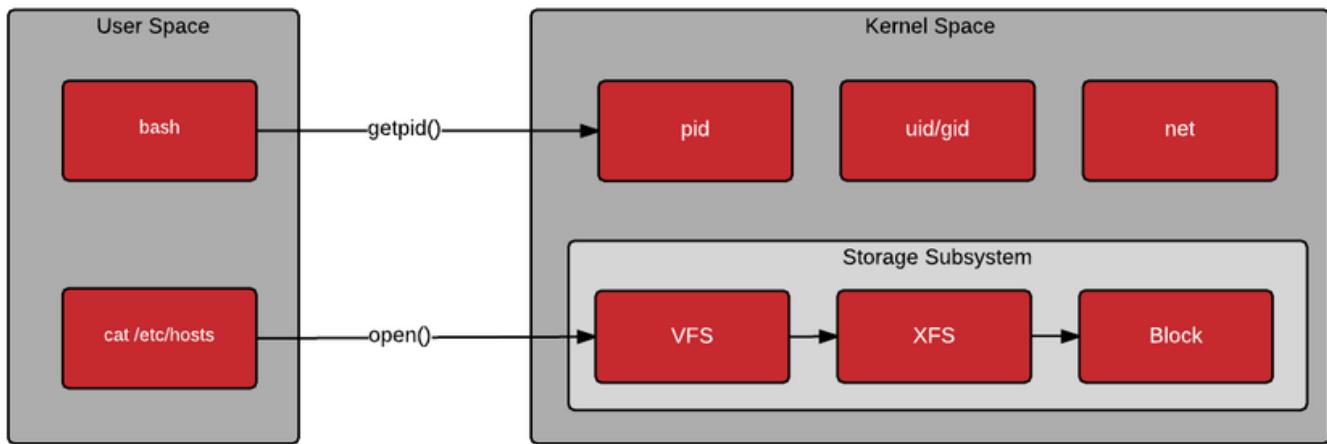


User space 与 Kernel space

学习 Linux 时，经常可以看到两个词：User space（用户空间）和 Kernel space（内核空间）。

简单说，Kernel space 是 Linux 内核的运行空间，User space 是用户程序的运行空间。为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。



Kernel space 可以执行任意命令，调用系统的一切资源；User space 只能执行简单的运算，不能直接调用系统资源，必须通过系统接口（又称 system call），才能向内核发出指令。

```
str = "my string" // 用户空间
x = x + 2
file.write(str) // 切换到内核空间

y = x + 4 // 切换回用户空间
```

上面代码中，第一行和第二行都是简单的赋值运算，在 User space 执行。第三行需要写入文件，就要切换到 Kernel space，因为用户不能直接写文件，必须通过内核安排。第四行又是赋值运算，就切换回 User space。

查看 CPU 时间在 User space 与 Kernel Space 之间的分配情况，可以使用 top 命令。它的第三行输出就是 CPU 时间分配统计。

```
top - 10:19:40 up 3 min,  3 users,  load average: 2.07, 2.14, 0.96
Tasks: 183 total,   1 running, 182 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.2 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem: 1545116 total, 930084 used, 615032 free, 80684 buffers
KiB Swap: 2095100 total,      0 used, 2095100 free. 360264 cached Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
1260 mysql    20   0 327548 36292 5792 S  0.7  2.3  0:04.74 mysqld
  1 root      20   0    4584  2652 1444 S  0.0  0.2  0:08.45 init
  2 root      20   0      0    0  0 S  0.0  0.0  0:00.00 kthreadd
  3 root      20   0      0    0  0 S  0.0  0.0  0:02.10 ksoftirqd/0
  4 root      20   0      0    0  0 S  0.0  0.0  0:00.00 kworker/0:0
  5 root      0 -20      0    0  0 S  0.0  0.0  0:00.00 kworker/0:+
  6 root      20   0      0    0  0 S  0.0  0.0  0:00.16 kworker/u4+
  7 root      20   0      0    0  0 S  0.0  0.0  0:12.15 rcu_sched
  8 root      20   0      0    0  0 S  0.0  0.0  0:00.00 rcu_bh
  9 root      rt   0      0    0  0 S  0.0  0.0  0:01.07 migration/0
 10 root      rt   0      0    0  0 S  0.0  0.0  0:00.04 watchdog/0
 11 root      rt   0      0    0  0 S  0.0  0.0  0:00.26 watchdog/1
 12 root      rt   0      0    0  0 S  0.0  0.0  0:00.30 migration/1
 13 root      20   0      0    0  0 S  0.0  0.0  0:00.64 ksoftirqd/1
 14 root      20   0      0    0  0 S  0.0  0.0  0:00.00 kworker/1:0
 15 root      0 -20      0    0  0 S  0.0  0.0  0:00.00 kworker/1:+
```

这一行有 8 项统计指标。

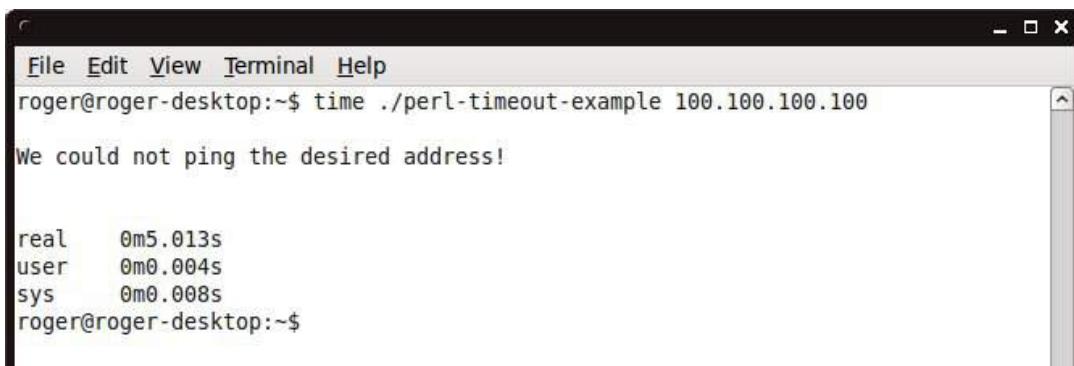
```
%Cpu(s): 24.8 us, 0.5 sy, 0.0 ni, 73.6 id, 0.4 wa, 0.0 hi, 0.2 si, 0.0 st
```

其中，第一项24.8 us (user 的缩写) 就是 CPU 消耗在 User space 的时间百分比，第二项0.5 sy (system 的缩写) 是消耗在 Kernel space 的时间百分比。

随便也说一下其他 6 个指标的含义。

- ni: niceness 的缩写，CPU 消耗在 nice 进程（低优先级）的时间百分比
- id: idle 的缩写，CPU 消耗在闲置进程的时间百分比，这个值越低，表示 CPU 越忙
- wa: wait 的缩写，CPU 等待外部 I/O 的时间百分比，这段时间 CPU 不能干其他事，但是也没有执行运算，这个值太高就说明外部设备有问题
- hi: hardware interrupt 的缩写，CPU 响应硬件中断请求的时间百分比
- si: software interrupt 的缩写，CPU 响应软件中断请求的时间百分比
- st: stole time 的缩写，该项指标只对虚拟机有效，表示分配给当前虚拟机的 CPU 时间之中，被同一台物理机上的其他虚拟机偷走的时间百分比

如果想查看单个程序的耗时，一般使用time命令。



The screenshot shows a terminal window with the following content:

```
File Edit View Terminal Help
roger@roger-desktop:~$ time ./perl-timeout-example 100.100.100.100
We could not ping the desired address!

real    0m5.013s
user    0m0.004s
sys     0m0.008s
roger@roger-desktop:~$
```

程序名之前加上time命令，会在程序执行完毕以后，默认显示三行统计。

- real：程序从开始运行到结束的全部时间，这是用户能感知到的时间，包括 CPU 切换去执行其他任务的时间。
- user：程序在 User space 执行的时间
- sys：程序在 Kernel space 执行的时间

user和sys之和，一般情况下，应该小于real。但如果是多核 CPU，这两个指标反映的是所有 CPU 的总耗时，所以它们之和可能大于real。

[参考链接]

- [User space vs kernel space](#)
- [Using the Linux Top Command](#)
- [Understanding Linux CPU stats](#)
- [What do 'real', 'user' and 'sys' mean in the output of time\(1\)?](#)

(完)

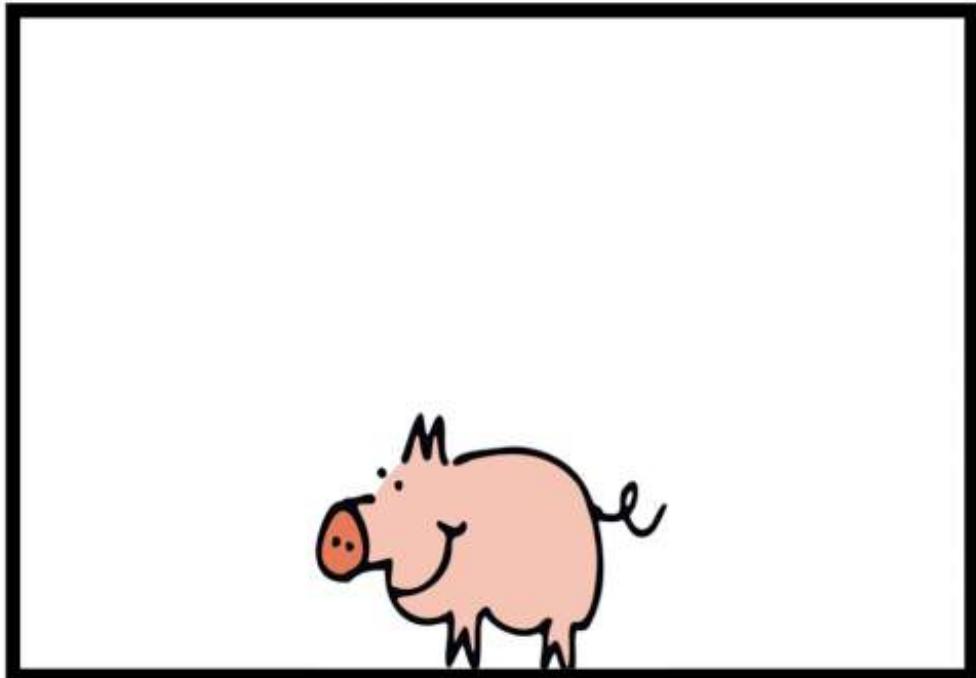
理解字节序

1.

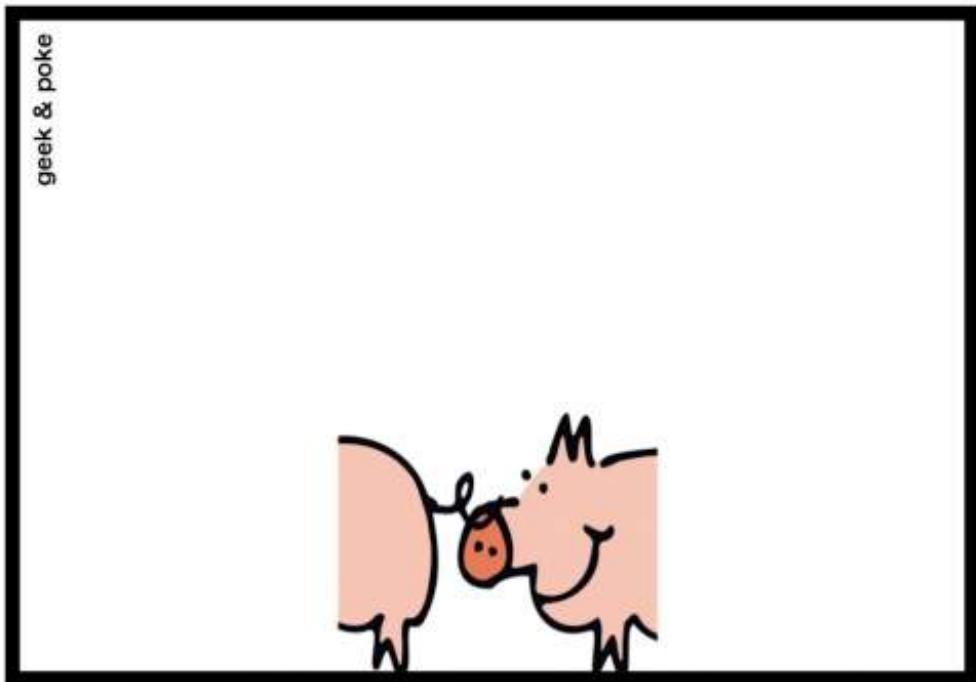
计算机硬件有两种储存数据的方式：大端字节序（big endian）和小端字节序（little endian）。

举例来说，数值0x2211使用两个字节储存：高位字节是0x22，低位字节是0x11。

- **大端字节序：**高位字节在前，低位字节在后，这是人类读写数值的方法。
- **小端字节序：**低位字节在前，高位字节在后，即以0x1122形式储存。

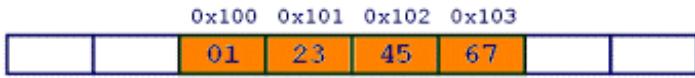


BIG-ENDIAN

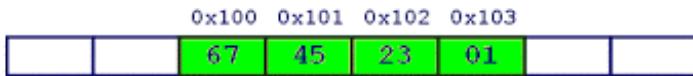


LITTLE-ENDIAN

同理，`0x1234567`的大端字节序和小端字节序的写法如下图。



Big Endian



Little Endian

2.

我一直不理解，为什么要有字节序，每次读写都要区分，多麻烦！统一使用大端字节序，不是更方便吗？

上周，我读到了一篇[文章](#)，解答了所有的疑问。而且，我发现原来的理解是错的，字节序其实很简单。

3.

首先，为什么会有小端字节序？

答案是，计算机电路先处理低位字节，效率比较高，因为计算都是从低位开始的。所以，计算机的内部处理都是小端字节序。

但是，人类还是习惯读写大端字节序。所以，除了计算机的内部处理，其他的场合几乎都是大端字节序，比如网络传输和文件储存。

4.

计算机处理字节序的时候，不知道什么是高位字节，什么是低位字节。它只知道按顺序读取字节，先读第一个字节，再读第二个字节。

如果是大端字节序，先读到的就是高位字节，后读到的就是低位字节。小端字节序正好相反。

理解这一点，才能理解计算机如何处理字节序。

5.

字节序的处理，就是一句话：

"只有读取的时候，才必须区分字节序，其他情况都不用考虑。"

处理器读取外部数据的时候，必须知道数据的字节序，将其转成正确的值。然后，就正常使用这个值，完全不用再考虑字节序。

即使是向外部设备写入数据，也不用考虑字节序，正常写入一个值即可。外部设备会自己处理字节序的问题。

6.

举例来说，处理器读入一个16位整数。如果是大端字节序，就按下面的方式转成值。

```
x = buf[offset] * 256 + buf[offset+1];
```

上面代码中，buf是整个数据块在内存中的起始地址，offset是当前正在读取的位置。第一个字节乘以256，再加上第二个字节，就是大端字节序的值，这个式子可以用逻辑运算符改写。

```
x = buf[offset]<<8 | buf[offset+1];
```

上面代码中，第一个字节左移8位（即后面添8个0），然后再与第二个字节进行或运算。

如果是小端字节序，用下面的公式转成值。

```
x = buf[offset+1] * 256 + buf[offset];
```

32位整数的求值公式也是一样的。

```
/* 大端字节序 */
i = (data[3]<<0) | (data[2]<<8) | (data[1]<<16) | (data[0]<<24);
```

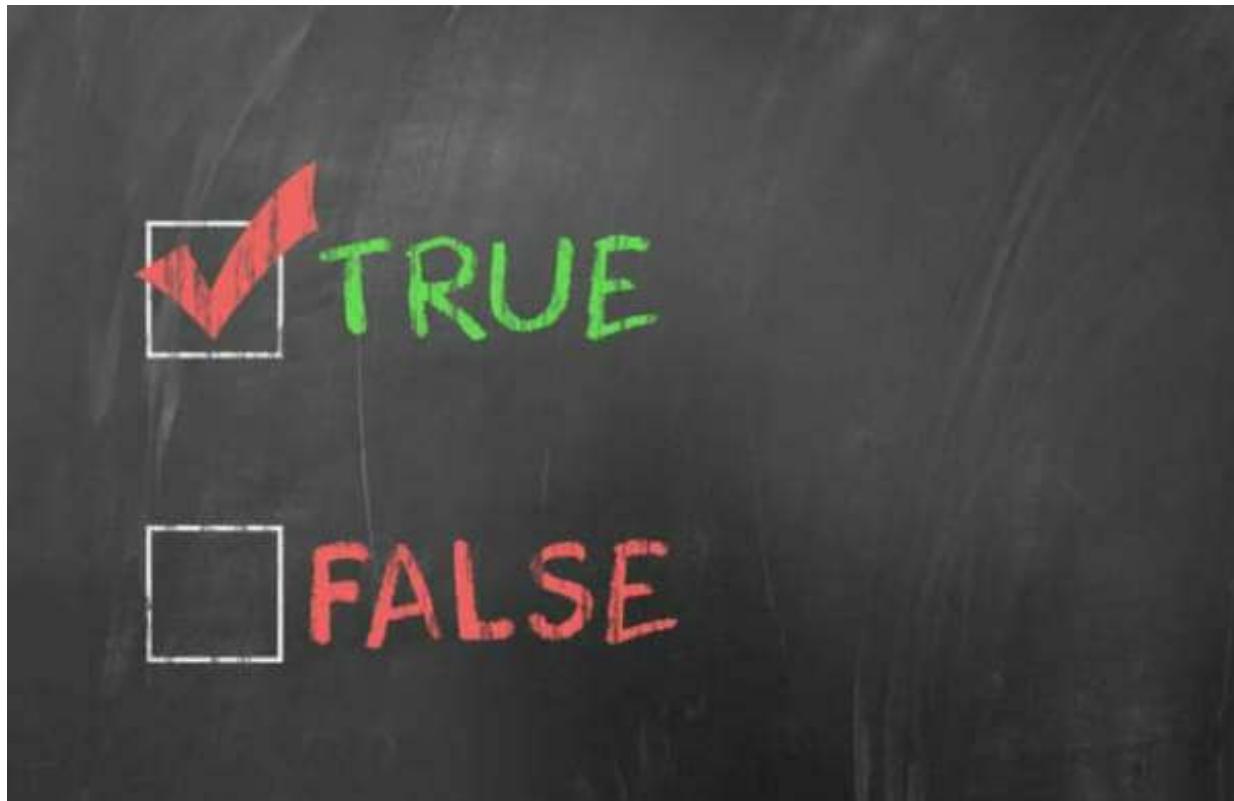
```
/* 小端字节序 */
i = (data[0]<<0) | (data[1]<<8) | (data[2]<<16) | (data[3]<<24);
```

(完)

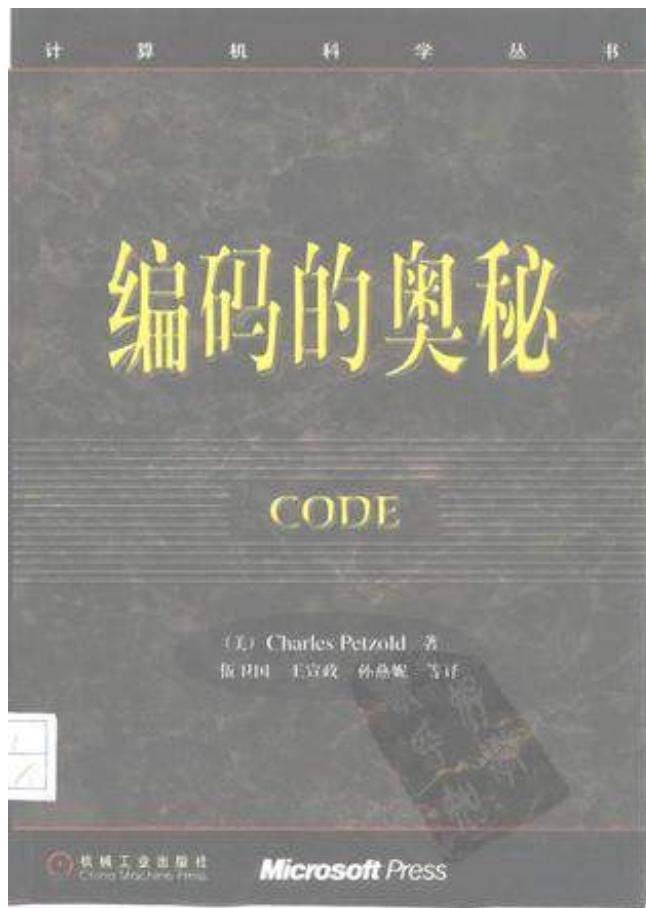
布尔代数入门

[布尔代数](#)是计算机的基础。没有它，就不会有计算机。

布尔代数发展到今天，已经非常抽象，但是它的核心思想很简单。本文帮助你理解布尔代数，以及为什么它促成了计算机的诞生。



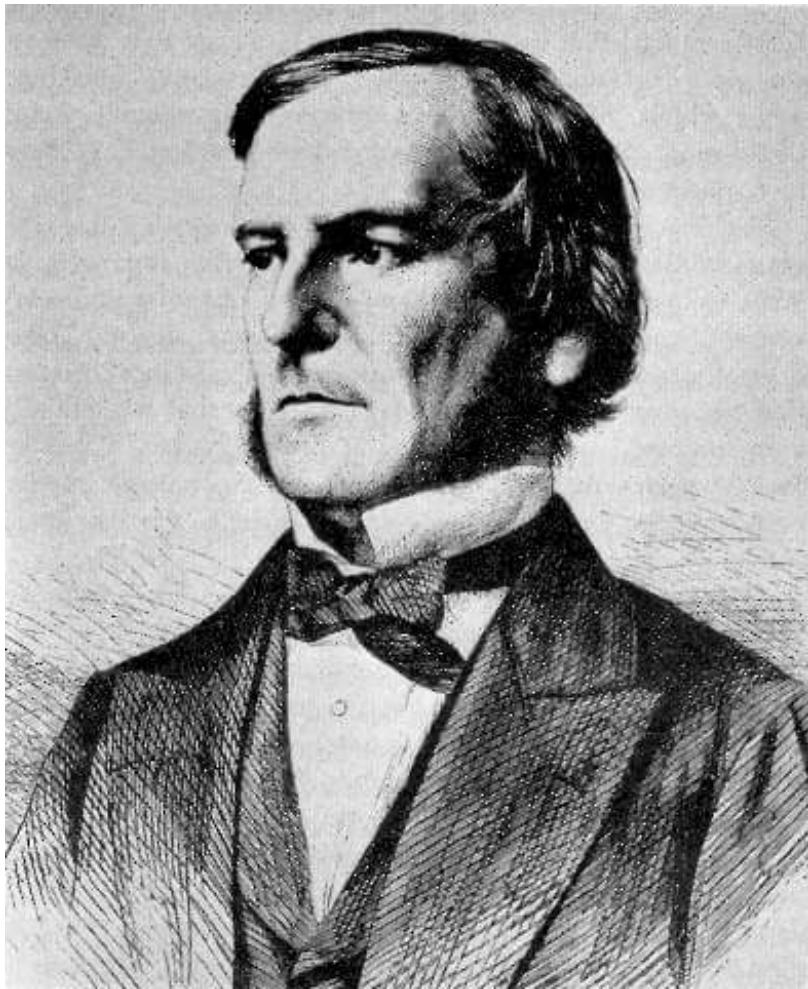
我依据的是《编码的奥妙》的第十章。这是一本好书，强烈推荐。



一、数理逻辑的起源

19世纪早期，英国数学家乔治·布尔（George Boole，1815—1864）突发奇想：人的思想能不能用数学表达？

此前，数学只用于计算，没有人意识到，数学还能表达人的逻辑思维。



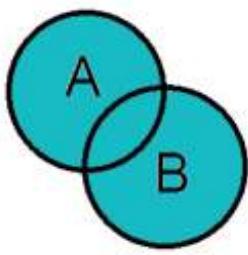
两千年，哲学书都是用文字写的。比如，最著名的三段论：

所有人都是要死的，
苏格拉底是人，
所以，苏格拉底是要死的。

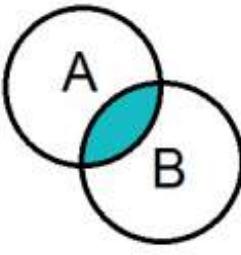
乔治·布尔认为，这种推理可以用数学表达，也就是说，哲学书完全可以用数学写。这就是数理逻辑的起源。

二、集合论

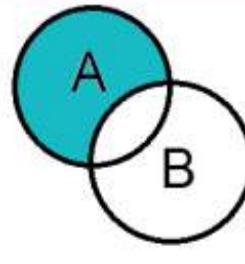
乔治·布尔发明的工具，叫做“集合论”（Set theory）。他认为，逻辑思维的基础是一一个个集合（Set），每一个命题表达的都是集合之间的关系。



A OR B



A AND B



A NOT B

比如，所有人类组成一个集合R，所有会死的东西组成一个集合D。

所有人都是要死的

集合论的写法就是：

$$R \times D = R$$

集合之间最基本的关系是并集和交集。乘号 (\times) 表示交集，加号 ($+$) 表示并集。上面这个式子的意思是，R与D的交集就是R。

同样的，苏格拉底也是一个集合S，这个集合里面只有苏格拉底一个成员。

苏格拉底是人

// 等同于

$$S \times R = S$$

上面式子的意思是，苏格拉底与人类的交集，就是苏格拉底。

将第一个式子代入第二个式子，就得到了结论。

$$\begin{aligned} S \times (R \times D) \\ &= (S \times R) \times D \\ &= S \times D \\ &= S \end{aligned}$$

这个式子的意思是，苏格拉底与会死的东西的交集，就是苏格拉底，即苏格拉底也属于会死的东西。

三、集合的运算法则

前面的三段论比较容易，一眼就能看出结论。但是，有些三段论比较复杂，不容易立即反应过来。

请看下面这两句话。

"鸭嘴兽是卵生的哺乳动物。鸭嘴兽是澳洲的动物。"

你能一眼得到结论吗？

鸭嘴兽 X 卵生 = 鸭嘴兽

鸭嘴兽 x 澳洲 = 鸭嘴兽

将第一个式子代入第二个，就会得到：

鸭嘴兽 X 卵生 x 澳洲 = 鸭嘴兽

// 相当于

卵生 x 澳洲 = 鸭嘴兽 + 其他

因此，结论就是“有的卵生动物是澳洲的动物”，或者“有的澳洲的动物是卵生动物”。

还有更不直观的三段论。

“哲学家都是有逻辑头脑的，一个没有逻辑头脑的人总是很顽固。”

请问结论是什么？

这道题会用到新的概念：全集和空集。集合A和所有不属于它的元素（记作-A）构成全集（I），这时A和-A的交集就是一个空集（0）。

$$A + (-A) = I$$

$$A \times (-A) = 0$$

因此，有下面的公式。

$$B$$

$$= B \times I$$

$$= B \times (A + -A)$$

$$= B \times A + B \times (-A)$$

回到上面那道题。

哲学家 X 逻辑 = 哲学家

无逻辑 X 顽固 = 无逻辑

根据第一个命题，可以得到下面的结论。

哲学家 X 无逻辑

= (哲学家 X 逻辑) X 无逻辑

= 哲学家 X (逻辑 X 无逻辑)

= 哲学家 X 0

= 0

即哲学家与没有逻辑的人的交集，是一个空集。

根据第二个命题，可以得到下面的结论。

无逻辑 X 顽固

= 无逻辑 X 顽固 X (哲学家 + 非哲学家)

$$\begin{aligned}
 &= \text{无逻辑} \times \text{顽固} \times \text{哲学家} + \text{无逻辑} \times \text{顽固} \times \text{非哲学家} \\
 &= 0 \times \text{顽固} + \text{无逻辑} \times \text{顽固} \times \text{非哲学家} \\
 &= \text{无逻辑} \times \text{顽固} \times \text{非哲学家} \\
 &= \text{无逻辑}
 \end{aligned}$$

也就是说，最终的结论如下。

$$\text{无逻辑} \times \text{顽固} \times \text{非哲学家} = \text{无逻辑}$$

// 相当于

$$\text{顽固} \times \text{非哲学家} = \text{无逻辑} + \text{其他}$$

结论就是顽固的人与非哲学家之间有交集。通俗的表达就是：一些顽固的人，不是哲学家，或者一些不是哲学家的人，很顽固。

由此可见，集合论可以帮助我们得到直觉无法得到的结论，保证推理过程正确，比文字推导更可靠。

$$\begin{aligned}
 &(\bar{A} + \bar{B} + C)(\bar{A}\bar{B} + \bar{A}\bar{C}) \\
 &= (\bar{A} + \bar{B} + C)(\bar{A}\bar{B} \cdot \bar{A}\bar{C}) \\
 &= (\bar{A} + \bar{B} + C)(\bar{A} + \bar{B})(\bar{A} + \bar{C}) \\
 &= (\bar{A} + \bar{B} + C)(\cancel{\bar{A}\bar{A}} + \bar{A}\bar{C} + \bar{A}\bar{B} + \bar{B}\bar{C}) \\
 &= (\bar{A} + \bar{B} + C)(\bar{A}\bar{C} + \bar{A}\bar{B} + \bar{B}\bar{C}) \\
 &= \cancel{A}\bar{A}\bar{C} + \cancel{A}\bar{A}\bar{B} + \cancel{A}\bar{B}\bar{C} + \cancel{\bar{A}\bar{B}} + \cancel{\bar{B}\bar{C}} + \cancel{\bar{A}\bar{C}} + \cancel{A}\bar{B}\bar{C} + \cancel{\bar{B}\bar{C}} \\
 &= \bar{A}\bar{B} + \bar{A}\bar{C} + \bar{B}\bar{C} +
 \end{aligned}$$

四、集合论到布尔代数

既然命题可以用集合论表达，那么逻辑推导无非就是一系列集合运算。

由于集合运算的结果还是集合，那么通过判断个体是否属于指定集合，就可以计算命题的真伪。

一名顾客走进宠物店，对店员说：“我想要一只公猫，白色或黄色均可；或者一只母猫，除了白色，其他颜色均可；或者只要是黑猫，我也要。”

这名顾客的要求用集合论表达，就是下面的式子。

公猫 X (白色 + 黄色)
+ 母猫 X 非白色
+ 黑猫

店员拿出一只灰色的公猫，请问是否满足要求？

布尔代数规定，个体属于某个集合用1表示，不属于就用0表示。 灰色的公猫属于公猫集合，就是1，不属于白色集合，就是0。

上面的表达式变成下面这样。

$$\begin{aligned} & 1 \times (0 + 0) \\ & + 0 \times 1 \\ & + 0 \\ & = 0 \end{aligned}$$

因此，就得到结论，灰色的公猫不满足要求。

这就是布尔代数：计算命题真伪的数学方法。

五、布尔代数的运算法则

布尔代数的运算法则与集合论很像。

交集的运算法则如下。

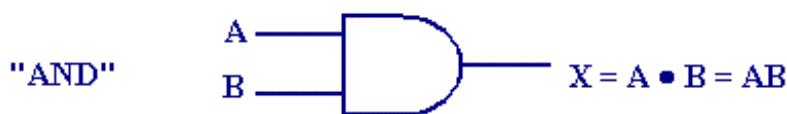
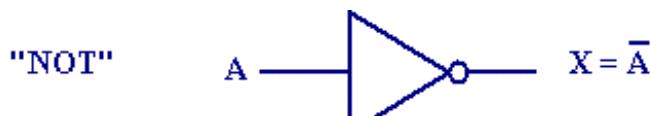
$$\begin{aligned} 1 \times 1 &= 1 \\ 1 \times 0 &= 0 \\ 0 \times 0 &= 0 \end{aligned}$$

并集的运算法则如下。

$$\begin{aligned} 1 + 1 &= 1 \\ 1 + 0 &= 1 \\ 0 + 0 &= 0 \end{aligned}$$

集合论可以描述逻辑推理过程，布尔代数可以判断某个命题是否符合这个过程。人类的推理和判断，因此就变成了数学运算。

20世纪初，英国科学家香农指出，布尔代数可以用来描述电路，或者说，电路可以模拟布尔代数。于是，人类的推理和判断，就可以用电路实现了。这就是计算机的实现基础。



六、布尔代数的局限

虽然布尔代数可以判断命题真伪，但是无法取代人类的理性思维。原因是它有一个局限。

它必须依据一个或几个已经明确知道真伪的命题，才能做出判断。比如，只有知道“所有人都会死”这个命题是真的，才能得出结论“苏格拉底会死”。

布尔代数只能保证推理过程正确，无法保证推理所依据的前提是否正确。如果前提是错的，正确的推理也会得到错误的结果。而前提的真伪要由科学实验和观察来决定，布尔代数无能为力。

(完)

DNS 原理入门

DNS 是互联网核心协议之一。不管是上网浏览，还是编程开发，都需要了解一点它的知识。

本文详细介绍DNS的原理，以及如何运用工具软件观察它的运作。我的目标是，读完此文后，你就能完全理解DNS。



一、DNS 是什么？

DNS（Domain Name System 的缩写）的作用非常简单，就是根据域名查出IP地址。你可以把它想象成一本巨大的电话本。

举例来说，如果你要访问域名math.stackexchange.com，首先要通过DNS查出它的IP地址是151.101.129.69。

如果你不清楚为什么一定要查出IP地址，才能进行网络通信，建议先阅读我写的[《互联网协议入门》](#)。

二、查询过程

虽然只需要返回一个IP地址，但是DNS的查询过程非常复杂，分成多个步骤。

工具软件dig可以显示整个查询过程。

```
$ dig math.stackexchange.com
```

上面的命令会输出六段信息。

```

; <>> DiG 9.9.5-12.1-Debian <>> math.stackexchange.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6328
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 4, ADDITIONAL: 4

;; QUESTION SECTION:
;math.stackexchange.com.          IN      A

;; ANSWER SECTION:
math.stackexchange.com. 600      IN      A      151.101.65.69
math.stackexchange.com. 600      IN      A      151.101.129.69
math.stackexchange.com. 600      IN      A      151.101.193.69
math.stackexchange.com. 600      IN      A      151.101.1.69

;; AUTHORITY SECTION:
stackexchange.com.    171469   IN      NS     ns-1832.awsdns-37.co.uk.
stackexchange.com.    171469   IN      NS     ns-1029.awsdns-00.org.
stackexchange.com.    171469   IN      NS     ns-463.awsdns-57.com.
stackexchange.com.    171469   IN      NS     ns-925.awsdns-51.net.

;; ADDITIONAL SECTION:
ns-463.awsdns-57.com. 171406   IN      A      205.251.193.207
ns-925.awsdns-51.net. 171393   IN      A      205.251.195.157
ns-1029.awsdns-00.org. 171469   IN      A      205.251.196.5
ns-1832.awsdns-37.co.uk. 171393 IN      A      205.251.199.40

;; Query time: 5 msec
;; SERVER: 192.168.1.253#53(192.168.1.253)
;; WHEN: Mon Jun 13 22:12:46 CST 2016
;; MSG SIZE  rcvd: 305

```

第一段是查询参数和统计。

```

; <>> DiG 9.9.5-12.1-Debian <>> math.stackexchange.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6328
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 4, ADDITIONAL: 4

```

第二段是查询内容。

```

;; QUESTION SECTION:
;math.stackexchange.com.          IN      A

```

上面结果表示，查询域名math.stackexchange.com的A记录，A是address的缩写。

第三段是DNS服务器的答复。

```

;; ANSWER SECTION:
math.stackexchange.com. 600      IN      A      151.101.65.69
math.stackexchange.com. 600      IN      A      151.101.129.69
math.stackexchange.com. 600      IN      A      151.101.193.69
math.stackexchange.com. 600      IN      A      151.101.1.69

```

上面结果显示，math.stackexchange.com有四个A记录，即四个IP地址。600是TTL值（Time to live 的缩写），表示缓存时间，即600秒之内不用重新查询。

第四段显示stackexchange.com的NS记录（Name Server的缩写），即哪些服务器负责管理stackexchange.com的DNS记录。

```
;; AUTHORITY SECTION:  
stackexchange.com. 171469 IN NS ns-1832.awsdns-37.co.uk.  
stackexchange.com. 171469 IN NS ns-1029.awsdns-00.org.  
stackexchange.com. 171469 IN NS ns-463.awsdns-57.com.  
stackexchange.com. 171469 IN NS ns-925.awsdns-51.net.
```

上面结果显示stackexchange.com共有四条NS记录，即四个域名服务器，向其中任一台查询就能知道math.stackexchange.com的IP地址是什么。

第五段是上面四个域名服务器的IP地址，这是随着前一段一起返回的。

```
;; ADDITIONAL SECTION:  
ns-463.awsdns-57.com. 171406 IN A 205.251.193.207  
ns-925.awsdns-51.net. 171393 IN A 205.251.195.157  
ns-1029.awsdns-00.org. 171469 IN A 205.251.196.5  
ns-1832.awsdns-37.co.uk. 171393 IN A 205.251.199.40
```

第六段是DNS服务器的一些传输信息。

```
;; Query time: 7 msec  
;; SERVER: 192.168.1.253#53(192.168.1.253)  
;; WHEN: Wed Jun 15 23:23:55 CST 2016  
;; MSG SIZE rcvd: 305
```

上面结果显示，本机的DNS服务器是192.168.1.253，查询端口是53（DNS服务器的默认端口），以及回应长度是305字节。

如果不想要看到这么多内容，可以使用+short参数。

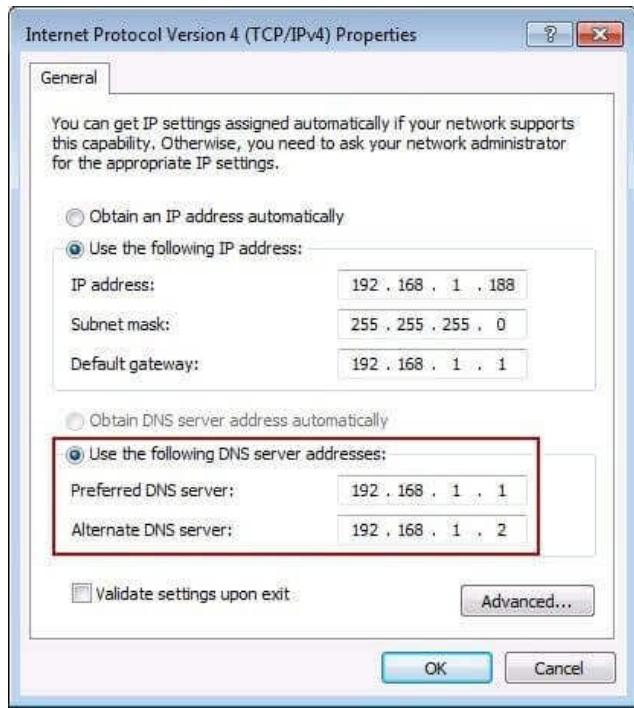
```
$ dig +short math.stackexchange.com  
  
151.101.129.69  
151.101.65.69  
151.101.193.69  
151.101.1.69
```

上面命令只返回math.stackexchange.com对应的4个IP地址（即A记录）。

三、DNS服务器

下面我们根据前面这个例子，一步步还原，本机到底怎么得到域名math.stackexchange.com的IP地址。

首先，本机一定要知道DNS服务器的IP地址，否则上不了网。通过DNS服务器，才能知道某个域名的IP地址到底是什么。



DNS服务器的IP地址，有可能是动态的，每次上网时由网关分配，这叫做DHCP机制；也有可能是事先指定的固定地址。Linux系统里面，DNS服务器的IP地址保存在/etc/resolv.conf文件。

上例的DNS服务器是192.168.1.253，这是一个内网地址。有一些公网的DNS服务器，也可以使用，其中最有名的就是Google的[8.8.8.8](#)和Level 3的[4.2.2.2](#)。

本机只向自己的DNS服务器查询，dig命令有一个@参数，显示向其他DNS服务器查询的结果。

```
$ dig @4.2.2.2 math.stackexchange.com
```

上面命令指定向DNS服务器4.2.2.2查询。

四、域名的层级

DNS服务器怎么会知道每个域名的IP地址呢？答案是分级查询。

请仔细看前面的例子，每个域名的尾部都多了一个点。

```
; QUESTION SECTION:  
;math.stackexchange.com.          IN      A
```

比如，域名math.stackexchange.com显示为math.stackexchange.com.。这不是疏忽，而是所有域名的尾部，实际上都有一个根域名。

举例来说，www.example.com真正的域名是www.example.com.root，简写为www.example.com.。因为，根域名.root对于所有域名都是一样的，所以平时是省略的。

根域名的下一级，叫做"顶级域名"（top-level domain，缩写为TLD），比如.com、.net；再下一级叫做"次级域名"（second-level domain，缩写为SLD），比如www.example.com里面的.example，这一级域名是用户可以注册的；再下一级是主机名（host），比如www.example.com里面的www，又称为"三级域名"，这是用户在自己的域里面为服务器分配的名称，是用户可以任意分配的。

总结一下，域名的层级结构如下。

主机名.次级域名.顶级域名.根域名

即

```
host.sld.tld.root
```

五、根域名服务器

DNS服务器根据域名的层级，进行分级查询。

需要明确的是，每一级域名都有自己的NS记录，NS记录指向该级域名的域名服务器。这些服务器知道下一级域名的各种记录。

所谓“分级查询”，就是从根域名开始，依次查询每一级域名的NS记录，直到查到最终的IP地址，过程大致如下。

1. 从“根域名服务器”查到“顶级域名服务器”的NS记录和A记录（IP地址）
2. 从“顶级域名服务器”查到“次级域名服务器”的NS记录和A记录（IP地址）
3. 从“次级域名服务器”查出“主机名”的IP地址

仔细看上面的过程，你可能发现了，没有提到DNS服务器怎么知道“根域名服务器”的IP地址。回答是“根域名服务器”的NS记录和IP地址一般是不会变化的，所以内置在DNS服务器里面。

下面是内置的根域名服务器IP地址的一个例子。

```
; formerly NS.INTERNIC.NET
;
.
3600000 IN NS A.ROOT-SERVERS.NET.
A.ROOT-SERVERS.NET. 3600000 A 198.41.0.4
A.ROOT-SERVERS.NET. 3600000 AAAA 2001:503:BA3E::2:30
;
; formerly NS1.ISI.EDU
;
.
3600000 NS B.ROOT-SERVERS.NET.
B.ROOT-SERVERS.NET. 3600000 A 192.228.79.201
;
; formerly C.PSI.NET
;
.
3600000 NS C.ROOT-SERVERS.NET.
C.ROOT-SERVERS.NET. 3600000 A 192.33.4.12
```

上面列表中，列出了根域名（.root）的三条NS记录A.ROOT-SERVERS.NET、B.ROOT-SERVERS.NET和C.ROOT-SERVERS.NET，以及它们的IP地址（即A记录）198.41.0.4、192.228.79.201、192.33.4.12。

另外，可以看到所有记录的TTL值是3600000秒，相当于1000小时。也就是说，每1000小时才查询一次根域名服务器的列表。

目前，世界上一共有十三组根域名服务器，从A.ROOT-SERVERS.NET一直到M.ROOT-SERVERS.NET。

六、分级查询的实例

dig命令的+trace参数可以显示DNS的整个分级查询过程。

```
$ dig +trace math.stackexchange.com
```

上面命令的第一段列出根域名的所有NS记录，即所有根域名服务器。

```
; <>> DiG 9.9.5-12.1-Debian <>> +trace math.stackexchange.com
;; global options: +cmd

.          318820  IN      NS      b.root-servers.net.
.          318820  IN      NS      h.root-servers.net.
.          318820  IN      NS      e.root-servers.net.
.          318820  IN      NS      l.root-servers.net.
.          318820  IN      NS      g.root-servers.net.
.          318820  IN      NS      k.root-servers.net.
.          318820  IN      NS      j.root-servers.net.
.          318820  IN      NS      c.root-servers.net.
.          318820  IN      NS      m.root-servers.net.
.          318820  IN      NS      i.root-servers.net.
.          318820  IN      NS      a.root-servers.net.
.          318820  IN      NS      d.root-servers.net.
.          318820  IN      NS      f.root-servers.net.
```

根据内置的根域名服务器IP地址，DNS服务器向所有这些IP地址发出查询请求，询问math.stackexchange.com的顶级域名服务器com的NS记录。最先回复的根域名服务器将被缓存，以后只向这台服务器发请求。

接着是第二段。

```
com.          172800  IN      NS      a.gtld-servers.net.  
com.          172800  IN      NS      b.gtld-servers.net.  
com.          172800  IN      NS      c.gtld-servers.net.  
com.          172800  IN      NS      d.gtld-servers.net.  
com.          172800  IN      NS      e.gtld-servers.net.  
com.          172800  IN      NS      f.gtld-servers.net.  
com.          172800  IN      NS      g.gtld-servers.net.  
com.          172800  IN      NS      h.gtld-servers.net.  
com.          172800  IN      NS      i.gtld-servers.net.  
com.          172800  IN      NS      j.gtld-servers.net.  
com.          172800  IN      NS      k.gtld-servers.net.  
com.          172800  IN      NS      l.gtld-servers.net.  
com.          172800  IN      NS      m.gtld-servers.net.
```

上面结果显示.com域名的13条NS记录，同时返回的还有每一条记录对应的IP地址。

然后，DNS服务器向这些顶级域名服务器发出查询请求，询问math.stackexchange.com的次级域名stackexchange.com的NS记录。

stackexchange.com.	172800	IN	NS	ns-463.awsdns-57.com.
stackexchange.com.	172800	IN	NS	ns-925.awsdns-51.net.
stackexchange.com.	172800	IN	NS	ns-1029.awsdns-00.org.
stackexchange.com.	172800	IN	NS	ns-1832.awsdns-37.co.uk.

上面结果显示stackexchange.com有四条NS记录，同时返回的还有每一条NS记录对应的IP地址。

然后，DNS服务器向上面这四台NS服务器查询math.stackexchange.com的主机名。

math.stackexchange.com.	300	IN	A	151.101.65.69
math.stackexchange.com.	300	IN	A	151.101.193.69
math.stackexchange.com.	300	IN	A	151.101.129.69
math.stackexchange.com.	300	IN	A	151.101.1.69
stackexchange.com.	172800	IN	NS	ns-1029.awsdns-00.org.
stackexchange.com.	172800	IN	NS	ns-1832.awsdns-37.co.uk.
stackexchange.com.	172800	IN	NS	ns-463.awsdns-57.com.
stackexchange.com.	172800	IN	NS	ns-925.awsdns-51.net.
;; Received 252 bytes from 205.251.193.207#53(ns-463.awsdns-57.com) in 226 ms				

上面结果显示，math.stackexchange.com有4条A记录，即这四个IP地址都可以访问到网站。并且还显示，最先返回结果的NS服务器是ns-463.awsdns-57.com，IP地址为205.251.193.207。

七、NS记录的查询

dig命令可以单独查看每一级域名的NS记录。

```
$ dig ns com
$ dig ns stackexchange.com
```

+short参数可以显示简化的结果。

```
$ dig +short ns com
$ dig +short ns stackexchange.com
```

八、DNS的记录类型

域名与IP之间的对应关系，称为"记录"（record）。根据使用场景，"记录"可以分成不同的类型（type），前面已经看到了有A记录和NS记录。

常见的DNS记录类型如下。

- (1) A：地址记录（Address），返回域名指向的IP地址。
- (2) NS：域名服务器记录（Name Server），返回保存下一级域名信息的服务器地址。该记录只能设置为域名，不能设置为IP地址。
- (3) MX：邮件记录（Mail eXchange），返回接收电子邮件的服务器地址。
- (4) CNAME：规范名称记录（Canonical Name），返回另一个域名，即当前查询的域名是另一个域名的跳转，详见下文。
- (5) PTR：逆向查询记录（Pointer Record），只用于从IP地址查询域名，详见下文。

一般来说，为了服务的安全可靠，至少应该有两条NS记录，而A记录和MX记录也可以有多条，这样就提供了服务的冗余性，防止出现单点失败。

CNAME记录主要用于域名的内部跳转，为服务器配置提供灵活性，用户感知不到。举例来说，facebook.github.io这个域名就是一个CNAME记录。

```
$ dig facebook.github.io  
...  
;; ANSWER SECTION:  
facebook.github.io. 3370 IN CNAME github.map.fastly.net.  
github.map.fastly.net. 600 IN A 103.245.222.133
```

上面结果显示，facebook.github.io的CNAME记录指向github.map.fastly.net。也就是说，用户查询facebook.github.io的时候，实际上返回的是github.map.fastly.net的IP地址。这样的好处是，变更服务器IP地址的时候，只要修改github.map.fastly.net这个域名就可以了，用户的facebook.github.io域名不用修改。

由于CNAME记录就是一个替换，所以域名一旦设置CNAME记录以后，就不能再设置其他记录了（比如A记录和MX记录），这是为了防止产生冲突。举例来说，foo.com指向bar.com，而两个域名各有自己的MX记录，如果两者不一致，就会产生问题。由于顶级域名通常要设置MX记录，所以一般不允许用户对顶级域名设置CNAME记录。

PTR记录用于从IP地址反查域名。dig命令的-x参数用于查询PTR记录。

```
$ dig -x 192.30.252.153  
...  
;; ANSWER SECTION:  
153.252.30.192.in-addr.arpa. 3600 IN PTR pages.github.com.
```

上面结果显示，192.30.252.153这台服务器的域名是pages.github.com。

逆向查询的一个应用，是可以防止垃圾邮件，即验证发送邮件的IP地址，是否真的有它所声称的域名。

dig命令可以查看指定的记录类型。

```
$ dig a github.com  
$ dig ns github.com  
$ dig mx github.com
```

九、其他DNS工具

除了dig，还有一些其他小工具也可以使用。

(1) host 命令

host命令可以看作dig命令的简化版本，返回当前请求域名的各种记录。

```
$ host github.com  
github.com has address 192.30.252.121  
github.com mail is handled by 5 ALT2.ASPMX.L.GOOGLE.COM.  
github.com mail is handled by 10 ALT4.ASPMX.L.GOOGLE.COM.  
github.com mail is handled by 10 ALT3.ASPMX.L.GOOGLE.COM.  
github.com mail is handled by 5 ALT1.ASPMX.L.GOOGLE.COM.  
github.com mail is handled by 1 ASPMX.L.GOOGLE.COM.
```

```
$ host facebook.github.com  
facebook.github.com is an alias for github.map.fastly.net.  
github.map.fastly.net has address 103.245.222.133
```

host命令也可以用于逆向查询，即从IP地址查询域名，等同于dig -x <ip>。

```
$ host 192.30.252.153  
153.252.30.192.in-addr.arpa domain name pointer pages.github.com.
```

(2) nslookup 命令

nslookup命令用于互动式地查询域名记录。

```
$ nslookup  
> facebook.github.io  
Server: 192.168.1.253  
Address: 192.168.1.253#53  
  
Non-authoritative answer:  
facebook.github.io canonical name = github.map.fastly.net.  
Name: github.map.fastly.net  
Address: 103.245.222.133  
  
>
```

(3) whois 命令

whois命令用来查看域名的注册情况。

```
$ whois github.com
```

十、参考链接

- [DNS: The Good Parts](#), by Pete Keen
- [DNS 101](#), by Mark McDonnell

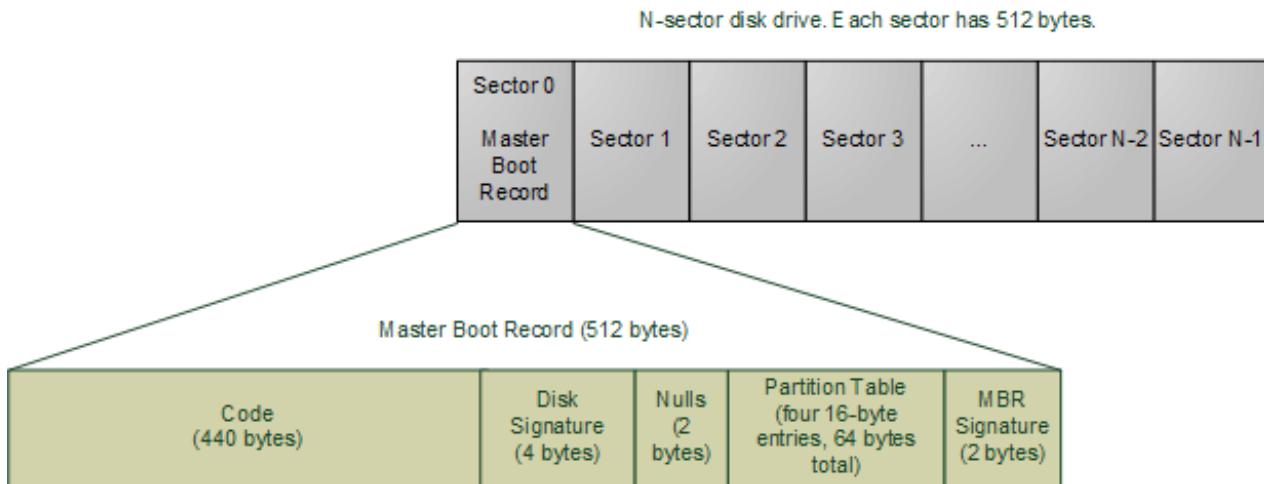
(完)

为什么主引导记录的内存地址是0x7C00?

《计算机原理》课本说，启动时，[主引导记录](#)会存入内存地址0x7C00。

这个奇怪的地址，是怎么来的，课本就不解释了。我一直有疑问，为什么不存入内存的头部、尾部、或者其他位置，而偏偏存入这个比32KB小1024字节的地方？

昨天，我读到一篇[文章](#)，终于解开了这个谜。



首先，如果你不知道，主引导记录（Master boot record，缩写为MBR）是什么，可以先读[《计算机是如何启动的？》](#)。

简单说，计算机启动是这样一个过程。

1. 通电
2. 读取ROM里面的BIOS，用来检查硬件
3. 硬件检查通过
4. **BIOS根据指定的顺序，检查引导设备的第一个扇区（即主引导记录），加载在内存地址 0x7C00**
5. 主引导记录把操作权交给操作系统

所以，主引导记录就是引导“操作系统”进入内存的一段小程序，大小不超过1个扇区（512字节）。



0x7C00这个地址来自Intel的第一代个人电脑芯片8088，以后的CPU为了保持兼容，一直使用这个地址。



1981年8月，IBM公司最早的个人电脑IBM PC 5150上市，就用了这个芯片。

```
A:asm mon  
Seattle Computer Products  
Copyright 1979,80,81 by Se  
  
Error Count = 0  
A:hex2bin mon  
A:_
```

当时，搭配的操作系统是[86-DOS](#)。这个操作系统需要的内存最少是32KB。我们知道，内存地址从0x0000开始编号，32KB的内存就是0x0000~0x7FFF。

8088芯片本身需要占用0x0000~0x03FF，用来保存[各种中断处理程序](#)的储存位置。（主引导记录本身就是中断信号INT 19h的处理程序。）所以，内存只剩下0x0400~0x7FFF可以使用。

为了把尽量多的连续内存留给操作系统，主引导记录就被放到了内存地址的尾部。由于一个扇区是512字节，主引导记录本身也会产生数据，需要另外留出512字节保存。所以，它的预留位置就变成了：

$$0x7FFF - 512 - 512 + 1 = 0x7C00$$

0x7C00就是这样来的。

计算机启动后，32KB内存的使用情况如下。

```
+----- 0x0  
| Interrupts vectors  
+----- 0x400  
| BIOS data area  
+----- 0x5??  
| OS load area  
+----- 0x7C00  
| Boot sector  
+----- 0x7E00  
| Boot data/stack  
+----- 0x7FFF  
| (not used)  
+----- (...)
```

(完)

图解 Monad

函数式编程有一个重要概念，叫做[Monad](#)。

Monad (functional programming)

网上有很多解释（[这里](#)和[这里](#)），但都很抽象，不容易看懂。我尝试了好多次，还是不明白Monad到底是什么。

Monad

In functional programming, a monad is a structure that represents computations defined as sequences of steps.

