# Go语言网络编程

本章重点为大家介绍如下的内容：

目录：

* 网络编程基础（网络协议、IP、端口）
* Socket编程（TCP Socket与UDP Socket、ICMP）
* HTTP编程
* RPC编程（rpc库、jsonrpc、protorpc）
* JSON处理

## 网络编程基础

### 通信协议

通信协议也叫网络传输协议或简称为传送协议（Communications Protocol），简称协议，网络协议是通信计算机双方必须共同遵从的一组约定。如怎么样建立连接、怎么样互相识别等。只有遵守这个约定，计算机之间才能相互通信交流。比如：语言、不同的厂家的设备通信。

现在最普及的计算机通信为网络通信，所以“传送协议”一般都指计算机通信的传送协议，如：TCP/IP、NetBEUI、HTTP、FTP等。

然而，传送协议也存在于计算机的其他形式通信，例如：面向对象编程里面对象之间的通信；操作系统内不同程序之间的消息，都需要有一个传送协议，以确保传信双方能够沟通。

go 的网络编程模块主要支持两种Internet协议: TCP 和 UDP.

### TCP/IP协议

在Internet中TCP/IP协议是使用最为广泛的通讯协议(互联网上的一种事实的标准)。TCP/IP是英文Transmission Control Protocol/Internet Protocol的缩写，意思是“传输控制协议/网际协议”

TCP/IP 协议是一个工业标准协议套件，专为跨广域网(WAN)的大型互联网络而设计。

TCP/IP 网络体系结构模型就是遵循TCP/IP 协议进行通信的一种分层体系，现今，Internet和Intranet所使用的协议一般都为TCP/IP 协议。

1、国际标准化组织为了规范协议层次的划分制定了OSI参考模型（Open System Interconnect Reference Model，开放式系统互联参考模型）。此模型根据网络功能制定出7层网络协议结构，由低到高分别为物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。

2、基于该协议的体系结构层次，而TCP/IP体系结构分为四层。

* 第 1 层 网络接口层 包括用于协作IP数据在已有网络介质上传输的协议，提供TCP/IP协议的数据结构和实际物理硬件之间的接口。比如地址解析协议(Address Resolution Protocol, ARP )等。
* 第 2 层 网络层 对应于OSI模型的网络层，主要包含了IP、RIP等相关协议，负责数据的打包、寻址及路由。还包括网间控制报文协议(ICMP)来提供网络诊断信息。
* 第 3 层 传输层 对应于OSI的传输层，提供了两种端到端的通信服务，分别是TCP和UDP协议。
* 第 4 层 应用层 对应于OSI的应用层、表达层和会话层，提供了网络与应用之间的对话接口。包含了各种网络应用层协议，比如Http、FTP等应用协议。
  + HTTP全称是HyperText Transfer Protocal，即：超文本传输协议，HTTP连接最显著的特点是客户端发送的每次请求都需要服务器回送响应，在请求结束后，会主动释放连接。从建立连接到关闭连接的过程称为“一次连接”。
  + ftp 是用来进行文件传输的规范，这种规范使用的传输层协议是tcp（可靠传送）
  + tftp 也是用于文件传输的规范，不过一般是小文件，因为它使用的传输层协议是udp（非可靠传送）
  + smtp 是邮件传送规范，用来进行电子邮件传送，这种规范使用的传输层协议是tcp（可靠传送）
  + telnet 是进行字符传送的规范，主要应用在bbs 设备控制，这种规范使用的传输层协议是tcp（可靠传送）
  + dns 是域名解析服务，比如将www.sina.com.cn解析成对应的ip地址（网络通信使用的是ip地址，但是人习惯于记域名），它使用的传输层协议是udp

### IP 地址

1、IP 地址

互联网协议地址（英语：Internet Protocol Address，又译为网际协议地址），缩写为IP地址（英语：IP Address）

IP 地址是分配给网络上使用网际协议（英语：Internet Protocol, IP）的设备的数字标签。常见的IP地址分为IPv4与IPv6两大类。

2、IPV4地址

IP地址由32位二进制数组成，为便于使用，常以XXX.XXX.XXX.XXX形式表现，每组XXX代表小于或等于255的10进制数。例如维基媒体的一个IP地址是208.80.152.2。

地址可分为A、B、C、D、E五大类，其中E类属于特殊保留地址。 IP地址是唯一的。目前IP技术可能使用的IP地址最多可有4,294,967,296个（即232）。骤看可能觉得很难会用尽，但由于早期编码和分配上的问题，使很多区域的编码实际上被空出或不能使用。加上互联网的普及，使大部分家庭都至少有一部电脑，连同公司的电脑，以及连接网络的各种设备都消耗了大量IPv4地址资源。

随着互联网的快速成长，IPv4的42亿个地址的分配最终于2011年2月3日用尽[1][2]。相应的科研组织已研究出128位的IPv6，其IP地址数量最高可达3.402823669 × 1038个，届时每个人家居中的每件电器，每件对象，甚至地球上每一粒沙子都可以拥有自己的IP地址。

在A类、B类、C类IP地址中，如果主机号是全1，那么这个地址为直接广播地址，它是用来使路由器将一个分组以广播形式发送给特定网络上的所有主机。32位全为1的IP地址“255.255.255.255”为受限广播地址（"limited broadcast" destination address），用来将一个分组以广播方式发送给本网络中的所有主机，路由器则阻挡该分组通过，将其广播功能限制在本网内部。

3、IPV6地址

IPv6地址为128位长但通常写作8组每组四个十六进制数的形式。例如： 2001:0db8:85a3:08d3:1319:8a2e:0370:7344 是一个合法的IPv6地址。

IPv4地址可以很容易的转化为IPv6格式。举例来说，如果IPv4的一个地址为135.75.43.52（十六进制为0x874B2B34），它可以被转化为0000:0000:0000:0000:0000:0000:874B:2B34或者::874B:2B34。同时，还可以使用混合符号（IPv4-compatible address），则地址可以为::135.75.43.52。

4、GO支持IP的类型

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，请看下面的例子:

name := "192.168.10.105:8282" addr := net.ParseIP(name) **if** addr == nil { fmt.Println("Invalid address") } **else** { fmt.Println("The address is ", addr.String()) }

执行之后你就会发现只要你输入一个IP地址就会给出相应的IP格式。

### 端口号

1、物理端口

在网络技术中，端口(Port)包括逻辑端口和物理端口两种类型。

物理端口指的是物理存在的端口，如ADSL Modem、集线器、交换机、路由器上用 于连接其他网络设备的接口，如RJ-45端口、SC端口等等。

2、逻辑端口

逻辑端口是指逻辑意义上用于区分服务的端口，如TCP/IP协议中的服务端口，端口号的范围从0到65535，比如用于浏览网页服务的80端口，用于FTP服务的21端口等。由于物理端口和逻辑端口数量较多，为了对端口进行区分，将每个端口进行了编号，这就是端口号。我们主要研究的是逻辑端口号.我们平时所说的端口号也是指的逻辑端口号。

端口是一个软件结构，被客户程序或服务程序用来发送和接收数据，一台服务器有 256\*256个端口。 端口号范围: 0 - 65535，0-1023是公认端口号，1024-65535是没有公共定义的动态端口号。端口与协议有关：TCP和UDP的端口互不相干。

3、公认端口

公认端口即众所周知的端口号，范围从0到1023，这些端口号一般固定分配给一些服务，比如：电话、饮食。公认端口如下:

FTP的端口是 21 SSH服务使用tcp 22 端口 TELNET 23 端口 DHCP server的端口号是67 MAIL 的端口号是25\110 pop3\smtp 的端口号是 110/25 DNS 的端口号是 53 HTTP通信用的端口号是80 mysql默认端口是3306 Sql服务的默认端口. 3389 tomcat默认端口是8080 windows远程终端 的端口号是3389

4、动态端口

动态端口的范围从1024到65535，这些端口号一般不固定分配给某个服务，也就是说许多服务都可以使用这些端口。只要运行的程序向系统提出访问网络的申请，那么系统就可以从这些端口号中分配一个供该程序使用。 比如1024端口就是分配给第一个向系统发出申请的程序。在关闭程序进程后，就会释放所占用的端口号。用户可以自己定义这些端口的作用。

### TCP与UDP

1、TCP

TCP是一种面向连接（连接导向）的、可靠的、基于IP的传输层协议。 弥补了IP协议的不足，属于一种较高级的协议，它实现了数据包的有力捆绑，通过排序和重传来确保数据传输的可靠(即数据的准确传输以及完整性)。 排序可以保证数据的读取是按照正确的格式进行，重传则保证了数据能够准确传送到目的地！

使用 TCP 协议通信是, 首先创建 TCP 连接, 主动发起连接的叫客户端, 被动响应连接的叫服务器。比如当我们在浏览器中访问新浪主页时，我们自己的计算机就是客户端，浏览器会主动向新浪的服务器发起连接。如果一切顺利，新浪的服务器接受了我们的连接，一个TCP连接就建立起来的，后面的通信就是发送网页内容了。

