Socket 编程

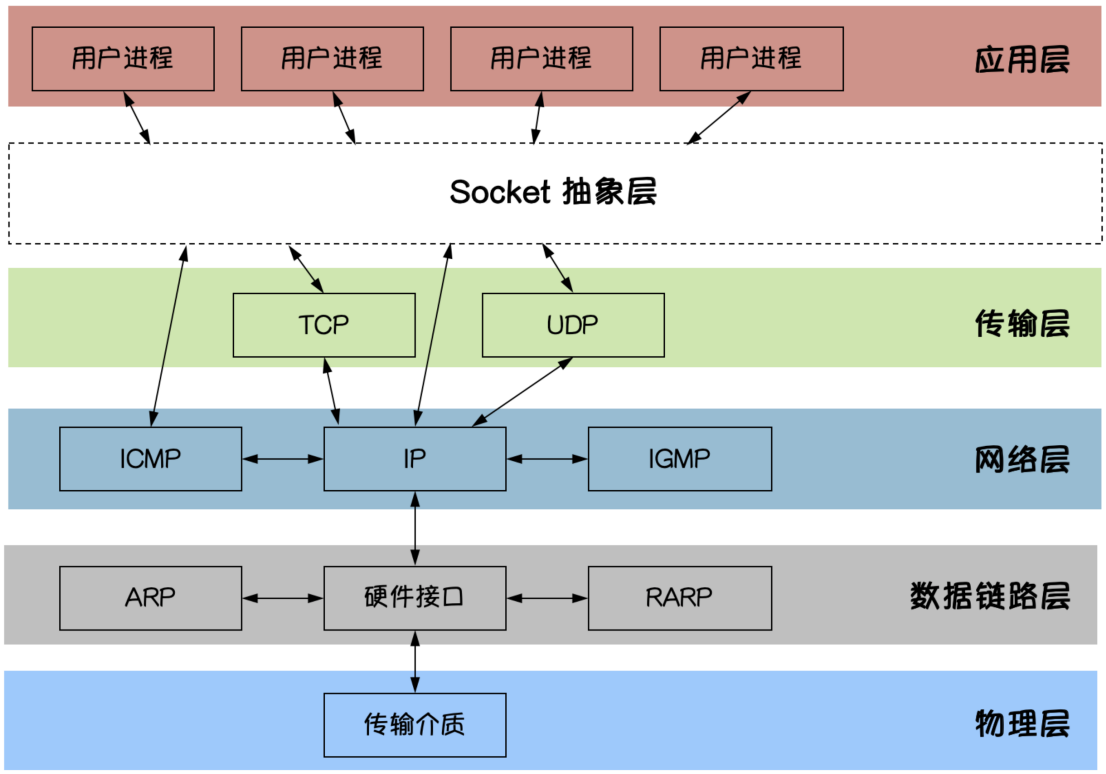
Socket是BSD UNIX的进程通信机制，通常也称作"套接字"，用于描述IP地址和端口，是一个通信链的句柄。Socket可以理解为TCP/IP网络的API，它定义了许多函数或例程，程序员可以用它们来开发TCP/IP网络上的应用程序。电脑上运行的应用程序通常通过"套接字"向网络发出请求或者应答网络请求。

Socket起源于Unix，而Unix基本哲学之一就是"一切皆文件"，都可以用"打开open –> 读写write/read –> 关闭close"模式来操作。Socket就是该模式的一个实现，网络的Socket数据传输是一种特殊的I/O，Socket也是一种文件描述符。Socket也具有一个类似于打开文件的函数调用：Socket()，该函数返回一个整型的Socket描述符，随后的连接建立、数据传输等操作都是通过该Socket实现的。

常用的Socket类型有两种：流式Socket（SOCK_STREAM）和数据报式Socket（SOCK_DGRAM）。流式是一种面向连接的Socket，针对于面向连接的TCP服务应用；数据报式Socket是一种无连接的Socket，对应于无连接的UDP服务应用。

Socket图解

Socket是应用层与TCP/IP协议族通信的中间软件抽象层。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket后面，对用户来说只需要调用Socket规定的相关函数，让Socket去组织符合指定的协议数据然后进行通信。

