中国科学院大学
University of Chinese Academy of Sciences

CS101

计算机科学导论

初识程序设计

徐志伟
中科院计算所 中国科学院大学
zxu@ict.ac.cn

7. 斐波那契兔子问题（接上周）

- 直观描述和数学公式
  - 有人在2021年1月送你一对刚诞生的兔子；一对兔子出生后，从第三个月开始就每月生一对小兔子。到了第n个月， 你家里有多少对兔子（记为F(n)）？
  - 斐波那契数列的数学公式： F(0)=0， F(1)=1，当n>1时F(n)=F(n-1)+F(n-2)

$$F(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

- 有四种难度
  - 不需要计算思维也能做
    - 这种问题有时被称为玩具问题（toy problem）
    - 习题1（小学）：到了2021年12月，你家里有多少对兔子？ **求F(12)**
  - **需要计算思维**
    - 习题2（中学）：到了从2021年1月开始的第50个月呢？ **求F(50)**
    - 习题3（大学）：到了从2021年1月开始的第10亿个月呢？ **求F(十亿)**
    - 习题4（研究生）：用斐波那契数列求解希尔伯特第十问题 **思维实验**

7.1 斐波那契数列习题1（小学）：求F(12)

- 用算法和程序刻画计算过程
  - 算法，往往用**伪代码**表示
    - 写给人看的

```
Output F(12)                          // 算法，伪代码
where F(n) is defined as
    if (n=0 or n=1) then F(n)=n
    else F(n)=F(n-1)+F(n-2)
```

要实现自动计算，
首先用算法刻画计算过程

这一步依赖人的智慧

小学手算的直觉思维过程表达	数学表达	
	n	**F(n)**
2021年0月：0对。	0	F(0)=0
2021年1月：1对。	1	F(1)=1
2021年2月：还是1对；因为第一对兔子还未成年。	2	F(2)=1
2021年3月：2对；第一对兔子成年了，生了一对小兔子。	3	F(3)=1+1=2
2021年4月：3对；第一对兔子又生了一对小兔子。	4	F(4)=2+1=3
2021年5月：5对；除了4月的3对兔子之外，第一对兔子又生了一对小兔子，第二对兔子生了一对小兔子。	5	F(5)=F(4)+F(3)=5
……	…	……
2021年12月：144对。	12	F(12)=F(11)+F(10)=144

概括总结出斐波那契公式：
每月兔子数是前两个月的之和

$$F(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

人工编程：再从算法导出程序

- 用算法和程序刻画计算过程
  - 算法，往往用**伪代码**表示
    - 写给人看的
  - 程序，计算机语言表达的算法
    - 高级语言程序fib-12.go


```
Output F(12)                    // 算法，伪代码
where F(n) is defined as
    if (n=0 or n=1) then F(n)=n
    else F(n)=F(n-1)+F(n-2)
```

```
package main                    // Go程序
import "fmt“                    // fib-12.go
func main() {
   fmt.Println("F(12)=",fibonacci(12))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 { return n }
   return fibonacci(n-1)+fibonacci(n-2)
}
```

编译：从高级语言程序生成低级语言程序

- 用算法和程序刻画计算过程
  - 算法，往往用伪代码表示
    - 写给人看的
  - 程序，计算机语言表达的算法

例子展示了三种程序

- 高级语言程序fib-12.go
- 低级语言程序
  - 汇编程序fib-12.asm
  - 机器代码fib-12

编译
go build fib-12.go

```
package main                    // Go程序
import "fmt“                    // fib-12.go
func main() {
   fmt.Println("F(12)=",fibonacci(12))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 { return n }
   return fibonacci(n-1)+fibonacci(n-2)
}
```

汇编

```
…
MOVQ   (TLS), CX          // 汇编程序
CMPQ   SP, 16(CX)         // fib-12.asm
JLS    181
SUBQ   $96, SP
MOVQ   BP, 88(SP)
LEAQ   88(SP), BP
…
```

```
01111111 01000101 01001100 01000110
00000010 00000001 00000001 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000010 00000000 00111110 00000000
00000001 00000000 00000000 00000000
```

二进制程序
fib-12

二进制程序在计算机上比特精准地自动执行

- 用算法和程序刻画计算过程
  - 算法，往往用伪代码表示
    - 写给人看的
  - 程序，计算机语言表达的算法

易理解	高级语言程序	
难理解	汇编语言程序	
很难理解		机器语言程序
	不能直接执行	能直接执行

小学手算的直觉思维过程表达	数学表达	
	n	F(n)
2021年0月：0对。	0	F(0)=0
2021年1月：1对。	1	F(1)=1
2021年2月：还是1对；因为第一对兔子还未成年。	2	F(2)=1
2021年3月：2对；第一对兔子成年了，生了一对小兔子。	3	F(3)=1+1=2
2021年4月：3对；第一对兔子又生了一对小兔子。	4	F(4)=2+1=3
2021年5月：5对；除了4月的3对兔子之外，第一对兔子又生了一对小兔子，第二对兔子生了一对小兔子。	5	F(5)=F(4)+F(3)=5
……	…	……
2021年12月：144对。	12	F(12)=F(11)+F(10)=144

100字节

人工编程

391字节

编译

go build fib-12.go

22,653字节

汇编

2,011,793字节

```
Output F(12)                          // 算法，伪代码
where F(n) is defined as
    if (n=0 or n=1) then F(n)=n
    else F(n)=F(n-1)+F(n-2)
```

```
package main                          // Go程序
import "fmt“                          // fib-12.go
func main() {
   fmt.Println("F(12)=",fibonacci(12))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 { return n }
   return fibonacci(n-1)+fibonacci(n-2)
}
```

比特精准

```
…
MOVQ   (TLS), CX                      // 汇编程序
CMPQ   SP, 16(CX)                     // fib-12.asm
JLS    181
SUBQ   $96, SP
MOVQ   BP, 88(SP)
LEAQ   88(SP), BP
…
```

```
01111111 01000101 01001100 01000110
00000010 00000001 00000001 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000010 00000000 00111110 00000000
00000001 00000000 00000000 00000000
```

机器语言程序

fib-12

7.2 斐波那契数列习题2（中学）：求F(50)

- 编译运行fib-50.go（**演示**）
  - 计算速度太慢了
- 重新建模，利用比内公式，开发出fib.binet.go程序

$$F(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}},$$

其中 $\varphi = \frac{1+\sqrt{5}}{2} = 1.618\ldots$

- 编译运行fib.binet.go**演示**
  - 速度很快，但只得到近似结果　F(50) = 1.2586269024999998e+10
    = 12586269024.999998
    0.000002 误差
    正确结果是 F(50) = 12586269025
  - 出现**截断误差**（roundoff error），10^{-6}量级

- 可容忍的截断误差由各个应用问题决定
  - 很多日常问题使用牛顿力学公式且假设10^{-6}米精度就足够了
  - 引力波探测则要求严格得多的精度，达到5×10^{-23}米

7.3 巧妙构造 Acu-Exams

- 假设我们要求精准结果F(50)=12586269025
  - 不准有截断误差，且速度要快
- 采用一种聪明的动态规划算法开发出新程序fib.dp.go
- 编译运行fib.dp.go（**演示**），得到精准结果F(50)=12586269025
  - 速度很快
  - 时间复杂度降低
    - fib.go的时间复杂度：$O(2^n)$
    - fib.dp.go的时间复杂度：$O(n)$

```
package main            // fib-50.go
import "fmt“
func main() {
   fmt.Println("F(50)=",fibonacci(50))
}
func fibonacci(n int) int {
   if n == 0 || n == 1 { return n }
   return fibonacci(n-1)+fibonacci(n-2)
}
```

```
package main            // fib.dp-50.go
import "fmt"
const N = 50
var mem [N + 1]int
func main() {
   for i := 0; i <= N; i++ { mem[i] = -1 }
   fmt.Printf("F(%d)=%d\n", N, fibonacci(N))
}
func fibonacci(n int) int {
   if mem[n] != -1 { return mem[n] }
   if n == 0 || n == 1 {
      mem[n] = n
      return mem[n]
   }
   mem[n] = fibonacci(n-1) + fibonacci(n-2)
   return mem[n]
}
```

