线程池

线程池有三个类：

1. 任务类
2. 任务队列
3. 线程池

主要组成部分是三个：

1. 任务队列，存储需要分配的任务，用队列实现
2. 工作线程。
   工作线程不停地读取任务队列，读取里面的任务并处理。
   如果队列为空，工作线程将会被阻塞，条件变量
   队列不为空，唤醒线程，工作
3. 管理线程。
   周期性的对任务队列中的任务数量以及处于忙碌状态的工作线程个数进行检测，当任务过多就适当创建一些新线程，当任务过少就销毁一些线程。

```cpp
#include<pthread.h>
#include<queue>
#include<iostream>
#include<string.h>
#include<exception>
#include<unistd.h>

const int NUMBER = 2;

using callback = void(*)(void* arg);


class Task{
private:
public:
    callback fun;
    void* arg;
public:
    Task(){
        fun = nullptr;
        arg = nullptr;
    }
    Task(callback f, void* arg){
        fun = f;
        this->arg = arg;
    }
};

class Task_queue
{
private:
    pthread_mutex_t m_mutex;
    std::queue<Task> m_queue;
public:
    Task_queue();
    ~Task_queue();

    //添加任务
    void add_task(Task& task);
```

```
    void add_task(callback f, void* arg);

    //取出任务
    Task& get_task();

    int task_size(){
        return m_queue.size();
    }
};

class Thread_pool{
private:
    pthread_mutex_t m_mutex_pool;
    pthread_cond_t m_not_empty;     //条件变量，用来唤醒等待的线程，假设队列是无限的
    pthread_t* thread_compose;
    pthread_t m_manger_thread;
    Task_queue* m_task_queue;
    int m_min_num;                  //最小线程数
    int m_max_num;                  //最大线程数
    int m_busy_num;                 //工作的线程数
    int m_live_num;                 //存活的线程数一
    int m_destory_num;              //待销毁的线程数
    bool shutdown = false;          //是否销毁

    //在使用pthread_create时候，第三个参数的函数必须是静态的
    //要在静态函数中使用类的成员(成员函数和变量)只能通过两种方式
    //1.通过类的静态对象，单例模式中使用类的全局唯一实例来访问类成员
    //2.类对象作为参数传递给该静态函数中
    //管理线程的任务函数，形参是线程池类型的参数
    static void* manager(void* arg);
    //工作线程的任务函数
    static void* worker(void* arg);
    //销毁线程池
    void destory_thread();
public:
    Thread_pool(int min, int max);
    ~Thread_pool();

    void add_task(Task task);
```

```
    int get_busy_num();
    int get_live_num();
};


Task_queue::Task_queue()
{
    pthread_mutex_init(&m_mutex, nullptr);
}


Task_queue::~Task_queue()
{
    pthread_mutex_destroy(&m_mutex);
}


void Task_queue::add_task(Task& task){
    pthread_mutex_lock(&m_mutex);
    m_queue.push(task);
    pthread_mutex_unlock(&m_mutex);
}

void Task_queue::add_task(callback f, void* arg){
    pthread_mutex_lock(&m_mutex);
    Task t(f, arg);
    m_queue.push(t);
    pthread_mutex_unlock(&m_mutex);
}


Task& Task_queue::get_task(){
    Task t;
    pthread_mutex_lock(&m_mutex);
    if(m_queue.size() > 0){
        t = m_queue.front();
        m_queue.pop();
    }
    pthread_mutex_unlock(&m_mutex);
```

```
    return t;
}

Thread_pool::Thread_pool(int min, int max){
    //实例化任务队列
    m_task_queue = new Task_queue;
    //do while的好处，但是c++有析构，所以不太用的上
    do
    {
        //初始化线程池
        m_min_num = min;
        m_max_num = max;
        m_busy_num = 0;
        m_live_num = min;

        //根据最大上限给线程数组分配内存
        thread_compose = new pthread_t[m_max_num];
        if(thread_compose == nullptr){
            std::cout<<"create thread array fail~\n";
            break;
        }
        //初始化线程数组
        memset(&thread_compose, 0, sizeof(thread_compose) * m_max_num);

        //初始化互斥锁和条件变量
        if(pthread_mutex_init(&m_mutex_pool, nullptr) != 0){
            throw::std::exception();
        }
        if(pthread_cond_init(&m_not_empty, nullptr) != 0){
            throw::std::exception();
        }

        //创建管理者线程
        pthread_create(&m_manger_thread, nullptr, manager, this);

        //创建工作线程
        for(int i = 0; i < m_min_num; i++){
            //传this指针，才能访问类内成员函数
            pthread_create(&thread_compose[i], nullptr, worker, this);
```

```
        }
    } while (0);

}

Thread_pool::~Thread_pool(){
    shutdown = 1;
    //销毁管理者线程
    pthread_join(m_manger_thread, NULL);
    //唤醒所有消费者线程
    for (int i = 0; i < m_live_num; ++i)
    {   //signal是唤醒某一个，broadcast是唤醒所有，但是还是要抢一把锁，所以都一样
        pthread_cond_signal(&m_not_empty);
    }

    if(m_task_queue){
        delete m_task_queue;
    }
    if(thread_compose){
        delete [] thread_compose;
    }
    pthread_mutex_destroy(&m_mutex_pool);
    pthread_cond_destroy(&m_not_empty);
}

void Thread_pool::add_task(Task task){
    if(shutdown)
    {
        std::cout<<"the thread pool will destory!\n";
        return;
    }
    //添加任务不需要加锁，因为任务队列中有锁
    m_task_queue->add_task(task);
