IO模型

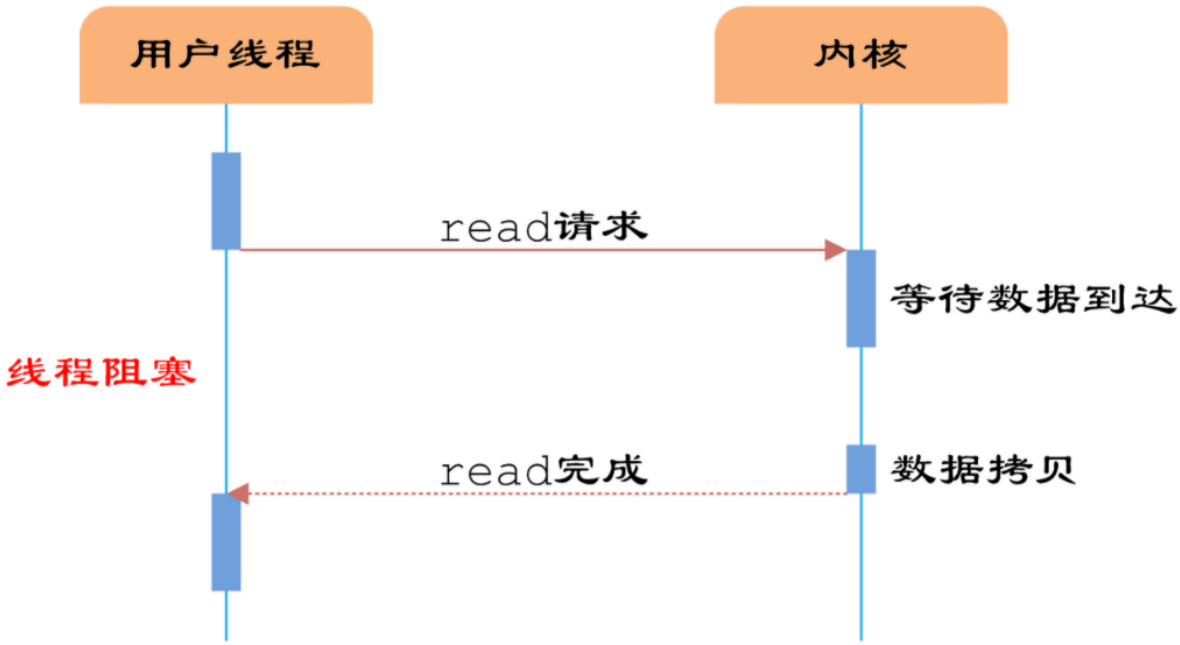
常见的IO模型可以从两个角度画风，同步或异步，阻塞或非阻塞

- 同步：发起一个调用，得到结果才返回。
- 异步：调用发起后，调用直接返回；调用方主动询问被调用方获取结果，或被调用方通过回调函数。
- 阻塞：调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。
- 非阻塞：调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

同步才有阻塞和非阻塞之分；
阻塞与非阻塞关乎如何对待事情产生的结果（阻塞：不等到想要的结果我就不走了）

主要模型

1、同步阻塞IO(Blocking IO)



最传统的一种 IO 模型，即在读写数据过程中会发生阻塞现象。当用户线程发出 IO 请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才能解除block状态，

优点：1、能够及时返回数据，无延迟 2、对内核开发者来说省事

缺点：对用户来说处于等待就要付出性能的代价

用户线程使用同步阻塞IO模型的伪代码描述为：

```
{
  read(socket, buffer);
  process(buffer);
}
```

即用户需要等待read将socket中的数据读取到buffer后，才继续处理接收的数据。整个IO请求的过程中，用户线程是被阻塞的，这导致用户在发起IO请求时，不能做任何事情，对CPU的资源利用率不够。

测试：

bio

启动服务端

```
BioServer [Java Application] /Library/Java/JavaVirtualMac
服务器已经启动，监听端口是：23456
```

