**Linux基础**

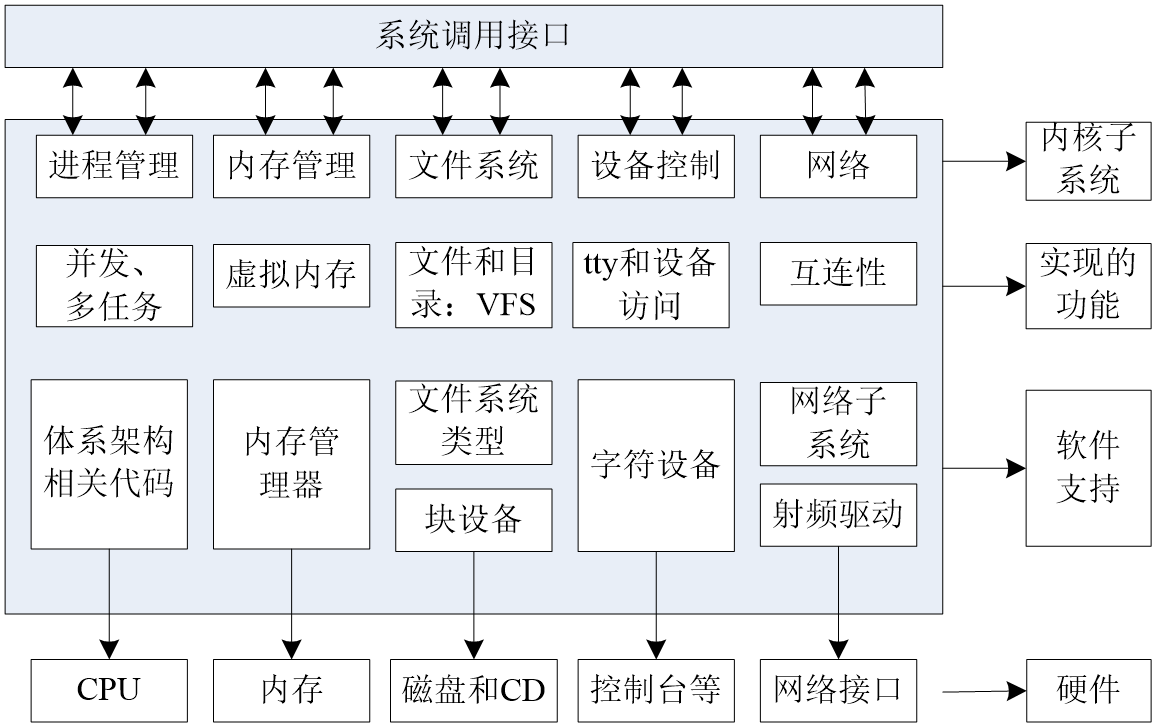
操作系统介绍

操作系统的作用

* 方便：使计算机系统易于使用
* 有效：以更有效的方式使用计算机系统资源
* 扩展：方便用户有效开发、测试和引进新功能

操作系统的地位

操作系统在计算机系统中承上启下的地位：向下封装硬件，向上提供操作接口。



Unix家族

* 1965：贝尔实验室（Bell Labs）加入一项由通用电气和麻省理工学院合作的计划，该计划要建立一套多使用者、多任务、多层次的MULTICS操作系统。后来因为项目太为复杂失败。
* 1969：其主要开发者Thompson（后被称为UNIX之父）和Ritchie领导一组开发者，开发了一个新的多任务操作系统—UNICS，后来被改名为Unix，最初的Unix是用B语言和汇编语言混合编写而成。
* 1971：两人在贝尔实验室共同发明了C语言，并于1973用C语言重写了Unix。
* 1974：UNIX第一次出现在贝尔实验室以外。此后UNIX被政府机关，研究机构，企业和大学注意到，并逐渐流行开来。
* 1980：有两个最主要的Unix的版本线，一个是Berkeley的BSD UNIX，另一个是AT&T的Unix，两者的竞争最终引发了Unix的战争，最终导致Unix出现各种各样的变种。
* 1982：AT&T基于版本7开发了UNIX System Ⅲ的第一个商业版本，并不再开源。
* 1992~2001：由于版权问题，AT&T公司与BSD开发组开始了一场将近10年的版权官司。UNIX由于其昂贵的费用，仅局限于大型机的应用；BSD因为版权问题，失去了宝贵的发展时期。

Linux家族

* Minix（mini-UNIX）最初是由Andrew Tanenbaum教授，仿照4.3BSD的源代码，白手起家完成了12000行C语言的编写工作这个系统只是一个教学工具，没有什么实际应用价值。
* 1990年，Linus Torvalds决定编写一个自己的Minix内核，初名为Linus' Minix，意为Linus的Minix内核，后来改名为Linux，此内核于1991年正式发布，并逐渐引起人们的注意。
* Linux操作系统的诞生、发展、和成长过程依赖于五个重要支柱：unix操作系统、minix操作系统、GNU计划、POSIX标准和互联网。
* GNU计划：GNU是“GNU is Not Unix”的递归缩写，由Richard M.Stallman于1984年创办,旨在开发一个免费、类unix的操作系统-GNU系统及其开发工具；Emacs编辑系统、BASH shell程序、GCC、GDB等开发工具都是GNU组织的产品。
* 1992年Linux与其他GNU软件结合，完全自由的操作系统正式诞生。该操作系统往往被称为“GNU/Linux”或简称。
* POSIX标准：POSIX标准定义了操作系统应该为应用程序提供的接口标准，POSIX标准用来统一Unix、Linux各分支编程接口，以提高其通用型和可移植性。

Linux和Unix的联系

* UNIX系统是工作站上最常用的操作系统，它是一个多用户、多任务的实时操作系统，允许多人同时访问计算机， 并同时运行多个任务。UNIX系统具有稳定、高效、安全、方便、功能强大等诸多优点，自20世纪70年代开始便运行在许多大型和小型计算机上。
* UNIX虽然是一个安全、稳定且功能强大的操作系统，但它也一直是一种大型的而且对运行平台要求很高的操作系统，只能在工作站或小型机上才能发挥全部功能，并且价格昂贵，对普通用户来说是可望而不可及的，这为后来Linux的崛起提供了机会，Linux是一个类UNIX操作系统。
* Linux是免费的、不受版权制约、与UNIX兼容的操作系统。
* Linux在x86架构上实现了UNIX系统的全部特性，具有多用户多任务的能力，同时保持了高效性和稳定性，Linux具有如下的优秀的特点：

1. 开放性；
2. 完全免费；
3. 多用户，多任务；
4. 设备独立性；
5. 丰富的网络功能；
6. 可靠的系统安全性；

类Unix系统

目录

* 目录是一组相关文件的集合。
* 一个目录下面除了可以存放文件之外还可以存放其他目录，即可包含子目录。
* 在确定文件、目录位置时，DOS和Unix/Linux都采用“路径名+文件名”的方式。路径反映的是目录与目录之间的关系。

路径

Unix/Linux路径由到达定位文件的目录组成。在Unix/Linux系统中组成路径的目录分割符为斜杠“/”，而DOS则用反斜杠“\”来分割各个目录。

路径分为**绝对路径**和**相对路径**：

* 绝对路径

1. 绝对路径是从目录树的树根“/”目录开始往下直至到达文件所经过的所有节点目录。
2. 下级目录接在上级目录后面用“/”隔开。
3. 注意：绝对路径都是从“/”开始的，所以第一个字符一定是“/”。

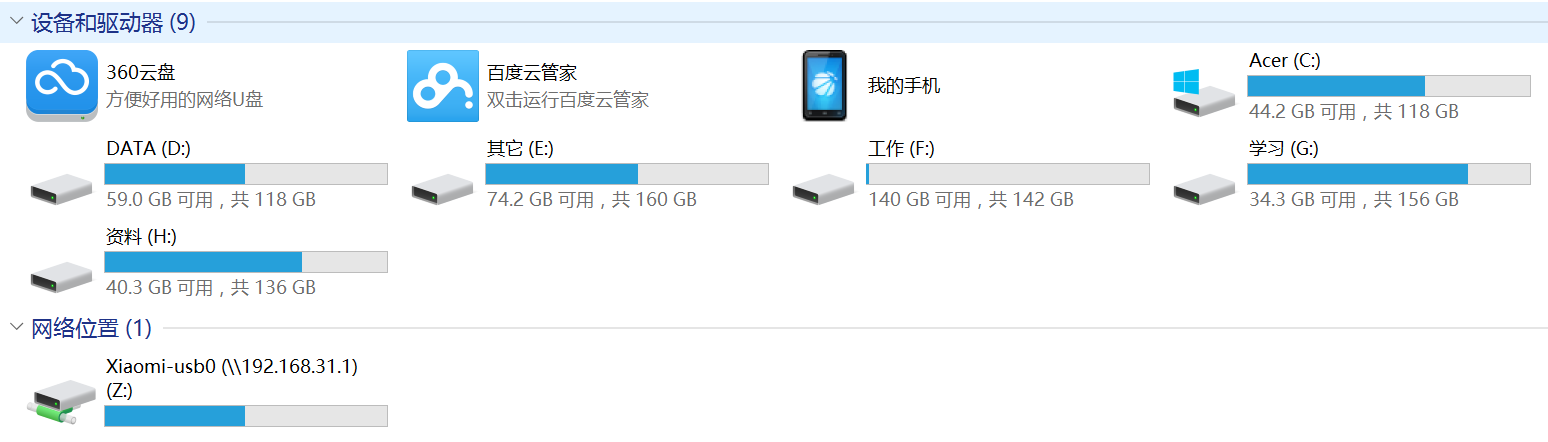
* 相对路径

1. 相对路径是指目标目录相对于当前目录的位置。
2. 如果不在当前目录下，则需要使用两个特殊目录“.”和“”了。目录“.”指向当前目录，而目录“..”。

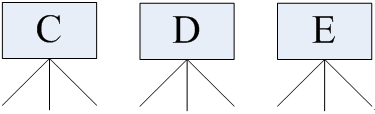
文件系统

文件系统区别

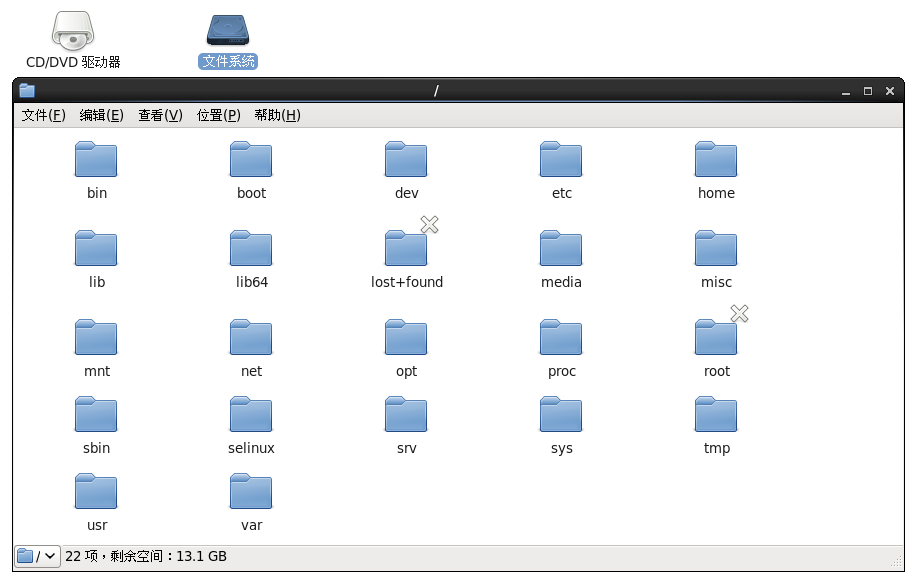
在 windows 平台下，打开“计算机”，我们看到的是一个个的驱动器盘符：



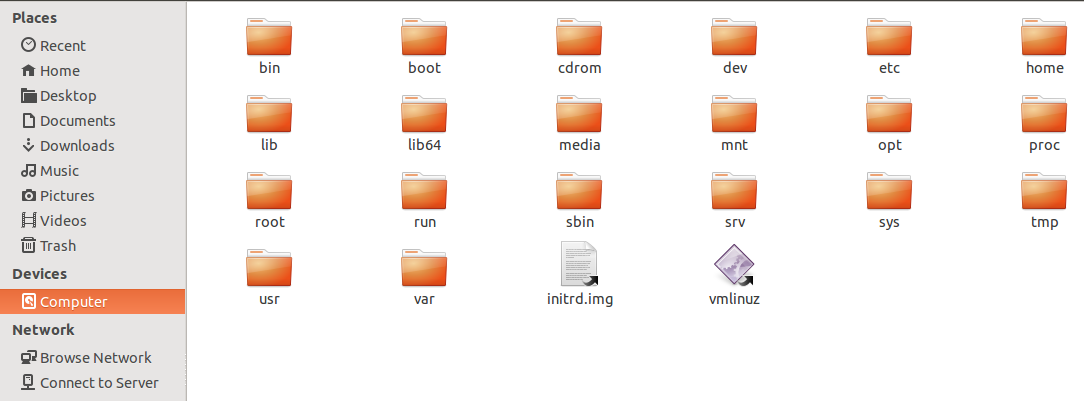
每个驱动器都有自己的根目录结构，这样形成了多个树并列的情形，如图所示：



在 Linux 下，我们是看不到这些驱动器盘符，我们看到的是文件夹（目录）：



RedHat

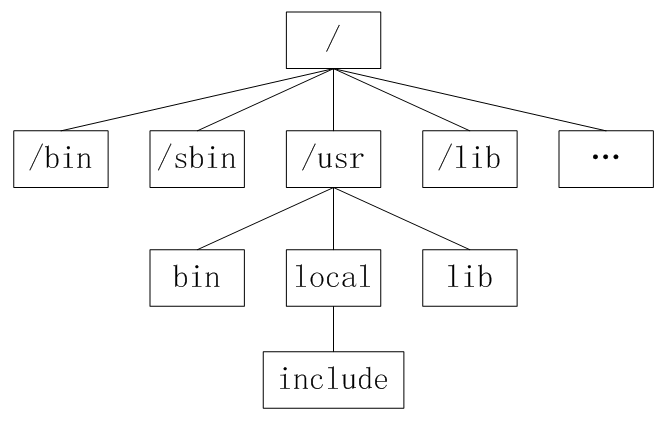


Ubuntu

在早期的 UNIX 系统中，各个厂家各自定义了自己的 UNIX 系统文件目录，比较混乱。Linux 面世不久后，对文件目录进行了标准化，于1994年对根文件目录做了统一的规范，推出 FHS ( Filesystem Hierarchy Standard ) 的 Linux 文件系统层次结构标准。FHS 标准规定了 Linux 根目录各文件夹的名称及作用，统一了Linux界命名混乱的局面。

与Windows操作系统类似，所有Unix/Linux的数据都是由文件系统按照树型目录结构管理的。而且Unix/Linux操作系统同样要区分文件的类型，判断文件的存取属性和可执行属性。

Unix/Linux也采用了树状结构的文件系统，它由目录和目录下的文件一起构成。但Unix/Linux文件系统不使用驱动器这个概念，而是使用单一的根目录结构，所有的分区都挂载到单一的“/”目录上，其结构示意图如图所示：



无论何种版本的 Linux 发行版，桌面、应用是 Linux 的外衣，文件组织、目录结构才是Linux的内心。

Linux根目录

* **/：**根目录，一般根目录下只存放目录，在Linux下有且只有一个根目录。所有的东西都是从这里开始。当你在终端里输入“/home”，你其实是在告诉电脑，先从/（根目录）开始，再进入到home目录。
* **/bin: /usr/bin:** 可执行二进制文件的目录，如常用的命令ls、tar、mv、cat等。
* **/boot：**放置linux系统启动时用到的一些文件，如Linux的内核文件：/boot/vmlinuz，系统引导管理器：/boot/grub。
* **/dev：**存放linux系统下的设备文件，访问该目录下某个文件，相当于访问某个设备，如使用cat命令读取/dev/input/mice 即是在读取鼠标事件。
* **/etc**：系统配置文件存放的目录，不建议在此目录下存放可执行文件，重要的配置文件有 /etc/inittab、/etc/fstab、/etc/init.d、/etc/X11、/etc/sysconfig、/etc/xinetd.d。
* **/home：**系统默认的用户家目录，新增用户账号时，用户的家目录都存放在此目录下，~表示当前用户的家目录，~edu 表示用户 edu 的家目录。
* **/lib**:/usr/lib: /usr/local/lib：系统使用的函数库的目录，程序在执行过程中，需要调用一些额外的参数时需要函数库的协助。
* /lost+fount：系统异常产生错误时，会将一些遗失的片段放置于此目录下。
* /mnt: /media：光盘默认挂载点，通常光盘挂载于 /mnt/cdrom 下，也不一定，可以选择任意位置进行挂载。
* /opt：给主机额外安装软件所摆放的目录。
* **/proc**：此目录的数据都在内存中，如系统核心，外部设备，网络状态，由于数据都存放于内存中，所以不占用磁盘空间，比较重要的目录有 /proc/cpuinfo、/proc/interrupts、/proc/dma、/proc/ioports、/proc/net/\* 等。
* **/root**：系统管理员root的家目录（宿主目录）。
* /sbin: /usr/sbin: /usr/local/sbin：放置系统管理员使用的可执行命令，如fdisk、shutdown、mount 等。与 /bin 不同的是，这几个目录是给系统管理员 root使用的命令，一般用户只能"查看"而不能设置和使用。
* **/tmp**：一般用户或正在执行的程序临时存放文件的目录，任何人都可以访问，重要数据不可放置在此目录下。
* **/srv：**服务启动之后需要访问的数据目录，如 www 服务需要访问的网页数据存放在 /srv/www 内。
* **/usr**：应用程序存放目录，/usr/bin 存放应用程序，/usr/share 存放共享数据，/usr/lib 存放不能直接运行的，却是许多程序运行所必需的一些函数库文件。/usr/local: 存放软件升级包。/usr/share/doc: 系统说明文件存放目录。/usr/share/man: 程序说明文件存放目录。/usr/include:存放头文件。
* **/var：**放置系统执行过程中经常变化的文件，如随时更改的日志文件 /var/log，/var/log/message：所有的登录文件存放目录，/var/spool/mail：邮件存放的目录，/var/lib/mysql：MySQL数据库成功安装后，使用的库、表相关文件，存放在该目录下。

Linux 命令

概述

很多人在电视或电影中看到过类似的场景，黑客面对一个黑色的屏幕，上面飘着密密麻麻的字符，一通噼里啪啦的按键过后，各种安全机构的资料就被窃取了。

Linux 刚出世时没有什么图形界面，所有的操作全靠命令完成，就如同电视里的黑客那样，充满了神秘与晦涩。

近几年来，尽管 Linux 发展得非常迅速，图形界面越来越友好，但是在真正的开发过程中，Linux 命令行依然占有非常重要的地位，许多功能在Linux命令行处理起来要远快于图形化界面。可以说**不会命令行操作Linux系统，就不算会 Linux**。

Linux系统提供了大量的命令，利用它可以有效地完成大量的工作，如磁盘操作、文件存取、目录操作、进程管理、文件权限设定等。Linux 发行版本最少的命令也有 200多个，这里只介绍比较重要和使用频率最多的命令。

命令格式

command [-options] [parameter1] …

说明：

* command：命令名，相应功能的英文单词或单词的缩写
* [-options]：选项，可用来对命令进行控制，可以省略，[]表可选
* parameter1 …：传给命令的参数，可以是零个一个或多个



查看帮助

1) --help

一般Linux命令自带的帮助信息。可以使用 –help查看。如：想查看命令 “ls”的用法，可以在命令行中键入：**ls –help**

当然，并不是所有命令都自带这个选项。

2) man

man 是 Linux系统支持的一款帮助手册，当然需要事先安装好该手册才能使用。手册中包含了绝大部分的命令、函数使用说明。共包含9个章节（section），使用 man man 命令可以查看：

**man 中各个 section 意义如下：**

1. Standard commands（标准命令）
2. System calls（系统调用，如open,write）
3. Library functions（库函数，如printf,fopen）
4. Special devices（设备文件的说明，/dev下各种设备）
5. File formats（文件格式，如passwd）
6. Games and toys（游戏和娱乐）
7. Miscellaneous（杂项、惯例与协定等，例如Linux档案系统、网络协定、ASCII 码；environ全局变量）
8. Administrative Commands（管理员命令，如ifconfig）
9. Kernel routines [non-standard]

man使用格式如下：

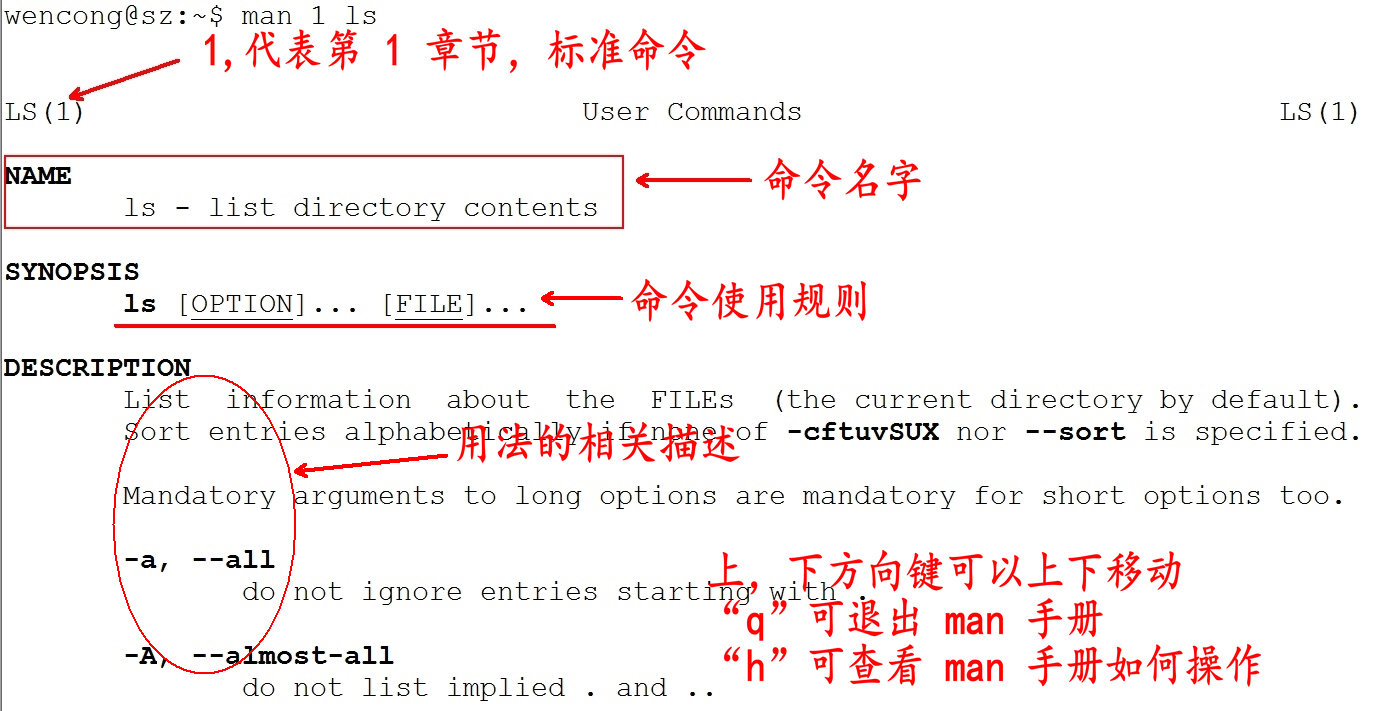
man [选项] 命令名

man设置了如下的功能键：

| **功能键** | **功能** |
| --- | --- |
| 空格键 | 显示手册页的下一屏 |
| Enter键 | 一次滚动手册页的一行 |
| b | 回滚一屏 |
| f | 前滚一屏 |
| q | 退出man命令 |
| h | 列出所有功能键 |
| /word | 搜索word字符串 |

默认首先从第一章起进行筛选，提供帮助。使用后面的章节时，应手动指定章节号，来浏览所需内容。如，我们想查看命令“ls”的用法，可以在命令行键入：

**man 1 ls** ( 1：为数字“1”，代表第 1 个 section，标准命令 )



实际上，我们不用指定第几个章节也可以查看，如，**man ls**。但是，有这个一种情况，假如，命令的名字和函数的名字刚好重名（如：printf），它既是命令，也可以是库函数，如果，我们不指定章节号，man printf，它只查看命令的用法，不会查询函数的用法，因为 man 是按照手册的章节号的顺序进行搜索的。



所以，使用 man 手册时，最好指定章节号。

使用技巧

1) 自动补全

在敲出命令的前几个字母的同时，按下tab键，系统会自动帮我们补全命令。

2) 历史命令

当系统执行过一些命令后，可按↓↑键翻看以前的命令。

History命令可以列出历史执行过的命令记录，并且每条记录带有一个序号。

使用 "!"+过往命令序号 可快速执行该命令。

文件管理

Unix/Linux对数据文件(\*.mp3、\*.bmp)，程序文件(\*.c、\*.h、\*.o)，设备文件（LCD、触摸屏、鼠标），网络文件( socket ) 等的管理都抽象为文件，使用统一的方式方法管理。

在Unix/Linux操作系统中也必须区分文件类型，通过文件类型可以判断文件属于可执行文件、文本文件还是数据文件。在Unix/Linux系统中文件可以没有扩展名。

1. 切换工作目录：cd

在使用命令行操作Unix/Linux系统时，需要频繁更换工作目录。cd命令可以帮助用户切换。Linux所有的目录和文件名大小写敏感。

cd后面可跟**绝对路径**，也可以跟**相对路径**。如果省略目录，则默认切换到当前用户的主目录。常用方法：

| **命令** | **含义** |
| --- | --- |
| cd | 切换到当前用户的主目录(/home/用户目录)，用户登陆的时候，默认的目录就是用户的主目录。 |
| cd ~ | 切换到当前用户的主目录(/home/用户目录) |
| cd . | 切换到当前目录 |
| cd .. | 切换到上级目录 |
| cd - | 可进入上一个进入的目录 |

**注意：**如果路径是从根路径开始，则最前面需要添加 “/”，如“/mnt”。这是采用绝对路径法进入目录。如果没有“/”，直接写目录名，则相当于默认选用相对路径法进入目录。等同于 进入“./目录名/”



1. 查看文件信息：ls

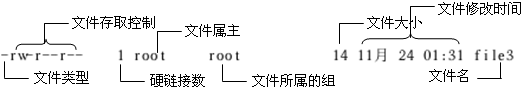
ls是英文单词list的简写，其功能为列出目录的内容，是用户最常用的命令之一，它类似于DOS下的dir命令。

Linux文件或者目录名称最长可以有256个字符，“.”代表当前目录，“..”代表上一级目录，以“.”开头的文件为隐藏文件，需要用 -a 参数才能显示。

**ls常用参数**：

| **参数** | **含义** |
| --- | --- |
| -a | 显示指定目录下所有子目录与文件，包含隐藏文件。 |
| -l | 以列表方式显示文件的详细信息。 |
| -d | 查看目录本身的详细信息。 |
| -F | 显示出文件类型提示符。 |
| -i | 查看inode号。 |
| -h | 配合-l使用，以人类习惯单位命名方式显示文件大小。 |

如下图所示：



在Unix/Linux系统中，允许使用特殊字符来同时引用多个文件名，这些特殊字符被称为通配符。

| **通配符** | **含义** |
| --- | --- |
| \* | 文件代表文件名中所有字符 |
| ls te\* | 查找以te开头的文件 |
| ls \*html | 查找结尾为html的文件 |
| ？ | 代表文件名中任意一个字符 |
| ls ?.c | 只找第一个字符任意，后缀为.c的文件 |
| ls a.? | 只找只有3个字符，前2字符为a.，最后一个字符任意的文件 |
| ls [a-f]\* | 找到从a到f范围内的的任意一个字符开头的文件 |
| \ | 如果要使通配符作为普通字符使用，可以在其前面加上转义字符。“?”和“\*”处于方括号内时不用使用转义字符就失去通配符的作用。 |
| ls \\*a | 查找文件名为\*a的文件 |

#### 文件分类

通常，Unix/Linux系统中常用的文件类型有7种：

普通文件 -

目录文件 d

管道文件 p

套接字文件 s

软链接文件 l

块设备文件 b

字符设备文件 c

* 普通文件

普通文件是计算机操作系统用于存放数据、程序等信息的文件，一般都长期存放于外存储器（磁盘、磁带等）中。普通文件一般包括文本文件、数据文件、可执行的二进制程序文件等。

在Unix/Linux中可以通过file命令来查看文件的类型。如果file文件后面携带文件名，则查看指定文件的类型，如果携带通配符“\*”，则可以查看当前目录下的所有文件的类型。

* 目录文件

Unix/Linux系统把目录看成一种特殊的文件，利用它构成文件系统的树型结构。

目录文件只允许系统管理员对其进行修改，用户进程可以读取目录文件，但不能对它们进行修改。

每个目录文件至少包括两个条目，“..”表示上一级目录，“.”表示该目录本身。

* 设备文件

Unix/Linux系统把每个设备都映射成一个文件，这就是设备文件。它是用于向I/O设备提供连接的一种文件，分为字符设备和块设备文件。

字符设备的存取以一个字符为单位，块设备的存取以字符块为单位。每一种I/O设备对应一个设备文件，存放在/dev目录中，如行式打印机对应/dev/lp，第一个软盘驱动器对应/dev/fd0。

* 管道文件

管道文件也是Unix/Linux中较特殊的文件类型，这类文件多用于进程间的通信。

* 软链接文件

软连接文件，类似于 windows 下的快捷方式。

#### 文件权限

文件权限就是文件的访问控制权限，即哪些用户和组群可以访问文件以及可以执行什么样的操作。

Unix/Linux系统是一个典型的多用户系统，不同的用户处于不同的地位，对文件和目录有不同的访问权限。为了保护系统的安全性，Unix/Linux系统除了对用户权限作了严格的界定外，还在用户身份认证、访问控制、传输安全、文件读写权限等方面作了周密的控制。

在 Unix/Linux中的每一个文件或目录都包含有访问权限，这些访问权限决定了谁能访问和如何访问这些文件和目录。

#### 用户权限

通过设定权限可以从以下三种访问方式限制访问权限：

* 只允许用户自己访问（所有者）

所有者就是创建文件的用户，用户是所有用户所创建文件的所有者，用户可以允许所在的用户组能访问用户的文件。

* 允许一个预先指定的用户组中的用户访问（用户组）

用户都组合成用户组，例如，某一类或某一项目中的所有用户都能够被系统管理员归为一个用户组，一个用户能够授予所在用户组的其他成员的文件访问权限。

* 允许系统中的任何用户访问（其他用户）

用户也将自己的文件向系统内的所有用户开放，在这种情况下，系统内的所有用户都能够访问用户的目录或文件。在这种意义上，系统内的其他所有用户就是 other 用户类

可以类比 QQ 空间的访问权限：

* + 这个 QQ 空间是属于我的，我相当于管理者（也就是“所有者”），我想怎么访问就怎么访问。
  + 同时，我可以设置允许 QQ 好友访问，而这些 QQ 好友则类似于“用户组”。
  + 当然，我可以允许所有人访问，这里的所有人则类似于“其他用户”。

#### 访问权限

用户能够控制一个给定的文件或目录的访问程度，一个文件或目录可能有读、写及执行权限：

* 读权限（r）

对文件而言，具有读取文件内容的权限；对目录来说，具有浏览目录的权限。

* 写权限（w）

对文件而言，具有新增、修改文件内容的权限；对目录来说，具有删除、移动目录内文件的权限。

* 可执行权限（x）

对文件而言，具有执行文件的权限；对目录了来说该用户具有进入目录的权限。

注意：Unix/Linux系统只允许文件的属主(所有者)或超级用户改变文件的读写权限。

1. 清屏命令：clear

Clear命令的作用是，清除终端上显示的内容。实际上是将终端中已有的内容向前翻滚一屏。也可使用快捷键实现：Ctrl + l (“l”为字母 )。

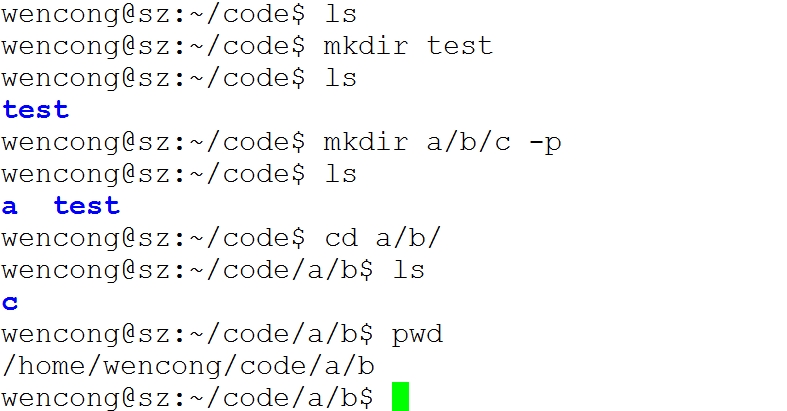
1. 创建空文件： touch

touch 命令，可以在当前目录下创建一个空文件。

1. 创建空目录：mkdir

mkdir命令可以创建一个新的空目录。参数 -p可递归创建多级空目录。

**注意**：新建目录的名称不能与当前目录中已有目录、文件同名。且目录创建者必须对当前目录具有写权限。



1. 删除目录：rmdir

使用rmdir命令可删除一个空目录。必须离开目录，并且目录必须为空，否则删除失败。

1. 删除文件、目录：rm -r

可通过rm删除文件或目录。

**特别注意：rm命令删除的文件、目录“不能”恢复！！！**

为了防止文件误删，可在rm后使用-i参数以逐个确认要删除的文件。

常用参数：

| **参数** | **含义** |
| --- | --- |
| -i | 以进行交互式方式执行 |
| -f | 强制删除，忽略不存在的文件，无需提示 |
| -r | 递归地删除目录下的内容，删除文件夹时必须加此参数 |

