

# Java IO

Ning Ding

February 2024

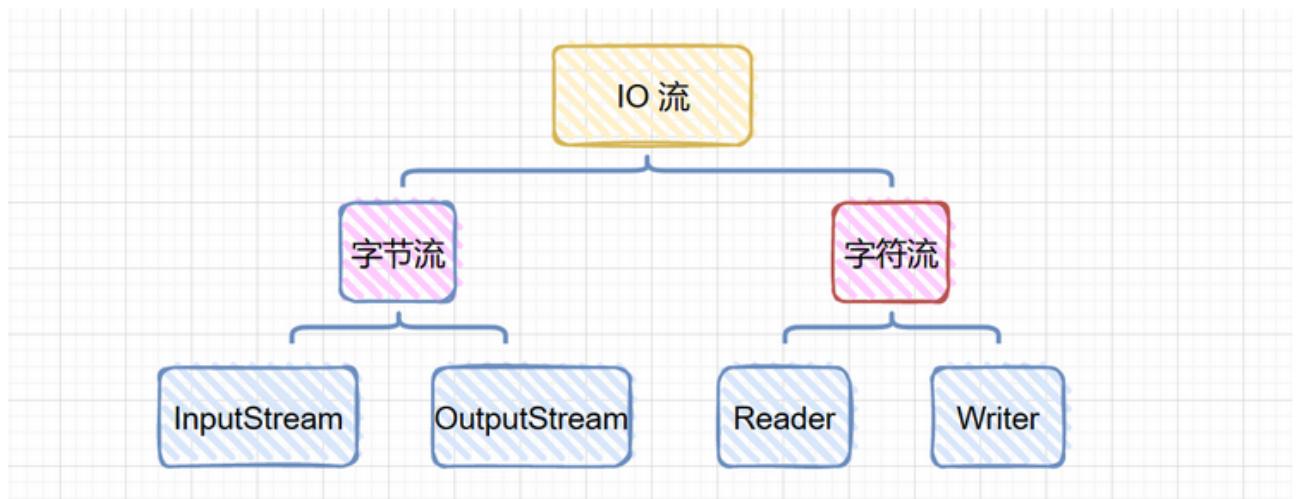
## 1 Introduction

Java中的IO操作主要是通过File, InputStream, OutputStream, Reader, Writer 来实现的。

什么是流:

Streams handle data as a sequential flow of bytes. While you might be working with text, images, or complex objects, the stream itself sees everything as a series of bytes.

I/O流的分类:



## 2 IO流

### 2.1 IO流读取

从硬盘读取: FileInputStream

```
1 import java.io.FileInputStream;
2 import java.io.IOException;
3
4 public class FileInputStreamExample {
5     public static void main(String[] args) {
6         String filePath = "myTextFile.txt";
7
8         try (FileInputStream fileStream = new FileInputStream(filePath)) {
9             int byteData;
10            while ((byteData = fileStream.read()) != -1) {
11                System.out.print((char) byteData);
12            }
14        } catch (IOException e) {
15            e.printStackTrace();
16        }
17    }
18}
```

```

12     }
13 } catch (IOException e) {
14     System.out.println("Error reading file: " + e.getMessage());
15 }
16 }
17 }

```

从内存读取: ByteArrayInputStream

```

1 import java.io.ByteArrayInputStream;
2
3 public class ByteArrayInputStreamExample {
4     public static void main(String[] args) {
5         byte[] inputData = {65, 66, 67, 68}; // Sample data
6
7         try (ByteArrayInputStream byteStream = new ByteArrayInputStream(
8             inputData)) {
9             int byteData;
10            while ((byteData = byteStream.read()) != -1) {
11                System.out.print((char) byteData);
12            }
13        } catch (IOException e) {
14            System.out.println("Error reading data: " + e.getMessage());
15        }
16    }
17 }

```

从用户输入读取: System.in

```

1 import java.io.InputStreamReader;
2 import java.io.BufferedReader;
3 import java.io.IOException;
4
5 public class SystemInExample {
6     public static void main(String[] args) {
7         try (BufferedReader reader = new BufferedReader(new
8             InputStreamReader(System.in))) {
9             System.out.println("Enter some text: ");
10            String inputText = reader.readLine();
11            System.out.println("You entered: " + inputText);
12        } catch (IOException e) {
13            System.out.println("Error reading input: " + e.getMessage());
14        }
15    }
16 }

```

## 2.2 File类

File类是Java中为文件进行创建、删除、重命名、移动等操作而设计的一个类。

File本身不进行写入或读取，但可以通过`FileInputStream`和`FileOutputStream`进行读取和写入

```

1 import java.io.FileInputStream;
2 import java.io.FileOutputStream;
3 import java.io.IOException;
4
5 public class FileStreamReadWrite {
6     public static void main(String[] args) {

```

```

7         String filePath = "example.txt";
8
9         // Writing to the file
10        // 在try里使用结束时会自动关闭流(JAVA7), 否则需要手动关闭流以避免阻塞
11        try (FileOutputStream outputStream = new FileOutputStream(filePath))
12        {
13            String textToWrite = "Hello, this is some text for the file!\n";
14            byte[] textBytes = textToWrite.getBytes();
15            outputStream.write(textBytes);
16        } catch (IOException e) {
17            System.out.println("Error writing to file: " + e.getMessage());
18        }
19
20        // Reading from the file
21        try (FileInputStream inputStream = new FileInputStream(filePath)) {
22            int byteData;
23            while ((byteData = inputStream.read()) != -1) {
24                System.out.print((char) byteData);
25            }
26        } catch (IOException e) {
27            System.out.println("Error reading from file: " + e.getMessage());
28        };
29    }

```

## 2.3 Read方法

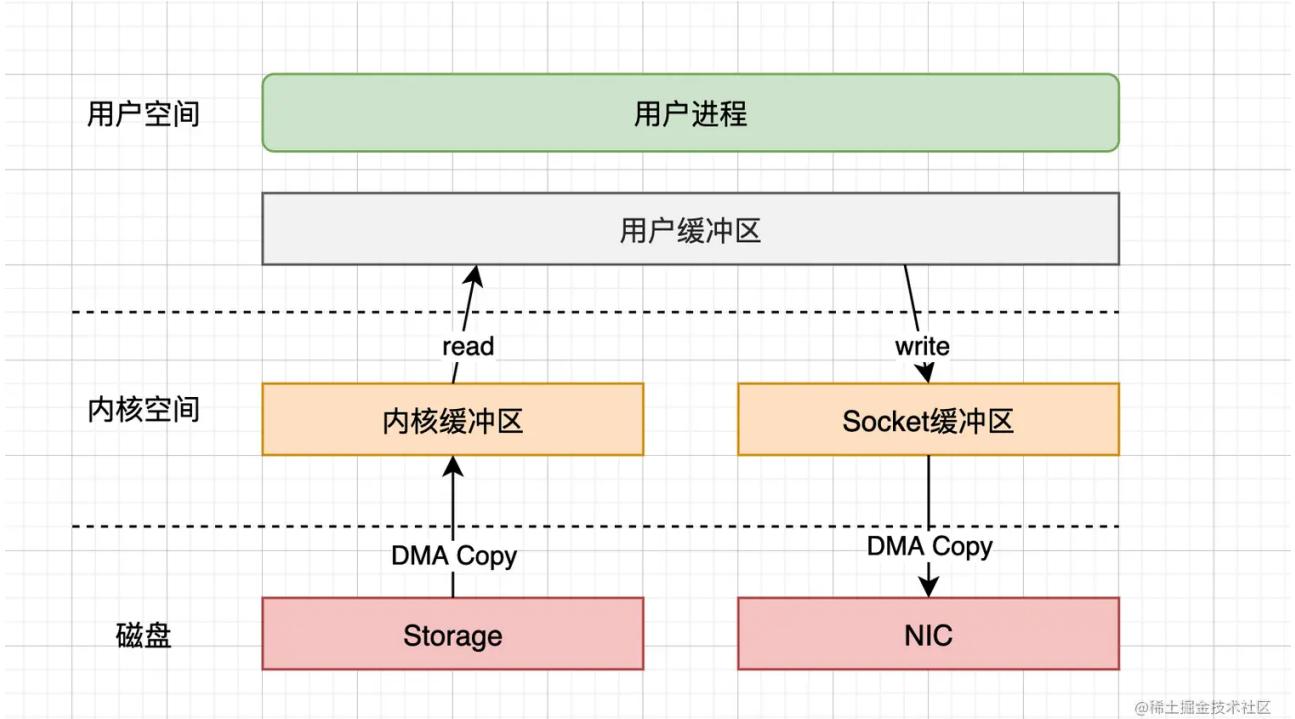
```

1 try(FileInputStream fileInputStream = new FileInputStream("test.txt");
2 FileOutputStream fileOutputStream = new FileOutputStream("dup.txt")) {
3     // 读取下一个字节, 如果没有输入, 会阻塞
4     int i = fileInputStream.read();
5
6     int i = 0;
7     // 直到读取到eof, 一直输出字节
8     while ((i = fileInputStream.read()) != -1) {
9         fileOutputStream.write(i); // 如果有1000个字节, 会执行1000次, 严重影响性能
10    }
11
12    // 可以使用buffer
13    byte[] buffer = new byte[1024];
14    while ((i = fileInputStream.read(buffer)) != -1) { // 先把数据读取到缓冲区
15        System.out.print(new String(buffer, 0, i));
16    }
17}

```

## 2.4 Read原理

操作系统的内核是核心，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证内核的安全，用户进程不能直接操作内核，操作系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。



