# 什么是同步？什么是异步？

重点在于多个任务和事件发生时，一个事件的发生或执行是否会导致整个流程的暂时等待。

# 什么是阻塞？什么是非阻塞？

阻塞和非阻塞的区别关键在于当发出请求一个操作时，如果条件不满足，是会一直等待还是返回一个标志信息。

# 什么是阻塞IO？什么是非阻塞IO？

通常来说，IO操作包括：对硬盘的读写、对socket的读写以及外设的读写。

当用户线程发起一个IO请求（以读请求操作为例），内核会去查看要读取的数据是否就绪，对于阻塞IO来说，如果数据没有就绪，则会一直在那等待，直到数据就绪；对于非阻塞IO来说，如果数据没有就绪，则会返回一个标志信息告知用户线程当前要读的数据没有就绪。当数据就绪之后，便将数据拷贝到用户线程，这样才完成了一个完整的IO读请求操作，也就是说一个完整的IO读请求操作包括两个阶段：

1. 查看数据是否就绪；
2. 进行数据拷贝（内核将数据拷贝到用户线程）。

阻塞IO和非阻塞IO的区别就在于第一个阶段，如果数据没有就绪，在查看数据是否就绪的过程中是一直等待，还是直接返回一个标志信息。

Java中传统的IO都是阻塞IO，比如通过socket来读数据，调用read()方法之后，如果数据没有就绪，当前线程就会一直阻塞在read方法调用那里，直到有数据才返回；而如果是非阻塞IO的话，当数据没有就绪，read()方法应该返回一个标志信息，告知当前线程数据没有就绪，而不是一直在那里等待。

# 什么是同步IO？什么是异步IO？

同步IO即如果一个线程请求进行IO操作，在IO操作完成之前，该线程会被阻塞；而异步IO为，如果一个线程请求进行IO操作，IO操作不会导致请求线程被阻塞。

事实上，同步IO和异步IO模型是针对用户线程和内核的交互来说的：

* 对于同步IO，当用户发出IO操作请求之后，如果数据没有就绪，需要通过用户线程或者内核不断地去轮询数据是否就绪，当数据就绪时，再将数据从内核拷贝到用户线程；
* 而异步IO，只有IO请求操作的发出是由用户线程来进行的，IO操作的两个阶段都是由内核自动完成，然后发送通知告知用户线程IO操作已经完成。也就是说在异步IO中，不会对用户线程产生任何阻塞。

同步IO和异步IO的关键区别反映在数据拷贝阶段是由用户线程完成还是内核完成。所以说异步IO必须要有操作系统的底层支持。

# 同步与异步、阻塞与非阻塞对于IO形式的区别

同步IO和异步IO与阻塞IO和非阻塞IO是不同的两组概念。

阻塞IO和非阻塞IO是反映在当用户请求IO操作时，如果数据没有就绪，是用户线程一直等待数据就绪，还是会收到一个标志信息这一点上面的。也就是说，阻塞IO和非阻塞IO是反映在IO操作的第一个阶段，在查看数据是否就绪时是如何处理的。

# 五种IO模型

## 阻塞IO模型

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，用户线程才解除block状态。

典型的阻塞IO模型的例子为：

data = socket.read();

如果数据没有就绪，就会一直阻塞在read方法。

## 非阻塞IO模型

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。

典型的非阻塞IO模型一般如下：

while( true ){

data = socket.read();

if(data != error){

处理数据

break;

}

}

但是对于非阻塞IO就有一个非常严重的问题，在while循环中需要不断地去询问内核数据是否就绪，这样会导致CPU占用率非常高，因此一般情况下很少使用while循环这种方式来读取数据。

## 多路复用IO模型

多路复用IO模型是目前使用的比较多的模型。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。

也许我们可以采用“多线程+阻塞IO”达到类似的效果，但是由于在“多线程+阻塞IO”中，每个socket对应一个线程，这样会造成很大的资源占用，并且尤其是对于长连接来说，线程的资源一直不会释放，如果后面陆续有很多连接的话，就会造成性能上的瓶颈。

而多路复用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状态是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高的多。

不过要注意的是，多路复用IO模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用IO模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

## 信号驱动IO模型

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。

## 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，当它收到一个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。也就是说用户线程完全不需要知道实际的整个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。

