# C/C++相关面试总结

## static

- 修饰全局变量和全局函数: 改变变量和函数的外链接属性， 被修饰的变量和函数仅在当前文件可见。

- 修饰局部变量: 改变变量的生命周期，变量存放在静态数据区，第一次执行的时候申请空间，进程结束之后释放空间。

- 修饰类成员属性和类成员函数: 被修饰的属性和函数不再属于对象，而是类本身。所有类对象都共享该属性。可通过类名或对象名进行访问，并且静态类成员函数仅可访问静态成员属性。

- static修饰头文件中的inline函数(inline函数只有定义在头文件中才可被内联): 用于解决多个.c文件引用同一个头文件中函数重复定义的问题。

## const

- const char \*p: const用于修饰\*p, 表明指向的内容不可被修改，但是指向可以被修改

- char const \*p: 与const char \*p相同

- char \*const p: const用于修饰p, 表明p的内容不可被修改，也就是指向不可以被修改

- const char \*const p: 指向和指向的内容均不可被修改

const的意义:

- 修饰传入参数: 告诉调用者，这是一个传入参数，该参数在函数内不会被修改。

- 修饰类成员函数: 表示当前方法不会修改任何成员属性（可修改mutable修饰的成员， 也可使用const\_cast强制转换为可修改类型）。

- 修饰类成员函数返回值：用于限定类成员函数返回的指针或引用不可被修改。

## C、C++其他关键字

- explicit: 避免c++对象构造函数只有一个参数时产生隐式类型转换。

- volatie: 直接从原始内存地址中存取数据，而非cpu cache中的缓存。保证变量对所有线程的可变性，禁止指令重排。 (通常用于一个线程修改，其他线程监听修改后的状态的场景，如典型的copy on rewrite， event loop中的stop标志等。 也被用在无锁队列中，避免指令乱序等问题)

- mutable:可变的,被修饰的变量允许在常成员函数中被修改.

- define: 只是简单的宏替换, 不占用内存，没有数据类型。

- \ 多条语句换行。 #将参数看做是一个字符串。 ##连接前后两个符号。

- define中的 do{}while(0)意义: 保证一个复杂的代码块语意不会错乱。

- const: 参与运行变量. const修饰的是变量, 占用内存空间，编译器可对其进行类型安全检查。

- inline: 由编译器完成函数展开，一般用于函数体小、频繁访问，并且不没有循环等复杂结构的函数 (如, 对accept、connect等底层api进行封装, 规范它们的返回值及errno，使用inline就很合适)。 在头文件中定义inline函数，需要使用static (为什么需要static，见上文)。

- extren: 显示说明了a的空间是在其他地方分配的。如果将全局变量定义在头文件中，可能会出现多个.c文件包含同一个全局变量而导致的重复定义问题。通常我们将全局变量定义在.c文件中。使用extren在头文件中申明。

- #ifndef \_\_cplusplus, extren “C”{, $ifdef \_\_cplusplus }意义。为了实现c++与c兼容, 告知编译器，使用c的方式使用这些函数申明等。 (c++为了实现函数重载，内部对函数名进行了重定义)

## 类型判0和类型转换

- 浮点型: a=0.0; if (a > -0.0001 && a < 0.0001)

- 布尔类型: b = false; if (!b)

- C语言中有隐式类型转换和强制类型转化。隐式类型转换发生在将一个小容器的变量赋值给一个大容器变量。而强制类型则可能将一个大容器的变量转换为小容器的变量，因此需要考虑转换过程中的精度问题。

- c++中的类型转换关键字

- static\_cast: 类似于c语言中的强制类型转换

- const\_cast: 去掉const变量的const属性

- dynamic\_cast: 用于安全的向下转型，如果转换失败则为NULL (强制转换可能出现非预期的数据操作)。

- reinterpret\_cast: 仅做重新解释，可以将一个指针转换为整数，可以将整数转换为指针

## 数组名、对数组名取地址等

- int a[10]; a, a[0], &a[0], &a分别有什么区别

- a, &a[0], &a它们的地址相同, 都表示数组的首地址。但a, &a[0]都代表数组中的第一个元素, 它的步长是sizeof(int)，而&a代表整个数组的首地址，它的步长是整个数组。

- 如何验证: printf(“%p %p, %p”, a+1, &a[0]+1, &a+1)

## 指针的意义

- 指针本质上是一个变量，这个变量存放的是一个地址，这个地址是对应内容的首地址。而指针类型代表的是这个内容的数据长度，根据首字节向后获取几个字节就是对应的内容。

- 相同类型的指针可以做差值，表示第一个首地址到第二个首地址之间的步长。但是不能相加或不同类型的指针运算，因为这是没有意义的。

- 指针除了保存一个空间的地址外，还具备寻址的能力，这就是多级指针的意义。而指针的层级决定了指针的寻址深度。

- 指针数组: 这个变量是一个数组, 内部的元素是指针。 char \*arr[] = {“aa”, “bb”}

- 数组指针: 这个变量是指针，指向一个数组。 int a[2][4]; int(\*pa)[4]=a; pa+1表示跳过一行。（区分清楚int \*p[]和 int(\*p)[]. 其中int \*p[]，\*修饰的是p[]）

- 函数指针: 指向函数的指针, int(\*p)(int, int). 这个函数指针变量为p, 返回值为int, 有两个int类型的行参。使用typedef可以简化函数指针的复杂度，如当返回值也是一个函数指针的时候。

- 指针函数: 返回值为指针的函数。

- 空指针、野指针、悬空指针。

## 构造、析构等相关内容

- 构造函数:

- 构造顺序: 首先调用基类构造函数、子对象构造函数、再调用派生类自身的构造函数。

- 虚基类: 虚基类由最最派生类进行构造，并且优先于非虚基类构造函数先执行。

- 拷贝构造

- A(const A &a)

- 赋值函数

- const A & operator=(const A &a){} 在赋值函数中需要判断传入参数与自身是否是统一个。

- 析构函数: virtual~A()