    //添加完任务就要唤醒工作线程取任务
    pthread_cond_signal(&m_not_empty);
}

int Thread_pool::get_busy_num(){
    int busy_num = 0;
```

```
    pthread_mutex_lock(&m_mutex_pool);
    busy_num = m_busy_num;
    pthread_mutex_unlock(&m_mutex_pool);
    return busy_num;
}

int Thread_pool::get_live_num(){
    int live_num = 0;
    pthread_mutex_lock(&m_mutex_pool);
    live_num = m_live_num;
    pthread_mutex_unlock(&m_mutex_pool);
    return live_num;
}

//管理线程任务函数
//主要任务：不断检测工作线程数量，存活线程的数量，然后再决定增加还是删除
void* Thread_pool::manager(void* arg){
    Thread_pool* pool = static_cast<Thread_pool*>(arg);
    while(pool->shutdown){
        //每五秒检测一次
        sleep(5);

        //取出线程数量
        pthread_mutex_lock(&pool->m_mutex_pool);
        int queue_size = pool->m_task_queue->task_size();
        int live_size = pool->m_live_num;
        int busy_size = pool->m_busy_num;
        pthread_mutex_unlock(&pool->m_mutex_pool);

        //创建线程，最多创建两个
        if(queue_size > live_size && live_size <pool->m_max_num){
            pthread_mutex_lock(&pool->m_mutex_pool);
            int count = 0;
            for(int i = 0; i < pool->m_max_num && count < NUMBER
                && pool->m_live_num < pool->m_max_num; i++){
                    //之前memset了
                    if(pool->thread_compose[i] == 0){
                        pthread_create(&pool->thread_compose[i], nullptr, worker, pool);
                        count++;
```

```
                    pool->m_live_num++;
                }
            }
            pthread_mutex_unlock(&pool->m_mutex_pool);
        }

        //销毁多余的线程
        //判断条件：忙线程*2 < 存活的线程 && 存活的线程数 > 最小线程数
        if(2*busy_size < live_size && live_size > pool->m_min_num){
            pthread_mutex_lock(&pool->m_mutex_pool);
            pool->m_destory_num = NUMBER;
            pthread_mutex_unlock(&pool->m_mutex_pool);
            //让工作线程自杀—唤醒但没事儿干就自杀，定义在worker里面
            //没事儿干的线程被阻塞了，阻塞在m_not_empty()条件变量上
            //唤醒后的线程就自己退出了，代码在worker里面
            for (int i = 0; i < NUMBER; ++i)
            {
                pthread_cond_signal(&pool->m_not_empty);
            }
        }
    }
    return nullptr;
}

//工作线程任务函数
void* Thread_pool::worker(void* arg){
    Thread_pool* pool = static_cast<Thread_pool*>(arg);
    //工作线程一直不停工作
    while(true){
        //当线程访问任务队列的时候加锁
        pthread_mutex_lock(&pool->m_mutex_pool);
        //判断工作队列是否为空，为空阻塞线程
        while(pool->m_task_queue->task_size() == 0 && pool->shutdown == 1){
            std::cout<<"thread "<<std::to_string(pthread_self())<<"waiting...\n";
            //将调用线程放入条件变量的等待队列中
            pthread_cond_wait(&pool->m_not_empty, &pool->m_mutex_pool);

            //解除阻塞之后判断是否要销毁线程
            //由于管理线程中满足线程销毁条件了，就通过条件变量唤醒线程
```

```
            //然后由于唤醒的线程是空闲的即任务队列中没东西，如果有东西就不会空闲，
            //不会满足manager线程中销毁线程的条件
            if(pool->m_destory_num > 0){
                pool->m_destory_num--;
                if(pool->m_live_num > pool->m_min_num){
                    pool->m_live_num--;
                    //先解锁再销毁
                    pthread_mutex_unlock(&pool->m_mutex_pool);
                    pool->destory_thread();
                }
            }
        }

        if(pool->shutdown){
            pthread_mutex_unlock(&pool->m_mutex_pool);
            pool->destory_thread();
        }

        //取任务
        Task task = pool->m_task_queue->get_task();
        pool->m_busy_num++;
        pthread_mutex_unlock(&pool->m_mutex_pool);
        //执行任务
        std::cout<<"thread "<<std::to_string(pthread_self())<<" start working.....\n";
        task.fun(task.arg);
        delete task.arg;
        task.arg = nullptr;

        //任务处理结束
        std::cout<<"thread "<<std::to_string(pthread_self())<<" end work.....\n";

        pthread_mutex_lock(&pool->m_mutex_pool);
        pool->m_busy_num--;
        pthread_mutex_unlock(&pool->m_mutex_pool);

    }
    return nullptr;
}
```

```
void Thread_pool::destory_thread(){
    pthread_t tid = pthread_self();
    for(int i = 0; i < m_max_num; i++){
        if(thread_compose[i] == tid){
            std::cout << "threadExit() function: thread "
                << std::to_string(pthread_self()) << " exiting..." << std::endl;
        }
        thread_compose[i] = 0;
        break;
    }
    pthread_exit(nullptr);
}
```

