

# 重点内容

对于《深入理解计算机系统》（CSAPP）这本书，它被广泛认为是计算机科学领域的经典之作。因此，**如果时间和精力允许，通读全书无疑是最佳选择**。这本书的每一章都为你理解计算机系统搭建了坚实的基础，并且章节之间层层递进，相互关联。

然而，如果你希望在有限的时间内抓住核心，或者想在通读的过程中有所侧重，那么以下是一些通常被认为是**必读或非常重要的**内容。我会尽量精确到三级标题，但请注意，不同版本和翻译的CSAPP在标题细节上可能略有差异。这里的推荐是基于CSAPP第三版（英文版）的常见结构，中文版通常会做对应翻译。

## 推荐的学习模式：

建议你**先快速通读一遍**，对全书内容有一个整体的印象。然后，**针对以下列出的重点章节和主题进行精读和深入理解**，包括完成相关的练习题（Labs）。其他章节可以作为辅助阅读或在需要时查阅。这种模式既能保证你掌握核心知识，又能对整个计算机系统有全面的认识。

## 必读/重点内容（精确到三级标题）：

以下内容是CSAPP中的核心，对于理解计算机系统的工作原理至关重要：

### 第一部分：程序结构和执行 (Program Structure and Execution)

#### • 第1章：计算机系统漫游 (A Tour of Computer Systems)

- 1.1 信息就是位 + 上下文 (Information is Bits + Context)
- 1.2 程序被其他程序翻译成不同格式 (Programs Are Translated by Other Programs into Different Forms)
- 1.3 了解编译系统如何工作是大有益处的 (It Pays to Understand How Compilation Systems Work)
- 1.4 处理器读并解释存储在存储器中的指令 (Processors Read and Interpret Instructions Stored in Memory)
  - 1.4.1 系统的硬件组成 (Hardware Organization of a System)
  - 1.4.2 运行 Hello 程序 (Running the hello Program)
- 1.5 高速缓存至关重要 (Caches Matter)
- 1.6 存储设备形成层次结构 (Storage Devices Form a Hierarchy)
- 1.7 操作系统管理硬件 (The Operating System Manages the Hardware)

- 1.7.1 进程 (Processes)
  - 1.7.2 线程 (Threads)
  - 1.7.3 虚拟内存 (Virtual Memory)
  - 1.7.4 文件 (Files)
  - 1.9 并发和并行 (Concurrency and Parallelism)
    - 1.9.1 线程级并发 (Thread-Level Concurrency)
    - 1.9.2 指令级并行 (Instruction-Level Parallelism)
    - 1.9.3 单指令、多数据并行 (Single-Instruction, Multiple-Data (SIMD) Parallelism)
  - 1.10 计算机系统中抽象的重要性 (The Importance of Abstractions in Computer Systems)
- **第2章：信息的表示和处理 (Representing and Manipulating Information)**
    - 2.1 信息存储 (Information Storage)
      - 2.1.1 十六进制表示法 (Hexadecimal Notation)
      - 2.1.2 字 (Words)
      - 2.1.3 数据大小 (Data Sizes)
      - 2.1.4 寻址和字节顺序 (Addressing and Byte Ordering) (重点理解大端和小端)
      - 2.1.7 C语言中的位级运算 (Bit-Level Operations in C)
      - 2.1.8 C语言中的逻辑运算 (Logical Operations in C)
      - 2.1.9 C语言中的移位运算 (Shift Operations in C)
    - 2.2 整数表示 (Integer Representations)
      - 2.2.1 整型数据类型 (Integral Data Types)
      - 2.2.2 无符号编码 (Unsigned Encodings)
      - 2.2.3 补码编码 (Two's-Complement Encodings)
      - 2.2.4 有符号数和无符号数之间的转换 (Conversions Between Signed and Unsigned)
      - 2.2.5 C语言中的有符号数与无符号数 (Signed vs. Unsigned in C)
      - 2.2.6 扩展一个数字的位表示 (Expanding the Bit Representation of a Number)
      - 2.2.7 截断数字 (Truncating Numbers)
      - 2.2.8 关于有符号数和无符号数的建议 (Advice on Signed vs. Unsigned)
    - 2.3 整数运算 (Integer Arithmetic)
      - 2.3.1 无符号加法 (Unsigned Addition)

- 2.3.2 补码加法 (Two's-Complement Addition)
  - 2.3.3 补码的非 (Two's-Complement Negation)
  - 2.3.4 无符号乘法 (Unsigned Multiplication)
  - 2.3.5 补码乘法 (Two's-Complement Multiplication)
  - 2.3.6 乘以常数 (Multiplying by Constants)
  - 2.3.7 除以2的幂 (Dividing by Powers of 2)
  - 2.3.8 关于整数运算的最后思考 (Final Thoughts on Integer Arithmetic)
- 2.4 浮点数 (Floating Point)
    - 2.4.1 二进制小数 (Fractional Binary Numbers)
    - 2.4.2 IEEE浮点表示 (IEEE Floating-Point Representation)
    - 2.4.3 数字示例 (Example Numbers)
    - 2.4.4 舍入 (Rounding)
    - 2.4.5 浮点运算 (Floating-Point Operations)
    - 2.4.6 C语言中的浮点数 (Floating Point in C)

- **第3章：程序的机器级表示 (Machine-Level Representation of Programs)** (这一章极为重要)

- 3.1 历史观点 (A Historical Perspective)
- 3.2 程序编码 (Program Encodings)
  - 3.2.1 机器级代码 (Machine-Level Code)
  - 3.2.2 代码示例 (Code Examples)
  - 3.2.3 关于格式的注解 (Notes on Formatting)
- 3.3 数据格式 (Data Formats)
- 3.4 访问信息 (Accessing Information)
  - 3.4.1 操作数指示符 (Operand Specifiers)
  - 3.4.2 数据传送指令 (Data Movement Instructions) (MOV类指令)
  - 3.4.3 数据传送示例 (Data Movement Examples)
  - 3.4.4 压入和弹出栈数据 (Pushing and Popping Stack Data)
- 3.5 算术和逻辑操作 (Arithmetic and Logical Operations)
  - 3.5.1 加载有效地址 (Load Effective Address) (leaq指令)

- 3.5.2 一元和二元操作 (Unary and Binary Operations)
  - 3.5.3 移位操作 (Shift Operations)
  - 3.5.4 特殊的算术操作 (Special Arithmetic Operations)
- 3.6 控制 (Control) (理解汇编如何实现if/else, switch, loop)
    - 3.6.1 条件码 (Condition Codes)
    - 3.6.2 访问条件码 (Accessing the Condition Codes)
    - 3.6.3 跳转指令及其编码 (Jump Instructions and their Encodings)
    - 3.6.4 跳转指令的实现 (Implementing Conditional Branches with Jumps)
    - 3.6.5 循环 (Loops) (do-while, while, for)
    - 3.6.6 条件传送指令 (Conditional Move Instructions)
    - 3.6.7 Switch语句 (Switch Statements)
- 3.7 过程 (Procedures) (函数调用栈帧结构是核心)
    - 3.7.1 栈帧结构 (Stack Frame Structure)
    - 3.7.2 转移控制 (Transferring Control) (call, ret)
    - 3.7.3 寄存器使用惯例 (Register Usage Conventions)
    - 3.7.4 过程示例 (Procedure Example)
    - 3.7.5 递归过程 (Recursive Procedures)
- 3.8 数组分配和访问 (Array Allocation and Access)
    - 3.8.1 基本原则 (Basic Principles)
    - 3.8.2 指针运算 (Pointer Arithmetic)
    - 3.8.3 嵌套的数组 (Nested Arrays)
    - 3.8.4 定长数组 (Fixed-Size Arrays)
    - 3.8.5 变长数组 (Variable-Size Arrays)
- 3.9 异构数据结构 (Heterogeneous Data Structures) (结构体和联合体)
    - 3.9.1 结构 (Structures)
    - 3.9.2 联合 (Unions)
    - 3.9.3 数据对齐 (Data Alignment)
- 3.10 将控制流与数据流结合起来以避免过程调用开销 (Combining Control with Data Flow to Avoid Procedure Call Overhead) (如循环展开)

- 3.11 浮点代码 (Floating-Point Code) (了解即可, 优先级稍低)
- 3.12 x86-64中的IA32兼容性 (IA32 Compatibility in x86-64) (了解即可)

## 第二部分：在系统中运行代码 (Running Code on a System)

### • 第5章：优化程序性能 (Optimizing Program Performance)

- 5.1 优化编译器的能力和局限性 (Capabilities and Limitations of Optimizing Compilers)
- 5.2 程序性能的表示 (Expressing Program Performance)
- 5.3 程序示例 (Program Example)
- 5.4 消除循环的低效率 (Eliminating Loop Inefficiencies) (代码移动)
- 5.5 减少过程调用 (Reducing Procedure Calls)
- 5.6 消除不必要的存储器引用 (Eliminating Unneeded Memory References) (使用临时变量)
- 5.7 理解现代处理器 (Understanding Modern Processors) (重点理解流水线和超标量)
  - 5.7.1 整体操作 (Overall Operation)
  - 5.7.2 功能单元的性能 (Functional Unit Performance)
  - 5.7.3 处理器操作的抽象模型 (An Abstract Model of Processor Operation)
- 5.8 循环展开 (Loop Unrolling)
- 5.9 增强并行性 (Enhancing Parallelism)
  - 5.9.1 多重累积 (Multiple Accumulators)
  - 5.9.2 重新结合变换 (Reassociation Transformation)
- 5.11 理解存储器性能 (Understanding Memory Performance)
  - 5.11.1 加载的延迟 (Load Latency)
  - 5.11.2 存储操作的延迟 (Store Latency)
- 5.13 应用：性能改进技术 (Applying: Performance Improvement Techniques)
- 5.14 确认和消除性能瓶颈 (Identifying and Eliminating Performance Bottlenecks) (Amdahl定律)
  - 5.14.1 程序剖析 (Program Profiling)
  - 5.14.2 使用剖析程序来指导优化 (Using a Profiler to Guide Optimization)
  - 5.14.3 Amdahl定律 (Amdahl's Law)

### • 第6章：存储器层次结构 (The Memory Hierarchy) (这一章极为重要)

- 6.1 存储技术 (Storage Technologies)
  - 6.1.1 随机访问存储器 (Random-Access Memory) (SRAM, DRAM)
  - 6.1.2 磁盘存储 (Disk Storage)
  - 6.1.3 固态硬盘 (Solid-State Disks)
  - 6.1.4 存储技术趋势 (Storage Technology Trends)
- 6.2 局部性 (Locality) (时间局部性和空间局部性是核心概念)
  - 6.2.1 对程序数据引用的局部性 (Locality of References to Program Data)
  - 6.2.2 取指令的局部性 (Locality of Instruction Fetches)
  - 6.2.3 局部性小结 (Summary of Locality)
- 6.3 存储器层次结构 (The Memory Hierarchy)
  - 6.3.1 存储器层次结构的中心思想 (Central Idea of a Memory Hierarchy)
  - 6.3.2 存储器层次结构概念小结 (Summary of Memory Hierarchy Concepts)
- 6.4 高速缓存存储器 (Cache Memories) (核心中的核心)
  - 6.4.1 通用高速缓存组织结构 (Generic Cache Memory Organization)
  - 6.4.2 直接映射高速缓存 (Direct-Mapped Caches)
  - 6.4.3 组相联高速缓存 (Set Associative Caches)
  - 6.4.4 全相联高速缓存 (Fully Associative Caches)
  - 6.4.5 有关写的问题 (Issues with Writes) (写直通, 写回)
  - 6.4.6 真实的高速缓存层次结构剖析 (Anatomy of a Real Cache Hierarchy)
  - 6.4.7 高速缓存参数的性能影响 (Performance Impact of Cache Parameters)
- 6.5 编写高速缓存友好的代码 (Writing Cache-friendly Code)
- 6.6 综合：高速缓存对程序性能的影响 (Putting It Together: The Impact of Caches on Program Performance)
  - 6.6.1 存储器山 (The Memory Mountain)
  - 6.6.2 重新排列循环以提高空间局部性 (Rearranging Loops to Increase Spatial Locality)
  - 6.6.3 在程序中利用局部性 (Exploiting Locality in Your Programs)

### 第三部分：程序间的交互和通信 (Interaction and Communication Between Programs)

- 第7章：链接 (Linking) (理解链接过程对理解大型项目和解决链接错误至关重要)

- 7.1 编译器驱动程序 (Compiler Drivers)
  - 7.2 静态链接 (Static Linking)
  - 7.3 目标文件 (Object Files)
  - 7.4 可重定位目标文件 (Relocatable Object Files)
  - 7.5 符号和符号表 (Symbols and Symbol Tables)
  - 7.6 符号解析 (Symbol Resolution) (理解符号如何被解析是重点)
    - 7.6.1 链接器如何解析多重定义的全局符号 (How Linkers Resolve Multiply Defined Global Symbols)
    - 7.6.2 与静态库链接 (Linking with Static Libraries)
    - 7.6.3 链接器如何使用静态库来解析引用 (How Linkers Use Static Libraries to Resolve References)
  - 7.7 重定位 (Relocation)
    - 7.7.1 重定位条目 (Relocation Entries)
    - 7.7.2 重定位符号引用 (Relocating Symbol References)
  - 7.8 可执行目标文件 (Executable Object Files)
  - 7.9 加载可执行目标文件 (Loading Executable Object Files)
  - 7.10 动态链接共享库 (Dynamic Linking with Shared Libraries)
  - 7.11 从应用程序中加载和链接共享库 (Loading and Linking Shared Libraries from Applications)
  - 7.12 位置无关代码 (Position-Independent Code (PIC))
  - 7.13 处理目标文件的工具 (Tools for Manipulating Object Files)
- **第8章：异常控制流 (Exceptional Control Flow)** (理解ECF是理解操作系统和高级程序行为的基础)
    - 8.1 异常 (Exceptions)
      - 8.1.1 异常处理 (Exception Handling)
      - 8.1.2 异常的类别 (Classes of Exceptions) (中断、陷阱、故障、终止)
      - 8.1.3 Linux/x86-64系统中的异常 (Exceptions in Linux/x86-64 Systems)
    - 8.2 进程 (Processes)
      - 8.2.1 逻辑控制流 (Logical Control Flow)
      - 8.2.2 并发流 (Concurrent Flows)

- 8.2.3 用户模式和内核模式 (User Mode and Kernel Mode)
  - 8.2.4 上下文切换 (Context Switches)
  - 8.3 系统调用错误处理 (System Call Error Handling)
  - 8.4 进程控制 (Process Control)
    - 8.4.1 获取进程ID (Obtaining Process IDs)
    - 8.4.2 创建和终止进程 (Creating and Terminating Processes) (fork, exit, wait, waitpid)
    - 8.4.3 回收子进程 (Reaping Child Processes)
    - 8.4.4 让进程休眠 (Putting Processes to Sleep) (sleep, pause)
    - 8.4.5 加载并运行程序 (Loading and Running Programs) (execve)
    - 8.4.6 利用fork和execve运行程序 (Using fork and execve to Run Programs)
  - 8.5 信号 (Signals)
    - 8.5.1 信号术语 (Signal Terminology)
    - 8.5.2 发送信号 (Sending Signals) (kill, alarm)
    - 8.5.3 接收信号 (Receiving Signals) (signal函数)
    - 8.5.4 信号处理问题 (Signal Handling Issues) (避免竞争条件)
    - 8.5.5 可移植的信号处理 (Portable Signal Handling)
  - 8.6 非本地跳转 (Nonlocal Jumps) (setjmp, longjmp, 了解即可)
  - 8.7 操作进程的工具 (Tools for Manipulating Processes)
- **第9章：虚拟内存 (Virtual Memory)** (这一章极为重要)
- 9.1 物理和虚拟寻址 (Physical and Virtual Addressing)
  - 9.2 地址空间 (Address Spaces)
  - 9.3 虚拟内存作为缓存的工具 (VM as a Tool for Caching) (理解VM如何使用DRAM作为SRAM的缓存)
    - 9.3.1 为什么使用虚拟内存? (Why Virtual Memory?)
    - 9.3.2 DRAM缓存的组织结构 (Organization of a DRAM Cache)
    - 9.3.3 页表 (Page Tables)
    - 9.3.4 页命中 (Page Hits)
    - 9.3.5 缺页 (Page Faults)
    - 9.3.6 分配页面 (Allocating Pages)

- 9.3.7 又是局部性 (Locality to the Rescue Again)
- 9.4 虚拟内存作为内存管理的工具 (VM as a Tool for Memory Management) (每个进程独立的地址空间)
  - 9.4.1 简化链接 (Simplifying Linking)
  - 9.4.2 简化共享 (Simplifying Sharing)
  - 9.4.3 简化内存分配 (Simplifying Memory Allocation)
  - 9.4.4 简化加载 (Simplifying Loading)
- 9.5 虚拟内存作为内存保护的工具 (VM as a Tool for Memory Protection)
- 9.6 地址翻译 (Address Translation) (核心机制, 理解MMU和页表如何工作)
  - 9.6.1 地址翻译概览 (Overview of Address Translation)
  - 9.6.2 集成高速缓存和虚拟内存 (Integrating Caches and VM)
  - 9.6.3 利用TLB加速地址翻译 (Speeding up Address Translation with a TLB)
  - 9.6.4 多级页表 (Multi-Level Page Tables)
  - 9.6.5 端到端的地址翻译: 一个案例研究 (End-to-end Address Translation: A Case Study)
- 9.7 案例研究: Intel Core i7内存系统 (Case Study: The Intel Core i7 Memory System)
  - 9.7.1 Core i7地址翻译 (Core i7 Address Translation)
  - 9.7.2 Core i7页表条目 (Core i7 Page Table Entries)
  - 9.7.3 Core i7 TLB 操作 (Core i7 TLB Operation)
  - 9.7.4 Core i7 缓存操作 (Core i7 Cache Operation)
- 9.8 Linux虚拟内存系统 (Linux Virtual Memory System)
  - 9.8.1 Linux虚拟内存区域 (Linux Virtual Memory Areas)
  - 9.8.2 Linux缺页异常处理 (Linux Page Fault Handling)
- 9.9 内存映射 (Memory Mapping) (mmap函数是重点)
  - 9.9.1 再谈共享对象 (Shared Objects Revisited)
  - 9.9.2 fork函数如何工作 (How the fork Function Works)
  - 9.9.3 execve函数如何工作 (How the execve Function Works)
  - 9.9.4 使用mmap函数的用户级内存映射 (User-Level Memory Mapping with the mmap Function)
- 9.10 动态内存分配 (Dynamic Memory Allocation) (理解malloc和free的内部机制)