1. 建立链接文件：ln

链接文件分为软链接、硬链接两种：

#### 硬链接

ln 源文件 链接文件

硬链接文件和源文件之间具备“同步”功能。

其原因是，链接文件间具有相同的inode号，对应计算机磁盘上相同的盘块。

硬链接只能链接普通文件，不能链接目录。

#### 软链接

ln -s 源文件 链接文件

如果没有-s选项代表建立一个硬链接文件。

源文件删除则软链接失效。源文件搬移也可能造成连接失效，因此建议使用绝对路径法创建软连接。

1. 拷贝文件：cp

cp命令的功能是将给出的文件或目录复制到另一个文件或目录中，相当于DOS下的copy命令。

常用选项说明：

| **选项** | **含义** |
| --- | --- |
| -a | 该选项通常在复制目录时使用，它保留链接、文件属性，并递归地复制目录，简单而言，保持文件原有属性。 |
| -r | 若给出的源文件是目录文件，则cp将递归复制该目录下的所有子目录和文件，目标文件必须为一个目录名。 |
| -i | 交互式复制，在覆盖目标文件之前将给出提示要求用户确认 |
| -v | 显示拷贝进度 |

1. 移动文件：mv

用户可以使用mv命令来移动文件或目录，也可以给文件或目录重命名。

常用选项说明：

| **选项** | **含义** |
| --- | --- |
| -f | 禁止交互式操作，如有覆盖也不会给出提示 |
| -i | 确认交互方式操作，如果mv操作将导致对已存在的目标文件的覆盖，系统会询问是否重写，要求用户回答以避免误覆盖文件 |
| -v | 显示移动进度 |

1. 压缩解压

#### 获取文件类型：file

Linux系统文件类型不是根据文件扩展名分类的，通过file命令可以确认文件具体类型。

#### 归档管理：tar

计算机中的数据经常需要备份，tar是Unix/Linux中最常用的备份工具，此命令可以把一系列文件归档到一个大文件中，也可以把档案文件解开以恢复数据。

tar使用格式

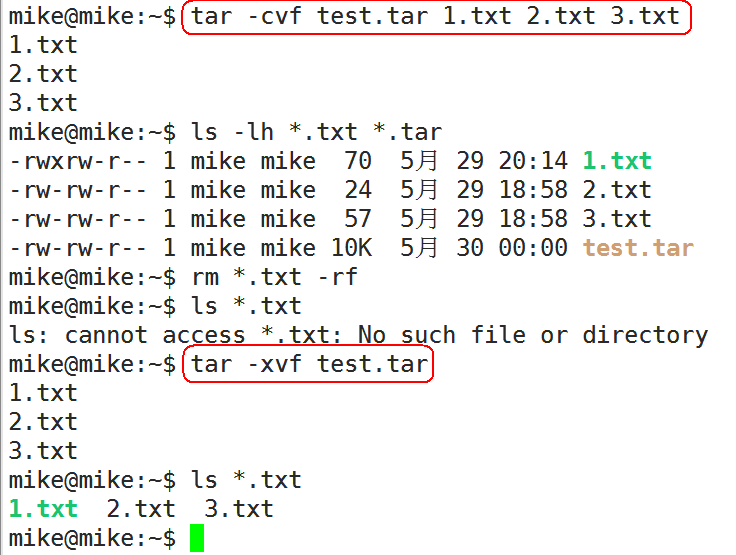
tar [参数] 打包文件名 文件

tar命令比较特殊，参数前面可以使用“-”，也可以不使用。

常用参数：

| **参数** | **含义** |
| --- | --- |
| -c | 生成档案文件，创建打包文件 |
| -v | 列出归档解档的详细过程，显示进度 |
| -f | 指定档案文件名称，f后面一定是.tar文件，放选项最后 |
| -t | 列出档案中包含的文件 |
| -x | 解开档案文件 |

注意：除了f需要放在参数的最后，其它参数的顺序任意。



#### 文件压缩解压：gzip

gzip使用格式如下：

gzip [选项] 被压缩文件

常用选项：

| **选项** | **含义** |
| --- | --- |
| -d | 解压 |
| -r | 压缩所有子目录 |



但，gzip在应用中有两个严重的问题，一是不保留源文件。二是不能压缩多个文件和目录。而实际使用中，通常需要将多个文件压缩成一个文件，或直接将某一个目录进行压缩。

因此，通常将 tar 和 gzip结合使用，完成打包压缩。tar命令并没有压缩的功能，只负责打包文件，但不压缩，用gzip压缩tar打包后的文件。其扩展名一般用xxxx.tar.gz。

**压缩用法：**tar czvf 压缩包包名 文件1 文件2 ...

例如：**tar zcvf test.tar.gz 1.c 2.c 3.c 4.c**把 1.c 2.c 3.c 4.c 压缩成 test.tar.gz

**解压用法：** tar zxvf 压缩包包名

**解压到指定目录：**-C （大写字母“C”）

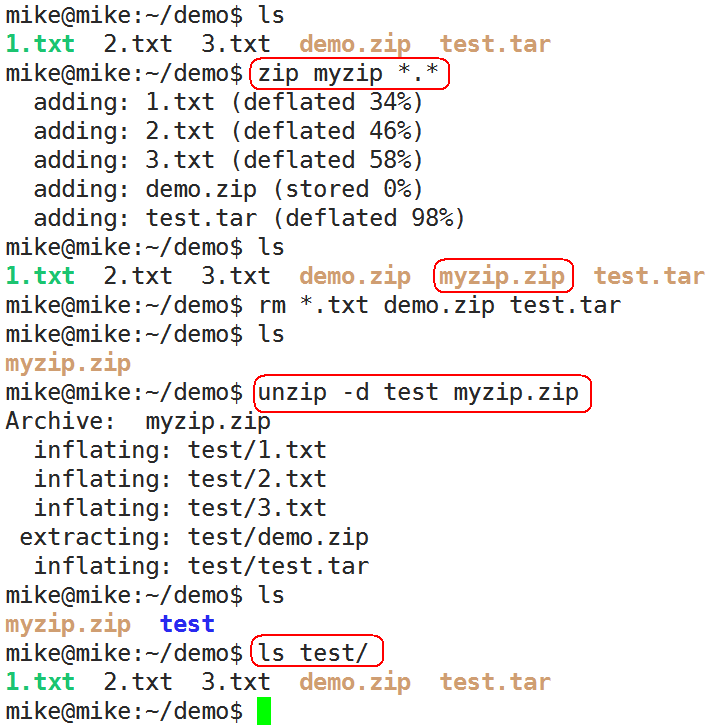
例如：tar -xvf new.tar.gz -C ./test/ 将new.tar.gz压缩包内文件解压到当前目录下的 test 目录中。

#### 文件压缩解压：bzip2

* tar与bzip2命令结合使用实现文件打包、压缩(用法和gzip一样)。
* tar只负责打包文件，但不压缩，用bzip2压缩tar打包后的文件，其扩展名一般用xxxx.tar.bz2。
* 在tar命令中增加一个选项(-j)可以调用bzip2实现了一个压缩的功能，实行一个先打包后压缩的过程。
* 压缩用法：tar cjvf 压缩包包名 文件...(tar jcvf bk.tar.bz2 \*.c)
* 解压用法：tar xjvf 压缩包包名 (tar jxvf bk.tar.bz2)

#### 文件压缩解压：zip、rar

* 通过zip压缩文件的目标文件不需要指定扩展名，默认扩展名为zip。
* 压缩文件：zip [-r] 目标文件(没有扩展名) 源文件
* 解压文件：unzip -d 解压后目录文件 压缩文件



类似的，Linux同样支持rar格式文件的压缩。不过需要事先安装rar工具。

**压缩**： rar a -r xxx.rar 待压缩文件群

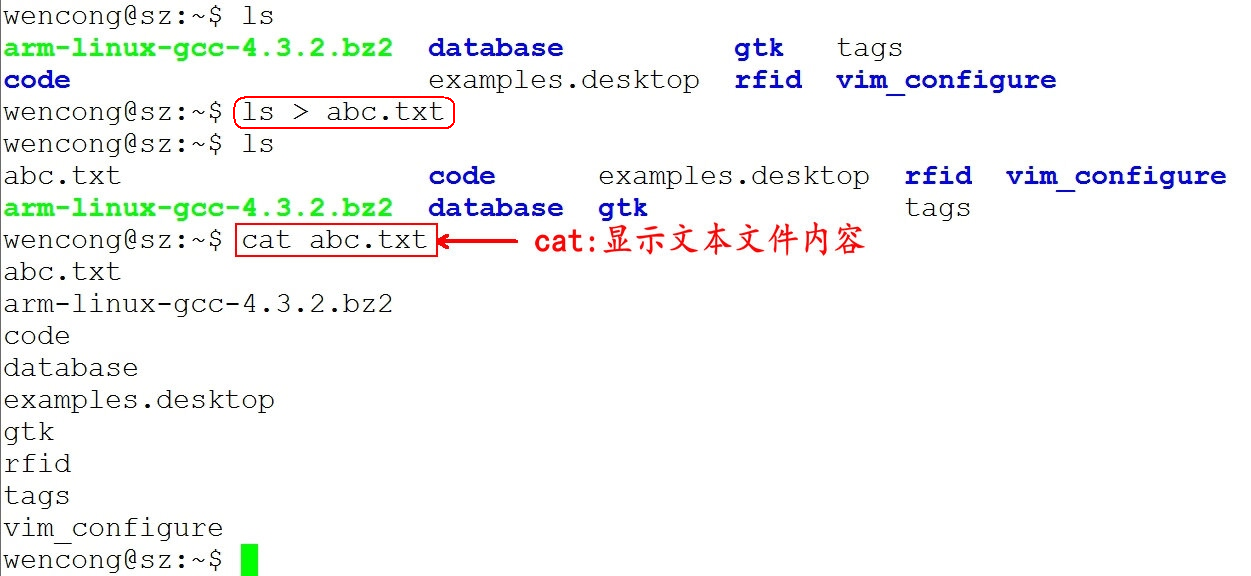
**解压**：rar x xxx.rar

1. 其他较常用命令

#### 输出重定向命令：>

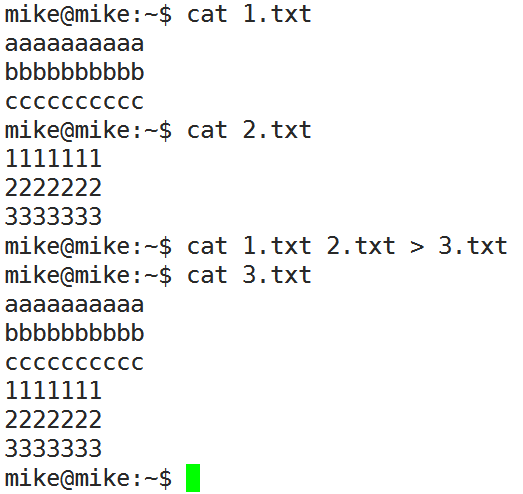
Linux允许将命令执行结果重定向到一个文件，本应显示在终端上的内容保存到指定文件中。

如：**ls > test.txt** ( test.txt 如果不存在，则创建，存在则覆盖其内容 )



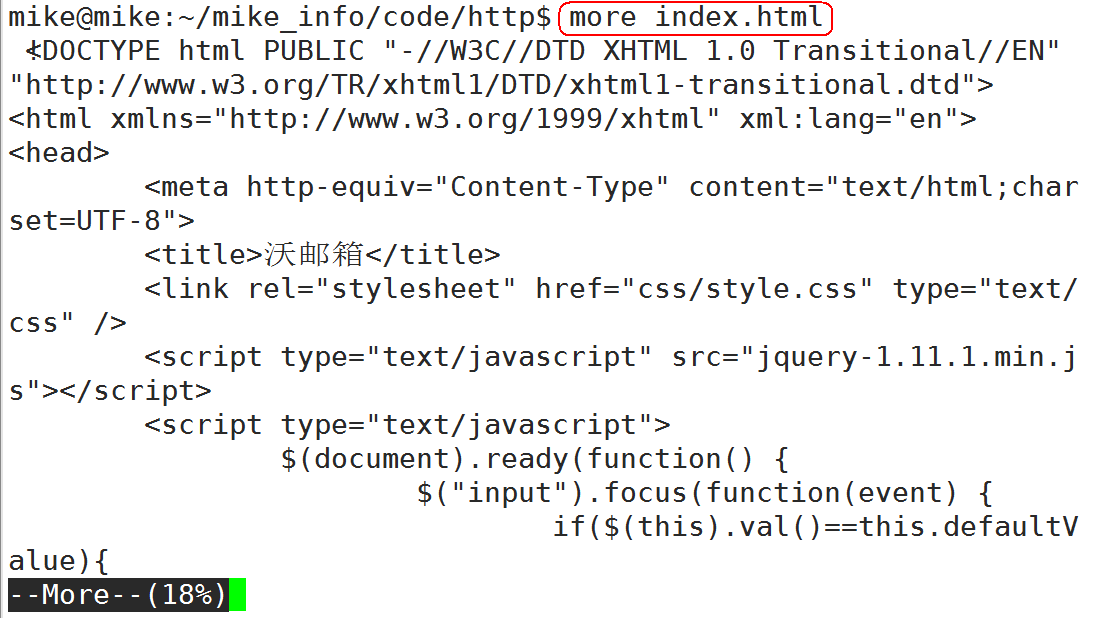
> 输出重定向会覆盖原来的内容，**>>** 输出重定向则会追加到文件的尾部。

#### 查看文件内容：cat



#### 分屏显示：more、less、head、tail

查看内容时，在信息过长无法在一屏上显示时，会出现快速滚屏，使得用户无法看清文件的内容，此时可以使用more命令，每次只显示一页，按下空格键可以显示下一页，按下Enter 键，显示下一行。按下q键退出显示，按下h键可以获取帮助。



也可以使用 less命令。每次只显示一页，按下空格键可以显示下一页，按下Enter 键，显示下一行。与more 的区别是，less命令可以借助↑↓按键前后逐行翻看。Q退出。

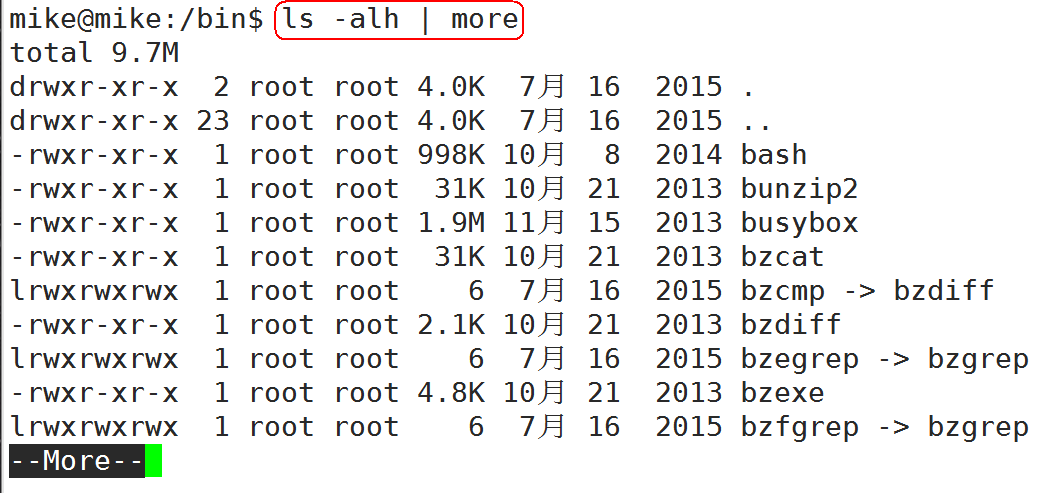
head -5 file.txt 查看文件前 5 行

tail -5 file.txt 查看文件后 5 行

#### 管道：|

管道：一个命令的输出可以通过管道做为另一个命令的输入。

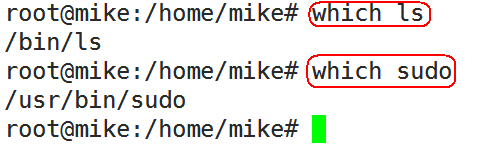
可以理解现实生活中的管子，管子的一头塞东西进去，另一头取出来。这里“|”的左右分为两端，左端塞入(写)东西，右端取出(读)东西。



#### 显示当前工作路径：pwd

使用pwd命令可以显示当前的工作目录，直接输入pwd即可，无需参数。

#### 查看命令位置：which



用户管理

用户是Unix/Linux系统工作中重要的一环，用户管理包括用户与组账号的管理。在Unix/Linux系统中，不论是由本机或是远程登录系统，每个系统都必须拥有一个账号，并且对于不同的系统资源拥有不同的使用权限。

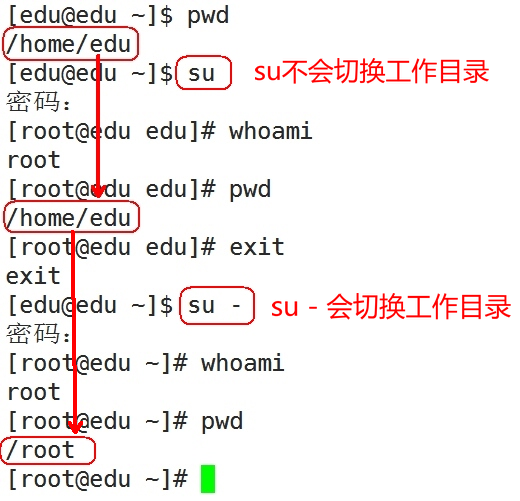
Unix/Linux系统中的root账号通常用于系统的维护和管理，它对Unix/Linux操作系统的所有部分具有不受限制的访问权限。

在Unix/Linux安装的过程中，系统会自动创建许多用户账号，而这些默认的用户就称为“标准用户”。

不推荐直接使用root账号登录系统。

### 切换用户：su

可以通过su命令切换用户，su后面可以加“-”。su和su –命令不同之处在于，su -切换到对应的用户时会将当前的工作目录自动转换到切换后的用户主目录：



**注意：**如果是ubuntu平台，需要在命令前加“sudo”，如果在某些操作需要管理员才能操作，ubuntu无需切换到root用户即可操作，只需加“sudo”即可。sudo是ubuntu平台下允许系统管理员让普通用户执行一些或者全部的root命令的一个工具，减少了root 用户的登陆和管理时间，提高了安全性。

### 添加、删除用户：adduser、deluser

* adduser 新建用户
* deluser 删除用户
* cat /etc/passwd 查看用户组

### 添加、删除用户组：addgroup、delgroup

* addgroup 新建用户组
* delgroup 删除用户组
* cat /etc/group 查看用户组

### 设置用户密码：passwd

在Unix/Linux中，超级用户可以使用passwd命令为普通用户设置或修改用户口令。用户也可以直接使用该命令来修改自己的口令，而无需在命令后面使用用户名。

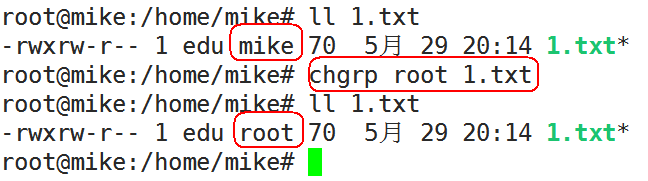
### 修改文件所有者：chown

使用方法：chown 用户名 文件或目录名



### 修改文件所属组：chgrp

使用方法：chgrp 用户组名 文件或目录名



修改文件到指定用户、用户组：

语法：sudo chown 用户名:用户组名 文件、目录名

可直接同时修改文件的所有者和所属组。如：

sudo chown nobody:nogroup a.c

可将a.c文件设置到 nobody用户、nogroup 用户组下。

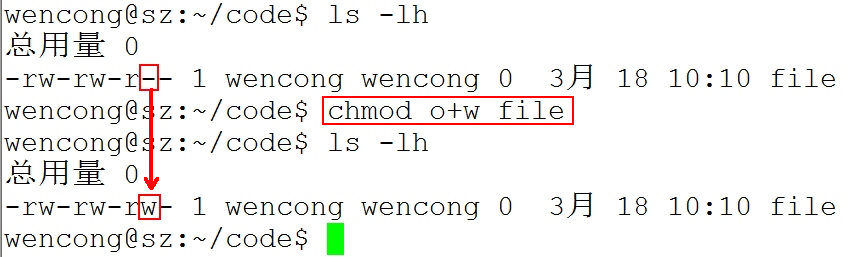
### 修改文件权限：chmod

chmod 修改文件权限有两种使用格式：字母法与数字法。

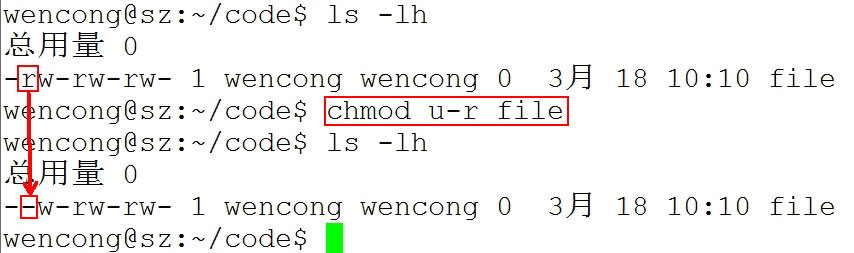
**字母法：**chmod u/g/o/a +/-/= rwx 文件

| **[ u/g/o/a ]** | **含义** |
| --- | --- |
| u | user 表示该文件的所有者 |
| g | group 表示与该文件的所有者属于同一组( group )者，即用户组 |
| o | other 表示其他以外的人 |
| a | all 表示这三者皆是 |
| **[ +-= ]** | **含义** |
| + | 增加权限 |
| - | 撤销权限 |
| = | 设定权限 |
| **rwx** | **含义** |
| r | read 表示可读取，对于一个目录，如果没有r权限，那么就意味着不能通过ls查看这个目录的内容。 |
| w | write 表示可写入，对于一个目录，如果没有w权限，那么就意味着不能在目录下创建新的文件。 |
| x | excute 表示可执行，对于一个目录，如果没有x权限，那么就意味着不能通过cd进入这个目录。 |

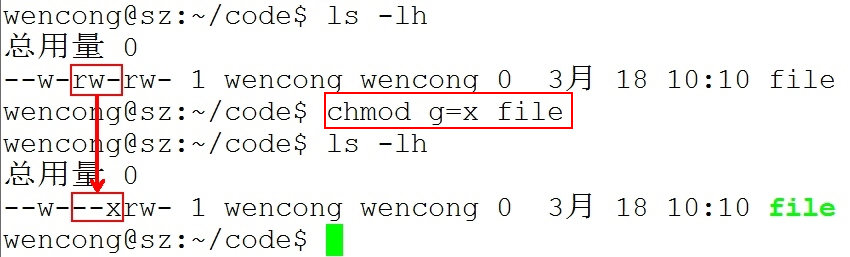
**chmod o+w file** 给文件file的其它用户增加写权限：



**chmod u-r file** 给文件file的拥有者减去读的权限：



**chmod g=x file**设置文件file的同组用户的权限为可执行，同时去除读、写权限：



**数字法：**“rwx” 这些权限也可以用数字来代替

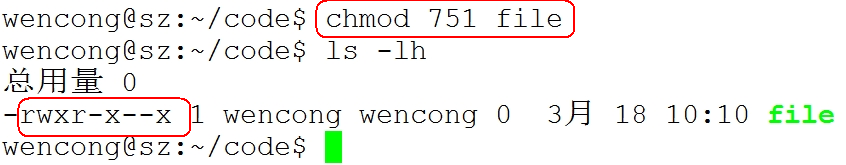
|  |  |
| --- | --- |
| r | 读取权限，数字代号为 "4" |
| w | 写入权限，数字代号为 "2" |
| x | 执行权限，数字代号为 "1" |
| - | 不具任何权限，数字代号为 "0" |

如执行：chmod u=rwx,g=rx,o=r filename

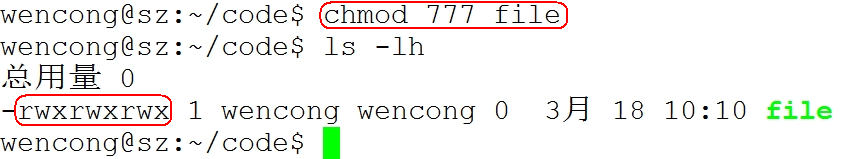
就等同于：chmod u=7,g=5,o=4 filename

**chmod 751 file：**

* 文件所有者：读、写、执行权限
* 同组用户：读、执行的权限
* 其它用户：执行的权限



chmod 777 file：所有用户拥有读、写、执行权限



注意：如果想递归所有目录加上相同权限，需要加上参数“ -R ”。

如：chmod 777 test/ -R 递归 test 目录下所有文件加 777 权限。

系统管理

1. 查看进程信息：ps

进程是一个具有一定独立功能的程序，它是操作系统动态执行的基本单元。

ps命令可以查看进程的详细状况，常用选项(选项可以不加“-”)如下：

| **选项** | **含义** |
| --- | --- |
| -a | 显示终端上的所有进程，包括其他用户的进程 |
| -u | 显示进程的详细状态 |
| -x | 显示没有控制终端的进程 |

使用频率最高的组合为 ps aux 。 通常借助管道，与grep命令结合，筛选当前系统中所需要查看的进程的全部详细信息。 ps aux | grep xxx

另外，windows下任务管理器类似的功能，可以在命令行中使用 “top”命令来代替。不过只能用来查看进程信息。查看结束，可以使用“q”退出。

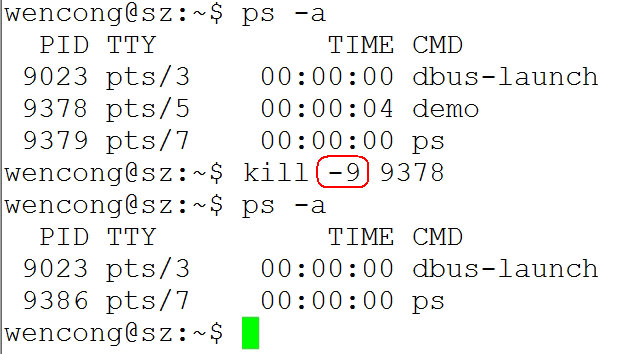
1. 终止进程：kill

kill命令可以发送指定的信号给指定进程，从而达到杀死、暂停等操作该进程的目的。需要配合进程id使用。

使用格式：

kill [-signal] pid

有些进程不能直接杀死，这时我们需要加参数“-9”，代表强制结束：



**kill -l** ：可以查看当前Linux系统所支持的所有的信号。

1. 关机重启

| **命令** | **含义** |
| --- | --- |
| reboot | 重新启动操作系统 |
| shutdown –r now | 重新启动操作系统，shutdown会给别的用户提示 |
| shutdown -h now | 立刻关机，其中now相当于时间为0的状态 |
| shutdown -h 20:25 | 系统在今天的20:25 会关机 |
| shutdown -h +10 | 系统再过十分钟后自动关机 |

1. 字符、图形界面切换

通过快捷键切换：

|  |  |
| --- | --- |
| Ctrl + Alt + F3 | 切换到字符界面 |
| Ctrl + Alt + F1 | 切换到图形界面 |

适用于 18.04系统。

查找与检索

1. find

语法：find 搜索目录位置 参数 搜索条件

**-name：**按名称搜索

find ./ -name "for\*.sh"

**-type：**按类型搜索

find ./ -type f/d/l/b/c/s/p

**-size：**按大小搜索

find ~/ -size +3M -size -8M M大写

find ~/ -size +3k -size -8k k小写

find ./ -size +3 -size -8 无单位，按扇区个数计算（一个扇区大小为 512B）

**-maxdepth：**按层级搜索：

find ./ -maxdepth 1 -name "\*.sh"

**-exec：**对搜索结果，执行某些命令

find ./ -maxdepth 1 -name "\*.sh" -exec ls -l {} \;

**xargs：**需要结合管道，将搜索结果指定给某个命令使用。

find ./ -type d | xargs ls -l

1. grep

按文件内容搜索文件。

grep -R/-r "待搜索的内容" 目录位置

1. find和grep命令结合

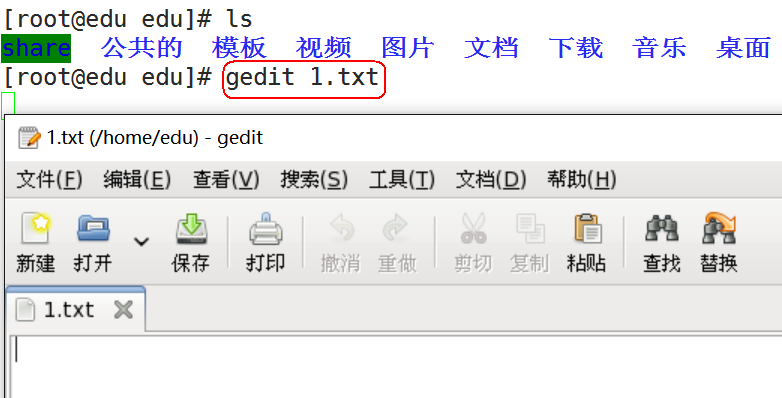
先使用find命令查找文件, 然后使用grep命令查找哪些文件包含某个字符串

find . -name "\*.c" | xargs grep -n "main"

Linux 编辑器

gedit

gedit是一个Linux环境下的文本编辑器，类似windows下的写字板程序，在不需要特别复杂的编程环境下，作为基本的文本编辑器比较合适。



vi/vim

vi介绍

vi 编辑器是 Unix系统中最常见的基础文本编辑器。Linux下升级为vim编辑器，它不仅兼容 vi 的所有指令，而且还有一些新的特性，例如 vim 可以撤消无限次、支持关键词自动完成、可以用不同的颜色来高亮你的代码。vim 普遍被推崇为类 vi 编辑器中最好的一个。出于历史的原因使用时大家习惯性的统一简称vi。

vim 编辑器在Linux界有编辑器之神的美誉，几乎所有的 Linux 发行版中都包含 vi 程序。

可以联网使用命令快捷安装： sudo apt-get install vim

由于vi 工作时，不需要图形界面，非常适合远程及嵌入式工作，是效率很高的文本编辑器，尽管在 Linux 上也有很多图形界面的编辑器可用，但vi的功能是那些图形编辑器所无法比拟的。

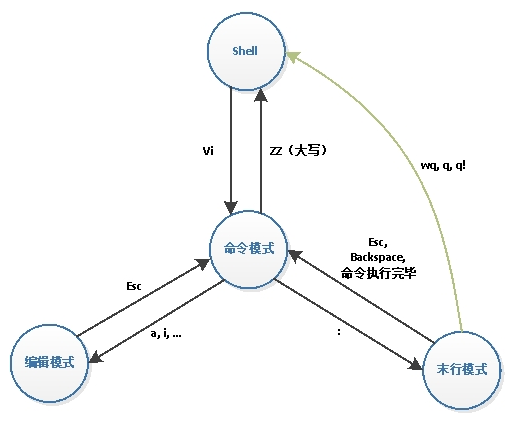
vi有三种基本工作模式：

+ 命令模式

+ 文本输入模式

+ 末行模式。

要想使用vi对文件进行编辑，必须熟练掌握这三种工作模式之间的工作关系。



vi基本操作

#### 1) 打开文件

**vi filename** 打开或新建一个文件，并将光标置于第一行行首，如果文件不存在，则会新建文件。

#### 2) 编辑文件

如果通过vi打开一个已经存在的文件，首先进入命令模式，此时输入的任何字符都被视为命令，不能输入内容。

#### 3) 保存文件

一定要先退出插入模式(编辑模式)按Esc进入命令模式，然后按“:wq”保存退出。或者使用shift + zz （按住 “shift” + 按两下“z”键）保存退出当前文件。

vi实用命令

#### 切换编辑模式

| **按键** | **功能** |
| --- | --- |
| a | 光标位置右边插入文字 |
| i | 光标位置当前处插入文字 |
| o(字母) | 光标位置下方开启新行 |
| O(字母) | 光标位置上方开启新行 |
| I | 光标所在行首插入文字 |
| A | 光标所在行尾插入文字 |
| s | 以删除一个字符为条件，切换工作模式 |
| S | 以删除一行为条件，切换工作模式 |

#### 退出vim

| **按键** | **功能** |
| --- | --- |
| ZZ(shift+z+z) | 保存退出 |
| :wq | 保存退出 |
| :x(小写) | 保存退出 |
| :w filename | 保存指定文件 |
| :q | 退出，如果文件修改但没有保存，会提示无法退出 |
| :q! | 退出，不保存 |

#### 移动光标

hjkl： 左下上右

gg：将光标移动至行首

G：将光标移动至行尾

gg = G 排版代码

nG 跳转至 n行（末行：行号、回车）

#### 复制粘贴

yy：复制光标所在行

nyy：复制光标所在行向后 n 行

p（小写）：将复制内容粘贴至，光标所在位置的下一行。

P（大写）：将复制内容粘贴至，光标所在位置的上一行。

区域选择复制：

1. 选择： 命令模式下，将光标移动至 待复制内容首字符，按 v （左下角 “可视”字样），使用hjkl 挪移光标，选中区域

2. 复制： y

3. 粘贴：p 将内容粘贴至，光标后。 P向前粘。

#### 剪切删除

dd：剪切光标所在行

ndd：剪切光标所在行向后 n 行

p（小写）：将剪切内容粘贴至，光标所在位置的下一行。

P（大写）：将剪切内容粘贴至，光标所在位置的上一行。

区域选择剪切：

1. 选择： 命令模式下，将光标移动至 待剪切内容首字符，按 v （左下角 “可视”字样），使用hjkl 挪移光标，选中区域

2. 复制： d

3. 粘贴：p 将内容粘贴至，光标后。 P向前粘。

删除一个字符：

命令模式下，光标选中待删除字符，按 x。

删除一个单词：

命令模式下，光标选中待删除单词首字符，按 dw。

#### 查找

| **按键** | **功能** |
| --- | --- |
| /字符串 | 查找指定字符串 |
| n | 寻找下一个 |
| \* | 匹配一个已有字符。向后找寻 |
| # | 匹配一个已有字符。向前找寻 |