- 析构函数建议申明为虚函数，避免通过基类引用子类对象，调用基类的析构函数时，无法释放子类的相关成员。

- 派生类自身的析构函数、子对象的析构函数、基类析构函数。

## 面向对象三大特性

封装、继承、多态。

- 继承:

- 多继承: 利用成员名限定法消除二义性，在类中定义一个同名成员进行隐藏，访问时通过类名.成员名进行访问。

- 菱形继承: 使用虚基类解决二义性问题。虚基类由派生类最先构造(只有一份)。

- java中是单继承、多接口。

- 多态: 分编译期多态和运行时多态。

- 编译期多态: 模板编程、函数重载、运算符重载

- 运行时多态: 通过基类指针或引用调用不同派生类同名虚函数，表现出不同的行为。

- 纯虚函数与抽象类

## 指针和引用

- 引用时某个变量的”别名”，对引用的操作和对变量的操作完全相同。申明引用时一定要对其进行初始化。引用不是一种数据类型，因此不占用存储单元。

- 指针是一个变量，存放另一个变量的地址，是对另一个变量的间隔操作。

将引用或指针作为返回值需要注意如下问题:

- 切记不要返回局部变量的引用和地址。

- 不要返回函数内部new分配的内存的引用。

- 在返回一个对象的成员时，建议使用引用提高效率。

## struct和class的区别

在c++中的struct是对c种struct的扩充。 它与类没有本质的区别，主要的区别是默认访问权限。在struct默认是public，在class中默认是private。

## 虚函数实现原理

如何一个类中包含虚函数，那么它会有一个虚函数指针，指向一张虚函数表，在这张虚函数表中记录了对应类的虚函数地址。

一个基类对象引用了子类对象，数据空间本质上还是子类对象的。因此在访问存虚函数时，并不像访问本类型的普通函数一样，而是通过虚函数指针访问对应虚函数表中的函数记录。

## stl标准模板库

- vector、dequeue、list的原理及优缺点

- vector是一个动态数组, 单向增长，在尾部添加或删除快速，在中间或开始位置添加或删除费时。由于是一个数组，所以可以通过下标访问(随机访问)。

- dequeue是一个动态数组，双向增长，在头部和尾部添加和删除快速，中间位置添加或删除费事。由于是一个数组，所以可以通过下标访问(随机访问)。

- list: 双向链表，不能随机存取元素，在任何位置添加或删除都很快速。

- set、map

- 两种实现，红黑树和哈希表。

- 使用自定义类型做key时需要实现什么？ 需要实现比较运算符重载。

- stl中sort的实现。

- 在数据量大时，采用快排，分段递归。当分段后的数量小于每个门限值时，改用插入排序，避免快排深度递归带来过大额外负担。如果递归层次太深，还可改用堆排序。

- 迭代器:

- 迭代器实际上可以看做是容器的一个内部类，它指向容器的某个节点，通过重载++，->，=等运算符，对外提供简便的操作接口。在对迭代器进行++等操作时，由运算符重载等函数进行内部的指针等操作。

- 而容器.begin、end等操作，实际上是将对应容器的首节点和尾节点封装成一个迭代器类型返回。

- 仿函数和函数适配器

- 首先仿函数不是一个函数，而是一个对象。这意味着仿函数具有数据和操作数据的方法（区别于c语言中的回调指针，需要在同时传递回调指针和data数据）。

- 防函数重载了()运算符，使得可以像调用函数的方式进行调用。

- 函数适配器是能够将仿函数和另一个仿函数结合起来的仿函数。因仿函数本身是一个特定对象，因此我们可以在这个对象中添加一些数据，通过一定的规则，让适配器将按照绑定的数据完成底层数据接口和上层数据接口的转换。

- 迭代器失效的几种情况

- 建议使用while, 而不建议使用for (iterator++)

- 对序列式容器，删除当前的iterator会使后面所有的元素的iterator都失效，这是因为vecor和dequeue使用了连续分配内存，删除一个元素导致后面所有元素都会前移一个位置。使用erase，它会返回下一个有效的迭代器。

- 关联式容器: 只会导致被删除的数据无效。在遍历时需要先记录iterator的下个位置，然后删除当前节点。

- map中[]和find有什么区别，那个更快。

- map中find如果未找到时，将返回一个空迭代器。

- 使用map[]查找时，如果元素不存在时，它会插入一个默认的值，可能导致非预期的结果。

- 因此map.find要更快。

## 部分C++11特性

- nullptr (专门用来区分空指针和0)、auto(简化迭代器的使用)、decltype (为解决auto的缺陷而生，auto是自动类型推到，不再表述实际的类型)

- 范围for循环

- final和override

- 右值引用、std::move、完美转发

- 左值、右值: 相对的，左值表示赋值的左边部分， 右值表示赋值的右边部分 (在某些描述中，将能被赋值看做的表达式左值， 将不能被赋值的表达式看做是右值)。

- 左值引用: int &x = y; 表示x指向y, 其中=右边是一个可修改的左值。

- 右值引用: int &&x = 5; 5是一个右值，不可以被赋值。

- std::move: 移动语义，在不拷贝数据的的情况下，将另一个对象的数据移动到当前对象。

- std::move通常用来优化传参拷贝和返回值拷贝。 当然，将一个对象的数据转移到另一个对象时，也可以使用移动语义。 (c++中的临时对象, 传参、赋值、返回值)

- 完美转发: 在模板函数中可能将模板参数传递给另一个模板函数，此时有几种办法，分别是值传递、左值引用、指针传递。如果我们想传递右值，就需要使用std::forward了。

- 智能指针:

- unique\_ptr: 独占式拥有。保证同一时间内只有一个智能指针可以指向该对象。可以使用std::move转移对象。

- shared\_ptr: 使用引用计数管理对象。当引用计数为0时，释放对象。该模式有个问题，就是可能出现循环引用，导致无法释放。

- weak\_ptr: 不控制对象生命周期的智能指针。它只提供了一种访问手段。

## 编写一个String类

class String {

public:

String(char \*str) {

if (str == nullptr) {

data = nullptr;

return;

