

2022 年华为精选 50 面试题及答案

1. static 有什么用途？（请至少说明两种）

- 1) 在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。
- 2) 在模块内（但在函数体外），一个被声明为静态的变量可以被模块内所用函数访问，但不能被模块外其它函数访问。它是一个本地的全局变量。
- 3) 在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用

2. 引用与指针有什么区别？

- 1) 引用必须被初始化，指针不必。
- 2) 引用初始化以后不能被改变，指针可以改变所指的对象。
- 3) 不存在指向空值的引用，但是存在指向空值的指针。

3. 描述实时系统的基本特性

在特定时间内完成特定的任务，实时性与可靠性。

4. 全局变量和局部变量在内存中是否有区别？如果有，是什么区别？

全局变量储存在静态数据库，局部变量在堆栈。

5. 什么是平衡二叉树？

左右子树都是平衡二叉树 且左右子树的深度差值的绝对值不大于 1。

6. 堆栈溢出一般是由什么原因导致的？

没有回收垃圾资源。

7. 什么函数不能声明为虚函数？

constructor 函数不能声明为虚函数。

8. 冒泡排序算法的时间复杂度是什么？

时间复杂度是 $O(n^2)$ 。

9. Internet 采用哪种网络协议？该协议的主要层次结构？

Tcp/Ip 协议

主要层次结构为：应用层/传输层/网络层/数据链路层/物理层。

10. IP 地址的编码分为哪两部分？

IP 地址由两部分组成，网络号和主机号。不过是要和“子网掩码”按位与上之后才能区分哪些是网络位哪些是主机位。

11. 用户输入 M,N 值，从 1 至 N 开始顺序循环数数，每数到 M 输出该数值，直至全部输出。写出 C 程序。

循环链表，用取余操作做

12. 某 32 位系统下, C++ 程序，请计算 sizeof 的值。

```
char str[] = "http://www.ibegroup.com/"
char *p = str ;
int n = 10;
```

请计算

sizeof (str) = ? (1)

sizeof (p) = ? (2)

sizeof (n) = ? (3)

```
void Foo ( char str[100]){
```

 请计算

 sizeof(str) = ? (4)

```
}
```

```
void *p = malloc( 100 );
```

请计算

sizeof (p) = ? (5)

(1) 17 (2) 4 (3) 4 (4) 4 (5) 4

13. 阅读下面代码,回答问题.

```
1). void GetMemory(char **p, int num) {  
    *p = (char *)malloc(num);  
}
```

```
void Test(void) {  
    char *str = NULL;  
    GetMemory(&str, 100);  
    strcpy(str, "hello");  
    printf(str);  
}
```

请问运行 Test 函数会有什么样的结果?

输出 “hello”

```
2). char *GetMemory(void) {  
    char p[] = "hello world";  
    return p;  
}
```

```
void Test(void) {  
    char *str = NULL;  
    str = GetMemory();  
    printf(str);  
}
```

请问运行 Test 函数会有什么样的结果?

无效的指针, 输出不确定

14. C++中为什么用模板类。

- (1) 可用来创建动态增长和减小的数据结构
- (2) 它是类型无关的，因此具有很高的可复用性。
- (3) 它在编译时而不是运行时检查数据类型，保证了类型安全
- (4) 它是平台无关的，可移植性
- (5) 可用于基本数据类型

15. 程序什么时候应该使用线程，什么时候单线程效率高。

- 1. 耗时的操作使用线程，提高应用程序响应
 - 2. 并行操作时使用线程，如 C/S 架构的服务器端并发线程响应用户的请求。
 - 3. 多 CPU 系统中，使用线程提高 CPU 利用率
 - 4. 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。
- 其他情况都使用单线程。

16. Linux 有内核级线程吗？

线程通常被定义为一个进程中代码的不同执行路线。从实现方式上划分，线程有两种类型：“用户级线程”和“内核级线程”。

用户线程指不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。这种线程甚至在象 DOS 这样的操作系统中也可实现，但线程的调度需要用户程序完成，这有些类似 Windows 3.x 的协作式多任务。另外一种则需要内核的参与，由内核完成线程的调度。其依赖于操作系统核心，由内核的内部需求进行创建和撤销，这两种模型各有其好处和缺点。用户线程不需要额外的内核开支

，并且用户态线程的实现方式可以被定制或修改以适应特殊应用的要求，但是当个线程因 I/O 而处于等待状态时，整个进程就会被调度程序切换为等待状态，其他线程得不到运行的机会；而内核线程则没有各个限制，有利于发挥多处理器的并发优势，但却占用了更多的系统开支。

Windows NT 和 OS/2 支持内核线程。Linux 支持内核级的多线程。

17. C++中什么数据分配在栈或堆中，New 分配数据是在近堆还是远堆中？

栈：存放局部变量，函数调用参数，函数返回值，函数返回地址。由系统管理
堆：程序运行时动态申请，new 和 malloc 申请的内存就在堆上。

18. 使用线程是如何防止出现大的波峰。

意思是如何防止同时产生大量的线程，方法是使用线程池，线程池具有可以同时提高调度效率和限制资源使用的好处，线程池中的线程达到最大数时，其他线程就会排队等候。

19. 函数模板与类模板有什么区别？

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。

20. winsock 建立连接的主要实现步骤？

服务器端：socket() 建立套接字，绑定(bind)并监听(listen)，用 accept() 等待客户端连接。

客户端：socket() 建立套接字，连接(connect)服务器，连接上后使用 send() 和 recv()，在套接字上写读数据，直至数据交换完毕，closesocket() 关闭套接字。

服务器端：accept() 发现有客户端连接，建立一个新的套接字，自身重新开始等待连接。该新产生的套接字使用 send() 和 recv() 写读数据，直至数据交换完毕，closesocket() 关闭套接字。

21. 动态连接库的两种方式？

调用一个 DLL 中的函数有两种方法：

1. 载入时动态链接(load-time dynamic linking)，模块非常明确调用某个导出函数，使得他们就像本地函数一样。这需要链接时链接那些函数所在 DLL 的导入库，导入库向系统提供了载入 DLL 时所需的信息及 DLL 函数定位。
2. 运行时动态链接(run-time dynamic linking)，运行时可以通过 LoadLibrary 或 LoadLibraryEx 函数载入 DLL。DLL 载入后，模块可以通过调

用 GetProcAddress 获取 DLL 函数的出口地址，然后就可以通过返回的函数指针调用 DLL 函数了。如此即可避免导入库文件了。

22. IP 组播有那些好处？

Internet 上产生的许多新的应用，特别是高带宽的多媒体应用，带来了带宽的急剧消耗和网络拥挤问题。组播是一种允许一个或多个发送者（组播源）发送单一的数据包到多个接收者（一次的，同时的）的网络技术。组播可以大大的节省网络带宽，因为无论有多少个目标地址，在整个网络的任何一条链路上只传送单一的数据包。所以说组播技术的核心就是针对如何节约网络资源的前提下保证服务质量。

23. 列举几种进程的同步机制及优缺点

1) **信号量机制**： 一个信号量只能置一次初值，以后只能对之进行 p 操作或 v 操作。

由此也可以看到，信号量机制必须有公共内存，**不能用于分布式操作系统，这是它最大的弱点。**

2) **自旋锁**： 旋锁是为了保护共享资源提出的一种锁机制。

调用者申请的资源如果被占用，即自旋锁被已经被别的执行单元保持，则调用者一直循环在那里看是否该自旋锁的保持着已经释放了锁。自旋锁是一种比较低级的保护数据结构和代码片段的原始方式，可能会引起以下两个问题；

（1）死锁

（2）过多地占用 CPU 资源

3) **管程**： 信号量机制功能强大，但使用时对信号量的操作分散，而且难以控制，读写和维护都很困难。因此后来又提出了一种集中式同步进程——管程。其基本思想是将共享变量

和对它们的操作集中在一个模块中，操作系统或并发程序就由这样的模块构成。这样模块之间联系清晰，便于维护和修改，易于保证正确性。

4) **会合**： 进程直接进行相互作用

5) **分布式系统**： 由于在分布式操作系统中没有公共内存，因此参数全为值参，

而且不可为指针。

优缺点：

信号量（Semaphore）及 PV 操作

优：PV 操作能够实现对临界区的管理要求；实现简单；允许使用它的代码休眠，持有锁的时间可相对较长。

缺：信号量机制必须有公共内存，不能用于分布式操作系统，这是它最大的弱点。信号量机制功能强大，但使用时对信号量的操作分散，而且难以控制，读写和维护都很困难。

加重了程序员的编码负担; 核心操作 P-V 分散在各用户程序的代码中, 不易控制和管理; 一旦错误, 后果严重, 且不易发现和纠正。

自旋锁:

优: 旋锁是为了保护共享资源提出的一种锁机制; 调用者申请的资源如果被占用, 即自旋锁已经被别的执行单元保持, 则调用者一直循环在那里看是否该自旋锁的保持者

已经释放了锁; 低开销; 安全和高效;

缺: 自旋锁是一种比较低级的保护数据结构和代码片段的原始方式, 可能会引起以下两个问题;

(1) 死锁

(2) 过多地占用 CPU 资源

传统自旋锁由于无序竞争会导致“公平性”问题

管程:

优: 集中式同步进程——管程。其基本思想是将共享变量和对它们的操作集中在一个模块中, 操作系统或并发程序就由这样的模块构成。这样模块之间联系清晰, 便于维护和修改, 易于保证正确性。

缺: 如果一个分布式系统具有多个 CPU, 并且每个 CPU 拥有自己的私有内存, 它们通过一个局域网相连, 那么这些原语将失效。而管程在少数几种编程语言之外又无

法使用, 并且, 这些原语均未提供机器间的信息交换方法。

会合: 进程直接进行相互作用

分布式系统: 消息和 rpc

由于在分布式操作系统中没有公共内存, 因此参数全为值参, 而且不可为指针

24. 什么是预编译,何时需要预编译?