- 9.10.1 malloc和free函数 (The malloc and free Functions)
  - 9.10.2 为什么要使用动态内存分配 (Why Dynamic Memory Allocation?)
  - 9.10.3 分配器的需求和目标 (Allocator Requirements and Goals)
  - 9.10.4 碎片 (Fragmentation) (内部碎片和外部碎片)
  - 9.10.5 实现问题 (Implementation Issues)
  - 9.10.6 隐式空闲链表 (Implicit Free Lists)
  - 9.10.7 放置已分配的块 (Placing Allocated Blocks)
  - 9.10.8 分割空闲块 (Splitting Free Blocks)
  - 9.10.9 获取额外的堆内存 (Getting Additional Heap Memory)
  - 9.10.10 合并空闲块 (Coalescing Free Blocks)
  - 9.10.11 带边界标记的合并 (Coalescing with Boundary Tags)
  - 9.10.13 显式空闲链表 (Explicit Free Lists)
  - 9.10.14 分离的空闲链表 (Segregated Free Lists)
- 9.11 C程序中常见的与内存有关的错误 (Common Memory-Related Bugs in C Programs)
    - 9.11.1 解引用坏指针 (Dereferencing Bad Pointers)
    - 9.11.2 读未初始化的内存 (Reading Uninitialized Memory)
    - 9.11.3 允许栈缓冲区溢出 (Allowing Stack Buffer Overflows)
    - 9.11.4 假设指针和它们指向的对象是相同大小的 (Assuming that Pointers and the Objects They Point to Are the Same Size)
    - 9.11.5 造成错位错误 (Making Off-by-One Errors)
    - 9.11.6 引用指针，而不是它所指向的对象 (Referencing a Pointer Instead of the Object It Points To)
    - 9.11.7 误解指针运算 (Misunderstanding Pointer Arithmetic)
    - 9.11.8 引用不存在的变量 (Referencing Nonexistent Variables)
    - 9.11.9 引用空闲堆块中的数据 (Referencing Data in Free Heap Blocks)
    - 9.11.10 引起内存泄漏 (Introducing Memory Leaks)

## 第四部分：并发编程 (Concurrent Programming)

- 第12章：并发编程 (Concurrent Programming)  
(对现代多核编程非常重要)

- 12.1 基于进程的并发编程 (Process-Based Concurrent Programming)
- 12.2 基于I/O多路复用的并发编程 (Event-Based Concurrent Programming) (select, epoll)
  - 12.2.1 基本思想 (Basic Idea)
  - 12.2.2 I/O多路复用 (I/O Multiplexing)
  - 12.2.3 基于I/O多路复用的并发事件驱动服务器 (A Concurrent Event-Driven Server Based on I/O Multiplexing)
  - 12.2.5 优缺点 (Pros and Cons)
- 12.3 基于线程的并发编程 (Thread-Based Concurrent Programming)
  - 12.3.1 线程执行模型 (Thread Execution Model)
  - 12.3.2 Posix线程 (Posix Threads)
  - 12.3.3 创建线程 (Creating Threads)
  - 12.3.4 终止线程 (Terminating Threads)
  - 12.3.5 回收已终止线程的资源 (Reaping Terminated Threads)
  - 12.3.6 分离线程 (Detaching Threads)
  - 12.3.7 初始化线程 (Initializing Threads)
  - 12.3.8 基于线程的并发服务器 (A Thread-Based Concurrent Server)
- 12.4 多线程程序中的共享变量 (Shared Variables in Multithreaded Programs)
  - 12.4.1 线程内存模型 (Threads Memory Model)
  - 12.4.2 将变量映射到内存 (Mapping Variables to Memory)
  - 12.4.3 共享变量 (Shared Variables)
- 12.5 用信号量同步线程 (Synchronizing Threads with Semaphores) (理解信号量PV操作)
  - 12.5.1 进度图 (Progress Graphs)
  - 12.5.2 信号量 (Semaphores)
  - 12.5.3 使用信号量来实现互斥 (Using Semaphores for Mutual Exclusion)
  - 12.5.4 利用信号量来调度共享资源 (Using Semaphores to Schedule Shared Resources) (生产者-消费者问题, 读者-写者问题)
  - 12.5.5 基于预线程化的并发服务器 (A Prethreaded Concurrent Server)
- 12.6 使用线程提高并行性 (Using Threads for Parallelism)
- 12.7 其他并发问题 (Other Concurrency Issues) (线程安全, 可重入性, 死锁)

- 12.7.1 线程安全 (Thread Safety)
- 12.7.2 可重入性 (Reentrancy)
- 12.7.3 在线程化的程序中使用已存在的库函数 (Using Existing Library Functions in Threaded Programs)
- 12.7.4 竞争 (Races)
- 12.7.5 死锁 (Deadlocks)

### 其他章节的重要性：

- **第4章：处理器体系结构 (Processor Architecture):** 如果你对计算机组成原理有浓厚兴趣，或者想深入理解CPU如何执行指令，这一章值得细读。但对于大多数应用层开发者，了解第3章的机器级表示可能已经足够。
- 第10章：系统级I/O (System-Level I/O)
  - : 对于网络编程和系统编程非常重要，特别是文件I/O和Unix I/O的部分。
    - 10.1 Unix I/O (Unix I/O)
    - 10.2 打开和关闭文件 (Opening and Closing Files) (open, close)
    - 10.3 读和写文件 (Reading and Writing Files) (read, write)
    - 10.4 用RIO包健壮地读写 (Robust Reading and Writing with the RIO Package)
    - 10.5 读取文件元数据 (Reading File Metadata)
    - 10.6 共享文件 (Sharing Files)
    - 10.7 I/O重定向 (I/O Redirection)
    - 10.8 标准I/O (Standard I/O)
    - 10.9 综合：我应该使用哪些I/O函数？ (Putting It Together: Which I/O Functions Should I Use?)
  - 第11章：网络编程 (Network Programming)
    - : 如果你要进行网络相关的开发，这一章是必读的。
      - 11.1 客户端-服务器编程模型 (The Client-Server Programming Model)
      - 11.2 网络 (Networks)
      - 11.3 全球IP因特网 (The Global IP Internet)
        - 11.3.1 IP地址 (IP Addresses)
        - 11.3.2 因特网域名 (Internet Domain Names)
        - 11.3.3 因特网连接 (Internet Connections)

- 11.4 Sockets接口 (The Sockets Interface) (socket, connect, bind, listen, accept是核心)

- 11.4.1 socket地址结构 (Socket Address Structures)
- 11.4.2 socket函数 (The socket Function)
- 11.4.3 connect函数 (The connect Function)
- 11.4.4 bind函数 (The bind Function)
- 11.4.5 listen函数 (The listen Function)
- 11.4.6 accept函数 (The accept Function)
- 11.4.7 主机和服务的转换 (Host and Service Conversion)
- 11.4.8 套接字接口的辅助函数 (Helper Functions for the Sockets Interface)
- 11.4.9 示例：Echo客户端和服务器 (Example Echo Client and Server)

- 11.5 Web服务器 (Web Servers)

- 11.5.1 Web基础 (Web Basics)
- 11.5.2 Web内容 (Web Content)
- 11.5.3 HTTP事务 (HTTP Transactions)
- 11.5.4 服务动态内容 (Serving Dynamic Content)

## 总结一下：

### 1. 绝对核心，必须精读并做实验：

- 第2章 (信息表示)
- 第3章 (机器级表示)
- 第5章 (程序性能优化基础)
- 第6章 (存储器层次结构)
- 第8章 (异常控制流)
- 第9章 (虚拟内存)

### 2. 非常重要，强烈建议精读：

- 第1章 (系统漫游，建立整体概念)
- 第7章 (链接)
- 第12章 (并发编程)

### 3. 根据你的方向和兴趣选择性精读或作为后续学习：

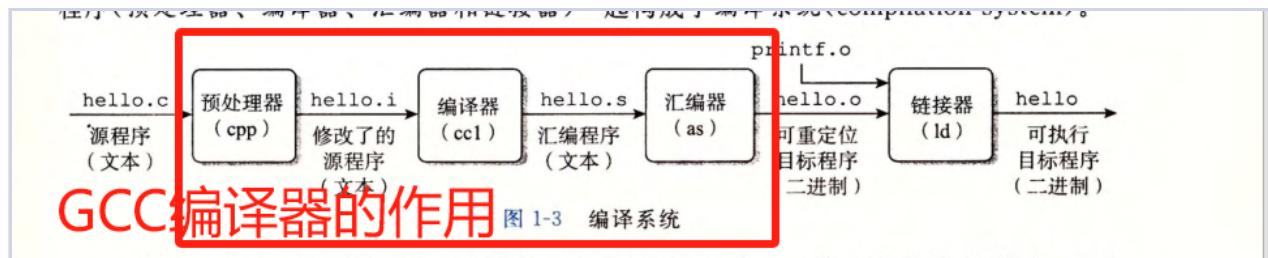
- 第4章 (处理器体系结构)

- 第10章 (系统级I/O)
- 第11章 (网络编程)

请记住，即使是“不太重要”的章节，也包含了有价值的信息。CSAPP的魅力在于它的全面性和深度。祝你学习愉快！

## 计算机系统漫游 程序的编译过程

就全面的讲，从编译再到运行的整个编译系统可以概括为：



1. 预处理器用来**处理#开头的注释或者头文件**
2. 编译器用来翻译成文本文件，是一个**汇编语言**程序（用符号和助记符）来表示原本的文本文件，主要是做一些基本的语法分析，如**定义的类型有无错误**
3. 汇编器：将汇编语言文件翻译成**机器语言**指令，生成**二进制的.o文件**。**包括一些地址分配**
4. 链接器：将hello程序中调用的一些库函数，如printf，找到其printf.o的位置，使某种方式合并\*链接）到我们源程序中

## shell

就是linux系统终端中的命令行：它一直在等待我们的输入，我们的命令。输入一个单词，先判断是不是shell内置命令，不是的话，就认为是一个可执行文件名

## 系统硬件组成

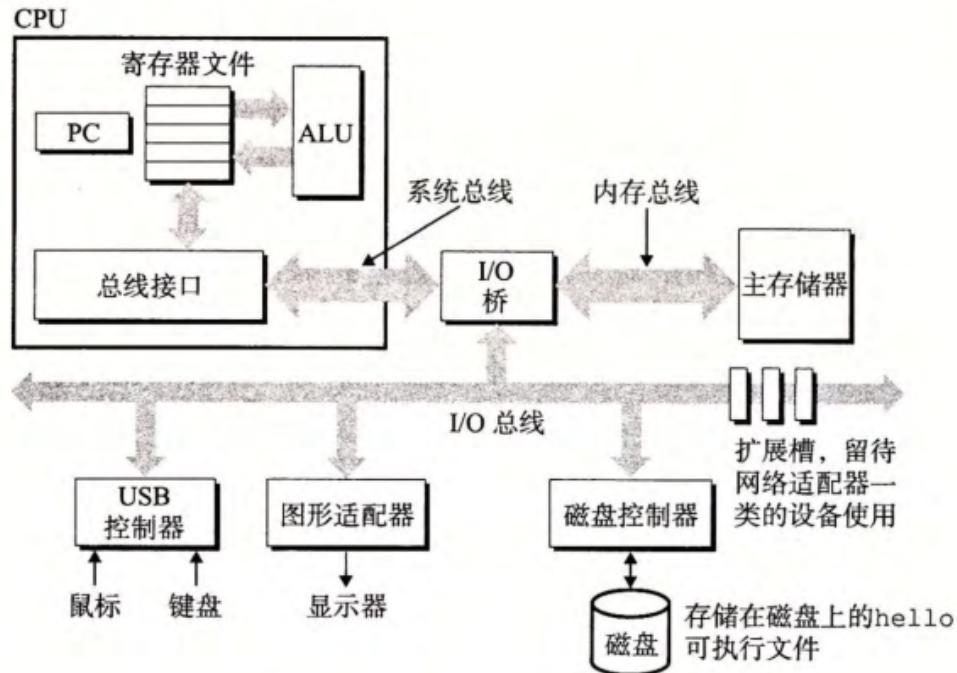


图 1-4 一个典型的系统的硬件组成

CPU：中央处理单元；ALU：算术/逻辑单元；PC：程序计数器；USB：通用串行总线

## 1. 总线Bus

贯穿整个系统的是组电子管道，称作总线，它携带信息字节并负责在各个部件间传递。通常总线被设计成传送定长的字节块，也就是字 (word)。就目前的计算机基本都是8个字节64位为一个字作为基本单位的

- **数据总线 (Data Bus)**：用于传输数据。
- **地址总线 (Address Bus)**：用于传输内存地址或设备地址。
- **控制总线 (Control Bus)**：用于传输控制信号，如读写信号、中断信号等。
- **系统总线 (System Bus)**：连接CPU、内存和I/O设备的总线，是**计算机内部的主要通信通道**。

## 2. I/O 设备

I/O(输入 / 输出)设备是系统与外部世界的联系通道。每个 I/O 设备都通过一个控制器或适配器与 I/O 总线(总线的一种)相连。控制器和适配器之间的区别主要在于它们的封装方式。**控制器是 I/O 设备本身或者系统的主印制电路板(通常称作主板)上的芯片组**。而**适配器则是一块插在主板插槽上的卡**。无论如何，它们的功能都是在 I/O 总线和 I/O 设备之间传递信息。

- **PCI总线 (Peripheral Component Interconnect)**：用于连接高速外设，如显卡、声卡等。
- **USB总线 (Universal Serial Bus)**：用于连接各种低速和中速外设，如键盘、鼠标、U盘等。
- **SATA总线 (Serial ATA)**：用于连接硬盘等存储设备。

- **SPI总线（Serial Peripheral Interface）**：用于连接低速的串行外设，如传感器、存储芯片等。

### 3. 主存

就像计算机的一个临时的草稿纸，这张草稿纸是由动态随机存取存储器（DRAM）芯片组成的，也就是说DRAM是主存的子集。这张草稿纸上的每一个位置当然对应着其唯一的数组。主存的特性是断电后数据丢失，正如一个ipad上的草稿纸一样，没电且没保存到硬盘中，草稿纸就没了

### 4. CPU

之所以说是64位的CPU是因为**处理器的核心是大小一个字（64位）的存储设备（寄存器），也叫做程序计数器（PC）**。程序计数器充当类似指针的作用，指向主存中的某个指令的地址。CPU不断的根据程序计数器指向的指令来执行相应的指令，并使PC指向下一个指令（不一定相邻按顺序）

一些常见的指令操作会涉及到主存、寄存器文件、**算术/逻辑单元（ALU）**。寄存器文件就是寄存器的集合，其中寄存器名字是唯一的。例如算术运算：两个寄存器充当算数，ALU充当算术运算符，得到的结果作为一个新的寄存器覆盖原来寄存器的内容。

