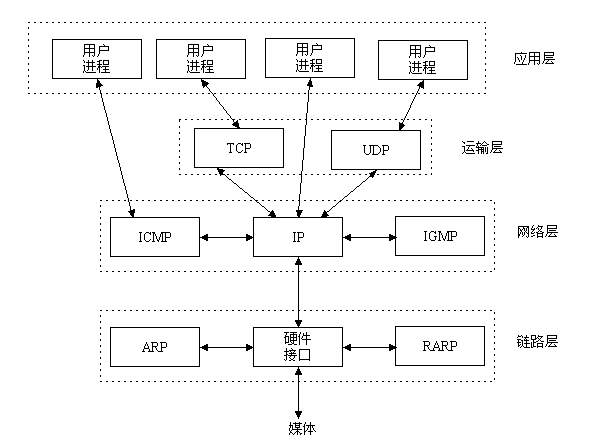
# TCP/IP协议簇

TCP/IP（Transaction Control Protocol/Internet Protocol）传输控制协议/网间协议，是一个工业标准的协议集，它是为广域网（WANs）设计的。

UDP（User Data Protocol，用户数据报协议）是与TCP相对应的协议，它属于TCP/IP协议簇中的一种。

TCP/IP协议族包括运输层、网络层、链路层。



问：网络中的进程如何通信？

在本地可以通过进程PID唯一标识一个进程，但是在网络中不行。

其实TCP/IP协议簇已经帮我们解决了这个问题：“网络层的IP地址”可以唯一标识网络中的主机，而传输层的“协议+端口”可以唯一标识主机中的应用程序（进程），这样利用三元组（IP地址，协议，端口）就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其它进程进行交互。

使用TCP/IP协议的应用程序通常采用应用编程接口：套接字(Socket)，来实现网路进程之间的通信。就目前而言，几乎所有的应用程序都是采用socket，而现在又是网络时代，网络中进程通信是无处不在，这就是我为什么说“一切皆socket”。

# Socket介绍

进程间通信通过socket来进行。

先从服务器端说起。服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。



## Socket工作原理

各层之间的协议：

应用层：http协议

传输层：tcp协议，socket属于传输层的这么一个变成规范

网络层：ip协议

数据链路层：以太网协议

Socket工作原理：

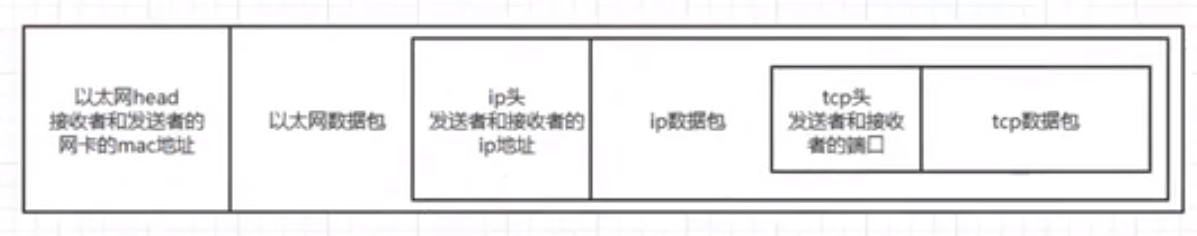
Socket在应用层和传输层之间的一个抽象层，它把 TCP/IP 层复杂的操作抽象为几个简单的接口，供应用层调用实现进程在网络中的通信。

说白了，socket就是在传输层把tcp/ip协议给封装了一下，我们程序员一般都是面向socket来编程的。

大体来说这个步骤，就是我们创建一个ServerSocket来无限等待别人来连接你，然后某个机器要跟你连接，就在本地创建一个socket去连接你，然后与服务器建立连接后，ServerSocket也会创建一个socket。通过客户端的socket与服务端的socket进行通信我给你写数据，你读数据，你给我写数据，我读数据，就这个过程。

其它底层，如建立连接和释放连接，都是基于tcp三次握手和四次挥手的规范来实现的。数据传输也是基于tcp协议的，把数据封装在tcp数据包，里面包括有tcp报头、端口号这些，然后封装在ip数据包，最后封装在以太网数据包进行传输。