2、UDP

UDP也叫用户数据报协议，UDP编程相比TCP编程简单了很多。因为UDP不是面向连接的，而是面向无连接的。TCP是面向连接的，客户端和服务端必须连接之后才能通讯，就像打电话，必须先接通才能通话。

UDP是面向无连接的，一方负责发送数据(客户端)，只要知道对方(接受数据:服务器) 的地址就可以直接发数据了， 但是能不能达到就没有办法保证了。

3、TCP与UDP区别总结

* TCP：传送控制协议(Transmission Control Protocol)
* UDP：用户数据报协议 （UDP：User Datagram Protocol）
* 1、TCP面向连接 如打电话要先拨号建立连接）; UDP是无连接 的，即发送数据之前不需要建立连接
* 2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付Tcp通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。
* 3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。
* 4.每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信。
* 5、TCP对系统资源要求较多，UDP对系统资源要求较少。
* 虽然用UDP传输面向无连接, 数据不可靠，但它的优点是和TCP比，速度快，对于不要求可靠到达的数据，就可以使用UDP协议。 比如局域网的视频同步, 使用 udp 是比较合适的:快, 延迟越小越好

4、ICMP

ICMP与IP位于同一层，它被用来传送IP的控制信息。让我们能够检测网路的连线状况，也能确保连线的准确性。

ICMP协议是一种面向无连接的协议，用于传输出错报告控制信息。它对于网络安全具有极其重要的意义。

主要用于在主机与路由器之间传递控制信息，包括报告错误、交换受限控制和状态信息等。当遇到IP数据无法访问目标、IP路由器无法按当前的传输速率转发数据包等情况时，会自动发送ICMP消息。

我们在网络中经常会使用到ICMP协议，比如我们经常使用的用于检查网络通不通的Ping命令，这个“Ping”的过程实际上就是ICMP协议工作的过程。还有其他的网络命令如跟踪路由的Tracert命令也是基于ICMP协议的。

### Socket编程

socket编程主要是面向二层和三层的协议，就是针对IP协议、TCP、UDP的。

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，使主机间或者一台计算机上的进程间可以通讯。

Socket接口是TCP/IP网络的API，Socket接口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。

Socket接口设计者最先是将接口放在Unix操作系统里面的。如果了解Unix系统的输入和输出的话，就很容易了解Socket了。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。

可以把Socket理解成类似电话的东西, 通过Socket就可以发送和接受数据了, 就像你的电话可以作为服务端，别的很多人的电话作为客户端，他们任意的人都可以给你打电话，打通了你们之间就可以通话了。

应用层通过传输层进行数据通信时，TCP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要通过同一个 TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了套接字(Socket)接口。应用层可以和传输层通过Socket接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

TCP是面向连接的套接字, 而UDP是面向无连接的套接字。当我们知道如何通过网络端口访问到一个服务时，那么我们能够做什么呢？其实作为客户端来说，我们可以通过网络端口发送一个请求，然后得到服务器端反馈的信息。作为服务端，我们需要把我们的服务绑定到端口，并且在相应的端口上监听，当有客户端来访问时能够读取信息并且写入反馈信息。

## Go实现Socket

### 实现步骤

建立Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket ，另一个运行于服务器端，称为ServerSocket 。套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

Go中socket编程的API都在net包中Go提供了Dial函数来连接服务器，使用Listen监听，Accept接受连接。

客户端实现步骤：

1. 连接：net.Dial
2. 发送/接受：conn.Write()/conn.Read()

服务端实现步骤：

1. 监听：net.Listen()
2. 等待接受连接：listen.Accept()
3. 接受/发送：conn.Read()/conn.Write()

1. Conn

先来看看Go socket编程中最常用的类型是net.go文件里的Conn 。Conn是一个接口，后面我们会看到TCP和UDP分别实现了该接口。

**type** **Conn** interface { **Read**(b []byte) (n int, err error) **Write**(b []byte) (n int, err error) **Close**() error **LocalAddr**() **Addr** **RemoteAddr**() **Addr** //...... }

这个类型可以用来作为客户端和服务器端交互的通道，他有两个主要的函数：

**func** (c \*conn) **Read**(b []**byte**) (**int**, error) **func** (c \*conn) **Write**(b []**byte**) (**int**, error)

2. Dial

再看dial.go文件里的Dial函数：

**func** **Dial**(network, address **string**) (Conn, error)

该函数用于连接服务器。

* network：用于指定网络类型，目前支持的值有：tcp、tcp4(IPv4-only)、tcp6(IPv6-only)、udp、udp4(IPv4-only)、udp6(IPv6-only)、ip、ip4(IPv4-only)、ip6(IPv6-only)等。
* address：指定要连接的地址，对于TCP和UDP来说，地址格式为host:port。对于IPv6因为地址中已经有冒号了，所以需要用中括号将IP地址括起来，比如[::1]:80。如果省略掉host的话，比如:80，就认为是本地系统。

3. 现在来写一个简单的客户端

**func** **main**() { conn, err := net.Dial("tcp", ":5432") checkErr(err) fmt.Println("客户端已经连接成功。。。") fmt.Println("服务器的地址：",conn.RemoteAddr()) \_, err = conn.Write([]**byte**("你好")) checkErr(err) bs := make([] **byte**,512) n,err:=conn.Read(bs) checkErr(err) fmt.Println("客户端说：", **string**(bs[:n])) } **func** **checkErr**(err error) { **if** err != nil { panic(err) } }

4. Listen

**func** **Listen**(network, address **string**) (Listener, error)

Listen函数创建一个监听，

* net：网络类型必须是面向流的网络，目前可选值为tcp、tcp4、tcp6、unix、unixpacket。
* laddr：指定监听本地哪些网络接口，对于TCP和UDP，格式为host:port。如果省略host，表示监听所有本地地址。

5. Accept

接下来我们看Accept：

**type** Listener **interface** { Accept() (Conn, error) Close() error Addr() Addr }

Listener接口定义了普通的面向流的监听器，不同的goroutine可以并发的调用同一个监听器的方法。

6. 现在就可以写一个简单的服务器程序了

**func** **main**() { listen, err := net.Listen("tcp", ":5432") checkErr(err) fmt.Println(listen,err) fmt.Printf("%T\n",listen) *//\*net.TCPListener* **defer** listen.Close() conn, err := listen.Accept() fmt.Println("已有客户端连入。。",conn.RemoteAddr()) **if** err != nil { log.Fatal(err) } bs := make([] **byte**,512) n,err:=conn.Read(bs) fmt.Println("客户端说：", **string**(bs[:n])) conn.Write([] **byte**("我是服务器")) conn.Close() }

### 总结及注意事项

所以，掌握了Conn、Dial、Listen、Accept等API，我们就可以在简单的Go socket编程了。上面我们也看到，这些API一般都有一个参数用来指定具体的网络类型，比如是IP还是TCP，亦或是UDP、UNIX套接字等，也就是说它将多种不同的协议都整合到了同一些API之中，记忆和使用起来也比较方便。当然，针对每种具体的协议，也有特定的API，其实质也就是实现了诸如Conn等接口，当然通用的socket API也实现了这些接口(比如conn类型就是Conn接口的通用实现)，只不过只是实现了一些比较通用的方法等，但是特定协议的socket API会针对本协议多提供一些方法等。Go推荐使用通用的API进行socket编程，而不是具体某种协议的socket API，除非我们需要使用某些特定的方法。

下面我们分别对各种协议特定的socket API进行简单的介绍，主要目的是熟悉一下类型定义和API。

注意

1. conn.Read的行为特点

1）、Socket中无数据 连接建立后，如果对方未发送数据到socket，接收方(Server)会阻塞在Read操作上。执行该Read操作的goroutine也会被挂起。runtime会监视该socket，直到其有数据才会重新调度该socket对应的Goroutine完成read。

2）、Socket中有部分数据 如果socket中有部分数据，且长度小于一次Read操作所期望读出的数据长度，那么Read将会成功读出这部分数据并返回，而不是等待所有期望数据全部读取后再返回。

3）、Socket中有足够数据 如果socket中有数据，且长度大于等于一次Read操作所期望读出的数据长度，那么Read将会成功读出这部分数据并返回。这个情景是最符合我们对Read的期待的了：Read将用Socket中的数据将我们传入的slice填满后返回：n = 10, err = nil

4）、Socket关闭 如果client端主动关闭了socket，那么Server的Read将会读到什么呢？ 这里分为“有数据关闭”和“无数据关闭”。

有数据关闭是指在client关闭时，socket中还有server端未读取的数据。当client端close socket退出后，server依旧没有开始Read，10s后第一次Read成功读出了所有的数据，当第二次Read时，由于client端 socket关闭，Read返回EOF error

