网络通信

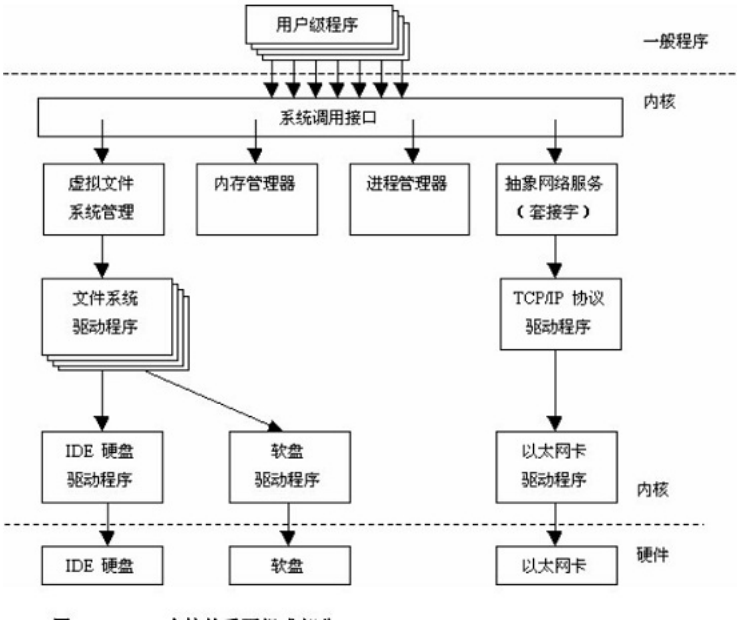
2022 年 3 月 18 日

1 Linux 基础知识

1.1 linux 内核

内核是操作系统的核心，具有很多最基本功能，它负责管理系统的进程、内存、设备驱动程序、文件和网络系统，决定着系统的性能和稳定性。

Linux 内核由如下几部分组成：内存管理、进程管理、设备驱动程序、文件系统和网络管理等。如图所示：



其中系统调用接口：SCI 层提供了某些机制执行从用户空间到内核的函数调用。

1.2 Linux 远程登录

Linux 系统中是通过 ssh 服务实现的远程登录功能，默认 ssh 服务端口号为 22。

安装 ssh：yum install ssh

启动 ssh：service sshd start

登录远程服务器：ssh -p 50022 my@127.0.0.1

输入密码：my@127.0.0.1: (-p 后面是端口，my 是服务器用户名，127.0.0.1 是服务器 ip）

回车输入密码即可登录。

1.3 Linux 进程

在 Linux 系统中，能够同时运行多个进程，Linux 通过在短的时间间隔内轮流运行这些进程而实现“多任务”。这一短的时间间隔称为“时间片”，让进程轮流运行的方法称为“进程调度”，完成调度的程序称为调度程序。

Linux 中常见的进程间通讯机制有信号、管道、共享内存、信号量和套接字等。

1.3.1 相关命令

可以使用 $ps 命令来查询正在运行的进程，比如 $ps -e、-o pid,-o comm,-o cmd，

其中-e 表示列出全部进程，-o pid,-o comm,-o cmd 分别表示需要 PID，COMMAND，CMD 信息。

三个信息的意义依次为：PID(process IDentity) 是每一个进程的身份(唯一)，是一个整数，进程也可以根据 PID 来识别其他的进程；COMMAND 是这个进程的简称；CMD 是进程所对应的程序以及运行时所带的参数。

1.3.2 Linux 进程的产生

当计算机开机的时候，内核 (kernel) 只建立了一个 init 进程。Linux 内核并不提供直接建立新进程的系统调用。其他所有进程都是 init 进程通过

fork 机制建立的。

fork 表示：新的进程要通过老的进程复制自身得到，它是一个系统调用。进程存活于内存中。每个进程都在内存中分配有属于自己的一片空间 (address space)。当进程 fork 的时候，Linux 在内存中开辟出一片新的内存空间给新的进程，并将老的进程空间中的内容复制到新的空间中，此后两个进程同时运行。

老进程成为新进程的父进程 (parent process)，而相应的，新进程就是老的进程的子进程 (child process)。一个进程除了有一个 PID 之外，还会有一个 PPID(parent PID) 来存储的父进程 PID。如果我们循着 PPID 不断向上追溯的话，总会发现其源头是 init 进程。所以说，所有的进程也构成一个以 init 为根的树状结构。

1.3.3 Linux 进程的消失

当子进程终结时，它会通知父进程，并清空自己所占据的内存，并在内核里留下自己的退出信息 (exit code，如果顺利运行，为 0；如果有错误或异常状况，为 >0 的整数)。在这个信息里，会解释该进程为什么退出。

父进程在得知子进程终结时，有责任对该子进程使用 wait 系统调用。这个 wait 函数能从内核中取出子进程的退出信息，并清空该信息在内核中所占据的空间。

但是，如果父进程早于子进程终结，子进程就会成为一个孤儿 (orphand) 进程。孤儿进程会被过继给 init 进程，init 进程也就成了该进程的父进程。init 进程负责该子进程终结时调用 wait 函数。

注意：在 Linux 中，线程只是一种特殊的进程。多个线程之间可以共享内存空间和 IO 接口。所以，进程是 Linux 程序的唯一的实现方式。

1.3.4 轻量化进程 LWP

与普通进程相比，LWP 与其他进程共享所有（或大部分）它的逻辑地址空间和系统资源；与线程相比，LWP 有它自己的进程标识符，并和其他进程有着父子关系。另外，线程既可由应用程序管理，又可由内核管理，而 LWP 只能由内核管理并像普通进程一样被调度。

默认情况下，对于一个拥有 n 个线程的程序，启动多少轻进程，由哪些轻进程来控制哪些线程由操作系统来控制，这种状态被称为非绑定的状态。因此线程结束后并不会主动释放资源。

而指定了某个线程“绑”在某个轻进程上，就称之为绑定的状态。被绑定的线程具有较高的响应速度，绑定线程可以保证在需要的时候它总有一个轻进程可用。

1.4 Linux 线程

https://blog.csdn.net/networkhunter/article/details/100218945
需要头文件：
#include <pthread.h>
编译方式：
$ gcc thread.c -o thread -lpthread
CMakeList.txt 中需要加入如下内容来添加链接库：
find_package(Threads)
target_link_libraries(…$CMAKE_THREAD_LIBS_INIT)

1.4.1 pthread_create 函数

函数概述:

- 功能：创建线程（实际上就是确定调用该线程函数的入口点）。
- 返回值：返回 0 表示成功，返回-1 表示出错。
- 范围限制：主线程，子线程属性设置完成之后。
- 注意：无。

函数声明:

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);