7.4 斐波那契数列习题3（大学）：求F(1,000,000,000)

- 程序fib.dp.go能够求解F(1,000,000,000)吗？
- 编译运行程序fib.dp-93.go（**演示**），求F(93)
  - 出错：F(93)= -6246583658587674878
  - 正确值F(93)= 12200160415121876738
  - 原因：一个64位整数装不下，**溢出**
  - 64比特整数能够表示的最大值是
    $2^{63}-1$ = 9223372036854775807
  - F(93) = 12200160415121876738
    太大了
- 需要系统支持**任意字长的整数**

```
package main              // fib.dp-93.go
import "fmt"
const N = 93
var mem [N + 1]int
func main() {
   for i := 0; i <= N; i++ { mem[i] = -1 }
   fmt.Printf("F(%d)=%d\n", N, fibonacci(N))
}
func fibonacci(n int) int {
   if mem[n] != -1 { return mem[n] }
   if n == 0 || n == 1 {
      mem[n] = n
      return mem[n]
   }
   mem[n] = fibonacci(n-1) + fibonacci(n-2)
   return mem[n]
}
```

正确、巧妙、实用的计算过程 Acu-Exams

- 斐波那契数列习题3（大学）：求F(1,000,000,000)
- 重新建模得到程序fib.matrix.go，利用了斐波那契数列的矩阵性质
  - $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^n = \begin{bmatrix}F(n+1) & F(n)\\ F(n) & F(n-1)\end{bmatrix}$ $A^{16} = (((A^2)^2)^2)^2$
  - 并通过矩阵平方求矩阵n次方，将计算复杂度从$O(n)$降为~$O(\log n)$
- 利用系统提供的**任意长整数big.Int**抽象，避免了溢出
  - **计算F(5,000,000)仅需要4秒；计算F(1,000,000,000)需要十多万秒**
    - F(1,000,000,000)是一个很大的正整数，有2000多万个十进制数字
    - 进阶实验：十小时内计算出F(1,000,000,000)

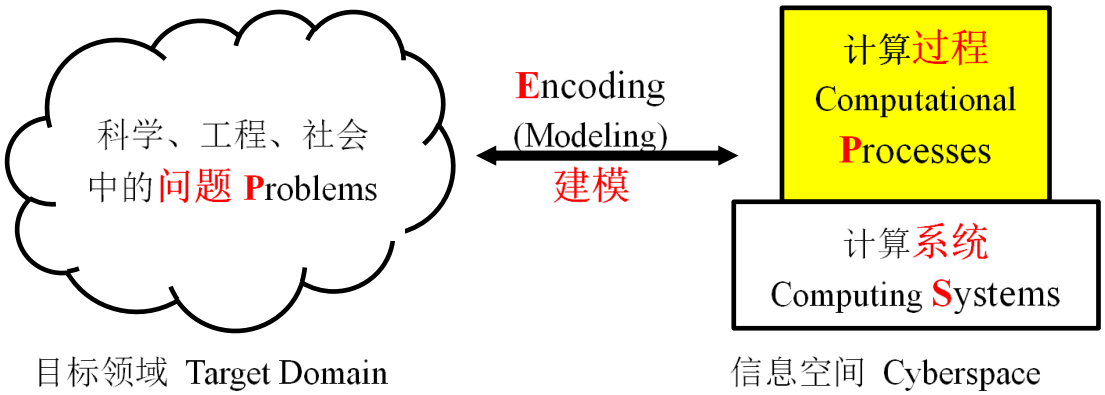
求解F(n)的执行时间（秒）

n	fib.go $O(2^n)$	fib.dp.go $O(n)$	fib.matrix.go ~$O(\log n)$
50	725	0.059	0.000012
500	出错、极慢	出错	0.000022
5,000,000	出错、极慢	出错	4.13
1,000,000,000	出错、极慢	出错、很慢	187,160

提纲

- 目的和目标
  - 上周概述了计算思维及其影响
    - 计算过程，4个渗透命题，计算就是力量，敏锐审视活力法，冯诺依曼模型

      Acu-Exams PEPS
  - 本周初识**程序设计**
    - 也称为programming，即**编程**
    - 如何精确地表达“操纵数字符号变换信息”的自动逐步计算过程
  - 回忆一下：什么是程序？计算机语言表达的算法



场景思维

- 场景目标（果壳同学完成任务的能力）
  - 一周后，能够理解10行级别的Go程序
  - 二周后，能够读写20行级别的Go程序
  - 三周后，能够读写30行级别的Go程序
- 具体任务
  - 正确理解（读、写、执行、改错）10~30行级别的Go程序
  - 开发姓名编码程序
    - 将“王果壳”的拼音**字符串**“Wang Guo Ke”变换成**整数**936，并逐**字符**（9、3、6）打印出来
  - 开发简版快速排序程序
    - 将 [8 3 6 7 2 1 4 5] 从小到大排序成 [1 2 3 4 5 6 7 8]
- 知识理解
  - 进制转换：十、二、十六进制；整数与英文字符（ASCII）的表示
  - 姓名编码程序：程序、语句、字符、整数、数组与循环
  - 简版快速排序程序：切片与递归

体认思维：符号是文明的载体

- 永乐大典
  - “兴”的数千年历史演变
  - 篆书
  - 隶书
  - 真书（楷书）
  - 草书
  - 章草
    - 藏于国家图书馆

- **数字符号是当代文明的载体**
  - **“兴”的Unicode编码：U+5174**

计算过程是操作数字符号的步骤序列

- 通过**数据类型**表示数字符号
- 通过**程序语句**实现步骤序列

1. 数字符号表示

计算机使用数据类型来表示数字符号
计算机硬件只能理解二进制符号

1.1 二进制-十进制-十六进制表示与转换

- 关键是正确利用二进制-十进制的对应关系
- 下面两个例子展示了一种转换方法

- **例1：将二进制数转换为十进制数**
- $(110.101)_2$ = (?)

$$= 1\times2^2+1\times2^1+1\times2^{-1}+1\times2^{-3}$$

$$= \quad 4 \quad + \quad 2 \quad + 0.5 \quad + 0.125 = 6.625_{10}$$

二进制	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
	10000	1000	100	10	1	0.1	0.01	0.001	0.0001	0.00001
十进制	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

二进制数位的十进制数值对应表

例2：将十进制数转换为二进制数

- $6.625 = (110.101)_2$
- 分别算出整数部分$6. = (110.)_2$和分数部分$.625 = (.101)_2$

- 第1步：6-4=2；
  够减，记录**1**(2)。

6.625

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
8	4	2	1	0.5	0.25	0.125	0.0625	0.03125
	1 (2)							

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
10000.	1000.	100.	10.	1.	0.1	0.01	0.001	0.0001	0.00001
16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

$6.625 = (110.101)_2$

6.625

- 第1步：6-4=2；
  够减，记录**1**(2)。

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)							

- 第2步：2-2=0；
  够减，记录**1**(0)。

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	**1** (0)						

- 第3步：余数是0；终止。
  整数部分是110。

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	**1** (0)	**0**					

- 第4步：0.625-0.5=0.125；
  够减，记录**1**(.125)。

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	**1** (0)	**0**	**1** (.125)				

- 第5步： 0.125-0.25；
  不够减，记录**0**(.125)。

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	**1** (0)	**0**	**1** (.125)	**0** (.125)			

- 第6步：0.125-0.125=0；
  够减，记录**1**(0)。

2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (2)	**1** (0)	**0**	**1** (.125)	**0** (.125)	**1** (0)		

