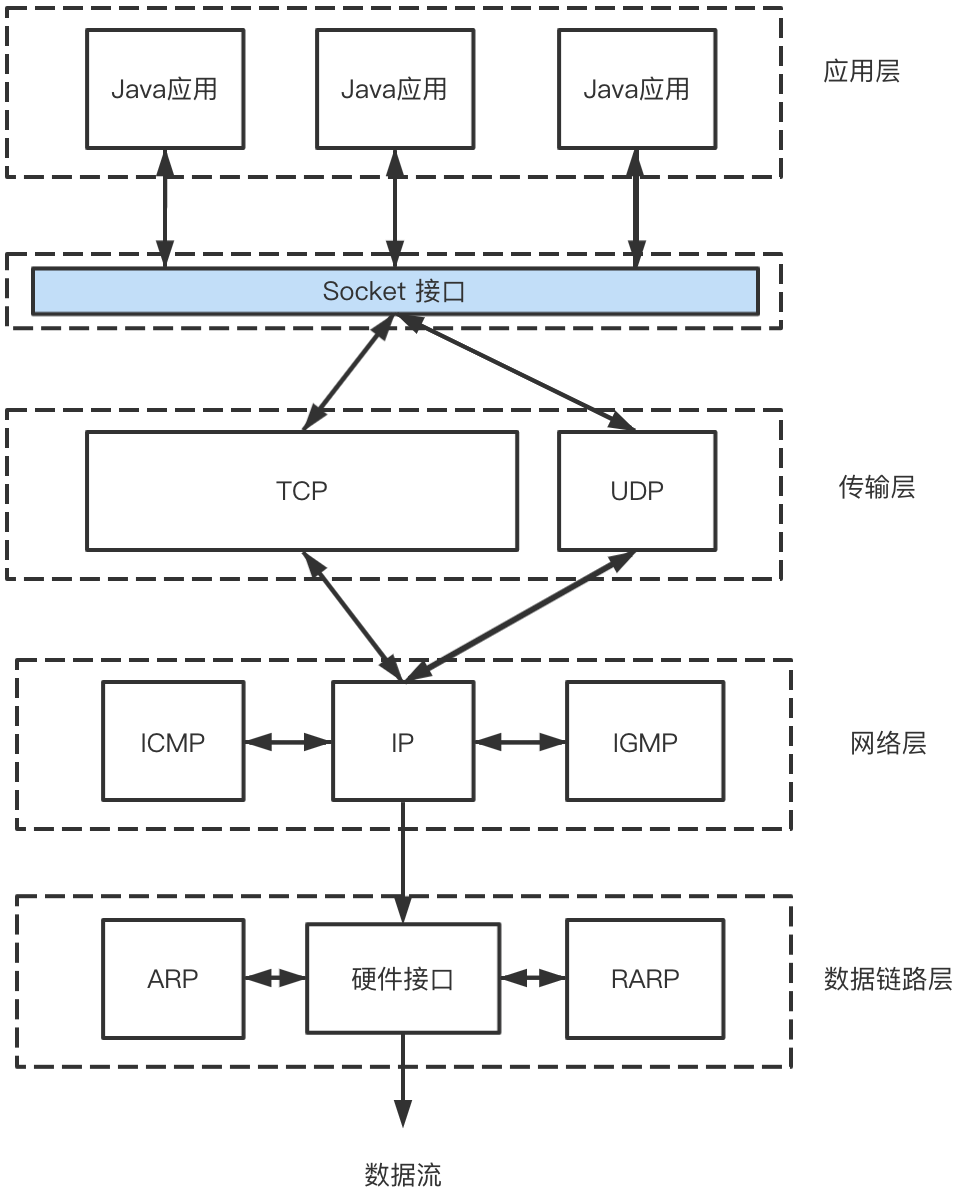
网络编程 — Socket编程与IO模型

前言

前面的两小节，我们介绍了网络的基础知识包括一些网络模型和协议。我们常说Talk is cheap,show me code.这一小节我们将解锁Socket网络编程，通过Socket来编写简单一些简单的网络小程序。我很早之前就接触过Socket网络编程，也能写一些简单的程序，但是始终感觉socket离我很遥远，不清楚socket到底是什么以及socket是如何工作的。如果你也有和我一样的疑问，通过这一小节的梳理让我们一起搞懂socket，解锁socket技能点。在编写完socket程序之后，我们还会思考如何提高我们socket网络程序的处理吞吐量连接数，并且引入我们的IO模型，深入理解几种常见的IO模型。

什么是Socket

Socket这个名字很有意思，在很多书中翻译为套接字，我以前看到套接字总是感觉很陌生难以理解。Socket在英语中是插座，插槽的意思。我们写的Socket虽然是代码，但是可以想象成**我们的服务端和客户端中间有“一根网线并且两头有插座”**。我们的程序只要适配这个插座，两个端之间就可以相互通信了，而这个“插座”就是我们的Socket。Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。这一层加入到我们的网络模型后的效果如下图所示。



在上面的图中不难看出Socket接口是位于传输层上方的一个中间层，**通过这个中间层提供的接口，Java应用可以进行网络通信**，这也体现了“socket插座”的字面意思。

代码示例以及调用过程

我们常说talk is chep，show me code。在这个部分，将使用Socket实现一个HTTP服务器，还会用Scoket实现一个互相通信的Server和Client。从代码的角度看看socket是怎么操作对接上下层，分析在

TCP下和UDP下Socket的调用处理逻辑。

代码示例

以下是使用Socket实现的一个简单的HTTP服务。

Java 复制代码

```java
package com.daiwei.socket.http;

import java.io.IOException;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Created by Daiwei on 2021/9/9
 */
public class HttpServer {

    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(8081);
        while (true) {
            Socket socket = serverSocket.accept();
            service(socket);
        }
    }

    /**
     * socket 处理服务
     * @param socket
     */
    private static void service(Socket socket) {
        try {
            String body = "hello socket";
            StringBuilder response = new StringBuilder();
            response.append("HTTP/1.1 200 OK\r\n")
                    .append("Content-Length:
").append(body.getBytes().length).append("\r\n")
                    .append("Content-Type: text/plain; charset-utf-8\r\n")
                    .append("\r\n")
                    .append(body).append("\r\n");

            OutputStream outputStream = socket.getOutputStream();
            outputStream.write(response.toString().getBytes());
            outputStream.flush();

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

使用`wrk`进行简单的压测，可以得到以下的结果。

```Bash
daiwei@daiweideMacBook-Pro ~ % wrk -t2 -c10 -d10s http://localhost:8081
Running 10s test @ http://localhost:8081
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     2.59ms   14.88ms 213.08ms   96.55%
    Req/Sec     9.91k     2.65k   11.99k    87.50%
  192565 requests in 10.03s, 16.90MB read
  Socket errors: connect 0, read 191661, write 903, timeout 0
Requests/sec:  19192.30
Transfer/sec:      1.68MB
```

我们还可以使用Socket实现客户端和服务端间简单的通行，以下的代码是客户端client向服务端server连接并发送一段message。

