的客户端，实现群聊的功能。

客户端：

```
public class Client5 {
    public static void main(String[] args) {
        Socket socket = new Socket();
        SocketAddress address = new InetSocketAddress("127.0.0.1", 6666);
        try {
```

```
            socket.connect(address, 2000);
            new Thread(new ClientHandler(socket)).start();
            PrintWriter socketPrintWriter = new PrintWriter(socket.getOutputStream(
            BufferedReader bufferedInputReader = new BufferedReader(new InputStream
            String clientMsg;
            System.out.println("请输入消息:");
            while ((clientMsg = bufferedInputReader.readLine()) != null) {
                socketPrintWriter.println(clientMsg);
                System.out.println("请输入消息:");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    class ClientHandler implements Runnable {
        private Socket socket;

        public ClientHandler(Socket socket) {
            this.socket = socket;
        }

        @Override
        public void run() {
            try {
                BufferedReader socketBufferedReader = new BufferedReader(new InputStrea
                String msgFromServer;
                while ((msgFromServer = socketBufferedReader.readLine()) != null) {
                    System.out.println("收到来自服务端的消息:" + msgFromServer);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

上述客户端单独开启了一个线程来读取服务器响应的数据。主线程只负责接收客户端用户输入的数据，并发送给服务器。

服务端：

```
    public class Server5 {
        public static List<Socket> socketList = new ArrayList<>();
```

```
    public static void main(String[] args) throws IOException {
        //开启一个 TCP 服务端,占用一个本地端口
        ServerSocket serverSocket = new ServerSocket(6666);
        //服务端循环不断地接受客户端的连接
        System.out.println("server start...");
        while (true) {
            Socket socket;
            try {
                socket = serverSocket.accept();
                socketList.add(socket);
                System.out.println("客户端" + socket.getRemoteSocketAddress() + "上线
                //为每一个客户端分配一个线程
                Thread workThread = new Thread(new ServerHandler(socket));
                workThread.start();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

class ServerHandler implements Runnable {
    private Socket socket;
    private BufferedReader socketBufferedReader;

    public ServerHandler(Socket socket) throws IOException {
        this.socket = socket;
        this.socketBufferedReader = new BufferedReader(new InputStreamReader(socket
    }

    @Override
    public void run() {
        try {
            //从 Socket 中获得输入输出流,接收和发送数据
            String msg;
            while ((msg = readMsgFromClient()) != null) {
                System.out.println("收到来自客户端" + socket.getRemoteSocketAddress(
                String massMsg = "客户端【" + socket.getRemoteSocketAddress() + "】说
                for (Socket socket : Server5.socketList) {
                    PrintWriter socketPrintWriter = new PrintWriter(socket.getOutpu
                    socketPrintWriter.println(massMsg);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //当与一个客户端通信结束后，需要关闭对应的 socket
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

```
    private String readMsgFromClient() {
        try {
            return socketBufferedReader.readLine();
        } catch (IOException e) {
            e.printStackTrace();
            //如果捕获到异常，则将该客户端对应的 socket 删除
            Server5.socketList.remove(socket);
        }
        return null;
    }
}
```

上面的代码粗略地实现了一个网络聊天室的功能。使用传统的 IO 编程，比如 BufferedReader 的 readLine() 方法读取数据时，方法成功返回之前线程会被阻塞，因此要能同时处理多个客户端请求的话，服务端需要为每个客户端的 Socket 连接启动一个线程单独处理与单个客户端的通信。同样的，客户端在读取服务端数据时同样会被阻塞，因此需要单独启动一个线程从流中去读取服务端的数据。

版本 6：实现一对一聊天

上一个版本中，聊天室的客户端信息都是群发的，包括发送者也会收到服务器广播的消息。这里再次改进，发送者自己无需收到自己发出去的消息；并且发送者可以指定接受者的名称，实现一对一私聊。实现上述功能的关键就是在 Server 端记录每个客户端的信息。

消息格式约定：

- 客户端发送的消息用冒号分割消息体。比如“消息类型 : 消息接收人(用户名) : 消息内容”。
- 消息类型有两种，login、chat，分别表示登录消息和普通聊天消息；消息接收人可以是 all 或者具体的用户名，分别表示群聊消息和私聊对象。

客户端

客户端连上服务端后先发送登录消息，再发送聊天消息，控制台输入示例如下：

```
请输入消息：
login:zhou
收到来自服务端的消息：用户【zhou】登录成功！
请输入消息：
chat:all:大家好哈
请输入消息：
chat:laowang:老王你好哈
```

代码：

```
public class Client6 {
    public static void main(String[] args) {
        Socket socket = new Socket();
        SocketAddress address = new InetSocketAddress("127.0.0.1", 6666);
        try {
            socket.connect(address, 2000);
            new Thread(new ClientHandler6(socket)).start();
            PrintWriter socketPrintWriter = new PrintWriter(socket.getOutputStream(
            BufferedReader bufferedInputReader = new BufferedReader(new InputStream
            String clientMsg;
            System.out.println("请输入消息:");
            while ((clientMsg = bufferedInputReader.readLine()) != null) {
                socketPrintWriter.println(clientMsg);
                System.out.println("请输入消息:");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (socket != null) {
                try {
                    socket.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

class ClientHandler6 implements Runnable {
    private Socket socket;

    public ClientHandler6(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            BufferedReader socketBufferedReader = new BufferedReader(new InputStrea
            String msgFromServer;
            while ((msgFromServer = socketBufferedReader.readLine()) != null) {
                System.out.println("收到来自服务端的消息:" + msgFromServer);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

服务端：

```
public class Server6 {
```

```
    public static Map<String, Socket> userConnectionInfo = new HashMap<>();

    public static void main(String[] args) throws IOException {
        //开启一个 TCP 服务端,占用一个本地端口
        ServerSocket serverSocket = new ServerSocket(6666);
        //服务端循环不断地接受客户端的连接
        System.out.println("server start...");
        while (true) {
            Socket socket;
            try {
                socket = serverSocket.accept();
                System.out.println("客户端" + socket.getRemoteSocketAddress() + "上线
                //为每一个客户端分配一个线程
                Thread workThread = new Thread(new ServerHandler6(socket));
                workThread.start();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

class ServerHandler6 implements Runnable {
    private Socket socket;
    private BufferedReader socketBufferedReader;

    public ServerHandler6(Socket socket) throws IOException {
        this.socket = socket;
        this.socketBufferedReader = new BufferedReader(new InputStreamReader(socket
    }

    @Override
    public void run() {
        try {
            //从 Socket 中获得输入输出流,接收和发送数据
            String msg;
            while ((msg = readMsgFromClient()) != null) {
                String[] split = msg.split(":");
                if (("login".equals(split[0]) && split.length != 2) || (!"login".eq
                    response("消息格式错误,请用冒号分割,形如: 消息类型:消息接收人(用户名
                    continue;
                }

                String msgType = split[0];
                String userName = split[1];
                if ("login".equals(msgType)) {
                    if (Server6.userConnectionInfo.get(userName) == null) {
                        Server6.userConnectionInfo.put(userName, socket);
                        response("用户【" + userName + "】登录成功! ");
                    } else {
                        response("用户【" + userName + "】已登录,无需重复登录");
                    }
                } else if ("chat".equals(msgType)) {
                    if ("all".equals(userName)) {
                        String senderName = getUname();
                        //群发消息
```

```
                    for (Map.Entry<String, Socket> entry : Server6.userConnecti
                        Socket userSocket = entry.getValue();
                        if (userSocket == socket) {
                            continue;
                        }
                        PrintWriter socketPrintWriter = new PrintWriter(userSoc
                        String sendMsg = "【" + senderName + "】对大家说：" + sp
                        socketPrintWriter.println(sendMsg);
                    }
                } else {
                    if (Server6.userConnectionInfo.get(userName) == null) {
                        response("用户【" + userName + "】不在线");
                    } else {
                        Socket userSocket = Server6.userConnectionInfo.get(user
                        PrintWriter socketPrintWriter = new PrintWriter(userSoc
                        String sendMsg = "【" + getUname() + "】对你说：" + spli
                        socketPrintWriter.println(sendMsg);
                    }
                }
            } else {
                response("消息类型错误，只支持 login/chat");
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        //当与一个客户端通信结束后，需要关闭对应的 socket
        if (socket != null) {
            try {
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

private String getUname() {
    String uname = "";
    //找出该 socket 对应的用户名
    for (Map.Entry<String, Socket> entry : Server6.userConnectionInfo.entrySet(
        String userNameInfo = entry.getKey();
        Socket userSocket = entry.getValue();
        if (userSocket == socket) {
            uname = userNameInfo;
            break;
        }
    }
    return uname;
}

private void response(String msg) throws IOException {
    PrintWriter socketPrintWriter = new PrintWriter(socket.getOutputStream(), t
    socketPrintWriter.println(msg);
}
```

