High-frequency interview algorithm questions

1. 合并有序链表

题目描述：

讲两个有序链表合并为一个新的有序链表

```
输入：1->2->4 , 1->3->4
输出：1->1->2->3->4->4
```

算法思路:

1. 将链表1和链表2进行比较，谁小就放在dummy节点后面
2. 若链表1比较完，将链表2后面的所有节点放在主链表后
3. 若链表2比较完，将链表1后面的所有节点放在主链表后

```
struct ListNode{
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* mergeTwoLists(ListNode *list1, ListNode *list2){
    ListNode *dummy =new ListNode(-1), *p = dummy;

    ListNode *p1 = list1, *p2 = list2;
    while(p1 != nullptr && p2 != nullptr){
        if(p1->val < p2->val){
            p->next = p1;
            p1 = p1->next;
        }else{
            p->next = p2;
            p2 = p2->next;
        }
        p = p->next;
    }
    if(p1 == nullptr){
        p->next = p2;
    }
    if(p2 == nullptr){
        p->next = p1;
    }

    return dummy->next;
}
```

2. 反转链表

题目描述：

定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。

```
输入: 1->2->3->4->5->nullptr
输出: 5->4->3->2->1->nullptr
```

算法思路：

定义三个节点，prev, cur, nxt 依次交换nxt， cur， prev 反转后返回prev

```
struct ListNode{
    int val;
    ListNode *next;
    ListNode(int x): val(x), next(nullptr) {}
};
ListNode* revert(ListNode *node){
    ListNode *prev , *cur, *nxt;
    prev = nullptr;
    cur = node;
    nxt = node;

    while(cur){
        nxt = cur->next;
        cur->next = prev;
        prev = nxt;
        nxt = cur;
    }
    return prev;
}
```

3. 单例模式

题目描述：

实现饿汉单例模式和懒汉单例模式

算法思路：

1. 构造函数私有化 -> 防止从类外调用构造函数，保证在任何情况下只生成一个实例。
2. 在类中定义一个静态指针，指向本类的变量的静态变量指针。
3. 提供一个全局的静态方法getInstance（全局访问点）-> 便于提供从类外部获取类的唯一实例方法

饿汉单例模式代码：

```
//饿汉单例模式在类加载的时候就立即初始化，并且创建单例对象，
//不管有没有使用到，先创建好了再说。在线程创建之前就实例化了，所以不会出现线程安全的问题
class Single{
private:
    static Single* p;
    Single (){};

public:
    static Single* getInstance();
};
Single* Single::p = new Single();                    // 饿汉模式类加载的时候就已经创建了对象
Single* Single::getInstance(){return p;}
```

懒汉单例模式代码：

```
//懒汉单列模式，顾名思义，比较“懒”，  用的时候才去检查有没有实例，  如果有则直接返回，没有则创建。
class Single{
private:
    static Single* p;
    Single (){};
public:
    static Single* getInstance(){                          // 懒汉模式在使用的时候才创建该对象
        if(p == nullptr){
            std::lock_guard<std::mutex> lock(m);
            if(p == nullptr){
                return new Single();
            }
        }
        return p;
    };
};
Single* Single::p = nullptr;                    //懒汉模式类加载的时候没有创建该类
```