```Java
package com.daiwei.socket.app;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Created by Daiwei on 2021/9/9
 */
public class SocketAppServer {

    public static void main(String[] args) {
        int port = 8888;
        try {
            ServerSocket serverSocket = new ServerSocket(port);
            System.out.println("server is running and listening at " + port);
            while (true) {
                Socket socket = serverSocket.accept();
                InputStream is = socket.getInputStream();
                BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(is));
                String res;
                while ((res= bufferedReader.readLine()) != null) {
                    System.out.println("message checked from [" + res + "]");
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Java 复制代码

```java
package com.daiwei.socket.app;

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

/**
 * Created by Daiwei on 2021/9/9
 */
public class SocketAppClient {

    public static void main(String[] args) {
        for (int i = 0; i < 200; i++) {
            socketSendMsg("hello server of " + i);
        }
    }

    /**
     * 通过socket 发送消息到server端
     * @param msg
     */
    private static void socketSendMsg(String msg) {
        try {
            Socket socket = new Socket("127.0.0.1", 8888);
            OutputStream outputStream = socket.getOutputStream();
            PrintWriter printWriter = new PrintWriter(outputStream);
            printWriter.write(msg);
            printWriter.flush();

            printWriter.close();
            outputStream.close();
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Socket调用过程分析

在上面的代码中我们也不难发现，socket建立的连接的参数是非常简洁的，客户端只需要提供一个目标主机IP和端口号，服务端则需要提供一个服务监听的端口号即可。socket编程进行是端到端的通行。承接上面的应用层，对接下面的传输层和网络层。针对网络层，socket需要指定是`IPv4`还是`IPv6`。分别对应这实现类`java.net.Inet4Address`和`java.net.Inet6Address`。我们前面提到过，TCP协议是基于数据流的而UDP是基于数据报的。在socket源码中有以下的一个私有构造函数：