```
    private String readMsgFromClient() {
        try {
            return socketBufferedReader.readLine();
        } catch (IOException e) {
            e.printStackTrace();
            //如果捕获到异常，则将该客户端对应的 socket 删除
            System.out.println("客户端" + socket.getRemoteSocketAddress() + "下线了"
            Server6.userConnectionInfo.remove(getUname());
        }
        return null;
    }
}
```

基于 NIO 的网络编程

讲完了传统的 Java I/O 编程，下面我们讲解如何使用 Java NIO 的方式实现非阻塞的网络通信。

之前的网络通信程序中，我们都是用的传统 I/O 方式，即阻塞式 IO，顾名思义，在程序运行过程中常常会阻塞。比如在前文的例子中，当一个线程执行到 ServerSocket 的 accept() 方法时，该线程会一直阻塞，直到有了客户端连接才从 accept() 方法返回。再比如，当某个线程执行 Socket 的 read() 方法时，如果输入流中没有数据，则该线程会一直阻塞到读入了足够的数据才从 read() 方法返回。

Java NIO（New I/O）提供了非阻塞的实现方式。NIO 也可以理解为 non-blocking I/O 的简称。所谓非阻塞 I/O，就是当线程执行这些 I/O 方法时，如果某个操作还没有准备好，就立即返回，而不会因为某个操作还没就绪就进入线程阻塞状态，一直在那等。比如，当服务端的线程接收客户端连接时，如果没有客户端连接，就立即返回。再比如，当某个线程从输入流中读取数据时，如果流中还没有数据，就立即返回。或者如果输入流中没有足够的数据就直接读取现有的数据并返回。很明显，这种非阻塞的方式效率会更高。

有人说，那我用多线程方式处理阻塞式通信不香么？

是的，确实不香！前面我们的服务端代码演示了如何使用多线程同时处理多个客户端的连接。通常是主线程负责接收客户端的连接，每当主线程接收到一个客户端连接后，就把具体的数据交互任务交给一个单独的线程去完成，主线程继续接收下一个客户端的连接。

尽管使用多线程能够满足同时相应多个客户端的要求，但是这种方式有下列局限性：

- 如果服务端对于每个客户端的连接请求都单独开启一个线程来处理，那么在客户端数量庞大时，势必导致服务端开启的线程数过多。即使是使用线程池，也得设置池中放多少个线程，放多放少都是个问题。我们知道，JVM 会为每个线程分配一个 Java 虚拟机栈，线程越多，系统开销就越大，线程的调度负担就越重，甚至会由于线程同步的复杂性导致线程死锁。

- 负责读写数据的工作线程很多时间浪费在 I/O 阻塞中，因为要等流中的数据准备好。这就会导致 JVM 频繁转让 CPU 的使用权，让阻塞状态的线程放弃 CPU，让可运行状态的线程获得 CPU 使用权。

实践经验告诉我们，工作线程并不是越多越好。保持适当的线程数，可以提高服务器的并发性能。但是当线程数到达某个阈值，超出系统负荷，反而会导致并发性能降低，增大响应时间。Java NIO 可以做到用一个线程来处理多个 I/O 操作，再也不要来一个客户端分配一个线程了，比如来 10000 个并发连接，可以只分配 1 个、50 个或者 100 个线程来处理。

Java NIO 提供了支持阻塞/非阻塞 I/O 通信的类。下面介绍几个核心的类。

1. ServerSocketChannel

可以看成是 ServerSocket 的替代类，既支持非阻塞通信，也支持阻塞式通信，同时也有负责接收客户端连接的 accept() 方法。每一个 ServerSocketChannel 对象都和一个 ServerSocket 对象关联。前面提到了，ServerSocketChannel 没有 public 的构造器，只能通过它自身的静态方法 open() 来创建 ServerSocketChannel 对象。ServerSocketChannel 是 SelectableChannel 的派生类。

2. SocketChannel

可以看成是 Socket 的替代类，既支持非阻塞通信，又支持阻塞式通信。SocketChannel 具有读数据的 read(ByteBuffer dst) 方法和写数据的 write(ByteBuffer src) 方法。SocketChannel 也没有 public 类型的构造器，也是通过静态方法 open() 来创建自身的对象。每一个 SocketChannel 对象都和一个 Socket 对象关联。SocketChannel 也是 SelectableChannel 的派生类。

SocketChannel 提供了发送和接收数据的方法。

- read(ByteBuffer dst)：接收数据，并把接收到的数据存到指定的 ByteBuffer 中。假设 ByteBuffer 的剩余容量为 n，在阻塞模式下，read() 方法会争取读入 n 个字节，如果通道中不足 n 个字节，就会阻塞，直到读入了 n 个字节或者读到了输入流的末尾，或者出现了 I/O 异常。在非阻塞模式下，read() 方法奉行能读多少就读多少的原则。不会等待数据，而是读取之后立即返回。可能读取了不足 n 个字节的数据，也可能就是 0。如果返回 -1 则表示读到了流的末尾。
- write(ByteBuffer src)：发送数据，即把指定的 ByteBuffer 中的数据发送出去。假设 ByteBuffer 的剩余容量为 n，在阻塞模式下，write() 方法会争取输出 n 个字节，如果底层的网络输出缓冲区不能容纳 n 个字节，就会进入阻塞状态，直到输出 n 个字节，或者出现 I/O 异常才返回。在非阻塞模式下，write() 方法奉行能输出多少就输出多少的原则，有可能不足 n 个字节，有可能是 0 个字节，总之立即返回。

3. Selector（选择器）

用一个线程就能处理多个的客户端连接的关键就在于 Selector。Selector 是 SelectableChannel 对象的多路复用器，用于判断 channle 上是否发生 IO 事件，所有希望使用非阻塞方式通信的 Channel 都需要注册到 Selector 上。Selector 可以同时监控多个 SelectableChannel 的 IO 状态，即只要 ServerSocketChannel 或者 SocketChannel 向 Selector 注册了特定的事件，Selector 就会监控这些事件是否发生。Selector 为 ServerSocketChannel 监听连接就绪的事件，为 SocketChannel 监控连接就绪、读就绪、写就绪事件。Selector 实例对象的创建通常是通过调用其静态的 open() 方法。

Selector 有如下几种方法来返回 I/O 相关事件已经发生的 SelectionKey 的数目。

- selectNow()：该方法使用非阻塞的方式返回相关事件已经发生的 SelectionKey 的数目，如果没有任何事件发生，立即返回 0。
- select() 和 select(long timeout)：该方法使用阻塞的方式。如果没有一个事件发生，就进入阻塞状态。直到有事件发生或者超出 timeout 设置的等待时间，才会正常返回。

使用 Selector 能够保证只在真正有读写事件发生时，才会进行读写，若通道中没有数据可用，该线程可以执行其它任务，不必阻塞。比如一个通道没有准备好数据时，可以将空闲时间用于其它通道执行 IO 操作。由于单个线程可以管理多个 Channel 的输入输出，避免了频繁的线程切换和阻塞，提升了 I/O 效率。实际上 Netty 的 I/O 线程 NioEventLoop 就是聚合了 Selector（多路复用器），因此能够处理成千上万的客户端连接。

4. SelectionKey

ServerSocketChannel 或者 SocketChannel 通过 register() 方法向 Selector 注册事件时，会返回一个 SelectionKey 对象，用来跟踪注册事件。Selector 会一直监控与 SelectionKey 相关的事件。当一个 SelectionKey 对象被放到 Selector 对象的 selected-keys 集合中时，就表示与这个 SelectionKey 相关的事件发生了。

ServerSocketChannel 及 SocketChannel 都继承自 SelectableChannel 类，该类及其子类可以委托 Selector 来监控它们可能发生的一些事件，这种委托过程就是事件注册。比如下列代码展示了 ServerSocketChannel 向 Selector 注册接收连接事件。

```
serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT)
```

ServerSocketChannel 只会发生一种事件，即 SelectionKey.OP_ACCEPT，接受连接就绪事件。该事件的发生表明至少有一个客户端连接了，服务端可以通过 accept() 去接受这个连接了。

SocketChannel 可以发生下列 3 种事件。

- SelectionKey.OP_CONNECT，连接就绪事件，表示客户端和服务端已经成功建立连接。
- SelectionKey.OP_READ，读就绪事件，表示通道中已经有了可读的数据，可以执行读操作了。
- SelectionKey.OP_WRITE，写就绪事件，表示可以向通道中写数据了。

默认情况下，所有的 Channel 都是阻塞模式的，要想使用非阻塞模式，可以通过下列方式设置：

```
serverSocketChannel.configureBlocking(false);
```

此外，前面已经介绍了 NIO 的 Buffer、Channel 相关概念，此处不再赘述。

下面我们就使用 NIO 的方式来编写网络通信程序的案例。

需求描述：实现客户端服务端的网络通信，客户端每发送一条消息，服务端就原样回复，并加一句前缀以示区分。

版本 1：使用 NIO 的阻塞模式，并配以线程池方式

服务端：

```
public class NIOServer1 {
    private int port = 6666;
    private ServerSocketChannel serverSocketChannel;
    private ExecutorService executorService;

    public NIOServer1() throws IOException {
        executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availab
        serverSocketChannel = ServerSocketChannel.open();
        //允许地址重用，即关闭了服务端程序之后，哪怕立即再启动该程序时可以顺利绑定相同的端
        serverSocketChannel.socket().setReuseAddress(true);
        serverSocketChannel.socket().bind(new InetSocketAddress(port));
        System.out.println("server started...");
    }

    public static void main(String[] args) throws IOException {
        new NIOServer1().service();
    }

    private void service() {
        while (true) {
            SocketChannel socketChannel;
            try {
```

```
                socketChannel = serverSocketChannel.accept();
                executorService.execute(new NioHandler1(socketChannel));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

class NioHandler1 implements Runnable {
    private SocketChannel socketChannel;

    public NioHandler1(SocketChannel socketChannel) {
        this.socketChannel = socketChannel;
    }

    @Override
    public void run() {
        Socket socket = socketChannel.socket();
        System.out.println("接受到客户端的连接，来自" + socket.getRemoteSocketAddress
        try {
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket
            PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
            String msg;
            while ((msg = reader.readLine()) != null) {
                System.out.println("客户端【" + socket.getInetAddress() + ":" + sock
                writer.println(genResponse(msg));
                if ("bye".equals(msg)) {
                    break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private String genResponse(String msg) {
        return "服务器收到了您的消息：" + msg;
    }
}
```

客户端:

```
public class NIOClient1 {
    private SocketChannel socketChannel;

    public NIOClient1() throws IOException {
        socketChannel = SocketChannel.open();
        InetAddress localHost = InetAddress.getLocalHost();
        InetSocketAddress socketAddress = new InetSocketAddress(localHost, 6666);
        //采用阻塞模式连接服务器
        socketChannel.connect(socketAddress);
        System.out.println("与服务端连接成功！");
    }
```

```
    public static void main(String[] args) throws IOException {
        new NIOClient1().chat();
    }

    public void chat() {
        Socket socket = socketChannel.socket();
        try {
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket
            PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader inputReader = new BufferedReader(new InputStreamReader(S
            String msg;
            while ((msg = inputReader.readLine()) != null) {
                writer.println(msg);
                System.out.println("【服务器】说:" + reader.readLine());
                //如果输入 bye，则终止聊天
                if ("bye".equals(msg)) {
                    break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

上述案例中，我们使用的是 ServerSocketChannel 和 SocketChannel 的默认模式，即阻塞模式。为了能同时响应多个客户端，服务端依然是使用多线程的方式，只不过这次使用的是线程池。

