1. 同步\异步and阻塞\非阻塞

1. 同步

```
#所谓同步，就是在发出一个功能调用时，在没有得到结果之前，该调用就不会返回。
按照这个定义，其实绝大多数函数都是同步调用。但是一般而言，我们在说同步、异步的时候，
特指那些需要其他部件协作或者需要一定时间完成的任务。

举例：
1. multiprocessing.Pool下的apply #发起同步调用后，就在原地等着任务结束，
根本不考虑任务是在计算还是在io阻塞，总之就是一股脑地等任务结束
2. concurrent.futures.ProcessPoolExecutor().submit(func,).result()
3. concurrent.futures.ThreadPoolExecutor().submit(func,).result()
```

2. 异步

```
异步的概念和同步相对。当一个异步功能调用发出后，调用者不能立刻得到结果。当该异步功能完成后，
通过状态、通知或回调来通知调用者。如果异步功能用状态来通知，那么调用者就需要每隔一定时间检查一次，
效率就很低（有些初学多线程编程的人，总喜欢用一个循环去检查某个变量的值，这其实是一 种很严重的错误）。
如果是使用通知的方式，效率则很高，因为异步功能几乎不需要做额外的操作。至于回调函数，其实和通知没太多区别。

举例：
1. multiprocessing.Pool().apply_async() #发起异步调用后，并不会等待任务结束才返回，相反，
会立即获取一个临时结果（并不是最终的结果，可能是封装好的一个对象）。
2. concurrent.futures.ProcessPoolExecutor(3).submit(func,)
3. concurrent.futures.ThreadPoolExecutor(3).submit(func,)
```

3. 阻塞

```
阻塞调用是指调用结果返回之前，当前线程会被挂起（如遇到io操作）。
函数只有在得到结果之后才会将阻塞的线程激活。有人也许会把阻塞调用和同步调用等同起来，
实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。

举例：
1. 同步调用：apply一个累计1亿次的任务，该调用会一直等待，直到任务返回结果为止，但并未阻塞住
（即便是被抢走cpu的执行权限，那也是处于就绪态）；
2. 阻塞调用：当socket工作在阻塞模式的时候，如果没有数据的情况下调用recv函数，则当前线程就会被挂起，
直到有数据为止。
```

4. 非阻塞

```
非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前也会立刻返回，同时该函数不会阻塞当前线程。
```

5. 小结

```
1. 同步与异步针对的是函数/任务的调用方式：同步就是当一个进程发起一个函数（任务）调用的时候，一直等到函数（任务）完成，而进程继续处于激活状态。而异步情况下是当一个进程发起一个函数（任务）调用的时候，不会等函数返回，而是继续往下执行当，函数返回的时候通过状态、通知、事件等方式通知进程任务完成。
2. 阻塞与非阻塞针对的是进程或线程：阻塞是当请求不能满足的时候就将进程挂起，而非阻塞则不会阻塞当前进程
```

2. IO模型介绍

1. 四种IO Model
    - blocking IO(阻塞IO)
    - nonblocking IO(非阻塞IO)
    - IO multiplexing(IO多路复用)
    - asynchronous IO(异步IO)
2. IO发生时涉及的对象和步骤
    1. 对象