+++

反转链表的递归写法

```
ListNode* reverseList(ListNode* head) {
    if(head == nullptr){
        return head;
    }
    if(head->next == nullptr){
        return head;
    }
    ListNode *temp = reverseList(head->next);
    head->next->next = head;
    head->next = nullptr;
    return temp;
}
```

引用计数实现共享指针

```
template <class T>
class Ref_count{
private:
    T* ptr;          //数据对象指针
    int* count;      //引用计数器指针
public:
    //普通指针构造共享指针
    Ref_count(T* t):ptr(t),count(new int(1)){}

    ~Ref_count(){
        decrease();
    }

    //拷贝构造
    Ref_count(const Ref_count<T>& tmp){
        count = tmp->count;
        ptr = tmp->ptr
        increase();
    }

    //注意=在指针里面是指向的意思，因此说明=左边的共享指针指向了=右边的
    //因此=左边的共享指针-1，=右边的共享指针+1
    Ref_count<T>& operator=(const Ref_count& tmp){
        if(tmp != this){
            decrease();
            ptr = tmp->ptr;
            count = tmp->count;
            increase();
        }
        return *this
    }

    T* operator ->() const{
        return ptr;
    }

    T& operator *() const{
        return *ptr;
    }
```

```
    void increase(){
        if(count){
            *(count)++;
        }
    }

    void decrease(){
        if(count){
            *(count)--;
            if(*count == 0){
                //引用计数为0的时候就删除数据对象指针和引用对象指针
                delete ptr;
                ptr = nullptr;
                delete count;
                count = nullptr;
            }
        }
    }

    T* get() const{
        return ptr;
    }

    int get_count() const{
        if(!count){
            return 0;
        }
        return *count;
    }
};
```

+++

二叉树遍历

递归

```
//前序
void pre_traverse(BinaryTree *root){
    if(root){
        cout<<T->val;
        pre_traverse(root->left);
            pre_traverse(root->right);
    }
}

//中序
void mid_traverse(BinaryTree *root){
    if(root){
        mid_traverse(root->left);
        cout<<T->val;
            mid_traverse(root->right);
    }
}

//后序
void back_traverse(BinaryTree *root){
    if(root){
        mid_traverse(root->left);
            mid_traverse(root->right);
        cout<<T->val;
    }
}
```

非递归

```
//前序
void pre_traverse(BinaryTree *root){
    stack<BinaryTree *> s;
    BinaryTree* cur = root;
    while(cur || !s.empty()){
        while(cur){
                        cout<<cur->Val;
                s.push(cur);
                   cur = cur->left;
         }
        //当cur为空，开始出栈，出栈肯定要判断栈是否为空
        if(!s.empty()){
            cur = s.top();
            s.pop();
            cur = cur->right
        }
    }
}


//中序
void mid_traverse(BinaryTree *root){
    stack<BinaryTree*> s;
    BinaryTree* cur = root;
    while(cur || !s.empty()){
        while(cur){
            cur.push(cur);
            cur = cur->left;
        }
        if(!s.empty()){
            cur = s.top();
            s.pop();
            cout<<cur->val;
            cur = cur->right;
        }
    }
}

//后序
```

```
void back_traverse(BinaryTree *root){
    stack<BinaryTree*> s;
    BinaryTree* cur = root;
    unordered_set<BinaryTree*> res;
    while(cur || !s.empty()){
        while(cur){
            s.push(cur);
            cur = cur->left
        }
        if(!s.empty()){
            cur = s.top();
            s.top();
            //当前节点是第一次入栈,加入到容器中，并重新入栈
            if(res.find(cur) == res.end()){
                res.insert(cur);
                s.push(cur);
                cur = cur->right
            }else{
                cout<<cur->val;
                cur = nullptr;
            }
        }
    }
}

//层序
//基本思想:
//1.先将根节点放到队列中
//2.根节点弹出队列，然后将根节点的左、右儿子入队
//3.弹出左儿子，放入左儿子的左右儿子
//4.弹出右儿子，放入右儿子的左右儿子
//5.重复3、4步
void LevelOrder(BinTree* root){
    queue<BinTree*> m_queue;
    if(!root){
        return;
    }
    BinTree* tmp;
    q.push(root);
```

```
    while(!q.empty()){
            tmp = q.front();
        q.pop();
        cout<<tmp->val<<endl;
        if(tmp->left){
            q.push(tmp->left);
        }
        if(tmp->right){
            q.push(tmp->right);
        }
    }
}
```

写一个string类

```
class String
{
public:
    String(const char *str = NULL); // 普通构造函数
    String(const String &other); // 拷贝构造函数
    String(const String &&other); // 移动构造函数
    ~ String(void); // 析构函数
    String & operator =(const String &other); // 赋值函数
private:
    char *m_data; // 用于保存字符串
    int size;
};

//普通构造函数
String::String(const char *str)
{
     if(str==NULL)
     {
         m_data = new char[1]; // 得分点：对空字符串自动申请存放结束标志'\0'的空
         //加分点：对m_data加NULL 判断
         *m_data = '\0';
     }
     else
     {
         int length = strlen(str);
         m_data = new char[length+1];
         strcpy(m_data, str);
     }
}
// String的析构函数
String::~String(void)
{
         delete [] m_data; // 或delete m_data;
}
//拷贝构造函数
String::String(const String &other)       // 得分点：输入参数为const型
{
     int length = strlen(other.m_data);
     m_data = new char[length+1];
```

```
     strcpy(m_data, other.m_data);
}
String::String(const String &&other)       // 得分点：输入参数为const型
{
     m_data = other.data;
     other.data = nullptr;
}
//赋值函数
String & String::operator =(const String &other) // 得分点：输入参数为const型
{
    if(this == &other)     //得分点：检查自赋值
       return *this;
    if(m_data){
        delete [] m_data;         //得分点：释放原有的内存资源
    }
    int length = strlen( other.m_data );
    m_data = new char[length+1];
    strcpy( m_data, other.m_data );
         return *this;                //得分点：返回本对象的引用
}
//赋值函数
String & String::operator =(const String &other) // 得分点：输入参数为const型
{
    if(this == &other){
        //得分点：检查自赋值
            return *this;
    }
    if(m_data){
        delete [] m_data;         //得分点：释放原有的内存资源
    }
    m_data = other.data;
    other.data = nullptr;
         return *this;                //得分点：返回本对象的引用
}
```