A type with a monad structure defines what it means to chain operations, or nest functions of that type together

http://en.wikipedia.org/wiki/Monad_%28functional_programming%29

p3 [rammina%29](#)

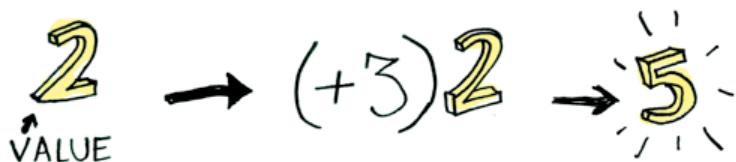
昨天，我读到了[Aditya Bhargava](#)的文章，他画了很多图。我想了半天，终于恍然大悟。下面，我就用这些图来解释Monad。

1.



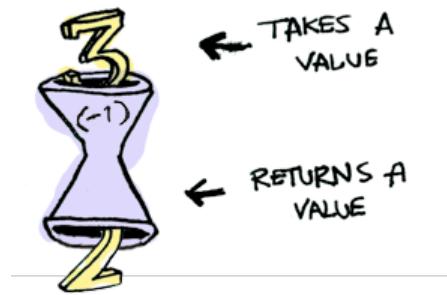
软件最基本的数据，就是各种值（value）。

2.



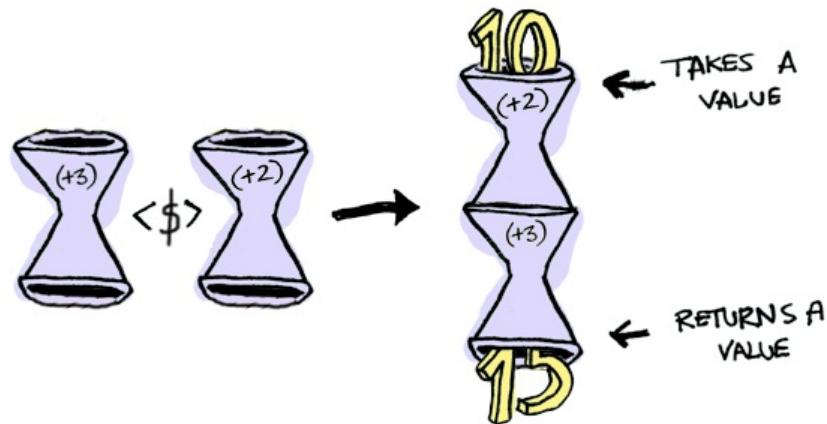
处理值的一系列操作，可以封装成函数。输入一个值，会得到另一个值。上图的"(+3)"就是一个函数，对输入的值加上3，再输出。

3.



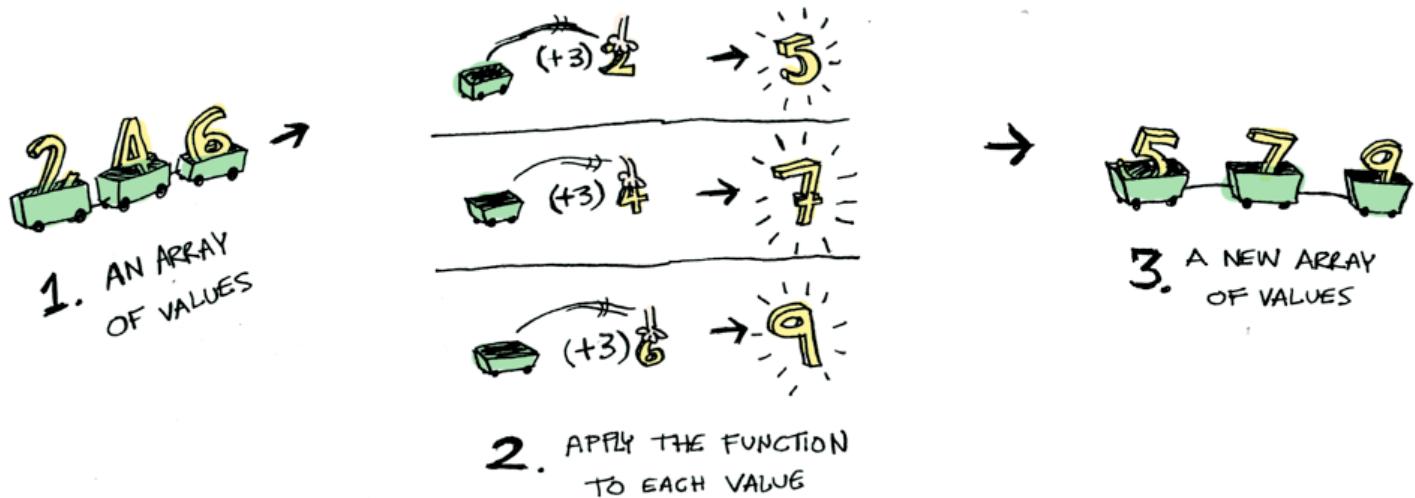
函数很像漏斗，上面进入一个值，下面出来一个值。

4.



函数可以连接起来使用，一个函数接着另一个函数。

5.



函数还可以依次处理数据集合的每个成员。

6.

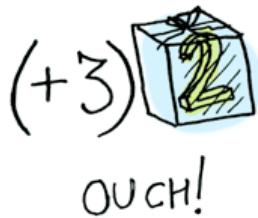


Just 2
VALUE
AND
CONTEXT

说完了函数，再来看第二个概念：数据类型（type）。

数据类型就是对值的一种封装，不仅包括值本身，还包括相关的属性和方法。上图就是2的封装，从此2就不是一个单纯的值，而是一种数据类型的实例，只能在数据类型的场景（context）中使用。

7.



2变成数据类型以后，原来的函数就不能用了。因为"(+3)"这个函数是处理值的（简称"值函数"），而不是处理数据类型的。

8.



我们需要重新定义一种运算。它接受"值函数"和数据类型的实例作为输入参数，使用"值函数"处理后，再输出数据类型的另一个实例。上图的fmap就代表了这种运算。

9.



fmap的内部，实际上是这样：打开封装的数据类型，取出值，用值函数处理以后，再封装回数据类型。

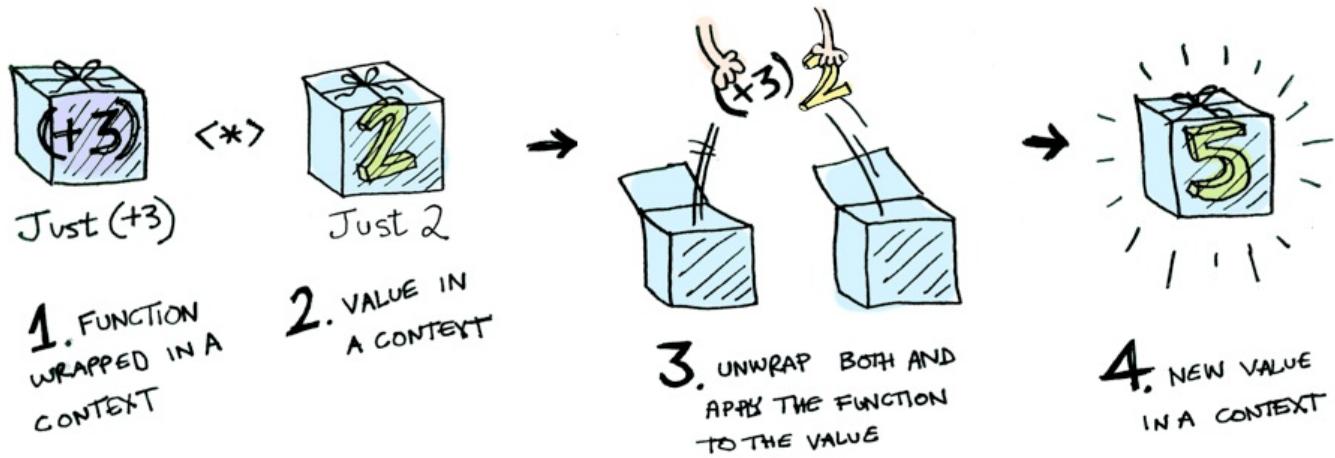
10.



一个有趣的问题来了。如果我们把函数也封装成数据类型，会怎样？

上图就是把函数"(+3)"封装成一种数据类型。

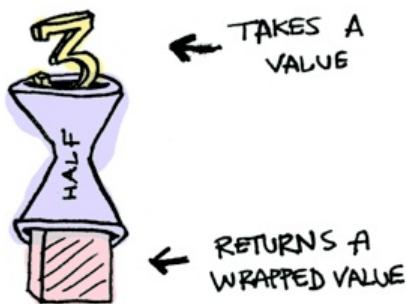
11.



这时，就需要再定义一种新的运算。它不是值与值的运算，也不是值与数据类型的运算，而是数据类型与数据类型的运算。

上图中，两个数据类型进行运算。首先，取出它们各自的值，一个是函数，一个是数值；然后，使用函数处理数值；最后，将函数的返回结果再封装进数据类型。

12.



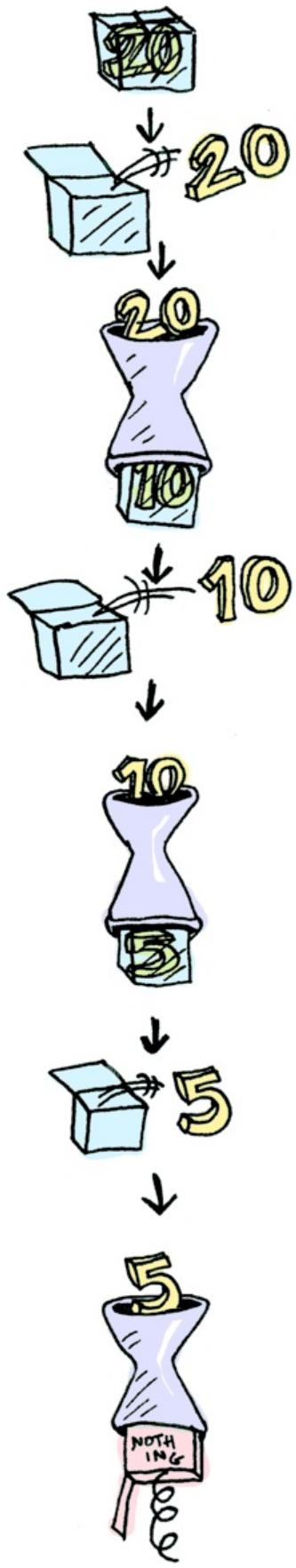
函数可以返回值，当然也可以返回数据类型。

13.



我们需要的是这样一种函数：它的输入和输出都是数据类型。

14.



这样做好处是什么？

因为数据类型是带有运算方法的，如果每一步返回的都是数据类型的实例，我们就可以把它们连接起来。

15.



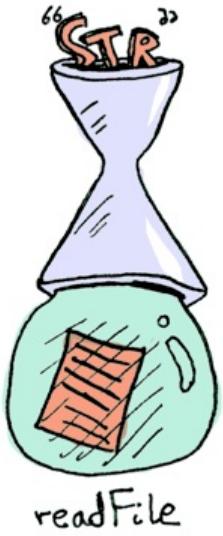
来看一个实例，系统的I/O提供了用户的输入。

16.



getLine函数可以将用户的输入处理成一个字符串类型（STR）的实例。

17.



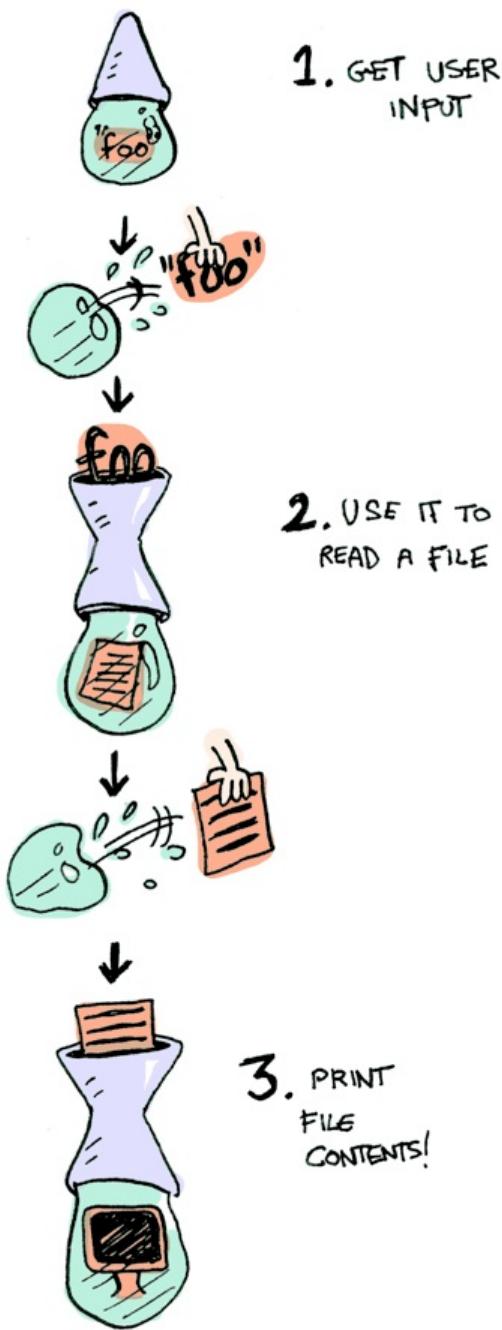
readfile函数接受STR实例当作文件名，返回一个文件类型的实例。

18.



putStrLn函数将文件内容输出。

19.



所有这些运算连起来，就叫做Monad。

简单说，Monad就是一种设计模式，表示将一个运算过程，通过函数拆解成互相连接的多个步骤。你只要提供下一步运算所需的函数，整个运算就会自动进行下去。

(完)

编译器的工作过程

源码要运行，必须先转成二进制的机器码。这是编译器的任务。

比如，下面这段源码（假定文件名叫做test.c）。

```
#include <stdio.h>

int main(void)
{
    fputs("Hello, world!\n", stdout);
    return 0;
}
```

要先用编译器处理一下，才能运行。

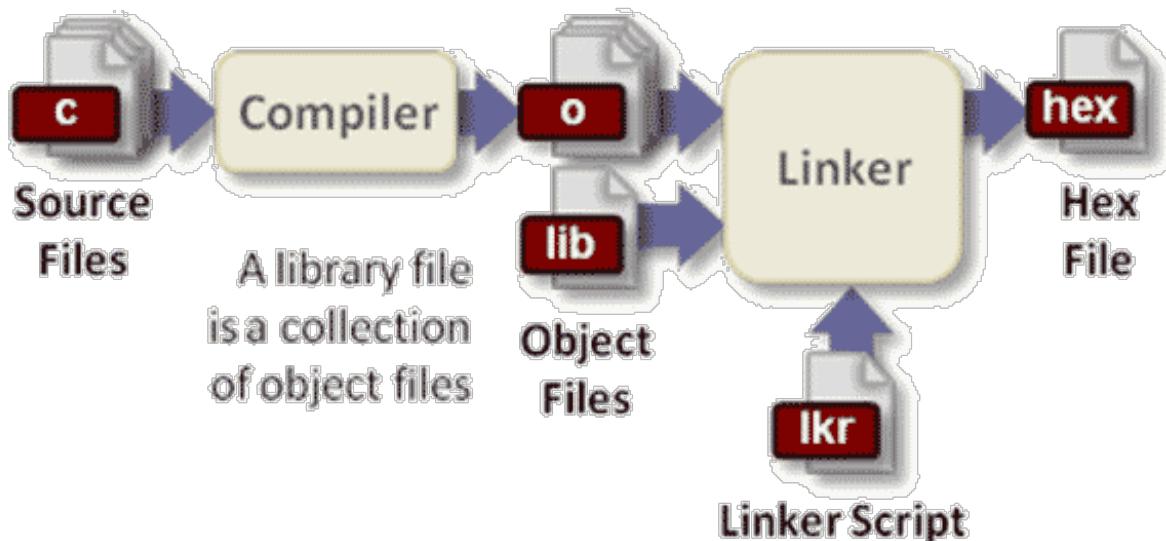
```
$ gcc test.c
$ ./a.out
Hello, world!
```

对于复杂的项目，编译过程还必须分成三步。

```
$ ./configure
$ make
$ make install
```

这些命令到底在干什么？大多数的书籍和资料，都语焉不详，只说这样就可以编译了，没有进一步的解释。

本文将介绍编译器的工作过程，也就是上面这三个命令各自的任务。我主要参考了Alex Smith的文章[《Building C Projects》](#)。需要声明的是，本文主要针对gcc编译器，也就是针对C和C++，不一定适用于其他语言的编译。



第一步 配置 (configure)

编译器在开始工作之前，需要知道当前的系统环境，比如标准库在哪里、软件的安装位置在哪里、需要安装哪些组件等等。这是因为不同计算机的系统环境不一样，通过指定编译参数，编译器就可以灵活适应环境，编译出各种环境都能运行的机器码。这个确定编译参数的步骤，就叫做“配置”(configure)。

这些配置信息保存在一个配置文件之中，约定俗成是一个叫做configure的脚本文件。通常它是由[autoconf](#)工具生成的。编译器通过运行这个脚本，获知编译参数。

configure脚本已经尽量考虑到不同系统的差异，并且对各种编译参数给出了默认值。如果用户的系统环境比较特别，或者有一些特定的需求，就需要手动向configure脚本提供编译参数。

```
$ ./configure --prefix=/www --with-mysql
```

上面代码是php源码的一种编译配置，用户指定安装后的文件保存在www目录，并且编译时加入mysql模块的支持。

第二步 确定标准库和头文件的位置

源码肯定会用到标准库函数 (standard library) 和头文件 (header)。它们可以存放在系统的任意目录中，编译器实际上没办法自动检测它们的位置，只有通过配置文件才能知道。

编译的第二步，就是从配置文件中知道标准库和头文件的位置。一般来说，配置文件会给出一个清单，列出几个具体的目录。等到编译时，编译器就按顺序到这几个目录中，寻找目标。

第三步 确定依赖关系

对于大型项目来说，源码文件之间往往存在依赖关系，编译器需要确定编译的先后顺序。假定A文件依赖于B文件，编译器应该保证做到下面两点。

- (1) 只有在B文件编译完成后，才开始编译A文件。
- (2) 当B文件发生变化时，A文件会被重新编译。

编译顺序保存在一个叫做makefile的文件中，里面列出哪个文件先编译，哪个文件后编译。而makefile文件由configure脚本运行生成，这就是为什么编译时configure必须首先运行的原因。

在确定依赖关系的同时，编译器也确定了，编译时会用到哪些头文件。

第四步 头文件的预编译 (precompilation)

不同的源码文件，可能引用同一个头文件（比如stdio.h）。编译的时候，头文件也必须一起编译。为了节省时间，编译器会在编译源码之前，先编译头文件。这保证了头文件只需编译一次，不必每次用到的时候，都重新编译了。

不过，并不是头文件的所有内容，都会被预编译。用来声明宏的#define命令，就不会被预编译。

第五步 预处理（Preprocessing）

预编译完成后，编译器就开始替换掉源码中bash的头文件和宏。以本文开头的那段源码为例，它包含头文件stdio.h，替换后的样子如下。

```
extern int fputs(const char *, FILE *);  
extern FILE *stdout;  
  
int main(void)  
{  
    fputs("Hello, world!\n", stdout);  
    return 0;  
}
```

为了便于阅读，上面代码只截取了头文件中与源码相关的那部分，即fputs和FILE的声明，省略了stdio.h的其他部分（因为它们非常长）。另外，上面代码的头文件没有经过预编译，而实际上，插入源码的是预编译后的结果。编译器在这一步还会移除注释。

这一步称为"预处理"（Preprocessing），因为完成之后，就要开始真正的处理了。

第六步 编译（Compilation）

预处理之后，编译器就开始生成机器码。对于某些编译器来说，还存在一个中间步骤，会先把源码转为汇编码（assembly），然后再把汇编码转为机器码。

下面是本文开头的那段源码转成的汇编码。

```
.file "test.c"  
.section .rodata  
.LC0:  
.string "Hello, world!\n"  
.text  
.globl main  
.type main, @function  
main:  
.LFB0:  
.cfi_startproc  
pushq %rbp  
.cfi_def_cfa_offset 16  
.cfi_offset 6, -16  
movq %rsp, %rbp  
.cfi_def_cfa_register 6  
movq stdout(%rip), %rax  
movq %rax, %rcx  
movl $14, %edx  
movl $1, %esi  
movl $.LC0, %edi  
call fwrite
```

```
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Debian 4.9.1-19) 4.9.1"
.section .note.GNU-stack,"",@progbits
```

这种转码后的文件称为对象文件（object file）。

第七步 连接（Linking）

对象文件还不能运行，必须进一步转成可执行文件。如果你仔细看上一步的转码结果，会发现其中引用了stdout函数和fwrite函数。也就是说，程序要正常运行，除了上面的代码以外，还必须有stdout和fwrite这两个函数的代码，它们是由C语言的标准库提供的。

编译器的下一步工作，就是把外部函数的代码（通常是后缀名为.lib和.a的文件），添加到可执行文件中。这就叫做连接（linking）。这种通过拷贝，将外部函数库添加到可执行文件的方式，叫做[静态连接](#)（static linking），后文会提到还有[动态连接](#)（dynamic linking）。

make命令的作用，就是从第四步头文件预编译开始，一直到做完这一步。

第八步 安装（Installation）

上一步的连接是在内存中进行的，即编译器在内存中生成了可执行文件。下一步，必须将可执行文件保存到用户事先指定的安装目录。

表面上，这一步很简单，就是将可执行文件（连带相关的数据文件）拷贝过去就行了。但是实际上，这一步还必须完成创建目录、保存文件、设置权限等步骤。这整个的保存过程就称为“安装”（Installation）。

第九步 操作系统连接

可执行文件安装后，必须以某种方式通知操作系统，让其知道可以使用这个程序了。比如，我们安装了一个文本阅读程序，往往希望双击txt文件，该程序就会自动运行。

这就要求在操作系统中，登记这个程序的元数据：文件名、文件描述、关联后缀名等等。Linux系统中，这些信息通常保存在/usr/share/applications目录下的.desktop文件中。另外，在Windows操作系统中，还需要在Start启动菜单中，建立一个快捷方式。

这些事情就叫做“操作系统连接”。make install命令，就用来完成“安装”和“操作系统连接”这两步。

第十步 生成安装包

写到这里，源码编译的整个过程就基本完成了。但是只有很少一部分用户，愿意耐着性子，从头到尾

做一遍这个过程。事实上，如果你只有源码可以交给用户，他们会认定你是一个不友好的家伙。大部分用户要的是一个二进制的可执行程序，立刻就能运行。这就要求开发者，将上一步生成的可执行文件，做成可以分发的安装包。

所以，编译器还必须有生成安装包的功能。通常是将可执行文件（连带相关的数据文件），以某种目录结构，保存成压缩文件包，交给用户。

第十一步 动态连接 (Dynamic linking)

正常情况下，到这一步，程序已经可以运行了。至于运行期间（runtime）发生的事情，与编译器一概无关。但是，开发者可以在编译阶段选择可执行文件连接外部函数库的方式，到底是静态连接（编译时连接），还是动态连接（运行时连接）。所以，最后还要提一下，什么叫做动态连接。

前面已经说过，静态连接就是把外部函数库，拷贝到可执行文件中。这样做的好处是，适用范围比较广，不用担心用户机器缺少某个库文件；缺点是安装包会比较大，而且多个应用程序之间，无法共享库文件。动态连接的做法正好相反，外部函数库不进入安装包，只在运行时动态引用。好处是安装包会比较小，多个应用程序可以共享库文件；缺点是用户必须事先安装好库文件，而且版本和安装位置都必须符合要求，否则就不能正常运行。

现实中，大部分软件采用动态连接，共享库文件。这种动态共享的库文件，Linux平台是后缀名为.so的文件，Windows平台是.dll文件，Mac平台是.dylib文件。

（文章完）

=====

以下为广告部分。欢迎大家在我的网络日志[投放广告](#)，推广自己的产品。今天介绍的是[100offer](#)。

[赞助商广告]

优秀的人才找到合适的归宿，是这个世界最幸福的事情之一。[100offer程序员拍卖网站](#)通过创新的拍卖方式，致力于帮助优秀程序员寻找归宿，给予求职者更多更好的职业选择。

• PHP、Android、Web前端、iOS 火热拍卖中 •

100offer程序员拍卖

提交申请，在两周以内获得3-5个offer

提交申请

过去三个月，100offer中成功的求职者，平均薪资涨幅高于30%，在2周内拿到3-5个offer。100offer与传统招聘网站存在极大差异，主要为下：

- 1、只接受部分候选人：100offer目前仅仅接受年薪高于15万，有一二线知名互联网公司工作经验的优秀程序员申请者。
- 2、反向模式：传统招聘网站是写简历投递给多家公司，而这里程序员只需要提交一次简历给offer，待审核通过后，100offer会邀约平台企业来竞拍候选人，产生一次投递数百家互联网公司的效果。拍卖时程序员会接受到来自各公司新鲜热辣的面试邀请，体验与传统网站截然不同。
- 3、绝对隐私：担心自己的隐私被雇主看到是完全不必要的：1、候选人同意面试邀请前，公司是完全看不到候选人的姓名、联系方式等隐私信息。2、拍卖开始前，候选人可以手动屏蔽掉3家公司，他们将永远看不到你的简历！

如何在1天内

应聘 **439** 家

互联网公司？

已经有众多大牛程序员通过100offer找到心仪的工作，目前11月候选人在征集中，点击图片[注册100offer](#)并提交完整简历的程序员朋友，即可获赠15元亚马逊礼品卡！（活动截止期为2014年12月30日）

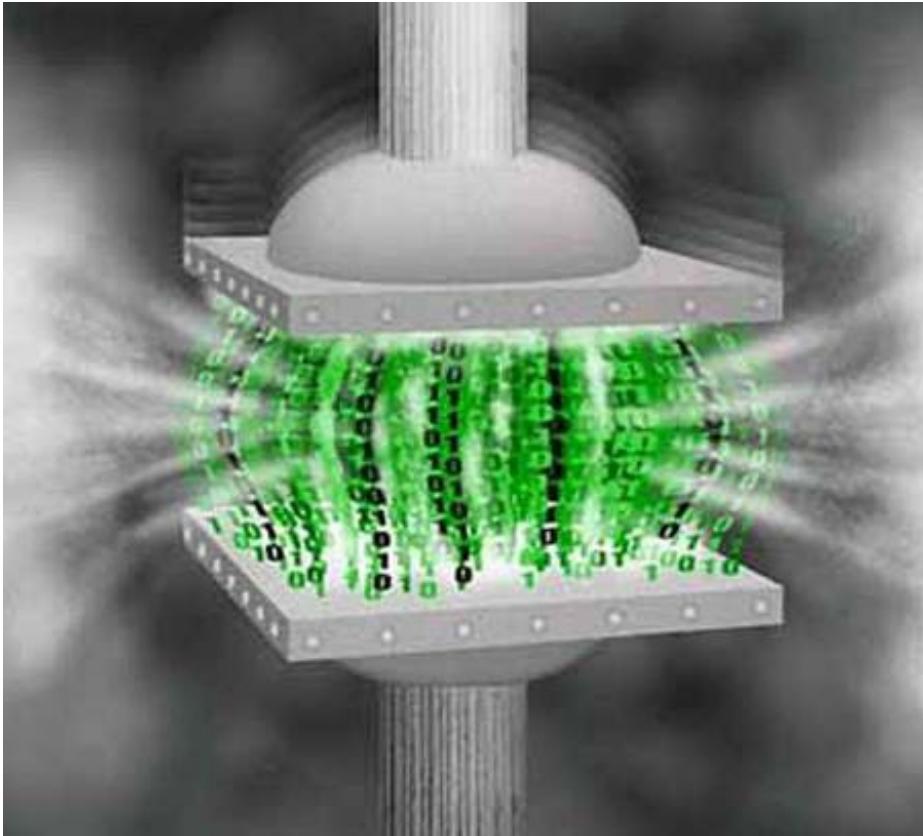
100offer目前阶段对企业免费，欢迎极客型创业公司和有实力的互联网公司前来[注册招聘](#)！

(完)

数据压缩与信息熵

1992年，美国佐治亚州的WEB Technology公司，宣布做出了重大的技术突破。

该公司的DataFiles/16软件，号称可以将任意大于64KB的文件，压缩为原始大小的16分之一。业界议论纷纷，如果消息属实，无异于压缩技术的革命。



许多专家还没有看到软件，就断言这是不可能的。因为根据压缩原理，你不可能将**任意文件**压缩到16分之一。事实上，有一些文件是无法压缩的，哪怕一个二进制位，都压缩不掉。

后来，事实果然如此，这款软件从来没有正式发布。没过几年，就连WEB Technology公司都消失了。

那么，为何不是所有的文件都可以被压缩？是否存在一个压缩极限呢，也就是说，到了一定大小，就没法再压缩了？

一、压缩的有限性

首先，回答第一个问题：为什么WEB Technology公司的发明不可能是真的。

反证法可以轻易地证明这一点。假定任何文件都可以压缩到n个二进制位（bit）以内，那么最多有 2^n 种不同的压缩结果。也就是说，如果有 2^n+1 个文件，必然至少有两个文件会产生同样的压缩结果。这意味着，这两个文件不可能无损地还原（解压缩）。因此，得到证明，并非所有文件都可以压缩到n个二进制位以下。

很自然地，下一个问题就是，这个n到底是多少？

二、压缩原理

要回答一个文件最小可以压缩到多少，必须要知道压缩的原理。

压缩原理其实很简单，就是找出那些重复出现的字符串，然后用更短的符号代替，从而达到缩短字符串的目的。比如，有一篇文章大量使用“中华人民共和国”这个词语，我们用“中国”代替，就缩短了5个字符，如果用“华”代替，就缩短了6个字符。事实上，只要保证对应关系，可以用任意字符代替那些重复出现的字符串。

本质上，所谓“压缩”就是找出文件内容的概率分布，将那些出现概率高的部分代替成更短的形式。所以，内容越是重复的文件，就可以压缩得越小。比如，“ABABABABABABAB”可以压缩成“7AB”。

相应地，如果内容毫无重复，就很难压缩。极端情况就是，遇到那些均匀分布的随机字符串，往往连一个字符都压缩不了。比如，任意排列的10个阿拉伯数字（5271839406），就是无法压缩的；再比如，无理数（比如π）也很难压缩。

压缩就是一个消除冗余的过程，相当于用一种更精简的形式，表达相同的内容。可以想象，压缩过一次以后，文件中的重复字符串将大幅减少。好的压缩算法，可以将冗余降到最低，以至于再也没有办法进一步压缩。所以，压缩已经压缩过的文件（递归压缩），通常是没有意义的。

三、压缩的极限

知道了压缩原理之后，就可以计算压缩的极限了。

上一节说过，压缩可以分解成两个步骤。第一步是得到文件内容的概率分布，哪些部分出现的次数多，哪些部分出现的次数少；第二步是对文件进行编码，用较短的符号替代那些重复出现的部分。

第一步的概率分布一般是确定的，现在就来考虑第二步，怎样找到最短的符号作为替代符。

如果文件内容只有两种情况（比如扔硬币的结果），那么只要一个二进制位就够了，1表示正面，0表示负面。如果文件内容包含三种情况（比如球赛的结果），那么最少需要两个二进制位。如果文件内容包含六种情况（比如扔筛子的结果），那么最少需要三个二进制位。

一般来说，在均匀分布的情况下，假定一个字符（或字符串）在文件中出现的概率是p，那么在这个位置上最多可能出现 $1/p$ 种情况。需要 $\log_2(1/p)$ 个二进制位表示替代符号。

这个结论可以推广到一般情况。假定文件有n个部分组成，每个部分的内容在文件中的出现概率分别为 p_1, p_2, \dots, p_n 。那么，替代符号占据的二进制最少为下面这个式子。

$$\log_2(1/p_1) + \log_2(1/p_2) + \dots + \log_2(1/p_n)$$

$$= \sum \log_2(1/p_i)$$

这可以被看作一个文件的压缩极限。

四、信息熵的公式

上一节的公式给出了文件压缩的极限。对于n相等的两个文件，概率p决定了这个式子的大小。p越大，表明文件内容越有规律，压缩后的体积就越小；p越小，表明文件内容越随机，压缩后的体积就越大。

为了便于文件之间的比较，将上式除以n，可以得到平均每个符号所占用的二进制位。

$$\begin{aligned} & \sum \log_2(1/p_n) / n \\ &= \log_2(1/p_1)/n + \log_2(1/p_2)/n + \dots + \log_2(1/p_n)/n \end{aligned}$$

由于p是根据频率统计得到的，因此上面的公式等价于下面的形式。

$$\begin{aligned} & p_1 * \log_2(1/p_1) + p_2 * \log_2(1/p_2) + \dots + p_n * \log_2(1/p_n) \\ &= \sum p_n * \log_2(1/p_n) \\ &= E(\log_2(1/p)) \end{aligned}$$

上面式子中最后的E，表示数学期望。可以理解成，每个符号所占用的二进制位，等于概率倒数的对数的数学期望。

下面是一个例子。假定有两个文件都包含1024个符号，在ASCII码的情况下，它们的长度是相等的，都是1KB。甲文件的内容50%是a，30%b，20%c，则平均每个符号要占用1.49个二进制位。

$$\begin{aligned} & 0.5 * \log_2(1/0.5) + 0.3 * \log_2(1/0.3) + 0.2 * \log_2(1/0.2) \\ &= 1.49 \end{aligned}$$

既然每个符号要占用1.49个二进制位，那么压缩1024个符号，理论上最少需要1526个二进制位，约0.186KB，相当于压缩掉了81%的体积。

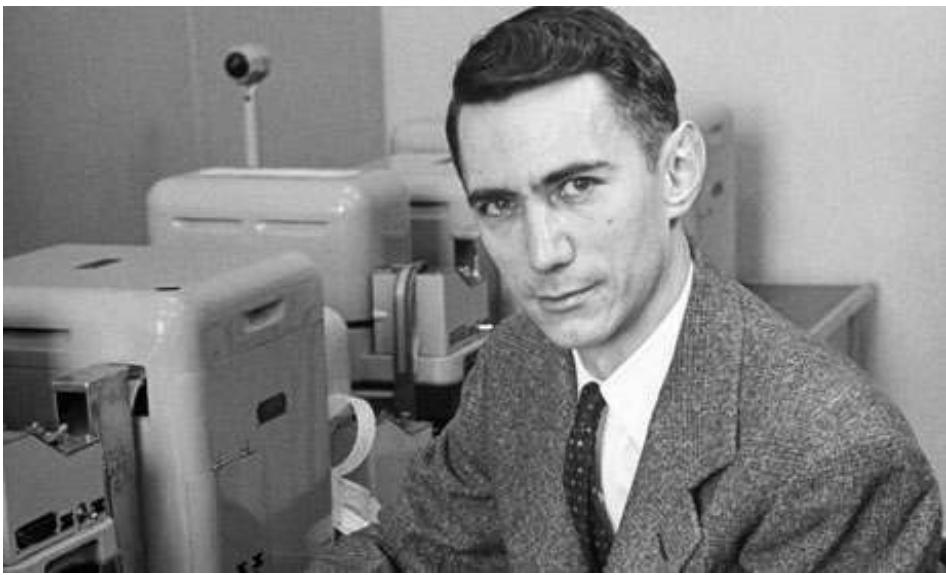
乙文件的内容10%是a，10%是b，……，10%是j，则平均每个符号要占用3.32个二进制位。

$$\begin{aligned} & 0.1 * \log_2(1/0.1) * 10 \\ &= 3.32 \end{aligned}$$

既然每个符号要占用3.32个二进制位，那么压缩1024个符号，理论上最少需要3400个二进制位，约0.415KB，相当于压缩掉了58%的体积。

对比上面两个算式，可以看到文件内容越是分散（随机），所需要的二进制位就越长。所以，这个值可以用来衡量文件内容的随机性（又称不确定性）。这就叫做信息熵（information entropy）。

它是1948年由美国数学家克劳德·香农（Claude Shannon）在经典论文《通信的数学理论》中，首先提出的。



五、信息熵的含义

想要理解信息熵这个概念，有几点需要注意。

(1) **信息熵只反映内容的随机性，与内容本身无关。**不管是什么样内容的文件，只要服从同样的概率分布，就会计算得到同样的信息熵。

(2) **信息熵越大，表示占用的二进制位越长，因此就可以表达更多的符号。**所以，人们有时也说，信息熵越大，表示信息量越大。不过，由于第一点的原因，这种说法很容易产生误导。较大的信息熵，只表示可能出现的符号较多，并不意味着你可以从中得到更多的信息。

(3) **信息熵与热力学的熵，基本无关。**这两个熵不是同一件事，信息熵表示无序的信息，热力学的熵表示无序的能量（参见我写的[《熵的社会学意义》](#)）。

(文章完)

=====

以下为广告部分。欢迎大家在我的网络日志[投放广告](#)，推广自己的产品。今天介绍的是[生命链记忆网](#)。

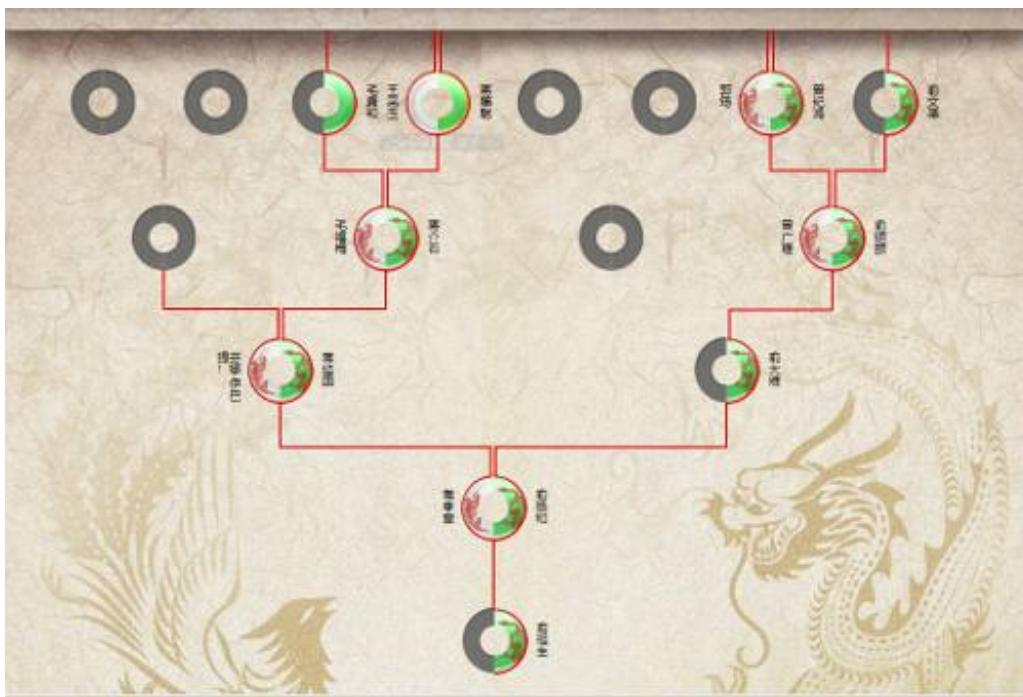
[赞助商广告]

别人的终究是别人的，今天我们书写自己的传奇！个人史是汇入正史河流的涓涓细流。

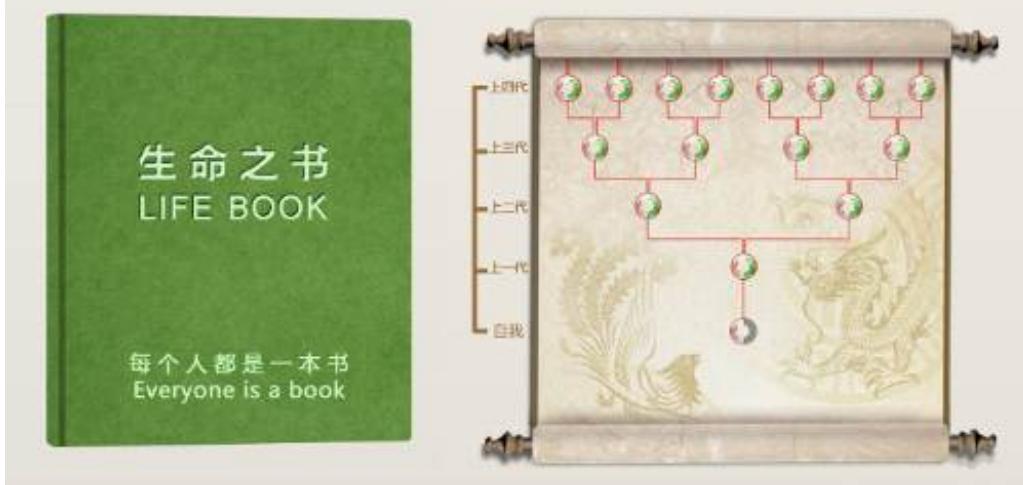
[生命链记忆网](#)是永远在线的个人史馆，是永恒的信息载体，永久保留每个人的人生记忆和生命轨迹直至千秋万代。

花一分钟成为[注册用户](#)，就可以使用两大基本功能：**生命链**和**生命之书**。

(1) **生命链**就是你的根，你可以邀请祖先(父系和母系)及子女合链，让自己的生命链有形化、可视化；所有血亲和姻亲成员之间可互动。



(2) **生命之书**就是书写你的人生故事，包括日记、自传、随笔等等；自传可选择公开，在自传馆里能被所有人看到。



生命链功能永久免费！

生命之书功能每年付费10元，可使用100M空间（免费试用6个月）；累计付费1000元，生命之书永久开启。

永恒之旅，从此开始，[点击试用！](#)

(完)

数据库的最简单实现

所有应用软件之中，数据库可能是最复杂的。

MySQL的手册有3000多页，PostgreSQL的手册有2000多页，Oracle的手册更是比它们相加还要厚。



但是，自己写一个最简单的数据库，做起来并不难。Reddit上面有一个[帖子](#)，只用了几百个字，就把原理讲清楚了。下面是我根据这个帖子整理的内容。

一、数据以文本形式保存

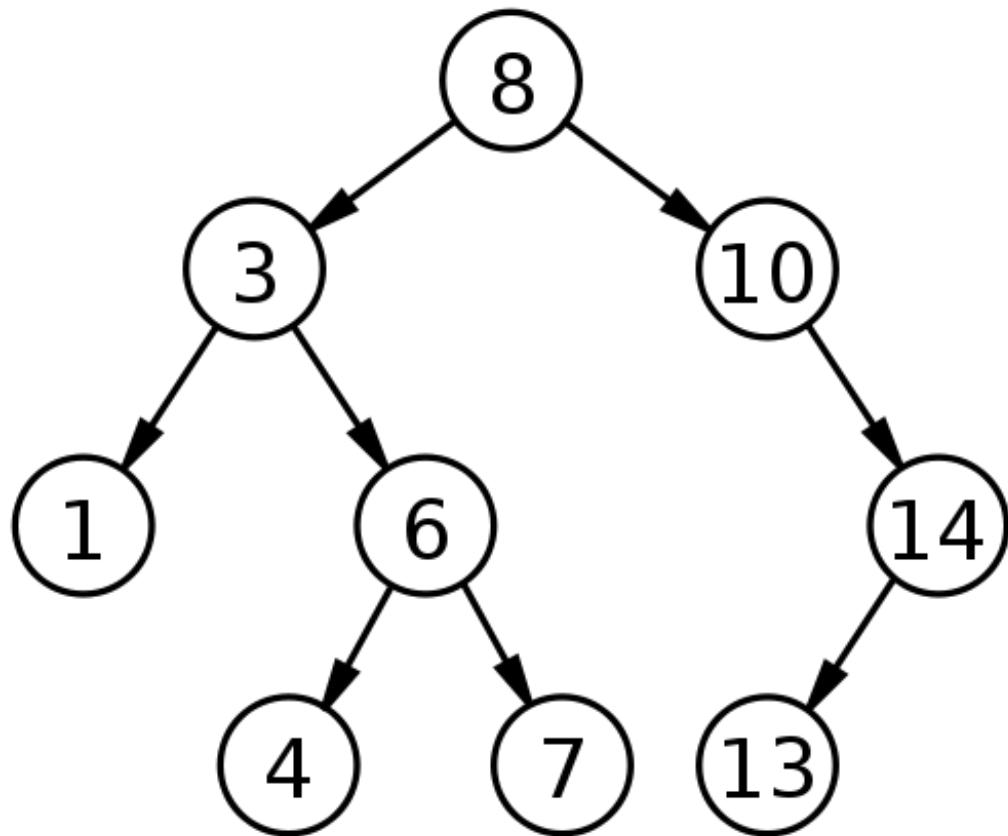
第一步，就是将所要保存的数据，写入文本文件。这个文本文件就是你的数据库。

为了方便读取，数据必须分成记录，每一条记录的长度规定为等长。比如，假定每条记录的长度是800字节，那么第5条记录的开始位置就在3200字节。

大多数时候，我们不知道某一条记录在第几个位置，只知道[主键](#)（primary key）的值。这时为了读取数据，可以一条条比对记录。但是这样做效率太低，实际应用中，数据库往往采用[B树](#)（B-tree）格式储存数据。

二、什么是B树？

要理解B树，必须从[二叉查找树](#)（Binary search tree）讲起。

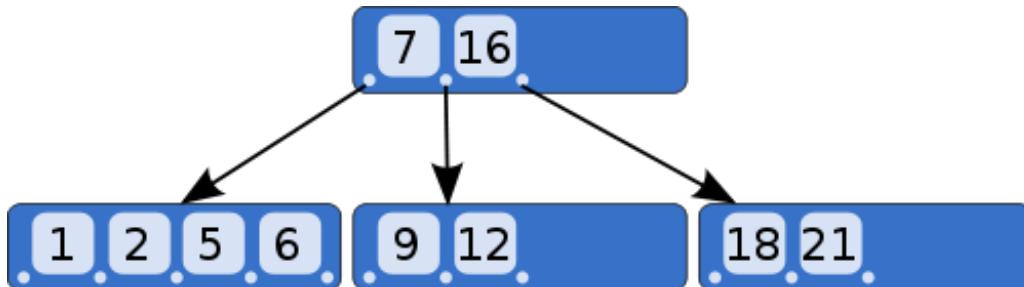


二叉查找树是一种查找效率非常高的数据结构，它有三个特点。

- (1) 每个节点最多只有两个子树。
- (2) 左子树都为小于父节点的值，右子树都为大于父节点的值。
- (3) 在n个节点中找到目标值，一般只需要 $\log(n)$ 次比较。

二叉查找树的结构不适合数据库，因为它的查找效率与层数相关。越处在下层的数据，就需要越多次比较。极端情况下，n个数据需要n次比较才能找到目标值。对于数据库来说，每进入一层，就要从硬盘读取一次数据，这非常致命，因为硬盘的读取时间远远大于数据处理时间，数据库读取硬盘的次数越少越好。

B树是对二叉查找树的改进。它的设计思想是，将相关数据尽量集中在一起，以便一次读取多个数据，减少硬盘操作次数。



B树的特点也有三个。

- (1) 一个节点可以容纳多个值。比如上图中，最多的一个节点容纳了4个值。

(2) 除非数据已经填满，否则不会增加新的层。也就是说，B树追求"层"越少越好。

(3) 子节点中的值，与父节点中的值，有严格的大小对应关系。一般来说，如果父节点有 a 个值，那么就有 $a+1$ 个子节点。比如上图中，父节点有两个值（7和16），就对应三个子节点，第一个子节点都是小于7的值，最后一个子节点都是大于16的值，中间的子节点就是7和16之间的值。

这种数据结构，非常有利于减少读取硬盘的次数。假定一个节点可以容纳100个值，那么3层的B树可以容纳100万个数据，如果换成二叉查找树，则需要20层！假定操作系统一次读取一个节点，并且根节点保留在内存中，那么B树在100万个数据中查找目标值，只需要读取两次硬盘。

三、索引

数据库以B树格式储存，只解决了按照"主键"查找数据的问题。如果想查找其他字段，就需要建立索引（index）。

所谓索引，就是以某个字段为关键字的B树文件。假定有一张"雇员表"，包含了员工号（主键）和姓名两个字段。可以对姓名建立索引文件，该文件以B树格式对姓名进行储存，每个姓名后面是其在数据库中的位置（即第几条记录）。查找姓名的时候，先从索引中找到对应第几条记录，然后再从表格中读取。

这种索引查找方法，叫做"[索引顺序存取方法](#)"（Indexed Sequential Access Method），缩写为ISAM。它已经有多种实现（比如C-ISAM库和D-ISAM库），只要使用这些代码库，就能自己写一个最简单的数据库。

四、高级功能

部署了最基本的数据存取（包括索引）以后，还可以实现一些高级功能。

(1) SQL语言是数据库通用操作语言，所以需要一个SQL解析器，将SQL命令解析为对应的ISAM操作。

(2) 数据库连接（join）是指数据库的两张表通过"外键"，建立连接关系。你需要对这种操作进行优化。

(3) 数据库事务（transaction）是指批量进行一系列数据库操作，只要有一步不成功，整个操作都不成功。所以需要有一个"操作日志"，以便失败时对操作进行回滚。

(4) 备份机制：保存数据库的副本。

(5) 远程操作：使得用户可以在不同的机器上，通过TCP/IP协议操作数据库。

（完）

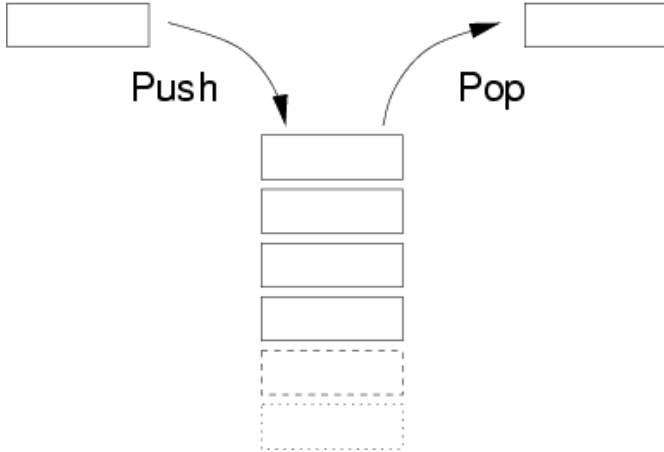
Stack的三种含义

学习编程的时候，经常会看到stack这个词，它的中文名字叫做“栈”。

理解这个概念，对于理解程序的运行至关重要。容易混淆的是，这个词其实有三种含义，适用于不同的场合，必须加以区分。

含义一：数据结构

stack的第一种含义是一组数据的[存放方式](#)，特点为LIFO，即后进先出（Last in, first out）。



在这种数据结构中，数据像积木那样一层层堆起来，后面加入的数据就放在最上层。使用的时候，最上层的数据第一个被用掉，这就叫做“后进先出”。

与这种结构配套的，是一些特定的方法，主要为下面这些。

- push：在最顶层加入数据。
- pop：返回并移除最顶层的数据。
- top：返回最顶层数据的值，但不移除它。
- isempty：返回一个布尔值，表示当前stack是否为空栈。

含义二：代码运行方式

stack的第二种含义是[“调用栈”](#)（call stack），表示函数或子例程像堆积木一样存放，以实现层层调用。

下面以一段Java代码为例（[来源](#)）。

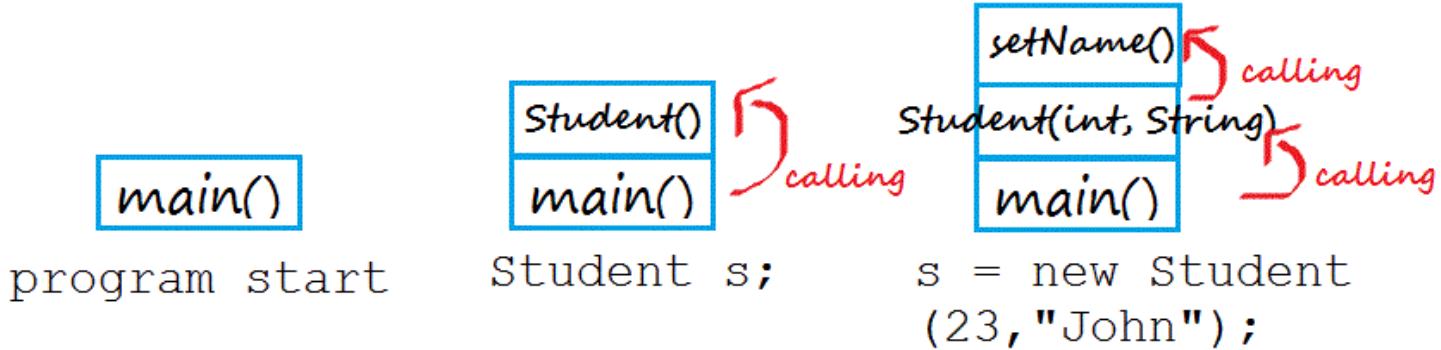
```
class Student{
    int age;
    String name;

    public Student(int Age, String Name)
    {
        this.age = Age;
        setName(Name);
    }
    public void setName(String Name)
    {
        this.name = Name;
    }
}

public class Main{
    public static void main(String[] args) {
        Student s;
```

```
s = new Student(23,"John");
}
}
```

上面这段代码运行的时候，首先调用main方法，里面需要生成一个Student的实例，于是又调用Student构造函数。在构造函数中，又调用到setName方法。



这三次调用像积木一样堆起来，就叫做“调用栈”。程序运行的时候，总是先完成最上层的调用，然后将它的值返回到下一层调用，直至完成整个调用栈，返回最后的结果。

含义三：内存区域

stack的第三种含义是存放数据的一种内存区域。程序运行的时候，需要内存空间存放数据。一般来说，系统会划分出两种不同的内存空间：一种叫做stack（栈），另一种叫做heap（堆）。



它们的主要区别是：stack是有结构的，每个区块按照一定次序存放，可以明确知道每个区块的大小；heap是没有结构的，数据可以任意存放。因此，stack的寻址速度要快于heap。