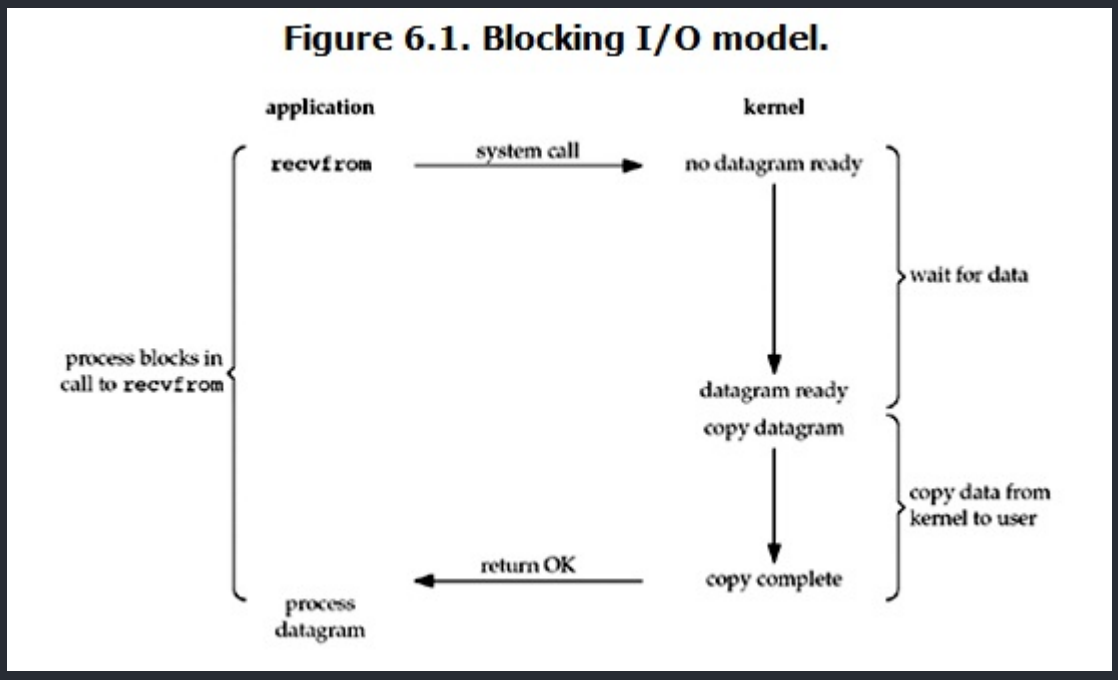
```
1. IO的process (or thread)
2. 系统内核(kernel)
```

    2. 步骤

```
1. 等待数据准备 (Waiting for the data to be ready)
2. 将数据从内核拷贝到进程中(Copying the data from the kernel to the process)
```

3. 阻塞IO(blocking IO)

1. 简析



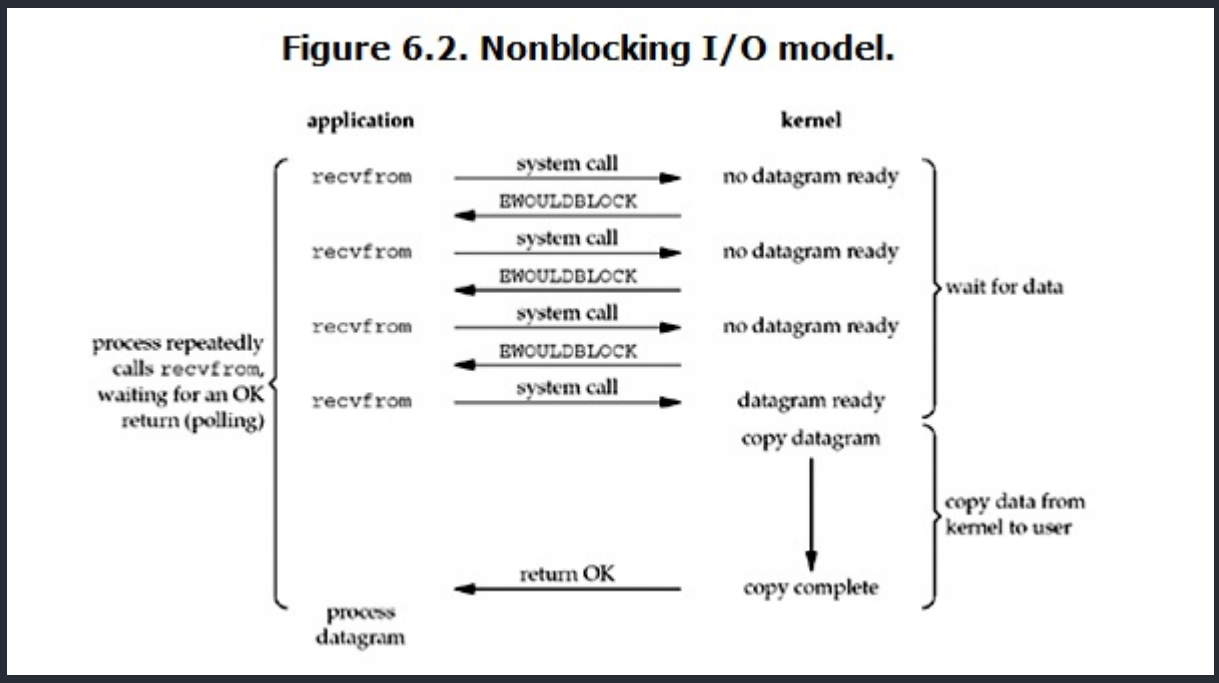

```
当用户进程调用了recvfrom这个系统调用，kernel就开始了IO的第一个阶段：准备数据。
```