(1) 总是使用不经常改动的大型代码体

(2) 程序由多个模块组成, 所有模块都使用一组标准的包含文件和相同的编译选项。在这种

情况下, 可以将所有包含文件预编译为一个预编译头

<<预编译又称为预处理, 是做些代码文本的替换工作

处理#开头的指令, 比如拷贝# include 包含的文件代码, # define 宏定义的替换, 条件编译等

就是为编译做的预备工作的阶段

主要处理#开始的预编译指令

预编译指令指示了在程序正式编译前就由编译器进行的操作, 可以放在程序中的任何位置。常见的预编译指令有:>>

25. `int (*s[10])(int)`表示的是什么?

`int (*s[10])(int)` 函数指正数组, 每个指正指向一个 `int func(int param)` 的函数.

26. 交换两个变量的值,不使用第三个变量.即 `a=3,b=5`, 交换后 `a=5,b=3`.

```
Internet 上产生的许多有两种解放, 一种用算术算法, 一种用 ^ (异或)
a=a+b;
b=a-b;
a=a-b;
或者
a=a^b; //只能对 int, char..
b=a^b;
a=a^b
```

27. 要对绝对地址 `0x100000` 赋值,我们可以用 `unsigned int)0x100000=1234`;那么要是想让程序跳转到绝对地址是 `0x100000` 去执行,应该怎么做?

```
*((void(*)())0x100000)();
先将 0x100000 强制转换成函数指针即: (void(*)())0x100000。然后再调用它:
*((void(*)())0x100000)(); 用 typedef 可以看得更直观些:
typedef void(*)() voidFuncPtr;
*((voidFuncPtr)0x100000)();
```

28. 线程与进程的区别和联系?线程是否具有相同的堆栈?d 是否有独立的堆栈?

进程是死的, 只是一些资源的集合, 真正的程序执行都是线程来完成的, 程序启动的时候操作系统就帮你创建线程。每个线程有自己的堆栈。DLL 中有没有独立的堆栈, 这个问题不好回答, 或者说这个问题本身是否有问题。因为

DLL 中的代码是被某些线程所执行;只有线程拥有堆栈,如果 DLL 中的代码是 EXE 中的线程所调用,那么这个时候是不是说这个 DLL 没有自己独立的堆栈?如果 DLL 中的代码是由 DLL 自己创建的线程所执行,那么是不是说 DLL 有独立的堆栈?

以上讲的是堆栈,如果对于堆来说,每个 DLL 有自己的堆,所以如果是从 DLL 中动态分配的内存,最好是从 DLL 中删除,如果你从 DLL 中分配内存,然后在 EXE 中,或者另外一个 DLL 中删除,很有可能导致程序崩溃.

29. 用两个栈实现一个队列的功能?要求给出算法和思路.

设 2 个栈为 AB,一开始均为空

将新元素 push 入栈 A

出队

(1)判断栈 B 是否为空;

(2)如果不为空,则将栈 A 中所有元素依次 pop 出并 push 到栈 B

(3)将栈 B 的栈顶元素 pop 出

这样实现的队列入队和出队的平摊复杂度都还是 $O(1)$.

30. 已知一个单向链表的头,请写出删除其某一个结点的算法,要求,先找到此结点,然后删除。

```
slnodetype *Delete(slnodetype *Head, int key) {
    if (Head->number==key) {
        Head=Pointer->next;
        free(Pointer);
        break;
    }
    Back = Pointer;
    Pointer = Pointer->next;
    if (Pointer->number==key) {
        Back->next=Pointer->next;
        free(Pointer);
        break;
    }
}

void delete(Node* p) {
    if (Head==Node)
        while(p)
    }
}
```

31. 堆栈溢出一般是由什么原因导致的?

1. 函数调用层次太深。函数递归调用时,系统要在栈中不断保存函数调用时的现场和产生的变量,如果递归调用太深,就会造成栈溢出,这时递归无法返回。再有,当函数调用层次过深时也可能导致栈无法容纳这些调用的返回地址而造成栈溢出。
2. 动态申请空间使用之后没有释放。由于 C 语言中没有垃圾资源自动回收机制,因此,需要程序主动释放已经不再使用的动态地址空间。申请的动态空间使用的是堆空间,动态空间使用不会造成堆溢出。
3. 数组访问越界。C 语言没有提供数组下标越界检查,如果在程序中出现数组下标访问超出数组范围,在运行过程中可能会内存访问错误。
4. 指针非法访问。指针保存了一个非法的地址,通过这样的指针访问所指向的地址时会产生内存访问错误。

32. 阅读下面代码,回答问题.

```
void GetMemory(char **p, int num){
    *p = (char *)malloc(num);
}
void Test(void){
    char *str = NULL;
    GetMemory(&str, 100);
    strcpy(str, "hello");
    printf(str);
}
main() {
    int a[5]={1, 2, 3, 4, 5};
    int *ptr=(int *)(&a+1);
    printf("%d,%d", *(a+1), *(ptr-1));
}
```

请问输出的结果是?

2, 5

解释:

*(&a+1) 就是 a[1], *(ptr-1) 就是 a[4], 执行结果是 2, 5。&a+1 不是首地址+1, 系统会认为加

一个 a 数组的偏移, 是偏移了一个数组的大小(本例是 5 个 int)。 int

ptr=(int) (&a+1); 则

ptr 实际是&(a[5]) 也就是 a+5

原因如下:

&a 是数组指针, 其类型为 int (*) [5]; 而指针加 1 要根据指针类型加上一定的值, 不同类型

的指针+1 之后增加的大小不同;a 是长度为 5 的 int 数组指针,所以要加 5*sizeof(int)。所以 ptr 实际是 a[5]。但是 prt 与 (&a+1) 类型是不一样的(这点很重要), 所以 ptr-1 只会减去 sizeof(int*)。a&a 的地址是一样的, 但意思不一样, a 是数组首地址, 也就是 a0 的地址, &a 是对象(数组)首地址, a+1 是数组下一元素的地址, 即 a[1], &a+1 是下一个对象的地址, 即 a[5]。

33. static 全局变量与普通的全局变量有什么区别? static 局部变量和普通局部变量有什么区别?

static 函数与普通函数有什么区别?

全局变量(外部变量)的说明之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式, 静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。这两者的区别虽在于非静态全局变量的作用域是整个源程序, 当一个源程序由多个源文件组成时, 非静态的全局变量在各个源文件中都是有效的;

而静态全局变量则限制了其作用域, 即只在定义该变量的源文件内有效, 在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内, 只能为该源文件内的函数公用, 因此可以避免在其它源文件中引起错误。从以上分析可以看出, 把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域, 限制了它的使用范围;

static 函数与普通函数作用域不同。仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(static), 内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数, 应该在一个头文件中说明, 要使用这些函数的源文件要包含这个头文件;

static 全局变量与普通的全局变量有什么区别: static 全局变量只初使化一次, 防止在其他文件单元中被引用;

static 局部变里和普通局部变里有什么区别: static 局部变里只被初始化一次, 下一次依据上一次结果值;

static 函数与普通函数有什么区别:static 函数在内存中只有一份, 普通函数在每个被条用中维持一份拷贝;

34. C++中为什么用模板类。

- (1) 可用来创建动态增长和减小的数据结构
- (2) 它是类型无关的, 因此具有很高的可复用性。
- (3) 它在编译时而不是运行时检查数据类型, 保证了类型安全

- (4) 它是平台无关的，可移植性
- (5) 可用于基本数据类型

35. 如何理解软件的健壮性和高可靠性。

健壮性：

健壮性具体指的是系统在不正常的输入或不正常的外部环境下仍能表现出正常的程度。

面向健壮性的编程有以下几点要求或优点：

处理未期望的行为和错误终止

即使终止执行，也要准确/无歧义的向用户展示全面的错误信息

错误信息有助于进行 debug

健壮性原则：

总是假定用户为恶意用户，假定自己的代码会失败

把用户想象成一个 silly b，可能输出任何东西

注意，因为用户很 silly，最好要返回给用户错误提示信息，而且要详细准确无歧义！（其实这对 debug 非常有帮助，尤其是像我这样喜欢用 syso 找虫子的白痴 CodeDog）

对自己的代码要保守，对用户的行为要开放

面向健壮性编程的原则：

封闭实现细节，限定用户的恶意行为

考虑各种各样的极端情况，没有 impossible

高可靠性：

高可靠性（high reliability）指的是运行时间能够满足预计时间的一个系统或组件。

在信息技术领域，高可靠性（high reliability）指的是运行时间能够满足预计时间的一个系统或组件。可靠性可以用“100%可操作性”或者“从未失败”这两种标准来表示。一个被广泛应用但却难以达到的标准是著名的“5个9标准”，就是说工作的可靠性要达到 99.999%。

由于一个计算机系统或网络由许多部件组成，而且这些部件都要保证高可靠性才能维持正常的操作过程。因此，许多可靠性计划侧重于备份、故障处理、数据存储以及访问方面。对存储而言，一个普遍采用的方法是冗余磁盘阵列，最近采用存储局域网。