- thread：线程句柄，指针，指向线程标识符的指针。

  当一个新的线程调用成功之后，就会通过这个参数将线程的句柄返回给调用者，以便对这个线程进行管理。

  类型为：pthread_t。注意传入的是地址。

- attr：线程属性对象。指针类型，这个参数是可选的，默认属性下为NULL，表示非分离属性。以下众多函数均与线程属性有关。

  也可以自己设置，类型为：const pthread_attr_t。注意传入的也是地址。

  属性总览：

  typedef struct{

  int detachstate; 线程的分离状态

  int schedpolicy; 线程调度策略

  structsched_param schedparam; 线程的调度参数

  int inheritsched; 线程的继承性

  int scope; 线程的作用域

  size_t guardsize; 线程栈末尾的警戒缓冲区大小

  int stackaddr_set;

  void* stackaddr; 线程栈的位置

  size_t stacksize; 线程栈的大小

  }pthread_attr_t;

- start_routine()：入口函数，线程运行函数的地址。传入函数名即可。

  当程序调用了这个接口之后，就会产生一个线程，而这个线程的入口函数就是 start_routine()。如果线程创建成功，这个接口会返回 0。

- arg：入口函数参数指针，即指多线程要运行函数的参数。注意传入的是地址。

  注意：

  - 入口函数参数只能是一个 void * 型的指针，即只能通过结构体封装超过一个以上的参数作为一个整体传递。或者使用类中的 this 指针。（this 指针：每一个对象都能通过 this 指针来访问自己的地址。this 指针是所有成员函数的隐含参数。因此，在成员函数内部，它可以用来指向调用对象）
  - 不论是函数调用还是线程内外使用，必须对参数进行强制类型转换（void *）。

- 入口函数的返回值也同样必须是一个 void * 类型的指针，即为传入的参数。这个返回值可以通过 pthread_join() 接口获得。

使用示例：ret = pthread_create(&th, NULL, thread, &arg);

1.4.2 pthread_attr_init 函数—属性的初始化

函数概述:

- 功能：初始化一个线程属性对象。
- 返回值：0 成功，非 0 失败。
- 范围限制：主线程函数，创建线程之前。
- 注意：在对线程进行处理之前必须进行初始化。调用该函数之后，线程属性结构所包含的内容就是操作系统实现支持的线程所有属性的**默认值**。

函数声明:

int pthread_attr_init(pthread_attr_t *attr);

- attr：是一个线程属性对象。地址。定义类型为 pthread_attr_t。

使用示例：pthread_attr_init(&attr);

1.4.3 pthread_attr_destroy 函数—属性的销毁

函数概述:

- 功能：销毁一个线程属性对象。删除初始化期间分配的存储空间。
- 返回值：0 成功，非 0 失败。
- 范围限制：主线程，join 之后。
- 注意：无。

函数声明：

int pthread_attr_destroy(pthread_attr_t *attr);

- attr：是一个线程属性对象。地址。

使用示例：pthread_attr_destroy(&tattr);

1.4.4 pthread_join 函数—分离属性：线程合并

函数概述:

- 功能：子线程合入主线程，主线程阻塞等待子线程结束，然后回收子线程资源。避免内存泄漏。
- 返回值：0 成功，非 0 失败。
  - ESRCH: 没有找到与给定的线程 ID 相对应的线程。（如果多个线程等待同一个线程终止，则所有等待线程将一直等到目标线程终止。然后一个等待线程成功返回。其余的等待线程将失败返回 ESRCH 错误）
  - EDEADLK：将出现死锁，如一个线程等待其本身，或者线程 A 和线程 B 互相等待。
  - EINVAL：与给定的线程 ID 相对应的线程是分离线程。
- 范围限制：主线程函数，需等待子线程执行结束位置。
- 注意：
  - 非分离线程属性下，只有在主线程调用该函数时，该线程才会释放自己的资源。
  - 多线程编程中往往以 for 循环的形式调用该函数。
  - 该函数会让主线程阻塞，直到所有线程都已经退出。
  - 线程合并和分离必须二选一。

函数声明：

int pthread_join(pthread_t thread, void **status);

- thread：需要等待的线程，指定的线程必须位于当前的进程中，而且不得是分离属性的线程。不需要写为线程的地址。
- status：线程 thread 所执行的函数返回值存放地址（返回值地址需要保证有效），其中 status 可以为 NULL。需要强制类型转换 (void **)&。

  注意：在主函数中定义，类型与入口函数的参数类型一致即可。

使用示例：pthread_join(th, (void**)&thread_ret);

1.4.5 pthread_detach 函数—分离属性：线程分离

函数概述:

- 功能：设置分离属性，使主线程与子线程分离，子线程结束后，资源自动回收。
- 返回值：调用成功完成之后返回零。其他任何返回值都表示出现了错误：

  EINVAL：thread 是分离线程；

  ESRCH：thread 不是当前进程中有效的为分离线程。
- 范围限制：子线程中，函数开头位置设置。
- 注意:调用该函数后,入口函数结束时自动回收资源,不用调用 pthread_join()。往往适用于不需要主线程等待的情况。

函数声明：

int pthread_detach(pthread_t thread);

- thread：线程标识符。

使用示例：pthread_detach(pthread_self()); 其中pthread_self() 是获得自身线程标识符函数。

1.4.6 pthread_attr_setdetachstate 函数—分离属性

用来决定一个线程以什么样的方式来终止自己，何时释放自己的资源。

函数概述:

- 功能：分离属性就是让线程在创建之前就决定它应该是分离的。如果设置了这个属性，就没有必要调用 pthread_join() 或 pthread_detach() 来回收线程资源了。
- 返回值：成功返回 0，若失败返回-1。
- 范围限制：主线程函数，创建线程之前，定义属性后。
- 注意：无。

函数声明:

pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

- attr：是一个线程属性对象。地址。
- detachstate：属性，可取值为：

  PTHREAD_CREATE_DETACHED（分离的）

  PTHREAD_CREATE_JOINABLE（可合并的，也是默认属性）

  使用示例：ret = pthread_attr_setdetachstate(&tattr, PTHREAD_CREATE_DETACHED);

1.4.7 pthread_attr_setscope 函数—绑定属性

绑定属性：指一个用户线程固定地分配给一个内核线程。

函数概述:

- 功能：设置线程的绑定属性。
- 返回值：0 成功，-1 失败。
- 范围限制：主线程函数，分离属性之后。

- 注意：Linux 的线程永远都是绑定的，所以非绑定状态在 Linux 中不管用，而且会返回 ENOTSUP 错误。设置为绑定状态后则不能重新设置为非绑定状态。

函数声明：

int pthread_attr_setscope(pthread_attr_t *attr, int scope);

- attr：线程属性对象的指针，传入的是地址，定义类型为 pthread_attr_t。
- scope：传入类型。包括：
  - PTHREAD_CREATE_DETACHED：绑定状态；
  - PTHREAD_CREATE_JOINABLE：非绑定状态；

