# 计算机系统漫游

## 信息就是位+上下文

1、像hello.c这样只由ASCII字符构成的文件称为文本文件

2、基本思想：系统中的所有信息--包括磁盘文件、内存中的程序、内存中存放的用户数据以及网络上传送的数据，都是由一串bit表示的，区分不同数据对象的唯一方法使我们读到这些数据对象时的上下文。例如，在不同的上下文中，一个同样的字节序列可能表示一个整数、浮点数、字符串或者机器指令

3、C语言的起源

1. C语言与Unix操作系统关系密切：C从一开始就是作为一种用于Unix系统的程序语言开发出来的，大部分的Unix内核(操作系统的核心部分)，以及所有支撑工具和函数库都是用C语言编写的
2. C语言小而简单：C语言的设计师由一个人而非一个协会掌控的，因此这是一个简洁明了、没有什么冗赘的设计
3. C语言是为实践目的的设计：C语言是用来实现Unix操作系统的，后来其他人发现能够用这门语言无障碍地编写他们想要的程序

## 程序被其他程序翻译成不同的格式

1、hello程序的生命周期是从一个高级C语言程序开始的，因为这种形式能够被人读懂。为了在系统上运行hello.c程序，每条C语句都必须被其他程序转化为一系列低级机器语言指令。然后这些指令按照一种称为可执行目标程序的格式打包，并以二进制磁盘文件的形式存放起来。目标程序也成为可执行文件

2、在Unix系统上，从源文件到目标文件的转化是由编译器驱动程序完成的

gcc -o hello hello.c

* GCC编译器驱动程序读取源程序文件hello.c，并把它翻译成一个可执行目标文件hello。这个翻译过程可分为四个阶段完成，执行这四个阶段的程序(预处理器、编译器、汇编器、链接器)一起构成了编译系统

1. 预处理阶段：预处理器(cpp)根据以字符#开头的命令，修改原始的C程序。比如hello.c中的第一行的#include<stdio.h>命令告诉预处理器读取系统头文件stdio.h的内容，并把它直接插入程序文本中。结果得到了另一个C程序，通常以.i作为文件扩展名
2. 编译阶段：编译器(ccl)将文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序
3. 汇编阶段：汇编器(as)将hello.s翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序(relocatable object program)的格式，并将结果保存在目标文件hello.o中。hello.o是一个二进制文件
4. 链接阶段：hello程序调用了printf函数，它是每个C编译器都提供的标准C库中的一个函数。printf函数存在于一个名为printf.o的单独预编译好了的目标文件中，而这个文件必须以某种方式合并到我们的hello.o程序中。链接器(ld)就负责处理这种合并，结果就得到hello文件，它是一个可执行目标文件，可以被加载到内存中，由系统执行

## 了解编译系统如何工作是大有益处的

1、优化程序性能

2、理解链接时出现的错误

3、避免安全漏洞

## 处理器读并解释存储在内存中的指令

1、在Unix系统上运行可执行文件，将它的文件名输入到称为shell的应用程序中

### 系统的硬件组成

1、总线：

* 贯穿整个系统的是一组电子管道，称为总线，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)。字中的字节数(即字长)是一个基本的系统参数，各个系统不尽相同。现在大多数机器字长要么是4个字节(32位)，要么是8个字节(64位)

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总线相连
* 控制器与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(通常称为主板)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

3、主存

* 主存是一个临时存储设备，在处理器执行程序时，用来存放程序和程序处理的数据
* 物理上来说，主存是由一组动态随机存取存储器(DRAM)芯片组成的
* 从逻辑上来说，存储器是一个线性的字节数组，每个字节都具有唯一的地址(数组索引)，这些地址是零开始的
* 一般来说，组成程序的每条机器指令都由不同数量的字节构成
* 与C程序变量相对应的数据项的大小是根据类型变化的

4、处理器

* 中央处理单元(CPU)，简称处理器，是解释(或执行)存储在主存中指令的引擎
* 处理器的核心是一个大小为一个字的存储设备(或寄存器)，称为程序计数器(PC)。在任何时候，PC都指向主存中的某条机器语言指令(即含有该条指令的地址)
* 从系统通电开始，直到系统断电，处理器一直不断地执行程序计数器指向的指令，再更新程序计数器，使其指向下一条指令。处理器看上去是按照一个非常简单的指令执行模型来操作的，这个模型是由指令集架构决定的。在这个模型中，指令按照严格的顺序执行，而执行一条指令包含执行一系列的步骤。处理器从程序计数器指向的内存处读取指令，解释指令中的位，执行该指令指示的简单操作，然后更新PC，使其指向下一条指令，而这条指令并不一定和在内存中刚刚执行的指令相邻
* 以下是CPU在指令的要求下可能执行的操作

1. 加载：从主存复制一个字节或一个字到寄存器，以覆盖寄存器原来的内容
2. 存储：从寄存器复制一个字节或一个字到主存的某个位置，以覆盖这个位置上原来的内容
3. 操作：把两个寄存器的内容复制到ALU，ALU对这两个字做算数运算，并将结果存放到一个寄存器中，以覆盖该寄存器中原来的内容
4. 跳转：从指令本身中抽取一个字，并将这个字复制到程序计数器(PC)中，以覆盖PC中原来的值

## 高速缓存至关重要

1、系统花费了大量的时间把信息从一个地方挪到另一个地方

* hello程序的机器指令最初存放在磁盘上，当程序加载时，它们被复制到主存
* 当处理器运行程序时，指令又从主存复制到处理器
* 数据串"hello,world\n"开始在磁盘上，然后被复制到主存，最后从主存上复制到显示设备

2、从程序员的角度看，这些复制就是开销，减慢了程序"真正"的工作，因此系统设计者的一个主要目标就是使这些复制操作尽可快地完成

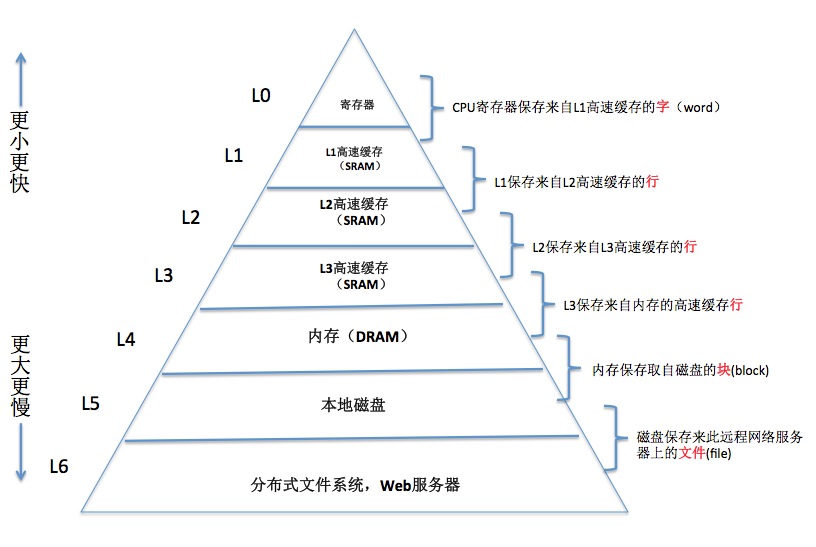
3、根据机械原理

* 较大的存储设备要比较小的存储设备运行得慢，而快速设备的造价远高于同类的低速设备。例如，一个典型系统上的磁盘驱动器可能比主存大1000倍，但是对于处理器而言，从磁盘驱动器上读取一个字的时间开销要比从主存中读取的开销大1000万倍
* 类似地，一个典型的寄存器文件只存储几百字节的信息，而主存里可存放几十亿字节，然而，处理器从寄存器文件中读取数据比从主存中读取几乎要快100倍
* 更麻烦的是，随着半导体技术的进步，这种处理器与主存之间的差距还在持续增大。加快处理器运行速度比加快主存运行速度要容易得多

4、针对这种处理器与主存之间的差异，系统设计采用了更小更快的存储设备，称为高速缓存存储器(cache memory，简称为cache或高速缓存)，作为暂时的集结区域，存放处理器近期可能会需要的信息

## 存储设备形成层次结构

1、在处理器和一个较大较慢的设备(例如主存)之间插入一个更小更快的存储设备(例如高速缓存)的想法已经成为一个普遍的观念。实际上，每个计算机系统中的存储设备都被组织成了一个存储器层次结构



2、存储器层次结构的主要思想是上一层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

## 操作系统管理硬件

1、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，所有程序对硬件的操作尝试都必须通过操作系统

2、操作系统的基本功能

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

3、操作系统通过几个基本抽象概念来实现这两个功能

1. 进程
2. 虚拟内存
3. 文件

4、计算机系统分层图以及操作系统抽象表示如下图所示





* 文件是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

### 进程

1、像hello这样的程序在现代操作系统上运行时，操作系统会提供一种假象，就像系统上只有这个程序在运行，程序看上去独占使用处理器、主存和I/O设备。处理器看上去像不间断地一条接一条地执行程序中的命令，即该程序代码和数据是系统内存中唯一的对象。这些假象是通过进程的概念来实现的，进程是计算机科学中最重要和最成功的的概念之一

2、进程是操作系统对一个正在运行的程序的一种抽象，在一个系统上可以同时运行多个进程，而每个进程都好像独占地使用硬件。而并发运行，则是说一个进程的指令和另一个进程的指令是交错执行的。在大多数系统中，需要运行的进程是多于可以运行它们的CPU个数的

3、传统系统在一个时刻只能执行一个程序，而先进的多核处理器同时能够执行多个程序。无论在单核还是多核系统中，一个CPU看上去都像是在并发地执行多个进程，这是通过处理器在进程间切换来实现的。操作系统实现这种交错执行的机制称为上下文切换

4、操作系统保持跟踪进程运行所需的所有状态信息。这种状态，就是上下文，包括许多信息，比如PC和寄存器文件的当前值，以及主存的内容。当操作系统决定要把控制权从当前进程转移到某个新进程时，就会进行上下文切换，即保存当前进程的上下文、恢复进程的上下文，然后将控制权传递到新进程，新进程就会从它上次停止的地方开始

5、从一个进程到另一个进程的转换时由操作系统内核(kernel)管理的，内核是操作系统代码常驻主存的部分。当应用程序需要操作系统的某些操作时，比如读文件，它就执行一条特殊的系统调用(system call)指令，将控制权传递给内核，然后内核执行被请求的操作并返回给应用程序。

6、注意，内核不是一个独立的进程，相反，它是系统管理全部进程所用代码和数据结构的集合

### 线程

1、通常我们认为一个进程只有单一的控制流，但在现代系统中，一个进程实际上可以由多个称为线程的执行单元组成，每个线程都运行在进程的上下文中，并共享同样的代码和全局数据。

2、由于网络服务器中对并行处理的需求，线程成为越来越重要的编程模型，因为多线程之间比多进程之间**更容易**共享数据(进程间也能共享数据)，也因为线程一般来说比进程更高效

3、到底什么是线程和进程呢

### 虚拟内存

1、虚拟内存是一个抽象概念，它为每个进程提供了一个假象，即每个进程都在独占地使用主存，每个进程看到的内存都是一致的，称为虚拟地址空间



* 图中地址从下往上是增大的

2、逐个介绍每个内存区域

1. 程序代码和数据

* 对所有进程来说，代码是从同一固定地址开始，紧接着是和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

1. 堆

* 代码和数据区后紧随着是运行时堆，代码区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

1. 共享库

* 大约在内存地址空间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，第七章有详细介绍

1. 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
* 每次调用一个函数时，栈就会增长
* 每次从一个函数返回，栈就会收缩
* 第三章将会学习到编译器如何使用栈