- 第7步：余数是0；终止。
  分数部分是.101。

- 结果是110.101。

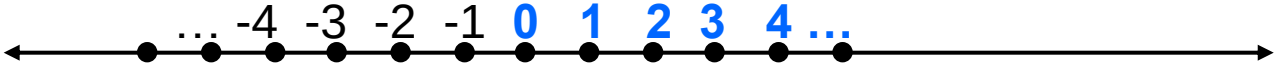
2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
10000.	1000.	100.	10.	1.	0.1	0.01	0.001	0.0001	0.00001
16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

二进制-十进制-十六进制表示与转换

- 十六进制只需四分之一的数位
  - 常常用在程序中
    - 63_{10} = 111111 = 00111111
      = 0011 1111
      = $3F_{16}$
- 二进制转换到十六进制
  - 用A、B、C、D、E、F指代十进制的10、11、12、13、14、15
    - 避免$63_{10}=3F_{16}$被误写为 $63_{10} = 3(15)_{16} = 315_{16}$
  - 将二进制值从右到左以4比特为单位分组，再将每个4比特组对应到相应的十六进制值
- 程序中记为
  0x3f，0X3F，或 0x3F

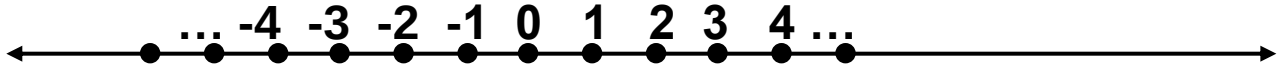
二进制 Binary	十进制 Decimal	十六进制 Hexadecimal
$2^3 2^2 2^1 2^0$	$10^1 10^0$	16^0
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

1.2 表示整数



- 数据类型（type）规定数据表示和允许的操作
- 使用无符号整数（unsigned integers）表示自然数
  - 采用n比特可表示闭区间[0, 2^n -1]内的任意自然数
    - 数据类型uint8可表示[0, 2^8 -1]（即[0, 255]）内的2^8=256个自然数
    - 数据类型uint64可表示[0, 2^{64} -1] 内的2^{64} 个自然数
  - 超出范围称为**溢出**（overflow）；uint8的溢出条件是: **<0** 或 **>255**
    - 这是**有限抽象**特征带来的第二种差错，第一种是斐波那契例子中的截断误差
- 允许的操作：+, -, *, /, %; ++, --; >>, <<; &, |, ^
  - 数据类型uint8（byte）例子
  - 字面量（literal）：在程序中直接出现值，如自然数35
  - 与小学数学相同的正常操作（加、减、乘）
    - 63 + 64 = 127；64 -63 = 1；63 * 2 = 126
  - 与小学数学不同的操作（除、求余、溢出overflow ）
    - 63 / 2 = 31（不是31.5）； 63 % 2 = 1（63除2的余数为1）
    - 63 - 64 = -1（<0）， 64 * 64 = 4096（>255）；两者皆产生溢出（overflow）

表示整数



- 假如需要负数怎么办？使用带符号整数（signed integers）
  - 最左位（最高位）表示符号：0 (+)，1(-)

- 简单带符号整数 (原码)：符号位 + 7位绝对值
  会出错误结果

63 = 00111111， 64 = 01000000

(-63) = 10111111， (-64) = 11000000

63 + 64 = 00111111 + 01000000 = 01111111 = 127 √

(-63) + (-64) = 10111111 + 11000000 = 11111111 = (-127) √

63 + (-63) = 00111111 + 10111111 = 11111110 = (-126) X

二进制补码表示带符号整数，8比特例子

- 简单带符号整数（原码表示），可能出错
  - 63　　= 00111111，　64　　= 01000000
  - (-63)　= 10111111，　(-64)　= 11000000
  - 63 + 64 = 00111111 + 01000000　　= 01111111 = 127　　√
  - (-63) + (-64) = 10111111 + 11000000　= 11111111 = (-127)　√
  - 63 + (-63) = 00111111 + 10111111　　= 11111110 = (-126)　X
- 补码表示（Two's complement representation）
  - 正整数（如63）：保持原码表示，63 = 00111111
  - 负整数（如-63）：绝对值按位求非，然后加1（视为无符号整数）
    - Bitwise negate absolute(N), and add 00000001
    - (-63) = (00111111)按位求非+ 00000001
      = 11000000　　+ 00000001 = 11000001
  - 补码表示的整数加法：采用无符号整数加法并忽略溢出
  - 63 + (-63) = 00111111 + 11000001 = 00000000 = 0　　√

8比特补码加法的详细步骤

- 63+63 = 00111111+00111111

  00111111

  01111110 = 126

- 63+(-63) = 00111111+11000001

  11000001

  100000000 = 00000000 = 0 (ignore overflowing **1**)

- (-63)+(-63)= 11000001+11000001

  11000001

  110000010 = 10000010 = -126 (ignore overflowing **1**)

- 思考题：如何做减法?
  - 63 – 63 = 0 63 – (-63) = 126
  - (-63) – 63 = -126 (-63) – (-63) = 0

1.3 表示英文字符：ASCII字符集

ASCII encodings for "Alan Turing" = [65, 108, 97, 110, 32, 84, 117, 114, 105, 110, 103]

使用8位无符号整数（uint8）表示英文字符

可以表示256个字符，本课程只用到128个

类型uint8 = 类型byte

空格符在哪里？

$D_3D_2D_1D_0$ \ $D_6D_5D_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Control Characters 控制字符

Printable Characters 可打印字符

$D_3D_2D_1D_0$ \ $D_6D_5D_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

ASCII control characters (code 0-31, 127)

Decimal	Hexadecimal	Symbol	Description	Chinese
0	0x00	NUL	Null Character	空字符
1	0x01	SOH	Start of Heading	标题开始
2	0x02	STX	Start of Text	正文开始
3	0x03	ETX	End of Text	正文结束
4	0x04	EOT	End of Transmission	传输结束
5	0x05	ENQ	Enquiry	请求
6	0x06	ACK	Acknowledgment	收到通知
7	0x07	BEL	Bell	响铃
8	0x08	BS	Back Space	退格
9	0x09	HT	Horizontal Tab	水平制表符
10	0x0A	LF, NL	Line Feed, New Line	换行键
11	0x0B	VT	Vertical Tab	垂直制表符
12	0x0C	FF, NP	Form Feed, New Page	换页键
13	0x0D	CR	Carriage Return	回车键
14	0x0E	SO	Shift Out	不用切换
15	0x0F	SI	Shift In	启用切换
16	0x10	DLE	Data Line Escape	数据链路转义
17	0x11	DC1	Device Control 1	设备控制1
18	0x12	DC2	Device Control 2	设备控制2
19	0x13	DC3	Device Control 3	设备控制3
20	0x14	DC4	Device Control 4	设备控制4
21	0x15	NAK	Negative Acknowledgement	拒绝接收
22	0x16	SYN	Synchronous Idle	同步空闲
23	0x17	ETB	End of Transmit Block	结束传输块
24	0x18	CAN	Cancel	取消
25	0x19	EM	End of Medium	媒介结束
26	0x1A	SUB	Substitute	代替
27	0x1B	ESC	Escape	换码(溢出)
28	0x1C	FS	File Separator	文件分隔符
29	0x1D	GS	Group Separator	分组符
30	0x1E	RS	Record Separator	记录分隔符
31	0x1F	US	Unit Separator	单元分隔符
127	0x7F	DEL	Delete	删除

ASCII printable characters (code 32-126)

Decimal	Hexadecimal	Symbol	Description	Chinese
32	0x20	SP	Space	空格
33	0x21	!	Exclamation mark	叹号
34	0x22	"	Double quotes	双引号
35	0x23	#	Number, sharp	井号
36	0x24	$	Dollar sign	美元符
37	0x25	%	Percent sign	百分号
38	0x26	&	Ampersand	和号
39	0x27	'	Single quote	闭单引号
40	0x28	(	Open parenthesis (or open bracket)	开括号
41	0x29	)	Close parenthesis (or close bracket)	闭括号
42	0x2A	*	Asterisk, multiply	星号
43	0x2B	+	Plus	加号
44	0x2C	,	Comma	逗号
45	0x2D	-	Hyphen	减号/破折号
46	0x2E	.	Period, dot	句号
47	0x2F	/	Slash, divide	斜杠
48	0x30	0	Zero	字符0
49	0x31	1	One	字符1
57	0x39	9	Nine	字符9
58	0x3A	:	Colon	冒号
59	0x3B	;	Semicolon	分号
60	0x3C	<	Less than (or open angled bracket)	小于
61	0x3D	=	Equals	等号
62	0x3E	>	Greater than (or close angled bracket)	大于
63	0x3F	?	Question mark	问号
64	0x40	@	At symbol	电子邮件符号