版本 2：使用 NIO 非阻塞模式

在非阻塞模式下，服务端只需启动一个主线程，就能同时完成 3 件事：

- 接受客户端的连接
- 接收客户端发送的数据
- 向客户端发送响应数据

服务端会委托 Selector 来监听接收连接就绪事件、读就绪事件、写就绪事件，如有特定的事件发生，就处理该事件。

服务端：

```
public class NIOServer2 {
    private int port = 6666;
    private ServerSocketChannel serverSocketChannel;
    private Selector selector;
    private Charset charset = Charset.forName("UTF-8");
```

```
public NIOServer2() throws IOException {
    selector = Selector.open();
    serverSocketChannel = ServerSocketChannel.open();
    serverSocketChannel.socket().setReuseAddress(true);
    //设置为非阻塞模式
    serverSocketChannel.configureBlocking(false);
    serverSocketChannel.socket().bind(new InetSocketAddress(port));
    System.out.println("server started...");
}

public static void main(String[] args) throws IOException {
    new NIOServer2().service();
}

private void service() throws IOException {
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
    while (selector.select() > 0) {
        Set<SelectionKey> selectionKeys = selector.selectedKeys();
        Iterator<SelectionKey> iterator = selectionKeys.iterator();
        while (iterator.hasNext()) {
            SelectionKey key = null;
            //处理每个 SelectionKey 的代码放在一个 try/catch 块中，如果出现异常，就
            try {
                key = iterator.next();
                if (key.isAcceptable()) {
                    doAccept(key);
                }
                if (key.isWritable()) {
                    sendMsg(key);
                }

                if (key.isReadable()) {
                    receiveMsg(key);
                }
                //从 Selector 的 selected-keys 集合中删除处理过的 SelectionKey
                iterator.remove();
            } catch (Exception e) {
                e.printStackTrace();
                try {
                    //发生异常时，使这个 SelectionKey 失效，Selector 不再监控这个
                    if (key != null) {
                        key.cancel();
                        //关闭这个 SelectionKey 关联的 SocketChannel
                        key.channel().close();
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
    }
}

private void receiveMsg(SelectionKey key) throws IOException {
    ByteBuffer buffer = (ByteBuffer) key.attachment();
```

```
        SocketChannel socketChannel = (SocketChannel) key.channel();
        //创建一个 ByteBuffer 存放读取到的数据
        ByteBuffer readBuffer = ByteBuffer.allocate(64);
        socketChannel.read(readBuffer);
        readBuffer.flip();
        buffer.limit(buffer.capacity());
        //把 readBuffer 中的数据拷贝到 buffer 中，假设 buffer 的容量足够大，不会出现溢出
        //在非阻塞模式下，socketChannel.read(readBuffer)方法一次读入多少字节的数据是不确
        //因此需要将其每次读取的数据放到 buffer 中，当凑到一行数据时再回复客户端
        buffer.put(readBuffer);
    }

    private void sendMsg(SelectionKey key) throws IOException {
        ByteBuffer buffer = (ByteBuffer) key.attachment();
        SocketChannel socketChannel = (SocketChannel) key.channel();
        buffer.flip();
        String data = decode(buffer);
        //当凑满一行数据时再回复客户端
        if (data.indexOf("\r\n") == -1) {
            return;
        }
        //读取一行数据
        String recvData = data.substring(0, data.indexOf("\n") + 1);
        System.out.print("客户端【" + socketChannel.socket().getInetAddress() + ":"
        ByteBuffer outputBuffer = encode(genResponse(recvData));
        while (outputBuffer.hasRemaining()) {
            socketChannel.write(outputBuffer);
        }

        ByteBuffer temp = encode(recvData);
        buffer.position(temp.limit());
        //删除 buffer 中已经处理过的数据
        buffer.compact();

        if ("bye\r\n".equals(recvData)) {
            key.cancel();
            key.channel().close();
            System.out.println("关闭与客户端" + socketChannel.socket().getRemoteSock
        }
    }

    private ByteBuffer encode(String msg) {
        return charset.encode(msg);//转为字节
    }

    private String decode(ByteBuffer buffer) {
        CharBuffer charBuffer = charset.decode(buffer);//转为字符
        return charBuffer.toString();
    }

    private void doAccept(SelectionKey key) throws IOException {
        ServerSocketChannel ssc = (ServerSocketChannel) key.channel();
        SocketChannel socketChannel = ssc.accept();
        System.out.println("接受到客户端的连接，来自" + socketChannel.socket().getRem
        //设置为非阻塞模式
        socketChannel.configureBlocking(false);
```

```
            //创建一个用于接收客户端数据的缓冲区
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            //向 Selector 注册读、写就绪事件,并关联一个 buffer 附件
            socketChannel.register(selector, SelectionKey.OP_WRITE | SelectionKey.OP_RE
        }

        private String genResponse(String msg) {
            return "服务器收到了您的消息：" + msg;
        }
    }
```

上述例子中，服务端使用一个线程就完成了连接接收、数据接收、数据发送的功能。假设有许多的客户端连接，并且每此与客户端的数据交互都很多，势必会影响服务器的响应效率。如果把接收客户端连接的操作单独由一个线程处理，把接收数据和发送数据的操作交给另外的线程完成，就可以提高服务器的并发性能。读者可以尝试自己来实现一个主从线程模式的服务端程序，欢迎在在评论区留言哦！

下面再来看看客户端的实现。客户端和服务端的通信按照它们接收数据和发送数据的协调程度来区分可以分为同步通信和异步通信。比如前面我们演示的传统阻塞式 IO 案例版本 2 就是同步通信，即每次客户端发送一行消息后，必须等到收到了服务端的响应数据后才能再发送下一行数据。而异步通信指的是数据的发送操作和接收操作互不影响，各自独立进行。异步通信使用非阻塞方式更容易实现。

比如下面这个 NIOClient2 类就是采用非阻塞方式来实现异步通信。在 NIOClient2 中定义了两个 ByteBuffer：recvBuf 和 sendBuf。NIOClient2 把用户从控制台输入的数据存放到 sendBuf 中，并将 sendBuf 中的数据发给服务器。把从服务器接收到的数据放在 recvBuf 中，并打印到控制台。由于接收用户控制台输入的线程和发送数据给服务器的线程都会使用 sendBuf，因此加了 synchronized 进行同步。

客户端：

```
public class NIOClient2 {
    private ByteBuffer recvBuf = ByteBuffer.allocate(1024);
    private ByteBuffer sendBuf = ByteBuffer.allocate(1024);
    private Charset charset = Charset.forName("UTF-8");
    private SocketChannel socketChannel;
    private Selector selector;

    public NIOClient2() throws IOException {
        socketChannel = SocketChannel.open();
        InetAddress localHost = InetAddress.getLocalHost();
        InetSocketAddress socketAddress = new InetSocketAddress(localHost, 6666);
        //采用阻塞模式连接服务器
        socketChannel.connect(socketAddress);
        //设置为非阻塞模式
        socketChannel.configureBlocking(false);
        System.out.println("与服务端连接成功！");
```

```
        selector = Selector.open();
    }

    public static void main(String[] args) throws IOException {
        NIOClient2 nioClient2 = new NIOClient2();
        Thread inputThread = new Thread() {
            @Override
            public void run() {
                nioClient2.receiveInput();
            }
        };

        inputThread.start();
        nioClient2.chat();
    }

    private void chat() throws IOException {
        //接收和发送数据
        socketChannel.register(selector, SelectionKey.OP_WRITE | SelectionKey.OP_RE
        while (selector.select() > 0) {
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = null;
                try {
                    key = iterator.next();
                    iterator.remove();
                    if (key.isWritable()) {
                        sendMsg(key);
                    }

                    if (key.isReadable()) {
                        receiveMsg(key);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                    try {
                        //发生异常时，使这个 SelectionKey 失效，Selector 不再监控这个
                        if (key != null) {
                            key.cancel();
                            //关闭这个 SelectionKey 关联的 SocketChannel
                            key.channel().close();
                        }
                    } catch (Exception ex) {
                        ex.printStackTrace();
                    }
                }
            }
        }
    }

    private void receiveMsg(SelectionKey key) throws IOException {
        //接收服务端发来的数据，放到 recvBuf 中，如满一行数据，就输出，然后从 recvBuf 中
        SocketChannel channel = (SocketChannel) key.channel();
        channel.read(recvBuf);
        recvBuf.flip();
```

```
        String recvMsg = decode(recvBuf);
        if (recvMsg.indexOf("\n") == -1) {
            return;
        }
        String recvMsgLine = recvMsg.substring(0, recvMsg.indexOf("\n") + 1);
        System.out.print("【服务器】说:" + recvMsgLine);
        if (recvMsgLine.contains("bye")) {
            key.cancel();
            socketChannel.close();
            System.out.println("与服务器断开连接");
            selector.close();
            System.exit(0);
        }

        ByteBuffer temp = encode(recvMsgLine);
        recvBuf.position(temp.limit());
        //删除已经输出的数据
        recvBuf.compact();
    }

    private void sendMsg(SelectionKey key) throws IOException {
        //发送 sendBuf 中的数据
        SocketChannel channel = (SocketChannel) key.channel();
        synchronized (sendBuf) {
            //为取出数据做好准备
            sendBuf.flip();
            //将 sendBuf 中的数据写入到 Channel 中去
            channel.write(sendBuf);
            //删除已经发送的数据(通过压缩的方式)
            sendBuf.compact();
        }
    }

    private void receiveInput() {
        try {
            BufferedReader inputReader = new BufferedReader(new InputStreamReader(S
            String msg;
            while ((msg = inputReader.readLine()) != null) {
                synchronized (sendBuf) {
                    sendBuf.put(encode(msg + "\r\n"));
                }
                //如果输入 bye，则终止聊天
                if ("bye".equals(msg)) {
                    break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private ByteBuffer encode(String msg) {
        return charset.encode(msg);//转为字节
    }

    private String decode(ByteBuffer buffer) {
```

```
        CharBuffer charBuffer = charset.decode(buffer);//转为字符
        return charBuffer.toString();
    }
}
```

版本 3：基于 NIO 重写网络聊天室的案例

我们对照传统 IO 方式的实现的简单网络聊天室，使用 NIO 来实现同样的功能。传统方式请参照前文的 Server5/Client5。

服务端

```
public class NIOServer3 {
    private int port = 6666;
    private ServerSocketChannel serverSocketChannel;
    private Selector selector;
    private Charset charset = Charset.forName("UTF-8");

    public NIOServer3() throws IOException {
        selector = Selector.open();
        serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().setReuseAddress(true);
        //设置为非阻塞模式
        serverSocketChannel.configureBlocking(false);
        serverSocketChannel.socket().bind(new InetSocketAddress(port));
        System.out.println("server started...");
    }

    public static void main(String[] args) throws IOException {
        new NIOServer3().service();
    }

    private void service() throws IOException {
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        while (selector.select() > 0) {
            for (SelectionKey key : selector.selectedKeys()) {
                selector.selectedKeys().remove(key);
                if (key.isAcceptable()) {
                    ServerSocketChannel ssc = (ServerSocketChannel) key.channel();
                    SocketChannel socketChannel = ssc.accept();
                    System.out.println("接受到客户端的连接，来自" + socketChannel.soc
                    //设置为非阻塞模式
                    socketChannel.configureBlocking(false);
                    socketChannel.register(selector, SelectionKey.OP_READ);
                }

                if (key.isReadable()) {
                    SocketChannel sc = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(1024);
                    String msg = "";
                    try {
                        while (sc.read(buffer) > 0) {
```

```
                            buffer.flip();
                            msg += charset.decode(buffer);
                        }
                        System.out.println("客户端【" + sc.getRemoteAddress() + "】i
                    } catch (IOException e) {
                        e.printStackTrace();
                        try {
                            //对某个 Client 对应的 Channel 读写发生异常时，使这个 Sele
                            if (key != null) {
                                key.cancel();
                                //关闭这个 SelectionKey 关联的 SocketChannel
                                System.out.println("客户端【" + ((SocketChannel) key
                                key.channel().close();
                            }
                        } catch (Exception ex) {
                            ex.printStackTrace();
                        }
                    }

                    if (msg.length() > 0) {
                        for (SelectionKey selectedKey : selector.keys()) {
                            Channel channel = selectedKey.channel();
                            //遍历 Selector 中的所有注册的 Channel，如果是客户端的 Soc
                            if (channel instanceof SocketChannel && channel != sc)
                                SocketChannel socketChannel = (SocketChannel) chann
                                socketChannel.write(charset.encode("用户【" + sc.get
                            }
                        }
                    }
                }
            }
        }
    }
}
```