```java
private Socket(SocketAddress address, SocketAddress localAddr,
               boolean stream) throws IOException
```

其中当参数stream设置为`true`时，当前socket使用则是TCP协议，反正则为UDP协议。但是我发现调用这个构造的**未被废弃**的方法stream参数都是true。这就意味着我们创建的socket都是基于TCP的。有两个**废弃了**的构造方法可以传入stream参数指定使用UDP协议。并且在注释中可以发现以下一段说明。

> If UDP socket is used, TCP/IP related socket options will not apply.
> Deprecated Use DatagramSocket instead for UDP transport.
> ---
> 如果使用UDP socket被使用，那么TCP/IP有关的一些socket配置参数将不生效，
> 废弃使用数据报文Socket代替UDP数据传输。

基于TCP协议的调用过程分析

两端创建了socket之后，接下来的过程中，TCP和UDP稍有不同。TCP的服务端要监听一个端口，一般是先调用`bind()`函数，给这个socket赋予一个IP地址和端口。为什么需要端口？当一个数据包到达后，内核要通过TCP头的端口号找到这个数据包所属的应用程序。如果一台机器有多个IP地址，我们可以选择监听所有网卡，也可以监听一个网卡（监听`0.0.0.0`即可监听所有网卡）。当服务端有了IP和端口号，就可以调用`listen()`函数进行监听。在TCP的状态图里面，有一个`listen`状态，当调用这个函数之后，服务端就进入了这个状态。这个时候客户端就可以发起连接。接下来，服务端调用`accept()`函数，拿出一个已经完成连接进行处理。

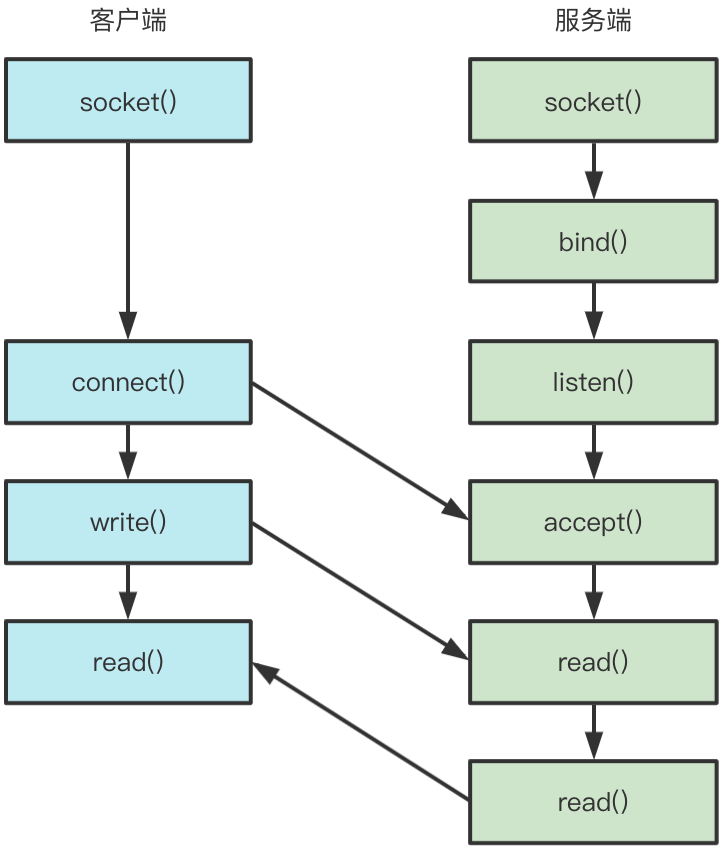
> 在内核中，为每个socket维护两个队列，一个是已经建立了连接的队列，这时候连接三次握手已经完成，处于`establish`状态；一个是还没有完全建立连接的队列，这个时候三次握手还没完成，处于`syn_rcvd`的状态。

在服务端等待过程中，客户端可以通过`connect`函数发起连接。先在参数中要明确连接的IP地址和端口号，然后发起三次握手。内核会给客户端分配一个临时的端口。一旦握手成功，服务端的`accept`就会返回另一个socket。

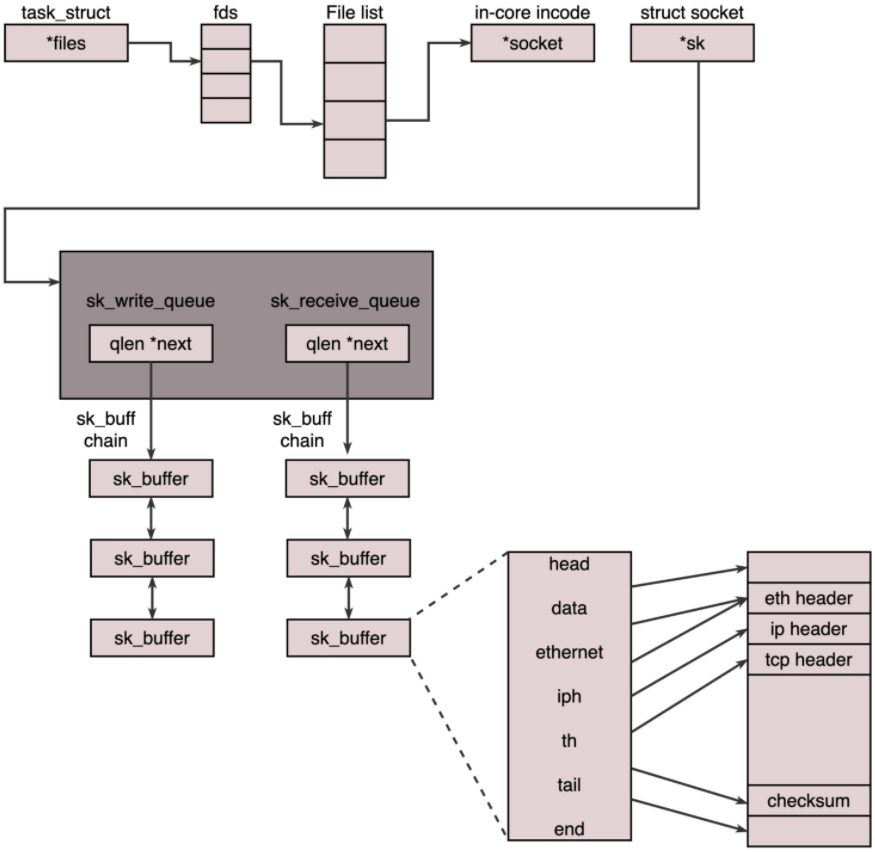
> 用来监听的socket和真正用来传数据的socket是两个，一个叫做**监听socket**，一个叫做**已经连接socket**。
> 客户端：
> `scoket()` -> 创建`active_socket_fd`（`client_socket_fd`）
> `bind()` -> 把`active_socket_fd`与`ip, port`绑定起来。
> `connect()` -> `client_socket_fd`主动请求服务端的`listen_socket_fd`
> `read()/write()` -> 读/写`socket io`
> `close()` -> 关闭`socket_fd`

> 服务端：
> `socket()` -> 创建`active_socket_fd`
> `bind()` -> 把`active_socket_fd`与`ip, port`绑定起来
> `listen()` -> `active_socket_fd` -> `listen_socket_fd` 等待客户端的`client_socket_fd`来请求连接。
> `accept()` -> `listen_socket_fd` -> `connect_socket_fd` 把监听socket转变为连接socket，用于建立连接通道的数据读写。
> `read()/write()` -> 读/写 `socket io`
> `close()` -> 关闭`socket_fd`
>
> 为什么需要两种状态的socket?
> 现在的网络程序中是C/S结构，一般是客户端主动向服务端请求建立连接。这个过程中，主要涉及两个状态，一个是主动一个是被动的。因此，客户端的socket只用于主动服务端的socket请求建立连接，服务器端的socket一直被动的等待客户端的请求连接就ok了。所以这就解答了为什么需要两种状态的socket，**只有一个方是主动的，另一方是被动的才能完成上述操作，如果双方都是主动或是被动的，就完成不了上面的过程。**

在成功建立连接之后，双方开始通过`read()`和`write()`函数来读写数据，就像是往一个文件流里面写东西一样。下面的图就是基于TCP的socket程序函数调用过程。

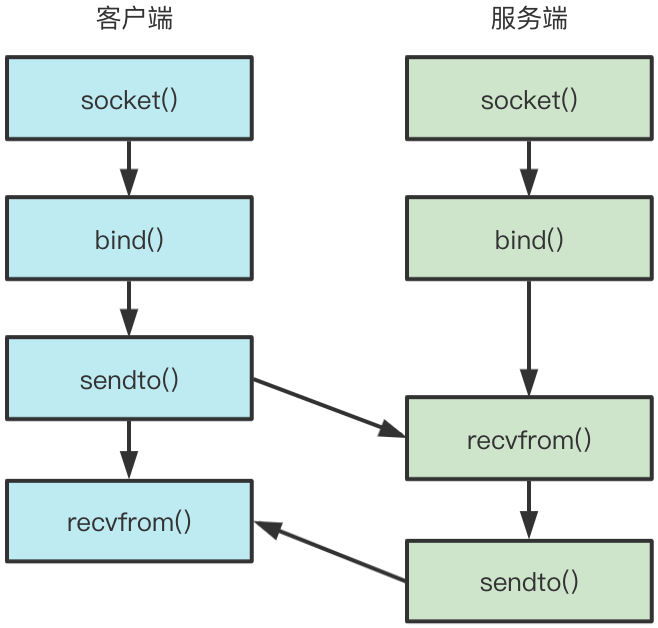


说TCP的socket就是一个文件流，是非常准确的。因为socket在Linux中是以文件的形式存在的。除此之外，还存在文件描述符。写入和读出，也是通过文件描述符。在内核中，**socket是一个文件，那对应的就有文件描述符**。每一个进程都有一个数据结构`task_struct`，里面指向一个文件描述符数组。来列出这个进程打开的所有文件的文件描述符。**文件描述符是一个整数，是这个数组的下标**。其中数组的内容是一个指针，值向内核中所有打开的文件的列表，既然是一个文件，就会有一个inode，只不过Socket对应的inode保存在内存中，而不是像真正的文件系统保存在硬盘上。在这个inode中，指向Socket在内核中的Socket结构。在这个结构中，主要有两个队列，**一个是发送队列，一个是接收队列**。在这两个队列里面保存的是一个缓存**sk_buff**。这个里面能看到完整的包结构。



基于UDP协议的调用过程分析

对于UDP来讲，过程有一些不一样。**UDP是没有连接的，所以不需要三次握手**，也就不需要调用 listen 和 connent，但是UDP的交互仍然需要IP和端口号，因而也需要`bind`操作。UDP是没有维护连接状态的，因而**不需要每队连接都建立一组Socket**，而是只要一个Socket就能够和多个客户端通行。也正是因为没有连接状态，每次通信的时候，都调用`sendto`和`recvfrom`，都可以传入IP地址和端口，下图内容就是基于UDP协议的Socket函数调用过程。



更多的连接

在完成上述的调用过程分析之后，我们可以用socket写出一个网络交互的程序了。在我们生产环境中，我们往往要服务要接入很多的服务。在上面的代码中，我们服务端通过`accept()`监听socket连接，当客户端的socket连接后。双方进行连接服务器处理逻辑然后将结果通过socket写回客户端。这个过程中，服务端只能处理这一个连接，不能处理其他的socket连接，其他的socket连接只能排队等待处理，这样的处理模式很明显是不能满足我们的期望。

在我们尝试提升连接数之前，我们先计算下理论值，也就是最大连接数。系统会使用一个四元组来标识唯一一个TCP连接。

