# Java网络编程

## 网络通信

### 概念

网络编程的本质是两个设备之间的数据交换，在[计算机网络](http://lib.csdn.net/base/computernetworks" \t "_blank" \o "计算机网络知识库)中，设备主要指计算机。数据传递就是把一个设备中的数据发送给列外一个设备，然后接受另外一个设备反馈的数据。

现在的网络编程基本上都是基于请求/响应方式的，也就是一个设备发送请求数据给另外一个，然后接收另一个设备的反馈。

### 服务端与客户端

在网络编程中，发起连接程序，也就是发送第一次请求的程序，被称作客户端(Client)，等待其他程序连接的程序被称作服务器(Server)。客户端程序可以在需要的时候启动，而服务器为了能够时刻相应连接，则需要一直启动。

连接一旦建立以后，就客户端和服务器端就可以进行数据传递了，而且两者的身份是等价的。

### IP地址

IP地址是一个规定，现在使用的是IPv4，既由4个0-255之间的数字组成，在计算机内部存储时只需要4个字节即可。在计算机中，IP地址是分配给网卡的，每个网卡有一个唯一的IP地址，如果一个计算机有多个网卡，则该台计算机则拥有多个不同的IP地址，在同一个网络内部，IP地址不能相同。IP地址的概念类似于电话号码、身份证这样的概念。

### 域名

由于IP地址不方便记忆，所以有专门创造了域名(Domain Name)的概念，其实就是给IP取一个字符的名字，例如163.com、sina.com等。IP和域名之间存在一定的对应关系。如果把IP地址类比成身份证号的话，那么域名就是你的姓名。

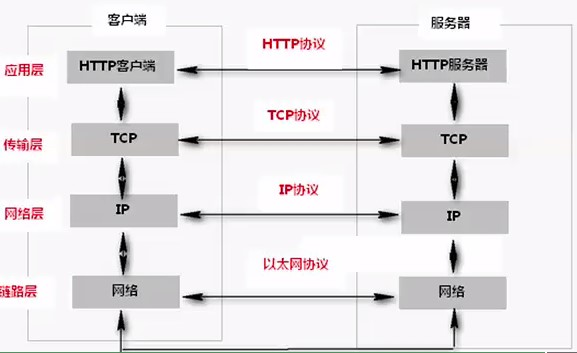
其实在网络中只能使用IP地址进行数据传输，所以在传输以前，需要把域名转换为IP，这个由称作DNS（**D**omain **N**ame **S**ystem:域名系统）的服务器解析。在网络编程中，使用IP或域名来标识网络上的一台设备

端口

为了在一台设备上可以运行多个程序，人为的设计了端口(Port)的概念，即ip找到具体主机再根据端口找到这个主机上的具体应用程序。规定一个设备有216个，也就是65536个端口，每个端口对应一个唯一的程序。每个网络程序，无论是客户端还是服务器端，都对应一个或多个特定的端口号。由于0-1024之间多被[操作系统](http://lib.csdn.net/base/operatingsystem" \t "_blank" \o "操作系统知识库)占用，所以实际编程时一般采用1024以后的端口号。使用端口号，可以找到一台设备上唯一的一个程序。

所以如果需要和某台计算机建立连接的话，只需要知道IP地址或域名即可，但是如果想和该台计算机上的某个程序交换数据的话，还必须知道该程序使用的端口号。

### 网络模型图



### tcp/ip协议族

网络通信个层通信协议的统称。

## TCP/UDP

### 概念

TCP（Transmission Control Protocol [传输控制协议](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E6%8E%A7%E5%88%B6%E5%8D%8F%E8%AE%AE/9727741" \t "https://baike.baidu.com/item/TCP/_blank)）是一种面向连接的、可靠的、基于字节流的[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82/4329536" \t "https://baike.baidu.com/item/TCP/_blank)通信协议

UDP (User Datagram Protocol用户数据报协议)是一种无连接的[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82" \t "https://baike.baidu.com/item/UDP/_blank)通信协议

### Tcp和udp的区别

udp:

* 是面向无连接, 将数据及源的封装成数据包中,不需要建立建立连接
* 每个数据报的大小在限制64k内
* 因无连接,是不可靠协议
* 不需要建立连接,速度快，如传长视频用此协议

tcp：

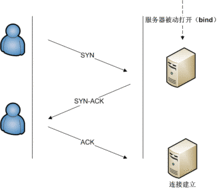
* 建议连接，形成传输数据的通道.
* 在连接中进行大数据量传输，以字节流方式
* 通过三次握手完成连接,是可靠协议
* 必须建立连接m效率会稍低