}

data = (char \*)malloc(strlen(str) + 1);

assert(data != nullptr);

strcpy(data, str);

}

String(const String &other) {

if (other.data == nullptr) {

data = nullptr;

return;

}

data = (char \*)malloc(strlen(other.data) + 1)

assert(data != nullptr)

strcpy(data, other.data);

}

String & operator=(const String &other) {

if (this == &other) { // 一定要判等

return;

}

if (other.data == nullptr) {

data = nullptr;

return;

}

data = (char \*)malloc(strlen(other.data) + 1);

assert(data != nullptr);

strcpy(data, other.data);

}

~String() {

if (data != nullptr) {

free(data);

}

}

private:

char \*data;

}

## 结构体字节对齐

遵循两个原则:

- 结构体大小是其内部最大的基本数据元素的整数倍 (最大元素大小为8)。

- 结构体中的每个元素距离结构体开始地址的大小是当前基本元素的整数倍, 如果不够则补0。

## sizeof和strlen

- char \*str = “aa”; sizeof(str) --- 指针大小（32位系统中为4个字节, 64位系统中未8个字节）

- char str1[] = “aa”; sizeof(str1) --- 3

- char a[10]; sizeof(a) -- 10, strlen(a)大小不确定(strlen只识别\0)。

## strcpy和memcopy

- strcpy没有缓冲区溢出检查，替代函数strncpy (strncpy没有对字符串使用\n结尾, 需手动添加)

- memcopy中dest和src重叠问题: 可以简单判断src和dest是否相等，决定是否拷贝。

- strcpy实现

char \*strcpy(char \*dest, const char \*src) // 考点1: const对传入参数进行接口约定

{

assert(dest != NULL && src != NULL); // 考点2: 对入口参数进行检测

char \*res = dest; // 考点3: 功能实现

while(\*src != '\0')

\*dest++ = \*src++;

\*dest = '\0'; // 考点4: 字符串结束符

return res; // 考点5: 为什么要返回dest的首地址? 答, 为了实现链式操作

}

- memcopy实现

void \*memcpoy(void \*dest, const void \*src, size\_t n)

{

assert(dest != NULL && dest != NULL);

char \*t\_dest = (char \*)dest;

while(n--)

\*t\_dest++ = \*t\_src++;

return dest;

}

# Java/Go相关面试补充

synchronized实现原理

ConcurrentHashMap实现原理

Golang数组和切片 (slice自动扩容导师制底层数组变化问题)

Golang协程、channel（有无缓冲区）

Golang协程同步(channel、mutex、rwmutex、atomic)

Golang select、context

Data race问题。

Golang垃圾回收

Golang内存逃逸

Golang Goroutine调度

<https://blog.csdn.net/weixin_34109408/article/details/91907482>

# 数据结构与算法

## 基础数据结构

- 红黑树

- 哈希表

- 跳表

- B+树

- 最小/大堆

## 排序算法

- 插入排序: 时间复杂度O(n2),空间复杂度O(1)

- 冒泡排序: 时间复杂度O(n2),空间复杂度O(1)

- 选择排序: 时间复杂度O(n2),空间复杂度O(1)

- 快速排序: 时间复杂度O(nlogn), 最坏情况下为O(n2)

- 堆排序: 时间复杂度O(nlogn)

<https://www.cnblogs.com/fnlingnzb-learner/p/9083552.html>

## 二分查找

- 匹配查找

- 查找第一个小于val的值

- 查找第一个大于val的值

- 二维数组中查找某个元素 (不同的排列方式).

## 链表相关

- 双向链表增删操作

- 判断链表是否有环，环大小，环节点位置

- 重排链表

- 在单链表中，在O(1)的时间复杂度内删除当前节点。 (将下个节点的内容拷贝到当前节点，删除下个节点)

- 删除链表中的倒数第n个节点

- 判断一个链表是否为回文结构

- 链表转置

## 二叉树相关

- 二叉树深度遍历和按层遍历

// 深度遍历使用递归非常简单, 如果使用迭代需要借助stack

void dfs(tree\_node \*root) {

    if (root == NULL) {

        return;

    }

    if (root->left != NULL) {

        dfs(root);

    }

    printf("%d", root->val);

    if (root->right != NULL) {

        dfs(root);

    }

}

// 按层遍历, 在使用迭代时需要使用queue

void bfs(tree\_node \*root) {

    if (root == NULL) {

        return;

    }

    queue<tree\_node \*> q;

    q.push(root);

    while(q.empty()) {

        tree\_node \*node = q.fornt()

        if (node->left != NULL) {

            q,push(node->left);

        }

        if (node->right != NULL) {

            q.push(node->right);

        }

        printf("%d", node->val);

    }

}

- 二叉树的镜像

// 二叉树的镜像, 深度遍历, 交换左右节点

void mirror(tree\_node \*root) {

    if (root == NULL) {

        return;

}

    // swap

    tree\_node \*tmp = root->left;

    root->left = root->right;

    root->right = tmp;

    mirror(root->left);

mirror(root->right);

}

- 求二叉树的核之和

// 求二叉树的核之和

// 深度遍历, 记录分支的权重

int total\_sum = 0;

void get\_sum(tree\_node \*root, int sum) {

    if (root == NULL) {

        return;

    }

    sum = sum \* 10 + root->val;

    if (root->left == NULL && root->right == NULL) {

        total\_sum += sum;

    }

    if (root->left != NULL) {

        get\_sum(root->left);

    }

    if (root->right != NULL) {

        get\_sum(root->right);

    }

}

void main() {

    get\_sum(root, 0);

    printf("%d\n", total\_sum);

}

- 求二叉树的最长路径和路径和为n的所有路径

// 求二叉树的最长路径

int maxsum = 0;

int get\_max\_high(btree\_node \*root) {

    if (root->left == NULL && root->right == NULL) {

        return 0;

    }

    int left = 0;

    int right = 0;

    if (root->left != NULL) {

        left = get\_max\_high(root->left) + 1;