4. 抽象工厂模式

题目描述：

实现一个抽象工厂模式

算法思路：

1. 实现抽象产品接口类
2. 实现具体产品类
3. 实现抽象工厂接口类

4. 实现具体工厂类
5. 工厂可以生产不同的类别的产品，同时有关联的产品在同一工厂中生产

```
class ProductA{
public:
    virtual void show() = 0;
    virtual ~ProductA(){} ;
};

class BrandAproductA : public ProductA {
public:
    void show() {
        cout<<"BrandAproductA show"<<endl;
    }
    ~BrandAproductA() {
        cout<<"BrandAproductA destroy"<<endl;
    }
};

class BrandBproductA : public ProductA {
public:
    void show(){
        cout<<"BrandBproductA show"<<endl;
    }
    ~BrandBproductA() {
        cout<<"BrandBproductA destroy"<<endl;
    }
};

class BrandCproductA : public ProductA {
public:
    void show() {
        cout<<"BrandCproductA show"<<endl;
    }
    ~BrandCproductA() {
        cout<<"BrandCproductA destroy"<<endl;
    }
};

class ProductB{
public:
    virtual void show() = 0;
    virtual ~ProductB(){};
};

class BrandAproductB: public ProductB{
public:
    void show(){
        cout<<"BrandAproductB show"<<endl;
    }
    ~BrandAproductB() {
        cout<<"BrandAproductB destory"<<endl;
```

```
    }
};

class BrandBproductB: public ProductB{
public:
    void show(){
        cout<< "BrandBproductB show"<<endl;
    }
    ~BrandBproductB(){
        cout <<"BrandBproductB destory"<<endl;
    }
};

//工厂可以创建不同的产品，同时有关联的产品在同一工厂创建
class Factory{
public:
    virtual ProductA* createProductA() = 0;
    virtual ProductB* createProductB() = 0;
};

class BrandAFactory: public Factory{
public:
    ProductA* createProductA(){
        return new BrandAproductA();
    }
    ProductB* createProductB(){
        return new BrandAproductB();
    }
};

class BrandBFactory: public Factory{
public:
    ProductA* createProductA(){
        return new BrandBproductA();
    }
    ProductB *createProductB(){
        return new BrandBproductB();
    }
};

class BrandCFactory: public Factory{
public:
    ProductA* createProductA(){
        return new  BrandCproductA();
    }
    ProductB* createProductB(){
        return nullptr;
    }
};

void ClientCode(){
    std::unique_ptr<Factory> brandAfty = std::make_unique<BrandAFactory>();
    std::unique_ptr<Factory> brandBfty = std::make_unique<BrandBFactory>();
    std::unique_ptr<Factory> brandCfty = std::make_unique<BrandCFactory>();
```

```
    std::unique_ptr<ProductA> brandAPrctA(brandAfty->createProductA());
    std::unique_ptr<ProductA> brandBPrctA(brandBfty->createProductA());
    std::unique_ptr<ProductA> brandCPrctA(brandCfty->createProductA());

    std::unique_ptr<ProductB> brandAPrctB(brandAfty->createProductB());
    std::unique_ptr<ProductB> brandBPrctB(brandBfty->createProductB());

    brandAPrctA->show();
    brandBPrctA->show();
    brandCPrctA->show();

    brandAPrctB->show();
    brandBPrctB->show();

}
```

5. 快速排序

题目描述：

实现快速排序

算法思路：

1. 整体来看快速排序可以看成二叉树的前序遍历
2. 选一个元素作为pivot，然后将所有小于pivot位置的数，放在pivot的左边所有大于pivot位置的数，放在pivot的右边
3. 通过pivot将数组一分为二，递归调用

```
void swap(vector<int> &nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
int partition(vector<int> &nums, int left, int right) {
    int i = left, j = right;
    while(i<j){
        while(i<j && nums[j] >= nums[left]) j--;
        while(i<j && nums[i] <= nums[left]) i++;
        swap(nums, i, j);
    }
    swap(nums, left, i);
    return i;
}
void QuickSort(vector<int> &nums, int lo, int hi){
    if(lo > hi) return;

    int pivot = partition(nums, lo, hi);
```

```
    QuickSort(nums, lo, pivot -1);
    QuickSort(nums, pivot + 1, hi);
}
```

6. 归并排序

题目描述：

实现归并排序

算法思路：

1. 整体来看可以将归并排序看成二叉树的后序遍历
2. 将数组对半划分 ，递归调用归并排序函数将数组划分成只有一个元素的数组
3. 将左边有序的数组和右边有序的数组合并成一个有序的数组

```
void merge(vector<int> &nums, int lo, int mid, int hi){
    vector<int> temp(nums.size());
    for(int i= lo; i<=hi; i++){
        temp[i] = nums[i];
    }
    int i = lo, j = mid + 1;
    for(int p = lo; p<=hi; p++) {
        if(i == mid + 1){
            nums[p] = temp[j++];
        }else if(j == hi + 1){
            nums[p] = temp[i++];
        }else if(temp[i] < temp[j]){
            nums[p] = temp[i++];
        }else{
            nums[p] = temp[j++];
        }
    }
}
void MergeSort(vector<int> &nums, int left, int right){
    if(left == right) return;

    int mid = (left + right) / 2;
    MergeSort(nums, left, mid );
    MergeSort(nums, mid + 1, right);
    merge(nums, left, mid, right);
}
```