手写一个memcpy

```
void * my_memcpy(void *dst, const void *src, size_t count){
        assert(dst != nullptr);
    assert(src != nullptr);
    char* temp_dst=(char*)dst;
    char* temp_src=(char *)src;
    if(temp_dst>temp_src&&temp_dst<temp_src+count){
        //有内存重叠的情况
               temp_dst=temp_dst+count-1;
        temp_src=temp_src+count-1;
        while(count--){
            *temp_dst--=*temp_src--;
        }
    }else{
        //没有内存重叠的情况
         while(count--){
            *temp_dst++=*temp_src++;
        }
    }
    return (void *)dst;
}
```

注意，不能用是否碰到`\0`判断是否拷贝结束，因为是整块内存，如果用`\0`来判断，可能到中间就停止拷贝了。

提升：一次复制 1 个字节和一次复制 4 个字节占用的 cpu 指令周期是一样的，写一个一次复制 8个字节的？

```
void * nmemcy(void *dst, const void *src, size_t count){
    size_t times8 = count / 8;
    //剩下不够8字节的按照1字节复制
    size_t times1 = count % 8;
    static_cast<int*>(dst);
    static_cast<int*>(src);
    while(times8--){
        *dst = *src;
        dst++;
        src++;
    }
    static_cast<char*>(dst);
    static_cast<char*>(src);
    while(times1--){
        *dst = *src;
        dst++;
        src++;
    }
    return dst;
}
```

在地址按8字节对齐的时候，上述算法的效率比单字节 memcpy 实现高很多，但如果地址没有按8字节对齐，则其效率并不高，有时甚至还比普通 memcpy 还低。这可能是因为，虽然上述算法减少了 cpu 的指令数，但内存的速度比 cpu 慢得多，速度的瓶颈还是在内存。

手写strcpy

```
//把src所指向的字符串复制到dest，注意：dest定义的空间应该比src大。
char* strcpy(char *dest, const char *src) {
     char *ret = dest;
     assert(dest!=NULL);//优化点1：检查输入参数
     assert(src!=NULL);
     while(*src != '\0'){
        *(dest++)=*(src++);
     }
    *dest = '\0';//优化点2：手动地将最后的'\0'补上
     return ret;
}

//考虑内存重叠的字符串拷贝函数优化的情况
char* strcpy(char *dest,char *src) {
     char *ret = dest;
     assert(dest!=NULL);
     assert(src!=NULL);
     memmove(dest,src,strlen(src)+1);
     return ret;
}

//对于以上代码，我们可以看出来，它是存在隐患的
//当源字符串的长度超出目标字符串时，会导致把数据写入到我们无法控制的地址中去，存在很大的风险，所以就有了str
char *strncpy(char* dest, const char* src, size_t n)
{
    assert( (dest != NULL) && (src != NULL));
    char *address = dest;
    while ( n-- && (*src++ != '\0')){
        *dest++ = *src++;
    }
    *dest = '\0';
    return address;
}
```

手写strcat

```
//把 src 所指向的字符串追加到 dest 所指向的字符串的结尾。
char* strcat(char *dest,const char *src) {
    //1. 将目的字符串的起始位置先保存，最后要返回它的头指针
    //2. 先找到dest的结束位置,再把src拷贝到dest中，记得在最后要加上'\0'
    char *ret = dest;
    assert(dest!= nullptr);
    assert(src!= nullptr);
    while(*dest!='\0')
        dest++;
    while(*src!='\0')
        *(dest++)=*(src++);
    *dest='\0';
    return ret;
}
```

手写strcmp

```
//把 str1 所指向的字符串和 str2 所指向的字符串进行比较。
//该函数返回值如下：
//如果返回值 < 0，则表示 str1 小于 str2。
//如果返回值 > 0，则表示 str1 大于 str2。
//如果返回值 = 0，则表示 str1 等于 str2。
//'\0'的ascii码是0
int strcmp(const char* str1, const char*str2){
        assert(str1 != nullptr && str2 != nullptr);
    while (*str1 != '\0' && *str2 != '\0' && *str1 == *str2){
        str1++;
        str2++;
    }
    if(*(unsigned char*)str1 < *(unsigned char*)str2){
        return -1;
    }
    else if (*(unsigned char*)str1 > *(unsigned char*)str2){
        return 1;
    }
    return 0;
    //return *str1 - *str2;
}
```

手写strlen

```
int strlen(const char *str) {
        assert(str != NULL);
        int len = 0;
        while( (*str++) != '\0'){
        len++;
    }
        return len;
}
```

手写strfind

```
//在字符串 str1中查找第一次出现字符串 str2 的位置，不包含终止符 '\0'。
char* strstr(const char *str1, const char *str2) {
     char* tmp_s = str1;
     assert(str1 != nullptr);
     assert(str2!= nullptr);
    //若str2为空，则直接返回空
     if(!str2){
         return nullptr;
     }
    //若不为空，则进行查询
     while(*tmp_s != '\0') {
             char* s1 = tmp_s;
             char* s2 = str2;
             while(*s1 != '\0' && *s2!='\0' && *s1 == *s2){
            s1++;
            s2++;
        }
        //若s2先结束
        if(*s2 == '\0'){
                        return str2;
        }
         //若s1先结束而s2还没结束，则返回空
             if(*s2 != '\0' && *s1== '\0'){
             return nullptr;
        }
            tmp_s++;
     }
     return nullptr;
}
```