1. 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

1、文件就是字节序列，仅此而已

2、每个I/O设备，包括磁盘、键盘、显示器，甚至网络，都可以看成是文件

3、系统中的所有输入输出都是通过一小组称为Unix I/O的系统函数调用读写文件来实现的

4、文件这个简单而精致的概念是非常强大的，因为它向应用程序提供了一个统一的视图，来看待系统中可能含有的所有各式各样的I/O设备

## 系统之间利用网络通信

1、实际上，现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，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来的数据，并把数据复制到自己的主存

2、客户端和服务器之间交互的类型在所有的网络应用中是非常典型的

## 重要主题

### Amdahl定律

1、想要显著加速整个系统，必须提升全系统中相当大的部分的速度

### 并发和并行

1、两个术语

1. 并发：指一个同时具有多个活动的系统
2. 并行：指的是用并发来使系统运行得更快

2、并行可以在计算机系统的多个抽象层次上运用

1. 线程级并发
2. 指令级并发
3. 单指令多数据并行

### 计算机系统中抽象的重要性

1、抽象的使用时计算机科学中最为重要的概念之一。例如，为一组函数规定一个简单的应用程序接口(API)就是一个很好地编程习惯，程序员无需了解它内部的工作便可以使用这些代码

2、在处理器里的抽象

1. 指令集架构提供了对实际处理器硬件的抽象，使用这个抽象，机器代码程序表现的就好像运行在一个一次只执行一条指令的处理器上
2. 底层的硬件远比抽象描述的要复杂精细，它并行地执行多条指令，但又总是与那个简单有序的模型保持一致，只要执行模型一样，不同的处理器实现也能执行相同的机器码，而又能提供不同的开销和性能

3、操作系统里的抽象

1. 文件：对I/O设备的抽象
2. 虚拟内存：对程序存储器的抽象
3. 进程：对一个正在运行的程序的抽象
4. 虚拟机：它提供对整个计算机的抽象

## 再探进程与线程

### 进程

1、进程(process)是一块包含了某些资源的内存区域。操作系统利用进程把它的工作划分为一些功能单元。进程中所包含的一个或多个执行单元称为线程(thread)。进程还拥有一个私有的虚拟地址空间，该空间仅能被它所包含的线程访问

2、应用程序(application)是由一个或多个相互协作的进程组成的。例如，Visual Studio开发环境就是利用一个进程编辑源文件，并利用另一个进程完成编译工作的应用程序

3、在Windows NT/2000/XP操作系统下，我们可以通过任务管理器在任意时间查看所有的应用程序和进程。尽管只打开了几个应用程序，但是通常情况下将有大约30个进程同时运行。事实上，为了管理当前的会话和任务栏以及其他一些任务，系统执行了大量的进程

4、进程，是并发执行的程序在执行过程中分配和管理资源的基本单位，是一个动态概念，竟争计算机系统资源的基本单位。每一个进程都有一个自己的地址空间，即进程空间或(虚空间)。进程空间的大小只与处理机的位数有关，一个16位长处理机的进程空间大小为216，而32位处理机的进程空间大小为232。进程至少有5种基本状态，它们是：初始态，执行态，等待状态，就绪状态，终止状态

5、在运行于32位处理器上的32位Windows操作系统中，可将一个进程视为一段大小为4GB(232字节)的线性内存空间，它起始于0x00000000结束于0xFFFFFFFF。这段内存空间不能被其他进程所访问，所以称为该进程的私有空间。这段空间被平分为两块，2GB被系统所有，剩下2GB被用户所有。如果有N个进程运行在同一台机器上，那么将需要N×4GB的海量RAM，还好事实并非如此。

* Windows是按需为每个进程分配内存的，4GB是32位系统中一个进程所占空间的上限
* 将进程所需的内存划分为4KB大小的内存页，并根据使用情况将这些内存页存储在硬盘上或加载到RAM中，通过系统的这种虚拟内存机制，我们可以有效地减少对实际内存的需求量。当然这些对用户和开发者来说都是透明的

### 线程

1、线程只能归属于一个进程并且它只能访问该进程所拥有的资源。当操作系统创建一个进程后，该进程会自动申请一个名为主线程或首要线程的线程。主线程将执行运行时宿主，而运行时宿主会负责载入CLR

2、一个线程包含以下内容

1. 一个指向当前被执行指令的指令指针
2. 一个栈
3. 一个寄存器值的集合，定义了一部分描述正在执行线程的处理器状态的值
4. 一个私有的数据区

* 所有这些元素都归于线程执行上下文的名下。处在同一个进程中的所有线程都可以访问该进程所包含的地址空间，当然也包含存储在该空间中的所有资源

3、进程的执行过程是线状的，尽管中间会发生中断或暂停，但该进程所拥有的资源只为该线状执行过程服务。一旦发生进程上下文切换，这些资源都是要被保护起来的。这是进程宏观上的执行过程。而进程又可有单线程进程与多线程进程两种。我们知道，进程有一个进程控制块PCB，相关程序段和该程序段对其进行操作的数据结构集这三部分，单线程进程的执行过程在宏观上是线性的，微观上也只有单一的执行过程；而多线程进程在宏观上的执行过程同样为线性的，但微观上却可以有多个执行操作(线程)，如不同代码片段以及相关的数据结构集。线程的改变只代表了CPU执行过程的改变，而没有发生进程所拥有的资源变化。除了CPU之外，计算机内的软硬件资源的分配与线程无关，线程只能共享它所属进程的资源。与进程控制表和PCB相似，每个线程也有自己的线程控制表TCB，而这个TCB中所保存的线程状态信息则要比PCB表少得多，这些信息主要是相关指针用堆栈(系统栈和用户栈)，寄存器中的状态数据。进程拥有一个完整的虚拟地址空间，不依赖于线程而独立存在；反之，线程是进程的一部分，没有自己的地址空间，与进程内的其他线程一起共享分配给该进程的所有资源

### 进程间的通信方式

#### 无名管道通信

1、无名管道(pipe)：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系

#### 高级管道通信

1、高级管道(popen)：将另一个程序当做一个新的进程在当前程序进程中启动，则它算是当前程序的子进程，这种方式称为高级管道方式

#### 有名管道通信

1、有名管道(named pipe)：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信

#### 消息队列通信

1、消息队列(message queue)：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点

#### 信号量通信

1、信号量(semophore)：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段

#### 信号

1、信号(sinal)：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生

#### 共享内存通信

1、共享内存(shared memory)：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的IPC方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信

#### 套接字通信

1、套接字(socket)： 套接口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同机器间的进程通信

#### 系列问题

1、管道与文件描述符、文件指针的关系？

* 答：其实管道的使用方法与文件类似，都能使用read，write，open等普通IO函数。管道描述符来类似于文件描述符。事实上，管道使用的描述符，文件指针和文件描述符最终都会转化成系统中SOCKET描述符。都受到系统内核中SOCKET描述符的限制。本质上LINUX内核源码中管道是通过空文件来实现

2、管道的使用方法？

* 答：主要有下面几种方法:
* pipe：创建一个管道，返回2个管道描述符。通常用于父子进程之间通讯
* popen，pclose：这种方式只返回一个管道描述符，常用于通信另一方是stdin or stdout
* mkpipe：命名管道，在许多进程之间进行交互

3、管道与系统IPC之间的优劣比较？

* 答：管道
* 优点是所有的UNIX实现都支持，并且在最后一个访问管道的进程终止后，管道就被完全删除
* 缺陷是管道只允许单向传输或者用于父子进程之间
* 系统IPC
* 优点是功能强大，能在毫不相关进程之间进行通讯
* 缺陷是关键字KEY\_T使用了内核标识，占用了内核资源，而且只能被显式删除，而且不能使用SOCKET的一些机制，例如select,epoll等.

4、WINDOS进程间通信与LINUX进程间通信的关系？

* 答：事实上，WINDOS的进程通信大部分移植于UNIX，WINDOS的剪贴板，文件映射等都可从UNIX进程通信的共享存储中找到影子

5、进程间通信与线程间通信之间的关系？

* 答：因为WINDOWS运行的实体是线程，狭义上的进程间通信其实是指分属于不同进程的线程之间的通讯。而单个进程之间的线程同步问题可归并为一种特殊的进程通信。它要用到内核支持的系统调用来保持线程之间同步。通常用到的一些线程同步方法包括：Event，Mutex，信号量Semaphore，临界区资源等.

# 信息的表示和处理

1、二进制信号能够容易地被表示、存储和传输。例如可以表示为穿孔卡片上的有洞或无洞、导线上的高压电和低压电，或者顺时针或逆时针的磁场

2、二进制信号进行存储和执行计算的电子电路非常简单和可靠，制造商能够在一个单独的硅片上集成数百万甚至数亿个这样的电路

3、孤立地讲，单个位不是非常有用，当把位组合在一起，再加上某种解释，即赋予不同的可能位模式以含义，我们就能够表示任何有限集合的元素

## 信息存储

1、大多数计算机使用8位的块，或者字节，作为最小的可寻址的内存单位，而不是访问内存中单独的位

2、机器级程序将内存视为一个非常大的字节数组，称为虚拟内存

3、内存的每个字节都由一个唯一的数字来标识，称为它的地址，所有可能地址的集合就成为虚拟地址空间

### 十六进制表示法

1、十六进制采用0~9，以及A~F来表示16个可能的值

### 字数据大小

1、每台计算机都由一个字长(word size)，指明指针数据的标称大小(nominal size)。因为虚拟地址是以这样的一个字来编码的，所以字长决定的最重要的系统参数就是虚拟地址空间的最大大小。对于一个字长为w位的机器而言，虚拟地址的范围为0~2w-1，即最多访问2w个字节

2、大多数64位机器可以运行32位机器编译的程序，这是一种向后兼容

3、32位程序或64位程序，区别在于该程序是如何编译的，而不是其运行的机器类型

### 寻址和字节顺序

1、对于跨多字节的程序对象，我们必须建立两个规则：这个对象地址是什么，以及在内存中如何排列这些字节(注意本小结讨论的是字节顺序，而非一个字节中的位的顺序)

2、在几乎所有机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址

3、排列表示一个对象的字节由两个通用的规则

* 考虑一个w位的整数，其为表示为[xw-1,xw-1,...,x1,x0]，其中xw-1是最高有效位，x0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[xw-1,xw-1,...,xw-8]，最低有效字节包含[x7,x6,..,x0]
* 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
* 最低有效字节在最前面的方式，称为小端法(little endian)
* 最高有效字节在最前面的方式，称为大端法(big endian)

例如整型变量x，其值为0x01234567

大端法为：01 23 45 67

小端法为：67 45 23 01

* 注意，无论大端还是小端，单个字节中的顺序都是一样的，不同的是多个字节的排列顺序。因此char类型，无论是大端法还是小端法都是相同的，因为只有一个字节，无所谓字节顺序

4、大多数Intel兼容机都只用小端模式

5、对于大多数应用程序员来说，其机器所使用的的字节顺序是完全不可见的，无论为哪种类型的机器所编译的程序都会得到相同的结果

6、不过有时候，字节顺序会造成问题

* 首先是在不同类型的机器之间通过网络传送二进制数据时，一个常见的问题是当小端法机器产生的数据被发送到大端法机器或者反过来说时，接受程序会发现，字里的字成了反序的，为了避免这种问题，网络应用程序的代码编写必须遵守已建立的关于字节顺序的规则，以确保发送方机器将它的内部表示转换成网络标准，而接收方机器则将网络标准转换为它的内部表示
* 当阅读表示整数数据的字节序列时字节顺序也很重要

### 表示字符串

1、C语言中字符串被编码为一个以null(置为0，即'\0')字符结尾的字符数组，每个字符都由某个标准编码来表示，最常见的是ASCII码

2、使用ASCII码作为字符码的任何系统上都将得到相同的结果，与字节顺序与字大小规则无关(因为char就一个字节，无所谓字节顺序)，因此，文本数据比二进制数据更具有更强的平台独立性

### 表示代码

1、不同的机器类型使用不同的且不兼容的指令和编码方式，即使是完全一样的进程，运行在不同操作系统上也会有不同的编码规则。因此二进制代码是不兼容的，二进制代码很少可能在不同机器和操作系统组合之间移植

### 布尔代数简介

1、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

2、最简单的布尔代数是在二元集合{0，1}基础上定义的

3、位向量：位向量一个很有用的应用就是表示有限集合，我们可以用位向量[aw-1,...,a1,a0]编码任何子集A⊆{0,1,...,w-1}，其中ai=1当且仅当i∈A。aw-1写在左边，将a0写在右边，例如位向量a=[01101001]表示集合A={0,3,5,6}

### C语言中的位级运算

1、位运算

1. |
2. &
3. ^

2、神奇的inplace\_swap

void inplace\_swap(int& x ,int& y){

y=x^y;

x=x^y;//x=x^(x^y)=0^y=y

y=x^y;//y^(x^y)=y^(y^x)=0^x=x

}

### C语言中的逻辑运算

1、逻辑运算

1. ||
2. &&
3. !