## 程序在硬件中的工作流程

第一步：通过键盘这个输入设备输入文件名经过IO总线（未回车），文件名会通过系统总线读入寄存器中，之后通过内存总线存到内存中

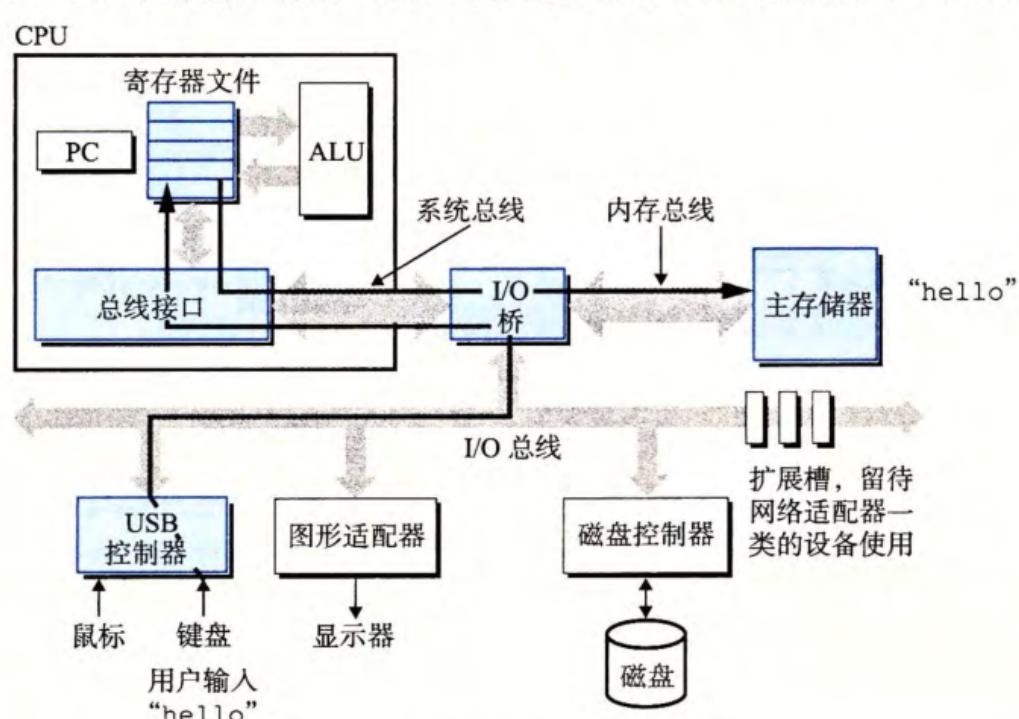


图 1-5 从键盘上读取 hello 命令

第二步：回车后，通过IO总线，将磁盘中的数据复制到主存（通过直接存储器存取DMA，不经过处理器，处理器只是发送根据指令执行命令的）

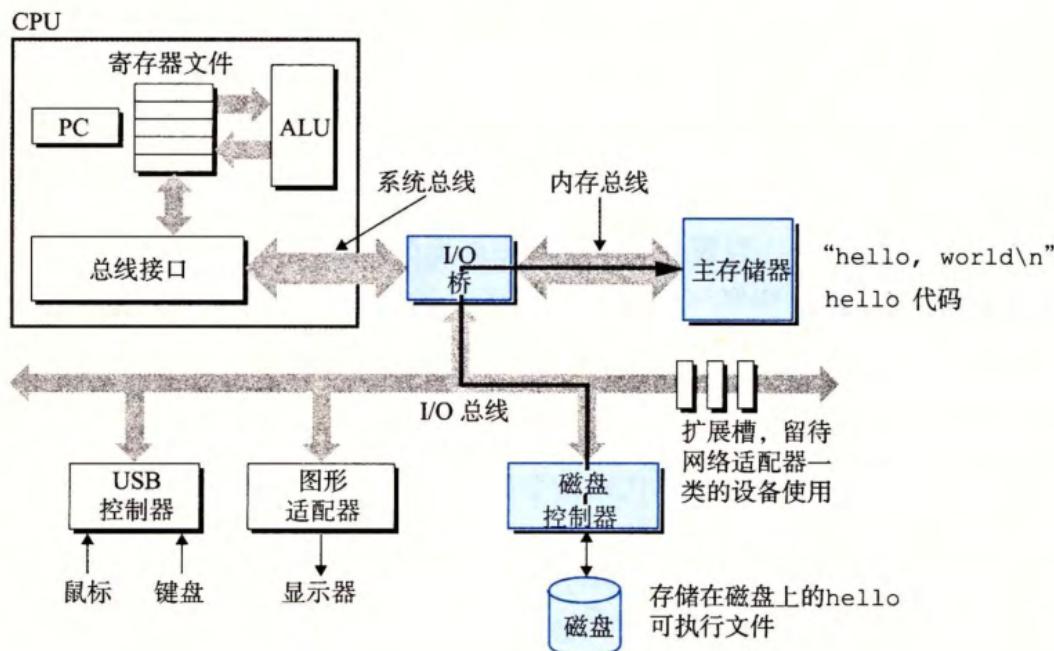


图 1-6 从磁盘加载可执行文件到主存

第三步：主存有内容后通过内存中下和系统总线，为寄存器文件提供相应的数据，同时把对应的信息输出到显示器上

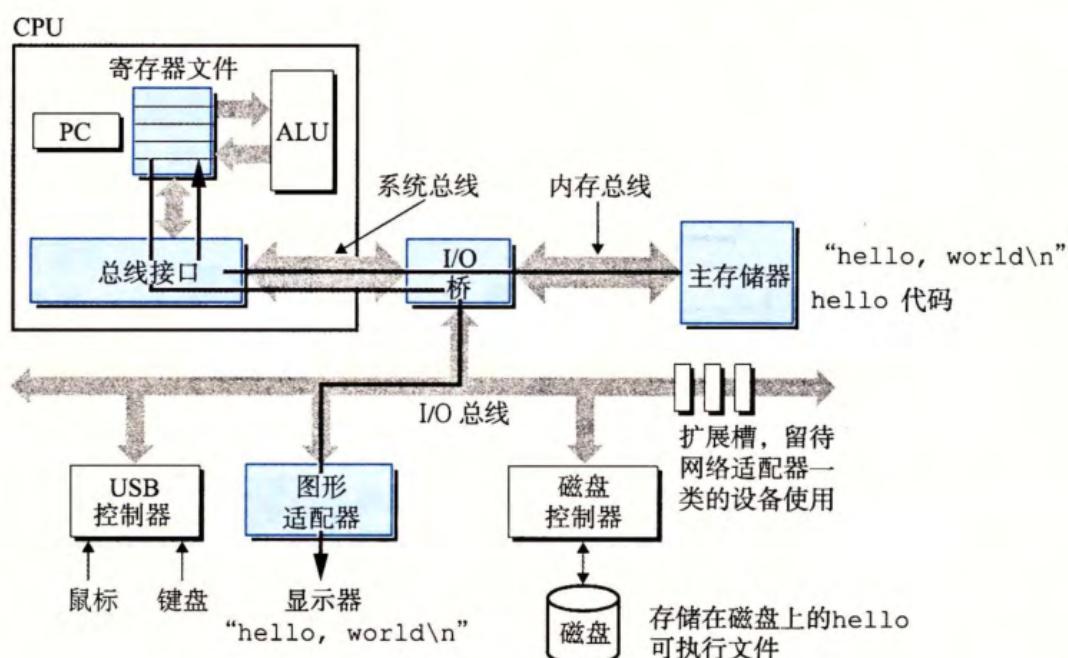


图 1-7 将输出字符串从存储器写到显示器

# 高速缓存cache

由于cpu和主存之间的运行速度有着巨大的差异，也就诞生了cache技术来解决这类问题。

cache分为L1、L2、L3，其中1-2是有一种静态随机访问存储器（SRAM）的硬件组成的。

## 存储设备的层次结构

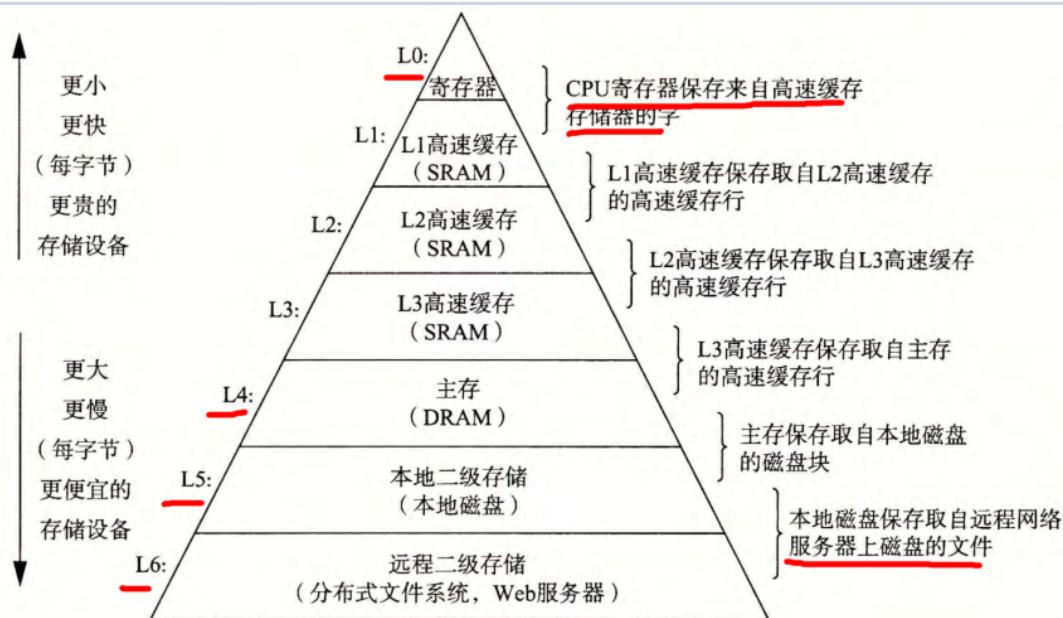


图 1-9 一个存储器层次结构的示例

存储器层次结构的主要思想是上一层的存储器作为低一层存储器的高速缓存。因此，寄存器文件就是 L1 的高速缓存，L1 是 L2 的高速缓存，L2 是 L3 的高速缓存，L3 是主存的高速缓存，而主存又是磁盘的高速缓存。在某些具有分布式文件系统的网络系统中，本地磁盘就是存储在其他系统中磁盘上的数据的高速缓存。

相邻的存储器，高的一层称为低的一层的高速缓存

## 操作系统管理硬件

## 1.7 操作系统管理硬件

让我们回到 hello 程序的例子。当 shell 加载和运行 hello 程序时，以及 hello 程序输出自己的消息时，shell 和 hello 程序都没有直接访问键盘、显示器、磁盘或者主存。取而代之的是，它们依靠操作系统提供的服务。我们可以把操作系统看成是应用程序和硬件之间插入的一层软件，如图 1-10 所示。所有应用程序对硬件的操作尝试都必须通过操作系统。

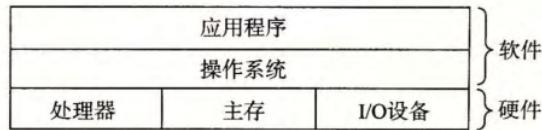


图 1-10 计算机系统的分层视图

操作系统有两个基本功能：(1)防止硬件被失控的应用程序滥用；(2)向应用程序提供简单一致的机制来控制复杂而又通常大不相同的低级硬件设备。操作系统通过几个基本的抽象概念（进程、虚拟内存和文件）来实现这两个功能。如图 1-11 所示，文件是对

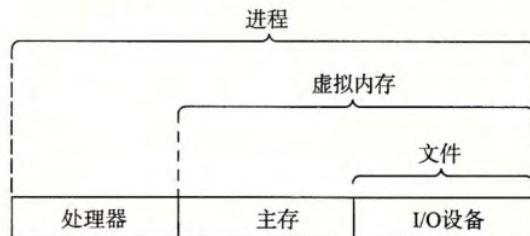


图 1-11 操作系统提供的抽象表示

I/O 设备的抽象表示，虚拟内存是对主存和磁盘 I/O 设备的抽象表示，进程则是对处理器、主存和 I/O 设备的抽象表示。我们将依次讨论每种抽象表示。

## 进程

进程是程序在操作系统中运行时的基本单位，包含了程序代码、执行信息等

操作系统使用了“上下文”的机制来实现并发运行（CPU 同时处理多个任务，但每次只能处理一个；区别于并行：同时运行多个任务，一起推进任务）

- 并发：一个人同时吃三个馒头。
- 并行：三个人同时吃三个馒头。

上下文，包括 PC 和寄存器文件的值，主存内容，为操作系统提供了保持跟踪进程运行所需的“所有运行信息”

而实际的进行进程之间的切换是由系统内核（Kernel）来完成和管理的。每一次的进程切换后，会根据其他进程中保存起来的上下文信息，回复到中断的地方继续执行

## 进程切换的步骤

以下是进程切换的典型步骤，由内核完成：

- 1. 选择下一个进程：**内核的调度器Kernel根据调度算法选择下一个要运行的进程。
- 2. 保存当前进程的上下文：**内核保存当前进程的寄存器状态、栈状态、内存映射信息等。
- 3. 恢复下一个进程的上下文：**内核恢复下一个进程的寄存器状态、栈状态等，使其能够从中断的地方继续执行。
- 4. 切换进程：**内核将控制权交给下一个进程，使其开始运行。

## 线程

**线程是进程中的一个执行单元**，线程运行在某一个进程中的上下文中，共同享有代码和数据

线程比进程的资源消耗更小，开辟和实现多线程也就比多进程更容易

## 虚拟内存

==以进程虚拟内存的形式寄生在磁盘上，并以此利用主存作为磁盘的高速缓存来减少之间的消耗

虚拟内存的核心就在于虚拟，“假象”，欺骗进程，让进程以为当前是内存，真面目却是在硬盘上的。由于“假象”的作用，导致每个进程看到的内存都是一致的，确保了“一致性”。

用户进程定义的代码和数据。请注意，图中的地址是从下往上增大的。

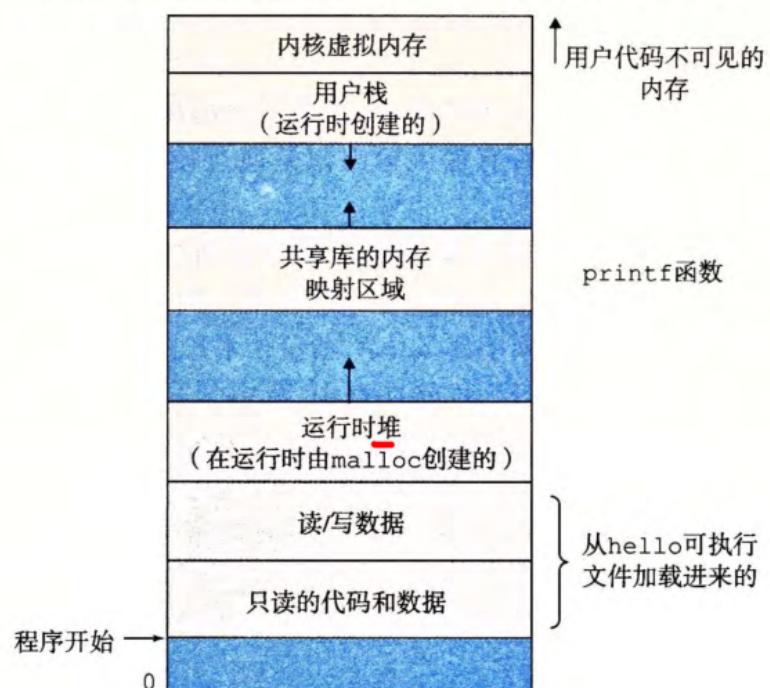


图 1-13 进程的虚拟地址空间

毕竟肯定先要有相应的程序，的地址，那才有调用堆、栈这些的地址，一直传递到内核虚拟内存，也就是与kernel相关的

1. 代码（二进制）和数据（全局变量、静态变量、常量）在程序运行时占用的内存空间是固定部分
2. 而可变部分包括：用户栈（用户运行时产生的形参、局部变量、返回值、递归空间，由系统自动回收）和运行堆区（包括动态开辟的空间，如malloc或new，需要手动回收）
3. 固定区所占用的空间非常小，运行时消耗的内存主要取决于可变部分。
4. 由于堆区系统不会自动回收，所以很容易出现内存泄漏的问题，但是java、python中不需要考虑

- 程序代码和数据。对所有的进程来说，代码是从同一固定地址开始，紧接着的是和C全局变量相对应的数据位置。代码和数据区是直接按照可执行目标文件的内容初始化的，在示例中就是可执行文件hello。在第7章我们研究链接和加载时，你会学习更多有关地址空间的内容。
- 堆。代码和数据区后紧随着的是运行时堆。代码和数据区在进程一开始运行时就被指定了大小，与此不同，当调用像malloc和free这样的C标准库函数时，堆可以在运行时动态地扩展和收缩。在第9章学习管理虚拟内存时，我们将更详细地研究堆。
- 共享库。大约在地址空间的中间部分是一块用来存放像C标准库和数学库这样的共享库的代码和数据的区域。共享库的概念非常强大，也相当难懂。在第7章介绍动态链接时，将学习共享库是如何工作的。
- 栈。位于用户虚拟地址空间顶部的是用户栈，编译器用它来实现函数调用。和堆一样，用户栈在程序执行期间可以动态地扩展和收缩。特别地，每次我们调用一个函数时，栈就会增长；从一个函数返回时，栈就会收缩。在第3章中将学习编译器是如何使用栈的。
- 内核虚拟内存。地址空间顶部的区域是为内核保留的。不允许应用程序读写这个区域的内容或者直接调用内核代码定义的函数。相反，它们必须调用内核来执行这些操作。

## 文件

文件是以字节为单位的，因此也是一个字节序列。IO设备包括网络都算做文件体系

## 为什么说网络也是IO设备



代系统经常通过网络和其他系统连接到一起。从一个单独的系统来看，网络可视为一个I/O设备，如图1-14所示。当系统从主存复制一串字节到网络适配器时，数据流经过网络到达另一台机器，而不是比如说到达本地磁盘驱动器。相似地，系统可以读取从其他机器发送来的数据，并把数据复制到自己的主存。

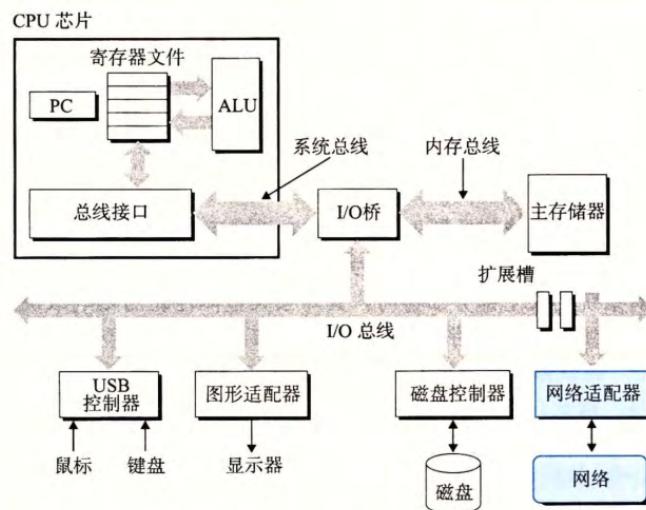


图1-14 网络也是一种I/O设备

随着Internet这样的全球网络的出现，从一台主机复制信息到另外一台主机已经成为计算机系统最重要的用途之一。比如，像电子邮件、即时通信、万维网、FTP和telnet这样的应用都是基于网络复制信息的功能。

回到hello示例，我们可以使用熟悉的telnet应用在一个远程主机上运行hello程序。假设用本地主机上的telnet客户端连接远程主机上的telnet服务器。在我们登录到远程主机并运行shell后，远端的shell就在等待接收输入命令。此后在远端运行hello程序包括如图1-15所示的五个基本步骤。



图1-15 利用telnet通过网络远程运行hello

## 并发和并行

### 1. 线程级并发

一个进程中多个线程同时执行，也就是多个控制流

操作系统的并发本质是操作系统中多个进程（资源分配的基本单位）间的快速切换。就好像人类多任务处理一样，吃饭和看手机，看似是两个同时执行，其实也是并发的概念：在吃饭和手机中快速来回切换。

我现在的电脑是将多个CPU（也叫“核”）集成到一个集成电路芯片上的一个多核处理器

其中的每个核都有自己独立的L1（包括数据和指令）高速缓存和L2高速缓存，但是只有一个所有核共享的L3高速缓存

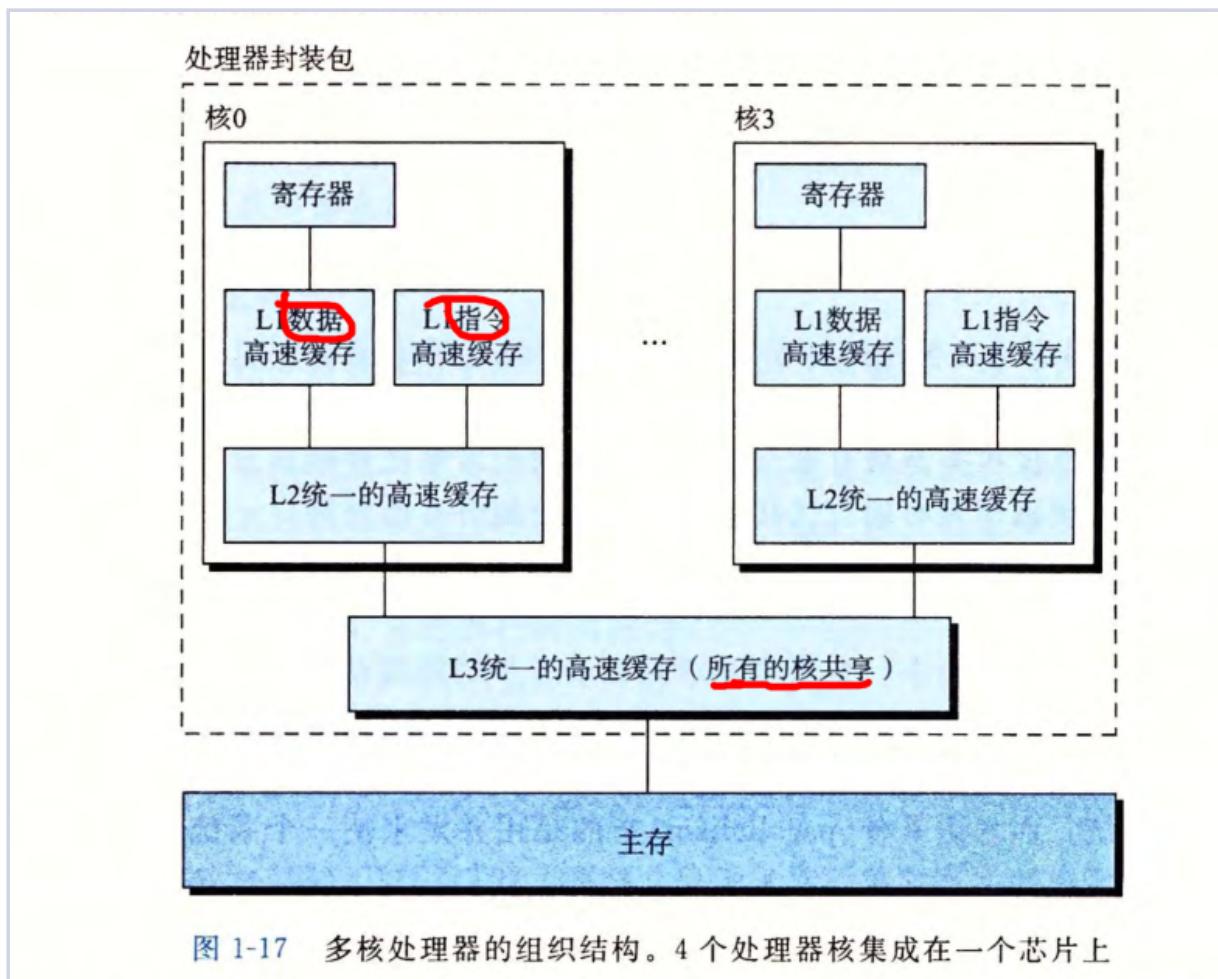


图 1-17 多核处理器的组织结构。4 个处理器核集成在一个芯片上

普通的一核心就是一线程，但是超线程就可以两线程

## 2. 指令集并行

当代的先进的cpu使用流水线来同时执行多条指令，使处理器能够在一个周期的时间内，执行比一个周期还多的指令数，并且这称之为超标量处理器

## 3. 单指令、多数据并行

如同字面意思，一条指令可以并行多条数据那么称之为SIMD

体系结构类型	结构	关键特性	代表
单指令流单数据流 SISD	控制部分：一个 处理 器：一个 主存模块：一个		单处理器系统
单指令流多数据流 SIMD	控制部分：一个 处理 器：多个 主存模块：多个	各处理器以异步的形式执行同一条指令	并行处理机 阵列处理机 超级向量处理机
多指令流单数据流 MISD	控制部分：多个 处理 器：一个 主存模块：多个	被证明不可能，至少是不实际	目前没有，有文献称流水线计算机为此类
多指令流多数据流 MIMD	控制部分：多个 处理 器：多个 主存模块：多个	能够实现作业、任务、指令等各级全面并行	多处理器系统 多计算机