排序算法

快排

```
//这个函数主要是求pivot的
int partition(int *a, int left, int right){
    //首先要随机选取一个pivot，这样做是为了根据pivot分成两个子数组数组。一下三种选择方式均可以
    int pivot = a[left];
    int pivot = a[right];
    int pivot = a[left + (right - left) / 2];//这样做是为了防止数组越界
    int pivot_index = left/right/left + (right - left) / 2
    //两个指针移动，当同时指到一个元素时候就退出循环
    //切记，必须从右边开始找！！！！
    while(left < right){
         while(a[right] >= pivot && left < right){
            right--;
        }
        while(a[left] <= pivot && left < right){
            left++;
        }
        if(left < right){
                swap(a[left], a[right]);
        }
    }
    //这一步是为了将选取的pivot值放在中间，保证左边的子数组均比pivot小，右边的子数组均比pivot大
    //这一步也是交换元素,交换选取pivot的索引和left的索引所对应的值
    a[povit_index] = a[left];
    a[left] = povit;
    return left;
    /*优化的写法：：
    int pivot = a[left];
    while(left<right){
        while(left<right&&a[right]<=pivot)right--;
        a[left] = a[right];
        while(left<right&&a[left]>=pivot)left++;
        a[right] = a[left];
    }
    a[left] = pivot;
    return left;
    */
}
void quicksort(int *a, int left, int right){
    //边界条件判断
```

```
    if(left >= right){
        return;
    }
    int pivot = partition();
    quicksort(a, left, pivot - 1);
    quicksort(a, pivot +1, right);
}
```

第二种

```
void Quick_sort(int left,int right,int arr[]){
        if(left>=right)return;
        int i,j,base,temp;
        i=left,j=right;
        base=arr[left];
        while(i<j){
                while(arr[j]>=base && i<j)j--;
                while(arr[i]<=base && i<j)i++;
                if(i<j){
                        temp=arr[i];
                        arr[i]=arr[j];
                        arr[j]=temp;
                }
        }
        arr[left]=arr[i];
        arr[i]=base;
        Quick_sort(left,i-1,arr);
        Quick_sort(i+1,right,arr);
}
```

高低指针不是轮流替换空余位置，而是同时找到不符合的元素，然后交换二者。

最后，高低指针相遇，再把基准元素与相遇位置上的元素交换即可。

堆排序

```
void heap_bulid(vector<int>& vec, int root, int len)
        int left_child = root*2 + 1;
        int righ_child = root*2 + 2;
        int max_root = root;
        if(left_child < len && vec[left_child] > vec[max_root]){
                max_root = left;
        }
        if(right_child < len && vec[right_child] > vec[max_root]){
                max_root = right;
        }
        //如果最大值的节点不是原先的父节点，表示需要
        if(max_root != root){
                swap(vec[root], vec[max_root]);
                heap_bulid(vec, max_root, len);
        }
}
void heap_sort(vector<int>& vec){
        //从右到左sink()方式构造堆，右指的不是最右边，而是从中间向左逼近
        int len = vec.size();
        //从最后一个节点的父节点开始调整
        for(int i = len / 2 - 1; i >=0; i--){
                heap_bulid(vec, i, len);
        }
        //构建完后开始排序
        for(int j = len - 1; j > 0; j--){
                swap(vec[0], vec[j]);
                //交换完后从新建堆,这个堆的长度就要减去被移除(堆顶的那个)的最大元素之后的长度，所以长度不
                heap_build(vec, 0, j);
        }
}
```

归并排序

```
void merge(vector<int>& vec, vector<int>& tmp1, vector<int>& tmp2){
    int len1 = tmp1.size();
    int len2 = tmp2.size();
    int p1 = 0;
    int p2 = 0;
    while(p1 < len1 && p2 < len2){
        if(tmp1[p1] <tmp2[p2]){
            vec.push_back(tmp1[p1++]);
        }else{
            vec.push_back(tmp2[p2++]);
        }
    }
    while(p1 <len1){
        vec.push_back(tmp1[p1++]);
    }
    while(p2 < len2){
        vec.push_back(tmp2[p2++]);
    }
}
void mergesort(vector<int>& vec){
    if(vec.size() <= 1){
        return;
    }
    auto mid = vec.begin() + vec.size()/2;
    vector<int> tmp1(vec.begin(), mid);
    vector<int> tmp2(mid, vec.end());
    mergesort(tmp1);
    mergesort(tmp2);
    vec.clear();
    merge(vec, tmp1, tmp2);
}
```

如何判断本机是大端序还是小端序？

```
int i=0x12345678;
char *p=(char *)&i;
if(*p == 0x78)
    printf("小端模式");
else // (*p == 0x12)
    printf("大端模式");
```

C++ 多线程打印奇偶数

参考链接

参考链接

```c
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>

int count = 0;
pthread_mutex_t m_mutex;
pthread_cond_t cond1;
pthread_cond_t cond2;


void* thread1(void* arg){
    while(1){
        pthread_mutex_lock(&m_mutex);
        //要先阻塞线程， 不能先打印再阻塞，不然就会执行thread1和thread2.
        pthread_cond_wait(&cond1, &m_mutex);
        printf("thread1:%d\n", count);
        pthread_mutex_unlock(&m_mutex);
    }
}


void* thread2(void* arg){
    while(1){
        pthread_mutex_lock(&m_mutex);
        pthread_cond_wait(&cond2, &m_mutex);
        printf("thread2:%d\n", count);
        pthread_mutex_unlock(&m_mutex);
    }
}

void* thread3(void* arg){
    while(1)
    {
        //thread3是管理线程，管理线程必须阻塞一段，不然可以注释看一下问题。
        sleep(1);
        pthread_mutex_lock(&m_mutex);
        count++;
        if(count%2 != 0){
            pthread_cond_signal(&cond1);
```

```
        }else{
            pthread_cond_signal(&cond2);
        }
        pthread_mutex_unlock(&m_mutex);
    }
}

int main(){
    pthread_mutex_init(&m_mutex, NULL);
    pthread_cond_init(&cond1, NULL);
    pthread_cond_init(&cond2, NULL);

    pthread_t p1;
    pthread_t p2;
    pthread_t p3;

    pthread_create(&p1, NULL, thread1, NULL);
    pthread_create(&p2, NULL, thread2, NULL);
    pthread_create(&p3, NULL, thread3, NULL);

    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    pthread_join(p3, NULL);

    pthread_mutex_destroy(&m_mutex);
    pthread_cond_destroy(&cond1);
    pthread_cond_destroy(&cond2);
    return 0;
}
```