客户端:

```
public class NIOClient3 {
    private ByteBuffer recvBuf = ByteBuffer.allocate(1024);
    private ByteBuffer sendBuf = ByteBuffer.allocate(1024);
    private Charset charset = Charset.forName("UTF-8");
    private SocketChannel socketChannel;
    private Selector selector;

    public NIOClient3() throws IOException {
        socketChannel = SocketChannel.open();
        InetAddress localHost = InetAddress.getLocalHost();
        InetSocketAddress socketAddress = new InetSocketAddress(localHost, 6666);
        //采用阻塞模式连接服务器
        socketChannel.connect(socketAddress);
        //设置为非阻塞模式
        socketChannel.configureBlocking(false);
        System.out.println("与服务端连接成功！");
        selector = Selector.open();
```

```
}

public static void main(String[] args) throws IOException {
    NIOClient3 nioClient3 = new NIOClient3();
    Thread inputThread = new Thread() {
        @Override
        public void run() {
            nioClient3.sendInputMsg();
        }
    };

    inputThread.start();
    nioClient3.receiveMsg();
}

private void receiveMsg() throws IOException {
    socketChannel.register(selector, SelectionKey.OP_READ);
    while (selector.select() > 0) {
        for (SelectionKey key : selector.selectedKeys()) {
            try {
                selector.selectedKeys().remove(key);
                if (key.isReadable()) {
                    SocketChannel sc = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(1024);
                    String msg = "";
                    while (sc.read(buffer) > 0) {
                        buffer.flip();
                        msg += charset.decode(buffer);
                    }
                    System.out.println(msg);
                }
            } catch (IOException e) {
                e.printStackTrace();
                try {
                    //发生异常时，使这个 SelectionKey 失效，Selector 不再监控这个
                    if (key != null) {
                        key.cancel();
                        //关闭这个 SelectionKey 关联的 SocketChannel
                        key.channel().close();
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }
    }
}

private void sendInputMsg() {
    //接收键盘输入的消息并发送数据到服务器
    try {
        BufferedReader inputReader = new BufferedReader(new InputStreamReader(S
        String msg;
        while ((msg = inputReader.readLine()) != null) {
            socketChannel.write(charset.encode(msg));
        }
```

```
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

相比传统的 I/O，基于 NIO 的 socket 编程复杂度提高了很多，这也是我们学习 Netty 的原因之一——简化网络编程。

上面我们以网络通信的例子展示了传统的阻塞式 IO 和新的非阻塞式 IO 的区别，相信通过多个实际的代码例子，能让大家有个直观的感受，有效复习了一下 Java 的 IO 体系。在介绍 Netty 这款封装了 Java NIO 的框架之前，我们稍安勿躁，先补充一下 NIO 相关的理论知识。

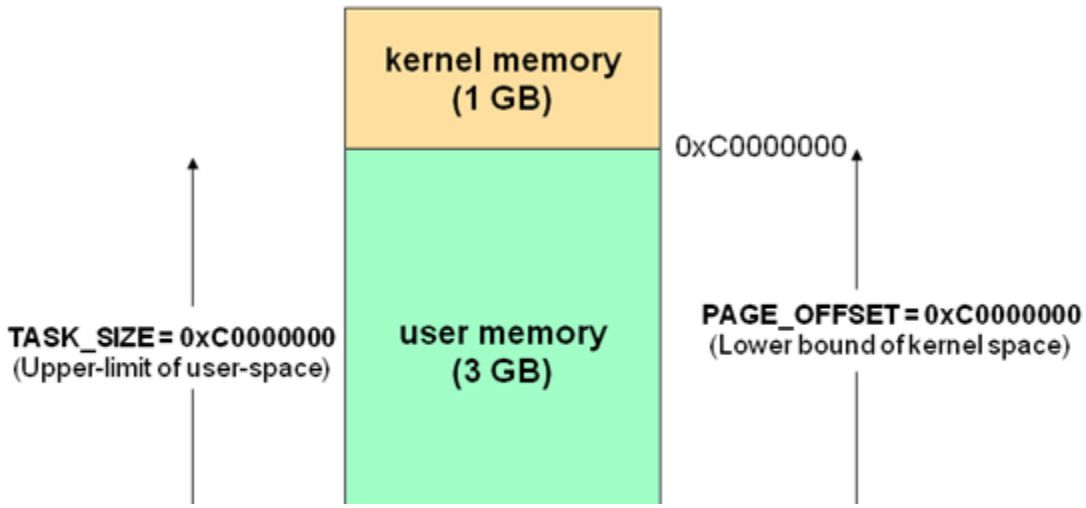
NIO 的理论基础

用户空间和内核空间

通俗地讲，内核空间（kernel space）是操作系统内核才能访问的区域，是受保护的内存区域，普通应用程序不能访问。而用户空间（user space）则是普通应用程序访问的内存空间。用户空间和内核空间概念的由来和 CPU 的发展有很大关系。在 CPU 的保护模式下，系统需要保护 CPU 赖以运行的资料；为了保证操作系统内核资料，需要把内存空间进行划分为 OS 内核运行的空间和普通应用程序运行的空间，两者不能越界。所谓的空间就是内存地址。操作系统为了保护自己不被普通应用程序破坏，对内核空间进行了一些约束，比如访问权限、页的换入换出，优先级等。

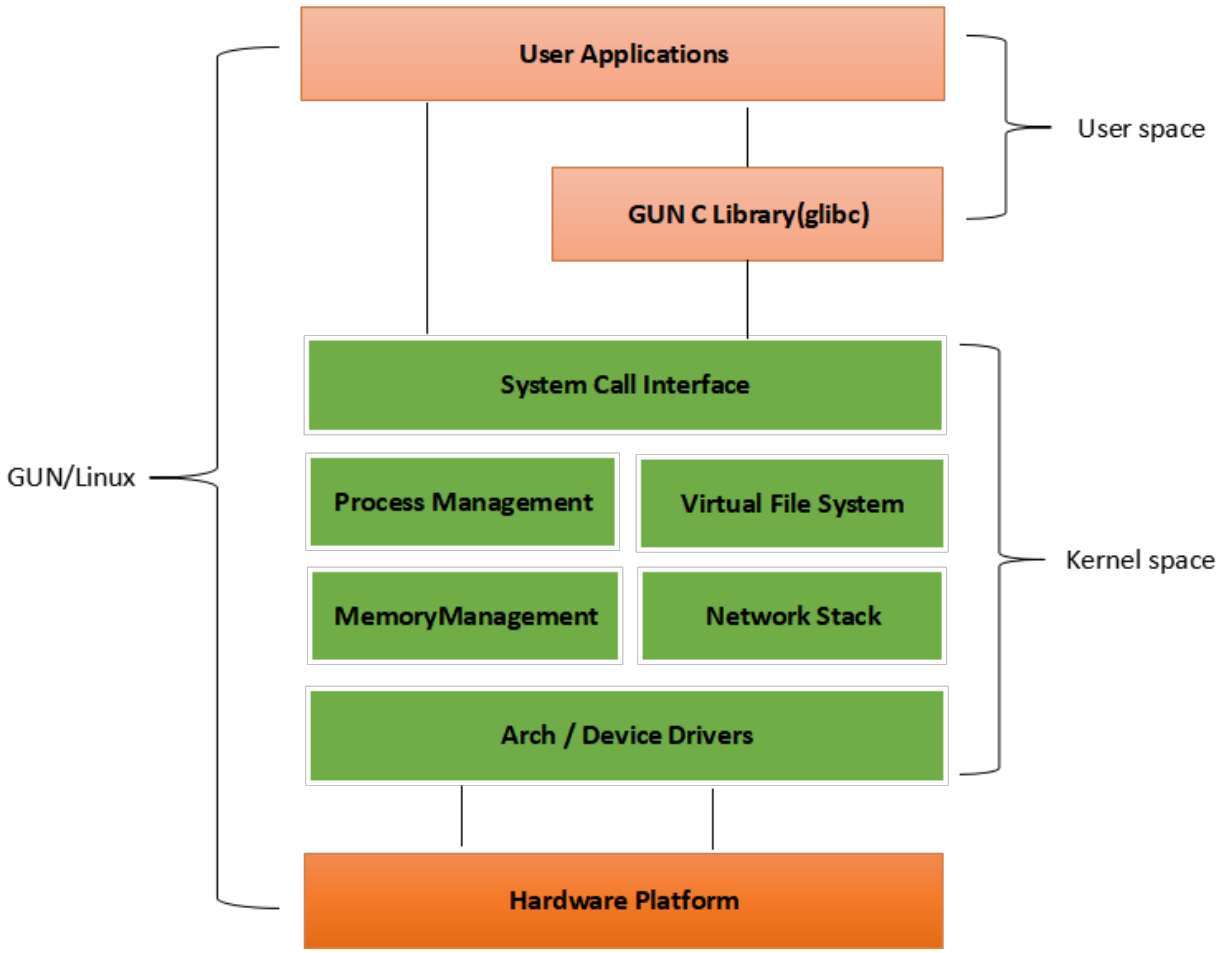
目前的操作系统都是采用虚拟存储器。因此内核空间和用户空间都是指的虚拟空间，也就是虚拟地址。

