

# 03-26模拟面试（测试/测试开发）

---

## 简要介绍一下你自己

---

## 树洞项目

---

讲一下你在你的那个树洞项目中主要负责哪些任务？

## 简单介绍一下你那个树洞项目

用了哪些技术栈、用的什么数据库、获得了什么奖项、用的什么平台、用的什么框架

## 博客项目

---

介绍一下你的博客网站项目

聊一下Django

Django的设计模式是什么？

MVC

Model

View

Controller

## Django默认的数据库是什么？

sqlite

## Django中的视图层有哪两种类型？

基于函数的视图

基于类的视图

## Django中控制页面路由是哪个文件模块？

urls.py

## 网络

---

### 聊一下TCP/IP体系结构

TCP/IP体系结构是一种计算机网络协议体系结构，由传输控制协议（TCP）和互联网协议（IP）组成。它定义了一系列协议，用于在互联网上进行数据通信。

TCP/IP体系结构分为四个层次：

1. 应用层：应用层协议定义了应用程序之间的通信规则，包括HTTP、FTP、SMTP等协议。
2. 传输层：传输层协议提供了端到端的数据传输服务，包括TCP和UDP协议。
3. 网络层：网络层协议定义了数据在网络上的传输规则，包括IP协议。
4. 链路层：链路层协议定义了数据在物理链路上的传输规则，包括以太网、ATM、PPP等协议。

TCP/IP体系结构是互联网的核心协议体系结构，它的设计思想是灵活、可扩展的，已经成为了计算机网络领域最为重要的协议体系结构之一。

# 聊一下DNS

DNS (Domain Name System, 域名系统) , 因特网上作为**域名和IP地址相互映射**的一个**分布式数据库**, 能够使用户更方便的访问互联网, 而不用去记住能够被机器直接读取的IP数串。

通过主机名, 最终得到该主机名对应的IP地址的过程叫做域名解析 (或主机名解析) 。

## DNS的工作原理?

将主机域名转换为ip地址, 属于应用层协议, 使用UDP传输。

## 为什么使用UDP使用UDP的原因主要是因为UDP有以下特点:

1. 速度快: UDP不需要建立连接, 数据包发送和接收的开销较小, 因此速度较快。
2. 轻量级: UDP头部只有8个字节, 相对于TCP的20个字节, UDP的头部更加轻量级。
3. 不可靠: UDP不需要建立连接, 也不保证数据的可靠性, 因此在传输过程中可能会出现数据丢失或者乱序的情况。

使用UDP的场景主要是在数据传输速度要求较高, 但可靠性要求较低的情况下, 例如视频流传输、游戏数据传输等。

相比之下, TCP具有以下特点:

1. 可靠性高: TCP通过建立连接、数据传输、确认等机制, 保证数据的可靠性, 数据传输过程中不会出现数据丢失或者乱序的情况。
2. 重量级: TCP头部相对于UDP较为复杂, 需要建立连接、维护状态等, 因此头部较重。
3. 速度慢: TCP需要建立连接、确认等过程, 因此在传输速度上相对于UDP较慢。

使用TCP的场景主要是在数据传输可靠性要求较高的情况下, 例如文件传输、网页浏览等。

总的来说，TCP和UDP各有优缺点，在不同的场景下使用不同的协议可以更好地满足需求。P? 聊一下TCP和UDP以及他们的使用场景

## HTTP请求方法你知道哪些？

序号	方法	描述
1	GET	请求指定的页面信息，并返回实体主体。
2	HEAD	类似于 GET 请求，只不过返回的响应中没有具体的内容，用于获取报头
3	POST	向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。
4	PUT	从客户端向服务器传送的数据取代指定的文档的内容。
5	DELETE	请求服务器删除指定的页面。
6	CONNECT	HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。
7	OPTIONS	允许客户端查看服务器的性能。
8	TRACE	回显服务器收到的请求，主要用于测试或诊断。
9	PATCH	是对 PUT 方法的补充，用来对已知资源进行局部更新。

## GET和POST的区别？简单聊一下

1. get是获取数据，post是修改数据
2. get把请求的数据放在url上，以?分割URL和传输数据，参数之间以&相连，所以get不太安全。而post把数据放在HTTP的包体内（request body 相对安全）
3. get提交的数据最大是2k（限制实际上取决于浏览器），post理论上没有限制。

4. GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据); POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。
5. GET请求会被浏览器主动缓存，而POST不会，除非手动设置。
6. 本质区别：GET是幂等的，而POST不是幂等的

## HTTP和HTTPS的区别

1、HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全， HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

2、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

## 聊一下TCP中的三次握手和四次挥手

TCP是一种面向连接的协议，为了建立可靠的连接，TCP使用了三次握手和四次挥手机制。

三次握手：

1. 客户端向服务端发送SYN报文，表示请求建立连接。
2. 服务端接收到SYN报文后，回复一个SYN+ACK报文，表示接受连接请求。
3. 客户端接收到服务端的SYN+ACK报文后，回复一个ACK报文，表示连接建立成功。

这样，客户端和服务端就完成了连接的建立。

四次挥手：

1. 客户端向服务端发送FIN报文，表示要关闭连接。
2. 服务端接收到FIN报文后，回复一个ACK报文，表示接受关闭请求。
3. 服务端发送一个FIN报文，表示要关闭连接。
4. 客户端接收到服务端的FIN报文后，回复一个ACK报文，表示接受关闭请求。

这样，客户端和服务端就完成了连接的关闭。

三次握手和四次挥手的目的是为了保证连接的可靠性和稳定性。通过三次握手，可以确保客户端和服务端都能够正确地接收到对方的请求和回复，从而建立可靠的连接；通过四次挥手，可以确保客户端和服务端都能够正确地关闭连接，避免出现数据丢失或者乱序的情况。

## 三次握手建立连接的过程中可以发送其他数据吗？为什么

---

### TCP头部中有哪些信息？

- 序号 (32bit)：传输方向上字节流的字节编号。初始时序号会被设置一个随机的初始值 (ISN)，之后每次发送数据时，序号值 = ISN + 数据在整个字节流中的偏移。假设A -> B且ISN = 1024，第一段数据512字节已经到B，则第二段数据发送时序号为1024 + 512。用于解决网络包乱序问题。
- 确认号 (32bit)：接收方对发送方TCP报文段的响应，其值是收到的序号值 + 1。
- 首部长 (4bit)：标识首部有多少个4字节 \* 首部长，最大为15，即60字节。
- 标志位 (6bit)：
  - URG：标志紧急指针是否有效。
  - ACK：标志确认号是否有效（确认报文段）。用于解决丢包问题。
  - PSH：提示接收端立即从缓冲读走数据。
  - RST：表示要求对方重新建立连接（复位报文段）。
  - SYN：表示请求建立一个连接（连接报文段）。
  - FIN：表示关闭连接（断开报文段）。
- 窗口 (16bit)：接收窗口。用于告知对方（发送方）本方的缓冲还能接收多少字节数据。用于解决流控。
- 校验和 (16bit)：接收端用CRC检验整个报文段有无损坏。

## 应用层常见协议你知道哪些？

协议	名称	默认端口	底层协议
HTTP	超文本传输协议	80	TCP
HTTPS	超文本传输安全协议	443	TCP
Telnet	远程登录服务的标准协议	23	TCP
FTP	文件传输协议	20传输和21连接	TCP
TFTP	简单文件传输协议	69	UDP
SMTP	简单邮件传输协议（发送用）	25	TCP
POP	邮局协议（接收用）	110	TCP
DNS	域名解析服务	53	服务器间进行域传输的时候用TCP 客户端查询DNS服务器时用UDP

## 操作系统

### 聊一下进程、线程和协程

进程	线程	协程	
定义	资源分配和拥有的基本单位	程序执行的基本单位	用户态的轻量级线程，线程内部调度的基本单位
切换情况	进程CPU环境(栈、寄存器、页表和文件句柄等)的保存以及新调度的进程CPU环境的设置	保存和设置程序计数器、少量寄存器和栈的内容	先将寄存器上下文和栈保存，等切换回来的时候再进行恢复
切换者	操作系统	操作系统	用户
切换过程	用户态->内核态->用户态	用户态->内核态->用户态	用户态(没有陷入内核)
调用栈	内核栈	内核栈	用户栈
拥有资源	CPU资源、内存资源、文件资源和句柄等	程序计数器、寄存器、栈和状态字	拥有自己的寄存器上下文和栈
并发性	不同进程之间切换实现并发，各自占有CPU实现并行	一个进程内部的多个线程并发执行	同一时间只能执行一个协程，而其他协程处于休眠状态，适合对任务进行分时处理