### C语言中的移位运算

1、移位运算，对于一个位表示为[xw-1,xw-2,...,x0]的操作数x

1. <<：x向左移动k位，丢弃最高k位，并在右端补k个0

[xw-k-1,xw-k-2,...,x0,0,...,0]

1. >>(逻辑右移)：逻辑右移在左端补k个0

[0,...,0,xw-1,xw-2,...,xk]

1. >>(算数右移)：算数右移在左端补k个最高有效位的值

[xw-1,...,xw-1,xw-1,xw-2,...,xk]

2、C语言标准并没有明确定义对于有符号数应该使用哪种类型的右移---算数右移和逻辑右移都可以，这就意味着任何假设一种或者另一种右移形式的代码都可能遇到可移植性的问题。实际上，几乎所有的编译器/机器组合都对有符号数使用算数右移，且许多程序员也都假设机器会使用这种右移(不对啊，VS上默认是符号右移啊)

## 整数表示

1、用位来编码整数的两种不同方式：一种只能表示非负数，一种能够表示负数、零和正数

2、符号说明

* B2Tw：二进制转补码
* B2Uw：二进制转无符号数
* U2Bw：无符号数转二进制
* U2Tw：无符号数转补码
* T2Bw：补码转二进制
* T2Uw：补码转无符号数
* TMinw：最小补码值
* TMaxw：最大补码值
* UMaxw：最大无符号数
* ：补码加法
* ：无符号数加法
* ：补码乘法
* ：无符号数乘法
* ：补码取反
* ：无符号数取反

### 整型数据类型

1、C语言支持多种整型数据类型---表示有限范围的整数

2、整数类型

1. char
2. short
3. int
4. long

3、C和C++都支持无符号数，Java只支持有符号数

4、C语言只规定了整形数据类型的最小取值范围，至于正真的取值范围与机器有关，例如long就是一个与机器相关的类型，32位机器就是32位，64位机器就是64位

1. [signed]char [-127,127]
2. unsigned char [0,255]
3. short [-32767,32767]
4. unsigned short [0,65535]
5. int [-32767,32767]
6. unsigned [0,65535]
7. long [-2147483647,2147483647]
8. unsigned long [0,4294967295]
9. int32\_t [-2147483648,2147483647]
10. uint32\_t [0,4294967295]
11. int64\_t [-9223372036854775808,9223372036854775807]
12. uint64\_t [0,18446744073709551615]

### 无符号数的编码

1、假设有一个整数数据类型有w位，可以将向量写成，表示整个向量，或者写成[xw-1,xw-2,...,x0]

2、无符号数编码的定义，我们用函数B2Uw(Binary to Unsigned的缩写，长度w)来表示



3、无符号编码满足唯一性：函数B2Uw是一个双射

### 补码编码

1、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)，在这个定义中，将字的最高有效位解释为负权(negative weight)，我们用B2Tw(Binary to Two's-complement的缩写，长度为w)来表示

2、补码的定义



3、补码所能表示的范围

1. 最小值：设置负权位，清除其他所有位，TMinw=-2w-1
2. 最大值：清除负权位，设置其他所有位，TMaxw=

4、补码编码满足唯一性：函数B2Tw是一个双射

5、有符号数的其他表示方法

1. 反码：
2. 原码：

* 这两种表示方式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
* 几乎所有现代机器都采用补码来表示有符号数

6、注意补码(Two's complement)和反码(Ones' complement)中撇号的位置是不同的

* 术语补码来源于这样一个情况，对于非负数x，我们用2w-x(这里只有一个2)来计算-x的w位表示
* 术语反码来源于这样一个情况，我们用[111...1]-x(这里有很多个1)来计算-x的反码表示

### 有符号数和无符号数之间的转换

1、C语言允许在各种不同的数字数据类型之间做强制类型转换

2、**强制类型转换的结果保持位值不变，只是改变了解释这些位的方式**

3、对于大多数C语言的实现，处理同样字长的有符号数和无符号数之间相互转换的一般规则是：数值可能会改变，但是位模式不变。为了用更数学的形式来描述这个规则，我们定义函数U2Bw和T2Bw，它们将数值映射为无符号数和补码形式的表示

* 给定0≤x≤UMaxw，函数U2Bw(x)会给出x的唯一w位无符号表示
* 给定TMinw≤x≤TMaxw，函数T2Bw(x)会给出x的唯一w位补码表示

4、现在将函数T2Uw定义为T2Uw=B2Uw(T2Bw(x))

5、补码转换为无符号数：对满足TMinw≤x≤TMaxw的x有



6、无符号数转为补码：对于满足0≤u≤UMaxx的u



### C语言中的有符号数和无符号数

1、C语言支持所有整型数据类型的有符号和无符号运算。尽管C语言标准并没有指定有符号数要采用某种表示，但是几乎所有的机器都使用补码

2、通常，要创建一个无符号常量，必须加上后缀字符'U'或者'u'，例如

12345U或者0x1A2Bu

3、当一种类型的表达式被赋值给另一种类型的变量时，转换是隐式发生的(转换被允许的话，对于基本类型由编译器决定，对于类类型由单个参数且非explicit的构造函数决定)

4、C语言对同时包含有符号和无符号数表达式，有奇特的行为

1. 当执行一个运算时，如果它的一个运算数是有符号的而另一个时无符号的，C语言会隐式地将有符号参数强制转换成无符号数，来执行这个运算

5、注意TMin的写法

1. 我们将TMin32写成-2147483647-1，而不是写成-2147483648或0x80000000

* 0x80000000大于INT\_MAX，int装不下，因此会被当成unsigned int
* 这是因为int的最小值是-2147483648，最大值是2147483647，但是我们不能用int n = -2147483648。因为编译器(VS2015)在看到int n = -2147483648;的时候，首先判断2147483648 > INT\_MAX，知道int装不下，于是决定使用unsigned int。然后发现前面还有个负号，但是编译器会拒绝无符号数的取反运算，因此错误

### 扩展一个数字的位表示

1、要将一个无符号数转换成一个更大数据类型时，我们只需要简单地在表示的开头添加0，这种运算被称为零扩展(zero extension)

[uw-1,uw-2,...,u0]--->[0,...,0,uw-1,uw-2,...,u0]

2、要将一个补码数字转换成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

[xw-1,xw-2,...,x0]--->[xw-1,...,xw-1,xw-1,xw-2,...,x0]

* 这样做是为了抵消由于符号位权值变大带来的增益

### 截断数字

1、当将一个w位的数截断为一个k位数字时，我们会丢弃高w-k位，截断一个数字可能会改变它的值，这是溢出的一种形式

2、无论是有符号数和无符号数

[xw-1,xw-2,...,x0]---> [xk-1,xk-2,...,x0]

### 关于有符号数和无符号数的建议

1、有符号数到无符号数的隐式强制类型转换导致了某些非直观的行为

2、避免这类错误的一种方法就是绝不使用无符号数。实际上除了C语言外，很少有语言支持无符号数，因为设计者认为他们带来的麻烦要比利益多得多。例如Java只支持有符号整数，并且要求以补码运算来实现。正常的右移运算符>>被定义为执行算数右移。特殊的运算符>>>被定义为执行逻辑右移

3、当我们想要把字仅仅看做是位的集合而没有任何数字意义时，无符号数值是非常有用的

## 整数运算

### 无符号数加法

1、无符号数加法，两个w位的无符号整数相加，其和是w+1位的整数



* 当多出的最高位是0，丢弃没有任何影响
* 当多出的最高位是1，丢弃会造成影响

2、当执行C程序时，不会将溢出作为错误而发出信号

3、检测无符号数加法中的溢出

在对范围0≤x，y≤UMaxw中的x和y，令，则对计算s，当且仅当s<x(或者等价地s<y)时，发生了溢出

### 补码的加法

1、对于补码加法，我们必须确定当结果太大(为正)或者太小(为负)

2、对于在给定范围-2w-1≤x,y≤2w-1-1之内的整数值x和y，它们的和就在范围  
-2w≤x+y≤2w-2之内，要想准确表示，可能需要w+1位

3、定义为整数和x+y被截断为w位的结果，并将这个结果看做是补码数

对于满足-2w-1≤x,y≤2w-1-1的整数x和y



* 正溢出由于xw-2位的和向xw-1产生了进位，导致符号位从0变成1
* 负溢出由于符号位xw-1向更高位产生了进位(被丢弃)，导致符号位从1变成0

### 补码的非

1、对于满足TMinw≤x≤TMaxw的x，其补码的非



* TMinw+TMinw=-2w-1+(-2w-1)=-2w，这将导致负溢出结果为0，因此TMinw的求反就是其本身

2、补码非聪明方法

-x=~x+1

### 无符号数乘法

1、对于在给定范围0≤x,y≤2w-1之内的整数值x和y，它们的乘积就在范围0到(2w-1)2=22w-2w-1+1之间，这就需要2w位来表示。但是C语言中的无符号数乘法被定义为产生w位的值，就是2w位的整数乘积的低w位表示的值

2、对于满足0≤x,y≤UMaxw的x和y



### 补码乘法

1、对于在给定范围-2w-1≤x,y≤2w-1-1之内的整数值x和y，它们的乘积就在范围  
-2w-1\*(2w-1-1)=-22w-2+2w-1到-2w-1\*-2w-1=-22w-2。想要用补码来表示这个乘积，就需要2w位。在C语言中有符号数的乘法是通过2w位的乘积截断为w位来实现的

2、未完成

### 乘以常数

1、在大多数机器上，整数乘法指令相当慢，需要10个或更多时钟周期，而其他整数运算(例如加法、减法、位级运算和移位)只需要一个时钟周期，即便在i7上，整数乘法也需要3个时钟周期

2、因此，编译器采用了一项重要优化，试着用位移和加法运算的组合来代替乘以常数因子的乘法

3、乘以2的幂：左移k位(无论是无符号数还是有符号数，对于溢出情况也能得到一致的结果，因为对于溢出的位都是采用截断处理)

4、乘以任意整数的幂

* 首先将整数分解成2的幂次求和，就是那些位为1的幂之和
* 然后在进行移位运算

### 除以2的幂

1、在大多数机器上，整数除法比整数乘法更慢---需要30个或者更多的时钟周期

2、除以2的幂也可以用位移运算来实现，只不过用的是右移，而不是左移

3、无符号数用逻辑移位来进行处理，有符号数用算数移位来处理

## 浮点数

### 二进制小数

1、理解浮点数的第一步是考虑含有小数值的二进制数字



2、我们仅考虑有限长度的编码，那么大部分十进制并不能准确表达，只能近似的表达，增加二进制表示的长度可以提高表示的精度

### IEEE浮点表示

1、我们希望通过给定x和y的值，来表示形如的数

2、IEEE浮点标准用

1. 符号(sign)：s决定这个数是负数(s=1)还是正数(s=0)，而对于数值0的符号位解释为作为特殊情况处理
2. 尾数(significand)：M是一个二进制小数，它的范围，或者是
3. 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)