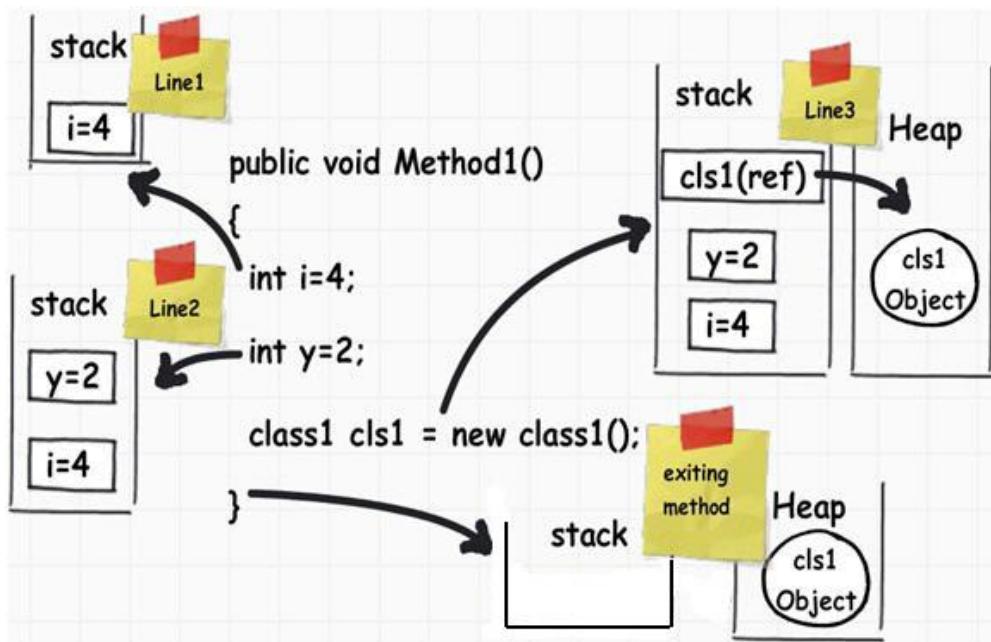
其他的区别还有，一般来说，每个线程分配一个stack，每个进程分配一个heap，也就是说，stack是线程独占的，heap是线程共用的。此外，stack创建的时候，大小是确定的，数据超过这个大小，就发生stack overflow错误，而heap的大小是不确定的，需要的话可以不断增加。

根据上面这些区别，数据存放的规则是：只要是局部的、占用空间确定的数据，一般都存放在stack里面，否则就放在heap里面。请看下面这段代码（[来源](#)）。

```
public void Method1()
{
    int i=4;
    int y=2;
    class1 cls1 = new class1();
}
```

上面代码的Method1方法，共包含了三个变量：i, y 和 cls1。其中，i和y的值是整数，内存占用空间是确定的，而且是局部变量，只用在Method1区块之内，不会用于区块之外。cls1也是局部变量，但是类型为指针变量，指向一个对象的实例。指针变量占用的大小是确定的，但是对象实例以目前的信息无法确知所占用的内存空间大小。

这三个变量和一个对象实例在内存中的存放方式如下。



从上图可以看到，i、y和cls1都存放在stack，因为它们占用内存空间都是确定的，而且本身也属于局部变量。但是，cls1指向的对象实例存放在heap，因为它的大小不确定。作为一条规则可以记住，所有的对象都存放在heap。

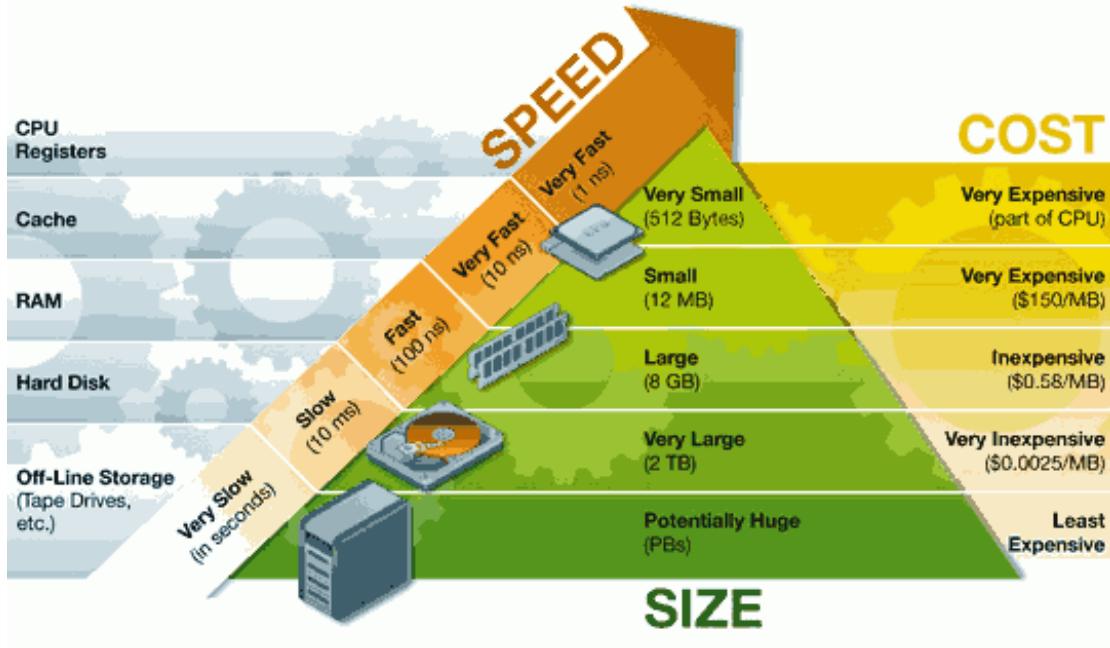
接下来的问题是，当Method1方法运行结束，会发生什么事？

回答是整个stack被清空，i、y和cls1这三个变量消失，因为它们是局部变量，区块一旦运行结束，就没必要再存在了。而heap之中的那个对象实例继续存在，直到系统的垃圾清理机制（garbage collector）将这块内存回收。因此，一般来说，内存泄漏都发生在heap，即某些内存空间不再被使用了，却因为种种原因，没有被系统回收。

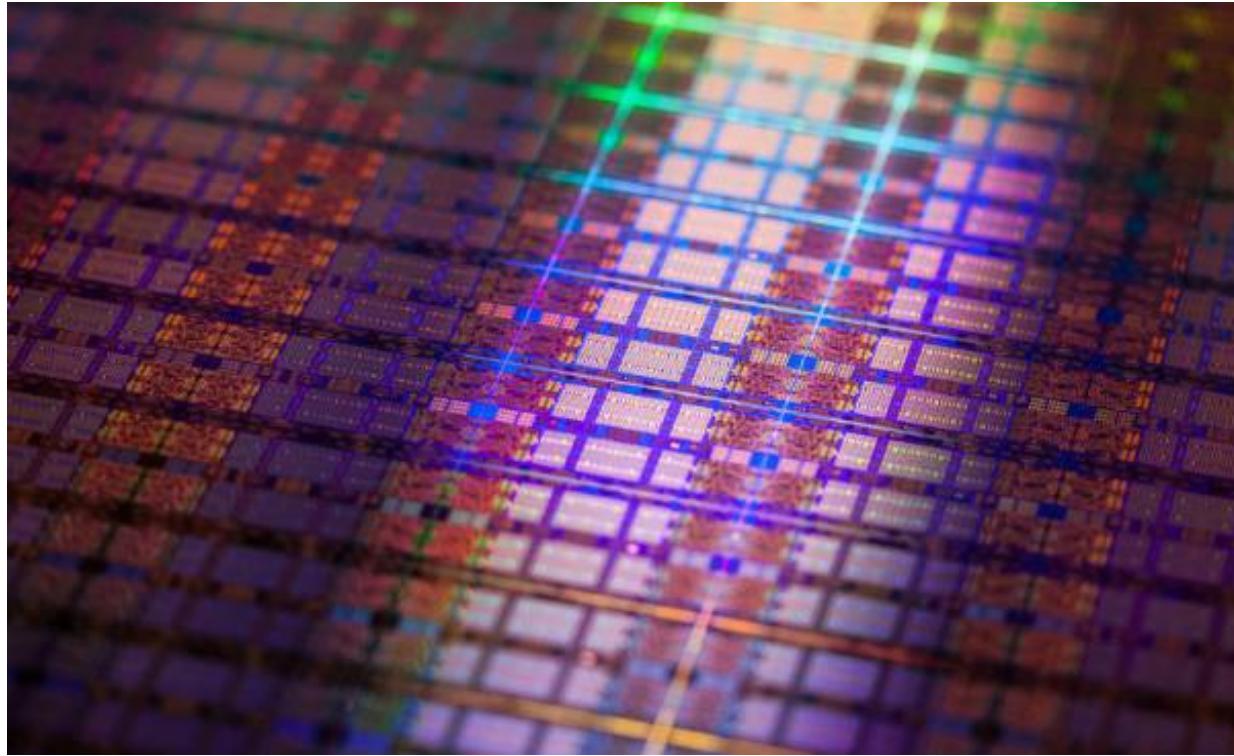
(完)

为什么寄存器比内存快？

计算机的存储层次（memory hierarchy）之中，寄存器（register）最快，内存其次，最慢的是硬盘。



同样都是晶体管存储设备，为什么寄存器比内存快呢？



[Mike Ash](#)写了一篇很好的解释，非常通俗地回答了这个问题，有助于加深对硬件的理解。下面就是我的简单翻译。

原因一：距离不同

距离不是主要因素，但是最好懂，所以放在最前面说。内存离CPU比较远，所以要耗费更长的时间读取。

以3GHz的CPU为例，电流每秒钟可以振荡30亿次，每次耗时大约为0.33纳秒。光在1纳秒的时间内，可以前进30厘米。也就是说，在CPU的一个时钟周期内，光可以前进10厘米。因此，如果内存距离CPU超过5厘米，就不可能在一个时钟周期内完成数据的读取，这还没有考虑硬件的限制和电流实际上达不到光速。相比之下，寄存器在CPU内部，当然读起来会快一点。

距离对于桌面电脑影响很大，对于手机影响就要小得多。手机CPU的时钟频率比较慢（iPhone 5s为1.3GHz），而且手机的内存紧挨着CPU。

原因二：硬件设计不同

苹果公司新推出的iPhone 5s，CPU是A7，寄存器有6000多位（31个64位寄存器，加上32个128位寄存器）。而iPhone 5s的内存是1GB，约为80亿位（bit）。这意味着，高性能、高成本、高耗电的设计可以用在寄存器上，反正只有6000多位，而不能用在内存上。因为每个位的成本和能耗只要增加一点点，就会被放大80亿倍。



事实上确实如此，内存的设计相对简单，每个位就是一个电容和一个晶体管，而寄存器的[设计](#)则完全不同，多出好几个电子元件。并且通电以后，寄存器的晶体管一直有电，而内存的晶体管只有用到的才有电，没用到的就没电，这样有利于省电。这些设计上的因素，决定了寄存器比内存读取速度更快。

原因三：工作方式不同

寄存器的工作方式很简单，只有两步：（1）找到相关的位，（2）读取这些位。

内存的工作方式就要复杂得多：

- (1) 找到数据的指针。（指针可能存放在寄存器内，所以这一步就已经包括寄存器的全部工作了。）
- (2) 将指针送往[内存管理单元](#)（MMU），由MMU将虚拟的内存地址翻译成实际的物理地址。
- (3) 将物理地址送往内存控制器（[memory controller](#)），由内存控制器找出该地址在哪

一根内存插槽（bank）上。

(4) 确定数据在哪一个内存块（chunk）上，从该块读取数据。

(5) 数据先送回内存控制器，再送回CPU，然后开始使用。

内存的工作流程比寄存器多出许多步。每一步都会产生延迟，累积起来就使得内存比寄存器慢得多。

为了缓解寄存器与内存之间的巨大速度差异，硬件设计师做出了许多努力，包括在CPU内部设置缓存、优化CPU工作方式，尽量一次性从内存读取指令所要用到的全部数据等等。

(完)

Linux 的启动流程

半年前，我写了[《计算机是如何启动的？》](#)，探讨BIOS和主引导记录的作用。

那篇文章不涉及操作系统，只与主板的板载程序有关。今天，我想接着往下写，探讨操作系统接管硬件以后发生的事情，也就是操作系统的启动流程。



这个部分比较有意思。因为在BIOS阶段，计算机的行为基本上被写死了，程序员可以做的事情并不多；但是，一旦进入操作系统，程序员几乎可以定制所有方面。所以，这个部分与程序员的关系更密切。

我主要关心的是Linux操作系统，它是目前服务器端的主流操作系统。下面的内容针对的是[Debian](#)发行版，因为我对其他发行版不够熟悉。

第一步、加载内核

操作系统接管硬件以后，首先读入 /boot 目录下的内核文件。



以我的电脑为例，/boot 目录下面大概是这样一些文件：

```
$ ls /boot

config-3.2.0-3-amd64
config-3.2.0-4-amd64
grub
initrd.img-3.2.0-3-amd64
initrd.img-3.2.0-4-amd64
System.map-3.2.0-3-amd64
System.map-3.2.0-4-amd64
vmlinuz-3.2.0-3-amd64
vmlinuz-3.2.0-4-amd64
```

第二步、启动初始化进程

内核文件加载以后，就开始运行第一个程序 `/sbin/init`，它的作用是初始化系统环境。



由于init是第一个运行的程序，它的进程编号（pid）就是1。其他所有进程都从它衍生，都是它的子进程。

第三步、确定运行级别

许多程序需要开机启动。它们在Windows叫做"服务"（service），在Linux就叫做"[守护进程](#)"（daemon）。

init进程的一大任务，就是去运行这些开机启动的程序。但是，不同的场合需要启动不同的程序，比如用作服务器时，需要启动Apache，用作桌面就不需要。Linux允许为不同的场合，分配不同的开机启动程序，这就叫做"[运行级别](#)"（runlevel）。也就是说，启动时根据"运行级别"，确定要运行哪些程序。



Linux预置七种运行级别（0-6）。一般来说，0是关机，1是单用户模式（也就是维护模式），6是重启。运行级别2-5，各个发行版不太一样，对于Debian来说，都是同样的多用户模式（也就是正常模式）。

init进程首先读取文件 `/etc/inittab`，它是运行级别的设置文件。如果你打开它，可以看到第一行是这样的：

```
id:2:initdefault:
```

`initdefault`的值是2，表明系统启动时的运行级别为2。如果需要指定其他级别，可以手动修改这个值。

那么，运行级别2有些什么程序呢，系统怎么知道每个级别应该加载哪些程序呢？……回答是每个运行级别在`/etc`目录下面，都有一个对应的子目录，指定要加载的程序。

```
/etc/rc0.d  
/etc/rc1.d  
/etc/rc2.d  
/etc/rc3.d  
/etc/rc4.d  
/etc/rc5.d  
/etc/rc6.d
```

上面目录名中的"rc"，表示run command（运行程序），最后的d表示directory（目录）。下面让我们看看 /etc/rc2.d 目录中到底指定了哪些程序。

```
$ ls /etc/rc2.d
```

```
README  
S01motd  
S13rpcbind  
S14nfs-common  
S16binfmt-support  
S16rsyslog  
S16sudo  
S17apache2  
S18acpid  
...
```

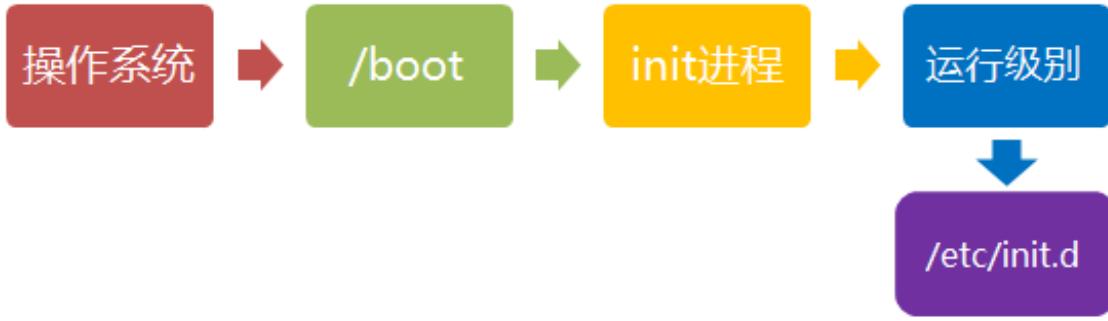
可以看到，除了第一个文件README以外，其他文件名都是"字母S+两位数字+程序名"的形式。字母S表示Start，也就是启动的意思（启动脚本的运行参数为start），如果这个位置是字母K，就代表Kill（关闭），即如果从其他运行级别切换过来，需要关闭的程序（启动脚本的运行参数为stop）。后面的两位数字表示处理顺序，数字越小越早处理，所以第一个启动的程序是motd，然后是rpcbind、nfs.....数字相同时，则按照程序名的字母顺序启动，所以rsyslog会先于sudo启动。

这个目录里的所有文件（除了README），就是启动时要加载的程序。如果想增加或删除某些程序，不建议手动修改 /etc/rcN.d 目录，最好是用一些专门命令进行管理（参考[这里](#)和[这里](#)）。

第四步、加载开机启动程序

前面提到，七种预设的"运行级别"各自有一个目录，存放需要开机启动的程序。不难想到，如果多个"运行级别"需要启动同一个程序，那么这个程序的启动脚本，就会在每一个目录里都有一个拷贝。这样会造成管理上的困扰：如果要修改启动脚本，岂不是每个目录都要改一遍？

Linux的解决办法，就是七个 /etc/rcN.d 目录里列出的程序，都设为链接文件，指向另外一个目录 /etc/init.d，真正的启动脚本都统一放在这个目录中。init进程逐一加载开机启动程序，其实就是运行这个目录里的启动脚本。



下面就是链接文件真正的指向。

```
$ ls -l /etc/rc2.d
README
S01motd -> ../init.d/motd
S13rpcbind -> ../init.d/rpcbind
S14nfs-common -> ../init.d/nfs-common
S16binfmt-support -> ../init.d/binfmt-support
S16rsyslog -> ../init.d/rsyslog
S16sudo -> ../init.d/sudo
S17apache2 -> ../init.d/apache2
S18acpid -> ../init.d/acpid
...
...
```

这样做的另一个好处，就是如果你要手动关闭或重启某个进程，直接到目录 /etc/init.d 中寻找启动脚本即可。比如，我要重启Apache服务器，就运行下面的命令：

```
$ sudo /etc/init.d/apache2 restart
```

/etc/init.d 这个目录名最后一个字母d，是directory的意思，表示这是一个目录，用来与程序 /etc/init 区分。

第五步、用户登录

开机启动程序加载完毕以后，就要让用户登录了。



一般来说，用户的登录方式有三种：

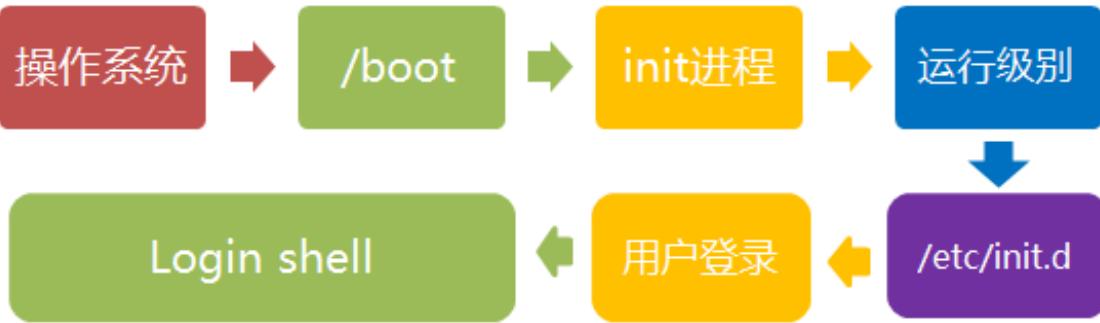
- (1) 命令行登录
- (2) ssh登录
- (3) 图形界面登录

这三种情况，都有自己的方式对用户进行认证。

- (1) 命令行登录：init进程调用getty程序（意为get teletype），让用户输入用户名和密码。输入完成后，再调用login程序，核对密码（Debian还会再多运行一个身份核对程序/etc/pam.d/login）。如果密码正确，就从文件/etc/passwd读取该用户指定的shell，然后启动这个shell。
- (2) ssh登录：这时系统调用sshd程序（Debian还会再运行/etc/pam.d/ssh），取代getty和login，然后启动shell。
- (3) 图形界面登录：init进程调用显示管理器，Gnome图形界面对应的显示管理器为gdm（GNOME Display Manager），然后用户输入用户名和密码。如果密码正确，就读取/etc/gdm3/Xsession，启动用户的会话。

第六步、进入 login shell

所谓shell，简单说就是命令行界面，让用户可以直接与操作系统对话。用户登录时打开的shell，就叫做login shell。



Debian默认的shell是[Bash](#)，它会读入一系列的配置文件。上一步的三种情况，在这一步的处理，也存在差异。

(1) 命令行登录：首先读入 `/etc/profile`，这是对所有用户都有效的配置；然后依次寻找下面三个文件，这是针对当前用户的配置。

```

~/.bash_profile
~/.bash_login
~/.profile

```

需要注意的是，这三个文件只要有一个存在，就不再读入后面的文件了。比如，要是 `~/.bash_profile` 存在，就不会再读入后面两个文件了。

(2) ssh登录：与第一种情况完全相同。

(3) 图形界面登录：只加载 `/etc/profile` 和 `~/.profile`。也就是说，`~/.bash_profile` 不管有没有，都不会运行。

第七步，打开 non-login shell

老实说，上一步完成以后，Linux的启动过程就算结束了，用户已经可以看到命令行提示符或者图形界面了。但是，为了内容的完整，必须再介绍一下这一步。

用户进入操作系统以后，常常会再手动开启一个shell。这个shell就叫做 non-login shell，意思是它不同于登录时出现的那个shell，不读取`/etc/profile`和`.profile`等配置文件。



non-login shell的重要性，不仅在于它是用户最常接触的那个shell，还在于它会读入用户自己的bash配置文件`~/.bashrc`。大多数时候，我们对于bash的定制，都是写在这个文件里面的。

你也许会问，要是不进入 non-login shell，岂不是`.bashrc`就不会运行了，因此bash也就不能完成定制了？事实上，Debian已经考虑到这个问题了，请打开文件`~/.profile`，可以看到下面的代码：

```

if [ -n "$BASH_VERSION" ]; then
    if [ -f "$HOME/.bashrc" ]; then
        . "$HOME/.bashrc"
    fi
fi
  
```

上面代码先判断变量`$BASH_VERSION`是否有值，然后判断主目录下是否存在`.bashrc`文件，如果存在就运行该文件。第三行开头的那个点，是`source`命令的简写形式，表示运行某个文件，写成`"source ~/.bashrc"`也是可以的。

因此，只要运行`~/.profile`文件，`~/.bashrc`文件就会连带运行。但是上一节的第一种情况提到过，如果存在`~/.bash_profile`文件，那么有可能不会运行`~/.profile`文件。解决这个问题很简单，把下面代码写入`.bash_profile`就行了。

```

if [ -f ~/.profile ]; then
    . ~/.profile
fi
  
```

这样一来，不管是哪种情况，`.bashrc`都会执行，用户的设置可以放心地都写入这个文件了。

Bash的设置之所以如此繁琐，是由于历史原因造成的。早期的时候，计算机运行速度很慢，载入配置文件需要很长时间，Bash的作者只好把配置文件分成了几个部分，阶段性载入。系统的通用设置放在`/etc/profile`，用户个人的、需要被所有子进程继承的设置放在`.profile`，不需要被继承的设置放

在.bashrc。

顺便提一下，除了Linux以外， Mac OS X 使用的shell也是Bash。但是，它只加载.bash_profile，然后在.bash_profile里面调用.bashrc。而且，不管是ssh登录，还是在图形界面里启动shell窗口，都是如此。

参考链接

- [1] Debian Wiki, [Environment Variables](#)
- [2] Debian Wiki, [Dot Files](#)
- [3] Debian Administration, [An introduction to run-levels](#)
- [4] Debian Admin, [Debian and Ubuntu Linux Run Levels](#)
- [5] Linux Information Project (LINFO), [Runlevel Definition](#)
- [6] LinuxQuestions.org, [What are run levels?](#)
- [7] Dalton Hubble, [Bash Configurations Demystified](#)

(完)

RSA算法原理（二）

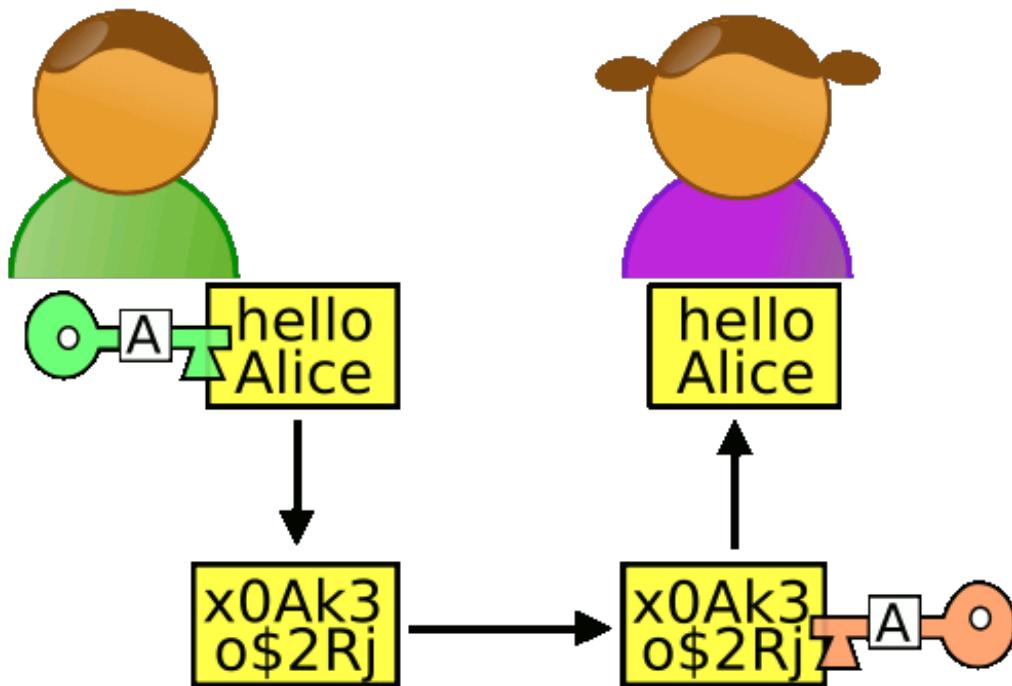
上一次，我介绍了一些[数论知识](#)。

有了这些知识，我们就可以看懂[RSA算法](#)。这是目前地球上最重要的加密算法。



六、密钥生成的步骤

我们通过一个例子，来理解RSA算法。假设[爱丽丝](#)要与鲍勃进行加密通信，她该怎么生成公钥和私钥呢？



第一步，随机选择两个不相等的质数p和q。

爱丽丝选择了61和53。（实际应用中，这两个质数越大，就越难破解。）

第二步，计算p和q的乘积n。

爱丽丝就把61和53相乘。

$$n = 61 \times 53 = 3233$$

n的长度就是密钥长度。3233写成二进制是110010100001，一共有12位，所以这个密钥就是12位。实际应用中，RSA密钥一般是1024位，重要场合则为2048位。

第三步，计算n的欧拉函数 $\phi(n)$ 。

根据公式：

$$\phi(n) = (p-1)(q-1)$$

爱丽丝算出 $\phi(3233)$ 等于 60×52 ，即3120。

第四步，随机选择一个整数e，条件是 $1 < e < \phi(n)$ ，且e与 $\phi(n)$ 互质。

爱丽丝就在1到3120之间，随机选择了17。（实际应用中，常常选择65537。）

第五步，计算e对于 $\phi(n)$ 的模反元素d。

所谓“模反元素”就是指有一个整数d，可以使得 ed 被 $\phi(n)$ 除的余数为1。

$$ed \equiv 1 \pmod{\phi(n)}$$

这个式子等价于

$$ed - 1 = k\phi(n)$$

于是，找到模反元素d，实质上就是对下面这个二元一次方程求解。

$$ex + \phi(n)y = 1$$

已知 $e=17$, $\phi(n)=3120$,

$$17x + 3120y = 1$$

这个方程可以用“扩展欧几里得算法”求解，此处省略具体过程。总之，爱丽丝算出一组整数解为 $(x,y)=(2753,-15)$ ，即 $d=2753$ 。

至此所有计算完成。

第六步，将n和e封装成公钥，n和d封装成私钥。

在爱丽丝的例子中， $n=3233$, $e=17$, $d=2753$ ，所以公钥就是 $(3233, 17)$ ，私钥就是 $(3233, 2753)$ 。

实际应用中，公钥和私钥的数据都采用[ASN.1](#)格式表达（[实例](#)）。

七、RSA算法的可靠性

回顾上面的密钥生成步骤，一共出现六个数字：

p
q
n
 $\Phi(n)$
e
d

这六个数字之中，公钥用到了两个（n和e），其余四个数字都是不公开的。其中最关键的是d，因为n和d组成了私钥，一旦d泄漏，就等于私钥泄漏。

那么，有无可能在已知n和e的情况下，推导出d？

- (1) $ed \equiv 1 \pmod{\Phi(n)}$ 。只有知道e和 $\Phi(n)$ ，才能算出d。
- (2) $\Phi(n) = (p-1)(q-1)$ 。只有知道p和q，才能算出 $\Phi(n)$ 。
- (3) $n = pq$ 。只有将n因数分解，才能算出p和q。

结论：如果n可以被因数分解，d就可以算出，也就意味着私钥被破解。

可是，大整数的因数分解，是一件非常困难的事情。目前，除了暴力破解，还没有发现别的有效方法。维基百科这样写道：

"对极大整数做因数分解的难度决定了RSA算法的可靠性。换言之，对一极大整数做因数分解愈困难，RSA算法愈可靠。

假如有人找到一种快速因数分解的算法，那么RSA的可靠性就会极度下降。但找到这样的算法的可能性是非常小的。今天只有短的RSA密钥才可能被暴力破解。到2008年为止，世界上还没有任何可靠的攻击RSA算法的方式。

只要密钥长度足够长，用RSA加密的信息实际上是不能被解破的。"

举例来说，你可以对3233进行因数分解（ 61×53 ），但是你没法对下面这个整数进行因数分解。

12301866845301177551304949
58384962720772853569595334
79219732245215172640050726
36575187452021997864693899
56474942774063845925192557
32630345373154826850791702
61221429134616704292143116
02221240479274737794080665
351419597459856902143413

它等于这样两个质数的乘积：

33478071698956898786044169

84821269081770479498371376
85689124313889828837938780
02287614711652531743087737
814467999489
 \times
36746043666799590428244633
79962795263227915816434308
76426760322838157396665112
79233373417143396810270092
798736308917

事实上，这大概是人类已经分解的最大整数（232个十进制位，768个二进制位）。比它更大的因数分解，还没有被报道过，因此目前被破解的最长RSA密钥就是768位。

八、加密和解密

有了公钥和密钥，就能进行加密和解密了。

（1）加密要用公钥 (n,e)

假设鲍勃要向爱丽丝发送加密信息m，他就要用爱丽丝的公钥 (n,e) 对m进行加密。这里需要注意，m必须是整数（字符串可以取ascii值或unicode值），且m必须小于n。

所谓“加密”，就是算出下式的c：

$$m^e \equiv c \pmod{n}$$

爱丽丝的公钥是 (3233, 17)，鲍勃的m假设是65，那么可以算出下面的等式：

$$65^{17} \equiv 2790 \pmod{3233}$$

于是，c等于2790，鲍勃就把2790发给了爱丽丝。

（2）解密要用私钥(n,d)

爱丽丝拿到鲍勃发来的2790以后，就用自己的私钥(3233, 2753) 进行解密。可以证明，下面的等式一定成立：

$$c^d \equiv m \pmod{n}$$

也就是说，c的d次方除以n的余数为m。现在，c等于2790，私钥是(3233, 2753)，那么，爱丽丝算出

$$2790^{2753} \equiv 65 \pmod{3233}$$

因此，爱丽丝知道了鲍勃加密前的原文就是65。

至此，“加密--解密”的整个过程全部完成。

我们可以看到，如果不知道d，就没有办法从c求出m。而前面已经说过，要知道d就必须分解n，这是

极难做到的，所以RSA算法保证了通信安全。

你可能会问，公钥(n, e) 只能加密小于 n 的整数 m ，那么如果要加密大于 n 的整数，该怎么办？有两种解决方法：一种是把长信息分割成若干段短消息，每段分别加密；另一种是先选择一种“对称性加密算法”（比如[DES](#)），用这种算法的密钥加密信息，再用RSA公钥加密DES密钥。

九、私钥解密的证明

最后，我们来证明，为什么用私钥解密，一定可以正确地得到 m 。也就是证明下面这个式子：

$$c^d \equiv m \pmod{n}$$

因为，根据加密规则

$$m^e \equiv c \pmod{n}$$

于是， c 可以写成下面的形式：

$$c = m^e - kn$$

将 c 代入要我们要证明的那个解密规则：

$$(m^e - kn)^d \equiv m \pmod{n}$$

它等同于求证

$$m^{ed} \equiv m \pmod{n}$$

由于

$$ed \equiv 1 \pmod{\phi(n)}$$

所以

$$ed = h\phi(n) + 1$$

将 ed 代入：

$$m^{h\phi(n)+1} \equiv m \pmod{n}$$

接下来，分成两种情况证明上面这个式子。

(1) m 与 n 互质。

根据欧拉定理，此时

$$m^{\phi(n)} \equiv 1 \pmod{n}$$

得到

$$(m^{\phi(n)})^h \times m \equiv m \pmod{n}$$

原式得到证明。

(2) m 与 n 不是互质关系。

此时，由于 n 等于质数 p 和 q 的乘积，所以 m 必然等于 kp 或 kq 。

以 $m = kp$ 为例，考虑到这时 k 与 q 必然互质，则根据欧拉定理，下面的式子成立：

$$(kp)^{q-1} \equiv 1 \pmod{q}$$

进一步得到

$$[(kp)^{q-1}]^{h(p-1)} \times kp \equiv kp \pmod{q}$$

即

$$(kp)^{ed} \equiv kp \pmod{q}$$

将它改写成下面的等式

$$(kp)^{ed} = tq + kp$$

这时 t 必然能被 p 整除，即 $t=t'p$

$$(kp)^{ed} = t'pq + kp$$

因为 $m=kp$, $n=pq$, 所以

$$m^{ed} \equiv m \pmod{n}$$

原式得到证明。

(完)

RSA算法原理（一）

如果你问我，哪一种算法最重要？

我可能会回答"公钥加密算法"。

$$\frac{(x+1)^2}{y^2} = \left(\frac{x(x+2)}{2}\right)1 + (x(x-1))0 + \left(\frac{x(x-1)}{2}\right)$$
$$= \left(\frac{(x-1)(x-2)}{2}\right)1 + (x(x-1))0 + \left(\frac{x(x-1)}{2}\right)$$
$$= \frac{f_{12}(x, y)}{(y+6x+3)^4(2x+y+8x^3-y+9x+5)^4(y+1)}$$
$$= \frac{(x+1)(x+6)^4(x+9)^4}{x(x+1)(x+2)^4(x+8x+9b+\sqrt{3}\sqrt{4x^3+27b^2})^4(y+10x+8)^4(x+1)}$$
$$= \frac{2^{1/3}3^{2/3}}{x(x+6)^2} \frac{(y+9x+1)^2}{(y+8x)^2}$$
$$= \frac{(1-i\sqrt{3})(-9b+\sqrt{3}\sqrt{4x^3+27b^2})^{1/3}}{(49+27x+9)^{1/3}} \frac{(y+8x+1)^2}{(y+7x+4)^4(y+1)^2}$$

因为它是计算机通信安全的基石，保证了加密数据不会被破解。你可以想象一下，信用卡交易被破解的后果。

进入正题之前，我先简单介绍一下，什么是"公钥加密算法"。

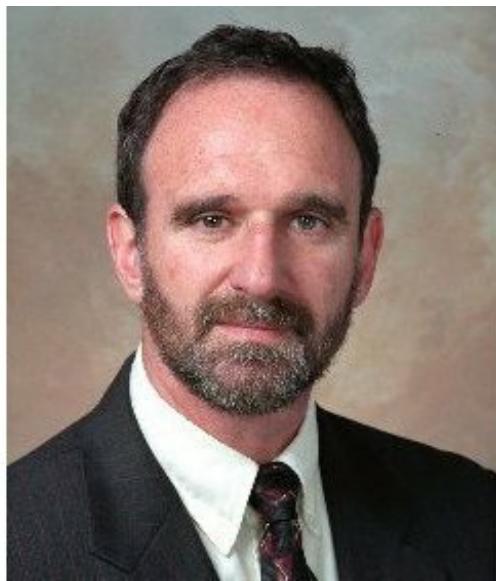
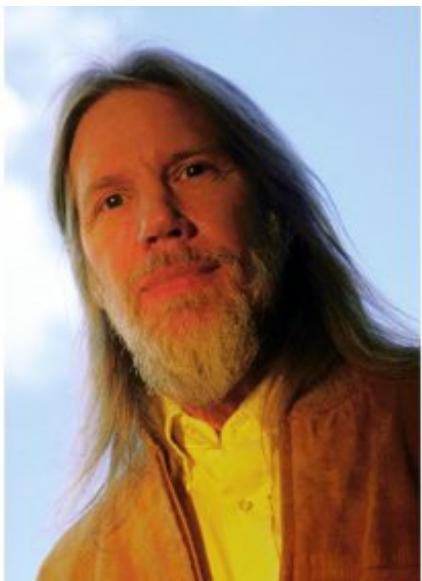
一、一点历史

1976年以前，所有的加密方法都是同一种模式：

- (1) 甲方选择某一种加密规则，对信息进行加密；
- (2) 乙方使用同一种规则，对信息进行解密。

由于加密和解密使用同样规则（简称"密钥"），这被称为"对称加密算法"（Symmetric-key algorithm）。

这种加密模式有一个最大弱点：甲方必须把加密规则告诉乙方，否则无法解密。保存和传递密钥，就成了最头疼的问题。

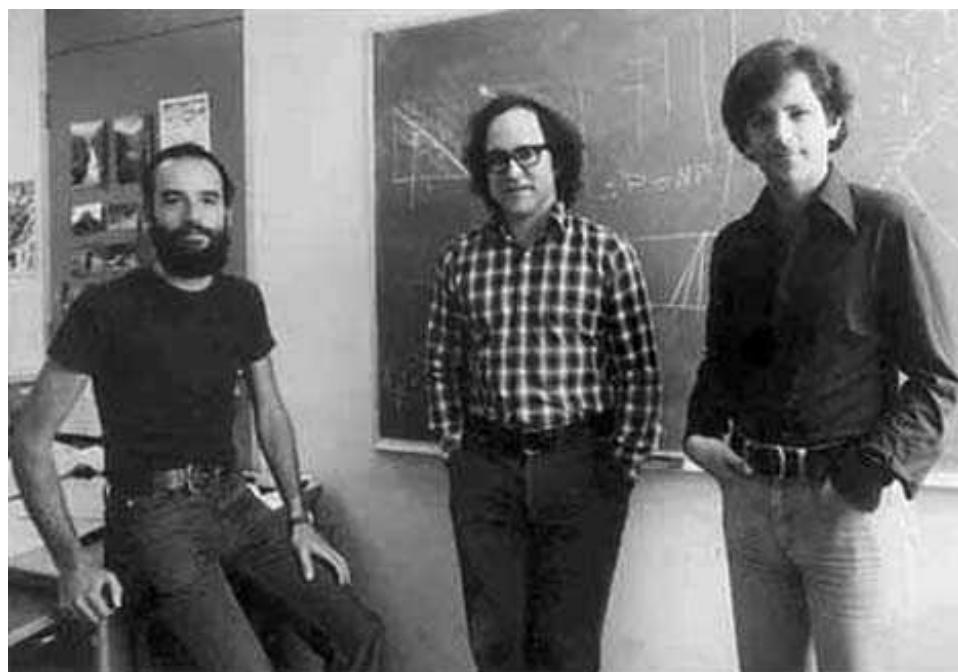


1976年，两位美国计算机学家Whitfield Diffie 和 Martin Hellman，提出了一种崭新构思，可以在不直接传递密钥的情况下，完成解密。这被称为"[Diffie-Hellman密钥交换算法](#)"。这个算法启发了其他科学家。人们认识到，加密和解密可以使用不同的规则，只要这两种规则之间存在某种对应关系即可，这样就避免了直接传递密钥。

这种新的加密模式被称为"非对称加密算法"。

- (1) 乙方生成两把密钥（公钥和私钥）。公钥是公开的，任何人都可以获得，私钥则是保密的。
- (2) 甲方获取乙方的公钥，然后用它对信息加密。
- (3) 乙方得到加密后的信息，用私钥解密。

如果公钥加密的信息只有私钥解得开，那么只要私钥不泄漏，通信就是安全的。



1977年，三位数学家Rivest、Shamir 和 Adleman 设计了一种算法，可以实现非对称加密。这种算法用他们三个人的名字命名，叫做[RSA算法](#)。从那时直到现在，RSA算法一直是最广为使用的"非对称加密算法"。毫不夸张地说，只要有计算机网络的地方，就有RSA算法。

这种算法非常[可靠](#)，密钥越长，它就越难破解。根据已经披露的文献，目前被破解的最长RSA密钥是768个二进制位。也就是说，长度超过768位的密钥，还无法破解（至少没人公开宣布）。因此可以认为，1024位的RSA密钥基本安全，2048位的密钥极其安全。

下面，我就进入正题，解释RSA算法的原理。文章共分成两部分，今天是第一部分，介绍要用到的四个数学概念。你可以看到，RSA算法并不难，只需要一点[数论知识](#)就可以理解。

二、互质关系

如果两个正整数，除了1以外，没有其他公因子，我们就称这两个数是[互质关系](#) (coprime)。比如，15和32没有公因子，所以它们是互质关系。这说明，不是质数也可以构成互质关系。

关于互质关系，不难得到以下结论：

1. 任意两个质数构成互质关系，比如13和61。
2. 一个数是质数，另一个数只要不是前者的倍数，两者就构成互质关系，比如3和10。
3. 如果两个数之中，较大的那个数是质数，则两者构成互质关系，比如97和57。
4. 1和任意一个自然数都是互质关系，比如1和99。
5. p 是大于1的整数，则 p 和 $p-1$ 构成互质关系，比如57和56。
6. p 是大于1的奇数，则 p 和 $p-2$ 构成互质关系，比如17和15。

三、欧拉函数

请思考以下问题：

任意给定正整数 n ，请问在小于等于 n 的正整数之中，有多少个与 n 构成互质关系？
(比如，在1到8之中，有多少个数与8构成互质关系？)

计算这个值的方法就叫做[欧拉函数](#)，以 $\phi(n)$ 表示。在1到8之中，与8形成互质关系的是1、3、5、7，所以 $\phi(n) = 4$ 。

$\phi(n)$ 的计算方法并不复杂，但是为了得到最后那个公式，需要一步步讨论。

第一种情况

如果 $n=1$ ，则 $\phi(1) = 1$ 。因为1与任何数（包括自身）都构成互质关系。

第二种情况

如果 n 是质数，则 $\phi(n)=n-1$ 。因为质数与小于它的每一个数，都构成互质关系。比如5与1、2、3、4

都构成互质关系。

第三种情况

如果n是质数的某一个次方，即 $n = p^k$ (p 为质数， k 为大于等于1的整数)，则

$$\phi(p^k) = p^k - p^{k-1}$$

比如 $\phi(8) = \phi(2^3) = 2^3 - 2^2 = 8 - 4 = 4$ 。

这是因为只有当一个数不包含质数p，才可能与n互质。而包含质数p的数一共有 $p^{(k-1)}$ 个，即 $1 \times p$ 、 $2 \times p$ 、 $3 \times p$ 、...、 $p^{(k-1)} \times p$ ，把它们去除，剩下的就是与n互质的数。

上面的式子还可以写成下面的形式：

$$\phi(p^k) = p^k - p^{k-1} = p^k \left(1 - \frac{1}{p}\right)$$

可以看出，上面的第二种情况是 $k=1$ 时的特例。

第四种情况

如果n可以分解成两个互质的整数之积，

$$n = p_1 \times p_2$$

则

$$\Phi(n) = \Phi(p_1 p_2) = \Phi(p_1) \Phi(p_2)$$

即积的欧拉函数等于各个因子的欧拉函数之积。比如， $\Phi(56) = \Phi(8 \times 7) = \Phi(8) \times \Phi(7) = 4 \times 6 = 24$ 。

这一条的证明要用到["中国剩余定理"](#)，这里就不展开了，只简单说一下思路：如果a与 p_1 互质($a < p_1$)，b与 p_2 互质($b < p_2$)，c与 $p_1 p_2$ 互质($c < p_1 p_2$)，则c与数对(a,b)是一一对应关系。由于a的值有 $\Phi(p_1)$ 种可能，b的值有 $\Phi(p_2)$ 种可能，则数对(a,b)有 $\Phi(p_1) \Phi(p_2)$ 种可能，而c的值有 $\Phi(p_1 p_2)$ 种可能，所以 $\Phi(p_1 p_2)$ 就等于 $\Phi(p_1) \Phi(p_2)$ 。

第五种情况

因为任意一个大于1的正整数，都可以写成一系列质数的积。

$$n = p_1^{k_1} p_2^{k_2} \dots p_r^{k_r}$$

根据第4条的结论，得到

$$\phi(n) = \phi(p_1^{k_1}) \phi(p_2^{k_2}) \dots \phi(p_r^{k_r})$$

再根据第3条的结论，得到

$$\phi(n) = p_1^{k_1} p_2^{k_2} \dots p_r^{k_r} \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_r}\right)$$

也就等于

$$\phi(n) = n \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \dots \left(1 - \frac{1}{p_r}\right)$$

这就是欧拉函数的通用计算公式。比如，1323的欧拉函数，计算过程如下：

$$\phi(1323) = \phi(3^3 \times 7^2) = 1323 \left(1 - \frac{1}{3}\right) \left(1 - \frac{1}{7}\right) = 756$$

四、欧拉定理

欧拉函数的用处，在于[欧拉定理](#)。"欧拉定理"指的是：

如果两个正整数a和n互质，则n的欧拉函数 $\phi(n)$ 可以让下面的等式成立：

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

也就是说，a的 $\phi(n)$ 次方被n除的余数为1。或者说，a的 $\phi(n)$ 次方减去1，可以被n整除。比如，3和7互质，而7的欧拉函数 $\phi(7)$ 等于6，所以3的6次方（729）减去1，可以被7整除（728/7=104）。

欧拉定理的证明比较复杂，这里就省略了。我们只要记住它的结论就行了。

欧拉定理可以大大简化某些运算。比如，7和10互质，根据欧拉定理，

$$7^{\phi(10)} \equiv 1 \pmod{10}$$

已知 $\phi(10)$ 等于4，所以马上得到7的4倍数次方的个位数肯定是1。

$$7^{4k} \equiv 1 \pmod{10}$$

因此，7的任意次方的个位数（例如7的222次方），心算就可以算出来。

欧拉定理有一个特殊情况。

假设正整数a与质数p互质，因为质数p的 $\phi(p)$ 等于 $p-1$ ，则欧拉定理可以写成

$$a^{p-1} \equiv 1 \pmod{p}$$

这就是著名的费马小定理。它是欧拉定理的特例。

欧拉定理是RSA算法的核心。理解了这个定理，就可以理解RSA。

五、模反元素

还剩下最后一个概念：

如果两个正整数a和n互质，那么一定可以找到整数b，使得 $ab-1$ 被n整除，或者说ab被n除的余数是1。

$$ab \equiv 1 \pmod{n}$$

这时，b就叫做a的“模反元素”。

比如，3和11互质，那么3的模反元素就是4，因为 $(3 \times 4)-1$ 可以被11整除。显然，模反元素不止一个，4加减11的整数倍都是3的模反元素 $\{-18, -7, 4, 15, 26, \dots\}$ ，即如果b是a的模反元素，则 $b+kn$ 都是a的模反元素。

欧拉定理可以用来证明模反元素必然存在。

$$a^{\phi(n)} = a \times a^{\phi(n)-1} \equiv 1 \pmod{n}$$

可以看到，a的 $\phi(n)-1$ 次方，就是a的模反元素。

=====

好了，需要用到的数学工具，全部介绍完了。RSA算法涉及的数学知识，就是上面这些，下一次我就来介绍公钥和私钥到底是怎么生成的。

(完)

字符串匹配的Boyer-Moore算法

上一篇文章，我介绍了[KMP算法](#)。

但是，它并不是效率最高的算法，实际采用并不多。各种文本编辑器的"查找"功能（Ctrl+F），大多采用[Boyer-Moore算法](#)。



Boyer-Moore算法不仅效率高，而且构思巧妙，容易理解。1977年，德克萨斯大学的Robert S. Boyer教授和J Strother Moore教授发明了这种算法。

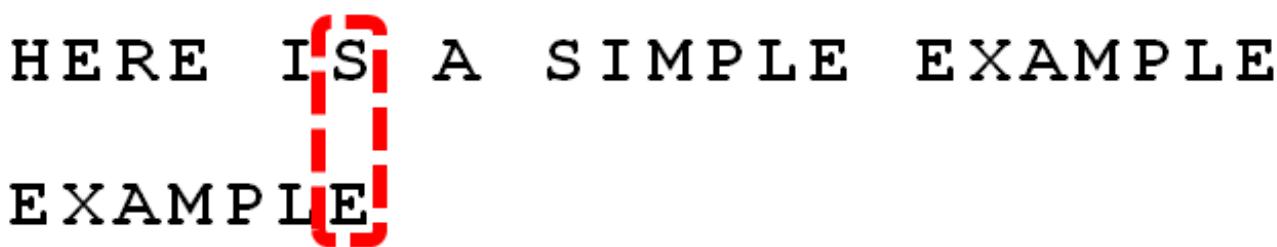
下面，我根据Moore教授自己的[例子](#)来解释这种算法。

1.



假定字符串为"HERE IS A SIMPLE EXAMPLE"，搜索词为"EXAMPLE"。

2.



首先，"字符串"与"搜索词"头部对齐，从尾部开始比较。

这是一个很聪明的想法，因为如果尾部字符不匹配，那么只要一次比较，就可以知道前7个字符（整体上）肯定不是要找的结果。

我们看到，"S"与"E"不匹配。这时，"S"就被称为"坏字符"（bad character），即不匹配的字符。我们还发现，"S"不包含在搜索词"EXAMPLE"之中，这意味着可以把搜索词直接移到"S"的后一位。

3.

HERE IS A SIMPLE EXAMPLE
EXAMPLE

依然从尾部开始比较，发现"P"与"E"不匹配，所以"P"是"坏字符"。但是，"P"包含在搜索词"EXAMPLE"之中。所以，将搜索词后移两位，两个"P"对齐。

4.

HERE IS A SIMPLE EXAMPLE
EXAMPLE

我们由此总结出"坏字符规则"：

后移位数 = 坏字符的位置 - 搜索词中的上一次出现位置

如果"坏字符"不包含在搜索词之中，则上一次出现位置为 -1。

以"P"为例，它作为"坏字符"，出现在搜索词的第6位（从0开始编号），在搜索词中的上一次出现位置为4，所以下移 $6 - 4 = 2$ 位。再以前面第二步的"S"为例，它出现在第6位，上一次出现位置是 -1（即未出现），则整个搜索词后移 $6 - (-1) = 7$ 位。

5.

HERE IS A SIMPLE EXAMPLE



依然从尾部开始比较，"E"与"E"匹配。

6.

HERE IS A SIMPLE EXAMPLE



比较前面一位，"LE"与"LE"匹配。

7.

HERE IS A SIMPLE EXAMPLE



比较前面一位，"PLE"与"PLE"匹配。

8.

HERE IS A SIMPLE EXAMPLE



比较前面一位，"MPL"与"MPLE"匹配。我们把这种情况称为"好后缀"（good suffix），即所有尾部匹配的字符串。注意，"MPLE"、"PLE"、"LE"、"E"都是好后缀。

9.

HERE IS A SIMPLE EXAMPLE



比较前一位，发现"I"与"A"不匹配。所以，"I"是"坏字符"。

10.

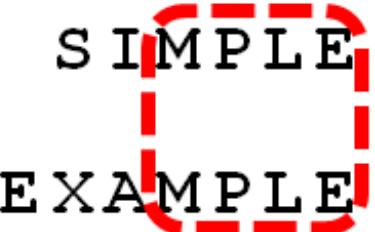
HERE IS A SIMPLE EXAMPLE



根据"坏字符规则"，此时搜索词应该后移 $2 - (-1) = 3$ 位。问题是，此时有没有更好的移法？

11.

HERE IS A SIMPLE EXAMPLE



我们知道，此时存在"好后缀"。所以，可以采用"好后缀规则"：

后移位数 = 好后缀的位置 - 搜索词中的上一次出现位置

举例来说，如果字符串"ABCDAB"的后一个"AB"是"好后缀"。那么它的位置是5（从0开始计算，取最后的"B"的值），在"搜索词中的上一次出现位置"是1（第一个"B"的位置），所以后移 $5 - 1 = 4$ 位，前一个"AB"移到后一个"AB"的位置。

再举一个例子，如果字符串"ABCDEF"的"EF"是好后缀，则"EF"的位置是5，上一次出现的位置是-1（即未出现），所以后移 $5 - (-1) = 6$ 位，即整个字符串移到"F"的后一位。

这个规则有三个注意点：

(1) "好后缀"的位置以最后一个字符为准。假定"ABCDEF"的"EF"是好后缀，则它的位置以"F"为准，即5（从0开始计算）。

(2) 如果"好后缀"在搜索词中只出现一次，则它的上一次出现位置为-1。比

如，"EF"在"ABCDEF"之中只出现一次，则它的上一次出现位置为-1（即未出现）。

(3) 如果"好后缀"有多个，则除了最长的那个"好后缀"，其他"好后缀"的上一次出现位置必须在头部。比如，假定"BABCDAB"的"好后缀"是"DAB"、"AB"、"B"，请问这时"好后缀"的上一次出现位置是什么？回答是，此时采用的好后缀是"B"，它的上一次出现位置是头部，即第0位。这个规则也可以这样表达：如果最长的那个"好后缀"只出现一次，则可以把搜索词改写成如下形式进行位置计算"(DA)BABCDAB"，即虚拟加入最前面的"DA"。

回到上文的这个例子。此时，所有的"好后缀"（MPLE、PLE、LE、E）之中，只有"E"在"EXAMPLE"还出现在头部，所以后移 $6 - 0 = 6$ 位。

12.

HERE IS A SIMPLE EXAMPLE
EXAMPLE

可以看到，"坏字符规则"只能移3位，"好后缀规则"可以移6位。所以，**Boyer-Moore算法的基本思想是，每次后移这两个规则之中的较大值。**

更巧妙的是，这两个规则的移动位数，只与搜索词有关，与原字符串无关。因此，可以预先计算生成《坏字符规则表》和《好后缀规则表》。使用时，只要查表比较一下就可以了。

13.