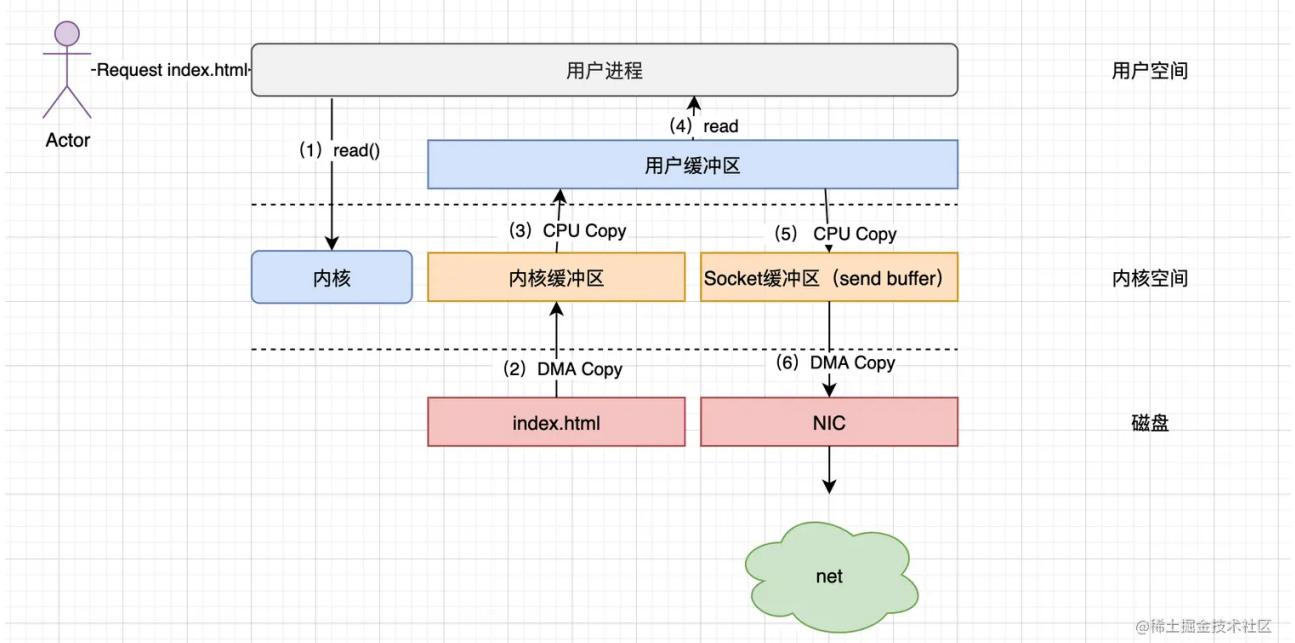
@稀土掘金技术社区

当一个用户进程要从磁盘读取数据时，内核一般不直接读磁盘，而是将内核缓冲区中的数据复制到进程缓冲区中。

但若是内核缓冲区中没有数据，内核会把对数据块的请求，加入到请求队列，然后把进程挂起，为其它进程提供服务。

等到数据已经读取到内核缓冲区时，把内核缓冲区中的数据读取到用户进程中，才会通知进程。

可以认为，read是把数据从内核缓冲区复制到进程缓冲区。write是把进程缓冲区复制到内核缓冲区。当然，write并不一定导致内核的写动作，比如os可能会把内核缓冲区的数据积累到一定量后，再一次写入。



@稀土掘金技术社区

## 2.5 ByteArrayOutputStream 基于内存字节输入输出

```

1 public class MemoryDemo{
2     static String str = "hello world";

```

```

3 //不需要关闭流，因为没有占用底层资源
4     public static void main(String[] args) {
5         //从内存中读取数据
6         ByteArrayInputStream inputStream = new ByteArrayInputStream(str.getBytes());
7
8         //写入到内存中
9         ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
10        int i = 0;
11        while ((i = inputStream.read()) != -1) {
12            char c = (char)i;
13            outputStream.write(Character.toUpperCase(c));
14        }
15        System.out.println(outputStream.toString());
16    }
17 }
```

## 2.6 BufferedStream 缓存流

优先从缓存区读取数据，如果读取不到，从磁盘读取并写入缓存区，再返回读取的数据。

其优点在于：与其每次都直接从磁盘或网络读取/写入很小的一块数据，缓冲流会把数据先聚集到缓冲区里。它们可以一次性执行更大规模的读写操作，从而减少开销，提高速度。

```

1 try(
2     BufferedInputStream bufferedIS = new BufferedInputStream(new
3         FileInputStream("test.txt"));
4     BufferedOutputStream bufferedOS = new BufferedOutputStream(new
5         FileOutputStream("dup.txt"))
6     ) {
7         int len = 0;
8         byte[] bys = new byte[1024];
9         while ((len = bufferedIS.read(bys)) != -1) {
10             System.out.println(new String(bys,0,len));
11             bufferedOutputStream.write(bys,0,len);
12             bufferedOutputStream.flush();
13         }
14     }
```

### 2.6.1 flush

当你在写文件时，数据可能先存放在缓冲区内。只有当你调用”flush”，或者关闭文件时，才会确保数据实际写入硬盘。类似地，在网络传输时，”flush”可以确保数据包被及时发送，而不是存留等待后续数据加入。

注意：过于频繁地使用”flush”可能会影响性能，因为每次”flush”都牵扯到磁盘或网络的实际写入操作。需要在数据完整性和性能之间找到适合的平衡。

## 3 字符流Writer and Reader

InputStream/OutputStream：专注于处理字节形式的数据。它们底层操作的是原始的二进制数据，不关心具体的字符编码。适合用于处理图片、音频、视频、网络传输等二进制数据流。

Reader/Writer: 专注于处理字符（也就是文本）形式的数据。它们可以自动进行编码和解码，你不需要直接处理字节。适合用于处理配置文件、日志、源代码等文本型数据。

```
1 import java.io.*;
2
3 public class ReaderWriterExample {
4     public static void main(String[] args) throws IOException {
5         try (BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
6              BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"))) {
6             String line;
7             while ((line = reader.readLine()) != null) { //直接读取一行字符，而不是像IOStream一样一个字节一个字节读取
8                 writer.write(line);
9                 writer.newLine();
10            }
11        }
12    }
13 }
14 }
```

### 3.1 序列化与反序列化ObjectStream

序列化是把对象的状态信息转化为可存储或传输的形式过程，也就是把对象转化为字节序列的过程称为对象的序列化。

反序列化是序列化的逆向过程，把字节数组反序列化为对象，把字节序列恢复为对象的过程成为对象的反序列化

```
1 \\\Object declaration
2 import java.io.Serializable;
3
4 //The object need to implements Serializable
5 public class Person implements Serializable {
6     private String name;
7     private int age;
8
9     public Person(String name, int age) {
10         this.name = name;
11         this.age = age;
12     }
13
14     public String getName() {
15         return name;
16     }
17
18     public int getAge() {
19         return age;
20     }
21
22     @Override
23     public String toString() {
24         return "Person{" +
25             "name='" + name + '\'' +
26             ", age=" + age +
27             '}';
28     }
29 }
30 }
```

```

31 \\read and write with object stream
32 import java.io.FileOutputStream;
33 import java.io.FileInputStream;
34 import java.io.ObjectOutputStream;
35 import java.io.ObjectInputStream;
36 import java.io.IOException;
37
38 public class ObjectStreamExample {
39     private static void serializePerson(String filename) {
40         Person person = new Person("Alice", 30);
41
42         try (ObjectOutputStream out = new ObjectOutputStream(new
43             FileOutputStream(filename))) {
44             out.writeObject(person);
45             System.out.println("object written: " + person);
46         } catch (IOException e) {
47             System.out.println("error: " + e.getMessage());
48         }
49     }
50
51     private static void deserializePerson(String filename) {
52         try (ObjectInputStream in = new ObjectInputStream(new
53             FileInputStream(filename))) {
54             Person serializedPerson = (Person) in.readObject();
55             System.out.println("Object read: " + serializedPerson);
56         } catch (IOException | ClassNotFoundException e) {
57             System.out.println("error: " + e.getMessage());
58         }
59     }
60
61     public static void main(String[] args) {
62         String filename = "person_data.bin";
63
64         // 1. 序列化- 写对象
65         serializePerson(filename);
66
67         // 2. 反序列化- 读对象
68         deserializePerson(filename);
69     }
70 }
```

## 4 网络IO

### 4.1 Socket和ServerSocket

Socket (套接字): 可以把Socket 看作是网络通信的一个端点。想要在网络上进行通讯的两个程序之间，就需要分别创建一个Socket。

Server Socket (服务器套接字): Server Socket 用于在服务器端监听来自客户端的连接请求。当一个请求到来时，Server Socket 会接受请求，创建专门用于处理这次通信的普通Socket 对象。

Client Socket (客户端套接字): Client Socket 代表客户端，用于向服务器发起连接请求。

服务端:

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.io.PrintWriter;
5 import java.net.ServerSocket;
6 import java.net.Socket;
7
8 public class SimpleServer {
9     public static void main(String[] args) {
10         int portNumber = 5000; // 服务器监听的端口
11
12         try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
13             System.out.println("Server is listening on port " + portNumber);
14
15             while (true) {
16                 // 阻塞操作，等待客户端的连接
17                 try (Socket clientSocket = serverSocket.accept()); // 接收连接请求
18
19                     PrintWriter out = new PrintWriter(clientSocket.
20                         getOutputStream(), true);
21                     BufferedReader in = new BufferedReader(new
22                         InputStreamReader(clientSocket.getInputStream()));
23
24                     ) {
25                         System.out.println("New client connected");
26
27                         String inputLine;
28                         while ((inputLine = in.readLine()) != null) {
29                             System.out.println("Client says: " + inputLine);
30
31                             // 简单地把收到的内容直接发回客户端
32                             out.println(inputLine);
33                         }
34                     } catch (IOException e) {
35                         System.out.println("Exception caught when handling a
36                         client: " + e.getMessage());
37                     }
38                 }
39             }
40         }
41     }

```

客户端:

```

1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.io.PrintWriter;
5 import java.net.Socket;
6 import java.util.Scanner;
7
8 public class SimpleClient {
9     public static void main(String[] args) {
10         String serverHostname = "localhost"; // 服务器地址
11         int portNumber = 5000;
12
13         try (Socket socket = new Socket(serverHostname, portNumber);
14             PrintWriter out = new PrintWriter(socket.getOutputStream(),

```

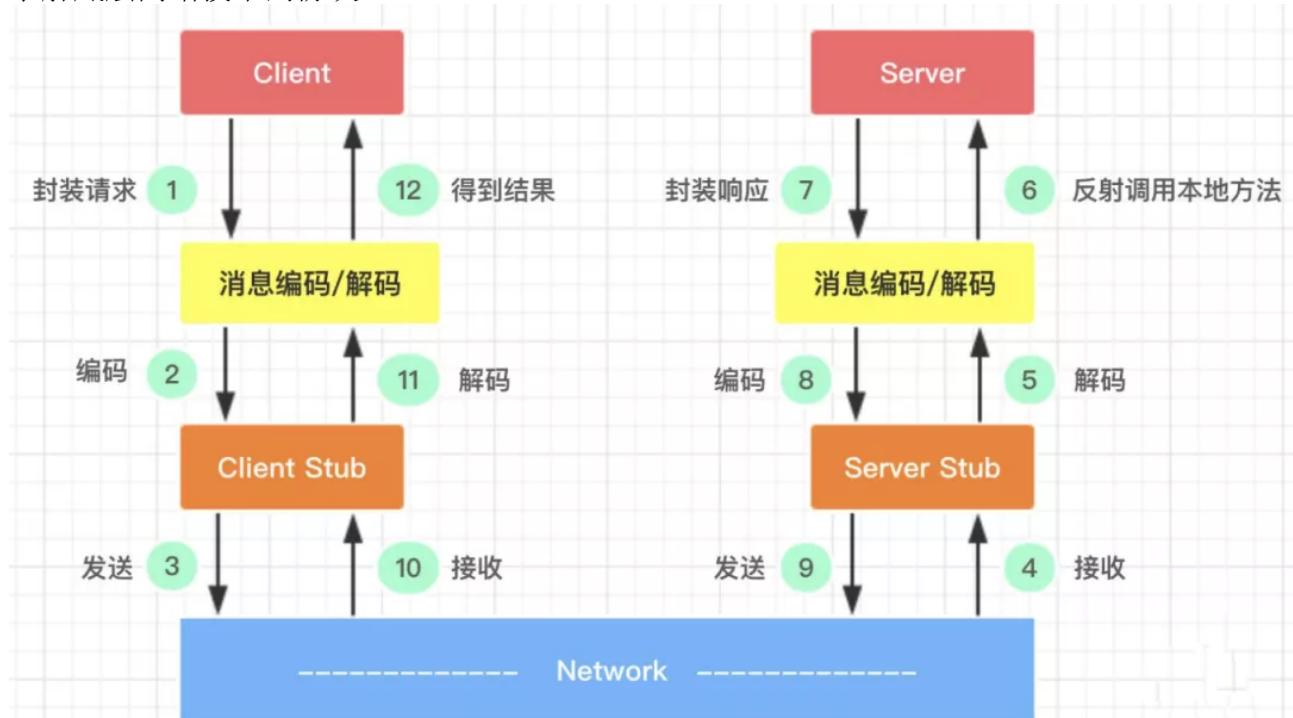
```