一些可靠性专家强调，为保证高可靠性，系统的任何部件都要进行仔细的规划设计，并在投入运行前进行彻底的检查测试工作。比如说，一个未经彻底测试的新的应用程序在运行过程中很可能出现频繁的中断。

36. 了解哪些 linux 内核的模块。

Linux 内核的五大模块 1. 进程调度模块 2. 内存管理模块 3. 文件系统模块
4. 进程间通信模块 5. 网络接口模块

进程调度模块

用来负责控制进程对 CPU 资源的使用。所采取的调度策略是各进程能够公平合理地访问 CPU，同时保证内核能及时地执行硬件操作。

内存管理模块

用于确保所有进程能够安全地共享机器主内存区，同时，内存管理模块还支持虚拟内存管理方式，使得 Linux 支持进程使用比实际内存空间更多的内存容量。并可以利用文件系统，对暂时不用的内存数据块交换到外部存储设备上去，当需要时再交换回来。

文件系统模块

用于支持对外部设备的驱动和存储。虚拟文件系统模块通过向所有的外部存储设备提供一个通用的文件接口，隐藏了各种硬件设备不同细节。从而提供并支持与其它操作系统兼容的多种文件系统格式。

进程间通信模块

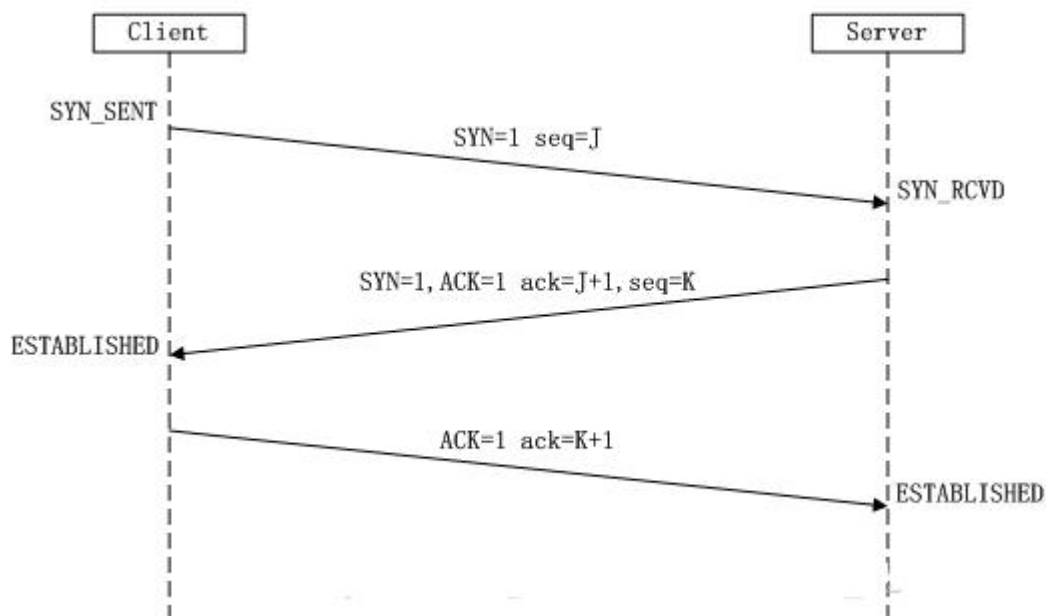
用于支持多种进程间的信息交换方式

网络接口模块

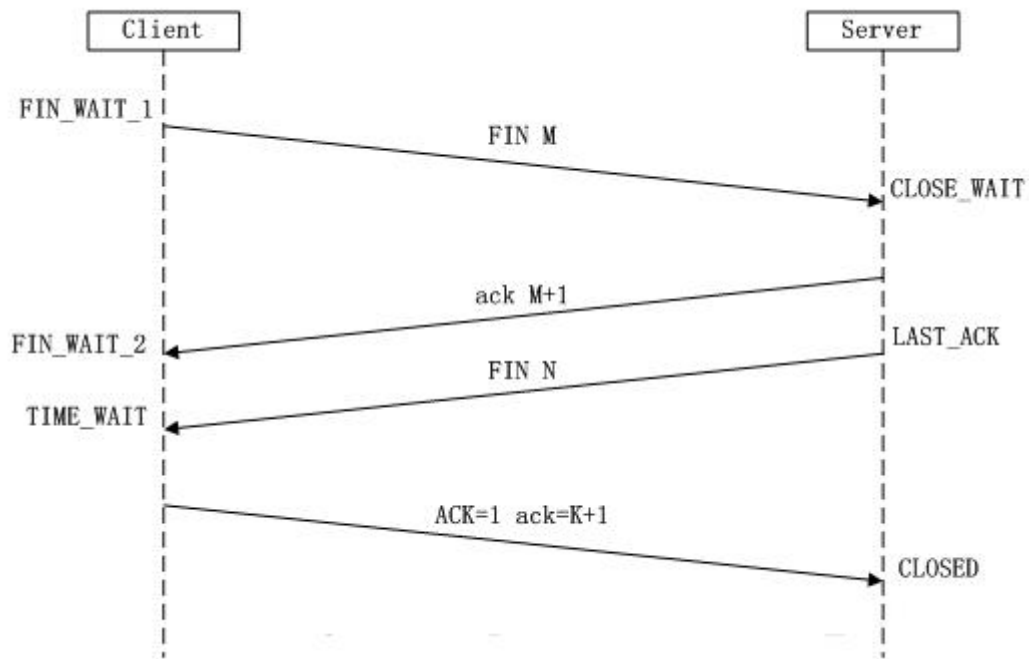
提供对多种网络通信标准的访问并支持许多网络硬件

37. 画出三次握手和四次挥手流程图。

TCP 三次握手



TCP 四次挥手



38. 请阐释 https 建立连接过程。

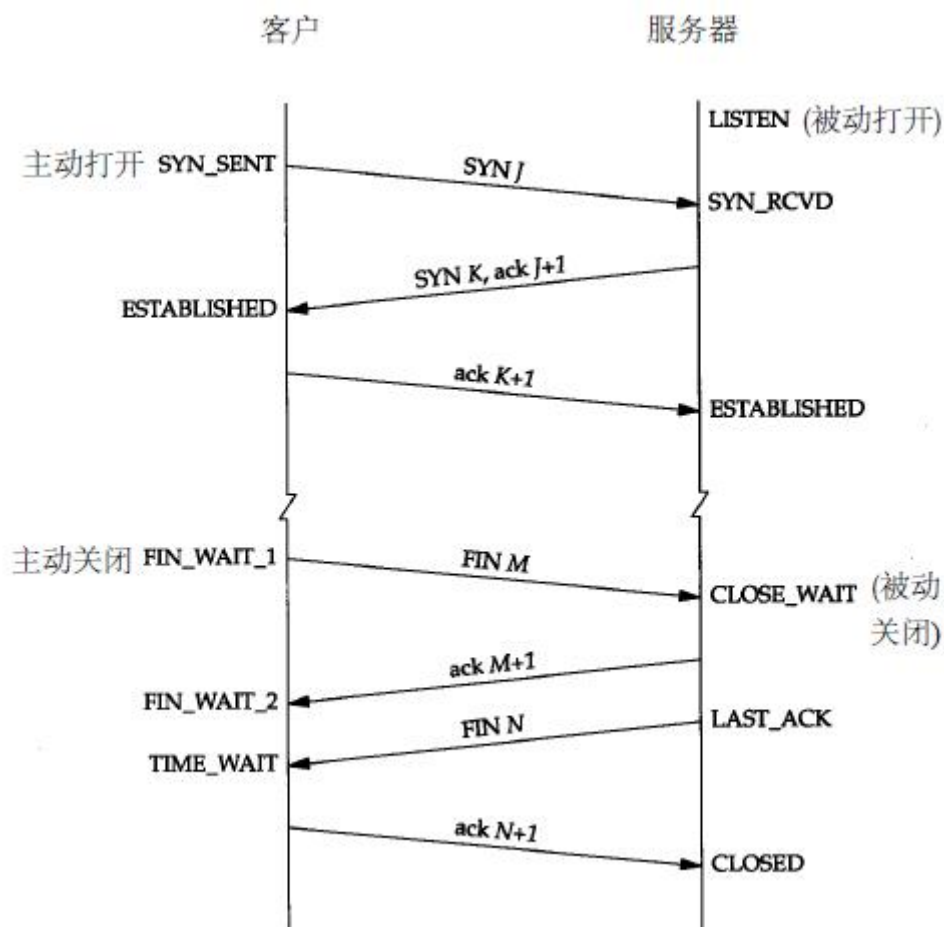
建立连接

HTTP 和 HTTPS 都需要在建立连接的基础上来进行数据传输，是基本操作。当客户在浏览器中输入网址的并且按下回车，浏览器会在浏览器 DNS 缓存，本地 DNS 缓存，和 Hosts 中寻找对应的记录，如果没有获取到则会请求 DNS 服务来获取对应的 ip。

当获取到 ip 后，tcp 连接会进行三次握手建立连接。

tcp 的三次握手和四次挥手。

过程简图



三次挥手(建立连接)

第一次：建立连接时，客户端发送 SYN 包 (syn=j) 到服务器，并进入 SYN_SENT 状态，等待服务器确认；

第二次：服务器收到 SYN 包，向客户端返回 ACK (ack=j+1)，同时自己也发送一个 SYN 包 (syn=k)，即 SYN+ACK 包，此时服务器进入 SYN_RCVD 状态；

第三次：客户端收到服务器的 SYN+ACK 包，向服务器发送确认包 ACK (ack=k+1)，此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。