无数据关闭情形下的结果，那就是Read直接返回EOF error

5）、读取操作超时 有些场合对Read的阻塞时间有严格限制，在这种情况下，Read的行为到底是什么样的呢？在返回超时错误时，是否也同时Read了一部分数据了呢？ 不会出现“读出部分数据且返回超时错误”的情况

2、conn.Write的行为特点

1）、成功写 前面例子着重于Read，client端在Write时并未判断Write的返回值。所谓“成功写”指的就是Write调用返回的n与预期要写入的数据长度相等，且error = nil。这是我们在调用Write时遇到的最常见的情形，这里不再举例了

2）、写阻塞 TCP连接通信两端的OS都会为该连接保留数据缓冲，一端调用Write后，实际上数据是写入到协议栈的数据缓冲的。TCP是全双工通信，因此每个方向都有独立的数据缓冲。当发送方将对方的接收缓冲区以及自身的发送缓冲区写满后，Write就会阻塞

3）、写入部分数据 Write操作存在写入部分数据的情况。没有按照预期的写入所有数据。这时候循环写入便是

3. Goroutine safe

基于goroutine的网络架构模型，存在在不同goroutine间共享conn的情况，那么conn的读写是否是goroutine safe的呢？在深入这个问题之前，我们先从应用意义上来看read操作和write操作的goroutine-safe必要性。

对于read操作而言，由于TCP是面向字节流，conn.Read无法正确区分数据的业务边界，因此多个goroutine对同一个conn进行read的意义不大，goroutine读到不完整的业务包反倒是增加了业务处理的难度。对与Write操作而言，倒是有多个goroutine并发写的情况。

每次Write操作都是受lock保护，直到此次数据全部write完。因此在应用层面，要想保证多个goroutine在一个conn上write操作的Safe，需要一次write完整写入一个“业务包”；一旦将业务包的写入拆分为多次write，那就无法保证某个Goroutine的某“业务包”数据在conn发送的连续性。

同时也可以看出即便是Read操作，也是lock保护的。多个Goroutine对同一conn的并发读不会出现读出内容重叠的情况，但内容断点是依 runtime调度来随机确定的。存在一个业务包数据，1/3内容被goroutine-1读走，另外2/3被另外一个goroutine-2读 走的情况。比如一个完整包：world，当goroutine的read slice size < 5时，存在可能：一个goroutine读到 “worl”,另外一个goroutine读出”d”。

4. \t

若是标准输入那就是两个字符，若是字符串常量则是一个字符（转义字符）

## TCP Socket

### 实现socket从易到难

连接－>发送/接收－>返回－>输入发送/输入返回－>循环收发－>多人连接与服务器互动－>多人相互聊天

1、TCP编程

我们需要知道一个TCPAddr类型，他表示一个TCP的地址信息，他的定义如下：

**type** TCPAddr **struct** { IP IP Port **int** }

在Go语言中通过ResolveTCPAddr获取一个TCPAddr

**func** **ResolveTCPAddr**(net, addr **string**) (\*TCPAddr, os.Error)

* net参数是"tcp4"、"tcp6"、"tcp"中的任意一个，分别表示TCPv4、TCPv6或者任意
* addr表示域名或者IP地址，例如"[www.google.com:80](http://www.google.com:80)" 或者"127.0.0.1:22".

2、TCP client

Go语言中通过net包中的DialTCP函数建议一个TCP连接，返回一个TCPConn类型，客户端和服务器段通过这个类型来进行数据交换。一般而言，客户端通过TCPConn写入请求信息发送到服务器端，读取服务器端反馈的信息。这个链接只有当任意一边关闭了连接之后才失效，不然我们都可以一直使用。函数的定义如下：

**func** **DialTCP**(net **string**, laddr, raddr \*TCPAddr) (c \*TCPConn, err os.Error)

* net参数是"tcp4"、"tcp6"、"tcp"中的任意一个，分别表示TCPv4、TCPv6或者任意
* laddr表示本机地址，一般设置为nil
* raddr表示远程的服务地址

3、TCP server

创建服务器端程序，我们知道服务器端我们需要绑定服务到对应的端口，然后监听端口，当有客户端请求到达的时候接收客户端连接。net包中有相应的函数，函数定义如下：

**func** **ListenTCP**(net **string**, laddr \*TCPAddr) (l \*TCPListener, err os.Error) **func** (l \*TCPListener) **Accept**() (c Conn, err os.Error)

### 简单实现TCP socket

1、客户端

**func** **main**() { *//TCP的客户端程序* *//step1:提供要连接的服务端的地址* service:="192.168.1.147:54321" *//是服务器的地址，不是自己的，自己的端口由系统自动分配* tcpAddr,err:=net.ResolveTCPAddr("tcp4",service) fmt.Println(tcpAddr,err) *//10.0.154.238:54321 nil* *// step2：申请连接服务器* tcpConn, err:=net.DialTCP("tcp",nil,tcpAddr) fmt.Println(tcpConn,err) *//&{{0xc042088000}} <nil>* fmt.Printf("%T\n", tcpConn) *//\*net.TCPConn* fmt.Println("客户端已经连接成功。。。") fmt.Println("服务器的地址：",tcpConn.RemoteAddr()) *//step3：数据交互* n,err:=tcpConn.Write([] **byte**("hello"))*//写出的字节的数量* fmt.Println(n,err) fmt.Println("数据已经写出。。") bs := make([] **byte**,512) n,err=tcpConn.Read(bs) fmt.Println(**string**(bs[:n])) *//step4.关闭资源，断开连接* tcpConn.Close()*//* }

2、服务端

**func** **main**() { *//基于TCP的服务器* *//step1：确定本机的地址：ip:port--->TCPAddr烈性* service := ":54321" *// string* tcpAddr,err:=net.ResolveTCPAddr("tcp4",service) fmt.Println(tcpAddr,err) fmt.Printf("%T\n",tcpAddr) *//\*net.TCPAddr* *//step2：监听该端口* listener,err:=net.ListenTCP("tcp", tcpAddr) fmt.Println(listener,err) fmt.Printf("%T\n",listener) *//\*net.TCPListener* *//step3：接收客户端的连接请求* fmt.Println("服务器程序已经就绪，等待客户端的链接。。。") conn,err:=listener.Accept()*//阻塞式* fmt.Println(conn, err) *//&{{0xc042076000}} <nil>* fmt.Printf("%T\n", conn) *//\*net.TCPConn* fmt.Println("已有客户端连入。。",conn.RemoteAddr()) *//step4：数据交互* bs := make([] **byte**,512) n,err:=conn.Read(bs) fmt.Println(n,err) fmt.Println("客户端说：", **string**(bs[:n])) conn.Write([] **byte**("我是服务器")) *//step5：关闭资源* conn.Close() }

### 完善TCP socket

1、客户端

**func** **main**() { *//1.提供服务器地址* tcpAddr, \_ := net.ResolveTCPAddr("tcp4", "192.168.1.147:9527") *//2.连接* tcpConn, \_ := net.DialTCP("tcp", nil, tcpAddr) *//\*TCPConn* *//交互数据：* handleData2(tcpConn) } **func** **handleData2**(tcpConn \*net.TCPConn){ *//* *//3.读写* **for** { *//A.读取键盘* line := "" fmt.Scanln(&line) *//键盘输入，阻塞的* fmt.Println("--",line,"--") *//B.写给服务器* n,\_:=tcpConn.Write([] **byte**(line)) fmt.Println("客户端写出数据：",n) **if** line == "over"{ fmt.Println("客户端即将结束。。") **break** } *//F:读取服务器* bs := make([] **byte**, 512) n,\_= tcpConn.Read(bs) *//阻塞式* fmt.Println("服务器说：", **string**(bs[:n])) } tcpConn.Close() }

2、服务端

**func** **main**() { *//1.提供本机的地址* tcpAddr, \_ := net.ResolveTCPAddr("tcp4", ":9527") *//2.提供监听对象* listener, \_ := net.ListenTCP("tcp", tcpAddr) fmt.Println("服务器端已经就绪，等待客户端链接。。。") **for**{ *//3.等待客户端链接* conn, \_ := listener.Accept() fmt.Println("已有客户端连入。。", conn.RemoteAddr()) *//4.读写数据* *//C：服务器读取客户端* **go** handleData(conn) } } **func** **handleData**(conn net.Conn){ **for** { bs := make([] **byte**, 512) n, \_ := conn.Read(bs) *//阻塞的* content:=**string**(bs[:n]) **if** content=="over" ||n == 0{ fmt.Println("客户端已经跑了。。") **break** } fmt.Println("客户端说：", content, n) *//E：写给客户端* conn.Write([] **byte**("ECHO:"+content)) } conn.Close() }

## UDP Socket

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)

### 客户端

**func** **main**() { conn, err := net.Dial("udp", "127.0.0.1:1200") **defer** conn.Close() **if** err != nil { os.Exit(1) } conn.Write([]**byte**("Hello world!")) fmt.Println("send msg") msg := make([] **byte**,512) n,err :=conn.Read(msg) fmt.Println("msg is", **string**(msg[:n])) }