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的，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。这一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的。在Java 7中，提供了Asynchronous IO。

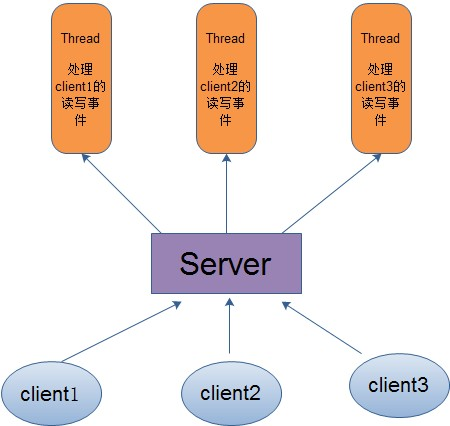
## 总结

前四种IO模型实际上都属于同步IO，只有最后一种是真正的异步IO，因为无论是多路复用IO还是信号驱动模型，IO操作的第2个阶段都会引起用户线程阻塞，也就是内核进行数据拷贝的过程都会让用户线程阻塞。

# 两种高性能IO设计模式

在传统的网络服务设计模式中，有两种比较经典的模式：一种是多线程，一种是线程池。

对于多线程模式，接收到client请求后，服务器就会新建一个线程来处理该client的读写事件。



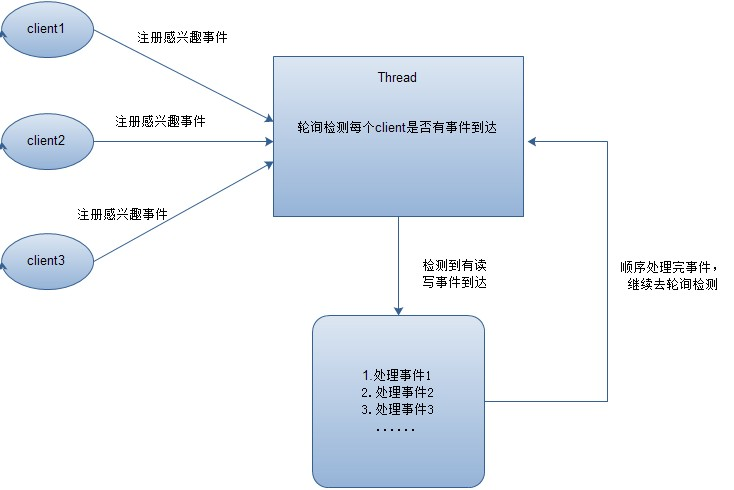
由于服务器为每个client的连接都采用一个线程去处理，使得资源占用非常大。因此，当连接数量达到上限时，再有用户请求链接，直接会导致资源瓶颈，严重的可能会直接导致服务器崩溃。

因此，为了解决这种一个线程对应一个客户端模式带来的问题，提出了采用线程池的方式。就是创建一个固定大小的线程池，收到一个客户端请求，就从线程池取一个空闲线程来处理，当客户端处理完读写操作之后，就交出对线程的占用。这样就避免为每一个客户端都要创建线程带来的资源浪费，使得线程可以重用。

但是，线程池也有它的弊端，如果连接大多是长连接，可能会导致在一段时间内，线程池中的线程都被占用，那么当再有用户请求连接时，由于没有可用的空闲线程来处理，就会导致客户端连接失败，从而影响用户体验。因此，线程池比较适合大量的短连接应用。

因此便出现了下面的两种高性能IO设计模式：Reactor和Proactor。

在Reactor模式中，会先对每个client注册感兴趣的事件，然后有一个线程专门去轮询每个client是否有事件发生，当有事件发生时，便顺序处理每个事件，当所有事件处理完之后，便再转去继续轮询，如下图所示：



从这里可以看出，前面的五种IO模型中的多路复用IO就是采用Reactor模式。注意，处理事件的方式除了可以顺序处理外，也可以通过多线程或者线程池的方式来处理事件。

在Proactor模式中，当检测到有事件发生时，会新起一个异步操作，然后交由内核线程去处理，当内核线程完成IO操作之后，发送一个通知告知操作已完成，可以得知，异步IO模型采用的就是Proactor模式。