#### 替换

单行替换：末行模式下， ：s/旧字符串/新字符串。 默认情况下，只替换当前行首个成功匹配的字符串。 全部替换 加 /g

**:s/append/add/g**

通篇替换：末行模式下， ：%s/旧字符串/新字符串。 默认情况下，只替换当前行首个成功匹配的字符串。 全部替换 加 /g

**:%s/append/add/g**

部分替换：末行模式下， :起始行号，终止行号s/旧字符串/新字符串/g。 默认情况下，只替换当前行首个成功匹配的字符串。 全部替换 加 /g

**:11，31s/append/add/g**

#### 其他命令

**分屏：**

**:sp 横向分屏**

**:vsp 纵向分屏**

**使用 ctrl + w + w 可以在各个分屏中切换光标进行编辑。**

**:wqall 可以一次保存退出所有分屏。**

**:!shell命令， 可在vim的末行中执行shell命令。如：！ls -l**

Vim配置

可以通过末行命令，修改vim显示文件的形式。如：

| **按键** | **功能** |
| --- | --- |
| :set nu | 显示行号 |
| :set nonu | 不显示行号 |

但这种方法的配置，只在当前vim文件中生效。文件关闭再次启动，之前的配置即失效，需重新配置。因此，我们通常将末行命令的配置写入vim的配置文件。使之每次打开文件时，都能按该配置加载文件。有两处配置位置：

1. 系统的配置目录位置： /etc/vim/vimrc

2. 用户的配置：~/.vimrc (注意：是隐藏文件。该文件默认不存在，需要手动创建。)

Linux 远程操作

SSH介绍

SSH为Secure Shell的缩写，由 IETF 的网络工作小组（Network Working Group）所制定；SSH 为建立在应用层和传输层基础上的安全协议。

SSH是目前较可靠，专为远程登录会话和其他网络服务提供安全性的协议。常用于远程登录，以及用户之间进行资料拷贝。

利用SSH协议可以有效防止远程管理过程中的信息泄露问题。SSH最初是 UNIX 系统上的一个程序，后来又迅速扩展到其他操作平台。SSH 在正确使用时可弥补网络中的漏洞。SSH 客户端适用于多种平台。几乎所有 UNIX 平台—包括 HP-UX、Linux、AIX、Solaris、Digital UNIX、Irix，以及其他平台，都可运行SSH。

使用SSH服务，需要安装相应的服务器和客户端。**客户端和服务器的关系：**如果，A机器想被B机器远程控制，那么，A机器需要安装SSH服务器，B机器需要安装SSH客户端。

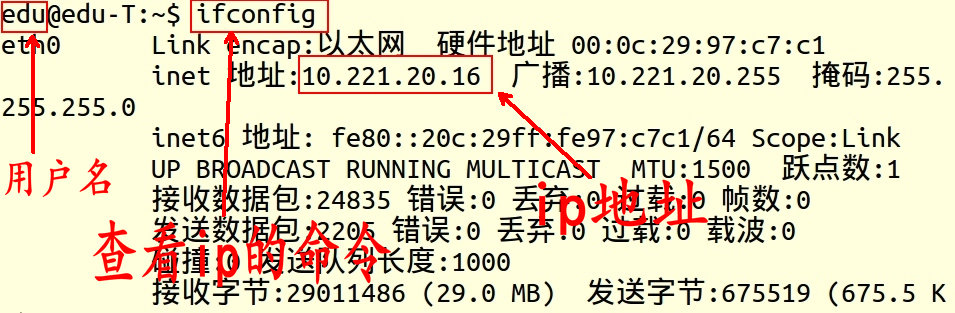
远程登录

操作命令：ssh -l username hostip

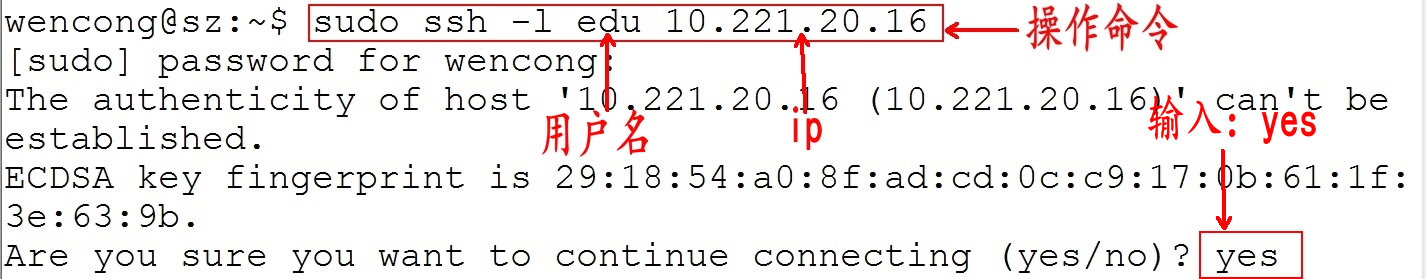
| **参数** | **含义** |
| --- | --- |
| -l | 选项， 是字母“l”，不是数字“1” |
| username | 被远程登录的用户名 |
| hostip | 被远程登录的ip地址 |

注意：远程登录的两台机器必须要能ping通。

首先，查看需要被远程机器的ip：

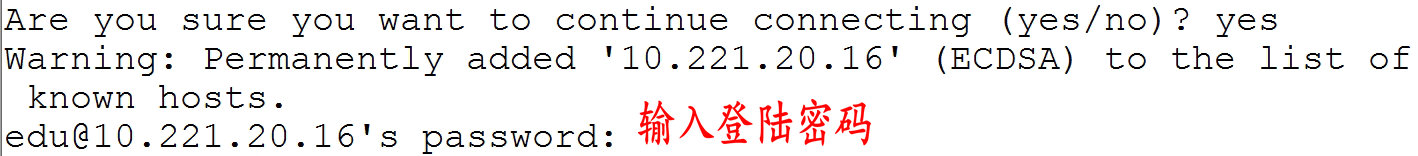


**远程登录(这里是用户 wencong ( A 机器 ) 远程登录 edu ( B 机器 ) )， 可以不用sudo ：**

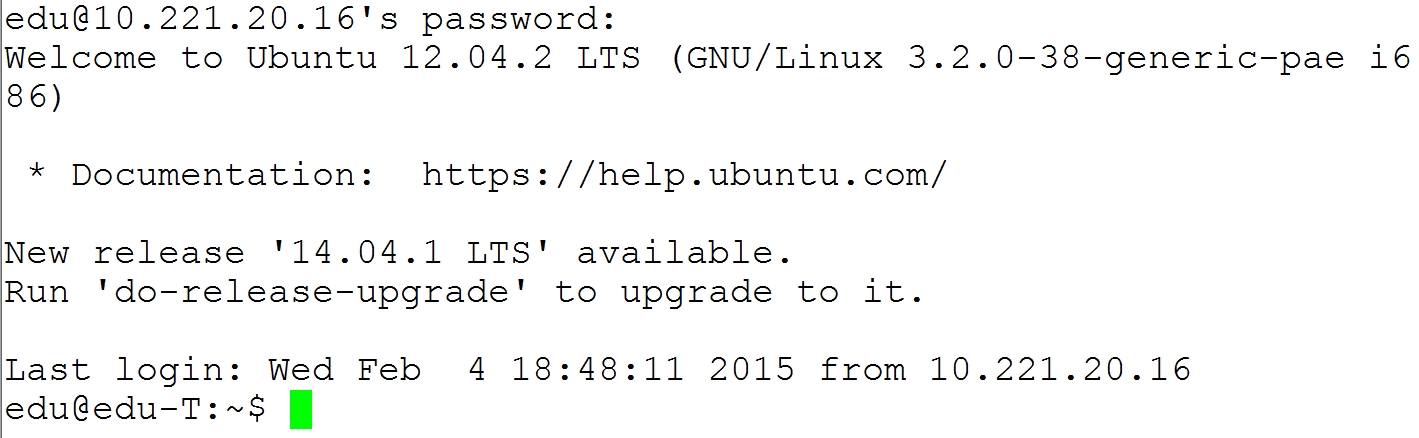


SSH 告知用户，这个主机不能识别，这时键入"yes"，SSH 就会将相关信息，写入"~/.ssh/know\_hosts" 中，再次访问，就不会有这些信息了。然后输入完口令,就可以登录到主机了。

提示输入登陆密码：



登陆成功：





远程传输文件

SSH 提供了一些命令和shell用来登录远程服务器。在默认情况下，不允许用户拷贝文件，但还是提供了一个“scp”命令。

本地文件、目录复制到远程（上传）：

scp -r 待上传文件路径 用户名@用户主机iP地址：存放位置（必须是绝对路径 ~ 不能用）

scp -r ./test.aaa itcast@192.168.6.199:/home/itcast/2go

远程文件、目录复制到本地（下载）：

scp -r 用户名@用户主机iP地址：待下载的文件存储位置（绝对路径） 存放到本地的位置。

scp -r [itcast@192.168.6.199:/home/itcast/test.tar.gz](mailto:itcast@192.168.6.199:/home/itcast/test.tar.gz) ./abc/

**Go语言高级编程**

**Golang标准库文档**： https://studygolang.com/pkgdoc

# 文件操作

## 字符串处理函数

字符串在开发中使用频率较高，操作文件时经常需要使用到字符串的拆分、判断等。可以借助Go标准库中的strings包快速达到处理字符串的目的。以下函数常常会被用到：

### 判断子串

**f**unc Contains(s, subStr string) bool

功能：判断字符串s是否包含子串substr

参1：s，表示待判断的字符串

参2：subStr，表示子串，

返回值：布尔类型true or false

示例代码：

fmt.Println(strings.Contains("seafood", "foo"))

fmt.Println(strings.Contains("seafood", "bar"))

fmt.Println(strings.Contains("seafood", ""))

fmt.Println(strings.Contains("", ""))

### 字符串拼接

func Join(a [][string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string), sep [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string)) [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string)

功能：将一系列字符串连接为一个字符串，之间用sep来分隔。

参1：a，表示待拼接字符串切片

参2：sep，表连接使用的字符串，

返回值：使用sep拼接完成的字符串。

示例代码：

s := []string{"foo", "bar", "baz"}

fmt.Println(strings.Join(s, ", "))

### 字符串去首尾字符

func Trim(s [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string), cutset [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string)) [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string)

功能：将s前后端所有cutset字符串去除。返回处理后的新字符串

参1：s，表示待处理的字符串

参2：cutset，表连待去除的字符串，

返回值：处理完成的新字符串。

通常使用该函数去除字符串中包含的多余空格。

示例代码：

fmt.Printf("[%q]\n", strings.Trim(" !!! Achtung! Achtung! !!! ", "! "))

fmt.Printf("[%q]\n", strings.Trim(" hello ", " "))

//运行结果:

["Achtung! Achtung"]

["hello"]

### 字符串替换

func Replace(s, old, new [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string), n [int](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "int)) [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string)

功能：将s中的old子串替换为new，替换n处。返回替换后的新字符串

参1：s，表示待处理的字符串

参2：old，表s中待被替换的字符串

参3：new，表s中替换old的新字符串

参4：替换的次数。n取值 -1时，全部替换。

返回值：替换完成的新字符串。

示例代码：

fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))

fmt.Println(strings.Replace("oink oink oink", "in", "moo", -1))

//运行结果:

oinky oinky oink

omook omook omook

### 字符串分割

**func** **Split**(s, sep string) []string

功能：把s字符串按照sep分割，返回slice

参1：s，表示待拆分的字符串

参2：sep，表示分割符，该参数为string 类型

返回值：切片，存储拆分好的子串

示例代码：

    fmt.Printf("%q\n", strings.Split("a,b,c", ","))

    fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))

    fmt.Printf("%q\n", strings.Split(" xyz ", ""))

    fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))

    //运行结果:

    //["a" "b" "c"]

    //["" "man " "plan " "canal panama"]

    //[" " "x" "y" "z" " "]

    //[""]

### 按空格拆分字符串

**func** **Fields**(s string) []string

功能：去除s字符串的空格符，并且按照空格分割，返回slice

参1：s，表示待拆分的字符串

返回值：切片，存储拆分好的子串

示例代码：

    fmt.Printf("Fields are: %q", strings.Fields(" foo bar baz "))

    //运行结果:Fields are: ["foo" "bar" "baz"]

### 判断字符串后缀

**func HasSuffix**(s, suffix string) **bool**

功能：判断s字符串是否有后缀子串suffix

参1：s，表示待判定字符串

参2：suffix，表示前缀子串

返回值：true or false

示例代码：

    fmt.Printf("%v\n", strings.HasSuffix("World Cup.png", ".png"))

    //运行结果:true

### 判断字符串前缀

**func HasPrefix**(s, prefix string) **bool**

功能：判断s字符串是否有前缀子串suffix

参1：s，表示待判定字符串

参2：prefix，表示前缀子串

返回值：true or false

示例代码：

    fmt.Printf("%v\n", strings.HasPrefix("World Cup.png", "world"))

    //运行结果:false

strings包实现了用于操作字符及字符串的很多函数。同学们可以以上面几个函数为例，自行学习帮助文档陈列的其他常用字符串处理函数，在开发中进一步提高工作效率。

## 文件操作常用API

### 建立与打开文件

新建文件可以通过如下两个方法：

**func** **Create**(name string) (file \*File, err Error)

根据提供的文件名创建新的文件，返回一个文件对象，默认权限是0666的文件，返回的文件对象是可读写的。

通过如下两个方法来打开文件：

**func** **Open**(name string) (file \*File, err Error)

Open()是以**只读权限**打开文件名为name的文件，得到的文件指针file，只能用来对文件进行“读”操作。如果我们有“写”文件的需求，就需要借助Openfile函数来打开了。

**func** **OpenFile**(name string, flag int, perm uint32) (file \*File, err Error)

OpenFile()可以选择打开name文件的读写权限。这个函数有三个默认参数：

参1：name，表示打开文件的路径。可使用相对路径 或 绝对路径

参2：flg，表示读写模式，常见的模式有：

**O\_RDONLY**(只读模式), **O\_WRONLY**(只写模式), **O\_RDWR**(可读可写模式)。

参3：perm，表示打开权限。来源于Linux系统调用中的open函数，参2为：**O\_CREATE**时，可创建新文件。

权限取值范围（0-7），表示如下：

0：没有任何权限

1：执行权限(如果是可执行文件，是可以运行的)

2：写权限

3: 写权限与执行权限

4：读权限

5: 读权限与执行权限

**6: 读权限与写权限**

7: 读权限，写权限，执行权限

关闭文件函数：

**func** (f \*[File](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "File)) **Close()** [error](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "error)

### 写文件

写入字符串到文件：

**func** (file \*File) **WriteString**(s string) (ret int, err Error)

获取文件读写偏移量：

**func** (f \*File) **Seek**(offset int64, whence int) (ret int64, err error)

**参1**： 偏移量。 正：向文件尾偏。 负：向文件头偏

**参2**： 偏移起始位置：

io.**SeekStart**: 文件起始位置

io.**SeekCurrent**： 文件当前位置

io.**SeekEnd**: 文件结尾位置

**返回值**：表示从文件起始位置，到当前文件读写指针位置的偏移量。

在指定位置写入数据：

**func** (file \*File) **WriteAt**(b []byte, off int64) (n int, err Error)

**参1**：待写入的数据内容

**参2**：偏移量（通常是Seek函数返回值）

按字节写数据到文件：

**func** (file \*File) **Write**(b []byte) (n int, err Error)

### 读文件

**按行读文件**：可以通过两步来实现。

首先，创建一个带有缓冲区的Reader，使用bufio包下的 NewReader() 函数来实现。

**func** NewReader(rd io.Reader) \*Reader

然后，从Reader 自带的缓冲区中，按指定的“分隔符（delim）”获取数据。我们通常使用ReadBytes(‘\n’) 来有效的提取一行数据。

**func** (b \*Reader) ReadBytes(delim byte) ([]byte, error)

读到的数据成功保存在返回的 []byte 中。可以循环按行读取整个文件。

**按字节读文件：**

**func** (file \*File) Read(b []byte) (n int, err Error)

读取数据到b中

**func** (file \*File) ReadAt(b []byte, off int64) (n int, err Error)

从off开始读取数据到b中

### 删除文件

**func** Remove(name string) Error

调用该函数就可以删除文件名为name的文件

### 练习：大文件拷贝

示例代码：

**package** main

**import** (

    "fmt"

    "io"

    "os"

)

**func** main() {

    args := os.Args //获取命令行参数， 并判断输入是否合法

**if** args == nil || len(args) != 3 {

        fmt.Println("useage : xxx srcFile dstFile")

**return**

    }

    srcPath := args[1] //获取参数1

    dstPath := args[2] //获取参数2

    fmt.Printf("srcPath = %s, dstPath = %s\n", srcPath, dstPath)

**if** srcPath == dstPath {

        fmt.Println("error：源文件名 与 目的文件名雷同")

**return**

    }

    srcFile, err1 := os.Open(srcPath) // 打开源文件

**if** err1 != nil {

        fmt.Println(err1)

**return**

    }

    dstFile, err2 := os.Create(dstPath) //创建目标文件

**if** err2 != nil {

        fmt.Println(err2)

**return**

    }

    buf := make([]byte, 1024) //切片缓冲区

**for** {

        //从源文件读取内容，n为读取文件内容的长度

        n, err := srcFile.Read(buf)

**if** err != nil && err != io.EOF {

            fmt.Println(err)

**break**

        }

**if** n == 0 {

            fmt.Println("文件处理完毕")

**break**

        }

        //切片截取

        tmp := buf[:n]

        //把读取的内容写入到目的文件

        dstFile.Write(tmp)

    }

    //关闭文件

    srcFile.Close()

    dstFile.Close()

}

## 目录操作常用API

我们读写的文件一般存放于目录中。因此，有时需要指定到某一个目录下，根据目录存储的状况再进行文件的特定操作。接下来我们看看目录的基本操作方法。

### 打开目录

打开目录我们也使用 OpenFile 函数，但要指定不同的参数来通知系统，要打开的是一个目录文件。

**func** OpenFile(name string, flag int, perm FileMode) (\*File, error)

参数1：name，表示要打开的目录名称。使用绝对路径较多

参数2：flg，表示打开文件的读写模式：

**O\_RDONLY只读模式**

参数3：perm，表示打开权限。但对于**目录**来说略有不同。通常传os.**ModeDir。**

返回值：由于是操作目录，所以file是指向目录的文件指针。error中保存错误信息。

### 读目录内容

这与读文件有所不同。目录中存放的是文件名和子目录名。所以使用Readdir函数来完成。

**func** (f \*File) Readdir(n int) ([]FileInfo, error)

参数：n, 表读取目录的成员个数。**通常传-1**,表读取目录所有文件对象。

返回值：FileInfo类型的切片。其内部保存了文件名。error中保存错误信息。

**type** FileInfo **interface** {  
 **Name()** string *// base name of the file* **Size()** int64 *// length in bytes for regular files; system-dependent for others* Mode() FileMode *// file mode bits* ModTime() time.Time *// modification time* **IsDir()** bool *// abbreviation for Mode().IsDir()* Sys() **interface**{} *// underlying data source (can return nil)*}

得到 FileInfo类型切片后，我们可以range遍历切片元素，使用.Name()获取文件名。使用.Size()获取文件大小，使用.IsDir()判断文件是目录还是非目录文件。

如：我们可以提示用户提供一个目录位置，打开该目录，查看目录下的所有成员，并判别他们是文件还是目录。

示例代码：

**func** main() {  
 fmt.Println(**"请输入要找寻的目录："**)  
 **var** path string  
 fmt.Scan(&path) // 获取用户指定的目录名  
  
 dir, \_ := os.OpenFile(path, os.**O\_RDONLY**, os.**ModeDir**) // 只读打开该目录  
  
 names, \_ := dir.Readdir(-1) // 读取当前目录下所有的文件名和目录名，存入names切片  
  
 **for** \_, name := **range** names { // 遍历切片，获取文件/目录名  
 **if** !name.IsDir() {  
 fmt.Println(name.Name(), **"是一个文件"**)  
 } **else** {  
 fmt.Println(name.Name(), **"是一个目录"**)  
 }  
 }  
}

### 其他目录操作API

其实，目录也可以看成“文件”。我们通常读写的文件内容是可见的ASCII码。目录文件的内容就是文件名和目录名，称之为目录项。我们读写目录文件，实质上就是在读写目录项。

目录操作还有其他的一系列API，这里简单罗列几个，大家可酌情自行学习。

将当前工作目录修改为dir指定的目录：

func Chdir(dir [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string)) [error](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "error)

返回当前工作目录的绝对路径：

func Getwd() (dir [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string), err [error](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "error))

使用指定的权限和名称创建一个目录：

func Mkdir(name [string](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "string), perm [FileMode](https://studygolang.com/static/pkgdoc/pkg/os.htm" \l "FileMode)) [error](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "error)

获取更多文件、目录操作API可查看Go标库文档： https://studygolang.com/pkgdoc

## 文件/目录操作练习：

### 初级练习：

**指定目录检索特定文件：**

从用户给出的目录中，找出所有的 .jpg 文件。

### 中级练习：

**指定目录拷贝特定文件：**

从用户给出的目录中，拷贝 .mp3文件到指定目录中。

### 高级练习：

**统计指定单词出现次数：**

从用户指定的一级目录下，统计所有.txt文件中，“Love”这个单词 出现的次数。

# Go并发编程

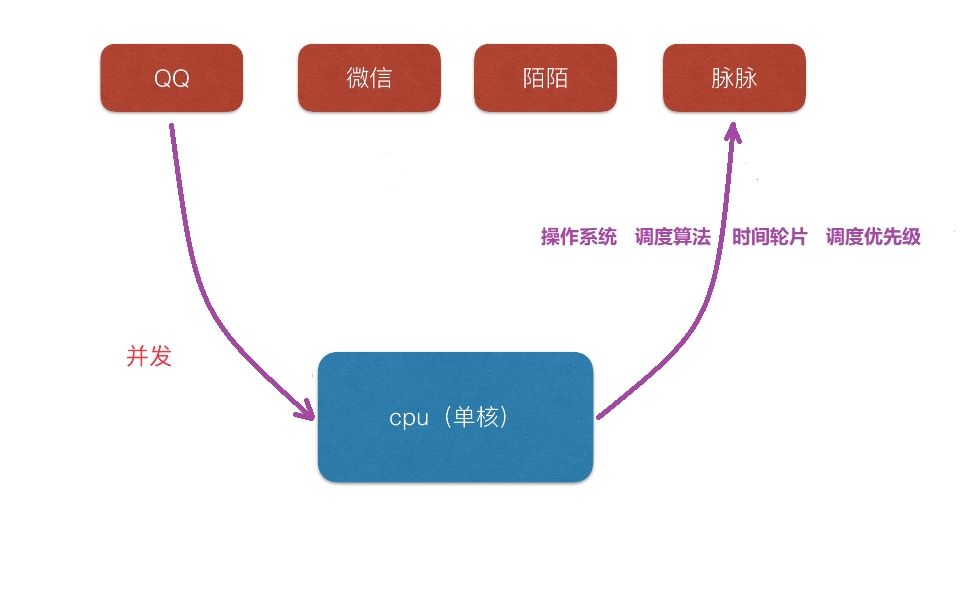
## 概述

简而言之，所谓并发编程是指在一台处理器上“同时”处理多个任务。

随着硬件的发展，并发程序变得越来越重要。Web服务器会一次处理成千上万的请求。平板电脑和手机app在渲染用户画面同时还会后台执行各种计算任务和网络请求。即使是传统的批处理问题--读取数据，计算，写输出--现在也会用并发来隐藏掉I/O的操作延迟以充分利用现代计算机设备的多个核心。计算机的性能每年都在以非线性的速度增长。

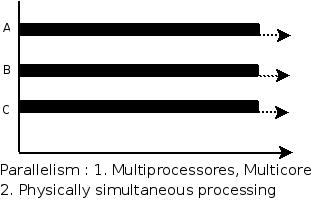
宏观的并发是指在一段时间内，有多个程序在同时运行。

并发在微观上，是指在同一时刻只能有一条指令执行，但多个程序指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个程序快速交替的执行。

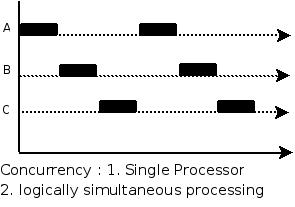


### 并行和并发

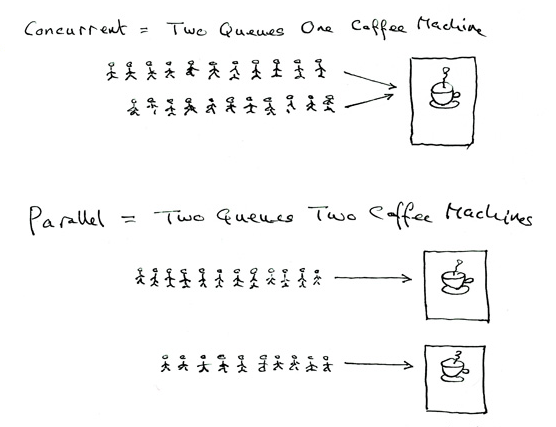
**并行(parallel)：**指在同一时刻，有多条指令在多个处理器上同时执行。



**并发(concurrency)：**指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，通过cpu**时间片轮转**使多个进程快速交替的执行。



大师曾以咖啡机的例子来解释并行和并发的区别。



* **并行**是两个队列**同时**使用**两台**咖啡机 （真正的多任务）
* **并发**是两个队列**交替**使用**一台**咖啡机 （ 假 的多任务）

## 常见并发编程技术

### 进程并发

程序和进程

程序，是指编译好的二进制文件，在磁盘上，不占用系统资源(内存、打开的文件、设备、锁....)

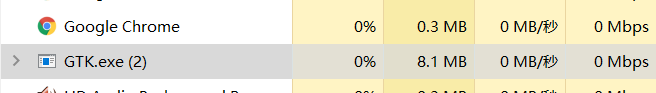
进程，是一个抽象的概念，与操作系统原理联系紧密。进程是活跃的程序，占用系统资源。在内存中执行。(程序运行起来，产生一个进程)

程序 → 剧本(纸) 进程 → 戏 (舞台、演员、灯光、道具...)

同一个剧本可以在多个舞台同时上演。同样，同一个程序也可以加载为不同的进程(彼此之间互不影响)

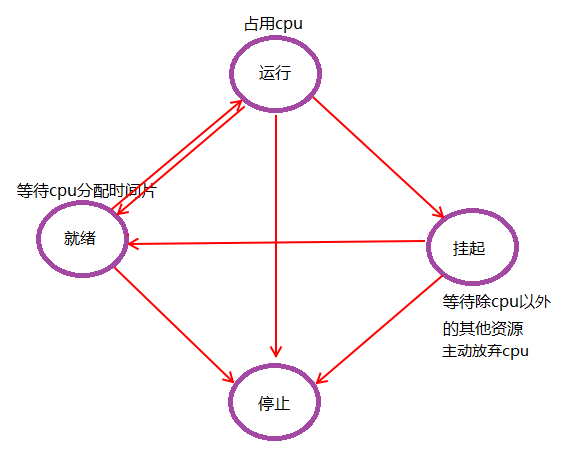
如：同时开两个终端。各自都有一个bash但彼此ID不同。

在windows系统下，通过查看“任务管理器”，可以查看相应的进程。包括我们在基础班写的“飞机大战”等程序，运行起来后也可以在“任务管理器”中查看到。运行起来的程序就是一个进程。如下图所示：



进程状态

进程基本的状态有5种。分别为初始态，就绪态，[运行态](http://baike.baidu.com/subview/1730379/1730379.htm" \t "_blank)，挂起态与终止态。其中初始态为进程准备阶段，常与就绪态结合来看。



进程并发

在使用进程 实现并发时会出现什么问题呢？

1：系统开销比较大，占用资源比较多，开启进程数量比较少。

2：在unix/linux系统下，还会产生“孤儿进程”和“僵尸进程”。

在操作系统运行过程中，可以产生很多的进程。在unix/linux系统中，正常情况下，子进程是通过父进程fork创建的，子进程再创建新的进程。

并且父进程永远无法预测子进程到底什么时候结束。 当一个进程完成它的工作终止之后，它的父进程需要调用系统调用取得子进程的终止状态。

**孤儿进程**

孤儿进程: 父进程先于子进程结束，则子进程成为孤儿进程，子进程的父进程成为init进程，称为init进程领养孤儿进程。

**僵尸进程**

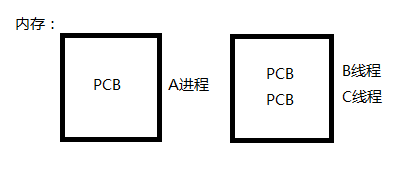
僵尸进程: 进程终止，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。

Windows下的进程和Linux下的进程是不一样的，它比较懒惰，从来不执行任何东西，只是为线程提供执行环境。然后由线程负责执行包含在进程的地址空间中的代码。当创建一个进程的时候，操作系统会自动创建这个进程的第一个线程，成为主线程。

### 线程并发

什么是线程

LWP：light weight process 轻量级的进程，本质仍是进程 (Linux下)

 进程：独立地址空间，拥有PCB

线程：有独立的PCB，但没有独立的地址空间(共享)

区别：在于是否共享地址空间。独居(进程)；合租(线程)。

线程：最小的执行单位------------cpu时间轮片分配单位

进程：最小分配资源单位，可看成是只有一个线程的进程。-------------进程地址空间(0-4G)

一个进程内可以有多个线程，当一个进程创建一个线程时，原来的进程也就沦落成线程了，一个进程地址空间内有两个pcb

内核栈：当时钟中断时，保存线程运行状态

Windows系统下，可以直接忽略进程的概念，只谈线程。因为线程是最小的执行单位，是被系统独立调度和分派的基本单位。而进程只是给线程提供执行环境。

线程同步

同步即协同步调，按预定的先后次序运行。

**线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其它线程为保证数据一致性，不能调用该功能。**

举例1： 银行存款 5000。柜台，折：取3000；提款机，卡：取 3000。剩余：2000

举例2： 内存中100字节，线程T1欲填入全1， 线程T2欲填入全0。但如果T1执行了50个字节失去cpu，T2执行，会将T1写过的内容覆盖。当T1再次获得cpu继续 从失去cpu的位置向后写入1，当执行结束，内存中的100字节，既不是全1，也不是全0。

产生的现象叫做“与时间有关的错误”(time related)。为了避免这种数据混乱，线程需要同步。

“同步”的目的，是为了避免数据混乱，解决与时间有关的错误。实际上，不仅线程间需要同步，进程间、信号间等等都需要同步机制。

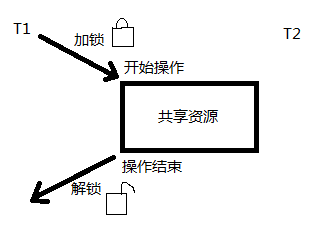
因此，**所有“多个控制流，共同操作一个共享资源”的情况，都需要同步。**

锁的应用

##### 互斥量mutex

Linux中提供一把互斥锁mutex（也称之为互斥量）。每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束解锁。

资源还是共享的，线程间也还是竞争的，但通过“锁”就将资源的访问变成互斥操作，而后与时间有关的错误也不会再产生了。



但，应注意：同一时刻，只能有一个线程持有该锁。

当A线程对某个全局变量加锁访问，B在访问前尝试加锁，拿不到锁，B阻塞。C线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱。

所以，互斥锁实质上是操作系统提供的一把“建议锁”（又称“协同锁”），建议程序中有多线程访问共享资源的时候使用该机制。但，并没有强制限定。

因此，即使有了mutex，如果有线程不按规则来访问数据，依然会造成数据混乱。

##### 读写锁

与互斥量类似，但读写锁允许更高的并行性。其特性为：**写独占，读共享。**

* **读写锁状态：**

特别强调：读写锁**只有一把**，但其具备两种状态：

1. 读模式下加锁状态 (读锁)

2. 写模式下加锁状态 (写锁)

* **读写锁特性：**

1. 读写锁是“写模式加锁”时， 解锁前，所有对该锁加锁的线程都会被阻塞。
2. 读写锁是“读模式加锁”时， 如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞。
3. 读写锁是“读模式加锁”时， 既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。**读锁、写锁并行阻塞，写锁优先级高**

读写锁也叫共享-独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享。**

读写锁非常适合于对数据结构读的次数远大于写的情况。

### 协程并发

协程：coroutine。也叫轻量级线程。

与传统的系统级线程和进程相比，协程最大的优势在于“轻量级”。可以轻松创建上万个而不会导致系统资源衰竭。而线程和进程通常很难超过1万个。这也是协程别称“轻量级线程”的原因。

一个线程中可以有任意多个协程，但某一时刻只能有一个协程在运行，**多个协程共享该线程分配到的计算机资源。**

多数语言在语法层面并不直接支持协程，而是通过库的方式支持，但用库的方式支持的功能也并不完整，比如仅仅提供协程的创建、销毁与切换等能力。如果在这样的轻量级线程中调用一个同步 IO 操作，比如网络通信、本地文件读写，都会阻塞其他的并发执行轻量级线程，从而无法真正达到轻量级线程本身期望达到的目标。

在协程中，调用一个任务就像调用一个函数一样，消耗的系统资源最少！但能达到进程、线程并发相同的效果。

在一次并发任务中，进程、线程、协程均可以实现。从系统资源消耗的角度出发来看，进程相当多，线程次之，协程最少。

进程的优点：

1. 独立的进程地址空间，更安全

线程的优点：

1. 共享进程地址空间，节省内存空间，数据传输通信方便

### 协程的优点

1. 比线程更轻量级，节省了函数栈针调度、内核空间与用户空间切换系统资源的开销
2. 更高的利用cpu资源，可以在线程阻塞期间(线程空闲时间)运行其他指令来提高cpu利用率