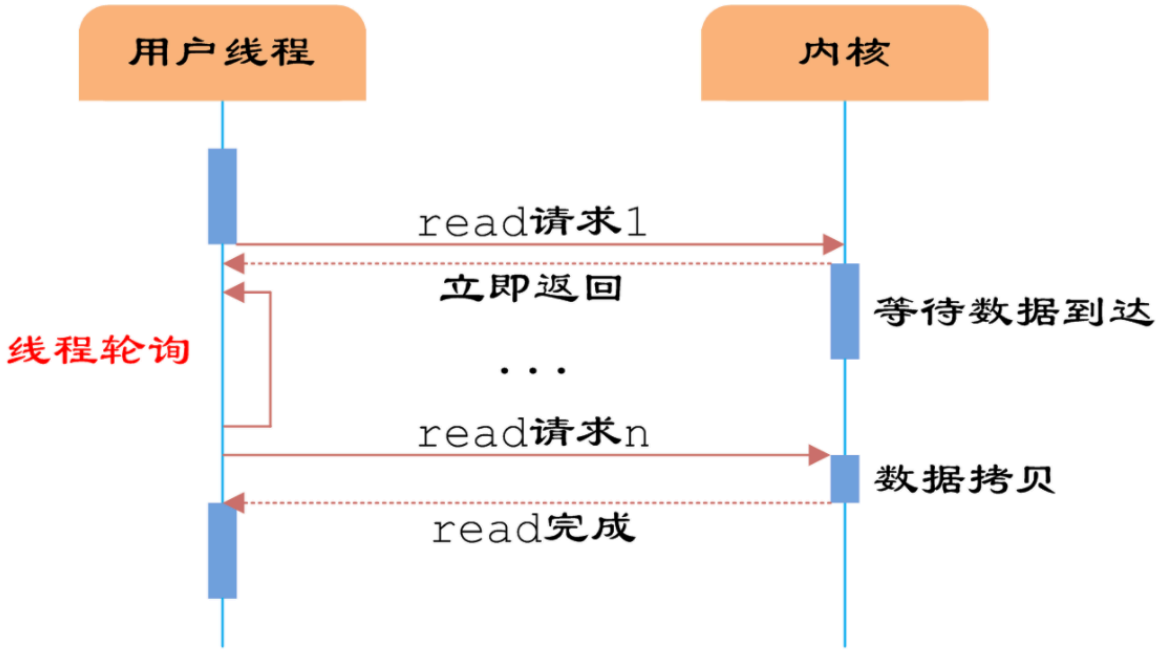
启动客户端

```
➜  Desktop java BioClient
客户端发送数据:d8136e8c-fa28-4d03-89a4-df0a04a7bed9
```

服务端收到数据并打印出客户端端口

```
BioServer [Java Application] /Library/Java/JavaVirtualMachines/jdk1
服务器已经启动，监听端口是：23456
53929
收到：d8136e8c-fa28-4d03-89a4-df0a04a7bed9
```

2、同步非阻塞IO



当用户线程发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。如果结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦内核中的数据准备好了，并且又再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。 所以事实上，在非阻塞 IO 模型中，用户线程需要不断地询问内核数据是否就绪，也就说非阻塞 IO 不会交出 CPU，而会一直占用 CPU。

优点：能够在等待任务完成的时间里干其他活了（包括提交其他任务，也就是 “后台” 可以有多个任务在同时执行）。

缺点：任务完成的响应延迟增大了，因为每过一段时间才去轮询一次read操作，而任务可能在两次轮询之间的任意时间完成。这会导致整体数据吞吐量的降低。

用户线程使用非阻塞IO模型的伪代码描述为：

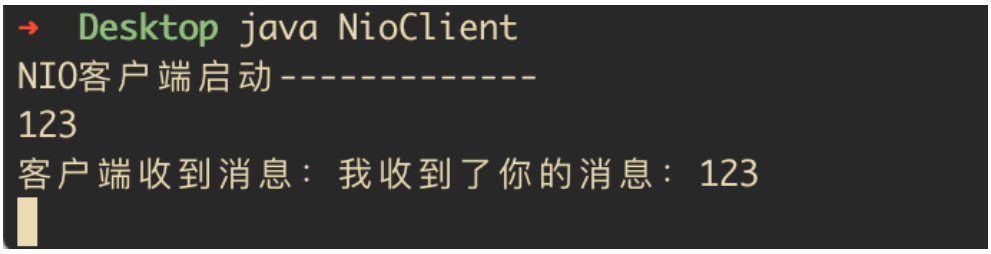
```
{
while(read(socket, buffer) != SUCCESS);
process(buffer);
}
```