# 三种IO介绍

主要有三种IO模型：BIO、NIO、AIO

**BIO、NIO、AIO代码demo：（网上找）**

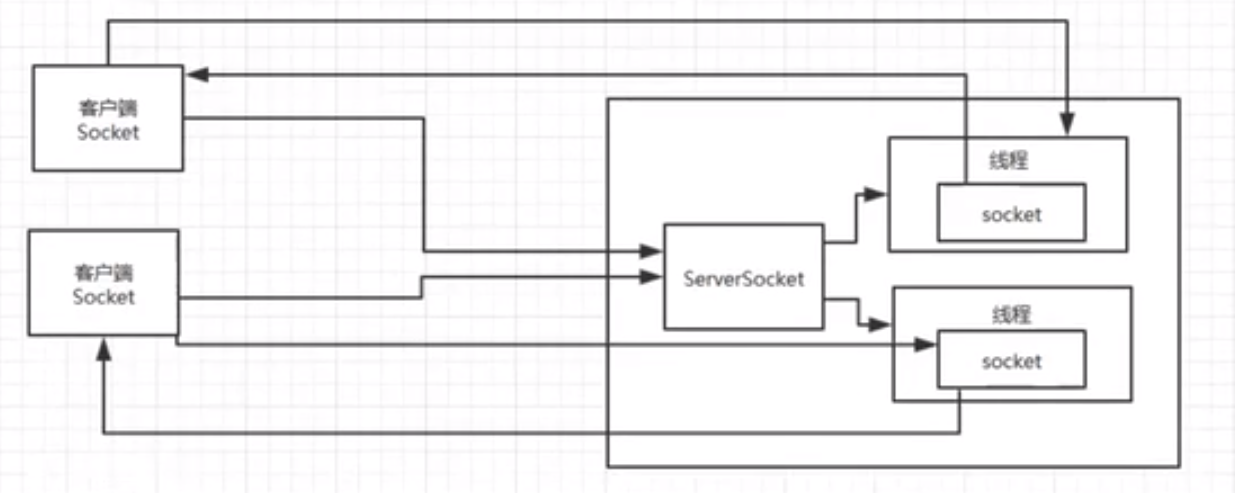
## BIO

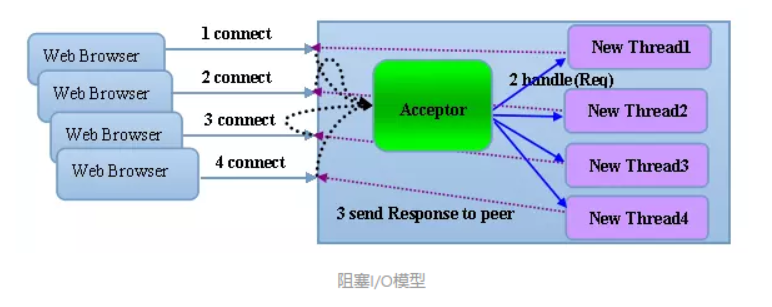
BIO是最传统的网络通信协议，同步阻塞式IO。就是服务端创建一个ServerSocket，然后客户端用一个Socket去连接该ServerSocket，然后ServerSocket接收到一个Socket的连接请求就创建一个Socket和一个线程去跟客户端Socket进行通信。

客户端和服务端的socket进行同步阻塞式的通信，客户端socket发送一个请求，服务端socket进行处理后返回响应，响应必须是等处理完后才会返回，在此之前啥事也干不了，这就是同步。

这个坑在于，每次一个客户端接入，都是要在服务端创建一个线程来服务这个客户端的，这会导致服务端的线程数量过大，而导致服务端程序的负载过高，最后宕机。

优化：可以搞一个线程池来处理请求，但是高并发请求时，还是可能会导致各种排队和延时，因为没那么多线程来处理。





阻塞IO模型：

在调用ServerSocket.accept()方法时，会一直阻塞到客户端连接才会返回。

在调用InputStream.read()方法是阻塞的，它会一直等到数据到来是（或超时）才会返回；

两点缺点：当客户端多时，会创建大量的处理线程，且每个线程都要占用栈空间和一些CPU时间，且大量线程会带来频繁的上下文切换。

## NIO

JDK1.4中引入NIO，这是一种同步非阻塞IO，基于Reactor模型。

Java NIO采用了双向通道（channel）进行数据传输。

### NIO中的一些概念

Channel：NIO都是通过Channel进行数据读写的。

Buffer：缓冲区，将数据写入Buffer中，然后从Buffer中读取数据。

Selector：多路复用器，selector会不断轮询注册的channel，如果某个channel上发送了读写事件，selector就会将这些channel获取出来，我们通过SelectionKey获取有读写事件的channel，就可以进行IO操作了。

一个Selector就使用一个线程，去轮询成千上万个channel，这就意味着你的服务端可以接入成千上万个客户端。

BIO的服务端接收到客户端的socket连接后，会创建一个socket和一个线程去跟客户端socket通信；而NIO的服务端接收到客户端socket连接后，只创建一个连接并注册到select，等selector轮询channel有读写事件时，再创建线程去进行IO操作。