3、将浮点数表示划分为三个阶段，分别对这些值进行编码

1. 一个单独的符号位s直接编码符号s
2. k位的阶码字段exp=ek-1...e1e0编码阶码E
3. n位小数字段frac=fn-1...f1f0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

4、单精度和双精度如下



5、给定位表示，根据exp的值，被编码的值可以分为三种不同情况(最后一种情况有两个变种)



#### 情况1：规格化的值

1、当exp的位模式既不是全0(数值0)也不是全1(单精度255，双精度2047)

2、在这种情况下，阶码字段被解释为以偏置(biased)形式表示的有符号整数。阶码的值时E=e-Bias，其中e是假定阶码字段是一个无符号整数时所表示的值，其位表示为ek-1...e1e0，而Bias是一个等于2k-1-1(单精度是127，双精度是1023)的偏置值，由此产生指数的取值范围

* 对于单精度是-126~127
* 对于双精度是-1022~1023

3、小数字段frac被解释为描述小数值f，其中0≤f<1，其二进制表示为0.fn-1...f1f0，也就是二进制小数点在最高有效位的左边。尾数定义为M=1+f

* 有时，这种方式也叫作隐含的以1开头的(implied leading 1)表示，因为我们可以把M看成一个二进制表达式为1.fn-1...f1f0的数字
* 既然我们总是能够调整阶码E，使得尾数M在范围1≤M<2之中，那么这种表示方法是一种轻松获得一个额外精度位的技巧。既然第一位总是等于1，那我们就不需要显式地表示它

#### 情况2：非规格化的值

1、当阶码域全为0时，所表示的数是非规格化形式。在这种情况下，阶码值时E=1-Bias，而尾数的值时M=f，也就是小数字段的值，不包含隐含的开头1

2、为什么要这样设置偏置值

* 使阶码值为1-Bias而不是简单的-Bias似乎是违反直觉的(因为按照规格化值得定义，E=e-Bias=0-Bias=-Bias)
* 这种方式提供了一种从非规格化值平滑转换到规格化值得方法(详见P80)

3、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，因为使用规格化数，我们必须总是使M≥1，因此我们就不能表示0
* +0.0的浮点表示的位模式为全0：符号位是0，阶码字端全为0，小数域全为0，这就得到了M=f=0
* -0.0的浮点表示的位模式为：符号位1，阶码字段全0，小数域全0
* 根据IEEE的浮点格式，值+0.0和-0.0在某些方面是不同的，而在其他方面是相同的
* 非规格化数的另外一个功能是表示那些非常接近于0.0的数，它们提供了一种属性，称为逐渐溢出(gradual underflow)

#### 情况3：特殊值

1、当阶码全为1，小数域全为0时，得到的值表示无穷大

* 当s=0表示正无穷大
* 当s=1表示负无穷大

2、当阶码全为1，小数域不为0时，得到的值称为"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，就会返回这样的NaN值

### 数字示例

1、未完成

### 舍入

1、因为表示方法限制了浮点数的范围和精度，所以浮点运算只能近似地表示实数运算

2、因此，对于值x，我们一般想用一种系统的方法，能够找到"最接近的"匹配值x'，它可以用期望的浮点数形式表示出来。这就是舍入(rounding)运算的任务

3、一个关键的问题是在两个可能值得中间确定舍入方向。例如我有1.50美元，想把它舍入到最接近的美元数，应该是1还是2呢。一种可选的方法是维持实际数字的上届和下届。例如，我们可以确定可表示的值x-和x+，使得x的值位于它们之间：x-≤x≤x+

4、四种舍入方式

1. 向偶数舍入：也被称为向最接近的值舍入
2. 向零舍入
3. 向上舍入
4. 向下舍入

5、向偶数舍入初看上去好像是个相当随意的目标，但是向偶数舍入可以避免统计偏差

* 例如，如果采用向上舍入，会在计算平均值数中引入统计偏差，得到的平均值会略大一些
* 而向偶数舍入在50%的时间里，向上舍入，在50%的时间里，向下舍入

### 浮点运算

1、IEEE标准制定了一个简单的规则，来确定诸如加法和乘法这样的算数运算的结果。把浮点值x和y看成实值，而某个运算定义在实数上，计算将产生Round(xy)，这是对实际运算的精确结果进行舍入后的结果

2、IEEE标准中制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实现或软件实现。因此，我们可以检查他的抽象数学属性，而不必考虑它实际上是如何实现的

3、浮点加法满足了单调性属性，如果a≥b，那么对于任何a、b以及x值，除了NaN，满足x+a≥x+b

4、浮点乘法遵循通常乘法的许多属性，我们定义

1. 这个运算在乘法中是封闭的(虽然可能产生无穷大或者NaN)，它是可交换的，而且它的乘法单元为1.0
2. 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(1e20\*1e20)\*1e-20=+∞，而1e20(1e20\*1e-20)将得出1e20
3. 浮点乘法在加法上不具有分配性。例如1e20\*(1e20-1e20)求值为0.0，而1e20\*1e20-1e20\*1e20会得出NaN
4. 另一方面，对于任何a、b和c，并且a、b和c都不等于NaN，浮点乘法满足下列单调性



1. 只要a不为NaN，就有，而无符号数或补码的乘法没有这些单调性属性



5、未完成

### C语言中的浮点数

1、所有C语言版本提供了两种不同的浮点数据类型：float和double

# 程序的机器表示

## 历史观点

## 程序编码

1、编译例子

gcc -Og -o p p1.c p2.c

1. 编译选项Og告诉编译器使用会生成符合原始C代码整体结构的机器代码的优化等级。使用较高级别的优化产生的代码会严重变形，以至于产生的机器代码和初始源码之间的关系非常难以理解

### 机器级代码

1、计算机系统使用了许多不同形式的抽象，利用更简单的抽象模型来隐藏实现的细节

2、对于机器级编程来说，两种抽象尤为重要

* 第一种抽象：指令集体系结构或指令集架构(Instruction Set Architecture, ISA)来定义机器级程序的格式和行为，它定义了处理器状态、指令格式，以及每条指令对状态的影响
* 第二种抽象：机器级程序使用的内存地址是虚拟地址，提供的内存模型看上去是一个非常大的字节数组。存储系统的实际实现是将多个硬件存储器和操作系统软件组合起来

3、在整个编译过程中，编译器会完成大部分的工作，将把用C语言提供的相对比较抽象的执行模型表示的程序转化成处理器执行的非常基本的命令

4、汇编代码表示非常接近于机器代码，与机器代码的二进制格式相比，汇编代码主要特点是它用可读性更好的文本格式表示。能够理解汇编代码以及它与原始C代码的联系，是理解计算机如何执行程序的关键一步

5、一些通常对C语言程序员隐藏的处理器状态

1. 程序计数器：通常称为"PC"，在x86-64中用%rip表示，给出将要执行的下一条指令在内存中的地址
2. 整数寄存器文件：包含16个命名的位置，分别存储64位的值，这些寄存器可以存储地址(对应于C语言的指针)或整数数据。有的寄存器被用来记录某些重要的程序状态，而其他的寄存器用来保存临时数据，例如过程的参数和局部变量，以及函数的返回值
3. 条件码寄存器：保存着最近执行的算术或逻辑指令的状态信息。它们用来实现控制或数据流中的条件变化，比如说用来实现if和while语句
4. 向量寄存器：一组向量寄存器可以存放一个或多个整数或浮点数值

6、虽然C语言提供了一种模型，可以在内存中声明和分配各种数据类型的对象，但是机器代码只是简单地将内存看成一个很大的、按字节寻址的数组

1. 数组和结构，在机器代码中用一组连续的字节来表示
2. 对于标量数据，汇编代码也不区分有符号和无符号整数，不区分各种类型的指针，甚至不区分指针和整数

7、程序内存包含

1. 程序的可执行机器代码
2. 操作系统需要的一些信息，用来管理过程调用和返回的运行时栈，以及用户分配的内存块(比如malloc库函数分配的)
3. 程序内部用虚拟地址来寻址，在任意给定的时刻，只有有限的一部分虚拟地址被认为是合法的。例如x86-64的虚拟地址由64位的字来表示，在目前的实现中，这些地址的高16位必须设置为0，所以一个地址实际上能够指定的是248即64TB范围内的一个字节。较为典型的程序只会访问几兆字节或者几千兆字节的数据。操作系统负责管理虚拟地址空间，将虚拟地址翻译成实际处理器内存中的物理地址

8、一条指令只执行一个非常基本的操作，例如将放在寄存器中的两个数字相加，在存储器和寄存器之间传送数据，或是条件分支转移到新的指令地址。编译器必须产生这些指令序列，从而实现(像算术表达式求值、循环或过程调用和返回这样的)程序结构

### 代码示例

1、命令

gcc -Og -S mstore.c

2、反汇编器(disassembler)：这些程序根据机器代码产生一种类似于汇编代码的格式

objdump -d mstore.o

3、一些关于机器代码和它的反汇编表示的特性值

1. x86-64的指令长度从1到15个字节不等。常用的指令以及操作数较少的指令所需的字节数较少，那些不太常用或操作数较多的指令所需字节较多
2. 设计指令格式的方式是，从某个给定位置开始，可以将字节唯一地解码城机器指令，例如只有pushq %rbx是以字节值53开头的
3. 反汇编器只是基于及其代码文件中的字节序列来确定汇编代码，他不需要访问该程序的源代码或汇编代码
4. 反汇编器使用的指令命名规则与GCC生成的汇编代码使用的有些细微的差别

4、关于格式的注解

1. GCC产生的汇编代码对我们来说有点难度，它包含了一些我们不需要关心的信息，同时它不提供任何程序的描述或它是如何工作的描述
2. 所有以'.'开头的行都是指导汇编器和链接器工作的伪指令，通常可以忽略

5、ATT和Intel汇编代码格式

1. 我们表述的是ATT(根据"AT&T"命名的，AT&T是运营贝尔实验室多年的公司)格式的汇编代码，这是GCC、OBJDUMP和其他一些我们使用的工具的默认格式
2. 其他一些编程工具，包括Microsoft工具，以及来自Intel的文档，其汇编代码都是Intel格式的
3. Gcc可以产生Intel格式的代码

gcc -Og -S -masm=intel mstore.c

1. Intel和ATT格式的不同之处

* Intel代码省略了指示大小的后缀，例如指令push和mov，而不是pushq和movq
* Intel代码省略了寄存器名字前面的'%'符号，用的是rbx，而不是%rbx
* Intel代码用不同的方式来描述内存中的位置，例如是'QWORD PTR [rbx]'而不是'(%rbx)'
* 在带有多个操作数的指令下，列出操作数的顺序相反，这一点非常令人困惑

6、在C语言中插入汇编代码有两种方法

1. 我们可以编写完整的函数，放进一个独立的汇编代码文件中，让汇编器和链接器把它和用C语言书写的代码合并起来
2. 我们可以使用GCC的内联汇编(inline assembly)特性，用asm伪指令可以在C语言程序中包含简短的汇编代码

## 数据格式

1、由于从16位体系结构扩展成32位，Intel用术语"字(word)"表示16位数据类型(网上的说法好像是字长是可变的，比如32位的字长就是32位)。因此32位数称为"双字(double words)"，称64位数为"四字(quad words)"

2、C语言数据类型在x86-64中的大小，在64位机器中，指针8个字节

|  |  |  |  |
| --- | --- | --- | --- |
| C声明 | Intel数据类型 | 汇编代码后缀 | 大小 |
| char | 字节 | b | 1 |
| short | 字 | w | 2 |
| int | 双字 | l | 4 |
| long | 四字 | q | 8 |
| char\* | 四字 | q | 8 |
| float | 单精度 | s | 4 |
| double | 双精度 | l | 8 |

3、大多数GCC生成的汇编代码指令都有一个字符后缀，表明操作数的大小。例如数据传送指令有四个变种

1. movb：传送字节(byte)
2. movw：传送字(word)
3. movl：传送双字(long word)
4. movq：传送四字(quad words)