完成三次握手，客户端与服务器开始传送数据，也就是 ESTABLISHED 状态。三次握手保证了不会建立无效的连接，从而浪费资源。

四次挥手(断开连接)

第一次：TCP 客户端发送一个 FIN，用来关闭客户到服务器的数据传送。

第二次：服务器收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加 1。和 SYN 一样，一个 FIN 将占用一个序号。

第三次：服务器关闭客户端的连接，发送一个 FIN 给客户端。

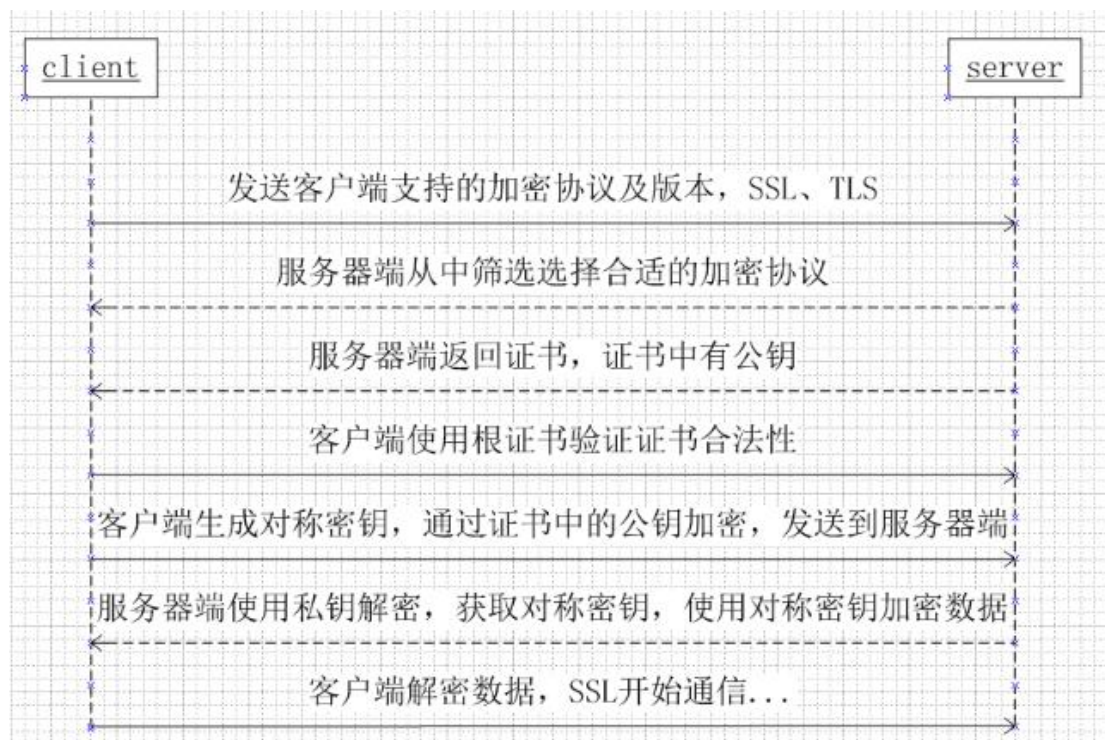
第四次：客户端发回 ACK 报文确认，并将确认序号设置为收到序号加 1。

HTTP 请求过程

建立连接完毕以后客户端会发送响应给服务端

服务端接受请求并且做出响应发送给客户端

客户端收到响应并且解析响应响应给客户



在使用 HTTPS 是需要保证服务端配置正确了对应的安全证书
客户端发送请求到服务端
服务端返回公钥和证书到客户端
客户端接收后会验证证书的安全性，如果通过则会随机生成一个随机数，用公钥对其加密，发送到服务端
服务端接受到这个加密后的随机数后会用私钥对其解密得到真正的随机数，随后用这个随机数当做私钥对需要发送的数据进行对称加密
客户端在接收到加密后的数据使用私钥(即生成的随机值)对数据进行解密并且解析数据呈现结果给客户
SSL 加密建立

39. innodb 引擎的 4 大特性。

innodb 引擎的 4 大特性 1. 插入缓冲；2. 二次写；3. 自适应哈希；4. 预读

1. 插入缓冲 (insert buffer)

插入缓冲(Insert Buffer/Change Buffer): 提升插入性能, change buffering 是 insert buffer 的加强, insert buffer 只针对 insert 有效, change buffering 对 insert、delete、update(delete+insert)、purge 都有效
只对于非聚集索引(非唯一)的插入和更新有效, 对于每一次的插入不是写到索引页中, 而是先判断插入的非聚集索引页是否在缓冲池中, 如果在则直接插入; 若不在, 则先放到 Insert Buffer 中, 再按照一定的频率进行合并操作, 再写回 disk。这样通常能将多个插入合并到一个操作中, 目的还是为了减少随机 IO 带来性能损耗。

使用插入缓冲的条件:

* 非聚集索引

* 非唯一索引

Change buffer 是作为 buffer pool 中的一部分存在。

Innodb_change_buffering 参数缓存所对应的操作: (update 会被认为是 delete+insert)

innodb_change_buffering, 设置的值有: inserts、deletes、purges、changes (inserts 和 deletes)、all (默认)、none。

all: 默认值, 缓存 insert, delete, purges 操作

none: 不缓存

inserts: 缓存 insert 操作

deletes: 缓存 delete 操作

changes: 缓存 insert 和 delete 操作

purges: 缓存后台执行的物理删除操作

可以通过参数控制其使用的大小:

innodb_change_buffer_max_size, 默认是 25%, 即缓冲池的 1/4。最大可设置为 50%。当 MySQL 实例中有大量的修改操作时, 要考虑增大

innodb_change_buffer_max_size

上面提过在一定频率下进行合并, 那所谓的频率是什么条件?

1) 辅助索引页被读取到缓冲池中。正常的 select 先检查 Insert Buffer 是否有该非聚集索引页存在, 若有则合并插入。

2) 辅助索引页没有可用空间。空间小于 1/32 页的大小, 则会强制合并操作。

3) Master Thread 每秒和每 10 秒的合并操作。

2. 二次写(double write)

Doublewrite 缓存是位于系统表空间的存储区域, 用来缓存 InnoDB 的数据页从 innodb buffer pool 中 flush 之后并写入到数据文件之前, 所以当操作系统或者数据库进程在数据页写磁盘的过程中崩溃, InnoDB 可以在 doublewrite 缓存中找到数据页的备份而用来执行 crash 恢复。数据页写入到 doublewrite 缓存的动作所需要的 I/O 消耗要小于写入到数据文件的消耗, 因此写入操作会以一次大的连续块的方式写入

在应用 (apply) 重做日志前, 用户需要一个页的副本, 当写入失效发生时, 先通过页的副本来还原该页, 再进行重做, 这就是 double write

doublewrite 组成:

内存中的 doublewrite buffer, 大小 2M。

物理磁盘上共享表空间中连续的 128 个页, 即 2 个区 (extend), 大小同样为 2M。

对缓冲池的脏页进行刷新时, 不是直接写磁盘, 而是会通过 memcpy() 函数将脏页先复制到内存中的 doublewrite buffer, 之后通过 doublewrite 再分两次, 每次 1M 顺序地写入共享表空间的物理磁盘上, 在这个过程中, 因为 doublewrite 页是连续的, 因此这个过程是顺序写的, 开销并不是很大。在完成 doublewrite 页的写入后, 再将 doublewrite buffer 中的页写入各个表空间文件中, 此时的写入则是离散的。如果操作系统在将页写入磁盘的过程中发生了崩溃, 在恢复过程中, innodb 可以从共享表空间中的 doublewrite 中找到该页的一个副本, 将其复制到表空间文件, 再应用重做日志。

3. 自适应哈希索引(ahi)

Adaptive Hash index 属性使得 InnoDB 更像是内存数据库。该属性通过 `innodb_adaptive_hash_index` 开启, 也可以通过 `skip-innodb_adaptive_hash_index` 参数关闭

InnoDB 存储引擎会监控对表上二级索引的查找, 如果发现某二级索引被频繁访问, 二级索引成为热数据, 建立哈希索引可以带来速度的提升
经常访问的二级索引数据会自动被生成到 hash 索引里面去(最近连续被访问三次的的数据), 自适应哈希索引通过缓冲池的 B+树构造而来, 因此建立的速度很快。

哈希(hash)是一种非常快的等值查找方法, 在一般情况下这种查找的时间复杂度为 $O(1)$, 即一般仅需要一次查找就能定位数据。而 B+树的查找次数, 取决于 B+树的高度, 在生产环境中, B+树的高度一般 3-4 层, 故需要 3-4 次的查询。

innodb 会监控对表上个索引页的查询。如果观察到建立哈希索引可以带来速度提升, 则自动建立哈希索引, 称之为自适应哈希索引 (Adaptive Hash Index, AHI)。

AHI 有一个要求, 就是对这个页的连续访问模式必须是一样的。

例如对于 (a, b) 访问模式情况:

where a = xxx

where a = xxx and b = xxx

特点:

- 1、无序, 没有树高
- 2、降低对二级索引树的频繁访问资源, 索引树高 ≤ 4 , 访问索引: 访问树、根节点、叶子节点
- 3、自适应

缺陷:

- 1、hash 自适应索引会占用 innodb buffer pool;
- 2、自适应 hash 索引只适合搜索等值的查询, 如 `select * from table where index_col='xxx'`,
而对于其他查找类型, 如范围查找, 是不能使用的;
- 3、极端情况下, 自适应 hash 索引才有比较大的意义, 可以降低逻辑读。