15         true);
16         BufferedReader in = new BufferedReader(new InputStreamReader(
17             socket.getInputStream()));
18         Scanner scanner = new Scanner(System.in)
19     ) {
20         System.out.println("Connected to server");
21
22         while (true) {
23             System.out.print("Enter message to send (type 'exit' to quit
24 ): ");
25             String messageToSend = scanner.nextLine();
26
27             if (messageToSend.equals("exit")) {
28                 break;
29             }
30
31             out.println(messageToSend);
32             String response = in.readLine();
33             System.out.println("Server response: " + response);
34         }
35     }
36 }

```

## 5 RPC协议

RPC(Remote Procedure Call Protocol)远程过程调用协议。客户端在不知道调用细节的情况下，调用存在于远程计算机上的某个对象，就像调用本地应用程序中的对象一样，而不需要了解底层网络技术的协议。



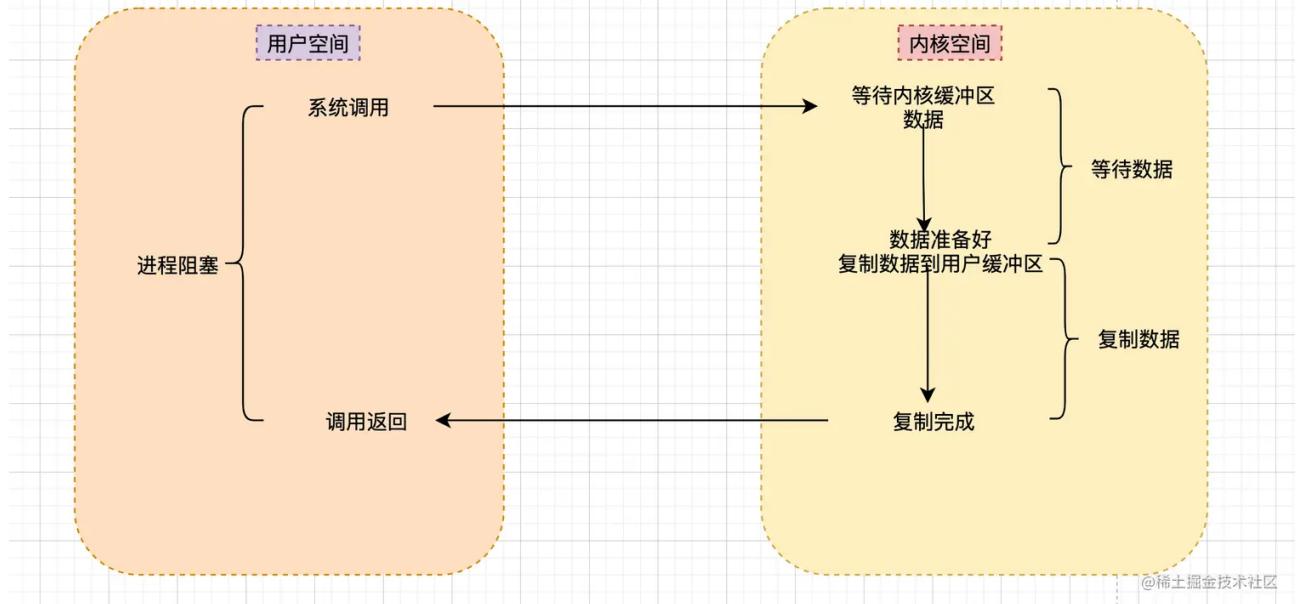
RPC的目标就是要2 8这些步骤通过动态代理模式，在执行该方法的前后对数据进行封装和解码等，让用于感觉就像是直接调用该方法一样。

# 6 阻塞与非阻塞(Blocking and non-Blocking IO)

正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态。(当进程进入阻塞状态，是不占用CPU资源的。)

## 6.1 同步阻塞

用户空间的应用程序执行一个系统调用，这会导致应用程序阻塞，什么也不干，直到数据准备好，并且将数据从内核复制到用户进程，最后进程再处理数据，在等待数据到处理数据的两个阶段，整个进程都被阻塞，不能处理别的网络IO。

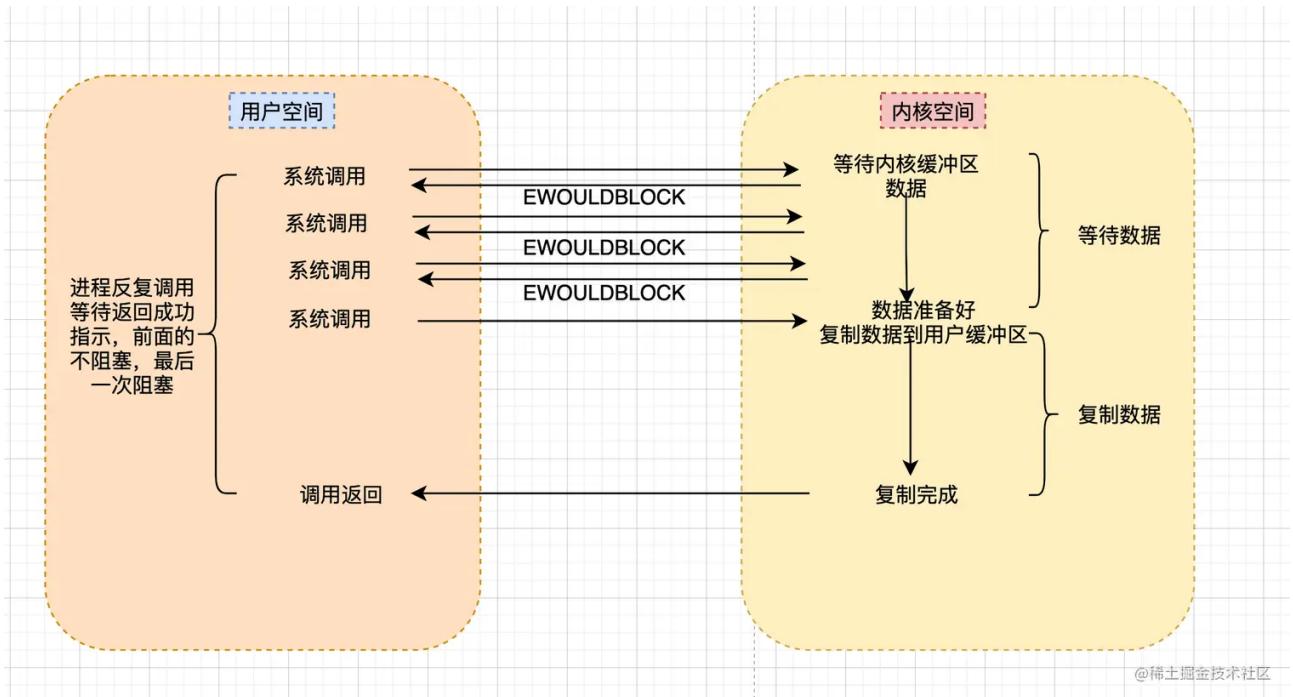


## 6.2 同步非阻塞

非阻塞的系统调用调用之后，进程并没有被阻塞，内核马上返回给进程，如果数据还没准备好，此时会返回一个error。

- 进程在返回之后，可以干点别的事情，然后再发起系统调用。
- 重复上面的过程，循环往复的进行系统调用。这个过程通常被称之为轮询。
- 轮询检查内核数据，直到数据准备好，再拷贝数据到进程，进行数据处理。
- 需要注意，拷贝数据整个过程，进程仍然是属于阻塞的状态。
- 这种方式在编程中对Socket设置`O_NONBLOCK`即可。

缺点是过多轮询遍历会增加cpu的消耗。



### 6.3 IO多路复用

IO多路复用，这是一种进程预先告知内核的能力，让内核发现进程指定的一个或多个IO条件就绪了，就通知进程。使得一个进程能在一连串的事件上等待。

IO复用的实现方式目前主要有Select、Poll和Epoll:

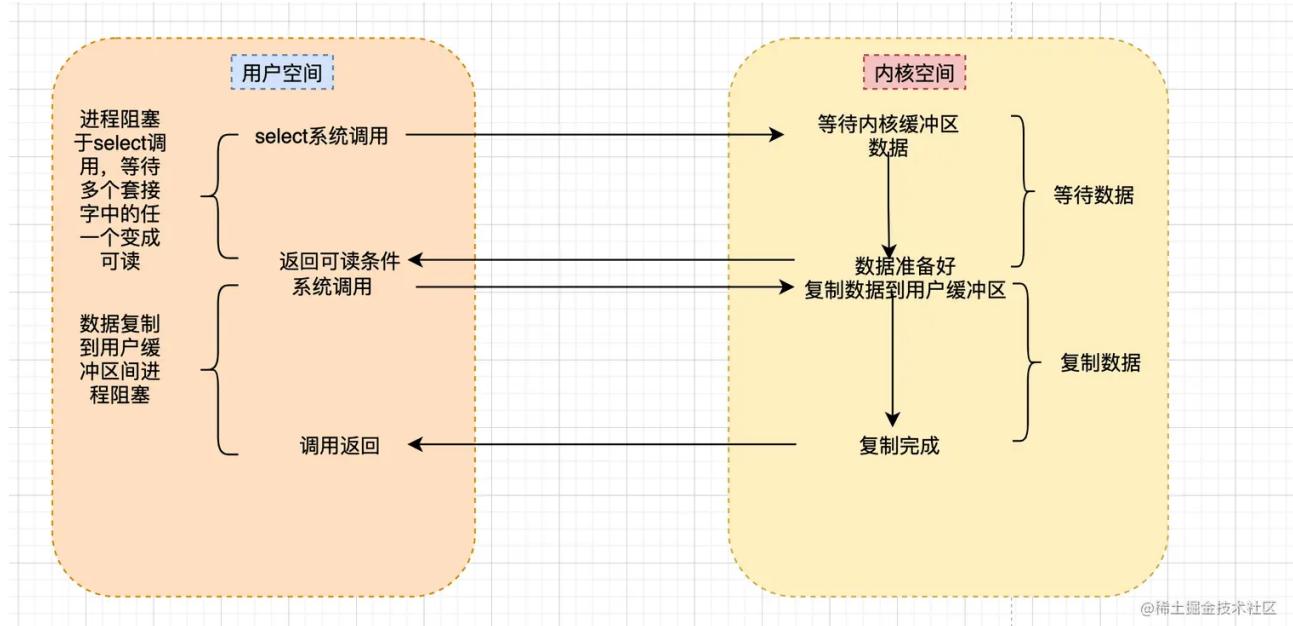
- 优点是：单个进程可以同时处理多个网络连接请求。系统资源消耗较小。
- 跟同步非阻塞类似，会多次轮询所有打开的fd，性能消耗大

select, poll, epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select, poll, epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现。

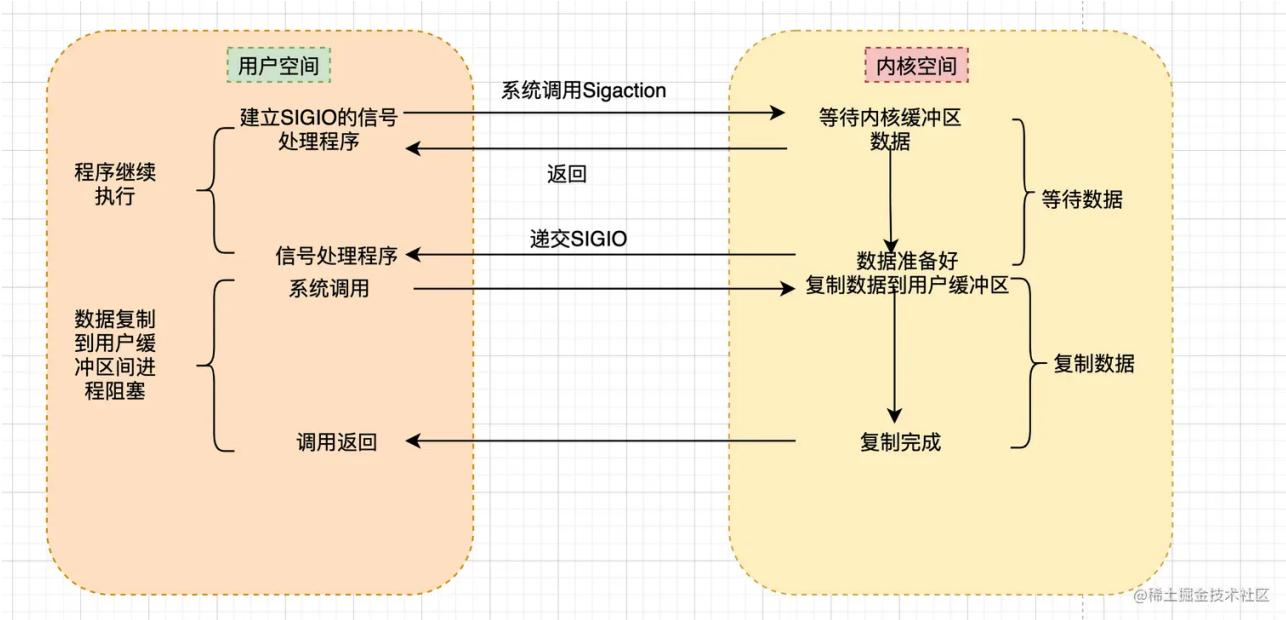
	<b>select</b>	<b>poll</b>	<b>epoll</b>
操作方式	遍历	遍历	回调
数据结构	bitmap	数组	红黑树
最大连接数	1024 (x86) 或 2048 (x64)	无上限	无上限
最大支持文件描述符数	一般有最大值限制	65535	65535
fd拷贝	每次调用select，都需要把fd集合从用户态拷贝到内核态	每次调用poll，都需要把fd集合从用户态拷贝到内核态	fd首次调用epoll_ctl拷贝，每次调用epoll_wait不拷贝
工作模式	LT	LT	支持ET高效模式
工作效率	每次调用都进行线性遍历，时间复杂度为O(n)	每次调用都进行线性遍历，时间复杂度为O(n)	事件通知方式，每当fd就绪，系统注册的回调函数就会被调用，将就绪fd放到readyList里面，时间复杂度O(1)

epoll是Linux目前大规模网络并发程序开发的首选模型。在绝大多数情况下性能远超select和poll。目前流行的高性能web服务器Nginx正式依赖于epoll提供的高效网络套接字轮询服务。但是，在并发连接不高的情况下，多线程+阻塞I/O方式可能性能更好。



## 6.4 信号驱动

首先我们允许Socket进行信号驱动IO，并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。

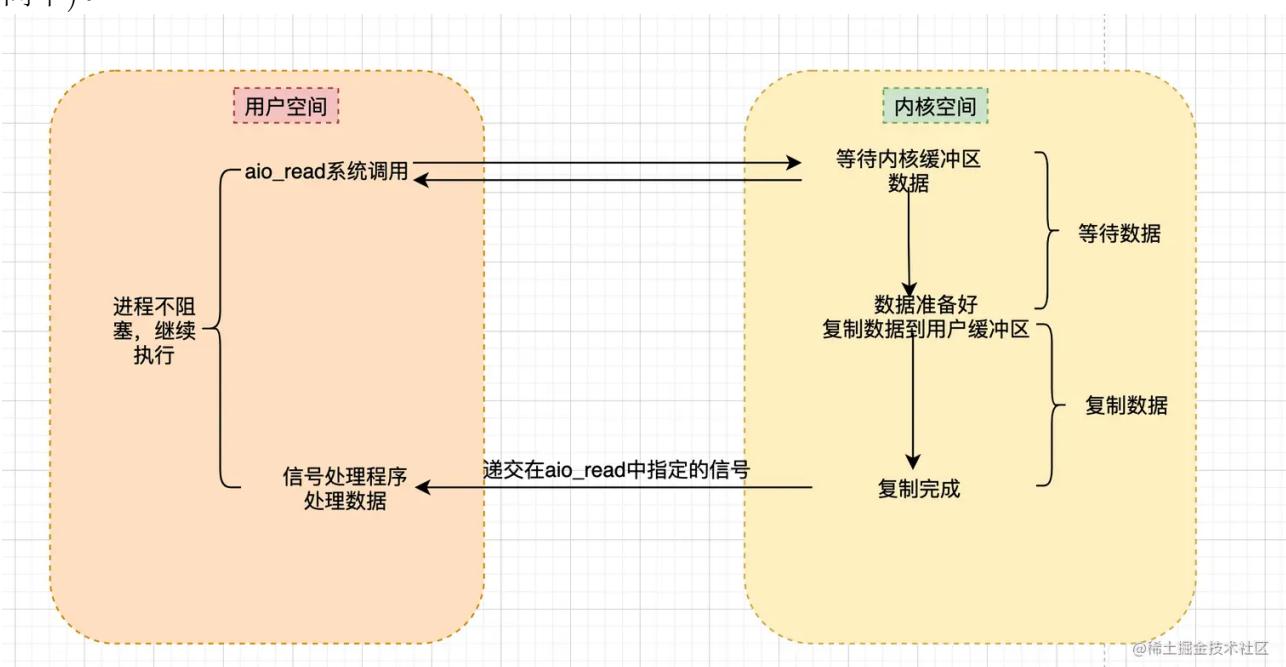


## 6.5 异步非阻塞

- 当用户线程调用了[aio\\_read](#)系统调用，立刻就可以开始去做其它的事，用户线程不阻塞
- 内核就开始了IO的第一个阶段：准备数据。当内核一直等到数据准备好了，它就会将数据从内核内核缓冲区，拷贝到用户缓冲区
- 内核会给用户线程发送一个信号，或者回调用户线程注册的回调接口，告诉用户线程Read操作完成了
- 用户线程读取用户缓冲区的数据，完成后续的业务操作

相对于同步IO，异步IO不是顺序执行。

对比信号驱动IO，异步IO的主要区别在于：信号驱动由内核告诉我们何时可以开始一个IO操作(数据在内核缓冲区中)，而异步IO则由内核通知IO操作何时已经完成(数据已经在用户空间中)。



## 7 NIO

1. 非阻塞I/O: NIO采用IO多路复用技术，一个线程可以同时处理多个连接，有效避免了传统I/O模型中单线程无法处理大量请求的问题。
2. 通道（Channel）：NIO中的通道类似于传统I/O中的流，但通道是双向的，支持读写操作，并且可以异步地进行操作。
3. 缓冲区(Buffer)操作：NIO中，I/O操作是面向缓冲区的，数据需要先读取到缓冲区中再进行处理，从而提高了I/O效率。
4. 选择器（Selector）：选择器是NIO中的核心组件，用于多个监听通道（channel）上的事件，并处理这些事件。NIO使用的是事件驱动模式，通过选择器可以实现非阻塞I/O操作。

NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器、弹幕系统、服务器间通讯、编程比较复杂。

服务端：