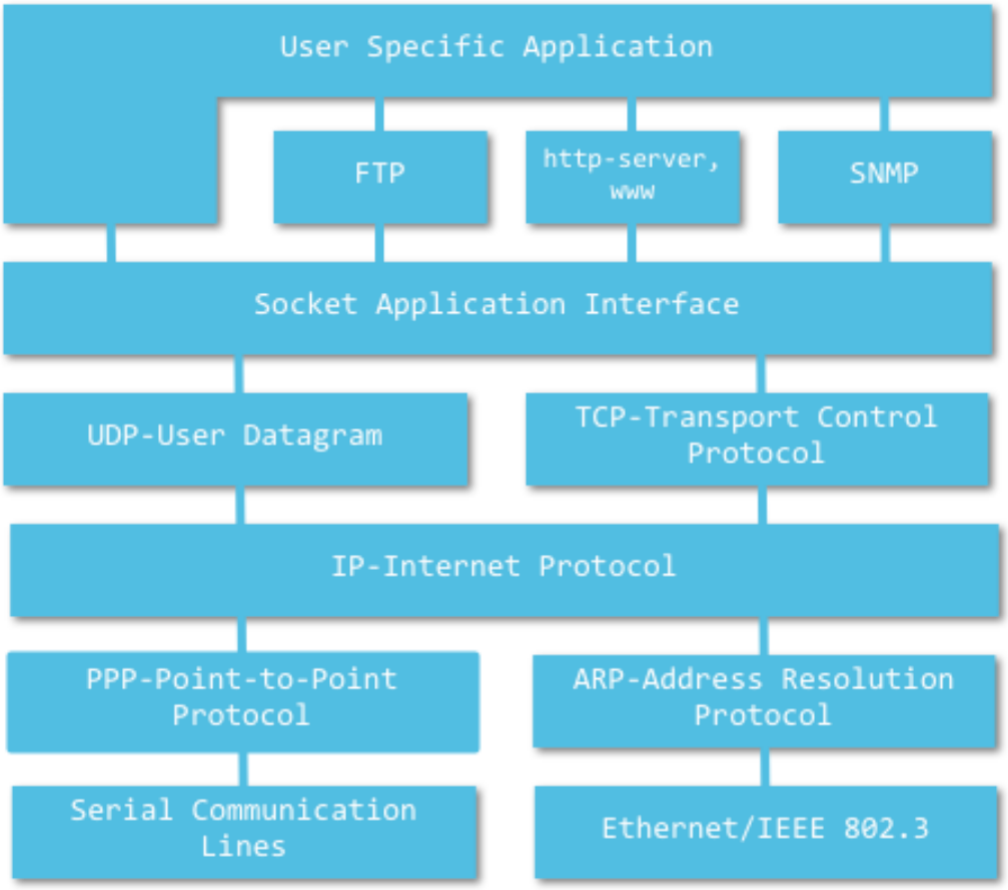


- Socket又称"套接字"，应用程序通常通过"套接字"向网络发出请求或者应答网络请求
- 常用的Socket类型有两种：流式Socket和数据报式Socket，流式是一种面向连接的Socket，针对于面向连接的TCP服务应用，数据报式Socket是一种无连接的Socket，针对于无连接的UDP服务应用
- TCP：比较靠谱，面向连接，比较慢
- UDP：不是太靠谱，比较快

举个例子：TCP就像货到付款的快递，送到家还必须见到你人才算一整套流程。UDP就像某快递快递柜一扔就走管你收到收不到，一般直播用UDP。

Socket如何通信

网络中的进程之间如何通过Socket通信呢？首要解决的问题是如何唯一标识一个进程，否则通信无从谈起！在本地可以通过进程PID来唯一标识一个进程，但是在网络中这是行不通的。其实TCP/IP协议族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，网络中需要互相通信的进程，就可以利用这个标志在他们之间进行交互。



使用TCP/IP协议的应用程序通常采用应用编程接口：UNIX BSD的套接字（socket）和UNIX System V的TLI（已经被淘汰），来实现网络进程之间的通信。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，这就是为什么说“一切皆Socket”。

Socket基础知识

通过上面的介绍我们知道Socket有两种：TCP Socket和UDP Socket，TCP和UDP是协议，而要确定一个进程的需要三元组，需要IP地址和端口。

IPv4地址

目前的全球因特网所采用的协议族是TCP/IP协议。IP是TCP/IP协议中网络层的协议，是TCP/IP协议族的核心协议。目前主要采用的IP协议的版本号是4(简称为IPv4)，发展至今已经使用了30多年。

IPv4的地址位数为32位，也就是最多有2的32次方的网络设备可以联到Internet上。近十年来由于互联网的蓬勃发展，IP位址的需求量愈来愈大，使得IP位址的发放愈趋紧张，前一段时间，据报道IPV4的地址已经发放完毕，我们公司目前很多服务器的IP都是一个宝贵的资源。

