2013/2/23

GOLANG

Go 网络编程

Network programming with Go

http://biz366.com/Network-programming-with-Go/zh/index.html

目　录

Architecture

架构

This chapter covers the major architectural features of distributed sytems.

本章涵盖了分布式系统架构的主要特性。

Introduction

前言

You can't build a system without some idea of what you want to build. And you can't build it if you don't know the environment in which it will work. GUI programs are different to batch processing programs; games programs are different to business programs; and distributed programs are different to standalone programs. They each have their approaches, their common patterns, the problems that typically arise and the solutions that are often used.

你若不知道你想构建什么，就无法构建一个系统。而如果你不知道它会在何种环境下工作，也同样不行。就像 GUI 程序不同于批处理程序，游戏程序不同于商业程序一样，分布式程序也不同于独立的程序。它们都有各自的方法，常见的模式，经常出现的问题以及常用的解决方案。

This chapter covers the highl evel architectural aspects of distributed systems. There are many ways of looking at such systems, and many of these are dealt with.

本章涵盖了分布式系统的上层架构，从多种角度考虑了这样的系统及其依赖。

Protocol Layers

协议层

Distributed systems are *hard*. There are multiple computers involved, which have to be connected in some way. Programs have to be written to run on each computer in the system and they all have to co-operate to get a distributed task done.

分布式系统很*复杂*，它涉及到多台计算机的连接方式。我们编写的程序必须能在该系统中的每一台计算机上运行，它们必须都能协同操作来完成一项分布式任务。

The common way to deal with complexity is to break it down into smaller and simpler parts. These parts have their own structure, but they also have defined means of communicating with other related parts. In distributed systems, the parts are called *protocol layers* and they have clearly defined functions. They form a stack, with each layer communicating with the layer above and the layer below. The communication between layers is defined by protocols.

解决这种复杂性的一般方法，就是将它分解为更小更简单的部分。这些部分都有它们自己的结构，但也定义了与其它相关部分进行通信的方式。在分布式系统中，这种部分称为*协议层*，它们的功能都有明确的定义。它们在一起形成层次结构，并与其各自的上下层进行通行。层次之间的通信则由协议来定义。

Network communications requires protocols to cover high-level application communication all the way down to wire communication and the complexity handled by encapsulation in protocol layers.

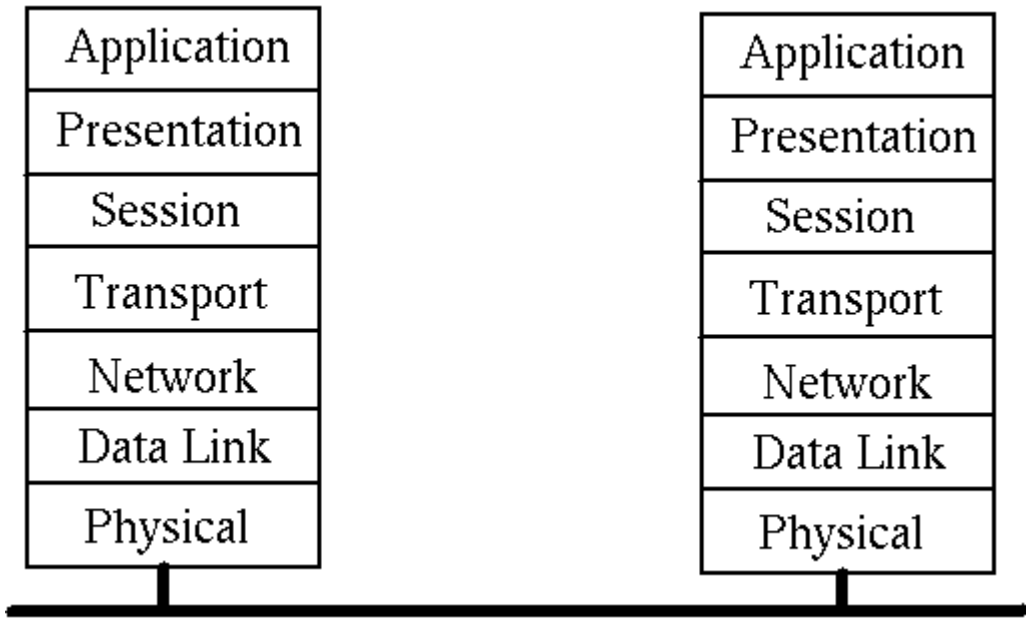
网络通信所需的协议覆盖了从上层应用通信一直到底层有线通信的所有方式，它们的复杂性通过在协议层中进行封装来处理。

ISO OSI Protocol

ISO OSI 协议

Although it was never properly implemented, the OSI (Open Systems Interconnect) protocol has been a major influence in ways of talking about and influencing distributed systems design. It is

commonly given in the following figure:



尽管 OSI（开放系统互联）协议从未被完整地实现过，但它仍对分布式系统的讨论和设计产生了十分重要的影响。它的结构大致为下图所示：

OSI layers

OSI 层

The function of each layer is:

每一层的功能为：

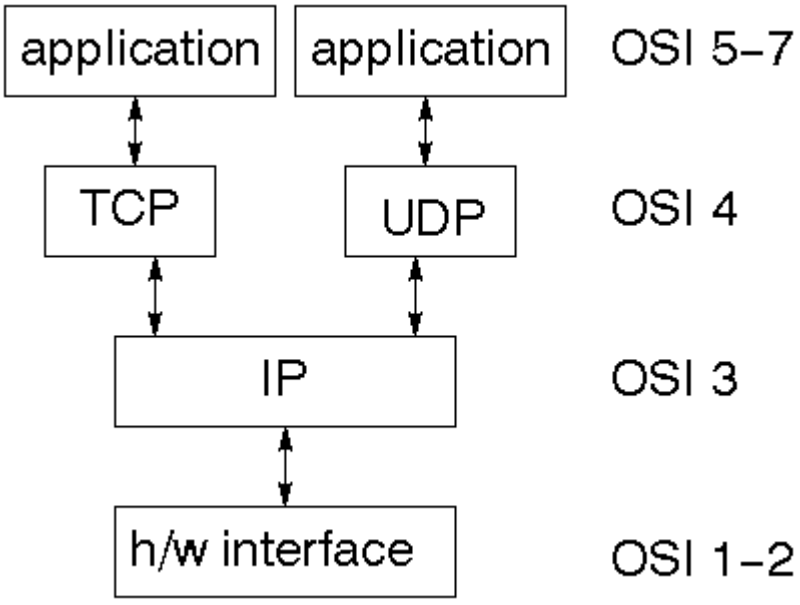
- Network layer provides switching and routing technologies
- Transport layer provides transparent transfer of data between end systems and is responsible for end-to-end error recovery and flow control
- Session layer establishes, manages and terminates connections between applications.
- Presentation layer provides independance from differences in data representation (e.g. encryption)
- Application layer supports application and end-user processes

- 网络层提供交换和路由技术
- 传输层在终端系统间提供透明的数据传输，并负责端对端的错误恢复及流程控制
- 会话层在应用间建立、管理并结束连接
- 表现层提供数据表现差异的独立性（例如加密）
- 应用层支持应用与最终用户的处理

TCP/IP Protocol

TCP/IP 协议

While the OSI model was being argued, debated, partly implemented and fought over, the DARPA internet research project was busy building the TCP/IP protocols. These have been immensely succesful and have led to The Internet (with capitals). This is a much simpler stack:



当 OSI 标准模型正在为实现细节闹得不可开交时，DARPA 互联网技术项目却在忙着构建 TCP/IP 协议。它们取得了极大的成功，并引领了 Internet（首字母大写），因为这是个更简单的层次结构：

Some Alternative Protocols

一些可选的协议

Although it almost seems like it, the TCP/IP protocols are not the only ones in existence and in the long run may not even be the most successful. There are many protocols occupying significant niches, such as

尽管现在到处都是 TCP/IP 协议，但它并不是唯一存在的。从长远来看，它甚至不会是最成功的。还有些协议占有重要的地位，比如：

- Firewire
- USB
- Bluetooth
- WiFi

- 火线
- USB
- 蓝牙
- WiFi

Thre is active work continuing on many other protocols, even quite bizarre ones such as those for the "internet in space."

还有些其它的协议在继续活跃地工作，甚至还有些像“太空互联网”这样奇怪的协议。

The focus in this book will be on the TCP/IP, but you should be aware of these other ones.

本书将重点介绍 TCP/IP，但你也应当了解一些其它的协议。

Networking

网络

A network is a communications system for connecting end systems called hosts. The mechanisms of connection might be copper wire, ethernet, fibre optic or wireless, but that won't concern us here. A

local area network (LAN) connects computers that are close together, typically belonging to a home, small organisation or part of a larger organisation.

网络是一个通信系统，它连接了称为主机的最终系统。这种连接机制可以是铜线、以太网、光纤或无线，但这些与我们无关。局域网（LAN）将计算机紧密地连接在一起，一般为家庭、小型组织或大型组织的一部分。

A Wide Area Network (WAN) connects computers across a larger physical area, such as between cities. There are other types as well, such as MANs (Metropolitan Area Network), PANs (Personal Are Networks) and even BANs (Body Are Network).

广域网（WAN）连接起一个更大物理区域的计算机，例如城际间。还有些其它的类型，如城域网（MAN）、个人域网（PAN）甚至人体域网（BAN）。

An internet is a connection of two or more distinct networks, typically LANs or WANs. An intranet is an internet with all networks belonging to a single organisation.

互联网是多个不同网络的连接，一般为 LAN 或 WAN。内联网是属于某个组织的所有网络加上互联网。

There are significant differences between an internet and an intranet. Typically an intranet will be under a single administrative control, which will impose a single set of coherent policies. An internet on the other hand will not be under the control of a single body, and the controls exercised over different parts may not even be compatable.

互联网与内联网之间有明显的不同。一般来说，一个内联网处在单一的管控之下，它将被应用一组统一的策略。另一方面，一个互联网则不会在单一主体的控制之下，控制的不同部分甚至可能会不兼容。

A trivial example of such differences is that an intranet will often be restricted to computers by a small number of vendors running a standardised version of a particular operating system. On the other hand, an internet will often have a smorgasborg of different computers and operating systems.

这种不同的一个例子，就是一个内联网通常被少量供应商提供的，运行着特定操作系统标准化版本的计算机所限制。另一方面，一个互联网通常有各种各样的计算机和操作系统。

The techniques of this book will be applicable to internets. They will also be valid for intranets, but there you will also find specialised, non-portable systems.

本书中的技术可应用于互联网。它们对内联网也是有效的，但你也会发现一些专有的，不可移植的系统。

And then there is the "mother" of all internets: The Internet. This is just a very, very large internet that connects us to Google, my computer to your computer and so on.

所有互联网都有一个“母网”：因特网。它其实就是个非常巨大的互联网，它将我们与 Google、我们的计算机等等互相连接起来。

Gateways

网关

A gateway is a generic term for an entity used to connect two or more networks. A repeater operates at the physical level copies the information from one subnet to another. A bridge operates at the data link layer level and copies frames between networks. A router operates at the network level and not only moves information between networks but also decides on the route.

网关是一个统称，它用于连接起一个或多个网络。其中的中继器在物理层面上进行操作，它将信息从一个子网复制到另一个子网上。桥接在数据连接层面上进行操作，它在网络之间复制帧。路由器在网络层面上进行操作，它不仅在网络之间复制信息，还决定了信息的传输路线。

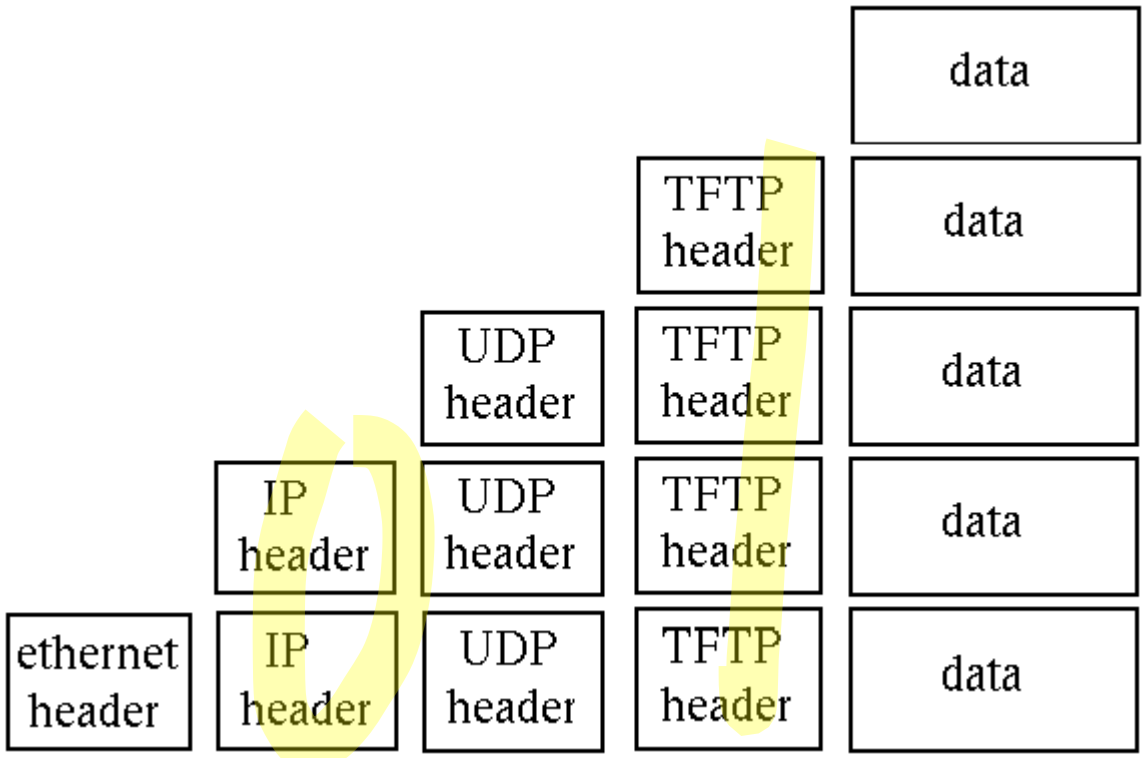
Packet encapsulation

数据包封装

The communication between layers in either the OSI or the TCP/IP stacks is done by sending packets of data from one layer to the next, and then eventually across the network. Each layer has administrative information that it has to keep about its own layer. It does this by adding header information to the packet it receives from the layer above, as the packet passes down. On the receiving side, these headers are removed as the packet moves up.

在 OIS 或 TCP/IP 协议栈层与层之间的通信，是通过将数据包从一个层发送到下一个层，最终穿过整个网络的。每一层都有必须保持其自身层的管理信息。从上层接收到的数据包在向下传递时，会添加头信息。在接收端，这些头信息会在向上传递时移除。

For example, the TFP (Trivial File Transfer Protocol) moves files from one computer to another. It uses the UDP protocol on top of the IP protocol, which may be sent over Ethernet. This looks like:



The packet transmitted over ethernet, is of course the bottom one.

例如，TFTP（普通文件传输协议）将文件从一台计算机移动到另一台上。它使用 IP 协议上的 UDP 协议，该协议可通过以太网发送。看起来就像这样：

通过以太网发送的数据包，当然是底部那个。

Connection Models

连接模型

In order for two computers to communicate, they must set up a path whereby they can send at least one message in a session. There are two major models for this:

为了两个计算机进行通信，就必须建立一个路径，使他们能够在一个会话中发送至少一条消息。有两个主要的模型：

- Connection oriented
- Connectionless

- 面向连接模型
- 无连接模型

Connection oriented

面向连接模型

A single connection is established for the session. Two-way communications flow along the connection. When the session is over, the connection is broken. The analogy is to a phone conversation. An example is TCP

即为会话建立单个连接，沿着连接进行双向通信。当会话结束后，该连接就会断开。这类似于电话交谈。例子就是 TCP。

Connectionless

无连接模型

In a connectionless system, messages are sent independant of each other. Ordinary mail is the analogy. Connectionless messages may arrive out of order. An example is the IP protocol. Connection oriented transports may be established on top of connectionless ones - TCP over IP. Connectionless transports my be established on top of connection oriented ones - HTTP over TCP.

在无连接系统中，消息的发送彼此独立。这类似于普通的邮件。无连接模型的消息可能不按顺序抵达。例子就是 IP 协议。面向连接的传输可通过无连接模型——基于 IP 的 TCP 协议建立。无连接传输可通过面向连接模型——基于 IP 的 HTTP 协议建立。

There can be variations on these. For example, a session might enforce messages arriving, but might not guarantee that they arrive in the order sent. However, these two are the most common.

这些是可变的。例如，会话可能会强制消息抵达，但可能无法保证它们按照发送的顺序抵达。不过这两个是最常见的。

Communications Models

通信模型

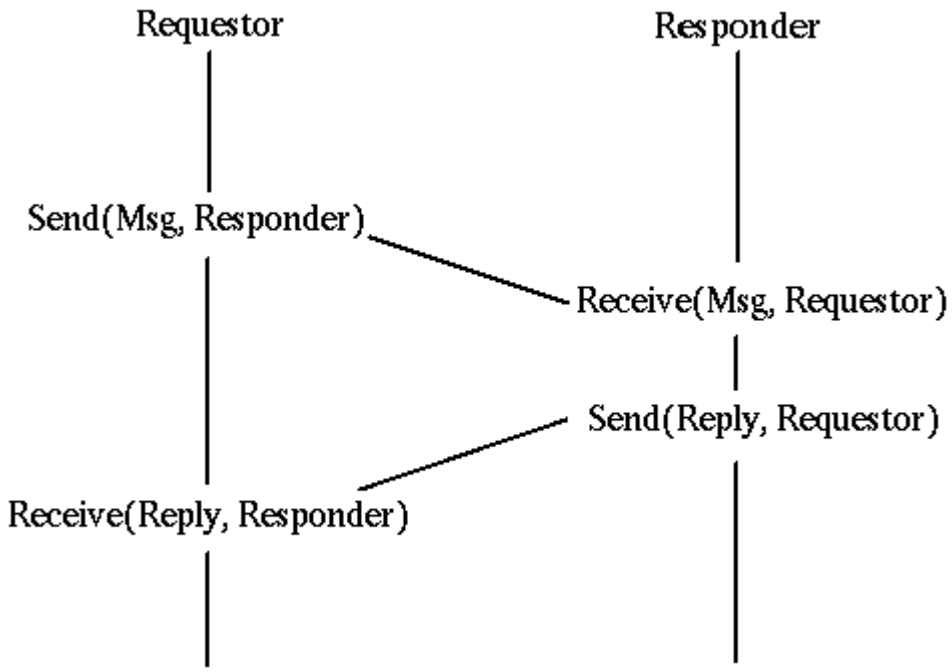
Message passing

消息传递

Some non-procedural languages are built on the principle of *message passing*. Concurrent languages often use such a mechanism, and the most well known example is probably the Unix pipeline. The Unix pipeline is a pipeline of bytes, but there is not an inherent limitation: Microsoft's PowerShell can send objects along its pipelines, and concurrent languages such as Parlog could send arbitrary logic data structures in messages between concurrent processes.

一些非过程化语言建立在*消息传递*原理上。并发语言经常使用这种机制，最有名的大概要数 Unix 的管道了。Unix 管道就是一管字节，但它并没有固定的限制：微软的 PowerShell 可沿着其管道发送对象；而像 Parlog 这样的并发语言，则能在并发的进程之间，将任意的逻辑数据结构当做消息来发送。

Message passing is a primitive mechanism for distributed systems. Set up a connection and pump some data down it. At the other end, figure out what the message was and respond to it, possibly sending messages back. This is illustrated by



消息传递是分布式系统最基本的机制，也就是建立连接并通过它传输一些数据。在另一端则需要理解这些消息的意思并做出响应，有时还需要返回一些消息。如下图所示：

Low level event driven systems such as the X Window System function in a somewhat similar way: wait for message from a user (mouse clicks, etc), decode them and act on them.

诸如 X 窗口系统之类的底层事件驱动系统功能也采用了类似的方式：等待用户的消息（如鼠标点击等），对它们进行解码并做出反应。

Higher level event driven systems assume that this decoding has been done by the underlying system and the event is then dispatched to an appropriate object such as a ButtonPress handler. This can also be done in distributed message passing systems, whereby a message received across the network is partly decoded and dispatched to an appropriate handler.

更高层的事件驱动系统则假定底层系统已经解码完成，接着该事件被分配给适当的对象，如 ButtonPress 处理程序。这也适用于分布式消息传递系统，通过对从网络接收的消息进行部分解码，并分配给适当的处理程序。

Remote procedure call

远程过程调用

In any system, there is a transfer of information and flow control from one part of the system to another. In procedural languages this may consist of the procedure call, where information is placed on a call stack and then control flow is transferred to another part of the program.

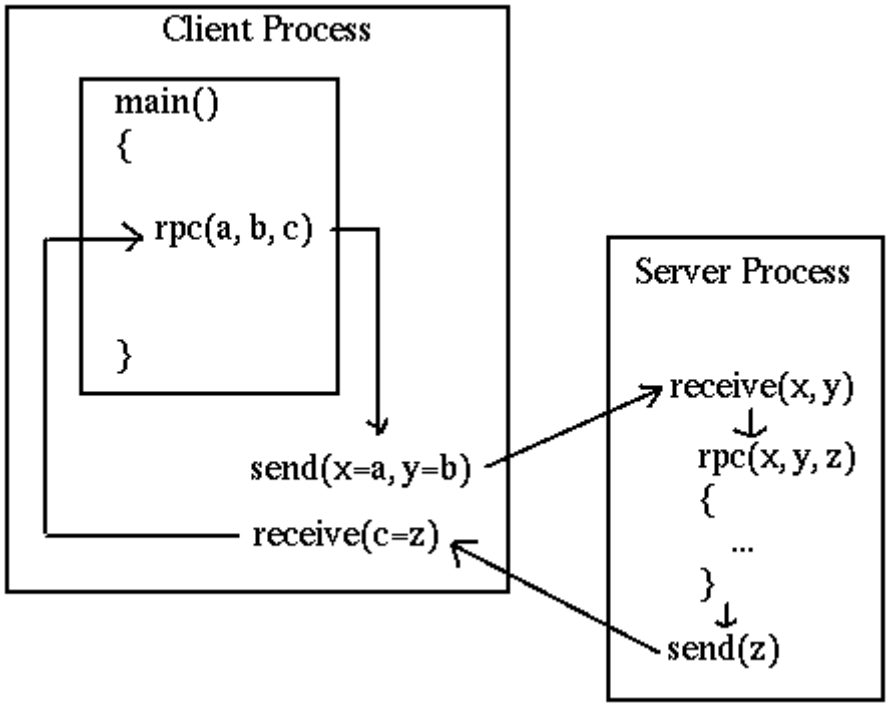
在任何系统中，都有信息传输和流程控制来将该系统的一部分传到另一部分。在过程化语言中，它由过程调用来组成，其中的信息被放置到调用栈上，接着控制流程被传递至该程序的另一部分。

Even with procedure calls, there are variations. The code may be statically linked so that control transfers from one part of the program's executable code to another part. Due to the increasing use of library routines, it has become commonplace to have such code in dynamic link libraries (DLLs), where control transfers to an independent piece of code.

甚至过程调用也有变化。代码可被静态链接，以便于控制从该程序可执行代码的一部分传输到另一部分。随着库例程的使用日益增多，将这类代码作为动态链接库（DLL）也变得司空见惯了，它用来控制传输独立的代码片段。

DLLs run in the same machine as the calling code. it is a simple (conceptual) step to transfer control to a procedure running in a different machine. The mechanics of this are not so simple! However, this model of control has given rise to the "remote procedure call" (RPC) which is discussed in much

detail in a later chapter. This is illustrated by



DLL 作为调用代码运行在相同的机器上。尽管对于不同机器上运行的过程传输控制来说，这种机制（在概念上）是一种简单的手段，但它实际上可不怎么简单！不过，这种控制模型却催生了“远程过程调用”（RPC），更多关于它的详情会在后面的章节中讨论。如下图所示：

There is an historical oddity called the "lightweight remote procedure call" invented by Microsoft as they transitioned from 16-bit to 32-bit applications. A 16-bit application might need to transfer data to a 32-bit application *on the same machine*. That made it lightweight as there was no networking! But it had many of the other issues of RPC systems in data representations and conversion.

微软在从 16 位应该过渡到 32 位时，曾发明过一种称为“轻量远程过程调用”的奇怪东西。16 位应用可能需要*在相同的机器上*向 32 位应用传输数据。由于没有网络，竟使得它很轻量！不过，它也有 RPC 系统在数据表达和转换上的其它问题。

Distributed Computing Models

分布式计算模型

At the highest lvel, we could consider the equivalence or the non-equivalence of components of a distributed system. The most common occurrence is an asymmetric one: a client sends requests to a server, and the server responds. This is a *client-server* system.

在最上层，我们可以考虑分布式系统的组件是否等价。最常见的就是不对等的情况：客户端向服务器发送请求，然后服务端响应。这就是*客户端-服务器*系统。

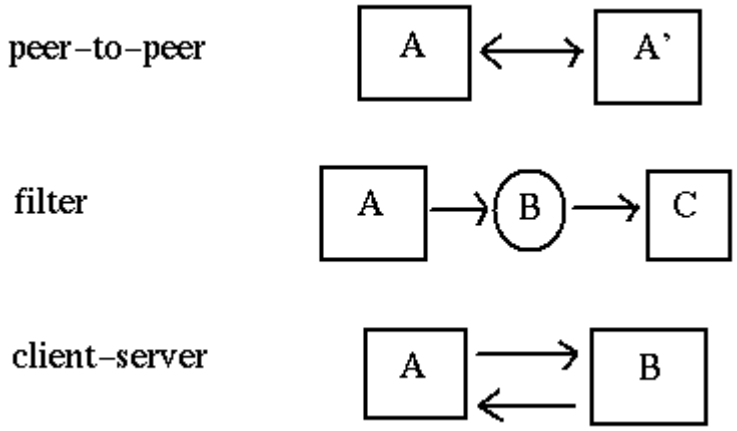
If both components are equivalent, both able to initiate and to respond to messages, then we have a *peer-to-peer* system. Note that this is a logical classification: one peer may be a 16,000 core mainframe, the other might be a mobile phone. But if both can act similarlym then they are peers.

若两个组件等价，且均可发起并响应信息，那么我们就有了一个*点对点*系统。注意这是个逻辑上的分类：一点可能是 16,000 个核心主机，而另一点可能只是个移动电话。但如果二者的行为类似，那么它们就都是点。

A third model is the so-called *filter*. Here one component passes information to another which modifies it before passing it to a third. This is a fairly common model: for example, the middle component gets information from a database as SQl records and transforms it into an HTML table for the third component (which might be a browser).

第三种模型也就是所谓的*过滤器*。有一个组件将信息传至另一个组件，它在修改该信息后会传至第三个组件。这是个相当普遍的模型：例如，中间组件通过 SQL 从数据库中获取信息，并将其转换为 HTML 表单提供给第三个组件（它可能是个浏览器）。

These are illustrated as:

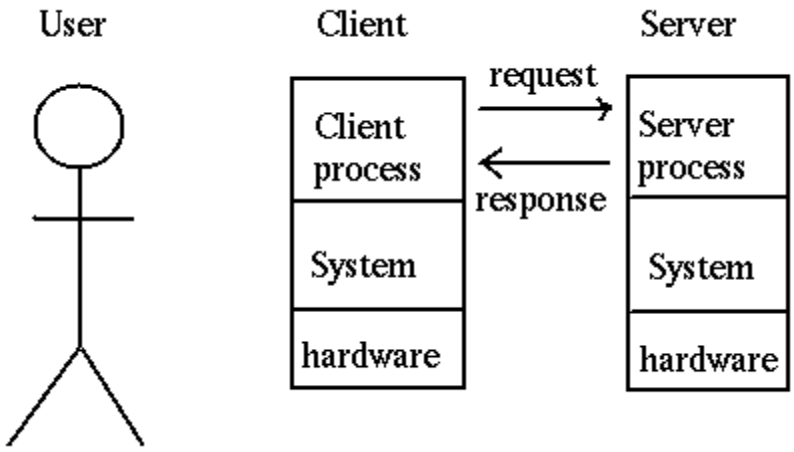


如下所示：

Client/Server System

客户端/服务器系统

Another view of a client server system is

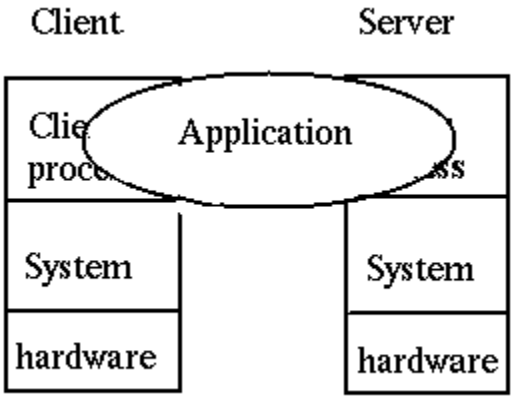


客户端/服务器系统的另一种方式：

Client/Server Application

客户端/服务器应用

And a third view is

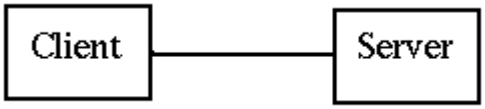


第三种方式：

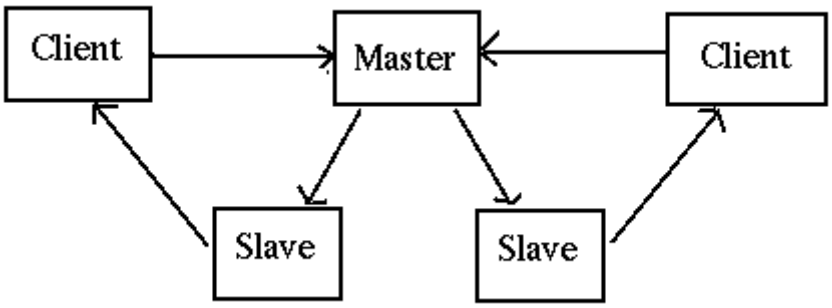
Server Distribution

服务器分布

A client-server systems need not be simple. The basic model is single client, single server



but you can also have multiple clients, single server



In this, the master receives requests and instead of handling them one at a time itself, passes them off to other servers to handle. This is a common model when concurrent clients are possible.

客户端/服务器系统并不简单。其基本模型是单一客户端，单一服务器：

不过你也可以有多个客户端，单一服务器：

这样，主站只需接收请求并处理一次，而无需将它们传递给其它服务器来处理。当客户端可能并发时，这就是个通用的模型。

There are also single client, multiple servers

which occurs frequently when a server needs to act as a client to other servers, such as a business logic server getting information from a database server. And of course, there could be multiple clients with multiple servers

还有单一客户端，多个服务器的情况：

当一个服务器需要作为其它服务器的客户端时，这种情况就会经常发生，例如当业务逻辑服务器从数据库服务器获取信息时。当然，还可以有多个客户端，多个服务器的情况。

Component Distribution

组件分布

A simple but effective way of decomposing many applications is to consider them as made up of three parts:

分解一些应用的一个简单有效的方式就是把它们看做三部分：

- Presentation component
- Application logic
- Data access

- 表现组件
- 应用逻辑
- 数据访问

The *presentation component* is responsible for interactions with the user, both displaying data and gathering input. it may be a modern GUI interface with buttons, lists, menus, etc, or an older command-line style interface, asking questions and getting answers. The details are not important at this level.

*表现组件*负责与用户进行交互，即显示数据和采集输入。它可以是带有按钮、列表和菜单等等的现代 GUI 界面，或较老的命令行式界面，询问问题并获取答案。在这一层上，具体详情并不重要。

The *application logic* is responsible for intrepreting the users' responses, for applying business rules, for preparing queries and managing responses from the thir component.

*应用逻辑*组件负责解释用户的响应，根据应用业务规则，准备查询并管理来自其组件的响应。

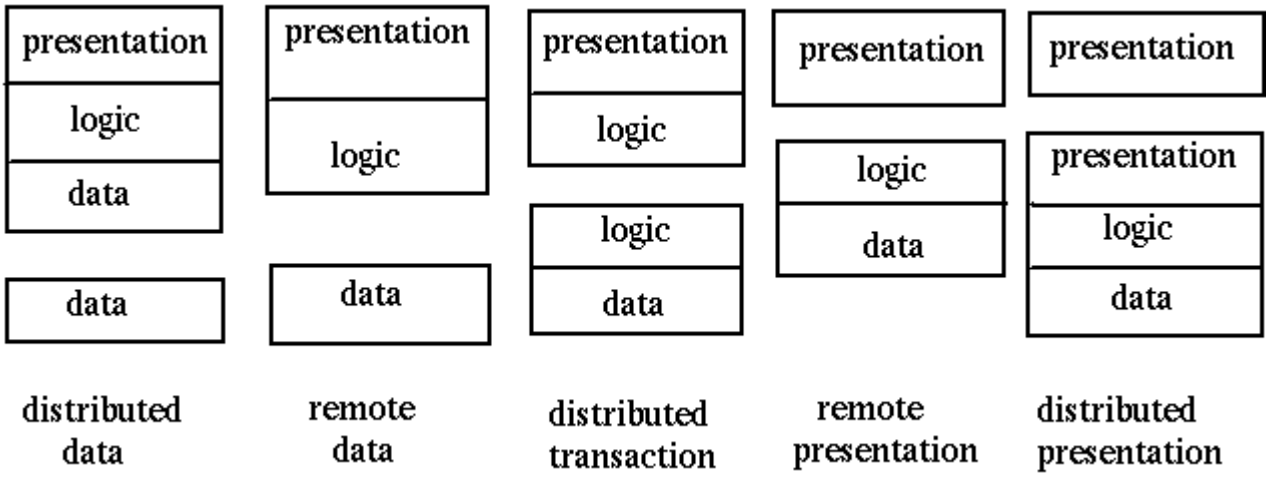
The *data access* component is responsible for stroing and retrieving data. This will often be through a database, but not necessarily.

*数据访问*组件负责存储并检索数据。这一般是通过数据库进行，不过也不一定。

Gartner Classification

Gartner 分类

Based on this threefold decomposition of applicaitons, Gartner considered how the components might be distributed in a client-server sysem. They came up with five models:

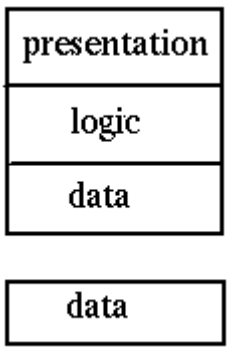


基于这三部分的应用划分，Gartner 公司考虑了这些组件在客户端-服务器系统中如何分布。他们想出了五种模型：

Example: Distributed Database

示例：分布式数据库

- Gartner classification: 1
- Gartner 第一种分类：



Modern mobile phones make good examples of this: due to limited memory they may store a small

part of a database locally so that they can usuall respond quickly. However, if data is required that is not held locally, then a request may be made to a remote database for that additional data.

现代的移动电话就是个很好的例子:由于内存有限,它们只能通过存储一小部分本地数据库,因此它们通常能快速响应。若请求的数据不在本地,那么可为该附加数据请求远程数据库。

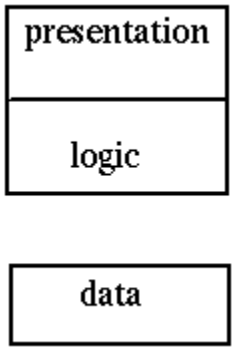
Google maps forms another good example. Al of the maps reside on Google's servers. When one is requested by a user, the "nearby" maps are also downloaded into a small database in the browser. When the user moves the map a little bit, the extra bits required are already in the local store for quick response.

Google 地图的形式是另一个很好的例子。所有的地图都在 Google 的服务器上。当用户请求时,“附近的”地图也会下载为一个浏览器中的小型数据库。当用户移动了一点地图时,额外的一点请求已经为快速响应在本地存储中了。

Example: Network File Service

示例:网络文件服务

Gartner classification 2 allows remote clients acess to a shared file system



There are many examples of scuh systems: NFS, Microsoft shares, DCE, etc

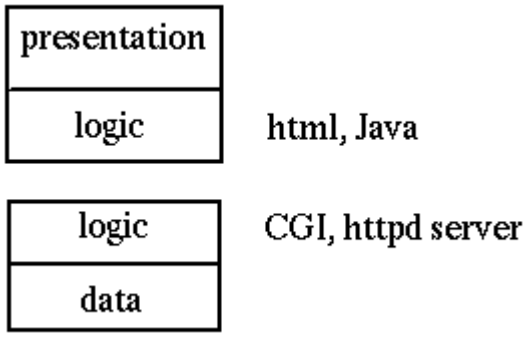
Gartner 第二种分类允许远程客户端访问已共享的文件系统:

这里有一些这类系统的例子:NFS、Microsoft 共享和 DCE 等等。

Example: Web

示例：Web

An example of Gartner classification 3 is the Web with Java applets. This is a distributed hypertext system, with many additional mechanisms

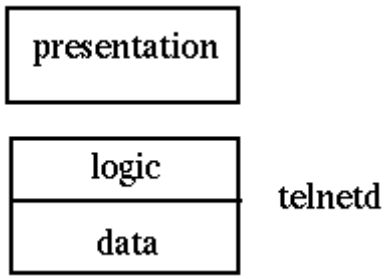


Gartner 第三种分类的一个例子就是 Web 上的小型 Java 应用。以下为带有一些附加机制的分布式超文本系统：

Example: Terminal Emulation

示例：终端仿真

An example of Gartner classification 4 is terminal emulation. This allows a remote system to act as a normal terminal on a local system.



Telnet is the most common example of this.

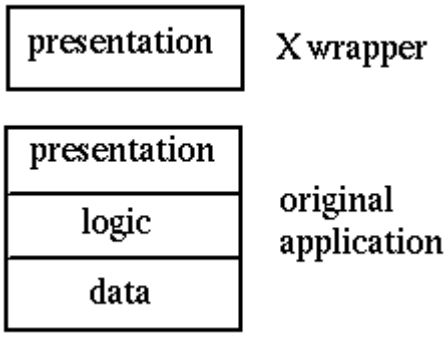
Gartner 第四种分类就是终端仿真。这允许远程系统在本地系统上作为普通的终端：

Telnet 就是最常见的例子。

Example: Expect

示例：预期（Expect）

Expect is a novel illustration of Gartner classification 5. It acts as a wrapper around a classical system such as a command-line interface. It builds an X Window interface around this, so that the user interacts with a GUI, and the GUI in turn interacts with the command-line interface.



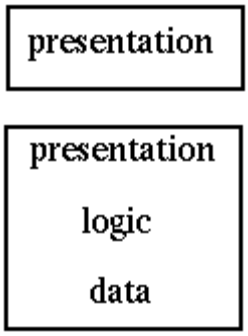
预期（Expect）是 Gartner 第五种分类的一种另类的演示。它的行为类似于命令行接口这样的经典系统。它在此之上建立了 X 窗口界面，以此来让用户与 GUI 进行交互，然后 GUI 转而与命令行界面进行交互。

Example: X Window System

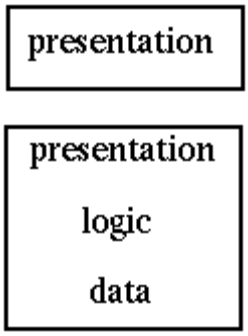
示例：X 窗口系统

The X Window System itself is an example of Gartner classification 5. An application makes GUI calls such as DrawLine, but these are not handled directly but instead passed to an X Window server

for rendering. This decouples the application view of windowing and the display view of windowing.



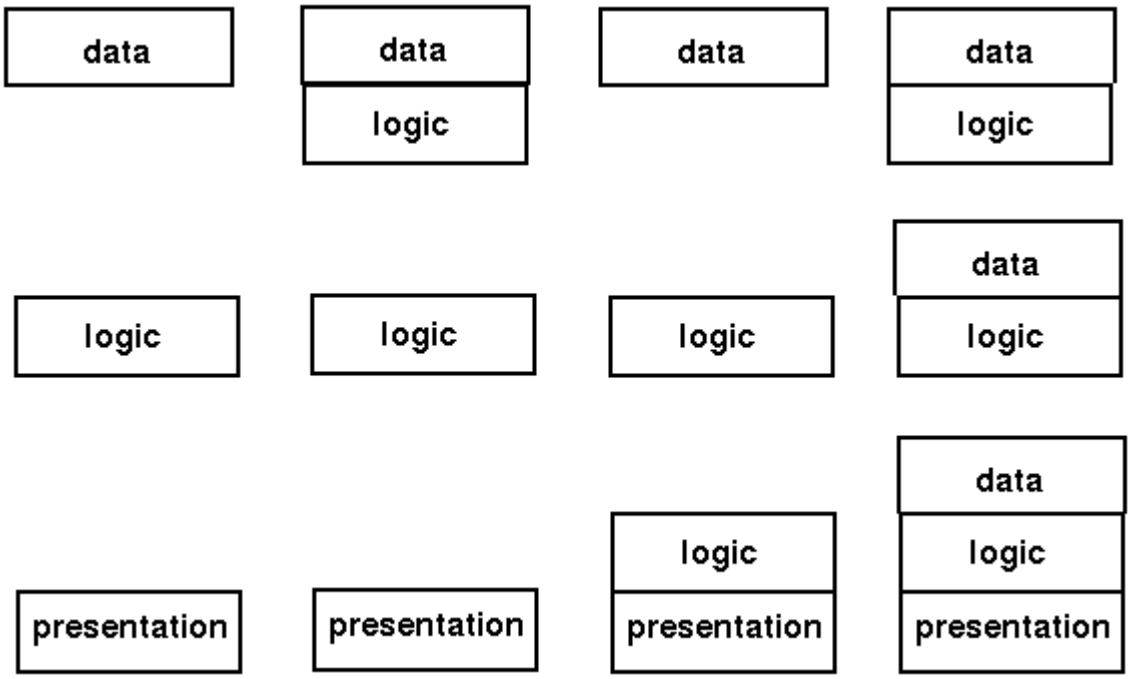
X 窗口系统本身也是 Gartner 第五种分类的一个例子。一个应用进行一次像 DrawLine 这样的 GUI 调用，但它并不直接进行处理，而是传递给 X 窗口服务来渲染。这可以解耦窗口应用视图和窗口显示视图。



Three Tier Models

三层模型

of course, if you have two tiers, then you can have three, four, or more. Some of the three tier possibilities are shown in this diagram:



当然，如果你有两层，你也可以有三层、四层甚至多层。下图展示了一些可能的三层模型：

The modern Web is a good example of the rightmost of these. The backend is made up of a database, often running stored procedures to hold some of the database logic. The middle tier is an HTTP server such as Apache running PHP scripts (or Ruby on Rails, or JSP pages, etc). This will manage some of the logic and will have data such as HTML pages stored locally. The frontend is a browser to display the pages, under the control of some Javascript. In HTML 5, the frontend may also have a local database.

现代 Web 就是最右边那种模型很好的例子。后端建立为一个数据库，经常运行存储过程来保存一些数据库逻辑。中间层是一个 Apache 这样的运行 PHP 脚本（或 Ruby on Rails，或 JSP 页面等）的 HTTP 服务器。这会管理一些逻辑和存储在本地的像 HTML 页面这样的数据。前端为显示由 JavaScript 控制的页面的浏览器。在 HTML5 中，前端也可以有一个本地数据库。

Fat vs thin

“胖”与“瘦”

A common labelling of components is "fat" or "thin". Fat components take up lots of memory and do complex processing. Thin components on the other hand, do little of either. There don't seem to be any "normal" size components, only fat or thin!

组件一般分为“胖”或“瘦”。“胖”组件占用大量的内存来做复杂的处理；“瘦”组件则恰恰相反，只占少量内存，做简单处理。似乎没有任何“正常”大小的组件，只有“胖”或“瘦”！

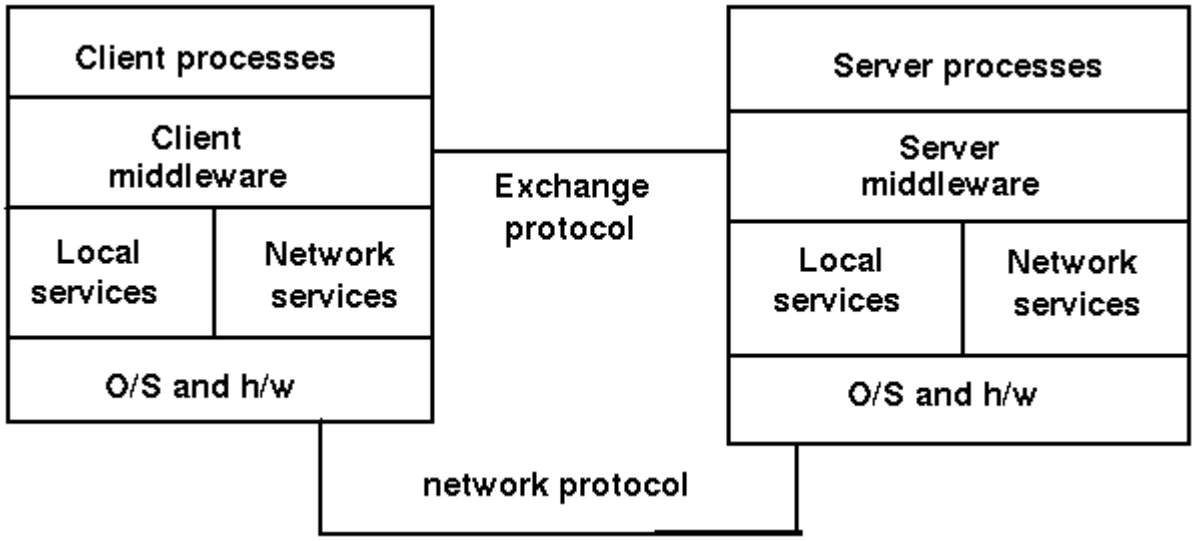
Fatness or thinness is a relative concept. Browsers are often laelled as thin because "all they do is diplay web pages". Firefox on my Linux box takes nearly 1/2 a gigabyte of memory, which I don't regard as small at all!

“胖”或“瘦”的概念是相对的。浏览器经常被分为“瘦”组件，因为“它仅仅显示 Web 页面”。但我的 Linux 盒子中的 Firefox 用了将近 1/2GB 的内存，我可一点也不觉得它很小！

Middleware model

中间件模型

Middleware is teh "glue" connecting components of a distributed system. The middleware model is



中间件是连接器分布式系统组件的“胶水”层。中间件模型如图所示：

Middleware

中间件

Components of middleware include

中间件组件包括：

- The network services include things like TCP/IP
- The middleware layer is application-independent s/w using the network services
- Examples of middleware are: DCE, RPC, Corba
- Middleware may only perform one function (such as RPC) or many (such as DCE)

- 像 TCP/IP 这样的网络服务
- 中间件层是应用独立的，使用网络服务的软件
- 中间件的例子：DCE、RPC、Corba
- 中间件可能只执行一种功能（比如 RPC）或多种功能（比如 DCE）

Middleware examples

中间件示例

Examples of middleware include

中间件的例子包括：

- Primitive services such as terminal emulators, file transfer, email
- Basic services such as RPC
- Integrated services such as DCE, Network O/S
- Distributed object services such as CORBA, OLE/ActiveX
- Mobile object services such as RMI, Jini
- World Wide Web

- 像终端模拟器、文件传输或电子邮件这样的基础服务
- 像 RPC 这样的基础服务
- 像 DCE、网络 O/S 这样的一体化服务
- 像 CORBA、OLE/ActiveX 这样的分布式对象服务
- 像 RMI、Jini 这样的移动对象服务
- 万维网

Middleware functions

中间件的功能

The functions of middleware include

中间件的功能包括：

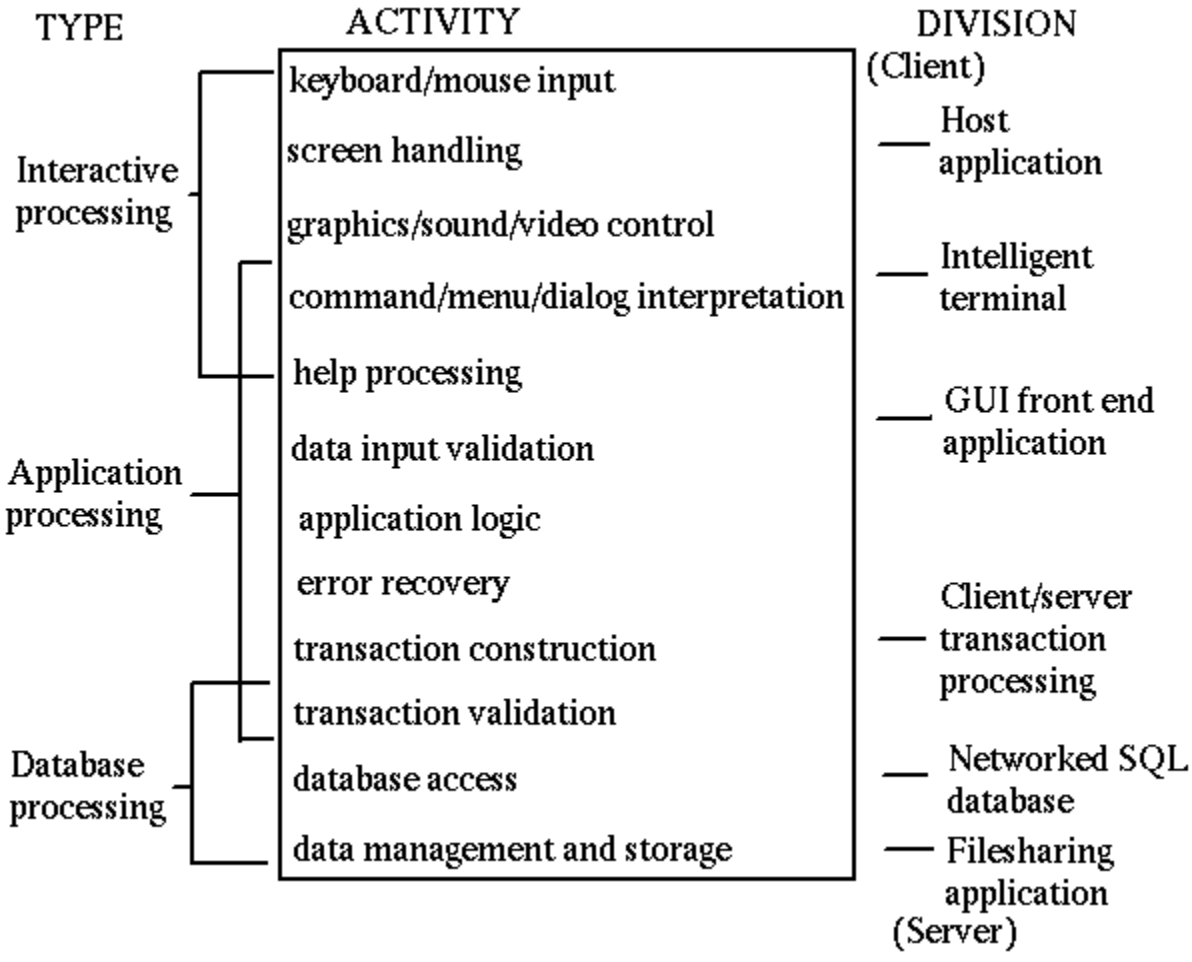
- 在不同计算机上初始化过程
- 进行会话管理
- 允许客户端定位服务器的目录服务
- 进行远程数据访问

- 允许服务器处理多个客户端的并发控制
- 保证安全性和完整性
- 监控
- 终止本地处理和远程处理

Continuum of Processing

连续处理

The Gartner model is based on a breakdown of an application into the components of presentation, application logic and data handling. A finer grained breakdown is



Gartner 模型基于将一个应用分解为表现组件、应用逻辑和数据处理。一个更细粒度的分解方式为：

Points of Failure

故障点

Distributed applications run in a complex environment. This makes them much more prone to failure than standalone applications on a single computer. The points of failure include

分布式应用一般运行在复杂的环境中。这使得它比单一计算机上的独立应用更易发生故障。故障点包括：

- The client side of the application could crash
- The client system may have h/w problems
- The client's network card could fail
- Network contention could cause timeouts
- There may be network address conflicts
- Network elements such as routers could fail
- Transmission errors may lose messages
- The client and server versions may be incompatable
- The server's network card could fail
- The server system may have h/w problems
- The server s/w may crash
- The server's database may become corrupted

- 应用可能会在客户端崩溃
- 客户端系统可能发生硬件问题
- 客户端的网卡可能发生故障
- 网络连接可能超时
- 网络地址可能冲突
- 像路由器这样的网络基础设备可能发生故障
- 传输错误可能会失去消息
- 客户端与服务器的版本可能不兼容
- 服务器的网卡可能发生故障

- 服务器系统可能发生硬件问题
- 服务器的软件可能崩溃
- 服务器的数据库可能损坏

Applications have to be designed with these possible failures in mind. Any action performed by one component must be recoverable if failure occurs in some other part of the system. Techniques such as transactions and continuous error checking need to be employed to avoid errors.

