异步线程、同步并发：

异步线程：就是普通的多线程同时运行，没有锁约束。

同步并发：并发指是一个进行，再进行另一个。及通过synchronized或lock达到一个接一个的效果。

Lock

传统的`synchronized`代码：

```
public class Counter {
    private int count;

    public void add(int n) {
        synchronized(this) {
            count += n;
        }
    }
}
```

如果用`ReentrantLock`替代，可以把代码改造为：

```
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int count;

    public void add(int n) {
        lock.lock();
        try {
            count += n;
        } finally {
            lock.unlock();
        }
    }
}
```

因为`synchronized`是Java语言层面提供的语法，所以我们不需要考虑异常，而`ReentrantLock`是Java代码实现的锁，我们就必须先获取锁，然后在`finally`中正确释放锁。

顾名思义，`ReentrantLock`是可重入锁，它和`synchronized`一样，一个线程可以多次获取同一个锁。

- `ReentrantLock`可以替代`synchronized`进行同步；
- `ReentrantLock`获取锁更安全；
- 必须先获取到锁，再进入`try {...}`代码块，最后使用`finally`保证释放锁。

Deadlock

相互等待的两个以上的锁。

```
public class DeadLock {
    public static void main(String[] args) {
        Object obj1 = new Object();
```

```
        Object obj2 = new Object();
        Thread t1 = new Thread(new Dead(obj1, obj2));
        Thread t2 = new Thread(new Dead(obj2, obj1));
        t1.start();
        t2.start();
    }
}

class Dead implements Runnable {
    Object obj1;
    Object obj2;

    public Dead(Object obj1, Object obj2) {
        super();
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    public void run() {
        synchronized (obj1) {
            System.out.println(Thread.currentThread().getName() + obj1);
            synchronized (obj2) {
                System.out.println(Thread.currentThread().getName() + obj2);
            }
        }
    }
}
```

当t1开始时，obj1作为锁被占有；

然后t2开始，obj2作为锁被占有；

此时t1中需要等待obj2被释放，而t2则等待obj1被释放，两者相互等待，构成了死锁。

例子

异步线程-同时打印日志

```
public static void main(String[] args) {

    System.out.println("begin:" + (System.currentTimeMillis() / 1000));
    final BlockingQueue<String> queue = new ArrayBlockingQueue<>(16);

    // 循环16次，打印16个目标对象
    for (int i = 0; i < 16; i++) {
        final String log = "" + (i + 1);
        try {
            queue.put(log);//将日志信息放入队列
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // 启动4个线程
    for (int i = 0; i < 4; i++) {//同时开启四个线程
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                // 无限从堵塞队列中取出数据并打印
                while (true) {
                    String log;
                    try {
                        log = queue.take();
                        parseLog(log);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    if(queue.isEmpty())
                        break;
                }
            }
        }).start();
    }
}

private static void parseLog(String log) {
    System.out.println(log + ":" + (System.currentTimeMillis() / 1000));
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

生产者-消费者模型

```
class Goods {}

class Producer implements Runnable {
    private Goods goods;

    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (TestPC.queue) {
                goods = new Goods();
                if (TestPC.queue.size() < 10) {
                    TestPC.queue.add(goods);
                    System.out.println(Thread.currentThread().getName() + "生产商
品");
                } else {
                    try {
                        TestPC.queue.wait();//wait在锁上调用
                    } catch (InterruptedException e) {
                        e.printStackTrace();
```

```
                    }
                }
            }
        }
    }
}

class Consumer implements Runnable {
    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (TestPC.queue) {
                if (!TestPC.queue.isEmpty()) {
                    TestPC.queue.poll();
                    System.out.println(Thread.currentThread().getName() + "消费商
品");
                } else {
                    TestPC.queue.notifyAll();//notify
                }
            }
        }
    }
}

public class TestPC {
    public static final int MAX_POOL = 10;
    public static final int MAX_PRODUCER = 5;
    public static final int MAX_CONSUMER = 4;
    public static Queue<Goods> queue = new ArrayBlockingQueue<>(MAX_POOL);

    public static void main(String[] args) {
        Producer producer = new Producer();
        Consumer consumer = new Consumer();
        for (int i = 0; i < MAX_PRODUCER; i++) {
            Thread threadA = new Thread(producer, "生产者线程" + i);
            threadA.start();
        }
        for (int j = 0; j < MAX_CONSUMER; j++) {
            Thread threadB = new Thread(consumer, "消费者线程" + j);
            threadB.start();
        }
    }
}
```

producer生成产品，如果产品挤满了仓库则wait；consumer消费产品，如果产品没有库存了，则唤醒所有生产者开始生产。注意wait和notify都是对锁对象上进行的。

TaskQueue

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
```

```
        TaskQueue q = new TaskQueue();
        ArrayList<Thread> ts = new ArrayList<Thread>();
        for (int i = 0; i < 5; i++) {
            Thread t = new Thread() {
                public void run() {
                    // 执行task:
                    while (true) { //不断执行getTask任务
                        try {
                            String s = q.getTask();
                            System.out.println("execute task: " + s);
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                }
            };
            t.start();//开启多个一直getTask的线程
            ts.add(t);
        }

        Thread add1 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {//一次放入10个task的线程
                // 放入task:
                String s = "t-" + Math.random();
                System.out.println("add task: " + s);
                q.addTask(s);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            }
        });

        Thread add2 = new Thread(() -> {
            for (int i = 0; i < 10; i++) {//一次放入10个task的线程
                // 放入task:
                String s = "t-" + Math.random();
                System.out.println("add task: " + s);
                q.addTask(s);
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                }
            }
        });

        add1.start();//开始执行放入task的线程
        add2.start();//开始执行放入task的线程
        add1.join();//等待放入完之后再执行main线程
        add2.join();//等待放入完之后再执行main线程
        Thread.sleep(100);//让getTask的线程跑一会儿

        for (Thread t : ts) {//停止所有getTask线程
            t.interrupt();
        }
    }
}
```

```
class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
        this.notifyAll();
    }

    public synchronized String getTask() throws InterruptedException {
        while (queue.isEmpty()) {
            this.wait();
        }
        return queue.remove();
    }
}
```

两个add线程，通过synchronized保证不会同时加入task；

10个get线程，通过synchronized保证不会同时get同一task；

且add线程和get线程之间有顺序关系，必须有task才get，空的时候add；

所以通过wait和notify达到效果。

输出有时两个add之后两个execute，有时一个add一个execute，看谁更能抢。