4. 预读(read ahead)

InnoDB 使用两种预读算法来提高 I/O 性能: 线性预读 (linear read-ahead) 和随机预读 (random read-ahead)

为了区分这两种预读的方式, 我们可以把线性预读放到以 extent 为单位, 而随机预读放到以 extent 中的 page 为单位。线性预读着眼于将下一个 extent 提前读取到 buffer pool 中, 而随机预读着眼于将当前 extent 中的剩余的 page 提前读取到 buffer pool 中。

线性预读 (linear read-ahead)

方式有一个很重要的变量控制是否将下一个 extent 预读到 buffer pool 中, 通过使用配置参数 `innodb_read_ahead_threshold`, 可以控制 InnoDB 执行预读操作的时间。如果一个 extent 中的被顺序读取的 page 超过或者等于该参数变量时, InnoDB 将会异步的将下一个 extent 读取到 buffer pool 中, `innodb_read_ahead_threshold` 可以设置为 0-64 的任何值, 默认值为 56, 值越高, 访问模式检查越严格

例如, 如果将值设置为 48, 则 InnoDB 只有在顺序访问当前 extent 中的 48 个 pages 时才触发线性预读请求, 将下一个 extent 读到内存中。如果值为 8, InnoDB 触发异步预读, 即使程序段中只有 8 页被顺序访问。你可以在 MySQL 配置文件中设置此参数的值, 或者使用 SET GLOBAL 需要该 SUPER 权限的命令动态更改该参数。

在没有该变量之前, 当访问到 extent 的最后一个 page 的时候, InnoDB 会决定是否将下一个 extent 放入到 buffer pool 中。

随机预读 (randomread-ahead)

随机预读方式则是表示当同一个 extent 中的一些 page 在 buffer pool 中发现时, InnoDB 会将该 extent 中的剩余 page 一并读到 buffer pool 中, 由于随机预读方式给 InnoDB code 带来了一些不必要的复杂性, 同时在性能也存在不稳定性, 在 5.5 中已经将这种预读方式废弃。要启用此功能, 请将配置变量设置 `innodb_random_read_ahead` 为 ON。

40. MyISAM 和 InnoDB select count(*)哪个更快, 为什么?

MyISAM 快, 因为 MyISAM 本身就记录了数量, 而 InnoDB 要扫描数据。

41. MySQL 中 myisam 与 innodb 的区别。

1、存储结构

每个 MyISAM 在磁盘上存储成三个文件。第一个文件的名称以表的名称开始, 扩展名指出文件类型。

.frm 文件存储表定义。

数据文件的扩展名为 .MYD (MYData)。

索引文件的扩展名是 .MYI (MYIndex)。

2、存储空间

MyISAM: 可被压缩, 存储空间较小。

InnoDB: 需要更多的内存和存储, 它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

MyISAM 的索引和数据是分开的, 并且索引是有压缩的, 内存使用率就对应提高了不少。能加载更多索引, 而 InnoDB 是索引和数据是紧密捆绑的, 没有使用压缩从而会造成 InnoDB 比 MyISAM 体积庞大不小

3、事务处理

MyISAM 类型的表强调的是性能，其执行速度比 InnoDB 类型更快，但是不支持外键、不提供事务支持。

InnoDB 提供事务支持事务，外部键（foreign key）等高级数据库功能。

SELECT、UPDATE、INSERT、Delete 操作

如果执行大量的 SELECT，MyISAM 是更好的选择。

如果你的数据执行大量的 INSERT 或 UPDATE，出于性能方面的考虑，应该使用 InnoDB 表。

DELETE FROM table 时，InnoDB 不会重新建立表，而是一行一行的删除。而 MyISAM 则是重新建立表。在 innodb 上如果要清空保存有大量数据的表，最好使用 truncate table 这个命令。

AUTO_INCREMENT

MyISAM: 可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。

InnoDB: InnoDB 中必须包含只有该字段的索引。引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

4、表的具体行数

MyISAM: 保存有表的总行数，如果 select count(*) from table; 会直接取出该值。

InnoDB: 没有保存表的总行数，如果使用 select count(*) from table; 就会遍历整个表，消耗相当大，但是在加了 where 后，myisam 和 innodb 处理的方式都一样。

5、全文索引

MyISAM: 支持 FULLTEXT 类型的全文索引。不支持中文。

InnoDB: 不支持 FULLTEXT 类型的全文索引，但是 innodb 可以使用 sphinx 插件支持全文索引，并且效果更好。

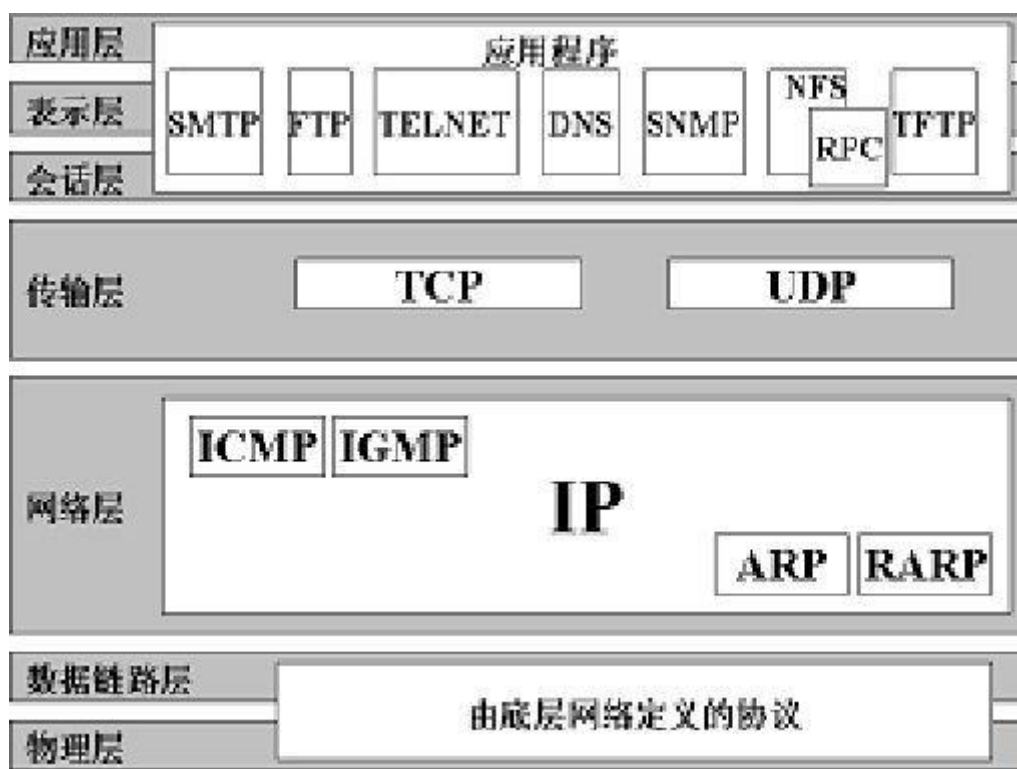
6、表锁差异

MyISAM: 只支持表级锁，只支持表级锁，用户在操作 myisam 表时，select, update, delete, insert 语句都会给表自动加锁。

InnoDB: 支持事务和行级锁，是 innodb 的最大特色。行锁大幅度提高了多用户并发操作的新能。但是 InnoDB 的行锁也不是绝对的，如果在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表，例如 update table set num=1 where name like “%aaa%”

42. 画出 OSI 和 TCP/IP 协议栈的对应关系.

OSI/RM 共分为七层，TCP/IP 分为四层。TCP/IP 中的网络接口层相当于 OSI 的物理层和数据链路层，TCP 的应用层相当于 OSI 的应用层、表示层和会话层。其余层次基本对应。见图，其中外围深颜色的是 OSI 层次，内部白颜色的是 TCP 层次。



43. 简述停止-等待协议（ARQ）的工作原理？

全双工通信的双方既是发送方也是接收方。为了讨论方便，仅考虑 A 发送数据而 B 接收数据并发送确认。A 叫发送方，B 叫接收方。“停止等待”就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组。

1、无差错情况

A 发送分组 M1，发送完后就暂停发送，等待 B 的确认。B 收到 M1 后就向 A 发送确认。A 在收到对 M1 的确认后，就继续发送下一个分组 M2。同样，在收到 B 对 M2 的确认后，再继续发送下一个分组。

2、出现差错

A 只要超过一段时间后仍没有收到确认，就认为刚发送的分组丢失，因而重传前面发送过的分组。实现这个功能应该保证：

一、A 在发送完一个分组后，必须暂时保留已发送的分组的副本。只有在收到相应的确认后才能清除暂时保留的分组的副本。

二、分组和确认分组都必须进行编号。

三、超时计时器设置的重传时间应当比数据在分组传 a 输的平均往返时间更长一些。

3、确认丢失和确认迟到

假设当 B 发送的对 M2 确认丢失后，A 在设定的超时重传时间内没有收到 M2 的确认，但并不知道是自己发送的分组出错、丢失，或者 B 发送的确认丢失。因此 A 在超时计时器到期后就要重传分组 M2。B 在收到 M2 后应采取的两个动作：

一、丢弃这个重复的分组 M2。

二、向 A 发送确认。

这种可靠传输协议称为自动重传请求 ARQ (Automatic Repeat reQuest), 可以在不可靠的传输网络上实现可靠的通信。

44. redis 的持久化有哪几种方式? 不同的持久化机制都有什么优缺点?