## 小结

### 1.10 小结

计算机系统是由硬件和系统软件组成的，它们共同协作以运行应用程序。计算机内部的信息被表示为一组组的位，它们依据上下文有不同的解释方式。程序被其他程序翻译成不同的形式，开始时是 ASCII 文本，然后被编译器和链接器翻译成二进制可执行文件。

处理器读取并解释存放在主存里的二进制指令。因为计算机花费了大量的时间在内存、I/O 设备和 CPU 寄存器之间复制数据，所以将系统中的存储设备划分成层次结构——CPU 寄存器在顶部，接着是多层的硬件高速缓存存储器、DRAM 主存和磁盘存储器。在层次模型中，位于更高层的存储设备比低层的存储设备要更快，单位比特造价也更高。层次结构中较高层次的存储设备可以作为较低层次设备的高速缓存。通过理解和运用这种存储层次结构的知识，程序员可以优化 C 程序的性能。

操作系统内核是应用程序和硬件之间的媒介。它提供三个基本的抽象：1)文件是对 I/O 设备的抽象；2)虚拟内存是对主存和磁盘的抽象；3)进程是处理器、主存和 I/O 设备的抽象。

最后，网络提供了计算机系统之间通信的手段。从特殊系统的角度来看，网络就是一种 I/O 设备。

## 信息的表示和处理

显然，信息的表示是用0、1的数字表示的，称之为位

没想到有符号整数的最常见的表示方式竟然是补码编码

可以了解到整数的数值表示的范围是很小的，但是却是一个很精确的数值；相对于浮点数而言，浮点数的数值范围很大，但相应的失去了精度，也就代表了浮点数的值的近似的

## 信息存储

在内存或的任意的存储设备之中，作为最小的可寻址的内存单位，那首先想到的就是字节了，也就是8位

那么，以字节为最小的可寻址内存单位的内存中的，每一个字节，都可以用一串唯一的数字来表示，这个数字就是该字节的地址，地址的集合也就构成了虚拟内存空间。

注意区别虚拟地址

那么同理，在C语言当中，每个指针的值，都指向着对应的存储块的第一个字节的虚拟地址，例如指向字符数组的指针一样，是只指向  $a[0]$  的地址的

程序是很多字节组成的**字节序列**，而**程序对象**就只是这些序列中的一个**字节块**

## 信息存储位表示中常见的缩写含义：

1. **HEX**：十六进制 Hexadecimal

2. **OCT**：八进制 Octal

3. **DEC**：十进制 Decimal

4. **BIN**：二进制 Binary

以下内容对应下一小节的内容 ↓

5. 字**Word**: 两个字节称为一个字，即16位

6. 双字**Dword**: 两个字称为一个双字，两个Word，为32位 D为double

7. 四字**Qword**: 两个双字称为一个四字，四个Word，为64位 Q为 quadra

## 字（字长）数据大小

用来指明指针数据的指标的大小为一个**字长（位）**，存在与每一台计算机当中。

而**虚拟地址是以每一位来编码的，所以字节决定了字长（位），也就决定了虚拟地址空间的大小**

**虚拟地址空间并不直接等同于内存，内存更像是物理地址空间，而虚拟地址空间决定了内存理论上的上限，而内存是真实的实现。**

而虚拟地址空间和物理地址空间直接的转换一般是，虚拟地址空间通过页表映射成为物理地址。而具体的映射技术（虚拟内存管理）技术包括分页、分段的方式，毕竟，肯定不能一一映射啦，那效率太低了~

对于一个字节为  $w$  位的机器，对应的虚拟地址范围总共有  $2^w$  个字节：  $0 \sim 2^w - 1$

之所以这里指数函数的底是2，是因为**信息以位为单位的表示只有0和1两种**

那么，32位字长的虚拟地址空间为4GB，而64位字长的虚拟地址空间有16EB，因此，内存是完全够用的，大概也就不会出现128位的系统了~

## 寻址和字节顺序（大端法与小端法）

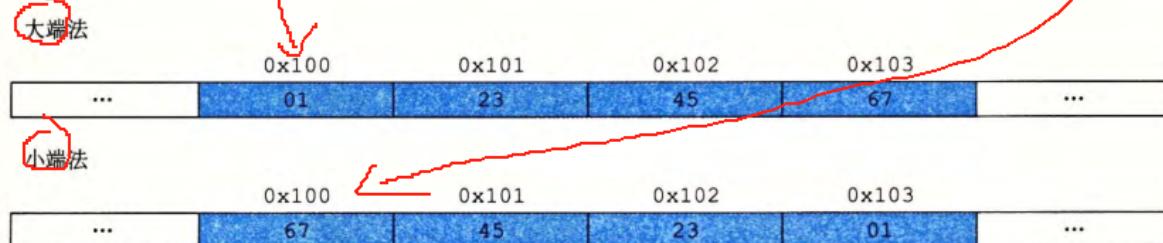
## 也叫排列一个对象的字节的两种规则

一个程序对象，不可能说只占用一个字节，那么在占用多个字节的程序对象中，它的存储肯定是一个连续的字节序列，并且该程序对象的地址为该连续的字节序列中的最小地址，也就是第一个地址（因为这些地址是按顺序排列的）。例如一个4字节的int的变量x的地址为0x100，那么其对应的字节序列就是相应的四个地址，即0x100、0x101、0x102、0x103

字节排列顺序分为小端法和大端法：

排列表示一个对象的字节有两个通用的规则。考虑一个w位的整数，其位表示为 $[x_{w-1}, x_{w-2}, \dots, x_1, x_0]$ ，其中 $x_{w-1}$ 是最有效位，而 $x_0$ 是最低有效位。假设w是8的倍数，这些位就能被分组成为字节，其中最高有效字节包含位 $[x_{w-1}, x_{w-2}, \dots, x_{w-8}]$ ，而最低有效字节包含位 $[x_7, x_6, \dots, x_0]$ ，其他字节包含中间的位。某些机器选择在内存中按照从最低有效字节到最高有效字节的顺序存储对象，而另一些机器则按照从最高有效字节到最低有效字节的顺序存储。前一种规则——最低有效字节在最前面的方式，称为小端法(little endian)。后一种规则——最高有效字节在最前面的方式，称为大端法(big endian)。

假设变量x的类型为int，位于地址0x100处，它的十六进制值为0x01234567。地址范围0x100~0x103的字节顺序依赖于机器的类型：



一个字节（8位二进制）对应的是2个十六进制数

然鹅遗憾的是，目前inter处理器是用的小端法的，有点没想到。而且在ARM架构中，只能运行小端模式

也就是说，一般而言，书写字节序列的方式是：低位的字节 代表的数据放在左边，而高位的字节 代表的数据放在右边

然而，这个字节序列的排列方式只针对二进制数据生效，对于文本数据也就是ASCII数据来说，都是一致的，不存在不同系统不同的情况，也就是说文本数据中就没有排列这个概念了

总结就是，目前消费级cpu、ARM架构都是小端法

## 编码问题

以字符编码为代表的ASCII编码，其每个字符占用一个字节，但是它的局限性是没办法支持中文以及对很多符号都不支持，表示的范围还太小了。不过，中国制定了GB2312编码，用来把中文编进去。

为了解决这个问题，推出了Unicode编码（统一字符集），可以完美的适用于中文之中，它用16位来表示字符，也就是说每个字符都要占用2个字节来表示。它最常用的是UCS-16编码，用两个字节表示一个字符（如果要用到非常偏僻的字符，就需要4个字节）。

Unicode是固定为字符分配码点的标准，也可以理解为一般是定长了的，要动只能在2字节或4字节中动

为了更好的优化，推出了可变长的utf-8编码，它可以完美适配ASCII，一个文本数据它只分配1个字节，而如果一个汉字的话它可以分配到4个字节，它是动态调整的。

在某系统或某编译器上的二进制代码（机器语言），并不能适用于每个系统或编译器中，即使都是二进制代码，即**二进制代码是不兼容的**

二进制不兼容不代表着就是不可移植或者可移植性差的，可移植性好指的是源代码可移植性好，并非是编译后生成的二进制文件。

## 布尔代数

### 二进制值是计算机编码、存储和操作信息的核心

**二进制值是计算机编码、存储和操作信息的核心**，所以围绕数值0和1的研究已经演化出了丰富的数学知识体系。这起源于1850年前后乔治·布尔(George Boole, 1815—1864)的工作，因此也称为布尔代数(Boolean algebra)。布尔注意到通过将逻辑值TRUE(真)和FALSE(假)编码为二进制值1和0，能够设计出一种代数，以研究逻辑推理的基本原则。

最简单的布尔代数是在二元集合{0, 1}基础上的定义。图2-7定义了这种布尔代数中的几种运算。我们用来表示这些运算的符号与C语言位级运算使用的符号是相匹配的，这些将在后面讨论到。**布尔运算~对应逻辑运算 NOT，在命题逻辑中用符号~**

表示。也就是说，当P不是真的时候，我

们就说 $\neg P$ 是真的，反之亦然。相应地，当P等于0时， $\neg P$ 等于1，反之亦然。**布尔运算& 对应于逻辑运算 AND，在命题逻辑中用符号 $\wedge$  表示**

。当P和Q都为真时，我们说 $P \wedge Q$ 为真。相应地，只有当 $p=1$ 且 $q=1$ 时， $p \wedge q$ 才等于1。**布尔运算| 对应于逻辑运算 OR，在命题逻辑中用符号 $\vee$  表示**

。当P或者Q为真时，我们说 $P \vee Q$ 成立。相应地，当 $p=1$ 或者 $q=1$ 时， $p \vee q$ 等于1。**布尔运算^ 对应于逻辑运算异或，在命题逻辑中用符号 $\oplus$  表示**

。当P或者Q为真但不同时为真时，我们说 $P \oplus Q$ 成立。相应地，当 $p=1$ 且 $q=0$ ，或者 $p=0$ 且 $q=1$ 时， $p \oplus q$ 等于1。

$\sim$	&		$\wedge$
0	0 0	0 1	0 0
1	1 0	1 1	1 1

图2-7 布尔代数的运算。二进制值1和0表示逻辑值TRUE或者FALSE，而运算符~、&、|和 $\wedge$ 分别表示逻辑运算NOT、AND、OR和EXCLUSIVE-OR

注意and里的命题逻辑是倒写的大V，而异或是倒写的小v

# 位移运算

## 1. 左移

就是丢弃最高位k位，右侧填充0

## 2. 右移：分为逻辑右移和算术右移

a. **逻辑右移**：同左移一样，适用于无符号数

b. **算术右移**：左侧填充时，根据原值x的最高位，如果最高位是1就填充1，否则0。例如 $1000 \gg 2 == 1110$ ，这里用了最高位1填充了左侧。适用于有符号数

**如果没有告知有无符号，则默认1开头的（负数）为有符号数，此时右移就是算数右移**

无符号数只能逻辑右移

**位移量k = 移动n位 % w位的数据**

对于一个由w位组成的数据类型，如果要移动 $k \geq w$ 位会得到什么结果呢？例如，计算下面的表达式会得到什么结果，假设数据类型int为w=32：

```
int lval = 0xFEDCBA98 << 32;
int aval = 0xFEDCBA98 >> 36;
unsigned uval = 0xFEDCBA98u >> 40;
```

C语言标准很小心地规避了说明在这种情况下该如何做。在许多机器上，当移动一个w位的值时，移位指令只考虑位移量的低 $\log_2 w$ 位，因此实际上位移量就是通过计算 $k \bmod w$ 得到的。例如，当w=32时，上面三个移位运算分别是移动0、4和8位，得到结果：

lval	0xFEDCBA98
aval	0xFFEDCBA9
uval	0x00FEDCBA

# 加法逆元（异或运算）

例如任意整数x都有一个加法逆元“-x”，使 $x + (-x) = 0$ 。而在布尔代数中，我们用“^”，也就是异或来表示加法逆元

如 $x \wedge x = 0$

满足以下性质的一个数 $\neg_w^{u/t} x$ ，称之为加法逆元：

$$(x + \neg_w^{u/t} x) \% 2^w == 0$$

**异或运算的性质：**

- 交换律：调整运算顺序不影响结果。如 $x \wedge (x \wedge y) == (x \wedge x) \wedge y == 0 \wedge y == y$

- 异或操作针对每个二进制位，结果是一个完整的二进制数。如4位的运算  $0 \wedge y == 0000$   
 $\wedge xxxx == xxxx == y$

## 整数表示

包括有无符号数的char、int、long

本章中用到的缩写的意思：

1. T——补码
2. B——原码
3. U——无符号数

## 无符号数的编码

无符号数和它的编码（二进制）关系是双射的关系。即无**符号数可以唯一确认一个二进制编码**，而**一个二进制编码也能唯一确认一个无符号数**，也就是说将这二者定义成一个函数，他们可以用反函数来表示，存在反函数。

无符号数的第一位不是符号位

## 有符号数的编码

### 原码和反码（Ones' complement）

假设有一个整数数据类型有  $w$  位。我们可以将位向量写成  $\vec{x}$ , 表示整个向量, 或者写成  $[x_{w-1}, x_{w-2}, \dots, x_0]$ , 表示向量中的每一位。把  $\vec{x}$  看做一个二进制表示的数, 就获得

了  $\vec{x}$  的无符号表示。在这个编码中, 每个位  $x_i$  都取值为 0 或 1, 后一种取值意味着数值  $2^i$  应为数字值的一部分。我们用一个函数  $B2U_w$  (Binary to Unsigned 的缩写, 长度为  $w$ ) 来表示:

原理: 无符号数编码的定义

对向量  $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$ :

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \quad (2.1)$$

在这个等式中, 符号 “ $\doteq$ ” 表示左边被定义为等于右边。函数  $B2U_w$  将一个长度为  $w$  的 0、1 串映射到非负整数。举一个示例, 图 2-11 展示的是下面几种情况下  $B2U$  给出的从位向量到整数的映射:

$$\begin{aligned} B2U_4([0001]) &= 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1 \\ B2U_4([0101]) &= 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \\ B2U_4([1011]) &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11 \\ B2U_4([1111]) &= 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15 \end{aligned} \quad (2.2)$$

#### 旁注 有符号数的其他表示方法

有符号数还有两种标准的表示方法:

反码(Ones' Complement): 除了最高有效位的权是  $-(2^{w-1} - 1)$  而不是  $-2^{w-1}$ , 它和补码是一样的:

$$B2O_w(\vec{x}) \doteq -x_{w-1}(2^{w-1} - 1) + \sum_{i=0}^{w-2} x_i 2^i$$

原码(Sign-Magnitude): 最高有效位是符号位, 用来确定剩下的位应该取负权还是正权:

$$B2S_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left( \sum_{i=0}^{w-2} x_i 2^i \right)$$

这两种表示方法都有一个奇怪的属性, 那就是对于数字 0 有两种不同的编码方式。这两种表示方法, 把  $[00\dots0]$  都解释为 +0。而值 -0 在原码中表示为  $[10\dots0]$ , 在反码中表示为  $[11\dots1]$ 。虽然过去生产过基于反码表示的机器, 但是几乎所有的现代机器都使用补码。我们将看到在浮点数中有使用原码编码。

补码基础上+1

反码的目的是为了将减法运算转变成一个用负值表示的加法运算, 即统一减法为加法:  $A - B = A + (-B)$ ; 补码的目的在于消除-0,

原码、反码、补码是针对有符号数而言的, 而有符号数中, 正数值的原码、反码、补码都相同, 只有负数时需要考虑反码和补码

## 补码 (Two's complement)

## 补码是用来表示一个有符号（正数或负数）数的

### 2.2.3 补码编码

对于许多应用，我们还希望表示负数值。最常见的有符号数的计算机表示方式就是补码(two's-complement)形式。在这个定义中，将字的最高有效位解释为负权(negative weight)。我们用函数  $B2T_w$ (Binary to Two's-complement) 的缩写，长度为  $w$  来表示：

原理：补码编码的定义

对向量  $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$ ：

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \quad (2.3)$$

最高有效位  $x_{w-1}$  也称为符号位，它的“权重”为  $-2^{w-1}$ ，是无符号表示中权重的负数。符号位被设置为 1 时，表示值为负，而当设置为 0 时，值为非负。这里来看一个示例，图 2-13 展示的是下面几种情况下  $B2T$  给出的从位向量到整数的映射。

$$B2T_4([0001]) = -0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 0 + 0 + 1 = 1$$

$$B2T_4([0101]) = -0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 0 + 4 + 0 + 1 = 5 \quad (2.4)$$

✓  $B2T_4([1011]) = -1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 0 + 2 + 1 = -5$

✓  $B2T_4([1111]) = -1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = -8 + 4 + 2 + 1 = -1$

如图中所示，原码 1011 的补码为 1101，即 -5，就是上面求得的结果。

所以负数原码计算其真实值（最高位为1的补码）有两种方式：

1. 将原码最高位的位权取负，其他位权取正，相加的结果就是该值的补码（真值）
2. 将原码除最高位外取反，之后加1

因此补码的最小值为100...0即只有最高位的负值，即  $-2^{w-1}$

最大值为 011...1，去除了最高位的权值，即

$$TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$$

## 无符号数和补码的表示范围

因此，只用记下面特殊的数据类型表示的大致范围：

数据类型

范围

Char	-127 - 128
Short (有符号) 2字节16位	$-32768 - 32767 = -2^{15} - 2^{15} - 1$
Short (无符号) 2字节16位	0 - 65535 ( $2^{16} - 1$ )
Int (有符号) 4字节32位	$-2147483648 - 2147483647 = -2^{31} - 2^{31} - 1$
Int (无符号) 4字节32位	$-4294967296 - 4294967295 = -2^{32} - 2^{32} - 1$

只需记:  $2^{15} = 32768$  和  $2^{31} = 2147483648$

## 无符号数和补码的关系

- $|TMin| = |TMax| + 1$  (因为正数里还包括了0)
- 16位位模式的情况下, 有:  $|补码| + |无符号数表示| = 2^{16}$

如

$U2T_{16}(53\ 191) = -12\ 345$ 。也就是说, 十六进制表示写作 0xCFC7 的 16 位位模式既是  $-12\ 345$  的补码表示, 又是 53 191 的无符号表示。同时请注意  $12\ 345 + 53\ 191 = 65\ 536 = 2^{16}$ 。这个属性可以推广到给定位模式的两个数值(补码和无符号数)之间的关系。类似地,