HERE IS A SIMPLE EXAMPLE
EXAMPLE

继续从尾部开始比较，"P"与"E"不匹配，因此"P"是"坏字符"。根据"坏字符规则"，后移 $6 - 4 = 2$ 位。

14.

HERE IS A SIMPLE EXAMPLE
EXAMPLE

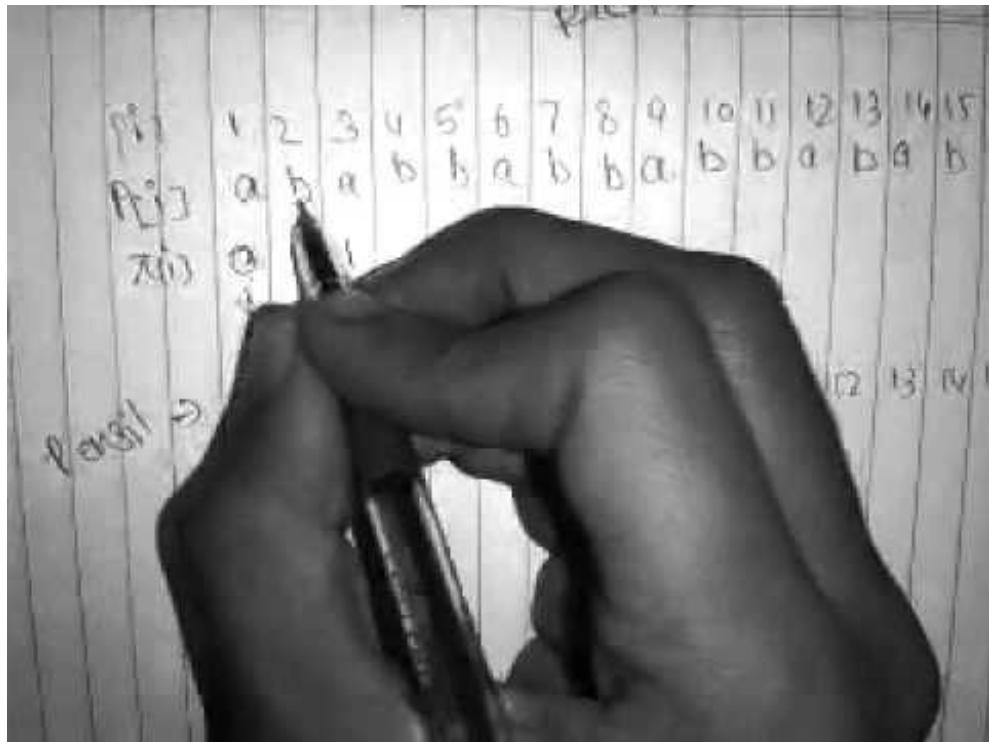
从尾部开始逐位比较，发现全部匹配，于是搜索结束。如果还要继续查找（即找出全部匹配），则根据“好后缀规则”，后移 $6 - 0 = 6$ 位，即头部的“E”移到尾部的“E”的位置。

(完)

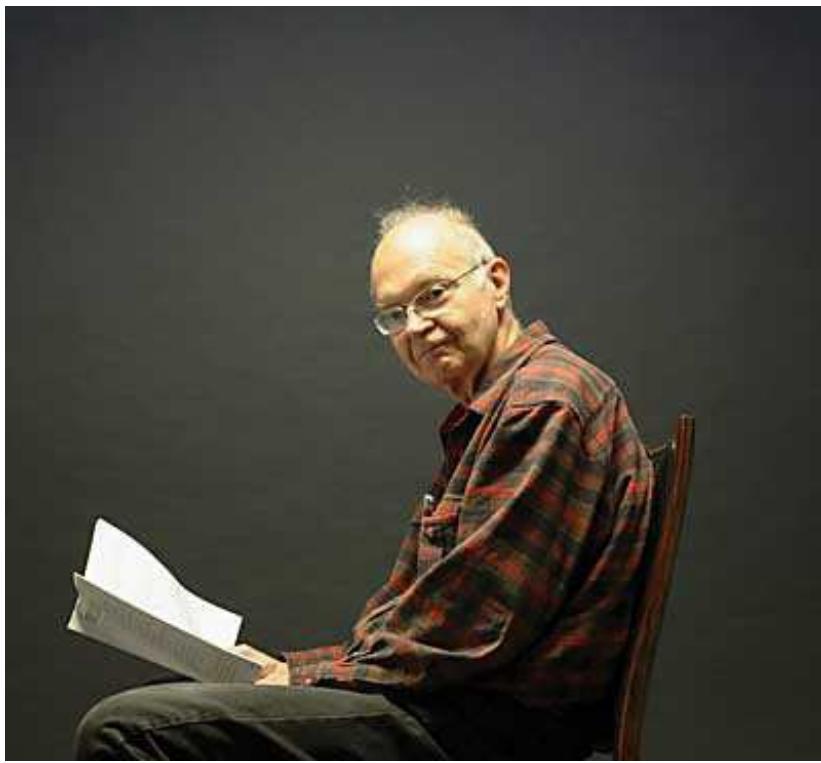
字符串匹配的KMP算法

[字符串匹配](#)是计算机的基本任务之一。

举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？



许多算法可以完成这个任务，[Knuth-Morris-Pratt算法](#)（简称KMP）是最常用的之一。它以三个发明者命名，起头的那个K就是著名科学家Donald Knuth。



这种算法不太容易理解，网上有很多[解释](#)，但读起来都很费劲。直到读到[Jake Boxer](#)的文章，我才真正理解这种算法。下面，我用自己的语言，试图写一篇比较好懂的KMP算法解释。

1.

BBC ABCDAB ABCDABCDABDE
ABCDABD

首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。

2.

BBC ABCDAB ABCDABCDABDE
ABCDABD

因为B与A不匹配，搜索词再往后移。

3.

BBC ABCDAB ABCDABCDABDE
ABCDABD

就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。

4.

BBC ABCDAB ABCDABCDABDE
ABCDABD

接着比较字符串和搜索词的下一个字符，还是相同。

5.

BBC ABCDAB ABCDABCDABDE
ABCDABD

直到字符串有一个字符，与搜索词对应的字符不相同为止。

6.

BBC ABCDAB ABCDABCDABDE
ABCDABD

这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

7.

BBC ABCDAB [] ABCDABCDABDE
ABCDABD []

一个基本事实是，当空格与D不匹配时，你其实知道前面六个字符是"ABCDAB"。KMP算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。

8.

搜索词	A B C D A B D
部分匹配值	0 0 0 0 1 2 0

怎么做到这一点呢？可以针对搜索词，算出一张《部分匹配表》（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。

9.

BBC ABCDAB [] ABCDABCDABDE
ABCDABD []

已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符B对应的"部分匹配值"为2，因此按照下面的公式算出向后移动的位数：

$$\text{移动位数} = \text{已匹配的字符数} - \text{对应的部分匹配值}$$

因为 $6 - 2$ 等于4，所以将搜索词向后移动4位。

10.

BBC ABCDAB [] ABCDABCDABDE
ABCDABD

因为空格与C不匹配，搜索词还要继续往后移。这时，已匹配的字符数为2 ("AB")，对应的"部分匹配值"为0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移2位。

11.

BBC ABCDAB [] ABCDABCDABDE
ABCDABD

因为空格与A不匹配，继续后移一位。

12.

BBC ABCDAB ABCDABC [] DABDE
ABCDABD

逐位比较，直到发现C与D不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动4位。

13.

□

逐位比较，直到搜索词的最后一一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动7位，这里就不再重复了。

14.

□

下面介绍《部分匹配表》是如何产生的。

首先，要了解两个概念："前缀"和"后缀"。"前缀"指除了最后一个字符以外，一个字符串的全部头部组合；"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

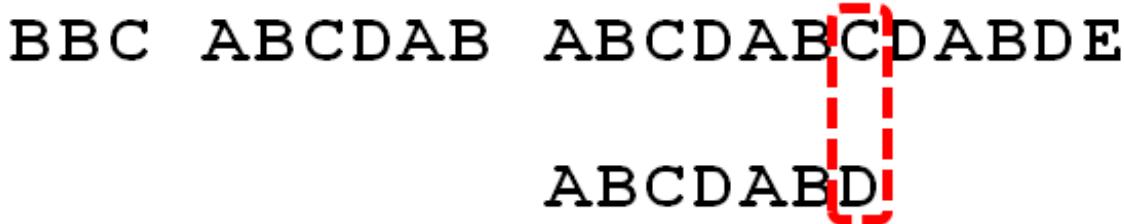
15.

搜索词	A	B	C	D	A	B	D
部分匹配值	0	0	0	0	1	2	0

"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以"ABCDABD"为例，

- "A"的前缀和后缀都为空集，共有元素的长度为0；
- "AB"的前缀为[A]，后缀为[B]，共有元素的长度为0；
- "ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；
- "ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；
- "ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为1；
- "ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为2；
- "ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。

16.



BBC ABCDAB ABCDAB**DABD**E
ABCDABD

"部分匹配"的实质是，有时候，字符串头部和尾部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是2 ("AB"的长度)。搜索词移动的时候，第一个"AB"向后移动4位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

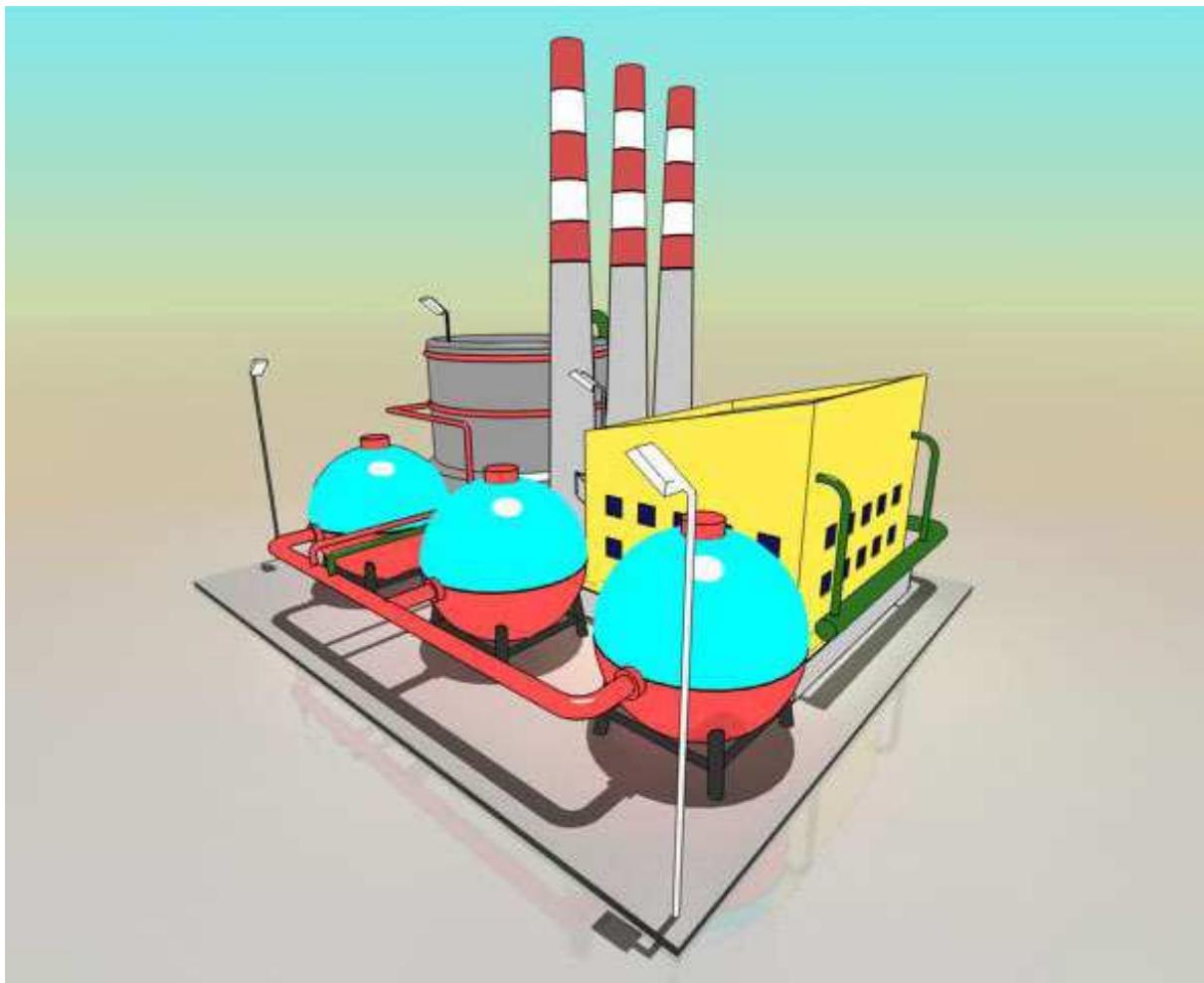
(完)

进程与线程的一个简单解释

[进程](#) (process) 和[线程](#) (thread) 是操作系统的基本概念，但是它们比较抽象，不容易掌握。

最近，我读到一篇[材料](#)，发现有一个很好的类比，可以把它们解释地清晰易懂。

1.



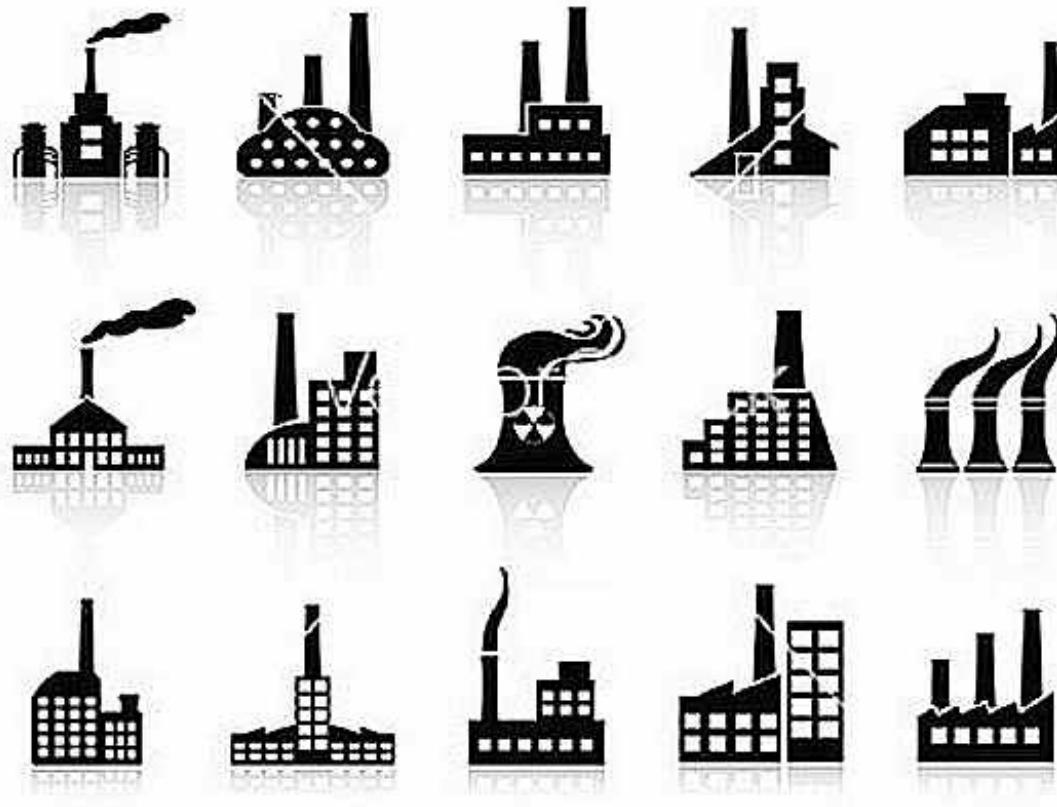
计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。

2.



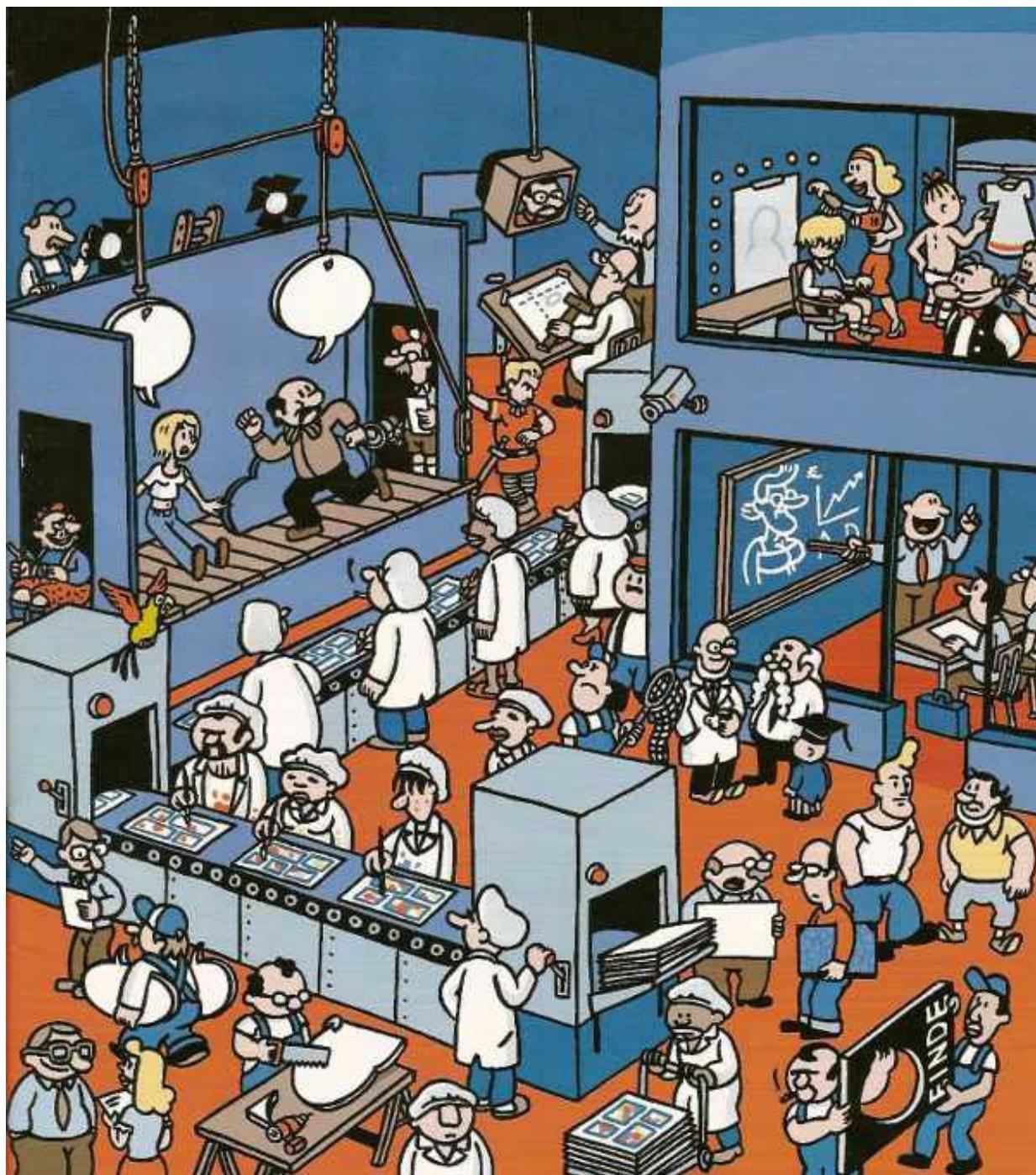
假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个CPU一次只能运行一个任务。

3.



进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。

4.



一个车间里，可以有很多工人。他们协同完成一个任务。



线程就好比车间里的工人。一个进程可以包括多个线程。

6.

□

车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。

7.



可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。

8.



一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫“互斥锁”（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。

9.



©joy-of-cartoon-pictures.com

还有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。

10.



这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做“信号量”（Semaphore），用来保证多个线程不会互相冲突。

不难看出，mutex是semaphore的一种特殊情况（n=1时）。也就是说，完全可以用后者替代前者。

但是，因为mutex较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。

11.



操作系统的*设计*，因此可以归结为三点：

- (1) 以多进程形式，允许多个任务同时运行；
- (2) 以多线程形式，允许单个任务分成不同的部分运行；
- (3) 提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

(完)

计算机是如何启动的？

从打开电源到开始操作，计算机的启动是一个非常复杂的过程。



我一直搞不清楚，这个过程到底是怎么回事，只看见屏幕快速滚动各种提示……这几天，我查了一些资料，试图搞懂它。下面就是我整理的笔记。

零、boot的含义

先问一个问题，"启动"用英语怎么说？

回答是boot。可是，boot原来的意思是靴子，"启动"与靴子有什么关系呢？原来，这里的boot是bootstrap（鞋带）的缩写，它来自一句谚语：

"pull oneself up by one's bootstraps"

字面意思是"拽着鞋带把自己拉起来"，这当然是不可能的事情。最早的时候，工程师们用它来比喻，计算机启动是一个很矛盾的过程：必须先运行程序，然后计算机才能启动，但是计算机不启动就无法运行程序！

早期真的是这样，必须想尽各种办法，把一小段程序装进内存，然后计算机才能正常运行。所以，工程师们把这个过程叫做"拉鞋带"，久而久之就简称为boot了。

计算机的整个启动过程分成四个阶段。

一、第一阶段：BIOS

上个世纪70年代初，"只读内存"（read-only memory，缩写为ROM）发明，开机程序被刷入ROM芯片，计算机通电后，第一件事就是读取它。



这块芯片里的程序叫做"基本輸出輸入系統"（Basic Input/Output System），简称为[BIOS](#)。

1.1 硬件自检

BIOS程序首先检查，计算机硬件能否满足运行的基本条件，这叫做"硬件自检"（Power-On Self-Test），缩写为[POST](#)。

如果硬件出现问题，主板会发出不同含义的[蜂鸣](#)，启动中止。如果没有问题，屏幕就会显示出CPU、内存、硬盘等信息。

Diskette Drive B : None	Serial Port(s) : 3F0 2F0
Pri. Master Disk : LBA,ATA 100, 250GB	Parallel Port(s) : 370
Pri. Slave Disk : LBA,ATA 100, 250GB DDR at Bank(s)	: 0 1 2
Sec. Master Disk : None	
Sec. Slave Disk : None	

Pri. Master Disk HDD S.M.A.R.T. capability ... Disabled
 Pri. Slave Disk HDD S.M.A.R.T. capability ... Disabled

PCI Devices Listing ...

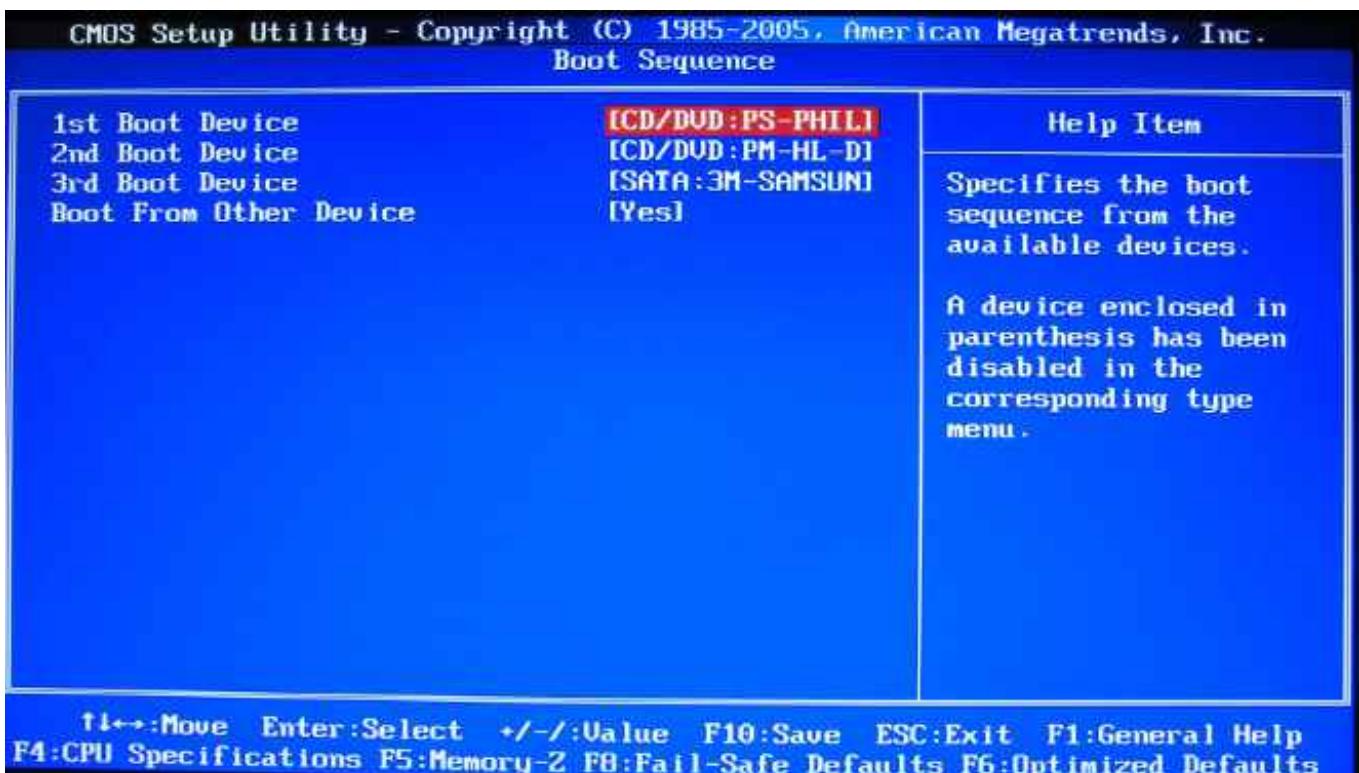
Bus	Dev	Fun	Vendor	Device	SVID	SSID	Class	Device Class	IRQ
0	27	0	8086	2668	1458	A005	0403	Multimedia Device	5
0	29	0	8086	2658	1458	2658	0C03	USB 1.1 Host Cntrlr	9
0	29	1	8086	2659	1458	2659	0C03	USB 1.1 Host Cntrlr	11
0	29	2	8086	265A	1458	265A	0C03	USB 1.1 Host Cntrlr	11
0	29	3	8086	265B	1458	265A	0C03	USB 1.1 Host Cntrlr	5
0	29	7	8086	265C	1458	5006	0C03	USB 1.1 Host Cntrlr	9
0	31	2	8086	2651	1458	2651	0101	IDE Cntrlr	14
0	31	3	8086	266A	1458	266A	0C05	SMBus Cntrlr	11
1	0	0	10DE	0421	10DE	0479	0300	Display Cntrlr	5
2	0	0	1283	8212	0000	0000	0180	Mass Storage Cntrlr	10
2	5	0	11AB	4320	1458	E000	0200	Network Cntrlr	12
								ACPI Controller	9

1.2 启动顺序

硬件自检完成后，BIOS把控制权转交给下一阶段的启动程序。

这时，BIOS需要知道，“下一阶段的启动程序”具体存放在哪一个设备。也就是说，BIOS需要有一个外部储存设备的排序，排在前面的设备就是优先转交控制权的设备。这种排序叫做“启动顺序”（Boot Sequence）。

打开BIOS的操作界面，里面有一项就是“设定启动顺序”。



二、第二阶段：主引导记录

BIOS按照"启动顺序"，把控制权转交给排在第一位的储存设备。

这时，计算机读取该设备的第一个扇区，也就是读取最前面的512个字节。如果这512个字节的最后两个字节是0x55和0xAA，表明这个设备可以用于启动；如果不是，表明设备不能用于启动，控制权于是被转交给"启动顺序"中的下一个设备。

这最前面的512个字节，就叫做["主引导记录"](#)（Master boot record，缩写为MBR）。

2.1 主引导记录的结构

"主引导记录"只有512个字节，放不了太多东西。它的主要作用是，告诉计算机到硬盘的哪一个位置去找操作系统。

主引导记录由三个部分组成：

- (1) 第1-446字节：调用操作系统的机器码。
- (2) 第447-510字节：分区表（Partition table）。
- (3) 第511-512字节：主引导记录签名（0x55和0xAA）。

其中，第二部分"分区表"的作用，是将硬盘分成若干个区。

2.2 分区表

硬盘分区有很多[好处](#)。考虑到每个区可以安装不同的操作系统，"主引导记录"因此必须知道将控制权转交给哪个区。

分区表的长度只有64个字节，里面又分成四项，每项16个字节。所以，一个硬盘最多只能分四个一级分区，又叫做"主分区"。

每个主分区的16个字节，由6个部分组成：

- (1) 第1个字节：如果为0x80，就表示该主分区是激活分区，控制权要转交给这个分区。四个主分区里面只能有一个是激活的。
- (2) 第2-4个字节：主分区第一个扇区的物理位置（柱面、磁头、扇区号等等）。
- (3) 第5个字节：[主分区类型](#)。
- (4) 第6-8个字节：主分区最后一个扇区的物理位置。
- (5) 第9-12字节：该主分区第一个扇区的逻辑地址。
- (6) 第13-16字节：主分区的扇区总数。

最后的四个字节（"主分区的扇区总数"），决定了这个主分区的长度。也就是说，一个主分区的扇区总数最多不超过 2^{32} 次方。

如果每个扇区为512个字节，就意味着单个分区最大不超过2TB。再考虑到扇区的逻辑地址也是32位，所以单个硬盘可利用的空间最大也不超过2TB。如果想使用更大的硬盘，只有2个方法：一是提高每个扇区的字节数，二是[增加扇区总数](#)。

三、第三阶段：硬盘启动

这时，计算机的控制权就要转交给硬盘的某个分区了，这里又分成三种情况。

3.1 情况A：卷引导记录

上一节提到，四个主分区里面，只有一个分区是激活的。计算机会读取激活分区的第一个扇区，叫做"[卷引导记录](#)"（Volume boot record，缩写为VBR）。

"卷引导记录"的主要作用是，告诉计算机，操作系统在这个分区里的位置。然后，计算机就会加载操作系统了。

3.2 情况B：扩展分区和逻辑分区

随着硬盘越来越大，四个主分区已经不够了，需要更多的分区。但是，分区表只有四项，因此规定有且仅有一个分区可以被定义成"扩展分区"（Extended partition）。

所谓"扩展分区"，就是指这个分区里面又分成多个分区。这种分区里面的分区，就叫做"逻辑分区"（logical partition）。

计算机先读取扩展分区的第一个扇区，叫做"[扩展引导记录](#)"（Extended boot record，缩写为EBR）。它里面也包含一张64字节的分区表，但是最多只有两项（也就是两个逻辑分区）。

计算机接着读取第二个逻辑分区的第一个扇区，再从里面的分区表中找到第三个逻辑分区的位置，以此类推，直到某个逻辑分区的分区表只包含它自身为止（即只有一个分区项）。因此，扩展分区可以

包含无数个逻辑分区。

但是，似乎很少通过这种方式启动操作系统。如果操作系统确实安装在扩展分区，一般采用下一种方式启动。

3.3 情况C：启动管理器

在这种情况下，计算机读取“主引导记录”前面446字节的机器码之后，不再把控制权转交给某一个分区，而是运行事先安装的“启动管理器”（boot loader），由用户选择启动哪一个操作系统。

Linux环境中，目前最流行的启动管理器是[Grub](#)。

□

四、第四阶段：操作系统

控制权转交给操作系统后，操作系统的内核首先被载入内存。

以Linux系统为例，先载入/boot目录下面的kernel。内核加载成功后，第一个运行的程序是/sbin/init。它根据配置文件（Debian系统是/etc/inittab）产生init进程。这是Linux启动后的第一个进程，pid进程编号为1，其他进程都是它的后代。

然后，init线程加载系统的各个模块，比如窗口程序和网络程序，直至执行/bin/login程序，跳出登录界面，等待用户输入用户名和密码。

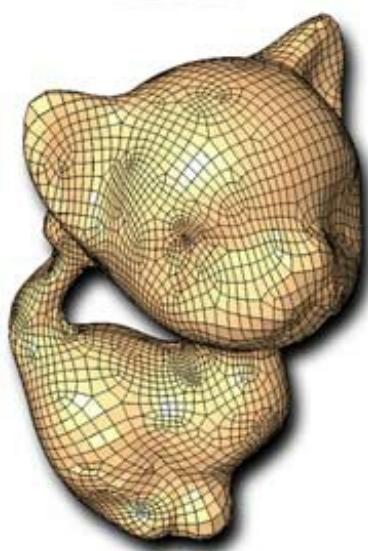
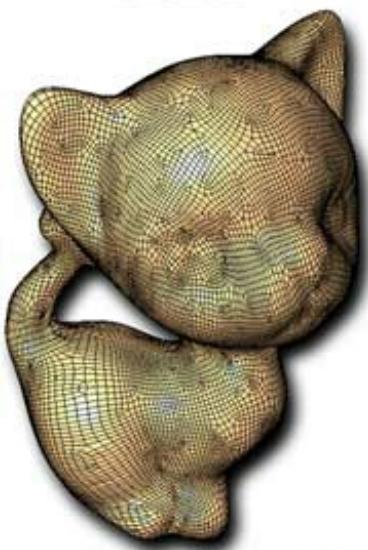
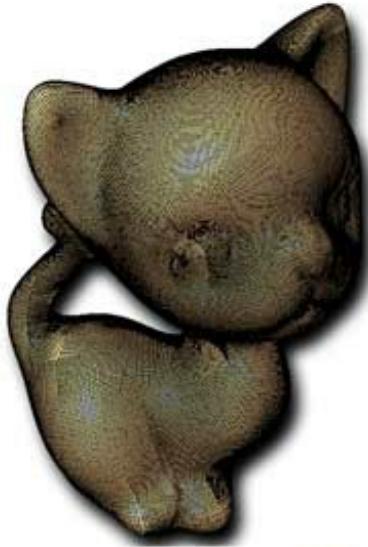
至此，全部启动过程完成。

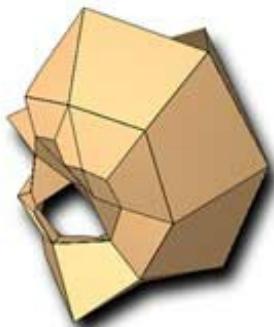
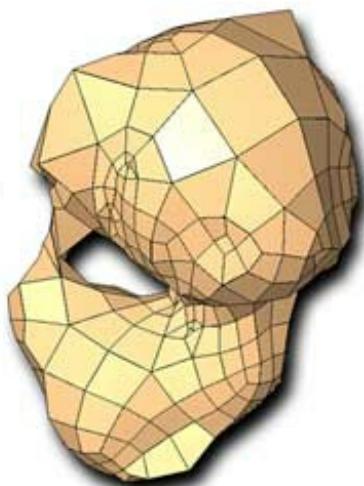
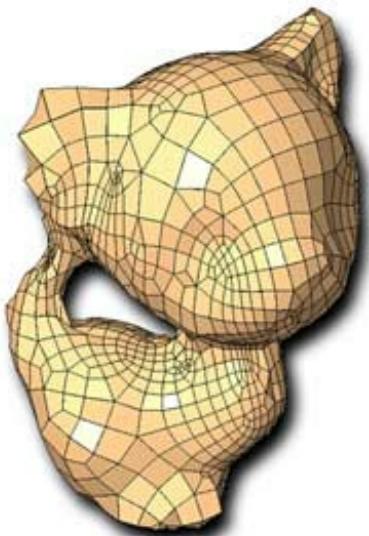
（完）

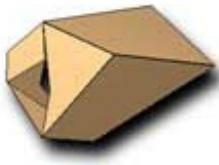
代码的抽象三原则

软件开发是“抽象化”原则 (Abstraction) 的一种体现。

所谓“抽象化”，就是指从具体问题中，提取出具有共性的模式，再使用通用的解决方法加以处理。







开发软件的时候，一方面，我们总是希望使用别人已经写好的代码，另一方面，又希望自己写的代码尽可能重用，以求减少工作量。要做到这两个目标，这需要“抽象化”。

最近，我读到美国程序员[Derick Bailey](#)的一篇文章，谈到“抽象化”应该遵循的三个原则，觉得很有启发。

一、DRY原则

[DRY](#)是 Don't repeat yourself 的缩写，意思是“不要重复自己”。

□

软件工程名著[《The Pragmatic Programmer》](#)首先提出了这个原则。它的涵义是，系统的每一个功能都应该有唯一的实现。也就是说，如果多次遇到同样的问题，就应该抽象出一个共同的解决方法，不要重复开发同样的功能。

这个原则有时也称为[“一次且仅一次”原则](#)（Once and Only Once）。

二、YAGNI原则

[YAGNI](#)是 You aren't gonna need it 的缩写，意思是“你不会需要它”。

□

这是[“极限编程”](#)提倡的原则，指的是你自以为有用的功能，实际上都是用不到的。因此，除了最核心的功能，其他功能一概不要部署，这样可以大大加快开发。

它背后的指导思想，就是尽可能快、尽可能简单地让软件运行起来（do the simplest thing that could possibly work）。

但是，这里出现了一个问题。仔细推敲的话，你会发现DRY原则和YAGNI原则并非完全兼容。前者追求“抽象化”，要求找到通用的解决方法；后者追求“快和省”，意味着不要把精力放在抽象化上面，因为很可能“你不会需要它”。所以，就有了第三个原则。

三、Rule Of Three原则

[Rule of three](#) 称为“三次原则”，指的是当某个功能第三次出现时，才进行“抽象化”。

REFACTORING

IMPROVING THE DESIGN
OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant,
William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International Inc.



这是软件开发大家[Martin Fowler](#)在《Refactoring》一书中提出的。

它的涵义是，第一次用到某个功能时，你写一个特定的解决方法；第二次又用到的时候，你拷贝上一次的代码；第三次出现的时候，你才着手"抽象化"，写出通用的解决方法。

这样做有几个理由：

- (1) 省事。如果一种功能只有一到两个地方会用到，就不需要在"抽象化"上面耗费时间了。
- (2) 容易发现模式。"抽象化"需要找到问题的模式，问题出现的场合越多，就越容易看出模式，从而可以更准确地"抽象化"。

比如，对于一个数列来说，两个元素不足以判断出规律：

1, 2, __, __, __, __,

第三个元素出现后，规律就变得较清晰了：

1, 2, 4, __, __, __,

- (3) 防止过度冗余。如果一种功能同时有多个实现，管理起来非常麻烦，修改的时候需要修改多处。在实际工作中，重复实现最多可以容忍出现一次，再多就无法接受了。

综上所述，"三次原则"是DRY原则和YAGNI原则的折衷，是代码冗余和开发成本的平衡点，值得我们在"抽象化"时遵循。

=====

附：

[广告]

图灵公司打算重新出版《[Joel on Software](#)》的中文版，正在寻找译者。如果你对此有兴趣，请与朱巍编辑联系（Email：[zhuw\(at\)turingbook.com](mailto:zhuw(at)turingbook.com)）试译。关于此书的更多情况，可参考我翻译的续集[《More Joel on Software》](#)。特别提醒：翻译是非常辛苦、但是报酬很低的工作，写信前请想清楚，你是真的想翻译这本书。

(完)

互联网协议入门（二）

[上一篇文章](#)分析了互联网的总体构思，从下至上，每一层协议的设计思想。

这是从设计者的角度看问题，今天我想切换到用户的角度，看看用户是如何从上至下，与这些协议互动的。

=====

互联网协议入门（二）

作者：阮一峰

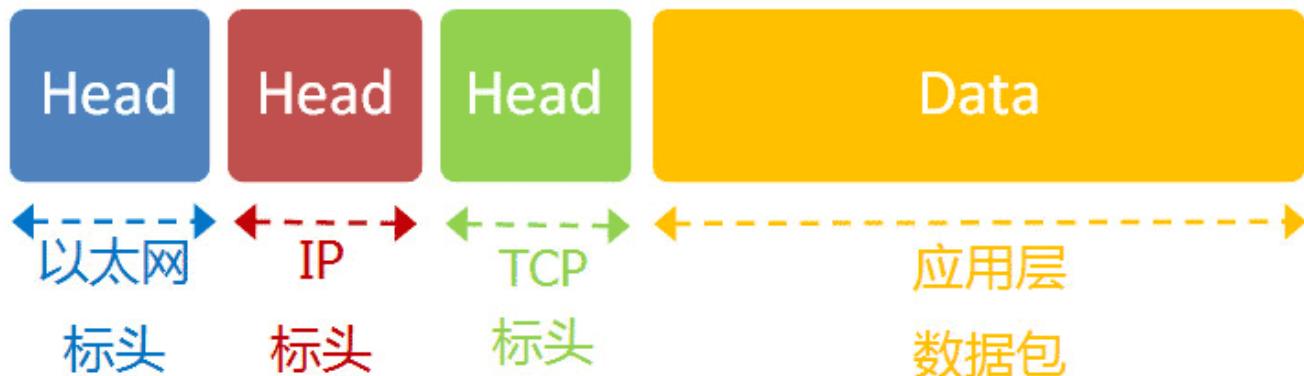
□

（接上文）

七、一个小结

先对前面的内容，做一个小结。

我们已经知道，网络通信就是交换数据包。电脑A向电脑B发送一个数据包，后者收到了，回复一个数据包，从而实现两台电脑之间的通信。数据包的结构，基本上是下面这样：



发送这个包，需要知道两个地址：

- * 对方的MAC地址
- * 对方的IP地址

有了这两个地址，数据包才能准确送到接收者手中。但是，前面说过，MAC地址有局限性，如果两台电脑不在同一个子网络，就无法知道对方的MAC地址，必须通过网关（gateway）转发。

□

上图中，1号电脑要向4号电脑发送一个数据包。它先判断4号电脑是否在同一个子网络，结果发现不是（后文介绍判断方法），于是就把这个数据包发到网关A。网关A通过路由协议，发现4号电脑位于子网络B，又把数据包发给网关B，网关B再转发到4号电脑。

1号电脑把数据包发到网关A，必须知道网关A的MAC地址。所以，数据包的目标地址，实际上分成两种情况：

场景	数据包地址
同一个子网络	对方的MAC地址，对方的IP地址

非同一个子网络

网关的MAC地址，对方的IP地址

发送数据包之前，电脑必须判断对方是否在同一个子网络，然后选择相应的MAC地址。接下来，我们就来看，实际使用中，这个过程是怎么完成的。

八、用户的上网设置

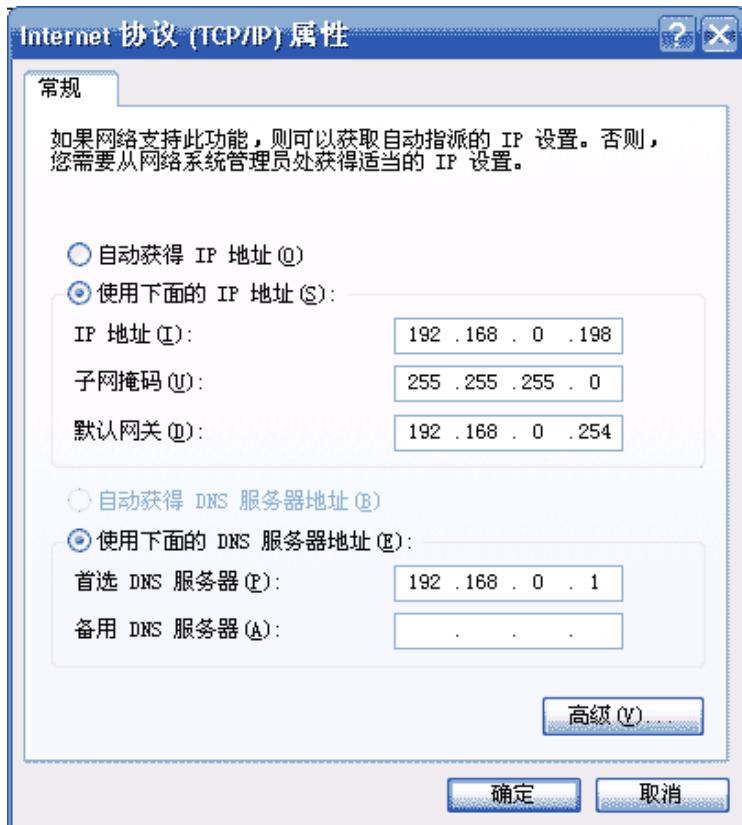
8.1 静态IP地址

你买了一台新电脑，插上网线，开机，这时电脑能够上网吗？

通常你必须做一些设置。有时，管理员（或者ISP）会告诉你下面四个参数，你把它们填入操作系统，计算机就能连上网了：

- * 本机的IP地址
- * 子网掩码
- * 网关的IP地址
- * DNS的IP地址

下图是Windows系统的设置窗口。



这四个参数缺一不可，后文会解释为什么需要知道它们才能上网。由于它们是给定的，计算机每次开机，都会分到同样的IP地址，所以这种情况被称作“静态IP地址上网”。

但是，这样的设置很专业，普通用户望而生畏，而且如果一台电脑的IP地址保持不变，其他电脑就不能使用这个地址，不够灵活。出于这两个原因，大多数用户使用“动态IP地址上网”。

8.2 动态IP地址

所谓“动态IP地址”，指计算机开机后，会自动分配到一个IP地址，不用人为设定。它使用的协议叫做[DHCP协议](#)。

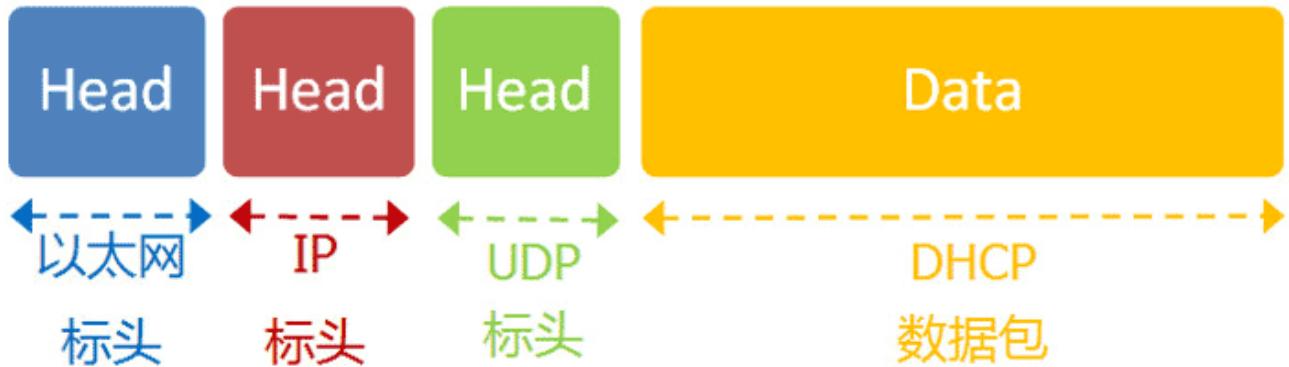
这个协议规定，每一个子网络中，有一台计算机负责管理本网络的所有IP地址，它叫做“DHCP服务器”。新的计算机加入网络，必须向“DHCP服务器”发送一个“DHCP请求”数据包，申请IP地址和相关的网络参数。

前面说过，如果两台计算机在同一个子网络，必须知道对方的MAC地址和IP地址，才能发送数据包。但是，新加入的计算机不知道这两个地址，怎么发送数据包呢？

DHCP协议做了一些巧妙的规定。

8.3 DHCP协议

首先，它是一种应用层协议，建立在UDP协议之上，所以整个数据包是这样的：



(1) 最前面的“以太网标头”，设置发出方（本机）的MAC地址和接收方（DHCP服务器）的MAC地址。前者就是本机网卡的MAC地址，后者这时不知道，就填入一个广播地址：FF-FF-FF-FF-FF-FF。

(2) 后面的“IP标头”，设置发出方的IP地址和接收方的IP地址。这时，对于这两者，本机都不知道。于是，发出方的IP地址就设为0.0.0.0，接收方的IP地址设为255.255.255.255。

(3) 最后的“UDP标头”，设置发出方的端口和接收方的端口。这一部分是DHCP协议规定好的，发出方是68端口，接收方是67端口。

这个数据包构造完成后，就可以发出了。以太网是广播发送，同一个子网络的每台计算机都收到了这个包。因为接收方的MAC地址是FF-FF-FF-FF-FF-FF，看不出是发给谁的，所以每台收到这个包的计算机，还必须分析这个包的IP地址，才能确定是不是发给自己的。当看到发出方IP地址是0.0.0.0，接收方是255.255.255.255，于是DHCP服务器知道“这个包是发给我的”，而其他计算机就可以丢弃这个包。

接下来，DHCP服务器读出这个包的数据内容，分配好IP地址，发送回去一个“DHCP响应”数据包。这个响应包的结构也是类似的，以太网标头的MAC地址是双方的网卡地址，IP标头的IP地址是DHCP服务器的IP地址（发出方）和255.255.255.255（接收方），UDP标头的端口是67（发出方）和68（接收方），分配给请求端的IP地址和本网络的具体参数则包含在Data部分。

新加入的计算机收到这个响应包，于是就知道了自己的IP地址、子网掩码、网关地址、DNS服务器等等参数。

8.4 上网设置：小结

这个部分，需要记住的就是一点：不管是“静态IP地址”还是“动态IP地址”，电脑上网的首要步骤，是确定四个参数。这四个值很重要，值得重复一遍：

- * 本机的IP地址
- * 子网掩码

- * 网关的IP地址
- * DNS的IP地址

有了这几个数值，电脑就可以上网“冲浪”了。接下来，我们来看一个实例，当用户访问网页的时候，互联网协议是怎么运作的。

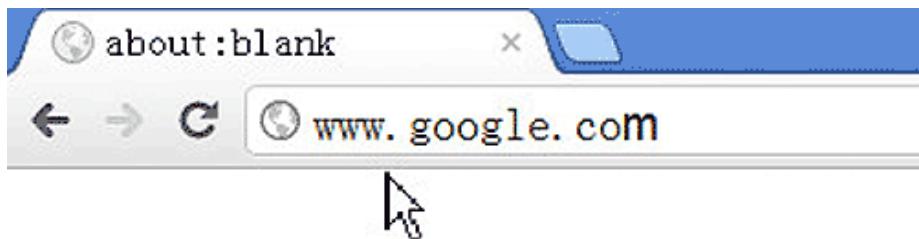
九、一个实例：访问网页

9.1 本机参数

我们假定，经过上一节的步骤，用户设置好了自己的网络参数：

- * 本机的IP地址：192.168.1.100
- * 子网掩码：255.255.255.0
- * 网关的IP地址：192.168.1.1
- * DNS的IP地址：8.8.8.8

然后他打开浏览器，想要访问Google，在地址栏输入了网址：www.google.com。

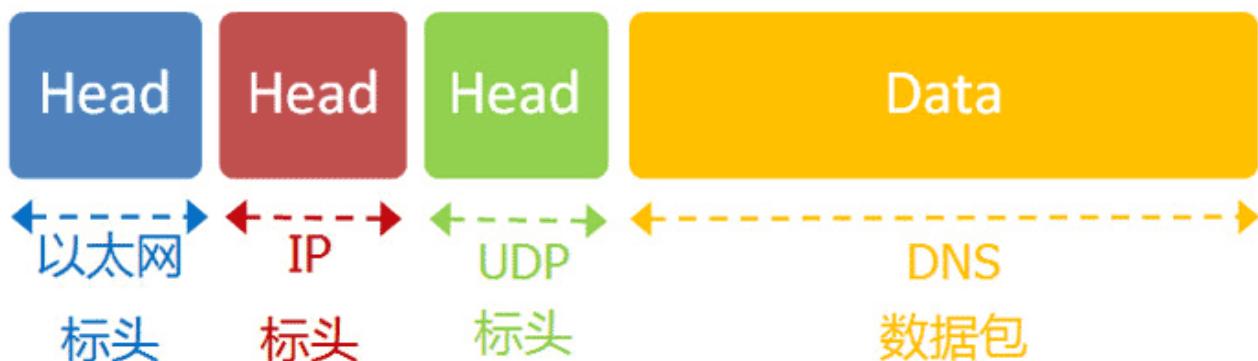


这意味着，浏览器要向Google发送一个网页请求的数据包。

9.2 DNS协议

我们知道，发送数据包，必须要知道对方的IP地址。但是，现在，我们只知道网址www.google.com，不知道它的IP地址。

[DNS协议](#)可以帮助我们，将这个网址转换成IP地址。已知DNS服务器为8.8.8.8，于是我们向这个地址发送一个DNS数据包（53端口）。



然后，DNS服务器做出响应，告诉我们Google的IP地址是172.194.72.105。于是，我们知道了对方的IP地址。

9.3 子网掩码

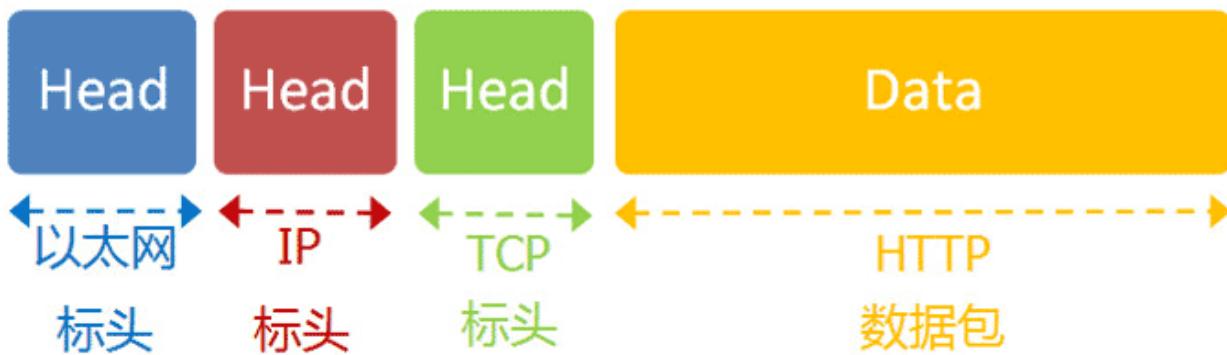
接下来，我们要判断，这个IP地址是不是在同一个子网络，这就要用到子网掩码。

已知子网掩码是255.255.255.0，本机用它对自己的IP地址192.168.1.100，做一个二进制的AND运算（两个数位都为1，结果为1，否则为0），计算结果为192.168.1.0；然后对Google的IP地址172.194.72.105也做一个AND运算，计算结果为172.194.72.0。这两个结果不相等，所以结论是，Google与本机不在同一个子网络。

因此，我们要向Google发送数据包，必须通过网关192.168.1.1转发，也就是说，接收方的MAC地址将是网关的MAC地址。

9.4 应用层协议

浏览网页用的是HTTP协议，它的整个数据包构造是这样的：



HTTP部分的内容，类似于下面这样：

```
GET / HTTP/1.1
Host: www.google.com
Connection: keep-alive
User-Agent: Mozilla/5.0 (Windows NT 6.1) .....
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: zh-CN,zh;q=0.8
Accept-Charset: GBK,utf-8;q=0.7,*;q=0.3
Cookie: ... ...
```