使用示例：pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

1.4.8 pthread_attr_setschedpolicy 函数—调度属性（算法）

线程的调度属性有三个，分别是：算法、优先级和继承权。
算法：Linux 提供的线程调度算法有三个：轮询、先进先出和其它。

- 轮询：指时间片轮转，当线程的时间片用完，系统将重新分配时间片，并将它放置在就绪队列尾部，以保证具有相同优先级的轮询任务获得公平的 CPU 占用时间。
- 先进先出：就是先到先服务，一旦线程占用了 CPU 则一直运行，直到有更高优先级的线程出现或自己放弃。
- 其他：代表采用 Linux 自己认为更合适的调度算法，默认算法。
- 注意：其中轮询和先进先出调度算法是 POSIX 标准所规定，属于实时调度算法。

函数概述：

- 功能：设置线程调度算法的接口。
- 返回值：成功则返回 0，失败返回其他值。
- 范围限制：主线程函数，绑定属性之后。
- 注意：无。

函数声明：

pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

- attr：是一个线程属性对象。地址。
- policy：线程调度策略属性值中指定的值。可为：

  SCHED_RR（轮询）

  SCHED_FIFO（先进先出）

  SCHED_OTHER（其它）

  使用示例：pthread_attr_setschedpolicy(&attr[i], SCHED_FIFO);

1.4.9 pthread_attr_setschedparam 函数—调度属性（优先级）

优先级是从 1 到 99 的数值，数值越大代表优先级越高。

只有采用 SHCED_RR 或 SCHED_FIFO 调度算法时，优先级才有效。其他情况下，优先级恒为 0。

函数概述：

- 功能：设置线程优先级的接口。
- 返回值：成功则返回 0，失败返回其他值。
- 范围限制：主线程函数，创建线程之前，调度算法之后。
- 注意：如果要设置这个值，需满足：进程必须是以 root 账号运行的；
  需要放弃线程的继承权。

函数声明：

int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *param)

- attr：是一个线程属性对象。地址。
- param：结构体，其 sched_priority 字段就是线程的优先级。

  使用方式：

  struct sched_param param;

  param.sched_priority = 20;

使用示例：pthread_attr_setschedparam(&attr[i], param);

1.4.10 pthread_attr_setinheritsched 函数—调度属性（继承权）

函数概述:

- 功能：设置线程继承权的接口。
- 返回值：成功则返回 0，失败返回其他值。
- 范围限制：主线程函数，调度优先级之后。
- 注意：无。

函数声明:

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);

- atttr：是一个线程属性对象。地址。
- inheritsched：有两个取值：

  PTHREAD_INHERIT_SCHED（拥有继承权）（默认）

  PTHREAD_EXPLICIT_SCHED（放弃继承权）

使用示例:pthread_attr_setinheritsched(&attr[i], PTHREAD_EXPLICIT_SCHED);

1.4.11 pthread_attr_setstacksize 函数—作用域属性

函数概述:

- 功能：修改线程堆栈大小属性的接口。
- 返回值：0 成功，非 0 失败。
- 范围限制：主线程函数，创建线程之前，调度属性之后。
- 注意：Linux 系统为每个线程默认分配了 8MB 的堆栈空间，如果觉得这个空间不够用，可以通过修改线程的堆栈大小属性进行扩容。

函数声明：

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

- attr：是一个线程属性对象。地址。
- stacksize：堆栈大小，以字节为单位。需要注意的是，线程堆栈不能小于 16KB，而且尽量按 4KB(32 位系统) 或 2MB（64 位系统）的整数倍分配，也就是内存页面大小的整数倍。

使用示例：略。

1.4.12 pthread_attr_setguardsize 函数—满栈警戒区属性

函数概述:

警戒区：

为了防止线程的堆栈用满，Linux 为线程堆栈设置了一个满栈警戒区。这个区域一般就是一个页面，属于线程堆栈的一个扩展区域。一旦有代码访问了这个区域，就会发出 SIGSEGV 信号进行通知。

- 功能：修改满栈警戒区属性的接口。
- 返回值：0 成功，非 0 失败。
- 范围限制：主线程函数，创建线程之前，作用域之后。
- 注意：如果修改了线程堆栈的大小，那么系统会认为用户会自己管理堆栈，也会将警戒区取消掉，如果有需要就要开启它。

函数声明：

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);

- attr：是一个线程属性对象。地址。
- guardsize：警戒区大小了，以字节为单位。与设置线程堆栈大小属性相仿，应该尽量按照 4KB 或 2MB 的整数倍来分配。当设置警戒区大小为 0 时，就关闭了这个警戒区。

使用示例：略。

获取栈溢出保护区大小：

int pthread_attr_getguardsize(pthread_attr_t *attr, size_t guardsize)

其中第二个参数为输出的单位，1 表示 1 字节。

1.4.13 线程本地存储

Thread Local Storage，简称 TLS，线程私有的全局变量，用于避免全局变量的值（如 error）被所有子线程修改，导致不确定性的问题。

私有化全局变量方法

在全局变量的定义前增加 ___thread 即可。这样每个线程都有一个自己私有的全局变量 g，子线程对值的修改不影响主线程的值，主线程也不影响子线程的值。

pthread_key_create 函数

int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));

- 功能：用于创建一个线程本地存储区。
- 返回值：0 成功，非 0 失败。
- 范围限制：主线程函数，创建线程之前。
- 注意：参数涉及强制类型转换。内部函数需要通过指针赋值方式来释放私有化全局变量。
- 参数：
  - key ： 用来返回这个存储区的句柄，需要使用一个全局变量保存，以便所有线程都能访问到。类型：pthread_key_t。传入地址。
  - *destructor ： 指明了一个 destructor 函数，如果这个参数不为空，那么当每个线程结束时，系统将调用这个函数来释放绑定在这个键上的内存块，否则可以传递 NULL。类型必须为 void *，参数也需为 void *。

pthread_key_delete 函数

int pthread_key_delete(pthread_key_t key);

- 功能：用于回收线程本地存储区。
- 返回值：0 成功，非 0 失败。
- 范围限制：主线程函数，线程回收之后。

- 注意：无。
- 参数：唯一的参数就要回收的存储区的句柄。

pthread_getspecific 函数

void* pthread_getspecific(pthread_key_t key);

- 功能：用于获取线程本地存储区的数据。
- 返回值：0 成功，非 0 失败。
- 范围限制：子线程中。
- 注意：无。
- 参数：句柄。

pthread_setspecific 函数

int pthread_setspecific(pthread_key_t key, const void *value);

- 功能：用于设置线程本地存储区的数据。
- 返回值：0 成功，非 0 失败。
- 范围限制：子线程中，使用该值前。
- 注意：无。
- 参数：
  - key：句柄。
  - value：需要采用本地存储区的值的指针，不需要强制类型转换，不需要常量，类型 thread_data_t *。