```Java
{本机IP，本机端口PORT ： 对端IP，对端端口PORT}
```

服务器通常固定在某个本地端口上监听，等待客户端的连接请求。因此服务端TCP连接四元组中只有对端IP，也就是客户端的IP和对端的端口。因此**最大TCP连接数=对端IP * 对端端口数**。对IPv4，客户端的IP数最多为2^32。客户端的端口数最多为2^16，也就是服务端单机最大TCP连接数为2^48。这只是理论值，实际上并不会有这么多连接数。其中**最大的限制是文件文件描述符**。我们前面分析过socket都是文件，所以首先要通`ulimit`配置文件描述符的数量。其次系统内存的限制，按照上面的分析，每个TCP连接都需要**占用一定的内存，内存资源也是有限**的。在有限的资源限制下，连接尽可能多的客户端。我们可以从以下四个方式去优化：

- **多进程方式**，如果监听到有新的请求进来就创建一个子进程，然后将基于已经连接socket交给这个新的进程来处理。

> 在Linux下，使用fork函数创建子进程。**在父进程的基础上完全拷贝一个子进程**。在Linux内核中，会复制文件描述符列表，也会复制内存空间，还会复制一条记录当前执行到了哪一行程序的进程。在复

> 制完成后，父子进程几乎一摸一样，只是根据fork的返回值来区分是父进程还是子进程。如果返回0则是子进程，其他整数则是父进程。

- **多线程方式**，我们很容易就容易想到使用多线程的方式提升性能。相较于进程来说。线程轻量的多。在linux下，通过pthread_create创建一个线程，也是调用do_fork。不同的是，虽然新的线程在task列表会新创建一项，但是很多资源都是可以共享，比进程方式轻量的多。新的连接可以通过多线程的方式快速处理，从而避免监听线程被阻塞。但是如果是一个台机器要维护1万个连接，就要创建1万个进程或者线程，操作系统是无法承受的，这也就是C10K问题。
- **IO多路复用，一个线程维护多个socket**，由于socket是文件描述符，某个线程轮训所有的socket都放在一个文件集合fd_set中。调用`select`函数来监听文件描述符在fd_set对应的卫视都设为1，标识socket可读或可写，从而可以进行读写操作，然后再调用select继续轮训下去。
- **事件驱动IO**，上面select函数还是存在问题，因为每次socket所在的文件描述符集合中有socket发生变化的时候，都需要通过轮询的方式，也就是需要将全部的项目都过一遍的方式查看进度，这大大影响了一个项目组能支撑的最大项目数量。因为使用select，能够同时查询的项目数量由`FD_SETSIZE`限制。如果改成事件通知的方式，从原来主动去查询socket的状态到被动通知socket可读或可写，性能将极大提升。也就是我们经常提到的`epoll`函数。在Linux内核中的实现不是通过轮训的方式，而是通过**注册callback函数的方式**，当某个文件描述符号发生变化的时候，就会主动通知。