7. LRU缓存

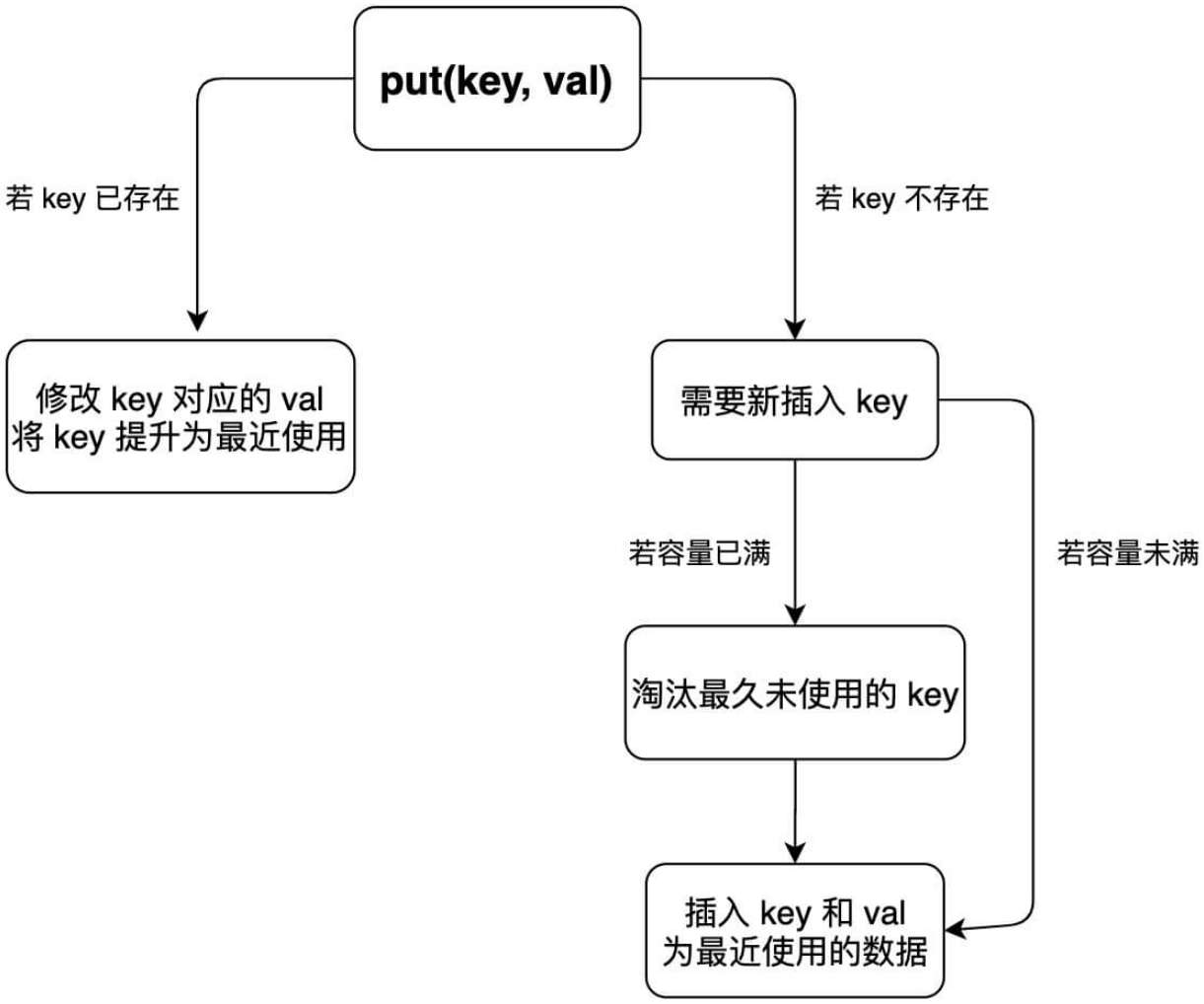
题目描述：

请你设计并实现一个满足 LRU (最近最少使用) 缓存 约束的数据结构。 实现 LRUCache 类：

1. LRUCache(int capacity) 以 正整数 作为容量 capacity 初始化 LRU 缓存
2. int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
3. void put(int key, int value) 如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则向缓存中插入该组 key-value 。如果插入操作导致关键字数量超过 capacity ，则应该 逐出 最久未使用的关键字。