    }

    if (root->right != NULL) {

        right = get\_max\_high(root->left) + 1;

    }

    // 每个节点的左右深度之和的最大值

    maxsum = max(maxsum, left +right);

    return max(left, right);

}

- 之字型打印二叉树和二叉树的左右视角

// 之字型打印二叉树,和二叉树的左右视角

// 按层遍历的基础上可以跟踪到每一层的开始和截止位置

void dump\_btree(tree\_node \*root) {

    if (root == NULL) {

        return;

    }

    queue<tree\_node \*> q;

    q.push(root);

    tree\_node \*tmp;

    int row = 0;

    while(q.empty()) {

        // 大循环嵌套小循环, 而小循环就是遍历当前层

        for(int i = 0; i < len; ++i) {

            tmp = q.fornt();

            // 左视角

            if (i == 0) {

                printf("%d\n", tmp->val);

            }

            if (tmp->left != NULL) {

                q.push(tmp->left);

            }

            if (tmp->right != NULL) {

                q.push(tmp->right);

            }

        }

    }

}

- 判断二叉树是否为平衡二叉树

- 判断两个二叉树是否相等

- 二叉树的镜像

## 穷举法

- 全排列

- n个筛子出现和为m的概率

- 跳台阶问题

- 整数数组和为n的非空子集

// 和为n的非空子集合

// 穷举所有可能的情况, 进行过滤输出.

void demo(vector<int> &vec, int pos, int val, stack<int> &s) {

    if (val == 0) {

        dump(s);

    }

    for (int i := pos; i < vec.lenght(); i++) {

        s.push(vec.at(i));

        demo(vec, val - vec.at(i))

        s.pop()

    }

}

void main() {

    vector<int> vec = {1, 3, 6, -2, -4, 10}

    stack<int> s;

    demo(vec, 0, 10, s);

}

- 给定一个数组和一个滑动窗口值，求滑动窗口里的最大值

- 和为n的连续正整数

// 和为n的最大连续子串. 滑动窗口的解题思路与此类似

// 首先是连续的, 我们在一个基准值上向后找m个元素, 如果和为n则输出.

// 找完之后, 在下一个基准值上向后累加

void demo(vector<int> &vec, int n) {

    for (int i = 0; i < vec.length(); i++) {

        int sum = 0;

        for(int j = i; j < vec.length(); j++) {

            sum += vec.at(j)

            if (sum == n) {

                dump\_vec(vec, i, j) // vec, start, end

            }

        }

    }

}

## 动态规划

- 求数组的最大和的连续子串

// 数组的最大和的连续子串

// [1, 10, -12, 2, 1]

// [1, 10, -9, 1, 2, 12]

// [1, 10, -12, 11, 2]

// 在遍历过程中, 最大值可能已经出现. 因此使用max\_sum记录最大值. 并且最大值每次更新, 都要修改last\_start和last\_end的位置

// 在遍历过程中, 如果左边的值 < 0, 则不可以累加(累加之后反而小了), 需要丢弃, 并修改本次start的位置.

void demo(vector<int> &vec) {

    int start = 0;

    int cur\_sum = 0;

    int max\_sum = 0;

    int last\_start = 0;

    int last\_end = 0;

    for (int i = 0; i < vec.length(); i++) {

        cur\_sum += vec.at(i);

        if (cur\_sum < 0) {

            start = i+1;

        }

        if max\_sum < cur\_sum {

            max\_sum = cur\_sum;

            last\_start = start;

            last\_end = i;

        }

    }

    // 都是负数

    if (end < start) {

        return

    }

    dump\_vector(vec, start, end);

}

- 分糖果

- 最大连续子串

- 买股票最佳时机

## TopK问题

- 大量的URL, 如何去重，如何找到重复出现最多次的n条记录。

- 寻找第k大。

## 栈问题

- 最大频率栈

// 最大频率栈

// push和pop, 每次pop时都pop出栈内出现频率最大的那个元素

// 如果有多个元素的频率相同, 则优先pop出最后push的那个元素

// 解题思路, 想要统计元素的频率, 需要借助使用map. 我们需要获取那个出现频率最大的元素, 需要使用max\_freq记录出现的次数.

// 再引入一个额外的map, 这个map中保存了每一层的入栈列表 (用于维护同一频率下的入栈顺序)

class fstack {

    int max\_freq;

    map<int, int> m; // val-->freq

    map<int, stack<int> \*> ms; // freq--->stack

    void push(int v) {

        int freq = m[v] + 1; // 如果不存在, 当做0对待. 不同的语言此处行为不一样.

        m[v] = freq;

        stack<int> \*s;

        // 有了新的频率, 压入新的栈

        if (freq > max\_freq) {

            s = new stack<int>()

            s->push(v)

            ms[freq] = s;

        } else {

            s = ms[freq];

            s->push(v)

        }

    }

    int pop() {

        // no elements

        if (max\_freq == 0) {

            return 0;

        }

        stack<int> \*s = ms[max\_freq]

        int v = s.pop()

        if (s.empty()) {

            delete s;

            ms[max\_freq] = NULL;

            max\_freq--;

        }

        return v;