地址格式类似这样：127.0.0.1 172.122.121.111

IPv6地址

IPv6是下一版本的互联网协议，也可以说是下一代互联网的协议，它是为了解决IPv4在实施过程中遇到的各种问题而被提出的，IPv6采用128位地址长度，几乎可以不受限制地提供地址。按保守方法估算IPv6实际可分配的地址，整个地球的每平方米面积上仍可分配1000多个地址。在IPv6的设计过程中除了一劳永逸地解决了地址短缺问题以外，还考虑了在IPv4中解决不好的其它问题，主要有端到端IP连接、服务质量（QoS）、安全性、多播、移动性、即插即用等。

地址格式类似这样：2002:c0e8:82e7:0:0:0:c0e8:82e7

Go支持的IP类型

在Go的`net`包中定义了很多类型、函数和方法用来网络编程，其中IP的定义如下：

```
type IP []byte
```

在`net`包中有很多函数来操作IP，但是其中比较有用的也就几个，其中`ParseIP(s string) IP`函数会把一个IPv4或者IPv6的地址转化成IP类型。

TCP 编程

TCP协议

TCP/IP(Transmission Control Protocol/Internet Protocol) 即传输控制协议/网间协议，是一种面向连接（连接导向）的、可靠的、基于字节流的传输层（Transport layer）通信协议，因为是面向连接的协议，数据像水流一样传输，会存在黏包问题。

TCP服务端

一个TCP服务端可以同时连接很多个客户端，例如世界各地的用户使用自己电脑上的浏览器访问淘宝网。因为Go语言中创建多个goroutine实现并发非常方便和高效，所以我们可以每建立一次链接就创建一个goroutine去处理。

TCP服务端程序的处理流程：

1. 监听端口
2. 接收客户端请求建立链接
3. 创建goroutine处理链接。

可以通过net包来创建一个服务器端程序，在服务器端我们需要绑定服务到指定的非激活端口，并监听此端口，当有客户端请求到达的时候可以接收到来自客户端连接的请求。net包中有相应功能的函数，函数定义如下：

```
func ListenTCP(network string, laddr *TCPAddr) (*TCPListener, error)
func (l *TCPListener) Accept() (Conn, error)
```

- net参数是"tcp4"、"tcp6"、"tcp"中的任意一个，分别表示TCP(IPv4-only)、TCP(IPv6-only)或者TCP(IPv4,IPv6的任意一个)
- laddr表示本机地址，一般设置为nil
- raddr表示远程的服务地址

示例：

```
```

```
// tcp\server\server.go
// TCP server端

package main

import (
    "bufio"
    "fmt"
    "net"
)

// 处理函数
func process(conn net.Conn) {
    defer conn.Close() // 关闭连接
    for {
        reader := bufio.NewReader(conn)
        var buf [128]byte
        n, err := reader.Read(buf[:]) // 读取数据
        if err != nil {
            fmt.Println("read from client failed, err:", err)
            break
        }
        recvStr := string(buf[:n])
        fmt.Println("收到client端发来的数据：", recvStr)
        conn.Write([]byte(recvStr)) // 发送数据
    }
}

func main() {
    listen, err := net.Listen("tcp", "127.0.0.1:20000")
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    for {
        conn, err := listen.Accept() // 建立连接
        if err != nil {
            fmt.Println("accept failed, err:", err)
            continue
        }
        go process(conn) // 启动一个goroutine处理连接
    }
}
```

TCP客户端

一个TCP客户端进行TCP通信的流程如下：

1. 建立与服务端的链接
2. 进行数据收发
3. 关闭链接

Go语言中通过net包中的`DialTCP`函数来建立一个TCP连接，并返回一个`TCPConn`类型的对象，当连接建立时服务器端也创建一个同类型的对象，此时客户端和服务器段通过各自拥有的`TCPConn`对象来进行数据交换。一般而言，客户端通过`TCPConn`对象将请求信息发送到服务器端，读取服务器端响应的信息。服务器端读取并解析来自客户端的请求，并返回应答信息，这个连接只有当任一端关闭了连接之后才失效，不然这连接可以一直在使用。建立连接的函数定义如下：

```
func DialTCP(network string, laddr, raddr *TCPAddr) (*TCPConn, error)
```

- net参数是"tcp4"、"tcp6"、"tcp"中的任意一个，分别表示TCP(IPv4-only)、TCP(IPv6-only)或者TCP(IPv4,IPv6的任意一个)
- laddr表示本机地址，一般设置为nil
- raddr表示远程的服务地址