在设计应用时必须考虑这些可能发生的故障。如果故障发生在系统的其它部分，那么由任何一个组件执行的操作都必须可恢复。这就需要采用事务和持续错误检测这类的计算来避免错误。

Acceptance Factors

接受因素

- Reliability
- Performance
- Responsiveness
- Scalability
- Capacity
- Security

- 可靠性
- 性能
- 响应性
- 可扩展性
- 可容性
- 安全性

Transparency

透明度

The "holy grails" of distributed systems are to provide the following:

分布式系统的“圣杯”就是提供以下几点：

- access transparency
- location transparency
- migration transparency
- replication transparency
- concurrency transparency
- scalability transparency
- performance transparency
- failure transparency

- 访问透明度
- 位置透明度
- 迁移透明度
- 赋值透明度
- 并发透明度
- 扩展透明度
- 性能透明度
- 故障透明度

Eight fallacies of distributed computing

分布式计算的八个误区

Sun Microsystems was a company that performed much of the early work in distributed systems, and even had a mantra "The network is the computer." Based on their experience over many years a number of the scientists at Sun came up with the following list of fallacies commonly assumed:

Sun 微系统公司在分布式系统上做很很多早期的工作，他们甚至有一个口头禅：“网络就是计算机”。基于他们多年的经验，Sun 的科学家总结了以下常见误区：

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

1. 网络是可靠的。
2. 风险为零。
3. 带宽是无限的。
4. 网络是安全的。
5. 拓扑结构不会改变。
6. 没有管理员。
7. 传输成本为零。
8. 网络是均等的。

Many of these directly impact on network programming. For example, the design of most remote procedure call systems is based on the premise that the network is reliable so that a remote procedure call will behave in the same way as a local call. The fallacies of zero latency and infinite bandwidth also lead to assumptions about the time duration of an RPC call being the same as a local call, whereas they are magnitudes of order slower.

这些问题直接影响着网络编程。例如，大部分远程过程调用系统的设计都基于网络是可靠的前提，从而导致了远程过程调用的行为与本地调用如出一辙。零风险和无限带宽的误区也导致了 RPC 调用的持续时间与本地调用相同的臆断，但实际上它要比本地调用慢很多。

The recognition of these fallacies led Java's RMI (remote method invocation) model to require every RPC call to potentially throw a **RemoteException**. This forced programmers to at least recognise the

possibility of network error and to remind them that they could not expect the same speeds as local calls.

对于这些错误的认识导致了 Java 的 RMI（远程方法调用）模型要求每一个潜在的 RPC 调用都要抛出一个 **RemoteException** 异常。这迫使程序员至少认识到了网络错误的可能性，并提醒他们不要期望这会与本地调用的速度相同。

Overview of the Go language

Go 语言概括

Introduction

介绍

Please go to the main index for the content pages for network computing.

请访问导航页获取 Go 网络编程的其他页面

I don't feel like writing a chapter introducing Go right now, as there are other materials already available. There are several tutorials on the Go web site:

我目前还不想写介绍 Go 的章节，因为目前已经有很多这方面的材料存在，在 Go 官方网站上有很多这样的入门:

- Getting started
- A Tutorial for the Go Programming Language
- Effective Go

- 安装入门
- Go 程序设计入门
- Go 高级编程

There is an introductory textbook on Go: "Go Programming" by John P. Baugh available from Amazon

目前有一本介绍 Go 的书: "Go Programming" 作者 John P. Baugh Amazon

There is a #golang group on Google+

Google+上有一个讨论组：#golang

Socket-level Programming

套接字级编程

This chapter looks at the basic techniques for network programming. It deals with host and service addressing, and then considers TCP and UDP. It shows how to build both servers and clients using the TCP and UDP Go APIs. It also looks at raw sockets, in case you need to implement your own protocol above IP.

本章将着眼于网络编程的基础方法，将涉及到主机和服务寻址，也会考虑到 TCP 和 UDP。同时也将展示如何使用 GO 的 TCP 和 UDP 相关的 API 来构建服务器和客户端。最后介绍了原生套接字，如果你需要基于 IP 协议实现你自己的协议的话。

Introduction

介绍

There are many kinds of networks in the world. These range from the very old such as serial links, through to wide area networks made from copper and fibre, to wireless networks of various kinds, both for computers and for telecommunications devices such as phones. These networks obviously differ at the physical link layer, but in many cases they also differed at higher layers of the OSI stack.

世上存在很多种网络。它们涵盖了从古老如串行链路，到为计算机或手机这样的通讯设备所搭建的铜缆和光纤的广域网，或各种各样的无线网络。在物理链路层它们区别明显，但很多时候，在更高层次的 OSI 模型它们也存在差异。

Over the years there has been a convergence to the "internet stack" of IP and TCP/UDP. For example, Bluetooth defines physical layers and protocol layers, but on top of that is an IP stack so that the same internet programming techniques can be employed on many Bluetooth devices. Similarly, developing 4G wireless phone technologies such as LTE (Long Term Evolution) will also use an IP stack.

多年的发展，使得 IP 和 TCP/UDP 协议基本上就等价于网络协议栈。例如，蓝牙定义了物理层和协议层，但最重要的是 IP 协议栈，可以在许多蓝牙设备使相同的互联网编程技术。同样，开发 4G 无线手机技术，如 LTE（Long Term Evolution）也将使用 IP 协议栈。

While IP provides the networking layer 3 of the OSI stack, TCP and UDP deal with layer 4. These are not the final word, even in the interenet world: SCTP has come from the telecommunications to challenge both TCP and UDP, while to provide internet services in interplanetary space requires new, under development protocols such as DTN. Nevertheless, IP, TCP and UDP hold sway as principal networking technologies now and at least for a considerable time into the future. Go has full support for this style of programming

IP 提供了第 3 层的 OSI 网络协议栈，TCP 和 UDP 则提供了第 4 层。即使在因特网世界，这些都不是固定不变的：TCP 和 UDP 将面临来自 SCTP（STREAM CONTROL TRANSMISSION PROTOCOL 流控制传输协议）的挑战，同时在星际空间中提供互联网服务需要新的像正在开发的 DTN 协议。不过，IP, TCP 和 UDP 至少在当前甚至未来相当长的时间内是主要的网络技术。Go 语言提供了对这种编程的全面支持。

This chapter shows how to do TCP and UDP programming using Go, and how to use a raw socket for other protocols.

本章介绍如何使用 GO 编写 TCP 和 UDP 程序，以及如何使用其他协议的原始套接字。

The TCP/IP stack

TCP/IP 协议栈

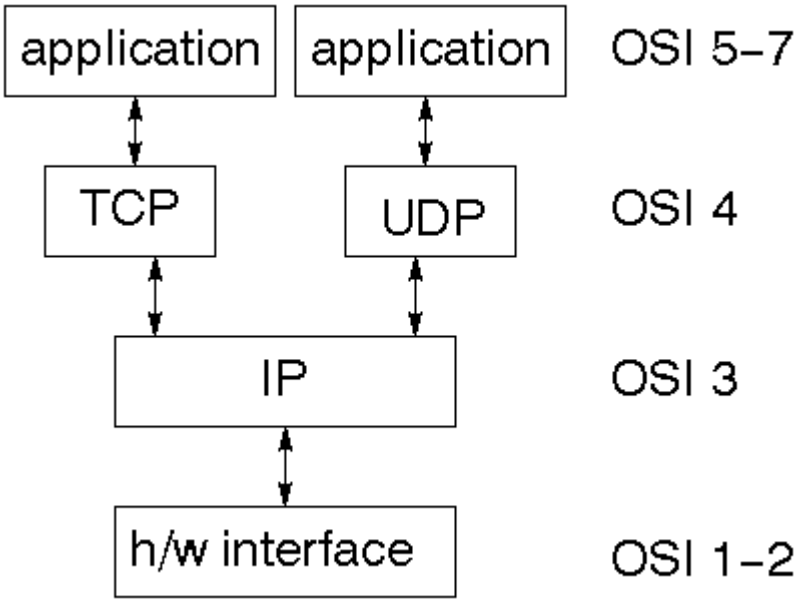
The OSI model was devised using a committee process wherein the standard was set up and then implemented. Some parts of the OSI standard are obscure, some parts cannot easily be implemented, some parts have not been implemented.

OSI 模型标准的建立和实施是一个委员会（国际标准化组织 ISO--译者注）设计的。OSI 标准中的一些部分是模糊的，有些部件不能很容易地实现，一些地方还没有得到落实。

The TCP/IP protocol was devised through a long-running DARPA project. This worked by implementation followed by RFCs (Request For Comment). TCP/IP is the principal Unix networking protocol. TCP/IP = Transmission Control Protocol/Internet Protocol.

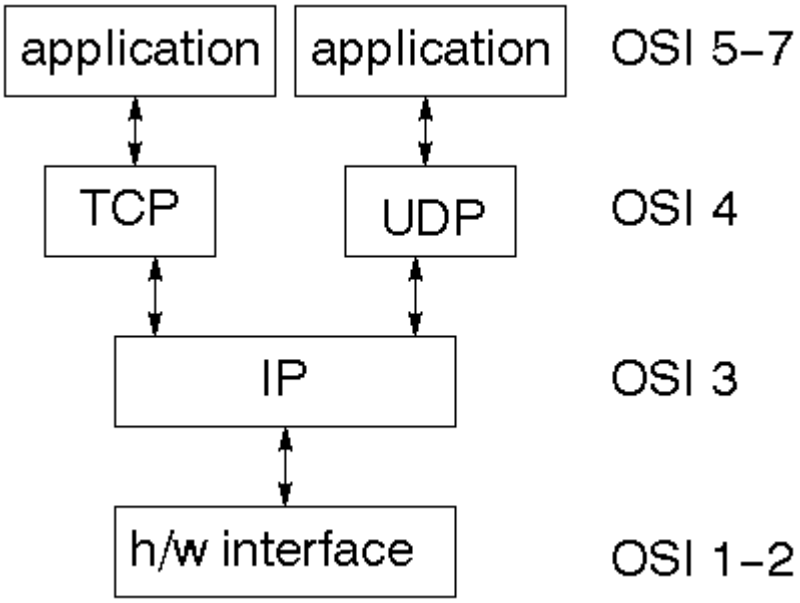
TCP/IP 协议由长期运行的一个 DARPA（美国国防先进研究项目局）项目设计。该工作其次由 RFC (Request For Comment)实施。TCP/IP 是 Unix 的首要网络协议。TCP/IP 等于传输控制协议/互联网协议。

The TCP/IP stack is shorter than the OSI one:



TCP is a connection-oriented protocol, UDP (User Datagram Protocol) is a connectionless protocol.

TCP/IP 协议栈是 OSI 模型的一部分：



TCP 是一个面向连接的协议，UDP（User Datagram Protocol，用户数据报协议）是一种无连接的协议。

IP datagrams

IP 数据包

The IP layer provides a connectionless and unreliable delivery system. It considers each datagram independently of the others. Any association between datagrams must be supplied by the higher layers.

IP 层提供了无连接的不可靠的传输系统，任何数据包之间的关联必须依赖更高的层来提供。

The IP layer supplies a checksum that includes its own header. The header includes the source and destination addresses.

IP 层包头支持数据校验，在包头包括源地址和目的地址。

The IP layer handles routing through an Internet. It is also responsible for breaking up large datagrams into smaller ones for transmission and reassembling them at the other end.

IP 层通过路由连接到因特网，还负责将大数据包分解为更小的包，并传输到另一端后进行重组。

UDP

UDP is also connectionless and unreliable. What it adds to IP is a checksum for the contents of the datagram and *port numbers*. These are used to give a client/server model - see later.

UDP 是无连接的，不可靠的。它包括 IP 数据报的内容和*端口号*的校验。在后面，我们会用它来构建一些客户端/服务器例子。

TCP

TCP supplies logic to give a reliable connection-oriented protocol above IP. It provides a *virtual circuit* that two processes can use to communicate. It also uses port numbers to identify services on a host.

TCP 是构建于 IP 之上的面向链接的协议。它提供了一个*虚电路*使得两个应用进程可以通过它来通信。它通过端口号来识别主机上的服务。

Internet addresses

互联网地址

In order to use a service you must be able to find it. The Internet uses an address scheme for devices such as computers so that they can be located. This addressing scheme was originally devised when there were only a handful of connected computers, and very generously allowed upto 2^32 addresses, using a 32 bit unsigned integer. These are the so-called IPv4 addresses. In recent years, the number of connected (or at least directly addressable) devices has threatened to exceed this number, and so "any day now" we will switch to IPv6 addressing which will allow upto 2^128 addresses, using an unsigned 128 bit integer. The changeover is most likely to be forced by emerging countries, as the developed world has already taken nearly all of the pool of IPv4 addresses.

要想使用一项服务，你必须先能找到它。互联网使用地址定位例如计算机的设备。这种寻址方案最初被设计出来只允许极少数的计算机连接上，使用 32 位无符号整形，拥有高达 2^32 个地址。这就是所谓的 IPv4 地址。近年来，连接（至少可以直接寻址）的设备的数量可能超过这个数字，所以在不久的某一天我们将切换到利用 128 位无符号整数，拥有高 2^128 个地址的 IPv6 寻址。这种转换最有可能被已经耗尽了所有的 IPv4 地址的新兴国家发达地区。

IPv4 addresses

IPv4 地址

The address is a 32 bit integer which gives the IP address. This addresses down to a network interface card on a single device. The address is usually written as four bytes in decimal with a dot '.' between them, as in "127.0.0.1" or "66.102.11.104".

IP 地址是一个 32 位整数构成。每个设备的网络接口都有一个地址。该地址通常使用'.'符号分割的 4 字节的十进制数，例如："127.0.0.1" 或 "66.102.11.104"。

The IP address of any device is generally composed of two parts: the address of the network in which the device resides, and the address of the device within that network. Once upon a time, the split between network address and internal address was simple and was based upon the bytes used in the IP address.

所有设备的 IP 地址，通常是由两部分组成：网段地址和网内地址。从前，网络地址和网内地址的分辨很简单，使用字节构建 IP 地址。

- In a class A network, the first byte identifies the network, while the last three identify the device. There are only 128 class A networks, owned by the very early players in the internet space such as IBM, the General Electric Company and MIT (http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml)
- Class B networks use the first two bytes to identify the network and the last two to identify devices within the subnet. This allows upto 2^16 (65,536) devices on a subnet

- Class C networks use the first three bytes to identify the network and the last one to identify devices within that network. This allows upto 2^8 (actually 254, not 256) devices

- 一个 A 类 IP 地址由 1 字节的网络地址和 3 字节主机地址组成。A 类网络只有 128 个，被很早的互联网成员例如 IBM，通用电气公司(the General Electric Company)和 MIT 所拥有。(http://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml)
- 一个 A 类 IP 地址由 1 字节的网络地址和 3 字节主机地址组成。这最多允许 2^16 (65,536)个设备在同一个子网。
- 一个 C 类 IP 地址由 3 字节的网络地址和 1 字节的主机地址组成。这最多允许 2^8 (其实是 254, 不是 256)个设备。

This scheme doesn't work well if you want, say, 400 computers on a network. 254 is too small, while 65,536 is too large. In binary arithmetic terms, you want about 512. This can be achieved by using a 23 bit network address and 9 bits for the device addresses. Similarly, if you want upto 1024 devices, you use a 22 bit network address and a 10 bit device address.

但是，比如你需要 400 台计算机在同一个网络，该方案是不可行的。254 太小，而 65,536 又太大。根据二进制计算，你大约需要 512(2^9，译者注)。这样就可以通过使用一个 23 位的网络地址和 9 位的设备地址实现。同样，如果您需要高达 1024 台设备，使用一个 22 位网络地址和一个 10 位的设备地址。

Given an IP address of a device, and knowing how many bits N are used for the network address gives a relatively straightforward process for extracting the network address and the device address within that network. Form a "network mask" which is a 32-bit binary number with all ones in the first N places and all zeroes in the remaining ones. For example, if 16 bits are used for the network address, the mask is 11111111111111110000000000000000. It's a little inconvenient using binary, so decimal bytes are usually used. The netmask for 16 bit network addresses is 255.255.0.0, for 24 bit network addresses it is 255.255.255.0, while for 23 bit addresses it would be 255.255.254.0 and for 22 bit addresses it would be 255.255.252.0.

知道设备的 IP 地址和多少字节用于网络地址，那么可以比较直接的提取出这个网络中的网络地址和设备地址。例如：“网络掩码”是一个前面 N 位为 1，其他所有位为 0 的 32 位二

进制数。例如，如果使用 16 位的网络地址，掩码为 11111111111111110000000000000000。使用二进制有一点不方便，所以通常使用十进制字节。16 位网络地址的子网掩码是 255.255.0.0，而对于 23 位地址，这将是 255.255.254.0，和 22 位地址，这将是 255.255.252.0。

Then to find the network of a device, bit-wise AND it's IP address with the network mask, while the device address within the subnet is found with bit-wise AND of the 1's complement of the mask with the IP address.

接着查找设备的网络，并将其 IP 地址与网络掩码进行按位与操作，而该设备在子网中的地址，可通过其 IP 地址同掩码与 1 的补码的按位与操作发现。

IPv6 addresses

IPv6 地址

The internet has grown vastly beyond original expectations. The initially generous 32-bit addressing scheme is on the verge of running out. There are unpleasant workarounds such as NAT addressing, but eventually we will have to switch to a wider address space. IPv6 uses 128-bit addresses. Even bytes becomes cumbersome to express such addresses, so hexadecimal digits are used, grouped into 4 digits and separated by a colon ':'. A typical address might be 2002:c0e8:82e7:0:0:0:c0e8:82e7.

因特网的迅速发展大大超出了原来的预期。最初富余的 32 位地址解决方案已经接近用完。虽然有一些例如 NAT 地址输入这样不是很完美的解决方法，但最终我们将不得不切换到更广阔的地址空间。IPv6 使用 128 位地址，即使表达同样的地址，字节数变得很麻烦，由':'分隔的 4 位 16 进制组成。一个典型的例子如：2002:c0e8:82e7:0:0:0:c0e8:82e7。

These addresses are not easy to remember! DNS will become even more important. There are tricks to reducing some addresses, such as eliding zeroes and repeated digits. For example, "localhost" is 0:0:0:0:0:0:0:1, which can be shortened to ::1

要记住这些地址并不容易！DNS 将变得更加重要。有一些技巧用来介绍一些地址，如省略一些零和重复的数字。例如："localhost"地址是：0:0:0:0:0:0:0:1，可以缩短到::1。

IP address type

IP 地址类型

The type IP

IP 类型

The package "net" defines many types, functions and methods of use in Go network programming. The type **IP** is defined as an array of bytes

"net"包定义了许多类型，函数，方法用于 Go 网络编程。**IP** 类型被定义为一个字节数组。

```
type IP []byte
```

There are several functions to manipulate a variable of type **IP**, but you are likely to use only some of them in practice. For example, the function **ParseIP(String)** will take a dotted IPv4 address or a colon IPv6 address, while the **IP** method **String** will return a string. Note that you may not get back what you started with: the string form of 0:0:0:0:0:0:0:1 is ::1.

有几个函数来处理一个 **IP** 类型的变量，但是在实践中你很可能只用到其中的一些。例如，**ParseIP(String)**函数将获取逗号分隔的 IPv4 或者冒号分隔的 IPv6 地址，而 **IP** 方法的**字符串**将返回一个字符串。请注意，您可能无法取回你期望的：字符串 0:0:0:0:0:0:0:1 是::1。

A program to illustrate this is

下面用一个程序来说明

```
/* IP
 */
```

```
package main

import (
	"net"
	"os"
	"fmt"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Fprintf(os.Stderr, "Usage: %s ip-addr\n", os.Args[0])
		os.Exit(1)
	}
	name := os.Args[1]

	addr := net.ParseIP(name)
	if addr == nil {
		fmt.Println("Invalid address")
	} else {
		fmt.Println("The address is ", addr.String())
	}
	os.Exit(0)
}
```

If this is compiled to the executable **IP** then it can run for example as

如果编译它为可执行文件 **IP**，那么它可以运行如

```
IP 127.0.0.1
```

```

```

with response

得到结果

```
The address is 127.0.0.1
```

or as

或

```
IP 0:0:0:0:0:0:0:1
```

得到结果

```
The address is ::1
```

The type IPmask

IP 掩码

In order to handle masking operations, there is the type

为了处理掩码操作，有下面类型：

```
type IPMask []byte
```

There is a function to create a mask from a 4-byte IPv4 address

下面这个函数用一个 4 字节的 IPv4 地址来创建一个掩码

```
func IPv4Mask(a, b, c, d byte) IPMask
```

Alternatively, there is a method of **IP** which returns the default mask

另外，这是一个 **IP** 的方法返回默认的掩码

```
func (ip IP) DefaultMask() IPMask
```

Note that the string form of a mask is a hex number such as ffff0000 for a mask of 255.255.0.0.

需要注意的是一个掩码的字符串形式是一个十六进制数，如掩码 255.255.0.0 为 ffff0000。

A mask can then be used by a method of an IP address to find the network for that IP address

一个掩码可以使用一个 IP 地址的方法，找到该 IP 地址的网络

```
func (ip IP) Mask(mask IPMask) IP
```

An example of the use of this is the following program:

下面的程序是一个使用了这个的例子：

```
/* Mask
 */

package main

import (
	"fmt"
	"net"
	"os"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Fprintf(os.Stderr, "Usage: %s dotted-ip-addr\n", os.Args[0])
		os.Exit(1)
	}
	dotAddr := os.Args[1]

	addr := net.ParseIP(dotAddr)
	if addr == nil {
		fmt.Println("Invalid address")
		os.Exit(1)
	}
	mask := addr.DefaultMask()
	network := addr.Mask(mask)
	ones, bits := mask.Size()
	fmt.Println("Address is ", addr.String(),
```

```
                        " Default mask length is ", bits,
                        "Leading ones count is ", ones,
                        "Mask is (hex) ", mask.String(),
                        " Network is ", network.String())
                os.Exit(0)
}
```

If this is compiled to **Mask** and run by

编译并运行 **Mask**

```
Mask 127.0.0.1
```

it will return

将返回

```
Address is   127.0.0.1   Default mask length is   8   Network is   127.0.0.0
```

The type IPAddr

IPAddr 类型

Many of the other functions and methods in the net package return a pointer to an **IPAddr**. This is simply a structure containing an **IP**.

在 net 包的许多函数和方法会返回一个指向 **IPAddr** 的指针。这不过只是一个包含 **IP** 类型的结构体。

```
```

```
type IPAddr {
        IP IP
}
```

A primary use of this type is to perform DNS lookups on IP host names.

这种类型的主要用途是通过 IP 主机名执行 DNS 查找。

```
func ResolveIPAddr(net, addr string) (*IPAddr, os.Error)
```

where net is one of "ip", "ip4" or "ip6". This is shown in the program

其中 net 是"ip","ip4"或者"ip6"的其中一个. 下面的程序中将会展示。

```
/* ResolveIP
 */

package main

import (
            "net"
            "os"
            "fmt"
)

func main() {
```

```
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s hostname\n", os.Args[0])
                fmt.Println("Usage: ", os.Args[0], "hostname")
                os.Exit(1)
        }
        name := os.Args[1]

        addr, err := net.ResolveIPAddr("ip", name)
        if err != nil {
                fmt.Println("Resolution error", err.Error())
                os.Exit(1)
        }
        fmt.Println("Resolved address is ", addr.String())
        os.Exit(0)
}
```

Running **ResolveIP www.google.com** returns

运行 **ResolveIP www.google.com** 返回

```
Resolved address is    66.102.11.104
```

Host lookup

主机查询

The function **ResolveIPAddr** will perform a DNS lookup on a hostname, and return a single IP address. However, hosts may have multiple IP addresses, usually from multiple network interface cards. They may also have multiple host names, acting as aliases.

ResolveIPAddr 函数将对某个主机名执行 DNS 查询，并返回一个简单的 IP 地址。然而，通常主机如果有多个网卡，则可以有多个 IP 地址。它们也可能有多个主机名，作为别名。

```
func LookupHost(name string) (cname string, addrs []string, err os.Error)
```

One of these addresses will be labelled as the "canonical" host name. If you wish to find the canonical name, use func LookupCNAME(name string) (cname string, err os.Error)

这些地址将会被归类为“canonical”主机名。如果你想找到的规范名称，使用 func LookupCNAME(name string) (cname string, err os.Error)

This is shown in the following program

下面是一个演示程序

```
/* LookupHost
 */

package main

import (
        "net"
        "os"
        "fmt"
)

func main() {
        if len(os.Args) != 2 {
```

```
                fmt.Fprintf(os.Stderr, "Usage: %s hostname\n", os.Args[0])
                os.Exit(1)
        }
        name := os.Args[1]

        addrs, err := net.LookupHost(name)
        if err != nil {
                fmt.Println("Error: ", err.Error())
                os.Exit(2)
        }

        for _, s := range addrs {
                fmt.Println(s)
        }
        os.Exit(0)
}
```

Note that this function returns strings, not **IPAddress** values.

注意，这个函数返回字符串，而不是 **IPAddress**。

Services

服务

Services run on host machines. They are typically long lived and are designed to wait for requests and respond to them. There are many types of services, and there are many ways in which they can offer their services to clients. The internet world bases many of these services on two methods of communication, TCP and UDP, although there are other communication protocols such as SCTP waiting in the wings to take over. Many other types of service, such as peer-to-peer, remote procedure calls, communicating agents, and many others are built on top of TCP and UDP.

服务运行在主机。它们通常长期存活，同时被设计成等待的请求和响应请求。有许多类型的服务，有他们能够通过各种方法向客户提供服务。互联网的世界基于 TCP 和 UDP 这两种通信方法提供许多这些服务，虽然也有其他通信协议如 SCTP 伺机取代。许多其他类型的服务，例如点对点，远过程调用，通信代理，和许多其他建立在 TCP 和 UDP 之上的服务之上。

Ports

端口

Services live on host machines. The IP address will locate the host. But on each computer may be many services, and a simple way is needed to distinguish between them. The method used by TCP, UDP, SCTP and others is to use a *port number*. This is an unsigned integer beween 1 and 65,535 and each service will associate itself with one or more of these port numbers.

服务存活于主机内。IP 地址可以定位主机。但在每台计算机上可能会提供多种服务，需要一个简单的方法对它们加以区分。TCP，UDP，SCTP 或者其他协议使用*端口号*来加以区分。这里使用一个 1 到 65,535 的无符号整数，每个服务将这些端口号中的一个或多个相关联。

There are many "standard" ports. Telnet usually uses port 23 with the TCP protocol. DNS uses port 53, either with TCP or with UDP. FTP uses ports 21 and 20, one for commands, the other for data transfer. HTTP usually uses port 80, but it often uses ports 8000, 8080 and 8088, all with TCP. The X Window System often takes ports 6000-6007, both on TCP and UDP.

有很多“标准”的端口。Telnet 服务通常使用端口号 23 的 TCP 协议。DNS 使用端口号 53 的 TCP 或 UDP 协议。FTP 使用端口 21 和 20 的命令，进行数据传输。HTTP 通常使用端口 80，但经常使用，端口 8000，8080 和 8088，协议为 TCP。X Window 系统往往需要端口 6000-6007，TCP 和 UDP 协议。

On a Unix system, the commonly used ports are listed in the file **/etc/services**. Go has a function to interrogate this file

在 Unix 系统中，**/etc/services** 文件列出了常用的端口。Go 语言有一个函数可以获取该文件。

```
func LookupPort(network, service string) (port int, err os.Error)
```

The network argument is a string such as "tcp" or "udp", while the service is a string such as "telnet" or "domain" (for DNS).

network 是一个字符串例如"tcp"或"udp", service 也是一个字符串, 如"telnet"或"domain"(DNS)。

A program using this is

示例程序如下

```
/* LookupPort
 */

package main

import (
        "net"
        "os"
        "fmt"
)

func main() {
        if len(os.Args) != 3 {
                fmt.Fprintf(os.Stderr,
                        "Usage: %s network-type service\n",
                        os.Args[0])
                os.Exit(1)
```

```
        }
        networkType := os.Args[1]
        service := os.Args[2]

        port, err := net.LookupPort(networkType, service)
        if err != nil {
                fmt.Println("Error: ", err.Error())
                os.Exit(2)
        }

        fmt.Println("Service port ", port)
        os.Exit(0)
}
```

For example, running LookupPort tcp telnet prints Service port: 23

举个例子，运行 LookupPort tcp telnet 打印 Service port: 23

The type TCPAddr

TCPAddr 类型

The type **TCPAddr** is a structure containing an **IP** and a **port**:

TCPAddr 类型包含一个 **IP** 和一个 **port** 的结构:

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

The function to create a **TCPAddr** is **ResolveTCPAddr**

函数 **ResolveTCPAddr** 用来创建一个 **TCPAddr**

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

where **net** is one of "tcp", "tcp4" or "tcp6" and the **addr** is a string composed of a host name or IP address, followed by the port number after a ":", such as "www.google.com:80" or '127.0.0.1:22". if the address is an IPv6 address, which already has colons in it, then the host part must be enclosed in square brackets, such as "[::1]:23". Another special case is often used for servers, where the host address is zero, so that the TCP address is really just the port name, as in ":80" for an HTTP server.

net 是"tcp", "tcp4"或"tcp6"其中之一，**addr** 是一个字符串，由主机名或 IP 地址，以及":"后跟随着端口号组成，例如： "www.google.com:80" 或 '127.0.0.1:22"。如果地址是一个 IPv6 地址，由于已经有冒号，主机部分，必须放在方括号内，例如："[::1]:23". 另一种特殊情况是经常用于服务器，主机地址为 0，因此，TCP 地址实际上就是端口名称，例如："":80" 用来表示 HTTP 服务器。

TCP Sockets

TCP 套接字

When you know how to reach a service via its network and port IDs, what then? If you are a client you need an API that will allow you to connect to a service and then to send messages to that service and read replies back from the service.

当你知道如何通过网络和端口 ID 查找一个服务时，然后呢？如果你是一个客户端，你需要一个 API，让您连接到服务，然后将消息发送到该服务，并从服务读取回复。

If you are a server, you need to be able to bind to a port and listen at it. When a message comes in you need to be able to read it and write back to the client.

如果你是一个服务器，你需要能够绑定到一个端口，并监听它。当有消息到来，你需要能够读取它并回复客户端。

The **net.TCPConn** is the Go type which allows full duplex communication between the client and the server. Two major methods of interest are

net.TCPConn 是允许在客户端和服务器之间的全双工通信的 Go 类型。两种主要方法是

```
func (c *TCPConn) Write(b []byte) (n int, err os.Error)
func (c *TCPConn) Read(b []byte) (n int, err os.Error)
```

A **TCPConn** is used by both a client and a server to read and write messages.

TCPConn 被客户端和服务器用来读写消息。/p>

TCP client

TCP 客户端

Once a client has established a TCP address for a service, it "dials" the service. If succesful, the dial returns a **TCPConn** for communication. The client and the server exchange messages on this. Typically a client writes a request to the server using the **TCPConn**, and reads a response from the **TCPConn**. This continues until either (or both) sides close the connection. A TCP connection is established by the client using the function

一旦客户端已经建立 TCP 服务，就可以和对方设备"通话"了．如果成功，该调用返回一个用于通信的 **TCPConn**。客户端和服务器通过它交换消息。通常情况下，客户端使用 **TCPConn** 写入请求到服务器，并从 **TCPConn** 的读取响应。持续如此，直到任一（或两者）的两侧关闭连接。客户端使用该函数建立一个 TCP 连接。

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

where **laddr** is the local address which is usually set to **nil** and **raddr** is the remote address of the service, and the **net** string is one of "tcp4", "tcp6" or "tcp" depending on whether you want a TCPv4 connection, a TCPv6 connection or don't care.

其中 **laddr** 是本地地址，通常设置为 **nil** 和 **raddr** 是一个服务的远程地址, **net** 是一个字符串，根据您是否希望是一个 TCPv4 连接，TCPv6 连接来设置为"tcp4", "tcp6"或"tcp"中的一个，当然你也可以不关心链接形式。

A simple example can be provided by a client to a web (HTTP) server. We will deal in substantially more detail with HTTP clients and servers in a later chapter, but for now we will keep it simple.

一个简单的例子，展示个客户端连接到一个网页(HTTP)服务器。在后面的章节，我们将处理大量的 HTTP 客户端和服务器细节，现在我们先从简单的看看。

One of the possible messages that a client can send is the "HEAD" message. This queries a server for information about the server and a document on that server. The server returns information, but does not return the document itself. The request sent to query an HTTP server could be

客户端可能发送的消息之一就是“HEAD”消息。这用来查询服务器的信息和文档信息。服务器返回的信息，不返回文档本身。发送到服务器的请求可能是

```
"HEAD / HTTP/1.0\r\n\r\n"
```

which asks for information about the root document and the server. A typical response might be

这是在请求服务器的根文件信息。 一个典型的响应可能是

```
```

```
HTTP/1.0 200 OK
ETag: "-9985996"
Last-Modified: Thu, 25 Mar 2010 17:51:10 GMT
Content-Length: 18074
Connection: close
Date: Sat, 28 Aug 2010 00:43:48 GMT
Server: lighttpd/1.4.23
```

We first give the program (GetHeadInfo.go) to establish the connection for a TCP address, send the request string, read and print the response. Once compiled it can be invoked by e.g.

我们首先通过(GetHeadInfo.go)程序来建立 TCP 连接，发送请求字符串，读取并打印响应。编译后就可以调用，例如：

```
GetHeadInfo www.google.com:80
```

The program is

程序

```
/* GetHeadInfo
 */
package main

import (
        "net"
        "os"
        "fmt"
```

```
        "io/ioutil"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s host:port ", os.Args[0])
                os.Exit(1)
        }
        service := os.Args[1]

        tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
        checkError(err)

        conn, err := net.DialTCP("tcp", nil, tcpAddr)
        checkError(err)

        _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
        checkError(err)

        //result, err := readFully(conn)
 result, err := ioutil.ReadAll(conn)
        checkError(err)

        fmt.Println(string(result))

        os.Exit(0)
}

func checkError(err error) {
```

```
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

The first point to note is the almost excessive amount of error checking that is going on. This is normal for networking programs: the opportunities for failure are substantially greater than for standalone programs. Hardware may fail on the client, the server, or on any of the routers and switches in the middle; communication may be blocked by a firewall; timeouts may occur due to network load; the server may crash while the client is talking to it. The following checks are performed:

第一个要注意的点是近乎多余的错误检查。因为正常情况下，网络程序失败的机会大大超过单机的程序。在客户端，服务器端或任何路由和中间交换上，硬件可能失败；通信可能会被防火墙阻塞;因网络负载可能会出现超时;当客户端联系服务器，服务器可能会崩溃，下列检查是必须的：

1. There may be syntax errors in the address specified
2. The attempt to connect to the remote service may fail. For example, the service requested might not be running, or there may be no such host connected to the network
3. Although a connection has been established, writes to the service might fail if the connection has died suddenly, or the network times out
4. Similarly, the reads might fail

1. 指定的地址中可能存在语法错误
2. 尝试连接到远程服务可能会失败。例如，所请求的服务可能没有运行，或者有可能是主机没有连接到网络
3. 虽然连接已经建立，如果连接突然丢失也可能导致写失败，或网络超时
4. 同样，读操作也可能会失败

Reading from the server requires a comment. In this case, we read essentially a single response from the server. This will be terminated by end-of-file on the connection. However, it may consist of several TCP packets, so we need to keep reading till the end of file. The **io/ioutil** function **ReadAll** will look after these issues and return the complete response. (Thanks to Roger Peppe on the golang-nuts mailing list.).

值得一提的是,如何从服务端读取数据。在这种情况下，读本质上是一个单一的来自服务器的响应，这将终止文件结束的连接。但是，它可能包括多个 TCP 数据包，所以我们需要不断地读，直到文件的末尾。在 **io/ioutil** 下的 **ReadAll** 函数考虑这些问题，并返回完整响应。(感谢 Roger Peppe 在 golang-nuts 上的邮件列表。)。

There are some language issues involved. First, most of the functions return a dual value, with possible error as second value. If no error occurs, then this will be **nil**. In C, the same behaviour is gained by special values such as **NULL**, or -1, or zero being returned - if that is possible. In Java, the same error checking is managed by throwing and catching exceptions, which can make the code look very messy.

有一些涉及语言的问题，首先，大多数函数返回两个值，第二个值是可能出现的错误。如果没有错误发生，那么它的值为 **nil**。在 C 中，如果需要的话，同样的行为通过定义特殊值例如 **NULL**，或 -1，或 0 来返回。在 Java 中，同样的错误检查通过抛出和捕获异常来管理，它会使代码看起来很凌乱。

In earlier versions of this program, I returned the result in the array **buf**, which is of type **[512]byte**. Attempts to coerce this to a string failed - only byte arrays of type **[]byte** can be coerced. This is a bit of a nuisance.

在这个程序的早期版本，我在返回结果中返回 **buf** 数组，它的类型是**[512]byte**。我试图强迫类型为一个字符串但失败了- 只有字节数组类型**[]byte** 可以强制转换。这确实有点困扰。

A Daytime server

一个时间(Daytime)服务器

About the simplest service that we can build is the daytime service. This is a standard Internet service, defined by RFC 867, with a default port of 13, on both TCP and UDP. Unfortunately, with the (justified) increase in paranoia over security, hardly any sites run a daytime server any more. Never mind, we can build our own. (For those interested, if you install **inetd** on your system, you usually get a daytime server thrown in.)

最简单的服务，我们可以建立是时间(Daytime)服务。这是一个标准的互联网服务，由 RFC 867 定义，默认的端口 13，协议是 TCP 和 UDP。很遗憾，对安全的偏执，几乎没有任何站点运行着时间(Daytime)服务器。不过没关系，我们可以建立我们自己的。（对于那些有兴趣，你可以在你的系统安装 **inetd**，你通常可以得到一个时间(Daytime)服务器。）

A server registers itself on a port, and listens on that port. Then it blocks on an "accept" operation, waiting for clients to connect. When a client connects, the accept call returns, with a connection object. The daytime service is very simple and just writes the current time to the client, closes the connection, and resumes waiting for the next client.

在一个服务器上注册并监听一个端口。然后它阻塞在一个"accept"操作，并等待客户端连接。当一个客户端连接，accept 调用返回一个连接(connection)对象。时间(Daytime)服务非常简单，只是将当前时间写入到客户端，关闭该连接，并继续等待下一个客户端。

The relevant calls are

有关调用

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

The argument **net** can be set to one of the strings "tcp", "tcp4" or "tcp6". The IP address should be set to zero if you want to listen on all network interfaces, or to the IP address of a single network interface if you only want to listen on that interface. If the port is set to zero, then the O/S will

choose a port for you. Otherwise you can choose your own. Note that on a Unix system, you cannot listen on a port below 1024 unless you are the system supervisor, root, and ports below 128 are standardised by the IETF. The example program chooses port 1200 for no particular reason. The TCP address is given as ":1200" - all interfaces, port 1200.

net 参数可以设置为字符串"tcp", "tcp4"或者"tcp6"中的一个。如果你想监听所有网络接口，IP 地址应设置为 0，或如果你只是想监听一个简单网络接口，IP 地址可以设置为该网络的地址。如果端口设置为 0，O/S 会为你选择一个端口。否则，你可以选择你自己的。需要注意的是，在 Unix 系统上，除非你是监控系统，否则不能监听低于 1024 的端口，小于 128 的端口是由 IETF 标准化。该示例程序选择端口 1200 没有特别的原因。TCP 地址如下":1200" - 所有网络接口，端口 1200。

The program is

程序

```
/* DaytimeServer
 */
package main

import (
        "fmt"
        "net"
        "os"
        "time"
)

func main() {

        service := ":1200"
```

```
        tcpAddr, err := net.ResolveTCPAddr("ip4", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }

                daytime := time.Now().String()
                conn.Write([]byte(daytime)) // don't care about return value
          conn.Close()                    // we're finished with this client
 }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

If you run this server, it will just wait there, not doing much. When a client connects to it, it will respond by sending the daytime string to it and then return to waiting for the next client.

如果你运行该服务器，它会在那里等待， 没有做任何事。当一个客户端连接到该服务器，它会响应发送时间(Daytime)字符串，然后继续等待下一个客户端。

Note the changed error handling in the server as compared to a client. The server should run forever, so that if any error occurs with a client, the server just ignores that client and carries on. A client could otherwise try to mess up the connection with the server, and bring it down!

相比客户端服务器更要注意对错误的处理。服务器应该永远运行，所以，如果出现任何错误与客户端，服务器只是忽略客户端继续运行。否则，客户端可以尝试搞砸了与服务器的连接，并导致服务器宕机。

We haven't built a client. That is easy, just changing the previous client to omit the initial write. Alternatively, just open up a **telnet** connection to that host:

我们还没有建立一个客户端。这很简单，只是改变以前的客户端省略的初始写入。另外，只需打开一个 **telnet** 连接到该主机：

```
telnet localhost 1200
```

This will produce output such as

输出如下：

```
$telnet localhost 1200
Trying ::1...
Connected to localhost.
Escape character is '^]'.
Sun Aug 29 17:25:19 EST 2010Connection closed by foreign host.
```

where "Sun Aug 29 17:25:19 EST 2010" is the output from the server.

服务器输出："Sun Aug 29 17:25:19 EST 2010"。

Multi-threaded server

多线程服务器

"echo" is another simple IETF service. This just reads what the client types, and sends it back:

"echo"是另一种简单的 IETF 服务。只是读取客户端数据，并将其发送回去:

```
/* SimpleEchoServer
 */
package main

import (
	"net"
	"os"
	"fmt"
)

func main() {

	service := ":1201"
	tcpAddr, err := net.ResolveTCPAddr("tcp4", service)
	checkError(err)

	listener, err := net.ListenTCP("tcp", tcpAddr)
	checkError(err)

	for {
		conn, err := listener.Accept()
		if err != nil {
```

```
                        continue
                }
                handleClient(conn)
                conn.Close() // we're finished
        }
}

func handleClient(conn net.Conn) {
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                if err != nil {
                        return
                }
                fmt.Println(string(buf[0:]))
                _, err2 := conn.Write(buf[0:n])
                if err2 != nil {
                        return
                }
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

While it works, there is a significant issue with this server: it is single-threaded. While a client has a connection open to it, no other cllient can connect. Other clients are blocked, and will probably time out. Fortunately this is easly fixed by making the client handler a go-routine. We have also moved the connection close into the handler, as it now belongs there

工作时，此服务器有一个明显的问题：它是单线程的。当有一个客户端连接到它，就没有其他的客户端可以连接上。其他客户端将被阻塞，可能会超时。幸好客户端很容易使用 go-routine 扩展。我们仅仅需要把连接关闭移到处理程序结束后，示例代码如下：

```
/* ThreadedEchoServer
 */
package main

import (
        "net"
        "os"
        "fmt"
)

func main() {

        service := ":1201"
        tcpAddr, err := net.ResolveTCPAddr("ip4", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
```

```
                        conn, err := listener.Accept()
                        if err != nil {
                                continue
                        }
                        // run as a goroutine
                go handleClient(conn)
            }
}

func handleClient(conn net.Conn) {
            // close connection on exit
  defer conn.Close()

            var buf [512]byte
            for {
                        // read upto 512 bytes
                n, err := conn.Read(buf[0:])
                        if err != nil {
                                return
                        }

                        // write the n bytes read
                _, err2 := conn.Write(buf[0:n])
                        if err2 != nil {
                                return
                        }
            }
}
```

```
func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

Controlling TCP connections

控制 TCP 连接

Timeout

超时

The server may wish to timeout a client if it does not respond quickly enough i.e. does not write a request to the server in time. This should be a long period (several minutes), because the user may be taking their time. Conversely, the client may want to timeout the server (after a much shorter time). Both do this by

服务端会断开那些超时的客户端，如果他们响应不够快，比如没有及时往服务端写一个请求。这应该是长时间(几分钟)的，因为用户可能花费了时间。相反，客户端可能希望超时服务器(一个更短的时间后)。通过下面的来实现这两种：

```
func (c *TCPConn) SetTimeout(nsec int64) os.Error
```

before any reads or writes on the socket.

套接字读写前。

Staying alive

存活状态

A client may wish to stay connected to a server even if it has nothing to send. It can use

即使没有任何通信，一个客户端可能希望保持连接到服务器的状态。可以使用

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

There are several other connection control methods, documented in the "net" package.

还有几个其他的连接控制方法，可以查看"net"包。

UDP Datagrams

UDP 数据报

In a connectionless protocol each message contains information about its origin and destination. There is no "session" established using a long-lived socket. UDP clients and servers make use of datagrams, which are individual messages containing source and destination information. There is no state maintained by these messages, unless the client or server does so. The messages are not guaranteed to arrive, or may arrive out of order.

在一个无连接的协议中，每个消息都包含了关于它的来源和目的地的信息。没有"session"建立在使用长寿命的套接字。UDP 客户端和服务器使用的数据包，单独包含来源和目的地的信息。除非客户端或服务器这样做，否则消息的状态不会保持。这些消息不能保证一定到达，也可能保证按顺序到达。

The most common situation for a client is to send a message and hope that a reply arrives. The most common situation for a server would be to receive a message and then send one or more replies back to that client. In a peer-to-peer situation, though, the server may just forward messages to other peers.