## 访问信息

1、一个x86-64的中央处理单元(CPU)包含一组16个存储位值的通用目的寄存器，这些寄存器用来存储整数数据和指针

1. 这16个寄存器名字都以%r开头，后面还跟着一些不同的命名规则的名字，这是由于指令集历史演化造成的
2. 最初8086中有8个16位的寄存器，即%ax-%bp，每个寄存器都有特殊的用途
3. 扩展到IA32架构时，这些寄存器也扩展成32位寄存器，标号从%aex到%ebp
4. 扩展到x86-64后，原来的8个寄存器扩展成64位，标号从%rax-%rbp，除此之外，还增加了8个新的寄存器，他们的标号是按照新的命名规则制定的，从%r8到%r15



2、如上的嵌套结构表明，指令可以对这16个寄存器的低位字节中存放的不同大小的数据进行操作。字节级操作可以访问最低的字节，16位操作可以访问最低的2个字节，32位操作可以访问最低的4个字节，而64位操作可以访问整个寄存器

3、两条规则

1. 生成1字节和2字节数字的指令会保持剩下的字节不变
2. 生成4字节数字的指令会把高位的4个字节置为0
3. 后面这条规则是作为从IA32到x86-64的扩展的一部分而采用的

4、在常见的程序里，不同的寄存器扮演者不同的角色，其中最特别的是%rsp，用来指明运行时栈的结束位置。有些程序会明确地读写这个寄存器。值得注意的是，有一组标准的编程规范控制者如何使用寄存器来管理栈、传递函数参数、从函数的返回值，以及存储局部和临时数据

### 操作数指示符

1、大多数指令有一个或多个操作数(operand)，指示出执行一个操作中要使用的源数据值，以及放置结果的目的位置

2、x86-64支持多种操作数格式，源数据值可以以常数形式给出，或是从寄存器或内存中读出。结果可以放在寄存器或内存中

3、各种不同的操作数的可能性被分为三种类型

1. 第一种类型是立即数(immediate)，用来表示常数值。在ATT格式的汇编代码中，立即数的书写方式是$后跟一个用标准C表示法表示的整数。例如$-557，$0x1F。不同的指令允许的立即数值范围不同，汇编器会自动选择最紧凑的方式进行数值编码
2. 第二种类型是寄存器(register)，它表示某个寄存器的内容，16个寄存器的低位1字节、2字节、4字节或8字节中的一个座位操作数，这些字节数分别对应于8位，16位，32位，64位。我们用符号ra表示任意寄存器a，用引用R[ra]来表示它的值，这是将寄存器集合看成是一个数组R，用寄存器标识符作为索引
3. 第三类操作数是内存引用，它会根据计算出来的地址(通常称为有效地址)访问某个内存位置。因为将内存看成一个很大的字节数组，我们用符号Mb[Addr]表示对存储在内存中从地址Addr开始的b个字节值的的引用。为了方便，我们通常省去下标b

4、有多种不同的寻址模式，允许不同形式的内存引用

* 表中底部用语法*Imm*(rb,ri,s)表示的是最常用的形式。这样的引用有四个组成部分：一个立即数偏移*Imm*，一个基址寄存器rb，一个变址寄存器ri和一个比例因子s，这里s必须是1、2、4或者8，基址和变址寄存器都必须是64位寄存器。有效地址被计算为*Imm*+R[rb]+R[ri]\*s

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 格式 | 操作数值 | 名称 |
| 立即数 | $*Imm* | *Imm* | 立即数寻址 |
| 寄存器 | ra | R[ra] | 寄存器寻址 |
| 存储器 | *Imm* | M[*Imm*] | 绝对寻址 |
| 存储器 | (ra) | M[R[ra]] | 间接寻址 |
| 存储器 | *Imm*(rb) | M[*Imm*+R[rb]] | (基址+偏移量)寻址 |
| 存储器 | (rb,ri) | M[R[rb]+R[ri]] | 变址寻址 |
| 存储器 | *Imm*(rb,ri) | M[*Imm*+R[rb]+R[ri]] | 变址寻址 |
| 存储器 | (,ri,s) | M[R[ri]\*s] | 比例变址寻址 |
| 存储器 | *Imm*(,ri,s) | M[*Imm*+R[ri]\*s] | 比例变址寻址 |
| 存储器 | (rb,ri,s) | M[R[rb]+R[ri]\*s] | 比例变址寻址 |
| 存储器 | *Imm*(rb,ri,s) | M[*Imm*+R[rb]+R[ri]\*s] | 比例变址寻址 |

### 数据传送指令

1、最频繁使用的指令是将数据从一个位置复制到另一个位置的指令。操作数表示的通用性使得一条简单的数据传送指令能够完成在许多机器中要好几条不同指令才能完成的功能

#### MOV类

1、下标是最简单形式的数据传输指令---MOV类

* 这些指令把数据从源位置复制到目的位置，不做任何变化
* MOV类主要由四条指令组成：movb、movw、movl和movq，这些指令都执行相同的操作，主要区别在于它们操作的数据大小不同，分别是1、2、4、8字节

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOV S,D | D<--S | 传送 |
| movb | R<--I | 传送字节 |
| movw | 传送字 |
| movl | 传送双字 |
| movq | 传送四字 |
| movabsq I,R | 传送绝对的四字 |

2、规则

1. 源操作数指定的是一个立即数，存储在寄存器中或者内存中
2. 目的操作数指定一个位置，要么是一个寄存器，要么是一个内存地址
3. x86-64增加了一条限制，传送指令的两个操作数不能都指向内存位置，将一个值从一个内存位置复制到另一个内存位置需要两条指令

* 第一条指令将原值加载到寄存器中
* 第二条指令将该寄存器值写入目的的位置

1. 这些指令的寄存器操作数可以是16个寄存器有标号部分的任意一个，寄存器部分的大小必须与指令最后一个字符('b','w','l','q')指定的大小匹配
2. 大多数情况中，MOV指令只会更新目的操作数指定的那些寄存器字节或内存位置。唯一的例外是movl指令以寄存器作为目的时，它会把该寄存器的最高4位字节设置为0，造成这个例外的原因是x86-64的惯例，即任何为寄存器生成32位值的指令都会把该寄存器的高位部分置0

3、示例

movl $0x4050,%eax Immediate-->Register,4bytes

movw %bp,%sp Register-->Register,2bytes

movb (%rdi,%rcx),%al Memory-->Register,1byte

movb $-17,(%rsp) Immediate-->Memory,1byte

movq %rax,-12(%rbp) Register-->Memory,8bytes

4、常规的movq指令只能以表示为32位补码数字的立即数作为源操作数，然后把这个值符号扩展得到64位的值，放到目的位置。movabsp指令能够以任意64位立即数值作为源操作数，并且只能以寄存器作为目的

#### MOVZ类

1、MOVZ类指令把数据从源(寄存器或内存)复制到目的寄存器

2、MOVZ类中的指令把目的中剩余的字节填充为0

3、每条指令名字最后两个字符都是大小指示符：

1. 第一个字符指定源的大小
2. 第二个字符指定目的大小
3. 满足源大小<目的大小

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOVZ S,R | R<--零扩展(S) | 以零扩展进行传送 |
| movzbw |  | 将做了零扩展的字节传送到字 |
| movzbl | 将做了零扩展的字节传送到双字 |
| movzwl | 将做了零扩展的字传送到双字 |
| movzbq | 将做了零扩展的字节传送到四字 |
| movzwq | 将做了零扩展的字传送到四字 |

4、注意，只有两种较小的源类型(字节，字)有零扩展传送，但是双字并没有零扩展传送，即并没有movzlq，但是将做了零扩展的双字传送到四字可以通过movl指令来实现(特殊规则)

#### MOVS类

1、MOVS类指令把数据从源(寄存器或内存)复制到目的寄存器

2、MOVS类中的指令把目的中剩余的字节通过符号扩展来填充，把源操作的最高位进行复制

3、每条指令名字最后两个字符都是大小指示符：

1. 第一个字符指定源的大小
2. 第二个字符指定目的大小
3. 满足源大小<目的大小

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| MOVS S,R | R<--符号扩展(S) | 以零扩展进行传送 |
| movsbw |  | 将做了符号扩展的字节传送到字 |
| movsbl | 将做了符号扩展的字节传送到双字 |
| movswl | 将做了符号扩展的字传送到双字 |
| movsbq | 将做了符号扩展的字节传送到四字 |
| movswq | 将做了符号扩展的字传送到四字 |
| movslq | 将做了符号扩展的双字传送到四字 |
| cltq | %rax<--符号扩展(%eax) | 把%eax符号扩展到%rax |

4、所有三种源类型(字节，字，双字)都有对应的符号扩展传送

5、cltq指令与movslq %eax,%rax完全一致，不过编码更紧凑

### 数据传送示例

1、C代码

long exchange(long \*xp,long y){

long x=\*xp;

\*xp=y;

return x;

}

*long exchange(long \*xp, long y)*

*xp in %rdi, y in %rsi*

exchange:

movq (%rdi),%rax *Get x at xp，Set as return value*

movq %rsi,(%rdi) *Store y at xp*

ret *Return*

2、注意点

1. C语言中的指针其实就是地址，间接引用指针就是将该指针放在一个寄存器中，然后在内存引用中使用这个寄存器
2. 像x这样的局部变量通常保存在寄存器中，而不是内存中

### 压入和弹出栈数据

1、最后两个数据传送操作可以将数据压入程序栈中，以及从程序栈中弹出数据

2、栈在处理过程中起到至关重要的作用

* 栈是一种数据结构，可以添加或删除值，不过要遵循"先进后出"的原则。通过push操作把数据压入栈中，通过pop操作删除数据
* 弹出的值永远是最近被压入而且仍然在栈中的值
* 栈可以实现为一个数组，总是从数组的一端插入和删除元素，这一端被称为栈顶

3、pushq指令的功能是把数据压入到栈上，而popq指令是弹出数据。这些指令只有一个操作数---压入的数据源和弹出的数据数目

4、将一个四字值压入栈中，首先要将栈指针减8，然后将值写到新的栈顶地址。因此pushq %rbp的行为等价于下面两条指令

subq $8,%rsp

movq %rbp,(%rsp)

5、弹出一个四字操作包括从栈顶位置读出数据，然后将栈指针加8。因此，指令popq %rax等价于下面两条指令

movq (%rsp),%rax

addq %8,%rsp

6、因为栈和程序代码以及其他形式的程序数据都是放在同一内存中，所以程序可以用标准的内存寻址方法访问栈内的任意位置。假设栈顶元素是四字，指令movq 8(%rsp),%rdx会将第二个四字从栈中复制到寄存器%rdx

## 算数和逻辑操作

1、下表列出了x86-64的一些整数和逻辑操作。大多数操作都分成了指令类，即这些指令类有各种带不同大小操作数的变种(只有leaq没有其他大小的变种)。指令类ADD由四条加法指令组成：addb、addw、addl和addq，分别是字节加法、字加法、双字加法和四字加法

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| leaq S,D | D<--&S | 加载有效地址 |
| INC D  DEC D  NEG D  NOT D | D<--D+1  D<--D-1  D<-- -D  D<-- ~D | 加1  减1  取负  取补 |
| ADD S,D  SUB S,D  IMUL S,D  XOR S,D  OR S,D  AND S,D | D<-- D+S  D<-- D-S  D<-- D\*S  D<-- D^S  D<--D|S  D<-- D&S | 加  减  乘  异或  或  与 |
| SAL k,D  SHL k,D  SAR k,D  SHR k,D | D<--D<<k  D<--D<<k  D<--D>>Ak  D<--D>>Lk | 左移  左移(等同于SAL)  算数右移  逻辑右移 |