算法思路：

1. 使用双向链表list和和hash表unordered_map实现
2. put的流程




```
class LRU{
private:
    list<pair<int,int>> cache;
    unordered_map<int, list<pair<int,int>>::iterator> map;
    int cap;

public:
    LRU(int capacity): cap(capacity){}
    int get(int key){
        if(map.find(key) == map.end()) return -1;        // 若没有key值，则返回-1；
        auto key_value = *map[key];                      //若有key则把cacche中的key
```

```
移到最前面
        cache.erase(map[key]);
        cache.push_front(key_value);
        map[key] = cache.begin();                          //同时改变map中key的引用
        return key_value.second;
    }

    void put(int key, int value){
        if(map.find(key) != map.end()){
            auto key_value = *map[key];                    //如果存在key，则将cache中
key对应的value改掉，然后移到最前面
            key_value.second = value;
            cache.erase(map[key]);
            cache.push_front(key_value);
            map[key] = cache.begin();                      //同时修改map中key的引用
            return;
        }
        if(cache.size() == cap){                           //如果容量满，则删除最久未使
用的
            auto key_ = cache.back().first;
            cache.pop_back();
            map.erase(key_);
        }
        cache.push_front({key, value});                    //添加新的key 和 value
        map[key] = cache.begin();
    }
};
```

8. 重排链表

题目描述：

给定一个单链表 L：L0→L1→…→Ln-1→Ln， 将其重新排列后变为： L0→Ln→L1→Ln-1→L2→Ln-2→…

```
示例 1:
输入链表 1->2->3->4     输出排列 1->4->2->3.
示例 2:
输入链表 1->2->3->4->5, 输出排列 1->5->2->4->3.
```

算法思路：

1. 将链表对半分开
2. 将第二个链表反转
3. 然后将两个链表依次插入

```
struct ListNode{
    int val;
```

```
    ListNode *next;
    ListNode(int x):val(x),next(nullptr) {}
};

ListNode* revert(ListNode *list){
    ListNode *prev, *cur, *nxt;
    prev = nullptr;
    cur = list;
    nxt = list;
    while(cur){
        nxt = cur->next;
        cur->next = prev;
        prev = cur;
        cur = nxt;
    }
    return prev;
}

void reorderedList(ListNode *list){
    ListNode *slow = list, *fast = list;
    while(fast->next && fast->next->next){
        slow = slow->next;
        fast = fast->next->next;
    }
    ListNode *p2 = slow->next, *p1 = list;
    ListNode *dummy = new ListNode(-1), *p = dummy;
    slow->next = nullptr;                                    //断链
    p2 = revert(p2);
    while(p1){
        p->next = p1;
        p1 = p1->next;
        p = p->next;
        p->next = p2;
        if(p2){
            p2 = p2->next;
            p = p->next;
        }
    }
    list = dummy->next;
};
```