### Go并发

Go 在语言级别支持协程，叫goroutine。Go 语言标准库提供的所有系统调用操作（包括所有同步IO操作），都会出让CPU给其他goroutine。这让轻量级线程的切换管理不依赖于系统的线程和进程，也不需要依赖于CPU的核心数量。

有人把Go比作21世纪的C语言。第一是因为Go语言设计简单，第二，21世纪最重要的就是并行程序设计，而Go从语言层面就支持并行。同时，并发程序的内存管理有时候是非常复杂的，而Go语言提供了自动垃圾回收机制。

Go语言为并发编程而内置的上层API基于顺序通信进程模型CSP(communicating sequential processes)。这就意味着显式锁都是可以避免的，因为Go通过相对安全的通道发送和接受数据以实现同步，这大大地简化了并发程序的编写。

Go语言中的并发程序主要使用两种手段来实现。goroutine和channel。

## Goroutine

### 什么是Goroutine

goroutine是Go语言并行设计的核心，有人称之为go程。 Goroutine从量级上看很像协程，它比线程更小，十几个goroutine可能体现在底层就是五六个线程，Go语言内部帮你实现了这些goroutine之间的内存共享。执行goroutine只需极少的栈内存(大概是4~5KB)，当然会根据相应的数据伸缩。也正因为如此，可同时运行成千上万个并发任务。goroutine比thread更易用、更高效、更轻便。

一般情况下，一个普通计算机跑几十个线程就有点负载过大了，但是同样的机器却可以轻松地让成百上千个goroutine进行资源竞争。

### Goroutine的创建

只需在函数调⽤语句前添加 **go** 关键字，就可创建并发执⾏单元。开发⼈员无需了解任何执⾏细节，调度器会自动将其安排到合适的系统线程上执行。

在并发编程中，我们通常想将一个过程切分成几块，然后让每个goroutine各自负责一块工作，当一个程序启动时，主函数在一个单独的goroutine中运行，我们叫它main goroutine。新的goroutine会用go语句来创建。而go语言的并发设计，让我们很轻松就可以达成这一目的。

示例代码：

**package** main

**import** (

    "fmt"

    "time"

)

**func** newTask() {

    i := 0

**for** {

        i++

        fmt.Printf("new goroutine: i = %d\n", i)

        time.Sleep(1\*time.Second) //延时1s

    }

}

**func** main() {

    //创建一个 goroutine，启动另外一个任务

**go** newTask()

    i := 0

    //main goroutine 循环打印

**for** {

        i++

        fmt.Printf("main goroutine: i = %d\n", i)

        time.Sleep(1 \* time.Second) //延时1s

    }

}

程序运行结果：



### Goroutine特性

**主goroutine退出后，其它的工作goroutine也会自动退出：**

**package main**

**import (**

**"fmt"**

**"time"**

**)**

**func** newTask() {

    i := 0

**for** {

        i++

        fmt.Printf("new goroutine: i = %d\n", i)

        time.Sleep(1 \* time.Second) //延时1s

    }

}

**func** main() {

    //创建一个 goroutine，启动另外一个任务

**go** newTask()

    fmt.Println("main goroutine exit")

}

程序运行结果：

2018-01-15_100253

### runtime包

Gosched

runtime.Gosched() （go schedule）**用于让出CPU时间片**，让出当前goroutine的执行权限，调度器安排其他等待的任务运行，并在下次再获得cpu时间轮片的时候，从该出让cpu的位置恢复执行。

有点像跑接力赛，A跑了一会碰到代码runtime.Gosched() 就把接力棒交给B了，A歇着了，B继续跑。

示例代码：

**package main**

**import (**

**"fmt"**

**"runtime"**

**)**

**func** main() {

    //创建一个goroutine

**go** **func**(s string) {

**for** i := 0; i < 2; i++ {

            fmt.Println(s)

        }

    }("world")

**for** i := 0; i < 2; i++ {

        runtime.Gosched() //import "runtime" 包

        /\*

            屏蔽runtime.Gosched()运行结果如下：

                hello

                hello

            没有runtime.Gosched()运行结果如下：

                world

                world

                hello

                hello

        \*/

        fmt.Println("hello")

    }

}

以上程序的执行过程如下：

主go程进入main()函数，进行代码的执行。当执行到go func()匿名函数时，创建一个新的go程，开始执行匿名函数中的代码，主go程继续向下执行，执行到runtime.Gosched()时会暂停向下执行，直到其它go程执行完后，再回到该位置，主go程继续向下执行。

Goexit

调用 runtime.Goexit() 将立即终止当前 **goroutine 执⾏，调度器确保所有已注册 defer 延迟调用被执行。**

示例代码：

**package main**

**import (**

**"fmt"**

**"runtime"**

**)**

**func** main() {

**go** **func**() {

**defer** fmt.Println("A.defer")

**func**() {

**defer** fmt.Println("B.defer")

            runtime.Goexit() // 终止当前 goroutine, import "runtime"

            fmt.Println("B") // 不会执行

        }()

        fmt.Println("A") // 不会执行

    }() //不要忘记()

    //死循环，目的不让主goroutine结束

**for** {

    }

}

程序运行结果：

2018-01-15_104908

GOMAXPROCS

调用 runtime.GOMAXPROCS() 用来设置可以并行计算的CPU核数的最大值，并返回之前的值。

示例代码：

**package main**

**import (**

**"fmt"**

**)**

**func** main() {

//n := runtime.GOMAXPROCS(1) // 第一次 测试

//打印结果：111111111111111111110000000000000000000011111...

n := runtime.GOMAXPROCS(2)     // 第二次 测试

//打印结果：010101010101010101011001100101011010010100110...

    fmt.Printf("n = %d\n", n)

**for** {

**go** fmt.Print(0)

        fmt.Print(1)

    }

}

在第一次执行runtime.GOMAXPROCS(1) 时，最多同时只能有一个goroutine被执行。所以会打印很多1。过了一段时间后，GO调度器会将其置为休眠，并唤醒另一个goroutine，这时候就开始打印很多0了，在打印的时候，goroutine是被调度到操作系统线程上的。

在第二次执行runtime.GOMAXPROCS(2) 时， 我们使用了两个CPU，所以两个goroutine可以一起被执行，以同样的频率交替打印0和1。

总结：

1. return：返回当前**函数调用，执行已成功注册的defer**
2. Goexit: 结束当前**go程，执行已成功注册的defer，不能在主go程中使用（不能执行主函数的右花括号，无法释放进程地址空间，报错）**
3. Os.Exit(int): 结束当前**进程，不会执行defer注册成功的执行**

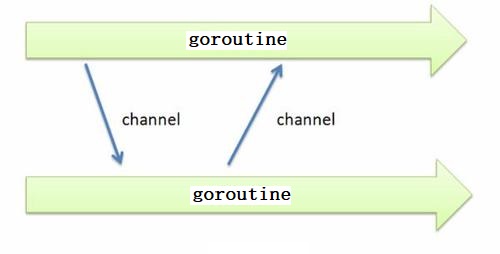
## channel

channel是Go语言中的一个**核心类型**，可以把它看成管道。并发核心单元通过它就可以发送或者接收数据进行通讯，这在一定程度上又进一步降低了编程的难度。

channel是一个**数据类型**，主要用来解决go程的同步问题以及go程之间数据共享（数据传递）的问题。

goroutine运行在相同的地址空间，因此访问共享内存必须做好同步。goroutine 奉行**通过通信来共享内存，而不是共享内存来通信。**

引⽤类型 channel可用于多个 goroutine 通讯。其内部实现了同步，确保并发安全。



### 定义channel变量

和map类似，channel也一个对应make创建的底层数据结构的**引用**。

定义一个channel时，也需要定义发送到channel的值的类型。channel可以使用内置的make()函数来创建：

**chan**是创建channel所需使用的关键字。Type 代表指定channel收发数据的类型。

    make(**chan** Type) //等价于make(chan Type, 0)

    make(**chan** Type, capacity)

当我们复制一个channel或用于函数参数传递时，我们只是拷贝了一个channel引用，因此调用者和被调用者将引用同一个channel对象。和其它的引用类型一样，**channel的零值也是nil**。

当 参数**capacity= 0 时，channel 是无缓冲阻塞读写的**；**当capacity > 0 时，channel 有缓冲、是非阻塞的，直到写满 capacity个元素才阻塞写入。**

channel非常像生活中的管道，一边可以存放东西，另一边可以取出东西。channel通过操作符 <- 来接收和发送数据，发送和接收数据语法：

    channel <- value      //发送value到channel

    <-channel             //接收并将其丢弃

    x := <-channel        //从channel中接收数据，并赋值给x

    x, ok := <-channel    //功能同上，同时检查通道是否已关闭或者是否为空

默认情况下，channel接收和发送数据都是阻塞的，除非另一端已经准备好，这样就使得goroutine同步变的更加的简单，而**不需要显式的lock。**

示例代码：

**package main**

**import (**

**"fmt"**

**)**

**func** main() {

    c := make(**chan** int)

**go** **func**() {

**defer** fmt.Println("子go程结束")

        fmt.Println("子go程正在运行……")

        c <- 666 //666发送到c

    }()

    num := <-c //从c中接收数据，并赋值给num

    fmt.Println("num = ", num)

    fmt.Println("main go程结束")

}

程序运行结果：



### 无缓冲的channel

无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何数据值的通道。

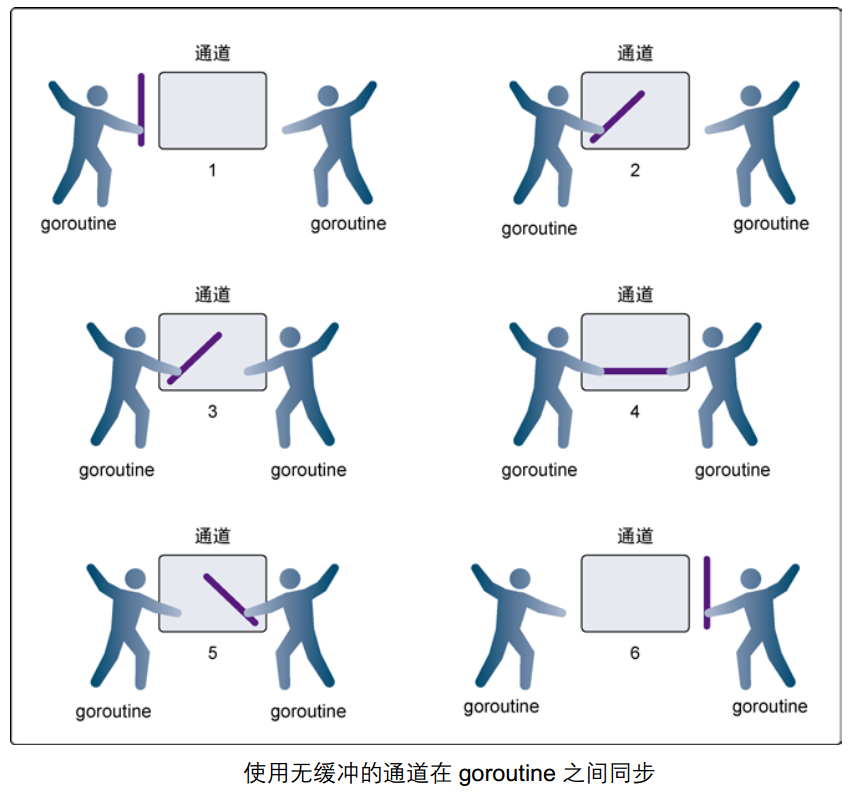
这种类型的通道要求发送goroutine和接收goroutine同时准备好，才能完成发送和接收操作。否则，通道会导致先执行发送或接收操作的 goroutine 阻塞等待。

这种对通道进行发送和接收的交互行为本身就是同步的。其中任意一个操作都无法离开另一个操作单独存在。

**阻塞：**由于某种原因数据没有到达，当前go程（线程）持续处于等待状态，直到条件满足，才解除阻塞。

**同步：**在两个或多个go程（线程）间，保持数据内容一致性的机制。

下图展示两个 goroutine 如何利用无缓冲的通道来共享一个值：



* 在第 1 步，两个 goroutine 都到达通道，但哪个都没有开始执行发送或者接收。
* 在第 2 步，左侧的 goroutine 将它的手伸进了通道，这模拟了向通道发送数据的行为。这时，这个 goroutine 会在通道中被锁住，直到交换完成。
* 在第 3 步，右侧的 goroutine 将它的手放入通道，这模拟了从通道里接收数据。这个 goroutine 一样也会在通道中被锁住，直到交换完成。
* 在第 4 步和第 5 步，进行交换，并最终，在第 6 步，两个 goroutine 都将它们的手从通道里拿出来，这模拟了被锁住的 goroutine 得到释放。两个 goroutine 现在都可以去做其他事情了。

无缓冲的channel创建格式：

    make(**chan** Type) //等价于make(chan Type, 0)

如果没有指定缓冲区容量，那么该通道就是同步的，因此会阻塞到发送者准备好发送和接收者准备好接收。

示例代码：

**package main**

**import (**

**"fmt"**

**"time"**

**)**

**func** main() {

    c := make(**chan** int, 0) //创建无缓冲的通道 c

    //内置函数 len 返回未被读取的缓冲元素数量，cap 返回缓冲区大小

    fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

**go** **func**() {

**defer** fmt.Println("子go程结束")

**for** i := 0; i < 3; i++ {

            c <- i

            fmt.Printf("子go程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i, len(c), cap(c))

        }

    }()

    time.Sleep(2 \* time.Second) //延时2s

**for** i := 0; i < 3; i++ {

        num := <-c //从c中接收数据，并赋值给num

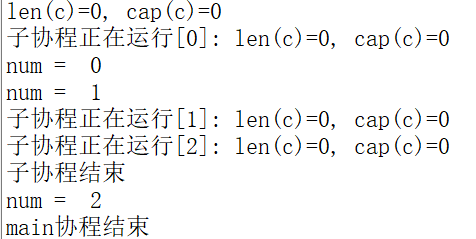
        fmt.Println("num = ", num)

    }

    fmt.Println("main进程结束")

}

程序运行结果：



### 有缓冲的channel

有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个数据值的通道。

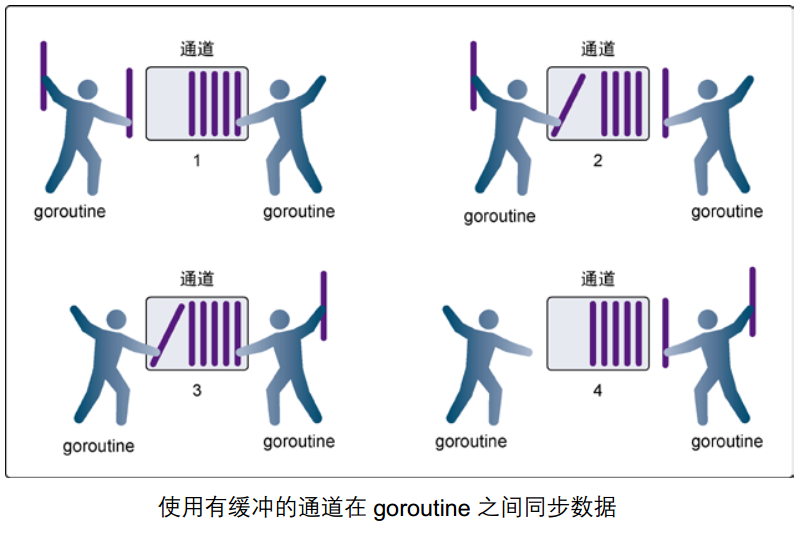
这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收。通道会阻塞发送和接收动作的条件也不同。

只有通道中没有要接收的值时，接收动作才会阻塞。

只有通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：无缓冲的通道保证进行发送和接收的 goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。

示例图如下：



* 在第 1 步，右侧的 goroutine 正在从通道接收一个值。
* 在第 2 步，右侧的这个 goroutine独立完成了接收值的动作，而左侧的 goroutine 正在发送一个新值到通道里。
* 在第 3 步，左侧的goroutine 还在向通道发送新值，而右侧的 goroutine 正在从通道接收另外一个值。这个步骤里的两个操作既不是同步的，也不会互相阻塞。
* 最后，在第 4 步，所有的发送和接收都完成，而通道里还有几个值，也有一些空间可以存更多的值。

有缓冲的channel创建格式：

    make(**chan** Type, capacity)

如果给定了一个缓冲区容量，通道就是异步的。只要缓冲区有未使用空间用于发送数据，或还包含可以接收的数据，那么其通信就会无阻塞地进行。

借助函数 **len(ch)** 求取缓冲区中剩余元素个数， **cap(ch)** 求取缓冲区元素容量大小。

示例代码：

**func** main() {

    c := make(**chan** int, 3) //带缓冲的通道

    //内置函数 len 返回未被读取的缓冲元素数量， cap 返回缓冲区大小

    fmt.Printf("len(c)=%d, cap(c)=%d\n", len(c), cap(c))

**go** **func**() {

**defer** fmt.Println("子go程结束")

**for** i := 0; i < 3; i++ {

            c <- i

            fmt.Printf("子go程正在运行[%d]: len(c)=%d, cap(c)=%d\n", i, len(c), cap(c))

        }

    }()

    time.Sleep(2 \* time.Second) //延时2s

**for** i := 0; i < 3; i++ {

        num := <-c //从c中接收数据，并赋值给num

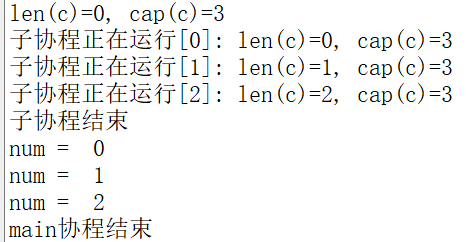
        fmt.Println("num = ", num)

    }

    fmt.Println("main进程结束")

}

程序运行结果：



### 关闭channel

如果发送者知道，没有更多的值需要发送到channel的话，那么让接收者也能及时知道没有多余的值可接收将是有用的，因为接收者可以停止不必要的接收等待。这可以通过内置的close函数来关闭channel实现。

示例代码：

**package main**

**import (**

**"fmt"**

**)**

**func** main() {

    c := make(**chan** int)

**go** **func**() {

**for** i := 0; i < 5; i++ {

            c <- i

        }

        //把 close(c) 注释掉，程序会一直阻塞在 if data, ok := <-c; ok 那一行

        close(c)

    }()

**for** {

        //ok为true说明channel没有关闭，为false说明管道已经关闭

**if** data, ok := <-c; ok {

            fmt.Println(data)

        } **else** {

**break**

        }

    }

    fmt.Println("Finished")

}

程序运行结果：



**注意：**

* channel不像文件一样需要经常去关闭，只有当你确实没有任何发送数据了，或者你想显式的结束range循环之类的，才去关闭channel；
* 关闭channel后，无法向channel 再发送数据(引发 panic 错误后导致接收立即返回零值)；
* 关闭channel后，可以继续从channel接收数据；
* 对于nil channel，无论收发都会被阻塞。

可以使用 **range** 来迭代不断操作channel：

**package main**

**import (**

**"fmt"**

**)**

**func** main() {

    c := make(**chan** int)

**go** **func**() {

**for** i := 0; i < 5; i++ {

            c <- i

        }

        //把 close(c) 注释掉，程序会一直阻塞在 for data := range c 那一行

        close(c)

    }()

**for** data := **range** c {

        fmt.Println(data)

    }

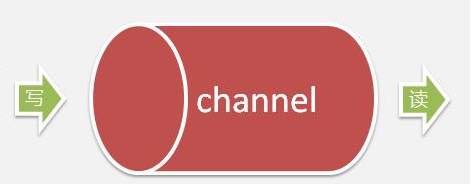
    fmt.Println("Finished")

}

### 单向channel及应用

默认情况下，通道channel是双向的，也就是，既可以往里面发送数据也可以同里面接收数据。

但是，我们经常见一个通道作为参数进行传递而只希望对方是单向使用的，要么只让它发送数据，要么只让它接收数据，这时候我们可以指定通道的方向。



单向channel变量的声明非常简单，如下：

**var** ch1 **chan** int // ch1是一个正常的channel，是双向的

**var** ch2 **chan**<- float64 // ch2是单向channel，只用于**写**float64数据

**var** ch3 <-**chan** int // ch3是单向channel，只用于**读**int数据

* chan<- 表示数据进入管道，要把数据写进管道，对于调用者就是输出。
* <-chan 表示数据从管道出来，对于调用者就是得到管道的数据，当然就是输入。

可以将 channel 隐式转换为单向队列，只收或只发，不能将单向 channel 转换为普通 channel：

    c := make(**chan** int, 3)

**var** send **chan**<- int = c // send-only

**var** recv <-**chan** int = c // receive-only

    send <- 1

    //<-send //invalid operation: <-send (receive from send-only type chan<- int)

    <-recv

    //recv <- 2 //invalid operation: recv <- 2 (send to receive-only type <-chan int)

    //不能将单向 channel 转换为普通 channel

    d1 := (**chan** int)(send) //cannot convert send (type chan<- int) to type chan int

    d2 := (**chan** int)(recv) //cannot convert recv (type <-chan int) to type chan int

示例代码：

// chan<- //只写

**func** counter(out **chan**<- int) {

**defer** close(out)

**for** i := 0; i < 5; i++ {

        out <- i //如果对方不读 会阻塞

    }

}

// <-chan //只读

**func** printer(in <-**chan** int) {

**for** num := **range** in {

        fmt.Println(num)

    }

}

**func** main() {

    c := make(**chan** int) // chan //读写

**go** counter(c) //生产者

    printer(c) //消费者

    fmt.Println("done")

}

生产者消费者模型

单向channel最典型的应用是“生产者消费者模型”

所谓“生产者消费者模型”: 某个模块（函数等）负责产生数据，这些数据由另一个模块来负责处理（此处的模块是广义的，可以是类、函数、go程、线程、进程等）。产生数据的模块，就形象地称为生产者；而处理数据的模块，就称为消费者。

单单抽象出生产者和消费者，还够不上是生产者／消费者模型。该模式还需要有一个缓冲区处于生产者和消费者之间，作为一个中介。生产者把数据放入缓冲区，而消费者从缓冲区取出数据。大概的结构如下图：



举一个寄信的例子来辅助理解一下，假设你要寄一封平信，大致过程如下：

1．把信写好——相当于生产者制造数据

2．把信放入邮筒——相当于生产者把数据放入缓冲区

3．邮递员把信从邮筒取出——相当于消费者把数据取出缓冲区

4．邮递员把信拿去邮局做相应的处理——相当于消费者处理数据

那么，这个缓冲区有什么用呢？为什么不让生产者直接调用消费者的某个函数，直接把数据传递过去，而画蛇添足般的设置一个缓冲区呢？

缓冲区的好处大概如下：

**1：解耦**

假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某个方法，那么生产者对于消费者就会产生依赖（也就是耦合）。将来如果消费者的代码发生变化，可能会直接影响到生产者。而如果两者都依赖于某个缓冲区，两者之间不直接依赖，耦合度也就相应降低了。

接着上述的例子，如果不使用邮筒（缓冲区），须得把信直接交给邮递员。那你就必须要认识谁是邮递员。这就产生和你和邮递员之间的依赖（相当于生产者和消费者的强耦合）。万一哪天邮递员换人了，你还要重新认识下一个邮递员（相当于消费者变化导致修改生产者代码）。 而邮筒相对来说比较固定，你依赖它的成本也比较低（相当于和缓冲区之间的弱耦合）。

**2：处理并发**

生产者直接调用消费者的某个方法，还有另一个弊端。由于函数调用是同步的（或者叫阻塞的），在消费者的方法没有返回之前，生产者只好一直等在那边。万一消费者处理数据很慢，生产者只能无端浪费时间。

使用了生产者／消费者模式之后，生产者和消费者可以是两个独立的并发主体。生产者把制造出来的数据往缓冲区一丢，就可以再去生产下一个数据。基本上不用依赖消费者的处理速度。

其实最当初这个生产者消费者模式，主要就是用来处理并发问题的。

从寄信的例子来看。如果没有邮筒，你得拿着信傻站在路口等邮递员过来收（相当于生产者阻塞）；又或者邮递员得挨家挨户问，谁要寄信（相当于消费者轮询）。

**3：缓存**

如果生产者制造数据的速度时快时慢，缓冲区的好处就体现出来了。当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。等生产者的制造速度慢下来，消费者再慢慢处理掉。

假设邮递员一次只能带走1000封信。万一某次碰上情人节送贺卡，需要寄出去的信超过1000封，这时候邮筒这个缓冲区就派上用场了。邮递员把来不及带走的信暂存在邮筒中，等下次过来时再拿走。

**示例代码：**

**package** main  
  
**import "fmt"**// 此通道只能写，不能读。  
**func** producer(out **chan**<- int) {  
 **for** i:= 0; i < 10; i++ {  
 out <- i\*i // 将 i\*i 结果写入到只写channel  
}  
 close(out)  
}  
  
// 此通道只能读，不能写  
**func** consumer(in <-**chan** int) {  
 **for** num := **range** in { // 从只读channel中获取数据  
 fmt.Println(**"num ="**, num)  
 }  
}  
  
**func** main() {  
ch := make(**chan** int) // 创建一个双向channel  
  
 // 新建一个groutine， 模拟生产者，产生数据，写入 channel  
**go** producer(ch) // channel传参， 传递的是引用。 // 主go程，模拟消费者，从channel读数据，打印到屏幕  
consumer(ch) // 与 producer 传递的是同一个 channel  
}

简单说明：首先创建一个双向的channel，然后开启一个新的goroutine，把双向通道作为参数传递到producer方法中，同时转成只写通道。子go程开始执行循环，向只写通道中添加数据，这就是生产者。主go程，直接调用consumer方法，该方法将双向通道转成只读通道，通过循环每次从通道中读取数据，这就是消费者。

注意：channel作为参数传递，是**引用传递**。

模拟订单

在实际的开发中，生产者消费者模式应用也非常的广泛，例如：在电商网站中，订单处理，就是非常典型的生产者消费者模式。

当很多用户单击下订单按钮后，订单生产的数据全部放到缓冲区（队列）中，然后消费者将队列中的数据取出来发送者仓库管理等系统。

通过生产者消费者模式，将订单系统与仓库管理系统隔离开，且用户可以随时下单（生产数据）。如果订单系统直接调用仓库系统，那么用户单击下订单按钮后，要等到仓库系统的结果返回。这样速度会很慢。

下面模拟一个下订单处理的过程。

**package** main  
  
**import "fmt"  
  
type** OrderInfo **struct** { // 创建结构体类型OrderInfo，只有一个id 成员  
id int  
}  
  
**func** producer2(out **chan** <- OrderInfo) { // 生成订单——生产者  
  
**for** i:=0; i<10; i++ { // 循环生成10份订单  
order := OrderInfo{id: i+1}  
 out <- order // 写入channel}  
 close(out) // 写完，关闭channel  
}  
  
**func** consumer2(in <- **chan** OrderInfo) { // 处理订单——消费者  
**for** order := **range** in { // 从channel 取出订单  
fmt.Println(**"订单id为："**, order.id) // 模拟处理订单  
}  
}  
  
**func** main() {  
 ch := make(**chan** OrderInfo) // 定义一个双向 channel， 指定数据类型为OrderInfo  
**go** producer2(ch) // 建新go程，传只写channel  
consumer2(ch) // 主go程，传只读channel  
}

OrderInfo为订单信息，这里为了简单只定义了一个订单编号属性，然后生产者模拟10个订单，消费者对产生的订单进行处理。

### 定时器

time.Timer

Timer是一个定时器。代表未来的一个单一事件，你可以告诉timer你要等待多长时间。

**type** Timer **struct** {  
 C <-**chan** Time  
 r runtimeTimer  
}

它提供一个channel，在定时时间到达之前，没有数据写入timer.C会一直阻塞。直到定时时间到，系统会自动向timer.C 这个channel中写入**当前时间**，阻塞即被解除。

示例代码：

**package main**

**import (**

"fmt"

"time"

**)**

**func** main() {

    //创建定时器，2秒后，定时器就会向自己的C字节发送一个time.Time类型的元素值

    timer1 := time.NewTimer(time.Second \* 2)

    t1 := time.Now() //当前时间

    fmt.Printf("t1: %v\n", t1)

    t2 := <-timer1.C

    fmt.Printf("t2: %v\n", t2)

    //如果只是想单纯的等待的话，可以使用 time.Sleep 来实现

    timer2 := time.NewTimer(time.Second \* 2)

    <-timer2.C

    fmt.Println("2s后")

    time.Sleep(time.Second \* 2)

    fmt.Println("再一次2s后")

    <-time.After(time.Second \* 2)

    fmt.Println("再再一次2s后")

    timer3 := time.NewTimer(time.Second)

**go** **func**() {

        <-timer3.C

        fmt.Println("Timer 3 expired")

    }()

    stop := timer3.Stop() //停止定时器

**if** stop {

        fmt.Println("Timer 3 stopped")

    }

    fmt.Println("before")

    timer4 := time.NewTimer(time.Second \* 5) //原来设置3s

    timer4.Reset(time.Second \* 1) //重新设置时间

    <-timer4.C

    fmt.Println("after")

}

定时器的常用操作：

1. 实现延迟功能
2. <-time.After(2 \* time.Second) //定时2s，阻塞2s,2s后产生一个事件，往channel写内容

fmt.Println("时间到")

1. time.Sleep(2 \* time.Second)

fmt.Println("时间到")

1. 延时2s后打印一句话

timer := time.NewTimer(2 \* time.Second)

<- timer.C

fmt.Println("时间到")

1. 定时器停止

timer := time.NewTimer(3 \* time.Second)

   go func() {

        <-timer.C

        fmt.Println("子go程可以打印了，因为定时器的时间到")

    }()

   timer.Stop() //停止定时器

    for {

}

1. 定时器重置

timer := time.NewTimer(3 \* time.Second)

ok := timer.Reset(1 \* time.Second) //重新设置为1s

fmt.Println("ok = ", ok)

<-timer.C

fmt.Println("时间到")

time.Ticker

Ticker是一个周期触发定时的计时器，它会按照一个时间间隔往channel发送系统当前时间，而channel的接收者可以以固定的时间间隔从channel中读取事件。

**type** Ticker **struct** {  
 C <-**chan** Time // The channel on which the ticks are delivered.r runtimeTimer  
}

示例代码：

**package main**

**import (**

"fmt"

"time"

**)**

**func** main() {

    //创建定时器，每隔1秒后，定时器就会给channel发送一个事件(当前时间)

    ticker := time.NewTicker(time.Second \* 1)

    i := 0

**go** **func**() {

**for** { //循环

            <-ticker.C

            i++

            fmt.Println("i = ", i)

**if** i == 5 {

                ticker.Stop() //停止定时器

break/return/Goexit()

            }

        }

    }() //别忘了()

    //死循环，特地不让main goroutine结束

**for** {

    }

}

**总结：**

Channel:

1. **基本特性** 
   1. 管道内的数据只能单项流动
   2. 管道内的数据不能重复被读取
   3. 管道中数据正确流动，必须读写两端同时在线
   4. Var ch chan int ---- 没有空间，无效的channel
2. **channel种类：**
   1. 无缓冲channel： 没有缓冲区 —— channel 不能缓冲任何数据。 有 r 没有写 ， r端go程阻塞。 有w，没有 r， w端go程阻塞。 要求 r、w同时在线。

chr := make(chan int)

chr := make(chan int, 0)

b) 有缓冲channel： 自带缓冲区 —— channel 可以根据缓冲区大小，存储适量的数据。

chr := make(chan int, 3)

len(chr): 统计缓冲区剩余元素个数。

cap(chr): 统计缓冲区容量。

**同步通信**： —— 无缓冲channel

当一个调用发出，如果没有得到结果，那么这个调用不返回。 —— 阻塞

相当于 打电话： 要求通信双方必须同时在线。

**异步通信**：—— 有缓冲channel

当一个调用发出，不等待调用结果，直接返回。 —— 非阻塞

相当于 发短信 ：信息发出方，发送完，直接结束。

1. **关闭channel**

当写端写完全部数据，使用 close(ch) 关闭 channel。

判断 chanel关闭 的语法：

for {

if num，has := <-ch; has == true { 【注意】： ch必须 写成 <-ch

}

}

for {

if num，has := <-ch; has {

}

}

当channel已关闭：

读： 可以！ 读到数据类型的 默认值： int - 0、 string -“”、bool-false

写：不可以！！panic: send on closed channel

判断channel关闭 简化写法：

for num := range ch {

}

【注意】： ch 不能写成 <- ch ！！！

1. **生产者消费者模型**
2. **Time**

延迟定时

1. Sleep
2. Time.newtimer
3. Time.after

定时器停止、重置

1. t1 := time.NewTimer()

t1.stop()

2. t1 := time.NewTimer()

t1(新时间)

周期定时

Ticker： —— 周期定时。

特性：一次设定，系统循环写入C 系统时间。

ticker 只有 stop() 没有 reset() 方法

注意 每次到周期定时

## select

### select作用

Go里面提供了一个关键字select，通过select可以**监听channel上的数据流动。**

select的用法与switch语言非常类似，由select开始一个新的选择块，每个选择条件由case语句来描述。

与switch语句相比，select有比较多的限制，其中最大的一条限制就是每个**case语句里必须是一个IO操作**，大致的结构如下：

**select** {

**case** <- chan1:

        // 如果chan1成功读到数据，则进行该case处理语句

**case** chan2 <- 1:

        // 如果成功向chan2写入数据，则进行该case处理语句

**default**:

        // 如果上面都没有成功，则进入default处理流程

    }

在一个select语句中，Go语言会按顺序从头至尾评估每一个发送和接收的语句。

如果其中的任意一语句可以继续执行(即没有被阻塞)，那么就从那些可以执行的语句中任意选择一条来使用。

如果没有任意一条语句可以执行(即所有的通道都被阻塞)，那么有两种可能的情况：

* 如果给出了default语句，那么就会执行default语句，同时程序的执行会从select语句后的语句中恢复。
* 如果没有default语句，那么select语句将被阻塞，直到至少有一个通信可以进行下去。