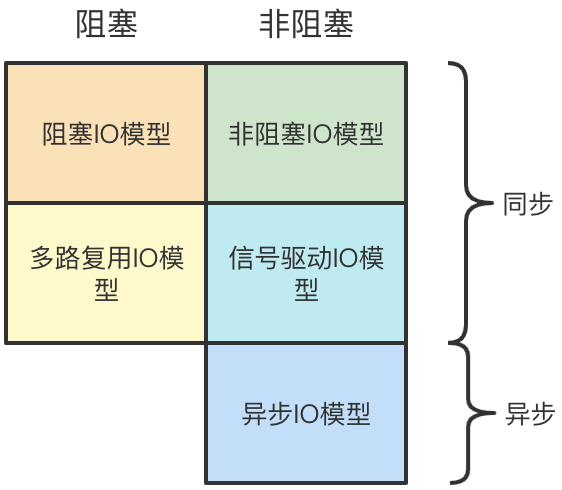
> 事件驱动IO使得监听Socket数量增加，但是效率不会大幅降低，能同时监听的Socket的数量也非常多，上限就为系统定义的、进程打开的最大文件表述符个数。因而，**epoll被称为解决C10K问题的利器。**

IO模型

我们梳理网络的过程中必然会介绍网络模型。因为网络通信也是最常见的输入输出IO之一。在计算机世界中，我们将输入输出过程高度抽象可以描述为这样一个过程，即通过外部条件或数据的输入，然后经过系统的逻辑处理之后产生新的结果并将其输出。IO模型描述的计算机世界输入和输出经过的抽象模型。

五种IO模型概述

我们熟知的IO有阻塞，非阻塞，同步，异步这几种类型。依据这几种类型我们可以划分出五种IO模型。他们分别为`阻塞IO(BIO)`、`非阻塞IO(NIO)`、`多路复用IO(multiplexing IO)`、`信号驱动IO(signal-driven IO)`、`异步IO(AIO)`。每一种IO都有他们的使用场景和优势，以下这张图各个IO和阻塞非阻塞，同步异步之间的关系。



我们不难发现这些IO模型中，绝大多数都是同步的，只有异步模型是异步的。在深入梳理每一种IO模型之前，我们要先明确模型中阻塞和同步的定义。

> I/O 操作分为两个部分：
>
> 1. 数据准备，将数据加载到内核缓存。（数据加载到操作系统）
> 2. 将内核缓存中的数据数据加载到用户缓存（从操作系统复制到应用中）

异步同步的概念描述的是用户线程和内核线程之间的交互方式，而阻塞和非阻塞描述的是用户线程调用内核IO的操作方式。同时只有同步才有阻塞和非阻塞之分。

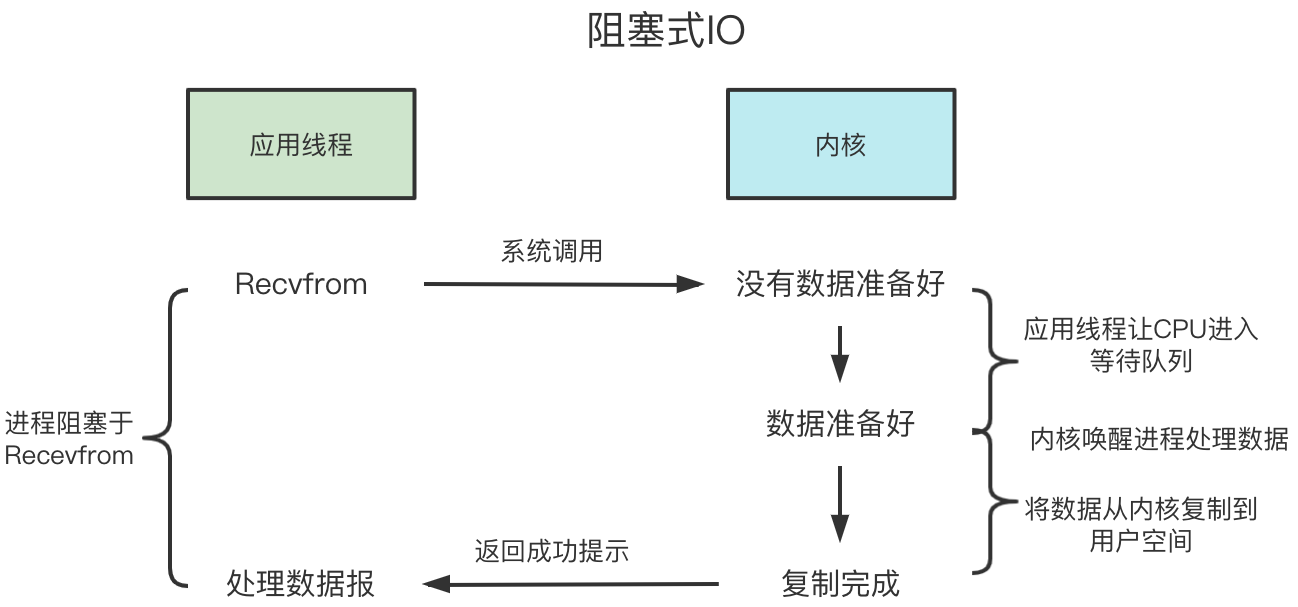
> 阻塞IO和非阻塞IO的区别：
> **第一步发起IO请求是否会被阻塞**，如果阻塞知道完成就是传统的阻塞IO，如果不阻塞那就是非阻塞IO。
> 同步IO和异步IO的区别：
> **第二步是否阻塞**，而**操作系统帮你做完IO操作再将结果返回**，那就是异步IO。

我们说的阻塞和非阻塞要分场和范围，从根本上来说阻塞是进程“被休息”，**CPU处理其他进程，而这里的非阻塞IO可以理解为将大的整片时间阻塞分成N多小的阻塞**，进程依然可以获得CPU执行时间，同时CPU也可以处理其他进程。对于Linux来说，阻塞IO还是要比非阻塞IO好，因为**CPU仍然有很大几率因socket没有数据而空转**，从整体机器性能开销上来看这样的浪费更大。所以多路复用IO中的Selector.select() 函数还是阻塞的，因此这里把多路复用IO仍然划分为阻塞IO。