对于network io来说，很多时候数据在一开始还没有到达（比如，还没有收到一个完整的UDP包），
这个时候kernel就要等待足够的数据到来。
而在用户进程这边，整个进程会被阻塞。当kernel一直等到数据准备好了，
它就会将数据从kernel中拷贝到用户内存，然后kernel返回结果，用户进程才解除block的状态，重新运行起来。

2. 特点：
   blocking IO的特点就是在IO执行的两个阶段（等待数据和拷贝数据两个阶段）都被block了

4. 非阻塞IO(non-blocking IO)

1. 解析:



非阻塞的recvform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recvform系统调用。
重复上面的过程，循环往复的进行recvform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。

2. 特点

   1. 优点：

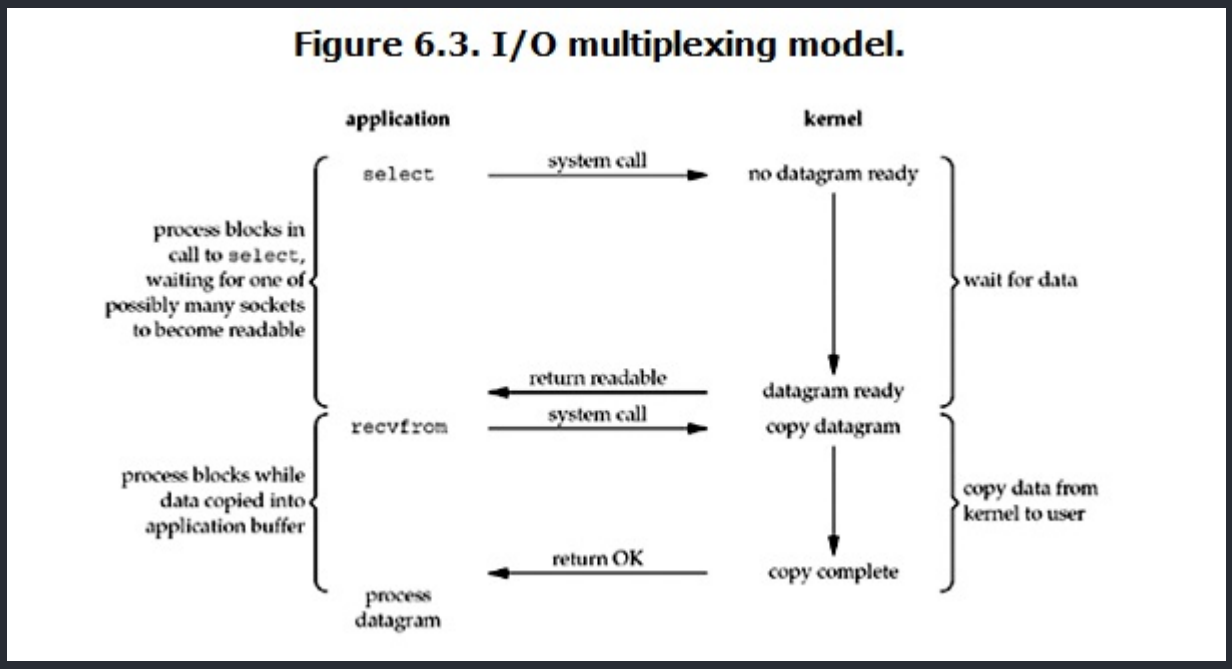
      能够在等待任务完成的时间里干其他活了（包括提交其他任务，也就是“后台”可以有多个任务在“”同时“”执行）。

   2. 缺点：

      1. 循环调用recv()将大幅度推高CPU占用率；这也是我们在代码中留一句time.sleep(2)的原因，
      否则在低配主机下极容易出现卡机情况
      2. 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。

5. 多路复用IO(IO multiplexing)

1. 解析




```
当用户进程调用了select，那么整个进程会被block，而同时，kernel会“监视”所有select负责的socket，
当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，
将数据从kernel拷贝到用户进程。
这个图和blocking IO的图其实并没有太大的不同，事实上还更差一些。
因为这里需要使用两个系统调用(select和recvfrom)，而blocking IO只调用了一个系统调用(recvfrom)。
但是，用select的优势在于它可以同时处理多个connection。
```