我们假定这个部分的长度为4960字节，它会被嵌在TCP数据包之中。

9.5 TCP协议

TCP数据包需要设置端口，接收方（Google）的HTTP端口默认是80，发送方（本机）的端口是一个随机生成的1024-65535之间的整数，假定为51775。

TCP数据包的标头长度为20字节，加上嵌入HTTP的数据包，总长度变为4980字节。

9.6 IP协议

然后，TCP数据包再嵌入IP数据包。IP数据包需要设置双方的IP地址，这是已知的，发送方是192.168.1.100（本

机），接收方是172.194.72.105（Google）。

IP数据包的标头长度为20字节，加上嵌入的TCP数据包，总长度变为5000字节。

9.7 以太网协议

最后，IP数据包嵌入以太网数据包。以太网数据包需要设置双方的MAC地址，发送方为本机的网卡MAC地址，接收方为网关192.168.1.1的MAC地址（通过ARP协议得到）。

以太网数据包的数据部分，最大长度为1500字节，而现在的IP数据包长度为5000字节。因此，IP数据包必须分割成四个包。因为每个包都有自己的IP标头（20字节），所以四个包的IP数据包的长度分别为1500、1500、1500、560。

□

9.8 服务器端响应

经过多个网关的转发，Google的服务器172.194.72.105，收到了这四个以太网数据包。

根据IP标头的序号，Google将四个包拼起来，取出完整的TCP数据包，然后读出里面的“HTTP请求”，接着做出“HTTP响应”，再用TCP协议发回来。

本机收到HTTP响应以后，就可以将网页显示出来，完成一次网络通信。

□

这个例子就到此为止，虽然经过了简化，但它大致上反映了互联网协议的整个通信过程。

（完）

互联网协议入门（一）

我们每天使用互联网，你是否想过，它是如何实现的？

全世界几十亿台电脑，连接在一起，两两通信。上海的某一块网卡送出信号，洛杉矶的另一块网卡居然就收到了，两者实际上根本不知道对方的物理位置，你不觉得这是很神奇的事情吗？

互联网的核心是一系列协议，总称为"互联网协议"（Internet Protocol Suite）。它们对电脑如何连接和组网，做出了详尽的规定。理解了这些协议，就理解了互联网的原理。

下面就是我的学习笔记。因为这些协议实在太复杂、太庞大，我想整理一个简洁的框架，帮助自己从总体上把握它们。为了保证简单易懂，我做了大量的简化，有些地方并不全面和精确，但是应该能够说清楚互联网的原理。

=====

互联网协议入门

作者：阮一峰



copyright © 2007 Bill Frymire

一、概述

1.1 五层模型

互联网的实现，分成好几层。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。

用户接触到的，只是最上面的一层，根本没有感觉到下面的层。要理解互联网，必须从最下层开始，自下而上理解每一层的功能。

如何分层有不同的模型，有的模型分七层，有的分四层。我觉得，把互联网分成五层，比较容易解释。

应用层 (Application Layer)

传输层 (Transport Layer)

网络层 (Network Layer)

链接层 (Link Layer)

实体层 (Physical Layer)

如上图所示，最底下的一层叫做“实体层”（Physical Layer），最上面的一层叫做“应用层”（Application Layer），中间的三层（自下而上）分别是“链接层”（Link Layer）、“网络层”（Network Layer）和“传输层”（Transport Layer）。越下面的层，越靠近硬件；越上面的层，越靠近用户。

它们叫什么名字，其实并不重要。只需要知道，互联网分成若干层就可以了。

1.2 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。

大家都遵守的规则，就叫做“协议”（protocol）。

互联网的每一层，都定义了很多协议。这些协议的总称，就叫做“互联网协议”（Internet Protocol Suite）。它们是互联网的核心，下面介绍每一层的功能，主要就是介绍每一层的主要协议。

二、实体层

我们从最底下的一层开始。

电脑要组网，第一件事要干什么？当然是先把电脑连起来，可以用光缆、电缆、双绞线、无线电波等方式。



这就叫做"实体层"，它就是把电脑连接起来的物理手段。它主要规定了网络的一些电气特性，作用是负责传送0和1的电信号。

三、链接层

3.1 定义

单纯的0和1没有任何意义，必须规定解读方式：多少个电信号算一组？每个信号位有何意义？

这就是"链接层"的功能，它在"实体层"的上方，确定了0和1的分组方式。

3.2 以太网协议

早期的时候，每家公司都有自己的电信号分组方式。逐渐地，一种叫做["以太网"](#)（Ethernet）的协议，占据了主导地位。

以太网规定，一组电信号构成一个数据包，叫做"帧"（Frame）。每一帧分成两个部分：标头（Head）和数据（Data）。



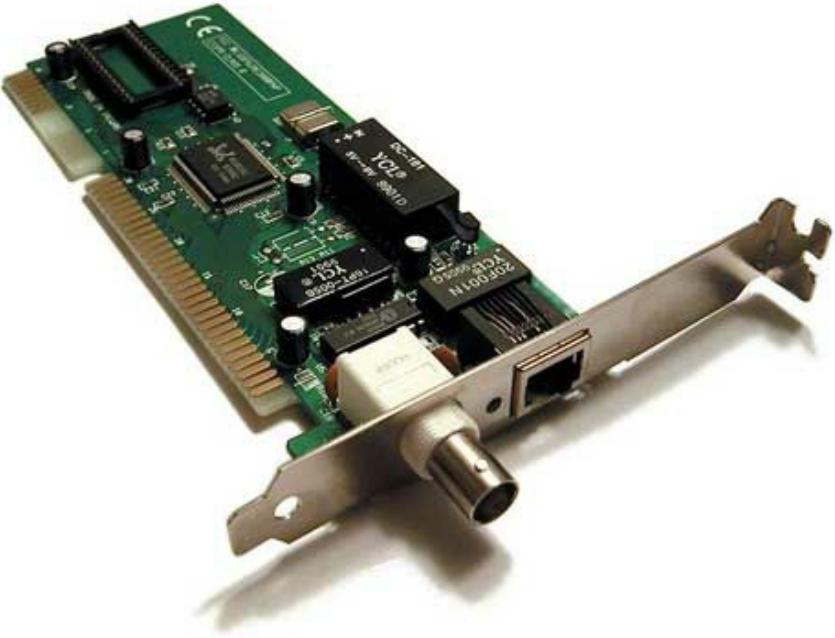
"标头"包含数据包的一些说明项，比如发送者、接受者、数据类型等等；"数据"则是数据包的具体内容。

"标头"的长度，固定为18字节。"数据"的长度，最短为46字节，最长为1500字节。因此，整个"帧"最短为64字节，最长为1518字节。如果数据很长，就必须分割成多个帧进行发送。

3.3 MAC地址

上面提到，以太网数据包的"标头"，包含了发送者和接受者的信息。那么，发送者和接受者是如何标识呢？

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。网卡的地址，就是数据包的发送地址和接收地址，这叫做MAC地址。



每块网卡出厂的时候，都有一个全世界独一无二的MAC地址，长度是48个二进制位，通常用12个十六进制数表示。

□

前6个十六进制数是厂商编号，后6个是该厂商的网卡流水号。有了MAC地址，就可以定位网卡和数据包的路径了。

3.4 广播

定义地址只是第一步，后面还有更多的步骤。

首先，一块网卡怎么会知道另一块网卡的MAC地址？

回答是有一种ARP协议，可以解决这个问题。这个留到后面介绍，这里只需要知道，以太网数据包必须知道接收方的MAC地址，然后才能发送。

其次，就算有了MAC地址，系统怎样才能把数据包准确送到接收方？

回答是以太网采用了一种很"原始"的方式，它不是把数据包准确送到接收方，而是向本网络内所有计算机发送，让每台计算机自己判断，是否为接收方。

□

上图中，1号计算机向2号计算机发送一个数据包，同一个子网络的3号、4号、5号计算机都会收到这个包。它们读取这个包的"标头"，找到接收方的MAC地址，然后与自身的MAC地址相比较，如果两者相同，就接受这个包，做进一步处理，否则就丢弃这个包。这种发送方式就叫做"广播"（broadcasting）。

有了数据包的定义、网卡的MAC地址、广播的发送方式，"链接层"就可以在多台计算机之间传送数据了。

四、网络层

4.1 网络层的由来

以太网协议，依靠MAC地址发送数据。理论上，单单依靠MAC地址，上海的网卡就可以找到洛杉矶的网卡了，技术上是可以实现的。

但是，这样做有一个重大的缺点。以太网采用广播方式发送数据包，所有成员人手一"包"，不仅效率低，而且局限在发送者所在的子网络。也就是说，如果两台计算机不在同一个子网络，广播是传不过去的。这种设计是合理的，否则互联网上每一台计算机都会收到所有包，那会引起灾难。

互联网是无数子网络共同组成的一个巨型网络，很像想象上海和洛杉矶的电脑会在同一个子网络，这几乎是不可能的。

因此，必须找到一种方法，能够区分哪些MAC地址属于同一个子网络，哪些不是。如果是同一个子网络，就采用广播方式发送，否则就采用"路由"方式发送。（"路由"的意思，就是指如何向不同的子网络分发数据包，这是一个很大的主题，本文不涉及。）遗憾的是，MAC地址本身无法做到这一点。它只与厂商有关，与所处网络无关。

这就导致了"网络层"的诞生。它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"，简称"网址"。

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。

4.2 IP协议

规定网络地址的协议，叫做IP协议。它所定义的地址，就被称为IP地址。

目前，广泛采用的是IP协议第四版，简称IPv4。这个版本规定，网络地址由32个二进制位组成。

习惯上，我们用分成四段的十进制数表示IP地址，从0.0.0.0一直到255.255.255.255。

互联网上的每一台计算机，都会分配到一个IP地址。这个地址分成两个部分，前一部分代表网络，后一部分代表主机。比如，IP地址172.16.254.1，这是一个32位的地址，假定它的网络部分是前24位（172.16.254），那么主机部分就是后8位（最后的那个1）。处于同一个子网络的电脑，它们IP地址的网络部分必定是相同的，也就是说172.16.254.2应该与172.16.254.1处在同一个子网络。

但是，问题在于单单从IP地址，我们无法判断网络部分。还是以172.16.254.1为例，它的网络部分，到底是前24位，还是前16位，甚至前28位，从IP地址上是看不出来的。

那么，怎样才能从IP地址，判断两台计算机是否属于同一个子网络呢？这就要用到另一个参数"子网掩码"（subnet mask）。

所谓"子网掩码"，就是表示子网络特征的一个参数。它在形式上等同于IP地址，也是一个32位二进制数字，它的网络部分全部为1，主机部分全部为0。比如，IP地址172.16.254.1，如果已知网络部分是前24位，主机部分是后8位，那么子网络掩码就是11111111.11111111.11111111.00000000，写成十进制就是255.255.255.0。

知道"子网掩码"，我们就能判断，任意两个IP地址是否处在同一个子网络。方法是将两个IP地址与子网掩码分别进行AND运算（两个数位都为1，运算结果为1，否则为0），然后比较结果是否相同，如果是的话，就表明它们在同一个子网络中，否则就不是。

比如，已知IP地址172.16.254.1和172.16.254.233的子网掩码都是255.255.255.0，请问它们是否在同一个子网络？两者与子网掩码分别进行AND运算，结果都是172.16.254.0，因此它们在同一个子网络。

总结一下，IP协议的作用主要有两个，一个是为每一台计算机分配IP地址，另一个是确定哪些地址在同一个子网络。

4.3 IP数据包

根据IP协议发送的数据，就叫做IP数据包。不难想象，其中必定包括IP地址信息。

但是前面说过，以太网数据包只包含MAC地址，并没有IP地址的栏位。那么是否需要修改数据定义，再添加一个栏位呢？

回答是不需要，我们可以把IP数据包直接放进以太网数据包的“数据”部分，因此完全不用修改以太网的规格。这就是互联网分层结构的好处：上层的变动完全不涉及下层的结构。

具体来说，IP数据包也分为“标头”和“数据”两个部分。

□

“标头”部分主要包括版本、长度、IP地址等信息，“数据”部分则是IP数据包的具体内容。它放进以太网数据包后，以太网数据包就变成了下面这样。

□

IP数据包的“标头”部分的长度为20到60字节，整个数据包的总长度最大为65,535字节。因此，理论上，一个IP数据包的“数据”部分，最长为65,515字节。前面说过，以太网数据包的“数据”部分，最长只有1500字节。因此，如果IP数据包超过了1500字节，它就需要分割成几个以太网数据包，分开发送了。

4.4 ARP协议

关于“网络层”，还有最后一点需要说明。

因为IP数据包是放在以太网数据包里发送的，所以我们必须同时知道两个地址，一个是对方的MAC地址，另一个是对方的IP地址。通常情况下，对方的IP地址是已知的（后文会解释），但是我们不知道它的MAC地址。

所以，我们需要一种机制，能够从IP地址得到MAC地址。

这里又可以分成两种情况。第一种情况，如果两台主机不在同一个子网络，那么事实上没有办法得到对方的MAC地址，只能把数据包传送到两个子网络连接处的“网关”（gateway），让网关去处理。

第二种情况，如果两台主机在同一个子网络，那么我们可以用ARP协议，得到对方的MAC地址。ARP协议也是发出一个数据包（包含在以太网数据包中），其中包含它所要查询主机的IP地址，在对方的MAC地址这一栏，填的是FF:FF:FF:FF:FF:FF，表示这是一个“广播”地址。它所在子网络的每一台主机，都会收到这个数据包，从中取出IP地址，与自身的IP地址进行比较。如果两者相同，都做出回复，向对方报告自己的MAC地址，否则就丢弃这个包。

总之，有了ARP协议之后，我们就可以得到同一个子网络内的主机MAC地址，可以把数据包发送到任意一台主机上了。

五、传输层

5.1 传输层的由来

有了MAC地址和IP地址，我们已经可以在互联网上任意两台主机上建立通信。

接下来的问题是，同一台主机上有许多程序都需要用到网络，比如，你一边浏览网页，一边与朋友在线聊天。当一个数据包从互联网上发来的时候，你怎么知道，它是表示网页的内容，还是表示在线聊天的内容？

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"是0到65535之间的一个整数，正好16个二进制位。0到1023的端口被系统占用，用户只能选用大于1023的端口。不管是浏览网页还是在线聊天，应用程序会随机选用一个端口，然后与服务器的相应端口联系。

"传输层"的功能，就是建立"端口到端口"的通信。相比之下，"网络层"的功能是建立"主机到主机"的通信。只要确定主机和端口，我们就能实现程序之间的交流。因此，Unix系统就把主机+端口，叫做"套接字"（socket）。有了它，就可以进行网络应用程序开发了。

5.2 UDP协议

现在，我们必须在数据包中加入端口信息，这就需要新的协议。最简单的实现叫做UDP协议，它的格式几乎就是在数据前面，加上端口号。

UDP数据包，也是由"标头"和"数据"两部分组成。

□

"标头"部分主要定义了发出端口和接收端口，"数据"部分就是具体的内容。然后，把整个UDP数据包放入IP数据包的"数据"部分，而前面说过，IP数据包又是放在以太网数据包之中的，所以整个以太网数据包现在变成了下面这样：



UDP数据包非常简单，"标头"部分一共只有8个字节，总长度不超过65,535字节，正好放进一个IP数据包。

5.3 TCP协议

UDP协议的优点是比较简单，容易实现，但是缺点是可靠性较差，一旦数据包发出，无法知道对方是否收到。

为了解决这个问题，提高网络可靠性，TCP协议就诞生了。这个协议非常复杂，但可以近似认为，它就是有确认机制的UDP协议，每发出一个数据包都要求确认。如果有一个数据包遗失，就收不到确认，发出方就知道有必要重发这个数据包了。

因此，TCP协议能够确保数据不会遗失。它的缺点是过程复杂、实现困难、消耗较多的资源。

TCP数据包和UDP数据包一样，都是内嵌在IP数据包的"数据"部分。TCP数据包没有长度限制，理论上可以无限长，但是为了保证网络的效率，通常TCP数据包的长度不会超过IP数据包的长度，以确保单个TCP数据包不必再分割。

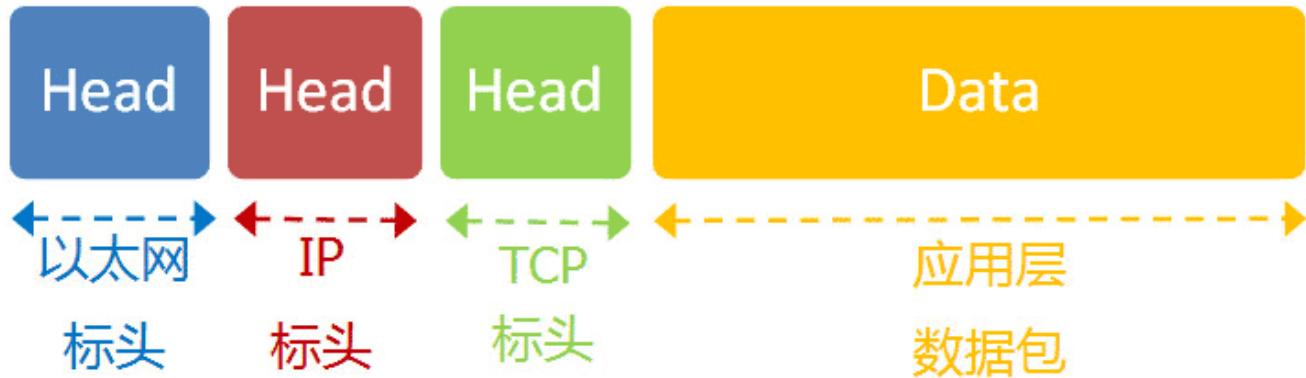
六、应用层

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。

举例来说，TCP协议可以为各种各样的程序传递数据，比如Email、WWW、FTP等等。那么，必须有不同协议规定电子邮件、网页、FTP数据的格式，这些应用程序协议就构成了"应用层"。

这是最高的一层，直接面对用户。它的数据就放在TCP数据包的"数据"部分。因此，现在的以太网的数据包就变成下面这样。



至此，整个互联网的五层结构，自下而上全部讲完了。这是从系统的角度，解释互联网是如何构成的。[下一篇](#)，我反过来，从用户的角度，自上而下看看这个结构是如何发挥作用的，完成一次网络数据交换的。

(完)

函数式编程初探

诞生50多年之后，[函数式编程](#)（functional programming）开始获得越来越多的关注。

不仅最古老的函数式语言Lisp重获青春，而且新的函数式语言层出不穷，比如Erlang、clojure、Scala、F#等等。目前最当红的Python、Ruby、Javascript，对函数式编程的支持都很强，就连老牌的面向对象的Java、面向过程的PHP，都忙不迭地加入对匿名函数的支持。越来越多的迹象表明，函数式编程已经不再是学术界的最爱，开始大踏步地在业界投入实用。

也许继“面向对象编程”之后，“函数式编程”会成为下一个编程的主流范式（paradigm）。未来的程序员恐怕或多或少都必须懂一点。

□

但是，“函数式编程”看上去比较难，缺乏通俗的入门教程，各种介绍文章都充斥着数学符号和专用术语，让人读了如坠云雾。就连最基本的问题“什么是函数式编程”，网上都搜不到易懂的回答。

下面是我的“函数式编程”学习笔记，分享出来，与大家一起探讨。内容不涉及数学（我也不懂[Lambda Calculus](#)），也不涉及高级特性（比如[lazy evaluation](#)和[currying](#)），只求尽量简单通俗地整理和表达，我现在所理解的“函数式编程”以及它的意义。

我主要参考了Slava Akhmechet的[*Functional Programming For The Rest of Us*](#)。

一、定义

简单说，“函数式编程”是一种[“编程范式”](#)（programming paradigm），也就是如何编写程序的方法论。

它属于[“结构化编程”](#)的一种，主要思想是把运算过程尽量写成一系列嵌套的函数调用。举例来说，现在有这样一个数学表达式：

$$(1 + 2) * 3 - 4$$

传统的过程式编程，可能这样写：

```
var a = 1 + 2;  
var b = a * 3;  
var c = b - 4;
```

函数式编程要求使用函数，我们可以把运算过程[定义](#)为不同的函数，然后写成下面这样：

```
var result = subtract(multiply(add(1,2), 3), 4);
```

这就是函数式编程。

二、特点

函数式编程具有五个鲜明的特点。

1. 函数是"第一等公民"

所谓"第一等公民" (first class) , 指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

举例来说，下面代码中的print变量就是一个函数，可以作为另一个函数的参数。

```
var print = function(i){ console.log(i);}  
[1,2,3].forEach(print);
```

2. 只用"表达式"，不用"语句"

"表达式" (expression) 是一个单纯的运算过程，总是有返回值；"语句" (statement) 是执行某种操作，没有返回值。函数式编程要求，只使用表达式，不使用语句。也就是说，每一步都是单纯的运算，而且都有返回值。

原因是函数式编程的发动机，一开始就是为了处理运算 (computation) ，不考虑系统的读写 (I/O) 。"语句"属于对系统的读写操作，所以就被排斥在外。

当然，实际应用中，不做I/O是不可能的。因此，编程过程中，函数式编程只要求把I/O限制到最小，不要有不必要的读写行为，保持计算过程的单纯性。

3. 没有"副作用"

所谓"副作用" (side effect) , 指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

4. 不修改状态

上一点已经提到，函数式编程只是返回新的值，不修改系统变量。因此，不修改变量，也是它的一个重要特点。

在其他类型的语言中，变量往往用来保存"状态" (state) 。不修改变量，意味着状态不能保存在变量中。函数式编程使用参数保存状态，最好的例子就是递归。下面的代码是一个将字符串逆序排列的函数，它演示了不同的参数如何决定了运算所处的"状态"。

```
function reverse(string) {  
    if(string.length == 0) {  
        return string;  
    } else {
```

```
        return reverse(string.substring(1, string.length)) + string.substring(0, 1);  
    }  
}
```

由于使用了递归，函数式语言的运行速度比较慢，这是它长期不能在业界推广的主要原因。

5. 引用透明

引用透明（Referential transparency），指的是函数的运行不依赖于外部变量或"状态"，只依赖于输入的参数，任何时候只要参数相同，引用函数所得到的返回值总是相同的。

有了前面的第三点和第四点，这点是很显然的。其他类型的语言，函数的返回值往往与系统状态有关，不同的状态之下，返回值是不一样的。这就叫"引用不透明"，很不利于观察和理解程序的行为。

三、意义

函数式编程到底有什么好处，为什么会变得越来越流行？

1. 代码简洁，开发快速

函数式编程大量使用函数，减少了代码的重复，因此程序比较短，开发速度较快。

Paul Graham在[《黑客与画家》](#)一书中写道：同样功能的程序，极端情况下，Lisp代码的长度可能是C代码的二十分之一。

如果程序员每天所写的代码行数基本相同，这就意味着，"C语言需要一年时间完成开发某个功能，Lisp语言只需要不到三星期。反过来说，如果某个新功能，Lisp语言完成开发需要三个月，C语言需要写五年。"当然，这样的对比故意夸大了差异，但是"在一个高度竞争的市场中，即使开发速度只相差两三倍，也足以使得你永远处在落后的位置。"

2. 接近自然语言，易于理解

函数式编程的自由度很高，可以写出很接近自然语言的代码。

前文曾经将表达式 $(1 + 2) * 3 - 4$ ，写成函数式语言：

```
subtract(multiply(add(1,2), 3), 4)
```

对它进行变形，不难得到另一种写法：

```
add(1,2).multiply(3).subtract(4)
```

这基本就是自然语言的表达了。再看下面的代码，大家应该一眼就能明白它的意思吧：

```
merge([1,2],[3,4]).sort().search("2")
```

因此，函数式编程的代码更容易理解。

3. 更方便的代码管理

函数式编程不依赖、也不会改变外界的状态，只要给定输入参数，返回的结果必定相同。因此，每一个函数都可以被看做独立单元，很有利于进行单元测试（unit testing）和除错（debugging），以及模块化组合。

4. 易于“并发编程”

函数式编程不需要考虑“死锁”（deadlock），因为它不修改变量，所以根本不存在“锁”线程的问题。不必担心一个线程的数据，被另一个线程修改，所以可以很放心地把工作分摊到多个线程，部署“并发编程”（concurrency）。

请看下面的代码：

```
var s1 = Op1();
var s2 = Op2();
var s3 = concat(s1, s2);
```

由于s1和s2互不干扰，不会修改变量，谁先执行是无所谓的，所以可以放心地增加线程，把它们分配在两个线程上完成。其他类型的语言就做不到这一点，因为s1可能会修改系统状态，而s2可能会用到这些状态，所以必须保证s2在s1之后运行，自然也就不能部署到其他线程上了。

多核CPU是将来的潮流，所以函数式编程的这个特性非常重要。

5. 代码的热升级

函数式编程没有副作用，只要保证接口不变，内部实现是外部无关的。所以，可以在运行状态下直接升级代码，不需要重启，也不需要停机。[Erlang](#)语言早就证明了这一点，它是瑞典爱立信公司为了管理电话系统而开发的，电话系统的升级当然是不能停机的。

（完）

Unix目录结构的来历

Unix（包含Linux）的初学者，常常会很困惑，不明白目录结构的含义何在。

举例来说，根目录下面有一个子目录/bin，用于存放二进制程序。但是，/usr子目录下面还有/usr/bin，以及/usr/local/bin，也用于存放二进制程序；某些系统甚至还有/opt/bin。它们有何区别？

长久以来，我也感到很费解，不明白为什么这样设计。像大多数人一样，我只是根据[《Unix文件系统结构标准》](#)（Filesystem Hierarchy Standard），死记硬背不同目录的区别。

昨天，我读到了Rob Landley的[简短解释](#)，这才恍然大悟，原来Unix目录结构是历史造成的。

话说1969年，[Ken Thompson](#)和[Dennis Ritchie](#)在小型机PDP-7上发明了Unix。1971年，他们将主机升级到了PDP-11。

当时，他们使用一种叫做RK05的储存盘，一盘的容量大约是1.5MB。

没过多久，操作系统（根目录）变得越来越大，一块盘已经装不下了。于是，他们加上了第二盘RK05，并且规定第一块盘专门放系统程序，第二块盘专门放用户自己的程序，因此挂载的目录点取名为/usr。也就是说，根目录"/"挂载在第一块盘，"/usr"目录挂载在第二块盘。除此之外，两块盘的目录结构完全相同，第一块盘的目录（/bin, /sbin, /lib, /tmp...）都在/usr目录下重新出现一次。

后来，第二块盘也满了，他们只好又加了第三盘RK05，挂载的目录点取名为/home，并且规定/usr用于存放用户的程序，/home用于存放用户的数据。

从此，这种目录结构就延续了下来。随着硬盘容量越来越大，各个目录的含义进一步得到明确。

/：存放系统程序，也就是At&t开发的Unix程序。

/usr：存放Unix系统商（比如IBM和HP）开发的程序。

/usr/local：存放用户自己安装的程序。

/opt：在某些系统，用于存放第三方厂商开发的程序，所以取名为option，意为"选装"。

（完）

SSH原理与运用（二）：远程操作与端口转发

接着前一次的[文章](#)，继续介绍SSH的用法。

=====

SSH原理与运用（二）：远程操作与端口转发

作者：阮一峰

□

(Image credit: [Tony Narlock](#))

七、远程操作

SSH不仅可以用于远程主机登录，还可以直接在远程主机上执行操作。

上一节的操作，就是一个例子：

```
$ ssh user@host 'mkdir -p .ssh && cat >> .ssh/authorized_keys' < ~/.ssh/id_rsa.pub
```

单引号中间的部分，表示在远程主机上执行的操作；后面的输入重定向，表示数据通过SSH传向远程主机。

这就是说，SSH可以在用户和远程主机之间，建立命令和数据的传输通道，因此很多事情都可以通过SSH来完成。

下面看几个例子。

【例1】

将\$HOME/src/目录下面的所有文件，复制到远程主机的\$HOME/src/目录。

```
$ cd && tar czv src | ssh user@host 'tar xz'
```

【例2】

将远程主机\$HOME/src/目录下面的所有文件，复制到用户的当前目录。

```
$ ssh user@host 'tar cz src' | tar xzv
```

【例3】

查看远程主机是否运行进程httpd。

```
$ ssh user@host 'ps ax | grep [h]ttpd'
```

八、绑定本地端口

既然SSH可以传送数据，那么我们可以让那些不加密的网络连接，全部改走SSH连接，从而提高安全性。

假定我们要让8080端口的数据，都通过SSH传向远程主机，命令就这样写：

```
$ ssh -D 8080 user@host
```

SSH会建立一个socket，去监听本地的8080端口。一旦有数据传向那个端口，就自动把它转移到SSH连接上面，发往远程主机。可以想象，如果8080端口原来是一个不加密端口，现在将变成一个加密端口。

九、本地端口转发

有时，绑定本地端口还不够，还必须指定数据传送的目标主机，从而形成点对点的"端口转发"。为了区别后文的"远程端口转发"，我们把这种情况称为"本地端口转发"（Local forwarding）。

假定host1是本地主机，host2是远程主机。由于种种原因，这两台主机之间无法连通。但是，另外还有一台host3，可以同时连通前面两台主机。因此，很自然的想法就是，通过host3，将host1连上host2。

我们在host1执行下面的命令：

```
$ ssh -L 2121:host2:21 host3
```

命令中的L参数一共接受三个值，分别是"本地端口:目标主机:目标主机端口"，它们之间用冒号分隔。这条命令的意思，就是指定SSH绑定本地端口2121，然后指定host3将所有的数据，转发到目标主机host2的21端口（假定host2运行FTP，默认端口为21）。

这样一来，我们只要连接host1的2121端口，就等于连上了host2的21端口。

```
$ ftp localhost:2121
```

"本地端口转发"使得host1和host3之间仿佛形成一个数据传输的秘密隧道，因此又被称为"SSH隧道"。

下面是一个比较有趣的例子。

```
$ ssh -L 5900:localhost:5900 host3
```

它表示将本机的5900端口绑定host3的5900端口（这里的localhost指的是host3，因为目标主机是相对host3而言的）。

另一个例子是通过host3的端口转发，ssh登录host2。

```
$ ssh -L 9001:host2:22 host3
```

这时，只要ssh登录本机的9001端口，就相当于登录host2了。

```
$ ssh -p 9001 localhost
```

上面的-p参数表示指定登录端口。

十、远程端口转发

既然"本地端口转发"是指绑定本地端口的转发，那么"远程端口转发"（remote forwarding）当然是指绑定远程端口的转发。

还是接着看上面那个例子，host1与host2之间无法连通，必须借助host3转发。但是，特殊情况出现了，host3是一台内网机器，它可以连接外网的host1，但是反过来就不行，外网的host1连不上内网的host3。这时，"本地端口转发"就不能用了，怎么办？

解决办法是，既然host3可以连host1，那么就从host3上建立与host1的SSH连接，然后在host1上使用这条连接就可以了。

我们在host3执行下面的命令：

```
$ ssh -R 2121:host2:21 host1
```

R参数也是接受三个值，分别是"远程主机端口:目标主机:目标主机端口"。这条命令的意思，就是让host1监听它自己的2121端口，然后将所有数据经由host3，转发到host2的21端口。由于对于host3来说，host1是远程主机，所以这种情况就被称为"远程端口绑定"。

绑定之后，我们在host1就可以连接host2了：

```
$ ftp localhost:2121
```

这里必须指出，"远程端口转发"的前提条件是，host1和host3两台主机都有sshD和ssh客户端。

十一、SSH的其他参数

SSH还有一些别的参数，也值得介绍。

N参数，表示只连接远程主机，不打开远程shell；T参数，表示不为这个连接分配TTY。这两个参数可以放在一起用，代表这个SSH连接只用来传数据，不执行远程操作。

```
$ ssh -NT -D 8080 host
```

f参数，表示SSH连接成功后，转入后台运行。这样一来，你就可以在不中断SSH连接的情况下，在本地shell中执行其他操作。

```
$ ssh -f -D 8080 host
```

要关闭这个后台连接，就只有用kill命令去杀掉进程。

十二、参考文献

* SSH, The Secure Shell: The Definitive Guide:[2.4. Authentication by Cryptographic Key](#), O'reilly

* SSH, The Secure Shell: The Definitive Guide:[9.2. Port Forwarding](#), O'reilly

* Shebang: [Tips for Remote Unix Work \(SSH, screen, and VNC\)](#)

* brihatch: [SSH Host Key Protection](#)

* brihatch: [SSH User Identities](#)

* IBM developerWorks: [实战 SSH 端口转发](#)

* Jianing YANG: [ssh隧道技术简介](#)

* WikiBooks: [Internet Technologies/SSH](#)

* Buddhika Chamith: [SSH Tunneling Explained](#)

(完)

SSH原理与运用（一）：远程登录

SSH是每一台Linux电脑的标准配置。

随着Linux设备从电脑逐渐扩展到手机、外设和家用电器，SSH的使用范围也越来越广。不仅程序员离不开它，很多普通用户也每天使用。

SSH具备多种功能，可以用于很多场合。有些事情，没有它就是办不成。本文是我的学习笔记，总结和解释了SSH的常见用法，希望对大家有用。

虽然本文内容只涉及初级应用，较为简单，但是需要读者具备最基本的"Shell知识"和了解"公钥加密"的概念。如果你对它们不熟悉，我推荐先阅读[《UNIX / Linux 初学者教程》](#)和[《数字签名是什么？》](#)。

=====

SSH原理与运用

作者：阮一峰

□

一、什么是SSH？

简单说，SSH是一种网络协议，用于计算机之间的加密登录。

如果一个用户从本地计算机，使用SSH协议登录另一台远程计算机，我们就可以认为，这种登录是安全的，即使被中途截获，密码也不会泄露。

最早的时候，互联网通信都是明文通信，一旦被截获，内容就暴露无疑。1995年，芬兰学者Tatu Ylonen设计了SSH协议，将登录信息全部加密，成为互联网安全的一个基本解决方案，迅速在全世界获得推广，目前已经成为Linux系统的标准配置。

需要指出的是，SSH只是一种协议，存在多种实现，既有商业实现，也有开源实现。本文针对的实现是[OpenSSH](#)，它是自由软件，应用非常广泛。

此外，本文只讨论SSH在Linux Shell中的用法。如果要在Windows系统中使用SSH，会用到另一种软件[PuTTY](#)，这需要另文介绍。

二、最基本的用法

SSH主要用于远程登录。假定你要以用户名user，登录远程主机host，只要一条简单命令就可以了。

```
$ ssh user@host
```

如果本地用户名与远程用户名一致，登录时可以省略用户名。

```
$ ssh host
```

SSH的默认端口是22，也就是说，你的登录请求会送进远程主机的22端口。使用p参数，可以修改这个端口。

```
$ ssh -p 2222 user@host
```

上面这条命令表示，ssh直接连接远程主机的2222端口。

三、中间人攻击

SSH之所以能够保证安全，原因在于它采用了公钥加密。

整个过程是这样的：（1）远程主机收到用户的登录请求，把自己的公钥发给用户。（2）用户使用这个公钥，将登录密码加密后，发送回来。（3）远程主机用自己的私钥，解密登录密码，如果密码正确，就同意用户登录。

这个过程本身是安全的，但是实施的时候存在一个风险：如果有人截获了登录请求，然后冒充远程主机，将伪造的公钥发给用户，那么用户很难辨别真伪。因为不像https协议，SSH协议的公钥是没有证书中心（CA）公证的，也就是说，都是自己签发的。

可以设想，如果攻击者插在用户与远程主机之间（比如在公共的wifi区域），用伪造的公钥，获取用户的登录密码。再用这个密码登录远程主机，那么SSH的安全机制就荡然无存了。这种风险就是著名的“[中间人攻击](#)”（Man-in-the-middle attack）。

SSH协议是如何应对的呢？

四、口令登录

如果你是第一次登录对方主机，系统会出现下面的提示：

```
$ ssh user@host
```

```
The authenticity of host 'host (12.18.429.21)' can't be established.
```

```
RSA key fingerprint is 98:2e:d7:e0:de:9f:ac:67:28:c2:42:2d:37:16:58:4d.
```

```
Are you sure you want to continue connecting (yes/no)?
```

这段话的意思是，无法确认host主机的真实性，只知道它的公钥指纹，问你还想继续连接吗？

所谓“公钥指纹”，是指公钥长度较长（这里采用RSA算法，长达1024位），很难比对，所以对其进行MD5计算，将它变成一个128位的指纹。上例中是
98:2e:d7:e0:de:9f:ac:67:28:c2:42:2d:37:16:58:4d，再进行比较，就容易多了。

很自然的一个问题就是，用户怎么知道远程主机的公钥指纹应该是多少？回答是没有好办法，远程主机必须在自己的网站上贴出公钥指纹，以便用户自行核对。

假定经过风险衡量以后，用户决定接受这个远程主机的公钥。

```
Are you sure you want to continue connecting (yes/no)? yes
```

系统会出现一句提示，表示host主机已经得到认可。

```
Warning: Permanently added 'host,12.18.429.21' (RSA) to the list of known hosts.
```

然后，会要求输入密码。

```
Password: (enter password)
```

如果密码正确，就可以登录了。

当远程主机的公钥被接受以后，它就会被保存在文件\$HOME/.ssh/known_hosts之中。下次再连接这台主机，系统就会认出它的公钥已经保存在本地了，从而跳过警告部分，直接提示输入密码。

每个SSH用户都有自己的known_hosts文件，此外系统也有一个这样的文件，通常是在/etc/ssh/ssh_known_hosts，保存一些对所有用户都可信赖的远程主机的公钥。

五、公钥登录

使用密码登录，每次都必须输入密码，非常麻烦。好在SSH还提供了公钥登录，可以省去输入密码的步骤。

所谓“公钥登录”，原理很简单，就是用户将自己的公钥储存在远程主机上。登录的时候，远程主机会向用户发送一段随机字符串，用户用自己的私钥加密后，再发回来。远程主机用事先储存的公钥进行解密，如果成功，就证明用户是可信的，直接允许登录shell，不再要求密码。

这种方法要求用户必须提供自己的公钥。如果没有现成的，可以直接用ssh-keygen生成一个：

```
$ ssh-keygen
```

运行上面的命令以后，系统会出现一系列提示，可以一路回车。其中有一个问题是，要不要对私钥设置口令（passphrase），如果担心私钥的安全，这里可以设置一个。

运行结束以后，在\$HOME/.ssh/目录下，会新生成两个文件：id_rsa.pub和id_rsa。前者是你的公钥，后者是你的私钥。

这时再输入下面的命令，将公钥传送到远程主机host上面：

```
$ ssh-copy-id user@host
```

好了，从此你再登录，就不需要输入密码了。

如果还是不行，就打开远程主机的/etc/ssh/sshd_config这个文件，检查下面几行前面“#”注释是否去掉。

```
RSAAuthentication yes  
PubkeyAuthentication yes  
AuthorizedKeysFile .ssh/authorized_keys
```

然后，重启远程主机的ssh服务。

```
// ubuntu系统  
service ssh restart  
  
// debian系统  
/etc/init.d/ssh restart
```

六、authorized_keys文件

远程主机将用户的公钥，保存在登录后的用户主目录的\$HOME/.ssh/authorized_keys文件中。公钥就是一段字符串，只要把它追加在authorized_keys文件的末尾就行了。

这里不使用上面的ssh-copy-id命令，改用下面的命令，解释公钥的保存过程：

```
$ ssh user@host 'mkdir -p .ssh && cat >> .ssh/authorized_keys' < ~/.ssh/id_rsa.pub
```

这条命令由多个语句组成，依次分解来看：（1）"\$ ssh user@host"，表示登录远程主机；（2）单引号中的mkdir .ssh && cat >> .ssh/authorized_keys，表示登录后在远程shell上执行的命令：（3）"\$ mkdir -p .ssh"的作用是，如果用户主目录中的.ssh目录不存在，就创建一个；（4）'cat >> .ssh/authorized_keys' < ~/.ssh/id_rsa.pub的作用是，将本地的公钥文件~/.ssh/id_rsa.pub，重定向追加到远程文件authorized_keys的末尾。

写入authorized_keys文件后，公钥登录的设置就完成了。

=====

关于shell远程登录的部分就写到这里，下一次接着介绍[《远程操作和端口转发》](#)。

(完)

理解inode

[inode](#)是一个重要概念，是理解Unix/Linux文件系统和硬盘储存的基础。

我觉得，理解inode，不仅有助于提高系统操作水平，还有助于体会Unix设计哲学，即如何把底层的复杂性抽象成一个简单概念，从而大大简化用户接口。

下面就是我的inode学习笔记，尽量保持简单。

=====

理解inode

作者：阮一峰

□

一、 inode是什么？

理解inode，要从文件储存说起。

文件储存在硬盘上，硬盘的最小存储单位叫做"扇区"（Sector）。每个扇区储存512字节（相当于0.5KB）。

操作系统读取硬盘的时候，不会一个个扇区地读取，这样效率太低，而是一次性连续读取多个扇区，即一次性读取一个"块"（block）。这种由多个扇区组成的"块"，是文件存取的最小单位。"块"的大小，最常见的是4KB，即连续八个sector组成一个block。

文件数据都储存在"块"中，那么很显然，我们还必须找到一个地方储存文件的元信息，比如文件的创建者、文件的创建日期、文件的大小等等。这种储存文件元信息的区域就叫做inode，中文译名为"索引节点"。

每一个文件都有对应的inode，里面包含了与该文件有关的一些信息。

二、 inode的内容

inode包含文件的元信息，具体来说有以下内容：

- * 文件的字节数
- * 文件拥有者的User ID
- * 文件的Group ID
- * 文件的读、写、执行权限
- * 文件的时间戳，共有三个：ctime指inode上一次变动的时间，mtime指文件内容上一次变动的时间，atime指文件上一次打开的时间。
- * 链接数，即有多少文件名指向这个inode

* 文件数据block的位置

可以用stat命令，查看某个文件的inode信息：

```
stat example.txt
```

□

总之，除了文件名以外的所有文件信息，都存在inode之中。至于为什么没有文件名，下文会有详细解释。

三、inode的大小

inode也会消耗硬盘空间，所以硬盘格式化的时候，操作系统自动将硬盘分成两个区域。一个是数据区，存放文件数据；另一个是inode区（inode table），存放inode所包含的信息。

每个inode节点的大小，一般是128字节或256字节。inode节点的总数，在格式化时就给定，一般是每1KB或每2KB就设置一个inode。假定在一块1GB的硬盘中，每个inode节点的大小为128字节，每1KB就设置一个inode，那么inode table的大小就会达到128MB，占整块硬盘的12.8%。

查看每个硬盘分区的inode总数和已经使用的数量，可以使用df命令。

```
df -i
```

□

查看每个inode节点的大小，可以用如下命令：

```
sudo dumpe2fs -h /dev/hda | grep "Inode size"
```

```
$ dumpe2fs -h /dev/sda1 |grep "Inode size"  
dumpe2fs 1.41.14 (22-Dec-2010)  
Inode size: 256  
$ █
```

由于每个文件都必须有一个inode，因此有可能发生inode已经用光，但是硬盘还未存满的情况。这时，就无法在硬盘上创建新文件。

四、inode号码

每个inode都有一个号码，操作系统用inode号码来识别不同的文件。

这里值得重复一遍，Unix/Linux系统内部不使用文件名，而使用inode号码来识别文件。对于系统来说，文件名只是inode号码便于识别的别称或者绰号。

表面上，用户通过文件名，打开文件。实际上，系统内部这个过程分成三步：首先，系统找到这个文件名对应的inode号码；其次，通过inode号码，获取inode信息；最后，根据inode信息，找到文件数据所在的block，读出数据。

使用ls -i命令，可以看到文件名对应的inode号码：

```
ls -i example.txt
```

五、目录文件

Unix/Linux系统中，目录（directory）也是一种文件。打开目录，实际上就是打开目录文件。

目录文件的结构非常简单，就是一系列目录项（dirent）的列表。每个目录项，由两部分组成：所包含文件的文件名，以及该文件名对应的inode号码。

ls命令只列出目录文件中的所有文件名：

```
ls /etc
```

```
$ ls ./code
helloworld.js  imgHash.jpg  imgHash.py  server.js  tmp
$ █
```

ls -i命令列出整个目录文件，即文件名和inode号码：

```
ls -i /etc
```

```
$ ls -i ./code
388776 helloworld.js  389698 imgHash.py    13886 tmp
389031 imgHash.jpg     395507 server.js
$ █
```

如果要查看文件的详细信息，就必须根据inode号码，访问inode节点，读取信息。ls -l命令列出文件的详细信息。

```
ls -l /etc
```

```
$ ls -l ./code
total 84
-rw-r--r-- 1 ruanyf ruanyf   28 2011-07-15 18:57 helloworld.js
-rw-r--r-- 1 ruanyf ruanyf 69007 2011-07-21 14:48 imgHash.jpg
-rw-r--r-- 1 ruanyf ruanyf  1589 2011-07-21 14:46 imgHash.py
-rw-r--r-- 1 ruanyf ruanyf   298 2011-07-15 19:01 server.js
drwxr-xr-x 2 ruanyf ruanyf  4096 2011-07-21 15:00 tmp
$ █
```

理解了上面这些知识，就能理解目录的权限。目录文件的读权限（r）和写权限（w），都是针对目录文件本身。由于目录文件内只有文件名和inode号码，所以如果只有读权限，只能获取文件名，无法获取其他信息，因为其他信息都储存在inode节点中，而读取inode节点内的信息需要目录文件的执行

权限 (x)。

六、硬链接

一般情况下，文件名和inode号码是“一一对应”关系，每个inode号码对应一个文件名。但是，Unix/Linux系统允许，多个文件名指向同一个inode号码。

这意味着，可以用不同的文件名访问同样的内容；对文件内容进行修改，会影响到所有文件名；但是，删除一个文件名，不影响另一个文件名的访问。这种情况就被称为“硬链接”（hard link）。

ln命令可以创建硬链接：

ln 源文件 目标文件



运行上面这条命令以后，源文件与目标文件的inode号码相同，都指向同一个inode。inode信息中有一项叫做“链接数”，记录指向该inode的文件名总数，这时就会增加1。

反过来，删除一个文件名，就会使得inode节点中的“链接数”减1。当这个值减到0，表明没有文件名指向这个inode，系统就会回收这个inode号码，以及其所对应block区域。

这里顺便说一下目录文件的“链接数”。创建目录时，默认会生成两个目录项：“.”和“..”。前者的inode号码就是当前目录的inode号码，等同于当前目录的“硬链接”；后者的inode号码就是当前目录的父目录的inode号码，等同于父目录的“硬链接”。所以，任何一个目录的“硬链接”总数，总是等于2加上它的子目录总数（含隐藏目录）。

七、软链接

除了硬链接以外，还有一种特殊情况。

文件A和文件B的inode号码虽然不一样，但是文件A的内容是文件B的路径。读取文件A时，系统会自动将访问者导向文件B。因此，无论打开哪一个文件，最终读取的都是文件B。这时，文件A就称为文件B的“软链接”（soft link）或者“符号链接”（symbolic link）。

这意味着，文件A依赖于文件B而存在，如果删除了文件B，打开文件A就会报错：“No such file or directory”。这是软链接与硬链接最大的不同：文件A指向文件B的文件名，而不是文件B的inode号码，文件B的inode“链接数”不会因此发生变化。

ln -s命令可以创建软链接。

ln -s 源文文件或目录 目标文件或目录



八、inode的特殊作用

由于inode号码与文件名分离，这种机制导致了一些Unix/Linux系统特有的现象。

1. 有时，文件名包含特殊字符，无法正常删除。这时，直接删除inode节点，就能起到删除文件的作用。

2. 移动文件或重命名文件，只是改变文件名，不影响inode号码。

3. 打开一个文件以后，系统就以inode号码来识别这个文件，不再考虑文件名。因此，通常来说，系统无法从inode号码得知文件名。

第3点使得软件更新变得简单，可以在不关闭软件的情况下进行更新，不需要重启。因为系统通过inode号码，识别运行中的文件，不通过文件名。更新的时候，新版文件以同样的文件名，生成一个新的inode，不会影响到运行中的文件。等到下一次运行这个软件的时候，文件名就自动指向新版文件，旧版文件的inode则被回收。

(完)

EOF是什么？

我学习C语言的时候，遇到的一个问题就是[EOF](#)。

它是end of file的缩写，表示"文字流"（stream）的结尾。这里的"文字流"，可以是文件（file），也可以是标准输入（stdin）。



比如，下面这段代码就表示，如果不是文件结尾，就把文件的内容复制到屏幕上。

```
int c;  
  
while ((c = fgetc(fp)) != EOF) {  
    putchar (c);  
}
```

很自然地，我就以为，每个文件的结尾处，有一个叫做EOF的特殊字符，读取到这个字符，操作系统就认为文件结束了。

但是，后来我发现，EOF不是特殊字符，而是一个定义在头文件stdio.h的常量，一般等于-1。

```
#define EOF (-1)
```

于是，我就困惑了。

如果EOF是一个特殊字符，那么假定每个文本文件的结尾都有一个EOF（也就是-1），还是可以做到的，因为文本对应的ASCII码都是正值，不可能有负值。但是，二进制文件怎么办呢？怎么处理文件内部包含的-1呢？

这个问题让我想了很久，后来查了资料才知道，在Linux系统之中，EOF根本不是一个字符，而是当系统读取到文件结尾，所返回的一个信号值（也就是-1）。至于系统怎么知道文件的结尾，资料上说是通过比较文件的长度。

所以，处理文件可以写成下面这样：

```
int c;  
  
while ((c = fgetc(fp)) != EOF) {  
  
    do something  
  
}
```

这样写有一个问题。fgetc()不仅是遇到文件结尾时返回EOF，而且当发生错误时，也会返回EOF。因此，C语言又提供了feof()函数，用来保证确实是到了文件结尾。上面的代码feof()版本的写法就是：

```
int c;  
  
while (!feof(fp)) {  
  
    c = fgetc(fp);  
  
    do something;  
  
}
```

但是，这样写也有问题。fgetc()读取文件的最后一个字符以后，C语言的feof()函数依然返回0，表明没有到达文件结尾；只有当fgetc()向后再读取一个字符（即越过最后一个字符），feof()才会返回一个非零值，表示到达文件结尾。

所以，按照上面这样写法，如果一个文件含有n个字符，那么while循环的内部操作会运行n+1次。所以，最保险的写法是像下面这样：

```
int c = fgetc(fp);  
  
while (c != EOF) {  
  
    do something;  
  
    c = fgetc(fp);  
  
}  
  
if (feof(fp)) {  
  
    printf("\n End of file reached.");  
  
} else {
```

```
    printf("\n Something went wrong.\");\n"};
```

除了表示文件结尾，EOF还可以表示标准输入的结尾。

```
int c;\n\nwhile ((c = getchar()) != EOF) {\n    putchar(c);\n}
```

但是，标准输入与文件不一样，无法事先知道输入的长度，必须手动输入一个字符，表示到达EOF。

Linux中，在新的一行的开头，按下Ctrl-D，就代表EOF（如果在一行的中间按下Ctrl-D，则表示输出“标准输入”的缓存区，所以这时必须按两次Ctrl-D）；Windows中，Ctrl-Z表示EOF。（顺便提一句，Linux中按下Ctrl-Z，表示将该进程中断，在后台挂起，用fg命令可以重新切回到前台；按下Ctrl-C表示终止该进程。）

那么，如果真的想输入Ctrl-D怎么办？这时必须先按下Ctrl-V，然后就可以输入Ctrl-D，系统就不会认为这是EOF信号。[Ctrl-V](#)表示按“字面含义”解读下一个输入，要是想按“字面含义”输入Ctrl-V，连续输入两次就行了。

(完)

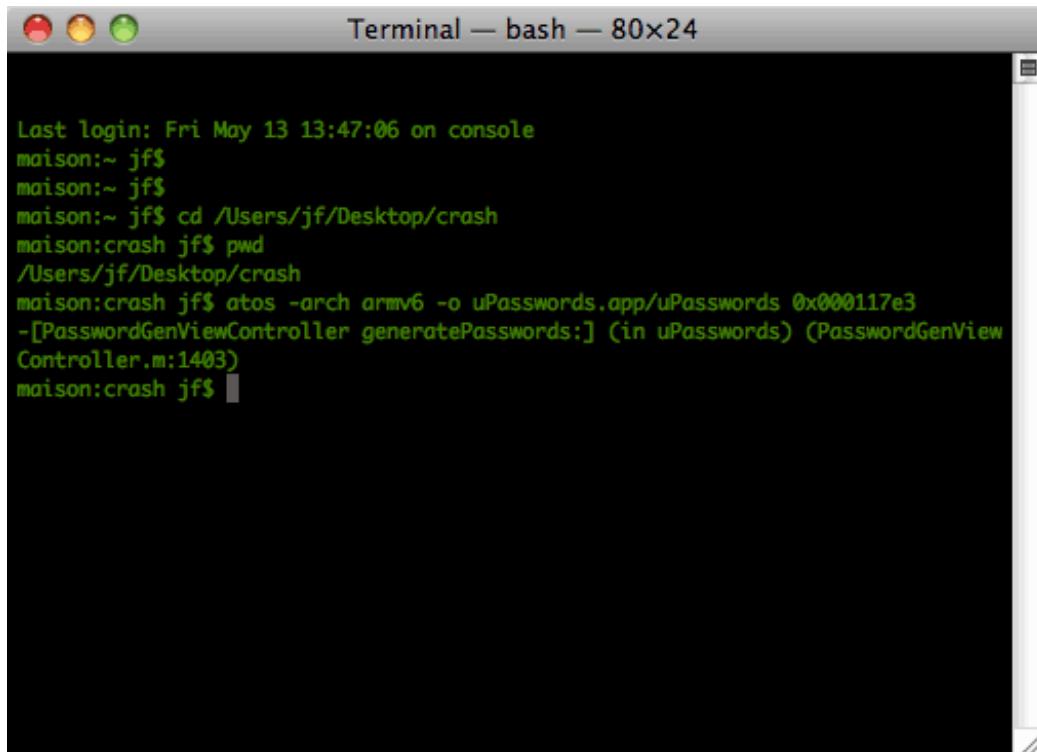
每行字符数 (CPL) 的起源

前几天，我收到网友小龙的Email。

他想与我讨论一个问题：

"各种计算机语言的编码风格，有的建议源码每行的字符数 (characters per line) 不超过72个，还有的建议不超过80个，这是为什么？区别在哪里？怎么来的？"

我一下子就被问住了。



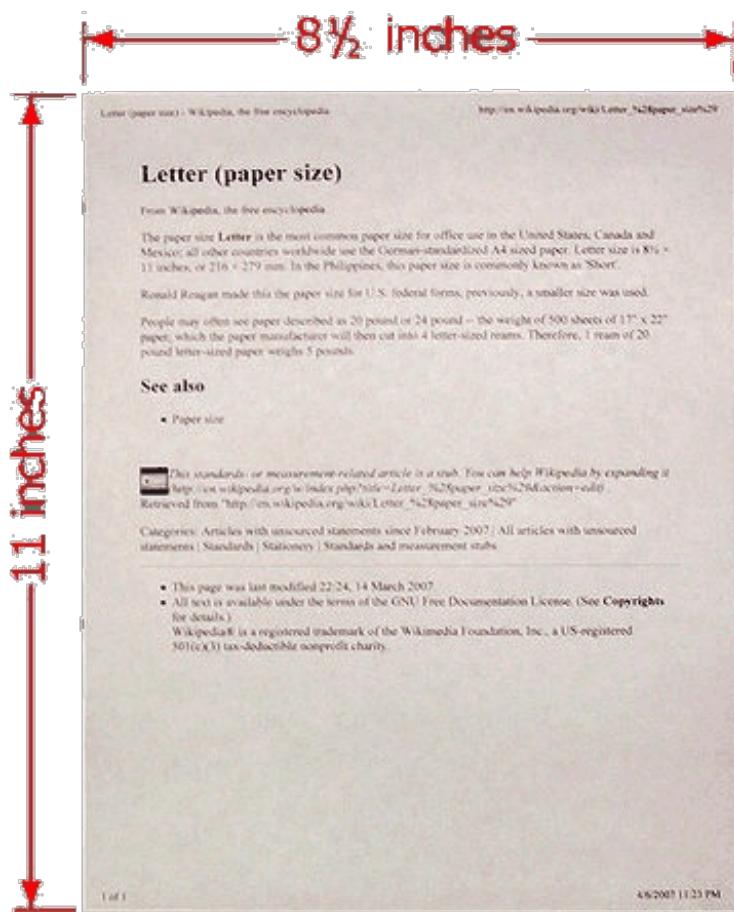
```
Last login: Fri May 13 13:47:06 on console
maison:~ jf$
maison:~ jf$
maison:~ jf$ cd /Users/jf/Desktop/crash
maison:crash jf$ pwd
/Users/jf/Desktop/crash
maison:crash jf$ atos -arch armv6 -o uPasswords.app/uPasswords 0x000117e3
-[PasswordGenViewController generatePasswords:] (in uPasswords) (PasswordGenView
Controller.m:1403)
maison:crash jf$
```

命令行状态下，终端窗口的显示宽度，默认是80个字符，这个我早就知道，但是并不清楚原因；至于72个字符，更是从未注意过。

幸好，世界上还有Wikipedia，我在里面找到了[答案](#)。



每行72个字符的限制，来源于打字机。上图是20世纪60年代初，非常流行的IBM公司生产的Selectric电动打字机。



当时，美国最通用的信笺大小是8.5英寸x11英寸（215.9 mm × 279.4 mm），叫做US Letter。打字的时候，左右两边至少要留出1英寸的页边距，因此每行的长度实际为6英寸。打字机使用等宽字体（monospaced）的情况下，每英寸可以打12个字符，就相当于一行72个字符。

A. Collected Grammar 43

1. Introduction

The first document in this set, RFC 2045, defines a number of header fields, including Content-Type. The Content-Type field is used to specify the nature of the data in the body of a MIME entity, by giving media type and subtype identifiers, and by providing auxiliary information that may be required for certain media types. After the

早期，源码必须用打字机打出来阅读，所以有些语言就规定，每行不得超过72个字符。直到今天，[RFC](#)文档依然采用这个规定，因为它从诞生起就采用打字稿的形式。



20世纪70年代，显示器出现了。它的主要用途之一，是将[打孔卡](#)（punched card）的输入显示出来。当时，最流行的打孔卡是IBM公司生产的80栏打孔卡，每栏为一个字符，80栏就是80个字符。



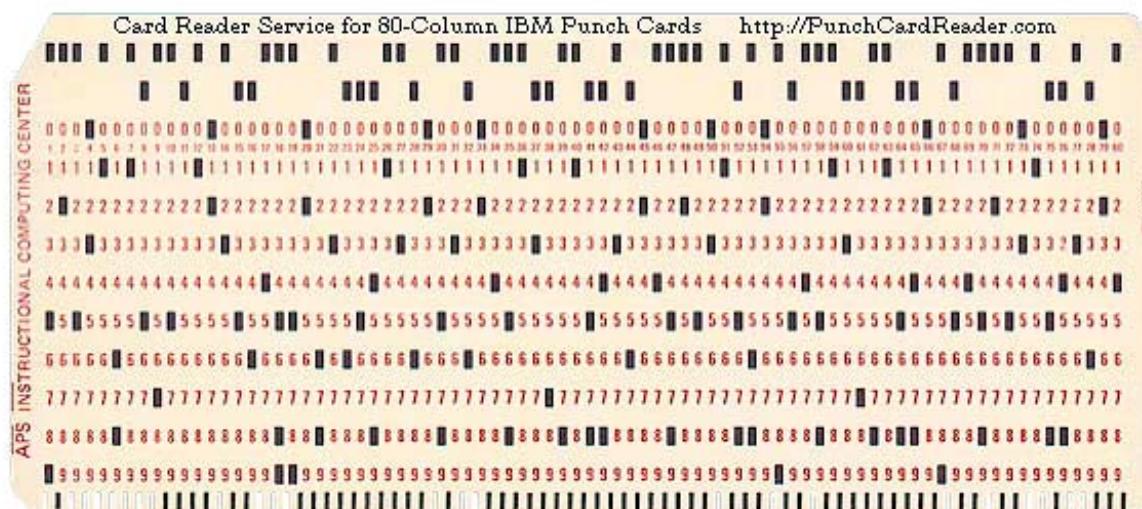
FORTRAN Coding Form

IBM FORM 14
Revised 5-5-64

PROGRAM PROGRAMMER		DATE		PUNCHING INSTRUCTIONS	GRAPHIC	PAGE OF	
STATEMENT NUMBER	COL.			PUNCH	CARD NUMBER		IDENTIFICATION SEQUENCE
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80	FORTRAN STATEMENT		1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80		

* A standard card form, ideal for easy data entry, is available for punching statements from this form.