即用户需要不断地调用read，尝试读取socket中的数据，直到读取成功后，才继续处理接收的数据。整个IO请求的过程中，虽然用户线程每次发起IO请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，消耗了大量的CPU的资源。一般很少直接使用这种模型，而是在其他IO模型中使用非阻塞IO这一特性。

测试：

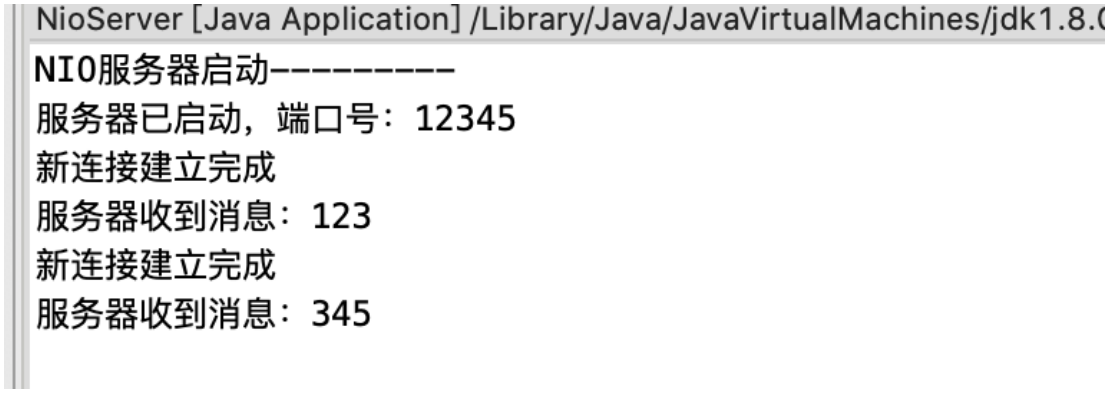
nio

分别启动服务端和客户端，并在客户端传输：



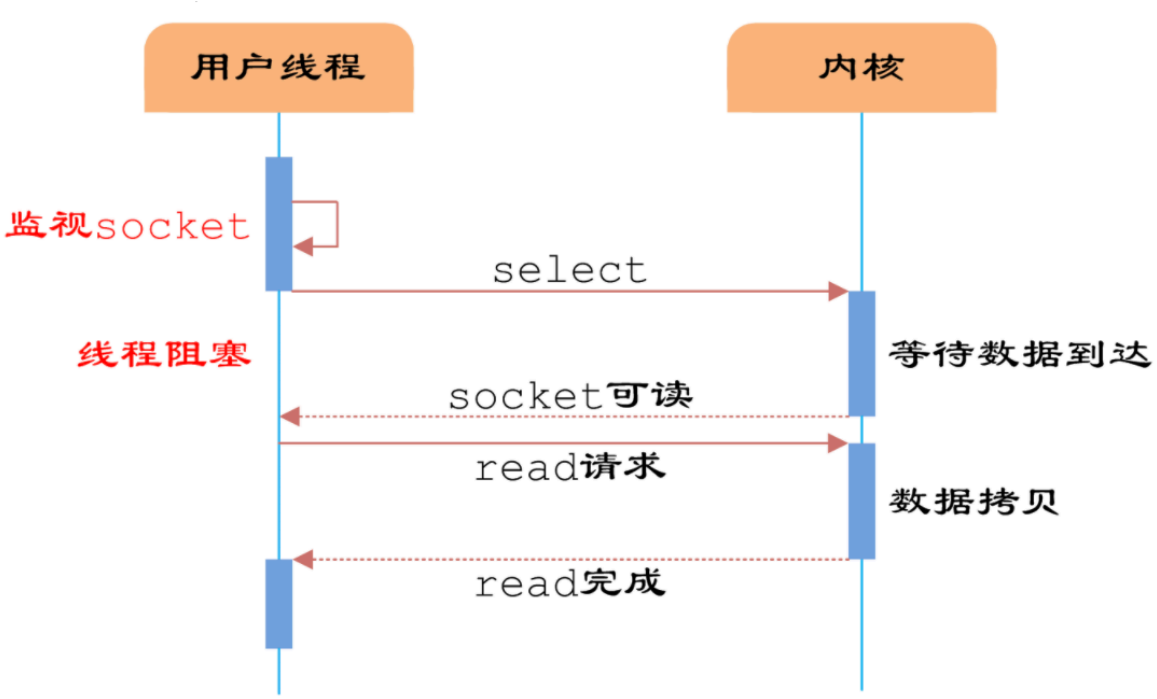
启动另一个客户端，并传输：

服务端：



可以看到，前面的连接不会阻塞到后面的连接，对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用NIO的非阻塞模式来开发。