### 加载有效地址

1、加载有效地址(load effective address)指令leaq实际上是movq指令的变形

* 它的指令形式是从内存读取数据到寄存器，但实际上它根本就没有引用内存
* 它的第一个操作数看上去是一个内存引用，但该指令并不是从指定的位置读取数据，而是将有效地址写入到目的操作数。例如如果寄存器%rdx的值为x，那么指令leaq 7(%rdx,%rdx,4),%rax将设置寄存器%rax的值为5x+7

2、leaq的灵活用法：leaq指令能执行加法和有限形式的乘法，在编译如下算数表达式时，很有用处

long scale(long x,long y,long z){

long t=x+4\*y+12\*z;

return t;

}

*long scale(long x,long y,long z)*

*x in %rdi, y in %rsi, z in %rdx*

scale:

leaq (%rdi,%rsi,4),%rax *x+4\*y*

leaq (%rdx,%rdx,2),%rdx *z+2\*z=3z*

leaq (%rax,%rdx,4),%rax *(x+4\*y)+4\*(3\*z)=x+4\*y+12\*z*

ret

### 一元和二元操作

1、第二组中的操作是一元操作，只有一个操作数，既是源又是目的。这个操作数可以是一个寄存器，也可以是一个内存位置。例如指令incq(%rsp)会使栈顶的8字节元素加1，这种语法让人想起C语言中的自增运算符和自减运算符

2、第三组是二元操作，其中第二个操作数既是源又是目的，这种语法让人想起C语言中的符合赋值运算符(+=、-=、\*=、/=、^=、|=、&=)

* 源操作数是第一个，目的操作数是第二个
* 例如指令subq %rax,%rdx使寄存器%rdx的值减去%rax中的值
* 第一个操作数可以使立即数，寄存器或是内存位置
* 第二个操作数可以是寄存器或内存位置。当第二个操作数为内存地址时，处理器必须从内存读出值，执行操作，再把结果写回内存

### 移位操作

1、最后一组是移位操作，先给出移位量，然后第二项给出的是要移位的数，可以进行算数和逻辑右移

2、移位量可以是一个立即数，或者放在单字节寄存器%cl中(这些指令很特别，因为只允许以这个特定的寄存器作为操作数)

* 原则上来说，1个字节的位移量使得移位量的编码范围可以达到28-1=255
* x86-64中，移位操作对w位长的数据进行操作，移位量是由%cl寄存器的低m位决定的，满足2m=w，高位会被忽略。例如当寄存器%cl的十六进制值为0xFF时，指令salb移动7位(23=8，111=7)，salw会移动15位(24=16，1111=15)，sall会移动31位(25=32，11111=31)，salq会移动63位(26=64，111111=63)

3、左移指令有两个名字：SAL和SHL。两者效果一样，都是将右边填上0

4、右移指令不同，SAL执行算数移位(填上符号位)，而SHR执行逻辑移位(填上0)。移位操作的目的操作数可以是一个寄存器或是一个内存位置

### 讨论

1、对于大多数指令，既可以用于无符号运算，也可以用于补码运算，只有右移操作需要区分有符号和无符号数，这个特性使补码运算成为实现有符号整数运算的一种比较好的方法的原因之一

### 特殊的算数操作

1、两个64位有符号或无符号整数相乘得到的乘积需要128位来表示。x86-64指令集对128位(16字节)数的操作提供有限的支持。延续字(2字节)、双字(4字节)、四字(8字节)的命名管理，Intel把16字节的数称为八字(oct word)

2、下表描述的是支持产生两个64位数字的全128位乘积以及整数除法指令

|  |  |  |
| --- | --- | --- |
| 指令 | 效果 | 描述 |
| imulq S  mulq S | R[%rdx]：R[%rax]<--SxR[%rax]  R[%rdx]：R[%rax]<--SxR[%rax] | 有符号全乘法  无符号全乘法 |
| clto | R[%rdx]：R[%rax]<--符号扩展(R[%rax]) | 转换为八字 |
| idivq S | R[%rdx]<--R[%rdx]：R[%rax] mod S  R[%rdx]<--R[%rdx]：R[%rax] ÷ S | 有符号除法 |
| divq S | R[%rdx]<--R[%rdx]：R[%rax] mod S  R[%rdx]<--R[%rdx]：R[%rax] ÷ S | 无符号除法 |

3、乘法指令有两种不同的形式

* 双操作数乘法指令
* 单操作数乘法指令，mulq和imulq，这两条指令都要求一个参数必须放在寄存器%rax中，而另一个作为指令的源操作数给出。乘积结果放在寄存器%rdx(高位)和%rax(低位)中

4、除法或取模操作是由单操作数除法指令来提供的

* 类似于单操作数乘法指令，有符号数除法指令idivl将寄存器%rdx(高64位)和%rax(低64位)中的128位数作为被除数，而除数作为指令的操作数给出
* 指令将商存在寄存器%rax中，将余数存在寄存器%rdx中
* 对于大多数64位除法应用来说，被除数也常常是一个64位的值，这个值应该存放在%rax中，%rdx的位应该设置为全0(无符号运算)或者%rax的符号位(有符号运算)。这个操作可以用cqto来完成，这条指令不需要操作数---它隐含读出%rax的符号位，并将它复制到%rdx的所有位

## 控制

1、到目前为止，我们只考虑了直线代码的行为，也就是指令一条接着一条顺序地执行。C语言中的某些结构，比如条件语句、循环语句和分支语句，要求有条件的执行，根据数据测试的结果来决定操作执行的顺序。机器代码提供两种基本的低级机制来实现由条件的行为：测试数据值，然后根据测试的结果来改变控制流或者数据流

2、与数据相关的控制流是实现有条件行为的更一般和常见的方法。通常，C语言中的语句和机器代码中指令都是按照它们在程序中出现的次序，顺序执行的。用jump指令可以改变一组既期待吗指令的执行顺序，jump指令指定控制应该被传递到程序的某个其他部分，可能是依赖于某个测试的结果。编译器必须产生构建在这种低级机制基础之上的指令序列，来实现C语言的控制结构

### 条件码

1、除了整数寄存器，CPU还维护着一组单个位的条件码(condition code)寄存器，它们描述了最近的算数或逻辑操作的属性，可以检测这些寄存器来执行条件分支指令

2、下面给出常用条件码

1. CF：进位标志。最近的操作使最高位产生了进位，可用来检查无符号操作的溢出
2. ZF：零标志。最近的操作得出结果为0
3. SF：符号标志。最近的操作得到的结果为负数
4. OF：溢出标志。最近的操作导致一个补码溢出(正溢出或负溢出)

3、leaq指令不改变任何条件码，因为它是用来进行地址计算的。除此之外，素有算数和逻辑操作指令都会设置条件码

4、还有两类指令，它们只设置条件码而不改变任何其他寄存器

|  |  |  |
| --- | --- | --- |
| 指令 | 基于 | 描述 |
| CMP S1,S2  cmpb  cmpw  cmpl  cmpq  TEST S1,S2  testb  testw  testl  testq | S2-S1  S1&S2 | 比较  比较字节  比较字  比较双字  比较四字  测试  测试字节  测试字  测试双字  测试四字 |

* CMP指令根据两个操作数之差来设置条件码，除了只设置条件码而不更新目的寄存器之外，CMP指令与SUB指令的行为是一样的。在ATT格式中，列出操作数的顺序是相反的，例如CMP S1,S2是比较(S2-S1)的结果。如果两个操作数相等，该类指令会将零标志位置为1，其他标志可以用来确定两个操作数的大小关系
* TEST指令的行为与AND指令一样，除了它们只设置条件码而不改变目的寄存器的值。典型的用法是，两个操作数是样的(例如testq %rax,%rax用来检查%rax是负数、零还是正数)，或者其中一个操作数是一个掩码，用来指示哪些位应该被测试

### 访问条件码

1、条件码通常不会直接读取，常用的方法有三种

1. 可以根据条件码的某种组合，将一个字节设置为0或者1
2. 可以条件跳转到程序的某个其他部分
3. 可以有条件地传送数据

2、将一个字节设置为0或1，这类指令称为SET指令

* 它们之间的区别就在于它们考虑的条件码的组合是什么，这些指令名字的不同后缀指明了它们所考虑的条件码的组合
* 这些指令的后缀表示不同的条件而不是操作数大小。例如setl和setb表示"小于时设置(set less)"和"低于时设置(set below)"，而不是"设置长字(set long word)"和"设置字节(set byte)"

|  |  |  |  |
| --- | --- | --- | --- |
| 指令 | 同义名 | 效果 | 设置条件 |
| sete D  setne D  sets D  setns D  setg D  setge D  setl D  setle D  seta D  setae D  setb D  setbe D | setz  setnz  setnle  setnl  setnge  setng  setnbe  setnb  setnae  setna | D<--ZF  D<--~ZF  D<--SF  D<--~SF  D<--~(SF^OF)&~ZF  D<--~(SF^OF)  D<--SF^OF  D<(SF^OF)|ZF  D<--~CF&~ZF  D<--~CF  D<--CF  D<--CF|ZF | 相等/零  不等/非零  负数  非负数  大于(有符号>)  大于等于(有符号>=)  小于(有符号<)  小于等于(有符号<=)  超过(无符号>)  超过或相等(无符号>=)  低于(无符号<)  低于或相等(无符号<=) |

* g：greater
* e：equal
* l：less
* a：???
* b：below
* 某些底层的机器指令可能有多个名字，称之为"同义名(synonym)"，编译器和反汇编器会随意决定使用哪个名字

3、虽然所有的算数和逻辑操作都会设置条件码，但是各个SET命令的描述都适用的情况是：执行比较指令，根据计算t=a-b设置条件码。更具体地说，假设a、b和t分别是变量a、b和t的补码形式表示的整数，因此t=a-twb，这里w取决于a和b的大小

1. sete：当相等时设置指令。当a=b时，t=0，因此零标志位就表示相等
2. setl：当小于时设置

* 当没有发生溢出时(OF为0)，t<0时SF设置为1，t>=0时SF设置为0
* 当发生溢出时(OF为1)，t>0(正溢出，此时a<b)时SF设置为0，t<0(负溢出，此时a>b)时SF设置为1。t不可能为0，因为发生了溢出
* 因此SF^OF提供了真测试

4、对于无符号比较的测试，现在设a和b是变量a和b的无符号形式表示的整数，在执行t=a-b中

* 当a-b<0时，CMP指令会设置进位标志，因而无符号比较使用的是进位标志和零标志的组合

5、注意到机器代码如何区分有符号和无符号值是很重要的。与C语言不同，机器代码不会将程序值都和一个数据类型联系起来。相反，大多数情况下，机器代码对于有符号和无符号两种情况使用一样的指令，这是因为许多算数运算对无符号和补码算数都有一样的位级行为

# IO

## 文件描述符

1、POSIX标准

* POSIX表示可移植操作系统接口(Portable Operating System Interface of UNIX，缩写为 POSIX)，POSIX标准定义了操作系统应该为应用程序提供的接口标准，是IEEE为要在各种UNIX操作系统上运行的软件而定义的一系列API标准的总称
* POSIX标准意在期望获得源代码级别的软件可移植性。换句话说，为一个POSIX兼容的操作系统编写的程序，应该可以在任何其它的POSIX操作系统(即使是来自另一个厂商)上编译执行

2、文件描述符(file descriptor，缩写为fd)在Linux编程里随处可见，设备读写、网络通信、进程通信，fd可谓是关键中的关键

3、先看三段简化后的内核代码

**sys\_open**：

...

fd = get\_unused\_fd();

if (fd >= 0) {

struct file \*f = filp\_open(tmp, flags, mode);

fd\_install(fd, f);

}

...

**sys\_socket**：

...

fd = get\_unused\_fd();

if (fd >= 0) {

struct file \*file = get\_empty\_filp();

fd\_install(fd, file)

}

...

**do\_pipe**：

...

struct file \*f1 = get\_empty\_filp();

struct file \*f2 = get\_emtpy\_filp();

i = get\_unused\_fd();

j = get\_unused\_fd();

fd\_install(i, f1);

fd\_install(j,f2);

fd[0] = i;

fd[1] = j;

...