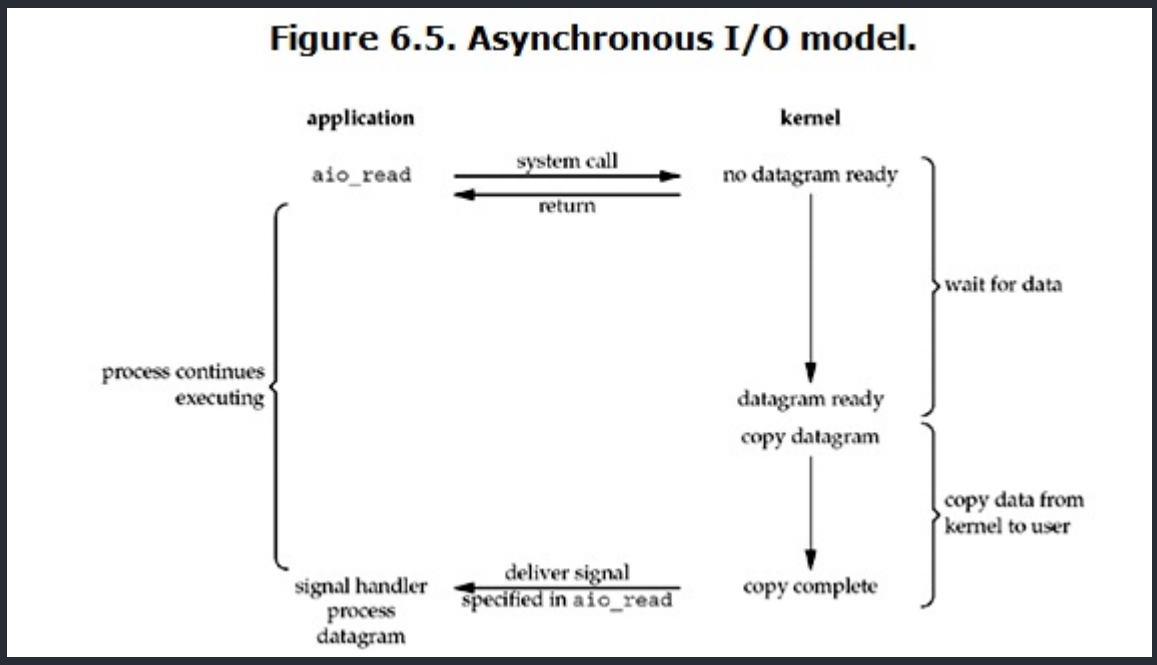
2. 特点

```
1. 如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用
multi-threading + blocking IO的web server性能更好，可能延迟还更大。
select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。
2. 在多路复用模型中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，
整个用户的process其实是一直被block的。只不过process是被select这个函数block，
而不是被socket IO给block。
```

3. 结论:
   select的优势在于可以处理多个连接，不适用于单个连接

6. 异步IO(Asynchronous I/O)

1. 解析




```
用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，
当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。
然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，
kernel会给用户进程发送一个signal，告诉它read操作完成了。
```