1.5 线程的同步

Linux 提供的线程同步机制主要有互斥锁和条件变量。

1.5.1 互斥锁

用来保证多线程的共享数据的完整性，即在一个线程操作某共享数据时对其上锁，使其他的线程无法对该数据进行操作。

特点：

- 原子性：把一个互斥量锁定为一个原子操作，如果一个线程锁定了一个互斥量，没有其他线程在同一时间可以成功锁定这个互斥量；
- 唯一性：如果一个线程锁定了一个互斥量，在它解除锁定之前，没有其他线程可以锁定这个互斥量；
- 非繁忙等待：如果一个线程已经锁定了一个互斥量，第二个线程又试图去锁定这个互斥量，则第二个线程将被挂起（不占用任何 cpu 资源），直到第一个线程解除对这个互斥量的锁定为止，第二个线程则被唤醒并继续执行，同时锁定这个互斥量。
- 使用互斥锁会影响代码的执行效率，多任务改成了单任务执行。这段代码称为临界区。

死锁

两个线程已经各自拥有一把锁了，但是还想得到对方的锁，这个时候两个线程都会被暂停，称为死锁。

互斥锁基本函数

- pthread_mutex_init(A,B) //初始化互斥锁 [主函数]

  A：pthread_mutex_t *restrict mutex：指向互斥量的指针，类型为 pthread_mutex_t，传入地址。

  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;//定义

  B：const pthread_mutexattr_t *restrict attr：指向互斥量属性的指针，NULL 即可。

- pthread_mutex_destroy(A) //销毁互斥锁 [主函数]
- pthread_mutex_lock(A) //互斥锁获得锁 [子线程]
- pthread_mutex_trylock(A) //尝试获得锁 [子线程]

  如果该互斥锁已经锁定，则返回一个不为的错误值，如果该互斥锁没有锁定，则返回 0，表示尝试加锁成功。

用它试图加锁的线程永远都不会被系统暂停，只是通过返回 EBUSY 来告诉程序员这个锁已经有人用了。至于是否继续 “强闯” 临界区，则由程序员决定。

- pthread_mutex_unlock(A) //释放互斥锁 [子线程]

注意事项

- 对共享资源操作前一定要获得锁。
- 完成操作以后一定要由上锁的进 (线) 程释放锁释放锁。
- 锁并不是与某个具体的变量进行关联，它本身是一个独立的对象。
- 在获得锁和释放锁之间的操作可视为单线程的，因此应尽量短时间地占用锁。
- 如果有多锁, 如获得顺序是 ABC 连环扣, 释放顺序也应该是 ABC。
- 线程错误返回时应该释放它所获得的锁。

1.5.2 条件变量

条件变量是一种事件机制。由一类线程来**控制 “事件”** 的发生，另外一类线程**等待 “事件”** 的发生。主要包括两个动作：

一个线程等待” 条件变量的条件成立” 而挂起暂停；

另一个线程使” 条件成立”（给出条件成立信号）。

条件变量必须是**共享于线程之间的全局变量**。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。

条件变量基本函数

- pthread_cond_init(C,D) //初始化条件变量的接口 [主函数]

  C：pthread_cond_t *cond：是一个指向条件变量的指针，类型为 pthread_cond_t，传入地址。

  pthread_cond_t has_producer = PTHREAD_COND_INITIALIZER;//定义

  D：const pthread_condattr_t *attr：是一个指向条件变量属性的指针，NULL 即可。

- pthread_cond_destory(C) //销毁条件变量的接口，非释放内存空间 [主函数]
- pthread_cond_signal(C) //控制“事件”发生的接口，激活等待列表中的线程，即广播形式 [子线程]
- pthread_cond_broadcast(C) //控制“事件”发生的接口，激活所有等待线程列表中最先入队的线程，即单播形式 [子线程]
- pthead_cond_wait(C,A) //等待“事件”发生的接口，即等待条件变量，挂起线程，没有信号就一直等待 [子线程]
- pthread_cond_timedwait(C,A,E) //等待“事件”发生的接口，不同之处在于会有 timeout 时间，如果到了 timeout，线程自动解除阻塞 [子线程]

  E：const timespec *abstime：等待时间（其值为系统时间 + 等待时间）

注意事项

- 函数 pthead_cond_wait 的工作流程为：对于一个已经成功锁定了的进程，首先将本线程休眠，放在等待队列（称为阻塞）；然后对本线程的互斥量解锁，接着等待别的线程的解锁信号后去竞争锁（称为唤醒），继续执行线程。（解锁并阻塞是一个原子操作）
- 解锁信号：另一个占据了该互斥量的线程在完成了对全局变量的操作后，执行解锁，然后调用 thread_cond_broadcast 或 pthread_cond_signal 函数（广播）。
- 但由于解锁信号对应的情况下并不一定与唤起该线程的情况相同，则往往在调用 pthead_cond_wait 时会加入一个额外的判断来确定是否还需要等待。
- 判断可以采用 if 的形式，也可以采用 while 的形式。前者在总计只有 2 个线程的情况下与后者相同。否则前者在下次竞争不到锁的时候就会继续执行线程的操作，而后者则会一直等待下去。
- 先发解锁信号再解锁。避免有线程在收到信号前恰好获得锁。

与互斥锁关系

- 与互斥锁同时采用，可用于让线程按一定的顺序来操作全局变量数据。即一个线程用互斥锁锁住了某一个操作过程，但如果全局变量不满足条件变量的要求，则会解锁该操作规程，让其他线程来操作。当满足要求时，再锁住操作过程执行。(在此期间一直处于临界区)
- pthread_cond_wait 与 thread_mutex_lock 连用，在线程通过后者获得锁后，如果不满足条件，则调用前者释放锁，并将本线程放在线程等待队列之后。
- 额外的解锁条件需要自己编写。

1.5.3 基本流程

提前需定义的参数:

- pthread_t //线程对象
- pthread_attr_t //线程属性对象
- pthread_key_t //本地存储区键
- pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; //互斥锁对象
- pthread_cond_t cond = PTHREAD_COND_INITIALIZER; //条件变量对象

主线程:

- pthread_attr_init //初始化线程属性
- pthread_attr_setdetachstate //设置分离属性，无 join、detach
- pthread_attr_setscope //设置绑定属性
- pthread_attr_setschedpolicy //设置调度属性 (算法)
- pthread_attr_setschedparam //设置调度属性 (优先级)
- pthread_attr_setinheritsched //设置调度属性 (继承权)

- pthread_attr_setstacksize //设置作用域属性
- pthread_attr_setguardsize //设置满栈警戒区属性
- pthread_key_create //创建线程本地存储区
- pthread_key_delete //回收线程本地存储区
- pthread_mutex_init //初始化互斥锁
- pthread_cond_init //初始化条件变量
- pthread_create //创建线程
- pthread_join //调用回收线程
- pthread_mutex_destroy //销毁互斥锁
- pthread_cond_destory //销毁条件变量
- pthread_attr_destroy //销毁线程

子线程:

- pthread_detach //自动回收，无 join
- pthread_setspecific //设置线程本地存储区的数据
- pthread_getspecific //获取线程本地存储区的数据
- pthread_mutex_lock //互斥锁获得锁
- pthead_cond_wait //等待条件变量，挂起线程
- 公共操作
- pthread_cond_signal 或 pthread_cond_broadcast 释放解锁信号广播/单播
- pthread_mutex_unlock //释放互斥锁