### 3. 补码转换为无符号数T2U:

原理: 补码转换为无符号数

对满足  $TMin_w \leq x \leq TMax_w$  的  $x$  有:

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \quad (2.5)$$

比如, 我们看到  $T2U_{16}(-12\ 345) = -12\ 345 + 2^{16} = 53\ 191$ , 同时  $T2U_w(-1) = -1 + 2^w = UMax_w$ 。

无符号数只有正值, 所以当补码为负值时, 要想办法搞成正值

推导:

### 原理：补码转换为无符号数

对满足  $TMin_w \leq x \leq TMax_w$  的  $x$  有：

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \quad \text{无符号数不能有负的} \quad (2.5)$$

比如，我们看到  $T2U_{16}(-12345) = -12345 + 2^{16} = 53191$ ，同时  $T2U_w(-1) = -1 + 2^w = UMax_w$ 。

该属性可以通过比较公式(2.1)和公式(2.3)推导出来。

### 推导：补码转换为无符号数

比较等式(2.1)和等式(2.3)，我们可以发现对于位模式  $\vec{x}$ ，如果我们计算  $B2U_w(\vec{x}) - B2T_w(\vec{x})$  之差，从 0 到  $w-2$  的位的加权和将互相抵消掉，剩下一个值： $B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - (-2^{w-1})) = x_{w-1}2^w$ 。这就得到一个关系： $B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$ 。我们因此就有

$$B2U_w(T2B_w(x)) = T2U_w(x) = x + x_{w-1}2^w \quad (2.6)$$

根据公式(2.5)的两种情况，在  $x$  的补码表示中，位  $x_{w-1}$  决定了  $x$  是否为负。 ■

比如说，图 2-16 比较了当  $w=4$  时函数  $B2U$  和  $B2T$  是如何将数值变成位模式的。对补码来说，最高有效位是符号位，我们用带向左箭头的条来表示。对于无符号数来说，最高有效位是正权重，我们用带向右的箭头的条来表示。从补码变为无符号数，最高有效位

的权重从  $-8$  变为  $+8$ 。因此，补码表示的负数如果看成无符号数，值会增加  $2^4 = 16$ 。因而， $-5$  变成了  $+11$ ，而  $-1$  变成了  $+15$ 。

## 4. 无符号数转为补码U2T：

### 原理：无符号数转换为补码

对满足  $0 \leq u \leq UMax_w$  的  $u$  有：

$$U2T_w(u) = \begin{cases} u, & u \leq TMax_w \\ u - 2^w, & u > TMax_w \end{cases} \quad (2.7)$$

当无符号数值超出补码最大值时，要想办法搞成补码范围的值。一般超出多少，就从补码最小值往右多少，如图所示：

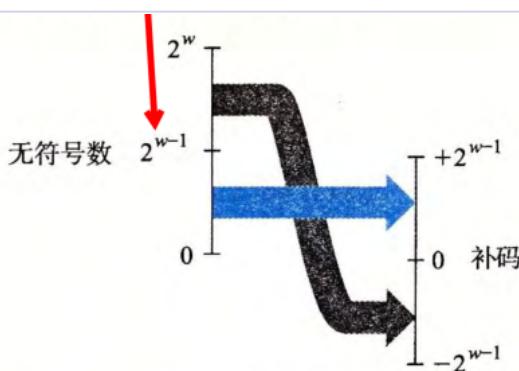


图 2-18 从无符号数到补码的转换。函数  $U2T$  把大于  $2^{w-1}-1$  的数字转换为负值

推导：

原理：无符号数转换为补码

对满足  $0 \leq u \leq UMax_w$  的  $u$  有：

$$U2T_w(u) = \begin{cases} u, & u \leq TMax_w \\ u - 2^w, & u > TMax_w \end{cases} \quad (2.7)$$

该原理证明如下：

推导：无符号数转换为补码

设  $\vec{u} = U2B_w(u)$ , 这个位向量也是  $U2T_w(u)$  的补码表示。公式(2.1)和公式(2.3)结合起来有

$$U2T_w(u) = -u_{w-1}2^{w-1} + u \quad (2.8)$$

在  $u$  的无符号表示中，对公式(2.7)的两种情况来说，位  $u_{w-1}$  决定了  $u$  是否大于  $TMax_w = 2^{w-1} - 1$ 。

图 2-18 说明了函数  $U2T$  的行为。对于小的数 ( $\leq TMax_w$ )，从无符号到有符号的转换将保留数字的原值。对于大的数 ( $> TMax_w$ )，数字将被转换为一个负数值。

## 5. $1 + UMax_w = 2^w$ | $UMax_w = 2^w - 1$

这三条公式具体例子可以参考下图：

图 2-14 展示了针对不同字长，几个重要数字的位模式和数值。前三个给出的是可表示的整数的范围，用  $UMax_w$ 、 $TMin_w$  和  $TMax_w$  来表示。在后面的讨论中，我们还会经常引用到这三个特殊的值。如果可以从上下文中推断出  $w$ ，或者  $w$  不是讨论的主要内容时，我们会省略下标  $w$ ，直接引用  $UMax$ 、 $TMin$  和  $TMax$ 。

数	字长 $w$			
	8	16 无符号则显示这里	32	64
$UMax_w$	0xFF 255	0xFFFF 65 535	0xFFFFFFFF 4 294 967 295	0xFFFFFFFFFFFFFF 18 446 744 073 709 551 615
$TMin_w$	0x80 -128	0x8000 -32 768	0x80000000 -2 147 483 648	0x8000000000000000 -9 223 372 036 854 775 808
$TMax_w$	0x7F 127	0x7FFF 32 767	0x7FFFFFFF 2 147 483 647	0x7FFFFFFFFFFFFF 9 223 372 036 854 775 807
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFF 0xFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

T 表示补码的 Two's

图 2-14 重要的数字。图中给出了数值和十六进制表示

为了更好地理解补码表示，考虑下面的代码：

```

1     short x = 12345;
2     short mx = -x;
3
4     show_bytes((byte_pointer) &x, sizeof(short));
5     show_bytes((byte_pointer) &mx, sizeof(short));

```

当在大端法机器上运行时，这段代码的输出为 30 39 和 cf c7，指明 x 的十六进制表示为 0x3039，而 mx 的十六进制表示为 0xCFC7。将它们展开为二进制，我们得到 x 的位模式为[0011000000111001]，而 mx 的位模式为[110011111000111]。如图 2-15 所示，等式(2.3)对这两个位模式生成的值为 12 345 和-12 345。

权	12 345		-12 345		=		53 191	
	位	值	位	值	位	值	位	值
1	1	1	1	1	1	1	1	1
2	0	0	1	1	2	1	1	2
4	0	0	1	4	1	1	1	4
8	1	8	0	0	0	0	0	0
16	1	16	0	0	0	0	0	0
32	1	32	0	0	0	0	0	0
64	0	0	1	64	1	1	64	64
128	0	0	1	128	1	1	128	128
256	0	0	1	256	1	1	256	256
512	0	0	1	512	1	1	512	512
1 024	0	0	1	1 024	1	1	1 024	1 024
2 048	0	0	1	2 048	1	1	2 048	2 048
4 096	1	4096	0	0	0	0	0	0
8 192	1	8192	0	0	0	0	0	0
16 384	0	0	1	16 384	1	1	16 384	16 384
±32 768	0	0	1	-32 768	1	1	32 768	32 768
总计	12 345		-12 345		=		53 191	

图 2-15 12 345 和-12 345 的补码表示，以及 53 191 的无符号表示。注意后面两个数有相同的位表示

总之这两者的转换的图像解释可以理解为：

原理：无符号数转换为补码

对满足  $0 \leq u \leq UMax_w$  的  $u$  有：

$$U2T_w(u) = \begin{cases} u, & u \leq TMax_w \\ u - 2^w, & u > TMax_w \end{cases} \quad (2.7)$$

该原理证明如下：

推导：无符号数转换为补码

设  $\vec{u} = U2B_w(u)$ , 这个位向量也是  $U2T_w(u)$  的补码表示。公式(2.1)和公式(2.3)结合起来有

$$U2T_w(u) = -u_{w-1}2^{w-1} + u \quad (2.8)$$

在  $u$  的无符号表示中，对公式(2.7)的两种情况来说，位  $u_{w-1}$  决定了  $u$  是否大于  $TMax_w = 2^{w-1} - 1$ 。

图 2-18 说明了函数  $U2T$  的行为。对于小的数( $\leq TMax_w$ )，从无符号到有符号的转换将保留数字的原值。对于大的数( $> TMax_w$ )，数字将被转换为一个负数值。

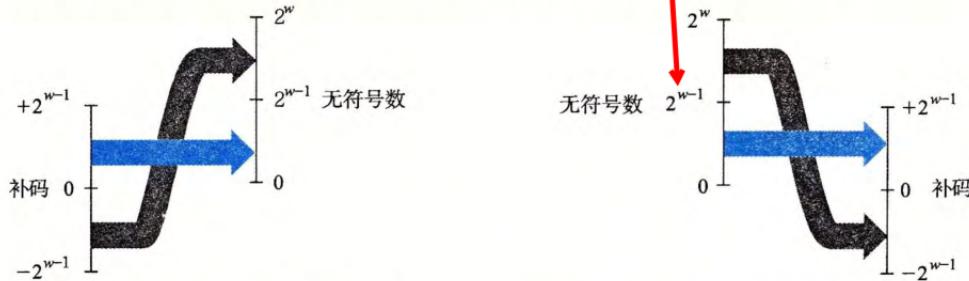


图 2-17 从补码到无符号数的转换。函数  $T2U$  将负数转换为大的正数

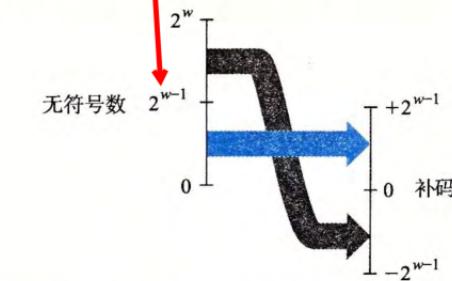


图 2-18 从无符号数到补码的转换。函数  $U2T$  把大于  $2^{w-1} - 1$  的数字转换为负值

## C语言中的无符号数

C语言中只有特别的给定u或U指定为无符号数，否则默认都是认为是一个有符号数，如 123U

C语言的运算中，**一个有符号数与一个无符号数进行运算，则系统会隐式的将有符号数变为无符号数，这就有可能导致错误的结果**，如  $-1 < 0U$ ，结果应该是1，但是有符号数-1自动转换为无符号数xxx，导致得出错误的结果0

## 网络旁注 DATA:TMIN C 语言中 TMin 的写法

在图 2-19 和练习题 2.21 中，我们很小心地将  $TMin_{32}$  写成 -2147483647-1。为什么不简单地写成 -2147483648 或者 0x80000000? 看一下 C 头文件 limits.h，注意到它们使用了跟我们写  $TMin_{32}$  和  $TMax_{32}$  类似的方法：

```
/* Minimum and maximum values a 'signed int' can hold. */
#define INT_MAX    2147483647
#define INT_MIN   (-INT_MAX - 1)
```

不幸的是，补码表示的不对称性和 C 语言的转换规则之间奇怪的交互，迫使我们用

这种不寻常的方式来写  $TMin_{32}$ 。虽然理解这个问题需要我们钻研 C 语言标准的一些比较隐晦的角落，但是它能够帮助我们充分领会整数数据类型和表示的一些细微之处。

## C 语言中，系统会对有符号数和无符号数之间的运算时会隐式的将有符号数变为无符号数

C 语言 Tmin 写法： **Tmin = -2147483647 - 1**

在图 2-19 和练习题 2.21 中，我们很小心地将  $TMin_{32}$  写成 -2147483647-1。为什么不简单地写成 -2147483648 或者 0x80000000? 看一下 C 头文件 limits.h，注意到它们使用了跟我们写  $TMin_{32}$  和  $TMax_{32}$  类似的方法：

```
/* Minimum and maximum values a 'signed int' can hold. */
#define INT_MAX    2147483647
#define INT_MIN   (-INT_MAX - 1)
```

## 位扩展

从一个较小的位数扩展到一个较大的位数是就会触发位扩展，目的是为了保持原始的值不变。例如从 16 位的数据类型扩展到 32 位的数据类型，即 short 扩展到 int

分为两种位扩展：

### 1. 无符号数用 0 扩展

如 16->32 多出来的两个字节全部用 0 来存放

### 2. 有符号数用符号位扩展

多出的两个字节用最高位 1 或者 0 来存放

**原理：无符号数的零扩展** 从16位到32位中多出的两个字节，用0来扩展

定义宽度为  $w$  的位向量  $\vec{u} = [u_{w-1}, u_{w-2}, \dots, u_0]$  和宽度为  $w'$  的位向量  $\vec{u}' = [0, \dots, 0, u_{w-1}, u_{w-2}, \dots, u_0]$ ，其中  $w' > w$ 。则  $B2U_w(\vec{u}) = B2U_{w'}(\vec{u}')$ 。

按照公式(2.1)，该原理可以看作是直接遵循了无符号数编码的定义。

要将一个补码数字转换为一个更大的数据类型，可以执行一个符号扩展(sign extension)，在表示中添加最高有效位的值，表示为如下原理。我们用蓝色标出符号位  $x_{w-1}$  来突出它在符号扩展中的角色。

**原理：补码数的符号扩展** 从16位到32位中多出的两个字节，用符号位来扩展，即1/0

定义宽度为  $w$  的位向量  $\vec{x} = [x_{w-1}, x_{w-2}, \dots, x_0]$  和宽度为  $w'$  的位向量  $\vec{x}' = [x_{w-1}, \dots, x_{w-1}, x_{w-2}, \dots, x_0]$ ，其中  $w' > w$ 。则  $B2T_w(\vec{x}) = B2T_{w'}(\vec{x}')$ 。

例如，考虑下面的代码：

```
1 short sx = -12345;           /* -12345 */
2 unsigned short usx = sx;     /* 53191 */
3 int x = sx;                 /* -12345 */
4 unsigned ux = usx;          /* 53191 */
5
6 printf("sx = %d:\t", sx);
7 show_bytes((byte_pointer) &sx, sizeof(short));
8 printf("usx = %u:\t", usx);
9 show_bytes((byte_pointer) &usx, sizeof(unsigned short));
10 printf("x = %d:\t", x);
11 show_bytes((byte_pointer) &x, sizeof(int));
12 printf("ux = %u:\t", ux);
13 show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

宽度更多，用符号扩展

在采用补码表示的 32 位大端法机器上运行这段代码时，打印出如下输出：

```
sx = -12345: cf c7 >216
usx = 53191: cf c7
x = -12345: ff ff cf c7
ux = 53191: 00 00 cf c7
```

但结果还是相同的，类似于拿不同刻度的测量工具去测量同一个物品，那么测量的数据肯定不同

我们看到，尽管 -12345 的补码表示和 53191 的无符号表示在 16 位字长时是相同的，但是在 32 位字长时却是不同的。特别地，-12345 的十六进制表示为 0xFFFFFCFC7，而 53191 的十六进制表示为 0x0000CFC7。前者使用的是符号扩展——最开头加了 16 位，都是最高有效位 1，表示为十六进制就是 0xFFFF。后者开头使用 16 个 0 来扩展，表示为十六进制就是 0x0000。

位扩展保持原始的值不变的例子：

-3。对它应用符号扩展，得到位向量 [1101]，表示的值  $-8+4+1=-3$ 。我们可以看到，对于  $w=4$ ，最高两位的组合值是  $-8+4=-4$ ，与  $w=3$  时符号位的值相同。类似地，位向量 [111] 和 [1111] 都表示值 -1。

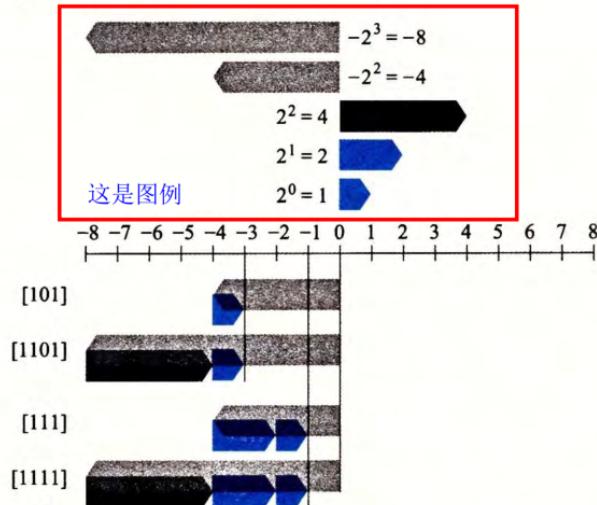


图 2-20 从  $w=3$  到  $w=4$  的符号扩展示例。对于  $w=4$ ，最高两位组合权重为  $-8+4=-4$ ，与  $w=3$  时的符号位的权重一样

## 转换类型中的隐式的位扩展：short 转成无符号数时会先自动扩展成 int 型