手撕LRU算法

- 版本1：自己实现循环链表存储，没有用API

```cpp
/*******************不用API的版本************************/
/*******************简单说一下思路************************/
//1.首先hash表用的是unordered_map来实现，用来查找key对应的node节点，所以hash表应该是[key,no
//2.LRUCache这个类实现双向链表的添加，删除，更新和遍历
//3.同时这个类还要实现get和put两个功能
//4.我这里用的是循环双向链表，因此查找链表尾端的元素为O(1)，正常的双向链表是O(n)
//总结：最重要的就是hash表中的key对应的不是int而是一个node节点，这个要记住
#include<unordered_map>
#include<iostream>
struct Node{
    int key;
    int value;
    Node* pre;
    Node* next;
    Node(){}
    Node(int k, int v):key(k), value(v), pre(nullptr), next(nullptr){}
};

class LRUCache{
private:
    //通过key可以找到位于链表中的节点
    std::unordered_map<int, Node*> hash;
    int capacity;
    Node* head_node;
public:
    LRUCache(int cap){
        capacity = cap;
        head_node = new Node();
        //初始化dummy_Node,next和pre都指向自己
        head_node->next = head_node->pre = head_node;
    }
    //将新来的插入双向链表头部
    void add_Node(Node* n);
    //将某个节点拿出来重新插入头部
    void update_Node(Node* n);
    //移除链表中最后一个（最久未使用）
    void pop_back();
    //输出LRU结构
    void show();
```

```
    int get(int key);
    void put(int key, int value);
};

//注意，该节点可能是新节点，也可能是已经存在的有重新入链表的节点
void LRUCache::add_Node(Node* n){
    //表示当前节点n就是dummy的next节点，不用加入
    if(n->pre == head_node){
        return;
    }
    //将节点n插入head_node后面
    n->pre = head_node;
    n->next = head_node->next;
    head_node->next->pre = n;
    head_node->next = n;
}

void LRUCache::update_Node(Node* n){
    //表示当前节点n就是dummy的next节点，不用断掉
    if(n->pre == head_node){
        return;
    }
    n->next->pre = n->pre;
    n->pre->next = n->next;
    add_Node(n);
}

//弹出链表的最后一个，由于是循环链表，就是head_node->pre
void LRUCache::pop_back(){
    Node* tmp = head_node->pre;
    head_node->pre = tmp->pre;
    tmp->pre->next = head_node;
    //删除unordered_map中的key
    hash.erase(tmp->key);
}

void LRUCache::show(){
    //链表中没有节点，退出
    if(head_node->next = head_node){
```

```
        return;
    }
    Node* tmp = head_node->next;
    while(tmp->next != head_node){
        std::cout<<"key:"<<tmp->key<<",vlaue:"<<tmp->value<<std::endl;
    }
}
int LRUCache::get(int key){
    auto it = hash.find(key);
    if(it == hash.end()){
        std::cout<<"there is no key"<<std::endl;
        return -1;
    }
    //取出key对应的node节点
    Node* node = it->second;
    update_Node(node);
    return node->value;

}
void LRUCache::put(int key, int value){
    auto it = hash.find(key);
    if(it == hash.end()){
        Node* node = new Node(key, value);
        add_Node(node);
        hash.insert({key, node});
        if(hash.size() > capacity){

            pop_back();
        }
    }else{
        it->second->value = value;
        update_Node(it->second);
    }
}
```

- 版本2：使用deque，为什么使用deque说的很清楚

```
/***************注意unordered_map的插入************/

#include <iostream>
#include <deque>
#include <unordered_map>
#include <list>

class LRUCache{
private:
    int capacity;
    //1.之所以用deque不用list是因为移除尾部元素的时候，deque方便
    //2.deque里面可以存储自定的node类型，也可以用pair表示，这里我用pair了
    std::deque<std::pair<int, int>> my_deque;
    //通过key找到对应key在deque中的位置
    std::unordered_map<int, std::deque<std::pair<int, int>>::iterator> hash;
public:
    LRUCache(int cap):capacity(cap){}
    int get(int key);
    void put(int key, int value);
};

int LRUCache::get(int key){
    if(hash.find(key) == hash.end()){
        std::cout<<"there is no key"<<std::endl;
        return -1;
    }
    std::pair<int, int> tmp = *hash[key];
    my_deque.erase(hash[key]);
    my_deque.push_front(tmp);
    //更新hash表中对应key位于deque的位置
    hash[key] = my_deque.begin();
    return tmp.second;
}

void LRUCache::put(int key, int value){
    if(hash.find(key) == hash.end()){
        if(my_deque.size() >= capacity){
            //把hash表中的抹除，然后删除deque中的
            auto it = my_deque.back();
```

```
            hash.erase(it.first);
            my_deque.pop_back();
            my_deque.push_front({key, value});
            hash.insert({key, my_deque.begin()});
        }else{
            my_deque.push_front({key, value});
            hash.insert({key, my_deque.begin()});
        }
    }else{
        //更新就行
        my_deque.erase(hash[key]);
        my_deque.push_front({key, value});
        //更新hash表中key的位置
        hash[key] = my_deque.begin();
    }
}
```

vector数组创建二叉树

```
struct BTree{
    int val;
    BTree* left;
    BTree* right;
    BTree(int v):val(v), left(nullptr), right(nullptr){}
};

/******普通二叉树*******/
BTree* create_tree(vector<int>& node, int index = 0){
    int len = node.size();
    if(node[index] == 0){
        return nullptr;
    }
    BTree* root = new BTree(node[index]);
    int left_index = 2*index+1;
    int right_index = 2*index+2;
    if(left_index > len - 1){
        root->left = nullptr;
    }else{
        root->left = create_tree(node, left_index);
    }
    if(right_index > len - 1){
        root->right = nullptr;
    }else{
        root->right = create_tree(node, right_index);
    }
    return root;
}
```