* 这三段代码分别是三个POSIX标准系统调用open，socket，pipe的内核态例程，简化后的代码可以清楚看到，文件描述符的获取和安装在不同模块中都有着几乎相同的套路

1. get\_unused\_fd顾名思义，它从当前进程描述符中的打开文件数组(current->files)里取得一个空闲的项，然后返回其数组下标
2. filp\_open最终将调用get\_empty\_filp，因此三段代码都使用get\_empty\_filp分配了一个新的文件对象(struct file)
3. fd\_install将上一个步骤创建的文件对象指针存到该进程的打开文件数组中

4、至此，文件描述符安装完毕，返回的fd即为该文件对象在当前进程的task\_struct中的files数组中的下标，也就是所谓的文件描述符，但究其本质，我们对文件描述符的所有使用其实就是在操作file对象

5、接下来就是程序员最拿手的read，write，以及各种IO控制了。

6、file作为一个通用的结构体，在不同模块表示的对象是不同的，在文件系统中，它表示为一个文件，在进程通信中，它表示为一个管道，在网络通信中，它又表示为一个套接字

7、那么问题来了，作为一个通用的结构体，在表示为不同的通信对象的时候，操作函数肯定是不同的，举个例子来说，write一个文件和write一个socket肯定是不一样的，那么同一个函数入口怎么做到多态的实现呢？

8、我们同样用write来解释，write()->sys\_write()->vfs\_write()->file->f\_op->write()，重点在write调用流程的最后一步file->f\_op->write()，从这里开始，针对不同的file对象将进入不同的write实现。这是因为不同file里注册的各类操作函数是不同的。

9、struct file中的f\_op指针，指向一系列函数指针的集合，这个结构名叫file\_operation，这一系列函数指针都指向着当前这个file对象(设备也好，管道也好，套接字也好)对应的各种操作函数的具体实现

* 在2.6.11的内核里看起来如下

struct file\_operations {

struct module \*owner;

loff\_t (\*llseek) (struct file \*, loff\_t, int);

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

...

}

10、所以，一个file是文件、是设备、是管道还是套接字，就是根据其中存放的file\_operations来区分，在创建file的时候就会针对这个file的类型注册不同的操作函数，这就解释了同一个通用函数的多态实现

11、文件描述符的生命周期：

1. 创建file对象
2. 根据IO对象的类型注册各个操作函数(注册file\_operations)
3. 将file对象的指针注册到进程描述符的files数组里的fd下标处
4. read、write等等IO操作
5. 调用close(fd)释放file对象

## 文件描述符与打开文件的关系

### 简介

1、在Linux系统中一切皆可以看成是文件，文件又可分为：普通文件、目录文件、链接文件和设备文件。文件描述符(file descriptor)是内核为了高效管理已被打开的文件所创建的索引，其是一个非负整数(通常是小整数)，用于指代被打开的文件，所有执行I/O操作的系统调用都通过文件描述符。程序刚刚启动的时候，0是标准输入，1是标准输出，2是标准错误。如果此时去打开一个新的文件，它的文件描述符会是3。POSIX标准要求每次打开文件时(含socket)必须使用当前进程中最小可用的文件描述符号码，因此，在网络通信过程中稍不注意就有可能造成串话。标准文件描述符图如下：



2、文件描述与打开的文件对应模型如下图



### 文件描述符的限制

1、在编写文件操作的或者网络通信的软件时，初学者一般可能会遇到"Too many open files"的问题。这主要是因为文件描述符是系统的一个重要资源，虽然说系统内存有多少就可以打开多少的文件描述符，但是在实际实现过程中内核是会做相应的处理的，一般最大打开文件数会是系统内存的10%(以KB来计算)(称之为系统级限制)，查看系统级别的最大打开文件数可以使用sysctl -a | grep fs.file-max命令查看。与此同时，内核为了不让某一个进程消耗掉所有的文件资源，其也会对单个进程最大打开文件数做默认值处理(称之为用户级限制)，默认值一般是1024，使用ulimit -n命令可以查看。在Web服务器中，通过更改系统默认值文件描述符的最大值来优化服务器是最常见的方式之一

### 文件描述符和打开文件之间的关系

1、每一个文件描述符会与一个打开文件相对应，同时，不同的文件描述符也会指向同一个文件。相同的文件可以被不同的进程打开也可以在同一个进程中被多次打开。系统为每一个进程维护了一个文件描述符表，该表的值都是从0开始的，所以在不同的进程中你会看到相同的文件描述符，这种情况下相同文件描述符有可能指向同一个文件，也有可能指向不同的文件。具体情况要具体分析，要理解具体其概况如何，需要查看由内核维护的3个数据结构

1. 进程级的文件描述符表
2. 系统级的打开文件描述符表
3. 文件系统的i-node表

2、进程级的描述符表的每一条目记录了单个文件描述符的相关信息

1. 控制文件描述符操作的一组标志(目前，此类标志仅定义了一个，即close-on-exec标志)
2. 对打开文件句柄的引用

3、内核对所有打开的文件的文件维护有一个系统级的描述符表格(open file description table)。有时，也称之为打开文件表(open file table)，并将表格中各条目称为打开文件句柄(open file handle)。一个打开文件句柄存储了与一个打开文件相关的全部信息，如下所示：

1. 当前文件偏移量(调用read()和write()时更新，或使用lseek()直接修改)
2. 打开文件时所使用的状态标识(即，open()的flags参数)
3. 文件访问模式(如调用open()时所设置的只读模式、只写模式或读写模式)
4. 与信号驱动相关的设置
5. 对该文件i-node对象的引用
6. 文件类型(例如：常规文件、套接字或FIFO)和访问权限
7. 一个指针，指向该文件所持有的锁列表
8. 文件的各种属性，包括文件大小以及与不同类型操作相关的时间戳

4、下图展示了文件描述符、打开的文件句柄以及i-node之间的关系，图中，两个进程拥有诸多打开的文件描述符



* 在进程A中，文件描述符1和30都指向了同一个打开的文件句柄(标号23)。这可能是通过调用dup()、dup2()、fcntl()或者对同一个文件多次调用了open()函数而形成的
* 进程A的文件描述符2和进程B的文件描述符2都指向了同一个打开的文件句柄(标号73)。这种情形可能是在调用fork()后出现的(即，进程A、B是父子进程关系)，或者当某进程通过UNIX域套接字将一个打开的文件描述符传递给另一个进程时，也会发生。再者是不同的进程独自去调用open函数打开了同一个文件，此时进程内部的描述符正好分配到与其他进程打开该文件的描述符一样
* 此外，进程A的描述符0和进程B的描述符3分别指向不同的打开文件句柄，但这些句柄均指向i-node表的相同条目(1976)，换言之，指向同一个文件。发生这种情况是因为每个进程各自对同一个文件发起了open()调用。同一个进程两次打开同一个文件，也会发生类似情况

### 总结

1、由于进程级文件描述符表的存在，不同的进程中会出现相同的文件描述符，它们可能指向同一个文件，也可能指向不同的文件

2、两个不同的文件描述符，若指向同一个打开文件句柄，将共享同一文件偏移量。因此，如果通过其中一个文件描述符来修改文件偏移量（由调用read()、write()或lseek()所致），那么从另一个描述符中也会观察到变化，无论这两个文件描述符是否属于不同进程，还是同一个进程，情况都是如此

3、要获取和修改打开的文件标志(例如：O\_APPEND、O\_NONBLOCK和O\_ASYNC)，可执行fcntl()的F\_GETFL和F\_SETFL操作，其对作用域的约束与上一条颇为类似

4、文件描述符标志(即，close-on-exec)为进程和文件描述符所私有。对这一标志的修改将不会影响同一进程或不同进程中的其他文件描述符

## 阻塞与非阻塞，同步与异步，I/O模型

### 概念理解

1、同步/异步主要针对Client端

* 同步：
* 所谓同步，就是在c端发出一个功能调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做，等前一件做完了才能做下一件事
* 例如普通B/S模式(同步)：提交请求->等待服务器处理->处理完毕返回 这个期间客户端浏览器不能干任何事
* 异步：
* 异步的概念和同步相对。当c端一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者
* 例如ajax请求(异步)：请求通过事件触发->服务器处理(这时浏览器仍然可以作其他事情)->处理完毕

2、阻塞/非阻塞主要针对S端

* 阻塞
* 阻塞调用是指调用结果返回之前，当前线程会被挂起(线程进入非可执行状态，在这个状态下，cpu不会给线程分配时间片，即线程暂停运行)，函数只有在得到结果之后才会返回
* 有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。例如，我们在socket中调用recv函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消息
* 快递的例子：比如到你某个时候到A楼一层(假如是内核缓冲区)取快递，但是你不知道快递什么时候过来，你又不能干别的事，只能死等着。但你可以睡觉(进程处于休眠状态)，因为你知道快递把货送来时一定会给你打个电话(假定一定能叫醒你)
* 非阻塞
* 非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回
* 还是等快递的例子：如果用忙轮询的方法，每隔5分钟到A楼一层(内核缓冲区)去看快递来了没有。如果没来，立即返回。而快递来了，就放在A楼一层，等你去取

3、对象的阻塞模式和阻塞函数调用

* 对象是否处于阻塞模式和函数是不是阻塞调用有很强的相关性，但是并不是一一对应的。阻塞对象上可以有非阻塞的调用方式，我们可以通过一定的API去轮询状态，在适当的时候调用阻塞函数，就可以避免阻塞。而对于非阻塞对象，调用特殊的函数也可以进入阻塞调用。函数select就是这样的一个例子

1. 同步，就是我客户端(c端调用者)调用一个功能，该功能没有结束前，我(c端调用者)死等结果
2. 异步，就是我(c端调用者)调用一个功能，不需要知道该功能结果，该功能有结果后通知我(c端调用者)即回调通知
3. 同步/异步主要针对C端，但是跟S端不是完全没有关系，同步/异步机制必须S端配合才能实现。同步/异步是由c端自己控制，但是S端是否阻塞/非阻塞，C端完全不需要关心
4. 阻塞，就是调用我(s端，被调用者，函数)，我(s端，被调用者，函数)没有接收完数据或者没有得到结果之前，我不会返回
5. 非阻塞，就是调用我(s端，被调用者，函数)，我(s端，被调用者，函数)立即返回，通过select通知调用者

4、同步IO和异步IO的区别就在于：数据访问的时候进程是否阻塞！

5、阻塞IO和非阻塞IO的区别就在于：应用程序的调用是否立即返回！

6、阻塞和非阻塞是指当server端的进程访问的数据如果尚未就绪，进程是否需要等待，简单说这相当于函数内部的实现区别，也就是未就绪时是直接返回还是等待就绪。而同步和异步是指client端访问数据的机制，同步一般指主动请求并等待I/O操作完毕的方式，当数据就绪后在读写的时候必须阻塞(区别就绪与读写二个阶段，同步的读写必须阻塞)，异步则指主动请求数据后便可以继续处理其它任务，随后等待I/O，操作完毕的通知，这可以使进程在数据读写时也不阻塞。(等待"通知")

### Linux下的五种I/O模型

1、阻塞I/O(blocking I/O)

2、非阻塞I/O(nonblocking I/O)

3、I/O复用(select 和poll)(I/O multiplexing)

4、信号驱动I/O(signal driven I/O (SIGIO))

5、异步I/O(asynchronous I/O (the POSIX aio\_functions))