### Tcp连接和终止

连接建立

TCP是因特网中的传输层协议，使用[三次握手协议](https://baike.baidu.com/item/%E4%B8%89%E6%AC%A1%E6%8F%A1%E6%89%8B%E5%8D%8F%E8%AE%AE" \t "https://baike.baidu.com/item/TCP/_blank)建立连接。当主动方发出SYN连接请求后，等待对方回答SYN+ACK [1] ，并最终对对方的 SYN 执行 ACK 确认。这种建立连接的方法可以防止产生错误的连接，TCP使用的流量控制协议是可变大小的滑动窗口协议。

TCP三次握手的过程如下：

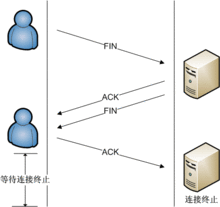
[](https://baike.baidu.com/pic/TCP/33012/0/a1ad16fa330e9cae59ee90ca?fr=lemma%26ct=single)

1. 客户端发送SYN（SEQ=x）报文给服务器端，进入SYN\_SEND状态。
2. 服务器端收到SYN报文，回应一个SYN （SEQ=y）ACK(ACK=x+1）报文，进入[SYN\_RECV](https://baike.baidu.com/item/SYN_RECV" \t "https://baike.baidu.com/item/TCP/_blank)状态。
3. 客户端收到服务器端的SYN报文，回应一个ACK(ACK=y+1）报文，进入Established状态。

三次握手完成，TCP客户端和服务器端成功地建立连接，可以开始传输数据了。

连接终止

建立一个连接需要三次握手，而终止一个连接要经过四次握手，这是由TCP的半关闭（half-close）造成的。具体过程如下图所示。

[](https://baike.baidu.com/pic/TCP/33012/0/bf4875638e06d1590c33faf9?fr=lemma%26ct=single)

1. 某个应用进程首先调用close，称该端执行“主动关闭”（active close）。该端的TCP于是发送一个FIN分节，表示数据发送完毕。
2. 接收到这个FIN的对端执行 “被动关闭”（passive close），这个FIN由TCP确认。
3. 注意：FIN的接收也作为一个文件结束符（end-of-file）传递给接收端应用进程，放在已排队等候该应用进程接收的任何其他数据之后，因为，FIN的接收意味着接收端应用进程在相应连接上再无额外数据可接收。
4. 一段时间后，接收到这个文件结束符的应用进程将调用close关闭它的套接字。这导致它的TCP也发送一个FIN。接收这个最终FIN的原发送端TCP（即执行主动关闭的那一端）确认这个FIN。

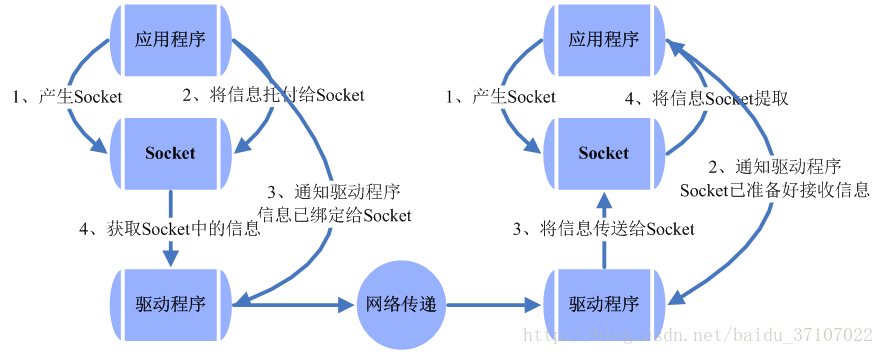
既然每个方向都需要一个FIN和一个ACK，因此通常需要4个分节。

## Socket入门

### Socket概念

Socket（套接字/插座）就是为网络服务提供的一种机制。通讯的两端都有Sokcet，网络通讯其实就是Sokcet间的通讯，数据在两个Sokcet间通过IO传输。