redis 持久化的两种方式

RDB: RDB 持久化机制, 是对 redis 中的数据执行周期性的持久化。

AOF: AOF 机制对每条写入命令作为日志, 以 append-only 的模式写入一个日志文件中, 在 redis 重启的时候, 可以通过回放 AOF 日志中的写入指令来重新构建整个数据集。

通过 RDB 或 AOF, 都可以将 redis 内存中的数据给持久化到磁盘上面来, 然后可以将这些数据备份到别的地方去, 比如说阿里云等云服务。

如果 redis 挂了, 服务器上的内存和磁盘上的数据都丢了, 可以从云服务上拷贝回来之前的数据, 放到指定的目录中, 然后重新启动 redis, redis 就会自动根据持久化数据文件中的数据, 去恢复内存中的数据, 继续对外提供服务。

如果同时使用 RDB 和 AOF 两种持久化机制, 那么在 redis 重启的时候, 会使用 AOF 来重新构建数据, 因为 AOF 中的数据更加完整。

RDB 优缺点

RDB 会生成多个数据文件, 每个数据文件都代表了某一个时刻中 redis 的数据, 这种多个数据文件的方式, 非常适合做冷备, 可以将这种完整的数据文件发送到一些远程的安全存储上去, 比如说 Amazon 的 S3 云服务上去, 在国内可以是阿里云的 ODPS 分布式存储上, 以预定好的备份策略来定期备份 redis 中的数据。

RDB 对 redis 对外提供的读写服务, 影响非常小, 可以让 redis 保持高性能, 因为 redis 主进程只需要 fork 一个子进程, 让子进程执行磁盘 IO 操作来进行 RDB 持久化即可。

相对于 AOF 持久化机制来说, 直接基于 RDB 数据文件来重启和恢复 redis 进程, 更加快速。

如果想要在 redis 故障时, 尽可能少的丢失数据, 那么 RDB 没有 AOF 好。一般来说, RDB 数据快照文件, 都是每隔 5 分钟, 或者更长时间生成一次, 这个时候就得接受一旦 redis 进程宕机, 那么会丢失最近 5 分钟的数据。RDB 每次在 fork 子进程来执行 RDB 快照数据文件生成的时候, 如果数据文件特别大, 可能会导致对客户端提供的服务暂停数毫秒, 或者甚至数秒。

AOF 优缺点

AOF 可以更好的保护数据不丢失, 一般 AOF 会每隔 1 秒, 通过一个后台线程执行一次 fsync 操作, 最多丢失 1 秒钟的数据。

AOF 日志文件以 append-only 模式写入, 所以没有任何磁盘寻址的开销, 写入性能非常高, 而且文件不容易破损, 即使文件尾部破损, 也很容易修复。

AOF 日志文件即使过大的时候, 出现后台重写操作, 也不会影响客户端的读写。因为在 rewrite log 的时候, 会对其中的指导进行压缩, 创建出一份需要恢复数据的最小日志出来。再创建新日志文件的时候, 老的日志文件还是照常写入。当新的 merge 后的日志文件 ready 的时候, 再交换新老日志文件即可。

AOF 日志文件的命令通过非常可读的方式进行记录, 这个特性非常适合做灾难性的误删除的紧急恢复。比如某人不小心用 flushall 命令清空了所有数据, 只要这个时候后台 rewrite 还没有发生, 那么就可以立即拷贝 AOF 文件, 将最后一条 flushall 命令给删了, 然后再将该 AOF 文件放回去, 就可以通过恢复机制, 自动恢复所有数据。

对于同一份数据来说, AOF 日志文件通常比 RDB 数据快照文件更大。

AOF 开启后, 支持的写 QPS 会比 RDB 支持的写 QPS 低, 因为 AOF 一般会配置成每秒 fsync 一次日志文件, 当然, 每秒一次 fsync, 性能也还是很高的。(如果实时写入, 那么 QPS 会大降, redis 性能会大大降低)

以前 AOF 发生过 bug, 就是通过 AOF 记录的日志, 进行数据恢复的时候, 没有恢复一模一样的数据出来。所以说, 类似 AOF 这种较为复杂的基于命令日志/merge/回放的方式, 比基于 RDB 每次持久化一份完整的数据快照文件的方式, 更加脆弱一些, 容易有 bug。不过 AOF 就是为了避免 rewrite 过程导致的 bug, 因此每次 rewrite 并不是基于旧的指令日志进行 merge 的, 而是基于当时内存中的数据进行指令的重新构建, 这样健壮性会好很多。

45. 如何理解 Paxos 算法?

Paxos 算法解决的是一个分布式系统如何就某个值 (决议) 达成一致。一个典型的场景是, 在一个分布式数据库系统中, 如果各个节点的初始状态一致, 每个节点执行相同的操作序列, 那么他们最后能够得到一个一致的状态。为了保证每个节点执行相同的命令序列, 需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。zookeeper 使用的 zab 算法是该算法的一个实现。在 Paxos 算法中, 有三种角色: Proposer (提议), Acceptor (接受者), Learners (记录员)

Proposer 提议者: 只要 Proposer 发的提案 Propose 被半数以上的 Acceptor 接受, Proposer 就认为该提案的 value 被选定了。

Acceptor 接受者: 只要 Acceptor 接受了某个提案, Acceptor 就认为该提案的 value 被选定了

Learner 记录员: Acceptor 告诉 Learner 哪个 value 就是提议者的提案被选定, Learner 就认为哪个 value 被选定。

Paxos 算法分为两个阶段, 具体如下:

阶段一 (准 leader 确定):

(a) Proposer 选择一个提案编号 N, 然后向半数以上的 Acceptor 发送编号为 N 的 Prepare 请求。

(b) 如果一个 Acceptor 收到一个编号为 N 的 Prepare 请求, 且 N 大于该 Acceptor 已经响应过的所有 Prepare 请求的编号, 那么它就会将它已

经接受过的编号最大的提案（如果有的话）作为响应反馈给 Proposer，同时该 Acceptor 承诺不再接受任何编号小于 N 的提案。

阶段二（leader 确认）：

(a) 如果 Proposer 收到半数以上 Acceptor 对其发出的编号为 N 的 Prepare 请求的响应，那么它就会发送一个针对 $[N, V]$ 提案的 Accept 请求给半数以上的 Acceptor。注意： V 就是收到的响应中编号最大的提案的 value，如果响应中不包含任何提案，那么 V 就由 Proposer 自己决定。

(b) 如果 Acceptor 收到一个针对编号为 N 的提案的 Accept 请求，只要该 Acceptor 没有对编号大于 N 的 Prepare 请求做出过响应，它就接受该提案。

46. 如何理解 raft 算法？

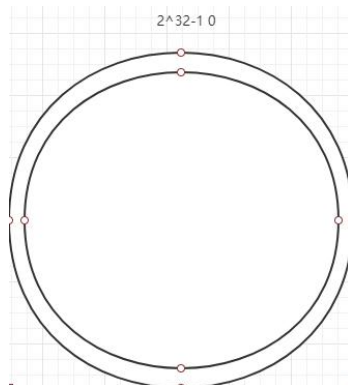
Raft 算法包括三种角色：Leader（领导者）、Candidate（候选领导者）和 Follower（跟随者），决策前通过选举一个全局的 leader 来简化后续的决策过程。Leader 角色十分关键，决定日志(log)的提交。日志只能由 Leader 向 Follower 单向复制。

典型的过程包括以下两个主要阶段：

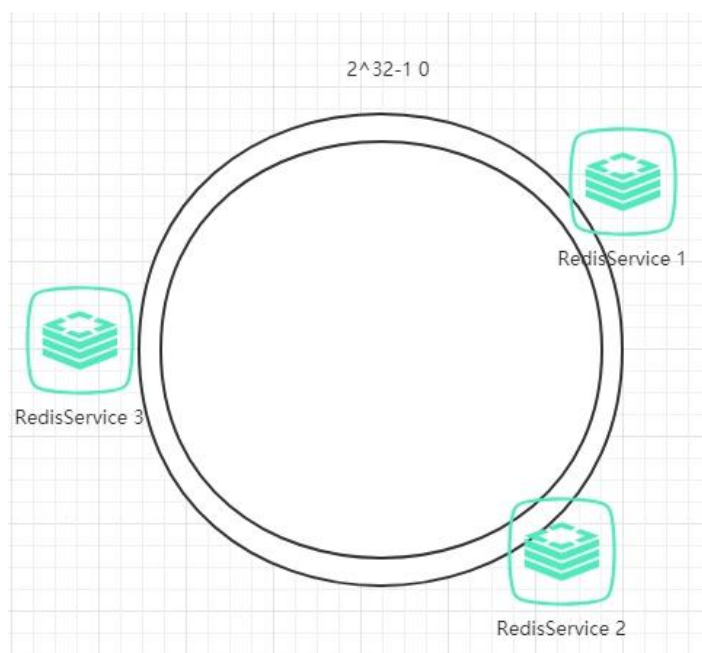
Leader 选举：开始所有节点都是 Follower，在随机超时发生后未收到来自 Leader 或 Candidate 消息，则转变角色为 Candidate，提出选举请求。最近选举阶段(Term)中 得票超过一半者被选为 Leader；如果未选出，随机超时后进入新的阶段重试。Leader 负责从客户端接收 log，并分发到其他节点；同步日志：Leader 会找到系统中日志最新的记录，并强制所有的 Follower 来刷新到这个记录，数据的同步是单向的。