上图是一张Fortran语言的源码填写单，一共有80栏，程序员在每一栏选择想要输入的字符，最多为80个字符。



然后，用机器自动生成打孔卡，在每栏选定的位置打一个孔。

计算机读取打孔卡以后，把每个孔转换为相应的字符。如果显示器每行显示80个字符，就正好与打孔卡一一对应，终端窗口的每行字符数（CPL）就这样确定下来了。

(完)

数字签名是什么？

今天，我读到一篇[好文章](#)。

它用图片通俗易懂地解释了，"数字签名"（digital signature）和"数字证书"（digital certificate）到底是什么。

我对这些问题的理解，一直是模模糊糊的，很多细节搞不清楚。读完这篇文章后，发现思路一下子就理清了。为了加深记忆，我把文字和图片都翻译出来了。

文中涉及的密码学基本知识，可以参见我以前的[笔记](#)。

=====

数字签名是什么？

作者：David Youd

翻译：阮一峰

原文网址：<http://www.youdzone.com/signature.html>

1.



鲍勃



鲍勃的公钥



鲍勃的私钥

鲍勃有两把钥匙，一把是公钥，另一把是私钥。

2.



帕蒂



道格



苏珊



每人一把

鲍勃把公钥送给他的朋友们----帕蒂、道格、苏珊----每人一把。

3.



苏珊

"Hey Bob,
how about
lunch at
Taco Bell. I
hear they
have free
refills!"



公钥加密

HNFmsEm6Un
BejhhyCGKO
KJUxhiygSBC
EiC0QYIh/Hn
3xgiKBcyLK1
UcYiYlxz2lCF
HDC/A

苏珊要给鲍勃写一封保密的信。她写完后用鲍勃的公钥加密，就可以达到保密的效果。

4.

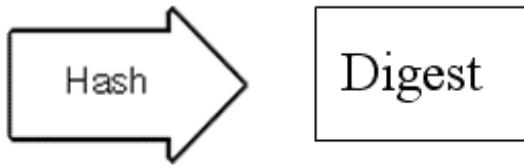
□

鲍勃收信后，用私钥解密，就看到了信件内容。这里要强调的是，只要鲍勃的私钥不泄露，这封信就是安全的，即使落在别人手里，也无法解密。

5.

The creator of PGP (Pretty Good Privacy), a public-key encryption software package for the protection of electronic mail. Since PGP was published commercially in February in June of 1995, it has spread rapidly all over the world, and has since become the de facto worldwide standard for encryption of e-mail, replacing numerous industry standards along the way. For three years I was the target of a criminal investigation by the US Customs Service, who claimed that keys were broken when PGP spread outside the US. That investigation was closed without indictment in January 1998.

Computers were developed in secret back in World War II mainly to break codes. Ordinary people did not have access to computers because they were new in number and too expensive. Some people predicted that there would never be a need for more than half a dozen computers in the country, and assumed that ordinary people would never have a need for computers. Some of the government's attitudes towardryptography today were learned in that period, and reflect the old attitudes toward computers. Why should ordinary people need to have access to good cryptography?



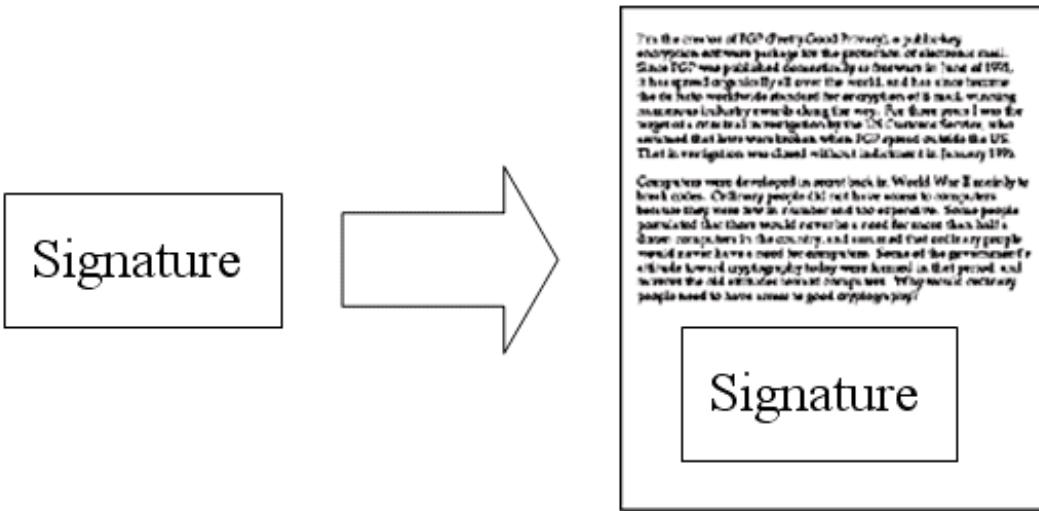
鲍勃给苏珊回信，决定采用"数字签名"。他写完后先用Hash函数，生成信件的摘要（digest）。

6.

□

然后，鲍勃使用私钥，对这个摘要加密，生成"数字签名"（signature）。

7.



鲍勃将这个签名，附在信件下面，一起发给苏珊。

8.

Signature



公钥解密

Digest

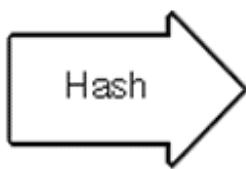
苏珊收信后，取下数字签名，用鲍勃的公钥解密，得到信件的摘要。由此证明，这封信确实是鲍勃发出的。

9.

This is the history of PGP (Pretty Good Privacy), a public-key encryption software package for the protection of electronic mail. Since PGP was published domestically in January in '91, it has spread rapidly all over the world, and has since become the de facto worldwide standard for encryption of e-mail. Many countries' industry needs change too over. For three years I was the target of a criminal investigation by the FBI/CIA/Central Service, who accused that letters were broken when PGP spread outside the US. That investigation was closed without indictment in February 1995.

Computers were developed in secret back in World War II mainly to break codes. Ordinary people did not have access to computers because they were few in number and too expensive. Some people predicted that there would never be a need because there isn't that many computers in the country, and insisted that ordinary people would never have a need for computers. Some of the government's code破壞 agency today were located in that period, and to meet the old situation there is no choice. Why would ordinary people need to have access to good cryptology?

Signature



Digest

|| ?



Digest

苏珊再对信件本身使用Hash函数，将得到的结果，与上一步得到的摘要进行对比。如果两者一致，就证明这封信未被修改过。

10.



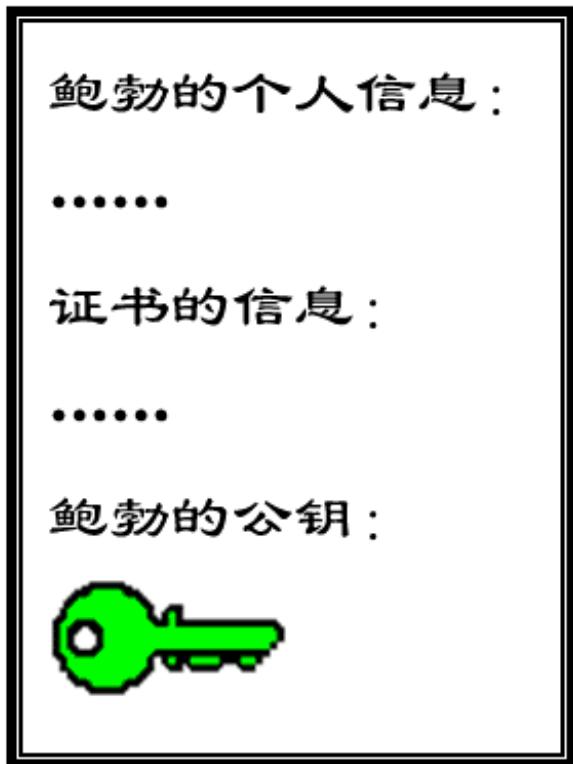
道格

假的公钥

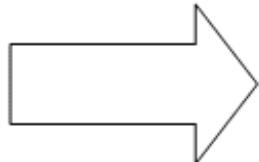
苏珊

复杂的情况出现了。道格想欺骗苏珊，他偷偷使用了苏珊的电脑，用自己的公钥换走了鲍勃的公钥。此时，苏珊实际拥有的是道格的公钥，但是还以为这是鲍勃的公钥。因此，道格就可以冒充鲍勃，用自己的私钥做成“数字签名”，写信给苏珊，让苏珊用假的鲍勃公钥进行解密。

11.



CA 的私钥



数字
证书

后来，苏珊感觉不对劲，发现自己无法确定公钥是否真的属于鲍勃。她想到了一个办法，要求鲍勃去找“证书中心”(certificate authority, 简称CA)，为公钥做认证。证书中心用自己的私钥，对鲍勃的公钥和一些相关信息一起加密，生成“数字证书”(Digital Certificate)。

12.

This is the history of PGP (Pretty Good Privacy), a public-key encryption software package for the protection of electronic mail. Since PGP was published commercially as freeware in June of 1991, it has spread organically all over the world, and has since become the de facto worldwide standard for encryption of E-mail, surpassing numerous industry standards along the way. For three years I was the target of a criminal investigation by the FBI Countermeasures, who accused that my work broken when PGP spread outside the US. That investigation was closed without indictment in January 1995.

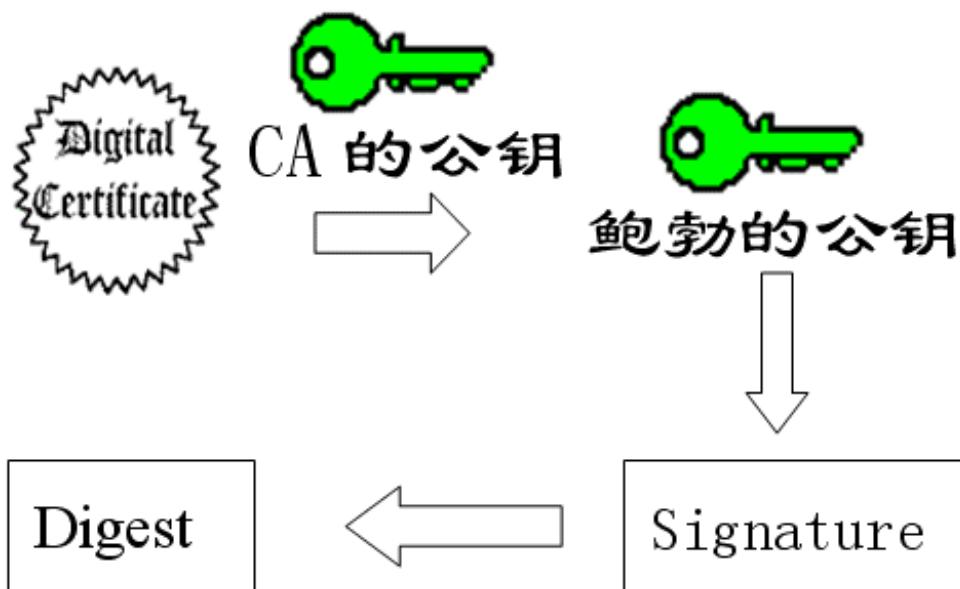
Computers were developed in secret back in World War II mainly to break codes. Ordinary people did not have access to computers because they were few in number and too expensive. Some people predicted that there would never be a need because less than half a dozen computers in the country, and assumed that ordinary people would never have a need for computers. Some of the government's attitudes toward cryptography today were formed in that period, and to repeat the old estimates about computers: "Why would ordinary people need to have access to good cryptography?"

Signature



鲍勃拿到数字证书以后，就可以放心了。以后再给苏珊写信，只要在签名的同时，再附上数字证书就行了。

13.



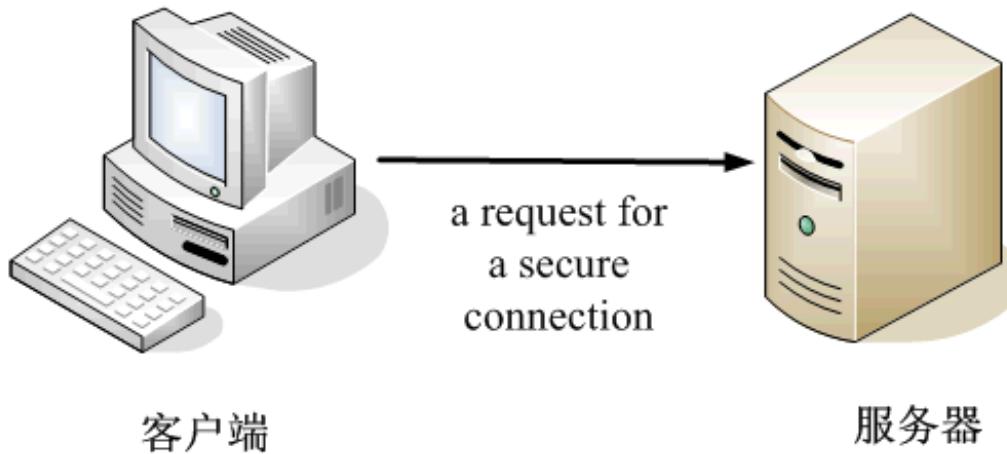
苏珊收信后，用CA的公钥解开数字证书，就可以拿到鲍勃真实的公钥了，然后就能证明"数字签名"是否真的是鲍勃签的。

14.



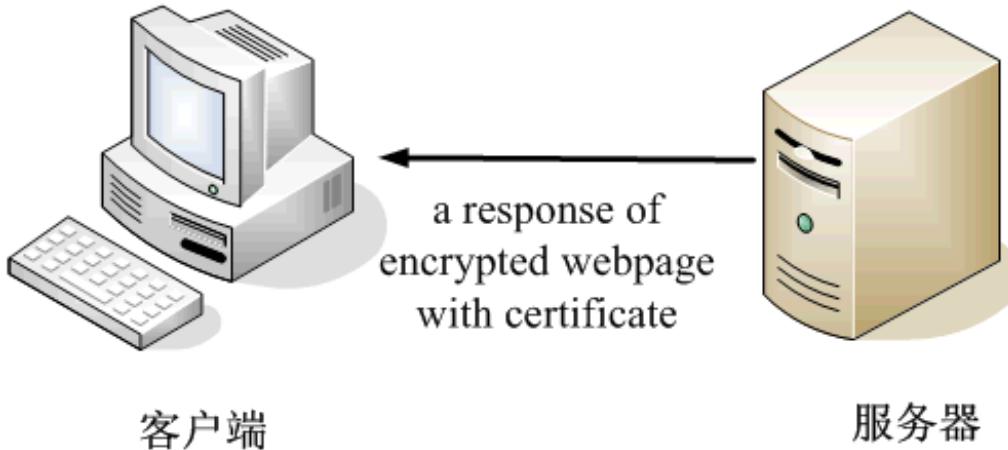
下面，我们看一个应用"数字证书"的实例：https协议。这个协议主要用于网页加密。

15.



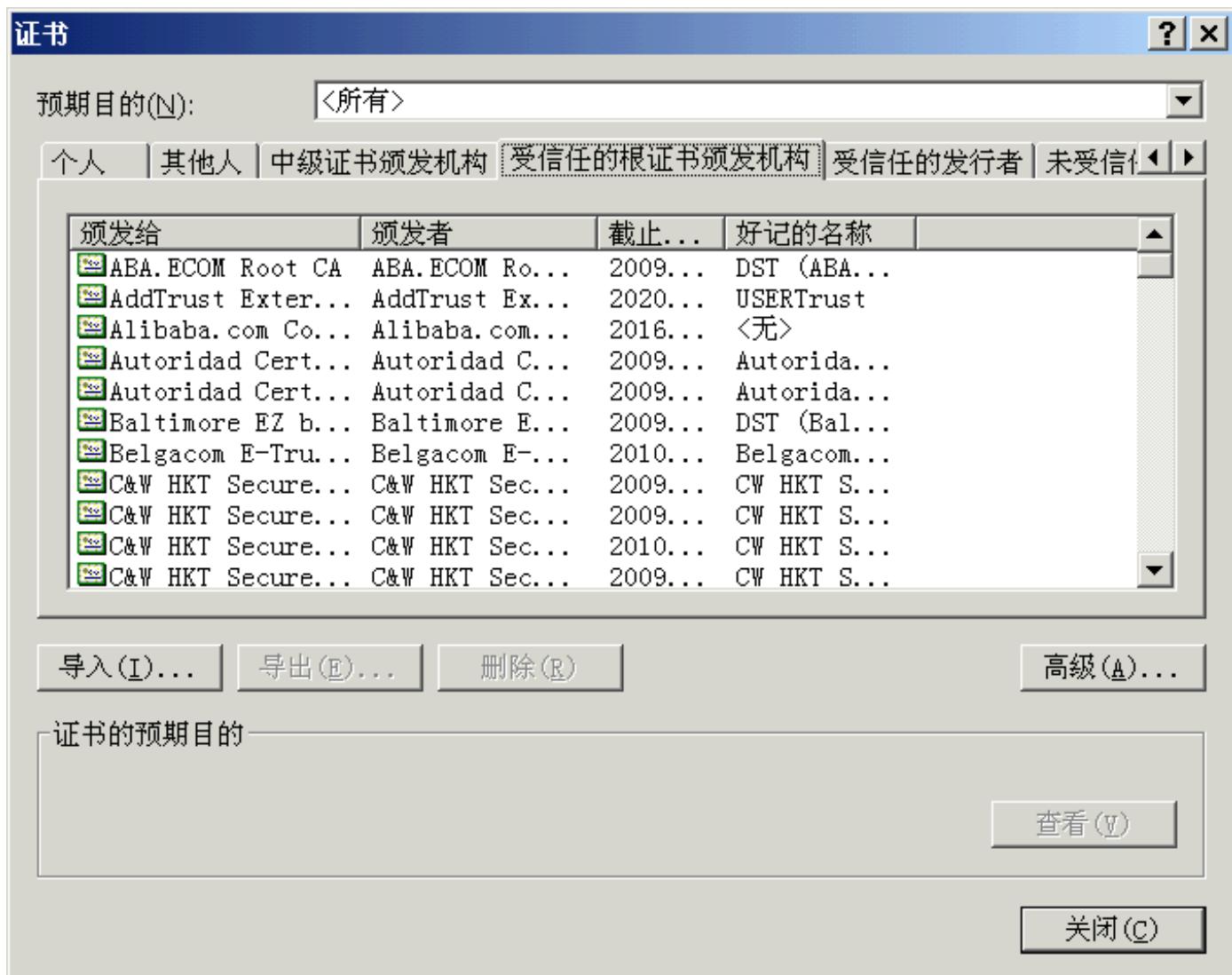
首先，客户端向服务器发出加密请求。

16.



服务器用自己的私钥加密网页以后，连同本身的数字证书，一起发送给客户端。

17.



客户端（浏览器）的“证书管理器”，有“受信任的根证书颁发机构”列表。客户端会根据这张列表，查

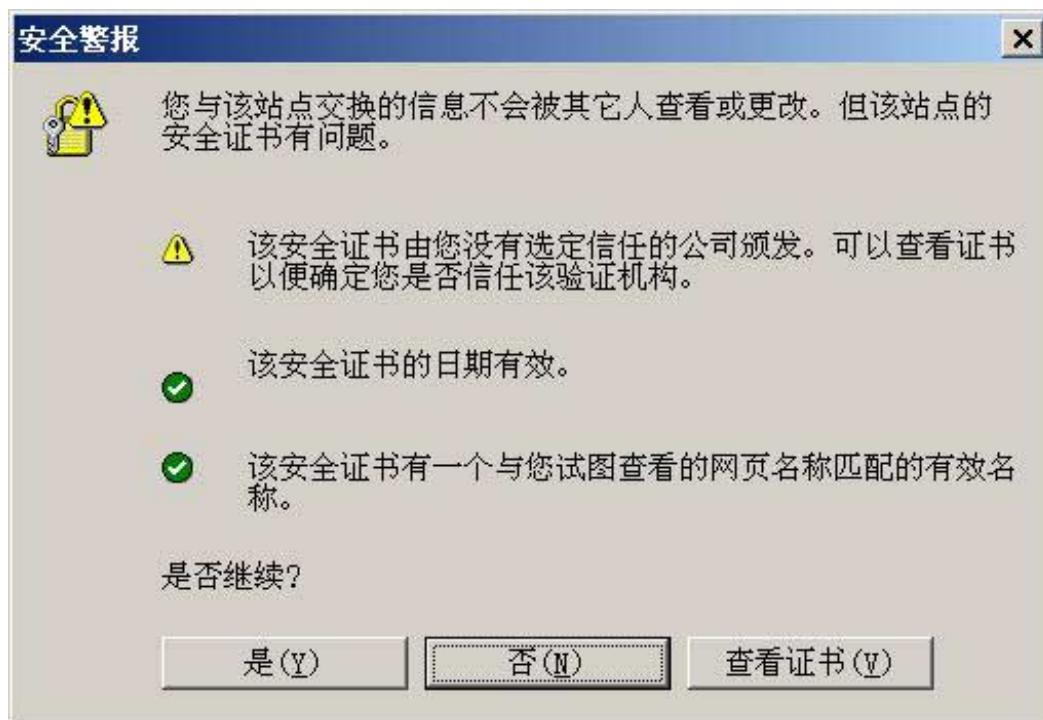
看解开数字证书的公钥是否在列表之内。

18.



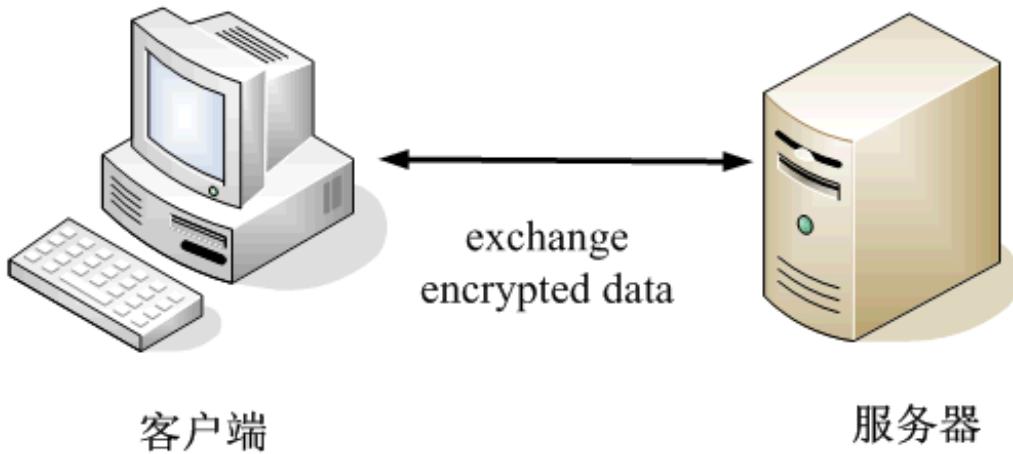
如果数字证书记载的网址，与你正在浏览的网址不一致，就说明这张证书可能被冒用，浏览器会发出警告。

19.



如果这张数字证书不是由受信任的机构颁发的，浏览器会发出另一种警告。

20.



如果数字证书是可靠的，客户端就可以使用证书中的服务器公钥，对信息进行加密，然后与服务器交换加密信息。

(完)

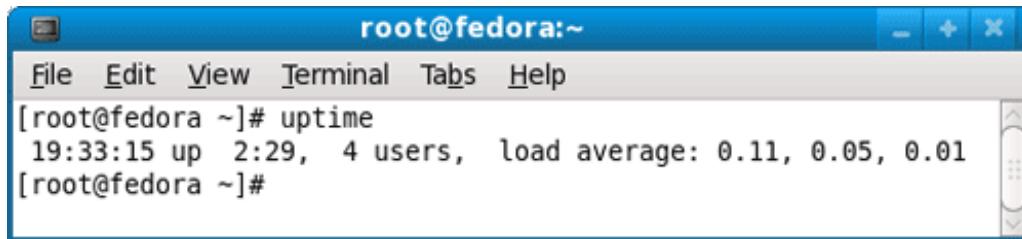
理解Linux系统负荷

一、查看系统负荷

如果你的电脑很慢，你或许想查看一下，它的工作量是否太大了。

在Linux系统中，我们一般使用uptime命令查看（w命令和top命令也行）。（另外，它们在苹果公司的Mac电脑上也适用。）

你在终端窗口键入uptime，系统会返回一行信息。



```
root@fedora:~# uptime
19:33:15 up 2:29, 4 users, load average: 0.11, 0.05, 0.01
[root@fedora ~]#
```

这行信息的后半部分，显示"load average"，它的意思是"系统的平均负荷"，里面有三个数字，我们可以从中判断系统负荷是大还是小。

load average: 0.09, 0.05, 0.01

为什么会有三个数字呢？你从手册中查到，它们的意思分别是1分钟、5分钟、15分钟内系统的平均负荷。

如果你继续看手册，它还会告诉你，当CPU完全空闲的时候，平均负荷为0；当CPU工作量饱和的时候，平均负荷为1。

那么很显然，"load average"的值越低，比如等于0.2或0.3，就说明电脑的工作量越小，系统负荷比较轻。

但是，什么时候能看出系统负荷比较重呢？等于1的时候，还是等于0.5或等于1.5的时候？如果1分钟、5分钟、15分钟三个值不一样，怎么办？

二、一个类比

判断系统负荷是否过重，必须理解load average的真正含义。下面，我根据"[Understanding Linux CPU Load](#)"这篇文章，尝试用最通俗的语言，解释这个问题。

首先，假设最简单的情况，你的电脑只有一个CPU，所有的运算都必须由这个CPU来完成。

那么，我们不妨把这个CPU想象成一座大桥，桥上只有一根车道，所有车辆都必须从这根车道上通过。（很显然，这座桥只能单向通行。）

系统负荷为0，意味着大桥上一辆车也没有。



系统负荷为0.5，意味着大桥一半的路段有车。



系统负荷为1.0，意味着大桥的所有路段都有车，也就是说大桥已经"满"了。但是必须注意的是，直到此时大桥还是能顺畅通行的。



系统负荷为1.7，意味着车辆太多了，大桥已经被占满了（100%），后面等着上桥的车辆为桥面车辆的70%。以此类推，系统负荷2.0，意味着等待上桥的车辆与桥面的车辆一样多；系统负荷3.0，意味着等待上桥的车辆是桥面车辆的2倍。总之，当系统负荷大于1，后面的车辆就必须等待了；系统负荷越大，过桥就必须等得越久。



CPU的系统负荷，基本上等同于上面的类比。大桥的通行能力，就是CPU的最大工作量；桥梁上的车辆，就是一个个等待CPU处理的进程（process）。

如果CPU每分钟最多处理100个进程，那么系统负荷0.2，意味着CPU在这1分钟里只处理20个进程；系统负荷1.0，意味着CPU在这1分钟里正好处理100个进程；系统负荷1.7，意味着除了CPU正在处理的100个进程以外，还有70个进程正排队等着CPU处理。

为了电脑顺畅运行，系统负荷最好不要超过1.0，这样就没有进程需要等待了，所有进程都能第一时间得到处理。很显然，1.0是一个关键值，超过这个值，系统就不在最佳状态了，你要动手干预了。

三、系统负荷的经验法则

1.0是系统负荷的理想值吗？

不一定，系统管理员往往会留一点余地，当这个值达到0.7，就应当引起注意了。经验法则是这样的：

当系统负荷持续大于0.7，你必须开始调查了，问题出在哪里，防止情况恶化。

当系统负荷持续大于1.0，你必须动手寻找解决办法，把这个值降下来。

当系统负荷达到5.0，就表明你的系统有很严重的问题，长时间没有响应，或者接近死机了。你不应该让系统达到这个值。

四、多处理器

上面，我们假设你的电脑只有1个CPU。如果你的电脑装了2个CPU，会发生什么情况呢？

2个CPU，意味着电脑的处理能力翻了一倍，能够同时处理的进程数量也翻了一倍。

还是用大桥来类比，两个CPU就意味着大桥有两根车道了，通车能力翻倍了。



所以，2个CPU表明系统负荷可以达到2.0，此时每个CPU都达到100%的工作量。推广开来，n个CPU的电脑，可接受的系统负荷最大为n.0。

五、多核处理器

芯片厂商往往在一个CPU内部，包含多个CPU核心，这被称为多核CPU。

在系统负荷方面，多核CPU与多CPU效果类似，所以考虑系统负荷的时候，必须考虑这台电脑有几个CPU、每个CPU有几个核心。然后，把系统负荷除以总的核心数，只要每个核心的负荷不超过1.0，就表明电脑正常运行。

怎么知道电脑有多少个CPU核心呢？

"cat /proc/cpuinfo"命令，可以查看CPU信息。"grep -c 'model name' /proc/cpuinfo"命令，直接返回CPU的总核心数。

六、最佳观察时长

最后一个问题，"load average"一共返回三个平均值----1分钟系统负荷、5分钟系统负荷，15分钟系统负荷，----应该参考哪个值？

如果只有1分钟的系统负荷大于1.0，其他两个时间段都小于1.0，这表明只是暂时现象，问题不大。

如果15分钟内，平均系统负荷大于1.0（调整CPU核心数之后），表明问题持续存在，不是暂时现象。所以，你应该主要观察"15分钟系统负荷"，将它作为电脑正常运行的指标。

=====

[参考文献]

1. [Understanding Linux CPU Load](#)
2. [Wikipedia - Load \(computing\)](#)

(完)

DOS的历史

昨日（7月27日），微软公司的[DOS操作系统](#)迎来了30岁生日。

DOS是历史上一个划时代的产品，标识着PC（个人电脑）的崛起和普及，对计算机行业影响深远。

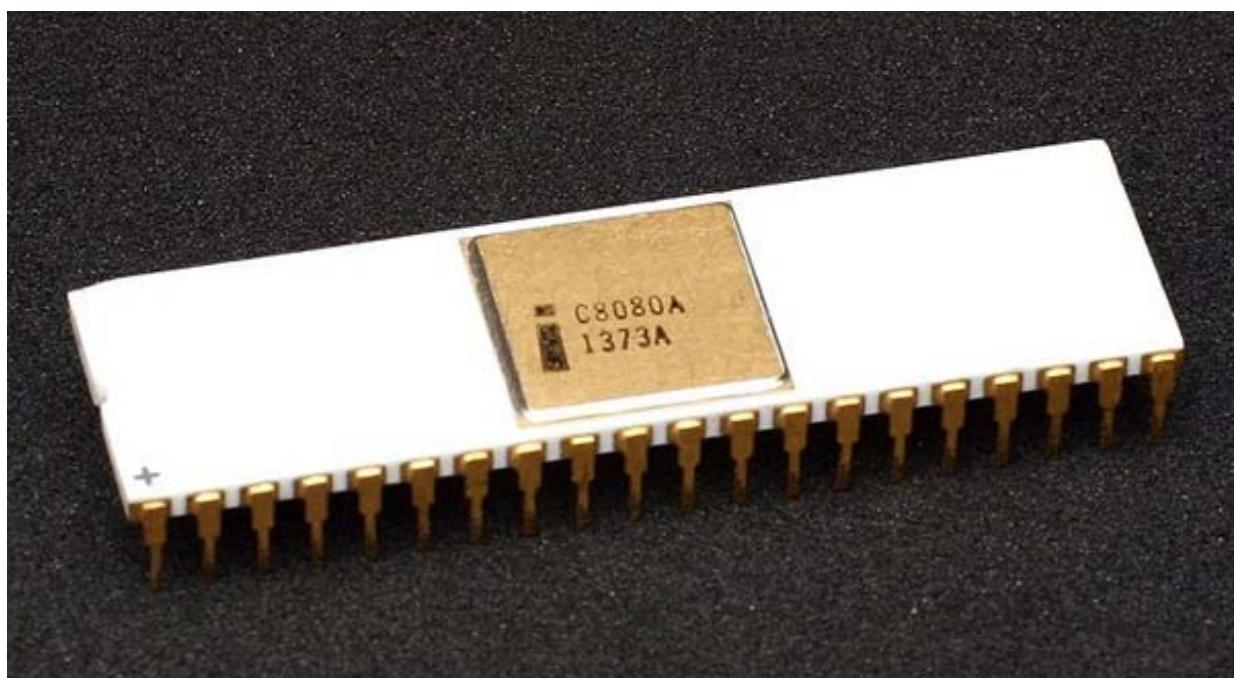
只有了解DOS的历史，才能理解今天的计算机工业从何而来。下面就是我对这一段历史的介绍。

=====

DOS的历史

作者：阮一峰

1.



1974年4月，Intel推出8位芯片8080。这块芯片的体积和性能，已经能够满足开发微型电脑的需要，标志微机时代即将来临。

2.



1975年初，MITS电脑公司推出了基于8080芯片的Altair 8800微机，这是人类历史上第一台PC（个人电脑）。

3.

HOW TO "READ" FM TUNER SPECIFICATIONS

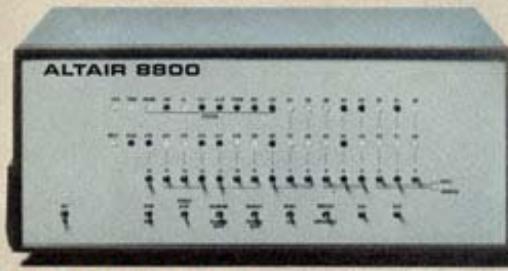
Popular Electronics

WORLD'S LARGEST-SELLING ELECTRONICS MAGAZINE JANUARY 1975/75¢

PROJECT BREAKTHROUGH!

**World's First Minicomputer Kit
to Rival Commercial Models...**

"ALTAIR 8800" **SAVE OVER \$1000**



ALSO IN THIS ISSUE:

- An Under-\$90 Scientific Calculator Project
- CCD's—TV Camera Tube Successor?
- Thyristor-Controlled Photoflashers

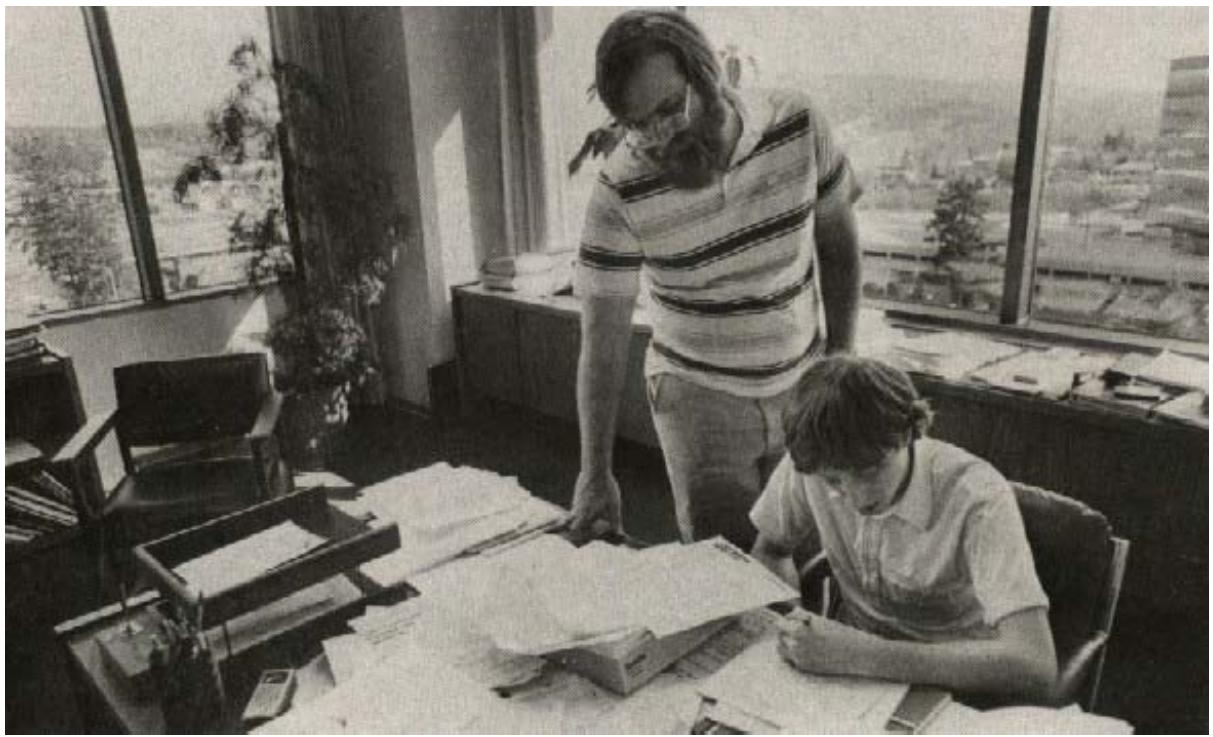
TEST REPORTS:

- Technics 200 Speaker System
- Pioneer RT-1011 Open-Reel Recorder
- Tram Diamond-40 CB AM Transceiver
- Edmund Scientific "Kirkian" Photo Kit
- Hewlett-Packard 5381 Frequency Counter



16101

1975年1月，Popular Electronics杂志以封面报道的形式，介绍了Altair 8800。这是历史上第一篇关于微机的新闻报道。



22岁的西雅图程序员Paul Allen看到了这一期杂志，深感震动，就把它拿给好友20岁的Bill Gates看。

两人决定为Altair 8800开发一套BASIC解释器，卖给MITS公司。1975年7月，他们用这个产品成立了微软公司。

5.

Loading CPM.SYS...

```
CP/M-86 for the IBM PC/XT/AT, Vers. 1.1 (Patched)
Copyright (C) 1983, Digital Research
```

Hardware Supported :

```
Diskette Drive(s) : 3
Hard Disk Drive(s) : 1
Parallel Printer(s) : 1
Serial Port(s) : 1
Memory (Kb) : 640
```

```
D>a:
A>dir
A: PIP      CMD : STAT      CMD : SUBMIT    CMD : ASM86      CMD
A: GENCMD   CMD : DDT86     CMD : TOD       CMD : ED        CMD
A: HELP     CMD : HELP      HLP : SYS       CMD : ASSIGN    CMD
A: FORMAT   CMD : CLDIR     CMD : WRTLDR    CMD : BOOTPCDS  SYS
A: BOOTWIN  SYS : CPM      H86 : WINSTALL   SUB : PD        CMD
A: WCPM     SYS : DISKUTIL CMD
A>_
```

User 0 0:00:11 Jan. 1, 2000

1975年，另一家公司Digital Research为Altair 8800开发了操作系统CP/M。它很快成为Intel 8080芯

片的标准操作系统。（上图为CP/M的运行界面。）

6.



1978年，Intel公司推出历史上第一块16位芯片8086。

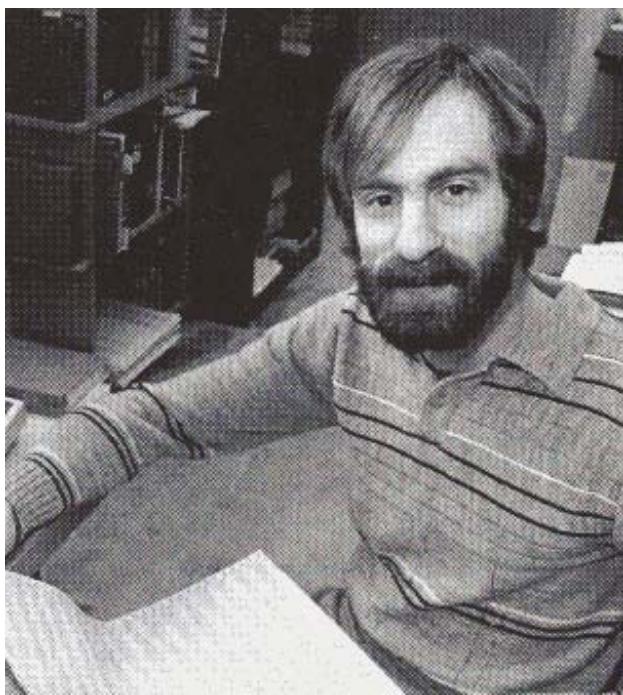
7.



1979年，一家名叫Seattle Computer Products (SCP) 的公司，决定开发基于8086芯片的个人电脑。

它原计划采用CP/M作为操作系统，但是此时CP/M还未完成针对16位芯片的升级。

8.



1980年4月，足足一年之后，CP/M还是没有推出16位的版本。SCP决定不等了，自己开发16位操作系统。

24岁的程序员Tim Paterson负责这个任务。

9.



1980年8月，Tim Paterson完成了原始的操作系统，取名为QDOS，意思是"简易的操作系统"（Quick and Dirty Operating System）。

在设计上，他充分借鉴了CP/M，用户界面和编程接口几乎完全一样，这使得CP/M上的应用程序，可以直接在QDOS上运行。他做出的最大改变，就是为QDOS引入了微软公司BASIC解释器的FAT文件系统。

```
MS-DOS Command release 1.00, version 1.19
Current date is Mon 7-27-2009
Enter new date:
Current time is 15:05:53.11
Enter new time:

C:dir
COMMAND COM      4986  1-18-84   2:01p
SYS      COM      914   1-18-84   2:50p
CHKDSK   COM     1754  8-16-83  12:56p
CONFIGUR COM    19724 5-03-84  10:33a
DEBUG    COM     6003  8-16-83   1:02p
DISKCOMP COM     5344  11-11-83  1:37p
DISKCOPY COM     5728  12-13-83  1:37p
EDLIN    COM     2313  8-16-83  12:56p
EXE2BIN  EXE     1280  8-16-83   1:01p
FILCOM   COM     8320  8-16-83  12:56p
FORMAT   COM     3856  4-24-84   3:50p
LIB      EXE    32128  8-16-83   1:00p
LINK    EXE    41856  8-16-83   1:00p
PRINT    COM     1740  8-16-83   1:43p
RDCPM    COM     3548  12-13-83  1:28p
15 File(s)
C:d
```

1980年10月，IBM公司决定推出基于Intel 8086芯片的PC。

它找到Digital Research公司，要求获得授权使用CP/M系统。但是，协议没有谈成。于是，IBM又去找微软公司，要求微软为它提供操作系统。

当时，微软没有操作系统产品，但是Bill Gates知道SCP公司正在开发QDOS。微软支付2.5万美元给SCP，获得了QDOS的使用许可。（上图为DOS的运行界面。）



1981年7月，微软对IBM PC的整个设计已经相当了解，Bill Gates意识到未来PC市场的巨大规模，决定不再使用许可证模式，而是直接把QDOS买下来。这又花费了微软公司5-7万美元。

与此同时，Tim Paterson也从SCP辞职了，微软雇用了他。

12.



1981年7月27日，协议达成。QDOS成了微软的财产，名称正式改为MS-DOS。微软对DOS的解释是"磁盘操作系统"（Disk Operating System）。

这一天，就是微软公司DOS操作系统的诞生纪念日。

13.



1981年8月12日，IBM公司正式推出个人电脑产品IBM PC，使用的操作系统是MS-DOS 1.14版。

14.



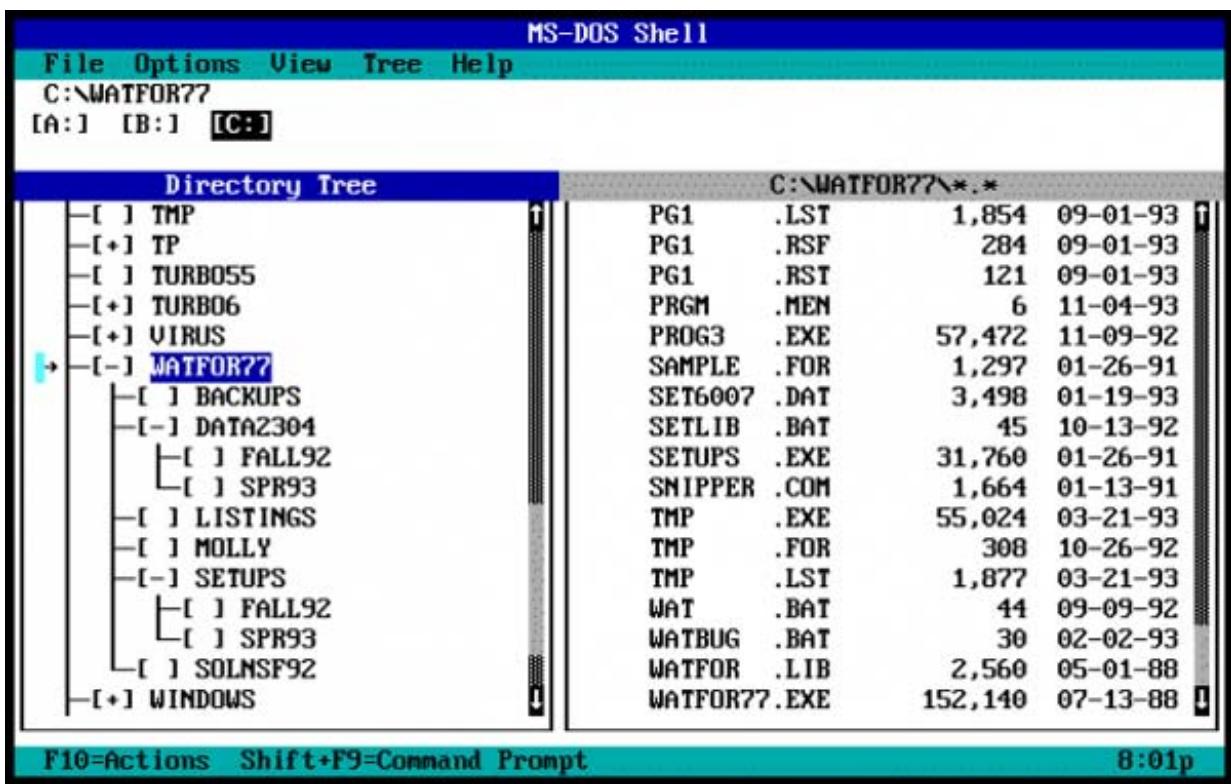
1983年3月8日，IBM又推出增强版IBM PC/XT，第一次在PC上配备了硬盘，使用的操作系统是MS-DOS 2.0版。

15.



1984年，IBM推出了下一代个人电脑IBM PC/AT，操作系统是MS-DOS 3.0版。

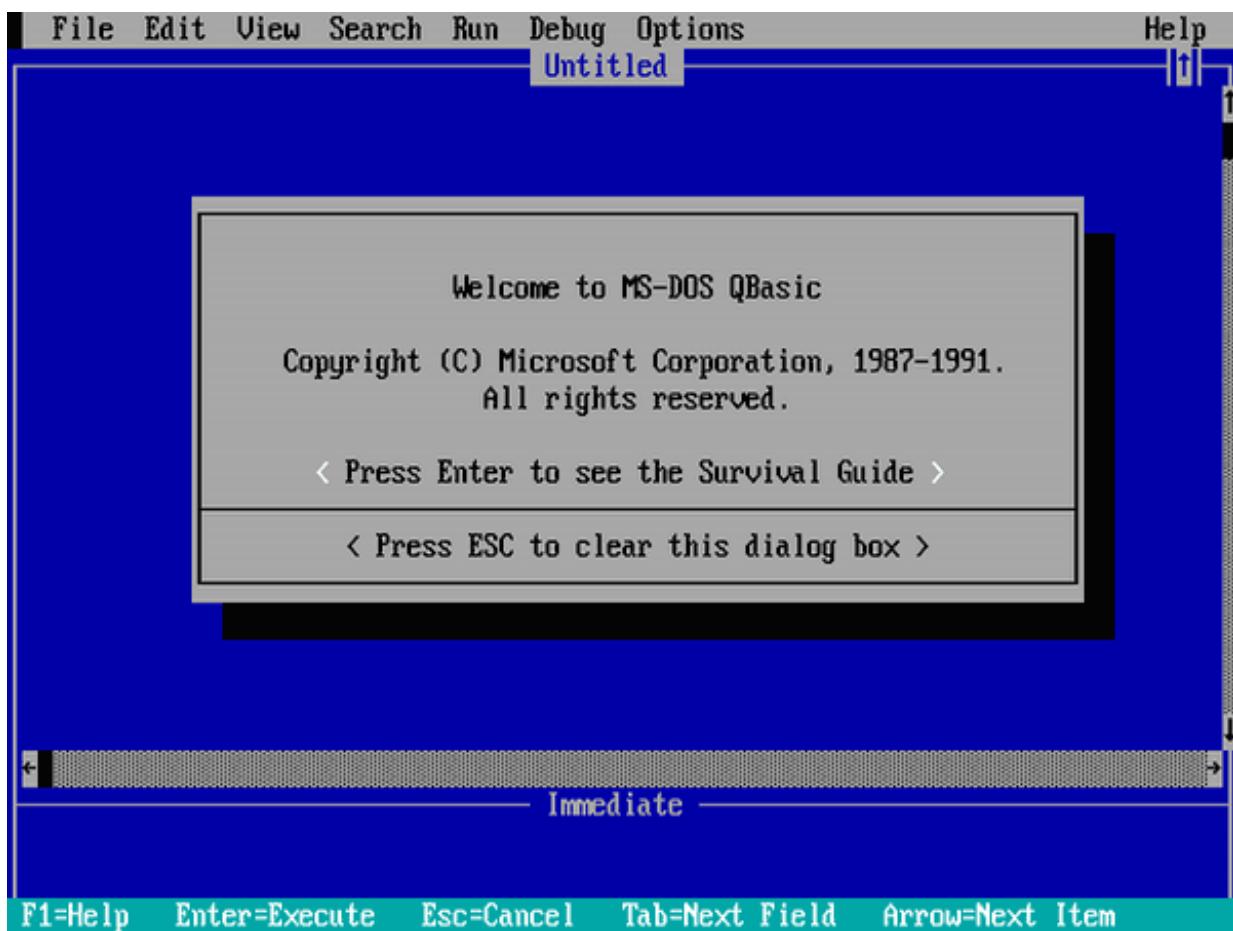
16.