比如对于 32 位的 Linux 系统而言，用户空间和内核空间划分如下：



32 位操作系统的寻址空间（虚拟地址空间）为 4G（2 的 32 次方）。在 Linux 中，4G 虚拟地址空间中的最高的 1G 字节空间分配给内核独享使用。低地址的 3G 空间为应用程序共享，即每个应用程序都有最大 3G 的虚拟地址空间。每个进程可以通过系统调用切换进入内核，所有进程可以共享 Linux 内核。因此可以认为每个进程都有 4G 字节的虚拟空间。

Linux 内部结构图如下：



Linux 的五种 I/O 模型

众所周知，出于对 OS 安全性的考虑，用户进程是不能直接操作 I/O 设备的。必须通过系统调用请求操作系统内核来协助完成 I/O 动作。

下图展示了 Linux I/O 的过程。

将数据从内核缓冲区
拷贝到用户进程空间

从 I/O 设备获取
数据到内核缓冲区

操作系统内核收到用户进程发起的请求后，从 I/O 设备读取数据到 kernel buffer 中，再将 buffer 中的数据拷贝到用户进程的地址空间，用户进程获取到数据后返回给客户端。

在 I/O 过程中，对于输入操作通常有两个不同的阶段：

- 等待数据准备好
- 将数据从内核缓冲区拷贝到用户进程

根据这两个阶段等待方式的不同，可以将 Linux I/O 分为 5 种模式：

- blocking I/O，阻塞式 I/O
- nonblocking I/O，非阻塞式 I/O
- I/O multiplexing（select and poll），I/O 多路复用
- signal driven I/O（SIGIO），信号驱动 I/O
- asynchronous I/O（the POSIX aio_functions），异步 I/O

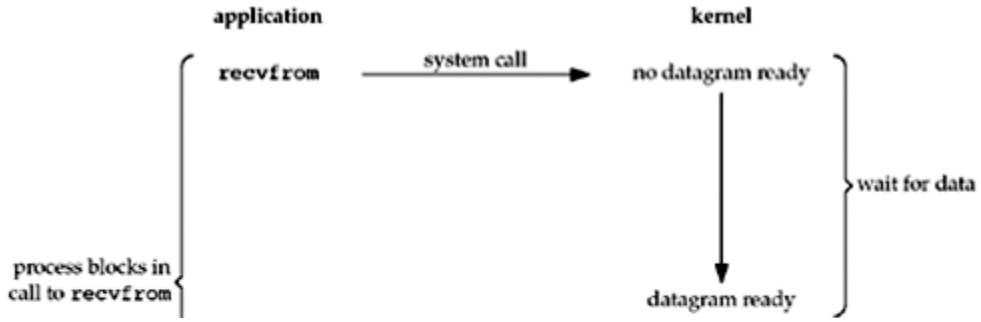
对于 Socket 上的输入操作，第 1 步通常是等待网络上的数据到达。当数据包到达时，它被复制到内核的缓冲区中。第 2 步是从内核缓冲区复制数据到应用程序缓冲区。

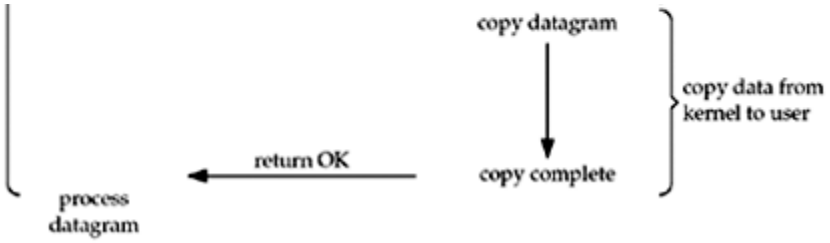
下面详细介绍 Linux 中的 5 种 I/O 模式。

1. Blocking I/O

默认情况下，所有的 Socket 都是阻塞式的。下图展示了一个基于 UDP 的网络数据获取流程。

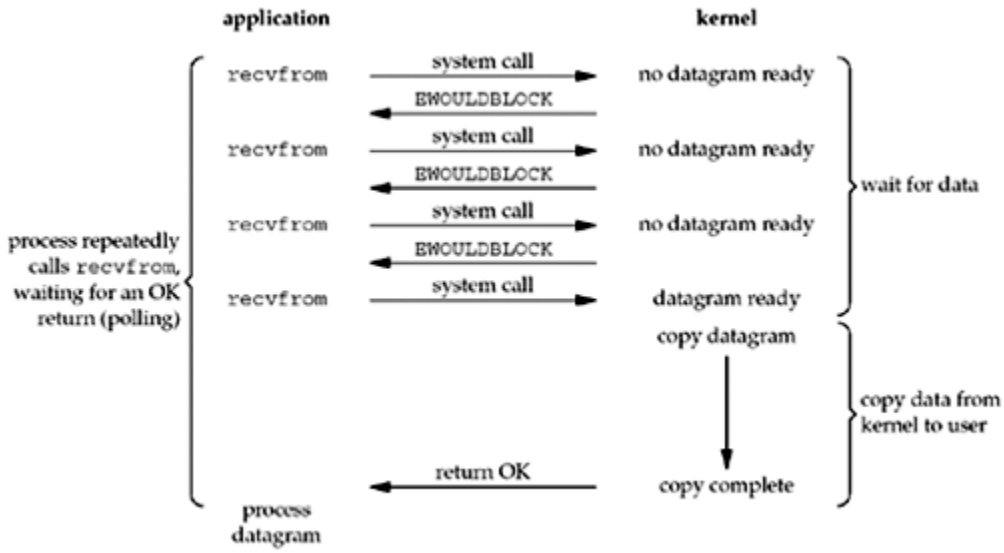
用户进程调用了 recvfrom 系统调用，此后一直处于等待状态，直到数据包到达并被拷贝到应用程序缓冲区，或者发生 error 才返回。整个过程从开始 recvfrom 调用到它返回一直处于阻塞状态。当 recvfrom 调用返回后，应用进程才能处理数据。





2. Nonblocking I/O

可以设置 Socket 为非阻塞模式。这种设置相当于告诉内核“当 I/O 操作时，如果请求是不可能完成的，不要把进程进入睡眠状态，返回一个错误即可“。下图展示了整个流程：在前三次调用 recvfrom 系统调用时，没有就绪的数据返回，所以内核立即返回 EWOULDBLOCK 错误。第四次调用 recvfrom 时，数据报已经准备好，它被复制到应用程序缓冲区中，然后 recvfrom 成功返回。最后应用进程对数据进行处理。当应用程序在一个非阻塞描述符上循环调用 recvfrom 系统调用时，这种方式也被称为轮询。应用程序不断轮询内核，以查看是否有某些操作准备好了。很明显，这通常会浪费 CPU 时间，但这种模式偶尔也会被使用。通常在专门用于一个功能的系统上使用。

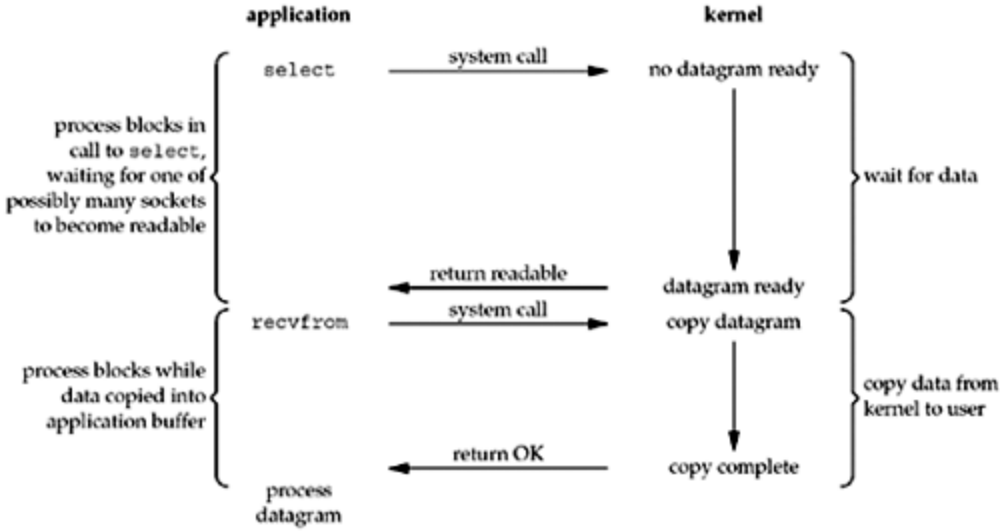


3. I/O Multiplexing

I/O 多路复用通常使用 select 或者 poll 系统调用。这种方式下的阻塞只是被 select 或者 poll 这两个系统调用阻塞，而不会阻塞实际的 I/O 系统调用（即数据输入、输出不会被阻塞）。下图展示了整个过程。当调用 select 时，应用进程被阻塞。同时，系统内核会“监视”所有 select 负责的 Socket。只要其中有 1 个 Socket 的数据准备好了，select 调用就返回。然后调用 recvfrom 将数据报复制到应用程序缓冲区，最后返回给用户进程。

乍一看，这种方式和 blocking I/O 相比似乎更差，因为整个过程产生了 2 次系统调用，select 和 recvfrom。但是使用 select 的好处是可以同时等待多个描述符准备好。换句话说可以同时“聆听”多个 Socket 通道，同时处理多个连接。select 的优势不是对于单个连

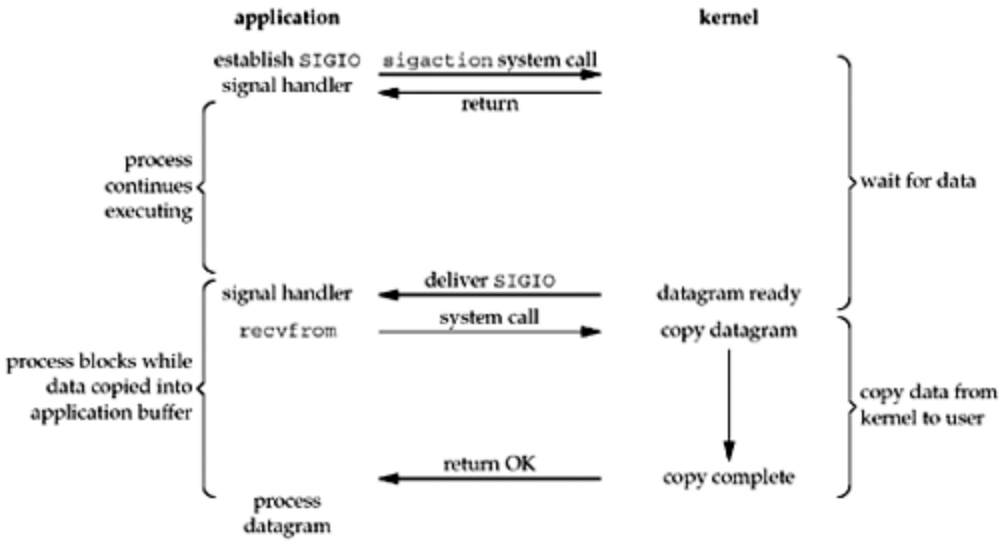
接处理得更快，而是能同时处理更多的连接。这和多线程阻塞式 I/O 有点类似。只不过后者是使用多个线程（每个文件描述符对应一个线程）来处理 I/O，每个线程都可以自由地调用阻塞式系统调用，比如 recvfrom。我们知道线程多了会带来上下文切换的开销，因此未必优于 select 方式。在前面 Java NIO 的例子中，我们已经体会到了 selector 带来的性能提升。



Linux 内核将所有外部设备都当成一个个文件来操作。我们对文件的读写都通过调用内核提供的系统调用；内核给我们返回一个文件描述符（file descriptor）。而对一个 Socket 的读写也会有相应的描述符，称为 socketfd。应用进程对文件的读写通过对 fd 的读写完成。

4. Signal Driven I/O

信号驱动方式就是等数据准备好后，由内核发出 SIGIO 信号通知应用进程。示意图如下：

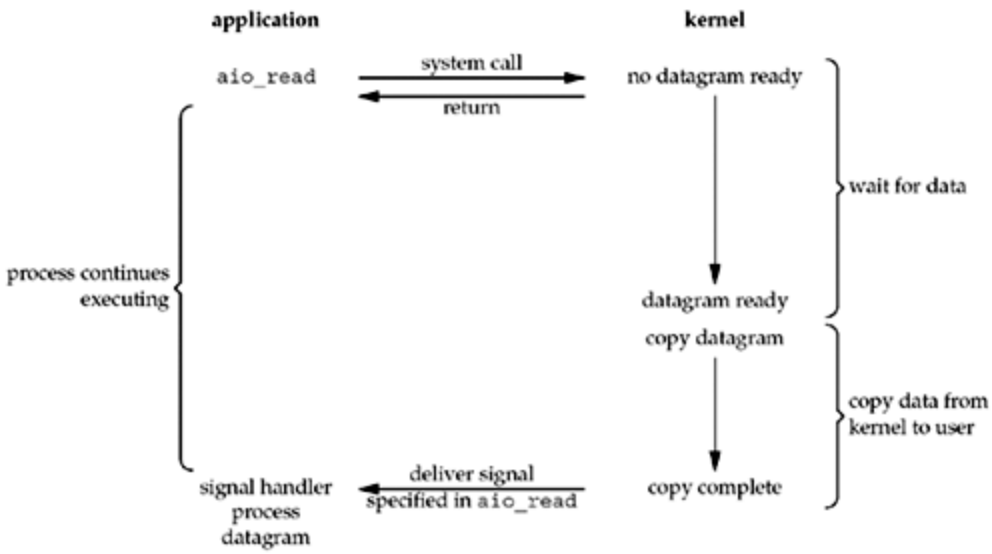


应用进程通过 sigaction 系统调用建立起 SIGIO 信号处理通道，然后此系统调用就返回，不阻塞。当数据准备好后，内核会产生一个 SIGIO 信号通知到应用进程。此时既可以使用

SIGIO 信号处理器通过 recvfrom 系统调用读取数据，然后通知应用进程数据准备好了，可以处理了；也可以直接通知应用进程读取数据。不管使用何种方式，好处都是应用进程不会阻塞，可以继续执行，只要等待信号通知数据准备好被处理了、数据准备好被读取了。

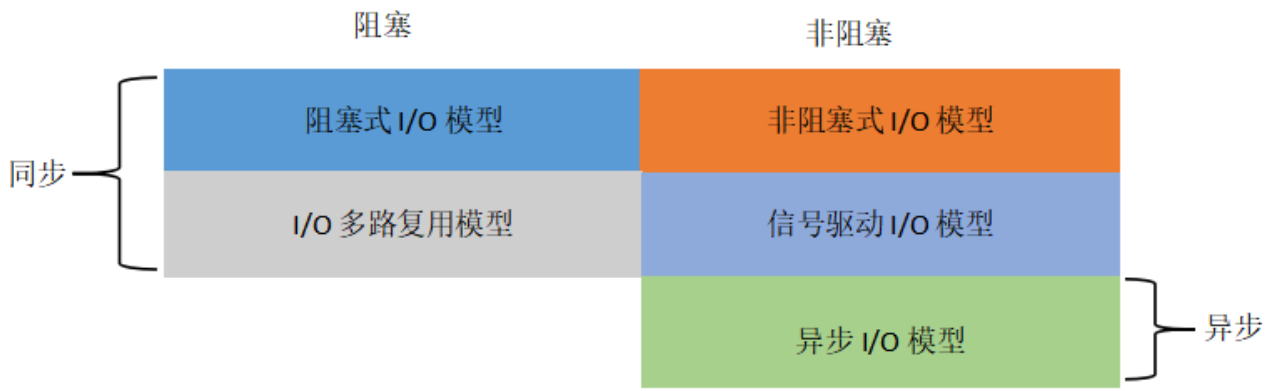
5. asynchronous I/O

异步 I/O 是由 POSIX 规范定义的。和信号驱动 I/O 模型的区别是前者内核告诉我们何时可以开始一个 I/O 操作，而后者内核会告诉我们一个 I/O 操作何时完成。示意图如下：



当用户进程发起系统调用后会立刻返回，并把所有的任务都交给内核去完成，不会被阻塞等待 I/O 完成。内核完成之后，只需返回一个信号告诉用户进程已经完成就可以了。

五种 I/O 模式可以从同步、异步，阻塞、非阻塞两个维度来划分：



零拷贝（Zero-copy）

在介绍零拷贝之前我们先看看传统的 Java 网络 IO 编程是怎样的。

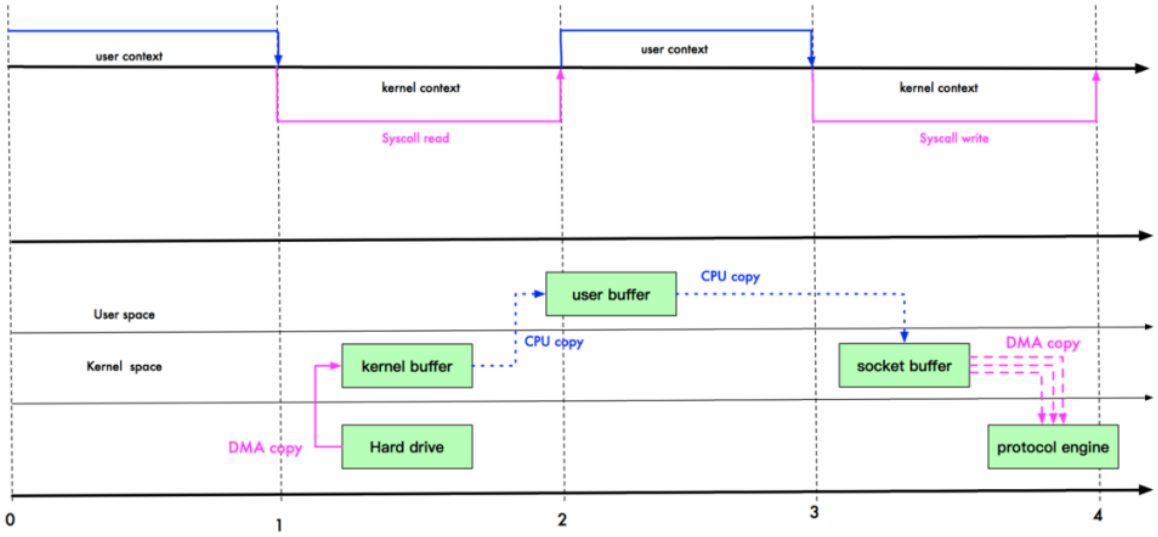
下面代码展示了一个典型的 Java 网络程序。