示例代码：

**package main**

**import (**

"fmt"

**)**

**func** fibonacci(c, quit **chan** int) {

    x, y := 1, 1

**for** {

**select** {

**case** c <- x:

            x, y = y, x+y

**case** <-quit:

            fmt.Println("quit")

**return**

        }

    }

}

**func** main() {

    c := make(**chan** int)

    quit := make(**chan** int)

**go** **func**() {

**for** i := 0; i < 6; i++ {

            fmt.Println(<-c)

        }

        quit <- 0

    }()

    fibonacci(c, quit)

}

运行结果如下：



### 超时

有时候会出现goroutine阻塞的情况，那么我们如何避免整个程序进入阻塞的情况呢？我们可以利用select来设置超时，通过如下的方式实现：

**func** main() {

    c := make(**chan** int)

    out := make(**chan** bool)

**go** **func**() {

**for** {

**select** {

**case** v := <-c:

                fmt.Println(v)

**case** <-time.After(5 \* time.Second):

                fmt.Println("timeout")

                out <- true

**return**

            }

        }

    }()

    //c <- 666 // 注释掉，引发 timeout

    <-out

}

### 总结：

select ：监听 channel 上的 数据流动（r、w） 【重点】

语法：

select {

case channnel IO 操作

case channnel IO 操作：

.....

default :

}

select 使用注意事项：

1. case分支中必须是一个 IO操作

2. 当case分支不满足监听条件，阻塞当前 case 分支

3. 如果同时有多个case分支满足，select随机选定一个执行（ select底层实现，case对应一个 goroutine）

4. 一次select 监听，只能执行一个case分支。通常将select放于for 循环中

5. default 在所有case均不满足时，默认执行的分支，为防止忙轮询，通常将 for 里 select中的 default 省略。

【重要结论】：使用 select的 go程，与其他go程间 采用 【异步通信】 通信方式。

## 锁和条件变量

前面我们为了解决go程同步的问题我们使用了channel，但是GO也提供了传统的同步工具。

它们都在GO的标准库代码包sync和sync/atomic中。

下面我们看一下锁的应用。

什么是锁呢？就是某个go程（线程）在访问某个资源时先锁住，防止其它go程的访问，等访问完毕解锁后其他go程再来加锁进行访问。这和我们生活中加锁使用公共资源相似，例如：公共卫生间。

### 死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，

示例代码：

**package main**

**import** "fmt"

**func** main() {

ch := make(**chan** int)

ch <- 1 // I'm blocked because there is no channel read yet.

fmt.Println("send")

**go** func() {

<-ch // I will never be called for the main routine is blocked!

fmt.Println("received")

}()

fmt.Println("over")

**}**

### 互斥锁

每个资源都对应于一个可称为 "互斥锁" 的标记，这个标记用来保证在任意时刻，只能有一个go程（线程）访问该资源。其它的go程只能等待。

互斥锁是传统并发编程对共享资源进行访问控制的主要手段，它由标准库sync中的Mutex结构体类型表示。sync.Mutex类型只有两个公开的指针方法，Lock和Unlock。Lock锁定当前的共享资源，Unlock进行解锁。

在使用互斥锁时，一定要注意：对资源操作完成后，一定要解锁，否则会出现流程执行异常，死锁等问题。通常借助defer。锁定后，立即使用defer语句保证互斥锁及时解锁。如下所示：

**var** mutex sync.Mutex // 定义互斥锁变量 mutex

**func** write(){  
 mutex.Lock( )  
 **defer** mutex.Unlock( )  
}

我们可以使用互斥锁来解决前面提到的多任务编程的问题，如下所示：

**package** main  
  
**import** (  
 **"fmt"  
 "time"  
 "sync"**)  
  
**var** mutex sync.Mutex  
  
**func** printer(str string) {  
 mutex.Lock() // 添加互斥锁  
**defer** mutex.Unlock() // 使用结束时解锁  
**for** \_, data := **range** str { // 迭代器  
fmt.Printf(**"%c"**, data)  
 time.Sleep(time.***Second***) // 放大go程竞争效果  
}  
 fmt.Println() }  
  
**func** person1(s1 string) {  
 printer(s1)  
}  
  
**func** person2() {  
 printer(**"world"**) // 调函数时传参  
}  
  
**func** main() {  
 **go** person1(**"hello"**) // main 中传参  
**go** person2()  
 **for** {  
 ;  
 }  
}

程序执行结果与多任务资源竞争时一致。最终由于添加了互斥锁，可以按序先输出hello再输出 world。但这里需要我们自行创建互斥锁，并在适当的位置对锁进行释放。

### 读写锁

互斥锁的本质是当一个goroutine访问的时候，其他goroutine都不能访问。这样在资源同步，避免竞争的同时也降低了程序的并发性能。程序由原来的并行执行变成了串行执行。

其实，当我们对一个不会变化的数据只做“读”操作的话，是不存在资源竞争的问题的。因为数据是不变的，不管怎么读取，多少goroutine同时读取，都是可以的。

所以问题不是出在“读”上，主要是修改，也就是“写”。修改的数据要同步，这样其他goroutine才可以感知到。所以真正的互斥应该是读取和修改、修改和修改之间，读和读是没有互斥操作的必要的。

因此，衍生出另外一种锁，叫做**读写锁**。

读写锁可以让多个读操作并发，同时读取，但是对于写操作是完全互斥的。也就是说，当一个goroutine进行写操作的时候，其他goroutine既不能进行读操作，也不能进行写操作。

GO中的读写锁由结构体类型sync.RWMutex表示。此类型的方法集合中包含两对方法：

一组是对写操作的锁定和解锁，简称“写锁定”和“写解锁”：

func (\*RWMutex)Lock()

func (\*RWMutex)Unlock()

另一组表示对读操作的锁定和解锁，简称为“读锁定”与“读解锁”：

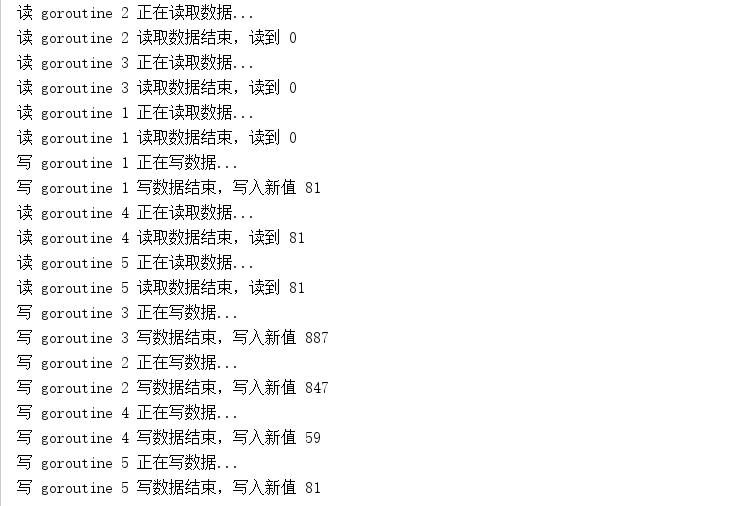
func (\*RWMutex)RLock()

func (\*RWMutex)RUnlock()

读写锁基本示例：

**package** main  
  
**import** (  
 **"sync"  
 "fmt"  
 "math/rand"**)  
  
**var** count int **// 全局变量count  
var** rwlock sync.RWMutex **// 全局读写锁 rwlock****func** read(n int) {  
 rwlock.RLock()  
 fmt.Printf(**"读 goroutine %d 正在读取数据...\n"**, n)  
 num := count  
 fmt.Printf(**"读 goroutine %d 读取数据结束，读到 %d\n"**, n, num)  
 **defer** rwlock.RUnlock()  
}  
**func** write(n int) {  
 rwlock.Lock()  
 fmt.Printf(**"写 goroutine %d 正在写数据...\n"**, n)  
 num := rand.Intn(1000)  
 count = num  
 fmt.Printf(**"写 goroutine %d 写数据结束，写入新值 %d\n"**, n, num)  
 **defer** rwlock.Unlock()  
}  
  
**func** main() {  
 **for** i:=0; i<5; i++ {  
 **go** read(i+1)  
 }  
 **for** i:=0; i<5; i++ {  
 **go** write(i+1)  
 }  
 **for** {  
 ;  
 }  
}

程序的执行结果：



我们在read里使用读锁，也就是RLock和RUnlock，写锁的方法名和我们平时使用的一样，是Lock和Unlock。这样，我们就使用了读写锁，可以并发地读，但是同时只能有一个写，并且写的时候不能进行读操作。

我们从结果可以看出，读取操作可以并行，例如2,3,1正在读取，但是同时只能有一个写，例如1正在写，只能等待1写完，这个过程中不允许进行其它的操作。

处于读锁定状态，那么针对它的写锁定操作将永远不会成功，且相应的Goroutine也会被一直阻塞。因为它们是互斥的。

总结：读写锁控制下的多个写操作之间都是互斥的，并且写操作与读操作之间也都是互斥的。但是，多个读操作之间不存在互斥关系。

从互斥锁和读写锁的源码可以看出，它们是同源的。读写锁的内部用互斥锁来实现写锁定操作之间的互斥。可以把读写锁看作是互斥锁的一种扩展。

### 条件变量

在讲解条件变量之前，先回顾一下前面我们所涉及的“生产者消费者模型”：

**package** main  
  
**import "fmt"**//只写，不读。  
**func** producer(out **chan**<- int) {  
 **for** i:= 0; i < 10; i++ {  
 out <- i\*i   
}  
 close(out)  
}  
//只读，不写  
**func** consumer(in <-**chan** int) {  
 **for** num := **range** in {   
 fmt.Println(**"num = "**, num)  
 }  
}  
**func** main() {  
ch := make(**chan** int) // 创建一个双向channel  
**go** producer(ch) // 生产者，产生数据，写入 channel  
consumer(ch) // 消费者，从channel读数据，打印到屏幕  
}

这个案例中，虽然实现了生产者消费者的功能，但有一个问题。如果有多个消费者来消费数据，并且并不是简单的从channel中取出来进行打印，而是还要进行一些复杂的运算。在consumer( )方法中的实现是否有问题呢？如下所示：

**package** main

**import** "fmt"

**import** "sync"

**import** "time"

**var** sum int

**func** producer(out **chan**<- int) {

**for** i := 0; i < =100; i++ {

out <-i

}

close(out);

}

// 此chanel 只能读，不能写

**func** consumer(in <-**chan** int) {

**for** num := range in {

sum +=num

}

fmt.println(“sum = ”, sum)

}

**func** main() {

    ch:= make(**chan** int) // 创建一个双向通道

**go** producer(ch) // go程1，生产者，生产数字，写入channel

**go** consumer(ch) // go程2，消费者1

consumer(ch) // 主go程，消费者。从channel读取内容打印

**for** {

;

    }

}

在上面的代码中，加了一个消费者，同时在consumer方法中，将数据取出来后，又进行了一组运算。这时可能会出现一个go程从管道中取出数据，参与加法运算，但是还没有算完另外一个go程又从管道中取出一个数据赋值给了num变量。所以这样累加计算，很有可能出现问题。当然，按照前面的知识，解决这个问题的方法很简单，就是通过加锁的方式来解决。增加生产者也是一样的道理。

另外一个问题，如果消费者比生产者多，仓库中就会出现没有数据的情况。我们需要不断的通过循环来判断仓库队列中是否有数据，这样会造成cpu的浪费。反之，如果生产者比较多，仓库很容易满，满了就不能继续添加数据，也需要循环判断仓库满这一事件，同样也会造成CPU的浪费。

我们希望当仓库满时，生产者停止生产，等待消费者消费；同理，如果仓库空了，我们希望消费者停下来等待生产者生产。为了达到这个目的，这里引入条件变量。（需要注意：如果仓库队列用channel，是不存在以上情况的，因为channel被填满后就阻塞了，或者channel中没有数据也会阻塞）。

**条件变量：**条件变量的作用并不保证在同一时刻仅有一个go程（线程）访问某个共享的数据资源，而是在对应的共享数据的状态发生变化时，通知阻塞在某个条件上的go程（线程）。条件变量不是锁，在并发中不能达到同步的目的，因此**条件变量总是与锁一块使用。**

例如，我们上面说的，如果仓库队列满了，我们可以使用条件变量让生产者对应的goroutine暂停（阻塞），但是当消费者消费了某个产品后，仓库就不再满了，应该唤醒（发送通知给）阻塞的生产者goroutine继续生产产品。

GO标准库中的sync.Cond类型代表了条件变量。条件变量要与锁（互斥锁，或者读写锁）一起使用。成员变量L代表与条件变量搭配使用的锁。

**type** Cond **struct** {  
 noCopy noCopy  
 *// L is held while observing or changing the condition* **L Locker**  
 notify notifyList  
 checker copyChecker  
}

对应的有3个常用方法，Wait，Signal，Broadcast。

1. **func** (c \*Cond) Wait()

该函数的作用可归纳为如下三点：

1. 阻塞等待条件变量满足
2. 释放已掌握的互斥锁相当于cond.L.Unlock()。 注意：**两步为一个原子操作。**
3. 当被唤醒，Wait()函数返回时，解除阻塞并重新获取互斥锁。相当于cond.L.Lock()
4. **func** (c \*Cond) Signal()

单发通知，给一个正等待（阻塞）在该条件变量上的goroutine（线程）发送通知。

1. **func** (c \*Cond) Broadcast()

广播通知，给正在等待（阻塞）在该条件变量上的所有goroutine（线程）发送通知。

下面我们用条件变量来编写一个“生产者消费者模型”

示例代码：

**package** main  
**import "fmt"  
import "sync"  
import "math/rand"  
import "time"  
  
var** cond sync.Cond // 创建全局条件变量  
// 生产者**func** producer(out **chan**<- int, idx int) {  
 **for** {  
 cond.L.Lock() // 条件变量对应互斥锁加锁**for** len(out) == 3{ // 产品区满 等待消费者消费cond.Wait() // 挂起当前go程， 等待条件变量满足，被消费者唤醒  
}  
 num := rand.Intn(1000) // 产生一个随机数  
out <- num // 写入到 channel 中 （生产）  
fmt.Printf(**"%dth 生产者，产生数据 %3d, 公共区剩余%d个数据\n"**, idx, num, len(out))

cond.Signal() // 唤醒 阻塞的 消费者  
 cond.L.Unlock() // 生产结束，解锁互斥锁time.Sleep(time.***Second***) // 生产完休息一会，给其他go程执行机会  
}  
}  
//消费者**func** consumer(in <-**chan** int, idx int) {  
 **for** {  
 cond.L.Lock() // 条件变量对应互斥锁加锁（与生产者是同一个）  
**for** len(in) == 0 { // 产品区为空 等待生产者生产  
cond.Wait() // 挂起当前go程， 等待条件变量满足，被生产者唤醒  
}  
 num := <-in // 将 channel 中的数据读走 （消费）  
fmt.Printf(**"---- %dth 消费者, 消费数据 %3d,公共区剩余%d个数据\n"**, idx, num, len(in))

cond.Signal() // 唤醒 阻塞的 生产者  
 cond.L.Unlock() // 消费结束，解锁互斥锁time.Sleep(time.***Millisecond*** \* 500) //消费完 休息一会，给其他go程执行机会  
}  
}  
**func** main() {  
 rand.Seed(time.Now().UnixNano()) // 设置随机数种子  
quit := make(**chan** bool) // 创建用于结束通信的 channel  
  
product := make(**chan** int, 3) // 产品区（公共区）使用channel 模拟  
cond.L = new(sync.Mutex) // 创建互斥锁和条件变量  
**for** i := 0; i < 5; i++ { // 5个消费者  
**go** producer(product, i+1)  
 }  
 **for** i := 0; i < 3; i++ { // 3个生产者  
**go** consumer(product, i+1)  
 }  
 <-quit // 主go程阻塞 不结束  
}

1. main函数中定义quit，其作用是让主go程阻塞。
2. 定义product作为队列，生产者产生数据保存至队列中，最多存储3个数据，消费者从中取出数据模拟消费
3. 条件变量要与锁一起使用，这里定义全局条件变量cond，它有一个属性：L Locker。是一个互斥锁。
4. 开启5个消费者go程，开启3个生产者go程。
5. producer生产者，在该方法中开启互斥锁，保证数据完整性。并且判断队列是否满，如果已满，调用wait()让该goroutine阻塞。当消费者取出数后执行cond.Signal()，会唤醒该goroutine，继续生产数据。
6. consumer消费者，同样开启互斥锁，保证数据完整性。判断队列是否为空，如果为空，调用wait()使得当前goroutine阻塞。当生产者产生数据并添加到队列，执行cond.Signal() 唤醒该goroutine。

### 总结：

死锁：

运行时错误 —— 在编码期间提早预见，提早规避。

**死锁1**： 同一个go程，使用一个channel 自己读自己写。

**死锁2**： 读写位于两个go程间。但是go程的创建在 r 或 w 之后。r 或 w 会造成go程阻塞，导致另一个go程无法创建。

**死锁3**： 两个go程，两个channel。 go程1 对channel1读，成功写入 channel2. go程2 对channel2读，成功写入 channel1. 交叉锁

**死锁4**： channel 和 读写锁、互斥锁。

互斥锁

建议锁。不具有强制性。

保护公共区，被锁住后，只有成功加锁的 go 程能正常访问。其他go程阻塞在 锁的等待事件上。

使用注意：

1. 公共区访问开始之前，加锁

2. 访问结束后 立即解锁。 锁的 粒度 越小越好。

使用方法：

1. 定义 互斥锁。 var mutex sync.Mutex

2. 加锁 mutex.Lock()

。。。 访问公共区数据

3. 解锁 mutex.Unlock()

读写锁

读共享，写独占。

写锁优先级高于读锁。

使用注意：

1. 公共区访问开始之前，加锁

2. 访问结束后 立即解锁。 锁的 粒度 越小越好。

3. 锁只有一把。但是具备两种加锁属性。读属性加锁。写属性加锁。

使用方法：

1. 定义 读写锁。 var rwmutex sync.RWMutex

2. 加读锁 rwmutex.RLock() 加写锁 rwmutex.Lock()

。。。 读公共区数据 。。。 写公共区数据

3. 解锁 rwmutex.RUnlock() 解写锁 rwmutex.UnLock()

多go程同步时，尽量少 锁（读写锁、互斥锁）、和 channel 【混用】。 —— 条件变量。

条件变量

条件变量：

Cond 语法：

type Cond struct {

noCopy noCopy

L Locker // 锁：互斥、读写

notify notifyList

checker copyChecker

}

Cond.Wait():

1) 阻塞等待条件变量满足

2）释放已经掌握的互斥锁 （1/2 两步为 原子操作。）

。。。 阻塞等。

3） 被唤醒时，解除阻塞，重新加锁

Cond.Signal()：

唤醒阻塞在条件变量上的 一个 对象。

Cond.Broadcast()

唤醒阻塞在条件变量上的 所有 对象。 引发惊群，增加服务器负载

signal和Unlock的调用顺序，在调用signal时建议保持c.L的锁定

条件变量使用流程： --- 生产者为例

1. 创建条件变量 var cond Sync.Cond —— 结构体

2. 初始化 条件变量使用的 锁 cond.L = new(sysc.Mutex)

3. 给条件变量的 互斥锁 加锁 cond.L.Lock()

4. 判断条件变量是否满足。 调用wait ：1） 2） 3）

for len(ch) == cap(ch) { // 此处判断，必须使用 for ，而不能使用 if

cond.Wait()

}

5. 生产数据： 产生随机数 num := rand.Intn()

6. 写入公共区（缓冲区） ch <- num

7. 唤醒对端（消费者） cond.Signal()

8. 给条件变量的 互斥锁 解锁 cond.L.UnLock()

# Go网络编程

## 网络概述

### 协议

从应用的角度出发，协议可理解为**“规则”**，是数据传输和数据的解释的规则。假设，A、B双方欲传输文件。规定：

* 第一次，传输文件名，接收方接收到文件名，应答OK给传输方；
* 第二次，发送文件的尺寸，接收方接收到该数据再次应答一个OK；
* 第三次，传输文件内容。同样，接收方接收数据完成后应答OK表示文件内容接收成功。

由此，无论A、B之间传递何种文件，都是通过三次数据传输来完成。A、B之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B之间达成的这个相互遵守的规则即为协议。

这种仅在A、B之间被遵守的协议称之为原始协议。

当此协议被更多的人采用，不断的增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛应用于各种文件传输过程中。该协议就成为一个标准协议。最早的ftp协议就是由此衍生而来。

### 典型协议

应用层 常见的协议有HTTP协议，FTP协议。

传输层 常见协议有TCP/UDP协议。

网络层 常见协议有IP协议、ICMP协议、IGMP协议。

链路层 常见协议有ARP协议、RARP协议。

TCP[传输控制协议](http://baike.baidu.com/view/544903.htm" \t "_blank)（Transmission Control Protocol）是一种面向连接的、可靠的、基于字节流的[传输层](http://baike.baidu.com/view/239605.htm" \t "_blank)通信协议。

UDP用户数据报协议（User Datagram Protocol）是[OSI](http://baike.baidu.com/view/113948.htm" \t "_blank)参考模型中一种无连接的[传输层](http://baike.baidu.com/view/239605.htm" \t "_blank)协议，提供面向事务的简单不可靠信息传送服务。

HTTP[超文本传输协议](http://baike.baidu.com/view/468465.htm" \t "_blank)（Hyper Text Transfer Protocol）是[互联网](http://baike.baidu.com/view/6825.htm" \t "_blank)上应用最为广泛的一种[网络协议](http://baike.baidu.com/view/16603.htm" \t "_blank)。

FTP文件传输协议（File Transfer Protocol）

IP协议是[因特网](http://baike.baidu.com/view/1706.htm" \t "_blank)互联协议（Internet Protocol）

ICMP协议是Internet控制[报文](http://baike.baidu.com/view/175122.htm" \t "_blank)协议（Internet Control Message Protocol）它是[TCP/IP协议族](http://baike.baidu.com/view/2221037.htm" \t "_blank)的一个子协议，用于在IP[主机](http://baike.baidu.com/view/23880.htm" \t "_blank)、[路由](http://baike.baidu.com/view/18655.htm" \t "_blank)器之间传递控制消息。

IGMP协议是 Internet 组管理协议（Internet Group Management Protocol），是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间。

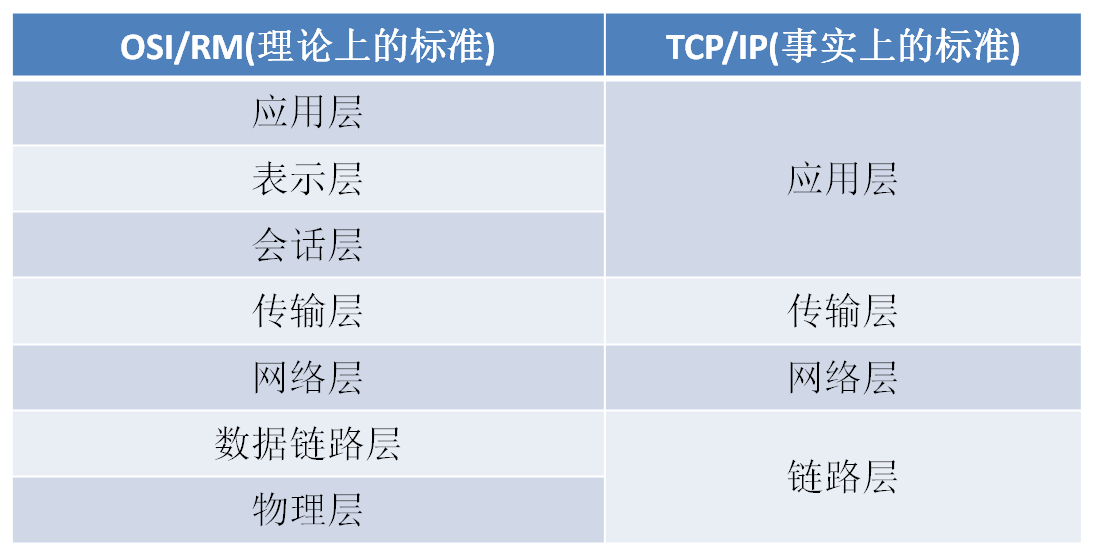
[ARP](http://baike.baidu.com/view/32698.htm" \t "_blank)协议是正向[地址解析协议](http://baike.baidu.com/view/149421.htm" \t "_blank)（Address Resolution Protocol），通过已知的IP，寻找对应主机的[MAC地址](http://baike.baidu.com/view/69334.htm" \t "_blank)。

[RARP](http://baike.baidu.com/view/32772.htm" \t "_blank)是反向地址转换协议，通过MAC地址确定IP地址。

### 分层模型

网络分层架构

为了减少协议设计的复杂性，大多数网络模型均采用分层的方式来组织。每一层都有自己的功能，就像建筑物一样，每一层都靠下一层支持。每一层利用下一层提供的服务来为上一层提供服务，本层服务的实现细节对上层屏蔽。



越下面的层，越靠近硬件；越上面的层，越靠近用户。至于每一层叫什么名字，对应编程而言不重要，但面试的时候，面试官可能会问每一层的名字。

业内普遍的分层方式有两种。OSI七层模型 和TCP/IP四层模型。可以通过背诵两个口诀来快速记忆：

OSI七层模型：物、数、网、传、会、表、应

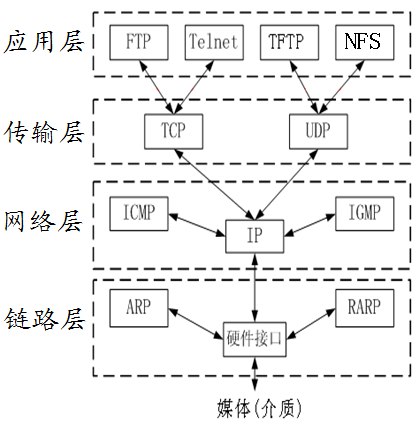
TCP/IP四层模型：链、网、传、应

1. **物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输，到达目的地后再转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。
2. **数据链路层**：定义了如何让格式化数据以帧为单位进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。如：串口通信中使用到的115200、8、N、1
3. **网络层**：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。
4. **传输层**：定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（用户数据报协议，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。
5. **会话层**：通过传输层(端口号：传输端口与接收端口)建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）。
6. **表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码(EBCDIC)，而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。
7. **应用层**：是最靠近用户的OSI层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。

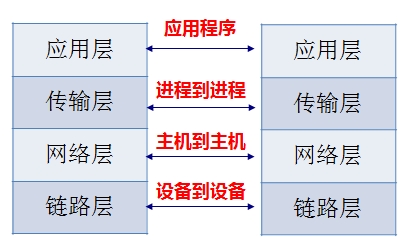
层与协议

每一层都是为了完成一种功能，为了实现这些功能，就需要大家都遵守共同的规则。大家都遵守这规则，就叫做“协议”（protocol）。

网络的每一层，都定义了很多协议。这些协议的总称，叫“TCP/IP协议”。TCP/IP协议是一个大家族，不仅仅只有TCP和IP协议，它还包括其它的协议，如下图：



### 各层功能



#### 链路层

以太网规定，连入网络的所有设备，都必须具有“网卡”接口。数据包必须是从一块网卡，传送到另一块网卡。通过网卡能够使不同的计算机之间连接，从而完成数据通信等功能。网卡的地址——MAC地址，就是数据包的物理发送地址和物理接收地址。

#### 网络层

网络层的作用是引进一套新的地址，使得我们能够区分不同的计算机是否属于同一个子网络。这套地址就叫做“网络地址”，就是我们平时所说的IP地址。这个IP地址好比我们的手机号码，通过手机号码可以得到用户所在的归属地。

网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。网络层协议包含的主要信息是源IP和目的IP。

于是，“网络层”出现以后，每台计算机有了两种地址，一种是 MAC 地址，另一种是网络地址。**两种地址之间没有任何联系**，MAC 地址是绑定在网卡上的，网络地址则是管理员分配的，它们只是随机组合在一起。

网络地址帮助我们确定计算机所在的子网络，MAC 地址则将数据包送到该子网络中的目标网卡。因此，从逻辑上可以推断，必定是先处理网络地址，然后再处理 MAC 地址。

#### 传输层

当我们一边聊QQ，一边聊微信，当一个数据包从互联网上发来的时候，我们怎么知道，它是来自QQ的内容，还是来自微信的内容？

也就是说，我们还需要一个参数，表示这个数据包到底供哪个程序（进程）使用。这个参数就叫做“端口”（port），它其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。

端口特点：

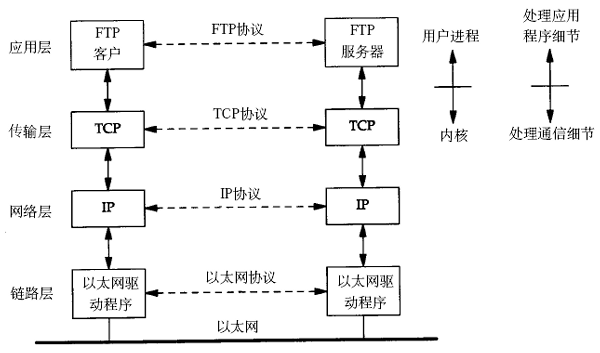
* 对于同一个端口，在不同系统中对应着不同的进程
* 对于同一个系统，一个端口只能被一个进程拥有

#### 应用层

应用程序收到“传输层”的数据，接下来就要进行解读。由于互联网是开放架构，数据来源五花八门，必须事先规定好格式，否则根本无法解读。“应用层”的作用，就是规定应用程序的数据格式。

### 通信过程

两台计算机通过TCP/IP协议通讯的过程如下所示：



TCP/IP通信过程

### 总结

协议：

一组规范。要求使用协议的双方，必须要严格准守。

常见协议：

应用层： http、ftp、ssh、telnet、NFS。。。

传输层：TCP、UDP

网络层；IP、ICMP、IGMP

链路层：RARP、ARP

网络分层模式：

OSI 七层模型： 物、数、 网、传、会、表、应

TCP/IP 四层模型： 链、 网、传、 应

数据通信流程：

封装过程：数据 ——> 应用层 ——> 传输层 ——> 网络层 ——> 链路层 ——> 网卡

。。。。进入网络环境传输。

解封过程：网卡——> 链路层 ——>网络层——> 传输层——> 应用层 ——> 数据

链路层：

【mac地址】 —— 【目标mac地址】

ARP协议：借助 IP 获取 mac 地址

网络层：

Ip地址： 在网络环境中唯一标识一台主机。

【主机】 —— 【目标主机】

传输层：

端口（port）号：在一台网络主机中唯一标识 一个 进程。

【进程】 —— 【目标进程】

Ip + port = 在网络中 唯一标识一个 进程。 —— 套接字！！！

--------------------------------------以上为 内核 完成操作。

应用层：（非必须协议）

【应用数据】—— 【目标应用数据】

socket 套接字： 伪文件。—— 内存。 双向全双工

在一次网络通信过程中，参与通信的 socket 必须 “成对”出现。

socket 内核实现原理：

一个 socket 内部相当于有 两个 channel （一个读、另一个写）， 外部对这两个 channel 不可见。

Ip + port = 在网络中 唯一标识一个 进程。 —— 套接字！！！

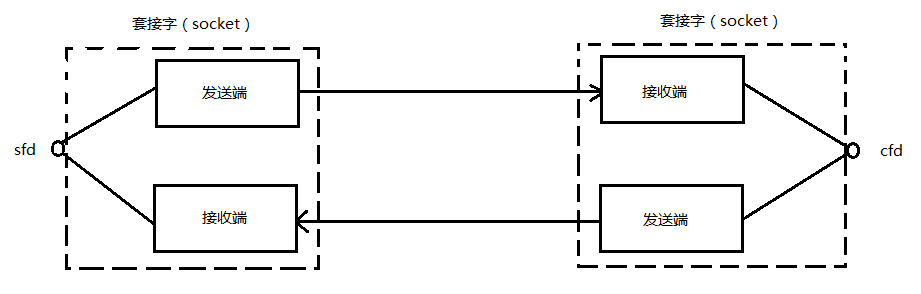
## Socket编程

### 什么是Socket

Socket，英文含义是【插座、插孔】，一般称之为套接字，用于描述IP地址和端口。可以实现不同程序间的数据通信。

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

套接字的内核实现较为复杂，不宜在学习初期深入学习，了解到如下结构足矣。



套接字通讯原理示意

在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一标识网络通讯中的一个进程。“IP地址+端口号”就对应一个socket。欲建立连接的两个进程各自有一个socket来标识，那么这两个socket组成的socket pair就唯一标识一个连接。因此可以用Socket来描述网络连接的一对一关系。

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

### 网络应用程序设计模式

C/S模式

传统的网络应用设计模式，客户机(client)/服务器(server)模式。需要在通讯两端各自部署客户机和服务器来完成数据通信。

B/S模式

浏览器(Browser)/服务器(Server)模式。只需在一端部署服务器，而另外一端使用每台PC都默认配置的浏览器即可完成数据的传输。

优缺点

对于C/S模式来说，其优点明显。客户端位于目标主机上可以保证性能，将数据缓存至客户端本地，从而**提高数据传输效率**。且，一般来说客户端和服务器程序由一个开发团队创作，所以他们之间**所采用的协议相对灵活**。可以在标准协议的基础上根据需求裁剪及定制。例如，腾讯所采用的通信协议，即为ftp协议的修改剪裁版。

