原地交换

> 异或运算：
>
> a ^ 0 = a
>
> a ^ a = 0
>
> a ^ 全1 等价于~a

```
// 注意 这个技巧基本没啥用
// 参考：https://blog.csdn.net/solstice/article/details/5166912
void inplace_swap(int *x, int *y) {
    *y = *x ^ *y;
    *x = *x ^ *y;
    *y = *x ^ *y;
}
```

管程

管程模型（Hasen模型、Hoare模型、MESA模型，以及Java中的实现）相关参考文章

- https://www.cnblogs.com/upnote/p/13030741.html
- https://www.cnblogs.com/xidongyu/p/10891303.html

管程相关知识点

- 一个管程是一个由过程、变量及数据结构等组成的一个集合，它们组成一个特殊的模块或软件包（`类`）
- 进程可以在任何需要的时候调用管程中的过程，但不能在管程之外声明的过程中直接访问管程内的数据
- `管程是语言概念`，C语言不支持管程，Java支持（但与经典定义有区别，∵无内嵌条件变量）

  编译器一般会知道管程的特殊性，因此可以采用与其他过程调用不同的方法来处理对管程的调用

  进入管程时的互斥由编译器负责，通常做法是用一个互斥信号量
- `任一时刻管程中只能有一个活跃进程`，这一特性使得管程能够有效地完成互斥
- 条件变量
  - 额外同步机制由`Condition变量`提供，对Condition变量的操作仅有`wait`、`signal`
  - `条件变量不是计数器，也不能像信号量那样积累信号以便以后使用`
  - Condition x
    - x.wait()：把调用该操作的进程挂起
    - x.signal()：重启一个悬挂的进程，若无进程悬挂则没有作用

      注意与信号量的区别：信号量signal会影响信号量值的状态，而条件变量signal不会
  - Java原生语言的实现没有内嵌的条件变量，但提供了wait和notify方法

Java的并发库（j.u.c）提供了`Condition类`，使用`await()`和`signalAll`代替`wait`和`notify`

管程实现生产者消费者问题的代码

- 体会一下使用管程实现生产者消费者问题和常规实现的区别：Producer类和Consumer类不用直接在资源上进行同步操作了，取而代之的是通过调用Monitor管程类来实现
- 下面源代码是使用常规的`wait`和`notify`方法实现的管程

  `/* */`这种注释包裹的代码是使用`Condition类`作为条件变量实现的管程
- 体会一下管程相关的知识点是如何在代码中体现的
  - 【管程是xxxx的封装】：管程本身是个类
  - 【任一时刻管程中只能有一个活跃进程】：`synchronized关键字`保证每次只有一个线程进入管程
  - 【条件变量】：Condition类的使用（原生的wait notify方法不像直接使用Condition类那样体现的很明显）

```
public class ProducerConsumer {
    static final int N = 100;            // 缓冲区大小
    static Producer p = new Producer();  // 生产者线程
    static Consumer c = new Consumer();  // 消费者线程
    static Monitor mon = new Monitor();  // 管程
    /*
        以下为使用Condition类的代码
        static Lock lock = new ReentrantLock();
        static Condition condition = lock.newCondition();
     */
    public static void main(String[] args) {
        p.start();
        c.start();
    }

    /**
     * 消费者
     */
    static class Producer extends Thread {
        @Override
        public void run() {
            int item;
            while (true) {
                item = produceItem();
                mon.insert(item);
            }
        }

        private int produceItem() {
            // 此处写实际生产的代码
            return 0;
        }
    }

    /**
     * 生产者
```

```
37          */
38         static class Consumer extends Thread {
39             @Override
40             public void run() {
41                 int item;
42                 while (true) {
43                     item = mon.remove();
44                     consumeItem(item);
45                 }
46             }
47 
48             private void consumeItem(int item) {
49                 // 此处写实际消费的代码
50             }
51         }
52 
53         /**
54          * 管程类
55          */
56         static class Monitor {
57             private int[] buffer = new int[N];
58             private int count = 0, lo = 0, hi = 0;
59 
60             public synchronized void insert(int val) {
61                 // 缓冲区满则休眠
62                 if (count == N) {
63                     goToSleep();
64                 }
65                 buffer[hi] = val;
66                 hi = (hi+1) % N;
67                 count++;
68                 // 如果消费者在休眠则将其唤醒
69                 if (count == 1) {
70                     /* condition.signal(); */
71                     notify();
72                 }
73             }
74 
75             public synchronized int remove() {
76                 int val;
77                 // 缓冲区为空则睡眠
78                 if (count == 0) {
79                     goToSleep();
80                 }
81                 val = buffer[lo];
82                 lo = (lo+1) % N;
83                 count--;
84                 // 如果生产者在休眠则将其唤醒
85                 if (count == N-1) {
86                     /* condition.signal(); */
87                     notify();
88                 }
89                 return  val;
90             }
91 
92             private void goToSleep() {
93                 try {
94                     /* condition.await(); */
```

```
                    wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