### 服务端

**func** **checkError**(err error){ **if** err != nil { fmt.Println("Error: %s", err.Error()) os.Exit(1) } } **func** **recvUDPMsg**(conn \*net.UDPConn){ **var** buf [512]**byte** n, raddr, err := conn.ReadFromUDP(buf[0:]) **if** err != nil { **return** } fmt.Println("msg is ", **string**(buf[0:n])) *//WriteToUDP* \_, err = conn.WriteToUDP([]**byte**("nice to see u"), raddr) checkError(err) } **func** **main**() { udp\_addr, err := net.ResolveUDPAddr("udp", ":1200") checkError(err) conn, err := net.ListenUDP("udp", udp\_addr) **defer** conn.Close() checkError(err) *//go recvUDPMsg(conn)* recvUDPMsg(conn) }

## ICMP编程

*//运行这个程序需要root权限。* **func** **main**() { **if** len(os.Args) != 2 { fmt.Println("Usage: ", os.Args[0], "host") os.Exit(1) } addr, err := net.ResolveIPAddr("ip", os.Args[1]) **if** err != nil { fmt.Println("Resolution error", err.Error()) os.Exit(1) } conn, err := net.DialIP("ip4:icmp", addr, addr) checkErr(err) **var** msg [512]**byte** msg[0] = 8 *// echo* msg[1] = 0 *// code 0* msg[2] = 0 *// checksum* msg[3] = 0 *// checksum* msg[4] = 0 *// identifier[0]* msg[5] = 13 *// identifier[1]* msg[6] = 0 *// sequence[0]* msg[7] = 37 *// sequence[1]* len := 8 check := checkSum(msg[0:len]) msg[2] = **byte**(check >> 8) msg[3] = **byte**(check & 255) \_, err = conn.Write(msg[0:len]) checkErr(err) \_, err = conn.Read(msg[0:]) checkErr(err) fmt.Println("Got response") **if** msg[5] == 13 { fmt.Println("identifier matches") } **if** msg[7] == 37 { fmt.Println("Sequence matches") } os.Exit(0) } **func** **checkSum**(msg []**byte**) **uint16** { sum := 0 *// assume even for now* **for** n := 1; n < len(msg) - 1; n += 2 { sum += **int**(msg[n])\*256 + **int**(msg[n+1]) } sum = (sum >> 16) + (sum & 0xffff) sum += (sum >> 16) **var** answer **uint16** = **uint16**(^sum) **return** answer } **func** **checkErr**(err error) { **if** err != nil { panic(err) } }

## Http概述

### 概念

HTTP是一个基于TCP/IP通信协议来传递数据，服务器传输超文本到本地浏览器的传送协议。HTTP协议工作于客户端-服务端架构上。浏览器可作为HTTP客户端通过URL向HTTP服务端即WEB服务器发送所有请求。Web服务器根据接收到的请求后，向客户端发送响应信息。它是一个无状态的请求/响应协议。

客户端请求消息和服务器响应消息都会包含请求头和请求体。HTTP请求头提供了关于请求或响应，发送实体的信息，如：Content-Type、Content-Length、Date等。当浏览器接收并显示网页前，此网页所在的服务器会返回一个包含HTTP状态码的信息头（server header）用以响应浏览器的请求。

HTTP状态码的英文为HTTP Status Code。

下面是常见的HTTP状态码：

* 200 - 请求成功
* 301 - 资源（网页等）被永久转移到其它URL
* 404 - 请求的资源（网页等）不存在
* 413 - 由于请求的实体过大，服务器无法处理，因此拒绝请求。为防止客户端的连续请求，服务器可能会关闭连接。
* 500 - 内部服务器错误

### HTTP请求方法

常用方法: Get\Post\Head

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是\*\*GET，POST，PUT，DELETE，\*\*对应着对这个资源的查 ，改 ，增 ，删 4个操作。

另外还有个Head方法. 类似GET方法，只请求页面的首部，不响应页面Body部分，用于获取资源的基本信息，即检查链接的可访问性及资源是否修改。

GET和POST的区别

* GET在浏览器回退时是无害的，而POST会再次提交请求。
* GET产生的URL地址可以被Bookmark，而POST不可以。
* GET请求会被浏览器主动cache，而POST不会，除非手动设置。
* GET请求只能进行url编码，而POST支持多种编码方式。
* GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
* GET请求在URL中传送的参数是有长度限制的，而POST没有。
* 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
* GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
* GET参数通过URL传递，POST放在Request body中。

HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。

* GET产生一个TCP数据包；POST产生两个TCP数据包。
* 对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据））

### HTTPS通信原理

HTTPS（Secure Hypertext Transfer Protocol）安全超文本传输协议 它是一个安全通信通道

HTTPS是HTTP over SSL/TLS，HTTP是应用层协议，TCP是传输层协议，在应用层和传输层之间，增加了一个安全套接层SSL。

服务器 用RSA生成公钥和私钥把公钥放在证书里发送给客户端，私钥自己保存客户端首先向一个权威的服务器检查证书的合法性，如果证书合法，客户端产生一段随机数，这个随机数就作为通信的密钥，我们称之为对称密钥，用公钥加密这段随机数，然后发送到服务器服务器用密钥解密获取对称密钥，然后，双方就已对称密钥进行加密解密通信了。

### Https的作用

* 内容加密 建立一个信息安全通道，来保证数据传输的安全；
* 身份认证 确认网站的真实性
* 数据完整性 防止内容被第三方冒充或者篡改

Https和Http的区别

* https协议需要到CA申请证书。
* http是超文本传输协议，信息是明文传输；https 则是具有安全性的ssl加密传输协议。
* http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
* http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## http协议客户端实现

go语言中提供了net/http库进行网络请求。

### 客户端代码实现

1、一个简单的get实现

func main() { requestUrl := "http://www.baidu.com" response, err := http.Get(requestUrl) if err != nil { fmt.Println("err:",err); } defer response.Body.Close(); fmt.Println(response.Body) }

err: Get www.baidu.com: unsupported protocol scheme ""

错误：不支持scheme为空的协议，这是什么情况呢，

requestUrl改为requestUrl := "http://www.baidu.com"

scheme反应出的就是协议名称

大家可以看到打印的是 &{0xc420124200 <nil> <nil>}

Body是io.ReadCloser这个类型的对象

response中的参数一般也比较多，我们需要的最多的通常是Body参数，用ioutil.ReadAll去转化io.ReadCloser类型，输出是byty[]，再通过string()强制转换就能看到string了

readBody, \_ := ioutil.ReadAll(response.Body) fmt.Println(string(readBody))

咱们这个是正确获取到了服务器响应的数据才能这么打印出来了，加入失败了呢，因此需要判断返回码，再去获取body的数据。

if response.StatusCode == 200 { r, err := ioutil.ReadAll(response.Body) if err != nil { fmt.Println(err) } return string(r) }

2、方式一 使用http.NewRequest

先生成http.client -> 再生成 http.request -> 之后提交请求：[client.Do](http://client.Do)(request) -> 处理返回结果，每一步的过程都可以设置一些具体的参数

3、方式二 先生成client，之后用client.get/post..

client结构自己也有一些发送api的方法，比如client.get,client.post,client.postform..等等。基本上涵盖了主要的http请求的类型，通常不进行什么特殊的配置的话，这样就可以了，其实client的get或者post方法，也是对http.Newerequest方法的封装，里面还额外添加了req.Header.Set("Content-Type", bodyType)一般用的话，也是ok的

4、方式三 http. Get/Post..