示例：

```
// tcp\client\client.go
// TCP client端

package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)

// 客户端
func main() {
    conn, err := net.Dial("tcp", "127.0.0.1:20000")
    if err != nil {
        fmt.Println("err :", err)
        return
    }
    defer conn.Close() // 关闭连接
    inputReader := bufio.NewReader(os.Stdin)
    for {
        input, _ := inputReader.ReadString('\n') // 读取用户输入
        inputInfo := strings.Trim(input, "\r\n")
        if strings.ToUpper(inputInfo) == "Q" { // 如果输入q就退出
            return
        }
        _, err = conn.Write([]byte(inputInfo)) // 发送数据
        if err != nil {
            return
        }
        buf := [512]byte{}
        n, err := conn.Read(buf[:])
        if err != nil {
            fmt.Println("recv failed, err:", err)
            return
        }
        fmt.Println(string(buf[:n]))
```

```
    }
}
```

先启动server端再启动client端，在client端输入任意内容回车之后就能够在server端看到client端发送的数据，从而实现TCP通信。

控制TCP连接

TCP有很多连接控制函数，我们平常用到比较多的有如下几个函数：

- 设置建立连接的超时时间，客户端和服务器端都适用，当超过设置时间时，连接自动关闭。

```
func DialTimeout(net, addr string, timeout time.Duration) (Conn, error)
```

- 用来设置写入/读取一个连接的超时时间。当超过设置时间时，连接自动关闭。

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

- 设置keepAlive属性，是操作系统层在tcp上没有数据和ACK的时候，会间隔性的发送keepalive包，操作系统可以通过该包来判断一个tcp连接是否已经断开，在windows上默认2个小时没有收到数据和keepalive包的时候人为tcp连接已经断开，这个功能和我们通常在应用层加的心跳包的功能类似。

```
func (c *TCPConn) SetKeepAlive(keepalive bool) os.Error
```

UDP 编程

UDP协议

UDP协议（User Datagram Protocol）中文名称是用户数据报协议，是OSI（Open System Interconnection，开放式系统互联）参考模型中一种无连接的传输层协议，不需要建立连接就能直接进行数据发送和接收，属于不可靠的、没有时序的通信，但是UDP协议的实时性比较好，通常用于视频直播相关领域。

Go语言包中处理UDP Socket和TCP Socket不同的地方就是在服务器端处理多个客户端请求数据包的方式不同,UDP缺少了对客户端连接请求的Accept函数。其他基本几乎一模一样，只有TCP换成了UDP而已。UDP的几个主要函数如下所示：

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, os.Error)
func DialUDP(net string, laddr, raddr *UDPAddr) (c *UDPConn, err os.Error)
func ListenUDP(net string, laddr *UDPAddr) (c *UDPConn, err os.Error)
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err os.Error)
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (n int, err os.Error)
```

UDP服务端

使用Go语言的net包实现的UDP服务端代码如下：

```
// UDP/server/server.go

// UDP server端
```

```
package main

import (
    "fmt"
    "net"
)

func main() {
    listen, err := net.ListenUDP("udp", &net.UDPAddr{
        IP:   net.IPv4(0, 0, 0, 0),
        Port: 30000,
    })
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
    }
    defer listen.Close()
    for {
        var data [1024]byte
        n, addr, err := listen.ReadFromUDP(data[:]) // 接收数据
        if err != nil {
            fmt.Println("read udp failed, err:", err)
            continue
        }
        fmt.Printf("data:%v addr:%v count:%v\n", string(data[:n]), addr, n)
        _, err = listen.WriteToUDP(data[:n], addr) // 发送数据
        if err != nil {
            fmt.Println("write to udp failed, err:", err)
            continue
        }
    }
}
```

UDP客户端

使用Go语言的net包实现的UDP客户端代码如下：

```
// UDP/client/client.go

// UDP client端

package main

import (
    "fmt"
    "net"
)

// UDP 客户端
func main() {
    socket, err := net.DialUDP("udp", nil, &net.UDPAddr{
        IP:   net.IPv4(0, 0, 0, 0),
        Port: 30000,
    })
```

```
    if err != nil {
        fmt.Println("连接服务端失败，err:", err)
        return
    }
    defer socket.Close()
    sendData := []byte("Hello server")
    _, err = socket.Write(sendData) // 发送数据
    if err != nil {
        fmt.Println("发送数据失败，err:", err)
        return
    }
    data := make([]byte, 4096)
    n, remoteAddr, err := socket.ReadFromUDP(data) // 接收数据
    if err != nil {
        fmt.Println("接收数据失败，err:", err)
        return
    }
    fmt.Printf("recv:%v addr:%v count:%v\n", string(data[:n]), remoteAddr, n)
}
```