客户端最常见的情况发送消息，并希望响应正常到达。服务器最常见的情况为将收到一条消息，然后发送一个或多个回复给客户端。而在点对点的情况下， 服务器可能仅仅是把消息转发到其他点。

The major difference between TCP and UDP handling for Go is how to deal with packets arriving from possibly multiple clients, without the cushion of a TCP session to manage things. The major calls needed are

Go 下处理 TCP 和 UDP 之间的主要区别是如何处理多个客户端可能同时有数据包到达，没有一个管理 TCP 会话的缓冲。主要需要调用的是

```
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)
```

The client for a UDP time service doesn't need to make many changes, just changing ...TCP... calls to ...UDP... calls:

UDP 时间服务的客户端并不需要做很多的变化，仅仅改变...TCP...调用为...UDP...调用:

```
/* UDPDaytimeClient
 */
package main

import (
        "net"
        "os"
        "fmt"
)

func main() {
```

```
        if len(os.Args) != 2 {
                fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
                os.Exit(1)
        }
        service := os.Args[1]

        udpAddr, err := net.ResolveUDPAddr("up4", service)
        checkError(err)

        conn, err := net.DialUDP("udp", nil, udpAddr)
        checkError(err)

        _, err = conn.Write([]byte("anything"))
        checkError(err)

        var buf [512]byte
        n, err := conn.Read(buf[0:])
        checkError(err)

        fmt.Println(string(buf[0:n]))

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error ", err.Error())
                os.Exit(1)
        }
```

```
}
```

while the server has to make a few more:

服务器也有很少的改动:

```
/* UDPDaytimeServer
 */
package main

import (
	"fmt"
	"net"
	"os"
	"time"
)

func main() {

	service := ":1200"
	udpAddr, err := net.ResolveUDPAddr("up4", service)
	checkError(err)

	conn, err := net.ListenUDP("udp", udpAddr)
	checkError(err)

	for {
		handleClient(conn)
	}
}
```

```
func handleClient(conn *net.UDPConn) {

        var buf [512]byte

        _, addr, err := conn.ReadFromUDP(buf[0:])
        if err != nil {
                return
        }

        daytime := time.Now().String()

        conn.WriteToUDP([]byte(daytime), addr)
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Server listening on multiple sockets

服务器侦听多个套接字

A server may be attempting to listen to multiple clients not just on one port, but on many. In this case it has to use some sort of polling mechanism between the ports.

一个服务器可能不止在一个端口监听多个客户端，或是更多端口，在这种情况下，它在端口之间使用某种轮询机制。

In C, the select() call lets the kernel do this work. The call takes a number of file descriptors. The process is suspended. When I/O is ready on one of these, a wakeup is done, and the process can continue. This is cheaper than busy polling. In Go, accomplish the same by using a different goroutine for each port. A thread will become runnable when the lower-level select() discovers that I/O is ready for this thread.

在 C 中，调用的内核 select()可以完成这项工作。 调用需要一个文件描述符的数字。该进程被暂停。当 I/O 准备好其中一个，一个唤醒被完成，并且该过程可以继续。This is cheaper than busy polling. 在 G 中，完成相同的功能，通过为每个端口使用一个不同的 goroutine。低级别的 select()时发现，I/O 已经准备好该线程，一个线程将运行。

The types Conn, PacketConn and Listener

Conn，PacketConn 和 Listener 类型

So far we have differentiated between the API for TCP and the API for UDP, using for example **DialTCP** and **DialUDP** returning a **TCPConn** and **UDPConn** respectively. The type **Conn** is an interface and both **TCPConn** and **UDPConn** implement this interface. To a large extent you can deal with this interface rather than the two types.

迄今为止我们已经区分 TCP 和 UDP API 的不同，使用例子 **DialTCP** 和 **DialUDP** 分别返回一个 **TCPConn** 和 **UDPConn**。**Conn** 类型是一个接口，**TCPConn** 和 **UDPConn** 实现了该接口。在很大程度上，你可以通过该接口处理而不是用这两种类型。

Instead of separate dial functions for TCP and UDP, you can use a single function

你可以使用一个简单的函数，而不是单独使用 TCP 和 UDP 的 dial 函数。

```
func Dial(net, laddr, raddr string) (c Conn, err os.Error)
```

The **net** can be any of "tcp", "tcp4" (IPv4-only), "tcp6" (IPv6-only), "udp", "udp4" (IPv4-only), "udp6" (IPv6-only), "ip", "ip4" (IPv4-only) and "ip6" (IPv6-only). It will return an appropriate implementation of the **Conn** interface. Note that this function takes a string rather than address as **raddr** argument, so that programs using this can avoid working out the address type first.

net 可以是"tcp", "tcp4" (IPv4-only), "tcp6" (IPv6-only), "udp", "udp4" (IPv4-only), "udp6" (IPv6-only), "ip", "ip4" (IPv4-only)和"ip6" (IPv6-only)任何一种。它将返回一个实现了 **Conn** 接口的类型。注意此函数接受一个字符串而不是 **raddr** 地址参数，因此，使用此程序可避免的地址类型。

Using this function makes minor changes to programs. For example, the earlier program to get HEAD information from a Web page can be re-written as

使用该函数需要对程序轻微的调整。例如，前面的程序从一个 Web 页面获取 HEAD 信息可以被重新写为

```
/* IPGetHeadInfo
 */
package main

import (
        "bytes"
        "fmt"
        "io"
        "net"
        "os"
)

func main() {
        if len(os.Args) != 2 {
```

```
                fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := net.Dial("tcp", service)
        checkError(err)

        _, err = conn.Write([]byte("HEAD / HTTP/1.0\r\n\r\n"))
        checkError(err)

        result, err := readFully(conn)
        checkError(err)

        fmt.Println(string(result))

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()
```

```
        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

Writing a server can be similarly simplified using the function

使用该函数同样可以简化一个服务器的编写

```
func Listen(net, laddr string) (l Listener, err os.Error)
```

which returns an object implementing the **Listener** interface. This interface has a method

返回一个实现 **Listener** 接口的对象. 该接口有一个方法

```
func (l Listener) Accept() (c Conn, err os.Error)
```

which will allow a server to be built. Using this, the multi-threaded Echo server given earlier becomes

这将允许构建一个服务器。使用它，将使前面给出的多线程 Echo 服务器改变

```
/* ThreadedIPEchoServer
 */
package main

import (
        "fmt"
        "net"
        "os"
)

func main() {

        service := ":1200"
        listener, err := net.Listen("tcp", service)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }
                go handleClient(conn)
        }
}
```

```
func handleClient(conn net.Conn) {
        defer conn.Close()

        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                if err != nil {
                        return
                }
                _, err2 := conn.Write(buf[0:n])
                if err2 != nil {
                        return
                }
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

If you want to write a UDP server, then there is an interface **PacketConn** and a method to return an implementation of this:

如果你想写一个 UDP 服务器，这里有一个 **PacketConn** 的接口，和一个实现了该接口的方法:

```
func ListenPacket(net, laddr string) (c PacketConn, err os.Error)
```

This interface has primary methods **ReadFrom** and **WriteTo** to handle packet reads and writes.

这个接口的主要方法 **ReadFrom** 和 **WriteTo** 用来处理数据包的读取和写入。

The Go **net** package recommends using these interface types rather than the concrete ones. But by using them, you lose specific methods such as **SetKeepAlive** or **TCPConn** and **SetReadBuffer** of **UDPConn**, unless you do a type cast. It is your choice.

Go 的 **net** 包建议使用接口类型而不是具体的实现类型。但是，通过使用它们，你失去了具体的方法，比如 **SetKeepAlive** 或 **TCPConn** 和 **UDPConn** 的 **SetReadBuffer**，除非你做一个类型转换。如何选择在于你。

Raw sockets and the type IPConn

原始套接字和 IPConn 类型

This section covers advanced material which most programmers are unlikely to need. it deals with *raw sockets*, which allow the programmer to build their own IP protocols, or use protocols other than TCP or UDP

本节涵盖了大多数程序员可能需要的高级资料。它涉及 *raw sockets*，，允许程序员建立自己的 IP 协议，或使用 TCP 或 UDP 协议。

TCP and UDP are not the only protocols built above the IP layer. The site http://www.iana.org/assignments/protocol-numbers lists about 140 of them (this list is often available on Unix systems in the file **/etc/protocols**). TCP and UDP are only numbers 6 and 17 respectively on this list.

TCP 和 UDP 并不是建立在 IP 层之上唯一的协议。该网站：http://www.iana.org/assignments/protocol-numbers 列表上大约有 140 关于它们(该列表往往在 Unix 系统的**/etc/protocols** 文件上。)。TCP 和 UDP 在这个名单上分别为 6 和 17。

Go allows you to build so-called raw sockets, to enable you to communicate using one of these other protocols, or even to build your own. But it gives minimal support: it will connect hosts, and write

and read packets between the hosts. In the next chapter we will look at designing and implementing your own protocols above TCP; this section considers the same type of problem, but at the IP layer.

Go 允许你建立所谓的原始套接字，使您可以使用这些其它协议通信，或甚至建立你自己的。但它提供了最低限度的支持：它会连接主机，写入和读取和主机之间的数据包。在接下来的章节中，我们将着眼于设计和实现自己的基于 TCP 之上的协议；这部分认为同样的问题存在于 IP 层。

To keep things simple, we shall use almost the simplest possible example: how to send a ping message to a host. Ping uses the "echo" command from the ICMP protocol. This is a byte-oriented protocol, in which the client sends a stream of bytes to another host, and the host replies. the format is:

为了简单起见，我们将使用几乎最简单的例子：如何发送一个 ping 消息给主机。Ping 使用"echo"命令的 ICMP 协议。这是一个面向字节协议，客户端发送一个字节流到另一个主机，并等待主机的答复。格式如下：

- The first byte is 8, standing for the echo message
- The second byte is zero
- The third and fourth bytes are a checksum on the entire message
- The fifth and sixth bytes are an arbitrary indentifier
- The seventh and eight bytes are an arbitrary sequence number
- The rest of the packet is user data

- 首字节是 8，表示 echo 消息
- 第二个字节是 0
- 第三和第四字节是整个消息的校验和
- 第五和第六字节是一个任意标识
- 第七和第八字节是一个任意的序列号
- 该数据包的其余部分是用户数据

The following program will prepare an IP connection, send a ping request to a host and get a reply. You may need to have root access in order to run it successfully.

下面的程序将准备一个 IP 连接，发送一个 ping 请求到主机，并得到答复。您可能需要 root 权限才能运行成功。

```
/* Ping
 */
package main

import (
        "bytes"
        "fmt"
        "io"
        "net"
        "os"
)

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)
        checkError(err)
```

```
        var msg [512]byte
        msg[0] = 8    // echo
msg[1] = 0    // code 0
msg[2] = 0    // checksum, fix later
msg[3] = 0    // checksum, fix later
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])
        checkError(err)

        _, err = conn.Read(msg[0:])
        checkError(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 checkError(err error) {
            if err != nil {
                        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                        os.Exit(1)
            }
}

func readFully(conn net.Conn) ([]byte, error) {
            defer conn.Close()

            result := bytes.NewBuffer(nil)
            var buf [512]byte
```

```
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

Conclusion

结论

This chapter has considered programming at the IP, TCP and UDP levels. This is often necessary if you wish to implement your own protocol, or build a client or server for an existing protocol.

本章着重 IP, TCP 和 UDP 级别的编程。如果你想实现自己的协议，或用现有的协议建立一个客户端或服务器，这些内容往往很重要。

Data serialisation

数据序列化

Communication between a client and a service requires the exchange of data. This data may be highly structured, but has to be serialised for transport. This chapter looks at the basics of serialisation and then considers several techniques supported by Go APIs.

客户端与服务之间通过数据交换来通信。因为数据可能是高度结构化的，所以在传输前必须进行序列化。这一章将研究序列化基础并介绍一些 Go API 提供的序列化技术。

Introduction

简介

A client and server need to exchange information via messages. TCP and UDP provide the transport mechanisms to do this. The two processes also have to have a protocol in place so that message exchange can take place meaningfully.

客户端与服务器需要通过消息来交换信息。TCP 与 UDP 是消息传递的两种机制，在这两种机制之上就需要有合适的协议来约定传输的内容的含义。

Messages are sent across the network as a sequence of bytes, which has no structure except for a linear stream of bytes. We shall address the various possibilities for messages and the protocols that define them in the next chapter. In this chapter we concentrate on a component of messages - the data that is transferred.

在网络上，消息被当作字节序列来传输，它们是没有结构的，仅仅只是一串字节流。我们将在下一章讨论定义消息与协议涉及到的的各种问题。本章，我们只重点关注消息的一个方面 - 被传输的数据

A program will typically build complex data structures to hold the current program state. In conversing with a remote client or service, the program will be attempting to transfer such data structures across the network - that is, outside of the application's own address space.

程序通常构造一个复杂的数据结构来保存其自身当前的状态。在与远程的客户端或服务的交互中，程序会通过网络将这样的数据结构传输到 -应用程序所在的地址空间之外的地方

Programming languages use structured data such as

编程语言使用的结构化的数据类型有

- records/structures
- variant records
- array - fixed size or varying
- string - fixed size or varying
- tables - e.g. arrays of records
- non-linear structures such as
    - circular linked list
    - binary tree
    - objects with references to other objects

- 记录/结构
- 可变记录
- 数组 - 固定大小或可变大小
- 字符串 - 固定大小或可变大小
- 表 - 例如:记录构成的数组
- 非线程结构，比如
    - 循环链表
    - 二叉树
    - 含有其他对象引用的对象

None of IP, TCP or UDP packets know the meaning of any of these data types. All that they can contain is a sequence of bytes. Thus an application has to *serialise* any data into a stream of bytes in order to write it, and deserialise the stream of bytes back into suitable data structures on reading it. These two operations are known as *marshalling* and *unmarshalling* respectively.

IP, TCP 或者 UDP 网络包并不知道这些数据类型的含义，它们只是字节序列的载体。因此，写入网络包的时候，应用需要将要传输的(有类型的)数据 *序列化* 成字节流，反之，读取网络包的时候，应用需要将字节流*反序列化*成合适的数据结构，这两个操作被分别称为*编组*和*解组*。

For example, consider sending the following variable length table of two columns of variable length strings:

例如:考虑发送如下这样一个由两列可变长度字符串构成的可变长度的表格

fred	programmer
liping	analyst
sureerat	manager

This could be done by in various ways. For example, suppose that it is known that the data will be an unknown number of rows in a two-column table. Then a marshalled form could be

这可以通过多种方式来完成。比如：假设知道数据是一个未知行数的两列表格，那么编组形式可能是:

```
3                    // 3 rows, 2 columns assumed
4 fred               // 4 char string,col 1
10 programmer        // 10 char string,col 2
6 liping             // 6 char string, col 1
7 analyst            // 7 char string, col 2
8 sureerat           // 8 char string, col 1
7 manager            // 7 char string, col 2
```

Variable length things can alternatively have their length indicated by terminating them with an "illegal" value, such as '\0' for strings:

可变长度的事物都可以通过用一个“非法”的终结值,比如对于字符串来说的'\0',来间接获得它们的长度

```
3
fred\0
programmer\0
liping\0
analyst\0
sureerat\0
manager\0
```

Alternatively, it may be known that the data is a 3-row fixed table of two columns of strings of length 8 and 10 respectively. Then a serialisation could be

假设知道数据是一个三行两列且每列长度分别是 8 或 10 的表格，那么序列化的结果可能是:

```
fred\0\0\0\0
programmer
liping\0\0
analyst\0\0\0
sureerat
manager\0\0\0
```

Any of these formats is okay - but the message exchange protocol must specify which one is used, or allow it to be determined at runtime.

这些格式中的任意一种都是可行的 - 但是消息交换协议必须指定使用哪一种(格式)，或者约定在运行期再做决定。

Mutual agreement

交互协议

The previous section gave an overview of the issue of data serialisation. In practise, the details can be considerably more complex. For example, consider the first possibility, marshalling a table into the stream

前一小节总结了在数据序列化过程中可能遇到的各种问题。而在实际操作中，需要考虑的细节还更多一些，例如：先考虑下面这个问题，如何将下面这个表编组成流.

```
3
4 fred
10 programmer
6 liping
7 analyst
8 sureerat
7 manager
```

Many questions arise. For example, how many rows are possible for the table - that is, how big an integer do we need to describe the row size? If it is 255 or less, then a single byte will do, but if it is more, then a short, integer or long may be needed. A similar problem occurs for the length of each string. With the characters themselves, to which character set do they belong? 7 bit ASCII? 16 bit Unicode? The question of character sets is discussed at length in a later chapter.

许多问题冒出来了。例如：这个表格可能有多少行？ - 即我们需要多大的整数来表示表格的大小，如果它只有 255 行或者更少，那么一个字节就够了，如果更大一些，就可能需要 short，integer 或者 long 来表示了。对于字符串的长度也存在同样的问题，对字符本身来说，它们属于哪种字符集? 7 位的 ASCII？ 16 位的 Unicode？ 字符集的问题将会在后面的章节里详细讨论。

The above serialisation is *opaque* or *implicit*. If data is marshalled using the above format, then there is nothing in the serialised data to say how it should be unmarshalled. The unmarshalling side has to know exactly how the data is serialised in order to unmarshal it correctly. For example, if the number of rows is marshalled as an eight-bit integer, but unmarshalled as a sixteen-bit integer, then an incorrect result will occur as the receiver tries to unmarshall 3 and 4 as a sixteen-bit integer, and the receiving program will almost certainly fail later.

上面的序列化是*不透明的*或者被称为*隐式的*，如果采用这种格式来编组数据，那么序列化后的数据中没有包含任何指示它应该被如何解组的信息。为了正确的解组，解组的一端需要精确的知晓编组的方式。如果数据的行数以 8 位整型数的方式编组，却以 16 位整型的方式解组，那么接收者将得到错误的解码结果。比如接受者尝试将 3 与 4 当作 16 位整型解组，在后续的程序运行的时候肯定会失败。

An early well-known serialisation method is XDR (external data representation) used by Sun's RPC, later known as ONC (Open Network Computing). XDR is defined by RFC 1832 and it is instructive to see how precise this specification is. Even so, XDR is inherently type-unsafe as serialised data contains no type information. The correctness of its use in ONC is ensured primarily by compilers generating code for both marshalling and unmarshalling.

早期比较出名的序列化方法是 Sun 公司的 RPC 中使用的 XDR(外部资料表示法)。后来就是 ONC(开放式网络运算)。XDR 由 RFC 1832 定义，阅读一下这个规范的详细定义是有意义的，即便如此，由于序列化的数据中不包含类型信息，XDR 是天生不安全的。ONC 中主要通过由编译器为编、解组生成额外的代码来确保数据的正确性。

Go contains no explicit support for marshalling or unmarshalling opaque serialised data. The RPC package in Go does not use XDR, but instead uses "gob" serialisation, described later in this chapter.

Go 没有为编、解组不透明的序列化数据提供显式的支持,标准包中的 RPC 包也没有使用 XDR，而是使用了这一章后面的小节中将要介绍的 gob 来作为替代方案。

Self-describing data

自描述数据

Self-describing data carries type information along with the data. For example, the previous data might get encoded as

自描述数据在最终的结果数据中附带了类型信息,例如，前面提到的数据可能被编码为:

```
table
    uint8 3
    uint 2
string
    uint8 4
    []byte fred
string
    uint8 10
    []byte programmer
string
    uint8 6
    []byte liping
string
    uint8 7
    []byte analyst
string
    uint8 8
    []byte sureerat
string
    uint8 7
    []byte manager
```

Of course, a real encoding would not normally be as cumbersome and verbose as in the example: small integers would be used as type markers and the whole data would be packed in as small a byte array as possible. (XML provides a counter-example, though.). However, the principle is that the marshaller will generate such type information in the serialised data. The unmarshaller will know the type-generation rules and will be able to use this to reconstruct the correct data structure.

当然，实际使用的编码方式不会如此啰嗦。小整数可能被用作类型标记，并且整个数据编码后的字节数组会尽量的小（XML 是一个反例）。原则就是编组器会在序列化后的数据中包含类型信息。解组器知道类型生成的规则，并使用此规则重组出正确的数据结构。

ASN.1

抽象语法表示法

Abstract Syntax Notation One (ASN.1) was originally designed in 1984 for the telecommunications industry. ASN.1 is a complex standard, and a subset of it is supported by Go in the package "asn1". It builds self-describing serialised data from complex data structures. Its primary use in current networking systems is as the encoding for X.509 certificates which are heavily used in authentication systems. The support in Go is based on what is needed to read and write X.509 certificates.

抽象语法表示法/1(ASN.1)最初出现在 1984 年，它是一个为电信行业设计的复杂标准，Go 的标准包 asn1 实现了它的一个子集，它可以将复杂的数据结构序列化成自描述的数据。在当前的网络系统中，它主要用于对认证系统中普遍使用的 X.509 证书的编码。Go 对 ASN.1 的支持主要是 X.509 证书的读写上。

Two functions allow us to marshal and unmarshal data

以下两个函数用以对数据的编、解组

```
func Marshal(val interface{}) ([]byte, os.Error)
func Unmarshal(val interface{}, b []byte) (rest []byte, err os.Error)
```

The first marshals a data value into a serialised byte array, and the second unmarshals it. However, the first argument of type **interface** deserves further examination. Given a variable of a type, we can marshal it by just passing its value. To unmarshal it, we need a variable of a named type that will match the serialised data. The precise details of this are discussed later. But we also need to make sure that the variable is allocated to memory for that type, so that there is actually existing memory for the unmarshalling to write values into.

前一个将数据值编组成序列化的字节数组，后一个将其解组出来，需要对 **interface** 类型的参数进行更多的类型检查。编组时，我们只需要传递某个类型的变量的值即可，解组它，则需要一个与被序列化过的数据匹配的确定类型的变量，我们将在后面讨论这部分的细节。除了有确定类型的变量外，我们同时需要保证那个变量的内存已经被分配，以使被解组后的数据能有实际被写入的地址。

We illustrate with an almost trivial example, of marshalling and unmarshalling an integer. We can pass an integer value to **Marshal** to return a byte array, and unmarshal the array into an integer variable as in this program:

我们将举一个整数编、解组的小例子。在这个例子中。我们先将一个整数传递给 *Marshal* 得到一个字节数组，然后又将此数组解组成一个整数。

```
/* ASN.1
 */

package main

import (
        "encoding/asn1"
        "fmt"
        "os"
)
```

```
func main() {
	mdata, err := asn1.Marshal(13)
	checkError(err)

	var n int
	_, err1 := asn1.Unmarshal(mdata, &n)
	checkError(err1)

	fmt.Println("After marshal/unmarshal: ", n)
}

func checkError(err error) {
	if err != nil {
		fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
		os.Exit(1)
	}
}
```

The unmarshalled value, is of course, 13.

当然，被解组后的值，是 13

Once we move beyond this, things get harder. In order to manage more complex data types, we have to look more closely at the data structures supported by ASN.1, and how ASN.1 support is done in Go.

一旦我们越过了这个小关卡，事情开始变得复杂。为了管理更复杂的数据类型，我们需要更深入的了解 ASN.1 支持的数据类型，以及 Go 是如何支持 ASN.1 的。

Any serialisation method will be able to handle certain data types and not handle some others. So in order to determine the suitability of any serialisation such as ASN.1, you have to look at the possible

data types supported versus those you wish to use in your application. The following ASN.1 types are taken from http://www.obj-sys.com/asn1tutorial/node4.html

任何序列化方法都只能处理某些数据类型，而对其他的数据类型无能为力。因此为了评估类似 ASN.1 等序列化方案的可行性，你必须先将要在程序中使用的数据类型与它们支持的数据类型做个比较，下面是 ASN.1 支持的数据类型，它们来自于 http://www.obj-sys.com/asn1tutorial/node4.html

The simple types are

简单数据类型有:

- BOOLEAN: two-state variable values
- INTEGER: Model integer variable values
- BIT STRING: Model binary data of arbitrary length
- OCTET STRING: Model binary data whose length is a multiple of eight
- NULL: Indicate effective absence of a sequence element
- OBJECT IDENTIFIER: Name information objects
- REAL: Model real variable values
- ENUMERATED: Model values of variables with at least three states
- CHARACTER STRING: Models values that are strings of characters fro

- BOOLEAN：两态变量值
- INTEGER：表征整型变量值
- BIT STRING：表征任意长度的二进制数据
- OCT STRING：表征长度是 8 的倍数的二进制数据
- NULL：指示一个没有有效数据的序列
- OBJECT IDENTIFIER：命名信息对象
- REAL：表征一个 real 变量值
- ENUMERATED：表征一个至少有三个状态的变量值
- CHARACTER STRING：表征一个字符串值

Character strings can be from certain character sets

字符串可以来自于确定的字符集

- NumericString: 0,1,2,3,4,5,6,7,8,9, and space
- PrintableString: Upper and lower case letters, digits, space, apostrophe, left/right parenthesis, plus sign, comma, hyphen, full stop, solidus, colon, equal sign, question mark
- TeletexString (T61String): The Teletex character set in CCITT's T61, space, and delete
- VideotexString: The Videotex character set in CCITT's T.100 and T.101, space, and delete
- VisibleString (ISO646String): Printing character sets of international ASCII, and space
- IA5String: International Alphabet 5 (International ASCII)
- GraphicString 25 All registered G sets, and space GraphicString

- NumericString: 0,1,2,3,4,5,6,7,8,9, 与空格(space)
- PrintableString: 大、小写字母，数字，空格，省略号，左、右小括号，加号，逗号，连字符，句号，斜线，冒号，等号，问号
- TeletexString(T61String): CCITT 的 Teletex 字符集中的 T61，空格和删除(delete)
- VideotexString:CCITT 的 Videotex 字符集中的 T.100 与 T.101，空格和删除(delete)
- VisibleString (ISO646String):国际 ASCII 中的打印字符集和空格
- IA5String:国际字母表 5(国际 ASCII)
- GraphicString:所有被注册的 G 集和空格

And finally, there are the structured types:

最后,以下是结构化的类型:

- SEQUENCE: Models an ordered collection of variables of different type
- SEQUENCE OF: Models an ordered collection of variables of the same type
- SET: Model an unordered collection of variables of different types
- SET OF: Model an unordered collection of variables of the same type
- CHOICE: Specify a collection of distinct types from which to choose one type
- SELECTION: Select a component type from a specified CHOICE type

- ANY: Enable an application to specify the type Note: ANY is a deprecated ASN.1 Structured Type. It has been replaced with X.680 Open Type.

- SEQUENCE:表征不同类型变量构成的有序集合
- SEQUENCE OF: 表征相同类型的变量构成的有序集合
- SET: 表征不同类型的变量构成的无序集合
- SET OF:表征相同类型的变量构成的有序集合
- CHOICE:从一个不同类型构成的特定集合中选出一个类型
- SELECTION: 从一个特定的 CHOICE 类型中选取一个组件类型
- ANY:启用一个用以指定类型的应用. 注意:ANY 是一个弃用的 ASN.1 结构类型,它被 x.680 的 Open Type 所替代

Not all of these are supported by Go. Not all possible values are supported by Go. The rules as given in the Go "asn1" package documentation are

不是以上所有的类型、可能的值都被 Go 支持，在 Go 'asn1'包文档中定义的规则如下:

- An ASN.1 INTEGER can be written to an int or int64. If the encoded value does not fit in the Go type, Unmarshal returns a parse error.
- An ASN.1 BIT STRING can be written to a BitString.
- An ASN.1 OCTET STRING can be written to a []byte.
- An ASN.1 OBJECT IDENTIFIER can be written to an ObjectIdentifier.
- An ASN.1 ENUMERATED can be written to an Enumerated.
- An ASN.1 UTCTIME or GENERALIZEDTIME can be written to a *time.Time.
- An ASN.1 PrintableString or IA5String can be written to a string.
- Any of the above ASN.1 values can be written to an interface{}. The value stored in the interface has the corresponding Go type. For integers, that type is int64.
- An ASN.1 SEQUENCE OF x or SET OF x can be written to a slice if an x can be written to the slice's element type.
- An ASN.1 SEQUENCE or SET can be written to a struct if each of the elements in the sequence can be written to the corresponding element in the struct.

- ASN.1 INTEGER 可以被写入 int 或者 int64 中. 如果被编码的值与 Go 类型不匹配,Unmarshal 将返回一个解析错误.
- ASN.1 BIT STRING 可以被写入 BitString 中.
- ASN.1 OCT STRING 可以被写入[]byte 中.
- ASN.1 OBJECT IDENTIFIER 可以被写入 ObjectIdentifier 中.
- ASN.1 ENUMERATED 可以被写入 Enumerated 中.
- ASN.1 UTCTIME 或者 GENERALIZEDTIME 可以被写入*time.Time 中.
- ASN.1 PrintableString 或者 IA5String 可以被写入 string 中.
- 以上的任何 ASN.1 类型的值都可以作为对应的 Go 类型的值写入 interface{}中。比如整数放入 interface{}的话，它对应的类型是 int64。
- 如果一个变量 x 可以被当做某个类型写入，那么 ASN.1 中的 x 构成的有序列或者集合就可以当做这个类型的 slice 写入了。
- 如果某个有序列或者集合中的所有元素都可以被写入到某个结构里与之对应的元素中，那么此 ASN.1 SEQUENCE 或者 SET 就可以写入到这个结构中。

Go places real restrictions on ASN.1. For example, ASN.1 allows integers of any size, while the Go implementation will only allow upto signed 64-bit integers. On the other hand, Go distinguishes between signed and unsigned types, while ASN.1 doesn't. So for example, transmitting a value of **uint64** may fail if it is too large for **int64**,

Go 在实现上,为 ASN.1 添加了一些约束。例如 ASN.1 允许任意大小的整数,而 GO 只允许最大为 64 位有符号整数能表示的值.另一方面，Go 区分有符号类型与无符号类型,而在 ASN.1 则没有分别.因此传递一个大于 **int64** 最大值能表示的 **uint64** 的值，则可能会失败。

In a similar vein, ASN.1 allows several different character sets. Go only supports PrintableString and IA5String (ASCII). ASN.1 does not support Unicode characters (which require the BMPString ASN.1 extension). The basic Unicode character set of Go is not supported, and if an application requires transport of Unicode characters, then an encoding such as UTF-7 will be needed. Such encodings are discussed in a later chapter on character sets.

同理，ASN.1 允许多个不同的字符集,而 Go 只支持 PrintableString 和 IA5String(ASCII). ASN.1 不支持 Unicode 字符(它需要 BMPString ASN.1 扩展)，连 Go 中的基本 Unicode 字符集它都不支持，如果应用程序需要传输 Unicode 字符，则可能需要类似 UTF-7 的编码。有关编码的内容将会在后边字符集相关的章节来讨论。

We have seen that a value such as an integer can be easily marshalled and unmarshalled. Other basic types such as booleans and reals can be similarly dealt with. Strings which are composed entirely of ASCII characters can be marshalled and unmarshalled. However, if the string is, for example, "hello \u00bc" which contains the non-ASCII character '¼' then an error will occur: "ASN.1 structure error: PrintableString contains invalid character". This code works, as long as the string is only composed of printable characters:

我们已经看到，整型的值很容易被编、解组。类似的 boolean 与 real 等基本类型处理手法也类似。由 ASCII 字符构成的字符串也很容易。但当处理 "hello \u00bc"这种含有 '¼'这个非 ASCII 字符的字符串，则会出现错误：“ASN.1 结构错误:PrintableString 包含非法字符”。以下的代码仅在处理由可打印字符（printable characters）构成的字符串时,工作良好。

```
s := "hello"
mdata, _ := asn1.Marshal(s)

var newstr string
asn1.Unmarshal(mdata, &newstr)
```

ASN.1 also includes some "useful types" not in the above list, such as UTC time. Go supports this UTC time type. This means that you can pass time values in a way that is not possible for other data values. ASN.1 does not support pointers, but Go has special code to manage pointers to time values. The function **GetLocalTime** returns ***time.Time**. The special code marshals this, and it can be unmarshalled into a pointer variable to a **time.Time** object. Thus this code works

ASN.1 还包含一些未在上边列表中出现的“有用的类型(useful types)”，比如 UTC 时间类型，GO 支持此 UTC 时间类型。就是说你可以用一种特有的类型来传递时间值。ASN.1 不支持指针,Go 中却有指向时间值的指针。比如函数 GetLocalTime 返回*time.Time。asn1 包编组这个 time 结构，也使用这个包解组到一个 time.Time 对象指针中。代码如下

```
t := time.LocalTime()
mdata, err := asn1.Marshal(t)

var newtime = new(time.Time)
_, err1 := asn1.Unmarshal(&newtime, mdata)
```

Both LocalTime and new handle pointers to a *time.Time, and Go looks after this special case.

LocalTime 与 new 函数都返回的是*time.Time 类型的指针，GO 将内部对这些特殊类型进行处理。

In general, you will probably want to marshal and unmarshal structures. Apart from the special case of time, Go will happily deal with structures, but not with pointers to structures. Operations such as new create pointers, so you have to dereference them before marshalling/unmarshalling them. Go normally dereferences pointers for you when needed, but not in this case. These both work for a type T:

除了 time 这种特殊情况外，你可能要编、解组结构类型。除了上面提到的 Time 结构外，其他的结构 Go 还是很好处理的。类以 new 的操作将会创建指针，因此在编、解组之前，你需要解引用它。通常，Go 会随需自动对指针进行解引用，但是下面这个例子并不是这么个情况。对于类型 T，以下两种方式均可.

```
// using variables
var t1 T
```

```
t1 = ...
mdata1, _ := asn1.Marshal(t)

var newT1 T
asn1.Unmarshal(&newT1, mdata1)

/// using pointers
var t2 = new(T)
*t2 = ...
mdata2, _ := asn1.Marshal(*t2)

var newT2 = new(T)
asn1.Unmarshal(newT2, mdata2)
```

Any suitable mix of pointers and variables will work as well.

恰当地的使用指针与变量能让代码工作得更好。

The fields of a structure must all be exportable, that is, field names must begin with an uppercase letter. Go uses the **reflect** package to marshal/unmarshal structures, so it must be able to examine all fields. This type cannot be marshalled:

结构的所有字段必须是公共的，即字段名必须以大写字母开头。Go 内部实际是使用 **reflect** 包来编、解组结构，因此 reflect 包必须能访问所有的字段。比如下面这个类型是不能被编组的:

```
type T struct {
    Field1 int
    field2 int // not exportable
}
```

ASN.1 only deals with the data types. It does not consider the names of structure fields. So the following type **T1** can be marshalled/unmarshalled into type **T2** as the corresponding fields are the same types:

ASN.1 只处理数据类型，它并不关心结构字段的名字。因此只要对应的字段类型相同那么下面的 T1 类型将可以被解、解组到 T2 类型中。

```
type T1 struct {
	F1 int
	F2 string
}

type T2 struct {
	FF1 int
	FF2 string
}
```

Not only the types of each field must match, but the number must match as well. These two types don't work:

不仅每个字段的类型必须匹配，而且字段数目也要相等，下面两个类型将不能互编、解码：

```
type T1 struct {
	F1 int
}

type T2 struct {
```

```
    F1 int
    F2 string // too many fields
}
```

ASN.1 daytime client and server

ASN.1 日期查询服务客户端与服务器

Now (finally) let us turn to using ASN.1 to transport data across the network.

现在（最后）让我们使用 ASN.1 来跨网络传输数据

We can write a TCP server that delivers the current time as an ASN.1 Time type, using the techniques of the last chapter. A server is

我们可以使用上一章的技术来编写一个将当前时间作为 ASN.Time 类型时间来传送的 TCP 服务器。服务器是:

```
/* ASN1 DaytimeServer
 */
package main

import (
        "encoding/asn1"
        "fmt"
        "net"
        "os"
        "time"
)
```

```
func main() {


        service := ":1200"
        tcpAddr, err := net.ResolveTCPAddr("tcp", service)
        checkError(err)


        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)


        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }


                daytime := time.Now()
                // Ignore return network errors.
          mdata, _ := asn1.Marshal(daytime)
                conn.Write(mdata)
                conn.Close() // we're finished
 }
}


func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}
```

which can be compiled to an executable such as **ASN1DaytimeServer** and run with no arguments. It will wait for connections and then send the time as an ASN.1 string to the client.

它可以被编译为一个诸如名为 *ASN1DaytimeServer* 的可执行程序，运行它不需要任何实际参数，（启动后）它将等待来自客户端的连接，当有新连接后它会将当前时间当作 ASN.1 字符串传回给客户端.

A client is

客户端代码是

```
/* ASN.1 DaytimeClient
 */
package main

import (
	"bytes"
	"encoding/asn1"
	"fmt"
	"io"
	"net"
	"os"
	"time"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Fprintf(os.Stderr, "Usage: %s host:port", os.Args[0])
		os.Exit(1)
	}
```

```
        service := os.Args[1]

        conn, err := net.Dial("tcp", service)
        checkError(err)

        result, err := readFully(conn)
        checkError(err)

        var newtime time.Time
        _, err1 := asn1.Unmarshal(result, &newtime)
        checkError(err1)

        fmt.Println("After marshal/unmarshal: ", newtime.String())

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
```

```
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
        return result.Bytes(), nil
}
```

This connects to the service given in a form such as **localhost:1200**, reads the TCP packet and decodes the ASN.1 content back into a string, which it prints.

连接字符串形如：**localhost:1200**。它将读取应答 TCP 包然后将 ASN.1 内容解码成字符串并输出。

We should note that neither of these two - the client or the server - are compatable with the text-based clients and servers of the last chapter. This client and server are exchanging ASN.1 encoded data values, not textual strings.

我们应当注意，无论是客户端还是服务器都不兼容前一章介绍的基于文本的客户端与服务器。此地的客户端与服务器交换的是 ASN.1 编码的数据值，而非文本串。

JSON

JSON

JSON stands for JavaScript Object Notation. It was designed to be a lighweight means of passing data between JavaScript systems. It uses a text-based format and is sufficiently general that it has become used as a general purpose serialisation method for many programming languages.

JSON 全称是JavaScript Object Notation,它是一种应用于JavaScript 系统之间传递数据的轻量级格式。它使用基于文本的格式，因为足够通用，现在已经成为了多种编程语言采用的通用的序列化方法了。

JSON serialises objects, arrays and basic values. The basic values include string, number, boolean values and the null value. Arrays are a comma-separated list of values that can represent arrays, vectors, lists or sequences of various programming languages. They are delimited by square brackets "[...]". Objects are represented by a list of "field: value" pairs enclosed in curly braces "{ ... }".

JSON 序列化对象，数组和基本值。基本值包括：字符串，数字，布尔值和 NULL 值。数组是逗号分割的一组值的列表，可以用来表示各种编程语言中的数组、向量、列表或者序列。它们由方括号来界定，对象则由一个包含在大括号中的"field: values"对构成的列表来表示。

For example, the table of employees given earlier could be written as an array of employee objects:

例如.前面提到过的雇员表可以被编码成如下的一个雇员对象的数组.

```
[
    {Name: fred, Occupation: programmer},
    {Name: liping, Occupation: analyst},
    {Name: sureerat, Occupation: manager}
]
```

There is no special support for complex data types such as dates, no distinction between number types, no recursive types, etc. JSON is a very simple language, but nevertheless can be quite useful. Its text-based format makes it easy for people to use, even though it has the overheads of string handling.

JSON 没有为类似日期这这样的复杂数据类型提供特别的格式支持，不区分各种数字类型，也没有递归类型等。JSON 是一个非常简单但却十分有用的语言，尽管他基于文本的格式在字符传递上开销过多，但是却很适合人类阅读和使用。

From the Go JSON package specification, marshalling uses the following type-dependent default encodings:

从 Go JSON 包的规范文档可知，JSON 包将在编组时使用以下类型相关的默认编码方法：

- Boolean values encode as JSON booleans.
- Floating point and integer values encode as JSON numbers.
- String values encode as JSON strings, with each invalid UTF-8 sequence replaced by the encoding of the Unicode replacement character U+FFFD.
- Array and slice values encode as JSON arrays, except that []byte encodes as a base64-encoded string.
- Struct values encode as JSON objects. Each struct field becomes a member of the object. By default the object's key name is the struct field name converted to lower case. If the struct field has a tag, that tag will be used as the name instead.
- Map values encode as JSON objects. The map's key type must be string; the object keys are used directly as map keys.
- Pointer values encode as the value pointed to. (Note: this allows trees, but not graphs!). A nil pointer encodes as the null JSON object.
- Interface values encode as the value contained in the interface. A nil interface value encodes as the null JSON object.
- Channel, complex, and function values cannot be encoded in JSON. Attempting to encode such a value causes Marshal to return an InvalidTypeError.
- JSON cannot represent cyclic data structures and Marshal does not handle them. Passing cyclic structures to Marshal will result in an infinite recursion.

- 布尔值被编码为 JSON 的布尔值。
- 浮点数与整数被编码为 JSON 的数字值。

- 字符串被编码为 JSON 的字符串，每一个非法的 UTF-8 序列将会被 UTF8 替换符 U+FFFD 替换。
- 数组与 Slice 会被编码为 JSON 数组，但是[]byte 是会被编码为 base64 字符串。
- 结构体被编码为 JSON 对象。每一个结构体字段被编码为此对象的对应成员，默认情况下对象的 key 的名字是对应结构体字段名的小写。如果此字段含有 tag，则此 tag 将是最终对象 key 的名字。
- map 值被编码为 JSON 对象，此 map 的 key 的类型必须是 string；map 的 key 直接被当作 JSON 对象的 key。
- 指针值被编码为指针所指向的值（注意：此处只允许出现树(tree),而不允许出现图（graph）！）。空指针被编码为空 JSON 对象.
- 接口值被编码为接口实际包含的值。空接口被编码为空 JSON 对象。
- 程道，复数，函数不能被编码为 JSON 格式。如果尝试这样做，Marshal 将会返回一个 InvalidTypeError 错误。
- JSON 不能表示环形数据结构。Go 的 Marshal 函数也不处理它们，将一个环形结构传递给 Marshal 将会导致死循环。

A program to store JSON serialised data into a file is

将 JSON 数据存入文件的示例如下:

```
/* SaveJSON */

package main

import (
        "encoding/json"
        "fmt"
        "os"
)
```

```
type Person struct {
	Name   Name
	Email []Email
}

type Name struct {
	Family     string
	Personal string
}

type Email struct {
	Kind       string
	Address string
}

func main() {
	person := Person{
		Name: Name{Family: "Newmarch", Personal: "Jan"},
		Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
			Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

	saveJSON("person.json", person)
}

func saveJSON(fileName string, key interface{}) {
	outFile, err := os.Create(fileName)
	checkError(err)
	encoder := json.NewEncoder(outFile)
	err = encoder.Encode(key)
```

```
        checkError(err)
        outFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

and to load it back into memory is

可以这样将之重新加载到内存中:

```
/* LoadJSON
 */

package main

import (
        "encoding/json"
        "fmt"
        "os"
)

type Person struct {
        Name  Name
        Email []Email
}
```

```
type Name struct {
	Family     string
	Personal string
}

type Email struct {
	Kind       string
	Address string
}

func (p Person) String() string {
	s := p.Name.Personal + " " + p.Name.Family
	for _, v := range p.Email {
		s += "\n" + v.Kind + ": " + v.Address
	}
	return s
}
func main() {
	var person Person
	loadJSON("person.json", &person)

	fmt.Println("Person", person.String())
}

func loadJSON(fileName string, key interface{}) {
	inFile, err := os.Open(fileName)
	checkError(err)
	decoder := json.NewDecoder(inFile)
```

```
        err = decoder.Decode(key)
        checkError(err)
        inFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The serialised form is (formatted nicely)

被序列化后的结果如：（经过了美化处理）

```
{"Name":{"Family":"Newmarch",
         "Personal":"Jan"},
 "Email":[{"Kind":"home","Address":"jan@newmarch.name"},
          {"Kind":"work","Address":"j.newmarch@boxhill.edu.au"}
         ]
}
```

A client and server

客户端与服务器

A client to send a person's data and read it back ten times is

一个将 person 数据收发 10 次的客户端

```
/* JSON EchoClient
 */
package main

import (
	"fmt"
	"net"
	"os"
	"encoding/json"
	"bytes"
	"io"
)

type Person struct {
	Name  Name
	Email []Email
}

type Name struct {
	Family   string
	Personal string
}

type Email struct {
	Kind    string
	Address string
}
```

```
func (p Person) String() string {
	s := p.Name.Personal + " " + p.Name.Family
	for _, v := range p.Email {
		s += "\n" + v.Kind + ": " + v.Address
	}
	return s
}

func main() {
	person := Person{
		Name: Name{Family: "Newmarch", Personal: "Jan"},
		Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
			Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "host:port")
		os.Exit(1)
	}
	service := os.Args[1]

	conn, err := net.Dial("tcp", service)
	checkError(err)

	encoder := json.NewEncoder(conn)
	decoder := json.NewDecoder(conn)

	for n := 0; n < 10; n++ {
		encoder.Encode(person)
		var newPerson Person
```

```
                decoder.Decode(&newPerson)
                fmt.Println(newPerson.String())
        }

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
                        }
                        return nil, err
                }
        }
```

```
        return result.Bytes(), nil
}
```

and the corrsponding server is

对应的服务器

```
/* JSON EchoServer
 */
package main

import (
        "fmt"
        "net"
        "os"
        "encoding/json"
)

type Person struct {
        Name   Name
        Email []Email
}

type Name struct {
        Family     string
        Personal string
}

type Email struct {
        Kind       string
```

```
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}

func main() {

        service := "0.0.0.0:1200"
        tcpAddr, err := net.ResolveTCPAddr("tcp", service)
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }

                encoder := json.NewEncoder(conn)
                decoder := json.NewDecoder(conn)
```

```
                for n := 0; n < 10; n++ {
                        var person Person
                        decoder.Decode(&person)
                        fmt.Println(person.String())
                        encoder.Encode(person)
                }
                conn.Close() // we're finished
 }
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The gob package

gob 包

Gob is a serialisation technique specific to Go. It is designed to encode Go data types specifically and does not at present have support for or by any other languages. It supports all Go data types except for channels, functions and interfaces. It supports integers of all types and sizes, strings and booleans, structs, arrays and slices. At present it has some problems with circular structures such as rings, but that will improve over time.

gob 是 Go 中特有的序列化技术。它只能编码 Go 的数据类型，目前它不支持其他语言，反之亦然。它支持除 interface，function，channel 外的所有的 Go 数据类型。它支持任何类型和任何大小的整数，还有字符串和布尔值，结构，数组与切片。目前它在处理 ring 等环型数据结构方面还存在一些问题，但假以时日，将会得到改善。

Gob encodes type information into its serialised forms. This is far more extensive than the type information in say an X.509 serialisation, but far more efficient than the type information contained in an XML document. Type information is only included once for each piece of data, but includes, for example, the names of struct fields.

Go 将类型信息编码到序列化后的表单中，在扩展性方面这远比对应的 X.509 序列化方法要好。而同时与将类型信息包含在表单中的 XML 文档相比，则更加高效。对于每个数据，类型信息只包含一次。当然，包含的是字段名称这样的信息。

This inclusion of type information makes Gob marshalling and unmarshalling fairly robust to changes or differences between the marshaller and unmarshaller. For example, a struct

包含类型信息使得 Gob 在编、解组操作上，当 marshaler 与 unmarshaler 不同或者有变化时，具有相当高的健壮性。例如，如下这个结构:

```
struct T {
        a int
        b int
}
```

can be marshalled and then unmarshalled into a different struct

可以被编组并随需解组到不同的结构中.

```
struct T {
        b int
        a int
}
```

where the order of fields has changed. It can also cope with missing fields (the values are ignored) or extra fields (the fields are left unchanged). It can cope with pointer types, so that the above struct could be unmarshalled into

此处变更了字段的顺序.它也可以处理缺少字段（值将被忽略）或多出字段（此字段原样保持）的情况。它也可以处理指针类型，因此上边的结构可以被解组到下面的结构中.

```
 struct T {
        *a int
        **b int
}
```

To some extent it can cope with type coercions so that an **int** field can be broadened into an **int64**, but not with incompatable types such as **int** and **uint**.

在一定程度上，它也可以强制执行类型转换，比如 int 字段被扩展成为 int64。而对于不兼容类型，比如 int 与 uint,就无能为力了.

To use Gob to marshall a data value, you first need to create an Encoder. This takes a Writer as parameter and marshalling will be done to this write stream. The encoder has a method Encode which marshalls the value to the stream. This method can be called multiple times on multiple pieces of data. Type information for each data type is only written once, though.

为了使用 gob 编组一个数据值，首先你得创建 Encoder。它使用 Writer 作为参数，编组操作会将最终结果写入此流中。encoder 有个 Encode 方法，它执行将值编组成流的操作。此方法可以在多份数据上被调用多次。但是对于每一种数据类型，类型信息却只会被写入一次。

You use a Decoder to unmarshall the serialised data stream. This takes a Reader and each read returns an unmarshalled data value.

你将使用 Decoder 来执行解组序列化后的数据流的操作。它持有一个 Reader 参数，每次读取都将返回一个解组后的数据值。

A program to store gob serialised data into a file is

将 gob 序列化后的数据存入文件的示例程序如下：

```
/* SaveGob
 */

package main

import (
        "fmt"
        "os"
        "encoding/gob"
)

type Person struct {
        Name   Name
        Email []Email
}

type Name struct {
        Family     string
        Personal string
}

type Email struct {
        Kind       string
```

```
        Address string
}

func main() {
        person := Person{
                Name: Name{Family: "Newmarch", Personal: "Jan"},
                Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
                        Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

        saveGob("person.gob", person)
}

func saveGob(fileName string, key interface{}) {
        outFile, err := os.Create(fileName)
        checkError(err)
        encoder := gob.NewEncoder(outFile)
        err = encoder.Encode(key)
        checkError(err)
        outFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

and to load it back into memory is

将之重新加载回内存的操作如下：

```
/* LoadGob
 */

package main

import (
	"fmt"
	"os"
	"encoding/gob"
)

type Person struct {
	Name  Name
	Email []Email
}

type Name struct {
	Family   string
	Personal string
}

type Email struct {
	Kind    string
	Address string
}

func (p Person) String() string {
```

```
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}
func main() {
        var person Person
        loadGob("person.gob", &person)

        fmt.Println("Person", person.String())
}

func loadGob(fileName string, key interface{}) {
        inFile, err := os.Open(fileName)
        checkError(err)
        decoder := gob.NewDecoder(inFile)
        err = decoder.Decode(key)
        checkError(err)
        inFile.Close()
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

A client and server

一个客户端与服务器的例子

A client to send a person's data and read it back ten times is

一个将 person 数据收发 10 次的客户端

```
/* Gob EchoClient
 */
package main

import (
        "fmt"
        "net"
        "os"
        "encoding/gob"
        "bytes"
        "io"
)

type Person struct {
        Name    Name
        Email []Email
}

type Name struct {
        Family     string
        Personal string
}

type Email struct {
```

```
        Kind      string
        Address string
}

func (p Person) String() string {
        s := p.Name.Personal + " " + p.Name.Family
        for _, v := range p.Email {
                s += "\n" + v.Kind + ": " + v.Address
        }
        return s
}

func main() {
        person := Person{
                Name: Name{Family: "Newmarch", Personal: "Jan"},
                Email: []Email{Email{Kind: "home", Address: "jan@newmarch.name"},
                        Email{Kind: "work", Address: "j.newmarch@boxhill.edu.au"}}}

        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := net.Dial("tcp", service)
        checkError(err)

        encoder := gob.NewEncoder(conn)
        decoder := gob.NewDecoder(conn)
```

```
        for n := 0; n < 10; n++ {
                encoder.Encode(person)
                var newPerson Person
                decoder.Decode(&newPerson)
                fmt.Println(newPerson.String())
        }

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}

func readFully(conn net.Conn) ([]byte, error) {
        defer conn.Close()

        result := bytes.NewBuffer(nil)
        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                result.Write(buf[0:n])
                if err != nil {
                        if err == io.EOF {
                                break
```

```
                                }
                                return nil, err
                        }
                }
                return result.Bytes(), nil
}
```

and the corrsponding server is

对应的服务器：

```
/* Gob EchoServer
 */
package main

import (
        "fmt"
        "net"
        "os"
        "encoding/gob"
)

type Person struct {
        Name    Name
        Email []Email
}

type Name struct {
        Family      string
        Personal string
```

```
}

type Email struct {
	Kind	string
	Address string
}

func (p Person) String() string {
	s := p.Name.Personal + " " + p.Name.Family
	for _, v := range p.Email {
		s += "\n" + v.Kind + ": " + v.Address
	}
	return s
}

func main() {

	service := "0.0.0.0:1200"
	tcpAddr, err := net.ResolveTCPAddr("tcp", service)
	checkError(err)

	listener, err := net.ListenTCP("tcp", tcpAddr)
	checkError(err)

	for {
		conn, err := listener.Accept()
		if err != nil {
			continue
		}
```

```
                encoder := gob.NewEncoder(conn)
                decoder := gob.NewDecoder(conn)

                for n := 0; n < 10; n++ {
                        var person Person
                        decoder.Decode(&person)
                        fmt.Println(person.String())
                        encoder.Encode(person)
                }
                conn.Close() // we're finished
 }
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Encoding binary data as strings

将二进制数据编码为字符串

Once upon a time, transmtting 8-bit data was problematic. It was often transmitted over noisy serial lines and could easily become corrupted. 7-bit data on the other hand could be transmitted more reliably because the 8th bit could be used as check digit. For example, in an "even parity" scheme, the check digit would be set to one or zero to make an even number of 1's in a byte. This allows detection of errors of a single bit in each byte.

以前，传输 8-bit 数据总是会出现各种问题。它通常由充满噪声的串行线来输入，因此会出错。因为第 8 个比特位可以被用来做数字检验，所以 7-bit 的传输要值得信任一些。例如在“偶数奇偶校检”模式下，为了让一个字节中出现偶数个 1,校检位可以被设置 1 或 0，这将可侦测每个字节中的单个 bit 位出现的错误.

ASCII is a 7-bit character set. A number of schemes have been developed that are more sophisticated than simple parity checking, but which involve translating 8-bit binary data into 7-bit ASCII format. Essentially, the 8-bit data is stretched out in some way over the 7-bit bytes.

ASCII 是一种 7-bit 字符集。很多比‘奇偶检验’精巧的模式被开发出来，但是本质上都是将 8-bit 二进制数据转化成 7-bit ASCII 格式。本质上 8-bit 数据是 7-bit 数据的延伸。

Binary data transmitted in HTTP responses and requests is often translated into an ASCII form. This makes it easy to inspect the HTTP messages with a simple text reader without worrying about what strange 8-bit bytes might do to your display!

在 HTTP 的请求与应答中，二进制数据常被转化为 ASCII 的形式。这使得通过一个简单的文本阅读器来检视 HTTP 消息变得容易，而不需要担心 8-bit 字节造成的显示乱码的问题!

One common format is Base64. Go has support for many binary-to-text formats, including base64.

一个通用的格式是 Base64，Go 支持包括 base64 在内的多种 binary-to-text 格式.

There are two principal functions to use for Base64 encoding and decoding:

两个编、解码 Base64 的主要函数:

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

A simple program just to encode and decode a set of eight binary digits is

一个用以演示编解码 8 位二进制数的简单程序如下:

```
/**
 * Base64
 */

package main

import (
        "bytes"
        "encoding/base64"
        "fmt"
)

func main() {

        eightBitData := []byte{1, 2, 3, 4, 5, 6, 7, 8}
        bb := &bytes.Buffer{}
        encoder := base64.NewEncoder(base64.StdEncoding, bb)
        encoder.Write(eightBitData)
        encoder.Close()
        fmt.Println(bb)

        dbuf := make([]byte, 12)
        decoder := base64.NewDecoder(base64.StdEncoding, bb)
        decoder.Read(dbuf)
        for _, ch := range dbuf {
                fmt.Print(ch)
        }
```

```
}
```

Application-Level Protocols

应用层协议

A client and a server exchange messages consisting of message types and message data. This requires design of a suitable message exchange protocol. This chapter looks at some of the issues involved in this, and gives a complete example of a simple client-server application.

客户端和服务器的交互包括消息类型和消息数据，这就需要有适当的交互协议。本章着重讨论客户端和服务器交互相关的问题，并给出一个完整又简单的客户端服务器交互的例子。

Introduction

介绍

A client and server need to exchange information via messages. TCP and UDP provide the transport mechanisms to do this. The two processes also need to have a protocol in place so that message exchange can take place meaningfully. A protocol defines what type of conversation can take place between two components of a distributed application, by specifying messages, data types, encoding formats and so on.