具体实现的时候，还是采用的方式一模式，先生成一个默认的client，之后调用http.Newrequest方法。

http.NewRequest

client := &http.Client{} request, err := http.NewRequest("GET", requestUrl, nil) if err != nil { fmt.Println(err) } cookie := &http.Cookie{Name: "userId", Value: strconv.Itoa(12345)} request.AddCookie(cookie) //request中添加cookie //设置request的header request.Header.Set("Content-Type", "text/html") //Content-Length等 response, err := client.Do(request)

5、Post

postvalue := url.Values{ "theCityName": {"天津"}, } body := bytes.NewBufferString(postvalue.Encode()) response, err := http.Post(requestUrl, "application/x-www-form-urlencoded", body) //Post方法

## http协议服务端实现

### 服务端代码实现

func ServeHTTP(w http.ResponseWriter, r \*http.Request) { r.ParseForm() fmt.Println("PATH: ", r.URL.Path) fmt.Println("HOST: ", r.URL.Host) fmt.Println("METHOD: ", r.Method) fmt.Println() //fmt.Fprintf(w, "<h1>Index Page</h1>") io.WriteString(w, "<h1>Index Page3</h1>") } func main() { http.HandleFunc("/", ServeHTTP) err := http.ListenAndServe(":9000", nil) if err != nil { log.Fatal("ERROR: ", err) } }

函数func ListenAndServe(addr string, handler Handler) error内部进行封装更简洁地实现HTTP服务器。你可以看到它创建了Server类的一个对象，然后调用了刚才说的第一个函数。第2个参数是一个Hander，它是一个接口。这个接口很简单，只要某个struct有ServeHTTP(http.ResponseWriter, \*http.Request)这个方法，那这个struct就自动实现了Hander接口。

ServeHTTP方法，他需要2个参数，一个是http.ResponseWriter，另一个是http.Request往http.ResponseWriter写入什么内容，浏览器的网页源码就是什么内容。http.Request里面是封装了，浏览器发过来的请求（包含路径、浏览器类型等等）。

OK，作为服务器我们会处理很多的请求，那下一步如何处理呢，难道switch r.URL.Path的值吗？那得多辛苦。那有什么好的办法呢，这点go官方已经考虑到这点帮我们提供了一个方法叫做ServeMux，去分发任务。

下面介绍下ServeMux

ServeMux大致作用是，他有一张map表，map里的key记录的是r.URL.String()，而value记录的是一个方法，这个方法和ServeHTTP是一样的，这样ServeMux是实现Handler接口的。这个方法有一个别名，叫HandlerFunc。

func main() { mux := http.NewServeMux() mux.HandleFunc("/h", func(w http.ResponseWriter, r \*http.Request) { io.WriteString(w, "hello") }) mux.HandleFunc("/bye", func(w http.ResponseWriter, r \*http.Request) { io.WriteString(w, "byebye") }) mux.HandleFunc("/baidu", func(w http.ResponseWriter, r \*http.Request) { http.Redirect(w, r, "http://www.baidu.com", http.StatusTemporaryRedirect) }) mux.HandleFunc("/hello", sayhello) http.ListenAndServe(":9000", mux) } func sayhello(w http.ResponseWriter, r \*http.Request) { io.WriteString(w, "hello world") }

回到开头，有让大家先忘掉http.HandleFunc("/", HandleIndex) 当http.ListenAndServe(":9000", nil)的第2个参数是nil时，http内部会自己建立一个叫DefaultServeMux的ServeMux，因为这个ServeMux是http自己维护的，如果要向这个ServeMux注册的话，就要用http.HandleFunc这个方法啦，现在看很简单吧。

func FileServer

func FileServer(root FileSystem) Handler

FileServer返回一个使用FileSystem接口root提供文件访问服务的HTTP处理器。要使用操作系统的FileSystem接口实现，可使用http.Dir：

log.Fatal(http.ListenAndServe(":8080", http.FileServer(http.Dir("/usr/share/doc"))))

### 服务端获取客户端请求数据

1. 获取GET参数

比较常见的是如下方式获取：

r.ParseForm() if len(r.Form["id"]) > 0 { fmt.Fprintln(w, r.Form["id"][0]) }

其中r表示\*http.Request类型，w表示http.ResponseWriter类型。

r.Form是url.Values字典类型，r.Form["id"]取到的是一个数组类型。因为http.request在解析参数的时候会将同名的参数都放进同一个数组里，所以这里要用[0]获取到第一个。

2. 获取POST参数

这里要分两种情况：

普通的post表单请求，Content-Type=application/x-www-form-urlencoded 有文件上传的表单，Content-Type=multipart/form-data

第一种情况比较简单，直接用PostFormValue就可以取到了。

fmt.Fprintln(w, r.PostFormValue("id"))

第二种文件上传的表单这里不作讲解。

3. 获取COOKIE参数

cookie, err := r.Cookie("id") if err == nil { fmt.Fprintln(w, "Domain:", cookie.Domain) fmt.Fprintln(w, "Expires:", cookie.Expires) fmt.Fprintln(w, "Name:", cookie.Name) fmt.Fprintln(w, "Value:", cookie.Value) }

r.Cookie返回\*http.Cookie类型，可以获取到domain、过期时间、值等数据。

注意

application/x-www-form-urlencoded与 text/html 的区别

若需使用r.PostForm、r.Form等方法，必须先调用r.ParseForm()

客户端使用post提交数据到body时，若服务端需使用r.PostForm、r.Form等方法，客户端的请求体必须使用"Content-Type":"application/x-www-form-urlencoded") 类型，并且r.ParseForm()须放在ioutil.ReadAll(r.Body)解析body之前。否则数据将在body中。

## RPC概述

### 什么是RPC

远程过程调用（Remote Procedure Call，缩写为 RPC）是一个计算机通信协议。 该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互编程。远程过程调用是构建分布式应用的理论基础，它简单而又广受欢迎。 远程过程调用总是由客户端对服务器发出一个执行若干过程请求，并用客户端提供的参数。执行结果将返回给客户端。

一个通俗的描述是：客户端在不知道调用细节的情况下，调用存在于远程计算机上的某个对象，就像调用本地应用程序中的对象一样。

那么我们至少从这样的描述中挖掘出几个要点：

* RPC是协议：既然是协议就只是一套规范，那么就需要有人遵循这套规范来进行实现。目前典型的RPC实现包括：Dubbo(Alibaba的分布式服务框架)、Thrift(Facebook)、GRPC(Google)、Hetty等。这里要说明一下，目前技术的发展趋势来看，实现了RPC协议的应用工具往往都会附加其他重要功能，例如Dubbo还包括了服务管理、访问权限管理等功能。
* 网络协议对其透明：既然RPC的客户端认为自己是在调用本地对象。那么传输层使用的是TCP/UDP还是HTTP协议，又或者是一些其他的网络协议它就不需要关心了。
* 信息格式对其透明：我们知道在本地应用程序中，对于某个对象的调用需要传递一些参数，并且会返回一个调用结果。至于被调用的对象内部是如何使用这些参数，并计算出处理结果的，调用方是不需要关心的。那么对于远程调用来说，这些参数会以某种信息格式传递给网络上的另外一台计算机，这个信息格式是怎样构成的，调用方是不需要关心的。
* 应该有跨语言能力：为什么这样说呢？因为调用方实际上也不清楚远程服务器的应用程序是使用什么语言运行的。那么对于调用方来说，无论服务器方使用的是什么语言，本次调用都应该成功，并且返回值也应该按照调用方程序语言所能理解的形式进行描述。

### golang中如何实现RPC

在golang中实现RPC非常简单，有封装好的官方库和一些第三方库提供支持。Go RPC可以利用tcp或http来传递数据，可以对要传递的数据使用多种类型的编解码方式。golang官方的net/rpc库使用encoding/gob进行编解码，支持tcp或http数据传输方式，由于其他语言不支持gob编解码方式，所以使用net/rpc库实现的RPC方法没办法进行跨语言调用。

golang官方还提供了net/rpc/jsonrpc库实现RPC方法，JSON RPC采用JSON进行数据编解码，因而支持跨语言调用。但目前的jsonrpc库是基于tcp协议实现的，暂时不支持使用http进行数据传输。

除了golang官方提供的rpc库，还有许多第三方库为在golang中实现RPC提供支持，如：protorpc库，大部分第三方rpc库的实现都是使用protobuf进行数据编解码，根据protobuf声明文件自动生成rpc方法定义与服务注册代码，在golang中可以很方便的进行rpc服务调用。

### net/rpc 包实现

先介绍Go官方提供的 net/rpc包，它提供了通过网络访问一个对象的方法的能力。服务器需要注册对象， 通过对象的类型名暴露这个服务。注册后这个对象的输出方法就可以远程调用，这个库封装了底层传输的细节，包括序列化。服务器可以注册多个不同类型的对象，但是注册相同类型的多个对象的时候会出错。

如果对象的方法要能远程访问，它们必须满足一定的条件，否则这个对象的方法会被忽略。

这些条件是：

* 方法的类型是可输出的
* 方法必须由两个参数
* 方法的第二个参数是指针类型
* 方法返回类型为 error

所以一个输出方法的格式如下：

func (t \*T) MethodName(argType T1, replyType \*T2) error

这里的T、T1、T2能够被encoding/gob序列化。

这个方法的第一个参数代表调用者(client)提供的参数，第二个参数代表要返回给调用者的计算结果，方法的返回值如果不为空， 那么它作为一个字符串返回给调用者。

如果返回error，则reply参数不会返回给调用者。

服务器通过调用ServeConn在一个连接上处理请求，更典型地， 它可以创建一个network listener然后accept请求。对于HTTP listener来说，可以调用 HandleHTTP 和 http.Serve。

客户端可以调用Dial和DialHTTP建立连接。 客户端有两个方法调用服务: Call 和 Go，可以同步地或者异步地调用服务。当然，调用的时候，需要把服务名、方法名和参数传递给服务器。异步方法调用Go通过 Done channel通知调用结果返回。