```

1 short sx = -12345;      /* -12345 */
2 unsigned uy = sx;        /* Mystery! */
3
4 printf("uy = %u:\t", uy);
5 show_bytes((byte_pointer) &uy, sizeof(unsigned));

```

在一台大端法机器上，这部分代码产生如下输出：

uy = 4294954951: ff ff cf c7

2: T2U

1: 扩展

这表明当把 short 转换成 unsigned 时，我们先要改变大小，之后再完成从有符号到无符号的转换。也就是说 `(unsigned) sx` 等价于 `(unsigned) (int) sx`，求值得到 4 294 954 951，而不等价于 `(unsigned) (unsigned short) sx`，后者求值得到 53 191。事

补码符号扩展值不变的证明：

**推导：**补码数值的符号扩展

令  $w' = w+k$ , 我们想要证明的是

$$B2T_{w+k}(\underbrace{[x_{w-1}, \dots, x_{w-1}]}_{k\text{次}}, x_{w-1}, x_{w-2}, \dots, x_0) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0])$$

下面的证明是对  $k$  进行归纳。也就是说，如果我们能够证明符号扩展一位保持了数值不变，那么符号扩展任意位都能保持这种属性。因此，证明的任务就变为了：

$$B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) = B2T_w([x_{w-1}, x_{w-2}, \dots, x_0])$$

用等式(2.3)展开左边的表达式，得到：

$$\begin{aligned} B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\ &= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}(2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x_i 2^i \\ &= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\ &= B2T_w([x_{w-1}, x_{w-2}, \dots, x_0]) \end{aligned}$$

我们使用的关键属性是  $2^w - 2^{w-1} = 2^{w-1}$ 。因此，加上一个权值为  $-2^w$  的位，和将一个权值为  $-2^{w-1}$  的位转换为一个权值为  $2^{w-1}$  的位，这两项运算的综合效果就会保持原始的数值。 ■

## 截断位

### 截断就是取模

既然有位扩展，那相应的也就有截断位或者说减少位。然而遗憾的是，**截断后的数值和原来的数值不一定相同**

有两种截断的方式，设**截断为k位**的结果，**截断都是从最高位开始截断的（保留低 k 位的意思）**：

1. 截断无符号数

**截断后的值 =  $x \bmod 2^k$**

2. 截断有符号数（补码）

**截断后的值 =  $U2T_k(x \bmod 2^k)$**

**练习题 2.24** 假设将一个 4 位数值(用十六进制数字 0~F 表示)截断到一个 3 位数值(用十六进制数字 0~7 表示)。填写下表, 根据那些位模式的无符号和补码解释, 说明这种截断对某些情况的结果。

十六进制		无符号		补码	
原始值	截断值	原始值	截断值	原始值	截断值
0	0	0		0	
2	2	2		2	
9	1	9		-7	
B	3	11		-5	
F	7	15		-1	

解释如何将等式(2.9)和等式(2.10)应用到这些示例上。

先按照无符号数处理, 后调用U2T

### 2.2.8 关于有符号数与无符号数的建议

## 本章建议

建议不要使用无符号数

## 整数运算

### 无符号数加法 (运算溢出)

若两个4位的数值相加, 其值最大会有30, 也就是需要用5位来表示, 这就导致了溢出现象。即w位的数字x、y时,  $x + y$  结果实际为:

- $x + y \quad —— x + y < 2^w$  (**正常情况**)
- $x + y - 2^w \quad —— 2^w < x + y < 2^{w+1}$  (**溢出**)

**原理: 无符号数加法**

对满足  $0 \leq x, y < 2^w$  的 x 和 y 有:

$$x +_w y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \leq x + y < 2^{w+1} \end{cases} \quad \begin{array}{l} \text{正常} \\ \text{溢出} \end{array} \quad (2.11)$$

例如, 4位的运算:  $9 + 12 = 21$ , 此时需要用5位来表示21这个结果。丢掉最高位的话, 结果为5, 就回到了正常的4位表示范围, 也就是  $9 + 12$  最终的值为5 (当然, 是在只有4位表示的情况下)

由此可得：没溢出时满足条件： $x + y \geq x$

## 无符号数的加法逆元

若  $\exists X$ , 使得  $(x - X + 2^w) \bmod 2^w = 0$ , 则就可以称  $X$  是  $x$  的一个加法逆元。

$(y - 2^w) < x$

练习题 2.27 写出一个具有如下原型的函数

`/* Determine whether arguments can be added without overflow */` 所以这里的模数加法  
int uadd\_ok(unsigned x, unsigned y);

如果参数  $x$  和  $y$  相加不会产生溢出, 这个函数就返回 1。

模数加法形成了一种数学结构, 称为阿贝尔群 (Abelian group), 这是以丹麦数学家 Niels Henrik Abel(1802~1829)的名字命名。也就是说, 它是可交换的(这就是为什么叫“abelian”的地方)和可结合的。它有一个单位元 0, 并且每个元素有一个加法逆元。让我们考虑  $w$  位的无符号数的集合, 执行加法运算  $+_w$ 。对于每个值  $x$ , 必然有某个值  $-_w x$  满足  $-_w x +_w x = 0$ 。该加法的逆操作可以表述如下:

原理: 无符号数求反

对满足  $0 \leq x < 2^w$  的任意  $x$ , 其  $w$  位的无符号逆元  $-_w x$  由下式给出:

$$-_w x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \quad (2.12)$$

该结果可以很容易地通过案例分析推导出来:

推导: 无符号数求反

当  $x=0$  时, 加法逆元显然是 0。对于  $x>0$ , 考虑值  $2^w - x$ 。我们观察到这个数字在  $0 < 2^w - x < 2^w$  范围之内, 并且  $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$ 。因此, 它就是  $x$  在  $+_w$  下的逆元。 ■

阿贝尔群 (Abelian Group), 又称交换群或加群, 是这样一类群:

它由自身的集合  $G$  和二元运算 \* 构成。它除了满足一般的群公理, 即运算的结合律、 $G$  有单位元 (类似单位矩阵一样, 和一个数作乘法时不改变结果)、所有  $G$  的元素都有逆元之外, 还满足交换律公理。

整数、实数的加法都构成了一个阿贝尔群

## 补码加法 (运算溢出)

无符号数就只有正溢出这个概念, 轮到补码这, 就可能出现正溢出、负溢出的概念了。

原理: 补码加法

对满足  $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$  的整数  $x$  和  $y$ , 有:

$$x +_w y = \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} \\ x + y + 2^w, & x + y < -2^{w-1} \end{cases} \quad \begin{array}{ll} \text{正溢出} & \text{结果为负值} \\ \text{正常} & \\ \text{负溢出} & \text{结果为正值} \end{array} \quad (2.13)$$

目的是为了让正溢出部分化为负值；负溢出部分化为正值

### 推导：补码加法

既然补码加法与无符号数加法有相同的位级表示，我们就可以按如下步骤表示 $x +^t_w y$ ：将其参数转换为无符号数，执行无符号数加法，再将结果转换为补码。

图 2-24 整数和补码加法之间的关系。  
当  $x + y$  小于  $-2^{w-1}$  时，产生负溢出。当它大于  $2^{w-1}$  时，产生正溢出

$$x +^t_w y \doteq U2T_w(T2U_w(x) +^u_w T2U_w(y)) \quad (2.14)$$

根据等式(2.6)，我们可以把  $T2U_w(x)$  写成  $x_{w-1}2^w + x$ ，把  $T2U_w(y)$  写成  $y_{w-1}2^w + y$ 。使用属性，即  $+^u_w$  是模  $2^w$  的加法，以及模数加法的属性，我们就能得到：

$$\begin{aligned} x +^t_w y &= U2T_w(T2U_w(x) +^u_w T2U_w(y)) \\ &= U2T_w[(x_{w-1}2^w + x + y_{w-1}2^w + y) \bmod 2^w] \\ &= U2T_w[(x + y) \bmod 2^w] \end{aligned}$$

案例如下图所示：

图 2-25 展示了一些 4 位补码加法的示例作为说明。每个示例的情况都被标号为对应于等式(2.13)的推导过程中的情况。注意  $2^4 = 16$ ，因此负溢出得到的结果比整数和大 16，而正溢出得到的结果比之小 16。我们包括了运算数和结果的位级表示。可以观察到，能够通过对运算数执行二进制加法并将结果截断到 4 位，从而得到结果。

$x$	$y$	$x + y$	$x +^t_4 y$	情况
-8 [1000]	-5 [1011]	-13 [10011]	3 [0011]	1 负溢出
-8 [1000]	-8 [1000]	-16 [10000]	0 [0000]	1 } 负溢出
-8 [1000]	5 [0101]	-3 [11101]	-3 [1101]	2 正溢出
2 [0010]	5 [0101]	7 [00111]	7 [0111]	3
5 [0101]	5 [0101]	10 [01010]	-6 [1010]	4 正溢出

图 2-25 补码加法示例。通过执行运算数的二进制加法并将结果截断到 4 位，可以获得 4 位补码和的位级表示

在代码中检测补码加法是否溢出的情况：