客户端和服务器需要通过消息来进行交互。TCP 和 UDP 是信息交互的两种传输机制。在这两种传输机制之上就需要有协议来约定传输内容的含义。协议清楚说明分布式应用的两个模块之间交互消息的消息体、消息的数据类型、编码格式等。

Protocol Design

协议设计

There are many possibilities and issues to be decided on when designing a protocol. Some of the issues include:

当设计协议的时候，有许多许多的情况和问题需要考虑，比如：

- Is it to be broadcast or point to point?

  Broadcast must be UDP, local multicast or the more experimental MBONE. Point to point could be either TCP or UDP.
- Is it to be stateful vs stateless?

  Is it reasonable for one side to maintain state about the other side? It is often simpler to do so, but what happens if something crashes?
- Is the transport protocol reliable or unreliable?

  Reliable is often slower, but then you don't have to worry so much about lost messages.
- Are replies needed?

  If a reply is needed, how do you handle a lost reply? Timeouts may be used.
- What data format do you want?

  Two common possibilities are MIME or byte encoding.
- Is your communication bursty or steady stream?

  Ethernet and the Internet are best at bursty traffic. Steady stream is needed for video streams and particularly for voice. If required, how do you manage Quality of Service (QoS)?
- Are there multiple streams with synchronisation required?

  Does the data need to be synchronised with anything? e.g. video and voice.
- Are you building a standalone application or a library to be used by others?

  The standards of documentation required might vary.

- 是广播还是单播?

  广播必须使用 UDP，本地组播或者是更成熟的组播骨干网（MBONE）。单播就可以使用 TCP 或者 UDP。
- 消息应该是有状态还是无状态的?

  任意一边是否有必要维持另外一边的状态消息？虽然这个似乎很好实现，但是如果发生异常的时候又应该怎么做呢?
- 协议是可靠服务还是不可靠服务?

  一般来说，提供可靠服务就会更慢些，但好处是不需要考虑丢失消息的情况。
- 是否需要响应?

  如果是需要响应的话，如何处理没有响应回复的情况？或许可以设置超时时间。

- 你想要使用什么数据格式?

  一般使用两种格式：MIME 和字节流
- 消息流是使用突发性的还是稳定性的?

  Ethernet 和 Internet 最好使用突发性消息流。稳定性的消息流通常是用在视频和声频流传输上。如果要求的话，你如何保证你的服务质量（QoS）？
- 有多流同步的需求吗?

  是否有多种数据流需要进行同步？比如视频流和声频流。
- 建立的是单独的应用还是需要给别人使用的库?

  可能需要花很大的精力编写标准文档。

Version control

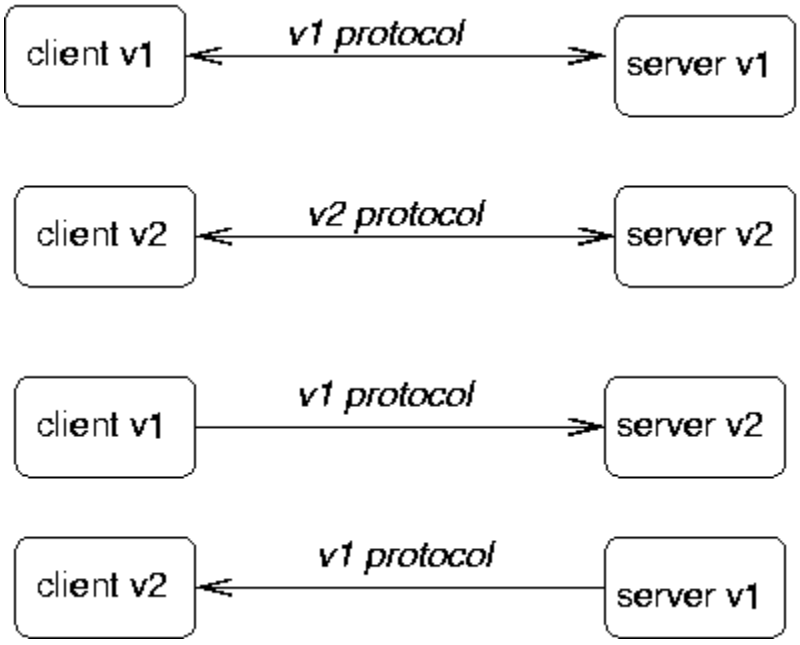
版本控制

A protocol used in a client/server system will evolve over time, changing as the system expands. This raises compatability problems: a version 2 client will make requests that a version 1 server doesn't understand, whereas a version 2 server will send replies that a version 1 client won't understand.

随着时间变化和系统的升级，客户端/服务器之间的协议也会升级。这可能会引起兼容性的问题：版本 2 的客户端发出的请求可能版本 1 的服务器无法解析，反之也一样，版本 2 的服务器回复的消息版本 1 的客户端无法解析。

Each side should ideally be able to understand messages for its own version and all earlier ones. It should be able to write replies to old style queries in old style response format.

理想情况下，不论是哪一端，都应该既能满足自己当前版本的消息规范，也能满足早期版本的消息规范。任意一端对于旧版本的请求应该返回旧版本的响应。



The ability to talk earlier version formats may be lost if the protocol changes too much. In this case, you need to be able to ensure that no copies of the earlier version still exist - and that is generally imposible.

但是如果协议变化太大的话，可能就很难保持与早期版本的兼容了。在这种情况下，你就需要保证已经不存在早期的版本了 -- 当然这个几乎是不可能的。

Part of the protocol setup should involve version information.

所以，协议应该包含有版本消息。

The Web

Web 协议

The Web is a good example of a system that is messed up by different versions. The protocol has been through three versions, and most servers/browsers now use the latest version. The version is given in each request

Web 协议就是一个由于有不同协议版本同时存在而出现混乱的例子。Web 协议已经有三个版本了，通常服务器和浏览器都是使用最新的版本，版本消息包含在请求中：

request	version
GET /	pre 1.0
GET / HTTP/1.0	HTTP 1.0
GET / HTTP/1.1	HTTP 1.1

But the *content* of the messages has been through a large number of versions:

但是*消息体的内容*已经被大量版本制定修改过：

- HTML versions 1-4 (all different), with version 5 on the horizon;
- non-standard tags recognised by different browsers;
- non-HTML documents often require content handlers that may or may not be present - does your browser have a handler for Flash?
- inconsistent treatment of document content (e.g. some stylesheet content will crash some browsers)
- Different support for JavaScript (and different versions of JavaScript)
- Different runtime engines for Java
- Many pages do not conform to *any* HTML versions (e.g. with syntax errors)

- HTML 版本 1-4（每个版本都不一样），还有即将到来的 HTML5;
- 不同浏览器各自支持非标准化的标签;
- HTML 文档之外的内容也通常需要不同的内容处理器 -- 比如你的浏览器支持 Flash 播放器吗?
- 文档内容的不一致的处理方法（例如，在一些浏览器上，有的 css 会冲突）
- 浏览器对 JavaScript 的不同支持程度（当然 JavaScript 也同时存在不同的版本）
- 不同的 Java 运行引擎
- 有的页面并没有遵守*任何* HTML 版本规范（比如 HTML 格式错误的页面）

Message Format

消息格式

In the last chapter we discussed some possibilities for representing data to be sent across the wire.

Now we look one level up, to the messages which may contain such data.

上一章我们讨论了数据传输的几种可能的表现形式。现在我们进一步研究包含数据的消息。

- The client and server will exchange messages with different meanings. e.g.
    - Login request,
    - get record request,
    - login reply,
    - record data reply.
- The client will prepare a request which must be understood by the server.
- The server will prepare a reply which must be understood by the client.

- 客户端和服务器会交换不同含义的消息，比如：
    - 登陆请求
    - 获取某些记录的请求
    - 登陆请求的回复
    - 获取某些记录请求的回复
- 客户端必须发送能被服务器解析的请求。
- 服务器必须回复能被客户端解析的响应。

Commonly, the first part of the message will be a message type.

通常来说，消息的头部必须包含消息类型。

- Client to server

```
LOGIN name passwd
GET cpe4001 grade
```

- Server to client

-
- LOGIN succeeded
- GRADE cpe4001 D

- 客户端发送给服务器

-
- LOGIN name passwd
- GET cpe4001 grade

- 服务器返回给客户端

-
- LOGIN succeeded
- GRADE cpe4001 D

The message types can be strings or integers. e.g. HTTP uses integers such as 404 to mean "not found" (although these integers are written as strings). The messages from client to server and vice versa are disjoint: "LOGIN" from client to server is different to "LOGIN" from server to client.

消息类型应该设置为字符型或者整型。比如，HTTP 使用整数 404 来表示“未找到资源”（尽管这个整型是被当做字符串使用）。客户端到服务器的消息和服务器到客户端的消息是不一样的：比如从客户端到服务器的“LOGIN”消息就不同于服务器到客户端的“LOGIN”消息。

Data Format

数据格式

There are two main format choices for messages: byte encoded or character encoded.

对于消息来说，有两种主要的数据格式可供选择：字节编码和字符编码。

Byte format

字节编码

In the byte format

对于字节编码

- the first part of the message is typically a byte to distinguish between message types.
- The message handler would examine this first byte to distinguish message type and then perform a switch to select the appropriate handler for that type.
- Further bytes in the message would contain message content according to a pre-defined format (as discussed in the previous chapter).

- 消息的头部通常使用一个字节来标示消息的类型。
- 消息处理者应该根据消息头部的类型字节来选择合适的方法来处理这个类型的消息。
- 消息后面的字节应该是根据事先定义的格式（上一章节讨论的）来包含消息具体内容。

The advantages are compactness and hence speed. The disadvantages are caused by the opaqueness of the data: it may be harder to spot errors, harder to debug, require special purpose decoding functions. There are many examples of byte-encoded formats, including major protocols such as DNS and NFS , upto recent ones such as Skype. Of course, if your protocol is not publicly specified, then a byte format can also make it harder for others to reverse-engineer it!

字节编码的优势就是紧凑小巧，传输速度快。劣势就是数据的不透明性：字节编码很难定位错误，也很难调试。往往是要求写一些额外的解码函数。有许多字节编码格式的例子，大部分协议都是使用字节编码，例如 DNS 和 NFS 协议，还有最近出现的 Skype 协议。当然，如果你的协议没有公开说明结构，使用字节编码可以让其他人使用反向工程手段很难破解!

Pseudocode for a byte-format server is

字节编码的服务器的伪代码如下

```
handleClient(conn) {
        while (true) {
                byte b = conn.readByte()
                switch (b) {
                        case MSG_1: ...
                        case MSG_2: ...
                        ...
                }
        }
}
```

Go has basic support for managing byte streams. The interface **Conn** has methods

Go 提供了基本的管理字节流的方法。 接口 **Conn** 包含有方法

```
(c Conn) Read(b []byte) (n int, err os.Error)
(c Conn) Write(b []byte) (n int, err os.Error)
```

and these methods are implemented by **TCPConn** and **UDPConn**.

这两个方法的具体实现类有 **TCPConn** and **UDPConn**。

Character Format

字符编码

In this mode, everything is sent as characters if possible. For example, an integer 234 would be sent as, say, the three characters '2', '3' and '4' instead of the one byte 234. Data that is inherently binary may be base64 encoded to change it into a 7-bit format and then sent as ASCII characters, as discussed in the previous chapter.

在这个编码模式下，所有消息都尽可能以字符的形式发送。例如，整型数字 234 会被处理成三个字符‘2’，‘3’，‘4’，而不会被处理成 234 的字节码。二进制数据将会使用 base64 编码变成为 7-bit 的格式，然后当做 ASCII 码传递，就和我们上一章讨论的一样。

In character format,

对于字符编码,

- A message is a sequence of one or more lines
- The start of the first line of the message is typically a word that represents the message type.
- String handling functions may be used to decode the message type and data.
- The rest of the first line and successive lines contain the data.
- Line-oriented functions and line-oriented conventions are used to manage this.

- 一条消息会是一行或者很多行内容
- 消息的第一行通常使用一个单词来说明消息的类型。
- 使用字符处理函数来解码消息类型和消息内容。
- 第一行之后的信息和其他行包含消息数据。
- 使用行处理函数和行处理规范来处理消息。

Pseudocode is

伪代码如下

```
handleClient() {
    line = conn.readLine()
    if (line.startsWith(...) {
        ...
    } else if (line.startsWith(...) {
        ...
    }
```

```
}
```

Character formats are easier to setup and easier to debug. For example, you can use telnet to connect to a server on any port, and send client requests to that server. It isn't so easy the other way, but you can use tools like tcpdump to snoop on TCP traffic and see immediately what clients are sending to servers.

很容易进行组装，也很容易调试。例如，你可以 telnet 连接到一台服务器的端口上，然后发送客户的请求到服务器。其他的编码方式无法轻易地监听请求。但是对于字符编码，你可以使用 tcpdump 这样的工具监听 TCP 的交互，并且立刻就能看到客户端发送给服务器端的消息。

There is not the same level of support in Go for managing character streams. There are significant issues with character sets and character encodings, and we will explore these issues in a later chapter.

在 Go 中没有像字节流那样专门处理字符流的工具。如何处理字符集和字符编码是非常重要的，我们将会在下一章专门讨论这些问题。

If we just pretend everything is ASCII, like it was once upon a time, then character formats are quite straightforward to deal with. The principal complication at this level is the varying status of "newline" across different operating systems. Unix uses the single character '\n'. Windows and others (more correctly) use the pair "\r\n". On the internet, the pair "\r\n" is most common - Unix systems just need to take care that they don't assume '\n'.

如果和以前一样，处理的所有字符都是 ASCII 码，那么我们能直接又简单地处理这些字符。但是实际上，字符处理复杂的原因是不同的操作系统上有各种不统一的“换行符”。Unix 使用简单的'\n' 来表示换行，Windows 和其他的系统（这种方法更正确）使用“\r\n”来表示。在实际的网络传输中，使用一对“\r\n”是更通用的方案 -- 因为 Unix 系统只需要注意不要设定换行符只有“\n”就可以满足这个方案。

Simple Example

简单的例子

This example deals with a directory browsing protocol - basically a stripped down version of FTP, but without even the file transfer part. We only consider listing a directory name, listing the contents of a directory and changing the current directory - all on the server side, of course. This is a complete worked example of creating all components of a client-server application. It is a simple program which includes messages in both directions, as well as design of messaging protocol.

这个例子展示的是一个文件夹浏览协议 -- 基本上就是一个简单的 FTP 协议，只是连 FTP 的文件传输都没有实现。我们考虑这个例子包含的功能有：展示文件夹名称，列出文件夹内包含的文件，改变当前文件夹路径 -- 当然所有这些文件都是在服务器的。这是一个完整的包含客户端和服务器的例子。这个简单的程序既需要两个方向的消息交互，也需要消息的具体协议设计。

Look at a simple non-client-server program that allows you to list files in a directory and change and print the directory on the server. We omit copying files, as that adds to the length of the program without really introducing important concepts. For simplicity, all filenames will be assumed to be in 7-bit ASCII. If we just looked at a standalone application first, then the pseudo-code would be

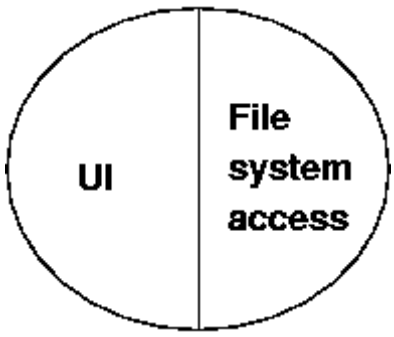
在开始例子之前，我们先看一个简单的程序，这个程序不是客户端和服务器交互的程序，它实现的功能包括：展示文件夹中的文件，打印出文件夹在服务器上的路径。在这里我们忽略正在拷贝中的文件，因为考虑这些细节会增加代码长度，却对我们要介绍的重要概念没有什么帮助。简单假设：所有的文件名都是 7 位的 ASCII 码。先考虑这个独立的程序，它的伪代码应该是：

```
read line from user
while not eof do
   if line == dir
      list directory
   else

   if line == cd <dir>
```

```
    change directory
else

if line == pwd
    print directory
else

if line == quit
    quit
else
    complain

read line from user
```

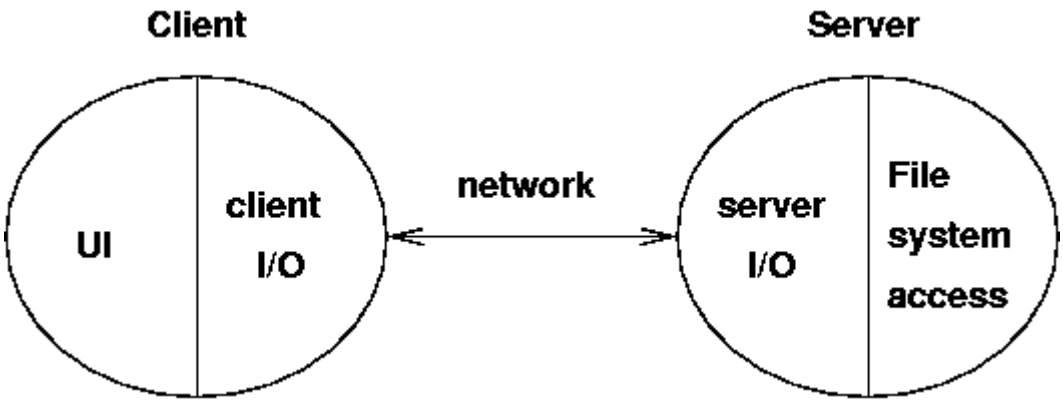
A non-distributed application would just link the UI and file access code

一个非分布式的应用是将 UI 和文件存储代码连接起来



In a client-server situation, the client would be at the user end, talking to a server somewhere else. Aspects of this program belong solely at the presentation end, such as getting the commands from the user. Some are messages from the client to the server, some are solely at the server end.

在包含有客户端和服务器的情况下，客户端就代表用户终端，用来和服务器交互。这个程序最独立的部分就是表现层，比如如何获取用户的命令等。这个程序的消息有的是从客户端到服务器，有的只是在服务器。



For a simple directory browser, assume that all directories and files are at the server end, and we are only transferring file information from the server to the client. The client side (including presentation aspects) will become

对于简单的文件夹浏览器来说，假设所有的文件夹和文件都是在服务器端，我们也只需要从服务器传递文件消息给客户端。客户端的伪代码（包括表现层）应该如下：

```
read line from user
while not eof do
   if line == dir
      list directory
   else

   if line == cd <dir>
      change directory
   else

   if line == pwd
```

```
    print directory
else

if line == quit
    quit
else
    complain

read line from user
```

where the italicised lines involve communication with the server.

上面斜体字的部分是代表需要与服务器进行交互的命令。

Alternative presentation aspects

改变表现层

A GUI program would allow directory contents to be displayed as lists, for files to be selected and actions such as change directory to be be performed on them. The client would be controlled by actions associated with various events that take place in graphical objects. The pseudo-code might look like

GUI 程序可以很方便展示文件夹内容，选择文件，做一些诸如改变文件夹路径的操作。客户端被图形化对象中的各种定义好的事件所驱动从而实现功能。伪代码如下：

```
change dir button:
    if there is a selected file
        change directory
    if successful
```

update directory label

list directory

update directory list

The functions called from the different UI's should be the same - changing the presentation should not change the networking code

不同的 UI 实现的功能都是一样的 -- 改变表现层并不需要改变网络传输的代码

Protocol - informal

协议 -- 概述

client request	server response
dir	send list of files
cd <dir>	change dir send error if failed send ok if succeed
pwd	send current directory
quit	quit

Text protocol

文本传输协议

This is a simple protocol. The most complicated data structure that we need to send is an array of strings for a directory listing. In this case we don't need the heavy duty serialisation techniques of the last chapter. In this case we can use a simple text format.

这是一个简单的协议，最复杂的部分就是我们需要使用字符串数组来列出文件夹中内容。所以，我们就不使用最后一章讲到的繁琐复杂的序列化技术了，仅仅使用一种简单的文本格式就好了。

But even if we make the protocol simple, we still have to specify it in detail. We choose the following message format:

但是实际上，即使我们想尽量使得协议简单，在细节上也需要考虑清楚。我们使用下面的消息格式约定：

- All messages are in 7-bit US-ASCII
- The messages are case-sensitive
- Each message consists of a sequence of lines
- The first word on the first line of each message describes the message type. All other words are message data
- All words are separated by exactly one space character
- Each line is terminated by CR-LF

- 所有的消息都是 7 位的 US-ASCII 码
- 所有的消息都是大小写敏感
- 每条消息都是由一系列的行组成
- 第一行的第一个单词是用来说明消息类型，其他单词都是具体的消息数据
- 相邻的单词应该只有一个空格符分隔
- 每一行以 CR-LF 作为结束符

Some of the choices made above are weaker in real-life protocols. For example

实际上，上面的一些考虑在真实的协议中是远远不够的。比如

- Message types could be case-insensitive. This just requires mapping message type strings down to lower-case before decoding
- An arbitrary amount of white space could be left between words. This just adds a little more complication, compressing white space
- Continuation characters such as '\' can be used to break long lines over several lines. This starts to make processing more complex

- Just a '\n' could be used as line terminator, as well as '\r\n'. This makes recognising end of line a bit harder

- 消息类型类型应该是大小不写敏感的。 对于表示消息类型的字符串，我们就需要在解码前将它小写化
- 单词与单词间的多余空白字符应该被丢弃掉。当然这会增加一些代码的复杂度，去处理压缩空白符
- 像“\”这样的续行符应该被使用，它能将一个大的长句子分隔成几行。从这里开始，程序渐渐变得更复杂了
- 像“\n”这样的字符也应该能被解析为换行符，就和“\r\n”一样。这个就让辨识解析程序的结束符更为复杂了

All of these variations exist in real protocols. Cumulatively, they make the string processing just more complex than in our case.

所有以上的变化和考虑都会在真实使用的协议中出现。渐渐地，这些会导致实际的字符处理程序比我们的这个例子复杂。

client request	server response
send "DIR"	send list of files, one per line terminated by a blank line
send "CD <dir>"	change dir send "ERROR" if failed send "OK"
send "PWD"	send current working directory

Server code

服务器代码

```
/* FTP Server
 */
package main

import (
	"fmt"
	"net"
	"os"
)

const (
	DIR = "DIR"
	CD  = "CD"
	PWD = "PWD"
)

func main() {

	service := "0.0.0.0:1202"
	tcpAddr, err := net.ResolveTCPAddr("tcp", service)
	checkError(err)

	listener, err := net.ListenTCP("tcp", tcpAddr)
	checkError(err)

	for {
		conn, err := listener.Accept()
		if err != nil {
			continue
```

```
                }
                go handleClient(conn)
        }
}

func handleClient(conn net.Conn) {
        defer conn.Close()

        var buf [512]byte
        for {
                n, err := conn.Read(buf[0:])
                if err != nil {
                        conn.Close()
                        return
                }

                s := string(buf[0:n])
                // decode request
         if s[0:2] == CD {
                        chdir(conn, s[3:])
                } else if s[0:3] == DIR {
                        dirList(conn)
                } else if s[0:3] == PWD {
                        pwd(conn)
                }

        }
}
```

```
func chdir(conn net.Conn, s string) {
	if os.Chdir(s) == nil {
		conn.Write([]byte("OK"))
	} else {
		conn.Write([]byte("ERROR"))
	}
}

func pwd(conn net.Conn) {
	s, err := os.Getwd()
	if err != nil {
		conn.Write([]byte(""))
		return
	}
	conn.Write([]byte(s))
}

func dirList(conn net.Conn) {
	defer conn.Write([]byte("\r\n"))

	dir, err := os.Open(".")
	if err != nil {
		return
	}

	names, err := dir.Readdirnames(-1)
	if err != nil {
		return
	}
```

```
        for _, nm := range names {
                conn.Write([]byte(nm + "\r\n"))
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Client code

客户端代码

```
/* FTPClient
 */
package main

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

// strings used by the user interface
```

```
const (
	uiDir  = "dir"
	uiCd   = "cd"
	uiPwd  = "pwd"
	uiQuit = "quit"
)

// strings used across the network
const (
	DIR = "DIR"
	CD  = "CD"
	PWD = "PWD"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "host")
		os.Exit(1)
	}

	host := os.Args[1]

	conn, err := net.Dial("tcp", host+":1202")
	checkError(err)

	reader := bufio.NewReader(os.Stdin)
	for {
		line, err := reader.ReadString('\n')
		// lose trailing whitespace
```

```
		line = strings.TrimRight(line, " \t\r\n")
			if err != nil {
				break
			}

			// split into command + arg
		strs := strings.SplitN(line, " ", 2)
			// decode user request
		switch strs[0] {
			case uiDir:
				dirRequest(conn)
			case uiCd:
				if len(strs) != 2 {
					fmt.Println("cd <dir>")
					continue
				}
				fmt.Println("CD \"", strs[1], "\"")
				cdRequest(conn, strs[1])
			case uiPwd:
				pwdRequest(conn)
			case uiQuit:
				conn.Close()
				os.Exit(0)
			default:
				fmt.Println("Unknown command")
			}
		}
}
```

```
func dirRequest(conn net.Conn) {
        conn.Write([]byte(DIR + " "))

        var buf [512]byte
        result := bytes.NewBuffer(nil)
        for {
                // read till we hit a blank line
         n, _ := conn.Read(buf[0:])
                result.Write(buf[0:n])
                length := result.Len()
                contents := result.Bytes()
                if string(contents[length-4:]) == "\r\n\r\n" {
                        fmt.Println(string(contents[0 : length-4]))
                        return
                }
        }
}

func cdRequest(conn net.Conn, dir string) {
        conn.Write([]byte(CD + " " + dir))
        var response [512]byte
        n, _ := conn.Read(response[0:])
        s := string(response[0:n])
        if s != "OK" {
                fmt.Println("Failed to change dir")
        }
}

func pwdRequest(conn net.Conn) {
```

```
        conn.Write([]byte(PWD))
        var response [512]byte
        n, _ := conn.Read(response[0:])
        s := string(response[0:n])
        fmt.Println("Current dir \"" + s + "\"")
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

State

状态

Applications often make use of state information to simplify what is going on. For example

应用程序经常保存状态消息来简化下面要做的事情，比如

- Keeping file pointers to current file location
- Keeping current mouse position
- Keeping current customer value.

- 保存当前文件路径的文件指针状态
- 保存当前的鼠标位置状态
- 保存当前的客户值状态

In a distributed system, such state information may be kept in the client, in the server, or in both.

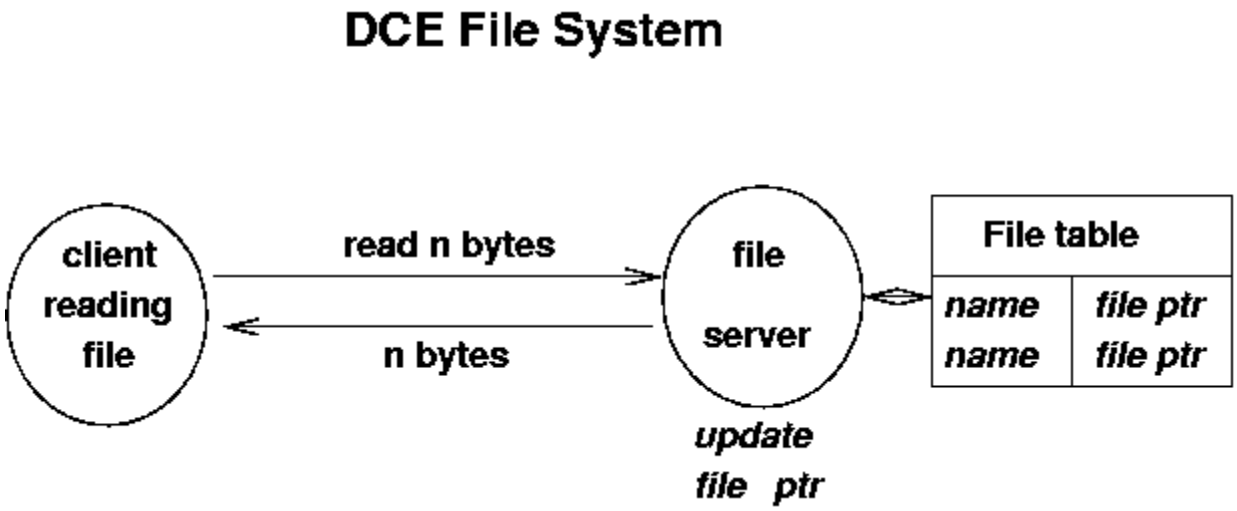
在分布式的系统中，这样的状态消息可能是保存在客户端，服务器，也可能两边都保存。

The important point is to whether one process is keeping state information about *itself* or about the *other* process. One process may keep as much state information about itself as it wants, without causing any problems. If it needs to keep information about the state of the other process, then problems arise: the process' actual knowledge of the state of the other may become incorrect. This can be caused by loss of messages (in UDP), by failure to update, or by s/w errors.

最重要的一点是，进程是否需要保存 *自身进程* 或者*其他进程* 的状态消息。一个进程保存再多自己的状态信息，也不会引发其他问题。如果需要保存其他进程的状态消息，这个问题就复杂了：当前保存的其他进程的状态消息和实际的状态消息可能是不一致的。这可能会引起消息丢失（在 UDP 中）、更新失败、或者 s/w 错误等。

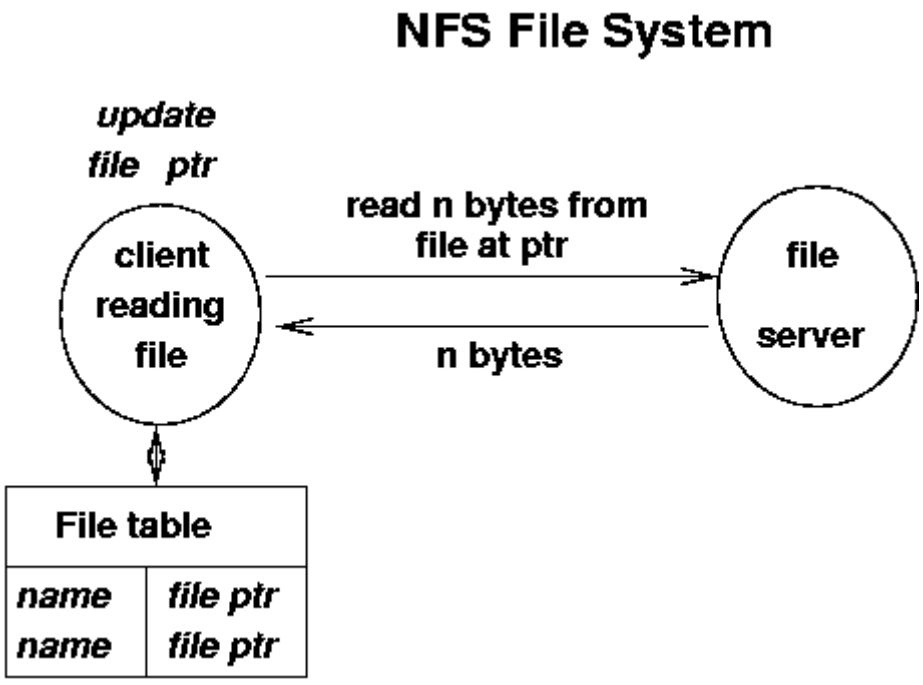
An example is reading a file. In single process applications the file handling code runs as part of the application. It maintains a table of open files and the location in each of them. Each time a read or write is done this file location is updated. In the DCE file system, the file server keeps track of a client's open files, and where the client's file pointer is. If a message could get lost (but DCE uses TCP) these could get out of synch. If the client crashes, the server must eventually timeout on the client's file tables and remove them.

一个例子就是读取文件。在单个进程中，文件处理代码是应用程序的一部分。它维持一个表，表中包含所有打开的文件和文件指针位置。每次文件读写的时候，文件指针位置就会更新。在数据通信（DCE）文件系统中，文件系统必须追踪客户端打开了哪些文件，客户端的文件指针在哪。如果一个消息丢失了（但是 DCE 是使用 TCP 的），这些状态消息就不能保持同步了。如果出现客户端崩溃了，服务器就必须对这个表触发超时并删除。



In NFS, the server does not maintain this state. The client does. Each file access from the client that reaches the server must open the file at the appropriate point, as given by the client, to perform the action.

在 NFS 文件系统中，服务器并没有保存这个状态消息，而是有客户端保存的。客户端每次在服务器进行的读取文件操作必须能在准确的文件位置打开文件，而这个文件位置是由客户端提供的，从而才能进行后续的操作。



If the server maintains information about the client, then it must be able to recover if the client crashes. If information is not saved, then on each transaction the client must transfer sufficient information for the server to function.

如果由服务器保持客户端的状态消息，服务器必须在客户端崩溃的时候进行修复。如果服务器没有储存状态消息，那么客户端的每次事务交互都需要提供足够的消息来让服务器进行操作。

If the connection is unreliable, then additional handling must be in place to ensure that the two do not get out of synch. The classic example is of bank account transactions where the messages get lost. A transaction server may need to be part of the client-server system.

如果连接是不可靠的，那么必须要有额外的处理程序来确保双方没有失去同步。一个消息丢失的典型例子就是银行账号交易系统。交易系统是客户端与服务器交互的一部分。

Application State Transition Diagram

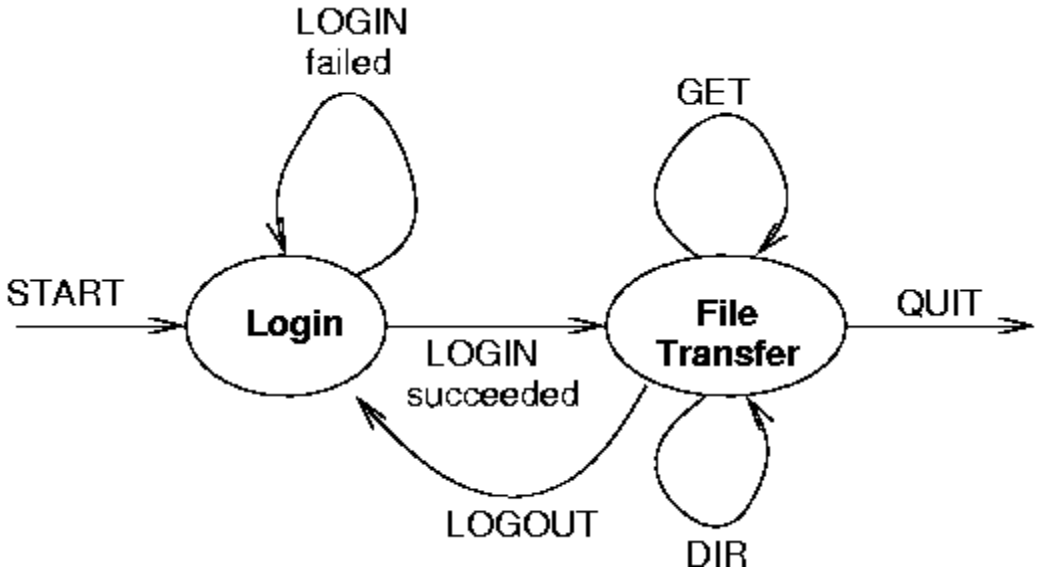
应用状态转换图

A state transition diagram keeps track of the current state of an application and the changes that move it to new states.

一个状态转换图清晰说明了当前应用的状态和进入到新的状态需要的转换。

Example: file transfer with login:

例如：带登陆功能的文件传输：



This can also be expressed as a table

这个也可以使用一个表来表示

Current state	Transition	Next state
login	login failed	login
	login succeeded	file transfer
file transfer	dir	file transfer
	get	file transfer
	logout	login
	quit	-

Client state transition diagrams

客户端状态转换图

The client state diagram must follow the application diagram. It has more detail though: it *writes* and then *reads*

客户端状态转换图就和应用转换图一样。不同的就是要注意更多细节：它包含有*读* 和 *写*操作

Current state	Write	Read	Next state
login	LOGIN name password	FAILED	login
		SUCCEEDED	file transfer
file transfer	CD dir	SUCCEEDED	file transfer
		FAILED	file transfer
	GET filename	#lines + contents	file transfer
		ERROR	file transfer
	DIR	#files + filenames	file transfer
		ERROR	file transfer
	quit	none	quit
logout	none	login	

Server state transition diagrams

服务器状态转换图

The server state diagram must also follow the application diagram. It also has more detail: it *reads* and then *writes*

服务器状态转换图也和应用转换图一样。不同的就是也要注意更多细节：它包含有 *读* 和 *写* 操作

Current state	Read	Write	Next state
login	LOGIN name password	FAILED	login
		SUCCEEDED	file transfer
file transfer	CD dir	SUCCEEDED	file transfer
		FAILED	file transfer
	GET filename	#lines + contents	file transfer

		ERROR	file transfer
	DIR	#files + filenames	file transfer
		ERROR	file transfer
	quit	none	quit
logout	none	login	

Server pseudocode

服务器伪代码

```
state = login
while true
    read line
    switch (state)
        case login:
            get NAME from line
            get PASSWORD from line
            if NAME and PASSWORD verified
                write SUCCEEDED
                state = file_transfer
            else
                write FAILED
                state = login
        case file_transfer:
            if line.startsWith CD
                get DIR from line
                if chdir DIR okay
                    write SUCCEEDED
                    state = file_transfer
```

```
            else
                write FAILED
                state = file_transfer
        ...
```

We don't give the actual code for this server or client since it is pretty straightforward.

由于这个伪代码已经足够清晰了，所以我们并不用给出具体的代码了。

Summary

总结

Building any application requires design decisions before you start writing code. For distributed applications you have a wider range of decisions to make compared to standalone systems. This chapter has considered some of those aspects and demonstrated what the resultant code might look like.

任何应用程序在开始编写前都需要详尽的设计。开发一个分布式的系统比开发一个独立系统需要更宽广的视野和思维来做决定和思考。这一章已经考虑到了一些这样的问题，并且展示了最终代码的大致样子。

Managing character sets and encodings

字符集和编码

There are many languages in use throughout the world, and they use many different character sets. There are also many ways of encoding character sets into binary formats of bytes. This chapter considers some of the issues in this.

全世界在用的语言众多，他们使用许多不同的字符集。同时有很多方法对字符集进行二进制字节编码。本章将讨论几个关于编码的问题。

Introduction

引言

Once upon a time there was EBCDIC and ASCII... Actually, it was never that simple and has just become more complex over time. There is light on the horizon, but some estimates are that it may be 50 years before we all live in the daylight on this!

以前用 EBCDIC 和 ASCII 编码，(别看只有两种编码)，但事情从来没有简单过，恰恰相反变得越来越复杂了。但据推测,编码简化就像(黎明前)地平线上闪过了一道光，但要等到天亮还得 50 年。

Early computers were developed in the english-speaking countries of the US, the UK and Australia. As a result of this, assumptions were made about the language and character sets in use. Basically, the Latin alphabet was used, plus numerals, punctuation characters and a few others. These were then encoded into bytes using ASCII or EBCDIC.

早期计算机是从美国、英国、澳大利亚这些英语国家发展起来的，结果计算机字符集就以这些国家使用的语言和字符进行设计，大体上，也就是拉丁字母，加上数字、标点和别的字符。他们使用 ASCII 或 EBCDIC 进行编码。

The character-handling mechanisms were based on this: text files and I/O consisted of a sequence of bytes, with each byte representing a single character. String comparison could be done by matching

corresponding bytes; conversions from upper to lower case could be done by mapping individual bytes, and so on.

字符处理的机制是基于此的：文本文件和基于字节序列的基本输入输出，每个字节代表一个单独的字符。字符串比较可以通过对比相对应的字节实现，字符串的大小写转换可以通过单个字节的操作完成，等等。

There are about 6,000 living languages in the world (3,000 of them in Papua New Guinea!). A few languages use the "english" characters but most do not. The Romanic languages such as French have adornments on various characters, so that you can write "j'ai arrêté", with two differently accented vowels. Similarly, the Germanic languages have extra characters such as 'ß'. Even UK English has characters not in the standard ASCII set: the pound symbol '£' and recently the euro '€'

世界上现存约有 6000 种语言（居然有 3000 种在巴布亚新几内亚）。一小部分使用英文字符，但更多的则不是。想法文这样的拉丁语系语言还会有字符修饰符号，所以你可以用两种不同的重读元音来拼写“j'ai arrêté”。同样地，德语也有像'ß'这样的字符，甚至是英式英语也会有不在 ASCII 编码中的字符：英镑和欧元('£'和 '€')

But the world is not restricted to variations on the Latin alphabet. Thailand has its own alphabet, with words looking like this: "ภาษาไทย". There are many other alphabets, and Japan even has two, Hiragana and Katagana.

但是世界上的语言并不严格局限在拉丁字母中，泰国有它自己的字母，像这样：“ภาษาไทย”。还有许多的字母形式像是日文，他居然有两种，平假文和片假文。

There are also the hierographic languages such as Chinese where you can write "百度一下，你就知道".

还有一些象形文字，比如汉语，你可以这样写“百度一下，你就知道”。

It would be nice from a technical viewpoint if the world just used ASCII. However, the trend is in the opposite direction, with more and more users demanding that software use the language that they are familiar with. If you build an application that can be run in different countries then users will demand

that it uses their own language. In a distributed system, different components of the system may be used by users expecting different languages and characters.

用理工科的眼光看，世界上只有 ASCII 一种编码就清静了。但实际正是相反的趋势，越来越多的人需要计算机软件中使用自己熟悉的语言。如果你的软件可以在不同的国家运行，那你的用户就需要软件使用他们自己的语言。在分布式的系统中，使用不同的系统模块的人可能希望不同的语言和字符。

Internationalisation (i18n) is how you write your applications so that they can handle the variety of languages and cultures. *Localisation* (l10n) is the process of customising your internationalised application to a particular cultural group.

国际化(i18n)是指你的应用怎么处理不同的语言和文化。*本地化*(l10n)是说你怎么把国际化的应用适配成小群体使用。

i18n and l10n are big topics in themselves. For example, they cover issues such as colours: while white means "purity" in Western cultures, it means "death" to the Chinese and "joy" to Egyptians. In this chapter we just look at issues of character handling.

国际化和本地化各自都是一个很大的课题。举个例子，关于颜色的话题：白色在西方表示纯洁，在中国表示死亡，在埃及表示喜悦。在这章中我们只关注字符的处理。

Definitions

定义

It is important to be careful about exactly what part of a text handling system you are talking about. Here is a set of definitions that have proven useful.

我们所关心的是系统处理你所表述的内容，十分重要。下面是有人做的一套行之有效的定义方法。

Character

字符

A character is a "unit of information that roughly corresponds to a grapheme (written symbol) of a natural language, such as a letter, numeral, or punctuation mark" (Wikipedia). A character is "the smallest component of written language that has a semantic value" (Unicode). This includes letters such as 'a' and 'À' (or letters in any other language), digits such as '2', punctuation characters such as ',' and various symbols such as the English pound currency symbol '£'.

字符是 " 自然语言中用符号表示信息的单位，比如字母、数字、标点 " （维基百科），字符是有价值的最小书写单位（Unicode）这就包括了 a 和 A，或其他语言字符，也包括数字 2 和标点',',还有像英镑这样的字符。

A character is some sort of abstraction of any actual symbol: the character 'a' is to any written 'a' as a Platonic circle is to any actual circle. The concept of character also includes control characters, which do not correspond to natural language symbols but to other bits of information used to process texts of the language.

字符实际上是符号的抽象组合，也就是说 a 代表了所有手写的 a，有点像柏拉图圆也是圆的关系。原则上字符也包括控制字符，也就是实际中不存在只是为了处理语言的格式用的。

A character does not have any particular appearance, although we use the appearance to help recognise the character. However, even the appearance may have to be understood in a context: in mathematics, if you see the symbol π (pi) it is the character for the ratio of circumference to radius of a circle, while if you are reading Greek text, it is the sixteenth letter of the alphabet: "πϱοσ" is the greek word for "with" and has nothing to do with 3.14159...

字符本身并不没有特定形状，只是我们通过形状来识别它。即使如此，我们也要联系上下文才能理解：数学中，如果你看到 π (pi)这个字符，它表示圆周率，但是如果你读希腊文，它只是 16 个字母；"πϱοσ"是希腊词语“with”，这个和 3.14159 没有半点关系。

Character repertoire/character set

字符体系和字符集

A character repertoire is a set of distinct characters, such as the Latin alphabet. No particular ordering is assumed. In English, although we say that 'a' is earlier in the alphabet than 'z', we wouldn't say that 'a' is less than 'z'. The "phone book" ordering which puts "McPhee" before "MacRea" shows that "alphabetic ordering" isn't critical to the characters.

字符集就是一个不同的且唯一的字符的集合，像拉丁字母，不需要指定顺序。在英语中，尽管我们说 a 是在 z 的前面，但我们不说 a 比 z 要小。电话联系人的排序方式里，McPhee 在 MacRea 的前面说明了字母排序不是严格的按字符的顺序。

A repertoire specifies the names of the characters and often a sample of how the characters might look. e.g the letter 'a' might look like 'a', '*a*' or '**a**'. But it doesn't force them to look like that - they are just samples. The repertoire may make distinctions such as upper and lower case, so that 'a' and 'A' are different. But it may regard them as the same, just with different sample appearances. (Just like some programming languages treat upper and lower as different - e.g. Go - but some don't e.g. Basic.). On the other hand, a repertoire might contain different characters with the same sample appearance: the repertoire for a Greek mathematician would have two different characters with appearance π. This is also called a noncoded character set.

字符体系就是字名和字形的结合，比如，a 可能写成 a,*a* 或 **a**，但这不是强制的，他们只是样本。字符体系可能区分大小写，所以 a 和 A 是不同的。但他们的意思可能是一样的，就算是长的不一样。（有点像编程语言对待大小写，有的大小写敏感，比如 Go 语言，有的就是一样的，比如 Basic。）。另一方面，字符系统可能包括长的一样但意义不同的：希腊字母的数学符号就有两个意思，比如 pai。他们也被叫成无法编码的字符集。

Character code

字符编码

A character code is a mapping from characters to integers. The mapping for a character set is also called a coded character set or code set. The value of each character in this mapping is often called a code point. ASCII is a code set. The codepoint for 'a' is 97 and for 'A' is 65 (decimal).

字符编码是字符到整数的映射。一个字符集的映射也被称为一个编码字符集或字符集。这个映射中的每个字符的值通常被称为一个编码（code point)。 ASCII 也是一个字符集，'a'的编码是 97，'A'是 65（十进制）。

The character code is still an abstraction. It isn't yet what we will see in text files, or in TCP packets. However, it is getting close. as it supplies the mapping from human oriented concepts into numerical ones.

字符编码仍然是一个抽象的概念。它不是我们可以看到的文件或者 TCP 的包。不过，确和这两个概念很像，它就是一种把人抽象出来的概念转化为数字的映射关系。

Character encoding

字符编码

To communicate or store a character you need to encode it in some way. To transmit a string, you need to encode all characters in the string. There are many possible encodings for any code set.

字符的交互（传输）和存储都要以某种方式编码。要发送一个字符串，你需要将字符串中的所有字符进行编码。每种字符集都有很多的编码方案。

For example, 7-bit ASCII code points can be encoded as themselves into 8-bit bytes (an octet). So ASCII 'A' (with codepoint 65) is encoded as the 8-bit octet 01000001. However, a different encoding would be to use the top bit for parity checking e.g. with odd parity ASCII 'A" would be the octet 11000001. Some protocols such as Sun's XDR use 32-bit word-length encoding. ASCII 'A' would be encoded as 00000000 00000000 0000000 01000001.

例如，7 位字节 ASCII 编码可以转换成 8 位字节（8 进制）。所以，ASCII 的'A'（编码值 65）可以被编码为 8 进制的 01000001。不过，另一种不同的编码方式对最高位别有用途，如奇偶校验，带有奇校验的 ASCII 编码“A”将是这个 8 进制数 11000001。还有一些协议，如 Sun 的 XDR，使用 32 位字长编码 ASCII 编码。所以，'A'将被编码为 00000000 00000000000000001000001。

The character encoding is where we function at the programming level. Our programs deal with encoded characters. It obviously makes a difference whether we are dealing with 8-bit characters with or without parity checking, or with 32-bit characters.

字符编码是在程序应用层面使用的。应用程序处理编码的字符时，是否带包含奇偶校验处理 8 位字符或 32 位字符，显然有很大的差别。

The encoding extends to strings of characters. A word-length even parity encoding of "ABC" might be 10000000 (parity bit in high byte) 0100000011 (C) 01000010 (B) 01000001 (A in low byte). The comments about the importance of an encoding apply equally strongly to strings, where the rules may be different.

把字符编码扩展到字符串。一个字节宽、带有奇偶校验的“ABC”编码为 10000000（高位奇偶校验）0100000011（C）01000010（B）01000001（A 在低位）。对于编码在字符串上的讨论也很重要，虽然编码规则可能不同。

Transport encoding

编码传输

A character encoding will suffice for handling characters within a single application. However, once you start sending text *between* applications, then there is the further issue of how the bytes, shorts or words are put on the wire. An encoding can be based on space-and hence bandwidth-saving techniques such as **zip**'ping the text. Or it could be reduced to a 7-bit format to allow a parity checking bit, such as **base64**.

某个应用程序的字符编码只要内部能处理字符串就足够了。然而，一旦你需要在不同应用程序*之间*交互，那怎么编码可就成了需要进一步讨论问题了：字节、字符、字是怎么传输的。字符编码可能有很多空白字符（待商议），从而可以使用如 **zip 算法**对文本进行压缩，从而节省带宽。或者，它可以减少到 7 位字节，奇偶校验位，使用 **base64 编码**来代替。

If we do know the character and transport encoding, then it is a matter of programming to manage characters and strings. If we don't know the character or transport encoding then it is a matter of

guesswork as to what to do with any particular string. There is no convention for files to signal the character encoding.

如果我们知道的字符编码和传输编码，那么问题就成了如何通过编程处理字符和字符串；如果我们不知道字符编码和传输编码，那么如何猜到某个特定字符串的编码方式就是大问题。因为没有约定发送文件的字符编码

There *is* however a convention for signalling encoding in text transmitted across the internet. It is simple: the header of a text message contains information about the encoding. For example, an HTTP header can contain lines such as

不过，在互联网上传输文本的编码是有约定的。很简单：文本消息头包含的编码信息。例如，HTTP 报头可以包含这么几行，如

```
Content-Type: text/html; charset=ISO-8859-4

Content-Encoding: gzip
```

which says that the character set is ISO 8859-4 (corresponding to certain countries in Europe) with the default encoding, but then **gzip**ed. The second part - content encoding - is what we are referring to as "transfer encoding" (IETF RFC 2130).

上面是说，将字符集是 ISO 8859-4（对应到欧洲的某些国家）作为默认编码，然后用 **gzip** 压缩。内容类型的第二部分就是我们指的是“传输编码”（IETF RFC2130）。

But how do you read this information? Isn't it encoded? Don't we have a chicken and egg situation? Well, no. The convention is that such information is given in ASCII (to be precise, US ASCII) so that a program can read the headers and then adjust its encoding for the rest of the document.