```
1 public class NIOServer {
2     public static void main(String[] args) throws IOException {
3         // 创建一个在本地端口进行监听的服务Socket通道，并设置为非阻塞方式
4         ServerSocketChannel ssc = ServerSocketChannel.open();
5         // 必须配置为非阻塞才能往selector上注册，否则会报错，selector模式本身就是非阻塞模式
6         ssc.configureBlocking(false);
7         ssc.socket().bind(new InetSocketAddress(9000));
8         // 创建一个选择器selector
9         Selector selector = Selector.open();
10        // 把ServerSocketChannel注册到selector上，并且selector对客户端accept连接操作感兴趣
11        ssc.register(selector, SelectionKey.OP_ACCEPT);
12
13        while (true) {
14            System.out.println("awaiting events");
15            // 轮询监听channel里的key，select是阻塞的，accept()也是阻塞的
16            selector.select();
17
18            System.out.println("event occurs");
19            // 有客户端请求，被轮询监听到
20            Iterator<SelectionKey> it = selector.selectedKeys().iterator();
21            while (it.hasNext()) {
22                SelectionKey key = it.next();
23                // 删除本次已处理的key，防止下次select重复处理
24                it.remove();
25                handle(key);
26            }
27        }
28    }
29
30    private static void handle(SelectionKey key) throws IOException {
31        if (key.isAcceptable()) {
32            System.out.println("Client connected");
33            ServerSocketChannel ssc = (ServerSocketChannel) key.channel();
34            // NIO非阻塞体现：此处accept方法是阻塞的，但是这里因为是发生了连接事件，所以这个
35            // 方法会马上执行完，不会阻塞
36            // 处理完连接请求不会继续等待客户端的数据发送
37            SocketChannel sc = ssc.accept();
```

```

37         sc.configureBlocking(false);
38         //通过Selector监听Channel时对读事件感兴趣
39         sc.register(key.selector(), SelectionKey.OP_READ);
40     } elseif (key.isReadable()) {
41         System.out.println("client data readable");
42         SocketChannel sc = (SocketChannel) key.channel();
43         ByteBuffer buffer = ByteBuffer.allocate(1024);
44         //NIO非阻塞体现:首先read方法不会阻塞,其次这种事件响应模型,当调用到read方法时肯定
45         //是发生了客户端发送数据的事件
46         int len = sc.read(buffer);
47         if (len != -1) {
48             System.out.println("data from client:" + new String(buffer.array()
49             (), 0, len));
49         }
50         ByteBuffer bufferToWrite = ByteBuffer.wrap("HelloClient".
51             getBytes());
52         sc.write(bufferToWrite);
53         key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
54     } elseif (key.isWritable()) {
55         SocketChannel sc = (SocketChannel) key.channel();
56         System.out.println("writing");
57         // NIO事件触发是水平触发
58         // 使用Java的NIO编程的时候,在没有数据可以往外写的时候要取消写事件,
59         // 在有数据往外写的时候再注册写事件
60         key.interestOps(SelectionKey.OP_READ);
61         //sc.close();
62     }
63 }

```

客户端:

```

1 public class NioClient {
2     //通道管理器
3     private Selector selector;
4
5     public static void main(String[] args) throws IOException {
6         NioClient client = new NioClient();
7         client.initClient("127.0.0.1", 9000);
8         client.connect();
9     }
10
11     // 获得一个Socket通道,并对该通道做一些初始化的工作
12     public void initClient(String ip, int port) throws IOException {
13         // 获得一个Socket通道
14         SocketChannel channel = SocketChannel.open();
15         // 设置通道为非阻塞
16         channel.configureBlocking(false);
17         // 获得一个通道管理器
18         this.selector = Selector.open();
19
20         // 客户端连接服务器,其实方法执行并没有实现连接,需要在listen()方法中调
21         //用channel.finishConnect() 才能完成连接
22         channel.connect(new InetSocketAddress(ip, port));
23         //将通道管理器和该通道绑定,并为该通道注册SelectionKey.OP_CONNECT事件。
24         channel.register(selector, SelectionKey.OP_CONNECT);
25     }
26
27     // 采用轮询的方式监听selector上是否有需要处理的事件,如果有,则进行处理
28     public void connect() throws IOException {

```

```

29         // 轮询访问selector
30         while (true) {
31             selector.select();
32             // 获得selector中选中的项的迭代器
33             Iterator<SelectionKey> it = this.selector.selectedKeys().iterator();
34             while (it.hasNext()) {
35                 SelectionKey key = (SelectionKey) it.next();
36                 // 删除已选的key,以防重复处理
37                 it.remove();
38                 // 连接事件发生
39                 if (key.isConnectable()) {
40                     SocketChannel channel = (SocketChannel) key.channel();
41                     // 如果正在连接,则完成连接
42                     if (channel.isConnectionPending()) {
43                         channel.finishConnect();
44                     }
45                     // 设置成非阻塞
46                     channel.configureBlocking(false);
47                     // 在这里可以给服务端发送信息哦
48                     ByteBuffer buffer = ByteBuffer.wrap("HelloServer".
49                         getBytes());
50                     channel.write(buffer);
51                     // 在和服务端连接成功之后,为了可以接收到服务端的信息,需要给通道设置读
52                     的权限。
53                     channel.register(this.selector, SelectionKey.OP_READ);
54                     // 获得了可读的事
55                     件
56             }
57         }
58
59         // 处理读取服务端发来的信息的事件
60         public void read(SelectionKey key) throws IOException {
61             // 和服务端的read方法一样
62             // 服务器可读取消息:得到事件发生的Socket通道
63             SocketChannel channel = (SocketChannel) key.channel();
64             // 创建读取的缓冲区
65             ByteBuffer buffer = ByteBuffer.allocate(1024);
66             int len = channel.read(buffer);
67             if (len != -1) {
68                 System.out.println("Client Receives:" + new String(buffer.array
69                 (), 0, len));
70             }
71         }
72     }

```

类型	面向 操作区域	处理数据 (字节流 & 字符流)	
Java IO	直接 面向 最初的数据源	<ul style="list-style-type: none"> <li>每次读取时 = 读取所有字节 / 字符，无缓存</li> <li>无法前后移动读取流中的数据</li> </ul>	阻塞 <ul style="list-style-type: none"> <li>当1个线程在读 / 写时：当 数据被完全读取 / 写入完毕前 &amp; 数据未准备好时，线程不能做其他任务，只能一直等待，直到数据准备好后继续读取 / 写入，即阻塞</li> <li>当线程处于活跃状态时 &amp; 外部未准备好时，则阻塞</li> </ul>
Java NIO	面向缓冲区	<ul style="list-style-type: none"> <li>先将数据读取 到 缓存区</li> <li>可在缓冲区中前后移动流数据</li> </ul>	非阻塞 <ul style="list-style-type: none"> <li>当1个线程向某通道发送请求 要求 读 / 写时，当 数据被完全读取 / 写入完毕前 &amp; 数据未准备好时，线程可以做其他任务（控制其他通道），直到数据准备好后再 切换回该通道，继续读取 / 写入，即选择器（Selector）的使用</li> <li>外部准备好时才唤醒线程，则不会阻塞</li> </ul>

## 7.1 NIO读写

读：

```

1 import java.io.*;
2 import java.nio.*;
3 import java.nio.channels.*;
4
5 public class Program {
6     static public void main( String args[] ) throws Exception {
7         FileInputStream fin = new FileInputStream("<span style="font-family:FangSong_GB2312;">f</span>:\\<span style="font-family:FangSong_GB2312;">data</span>.txt");
8
9         // 获取通道
10        FileChannel fc = fin.getChannel();
11
12        // 创建缓冲区
13        ByteBuffer buffer = ByteBuffer.allocate(1024);
14
15        // 读取数据到缓冲区
16        fc.read(buffer);
17
18        //模式转换为读取模式
19        buffer.flip();
20
21        while (buffer.remaining() > 0) {
22            byte b = buffer.get();
23            System.out.print(((char)b));
24        }
25
26        fin.close();
27    }
28}

```

写：