### Java中socket类通信原理



### 网络编程示例

基于tcp协议

#### IO方式

|  |
| --- |
| public class IOServer {  public static void main(String[] args) throws IOException {  //线程池管理线程  ExecutorService pool = Executors.newCachedThreadPool();  //新的 try-with-resources 语句，称为 ARM 块(Automatic Resource Management) ，自动资源管理,不用手动close。  try (ServerSocket serverSocket = new ServerSocket(8000)) {  while (true) {  //阻塞方法获取新的连接  Socket socket = serverSocket.accept();  // 每一个新的连接都创建一个任务，负责读取数据  pool.execute(() -> {  InputStream inputStream = null;  try {  inputStream = socket.getInputStream();  byte[] bytes = new byte[1024];  int length = -1;  // 按字节流方式读取数据  while ((length = inputStream.read(bytes)) != -1) {  System.out.println(new String(bytes, 0, length));  }  } catch (IOException e) {  e.printStackTrace();  } finally {  if (inputStream != null) {  try {  inputStream.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  });  }  } catch (Exception e) {  e.printStackTrace();  }  } } |
| public class IOClient {  public static void main(String[] args) throws IOException {  new Thread(() -> {  try {  Socket socket = new Socket("127.0.0.1", 8000);  while (true) {  try {  socket.getOutputStream().write((new Date() + ": hello world").getBytes());  socket.getOutputStream().flush();  //模拟每隔2秒发送一次请求  Thread.sleep(2000);  } catch (Exception e) {  }  }  } catch (IOException e) {  }  }).start();  } } |

总结：

客户端较少的情况下运行良好，但是对于客户端比较多的业务来说，单机服务端可能需要支撑成千上万的连接，IO模型可能就不太合适了

· 线程资源受限：线程是操作系统中非常宝贵的资源，同一时刻有大量的线程处于阻塞状态是非常严重的资源浪费，操作系统耗不起

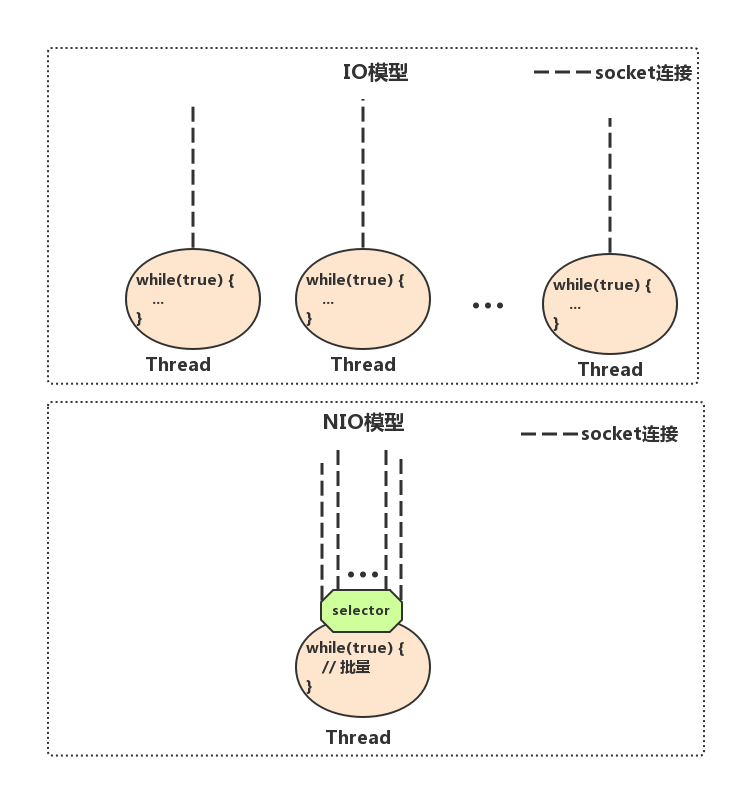
· 线程切换效率低下：单机cpu核数固定，线程爆炸之后操作系统频繁进行线程切换，应用性能急剧下降。