```
File file = new File("index.jsp");
RandomAccessFile rdf = new RandomAccessFile(file, "rw");

byte[] arr = new byte[(int) file.length()];
rdf.read(arr);

Socket socket = new ServerSocket(8080).accept();
socket.getOutputStream().write(arr);
```

程序中调用 RandomAccessFile 的 read 方法将 index.jsp 的内容读取到字节数组中。然后调用 write 方法将字节数组中的数据写入到 Socket 对应的输出流中发送给客户端。那么 Java 应用程序中的 read、write 方法对应到 OS 底层是怎样的呢。下图展示了这个过程。



图中上半部分记录了用户态和内核态的上下文切换。下半部分展示了数据的复制过程。上述 Java 代码对应的操作系统底层步骤：

1. read 方法触发操作系统从用户态到切换到内核态。同时通过 DMA 的方式从磁盘读取文件到内核缓冲区。DMA（Direct Memory Access）是 I/O 设备与主存之间由硬件组成的直接数据通路。即不需要 CPU 拷贝数据到内存，而是直接由 DMA 引擎传输数据到内存。
2. 紧接着发生第二次数据拷贝，即从内核缓冲区拷贝到用户缓冲区，同时发生一次内核态到用户态的上下文切换。
3. 调用 write 方法时，触发第三次数据拷贝，即从用户缓冲区拷贝到 Socket 缓冲区。同时发生一次用户态到内核态的上下文切换。
4. 最后数据从 Socket 缓冲区异步拷贝到网络协议引擎，这一步采用的是 DMA 方式。同

时没有发生上下文切换。

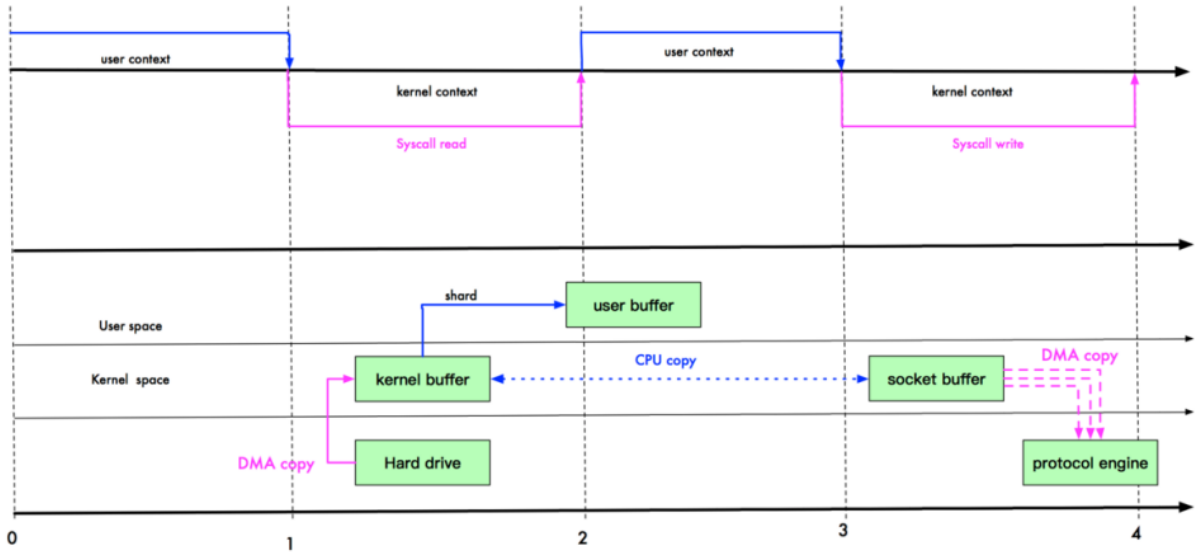
5. write 方法返回时，触发了最后一次内核态到用户态的切换。

由此可见，复制的操作太频繁，共有 2 次 DMA 拷贝、2 次 CPU 拷贝、4 次上下文切换。能否优化呢?

这就要介绍称之为"零拷贝"的技术。首先声明，零拷贝技术依赖底层 OS 内核提供的支持。Linux 中提供的这类支持有 mmap()，sendfile() 以及 splice() 系统调用。说白了就是减少数据在操作系统内核的缓冲区和用户应用程序地址空间的缓冲区之间进行拷贝。

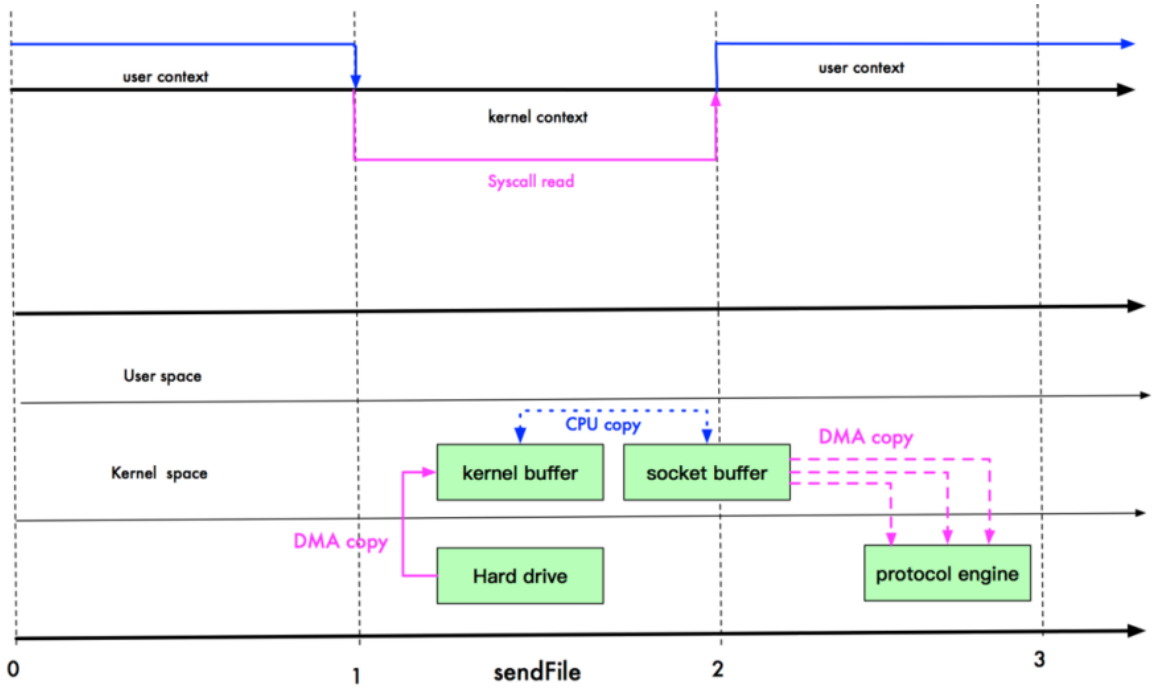
mmap

mmap 通过内存映射，将文件通过 DMA 的方式映射到内核缓冲区。操作系统会把这段内核缓冲区与应用程序（用户空间）共享。这样，在进行网络传输时，就能减少内核空间到用户空间的拷贝次数。此时输出数据时只要从内核缓冲区拷贝到 Socket 缓冲区即可。可见减少了一次 CPU 拷贝，但是上下文切换次数并没有减少。整个过程共 2 次 DMA 拷贝，1 次 CPU 拷贝，4 次上下文切换。示意图如下。

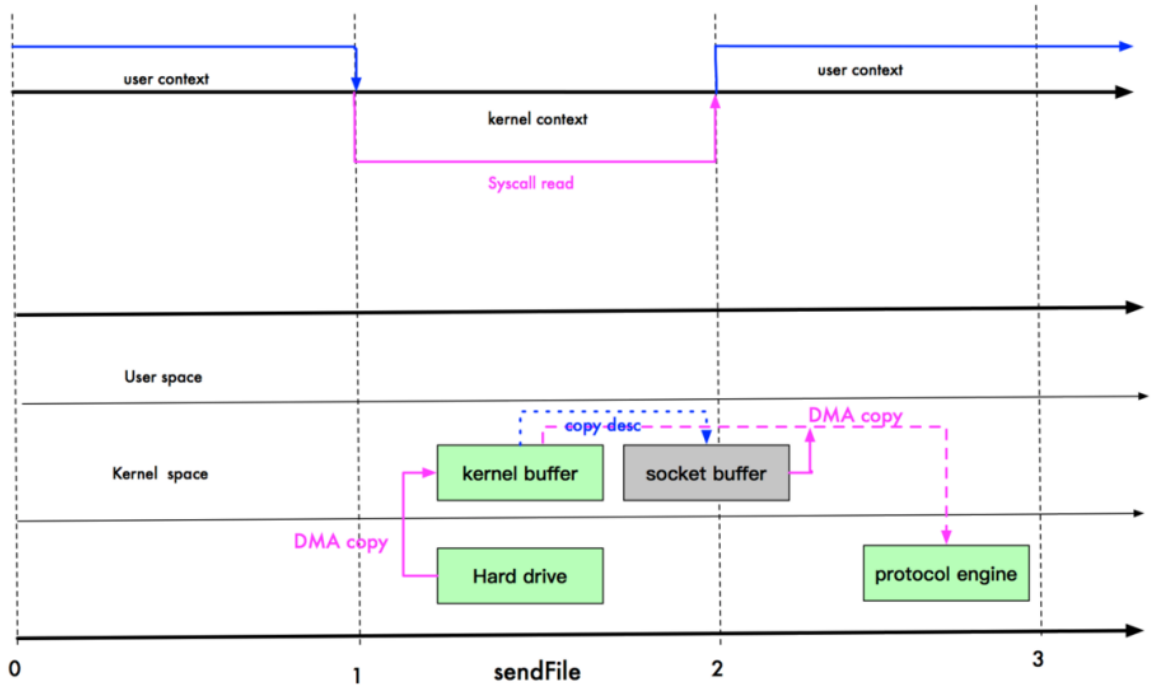


sendFile

Linux 2.1 开始提供了 sendFile 函数，其基本原理是：数据根本不经过用户态，直接从 Kernel Buffer 进入到 Socket Buffer，并且由于和用户态完全无关，这就避免了一次上下文切换。下图展示了整个过程。磁盘中的数据通过 DMA 引擎从复制到内核缓冲区。调用 write 方法时从内核缓冲区拷贝到 Socket 缓冲区。由于在同一个空间，因此没有发生上下文切换。最后由 Socket 缓冲区拷贝到协议引擎。整个过程共发生了 2 次 DMA 拷贝，1 次 CPU 拷贝，3 次上下文切换。



在 Linux 2.4 版本中，进一步做了优化。从 Kernel Buffer 拷贝到 Socket Buffer 的操作也省了，直接拷贝到协议栈，再次减少了 CPU 数据拷贝。下图展示了整个流程。本地文件 index.jsp 要传输到网络中，只需 2 次拷贝。第一次是 DMA 引擎从文件拷贝到内核缓冲区；第二次是从内核缓冲区将数据拷贝到网络协议栈；内核缓存区只会拷贝一些元信息，比如 offset 和 length 信息到 SocketBuffer，基本无消耗。