但是，怎么读懂这个信息呢？它没有编码？这不就是先有鸡还是先有蛋的问题么？嗯，不是的。按照惯例，这样的信息使用 ASCII 编码（准确地说，美国 ASCII），所以程序可以读取 headers，然后适配其文档的其余部分的编码。

ASCII

ASCII 编码

ASCII has the repertoire of the English characters plus digits, punctuation and some control characters. The code points for ASCII are given by the familiar table

ASCII 字符集包含的英文字符、数字，标点符号和一些控制字符。 下面这张熟悉的表给出了 ASCII 字符编码值

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH	101	65	41	A
002	2	02	STX	102	66	42	B
003	3	03	ETX	103	67	43	C
004	4	04	EOT	104	68	44	D
005	5	05	ENQ	105	69	45	E
006	6	06	ACK	106	70	46	F

007	7	07	BEL	'\a'	107	71	47	G
010	8	08	BS	'\b'	110	72	48	H
011	9	09	HT	'\t'	111	73	49	I
012	10	0A	LF	'\n'	112	74	4A	J
013	11	0B	VT	'\v'	113	75	4B	K
014	12	0C	FF	'\f'	114	76	4C	L
015	13	0D	CR	'\r'	115	77	4D	M
016	14	0E	SO		116	78	4E	N
017	15	0F	SI		117	79	4F	O
020	16	10	DLE		120	80	50	P
021	17	11	DC1		121	81	51	Q
022	18	12	DC2		122	82	52	R
023	19	13	DC3		123	83	53	S
024	20	14	DC4		124	84	54	T

025	21	15	NAK	125	85	55	U	
026	22	16	SYN	126	86	56	V	
027	23	17	ETB	127	87	57	W	
030	24	18	CAN	130	88	58	X	
031	25	19	EM	131	89	59	Y	
032	26	1A	SUB	132	90	5A	Z	
033	27	1B	ESC	133	91	5B	[	
034	28	1C	FS	134	92	5C	\	'\\'
035	29	1D	GS	135	93	5D	]	
036	30	1E	RS	136	94	5E	^	
037	31	1F	US	137	95	5F	_	
040	32	20	SPACE	140	96	60	`	
041	33	21	!	141	97	61	a	
042	34	22	"	142	98	62	b	
043	35	23	#	143	99	63	c	

044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q

062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	|
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

The most common encoding for ASCII uses the code points as 7-bit bytes, so that the encoding of 'A' for example is 65.

最常见的 ASCII 编码使用 7 位字节，所以 A 的码是 65。

This set is actually *US ASCII*. Due to European desires for accented characters, some punctuation characters are omitted to form a minimal set, ISO 646, while there are "national variants" with suitable European characters. The page http://www.cs.tut.fi/~jkorpela/chars.html by Jukka Korpela has more information for those interested. We shall not need these variants though.

这个字符集是实际的*美国 ASCII*。鉴于欧洲需要处理重音字符，于是省略一些标点字符，形成一个最小的字符集，ISO 646，同时有合适的欧洲本国字符的“国家变种字符集”。有兴趣的可以看看 Jukka Korpel 的这个网页 http://www.cs.tut.fi/～jkorpela/ chars.html。当然我们并不需要这些变种。

ISO 8859

ISO 8859 字符集

Octets are now the standard size for bytes. This allows 128 extra code points for extensions to ASCII. A number of different code sets to capture the repertoires of various subsets of European languages are the ISO 8859 series. ISO 8859-1 is also known as Latin-1 and covers many languages in western Europe, while others in this series cover the rest of Europe and even Hebrew, Arabic and Thai. For example, ISO 8859-5 includes the Cyrillic characters of countries such as Russia, while ISO 8859-8 includes the Hebrew alphabet.

8 进制是字节的标准长度。这使得 ASCII 可以有 128 个额外的编码。 ISO 8859 系列的字符集可以包含众多的欧洲语言字符集。。 ISO 8859-1 也被称为 Latin-1，覆盖了许多在西欧国家的语言，同时这一系列的其他字符集包括欧洲其他国家，甚至希伯来语，阿拉伯语和泰语。例如，ISO 8859-5 包括使用斯拉夫语字符的俄罗斯等，而 ISO 8859-8 则包含希伯来文字母。

The standard encoding for these character sets is to use their code point as an 8-bit value. For example, the character 'Á' in ISO 8859-1 has the code point 193 and is encoded as 193. All of the ISO

8859 series have the bottom 128 values identical to ASCII, so that the ASCII characters are the same in all of these sets.

这些字符集使用8进制作为标准的编码格式。例如,在ISO 8859-1字符' 'Á'的字符编码为193,同时被编码为193。所有的ISO 8859系列前128个保持和ASCII相同的值，所以，ASCII字符在所有这些集合都是相同的。

The HTML specifications used to recommend the ISO 8859-1 character set. HTML 3.2 was the last one to do so, and after that HTML 4.0 recommended Unicode. In 2010 Google made an estimate that of the pages it sees, about 20% were still in ISO 8859 format while 20% were still in ASCII ("Unicode nearing 50% of the web" http://googleblog.blogspot.com/2010/01/unicode-nearing-50-of-web.html).

HTML 语言规范曾经推荐 ISO 8859-1 字符集，不过 HTML3.2 之后的规范就不再推荐，4.0 开始推荐 Unicode 编码。2010 年 Google 通过它抓取的网页做出了一个估算，20%的网页使用 ISO 8859 编码，20%使用 ASCII（unicode 接近 50%，http://googleblog.blogspot.com/2010/01/unicode-nearing-50-of-web.html）

Unicode

Unicode 编码

Neither ASCII nor ISO 8859 cover the languages based on hieroglyphs. Chinese is estimated to have about 20,000 separate characters, with about 5,000 in common use. These need more than a byte, and typically two bytes has been used. There have been many of these two-byte character sets: Big5, EUC-TW, GB2312 and GBK/GBX for Chinese, JIS X 0208 for Japanese, and so on. These encodings are generally not mutually compatable.

ASCII 和 ISO 8859 都不能覆盖象形文字。中文大约有 20000 个独立的字符，其中 5000 个常用字符。这些字符需要不知一个字节，基本上双字节都会被用上。也有一些多字节的编码：中文的 Big5, EUC-TW, GB2312 和 GBK/GBX，日文的 JIS X 0208，等等。这些编码通常是不兼容的

Unicode is an embracing standard character set intended to cover all major character sets in use. It includes European, Asian, Indian and many more. It is now up to version 5.2 and has over 107,000 characters. The number of code points now exceeds 65,536, that is. more than 2^16. This has implications for character encodings.

Unincode 是一个受到拥护的字符集编码标准,旨在统一主要使用的编码。它包含了欧洲文字、亚洲文字和印度文字等。现在 Unicode 已经到了 5.2 的版本，包含 107,0000 个字符。编码字符超过 65536，也就是 2^16。这已经覆盖了整个编码。

The first 256 code points correspond to ISO 8859-1, with US ASCII as the first 128. There is thus a backward compatability with these major character sets, as the code points for ISO 8859-1 and ASCII are exactly the same in Unicode. The same is not true for other character sets: for example, while most of the Big5 characters are also in Unicode, the code points are not the same. The page http://moztw.org/docs/big5/table/unicode1.1-obsolete.txt contains one example of a (large) table mapping from Big5 to Unicode.

(Unicode 编码)前 256 个编码对应 ISO 8859-1，同时前 128 个也是美式 ASCII 编码。所以主流的编码都是相互兼容的，ISO 8859-1、ASCII 和 Unicode 是一样的。对其他字符集则不一定正确：例如，虽然 Big5 编码也在 Unicode 中，但他们的编码值并不相同。http://moztw.org/docs/big5/table/unicode1.1-obsolete.txt 这个页面就是证明：一张 Big5 到 Unicode 的大的映射表。

To represent Unicode characters in a computer system, an encoding must be used. The encoding UCS is a two-byte encoding using the code point values of the Unicode characters. However, since there are now too many characters in Unicode to fit them all into 2 bytes, this encoding is obsolete and no longer used. Instead there are:

为了在计算机系统中表示 Unicode 字符，必须使用一个编码方案。UCS 编码使用两个字节来编码一个字符值。然而，Unicode 现在有太多的字符需要对应到双字节的编码。以下方案是替代原来陈旧的编码方案的：

- UTF-32 is a 4-byte encoding, but is not commonly used, and HTML 5 warns explicitly against using it
- UTF-16 encodes the most common characters into 2 bytes with a further 2 bytes for the "overflow", with ASCII and ISO 8859-1 having the usual values
- UTF-8 uses between 1 and 4 bytes per character, with ASCII having the usual values (but not ISO 8859-1)
- UTF-7 is used sometimes, but is not common

- UTF-32 使用 4 个字节编码，但是已经不在推荐，HTML5 甚至严重警告反对使用
- UTF-16 是最常见的，它通过溢出两个字节来处理 ASCII 和 ISO 8859-1 外的字符
- UTF-8 每个字符使用 1 到 4 个字节，所以 ASCII 值不变，但 ISO 8859-1 的值会变化
- UTF-7 有时会用到，但不常见

UTF-8, Go and runes

UTF-8, Go 语言和 runes

UTF-8 is the most commonly used encoding. Google estimates that 50% of the pages that it sees are encoded in UTF-8. The ASCII set has the same encoding values in UTF-8, so a UTF-8 reader can read text consisting of just ASCII characters as well as text from the full Unicode set.

UTF - 8 是最常用的编码。谷歌估计它抓取的网页有 50%使用 UTF-8 编码。ASCII 字符集具有相同的在 UTF-8 中编码值相同，所以 UTF-8 的读取方法可以用 Unicode 字符集读取一个 ASCII 字符组成的网页。

Go uses UTF-8 encoded characters in its strings. Each character is of type **rune**. This is a alias for **int32** as a Unicode character can be 1, 2 or 4 bytes in UTF-8 encoding. In terms of characters, a string is an array of runes.

Go 语言使用 UTF-8 编码字符串。每个字符类型都是 **rune**。**rune** 是 **int32** 的一个别名，因为 Unicode 编码可以是 1,2 或 4 个字节。字符和字符串其实都是一个 runes 的数组

A string is also an array of bytes, but you have to be careful: only for the ASCII subset is a byte equal to a character. All other characters occupy two, three or four bytes. This means that the length of a string in characters (runes) is generally not the same as the length of its byte array. They are only equal when the string consists of ASCII characters only.

Unicode 中一个字符串其实是一个字节数组，但是你要注意：只有 ASCII 这个字符集是一个字节等于一个字符。所有其他字符占用 2 个，三个或四个字节。这意味着，一个字符串的长度（runes）通常是不一样的长度的字节数组。他们只有在全是 ASCII 字符是才相同。

The following program fragment illustrates this. If we take a UTF-8 string and test its length, you get the length of the underlying byte array. But if you cast the string to an array of runes []rune then you get an array of the Unicode code points which is generally the number of characters:

下面的程序片段可以说明这些。如果我们使用 utf-8 来检验它的长度，你只会得到它字符层面的长度。但如果你把字符串转换成 rues 数组[]rune，你就等到一个 Unicode 编码的数组：

```
str := "百度一下，你就知道"

println("String length", len([]rune(str)))

println("Byte length", len(str))
```

prints

输出为

```
```

```
String length 9

Byte length 27
```

UTF-8 client and server

UTF-8 编码的客户端和服务端

Possibly surprisingly, you need do nothing special to handle UTF-8 text in either the client or the server. The underlying data type for a UTF-8 string in Go is a byte array, and as we saw just above, Go looks after encoding the string into 1, 2, 3 or 4 bytes as needed. The length of the string is the length of the byte array, so you write any UTF-8 string by writing the byte array.

可能令人惊讶的是，无论是客户端或服务器你不需要对 utf-8 的文本做任何特殊的处理。UTF-8 字符串的数据类型是一个字节数组，如上所示。Go 语言自动处理编码后的字符串是 1，2，3 或 4 个字节。所以 utf-8 的字符串你可以随便写。

Similarly to read a string, you just read into a byte array and then cast the array to a string using **string([]byte)**. If Go cannot properly decode bytes into Unicode characters, then it gives the Unicode Replacement Character \uFFFD. The length of the resulting byte array is the length of the legal portion of the string.

类似于读取字符串，只要读入一个字节数组，然后使用 **string([]byte)**将数组转换成一个字符串。如果 Go 语言不能正确解码，将字节转换为 Unicode 字符，那么它给使用 Unicode 替换字符\uFFFD。生成的字节数组的长度是有效字符串的长度。

So the clients and servers given in earlier chapters work perfectly well with UTF-8 encoded text.

所以前面章节中提到的客户端和服务端使用 uft-8 编码表现的很好

ASCII client and server

ASCII 编码的客户端和服务器

The ASCII characters have the same encoding in ASCII and in UTF-8. So ordinary UTF-8 character handling works fine for ASCII characters. No special handling need to be done.

ASCII 字符的 ASCII 编码和 UTF-8 编码的值相同，所以普通的 UTF-8 字符能正常处理 ASCII 字符，不需要做任何特殊的处理。

UTF-16 and Go

Go 语言和 utf-16

UTF-16 deals with arrays of short 16-bit unsigned integers. The package **utf16** is designed to manage such arrays. To convert a normal Go string, that is a UTF-8 string, into UTF-16, you first extract the code points by coercing it into a **[]rune** and then use **utf16.Encode** to produce an array of type **uint16**.

utf-16 编码可以用 16 位字节无符号整形数组处理。 **utf16** 包就是用来处理这样的字串的。将一个 Go 语言的 utf-8 正常编码的字串转换 utf-16 的编码，你应先将字串转换成**[]rune**rune 数组，然后使用 **utf16.Encode** 生成一个 **uint16** 类型的数组。

Similarly, to decode an array of unsigned short UTF-16 values into a Go string, you use **utf16.Decode** to convert it into code points as type **[]rune** and then to a string. The following code fragment illustrates this

同样，解码一个无符号短整型的 utf-16 数组成一个 Go 字符串，你需要 **utf16.Decode** 将编码转换成**[]rune** ，然后才能改成一个字符串。如下面的代码所示：

```
str := "百度一下，你就知道"

runes := utf16.Encode([]rune(str))
```

```
ints := utf16.Decode(runes)


str = string(ints)
```

These type conversions need to be applied by clients or servers as appropriate, to read and write 16-bit short integers, as shown below.

类型转换需要客户端和服务器在合适的时机读取和写入 16 位的整数，如下图所示。（……图呢？）

Little-endian and big-endian

Little-endian 和 big-endian

Unfortunately, there is a little devil lurking behind UTF-16. It is basically an encoding of characters into 16-bit short integers. The big question is: for each short, how is it written as two bytes? The top one first, or the top one second? Either way is fine, as long as the receiver uses the same convention as the sender.

然而，UTF-16 编码潜藏着一个小的恶魔。它基本上是一个 16 字节字符编码。最大的问题是：每一个短字，是如何拼写的？高位在前还是高位在后？无论哪种方式，只要是发生器和接收器约定好就可以。

Unicode has addressed this with a special character known as the BOM (byte order marker). This is a zero-width non-printing character, so you never see it in text. But its value 0xfffe is chosen so that you can tell the byte-order:

Unicode 通过一个特殊字节标记了寻址方式，这个字节就被称为 BOM（字节顺序标记）。这是一个零宽度非打印字符，所以你永远不会在文本中看到它。但是它通过 0xFFFE 的值，可以告诉你编码的顺序

- In a big-endian system it is FF FE
- In a little-endian system it is FE FF

- 在 big-endian 系统中，它是 FF FE
- 在 little-endian 系统中，它是 FE FF

Text will sometimes place the BOM as the first character in the text. The reader can then examine these two bytes to determine what endian-ness has been used.

有时 BOM 会位于文本的第一个字符。文本被读入是可以检查，以确定使用的是那种系统。

UTF-16 client and server

UTF-16 编码的客户端和服务器

Using the BOM convention, we can write a server that prepends a BOM and writes a string in UTF-16 as

根据 BOM 的约定，服务器可以预先设置 BOM 来表示 utf-16,如下

```
/* UTF16 Server
 */
package main
import (
        "fmt"
        "net"
```

```
        "os"

        "unicode/utf16"

)

const BOM = '\ufffe'

func main() {

        service := "0.0.0.0:1210"

        tcpAddr, err := net.ResolveTCPAddr("tcp", service)

        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)

        checkError(err)

        for {

                conn, err := listener.Accept()

                if err != nil {

                        continue
```

```
                }

                str := "j'ai arrêté"

                shorts := utf16.Encode([]rune(str))

                writeShorts(conn, shorts)

                conn.Close() // we're finished

 }

}

func writeShorts(conn net.Conn, shorts []uint16) {

        var bytes [2]byte

        // send the BOM as first two bytes

 bytes[0] = BOM >> 8

        bytes[1] = BOM & 255

        _, err := conn.Write(bytes[0:])

        if err != nil {

                return
```

```
            }

            for _, v := range shorts {

                        bytes[0] = byte(v >> 8)

                        bytes[1] = byte(v & 255)

                        _, err = conn.Write(bytes[0:])

                        if err != nil {

                                    return

                        }

            }

}

func checkError(err error) {

            if err != nil {

                        fmt.Println("Fatal error ", err.Error())

                        os.Exit(1)
```

```
        }

}
```

while a client that reads a byte stream, extracts and examines the BOM and then decodes the rest of the stream is

但客户端读取一个字节流，提取并检查 BOM 时解码该流的其余部分的。

```
/* UTF16 Client

 */

package main

import (

        "fmt"

        "net"

        "os"

        "unicode/utf16"

)

const BOM = '\ufffe'
```

```
func main() {

        if len(os.Args) != 2 {

                fmt.Println("Usage: ", os.Args[0], "host:port")

                os.Exit(1)

        }

        service := os.Args[1]

        conn, err := net.Dial("tcp", service)

        checkError(err)

        shorts := readShorts(conn)

        ints := utf16.Decode(shorts)

        str := string(ints)

        fmt.Println(str)

        os.Exit(0)

}
```

```
func readShorts(conn net.Conn) []uint16 {

        var buf [512]byte

        // read everything into the buffer

 n, err := conn.Read(buf[0:2])

        for true {

                m, err := conn.Read(buf[n:])

                if m == 0 || err != nil {

                        break

                }

                n += m

        }

        checkError(err)

        var shorts []uint16

        shorts = make([]uint16, n/2)

        if buf[0] == 0xff && buf[1] == 0xfe {
```

```
            // big endian
 for i := 2; i < n; i += 2 {
                    shorts[i/2] = uint16(buf[i])<<8 + uint16(buf[i+1])
            }
} else if buf[1] == 0xff && buf[0] == 0xfe {
            // little endian
 for i := 2; i < n; i += 2 {
                    shorts[i/2] = uint16(buf[i+1])<<8 + uint16(buf[i])
            }
} else {
            // unknown byte order
 fmt.Println("Unknown order")
}
return shorts
```

```
}

func checkError(err error) {

        if err != nil {

                fmt.Println("Fatal error ", err.Error())

                os.Exit(1)

        }

}
```

Unicode gotcha's

Unicode 的疑难杂症

This book is not about i18n issues. In particular we don't want to delve into the arcane areas of Unicode. But you should know that Unicode is not a simple encoding and there are many complexities. For example, some earlier character sets used *non-spacing* characters, particularly for accents. This was brought into Unicode, so you can produce accented characters in two ways: as a single Unicode character, or as a pair of non-spacing accent plus non-accented character. For example, U+04D6 CYRILLIC CAPITAL LETTER IE WITH BREVE is a single character. It is equivalent to U+0415 CYRILLIC CAPITAL LETTER IE combined with the breve accent U+0306 COMBINING BREVE. This makes string comparison difficult on occassions. The Go specification does not at present address such issues.

这本书不是有关国际化问题。特别是，我们不想钻研的神秘的 Unicode。但是你应该知道，Unicode 不是一个简单的编码，也有很多的复杂的地方。例如，一些早期的字符集用*非空格*

字符, 尤其是重音字符。这些重音字符要转换成 Unicode 可以用两种办法：作为一个 Unicode 字符，或作为一个非空格字符和非重音字符的组合。例如， U+04D6 CYRILLIC CAPITAL LETTER IE WITH BREVE 是一个字符。这是相当于 U+0415 CYRILLIC CAPITAL LETTER IE 和 U+0306 加上 BREVE.。这使得字符串比较有时变得困难了。 GO 规范确目前没有对这个问题过深研究。

ISO 8859 and Go

ISO 8859 编码和 Go 语言

The ISO 8859 series are 8-bit character sets for different parts of Europe and some other areas. They all have the ASCII set common in the low part, but differ in the top part. According to Google, ISO 8859 codes account for about 20% of the web pages it sees.

ISO 8859 系列字符集都是 8 位字符集，他们为欧洲不同地区和其他一些地方设计。他们有相同的 ASCII 并且都在地位，但高位不同。据谷歌估计，ISO 8859 编码了尽 20%的网页。

The first code, ISO 8859-1 or Latin-1, has the first 256 characters in common with Unicode. The encoded value of the Latin-1 characters is the same in UTF-16 and in the default ISO 8859-1 encoding. But this doesn't really help much, as UTF-16 is a 16-bit encoding and ISO 8859-1 is an 8-bit encoding. UTF-8 is a 8-bit encoding, but it uses the top bit to signal extra bytes, so only the ASCII subset overlaps for UTF-8 and ISO 8859-1. So UTF-8 doesn't help much either.

第一个编码字符集，ISO 8859-1 或叫做 Latin-1，前 256 个字符和 Unicode 相同。 Latin-1 字符的 utf-16 和 ISO 8859-1 有相同的编码。但是，这并不真的有用，因为 UTF-16 是一个 16 位的编码字符集而 ISO 8859-1 是 8 位编码。 UTF-8 是一种 8 位编码，但是高位用来表示更多的字符，所以只有 ASCII 的一部分是 utf-8 和 ISO 8859-1 相同，所以 UTF-8 并没有多大实际用途（都是 8 位的）。

But the ISO 8859 series don't have any complex issues. To each character in each set corresponds a unique Unicode character. For example, in ISO 8859-2, the character "latin capital letter I with ogonek" has ISO 8859-2 code point 0xc7 (in hexadecimal) and corresponding Unicode code point of

U+012E. Transforming either way between an ISO 8859 set and the corresponding Unicode characters is essentially just a table lookup.

但 ISO8859 系列没有任何复杂的问题。每一组中的每个字符对应一个唯一的 Unicode 字符。例如，在 ISO 8859-2 中的字符“latin capital letter I with ogonek”在 ISO 8859-2 是 0xc7（十六进制），对应的 Unicode 的 U+012E。 ISO 8859 字符集和 Unicode 字符集之间转换其实只是一个表查找。

The table from ISO 8859 code points to Unicode code points could be done as an array of 256 integers. But many of these will have the same value as the index. So we just use a map of the different ones, and those not in the map take the index value.

这个从 ISO 8859 到 Unicode 的查找表，可以用一个 256 的数组完成。因为，许多字符索引相同。因此，我们只需要一个标注不同索引的映射就可以。

For ISO 8859-2 a portion of the map is

ISO 8859-2 的映射为

```
var unicodeToISOMap = map[int] uint8 {
    0x12e: 0xc7,
    0x10c: 0xc8,
    0x118: 0xca,
    // plus more
}
```

and a function to convert UTF-8 strings to an array of ISO 8859-2 bytes is

从 utf-8 转换成 ISO 8859-2 的函数

```
/* Turn a UTF-8 string into an ISO 8859 encoded byte array

*/

func unicodeStrToISO(str string) []byte {

        // get the unicode code points

        codePoints := []int(str)

        // create a byte array of the same length

        bytes := make([]byte, len(codePoints))

        for n, v := range(codePoints) {

                // see if the point is in the exception map

                iso, ok := unicodeToISOMap[v]

                if !ok {
```

```
                        // just use the value

                        iso = uint8(v)

                    }

                    bytes[n] = iso

                }

                return bytes

}
```

In a similar way you cacn change an array of ISO 8859-2 bytes into a UTF-8 string:

同样你可以将 ISO 8859-2 转换为 utf-8

```
var isoToUnicodeMap = map[uint8] int {

    0xc7: 0x12e,

    0xc8: 0x10c,

    0xca: 0x118,

    // and more
```

```
}

func isoBytesToUnicode(bytes []byte) string {

	codePoints := make([]int, len(bytes))

	for n, v := range(bytes) {

		unicode, ok :=isoToUnicodeMap[v]

		if !ok {

			unicode = int(v)

		}

		codePoints[n] = unicode

	}

	return string(codePoints)

}
```

These functions can be used to read and write UTF-8 strings as ISO 8859-2 bytes. By changing the mapping table, you can cover the other ISO 8859 codes. Latin-1, or ISO 8859-1, is a special case - the

exception map is empty as the code points for Latin-1 are the same in Unicode. You could also use the same technique for other character sets based on a table mapping, such as Windows 1252.

这些函数可以用来将 ISO 8859-2 当作 UTF-8 来读写。通过改变映射表，可以覆盖其他的 ISO 8859 字符集合。Latin-1 字符集（ISO 8859-1）是一个特殊的情况：地图映射为空，因为字符在 Latin-1 和 Unicode 中编码相同。同样的方法，你也可以使用其他字符集构建映射表，如 Windows1252。

Other character sets and Go

其他字符集和 Go 语言

There are very, very many character set encodings. According to Google, these generally only have a small use, which will hopefully decrease even further in time. But if your software wants to capture all markets, then you may need to handle them.

还有非常非常多的字符集编码。据谷歌称，这些字符集通常只有很少地方使用，所以可能用的会更少。但是，如果你的软件要占据所有市场，那么你可能需要对这些字符集进行处理。

In the simplest cases, a lookup table will suffice. But that doesn't always work. The character coding ISO 2022 minimised character set sizes by using a finite state machine to swap code pages in and out. This was borrowed by some of the Japanese encodings, and makes things very complex.

在最简单的情况下，查找表就够了。但是，这样也不是总是奏效。ISO 2022 字符编码方案通过……。这是从日本某写编码中借用来个，相当复杂。

Go does not at present give any language or package support for these other character sets. So you either avoid their use, fail to talk to applications that do use them, or write lots of your own code!

Go 语言目前在语言本身和包文件上支持其他字符集。所以，你要么避免使用其他字符集，虽然没法和用这些字符集的程序共存，要么自己动手写很多代码。

Conclusion

总结

There hasn't been much code in this chapter. Instead, there have been some of the concepts of a very complex area. It's up to you: if you want to assume everyone speaks US English then the world is simple. But if you want your applications to be usable by the rest of the world, then you need to pay attention to these complexities.

这一章没有什么代码，却有几个非常复杂的概念。当然，也取决于你：你要只满足说美式英语的人，那问题就简单了；要是你的应用也要让其他人可用，那你就要在这个复杂的问题上花点精力了。

Security

安全

Introduction

简介

Although the internet was originally designed as a system to withstand atacks by hostile agents, it developed in a co-operative environment of relatively trusted entities. Alas, those days are long gone. Spam mail, denial of service attacks, phishing attempts and so on are indicative that anyone using the internet does so at their own risk.

尽管互联网最初被设计为可以承受敌对代理攻击的系统，但它过去一直是在一个可信的实体和合作的环境中发展起来的。可惜现在已经时过境迁。垃圾邮件，拒绝服务攻击，网络钓鱼这些东西使得每一个上网者都需要自行承担风险。

Applications have to be built to work correctly in hostile situations. "correctly" no longer means just getting the functional aspects of the program correct, but also means ensuring privacy and integrity of data transferred, access only to legitimate users and other issues.

应用程序应当在复杂的互联网环境中仍然可以正确工作。“正确”不光意味着程序功能的正确，同时还意味着要确保数据传输过程中的保密性和完整性，甚至只允许合法用户进行访问和其它问题。

This of course makes your programs much more complex. There are *difficult* and *subtle* computing problems involved in making applications secure. Attempts to do it yourself (such as making up your own encryption libraries) are usually doomed to failure. Instead, you need to make use of libraries designed by security professionals

这自然使得编程更加复杂。在构建安全应用程序的过程中，会出现很*复杂*和*微妙*的问题。如果你想自己这样做(如实现一个自有安全库)，通常都会以失败而告终。相反，你需要使用安全专家设计的安全库。

ISO security architecture

ISO 安全架构

The ISO OSI (open systems interconnect) seven-layer model of distributed systems is well known and is repeated in this figure:

ISO OSI（开放系统互连）七层模型分布式系统是众所周知的，在此重复如下图：

What is less well known is that ISO built a whole series of documents upon this architecture. For our purposes here, the most important is the ISO Security Architecture model, ISO 7498-2.

少为人知的是，ISO 在此架构的基础上建立了一系列完整的文档。而我们这里最重要的是 ISO 安全体系结构模型（ISO Security Architecture model）ISO 7498-2。

Functions and levels

功能层次

The principal functions required of a security system are

主要的安全系统功能

- Authentication - proof of identity
- Data integrity - data is not tampered with
- Confidentiality - data is not exposed to others
- Notarization/signature
- Access control
- Assurance/availability

- 认证 - 身份校验文件

- 数据完整性 - 数据不被篡改
- 保密 - 数据不能暴露给他人
- 公证/签名
- 访问控制
- 保证/可用性

These are required at the following levels of the OSI stack:

必须的 OSI 协议栈

- Peer entity authentication (3, 4, 7)
- Data origin authentication (3, 4, 7)
- Access control service (3, 4, 7)
- Connection confidentiality (1, 2, 3, 4, 6, 7)
- Connectionless confidentiality (1, 2, 3, 4, 6, 7)
- Selective field confidentiality (6, 7)
- Traffic flow confidentiality (1, 3, 7)
- Connection integrity with recovery (4, 7)
- Connection integrity without recovery (4, 7)
- Connection integrity selective field (7)
- Connectionless integrity selective field (7)
- Non-repudiation at origin (7)
- Non-repudiation of receipt (7)

- 对等实体认证 (3, 4, 7)
- 数据源认证 (3, 4, 7)
- 访问控制服务 (3, 4, 7)
- 连接保密 (1, 2, 3, 4, 6, 7)
- 无连接的保密 (1, 2, 3, 4, 6, 7)
- 选择性字段的保密 (6, 7)
- 传输保密 (1, 3, 7)

- 恢复连接的完整性 (4, 7)
- 不可恢复连接的完整性 (4, 7)
- 选择字段连接完整性 (7)
- 选择字段的无连接完整性 (7)
- 接受源 (7)
- 接受回执 (7)

Mechanisms

机制

- Peer entity authentication
    - encryption
    - digital signature
    - authentication exchange
- Data origin authentication
    - encryption
    - digital signature
- Access control service
    - access control lists
    - passwords
    - capabilities lists
    - labels
- Connection confidentiality
    - ecryption
    - routing control
- Connectionless confidelity
    - encryption
    - routing control
- Selective field confidelity
    - encryption

- Traffic flow confidelity
    - encryption
    - traffic padding
    - routing control
- Connection integrity with recovery
    - encryption
    - data integrity
- Connection integrity without recovery
    - encryption
    - data integrity
- Connection integrity selective field
    - encryption
    - data integrity
- Connectionless integrity
    - encryption
    - digital signature
    - data integrity
- Connectionless integrity selective field
    - encryption
    - digital signature
    - data integrity
- Non-repudiation at origin
    - digital signature
    - data integrity
    - notarisation
- Non-repudiation of receipt
    - digital signature
    - data integrity
    - notarisation

- 对等实体认证
    - 加密
    - 数字签名
    - 交换验证
- 数据源认证
    - 加密
    - 数字签名
- 访问控制服务
    - 访问控制列表
    - 密码
    - 范围列表
    - 等级
- 连接保密
    - 密码
    - 路由控制
- 无连接的保密
    - 密码
    - 路由控制
- 选择性字段的保密
    - 密码
- 传输保密
    - 密码
    - 传输填充
    - 路由控制
- 恢复连接的完整性
    - 密码
    - 数据完整性
- 不可恢复连接的完整性
    - 密码

    - 数据完整性
- 选择字段连接完整性
    - 密码
    - 数据完整性
- 无连接完整性
    - 密码
    - 数字签名
    - 数据完整性
- 选择字段的无连接完整性
    - 密码
    - 数字签名
    - 数据完整性
- 接受源
    - 数字签名
    - 数据完整性
    - 公正
- 接受回执
    - 数字签名
    - 数据完整性
    - 公正

Data integrity

数据完整性

Ensuring data integrity means supplying a means of testing that the data has not been tampered with. Usually this is done by forming a simple number out of the bytes in the data. This process is called *hashing* and the resulting number is called a *hash* or *hash value*.

确保数据的完整性意味着要提供一个数据未被篡改的测试方法。通常是通过字节数据生成一个简单的数字。这个操作被称为 *hashing*，结果数字成为 *hash* 或者 *hash 值*。

A naive hashing algorithm is just to sum up all the bytes in the data. However, this still allows almost any amount of changing the data around and still preserving the hash values. For example, an attacker could just swap two bytes. This preserves the hash value, but could end up with you owing someone $65,536 instead of $256.

有一个幼稚的 hash 算法是将数据所有的字节进行总和。然而，这却仍然允许数据在保留 hash 值不变的情况下对数据进行任意改变。例如，攻击者只需要交换两个字节。这样 hash 值没有改变，但结果可能是你本来欠别人 256 美元却变成了 65535 美元。

Hashing algorithms used for security purposes have to be "strong", so that it is very difficult for an attacker to find a different sequence of bytes with the same hash value. This makes it hard to modify the data to the attacker's purposes. Security researchers are constantly testing hash algorithms to see if they can break them - that is, find a simple way of coming up with byte sequences to match a hash value. They have devised a series of *cryptographic* hashing algorithms which are believed to be strong.

用于安全目的的 hash 算法必须很“强”，这样攻击者才很难在保留相同的 hash 值时找到一个不同的字节序列。这样攻击者才很难修改数据以达到目的。安全研究人员不断测试是否能攻破 hash 算法 - 寻找一个简单方法，得到一个字节序列来匹配某个 hash 值。他们设计了一系列被认为很强的*加密* hash 算法.

Go has support for several hashing algorithms, including MD4, MD5, RIPEMD-160, SHA1, SHA224, SHA256, SHA384 and SHA512. They all follow the same pattern as far as the Go programmer is concerned: a function **New** (or similar) in the appropriate package returns a **Hash** object from the **hash** package.

Go 支持几个 hash 算法,包括 MD4, MD5, RIPEMD-160, SHA1, SHA224, SHA256, SHA384 and SHA512。它们都尽可能按照 Go 程序员关注的，遵循相同的模式：在适当的包中定义 **New** 或类似的方法，返回一个 **hash** 包中的 **Hash** 对象。

A **Hash** has an **io.Writer**, and you write the data to be hashed to this writer. You can query the number of bytes in the hash value by **Size** and the hash value by **Sum**.

一个 Hash 结构体拥有一个 io.Writer 接口，你可以通过 writer 方法写入被 hash 的数据.你可以通过 Size 方法获取 hash 值的长度，Sum 方法返回 hash 值。

A typical case is MD5 hashing. This uses the **md5** package. The hash value is a 16 byte array. This is typically printed out in ASCII form as four hexadecimal numbers, each made of 4 bytes. A simple program is

MD5 算法是个典型的例子。使用 **md5** 包，hash 值是一个 16 位的数组。通常以 ASCII 形式输出四个由 4 字节组成的十六进制数。程序如下

```
/* MD5Hash
 */

package main

import (
	"crypto/md5"
	"fmt"
)

func main() {
	hash := md5.New()
	bytes := []byte("hello\n")
	hash.Write(bytes)
	hashValue := hash.Sum(nil)
	hashSize := hash.Size()
	for n := 0; n < hashSize; n += 4 {
		var val uint32
		val = uint32(hashValue[n])<<24 +
			uint32(hashValue[n+1])<<16 +
```

```
                                uint32(hashValue[n+2])<<8 +
                                uint32(hashValue[n+3])
                        fmt.Printf("%x ", val)
                }
                fmt.Println()
}
```

which prints "b1946ac9 2492d234 7c6235b4 d2611184"

输出 "b1946ac9 2492d234 7c6235b4 d2611184"

A variation on this is the HMAC (Keyed-Hash Message Authentication Code) which adds a key to the hash algorithm. There is little change in using this. To use MD5 hashing along with a key, replace the call to **New** by

在此基础上的一个变化是 HMAC(Keyed-Hash Message Authentication Code)，它给 hash 算法增加了一个 key。使用时略有不同。要和 key 一起使用 MD5 算法时，可以通过以下形式替换 **New**

```
func NewMD5(key []byte) hash.Hash
```

Symmetric key encryption

key 对称加密

There are two major mechanisms used for encrypting data. The first uses a single key that is the same for both encryption and decryption. This key needs to be known to both the encrypting and the decrypting agents. How this key is transmitted between the agents is not discussed.

数据加密有两种机制。第一种方式是在加密和解密时都使用同一个 key。加密方和解密方都需要知道这个 key。此处如何在这两者之间传输这个 key。

As with hashing, there are many encryption algorithms. Many are now known to have weaknesses, and in general algorithms become weaker over time as computers get faster. Go has support for several symmetric key algorithms such as Blowfish and DES.

目前有很多使用 hash 算法的加密算法。其中很多都有弱点，而且随着时间的推移，计算机越来越快，通用 hash 算法变得越来越弱。Go 已经支持好几个对称加密算法，如 Blowfish 和 DES。

The algorithms are *block* algorithms. That is they work on blocks of data. If you data is not aligned to the block size, then you will have to pad it with extra blanks at the end.

这些算法都是 *block* 算法。因为它们必须基于数据块(block)。如果你的数据不匹配 block 的大小，那就必须在最后使用空格来填充多余的空间。

Each algorith is represented by a **Cipher** object. This is created by **NewCipher** in the appropriate package, and takes the symmetric key as parameter.

每个算法都被表示为一个 **Cipher** 对象。可通过在相应的包中使用对称 key 作为参数调用 **NewCipher** 方法来创建该对象。

Once you have a cipher, you can use it to encrypt and decrypt blocks of data. The blocks have to be 8-bit blocks for Blowfish. A program to illustrate this is

创建 cipher 对象后，就能通过它加密和解密数据块。Blowfish 需要 8 位的 block，详见以下程序

```
/* Blowfish
 */

package main

import (
```

```
        "bytes"
        "code.google.com/p/go.crypto/blowfish"
        "fmt"
)

func main() {
        key := []byte("my key")
        cipher, err := blowfish.NewCipher(key)
        if err != nil {
                fmt.Println(err.Error())
        }
        src := []byte("hello\n\n\n")
        var enc [512]byte

        cipher.Encrypt(enc[0:], src)

        var decrypt [8]byte
        cipher.Decrypt(decrypt[0:], enc[0:])
        result := bytes.NewBuffer(nil)
        result.Write(decrypt[0:8])
        fmt.Println(string(result.Bytes()))
}
```

Blowfish is not in the Go 1 distribution. Instead it is on the http://code.google.com/p/ site. You have to install it by running "go get" in a directory where you have source that needs to use it.

Blowfish 不在 GO 1 中，而是在 http://code.google.com/p/中。你可以在需要使用它的源码目录下运行“go get”进行安装。

Public key encryption

公钥加密

Public key encryption and decryption requires *two* keys: one to encrypt and a second one to decrypt. The encryption key is usually made public in some way so that anyone can encrypt messages to you. The decryption key must stay private, otherwise everyon would be able to decrypt those messages! Public key systems aer asymmetric, with different keys for different uses.

公钥加密和解密需要*两个* key：一个用来加密，另一个用来解密。加密 key 通常是公开的，这样任何人都可以给你发送加密数据。解密 key 必须保密，否则任何人都可以解密数据。公钥系统是非对称的，不同的 key 有不同的用途。

There are many public key encryption systems supported by Go. A typical one is the RSA scheme.

Go 支持很多公钥加密系统，RSA 就是一个典型的例子。

A program generating RSA private and public keys is

下面是一个生成 RSA 公钥和私钥的程序

```
/* GenRSAKeys
 */

package main

import (
        "crypto/rand"
        "crypto/rsa"
        "crypto/x509"
        "encoding/gob"
        "encoding/pem"
        "fmt"
```

```
        "os"
)

func main() {
        reader := rand.Reader
        bitSize := 512
        key, err := rsa.GenerateKey(reader, bitSize)
        checkError(err)

        fmt.Println("Private key primes", key.Primes[0].String(), key.Primes[1].String())
        fmt.Println("Private key exponent", key.D.String())

        publicKey := key.PublicKey
        fmt.Println("Public key modulus", publicKey.N.String())
        fmt.Println("Public key exponent", publicKey.E)

        saveGobKey("private.key", key)
        saveGobKey("public.key", publicKey)

        savePEMKey("private.pem", key)
}

func saveGobKey(fileName string, key interface{}) {
        outFile, err := os.Create(fileName)
        checkError(err)
        encoder := gob.NewEncoder(outFile)
        err = encoder.Encode(key)
        checkError(err)
        outFile.Close()
```

```
}

func savePEMKey(fileName string, key *rsa.PrivateKey) {

	outFile, err := os.Create(fileName)
	checkError(err)

	var privateKey = &pem.Block{Type: "RSA PRIVATE KEY",
		Bytes: x509.MarshalPKCS1PrivateKey(key)}

	pem.Encode(outFile, privateKey)

	outFile.Close()
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

The program also saves the certificates using **gob** serialisation. They can be read back by this program:

程序通过 **gob** 序列化保存证书。也可以取回证书:

```
/* LoadRSAKeys
 */

package main
```

```
import (
	"crypto/rsa"
	"encoding/gob"
	"fmt"
	"os"
)

func main() {
	var key rsa.PrivateKey
	loadKey("private.key", &key)

	fmt.Println("Private key primes", key.Primes[0].String(), key.Primes[1].String())
	fmt.Println("Private key exponent", key.D.String())

	var publicKey rsa.PublicKey
	loadKey("public.key", &publicKey)

	fmt.Println("Public key modulus", publicKey.N.String())
	fmt.Println("Public key exponent", publicKey.E)
}

func loadKey(fileName string, key interface{}) {
	inFile, err := os.Open(fileName)
	checkError(err)
	decoder := gob.NewDecoder(inFile)
	err = decoder.Decode(key)
	checkError(err)
	inFile.Close()
```

```
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

X.509 certificates

X.509 证书

A Public Key Infrastructure (PKI) is a framework for a collection of public keys, along with additional information such as owner name and location, and links between them giving some sort of approval mechanism.

公钥基础架构（PKI）是一个公钥集合框架，它连同附加信息，如所有者名称和位置，以及它们之间的联系提供了一些审批机制。

The principal PKI in use today is based on X.509 certificates. For example, web browsers use them to verify the identity of web sites.

目前主要使用的 PKI 是就是基于 X.509 证书的。例如浏览器使用它验证站点的身份。

An example program to generate a self-signed X.509 certificate for my web site and store it in a .cer file is

下面的程序是为自己的站点生成自签名 X.509 证书并保存到一个.cer 文件中

```
/* GenX509Cert
 */
```

```
package main

import (
	"crypto/rand"
	"crypto/rsa"
	"crypto/x509"
	"crypto/x509/pkix"
	"encoding/gob"
	"encoding/pem"
	"fmt"
	"math/big"
	"os"
	"time"
)

func main() {
	random := rand.Reader

	var key rsa.PrivateKey
	loadKey("private.key", &key)

	now := time.Now()
	then := now.Add(60 * 60 * 24 * 365 * 1000 * 1000 * 1000) // one year
	template := x509.Certificate{
		SerialNumber: big.NewInt(1),
		Subject: pkix.Name{
			CommonName:   "jan.newmarch.name",
			Organization: []string{"Jan Newmarch"},
```

```
                },
                //            NotBefore: time.Unix(now, 0).UTC(),
                //            NotAfter:    time.Unix(now+60*60*24*365, 0).UTC(),
                NotBefore: now,
                NotAfter:    then,

                SubjectKeyId: []byte{1, 2, 3, 4},
                KeyUsage:         x509.KeyUsageCertSign | x509.KeyUsageKeyEncipherment
| x509.KeyUsageDigitalSignature,

                BasicConstraintsValid: true,
                IsCA:                          true,
                DNSNames:                    []string{"jan.newmarch.name", "localhost"},
        }
        derBytes, err := x509.CreateCertificate(random, &template,
                &template, &key.PublicKey, &key)
        checkError(err)

        certCerFile, err := os.Create("jan.newmarch.name.cer")
        checkError(err)
        certCerFile.Write(derBytes)
        certCerFile.Close()

        certPEMFile, err := os.Create("jan.newmarch.name.pem")
        checkError(err)
        pem.Encode(certPEMFile, &pem.Block{Type: "CERTIFICATE", Bytes: derBytes})
        certPEMFile.Close()

        keyPEMFile, err := os.Create("private.pem")
```

```
	checkError(err)
	pem.Encode(keyPEMFile, &pem.Block{Type: "RSA PRIVATE KEY",
		Bytes: x509.MarshalPKCS1PrivateKey(&key)})
	keyPEMFile.Close()
}

func loadKey(fileName string, key interface{}) {
	inFile, err := os.Open(fileName)
	checkError(err)
	decoder := gob.NewDecoder(inFile)
	err = decoder.Decode(key)
	checkError(err)
	inFile.Close()
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

This can then be read back in by

下面这个程序可以读取这个证书

```
/* GenX509Cert
 */

package main
```

```
import (
	"crypto/x509"
	"fmt"
	"os"
)

func main() {
	certCerFile, err := os.Open("jan.newmarch.name.cer")
	checkError(err)
	derBytes := make([]byte, 1000) // bigger than the file
	count, err := certCerFile.Read(derBytes)
	checkError(err)
	certCerFile.Close()

	// trim the bytes to actual length in call
	cert, err := x509.ParseCertificate(derBytes[0:count])
	checkError(err)

	fmt.Printf("Name %s\n", cert.Subject.CommonName)
	fmt.Printf("Not before %s\n", cert.NotBefore.String())
	fmt.Printf("Not after %s\n", cert.NotAfter.String())

}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
```

```
        }
}
```

TLS

TLS

Encryption/decryption schemes are of limited use if you have to do all the heavy lifting yourself. The most popular mechanism on the internet to give support for encrypted message passing is currently TLS (Transport Layer Security) which was formerly SSL (Secure Sockets Layer).

如果自己实现所有的细节，加解密的方案在使用上是有限制的。当前互联网上最流行的加密消息传输方案是 TLS（Transport Layer Security 安全传输层），其前身为 SSL（Secure Sockets Layer 安全套接字层）。

In TLS, a client and a server negotiate identity using X.509 certificates. One this is complete, a secret key is invented between them, and all encryption/decryption is done using this key. The negotiation is relatively slow, but once complete a faster private key mechanism is used.

在 TLS 中，客户端和服务器之间使用 X.509 证书进行身份验证。身份验证完成后，两者之间会生成一个密钥，所有的加密和解密过程都使用这个密钥。虽然客户端和服务端协商的过程相对较慢，但一旦完成就会使用一个较快的私钥机制。

A server is

服务器端程序

```
/* TLSEchoServer
 */
package main

import (
```

```
        "crypto/rand"
        "crypto/tls"
        "fmt"
        "net"
        "os"
        "time"
)

func main() {

        cert, err := tls.LoadX509KeyPair("jan.newmarch.name.pem", "private.pem")
        checkError(err)
        config := tls.Config{Certificates: []tls.Certificate{cert}}

        now := time.Now()
        config.Time = func() time.Time { return now }
        config.Rand = rand.Reader

        service := "0.0.0.0:1200"

        listener, err := tls.Listen("tcp", service, &config)
        checkError(err)
        fmt.Println("Listening")
        for {
                conn, err := listener.Accept()
                if err != nil {
                        fmt.Println(err.Error())
                        continue
                }
```

```
                fmt.Println("Accepted")
                go handleClient(conn)
        }
}

func handleClient(conn net.Conn) {
        defer conn.Close()

        var buf [512]byte
        for {
                fmt.Println("Trying to read")
                n, err := conn.Read(buf[0:])
                if err != nil {
                        fmt.Println(err)
                }
                _, err2 := conn.Write(buf[0:n])
                if err2 != nil {
                        return
                }
        }
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The server works with the following client:

与服务器端程序对应的客户端程序:

```
/* TLSEchoClient
 */
package main

import (
        "fmt"
        "os"
        "crypto/tls"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]

        conn, err := tls.Dial("tcp", service, nil)
        checkError(err)

        for n := 0; n < 10; n++ {
                fmt.Println("Writing...")
                conn.Write([]byte("Hello " + string(n+48)))

                var buf [512]byte
                n, err := conn.Read(buf[0:])
                checkError(err)
```

```
                fmt.Println(string(buf[0:n]))
        }
        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Conclusion

结论

Security is a huge area in itself, and in this chapter we have barely touched on it. However, the major concepts have been covered. What has not been stressed is how much security needs to be built into the design phase: security as an afterthought is nearly always a failure.

安全本身是一个巨大的领域，在本章中，我们几乎没有触及，但已经覆盖了主要的概念。尚未强调的是，需要在设计阶段如何考虑安全构建：亡羊补牢几乎是没有意义的。

HTTP

关于 HTTP

Introduction

简介

The World Wide Web is a major distributed system, with millions of users. A site may become a Web host by running an HTTP server. While Web clients are typically users with a browser, there are many other "user agents" such as web spiders, web application clients and so on.

万维网是一个庞大的，拥有数以百万计用户的分布式系统。网站就是一个运行着 HTTP 服务器的 Web 主机。而 Web 客户端通常是浏览器用户，当然也还有许多其他的“用户”，如网络蜘蛛，Web 应用程序客户端等。

The Web is built on top of the HTTP (Hyper-Text Transport Protocol) which is layered on top of TCP. HTTP has been through three publically available versions, but the latest - version 1.1 - is now the most commonly used.

Web 使用的 HTTP（超文本传输协议）是基于 TCP 协议的。HTTP 有三个公开可用的版本，目前最常用的是最新的版本 1.1。

In this chapter we give an overview of HTTP, followed by the Go APIs to manage HTTP connections.

本章首先对 HTTP 进行概述，然后介绍如何通过 Go API 管理 HTTP 连接。

Overview of HTTP

HTTP 概述

URLs and resources

URL 和资源

URLs specify the location of a *resource*. A resource is often a static file, such as an HTML document, an image, or a sound file. But increasingly, it may be a dynamically generated object, perhaps based on information stored in a database.

URL 指定*资源*的位置。资源通常是 HTML 文档、图片、声音文件这样的静态文件，但越来越多的资源是动态生成的对象，比如根据数据库信息生成。

When a user agent requests a resource, what is returned is not the resource itself, but some *representation* of that resource. For example, if the resource is a static file, then what is sent to the user agent is a copy of the file.

“用户”请求资源时，返回的并不是资源本身，而是资源的*代表*。如果资源是静态文件，那么返回给用户的就是文件的一个副本。

Multiple URLs may point to the same resource, and an HTTP server will return appropriate representations of the resource for each URL. For example, an company might make product information available both internally and externally using different URLs for the same product. The internal representation of the product might include information such as internal contact officers for the product, while the external representation might include the location of stores selling the product.

不同的 URL 可以指向相同的资源，HTTP 服务器会给每个 URL 返回适当的代表。例如，针对同一个产品，某公司可以使用不同的 URL 给本地和外地的用户查看其产品信息，本地用户可以看到本地产品联系人这类内容，而外地用户看到的内容则包括产品销售门店的地址等等。

This view of resources means that the HTTP protocol can be fairly simple and straightforward, while an HTTP server can be arbitrarily complex. HTTP has to deliver requests from user agents to servers and return a byte stream, while a server might have to do any amount of processing of the request.

这其实就意味着，HTTP 协议本身非常简单直接，但 HTTP 服务器却可能非常复杂。HTTP 将用户请求发送到服务器，并返回字节流，而服务器针对该请求可能需要做很多很多处理。

HTTP characteristics

HTTP 的特点

HTTP is a stateless, connectionless, reliable protocol. In the simplest form, each request from a user agent is handled reliably and then the connection is broken. Each request involves a separate TCP connection, so if many reources are required (such as images embedded in an HTML page) then many TCP connections have to be set up and torn down in a short space of time.

HTTP 协议是无状态，面向连接和可靠的。最简单的形式是，每个从用户发起的请求被可靠地处理，然后断开连接。每次请求都包括一个独立的 TCP 连接，所以如果要请求很多资源（如在 HTML 页面中嵌入的图像），则必须在很短的时间内建立并断开许多 TCP 连接。

Thera are many optimisations in HTTP which add complexity to the simple structure, in order to create a more efficient and reliable protocol.

为构建更高效更可靠的协议，有许多在这种简单结构基础上添加复杂性的优化技术。

Versions

版本

There are 3 versions of HTTP

HTTP 有三个版本

- Version 0.9 - totally obsolete
- Version 1.0 - almost obsolete
- Version 1.1 - current

- Version 0.9 - 完全废弃
- Version 1.0 - 基本废弃
- Version 1.1 - 当前版本

Each version must understand requests and responses of earlier versions.

每个版本必须兼容早期的版本。

HTTP 0.9

Request format

请求格式

```
Request = Simple-Request

Simple-Request = "GET" SP Request-URI CRLF
```

Response format

响应格式

A response is of the form

响应形式类似：

```
Response = Simple-Response

Simple-Response = [Entity-Body]
```

HTTP 1.0

This version added much more information to the requests and responses. Rather than "grow" the 0.9 format, it was just left alongside the new version.

该版本在请求和响应中增加了很多信息。与其说是 0.9 的升级版，还不如说它是一个全新的版本。

Request format

请求格式

The format of requests from client to server is

从客户端到服务器端的请求格式：

```
Request = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

Full-Request = Request-Line
                    *(General-Header
                    | Request-Header
                    | Entity-Header)
                    CRLF
                    [Entity-Body]
```

A Simple-Request is an HTTP/0.9 request and must be replied to by a Simple-Response.

简单请求(Simple-Request)表明是一个 HTTP/0.9 请求，必须回复简单响应(Simple-Response)。

A Request-Line has format

请求行(Request-Line)的格式如下：

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

where

其中

```
Method = "GET" | "HEAD" | POST |
```

```
        extension-method
```

e.g.

如:

```
GET http://jan.newmarch.name/index.html HTTP/1.0
```

Response format

响应格式

A response is of the form

响应的形式如下:

```
Response = Simple-Response | Full-Response

Simple-Response = [Entity-Body]

Full-Response = Status-Line
                *(General-Header
                | Response-Header
                | Entity-Header)
                CRLF
                [Entity-Body]
```

The Status-Line gives information about the fate of the request:

状态行(Status-Line)会给出请求的最后的状态信息:

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

e.g.

如

```
HTTP/1.0 200 OK
```

The codes are

状态码:

```
Status-Code =       "200" ; OK
                    | "201" ; Created
                    | "202" ; Accepted
                    | "204" ; No Content
                    | "301" ; Moved permanently
                    | "302" ; Moved temporarily
                    | "304" ; Not modified
                    | "400" ; Bad request
                    | "401" ; Unauthorised
                    | "403" ; Forbidden
                    | "404" ; Not found
                    | "500" ; Internal server error
                    | "501" ; Not implemented
                    | "502" ; Bad gateway
                    | "503" | Service unavailable
                    | extension-code
```

The Entity-Header contains useful information about the Entity-Body to follow

实体头(Entity-Header)包含了有关实体(Entity-Body)的有用信息

```
Entity-Header = Allow
                    | Content-Encoding
                    | Content-Length
```

```
                | Content-Type
                | Expires
                | Last-Modified
                | extension-header
```

For example

例如：

```
HTTP/1.1 200 OK
Date: Fri, 29 Aug 2003 00:59:56 GMT
Server: Apache/2.0.40 (Unix)
Accept-Ranges: bytes
Content-Length: 1595
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

HTTP 1.1

HTTP 1.1 fixes many problems with HTTP 1.0, but is more complex because of it. This version is done by extending or refining the options available to HTTP 1.0. e.g.