3、IO多路复用



如图所示，用户首先将需要进行IO操作的socket添加到select中，然后阻塞等待select系统调用返回。当数据到达时，socket被激活，select函数返回。用户线程正式发起read请求，读取数据并继续执行。

多路复用 IO 模型是目前使用得比较多的模型。**Java NIO 实际上就是多路复用 IO**。在多路复用 IO 模型中，会有一个线程不断去轮询多个 socket 的状态，只有当 socket 真正有读写事件时，才真正调用实际的 IO 读写操作。因为在多路复用 IO 模型中，只需要使用一个线程就可以管理多个 socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有 socket 读写事件进行时，才会使用 IO 资源，所以它大大减少了资源占用。在 Java NIO 中，是通过selector.select()去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。多路复用 IO 模式，通过一个线程就可以管理多个 socket，只有当 socket 真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用 IO 比较适合连接数比较多的情况。

另外**多路复用 IO 为何比非阻塞 IO 模型的效率高**是因为在非阻塞 IO 中，不断地询问 socket 状态时通过用户线程去进行的，而在多路复用 IO 中，轮询每个 socket 状态是内核在进行的，这个效率要比用户线程要高的多。

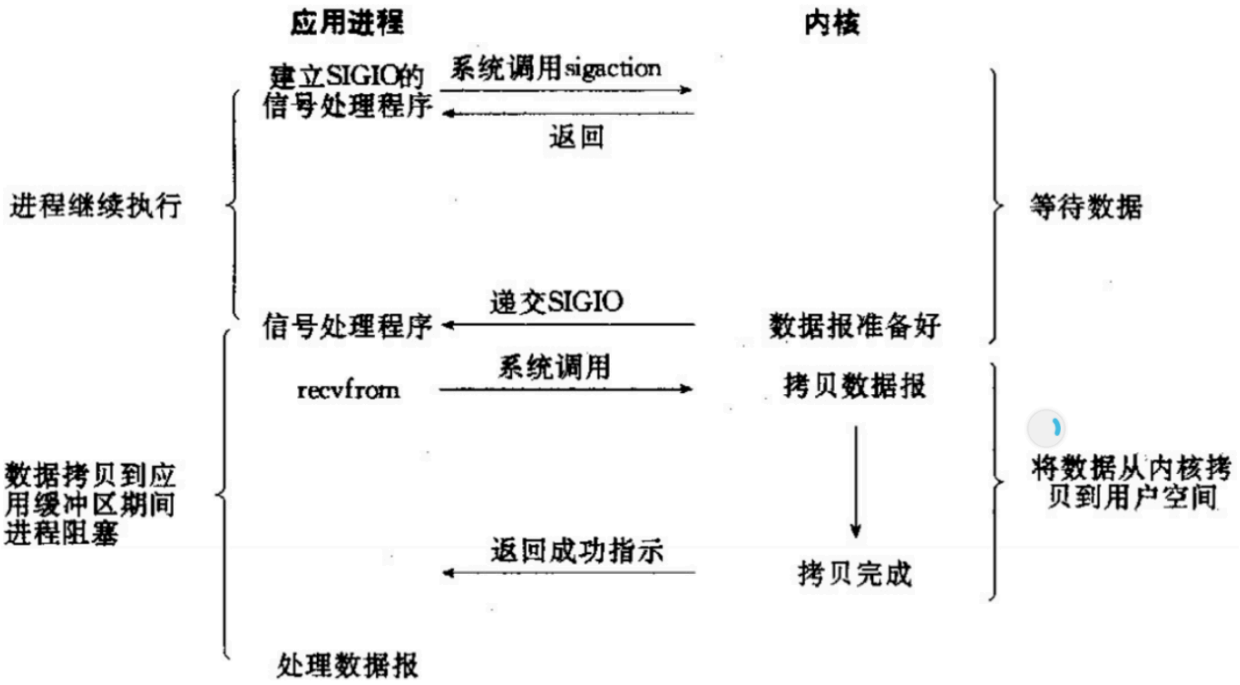
优点：专一进程解决多个进程IO的阻塞问题，性能好；Reactor模式;