因此，传统的网络应用程序及较大型的网络应用程序都首选C/S模式进行开发。如，知名的网络游戏魔兽世界。3D画面，数据量庞大，使用C/S模式可以提前在本地进行大量数据的缓存处理，从而提高观感。

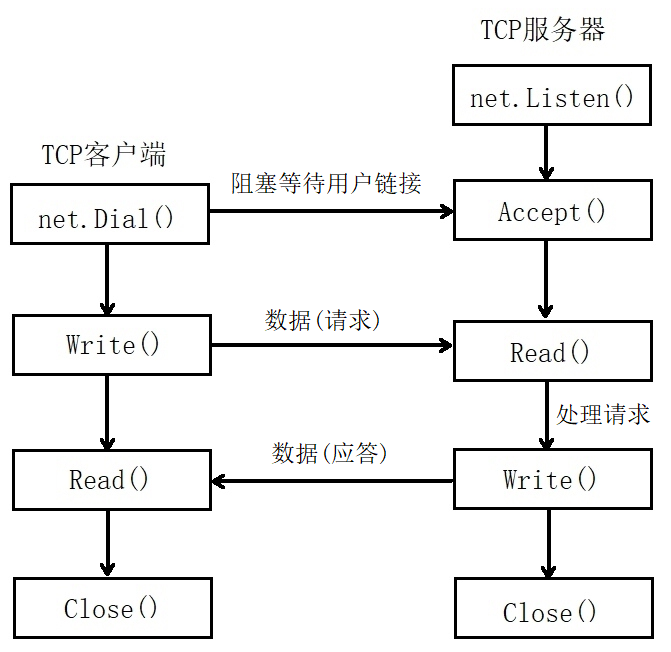
C/S模式的缺点也较突出。由于客户端和服务器都需要有一个开发团队来完成开发。**工作量**将成倍提升，开发周期较长。另外，从用户角度出发，需要将客户端安插至用户主机上，对用户主机的**安全性构成威胁**。这也是很多用户不愿使用C/S模式应用程序的重要原因。

B/S模式相比C/S模式而言，由于它没有独立的客户端，使用标准浏览器作为客户端，其工作**开发量较小**。只需开发服务器端即可。另外由于其采用浏览器显示数据，因此移植性非常好，**不受平台限制**。如早期的偷菜游戏，在各个平台上都可以完美运行。

B/S模式的缺点也较明显。由于使用第三方浏览器，因此**网络应用支持受限**。另外，没有客户端放到对方主机上，**缓存数据不尽如人意**，从而传输数据量受到限制。应用的观感大打折扣。第三，必须与浏览器一样，采用标准http协议进行通信，**协议选择不灵活**。

因此在开发过程中，模式的选择由上述各自的特点决定。根据实际需求选择应用程序设计模式。

### 简单C/S模型通信



**TCP的C/S架构**

Server端：

Listen函数：

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

network：选用的协议：TCP、UDP， 如：“tcp”或 “udp” **注意**：只支持小写字母

address：IP地址+端口号, 如：“127.0.0.1:8000”或 “:8000”

Listener 接口：

type Listener interface {

**Accept**() (**Conn**, error)

**Close**() error

**Addr**() Addr

}

**Conn** 接口：

type Conn interface {

**Read**(b []byte) (n int, err error)

**Write**(b []byte) (n int, err error)

**Close**() error

LocalAddr() Addr

RemoteAddr() Addr

SetDeadline(t time.Time) error

SetReadDeadline(t time.Time) error

SetWriteDeadline(t time.Time) error

}

参看 <https://studygolang.com/pkgdoc> 中文帮助文档中的demo：

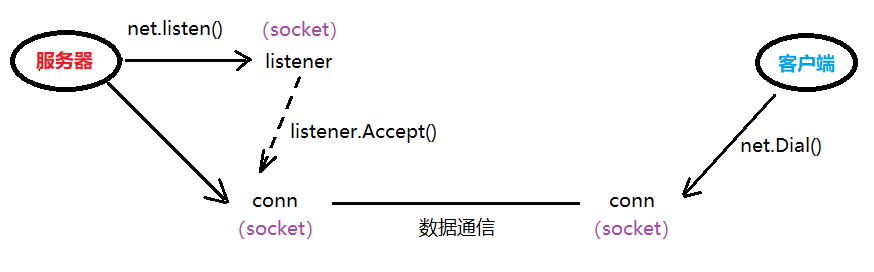
示例代码：

TCP服务器.go

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 // 创建监听  
listener, err:= net.Listen(**"tcp"**, **":8000"**) // tcp 不能使用大写  
 **if** err != nil {  
 fmt.Println(**"listen err:"**, err)  
 **return** }  
 **defer** listener.Close() // 主go程结束时，关闭listener  
fmt.Println(**"服务器等待客户端建立连接..."**)  
 // 等待客户端连接请求  
conn, err := listener.Accept()  
 **if** err != nil {  
 fmt.Println(**"accept err:"**, err)  
 **return** }

**defer** conn.Close() // 使用结束，断开与客户端链接  
 fmt.Println(**"客户端与服务器连接建立成功..."**)  
  
 // 接收客户端数据  
buf := make([]byte, 1024) // 创建1024大小的缓冲区，用于read  
n, err := conn.Read(buf)  
 **if** err != nil {  
 fmt.Println(**"read err:"**, err)  
 **return** }  
 fmt.Println(**"服务器读到:"**, string(buf[:n])) // 读多少，打印多少。}

如图，在整个通信过程中，服务器端有两个socket参与进来，但用于通信的只有 conn 这个socket。它是由 listener创建的。隶属于服务器端。



Client 端：

Dial函数：

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

network：选用的协议：TCP、UDP，如：“tcp”或 “udp”

address：**服务器**IP地址+端口号, 如：“121.36.108.11:8000”或 “www.itcast.cn:8000”

Conn 接口：

type Conn interface {

**Read**(b []byte) (n int, err error)

**Write**(b []byte) (n int, err error)

**Close**() error

LocalAddr() Addr

RemoteAddr() Addr

SetDeadline(t time.Time) error

SetReadDeadline(t time.Time) error

SetWriteDeadline(t time.Time) error

}

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 // 主动发起连接请求  
conn, err := net.Dial(**"tcp"**, **"127.0.0.1:8000"**) // tcp 不能使用大写  
 **if** err != nil {  
 fmt.Println(**"Dial err:"**, err)  
 **return** }  
 **defer** conn.Close() // 结束时，关闭连接  
 // 发送数据  
\_, err = conn.Write([]byte(**"Are u ready?"**))   
 **if** err != nil {  
 fmt.Println(**"Write err:"**, err)  
 **return** }  
}

### 并发C/S模型通信

并发Server

现在已经完成了客户端与服务端的通信，但是服务端只能接收一个用户发送过来的数据，怎样接收多个客户端发送过来的数据，实现一个高效的并发服务器呢？

Accept()函数的作用是等待客户端的链接，如果客户端没有链接，该方法会阻塞。如果有客户端链接，那么该方法返回一个Socket负责与客户端进行通信。所以，每来一个客户端，该方法就应该返回一个Socket与其通信，因此，可以使用一个死循环，将Accept()调用过程包裹起来。

需要注意的是，实现并发处理多个客户端数据的服务器，就需要**针对每一个客户端连接，单独产生一个Socket，并创建一个单独的goroutine与之完成通信。**

//监听

    listener, err := net.Listen("tcp", "127.0.0.1:8001") // tcp 不能使用大写

**if** err != nil {

         fmt.Println("err = ", err)

**return**

    }

**defer** listener.Close()

    //接收多个用户

**for** {

         conn, err := listener.Accept()

**if** err != nil {

           fmt.Println("err = ", err)

**return**

         }

         //处理用户请求, 新建一个go程

**go** HandleConn(conn)

}

将客户端的数据处理工作封装到HandleConn方法中，需将Accept()返回的Socket传递给该方法，变量conn的类型为：net.Conn。可以使用conn.RemoteAddr()来获取成功与服务器建立连接的客户端IP地址和端口号：

Conn 接口：

type Conn interface {

Read(b []byte) (n int, err error)

Write(b []byte) (n int, err error)

Close() error

LocalAddr() Addr

**RemoteAddr**() Addr

SetDeadline(t time.Time) error

SetReadDeadline(t time.Time) error

SetWriteDeadline(t time.Time) error

}

    //获取客户端的网络地址信息

    addr := conn.RemoteAddr().String()

    fmt.Println(addr, " conncet sucessful")

客户端可能持续不断的发送数据，因此接收数据的过程可以放在for循环中，服务端也持续不断的向客户端返回处理后的数据。

添加一个限定，如果客户端发送一个“exit”字符串，表示客户端通知服务器不再向服务端发送数据，此时应该结束HandleConn方法，同时关闭与该客户端关联的Socket。

buf := make([]byte, 2048)  //创建一个切片，存储客户端发送的数据

**for** {

        //读取用户数据

        n, err := conn.Read(buf)

**if** err != nil {

            fmt.Println("err = ", err)

**return**

        }

        fmt.Printf("[%s]: %s\n", addr, string(buf[:n]))

**if** "exit" == string(buf[:n-2]) { //自己写的客户端测试, 发送时，多了2个字符, "\r\n"

            fmt.Println(addr, " exit")

**return**

        }

        //服务器处理数据：把客户端数据转大写，再写回给client

        conn.Write([]byte(strings.ToUpper(string(buf[:n]))))

 }

在上面的代码中，Read()方法获取客户端发送过来的数据，填充到切片buf中，返回的是实际填充的数据的长度，所以将客户端发送过来的数据进行打印，打印的是实际接收到的数据。

fmt.Printf("[%s]: %s\n", addr, string(buf[:n])).同时也可以将客户端的网络地址信息打印出来。

在判断客户端数据是否为“exit”字符串时，要注意，客户端会自动的多发送2个字符：“\r\n”（这在windows系统下代表回车、换行）

Server使用Write方法将数据写回给客户端，参数类型是 []byte，需使用strings包下的ToUpper函数来完成大小写转换。转换的对象即为string(buf[:n])

综上，HandleConn方法完整定义如下：

//处理用户请求

**func** HandleConn(conn net.Conn) {

//函数调用完毕，自动关闭conn

**defer** conn.Close()

     //获取客户端的网络地址信息

     addr := conn.RemoteAddr().String()

      fmt.Println(addr, " conncet sucessful")

     buf := make([]byte, 2048)

**for** {

         //读取用户数据

         n, err := conn.Read(buf)

**if** err != nil {

             fmt.Println("err = ", err)

**return**

         }

          fmt.Printf("[%s]: %s\n", addr, string(buf[:n]))

          fmt.Println("len = ", len(string(buf[:n])))

         //if "exit" == string(buf[:n-1]) { // nc测试，发送时，只有 \n

**if** "exit" == string(buf[:n-2]) { // 自己写的客户端测试, 发送时，多了2个字符, "\r\n"

             fmt.Println(addr, " exit")

**return**

         }

         //把数据转换为大写，再给用户发送

         conn.Write([]byte(strings.ToUpper(string(buf[:n]))))

     }

}

并发Client

客户端不仅需要持续的向服务端发送数据，同时也要接收从服务端返回的数据。因此可将发送和接收放到不同的go程中。

主go程循环接收服务器回发的数据（该数据应已转换为大写），并打印至屏幕；子go程循环从键盘读取用户输入数据，写给服务器。读取键盘输入可使用 os.Stdin.Read(str)。定义切片str，将读到的数据保存至str中。

这样，客户端也实现了多任务。

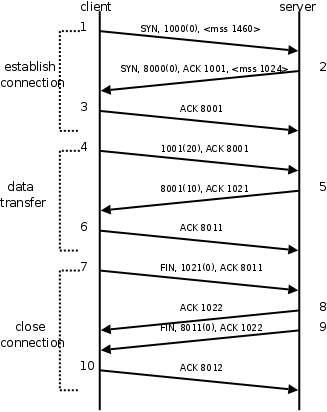
客户端代码实现：

**package** main  
  
**import** (  
 **"net"  
 "fmt"  
 "os"**)  
  
**func** main() {  
 // 主动发起连接请求  
conn, err := net.Dial(**"tcp"**, **"127.0.0.1:8001"**)  
 **if** err != nil {  
 fmt.Println(**"Dial err:"**, err)  
 **return** }  
 **defer** conn.Close() // 客户端终止时，关闭与服务器通信的 socket   
 // 启动子go程，接收用户键盘输入  
**go func**() {

str := make([]byte, 1024) // 创建用于存储用户键盘输入数据的切片缓冲区。  
 **for** { // 反复读取  
n, err :=os.Stdin.Read(str) // 获取用户键盘输入  
**if** err != nil {  
 fmt.Println(**"os.Stdin.Read err:"**, err)  
 **return** }  
 // 将从键盘读到的数据，发送给服务器  
\_, err = conn.Write(str[:n]) // 读多少，写多少  
**if** err != nil {  
 fmt.Println(**"conn.Write err:"**, err)  
 **return** }  
 }  
 }()  
  
 // 主go程，接收服务器回发数据，打印至屏幕  
buf := make([]byte, 1024) // 定义用于存储服务器回发数据的切片缓冲区  
**for** {  
 n, err := conn.Read(buf) // 从通信 socket 中读数据，存入切片缓冲区  
 **if** err != nil {  
 fmt.Println(**"conn.Read err:"**, err)  
 **return** }  
 fmt.Printf(**"服务器回发：%s\n"**, string(buf[:n]))  
 }  
}

### TCP通信过程

下图是一次TCP通讯的时序图。TCP连接建立断开。包含大家熟知的**三次握手**和四次握手。



在这个例子中，首先客户端主动发起连接、发送请求，然后服务器端响应请求，然后客户端主动关闭连接。两条竖线表示通讯的两端，从上到下表示时间的先后顺序。注意，数据从一端传到网络的另一端也需要时间，所以图中的箭头都是斜的。

三次握手

所谓三次握手（Three-Way Handshake）即建立TCP连接，就是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。好比两个人在打电话：

Client:“喂，你听得到吗？”

Server:“我听得到，你听得到我吗？”

Client:“我能听到你，今天balabala…”

建立连接（三次握手）的过程：

1. 客户端发送一个带SYN标志的TCP报文到服务器。这是上图中三次握手过程中的段1。客户端发出SYN位表示连接请求。序号是1000，这个序号在网络通讯中用作临时的地址，每发一个数据字节，这个序号要加1，这样在接收端可以根据序号排出数据包的正确顺序，也可以发现丢包的情况。

另外，规定SYN位和FIN位也要占一个序号，这次虽然没发数据，但是由于发了SYN位，因此下次再发送应该用序号1001。

mss表示最大段尺寸，如果一个段太大，封装成帧后超过了链路层的最大长度，就必须在IP层分片，为了避免这种情况，客户端声明自己的最大段尺寸，建议服务器端发来的段不要超过这个长度。

1. 服务器端回应客户端，是三次握手中的第2个报文段，同时带ACK标志和SYN标志。表示对刚才客户端SYN的回应；同时又发送SYN给客户端，询问客户端是否准备好进行数据通讯。

服务器发出段2，也带有SYN位，同时置ACK位表示确认，确认序号是1001，表示“我接收到序号1000及其以前所有的段，请你下次发送序号为1001的段”，也就是应答了客户端的连接请求，同时也给客户端发出一个连接请求，同时声明最大尺寸为1024。

1. 客户必须再次回应服务器端一个ACK报文，这是报文段3。

客户端发出段3，对服务器的连接请求进行应答，确认序号是8001。在这个过程中，客户端和服务器分别给对方发了连接请求，也应答了对方的连接请求，其中服务器的请求和应答在一个段中发出。

因此一共有三个段用于建立连接，称为“三方握手”。在建立连接的同时，双方协商了一些信息，例如，双方发送序号的初始值、最大段尺寸等。

数据传输的过程：

1. 客户端发出段4，包含从序号1001开始的20个字节数据。
2. 服务器发出段5，确认序号为1021，对序号为1001-1020的数据表示确认收到，同时请求发送序号1021开始的数据，服务器在应答的同时也向客户端发送从序号8001开始的10个字节数据。
3. 客户端发出段6，对服务器发来的序号为8001-8010的数据表示确认收到，请求发送序号8011开始的数据。

在数据传输过程中，ACK和确认序号是非常重要的，应用程序交给TCP协议发送的数据会暂存在TCP层的发送缓冲区中，发出数据包给对方之后，只有收到对方应答的ACK段才知道该数据包确实发到了对方，可以从发送缓冲区中释放掉了，如果因为网络故障丢失了数据包或者丢失了对方发回的ACK段，经过等待超时后TCP协议自动将发送缓冲区中的数据包重发。

四次挥手

所谓四次挥手（Four-Way-Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务器任一方执行close来触发。好比两个人打完电话要挂断：

Client:“我要说的事情都说完了，我没事了。挂啦？”

Server:“等下，我还有一个事儿。Balabala…”

Server:“好了，我没事儿了。挂了啊。”

Client:“ok！拜拜”

关闭连接（四次握手）的过程：

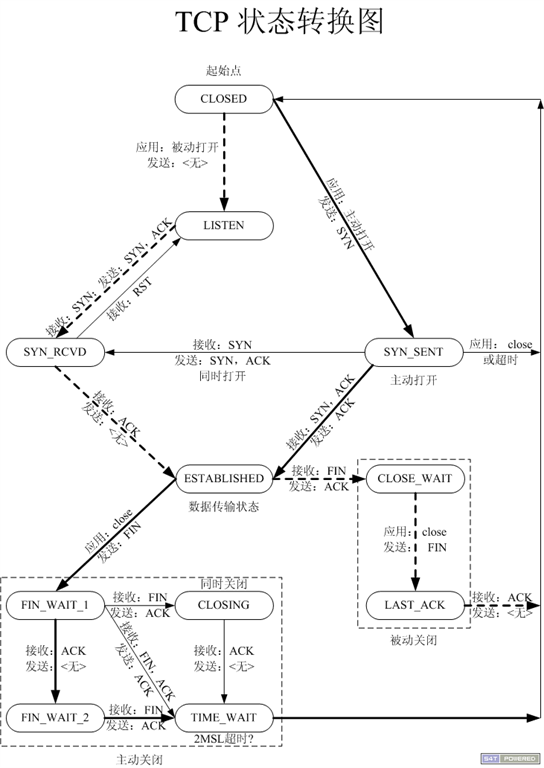
由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

1. 客户端发出段7，FIN位表示关闭连接的请求。
2. 服务器发出段8，应答客户端的关闭连接请求。
3. 服务器发出段9，其中也包含FIN位，向客户端发送关闭连接请求。
4. 客户端发出段10，应答服务器的关闭连接请求。

建立连接的过程是三次握手，而关闭连接通常需要4个段，服务器的应答和关闭连接请求通常不合并在一个段中，因为有连接半关闭的情况，这种情况下客户端关闭连接之后就不能再发送数据给服务器了，但是服务器还可以发送数据给客户端，直到服务器也关闭连接为止。

TCP状态转换

TCP状态图很多人都知道，它对排除和定位网络或系统故障时大有帮助。如果能熟练掌握这张图，了解图中的每一个状态，能大大提高我们对于TCP的理解和认识。下面对这张图的11种状态详细解析一下，以便加强记忆！不过在这之前，一定要熟练掌握TCP建立连接的三次握手过程，以及关闭连接的四次挥手过程。



**CLOSED：**表示初始状态。

**LISTEN：**该状态表示服务器端的某个SOCKET处于监听状态，可以接受连接。

**SYN\_SENT：**这个状态与SYN\_RCVD遥相呼应，当客户端SOCKET执行CONNECT连接时，它首先发送SYN报文，随即进入到了SYN\_SENT状态，并等待服务端的发送三次握手中的第2个报文。SYN\_SENT状态表示客户端已发送SYN报文。

**SYN\_RCVD:** 该状态表示接收到SYN报文，在正常情况下，这个状态是服务器端的SOCKET在建立TCP连接时的三次握手会话过程中的一个中间状态，很短暂。此种状态时，当收到客户端的ACK报文后，会进入到ESTABLISHED状态。

**ESTABLISHED：**表示连接已经建立。

**FIN\_WAIT\_1:** FIN\_WAIT\_1和FIN\_WAIT\_2状态的真正含义都是表示等待对方的FIN报文。区别是：

FIN\_WAIT\_1状态是当socket在ESTABLISHED状态时，想主动关闭连接，向对方发送了FIN报文，此时该socket进入到FIN\_WAIT\_1状态。

FIN\_WAIT\_2状态是当对方回应ACK后，该socket进入到FIN\_WAIT\_2状态，正常情况下，对方应马上回应ACK报文，所以FIN\_WAIT\_1状态一般较难见到，而FIN\_WAIT\_2状态可用netstat看到。

**FIN\_WAIT\_2：主动关闭链接的一方，发出FIN收到ACK以后进入该状态。称之为半连接或半关闭状态。**该状态下的socket只能接收数据，不能发。

**TIME\_WAIT:** 表示收到了对方的FIN报文，并发送出了ACK报文，等2MSL后即可回到CLOSED可用状态。如果FIN\_WAIT\_1状态下，收到对方同时带 FIN标志和ACK标志的报文时，可以直接进入到TIME\_WAIT状态，而无须经过FIN\_WAIT\_2状态。

**CLOSING:** 这种状态较特殊，属于一种较罕见的状态。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的 ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。什么情况下会出现此种情况呢？如果双方几乎在同时close一个SOCKET的话，那么就出现了双方同时发送FIN报文的情况，也即会出现CLOSING状态，表示双方都正在关闭SOCKET连接。

**CLOSE\_WAIT:** 此种状态表示在等待关闭。当对方关闭一个SOCKET后发送FIN报文给自己，系统会回应一个ACK报文给对方，此时则进入到CLOSE\_WAIT状态。接下来呢，察看是否还有数据发送给对方，如果没有可以 close这个SOCKET，发送FIN报文给对方，即关闭连接。所以在CLOSE\_WAIT状态下，需要关闭连接。

**LAST\_ACK:** 该状态是被动关闭一方在发送FIN报文后，最后等待对方的ACK报文。当收到ACK报文后，即可以进入到CLOSED可用状态。

2MSL (Maximum Segment Lifetime) 和与之对应的TIME\_WAIT状态，可以**让4次握手关闭流程更加可靠。**4次握手的最后一个ACK是是由主动关闭方发送出去的，若这个ACK丢失，被动关闭方会再次发一个FIN过来。若主动关闭方能够保持一个2MSL的TIME\_WAIT状态，则有更大的机会让丢失的ACK被再次发送出去。注意，TIME\_WAIT状态一定出现在**主动关闭这一方**。

### UDP通信

在之前的案例中，我们一直使用的是TCP协议来编写Socket的客户端与服务端。其实也可以使用UDP协议来编写Socket的客户端与服务端。

UDP与TCP的差异

|  |  |
| --- | --- |
| TCP | UDP |
| 面向连接 | 面向无连接 |
| 要求系统资源较多 | 要求系统资源较少 |
| TCP程序结构较复杂 | UDP程序结构较简单 |
| 使用流式 | 使用数据包式 |
| 保证数据准确性 | 不保证数据准确性 |
| 保证数据顺序 | 不保证数据顺序 |
| 通讯速度较慢 | 通讯速度较快 |

UDP服务器

由于UDP是“无连接”的，所以，服务器端不需要额外创建监听套接字，只需要指定好IP和port，然后监听该地址，等待客户端与之建立连接，即可通信。

创建监听地址：

**func** ResolveUDPAddr(network, address string) (\*UDPAddr, error)

创建用于通信的socket：

**func** ListenUDP(network string, laddr \*UDPAddr) (\*UDPConn, error)

接收udp数据：

**func** (c \*UDPConn) ReadFromUDP(b []byte) (int, \*UDPAddr, error)

写出数据到udp：

**func** (c \*UDPConn) WriteToUDP(b []byte, addr \*UDPAddr) (int, error)

服务端完整代码实现如下：

**package** main  
  
**import** (  
 **"fmt"  
 "net"**)  
  
**func** main() {  
 //创建监听的地址，并且指定udp协议  
udp\_addr, err := net.ResolveUDPAddr(**"udp"**, **"127.0.0.1:8002"**)  
 **if** err != nil {  
 fmt.Println(**"ResolveUDPAddr err:"**, err)  
 **return** }  
 conn, err := net.ListenUDP(**"udp"**, udp\_addr) // 创建数据通信socket  
**if** err != nil {  
 fmt.Println(**"ListenUDP err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 buf := make([]byte, 1024)  
 n, raddr, err := conn.ReadFromUDP(buf) //接收客户端发送过来的数据，填充到切片buf中。**if** err != nil {  
 **return** }  
 fmt.Println(**"客户端发送："**, string(buf[:n]))  
  
 \_, err = conn.WriteToUDP([]byte(**"nice to see u in udp"**), raddr) // 向客户端发送数据  
**if** err != nil {  
 fmt.Println(**"WriteToUDP err:"**, err)  
 **return** }  
}

UDP客户端

udp客户端的编写与TCP客户端的编写，基本上是一样的，只是将协议换成udp。注意只能使用小写。

代码如下：

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 conn, err := net.Dial(**"udp"**, **"127.0.0.1:8002"**)   
 **if** err != nil {  
 fmt.Println(**"net.Dial err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 conn.Write([]byte(**"Hello! I'm client in UDP!"**))  
  
 buf := make([]byte, 1024)  
 n, err1 := conn.Read(buf)  
 **if** err1 != nil {  
 **return** }  
 fmt.Println(**"服务器发来："**, string(buf[:n]))  
}

并发

其实对于UDP而言，服务器不需要并发，只要循环处理客户端数据即可。客户端也等同于TCP通信并发的客户端。

服务器：

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 *// 创建 服务器 UDP 地址结构。指定 IP + port* laddr, err := net.ResolveUDPAddr(**"udp"**, **"127.0.0.1:8003"**)  
 **if** err != nil {  
 fmt.Println(**"ResolveUDPAddr err:"**, err)  
 **return** }  
 *// 监听 客户端连接* conn, err := net.ListenUDP(**"udp"**, laddr)  
 **if** err != nil {  
 fmt.Println(**"net.ListenUDP err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 **for** {  
 buf := make([]byte, 1024)  
 n, raddr, err := conn.ReadFromUDP(buf)  
 **if** err != nil {  
 fmt.Println(**"conn.ReadFromUDP err:"**, err)  
 **return** }  
 fmt.Printf(**"接收到客户端[%s]：%s"**, raddr, string(buf[:n]))  
  
 conn.WriteToUDP([]byte(**"I-AM-SERVER"**), raddr) *// 简单回写数据给客户端* }  
}

客户端：

**package** main  
  
**import** (  
 **"net"  
 "os"  
 "fmt"**)  
  
**func** main() {  
 conn, err := net.Dial(**"udp"**, **"127.0.0.1:8003"**)  
 **if** err != nil {  
 fmt.Println(**"net.Dial err:"**, err)  
 **return** }  
 **defer** conn.Close()  
 **go func**() {  
 str := make([]byte, 1024)  
 **for** {  
 n, err := os.Stdin.Read(str) *//从键盘读取内容， 放在str* **if** err != nil {  
 fmt.Println(**"os.Stdin. err1 = "**, err)  
 **return** }  
 conn.Write(str[:n]) *// 给服务器发送* }  
 }()  
 buf := make([]byte, 1024)  
 **for** {  
 n, err := conn.Read(buf)  
 **if** err != nil {  
 fmt.Println(**"conn.Read err:"**, err)  
 **return** }  
 fmt.Println(**"服务器写来："**, string(buf[:n]))  
 }  
}

### 网络文件传输

#### 流程简析

借助TCP完成文件的传输，基本思路如下：

1：发送方（客户端）向服务端发送文件名，服务端保存该文件名。

2：接收方（服务端）向客户端返回一个消息ok，确认文件名保存成功。

3：发送方（客户端）收到消息后，开始向服务端发送文件数据。

4：接收方（服务端）读取文件内容，写入到之前保存好的文件中。



#### 获取文件属性和命令行参数

首先获取文件名。借助os包中的stat()函数来获取文件属性信息。在函数返回的文件属性中包含文件名和文件大小。Stat参数name传入的是文件访问的绝对路径。FileInfo中的Name()函数可以将不含路径的文件名单独提取出来。

**func** Stat(name string) (**FileInfo**, error)

**type FileInfo** **interface** {  
 **Name()** string

**Size()** int64

Mode() FileMode ModTime() time.Time IsDir() bool Sys() **interface**{} }

获取文件属性示例：

**package** main  
  
**import** (  
 **"os"  
 "fmt"**)  
  
**func** main() {  
 list := os.Args *// 获取命令行参数，存入list中* **if** len(list) != 2 {  *// 确保用户输入了一个命令行参数* fmt.Println(**"格式为：xxx.go 文件名"**)  
 **return** }  
 fileName := list[1] *// 从命令行保存文件名(含路径)* fileInfo, err := os.Stat(fileName) *//根据文件名获取文件属性信息 fileInfo* **if** err != nil {  
 fmt.Println(**"os.Stat err:"**, err)  
 **return** }  
 fmt.Println(**"文件name为："**, fileInfo.Name()) *// 得到文件名(不含路径)* fmt.Println(**"文件size为："**, fileInfo.Size()) *// 得到文件大小。单位字节*}

发送端实现

实现流程大致如下：

1. 提示用户输入文件名。接收文件名path（含访问路径）
2. 使用os.Stat()获取文件属性，得到纯文件名（去除访问路径）
3. 主动连接服务器，结束时关闭连接
4. 给接收端（服务器）发送文件名conn.Write()
5. 读取接收端回发的确认数据conn.Read()
6. 判断是否为“ok”。如果是，封装函数SendFile() 发送文件内容。传参path和conn
7. 只读Open文件, 结束时Close文件
8. 循环读文件，读到EOF终止文件读取
9. 将读到的内容原封不动Write给接收端（服务器）

代码实现：

**package** main  
  
**import** (  
 **"fmt"  
 "os"  
 "net"  
 "io"**)  
  
**func** SendFile(path string, conn net.Conn) {  
 *// 以只读方式打开文件* f, err := os.Open(path)  
 **if** err != nil {  
 fmt.Println(**"os.Open err:"**, err)  
 **return** }  
 **defer** f.Close() *// 发送结束关闭文件。  
  
 // 循环读取文件，原封不动的写给服务器* buf := make([]byte, 4096)  
 **for** {  
 n, err := f.Read(buf) *// 读取文件内容到切片缓冲中* **if** err != nil {  
 **if** err == io.EOF {  
 fmt.Println(**"文件发送完毕"**)  
 } **else** {  
 fmt.Println(**"f.Read err:"**, err)  
 }  
 **return** }  
 conn.Write(buf[:n]) *// 原封不动写给服务器* }  
}  
  
**func** main() {  
 *// 提示输入文件名* fmt.Println(**"请输入需要传输的文件："**)  
 **var** path string  
 fmt.Scan(&path)  
  
 *// 获取文件名 fileInfo.Name()* fileInfo, err := os.Stat(path)  
 **if** err != nil {  
 fmt.Println(**"os.Stat err:"**, err)  
 **return** }  
  
 *// 主动连接服务器* conn, err := net.Dial(**"tcp"**, **"127.0.0.1:8005"**)  
 **if** err != nil {  
 fmt.Println(**"net.Dial err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 *// 给接收端，先发送文件名* \_, err = conn.Write([]byte(fileInfo.Name()))  
 **if** err != nil {  
 fmt.Println(**"conn.Write err:"**, err)  
 **return** }  
  
 *// 读取接收端回发确认数据 —— ok* buf := make([]byte, 1024)  
 n, err := conn.Read(buf)  
 **if** err != nil {  
 fmt.Println(**"conn.Read err:"**, err)  
 **return** }  
  
 *// 判断如果是ok，则发送文件内容* **if "ok"** == string(buf[:n]) {  
 SendFile(path, conn) *// 封装函数读文件，发送给服务器，需要path、conn* }  
}

接收端实现

实现流程大致如下：

1. 创建监听listener，程序结束时关闭。
2. 阻塞等待客户端连接，程序结束时关闭conn。
3. 读取客户端发送文件名。保存fileName。
4. 回发“ok”给客户端做应答
5. 封装函数 RecvFile接收客户端发送的文件内容。传参fileName 和conn
6. 按文件名Create文件，结束时Close
7. 循环Read客户端发送的文件内容，当读到EOF说明文件读取完毕。
8. 将读到的内容原封不动Write到创建的文件中

代码实现：

**package** main  
  
**import** (  
 **"net"  
 "fmt"  
 "os"  
 "io"**)  
  
**func** RecvFile(fileName string, conn net.Conn) {  
 *// 创建新文件* f, err := os.Create(fileName)  
 **if** err != nil {  
 fmt.Println(**"Create err:"**, err)  
 **return** }  
 **defer** f.Close()  
  
 *// 接收客户端发送文件内容，原封不动写入文件* buf := make([]byte, 4096)  
 **for** {  
 n, err := conn.Read(buf)  
 **if** err != nil {  
 **if** err == io.EOF {  
 fmt.Println(**"文件接收完毕"**)  
 } **else** {  
 fmt.Println(**"Read err:"**, err)  
 }  
 **return** }  
 f.Write(buf[:n]) *// 写入文件，读多少写多少* }  
}  
  
**func** main() {  
 *// 创建监听* listener, err := net.Listen(**"tcp"**, **"127.0.0.1:8005"**)  
 **if** err != nil {  
 fmt.Println(**"Listen err:"**, err)  
 **return** }  
 **defer** listener.Close()  
  
 *// 阻塞等待客户端连接* conn, err := listener.Accept()  
 **if** err != nil {  
 fmt.Println(**"Accept err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 *// 读取客户端发送的文件名* buf := make([]byte, 1024)  
 n, err := conn.Read(buf)  
 **if** err != nil {  
 fmt.Println(**"Read err:"**, err)  
 **return** }  
 fileName := string(buf[:n]) *// 保存文件名  
  
 // 回复 0k 给发送端* conn.Write([]byte(**"ok"**))  
  