进程	线程	协程	
系统开销	切换虚拟地址空间，切换内核栈和硬件上下文，CPU高速缓存失效、页表切换，开销很大	切换时只需保存和设置少量寄存器内容，因此开销很小	直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快
通信方面	进程间通信需要借助操作系统	线程间可以直接读写进程数据段(如全局变量)来进行通信	共享内存、消息队列

## 线程与进程的区别？

- 1、线程启动速度快，轻量级
- 2、线程的系统开销小
- 3、线程使用有一定难度，需要处理数据一致性问题
- 4、同一线程共享的有堆、全局变量、静态变量、指针，引用、文件等，而独自占有栈

## 外中断和异常有什么区别？

外中断是指由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

而异常时由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

## 进程调度算法你了解多少？

## 先来先服务 (FCS)

非抢占式的调度算法，按照请求的顺序进行调度。

有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长

## 短作业优先 (SJF)

非抢占式的调度算法，按估计运行时间最短的顺序进行调度。

长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

## 最短剩余时间优先 (SRTN)

最短作业优先的抢占式版本，按剩余运行时间的顺序进行调度。当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。

如果新的进程需要的时间更少，则挂起当前进程，运行新的进程。否则新的进程等待。

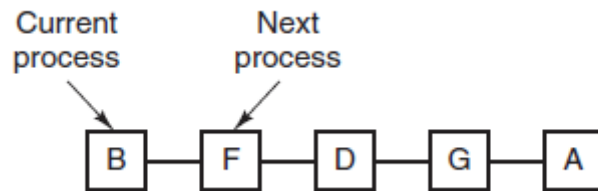
## 时间片轮转

将所有就绪进程按 FCFS 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。

当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系：

- 因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。
- 而如果时间片过长，那么实时性就不能得到保证。