```

HTTP和HTTPS相关

参考文章： https://www.cnblogs.com/wqhwe/p/5407468.html

SSL

- 为网络通信提供安全以及数据完整性的一种安全协议
- 是操作系统对外提供的API，SSL3.0后更名为TLS
- 采用身份验证和数据加密保证网络通信的安全和数据的完整性

加密的方式

- 对称加密：加密和解密都使用同一个密钥，e.g：DES
- 非对称加密：加密使用的密钥和解密使用的密钥是不同的，性能低但安全性极强，e.g：RSA
- 哈希算法：将任意长度的信息转换为固定长度的值，算法不可逆， e.g：SHA
- 数字签名：证明某个消息或者文件是某人发出/认同的

HTTPS传输流程

- 浏览器将支持的加密算法信息发送给服务器
- 服务器选择一套浏览器支持的加密算法，以 `证书` 的形式回发浏览器
- 浏览器验证证书合法性，并结合证书公钥加密信息发送给服务器
- 服务器使用私钥解密信息，验证哈希，加密响应消息回发浏览器
- 浏览器解密响应消息，并对消息进行验证，之后进行加密交互数据

HTTP和HTTPS的区别

- HTTPS需要用到CA申请证书（CA是指Certification Authority，CA机构就是发证书的机构，一般都收钱的），HTTP不需要
- HTTPS密文传输，HTTP明文传输
- HTTP是无状态的

  HTTPS=HTTP+加密+认证+完整性保护，比HTTPS更安全
- HTTP端口80，HTTPS端口443

HTTPS真的安全吗?

- 浏览器默认填充http://，然后再重定向成https://，但这个过程的请求容易被劫持

滑动窗口大小的总结

参考文章—下面有个70多赞的回答： https://zhidao.baidu.com/question/432120567493536524.html

快速排序相关

85.为什么快排平均性能好?

278.为什么快排最后要退化成n^2还叫快排?

A：与其他时间复杂度为O(nlogn)的排序算法相比，快排系数更小，仅为1.39，且可以通过排序前打乱数组、随机选取枢轴元素等方法提前预防出现n^2复杂度的情况

110.快排对于初始有序的操作怎么提高 效率，就是如何优化

315.快排优化

1. 保持随机性，有两种方法，一是在排序前打乱数组，二是每次递归时随机选取切分元素
2. 当数组规模较小时，切换为插入排序，因为插入排序在数组基本有序的情况下速度非常快，交换次数仅为数组中逆序对的个数，而快排在规模较小时仍会有递归开销

   具体实践的话，Java中的Arrays.sort中有一个`INSERTION_SORT_THRESHOLD=47`，当数组规模小于47时便会切换为插入排序
3. 三取样切分，选取子数组小部分元素的中位数来切分数组，减少采用左右端点碰到极端顺序的最坏情况（就是保住选的不是最小的，也不是最大的）

   举个例子，如果数组是(顺序/逆序)有序的，切分选择子数组第一个元素，那么会产生$n^2/2$次比较，这是最坏的情况，三取样切分就是保住不会出现太多这种极端情况
4. 如果重复元素较多的话，可以考虑三向切分，将数组切分为3部分，分别对应<、=、>切分元素的数组元素，但重复元素少的情况下这种方法会比标准方法产生更多次交换