阻塞、非阻塞、多路IO复用，都是同步IO，异步必定是非阻塞的，所以不存在异步阻塞和异步非阻塞的说法。真正的异步IO需要CPU的深入参与。换句话说，只有用户线程在操作IO的时候根本不用考虑IO的

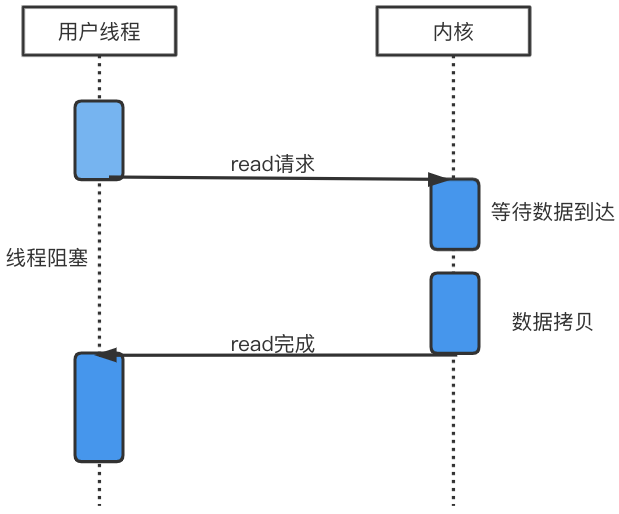
执行全部都交给CPU去完成，而自己只等待一个完成的信号的时候，才是真正的异步IO。所以**拉一个子线程去轮训或使用select、poll、epoll都不是异步。**

阻塞IO模型

阻塞式IO、BIO，一般通过在 while(true) 循环中服务端会调用`accept()`方法等待接收客户端的连接的方式监听请求，请求一旦接收到一个连接请求，就可以建立通信套接字上进行读写操作，**此时不能再接入其他客户端的操作执行完成**，不过可以通过多线程来支持多个客户端的连接。

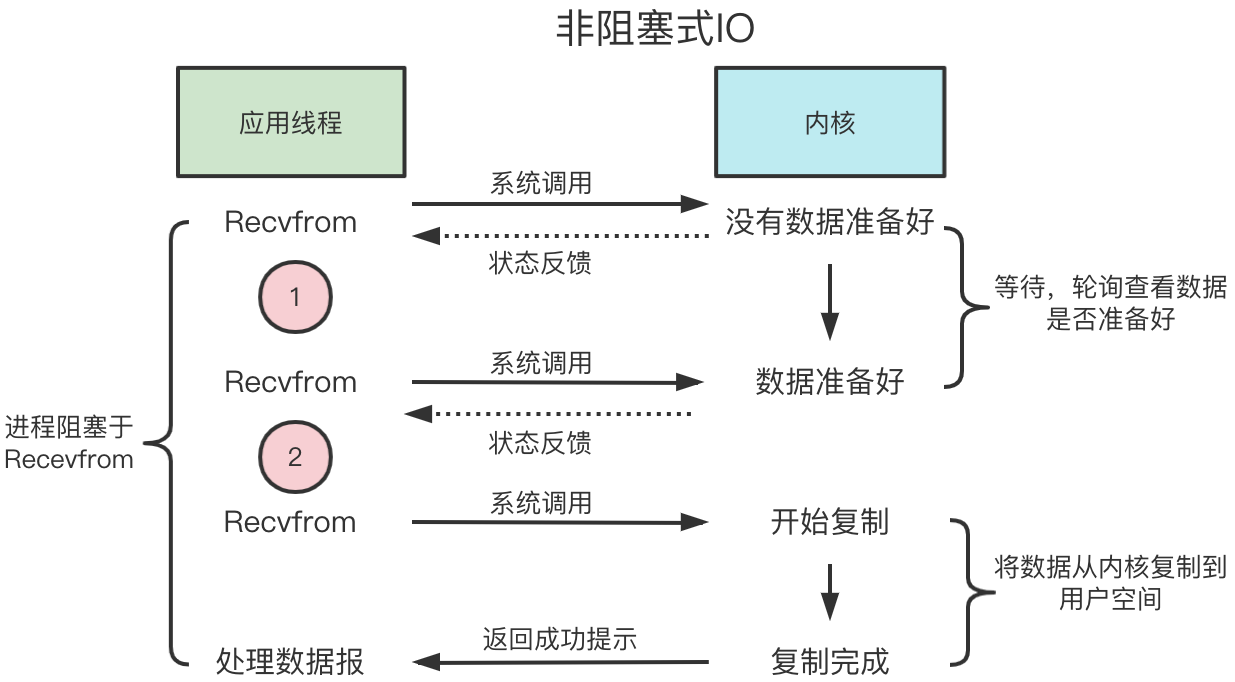


整个执行过程的时序图如下：

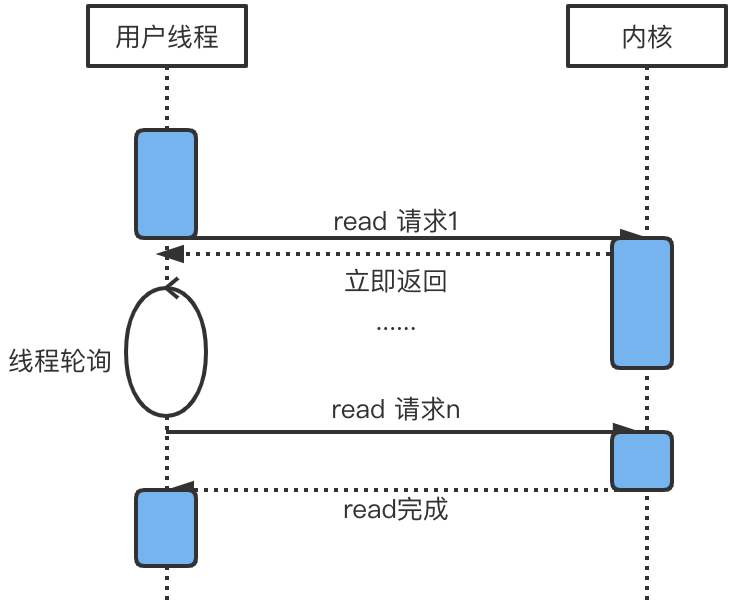


非阻塞IO模型

和阻塞IO类比，内核会立即返回，**返回后获得足够的CPU时间继续做其他的事情。用户进程第一阶段不是阻塞的，需要不断的主动循环kernel数据是否准备好;同时第二阶段依旧总是阻塞的**。



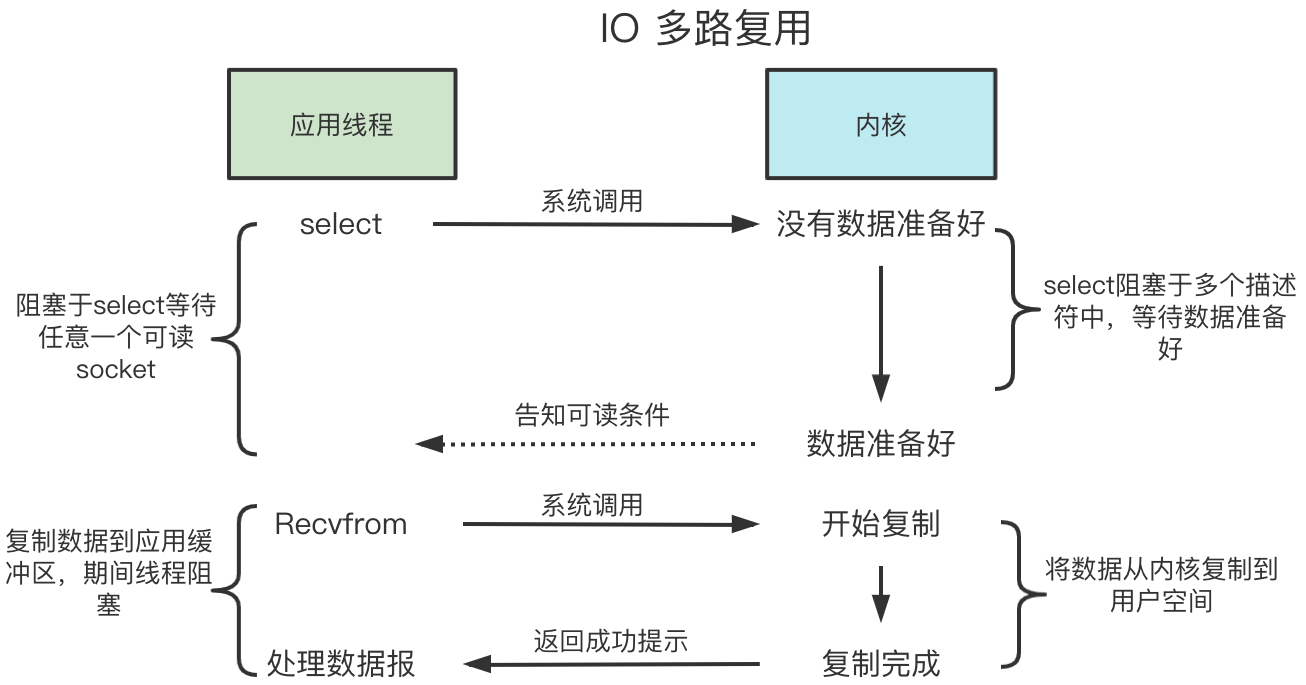
非阻塞式IO的执行过程时序图如下：



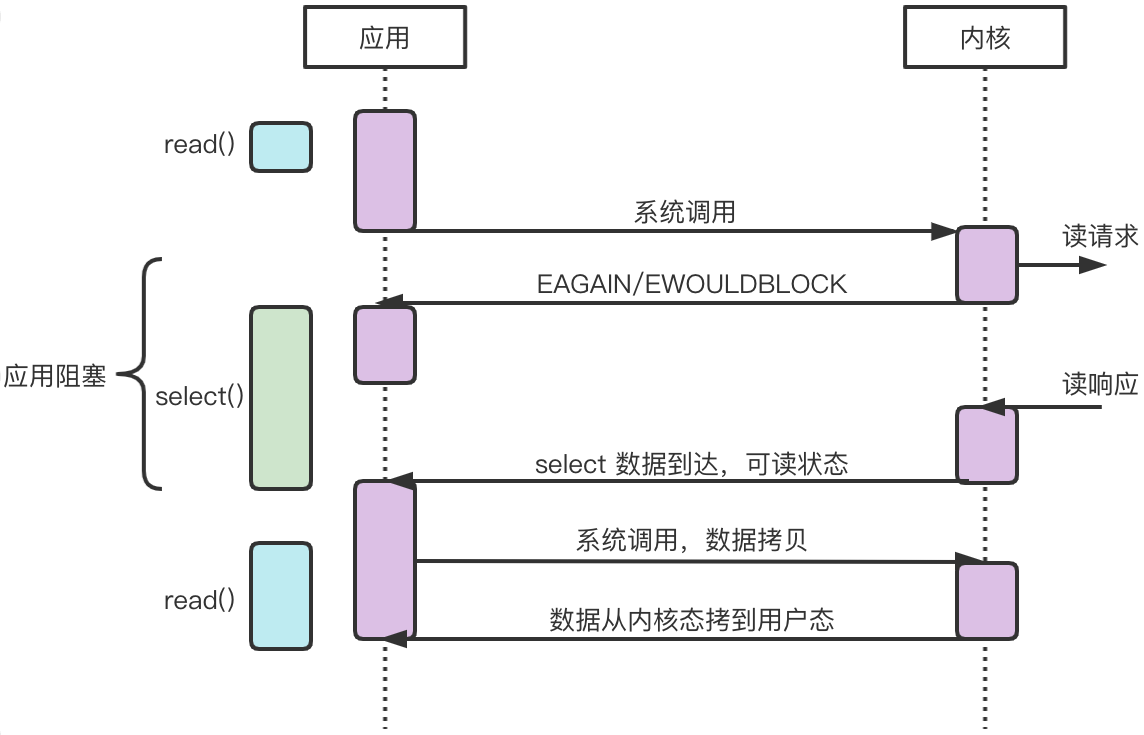
多路复用IO模型

IO多路复用（IO multiplexing），也称事件驱动IO（event-driven IO），就是在**单个线程里同时监控多个套接字**，通过`select`或`poll`轮询所负责的所有socket，有数据到达了就通知用户线程。**IO复用同非阻塞IO本质一样，不过利用了新的select系统调用**。由内核来负责本来请求进程应该做的轮询操作。

看似比本来的请求进程该做的轮询操作。看似比非阻塞IO还多了一个系统调用开销，不过**因为可以支持多路IO，才算提高了效率**。进程先是阻塞在select/poll上，再是**阻塞在读操作的第二个阶段上。**



整个执行过程的时序图如下：



> select/poll 的几大缺点：

1. 每次调用select都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时很大。（**拷贝大量fd开销大**）。
2. 每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多的时很大。（**遍历大量fd开销大**）。
3. **select支持的文件描述符(fd)数量太少**，默认1024。

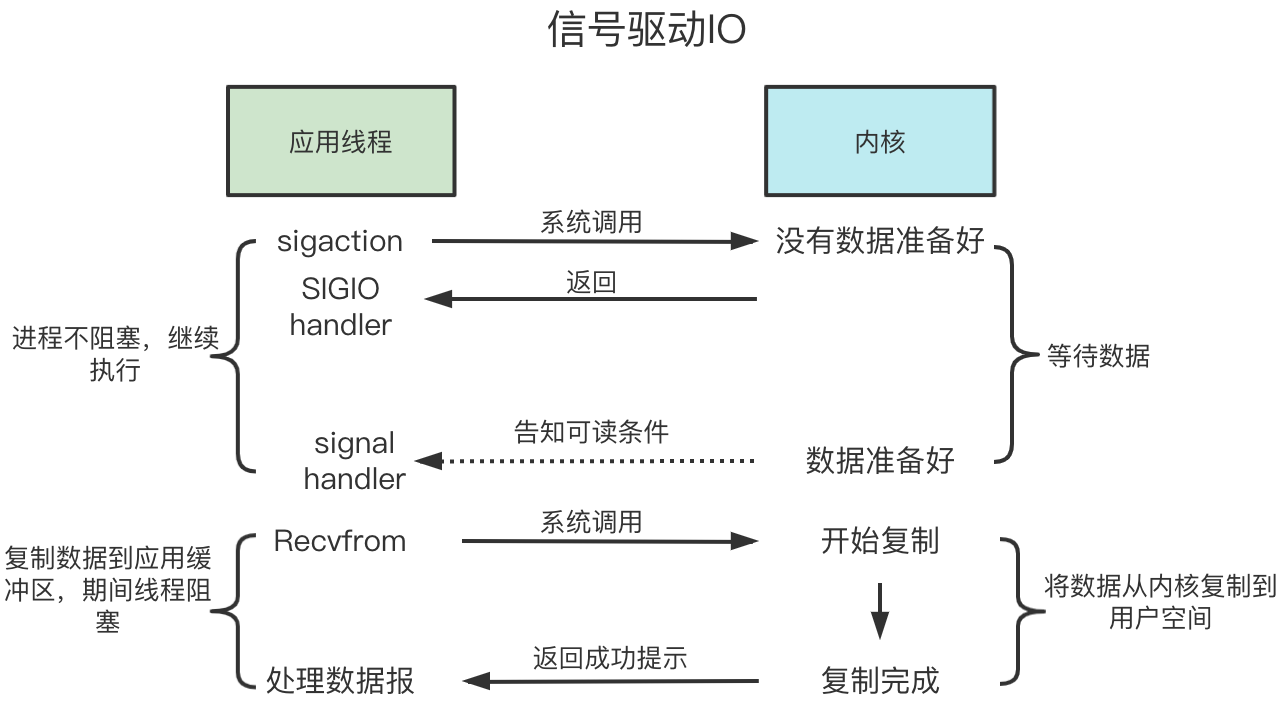