十进制转十六进制

```
string convert_to_hex(int number) {
    string result;
    int y;
    while(number > 0){
        y = number % 16;
        if(y < 10){
            result = char('0' + y) + result;
        }else{
            result = char('A' - 10 + y) +result;
        }
        number = number / 16;
    }
    return result;
}
```

二分查找

左闭右闭[left, right]

```
int binary_search(vector<int>& nums, int target){
    int left = 0;
    int right = nums.size() - 1;
    while(left <= right){
        int middle = left + (right - left) /2;
        if(nums[middle] > target){
            right = middle -1;
        }
        else if(nums[middle] < target){
            left = middle + 1;
        }else{
            return middle;
        }
    }
    return -1;
}
```

左闭右开[left, right)

```
int search(vector<int>& nums, int target) {
     int left = 0;
     int right = nums.size(); // 定义target在左闭右开的区间里，即：[left, right)
     while (left < right) { // 因为left == right的时候，在[left, right)是无效的空间，所以使用 <
         int middle = left + ((right - left) >> 1);
         if (nums[middle] > target) {
             right = middle; // target 在左区间，在[left, middle)中
         } else if (nums[middle] < target) {
            left = middle + 1; // target 在右区间，在[middle + 1, right)中
         } else { // nums[middle] == target
            return middle; // 数组中找到目标值，直接返回下标
         }
     }
     // 未找到目标值
     return -1;
 }
```

最左侧边界

```
//左闭右开
int left_bound(vector<int>& nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0;
    int right = nums.length(); // 注意

    while (left < right) { // 注意
        int mid = left + (right -left) / 2;
        if (nums[mid] < target) {
            left = mid + 1; //没问题
        } else{
            right = mid;  //这个有点意思，找到target不返回而是继续缩小边界，不断锁定左侧，最终会出现left
        }
    }
    //检查出界情况
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}

//左闭右闭
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    // 搜索区间为 [left, right]
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右侧边界
            right = mid - 1;
        }
    }
    // 检查出界情况
    if (left >= nums.length || nums[left] != target)
```

```
        return -1;
    return left;
}
```

右侧边界

```
//左闭右开
int right_bound(vector<int>& nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0;
    int right = nums.length(); // 注意

    while (left < right) { // 注意
        int mid = left + (right -left) / 2;
        if (nums[mid] < target) {
            left = mid ; //没问题
        } else{
            right = mid - 1;
        }
    }
    //检查出界情况
    if (right >= nums.length || nums[left] != target)
        return -1;
    return right;
}
```

C语言实现多态

```
//虚函数表结构
struct base_vtbl
{
        void(*dance)(void *);
        void(*jump)(void *);
};

//基类
struct base
{
    /*virtual table*/
        struct base_vtbl *vptr;
};
//基类的构造函数
struct base * new_base()
{
    struct base *temp = (struct base *)malloc(sizeof(struct base));
    //基类虚表结构中函数指针具体关联的函数名
        temp->vptr->dance = base_dance;
    temp->vptr->jump = base_jump;
        return temp;
}
//基类的成员函数
void base_dance()
{
        printf("base dance\n");
}
void base_jump()
{
        printf("base jump\n");
}


//派生类
struct derived1
{
        struct base super;
        int high;
};
```

```
//派生类的构造函数
struct derived1 * new_derived1(int h)
{
    struct derived1 * temp= (struct derived1 *)malloc(sizeof(struct derived1));
    //派生类虚表结构中函数指针具体关联的函数名
        temp->super->vptr->dance = derived1_table;
    temp->super->vptr->jump = derived1_jump;
        temp->high = h;
        return temp;
}
//派生类对象成员函数
void derived1_dance()
{
        printf("derived1 dance\n");
}
void derived1_jump(void * this)
{
        struct derived1* temp = (struct derived1 *)this;
        printf("derived1 jump:%d\n", temp->high);
}

/*******实际调用***********/
 struct base * bas = new_base();
//这里调用的是基类的成员函数
bas->vptr->dance();
bas->vptr->jump();

struct derived1 * child = new_derived1(100);
//基类指针指向派生类
bas  = (struct base *)child;

//这里调用的其实是派生类的成员函数
bas->vptr->dance();
bas->vptr->jump((void *)bas);
```

不用sizeof如何获得int所占的字节数？

```
int main(){
    int i = 1;
    int count = 0;
    while(i)
        i = i <<1;//一个循环，每次左移一位
            count++;
    cout << count/8 << endl;//因为一个字节8位
    return 0;
}
```