 *// 接收文件内容* RecvFile(fileName, conn) *// 封装函数接收文件内容， 传fileName 和 conn*}

### 总结：

网络应用程序设计模式：

优点 缺点

C/S： 缓冲大量数据、协议选择灵活 用户安全性较差、不能跨平台、开发工作量大

B/S： 用户安全性较好、能跨平台、开发工作量小 不能缓冲大量数据、协议选择受浏览器限制

socket建立流程

Server client

net.Listen

Accept net.Dial

Read Read

Write Write

Close Close

socket注意点

Server：

1. 并发服务器Accept错误应该continue，等待下一个链接，不应该是return
2. 封装handlerConn函数，使用goroutine调用
3. handlerConn需要注册defer close(conn)

Client:

1. os.Stdin.Read() 解决fmt.Scan()以空格为结束符的问题

【网络socket通信，重要结论】：

1. conn.Read() 如果读到 0 说明对端关闭！！！！
2. 读多少写多少 buf[:n]

三次握手：

主动建立连接请求端（客户端）， 发送 SYN 标志位， 携带 序号

被动接受连接请求端（服务器）， 接收 SYN 标志，发送 SYN 标志位，携带序号, 回发 ACK 携带 确认序号，

主动建立连接请求端（客户端）， 接收 SYN 标志， 回发 ACK 携带 确认序号。 —— 标志 3 次握手完成。

标志 3 次握手完成： 对应应用层：客户端：Dial() 返回。

服务器：Accept(） 反回。

TCP 数据通信：

发送端发送数据同时 ，携带序号。接收端，收到数据后 需要给发送端发送 ACK 应答以及 确认序号 （回执）。

确保TCP通信模式下，数据可靠的传输。

批量回执。tcp传输不丢包（错误，丢包是网络硬件层面的问题，不可避免，tcp丢包邮重传机制）

滑动窗口：实时通知对端，本端存储数据的缓冲区大小。

四次挥手：

主动断开连接请求端（客户端）， 发送 FIN 标志位， 携带 序号。

被动断开连接请求端（服务器）， 接收 FIN 标志位，回发 ACK 应答。携带确认序号。

—— 半关闭完成。

被动断开连接请求端（服务器），发送 FIN 标志位， 携带 序号。

主动断开连接请求端（客户端）， 接收 FIN 标志位，回发 ACK 应答。携带确认序号。

—— 最后一个 ACK 被 接收到 以后，4次挥手断开连接完成。

2MSL:

为了确保 tcp通信过程中的 “最后一个ACK”能被对端收到。等待的一个时长( 约 40s )。

TCP 状态转换图：

主动连接端（客户端）：

CLOSED --> 发送 SYN --> SYN\_SENT --> 接收 ACK、SYN，发送 ACK --> ESTABLISHED --> 数据通信

主动关闭端（客户端）：

ESTABLISHED --> 发送 FIN --> FIN\_WAIT\_1 --> 接收 ACK --> FIN\_WAIT\_2 (半关闭) -->

接收 FIN，发送 ACK --> TIME\_WAIT --> 等待 2MSL 时长 --> CLOSED

—— FIN\_WAIT\_2、TIME\_WAIT、2MSL 只出现 主动端。

被动连接端（服务器）：

CLOSED --> LISTEN --> 接收SYN, 发送 ACK, SYN --> SYN\_RCVD --> 接收 ACK --> ESTABLISHED --> 数据通信

被动关闭端（服务器）：

ESTABLISHED --> 接收 FIN, 发送ACK --> CLOSE\_WAIT (对应主动端的 FIN\_WAIT\_2) --> 发送 FIN --> LAST\_ACK

--> 接收 ACK --> CLOSED

UDP：

UDP-CS-server:

1. 创建 UDP地址结构：

net.ResolvUDPAddr("udp", "IP:port") ——> udpAddr

2. 创建 通信 socket

net.ListenUDP("udp", udpAddr) ——> udpConn

3. 读取客户端发送数据

udpConn.ReadFromUDP(buf) ——> n, clitAddr, err

4. 发送数据给 客户端

udpConn.WriteToUDP([]byte(), clitAddr) ——> n, err

5. 启动命令行，使用 nc 127.0.0.1 8003 -u 充当客户端测试。

UDP-CS-client:

1. 创建 通信 socket

net.Dial("udp", 服务器端的IP+port) ——> udpConn

2. 剩余代码，参考 TCP-CS-client 实现

UDP并发：

UDP 服务器 默认 就支持客户端并发访问。可以把read和处理数据的部分起一个匿名go程来提高效率

tcp udp对比：

|  |  |
| --- | --- |
| TCP | UDP |
| 面向连接 | 面向无连接 |
| 要求系统资源较多 | 要求系统资源较少 |
| TCP程序结构较复杂 | UDP程序结构较简单 |
| 使用流式 | 使用数据包式 |
| 保证数据准确性 | 不保证数据准确性 |
| 保证数据顺序 | 不保证数据顺序 |
| 通讯速度较慢 | 通讯速度较快 |

TCP：面向连接的可靠的数据包传递。 针对 不稳定的 网络层 做完全弥补。—— 借助回执，丢包重传

UDP：无连接的不可靠的报文传输。 针对 不稳定的 网络层 直接还原真实状态。 —— 丢包，不处理。

优 点 缺 点

TCP： 稳定（顺序、速度）、可靠的 系统资源占用多。发送速度慢。开发难度大

UDP： 系统资源占用少。发送速度块。开发难度小 不稳定（顺序、速度）、不可靠

使用场景：

TCP： 对数据稳定、准确性要求较高的场合。 上传、下载。

UDP：对数据的传输速度要求较高的场合，允许适当的数据丢失。 游戏、视频直播、在线电话会议。

网络文件传输：

获取文件属性函数：

fileInfo：os.stat(文件访问绝对路径)

fileInfo 接口 内包含常用两个 interface 。

Name() 获取文件名。 不带访问路径

Size() 获取文件大小。

命令行参数： 在 main 函数启动时，向整个程序传参。 【重点】

flag包

os.Args

语法： go run xxx.go argv1 argv2 argv3 argv4 。。。

xxx.exe: 第 0 个参数。

argv1 ：第 1 个参数。

argv2 ：第 2 个参数。

argv3 ：第 3 个参数。

argv4 ：第 4 个参数。

使用： list := os.Args 提取所有命令行参数。

/home/itcast/2go/libai.mp3

# 并发聊天室

并发编程和网络编程是现今行业开发中常用的技术。Go语言强大的语法设定使得并发和网络编程都变的简洁而高效。

下面我们利用前面学到的知识，使用并发和网络实现一个简单的网络在线聊天室。体会下这两种技术的实际应用。在整个聊天室的项目中，充分利用了go程并发，处理不同任务。

整个聊天室程序可简单划分为如下模块，都分别使用go程来实现：

**主go程（服务器）：**

负责监听、接收用户（客户端）连接请求，建立通信关系。同时启动相应的go程处理任务。

**处理连接用户数据go程：HandleConnect**

负责新上线用户的存储，用户消息读取、发送，用户改名、下线处理及超时处理。

为了提高并发效率，同时给一个用户维护多个go程来并行处理上述任务。

**用户消息广播go程：Manager**

负责在线用户遍历，用户消息广播发送。需要与**HandleConnect** go程及用户子go程协作完成。

**go程间应用数据及通信：**

map：存储所有登录聊天室的用户信息， key：用户的ip+port。Value：Client结构体。

Client结构体：包含成员：用户名Name，网络地址Addr（ip+port），发送消息的通道C（channel）

通道message：协调并发go程间消息的传递。

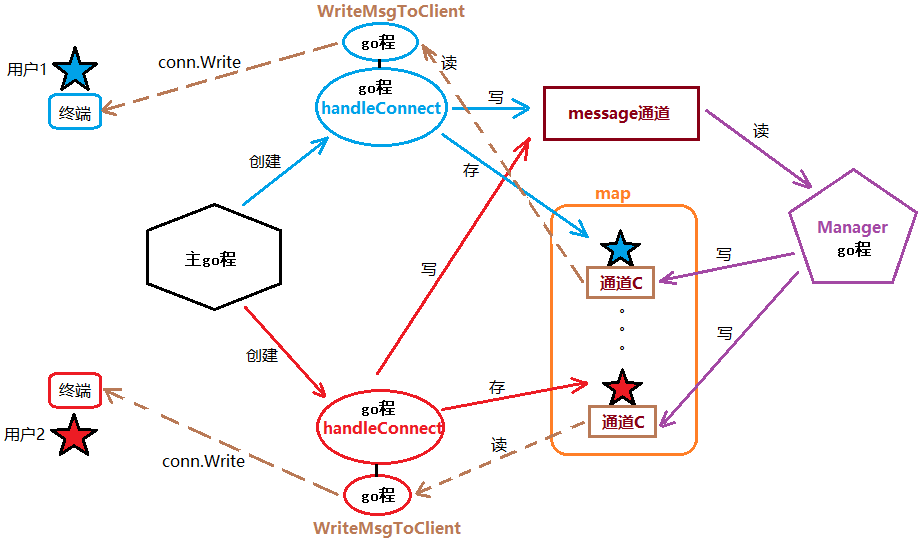
## 广播用户上线

首先，服务器启动，等待用户建立通信连接。当有用户连接上来，将其存储到map中，这样就维护了一个“在线用户”的列表。当再有新用户连接上来时，应向该列表中所有用户进行广播通知，提示xxx用户上线。

当然，简单实现手法可以循环读取列表中的用户，依次向其发送消息通知新用户上线。但这种方式无疑是一种串行的通信手段，实现简单，但执行效率较低。

在go语言中，我们利用go程轻便、高效、并发性好的特性，给每个登录用户维护多个go程来进行数据通信，借助channel不需要使用同步锁，就可以实现高效的并发通信。

下图充分利用goroutine和channel实现了新用户登录，向所有在线用户进行广播通知：



分析上图，主要分为几大模块。

全局位置定义用户结构体类型 Client，存储登录用户信息。成员包含channel、Name、Addr

type Client struct {

C chan string

Name string

Addr string

}

定义全局通道message处理消息。

定义全局map 存储在线用户信息。Key为用户网络地址。Value为用户结构体。

主go程，监听客户端连接请求，当有新的客户端连接，创建新go程handleConnet处理用户连接。

handleConnet go程，获取用户网络地址（Ip+port），创建新用户结构体，包含成员C、Name、Addr。新用户的Name和Addr初值都是用户网络地址（Ip+port）。将用户结构体存入map中。并创建WriteMsgToClient go程，专门负责给当前用户发送消息。组织新用户上线广播消息内容，写入全局通道message中。

WriteMsgToClient go程，读取用户结构体C中的数据，没有则阻塞等待，有数据写出给登录用户。

Manager go程，给map分配空间。循环读取 message 通道中是否有数据。没有，阻塞等待。有则解除阻塞，将message通道中读到的数据写到用户结构体中的C通道。

代码实现：

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
*// 定义用户结构体类型***type** Client **struct** {  
 C **chan** string  
 Name string  
 Addr string  
}  
*// 定义全局 map 存储在线用户 key:IP+port, value:Client***var** onlineMap **map**[string]Client  
  
*// 定义全局 channel 处理消息***var** message = make(**chan** string)  
  
**func** WriteMsgToClient(clnt Client, conn net.Conn) {  
 *// 循环跟踪 clnt.C，有消息则读走，Write 给客户端* **for** msg := **range** clnt.C {  
 conn.Write([]byte(msg + **"\n"**)) *// 发送消息 给客户端* }  
}  
  
**func** MakeMsg(clnt Client, msg string) (buf string) {  
 buf = **"["** + clnt.Addr + **"]"** + clnt.Name + **": "** + msg  
 **return**}  
  
**func** HandleConnect(conn net.Conn) {  
 **defer** conn.Close()  
 *// 获取新连接上来的用户的网络地址(IP+port)* netAddr := conn.RemoteAddr().String()  
 *// 给新用户创建结构体。用户名、网络地址一样* clnt := Client{make(**chan** string), netAddr, netAddr}  
 *// 将新创建的结构体，添加到 map 中，key值为获取到的网络地址（IP+port）* onlineMap[netAddr] = clnt  
  
 *// 新创建一个go程，专门给当前客户端发送消息。* **go** WriteMsgToClient(clnt, conn)  
  
 *// 广播新用户上线  
 // message <- "[" + clnt.Addr + "]" + clnt.Name + ": login"* message <- MakeMsg(clnt, **"login"**)  
  
 **for** { *// 不能让当前go程结束。* ;  
 }  
}  
  
**func** Manager() {  
 *// 给map分配空间* onlineMap = make(**map**[string]Client)  
  
 *// 循环读取 message 通道中的数据* **for** {  
 *// 通道 message 中有数据读到 msg 中。 没有，则阻塞* msg := <-message  
  
 *// 一旦执行到这里，说明message中有数据了，解除阻塞。 遍历 map* **for** \_, clnt := **range** onlineMap {  
 clnt.C <- msg *// 把从Message通道中读到的数据，写到 client 的 C 通道中。* }  
 }  
}  
  
**func** main() {  
 *// 创建监听 socket* listener, err := net.Listen(**"tcp"**, **"127.0.0.1: 8000"**)  
 **if** err != nil {  
 fmt.Println(**"Listen err:"**, err)  
 **return** }  
 **defer** listener.Close()  
  
 *// 创建go程 处理消息* **go** Manager()  
  
 *// 循环接收客户端连接请求* **for** {  
 conn, err := listener.Accept()  
 **if** err != nil {  
 fmt.Println(**"Accept err:"**, err)  
 **continue** *// 失败，监听其他客户端连接* }  
 **defer** conn.Close()  
  
 *// 给新连接的客户端，单独创建一个go程，处理客户端连接请求* **go** HandleConnect(conn)  
 }  
}

## 广播用户消息

当某个客户端向服务端发送消息后，服务端应将该消息广播给其它的客户端，达到聊天室的群聊效果。

开启一个新的go程，为方便传参，可以选择匿名go程。专门负责接收从客户端传递过来的数据，然后将接收到的数据写到messaage通道中。

在实现“广播用户上线”时，我已经完成：Manager go程会阻塞读message通道，一旦有数据，则遍历map中的在线用户。将数据写到结构体成员的C通道中。WriteMsgToClient go程会迭代C这个channel，最终将数据发送给客户端。

综上，实际上我们想完成“广播用户消息”给所有在线用户的功能，只需要将读到的数据写到message通道即可达到目的。

相关代码：

**func** HandleConnect(conn net.Conn) {

……

……  
 *// 广播新用户上线* message <- MakeMsg(clnt, **"login"**)  
  
 *// 创建一个新go程，循环读取用户发送的消息，广播给在线用户* **go func**() {buf := make([]byte, 2048)  *// 定义切片缓冲区，存储读到的用户消息*  
 **for** {  
 n, err := conn.Read(buf)  
 **if** n == 0 { *// 用户退出登录* fmt.Printf(**"用户%s退出登录\n"**, clnt.Name)  
 **return** }  
 **if** err != nil {  
 fmt.Println(**"Read err:"**, err)  
 **return** }  
 msg := string(buf[:n]) *// 保存用户写来的消息内容* message <-MakeMsg(clnt, msg) *// 将消息广播给所有在线用户* }  
 }()  
  
 **for** { *// 不能让当前go程结束。* ;  
 }  
}

## 展示在线用户

因为nc工具默认会添加‘\n’， 所以conn.Read()读取用户消息后，修改保存用户消息内容实现语句：

msg := string(buf[:n-1]) 重新读取用户消息。

读到后，对消息内容进行判断：如果用户发送了“who”，则当成一个查询指令处理。遍历map中所有在线用户，取出每个用户的相关描述信息，组成提示消息，写给当前用户即可。

由于这里客户端我们使用nc工具模拟，该工具对中文支持较差，所以我们组织的用户描述信息中不要包含中文字符。

代码片段如下：

msg := string(buf[:n-1]) *// 保存用户写来的消息内容, nc 工具默认添加‘\n’***if** msg == **"who"** && len(msg) == 3 { *// 判断用户发送了 who 指令* conn.Write([]byte(**"user list:\n"**))  
 **for** \_, user := **range** onlineMap { *// 遍历map获取在线用户* userInfo := user.Addr + **":"** + user.Name + **"\n"** *// 组织在线用户信息* conn.Write([]byte(userInfo)) *// 写给当前用户* }  
} **else** {  
 message <-MakeMsg(clnt, msg) *// 将消息广播给所有在线用户*}

## 修改用户名

前面我们查看用户信息时，用户名都是与用户网络地址相同的内容。主要由于用户登录时，创建该用户名不是用户自己完成的，无法洞悉用户的意图。当用户成功登录上来可以通过给服务器发送消息，来修改自己的用户名。

设定，如果用户发送“rename | Iron man”指令，既是想修改自己的用户名为“Iron man”。判断用户消息，是否包含“rename|”关键字：if len(msg) >= 8 && msg[:6] == "rename" {。如果是，那么拆分用户意欲修改的用户名保存。**strings.Split()**函数可以完成拆分字符串操作。

将该用户名替换当前用户的Name。使用用户的Addr作为key，找到map中当前用户，覆盖即可达到改名的目的。操作结束提示用户改名成功。

代码片段如下：

msg := string(buf[:n-1])  
**if** msg == **"who"** && len(msg) == 3 {  
 conn.Write([]byte(**"user list:\n"**))  
 **for** \_, user := **range** onlineMap {  
 userInfo := user.Addr + **":"** + user.Name + **"\n"** conn.Write([]byte(userInfo))  
 }  
 *// 判断用户输入的前6个字符是否为 rename*} **else if** len(msg) >= 8 && msg[:6] == **"rename"** { *// rename | Iron man* newName := strings.Split(msg, **"|"**)[1] *// 按"|"拆分，rename为[0], Iron man为[1]* clnt.Name = newName *// 替换掉当前用户原始Name* onlineMap[netAddr] = clnt *// 使用netAddr为key找到map中当前用户。覆盖* conn.Write([]byte(**"rename successful\n"**))  
} **else** {

message <- MakeMsg(clnt, msg)  
}

## 用户退出

前面在“广播用户消息”时，当conn.Read() 读到0时，我们在服务器端，简单打印了“用户xxx退出登录”的提示。

但实际上，在聊天室中，有在线用户离开，我们应该将这一事件广播给所有用户知晓，并且将该用户从map在线用列表中移除。需要实时的监看在线用户的状态。可以创建channel来检测用户退出状态，并使用select来监听channel上的数据流动。

当channel上有数据时，select对应阻塞case语句得以执行。将用户从map中移除。同时通知所有在线用户。

代码片段：

**func** HandleConnect(conn net.Conn) {

……

message <- MakeMsg(clnt, **"login"**)  
**isQuit := make(chan bool)** *// 检测用户主动退出*  
**go func**() {  
buf := make([]byte, 2048)  
 **for** {  
 n, err := conn.Read(buf)  
 **if** n == 0 {  
 **isQuit <- *true*** *// 用户主动退出登录* fmt.Printf(**"用户%s退出登录\n"**, clnt.Name)  
 **return** }  
 ……

}  
 }()  
  
 **for** { **select** {  
 **case** <-isQuit: *// 用户不主动退出，阻塞*

close(clnt.C)   
delete(onlineMap, netAddr) *// 将当前用户从map中移除* message <- MakeMsg(clnt, **"logout"**) *// 广播给在线用户，谁退出了* **return** *// 结束当前退出用户对应go程* }  
 }  
}

## 超时处理

如果客户端没有主动退出，并且长时间没有发送消息，会一直占用服务端的资源。服务器通常针对这种用户添加“超时强踢”机制，强制将该客户端与服务器连接断开。

可以借助并发编程时我们所学的select超时机制来实现。Select监听time.After(60 \* time.Second) 通道上的数据流动。如果在计时期间一直没有数据，通道中会被写入当前系统时间，select 的case满足读条件，不再阻塞。但，有一个问题，用户如果持续在输入数据，这个计时器依然在计时，时间到，依然会强制踢出用户。

因此，我们另外创建一个通道hasData来检测用户是否有数据发送，让Select也来监听这个channel。这样，当用户有数据输入时，select监听的这个hasData通道会满足case条件得以执行，但我们不做任何处理。目的是使得监听在select中的计时器被重新计时。

只有当真正持续60s没有数据发送时，select 中用于计时的case才满足条件，将用户与服务器连接断开。

代码片段：

**func** HandleConnect(conn net.Conn) {  
 ……

……  
isQuit := make(**chan** bool)  
 **hasData := make(chan bool)** *// 检测用户是否有消息发送*  
   
**go func**() {  
buf := make([]byte, 2048)  
 **for** {  
 n, err := conn.Read(buf)  
 ……  
 msg := string(buf[:n-1]) if {

……

} else if {

……

} else {

……

}  
 hasData <- ***true*** *// 只要执行到这里，就说明用户有数据发送* }  
 }()  
  
 **for** { **select** {  
 **case** <-isQuit:

close(clnt.C)delete(onlineMap, netAddr) message <- MakeMsg(clnt, **"logout"**) **return** **case** <-hasData:  
 *// 什么都不做，目的是让计时器归零* **case** <-time.After(60\*time.***Second***):

close(clnt.C)  
 delete(onlineMap, netAddr) *// 将当前用户从map中移除* message <- MakeMsg(clnt, **"time out leave"**) *// 广播给在线用户，超时退出* **return** *// 结束当前退出用户对应go程* }  
 }  
}

这里需要注意的是，每循环一次，第三个case后面的时间都会重新计算。（例如:执行完case<-hasData后，紧跟着执行第三个case，发现时间是10秒，不到60秒，条件不成立，不会执行该case后面的代码，进入下次循环，这时时间重新计算）

当hasData没有数据，isQuit没有数据，60s时间没有到，这时三个case都阻塞等待。直到60秒后，前两个case条件依然不成立，第三个case满足，执行后面代码，断开客户端连接，踢下线。

## 总结：

# 序列化和反射

## Json

### Json概述

JSON ：JavaScript 对象表示法（ JavaScript Object Notation ）。

是一种轻量级的数据交换格式。它基于 ECMAScript 的一个子集。 JSON 采用完全独立于语言的文本格式，但是也使用了类似于 C 语言家族的习惯（包括 C 、 C++ 、 C# 、 Java 、 JavaScript 、 Perl 、 Python 等）。这些特性使 JSON 成为理想的数据交换语言。 易于人阅读和编写，同时也易于机器解析和生成 ( 一般用于提升网络传输速率 ) 。目前，json 已经成为主流的数据格式。

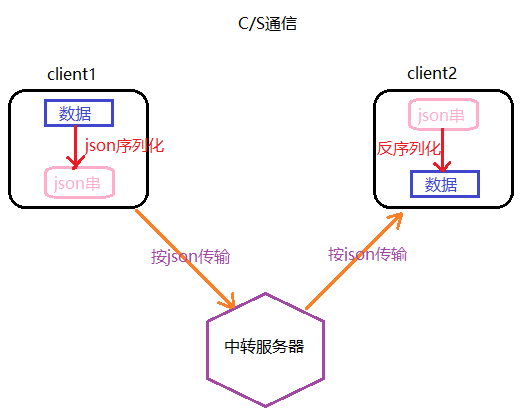
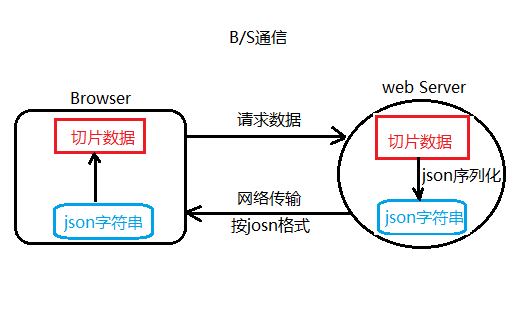
JSON 解析器和 JSON 库支持许多不同的编程语言。 JSON 文本格式在语法上与创建 JavaScript 对象的代码相同。由于这种相似性，无需解析器， JavaScript 程序能够使用内建的 eval() 函数，用 JSON 数据来生成原生的 JavaScript 对象。JSON 是存储和交换文本信息的语法。比 XML 更小、更快，更易解析。

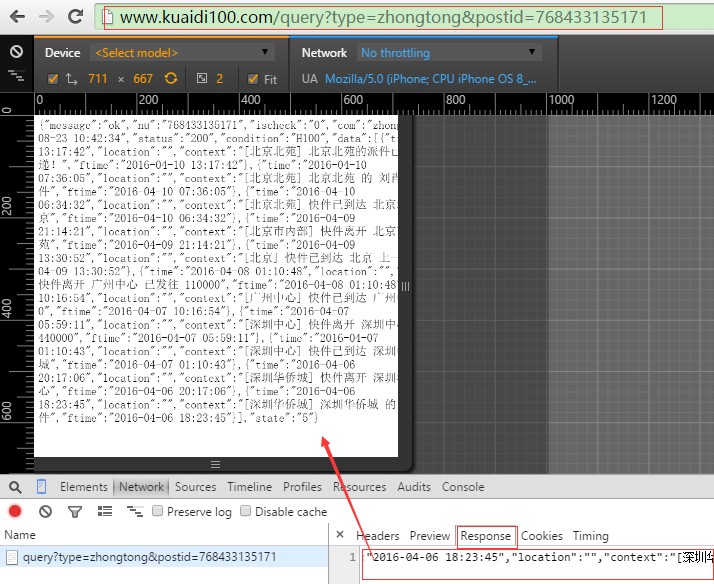
JSON 具有自我描述性，语法简洁，易于理解。

JSON数据主要有两种数据结构，一种是键/值，另一种是数组的形式来表示。

可登陆 json 中文官网，获取json相关资料： <http://www.json.org.cn/>

### Json的用途





### Json语法

首先，**任何数据类型都可以通过json来表示**。如：字符串、数字、对象、map、结构体、切片……

先来看一个简单的 JSON

{

"stars": [

{ "name":"Faye" ,"address":" 北京 " },

{ "name":"andy" ,"address":" 香港 " },

{ "name":"eddie","address":" 台湾 " },

]

}

JSON语法是JavaScript对象表示法语法的子集。

数据在键 / 值对中。数据由逗号分隔。花括号保存对象，也称一个文档对象。

方括号保存数组，每个数组成员用逗号隔开，并且每个数组成员可以是文档对象或者数组或者键值对。

JSON基于两种结构：“名称/值”对的集合（A collection of name/value pairs）。

不同的编程语言中，它被理解为对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hashtable），有键列表（keyed list），或者关联数组 （associative array）。值的有序列表（An ordered list of values）。在大部分语言中，它被实现为数组（array），矢量（vector），列表（list），序列（sequence）。

上述描述可归纳为JSON三种语法：

1. 键/值对。 key **:** value，冒号分割。

如： "name":"Faye"

key 必须是字符串类型，键/值对儿的“值”可以是：

数字（整数或浮点数）

字符串（在双引号中）

逻辑值（ true 或 false ）

null( 空值 未设置 )

对象（在花括号中）

数组（在方括号中）

key 是字符串类型，标准写法需加双引号，以免有可能报错。

1. Json对象。写在 { } 中，可包含多个key、value对。用“，”分隔。

如： { "name"**:**"Faye" ,"address"**:**"北京" }

1. Json数组。在 [ ] 中书写, 数组成员可以是对象、值，也可以是数组。

如： { "stars":[ {"name":"Faye" ,"address":"北京" },

{ "name":"andy" ,"address":"香港" },

{ "name":"eddie","address":"台湾"}, ]

}

### json格式化

可以使用在线格式化工具，验证json数据的正确性。

1. [www.json.cn](http://www.json.cn)
2. <http://www.json.org.cn/tools/index.htm>

{

"name":"矮大紧",

"sex":"man",

"age":131,

"grils":["金莲","凤姐","码蓉","春哥"],

"成绩": [2, 14, 9, 78, 96],

"家具":{"彩电":"海尔", "洗衣机":"三星"}

}

## 序列化

在前面讲解json用途时，我们有介绍到，数据在网络传输前后要进行序列化和反序列化。目的是将复杂的数据类型按照统一、简单且高效的形式转储，以达到网络传输的目的。

在go语言中，较为复杂的数据存储形式无外乎结构体、map、切片。因此这里我们以这三种数据类型序列化为例进行讲解。其他简单数据类型请同学们融会贯通，自主学习。

所谓“序列化”(Serialize)，简单理解就是将 结构体、map、切片这三种类型的数据相应的转换为josn字符串的过程。

### 结构体序列化

首先，创建结构体类型。由于之后要对该结构体进行序列化，而序列化函数是在 encoding/josn包中，因此内部字段应指定为首字母大写，以便导出作用域。

**type** Student **struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}

序列化函数：

import "encoding/json"

**func Marshal(v interface{}) ([][byte](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "byte), [error](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "error))**

参数：空接口类型。意味着任何数据类型（int、float、结构体…）都可以使用该函数进行序列化。

返回值：字符切片，err

示例代码：

**import** (  
 **"encoding/json"  
 "fmt"**)  
  
*// 定义结构体***type** Student **struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}  
*// 封装结构体序列化测试函数***func** structSerial() {  
 *// 定义结构体变量* stu := Student{  
 Name:**"瞎么海"**,  
 Id:110,  
 Age:33,  
 Addr:**"天津市海河"**,  
 }  
 *// 使用 encoding/json 包中的 Marshal() 函数将结构体变量进行序列化* data, err := json.Marshal(&stu)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"stu序列化后 = "**, string(data))  
}  
  
**func** main() {

// 测试结构体序列化

structSerial()  
}

输出结果为：

stu序列化后 = {"Name":"瞎么海","Id":110,"Age":33,"Addr":"天津市海河"}

### Map序列化

Map序列化的大致流程与结构体一致。首先也要先创建map变量。如：

var m map[string]interface{}

key值为string，value 可以是任意数据。但此m不能直接存储数据！必须使用make开辟空间再使用。

m = make(map[string]interface{})  
// 赋值  
m["name"] = "大晕头"  
m["sal"] = 3141.59  
m["age"] = 27  
m["addr"] = "北京朝阳"

同样，使用序列化函数json.Marshal() 可以将m完成序列化。由于m是引用可以直接传递。

示例代码：

*// 封装 map 序列化测试函数***func** mapSerial() {  
 *// 定义map变量* **var** m **map**[string]**interface**{}  
 *// 初始化map,获取空间* m = make(**map**[string]**interface**{})  
 *// 赋值* m[**"name"**] = **"大晕头"** m[**"sal"**] = 3141.59  
 m[**"age"**] = 27  
 m[**"addr"**] = **"北京朝阳"** *// 将 map 使用 Marshal() 函数进行序列化* data, err := json.Marshal(m) *// map本身为引用。* **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"map序列化后 = "**, string(data))  
}  
  
**func** main() {  
 *// 测试map序列化* mapSerial()  
}

输入结果为：

map序列化后 = {"addr":"北京朝阳","age":27,"name":"大晕头","sal":3141.59}

由于map是无序存储，因此查询序列化后的结果，可以与定义的顺序不一致。

### Slice序列化

首先，创建上面map类型的切片mySlice。

**var** mySlice []**map**[string]**interface**{}

依次创建map，并初始化、赋值，追加到切片中

*// 创建map1，初始化，赋值*m1 := make(**map**[string]**interface**{})  
m1[**"name"**] = **"李白"**m1[**"age"**] = 20  
m1[**"addr"**] = **"莫高窟"**mySlice = append(mySlice, m1)  
  
*// 创建map2，初始化，赋值*m2 := make(**map**[string]**interface**{})  
m2[**"name"**] = **"杜子腾"**m2[**"age"**] = 29  
m2[**"addr"**] = **[3]string{"毛纺厂","方便村","洗手街",}**mySlice = append(mySlice, m2)

由于map是**map**[string]**interface**{}类型，因此value可以是任意数据类型，数组也可以。

同样，使用相同的方法进行序列化并查看。

示例代码：

*// 封装 slice 序列化测试函数***func** sliceSerial() {  
 *// 创建一个 map[string]interface{}类型的 切片*

……  
 *// 创建map，初始化，赋值*

……  
 *// 将 slice 使用 Marshal() 函数进行序列化* data, err := json.Marshal(mySlice)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"slice序列化后 = "**, string(data))  
}  
**func** main() {  
 *// 测试slice序列化* sliceSerial()  
}

输入结果为：slice序列化后 =

[{"addr":"莫高窟","age":20,"name":"李白"},{"addr":["毛纺厂","方便村","洗手街"],"age":29,"name":"杜子腾"}]

可以将这个结果，导入到json在线格式化验证网站中查验。数据正确！

### 基础类型序列化

前面讲json的时候有提到，**任何数据类型都可以通过json来表示**。可以递推理解为：**任何数据类型都可以进行序列化**。

因此，普通的int数组、切片、普通的 string 变量等基础数据类型也都可以完成序列化。

示例代码：

*// int 类型切片序列化测试函数***func** sliceSerial2() {  
 **var** s = []int{1,2,3,4,5,6}  
 *// 将 slice 使用 Marshal() 函数进行序列化* data, err := json.Marshal(s)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"普通slice序列化后 = "**, string(data))  
}  
  
*// string 类型变量序列化函数***func** varSerial() {  
 **var** str string = **"hoho haha"** *// 将 slice 使用 Marshal() 函数进行序列化* data, err := json.Marshal(str)  
 **if** err != nil {  
 fmt.Println(**"Marshal err:"**, err)  
 **return** }  
 *// 查看序列化后的 json 字符串* fmt.Println(**"var序列化后 = "**, string(data))  
}  
  