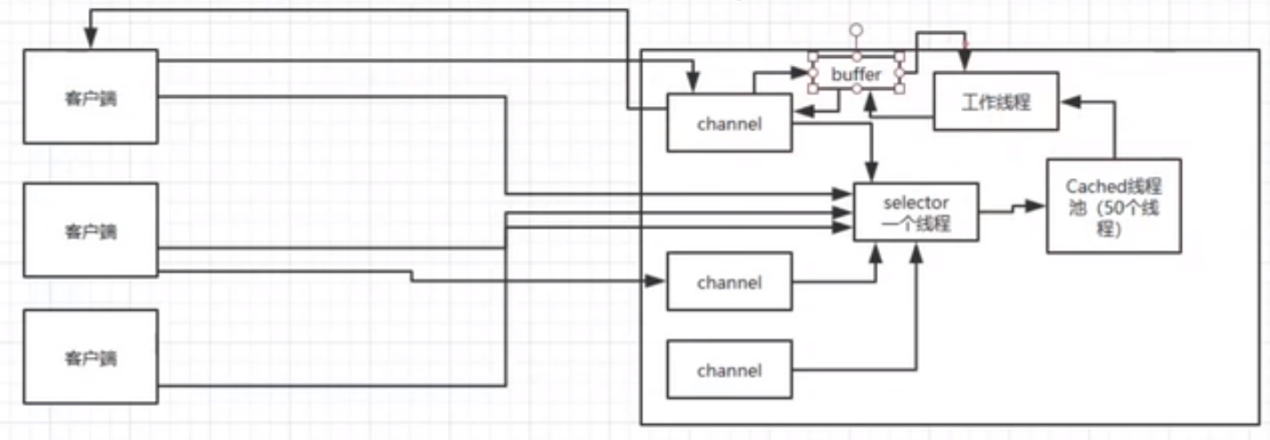
原因是一个客户端不是时时刻刻都要发送请求的，没必要死耗着一个线程不放。所以NIO的优化思想是一个请求一个线程。只有某个客户端发送了一个请求时，才会启动一个线程来处理。

问：为什么NIO是同步非阻塞的呢？

非阻塞理解：因为无论多少客户端都可以接入服务端，客户端接入并不会耗费一个线程只会创建一个连接然后注册到selector上，一个selector线程不断的轮询所有的socket连接，发现有事件了就会通知你，然后你就启动一个线程去处理该请求即可。但是这个处理过程中，你还是要先读取数据，处理，再返回的，这是个同步的过程。

工作线程向channel进行读写操作都是同步进行的：

从channel里读取数据，是工作线程自己去干这个事情，数据没读完，你就卡在那里；然后往channel里写数据，也是你自己去干的事情，数据没写完，你就卡在那里。



### 工作原理

每次客户端建立好连接，服务端就会创建一个对应的Channel并注册到selector上，然后selector轮询这些所有Channel，channel有请求过来的时候，它才创建一个工作线程去处理这个请求，处理后得到的结果也写入到该channel中，然后返回给客户端socket。

## AIO

AIO是基于Proactor模型的，就是异步非阻塞模型。

每个连接发送过来的请求，都会绑定一个buffer，然后通知操作系统去异步完成读，此时你的程序是会去干别的事情，等操作系统完成数据读取之后，就会回调你的接口，给你操作系统异步读完的数据。然后你对这个数据进行处理，接着把结果往回写。写的时候也是给操作系统一个buffer，让操作系统自己获取数据去完成写操作，写完之后再回来通知你。

工作线程的读数据和写数据：

读取数据时，工作线程会给操作系统提供一个空的buffer，然后就可以干别的事情了，也就是说把读数据的事情交给操作系统去干了。操作系统内核，将数据放入buffer后，就回调你的一个接口，告诉你以读取完毕。

写数据也是一样，把放了数据的buffer交给操作系统内核去处理，你就可以去干别的事情了，操作系统完成了数据的写之后，就会回调你，告诉你说，你交给我的数据，我都给你写回到客户端去了。

## BIO、NIO、AIO总结

1.BIO：

同步阻塞的IO模型，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务端就会启动一个线程去处理。

缺点：若连接不做任何事情会造成不必要的线程开销，可使用线程池机制改善。

BIO方式适用于连接数目比较小且固定的架构。

2.NIO：

描述：同步非阻塞的IO模型，服务器实现模式为一个请求一个线程，即客户端发送的连接都会注册到多路复用器上，多路复用器轮询到连接有IO请求时才启动一个线程进行处理。

NIO也是IO多路复用的基础，是解决高并发与大量连接、IO处理问题的有效方式。

NIO适用于连接数多且连接比较短（轻架构）的架构。比如聊天服务器，并发局限于应用中。

3.AIO：

描述：异步非阻塞的IO模型，服务器实现模式为一个有效请求一个线程，即客户端的IO请求都是由OS先完成了，再通知服务器应用区启动线程去处理。

AIO适用于连接数多且连接比较长（重架构）的架构。比如相册服务器，充分调用OS参与并发操作。