除非显示的设置codec,否则这个库默认使用包encoding/gob作为序列化框架。

## RPC库

首选介绍一个简单的例子。 这个例子中提供了面积和周长的两个方法。

### 服务端

第一步你需要定义传入参数和返回参数的数据结构：

**type** **Params** struct { **Width**, **Height** int }

第二步定义一个服务对象，这个服务对象可以很简单， 比如类型是int或者是interface{},重要的是它输出的方法。 这里我们定义一个算术类型Rect，它可以是任意类型，也可以是int类型，但是这个值我们在后面方法的实现中也没用到，所以它基本上就起一个辅助的作用。

**type** **Rect** **struct**{}

第三步实现这个类型的两个方法， 面积和周长：

**func** (r \*Rect) **Area**(p Params, ret \***int**) **error** { \*ret = p.Width \* p.Height **return** nil } **func** (r \*Rect) **Perimeter**(p Params, ret \***int**) **error** { \*ret = (p.Width + p.Height) \* 2 **return** nil }

目前为止，我们的准备工作已经完成，喝口茶继续下面的步骤。

第四步实现RPC服务器:

rect := new(Rect) *//注册一个rect服务* rpc.Register(rect) *//把服务处理绑定到http协议上* rpc.HandleHTTP() err := http.ListenAndServe(":8080", nil) **if** err != nil { log.Fatal(err) }

这里我们生成了一个rect对象，并使用rpc.Register注册这个服务，然后通过HTTP暴露出来。 客户端可以看到服务rect以及它的两个方法Rect.Area和Rect.Perimeter。

### 客户端

最后创建一个客户端，建立客户端和服务器端的连接:

rpc, **err** := rpc.DialHTTP("tcp", "127.0.0.1:8080") **if** **err** != nil { **log**.Fatal(**err**) }

然后客户端就可以进行远程调用了。

ret := 0 *//调用远程方法* *//注意第三个参数是指针类型* err2 := rpc.Call("Rect.Area", Params{50, 100}, &ret) **if** err2 != nil { log.Fatal(err2) } fmt.Println(ret) err3 := rpc.Call("Rect.Perimeter", Params{50, 100}, &ret) **if** err3 != nil { log.Fatal(err3) } fmt.Println(ret)

或者异步的方式：

divCall := client.Go("Arith.Divide", Params{50, 100}, &**ret**, nil) <-divCall.Done fmt.Println(**ret**)

## net/rpc/jsonrpc库

上面的例子我们演示了使用net/rpc实现RPC的过程，但是没办法在其他语言中调用上面例子实现的RPC方法。所以接下来的例子我们演示一下使用net/rpc/jsonrpc库实现RPC方法，此方式实现的RPC方法支持跨语言调用。

### 服务端

*// 算数运算结构体* **type** Arith **struct** { } *// 算数运算请求结构体* **type** ArithRequest **struct** { A **int** B **int** } *// 算数运算响应结构体* **type** ArithResponse **struct** { Pro **int** *// 乘积* Quo **int** *// 商* Rem **int** *// 余数* } *// 乘法运算方法* **func** (this \*Arith) **Multiply**(req ArithRequest, res \*ArithResponse) **error** { res.Pro = req.A \* req.B **return** nil } *// 除法运算方法* **func** (this \*Arith) **Divide**(req ArithRequest, res \*ArithResponse) **error** { **if** req.B == 0 { **return** errors.New("divide by zero") } res.Quo = req.A / req.B res.Rem = req.A % req.B **return** nil } **func** **main**() { rpc.Register(new(Arith)) *// 注册rpc服务* lis, err := net.Listen("tcp", ":8096") **if** err != nil { log.Fatalln("fatal error: ", err) } fmt.Fprintf(os.Stdout, "%s", "start connection") **for** { conn, err := lis.Accept() *// 接收客户端连接请求* **if** err != nil { **continue** } **go** **func**(conn net.Conn) { *// 并发处理客户端请求* fmt.Fprintf(os.Stdout, "%s", "new client in coming\n") jsonrpc.ServeConn(conn) }(conn) } }

### 客户端

*// 算数运算请求结构体* **type** ArithRequest **struct** { A **int** B **int** } *// 算数运算响应结构体* **type** ArithResponse **struct** { Pro **int** *// 乘积* Quo **int** *// 商* Rem **int** *// 余数* } **func** **main**() { conn, err := jsonrpc.Dial("tcp", "127.0.0.1:8096") **if** err != nil { log.Fatalln("dailing error: ", err) } req := ArithRequest{9, 2} **var** res ArithResponse err = conn.Call("Arith.Multiply", req, &res) *// 乘法运算* **if** err != nil { log.Fatalln("arith error: ", err) } fmt.Printf("%d \* %d = %d\n", req.A, req.B, res.Pro) err = conn.Call("Arith.Divide", req, &res) **if** err != nil { log.Fatalln("arith error: ", err) } fmt.Printf("%d / %d, quo is %d, rem is %d\n", req.A, req.B, res.Quo, res.Rem) }

### protorpc库

为了实现跨语言调用，在golang中实现RPC方法的时候我们应该选择一种跨语言的数据编解码方式，比如JSON，上述的jsonrpc可以满足此要求，但是也存在一些缺点，比如不支持http传输，数据编解码性能不高等。于是呢，一些第三方rpc库都选择采用protobuf进行数据编解码，并提供一些服务注册代码自动生成功能。我们使用protobuf来定义RPC方法及其请求响应参数，并使用第三方的protorpc库来生成RPC服务注册代码。

PrProtobuf， 是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。它很适合做数据存储或 RPC 数据交换格式。可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。

需要安装protobuf及protoc

### 有了HTTP为何还有RPC？

论复杂度，RPC框架肯定是高于简单的HTTP接口的。但毋庸置疑，HTTP接口由于受限于HTTP协议，需要带HTTP请求头，导致传输起来效率或者说安全性不如RPC。

下面展示了一个请求头，无用数据过多，响应头类似

GET /search/suggest/initial\_page/ HTTP/1.1 Host www.toutiao.com Content-Type application/x-www-form-urlencoded Accept-Encoding gzip, deflate Cookie CNZZDATA1259612802=569328305-1527816505-https%253A%252F%252Fwww.baidu.com%252F%7C1527821905; WEATHER\_CITY=%E5%8C%97%E4%BA%AC; \_\_tasessionId=i33ntavtt1527822487203; tt\_webid=6561930712081466884; UM\_distinctid=163b91509b88ff-0e1ee05ec87bae-3f636c4f-13c680-163b91509b96aa; \_ga=GA1.2.195827442.1493809988; sso\_login\_status=0; tt\_webid=6561930712081466884; uuid="w:b5453e80f63342d1afe07a5d3c3360f9" Connection keep-alive Proxy-Connection keep-alive Accept text/javascript, text/html, application/xml, text/xml, \**/\* User-Agent Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_6) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.1 Safari/605.1.15 Referer https://www.toutiao.com/ Accept-Language zh-cn X-Requested-With XMLHttpRequest*

http接口是在接口不多、系统与系统交互较少的情况下；优点就是简单、直接、开发方便。利用现成的http协议进行传输。但是如果是一个大型的网站，内部子系统较多、接口非常多的情况下，RPC框架的好处就显示出来了，首先就是长链接，不必每次通信都要像http 一样去3次握手什么的，减少了网络开销；其次就是RPC框架一般都有注册中心，有丰富的监控管理；发布、下线接口、动态扩展等，对调用方来说是无感知、统 一化的操作。

HTTP与RPC存在重大不同的是：请求是使用具有标准语义的通用的接口定向到资源的，这些语义能够被中间组件和提供服务的来源机器进行解释。结果是使得一个应用支持分层的转换和间接层，并且独立于消息的来源。

与之相比较，RPC的机制是根据语言的API来定义的，而不是根据基于网络的应用来定义的。

事实上，对于http,也可以作为RPC框架的通信层协议和实现。 只不过，对于大部分企业成熟的RPC框架，使用thrift等工具可以实现二进制传输，相比HTTP的文本传输无疑大大提高了传输效率; HTTP通常使用的json，一个需要用户序列化/反序列化，性能和复杂度较高。相比之下，Thrift等工具，使用了成熟的代码生成技术，将通信接口的文件生成了对应语言的代码接口，实现了远程调用接近于本地方法的调用。另外无论是网络传输编码、解码，还是传输内容大小还是网络开销都想比HTTP有较大的优势。

一般rpc框架包括：服务查找，负载均衡，服务降级、熔断，下游路由配置，数据格式约定，链接维护等几个方面。

## TCP、HTTP与Socket区别

### TCP连接

手机能够使用联网功能是因为手机底层实现了TCP/IP协议，可以使手机终端通过无线网络建立TCP连接。TCP协议可以对上层网络提供接口，使上层网络数据的传输建立在“无差别”的网络之上。

建立起一个TCP连接需要经过“三次握手”：

第一次握手：客户端发送syn包(syn=j)到服务器，并进入SYN\_SEND状态，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RECV状态；