HTTP 1.1 修复了 HTTP 1.0 中的很多问题，因此更加复杂。例如此版本中扩展和完善了 HTTP 1.0 中的可选项。

- there are more commands such as TRACE and CONNECT
- you should use absolute URLs, particularly for connecting by proxies e.g

```
GET http://www.w3.org/index.html HTTP/1.1
```

- there are more attributes such as If-Modified-Since, also for use by proxies

- 增加了命令，如 TRACE 和 CONNECT
- 注意在通过代理服务器进行连接时，应当使用绝对路径。如：

```
GET http://www.w3.org/index.html HTTP/1.1
```

- 增加了更多属性，例如针对代理服务器的 If-Modified-Since。

The changes include

这些变动包括：

- hostname identification (allows virtual hosts)
- content negotiation (multiple languages)
- persistent connections (reduces TCP overheads - this is very messy)
- chunked transfers
- byte ranges (request parts of documents)
- proxy support

- 主机名识别(支持虚拟主机)
- 内容协商(多语言)
- 持久连接(降低 TCP 开销)
- 分块传送
- 字节范围(请求文件部分内容)
- 代理支持

The 0.9 protocol took one page. The 1.0 protocol was described in about 20 pages. 1.1 takes 120 pages.

0.9 版本的协议只有一页，1.0 版本用了大约 20 页来说明，而 1.1 则用了 120 页。

Simple user-agents

简单用户代理(Simple user-agents)

User agents such as browsers make requests and get responses. The response type is

用户代理(User agent)(例如浏览器)用来发起请求和接收响应。代码中的 response type 如下：

```
type Response struct {
    Status        string // e.g. "200 OK"
    StatusCode int      // e.g. 200
    Proto          string // e.g. "HTTP/1.0"
    ProtoMajor int      // e.g. 1
    ProtoMinor int      // e.g. 0

    RequestMethod string // e.g. "HEAD", "CONNECT", "GET", etc.

    Header map[string]string

    Body io.ReadCloser

    ContentLength int64

    TransferEncoding []string

    Close bool

    Trailer map[string]string
}
```

We shall examine this data structure through examples. The simplest request is from a user agent is "HEAD" which asks for information about a resource and its HTTP server. The function

通过实例可以了解其数据结构。最简单的请求是由用户代理发起"HEAD"命令，其中包括请求的资源和 HTTP 服务器。函数

```
func Head(url string) (r *Response, err os.Error)
```

can be used to make this query.

可用来发起此请求。

The status of the response is in the response field **Status**, while the field **Header** is a map of the header fields in the HTTP response. A program to make this request and display the results is

响应状态对应 response 中的 **Status** 属性，而 **Header** 属性对应 HTTP 响应的 header 域。下面的程序用来发起请求和显示结果：

```
/* Head
 */

package main

import (
        "fmt"
        "net/http"
        "os"
)
```

```
func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "host:port")
		os.Exit(1)
	}
	url := os.Args[1]

	response, err := http.Head(url)
	if err != nil {
		fmt.Println(err.Error())
		os.Exit(2)
	}

	fmt.Println(response.Status)
	for k, v := range response.Header {
		fmt.Println(k+":", v)
	}

	os.Exit(0)
}
```

When run against a resource as in Head http://www.golang.com/ it prints something like

程序运行请求资源，Head http://www.golang.com/，输出结果类似：

```
200 OK
Content-Type: text/html; charset=utf-8
Date: Tue, 14 Sep 2010 05:34:29 GMT
Cache-Control: public, max-age=3600
Expires: Tue, 14 Sep 2010 06:34:29 GMT
```

```
Server: Google Frontend
```

Usually, we are want to retrieve a resource rather than just get information about it. The "GET" request will do this, and this can be done using

通常我们希望接收到一个资源内容而不是其有关信息。"GET"请求就是做来做这个的，使用如下函数即可：

```
func Get(url string) (r *Response, finalURL string, err os.Error)
```

The content of the response is in the response field **Body** which is of type **io.ReadCloser**. We can print the content to the screen with the following program

响应内容为 response 的 **Body** 属性。它是一个 **io.ReadCloser** 类型。我们可以用以下程序在屏幕上打印相应内容

```
/* Get
 */

package main

import (
        "fmt"
        "net/http"
        "net/http/httputil"
        "os"
        "strings"
)
```

```
func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "host:port")
		os.Exit(1)
	}
	url := os.Args[1]

	response, err := http.Get(url)
	if err != nil {
		fmt.Println(err.Error())
		os.Exit(2)
	}

	if response.Status != "200 OK" {
		fmt.Println(response.Status)
		os.Exit(2)
	}

	b, _ := httputil.DumpResponse(response, false)
	fmt.Print(string(b))

	contentTypes := response.Header["Content-Type"]
	if !acceptableCharset(contentTypes) {
		fmt.Println("Cannot handle", contentTypes)
		os.Exit(4)
	}

	var buf [512]byte
```

```
        reader := response.Body
        for {
                n, err := reader.Read(buf[0:])
                if err != nil {
                        os.Exit(0)
                }
                fmt.Print(string(buf[0:n]))
        }
        os.Exit(0)
}

func acceptableCharset(contentTypes []string) bool {
        // each type is like [text/html; charset=UTF-8]
        // we want the UTF-8 only
        for _, cType := range contentTypes {
                if strings.Index(cType, "UTF-8") != -1 {
                        return true
                }
        }
        return false
}
```

Note that there are important character set issues of the type discussed in the previous chapter. The server will deliver the content using some character set encoding, and possibly some transfer encoding. Usually this is a matter of negotiation between user agent and server, but the simple **Get** command that we are using does not include the user agent component of the negotiation. So the server can send whatever character encoding it wishes.

注意这里有一个重要的字符集类型问题，在前面章节也讨论过。服务器提供内容时使用的字符集编码，甚至传输编码，通常是用户代理和服务器之间协商的那结果，但我们使用的 **Get**

的命令很简单，它不包括用户代理的内容协商组件。因此，服务器可以自行决定使用什么字符编码。

At the time of first writing, I was in China. When I tried this program on **www.google.com**, Google's server tried to be helpful by guessing my location and sending me the text in the Chinese character set Big5! How to tell the server what character encoding is okay for me is discussed later.

我第一次写的时候是在中国。当我用这个程序访问 **www.google.com** 时，谷歌的服务器尝试猜测我的地理位置，然后很厉害地使用了 Big5 码给我发送文本！后面会讨论如何告知服务器给我什么字符编码最好。

Configuring HTTP requests

设置 HTTP 请求

Go also supplies a lower-level interface for user agents to communicate with HTTP servers. As you might expect, not only does it give you more control over the client requests, but requires you to spend more effort in building the requests. However, there is only a small increase.

Go 还提供一个较低级别的用户代理接口用来与 HTTP 服务器进行通信。你可能已经想到，这样可以更灵活地控制客户端请求，当然创建请求也会更费力气。不过这只需要多费一点点力气。

The data type used to build requests is the type **Request**. This is a complex type, and is given in the Go documentation as

用来创建请求的数据类型是 **Request**。这是个复杂的类型，Go 语言文档中给出的定义如下：

```
type Request struct {
    Method        string // GET, POST, PUT, etc.
    RawURL        string // The raw URL given in the request.
    URL           *URL   // Parsed URL.
```

```
Proto         string // "HTTP/1.0"
ProtoMajor int       // 1
ProtoMinor int       // 0



// A header maps request lines to their values.
// If the header says
//
//    accept-encoding: gzip, deflate
//    Accept-Language: en-us
//    Connection: keep-alive
//
// then
//
//    Header = map[string]string{
//                "Accept-Encoding": "gzip, deflate",
//                "Accept-Language": "en-us",
//                "Connection": "keep-alive",
//    }
//
// HTTP defines that header names are case-insensitive.
// The request parser implements this by canonicalizing the
// name, making the first character and any characters
// following a hyphen uppercase and the rest lowercase.
Header map[string]string


// The message body.
Body io.ReadCloser
```

```
// ContentLength records the length of the associated content.
// The value -1 indicates that the length is unknown.
// Values >= 0 indicate that the given number of bytes may be read from Body.
ContentLength int64

// TransferEncoding lists the transfer encodings from outermost to innermost.
// An empty list denotes the "identity" encoding.
TransferEncoding []string

// Whether to close the connection after replying to this request.
Close bool

// The host on which the URL is sought.
// Per RFC 2616, this is either the value of the Host: header
// or the host name given in the URL itself.
Host string

// The referring URL, if sent in the request.
//
// Referer is misspelled as in the request itself,
// a mistake from the earliest days of HTTP.
// This value can also be fetched from the Header map
// as Header["Referer"]; the benefit of making it
// available as a structure field is that the compiler
// can diagnose programs that use the alternate
// (correct English) spelling req.Referrer but cannot
// diagnose programs that use Header["Referrer"].
Referer string
```

```
    // The User-Agent: header string, if sent in the request.
    UserAgent string

    // The parsed form. Only available after ParseForm is called.
    Form map[string][]string

    // Trailer maps trailer keys to values.    Like for Header, if the
    // response has multiple trailer lines with the same key, they will be
    // concatenated, delimited by commas.
    Trailer map[string]string
}
```

There is a lot of information that can be stored in a request. You do not need to fill in all fields, only those of interest. The simplest way to create a request with default values is by for example

请求中可以存放大量的信息，但你不需要填写所有的内容，只填必要的即可。最简单的使用默认值创建请求的方法如下：

```
request, err := http.NewRequest("GET", url.String(), nil)
```

Once a request has been created, you can modify fields. For example, to specify that you only wish to receive UTF-8, add an "Accept-Charset" field to a request by

请求创建后，可以修改其内容字段(field)。比如，需指定只接受 UTF-8，可添加一个"Accept-Charset"字段：

```
request.Header.Add("Accept-Charset", "UTF-8;q=1, ISO-8859-1;q=0")
```

(Note that the default set ISO-8859-1 always gets a value of one unless mentioned explicitly in the list.).

(注意，若没有在列表中提及，则默认设置 ISO-8859-1 总是返回值 1).

A client setting a charset request is simple by the above. But there is some confusion about what happens with the server's return value of a charset. The returned resource *should* have a **Content-Type** which will specify the media type of the content such as **text/html**. If appropriate the media type should state the charset, such as **text/html; charset=UTF-8**. If there is no charset specification, then according to the HTTP specification it should be treated as the default ISO8859-1 charset. But the HTML 4 specification states that since many servers don't conform to this, then you can't make any assumptions.

如上所述，客户端设置字符集请求很简单。但对于服务器返回的字符集，发生的事情就比较复杂。返回的资源*理应*包含 **Content-Type**，用来指明内容的媒介类型，如：**text/html**。有些媒介类型应当声明字符集，如 **text/html; charset=UTF-8**。如果没有指明字符集，按照 HTTP 规范就应当作为默认的 ISO8859-1 字符集处理。但是很多服务器并不符合此约定，因此 HTML 4 规定此时不能做任何假设。

If there is a charset specified in the server's **Content-Type**, then assume it is correct. if there is none specified, since 50% of pages are in UTF-8 and 20% are in ASCII then it is safe to assume UTF-8. Only 30% of pages may be wrong :-(.

如果服务器的 **Content-Type** 指定了字符集，那么就认为它是正确的。如果未指定字符集，由于 50%的页面是 UTF-8 的，20%的页面是 ASCII 的，因此假设字符集是 UTF-8 的会比较安全，但仍然有 30%的页面可能会出错:-(。

The Client object

客户端对象

To send a request to a server and get a reply, the convenience object **Client** is the easiest way. This object can manage multiple requests and will look after issues such as whether the server keeps the TCP connection alive, and so on.

向服务器发送一个请求并取得回复，最简单的方法是使用方便对象 **Client**。此对象可以管理多个请求，并处理一些问题，如与服务器间的 TCP 连接是否保持活动状态等。

This is illustrated in the following program

下面的程序给出了示例：

```
/* ClientGet
 */

package main

import (
        "fmt"
        "net/http"
        "net/url"
        "os"
        "strings"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "http://host:port/page")
                os.Exit(1)
        }
        url, err := url.Parse(os.Args[1])
```

```
	checkError(err)


	client := &http.Client{}


	request, err := http.NewRequest("GET", url.String(), nil)
	// only accept UTF-8
request.Header.Add("Accept-Charset", "UTF-8;q=1, ISO-8859-1;q=0")
	checkError(err)


	response, err := client.Do(request)
	if response.Status != "200 OK" {
		fmt.Println(response.Status)
		os.Exit(2)
	}


	chSet := getCharset(response)
	fmt.Printf("got charset %s\n", chSet)
	if chSet != "UTF-8" {
		fmt.Println("Cannot handle", chSet)
		os.Exit(4)
	}


	var buf [512]byte
	reader := response.Body
	fmt.Println("got body")
	for {
		n, err := reader.Read(buf[0:])
		if err != nil {
			os.Exit(0)
```

```
                }
                fmt.Print(string(buf[0:n]))
        }

        os.Exit(0)
}

func getCharset(response *http.Response) string {
        contentType := response.Header.Get("Content-Type")
        if contentType == "" {
                // guess
          return "UTF-8"
        }
        idx := strings.Index(contentType, "charset:")
        if idx == -1 {
                // guess
          return "UTF-8"
        }
        return strings.Trim(contentType[idx:], " ")
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Proxy handling

代理处理

Simple proxy

简单代理

HTTP 1.1 laid out how HTTP should work through a proxy. A "GET" request should be made to a proxy. However, the URL requested should be the full URL of the destination. In addition the HTTP header should contain a "Host" field, set to the proxy. As long as the proxy is configured to pass such requests through, then that is all that needs to be done.

HTTP 1.1 规定了 HTTP 应当如何通过代理工作。向代理服务器发送一个"GET"请求。但是请求 URL 必须是完整的目标地址。此外，设置代理的 HTTP 头应当包括"Host"字段。只要代理服务器设置为允许这样的请求通过，那么做这些就够了。

Go considers this to be part of the HTTP transport layer. To manage this it has a class **Transport**. This contains a field which can be set to a *function* that returns a URL for a proxy. If we have a URL as a string for the proxy, the appropriate transport object is created and then given to a client object by

Go 把这看成 HTTP 传输层的一部分。可使用 **Transport** 类进行管理。可以使用*函数*将代理服务器的 URL 返回到它的一个字段。假设有一个代理服务器地址字符串 URL，相应的创建 Transport 对象并交给 Client 对象的代码就是：

```
proxyURL, err := url.Parse(proxyString)
transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
client := &http.Client{Transport: transport}
```

The client can then continue as before.

客户端可以像之前一样继续使用

The following program illustrates this:

下面是程序范例:

```
/* ProxyGet
 */

package main

import (
        "fmt"
        "io"
        "net/http"
        "net/http/httputil"
        "net/url"
        "os"
)

func main() {
        if len(os.Args) != 3 {
                fmt.Println("Usage: ", os.Args[0], "http://proxy-host:port
http://host:port/page")
                os.Exit(1)
        }
        proxyString := os.Args[1]
        proxyURL, err := url.Parse(proxyString)
        checkError(err)
        rawURL := os.Args[2]
```

```
url, err := url.Parse(rawURL)
checkError(err)

transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
client := &http.Client{Transport: transport}

request, err := http.NewRequest("GET", url.String(), nil)

dump, _ := httputil.DumpRequest(request, false)
fmt.Println(string(dump))

response, err := client.Do(request)

checkError(err)
fmt.Println("Read ok")

if response.Status != "200 OK" {
        fmt.Println(response.Status)
        os.Exit(2)
}
fmt.Println("Reponse ok")

var buf [512]byte
reader := response.Body
for {
        n, err := reader.Read(buf[0:])
        if err != nil {
                os.Exit(0)
        }
```

```
                fmt.Print(string(buf[0:n]))
        }

        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                if err == io.EOF {
                        return
                }
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

If you have a proxy at, say, XYZ.com on port 8080, test this by

假设有一个代理服务器 XYZ.com，端口 8080，测试命令就是

```
go run ProxyGet.go http://XYZ.com:8080/ http://www.google.com
```

If you don't have a suitable proxy to test this, then download and install the Squid proxy to your own computer.

如果没有合适的代理服务器可供测试，也可以在自己的计算机上下载安装 Squid proxy。

The above program used a known proxy passed as an argument to the program. There are many ways in which proxies can be made known to applications. Most browsers have a configuration menu in

which you can enter proxy information: such information is not available to a Go application. Some applications may get proxy information from an **autoproxy.pac** file somewhere in your network: Go does not (yet) know how to parse these JavaScript files and so cannot use them. Linux systems using Gnome have a configuration system called **gconf** in which proxy information can be stored: Go cannot access this. *But* it can find proxy information if it is set in operating system environment variables such as HTTP_PROXY or http_proxy using the function

上面的程序是将已知的代理服务器地址作为参数传入的。有很多办法可以将代理服务器的地址通知到应用程序。大多数浏览器可以通过配置菜单输入代理服务器的信息：但这些信息对 Go 应用没有用。有些应用程序可以从网络中某处找到 **autoproxy.pac** 文件取得其中的代理服务器信息，但 Go(目前还)不能解析 JavaScript 文件，因此也不能使用。Gnome Linux 系统使用的配置系统 **gconf** 里可以存储代理服务器信息，但 Go 也访问不了。*但是*，如果在操作系统环境变量中设置代理服务器信息（如 HTTP_PROXY 或 http_proxy），Go 可以通过以下函数访问到：

```
func ProxyFromEnvironment(req *Request) (*url.URL, error)
```

If your programs are running in such an environment you can use this function instead of having to explicitly know the proxy parameters.

假如你的程序运行在这样的环境中，就可以使用此功能，而不用明确指定代理服务器参数。

Authenticating proxy

身份验证代理

Some proxies will require authentication, by a user name and password in order to pass requests. A common scheme is "basic authentication" in which the user name and password are concatenated

into a string "user:password" and then BASE64 encoded. This is then given to the proxy by the HTTP request header "Proxy-Authorisation" with the flag that it is the basic authentication

有些代理服务器要求通过用户名和密码进行身份验证才能传递请求。一般的方法是“基本身份验证”：将用户名和密码串联成一个字符串“user:password”，然后进行 Base64 编码，然后添加到 HTTP 请求头的“Proxy-Authorization”中，再发送到代理服务器

The following program illlustrates this, adding the Proxy-Authentication header to the previous proxy program:

在前一个程序的基础上增加 Proxy-Authorization 头，示例如下：

```
/* ProxyAuthGet
 */

package main

import (
        "encoding/base64"
        "fmt"
        "io"
        "net/http"
        "net/http/httputil"
        "net/url"
        "os"
)

const auth = "jannewmarch:mypassword"

func main() {
```

```
        if len(os.Args) != 3 {
                fmt.Println("Usage: ", os.Args[0], "http://proxy-host:port
http://host:port/page")
                os.Exit(1)
        }
        proxy := os.Args[1]
        proxyURL, err := url.Parse(proxy)
        checkError(err)
        rawURL := os.Args[2]
        url, err := url.Parse(rawURL)
        checkError(err)

        // encode the auth
 basic := "Basic " + base64.StdEncoding.EncodeToString([]byte(auth))

        transport := &http.Transport{Proxy: http.ProxyURL(proxyURL)}
        client := &http.Client{Transport: transport}

        request, err := http.NewRequest("GET", url.String(), nil)

        request.Header.Add("Proxy-Authorization", basic)
        dump, _ := httputil.DumpRequest(request, false)
        fmt.Println(string(dump))

        // send the request
 response, err := client.Do(request)

        checkError(err)
        fmt.Println("Read ok")
```

```
	if response.Status != "200 OK" {
		fmt.Println(response.Status)
		os.Exit(2)
	}
	fmt.Println("Reponse ok")

	var buf [512]byte
	reader := response.Body
	for {
		n, err := reader.Read(buf[0:])
		if err != nil {
			os.Exit(0)
		}
		fmt.Print(string(buf[0:n]))
	}

	os.Exit(0)
}

func checkError(err error) {
	if err != nil {
		if err == io.EOF {
			return
		}
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

HTTPS connections by clients

客户端发起 HTTPS 连接

For secure, encrypted connections, HTTP uses TLS which is described in the chapter on security. The protocol of HTTP+TLS is called HTTPS and uses https:// urls instead of http:// urls.

为保证连接的安全和加密，HTTP 使用其在安全性章节中说明的 TLS 技术。HTTP+TLS 的协议被称为 HTTPS，它使用 https://地址，而不是 http://地址。

Servers are required to return valid X.509 certificates before a client will accept data from them. If the certificate is valid, then Go handles everything under the hood and the clients given previously run okay with https URLs.

服务器必须在客户端接受从其数据前返回有效的 X.509 证书。如果证书有效，Go 会在内部处理好所有的事情，而客户端会在使用 HTTPS 地址是和以前工作得一样出色。

Many sites have invalid certificates. They may have expired, they may be self-signed instead of by a recognised Certificate Authority or they may just have errors (such as having an incorrect server name). Browsers such as Firefox put a big warning notice with a "Get me out of here!" button, but you can carry on at your risk - which many people do.

许多网站都使用无效的证书。这些证书可能已经过期，或者是自行签名的，而没有让认可的证书颁发机构签名;又或者他们可能只是用错了(比如服务器名称不对)。浏览器(如 Firefox)，会显示一个很大的警告通知，通知上放着“立即离开！”按钮，但你也可以仍然继续此风险 - 很多人会这么做。

Go presently bails out when it encounters certificate errors. There is cautious support for carrying on but I haven't got it working yet. So there is no current example for "carrying on in the face of adversity :-)". Maybe later.

Go 目前在遇到证书错误时，会 bails out。对继续工作的支持非常谨慎，我还没有找到正确的方法。因此，目前也没有“继续此风险”任何示例 :-)。以后再说吧。

Servers

服务器

The other side to building a client is a Web server handling HTTP requests. The simplest - and earliest - servers just returned copies of files. However, any URL can now trigger an arbitrary computation in current servers.

这边创建客户端，另一边 Web 服务器则需要处理 HTTP 请求。最早最简单的服务器只是返回文件的副本。然而，目前的服务器上，随便一个 URL 都可能触发任何计算。

File server

文件服务器

We start with a basic file server. Go supplies a **multi-plexer**, that is, an object that will read and interpret requests. It hands out requests to **handlers** which run in their own thread. Thus much of the work of reading HTTP requests, decoding them and branching to suitable functions in their own thread is done for us.

我们从一个基本的文件服务器开始。Go 提供了一个 **multi-plexer**，即一个读取和解释请求的对象。它把请求交给运行在自己线程中的 **handlers**。这样，许多读取 HTTP 请求，解码并转移到合适功能上的工作都可以在各自的线程中进行。

For a file server, Go also gives a **FileServer** object which knows how to deliver files from the local file system. It takes a "root" directory which is the top of a file tree in the local system, and a pattern to match URLs against. The simplest pattern is "/" which is the top of any URL. This will match all URLs.

对于文件服务器，Go 也提供了一个 **FileServer** 对象，它知道如何发布本地文件系统中的文件。它需要一个“root”目录，该目录是在本地系统中文件树的顶端；还有一个针对 URL 的匹配模式。最简单的模式是“/”，这是所有 URL 的顶部，可以匹配所有的 URL。

An HTTP server delivering files from the local file system is almost embarrassingly trivial given these objects. It is

HTTP 服务器从本地文件系统中发布文件太简单了，让人都有点不好意思举例。如下：

```
/* File Server
 */

package main

import (
        "fmt"
        "net/http"
        "os"
)

func main() {
        // deliver files from the directory /var/www
        // fileServer := http.FileServer(http.Dir("/var/www"))
        fileServer := http.FileServer(http.Dir("/home/httpd/html/"))

        // register the handler and deliver requests to it
        err := http.ListenAndServe(":8000", fileServer)
        checkError(err)
        // That's it!
}

func checkError(err error) {
        if err != nil {
```

```
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

This server even delivers "404 not found" messages for requests for file resources that don't exist!

甚至当请求到一个不存在的文件资源时，这个服务器还提供了“404 未找到”的信息!

Handler functions

处理函数(Handler function)

In this last program, the handler was given in the second argument to **ListenAndServe**. Any number of handlers can be registered first by calls to **Handle** or **handleFunc**, with signatures

上一个程序中，handler 被作为第二个参数传给 **ListenAndServe**。可以先注册任意多个 handler 供 **Handle** 或 **handleFunc** 使用。调用方式：

```
func Handle(pattern string, handler Handler)
func HandleFunc(pattern string, handler func(*Conn, *Request))
```

The second argument to **HandleAndServe** could be **nil**, and then calls are dispatched to all registered handlers. Each handler should have a different URL pattern. For example, the file handler might have URL pattern "/" while a function handler might have URL pattern "/cgi-bin". A more specific pattern takes precedence over a more general pattern.

HandleAndServe 的第二个参数是可以是 **nil**，调用会被分派到所有已注册的 handler。每个对立对象都有不同的 URL 匹配模式。例如，可能文件 handler 的 URL 匹配模式是"/"，而一个函数 handler 的 URL 匹配模式是"/cgi-bin"。这里具体的模式优先级高于一般的模式。

Common CGI programs are **test-cgi** (written in the shell) or **printenv** (written in Perl) which print the values of the environment variables. A handler can be written to work in a similar manner.

常见的 CGI 程序有 **test-cgi**(shell 程序)或 **printenv**(Perl 程序)用来打印环境变量的值。可以让 handler 用类似的方式工作。

```
/* Print Env
 */

package main

import (
        "fmt"
        "net/http"
        "os"
)

func main() {
        // file handler for most files
        fileServer := http.FileServer(http.Dir("/var/www"))
        http.Handle("/", fileServer)

        // function handler for /cgi-bin/printenv
        http.HandleFunc("/cgi-bin/printenv", printEnv)

        // deliver requests to the handlers
        err := http.ListenAndServe(":8000", nil)
        checkError(err)
        // That's it!
```

```
}

func printEnv(writer http.ResponseWriter, req *http.Request) {
	env := os.Environ()
	writer.Write([]byte("<h1>Environment</h1>\n<pre>"))
	for _, v := range env {
		writer.Write([]byte(v + "\n"))
	}
	writer.Write([]byte("</pre>"))
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

Note: for simplicity this program does not deliver well-formed HTML. It is missing html, head and body tags.

注：为简单起见，本程序不提供完整的的 HTML。这里缺少 html、head 和 body 标签。

Using the **cgi-bin** directory in this program is a bit cheeky: it doesn't call an external program like CGI scripts do. It just calls a Go function. Go does have the ability to call external programs using **os.ForkExec**, but does not yet have support for dynamically linkable modules like Apache's **mod_perl**

这个程序在使用 **cgi-bin** 目录时有点耍赖。其实它并没有调用外部的 CGI 脚本程序，而只是使用了一个 Go 的内部函数。Go 确实可以通过 **os.ForkExec** 调用外部的程序，但还不能支持像 Apache 的 **mod_perl** 这样的动态连接库

Bypassing the default multiplexer

绕过默认的 multiplexer

HTTP requests received by a Go server are usually handled by a multiplexer the examines the path in the HTTP request and calls the appropriate file handler, etc. You can define your own handlers. These can either be registered with the default multiplexer by calling **http.HandleFunc** which takes a pattern and a function. The functions such as **ListenAndServe** then take a **nil** handler function. This was done in the last example.

Go 服务器接收到的 HTTP 请求通常是由一个 multiplexer 进行处理，检查 HTTP 请求的路径，然后调用合适的文件 handler 等等。你也可以定义自己的 handler。将一个匹配模式参数和一个函数作为参数，调用 **http.HandleFunc**，可以将其注册为默认的 multiplexer。像 **ListenAndServe** 这样的函数就可以使用 **nil** 作为 handler function。上一个例子就是这样做的。

If you want to take over the multiplexer role then you can give a non-zero function as the handler function. This function will then be totally responsible for managing the requests and responses.

如果你想扮演 multiplexer 的角色，那么你就可以给一个非零函数作为 handler function。这个函数将会全权负责管理请求和响应。

The following example is trivial, but illustrates the use of this: the multiplexer function simply returns a "204 No content" for *all* requests:

下面的例子非常简单，但它说明了如何使 multiplexer 对*所有*请求都只返回一个“204 No content”：

```
/* ServerHandler
 */

package main

import (
```

```
        "net/http"
)

func main() {

        myHandler := http.HandlerFunc(func(rw http.ResponseWriter, request *http.Request) {
                // Just return no content - arbitrary headers can be set, arbitrary body
                rw.WriteHeader(http.StatusNoContent)
        })

        http.ListenAndServe(":8080", myHandler)
}
```

Arbitrarily complex behaviour can be built, of course.

当然，也可以把它做成无所不能的。

Templates

模板

Many languages have mechanisms to convert strings from one form to another. Go has a template mechanism to convert strings based on the content of an object supplied as an argument. While this is often used in rewriting HTML to insert object values, it can be used in other situations. Note that this material doesn't have anything explicitly to do with networking, but may be useful to network programs.

很多编程语言都有字符串之间转换的机制，而 GO 语言则是通过模板来将一个对象的内容来作为参数传递从而字符串的转换。此方式不仅可以在重写 HTML 时插入对象值，也适用于其他方面。注意，本章内容并没有明确给出网络的工作方式，但对于网络编程方式很有用处。

Introduction

介绍

Most server-side languages have a mechanism for taking predominantly static pages and inserting a dynamically generated component, such as a list of items. Typical examples are scripts in Java Server Pages, PHP scripting and many others. Go has adopted a relatively simple scripting language in the **template** package.

大多数服务器端语言的机制主要是在静态页面插入一个动态生成的组件，如清单列表项目。典型的例子是在 JSP、PHP 和许多其他语言的脚本中。GO 的 **template** 包中采取了相对简单的脚本化语言。

At the time of writing a new template package has been adopted. There is very little documentation on the template packages. There is a small amount on the old package, which is currently still available in the **old/template**. There is no documentation on the new package as yet apart from the reference page. The template package changed with r60 (released 2011/09/07).

因为新的 template 包是刚刚被采用的，所有现在的 template 包中的文档少的可怜，旧的 **old/template** 包中也还存有少量的旧模板。新发布的帮助页面还没有关于新包的文档。关于 template 包的更改请参阅 r60 (released 2011/09/07).

We describe the new package here. The package is designed to take text as input and output different text, based on transforming the original text using the values of an object. Unlike JSP or similar, it is not restricted to HTML files but it is likely to find greatest use there.

在这里，我们描述了这个新包。该包是描述了通过使用对象值改变了原来文本的方式从而在输入和输出时获取不同的文本。与 JSP 或类似的不同，它的作用不仅限于 HTML 文件，但在那可能会有更大的作用。

The original source is called a *template* and will consist of text that is transmitted unchanged, and embedded commands which can act on and change text. The commands are delimited by **{{ ... }}** , similar to the JSP commands **<%= ... =%>** and PHPs **<?php ... ?>**.

源文件被称作 *template* ，包括文本传输方式不变，以嵌入命令可以作用于和更改文本。命令规定如 **{{ ... }}** ，类似于 JSP 命令 **<%= ... =%>** 和 PHP 命令 **<?php ... ?>**。

Inserting object values

插入对象值

A template is applied to a Go object. Fields from that Go object can be inserted into the template, and you can 'dig" into the object to find subfields, etc. The current object is represented as '.', so that to insert the value of the current object as a string, you use **{{.}}**. The package uses the **fmt** package by default to work out the string used as inserted values.

模板应用于 GO 对象中.GO 对象的字段被插入到模板后，你就能从域中“挖”到他的子域，等等。当前对象以'.'代替，所以把当前对象当做字符串插入时，你可以采用**{{.}}**的方式。这个包默认采用 **fmt** 包来作为插入值的字符串输出。

To insert the value of a field of the current object, you use the field name prefixed by '.'. For example, if the object is of type

要插入当前对象的一个字段的值，你使用的字段名前加前缀 '.'。 例如，如果要插入的对象的类型为

```
type Person struct {
        Name        string
        Age         int
        Emails      []string
        Jobs        []*Jobs
}
```

then you insert the values of Name and Age by

那么你要插入的字段 Name 和 Age 如下

```
The name is {{.Name}}.
The age is {{.Age}}.
```

We can loop over the elements of an array or other list using the **range** command. So to access the contents of the Emails array we do

我们可以使用 range 命令来循环一个数组或者链表中的元素。所以要获取 Emails 数组的信息，我们可以这么干

```
{{range .Emails}}
        ...
```

```
{{end}}
```

if **Job** is defined by

如果 Job 定义为

```
type Job struct {
    Employer string
    Role     string
}
```

and we want to access the fields of a **Person**'s **Jobs**, we can do it as above with a **{{range .Jobs}}**. An alternative is to switch the current object to the **Jobs** field. This is done using the **{{with ...}} ... {{end}}** construction, where now **{{.}}** is the **Jobs** field, which is an array:

如果我们想访问 Person 字段中的 Jobs，我们可以这么干 {{range .Jobs}}。这是一种可以将当前对象转化为 Jobs 字段的方式. 通过 {{with ...}} ... {{end}} 这种方式，那么{{.}} 就可以拿到 Jobs 字段了,如下:

```
{{with .Jobs}}
    {{range .}}
        An employer is {{.Employer}}
        and the role is {{.Role}}
    {{end}}
{{end}}
```

You can use this with any field, not just an array. Using templates

你可以用这种方法操作任何类型的字段，而不仅限于数组。亲，用模板吧！

Once we have a template, we can apply it to an object to generate a new string, using the object to fill in the template values. This is a two-step process which involves parsing the template and then applying it to an object. The result is output to a **Writer**, as in

当我们拥有了模板,我们将它应用在对象中生成一个字符串，用这个对象来填充这个模板的值。分两步来实现模块的转化和应用，并且输出一个 **Writer**，如下

```
t := template.New("Person template")
t, err := t.Parse(templ)
if err == nil {
        buff := bytes.NewBufferString("")
        t.Execute(buff, person)
}
```

An example program to apply a template to an object and print to standard output is

下面是一个例子来演示模块应用在对象上并且标准输入：

```
/**
 * PrintPerson
 */

package main

import (
        "fmt"
        "html/template"
```

```
        "os"
)

type Person struct {
        Name     string
        Age      int
        Emails []string
        Jobs     []*Job
}

type Job struct {
        Employer string
        Role         string
}

const templ = `The name is {{.Name}}.
The age is {{.Age}}.
{{range .Emails}}
        An email is {{.}}
{{end}}

{{with .Jobs}}
    {{range .}}
        An employer is {{.Employer}}
        and the role is {{.Role}}
    {{end}}
{{end}}
`
```

```
func main() {
	job1 := Job{Employer: "Monash", Role: "Honorary"}
	job2 := Job{Employer: "Box Hill", Role: "Head of HE"}

	person := Person{
		Name:     "jan",
		Age:      50,
		Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
		Jobs:     []*Job{&job1, &job2},
	}

	t := template.New("Person template")
	t, err := t.Parse(templ)
	checkError(err)

	err = t.Execute(os.Stdout, person)
	checkError(err)
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

The output from this is

输出如下：

```
```

```
The name is jan.
The age is 50.

        An email is jan@newmarch.name

        An email is jan.newmarch@gmail.com




        An employer is Monash
        and the role is Honorary

        An employer is Box Hill
        and the role is Head of HE
```

Note that there is plenty of whitespace as newlines in this printout. This is due to the whitespace we have in our template. If we wish to reduce this, eliminate newlines in the template as in

注意，上面有很多空白的输出，这是因为我们的模板中有很多空白。如果想消除它，模板设置如下：

```
{{range .Emails}} An email is {{.}} {{end}}
```

In the example, we used a string in the program as the template. You can also load templates from a file using the function template.ParseFiles(). For some reason that I don't understand (and which

wasn't required in earlier versions), the name assigned to the template must be the same as the basename of the first file in the list of files. Is this a bug?

在这个示例例中，我们用字符串应用于模板。你同样也可以用方法 template.ParseFiles()来从文件中下载模板。因为某些原因，我还不没搞清楚(在早期版本没有强制要求),关联模板的名字必须要与文件列表的第一个文件的基名相同。话说，这个是 BUG 吗?

Pipelines

管道

The above transformations insert pieces of text into a template. Those pieces of text are essentially arbitrary, whatever the string values of the fields are. If we want them to appear as part of an HTML document (or other specialised form) then we will have to escape particular sequences of characters. For example, to display arbitrary text in an HTML document we have to change "<" to "<". The Go templates have a number of builtin functions, and one of these is the function **html**. These functions act in a similar manner to Unix pipelines, reading from standard input and writing to standard output.

上述转换到模板中插入的文本块。这些字符基本上是任意的，是任何字符串的字段值。如果我们希望它们出现的是 HTML 文档（或其他的特殊形式）的一部分，那么我们将不得不脱离特定的字符序列。例如，要显示任意文本在 HTML 文档中，我们要将“<”改成“&lt”。GO 模板有一些内建函数，其中之一是 **html**。这些函数的作用与 Unix 的管道类似，从标准输入读取和写入到标准输出。

To take the value of the current object '.' and apply HTML escapes to it, you write a "pipeline" in the template

如果想用“.”来获取当前对象值并且应用于 HTML 转义，你可以在模板里写个“管道”：

```
{{. | html}}
```

and similarly for other functions.

其他方法类似。

Mike Samuel has pointed out a convenience function currently in the **exp/template/html** package. If all of the entries in a template need to be passed through the **html** template function, then the Go function **Escape(t *template.Template)** can take a template and add the **html** function to each node in the template that doesn't already have one. This will be useful for templates used for HTML documents and can form a pattern for similar function uses elsewhere.

Mike Samuel 指出，目前在 **exp/template/html** 包里有一个方便的方法。如果所有的模板中的条目需要通过 **html** 模板函数，那么 Go 语言方法 **Escape(t *template.Template)**就能获取模板而后将 **html** 函数添加到模板中不存在该函数的每个节点中。用于 HTML 文档的模板是非常有用的，并能在其他使用场合生成相似的方法模式。

Defining functions

定义方法

The templates use the string representation of an object to insert values, using the **fmt** package to convert the object to a string. Sometimes this isn't what is needed. For example, to avoid spammers getting hold of email addresses it is quite common to see the symbol '@' replaced by the word " at ", as in "jan at newmarch.name". If we want to use a template to display email addresses in that form, then we have to build a custom function to do this transformation.

模板使用对象化的字符串表示形式插入值，使用 **fmt** 包将对象转换为字符串。有时候，这并不是必需。例如，为了避免被垃圾邮件发送者掌握电子邮件地址，常见的方式是把字符号“@”替换为“at”，如“jan at newmarch.name”。如果我们要使用一个模板，显示在该表单中的电子邮件地址，那么我们就必须建立一个自定义的功能做这种转变。

Each template function has a name that is used in the templates themselves, and an associated Go function. These are linked by the type

每个模板函数中使用的模板本身有的一个名称，以及相关联的函数。他们用下面方式进行关联如下

```
type FuncMap map[string]interface{}
```

For example, if we want our template function to be "emailExpand" which is linked to the Go function EmailExpander then we add this to the functions in a template by

例如，如果我们希望我们的模板函数是“emailExpand”，用来关联到 Go 函数 EmailExpander，然后，我们像这样添加函数到到模板中

```
t = t.Funcs(template.FuncMap{"emailExpand": EmailExpander})
```

The signature for EmailExpander is typically

EmailExpander 通常像这样标记：

```
func EmailExpander(args ...interface{}) string
```

In the use we are interested in, there should only be one argument to the function which will be a string. Existing functions in the Go template library have some initial code to handle non-conforming cases, so we just copy that. Then it is just simple string manipulation to change the format of the email address. A program is

我们感兴趣的是在使用过程中，那是一个只有一个参数的函数，并且是个字符串。在 Go 模板库的现有功能有初步的代码来处理不符合要求的情况，所以我们只需要复制。然后，它就能通过简单的字符串操作来改变格式的电子邮件地址。程序如

```
/**
 * PrintEmails
 */

package main

import (
	"fmt"
	"os"
	"strings"
	"text/template"
)

type Person struct {
	Name	string
	Emails []string
}

const templ = `The name is {{.Name}}.
{{range .Emails}}
	An email is "{{. | emailExpand}}"
{{end}}
`

func EmailExpander(args ...interface{}) string {
	ok := false
	var s string
	if len(args) == 1 {
```

```
                s, ok = args[0].(string)
        }
        if !ok {
                s = fmt.Sprint(args...)
        }

        // find the @ symbol
 substrs := strings.Split(s, "@")
        if len(substrs) != 2 {
                return s
        }
        // replace the @ by " at "
 return (substrs[0] + " at " + substrs[1])
}

func main() {
        person := Person{
                Name:     "jan",
                Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
        }

        t := template.New("Person template")

        // add our function
 t = t.Funcs(template.FuncMap{"emailExpand": EmailExpander})

        t, err := t.Parse(templ)

        checkError(err)
```

```
        err = t.Execute(os.Stdout, person)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The output is

输出为：

```
The name is jan.

        An email is "jan at newmarch.name"

        An email is "jan.newmarch at gmail.com"
```

Variables

变量

The template package allows you to define and use variables. As motivation for this, consider how we might print each person's email address *prefixed* by their name. The type we use is again

template 包，允许您定义和使用变量。这样做的动机，可能我们会考虑通过把他们的名字当做电子邮件地址*前缀*打印出来。我们又使用这个类型

```
type Person struct {
        Name        string
        Emails      []string
}
```

To access the email strings, we use a **range** statement such as

为了访问 email 的所有字符串，可以用 **range**，如下

```
{{range .Emails}}
    {{.}}
{{end}}
```

But at that point we cannot access the **Name** field as '.' is now traversing the array elements and the **Name** is outside of this scope. The solution is to save the value of the **Name** field in a variable that can be accessed anywhere in its scope. Variables in templates are prefixed by '$'. So we write

但是需要指出的是，我们无法用'.' 的形式来访问字段 **Name**，因为当他被转化成数组元素时，字段 **Name** 并不包括其中。解决方法是，将字段 **Name** 存储为一个变量，那么它就能在任意范围内被访问。变量在模板中用法是加前缀'$'。所以可以这样

```
{{$name := .Name}}
{{range .Emails}}
    Name is {{$name}}, email is {{.}}
```

```
{{end}}
```

The program is

程序如下：

```
/**
 * PrintNameEmails
 */

package main

import (
	"html/template"
	"os"
	"fmt"
)

type Person struct {
	Name	string
	Emails []string
}

const templ = `{{$name := .Name}}
{{range .Emails}}
    Name is {{$name}}, email is {{.}}
{{end}}
`
```

```
func main() {
	person := Person{
		Name:	"jan",
		Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
	}

	t := template.New("Person template")
	t, err := t.Parse(templ)
	checkError(err)

	err = t.Execute(os.Stdout, person)
	checkError(err)
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

with output

输出为

```
Name is jan, email is jan@newmarch.name

Name is jan, email is jan.newmarch@gmail.com
```

Conditional statements

条件语句

Continuing with our **Person** example, supposing we just want to print out the list of emails, without digging into it. We can do that with a template

继续我们那个 **Person** 的例子，假设我们只是想打印出来的邮件列表，而不关心其中的字段。我们可以用模板这么干

```
Name is {{.Name}}
Emails are {{.Emails}}
```

This will print

```
Name is jan
Emails are [jan@newmarch.name jan.newmarch@gmail.com]
```

because that is how the **fmt** package will display a list.

因为这个 **fmt** 包会显示一个列表。

In many circumstances that may be fine, if that is what you want. Let's consider a case where it is *almost* right but not quite. There is a JSON package to serialise objects, which we looked at in Chapter 4. This would produce

在许多情况下，这样做也没有问题，如果那是你想要的。让我们考虑下一种情况，它 *几乎* 是对的但不是必须的。有一个 JSON 序列化对象的包，让我们看看第 4 章。它是这样的

```
{"Name": "jan",
```

```
 "Emails": ["jan@newmarch.name", "jan.newmarch@gmail.com"]
}
```

The JSON package is the one you would use in practice, but let's see if we can produce JSON output using templates. We can do something similar just by the templates we have. This is *almost* right as a JSON serialiser:

JSON 包是一个你会在实践中使用，但是让我们看看我们是否能够使用 JSON 输出模板。我们可以做一些我们有的类似的模板。这*几乎*就是一个 JSON 串行器：

```
{"Name": "{{.Name}}",
 "Emails": {{.Emails}}
}
```

It will produce

像这样组装

```
{"Name": "jan",
 "Emails": [jan@newmarch.name jan.newmarch@gmail.com]
}
```

which has two problems: the addresses aren't in quotes, and the list elements should be ',' separated.

其中有两个问题：地址没有在引号中，列表中的元素应该是'，'分隔。

How about this: looking at the array elements, putting them in quotes and adding commas?

这样如何：在数组中的元素，把它们放在引号中并用逗号分隔?

```
{"Name": {{.Name}},
  "Emails": [
    {{range .Emails}}
        "{{.}}",
    {{end}}
  ]
}
```

It will produce

像这样组装

```
{"Name": "jan",
 "Emails": ["jan@newmarch.name", "jan.newmarch@gmail.com",]
}
```

(plus some white space.).

(再加上一些空白)。

Again, almost correct, but if you look carefully, you will see a trailing ',' after the last list element. According to the JSON syntax (see http://www.json.org/, this trailing ',' is not allowed. Implementations may vary in how they deal with this.

同样，这样貌似几乎是正确的，但如果你仔细看，你会看到尾有“，”在最后的列表元素。根据 JSON 的语法（请参阅 http://www.json.org/，这个结尾的','是不允许的。这样实现结果可能会有所不同。

What we want is "print every element followed by a ',' except for the last one." This is actually a bit hard to do, so a better way is "print every element *preceded* by a ',' except for the *first* one." (I got this tip from "brianb" at Stack Overflow.). This is easier, because the first element has index zero and many programming languages, including the Go template language, treat zero as Boolean **false**.