C++通过递归实现字符串反转

```
#include <iostream>
#include <string>
using namespace std;
string  f(string str ){
        int len=str.length() ;
        if (len<=1)
                return str;
        return f(str.substr( 1)) + str.substr(0,1); //substr(0,1)表示从下标0开始取一个字符形成的串
}
```

```
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <string>
#include <vector>

std::string trim(const std::string& str)
{
    std::string::size_type pos = str.find_first_not_of(' '); //在字符串中查找第一个与str中的字符都不
    if (pos == std::string::npos) {
        return str;
    }

    std::string::size_type pos2 = str.find_last_not_of(' ');
    if (pos2 != std::string::npos) {
        return str.substr(pos, pos2 - pos + 1);
    }

    return str.substr(pos);
}

void split(const std::string& str, std::vector<std::string>* ret_, std::string sep = ",")
{
    if (str.empty()) {
        return;
    }

    std::string tmp;
    std::string::size_type pos_begin = str.find_first_not_of(sep);
    std::string::size_type comma_pos = 0;
    while (pos_begin != std::string::npos) {
        comma_pos = str.find(sep, pos_begin);
        if (comma_pos != std::string::npos) {
            tmp = str.substr(pos_begin, comma_pos - pos_begin);
            pos_begin = comma_pos + sep.length();
        } else {
            tmp = str.substr(pos_begin);
            pos_begin = comma_pos;
        }
```

```
        if (!tmp.empty()) {
            ret_->push_back(tmp);
            tmp.clear();
        }
    }
}

bool DateVerify(int year, int month, int day)
{
    // 这里限制了年份需要在2013-2020,可去掉
    if(year < 2013 || year > 2020 || month < 1 || month > 12 || day < 1 || day > 31) {
        return false;
    }

    switch (month) {
    case 4:
    case 6:
    case 9:
    case 11:
        if (day > 30) {   // 4.6.9.11月天数不能大于30
            return false;
        }
        break;
    case 2:
        {
            bool bLeapYear = (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
            if ((bLeapYear && day > 29) || (!bLeapYear && day > 28)) {
                // 闰年2月不能大于29天;平年2月不能大于28天
                return false;
            }
        }
        break;
    default:
        break;
    }

    return true;
}
```

```
// 校验yyyy/mm/dd
bool CheckDateValid(const std::string& strDate)
{
    std::string strPureDate = trim(strDate);
    if(strPureDate.length() < 8 || strPureDate.length() > 10) {
        return false;
    }

    std::vector<std::string> vecFields;
    split(strPureDate, &vecFields, "/");

    if(vecFields.size() != 3) {
        return false;
    }

    // TODO:这里最好再下判断字符转换是否成功
    int nYear = atoi(vecFields[0].c_str());
    int nMonth = atoi(vecFields[1].c_str());
    int nDay = atoi(vecFields[2].c_str());

    return DateVerify(nYear, nMonth, nDay);
}

// 校验HH:MM:SS
bool CheckTimeValid(const std::string& strTime)
{
    std::string strPureTime = trim(strTime);
    if(strPureTime.length() < 5 || strPureTime.length() > 8) {
        return false;
    }

    std::vector<std::string> vecFields;
    split(strPureTime, &vecFields, ":");

    if(vecFields.size() != 3) {
        return false;
    }
```

```
    int nHour = atoi(vecFields[0].c_str());
    int nMinute = atoi(vecFields[1].c_str());
    int nSecond = atoi(vecFields[2].c_str());

    bool bValid = (nHour >= 0 && nHour <= 23);
    bValid = bValid && (nMinute >=0 && nMinute <= 59);
    bValid = bValid && (nSecond >= 0 && nSecond <= 59);

    return bValid;
}

// 日期格式为: yyyy/mm/dd || yyyy/mm/dd HH:MM:SS
bool CheckDateTimeValid(const std::string& strDateTime)
{
    std::string strPureDateTime = trim(strDateTime);

    std::vector<std::string> vecFields;
    split(strPureDateTime, &vecFields, " ");

    if(vecFields.size() != 1 && vecFields.size() != 2) {
        return false;
    }

    // 仅有日期
    if(vecFields.size() == 1) {
        return CheckDateValid(vecFields[0]);
    }

    return CheckDateValid(vecFields[0]) && CheckTimeValid(vecFields[1]);
}

int main()
{
    assert(CheckDateTimeValid("2013/8/1"));
    assert(CheckDateTimeValid("  2013/8/1  "));
    assert(CheckDateTimeValid("2013/8/01"));
    assert(CheckDateTimeValid("2013/08/1"));
    assert(CheckDateTimeValid("2013/08/01"));
    assert(!CheckDateTimeValid("2013/ / "));
```

```
    assert(!CheckDateTimeValid("2013/   / "));
    assert(!CheckDateTimeValid("2013/  /  "));
    assert(!CheckDateTimeValid("2013/13/01"));
    assert(!CheckDateTimeValid("2013/-1/31"));
    assert(CheckDateTimeValid("2013/01/31"));
    assert(CheckDateTimeValid("2020/02/29"));
    assert(!CheckDateTimeValid("2021/02/29"));

    assert(CheckDateTimeValid("2013/8/1 8:8:8"));
    assert(CheckDateTimeValid(" 2013/8/1      8:8:8  "));
    assert(!CheckDateTimeValid("2013/8/1  18:8"));
    assert(!CheckDateTimeValid("2013/8/1  18"));
    assert(!CheckDateTimeValid("2013/8/1  18:8:60"));
    assert(CheckDateTimeValid("2013/8/1  00:8:00"));
    assert(CheckDateTimeValid("2013/8/1  00:00:00"));
    assert(CheckDateTimeValid("2013/8/1  0:0:0"));
    assert(!CheckDateTimeValid("2013/8/1  24:00:00"));
    assert(CheckDateTimeValid("2013/8/1  23:59:59"));
    assert(CheckDateTimeValid("2013/8/1 23:00:59"));
    assert(!CheckDateTimeValid("2013/8/1 23: 00: 59"));
    assert(CheckDateTimeValid("  2013/8/1  23:59:59"));
    assert(CheckDateTimeValid("  2013/8/1 23:00:59"));
    assert(!CheckDateTimeValid("  2013/8/1  23: 00: 59"));

    assert(!CheckDateTimeValid("2013-8/1 8:8:8"));
    return 0;
}
```