## 优先级调度

为每个进程分配一个优先级，按优先级进行调度。

为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

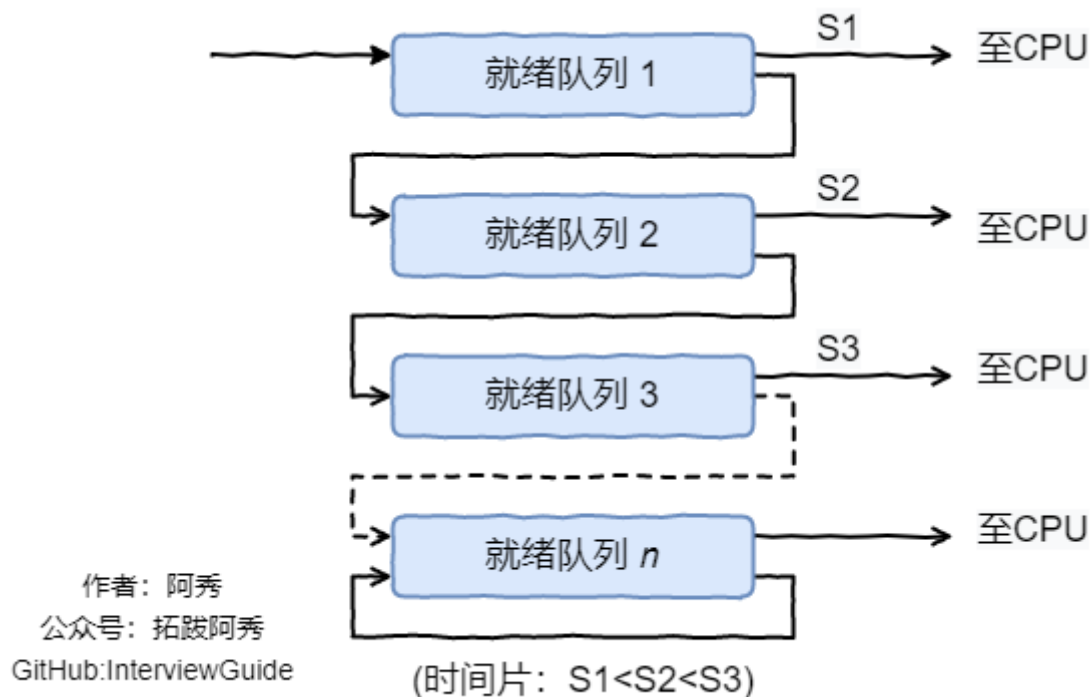
## 多级反馈队列

一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。

多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，例如 1,2,4,8,...。进程在第一个队列没执行完，就会被移到下一个队列。

这种方式下，之前的进程只需要交换 7 次。每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。



## Linux下进程间通信方式？

- 管道：
  - 无名管道（内存文件）：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之间使用。进程的亲缘关系通常是指父子进程关系。
  - 有名管道（FIFO文件，借助文件系统）：有名管道也是半双工的通信方式，但是允许在没有亲缘关系的进程之间使用，管道是先进先出的通信方式。
- 共享内存：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。它往往与信号量，配合使用来实现进程间的同步和通信。
- 消息队列：消息队列是有消息的链表，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。
- 套接字：适用于不同机器间进程通信，在本地也可作为两个进程通信的方式。
- 信号：用于通知接收进程某个事件已经发生，比如按下ctrl + C就是信号。

- 信号量：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，实现进程、线程的对临界区的同步及互斥访问。

## 进程同步的四种方法？

### 1. 临界区

对临界资源进行访问的那段代码称为临界区。

为了互斥访问临界资源，每个进程在进入临界区之前，需要先进行检查。

### 2. 同步与互斥

- 同步：多个进程因为合作产生的直接制约关系，使得进程有一定的先后执行关系。
- 互斥：多个进程在同一时刻只有一个进程能进入临界区。

### 3. 信号量

信号量（Semaphore）是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。

- **down**：如果信号量大于 0，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；
- **up**：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。

down 和 up 操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为 0 或者 1，那么就成为了 **互斥量（Mutex）**，0 表示临界区已经加锁，1 表示临界区解锁。

### 4. 管程

使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用更容易。

c 语言不支持管程，下面的示例代码使用了类 Pascal 语言来描述管程。示例代码的管程提供了 insert() 和 remove() 方法，客户端代码通过调用这两个方法来解决生产者-消费者问题。

管程有一个重要特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用管程，否则其它进程永远不能使用管程。

管程引入了 **条件变量** 以及相关的操作：**wait()** 和 **signal()** 来实现同步操作。对条件变量执行 wait() 操作会导致调用进程阻塞，把管程让出来给另一个进程持有。signal() 操作用于唤醒被阻塞的进程。

## 线程通信方法？

<b>Linux:</b>
信号：类似进程间的信号处理
锁机制：互斥锁、读写锁和自旋锁
条件变量：使用通知的方式解锁，与互斥锁配合使用
信号量：包括无名线程信号量和命名线程信号量
<b>Windows:</b>
全局变量：需要有多个线程来访问一个全局变量时，通常我们会在这个全局变量前加上volatile声明，以防编译器对此变量进行优化
Message消息机制：常用的Message通信的接口主要有两个：PostMessage和PostThreadMessage，PostMessage为线程向主窗口发送消息。而PostThreadMessage是任意两个线程之间的通信接口。
CEvent对象：CEvent为MFC中的一个对象，可以通过对CEvent的触发状态进行改变，从而实现线程间的通信和同步，这个主要是实现线程直接同步的一种方法。

# 聊聊哲学家进餐问题

五个哲学家围着一张圆桌，每个哲学家面前放着食物。哲学家的生活有两种交替活动：吃饭以及思考。当一个哲学家吃饭时，需要先拿起自己左右两边的两根筷子，并且一次只能拿起一根筷子。

下面是一种错误的解法，如果所有哲学家同时拿起左手边的筷子，那么所有哲学家都在等待其它哲学家吃完并释放自己手中的筷子，导致死锁。

```
1  #define N 5
2
3  void philosopher(int i) {
4      while(TRUE) {
5          think();
6          take(i);          // 拿起左边的筷子
7          take((i+1)%N);    // 拿起右边的筷子
8          eat();
9          put(i);
10         put((i+1)%N);
11     }
12 }
13
```