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连 接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“四次握手”（过程就不细写 了，就是服务器和客户端交互，最终确定断开）

### HTTP连接

HTTP协议即超文本传送协议(Hypertext Transfer Protocol )，是Web联网的基础，也是手机联网常用的协议之一，HTTP协议是建立在TCP协议之上的一种应用。

\*\*http 为短连接：\*\*客户端发送请求都需要服务器端回送响应。请求结束后，主动释放链接，因此为短连接。要保持客户端程序的在线状态，需要不断地向服务器发起连接请求。通常的 做法是即时不需要获得任何数据，客户端也保持每隔一段固定的时间向服务器发送一次“保持连接”的请求，服务器在收到该请求后对客户端进行回复，表明知道客 户端“在线”。若服务器长时间无法收到客户端的请求，则认为客户端“下线”，若客户端长时间无法收到服务器的回复，则认为网络已经断开。

TCP是底层通讯协议，定义的是数据传输和连接方式的规范 HTTP是应用层协议，定义的是传输数据的内容的规范。 http与tcp不是对等的两种协议，http是可以基于tcp传输的协议。

### Socket连接

\*\*Socket为长连接：\*\*Socket 是对 TCP/IP 协议的封装，Socket 只是个接口不是协议，通过 Socket 我们去使用 TCP/IP 协议，除了 TCP，也可以使用 UDP 协议来传递数据。

Socket也称作套接字，套接字是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示。建立Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket ，另一个运行于服务器端，称为ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。

客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户 端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

### 总结

HTTP连接使用的是"请求-响应"方式，不仅在请求时才建立连接，而且客户端向服务器端请求后，服务器才返回数据。

创建 Socket 连接的时候，可以指定传输层协议，可以是 TCP 或者 UDP。通常情况下Socket 连接就是 TCP 连接，因此 Socket 连接一旦建立,通讯双方开始互发数据内容，直到双方断开连接。

很多情况下，都是需要服务器端向客户端主动推送数据，保持客户端与服务端的实时同步。

若双方是 Socket 连接，可以由服务器直接向客户端发送数据。

若双方是 HTTP 连接，则服务器需要等客户端发送请求后，才能将数据回传给客户端。

因此，客户端定时向服务器端发送请求，不仅可以保持在线，同时也询问服务器是否有新数据，如果有就将数据传给客户端。

## JSON概述

### JSON简介

JSON是一种轻量级的数据交换格式(JavaScript Object Notation)，因其简单、可读性强广泛使用。

1、JSON语法规则

* 对象是一个无序的“‘名称/值’对”集合。一个对象以“{”（左括号）开始，“}”（右括号）结束。每个“名称”后跟一个“:”（冒号）；“‘名称/值’ 对”之间使用“,”（逗号）分隔。
* 数组是值（value）的有序集合。一个数组以“[”（左中括号）开始，“]”（右中括号）结束。值之间使用“,”（逗号）分隔。
* 值（value）可以是双引号括起来的字符串（string）、数值(number)、true、false、 null、对象（object）或者数组（array）。这些结构可以嵌套。
* 字符串（string）是由双引号包围的任意数量Unicode字符的集合，使用反斜线转义。一个字符（character）即一个单独的字符串（character string）。与C或者Java的字符串非常相似。
  + 比较下面两种数据的区别

"{\"**class**\":\"1803班\", \"**Count**\":\"35\"}" {"class":"1803班", "Count":"35"}

2、优点:

1. 数据格式比较简单, 易于读写, 格式都是压缩的, 占用带宽小
2. 易于解析这种语言, 客户端JavaScript可以简单的通过eval\_r()进行JSON数据的读取
3. 支持多种语言, 包括ActionScript, C, C#, ColdFusion, Java,JavaScript, Perl, PHP, Python, Ruby等语言服务器端语言, 便于服务器端的解析
4. 在PHP世界, 已经有PHP-JSON和JSON-PHP出现了, 便于PHP序列化后的程序直接调用. PHP服务器端的对象、数组等能够直接生JSON格式, 便于客户端的访问提取.
5. 因为JSON格式能够直接为服务器端代码使用, 大大简化了服务器端和客户端的代码开发量, 但是完成的任务不变, 且易于维护

3、缺点:

1. 没有XML格式这么推广的深入人心和使用广泛, 没有XML那么通用性
2. JSON格式目前在Web Service中推广还属于初级阶段

### XML

1、XML的概念

扩展标记语言 (Extensible Markup Language,XML) ，用于标记电子文件使其具有结构性的标记语言，可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。

XML一种类似于HTML的语言，他没有预先定义的标签。非常适合 Web 传输。

XML相比之JSON这种轻量级的数据交换格式，XML可以称为重量级的了。

2、优点：

1. 格式统一, 符合标准
2. 容易与其他系统进行远程交互, 数据共享比较方便

3、缺点:

1. XML文件格式文件庞大, 格式复杂, 传输占用带宽
2. 服务器端和客户端都需要花费大量代码来解析XML, 不论服务器端和客户端代码变的异常复杂和不容易维护
3. 客户端不同浏览器之间解析XML的方式不一致, 需要重复编写很多代码
4. 服务器端和客户端解析XML花费资源和时间

### 比较

假如有一个employee对象，它有“姓、名、员工编号、头衔”等信息，先看看JSON是如何来描述它的：

{ "employee" : { "firstName": "王", "lastName" : "五", "employeeNumber" : 123, "title" : "客户经理" } }

再来看看XML是如何表示的

<employee> <firstName>王</firstName> <lastName>五</lastName> <employeeNumber>123</employeeNumber> <title>客户经理</title> </employee>

1、 轻量级和重量级

那么XML相对于JSON的重量级体现在哪呢?我想应该体现在解析上，XML目前设计了两种解析方式：DOM和 SAX;

DOM是把一个数据交换格式XML看成一个DOM对象，需要把XML文件整个读入内存，这一点上JSON和XML的原理是一样的，但是XML要考虑父节点和子节点，这一点上JSON的解析难度要小很多，因为JSON构建于两种结构：key/value，键值对的集合;值的有序集合，可理解为数组;

SAX不需要整个读入文档就可以对解析出的内容进行处理，是一种逐步解析的方法。程序也可以随时终止解析。这样，一个大的文档就可以逐步的、一点一点的展现出来，所以SAX适合于大规模的解析。这一点，JSON目前是做不到得。

所以，JSON和XML的轻/重量级的区别在于：JSON只提供整体解析方案，而这种方法只在解析较少的数据时才能起到良好的效果;而XML提 供了对大规模数据的逐步解析方案，这种方案很适合于对大量数据的处理。

2. 编码及解析的难度

编码：它们都是基于文本的，且它们都使用Unicode编码。

解析：在普通的web应用领域，开发者经常为XML的解析伤脑筋，无论是服务器端生成或处理XML，还是客户端用 JavaScript解析XML，都常常导致复杂的代码，极低的开发效率。

主观上来看，JSON更为清晰且冗余更少些。从总体来看，XML比较适合于标记文档，而JSON却更适于进行数据交换处理。

实际上，对于大多数web应用来说，他们根本不需要复杂的XML来传输数据，XML宣称的扩展性在此就很少具有优势；许多Ajax应用甚至直接返回HTML片段来构建动态web页面。和返回XML并解析它相比，返回HTML片段大大降低了系统的复杂性，但同时缺少了一定的灵活性。同XML或 HTML片段相比，数据交换格式JSON 提供了更好的简单性和灵活性。在web serivice应用中，至少就目前来说XML仍有不可动摇的地位。

## JSON编码

Go的标准包encoding/json对JSON的编解码提供了完整的支持。

编码，即将Go数据类型转换为JSON。用到的函数：

**func** **Marshal**(v **interface**{}) ([]**byte**, error)

该函数递归遍历v的结构，生成对应的JSON。

### map转JSON

**package** main **import** ( "encoding/json" "fmt" ) **func** **main**() { m := **map**[**string**][]**string**{ "level": {"debug"}, "message": {"File not found", "Stack overflow"}, } **if** data, err := json.Marshal(m); err == nil { fmt.Printf("%s\n", data) } }

输出：

{"level":["debug"],"message":["File not found","Stack overflow"]}

大家可以看到Marshal函数返回的JSON字符串是没有空白字符和缩进的，这种紧凑的表示形式是最常用的传输形式，但是不好阅读。如果需要为前端生成便于阅读的格式，可以调json.MarshaIndent，该函数有两个参数表示每一行的前缀和缩进方式：

**if** data, err := json.MarshalIndent(m, "", " "); err == nil { fmt.Printf("%s\n", data) }

输出：

{ "level": [ "debug" ], "message": [ "File not found", "Stack overflow" ] }

在编码过程中，json包会将Go的类型转换为JSON类型，转换规则如下： bool -> JSON boolean 浮点数, 整数, Number -> JSON number string -> JSON string 数组、切片 -> JSON数组 []byte -> base64 string struct、map -> JSON object