    }

}

- 单调栈结构

- 设计一个栈，要求push、pop、min操作都是O(1)的时间复杂度

## 大数问题

+-\*/

## 其他问题

- lru

- atoi

- round\_robin、consistent hash

# 操作系统与网络编程

## 进程的内存布局

内核空间、栈空间、(mmap区, 位置无关)、堆空间、全局数据区(静态数据区、常量数据区)、代码段。

## 进程和线程

- 进程和线程的区别

- 进程是资源分配的最小单位，线程是任务调度的最小单位

- 进程至少包含一个线程，可能包含多个线程

- 进程的调度开销要高于线程

- 同一个进程的不同线程共享那些资源，独占那些资源

- 共享全局变量(静态变量)和堆空间等

- 每个线程有自己的栈空间和寄存器等

- 进程状态转换

- 就绪->运行->阻塞->结束

- 一个进程获得了除CPU以外的所有资源时，将进入就绪状态

- 当处于就绪状态的进程获得了CPU之后，将进入运行状态

- 处于运行态的进程在CPU时间片到期后将重新变成就绪状态

- 处于运行态的进程在某个条件未满足的情况下会陷入阻塞状态

- 处于阻塞状态的进程在条件满足之后会重新进入就绪状态

- 进程间通信方式

- 无名管道、有名管道: 是一种半双工管道，数据只能单向传输。

- 信号量: 是一个计数器，可以控制多个进程安全的访问共享资源，通常配合共享内存使用

- 共享内存: 多个进程共享同一块内存空间，无需复制、快捷。但是需要自行实现多线程安全策略 (如原子锁、借助信号量等)。

- 消息队列: 可以允许任意进程通过共享消息队列来进行数据传输，而且不需要考虑同步问题。但数据的传递和拷贝会带来大量的系统开销，不适合数据量大或操作频繁的场景。

- 信号: 信号是Linux上非常复杂的一种通信方式，它本身不能携带数据，并且在多线程程序中处理信号非常麻烦。

- 套接字: 优选套接字，如果对性能有更高的要求，建议在共享内存上进行封装。

- 线程间通信方式

- 使用全局变量进行通信，但是需要使用互斥锁等手段保证线程安全。

- 可以在全局变量的基础上封装一个消息队列。

- 线程调度为什么比进程调度开销更小

- 任务调度的主要开销:

- CPU执行任务调度开销

- 进程/线程的上下文

- 任务调度后，CPU Cache/TLB不命中，导致的却页中断开销

- 对于上述1，2点，进程和线程调度都是必须的。

- 对于第3点，新的进程不仅无法命中当前的CPU Cache, 而且还会将其他进程的CPU Cache给刷掉(淘汰)。

- 僵尸进程和孤儿进程

- 子进程退出后，父进程未执行join操作，对子进程进行回收。

- 父进程提前终止后，子进程将被Init进程领养。

## 用户态和内核态

- 用户态: 只能访问受限的内存，并且不允许直接操作外围设备，需要执行系统调用等操作后方可切换到内核态。

- 内核态： 可以访问内存的所有数据、包括外围设备，是权限最高的状态。

- 如何从用户态切换到内核态

- syscall

- 异常

- 外围设备的中断

## CPU任务调度

- 进程上下文切换

- 指执行该进程有关的各种寄存器: 通用寄存器、程序计数器、程序状态寄存器。

- 调度算法

- 先来先服务

- 短作业优先

- 时间片轮转

- 最终截止时间优先、多级反馈、最高优先级调度、最高响应比调度

## 内存管理

- 虚拟内存

- 作用: .为不同进程营造一个完整的地址空间，降低编程难度，对不同进程内存空间进行隔离，避免相互干扰。 (其他作用: 按需分配、页换入换出、写时复制)

- 段页式内存管理

- 分页: 用户程序的地址空间被划分为固定大小的块。而内存的分配是以页为单位的，也跟块的大小对应，这样可以保证进程的任何一个页都可存放在物理内存的任意一块中。

- 分段:分页方式的缺点是页常与程序的逻辑不相关，分段将用户程序地址空间划分为若干个不相等的段，每个段可以定义一组相对完整的逻辑信息。

- 寻址: 用户发起访问程序段的逻辑地址 <段号, 段内偏移量>，通过对这一落概念进行运算转换为访问页的虚拟地址 <页，页内偏移量>。再由MMU将其转换为内存的物理地址<页框号、页内偏移量>，通过这种方式，用户访问的虚拟空间经过两次地址隐射之后变成真实的物理地址。

- 内存分配原理

- 在申请内存时，分配的是虚拟内存，并没有分配物理内存。在第一次访问已分配的虚拟空间时，将触发缺页中断，进程将陷入内核态。操作系统将建立虚拟内存和物理内存之间的映射关系。

- 大页内存

- 物理内存是以页为单位进行分配，通常是4K。如果申请一个2M的内存空间，将触发512次中断，开销会非常高。可以使用mmap映射一块大内存到进程空间，以优化服务性能。

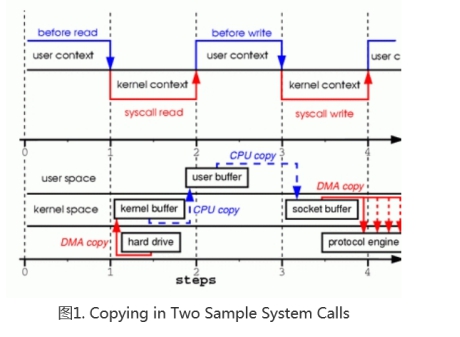
- 内存池

- 内存池的核心原理是对内存块的复用。

- 提高内存块的复用: 字节对齐

## IO读写流程

- read/write读写流程

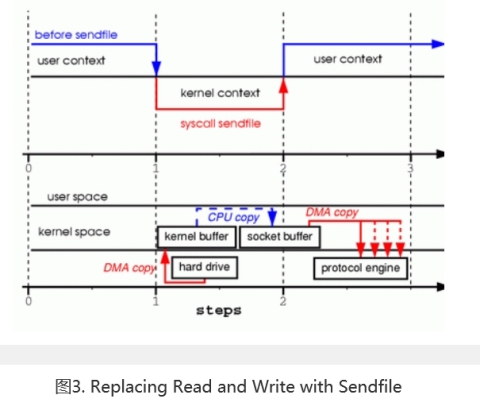


- 上下文切换: read/write操作，需要从用户态切入内核态，执行完毕后从内核态返回到用户态。

- 数据拷贝: 对于read操作(write类似)，DMA模块从磁盘或sockfd中读取数据并拷贝到内核缓冲区，这是第一次复制。数据从内核缓冲区拷贝到用户空间，这是第二层复制。