```

1 import java.io.*;
2 import java.nio.*;
3 import java.nio.channels.*;
4
5 public class Program {
6     static private final byte message[] = { 83, 111, 109, 101, 32,
7         98, 121, 116, 101, 115, 46 };
8
9     static public void main( String args[] ) throws Exception {
10         FileOutputStream fout = new FileOutputStream(" "<span style="font-family:FangSong_GB2312;">f</span>:\\<span style="font-family:FangSong_GB2312;">data</span>.txt");
11
12         // 将缓冲区中的数据写入文件
13         fout.write(message);
14
15         fout.close();
16    }
17}

```

```

11     FangSong_GB2312; ">data</span>.txt" );
12
13     FileChannel fc = fout.getChannel();
14
15     ByteBuffer buffer = ByteBuffer.allocate( 1024 );
16
17     for ( int i=0; i<message.length; ++i ) {
18         buffer.put( message[i] );
19     }
20
21     buffer.flip();
22
23     fc.write( buffer );
24
25     fout.close();
26 }
```

## 7.2 ByteBuffer

缓冲区本质上是一块可以写入数据，以及从中读取数据的内存，实际上也是一个byte[]数据，只是在NIO中被封装成NIO Buffer对象，并提供了一组方法来访问这个内存块。其属性有：

- Position: 指示当前在缓冲区中的读取或写入位置。
- Limit: 指示可读或可写数据的上限。
- Capacity: 缓冲区可以容纳的总字节数。



## 第二次读取数据



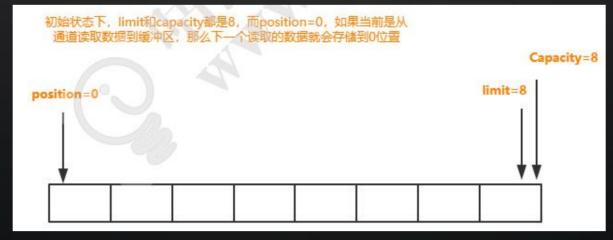
## flip操作



## 数据写出



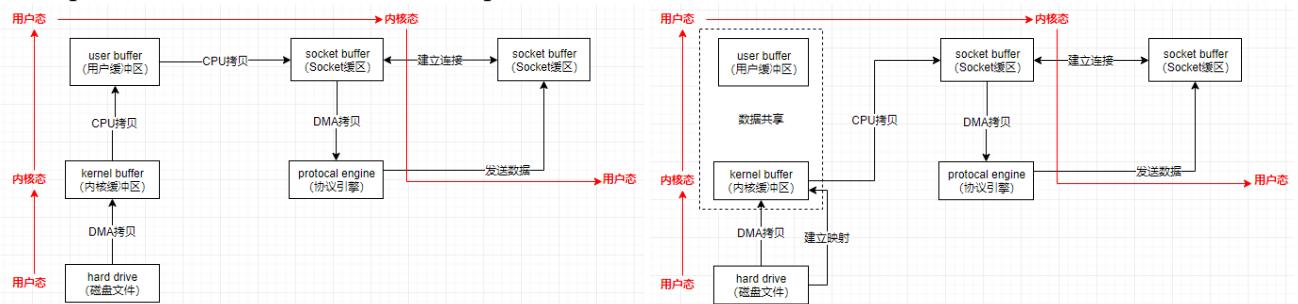
## Clear



## 7.3 零拷贝

### 7.3.1 MMAP

mmap 是一种内存映射技术，mmap 相比于传统的IO 来说，少了1 次CPU 拷贝。

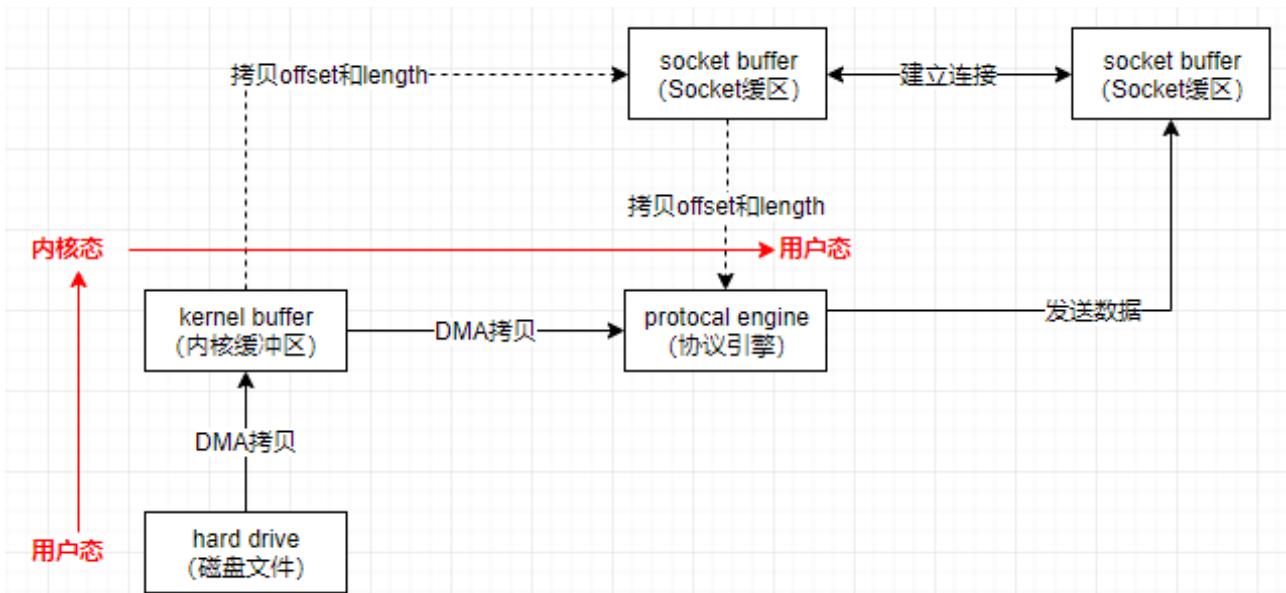


左为传统IO，右为MMAP。

传统IO 里面从内核缓冲区到用户缓冲区有一次CPU 拷贝，从用户缓冲区到Socket 缓冲区又有一次CPU 拷贝。mmap 则一步到位，直接基于CPU 将内核缓冲区的数据拷贝到了Socket 缓冲区。

之所以能够减少一次拷贝，就是因为mmap 直接将磁盘文件数据映射到内核缓冲区，这个映射的过程是基于DMA 拷贝的，同时用户缓冲区是跟内核缓冲区共享一块映射数据的，建立共享映射之后，就不需要从内核缓冲区拷贝到用户缓冲区了。

### 7.3.2 sendFile



可以看到在图中，已经没有了用户缓冲区，因为用户缓冲区是在用户空间的，所以没有了用户缓冲区也就意味着不需要上下文切换了，就省略了这一步的从内核态切换为用户态。

同时也不需要基于CPU 将内核缓冲区的数据拷贝到Socket 缓冲区了，只需要从内核缓冲区拷贝一些offset 和length 到Socket 缓冲区。接着从内核态切换到用户态，从内核缓冲区直接把数据拷贝到网络协议引擎里去；同时从Socket 缓冲区里拷贝一些offset 和length 到网络协议引擎里去，但是这个offset 和length 的量很少，几乎可以忽略。

sendFile 整个过程只有两次上下文切换和两次DMA 拷贝，很重要的一点是这里完全不需要CPU 来进行拷贝了，所以才叫做零拷贝，这里的拷贝指的就是操作系统的层面。

上下文切换(Context Switch): 挂起一个进程，将这个进程在CPU中的状态（上下文信息）存储于内存的PCB中。在PCB中检索下一个进程的上下文并将其在CPU的寄存器中恢复。

## 7.4 Reactor Pattern

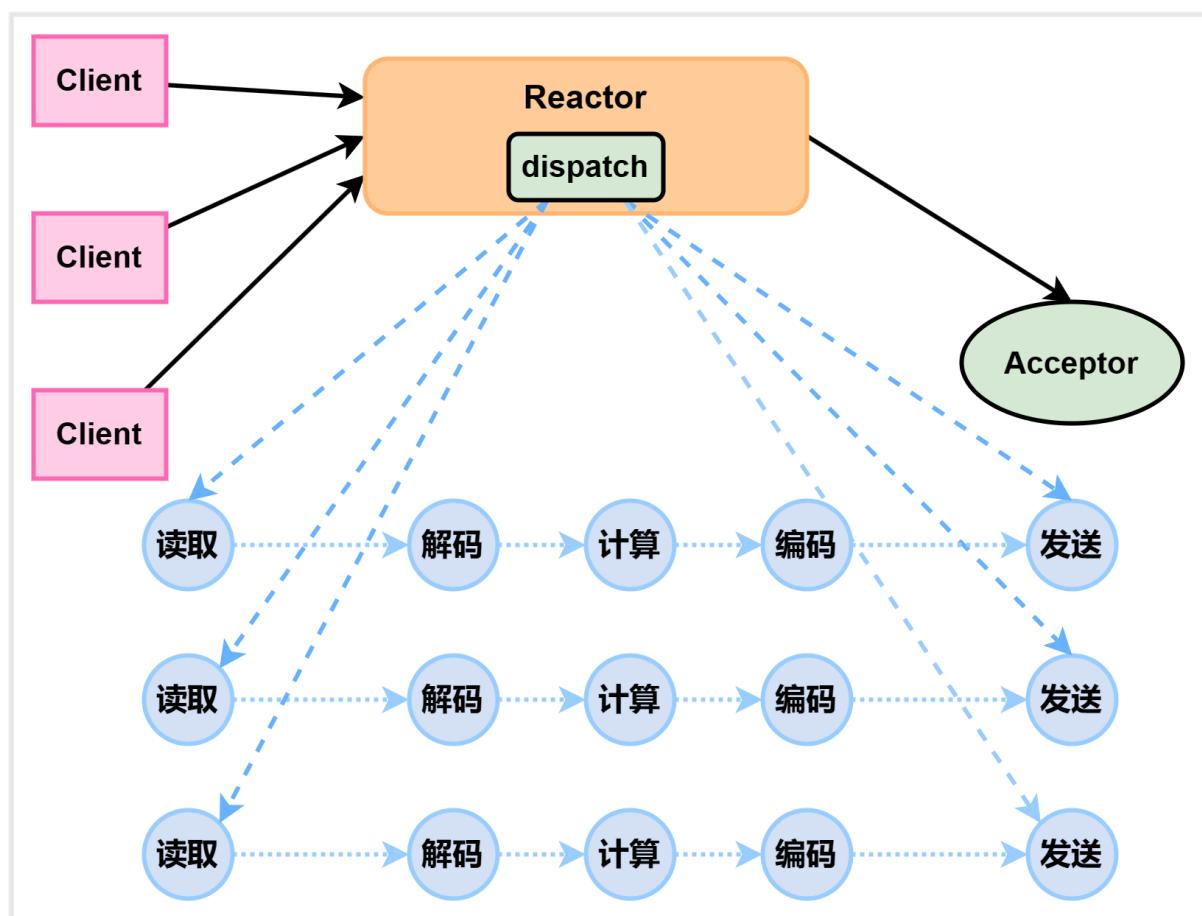
Reactor 即“反应器”，是应用最广泛的一种I/O 多路复用技术。当需要等待I/O 操作时，首先释放资源。一旦等待完成，便通过事件驱动的方式继续进行后续工作。以下是Reactor 模型中的五个重要角色：

- Handle （句柄或描述符）：它是资源在操作系统层面的一种抽象，表示与事件绑定了的资源，即各种SocketChannel。
  - Demultiplexer 的本质是一个系统调用，用于等待事件的发生。调用方在调用它后会被阻塞，一直阻塞到Demultiplexer 上有事件就绪为止。
  - 在Linux 中，同步事件分发器指的就是I/O 多路复用器，比如select、poll、epoll 等，Java NIO 中的Selector 就是对多路复用器的封装。
- Reactor （反应器）：事件管理的接口，内部使用Synchronous Event Demultiplexer 注册、注销Event Handler，当有事件进入“就绪”状态时，调用注册事件的回调函数处理事件。

- Event Handler (事件处理器接口)：事件处理器程序提供了一组接口，在Reactor 监听到相应的事件发生时调用，执行相应的事件处理。
  - 比如当Channel 被注册到Selector 时的回调方法、连接事件发生时的回调方法、写事件发生时的回调方法等都是事件处理器，我们可以实现这些回调来达到对某一事件进行特定反馈的目的。
  - 原生的Java 并不支持Event Handler，实际业务中需要自己实现，或使用Netty 等网络框架。
- Concrete Event Handler (事件处理器实现)：它是Event Handler 的实现类，用于实现回调方法指定的业务逻辑。

Reactor 模型有三种模型，分别是：单Reactor 单线程模型、单Reactor 多线程模型和主从Reactor 多线程模型。

#### 7.4.1 单Reactor 单线程模型



在上图中：

- Acceptor 专门处理连接事件，而Selector 则充当同步事件分发器。
- 客户端的请求可以分为连接请求和其他事件请求两种。
  - Selector 上注册了一系列的Channel，它不断监听这些Channel。

- 一旦某个Channel 上的事件处理器就绪， Selector 就会将该事件分发给事件处理器。

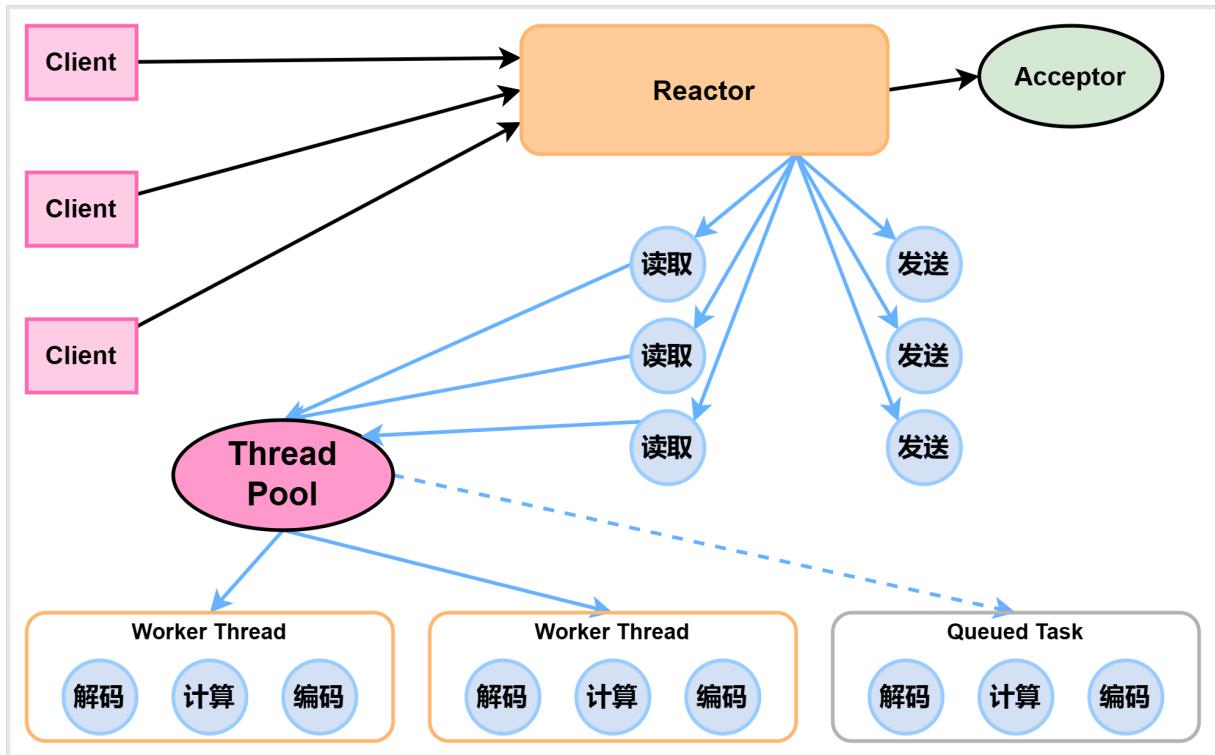
该模型仅依靠单线程处理请求，主循环承担了太多的任务，容易在高并发情境下造成请求积压甚至超时。此外，单线程无法有效利用多核资源。因此，更合适的做法是为解码、计算和编码操作引入额外的线程，并使用线程池进行管理。