Decimal	Hexadecimal	Symbol	Description	Chinese
65	0x41	A	Uppercase A	大写字母A
66	0x42	B	Uppercase B	大写字母B
90	0x5A	Z	Uppercase Z	大写字母Z
91	0x5B	[	Opening bracket	开方括号
92	0x5C	\	Backslash	反斜杠
93	0x5D	]	Closing bracket	闭方括号
94	0x5E	^	Caret	脱字符
95	0x5F	_	Underscore	下划线
96	0x60	`	Grave accent	开单引号
97	0x61	a	Lowercase a	小写字母a
98	0x62	b	Lowercase b	小写字母b
122	0x7A	z	Lowercase z	小写字母z
123	0x7B	{	Opening brace	开花括号
124	0x7C	|	Vertical bar	垂线
125	0x7D	}	Closing brace	闭花括号
126	0x7E	~	Tilde	波浪号

ASCII编码不是采用一个字节（8比特）吗？

为什么只有7比特，没有D_7？

ASCII encoding =
$D_7D_6D_5D_4D_3D_2D_1D_0$
= 0$D_6D_5D_4D_3D_2D_1D_0$
D_7 is 0

空格字符例子

SP = 00100000_2
=32_{10}

SP（空格）是ASCII字符

00100000
or 32
是它的
ASCII编码
也称为
ASCII值
或
ASCII码

$D_3D_2D_1D_0$ \ $D_6D_5D_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

表示ASCII字符例子：加号

在程序中使用单引号括上该字符‘+’

$D_3D_2D_1D_0$ \ $D_6D_5D_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

SP= 00100000_2 $=32_{10}$

SP is the ASCII character

00100000 or 32 Is its ASCII encoding or ASCII value

D_7 is 0

What is the ASCII value of +

‘+’ 00101011_2 43_{10}

容易搞混三个“空”相关字符

NUL（空字符），SP（空格），0（零，数字0）

SP = 00100000_2 = 32_{10}

SP is the ASCII character

00100000 or 32 Is its ASCII encoding or ASCII value

$D_3D_2D_1D_0$ \ $D_6D_5D_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

NUL 00000000_2 0_{10}

‘0’ 00110000_2 48_{10}

课堂小测验

- 姓名：　　学号：

- 请指出下述哪一个断言是错误的（）
  A. 计算机硬件只能直接执行二进制程序
  B. 计算机的冯诺依曼模型的要点之一是“指令驱动”，因此，当没有指令执行时，计算机不做任何操作，好像冻结了一样
  C. 十六进制数0x1F对应十进制数31
  D. 英文字符‘B’的ASCII码是4

2. 使用Go语言编程

编辑-编译-执行-调试的基本流程

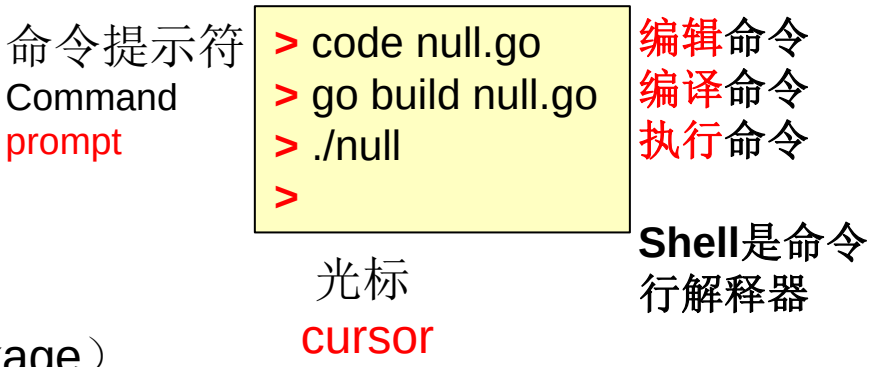
Go程序的基本结构和语句

Go语言的3种基本数据类型和两种复合类型

2.1 执行几个简单程序

演示：云计算环境

```
package main   // The main package
func main() {  // a main function that does nothing
}              // 它的主函数体是空的
```



- null.go
  - 程序有两个语句（statement）
  - 同学们写的程序属于主包（main package）
  - 每个主包有一个主函数（main function）

- 使用命令行界面工具（shell）的三种动作
  - shell解释命令
  - 通过解释go build null.go命令，计算机系统编译高级语言程序null.go，产生可执行代码null
  - 通过解释./null命令，计算机系统执行可执行代码null（或./null）

2.1 执行几个简单程序

演示：云计算环境

```
package main   // The main package
func main() {  // a main function that does nothing
}              // 它的主函数体是空的
```

命令提示符
Command
prompt

```
> code null.go
> go build null.go
> ./null
>
```

编辑命令
编译命令
执行命令

光标
cursor

Shell是命令行解释器

- null.go
  - 程序有两个语句（statement）
  - 同学们写的程序属于主包（main package）
  - 每个主包有一个主函数（main function）

- 常见错误：使用了中文字符

应为　　func main() {
写成了　　func main（）{

应为　　import "fmt"
写成了　　import “fmt”

应为　　'A'
写成了　　‘A’

执行几个简单程序

演示：云计算环境

```
package main   // The main package
func main() {  // a main function that does nothing
}              // 它的主函数体是空的
```

命令提示符
Command
prompt

```
> code null.go
> go build null.go
> ./null
>
```

编辑命令
编译命令
执行命令

Shell是命令行解释器

光标
cursor

- null.go
  - 程序有两条语句（statement）
  - 同学们写的程序属于主包（main package）
  - 每个主包有一个主函数（main function）
- hello.go
  - fmt是Go语言自带的程序包
    称为库（library）
    - 其他人开发了fmt供用户重用，
      fmt 包含函数fmt.Println
      - 采用点号标记法（dot notation）调用
  - 程序中的函数很像数学函数
    - 输入数据→输出数据

  但可包含副作用（side effect）
    - 此例hello.go程序的主函数
      只有副作用，即打印hello!

```
package main   // hello.go程序的主包
import "fmt"       // 该程序导入一个库包fmt
func main() {      // 该程序声明一个主函数
   fmt.Println("hello!")    // 输出语句打印出hello!
}
```

```
> go build hello.go
> ./hello
hello!
>
```

Compile and execute in two commands

```
> go run hello.go
hello!
>
```

Compile and execute in one command
两个命令可以合并成一个命令

2.2 Go程序的基本结构

主包声明语句	package main	// 定义主包
导入语句	import …	// 导入其他人写的包，如不调用则不出现
常量声明语句	const …	// 声明全局常量，可不出现
变量声明语句	var …	// 声明全局变量，可不出现
函数声明语句	func X(…) …{…}	// 声明程序员自定义的函数，可不出现
主函数声明语句	func main() {	// 声明主包中必须有的主函数
	…	// 主函数的函数体
	}	
	main()	**// 缺省的主函数调用，不出现**