前四种都是同步，最后一种才是异步I/O

#### 阻塞I/O模型

1、进程会一直阻塞，直到数据拷贝完成

2、应用程序调用一个IO函数，导致应用程序阻塞，等待数据准备好。如果数据没有准备好，一直等待。数据准备好了，从内核拷贝到用户空间，IO函数返回成功指示

3、我们第一次接触到的网络编程都是从listen()、send()、recv()等接口开始的。使用这些接口可以很方便的构建服务器/客户机的模型

4、阻塞I/O模型如下图：在调用recv()/recvfrom()函数时，发生在内核中等待数据和复制数据的过程



* 当调用recv()函数时，系统首先查是否有准备好的数据。如果数据没有准备好，那么系统就处于等待状态。当数据准备好后，将数据从系统缓冲区复制到用户空间，然后该函数返回。在套接应用程序中，当调用recv()函数时，未必用户空间就已经存在数据，那么此时recv()函数就会处于等待状态
* 当使用socket()函数和WSASocket()函数创建套接字时，默认的套接字都是阻塞的。这意味着当调用Windows Sockets API不能立即完成时，线程处于等待状态，直到操作完成
* 并不是所有Windows Sockets API以阻塞套接字为参数调用都会发生阻塞。例如，以阻塞模式的套接字为参数调用bind()、listen()函数时，函数会立即返回。将可能阻塞套接字的Windows Sockets API调用分为以下四种

1. 输入操作： recv()、recvfrom()、WSARecv()和WSARecvfrom()函数。以阻塞套接字为参数调用该函数接收数据。如果此时套接字缓冲区内没有数据可读，则调用线程在数据到来前一直睡眠
2. 输出操作： send()、sendto()、WSASend()和WSASendto()函数。以阻塞套接字为参数调用该函数发送数据。如果套接字缓冲区没有可用空间，线程会一直睡眠，直到有空间
3. 接受连接：accept()和WSAAcept()函数。以阻塞套接字为参数调用该函数，等待接受对方的连接请求。如果此时没有连接请求，线程就会进入睡眠状态
4. 外出连接：connect()和WSAConnect()函数。对于TCP连接，客户端以阻塞套接字为参数，调用该函数向服务器发起连接。该函数在收到服务器的应答前，不会返回。这意味着TCP连接总会等待至少到服务器的一次往返时间。

5、使用阻塞模式的套接字，开发网络程序比较简单，容易实现。当希望能够立即发送和接收数据，且处理的套接字数量比较少的情况下，使用阻塞模式来开发网络程序比较合适

6、阻塞模式套接字的不足表现为，在大量建立好的套接字线程之间进行通信时比较困难。当使用"生产者-消费者"模型开发网络程序时，为每个套接字都分别分配一个读线程、一个处理数据线程和一个用于同步的事件，那么这样无疑加大系统的开销。其最大的缺点是当希望同时处理大量套接字时，将无从下手，其扩展性很差

7、阻塞模式给网络编程带来了一个很大的问题，如在调用send()的同时，线程将被阻塞，在此期间，线程将无法执行任何运算或响应任何的网络请求。这给多客户机、多业务逻辑的网络编程带来了挑战。这时，我们可能会选择多线程的方式来解决这个问题

8、应对多客户机的网络应用，最简单的解决方式是在服务器端使用多线程(或多进程)。多线程(或多进程)的目的是让每个连接都拥有独立的线程(或进程)，这样任何一个连接的阻塞都不会影响其他的连接。具体使用多进程还是多线程，并没有一个特定的模式

* 传统意义上，进程的开销要远远大于线程，所以，如果需要同时为较多的客户机提供服务，则不推荐使用多进程
* 如果单个服务执行体需要消耗较多的CPU资源，譬如需要进行大规模或长时间的数据运算或文件访问，则进程较为安全
* 通常，使用pthread\_create()创建新线程，fork()创建新进程

9、多线程/进程服务器同时为多个客户机提供应答服务。模型如下



* 主线程持续等待客户端的连接请求，如果有连接，则创建新线程，并在新线程中提供为前例同样的问答服务
* 上述多线程的服务器模型似乎完美的解决了为多个客户机提供问答服务的要求，但其实并不尽然。如果要同时响应成百上千路的连接请求，则无论多线程还是多进程都会严重占据系统资源，降低系统对外界响应效率，而线程与进程本身也更容易进入假死状态
* 由此可能会考虑使用"线程池"或"连接池"。
* "线程池"旨在减少创建和销毁线程的频率，其维持一定合理数量的线程，并让空闲的线程重新承担新的执行任务
* "连接池"维持连接的缓存池，尽量重用已有的连接、减少创建和关闭连接的频率
* 这两种技术都可以很好的降低系统开销，都被广泛应用很多大型系统，如apache，MySQL数据库等。
* 但是，"线程池"和"连接池"技术也只是在一定程度上缓解了频繁调用IO 接口带来的资源占用。而且，所谓"池"始终有其上限，当请求大大超过上限时，"池"构成的系统对外界的响应并不比没有池的时候效果好多少。所以使用"池"必须考虑其面临的响应规模，并根据响应规模调整"池"的大小
* 对应上例中的所面临的可能同时出现的上千甚至上万次的客户端请求，"线程池"或"连接池"或许可以缓解部分压力，但是不能解决所有问题

#### 非阻塞IO模型

1、非阻塞IO通过进程反复调用IO函数(多次系统调用，并马上返回)。但是，在数据拷贝的过程中，进程是阻塞的

2、我们把一个SOCKET接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间

3、把SOCKET设置为非阻塞模式，即通知系统内核：在调用Windows Sockets API时，不要让线程睡眠，而应该让函数立即返回。在返回时，该函数返回一个错误代码。图所示，一个非阻塞模式套接字多次调用recv()函数的过程。前三次调用recv()函数时，内核数据还没有准备好。因此，该函数立即返回WSAEWOULDBLOCK错误代码。第四次调用recv()函数时，数据已经准备好，被复制到应用程序的缓冲区中，recv()函数返回成功指示，应用程序开始处理数据



* 当使用socket()函数和WSASocket()函数创建套接字时，默认都是阻塞的。在创建套接字之后，通过调用ioctlsocket()函数，将该套接字设置为非阻塞模式。Linux下的函数是：fcntl()
* 套接字设置为非阻塞模式后，在调用Windows Sockets API函数时，调用函数会立即返回。大多数情况下，这些函数调用都会调用"失败"，并返回WSAEWOULDBLOCK错误代码。说明请求的操作在调用期间内没有时间完成。通常，应用程序需要重复调用该函数，直到获得成功返回代码
* 需要说明的是并非所有的Windows Sockets API在非阻塞模式下调用，都会返回WSAEWOULDBLOCK错误。例如，以非阻塞模式的套接字为参数调用bind()函数时，就不会返回该错误代码。当然，在调用WSAStartup()函数时更不会返回该错误代码，因为该函数是应用程序第一调用的函数，当然不会返回这样的错误代码
* 要将套接字设置为非阻塞模式，除了使用ioctlsocket()函数之外，还可以使用WSAAsyncselect()和WSAEventselect()函数。当调用该函数时，套接字会自动地设置为非阻塞方式
* 由于使用非阻塞套接字在调用函数时，会经常返回WSAEWOULDBLOCK错误。所以在任何时候，都应仔细检查返回代码并作好对"失败"的准备。应用程序连续不断地调用这个函数，直到它返回成功指示为止。上面的程序清单中，在While循环体内不断地调用recv()函数，以读入1024个字节的数据。这种做法很浪费系统资源
* 要完成这样的操作，有人使用MSG\_PEEK标志调用recv()函数查看缓冲区中是否有数据可读。同样，这种方法也不好。因为该做法对系统造成的开销是很大的，并且应用程序至少要调用recv()函数两次，才能实际地读入数据。较好的做法是，使用套接字的"I/O模型"来判断非阻塞套接字是否可读可写
* 非阻塞模式套接字与阻塞模式套接字相比，不容易使用。使用非阻塞模式套接字，需要编写更多的代码，以便在每个Windows Sockets API函数调用中，对收到的WSAEWOULDBLOCK错误进行处理。因此，非阻塞套接字便显得有些难于使用
* 但是，非阻塞套接字在控制建立的多个连接，在数据的收发量不均，时间不定时，明显具有优势。这种套接字在使用上存在一定难度，但只要排除了这些困难，它在功能上还是非常强大的。通常情况下，可考虑使用套接字的"I/O模型"，它有助于应用程序通过异步方式，同时对一个或多个套接字的通信加以管理

4、非阻塞IO的特点是用户进程需要不断的主动询问kernel数据是否准备好

#### I/O复用模型

1、主要是select和epoll

* 对一个IO端口，两次调用，两次返回，比阻塞IO并没有什么优越性
* 关键是能实现同时对多个IO端口进行监听

2、I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这两个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数

3、可以看出，由于非阻塞的调用，轮询占据了很大一部分过程，轮询会消耗大量的CPU时间。结合前面两种模式。如果轮询不是进程的用户态，而是有人帮忙就好了。多路复用正好处理这样的问题。

4、多路复用有两个特别的系统调用select或poll。select调用是内核级别的，select轮询相对非阻塞的轮询的区别在于---前者可以等待多个socket，当其中任何一个socket的数据准好了，就能返回进行可读，然后进程再进行recvform系统调用，将数据由内核拷贝到用户进程，当然这个过程是阻塞的。多路复用有两种阻塞，select或poll调用之后，会阻塞进程，与第一种阻塞不同在于，此时的select不是等到socket数据全部到达再处理，而是有了一部分数据就会调用用户进程来处理。如何知道有一部分数据到达了呢？监视的事情交给了内核，内核负责数据到达的处理。也可以理解为"非阻塞"吧

5、多路复用的特点是通过一种机制一个进程能同时等待IO文件描述符，内核监视这些文件描述符(套接字描述符)，其中的任意一个进入读就绪状态，select，poll，epoll函数就可以返回。对于监视的方式，又可以分为select，poll，epoll三种方式



#### 信号驱动I/O

1、两次调用，两次返回

2、首先我们允许套接口进行信号驱动I/O，并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据



#### 异步I/O模型

1、数据拷贝的时候进程无需阻塞

2、当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作



3、同步IO引起进程阻塞，直至IO操作完成。异步IO不会引起进程阻塞。IO复用是先通过select调用阻塞

#### 五个模型的比较



## select、poll、epoll

1、epoll跟select都能提供多路I/O复用的解决方案。在现在的Linux内核里有都能够支持，其中epoll是Linux所特有，而select则应该是POSIX所规定，一般操作系统均有实现

#### select

1、select本质上是通过设置或者检查存放fd标志位的数据结构来进行下一步处理。这样所带来的缺点是：

1. 单个进程可监视的fd数量被限制，即能监听端口的大小有限。一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048
2. 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低：当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度，不管哪个Socket是活跃的，都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的
3. 需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

#### poll

1、poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

2、它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：

1. 大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。
2. poll还有一个特点是"水平触发"，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd

#### epoll

1、epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就需态，并且只会通知一次。还有一个特点是，epoll使用"事件"的就绪通知方式，通过epoll\_ctl注册fd，一旦该fd就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知

2、epoll的优点：

1. 没有最大并发连接的限制，能打开的FD的上限远大于1024(1G的内存上能监听约10万个端口)
2. 效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。只有活跃可用的FD才会调用callback函数。即Epoll最大的优点就在于它只管你"活跃"的连接，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll
3. 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递。即epoll使用mmap减少复制开销

#### select、poll、epoll区别总结

1、支持一个进程所能打开的最大连接数

|  |  |
| --- | --- |
| select | 单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小(在32位的机器上，大小就是32\*32，同理64位机器上FD\_SETSIZE为32\*64)，当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试 |
| poll | poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的 |
| epoll | 虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接 |

2、FD剧增后带来的IO效率问题

|  |  |
| --- | --- |
| select | 因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的"线性下降性能问题" |
| poll | 同上 |
| epoll | 因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题 |

3、消息传递方式

|  |  |
| --- | --- |
| select | 内核需要将消息传递到用户空间，都需要内核拷贝动作 |
| poll | 同上 |
| epoll | epoll通过内核和用户空间共享一块内存来实现的 |

4、总结：综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点

1. 表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调
2. select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善