9. 奇偶链表

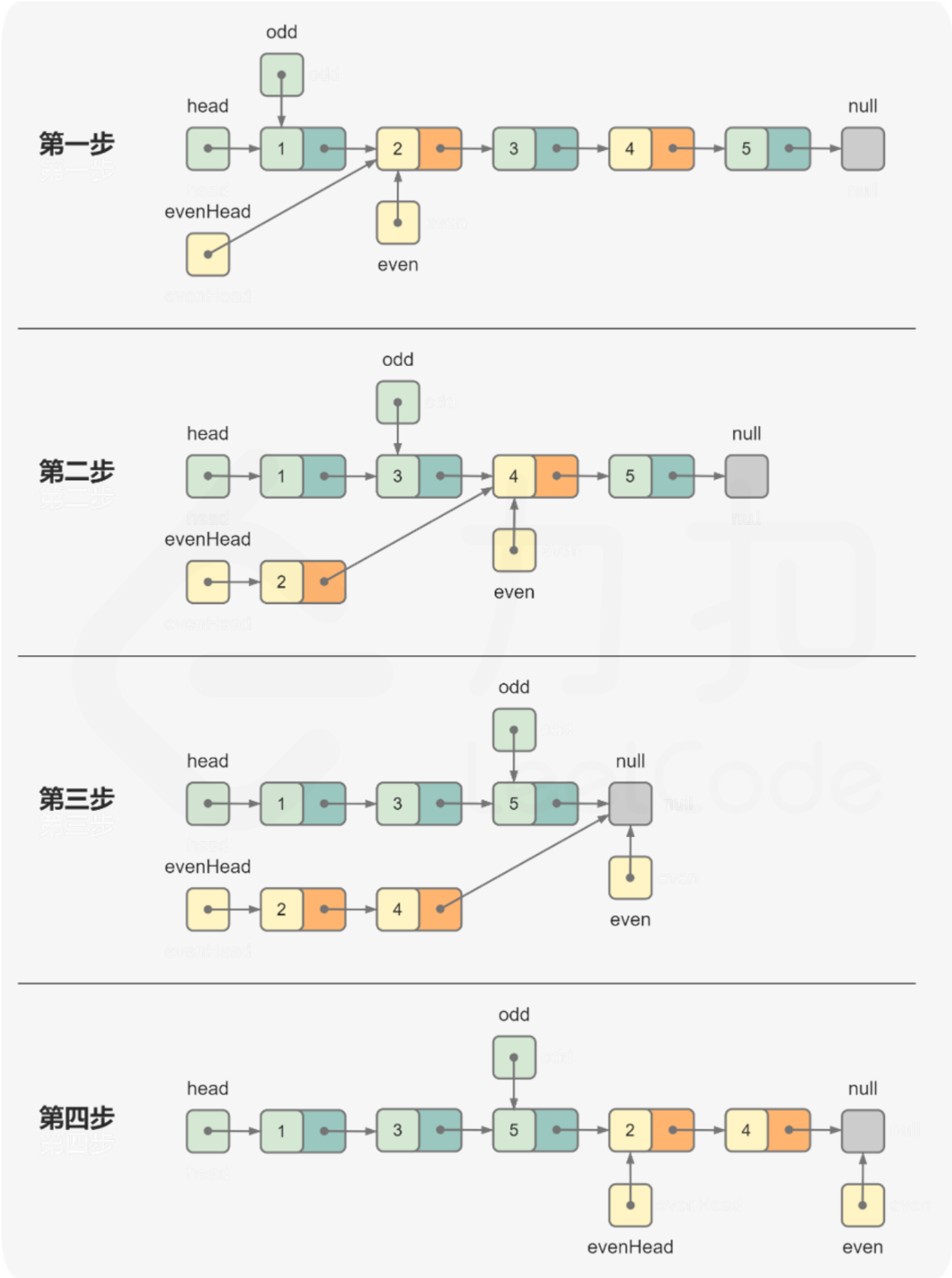
题目描述：

给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

```
示例 1:
输入: 1->2->3->4->5->NULL
输出: 1->3->5->2->4->NULL
```

```
示例 2:
输入: 2->1->3->5->6->4->7->NULL
输出: 2->3->6->7->1->5->4->NULL
```

算法思路：




```
struct ListNode{
    int val;
    ListNode *next;
```

```
    ListNode(int x):val(x),next(nullptr) {}
};

ListNode *oddEvenList(ListNode *head){
    if(head == nullptr) return nullptr;

    ListNode *evenHead = head->next;
    ListNode *odd = head;
    ListNode *even = evenHead;

    while(even && even->next){
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }

    odd->next = evenHead;
    return head;
}
```

10. 多线程轮流依次打印

题目描述：

1. 依次轮流打印abc
2. 依次打印奇数偶数

算法思路：

使用锁，条件变量

轮流打印abc代码

```
mutex m;
int max_count= 3;
condition_variable cv;
int flag = 0;
int number = 0;

void printA(){
    while(number < max_count){
        unique_lock<mutex> lock(m);
        cv.wait(lock, []{return flag==0;});             //如果条件为true就往下运行，否者线程等待
        cout << "thread1: a"<< endl;
        number++;
        flag = 1;
        cv.notify_all();
```

```
    }
}
void printB(){
    while(number < max_count){
        unique_lock<mutex> lock(m);
        cv.wait(lock, []{return flag==1;});
        cout<< "thread2: b"<< endl;
        number++;
        flag = 2;
        cv.notify_all();
    }
}
void printC(){
    while(number < max_count){
        unique_lock<mutex> lock(m);
        cv.wait(lock, []{return flag==2;});
        cout<< "thread3: c"<<endl;
        number++;
        flag = 0;
        cv.notify_all();
    }
}
```

轮流打印奇数偶数代码

```
mutex m;
int max_count = 10;
condition_variable cv;
int number = 0;

void printOdd(){
    while(number < max_count){
        unique_lock<mutex> lock(m);
        cv.wait(lock, []{return number%2==1;});     //如果条件为真则往下执行
        cout << std::this_thread::get_id() << ": "<< number <<endl;
        number++;
        cv.notify_one();
    }
}

void printEven(){
    while(number < max_count){
        unique_lock<mutex> lock(m);
        cv.wait(lock, []{return number%2==0;});   //如果条件为真则继续往下执行
        cout << std::this_thread::get_id() << ": "<<number <<endl;
        number++;
        cv.notify_one();
    }
}
```