我们想要打印所有在后面带','的元素除了最后一个。"这个确实有点难搞，一个好方法"在','*之前*打印所有元素除了*第一个*。" (我在 "brianb"的 Stack Overflow 上提了建议)。这样更易于实现，因为第一个元素索引为 0，很多编程语言包括 GO 模板都将 0 当做布尔型的 **false**。

One form of the conditional statement is **{{if pipeline}} T1 {{else}} T0 {{end}}**. We need the **pipeline** to be the index into the array of emails. Fortunately, a variation on the **range** statement gives us this. There are two forms which introduce variables

条件语句的一种形式是**{{if pipeline}} T1 {{else}} T0 {{end}}**。我们需要通过 **pipeline** 来获取电子邮件到数组的索引。幸运的是， **range** 的变化语句为我们提供了这一点。有两种形式，引进变量

```
{{range $elmt := array}}
{{range $index, $elmt := array}}
```

So we set up a loop through the array, and if the index is false (0) we just print the element, otherwise print it preceded by a ','. The template is

所以我们遍历数组，如果该索引是 false（0），我们只是打印的这个索引的元素，否则打印它前面是','的元素。模板是这样的

```
{"Name": "{{.Name}}",
 "Emails": [
 {{range $index, $elmt := .Emails}}
```

```
    {{if $index}}
        ,"{{$elmt}}"
    {{else}}
        "{{$elmt}}"
    {{end}}
 {{end}}
]
}
```

and the full program is

完整的程序如下

```
/**
 * PrintJSONEmails
 */

package main

import (
        "html/template"
        "os"
        "fmt"
)

type Person struct {
        Name    string
        Emails []string
}
```

```
const templ = `{"Name": "{{.Name}}",
 "Emails": [
{{range $index, $elmt := .Emails}}
    {{if $index}}
        , "{{$elmt}}"
    {{else}}
         "{{$elmt}}"
    {{end}}
{{end}}
 ]
}
`

func main() {
        person := Person{
                Name:     "jan",
                Emails: []string{"jan@newmarch.name", "jan.newmarch@gmail.com"},
        }

        t := template.New("Person template")
        t, err := t.Parse(templ)
        checkError(err)

        err = t.Execute(os.Stdout, person)
        checkError(err)
}

func checkError(err error) {
```

```
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

This gives the correct JSON output.

上面给出的是正确的 JSON 输出

Before leaving this section, we note that the problem of formatting a list with comma separators can be approached by defining suitable functions in Go that are made available as template functions. To re-use a well known saying, "There's more than one way to do it!". The following program was sent to me by Roger Peppe:

在结束本节之前，我们强调了用逗号分隔的列表格式的问题，解决方式是可以在模板函数中定义适当的函数。正如俗话说的，“道路不止一条！”下面的程序是 Roger Peppe 给我的：

```
/**
 * Sequence.go
 * Copyright Roger Peppe
 */

package main

import (
        "errors"
        "fmt"
        "os"
        "text/template"
)
```

```
var tmpl = `{{$comma := sequence "" ", "}}
{{range $}}{{$comma.Next}}{{.}}{{end}}
{{$comma := sequence "" ", "}}
{{$colour := cycle "black" "white" "red"}}
{{range $}}{{$comma.Next}}{{.}} in {{$colour.Next}}{{end}}
`

var fmap = template.FuncMap{
	"sequence": sequenceFunc,
	"cycle":    cycleFunc,
}

func main() {
	t, err := template.New("").Funcs(fmap).Parse(tmpl)
	if err != nil {
		fmt.Printf("parse error: %v\n", err)
		return
	}
	err = t.Execute(os.Stdout, []string{"a", "b", "c", "d", "e", "f"})
	if err != nil {
		fmt.Printf("exec error: %v\n", err)
	}
}

type generator struct {
	ss []string
	i  int
	f  func(s []string, i int) string
```

```
}

func (seq *generator) Next() string {
	s := seq.f(seq.ss, seq.i)
	seq.i++
	return s
}

func sequenceGen(ss []string, i int) string {
	if i >= len(ss) {
		return ss[len(ss)-1]
	}
	return ss[i]
}

func cycleGen(ss []string, i int) string {
	return ss[i%len(ss)]
}

func sequenceFunc(ss ...string) (*generator, error) {
	if len(ss) == 0 {
		return nil, errors.New("sequence must have at least one element")
	}
	return &generator{ss, 0, sequenceGen}, nil
}

func cycleFunc(ss ...string) (*generator, error) {
	if len(ss) == 0 {
		return nil, errors.New("cycle must have at least one element")
```

```
        }
        return &generator{ss, 0, cycleGen}, nil
}
```

Conclusion

结论

The Go template package is useful for certain kinds of text transformations involving inserting values of objects. It does not have the power of, say, regular expressions, but is faster and in many cases will be easier to use than regular expressions

template 包在对于某些类型的文本转换涉及插入对象值的情况是非常有用的。虽然它没有正则表达式功能强大，但它执行比正则表达式速度更快，在许多情况下比正则表达式更容易使用。

A Complete Web Server

一个完整的 Web 服务器

This chapter is principally a lengthy illustration of the HTTP chapter, building a complete Web server in Go. It also shows how to use templates in order to use expressions in text files to insert variable values and to generate repeated sections.

这章主要是针对 Http 的一个例子，用 Go 建立一个完整的 Web 服务器。它也演示了如何在模板中使用表达式在文本文件中插入变量和生成重复的部分。

Introduction

说明

I am learning Chinese. Rather, after many years of trying I am still *attempting* to learn Chinese. Of course, rather than buckling down and getting on with it, I have tried all sorts of technical aids. I tried

DVDs, videos, flashcards and so on. Eventually I realised that there *wasn't a good computer program for Chinese flashcards*, and so in the interests of learning, I needed to build one.

我正在学习中文，恰恰相反，我已经坚持学习了很多年。当然，为了拿下它，我尝试使用很多技术来帮助我学习中文。其中我尝试过 DVD 教程，视频教程，快速学习记忆卡等等。但是最后我意识到这里没有一个很好的中文学习记忆卡程序，所以出于学习的兴趣，我需要创建了一个。

I had found a program in Python to do some of the task. But sad to say it wasn't well written and after a few attempts at turning it upside down and inside out I came to the conclusion that it was better to start from scratch. Of course, a Web solution would be far better than a standalone one, because then all the other people in my Chinese class could share it, as well as any other learners out there. And of course, the server would be written in Go.

我发现了一使用 Python 写的程序可以完成一些这样的任务。但是遗憾的说，它写的不是很好，而且我经过几次尝试捣鼓它，得出一个结论是最好重头开始写一个。当然这个 Web 解决方案不是我一个人使用，它还会分享给其他学习中文的同学和更多学习中文的者。当然，这个服务我将使用 Go 语言来写。

The flashcards server is running at cict.bhtafe.edu.au:8000. The front page consists of a list of flashcard sets currently available, how you want a set displayed (random card order, Chinese, English or random), whether to display a set, add to it, etc. I've spent too much time building it - somehow my Chinese hasn't progressed much while I was doing it... It probably won't be too exciting as a program if you don't want to learn Chinese, but let's get into the structure.

flashcards 服务器运行在 cict.bhtafe.edu.au：8000。在当前页是目前的 flashcard 组列表，你想怎么显示（随机 card 的顺序，中文，英文或随机），是否显示一组，还要加上一套，我花了很多时间来构建它 - 不知何故我的中文还没有进展较快，而我该怎么做...它可能不会是太令人兴奋了一个程序，如果你不想学习中文，就让我们来看看程序的结构吧

Static pages

静态文件

Some pages will just have static content. These can be managed by a **fileServer**. For simplicity I put all of the static HTML pages and CSS files in the **html** directory and all of the JavaScript files in the **jscript** directory. These are then delivered by the Go code

有些页面只是一些静态内容。这些可以由 **FILESERVER** 管理。为了简单起见，我把所有的静态 HTML 页面和 CSS 文件放入 **html** 目录中，所有的 JavaScript 文件放日 **JScript** 目录。 然后这些交付给 Go 代码。

```
fileServer := http.FileServer("jscript", "/jscript/")
http.Handle("/jscript/", fileServer)

fileServer = http.FileServer("html", "/html/")
http.Handle("/html/", fileServer)
```

Templates

模板

The list of flashcard sets is open ended, depending on the number of files in a directory. These should not be hardcoded into an HTML page, but the content should be generated as needed. This is an obvious candidate for templates.

flashcard 组列表是开放式的，根据在一个目录中的文件的数量。这些不应该被硬编码到一个 HTML 页面，但内容应根据需要生成。这就是一个明显的候选模板。

The list of files in a directory is generated as a list of strings. These can then be displayed in a table using the template

目录中的文件列表可以被认为是一个字符串列表。 然后模板可将它们显示在一个表格中。

```
<table>
  {{range .}}
  <tr>
    <td>
      {{.}}
    </td>
  </tr>
</table>
```

The Chinese Dictionary

中文词典

Chinese is a complex language (aren't they all :-( ). The written form is hieroglyphic, that is "pictograms" instead of using an alphabet. But this written form has evolved over time, and even recently split into two forms: "traditional" Chinese as used in Taiwan and Hong Kong, and "simplified" Chinese as used in mainland China. While most of the characters are the same, about 1,000 are different. Thus a Chinese dictionary will often have two written forms of the same character.

中国是一个复杂的语言（不是所有人:-() 。书面形式的象形文字，那是“象形”，而不是使用一个字母，但这个书面形式随着时间的演变，以及最近甚至分裂成两种形式：“繁体”中文使用在中国的台湾和香港，和“简体”中文使用在中国大陆。虽然大多数的文字是相同的，但还是有大约 1000 中文字是不同的，因此，中文的字典往往有两种书面形式文字。

Most Westerners like me can't understand these characters. So there is a "Latinised" form called Pinyin which writes the characters in a phonetic alphabet based on the Latin alphabet. It isn't quite the Latin alphabet, because Chinese is a tonal language, and the Pinyin form has to show the tones (much like acccents in French and other European languages). So a typical dictionary has to show four things: the traditional form, the simplified form, the Pinyin and the English. For example,

和大多数西方人一样，我无法理解这些字符。所以，有一个“Latinised 的形式称为”拼音的字符写入以拉丁字母为基础的拼音第一个字母。这不是拉丁字母，因为这是中文的一种语言音调，和拼音形式显示的音调（很像在法国和其他欧洲语言的 acccents）。因此，一个典型的字典有四件事情：繁体形式和简化形式，拼音和英文显示。例如，

Traditional	Simplified	Pinyin	English
好	好	hǎo	good

But again there is a little complication. There is a free Chinese/English dictionary and even better, you can download it as a UTF-8 file, which Go is well suited to handle. In this, the Chinese characters are written in Unicode but the Pinyin characters are not: although there are Unicode characters for letters such as 'ǎ', many dictionaries including this one use the Latin 'a' and place the tone at the end of the word. Here it is the third tone, so "hǎo" is written as "hao3". This makes it easier for those who only have US keyboards and no Unicode editor to still communicate in Pinyin.

但是还是有点复杂。这里有一个更好的并且免费的中英文词典，你可以下载它是一个 UTF-8 的文件，非常适合 Go 去处理。在这，中文的字符集被写入在 Unicode 中但是拼音没有：尽管有 Unicode 字符的字母，如“ǎ”，很多词典包括本使用拉丁字母'a'和将音调放在词的结尾。在这里它是第三个音调，所以“hǎo”被写入“HAO3”。这使得它更容易为那些只有美国键盘和没有 Unicode 的编辑器的人们来与拼音沟通。

This data format mismatch is not a big deal: just that somewhere along the line, between the original text dictionary and the display in the browser, a data massage has to be performed. Go templates allow this to be done by defining a custom template, so I chose that route. Alternatives could have been to do this as the dictionary is read in, or in the Javascript to display the final characters.

这种数据格式不匹配并不是一个大的问题：只是在这行的某处地方，在原来的文本字典和显示在浏览器之间， 用数据来完成。Go 模板允许通过自定义一个模板， 所以我选择了这个思路。可以选择从 dictionary 中读取，或者由 JavaScript 来显示最终的字符。

The code for the Pinyin formatter is given below. Please don't bother reading it unless you are *really* interested in knowing the rules for Pinyin formatting.

拼音格式化的代码在下面给出。请不要仔细的阅读它，除非你真的有兴趣想知道拼音格式化的规则。

```
package pinyin

import (
        "io"
        "strings"
)

func PinyinFormatter(w io.Writer, format string, value ...interface{}) {
        line := value[0].(string)
        words := strings.Fields(line)
        for n, word := range words {
                // convert "u:" to "ü" if present
          uColon := strings.Index(word, "u:")
                if uColon != -1 {
                        parts := strings.SplitN(word, "u:", 2)
                        word = parts[0] + "ü" + parts[1]
                }
                println(word)
                // get last character, will be the tone if present
          chars := []rune(word)
                tone := chars[len(chars)-1]
                if tone == '5' {
                        words[n] = string(chars[0 : len(chars)-1])
```

```
                println("lost accent on", words[n])
                continue
            }
            if tone < '1' || tone > '4' {
                continue
            }
            words[n] = addAccent(word, int(tone))
        }
        line = strings.Join(words, ` `)
        w.Write([]byte(line))
}

var (
        // maps 'a1' to '\u0101' etc
 aAccent = map[int]rune{
            '1': '\u0101',
            '2': '\u00e1',
            '3': '\u01ce', // '\u0103',
  '4': '\u00e0'}
        eAccent = map[int]rune{
            '1': '\u0113',
            '2': '\u00e9',
            '3': '\u011b', // '\u0115',
  '4': '\u00e8'}
        iAccent = map[int]rune{
            '1': '\u012b',
            '2': '\u00ed',
            '3': '\u01d0', // '\u012d',
  '4': '\u00ec'}
```

```
		oAccent = map[int]rune{
			'1': '\u014d',
			'2': '\u00f3',
			'3': '\u01d2', // '\u014f',
		  '4': '\u00f2'}
		uAccent = map[int]rune{
			'1': '\u016b',
			'2': '\u00fa',
			'3': '\u01d4', // '\u016d',
		  '4': '\u00f9'}
		üAccent = map[int]rune{
			'1': 'ǖ',
			'2': 'ǘ',
			'3': 'ǚ',
			'4': 'ǜ'}
)

func addAccent(word string, tone int) string {
		/*
		  * Based on "Where do the tone marks go?"
		  * at http://www.pinyin.info/rules/where.html
		  */

		n := strings.Index(word, "a")
		if n != -1 {
			aAcc := aAccent[tone]
			// replace 'a' with its tone version
		  word = word[0:n] + string(aAcc) + word[(n+1):len(word)-1]
		} else {
```

```
n := strings.Index(word, "e")
if n != -1 {
        eAcc := eAccent[tone]
        word = word[0:n] + string(eAcc) +
                word[(n+1):len(word)-1]
} else {
        n = strings.Index(word, "ou")
        if n != -1 {
                oAcc := oAccent[tone]
                word = word[0:n] + string(oAcc) + "u" +
                        word[(n+2):len(word)-1]
        } else {
                chars := []rune(word)
                length := len(chars)
                // put tone onthe last vowel
 L:
                for n, _ := range chars {
                        m := length - n - 1
                        switch chars[m] {
                        case 'i':
                                chars[m] = iAccent[tone]
                                break L
                        case 'o':
                                chars[m] = oAccent[tone]
                                break L
                        case 'u':
                                chars[m] = uAccent[tone]
                                break L
                        case 'ü':
```

```
                                                chars[m] = üAccent[tone]
                                                break L
                                        default:
                                        }
                                }
                                word = string(chars[0 : len(chars)-1])
                        }
                }
        }

        return word
}
```

How this is used is illustrated by the function **lookupWord**. This is called in response to an HTML Form request to find the English words in a dictionary.

lookupWord 函数说明了怎样去使用它。这就是在字典中查找英文单词的 Html 表单请求的响应。

```
func lookupWord(rw http.ResponseWriter, req *http.Request) {
        word := req.FormValue("word")
        words := d.LookupEnglish(word)

        pinyinMap := template.FormatterMap {"pinyin": pinyin.PinyinFormatter}
        t, err := template.ParseFile("html/DictionaryEntry.html", pinyinMap)
        if err != nil {
                http.Error(rw, err.String(), http.StatusInternalServerError)
                return
        }
        t.Execute(rw, words)
```

```
}
```

The HTML code is

HTML 代码

```
<html>
  <body>
    <table border="1">
      <tr>
        <th>Word</th>
        <th>Traditional</th>
        <th>Simplified</th>
        <th>Pinyin</th>
        <th>English</th>
      </tr>
      {{with .Entries}}
      {{range .}}
      {.repeated section Entries}
      <tr>
        <td>{{.Word}}</td>
        <td>{{.Traditional}}</td>
        <td>{{.Simplified}}</td>
        <td>{{.Pinyin|pinyin}}</td>
        <td>
          <pre>
            {.repeated section Translations}
            {@|html}
            {.end}
```

```
            </pre>
          </td>
        </tr>
        {.end}
        {{end}}
        {{end}}
      </table>
  </body>
</html>
```

The Dictionary type

字典类型

The text file containing the dictionary has lines of the form

traditional simplified [pinyin] /translation/translation/.../

For example,

好 好 [hao3] /good/well/proper/good to/easy to/very/so/(suffix indicating completion or readiness)/

字典中的文本文件中一行的格式

繁体 简体 [拼音] /翻译/翻译/.../

例如，

好 好 [hao3] /good/well/proper/good to/easy to/very/so/(suffix indicating completion or readiness)/

We store each line as an **Entry** within the **Dictionary** package:

我们的 **Dictionary** 包里存储的一行 **Entry**:

```
type Entry struct {
```

```
        Traditional string
        Simplified string
        Pinyin          string
        Translations []string
}
```

The dictionary itself is just an array of these entries:

字典本身只是一个 entry 的数组：

```
type Dictionary struct {
        Entries []*Entry
}
```

Building the dictionary is easy enough. Just read each line and break the line into its various bits using simple string methods. Then add the line to the dictionary slice.

构建字典是很容易的。只要使用简单的字符串方法读取一行然后进行分割。最后添加到 dictionary slice 中。

Looking up entries in this dictionary is straightforward: just search through until we find the appropriate key. There are about 100,000 entries in this dictionary: brute force by a linear search is fast enough. If it were necessary, faster storage and search mechanisms could easily be used.

在这本词典中查找条目非常简单：只要通过搜索直到我们找到合适的答案。在这本字典有 10 万左右的条目：暴力的线性搜索速度不够快。如果有必要，可以使用更快的存储和搜索机制。

The original dictionary grows by people on the Web adding in entries as they see fit. Consequently it isn't that well organised and contains repetitions and multiple entries. So looking up any word - either

by Pinyin or by English - may return multiple matches. To cater for this, each lookup returns a "mini dictionary", just those lines in the full dictionary that match.

原词典发展的人添加条目他们认为网络更适合。所以它不是有条理的和包含了多个重复的条目。因此查找任何单词 - 无论是通过拼音或英语 - 可能返回多个匹配。为了应付这个问题，每个查询返回一个“mini dictionary”，只有那些在字典中匹配的。

The Dictionary code is

Dictionary 代码

```
package dictionary

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

type Entry struct {
        Traditional    string
        Simplified     string
        Pinyin         string
        Translations []string
}

func (de Entry) String() string {
        str := de.Traditional + ` ` + de.Simplified + ` ` + de.Pinyin
        for _, t := range de.Translations {
```

```
                str = str + "\n       " + t
        }
        return str
}

type Dictionary struct {
        Entries []*Entry
}

func (d *Dictionary) String() string {
        str := ""
        for n := 0; n < len(d.Entries); n++ {
                de := d.Entries[n]
                str += de.String() + "\n"
        }
        return str
}

func (d *Dictionary) LookupPinyin(py string) *Dictionary {
        newD := new(Dictionary)
        v := make([]*Entry, 0, 100)
        for n := 0; n < len(d.Entries); n++ {
                de := d.Entries[n]
                if de.Pinyin == py {
                        v = append(v, de)
                }
        }
        newD.Entries = v
        return newD
```

```
}

func (d *Dictionary) LookupEnglish(eng string) *Dictionary {
	newD := new(Dictionary)
	v := make([]*Entry, 0, 100)
	for n := 0; n < len(d.Entries); n++ {
		de := d.Entries[n]
		for _, e := range de.Translations {
			if e == eng {
				v = append(v, de)
			}
		}
	}
	newD.Entries = v
	return newD
}

func (d *Dictionary) LookupSimplified(simp string) *Dictionary {
	newD := new(Dictionary)
	v := make([]*Entry, 0, 100)

	for n := 0; n < len(d.Entries); n++ {
		de := d.Entries[n]
		if de.Simplified == simp {
			v = append(v, de)
		}
	}
	newD.Entries = v
	return newD
```

```
}

func (d *Dictionary) Load(path string) {

	f, err := os.Open(path)
	r := bufio.NewReader(f)
	if err != nil {
		println(err.Error())
		os.Exit(1)
	}

	v := make([]*Entry, 0, 100000)
	numEntries := 0
	for {
		line, err := r.ReadString('\n')
		if err != nil {
			break
		}
		if line[0] == '#' {
			continue
		}
		// fmt.Println(line)
	  trad, simp, pinyin, translations := parseDictEntry(line)

		de := Entry{
			Traditional:  trad,
			Simplified:   simp,
			Pinyin:       pinyin,
			Translations: translations}
```

```
                v = append(v, &de)
                numEntries++
        }
        // fmt.Printf("Num entries %d\n", numEntries)
 d.Entries = v
}

func parseDictEntry(line string) (string, string, string, []string) {
        // format is
 //      trad simp [pinyin] /trans/trans/.../
 tradEnd := strings.Index(line, " ")
        trad := line[0:tradEnd]
        line = strings.TrimSpace(line[tradEnd:])

        simpEnd := strings.Index(line, " ")
        simp := line[0:simpEnd]
        line = strings.TrimSpace(line[simpEnd:])

        pinyinEnd := strings.Index(line, "]")
        pinyin := line[1:pinyinEnd]
        line = strings.TrimSpace(line[pinyinEnd+1:])

        translations := strings.Split(line, "/")
        // includes empty at start and end, so
 translations = translations[1 : len(translations)-1]

        return trad, simp, pinyin, translations
}
```

Flash cards

Flash cards

Each individual flash card is of the type **Flashcard**

每个 flash card 的类型 **Flashcard**

```
type FlashCard struct {
        Simplified string
        English    string
        Dictionary *dictionary.Dictionary
}
```

At present we only store the simplified character and the english translation for that character. We also have a **Dictionary** which will contain only one entry for the entry we will have chosen somewhere.

目前我们只存储简体字符和该字符的英文翻译。我们将有选择性的加入一个 entry 到这里的 **Dictionary** 中。

A set of flash cards is defined by the type

flash cards 组的类型

```
type FlashCards struct {
        Name       string
        CardOrder string
        ShowHalf   string
        Cards      []*FlashCard
```

```
}
```

where the **CardOrder** will be "random" or "sequential" and the **ShowHalf** will be "RANDOM_HALF" or "ENGLISH_HALF" or "CHINESE_HALF" to determine which half of a new card is shown first.

其中 **CardOrder** 将是“random”或者“sequential”和 **ShowHalf** 将是“RANDOM_HALF”或“ENGLISH_HALF”或的“CHINESE_HALF”来确定一个新的卡中，意思是中文和英文其中有一个首先被显示。

The code for flash cards has nothing novel in it. We get data from the client browser and use JSON to create an object from the form data, and store the set of flashcards as a JSON string.

flash cards 的代码并没有新意。我们从浏览器客户端获取数据并根据表单的数据使用 JSON 来创建一个对象，并将其存储于 flashcards 中作为一个 JSON 字符串。

The Complete Server

完整的服务器

The complete server is

服务器代码如下

```
/* Server
 */

package main

import (
        "fmt"
```

```
	"io/ioutil"
	"net/http"
	"os"
	"regexp"
	"text/template"
)

import (
	"dictionary"
	"flashcards"
	"templatefuncs"
)

var d *dictionary.Dictionary

func main() {
	if len(os.Args) != 2 {
		fmt.Fprint(os.Stderr, "Usage: ", os.Args[0], ":port\n")
		os.Exit(1)
	}
	port := os.Args[1]

	// dictionaryPath := "/var/www/go/chinese/cedict_ts.u8"
 dictionaryPath := "cedict_ts.u8"
	d = new(dictionary.Dictionary)
	d.Load(dictionaryPath)
	fmt.Println("Loaded dict", len(d.Entries))

	http.HandleFunc("/", listFlashCards)
```

```
        //fileServer := http.FileServer("/var/www/go/chinese/jscript", "/jscript/")
 fileServer := http.StripPrefix("/jscript/", http.FileServer(http.Dir("jscript")))
        http.Handle("/jscript/", fileServer)
        // fileServer = http.FileServer("/var/www/go/chinese/html", "/html/")
 fileServer = http.StripPrefix("/html/", http.FileServer(http.Dir("html")))
        http.Handle("/html/", fileServer)


        http.HandleFunc("/wordlook", lookupWord)
        http.HandleFunc("/flashcards.html", listFlashCards)
        http.HandleFunc("/flashcardSets", manageFlashCards)
        http.HandleFunc("/searchWord", searchWord)
        http.HandleFunc("/addWord", addWord)
        http.HandleFunc("/newFlashCardSet", newFlashCardSet)


        // deliver requests to the handlers
 err := http.ListenAndServe(port, nil)
        checkError(err)
        // That's it!
}


func indexPage(rw http.ResponseWriter, req *http.Request) {
        index, _ := ioutil.ReadFile("html/index.html")
        rw.Write([]byte(index))
}


func lookupWord(rw http.ResponseWriter, req *http.Request) {
        word := req.FormValue("word")
        words := d.LookupEnglish(word)
```

```
		//t := template.New("PinyinTemplate")
	t := template.New("DictionaryEntry.html")
		t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
		t, err := t.ParseFiles("html/DictionaryEntry.html")
		if err != nil {
			http.Error(rw, err.Error(), http.StatusInternalServerError)
			return
		}
		t.Execute(rw, words)
}

type DictPlus struct {
		*dictionary.Dictionary
		Word		string
		CardName string
}

func searchWord(rw http.ResponseWriter, req *http.Request) {
		word := req.FormValue("word")
		searchType := req.FormValue("searchtype")
		cardName := req.FormValue("cardname")

		var words *dictionary.Dictionary
		var dp []DictPlus
		if searchType == "english" {
			words = d.LookupEnglish(word)
			d1 := DictPlus{Dictionary: words, Word: word, CardName: cardName}
			dp = make([]DictPlus, 1)
			dp[0] = d1
```

```
	} else {
		words = d.LookupPinyin(word)
		numTrans := 0
		for _, entry := range words.Entries {
			numTrans += len(entry.Translations)
		}
		dp = make([]DictPlus, numTrans)
		idx := 0
		for _, entry := range words.Entries {
			for _, trans := range entry.Translations {
				dict := new(dictionary.Dictionary)
				dict.Entries = make([]*dictionary.Entry, 1)
				dict.Entries[0] = entry
				dp[idx] = DictPlus{
					Dictionary: dict,
					Word:       trans,
					CardName:   cardName}
				idx++
			}
		}
	}

	//t := template.New("PinyinTemplate")
t := template.New("ChooseDictionaryEntry.html")
	t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
	t, err := t.ParseFiles("html/ChooseDictionaryEntry.html")
	if err != nil {
		fmt.Println(err.Error())
		http.Error(rw, err.Error(), http.StatusInternalServerError)
```

```
                return
        }
        t.Execute(rw, dp)
}

func newFlashCardSet(rw http.ResponseWriter, req *http.Request) {
        defer http.Redirect(rw, req, "http:/flashcards.html", 200)

        newSet := req.FormValue("NewFlashcard")
        fmt.Println("New cards", newSet)
        // check against nasties:
 b, err := regexp.Match("[/$~]", []byte(newSet))
        if err != nil {
                return
        }
        if b {
                fmt.Println("No good string")
                return
        }

        flashcards.NewFlashCardSet(newSet)
        return
}

func addWord(rw http.ResponseWriter, req *http.Request) {
        url := req.URL
        fmt.Println("url", url.String())
        fmt.Println("query", url.RawQuery)
```

```
		word := req.FormValue("word")
		cardName := req.FormValue("cardname")
		simplified := req.FormValue("simplified")
		pinyin := req.FormValue("pinyin")
		traditional := req.FormValue("traditional")
		translations := req.FormValue("translations")

		fmt.Println("word is ", word, " card is ", cardName,
			" simplified is ", simplified, " pinyin is ", pinyin,
			" trad is ", traditional, " trans is ", translations)
		flashcards.AddFlashEntry(cardName, word, pinyin, simplified,
			traditional, translations)
		// add another card?
	addFlashCards(rw, cardName)
}

func listFlashCards(rw http.ResponseWriter, req *http.Request) {

		flashCardsNames := flashcards.ListFlashCardsNames()
		t, err := template.ParseFiles("html/ListFlashcards.html")
		if err != nil {
			http.Error(rw, err.Error(), http.StatusInternalServerError)
			return
		}
		t.Execute(rw, flashCardsNames)
}

/*
 * Called from ListFlashcards.html on form submission
```

```
 */
func manageFlashCards(rw http.ResponseWriter, req *http.Request) {


	set := req.FormValue("flashcardSets")
	order := req.FormValue("order")
	action := req.FormValue("submit")
	half := req.FormValue("half")
	fmt.Println("set chosen is", set)
	fmt.Println("order is", order)
	fmt.Println("action is", action)


	cardname := "flashcardSets/" + set


	//components := strings.Split(req.URL.Path[1:], "/", -1)
 //cardname := components[1]
 //action := components[2]
 fmt.Println("cardname", cardname, "action", action)
	if action == "Show cards in set" {
		showFlashCards(rw, cardname, order, half)
	} else if action == "List words in set" {
		listWords(rw, cardname)
	} else if action == "Add cards to set" {
		addFlashCards(rw, set)
	}
}


func showFlashCards(rw http.ResponseWriter, cardname, order, half string) {
	fmt.Println("Loading card name", cardname)
	cards := new(flashcards.FlashCards)
```

```
		//cards.Load(cardname, d)
//flashcards.SaveJSON(cardname + ".json", cards)
flashcards.LoadJSON(cardname, &cards)
		if order == "Sequential" {
			cards.CardOrder = "SEQUENTIAL"
		} else {
			cards.CardOrder = "RANDOM"
		}
		fmt.Println("half is", half)
		if half == "Random" {
			cards.ShowHalf = "RANDOM_HALF"
		} else if half == "English" {
			cards.ShowHalf = "ENGLISH_HALF"
		} else {
			cards.ShowHalf = "CHINESE_HALF"
		}
		fmt.Println("loaded cards", len(cards.Cards))
		fmt.Println("Card name", cards.Name)

		//t := template.New("PinyinTemplate")
t := template.New("ShowFlashcards.html")
		t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
		t, err := t.ParseFiles("html/ShowFlashcards.html")
		if err != nil {
			fmt.Println(err.Error())
			http.Error(rw, err.Error(), http.StatusInternalServerError)
			return
		}
		err = t.Execute(rw, cards)
```

```
	if err != nil {
		fmt.Println("Execute error " + err.Error())
		http.Error(rw, err.Error(), http.StatusInternalServerError)
		return
	}
}

func listWords(rw http.ResponseWriter, cardname string) {
	fmt.Println("Loading card name", cardname)
	cards := new(flashcards.FlashCards)
	//cards.Load(cardname, d)
 flashcards.LoadJSON(cardname, cards)
	fmt.Println("loaded cards", len(cards.Cards))
	fmt.Println("Card name", cards.Name)

	//t := template.New("PinyinTemplate")
 t := template.New("ListWords.html")
	if t.Tree == nil || t.Root == nil {
		fmt.Println("New t is an incomplete or empty template")
	}
	t = t.Funcs(template.FuncMap{"pinyin": templatefuncs.PinyinFormatter})
	t, err := t.ParseFiles("html/ListWords.html")
	if t.Tree == nil || t.Root == nil {
		fmt.Println("Parsed t is an incomplete or empty template")
	}

	if err != nil {
		fmt.Println("Parse error " + err.Error())
		http.Error(rw, err.Error(), http.StatusInternalServerError)
```

```
                return
        }
        err = t.Execute(rw, cards)
        if err != nil {
                fmt.Println("Execute error " + err.Error())
                http.Error(rw, err.Error(), http.StatusInternalServerError)
                return
        }
        fmt.Println("No error ")
}

func addFlashCards(rw http.ResponseWriter, cardname string) {
        t, err := template.ParseFiles("html/AddWordToSet.html")
        if err != nil {
                fmt.Println("Parse error " + err.Error())
                http.Error(rw, err.Error(), http.StatusInternalServerError)
                return
        }
        cards := flashcards.GetFlashCardsByName(cardname, d)
        t.Execute(rw, cards)
        if err != nil {
                fmt.Println("Execute error " + err.Error())
                http.Error(rw, err.Error(), http.StatusInternalServerError)
                return
        }

}

func checkError(err error) {
```

```
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Other Bits: JavaScript and CSS

其他: JavaScript 和 CSS

On request, a set of flashcards will be loaded into the browser. A much abbreviated set is shown below. The display of these cards is controlled by JavaScript and CSS files. These aren't relevant to the Go server so are omitted. Those interested can download the code.

根据需求,flashcards 组将会被加载到浏览器中。下面展示了一个简短的 HTML 页。cards 的显示由 JavaScript 和 CSS 文件控制。这些都不是和 Go 服务器相关的技术所以在这省略了。有兴趣的可以下载代码。

```
<html>
  <head>
    <title>
      Flashcards for Common Words
    </title>

    <link type="text/css" rel="stylesheet"
          href="/html/CardStylesheet.css">
    </link>

    <script type="text/javascript"
            language="JavaScript1.2" src="/jscript/jquery.js">
      <!-- empty -->
```

```
    </script>

    <script type="text/javascript"
                language="JavaScript1.2" src="/jscript/slideviewer.js">
        <!-- empty -->
    </script>

    <script type="text/javascript"
                language="JavaScript1.2">
        cardOrder = RANDOM;
        showHalfCard = RANDOM_HALF;
    </script>
</head>
<body onload="showSlides();">
    <h1>
        Flashcards for Common Words
    </h1>
    <p>
        <div class="card">

            <div class="english">
                <div class="vcenter">
                    hello
                </div>
            </div>

                <div class="pinyin">
                    <div class="vcenter">
                        nǐ hǎo
```

```
            </div>


        </div>


        <div class="traditional">
            <div class="vcenter">
                你好
            </div>
        </div>


        <div class="simplified">
            <div class="vcenter">
                你好
            </div>

        </div>


        <div class ="translations">
            <div class="vcenter">
                hello <br />
                hi <br />
                how are you? <br />
            </div>

        </div>
    </div>
    <div class="card">
        <div class="english">
            <div class="vcenter">
```

```
            hello (interj., esp. on telephone)
        </div>
    </div>

        <div class="pinyin">

            <div class="vcenter">
                wèi
            </div>
        </div>

        <div class="traditional">
            <div class="vcenter">
                喂
            </div>
        </div>

        <div class="simplified">

            <div class="vcenter">
                喂
            </div>
        </div>

        <div class ="translations">
            <div class="vcenter">
                hello (interj., esp. on telephone) <br />
                hey <br />
```

```
                        to feed (sb or some animal) <br />
                    </div>
                </div>
            </div>
        </p>

        <p class ="return">
            Press <Space> to continue
            <br/>
            <a href="http:/flashcards.html"> Return to Flash Cards list</a>
        </p>
    </body>
</html>
```

HTML

关于 HTML

The Web Server was originally created to serve HTML documents. Now it is used to serve all sorts of documents as well as data of different kinds. Nevertheless, HTML is still the main document type delivered over the Web. Go has basic mechanisms for parsing HTML documents, which are covered in this chapter

Web 服务器的建立最开始是用来提供 HTML 文件服务的。现在它能为各种类型的文档和各种不同类型的数据提供服务。然而，HTML 依然是互联网网络中传递的主要文档类型。Go 有一套基本机制来解析 HTML，本章主要阐述此内容。

Introduction

介绍

The Web was originally created to serve HTML documents. Now it is used to serve all sorts of documents as well as data of dirrent kinds. Nevertheless, HTML is still the main document type delivered over the Web

Web 服务器的建立最开始是用来提供 HTML 文件服务的。现在它为各种类型的文档和各种不同类型的数据提供服务。然而，HTML 仍然是互联网网络中传递的主要文档类型。

HTML has been through a large number of versions, and HTML 5 is currently under development. There have also been many "vendor" versions of HTML, introducing tags that never made it into standards.

HTML 经历了大量的版本变迁，HTML5 目前还在开发阶段。此外出现不少“独立供应商”版的 HTML，但引入的标签从来没有做成标准。

HTML is simple enough to be edited by hand. Consequently, many HTML documents are "ill formed", not following the syntax of the language. HTML parsers generally are not very strict, and will accept many "illegal" documents.

HTML 足够简单，以至于可以纯手工编写。因此，许多 HTML 文件格式不规范，没有遵守标准准则的语法。HTML 解析器通常也不是很严格，而且能接受大多数格式“不严格”的文件。

There wasn't much in earlier versions of Go about handling HTML documents - basically, just a tokenizer. The incomplete nature of the package has led to its removal for Go 1. It can still be found in the **exp** (experimental) package if you really need it. No doubt some improved form will become available in a later version of Go, and then it will be added back into this book.

在早期版本的 Go 没有太多关于处理 HTML 文件的细节--基本上只是一个分词器。不完整的原始包在 Go 1 的版本中已移除。如果你真的需要它，仍然可以在 **exp**(试验)包中找到它。 毫无疑问，Go 未来版本在这方面会有一些改进的地方，那么到时将会添加到本书中。

There is limited support for HTML in the XML package, discussed in the next chapter.

在 XML 包中对 HTML 的支持是有限的，在下一章将会讨论。

Conclusion

结论

There isn't anything to this package at present as it is still under development.

目前这个包没有内容，因为它目前仍处于开发阶段。

XML

XML is a significant markup language mainly intended as a means of serialising data structures as a text document. Go has basic support for XML document processing.

XML 是一种重要的标记语言，旨在把数据结构序列化成文本文档。Go 基本支持 XML 文档处理。

Introduction

介绍

XML is now a widespread way of representing complex data structures serialised into text format. It is used to describe documents such as DocBook and XHTML. It is used in specialised markup languages such as MathML and CML (Chemistry Markup Language). It is used to encode data as SOAP messages for Web Services, and the Web Service can be specified using WSDL (Web Services Description Language).

现在 XML 是一个用序列化的文本格式表现复杂数据结构的普遍方式。它被用来描述文档例如 DocBook 和 XHTML。它还用于描述专用标记语言如 MathML 和 CML(化学标记语言)。Web 服务中它还用来将数据编码成 SOAP 消息,Web 服务也可以指定使用 WSDL(Web 服务描述语言)。

At the simplest level, XML allows you to define your own tags for use in text documents. Tags can be nested and can be interspersed with text. Each tag can also contain attributes with values. For example,

在最简单的层次上,XML 允许您定义您个人标记用于文本文档。标签可以嵌套,也穿插在文本里。每个标记也可以包含属性与值。例如,

```
<person>
  <name>
```

```
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
    jan@newmarch.name
  </email>
  <email type="work">
    j.newmarch@boxhill.edu.au
  </email>
</person>
```

The structure of any XML document can be described in a number of ways:

任何 XML 文档的结构可以用多种方式描述:

- A document type definition DTD is good for describing structure
- XML schema are good for describing the data types used by an XML document
- RELAX NG is proposed as an alternative to both

- 一个文档类型定义 DTD 有利于表现数据结构
- 在一个 XML 文档中，使用 XML 模式有利于描述数据类型
- RELAX NG 提出了替代方案

There is argument over the relative value of each way of defining the structure of an XML document. We won't buy into that, as Go does not suport any of them. Go cannot check for validity of any document against a schema, but only for well-formedness.

人们总会争论定义 XML 文档结构的每一个方式的好坏。我们不会陷入其中,因为 Go 不支持其中任何一个。Go 不能检查任何文档模式的有效性，但只知道良构性。

Four topics are discussed in this chapter: parsing an XML stream, marshalling and unmarshalling Go data into XML, and XHTML.

在本章中讨论四个主题:解析一个 XML 流,编组和解组 Go 数据成为 XML 和 XHTML。

Parsing XML

解析 XML

Go has an XML parser which is created using **NewParser**. This takes an **io.Reader** as parameter and returns a pointer to **Parser**. The main method of this type is **Token** which returns the next token in the input stream. The token is one of the types **StartElement**, **EndElement**, **CharData**, **Comment**, **ProcInst** or **Directive**.

Go 有一个使用 **NewParser**.创建的 XML 解析器。这需要一个 **io.Reader** 作为参数,并返回一个指向 **Parser** 的指针。这个类型的主要方法是 **Token** ,这个方法返回输入流中的下一个标记。该标记是 **StartElement**, **EndElement**, **CharData**, **Comment**, **ProcInst** 和 **Directive** 其中一种。

The types are

这些类有

StartElement

The type **StartElement** is a structure with two field types:

StartElement 类型是一个包含两个字段的结构:

```
type StartElement struct {
    Name Name
    Attr []Attr
}
```

```
type Name struct {
	Space, Local string
}

type Attr struct {
	Name	Name
	Value string
}
```

EndElement

This is also a structure

同样也是一个结构

```
type EndElement struct {
	Name Name
}
```

CharData

This type represents the text content enclosed by a tag and is a simple type

这个类表示一个被标签包住的文本内容，是一个简单类。

```
type CharData []byte
```

Comment

Similarly for this type

这个类也很简洁

```
type Comment []byte
```

ProcInst

A ProcInst represents an XML processing instruction of the form <?target inst?>

一个 ProcInst 表示一个 XML 处理指令形式，如<target inst?>

```
type ProcInst struct {
    Target string
    Inst   []byte
}
```

Directive

A Directive represents an XML directive of the form <!text>. The bytes do not include the <! and > markers.

一个指令用 XML 指令<!文本>的形式表示，内容不包含< !和> 构成部分。

```
type Directive []byte
```

A program to print out the tree structure of an XML document is

打印 XML 文档的树结构的一个程序，代码如下

```
/* Parse XML
```

```
 */


package main


import (
	"encoding/xml"
	"fmt"
	"io/ioutil"
	"os"
	"strings"
)


func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "file")
		os.Exit(1)
	}
	file := os.Args[1]
	bytes, err := ioutil.ReadFile(file)
	checkError(err)
	r := strings.NewReader(string(bytes))


	parser := xml.NewDecoder(r)
	depth := 0
	for {
		token, err := parser.Token()
		if err != nil {
			break
		}
```

```
                switch t := token.(type) {
                case xml.StartElement:
                        elmt := xml.StartElement(t)
                        name := elmt.Name.Local
                        printElmt(name, depth)
                        depth++
                case xml.EndElement:
                        depth--
                        elmt := xml.EndElement(t)
                        name := elmt.Name.Local
                        printElmt(name, depth)
                case xml.CharData:
                        bytes := xml.CharData(t)
                        printElmt("\""+string([]byte(bytes))+"\"", depth)
                case xml.Comment:
                        printElmt("Comment", depth)
                case xml.ProcInst:
                        printElmt("ProcInst", depth)
                case xml.Directive:
                        printElmt("Directive", depth)
                default:
                        fmt.Println("Unknown")
                }
        }
}

func printElmt(s string, depth int) {
        for n := 0; n < depth; n++ {
                fmt.Print("    ")
```

```
            }
            fmt.Println(s)
}

func checkError(err error) {
            if err != nil {
                        fmt.Println("Fatal error ", err.Error())
                        os.Exit(1)
            }
}
```

Note that the parser includes all CharData, including the whitespace between tags.

注意,解析器包括所有文本节点,包括标签之间的空白。

If we run this program against the **person** data structure given earlier, it produces

如果我们运行这个程序对前面给出的 **person** 数据结构,它就会打印出

```
person
   "
   "
   name
      "
      "
      family
         " Newmarch "
      family
      "
      "
      personal
```

```
        " Jan "
      personal
      "
   "
   name
   "
   "
   email
      "
      jan@newmarch.name
   "
   email
   "
   "
   email
      "
      j.newmarch@boxhill.edu.au
   "
   email
   "
"
person
"
"
```

Note that as no DTD or other XML specification has been used, the tokenizer correctly prints out all the white space (a DTD may specify that the whitespace can be ignored, but without it that assumption cannot be made.)

注意,因为没有使用 DTD 或其他 XML 规范, tokenizer 正确地打印出所有的空白(一个 DTD 可能指定可以忽略空格,但是没有它假设就不能成立。)

There is a potential trap in using this parser. It re-uses space for strings, so that once you see a token you need to copy its value if you want to refer to it later. Go has methods such as **func (c CharData) Copy() CharData** to make a copy of data.

在使用这个解析器过程中有一个潜在的陷阱值得注意:它会为字符串重新利用空间,所以,一旦你看到一个你想要复制它的值的标记,假设你想稍后引用它的话,Go 有类似的方法如 **func (c CharData) Copy() CharData** 来复制数据。

Unmarshalling XML

反编排 XML

Go provides a function **Unmarshal** and a method **func (*Parser) Unmarshal** to unmarshal XML into Go data structures. The unmarshalling is not perfect: Go and XML are different languages.

Go 提供一个函数 **Unmarshal** 和一个方法调用 **func (*Parser) Unmarshal** 解组 XML 转化为 Go 数据结构。解组并不是完美的:Go 和 XML 毕竟是是两个不同的语言。

We consider a simple example before looking at the details. We take the XML document given earlier of

我们先考虑一个简单的例子再查看细节。我们用前面给出的 XML 文档

```
<person>
  <name>
    <family> Newmarch </family>
    <personal> Jan </personal>
  </name>
  <email type="personal">
```

```
      jan@newmarch.name
   </email>
   <email type="work">
      j.newmarch@boxhill.edu.au
   </email>
</person>
```

We would like to map this onto the Go structures

接下来我们想把这个文档映射到 Go 结构

```
type Person struct {
        Name Name
        Email []Email
}

type Name struct {
        Family string
        Personal string
}

type Email struct {
        Type string
        Address string
}
```

This requires several comments:

这里需要一些说明:

1. Unmarshalling uses the Go reflection package. This requires that all fields by public i.e. start with a capital letter. Earlier versions of Go used case-insensitive matching to match fields such as the XML string "name" to the field **Name**. Now, though, *case-sensitive* matching is used. To perform a match, the structure fields must be tagged to show the XML string that will be matched against. This changes **Person** to

```
2.
3.  type Person struct {
4.          Name Name `xml:"name"`
5.          Email []Email `xml:"email"`
6.  }
```

7. While tagging of fields can attach XML strings to fields, it can't do so with the names of the structures. An additional field is required, with field name "XMLName". This only affects the top-level struct, **Person**

```
8.
9.  type Person struct {
10.         XMLName Name `xml:"person"`
11.         Name Name `xml:"name"`
12.         Email []Email `xml:"email"`
13. }
```

14. Repeated tags in the map to a slice in Go
15. Attributes within tags will match to fields in a structure only if the Go field has the tag ",attr". This occurs with the field **Type** of **Email**, where matching the attribute "type" of the "email" tag requires **`xml:"type,attr"`**
16. If an XML tag has no attributes and only has character data, then it matches a **string** field by the same name (case-sensitive, though). So the tag **`xml:"family"`** with character data "Newmarch" maps to the string field **Family**

17. But if the tag has attributes, then it must map to a structure. Go assigns the character data to the field with tag **,chardata**. This occurs with the "email" data and the field **Address** with tag **,chardata**

1. 使用 Go reflection 包去解组。这要求所有字段是公有，也就是以一个大写字母开始。早期版本的 Go 使用不区分大小写匹配来匹配字段,例如 XML 标签“name”对应 **Name** 字段。但是现在使用 **case-sensitive** 匹配，要执行一个匹配,结构字段后必须用标记来显示 XML 标签名,以应付匹配。**Person** 修改下应该是

```
2.
3.  type Person struct {
4.          Name Name `xml:"name"`
5.          Email []Email `xml:"email"`
6.  }
```

7. 虽然标记结构字段可以使用 XML 字符串,但是对于结构名不能这么做 ，这个解决办法是增加一个额外字段,命名“XMLName”。这只会影响上级结构，修改 **Person** 如下

```
8.
9.  type Person struct {
10.         XMLName Name `xml:"person"`
11.         Name Name `xml:"name"`
12.         Email []Email `xml:"email"`
13. }
```

14. 重复标记会映射到 Go 的 slice
15. 要包含属性的标签准确匹配对应的结构字段，只有在 Go 字段后标记”,attr”。举个下面例子中 **Email** 类型的 **Type** 字段，需要标记**`xml:"type,attr"`**才能匹配带有“type”属性的“email”

16. 如果一个 XML 标签没有属性而且只有文本内容,那么它匹配一个 string 字段是通过相同的名称(区分大小写的,不过如此)。所以标签`xml:"family"`将对应着文本”Newmarch”映射到 Family 的 string 字段中
17. 但如果一个标签带有属性,那么它这个特征必须反映到一个结构。Go 在字段后标记着 ,chardata 的文字。如下面例子中通过 Address 后标记,chardata 的字段来获取 email 的文本值

A program to unmarshal the document above is

解组上面文档的一个程序

```
/* Unmarshal
 */

package main

import (
        "encoding/xml"
        "fmt"
        "os"
        //"strings"
)

type Person struct {
        XMLName Name       `xml:"person"`
        Name    Name       `xml:"name"`
        Email   []Email `xml:"email"`
}

type Name struct {
```

```
        Family     string `xml:"family"`
        Personal string `xml:"personal"`
}

type Email struct {
        Type       string `xml:"type,attr"`
        Address string `xml:",chardata"`
}

func main() {
        str := `<?xml version="1.0" encoding="utf-8"?>
<person>
   <name>
      <family> Newmarch </family>
      <personal> Jan </personal>
   </name>
   <email type="personal">
      jan@newmarch.name
   </email>
   <email type="work">
      j.newmarch@boxhill.edu.au
   </email>
</person>`

        var person Person

        err := xml.Unmarshal([]byte(str), &person)
        checkError(err)
```

```
        // now use the person structure e.g.
 fmt.Println("Family name: \"" + person.Name.Family + "\"")
        fmt.Println("Second email address: \"" + person.Email[1].Address + "\"")
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

(Note the spaces are correct.). The strict rules are given in the package specification.

(注意空间是正确的)。Go 在包详解中给出了严格的规则。

Marshalling XML

编组 XML

Go 1 also has support for marshalling data structures into an XML document. The function is

Go1 也支持将数据结构编组为 XML 文档的。这个函数是

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

This was used as a check in the last two lines of the previous program.

这是用来检查前面程序的最后两行

XHTML

XHTML

HTML does not conform to XML syntax. It has unterminated tags such as '
'. XHTML is a cleanup of HTML to make it compliant to XML. Documents in XHTML can be managed using the techniques above for XML.

HTML 并不符合 XML 语法。 它包含无闭端的标签如“
”。XHTML 是 HTML 的一个自身兼容 XML 的子集。在 XHTML 文档中可以使用操作 XML 的技术。

HTML

There is some support in the XML package to handle HTML documents even though they are not XML-compliant. The XML parser discussed earlier can handle many HTML documents if it is modified by

XML 包的部分方法可支持处理 HTML 文档,即使他们本身不具备 XML 兼容性。前面讨论的 XML 解析器修改下就可以处理大部分 HTML 文件