47. 如何理解一致性哈希算法？

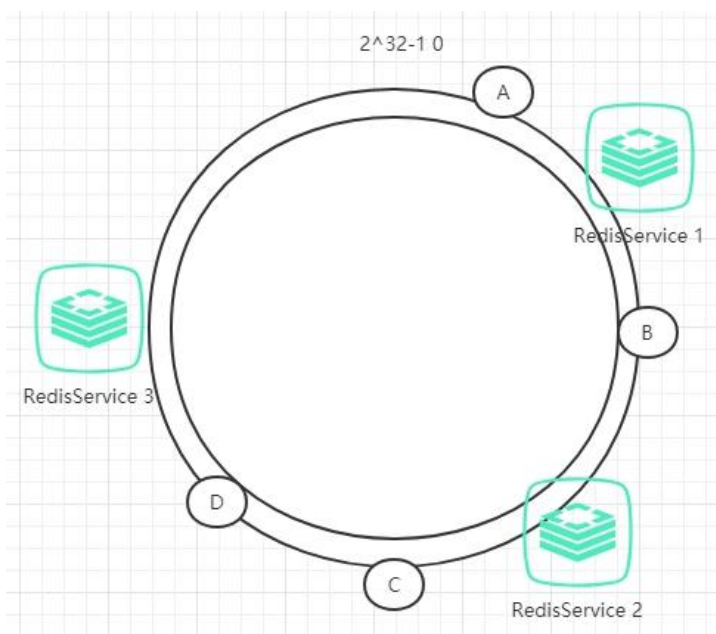
简单来说，一致性哈希是将整个哈希值空间组织成一个虚拟的圆环，如假设哈希函数 H 的值空间为 $0-2^{32}-1$ （哈希值是 32 位无符号整形），整个哈希空间环如下：



整个空间按顺时针方向组织, 0 和 $2^{32}-1$ 在零点中方向重合。
接下来, 把服务器按照 IP 或主机名作为关键字进行哈希, 这样就能确定其在哈希环的位置。



然后, 我们就可以使用哈希函数 H 计算值为 key 的数据在哈希环的具体位置 h, 根据 h 确定在环中的具体位置, 从此位置沿顺时针滚动, 遇到的第一台服务器就是其应该定位到的服务器。
例如我们有 A、B、C、D 四个数据对象, 经过哈希计算后, 在环空间上的位置如下:



根据一致性哈希算法, 数据 A 会被定为到 Server 1 上, 数据 B 被定为到 Server 2 上, 而 C、D 被定为到 Server 3 上。

48. voip 都用了那些协议?

H. 323 协议簇、SIP 协议、Skype 协议、H. 248 和 MGCP 协议

49. 阅读下面代码,回答问题?

```
unsigned char *p1;  
unsigned long *p2;  
p1=(unsigned char *)0x801000;  
p2=(unsigned long *)0x810000;  
请问:p1+5=?      p2+5=?
```

0x801005(相当于加上 5 位) ; 0x810014(相当于加上 20 位);

50. IP Phone 的原理是什么?

IP 电话（又称 IP PHONE 或 VoIP）是建立在 IP 技术上的分组化、数字化传输技术,其基本原理是：通过语音压缩算法对语音数据进行压缩编码处理,然后把这些语音数据按 IP 等相关协议进行打包,经过 IP 网络把数据包传输到接收地,再把这些语音数据包串起来,经过解码解压处理后,恢复成原来的语音信号,从而达到由 IP 网络传送语音的目的。

51. 1 号信令和 7 号信令有什么区别,我国某前广泛使用的是那一种?

1 号信令接续慢,但是稳定,可靠。

7 号信令的特点是：信令速度快,具有提供大量信令的潜力,具有改变和增加信令的灵活性,便于开放新业务,在通话时可以随意处理信令,成本低。目前得到广泛应用。

52. 请问交换机和路由器分别的实现原理是什么？分别在哪个层次上面实现的？

接下来，把服务器按照 IP 或主机名作为关键字进行哈希，这样就能确定其在哈希环的位置。

一般意义上说交换机是工作在数据链路层。但随着科技的发展，现在有了三层交换机，三层交换机已经扩展到了网络层。也就是说：它等于“数据链路层 + 部分网络层”。交换机中传的是帧。通过存储转发来实现的。路由器是工作在网络层。路由器中传的是 IP 数据报。主要是选址和路由。

53. 请问 C++ 的类和 C 里面的 struct 有什么区别？

整个空间按顺时针方向组织，0 和 $2^{32}-1$ 在零点中方向重合。

结构是一种将数据集合成组的方法，类是一种同时将函数和数据都集合成组的方法。结构和类在表面上的唯一区别是：类中的成员在默认情况下是私有的，而结构中的成员在默认情况下是公用的。

```
class    foo
{
private:
    int    data1;
public:
    void    func();
};
```

可以写成：

```
class    foo
{
    int    data1;
public:
    void    func();
};
```

因为在类中默认的是私有的，所以关键字 private 就可以不写了。

如果想用结构完成这个类所作的相同的事，就可以免去关键字 public，并将公有成员放置在私有成员之前：

```
struct    foo
{
    void    func();
private:
    int    data1;
};
```

54. 8086 是多少位的系统？在数据总线上是怎么实现的？

8086 微处理器初次发布时，这块 16 位芯片仅包含 29000 个晶体管，运行速度为 5MHz。而当今基于 x86 架构的奔腾 4 处理器，已经包含 5500 万个晶体管，运行速度提高了 600 倍以上，高达 3.06GHz。

8086 是高性能的第三代微处理器，是 Intel 系列的 16 位微处理器，它是采用 HMOS 工艺制造的，内部包含约 29,000 个晶体管。

8086 有 16 根数据线和 20 根地址线，因为可用 20 位地址，所以可寻址的地址空间达 220 即 1M 字节。8086 工作时，只要一个 5V 电源和一相时钟，时钟频率为 5MHz。后来，Intel 公司推出的 8086-1 型微处理器时钟频率高达 10MHz，8086-2 型微处理器时钟频率达 8MHz。

55. 请你详细的解释一下 IP 协议的定义，在哪个层上面，主要有什么作用？ TCP 与 UDP 呢？

UDP, TCP 在传输层，IP 在网络层，TCP/IP 是英文 Transmission Control Protocol/Internet Protocol 的缩写，意思是“传输控制协议/网际协议”。

TCP/IP 协议组之所以流行，部分原因是因为它可以用在各种各样的信道和底层协议（例如 T1 和 X.25、以太网以及 RS-232 串行接口）之上。

确切地说，TCP/IP 协议是一组包括 TCP 协议和 IP 协议，UDP (User Datagram Protocol) 协议、ICMP (Internet Control Message Protocol) 协议和其他一些协议的协议组。

TCP/IP 协议并不完全符合 OSI 的七层参考模型。传统的开放式系统互连参考模型，是一种通信协议的 7 层抽象的参考模型，其中每一层执行某一特定任务。

该模型的目的是使各种硬件在相同的层次上相互通信。

这 7 层是：物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。

而 TCP/IP 通讯协议采用了 4 层的层级结构，每一层都呼叫它的下一层所提供的网络来完成自己的需求。这 4 层分别为：

应用层：应用程序间沟通的层，如简单电子邮件传输 (SMTP)、文件传输协议 (FTP)、网络远程访问协议 (Telnet) 等。

传输层：在此层中，它提供了节点间的数据传送服务，如传输控制协议 (TCP)、用户数据报协议 (UDP) 等，TCP 和 UDP 给数据包加入传输数据并把它传输到下一层中，这一层负责传送数据，并且确定数据已被送达并接收。

互连网络层：负责提供基本的数据封包传送功能，让每一块数据包都能够到达目的主机（但不检查是否被正确接收），如网际协议 (IP)。

网络接口层: 对实际的网络媒体的管理, 定义如何使用实际网络 (如 Ethernet、Serial Line 等) 来传送数据。

56. 全局变量和局部变量有什么区别?是怎么实现的?操作系统和编译器是怎么知道的?

从作用域看:

全局变量具有全局作用域。全局变量只需在一个源文件中定义, 就可以作用于所有的源文件。当然, 其他不包含全局变量定义的源文件需要用 `extern` 关键字再次声明这个全局变量。

局部变量也只有局部作用域, 它是自动对象 (auto), 它在程序运行期间不是一直存在, 而是只在函数执行期间存在, 函数的一次调用执行结束后, 变量被撤销, 其所占用的内存也被收回。

从分配内存空间看:

全局变量, 静态局部变量, 静态全局变量都在静态存储区分配空间, 而局部变量在栈里分配空间

全局变量和局部变量的区别是作用域不同, 全局变量从定义位置开始到程序结束, 而局部变量只限定义的函数内可使用, 全局变量在数据段, 而局部变量在栈, 局部变量在函数结束时内存空间就被系统收回, 所以要返回的数组或字符串不要用局部变量定义

全局都放在静态存储区, 局部一般临时分配在栈里, 生命周期到, 自动释放内存!

57. 操作系统中的进程调度策略有哪几种?