**func** main() {  
 *// 测试int类型切片序列化* sliceSerial2()  
 *// 测试普通变量序列化* varSerial()  
}

可以看到，基础数据类型由于数据本身没有key-value这种存储形式，因此序列化后，得到的仅仅是字符串类型数据，实际使用时无太大意义，因此很少使用。

## 结构体标签

结构体的字段除了名字和类型外，还可以有一个可选的标签（tag）：它是一个附属于字段的字符串，可以是文档或其他的重要标记。比如在我们解析json或生成json文件时，常用到encoding/json包，它提供一些默认标签，例如：omitempty标签可以在序列化的时候忽略0值或者空值。而“-”标签的作用是不进行序列化，其效果和直接将结构体中的字段写成小写的效果一样。

type Info struct {

Name string

Age int `json:"age,omitempty"`

Sex string

}

在序列化和反序列化的时候，也支持类型转化等操作。如

type Info struct {

Name string

Age int `json:"age,string"`

//这样生成的json对象中，age就为字符串

Sex string

}

现在来了解下如何设置自定义的标签，以及如何像官方包一样，可以通过标签，对字段进行自定义处理。要实现这些，我们要用到reflect包。

package main

import (

"fmt"

"reflect"

)

const tagName = "Testing"

type Info struct {

Name string `Testing:"-"`

Age int `Testing:"age,min=17,max=60"`

Sex string `Testing:"sex,required"`

}

func main() {

info := Info{

Name: "benben",

Age: 23,

Sex: "male",

}

//通过反射，我们获取变量的动态类型

t := reflect.TypeOf(info)

fmt.Println("Type:", t.Name())

fmt.Println("Kind:", t.Kind())

for i := 0; i < t.NumField(); i++ {

field := t.Field(i) //获取结构体的每一个字段

tag := field.Tag.Get(tagName)

fmt.Printf("%d. %v (%v), tag: '%v'\n", i+1, field.Name, field.Type.Name(), tag)

}

}

假设有一个Person结构体定义如下

type Person struct {

Name string `label:"Person Name: " uppercase:"true"`

Age int `label:"Age is: "`

Sex string `label:"Sex is: "`

Description string

}

有四个字段，字段后面的使用` `引用的部分就是tag，我们希望使用一个名为lable的tag来定义打印时候的标题，默认使用字段名称加冒号作为label。如果是字符串类型的字段，通过名称为uppercase的tag控制是否显示字符串的大写形式，默认按照小写。例如有一个Person结构体变量为

{Tom 29 Male Cool}

按照上面Person中tag的使用，打印应该为

Person Name: TOM

Age is: 29

Sex is: male

Description: cool

Golang解析标签主要通过反射实现，下面就看看我们如何实现上面的功能：

package main

import (

"fmt"

"reflect"

"strings"

)

type Person struct {

Name string `label:"Person Name: " uppercase:"true"`

Age int `label:"Age is: "`

Sex string `label:"Sex is: "`

Description string

}

// 按照tag打印结构体

func PrintUseTag(ptr interface{}) error {

// 获取入参的类型

t := reflect.TypeOf(ptr)

// 入参类型校验

if t.Kind() != reflect.Ptr || t.Elem().Kind() != reflect.Struct {

return fmt.Errorf("参数应该为结构体指针")

}

// 取指针指向的结构体变量

v := reflect.ValueOf(ptr).Elem()

// 解析字段

for i := 0; i < v.NumField(); i++ {

// 取tag

fieldInfo := v.Type().Field(i)

tag := fieldInfo.Tag

// 解析label tag

label := tag.Get("label")

if label == "" {

label = fieldInfo.Name + ": "

}

// 解析uppercase tag

value := fmt.Sprintf("%v", v.Field(i))

if fieldInfo.Type.Kind() == reflect.String {

uppercase := tag.Get("uppercase")

if uppercase == "true" {

value = strings.ToUpper(value)

} else {

value = strings.ToLower(value)

}

}

fmt.Println(label + value)

}

return nil

}

func main() {

person := Person{

Name: "Tom",

Age: 29,

Sex: "Male",

Description: "Cool",

}

PrintUseTag(&person)

}

上面代码，main函数创建了一个名为person的Person结构体变量，然后调用PrintUseTag函数进行打印，所以主要的逻辑我们要看PrintUseTag函数。该函数通过反射，获取tag，然后按照不同的情况解析tag并执行不同的操作，只有字段类型为字符串时候，我们才去判断uppercase tag是否使用，之后打印每一个字段的label和value组成的字符串。value不一定都是字符串类型，所以我们借助于fmt包的Sprintf函数的%v，将其他类型转化为字符串表述。

掌握了tag的使用，我们可以使用Golang定义自己的很多工具tag，减少代码量，尤其在定义一些框架时作用更加明显。

### 序列化应用tag

通过前面序列化章节编写的测试代码，我们可以看到。生成的序列化json字符串是将struct 结构体的字段名作为json字符串的key，将字段值作为value的。

这在实际应用过程中容易出现问题。为了方便导出包作用域，我们在定义结构体时，要将字段名的首字母大写，而实际应用过程中，json的key值通常我们希望是小写字母开头的字串。

很自然的，可以想到，将struct 结构体中的字段名改为小写字母不就可以解决这个冲突吗？但实际测试发现，如果转为小写，则丢失了包作用域，Marshal() 函数不能正常完成序列化，得到的往往是空值。后续再来测试。

这里，就需要用到我们上节提到的结构体标签tag来解决这一问题了。从新定义结构体类型，注意语法：stu\_name一定要使用 " " 包裹起来。

**type** Student **struct** {  
 Name string `json: "stu\_name"`  
 Id int `json: "stu\_id"`  
 Age int // 剩余两个字段不添加tag，用以比较  
 Addr string  
}

序列化方法与之前的序列化方法一致。使用json.Marshal() 可以直接得到json字符串，输出结果为：

stu序列化后 = {"stu\_name":"瞎么海","stu\_id":110,"Age":33,"Addr":"天津市海河"}

可以看到，此时json字符串的key值为“stu\_name”和“stu\_id”，是我们定义结构体时指定的tag。而没有指定tag的Age和Addr，key值依然是原来字段的名字。

如果改为小写会怎样呢？测试：

**type** Student **struct** {  
 Name string `json: "stu\_name"`  
 Id int `json: "stu\_id"`  
 age int // 将两个不添加tag的字段名改为小写  
 addr string  
}

当然，结构体变量赋值时也要指定小写的字段名进行赋值。

输出结果为：

stu序列化后 = {"stu\_name":"瞎么海","stu\_id":110}

可以看到，程序没有报错。但指定为小写字段名的两个字段age、addr没有成功转换为json字串。

## 反序列化

既然序列化是将go语言的某种类型的数据转换为josn字符串，那么 “反序列化”(Deserialize)，就是将josn字符串转换回序列化之前数据的过程。简单理解就是序列化的逆操作。

这里按我们前面章节序列化后得到的json串依次进行反序列化。

### 结构体反序列化

首先，依然需要创建结构体类型。要求该类型必须与之前序列化时指定的结构体类型完全一致（个数、类型、顺序）。

**type Teacher struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}

定义结构体变量

**var** tea Teacher

准备待反序列化的json字符串，由于字串中包含 **" "**，使用反引号包裹该字串。当然，实际开发过程中，这个数据大多情况下是借助网络传输获取的。

str := **`{"Name":"瞎么海","Id":110,"Age":33,"Addr":"天津市海河"}`**

序列化函数：

import "encoding/json"

**func Unmarshal(data [][byte](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "byte), v interface{}) [error](https://studygolang.com/static/pkgdoc/pkg/builtin.htm" \l "error)**

参数1：待解析的 json编码字符串

参数2：解析后传出的结果

返回值：err

示例代码：

**import** (  
 **"encoding/json"  
 "fmt"**)  
*// 定义结构体***type** Teacher **struct** {  
 Name string  
 Id int  
 Age int  
 Addr string  
}  
*// 封装函数测试结构体反序列化***func** structDeserial() {  
 *// 准备待反序列化的json字符串* str := **`{"Name":"瞎么海","Id":110,"Age":33,"Addr":"天津市海河"}`** *// 定义Teacher类型变量* **var** tea Teacher  
 *// 使用 encoding/json 包中的 Unmarshal() 函数，反序列化* err := json.Unmarshal([]byte(str), &tea)  
 **if** err != nil {  
 fmt.Println(**"Unmarshal err:"**, err)  
 **return** }  
 *// 查看反序列化结果* fmt.Println(**"反序列化后：tea="**, tea, **"tea.Name="**, tea.Name)  
}  
**func** main() {  
 *// 测试结构体反序列化* structDeserial()  
}

输入结果为：

反序列化后：tea= {瞎么海 110 33 天津市海河} tea.Name= 瞎么海

### Map反序列化

准备待反序列化的json字符串。

str := **`{"addr":"北京朝阳","age":27,"name":"大晕头","sal":3141.59}`**

定义map变量，类型必须与之前序列化的类型完全一致。

**var** m **map**[string]**interface**{}

不需要使用make函数给m初始化，开辟空间。这是因为，在反序列化函数 Unmarshal() 中会判断传入的参数2，如果是map类型数据，会自动开辟空间。相当于是Unmarshal() 函数可以帮助我们做make操作。

但！传参时需要注意，**Unmarshal**的第二个参数，是用作传出，返回结果的。因此必须传m的地址值。

json.Unmarshal([]byte(str), &m)

示例代码：

*// 封装函数测试map反序列化***func** mapDeserial() {  
 *// 准备反序列化的json字符串* str := **`{"addr":"北京朝阳","age":27,"name":"大晕头","sal":3141.59}`** *// 定义map变量* **var** m **map**[string]**interface**{}  
 err := json.Unmarshal([]byte(str), &m)  
 **if** err != nil {  
 fmt.Println(**"Unmarshal err:"**, err)  
 **return** }  
 *// 查看反序列化结果* fmt.Println(**"反序列化后：m ="**, m)  
}  
**func** main() {  
 *// 测试map反序列化* mapDeserial()  
}

输出结果为：

反序列化后：m = map[addr:北京朝阳 age:27 name:大晕头 sal:3141.59]

### Slice反序列化

实现思路与前面两种的实现完全一致，这里不再赘述。直接参考代码：

*// 封装函数测试slice反序列化***func** sliceDeserial() {

*// 准备反序列化的json字符串* str := **`[{"addr":"莫高窟","age":20,"name":"李白"},{"addr":["毛纺厂","方便村","洗手街"],"age":29,"name":"杜子腾"}]`**

*// 定义slice变量* **var** slice []**map**[string]**interface**{}  
 err := json.Unmarshal([]byte(str), &slice)  
 **if** err != nil {  
 fmt.Println(**"Unmarshal err:"**, err)  
 **return** }

*// 查看反序列化结果* fmt.Println(**"反序列化后：slice ="**, slice)  
}

**func** main() {  
 *// 测试slice反序列化* sliceDeserial()  
}

输出结果为：

slice = [map[addr:莫高窟 age:20 name:李白] map[addr:[毛纺厂 方便村 洗手街] age:29 name:杜子腾]]

**特别强调：**

反序列化json字符串时，务必确保反序列化传出的数据类型，与之前序列化的数据类型完全一致。

## 回调

本质：函数指针。

语法：关键字type 函数指针名 关键字func(形参列表) 返回值类型

例如：type FUNCP func(x int, y bool）int

用户自定义一个函数，不直接在程序中显式调用，而当某一特定条件满足时，再进行调用，或由系统自动调用。实现本质：函数指针。

// 定义一个函数指针【类型】

type funcP func(x int, y bool) int

// 封装主调函数，函数指针作为参数。其他两个参数都是函数指针调用的函数要用的参数。

func useCallback(x int, y bool, p funcP) int {

return p(x, y)

}

// 回调函数1

func addOne(x int, y bool) int {

if y == true {

x++

}

return x

}

// 回调函数2

func subTen(x int, y bool) int {

if y == true {

x -= 10

}

return x

}

func main() {

var p funcP

fmt.Printf("type:%T, sizeof(P)=%d\n", p, unsafe.Sizeof(p))

p = addOne

fmt.Println(useCallback(5, true, p))

fmt.Println(useCallback(26, true, subTen))

}

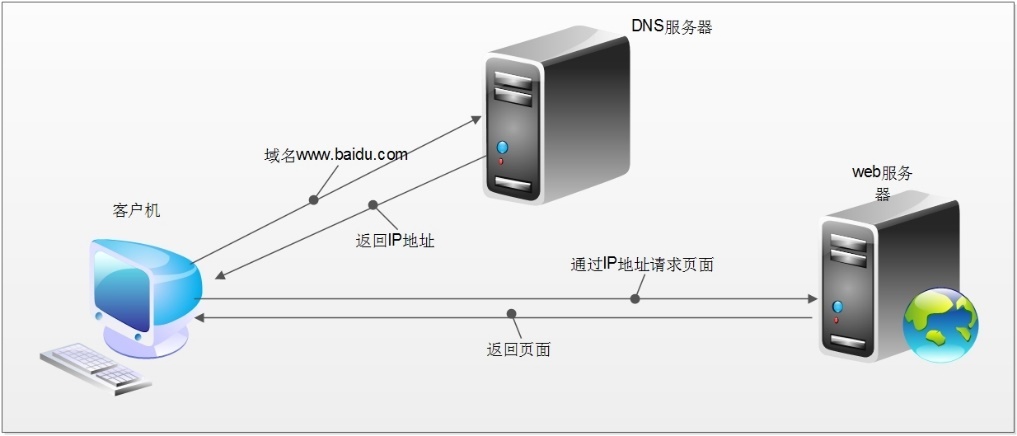
# HTTP编程

## 概述

### Web工作方式

我们平时浏览网页的时候,会打开浏览器，输入网址后按下回车键，然后就会显示出你想要浏览的内容。在这个看似简单的用户行为背后，到底隐藏了些什么呢？

对于普通的上网过程，系统其实是这样做的：浏览器本身是一个客户端，当你输入URL的时候，首先浏览器会去请求DNS服务器，通过DNS获取相应的域名对应的IP，然后通过IP地址找到IP对应的服务器后，要求建立TCP连接，等浏览器发送完HTTP Request（请求）包后，服务器接收到请求包之后才开始处理请求包，服务器调用自身服务，返回HTTP Response（响应）包；客户端收到来自服务器的响应后开始渲染这个Response包里的主体（body），等收到全部的内容随后断开与该服务器之间的TCP连接。



DNS域名服务器（Domain Name Server）是进行域名(domain name)和与之相对应的IP地址转换的服务器。DNS中保存了一张域名解析表，解析消息的域名。

一个**Web服务器也被称为HTTP服务器**，它通过HTTP (HyperText Transfer Protocol 超文本传输协议)协议与客户端通信。这个客户端通常指的是Web浏览器(其实手机端客户端内部也是浏览器实现的)。

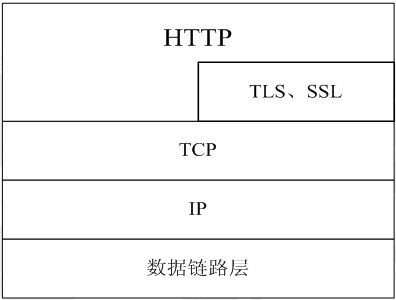
Web服务器的工作原理可以简单地归纳为：

* 客户机通过TCP/IP协议建立到服务器的TCP连接
* 客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档
* 服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端
* 客户机与服务器断开。由客户端解释HTML文档，在客户端屏幕上渲染图形结果

### HTTP协议

超文本传输协议(HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议，它详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

HTTP协议通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了我们常说的HTTPS。如下图所示：



### 地址（URL）

URL全称为Unique Resource Location，用来表示网络资源，可以理解为网络文件路径。

基本URL的结构包含模式（协议）、服务器名称（IP地址）、路径和文件名。常见的协议/模式如http、https、ftp等。服务器的名称或IP地址后面有时还跟一个冒号和一个[端口号](https://baike.baidu.com/item/%E7%AB%AF%E5%8F%A3%E5%8F%B7" \t "_blank)。再后面是到达这个文件的路径和文件本身的名称。如：

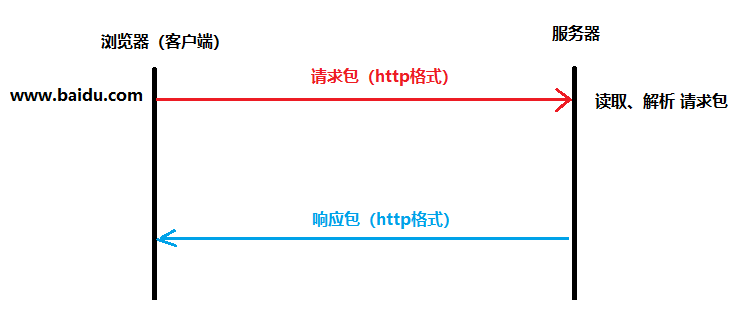
http://localhost[":"port][abs\_path]

<http://192.168.31.1/html/index>

https://pan.baidu.com/

URL的长度有限制，不同的服务器的限制值不太相同，但是不能无限长。

## HTTP协议



### 请求报文格式说明

HTTP 请求报文由请求行、请求头部、空行、请求包体4个部分组成，如下图所示：



1. **请求行**

请求行由方法字段、URL字段 和HTTP 协议版本字段 3个部分组成，他们之间使用空格隔开。常用的 HTTP 请求方法有 GET、POST。

GET：

* 当客户端要从服务器中读取某个资源时，使用GET 方法。GET 方法要求服务器将URL 定位的资源放在响应报文的数据部分，回送给客户端，即向服务器请求某个资源。
* 使用GET方法时，请求参数和对应的值附加在 URL 后面，利用一个问号(“?”)代表URL 的结尾与请求参数的开始，传递参数长度受限制，因此GET方法不适合用于上传数据。
* 通过GET方法来获取网页时，参数会显示在浏览器地址栏上，因此保密性很差。

POST：

* 当客户端给服务器提供信息较多时可以使用POST 方法，POST 方法向服务器提交数据，比如完成表单数据的提交，将数据提交给服务器处理。
* GET 一般用于获取/查询资源信息，POST 会附带用户数据，一般用于更新资源信息。POST 方法将请求参数封装在HTTP 请求数据中，而且长度没有限制，因为POST携带的数据，在HTTP的请求正文中，以名称/值的形式出现，可以传输大量数据。

1. **请求头部**

请求头部为请求报文添加了一些附加信息，由“名/值”对组成，每行一对，名和值之间使用冒号分隔。请求头部通知服务器有关于客户端请求的信息，典型的请求头有：

| **请求头** | **含义** |
| --- | --- |
| User-Agent | 请求的浏览器类型 |
| Accept | 客户端可识别的响应内容类型列表，星号“ \* ”用于按范围将类型分组，用“ \*/\* ”指示可接受全部类型，用“ type/\* ”指示可接受 type 类型的所有子类型 |
| Accept-Language | 客户端可接受的自然语言 |
| Accept-Encoding | 客户端可接受的编码压缩格式 |
| Accept-Charset | 可接受的应答的字符集 |
| Host | 请求的主机名，允许多个域名同处一个IP 地址，即虚拟主机 |
| connection | 连接方式(close或keepalive) |
| Cookie | 存储于客户端扩展字段，向同一域名的服务端发送属于该域的cookie |

1. **空行**

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

1. **请求包体**

请求包体不在GET方法中使用，而在POST方法中使用。POST方法适用于需要客户填写表单的场合。与请求包体相关的最常使用的是包体类型Content-Type和包体长度Content-Length。

#### 获取请求报文

为了更直观的看到浏览器发送的请求包，我们借助前面学习的TCP通信模型，编写一个简单的web服务器，只接收浏览器发送的内容，打印查看。

服务器测试代码：

**package** main  
  
**import** (  
 **"net"  
 "fmt"**)  
  
**func** main() {  
 *//创建、监听socket* listenner, err := net.Listen(**"tcp"**, **"127.0.0.1:8000"**)  
 **if** err != nil {  
 fmt.Println(**"Listen err:"**, err)  
 **return** }  
 **defer** listenner.Close()  
  
 *//阻塞等待客户端连接* conn, err := listenner.Accept()  
 **if** err != nil {  
 fmt.Println(**"Accept err:"**, err)  
 **return** }  
 **defer** conn.Close()  
  
 fmt.Println(conn.RemoteAddr().String(), **"连接成功"**) *//连接客户端的网络地址* buf := make([]byte, 4096) *//切片缓冲区，接收客户端发送数据* n, err := conn.Read(buf) *//n 接收数据的长度* **if** err != nil {  
 fmt.Println(**"Read err:"**, err)  
 **return** }  
 result := buf[:n] *//切片截取* fmt.Printf(**"#\n%s#"**, string(result))  
}

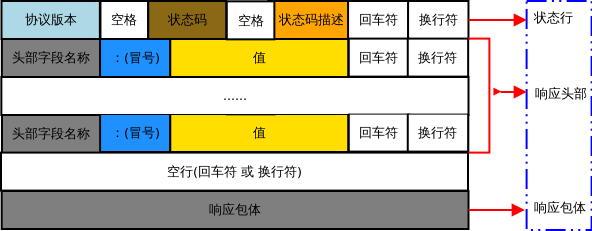
在浏览器中输入url地址： 127.0.0.1:8000

服务器端运行打印结果如下：



### 响应报文格式说明

HTTP 响应报文由状态行、响应头部、空行、响应包体4个部分组成，如下图所示：



1. 状态行

状态行由 HTTP 协议版本字段、状态码和状态码的描述文本3个部分组成，他们之间使用空格隔开。

状态码：状态码由三位数字组成，第一位数字表示响应的类型，常用的状态码有五大类如下所示：

| **状态码** | **含义** |
| --- | --- |
| 1xx | 表示服务器已接收了客户端请求，客户端可继续发送请求 |
| 2xx | 表示服务器已成功接收到请求并进行处理 |
| 3xx | 表示服务器要求客户端重定向 |
| 4xx | 表示客户端的请求有非法内容 |
| 5xx | 表示服务器未能正常处理客户端的请求而出现意外错误 |

常见的状态码举例：

| **状态码** | **含义** |
| --- | --- |
| 200 OK | 客户端请求成功 |
| 400 Bad Request | 请求报文有语法错误 |
| 401 Unauthorized | 未授权 |
| 403 Forbidden | 服务器拒绝服务 |
| 404 Not Found | 请求的资源不存在 |
| 500 Internal Server Error | 服务器内部错误 |
| 503 Server Unavailable | 服务器临时不能处理客户端请求(稍后可能可以) |

1. 响应头部

响应头可能包括：

| **响应头** | **含义** |
| --- | --- |
| Location | Location响应报头域用于重定向接受者到一个新的位置 |
| Server | Server 响应报头域包含了服务器用来处理请求的软件信息及其版本 |
| Vary | 指示不可缓存的请求头列表 |
| Connection | 连接方式 |

1. 空行

最后一个响应头部之后是一个空行，发送回车符和换行符，通知服务器以下不再有响应头部。

1. 响应包体

服务器返回给客户端的文本信息。

## Go语言HTTP编程

Go语言标准库内建提供了net/http包，涵盖了HTTP客户端和服务端的具体实现。使用net/http包，我们可以很方便地编写HTTP客户端或服务端的程序。

### HTTP服务端

示例代码：

**package** main

**import** (

    "fmt"

    "net/http"

)

//服务端编写的业务逻辑处理程序 —— 回调函数

**func** myHandler(w http.ResponseWriter, r \*http.Request) {

    fmt.Println("method = ", r.Method) //请求方法

    fmt.Println("URL = ", r.URL) // 浏览器发送请求文件路径

    fmt.Println("header = ", r.Header) // 请求头

    fmt.Println("body = ", r.Body) // 请求包体

    fmt.Println(r.RemoteAddr, "连接成功") //客户端网络地址

    w.Write([]byte("hello http")) //给客户端回复数据

}

**func** main() {

    http.HandleFunc("/hello", myHandler) // 注册处理函数

    //该方法用于在指定的 TCP 网络地址 addr 进行监听，然后调用服务端处理程序来处理传入的连接请求。

    //该方法有两个参数：第一个参数 addr 即监听地址；第二个参数表示服务端处理程序，通常为nil

    //当参2为nil时，服务端调用 http.DefaultServeMux 进行处理

    http.ListenAndServe("127.0.0.1:8000", nil)

}

浏览器输入url地址：127.0.0.1:8000/hello

回调函数myHandler的函数原型固定。**func** myHandler(w http.ResponseWriter, r \*http.Request)

参数：w http.ResponseWriter用来“给客户端回发数据”。它是一个interface：

**type** ResponseWriter **interface** {Header() Header **Write([]byte) (int, error)** WriteHeader(int)   
}

参数：r \*http.Request 用来“接收客户端发送的数据”。浏览器发送给服务器的http请求包的内容可以借助r来查看。它对应一个结构体：

**type** Request **struct** {

**Method string** // 浏览器请求方法 GET、POST…

**URL \*url.URL** // 浏览器请求的访问路径

……

**Header Header** // 请求头部

**Body io.ReadCloser** // 请求包体

**RemoteAddr string** // 浏览器地址

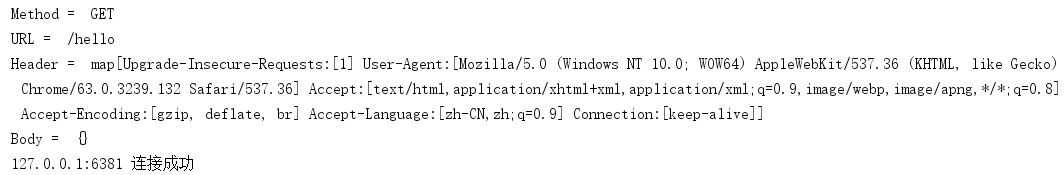
……

ctx context.Context  
}

查看一下结构体成员：

fmt.Println(**"Method = "**, r.Method)  
fmt.Println(**"URL = "**, r.URL)  
fmt.Println(**"Header = "**, r.Header)  
fmt.Println(**"Body = "**, r.Body)  
fmt.Println(r.RemoteAddr, **"连接成功"**)

查看到如下内容：



**练习：**

在计算机中选定一个目录，存放jpg、png、txt、mp3、gif、m4a等类型文件。编写一个服务器程序，可以给浏览器提供该目录下文件的访问服务。

如：目录中存有图片文件：lf.jpg。用户在浏览器中输入：127.0.0.1:8000**/**lf.jpg 可以查看该图片。

### HTTP客户端

客户端访问web服务器数据，主要使用**func Get**(url string) (resp \*Response, err error)函数来完成。读到的响应报文数据被保存在 Response 结构体中。

**type** Response **struct** {  
 Status string *// e.g. "200 OK"* StatusCode int *// e.g. 200* Proto string *// e.g. "HTTP/1.0"* ……Header HeaderBody io.ReadCloser ……  
}

服务器发送的响应包体被保存在Body中。可以使用它提供的Read方法来获取数据内容。保存至切片缓冲区中，拼接成一个完整的字符串来查看。

结束的时候，需要调用Body中的Close()方法关闭io。

示例代码：

**package** main  
  
**import** (  
 **"net/http"  
 "fmt"  
 "io"**)  
  
**func** main() {  
 *// 使用Get方法获取服务器响应包数据  
 //resp, err := http.Get("http://www.baidu.com")* resp, err := http.Get(**"http://127.0.0.1:8000/hello"**)  
 **if** err != nil {  
 fmt.Println(**"Get err:"**, err)  
 **return** }  
 **defer** resp.Body.Close()  
  
 *// 获取服务器端读到的数据* fmt.Println(**"Status = "**, resp.Status) *// 状态* fmt.Println(**"StatusCode = "**, resp.StatusCode) *// 状态码* fmt.Println(**"Header = "**, resp.Header) *// 响应头部* fmt.Println(**"Body = "**, resp.Body) *// 响应包体* buf := make([]byte, 4096) *// 定义切片缓冲区，存读到的内容* **var** result string  
 *// 获取服务器发送的数据包内容* **for** {  
 n, err := resp.Body.Read(buf) *// 读body中的内容。* **if** n == 0 {  
 fmt.Println(**"--Read finish!"**)  
 **break** }  
 **if** err != nil && err != io.EOF {  
 fmt.Println(**"resp.Body.Read err:"**, err)  
 **return** }  
 result += string(buf[:n]) *// 累加读到的数据内容* }  
 *// 打印从body中读到的所有内容* fmt.Println(**"result = "**, result)  
}

### 爬取豆瓣电影

#### 双向爬取

首先我们来熟悉两个基础技术名词：横向爬取 和 纵向爬取。

**横向爬取：**

所谓横向爬取，是指在爬取的网站页面中，以“页”为单位，找寻该网站分页器规律。一页一页的爬取网站数据信息。大多数网站，采用分页管理模式。针对这类网站，首先要确立横向爬取方法。

**纵向爬取：**

纵向爬取，是指在一个页面内，按不同的“条目”为单位。找寻各条目之间的规律。一条一条的爬取一个网页中的数据信息。也就是同时爬取一个页面内不同类别数据。

#### 爬取电影评分

类似的，我们再来试试豆瓣电影爬取。首先打“豆瓣”网站首页，然后单击菜单中“排行榜”。在页面右部偏下的位置有一个“豆瓣电影TOP250”，这里记录了全球影迷评出的世界级经典影片。当你剧荒的时候，参考下这个排行榜，选取任意一部来观赏下，都不会有浪费生命的感觉。

点击“全部”进入前250部电影排行榜页面。URL为：<https://movie.douban.com/top250> 同样，测试下翻页特性：

第二页：<https://movie.douban.com/top250?start=25&filter>=

第三页：<https://movie.douban.com/top250?start=50&filter>=

第四页：<https://movie.douban.com/top250?start=75&filter>=

得出横向爬取条件大致为：“下一页”= “前一页”+ 25。

寻找好的电影，主要参考依据为“分数”和“评分人数”两项。因此，这里我们将电影名、评分分数和该电影的参与评分的评分人数作为我们的爬取目标。

选择任意一个页面，探索纵向爬取条件。总结发现：

电影名：被包裹在 <img width="100" alt="电影名称" 之中

分数：被包裹在 <span class="rating\_num" property="v:average">分数</span> 之中

评分人数：被包裹在 <span>评分人数 人评价</span> 之中

同样，可以使用正则表达式 **(?s:(.\*?))** 来依次获取 电影名、分数、评分人数三部分数据信息。而后封装文件，将这三部分信息按格式存入即可。

借助前面的经验，我们直接实现一个并发版的爬虫，爬取豆瓣电影评分信息。

示例代码：

**package** main  
**import** (  
 **"fmt"  
 "net/http"  
 "os"  
 "regexp"  
 "strconv"  
 "time"**)  
  
*//发起请求，获取网页内容***func** HttpGet(url string) (result string, err error) {  
 resp, err1 := http.Get(url) *//发送get请求* **if** err1 != nil {  
 err = err1  
 **return** }  
 **defer** resp.Body.Close()  
  
 *//读取网页内容* buf := make([]byte, 4\*1024)  
 **for** {  
 n, \_ := resp.Body.Read(buf)  
 **if** n == 0 {  
 **break** }  
 result += string(buf[:n]) *//累加读取的内容* }  
 **return**}  
  
**func** SpiderPage(i int, page **chan** int) {  
 *// 明确爬取的url* fmt.Println(**"正在抓取第"** + strconv.Itoa(1) + **"页......"**)  
 url := **"https://movie.douban.com/top250?start="** + strconv.Itoa((i-1)\*25) + **"&filter="** time.Sleep(1 \* time.***Second***) *// 为防止访问IP 被网站屏闭，降低爬取频率  
  
 // 开始爬取页面内容，将结果保存至 result。* result, err := HttpGet(url)  
 **if** err != nil {  
 fmt.Println(**"HttpGet err = "**, err)  
 **return** }  
  
 *// 解析、编译正则表达式。—— 评价人数* ret1 := regexp.MustCompile(**`<span>(?s:(\d\*?))人评价</span>`**)  
 *// 取关键信息 评价人数* pepoleCount := ret1.FindAllStringSubmatch(result, -1)  
  
 *// 解析、编译正则表达式。—— 评分* patternScore := **`<span class="rating\_num" property="v:average">(?s:(\d\*?))</span>`** ret2 := regexp.MustCompile(patternScore)  
 filmScore := ret2.FindAllStringSubmatch(result, -1)  
  
 *// 解析、编译正则表达式。—— 电影名称* patternName := **`<img width="100" alt="(?s:(\d\*?))"`** ret3 := regexp.MustCompile(patternName)  
 filmName := ret3.FindAllStringSubmatch(result, -1)  
  
 *// 把内容写入到文件* Save2File(i, pepoleCount, filmScore, filmName)  
  
 page <- i *//写完一个文件，写i 到channel*}  
  
*//把内容写入到文件***func** Save2File(i int, pepoleCount, filmScore, filmName [][]string) {  
 *// 新建文件，每一页保存成一个文件* f, err := os.Create(**"第 "** + strconv.Itoa(i) + **" 页.txt"**)  
 **if** err != nil {  
 fmt.Println(**"os.Create err = "**, err)  
 **return** }  
 **defer** f.Close()  
  
 *// 写标题* f.WriteString(**"电影名称"** + **"\t\t\t"** + **"评分"** + **"\t\t"** + **"评价人数"** + **"\t"** + **"\r\n"**)  
  
 *// 写内容* n := len(pepoleCount)  
 **for** i := 0; i < n; i++ {  
 f.WriteString(filmName[i][1] + **"\t\t\t"** + filmScore[i][1] + **"\t\t"** + pepoleCount[i][1] + **"\t"** + **"\r\n"**)  
 }  
}  
  
**func** DoWork(start, end int) {  
 fmt.Printf(**"准备爬取第 %d 页到 %d 页的网址\n"**, start, end)  
 page := make(**chan** int)  
 **for** i := start; i <= end; i++ {  
 *//定义一个函数，爬主页面* **go** SpiderPage(i, page)  
 }  
  
 **for** i := start; i <= end; i++ {  
 fmt.Printf(**"第 %d 个页面爬取完成\n"**, <-page)  
 }  
}  
  
**func** main() {  
 **var** start, end int  
 fmt.Printf(**"请输入起始页( >= 1 ):"**)  
 fmt.Scan(&start)  
 fmt.Printf(**"请输入终止页( >= 起始页 ):"**)  
 fmt.Scan(&end)  
  
 DoWork(start, end)   
}