### 结构体转JSON

经常会使用结构体来转换成JSON。json包是通过反射机制来实现编解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析：

**package** main **import** ( "encoding/json" "fmt" ) **type** DebugInfo **struct** { Level **string** Msg **string** author **string** *// 未导出字段不会被json解析(首字母小写)* } **func** **main**() { dbgInfs := []DebugInfo{ DebugInfo{"debug", `File: "test.txt" Not Found`, "Cynhard"}, DebugInfo{"", "Logic error", "Gopher"}, } **if** data, err := json.Marshal(dbgInfs); err == nil { fmt.Printf("%s\n", data) } }

输出结果如下：

[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},{"Level":"","Msg":"Logic error"}]

### 结构体字段标签

json包在解析结构体时，如果遇到key为json的字段标签，则会按照一定规则解析该标签：第一个出现的是字段在JSON串中使用的名字，之后为其他选项，例如omitempty指定空值字段不出现在JSON中。如果整个value为"-"，则不解析该字段。例如将上例中的结构体改为如下：

**type** DebugInfo **struct** { Level **string** `json:"level,omitempty"` *// Level解析为level,忽略空值* Msg **string** `json:"message"` *// Msg解析为message* Author **string** `json:"-"` *// 忽略Author* }

则输出为：

[{"level":"debug","message":"File: \"test.txt\" Not Found"},{"message":"Logic error"}]

### 匿名字段

json包在解析匿名字段时，会将匿名字段的字段当成该结构体的字段处理：

**package** main **import** ( "encoding/json" "fmt" ) **type** Point **struct**{ X, Y **int** } **type** Circle **struct** { Point Radius **int** } **func** **main**() { **if** data, err := json.Marshal(Circle{Point{50, 50}, 25}); err == nil { fmt.Printf("%s\n", data) } }

输出结果：

{"X":50,"Y":50,"Radius":25}

### 注意

Marshal函数只有在转换成功的时候才会返回数据，在转换的过程中我们需要注意几点：

* JSON对象只支持string作为key，所以要编码一个map，那么必须是map[string]T这种类型(T是Go语言中任意的类型)
* Channel, complex和function是不能被编码成JSON的
* 指针在编码的时候会输出指针指向的内容，而空指针会输出null

## JSON解析

将JSON转换为Go数据类型。用到的函数：

**func** **Unmarshal**(data []**byte**, v **interface**{}) **error**

此函数将data表示的JSON转换为v：

### JSON转数组

**package** main **import** ( "encoding/json" "fmt" ) **func** **main**() { data := `[{"Level":"debug","Msg":"File: \"test.txt\" Not Found"},` + `{"Level":"","Msg":"Logic error"}]` **var** dbgInfos []**map**[**string**]**string** json.Unmarshal([]**byte**(data), &dbgInfos) fmt.Println(dbgInfos) }

输出为：

[**map**[Level:debug Msg:File: "test.txt" Not Found] **map**[Level: Msg:Logic error]]

在解码过程中，json包会将JSON类型转换为Go类型，转换规则如下： JSON boolean -> bool

JSON number -> float64 JSON string -> string JSON数组 -> []interface{} JSON object -> map null -> nil

### JSON转结构体

JSON可以转换成结构体。同编码一样，json包是通过反射机制来实现解码的，因此结构体必须导出所转换的字段，不导出的字段不会被json包解析，另外解析时不区分大小写：

**package** main **import** ( "encoding/json" "fmt" ) **type** DebugInfo **struct** { Level **string** Msg **string** author **string** *// 未导出字段不会被json解析* } **func** (dbgInfo DebugInfo) **String**() **string** { **return** fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg) } **func** **main**() { data := `[{"level":"debug","msg":"File Not Found","author":"Cynhard"},` + `{"level":"","msg":"Logic error","author":"Gopher"}]` **var** dbgInfos []DebugInfo json.Unmarshal([]**byte**(data), &dbgInfos) fmt.Println(dbgInfos) }

输出结果如下：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

### 结构体字段标签

解码时依然支持结构体字段标签，规则和编码时一样：

**package** main **import** ( "encoding/json" "fmt" ) **type** DebugInfo **struct** { Level **string** `json:"level"` *// level 解码为 Level* Msg **string** `json:"message"` *// message 解码为 Msg* Author **string** `json:"-"` *// 忽略Author* } **func** (dbgInfo DebugInfo) **String**() **string** { **return** fmt.Sprintf("{Level: %s, Msg: %s}", dbgInfo.Level, dbgInfo.Msg) } **func** **main**() { data := `[{"level":"debug","message":"File Not Found","author":"Cynhard"},` + `{"level":"","message":"Logic error","author":"Gopher"}]` **var** dbgInfos []DebugInfo json.Unmarshal([]**byte**(data), &dbgInfos) fmt.Println(dbgInfos) }

则结果为：

[{Level: debug, Msg: File Not Found} {Level: , Msg: Logic error}]

### 匿名字段

编码时，和解码类似，在解码JSON时，如果找不到字段，则查找字段的字段：

**package** main **import** ( "encoding/json" "fmt" ) **type** Point **struct**{ X, Y **int** } **type** Circle **struct** { Point Radius **int** } **func** **main**() { data := `{"X":80,"Y":80,"Radius":40}` **var** c Circle json.Unmarshal([]**byte**(data), &c) fmt.Println(c) }

输出结果：

{{80 80} 40}

## 解析未知结构的JSON

如果要解码一段未知结构的JSON，只需将这段JSON数据解码输出到一个空接口即可。在解码JSON数据的过程中，JSON数据里边的元素类型将做如下转换：

* JSON中的布尔值将会转换为Go中的bool类型；
* 数值会被转换为Go中的float64类型（包括int）；
* 字符串转换后还是string类型；
* JSON数组会转换为[]interface{}类型
* JSON对象会转换为map[string]interface{}类型；
* null值会转换为nil。

在Go的标准库encoding/json包中，允许使用map[string]interface{}和[]interface{}类型的值来分别存放未知结构的JSON对象和数组，示例代码如下：

**func** **main**() { b := []**byte**(`{"Title":"Go语言编程", "Authors":["XuShiwei","HughLv","Pandaman","GuaguaSong","HanTuo","BertYuan","XuDaoli"], "Publisher":"ituring.com.cn", "IsPublished":true, "Price":9.99, "Sales":1000000}`) **var** r **interface**{} err := json.Unmarshal(b, &r) **if** err != nil { log.Fatal(err) } fmt.Printf("%T %v", r, r) gobook, ok := r.(**map**[**string**]**interface**{}) **if** ok { **for** k, v := **range** gobook { **switch** v2 := v.(**type**) { **case** **string**: fmt.Println(k, "is string", v2) **case** **int**: fmt.Println(k, "is int", v2) **case** **float64**: fmt.Println(k, "is float64", v2) **case** **bool**: fmt.Println(k, "is bool", v2) **case** []**interface**{}: fmt.Println(k, "is an array:") **for** i, iv := **range** v2 { fmt.Println(i, iv) } **default**: fmt.Printf("%v is another type not handle yet with %T, %v\n",k, v2,v2) } } } }

输出结果如下：

**map**[**string**]**interface** {} **map**[Title:Go语言编程 Authors:[XuShiwei HughLv Pandaman GuaguaSong HanTuo BertYuan XuDaoli] Publisher:ituring.com.cn IsPublished:true Price:9.99 Sales:1e+06]Title is **string** Go语言编程 Authors is an array: 0 XuShiwei 1 HughLv 2 Pandaman 3 GuaguaSong 4 HanTuo 5 BertYuan 6 XuDaoli Publisher is **string** ituring.com.cn IsPublished is **bool** true Price is **float64** 9.99 Sales is **float64** 1e+06

## 流式编解码

json 包提供了Decoder 和 Encoder 用来支持JSON 数据流的读写。函数NewDecoder 和 NewEncoder 封装了io.Reader和io.Writer 接口类型。

**func** **NewDecoder**(r io.Reader) \***Decoder** **func** **NewEncoder**(w io.Writer) \***Encoder**

由于读写操作的普遍性，类型Encoder和Decoder可以用于多种场合，例如读写HTTP 链接，WebSockets或者文件。

**func** **main**() { data := `{"name":"1802班", "Count":54, "student":["周星", "刘德"], "address":{"floor":8, "classroom":"二教室"}}` *//流式编码* dec := json.NewDecoder(bytes.NewBufferString(data)) **var** v **map**[**string**]**interface**{} **if** err := dec.Decode(&v); err != nil { log.Println(err) **return** } **for** k := **range** v { **if** k == "name" { delete(v, k) } } fmt.Printf("%T %v\n", v, v) *//流式解码* buffer := bytes.Buffer{} enc := json.NewEncoder(&buffer) **if** err := enc.Encode(&v); err != nil { log.Println(err) } fmt.Printf("%v", buffer.String()) }