· 除了以上两个问题，IO编程中，我们看到数据读写是以字节流为单位，效率不高。

为了解决这三个问题，JDK在1.4之后提出了NIO。

#### NIO方式

解决资源受限



|  |
| --- |
| public class NIOServer {  public static void main(String[] args) throws IOException {  Selector serverSelector = Selector.open();  Selector clientSelector = Selector.open();  new Thread(() -> {  try {  // 对应IO编程中服务端启动  ServerSocketChannel listenerChannel = ServerSocketChannel.open();  listenerChannel.socket().bind(new InetSocketAddress(8000));  listenerChannel.configureBlocking(false);  listenerChannel.register(serverSelector, SelectionKey.OP\_ACCEPT);   while (true) {  // 监测是否有新的连接，这里的1指的是阻塞的时间为1ms  if (serverSelector.select(1) > 0) {  Set<SelectionKey> set = serverSelector.selectedKeys();  Iterator<SelectionKey> keyIterator = set.iterator();   while (keyIterator.hasNext()) {  SelectionKey key = keyIterator.next();   if (key.isAcceptable()) {  try {  // (1) 每来一个新连接，不需要创建一个线程，而是直接注册到clientSelector  SocketChannel clientChannel = ((ServerSocketChannel) key.channel()).accept();  clientChannel.configureBlocking(false);  clientChannel.register(clientSelector, SelectionKey.OP\_READ);  } finally {  keyIterator.remove();  }  }   }  }  }  } catch (IOException ignored) {  }   }).start();  } } |
| public class NIOClient {  public static void main(String[] args) throws IOException {  new Thread(() -> {  try {  while (true) {  // (2) 批量轮询是否有哪些连接有数据可读，这里的1指的是阻塞的时间为1ms  if (clientSelector.select(1) > 0) {  Set<SelectionKey> set = clientSelector.selectedKeys();  Iterator<SelectionKey> keyIterator = set.iterator();   while (keyIterator.hasNext()) {  SelectionKey key = keyIterator.next();   if (key.isReadable()) {  try {  SocketChannel clientChannel = (SocketChannel) key.channel();  ByteBuffer byteBuffer = ByteBuffer.allocate(1024);  // (3) 读取数据以块为单位批量读取  clientChannel.read(byteBuffer);  byteBuffer.flip();  System.out.println(Charset.defaultCharset().newDecoder().decode(byteBuffer)  .toString());  } finally {  keyIterator.remove();  key.interestOps(SelectionKey.OP\_READ);  }  }   }  }  }  } catch (IOException ignored) {  }  }).start();  } } |

总结：

· NIO模型中通常会有两个线程，每个线程绑定一个轮询器selector，在我们这个例子中serverSelector负责轮询是否有新的连接，clientSelector负责轮询连接是否有数据可读

· 服务端监测到新的连接之后，不再创建一个新的线程，而是直接将新连接绑定到clientSelector上，这样就不用IO模型中1w个while循环在死等，参见(1)

· clientSelector被一个while死循环包裹着，如果在某一时刻有多条连接有数据可读，那么通过 clientSelector.select(1)方法可以轮询出来，进而批量处理，参见(2)

· 数据的读写以内存块为单位，参见(3)

JDK的NIO犹如带刺的玫瑰，虽然美好，让人向往，但是使用不当会让你抓耳挠腮，痛不欲生，正因为如此，Netty横空出世！

#### Netty框架方式

|  |
| --- |
| public class NettyServer {  public static void main(String[] args) {  ServerBootstrap serverBootstrap = new ServerBootstrap();  NioEventLoopGroup boos = new NioEventLoopGroup();  NioEventLoopGroup worker = new NioEventLoopGroup();  serverBootstrap  .group(boos, worker)  .channel(NioServerSocketChannel.class)  .childHandler(new ChannelInitializer<NioSocketChannel>() {  protected void initChannel(NioSocketChannel ch) {  ch.pipeline().addLast(new StringDecoder());  ch.pipeline().addLast(new SimpleChannelInboundHandler<String>() {  @Override  protected void channelRead0(ChannelHandlerContext ctx, String msg) {  System.out.println(msg);  }  });  }  })  .bind(8000);  }  } |
| public class NettyClient {  public static void main(String[] args) throws InterruptedException {  Bootstrap bootstrap = new Bootstrap();  NioEventLoopGroup group = new NioEventLoopGroup();  bootstrap.group(group)  .channel(NioSocketChannel.class)  .handler(new ChannelInitializer<Channel>() {  @Override  protected void initChannel(Channel ch) {  ch.pipeline().addLast(new StringEncoder());  }  });  Channel channel = bootstrap.connect("127.0.0.1", 8000).channel();  while (true) {  channel.writeAndFlush(new Date() + ": hello world!");  Thread.sleep(2000);  }  }  } |