1.6 Linux 内存管理

了让有限的物理内存满足应用程序对内存的大需求量，Linux 采用了称为“虚拟内存”的内存管理方式。Linux 将内存划分为容易处理的“内存页”（对于大部分体系结构来说都是 4KB）。Linux 包括了管理可用内存的方式，以及物理和虚拟映射所使用的硬件机制。

1.7 Linux 文件系统

Linux 系统能支持多种目前流行的文件系统，如 EXT2、EXT3、FAT、FAT32、VFAT 和 ISO9660。

Linux 下面的文件类型主要有：

1) 普通文件：C 语言元代码、SHELL 脚本、二进制的可执行文件等。分为纯文本和二进制；

2) 目录文件：目录，存储文件的唯一地方；

3) 链接文件：指向同一个文件或目录的的文件；

4) 设备文件：与系统外设相关的，通常在/dev 下面。分为块设备和字符设备；

5）管道 (FIFO) 文件: 提供进程之间通信的一种方式；

6）套接字 (socket) 文件：该文件类型与网络通信有关；

可参考：https://www.linuxprobe.com/linux-system-structure.html

1.8 Linux 防火墙

1.8.1 常用命令

- 启动：systemctl start firewalld
- 重启：systemctl restart firewalld
- 停止：systemctl stop firewalld
- 重新加载：firewall-cmd –reload
- 查看 firewalld 的运行状态：

  firewall-cmd –state

- 取消开放 https 服务，即禁止 https 服务：

  firewall-cmd –zone=drop –remove-service=https

- 开放 22 端口：

  firewall-cmd –zone=drop –add-port=22/tcp

- 取消开放 22 端口：

  firewall-cmd –zone=drop –remove-port=22/tcp

- 开放两个端口：

  firewall-cmd –zone=drop –add-port=8080-8081/tcp

- 查询 drop 区域开放了哪些端口：

  firewall-cmd –zone=drop –list-ports

- 其他常用命令：

  允许 icmp 协议流量，即允许 ping：

  firewall-cmd –zone=drop –add-protocol=icmp

  取消允许 icmp 协议的流量，即禁 ping：

  firewall-cmd –zone=drop –remove-protocol=icmp

  查询 drop 区域开放了哪些协议：

  firewall-cmd –zone=drop –list-protocols

  将原本访问本机 888 端口的流量转发到本机 22 端口：

  firewall-cmd –zone=drop –add-forward-port =port=888:proto=tcp:toport=22

2 socket 基础

2.1 socket 通信的概念

套接字，运行在计算机中的两个程序通过 socket 建立起一个通道，数据在通道中传输。套接字可以看成是两个网络应用程序进行通信时，各自通信连接中的端点，这是一个逻辑上的概念。它是网络环境中进程间通信的 API(应用程序编程接口)（是 IP 地址和端口结合)。

socket 把复杂的 TCP/IP 协议族隐藏了起来，只要用好 socket 相关的函数，就可以完成网络通信。

2.2 socket 的分类

socket 提供了流（stream）和数据报（datagram）两种通信机制，即流 socket 和数据报 socket。

流 socket 基于 TCP 协议，是一个有序、可靠、双向字节流的通道，传输数据不会丢失、不会重复、顺序也不会错乱。类似于打电话。

数据报 socket 基于 UDP 协议，不需要建立和维持连接，可能会丢失或错乱。UDP 不是一个可靠的协议，对数据的长度有限制，但是它的速度比较高。类似于短信。

在实际开发中，数据报 socket 的应用场景极少。

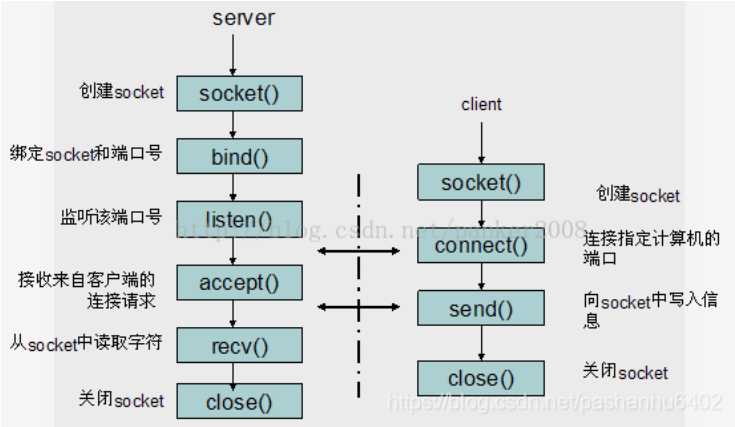
2.3 socket 通信的过程

2.3.1 服务端

- 创建服务端的 socket。socket()
- 把服务端指定的用于通信的 ip 地址和端口绑定到 socket 上。bind()
- 把 socket 设置为监听模式，以监听用户请求。listen()
- 接受客户端的连接。accept()
- 与客户端通信,接收客户端发过来的报文后,回复处理结果。send()/recv()
- 不断的重复上一步步，直到客户端断开连接。
- 关闭 socket，释放资源。close()

2.3.2 客户端

- 创建客户端的 socket。socket()
- 向服务器发起连接请求。connect()
- 与服务端通信,发送一个报文后等待回复,然后再发下一个报文。send()/recv()
- 不断的重复上一步，直到全部的数据被发送完。
- 关闭 socket，释放资源。close()



2.4 Socket

在 Linux 系统中，一切输入输出设备皆文件。而 socket 本质上可以视为一种特殊的文件，即通信的实现，因此 socket 的通信过程也可以理解为通过“打开 open –> 读写 write/read –> 关闭 close”模式的操作过程。

注意：

a. 客户端有一个 socket，是用于发送报文的 socket；而服务端有两个 socket，一个是用于监听的 socket，还有一个就是客户端连接成功后，由 accept 函数创建的用于与客户端收发报文的 socket。

b. socket 是系统资源，操作系统打开的 socket 数量是有限的，在程序退出之前必须关闭已打开的 socket，就像关闭文件指针一样，就像 delete 已分配的内存一样，极其重要。

c. 关闭 socket 的代码不能只在 main 函数的最后，那是程序运行的理想状态，还应该在 main 函数的每个 return 之前关闭。

d. socket 类型的 servaddr 的地址的强制转换，(struct sockaddr *)&servaddr。

2.5 Socket 主要参数及结构体

2.5.1 socket 函数返回的值

返回值称为 socket 描述符（socket descriptor），其本质是一个文件描述符，是一个整数。把它作为参数，通过它来进行一些读写操作。

0 表示标准输入，1 表示标准输出，2 表示标准错误。

注意：默认创建的 socket 是主动连接的。

2.5.2 有关定义

- 大端模式、小端模式："大端" 和" 小端" 表示多字节值的哪一端存储在该值的起始地址处; 小端存储在起始地址处, 即是小端字节序; 大端存储在起始地址处, 即是大端字节序。

  a. 大端字节序（Big Endian）：最高有效位存于最低内存地址处，最低有效位存于最高内存处；　　b. 小端字节序（Little Endian）：最高有效位存于最高内存地址，最低有效位存于最低内存处。