1989年，MS-DOS 4.0版发布，开始支持鼠标和图形界面。

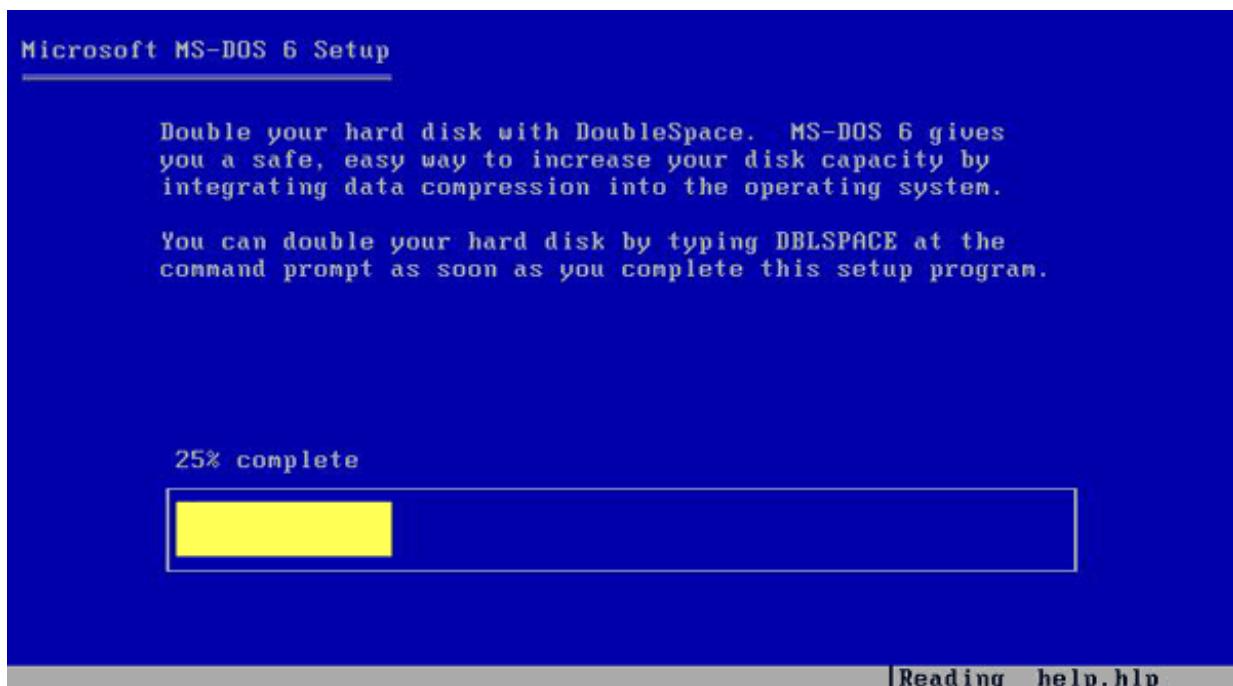
此时，微软已经准备终结DOS这个产品了。微软公开表态，用户可以考虑放弃DOS，转而使用由IBM和微软共同开发的OS/2操作系统。

但是不久以后，Windows 3.0获得巨大成功，微软也就不再考虑OS/2了。



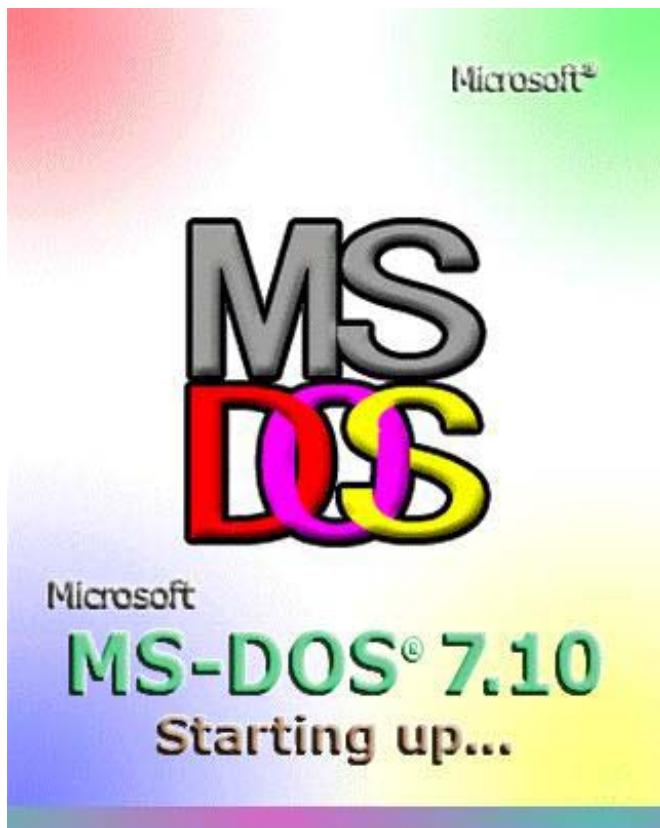
1991年，MS-DOS 5.0版发布，内置QBasic编程环境。这是MS-DOS最后一次作为单独产品出现。

18.



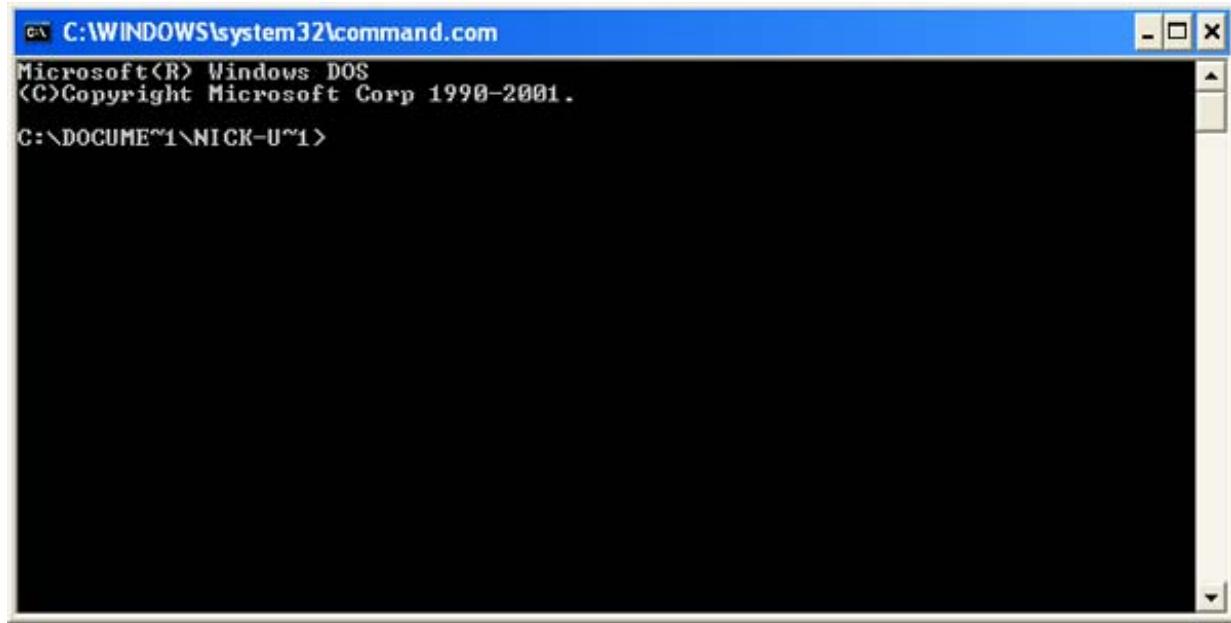
1993年，MS-DOS 6.0版发布，具备了磁盘压缩技术。

19.



1995年，MS-DOS 7.0版支持FAT32文件系统，它随同Windows 95一起发布。

20.



2000年9月14日，MS-DOS的最后一个版本8.0版发布，只用于Windows XP系统的启动盘。至此，微软公司的DOS开发正式宣告全部结束。

[参考文献]

* 维基百科, [Timeline Of x86 DOS operating systems](#)

* Tony Smith, [Microsoft's MS-DOS is 30 today](#)

* Sebastian Anthony, [MS-DOS is 30 years old today](#)

(完)

四位计算机的原理及其实现

你是否想过，计算机为什么会加减乘除？或者更直接一点，计算机的原理到底是什么？

[Waitingforfriday](#)有一篇详细的教程，讲解了如何自己动手，制作一台四位计算机。从中可以看到，二进制、数理逻辑、电子学怎样融合在一起，构成了现代计算机的基础。

一、什么是二进制？

首先，从最简单的讲起。

计算机内部采用二进制，每一个数位只有两种可能"0"和"1"，运算规则是"逢二进一"。举例来说，有两个位A和B，它们相加的结果只可能有四种。

A	B	Sum	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

这张表就叫做"真值表"（truth table），其中的sum表示"和位"，carry表示"进位"。如果A和B都是0，和就是0，因此"和位"和"进位"都是0；如果A和B有一个为1，另一个为0，和就是1，不需要进位；如果A和B都是1，和就是10，因此"和位"为0，"进位"为1。

二、逻辑门（Logic Gate）

布尔运算（Boolean operation）的规则，可以套用在二进制加法上。布尔运算有三个基本运算符：AND，OR，NOT，又称"与门"、"或门"、"非门"，合称"逻辑门"。它们的运算规则是：

AND：如果(A=1 AND B=1)，则输出结果为1。

OR：如果(A=1 OR B=1)，则输出结果为1。

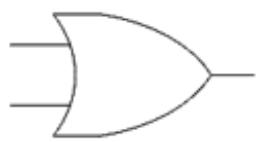
NOT：如果(A=1)，则输出结果为0。

两个输入（A和B）都为1，AND（与门）就输出1；只要有任意一个输入（A或B）为1，OR（或门）

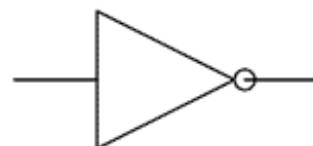
就输出1；NOT（非门）的作用，则是输出一个输入值的相反值。它们的图形表示如下：



AND



OR



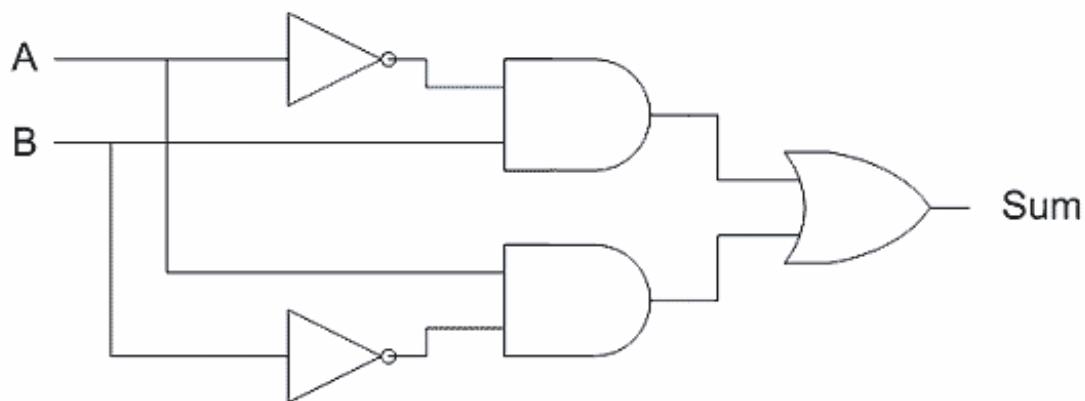
NOT

三、真值表的逻辑门表示

现在把"真值表"的运算规则，改写为逻辑门的形式。

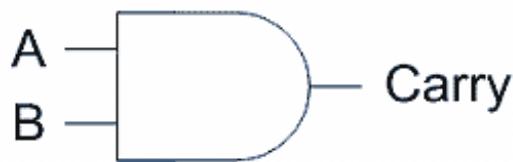
先看sum（和位），我们需要的是这样一种逻辑：当两个输入不相同时，输出为1，因此运算符应该是OR；当两个输入相同时，输出为0，这可以用两组AND和NOT的组合实现。最后的逻辑组合图如下：

A	B	Sum	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

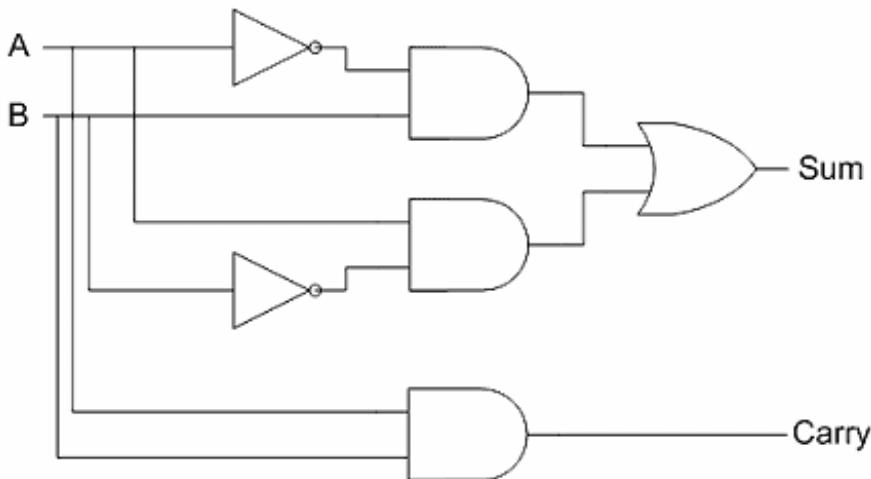


再看carry（进位）。它比较简单，两个输入A和B都为1就输出1，否则就输出0，因此用一个AND运算是就行了。

A	B	Sum	Carry
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1



现在把sum和carry组合起来，就能得到整张真值表了。这被称为"半加器"（half-adder），因为它只考虑了单独两个位的相加，没有考虑可能还存在低位进上来的位。



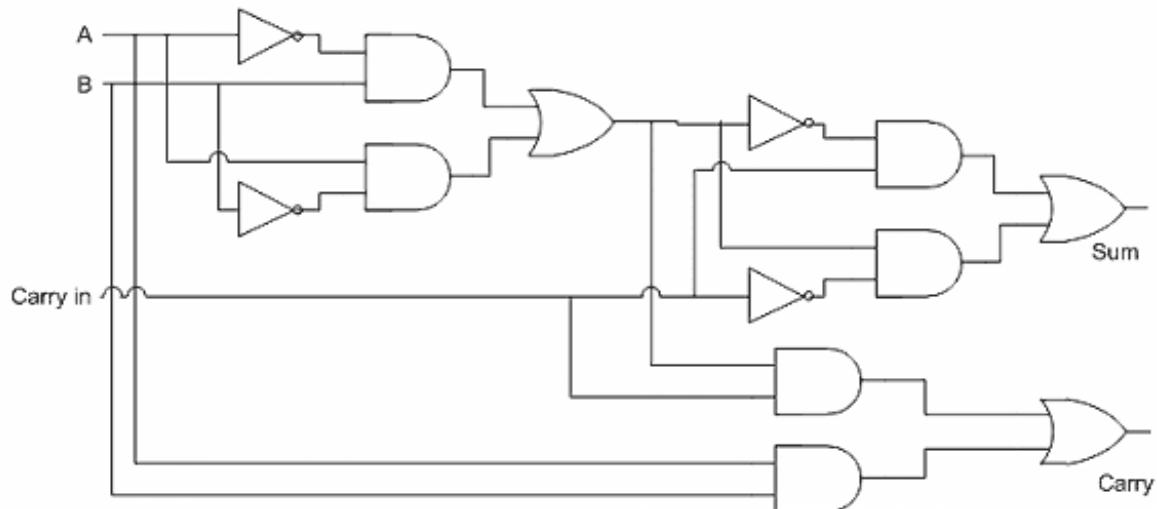
四、扩展的真值表和全加器

如果把低位进上来的位，当做第三个输入（input），也就是说，除了两个输入值A和B以外，还存在一个输入（input）的carry，那么问题就变成了如何在三个输入的情况下，得到输出（output）的sum（和位）和carry（进位）。

这时，真值表被扩展成下面的形式：

Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

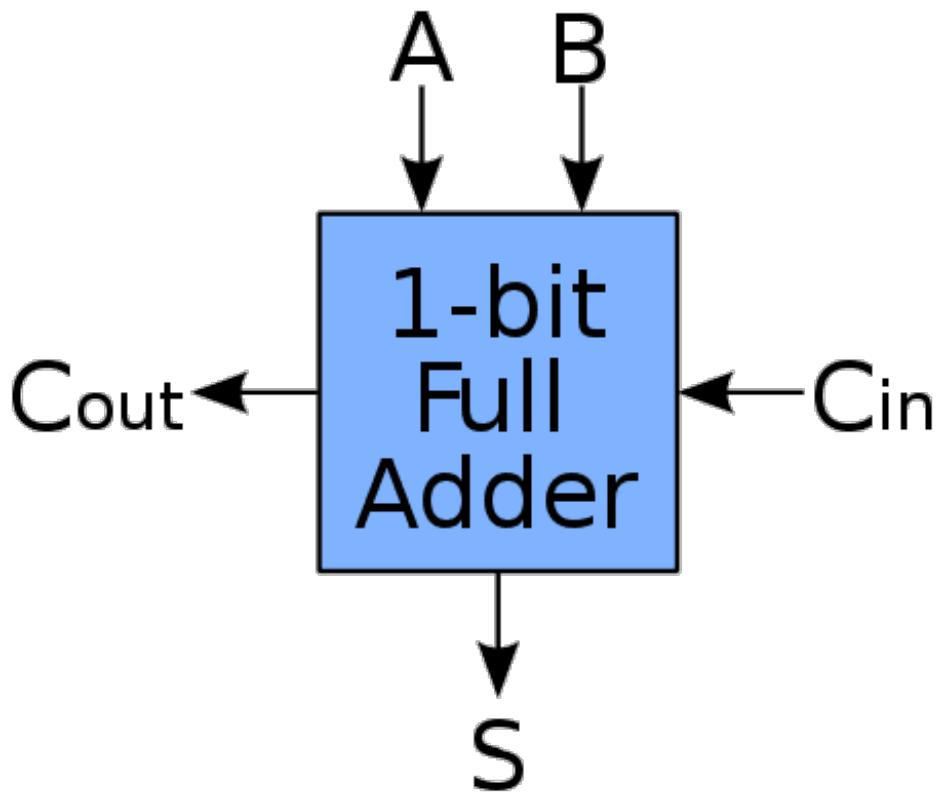
如果你理解了半加器的设计思路，就不难把它扩展到新的真值表，这就是“全加器”（full-adder）了。



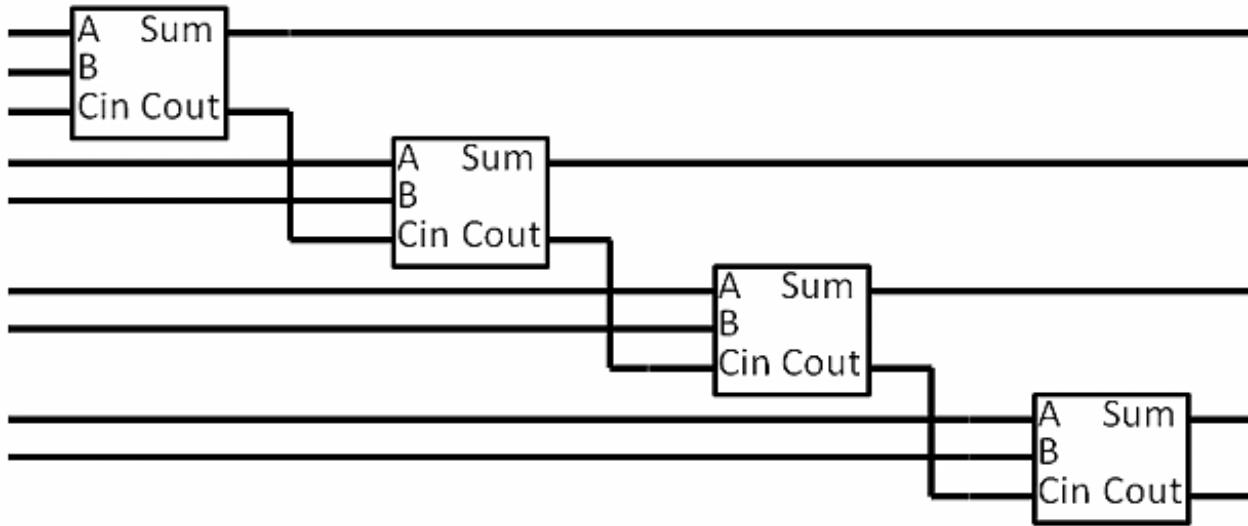
五、全加器的串联

多个全加器串联起来，就能进行二进制的多位运算了。

先把全加器简写成方块形式，注明三个输入（A、B、 C_{in} ）和两个输出（S和 C_{out} ）。



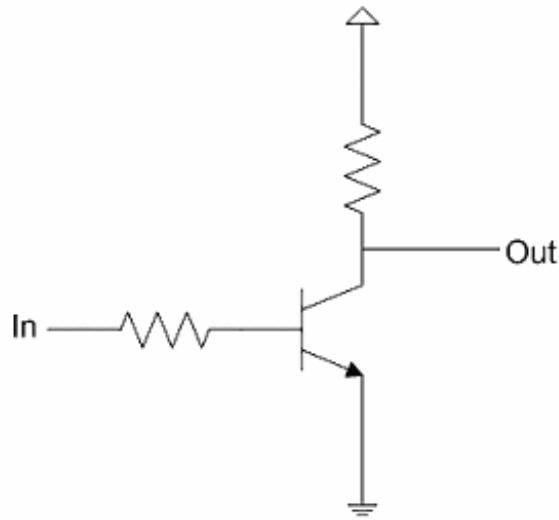
然后，将四个全加器串联起来，就得到了四位加法器的逻辑图。



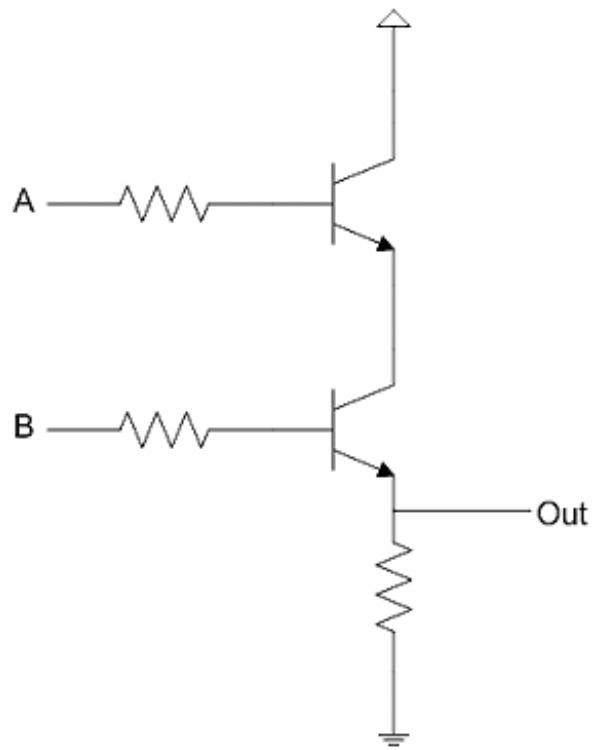
六、逻辑门的晶体管实现

下一步，就是用晶体管做出逻辑门的电路。

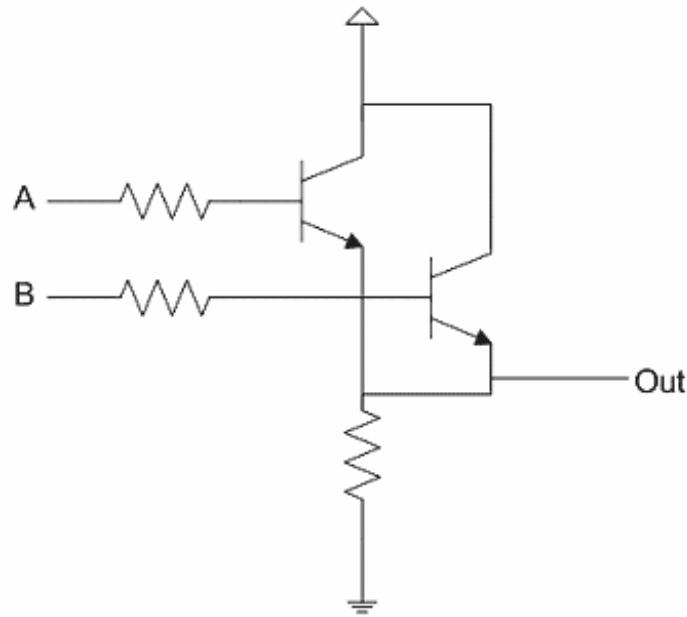
先看NOT。晶体管的基极（Base）作为输入，集电极（collector）作为输出，发射极（emitter）接地。当输入为1（高电平），电流流向发射极，因此输出为0；当输入为0（低电平），电流从集电极流出，因此输出为1。



接着是AND。这需要两个晶体管，只有当两个基极的输入都为1（高电平），电流才会流向输出端，得到1。

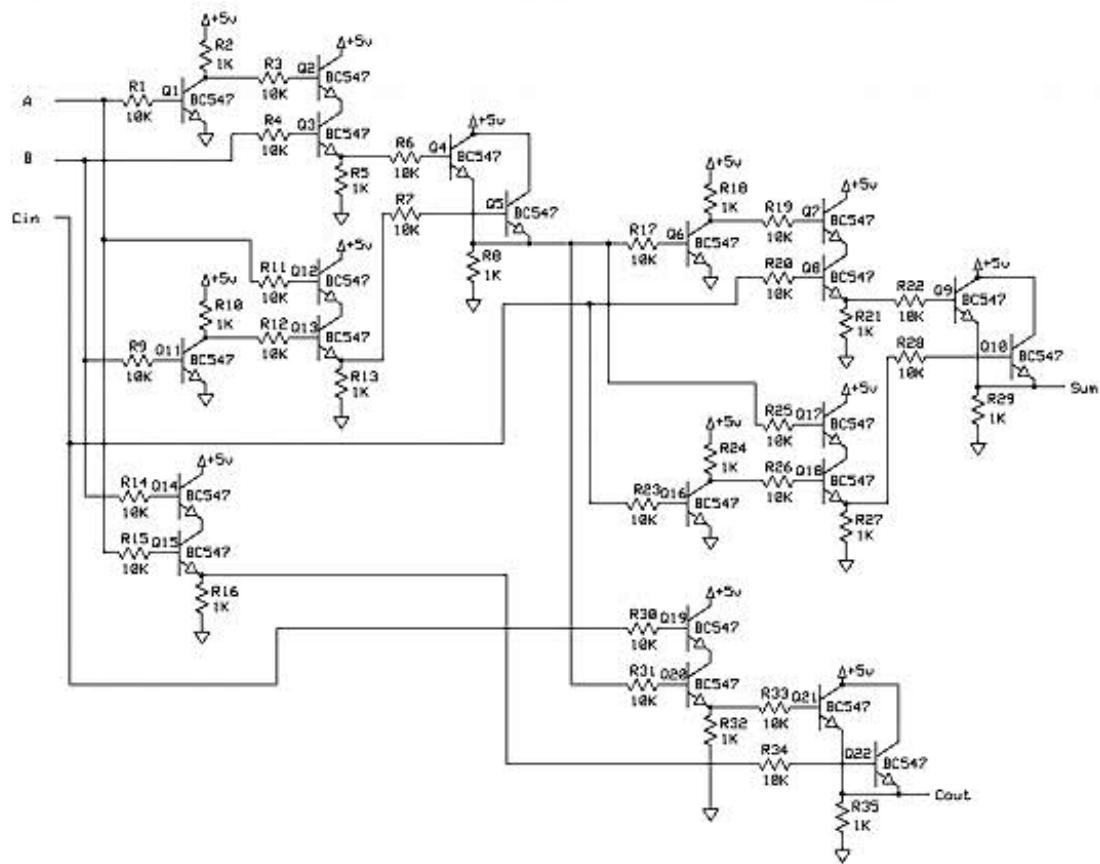


最后是OR。这也需要两个晶体管，只要两个基极中有一个为1（高电平），电流就会流向输出端，得到1。



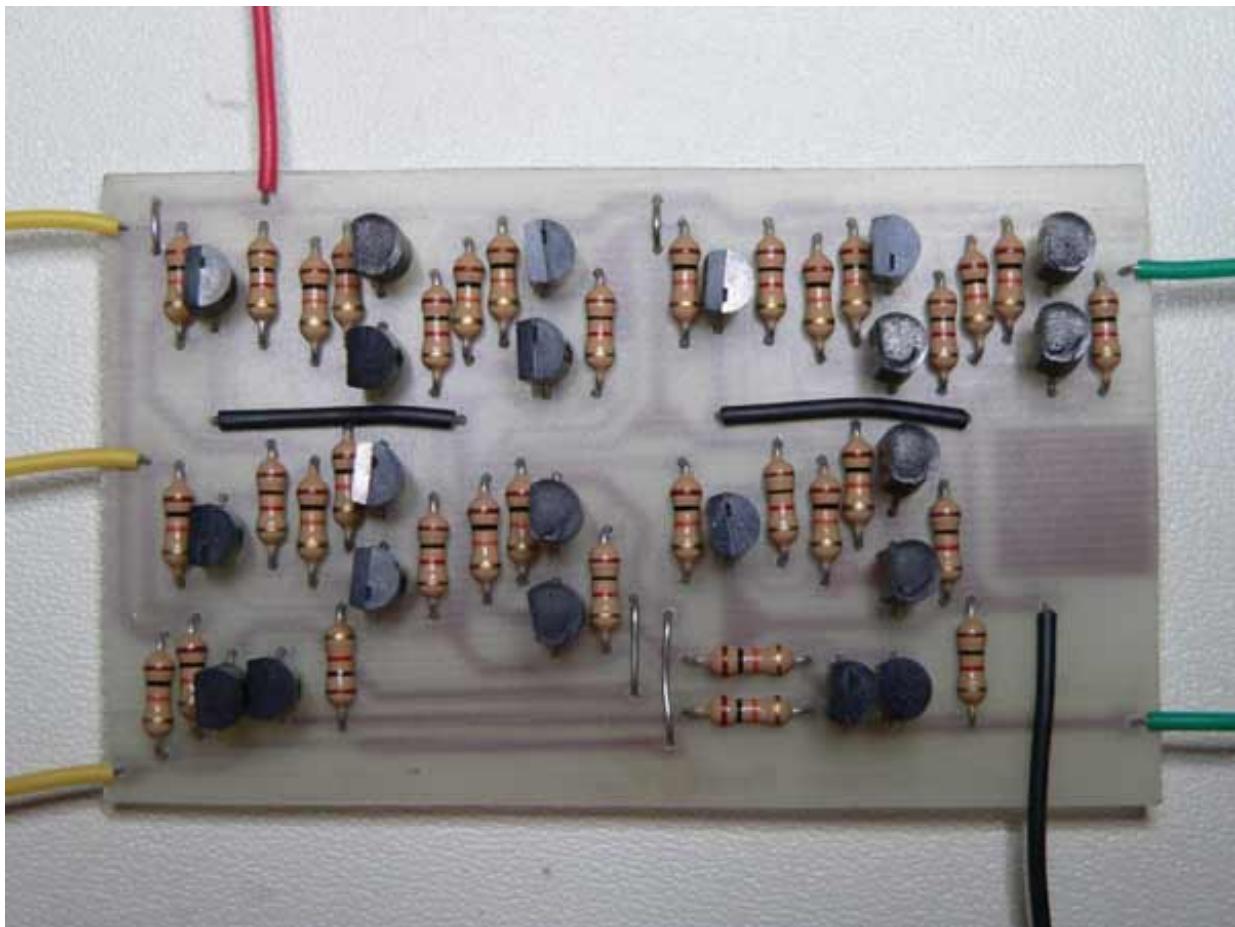
七、全加器的电路

将三种逻辑门的晶体管实现，代入全加器的设计图，就可以画出电路图了。



(点击看大图)

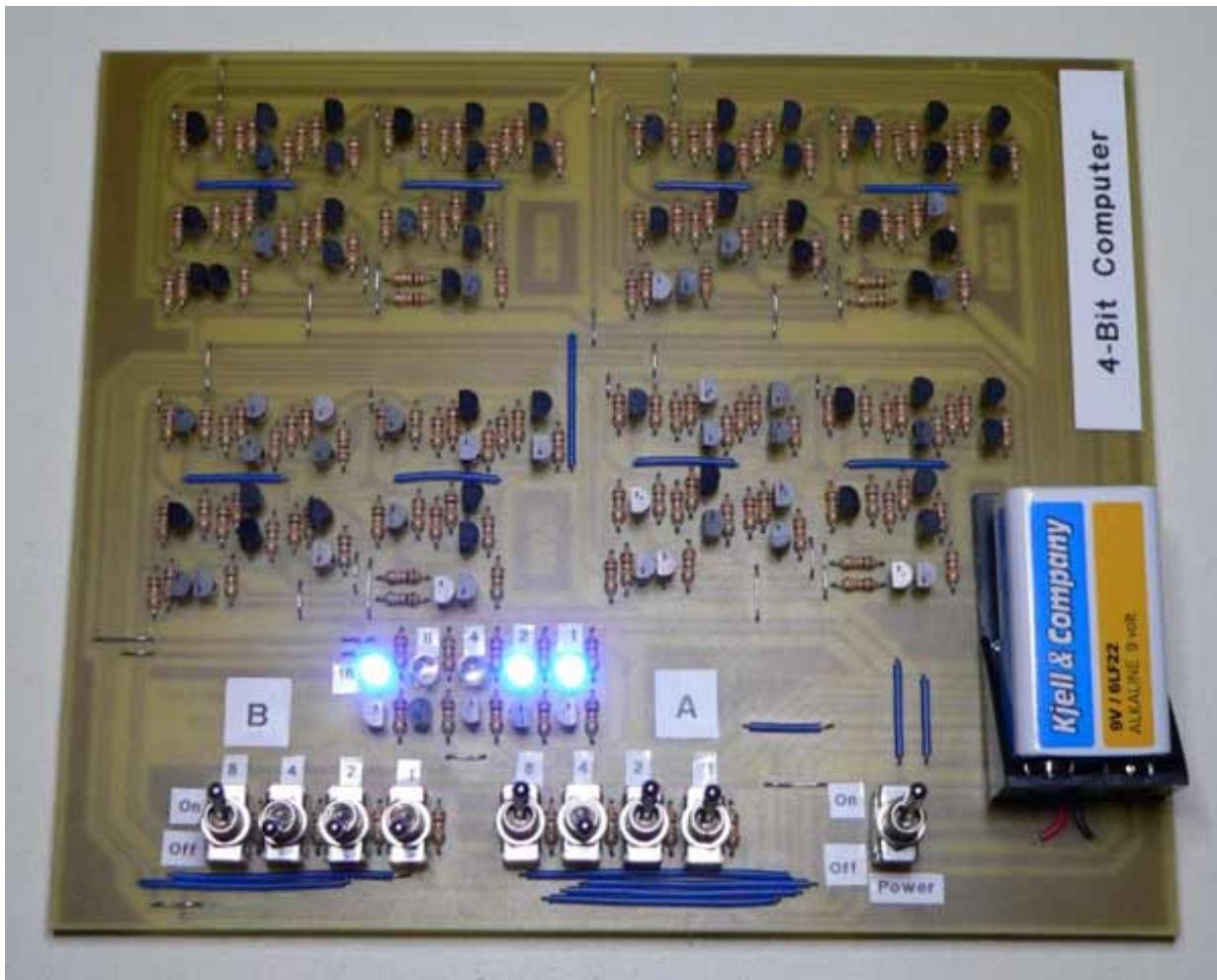
按照电路图，用晶体管和电路板组装出全加器的集成电路。



左边的三根黄线，分别代表三个输入A、B、 C_{in} ；右边的两根绿线，分别代表输出S和 C_{out} 。

八、制作计算机

将四块全加器的电路串联起来，就是一台货真价实的四位晶体管计算机了，可以计算0000~1111之间的加法。



电路板的下方有两组各四个开关，标注着"A"和"B"，代表两个输入数。从上图可以看到，A组开关是"上下上上"，代表1011 (11)；B组开关是"上下下下"，代表1000 (8)。它们的相加结果用五个LED灯表示，上图中是"亮暗暗亮亮"，代表10011 (19)，正是1011与1000的和。

九、结论

虽然这个四位计算机非常简陋，但是从中不难体会到现代计算机的原理。

完成上面的四位加法，需要用到88个晶体管。虽然当代处理器包含的晶体管数以亿计，但是本质上都是上面这样简单电路的累加。

(完)

最常见的电话号码

网上有许多电话号码，你知道最常见的是哪个？

有个[老外](#)找到了答案，这个号码就是2147483647。



事情是这样的，我们知道，在计算机中，整数往往用4个字节保存。4个字节就是32位，由于最高位是[符号位](#)，那么只剩下31位，也就是说，这种方式所能保存的最大整数是 $2^{31}-1$ （因为要去掉零），即2147483647。

2147483647									
0111	1111	1111	1111	1111	1111	1111	1111	1111	31

这个老外突然意识到，这是一个10位数，与美国电话号码的位数相同。于是，他就很好奇，214-748-3647会是谁的号码呢？

214是美国达拉斯市的区号，但是[搜索](#)后发现，超过1500个网站上有这个号码，遍布全美各地。

Ariana Pizza & Pasta Cafe

81

Pizza, Italian Restaurants
7032 Austin St
New York, NY
(214) 748-3647

为什么会这样？

很显然，大量程序员考虑不周，使用4个字节的整数保存电话号码。当用户输入的号码大于2147483647时，就会自动转成这个数字。因此，它就成了网上最常见的电话号码。

=====

我一时兴起，想看看国内有没有人犯这个错误。事实证明，真的是有一大把。

球场名称: 上海国际赛车场

球场类型: F1

电话: 2147483647

邮编:

详细地址: 嘉定区伊宁路2000号(近嘉定汽车城)

万盛家园05年框架房，顶加阁 出售信息 价格:**56万元, 元**

户型: 四室二厅二卫 面积: 146m² 访问(23) | 评论 收藏

地址: 万盛家园 电话: 2147483647

NO. 6 丝路造型美容美发

丝路造型美容美发

电话: 13818939860 21-47483647

地址: 黄陵路263号

简介:

设施齐全、提供各种美容美发项目服务。

上海英孚教育五角场百联

上海市,上海市,宝山区,淞沪路8号,200942

电话: 021-47483647

类别: 教育相关服务业

咨询

+

收藏

[商家信息](#) | [地图](#) |

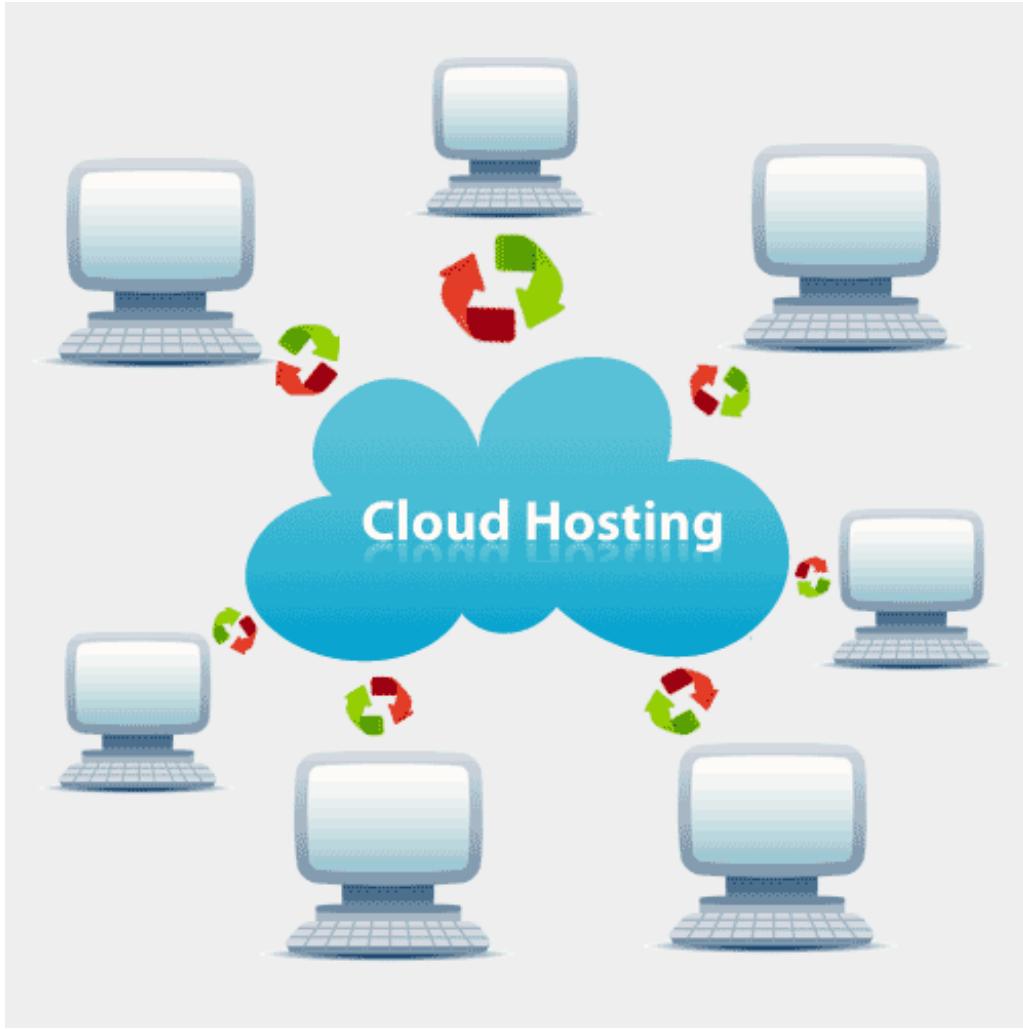
黑龙江上豪格家俱厂供应防静电工作鞋

上海市宝山区月浦镇段泾村西北宅仓库 / 021-47483647-

因为我国的长途区号没有0214，所以这个号码只能是在上海（区号021）。一想到他会接到各种各样的来电，要求提供租车服务、租房服务、美容服务、餐饮服务、耐磨材料等等，我就对021-47483647的主人充满同情。

(完)

云主机是什么？



一、共享主机和云主机

从互联网诞生至今，大部分站长都是从"共享主机"（shared hosting）开始学习建站的。所谓"共享主机"，就是一台服务器上有许多网站，大家共享这台服务器的硬件和带宽。如果它发生故障，那么上面的所有网站都无法访问。

"云主机"（Cloud hosting）可以看成是新一代的共享主机。

首先，主机公司将它的硬件和网络线路，做成一朵"云"，然后提供一些通向这朵"云"的网络接口API，供客户使用。这时，**每个客户共享的不再是某一台特定的服务器，而是云里的所有服务器。**

比如，假设你要把本机的文件备份到网上，你可以使用共享主机，把文件传到某一台服务器上；也可以使用云主机，通过某种形式的接口，把它们传到云里。也就是说，共享主机用户直接面对特定的服务器，而**云主机用户直接面对网络接口，看不到服务器内部。**

一个通俗的比喻是，你可以向银行租一个编号为"8888"的保险箱（共享主机），也可以把贵重物品直接交给保管公司（云主机），听任他们保管。

诸如Gmail、FaceBook、Twitter、Flickr这样的产品，都可以看作是基于"云主机"的服务。

二、云主机的优点

云主机主要有三大优点。

(1) 便宜。

因为服务可以分散到多台服务器，因此能够充分利用资源，这样就降低了硬件、电力和维护成本。而且，云主机是根据使用量计费的，多用多付，少用少付，所以对小网站特别有利。

(2) 可靠。

因为服务分布在多台服务器、甚至多个机房，所以不容易彻底宕机，抗灾容错能力强，可以保证长时间在线。

(3) 可扩展性好 (scalability)。

云主机的基本特点就是分布式架构，所以可以轻而易举地增加服务器，成倍扩展服务能力。

三、云主机的缺点

一些客户担心云主机的安全问题，感到对服务缺乏控制。

因为云主机只是提供网络接口，所以客户的数据必然全部服从云服务公司的安排，完全在后者控制之下。数据是否安全保密，取决于后者的职业道德和保护能力。

但是，这其实是一个"伪问题"，因为绝大多数情况下，云服务公司会比客户更在乎、也更善于保护数据。Paul Graham在《黑客与画家》一书中，就谈过这一点：

"反对者往往觉得我们的产品不安全，如果员工可以很容易地登录，那么坏人也可以很容易地登录。一些大公司觉得不能把客户的信用卡资料交给我们，而是放在自己的服务器上更安全。……但是事实上，他们的服务器就是没我们的安全，我们对数据的保护几乎肯定比他们好。

想想看，谁能雇到更高水平的网络安全专家，是一个所有业务就是管理服务器的技术型创业公司，还是一家服装零售商？……而且我们比他们更关心数据的安全。如果一家服装零售商的服务器被入侵，最多只影响到这家公司本身，这件事也很可能在公司内部被掩盖起来，最严重的情况下可能还会有一个员工被解雇。但是，如果我们的服务器被入侵，就有成千上万家公司可能受到影响，这件事也许还会被当作新闻，发表在业内网站上面，使得我们生意做不下去，不得不关门歇业。

如果你想把钱藏在安全的地方，请问你是选择家中床垫下面，还是选择银行？这个比喻对服务器管理的方方面面都适用，不仅是安全性，还包括正常运行时间、带宽、负载管理、备份等等，都是我们占优。"

四、如何选择云主机

一般来说，知名公司总是优先的选择。目前主要有三家：[Amazon Web Services](#)、[NetDepot](#)和[Rackspace](#)。但是，小公司也有自己的优势，比如满足个性化需求和更低的价格。

你可以根据客户服务、机房分布、可靠性、API的强大程度、安全措施、价格等因素，进行综合考虑。

(完)

为什么Lisp语言如此先进？（译文）

上周，《黑客与画家》总算翻译完成，已经交给出版社了。

翻译完这本书，累得像生了一场大病。把书稿交出去的时候，心里空荡荡的，也不知道自己得到了什么，失去了什么。

希望这个中译本和我的努力，能得到读者认同和肯定。

下面是此书中非常棒的一篇文章，原文写于八年前，至今仍然具有启发性，作者眼光之超前令人佩服。由于我不懂Lisp语言，所以田春同学帮忙校读了一遍，纠正了一些翻译不当之处，在此表示衷心感谢。

=====

为什么Lisp语言如此先进？

作者：Paul Graham

译者：阮一峰

英文原文：[*Revenge of the Nerds*](#)

（节选自即将出版的《黑客与画家》中译本）

一、

如果我们把流行的编程语言，以这样的顺序排列：Java、Perl、Python、Ruby。你会发现，排在越后面的语，越像Lisp。

Python模仿Lisp，甚至把许多Lisp黑客认为属于设计错误的功能，也一起模仿了。至于Ruby，如果回到1975年，你声称它是一种Lisp方言，没有人会反对。

编程语言现在的发展，不过刚刚赶上1958年Lisp语言的水平。

二、

1958年，John McCarthy设计了Lisp语言。我认为，当前最新潮的编程语言，只是实现了他在1958年的设想而已。

这怎么可能呢？计算机技术的发展，不是日新月异吗？1958年的技术，怎么可能超过今天的水平呢？

让我告诉你原因。

这是因为John McCarthy本来没打算把Lisp设计成编程语言，至少不是我们现在意义上的编程语言。他的原意只是想做一种理论演算，用更简洁的方式定义图灵机。

所以，为什么上个世纪50年代的编程语言，到现在还没有过时？简单说，因为这种语言本质上不是一种技术，而是数学。数学是不会过时的。你不应该把Lisp语言与50年代的硬件联系在一起，而是应该把它与快速排序（Quicksort）算法进行类比。这种算法是1960年提出的，至今仍然是最快的通用排序方法。

三、

Fortran语言也是上个世纪50年代出现的，并且一直使用至今。它代表了语言设计的一种完全不同的方向。Lisp是无意中从纯理论发展为编程语言，而Fortran从一开始就是作为编程语言设计出来的。但是，今天我们把Lisp看成高级语言，而把Fortran看成一种相当低层次的语言。

1956年，Fortran刚诞生的时候，叫做Fortran I，与今天的Fortran语言差别极大。Fortran I实际上是汇编语言加上数学，在某些方面，还不如今天的汇编语言强大。比如，它不支持子程序，只有分支跳转结构（branch）。

Lisp和Fortran代表了编程语言发展的两大方向。前者的基础是数学，后者的基础是硬件架构。从那时起，这两大方向一直在互相靠拢。Lisp刚设计出来的时候，就很强大，接下来的二十年，它提高了自己的运行速度。而那些所谓的主流语言，把更快的运行速度作为设计的出发点，然后再用超过四十年的时间，一步步变得更强大。

直到今天，最高级的主流语言，也只是刚刚接近Lisp的水平。虽然已经很接近了，但还是没有Lisp那样强大。

四、

Lisp语言诞生的时候，就包含了9种新思想。其中一些我们今天已经习以为常，另一些则刚刚在其他高级语言中出现，至今还有2种是Lisp独有的。按照被大众接受的程度，这9种思想依次是：

1. 条件结构（即"if-then-else"结构）。现在大家都觉得这是理所当然的，但是Fortran I就没有这个结构，它只有基于底层机器指令的goto结构。
2. 函数也是一种数据类型。在Lisp语言中，函数与整数或字符串一样，也属于数据类型的一种。它有自己的字面表示形式（literal representation），能够储存在变量中，也能当作参数传递。一种数据类型应该有的功能，它都有。
3. 递归。Lisp是第一种支持递归函数的高级语言。
4. 变量的动态类型。在Lisp语言中，所有变量实际上都是指针，所指向的值有类型之分，而变量本身没有。复制变量就相当于复制指针，而不是复制它们指向的数据。
5. 垃圾回收机制。
6. 程序由表达式（expression）组成。Lisp程序是一些表达式区块的集合，每个表达式都返回一个值。这与Fortran和大多数后来的语言都截然不同，它们的程序由表达式和语句（statement）组成。

区分表达式和语句，在Fortran I中是很自然的，因为它不支持语句嵌套。所以，如果你需要用数学公式计算一个值，那就只有用表达式返回这个值，没有其他语法结构可用，因为否则就无法处理这个

值。

后来，新的编程语言支持区块结构（block），这种限制当然也就不存在了。但是为时已晚，表达式和语句的区分已经根深蒂固。它从Fortran扩散到Algol语言，接着又扩散到它们两者的后继语言。

7. 符号（symbol）类型。符号实际上是一种指针，指向储存在哈希表中的字符串。所以，比较两个符号是否相等，只要看它们的指针是否一样就行了，不用逐个字符地比较。

8. 代码使用符号和常量组成的树形表示法（notation）。

9. 无论什么时候，整个语言都是可用的。Lisp并不真正区分读取期、编译期和运行期。你可以在读取期编译或运行代码；也可以在编译期读取或运行代码；还可以在运行期读取或者编译代码。

在读取期运行代码，使得用户可以重新调整（reprogram）Lisp的语法；在编译期运行代码，则是Lisp宏的工作基础；在运行期编译代码，使得Lisp可以在Emacs这样的程序中，充当扩展语言（extension language）；在运行期读取代码，使得程序之间可以用S-表达式（S-expression）通信，近来XML格式的出现使得这个概念被重新“发明”出来了。

五、

Lisp语言刚出现的时候，它的思想与其他编程语言大相径庭。后者的设计思想主要由50年代后期的硬件决定。随着时间流逝，流行的编程语言不断更新换代，语言设计思想逐渐向Lisp靠拢。

思想1到思想5已经被广泛接受，思想6开始在主流编程语言中出现，思想7在Python语言中有所实现，不过似乎没有专用的语法。

思想8可能是最有意思的一点。它与思想9只是由于偶然原因，才成为Lisp语言的一部分，因为它们不属于John McCarthy的原始构想，是由他的学生Steve Russell自行添加的。它们从此使得Lisp看上去很古怪，但也成为了这种语言最独一无二的特点。Lisp古怪的形式，倒不是因为它的语法很古怪，而是因为它根本没有语法，程序直接以解析树（parse tree）的形式表达出来。在其他语言中，这种形式只是经过解析在后台产生，但是Lisp直接采用它作为表达形式。它由列表构成，而列表则是Lisp的基本数据结构。

用一门语言自己的数据结构来表达该语言，这被证明是非常强大的功能。思想8和思想9，意味着你可以写出一种能够自己编程的程序。这可能听起来很怪异，但是对于Lisp语言却是再普通不过。最常用的做法就是使用宏。

术语“宏”在Lisp语言中，与其他语言中的意思不一样。Lisp宏无所不包，它既可能是某样表达式的缩略形式，也可能是一种新语言的编译器。如果你想真正地理解Lisp语言，或者想拓宽你的编程视野，那么你必须学习宏。

就我所知，宏（采用Lisp语言的定义）目前仍然是Lisp独有的。一个原因是是为了使用宏，你大概不得不让你的语言看上去像Lisp一样古怪。另一个可能的原因是，如果你想为自己的语言添上这种终极武器，你从此就不能声称自己发明了新语言，只能说发明了一种Lisp的新方言。

我把这件事当作笑话说出来，但是事实就是如此。如果你创造了一种新语言，其中有car、cdr、cons、quote、cond、atom、eq这样的功能，还有一种把函数写成列表的表示方法，那么在它们的基础上，你完全可以推导出Lisp语言的所有其他部分。事实上，Lisp语言就是这样定义的，John

McCarthy把语言设计成这个样子，就是为了让这种推导成为可能。

六、

就算Lisp确实代表了目前主流编程语言不断靠近的一个方向，这是否意味着你就应该用它编程呢？

如果使用一种不那么强大的语言，你又会有多少损失呢？有时不采用最尖端的技术，不也是一种明智的选择吗？这么多人使用主流编程语言，这本身不也说明那些语言有可取之处吗？

另一方面，选择哪一种编程语言，许多项目是无所谓的，反正不同的语言都能完成工作。一般来说，条件越苛刻的项目，强大的编程语言就越能发挥作用。但是，无数的项目根本没有苛刻条件的限制。大多数的编程任务，可能只要写一些很小的程序，然后用胶水语言把这些小程序连起来就行了。你可以用自己熟悉的编程语言，或者用对于特定项目来说有着最强大函数库的语言，来写这些小程序。如果你只是需要在Windows应用程序之间传递数据，使用Visual Basic照样能达到目的。

那么，Lisp的编程优势体现在哪里呢？

七、

语言的编程能力越强大，写出来的程序就越短（当然不是指字符数量，而是指独立的语法单位）。

代码的数量很重要，因为开发一个程序耗费的时间，主要取决于程序的长度。如果同一个软件，一种语言写出来的代码比另一种语言长三倍，这意味着你开发它耗费的时间也会多三倍。而且即使你多雇佣人手，也无助于减少开发时间，因为当团队规模超过某个门槛时，再增加人手只会带来净损失。Fred Brooks在他的名著《人月神话》（The Mythical Man-Month）中，描述了这种现象，我的所见所闻印证了他的说法。

如果使用Lisp语言，能让程序变得多短？以Lisp和C的比较为例，我听到的大多数说法是C代码的长度是Lisp的7倍到10倍。但是最近，New Architect杂志上有一篇介绍ITA软件公司的文章，里面说“一行Lisp代码相当于20行C代码”，因为此文都是引用ITA总裁的话，所以我想这个数字来自ITA的编程实践。如果真是这样，那么我们可以相信这句话。ITA的软件，不仅使用Lisp语言，还同时大量使用C和C++，所以这是他们的经验谈。

根据上面的这个数字，如果你与ITA竞争，而且你使用C语言开发软件，那么ITA的开发速度将比你快20倍。如果你需要一年时间实现某个功能，它只需要不到三星期。反过来说，如果某个新功能，它开发了三个月，那么你需要五年才能做出来。

你知道吗？上面的对比，还只是考虑到最好的情况。当我们只比较代码数量的时候，言下之意就是假设使用功能较弱的语言，也能开发出同样的软件。但是事实上，程序员使用某种语言能做到的事情，是有极限的。如果你想用一种低层次的语言，解决一个很难的问题，那么你将会面临各种情况极其复杂、乃至想不清楚的窘境。

所以，当我说假定你与ITA竞争，你用五年时间做出的东西，ITA在Lisp语言的帮助下只用三个月就完成了，我指的五年还是一切顺利、没有犯错误、也没有遇到太大麻烦的五年。事实上，按照大多数公司的实际情况，计划中五年完成的项目，很可能永远都不会完成。

我承认，上面的例子太极端。ITA似乎有一批非常聪明的黑客，而C语言又是一种很低层次的语言。但是，在一个高度竞争的市场中，即使开发速度只相差两三倍，也足以使得你永远处在落后的位置。

附录：编程能力

为了解释我所说的语言编程能力不一样，请考虑下面的问题。我们需要写一个函数，它能够生成累加器，即这个函数接受一个参数n，然后返回另一个函数，后者接受参数i，然后返回n增加(increment)了i后的值。

Common Lisp的写法如下：

```
(defun foo (n)
  (lambda (i) (incf n i)))
```

Ruby的写法几乎完全相同：

```
def foo (n)
  lambda {|i| n += i } end
```

Perl 5的写法则是：

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```

这比Lisp和Ruby的版本，有更多的语法元素，因为在Perl语言中，你不得不手工提取参数。

Smalltalk的写法稍微比Lisp和Ruby的长一点：

```
foo: n
|s|
s := n.
^[:i| s := s+i.]
```

因为在Smalltalk中，局部变量 (lexical variable) 是有效的，但是你无法给一个参数赋值，因此不得不设置了一个新变量，接受累加后的值。

Javascript的写法也比Lisp和Ruby稍微长一点，因为Javascript依然区分语句和表达式，所以你需要明确指定return语句，来返回一个值：

```
function foo (n) {
  return function (i) {
    return n += i } }
```

(实事求是地说，Perl也保留了语句和表达式的区别，但是使用了典型的Perl方式处理，使你可以省略return。)

如果想把Lisp/Ruby/Perl/Smalltalk/Javascript的版本改成Python，你会遇到一些限制。因为Python并不完全支持局部变量，你不得不创造一种数据结构，来接受n的值。而且尽管Python确实支持函数数据类型，但是没有一种字面量的表示方式 (literal representation) 可以生成函数 (除非函数体只有一个表达式)，所以你需要创造一个命名函数，把它返回。最后的写法如下：

```
def foo (n):
    s = [n]
    def bar (i):
        s[0] += i
        return s[0]
    return bar
```

Python用户完全可以合理地质疑，为什么不能写成下面这样：

```
def foo (n):
    return lambda i: return n += i
```

或者：

```
def foo (n):
    lambda i: n += i
```

我猜想，Python有一天会支持这样的写法。（如果你不想等到Python慢慢进化到更像Lisp，你总是可以直接……）

在面向对象编程的语言中，你能够在有限程度上模拟一个闭包（即一个函数，通过它可以引用由包含这个函数的代码所定义的变量）。你定义一个类（class），里面有一个方法和一个属性，用于替换封闭作用域（enclosing scope）中的所有变量。这有点类似于让程序员自己做代码分析，本来这应该是由支持局部作用域的编译器完成的。如果有多个函数，同时指向相同的变量，那么这种方法就会失效，但是在这个简单的例子中，它已经足够了。

Python高手看来也同意，这是解决这个问题的比较好的方法，写法如下：

```
def foo (n):
    class acc:
        def __init__(self, s):
            self.s = s
        def inc (self, i):
            self.s += i
            return self.s
    return acc (n).inc
```

或者

```
class foo:
    def __init__(self, n):
        self.n = n
    def __call__(self, i):
        self.n += i
        return self.n
```

我添加这一段，原因是想避免Python爱好者说我误解这种语言。但是，在我看来，这两种写法好像都比第一个版本更复杂。你实际上就是在做同样的事，只不过划出了一个独立的区域，保存累加器函

数，区别只是保存在对象的一个属性中，而不是保存在列表（list）的头（head）中。使用这些特殊的内部属性名（尤其是`__call__`），看上去并不像常规的解法，更像是一种破解。

在Perl和Python的较量中，Python黑客的观点似乎是认为Python比Perl更优雅，但是这个例子表明，最终来说，编程能力决定了优雅。Perl的写法更简单（包含更少的语法元素），尽管它的语法有一点丑陋。

其他语言怎么样？前文曾经提到过Fortran、C、C++、Java和Visual Basic，看上去使用它们，根本无法解决这个问题。Ken Anderson说，Java只能写出一个近似的解法：

```
public interface Inttoint {  
    public int call (int i);  
}  
  
public static Inttoint foo (final int n) {  
    return new Inttoint () {  
        int s = n;  
        public int call (int i) {  
            s = s + i;  
            return s;  
        };  
    };  
}
```

这种写法不符合题目要求，因为它只对整数有效。

当然，我说使用其他语言无法解决这个问题，这句话并不完全正确。所有这些语言都是图灵等价的，这意味着严格地说，你能使用它们之中的任何一种语言，写出任何一个程序。那么，怎样才能做到这一点呢？就这个小小例子而言，你可以使用这些不那么强大的语言，写一个Lisp解释器就行了。

这样做听上去好像开玩笑，但是在大型编程项目中，却不同程度地广泛存在。因此，有人把它总结出来，起名为"格林斯潘第十定律"（Greenspun's Tenth Rule）：

"任何C或Fortran程序复杂到一定程度之后，都会包含一个临时开发的、只有一半功能的、不完全符合规格的、到处都是bug的、运行速度很慢的Common Lisp实现。"

如果你想解决一个困难的问题，关键不是你使用的语言是否强大，而是好几个因素同时发挥作用

(a) 使用一种强大的语言，(b) 为这个难题写一个事实上的解释器，或者(c) 你自己变成这个难题的人肉编译器。在Python的例子中，这样的处理方法已经开始出现了，我们实际上就是自己写代码，模拟出编译器实现局部变量的功能。

这种实践不仅很普遍，而且已经制度化了。举例来说，在面向对象编程的世界中，我们大量听到"模式"（pattern）这个词，我觉得那些"模式"就是现实中的因素(c)，也就是人肉编译器。当我在自己的程序中，发现用到了模式，我觉得这就表明某个地方出错了。程序的形式，应该仅仅反映它所要解决的问题。代码中其他任何外加的形式，都是一个信号，（至少对我来说）表明我对问题的抽象还不够深，也经常提醒我，自己正在手工完成的事情，本应该写代码，通过宏的扩展自动实现。

(完)

浮点数的二进制表示

1.