```
// 如果参数x和y相加不会产生溢出，这个函数返回1
int tadd_ok(int x, int y) {
    int s = x + y;
    if (x > 0 && y > 0 && s <= 0) {
        return 0; // 正数相加溢出
    } else if (x < 0 && y < 0 && s >= 0) {
        return 2; // 负数相加溢出
    }
    return 1;
}
```

## 补码的加法逆元（求- $x$ ）

也称之为**补码的非**

• 方法一：

### 2.3.3 补码的非

可以看到范围在  $TMin_w \leq x \leq TMax_w$  中的每个数字  $x$  都有  $-_w^t$  下的加法逆元，我们将  $-_w^t x$  表示如下。

原理：补码的非

因为补码的表示范围只有  $2^{w-1}$ ，达不到  $2^w$

对满足  $TMin_w \leq x \leq TMax_w$  的  $x$ ，其补码的非  $-_w^t x$  由下式给出

$$-_w^t x = \begin{cases} TMin_w, & x = TMin_w \\ -x, & x > TMin_w \end{cases} \quad (2.15)$$

也就是说，对  $w$  位的补码加法来说， $TMin_w$  是自己的加法的逆，而对其他任何数值  $x$  都有  $-x$  作为其加法的逆。

推导：补码的非

观察发现  $TMin_w + TMin_w = -2^{w-1} + (-2^{w-1}) = -2^w$ 。这将导致负溢出，因此  $TMin_w + _w^t TMin_w = -2^w + 2^w = 0$ 。对满足  $x > TMin_w$  的  $x$ ，数值  $-x$  可以表示为一个  $w$  位的补码，它们的和  $-x + x = 0$ 。 ■

例如：



练习题 2.33 我们可以用一个十六进制数字来表示长度  $w=4$  的位模式。根据这些数字的补码的解释，填写下表，确定所示数字的加法逆元。

$x$	$-_4^t x$		
十六进制	十进制	十进制	十六进制
0	0	0	0
5	5	-5	B
8	-8	-8	8
D	-3	3	3
F	-1	1	1

对于补码和无符号(练习题 2.28)非(negation)产生的位模式，你观察到什么？

• 方法二（位级表示）：

对补码的包括符号位的每一位取反，后加1。即  $\neg x == \neg x + 1 == \neg(x - 1)$

#### 网络旁注 DATA:TNEG 补码非的位级表示

计算一个位级表示的值的补码非有几种聪明的方法。这些技术很有用(例如当你在调试程序的时候遇到值 0xfffffffffa)，同时它们也能够让你更了解补码表示的本质。

执行位级补码非的第一种方法是对每一位求补，再对结果加1。在 C 语言中，我们可以说，对于任意整数值  $x$ ，计算表达式  $\neg x$  和  $\neg x + 1$  得到的结果完全一样。

下面是一些示例，字长为 4：

$\vec{x}$	$\neg\vec{x}$	$incr(\neg\vec{x})$
[0101]	5	[1010]
[0111]	7	[1000]
[1100]	-4	[0011]
[0000]	0	[1111]
[1000]	-8	[0111]

注意这里是求补码的非  
(加法逆元或相反数)  
时，是包括符号位都取  
反；而一个负数的原码  
转换成补码时，是不包  
括符号位进行取反的，  
注意区分！

$TMin = -8$

从前面的例子我们知道 0xf 的补是 0x0，而 0xa 的补是 0x5，因而 0xfffffffffa 是  
-6 的补码表示。

并非补码的意思，而是每一位求反

注意这里是求补码的非（加法逆元或相反数）时，是包括符号位都取反；而一个负数的原码转换成补码时，是不包括符号位进行取反的，注意区分！

### • 方法三（位级表示）：

找到补码 $x$ 中最右边的1，接下来，将这个1左边的包括符号位的所有位取反。例如  $[1010] = -6$ ，那么结果就是 $[0110] = 6$

计算一个数 $x$ 的补码非的第二种方法是建立在将位向量分为两部分的基础之上的。假设  $k$  是最右边的 1 的位置，因而  $x$  的位级表示形如  $[x_{w-1}, x_{w-2}, \dots, x_{k+1}, 1, 0, \dots, 0]$ 。（只要  $x \neq 0$  就能够找到这样的  $k$ 。）这个值的非写成二进制格式就是  $[\sim x_{w-1}, \sim x_{w-2}, \dots,$

$\sim x_{k+1}, 1, 0, \dots, 0]$ 。也就是，我们对位  $k$  左边的所有位取反。

我们用一些 4 位数字来说明这个方法，这里我们用斜体来突出最右边的模式  $1, 0, \dots, 0$ ：

$x$	$\sim x$
$[1100]$	-4
$[1000]$	-8
$[0101]$	5
$[0111]$	7
$[0100]$	4
$[1000]$	-8
$[1011]$	-5
$[1001]$	-7

## 补码的负数表示

$-\mathbf{x}$  等价于  $\sim \mathbf{x} + 1$  （取反后加一）等价于  $\sim (\mathbf{x} - 1)$  （减一后取反）

例如：

$$1. -9 == \sim 0x9 + 1 == \sim(0x9 - 1) == \sim 0x8 == -9$$

## 无符号乘法

因为无符号数的表示范围有限，所以两个无符号数的乘积如果超过了 $w$ 位，则超过 $w$ 位的部分需要截断处理，而截断处理就是直接取 $2^w$ 的模，即：

#### 2.3.4 无符号乘法

范围在  $0 \leq x, y \leq 2^w - 1$  内的整数  $x$  和  $y$  可以被表示为  $w$  位的无符号数，但是它们的乘积  $x \cdot y$  的取值范围为 0 到  $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$  之间。这可能需要  $2w$  位来表示。不过，C 语言中的无符号乘法被定义为产生  $w$  位的值，就是  $2w$  位的整数乘积的低  $w$  位表示的值。我们将这个值表示为  $x * {}_w^u y$ 。

将一个无符号数截断为  $w$  位等价于计算该值模  $2^w$ ，得到：

原理：无符号数乘法

对满足  $0 \leq x, y \leq UMax_w$  的  $x$  和  $y$  有：

$$x * {}_w^u y = (x \cdot y) \bmod 2^w \quad (2.16)$$

注意， $w$  位的  $x$  和  $y$  相乘后，得到的是一个  $2w$  位的表示，那么此时可能会超出所规定的位的大小，那么就需要用到截断的知识来处理。

## 补码乘法

补码乘法就是将无符号乘法的结果转换成补码形式即可，即运用一次U2T，如下所示：

#### 2.3.5 补码乘法

范围在  $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$  内的整数  $x$  和  $y$  可以被表示为  $w$  位的补码数字，但是它们的乘积  $x \cdot y$  的取值范围为  $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$  到  $-2^{w-1} \cdot -2^{w-1} = -2^{2w-2}$  之间。要想用补码来表示这个乘积，可能需要  $2w$  位。然而，C 语言中的有符号乘法是通过将  $2w$  位的乘积截断为  $w$  位来实现的。我们将这个数值表示为  $x * {}_w^t y$ 。将一个补码数截断为  $w$  位相当于先计算该值模  $2^w$ ，再把无符号数转换为补码，得到：

原理：补码乘法

对满足  $TMin_w \leq x, y \leq TMax_w$  的  $x$  和  $y$  有：

$$x * {}_w^t y = U2T_w((x \cdot y) \bmod 2^w) \quad (2.17)$$

我们认为对于无符号和补码乘法来说，乘法运算的位级表示都是一样的，并用如下原理说明：

原理：无符号和补码乘法的位级等价性

给定长度为  $w$  的位向量  $\vec{x}$  和  $\vec{y}$ ，用补码形式的位向量表示来定义整数  $x$  和  $y$ ： $x = B2T_w(\vec{x})$ ， $y = B2T_w(\vec{y})$ 。用无符号形式的位向量表示来定义非负整数  $x'$  和  $y'$ ： $x' = B2U_w(\vec{x})$ ， $y' = B2U_w(\vec{y})$ 。则

$$T2B_w(x * {}_w^t y) = U2B_w(x' * {}_w^u y')$$

作为说明，图 2-27 给出了不同 3 位数字的乘法结果。对于每一对位级运算数，我们执行无符号和补码乘法，得到 6 位的乘积，然后再把这些乘积截断到 3 位。无符号的截断后的乘积总是等于  $x \cdot y \bmod 8$ 。虽然无符号和补码两种乘法乘积的 6 位表示不同，但是截断后的乘积的位级表示都相同。但并不意味着结果相同，因为无符号数不用符号位会多一位，数就更大

无符号和补码乘法，截断后的每一位数值相同，但是各自表示的值是不同的，因为补码第一位是 0 或负值表示。

学到这里可以发现，所有补码的运算：加法、乘法，都是在无符号数运算的基础上，最后把值转换为补码形式，即 U2T。但有时候如果给的值是补码形式，那么还要转换为原码进行运算之后才是转换

## 乘常数

由于乘法指令很慢，往往需要10+个时钟周期，而加法、减法、位移只需要1个时钟周期，这也就导致了编译器为了更好的优化，而采用位移的方式来表示乘法。如乘2的幂，就只需要左移k位即可

由于无符号数和补码的运算都满足结合律、交换律和分配律，因此编译器就能用位移的方式来表达一个乘法运算，如：

$$(x \ll 3) - x == 7*x$$

由于整数乘法比移位和加法的代价要大得多，许多C语言编译器试图以移位、加法和减法的组合来消除很多整数乘以常数的情况。例如，假设一个程序包含表达式  $x * 14$  利用  $14 = 2^3 + 2^2 + 2^1$ ，编译器会将乘法重写为  $(x \ll 3) + (x \ll 2) + (x \ll 1)$ ，将一个乘法替换为三个移位和两个加法。无论x是无符号的还是补码，甚至当乘法会导致溢出时，两个计算都会得到一样的结果。（根据整数运算的属性可以证明这一点。）更好的是，编译器还可以利用属性  $14 = 2^4 - 2^1$ ，将乘法重写为  $(x \ll 4) - (x \ll 1)$ ，这时只需要两个移位和一个减法。

乘法：可以用位移 + 加减法表示任意常数，如↑的重写一样

## 除2的幂

遗憾的是，除法并不能像乘法一般地表示任意的常数；对于2的幂确实可以明确表示，但对于不是2的幂的数只能是近似的表示

除法比乘法的运算速度会更慢，需要30个时钟周期左右

除法同样也是用右移来表示，这样的优化才能加速计算速度

当  $x \geq 0, y > 0$  时， $x / y$  的结果向下取整；当  $x < 0, y > 0$  时， $x / y$  的结果，向上取整

- 向下取整（舍入）的情况：

适用于无符号数或者正值的有符号数

原理：除以 2 的幂的无符号除法

C 变量 x 和 k 有无符号数值  $x$  和  $k$ ，且  $0 \leq k < w$ ，则 C 表达式  $x \gg k$  产生数值  $\lfloor x/2^k \rfloor$ 。

例如，图 2-28 给出了在 12340 的 16 位表示上执行逻辑右移的结果，以及对它执行除以 1、2、16 和 256 的结果。从左端移入的 0 以斜体表示。我们还给出了用真正的运算做除法得到的结果。这些示例说明，移位总是舍入到零的结果，这一点与整数除法的规则一样。

k	$\gg k$ (二进制)	十进制	$12340/2^k$
0	0011000000110100	12340	12340.0
1	0001100000011010	6170	6170.0
4	0000001100000011	771	771.25
8	0000000001100000	48	48.203125

图 2-28 无符号数除以 2 的幂(这个例子说明了执行一个逻辑右移  $k$  位与除以  $2^k$  再舍入到零有一样的效果)

- 向上取整（舍入）的情况，通过添加偏置值来修复无法向上取整（错误的向下取整了）的错误：

适用于负值的有符号数

我们可以通过在移位之前“偏置(biasing)”这个值，来修正这种不合适的舍入。

原理：除以 2 的幂的补码除法，向上舍入

C 变量 x 和 k 分别有补码值  $x$  和无符号数值  $k$ ，且  $0 \leq k < w$ ，则当执行算术移位时，C 表达式  $(x + (1 \ll k) - 1) \gg k$  产生数值  $\lfloor x/2^k \rfloor$ 。

图 2-30 说明在执行算术右移之前加上一个适当的偏置量是如何导致结果正确舍入的。在第 3 列，我们给出了 -12340 加上偏置值之后的结果，低  $k$  位(那些会向右移出的位)以斜体表示。我们可以看到，低  $k$  位左边的位可能会加 1，也可能不会加 1。对于不需要舍入的情况( $k=1$ )，加上偏量只影响那些被移掉的位。对于需要舍入的情况，加上偏量导致较高的位加 1，所以结果会向零舍入。

k	偏量	-12340 + 偏量	$\gg k$ (二进制)	十进制	$-12340/2^k$
0	0	110011111001100	110011111001100	-12340	-12340.0
1	1	110011111001101	111001111100110	-6170	-6170.0
4	15	110011111011011	1111110011111101	-771	-771.25 正确的向
8	255	1101000011001011	1111111110100000	-48	-48.203125

图 2-30 补码除以 2 的幂(右移之前加上一个偏量，结果就向零舍入了)

偏置技术利用如下属性：对于整数  $x$  和  $y$  ( $y > 0$ )， $\lceil x/y \rceil = \lfloor (x+y-1)/y \rfloor$ 。例如，当  $x = -30$  和  $y = 4$ ，我们有  $x+y-1 = -27$ ，而  $\lceil -30/4 \rceil = -7 = \lfloor -27/4 \rfloor$ 。当  $x = -32$  和  $y = 4$  时，我们有  $x+y-1 = -29$ ，而  $\lceil -32/4 \rceil = -8 = \lfloor -29/4 \rfloor$ 。

推导：除以 2 的幂的补码除法，向上舍入

这里的  $y-1$  就是  $k$  位无符号数的最大值  $2^k - 1$ ，即  $(1 \ll k) - 1$

即计算机通过偏置和向下取整的方式实现了负值补码除法中的向上取整。

在 C 语言中，用来表示这两种情况的表达式表示：

$(x < 0 ? x + (1 \ll k) - 1 : x) \gg k$

# 浮点数

当一个数字不能被准确地表示为浮点格式时，就必须向上调整或者向下调整

## 二进制小数

显然，二进制小数中的小数点向左移动一位就表示该数除以2；向右移动一位就表示该数乘以2

二进制小数和十进制小数的差异

- 小数的**二进制表示法**呢可以很容易的表达出  $x * 2^y$  的数，但其他的值只能够近似地表示
- 小数的**十进制表示法**呢可以表示一些特定的数，如  $1/5$  可以用  $0.2$  来表示，但同样的，对于  $1/3$  这类值，也是同样只能够近似的表示

所以，这就引出了浮点数的精度问题，如果一个浮点数类型的字长越长，尾数位就越多，能表示的有效数位数就越多，那么这个近似值就能够越接近于其真值。

### IEEE阶码位数为什么是8和11的扩展（属于下一小节，放在这里解释）

单精度的**8位阶码** 和双精度的**11位阶码** 是 IEEE 754 标准制定者基于**总位数限制**，对**数值范围**（由阶码位决定）和**数值精度**（由尾数位决定）进行**精心权衡**后得出的最优分配方案：

1. **单精度 (8位阶码, 23位尾数)**: 在32位约束下，提供了**足够宽泛的数值范围** ( $10^{\pm38}$ ) 和**可接受的计算精度** (约7位有效数字)，满足大多数通用计算需求。
2. **双精度 (11位阶码, 52位尾数)**: 在64位约束下，提供了**极其巨大的数值范围** ( $10^{\pm308}$ ) 和**非常高的计算精度** (约15-16位有效数字)，满足了科学计算、工程仿真、金融等对精度要求极高的领域的需求。11位阶码提供的范围已经绰绰有余，将更多位分配给尾数以追求极致精度是更明智的选择。

#### • **单精度 (32位):**

- $1 \text{ (符号)} + k \text{ (阶码)} + M \text{ (尾数)} = 32 \Rightarrow k + M = 31$
- 如果  $k=7$ ：指数范围太小（大约 -62 到 +63），很多实际应用（物理、天文、工程）会遇到上溢/下溢问题。尾数  $M=24$ ，精度提升有限（约 7.2 位十进制）。
- 如果  $k=9$ ：指数范围更大（大约 -254 到 +255），但尾数  $M=22$ ，精度显著下降（约 6.6 位十进制），这对于需要一定精度的计算来说损失太大。

- **k=8** 是最优解: 指数范围  $-126 \sim +127$  (约  $10^{\pm 38}$  数量级) 满足了绝大多数应用对范围的需求, 同时尾数  $M=23$  提供的精度 (约 7.2 位十进制) 也是可接受的。

### • 双精度 (64位):

- $1 \text{ (符号)} + k \text{ (阶码)} + M \text{ (尾数)} = 64 \Rightarrow k + M = 63$
- 如果  $k=10$ : 指数范围约 -510 到 +511 ( $10^{\pm 154}$  数量级)。尾数  $M=53$ 。
- 如果  $k=12$ : 指数范围约 -2046 到 +2047 ( $10^{\pm 616}$  数量级), 极其巨大, 但尾数  $M=51$ , 精度下降明显 (损失约 1 位二进制精度)。
- **k=11** 是最优解: 指数范围  $-1022 \sim +1023$  (约  $10^{\pm 308}$  数量级) 已经非常巨大, 足以覆盖天文数字和微观粒子等极端场景。同时尾数  $M=52$  提供的超高精度 (约 15-16 位十进制) 是科学计算、金融、高精度模拟等领域的核心需求。牺牲一点几乎用不到的额外指数范围来换取至关重要的精度提升是明智的选择。

## IEEE浮点表示

电气和电子工程师协会 (IEEE, 读做 "eye-triple-ee") 是一个包括所有电子和计算机技术的专业团体。

浮点数使用 IEEE 标准 754 编码

IEEE 浮点标准用

$$V = (-1)^s * M * 2^E$$

的形式来表示一个数:

IEEE 浮点标准用  $V=(-1)^s \times M \times 2^E$  的形式来表示一个数：

- 符号(sign)  $s$  决定这数是负数( $s=1$ )还是正数( $s=0$ )，而对于数值 0 的符号位解释作为特殊情况处理。
- 尾数(significand)  $M$  是一个二进制小数，它的范围是  $1 \sim 2 - \epsilon$ ，或者是  $0 \sim 1 - \epsilon$ 。
- 阶码(exponent)  $E$  的作用是对浮点数加权，这个权重是 2 的  $E$  次幂(可能是负数)。将浮点数的位表示划分为三个字段，分别对这些值进行编码：
  - 一个单独的符号位，直接编码符号  $s$ 。
  - $k$  位的阶码字段  $\text{exp} = e_{k-1} \dots e_1 e_0$  编码阶码  $E$ 。
  - $n$  位小数字段  $\text{frac} = f_{n-1} \dots f_1 f_0$  编码尾数  $M$ ，但是编码出来的值也依赖于阶码字段的值是否等于 0。

图 2-32 给出了将这三个字段装进字中两种最常见的格式。在单精度浮点格式(C 语言中的 float)中， $s$ 、 $\text{exp}$  和  $\text{frac}$  字段分别为 1 位、 $k=8$  位和  $n=23$  位，得到一个 32 位的表示。在双精度浮点格式(C 语言中的 double)中， $s$ 、 $\text{exp}$  和  $\text{frac}$  字段分别为 1 位、 $k=11$  位和  $n=52$  位，得到一个 64 位的表示。

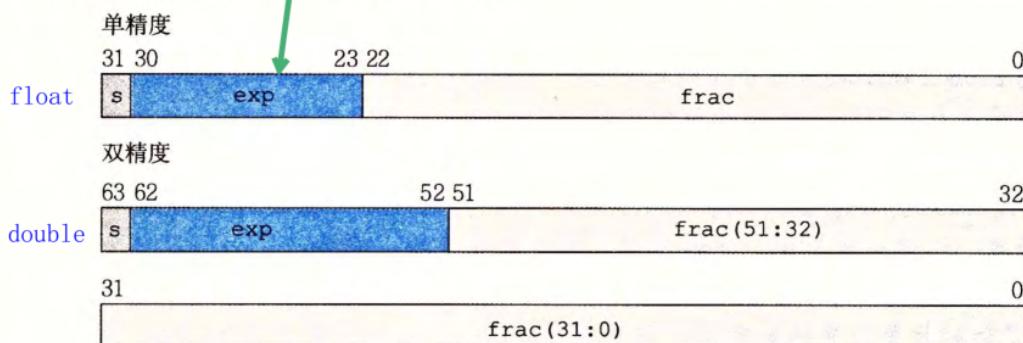


图 2-32 标准浮点格式(浮点数由 3 个字段表示。两种最常见的格式是它们被封装到 32 位(单精度)和 64 位(双精度)的字中)

其中单精度和双精度的表示还有所不同：

精度	符号位(s)	阶码位(exp 或 k 位 或 无符号数 e)	尾数位(frac 或 M)
单精度 (float) 32位	1	8 (bias = 127)	23
双精度 (double) 64位	1	11 (bias = 1023)	52

当阶码值不是边界值时，表示的是一个二进制位表示的无符号正数  $e$ ，用于计算规格化的阶码  $E$

根据阶码位，也就是exp位，可以分为下面三种单精度的表示情况（第三种有两个变体形式）：

给定位表示，根据  $\exp$  的值，被编码的值可以分成三种不同的情况（最后一种情况有两个变种）。图 2-33 说明了对单精度格式的情况。

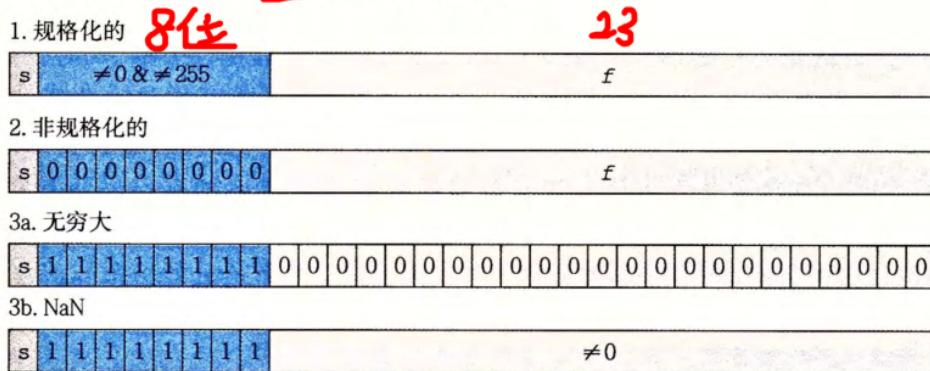


图 2-33 单精度浮点数值的分类(阶码的值决定了这个数是规格化的、非规格化的或特殊值)

### 1. 规格化（一般化）的情况：

当  $\exp$  或  $e$  的位不全为 0 或 1 时，就属于该情况。此时阶码  $E = e - \text{Bias}$ 。其中  $e$  就是上面蓝色部分的二进制位表示的无符号正数，而  $\text{Bias}$  是一个固定的值，其值根据阶码的位数来决定，如单精度时， $k = 8$ ，此时  $2^7 - 1 = 127$ （双精度是 1023），由此可得：

- $E = e - 127$  E 范围是 -126 到 +127（单精度的情况）
- $E = e - 1023$  E 范围是 -1022 到 +1023（双精度的情况）

因为  $e$  不全为 0 或 1，此时  $e$  的范围就只有 1-254，根据该  $e$  的范围才得出上述的  $E$  的范围

小数字段  $\text{frac}$  用来描述小数值  $f$ ， $0 \leq f < 1$ ，也就是小数点放在  $f$  最高有效位的左边。将尾数  $M$  定义为  $M = 1 + f$ 。该定义方式，也叫隐含的以 1 开头的表示，可得  $1 \leq M < 2$ 。但由于它的第一位总是 1，所以我们一般不显示它，也就是最终结果  $V = (-0.xxx * 2^E)$

$f$  可以这么算：

用小数字段  $\text{frac}$  所表示的无符号值 / 2<sup>该  $\text{frac}$  的位数</sup>

### 2. 非规格化（不一般）的情况：

当  $\exp$  或  $e$  的位全为 0 时，此时阶码  $E = 1 - \text{Bias} = 1 - 127 = -126$ （单精度情况下），而  $M = f$

非规格化数有两个用途：

- 首先，它们提供了一种表示数值 0 的方法，规格化因为  $M$  会加 1，所以最小值是 1，没法表示 0。
- 还可以用来表示非常接近于 0.0 的数。他们提供了一种属性，称为逐渐溢出，其中，数值分部均匀地接近于 0.0

### 3. 特殊情况：

- 当 **exp** 或 **e** 的位全为 1，且小数域全为 0 时，表示的无穷，正负无穷由符号位决定。正如 **xx / +- 0** 一般得到的是正负无穷
- 当小数域为非零时，结果值被称为 "**NaN**"，即“不是一个数 (Not a Number)" 的缩写。一些运算的结果不能是实数或无穷

比如当计算  $\sqrt{-1}$  或  $\infty - \infty$  时。在某些应用中，表示未初始化的数据时，它们也很有用处。

## IEEE浮点表示案例

可以结合上一讲的公式食用

图 2-35 展示了假定的 8 位浮点格式的示例，其中有  $k=4$  的阶码位和  $n=3$  的小数位。偏置量是  $2^{4-1}-1=7$ 。图被分成了三个区域，来描述三类数字。不同的列给出了阶码字段是如何编码阶码  $E$  的，小数字段是如何编码尾数  $M$  的，以及它们一起是如何形成要表示的值  $V=2^E \times M$  的。从 0 自身开始，最靠近 0 的是非规格化数。这种格式的非规格化数的  $E=1-7=-6$ ，得到权  $2^E=\frac{1}{64}$ 。小数  $f$  的值的范围是  $0, \frac{1}{8}, \dots, \frac{7}{8}$ ，从而得到数  $V$  的范围是  $0 \sim \frac{1}{64} \times \frac{7}{8} = \frac{7}{512}$ 。

描述	位表示 s k=4 n=3	指数			小数		值		
		e	E	$2^E$	f	M	$2^E \times M$	V 浮点数的值	十进制
0 最小的非规格化数	0 0000 000	0	-6	$\frac{1}{64}$	0	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
	0 0000 001	0	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	-6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{2}{256}$	0.003906
	0 0000 011	0	-6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
最大的非规格化数	0 0000 111	0	-6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
	最小的规格化数			$E=e-7(\text{bias})$	$M=f+1$				
	0 0001 000	1	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
1	0 0110 110	6	-1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	-1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
最大的规格化数	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
无穷大	0 1111 000	—	—	—	—	—	—	$\infty$	—

图 2-35 8 位浮点格式的非负值示例 ( $k=4$  的阶码位的和  $n=3$  的小数位。偏置量是 7)

这种形式的最小规格化数同样有  $E=1-7=-6$ ，并且小数取值范围也为  $0, \frac{1}{8}, \dots, \frac{7}{8}$ 。然而，尾数在范围  $1+0=1$  和  $1+\frac{7}{8}=\frac{15}{8}$  之间，得出数  $V$  在范围  $\frac{8}{512}=\frac{1}{64}$  和  $\frac{15}{512}$  之间。

可以发现十进制整数部分，是按照递增顺序排序的，这也是 IEEE 格式如此设计的核心

可以观察到最大非规格化数  $\frac{7}{512}$  和最小规格化数  $\frac{8}{512}$  之间的平滑转变。这种平滑性归功于我们对非规格化数的  $E$  的定义。通过将  $E$  定义为  $1 - Bias$ , 而不是  $-Bias$ , 我们可以补偿非规格化数的尾数没有隐含的开头的 1。

当增大阶码时, 我们成功地得到更大的规格化值, 通过 1.0 后得到最大的规格化数。这个数具有阶码  $E=7$ , 得到一个权  $2^E = 128$ 。小数等于  $\frac{7}{8}$  得到尾数  $M = \frac{15}{8}$ 。因此, 数值是  $V = 240$ 。超出这个值就会溢出到  $+\infty$ 。

这种表示具有一个有趣的属性, 假如我们将图 2-35 中的值的位表达式解释为无符号整数, 它们就是按升序排列的, 就像它们表示的浮点数一样。这不是偶然的——IEEE 格式如此设计就是为了浮点数能够使用整数排序函数来进行排序。当处理负数时, 有一个小的难点, 因为它们有开头的 1, 并且它们是按照降序出现的, 但是不需要浮点运算来进行比较也能解决这个问题(参见家庭作业 2.84)。

## 舍入

浮点运算只能近似地表示实数运算, 那么找到一个最近接的值就显得尤为重要。IEEE 定义了四种不同的舍入方式, 默认的舍入方式是向偶舍入 (round-to-even), 也叫做向最接近的值舍入 (round-to-nearest) :

值的舍入效果。向偶数舍入方式采用的方法是: 它将数字向上或者向下舍入, 使得结果的最低有效数字是偶数。因此, 这种方法将 1.5 美元和 2.5 美元都舍入成 2 美元。

方 式	1.40	1.60	1.50	2.50	-1.50
向偶数舍入	1	2	2	2	-2
向零舍入	1	1	1	2	-1
向下舍入	1	1	1	2	-2
向上舍入	2	2	2	3	-1

图 2-37 以美元舍入为例说明舍入方式(第一种方法是舍入到一个最接近的值, 而其他三种方法向上或向下限定结果, 单位为美元)

像 python 中的内置 round 库, 就是用的向偶数舍入的方式, 这一点要注意了

向偶舍入就是为了避免全部都是向上舍入导致的平均值增大和全部都是向下舍入导致的平均值减小, 就是为了避免这种统计偏差而产生的, 50% 的时间是向上舍入, 另一半则是向下舍入。

- 二进制小数里使用向偶数舍入法:

1.24, 因为 4 是偶数。

如果是1就+1向上舍入, 如果是0就向下舍入

相似地, 向偶数舍入法能够运用在二进制小数上。我们将最低有效位的值 0 认为是偶数, 值 1 认为是奇数。一般来说, 只有对形如  $XX\cdots X.YY\cdots Y100\cdots$  的二进制位模式的数, 这种舍入方式才有效, 其中  $X$  和  $Y$  表示任意位值, 最右边的  $Y$  是要被舍入的位置。只有这种位模式表示在两个可能的结果正中间的值。例如, 考虑舍入值到最近的四分之一的问题(也就是二进制小数点右边 2 位)。我们将  $10.00011_2 \left(2 \frac{3}{32}\right)$  向下舍入到  $10.00_2 (2)$ ,

$10.00110_2 \left(2 \frac{3}{16}\right)$  向上舍入到  $10.01_2 \left(2 \frac{1}{4}\right)$ , 因为这些值不是两个可能值的正中间值。我们将根据第三个1决定要向偶舍入, 小数第二位加1即可

$10.\boxed{111}00_2 \left(2 \frac{7}{8}\right)$  向上舍入成  $11.00_2 (3)$ , 而  $10.10100_2 \left(2 \frac{5}{8}\right)$  向下舍入成  $10.10_2 \left(2 \frac{1}{2}\right)$ , 因为这些值是两个可能值的中间值, 并且我们倾向于使最低有效位为零。

**注意: 二进制 $10.011$ 保留一位小数的话, 是舍入成 $10.1$ , 细节如下:**

1. **舍入位 = 0** (全为0) → 直接舍去 (不进位)。
2. **舍入位 > 100...0** (即最高位为1且后续有非0位) → 进位 (加1到保留的最后一 位)。
3. **舍入位 = 100...0** (恰好是0.5 ULP) → 向最近的偶数舍入:
  - 保留的最后一位为 **0 (偶数)** → 舍去 (不进位)。
  - 保留的最后一位为 **1 (奇数)** → 进位 (加1)。

## 舍入案例

### 练习题 2.52

考虑下列基于 IEEE 浮点格式的 7 位浮点表示。两个格式都没有符号位——它们只能表示非负的数字。

#### 1. 格式 A

- 有  $k=3$  个阶码位。阶码的偏置值是 3。
- 有  $n=4$  个小数位。

#### 2. 格式 B

- 有  $k=4$  个阶码位。阶码的偏置值是 7。
- 有  $n=3$  个小数位。

下面给出了一些格式 A 表示的位模式，你的任务是将它们转换成格式 B 中最接近的值。如果需要，请使用舍入到偶数的舍入原则。另外，给出由格式 A 和格式 B 表示的位模式对应的数字的值。给出整数（例如 17）或者小数（例如  $17/64$ ）。

格式A		格式B	
位	值	位	值
011 0000	1	0111 000	1
✓ 101 1110			
✓ 010 1001			
110 1111			
000 0001			

### 转换结果：

格式A位模式	格式A值	格式B位模式	格式B值	计算说明
011 0000	1	0111 000	1	[示例] $1 = 1.0 \times 2^0 \rightarrow E=0, e=7 (0111), f=000$
101 1110	$15/2$	1001 111	$15/2$	$7.5 = 1.111 \times 2^2 \rightarrow E=2, e=9 (1001), f=111$
010 1001	$25/32$	0110 100	$3/4$	$0.78125 \rightarrow 1.1001 \times 2^{-1} \rightarrow$ 舍入到 $1.100$ ( $0.75$ )
110 1111	$31/2$	1011 000	16	$15.5 \rightarrow 1.1111 \times 2^3 \rightarrow$ 舍入进位得 $16.0$

000 0001 1/64 0001 000 1/64

 $0.015625 = 1.0 \times 2^{-6} \rightarrow E=-6, e=1 (000 1)$ 

## 详细计算过程

**格式A** 101 1110 → **格式B**

- **阶码** 101 = 5  
→  $E = 5 - 3 = 2$
- **小数** 1110 =  $\frac{14}{16} = \frac{7}{8}$   
→  $M = 1 + \frac{7}{8} = \frac{15}{8}$
- **值**  $V = \frac{15}{8} \times 2^2 = \frac{15}{2} = 7.5$
- **格式B表示:**

- $7.5 = 1.875 \times 2^2 = 1.111_2 \times 2^2$
- 阶码  $e = 2 + 7 = 9 \rightarrow$  1001
- 小数 111 (保留3位)
- → 1001 111 (值仍为7.5)

**格式A** 010 1001 → **格式B**

- **阶码** 010 = 2  
→  $E = 2 - 3 = -1$
- **小数** 1001 =  $\frac{9}{16}$   
→  $M = 1 + \frac{9}{16} = \frac{25}{16}$
- **值**  $V = \frac{25}{16} \times 2^{-1} = \frac{25}{32} = 0.78125$
- **格式B表示:**

- $0.78125 = 1.5625 \times 2^{-1} = 1.1001_2 \times 2^{-1}$
- **舍入:** 1.1001 → 第4位=1 → 向偶数舍入 → 1.100 (末尾0为偶数)
- 阶码  $e = -1 + 7 = 6 \rightarrow$  0110
- 小数 100
- → 0110 100 (值  $1.5 \times 2^{-1} = 0.75 = \frac{3}{4}$ )

**格式A** 110 1111 → **格式B**

- 阶码  $\boxed{110} = 6$   
 $\rightarrow E = 6 - 3 = 3$
- 小数  $\boxed{1111} = \frac{15}{16}$   
 $\rightarrow M = 1 + \frac{15}{16} = \frac{31}{16}$
- 值  $V = \frac{31}{16} \times 2^3 = \frac{31}{2} = 15.5$

• 格式B表示:

- $15.5 = 1.9375 \times 2^3 = 1.1111_2 \times 2^3$

为什么  $15.5 = 1.9375 \times 2^3$ ?

**步骤 1: 整数部分转换**

- $15_{10} = 1111_2$
- $0.5_{10} = 0.1_2$
- 合并:  $15.5_{10} = 1111.1_2$

**步骤 2: 科学计数法规范化**

- 移动小数点, 使整数部分为  $\boxed{1}$ :  
 $1111.12 = 1.11112 \times 2^3$   
 $1111.12 = 1.11112 \times 2^3$   
(左移 3 位, 对应指数 +3)

**验证:**

$$1.11112 \times 2^3 = 1111.12 = 15.5$$

- 舍入:  $\boxed{1.1111} \rightarrow$  第4位=1 → 进位  $\rightarrow \boxed{10.0000} \rightarrow$  规格化为  $1.0000 \times 2^4$
- 阶码  $e = 4 + 7 = 11 \rightarrow \boxed{1011}$
- 小数  $\boxed{000}$
- $\rightarrow \boxed{1011 000}$  (值 16)

**格式A  $\boxed{000 0001} \rightarrow$  格式B**

• 阶码全0 → 非规格化:

$$E = 1 - 3 = -2$$

- 小数  $\boxed{0001} = \frac{1}{16}$   
 $\rightarrow M = f = \frac{1}{16}$
- 值  $V = \frac{1}{16} \times 2^{-2} = \frac{1}{64}$
- 格式B表示:

- $\frac{1}{64} = 1.0 \times 2^{-6}$
- 阶码  $e = -6 + 7 = 1 \rightarrow 0001$
- 小数  $000$
- $\rightarrow 0001\ 000$  (值仍为  $\frac{1}{64}$ )

## 关键点

- **舍入规则**: 向偶数舍入 (Round-to-even), 中间值时选择末尾为偶数的表示。

例如  $1.1001$  舍入到3位小数  $\rightarrow 1.100$  (因末尾0为偶数)。

## 浮点运算

1. 整数加法和**实数的加法（不等同于浮点数）都构成了一个阿贝尔群**, 即无符号数和补码运算满足结合律、交换律和分配率
2. **浮点数加法运算是可交换、具有单调性, 不可结合、分配的**

IEEE 标准指定了一个简单的规则, 来确定诸如加法和乘法这样的算术运算的结果。把浮点值  $x$  和  $y$  看成实数, 而某个运算  $\odot$  定义在实数上, 计算将产生  $\text{Round}(x \odot y)$ , 这是对实际运算的精确结果进行舍入后的结果。在实际中, 浮点单元的设计者使用一些聪明的小技巧来避免执行这种精确的计算, 因为计算只要精确到能够保证得到一个正确的舍入结果就可以了。当参数中有一个是特殊值(如  $-0$ 、 $-\infty$  或  $\text{NaN}$ )时, IEEE 标准定义了一些使之更合理的规则。例如, 定义  $1/-0$  将产生  $-\infty$ , 而定义  $1/+0$  会产生  $+\infty$ 。[在数学上才能](#)

IEEE 标准中指定浮点运算行为方法的一个优势在于, 它可以独立于任何具体的硬件或者软件实现。因此, 我们可以检查它的抽象数学属性, 而不必考虑它实际上是如何实现的。

前面我们看到了**整数(包括无符号和补码)加法形成了阿贝尔群**。实数上的加法也形成了**阿贝尔群**, 但是我们必须考虑舍入对这些属性的影响。我们将  $x+y$  定义为  $\text{Round}(x+y)$ 。这个运算的定义针对  $x$  和  $y$  的所有取值, 但是虽然  $x$  和  $y$  都是**实数**, 由于溢出, 该运算可能得到无穷值。对于所有  $x$  和  $y$  的值, 这个运算是**可交换的**, 也就是说  $x+y=y+x$ 。另一方面, 这个运算是**不可结合的**。例如, 使用单精度浮点, 表达式  $(3.14+1e10)-1e10$  求值得到  $0.0$ ——因为舍入, 值  $3.14$  会丢失。另一方面, 表达式  $3.14+(1e10-1e10)$  得出值  $3.14$ 。作为阿贝尔群, 大多数值在浮点加法下都有逆元, 也就是说  $x+y=0$ 。无穷( $\infty$ )和  $\text{NaN}$  是例外情况, 因为对于任何  $x$ , 都有  $\text{NaN}+x=\text{NaN}$ 。[符合我学过的数据库用NULL的运算](#)

浮点加法不具有结合性, 这是缺少的最重要的群属性。对于科学计算程序员和编译器编写者来说, 这具有重要的含义。例如, 假设一个编译器给定了如下代码片段:

```
x = a + b + c;
y = b + c + d;
```

**编译器可能试图通过产生下列代码来省去一个浮点加法**

```
t = b + c;
x = a + t;
y = t + d;
```

扩展: 各地方的空值表示

	空值	空字符
数据库	NULL	""
Python	None	"" 或 ''
pandas	NaN或NaT	"
Excel	空	""

### 3. 浮点数乘法运算是可交换、具有单调性，不可结合、分配的

然而，对于  $x$  来说，这个计算可能会产生与原始值不同的值，因为它使用了加法运算的不同结合方式。在大多数应用中，这种差异小得无关紧要。不幸的是，编译器无法知道在效率和忠实于原始程序的确切行为之间，使用者愿意做出什么样的选择。结果是，编译器倾向于保守，避免任何对功能产生影响的优化，即使是很轻微的影响。

- ① 另一方面，浮点加法满足了单调性属性：如果  $a \geq b$ ，那么对于任何  $a$ 、 $b$  以及  $r$  的值，除了  $\text{NaN}$ ，都有  $x+a \geq x+b$ 。无符号或补码加法不具有这个实数(和整数)加法的属性。(单调性)
- ② 浮点乘法也遵循通常乘法所具有的许多属性。我们定义  $x *^ f y$  为  $\text{Round}(x \times y)$ 。这个运算在乘法中是封闭的(虽然可能产生无穷大或  $\text{NaN}$ )，它是可交换的，而且它的乘法单位元

封闭(闭包)是指：在我们这个浮点的集合中的两个数，经过运算后的结果还属于该集合中，那么就说这个集合是封闭的。那么显然浮点数(实数)是封闭的；自然数就不行，例如  $3-7=-4$  就不属于自然数了。

为 1.0。另一方面，由于可能发生溢出，或者由于舍入而失去精度，它不具有可结合性。例如，单精度浮点情况下，表达式  $(1e20 * 1e20) * 1e-20$  求值为  $+\infty$ ，而  $1e20 * (1e20 * 1e-20)$  将得出  $1e20$ 。另外，浮点乘法在加法上不具备分配性。例如，单精度浮点情况下，表达式  $1e20 * (1e20 - 1e20)$  求值为 0.0，而  $1e20 * 1e20 - 1e20 * 1e20$  会得出  $\text{NaN}$ 。

另一方面，对于任何  $a$ 、 $b$  和  $c$ ，并且  $a$ 、 $b$  和  $c$  都不等于  $\text{NaN}$ ，浮点乘法满足下列单调性：

$$\begin{aligned} a \geq b \text{ 且 } c \geq 0 &\Rightarrow a *^ f c \geq b *^ f c \quad \text{就是上文的} \text{Round}(x*y) \\ a \geq b \text{ 且 } c \leq 0 &\Rightarrow a *^ f c \leq b *^ f c \end{aligned}$$

此外，我们还可以保证，只要  $a \neq \text{NaN}$ ，就有  $a *^ f a \geq 0$ 。像我们先前所看到的，无符号或补码的乘法没有这些单调性属性。

对于科学计算程序员和编译器编写者来说，缺乏结合性和分配性是很严重的问题。即使为了在三维空间中确定两条线是否交叉而写代码这样看上去很简单的任务，也可能成为一个很大的挑战。

阿贝尔群需要满足交换律和结合律

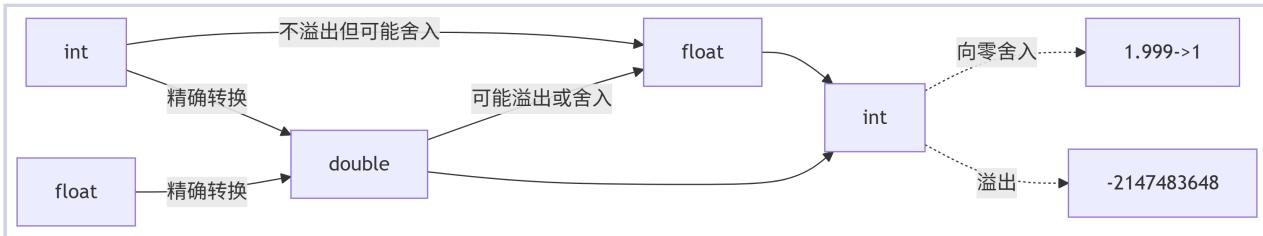
因此，我们不能说浮点数是满足阿贝尔群的，只是实数符合

## C语言中的浮点数

C语言中，float和double都是默认向偶舍入的方式，但是如果浮点数转成int，则是向零舍入了。

C语言中int、float和double之间强制转换的规则如下：

1. int -> float 不会溢出，但可能舍入
2. int/float -> double 保留精确值，更高精度
3. double -> float 可能溢出成正负无穷或者舍入
4. float,double -> int 向零舍入
5. 浮点到整数的溢出可能成TMin



```
#include <stdio.h>
#include <limits.h>
```

```
// 生成特殊浮点值的宏定义
#define POS_INFINITY (1.0e308 * 1.0e308) // 正无穷
#define NEG_INFINITY (-1.0e308 * 1.0e308) // 负无穷
#define NEG_ZERO (-1.0 / POS_INFINITY) // 负零

int main() {
    double pos_inf = POS_INFINITY;
    double neg_inf = NEG_INFINITY;
    double neg_zero = NEG_ZERO;

    printf("正无穷: %f\n", pos_inf); // 输出: inf
    printf("负无穷: %f\n", neg_inf); // 输出: -inf
    printf("负零: %f\n", neg_zero); // 输出: -0.000000
    printf("1/负零: %f\n", 1.0/neg_zero); // 输出: -inf

    // ----- 浮点数转换和舍入 -----
    // 1.int -> float 舍入
    int big_int = 16777217; // 2^24 + 1
    float f = (float)big_int;
    printf("int(%d) -> float: %.1f (舍入损失)\n", big_int, f);
    // 输出: 16777216.0 (丢失精度)

    // 2.int/float -> double 保留精确值
    double d1 = big_int;
    double d2 = f;
    printf("int(%d) -> double: %.1f\n", big_int, d1);
    printf("float(%f) -> double: %.1f\n", f, d2);
    // 输出精确值

    // 3.double -> float 溢出和舍入
```

```
double huge = 1.0e308;
float f_huge = (float)huge;
printf("double(%.1e) -> float: %f (溢出为inf)\n", huge, f_huge);
// 输出: inf

// 4.float,double -> int 向零舍入
double d3 = 1.999;
double d4 = -1.999;
int i1 = (int)d3;
int i2 = (int)d4;
printf("double(%.3f) -> int: %d\n", d3, i1); // 输出: 1
printf("double(%.3f) -> int: %d\n", d4, i2); // 输出: -1

// 5.浮点到整数的溢出 (Intel示例)
double overflow = 1e10;
int overflow_int = (int)overflow;
printf("double(%.0f) -> int: %d (溢出为负值)\n", overflow,
overflow_int);
// 输出: -2147483648 (TMin)

// 5的更多例子
d3 = INT_MAX + 1.0; // 2,147,483,648
printf("(int)%.0f = %d (溢出)\n", d3, (int)d3); // -2147483648
// 负数溢出
double d5 = INT_MIN - 1.0; // -2,147,483,649
printf("(int)%.0f = %d (负溢出)\n", d5, (int)d5); // -2147483648

return 0;
}
```

## 小结

## 2.5. 小结

计算机将信息编码为位(比特)，通常组织成字节序列。有不同的编码方式用来表示整数、实数和字符串。不同的计算机模型在编码数字和多字节数据中的字节顺序时使用不同的约定。

C语言的设计可以包容多种不同字长和数字编码的实现。64位字长的机器逐渐普及，并正在取代统治市场长达30多年的32位机器。由于64位机器也可以运行为32位机器编译的程序，我们的重点就放在区分32位和64位程序，而不是机器本身。64位程序的优势是可以突破32位程序具有的4GB地址限制。

大多数机器对整数使用补码编码，而对浮点数使用IEEE标准754编码。在位级上理解这些编码，并且理解算术运算的数学特性，对于想使编写的程序能在全部数值范围内正确运算的程序员来说，是很重要的。

在相同长度的无符号和有符号整数之间进行强制类型转换时，大多数C语言实现遵循的原则是底层的位模式不变。在补码机器上，对于一个 $w$ 位的值，这种行为是由函数 $T2U_w$ 和 $U2T_w$ 来描述的。C语言隐式的强制类型转换会出现许多程序员无法预计的结果，常常导致程序错误。

由于编码的长度有限，与传统整数和实数运算相比，计算机运算具有非常不同的属性。当超出表示范围时，有限长度能够引起数值溢出。当浮点数非常接近于0.0，从而转换成零时，也会下溢。

和大多数其他程序语言一样，C语言实现的有限整数运算和真实的整数运算相比，有一些特殊的属性。例如，由于溢出，表达式 $x*x$ 能够得出负数。但是，无符号数和补码的运算都满足整数运算的许多其他属性，包括结合律、交换律和分配律。这就允许编译器做很多的优化。例如，用 $(x << 3) - x$ 取代表达式 $7*x$ 时，我们就利用了结合律、交换律和分配律的属性，还利用了移位和乘以2的关系。

我们已经看到了几种使用位级运算和算术运算组合的聪明方法。例如，使用补码运算， $\sim x + 1$ 等价于 $-x$ 。另外一个例子，假设我们想要一个形如 $[0, \dots, 0, 1, \dots, 1]$ 的位模式，由 $w-k$ 个0后面紧跟着 $k$

这里是包括符号位取反

个1组成。这些位模式有助于掩码运算。这种模式能够通过C表达式 $(1 << k) - 1$ 生成，利用的是这样一个属性，即我们想要的位模式的数值为 $2^k - 1$ 。例如，表达式 $(1 << 8) - 1$ 将产生位模式0xFF。

浮点表示通过将数字编码为 $r \times 2^y$ 的形式来近似地表示实数。最常见的浮点表示方式是由IEEE标准754定义的。它提供了几种不同的精度，最常见的是单精度(32位)和双精度(64位)。IEEE浮点也能表示特殊值 $+\infty$ 、 $-\infty$ 和 $NaN$ 。

必须非常小心地使用浮点运算，因为浮点运算只有有限的范围和精度，而且并不遵守普遍的算术属性，比如结合性。

## 常见的位运算替代

符号	位操作表示	说明
$==$	$!(x \wedge y)$	1时表示相同，0表示不同
$* 2^n$	$x << n$	
$/ 2^n$	$x >> n$	
$\% 2$	$x \& 1$	
检查 $x$ 是否为偶数	$!(x \& 1)$	等价于 $x \% 2 == 0$

检查 x 是否为奇数Tmin	$x \& 11/-1 << 31$	等价于 $x \% 2 != 0$ -2147483648
Tmax	$0xffffffff >> 1$ 或者 $\sim 0 >> 1$ 又或者 $\sim(-1) >> 1$	2147483647
	$\sim 0u >> 1$	
isTmax	$!(x \wedge (\sim 0u >> 1))$	
$-x$	$\sim x + 1$ 或者 $\sim(x - 1)$	
$x ? y : z$	int condition = $\sim(!!x) + 1$ ; return condition & y   $\sim$ condition & z;	常将条件变量定为0和-1。这样的话 $\sim 0$ 就表示全要， $\sim(-1)$ 就表示0
0xFFFFFFFF	-1 或者 $\sim 0$	

## 程序的机器级表示