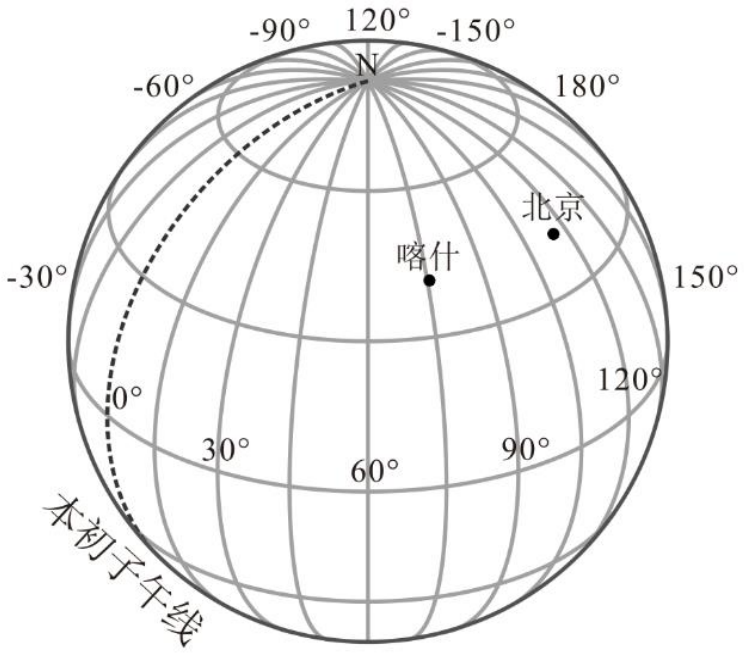
```
func X(…) …{                    // 如函数体等括起来的{…}称为一个代码块（code block）
   常量声明语句                  // 局部常量，作用域是本函数
   变量声明语句                  // 局部变量，作用域是本函数
   赋值语句
   循环语句
   条件判断语句
   函数调用语句，如打印语句
}
```

已知北京日出时间，求喀什日出时间

- 2022年12月5日，北京日出时间=07:20
  - **7点20分 = 7*3600 + 20*60 = 26400 秒**
- 问题：预测喀什日出时间。
- 逐步计算过程
  1. **求自转速度S。北纬40度线上任意一点的自转速度是S = 40000*cos(40) / (24*3600) = 355 米/秒。**
  2. **求经度之差L。北京与喀什的经度之差是L = 116-76 = 40度，即360经度的1/9。**
  3. **求两点的距离D。北京与喀什的距离是40000*cos(40)/9=3405千米。**
  4. **求两点的时差T。北京与喀什的时差是3405/355=9592秒，即2小时39分52秒。**
  5. **喀什的日出时间是26400+9592=35992秒，或早上7点20分+ 2小时39分52秒，即早上9点59分52秒。**
- 检验：
  - 中央气象台：2022年12月5日，喀什的日出时间是早上9点59分。

已知事实（近似数值）

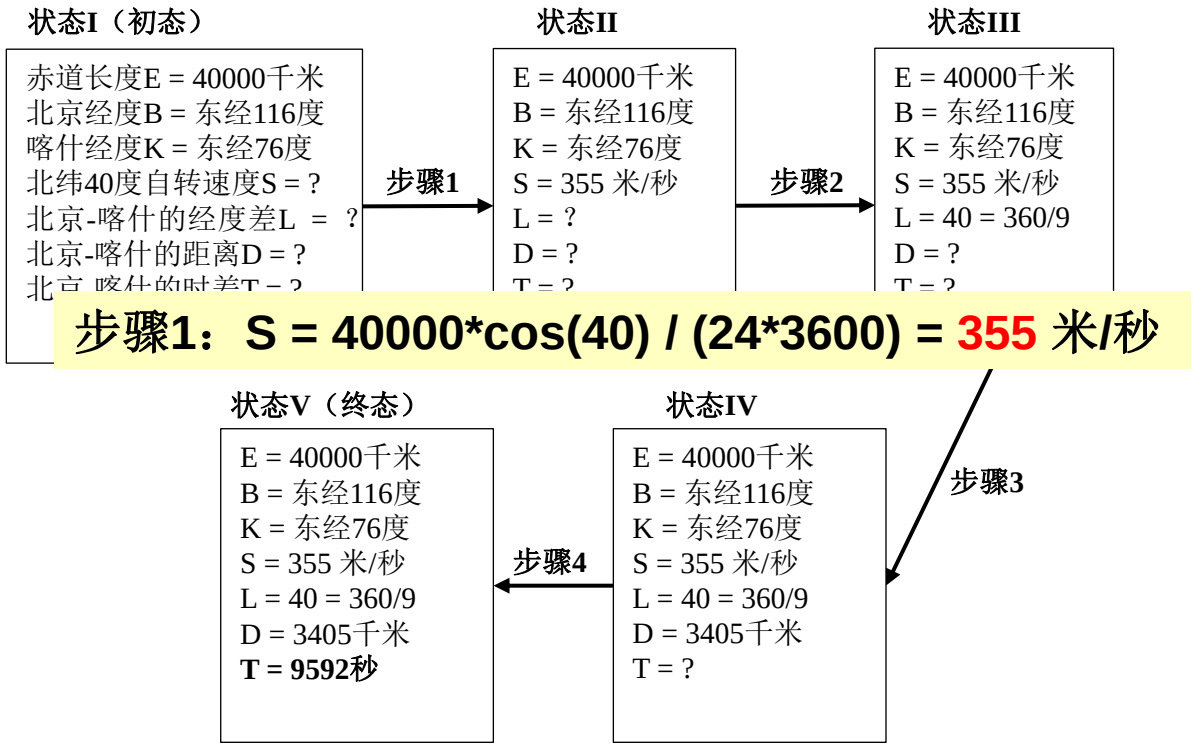
- 地球是一个球体
  - 赤道长度=子午线长度=40000千米
- 地球在24小时完成一圈自转。
- 北京的经纬度是东经116度北纬40度，喀什的经纬度是东经76度北纬40度。

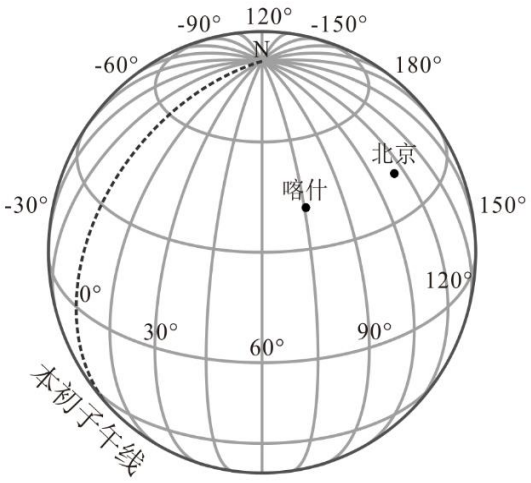


本质：用自动机刻画计算过程

通过一个变量赋值演变序列来表示（称为系统、自动机）

- 状态（state）：任何时刻，系统的变量赋值称为自动机的状态（state）
- 步骤（step）：一次状态转移（state transition）
  - 即自动机从当前状态转移到下一状态；如状态I→状态II，变量S的赋值从?变为355
- 过程（process）：自动机的一系列状态转移，从初始状态变换到最终状态
  - **状态I（初态）→ II → III → IV →状态V（终态）**





从手工数学计算过程到程序

已知北京日出时间为

7点20分 = 7*3600+20*60 = 26400 秒

手工计算过程

1. 北纬40度线上任意一点的自转速度是
   S = 40000*cos(40) / (24*3600)
   = 355 米/秒。
2. 北京与喀什的经度之差是
   L = 116-76 = 40度，
   即360经度的1/9。
3. 北京与喀什的距离是
   D = 40000*cos(40) / 9
   = 3405千米。
4. 北京与喀什的时差是
   T = 3405/355 = 9592秒，
   即2小时39分52秒。
5. 喀什的日出时间是
   26400+9592=35992秒，
   或早上7点20分+ 2小时39分52秒，
   即早上9点59分52秒。

顺便简化重复计算

length = 40000*cos(40)

Go语言程序

```
package main     // 简单但有错的代码
func main() {
    length := 40000*cos(40)
    S := length / (24*3600)
    D := length * (116-76) / 360
    T := D / S
    print(7*3600 + 20*60 + T)
}
```