- 网络字节序 NBO：网络字节序是大端字节序。在进行网络传输的时候，发送端发送的第一个字节是高位字节。

- 主机字节序 HBO：不同的机器 HBO 不相同，与 CPU 的设计有关，数据的顺序是由 CPU 决定的，而与操作系统无关。不同体系结构的机器之间不能直接通信，所以要转换成一种约定的顺序，也就是网络字节顺序。存在专门的函数来进行二者的转换。

- 转换函数命名规则：h 主机、n 是网络字节序，to 是转换为，s 是短型数据，l 是长型。

- 命名规则：

  socket 字符：sockfd 客户端，listenfd 服务端；

  网络地址：clientaddr 客户端，servaddr 服务端；

- 待添加

2.5.3 struct hostent* h

域名和网络地址结构体：
struct hostent
{
　　char *h_name; //主机名，即官方域名
　　char **h_aliases; //主机所有别名构成的字符串数组，同一 IP 可绑定多个域名

int h_addrtype; //主机 IP 地址的类型，AF_INET（ipv4）、AF_INET6（ipv6）

int h_length; //主机 IP 地址长度，IPV4 为 4，IPV6 为 16。

char **h_addr_list; //主机的 ip 地址，虽然为 char 形式，但以网络字节序格式存储，可以通过 memcpy 互相拷贝。

};

注意：a. 通常用于客户端已知对方 ip 情况下获得对方网络其他信息。

b. 与下面的结构体一样，定义时指的是哪个端，其内数据就指哪个端。

c. 具有预定义参数：# define h_addr h_addr_list[0]

d. 主要用于客户端存入端口和 ip 地址信息，然后通过 h_addr 来将信息传递给网络地址结构体 sin_addr。

操作函数

- *gethostbyname(const char *name);

  用于客户端指定服务端的 ip 地址，详见后文函数介绍，注意参数为 char 类型的常量。成功执行该函数后结构体内各变量均可获得。

- 新型网路地址转化函数 inet_pton 和 inet_ntop：

  这两个函数是随 IPv6 出现的函数，对于 IPv4 地址和 IPv6 地址都适用，函数中 p 和 n 分别代表表达（presentation）和数值（numeric）。地址的表达格式通常是 ASCII 字符串，数值格式则是存放到套接字地址结构的二进制值。

  int inet_pton(int family, const char *strptr, void *addrptr); //将点分十进制的 ip 地址转化为用于网络传输的数值格式

  返回值：若成功则为 1，若输入不是有效的表达式则为 0，若出错则为-1。

  const char * inet_ntop(int family, const void *addrptr, char *strptr, size_t len); //将数值格式转化为点分十进制的 ip 地址格式

  返回值：若成功则为指向结构的指针，若出错则为 NULL。

  其中：family 即主机 IP 地址的类型参数。

2.5.4 struct sockaddr_in servaddr

表示地址信息的数据结构（体），存放了目标地址和端口。与结构体 sockaddr 把目标地址和端口信息混在一起了不同，其把 port 和 addr 分开储存在两个变量中。

struct sockaddr_in

{

sa_family_t sin_family; //协议族，在 socket 编程中只能是 AF_INET。

unit16_t sin_port; //16 位的 TCP/UDP 端口号。

struct in_addr sin_addr; //32 位 IP 地址

char sin_zero[8]; //不使用。

}

其中：

struct in_addr

{

in_addr_t s_addr; //32 位 IP v4 地址。

(可表示为：servaddr.sin_addr.s_addr)

}

操作函数

- inet_addr()：将十进制 IP 地址字符串转为网络二进制数字（网络字节序），返回的 IP 地址是网络字节序的。

  (const char—-> in_addr_t)(ascii to network)

- inet_ntoa()：将网络二进制数字（网络字节序）转为十进制 IP 地址字符串。

  (in_addr—->const char)(network to ascii)

- inet_aton()：将十进制 IP 地址字符串转为网络二进制数字（网络字节序），与 inet_addr 的区别是，结果不是作为返回值，而是保存形参 inp 所指的 in_addr 结构体中（前者需要保存到 in_addr_t 变量中）。

  函数原型：int inet_aton(cont char* cp, struct in_addr_t *inp)

  (const char—-> in_addr)(ascii to network)

- htons()、htonl() 作用是将端口号由主机字节序转换为网络字节序的整数值。前者针对的是 16 位的（short）、后者针对的是 32 位的（long）。

  (int to in_addr_t)(host to net)

  (不一定是 in_addr_t，满足要求的字符即可，例如 unit16_t)

- 与 htonl() 和 htons() 作用相反的两个函数是：ntohl() 和 ntohs()。

  (net to host)

- 应用：

  a. 服务端绑定 (自身的)IP 地址：

  servaddr.sin_addr.s_addr = inet_addr("192.168.149.129"); // 指定 ip 地址

  servaddr.sin_addr.s_addr = htonl(INADDR_ANY); // 本主机的任意 ip 地址。在实际开发中，采用任意 ip 地址的方式比较多。

  b. 服务端绑定通信端口，也可用于客户端指定服务端的通信端口：

  servaddr.sin_port = htons(5000); // 通信端口 5000

- 注意：htons() 的参数为 int 型的无符号短整形数，若用 argv[] 来传递，需要强制转换类型函数 atoi()。同理 htonl()。

- 注意：其中 sin_addr 和 sin_port 的字节序都是网络字节序，而 sin_family 不是。因为前者是从 IP 和 UDP 协议层取出来的数据，是直接和网络相关的。但 sin_family 域只是内核用来判断 struct sockaddr_in 是存储的什么类型的数据，且永远不会被发送到网络上，所以可以使用主机字节序来存储。

- 注意：不论是那个端的程序采用，servaddr 指的是哪个端，端口、IP 地址就是哪个端。

注意：通常服务器在启动的时候都会绑定一个众所周知的地址（如 ip 地址 + 端口号），用于提供服务，客户就可以通过它来接连服务器；而客户端就不用指定，有系统自动分配一个端口号和自身的 ip 地址组合。这就是为什么通常服务器端在 listen 之前会调用 bind()，而客户端就不会调用，而是在 connect() 时由系统随机生成一个。

2.6 客户端服务端函数

以下为几种常用的接口函数:

2.6.1 socket 函数

函数概述:

- 功能：用于创建一个新的 socket。对应于普通文件的打开操作。
- 返回值：成功则返回一个 socket 描述符，失败返回-1，错误原因存于 errno 中。
- 范围限制：客户端、服务端。
- 注意：无。

函数声明:

int socket (int domain, int type, int protocol);

- domain：协议域，又称协议族（family）。协议族决定了 socket 的地址类型，在通信中必须采用对应的地址。

  AF_INET：决定了要用 ipv4 地址（32 位的）与端口号（16 位的）的组合;