综上所述，最后一种方式发生了 2 次 DMA 拷贝、0 次 CPU 拷贝、3 次上下文切换。这就

是所谓的“零拷贝”实现。

因此零拷贝通常是站在操作系统的角度看，即整个过程中，内核缓冲区之间是没有重复数据的。同时伴随着更少的上下文切换。这就带来了 IO 性能质的提升！

实际开发中，mmap 和 sendFile 都有应用，可以认为是“零拷贝”的两种实现方式。它们都有各自的适用场景。mmap 更适合少量数据读写，sendFile 适合大文件传输。sendFile 可以利用 DMA 方式将内核缓冲区将数据拷贝到网络协议栈，减少 CPU 拷贝，而 mmap 则不能（必须从内核拷贝到 Socket 缓冲区）。

案例：RocketMQ 在 CommitLog 和 CosumerQueue 的实现中都采用了 mmap。而 Kafka 的零拷贝实现则使用了 sendFile。

RocketMQ 和 Kafka 高性能的原因之一便是顺序写入和近似顺序读取 + 零拷贝。

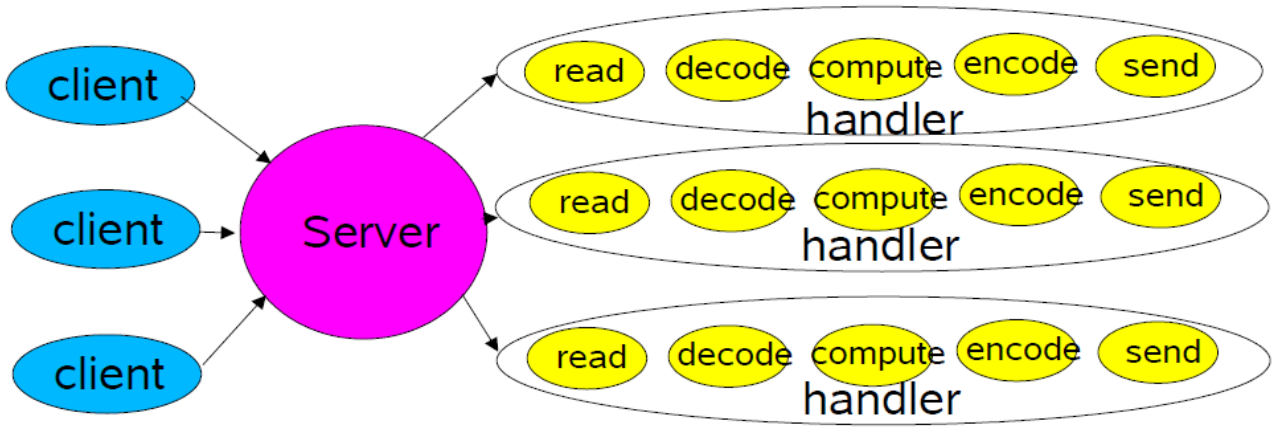
使用 Java NIO 实现零拷贝见前文 NIO 案例。

线程模型

线程模型通常是指线程的使用方式。在 Java I/O 中，主要有 2 种线程模型，即传统的阻塞式 I/O 模型和 Reactor 模型。

传统的阻塞式 I/O

正如我们前面写的传统 IO 通信案例版本 4。在版本 4 的例程中，为了同时处理多个客户端的请求，服务端为每一个连接都会分配一个新的线程处理。这个独立的线程完成数据的读写和业务处理。这虽然是“传统”的处理方式，但是也是最经典的 IO 线程模型。示意图如下：



该模型采用阻塞式 IO，连接创建后，如果当前线程暂时没有数据可读，该线程会阻塞在 read 操作，造成线程资源浪费。

当并发数很大，就会创建大量的线程，占用大量系统资源。

Reactor 模式

Reactor 模式针对传统 IO 的缺点，提出了解决方案。

- 方案 1：基于 I/O 复用模型。即多个连接共用一个阻塞对象，当某个连接有新的数据准备好时，操作系统通知应用程序，线程从阻塞状态返回，开始进行业务处理。
- 方案 2：基于线程池复用线程资源，不需要给每个连接创建一个线程。将连接完成后的业务处理任务分配给线程池中的线程进行处理。这样一个线程可以处理到多个客户端的业务。

总结一句话，I/O 多路复用 + 线程池，就是所谓的“Reactor 模式”的基本设计思想。其实我们前面 NIO 案例中的版本 1 的实现方式就有点这种味道，只不过不是严格意义上的 Reactor 模式罢了。

Reactor 模式中的两个核心组件：

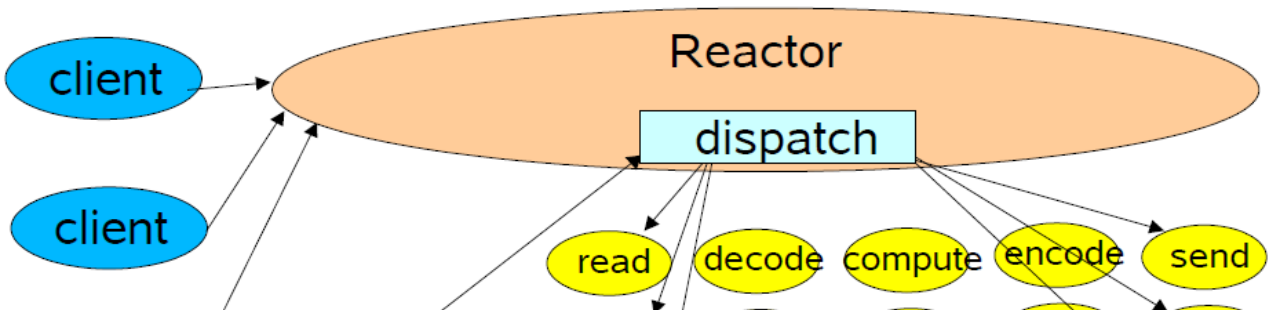
- 组件 1：Reactor。Reactor 在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理程序来对 I/O 事件做出反应。
- 组件 2：Handlers。完成实际 I/O 事件中数据的读写和要做的一系列业务处理。

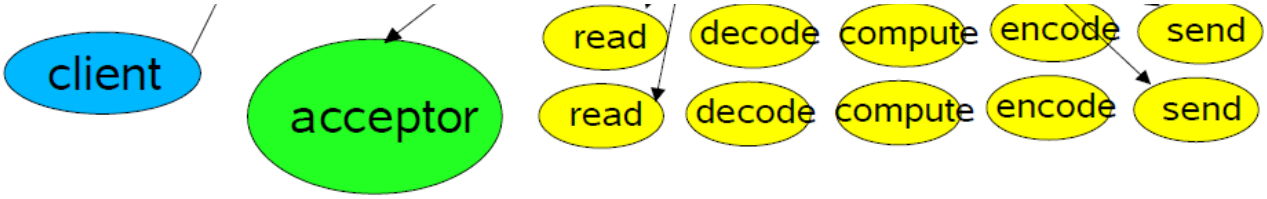
根据 Reactor 的数量和业务处理线程池线程数量不同，又分为 3 种具体实现：单 Reactor 单线程、单 Reactor 多线程、主从 Reactor。

单 Reactor 单线程

Reactor 对象通过 I/O 复用模型（在 Java NIO 中就是使用 Selector）监控客户端请求事件，收到事件后通过 dispatch 进行分发。如果是建立连接请求事件，则由 acceptor 通过 accept 处理连接请求，然后创建一个 Handler 对象处理连接完成后的数据**读** -> **业务处理** -> **写**。

注意，上述过程都是发生在一个线程里，只不过是非阻塞方式。工作原理示意图如下：