与手工数学计算过程相比，程序有何异同

已知北京日出时间为

7点20分 = 7*3600+20*60 = 26400 秒

手工计算过程

1. 北纬40度线上任意一点的自转速度是
   S = 40000*cos(40) / (24*3600)
   = 355 米/秒。
2. 北京与喀什的经度之差是
   L = 116-76 = 40度，
   即360经度的1/9。
3. 北京与喀什的距离是
   D = 40000*cos(40) / 9
   = 3405千米。
4. 北京与喀什的时差是
   T = 3405/355 = 9592秒，
   即2小时39分52秒。
5. 喀什的日出时间是
   26400+9592=35992秒，
   或早上7点20分+ 2小时39分52秒，
   即早上9点59分52秒。

Go语言程序

```
package main            // 计算出正确值
import "fmt"
import "math"
func main() {
    length := 40000*math.Cos(40*3.14159/180)
    S := length / (24*3600)
    fmt.Println("自转速度S =",S)
    D := length * (116-76) / 360
    fmt.Println("距离D =",D)
    T := D / S
    fmt.Println("时差T =",T)
    fmt.Println("喀什日出时间 =", 7*3600 + 20*60 + T)
}
```

```
> go run SunRise.go
自转速度S = 0.3546502086915901
距离D = 3404.6420034392645
时差T = 9600
喀什日出时间 = 36000
```

2.3 基本数据类型的表示

- 在计算机中的表示
  - 任何数都直接表示为二进制数
    - 自然数63、英文字符‘?‘，在计算机中都表示为8比特数00111111

- 打印格式
  - 打印出来时，可看到不同语义，如自然数63，或字符'?'
  - 掌握占位符
    - 二进制（%b）、十进制（%d）、十六进制（%x）、字符（%c）
  - 掌握转义码
    - 反斜杠（\\）、制表符（\t）、换行符（\n）、双引号（\"）

打印格式——占位符（格式动词）

- 运行symbols.go
- 在fmt.Printf语句中使用5种占位符
- 要打印出由字符'6' 和'3'形成的字符串"63"，粗体格式是错误的，下面三种格式都是正确的

  fmt.Printf("%s\n", "63")

  fmt.Printf("%c%c\n", '6', '3')

  fmt.Printf("%c%c\n",6+'0',3+'0')

Formatting Verb 占位符	含义	例子
%b	Binary 二进制	fmt.Printf("%b",63) 打印出111111 fmt.Printf("%08b",63) 打印出00111111
%c	Character 字符	fmt.Printf("%c",63) outputs ?
%d	Decimal 十进制	fmt.Printf("%d",63) outputs 63
%s	String 字符串	fmt.Printf("%s",string(63)) outputs ?
%x or %X	Hexadecimal 十六进制	fmt.Printf("%X",63) outputs 3F

```
package main   // symbols.go
import "fmt"
func main() {
    fmt.Printf("Decimal: %d\n",63)
    fmt.Printf("Hex: %X\n",63)
    fmt.Printf("Binary: %b\n",63)
    fmt.Printf("Character: %c\n",63)
    fmt.Printf("String: %c%c\n",63)
    fmt.Printf("String: %c%c\n",6,3)
    fmt.Printf("String: %c%c\n",6+'0',3+'0')
}
```

```
Decimal: 63              ; decimal representation of 63
Hex: 3F                  ; representation of 63
Binary: 111111           ; binary representation of 63
Character: ?             ; 63 is the ASCII code of '?'
String: ?%!c(MISSING)    ; explicit Error
String: =╚               ; implicit Error
String: 63               ; Output '6', '3'
```

转义码，如何打印出特殊符号
ASCII控制字符、已被Golang占用的符号

- 四个特殊符号

转移码值	含义	
\\	Backslash	反斜杠
\t	Tab	制表符
\n	Newline	换行符
\"	Double quote	双引号

- 下列语句有四处使用了转义码

```
fmt.Printf("\t Use \\\" to output %c\n",34)
```

34是双引号的ASCII码

屏幕输出

```
    Use \" to output "

>
```

三种缺省的标准输入输出设备（standard I/O）

- 打印（Printf、Println）并不一定是在打印机上打印
  - 往往是使用**标准输出设备**
- 标准输入Standard Input
  - Keyboard 键盘和鼠标
  - StdIn, stdin
- 标准输出Standard Output
  - Display screen 屏幕
  - StdOut, stdout
- 标准错误输出 Standard Error Output
  - Display screen 屏幕
  - StdErr, stderr

```
package main   // smbols.go
import "fmt"
func main() {
    fmt.Printf("Decimal: %d\n",63)
    fmt.Printf("Hex: %X\n",63)
    fmt.Printf("Binary: %b\n",63)
    fmt.Printf("Character: %c\n",63)
    fmt.Printf("String: %c%c\n",63)
    fmt.Printf("String: %c%c\n",6,3)
    fmt.Printf("String: %c%c\n",6+'0',3+'0')
}
```

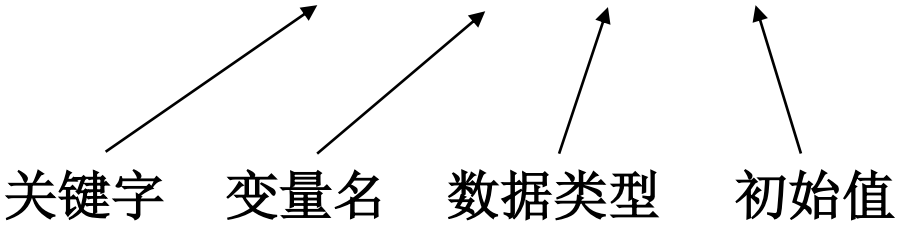
正常输出和错误输出都意味着在屏幕上显示出来

```
Decimal: 63                ; binary representation of 63
Hex: 3F                    ; representation of 63
Binary: 111111             ; binary representation of 63
Character: ?               ; 63 is the ASCII code of 63
String: ?%!c(MISSING)      ; explicit Error
String: ═╚                 ; implicit Error
String: 63                 ; Output '6', '3'
```

2.4 Go语言的基本数据类型

- 变量声明语句

```
var sum int = 1    // 常常可写为 sum := 1
```



  - 变量有三个属性：名字（sum），类型（int），值（初始值为1）

Go语言的基本数据类型

- 变量声明语句 var sum int = 1　// 常常可写为 sum := 1
  - 变量有三个属性：名字（sum），类型（int），值（初始值为1），
  - 类型包括允许的操作（如下表的规定）以及表示
  - 表示： sum的值是1，在内存中的格式 0x0000000000000001
    - 0000000000000000000000000000000000000000000000000000000000000001

Logic →
Character →
Integer →
Real number →

数据类型 Type	大小 Size	字面量 Literals	值 Values	操作 Operations
布尔类型 bool	1 bit	true, false	true, false	&&, ||, !
字节类型 byte, uint8	1 byte 8比特	63, '?'	[0, 255]	+, -, *, /, %; ++, --语句; >>, <<; &, |, ^
带符号整数 int	8 bytes 64比特	-12345, 69	$[-2^{63}, 2^{63}-1]$	
无符号整数 uint64	8 bytes	51234	$[0, 2^{64}-1]$	
浮点数 float64	8 bytes	3.14159	IEEE 754	+, -, *, /

操作和溢出条件（标红）

- 以字节类型（byte, uint8）为例
- 加减乘除求余操作
  - 63-64 = -1（<0）； 64*64 =4096（>255）；两者皆产生溢出（overflow）
  - 63 / 2 = 31（不是31.5）； 63 % 2 = 1（63除2的余数为1）

	数据类型 Type	大小 Size	字面量 Literals	值 Values	操作 Operations
Logic →	布尔类型 bool	1 bit	true, false	true, false	&&, ||, !
Character →	字节类型 byte, uint8	1 byte 8比特	63, '?'	[0, 255] **< 0; >255=2^8-1**	+, -, *, /, %;
Integer →	带符号整数 int	8 bytes 64比特	-12345, 69	[-2^{63}, 2^{63}-1] **< -2^{63}; > 2^{63}-1**	
Integer →	无符号整数 uint64	8 bytes	51234	[0, 2^{64}-1] **< 0; > 2^{64}-1**	
Real number →	浮点数 float64	8 bytes	3.14159	IEEE 754	+, -, *, /