  AF_UNIX：决定了要用一个绝对路径名作为地址。

- type：指定 socket 类型。

  流式 socket（SOCK_STREAM）是一种面向连接的 socket，针对于面向连接的 TCP 服务应用。数据报式 socket（SOCK_DGRAM）是一种无连接的 socket，对应于无连接的 UDP 服务应用。

- protocol：指定协议。

  常用协议有 IPPROTO_TCP、IPPROTO_UDP、IPPROTO_STCP、IPPROTO_TIPC 等,分别对应 TCP 传输协议、UDP 传输协议、STCP 传输协议、TIPC 传输协议。

  注意为 0 则与 type 相匹配，与 type 不能随意匹配。

- 正常情况,第一个参数只能填 AF_INET,第二个参数只能填 SOCK_STREAM,第三个参数只能填 0。

  使用示例：int sockfd = socket(AF_INET,SOCK_STREAM,0))

2.6.2 send 函数

函数概述:

- 功能：把数据通过 socket 发送给对端。
- 返回值：函数返回已发送的字符数。出错时返回-1，错误信息 errno 被标记。
- 范围限制：客户端、服务端。
- 注意：就算是网络断开，或 socket 已被对端关闭，send 函数不会立即报错，要过几秒才会报错。如果 send 函数返回的错误（<=0），表示通信链路已不可用。

函数声明:

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

- sockfd：已建立好连接的客户端socket。
- buf：需发送数据的内存地址。
- len：需发送数据的长度。
- flags：填 0, 其他数值意义不大。
  - MSG_CONFIRM ： 用来告诉链路层。
  - MSG_DONTROUTE：不要使用网关来发送数据，只发送到直接连接的主机上。通常只有诊断或者路由程序会使用，这只针对路由的协议族定义的，数据包的套接字没有。
  - MSG_DONTWAIT ： 启用非阻塞操作，如果操作阻塞，就返回 EAGAIN 或 EWOULDBLOCK。
  - MSG_EOR ： 当支持 SOCK_SEQPACKET 时，终止记录。

- MSG_MORE ： 调用方有更多的数据要发送。这个标志与 TCP 或者 udp 套接字一起使用。
- MSG_NOSIGNAL ：当另一端中断连接时，请求不向流定向套接字上的错误发送 SIGPIPE,EPIPE 错误仍然返回。
- MSG_OOB:在支持此概念的套接字上发送带外数据(例如,SOCK_STREAM 类型)；底层协议还必须支持带外数据。

使用示例：iret=send(sockfd,buffer,strlen(buffer),0);

2.6.3 recv 函数

函数概述:

- 功能：接收对方 socket 发送过来的数据。
- 返回值：函数返回已接收的字符数。出错时返回-1，失败时不会设置 errno 的值。
- 范围限制：客户端、服务端。
- 注意：如果 socket 的对端没有发送数据，recv 函数就会等待，如果对端发送了数据，函数返回接收到的字符数。出错时返回-1。如果 socket 被对端关闭，返回值为 0。如果 recv 函数返回的错误（<=0），表示通信通道已不可用。

函数声明:

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

- sockfd：为已建立好连接的客户端socket。
- buf：用于接收数据的内存地址。
- len：接收数据的长度，不能超过 buf 的大小，否则内存溢出。
- flags：填 0, 其他数值意义不大。同上。

使用示例：iret=recv(sockfd,buffer,sizeof(buffer),0);

2.6.4 gethostbyname() 函数

函数概述:

- 功能：把 ip 地址或域名转换为 hostent 结构体表达的地址。
- 返回值：如果成功，返回一个 hostent 结构指针，失败返回 NULL。
- 范围限制：客户端。
- 注意：只要地址格式没错，一般不会返回错误。失败时不会设置 errno 的值。

函数声明:

struct hostent *gethostbyname(const char *name);

- name：域名或者主机名，例如”192.168.1.3”、”www.freecplus.net” 等。

使用示例：struct hostent *h = gethostbyname(argv[1])

2.6.5 connect 函数

函数概述:

- 功能：客户端通过调用 connect 函数来建立客户端 socket 与服务器的连接。
- 返回值：成功则返回 0，失败返回-1，错误原因存于 errno 中。
- 范围限制：客户端。
- 注意:如果服务端的地址错了,或端口错了,或服务端没有启动,connect 一定会失败。

函数声明:

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

- sockfd：客户端 socket。
- *addr：服务端的网络地址（注意const struct sockaddr * 类型，需要强制转换）。

- addrlen：服务端网络地址的长度（addr 结构体的大小）。

使用示例：connect(sockfd, (struct sockaddr *)&servaddr,sizeof(servaddr))

2.6.6 bind 函数

函数概述:

- 功能：服务端把自身用于通信的地址和端口绑定到 socket 上。
- 返回值：成功则返回 0，失败返回-1，错误原因存于 errno 中。
- 范围限制：服务端。
- 注意：如果绑定的地址错误，或端口已被占用，bind 函数一定会报错，否则一般不会返回错误。

函数声明:

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

- sockfd：服务端 socket。
- *addr：服务端的网络地址。（const struct sockaddr * 类型的指针，需要使用强制类型转换。指向要绑定给 sockfd 的协议地址。存放了服务端用于通信的地址和端口）。
- addrlen：服务端网络地址的长度。

使用示例：bind(listenfd,(struct sockaddr *)&servaddr,sizeof(servaddr));

2.6.7 listen 函数

函数概述:

- 功能：用于把主动连接 socket 变为被动连接的 socket，使得这个 socket 可以接受其它 socket 的连接请求，从而成为一个服务端的 socket。调用之后，服务端的 socket 就可以调用 accept 来接受客户端的连接请求。

- 返回值：0-成功，-1-失败，错误原因存于 errno 中。
- 范围限制：服务端。
- 注意：listen 函数一般不会返回错误。

函数声明：

int listen(int sockfd, int backlog);

- sockfd：服务端已经被 bind 过的 socket。
- backlog：这个参数涉及到一些网络的细节，比较麻烦，填 5、10 都行，一般不超过 30。

使用示例：listen(listenfd,5);

2.6.8 accept 函数

函数概述：

- 功能：用于服务端接受客户端的连接。
- 返回值：成功则返回 0，失败返回-1，错误原因存于 errno 中。
- 范围限制：服务端。
- 注意：
- a. 函数在等待的过程中，如果被中断或其它的原因，函数返回-1，表示失败，如果失败，可以重新 accept。

  b. 函数等待客户端的连接，如果没有客户端连上来，它就一直等待，这种方式称之为阻塞。

  c. 函数等待到客户端的连接后，创建一个新的 socket，函数返回值就是这个新的 socket，服务端使用这个新的 socket 和客户端进行报文的收发。

函数声明：

int accept(int sockfd,struct sockaddr *addr,socklen_t *addrlen);