epoll（Linux2.5.44内核中引入，2.6内核正式引入，可被用于代理POSIX select 和 poll系统调用）：

1. **内核与用户空间共享一块内存**。
2. **通过回调解决遍历问题**。
3. **没有fd限制，可以支撑10w连接**。

可以说select/poll的缺点在epoll上都被优化掉了。

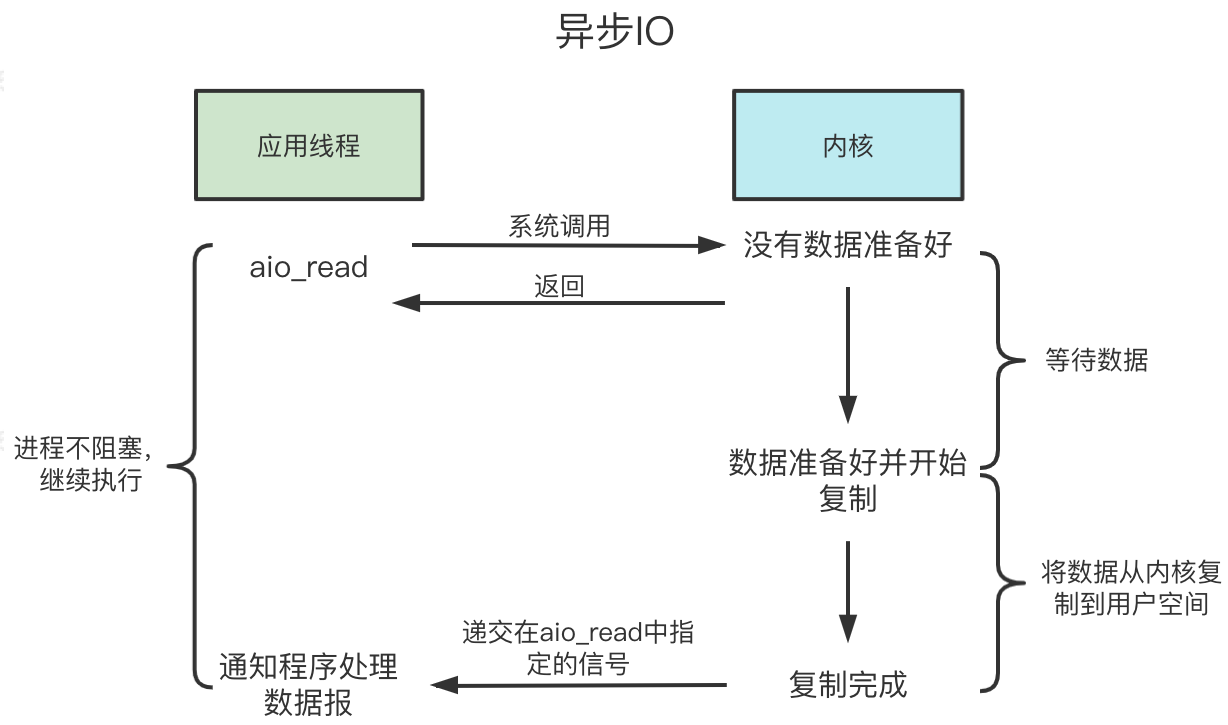
信号驱动IO

信号驱动IO是非阻塞的IO类型，我们梳理了以上的三种IO模型，除了非阻塞IO模型是非阻塞IO除外，其余的都是阻塞IO，但是非阻塞IO并不代表对CPU资源使用友好，尤其当CPU空转时系统开销可能更大，上面介绍的三种IO类型只有使用了epoll的多路复用IO模型对系统开销最为友好。信号驱动IO模型和使用epoll的多路复用IO模型比较相似，其中他们最大的不同点就是**IO执行的数据准备阶段，信号驱动IO不会阻塞用户进程**。如下图所示，当用户进程需要等待数据的时候，会向内核发送一个信号，然后用户进程继续向下执行。当内核中数据准备完成之后，内核向进程发送一个信号，然后**用户进程从内核态拷贝数据到用户态**，整个IO操作流程结束。我们可以看到在**第一阶段线程数据准备阶段应用并没有阻塞，而第二阶段数据拷贝阶段应用被阻塞**，因此信号驱动IO是**同步非阻塞**IO。



异步IO

异步IO真正实现了IO全流程的非阻塞，**用户进程发出系统调用后立即返回，内核等待数据处理完成，内核将数据从内核态拷贝到用户进程缓冲区**，然后发送信号通知用户线程IO操作执行完毕（与SIGIO相比，一个是发送信号告诉用户进程数据准备完毕，一个是IO执行完毕）。其中最为典型的是windows IOCP模型。



以上所有的网络模型的梳理与模型示意图，其中在Java中主要使用的有BIO、NIO和多路复用IO。BIO结构简单相较于除AIO以外的其他IO模型只有一次系统调用，在数据库连接等池化IO资源中还是多以BIO为主。Java使用的还是以NIO为主，其中我们后面即将梳理的Netty使用的就是采用epoll模式的多路复用NIO模型。AIO 适用于连接数多且需要长时间连接的场景，再加上AIO系统支持程度有限且底层实现复杂。NIO之前也尝试过AIO，但效果并不是很理想而最终废弃。

> 之前在群里看到朋友们发的很有意思的Java IO模型比喻总结：
>
> 例子：有一个养鸡的农场，里面养着来自各个农户（Thread）的鸡（Socket），每家农户都在农场中建立了自己的鸡舍（SocketChannel）
>
> 1. BIO：Block IO，每个农户盯着自己的鸡舍，一旦有鸡下蛋，就去做捡蛋处理；
> 2. NIO：No-BlockIO-单Selector，农户们花钱请了一个饲养员（Selector），并告诉饲养员（register）如果哪家的鸡有任何情况（下蛋）均要向这家农户报告（selectkeys）；