3. 姓名编码实例：从字符串到数的变换

- 两个程序例子："Alan Turing"变换到1045
  - 使用%d的简版程序　　name_to_number-0.go
  - 使用%c的程序　　name_to_number.go
  - 哈希进阶实验，实时查找10亿人（如微信登录）
    - O(N) → O(logN) → O(1)　10亿步 → 30步 → 1步

ASCII encodings for "Alan Turing"

= [65, 108, 97, 110, 32, 84, 117, 114, 105, 110, 103]

$D_3D_2D_1D_0$ \ $D_6D_5D_4$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

3.1 name_to_number-0.go

- 出现新的变量类型与语句
- 变量类型
  - 字符串：string
- 声明变量的两种方式

```
var name string                声明语句
name = “Alan Turing”           赋值语句
```

等价于

```
var name string = “Alan Turing”        赋初值的声明语句
```

基本等价于

```
name := “Alan Turing“                  自动推断出类型，并赋初值
```

变量name的作用域是本代码块

变量i的作用域是本函数

```
package main     //name_to_number-0.go
import "fmt"
func main() {
        var name string = "Alan Turing"
        sum := 0
        for i := 0; i < len(name); i++ {
                sum = sum + int(name[i])
        }
        fmt.Printf("%d\n", sum)
}
```

name_to_number-0.go

- 字符串string是
  不可修改的字符数组
  - 数组是一组同类元素
    - 有长度len(name)
    - 用索引i访问元素name[i]

- 除了package、import、
  主函数声明语句之外，
  还出现了新的
  - 声明语句
  - 赋值语句
    - :=只能赋初值一次
    - 不能只用 :=
  - 循环语句（for loop）
    - 循环体中，变量sum
      被多次赋值

```
package main    // name_to_number-0.go
import "fmt"
func main() {
        var name string = "Alan Turing"
        sum := 0
        for i := 0; i < len(name); i++ {
                sum = sum + int(name[i])
        }
        fmt.Printf("%d\n", sum)
}
```

注意：name[4] 是空格' ' =32

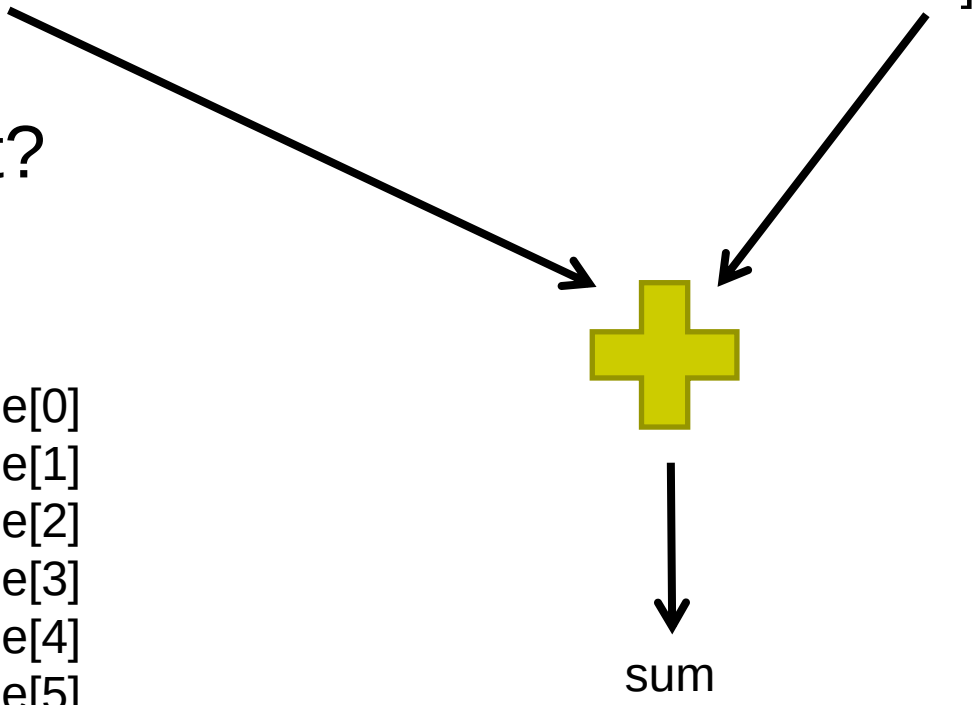
A	l	a	n		T	u	r	i	n	g
name = [65	108	97	110	32	84	117	114	105	110	103]
index →	0	1	2	3	4	5	6	7	8	9	10

len(name) is 11

name[0]='A'=65, name[1]='l'=108, name[2]='a'=97,
name[3]='n'=110, name[4]=' '=32, name[5]='T'=84,
name[6]='u'=117, name[7]='r'=114, name[8]='i'=105,
name[9]='n'=110, name[10]='g'=103.

如何求数组name的11个元素之和？

- Problem: add up the 11 elements of name[i]
  - name = [65 108 97 110 32 84 117 114 105 110 103]



- How to do it?
  - Solution 1

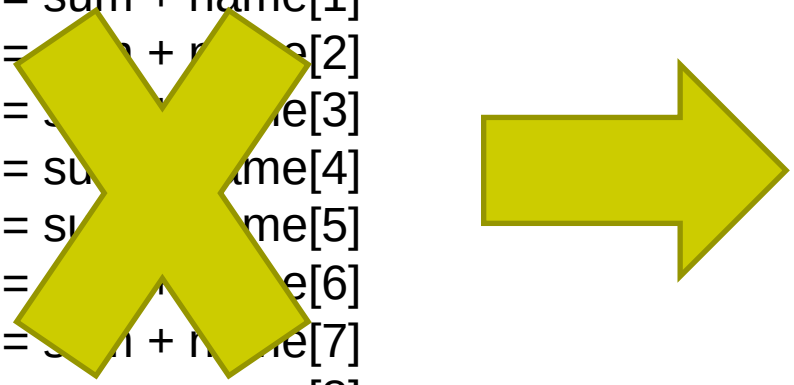
```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
sum = sum + name[2]
sum = sum + name[3]
sum = sum + name[4]
sum = sum + name[5]
sum = sum + name[6]
sum = sum + name[7]
sum = sum + name[8]
sum = sum + name[9]
sum = sum + name[10]
```

How to add up elements of an array?

- Problem: add up the 11 elements of name[i]
  - name = [65 108 97 110 32 84 117 114 105 110 103]

- How to do it?
  - Solution 1

```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
sum = sum + name[2]
sum = sum + name[3]
sum = sum + name[4]
sum = sum + name[5]
sum = sum + name[6]
sum = sum + name[7]
sum = sum + name[8]
sum = sum + name[9]
sum = sum + name[10]
```



Solution 2 Why?