为了防止死锁的发生，可以设置两个条件：

- 必须同时拿起左右两根筷子；
- 只有在两个邻居都没有进餐的情况下才允许进餐。

```
1  #define N 5
2  #define LEFT (i + N - 1) % N // 左邻居
3  #define RIGHT (i + 1) % N    // 右邻居
4  #define THINKING 0
5  #define HUNGRY 1
6  #define EATING 2
7  typedef int semaphore;
8  int state[N];                // 跟踪每个哲学家的状态
9  semaphore mutex = 1;         // 临界区的互斥，临界区是
                                // state 数组，对其修改需要互斥
10 semaphore s[N];              // 每个哲学家一个信号量
```

```
11
12 void philosopher(int i) {
13     while(TRUE) {
14         think(i);
15         take_two(i);
16         eat(i);
17         put_two(i);
18     }
19 }
20
21 void take_two(int i) {
22     down(&mutex);
23     state[i] = HUNGRY;
24     check(i);
25     up(&mutex);
26     down(&s[i]); // 只有收到通知之后才可以开始吃，否则会一直等
                下去
27 }
28
29 void put_two(i) {
30     down(&mutex);
31     state[i] = THINKING;
32     check(LEFT); // 尝试通知左右邻居，自己吃完了，你们可以开始
                吃了
33     check(RIGHT);
34     up(&mutex);
35 }
36
37 void eat(int i) {
38     down(&mutex);
39     state[i] = EATING;
40     up(&mutex);
41 }
42
43 // 检查两个邻居是否都没有用餐，如果是的话，就 up(&s[i])，使得
    down(&s[i]) 能够得到通知并继续执行
44 void check(i) {
45     if(state[i] == HUNGRY && state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
46         state[i] = EATING;
```



```
47         up(&s[i]);  
48     }  
49 }  
50
```

## 讲一下你知道哪些锁？

### 读写锁

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

### 互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃CPU进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阈值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

### 条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。总的来说**互斥锁是线程间互斥的机制，条件变量则是同步机制。**

# 自旋锁

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。如果别的线程长时期占有锁，那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高。

## 测试理论

---

**讲一下你对于软件测试的了解？**

**讲一下你知道的软件开发模型**

1. 瀑布模型
2. 敏捷开发模型
3. DevOps模型

**软件测试的作用是什么？ 软件测试的一般流程？**

书籍P23

**讲一下你知道的软件测试模型**

**讲一下软件测试分类**

## C++语言

---

**结构体内存占用问题**

**讲一下指针和引用的区别**

- 指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名
- 指针可以有多级，引用只有一级
- 指针可以为空，引用不能为NULL且在定义时必须初始化
- 指针在初始化后可以改变指向，而引用在初始化之后不可再改变

- sizeof指针得到的是本指针的大小，sizeof引用得到的是引用所指向变量的大小
- 当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。
- 引用本质是一个指针，同样会占4字节内存；指针是具体变量，需要占用存储空间（，具体情况还要具体分析）。
- 引用在声明时必须初始化为另一变量，一旦出现必须为typename refname &varname形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等用到时再指向具体变量。
- 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
- 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

## 讲一下你对于网络中的大小端存储的了解 (\*)

大端存储：字数据的高字节存储在低地址中

小端存储：字数据的低字节存储在低地址中

例如：32bit的数字0x12345678

**所以在Socket编程中，往往需要将操作系统所用的小端存储的IP地址转换为大端存储，这样才能进行网络传输**

## 讲一下C++的三大特性

三大特性：继承、封装和多态

### (1) 继承

**让某种类型对象获得另一个类型对象的属性和方法。**

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展

常见的继承有三种方式：

1. 实现继承：指使用基类的属性和方法而无需额外编码的能力

2. 接口继承：指仅使用属性和方法的名称、但是子类必须提供实现的能力
3. 可视继承：指子窗体（类）使用基窗体（类）的外观和实现代码的能力（C++里好像不怎么用）