- 对于上述的read/write来说，从一个fd中读取数据并写入另一个fd，需要四次上下文切换和四次内存拷贝操作。

- sendfie原理



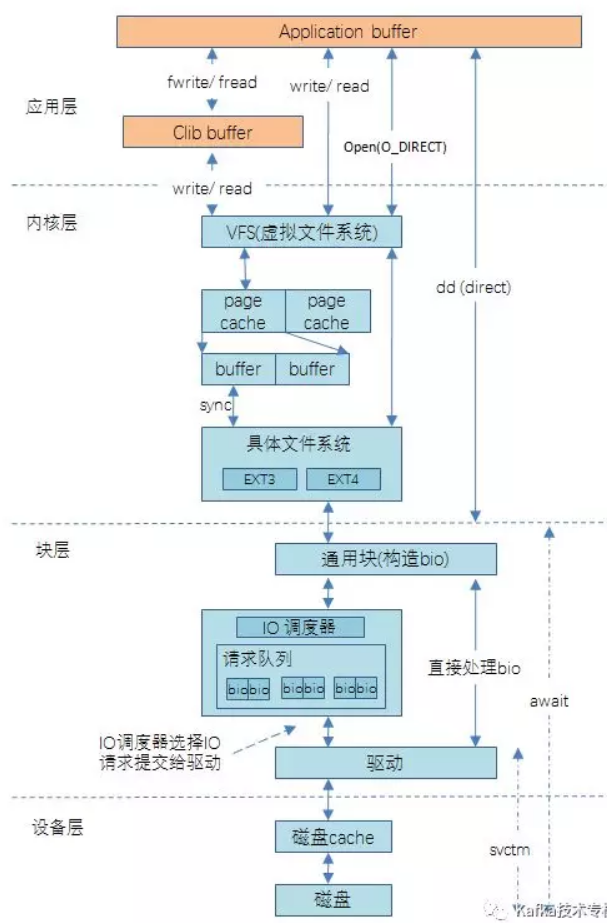
- 在sendfile中，不需要将数据拷贝到应用层，也不需要从应用层写入到内核，它的系统调用仅需要一次，上下文切换也只会产生两次。

- 当数据到达时，通过sendfile可以将数据拷贝到内核缓冲区，然后从内核缓冲区将数据拷贝到另一端。整个数据传输过程仅存在两次数据拷贝。

- 因内存中不存在数据的冗余副本，因此该技术也被称之为零拷贝技术。

## Linux磁盘IO模型

[https://mp.weixin.qq.com/s?src=11&timestamp=1605702718&ver=2714&signature=3FJzJzdJV9LvDIbVUE0VJy4JZuDWjmRru-n2rloYj9l2Slq9VuSUohjoFhI5Los0rGSZ85b2tBU6jJZk6jJyM6gYHz5hmy\*84Bh0xsn0phPS7yTpoh1UjyR2gMFyxcc2&new=1](https://mp.weixin.qq.com/s?src=11&timestamp=1605702718&ver=2714&signature=3FJzJzdJV9LvDIbVUE0VJy4JZuDWjmRru-n2rloYj9l2Slq9VuSUohjoFhI5Los0rGSZ85b2tBU6jJZk6jJyM6gYHz5hmy*84Bh0xsn0phPS7yTpoh1UjyR2gMFyxcc2&new=1)



通用块层: 通用块层根据IO请求构造一个或多个bio结构交给调度器层。

调度器: 调度器将bio结构进行排序和合并并组织成队列且确保读写操作尽可能理想（顺序读写要高于随机读写）。

常见的IO调度算法:

- Noop: 在FIFO的基础上对相邻IO做了合并，并不是严格按照FIFO的顺序进行处理。

- CFQ: 完全公平队列，它主要是确保磁盘IO带宽的公平分配。

- 最后期限算法、预测算法等(无研究)

## IP、TCP、UDP

- TCP/IP五层模型

- 物理层: 实现最终的电子信号传输

- 数据链路层: 实现最终的电子信号传输

- 网络层: 实现点到点通信，通过IP寻址来建立两个节点之间的联系和通信

- 传输层: 实现端到端通信， 网络层实现了两台计算机之间的寻址，传输层实现了两台计算机不同端口间的通信。

- 应用层

- 各层对应的协议

- 应用层: Ftp、Http、NFS等

- 传输层: TCP、UDP

- 网络层: IP、ICMP、ARP

- 各层对应设备

- 传输层: 四层交换机

- 网络层: 路由器和三层交换机

- 数据链路层: 网桥、以太网交换机

- 物理层: 中继器、集线器

- TCP分段和IP分片

- IP分片产生的原因是MTU。 TCP分段产生原因是MSS。

- IP分片由网络层完成，也在网络层进行重组。

- TCP分段是在传输层完成，并在传输层进行重组。

- 对于以太网，MSS为1460字节，而MTU往往会大于MSS。故，使用TCP协议传输数据时，是不会造成IP分片的。

- 对于UDP数据报来说，可能会照成IP分片，在对端重组，因为UDP是基于数据报的，数据单个UDP报文不可超过65535(send/recv)个字节 (过大则无法重组)。

- MTU: 最大传输单元，以太网MTU值为1500字节。

- MSS: 最大报文大小，一般不超过1460字节。

- 为什么TCP层要实现分段？ 如果分段工作全部交给IP层，在发生丢包的情况下, TCP层是无法感知到那一段丢失的，只能重传整个TCP报文。

- UDP和TCP的区别

- 无连接，有连接

- 数据报，数据流

- 不可靠，可靠

- UDP是尽最大努力交付

- TCP如何实现可靠通信

- 三次握手、四次挥手，连接的建立和释放

- 确认应答 和 超时重传 (丢包、乱序)

- 拥塞控制: 减少网络拥塞。 慢开始、拥塞避免、快重传、快恢复。

- 流量控制: 避免因对端接受能力不足导致丢包。 在每次发送数据时，都将自己的接受窗口发送给对方，对方根据接受窗口大小构造发送窗口，实现发送窗口和接收窗口同步。

- TCP三次握手及为什么需要三次握手

- 客户端发送的sync报文，可能因为某些原因，在网络上滞留。

- 服务端响应的ack可能丢失或延迟。

- 上述两种情况，对客户端来说都可能认为连接未建立成功。但是如果只有两次TCP握手，那么服务端可能认为连接已经建立了，但时这个连接客户端却并不会使用，导致服务端资源被白白浪费。