前几天，我在读一本C语言教材，有一道例题：

```
#include <stdio.h>

void main(void){

    int num=9; /* num是整型变量，设为9 */

    float* pFloat=&num; /* pFloat表示num的内存地址，但是设为浮点数 */

    printf("num的值为： %d\n",num); /* 显示num的整型值 */

    printf("*pFloat的值为： %f\n",*pFloat); /* 显示num的浮点值 */

    *pFloat=9.0; /* 将num的值改为浮点数 */

    printf("num的值为： %d\n",num); /* 显示num的整型值 */

    printf("*pFloat的值为： %f\n",*pFloat); /* 显示num的浮点值 */

}
```

运行结果如下：

```
num的值为： 9
*pFloat的值为： 0.000000
num的值为： 1091567616
*pFloat的值为： 9.000000
```

我很惊讶，num和*pFloat在内存中明明是同一个数，为什么浮点数和整数的解读结果会差别这么大？

要理解这个结果，一定要搞懂浮点数在计算机内部的表示方法。我读了一些资料，下面就是我的笔记。

2.

在讨论浮点数之前，先看一下整数在计算机内部是怎样表示的。

```
int num=9;
```

上面这条命令，声明了一个整数变量，类型为int，值为9（二进制写法为1001）。普通的32位计算机，用4个字节表示int变量，所以9就被保存为00000000 00000000 00000000 00001001，写成16进制就是0x00000009。

那么，我们的问题就简化成：**为什么0x00000009还原成浮点数，就成了0.000000？**

3.

根据国际标准IEEE 754，任意一个二进制浮点数V可以表示成下面的形式：

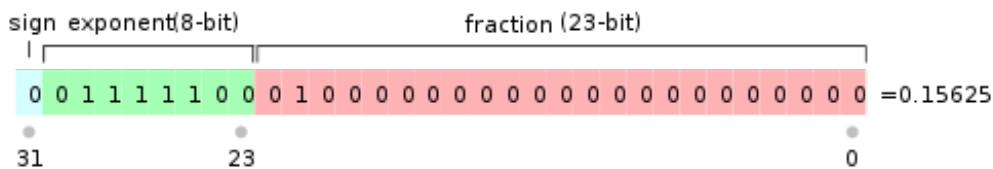
$$V = (-1)^s \times M \times 2^E$$

- (1) $(-1)^s$ 表示符号位，当 $s=0$, V 为正数；当 $s=1$, V 为负数。
 - (2) M 表示有效数字，大于等于 1，小于 2。
 - (3) 2^E 表示指数位。

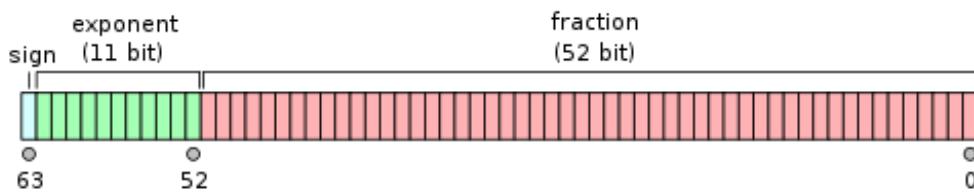
举例来说，十进制的5.0，写成二进制是101.0，相当于 1.01×2^2 。那么，按照上面V的格式，可以得出s=0，M=1.01，E=2。

十进制的-5.0，写成二进制是-101.0，相当于 -1.01×2^2 。那么，s=1，M=1.01，E=2。

IEEE 754规定，对于32位的浮点数，最高的1位是符号位s，接着的8位是指数E，剩下的23位为有效数字M。



对于64位的浮点数，最高的1位是符号位S，接着的11位是指数E，剩下的52位为有效数字M。



5.

IEEE 754对有效数字M和指数E，还有一些特别规定。

前面说过， $1 \leq M < 2$ ，也就是说，M可以写成 $1.xxxxxx$ 的形式，其中 $xxxxxx$ 表示小数部分。**IEEE 754规定，在计算机内部保存M时，默认这个数的第一位总是1，因此可以被舍去，只保存后面的xxxxxx部分。**比如保存1.01的时候，只保存01，等到读取的时候，再把第一位的1加上去。这样做的目的，是节省1位有效数字。以32位浮点数为例，留给M只有23位，将第一位的1舍去以后，等于可以保存24位有效数字。

至于指数E，情况就比较复杂。

首先，E为一个无符号整数（unsigned int）。这意味着，如果E为8位，它的取值范围为0~255；如果E为11位，它的取值范围为0~2047。但是，我们知道，科学计数法中的E是可以出现负数的，所以

IEEE 754规定，E的真实值必须再减去一个中间数，对于8位的E，这个中间数是127；对于11位的E，这个中间数是1023。

比如， 2^{10} 的E是10，所以保存成32位浮点数时，必须保存成 $10+127=137$ ，即10001001。

然后，指数E还可以再分成三种情况：

(1) E不全为0或不全为1。这时，浮点数就采用上面的规则表示，即指数E的计算值减去127（或1023），得到真实值，再将有效数字M前加上第一位的1。

(2) E全为0。这时，浮点数的指数E等于1-127（或者1-1023），有效数字M不再加上第一位的1，而是还原为0.xxxxxx的小数。这样做是为了表示 ± 0 ，以及接近于0的很小的数字。

(3) E全为1。这时，如果有效数字M全为0，表示 \pm 无穷大（正负取决于符号位s）；如果有效数字M不全为0，表示这个数不是一个数（NaN）。

6.

好了，关于浮点数的表示规则，就说到这里。

下面，让我们回到一开始的问题：**为什么0x00000009还原成浮点数，就成了0.000000？**

首先，将0x00000009拆分，得到第一位符号位s=0，后面8位的指数E=00000000，最后23位的有效数字M=000 0000 0000 0000 0000 1001。

由于指数E全为0，所以符合上一节的第二种情况。因此，浮点数V就写成：

$$V=(-1)^0 \times 0.00000000000000000000000001001 \times 2^{-126} = 1.001 \times 2^{-146}$$

显然，V是一个很小的接近于0的正数，所以用十进制小数表示就是0.000000。

7.

再看例题的第二部分。

请问浮点数9.0，如何用二进制表示？还原成十进制又是多少？

首先，浮点数9.0等于二进制的1001.0，即 1.001×2^3 。

那么，第一位的符号位s=0，有效数字M等于001后面再加20个0，凑满23位，指数E等于 $3+127=130$ ，即10000010。

所以，写成二进制形式，应该是s+E+M，即0 10000010 001 0000 0000 0000 0000。这个32位的二进制数，还原成十进制，正是1091567616。

(完)

几种计算机语言的评价（修订版）

编程新手都有一个同样的问题：“我应该学习哪一种语言？”。

《Unix编程艺术》（Eric Raymond著）第十四章，对各种语言进行了评价，正好可以用来回答这个问题。下面是我的笔记，整理了六种主要计算机语言的优缺点。读完就会知道，对于不同的任务，应该选择哪一种语言了。

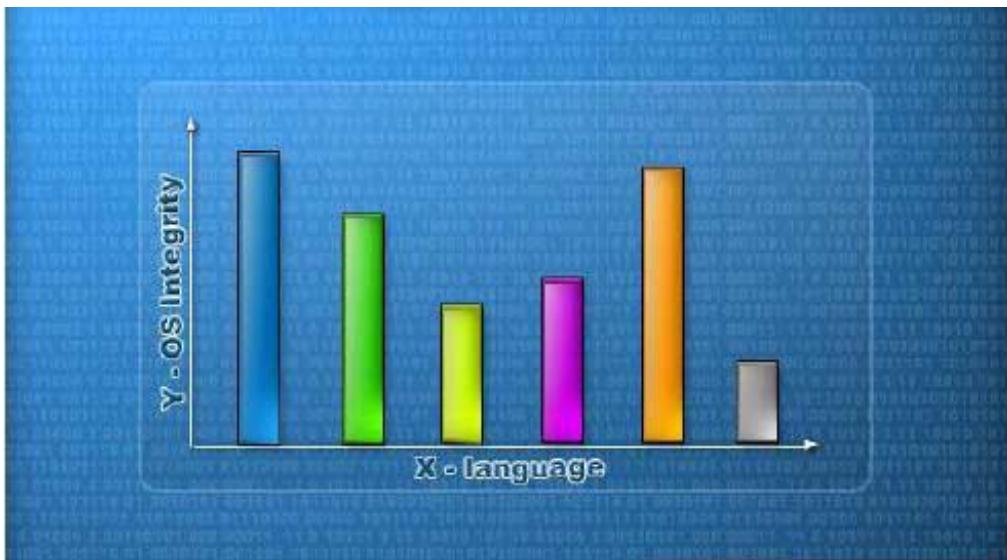
原文写于2003年。网上曾经有一个不完整的中译本，我也在这个网志中[转贴过](#)，所以今天贴的只能算修订版了。

=====

几种计算机语言的评价

作者：Eric Raymond

摘自《Unix编程艺术（第十四章）》



一、C

C语言的优点是，运行效率高和接近机器语言。它特别适用于以下几种程序：

1. 对运行速度要求很高的程序。
2. 与操作系统内核关系密切的程序。
3. 必须在多个操作系统上移植的程序。

除此之外，学习C语言有一个最大的理由，那就是它能帮助我们学会，在硬件层次上考虑问题。如果你想以编程作为自己的人生事业，C是一定要学习的。

C的最大缺点是，在编程过程中，你不得不花很多时间，考虑与你要解决的问题完全无关、且非常复杂的硬件资源管理问题。

二、C++

C++在80年代中期推出，支持OO（面向对象编程），原意是作为C语言的取代者。

但是它没能做到做一点，原因有以下几个：

1. 它也没有解决“内存管理”问题，虽然比C有所改善。
2. 标准化不成功。各个编译器都只支持自己选择的一个子集，导致跨平台性不如C。
3. 过分的精细和复杂了。C++的复杂度，超过了C和OO的复杂度之和。
4. OO并没有带来多少优势，反而带来了一些副作用，比如厚重的胶合层和庞大的代码体积。

总的来说，C++的优势还是程序效率，以及面向对象编程能力，糟糕之处是它鼓励复杂的设计。

三、Shell

Shell程序写起来很容易，对于简单的任务，开发速度很快。

当程序变大时，它就不太适合了，很难维护，而且将变得非常专用（只能在你自己的机器上使用），因为 Shell必须调用各种外部程序，无法保证每一台机器都同样安装了这些程序。

最常见的Shell是bash，它的一些语法规则很混乱，会产生很难阅读的代码。另外，shell只能在Unix上使用，无法跨平台。

Shell的最佳用途如下：

1. 执行简单的系统管理任务。
2. 编写系统启动时的初始化脚本。
3. 作为其他语言开发的程序的包装器。

四、Perl

Perl发布于1987年，基本上就是一个增强的Shell。

它的最大长处是强劲的文本处理能力，无以伦比的正则表达式支持，而且有全套Unix API的内部支持，显著减少了对C的需求。

Perl的主要缺点是某些部分设计得非常丑陋，且无法补救，某些部分也过于复杂。当程序规模增大时，必须严格遵守约定，才能保证模块化和设计的可控性，难于维护。

五、Python

Python发布于1991年，是一种与C语言紧密集成的脚本语言。

Python的优点是非常干净优雅，鼓励清晰易读的代码，易学易用；提供了面向对象编程的可能，但并不把这个选择强加于设计者；具有出色的模块化特性，同Java一样，适合用来做需要协同开发的大型

复杂项目。在很多方面，它都比Java简单。此外，Python标准中包括了对许多网络协议的支持，因此也很适合网络管理任务。

Python的缺点主要是效率低下，速度缓慢。在执行速度上，它不仅无法与C/C++竞争，而且也不如其他主要的脚本语言。但是，这其实并不是一个严重的问题，有时网络或磁盘的延迟，会完全抵消Python本身消耗的时间。而且因为Python特别容易和C结合起来，因此性能关键的Python模块，可以很方便地转化成C语言来提高速度。

总的来说，对于小型项目和大量依靠正则表达式的项目，Python不如Perl的表达能力强。至于更小的项目，Python则是大材小用，shell也许更适合。

六、Java

Java发布于1995年，设计目标有两个。

一个是"write once, run anywhere"（一次编写，到处运行），即不依赖于特定的平台；另一个是在网页中嵌入交互程序（applet），可以在任何一个浏览器中运行。由于它的所有者Sun公司的一系列失误，第一个目标并没有完全实现，第二个目标则是彻底失败。但是Java依然在系统编程和应用编程方面非常强大，足以挑战C和C++。

Java的优点是比C++小巧简单，可以自动管理内存，支持类似C的语法和OO编程，与C程序的结合也很好。

Java的缺点是某些部分过于复杂，比如内部类和匿名类的运用会产生很混乱费解的代码；某些部分功能不完善，也无法利用操作系统提供的功能接口，比如在Java中读取和处理文本文件，并不像其他语言那样容易。此外，Java配置环境和版本的混乱，也让人很头疼。

总的来说，除了系统编程和某些对运行速度要求很高的编程之外，Java都是比C++更好的选择。如果和Python相比，Java可能在大型项目上有优势，但是也不是绝对的。

（完）

Unix版权史

1.

这几天，我在读《Unix编程艺术》。



书中介绍了Unix的发展历史。我发现，这是一个很好的例子，说明现行版权制度具有阻碍社会发展的负面作用。

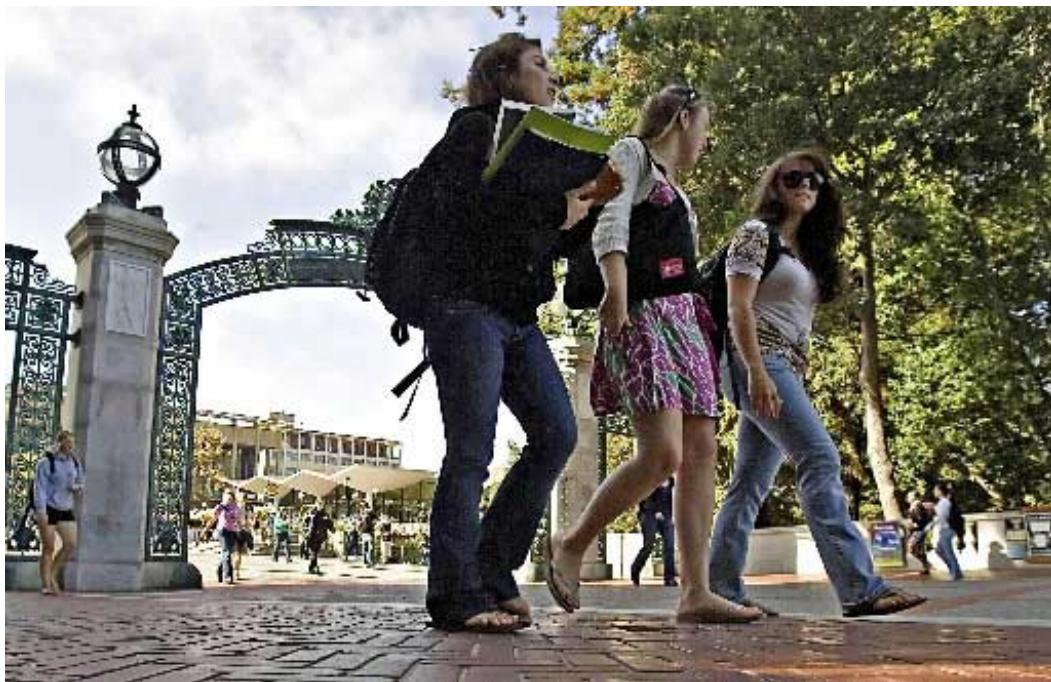
2.

[Unix](#)诞生于1969年，是贝尔实验室员工Ken Thompson的个人项目。由于贝尔实验室是AT&T（美国电话电报公司）的下属机构，所以Unix的版权归AT&T所有。



AT&T垄断了美国长途电话业务，所以美国司法部在1958年与它签了一个和解协议。AT&T同意不进入计算机业，不销售任何与计算机有关的产品，以避免司法部起诉它违反《反垄断法》。Unix是计算机的操作系统，所以AT&T不能销售它，任何要求得到源码的机构，都能免费得到。

加州大学伯克利分校得到源码后，为Unix添加了许多功能。然后在1979年，推出了一个自家的Unix版本，取名为Berkeley Software Distribution（伯克利软件套件），简称BSD。



3.

正当Unix蓬勃发展之际，发生了一件谁也没有想到的事情。

1974年，美国司法部再次起诉AT&T违反《反垄断法》。1982年，哥伦比亚地区法庭判决AT&T败

诉，必须被拆成8家小公司。但是，这个判决也意味着1958年的和解协议失效，AT&T从此可以进入计算机业。

1983年，AT&T发布了Unix最新版System V，这是一个商业化版本，付费才能使用，并且不得传播源码。这个决定对BSD构成了限制，为了减少纠纷，伯克利分校规定，BSD本身依然保持免费，但是只能提供给持有AT&T源码许可的公司。不过，与此同时，伯克利的师生也开始着手另一项工作：将AT&T的专有代码从BSD中逐渐去除。

80年代后期，几个伯克利毕业的学生，成立了一家Berkeley Software Design Inc.公司，简称BSDi，专门销售BSD的一个商业版本。他们在广告中宣称，自己的产品不包含任何AT&T代码。这句话惹恼了AT&T，1990年BSDi被告上法庭，稍后伯克利分校也被追加为被告。AT&T起诉BSD侵犯了Unix的版权。

这场诉讼对BSD打击极大，所有的开发活动都被迫停止，用户人心惶惶，担心自己也遭到AT&T的追究，因此BSD的使用急剧减少。最后在1994年，双方达到和解，BSD才恢复开发。

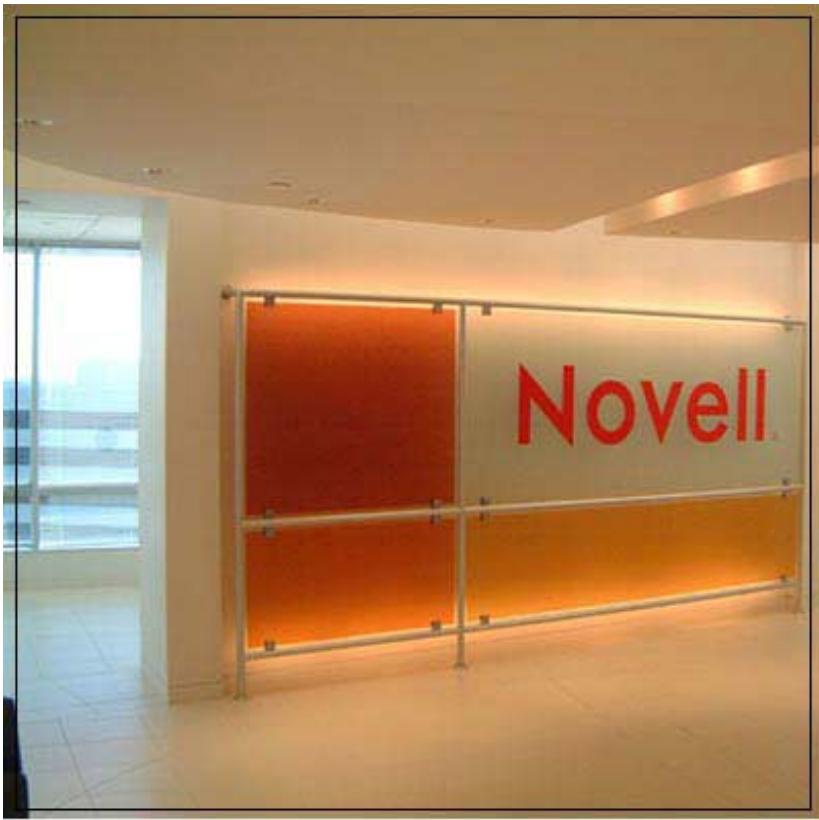
4.

AT&T与BSD之间的诉讼，是当代版权制度最恶劣的应用之一。

为什么这么说？

首先，起诉者其实与Unix毫无关系。这是AT&T经理层的决定，而不是开发者的决定。事实上，包括Ken Thompson在内的技术人员一直希望，公司能够公开源码。他们完全有理由这么要求，因为Unix从来不是AT&T的业务重点，最初是个人项目，后来也没有占用公司太多资源。销售Unix的利润，在公司全部业务中，几乎可以忽略不计。为了一点点钱，去打击一个使许多人受益的产品，何必这样做呢。

其次，AT&T根本不关心Unix的发展。它真正关心的是金钱和削弱对手。1994年，官司还没有结束，它就把Unix卖给了Novell公司，从此不再与Unix发生关系，官司也因此不了了之。既然你不想要这个产品，为什么要提起诉讼呢？真是不可理解。



最后，所谓的侵权几乎是不存在的。因为Novell从AT&T买下Unix版权后，检查了BSD的源码，在18000个组成文件中删除了3个，并对其他文件做了一些小修改，然后BSD就重新获得了自由发布源码的许可。这意味着，至多只有千分之一的BSD代码有版权问题，但是就因为这千分之一的问题，导致百分之百的产品被迫中断，完全不符合比例原则。

所以，这场版权官司就是一家利益至上的公司，以微不足道的理由，为了一个自己根本不在乎的产品，悍然发动一场损人不利己的战争。

5.

这场战争给Unix和BSD带来毁灭性的打击。

从八十年代中后期开始，AT&T固执地捍卫Unix版权，完全不顾它的创造者和开发者的愿望，导致Unix丧失活力、一蹶不振，大量开发者无法参与，只好离开了这个平台。

而BSD在1992~1994年期间，开发处于停滞，错过了发展的黄金时机。官司结束以后，又不幸发生分裂，变成了FreeBSD、NetBSD和OpenBSD三个版本。这些原因导致BSD直到今天，都还在操作系统的竞争中处在落后地位。

如果换个时间，官司的损失也许还没这么大。偏偏90年代初是计算机工业决定性的年代，错过了那几年，从此你就不要想翻身了。因为从80年代末期开始，Intel的80x86芯片有巨大的发展，性能快速上升，而成本快速下降，个人电脑的年代就要到来了。市场迫切需要能够运行在386芯片上的操作系统，但是Unix和BSD忙于打官司，都没有去做移植操作系统这件事。其他两个这样做的人，改变了人类历史。

一个是比尔·盖茨，他推出了Windows，占领了个人电脑市场，后来赚了几百亿美元。另一个是芬兰大学生Linus Torvalds，他想学习Unix，但是买不起工作站，就自己写了一个能在386上运行的Linux

操作系统，现在全世界超过一半的网络服务器都在使用这个系统。Linus Torvalds后来说，如果他早知道BSD没有法律问题，并且可以被移植到386，他就会加入BSD的开发，而不是自己写一个。

我们不禁要问，如果AT&T不打版权官司、不对Unix收费的话，会发生什么事？……人类的历史、市场的格局也许都会被改写。

6.

Novell买到Unix版权后，也没在手里放多久，1995年又转手卖给了别人。从此，Unix原始版本的开发正式结束。

以后的发展集中在两个方向，一个是各个商业公司自己修改的Unix版本，比如Sun的Solaris，HP的HP-UX，IBM的AIX，另一个则是开源项目的开发，比如BSD和Linux。

(完)

Web service是什么？

我认为，下一代互联网软件将建立在Web service（也就是“云”）的基础上。

我把学习笔记和学习心得，放到网志上，欢迎指正。

今天先写一个最基本的问题，Web service到底是什么？

一、Web service的概念

想要理解Web service，必须先理解什么是Service（服务）。

传统上，我们把计算机后台程序（Daemon）提供的功能，称为“服务”（service）。比如，让一个杀毒软件在后台运行，它会自动监控系统，那么这种自动监控就是一个“服务”。通俗地说，“服务”就是计算机可以提供的某一种功能。

根据来源的不同，“服务”又可以分成两种：一种是“本地服务”（使用同一台机器提供的服务，不需要网络），另一种是“网络服务”（使用另一台计算机提供的服务，必须通过网络才能完成）。

举例来说，我现在有一批图片，需要把它们的大小缩小一半。那么，我们可以把“缩放图片”看成是一种服务。你可以使用“本地服务”，在自己计算机上用软件缩小图片，也可以使用“网络服务”，将图片上传到某个网站，让服务器替你缩小图片，完成后再通过网络送回给你。这就好比，一件事你可以自己做，也可以交给另一个人去做。肚子饿了，你可以自己做饭，也可以打电话去订一份比萨，让店家替你做好送上门。

“网络服务”（Web Service）的本质，就是通过网络调用其他网站的资源。

举例来说，去年我写过一个“[四川大地震图片墙](#)”，它能动态显示关于四川地震的最新图片。但是，所有的图片都不是储存在我的服务器上，而是来自[flickr.com](#)。我只是发出一个动态请求，要求flickr.com向我提供图片。这种情况下，flickr.com提供的就是一种Web service。如果我把图片都存放在本地服务器，不调用flickr.com，那么我就是在使用“本地服务”。

所以，Web service让你的网站可以使用其他网站的资源，比如在网页上显示天气、地图、twitter上的最新动态等等。

二、Web Service架构和云

如果一个软件的主要部分采用了“网络服务”，即它把存储或计算环节“外包”给其他网站了，那么我们就说这个软件属于Web Service架构。

Web Service架构的基本思想，就是尽量把非核心功能交给其他人去做，自己全力开发核心功能。比如，如果你要开发一个相册软件，完全可以使用Flickr的网络服务，把相片都储存到它上面，你只要全力做好相册本身就可以了。总体上看，凡是不属于你核心竞争力的功能，都应该把它“外包”出去。

最近很红的“云计算”（cloud computing）或者“云服务”（cloud services），实际上就是Web Service的同义词，不过更形象一些罢了。它们不说你把事情交给其他计算机去做，而说你把事情交给“云”去做。

三、本地服务的缺陷

"网络服务"是未来软件开发和使用的趋势，本地服务将用得越来越少，主要因为以下三个原因：

- * 本地资源不足。很多数据和资料，本地得不到，只有向其他网站要。
- * 成本因素。本地提供服务，往往是不经济的，使用专业网站的服务更便宜。这里面涉及硬件和人员两部分，即使你买得起硬件，专门找一个人管理系统，也是很麻烦的事。
- * 可移植性差。如果你想把本机的服务，移植到其他机器上，往往很困难，尤其是在跨平台的情况下。

四、Web Service的优势

除了本地服务的缺点以外，Web Service还有以下的优越性：

- * 平台无关。不管你使用什么平台，都可以使用Web service。
- * 编程语言无关。只要遵守相关协议，就可以使用任意编程语言，向其他网站要求Web service。这大大增加了web service的适用性，降低了对程序员的要求。
- * 对于Web service提供者来说，部署、升级和维护Web service都非常单纯，不需要考虑客户端兼容问题，而且一次性就能完成。
- * 对于Web service使用者来说，可以轻易实现多种数据、多种服务的聚合（mashup），因此能够做出一些以前根本无法想像的事情。

五、Web service的发展趋势

根据我的观察，目前Web service有这样几种发展趋势。

- * 在使用方式上，RPC和soap的使用在减少，Restful架构占到了主导地位。
- * 在数据格式上，XML格式的使用在减少，json等轻量级格式的使用在增多。
- * 在设计架构上，越来越多的第三方软件让用户在客户端（即浏览器），直接与云端对话，不再使用第三方的服务器进行中转或处理数据。

(完)

关于2的补码

问一个基本的问题。

负数在计算机中如何表示？

举例来说，+8在计算机中表示为二进制的1000，那么-8怎么表示呢？

很容易想到，可以将一个二进制位（bit）专门规定为符号位，它等于0时就表示正数，等于1时就表示负数。比如，在8位机中，规定每个字节的最高位为符号位。那么，+8就是00001000，而-8则是10001000。

但是，随便找一本《计算机原理》，都会告诉你，实际上，计算机内部采用2的补码（Two's Complement）表示负数。

什么是2的补码？

它是一种数值的转换方法，要分二步完成：

第一步，每一个二进制位都取相反值，0变成1，1变成0。比如，00001000的相反值就是11110111。

第二步，将上一步得到的值加1。11110111就变成11111000。

所以，00001000的2的补码就是11111000。也就是说，-8在计算机（8位机）中就是用11111000表示。

不知道你怎么看，反正我觉得很奇怪，为什么要采用这么麻烦的方式表示负数，更直觉的方式难道不好吗？

昨天，我在一本书里又看到了这个问题，然后就花了一点时间到网上找资料，现在总算彻底搞明白了。

2的补码的好处

首先，要明确一点。计算机内部用什么方式表示负数，其实是无所谓的。只要能够保持一一对应的关系，就可以用任意方式表示负数。所以，既然可以任意选择，那么理应选择一种最方便的方式。

2的补码就是最方便的方式。它的便利体现在，所有的加法运算可以使用同一种电路完成。

还是以-8作为例子。

假定有两种表示方法。一种是直觉表示法，即10001000；另一种是2的补码表示法，即11111000。请问哪一种表示法在加法运算中更方便？

随便写一个计算式， $16 + (-8) = ?$

16的二进制表示是00010000，所以用直觉表示法，加法就要写成：

0 0 0 1 0 0 0 0

$$\begin{array}{r}
 +10001000 \\
 \hline
 10011000
 \end{array}$$

可以看到，如果按照正常的加法规则，就会得到10011000的结果，转成十进制就是-24。显然，这是错误的答案。也就是说，在这种情况下，正常的加法规则不适用于正数与负数的加法，因此必须制定两套运算规则，一套用于正数加正数，还有一套用于正数加负数。从电路上说，就是必须为加法运算做两种电路。

现在，再来看2的补码表示法。

$$\begin{array}{r}
 00010000 \\
 +11111000 \\
 \hline
 100001000
 \end{array}$$

可以看到，按照正常的加法规则，得到的结果是100001000。注意，这是一个9位的二进制数。我们已经假定这是一台8位机，因此最高的第9位是一个溢出位，会被自动舍去。所以，结果就变成了00001000，转成十进制正好是8，也就是 $16 + (-8)$ 的正确答案。这说明了，2的补码表示法可以将加法运算规则，扩展到整个整数集，从而用一套电路就可以实现全部整数的加法。

2的补码的本质

在回答2的补码为什么能正确实现加法运算之前，我们先看看它的本质，也就是那两个步骤的转换方法是怎么来的。

要将正数转成对应的负数，其实只要用0减去这个数就可以了。比如，-8其实就是 $0 - 8$ 。

已知8的二进制是00001000，-8就可以用下面的式子求出：

$$\begin{array}{r}
 00000000 \\
 -00001000 \\
 \hline
 \end{array}$$

因为00000000（被减数）小于0000100（减数），所以不够减。请回忆一下小学算术，如果被减数的某一位小于减数，我们怎么办？很简单，向左一位借1就可以了。

所以，0000000也向左一位借了1，也就是说，被减数其实是100000000，算式也就改写成：

$$\begin{array}{r}
 100000000 \\
 -00001000 \\
 \hline
 11111000
 \end{array}$$

进一步观察，可以发现 $100000000 = 11111111 + 1$ ，所以上面的式子可以拆成两个：

$$\begin{array}{r}
 11111111 \\
 -00001000 \\
 \hline
 \end{array}$$

$$\begin{array}{r} \hline & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{array}$$

2的补码的两个转换步骤就是这么来的。

为什么正数加法适用于2的补码？

实际上，我们要证明的是， $X - Y$ 或 $X + (-Y)$ 可以用 X 加上 Y 的2的补码完成。

Y 的2的补码等于 $(11111111 - Y) + 1$ 。所以， X 加上 Y 的2的补码，就等于：

$$X + (11111111 - Y) + 1$$

我们假定这个算式的结果等于 Z ，即 $Z = X + (11111111 - Y) + 1$

接下来，分成两种情况讨论。

第一种情况，如果 X 小于 Y ，那么 Z 是一个负数。这时，我们就对 Z 采用2的补码的逆运算，求出它对应的正数绝对值，再在前面加上负号就行了。所以，

$$Z = -[11111111 - (Z - 1)] = -[11111111 - (X + (11111111 - Y) + 1 - 1)] = X - Y$$

第二种情况，如果 X 大于 Y ，这意味着 Z 肯定大于 11111111 ，但是我们规定了这是8位机，最高的第9位是溢出位，必须被舍去，这相当于减去 100000000 。所以，

$$Z = Z - 100000000 = X + (11111111 - Y) + 1 - 100000000 = X - Y$$

这就证明了，在正常的加法规则下，可以利用2的补码得到正数与负数相加的正确结果。换言之，计算机只要部署加法电路和补码电路，就可以完成所有整数的加法。

(完)

关于Unix哲学

先讲两个很老的小故事。

第一个故事。

有一家日本最大的化妆品公司，收到了用户的投诉。用户抱怨买来的肥皂盒是空的。这家公司为了防止再发生这样的事故，很辛苦地发明了一台X光检查器，能够透视每一个出货的肥皂盒。

同样的事故，发生在一家小公司。他们的解决方法是买一台强力的工业电扇，对着肥皂盒猛吹，被吹走的就是空肥皂盒。

第二个故事。

美国太空总署（NASA）发现在太空失重状态下，航天员无法用墨水笔写字。于是，他们花了大量经费，研发出了一种可以在失重状态下写字的太空笔。猜猜看，俄国人是怎么解决的？（答案在本文结尾处。）

=====

这几天，我在看Unix，发现很多人在谈"Unix哲学"，也就是开发Unix系统的指导思想。

[Wikipedia](#)上列出了好几个版本，不同的人有不同的总结。发明管道命令的Doug McIlroy总结了三条，而Eric S. Raymond则在*The Art of Unix Programming*一书中，一口气总结了17条（[英文版](#)，[中文版](#)）。

但是我发现，所有人都同意，"简单原则"----尽量用简单的方法解决问题----是"Unix哲学"的根本原则。这也就是著名的KISS (keep it simple, stupid) ，意思是"保持简单和笨拙"。



Keep it simple, stupid...

下面就是我对"简单原则"的笔记。如果你想最简单地完成一项编程任务，我认为可以从四个方面入手：

1. 清晰原则。

代码要写得尽量清晰，避免晦涩难懂。清晰的代码不容易崩溃，而且容易理解和维护。重视注释。不为了性能的一丁点提升，而大幅增加技术的复杂性，因为复杂的技术会使得日后的阅读和维护更加艰难。

2. 模块原则。

每个程序只做一件事，不要试图在单个程序中完成多个任务。在程序的内部，面向用户的界面（前端）应该与运算机制（后端）分离，因为前端的变化往往快于后端。

3. 组合原则。

不同的程序之间通过接口相连。接口之间用文本格式进行通信，因为文本格式是最容易处理、最通用的格式。这就意味着尽量不要使用二进制数据进行通信，不要把二进制内容作为输出和输入。

4. 优化原则。

在功能实现之前，不要考虑对它优化。最重要的是让一切先能够运行，其次才是效率。“先求运行，再求正确，最后求快。”（Make it run, then make it right, then make it fast.）90%的功能现在能实现，比100%的功能永远实现不了强。先做出原型，然后找出哪些功能不必实现，那些不用写的代码显然无需优化。目前，最强大的优化工具恐怕是Delete键。

=====

答案是，俄国人用铅笔。

(完)

字符编码笔记：ASCII，Unicode和UTF-8

今天中午，我突然想搞清楚Unicode和UTF-8之间的关系，于是就开始在网上查资料。

结果，这个问题比我想象的复杂，从午饭后一直看到晚上9点，才算初步搞清楚。

下面就是我的笔记，主要用来整理自己的思路。但是，我尽量试图写得通俗易懂，希望能对其他朋友有用。毕竟，字符编码是计算机技术的基石，想要熟练使用计算机，就必须懂得一点字符编码的知识。

1. ASCII码

我们知道，在计算机内部，所有的信息最终都表示为一个二进制的字符串。每一个二进制位（bit）有0和1两种状态，因此八个二进制位就可以组合出256种状态，这被称为一个字节（byte）。也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从0000000到1111111。

上个世纪60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为ASCII码，一直沿用至今。

ASCII码一共规定了128个字符的编码，比如空格"SPACE"是32（二进制00100000），大写的字母A是65（二进制01000001）。这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的1位统一规定为0。

2、非ASCII编码

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。比如，在法语中，字母上方有注音符号，它就无法用ASCII码表示。于是，一些欧洲国家就决定，利用字节中闲置的最高位编入新的符号。比如，法语中的é的编码为130（二进制10000010）。这样一来，这些欧洲国家使用的编码体系，可以表示最多256个符号。

但是，这里又出现了新的问题。不同的国家有不同的字母，因此，哪怕它们都使用256个符号的编码方式，代表的字母却不一样。比如，130在法语编码中代表了é，在希伯来语编码中却代表了字母Gimel (ג)，在俄语编码中又会代表另一个符号。但是不管怎样，所有这些编码方式中，0--127表示的符号是一样的，不一样的只是128--255的这一段。

至于亚洲国家的文字，使用的符号就更多了，汉字就多达10万左右。一个字节只能表示256种符号，肯定是不够的，就必须使用多个字节表达一个符号。比如，简体中文常见的编码方式是GB2312，使用两个字节表示一个汉字，所以理论上最多可以表示 $256 \times 256 = 65536$ 个符号。

中文编码的问题需要专文讨论，这篇笔记不涉及。这里只指出，虽然都是用多个字节表示一个符号，但是GB类的汉字编码与后文的Unicode和UTF-8是毫无关系的。

3. Unicode

正如上一节所说，世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。为

什么电子邮件常常出现乱码？就是因为发信人和收信人使用的编码方式不一样。

可以想象，如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是Unicode，就像它的名字都表示的，这是一种所有符号的编码。

Unicode当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，U+4E25表示汉字“严”。具体的符号对应表，可以查询unicode.org，或者专门的[汉字对应表](#)。

4. Unicode的问题

需要注意的是，Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

比如，汉字“严”的unicode是十六进制数4E25，转换成二进制数足足有15位（100111000100101），也就是说这个符号的表示至少需要2个字节。表示其他更大的符号，可能需要3个字节或者4个字节，甚至更多。

这里就有两个严重的问题，第一个问题是，如何才能区别Unicode和ASCII？计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号呢？第二个问题是，我们已经知道，英文字母只用一个字节表示就够了，如果Unicode统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必然有二到三个字节是0，这对于存储来说是极大的浪费，文本文件的大小会因此大出二三倍，这是无法接受的。

它们造成的结果是：1) 出现了Unicode的多种存储方式，也就是说有许多种不同的二进制格式，可以用来表示Unicode。2) Unicode在很长一段时间内无法推广，直到互联网的出现。

5.UTF-8

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8就是在互联网上使用最广的一种Unicode的实现方式。其他实现方式还包括UTF-16（字符用两个字节或四个字节表示）和UTF-32（字符用四个字节表示），不过在互联网上基本不用。**重复一遍，这里的关系是，UTF-8是Unicode的实现方式之一。**

UTF-8最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

UTF-8的编码规则很简单，只有二条：

- 1) 对于单字节的符号，字节的第一位设为0，后面7位为这个符号的unicode码。因此对于英文字母，UTF-8编码和ASCII码是相同的。
- 2) 对于n字节的符号（n>1），第一个字节的前n位都设为1，第n+1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的unicode码。

下表总结了编码规则，字母x表示可用编码的位。

Unicode符号范围 | UTF-8编码方式
(十六进制) | (二进制)

```
-----+
0000 0000-0000 007F | 0xxxxxxx
0000 0080-0000 07FF | 110xxxxx 10xxxxxx
0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

根据上表，解读UTF-8编码非常简单。如果一个字节的第一位是0，则这个字节单独就是一个字符；如果第一位是1，则连续有多少个1，就表示当前字符占用多少个字节。

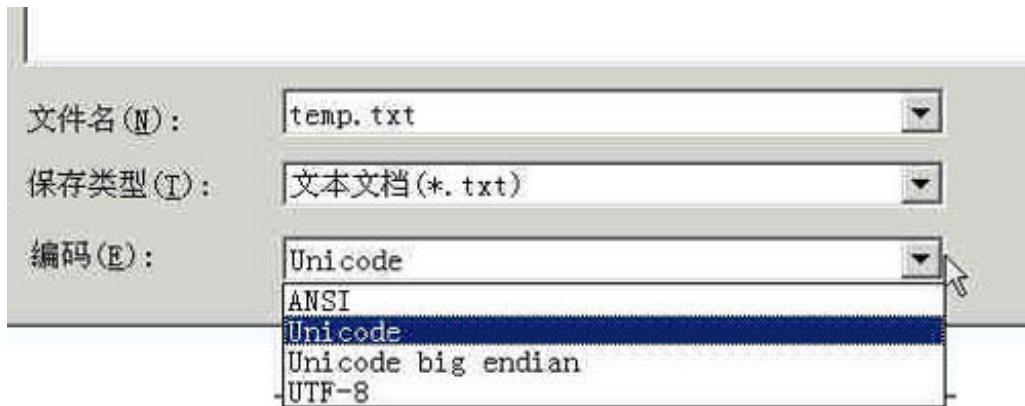
下面，还是以汉字"严"为例，演示如何实现UTF-8编码。

已知"严"的unicode是4E25（100111000100101），根据上表，可以发现4E25处在第三行的范围内（0000 0800-0000 FFFF），因此"严"的UTF-8编码需要三个字节，即格式是"1110xxxx 10xxxxxx 10xxxxxx"。然后，从"严"的最后一个二进制位开始，依次从后向前填入格式中的x，多出的位补0。这样就得到了，"严"的UTF-8编码是"11100100 10111000 10100101"，转换成十六进制就是E4B8A5。

6. Unicode与UTF-8之间的转换

通过上一节的例子，可以看到"严"的Unicode码是4E25，UTF-8编码是E4B8A5，两者是不一样的。它们之间的转换可以通过程序实现。

在Windows平台下，有一个最简单的转化方法，就是使用内置的记事本小程序Notepad.exe。打开文件后，点击"文件"菜单中的"另存为"命令，会跳出一个对话框，在最底部有一个"编码"的下拉条。



里面有四个选项：ANSI，Unicode，Unicode big endian 和 UTF-8。

- 1) ANSI是默认的编码方式。对于英文文件是ASCII编码，对于简体中文文件是GB2312编码（只针对Windows简体中文版，如果是繁体中文版会采用Big5码）。
- 2) Unicode编码指的是UCS-2编码方式，即直接用两个字节存入字符的Unicode码。这个选项用的little endian格式。
- 3) Unicode big endian编码与上一个选项相对应。我在下一节会解释little endian和big endian的涵义。

4) UTF-8编码，也就是上一节谈到的编码方法。

选择完"编码方式"后，点击"保存"按钮，文件的编码方式就立刻转换好了。

7. Little endian和Big endian

上一节已经提到，Unicode码可以采用UCS-2格式直接存储。以汉字"严"为例，Unicode码是4E25，需要用两个字节存储，一个字节是4E，另一个字节是25。存储的时候，4E在前，25在后，就是Big endian方式；25在前，4E在后，就是Little endian方式。

这两个古怪的名称来自英国作家斯威夫特的《格列佛游记》。在该书中，小人国里爆发了内战，战争起因是人们争论，吃鸡蛋时究竟是从大头(Big-Endian)敲开还是从小头(Little-Endian)敲开。为了这件事情，前后爆发了六次战争，一个皇帝送了命，另一个皇帝丢了王位。

因此，第一个字节在前，就是"大头方式"（Big endian），第二个字节在前就是"小头方式"（Little endian）。

那么很自然的，就会出现一个问题：计算机怎么知道某一个文件到底采用哪一种方式编码？

Unicode规范中定义，每一个文件的最前面分别加入一个表示编码顺序的字符，这个字符的名字叫做"零宽度非换行空格"（ZERO WIDTH NO-BREAK SPACE），用FEFF表示。这正好是两个字节，而且FF比FE大1。

如果一个文本文件的头两个字节是FE FF，就表示该文件采用大头方式；如果头两个字节是FF FE，就表示该文件采用小头方式。

8. 实例

下面，举一个实例。

打开"记事本"程序Notepad.exe，新建一个文本文件，内容就是一个"严"字，依次采用ANSI，Unicode，Unicode big endian 和 UTF-8编码方式保存。

然后，用文本编辑软件[UltraEdit](#)中的"十六进制功能"，观察该文件的内部编码方式。

1) ANSI：文件的编码就是两个字节"D1 CF"，这正是"严"的GB2312编码，这也暗示GB2312是采用大头方式存储的。

2) Unicode：编码是四个字节"FF FE 25 4E"，其中"FF FE"表明是小头方式存储，真正的编码是4E25。

3) Unicode big endian：编码是四个字节"FE FF 4E 25"，其中"FE FF"表明是大头方式存储。

4) UTF-8：编码是六个字节"EF BB BF E4 B8 A5"，前三个字节"EF BB BF"表示这是UTF-8编码，后三个"E4B8A5"就是"严"的具体编码，它的存储顺序与编码顺序是一致的。

9. 延伸阅读

* [The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About](#)

[Unicode and Character Sets](#) (关于字符集的最基本知识)

* [谈谈Unicode编码](#)

* [RFC3629: UTF-8, a transformation format of ISO 10646](#) (如果实现UTF-8的规定)

(完)

密码学笔记

1.

加密方法可以分为两大类。一类是单钥加密 (private key cryptography) , 还有一类叫做双钥加密 (public key cryptography) 。前者的加密和解密过程都用同一套密码，后者的加密和解密过程用的是两套密码。

历史上，人类传统的加密方法都是前一种，比如二战期间德军用的Enigma电报密码。莫尔斯电码也可以看作是一种私钥加密方法。

2.

在单钥加密的情况下，密钥只有一把，所以密钥的保存变得很重要。一旦密钥泄漏，密码也就被破解。

3.

在双钥加密的情况下，密钥有两把，一把是公开的公钥，还有一把是不公开的私钥。

双钥加密的原理如下：

- a) 公钥和私钥是一一对应的关系，有一把公钥就必然有一把与之对应的、独一无二的私钥，反之亦成立。
- b) 所有的（公钥, 私钥）对都是不同的。
- c) 用公钥可以解开私钥加密的信息，反之亦成立。
- d) 同时生成公钥和私钥应该相对比较容易，但是从公钥推算出私钥，应该是很困难或者是不可能的。

4.

目前，通用的单钥加密算法为DES (Data Encryption Standard) , 通用的双钥加密算法为RSA (Rivest-Shamir-Adleman) , 都产生于上个世纪70年代。

5.

在双钥体系中，**公钥用来加密信息，私钥用来数字签名。**

6.

因为任何人都可以生成自己的（公钥，私钥）对，所以为了防止有人散布伪造的公钥骗取信任，就需要一个可靠的第三方机构来生成经过认证的（公钥，私钥）对。目前，世界上最主要的数字服务认证商是位于美国加州的[Verisign](#)公司，它的主要业务就是分发RSA数字证书。

(完)

回车和换行

今天，我总算搞清楚"回车"（carriage return）和"换行"（line feed）这两个概念的来历和区别了。

在计算机还没有出现之前，有一种叫做电传打字机（Teletype Model 33）的玩意，每秒钟可以打10个字符。但是它有一个问题，就是打完一行换行的时候，要用去0.2秒，正好可以打两个字符。要是在这0.2秒里面，又有新的字符传过来，那么这个字符将丢失。

于是，研制人员想了个办法解决这个问题，就是在每行后面加两个表示结束的字符。一个叫做"回车"，告诉打字机把打印头定位在左边界；另一个叫做"换行"，告诉打字机把纸向下移一行。

这就是"换行"和"回车"的来历，从它们的英语名字上也可以看出一二。

后来，计算机发明了，这两个概念也就被搬到了计算机上。那时，存储器很贵，一些科学家认为在每行结尾加两个字符太浪费了，加一个就可以。于是，就出现了分歧。

Unix系统里，每行结尾只有"\n"，即"\n"；Windows系统里面，每行结尾是"\r\n"，即"\r\n"；Mac系统里，每行结尾是"\r"。一个直接后果是，Unix/Mac系统下的文件在Windows里打开的话，所有文字会变成一行；而Windows里的文件在Unix/Mac下打开的话，在每行的结尾可能会多出一个^M符号。

（完）