- sockfd：服务端已经被 listen 过的 socket。
- addr：客户端的网络地址（const struct sockaddr *）。如果不需要客户端的地址，可以填 0。也可以先定义一个空的客户端地址，然后填进去。
- addrlen：客户端网络地址长度，如果 addr 为 0，addrlen 也填 0。

使用示例：clientfd=accept(listenfd,(struct sockaddr *)&clientaddr,(socklen_t*)&socklen);

2.6.9 附注：计算大小函数、运算符

sizeof 运算符

- 参数可以是数组、指针、类型、对象、函数等。
- 功能是获得保证能容纳实现所建立的最大对象的字节大小。
- 不能用来返回动态分配的内存空间的大小。
- 返回值为 int 类型，大小与空间大小有关，跟对象、结构、数组所存储的内容没有关系。
- 内容区分为指针时，指的是存储该指针所用的空间大小（存储该指针的地址的长度，是长整型，应该为 4），而对于数组名字，则返回数组大小。

strlen 函数

- 参数必须是字符型指针（char*）（数组名视为指针）。
- 功能是返回字符串的长度。
- 可以返回动态内存空间大小。
- 返回值为 int 类型，大小与存储的数据内容有关，与空间的大小和类型无关。

- 不区分内容是数组还是指针，就读到
  0 为止返回长度。且不把
  0 计入字符串的长度。

2.6.10 附注：开辟空间函数、运算符

memset 函数

- 声明：void *memset(void *str, int c, size_t n)
  - str：指向要填充的内存块。
  - c ：为每个字节所赋的值。该值以 int 形式传递，但是函数在填充内存块时是使用该值的无符号字符形式。默认 0。
  - n ：内存块大小，单位是字节。
- 特点：
  - 它是以字节为单位进行赋值的。
  - 用 memset 对非字符型数组赋初值是不可取的，因为初值并不是数字，而是对应的 ASCII 码，会造成初始数值改变。（0 不受影响）
  - 可以说是初始化内存的 “万能函数”，在统一赋值时比较适合。一般跟 sizeof 一起使用。
  - 通常是给数组或结构体进行初始化。一般的变量如 char、int、float、double 等类型的变量直接初始化即可，没有必要用 memset。
- 使用示例：memset(str, 0, sizeof(str));

malloc 函数

- 声明：extern void *malloc(unsigned int num_bytes);
  - num_bytes：内存块大小。
- 特点：
  - 需要自己计算字节数。

  - 返回指向被分配内存空间的指针，本身无类型，需要强转成指定类型的指针。
  - 只管分配内存，不进行初始化，所以新内存中值是随机的。
  - 当内存不再使用的时候，应使用 free() 函数将内存块释放掉。申请后不释放就是内存泄露。一般跟 sizeof 一起使用。
- 使用示例：p=(int*)malloc(sizeof(int)*128);

new 运算符

- 声明：无
- 特点：
  - new 返回指定类型的指针。
  - new 可以自动计算所需要的大小。
  - 会进行初始化。
  - 开辟新空间用。
- 使用示例：p=new int;

2.6.11 附注：字符串复制函数

memcpy 函数

- 声明：void *memcpy(void *str1, const void *str2, size_t n)
  - str1：指向用于存储复制内容的目标数组，类型强制转换为 void* 指针。
  - str2：指向要复制的数据源，类型强制转换为 void* 指针。
  - n：要被复制的字节数。
- 特点：
  - 可以用来拷贝任何数据类型的对象。
  - 可以指定拷贝的数据长度（n 来指定）。
  - 会完整的复制 n 个字节, 不会因为遇到字符串结束'0' 而结束。
- 使用示例：memcpy(b,a,sizeof(b)) – （b<a）

strcpy 函数

- 声明：char* strcpy(char* dest, char* src)
  - dest：待被复制的数组指针
  - str2：要复制的字符串。
- 特点：
  - 只用于字符串复制，并且它不仅复制字符串内容之外，还会复制字符串的结束符。
  - 复制过程遇到字符串结束'
    0' 结束。
- 使用示例：p=new int;

2.6.12 附注：sprintf() 函数

int sprintf(char *str, const char *format, ...);
用途：把结果输出到指定的字符串中。
str – 这是指向一个字符数组的指针，该数组存储了要写入的字符串。
format – 这是字符串，包含了要被写入到字符串 str 的文本。与之后参数一起使用，类似于格式化输出中%d、%f 之类的用法。

2.7 注意事项

- 每一个端，具有一个 socket 指示符、一个地址信息。
- 凡是函数参数含有地址信息的，均需要强制格式化：
  (struct sockaddr *)& + 地址名
- 凡是需要网络地址参数的函数，都必须将类型强制转化为(struct sockaddr *)类型，因为函数在设计时没有考虑 sockaddr_in 类型。
- 最好 recv 函数采用 sizeof（最大值），send 采用 strlen（减少不必要的发送量）。
- socket 缓冲区：每一个 socket 在被创建之后，系统都会给它分配两个缓冲区，即输入缓冲区和输出缓冲区。一般来说，默认的输入输出缓冲区大小为 8K。

- send 函数并不是直接将数据传输到网络中，而是负责将数据写入输出缓冲区，数据从输出缓冲区发送到目标主机是由 TCP 协议完成的。数据写入到输出缓冲区之后，send 函数就可以返回了。（数据是否发送出去，是否发送成功，何时到达目标主机，都不由它负责了，而是由协议负责。）

  recv 函数同理，它是从输入缓冲区中读取数据。

- 套接字关闭的时候，输出缓冲区的数据不会丢失，会由协议发送到另一方；而输入缓冲区的数据则会丢失。

- 数据的发送和接收是独立的，并不是发送方执行一次 send，接收方就执行以此 recv。recv 函数不管发送几次，都会从输入缓冲区尽可能多的获取数据。如果发送方发送了多次信息，接收方没来得及进行 recv，则数据堆积在输入缓冲区中，取数据的时候会都取出来。换句话说，recv 并不能判断数据包的结束位置。

- 在数据进行发送的时候，需要先检查输出缓冲区的可用空间大小，如果可用空间大小小于要发送的数据长度，则 send 会被阻塞，直到缓冲区中的数据被发送到目标主机，有了足够的空间之后，send 函数才会将数据写入输出缓冲区。

- TCP 协议正在将数据发送到网络上的时候，输出缓冲区会被锁定（生产者消费者问题），不允许写入，send 函数会被阻塞，直到数据发送完，输出缓冲区解锁，此时 send 才能将数据写入到输出缓冲区。

- 函数先检查输入缓冲区，如果输入缓冲区中有数据，读取出缓冲区中的数据，否则的话，recv 函数会被阻塞，等待网络上传来数据。如果读取的数据长度小于输出缓冲区中的数据长度，没法一次性将所有数据读出来，需要多次执行 recv 函数，才能将数据读取完毕。

3

函数概述:

- 功能：

- 返回值：

- 范围限制：
- 注意：

函数声明：

-

  使用示例：

-
-
-