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
sum = sum + int(name[2])
sum = sum + int(name[3])
sum = sum + int(name[4])
sum = sum + int(name[5])
sum = sum + int(name[6])
sum = sum + int(name[7])
sum = sum + int(name[8])
sum = sum + int(name[9])
sum = sum + int(name[10])
```

Add up [65 108 97 110 32 84 117 114 105 110 103]

name[0] = 65 = 01000001

Value Name	Binary representation
sum (right-side)	0000000000000000000000000000000000000000000000000000000000000000
name[0]	01000001
int(name[0])	0000000000000000000000000000000000000000000000000000000001000001
sum (left-side)	0000000000000000000000000000000000000000000000000000000001000001

var name string = "Alan Turing"
var sum int = 0

Type of name: an array of byte, i.e., uint8
Type of sum: 64-bit integer

```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
…
sum = sum + name[10]
```

Solution 1

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
…
sum = sum + int(name[10])
```

Solution 2

Add up [65 108 97 110 32 84 117 114 105 110 103]

name[0] = 65 = 01000001

Value Name	Binary representation
sum (right-side)	00000000000000000000000000000000000000000000000000000000
name[0]	01000001
int(name[0])	00000000000000000000000000000000000000000000000001000001
sum (left-side)	00000000000000000000000000000000000000000000000001000001

类型转换操作
Type casting operation int(name[0])
converts byte type to int type
By padding 56 0's

```
sum := 0
sum = sum + name[0]
sum = sum + name[1]
…
sum = sum + name[10]
```

Solution 1

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
…
sum = sum + int(name[10])
```

Solution 2

Add up [65 108 97 110 32 84 117 114 105 110 103]

name[1] = 108 = 01101100

Value Name	Binary representation
sum (right-side)	00000000000000000000000000000000000000000000000000000001000001
name[1]	01101100
int(name[1])	00000000000000000000000000000000000000000000000000000001101100
sum (left-side)	00000000000000000000000000000000000000000000000000000010101101

sum := 0
sum = sum + int(name[0])
= 0 + 65 = 65
= 00000000000000000000000000000000000000000000000000000001000001
sum = sum + int(name[1])
= 65 + 108 = 173
= 00000000000000000000000000000000000000000000000000000010101101

Solution 2 still has problems

1. Tied to particular students 绑定了学生姓名
   - Does not work for students with len(name) != 11
2. Tedious, repetitive code 重复代码
3. The length of code is proportional to the problem size! 直线程序
   - **A sign of possible bad design**
     - What if len(name) == 1 million?
   - 违反了程序有限性原则

```
sum := 0
sum = sum + int(name[0])
sum = sum + int(name[1])
sum = sum + int(name[2])
sum = sum + int(name[3])
sum = sum + int(name[4])
sum = sum + int(name[5])
sum = sum + int(name[6])
sum = sum + int(name[7])
sum = sum + int(name[8])
sum = sum + int(name[9])
sum = sum + int(name[10])
```

幸好Go语言提供了for loop循环抽象

- 初始化: 数组索引从零开始 i = 0
- 重复执行循环体，每次迭代索引加1，直到i≥11

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   sum = sum + int(name[0])
   sum = sum + int(name[1])
   sum = sum + int(name[2])
   sum = sum + int(name[3])
   sum = sum + int(name[4])
   sum = sum + int(name[5])
   sum = sum + int(name[6])
   sum = sum + int(name[7])
   sum = sum + int(name[8])
   sum = sum + int(name[9])
   sum = sum + int(name[10])
   fmt.Printf("%d\n", sum)
}
```

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   fmt.Printf("%d\n", sum)
}
```

循环抽象的诀窍：综合变与不变
不变：for循环结构不变
变：索引值变化，累加值sum变化

3.2 name_to_number.go

采用占位符%c实现
占位符%d

```
package main //name_to_number-0.go
import "fmt"
func main() {
        var name string = "Alan Turing"
        sum := 0
        for i := 0; i < 11; i++ {
                sum = sum + int(name[i])
        }
        fmt.Printf("%d\n", sum)
}
```

```
> ./name_to_number-0
1045
>
```

```
package main  // name_to_number.go
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   var sum_bytes [4]byte
   var j int
   for j = 3; sum != 0; j-- {
      sum_bytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   fmt.Printf("%c", sum_bytes[0])
   fmt.Printf("%c", sum_bytes[1])
   fmt.Printf("%c", sum_bytes[2])
   fmt.Printf("%c", sum_bytes[3])
   fmt.Printf("\n")
}
```

```
> ./name_to_number
1045
>
```

如何用%c实现%d?

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
   var sum_bytes [4]byte
   var j int
   for j = 3; sum!= 0; j-- {
      sum_bytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   fmt.Printf("%c", sum_bytes[0])
   fmt.Printf("%c", sum_bytes[1])
   fmt.Printf("%c", sum_bytes[2])
   fmt.Printf("%c", sum_bytes[3])
   fmt.Printf("\n")
}
```

使用整数除法 / 和求余 % 操作，
从数值1045依次摘取出数字5, 4, 0, 1

```
sum % 10
sum = sum / 10
```

sum_bytes[3]	1045 % 10 = 5
sum	1045 / 10 = 104
sum_bytes[2]	104 % 10 = 4
sum	104 / 10 = 10
sum_bytes[1]	10 % 10 = 0
sum	10 / 10 = 1
sum_byte[0]	1 % 10 = 1
sum	1 / 10 = 0

sum_bytes = [‘1’, ‘0’, ‘4’, ‘5’]
数组索引 0 1 2 3

用字符占位符%c实现十进制占位符 %d
fmt.Printf("%d\n", sum)

四个打印语句fmt.Printf(“%c”, …)
依次打印出1, 0, 4, 5四个字符

为什么要做类型转换

sum_bytes[j] = byte(sum%10) + ‘0’
而不是sum_bytes[j] = sum % 10

```
package main
import "fmt"
func main() {
   var name string = "Alan Turing"
   sum := 0
   for i := 0; i < 11; i++ {
      sum = sum + int(name[i])
   }
                  // 此时，sum == 1045
   var sum_bytes [4]byte
   var j int
   for j = 3; sum != 0; j-- {
      sum_bytes[j] = byte(sum%10) + '0'
      sum = sum / 10
   }
   fmt.Printf("%c", sum_bytes[0])
   fmt.Printf("%c", sum_bytes[1])
   fmt.Printf("%c", sum_bytes[2])
   fmt.Printf("%c", sum_bytes[3])
   fmt.Printf("\n")
}
```

sum_bytes is a uint8 array
sum_byte[j] has type byte
However, sum is type int

Suppose j=3 (initially)
sum is 1045, and
sum%10 is
1045%10 = 5, a 64-bit int value
00000000……00000101

byte(sum%10), i.e., byte(5)
converts this 64-bit value
to a number of type uint8 and value
00000101

byte(5)+ ‘0’ evaluates to
= 5 + 48
= 00000101+00110000
= 00110101 = 53 = ‘5’

Thus, sum_bytes[3] holds ‘5’

谢谢 Thank You

Q&A

zxu@ict.ac.cn

中科院计算所
INSTITUTE OF COMPUTING TECHNOLOGY

十进制到二进制转换有可能产生无穷数列

- $(11.3)_{10} = (?)_2$
  $= 1011.010011001\ldots$

11.3

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
8	4	2	1	0.5	0.25	0.125	0.0625	0.03125
1 (3)	0 (3)	1 (1)	1 (0)	0 (.3)	1(.05)	0 (.05)	0 (.05)	1 (.01875)

2^{-6}	2^{-7}	2^{-8}	2^{-9}
0.015625	0.0078125	0.00390625	0.001953125
1 (.003125)	0 (.003125)	0 (.003125)	1 (.001171875)

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
10000.	1000.	100.	10.	1.	0.1	0.01	0.001	0.0001	0.00001
16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

fmt软件包中的三个函数

```
package fmt                  // The fmt package，它不是主程序包，而是库包
……
func Println(…) …{           // It contains a function Println which other
      …                      // programs can call by using fmt.Println
}                                          点号标记法（dot notation）
……
func Printf(…) …{            // It contains a function Printf which other
  …                          // programs can call by using fmt.Printf
}
……
func Scanf(…) …{             // It contains a function Scanf which other
     …                       // programs can call by using fmt.Scanf
}
```

两个输出语句 The following two statements do the same thing.

```
fmt.Printf("hello! %d\n",63)
fmt.Println("hello!",63)
```

输出结果

```
> hello! 63
```

Three functions in fmt

```
package fmt                  // It belongs to the fmt package
……
func Println(…) …{           // It contains a function Println which other
      …                      // programs can call by using fmt.Println
}
……
func Printf(…) …{            // It contains a function Printf which other
  …                          // programs can call by using fmt.Printf
}
……
func Scanf(…) …{             // It contains a function Scanf which other
     …                       // programs can call by using fmt.Scanf
}
```

一条输入语句

The following code receives a user-entered integer in variable A.

```
var A int
fmt.Scanf("%d",&A)          // &A indicates the address of A
```