> 3. NIO：No-BlockIO-多Selector，当农场中的鸡舍(Selector)逐渐增多时，一个饲养员巡视（轮询）一次所需时间就会不断地加长，这样农户知道自己家的鸡有下蛋的情况就会发生较大的延

> 迟。怎么解决呢？没错，多请几个饲养员（多Selector），每个饲养员分配管理鸡舍，这样就可以减轻一个饲养员的工作量，同时农户们可以更快的知晓自己家的鸡是否下蛋了；
> 4. Epoll模式：如果采用Epoll方式，农场问题应该如何改进呢？其实就是饲养员不需要再巡视鸡舍，而是听到哪间鸡舍(Selector)的鸡打鸣了（活跃连接），就知道哪家农户的鸡下蛋了；
> 5. AIO：Asynchronousl/O,鸡下蛋后，以前的NIO方式要求饲养员通知农户去取蛋，AIO模式出现以后，事情变得更加简单了，取蛋工作由饲养员自己负责，然后取完后，直接通知农户来拿即可，而不需要农户自己到鸡舍去取蛋。

总结

这一小节我们从socket编程开始，介绍了什么socket，socket是我们网络编程的一套接口，位于传输层和应用层之间。socket就像他的中文意思“插座”一样，通过编写适配这个“插座”我们的程序就可以“插入网线”实现网通信功能。随后我们写了几个简单的socket程序Demo，并分析了背后的调用过程，其中socket分两种类型，一种是客户端用的socket，服务端用的socketServer。SocketServer通过bind()方法绑定在某个端口上，并且调用accept()方法等待连接。客户端Socket构造时需要传入一个IP地址和端口号，再调用connect方法连接服务端SocketServer，随后开始数据数据传输。这里要注意服务端负责连接的socket和数据传输的socket是两个socket。通过分析源码我们发现socket基本都是基于TCP的，TCP的socket本质就是一个文件流，所以一个服务有多少个连接和fd（文件描述符）的限制有很大的关系。接下来我们梳理了IO模型，我们先是梳理了阻塞与非阻塞，同步与异步的概念。随后我们引入了五种IO模型，它们分别是阻塞IO（BIO）、非阻塞IO（NIO）、多路复用IO、信号驱动IO和异步IO。其中除了异步IO其他都是同步IO。阻塞IO和多路复用IO是阻塞IO，其他都是非阻塞IO，这里要注意的是多路复用IO是阻塞在selector上的，对于应用进程来说是非阻塞的。最后我们详细梳理了这五种IO模型以及部分的时序图，深入对这五个网络模型的理解。以上就是这一小节的全部内容了，接下来我们将开始梳理Java网络编程的“执牛耳框架”Netty。加油加油，冲冲冲。😉

学习资料

- 为什么有监听socket和连接socket,为什么产生两个socket
- 趣谈网络协议
- java 进阶训练营第四课