7. IO模型比较分析

1. 解析

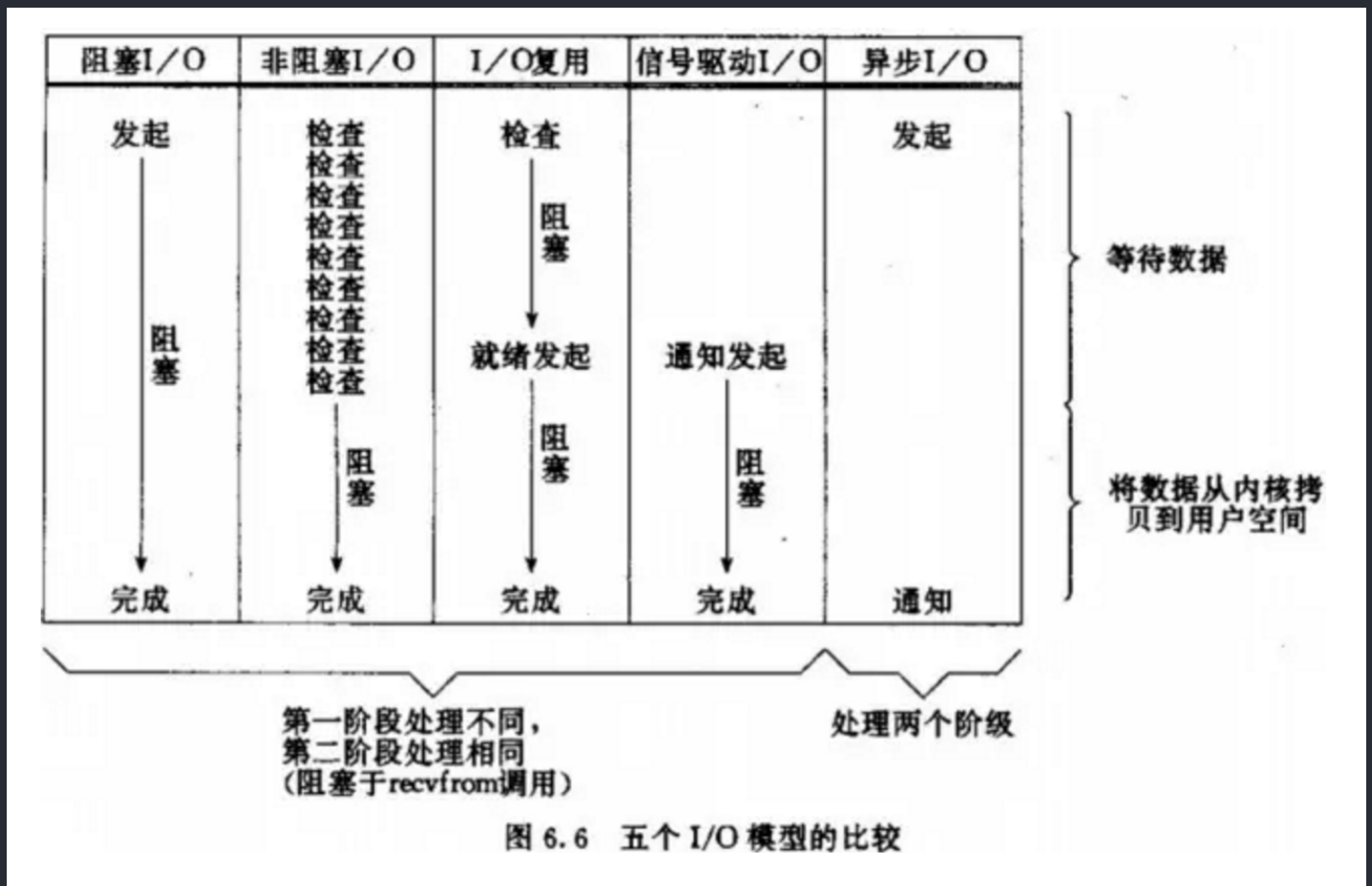


图 6.6 五个 I/O 模型的比较

```
会发现non-blocking IO和asynchronous IO的区别还是很明显的。在non-blocking IO中，
虽然进程大部分时间都不会被block，但是它仍然要求进程去主动的check，并且当数据准备完成以后，
```

```
也需要进程主动的再次调用recvfrom来将数据拷贝到用户内存。而asynchronous IO则完全不同。
它就像是用户进程将整个IO操作交给了他人（kernel）完成，然后他人做完后发信号通知。
在此期间，用户进程不需要去检查IO操作的状态，也不需要主动的去拷贝数据。
```

8. selectors模块

1. 用法

```
win:    select
linux:  select poll     epoll

select的缺点：
1. 每次调用select都要将所有的fd(文件描述符)拷贝到内核空间导致效率下降
2. 遍历所有的fd，是否有数据访问；(最重要的问题)
3. 最大链接数(1014)

poll：
        最大链接数没有限制

epoll:
1. 第一个函数：创建epoll句柄，将所有的fd(文件描述符)拷贝到内核空间，但是只需拷贝一次
2. 回调函数：某一个函数或者某一个动作成功完成后会触发的函数
             为所有的fd绑定一个回调函数，一旦有数据访问，
             触发该回调函数，回调函数将fd放到链表中；

selectors.DefaultSelector       selectors模块会根据系统平台自动选择一个IO多路复用的机制

sel.select()    #[(key,mask),(key,mask)]

key.date                accept函数
key.fileobj             sock
当前活动的socket对象以及对应绑定的函数
```

2. 示例代码

```python
import selectors
import socket

sel = selectors.DefaultSelector()

def accept(sock, mask):
    conn, addr = sock.accept()  # Should be ready
    print('accepted', conn, 'from', addr)
    conn.setblocking(False)
    sel.register(conn, selectors.EVENT_READ, read)

def read(conn, mask):
    data = conn.recv(1000)  # Should be ready
    if data:
        print('echoing', repr(data), 'to', conn)
        conn.send(data)  # Hope it won't block
```

```
    else:
        print('closing', conn)
        sel.unregister(conn)
        conn.close()

sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```