```
parser := xml.NewDecoder(r)
parser.Strict = false
parser.AutoClose = xml.HTMLAutoClose
parser.Entity = xml.HTMLEntity
```

Conclusion

结论

Go has basic support for dealing with XML strings. It does not as yet have mechanisms for dealing with XML specification languages such as XML Schema or Relax NG.

Go基本支持对XML字符的处理，而且它不像有着针对XML专用语言如XML Schema或Relax NG的处理机制。

Remote Procedure Call

远程过程调用

Introduction

介绍

Socket and HTTP programming use a message-passing paradigm. A client sends a message to a server which usually sends a message back. Both sides are responsible for creating messages in a format understood by both sides, and in reading the data out of those messages.

Socket 和 HTTP 编程 使用的是一种消息传递模式. 一个客户端发送了一个消息给服务器，通常会等回一个响应消息。两边都要创建出一种双方可理解的格式，然后从里面读出数据的实体。

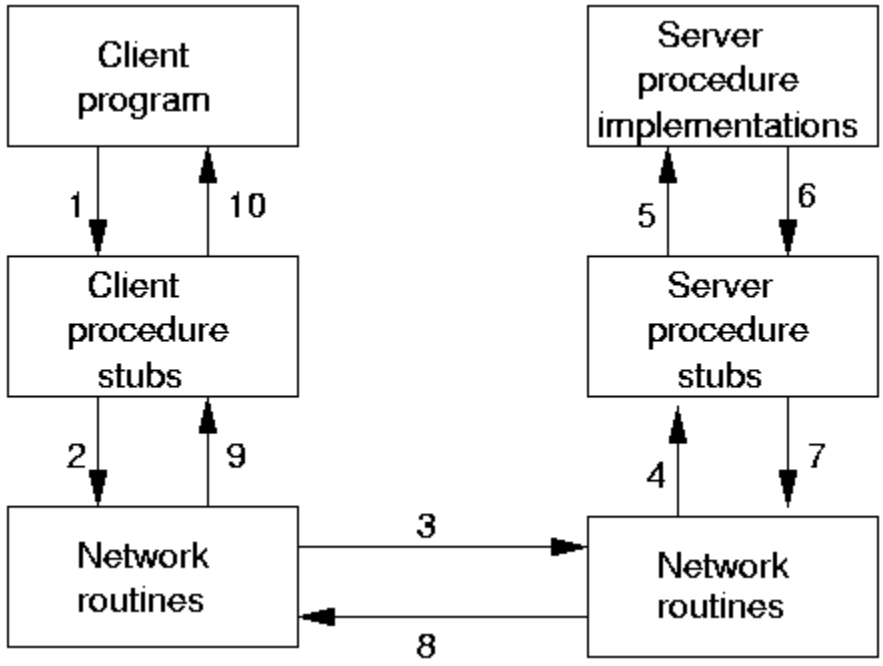
However, most standalone applications do not make so much use of message passing techniques. Generally the preferred mechanism is that of the function (or method or procedure) call. In this style, a program will call a function with a list of parameters, and on completion of the function call will have a set of return values. These values may be the function value, or if addresses have been passed as parameters then the contents of those addresses might have been changed.

然而，大多数独立主机应用不会做太多的消息传递技术。一般来说，函数调用(或者被称作method/procedure)的使用更为普遍。在函数风格下，程序会调用函数时会传入一系列参数，然后函数调用完毕后会返回一系列返回值。这些返回值会成为函数的值，或者传递进函数的是参数的地址引用，那么参数值可能最后会被修改。

The remote procedure call is an attempt to bring this style of programming into the network world. Thus a client will make what looks to it like a normal procedure call. The client-side will package this into a network message and transfer it to the server. The server will unpack this and turn it back into a procedure call on the server side. The results of this call will be packaged up for return to the client.

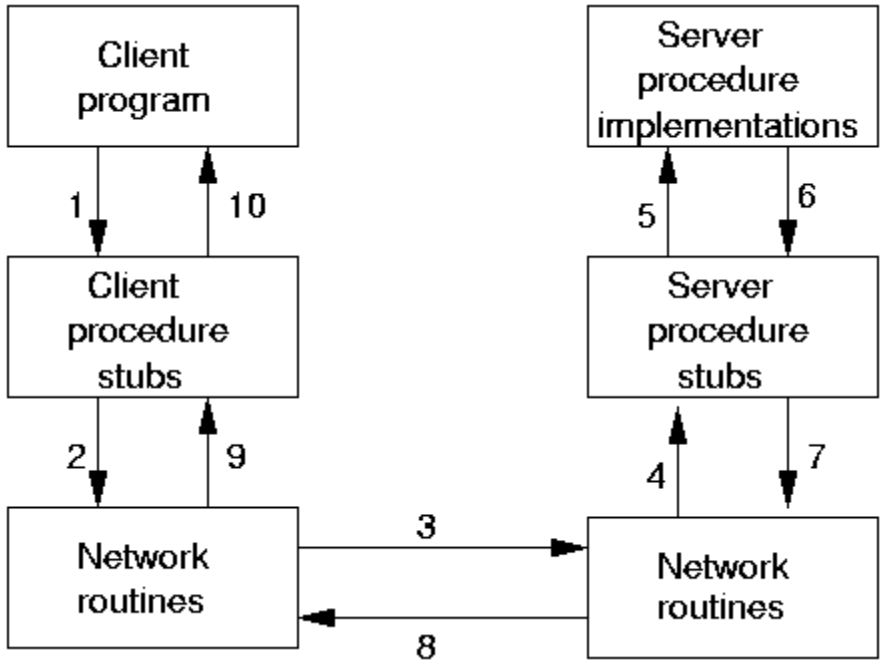
远程过程调用的初衷就是把这种风格带入网络世界。客户调用时候会让这一切看起来像是函数调用，而客户端会打包这些数据成为消息，然后传递到远端服务器。服务器再拆解包，然后把它变成在服务器端的过程调用，而最后的返回结果会被打包传回给客户。

Diagrammatically it looks like



where the steps are

用图示表示的话，看起来就会是这个样子



经历如下几个步骤

1. The client calls the local stub procedure. The stub packages up the parameters into a network message. This is called *marshalling*.
2. Networking functions in the O/S kernel are called by the stub to send the message.
3. The kernel sends the message(s) to the remote system. This may be connection-oriented or connectionless.

4. A server stub unmarshals the arguments from the network message.
5. The server stub executes a local procedure call.
6. The procedure completes, returning execution to the server stub.
7. The server stub marshals the return values into a network message.
8. The return messages are sent back.
9. The client stub reads the messages using the network functions.
10. The message is unmarshalled. and the return values are set on the stack for the local process.

1. 客户调用本地存根节点过程，存根节点会把参数打包成网络消息，这个过程被称为*编组*
2. OS 内核里的网络通信函数会被存根节点调用来发送消息。
3. 内核把消息传递给远端系统。这个可以使面向连接的或者是无连接传输模式。
4. 服务器端的存根节点会把参数从网络消息中拆解出来。
5. 服务器端的存根节点会执行一个本地过程调用
6. 等到过程完成，返回之行结果给服务器端的存根节点。
7. 服务器存根节点会把返回值编组成网络消息。
8. 消息被返回
9. 客户端存根节点用网络通信函数读取消息
10. 消息被拆解。然后返回值被放到本地程序的堆栈内。

There are two common styles for implementing RPC. The first is typified by Sun's RPC/ONC and by CORBA. In this, a specification of the service is given in some abstract language such as CORBA IDL (interface definition language). This is then compiled into code for the client and for the server. The client then writes a normal program containing calls to a procedure/function/method which is linked to the generated client-side code. The server-side code is actually a server itself, which is linked to the procedure implementation that you write.

远程过程调用有两种普遍使用的风格。第一个是以 SUN 开发的 CORBA 的 RPC/ONC 为代表。这里，服务的描述被某种像 CORBA IDL(接口定义语言)抽象语言提供，然后编译成可执行代码分别部署在 client 端和 server 端。客户接着就可以写一个常规的程序去连接那个生成出来的方法,而 server 端的代码实际上就是 server 服务的实体，然后连接到你实现的程序。

In this way, the client-side code is almost identical in appearance to a normal procedure call. Generally there is a little extra code to locate the server. In Sun's ONC, the address of the server must be known; in CORBA a naming service is called to find the address of the server; In Java RMI, the IDL is Java itself and a naming service is used to find the address of the service.

这样，客户端代码就基本上跟一个普通的程序调用没什么区别了。一般来说，在 server 端部署的代码量会有点多。在 SUN 开发的的 ONC 上，server 端的地址必须是公开的。在 CORBA 里面，一个命名服务会启动去寻找服务器的地址。而在 JAVA RMI 中，IDL 由 Java 类库实现，然后命名服务会被调用去寻找服务器地址。

In the second style, you have to make use of a special client API. You hand the function name and its parameters to this library on the client side. On the server side, you have to explicitly write the server yourself, as well as the remote procedure implementation.

在第二种风格中，你会用到一些特别的 client 端 API，这些 API，包括函数名，和参数是在生成的 client 代码中的。与此不同的是，在 server 端，你必须用你的手把代码敲出来，包括这些远程函数的实现。

This approach is used by many RPC systems, such as Web Services. It is also the approach used by Go's RPC.

很多 RPC 系统都采用了这种方法，比如 Web Services. 当然，Go 的 PRC 也采用了这样的方法。

Go RPC

Go RPC

Go's RPC is so far unique to Go. It is different to the other RPC systems, so a Go client will only talk to a Go server. It uses the Gob serialisation system discussed in chapter X, which defines the data types which can be used.

Go 的 RPC 是非常独特的。它与别的 RPC 系统不同，所以 Go 的 client 只能跟 Go 的 server 对话。它被用在第十章讨论的 Gob 序列化系统里面，用来定义可被使用的数据类型。

RPC systems generally make some restrictions on the functions that can be called across the network. This is so that the RPC system can properly determine what are value arguments to be sent, what are reference arguments to receive answers, and how to signal errors.

RPC 系统一般来说是对远程的函数调用的一些限定。这也就是为什么 RPC 系统可以恰当地决定哪些参数要被传递，哪些引用参数来接受数据，以及如何做错误警报。

- the function must be public (begin with a capital letter);
- have exactly two arguments, the first is a pointer to value data to be received by the function from the client, and the second is a pointer to hold the answers to be returned to the client; and
- have a return value of type os.Error

- 函数必须是公共的(也就是首字母大写)
- 有且仅有 2 个指针参数，第一个指向 “接收器”——接受从 client 端发过来的数据值，第二个指向 "发送器"——存放向 client 端发送的返回值。
- 有一个 os.Error 类型返回值

For example, a valid function is

比方说，一个合法的函数应该是如下这样的

```
F(&T1, &T2) os.Error
```

The restriction on arguments means that you typically have to define a structure type. Go's RPC uses the gob package for marshalling and unmarshalling data, so the argument types have to follow the rules of gob as discussed in an earlier chapter.

所谓的对参数的限定指的是你只需要定义数据类型。Go 的 RPC 会用 gob 包来编组和解编组数据，所以对于参数类型，你只需要按照之前讨论过的 gob 的规则定义就可以。

We shall follow the example given in the Go documentation, as this illustrates the important points. The server performs two operations which are trivial - they do not require the "grunt" of RPC, but are simple to understand. The two operations are to multiply two integers, and the second is to find the quotient and remainder after dividing the first by the second.

我们应该参考 Go 的官方文档的例子，因为这些例子展示了一些关键点。Server 端执行 2 种操作，这些操作看起来非常浅显易懂，这里没用 RPC 的那些难懂的细节，而是非常易于理解。 第一种操作是两个整数相乘，第二个则是第一个数字除以第二个数字然后求商取余。

The two values to be manipulated are given in a structure:

2 个操作数被放在了一个结构体里：

```
type Values struct {
    X, Y int
}
```

The sum is just an int, while the quotient/remainder is another structure

两数之和是一个 int，而商数和余数则在另一个结构体里

```
type Quotient struct {
    Quo, Rem int
}
```

We will have two functions, multiply and divide to be callable on the RPC server. These functions will need to be registered with the RPC system. The function Register takes a single parameter, which is an interface. So we need a type with these two functions:

我们会把这两个程序，也就是乘法和除法，部署在 RPC 的 server 端等待调用。这些函数过会儿会被注册到 RPC 系统里去。函数 Register 会带一个 interface 类型的参数。 所以我们要给这两个函数定义一个类型。

```
type Arith int

func (t *Arith) Multiply(args *Args, reply *int) os.Error {
        *reply = args.A * args.B
        return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) os.Error {
        if args.B == 0 {
                return os.ErrorString("divide by zero")
        }
        quo.Quo = args.A / args.B
        quo.Rem = args.A % args.B
        return nil
}
```

The underlying type of Arith is given as int. That doesn't matter - any type could have done.

Arith 背后的实际类型是 int. 这不要紧 - 任何类型都可以。

An object of this type can now be registered using Register, and then its methods can be called by the RPC system.

这个类型的对象现在可以用 Register 函数来注册，之后，RPC 系统就可以调用这个方法了。

HTTP RPC Server

HTTP RPC 服务器

Any RPC needs a transport mechanism to get messages across the network. Go can use HTTP or TCP. The advantage of the HTTP mechanism is that it can leverage off the HTTP suport library. You need to add an RPC handler to the HTTP layer which is done using HandleHTTP and then start an HTTP server. The complete code is

任何 RPC 系统都需要一个传输机制来跨网络地传递消息。Go 可以用 HTTP 或 TCP。用 HTTP 机制的优势就是可以借助 HTTP 来支持库文件。 你需要通过 HandleHTTP 在 HTTP 层上加一个 RPC 处理器， 然后启动一个 HTTP 服务器。完整的代码是这样

```
/**
* ArithServer
 */

package main

import (
        "fmt"
        "net/rpc"
        "errors"
        "net/http"
)

type Args struct {
        A, B int
}
```

```
type Quotient struct {
	Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
	*reply = args.A * args.B
	return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
	if args.B == 0 {
		return errors.New("divide by zero")
	}
	quo.Quo = args.A / args.B
	quo.Rem = args.A % args.B
	return nil
}

func main() {

	arith := new(Arith)
	rpc.Register(arith)
	rpc.HandleHTTP()

	err := http.ListenAndServe(":1234", nil)
	if err != nil {
```

```
                fmt.Println(err.Error())
        }
}
```

HTTP RPC client

HTTP RPC 客户端

The client needs to set up an HTTP connection to the RPC server. It needs to prepare a structure with the values to be sent, and the address of a variable to store the results in. Then it can make a Call with arguments:

客户端需要设置一个 HTTP 连接，来连接 RPC 服务器。客户端需要发起一个对 RPC 服务器的连接。它需要准备一个包含要发送数据的结构体，以及一个接受返回值的变量地址。之后，它就可以用参数来 调用了，参数如下

- The name of the remote function to execute
- The values to be sent
- The address of a variable to store the result in

- 想要调用的远程函数的名字
- 被发送的数据结构体
- 储存返回值的变量地址

A client that calls both functions of the arithmetic server is

一个调用在远端服务器上的这两个计算函数的客户端是这样的

```
/**
* ArithClient
 */
```

```
package main


import (
	"net/rpc"
	"fmt"
	"log"
	"os"
)


type Args struct {
	A, B int
}


type Quotient struct {
	Quo, Rem int
}


func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "server")
		os.Exit(1)
	}
	serverAddress := os.Args[1]


	client, err := rpc.DialHTTP("tcp", serverAddress+":1234")
	if err != nil {
		log.Fatal("dialing:", err)
	}
	// Synchronous call
```

```
 args := Args{17, 8}
            var reply int
            err = client.Call("Arith.Multiply", args, &reply)
            if err != nil {
                        log.Fatal("arith error:", err)
            }
            fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

            var quot Quotient
            err = client.Call("Arith.Divide", args, '')
            if err != nil {
                        log.Fatal("arith error:", err)
            }
            fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

TCP RPC server

TCP RPC 服务端

A version of the server that uses TCP sockets is

一个使用 TCP socket 的服务器是这样的

```
/**
* TCPArithServer
 */

package main
```

```
import (
	"fmt"
	"net/rpc"
	"errors"
	"net"
	"os"
)

type Args struct {
	A, B int
}

type Quotient struct {
	Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
	*reply = args.A * args.B
	return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
	if args.B == 0 {
		return errors.New("divide by zero")
	}
	quo.Quo = args.A / args.B
	quo.Rem = args.A % args.B
```

```
        return nil
}

func main() {

        arith := new(Arith)
        rpc.Register(arith)

        tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        /* This works:
        rpc.Accept(listener)
        */
 /* and so does this:
          */
 for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
                }
                rpc.ServeConn(conn)
        }

}
```

```
func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Note that the call to Accept is blocking, and just handles client connections. If the server wishes to do other work as well, it should call this in a goroutine.

留心一点，对于 Accept 的调用是阻塞式的，用来处理客户端连接。如果服务端希望也做点别的事情，那么就应该在 goroutine 中调用它。

TCP RPC client

TCP RPC 客户端

A client that uses the TCP server and calls both functions of the arithmetic server is

一个使用 TCP 连接，调用在远端计算服务器的两个函数的客户端是这样的。

```
/**
* TCPArithClient
 */

package main

import (
        "net/rpc"
        "fmt"
        "log"
```

```
        "os"
)

type Args struct {
        A, B int
}

type Quotient struct {
        Quo, Rem int
}

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "server:port")
                os.Exit(1)
        }
        service := os.Args[1]

        client, err := rpc.Dial("tcp", service)
        if err != nil {
                log.Fatal("dialing:", err)
        }
        // Synchronous call
 args := Args{17, 8}
        var reply int
        err = client.Call("Arith.Multiply", args, &reply)
        if err != nil {
                log.Fatal("arith error:", err)
        }
```

```
        fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)


        var quot Quotient
        err = client.Call("Arith.Divide", args, ")
        if err != nil {
                log.Fatal("arith error:", err)
        }
        fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

Matching values

数据值匹配

We note that the types of the value arguments are not the same on the client and server. In the server, we have used Values while in the client we used Args. That doesn't matter, as we are following the rules of gob serialisation, and the names an types of the two structures' fields match. Better programming practise would say that the names should be the same!*

我们注意到在 server 端和 client 端的数据类型并不相同。在服务器端，我们用的是 Values 而在客户端我们用了 Args。这并不成问题，因为我们按照了 gob 串行化规则，而且在两个结构体字段中的名称能匹配。但更好的编程实践却告诉我们，名字也应该相同。*

However, this does point out a possible trap in using Go RPC. If we change the structure in the client to be, say,

然而，这指出了 go 中可能存在的陷阱的可能性。要是我们改变了 client 端的结构体，比方说，

```
type Values struct {
```

```
        C, B int
}
```

then gob has no problems: on the server-side the unmarshalling will ignore the value of C given by the client, and use the default zero value for A.

而这对于 gob 来说却没有什么疑问: 在 server 端解编组的时候会忽略来自 client 的 C,然后将默认值零值赋给 A.

Using Go RPC will require a rigid enforcement of the stability of field names and types by the programmer. We note that there is no version control mechanism to do this, and no mechanism in gob to signal any possible mismatches.

用 Go RPC 会要求对字段名称和类型的一致性都进行严格加强。我们注意到，没有任何的版本控制机制或是 gob 本身，都没有任何提示数据不匹配的保护机制。

JSON

JSON

This section adds nothing new to the earlier concepts. It just uses a different "wire" format for the data, JSON instead of gob. As such, clients or servers could be written in other language that understand sockets and JSON.

这部分每增加什么新的概念。只是用了另一种数据的 "电报" 格式， 用 JSON 来代替 gob。由于这样做了，那么 client 端和 server 端要用另一种语言来来理解 socket 和 JSON。

JSON RPC client

JSON RPC 客户端

A client that calls both functions of the arithmetic server is

客户端调用计算服务器的两个函数如下

```
/* JSONArithCLient
 */

package main

import (
	"net/rpc/jsonrpc"
	"fmt"
	"log"
	"os"
)

type Args struct {
	A, B int
}

type Quotient struct {
	Quo, Rem int
}

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "server:port")
		log.Fatal(1)
	}
	service := os.Args[1]
```

```
        client, err := jsonrpc.Dial("tcp", service)
        if err != nil {
                log.Fatal("dialing:", err)
        }
        // Synchronous call
 args := Args{17, 8}
        var reply int
        err = client.Call("Arith.Multiply", args, &reply)
        if err != nil {
                log.Fatal("arith error:", err)
        }
        fmt.Printf("Arith: %d*%d=%d\n", args.A, args.B, reply)

        var quot Quotient
        err = client.Call("Arith.Divide", args, '')
        if err != nil {
                log.Fatal("arith error:", err)
        }
        fmt.Printf("Arith: %d/%d=%d remainder %d\n", args.A, args.B, quot.Quo, quot.Rem)

}
```

JSON RPC server

JSON RPC 服务器

A version of the server that uses JSON encoding is

JSON 版的服务器代码如下

```
/* JSONArithServer
```

```
 */

package main

import (
	"fmt"
	"net/rpc"
	"net/rpc/jsonrpc"
	"os"
	"net"
	"errors"
)
//import ("fmt"; "rpc"; "os"; "net"; "log"; "http")

type Args struct {
	A, B int
}

type Quotient struct {
	Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
	*reply = args.A * args.B
	return nil
}
```

```
func (t *Arith) Divide(args *Args, quo *Quotient) error {
        if args.B == 0 {
                return errors.New("divide by zero")
        }
        quo.Quo = args.A / args.B
        quo.Rem = args.A % args.B
        return nil
}

func main() {

        arith := new(Arith)
        rpc.Register(arith)

        tcpAddr, err := net.ResolveTCPAddr("tcp", ":1234")
        checkError(err)

        listener, err := net.ListenTCP("tcp", tcpAddr)
        checkError(err)

        /* This works:
        rpc.Accept(listener)
        */
 /* and so does this:
          */
 for {
                conn, err := listener.Accept()
                if err != nil {
                        continue
```

```
                }
                jsonrpc.ServeConn(conn)
        }

}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Conclusion

总结

RPC is a popular means of distributing applications. Several ways of doing it have been presented here. What is missing from Go is support for the currently fashionable (but extremely badly enginereed) SOAP RPC mechanism.

RPC 是一个流行的分布应用的方法。这里展示了许多实现它的方法。Go 所不支持的实现下很火的(却也是实现地很不好的) SOAP RPC 机制。

Network channels

网络 channels

Warning

警告

The netchan package is being reworked. While it was in earlier versions of Go, it is not in Go 1. It is available in the **old/netchan** package if you still need it. This chapter describes this old version. Do not use it for new code.

现在 netchan 包正在重新设计。出于对 Go 1 之前版本的兼容性考虑，可以在 **old/netchan** 下找到它。这一章描述的是旧版本的使用。请不要在新代码中使用它.

Introduction

简介

There are many models for sharing information between communicating processes. One of the more elegant is Hoare's concept of *channels*. In this, there is no shared memory, so that none of the issues of accessing common memory arise. Instead, one process will send a message along a channel to another process. Channels may be synchronous, or asynchronous, buffered or unbuffered.

关于进程间共享信息有过许多模型。其中较为优美的是 Hoare 提出的 *channels* 模型。在这一模型中，不需要共享内存，因此读取共享内存引起的问题都可以避免。取而代之的是使用 channel 传递消息：一个进程通过一个 channel 向另一个进程发送消息，channels 可以是同步的，也可以是异步的，可以是带缓冲的，也可以是不带缓冲的。

Go has channels as first order data types in the language. The canonical example of using channels is Erastophene's prime sieve: one goroutine generates integers from 2 upwards. These are pumped into a series of channels that act as sieves. Each filter is distinguished by a different prime, and it removes from its stream each number that is divisible by its prime. So the '2' goroutine filters out even

numbers, while the '3' goroutine filters out multiples of 3. The first number that comes out of the current set of filters must be a new prime, and this is used to start a new filter with a new channel.

Go 内建 channel 作为第一等数据类型。一个使用 channel 的经典例子是 Erastophene 的素数筛选器：使用一个 goroutine 从 2 开始生成整数，将这些数字送入一系列作为过滤器的 channel，每一个过滤器由一个不同的素数标识，它们把能被自身代表素数整除的数从流中删除，所以“2”goroutine 过滤掉所有偶数，“3”goroutine 过滤掉所有 3 的倍数。第一个从这一系列过滤器中走出来的必然是一个新的素数，然后再开启新的 channel，用新素数生成一个新的过滤器，循环往复。

The efficacy of many thousands of goroutines communicating by many thousands of channels depends on how well the implementation of these primitives is done. Go is designed to optimise these, so this type of program is feasible.

大量 goroutine 之间通过 channel 通信的效率取决于原语设计的好坏。Go 天生为此优化，所以这种程序是可行的

Go also supports distributed channels using the **netchan** package. But network communications are thousands of times slower than channel communications on a single computer. Running a sieve on a network over TCP would be ludicrously slow. Nevertheless, it gives a programming option that may be useful in many situations.

Go 也通过 **netchan** 包支持分布式 channel。但是网络间 channel 通信的效率远比单一电脑上 channel 间通信的效率低。在网络上通过 TCP 协议运行一个筛选器更是慢的可怕。然而，这还是给程序员多了一个选择，而且这在某些情况下十分有用。

Go's network channel model is somewhat similar in concept to the RPC model: a server creates channels and registers them with the network channel API. A client does a lookup for channels on a server. At this point both sides have a shared channel over which they can communicate. Note that communication is one-way: if you want to send information both ways, open two channels one for each direction.

Go 的网络 channel 模型某种意义上和 RPC 模型类似：服务器创建 channel 然后用网络 channel API 注册它们，客户端在服务器上查询 channel。这样服务器和客户端就有了一个可以相互通信的共享 channel。注意这种通信是单向的，如果你想要双向发送信息，为每个方向单独创建一个 channel。

Channel server

服务器端 Channel

In order to make a channel visible to clients, you need to *export* it. This is done by creating an exporter using **NewExporter** with no parameters. The server then calls **ListenAndServe** to lsiten and handle responses. This takes two parameters, the first being the underlying transport mechanism such as "tcp" and the second being the network listening address (usually just a port number.

要让一个 channel 对客户端可见，你需要*导出*它。这可以通过不带参数 **NewExporter** 创建一个新的导出，之后服务器调用 **ListenAndServe** 监听、处理请求。**ListenAndServe** 带有两个参数，第一个是底层的传输机制，比如“tcp”；第二个是监听地址（通常只是一个端口号）。

For each channel, the server creates a normal local channel and then calls **Export** to bind this to the network channel. At the time of export, the direction of communication must be specified. Clients search for channels by name, which is a string. This is specified to the exporter.

对每个 channel，服务器创建一个普通的本地 channel，然后调用 **Export** 将它绑定到网络 channel 上。在导出的同时，必须指定通信的方向。客户端可以通过名字（一个字符串）搜寻 channel。

The server then uses the local channels in the normal way, reading or writing on them. We illustrate with an "echo" server which reads lines and sends them back. It needs two channels for this. The channel that the client writes to we name "echo-out". On the server side this is a read channel. Similarly, the channel that the client reads from we call "echo-in", which is a write channel to the server.

服务器之后可以像使用本地 channel 一样，读取或是写入数据。我们以一个“echo”服务器为例，它读入文本，再原样发送回去。它需要两个 channel，我们把客户端用来写入数据的

channel 叫做“echo-out”，在服务器端，用它来读取数据。相似的，客户端用来读取数据的 channel 叫做“echo-in”，服务器往里写入数据。

The server program is

服务器程序如下

```
/* EchoServer
 */
package main

import (
        "fmt"
        "os"
        "old/netchan"
)

func main() {

        // exporter, err := netchan.NewExporter("tcp", ":2345")
 exporter := netchan.NewExporter()
        err := exporter.ListenAndServe("tcp", ":2345")
        checkError(err)

        echoIn := make(chan string)
        echoOut := make(chan string)
        exporter.Export("echo-in", echoIn, netchan.Send)
        exporter.Export("echo-out", echoOut, netchan.Recv)
        for {
                fmt.Println("Getting from echoOut")
```

```
                s, ok := <-echoOut
                if !ok {
                        fmt.Printf("Read from channel failed")
                        os.Exit(1)
                }
                fmt.Println("received", s)

                fmt.Println("Sending back to echoIn")
                echoIn <- s
                fmt.Println("Sent to echoIn")
        }

}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Note: at the time of writing, the server will sometimes fail with an error message "netchan export: error encoding client response". This is logged as Issue 1805

注意：在这篇教程写下的时候，服务器可能会收到“netchan export: error encoding client response”的错误消息，这个问题被登记为 Issue 1805。

Channel client

客户端 Channel

In order to find an exported channel, the client must *import* it. This is created using **Import** which takes a protocol and a network service address of "host:port". This is then used to import a network channel by name and bind it to a local channel. Note that channel variables are *references*, so you do not need to pass their addresses to functions that change them.

为了找到导出的 channel，客户端必须*导入*它。使用 **Import** 方法可以完成，它接受两个参数：协议名和形如“host:port”网络服务地址。之后就能通过名字导入，并绑定到本地 channel 上。注意，channel 变量是*引用变量*，不用向改变他们的函数传递 channel 的地址。

The following client gets two channels to and from the echo server, and then writes and reads ten messages:

以下客户端使用两个 channel 向/从服务器发送、接受消息，然后写入并读取收到的十条信息。

```
/* EchoClient
 */
package main

import (
        "fmt"
        "old/netchan"
        "os"
)

func main() {
        if len(os.Args) != 2 {
                fmt.Println("Usage: ", os.Args[0], "host:port")
                os.Exit(1)
        }
        service := os.Args[1]
```

```
	importer, err := netchan.Import("tcp", service)
	checkError(err)

	fmt.Println("Got importer")
	echoIn := make(chan string)
	importer.Import("echo-in", echoIn, netchan.Recv, 1)
	fmt.Println("Imported in")

	echoOut := make(chan string)
	importer.Import("echo-out", echoOut, netchan.Send, 1)
	fmt.Println("Imported out")

	for n := 0; n < 10; n++ {
		echoOut <- "hello "
		s, ok := <-echoIn
		if !ok {
			fmt.Println("Read failure")
			break
		}
		fmt.Println(s, n)
	}
	close(echoOut)
	os.Exit(0)
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
```

```
                os.Exit(1)
        }
}
```

Handling Timeouts

处理超时

Because these channels use the network, there is alwasy the possibility of network errors leading to timeouts. Andrew Gerrand points out a solution using timeouts: "[Set up a timeout channel.] We can then use a select statement to receive from either ch or timeout. If nothing arrives on ch after one second, the timeout case is selected and the attempt to read from ch is abandoned."

因为 channel 使用网络通信，存在因为网络错误导致超时的可能性。Andrew Gerrand 提出了一个办法 timeouts: "[Set up a timeout channel.]我们可以使用 select 语句从 ch 或是 timeout 接受信息。如果超过 1 秒钟 ch 没有收到信息，timeout 通道被选择，放弃从 ch 获取信息。

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1e9) // one second
    timeout <- true
}()

select {
case <- ch:
    // a read from ch has occurred
case <- timeout:
    // the read from ch has timed out
}
```

Channels of channels

传递 channel 的 channel

The online Go tutorial at http://golang.org/doc/go_tutorial.html has an example of multiplexing, where channels of channels are used. The idea is that instread of sharing one channel, a new communicator is given their own channel to have a privagye conversation. That is, a client is sent a channel from a server through a shared channel, and uses that private channel.

在线 Go 指导（http://golang.org/doc/go_tutorial.html）中展示了一个有点复杂的例子，其中使用了传递 channel 的 channel。这个方法避免了总是使用共享 channel，新的进程被赋予他们自己的 channel 进行私有交流。即，客户端通过共享 channel 从服务器端获得一个 channel，之后使用这个私有 channel 进行通信。

This doesn't work directly with network channels: a channel cannot be sent over a network channel. So we have to be a little more indirect. Each time a client connects to a server, the server builds new network channels and exports them with new names. Then it sends the *names* of these new channels to the client which imports them. It uses these new channels for communicaiton.

然而这对网络 channel 不起作用，网络 channel 不能发送 channel，所以我们要稍微绕点弯路。每次客户端连接服务器，服务器建立一个新的 channel，然后用新名字导出他们。之后向导入它们的客户端发送这些新 channel 的名字。最后使用这些新 channel 进行通信。

A server is

服务器代码如下

```
/* EchoChanServer
 */
package main

import (
```

```
	"fmt"
	"os"
	"old/netchan"
	"strconv"
)

var count int = 0

func main() {

	exporter := netchan.NewExporter()
	err := exporter.ListenAndServe("tcp", ":2345")
	checkError(err)

	echo := make(chan string)
	exporter.Export("echo", echo, netchan.Send)
	for {
		sCount := strconv.Itoa(count)
		lock := make(chan string)
		go handleSession(exporter, sCount, lock)

		<-lock
		echo <- sCount
		count++
		exporter.Drain(-1)
	}
}

func handleSession(exporter *netchan.Exporter, sCount string, lock chan string) {
```

```
	echoIn := make(chan string)
	exporter.Export("echoIn"+sCount, echoIn, netchan.Send)

	echoOut := make(chan string)
	exporter.Export("echoOut"+sCount, echoOut, netchan.Recv)
	fmt.Println("made " + "echoOut" + sCount)

	lock <- "done"

	for {
		s := <-echoOut
		echoIn <- s
	}
	// should unexport net channels
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

and a client is

客户端代码如下

```
/* EchoChanClient
 */
package main
```

```
import (
	"fmt"
	"old/netchan"
	"os"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "host:port")
		os.Exit(1)
	}
	service := os.Args[1]

	importer, err := netchan.Import("tcp", service)
	checkError(err)

	fmt.Println("Got importer")
	echo := make(chan string)
	importer.Import("echo", echo, netchan.Recv, 1)
	fmt.Println("Imported in")

	count := <-echo
	fmt.Println(count)

	echoIn := make(chan string)
	importer.Import("echoIn"+count, echoIn, netchan.Recv, 1)

	echoOut := make(chan string)
```

```
        importer.Import("echoOut"+count, echoOut, netchan.Send, 1)


        for n := 1; n < 10; n++ {
                echoOut <- "hello "
                s := <-echoIn
                fmt.Println(s, n)
        }
        close(echoOut)
        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Conclusion

总结

Network channels are a distributed analogue of local channels. They behave approximately the same, but due to limitations of the model some things have to be done a little differently.

网络 channel 是本地 channel 的分布式模拟。它们表现的近乎相同，但是由于模型的限制，存在一些不同。

Web sockets

Web sockets

Web sockets are designed to answer a common problem with web systems: the server is unable to initiate or push content to a user agent such as a browser. Web sockets allow a full duplex connection to be established to allow this. Go has nearly complete support for them.

Web sockets 规范的设计是为了解决网络系统中的一个常见问题：服务器端无法发起或推送内容到用户代理，例如浏览器。Web sockets 能够建立一个全双工的连接来进行这些操作。Go 语言对此有近乎完整的支持。

Warning

警告

The Web Sockets package is not currently in the main Go 1 tree and is not included in the current distributions. To use it, you need to install it by

Web Sockets 包当前并不在 Go 1 的主代码树里，也不包含在当前的分发包里。为了使用它，你必须先通过如下命令来安装它

```
go get code.google.com/p/go.net/websocket
```

Introduction

介绍

The websockets model will change for release r61. This describes the new package, not the package in r60 and earlier. If you do not have r61, at the time of writing, use **hg pull; hg update weekly** to download it.

websockets 模式(模型？？)会在 r61 版本里做些变更。此文介绍的是新的包，而不是 r60 或者更早之前的版本。如果你当前还没有 r61 版本，使用 **hg pull; hg update weekly** 来下载它。

The standard model of interaction between a web user agent such as a browser and a web server such as Apache is that the user agent makes HTTP requests and the server makes a single reply to each one. In the case of a browser, the request is made by clicking on a link, entering a URL into the address bar, clicking on the forward or back buttons, etc. The response is treated as a new page and is loaded into a browser window.

用户代理（例如浏览器）和 web 服务器（例如 Apache）之间进行交互的标准模型是这样的：用户代理发送 HTTP 请求，然后服务器响应每个请求。以浏览器举例，请求是指点击链接、在地址栏中输入网址、点击前进或后退按钮等行为。而响应则是在浏览器窗口里加载的页面。

This traditional model has many drawbacks. The first is that each request opens and closes a new TCP connection. HTTP 1.1 solved this by allowing persistent connections, so that a connection could be held open for a short period to allow for multiple requests (e.g. for images) to be made on the same server.

这种传统模型有很多缺点。首先，每个请求都会开启和关闭一个新的 TCP 连接。HTTP 1.1 通过持久化连接来解决这个问题，一个连接能够在较短的时期里保持打开状态，从而达到向同一个服务器发送多个请求(例如加载图片)的目的。

While HTTP 1.1 persistent connections alleviate the problem of slow loading of a page with many graphics, it does not improve the interaction model. Even with forms, the model is still that of submitting the form and displaying the response as a new page. JavaScript helps in allowing error checking to be performed on form data before submission, but does not change the model.

虽然 HTTP 1.1 的持久化连接减轻了有很多图片的页面的加载缓慢问题，但它还是没有改进旧的交互模型。特别是在有表单的情况，模型依然是提交表单然后展现一个新页面作为响应。虽然在 JavaScript 的帮助下能做到在提交表单前进行错误检查，但依然没有改变这种模型。

AJAX (Asynchronous JavaScript and XML) made a significant advance to the user interaction model. This allows a browser to make a request and just use the response to update the display in place using the HTML Document Object Model (DOM). But again the interaction model is the same. AJAX just affects how the browser manages the returned pages. There is no explicit extra support in Go for AJAX, as none is needed: the HTTP server just sees an ordinary HTTP POST request with possibly some XML or JSON data, and this can be dealt with using techniques already discussed.

AJAX(异步 JavaScript 和 XML)对交互模型做了极大的改进。它允许浏览器发送请求然后通过 DOM 将响应内容更新到页面上适当的部分。但是交互模型的本质还是没有变。AJAX 只是影响了浏览器对返回页面的处理方式。Go 没有为 AJAX 做明显的额外支持，其实也没有这个必要:HTTP 服务器看到的仍然是常规的 HTTP POST 请求(可能带有一些 XML 或者 JSON 数据)，这种请求可以被已经谈及的技术进行处理。

All of these are still browser to server communication. What is missing is server initiated communications to the browser. This can be filled by Web sockets: the browser (or any user agent) keeps open a long-lived TCP connection to a Web sockets server. The TCP connection allows either side to send arbitrary packets, so any application protocol can be used on a web socket.

前面提及的都还是浏览器向服务器通讯。所缺少的是服务器向浏览器发起通讯。Web sockets 正好可以填补这个空缺:浏览器(或者任何其它的用户代理)保持开启一条和 Web sockets 服务器的 TCP 长连接。这条 TCP 连接允许各边发送任意的数据包，因此可以在 web socket 上使用任何的应用层协议。

How a websocket is started is by the user agent sending a special HTTP request that says "switch to web sockets". The TCP connection underlying the HTTP request is kept open, but both user agent and server switch to using the web sockets protocol instead of getting an HTTP response and closing the socket.

websocket 是由用户代理发送一条“切换到 web sockets”特殊 HTTP 请求开始的。HTTP 请求所使用的 TCP 连接仍然保持开启状态，而不是在获取到一个 HTTP 响应后关闭，同时用户代理和服务器端则切换到使用 web sockets 协议。

Note that it is still the browser or user agent that initiates the Web socket connection. The browser does not run a TCP server of its own. While the specification is complex, the protocol is designed to be fairly easy to use. The client opens an HTTP connection and then replaces the HTTP protocol with its own WS protocol, re-using the same TCP connection.

值得注意的是，仍然是由浏览器或者用户代理来发起一条 Web socket 连接的。浏览器自身并没有运行一个 TCP 服务器？？虽然规范很复杂,但协议还是设计得相当易用的。客户端开启一条 HTTP 连接，接着用 WS 协议取代 HTTP 协议，重用了同一条 TCP 连接。

Web socket server

Web socket 服务器端

A web socket server starts off by being an HTTP server, accepting TCP conections and handling the HTTP requests on the TCP connection. When a request comes in that switches that connection to a being a web socket connection, the protocol handler is changed from an HTTP handler to a WebSocket handler. So it is only that TCP connection that gets its role changed: the server continues to be an HTTP server for other requests, while the TCP socket underlying that one connection is used as a web socket.

web socket 服务器端最初是 HTTP 服务器端，接受 TCP 连接，处理该连接上的 HTTP 请求。当将该连接变换成 web socket 连接的请求到来之后，协议处理器从一个 HTTP 处理器转变成 WebSocket 处理器。所以仅仅是 TCP 连接的角色变化了：当前连接所有的 TCP socket 被当成 web socket 来使用;对于其它请求而言，服务器仍然是一个 HTTP 服务器。

One of the simple servers HHTP we discussed in Chapter 8: HTTP registered varous handlers such as a file handler or a function handler. To handle web socket requests we simply register a different type of handler - a web socket handler. Which handler the server uses is based on the URL pattern. For example, a file handler might be registered for "/", a function handler for "/cgi-bin/..." and a web sockets handler for "/ws".

在章节 8: HTTP 里我们讨论的一个简单的服务器注册了各式各样的处理器，例如文件处理器、函数处理器等。为了处理 web socket 请求，我们仅需另外注册一种类型的处理器-web socket 处理器。服务器基于 URL 模式来选出对应处理器。例如，"/"为文件处理器，"/cgi-bin/..."为函数处理器，"/ws"为 web sockets 处理器。

An HTTP server that is only expecting to be used for web sockets might run by

如下将运行一个仅用于 web sockets 的 HTTP 服务器

```
func main() {
        http.Handle("/", websocket.Handler(WSHandler))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}
```

A more complex server might handle both HTTP and web socket requests simply by adding in more handlers.

通过添加更多的处理器，一个更为复杂的服务器可以同时处理 HTTP 请求和 web socket 请求。

The Message object

Message 对象

HTTP is a stream protocol. Web sockets are frame-based. You prepare a block of data (of any size) and send it as a set of frames. Frames can contain either strings in UTF-8 encoding or a sequence of bytes.

HTTP 是流协议。Web sockets 是基于帧的。你可以生成任意大小的一块数据，将其作为一组帧来发送。帧可以包含 UTF-8 编码的字符串或者字节序列。

The simplest way of using web sockets is just to prepare a block of data and ask the Go websocket library to package it as a set of frame data, send them across the wire and receive it as the same block. The **websocket** package contains a convenience object **Message** to do just that. The **Message** object has two methods, **Send** and **Receive** which take a websocket as first parameter. The second parameter is either the address of a variable to store data in, or the data to be sent. Code to send string data would look like

最简单的使用 web sockets 的方法就是准备好数据块然后让 Go 的 websocket 库将其封装成一组帧数据，通过网络线路发送，然后接收生成同样的数据块。**websocket** 包里有个很好使的 **Message** 对象来做这些。**Message** 对象有 **Send** 和 **Receive** 两个方法，它们的第一个参数是一个 websocket 对象，第二个参数是存放数据的地址。发送字符串数据的代码示例如下

```
msgToSend := "Hello"
err := websocket.Message.Send(ws, msgToSend)

var msgToReceive string
err := websocket.Message.Receive(conn, &msgToReceive)
```

Code to send byte data would look like

发送字节序列的代码示例如下

```
dataToSend := []byte{0, 1, 2}
err := websocket.Message.Send(ws, dataToSend)

var dataToReceive []byte
err := websocket.Message.Receive(conn, &dataToReceive)
```

An echo server to send and receive string data is given below. Note that in web sockets either side can initiate sending of messages, and in this server we send messages from the server to a client when it connects (send/receive) instead of the more normal receive/send server. The server is

下面将给出一个发送和接收字符串数据的 echo 服务器代码。值得注意的是，在 web sockets 协议里，各边都可以发起消息的发送。这回，当客户端连接后，服务器端先向客户端发送消息(发送/接收)，而不是传统的接收/发送。服务器端代码如下

```
/* EchoServer
 */
package main

import (
        "fmt"
        "net/http"
        "os"
        // "io"
 "code.google.com/p/go.net/websocket"
)

func Echo(ws *websocket.Conn) {
        fmt.Println("Echoing")

        for n := 0; n < 10; n++ {
                msg := "Hello    " + string(n+48)
                fmt.Println("Sending to client: " + msg)
                err := websocket.Message.Send(ws, msg)
                if err != nil {
                        fmt.Println("Can't send")
```

```
                        break
                }

                var reply string
                err = websocket.Message.Receive(ws, &reply)
                if err != nil {
                        fmt.Println("Can't receive")
                        break
                }
                fmt.Println("Received back from client: " + reply)
        }
}

func main() {

        http.Handle("/", websocket.Handler(Echo))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

A client that talks to this server is

和服务器端进行会话的客户端代码如下

```
/* EchoClient
 */
package main

import (
	"code.google.com/p/go.net/websocket"
	"fmt"
	"io"
	"os"
)

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "ws://host:port")
		os.Exit(1)
	}
	service := os.Args[1]

	conn, err := websocket.Dial(service, "", "http://localhost")
	checkError(err)
	var msg string
	for {
		err := websocket.Message.Receive(conn, &msg)
		if err != nil {
			if err == io.EOF {
				// graceful shutdown by server
			  break
			}
```

```
                        fmt.Println("Couldn't receive msg " + err.Error())
                        break
                }
                fmt.Println("Received from server: " + msg)
                // return the msg
            err = websocket.Message.Send(conn, msg)
                if err != nil {
                        fmt.Println("Coduln't return msg")
                        break
                }
        }
        os.Exit(0)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The url for the client running on the same machine as the server should be **ws://localhost:12345/**

当客户端和服务器端运行在同一台机器上时，客户端所需的 url 参数是 **ws://localhost:12345/**。

The JSON object

JSON 对象

It is expected that many websocket clients and servers will exchange data in JSON format. For Go programs this means that a Go object will be marshalled into JSON format as described in Chapter 4:

Serialisation and then sent as a UTF-8 string, while the receiver will read this string and unmarshal it back into a Go object.

正如所预期的那样，许多 websocket 客户端和服务器端希望通过 JSON 格式来进行数据交换。在 Go 程序里，这意味着 Go 中的对象将编组成 JSON 格式（在章节 4: Serialisation 里有描述）然后以 UTF-8 字符串发送；而接收方将读取该字符串然后将其解组成 Go 对象。

The **websocket** convenience object **JSON** will do this for you. It has methods **Send** and **Receive** for sending and receiving data, just like the **Message** object.

websocket 里的 JSON 对象将很方便地为你做这些。它带有 **Send** 和 **Receive** 两个方法来发送接收数据，正如 **Message** 对象。

A client that sends a **Person** object in JSON format is

下面这个客户端将以 JSON 格式来发送 **Person** 对象

```
/* PersonClientJSON
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "os"
)

type Person struct {
        Name    string
        Emails []string
}
```

```
func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "ws://host:port")
		os.Exit(1)
	}
	service := os.Args[1]

	conn, err := websocket.Dial(service, "",
		"http://localhost")
	checkError(err)

	person := Person{Name: "Jan",
		Emails: []string{"ja@newmarch.name", "jan.newmarch@gmail.com"},
	}

	err = websocket.JSON.Send(conn, person)
	if err != nil {
		fmt.Println("Couldn't send msg " + err.Error())
	}
	os.Exit(0)
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

and a server that reads it is

读取该数据的服务器端

```
/* PersonServerJSON
 */
package main

import (
	"code.google.com/p/go.net/websocket"
	"fmt"
	"net/http"
	"os"
)

type Person struct {
	Name	string
	Emails []string
}

func ReceivePerson(ws *websocket.Conn) {
	var person Person
	err := websocket.JSON.Receive(ws, &person)
	if err != nil {
		fmt.Println("Can't receive")
	} else {

		fmt.Println("Name: " + person.Name)
		for _, e := range person.Emails {
```

```
                                fmt.Println("An email: " + e)
                        }
                }
}

func main() {

        http.Handle("/", websocket.Handler(ReceivePerson))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The Codec type

Codec 类型

The **Message** and **JSON** objects are both instances of the type **Codec**. This type is defined by

Message 对象和 **JSON** 对象都是 类型 **Codec** 的实例。该类型的定义如下

```
type Codec struct {
        Marshal     func(v interface{}) (data []byte, payloadType byte, err os.Error)
        Unmarshal func(data []byte, payloadType byte, v interface{}) (err os.Error)
}
```

The type **Codec** implements the **Send** and **Receive** methods used earlier.

Codec 类型实现了前面提及的 **Send** 和 **Receive** 方法。

It is likely that websockets will also be used to exchange XML data. We can build an XML **Codec** object by wrapping the XML marshal and unmarshal methods discussed in Chapter 12: XML to give a suitable **Codec** object.

websocket 同样可能被用来交换 XML 数据。我们可以通过包含章节 12: XML 里提及的 XML marshal 和 unmarshal 函数来创建一个合适的 **Codec** 对象。

We can create a **XMLCodec** package in this way:

我们将以这种方式来创建 **XMLCodec** 包

```
package xmlcodec

import (
        "encoding/xml"
        "code.google.com/p/go.net/websocket"
)

func xmlMarshal(v interface{}) (msg []byte, payloadType byte, err error) {
        //buff := &bytes.Buffer{}
 msg, err = xml.Marshal(v)
        //msgRet := buff.Bytes()
 return msg, websocket.TextFrame, nil
```

```
}

func xmlUnmarshal(msg []byte, payloadType byte, v interface{}) (err error) {
        // r := bytes.NewBuffer(msg)
  err = xml.Unmarshal(msg, v)
        return err
}

var XMLCodec = websocket.Codec{xmlMarshal, xmlUnmarshal}
```

We can then serialise Go objects such as a **Person** into an XML document and send it from a client to a server by

接下来我们就可以序列化像 **Person** 这样的 Go 对象到 XML 文档，然后发送给服务器端

```
/* PersonClientXML
 */
package main

import (
        "code.google.com/p/go.net/websocket"
        "fmt"
        "os"
        "xmlcodec"
)

type Person struct {
        Name     string
        Emails []string
```

```
}

func main() {
	if len(os.Args) != 2 {
		fmt.Println("Usage: ", os.Args[0], "ws://host:port")
		os.Exit(1)
	}
	service := os.Args[1]

	conn, err := websocket.Dial(service, "", "http://localhost")
	checkError(err)

	person := Person{Name: "Jan",
		Emails: []string{"ja@newmarch.name", "jan.newmarch@gmail.com"},
	}

	err = xmlcodec.XMLCodec.Send(conn, person)
	if err != nil {
		fmt.Println("Couldn't send msg " + err.Error())
	}
	os.Exit(0)
}

func checkError(err error) {
	if err != nil {
		fmt.Println("Fatal error ", err.Error())
		os.Exit(1)
	}
}
```

A server which receives this and just prints information to the console is

服务器端接收到该文档后将里面的信息打印到终端

```
/* PersonServerXML
 */
package main

import (
	"code.google.com/p/go.net/websocket"
	"fmt"
	"net/http"
	"os"
	"xmlcodec"
)

type Person struct {
	Name	string
	Emails []string
}

func ReceivePerson(ws *websocket.Conn) {
	var person Person
	err := xmlcodec.XMLCodec.Receive(ws, &person)
	if err != nil {
		fmt.Println("Can't receive")
	} else {

		fmt.Println("Name: " + person.Name)
```

```
                for _, e := range person.Emails {
                        fmt.Println("An email: " + e)
                }
        }
}

func main() {

        http.Handle("/", websocket.Handler(ReceivePerson))
        err := http.ListenAndServe(":12345", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

Web sockets over TLS

Web sockets over TLS

A web socket can be built above a secure TLS socket. We discussed in Chapter 8: HTTP how to use a TLS socket using the certificates from Chapter 7: Security. That is used unchanged for web sockets. that is, we use **http.ListenAndServeTLS** instead of **http.ListenAndServe**.

web socket 可以建立在安全的 TLS 套接字上。在章节 8: HTTP 里，我们讨论了如果通过使用章节 7: Security 里的证书来使用 TLS 套接字。这同样适用于 web socket。也就是说，我们将使用 **http.ListenAndServeTLS** 而不是 **http.ListenAndServe**。

Here is the echo server using TLS

这里是一个使用了 TLS 的 echo 服务器

```
/* EchoServer
 */
package main

import (
	"code.google.com/p/go.net/websocket"
	"fmt"
	"net/http"
	"os"
)

func Echo(ws *websocket.Conn) {
	fmt.Println("Echoing")

	for n := 0; n < 10; n++ {
		msg := "Hello    " + string(n+48)
		fmt.Println("Sending to client: " + msg)
		err := websocket.Message.Send(ws, msg)
		if err != nil {
			fmt.Println("Can't send")
			break
		}

		var reply string
		err = websocket.Message.Receive(ws, &reply)
```

```
                if err != nil {
                        fmt.Println("Can't receive")
                        break
                }
                fmt.Println("Received back from client: " + reply)
        }
}

func main() {

        http.Handle("/", websocket.Handler(Echo))
        err := http.ListenAndServeTLS(":12345", "jan.newmarch.name.pem",
                "private.pem", nil)
        checkError(err)
}

func checkError(err error) {
        if err != nil {
                fmt.Println("Fatal error ", err.Error())
                os.Exit(1)
        }
}
```

The client is the same echo client as before. All that changes is the url, which uses the "wss" scheme instead of the "ws" scheme:

客户端仍然是前面的 echo 客户端，所做的改变仅仅是将 url 里的"ws"模式替换成"wss"模式。

```
EchoClient wss://localhost:12345/
```

Conclusion

结论

The web sockets standard is nearing completion and no major changes are anticipated. This will allow HTTP user agents and servers to set up bi-directional socket connections and should make certain interaction styles much easier. Go has nearly complete support for web sockets.

web sockets 标准已接近完成，预计也不会有较大的改动。它允许 HTTP 客户代理和服务器建立一个双向的套接字连接，从而易化了某些交互方式。Go 对 web sockets 有近乎完整的支持。