```

1 public class ReactorServer {
2
3     private final Selector selector;
4
5     public ReactorServer() throws IOException {
6         ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
7         ;
8         serverSocketChannel.configureBlocking(false);
9         ServerSocket serverSocket = serverSocketChannel.socket();
10        serverSocket.bind(new InetSocketAddress(1234));
11
12        selector = Selector.open();
13        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
14    }
15
16    public class Reactor {
17        public void run() {
18            try {
19                // Reactor 循环
20                while (true) {
21                    selector.select();
22                    Set<SelectionKey> selectionKeys = selector.selectedKeys()
23                    ();
24                    Iterator<SelectionKey> iterator = selectionKeys.iterator()
25                    ();
26                    while (iterator.hasNext()) {
27                        SelectionKey selectionKey = iterator.next();
28                        iterator.remove();
29                        if (selectionKey.isAcceptable()) {
30                            // 处理连接事件
31                            handleAcceptEvent(selectionKey);
32                        } else if (selectionKey.isReadable()) {
33                            // 处理可读事件
34                            handleReadEvent(selectionKey);
35                        } else if (selectionKey.isWritable()) {
36                            // 处理可写事件
37                            handleWriteEvent(selectionKey);
38                        }
39                    }
40                }
41            } catch (IOException e) {
42                throw new RuntimeException(e);
43            }
44        }
45
46        // 其他功能
47    }
48
49    public static void main(String[] args) throws IOException {
50        ReactorServer server = new ReactorServer();
51        System.out.println("Server start ...");
52        Reactor reactor = server.new Reactor();
53        reactor.run();
54    }
55}
```

#### 7.4.2 单Reactor 多线程模型

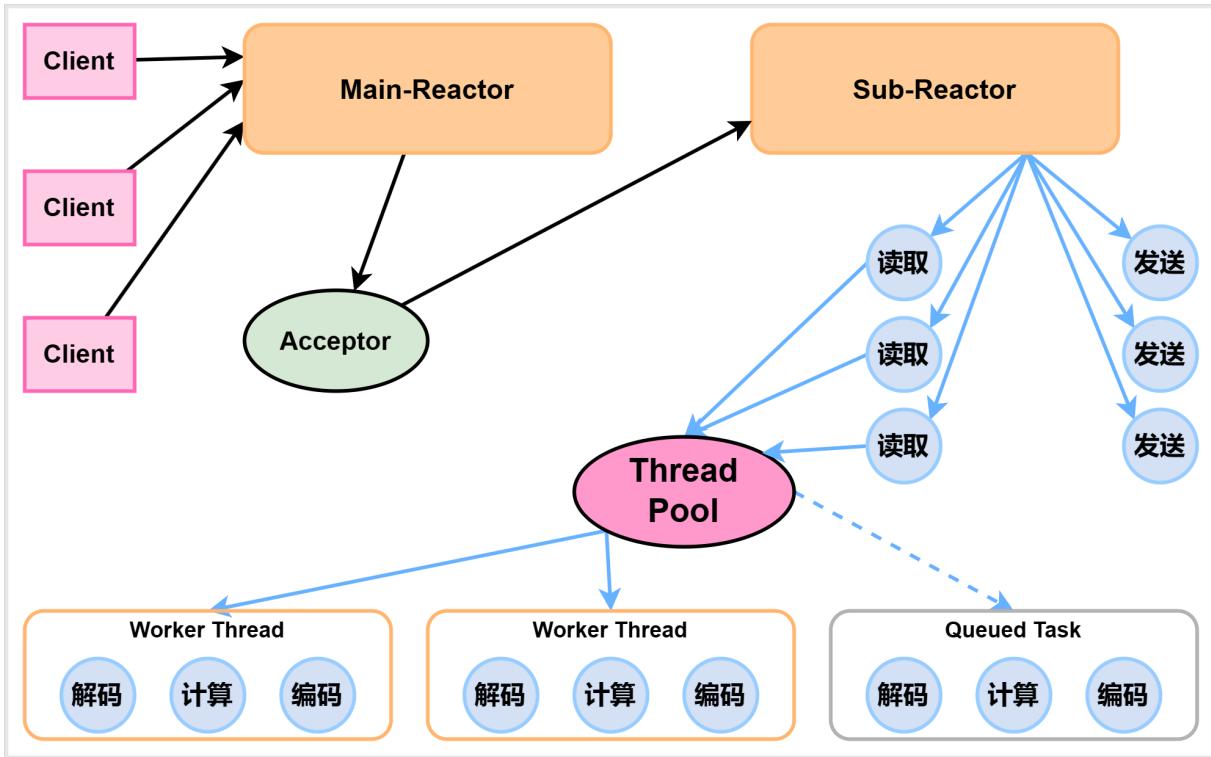
单Reactor 多线程模型是指仅有一个线程负责执行I/O 操作和处理连接请求，其他逻辑均由Worker 线程执行。其架构图如下：



与第一种模型相比，单Reactor 多线程模型将业务逻辑委托给线程池来处理，从而可以更有效地利用多核CPU 资源。然而，单个线程的Reactor 仍负责监听和响应所有事件，这在高并发环境下仍可能产生性能瓶颈。因此，主从Reactor 多线程模型应运而生。

#### 7.4.3 主从Reactor 多线程模型

在客户端连接众多且频繁进行I/O 操作的情况下，单Reactor 模型就会暴露出问题。因为Reactor 不支持异步I/O 操作，这意味着当Reactor 处理读写事件时，其他客户端的连接操作可能无法得到及时处理。主从Reactor 多线程模型就是专门用来解决这个问题的：



该模型将处理连接事件的Reactor 与处理读写事件的Reactor 分离，避免了读写事件较为频繁的情况下影响新客户端连接。

主从Reactor 多线程模型中存在多个Reactor，Main-Reactor 一般只有一个，它负责监听和处理连接请求；而Sub-Reactor 可以有多个，用线程池进行管理，主要负责监听和处理读写事件等。当然Main Reactor 也可以多个，也通过线程池管理，但是这样会增加系统复杂度，需要合理规划调度，否则反而会拖累性能。

## 7.5 Selector

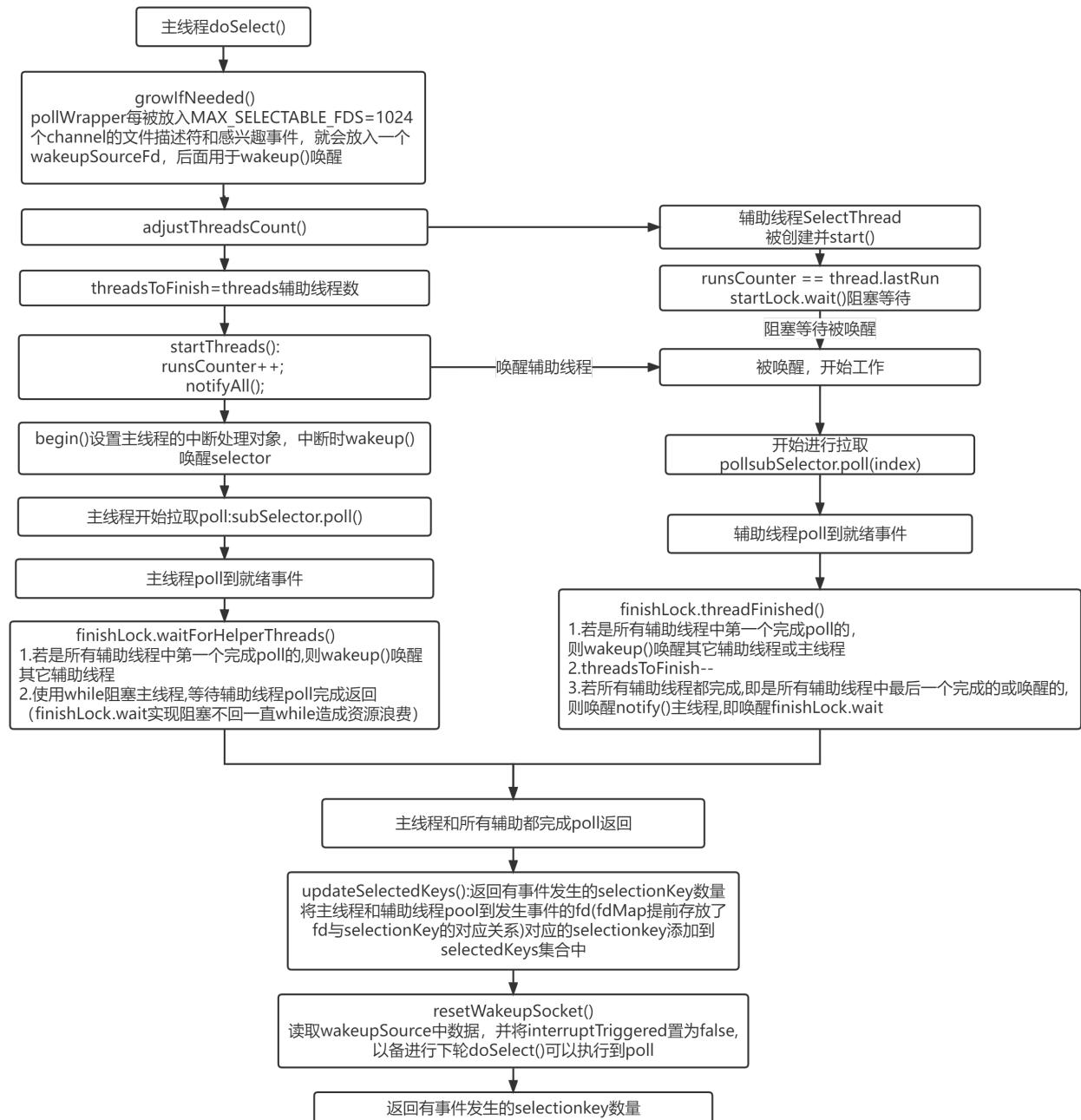
### 7.5.1 Selector.open()

1. 创建WindowsSelectorImpl对象时，创建了Pipe管道作为wakeupPipe，并保存pipe的sink和source通道文件描述符，并将source通道的数据和感兴趣事件Net.POLLIN存入到pollWrapper中，后面用于selector的唤醒
2. Pipe作用：pipe是通过两个连接的socket组成，sink和source，当sink有数据写入时，就可以从source中读取往sink写入的数据了
3. wakeupPipe主要用于唤醒selector.select()所在的线程

### 7.5.2 channel.register(Selector sel, int ops, Object att)

1. 创建selectionKey，selectionKey可以看成是channel，事件，selector的映射
2. register主要就是将创建的selectionKey放入SelectorImpl的keys集合中，供后面的selector.select()使用

### 7.5.3 selector.select()



### 7.5.4 selector.selectedKeys()

public SelectedKeys 即为 selectedKeys 封装的 set(构造函数中实现)，调用 selector.selectedKeys().iterator() 即调用

1. doSelect() 时，所有线程都 poll 完成后，会调用 updateSelectedKeys(action) -> SubSelector.processSelectedKeys -> SubSelector.processFDSet -> SelectorImpl.processReadyEvents 将就绪的 selectionKey 放入 selectedKeys 集合

2. selector.selectedKeys() 从 selectedKeys 中获取就绪 key

在每轮处理就绪 selectionKeys 时就 iterator.remove()，以防同一个 key 被重复处理

## 7.6 epoll

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）

当某一进程调用`epoll_create`方法时，Linux内核会创建一个`eventpoll`结构体，这个结构体中有两个成员与epoll的使用方式密切相关。`eventpoll`结构体如下所示：

```
1 #include <sys/epoll.h>
2
3 // 数据结构
4 // 每一个epoll对象都有一个独立的eventpoll结构体
5 // 用于存放通过epoll_ctl方法向epoll对象中添加进来的事件
6 // epoll_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可
7 struct eventpoll {
8     //红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件
9     struct rb_root rbr;
10    //双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件
11    struct list_head rdlist;
12};
13
14 // API
15 int epoll_create(int size); // 内核中间加一个ep 对象，把所有需要监听的socket 都放到ep 对象中
16 int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event); // epoll_ctl 负责把socket 增加、删除到内核红黑树
17 int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout); // epoll_wait 负责检测可读队列，没有可读socket 则阻塞进程
```

每一个epoll对象都有一个独立的`eventpoll`结构体，用于存放通过`epoll_ctl`方法向epoll对象中添加进来的事件。这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为红黑树元素个数)。

而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当相应的事件发生时会调用这个回调方法。这个回调方法在内核中叫`ep_poll_callback`,它会将发生的事件添加到rdlist双链表中。

在epoll中，对于每一个事件，都会建立一个`epitem`结构体，如下所示：

```
1 struct epitem{
2     struct rb_node rbn;//红黑树节点
3     struct list_head rdllink;//双向链表节点
4     struct epoll_filefd ffd;//事件句柄信息
5     struct eventpoll *ep;//指向其所属的eventpoll对象
6     struct epoll_event event;//期待发生的事件类型
7 }
```

当调用`epoll_wait`检查是否有事件发生时，只需要检查`eventpoll`对象中的`rdlist`双链表中是否有`epitem`元素即可。如果`rdlist`不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户。