例如，将人定义为一个抽象类，拥有姓名、性别、年龄等公共属性，吃饭、睡觉、走路等公共方法，在定义一个具体的人时，就可以继承这个抽象类，既保留了公共属性和方法，也可以在此基础上扩展跳舞、唱歌等特有方法

## (2) 封装

数据和代码捆绑在一起，避免外界干扰和不确定性访问。

封装，也就是**把客观事物封装成抽象的类**，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏，例如：将公共的数据或方法使用public修饰，而不希望被访问的数据或方法采用private修饰。

## (3) 多态

同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为（**重载实现编译时多态，虚函数实现运行时多态**）。

多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。**简单一句话：允许将子类类型的指针赋值给父类类型的指针**

实现多态有二种方式：覆盖（override），重载（overload）。

覆盖：是指子类重新定义父类的虚函数的做法。

重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。例如：基类是一个抽象对象——人，那教师、运动员也是人，而使用这个抽象对象既可以表示教师、也可以表示运动员。

# 算法题目

---

## 11. 盛最多水的容器

提示

中等



4.2K



— 亚马逊

字节跳动

微软 Microsoft



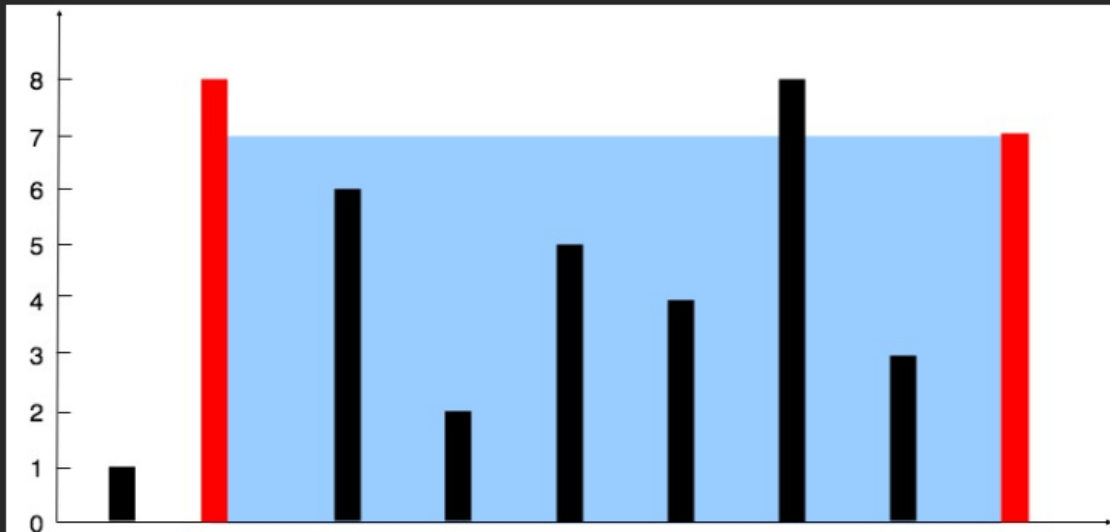
给定一个长度为  $n$  的整数数组 `height`。有  $n$  条垂线，第  $i$  条线的两个端点是  $(i, 0)$  和  $(i, \text{height}[i])$ 。

找出其中的两条线，使得它们与  $x$  轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例 1:



输入: [1,8,6,2,5,4,8,3,7]

输出: 49

解释: 图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

```
1 class Solution {
2     public:
3         int maxArea(vector<int>& height)
4         {
5             int l=0,r=height.size()-1,area=0,res=0;
6             while(l<r)
7             {
8                 area = min(height[l],height[r])*(r-l);
9                 res=max(res,area);
10                if(height[l]<height[r])
11                {
12                    l++;
13                }
14                else
15                {
16                    r--;
```

```

17         }
18     }
19     return res;
20 }
21 };

```

## 70. 爬楼梯

提示

简单



2.9K



— 亚马逊

谷歌 Google

微软 Microsoft

...