这种方式，服务器端使用一个线程基于多路复用就完成了所有的 IO 操作（包括连接，读数据、业务处理、写数据等），没有多线程间通信、竞争的问题，实现简单。但是如果客户端连接数较多，将无法支撑。因为只有一个线程，不能完全发挥多核 CPU 的性能。且 Handler 在处理某个连接上的业务时，整个线程无法处理其他连接事件。如果业务处理很耗时，很容易会导致性能瓶颈。如果线程意外终止，或者进入死循环，会导致整个系统不可用。

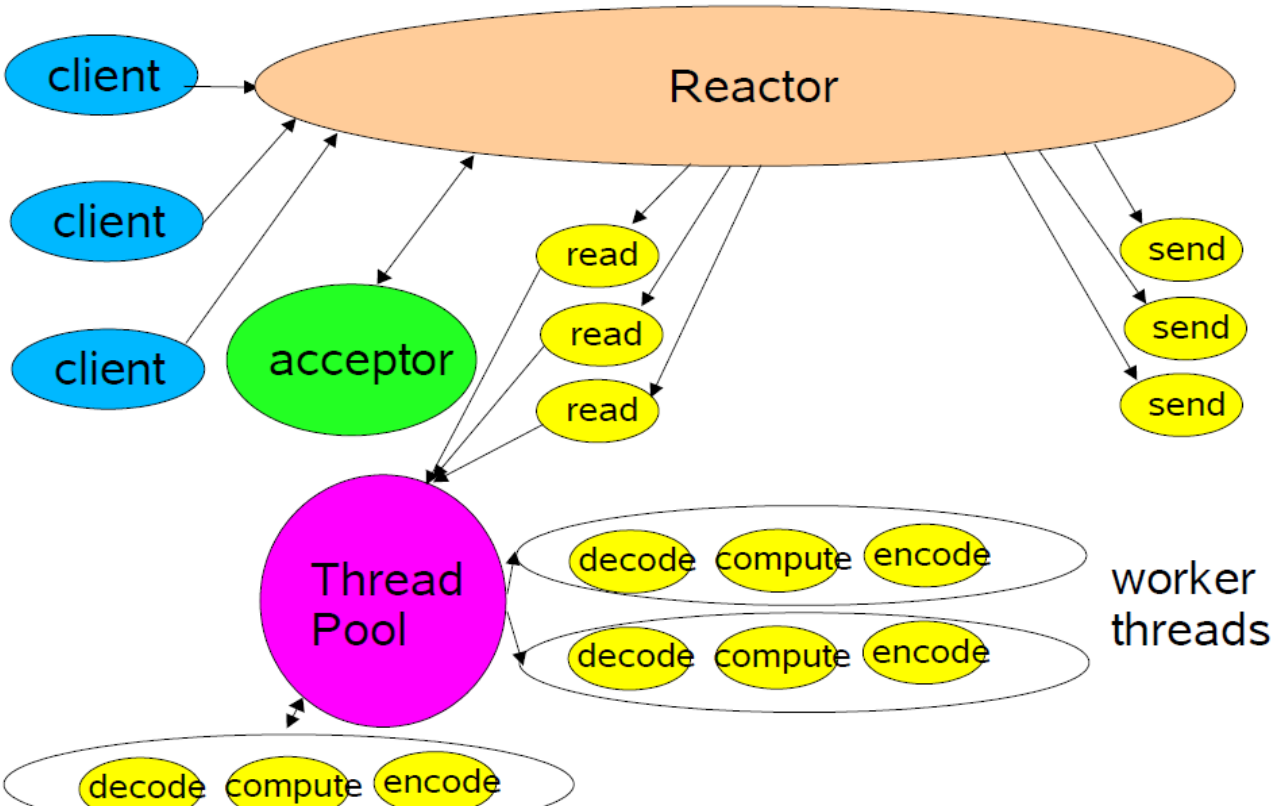
单 Reactor 多线程

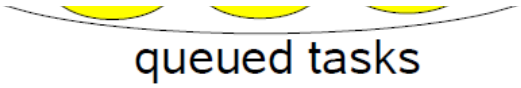
为了克服上述模型的缺点，我们可以考虑将**非 IO** 操作从 Reactor 线程的处理中移出，来提升 Reactor 线程的性能。

具体说明如下：

eactor 对象通过 select 监控 client 端的请求事件，收到事件后，通过 dispatch 进行分发。

如果是连接建立请求，则由 acceptor 通过 accept 处理连接请求，然后分配一个 Handler 对象处理完成连接后的数据读写。





如果不是连接请求，则由 reactor 分发（dispatch）给连接对应的 Handler 来处理。Handler 和 Reactor 运行在同一个线程中。

Handler 只负责响应 IO 事件，不做具体的业务处理。read 数据后，会分发给 Worker 线程池的某个线程进行业务逻辑处理。

Worker 线程池会分配单独的线程完成真正的业务处理，包括编解码、逻辑计算，完成处理后将结果数据返回给 handler。

Handler 收到响应后，通过 send 将数据返回给 client 端。

这种模型下，Reactor 线程只负责处理所有的事件的监听和响应（数据读、写），而不参与数据的业务处理（数据编解码、逻辑处理）。业务处理的任务交给线程池中的线程处理，提高了并发性能，特别是在业务复杂的情况下。工作原理示意图如下：

主从 Reactor 多线程

上述单 Reactor 多线程模型虽然可以充分压榨 CPU 的性能，但是由于 Reactor 是单线程运行的，所以在高并发场景下 Reactor 容易成为性能瓶颈。可以考虑让 Reactor 在多线程中运行，这就是多 Reactor 模型，也叫主从 Reactor 模型。

具体说明如下：

Reactor 主线程 mainReactor 通过 select 监听连接事件，收到事件后，通过 acceptor 处理连接事件。

当 acceptor 处理连接事件后，mainReactor 将连接分配给 subReactor。subReactor 是 Reactor 的子线程，和 mainReactor 不在一个线程中。

subReactor 将连接加入到连接队列进行监听，并负责创建 handler 进行各种事件的处理（数据的读、写）。

subReactor 也通过 select 监听，当有新事件发生时，subreactor 就会调用对应的 handler 处理。

handler 只负责数据的 I/O，针对数据的业务处理还是由 worker 线程池中的线程处理，并返回结果。

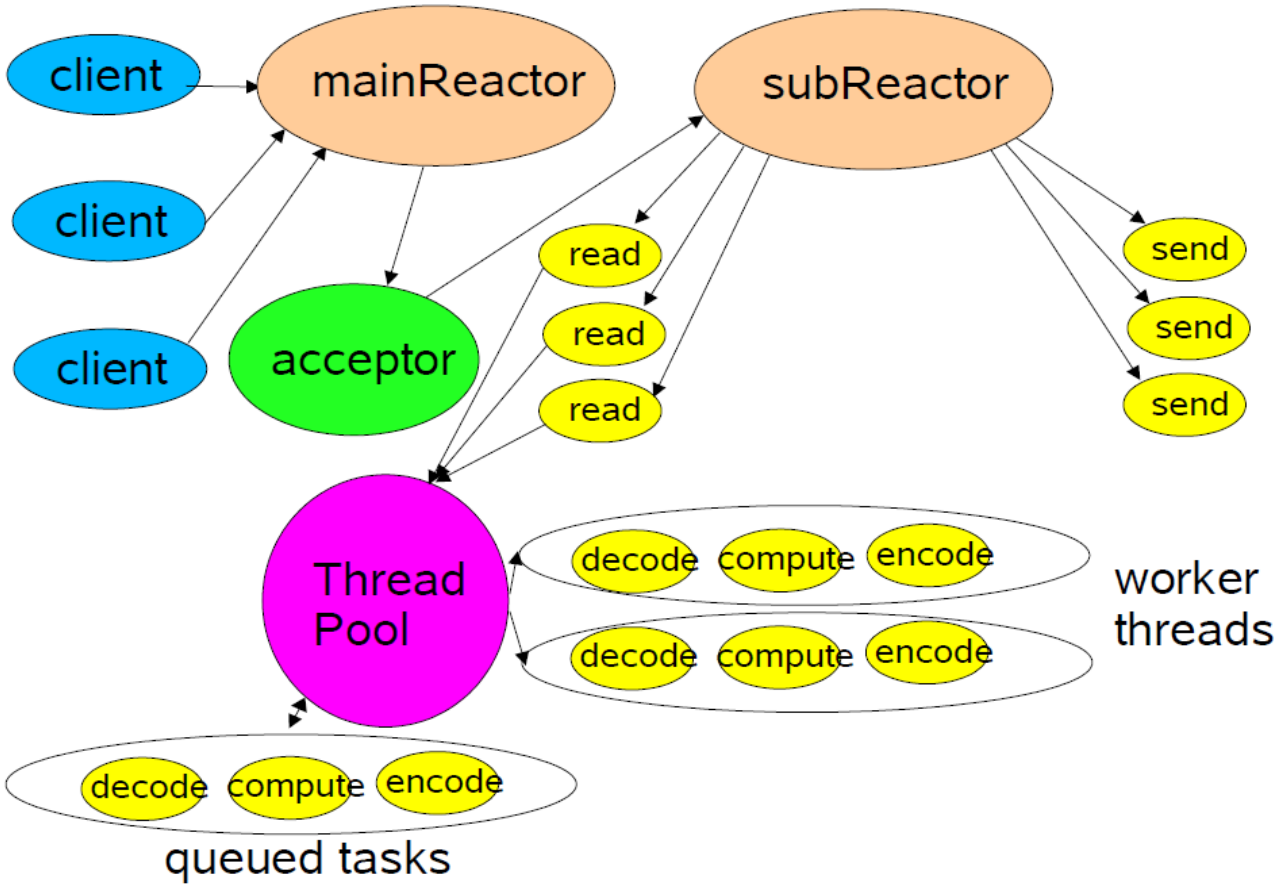
handler 收到 worker 线程的响应数据后，通过 send 将结果数据返回给 client。

Reactor 主线程可以对应多个 Reactor 子线程，即 MainRecator 能关联多个 SubReactor。和 worker 线程池一样，线程数都能配置。

这种方式的优点非常明显，就是减轻了 mainRecator 的负担，让其只负责处理连接请求，不包含 I/O 的处理。后续的处理统统交给 SubReactor。主、从 Reactor 分别运行在不同的线程中，且线程数可以配置。业务处理还是交给 worker 线程池中的线程执行。

主从 Reactor 线程模型在许多项目中都有应用，比如 Nginx 的主从 Reactor 多进程模型、Netty 的主从多线程模型等。

其工作原理示意图如下（注意观察和上面一个图的区别）：



总结

本篇我们带领读者回顾了一下 Java 中 IO 相关的理论知识，并通过多个代码案例加深了理解。

上一页 下一页