1. 先来先服务调度算法:

先来先服务 (FCFS) 调度算法是一种最简单的调度算法, 该算法既可用于作业调度, 也可用于进程调度。当在作业调度中采用该算法时, 每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业, 将它们调入内存, 为它们分配资源、创建进程, 然后放入就绪队列。在进程调度中采用 FCFS 算法时, 则每次调度是从就绪队列中选择一个最先进入该队列的进程, 为之分配处理机, 使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

2. 短作业 (进程) 优先调度算法:

短作业 (进程) 优先调度算法 SJ(P)F, 是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先 (SJF) 的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业, 将它们调入内存运行。而短进程优先 (SPF) 调度算法则是从就绪队列中选出一个估计运行时间最短的进程, 将处理机分配给它, 使它立即执行并一直执行到完成, 或发生某事件而被阻塞放弃处理机时再重新调度。

3. 高优先权优先调度算法:

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先 (FPP) 调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

(3.1) 非抢占式优先权算法：在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

(3.2) 抢占式优先权调度算法：在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程 (原优先权最高的进程) 的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程 i 时，就将其优先权 P_i 与正在执行的进程 j 的优先权 P_j 进行比较。如果 $P_i \leq P_j$ ，原进程 P_j 便继续执行；但如果是 $P_i > P_j$ ，则立即停止 P_j 的执行，做进程切换，使 i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

(3.3) 容易出现优先级倒置现象：优先级反转是指一个低优先级的任务持有一个被高优先级任务所需要的共享资源。高优先任务由于因资源缺乏而处于受阻状态，一直等到低优先级任务释放资源为止。而低优先级获得的 CPU 时间少，如果此时有优先级处于两者之间的任务，并且不需要那个共享资源，则该中优先级的任务反而超过这两个任务而获得 CPU 时间。如果高优先级等待资源时不是阻塞等待，而是忙循环，则可能永远无法获得资源，因为此时低优先级进程无法与高优先级进程争夺 CPU 时间，从而无法执行，进而无法释放资源，造成的后果就是高优先级任务无法获得资源而继续推进。

(3.4) 优先级反转案例解释：不同优先级线程对共享资源的访问的同步机制。优先级为高和低的线程 t_{all} 和线程 low 需要访问共享资源，优先级为中等的线程 mid 不访问该共享资源。当 low 正在访问共享资源时， t_{all} 等待该共享资源的互斥锁，但是此时 low 被 mid 抢先了，导致 mid 运行 t_{all} 阻塞。即优先级低的线程 mid 运行，优先级高的 t_{all} 被阻塞。

(3.5) 优先级倒置解决方案：

(3.5.1) 设置优先级上限，给临界区一个高优先级，进入临界区的进程都将获得这个高优先级，如果其他试图进入临界区的进程的优先级都低于这个高优先级，那么优先级反转就不会发生。

(3.5.2) 优先级继承，当一个高优先级进程等待一个低优先级进程持有的资源时，低优先级进程将暂时获得高优先级进程的优先级别，在释放共享资源后，低优先级进程回到原来的优先级别。嵌入式系统 VxWorks 就是采用这种策略。

(3.5.3) 第三种方法就是临界区禁止中断，通过禁止中断来保护临界区，采用此种策略的系统只有两种优先级：可抢占优先级和中断禁止优先级。前

者为一般进程运行时的优先级，后者为运行于临界区的优先级。火星探路者正是由于在临界区中运行的气象任务被中断发生的通信任务所抢占才导致故障，如果有临界区的禁止中断保护，此一问题也不会发生。

4、高响应比优先调度算法：

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率 a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

5、时间片轮转法：

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把 CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几 ms 到几百 ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

6、多级反馈队列调度算法：

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。

(6.1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第 $i+1$ 个队列的时间片要比第 i 个队列的时间片长一倍。

(6.2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按 FCFS 原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按 FCFS 原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第 n 队列后，在第 n 队列便采取按时间片轮转的方式运行。

(6.3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第 $1 \sim (i-1)$ 队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时，又有新进程进入优先权较高的队列(第 $1 \sim (i-1)$ 中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调

度程序把正在运行的进程放回到第 i 队列的末尾，把处理机分配给新到的高优先权进程。

58. 将二叉树的两个孩子换位置，即左变右，右变左。（不能用递归）

```
//二叉树的两个孩子换位置
//在二叉链表下的交换操作
void exchange(BiTree BT)
{
    BiTree    r,p;
    BiTree    stack[100];
    int    tp=0;
    stack[0]=BT;
    while(tp >= 0)
    {
        p=stack[tp];
        tp=tp-1;
        if(p!=NULL)
        {
            r=p-> lchild;
            p-> lchild=p-> rchild;
            BT-> rchild=r;
            stack[tp+1]=p-> lchild;
            tp=tp+1;
            stack[tp+1]=p-> rchild;
            tp=tp+1;
        }
    }
}
```

59. 网球中心共有 100 个网球场，每个单位可以来申请 1 到 100 的场地，申请的场地编号必须是连续的，如果场地已经被其他单位占用，就不能再次使用，而且单位在使用完场地后必须归还。请设计一个完整的系统(c 语言）。（限时 5 分钟）

```
Tennis.h
struct TennisGround
{
```



```
    int num;
    char *agentName;
};
typedef struct TennisGround TG;
void mallocTG(TG *total);
void freeTG(TG *total);
Tennis.c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include "Tennis.h"
void mallocTG(TG *total)
{
    int size, start, count = 0;
    char *agentName = (char*)malloc(sizeof(char)*10);
    printf("Please input your agentName:");
    scanf("%s", agentName);
    printf("Please input the size of the TennisGround:");
    scanf("%d", &size);
    printf("Please input the TennisGround number you want to start:");
    scanf("%d", &start);
    if((total+start)->agentName != " ")
    {
        printf("malloc failed!\n");
        exit(-1);
    }
    else
    {
        while(count < size)
        {
            (total+start+count)->agentName = agentName;
            count++;
        }
    }
}
void freeTG(TG* total)
{
    char *an = (char*)malloc(sizeof(char)*10);
    printf("please input agentName you want to free:");
    scanf("%s", an);
    int count = 0;
    while(count < 100)
    {
```

```
        if(strcmp((total+count)->agentName, an) == 0)
            (total+count)->agentName = " ";
        count++;
    }
}

int main()
{
    int i;
    int sw;
    TG *total = (TG*)malloc(sizeof(TG)*100);
    for(i=0; i<100; i++)
    {
        (total+i)->num = i;
        (total+i)->agentName = " ";
    }
    while(1)
    {
        printf("*****Tennis Ground
        Mallocation*****\n");
        for(i=0; i<100; i++)
        {
            printf("%d(%s) ", (total+i)->num, (total+i)->agentName);
            if(i%5 == 0)
                printf("\n");
        }
        printf("\n");
        printf("*****\n");
        printf("Please input your choosen:(1-malloc,2-free):");
        scanf("%d", &sw);
        if(sw == 1)
            mallocTG(total);
        else
            freeTG(total);
    }
    return 0;
}
```

60. 输入四个 IP 端,前两个为第一个 IP 段的起始和终止地址,后两个是第二个 IP 段的起始和终止地址,判断这两个 IP 段是否存在交集.

输入描述: 输入 4 个 IP

输出描述: 如果存在交集,输出 Overlap IP ; 如果不存在交集,输出 No

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>
#include<cassert>
using namespace std;

int *dec2bin(int decnum)
{
    int i, a, *b = { 0 };
    a = decnum;
    for (i = 7; i >= 0; i--)
    {
        b[i] = a % 2;
        a = a / 2;
    }
    return b;
}

int ipToInt(char *ipString)
{
    assert(ipString != NULL);
    int i = 0, j, n, count = 0, return_num = 0;
    char *tmp;
    int *tmp_num=NULL, *num=NULL, *d2b;
    char *s = ipString, *s_tmp=NULL;

    if (*s == '.')
        count++;
    count++;
    if (count != 4)
        return 0;

    while (*s != '\0')
    {
        if (*s != '.')
            count++;
```

```
{
    n = s - s_tmp;
    tmp = (char*)malloc(n*sizeof(char));
    memcpy(tmp, s, n);
    tmp_num[i] = atoi(tmp);
    d2b = dec2bin(tmp_num[i]);
    for (j = 0; j<8; j++)
        num[8 * i + j] = d2b[j];
    s++;
    i++;
    s_tmp = s;
}
s++;
}
if (*s == '\0')
{
    n = s - s_tmp;
    tmp = (char*)malloc(n*sizeof(char));
    memcpy(tmp, s, n);
    tmp_num[i] = atoi(tmp);
    d2b = dec2bin(tmp_num[i]);
    for (j = 0; j<8; j++)
        num[8 * i + j] = d2b[j];
}
for (j = 0; j<32; j++)
    return_num = return_num * 2 + num[j];

return return_num;
}


int main(void)
{
    char *s1, *s2, *s3, *s4;
    s1 = new char;
    s2 = new char;
    s3 = new char;
    s4 = new char;
    cin >> s1 >> s2 >> s3 >> s4;
    int n1, n2, n3, n4, i;
    n1 = ipToInt(s1);
    n2 = ipToInt(s2);
    n3 = ipToInt(s3);
    n4 = ipToInt(s4);
    if (n4<n1 || n3>n2)
```

```
        cout << "No Overlap IP" << endl;
    else
        cout << "Overlap IP" << endl;

    system("pause");
    return 0;
}
```

获取更多资料, 请联系【零声学院】Milo 老师 QQ:472251823





 面试分享.mp4


 TCPIP协议栈，一次课开启你的网络之门.mp4




 高性能服务器为什么需要内存池.mp4


 手把手写线程池.mp4


 reactor设计和线程池实现高并发服务.mp4


 nginx源码—线程池的实现.mp4

 MySQL的块数据操作.mp4


 高并发 tcpip 网络io.mp4


 去中心化，p2p，网络穿透一起搞定.mp4

 服务器性能优化 — 异步的效率.mp4

 区块链的底层，去中心化网络的设计.mp4

 深入浅出UDP传输原理及数据分片方法.mp4

 线程那些事.mp4

 后台服务进程挂了怎么办.mp4