假设你正在爬楼梯。需要  $n$  阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

示例 1:

输入:  $n = 2$

输出: 2

解释: 有两种方法可以爬到楼顶。

1. 1 阶 + 1 阶

2. 2 阶

示例 2:

输入:  $n = 3$

输出: 3

解释: 有三种方法可以爬到楼顶。

1. 1 阶 + 1 阶 + 1 阶

2. 1 阶 + 2 阶

3. 2 阶 + 1 阶

```

1  class solution {
2  public:
3      int climbStairs(int n) {
4          if(n==1)
5          {
6              // 直接返回1 上到1楼只有种
7              return 1;
8          }
9          if(n==2)
10         {
11             return 2;
12         }
13         int s[n];
14         s[0] = 1;
15         s[1] = 2;
16         for(int i=2;i<n;++i)

```


```

17         {
18             s[i]=s[i-1]+s[i-2];
19         }
20         return s[n-1];
21     }
22 };

```

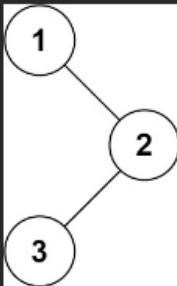
#### 94. 二叉树的中序遍历

简单  1.8K  

— 亚马逊  字节跳动  微软 Microsoft ...

给定一个二叉树的根节点 `root`，返回它的中序遍历。

示例 1:



输入: `root = [1,null,2,3]`  
输出: `[1,3,2]`

示例 2:

输入: `root = []`  
输出: `[]`

```

1  class solution {
2  public:
3      void subprocess(TreeNode* cur,vector<int>& vec)
4      {
5          if(cur==nullptr)
6          {
7              return;
8          }
9          subprocess(cur->left,vec); //左
10         vec.push_back(cur->val);    //中
11         subprocess(cur->right,vec); //右
12     }
13
14     vector<int> inorderTraversal(TreeNode* root)

```

```

15     {
16         vector<int> res;
17         subprocess(root,res);    //调用循环子进程
18         return res;
19     }
20 };

```

## 141. 环形链表

简单



1.8K



— 亚马逊

字节跳动

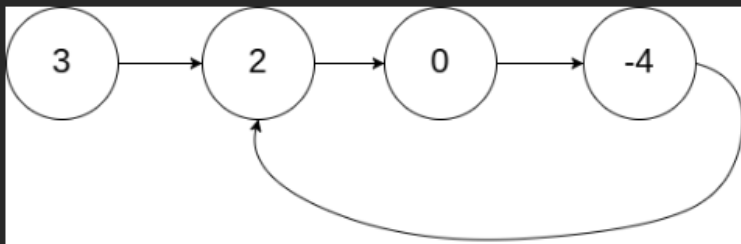


给你一个链表的头节点 `head`，判断链表中是否有环。

如果链表中有某个节点，可以通过连续跟踪 `next` 指针再次到达，则链表中存在环。为了表示给定链表中的环，评测系统内部使用整数 `pos` 来表示链表尾连接到链表中的位置（索引从 0 开始）。注意：`pos` 不作为参数进行传递。仅仅是为了标识链表的实际情况。

如果链表中存在环，则返回 `true`。否则，返回 `false`。

示例 1:

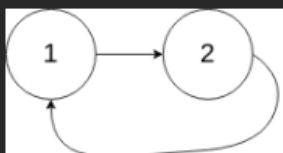


输入: `head = [3,2,0,-4]`, `pos = 1`

输出: `true`

解释: 链表中有一个环，其尾部连接到第二个节点。

示例 2:



输入: `head = [1,2]`, `pos = 0`

输出: `true`

解释: 链表中有一个环，其尾部连接到第一个节点。

```

1  /**
2   * Definition for singly-linked list.
3   * struct ListNode {
4   *     int val;
5   *     ListNode *next;
6   *     ListNode(int x) : val(x), next(NULL) {}
7   * };
8   */

```



```
9  class solution {
10  public:
11      bool hasCycle(ListNode *head)
12      {
13          //双指针法 （快慢指针）
14          ListNode* fast = head;
15          ListNode* slow = head;
16          while(fast!=nullptr && fast->next!=nullptr)
17          {
18              slow=slow->next;    //慢指针一次走一步
19              fast=fast->next->next; //快指针一次走两步
20              if(slow == fast) return true;
21          }
22          return false;
23      }
24  };
```