- TCP四次挥手,TIME\_WAIT和CLOSE\_WAIT

- TIME\_WAIT: 主动关闭的一方会进入这个状态。

- 最后一次ACK可能丢失, 如果对端未收到ACK可能重发FIN报文，通过维护TIME\_WAIT以实现可靠的TCP四次挥手。

- 另外这个重发的FIN报文可能在网络中滞留。

- 如果没有TIME\_WAIT的情况下，客户端可能在同一地址上建立新的链接。可能收到重发的FIN (或滞后的FIN)， 导致链接被关闭。

- nagle算法、延迟ack、bbr

- nagle: 为减少广域网中的小分组数目，从而减少网络拥塞。它会导致数据发送被延迟一段时间，竟可能的将小包聚合成一个大包，发送到网络中。它可能带来40ms的延迟，这在低延迟的网络场景中是无法接受的。

- 延迟ack: 为了避免网络中存在大量的ack报文，导致性能下降。

- 当nagle遇到delay ack: 两次写数据长度小于MSS，当第一次写数据到达对端后，对端延迟ack，不发送ack，而本端因为要发送的数据长度小于MSS，所以nagle算法起作用，数据并不会立即发送，而是等待对端发送的第一次数据确认ack；这样的情况下，需要等待对端超时发送ack，然后本段才能发送第二次写的数据，从而造成延迟。

- ack丢失会重传吗？

- ack丢失会重传请求报文， 当然也可能因为ack合并(收到一个更高序号的ack)无需重传。

## 其他问题

- 向一根已经关闭的管道中写入消息会出现什么情况，如何解决

- 会触发SIGPIPE信号，默认动作是终止程序，解决办法是忽略信号或捕捉信号。

- 互斥锁和条件锁

- 条件锁在陷入阻塞时会释放持有的互斥锁，在被唤醒之后会重新持有互斥锁。

- 大端字节序和小端字节序

- 死锁及避免/解除死锁

- 产生的必要条件: 互斥条件、请求保持、不可剥夺。循环等待

- 解决死锁的方式:

- 如果上锁失败,则先释放自己已经持有的资源,一段时间后重试(事件驱动模型)。

- 分布式锁场景，为防止进程崩溃导致锁无法释放问题，锁超过有效期后可以被抢占。

- 顺序持有，避免出现环形等待。

- 惊群现象

- 在内核2.6之前，一个新连接到来时会将所有的accept都唤醒，但是只有一个accept会接受连接并进行处理，这导致了大量的资源被浪费。

- 在使用条件锁的场景中，应该尽量使用pthread\_cond\_single来环形单个阻塞线程，而不是使用boardcast来换醒所有阻塞的线程。

- 还有一种惊群现象是, 多个线程的epoll同时监听同一个sockfd (accept)时，对应的socket可读时将唤醒所有的线程/进程。

# Linux系统编程

## bind、listen、accept、connect

- bind: 用于将一个socket绑定到特定的端口上。对于connect来说，linux内核会自动为socket绑定一个端口。

- listen: 为当前socket维护两个连接队列，用来实现TCP的三次握手，这两个队列分别为半连接队列和全连接队列。

- accept: accept其实是对全连接队列的一个出队操作。 如果sockfd是阻塞的，那么队列为空时accept将一直阻塞，直到已完成队列不为空时唤醒它。如果sockfd是非阻塞的，那么队列为空时，accept将立即返回EAGAIN。

- connect:阻塞场景下, connect等待TCP三次握手完成。非阻塞场景下，connect在发送sync之后就会返回。此时连接可能已经建立，也可能未建立，如果未建立的情况下降返回EINPROGRESS状态码，此时需要将sockfd加入到epoll中监听其可读可写事件，当读写事件触发时再检测sockfd相关标志位，判定是否建连成功。

## 同步、异步、阻塞、非阻塞

- 同步异步：同步是发出一个功能调用时，在没得到结果之前一直等待或继续处理。而异步是，调用者在没得到结果之前，就可以继续执行后续操作。

- 阻塞非阻塞： 阻塞是在条件不满足的情况下，一直阻塞等待条件满足。而非阻塞是在条件不满足的情况下立即返回。

## epoll和select

- select可监控的文件描述符个数受限(基于固定数组),而epoll可监控的文件描述符只受限于内存(基于链表)

- select是线性扫描文件描述符的,而epoll是通过为fd注册回调实现的,所以在文件描述符多的情况下,select的效率会呈线性下降. 而epoll在大量fd并且少量活跃的情况下具有明显的性能优势。

- select需要每次都需要将全部的sockfd拷贝到内核，而epoll只有在添加或移除fd时才会操作内核中的红黑树。

epoll et和lt

- LT (水平触发): 如何当前文件描述符一直处于可读状态或可写状态就一直返回。如果一个文件描述符上的数据没有读取完,那么它(select或epoll)会一直返回和通知.而对于写事件,一般来说只要写事件被监听,都会一直通知写事件(在一般情况下都是可写的,除非内核已满)

- ET模式(边沿触发):当文件描述符由不可读变为可读时,读事件被触发.当文件描述符由不可写变为可写时写事件被触发.所以当读事件被触发之后,即使缓冲区中还有未读取完的数据,它也不会再次产生回调(当然新的数据到来时还是会触发读事件，另外一个未监听的sockfd，重新监听时，如何内核中有数据，也会触发读事件).