缺点：实现、开发应用难度较大；

用户线程使用select函数的伪代码描述为：

```
{
  select(socket);
  while(1) {
    sockets = select();
    for(socket in sockets) {
      if(can_read(socket)) {
        read(socket, buffer);
        process(buffer);
      }
    }
  }
}
```

4、信号驱动式IO

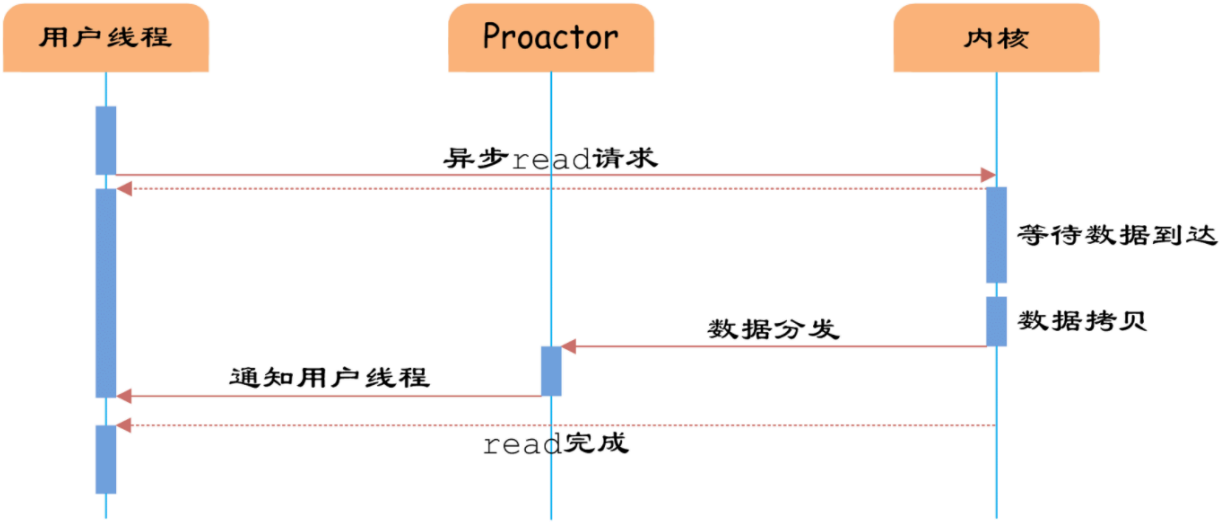


在信号驱动 IO 模型中，当用户线程发起一个 IO 请求操作，会给对应的 socket 注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用 IO 读写操作来进行实际的 IO 请求操作。

优点：回调机制

缺点：实现、开发应用难度大

5、异步 IO 模型



异步 IO 模型才是最理想的 IO 模型，在异步 IO 模型中，当用户线程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个 asynchronous read 之后， 它会立刻返回，说明 read 请求已经成功发起了，因此不会对用户线程产生任何 block。然后，内核会等待数据准备完成，然后将数据拷贝到用户线程，当这一切都完成之后，内核会给用户线程发送一个信号，告诉它 read 操作完成了。也就说用户线程完全不需要实际的整个 IO 操作是如何进行的，只需要先发起一个请求，当接收内核返回的成功信号时表示 IO 操作已经完成，可以直接去使用数据了。

也就说在异步 IO 模型中，IO 操作的两个阶段都不会阻塞用户线程，这两个阶段都是由内核自动完 成，然后发送一个信号告知用户线程操作已完成。用户线程中不需要再次调用 IO 函数进行具体的读写。这点是和信号驱动模型有所不同的，在信号驱动模型中，当用户线程接收到信号表示数据 已经就绪，然后需要用户线程调用 IO 函数进行实际的读写操作;而在异步 IO 模型中，收到信号表示 IO 操作已经完成，不需要再在用户线程中调用 IO 函数进行实际的读写操作。

注意，**异步 IO 是需要操作系统的底层支持**，在 Java 7 中，提供了 Asynchronous IO。

优点：不阻塞，数据一步到位；Proactor模式；非常适合高性能高并发应用

缺点：需要操作系统的底层支持，实现、开发应用难度大；

用户线程使用异步IO模型的伪码描述：

```
void UserCompletionHandler::handle_event(buffer) {
  process(buffer);
}
{
  aio_read(socket, new UserCompletionHandler);
}
```