对于写事件的应用:在非阻塞场景中,数据可能一次性发送不完,但是又不能进行while循环发送.所以需要将未发送完的数据先写到一个缓冲区,等待写事件回调时,将缓冲区的数据消费掉. 而写事件一般是在需要时监听写事件,在使用完成之后移除写事件. 读事件则不同,因为不确定在什么时候数据会到来,所以通常情况下读事件被一直监听,写事件在需要的时候注册,不需要的时候移除. (ET模式下也不一定会移除)

epoll ET在nginx中的使用:

nginx在一开始就监听了连接的读写事件，只是将它们的回调函数设置为傀儡函数(只打印日志)，在需要读写数据时，再切换对应的回调函数。在读写数据时，事件可能已经触发了，也可能没有。对于已经触发的情况，直接回调即可。如果事件未触发，需要等待事件系统异步回调。而nginx为每个事件都添加了一个ready标志，在事件触发时被设置为1, 在遇到EAGAIN时清0，以此来区分事件到底触发了没有。

## fd的exec和block属性

fd默认是在进程间传递的.有的时候我们不希望它在进程间传递,所以设置其FD\_CLOEXEC.

fd默认是阻塞的,如果在使用多路IO的场景,需要将其设置为非阻塞的.

设置方法:

通过fcntl先获取fd的属性,然后对该属性进行位运算,最后再调用fcntl将修改后的属性设置回去.

## IO读写及常见错误码

对一个非阻塞的文件描述符执行读操作时，如果ENO是EAGAIN，表示当前没有更多的数据了，需要下次继续读取。如果read 0或者ENO是ECONNREST的话，则表示对端关闭连接或关闭写操作了。

对一个非阻塞的文件描述符执行写操作，如果是EAGAIN, 则表示内核已满，等内核可写的时候继续写入。如果Write返回0、ECONNREST或EPIPE时，表示对方已经断开连接或者半关闭了读操作。

EPOLLRDHUP: EPOLLRDHUP是在内核中提前获取对端断开连接(不写)的手段,而不需要在应用层执行read 0才确定对端已经断开连接. 对方发送FIN、RST分节都会使内核同时触发EPOLLIN|EPOLLRDHUP事件。 (好处是少执行一次read操作, 比如有数据的情况下，第一次read返回数据, 第二次read返回0)。

## reactor模式的优缺点

## 什么是受控阻塞

## Linux原子操作、无锁队列

## C10K问题及解决之道

## 生产者消费者模型 (线程池)

## 内存池和连接池设计

## DPDK和EBPF

# ATS与Nginx

# 存储相关

计算机视野: CDN之我见、API网关之我见(apisix)、IM系统之我见、分布式存储之我见。

# mysql、redis、mq、leveldb

# 工作经验之谈

1.微服务架构

- 服务限流、限速、重试、熔断、降级

- 服务发现、弹性扩缩容

- 链路监控、zipkin和metrics等

- 安全认证、ACL管理

- 数据传输序列化和反序列化

2. DNS劫持、如何发现并解决

- 域名劫持是通过攻击域名解析服务器，后伪造域名解析服务器，将目标网站域名解析到错误的IP地址，从而导致用户无法访问或者访问伪造的服务的目的。

- 发现方式: 可以从服务端和客户端入手。如果域名被劫持，服务端相关的qps会大幅度下降。另外可以收集客户端访问的目标IP，与自己的IP库进行比对，如果不一致，可能发生了域名劫持、

- 应对方式，可以与APP端和运营商进行联动。一方面通知运营商进行修复，另一方面让APP从传统的DNS解析降级到HttpDNS。

3. 设计模式

（设计模式有很多种, 这里只介绍几种工作中常用的）

- 享元模式 (Flyweight)

可以看做是单利模式和pool的集合，pool中的每个对象都只有一份, 以实现共享。

- 单利模式 (Singleton)

懒汉式: 在需要时才创建

饥汉式: 在项目启动时就创建

- 工厂模式

-- 简单工厂

构造一个抽象类, 这个基类定义了子类的所有操作方法。

具体的子类继承这个抽象类, 也就向外暴露了对应的操作。

工厂类, 基于传入的类型生成对应的对象, 并向上转型, 返回基类对象。

优点是分工明确，模块清晰。而缺点是每次新增一个功能都得定义一个模型，并改动工厂的switch。

-- 工厂模式

与简单工厂模式相比，它将生成具体产品的任务分发给了产品工厂。

简单工厂模式是由工厂类提供的方法创建对象， 而工厂模式是由对应的子类创建工厂。

使用如下:

简单工厂: MyDB db = factor.CreateDB(“mysql”) / “sqlserver”

工厂模式: MyDB db = new MysqlDB()/SqlServer()

优缺点: 克服了简单工厂违背的开放-封闭原则。 缺点，改动基类不容易。

-- 抽象工厂

抽象工厂是围绕一个超级工厂去设计的，它是其他工厂的工厂。

抽象工厂提供一个创建子工厂的方法，创建出子工厂。

子工厂再使用简单工厂或者普通工厂模式的方式创建出对应的子类。

具备产品族的场景可以使用抽象工厂。 如QT中的绘图图、或者界面控件等。

- 装饰器模式

允许向一个现有的对象添加新的功能，同时又不改变其结构。 它可以看做现有类的一个包装。

比如一副画、画框、玻璃可以看做是对画的装饰。

优点: 装饰类和被装饰类可以独立发展，不会相互耦合，装饰模式是继承的一个替代模式。

其实就是一个类，包含了被修饰类的对象。 对当前类的操作可以同步作用的被修饰类上。如Java常见的字节流和字符流的转换，就是使用了装饰器模式。

4. java程序设计原则 （七种, 这里只描述个人熟悉的五种）

- 开闭原则: 对扩展开发、修改关闭。

- 单一原则: 一个类，只有一个引起它变化的原因，应该只有一个职责。 (跟KISS原理相同)

- 替换原则: 子类可以替换掉父类。 (老鼠的儿子会打洞)

- 依赖倒转: 高层模块不应该依赖底层模块。因此实现的细节应该围绕接口编程。（这也是目前Java程序设计的主要原则）

- 聚合原则: 尽量使用组合，而非继承。 因为父类的改变，必然会导致子类的改变。

1. 性能评测: atop、top、lsof、iostat等
2. 设计模式
3. 分布式锁 (redis、etcd)
4. lvs、负载均衡 、微服务架构的样子

<https://blog.csdn.net/weixin_34128839/article/details/94488565>